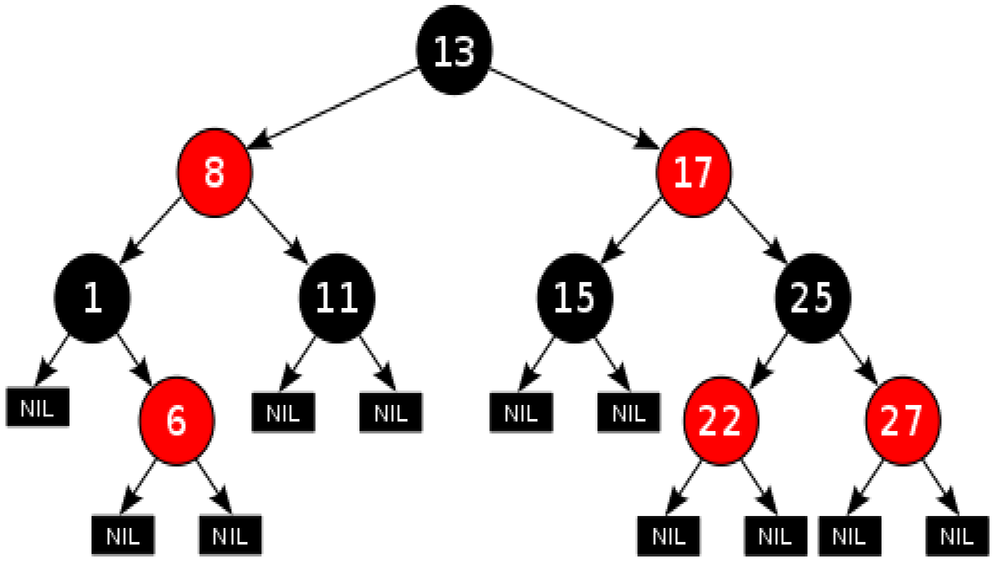
Lab 2

Implementing Red Black Tree & Treemap interface



Names:

Aya Gamal (01)

Linh Ahmed (50)

DESCRIPTION:

- **Red Black Tree**

  A redblack tree is a kind of self-balancing binary search tree in computer science. Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node. These color bits are used to ensure the tree remains approximately balanced during insertions and deletions. Balance is preserved by painting each node of the tree with one of two colors in a way that satisfies certain properties, which collectively constrain how unbalanced the tree can become in the worst case. When the tree is modified, the new tree is subsequently rearranged and repainted to restore the coloring properties. The properties are designed in such a way that this rearranging and recoloring can be performed efficiently.

- **TreeMap**

  A Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used. This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's Introduction to Algorithms.

Requirements:

You are required to implement the following interfaces:

1. **Red Black Tree**

   Which have methods to implement some basic procedures of red black tree

- **Search**:which runs in log(n) time , return the value associated with the given key or null if no value is found.
- **contains**: which runs in log(n) time , return true if the tree contains the given key and false otherwise.
- **insert**: which runs in log(n) time , insert the given key in the tree while maintaining the red black tree properties. If the key is already present in the tree, update its value.
- **delete**:which runs in log(n) time , delete the node associated with the given key. Return true in case of success and false otherwise.

2. Treemap

Which have methods to implement an interface similar to the one used in java Treemap.

- **ceilingEntry:** Returns a key-value mapping associated with the least key greater than or equal to the given key, or null if there is no such key.
- **ceilingKey:** Returns the least key greater than or equal to the given key, or null if there is no such key.
- **clear:** Removes all of the mappings from this map.
- **containsKey:** Returns true if this map contains a mapping for the specified key.
- **containsValue:** Returns true if this map maps one or more keys to the specified value.
- **entrySet:** Returns a Set view of the mappings contained in this map in ascending key order.
- **firstEntry:** Returns a key-value mapping associated with the least key in this map, or null if the map is empty.
- **firstKey:** Returns the first (lowest) key currently in this map, or null if the map is empty.

