RED BLACK TREE LAB

Team member 1 : Febronia Ashraf ID: 29

Team member 2 : Neveen Samir ID: 58

1- Problem Statement →

1-RedblackTree: 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.

2- 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.

2- Algorithm Description →

1- Red Black Tree :

- **getRoot: return the root of the given Red black tree.**
- **isEmpty: return whether the given tree isEmpty or not by checking if the root is null 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.**

2- Tree Map:

- **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.

3- Assumptions →

In IRed Black Tree Implementation :
If key or value of node that the user want to insert to the tree are null then the program will throw exception for that.
If the key that the user want to search about or know if the tree contains that key is null then the program will throw exception from that.
If the user want to delete an key from the tree and enter a null key the program will throw exception for that.
In ITree Map :
If the user enter a null key or map or entry to the functions then the program will throw exception for that.

4- Pseudo Code →

Red Black Tree :

Search :

```
search(T key)
  node = T.root;
  if (key = null)
    retrun;
  endIF;
  while (!node.isNull())
            if (node.getKey=key)
               break;
             else if (key>node.getKey())
                        node = node.getRightChild();
             else
                        node = node.getLeftChild();
            endIF;
  endWhile;
  return node.getValue();
END;
```

Contains :

```
contains(T key)
  if(key = null)
      retrun;
  endIF;
  Node = T.root;
  while(!node.isNull())
            if(node.getKey()=key)
                        return true;
            else if(node.getKey()<key)
                        node = node.getRightChild();
            else
                        node = node.getLeftChild();
            endIF;
  endWhile;
  return false;
  END;
```

Insert :

```
insert(z)
 if (key = null OR value = null) {
     return;
 endIF;
 x =T. root;
 while (!x.isNull())
                        y = x;
                        if(y.getKey()=z.key())
                                    y.setValue(z.getValue());
```

```
                    return;
            else if (z.getKey().>y.key)
                    x = x.getRightChild();
             else
                    x = x.getLeftChild();
            endIF;
        endWhile;
    z.setParent(y);
    z.setLeftChild(NIL);
    z.setRightChild(NIL);
    if (y.isNull())
        root = z;
    else if (z.getKey()>y.key()) {
        y.setRightChild(z);
    else
        y.setLeftChild(z);
    endIF;
    if(z=root)
        z.setColor(false);
        root.setColor(false);
    else
    z.setColor(true);
    endIF;
    if(z.getParent().getColor() = RED)
    checkColor(z);
    endIF;
   END;
```

Check Color :

```
checkColor(z)
while(z.getParent().getColor()==RED)
    if(z.getParent() Is (z.getParent().getParent().getLeftChild()))
        uncle = z.getParent().getParent().getRightChild();
        if(uncle.getColor()&&!uncle.isNull())
            uncle.setColor(Black);
            z.getParent().setColor(Black);
            z.getParent().getParent().setColor(Red);
            z = z.getParent().getParent();
        else if(z.equals(z.getParent().getRightChild()))
            z = z.getParent();
            leftRotate(z);
        else
            z.getParent().setColor(Black);
            z.getParent().getParent().setColor(Red);
            rightRotate(z.getParent().getParent());
        endIF;
    else
        uncle = z.getParent().getParent().getLeftChild();
        if(uncle.getColor()&&!uncle.isNull())
            uncle.setColor(Black);
            z.getParent().setColor(Black);
            z.getParent().getParent().setColor(Red);
            z = z.getParent().getParent();
        else if(z.equals(z.getParent().getLeftChild()))
            z = z.getParent();
            rightRotate(z);
        else
            z.getParent().setColor(Black);
            z.getParent().getParent().setColor(Red);
            leftRotate(z.getParent().getParent());
        endIF;
    endIF;
endWhile;
T.root.setColor(false);
END;
```

Delete :

```
delete(T key)
  if (key = null)
                return;
        endIF;
        if(!contains(key))
                return false;
        endIF;
        node = T.root;
        while (!node.isNull())
                if (node.getKey()=key)
                        deleteNode(node);
                        break;
                else if (key.>node.Key())
                        node = node.getRightChild();
                 else
                        node = node.getLeftChild();
                endIF;
        endWhile;
        return true;
       END;
```

