```
/******************************************************************************
 *  Compilation:  javac RedBlackBST.java
 *  Execution:    java RedBlackBST < 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.
 *
 *  % more tinyST.txt
 *  S E A R C H E X A M P L E
 *
 *  % java RedBlackBST < tinyST.txt
 *  A 8
 *  C 4
 *  E 12
 *  H 5
 *  L 11
 *  M 9
 *  P 10
 *  R 3
 *  S 0
 *  X 7
 *
 ******************************************************************************/

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

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

    private Node root;     // root of the 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
        private int N;             // subtree count

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

   /***************************************************************************
    *  Node helper methods
    ***************************************************************************/
    // is node x red; false if x is null ?
    private boolean isRed(Node x) {
        if (x == null) return false;
        return (x.color == RED);
    }

    // number of node in subtree rooted at x; 0 if x is null
    private int size(Node x) {
        if (x == null) return 0;
        return x.N;
    }
```

```
/***************************************************************************
 *  Size methods
 ***************************************************************************/

 // return number of key-value pairs in this symbol table
 public int size() { return size(root); }

 // is this symbol table empty?
 public boolean isEmpty() {
     return root == null;
 }

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

 // value associated with the given key; null if no such key
 public Value get(Key key) { return get(root, key); }

 // value associated with the given key in subtree rooted at x; null if no such key
 private 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 with the given key?
 public boolean contains(Key key) {
     return (get(key) != null);
 }

 // is there a key-value pair with the given key in the subtree rooted at x?
 private boolean contains(Node x, Key key) {
     return (get(x, key) != null);
 }

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

 // insert the key-value pair; overwrite the old value with the new value
 // if the key is already present
 public void put(Key key, Value val) {
     root = put(root, key, val);
     root.color = BLACK;
     assert check();
 }

 // insert the key-value pair in the subtree rooted at h
 private Node put(Node h, Key key, Value val) {
     if (h == null) return new Node(key, val, RED, 1);

     int cmp = key.compareTo(h.key);
     if      (cmp < 0) h.left  = put(h.left,  key, val);
     else if (cmp > 0) h.right = put(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);
        h.N = size(h.left) + size(h.right) + 1;

        return h;
    }

   /***************************************************************************
    *  Red-black deletion
    ***************************************************************************/

    // delete the key-value pair with the minimum key
    public void deleteMin() {
        if (isEmpty()) throw new RuntimeException("BST underflow");

        // if both children of root are black, set root to red
        if (!isRed(root.left) && !isRed(root.right))
            root.color = RED;

        root = deleteMin(root);
        if (!isEmpty()) root.color = BLACK;
        assert check();
    }

    // delete the key-value pair with the minimum key rooted at h
    private Node deleteMin(Node h) {
        if (h.left == null)
            return null;

        if (!isRed(h.left) && !isRed(h.left.left))
            h = moveRedLeft(h);

        h.left = deleteMin(h.left);
        return balance(h);
    }


    // delete the key-value pair with the maximum key
    public void deleteMax() {
        if (isEmpty()) throw new RuntimeException("BST underflow");

        // if both children of root are black, set root to red
        if (!isRed(root.left) && !isRed(root.right))
            root.color = RED;

        root = deleteMax(root);
        if (!isEmpty()) root.color = BLACK;
        assert check();
    }

    // delete the key-value pair with the maximum key rooted at h