- **floorEntry:** Returns a key-value mapping associated with the greatest key less than or equal to the given key, or null if there is no such key.
- **floorKey:** Returns the greatest key less than or equal to the given key, or null if there is no such key.
- **get:** Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
- **headMap:** Returns a view of the portion of this map whose keys are strictly less than toKey in ascending order.
- **headMap:** Returns a view of the portion of this map whose keys are less than (or equal to, if inclusive is true) toKey in ascending order.
- **keySet:** Returns a Set view of the keys contained in this map.
- **lastEntry:** Returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
- **lastKey:** Returns the last (highest) key currently in this map.
- **pollFirstElement:** Removes and returns a key-value mapping associated with the least key in this map, or null if the map is empty.
- **pollLastEntry:** Removes and returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
- **put:** Associates the specified value with the specified key in this map.
- **putAll:** Copies all of the mappings from the specified map to this map.
- **remove:** Removes the mapping for this key from this TreeMap if present.
- **size:** Returns the number of key-value mappings in this map.
- **values:** Returns a Collection view of the values contained in this map

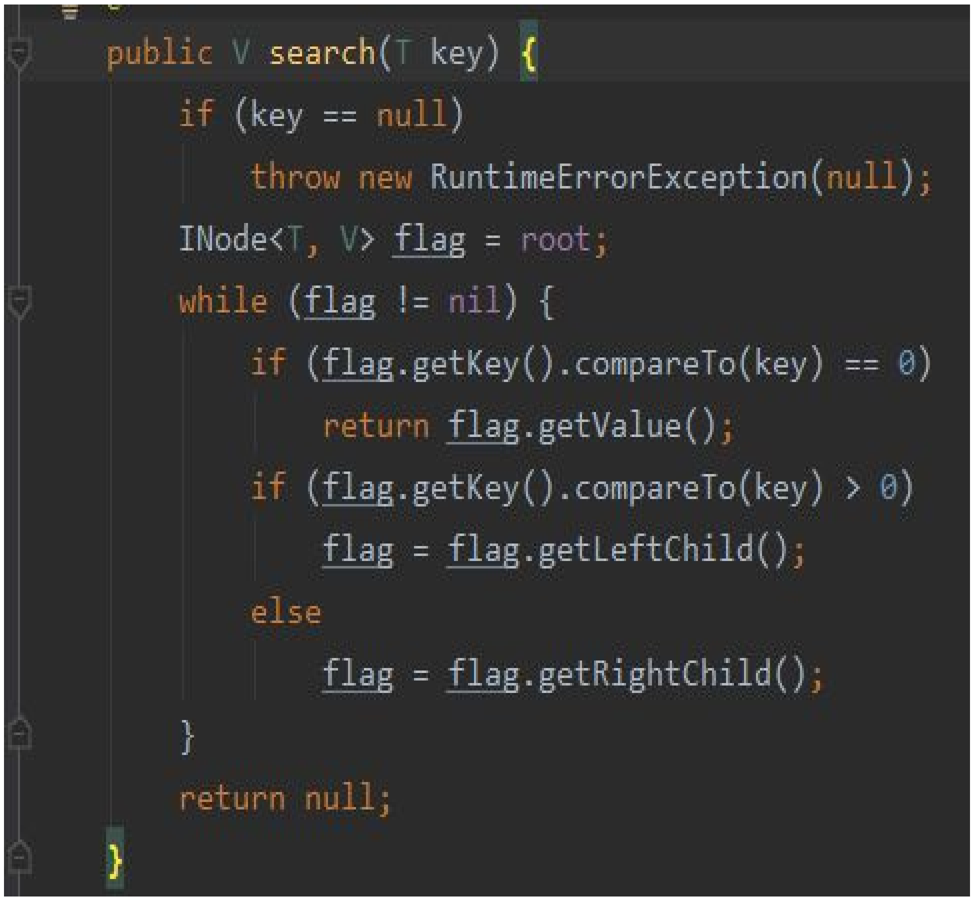
Time Analysis:

The height of a Red Black Tree is at most 2log(n+1)

