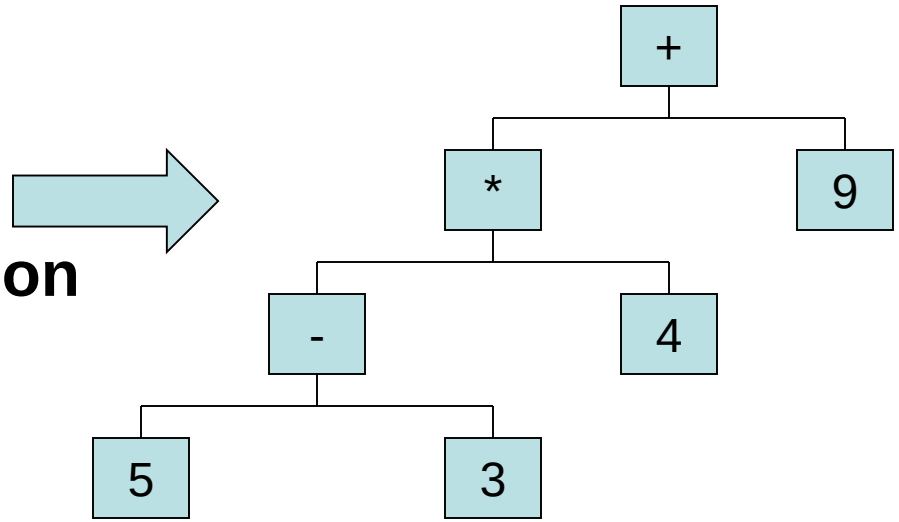
Trees and Binary Search Trees

- Using binary trees
- Definition of a binary search tree
- Implementing binary search trees
  - Add Element
  - Remove Element
- Using Binary Search Trees: Ordered Lists
- Reading: L&C 10.1 – 10.9

Using Binary Trees

- An expression tree has an operation at the root and at each internal node with an operand at each leaf
- An expression tree is evaluated from the bottom up

Infix Expression
(5 - 3) * 4 + 9
Postfix Expression
5 3 – 4 * 9 +



Using Binary Trees

- ExpressionTreeObj class

```
public class ExpressionTreeObj
{
  private int termtype;   // node or leaf
  private char operator;  // operator (node) or
  private int value       // operand value (leaf)
  // constructor
  public ExpressionTreeObj(int type, char op, int val)
  {
    // initialize the object attributes (not shown)
  }
  // typical accessor methods (also not shown)
}
```

Using Binary Trees

- ExpressionTree class

```
public class ExpressionTree extends
             LinkedBinaryTree<ExpressionTreeObj>
{
  public ExpressionTree (ExpressionTreeObj element,
                         ExpressionTree leftSubtree,
                         ExpressionTree rightSubtree)
  {
    super(element, leftSubtree, rightSubtree);
  }
  public int evaluateTree()
  {
    return evaluateNode(root);
  }
}
```

Using Binary Trees

- Postfix tree construction (See L&C Figure 12.11)

Assume the postfix expression is 5 3 – 4 * 9 +

```
Create a stack
Get 5 -> push(new ExpressionTree(5, null, null)
Get 3 -> push(new ExpressionTree(3, null, null)
Get - -> op2 = pop()
         op1 = pop()
         push(new ExpressionTree('-', op1, op2)
Get 4 -> push(new ExpressionTree(4, null, null)
Get * -> op2 = pop()
         op1 = pop()
         push(new ExpressionTree('*', op1, op2)
Get 9 -> push(new ExpressionTree(9, null, null)
Get + -> op2 = pop()
         op1 = pop()
         push(new ExpressionTree('+', op1, op2)
At end-> pop the completed ExpressionTree
```

