RECURSIVE BST OPERATIONS

Let's implement a BST class, avoiding iteration.

This will give us more practice with trees, and with recursion.

Interface for our BST Class

Our BST is a tree, and thus can be implemented in a number of ways. so we make BST an interface.

A BST holds its elements in order, so we use `Comparable` as the type of those elements. Then we can build BSTs containing any orderable type of object.

```
interface BST {

    /** Insert k into me, if it's not already there. */
    public void insert(Comparable k);

    /** Delete k from me, if it's there. */
    public void delete(Comparable k);

    /** Print the contents of me, in order. */
    public void inorderPrint();

    /** Return whether I contain k. */
    public boolean contains(Comparable k);

}
```


Implementation Decisions

Data Structure

We use objects and references to represent the nodes and edges of a tree.

Because which node is the root of a tree can change, we make two classes: one for tree nodes, and one to refer to the root. (We used the same approach with linked lists.)

For the nodes, we can use the same `BSTNode` class as on an earlier slide.

```
class BSTNode {
    public Comparable key;
    public BSTNode left;
    public BSTNode right;
    public BSTNode(Comparable key) {
        this.key = key;
    }
}
```


Because of our implementation, we call the class that acts as the tree `LinkedBST`.

Data Members

We need only one data member inside our `LinkedBST` class.

```
public class LinkedBST implements BST {

    private BSTNode root;

    /** Insert k into me, if it's not already there. */
    public void insert(Comparable k) { ... }

    /** Delete k from me, if it's there. */
    public void delete(Comparable k) { ... }

    /** Print the contents of me, in order. */
    public void inorderPrint() { ... }

    /** Return whether I contain k. */
    public boolean contains(Comparable k) { ... }

    ... maybe others ...

}
```

The `contains` method

What is our "basic strategy" (step 1 from "Writing a Recursive Method")?

What is the "flow of information"?

The method that searches a *subtree* must know the root of that subtree. There are (at least) two ways to implement this method:

1. As a static method in `LinkedBST`, passing the root `BSTNode` as a parameter.
2. As an instance method in `BSTNode`, calling it on the root `BSTNode`.

We take the first approach.

The `contains` method in `BST` doesn't have a node parameter since that's an implementation detail. So we make a helper method.

Now, develop the code using the remaining steps.

The code

```
/** Return whether I contain k. */
public boolean contains(Comparable k) {
    return contains(root, k);
}

/** Return whether k is in the tree rooted at t. */
private static boolean contains(BSTNode t,
                                Comparable k) {
    if (t == null) {
        return false;
    } else if (k.compareTo(t.key) == 0) {
        return true;
    } else if (k.compareTo(t.key) < 0) {
        return contains(t.left, k);
    } else { // k.compareTo(t.key) > 0
        return contains(t.right, k);
    }
}
```

Question: Why did we make `contains(BSTNode, Comparable)` static, but not `contains(Comparable)`?

Question: Can this be written elegantly with only iteration?

Exercise: Write `contains` without using an `if` statement.

The `insert` method

Design

We insert an element at the point where we 'fall off' the tree looking for it.

To insert `k` into tree `t`:

- If `t` is empty, replace `t` by a tree consisting of a single node with value `k`.
- If `t` has `k` at its root, `k` is already in `t`. Return without modifying `t`.
- If `k` is less than the value at the root of `t`, insert `k` into the left subtree of `t`.
- If `k` is greater than the value at the root of `t`, insert `k` into the right subtree of `t`.

Inserting a node requires a change to its parent. In our recursion, we'll pass information back to the parent so it can change itself.

The code

```
/** Insert k into me, if it's not already there. */
public void insert(Comparable k) {
    root = insert(root, k);
}

/** Insert k into the tree rooted at t, and
 *  return the root of the resulting tree. */
private static BSTNode insert(BSTNode t,
                              Comparable k) {
    if (t == null) {
        t = new BSTNode(k);
    } else if (k.compareTo(t.key) < 0) {
        t.left = insert(t.left, k);
    } else if (k.compareTo(t.key) > 0) {
        t.right = insert(t.right, k);
    } // else equal, don't do anything to t.

    return t;
}
```


Questions:

- Why does the statement "`t = new BSTNode(k)`" have an effect?
- We pass and return the reference `t`. How often during the recursion does it actually change in-between pass and return?

Exercises:

- Write a non-recursive insertion method for binary search trees.
- Write a recursive version that doesn't return a `BSTNode`, but instead looks ahead to see if there's a child.
- Write a recursive version that doesn't return a `BSTNode`, but instead passes information about the parent to the child in the recursive call.

The Delete Operation

Design

- Find the node you wish to delete (if it is there).
- If the node is a leaf, delete it.
- If the node has exactly one child, delete the node by making its parent refer to that child directly.
- If the node has two children, replace the value in the node by the value in its successor and then delete the successor.

Questions

In a binary search tree, where is the successor of a node with a right child?

The successor node has no left child. How do we know?

Must the successor be a leaf?

The code

Our code for `delete` is slightly shorter than our strategy suggested. Can you see how it differs, and why it still works?

```
/** Delete k from the tree rooted at t (if there)
  * and return the root of the resulting tree. */
private static BSTNode delete(BSTNode t, Comparable k) {
    if (t == null) { // k not in tree; do nothing.
    } else if (k.compareTo(t.key) < 0) {
        t.left = delete(t.left, k);
    } else if (k.compareTo(t.key) > 0) {
        t.right = delete(t.right, k);
    } else { // Found it; now delete it.
        if (t.right == null) {
            // t has at most one child, on the left.
            t = t.left;
        } else {
            // t has a right child. Replace t's value
            // with its successor value.
            Comparable successor = min(t.right);
            t.key = successor;
            // Delete that successor.
            t.right = delete(t.right, successor);
        }
    }
    return t;
}
```



```
/** Delete k from this BST, if it is there. */
public void delete(Comparable k) {
    root = delete(root, k);
}

/**
 * Return the minimum value in t.
 * Requires: t != null
 */
private static Comparable min(BSTNode t) {
    // To find the min, go left as far as possible.
    if (t.left == null) {
        return t.key;
    } else {
        return min(t.left);
    }
}
```

Questions: What is inefficient about our code in the two-children case? How could it be sped up?