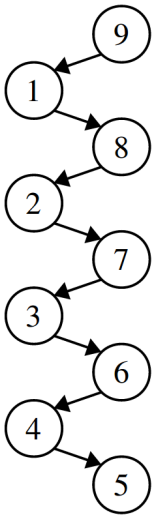
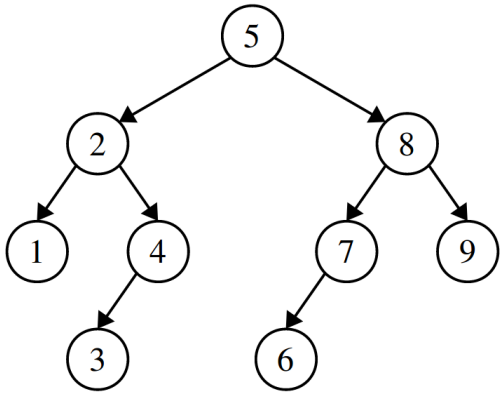
Section Solutions 7

Problem One: Binary Search Tree Warmup!

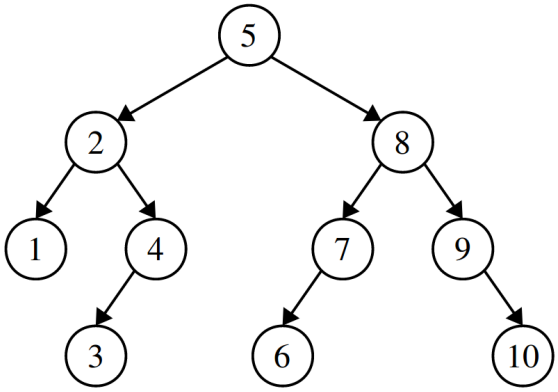


There are several trees that are tied for the tallest possible binary search tree we can make from these numbers, one of which is shown to the right. It has height eight, since the height measures the number of *links* in the path from the root to the deepest leaf. We can see that this is the greatest height possible because there's exactly one node at each level, and the height can only increase by adding in more levels. A fun math question to ponder over: how many different binary search trees made from these numbers have this height? And what's the probability that if you choose a random order of the elements 1 through 9 to insert into a binary search tree that you come up with an ordering like this one?

Similarly, there are several trees tied for the shortest possible binary search tree we can make from these numbers, one of which is shown below. It has height three, which is the smallest possible height we can have. One way to see this is to notice that each layer in the tree is, in a sense, as full as it can possibly be; there's no room to move any of the elements from the deeper layers of the tree any higher up:



If we insert 10, we'd get the following:



Problem Two: Walking Through the Trees

Here's an iterative function to search a tree for a value. It works by manually adjusting a pointer to a tree node to move to the left or right as appropriate. This version is written for the string BST we did in class, but it can easily be adjusted to work with nodes of any type.

```
bool contains(Node* root, const string& key) {
    while (root != nullptr) {
        if (key == root->value) return true;
        else if (key < root->value)  root = root->left;
        else /* key > root->value */ root = root->right;
    }
    return false;
}
```

Inserting into a BST iteratively is a bit trickier. The reason for this is that once we've walked off the tree and found the insertion point, we have to remember where we were most recently in the tree so that we can update that node to point to the newly-inserted value. This is most easily accomplished by keeping track of two pointers, a current pointer and a previous pointer. Here's some code for this:

```
void insert(Node*& root, const string& key) {
    Node* curr = root;
    Node* prev = nullptr;

    /* Walk the tree to find the insertion point. This is essentially the same
     * code for a lookup.
     *
     * Question to ponder: why did we make a new pointer curr here rather than
     * using root?
     */
    while (curr != nullptr) {
        prev = curr;

        if (key == curr->value) return; // Already present!
        else if (key < curr->value)  curr = curr->left;
        else /* key > curr->value */ curr = curr->right;
    }

    /* At this point we've walked off the tree. Let's build up a new node, then
     * wire it into the tree.
     */
    curr = new Node;
    curr->value = key;
    curr->left = curr->right = nullptr;

    /* There are two cases to consider here. First, we might have inserted into
     * an empty tree, in which case prev will be null.
     */
    if (prev == nullptr) {
        root = curr;
    }
    /* Otherwise, the new value should hang off the tree. We need to see how. */
    else {
        if (key < prev->key) prev->left = curr;
        else /* key > prev->key */ prev->right = curr;
    }
}
```

