

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 →

Node :

```
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;
597     }
598
599     @Override
600     public void setRightChild(INode<T,V> rightChild) {
601         // TODO Auto-generated method stub
602         this.rightChild = rightChild;
603     }
604
605     @Override
606     public INode<T,V> getRightChild() {
607         // TODO Auto-generated method stub
608         return rightChild;
609     }
610
611     // TODO Auto-generated method stub
612     return key;
613 }
614
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) {
```

Red Black Tree :


```

RedBlackTree.java TreeMap.java
26 public INode<T,V> getRoot() {
27     return root;
28 }
29
30
31
32 @Override
33 public boolean isEmpty() {
34     // TODO Auto-generated method stub
35     if (root.isNull()) {
36         return true;
37     }
38     return false;
39 }
40
41 @Override
42 public void clear() {
43     // TODO Auto-generated method stub
44     root = new Node<T, V>(null,null,null,null,false);
45 }
46 @Override
47 public V search(T key) {
48     // TODO Auto-generated method stub
49     if (key == null) {
50         throw new RuntimeException(null);
51     }
52     INode<T,V> node = root;
53     while (!node.isNull()) {
54         if (node.getKey().compareTo((T) key) == 0) {
55             break;
56         } else if (key.compareTo(node.getKey()) > 0) {
57             node = node.getRightChild();
58         } else {
59             node = node.getLeftChild();
60         }
61     }
62     return node.getValue();
63 }

*RedBlackTree.java TreeMap.java
64 public T ceilingKey(T key) {
65     node = root;
66     while (!node.isNull()) {
67         if (node.getKey().compareTo(key) == 0) {
68             return (T) node.getKey();
69         } else if (key.compareTo(node.getKey()) > 0) {
70             node = node.getRightChild();
71         } else {
72             if (node.getLeftChild().isNull()) {
73                 return (T) node.getKey();
74             } else {
75                 node = node.getLeftChild();
76             }
77         }
78     }
79     if( node == root && (key.compareTo(node.getKey()) < 0)) {
80         return (T) root.getKey();
81     }
82
83     return null;
84 }
85 public Map.Entry<T, V> ceilingEntry(T key) {
86     Map.Entry<T, V> e ;
87     node = getRoot();
88     while (!node.isNull()) {
89         if (node.getKey().equals(key)) {
90             e = new AbstractMap.SimpleEntry<T, V>((T)node.getKey(),(V) node.getValue());
91             return e;
92         } else if (key.compareTo(node.getKey()) > 0) {
93             node = node.getRightChild();
94         } else {
95             if (node.getLeftChild().isNull()) {
96                 e = new AbstractMap.SimpleEntry<T, V>((T)node.getKey(),(V) node.getValue());
97                 return e;
98             } else {
99                 node = node.getLeftChild();
100             }
101         }

```

```

*RedBlackTree.java  TreeMap.java
107 public Map.Entry<T, V> floorEntry(T key) {
108     Map.Entry<T, V> e;
109     INode<T,V> node = root;
110     while (!node.isNull()) {
111         if (node.getKey().equals(key)) {
112             e= new AbstractMap.SimpleEntry<T, V>((T)node.getKey(),(V) node.getValue());
113             return e;
114         } else if (key.compareTo(node.getKey()) > 0) {
115             if (node.getRightChild().isNull()) {
116                 e= new AbstractMap.SimpleEntry<T, V>((T)node.getKey(),(V) node.getValue());
117                 return e;
118             }
119             node = node.getRightChild();
120         } else {
121             node = node.getLeftChild();
122         }
123     }
124     return null;
125 }
126
127
128 @Override
129 public boolean contains(T key) {
130     // TODO Auto-generated method stub
131     if(key == null) {
132         throw new RuntimeException(null);
133     }
134     INode<T,V> node = root;
135     while(!node.isNull()) {
136         if(node.getKey().compareTo(key) == 0) {
137             return true;
138         } else if (node.getKey().compareTo(key) < 0) {
139             node = node.getRightChild();
140         } else {
141             node = node.getLeftChild();
142         }
143     }
144     return false;
145 }
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

*RedBlackTree.java TreeMap.java

