CSE 143

Lecture 21: Binary Search Trees; TreeSet

DR. SCIENTIST! THE SORT HAS FAILED! WHAT'S THE WORST-CASE SCENARIO FOR THE ALGORITHM?

THE WORST-CASE SCENARIO IS WHEN THE DATA IS ALREADY SORTED.
Recall: \( x = \text{change}(x) \)

- Methods that modify a tree should have the following pattern:
  - input (parameter): old state of the node
  - output (return): new state of the node

- In order to actually change the tree, you must reassign:

\[
\begin{align*}
\text{node} & = \text{change}(	ext{node}, \text{parameters}) ;; \\
\text{node.left} & = \text{change}(	ext{node.left}, \text{parameters}) ;; \\
\text{node.right} & = \text{change}(	ext{node.right}, \text{parameters}) ;; \\
\text{overallRoot} & = \text{change}(	ext{overallRoot}, \text{parameters}) ;
\end{align*}
\]
Exercise

- Add a method `getMin` to the `IntTree` class that returns the minimum integer value from the tree. Assume that the elements of the `IntTree` constitute a legal binary search tree. Throw a `NoSuchElementException` if the tree is empty.

```java
int min = tree.getMin(); // -3
```
Exercise solution

// Returns the minimum value from this BST.
// Throws a NoSuchElementException if the tree is empty.
public int getMin() {
    if (overallRoot == null) {
        throw new NoSuchElementException();
    }
    return getMin(overallRoot);
}

private int getMin(IntTreeNode root) {
    if (root.left == null) {
        return root.data;
    } else {
        return getMin(root.left);
    }
}
Exercise

- Add a method `remove` to the `IntTree` class that removes a given integer value from the tree, if present. Remove the value in such a way as to maintain BST ordering.

```java
    tree.remove(73);
    tree.remove(29);
    tree.remove(87);
    tree.remove(55);
```

```
overall root

55
  /   \
29    87
  /     /
42  60  91
     /  /
36  73
```
Cases for removal 1

1. a leaf: replace with null
2. a node with a left child only: replace with left child
3. a node with a right child only: replace with right child

```
tree.remove(-3);
tree.remove(29);
tree.remove(55);
tree.remove(29);
tree.remove(42);
tree.remove(42);
```

```
overall root

29

55

42

overall root

29

55

42

overall root

29

55

42

overall root

29

42

```
4. a node with both children: replace with min from right
   • (replacing with max from left would also work)

```
tree.remove(55);
```
public void remove(int value) {
    overallRoot = remove(overallRoot, value);
}

private IntTreeNode remove(IntTreeNode root, int value) {
    if (root == null) {
        return null;
    } else if (root.data > value) {
        root.left = remove(root.left, value);
    } else if (root.data < value) {
        root.right = remove(root.right, value);
    } else {
        // root.data == value; remove this node
        if (root.right == null) {
            return root.left; // no R child; replace w/ L
        } else if (root.left == null) {
            return root.right; // no L child; replace w/ R
        } else {
            // both children; replace w/ min from R
            root.data = getMin(root.right);
            root.right = remove(root.right, root.data);
        }
    }
    return root;
}
The BSTs below contain the same elements.

What orders are "better" for searching?

- Searching BSTs
Trees and balance

- **balanced tree**: One whose subtrees differ in height by at most 1 and are themselves balanced.
  - A balanced tree of \( N \) nodes has a height of \( \sim \log_2 N \).
  - A very unbalanced tree can have a height close to \( N \).

- The runtime of adding to / searching a BST is closely related to height.

- Some tree collections (e.g. TreeSet) contain code to balance themselves as new nodes are added.