Delete Node :

```
deleteNode(z)
 y = z;
 color  = 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 = Minimum(z.getRightChild());
         color = 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());
 endIF;
 if(color Is Black)
        fixDelete(x);
 endIF;
 END;
```

Fix Delete :

```
fixDelete(x)
 while(!x.equals(root) && !x.getColor())
        if(x.getParent().getLeftChild().equals(x))
                w = x.getParent().getRightChild();
                if(w.getColor())
                        w.setColor(Black);
                        x.getParent().setColor(Red);
                        leftRotate(x.getParent());
                        w = x.getParent().getRightChild();
                        endIf;
                        if(!w.getLeftChild().getColor() && !w.getRightChild().getColor())
                                w.setColor(Red);
                                x = x.getParent();
                                continue;
                        else if(!w.getRightChild().getColor() &&w.getLeftChild().getColor())
                                w.getLeftChild().setColor(Black);
                                w.setColor(Red);
                                rightRotate(w);
                                w = x.getParent().getRightChild();
                        endIf;
                        if(w.getRightChild().getColor())
                        w.setColor(x.getParent().getColor());
                        x.getParent().setColor(Black);
                        w.getRightChild().setColor(Black);
                        leftRotate(x.getParent());
                        x = root;
                        endIF;

                 else
                w = x.getParent().getLeftChild();
                if(w.getColor())
                        w.setColor(Black);
                        x.getParent().setColor(Red);
                        rightRotate(x.getParent());
                        w = x.getParent().getLeftChild();
                        endIF;
                        if(!w.getRightChild().getColor() && !w.getLeftChild().getColor())
                                w.setColor(Red);
                                x = x.getParent();
                                continue;
                        else if(!w.getLeftChild().getColor() && w.getRightChild().getColor())
                                w.getRightChild().setColor(Black);
                                w.setColor(true);
                                leftRotate(w);
                                w = x.getParent().getLeftChild();
                        endIF;
                        if(w.getLeftChild().getColor())
                        w.setColor(x.getParent().getColor());
                        x.getParent().setColor(Black);
                        w.getLeftChild().setColor(Black);
                        rightRotate(x.getParent());
                        x = root;
                        endIF;

        endIF;
 endWhile;
 x.setColor(Black);
END;
```

5- **Snipping** →

INode :

RedBlackTree.java | TreeMap.java

```
559     private class Node<T extends Comparable<T>, V> implements INode<T, V> {
560
561           private INode<T,V> parent ;
562            private INode<T,V> leftChild;
563            private INode<T,V> rightChild ;
564            private T key ;
565            private V value ;
566            private boolean color = false;
567            public Node (INode<T,V> parent,INode<T,V> left,INode<T,V> right,T key,V value,boolean color) {
568                this.parent = parent;
569                this.leftChild = left;
570                this.rightChild = right;
571                this.key = key;
572                this.value = value;
573                this.color = color;
574             }
575         @Override
576         public void setParent(INode<T,V> parent) {
577             // TODO Auto-generated method stub
578             this.parent = parent;
579         }
580
581         @Override
582         public INode<T,V> getParent() {
583             // TODO Auto-generated method stub
584             return parent;
585         }
586
587         @Override
588         public void setLeftChild(INode<T,V> leftChild) {
589             // TODO Auto-generated method stub
590             this.leftChild = leftChild;
591         }
592
593         @Override
594         public INode<T,V> getLeftChild() {
595             // TODO Auto-generated method stub
596             return leftChild;
```

RedBlackTree.java | TreeMap.java

```
613             // TODO Auto-generated method stub
614             return key;
615         }
616
617         @Override
618         public void setKey(T key) {
619             // TODO Auto-generated method stub
620             this.key = key;
621         }
622
623         @Override
624         public V getValue() {
625             // TODO Auto-generated method stub
626             return value;
627         }
628
629         @Override
630         public void setValue(V value) {
631             // TODO Auto-generated method stub
632             this.value = value;
633         }
634
635         @Override
636         public boolean getColor() {
637             // TODO Auto-generated method stub
638             return color;
639         }
640
641         @Override
642         public void setColor(boolean color) {
643             // TODO Auto-generated method stub
644             this.color = color;
645         }
646
647         @Override
648         public boolean isNull() {
649             // TODO Auto-generated method stub
650             if(key == null && value == null) {
```

