DEPARTMENT OF COMPUTER SCIENCE
Graham Roberts

An Example Binary Tree Class

This example shows how a generic binary tree class can be implemented using linked tree nodes that each hold a value along with left and right references to child nodes. This example is primarily for showing implementation techniques and should not be taken as a definitive binary tree class implementation.

The binary tree can hold objects of any class that implements the standard interface Comparable. An object must be Comparable as the binary tree implementation has to be able to compare objects in order to correctly position them within the tree.

The classes and interfaces are:

- interface Tree<E>
- class BinaryTree<E>
- class BinaryTreeTest (JUnit test class)

binary tree class diagram

The Tree interface declares the basic methods for using a tree, along with a single iterator method. The iterator does not state which iteration order is required. Remember that a tree can be traversed pre-order, post-order, in-order or level-order. This example code permits only one of these traversal orders, determined by the class implementing the tree interface.

```
/**
 * General interface for trees.
 * Copyright (c) 2006
 * Dept. of Computer Science, University College London
 * @author Graham Roberts
 * @version 2.0 01-Mar-06
 */

import java.util.*;

public interface Tree<E>
{
  /**
   * Store an object in the tree. The object must conform to type Comparable
   * in order to be inserted in the correct location. Multiple objects representing the
   * same value can be added.
   *
   * @param obj reference to Comparable object to add.
   */
  void add(E obj);

  /**
   * Determine whether the tree contains an object with the same value as the
   * argument.
   *
   * @param obj reference to Comparable object whose value will be searched for.
   * @return true if the value is found.
   */
  boolean contains(E obj);

  /**
   * Remove an object with a matching value from the tree. If multiple
   * matches are possible, only the first matching object is removed.
   *
   * @param obj Remove an object with a matching value from the tree.
   */
  void remove(E obj);

  /**
   * Return a new tree iterator object.
   *
   * @return new iterator object.
   */
  Iterator<E> iterator();
```

```
}
```

