Name/Makrm Wiiliam

Id/64

Lab 2

-tests:

```java
import org.junit.runner.JUnitCore;
public class MainTester {

    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(IntegrationTest.class);
        Result result2 = JUnitCore.runClasses(UnitTest.class);

        int totalNumOfTests = result.getRunCount() + result2.getRunCount();
        int totalFailures = result.getFailureCount() + result2.getFailureCount();

        System.out.println("Total tests passed: " + (totalNumOfTests - totalFailures) + "/" + t

        ArrayList<Failure> failures = new ArrayList<>();
        failures.addAll(result.getFailures());
        failures.addAll(result2.getFailures());

        for (Failure failure : failures) {
            System.out.println(failure.toString());
        }

    }
}
```

Console | Problems | Debug Shell | Call Hierarchy | Libraries | Coverage

```
<terminated> MainTester (2) [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (Apr 13, 2020, 3:42:17 PM)
field name interface eg.edu.alexu.csd.filestructure.redblacktree.IRedBlackTree
field name int
field name class java.util.ArrayList
field name long
field name int
field name class [Ljava.lang.Object;
field name class [Ljava.lang.Object;
field name class [Ljava.lang.Object;
field name int
field name int
Total tests passed: 71/71
```

-snapshots code:

-For INode we made an inner class in RedBlacktree class and implements all INode methods.

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

    public class Node<T extends Comparable<T>, V> implements INode {
      private Node parent,lefChild,rightChild;
      T key;
      V value;
      boolean black;
      boolean isleft;
      public Node(T key, V value) {
    this.key = key; this.value=value;
    parent = lefChild = rightChild =null;
    black = isleft =false;
    }
```

-For RedBlacktree class we make a nill Node to determine the null leaf in the tree with null key,null value and black color.

We also use root with nill at begening and when it empty so check it’s empty or not by check the root .

To clear the tree we used a recucisve methods to make all nodes in the tree to null .

```
}
private Node nill = new Node(null, null);

private Node root=nill ;
private int size=0;
private boolean update;
public RedBlackTree() {
  nill.black=true;  root =nill; size=0;
}

@Override
public INode getRoot() {
    return root;
}

@Override
public boolean isEmpty() {
    return (root == nill);
}
public int size() {
    return size;
}

@Override
public void clear() {
 clear(root);

}



private void clear(Node root2) { size=0;
    if(root2 == nill) {
        root=nill; return;
    }
    clear((Node) root2.getLeftChild());
    clear((Node) root2.getRightChild());
}
```

-For search and contian we make another methods to search and return node if it exsists and call it in search,contian.

```java
@Override
public Object search(Comparable key) { if(key == null) throw  new RuntimeErrorException(null);

INode node = searchNode(key);
if(node == null) return null;
    return node.getValue();
}

@Override
public boolean contains(Comparable key) { if(key == null) throw  new RuntimeErrorException(null);
return(search(key) != null );
}

private INode searchNode(Comparable key) { if(key == null) throw  new RuntimeErrorException(null);
INode node = root;
while(node != null && node.getKey()!= null ) {
    if(key.compareTo(node.getKey()) == 0 ) return node;
    else if(key.compareTo(node.getKey())>0) node= node.getRightChild();
    else node =node.getLeftChild();
}
return null;
```

-For insert: we need two recursive methods one for add node in its place by comparing keys,another for check the balanced Redblack tree by check if their any two red node in the tree untill root.

Then trying to correct it by rotate or swipe color.

```
public void insert(Comparable key, Object value) {  if(key == null || value ==null) throw  new RuntimeErrorException(null);
     if(root == nill || root ==null) {
        root = new Node( key, (V) value);
        root.setLeftChild(nill); root.setRightChild(nill);
        size++; root.black=true;
        return ;
    } Node node = new Node(key, value);
    update=false;
    add(root,node) ;
   node.setLeftChild(nill); node.setRightChild(nill);
if ( !update ) {
   CheckColor(node);
   getRoot().setColor(true);
   }
}

private void add(Node parent, Node node) {
    if(node.key.compareTo(parent.getKey()) == 0 ) {
        parent.setValue(node.value); update=true; return ; }
   else if(node.key.compareTo(parent.getKey()) > 0 ) {
        if(parent.getRightChild()== null || parent.getRightChild()==nill ) {
            node.parent=parent;
            parent.rightChild=node;
            size++; return ;
        } add((Node) parent.getRightChild(), node);
    } else {
        if(parent.getLeftChild() == null || parent.getLeftChild() == nill) {
        node.setParent(parent);
        parent.setLeftChild(node);
        node.isleft= true;
        size++; return ;
        }
        add((Node) parent.getLeftChild(),node);
    }
}

private void CheckColor(Node node) {
  if(node == root ) return;
  if(!node.black && !node.parent.black) {
   CorrecetColor(node);
  } if(node.parent == null || node.parent == nill) return ;
  CheckColor(node.parent);
```

```
private void Rotate(Node node) {
    if(node.isleft) {
        if(node.parent.isleft) {
    RightRotate(node.parent.parent);
    node.black=false;
    node.parent.black=true;
    if(node.parent.rightChild.isNull() ==false ) node.parent.rightChild.setColor(false);
        }else {
            RightLeftRotate(node.parent.parent);
    node.black=true;
    node.lefChild.setColor(false);
    node.rightChild.setColor(false);
        }
    return ;
    }else {
        if(node.parent.isleft == false) {
            LeftRotate(node.parent.parent);
            node.setColor(false);
            node.parent.setColor(true);
            if(node.parent.lefChild.isNull() == false) node.parent.lefChild.setColor(false);
        }else {
            LeftRightRotat(node.parent.parent);
            node.setColor(true);
            node.lefChild.setColor(false);
            node.rightChild.setColor(false);
        }
    }
}
```

-For delete: if node exisits and has two children, swape it with Min right chlidren and delet this leaf node.

Then checking the balnced tree if left black chlirden = right black children or not and trying correcting it.

```java
@Override
public boolean delete(Comparable key) { if(key == null ) throw  new RuntimeErrorException(null);

    if (root == nill)
        return false;

    INode<T,V> node = searchNode(key);

    if(node == nill || node == null)
        return false;
    size--;
    if (node == root && node.getLeftChild() == nill && node.getRightChild() == nill) {
        root = nill;
        return true;
    }
    if(node.getRightChild() == nill) {
        INode<T,V>node2 = deletleaf(node);
        if(!node.getColor())
            correctdel(node2);
        return true;
    }
    INode<T,V> leafnode = Min(node.getRightChild());
    node.setKey(leafnode.getKey());
    node.setValue(leafnode.getValue());
    node = leafnode;
    INode<T,V>sibling = deletleaf(node);
    if(!node.getColor())
        correctdel(sibling);

    return true;
}
```

the time analysis:

we can see that as we make our tree balanced by using RedBlack tree rules.

So insert, search, delete all take O(logn), as in every method we ignore half of tree by going left or right.

Same as implemented functions:

ceilingEntry, ceilingKey, containsKey, firstEntry, firstEntry, floorEntry, floorKey, lastEntry, lastKey, pollFirstElement, pollLastEntry, put, remove.

For putall it will need O(nlogn), as every put take O(logn).

For entrySet it will need O(n), as travers on the whole tree with inorder.

Same as implemented functions:

Clear, containsValue, headMap, headMap, keyset, values.