```
/******************************************************************************
 *  Compilation:  javac RedBlackLiteBST.java
 *  Execution:    java RedBlackLiteBST < input.txt
 *  Dependencies: StdIn.java StdOut.java
 *  Data files:   http://algs4.cs.princeton.edu/33balanced/tinyST.txt
 *
 *  A symbol table implemented using a left-leaning red-black BST.
 *  This is the 2-3 version.
 *
 *  This implementation implements only put, get, and contains.
 *  See RedBlackBST.java for a full implementation including delete.
 *
 *
 *  % more tinyST.txt
 *  S E A R C H E X A M P L E
 *
 *  % java RedBlackLiteBST < tinyST.txt
 *  A 8
 *  C 4
 *  E 12
 *  H 5
 *  L 11
 *  M 9
 *  P 10
 *  R 3
 *  S 0
 *  X 7
 *
 ******************************************************************************/

public class RedBlackLiteBST<Key extends Comparable<Key>, Value> {

    private static final boolean RED   = true;
    private static final boolean BLACK = false;

    private Node root;     // root of the BST
    private int N;         // number of key-value pairs in BST

    // BST helper node data type
    private class Node {
        private Key key;           // key
        private Value val;         // associated data
        private Node left, right;  // links to left and right subtrees
        private boolean color;     // color of parent link

        public Node(Key key, Value val, boolean color) {
            this.key = key;
            this.val = val;
            this.color = color;
        }
    }

   /***************************************************************************
    *  Standard BST search
    ***************************************************************************/

    // return value associated with the given key, or null if no such key exists
    public Value get(Key key) { return get(root, key); }
    public Value get(Node x, Key key) {
        while (x != null) {
            int cmp = key.compareTo(x.key);
            if      (cmp < 0) x = x.left;
            else if (cmp > 0) x = x.right;
            else              return x.val;
```

```
        }
        return null;
    }

    // is there a key-value pair in the symbol table with the given key?
    public boolean contains(Key key) {
        return (get(key) != null);
    }


   /***************************************************************************
    *  Red-black insertion
    ***************************************************************************/

    public void put(Key key, Value val) {
        root = insert(root, key, val);
        root.color = BLACK;
        assert check();
    }

    private Node insert(Node h, Key key, Value val) {
        if (h == null) {
            N++;
            return new Node(key, val, RED);
        }

        int cmp = key.compareTo(h.key);
        if      (cmp < 0) h.left  = insert(h.left,  key, val);
        else if (cmp > 0) h.right = insert(h.right, key, val);
        else              h.val   = val;

        // fix-up any right-leaning links
        if (isRed(h.right) && !isRed(h.left))      h = rotateLeft(h);
        if (isRed(h.left)  &&  isRed(h.left.left)) h = rotateRight(h);
        if (isRed(h.left)  &&  isRed(h.right))     flipColors(h);

        return h;
    }

   /***************************************************************************
    *  red-black tree helper functions
    ***************************************************************************/

    // is node x red (and non-null) ?
    private boolean isRed(Node x) {
        if (x == null) return false;
        return (x.color == RED);
    }

    // rotate right
    private Node rotateRight(Node h) {
        assert (h != null) && isRed(h.left);
        Node x = h.left;
        h.left = x.right;
        x.right = h;
        x.color = h.color;
        h.color = RED;
        return x;
    }

    // rotate left
    private Node rotateLeft(Node h) {
        assert (h != null) && isRed(h.right);
        Node x = h.right;
```

```
        h.right = x.left;
        x.left = h;
        x.color = h.color;
        h.color = RED;
        return x;
    }

    // precondition: two children are red, node is black
    // postcondition: two children are black, node is red
    private void flipColors(Node h) {
        assert !isRed(h) && isRed(h.left) && isRed(h.right);
        h.color = RED;
        h.left.color = BLACK;
        h.right.color = BLACK;
    }


   /***************************************************************************
    *  Utility functions
    ***************************************************************************/
    // return number of key-value pairs in symbol table
    public int size() {
        return N;
    }

    // is the symbol table empty?
    public boolean isEmpty() {
        return N == 0;
    }

    // height of tree (empty tree height = 0)
    public int height() { return height(root); }
    private int height(Node x) {
        if (x == null) return 0;
        return 1 + Math.max(height(x.left), height(x.right));
    }

    // return the smallest key; null if no such key
    public Key min() { return min(root); }
    private Key min(Node x) {
        Key key = null;
        while (x != null) {
            key = x.key;
            x = x.left;
        }
        return key;
    }

    // return the largest key; null if no such key
    public Key max() { return max(root); }
    private Key max(Node x) {
        Key key = null;
        while (x != null) {
            key = x.key;
            x = x.right;
        }
        return key;
    }


   /***************************************************************************
    *  Iterate using an inorder traversal.
