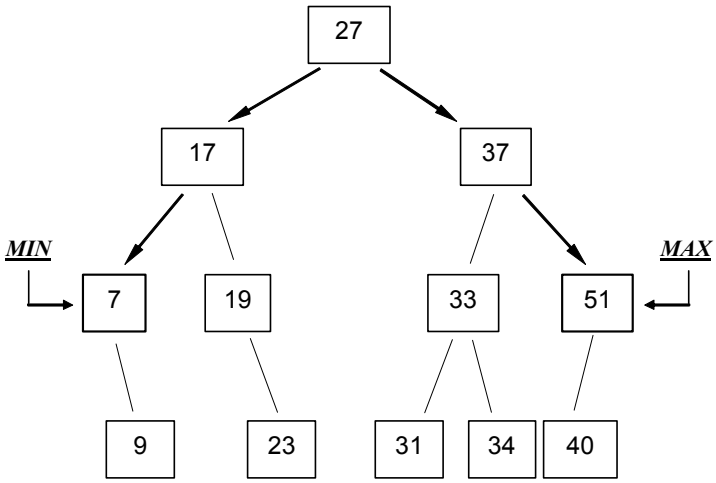

Figure 23-8. Location of the smallest and largest values in a BST

23.5 A Do-It-Yourself BST

Figure 23-9 shows a fragment from a simple class that implements a BST. The field `root` refers to the root of the tree. The no-args constructor creates an empty tree by setting `root` to `null`. We have provided the `contains`, `add`, `remove`, and `toString` methods. This implementation assumes that the nodes of the tree hold `Comparable` objects, but it would be easy to provide another constructor that takes a comparator as a parameter.

Each of our four methods (`contains`, `add`, `remove`, and `toString`) uses a respective private recursive helper method, which takes an additional parameter: the root node of a subtree for which the method is called (Figure 23-10).

```
// Implements a BST with TreeNode nodes.

public class MyTreeSet
{
  private TreeNode root;  // holds the root of this BST

  // Constructor: creates an empty BST.
  public MyTreeSet()
  {
    root = null;
  }

  // Returns true if this BST contains value; otherwise returns false.
  public boolean contains (Object value)
  {
    return contains(root, value);
  }

  // Adds value to this BST, unless this tree already holds value.
  // Returns true if value has been added; otherwise returns false.
  public boolean add(Object value)
  {
    if (contains(value))
      return false;
    root = add(root, value);
    return true;
  }

  // Removes value from this BST.  Returns true if value has been
  // found and removed; otherwise returns false.
  public boolean remove(Object value)
  {
    if (!contains(value))
      return false;
    root = remove(root, value);
    return true;
  }

  // Returns a string representation of this BST.
  public String toString()
  {
    String str = toString(root);
    if (str.endsWith(", "))
      str = str.substring(0, str.length() - 2);
    return "[" + str + "]";
  }
  ...  // other methods
}
```

Figure 23-9. Public methods from `MyTreeSet` (`JM\Ch23\BST\MyTreeSet.java`)

```
// Returns true if the BST rooted at node contains value; otherwise
// returns false (recursive version).
private boolean contains(TreeNode node, Object value)
{
  if (node == null)
    return false;
  else
  {
    int diff = ((Comparable<Object>)value).compareTo(node.getValue());
    if (diff == 0)
      return true;
    else if (diff < 0)
      return contains(node.getLeft(), value);
    else // if (diff > 0)
      return contains(node.getRight(), value);
  }
}

// Adds value to the BST rooted at node. Returns the root of the
// new tree.
// Precondition: the tree rooted at node does not contain value.
private TreeNode add(TreeNode node, Object value)
{
  if (node == null)
    node = new TreeNode(value);
  else
  {
    int diff = ((Comparable<Object>)value).compareTo(node.getValue());
    if (diff < 0)
      node.setLeft(add(node.getLeft(), value));
    else // if (diff > 0)
      node.setRight(add(node.getRight(), value));
  }
  return node;
}

// Removes value from the BST rooted at node.  Returns the root
// of the new tree.
// Precondition: the tree rooted at node contains value.
private TreeNode remove(TreeNode node, Object value)
{
  int diff = ((Comparable<Object>)value).compareTo(node.getValue());
  if (diff == 0)
    node = removeRoot(node);  // base case
  else if (diff < 0)
    node.setLeft(remove(node.getLeft(), value));
  else // if (diff > 0)
    node.setRight(remove(node.getRight(), value));
  return node;
}
```

