CS 151

Binary Search Trees and AVL Trees
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

Some BST operations are
• standard tree operations such as size(), isEmpty(), leafCount(), etc.
• insert(x) - insert element x into its correct position in the tree
• remove(x) - remove x from the tree, if it exists
• removeMin() - remove the minimum element from the tree (where is it?)
• removeMax() - remove the maximum element from the tree (where is it?)
• find(x) - return item that matches x
• findMin() - return the minimum element of the tree
• findMax() - return the maximum element of the tree
• traverse() / iterate() - go through the nodes of the tree in increasing order
We implemented our BST with one class that represents an arbitrary BST:

```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    // 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()

    public BST( )   // construct an empty tree
        this( null, null, null )
}
```
We implemented isEmpty, isLeaf, leafCount, height, inorder, and bfs. None of these were specific to a binary search tree (they are tree methods).

We then implemented find(x) and insert(x). They had a lot checks for null.

The running time of both was $O(\text{height of tree}) = O(n)$, because they both start from the root and may need to traverse one branch of the tree down to a leaf node. The longest such path is the height of the tree, hence the RT.
FindMax and findMin are very easy. We discussed the idea in class:

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 T findMax(T x) {
        if (isEmpty()) {
            return null; // or throw NoSuchElementException?
        } else if (right == null) { // if no right child, this is the max
            return data;
        } else {
            return right.findMax(x);
        }
    } // Question: what is the running time of findMax?
Let’s try removeMin(x) – two cases, depending on whether min is leaf or not

class BinarySearchTree<T extends Comparable<? super T>> {
    // BST class variables go here

    public void removeMin() {
        if (isEmpty())
            throw NoSuchElementException;
        if (isLeaf()) // this is the smallest element
            data = null;
        else if (left != null) // smaller elements exist
            if (left.isLeaf()) left = null; // b/c of our implem’n
        else
            // no left child exists, but a right
            this.data = right.data; // one does. shift it up
            this.left = right.left;
            this.right = right.right;
            size--;
    } // Question: what is the running time of removeMin()?
The trickiest is remove(x). 3 cases to consider...

class BinarySearchTree<T extends Comparable<? super T>> {
    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 L child
            else if( left != null ) // but right==null
                // set this node to be the left subtree (i.e. shift up)
            else if( right != null ) // but left==null
                // set this node to be the right subtree
            size--
} // Question: What is the running time of remove?
AVL Trees

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

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 either

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

So an AVL tree is just a BST with an additional height restriction on its subtrees. We will show that this height restriction guarantees that the height of the whole tree is $O(\log n)$. (This will take a bit of proof-work, but it’s true.)

The downside is that our implementation will have to maintain the new property. But we’ll ensure this maintenance doesn’t take longer than $O(\log n)$, either...

Thus, the RT of all the BST operations with an AVL tree will be $O(\log n)$, yay!
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.
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.

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

• 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 move upwards towards the root, checking unbalanced-ness as we go.)
• Since z is the smallest unbalanced tree, its children are balanced AVL trees, but their heights differ by > 1.
• Let Y be z’s tallest child.
• Let X be Y’s tallest child.
• Let t0, t1, t2, and t3 be the subtrees of X,Y,Z, labeled from left to right.
• Let a,b,c be data of X,Y,Z in increasing order.
• Replace the subtree at Z with the subtree at b, children a,c, etc... see fig.