Problem Three: The Ultimate and Penultimate Values

We could solve this problem by writing a function that searches over the entire BST looking for the biggest value, but we can do a lot better than this! It turns out that the biggest value in a BST is always the one that you get to by starting at the root and walking to the right until it's impossible to go any further. Here's a recursive solution that shows off why this works:

```
Node* biggestNodeIn(Node* root) {
    if (root == nullptr) error("Nothing to see here, folks.");

    /* Base case: If the root of the tree has no right child, then the root node
     * holds the largest value because everything else is smaller than it.
     */
    if (root->right == nullptr) return root;

    /* Otherwise, the largest value in the tree is bigger than the root, so it's
     * in the right subtree.
     */
    return biggestNodeIn(root->right);
}
```

And, of course, we should do this iteratively as well, just for funzies:

```
Node* biggestNodeIn(Node* root) {
    if (root == nullptr) error("Nothing to see here, folks.");

    while (root->right != nullptr) root = root->right;

    return root;
}
```

Getting the second-largest node is a bit trickier simply because there's more places it can be. The good news is that it's definitely going to be near the rightmost node – we just need to figure out exactly where.

There are two cases here. First, imagine that the rightmost node does not have a left child. In that case, the second-smallest value must be that node's parent. Why? Well, its parent has a smaller value, and there are no values between the node and its parent in the tree (do you see why?) That means that the parent holds the second-smallest value. The other option is that the rightmost node *does* have a left child. The largest value in that subtree is then the second-largest value in the tree, since that's the largest value smaller than the max. We can use this to write a nice iterative function for this problem that works by walking down the right spine of the tree (that's the fancy term for the nodes you get by starting at the root and just walking right), tracking the current node and its parent node. Once we get to the largest node, we either go into its left subtree and take the largest value, or we return the parent, whichever is appropriate.

```
Node* secondBiggestNodeIn(Node* root) {
    if (root == nullptr) error("Nothing to see here, folks.");

    Node* prev = nullptr;
    Node* curr = root;
    while (curr->right != nullptr) {
        prev = curr;
        curr = curr->right;
    }
    return curr->left == nullptr? prev : biggestNodeIn(curr->left);
}
```

Notice that all three of these functions work by walking down the tree, doing a constant amount of work at each node. This means that the runtime is O(h), where h is the height of the tree. In a balanced tree that's O(log n) work, and in an imbalanced tree that's O(n) work in the worst-case.

Problem Four: A Problem of Great Depth and Complexity

There are many ways to write this function. The easiest one that I know of is to use a nice recursive observation: the height of the empty tree is -1, and the height of a nonempty tree is always one plus the height of the larger of the heights of its two subtrees (do you see why?). To check this, think about what that says about the height of a tree with a single node, the height of a highly degenerate tree, etc. Here's what this looks like:

```
int heightOf(Node* root) {
    if (root == nullptr) return -1;

    return 1 + max(heightOf(root->left), heightOf(root->right));
}
```

So how efficient is this code? Well, notice that it visits every node in the tree once and exactly once, doing O(1) work at each node. There are O(n) total nodes in the tree, so this does a total of O(n) work.

Problem Five: Order Statistic Trees

The key insight you need to have to solve this problem is the following. Suppose you're looking for the kth-smallest node in the tree (zero-indexed) and the root node has k nodes in its left subtree. In that case, you know that the root node is the one you're looking for: it has k nodes smaller than it, so it's the kth-smallest value. On the other hand, suppose that you're looking for the kth-smallest node and the root node has more than k nodes in its left subtree. Then you should go look in the left subree for the kth-smallest node, since you know it must be one of them. Finally, suppose you're looking for the kth-smallest node and the root node has l nodes in its left subtree, with $l < k$. That means that the node you're looking for isn't in the left subtree, and it isn't the root node, so it's got to be in the right subtree. Specifically, it's going to be the $(k - l - 1)^{\text{st}}$-smallest value in that subtree, since you've skipped over $l + 1$ elements in the course of going there.

