LAB 2 REPORT

Red Black Tree

Name: Sherif Mohamed Mostafa. ID:20.

Name: Ahmed Ali Abdelmeguid Ali. ID: 5.

"Lab 2"

Problem statement:

The required was implementing the following two interfaces:

- Red Black Tree interface: an interface for the red black tree data structure containing all the procedures for its manipulation (insertion, deletion….).
- Node interface: an interface for the node of the red black tree includes its key, value and color bit and methods for manipulating its connections with other nodes (pointer based).
- Tree Map interface: an interface similar to the java Tree Map interface, which is a navigable map, made using the implemented red black tree data structure.

Notes on Red Black Tree:

The red black tree is a kind of a self-balancing tree that ensures that the longest path is at most twice the shortest which prevents any kind of linear chaining. It does so by manipulation f an extra color attribute to be added to the node and maintaining certain properties, which are:

1. The root is black.
2. Any node is either red or black.
3. A red node cannot have a red child.
4. The leaf nodes (which are here named nil) are also black and they represent the null entry (an empty tree has nil as its root).
5. The black height (number of black nodes) from any node to any of its leaves is the same.

Mostly properties 3, 5 are the ones balancing the tree.

As the red black tree is a kind of self-balancing BST, then its operations run in O(H) time where H is the height and due to the balancing effect running time is O(lg n) with n as the number of nodes.

Implementation:

First RbNode:

A class implementing the red black tree node with the color bits implemented as Booleans (Black = true, Red = false) providing the following methods:

- A constructor: that forms a new node and sets its value and key; it also sets the color as red (because a newly inserted node is assumed red).
- Setters and getters for the key, value and color.
- Methods for manipulating the connections with other nodes (parent and children).

Code Snippets:

```java
public class RbNode<T extends Comparable<T>, V> implements INode<T, V> {
    static final boolean RED   = true;
    static final boolean BLACK = false;
    private T key;
    private V value;
    private INode<T,V> parent;
    private INode<T,V> leftChild;
    private INode<T,V> rightChild;
    private boolean color;
    public RbNode() {

    }
    public RbNode(T key,V value) {
        setKey(key);
        setValue(value);
        setColor(RED);
        setLeftChild(null);
        setRightChild(null);
    }

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

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

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

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

```java
@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 boolean getColor() {
    return color;
}

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

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

Second :MyRedBlackTree:

A class implementing the red black tree data structure and provides the following procedures:

- Search : gives the value associated with a certain key after searching for it and null if the node wasn't found O(lgn).
- getNode: gets the node given its key and returns nil if node isn't found O(lgn).
- Contains: returns true if the given key is in the tree O(lgn).
- Insert: inserts the given key value mapping to the tree and updates the value if the given key was already inserted before, it then fixes the tree properties handling all the cases using insert fix up procedure (code and case comments are found in the snippets and in original code) O(lgn).
- Delete: given a key it deletes the node associated with it from the tree using the regular BST deletion procedure, then calls delete fix up to fix the tree properties handling all the cases (code and case comments are found in the snippets and in original code) O(lgn).
- Also provides the following procedures:
    1. Maximum: gets maximum key associated node in the sub tree rooted by the given node.
    2. Minimum: gets minimum key associated node in the sub tree rooted by the given node.
    3. Successor: gets the successor of the given node.
    4. Predecessor: gets the predecessor of the given node.

