Names:

Nada Maher Mahmoud (56)

Nada Mostafa Rashad (57)

Data Structures 2 - Lab 2

Implementing Red Black tree

and Tree map interface

Problem Statement:

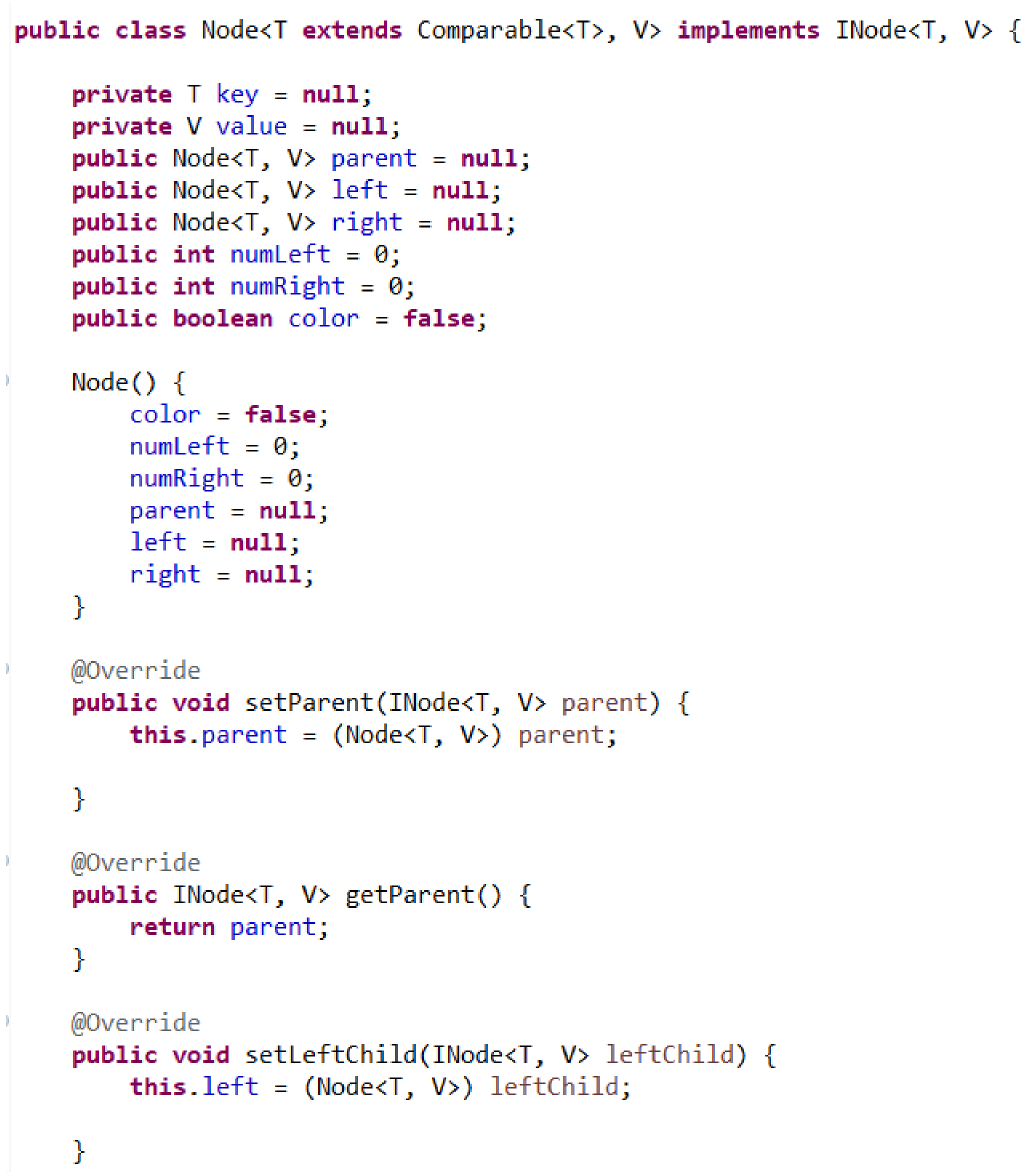
1) Implement Red-Black tree functions (insert, delete, search, clear, get root, is empty).
2) Implement a Tree map similar to java Tree map.
3) Implement Inode interface.
4) Implement IredBlackTree interface.
5) Implement ITrrMap interface.

Assumptions and decisions:

1) We have a nil node that is connected to all leaf nodes in the red black tree.
2) Any node is considered a nil if its key is null.
3) If you insert a new node with a key already exists in the tree, we delete the old node then insert the new one.

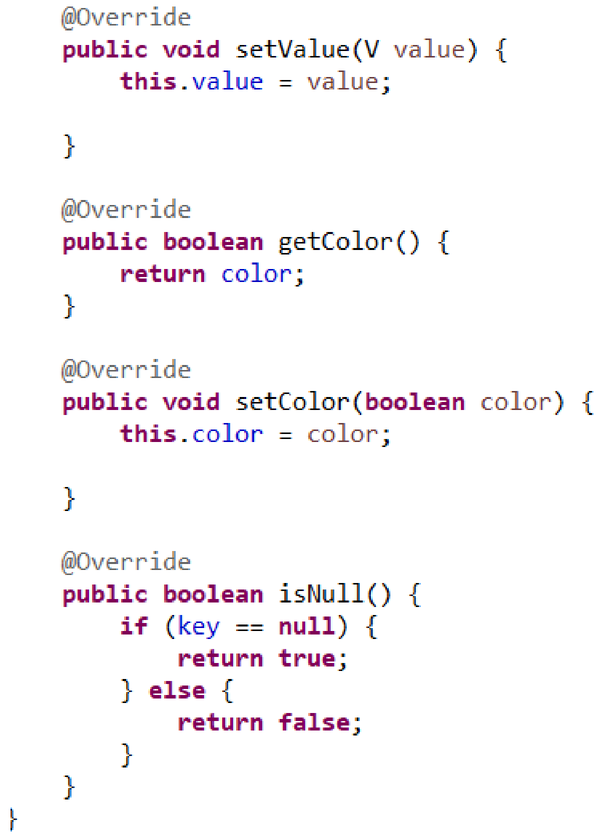
Code Snippets:

INode interface implementation:



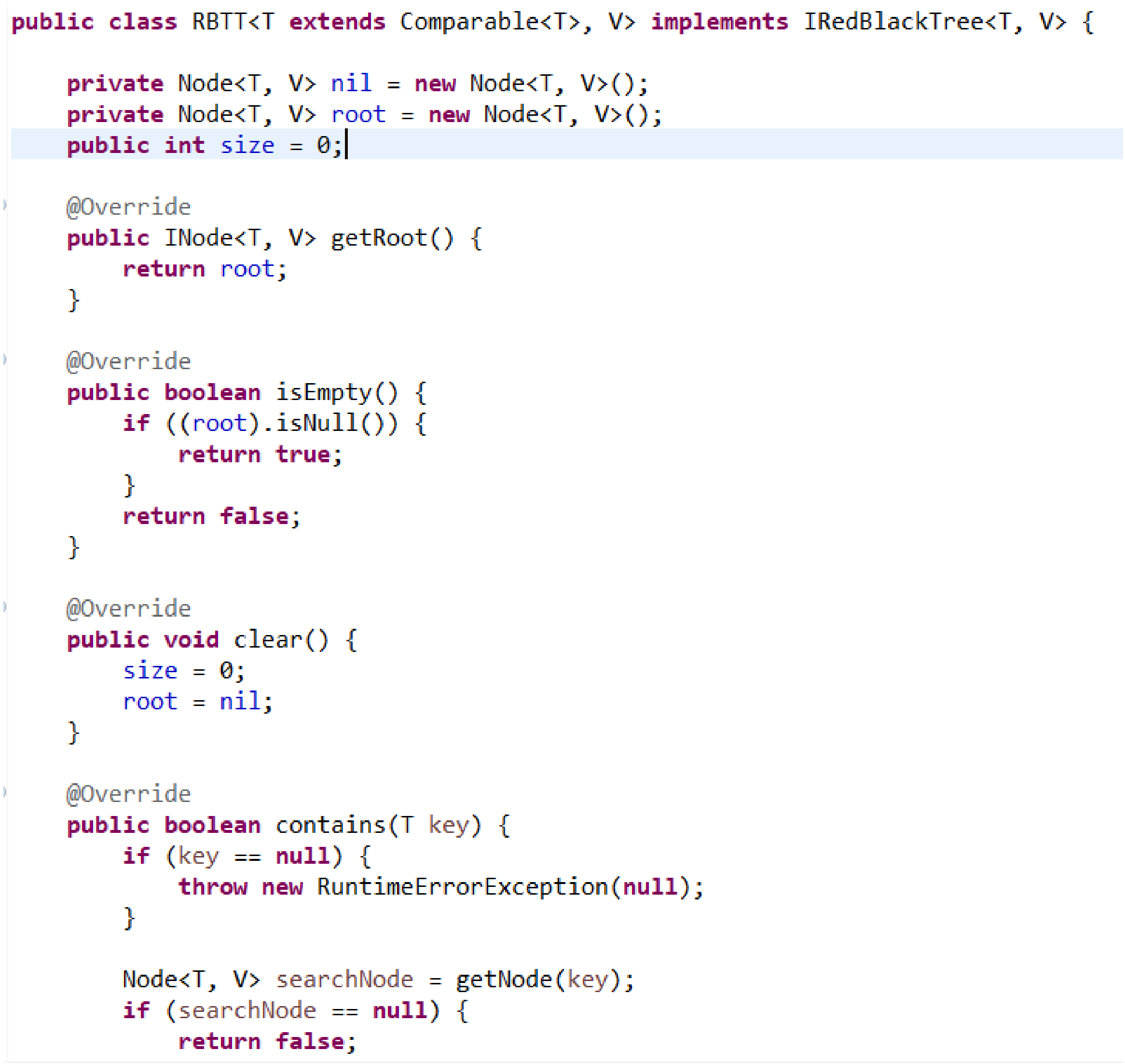
```java
public class Node<T extends Comparable<T>, V> implements INode<T, V> {

    private T key = null;
    private V value = null;
    public Node<T, V> parent = null;
    public Node<T, V> left = null;
    public Node<T, V> right = null;
    public int numLeft = 0;
    public int numRight = 0;
    public boolean color = false;

    Node() {
        color = false;
        numLeft = 0;
        numRight = 0;
        parent = null;
        left = null;
        right = null;
    }

    @Override
    public void setParent(INode<T, V> parent) {
        this.parent = (Node<T, V>) parent;

    }

    @Override
    public INode<T, V> getParent() {
        return parent;
    }

    @Override
    public void setLeftChild(INode<T, V> leftChild) {
        this.left = (Node<T, V>) leftChild;

    }
```

```java
@Override
public INode<T, V> getLeftChild() {
    return left;
}

@Override
public void setRightChild(INode<T, V> rightChild) {
    this.right = (Node<T, V>) rightChild;

}

@Override
public INode<T, V> getRightChild() {
    return right;
}

@Override
public T getKey() {
    return key;
}

@Override
public void setKey(T key) {
    this.key = key;

}

@Override
public V getValue() {
    return value;
}

@Override
public void setValue(V value) {
    this.value = value;

}
```

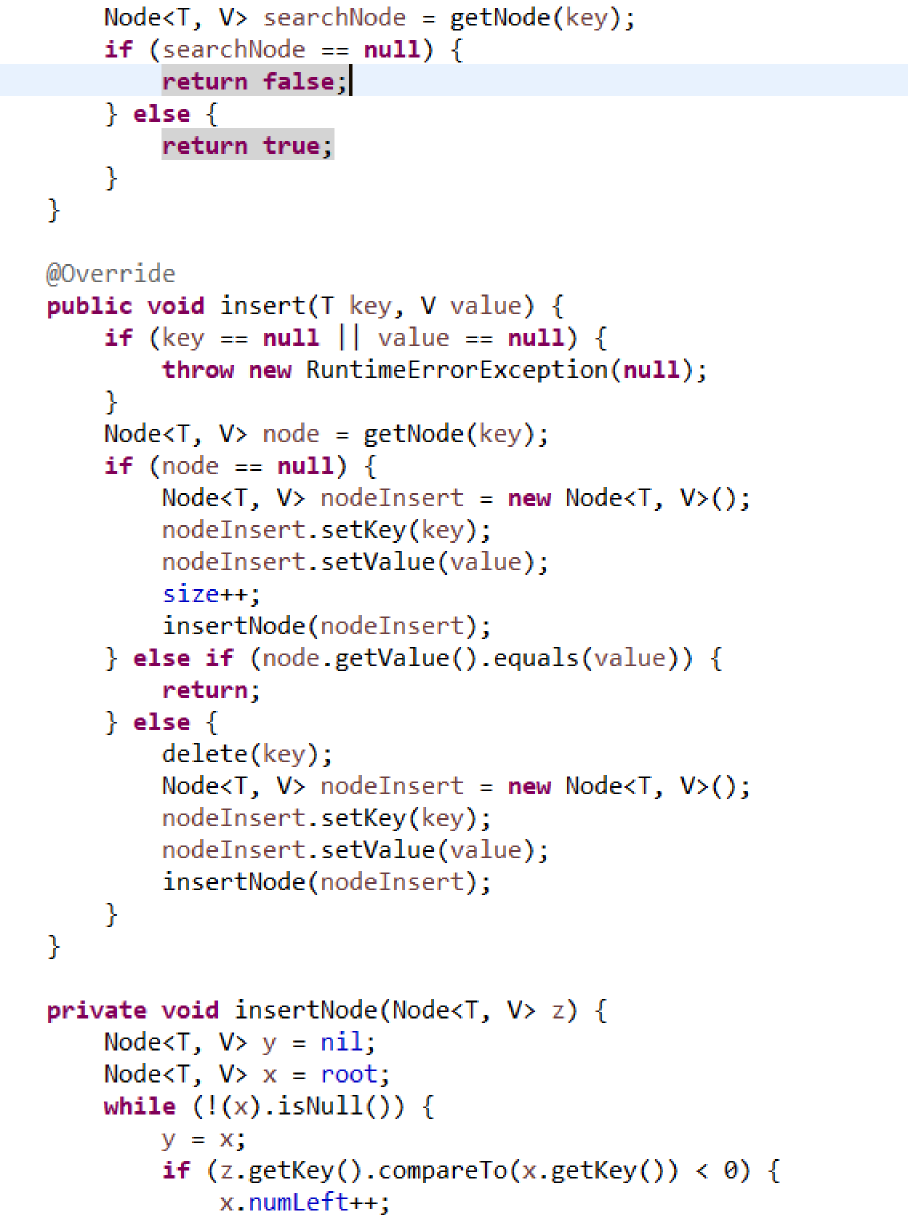


```
    @Override
    public void setValue(V value) {
        this.value = value;

    }

    @Override
    public boolean getColor() {
        return color;
    }

    @Override
    public void setColor(boolean color) {
        this.color = color;

    }

    @Override
    public boolean isNull() {
        if (key == null) {
            return true;
        } else {
            return false;
        }
    }
}
```