The resulting code is surprisingly short. Here's a recursive implementation:

```
Node* kthNodeIn(Node* root, int k) {
    /* Base case: If we walked off the tree, or if we're looking for an invalid
     * index, we've failed.
     */
    if (root == nullptr || k < 0) return nullptr;

    if (k < root->leftSubtreeSize) {
        return kthNodeIn(root->left, k);
    } else if (k == root->leftSubtreeSize) {
        return root;
    } else /* (k < root->leftSubtreeSize) */ {
        return kthNodeIn(root->right, k - 1 - root->leftSubtreeSize);
    }
}
```

This runs in time O(h), since we're descending from the root downward and doing O(1) work per step.

Order statistics trees are a special type of BST called an ***augmented binary search tree***. Augmented BSTs have all sorts of nifty properties and you can use them to solve a bunch of problems much faster than initially seems possible. Take CS161 or CS166 for details!

Problem Six: Custom Comparators

Internally, the `Map` and `Set` use binary search trees to organize their keys/elements, so they need to be able to compare two keys/elements against one another in order to do insertions, deletions, lookups, or really just about everything. That's why you can't just put a `struct` into a `Map` or `Set` directly: the internal implementation of `Map` and `Set` try to use the less-than operator to compare the `structs`, which fails.

The four rules that the less-than operator needs to obey are the following:

- ***Consistency:*** Making the same comparison multiple times should always return the same result.
- ***Irreflexivity:*** No object ever compares less than itself. Mathematically, $x < x$ should never be true.
- ***Transitivity:*** Comparisons should be consistent. That is, if $x < y$ and $y < z$, then $x < z$.
- ***Transitivity of Incomparability:*** If x and y are incomparable, then they behave identically to one another.

One way that we saw that works pretty well for implementing these sorts of comparisons is to use a ***lexicographical comparison*** that goes one field at a time seeing how they compare. Here's one way to do this for the `State struct`:

```
bool operator< (const State& lhs, const State& rhs) {
    if (lhs. name != rhs.name) {
        return lhs.name < rhs.name;
    }
    if (lhs.electoralVotes != rhs.electoralVotes) {
        return lhs.electoralVotes < rhs.electoralVotes;
    }
    return lhs.popularVotes < rhs.popularVotes;
}
```

That being said, in Assignment 4 you were guaranteed that no two states have the same name, which means that in principle we only need to compare the name field. If we did this, it would look like this:

```
bool operator< (const State& lhs, const State& rhs) {
    return lhs.name < rhs.name;
}
```

Want to learn more about where these properties of the less-than operator come from? Take CS103!

Problem Seven: Freeing Trees Efficiently

Here's one possible implementation of this algorithm:

```
void freeTree(Node* root) {
    while (root != nullptr) {
        /* Case 1: No left child. Delete the node and move right. */
        if (root->left == nullptr) {
            /* We run into the same problem we had with deleting all the nodes in
             * a linked list: we need to free this node and advance to the right.
             * So steal a page out of that playbook!
             */
            Node* next = root->right;
            delete root;
            root = next;
        }
        /* Case 2: Has a left child. Then do a rotation! */
        else {
            /* Remember the left node for later - we need to overwrite the pointer
             * to it in a second.
             */
            Node* leftChild = root->left;

            /* Have the root pick up the subtree between it and its left child. */
            root->left = leftChild->right;

            /* The left child now acquires the root as its right child. */
            leftChild->right = root;

            /* The root now becomes the left child - it's been hoisted up! */
            root = leftChild;
        }
    }
}
```

This algorithm is closely related to one called the ***Day-Stout-Warren algorithm*** (or DSW) that, given any tree, automatically rebalances it in time O(n) and space O(1). Check it out if you're curious!

