Recall that a BST stores comparable elements in a binary tree such that:

• for every node r:
  • all nodes in the left BST are < element at r
  • all nodes in the right BST are > element at r

We then implemented find(x) and insert(x). They had a lot of checks for null. We implemented isEmpty, isLeaf, leafCount, height, inorder, and bfs.

We implemented one class that represents an arbitrary BST:

```java
public class BinarySearchTree<T extends Comparable<? super T>> {
    private T data; // data in the root node
    private BST<T> left; // the left child, if it exists
    private BST<T> right; // the right child, if it exists
    private int size; // num nodes in the tree

    public BST(T data, BST<T> left, BST<T> right) {
        this.data = data;
        this.left = left;
        this.right = right;
        if (data == null) size = 0; // empty trees have null data
        else size = 1;
        if (left != null) size += left.size();
        if (right != null) size += right.size();
    }

    // construct an empty tree
    public BST() {
        this(null, null, null);
    }

    // traverse/inorder() - go through the nodes of the tree in increasing order
    public BST<T> inorder() {
        BST<T> result = new BST<T>();
        BST<T> current = this;
        while (current != null) {
            BST<T> left = current.left;
            if (left != null) {
                BST<T> leftInorder = left.inorder();
                result.insert(leftInorder.root);
            }
            result.insert(current.data);
            BST<T> right = current.right;
            if (right != null) {
                BST<T> rightInorder = right.inorder();
                result.insert(rightInorder.root);
            }
            current = current.next;
        }
        return result;
    }

    // find(x) - return item that matches x
    public T find(T x) {
        if (x == null) return null;
        BST<T> current = this;
        while (current != null) {
            if (x.compareTo(current.data) < 0) current = current.left;
            else if (x.compareTo(current.data) > 0) current = current.right;
            else return current.data;
        }
        return null;
    }

    // insert(x) - insert element x into its correct position in the tree
    public void insert(T x) {
        if (x == null) return;
        BST<T> current = this;
        while (current != null) {
            if (x.compareTo(current.data) < 0) {
                if (current.left == null) current.left = new BST<T>(x);
                else current = current.left;
            } else if (x.compareTo(current.data) > 0) {
                if (current.right == null) current.right = new BST<T>(x);
                else current = current.right;
            } else return;
        }
    }
}
```
BST Implementation

FindMax and findMin are very easy. We discussed the idea in class:

```java
class BinarySearchTree<T extends Comparable<? super T>> {
    private T data; // data in the root node
    private BST<T> left; // the left child, if it exists
    private BST<T> right; // the right child, if it exists
    private int size; // number of nodes in the tree

    public T findMax(T x) {
        if (isEmpty()) {
            return null; // or throw NoSuchElementException
        }
        if (right == null) { // if no right child, this is the max
            return data;
        }
        return right.findMax(x);
    }
}
```  

Question: What is the running time of findMax?

AVL Trees

An AVL tree is a binary search tree such that for every node r:

- the heights of the left and right subtrees differ by at most 1,
- the left child is an AVL tree, and
- the right child is an AVL tree.

Equivalently, we can define an AVL tree as:

- empty, or
- a root node r connected to up to two AVL subtrees such that the heights of the two subtrees differ by at most 1.

So if we order R-connected in a list of AVL subtrees such that the heights of:

- the left child is on AVL tree at.
- the right child is on AVL tree at.
- the subtrees differ by at most 1.

An AVL tree has a binary search tree each for every node r.

So all of the BST operations are O\left(\text{height of tree}\right) = O(n). Want height(log n).

The trickiest is remove(x). 3 cases to consider...

```java
public void remove(T x) {
    if (isEmpty()) {
        throw exception;
    }
    if (x.compareTo(data) < 0) { // if x < data
        left.remove(x); // may need case if left leaf
    }
    if (x.compareTo(data) > 0) { // if x > data
        right.remove(x); // may need case if right leaf
    }
    else { // x == data
        if (isLeaf()) {
            data = null; // would rather avoid this
        }
        else if (left != null) && (right != null) {
            data = right.findMin();
            right.removeMin(); // easy b/c min node has no left child
        }
        else if (left != null) { // but right==null
            this.data = left.data; // set this node to be the left subtree (i.e. shift up)
            this.left = left.left;
            this.right = left.right;
            size--;
        }
        else if (right != null) { // but left==null
            this.data = right.data;
            this.left = right.left;
            this.right = null;
            size--;
        }
    }
}
```  

Question: What is the running time of remove?

Thus, half of all the BST operations can all tree will be O(1), yay!

But we'll lose this advantage because the height of the tree.

The downside is that our implementation will have to maintain the new property:

- the heights of the left and right subtrees differ by at most 1,
- the left child is an AVL tree, and
- the right child is an AVL tree.

So an AVL tree is just a BST with an additional height restriction on its subtrees.

So our algorithm for BST:

- If the node is a leaf, search its node for every node r.
- If an AVL tree is removed, O(1).
- If an AVL tree is removed, O(n).

BST Implementation

AVL Trees
AVL Trees

An AVL tree is a binary search tree such that for every node $r$:

- The heights of the left and right subtrees differ by at most 1,
- The left child is an AVL tree, and
- The right child is an AVL tree.

Examples of AVL Trees (on board):

AVL Rebalancing

To rebalance an AVL tree $T$, do the following steps:

1. Find the smallest unbalanced subtree in $T$, and label its root $z$.
   (Note: find $z$ by starting at the node we inserted or deleted, and moving upwards towards the root, checking unbalanced-ness as we go.)

2. Since $z$ is the smallest unbalanced tree, its children are balanced AVL trees, but their heights differ by $\geq 1$.

3. Let $Y$ be $z$'s tallest child.
4. Let $X$ be $Y$'s tallest child.
5. Let $t_0, t_1, t_2, t_3$ be the subtrees of $X, Y, Z$, labeled from left to right.
6. Let $a, b, c$ be the data of $X, Y, Z$ in increasing order.

Replace the subtree at $Z$ with the subtree at $b$, children $a, c$, etc... see fig.

In particular, we need to re-implement our adds and removes, because they are the methods that mess with our tree, and can potentially unbalance it.

The idea is: do our usual insert (as a leaf), or remove. If the new tree is still balanced, great! Otherwise, we've unbalanced it, and we need to rebalance.