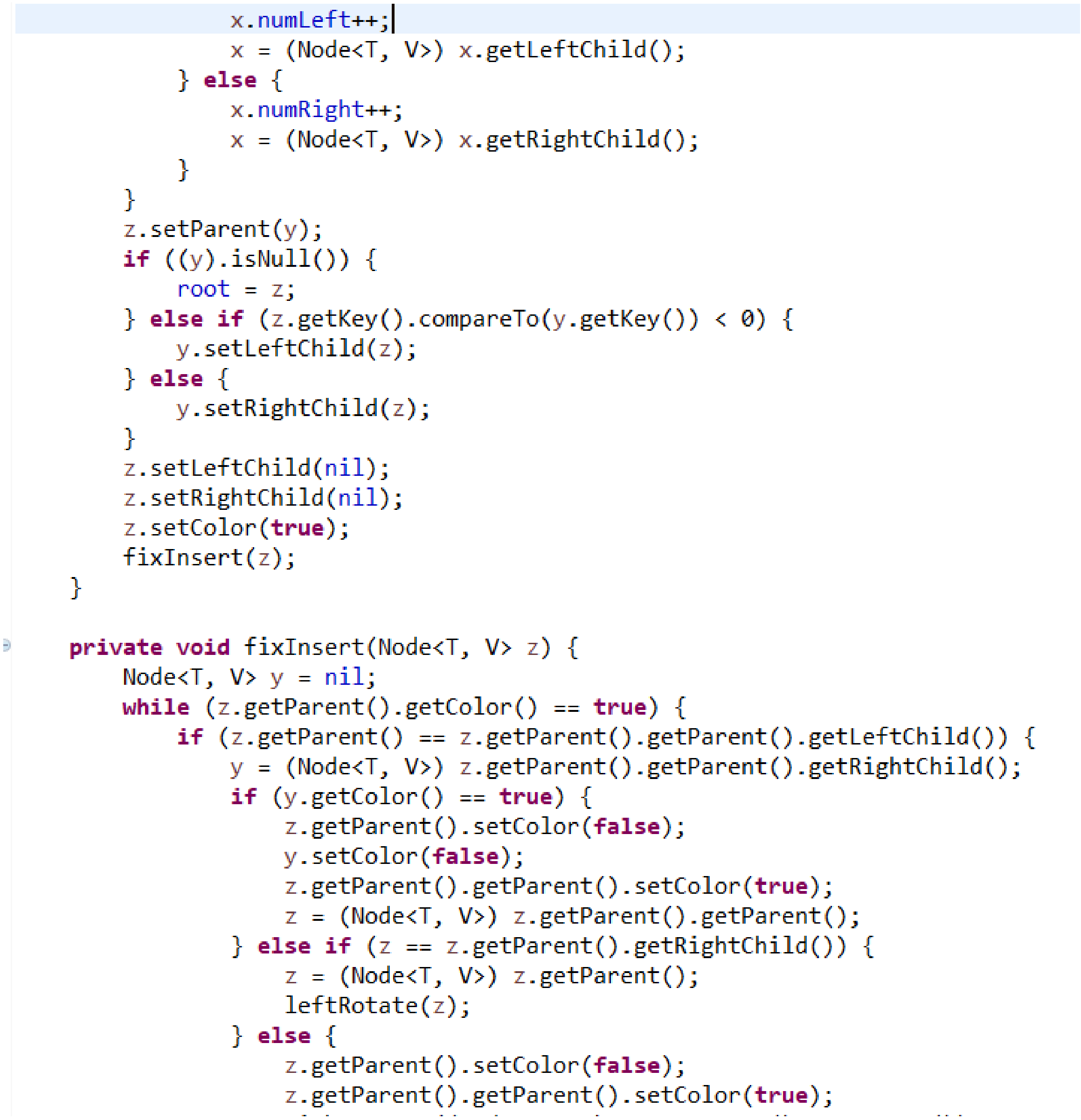
IRedBlackTree interface implementation:



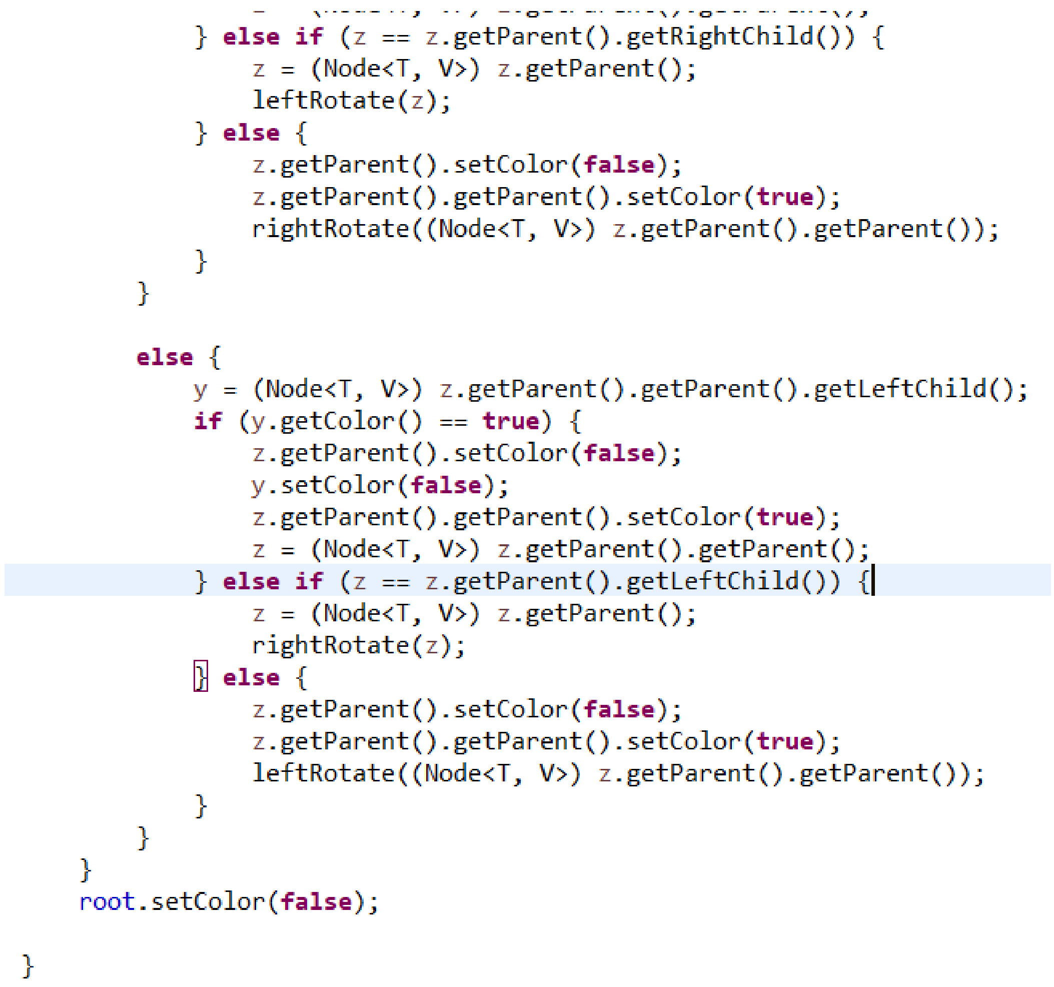
```java
public class RBTT<T extends Comparable<T>, V> implements IRedBlackTree<T, V> {

    private Node<T, V> nil = new Node<T, V>();
    private Node<T, V> root = new Node<T, V>();
    public int size = 0;

    @Override
    public INode<T, V> getRoot() {
        return root;
    }

    @Override
    public boolean isEmpty() {
        if ((root).isNull()) {
            return true;
        }
        return false;
    }

    @Override
    public void clear() {
        size = 0;
        root = nil;
    }

    @Override
    public boolean contains(T key) {
        if (key == null) {
            throw new RuntimeErrorException(null);
        }

        Node<T, V> searchNode = getNode(key);
        if (searchNode == null) {
            return false;
```



```
        Node<T, V> searchNode = getNode(key);
        if (searchNode == null) {
            return false;
        } else {
            return true;
        }
    }

    @Override
    public void insert(T key, V value) {
        if (key == null || value == null) {
            throw new RuntimeErrorException(null);
        }
        Node<T, V> node = getNode(key);
        if (node == null) {
            Node<T, V> nodeInsert = new Node<T, V>();
            nodeInsert.setKey(key);
            nodeInsert.setValue(value);
            size++;
            insertNode(nodeInsert);
        } else if (node.getValue().equals(value)) {
            return;
        } else {
            delete(key);
            Node<T, V> nodeInsert = new Node<T, V>();
            nodeInsert.setKey(key);
            nodeInsert.setValue(value);
            insertNode(nodeInsert);
        }
    }

    private void insertNode(Node<T, V> z) {
        Node<T, V> y = nil;
        Node<T, V> x = root;
        while (!(x).isNull()) {
            y = x;
            if (z.getKey().compareTo(x.getKey()) < 0) {
                x.numLeft++;
```

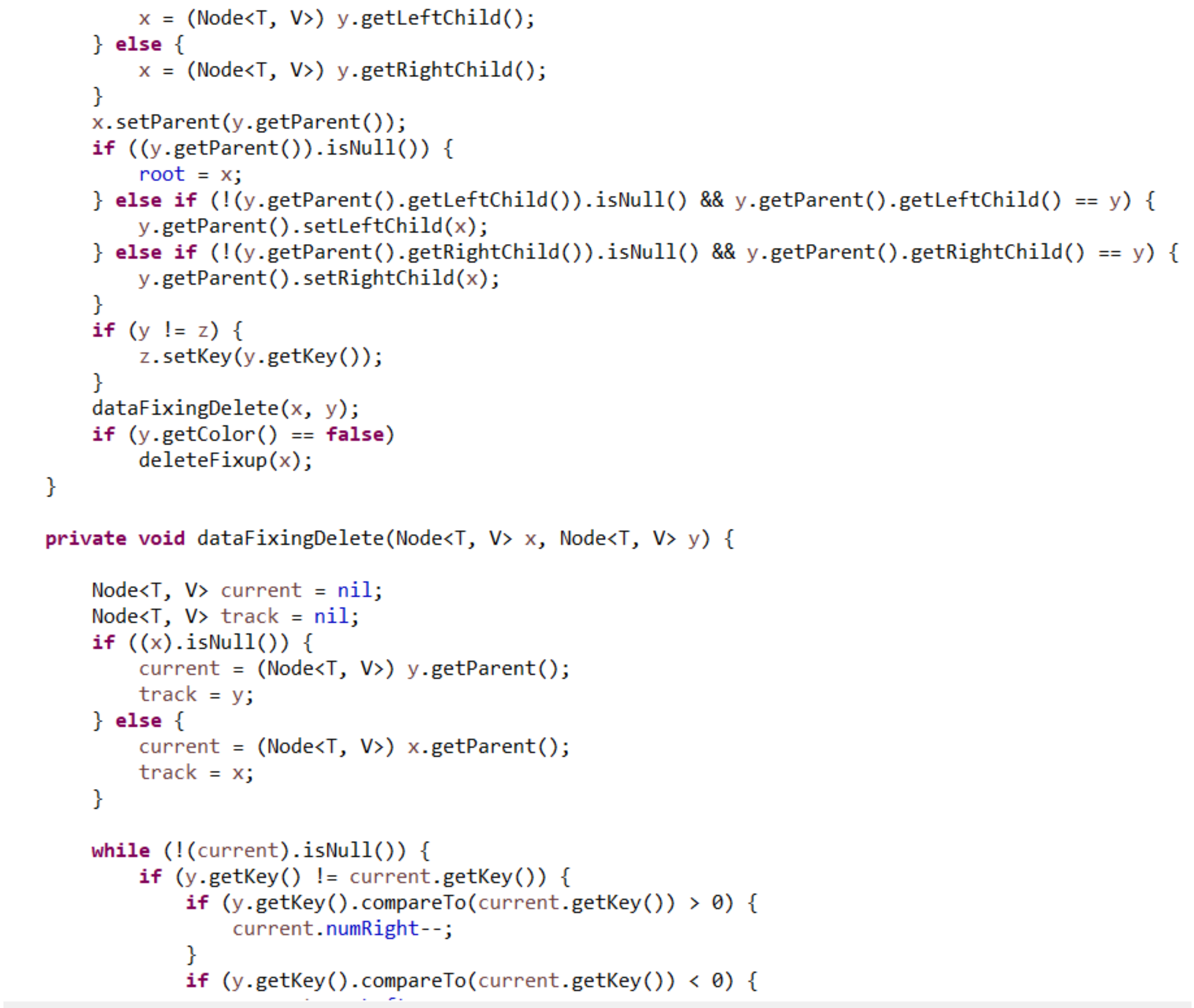


```
                x.numLeft++;
                x = (Node<T, V>) x.getLeftChild();
            } else {
                x.numRight++;
                x = (Node<T, V>) x.getRightChild();
            }
        }
        z.setParent(y);
        if ((y).isNull()) {
            root = z;
        } else if (z.getKey().compareTo(y.getKey()) < 0) {
            y.setLeftChild(z);
        } else {
            y.setRightChild(z);
        }
        z.setLeftChild(nil);
        z.setRightChild(nil);
        z.setColor(true);
        fixInsert(z);
    }

    private void fixInsert(Node<T, V> z) {
        Node<T, V> y = nil;
        while (z.getParent().getColor() == true) {
            if (z.getParent() == z.getParent().getParent().getLeftChild()) {
                y = (Node<T, V>) z.getParent().getParent().getRightChild();
                if (y.getColor() == true) {
                    z.getParent().setColor(false);
                    y.setColor(false);
                    z.getParent().getParent().setColor(true);
                    z = (Node<T, V>) z.getParent().getParent();
                } else if (z == z.getParent().getRightChild()) {
                    z = (Node<T, V>) z.getParent();
                    leftRotate(z);
                } else {
                    z.getParent().setColor(false);
                    z.getParent().getParent().setColor(true);
```