Other cool facts about tree rotations: using tree rotations, it's always possible to convert any BST for a set of values into any other BST for the same set of values. And if you have a good working implementation of tree rotations going, you can implement all sorts of nice balanced trees. I'd recommend checking out ***splay trees*** or ***treaps*** as starting points, as they're both relatively easy to code up.

Problem Eight: Counting BSTs

Let's imagine we have a group of n values and we want to form a BST from them. Let's suppose we pick the kth-smallest value and put it up at the root. That means that there will be k nodes in the left subtree and $(n - k - 1)$ nodes in the right subtree (do you see why?) If we build any BST we'd like out of the k nodes in the left and the $(n - k - 1)$ nodes in the right subtree, we can combine those trees together with the kth-smallest node as the root to form an overall BST for all the values. Essentially, for each way we can pick

- which node is at the root,
- which tree we want to use for the smaller values, and
- which tree we want to use for the largest values,

we'll get back one possible BST we can form, and in fact every BST we could make will fit into this framework. We can therefore use the following beautiful recursive algorithm to solve this problem:

```
int numBSTsOfSize(int n) {
    /* Base case: There's only one tree of size 0, namely, the empty BST. */
    if (n == 0) return 1;

    /* Recursive case: Imagine all possible ways to choose a root and build the
     * left and right subtrees.
     */
    int result = 0;

    /* Put the the nodes at indices 0, 1, 2, …, n-1 up at the root. */
    for (int i = 0; i < n; i++) {
        /* Each combination of a BST of i elements and a BST of n - 1 - i elements
         * can be used to build one BST of n elements. The number of pairs of
         * trees we can make this way is given by the product of the number of
         * trees of each type.
         */
        result += numBSTsOfSize(i) * numBSTsOfSize(n - 1 - i);
    }
    return result;
}
```

This problem is just *screaming* for memoization, since we'll end up recomputing the same values a *lot* of times. We'll leave that as an exercise to the reader. ☺

Fun fact: the exact number of BSTs you can make from n elements is given by the nth ***Catalan number***. Check out the Wikipedia entry: it's really interesting! There happen to be the same number of BSTs for n elements as, say, the number of ways you can write a string of n open and close parentheses that match one another, or the number of ways you can triangulate an $(n+2)$-vertex polygon, etc. Oh, and this shows up in a couple crazy cool algorithms. Take CS166 for details!

Problem Nine: Checking BST Validity

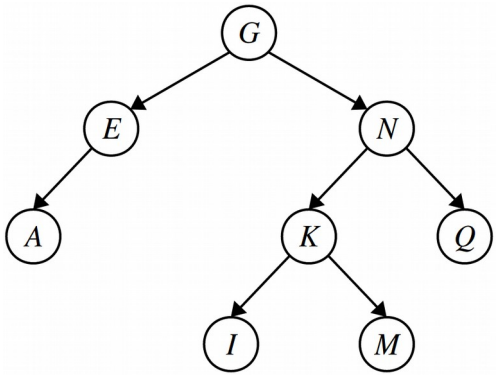
There are a bunch of different ways that you could write this function. The one that we'll use is based on recursive definition of a BST from lecture: a BST is either empty, or it's a node x whose left subtree is a BST of values smaller than x and whose right subtree is a BST of values greater than x.

This solution works by walking down the tree, at each point keeping track of two pointers to nodes that delimit the range of values we need to stay within.

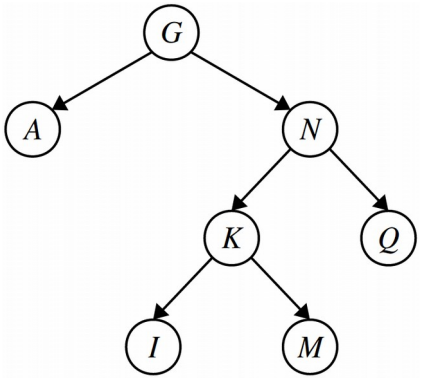
```
bool isBSTRec(Node* root, Node* lowerBound, Node* upperBound) {
    /* Base case: The empty tree is always valid.*/
    if (root == nullptr) return true;

    /* Otherwise, make sure this value is in the proper range. */
    if (lowerBound != nullptr && root->value <= lowerBound->value) return false;
    if (upperBound != nullptr && root->value >= upperBound->value) return false;

    /* Okay! We're in range. So now we just need to confirm that the left and
     * right subtrees are good as well. Notice how the range changes based on the
     * introduction of this node.
     */
    return isBSTRec(root->left,  lowerBound, root) &&
           isBSTRec(root->right, root, upperBound);
}

bool isBST(Node* root) {
    return isBSTRec(root, nullptr, nullptr);
}
```