IRed Black Tree :

RedBlackTree.java | TreeMap.java

```java
    public INode<T,V> getRoot() {

        return root;
    }


    @Override
    public boolean isEmpty() {
        // TODO Auto-generated method stub
        if (root.isNull()) {
            return true;
        }
        return false;
    }

    @Override
    public void clear() {
        // TODO Auto-generated method stub
        root = new Node<T, V>(null,null,null,null,null,false);
    }
    @Override
    public V search(T key) {
        // TODO Auto-generated method stub
        if (key == null) {
            throw new RuntimeErrorException(null);
        }
        INode<T,V> node = root;
        while (!node.isNull()) {
            if (node.getKey().compareTo((T) key) == 0) {
                break;
            } else if (key.compareTo(node.getKey()) > 0) {
                node = node.getRightChild();
            } else {
                node = node.getLeftChild();
            }
        }
        return node.getValue();
    }
```

*RedBlackTree.java | TreeMap.java

```java
    public T ceilingKey(T key) {
        node = root;
        while (!node.isNull()) {
            if (node.getKey().compareTo(key) == 0) {
                return (T) node.getKey();
            } else if (key.compareTo(node.getKey()) > 0) {
                node = node.getRightChild();
            } else {
                if (node.getLeftChild().isNull()) {
                    return (T) node.getKey();
                } else {
                    node = node.getLeftChild();
                }
            }
        }
        if( node == root && (key.compareTo(node.getKey()) < 0)) {
            return (T) root.getKey();
        }

        return null;
    }
    public Map.Entry<T, V> ceilingEntry(T key) {
        Map.Entry<T, V> e ;
        node = getRoot();
        while (!node.isNull()) {
            if (node.getKey().equals(key)) {
                e= new AbstractMap.SimpleEntry<T, V>((T)node.getKey(),(V) node.getValue());
                return e;
            } else if (key.compareTo(node.getKey()) > 0) {
                node = node.getRightChild();
            } else {
                if (node.getLeftChild().isNull()) {
                    e= new AbstractMap.SimpleEntry<T, V>((T)node.getKey(),(V) node.getValue());
                    return e;
                } else {
                    node = node.getLeftChild();
                }
            }
```

*RedBlackTree.java | TreeMap.java

```java
    public Map.Entry<T, V> floorEntry(T key) {
        Map.Entry<T, V> e;
        INode<T,V> node = root;
        while (!node.isNull()) {
            if (node.getKey().equals(key)) {
                e= new AbstractMap.SimpleEntry<T, V>((T)node.getKey(),(V) node.getValue());
                return e;
            } else if (key.compareTo(node.getKey()) > 0) {
                if (node.getRightChild().isNull()) {
                    e= new AbstractMap.SimpleEntry<T, V>((T)node.getKey(),(V) node.getValue());
                            return e;
                }
                node = node.getRightChild();
            } else {
                node = node.getLeftChild();
            }
        }

        return null;
    }

    @Override
    public boolean contains(T key) {
        // TODO Auto-generated method stub
        if(key == null) {
            throw new RuntimeErrorException(null);
        }
        INode<T,V> node = root;
        while(!node.isNull()) {
            if(node.getKey().compareTo(key) == 0) {
                return true;
            }else if(node.getKey().compareTo(key) < 0) {
                node = node.getRightChild();
            }else {
                node = node.getLeftChild();
            }
        }
        return false;
```