Class BinaryTree providing a straightforward binary tree implementation based on tree nodes holding a value and references to left and right child nodes. The iterator is pre-order.

```
/**
 *  A simple generic binary tree class to demonstrate the basic principles
 *  of implementing a tree data structure. This should not be taken as a production
 *  quality class (see the text book instead).
 *  Copyright (c) 2006
 *  Dept. of Computer Science, University College London
 *  @author Graham Roberts
 *  @version 2.0 01-Mar-06
 */

import java.util.Stack;
import java.util.Iterator;

/**
 * Objects stored in a tree must conform to Comparable so that their values can
 * be compared. The type parameter is constained to conform to Comparable to
 * enforce this.
 */
public class BinaryTree<E extends Comparable<E>> implements Tree<E>
{
  /**
   * A tree is a hierarchical structure of TreeNode objects. root references
   * the first node on the tree.
   */
  private TreeNode<E> root;

  /**
   *  Helper class used to implement tree nodes. As this is a private helper
   *  class it is acceptable to have public instance variables. Instances of
   *  this class are never made available to client code of the tree.
   */
  private static class TreeNode<T extends Comparable<T>>
  {
    /**
     *  Data object reference.
     */
    public T val;

    /**
     *  Left and right child nodes.
     */
    public TreeNode<T> left,right;

    /**
     *  Constructor for TreeNode.
     *
     *@param  val    data object reference
     *@param  left   left child node reference or null
     *@param  right  right child node reference or null
     */
    public TreeNode(T val, TreeNode<T> left, TreeNode<T> right)
    {
      this.val = val;
      this.left = left;
      this.right = right;
    }

    /**
     * Insert an object into the tree.
     *
     * @param obj object to insert into tree.
     */
    public void insert(T obj)
    {
      if (val.compareTo(obj) < 0)
      {
        if (right != null)
        {
          right.insert(obj) ;
        }
        else
        {
          right = new TreeNode<T>(obj,null,null) ;
        }
      }
      else
      {
        if (left != null)
```

```
      {
        left.insert(obj) ;
      }
      else
      {
        left = new TreeNode<T>(obj,null,null) ;
      }
    }
  }

  /**
   * Find an object in the tree. Objects are compared using the compareTo method, so
   * must conform to type Comparable.
   * Two objects are equal if they represent the same value.
   *
   * @param obj Object representing value to find in tree.
   * @return  reference to matching node or null.
   */
  public TreeNode<T> find(T obj)
  {
    int temp = val.compareTo(obj) ;
    if (temp == 0)
    {
      return this ;
    }
    if (temp < 0)
    {
      return (right == null) ? null : right.find(obj) ;
    }
    return (left == null) ? null : left.find(obj) ;
  }

  /**
   * Remove the node referencing an object representing the same value as the argument object.
   * This recursive method essentially restructures the tree as necessary and returns a
   * reference to the new root. The algorithm is straightforward apart from the case
   * where the node to be removed has two children. In that case the left-most leaf node
   * of the right child is moved up the tree to replace the removed node. Hand work some
   * examples to see how this works.
   *
   * @param obj Object representing value to remove from tree.
   * @param t Root node of the sub-tree currently being examined (possibly null).
   * @return reference to the (possibly new) root node of the sub-tree being examined or
   * null if no node.
   */
  private TreeNode<T> remove(T obj, TreeNode<T> t)
  {
    if (t == null)
    {
      return t;
    }
    if (obj.compareTo(t.val) < 0)
    {
      t.left = remove(obj,t.left);
    }
    else
    if (obj.compareTo(t.val) > 0 )
    {
      t.right = remove(obj, t.right);
    }
    else
    if (t.left != null && t.right != null)
    {
      t.val = findMin(t.right).val;
      t.right = remove(t.val,t.right);
    }
    else
    {
      t = (t.left != null) ? t.left : t.right;
    }
    return t;
  }

  /**
   * Helper method to find the left most leaf node in a sub-tree.
   *
   * @param t TreeNode to be examined.
   * @return reference to left most leaf node or null.
   */
  private TreeNode<T> findMin(TreeNode<T> t)
  {
    if(t == null)
    {
      return null;
    }
    else
```

```
      if(t.left == null)
      {
        return t;
      }
      return findMin(t.left);
    }
  }

  /**
   * Construct an empty tree.
   */
  public BinaryTree()
  {
    root = null ;
  }

  /**
   * Store an object in the tree. The object must conform to type Comparable
   * in order to be inserted in the correct location. Multiple objects representing the
   * same value can be added.
   *
   * @param obj reference to Comparable object to add.
   */
  public void add(E obj)
  {
    if (root == null)
    {
      root = new TreeNode<E>(obj,null,null) ;
    }
    else
    {
      root.insert(obj) ;
    }
  }

  /**
   * Determine whether the tree contains an object with the same value as the
   * argument.
   *
   * @param obj reference to Comparable object whose value will be searched for.
   * @return true if the value is found.
   */
  public boolean contains(E obj)
  {
    if (root == null)
    {
      return false ;
    }
    else
    {
      return (root.find(obj) != null) ;
    }
  }

  /**
   * Remove an object with a matching value from the tree. If multiple
   * matches are possible, only the first matching object is removed.
   *
   * @param obj Remove an object with a matching value from the tree.
   */
  public void remove(E obj)
  {
    if (root != null)
    {
      root = root.remove(obj,root) ;
    }
  }

  /**
   * Simple pre-order iterator class. An iterator object will sequence through
   * the tree contents in ascending order.
   * A stack is used to keep track of where the iteration has reached in the tree.
   * Note that if new items are added or removed during an iteration, there is no
   * guarantee that the iteration will continue correctly.
   */
  private class PreOrderIterator implements Iterator<E>
  {
    private Stack<TreeNode<E>> nodes = new Stack<TreeNode<E>>() ;

    /**
     * Construct a new iterator for the current tree object.
     */
    public PreOrderIterator()
    {
      pushLeft(root) ;
    }
```

```
    /**
     * Get next obnject in sequence.
     *
     * @return next object in sequence or null if the end of the sequence has
     * been reached.
     */
    public E next()
    {
      if (nodes.isEmpty())
      {
        return null ;
      }
      TreeNode<E> node = nodes.pop() ;
      pushLeft(node.right) ;
      return node.val ;
    }

    /**
     * Determine if there is another object in the iteration sequence.
     *
     * @return true if another object is available in the sequence.
     */
    public boolean hasNext()
    {
      return !nodes.isEmpty() ;
    }

    /**
     * The remove operation is not supported by this iterator. This illustrates
     * that a method required by an implemented interface can be written to not
     * support the operation but should throw an exception if called.
     * UnsupportedOperationException is a subclass of RuntimeException and is
     * not required to be caught at runtime, so the remove method does not
     * have a throws declaration. Calling methods do not have to use a try/catch
     * block pair.
     *
     * @throws UnsupportedOperationException if method is called.
     */
    public void remove()
    {
      throw new UnsupportedOperationException();
    }

    /**
     * Helper method used to push node objects onto the stack to keep
     * track of the iteration.
     */
    private void pushLeft(TreeNode<E> node)
    {
      while (node != null)
      {
        nodes.push(node) ;
        node = node.left ;
      }
    }
  }

  /**
   * Return a new tree iterator object.
   *
   * @return new iterator object.
   */
  public Iterator<E> iterator()
  {
    return new PreOrderIterator() ;
  }
}
```