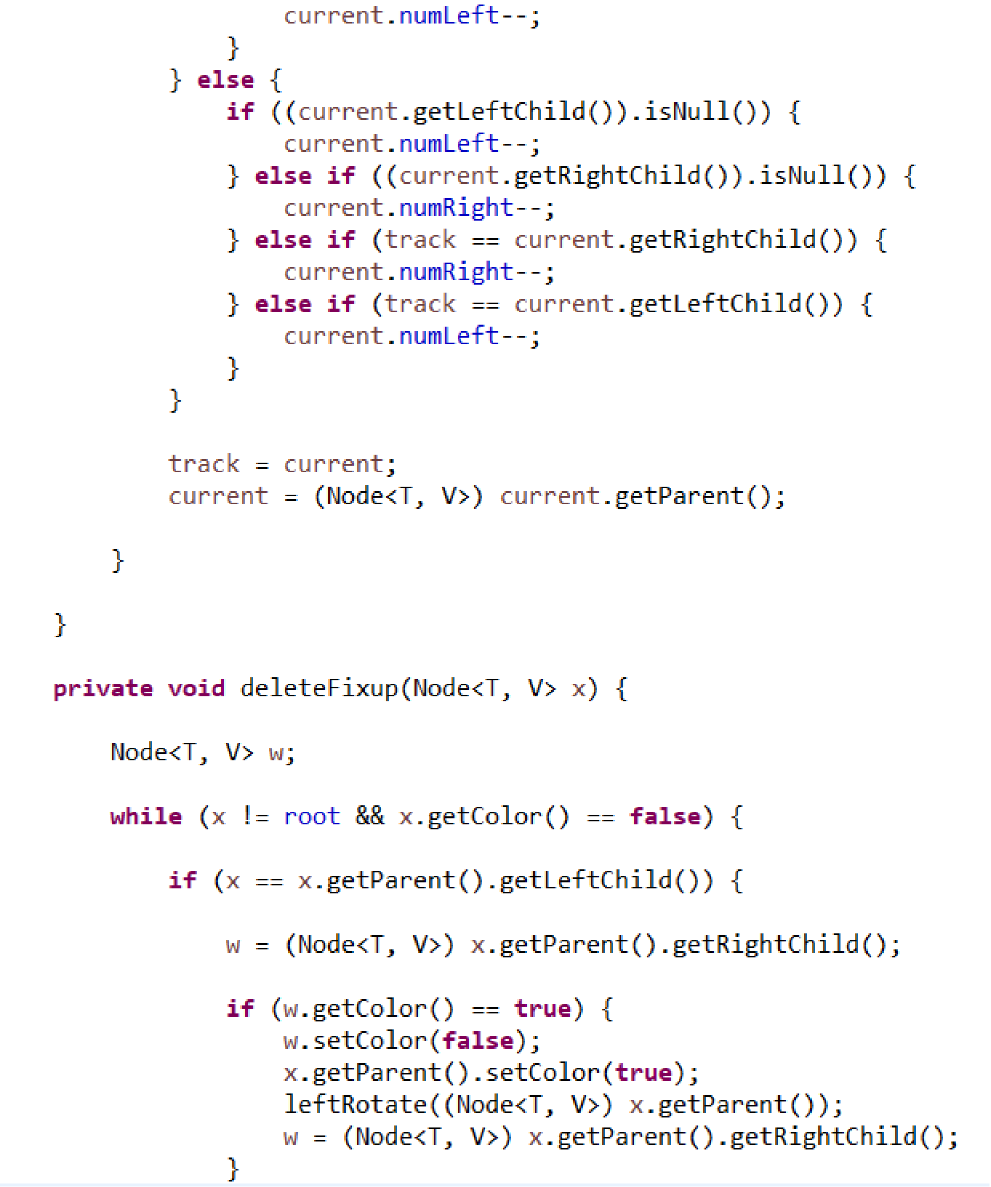


```
                z = (Node<T, V>) z.getParent().getParent();
            } else if (z == z.getParent().getRightChild()) {
                z = (Node<T, V>) z.getParent();
                leftRotate(z);
            } else {
                z.getParent().setColor(false);
                z.getParent().getParent().setColor(true);
                rightRotate((Node<T, V>) z.getParent().getParent());
            }
        }

        else {
            y = (Node<T, V>) z.getParent().getParent().getLeftChild();
            if (y.getColor() == true) {
                z.getParent().setColor(false);
                y.setColor(false);
                z.getParent().getParent().setColor(true);
                z = (Node<T, V>) z.getParent().getParent();
            } else if (z == z.getParent().getLeftChild()) {
                z = (Node<T, V>) z.getParent();
                rightRotate(z);
            } else {
                z.getParent().setColor(false);
                z.getParent().getParent().setColor(true);
                leftRotate((Node<T, V>) z.getParent().getParent());
            }
        }
    }
    root.setColor(false);

}
```

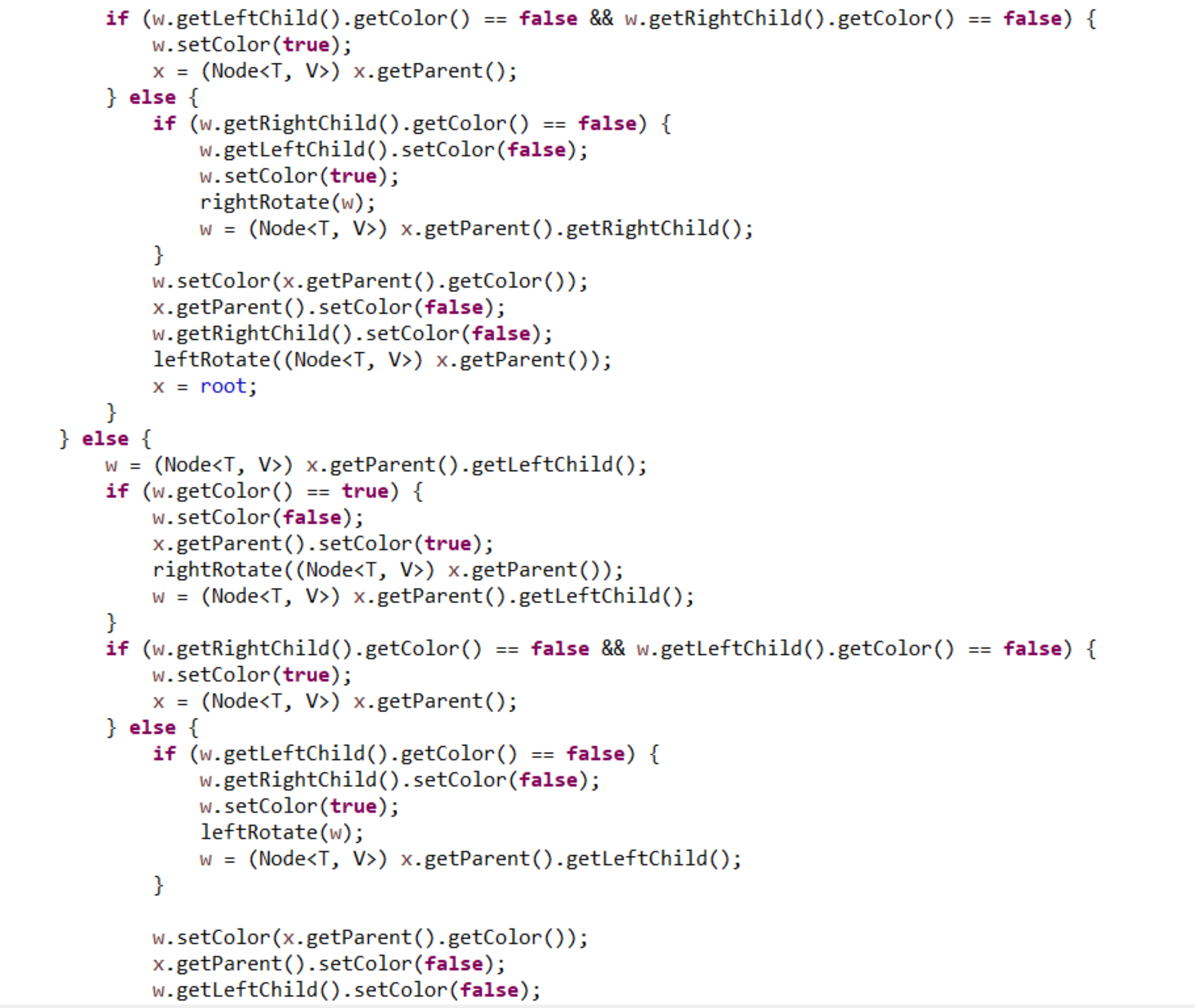
```java
@Override
public boolean delete(T key) {
    if (key == null) {
        throw new RuntimeErrorException(null);
    }
    Node<T, V> nodeDelete = getNode(key);
    if (nodeDelete == null) {
        return false;
    } else {
        deleteRBT(nodeDelete);
        size--;
        if (size == 0) {
            root = new Node<>();
            root.setColor(false);
            root.setKey(null);
            root.setLeftChild(nil);
            root.setRightChild(nil);
            root.setParent(nil);
            root.setValue(null);
        }
        return true;
    }

}

public void deleteRBT(Node<T, V> v) {

    Node<T, V> z = getNode(v.getKey());
    Node<T, V> x = nil;
    Node<T, V> y = nil;
    if ((z.getLeftChild()).isNull() || (z.getRightChild()).isNull()) {
        y = z;
    } else {
        y = treeSuccessor(z);
    }

    if (!(y.getLeftChild()).isNull()) {
```



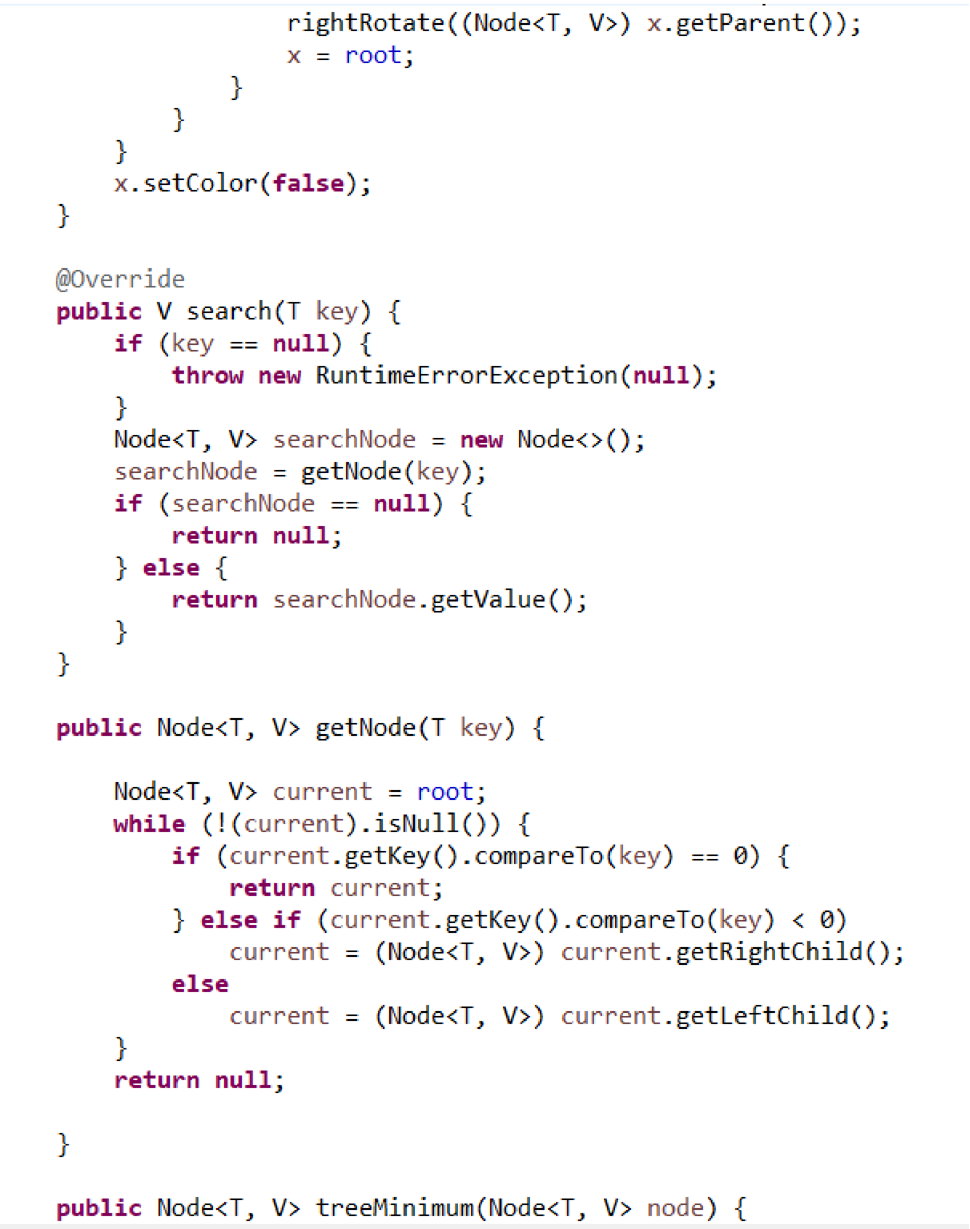
```
            x = (Node<T, V>) y.getLeftChild();
        } else {
            x = (Node<T, V>) y.getRightChild();
        }
        x.setParent(y.getParent());
        if ((y.getParent()).isNull()) {
            root = x;
        } else if (!(y.getParent().getLeftChild()).isNull() && y.getParent().getLeftChild() == y) {
            y.getParent().setLeftChild(x);
        } else if (!(y.getParent().getRightChild()).isNull() && y.getParent().getRightChild() == y) {
            y.getParent().setRightChild(x);
        }
        if (y != z) {
            z.setKey(y.getKey());
        }
        dataFixingDelete(x, y);
        if (y.getColor() == false)
            deleteFixup(x);
    }

    private void dataFixingDelete(Node<T, V> x, Node<T, V> y) {

        Node<T, V> current = nil;
        Node<T, V> track = nil;
        if ((x).isNull()) {
            current = (Node<T, V>) y.getParent();
            track = y;
        } else {
            current = (Node<T, V>) x.getParent();
            track = x;
        }

        while (!(current).isNull()) {
            if (y.getKey() != current.getKey()) {
                if (y.getKey().compareTo(current.getKey()) > 0) {
                    current.numRight--;
                }
                if (y.getKey().compareTo(current.getKey()) < 0) {
```