Figure 23-10 `MyTreeSet.java` *Continued*

```
// Returns a string representation of the tree rooted at node.
private String toString(TreeNode node)
{
  if (node == null)
    return "";
  else
    return toString(node.getLeft()) + node.getValue() + ", " +
                        toString(node.getRight());
}
```

Figure 23-10. Recursive helper methods in JM\Ch23\BST\MyTreeSet.java

Actually, the `contains`, `add`, and `remove` methods have simple iterative versions as well, because in these methods at each step we either explore the left subtree or the right subtree, but not both. For example:

```
// Iterative version:
private boolean contains(TreeNode node, Object value)
{
  while (node != null)
  {
    int  diff = ((Comparable<Object>)value).compareTo(node.getValue());
    if (diff == 0)
      return true;
    else if (diff < 0)
      node = node.getLeft();
    else // if (diff > 0)
      node = node.getRight();
  }
  return false;
}
```

❖ ❖ ❖

In our implementation, the `add` method adds a new node as a leaf, appending it ether as the root (if the tree was empty) or to the left or to the right subtree, depending on the value (Figure 23-10).

The `remove` method is a little more involved. The most complicated part is the base case, when the target value to be removed is found at the root (of some subtree). We have isolated this case into a separate method, `removeRoot`. It removes the root node and returns a reference to the root of the new tree. The idea of the algorithm is to replace the root with the smallest node in the right subtree. (In a slightly different version, we would replace the root with the largest node in the left subtree.) Such a node can have only one child (right), so it is easy to unlink that node from the tree by promoting its right child into its place. Figure 23-11 illustrates this algorithm.