Using Binary Trees

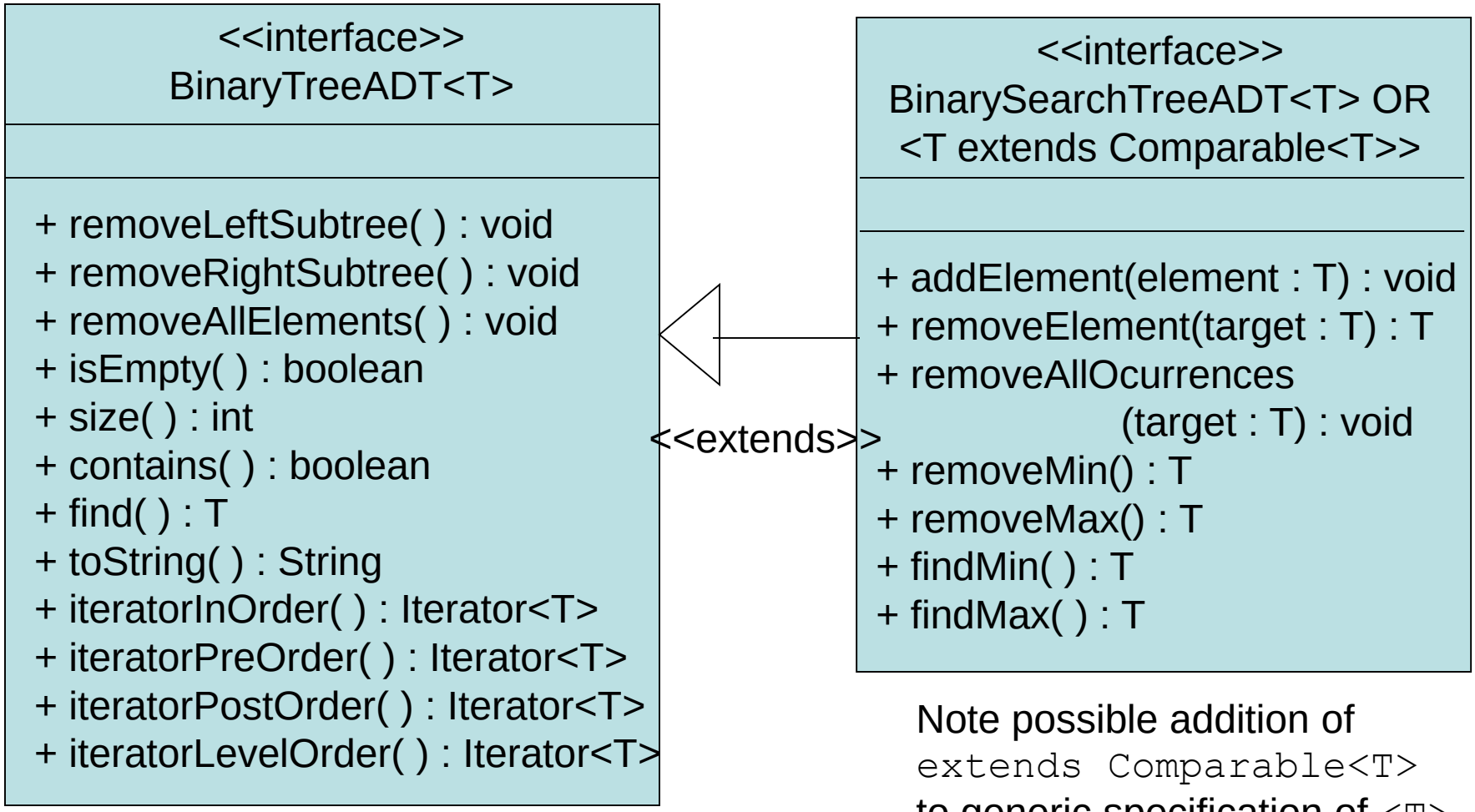
- Recursive tree evaluation method (my version)

```
private int evaluateNode(BinaryTreeNode<T> root)
{
  if (root != null) {
    ExpressionTreeObj temp = root.getElement();
    if (temp.isOperator())
      return computeTerm(temp.getOperator(),
                         evaluateNode(root.getLeft()),
                         evaluateNode(root.getRight()));
    else
      return temp.getValue();
  }
  return 0;
}
```

Binary Search Tree Definition

- A binary search tree is a binary tree with the added property that for each node:
  - The left child is less than the parent
  - The parent is less than or equal to the right child
- Class of objects stored in a binary search tree must implement Comparable interface
- Now we can define operations to add and remove elements according to search order