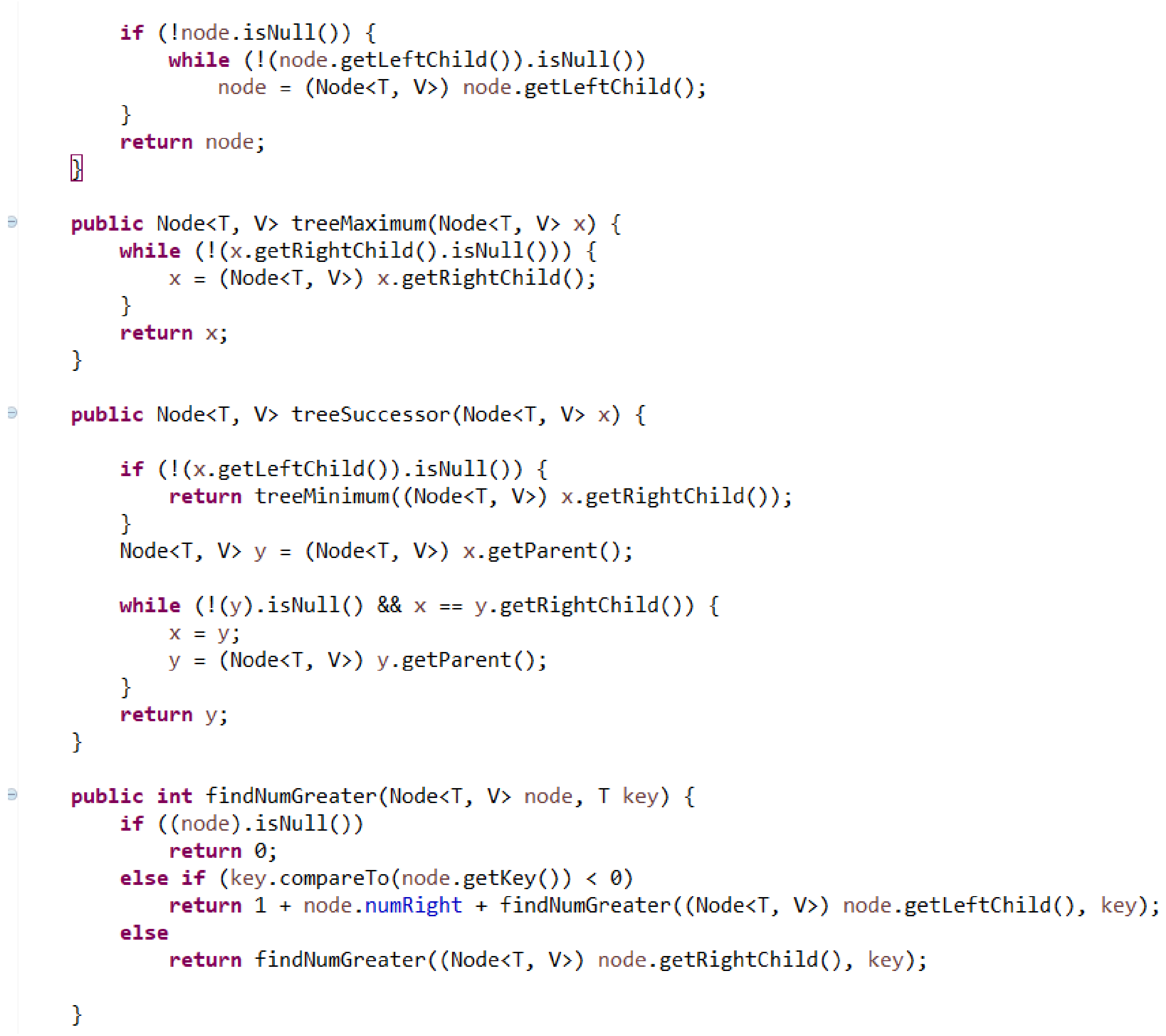
```
                    current.numLeft--;
                }
            } else {
                if ((current.getLeftChild()).isNull()) {
                    current.numLeft--;
                } else if ((current.getRightChild()).isNull()) {
                    current.numRight--;
                } else if (track == current.getRightChild()) {
                    current.numRight--;
                } else if (track == current.getLeftChild()) {
                    current.numLeft--;
                }
            }

            track = current;
            current = (Node<T, V>) current.getParent();

        }

    }

    private void deleteFixup(Node<T, V> x) {

        Node<T, V> w;

        while (x != root && x.getColor() == false) {

            if (x == x.getParent().getLeftChild()) {

                w = (Node<T, V>) x.getParent().getRightChild();

                if (w.getColor() == true) {
                    w.setColor(false);
                    x.getParent().setColor(true);
                    leftRotate((Node<T, V>) x.getParent());
                    w = (Node<T, V>) x.getParent().getRightChild();
                }
```



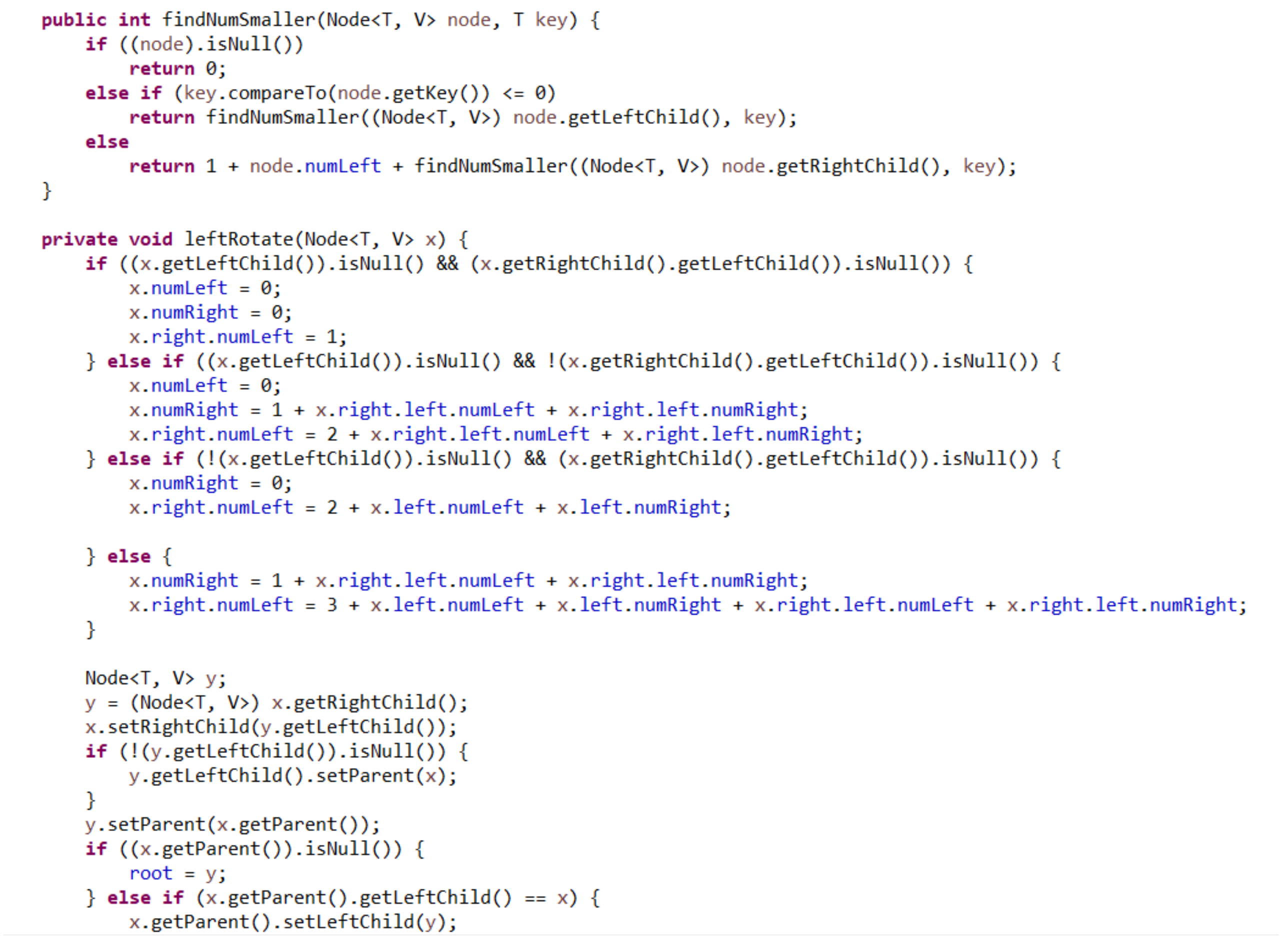
```
        if (w.getLeftChild().getColor() == false && w.getRightChild().getColor() == false) {
            w.setColor(true);
            x = (Node<T, V>) x.getParent();
        } else {
            if (w.getRightChild().getColor() == false) {
                w.getLeftChild().setColor(false);
                w.setColor(true);
                rightRotate(w);
                w = (Node<T, V>) x.getParent().getRightChild();
            }
            w.setColor(x.getParent().getColor());
            x.getParent().setColor(false);
            w.getRightChild().setColor(false);
            leftRotate((Node<T, V>) x.getParent());
            x = root;
        }
    } else {
        w = (Node<T, V>) x.getParent().getLeftChild();
        if (w.getColor() == true) {
            w.setColor(false);
            x.getParent().setColor(true);
            rightRotate((Node<T, V>) x.getParent());
            w = (Node<T, V>) x.getParent().getLeftChild();
        }
        if (w.getRightChild().getColor() == false && w.getLeftChild().getColor() == false) {
            w.setColor(true);
            x = (Node<T, V>) x.getParent();
        } else {
            if (w.getLeftChild().getColor() == false) {
                w.getRightChild().setColor(false);
                w.setColor(true);
                leftRotate(w);
                w = (Node<T, V>) x.getParent().getLeftChild();
            }

            w.setColor(x.getParent().getColor());
            x.getParent().setColor(false);
            w.getLeftChild().setColor(false);
```