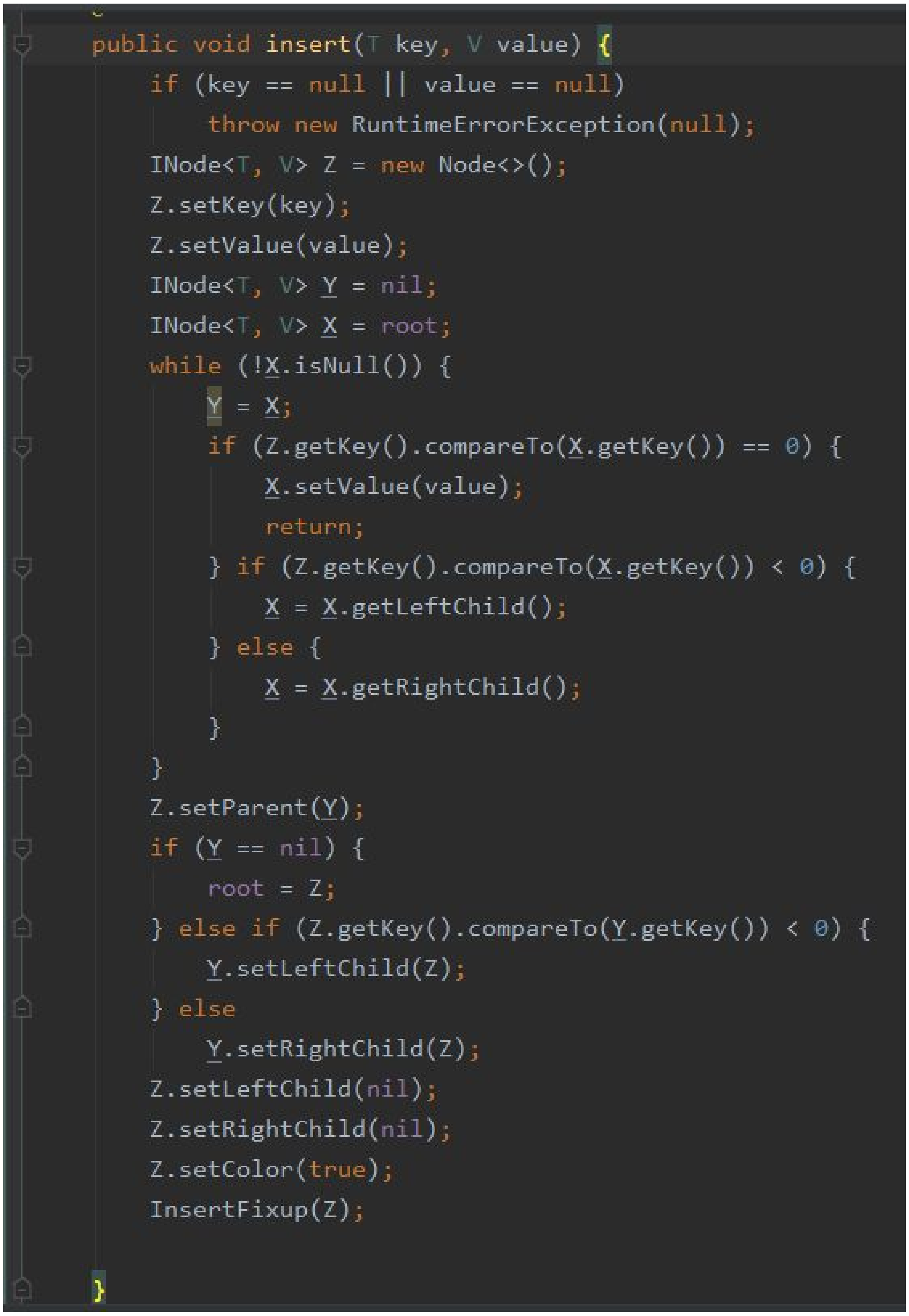
Function	Time Complexity
Search	O (log n)
Contains	O (log n)
Insert	O (log n)
Delete	O (log n)
getRoot	O (1)
isEmpty	O (1)
clear	O (1)

Function	Time Complexity	Function	Time Complexity
ceilingEntry	O(n)	headMap	O(n)
ceilingKey	O(n)	keySet	O()
clear	O(1)	lastEntry	O (n)
containsKey	O(log n)	lastKey	O (n)
containsValu e	O(log n)	pollFirstElement	O(log n)
entrySet	O(n)	pollLastEntry	O(log n)
firstEntry	O(n)	put	O(log n)
firstKey	O(n)	putAll	O(log n)
floorEntry	O(n)	remove	O(log n)
floorKey	O(n)	size	O(1)
get	O(log n)	values	O(logn)
headMap	O(n)		

