private BNode insert (Comparable x, BNode a) {
  if (a == null)  a = new BNode (x);
  else if (x.leaThan (a.data))
    a.left = insert (x, a.left);
  else if (a.data.leaThan (x))
    a.right = insert (x, a.right);
  else  // duplicate entry, so do nothing
    return a;
}

public void insert (Comparable x) {
  root = insert (x, root);
}

The recursion here is ...

root = insert (4, root)  no change yet to root — recursion opens insert

following the arrows here shows that only at the very bottom when
a new node is created, is there
any real change due to =
```java
private BNode remove (Comparable x, BNode a)
{
    if (a == null) return a; // not there to remove!!
    if (x.compareTo(a.data) < 0)
        a.left = remove (x, a.left);
    else if (x.compareTo(a.data) > 0)
        a.right = remove (x, a.right);
    else if (a.left != null && a.right != null)
    {
        a.data = findMin (a.right).data;
        a.right = remove (x, a.right);
    }
    else
        a = (a.left != null) ? a.left : a.right;
    return a;
}

public void remove (Comparable x)
{
    root = remove (x, root);
}

let's see what's going on with 'remove'...

Note that all the 'unfinished' 'spurs' are really null, not just the one marked.

Removing a leaf, such as 20, is easy...

remove (20) ➔ root = remove (20, root) ➔ * * * *
all return control
with no change
return Y ➔ Y = null;
```
Removing a node which has only 1 child, such as 6, is also easy — the non-trivial subtree of that node gets promoted to that node's position.

```
remove (6);
```

Situation after removing 6

The recursion is similar to the previous case.

Notice that the node 6 has not been formally deleted (although 'garbage collection' will take care of this), but that α's 'left' (which previously was directed to 6) is now directed to ε. This means of course that 6 is no longer being referenced, hence is subject to 'garbage collection'.

Finally, removing a node which has 2 children is a little more delicate, although the recursion is still fairly easy to follow (even though there are now 2 recursions going on!). This code could be made more efficient, but let's remove 15 for an illustration...

```
remove (15);
```

Notice that the smallest data value in α's right subtree, 20, was made as α's data.
The last method we need to implement is...

```java
private void printTree (TreeNode a) {
    if (a != null) {
        printTree(a.left);
        System.out.println(a.data);
        printTree(a.right);
    }
}
```

```java
public void printTree () {
    if (is Empty ()) System.out.println("Empty");
    else printTree(root);
}
```

With our standard example, printTree() produces from...

```
With our standard example, printTree() produces from...
```

```
1
   2
      null
    null

3
   4
    null

15
   6
     8
     null
   9
     null

23
  20
     24
     null
```

the recursion...

```
3
   2
      null
    null

15
   6
     8
     null
   9
     null

23
  20
     24
     null
```

```
private void printTree (TreeNode a) {
    if (a != null) {
        printTree(a.left);
        System.out.println(a.data);
        printTree(a.right);
    }
}
```