    private Node deleteMax(Node h) {
        if (isRed(h.left))
            h = rotateRight(h);

        if (h.right == null)
            return null;

        if (!isRed(h.right) && !isRed(h.right.left))
            h = moveRedRight(h);

        h.right = deleteMax(h.right);

        return balance(h);
```

```
    }

    // delete the key-value pair with the given key
    public void delete(Key key) {
        if (!contains(key)) {
            System.err.println("symbol table does not contain " + key);
            return;
        }

        // if both children of root are black, set root to red
        if (!isRed(root.left) && !isRed(root.right))
            root.color = RED;

        root = delete(root, key);
        if (!isEmpty()) root.color = BLACK;
        assert check();
    }

    // delete the key-value pair with the given key rooted at h
    private Node delete(Node h, Key key) {
        assert contains(h, key);

        if (key.compareTo(h.key) < 0)  {
            if (!isRed(h.left) && !isRed(h.left.left))
                h = moveRedLeft(h);
            h.left = delete(h.left, key);
        }
        else {
            if (isRed(h.left))
                h = rotateRight(h);
            if (key.compareTo(h.key) == 0 && (h.right == null))
                return null;
            if (!isRed(h.right) && !isRed(h.right.left))
                h = moveRedRight(h);
            if (key.compareTo(h.key) == 0) {
                h.val = get(h.right, min(h.right).key);
                h.key = min(h.right).key;
                h.right = deleteMin(h.right);
            }
            else h.right = delete(h.right, key);
        }
        return balance(h);
    }

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

    // make a left-leaning link lean to the 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 = x.right.color;
        x.right.color = RED;
        x.N = h.N;
        h.N = size(h.left) + size(h.right) + 1;
        return x;
    }

    // make a right-leaning link lean to the 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 = x.left.color;
        x.left.color = RED;
        x.N = h.N;
        h.N = size(h.left) + size(h.right) + 1;
        return x;
    }

    // flip the colors of a node and its two children
    private void flipColors(Node h) {
        // h must have opposite color of its two children
        assert (h != null) && (h.left != null) && (h.right != null);
        assert (!isRed(h) &&  isRed(h.left) &&  isRed(h.right))
            || (isRed(h)  && !isRed(h.left) && !isRed(h.right));
        h.color = !h.color;
        h.left.color = !h.left.color;
        h.right.color = !h.right.color;
    }

    // Assuming that h is red and both h.left and h.left.left
    // are black, make h.left or one of its children red.
    private Node moveRedLeft(Node h) {
        assert (h != null);
        assert isRed(h) && !isRed(h.left) && !isRed(h.left.left);

        flipColors(h);
        if (isRed(h.right.left)) {
            h.right = rotateRight(h.right);
            h = rotateLeft(h);
            // flipColors(h);
        }
        return h;
    }

    // Assuming that h is red and both h.right and h.right.left
    // are black, make h.right or one of its children red.
    private Node moveRedRight(Node h) {
        assert (h != null);
        assert isRed(h) && !isRed(h.right) && !isRed(h.right.left);
        flipColors(h);
        if (isRed(h.left.left)) {
            h = rotateRight(h);
            // flipColors(h);
        }
        return h;
    }

    // restore red-black tree invariant
    private Node balance(Node h) {
        assert (h != null);

        if (isRed(h.right))                      h = rotateLeft(h);
        if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);
        if (isRed(h.left) && isRed(h.right))     flipColors(h);

        h.N = size(h.left) + size(h.right) + 1;
        return h;
    }


/***************************************************************************
 *  Utility functions
```

```
     ***************************************************************************/

    // height of tree; 0 if empty
    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));
    }

   /***************************************************************************
    *  Ordered symbol table methods.