Code snippets:

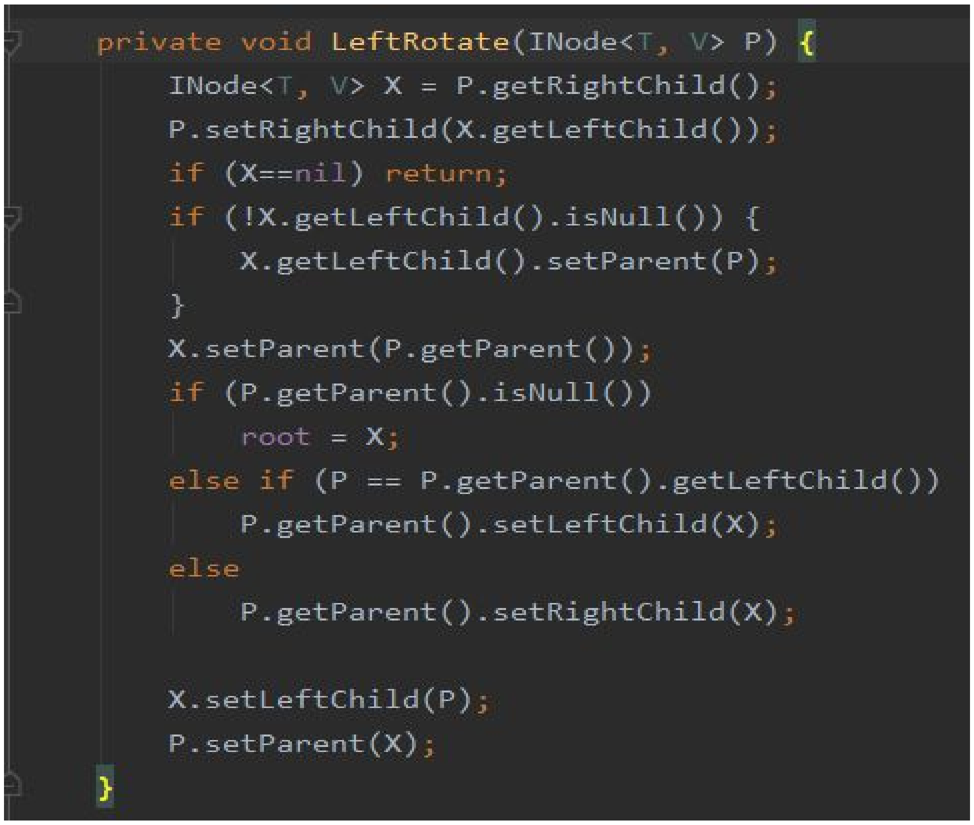


```java
public V search(T key) {
    if (key == null)
        throw new RuntimeErrorException(null);
    INode<T, V> flag = root;
    while (flag != nil) {
        if (flag.getKey().compareTo(key) == 0)
            return flag.getValue();
        if (flag.getKey().compareTo(key) > 0)
            flag = flag.getLeftChild();
        else
            flag = flag.getRightChild();
    }
    return null;
}
```