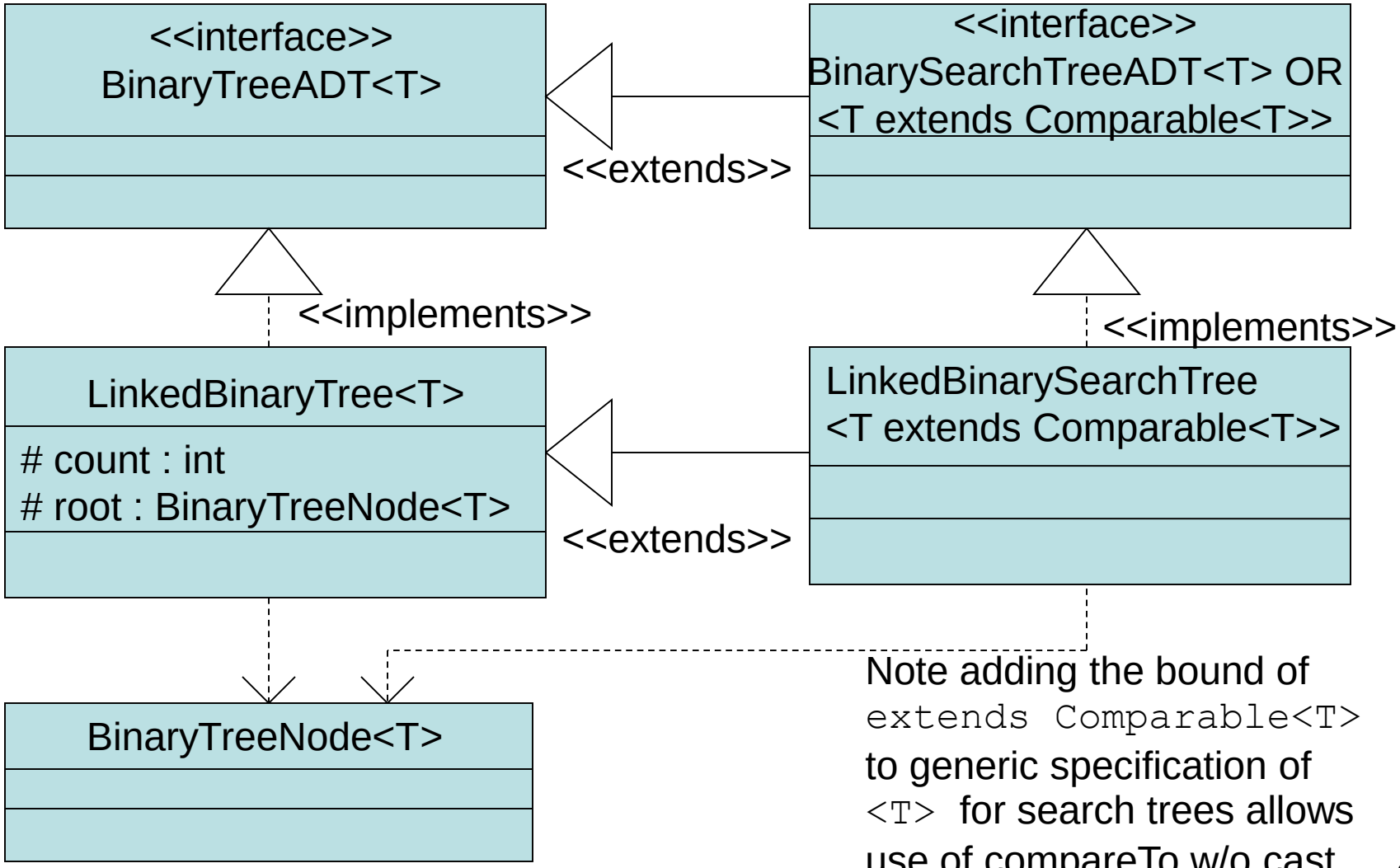
Implementing Binary Search Trees



Note: toString is missing in L&C Fig 10.2

Note possible addition of `extends Comparable<T>` to generic specification of `<T>` for search trees to optionally force a bound on type T

Implementing Binary Search Trees



Implementing Binary Search Trees

- LinkedBinarySearchTree
  - Header

```
public class LinkedBinarySearchTree
<T extends Comparable<T>> extends LinkedBinaryTree<T>
implements BinarySearchTreeADT<T>
```