Code Snippets:

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

    private int size;
    private INode<T, V> root;
    public INode<T, V> nil;

    public MyRedBlackTree() {// initialize the tree and create the T.Nil.
        nil = new RbNode<T, V>(null, null);
        nil.setColor(RbNode.BLACK);
        root = nil;
        setSize(0);
    }

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

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

    @Override
    public void clear() {// simply let the root be T.Nil.
        size = 0;
        this.root = this.nil;
    }

    @Override
    public V search(T key) {
        if (key == null)
            throw new RuntimeErrorException(null);
        return getNode(key).getValue();
    }

    @Override
    public boolean contains(T key) {
        if (key == null)
            throw new RuntimeErrorException(null);
        return getNode(key) != this.nil;
    }
```

Insert code:

```java
@Override
public void insert(T key, V value) {
    if (key == null || value == null)
        throw new RuntimeErrorException(null);
    INode<T, V> node = getNode(key);
    if (node != this.nil) {// the key is found so only update its value.
        node.setValue(value);
    } else {// the key is not found so we need to insert a new node.
        size++;
        INode<T, V> inserted = new RbNode<T, V>(key, value);
        INode<T, V> location = this.root;
        INode<T, V> parent = this.nil;
        while (location != this.nil) {// get the correct location to insert the new node (finding its parent).
            parent = location;// is going to be the parent.
            if (key.compareTo(location.getKey()) < 0)
                location = location.getLeftChild();
            else
                location = location.getRightChild();
        }
        inserted.setParent(parent);
        if (parent == this.nil)
            setRoot(inserted);// first node to be inserted.
        else if (key.compareTo(parent.getKey()) < 0)// left child
            parent.setLeftChild(inserted);
        else// right child
            parent.setRightChild(inserted);
        // set the children of the inserted node to be nil
        inserted.setLeftChild(nil);
        inserted.setRightChild(nil);
        insertFixup(inserted);// make sure that the red black tree properties are not violated.
    }
}
```

```
private void insertFixup(INode<T, V> node) {
    INode<T, V> grand;// the grandparent.
    INode<T, V> uncle;// the uncle.
    while (node.getParent().getColor() == RbNode.RED) {
        grand = node.getParent().getParent();
        if (node.getParent() == grand.getLeftChild()) {
            uncle = grand.getRightChild(); // uncle is the grandparent's right child.
            if (uncle != this.nil && uncle.getColor() == RbNode.RED) {// case 1 uncle is red, re color.
                node.getParent().setColor(RbNode.BLACK);
                uncle.setColor(RbNode.BLACK);
                grand.setColor(RbNode.RED);
                node = grand;
            }

            else {
                if (node == node.getParent().getRightChild()) {// case 2:the node is the right child and the parent
                                                               // is the left child so change to case 3.
                    node = node.getParent();
                    leftRotate(node);
                }
                node.getParent().setColor(RbNode.BLACK);// case 3.
                grand.setColor(RbNode.RED);
                rightRotate(grand);
            }
        } else {
            uncle = grand.getLeftChild(); // uncle is grandparent's left child.

            if (uncle != this.nil && uncle.getColor() == RbNode.RED) {// case 1 uncle is red, re color.
                node.getParent().setColor(RbNode.BLACK);
                uncle.setColor(RbNode.BLACK);
                grand.setColor(RbNode.RED);
                node = grand;
            }

            else {
                if (node == node.getParent().getLeftChild()) {// case 2: the node is the left child and the parent
                                                              // is the right child so change to case 3.
                    node = node.getParent();
                    rightRotate(node);
                }
                node.getParent().setColor(RbNode.BLACK);// case 3.
                grand.setColor(RbNode.RED);
                leftRotate(grand);
            }
```

```java
private void leftRotate(INode<T, V> x) {// rotate the subtree rooted at x anti clockwise.
    INode<T, V> y = x.getRightChild();
    x.setRightChild(y.getLeftChild());
    if (y.getLeftChild() != this.nil)
        y.getLeftChild().setParent(x);
    y.setParent(x.getParent());
    if (x.getParent() == this.nil)
        setRoot(y);
    else if (x == x.getParent().getLeftChild())
        x.getParent().setLeftChild(y);
    else
        x.getParent().setRightChild(y);
    y.setLeftChild(x);
    x.setParent(y);
}