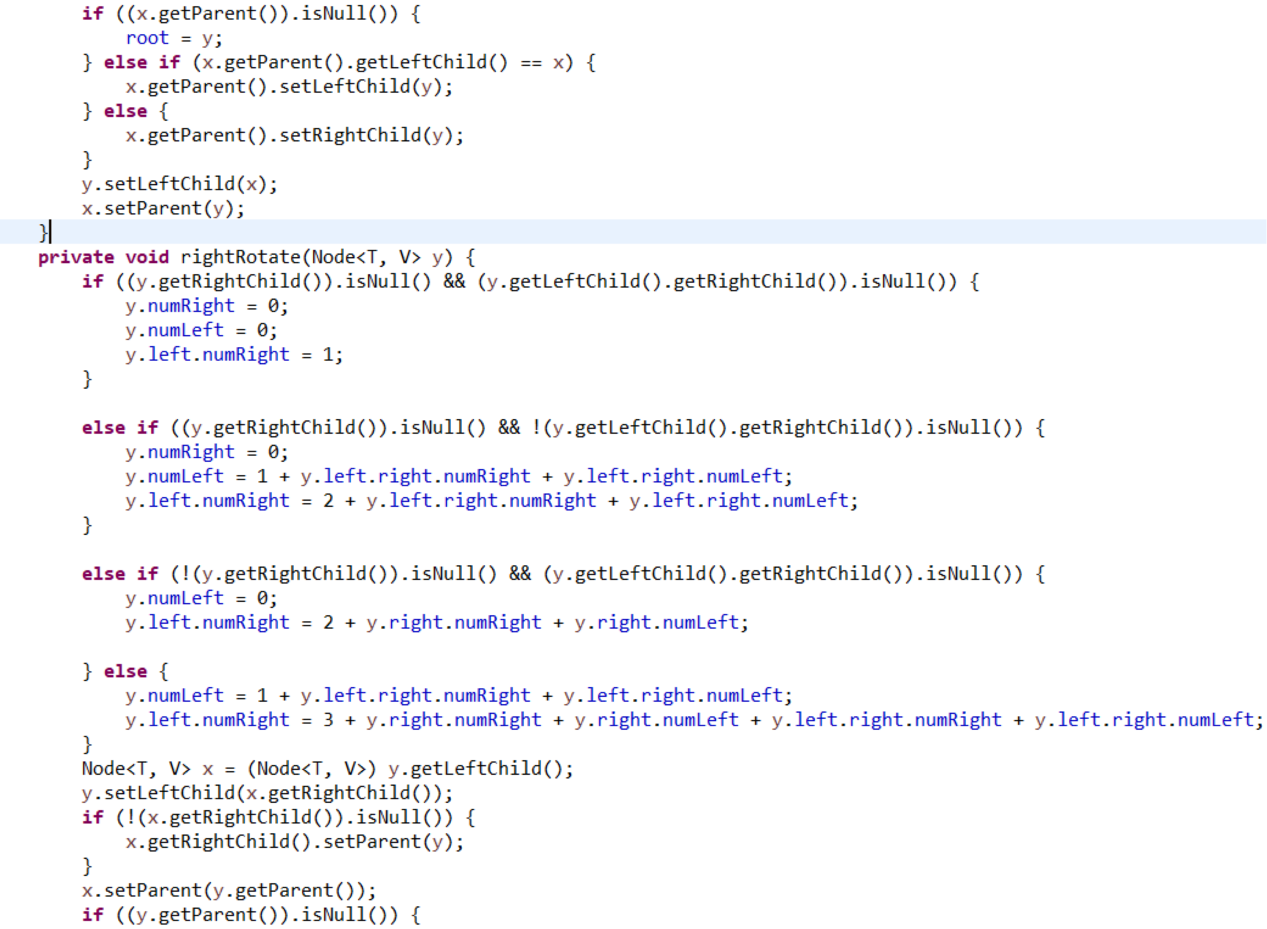
```
                    rightRotate((Node<T, V>) x.getParent());
                    x = root;
                }
            }
        }
        x.setColor(false);
    }

    @Override
    public V search(T key) {
        if (key == null) {
            throw new RuntimeErrorException(null);
        }
        Node<T, V> searchNode = new Node<>();
        searchNode = getNode(key);
        if (searchNode == null) {
            return null;
        } else {
            return searchNode.getValue();
        }
    }

    public Node<T, V> getNode(T key) {

        Node<T, V> current = root;
        while (!(current).isNull()) {
            if (current.getKey().compareTo(key) == 0) {
                return current;
            } else if (current.getKey().compareTo(key) < 0)
                current = (Node<T, V>) current.getRightChild();
            else
                current = (Node<T, V>) current.getLeftChild();
        }
        return null;

    }

    public Node<T, V> treeMinimum(Node<T, V> node) {
```



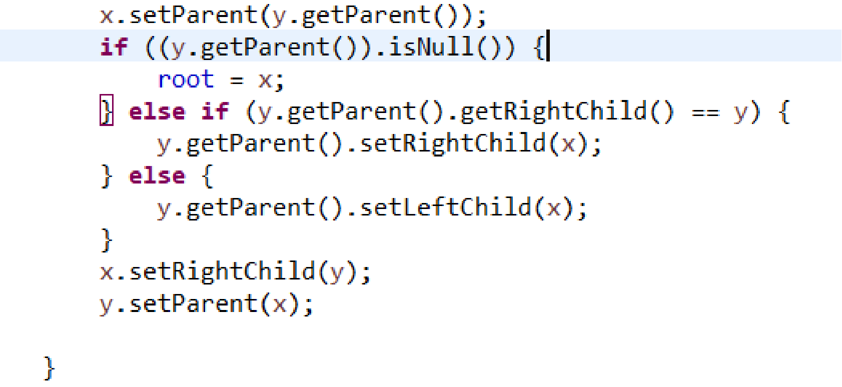
```java
        if (!node.isNull()) {
            while (!(node.getLeftChild()).isNull())
                node = (Node<T, V>) node.getLeftChild();
        }
        return node;
    }

    public Node<T, V> treeMaximum(Node<T, V> x) {
        while (!(x.getRightChild().isNull())) {
            x = (Node<T, V>) x.getRightChild();
        }
        return x;
    }

    public Node<T, V> treeSuccessor(Node<T, V> x) {

        if (!(x.getLeftChild()).isNull()) {
            return treeMinimum((Node<T, V>) x.getRightChild());
        }
        Node<T, V> y = (Node<T, V>) x.getParent();

        while (!(y).isNull() && x == y.getRightChild()) {
            x = y;
            y = (Node<T, V>) y.getParent();
        }
        return y;
    }

    public int findNumGreater(Node<T, V> node, T key) {
        if ((node).isNull())
            return 0;
        else if (key.compareTo(node.getKey()) < 0)
            return 1 + node.numRight + findNumGreater((Node<T, V>) node.getLeftChild(), key);
        else
            return findNumGreater((Node<T, V>) node.getRightChild(), key);

    }
```



```
public int findNumSmaller(Node<T, V> node, T key) {
    if ((node).isNull())
        return 0;
    else if (key.compareTo(node.getKey()) <= 0)
        return findNumSmaller((Node<T, V>) node.getLeftChild(), key);
    else
        return 1 + node.numLeft + findNumSmaller((Node<T, V>) node.getRightChild(), key);
}

private void leftRotate(Node<T, V> x) {
    if ((x.getLeftChild()).isNull() && (x.getRightChild().getLeftChild()).isNull()) {
        x.numLeft = 0;
        x.numRight = 0;
        x.right.numLeft = 1;
    } else if ((x.getLeftChild()).isNull() && !(x.getRightChild().getLeftChild()).isNull()) {
        x.numLeft = 0;
        x.numRight = 1 + x.right.left.numLeft + x.right.left.numRight;
        x.right.numLeft = 2 + x.right.left.numLeft + x.right.left.numRight;
    } else if (!(x.getLeftChild()).isNull() && (x.getRightChild().getLeftChild()).isNull()) {
        x.numRight = 0;
        x.right.numLeft = 2 + x.left.numLeft + x.left.numRight;

    } else {
        x.numRight = 1 + x.right.left.numLeft + x.right.left.numRight;
        x.right.numLeft = 3 + x.left.numLeft + x.left.numRight + x.right.left.numLeft + x.right.left.numRight;
    }

    Node<T, V> y;
    y = (Node<T, V>) x.getRightChild();
    x.setRightChild(y.getLeftChild());
    if (!(y.getLeftChild()).isNull()) {
        y.getLeftChild().setParent(x);
    }
    y.setParent(x.getParent());
    if ((x.getParent()).isNull()) {
        root = y;
    } else if (x.getParent().getLeftChild() == x) {
        x.getParent().setLeftChild(y);
```

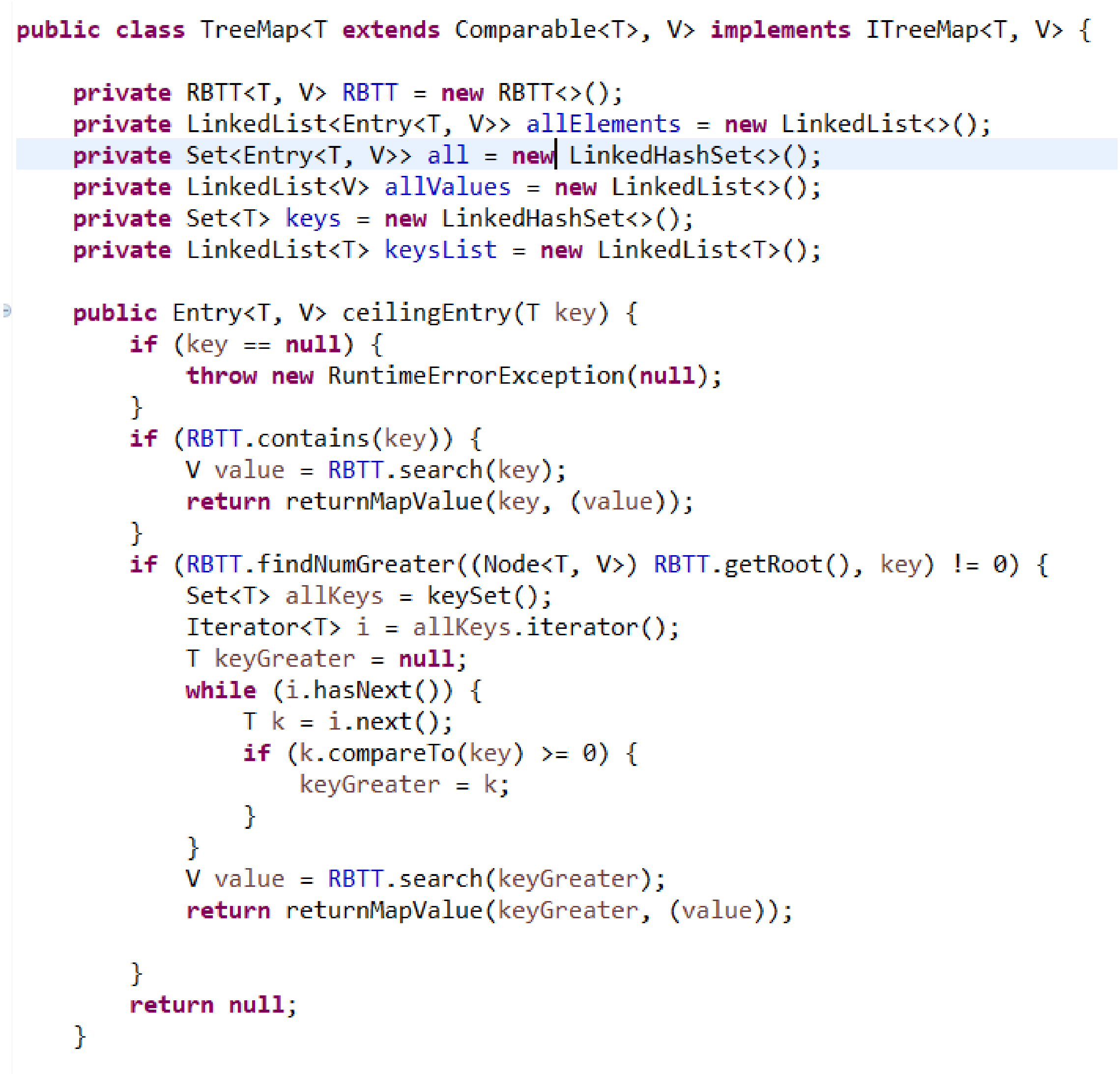


```
        if ((x.getParent()).isNull()) {
            root = y;
        } else if (x.getParent().getLeftChild() == x) {
            x.getParent().setLeftChild(y);
        } else {
            x.getParent().setRightChild(y);
        }
        y.setLeftChild(x);
        x.setParent(y);
    }
    private void rightRotate(Node<T, V> y) {
        if ((y.getRightChild()).isNull() && (y.getLeftChild().getRightChild()).isNull()) {
            y.numRight = 0;
            y.numLeft = 0;
            y.left.numRight = 1;
        }

        else if ((y.getRightChild()).isNull() && !(y.getLeftChild().getRightChild()).isNull()) {
            y.numRight = 0;
            y.numLeft = 1 + y.left.right.numRight + y.left.right.numLeft;
            y.left.numRight = 2 + y.left.right.numRight + y.left.right.numLeft;
        }

        else if (!(y.getRightChild()).isNull() && (y.getLeftChild().getRightChild()).isNull()) {
            y.numLeft = 0;
            y.left.numRight = 2 + y.right.numRight + y.right.numLeft;

        } else {
            y.numLeft = 1 + y.left.right.numRight + y.left.right.numLeft;
            y.left.numRight = 3 + y.right.numRight + y.right.numLeft + y.left.right.numRight + y.left.right.numLeft;
        }
        Node<T, V> x = (Node<T, V>) y.getLeftChild();
        y.setLeftChild(x.getRightChild());
        if (!(x.getRightChild()).isNull()) {
            x.getRightChild().setParent(y);
        }
        x.setParent(y.getParent());
        if ((y.getParent()).isNull()) {
```