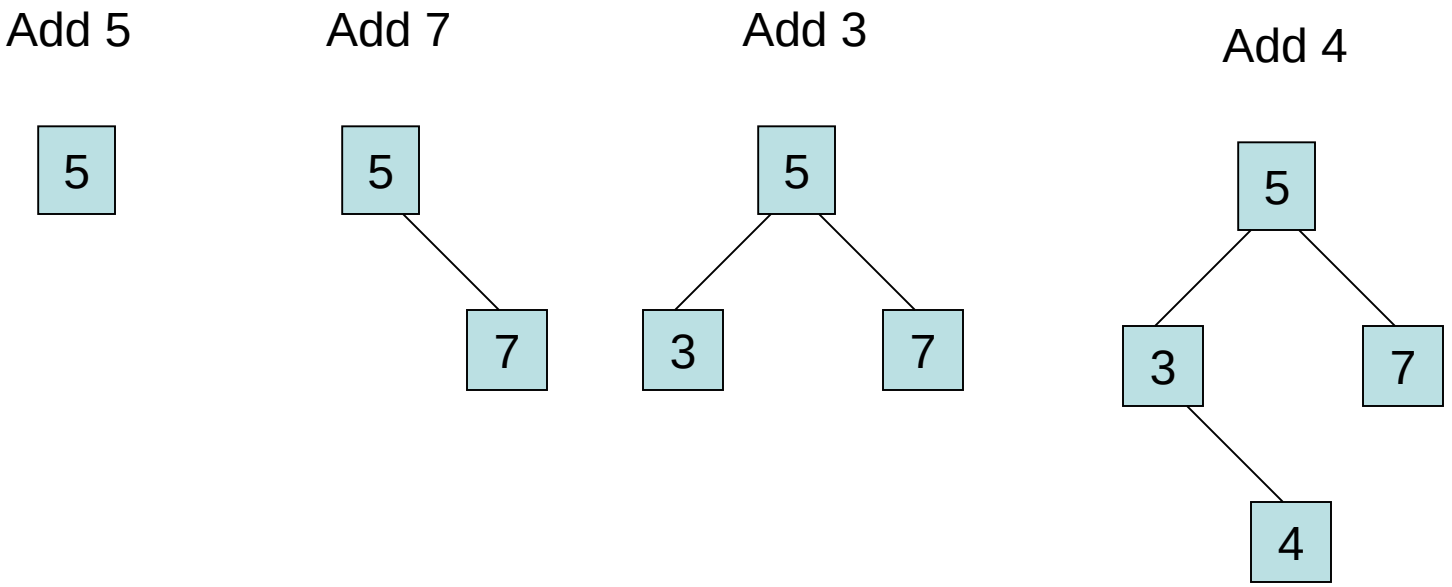
  - Constructors

```
public LinkedBinarySearchTree()
{
  super();
}
public LinkedBinarySearchTree(T element)
{
  super(element);
}
```

- Why not allow use of the overloaded three argument LinkedBinaryTree constructor?

Implementing Binary Search Trees

- Semantics for method addElement



Implementing Binary Search Trees

- LinkedBinarySearchTree method addElement

```
public void addElement(T element)
{
  BinaryTreeNode<T> temp =
                  new BinaryTreeNode<T>(element);
  if (isEmpty())  // using parent method
    root = temp;  // set parent attribute root
  else
  {               // need to add to existing tree
    BinaryTreeNode<T> current = root;
    boolean added = false;
```

Implementing Binary Search Trees

- LinkedBinarySearchTree method addElement

```
while (!added)
{ // compareTo allowed for <T extends Comparable<T>>
  if(element.compareTo(current.getElement()) < 0)
    if (current.getLeft() == null)
    {
      current.setLeft(temp);
      added = true;
    }
    else
      current = current.getLeft();
  else
```

