2019

Red Black Tree

NAME:MAHMOUD HAMED SHARSHAR
ID:46

Problem Statement:

A red black 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.

Specification requirements:

- Implementation of red black tree data structure.
- Implementation of Tree Map data structure based on red black tree implementation.
- Implementation of red black tree supports insert, delete and search.

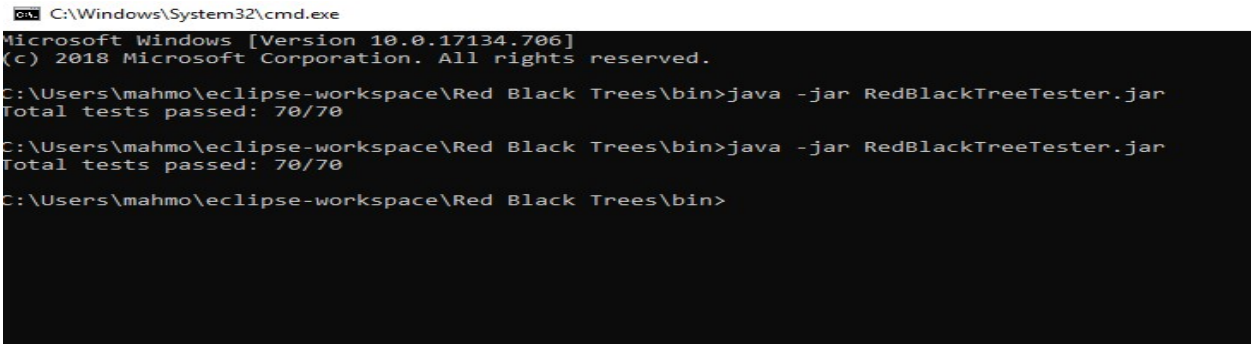
Data Structure used:

- Linked data structure to represent relations between nodes in the tree.

Main Modules:

- RBTNode class: data structure to represent each node in red black tree.
- RBTree class: module to represent red black tree and its operations like search, delete, insert, contain, etc.
- RBTreeMap class: represents tree map data structure based on RBTree class and support all operations of i.

Test cases :Test cases exist in bin folder as jar file and all of them passed.



.

Code snapshots:

Basic methods in RBTree class:

- Insert :.

```
@Override
public void insert(T key, V value) {
    if (key != null && value != null) {
        INode<T, V> y = this.NIL;
        INode<T, V> x = this.root;
        while (!x.equals(this.NIL)) {
            y = x;
            if (x.getKey().equals(key)) {
                x.setValue(value);// if key exist update its value
                return;
            } else if (x.getKey().compareTo(key) > 0)
                x = x.getLeftChild();
            else
                x = x.getRightChild();
        }

        INode<T, V> newNode = new RBTNode<>();
        newNode.setKey(key);
        newNode.setValue(value);
        newNode.setColor(INode.RED);

        if (y.equals(this.NIL))
            this.root = newNode;
        else if (y.getKey().compareTo(key) > 0)
            y.setLeftChild(newNode);
        else
            y.setRightChild(newNode);

        newNode.setParent(y);
        newNode.setLeftChild(this.NIL);
        newNode.setRightChild(this.NIL);
        this.size++;
        RBInsertFixUp(newNode);
    } else
        throw new RuntimeErrorException(new Error());
}
```

- RBInsertFixUp:

```
private void RBInsertFixUp(INode<T, V> node) {

    while (node.getParent().getColor() == this.red) {
        if (node.getParent().equals(node.getParent().getParent().getLeftChild())) {
            INode<T, V> uncle = node.getParent().getParent().getRightChild();
            // Case 1
            if (uncle.getColor() == this.red) {
                uncle.setColor(this.black);
                node.getParent().setColor(this.black);
                node.getParent().getParent().setColor(this.red);
                node = node.getParent().getParent();
            } else {
                // Case 2
                if (node.equals(node.getParent().getRightChild())) {
                    node = node.getParent();
                    leftRotate(node);
                }
                // Case 3
                node.getParent().setColor(this.black);
                node.getParent().getParent().setColor(this.red);
                rightRotate(node.getParent().getParent());
            }
        } else {
            INode<T, V> uncle = node.getParent().getParent().getLeftChild();
            // Case 1
            if (uncle.getColor() == this.red) {
                uncle.setColor(this.black);
                node.getParent().setColor(this.black);
                node.getParent().getParent().setColor(this.red);
                node = node.getParent().getParent();
            } else {
                // Case 2
                if (node.equals(node.getParent().getLeftChild())) {
                    node = node.getParent();
                    rightRotate(node);
                }
                // Case 3
                node.getParent().setColor(this.black);
                node.getParent().getParent().setColor(this.red);
                leftRotate(node.getParent().getParent());
            }
        }
    }

    this.root.setColor(this.black);
}
```