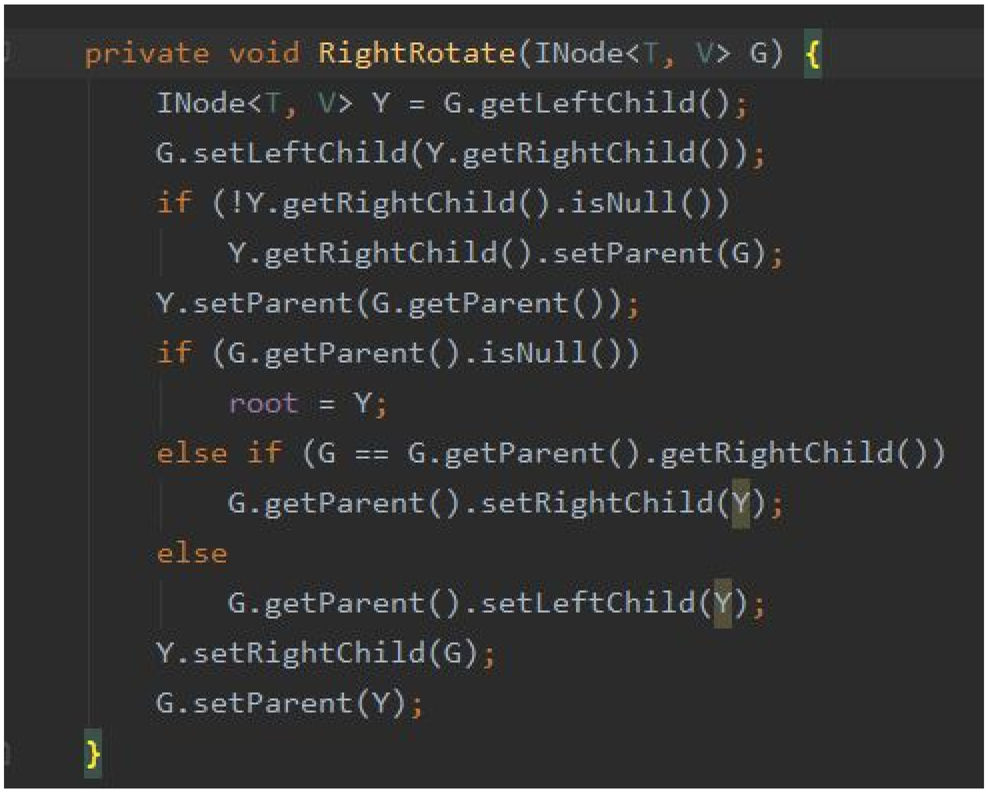
```
    public void insert(T key, V value) {
        if (key == null || value == null)
            throw new RuntimeErrorException(null);
        INode<T, V> Z = new Node<>();
        Z.setKey(key);
        Z.setValue(value);
        INode<T, V> Y = nil;
        INode<T, V> X = root;
        while (!X.isNull()) {
            Y = X;
            if (Z.getKey().compareTo(X.getKey()) == 0) {
                X.setValue(value);
                return;
            } if (Z.getKey().compareTo(X.getKey()) < 0) {
                X = X.getLeftChild();
            } else {
                X = X.getRightChild();
            }
        }
        Z.setParent(Y);
        if (Y == nil) {
            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);
        InsertFixup(Z);

    }
```