    ***************************************************************************/

    // the smallest key; null if no such key
    public Key min() {
        if (isEmpty()) return null;
        return min(root).key;
    }

    // the smallest key in subtree rooted at x; null if no such key
    private Node min(Node x) {
        assert x != null;
        if (x.left == null) return x;
        else                return min(x.left);
    }

    // the largest key; null if no such key
    public Key max() {
        if (isEmpty()) return null;
        return max(root).key;
    }

    // the largest key in the subtree rooted at x; null if no such key
    private Node max(Node x) {
        assert x != null;
        if (x.right == null) return x;
        else                 return max(x.right);
    }

    // the largest key less than or equal to the given key
    public Key floor(Key key) {
        Node x = floor(root, key);
        if (x == null) return null;
        else           return x.key;
    }

    // the largest key in the subtree rooted at x less than or equal to the given key
    private Node floor(Node x, Key key) {
        if (x == null) return null;
        int cmp = key.compareTo(x.key);
        if (cmp == 0) return x;
        if (cmp < 0)  return floor(x.left, key);
        Node t = floor(x.right, key);
        if (t != null) return t;
        else           return x;
    }

    // the smallest key greater than or equal to the given key
    public Key ceiling(Key key) {
        Node x = ceiling(root, key);
        if (x == null) return null;
        else           return x.key;
    }
```

```
    // the smallest key in the subtree rooted at x greater than or equal to the given k
ey
    private Node ceiling(Node x, Key key) {
        if (x == null) return null;
        int cmp = key.compareTo(x.key);
        if (cmp == 0) return x;
        if (cmp > 0)  return ceiling(x.right, key);
        Node t = ceiling(x.left, key);
        if (t != null) return t;
        else           return x;
    }


    // the key of rank k
    public Key select(int k) {
        if (k < 0 || k >= size())  return null;
        Node x = select(root, k);
        return x.key;
    }

    // the key of rank k in the subtree rooted at x
    private Node select(Node x, int k) {
        assert x != null;
        assert k >= 0 && k < size(x);
        int t = size(x.left);
        if      (t > k) return select(x.left,  k);
        else if (t < k) return select(x.right, k-t-1);
        else            return x;
    }

    // number of keys less than key
    public int rank(Key key) {
        return rank(key, root);
    }

    // number of keys less than key in the subtree rooted at x
    private int rank(Key key, Node x) {
        if (x == null) return 0;
        int cmp = key.compareTo(x.key);
        if      (cmp < 0) return rank(key, x.left);
        else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);
        else              return size(x.left);
    }

   /***********************************************************************
    *  Range count and range search.
    ***********************************************************************/

    // all of the keys, as an Iterable
    public Iterable<Key> keys() {
        return keys(min(), max());
    }

    // the keys between lo and hi, as an Iterable
    public Iterable<Key> keys(Key lo, Key hi) {
        Queue<Key> queue = new Queue<Key>();
        // if (isEmpty() || lo.compareTo(hi) > 0) return queue;
        keys(root, queue, lo, hi);
        return queue;
    }

    // add the keys between lo and hi in the subtree rooted at x
    // to the queue
    private void keys(Node x, Queue<Key> queue, Key lo, Key hi) {
```

```
        if (x == null) return;
        int cmplo = lo.compareTo(x.key);
        int cmphi = hi.compareTo(x.key);
        if (cmplo < 0) keys(x.left, queue, lo, hi);
        if (cmplo <= 0 && cmphi >= 0) queue.enqueue(x.key);
        if (cmphi > 0) keys(x.right, queue, lo, hi);
    }

    // number keys between lo and hi
    public int size(Key lo, Key hi) {
        if (lo.compareTo(hi) > 0) return 0;
        if (contains(hi)) return rank(hi) - rank(lo) + 1;
        else              return rank(hi) - rank(lo);
    }


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

    // 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);
    }

    // are the size fields correct?
    private boolean isSizeConsistent() { return isSizeConsistent(root); }
    private boolean isSizeConsistent(Node x) {
        if (x == null) return true;
        if (x.N != size(x.left) + size(x.right) + 1) return false;
        return isSizeConsistent(x.left) && isSizeConsistent(x.right);
    }

    // check that ranks are consistent
    private boolean isRankConsistent() {
        for (int i = 0; i < size(); i++)
            if (i != rank(select(i))) return false;
        for (Key key : keys())
            if (key.compareTo(select(rank(key))) != 0) return false;
        return true;
    }

    // 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) {
        RedBlackBST<String, Integer> st = new RedBlackBST<String, Integer>();
        for (int i = 0; !StdIn.isEmpty(); i++) {
            String key = StdIn.readString();
            st.put(key, i);
        }
        for (String s : st.keys())
            StdOut.println(s + " " + st.get(s));
        StdOut.println();
    }
}
```