Problem Ten: Deleting from a BST

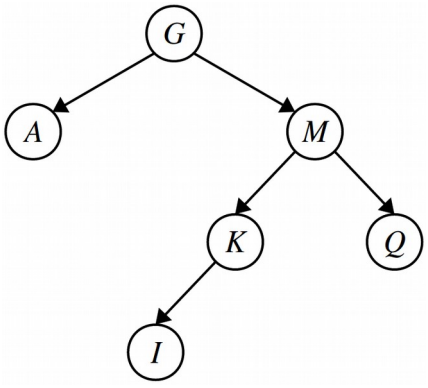
C is a leaf node, so we can delete it by just removing it from the tree:



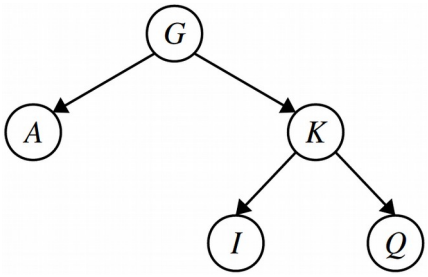
The node *E* has only a single child, so we just remove it and replace it with its child to get this tree:



The node *N* has two children. The largest value in its left subtree is *M*, so we replace *M* by *N*, then delete the node *N*. This is shown here:



The node *M* now has two children. To delete *M*, we start by finding the largest value in its left subtree (*K*) and replacing *M* by *K*. We then delete the node that formerly held *K*. Since that node has exactly one child, we just replace it with its one child. The result is shown here:



Here's some code for how to do this deletion.

```
void removeFrom(Node*& root, int value) {
    /* If the tree is empty, there's nothing to remove! */
    if (root == nullptr) return;

    /* If the node to delete is to the left, remove it from there. */
    else if (value < root->value) {
        removeFrom(root->left, value);
    }
    /* If the node to delete is to the right, remove from there. */
    else if (value > root->value) {
        removeFrom(root->right, value);
    }
    /* Otherwise, we've found the node to remove - so go remove it! */
    else {
        performDeletion(root);
    }
}

/* Actually does the deletion necessary to remove a node from the tree. */
void performDeletion(Node*& toRemove) {
    /* Case 1: The node is a leaf. Then we just delete it. */
    if (toRemove->left == nullptr && toRemove->right == nullptr) {
        delete toRemove;

        /* Inform whoever was pointing at us that we no longer exist. */
        toRemove = nullptr;
    }
    /* Case 2a: Only have a left child. */
    else if (toRemove->right == nullptr) {
        Node* replacement = toRemove->left;
        delete toRemove;
        toRemove = replacement;
    }
    /* Case 2b: Only have a right child. */
    else if (toRemove->left == nullptr) {
        Node* replacement = toRemove->right;
        delete toRemove;
        toRemove = replacement;
    }
    /* Case 3: Replace this node with the largest node in its left subtree. */
    else toRemove->value = removeLargestFrom(toRemove->left);
}

/* Deletes the largest value from the specified tree, returning that value. */
int removeLargestFrom(Node*& root) {
    if (root->right == nullptr) {
        int result = root->value;
        performDeletion(root);
        return result;
    }
    return removeLargestFrom(root->right);
}
```

In terms of efficiency, notice that this code always moves downward in the tree, never upward, and we only do O(1) work per node we visit. As a result, the time complexity of this code is O(h), where h is the height of the tree.