RECURSIVE BST OPERATIONS

with more Java generics

Let's implement a BST class, avoiding iteration.

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

It will also give us a chance for a continued example with more of the "generics" introduced in Java 1.5. We'll supply comments on generics where we need to do something more complicated than usual. Here are some points about generics that might be helpful:

- "`class Foo<T>`" introduces a class with a *type parameter* T.

- "`<T extends Bar>`" introduces a type parameter that is required to be a descendant of the class `Bar` — with `Bar` itself a possibility. In a type parameter, "`extends`" is also used to mean "`implements`".

- "`<? extends Bar>`" is a type parameter that can be any class that extends `Bar`. The difference between this and the previous type parameter is that in the previous one we were going to use T later, whereas here we'll never mention the type again.

- "`<? super Bar>`" is a type parameter that can be any class that is an ancestor of `Bar` (including, again, `Bar` itself).

- You can also have generic *methods* (as well as generic classes). Oddly, the type parameter for a generic method goes *before* the return type in the method header, like this:

  ```
  public static <T> void methName(T data) { ...
  ```

2

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<E extends Comparable<? super E>> {
    // The element type in a BST must be Comparable, so that we can
    // run searches. However, it may have inherited the compareTo(),
    // so it's Comparable<? super E>, not just Comparable<E>.

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

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

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

    /** Return whether I contain k. */
    public boolean contains(E 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<T> {
    // Unlike BST, BSTNode does not require its data to be Comparable,
    // and we expect all the BSTNodes in any actual tree to be of the
    // same type, so the type parameter is simply T.
    public T key;
    public BSTNode<T> left;
    public BSTNode<T> right;
    public BSTNode(T 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.

```java
public class LinkedBST<E extends Comparable<? super E>> implements BST<E> {
    // Notice that we have to implement BST<E>, and not just BST.

    private BSTNode<E> root;

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

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

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

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


/** Return whether k is in the tree rooted at t. */
private static <T extends Comparable<? super T>> boolean contains(BSTNode<T> t,
                                T k) {
    // Notice the type parameter, which is independent of the class parameter E.
    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<T>, T)` static, but not `contains(E)`?

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(E k) {
    root = insert(root, k);
}


/** Insert k into the tree rooted at t, and
 *  return the root of the resulting tree. */
private static <T extends Comparable<? super T>> BSTNode<T> insert(BSTNode<T> t,
                                    T k) {
    if (t == null) {
        t = new BSTNode<T>(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<T>(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?

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 <T extends Comparable<? super T>> BSTNode<T> delete(
        BSTNode<T> t, T 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.
            T 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(E k) {
    root = delete(root, k);
}


/**
 * Return the minimum value in t.
 * Requires: t != null
 */
private static <T extends Comparable<? super T>> T min(BSTNode<T> 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?