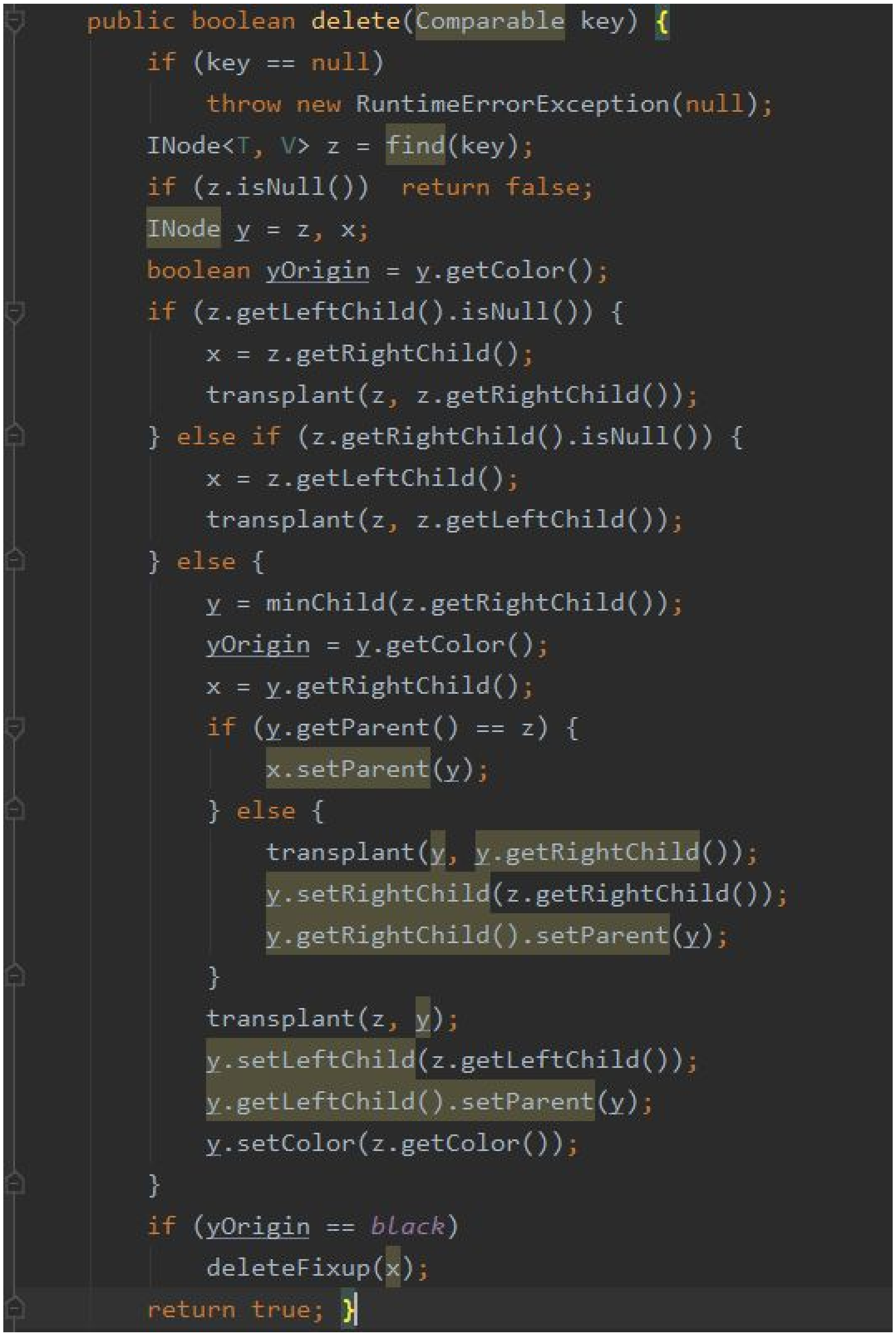


```java
private void LeftRotate(INode<T, V> P) {
    INode<T, V> X = P.getRightChild();
    P.setRightChild(X.getLeftChild());
    if (X==nil) return;
    if (!X.getLeftChild().isNull()) {
        X.getLeftChild().setParent(P);
    }
    X.setParent(P.getParent());
    if (P.getParent().isNull())
        root = X;
    else if (P == P.getParent().getLeftChild())
        P.getParent().setLeftChild(X);
    else
        P.getParent().setRightChild(X);

    X.setLeftChild(P);
    P.setParent(X);
}
```



```java
private void RightRotate(INode<T, V> G) {
    INode<T, V> Y = G.getLeftChild();
    G.setLeftChild(Y.getRightChild());
    if (!Y.getRightChild().isNull())
        Y.getRightChild().setParent(G);
    Y.setParent(G.getParent());
    if (G.getParent().isNull())
        root = Y;
    else if (G == G.getParent().getRightChild())
        G.getParent().setRightChild(Y);
    else
        G.getParent().setLeftChild(Y);
    Y.setRightChild(G);
    G.setParent(Y);
}
```


```java
    public boolean delete(Comparable key) {
        if (key == null)
            throw new RuntimeErrorException(null);
        INode<T, V> z = find(key);
        if (z.isNull())  return false;
        INode y = z, x;
        boolean yOrigin = y.getColor();
        if (z.getLeftChild().isNull()) {
            x = z.getRightChild();
            transplant(z, z.getRightChild());
        } else if (z.getRightChild().isNull()) {
            x = z.getLeftChild();
            transplant(z, z.getLeftChild());
        } else {
            y = minChild(z.getRightChild());
            yOrigin = y.getColor();
            x = y.getRightChild();
            if (y.getParent() == z) {
                x.setParent(y);
            } else {
                transplant(y, y.getRightChild());
                y.setRightChild(z.getRightChild());
                y.getRightChild().setParent(y);
            }
            transplant(z, y);
            y.setLeftChild(z.getLeftChild());
            y.getLeftChild().setParent(y);
            y.setColor(z.getColor());
        }
        if (yOrigin == black)
            deleteFixup(x);
        return true; }
```