*RedBlackTree.java | TreeMap.java

```java
    public void insert(T key, V value) {
        if (key == null || value == null) {
            throw new RuntimeErrorException(null);
        }
        INode<T,V> z = new Node<T,V>(null,null,null,key,value,true);
        INode<T,V> y = new Node<T,V>(null, null, null, null, null, false);
        INode<T,V> x = root;
        while (!x.isNull()) {
                y = x;
                if(y.getKey().compareTo(z.getKey())==0) {
                    y.setValue(z.getValue());
                    return;
                }
                else if (z.getKey().compareTo(x.getKey()) > 0) {
                    x = x.getRightChild();
                } else {
                    x = x.getLeftChild();
                }
            }
        z.setParent(y);
        z.setLeftChild(new Node<T,V>(null, null, null, null, null, false));
        z.setRightChild(new Node<T,V>(null, null, null, null, null, false));
        if (y.isNull()) {
            root = z;
        } else if (z.getKey().compareTo(y.getKey()) > 0) {
            y.setRightChild(z);
        } else {
            y.setLeftChild(z);
        }
        if(z==root) {
            z.setColor(false);
            root.setColor(false);
        }else {
        z.setColor(true);
        }
        z.setLeftChild(new Node<T,V>(null, null, null, null, null, false));
        z.setRightChild(new Node<T,V>(null, null, null, null, null, false));
        if(z.getParent().getColor()) {
```

*RedBlackTree.java | TreeMap.java

```
    private void checkColor(INode<T,V> z) {
        INode<T,V> uncle = new Node<T,V>(null,null,null,null,null,false);
        while(z.getParent().getColor()) {
            if(z.getParent().equals(z.getParent().getParent().getLeftChild())) {
                uncle = z.getParent().getParent().getRightChild();
                if(uncle.getColor()&&!uncle.isNull()) {
                    uncle.setColor(false);
                    z.getParent().setColor(false);
                    z.getParent().getParent().setColor(true);
                    z = z.getParent().getParent();
                }else if(z.equals(z.getParent().getRightChild())) {
                    z = z.getParent();
                    leftRotate(z);
                }else {
                    z.getParent().setColor(false);
                    z.getParent().getParent().setColor(true);
                    rightRotate(z.getParent().getParent());
                }
            }else {
                uncle = z.getParent().getParent().getLeftChild();
                if(uncle.getColor()&&!uncle.isNull()) {
                    uncle.setColor(false);
                    z.getParent().setColor(false);
                    z.getParent().getParent().setColor(true);
                    z = z.getParent().getParent();
                }else if(z.equals(z.getParent().getLeftChild())) {
                    z = z.getParent();
                    rightRotate(z);
                }else {
                    z.getParent().setColor(false);
                    z.getParent().getParent().setColor(true);
                    leftRotate(z.getParent().getParent());
                }
            }
        }
        root.setColor(false);
    }
```

*RedBlackTree.java | TreeMap.java

```

    @Override
    public boolean delete(T key) {
        // TODO Auto-generated method stub

        if (key == null) {
                    throw new RuntimeErrorException(null);
                }
                if(!contains(key)) {
                    return false;
                }
                size--;
                INode<T,V> node = root;
                while (!node.isNull()) {
                    if (node.getKey().compareTo((T) key) == 0) {
                        deleteNode(node);
                        break;
                    } else if (key.compareTo(node.getKey()) > 0) {
                        node = node.getRightChild();
                    } else {
                        node = node.getLeftChild();
                    }
                }

                return true;
    }
    private void transPlant(INode<T,V> u, INode<T,V> v) {
        if(u.getParent().isNull()) {
            root = v;
        }else if(u.getParent().getLeftChild().equals(u)) {
            u.getParent().setLeftChild(v);
        }else {
            u.getParent().setRightChild(v);
        }
        v.setParent(u.getParent());
    }

```

```
    }

    private void deleteNode(INode<T,V> z) {
        INode<T,V> x = new Node<T,V>(null,null,null,null,null,false);
        INode<T,V> y = z;
        boolean color  = 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 = Minimum(z.getRightChild());
             color = 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(!color) {
            fixDelete(x);
        }
    }
```