private void rightRotate(INode<T, V> x) {// rotate the subtree rooted at x clockwise.
    INode<T, V> y = x.getLeftChild();
    x.setLeftChild(y.getRightChild());
    if (y.getRightChild() != this.nil)
        y.getRightChild().setParent(x);
    y.setParent(x.getParent());
    if (x.getParent() == this.nil)
        setRoot(y);
    else if (x == x.getParent().getLeftChild())
        x.getParent().setLeftChild(y);
    else
        x.getParent().setRightChild(y);
    y.setRightChild(x);
    x.setParent(y);
}

private void setRoot(INode<T, V> newRoot) {// only sets the root and its parent without changing its children (to be
    // assigned outside this method)
    this.root = newRoot;
    this.root.setColor(RbNode.BLACK);
    this.root.setParent(nil);
}
```

Delete Code:

```
private void deleteFixUp(INode<T, V> fixNode) {// a function performing the fixUp operation to maintain the red
                                               // black tree properties after deletion handling all the possible
                                               // cases
    INode<T, V> sibling;// the sibling of the node that requires fixing.
    while (fixNode != getRoot() && fixNode.getColor() == INode.BLACK) {// we terminate when we reach a red node or
                                                                       // the root of the tree because they don't
                                                                       // require fixing.
        if (fixNode == fixNode.getParent().getLeftChild()) {// the node is the left child
            sibling = fixNode.getParent().getRightChild();
            if (sibling.getColor() == INode.RED) {// case1 : sibling is red so both its children are black: re color
                                                  // and rotate to change to one of cases 2,3,4.
                sibling.setColor(INode.BLACK);// parent's color must be black
                fixNode.getParent().setColor(INode.RED);
                leftRotate(fixNode.getParent());
                sibling = fixNode.getParent().getRightChild();
            }

            if (sibling.getLeftChild().getColor() == INode.BLACK
                    && sibling.getRightChild().getColor() == INode.BLACK) {// case2 : the sibling and both its
                                                                           // children are black: re color and move
                                                                           // the problem to parent of the node so
                                                                           // the loop continues again.
                sibling.setColor(INode.RED);// we remove one black from the node and its sibling.
                fixNode = fixNode.getParent();
            }

            else {// case3: sibling and its right child are black: perform re coloring and
                    // rotation
                    // to transform to case4.
                if (sibling.getRightChild().getColor() == INode.BLACK) {
                    sibling.getLeftChild().setColor(INode.BLACK);
                    sibling.setColor(INode.RED);
                    rightRotate(sibling);
                    sibling = fixNode.getParent().getRightChild();
                } // case4: sibling is black with a red right child: we perform some re coloring
                    // and rotation that solves the black height problem by equating the number of
                    // black nodes on both sides leaving the node singly black and setting it to be
                    // the root for termination.
                sibling.setColor(fixNode.getParent().getColor());
                fixNode.getParent().setColor(INode.BLACK);
                sibling.getRightChild().setColor(INode.BLACK);
                leftRotate(fixNode.getParent());
                fixNode = getRoot();
            }
```

```java
public boolean delete(T key) {
    if (key == null) {
        throw new RuntimeErrorException(null);
    }
    INode<T, V> foundNode = getNode(key);// we search for the node by its given key.
    if (foundNode == nil) {// node wasn't found in the tree.
        return false;
    }
    INode<T, V> replacingNode = foundNode;// node to replace the original node to be deleted.
    INode<T, V> fixNode = new RbNode<T, V>();// node to be really deleted that undergoes deleteFixup.
    boolean originalColor = foundNode.getColor();
    if (foundNode.getLeftChild() == nil) {// only has right child or no children at all so just replace with its
                                          // right child.
        fixNode = foundNode.getRightChild();
        transplant(foundNode, foundNode.getRightChild());
    } else if (foundNode.getRightChild() == nil) {// only has the left child so replace with its left child.
        fixNode = foundNode.getLeftChild();
        transplant(foundNode, foundNode.getLeftChild());
    } else {// has two children so we get the successor replace with it and now we have to
            // delete the successor.
        replacingNode = successor(foundNode);
        originalColor = replacingNode.getColor();
        fixNode = replacingNode.getRightChild();// the successor is the minimum in the right subtree so it has no
                                                // left children.
        if (replacingNode.getParent() == foundNode) {// the successor was the direct child.
            fixNode.setParent(replacingNode);
        } else {
            transplant(replacingNode, replacingNode.getRightChild());
            replacingNode.setRightChild(foundNode.getRightChild());
            replacingNode.getRightChild().setParent(replacingNode);
        } // exchange the replacing node with the original found node to be deleted.
        transplant(foundNode, replacingNode);
        replacingNode.setLeftChild(foundNode.getLeftChild());
        replacingNode.getLeftChild().setParent(replacingNode);
        replacingNode.setColor(foundNode.getColor());
    }
    if (originalColor == INode.BLACK) {// here the red black tree properties may be violated as the black height
                                       // etc....
        deleteFixUp(fixNode);
    }
    size--;
    return true;
```