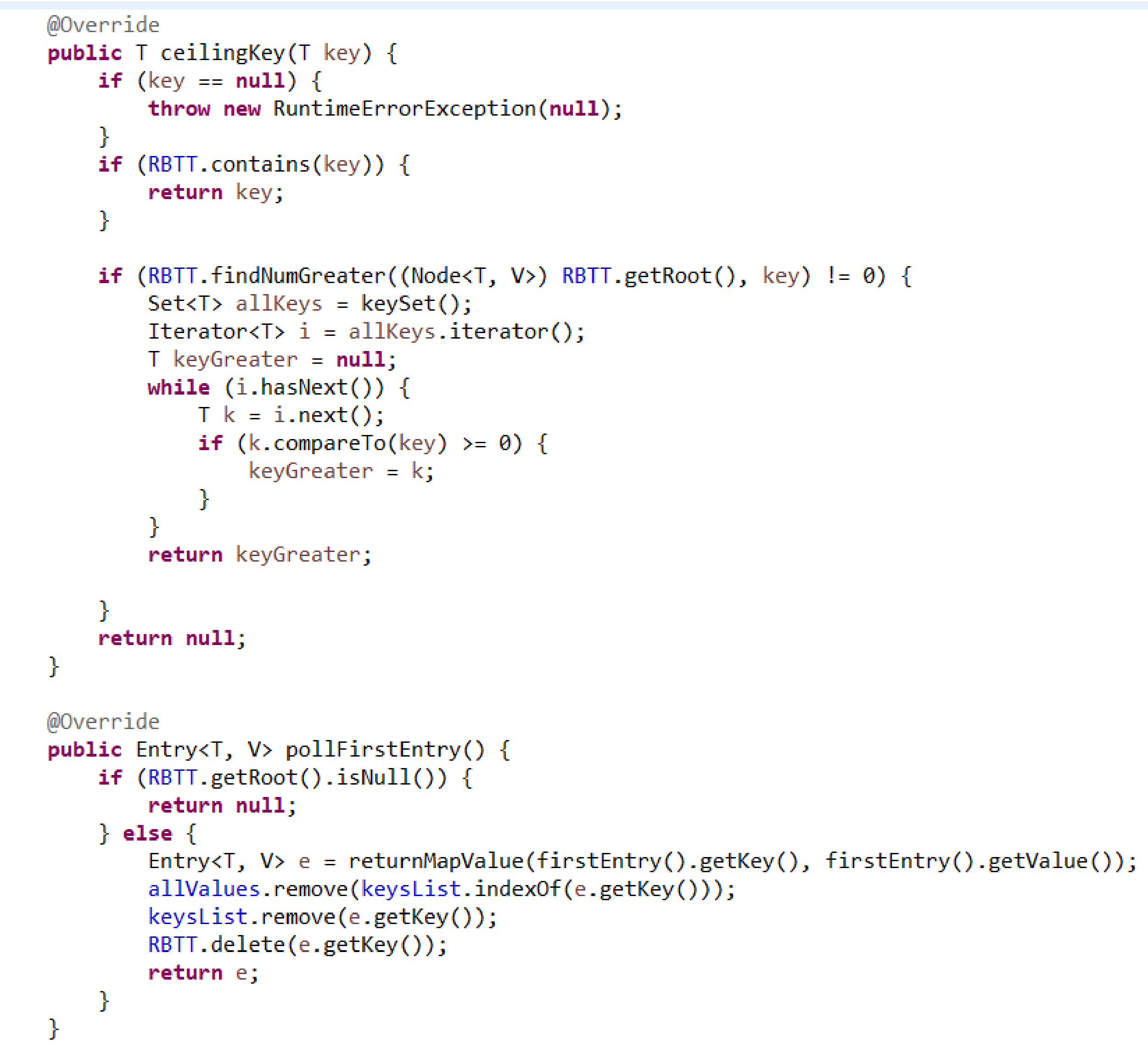


```
        x.setParent(y.getParent());
        if ((y.getParent()).isNull()) {
            root = x;
        } else if (y.getParent().getRightChild() == y) {
            y.getParent().setRightChild(x);
        } else {
            y.getParent().setLeftChild(x);
        }
        x.setRightChild(y);
        y.setParent(x);

    }
```

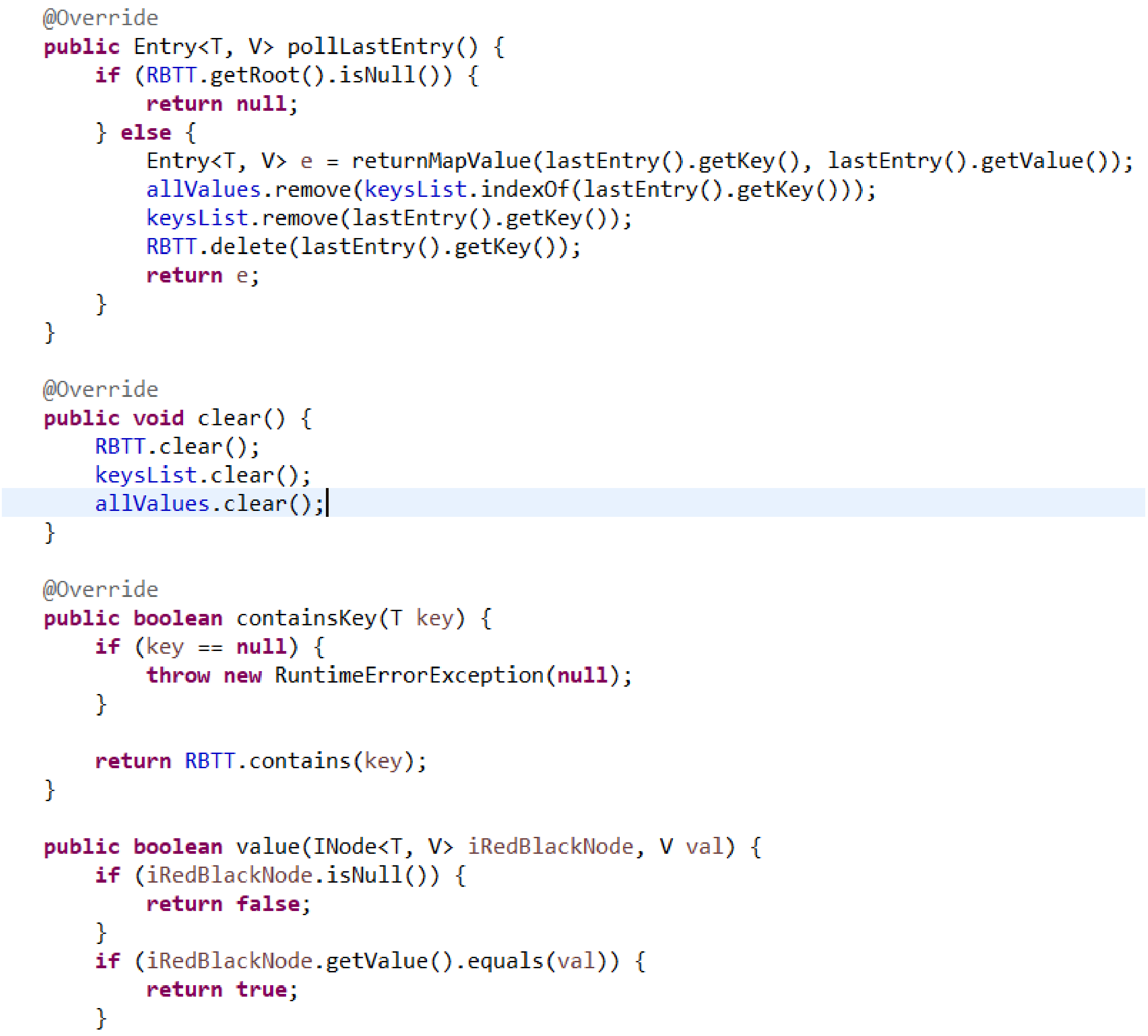
ITreeMap interface implementation:



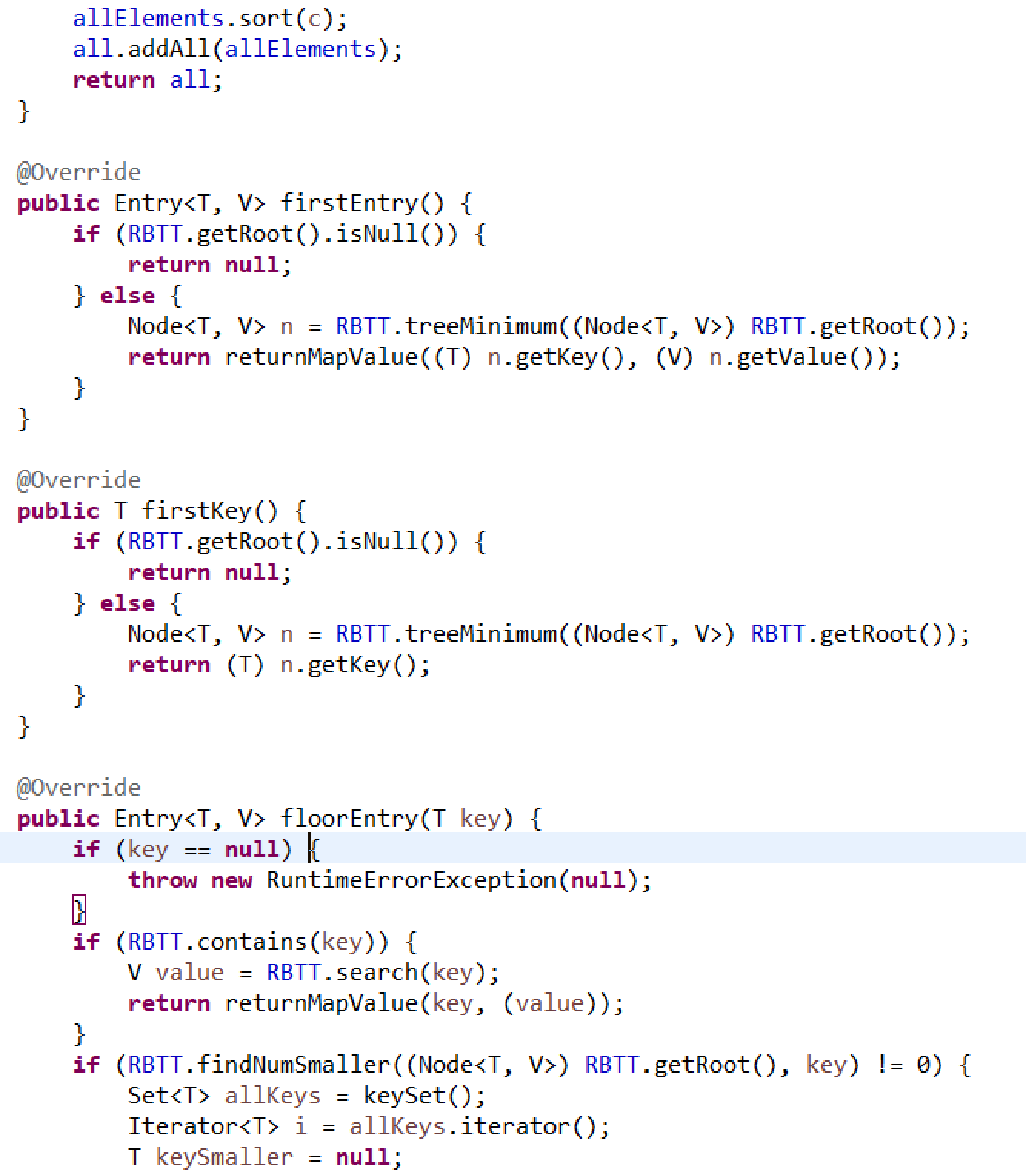
```java
public class TreeMap<T extends Comparable<T>, V> implements ITreeMap<T, V> {

    private RBTT<T, V> RBTT = new RBTT<>();
    private LinkedList<Entry<T, V>> allElements = new LinkedList<>();
    private Set<Entry<T, V>> all = new LinkedHashSet<>();
    private LinkedList<V> allValues = new LinkedList<>();
    private Set<T> keys = new LinkedHashSet<>();
    private LinkedList<T> keysList = new LinkedList<T>();

    public Entry<T, V> ceilingEntry(T key) {
        if (key == null) {
            throw new RuntimeErrorException(null);
        }
        if (RBTT.contains(key)) {
            V value = RBTT.search(key);
            return returnMapValue(key, (value));
        }
        if (RBTT.findNumGreater((Node<T, V>) RBTT.getRoot(), key) != 0) {
            Set<T> allKeys = keySet();
            Iterator<T> i = allKeys.iterator();
            T keyGreater = null;
            while (i.hasNext()) {
                T k = i.next();
                if (k.compareTo(key) >= 0) {
                    keyGreater = k;
                }
            }
            V value = RBTT.search(keyGreater);
            return returnMapValue(keyGreater, (value));

        }
        return null;
    }
```