The JUnit test class for the Binary Tree class.

```
/*
 * JUnit test class for BinaryTree class
 * Copyright (c) 2006
 * Dept. of Computer Science, University College London
 * @author Graham Roberts
 * @version 2.0 01-Mar-06
 */

import junit.framework.* ;
import java.util.Iterator;

public class BinaryTreeTest extends TestCase
{
```

```
  private Tree<Integer> empty ;
  private Tree<Integer> one ;
  private Tree<Integer> several ;

  public void setUp()
  {
    empty = new BinaryTree<Integer>() ;
    one = new BinaryTree<Integer>() ;
    one.add(0) ;
    several = new BinaryTree<Integer>() ;
    several.add(5) ;
    several.add(2) ;
    several.add(1) ;
    several.add(9) ;
    several.add(8) ;
    several.add(10) ;
  }

  public void testEmptyContainsZeroItems()
  {
    assertTreeEmpty(empty);
  }

  public void testOneContainsOneItem()
  {
    assertTrue("One should contain 0",one.contains(0)) ;
    assertIterationValid(one,new int[]{0});
  }

  public void testSeveralContainsSixItems()
  {
    assertContains(several,new int[]{1,2,5,8,9,10}) ;
    assertIterationValid(several,new int[]{1,2,5,8,9,10});
  }

  public void testSeveralDoesNotContain()
  {
    assertDoesNotContain(several,new int[]{-1,0,3,4,6,7,11}) ;
  }

  public void testRemoveFromEmpty()
  {
    empty.remove(0);
    assertTreeEmpty(empty);
  }

  public void testRemoveFromOne()
  {
    one.remove(0) ;
    assertTrue("0 not removed from one",!one.contains(0)) ;
    assertTreeEmpty(one);
  }

  public void testRemoveByLeaf()
  {
    assertRemoveAll(several,new int[]{5,2,1,8,10,9,5});
  }

  public void testRemoveByRoot()
  {
    assertRemoveAll(several,new int[]{5,8,9,10,2,1});
  }

  public void testDuplicates()
  {
    empty.add(1) ;
    empty.add(1) ;
    empty.add(1) ;
    assertIterationValid(empty,new int[] {1,1,1});
    assertTrue("Should contain 1",empty.contains(1)) ;
    empty.remove(1) ;
    assertTrue("Should still contain 1",empty.contains(1)) ;
    assertIterationValid(empty,new int[] {1,1});
    empty.remove(1) ;
    assertTrue("Should still contain 1",empty.contains(1)) ;
    assertIterationValid(empty,new int[] {1});
    empty.remove(1) ;
    assertTrue("Should not contain 1",!empty.contains(1)) ;
    assertTreeEmpty(empty);
  }

  private void assertTreeEmpty(Tree<Integer> tree)
  {
    Iterator<Integer> iterator = tree.iterator() ;
    assertTrue("Tree not empty",!iterator.hasNext()) ;
  }
```

```
  private void assertRemoveAll(Tree<Integer> tree, int[] elements)
  {
    for (int i = 0 ; i < elements.length ; i++)
    {
      tree.remove(elements[i]);
      assertFalse(elements[i] + " Still in tree after being removed",
                  tree.contains(elements[i])) ;
    }
    assertTreeEmpty(tree);
  }

  private void assertContains(Tree<Integer> tree, int[] elements)
  {
    for (int i = 0 ; i < elements.length ; i++)
    {
      assertTrue(elements[i] + " not in tree",
                 tree.contains(elements[i])) ;
    }
  }

  private void assertDoesNotContain(Tree<Integer> tree, int[] elements)
  {
    for (int i = 0 ; i < elements.length ; i++)
    {
      assertFalse(elements[i] + " unexpectedly found in tree",
                  tree.contains(elements[i])) ;
    }
  }

  private void assertIterationValid(Tree<Integer> tree, int[] elements)
  {
    Iterator<Integer> iterator = tree.iterator() ;
    for (int i = 0 ; i < elements.length ; i++)
    {
      assertEquals(elements[i] + " missing from tree",
                   new Integer(elements[i]),iterator.next()) ;
    }
    assertFalse("Not reached end of iteration",iterator.hasNext());
  }

  public static Test suite()
  {
    return new TestSuite(BinaryTreeTest.class) ;
  }

  public static void main (String[] args)
  {
    junit.textui.TestRunner.run(suite()) ;
  }
}
```