```java
private void transplant(INode<T, V> oldNode, INode<T, V> newNode) {// replace the old node with the new node without
    // setting its children (to be assigned from
    // outside this method).
    if (oldNode.getParent() == nil)
        setRoot(newNode);
    else if (oldNode == oldNode.getParent().getLeftChild())
        oldNode.getParent().setLeftChild(newNode);
    else
        oldNode.getParent().setRightChild(newNode);
    newNode.setParent(oldNode.getParent());
}
```

```
        } else {// same as previous but with exchanging left and right as here the node is the
                // right child.
            sibling = fixNode.getParent().getLeftChild();
            if (sibling.getColor() == INode.RED) {
                sibling.setColor(INode.BLACK);
                fixNode.getParent().setColor(INode.RED);
                rightRotate(fixNode.getParent());
                sibling = fixNode.getParent().getLeftChild();
            }

            if (sibling.getLeftChild().getColor() == INode.BLACK
                    && sibling.getRightChild().getColor() == INode.BLACK) {
                sibling.setColor(INode.RED);
                fixNode = fixNode.getParent();
            }

            else {
                if (sibling.getLeftChild().getColor() == INode.BLACK) {
                    sibling.getRightChild().setColor(INode.BLACK);
                    sibling.setColor(INode.RED);
                    leftRotate(sibling);
                    sibling = fixNode.getParent().getLeftChild();
                }
                sibling.setColor(fixNode.getParent().getColor());
                fixNode.getParent().setColor(INode.BLACK);
                sibling.getLeftChild().setColor(INode.BLACK);
                rightRotate(fixNode.getParent());
                fixNode = getRoot();
            }

        }
    }
    fixNode.setColor(INode.BLACK);// for case 2 and 4 (terminating cases) to make sure that the root is black and
                                  // that when the problem goes to the parent the parent is black not red.
}
```

Third MyTreeMap:

A class implementing the tree map interface using the red black tree data structure and utilizing its functions, providing the following procedures.

- CeilingEntry,ceilingKey,floorEntry and floorKey: using an algorithm like the one used in the search procedure while updating the ceil and the floor for the given key and then finally returns an entry containing the key and value required (ceil or floor) as required O(lgn).

- containsValue: performs in order traversal to look for the required value in each node O(n).
- entrySet: performs in order traversal on the tree forming entries from each node so the returned set is sorted (due to in order traversal of a BST) O(n).
- firstEntry,firstKey,lastEntry and lastKey: they return an entry or a key corresponding to the minimum or maximum key in the map respectively O(n).
- headMap: returns an array list filled with all the nodes corresponding to the keys till a given key (inclusive or not as specified) using inorder traversal and filling only the correct nodes so the array list is sorted O(n).
- keyset: returns a sorted set of the keys in the map using in order traversal O(n).
- pollFirstEntry,pollLastEntry: returns and removes the minimum or maximum key associated entry O(lg n).
- put: inserts a key value mapping entry into the map O(lgn).
- Put all: takes a map and puts all the key value mapping entries into the map O(nlgn).
- Remove: removes the entry associated with the given key from the map O(lgn).
- Values: returns a sorted collection (by keys not value) of the values in the map using in order traversal O(n).

Code Snippets:

