```
import java.util.ArrayList;

/**
 * @author Li Ersan
 */
public class RBTree<K extends Comparable<K>, V> {

    private static final boolean RED = true;

    private static final boolean BLACK = false;

    private class Node {

        K key;

        V value;

        Node left, right;

        boolean color;

        Node(K key, V value) {
            this.key = key;
            this.value = value;
            this.left = null;
            this.right = null;
            this.color = RED;
        }
    }

    private Node root;

    private int size;

    public RBTree() {
        root = null;
        size = 0;
    }

    /**
     * 判断结点 node 的颜色
     *
     * @param node
     * @return
     */
    private boolean isRed(Node node) {
        if (node == null) {
            return BLACK;
        }

        return node.color;
    }
```

```
/**
 *  左旋转
 * //          node                                    x
 * //        /      \          左旋转                /     \
 * //      T1        x       --------->           node      T3
 * //              /    \                        /     \
 * //            T2      T3                     T1      T2
 *
 * @param node
 * @return
 */
private Node leftRotate(Node node) {
    Node x = node.right;

    //左旋转
    node.right = x.left;
    x.left = node;
    x.color = node.color;
    node.color = RED;

    return x;
}


/**
 *  右旋转
 * //          node                                    x
 * //        /      \          右旋转                /     \
 * //      x         T2      --------->            y       node
 * //   /    \                                            /     \
 * // y      T1                                         T1      T2
 *
 * @param node
 * @return
 */
private Node rightRotate(Node node) {

    Node x = node.left;

    //右旋转
    node.left = x.right;
    x.right = node;

    x.color = node.color;
    node.color = RED;

    return x;
}

/**
 *  颜色翻转
 *
 * @param node
 */
```

```
private void flipColors(Node node) {
    node.color = RED;
    node.left.color = BLACK;
    node.right.color = BLACK;
}

/**
 * 向红黑树树中添加新的元素（key, value）
 *
 * @param key
 * @param value
 */
public void add(K key, V value) {
    root = add(root, key, value);
    root.color = BLACK; //最终根结点为黑黑色结点
}


/**
 * 向以 node 为根的红黑树中插入元素(key, value)，递归算法
 * 返回插入新结点后红黑树的根
 *
 * @param node
 * @param key
 * @param value
 * @return
 */
private Node add(Node node, K key, V value) {

    if (node == null) {
        size++;
        return new Node(key, value); //默认插入红色结点
    }

    if (key.compareTo(node.key) < 0) {
        node.left = add(node.left, key, value);
    } else if (key.compareTo(node.key) > 0) {
        node.right = add(node.right, key, value);
    } else {
        // key.compareTo(node.key) == 0
        node.value = value;
    }

    if (isRed(node.right) && !isRed(node.left)) {
        node = leftRotate(node);
    }

    if (isRed(node.left) && isRed(node.left.left)) {
        node = rightRotate(node);
    }

    if (isRed(node.left) && isRed(node.right)) {
        flipColors(node);
    }
```

```
        return node;
    }

    /**
     * 返回以 node 为根结点的二分搜索树中，key 所在的结点
     *
     * @param node
     * @param key
     * @return
     */
    private Node getNode(Node node, K key) {

        if (node == null) {
            return null;
        }

        if (key.compareTo(node.key) == 0) {
            return node;
        } else if (key.compareTo(node.key) < 0) {
            return getNode(node.left, key);
        } else { // if key.compareTo(node.key) > 0
            return getNode(node.right, key);
        }
    }

    /**
     * 返回以 node 为根的二分搜索树的最小键值所在的结点
     *
     * @param node
     */
    private Node minimum(Node node) {

        if (node.left == null) {
            return node;
        }
        return minimum(node.left);
    }

    /**
     * 删除以 node 为根的二分搜索树中的最小结点
     *
     * @param node
     * @return 返回删除结点后新的二分搜索树的根
     */
    private Node removeMin(Node node) {
        if (node.left == null) {
            Node rightNode = node.right;
            node.right = null;
            size--;
            return rightNode;
        }

        node.left = removeMin(node.left);
        return node;
    }

    /**
     * 从二分搜索树中删除键值为 key 的结点
     *
     * @param key
     * @return
     */
    public V remove(K key) {
        Node node = getNode(root, key);
        if (node != null) {
            root = remove(root, key);
            return node.value;
        }

        return null;
    }

    /**
     * 删除掉以 node 为根的二分搜索树中键为 key 的结点，递归算法
     *
     * @param node
     * @param key
     * @return 返回删除结点后新的二分搜索树的根
     */
    private Node remove(Node node, K key) {

        if (node == null) {
            return null;
        }

        if (key.compareTo(node.key) < 0) {
            node.left = remove(node.left, key);
            return node;
        } else if (key.compareTo(node.key) > 0) {
            node.right = remove(node.right, key);
            return node;
        } else {//key.compareTo(node.key) == 0

            //待删除结点左子树为空的情况
            if (node.left == null) {
                Node rightNode = node.right;
                node.right = null;
                size--;
                return rightNode;
            }

            //待删除结点右子树为空的情况
            if (node.right == null) {
                Node leftNode = node.left;
                node.left = null;
                size--;
                return leftNode;
            }

              /*待删除结点左右子树为空的情况
                找到比待删除结点大的最小结点，即待删除结点右子树的最小结点
                用这个结点顶替待删除结点的位置
                */

            Node successor = minimum(node.right);
            successor.right = removeMin(node.right);
            successor.left = node.left;
            node.left = node.right = null;

            return successor;

        }
    }


    public boolean contains(K key) {
        return getNode(root, key) != null;
    }


    public V get(K key) {
        Node node = getNode(root, key);
        return node == null ? null : node.value;
    }


    public void set(K key, V newValue) {
        Node node = getNode(root, key);
        if (node == null) {
            throw new IllegalArgumentException(key + " 不存在！ ");
        }

        node.value = newValue;
    }


    public int getSize() {
        return size;
    }


    public boolean isEmpty() {
        return size == 0;
    }

    public static void main(String[] args) {

        System.out.println("傲慢与偏见");

        ArrayList<String> words = new ArrayList<>();
        if (FileOperation.readFile("pride-and-prejudice.txt", words)) {
            System.out.println("共有单词数： " + words.size());

            RBTree<String, Integer> rbTree = new RBTree<>();
            for (String word : words) {
                if (rbTree.contains(word)) {
                    rbTree.set(word, rbTree.get(word) + 1);
                } else {
                    rbTree.add(word, 1);
                }
            }

            System.out.println("共有不同单词数： " + rbTree.getSize());
            System.out.println("出现 pride 的次数: " + rbTree.get("pride"));
            System.out.println("出现 prejudice 的次数: " + rbTree.get("prejudice"));
        }
    }
}
```