*RedBlackTree.java TreeMap.java

```
414    private void fixDelete(INode<T,V> x) {
415        INode<T,V> w = new Node<T,V>(null,null,null,null,null,false);
416        while(!x.equals(root) && !x.getColor()) {
417            if(x.getParent().getLeftChild().equals(x)) {
418                w = x.getParent().getRightChild();
419                if(w.getColor()) {
420                    w.setColor(false);
421                    x.getParent().setColor(true);
422                    leftRotate(x.getParent());
423                    w = x.getParent().getRightChild();
424                    }
425                    if(!w.getLeftChild().getColor() && !w.getRightChild().getColor()) {
426                        w.setColor(true);
427                        x = x.getParent();
428                        continue;
429                    }else if(!w.getRightChild().getColor() &&w.getLeftChild().getColor()){
430                        w.getLeftChild().setColor(false);
431                        w.setColor(true);
432                        rightRotate(w);
433                        w = x.getParent().getRightChild();
434                    }
435                    if(w.getRightChild().getColor()) {
436                    w.setColor(x.getParent().getColor());
437                    x.getParent().setColor(false);
438                    w.getRightChild().setColor(false);
439                    leftRotate(x.getParent());
440                    x = root;
441                    }
442
443            }else {
444                w = x.getParent().getLeftChild();
445                if(w.getColor()) {
446                    w.setColor(false);
447                    x.getParent().setColor(true);
448                    rightRotate(x.getParent());
449                    w = x.getParent().getLeftChild();
450                    }
451                    if(!w.getRightChild().getColor() && !w.getLeftChild().getColor()) {
```

```
                    }

            }else {
                w = x.getParent().getLeftChild();
                if(w.getColor()) {
                    w.setColor(false);
                    x.getParent().setColor(true);
                    rightRotate(x.getParent());
                    w = x.getParent().getLeftChild();
                    }
                    if(!w.getRightChild().getColor() && !w.getLeftChild().getColor()) {
                        w.setColor(true);
                        x = x.getParent();
                        continue;
                    }else if(!w.getLeftChild().getColor() && w.getRightChild().getColor()){
                        w.getRightChild().setColor(false);
                        w.setColor(true);
                        leftRotate(w);
                        w = x.getParent().getLeftChild();
                    }
                    if(w.getLeftChild().getColor()) {
                    w.setColor(x.getParent().getColor());
                    x.getParent().setColor(false);
                    w.getLeftChild().setColor(false);
                    rightRotate(x.getParent());
                    x = root;
                    }

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

*RedBlackTree.java TreeMap.java

```
473    private void rightRotate(INode<T,V> x) {
474        if(!x.isNull()) {
475            INode<T,V> y = x.getLeftChild();
476            x.setLeftChild(y.getRightChild());
477            if(!y.getRightChild().isNull()) {
478                y.getRightChild().setParent(x);
479            }
480            y.setParent(x.getParent());
481            if(x.getParent().isNull()) {
482                root = y;
483            }else if(x.equals(x.getParent().getLeftChild())) {
484                x.getParent().setLeftChild(y);
485            }else {
486                x.getParent().setRightChild(y);
487            }
488            y.setRightChild(x);
489            x.setParent(y);
490            }
491    }
492    private void leftRotate(INode<T,V> x) {
493        if(!x.isNull()) {
494        INode<T,V> y = x.getRightChild();
495        x.setRightChild(y.getLeftChild());
496        if(!y.getLeftChild().isNull()) {
497            y.getLeftChild().setParent(x);
498        }
499        y.setParent(x.getParent());
500        if(x.getParent().isNull()) {
501            root = y;
502        }else if(x.equals(x.getParent().getLeftChild())) {
503            x.getParent().setLeftChild(y);
504        }else {
505            x.getParent().setRightChild(y);
506        }
507        y.setLeftChild(x);
508        x.setParent(y);
509        }
510        }
```

```
    }
    private INode<T, V> Minimum(INode<T, V> node) {
        while(!node.getLeftChild().isNull()) {
            node = node.getLeftChild();
        }
        return node;
    }
}
```

TTree Map :

*RedBlackTree.java | TreeMap.java

```java
    @Override
    public Map.Entry<T, V> ceilingEntry(T key) {
        if(key == null) {
            throw new RuntimeErrorException(null);
        }
        return RBTree.ceilingEntry(key);
    }

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

    @Override
    public void clear() {
        RBTree = new RedBlackTree<>();