```java
@Override
public Entry<T, V> pollFirstEntry() {
    if (rbTree.isEmpty())
        return null;
    INode<T, V> largest = rbTree.minimum(rbTree.getRoot());
    rbTree.delete(largest.getKey());
    return new AbstractMap.SimpleEntry<T, V>(largest.getKey(), largest.getValue());
}

@Override
public Entry<T, V> pollLastEntry() {
    if (rbTree.isEmpty())
        return null;
    INode<T, V> largest = rbTree.maximum(rbTree.getRoot());
    rbTree.delete(largest.getKey());
    return new AbstractMap.SimpleEntry<T, V>(largest.getKey(), largest.getValue());
}

@Override
public void put(T key, V value) {
    rbTree.insert(key, value);

}

@Override
public void putAll(Map<T, V> map) {
    if (map == null)
        throw new RuntimeErrorException(null);
    for (Entry<T, V> entry : map.entrySet()) {
        if (entry != null)
            rbTree.insert(entry.getKey(), entry.getValue());
    }

}

@Override
public boolean remove(T key) {
    return rbTree.delete(key);
}
```

```java
@Override
public ArrayList<Entry<T, V>> headMap(T toKey, boolean inclusive) {
    ArrayList<Entry<T, V>> values = new ArrayList<Entry<T, V>>();
    if (toKey == null)
        throw new RuntimeErrorException(null);
    fillEntriesInorder(values, rbTree.getRoot(), toKey, inclusive);
    return values;
}

private void fillEntriesInorder(Collection<Entry<T, V>> values, INode<T, V> node, T toKey, boolean inclusive) {
    if (node == rbTree.nil)
        return;
    fillEntriesInorder(values, node.getLeftChild(), toKey, inclusive);
    if ((node.getKey().compareTo(toKey) < 0 || (node.getKey() == toKey && inclusive)))
        values.add(new AbstractMap.SimpleEntry<T, V>(node.getKey(), node.getValue()));
    fillEntriesInorder(values, node.getRightChild(), toKey, inclusive);
}

@Override
public Set<T> keySet() {
    if (rbTree.isEmpty())
        throw new RuntimeErrorException(null);
    Set<T> values = new LinkedHashSet<T>();
    fillKeysInorder(values, rbTree.getRoot());
    return values;
}