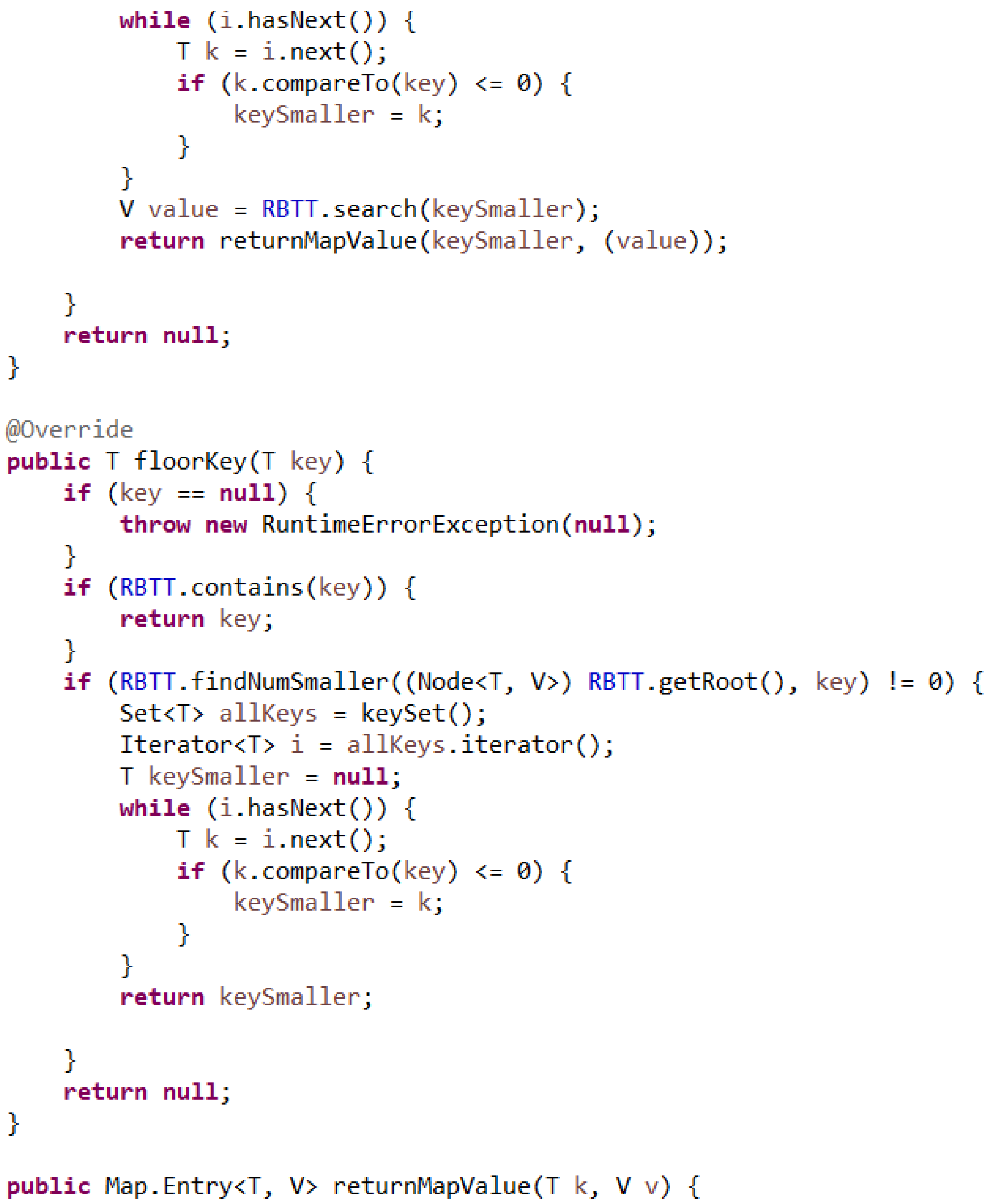
```
@Override
public T ceilingKey(T key) {
    if (key == null) {
        throw new RuntimeErrorException(null);
    }
    if (RBTT.contains(key)) {
        return key;
    }

    if (RBTT.findNumGreater((Node<T, V>) RBTT.getRoot(), key) != 0) {
        Set<T> allKeys = keySet();
        Iterator<T> i = allKeys.iterator();
        T keyGreater = null;
        while (i.hasNext()) {
            T k = i.next();
            if (k.compareTo(key) >= 0) {
                keyGreater = k;
            }
        }
        return keyGreater;

    }
    return null;
}

@Override
public Entry<T, V> pollFirstEntry() {
    if (RBTT.getRoot().isNull()) {
        return null;
    } else {
        Entry<T, V> e = returnMapValue(firstEntry().getKey(), firstEntry().getValue());
        allValues.remove(keysList.indexOf(e.getKey()));
        keysList.remove(e.getKey());
        RBTT.delete(e.getKey());
        return e;
    }
}
```



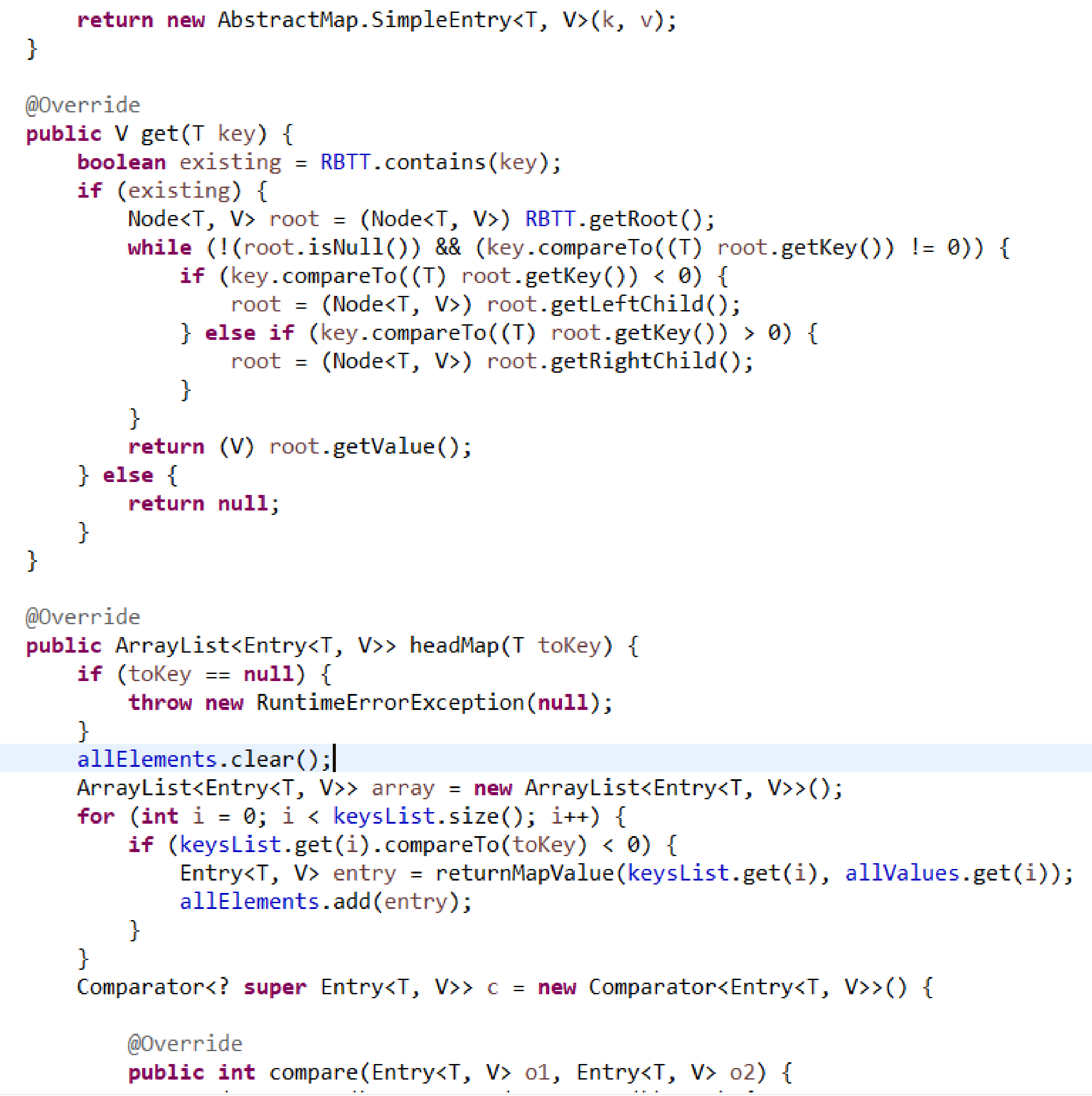
```
    @Override
    public Entry<T, V> pollLastEntry() {
        if (RBTT.getRoot().isNull()) {
            return null;
        } else {
            Entry<T, V> e = returnMapValue(lastEntry().getKey(), lastEntry().getValue());
            allValues.remove(keysList.indexOf(lastEntry().getKey()));
            keysList.remove(lastEntry().getKey());
            RBTT.delete(lastEntry().getKey());
            return e;
        }
    }

    @Override
    public void clear() {
        RBTT.clear();
        keysList.clear();
        allValues.clear();
    }

    @Override
    public boolean containsKey(T key) {
        if (key == null) {
            throw new RuntimeErrorException(null);
        }

        return RBTT.contains(key);
    }

    public boolean value(INode<T, V> iRedBlackNode, V val) {
        if (iRedBlackNode.isNull()) {
            return false;
        }
        if (iRedBlackNode.getValue().equals(val)) {
            return true;
        }
```



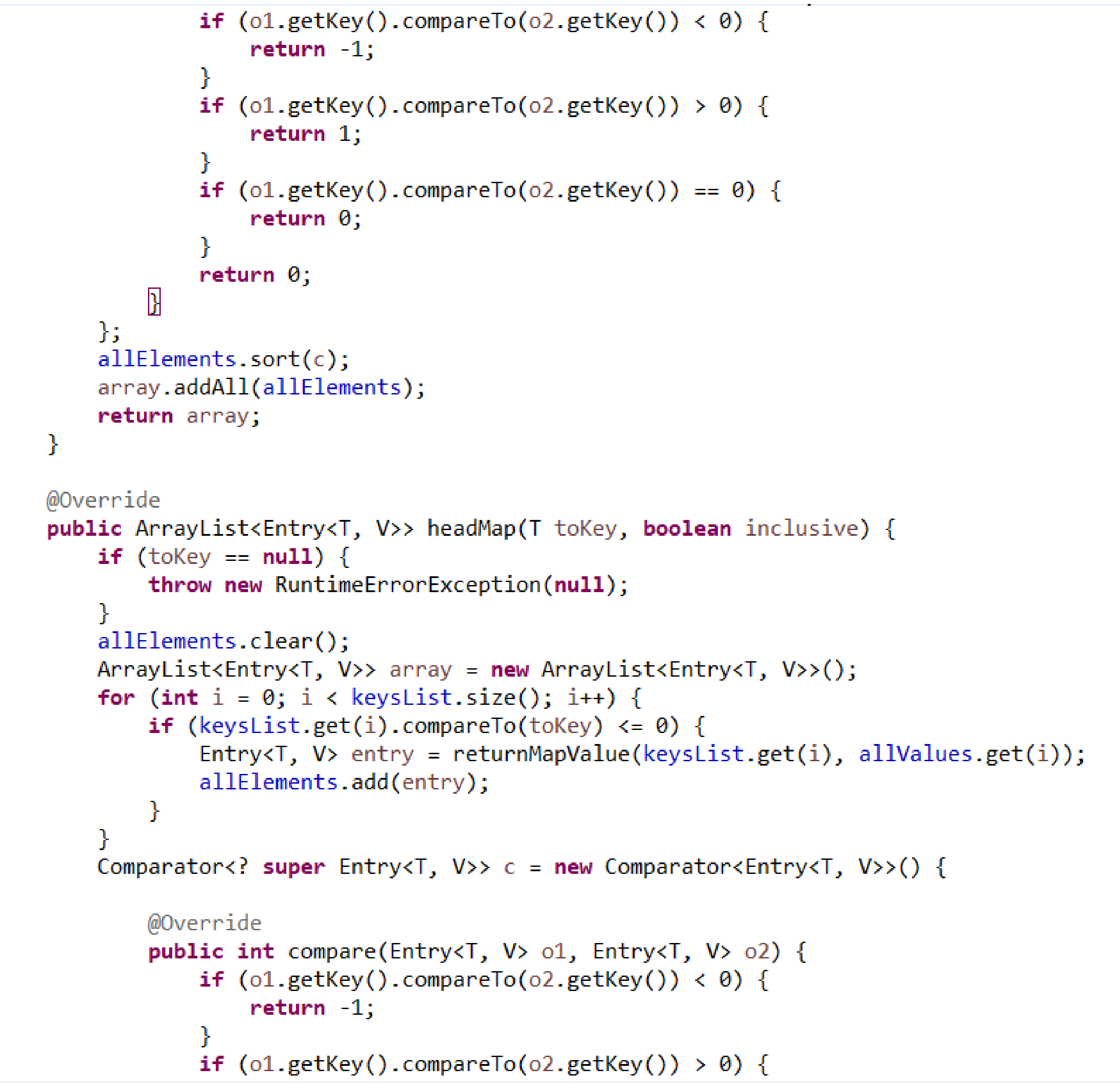
```java
        allElements.sort(c);
        all.addAll(allElements);
        return all;
    }

    @Override
    public Entry<T, V> firstEntry() {
        if (RBTT.getRoot().isNull()) {
            return null;
        } else {
            Node<T, V> n = RBTT.treeMinimum((Node<T, V>) RBTT.getRoot());
            return returnMapValue((T) n.getKey(), (V) n.getValue());
        }
    }

    @Override
    public T firstKey() {
        if (RBTT.getRoot().isNull()) {
            return null;
        } else {
            Node<T, V> n = RBTT.treeMinimum((Node<T, V>) RBTT.getRoot());
            return (T) n.getKey();
        }
    }

    @Override
    public Entry<T, V> floorEntry(T key) {
        if (key == null) {
            throw new RuntimeErrorException(null);
        }
        if (RBTT.contains(key)) {
            V value = RBTT.search(key);
            return returnMapValue(key, (value));
        }
        if (RBTT.findNumSmaller((Node<T, V>) RBTT.getRoot(), key) != 0) {
            Set<T> allKeys = keySet();
            Iterator<T> i = allKeys.iterator();
            T keySmaller = null;
```