    *  Iterating through N elements takes O(N) time.
    ***************************************************************************/
```

```
    public Iterable<Key> keys() {
        Queue<Key> queue = new Queue<Key>();
        keys(root, queue);
        return queue;
    }

    private void keys(Node x, Queue<Key> queue) {
        if (x == null) return;
        keys(x.left, queue);
        queue.enqueue(x.key);
        keys(x.right, queue);
    }


   /***************************************************************************
    *  Check integrity of red-black BST data structure
    ***************************************************************************/
    private boolean check() {
        if (!isBST())            StdOut.println("Not in symmetric order");
        if (!is23())             StdOut.println("Not a 2-3 tree");
        if (!isBalanced())       StdOut.println("Not balanced");
        return isBST() && is23() && isBalanced();
    }

    // does this binary tree satisfy symmetric order?
    // Note: this test also ensures that data structure is a binary tree since order is
strict
    private boolean isBST() {
        return isBST(root, null, null);
    }

    // is the tree rooted at x a BST with all keys strictly between min and max
    // (if min or max is null, treat as empty constraint)
    // Credit: Bob Dondero's elegant solution
    private boolean isBST(Node x, Key min, Key max) {
        if (x == null) return true;
        if (min != null && x.key.compareTo(min) <= 0) return false;
        if (max != null && x.key.compareTo(max) >= 0) return false;
        return isBST(x.left, min, x.key) && isBST(x.right, x.key, max);
    }

    // Does the tree have no red right links, and at most one (left)
    // red links in a row on any path?
    private boolean is23() { return is23(root); }
    private boolean is23(Node x) {
        if (x == null) return true;
        if (isRed(x.right)) return false;
        if (x != root && isRed(x) && isRed(x.left))
            return false;
        return is23(x.left) && is23(x.right);
    }

    // do all paths from root to leaf have same number of black edges?
    private boolean isBalanced() {
        int black = 0;     // number of black links on path from root to min
        Node x = root;
        while (x != null) {
            if (!isRed(x)) black++;
            x = x.left;
        }
        return isBalanced(root, black);
    }

    // does every path from the root to a leaf have the given number of black links?
```

```
    private boolean isBalanced(Node x, int black) {
        if (x == null) return black == 0;
        if (!isRed(x)) black--;
        return isBalanced(x.left, black) && isBalanced(x.right, black);
    }


   /***************************************************************************
    *  Test client
    ***************************************************************************/
    public static void main(String[] args) {

        String test = "S E A R C H E X A M P L E";
        String[] keys = test.split(" ");
        RedBlackLiteBST<String, Integer> st = new RedBlackLiteBST<String, Integer>();
        for (int i = 0; i < keys.length; i++)
            st.put(keys[i], i);

        StdOut.println("size = " + st.size());
        StdOut.println("min  = " + st.min());
        StdOut.println("max  = " + st.max());
        StdOut.println();


        // print keys in order using allKeys()
        StdOut.println("Testing keys()");
        StdOut.println("--------------------------------");
        for (String s : st.keys())
            StdOut.println(s + " " + st.get(s));
        StdOut.println();

        // insert N elements in order if one command-line argument supplied
        if (args.length == 0) return;
        int N = Integer.parseInt(args[0]);
        RedBlackLiteBST<Integer, Integer> st2 = new RedBlackLiteBST<Integer, Integer>()
;
        for (int i = 0; i < N; i++) {
            st2.put(i, i);
            int h = st2.height();
            StdOut.println("i = " + i + ", height = " + h + ", size = " + st2.size());
        }


        StdOut.println("size = " + st2.size());
    }
}
```