private void fillKeysInorder(Collection<T> values, INode<T, V> node) {
    if (node == rbTree.nil)
        return;
    fillKeysInorder(values, node.getLeftChild());
    values.add(node.getKey());
    fillKeysInorder(values, node.getRightChild());
}
```

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

    private MyRedBlackTree<T, V> rbTree = new MyRedBlackTree<T, V>();// the tree data structure to be used which is the
                                                                     // implemented red black tree.
    @Override
    public Entry<T, V> ceilingEntry(T key) {
        if (key == null)
            throw new RuntimeErrorException(null);
        INode<T, V> node = rbTree.nil;
        INode<T, V> traversal = rbTree.getRoot();
        while (traversal != rbTree.nil) {// traverse the tree while updating the ceil value if we go to the left
                                         // subtree.
            if (key.compareTo(traversal.getKey()) > 0) {
                traversal = traversal.getRightChild();
            } else if (key.compareTo(traversal.getKey()) < 0) {
                node = traversal;
                traversal = traversal.getLeftChild();
            } else {
                return new AbstractMap.SimpleEntry<T, V>(traversal.getKey(), traversal.getValue());
            }

        }
        if (node != rbTree.nil)
            return new AbstractMap.SimpleEntry<T, V>(node.getKey(), node.getValue());
        return null;

    }

    @Override
    public T ceilingKey(T key) {
        Entry<T, V> found = ceilingEntry(key);
        if (found == null)
            return null;
        return found.getKey();
    }

    @Override
    public void clear() {
        rbTree.clear();
    }
```

```java
@Override
public ArrayList<Entry<T, V>> headMap(T toKey) {
    ArrayList<Entry<T, V>> values = new ArrayList<Entry<T, V>>();
    if (toKey == null)
        throw new RuntimeErrorException(null);
    fillEntriesInorder(values, rbTree.getRoot(), toKey, false);
    return values;
}

@Override
public ArrayList<Entry<T, V>> headMap(T toKey, boolean inclusive) {
    ArrayList<Entry<T, V>> values = new ArrayList<Entry<T, V>>();
    if (toKey == null)
        throw new RuntimeErrorException(null);
    fillEntriesInorder(values, rbTree.getRoot(), toKey, inclusive);
    return values;
}

private void fillEntriesInorder(Collection<Entry<T, V>> values, INode<T, V> node, T toKey, boolean inclusive) {
    if (node == rbTree.nil)
        return;
    fillEntriesInorder(values, node.getLeftChild(), toKey, inclusive);
    if ((node.getKey().compareTo(toKey) < 0 || (node.getKey() == toKey && inclusive)))
        values.add(new AbstractMap.SimpleEntry<T, V>(node.getKey(), node.getValue()));
    fillEntriesInorder(values, node.getRightChild(), toKey, inclusive);
}
```

```java
@Override
public boolean containsKey(T key) {
    return rbTree.contains(key);
}

@Override
public boolean containsValue(V value) {// will have to traverse the tree which is O(n).
    if (value == null)
        throw new RuntimeErrorException(null);
    Collection<V> checker = new ArrayList<V>();
    fillValuesInorder(checker, rbTree.getRoot());// still O(n) if we traverse the whole tree to check for the value.
    return checker.contains(value);
}

@Override
public Set<Entry<T, V>> entrySet() {
    if (rbTree.isEmpty())
        throw new RuntimeErrorException(null);
    Set<Entry<T, V>> values = new LinkedHashSet<Entry<T, V>>();
    fillEntriesInorder(values, rbTree.getRoot());
    return values;
}

private void fillEntriesInorder(Collection<Entry<T, V>> values, INode<T, V> node) {
    if (node == rbTree.nil)
        return;
    fillEntriesInorder(values, node.getLeftChild());
    values.add(new AbstractMap.SimpleEntry<T, V>(node.getKey(), node.getValue()));
    fillEntriesInorder(values, node.getRightChild());
}

@Override
public Entry<T, V> firstEntry() {
    if (rbTree.isEmpty())
        return null;
    INode<T, V> node = rbTree.minimum(rbTree.getRoot());
    return new AbstractMap.SimpleEntry<T, V>(node.getKey(), node.getValue());
}

@Override
public T firstKey() {
    if (rbTree.isEmpty())
        return null;
```

```java
@Override
public Collection<V> values() {
    Collection<V> returned = new ArrayList<V>();
    fillValuesInorder(returned, rbTree.getRoot());
    return returned;
}

private void fillValuesInorder(Collection<V> values, INode<T, V> node) {
    if (node == rbTree.nil)
        return;
    fillValuesInorder(values, node.getLeftChild());
    values.add(node.getValue());
    fillValuesInorder(values, node.getRightChild());
}
```

Assumptions:

- The map procedures are done via the red black tree injected procedures except for the algorithms that require traversing the whole tree are written entirely manipulating the tree connections.
- The tree insert and delete algorithms were done via the help of Cormen's introduction to algorithms.
- The tree.nil node is implemented in the tree class as a node that is initialized on tree construction having a black color, null value, null key and null connections to parent and children.
- The implementation of the tree.nil helps in the deletion algorithm (in transplant algorithm).
- At first the implementation of the entry was via a made class (MyEntry) but then abstract map simple entry was finally used.