    }

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

    @Override
    public boolean containsValue(Object value) {
        if(value== null) {
            throw new RuntimeErrorException(null);
        }
        return RBTree.containsValue(value);
    }
```

*RedBlackTree.java | TreeMap.java

```java
    @Override
    public Set<Map.Entry<T, V>> entrySet() {
        return  RBTree.setMap();
    }

    @Override
    public Map.Entry<T, V> firstEntry() {
      return RBTree.Minimum();
    }

    @Override
    public T firstKey() {
        Entry<T, V> e = RBTree.Minimum();
        if(e == null) {
            return null;
        }
        return  (T) e.getKey();
    }

    @Override
    public Map.Entry<T, V> floorEntry(T key) {
        if(key == null) {
            throw new RuntimeErrorException(null);
        }
        return RBTree.floorEntry(key);
    }

    @Override
    public T floorKey(T key) {
        if(key == null) {
            throw new RuntimeErrorException(null);
        }
        Entry<T, V> e = RBTree.floorEntry(key);
        return  (T) e.getKey();
    }
```

*RedBlackTree.java | TreeMap.java

```
104     @Override
105     public ArrayList<Map.Entry<T, V>> headMap(T toKey) {
106         if(toKey == null) {
107             throw new RuntimeErrorException(null);
108         }
109         ArrayList<Map.Entry<T, V>> list = RBTree.heapMap(toKey);
110         Map.Entry<T, V> e = list.get(list.size()-1);
111         if(e.getKey().equals(toKey)) {
112             list.remove(list.size()-1);
113         }
114         return list;
115     }
116
117     @Override
118     public ArrayList<Map.Entry<T, V>> headMap(T toKey, boolean inclusive) {
119         if(toKey == null) {
120             throw new RuntimeErrorException(null);
121         }
122         ArrayList<Map.Entry<T, V>> list = RBTree.heapMap(toKey);
123         Map.Entry<T, V> e = list.get(list.size()-1);
124         if(!inclusive && e.getKey().equals(toKey)) {
125             list.remove(list.size()-1);
126         }
127         return list;
128     }
129
130
131     @Override
132     public Set<T> keySet() {
133         return RBTree.keySet();
134     }
135
136     @Override
137     public Map.Entry<T, V> lastEntry() {
138
139         return  RBTree.Maximum() ;
140     }
141
```

*RedBlackTree.java | TreeMap.java

```
136     @Override
137     public Map.Entry<T, V> lastEntry() {
138
139         return  RBTree.Maximum() ;
140     }
141
142     @Override
143     public T lastKey() {
144         Entry<T, V> e = RBTree.Maximum();
145         if(e == null) {
146             return null;
147         }
148             return (T) e.getKey();
149     }
150
151     @Override
152     public Entry<T, V> pollFirstEntry() {
153         Entry<T, V> e = RBTree.Minimum();
154         if(e == null) {
155             return null;
156             }
157         RBTree.delete((T) e.getKey());
158         size--;
159         return e;
160     }
161
162
163     @Override
164     public Entry<T, V> pollLastEntry() {
165         Entry<T, V> e = RBTree.Maximum();
166
167         if(e == null) {
168             return null;
169             }
170         RBTree.delete((T) e.getKey());
171         size--;
172         return e;
173     }
```

*RedBlackTree.java | TreeMap.java

```
175     @Override
176     public void put(T key, V value) {
177         RBTree.insert(key, value);
178         size++;
179         }
180
181     @Override
182     public void putAll(Map<T, V> map) {
183         if(map == null) {
184             throw new RuntimeErrorException(null);
185         }
186         Set<Entry<T, V>> set= map.entrySet();
187         for(Entry<T, V> e : set) {
188             RBTree.insert((T) e.getKey(), e.getValue());
189             size++;
190         }
191
192     }
193
194     @Override
195     public boolean remove(T key) {
196         if(key == null) {
197             throw new RuntimeErrorException(null);
198         }
199         if( RBTree.delete(key)) {
200         size--;
201         return true;
202         }
203         return false;
204         }
205
206     @Override
207     public int size() {
208
209         return RBTree.size;
210     }
211
```