- Delete:

```
@Override
public boolean delete(T key) {
    if (key != null) {
        // check if key exist or not
        INode<T, V> z = this.root;
        boolean flag = false;
        while (!z.equals(this.NIL) /* && key != null */) {
            if (z.getKey().equals(key)) {
                flag = true;
                break;
            } else if (z.getKey().compareTo(key) > 0)
                z = z.getLeftChild();
            else
                z = z.getRightChild();
        }
        if (!flag)
            return false;
        // implement deletion
        INode<T, V> x = new RBTNode<>();
        INode<T, V> y = z;
        boolean originalColor = y.getColor();
        if (z.getLeftChild().equals(this.NIL)) {
            x = z.getRightChild();
            transplant(z, z.getRightChild());
        } else if (z.getRightChild().equals(this.NIL)) {
            x = z.getLeftChild();
            transplant(z, z.getLeftChild());
        } else {
            y = RBTMin(z.getRightChild());
            originalColor = y.getColor();
            x = y.getRightChild();
            if (y.getParent().equals(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 (originalColor == black)
            RBDeleteFixUp(x);

        this.size--;
        return true;
    } else
        throw new RuntimeErrorException(new Error());

}
```

- RBDeleteFixUp:

```
    private void RBDeleteFixUp(INode<T, V> x) {
        while (!x.equals(this.root) && x.getColor() == black) {
            if (x.equals(x.getParent().getLeftChild())) {
                INode<T, V> w = x.getParent().getRightChild();
                if (w.getColor() == this.red) {
                    w.setColor(this.black);
                    x.getParent().setColor(this.red);
                    leftRotate(x.getParent());
                    w = x.getParent().getRightChild();
                }
                if (w.getLeftChild().getColor() == black && w.getRightChild().getColor() == black) {
                    w.setColor(red);
                    x = x.getParent();
                } else {
                    if (w.getRightChild().getColor() == black) {
                        w.getLeftChild().setColor(black);
                        w.setColor(red);
                        rightRotate(w);
                        w = x.getParent().getRightChild();

                    }
                    w.setColor(x.getParent().getColor());
                    x.getParent().setColor(black);
                    w.getRightChild().setColor(black);
                    leftRotate(x.getParent());
                    x = this.root;
                }
            } else {
                INode<T, V> w = x.getParent().getLeftChild();
                if (w.getColor() == this.red) {
                    w.setColor(this.black);
                    x.getParent().setColor(this.red);
                    rightRotate(x.getParent());
                    w = x.getParent().getLeftChild();
                }
                if (w.getRightChild().getColor() == black && w.getLeftChild().getColor() == black) {
                    w.setColor(red);
                    x = x.getParent();
                } else {
                    if (w.getLeftChild().getColor() == black) {
                        w.getRightChild().setColor(black);
                        w.setColor(red);
                        leftRotate(w);
                        w = x.getParent().getLeftChild();


                    }
                    w.setColor(x.getParent().getColor());
                    x.getParent().setColor(black);
                    w.getLeftChild().setColor(black);
                    rightRotate(x.getParent());
                    x = this.root;
                }
            }
        }

        x.setColor(black);
    }
}
```

Functions Description of TreeMap Class:

- **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.

Functions Description of RBTree Class:

- **getRoot:** return the root of the given Red black tree.
- **isEmpty:** return whether the given tree isEmpty or not.
- **Clear:** Clear all keys in the given tree.
- **Search:** return the value associated with the given key or null if no value is found.
- **Contains :** return true if the tree contains the given key and false otherwise.
- **Insert :** 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:** Delete the node associated with the given key. Return true in case of success and false otherwise.