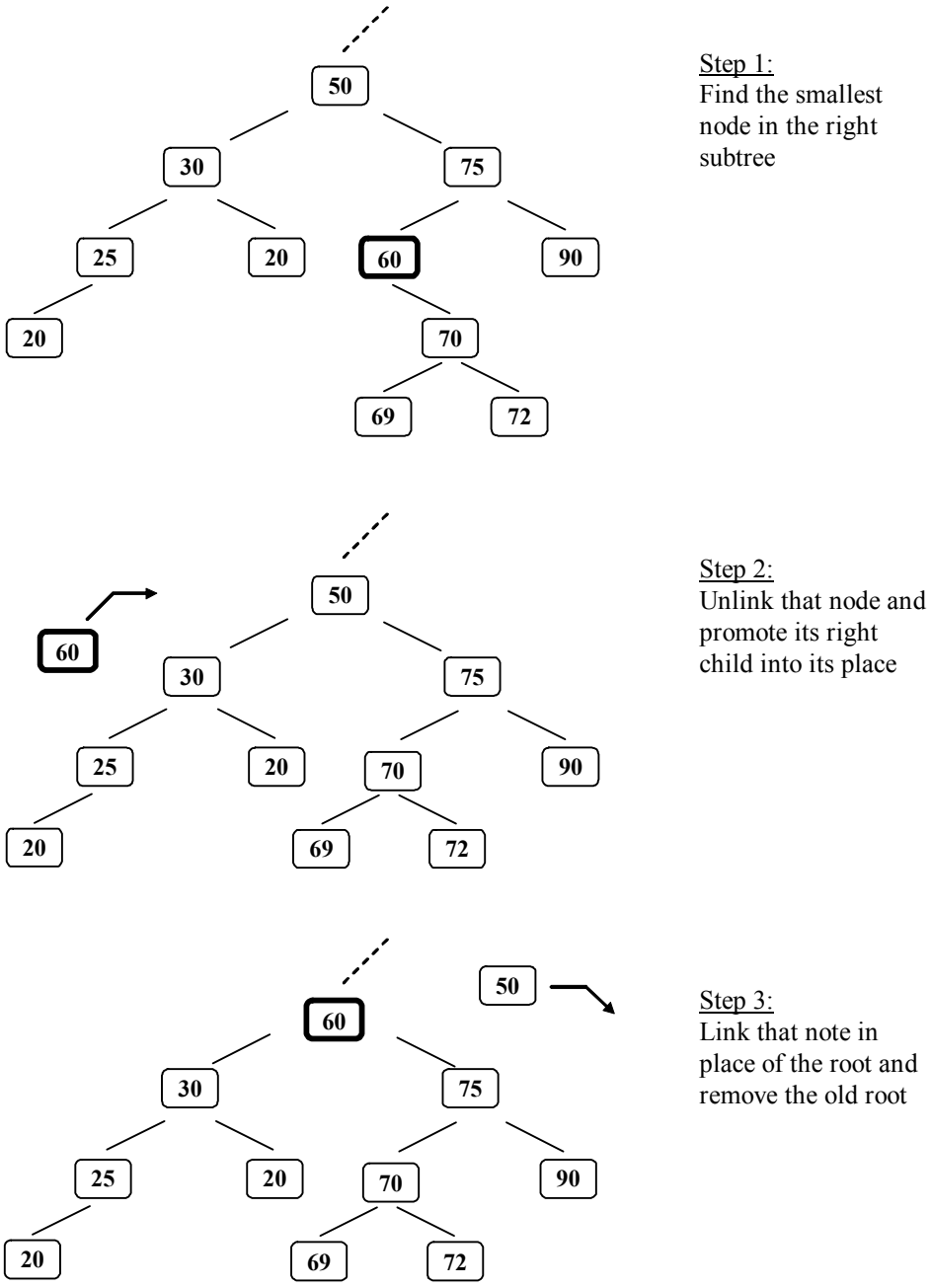

Figure 23-11. Removing the root node from a subtree in a BST

As you can see, the `contains`, `add`, and `remove` methods take, in the worst case, an amount of time proportional to the height of the tree. When the tree is reasonably "bushy," this is $O(\log n)$, where n is the number of nodes in the tree.

> **In our simplified implementation, if the values to be added to the BST arrive not in random order but close to ascending or descending order, the tree may lose its shape and deteriorate into an almost linear structure.**

A more sophisticated implementation would use algorithms that keep the tree "balanced" (but such algorithms are outside the scope of this book).

As a lab exercise, fill in the blanks in the `removeRoot` method in `JM\Ch23\BST\MyTreeSet.java`. For "extra credit," make `MyTreeSet Iterable` by implementing an `iterator` method and a `MyTreeSetIterator` class. See Section <...> for an example. Your iterator should perform inorder traversal of the tree: left subtree, root, right subtree. Use a `Stack<TreeNode>` in your `MyTreeSetIterator` class. Do not implement the `remove` method for your iterator.

23.6 *Lab:* Morse Code

Morse Hall, the Mathematics Department building at Phillips Academy in Andover, Massachusetts, is named after Samuel F. B. Morse, who graduated from the academy in 1805.

In 1838, Samuel Morse devised a signaling code for use with his electromagnetic telegraph. The code used two basic signaling elements: the "dot," a short-duration electric current, and the "dash," a longer-duration signal. The signals lowered an ink pen mounted on a special arm, which left dots and dashes on the strip of paper moving beneath. Morse's code gained wide acceptance and, in its international form, is still in use. (Samuel Morse also achieved distinction as an artist, particularly as a painter of miniatures, and between 1826 and 1845 served as the first president of the National Academy of Design.)

In 1858 Queen Victoria sent the first transatlantic telegram of ninety-eight words to congratulate President James Buchanan of the United States. The telegram started a