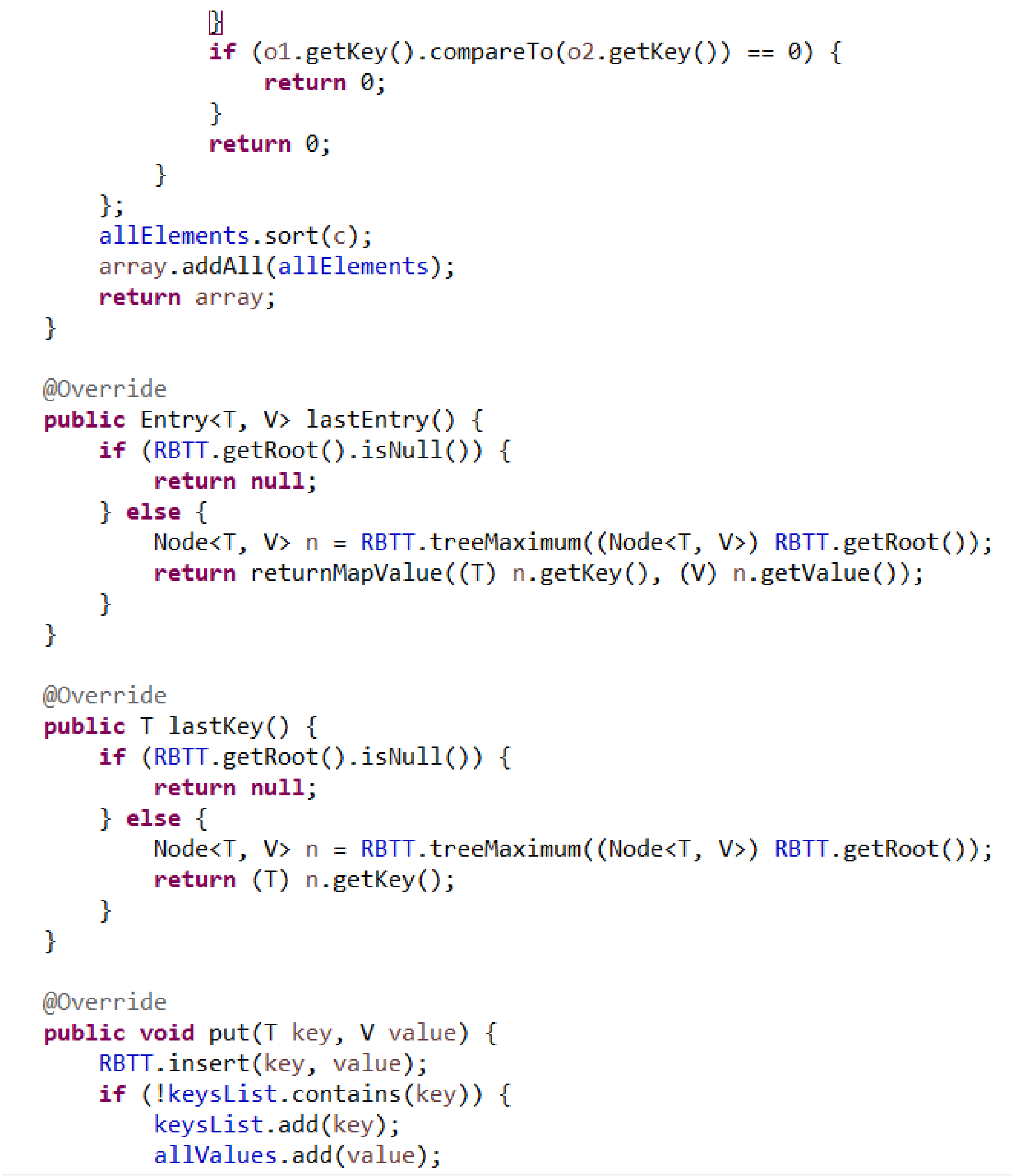
```
            while (i.hasNext()) {
                T k = i.next();
                if (k.compareTo(key) <= 0) {
                    keySmaller = k;
                }
            }
            V value = RBTT.search(keySmaller);
            return returnMapValue(keySmaller, (value));

        }
        return null;
    }

    @Override
    public T floorKey(T key) {
        if (key == null) {
            throw new RuntimeErrorException(null);
        }
        if (RBTT.contains(key)) {
            return key;
        }
        if (RBTT.findNumSmaller((Node<T, V>) RBTT.getRoot(), key) != 0) {
            Set<T> allKeys = keySet();
            Iterator<T> i = allKeys.iterator();
            T keySmaller = null;
            while (i.hasNext()) {
                T k = i.next();
                if (k.compareTo(key) <= 0) {
                    keySmaller = k;
                }
            }
            return keySmaller;

        }
        return null;
    }

    public Map.Entry<T, V> returnMapValue(T k, V v) {
```



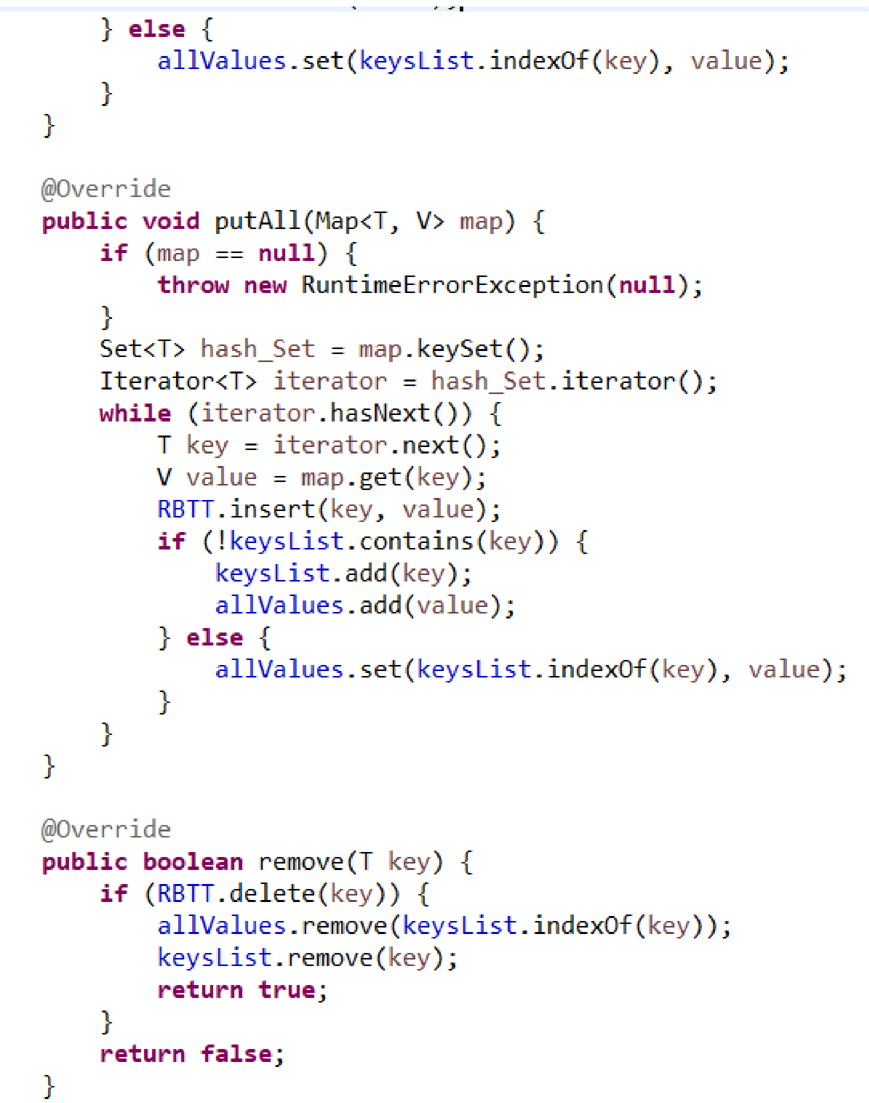
```
        return new AbstractMap.SimpleEntry<T, V>(k, v);
    }

    @Override
    public V get(T key) {
        boolean existing = RBTT.contains(key);
        if (existing) {
            Node<T, V> root = (Node<T, V>) RBTT.getRoot();
            while (!(root.isNull()) && (key.compareTo((T) root.getKey()) != 0)) {
                if (key.compareTo((T) root.getKey()) < 0) {
                    root = (Node<T, V>) root.getLeftChild();
                } else if (key.compareTo((T) root.getKey()) > 0) {
                    root = (Node<T, V>) root.getRightChild();
                }
            }
            return (V) root.getValue();
        } else {
            return null;
        }
    }

    @Override
    public ArrayList<Entry<T, V>> headMap(T toKey) {
        if (toKey == null) {
            throw new RuntimeErrorException(null);
        }
        allElements.clear();
        ArrayList<Entry<T, V>> array = new ArrayList<Entry<T, V>>();
        for (int i = 0; i < keysList.size(); i++) {
            if (keysList.get(i).compareTo(toKey) < 0) {
                Entry<T, V> entry = returnMapValue(keysList.get(i), allValues.get(i));
                allElements.add(entry);
            }
        }
        Comparator<? super Entry<T, V>> c = new Comparator<Entry<T, V>>() {

            @Override
            public int compare(Entry<T, V> o1, Entry<T, V> o2) {
```



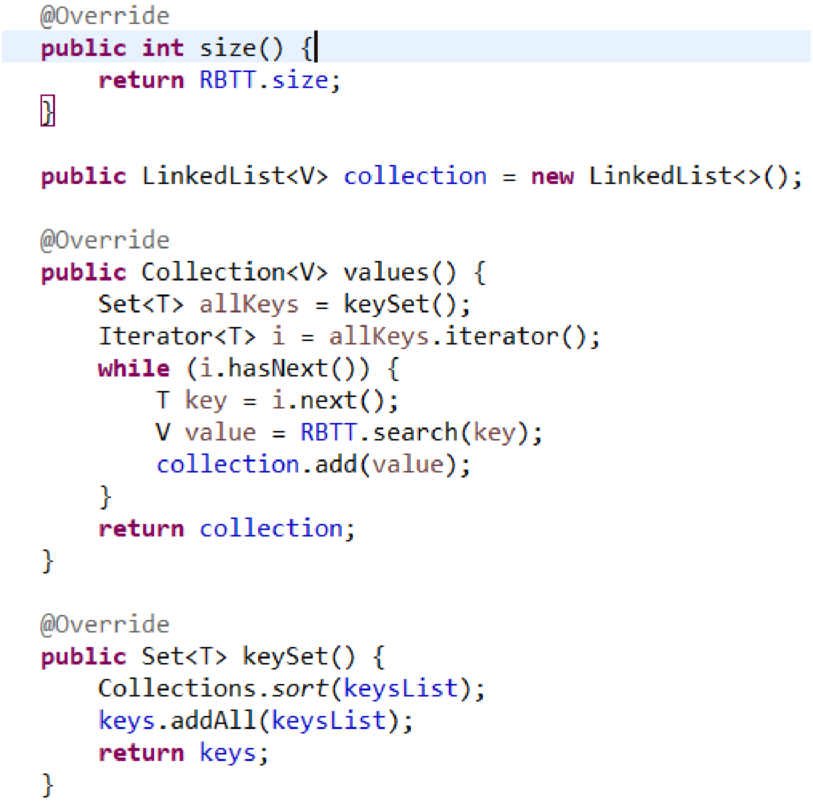
```
                if (o1.getKey().compareTo(o2.getKey()) < 0) {
                    return -1;
                }
                if (o1.getKey().compareTo(o2.getKey()) > 0) {
                    return 1;
                }
                if (o1.getKey().compareTo(o2.getKey()) == 0) {
                    return 0;
                }
                return 0;
            }
        };
        allElements.sort(c);
        array.addAll(allElements);
        return array;
    }

    @Override
    public ArrayList<Entry<T, V>> headMap(T toKey, boolean inclusive) {
        if (toKey == null) {
            throw new RuntimeErrorException(null);
        }
        allElements.clear();
        ArrayList<Entry<T, V>> array = new ArrayList<Entry<T, V>>();
        for (int i = 0; i < keysList.size(); i++) {
            if (keysList.get(i).compareTo(toKey) <= 0) {
                Entry<T, V> entry = returnMapValue(keysList.get(i), allValues.get(i));
                allElements.add(entry);
            }
        }
        Comparator<? super Entry<T, V>> c = new Comparator<Entry<T, V>>() {

            @Override
            public int compare(Entry<T, V> o1, Entry<T, V> o2) {
                if (o1.getKey().compareTo(o2.getKey()) < 0) {
                    return -1;
                }
                if (o1.getKey().compareTo(o2.getKey()) > 0) {
```



```java
                }
                if (o1.getKey().compareTo(o2.getKey()) == 0) {
                    return 0;
                }
                return 0;
            }
        };
        allElements.sort(c);
        array.addAll(allElements);
        return array;
    }

    @Override
    public Entry<T, V> lastEntry() {
        if (RBTT.getRoot().isNull()) {
            return null;
        } else {
            Node<T, V> n = RBTT.treeMaximum((Node<T, V>) RBTT.getRoot());
            return returnMapValue((T) n.getKey(), (V) n.getValue());
        }
    }

    @Override
    public T lastKey() {
        if (RBTT.getRoot().isNull()) {
            return null;
        } else {
            Node<T, V> n = RBTT.treeMaximum((Node<T, V>) RBTT.getRoot());
            return (T) n.getKey();
        }
    }

    @Override
    public void put(T key, V value) {
        RBTT.insert(key, value);
        if (!keysList.contains(key)) {
            keysList.add(key);
            allValues.add(value);
```



```
        } else {
            allValues.set(keysList.indexOf(key), value);
        }
    }

    @Override
    public void putAll(Map<T, V> map) {
        if (map == null) {
            throw new RuntimeErrorException(null);
        }
        Set<T> hash_Set = map.keySet();
        Iterator<T> iterator = hash_Set.iterator();
        while (iterator.hasNext()) {
            T key = iterator.next();
            V value = map.get(key);
            RBTT.insert(key, value);
            if (!keysList.contains(key)) {
                keysList.add(key);
                allValues.add(value);
            } else {
                allValues.set(keysList.indexOf(key), value);
            }
        }
    }

    @Override
    public boolean remove(T key) {
        if (RBTT.delete(key)) {
            allValues.remove(keysList.indexOf(key));
            keysList.remove(key);
            return true;
        }
        return false;
    }
```



```
@Override
public int size() {
    return RBTT.size;
}

public LinkedList<V> collection = new LinkedList<>();

@Override
public Collection<V> values() {
    Set<T> allKeys = keySet();
    Iterator<T> i = allKeys.iterator();
    while (i.hasNext()) {
        T key = i.next();
        V value = RBTT.search(key);
        collection.add(value);
    }
    return collection;
}

@Override
public Set<T> keySet() {
    Collections.sort(keysList);
    keys.addAll(keysList);
    return keys;
}
```