Implementing Binary Search Trees

- LinkedBinarySearchTree method addElement

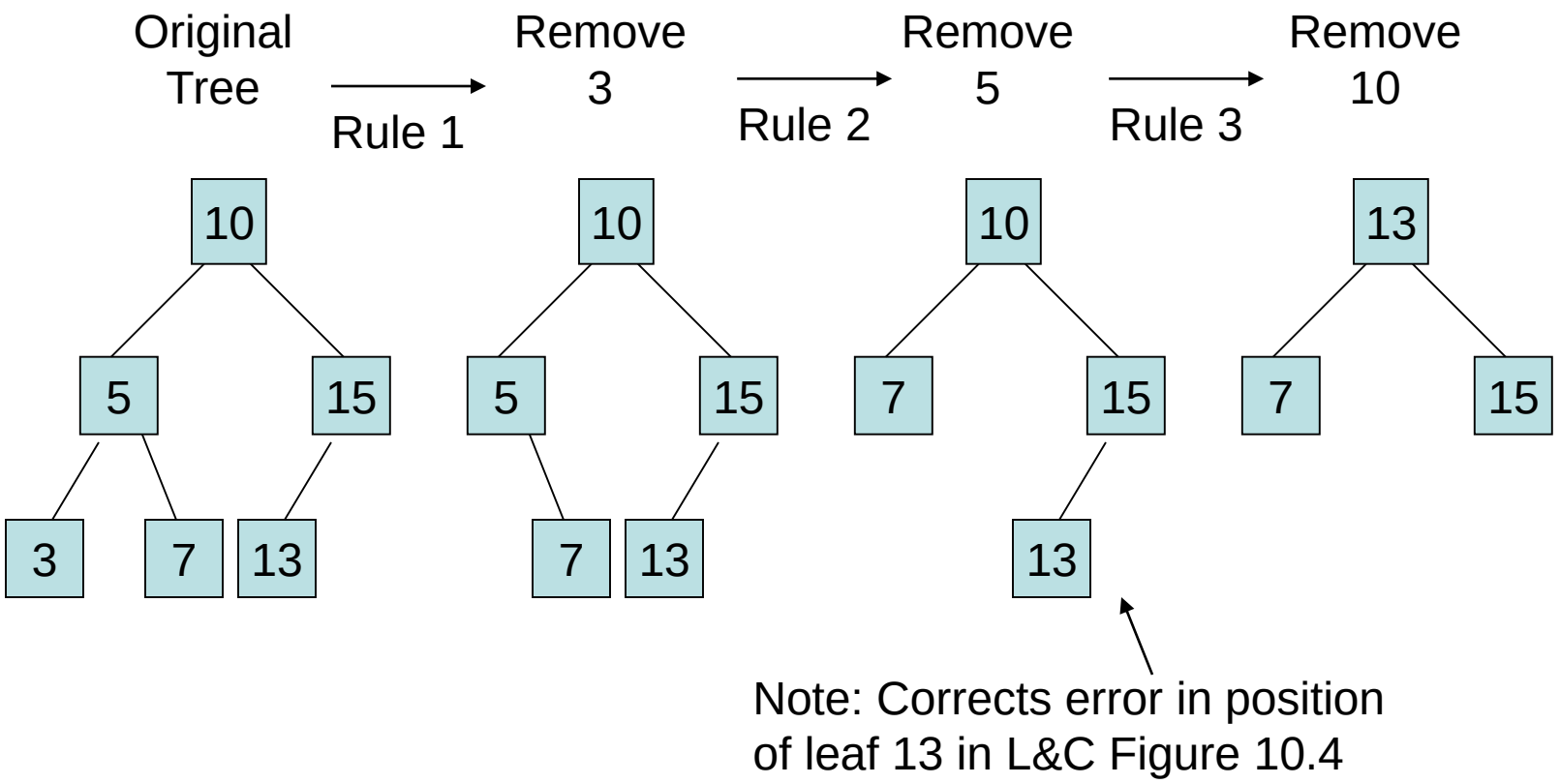
```
        if (current.getRight() == null)
        {
          current.setRight(temp);
          added = true;
        }
        else
          current = current.getRight();
      }
    }
    count++;      // increment parent attribute
  }
```

Implementing Binary Search Trees

- Semantics for method removeElement
- Can’t remove a node by making a reference point around the node being removed
- Must promote another node to take its place
  1. If node has no children, replacement is null
  2. If node has only one child, replacement is child
  3. If node has two children, replacement is the in-order successor of the node (equal elements are placed to the right)

Implementing Binary Search Trees

- Semantics for method removeElement



Implementing Binary Search Trees

- LinkedBinarySearchTree method replacement

```
protected BinaryTreeNode<T> replacement
                                  (BinaryTreeNode<T> node)
{
  if (node == null)       // paranoid programming?
    return null;
  if(node.getLeft() == null && node.getRight() == null)
    return null;
  if (node.getLeft() != null && node.getRight() == null)
    return node.getLeft();
  if (node.getLeft() == null && node.getRight() != null)
    return node.getRight();

  // easy cases done - need to work harder now
```

Implementing Binary Search Trees

- LinkedBinarySearchTree method replacement

```
  BinaryTreeNode<T> current = node.getRight();
  BinaryTreeNode<T> parent = node;
  while (current.getLeft() != null)
  {
    parent = current;
    current = current.getLeft();
  }
  if (current != node.getRight())
  {
    parent.setLeft(current.getRight());
    current.setRight(node.getRight());
  }
  current.setLeft(node.getLeft());
  return current;
}
```

Using Binary Search Trees: Implementing Ordered Lists

- BinarySearchTreeList<T> class
- A List API implemented using a BinarySearchTree
- Each List method is mapped one-to-one to a method of the LinkedBinarySearchTree<T> class

List	LinkedBinarySearchTree
void add(T element)	void addElement(T element)
T removeFirst()	T removeMin()
T removeLast()	T removeMax()
T remove(T element)	T removeElement(T element)
T first()	T findMin()
T last()	T findMax()
Iterator<T> iterator()	{return iteratorInOrder();}

Using Binary Search Trees: Implementing Ordered Lists

- Analysis of BinarySearchTreeList class

Method	LinkedList	BinarySearchTreeList
removeFirst	O(1)	O(log n)
removeLast	O(n)	O(log n)
remove	O(n)	O(log n) *
contains	O(n)	O(log n)
isEmpty/size	O(1)	O(1)
add	O(n)	O(log n) *

* Add and remove may cause tree to become unbalanced, but there is an O(log n) operation to rebalance the tree at the correct point