```
310 private void checkColor(INode<T,V> z) {
311     INode<T,V> uncle = new Node<T,V>(null,null,null,null,null,false);
312     while(z.getParent().getColor()) {
313         if(z.getParent().equals(z.getParent().getParent().getLeftChild())) {
314             uncle = z.getParent().getParent().getRightChild();
315             if(uncle.getColor() && !uncle.isNull()) {
316                 uncle.setColor(false);
317                 z.getParent().setColor(false);
318                 z.getParent().getParent().setColor(true);
319                 z = z.getParent().getParent();
320             } else if(z.equals(z.getParent().getRightChild())) {
321                 z = z.getParent();
322                 leftRotate(z);
323             } else {
324                 z.getParent().setColor(false);
325                 z.getParent().getParent().setColor(true);
326                 rightRotate(z.getParent().getParent());
327             }
328         } else {
329             uncle = z.getParent().getParent().getLeftChild();
330             if(uncle.getColor() && !uncle.isNull()) {
331                 uncle.setColor(false);
332                 z.getParent().setColor(false);
333                 z.getParent().getParent().setColor(true);
334                 z = z.getParent().getParent();
335             } else if(z.equals(z.getParent().getLeftChild())) {
336                 z = z.getParent();
337                 rightRotate(z);
338             } else {
339                 z.getParent().setColor(false);
340                 z.getParent().getParent().setColor(true);
341                 leftRotate(z.getParent().getParent());
342             }
343         }
344     }
345     root.setColor(false);
346 }
```

*RedBlackTree.java TreeMap.java

```
347
348 @Override
349 public boolean delete(T key) {
350     // TODO Auto-generated method stub
351
352     if (key == null) {
353         throw new RuntimeException(null);
354     }
355     if (!contains(key)) {
356         return false;
357     }
358     size--;
359     INode<T,V> node = root;
360     while (!node.isNull()) {
361         if (node.getKey().compareTo((T) key) == 0) {
362             deleteNode(node);
363             break;
364         } else if (key.compareTo(node.getKey()) > 0) {
365             node = node.getRightChild();
366         } else {
367             node = node.getLeftChild();
368         }
369     }
370
371     return true;
372 }
373 private void transPlant(INode<T,V> u, INode<T,V> v) {
374     if(u.getParent().isNull()) {
375         root = v;
376     } else if(u.getParent().getLeftChild().equals(u)) {
377         u.getParent().setLeftChild(v);
378     } else {
379         u.getParent().setRightChild(v);
380     }
381     v.setParent(u.getParent());
382 }
383 }
```

```

    }

    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         } else {
443             w = x.getParent().getLeftChild();
444             if(w.getColor()) {
445                 w.setColor(false);
446                 x.getParent().setColor(true);
447                 rightRotate(x.getParent());
448                 w = x.getParent().getLeftChild();
449             }
450             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()) {
            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 }
}
}
1 private INode<T, V> Minimum(INode<T, V> node) {
5     while(!node.getLeftChild().isNull()) {
5         node = node.getLeftChild();
7     }
3     return node;
) }
)

```

Tree Map :

```
*RedBlackTree.java  TreeMap.java
21 @Override
22 public Map.Entry<T, V> ceilingEntry(T key) {
23     if(key == null) {
24         throw new RuntimeException(null);
25     }
26     return RBTREE.ceilingEntry(key);
27 }
28
29 @Override
30 public T ceilingKey(T key) {
31     if(key == null) {
32         throw new RuntimeException(null);
33     }
34     return RBTREE.ceilingKey(key);
35 }
36
37 @Override
38 public void clear() {
39     RBTREE = new RedBlackTree<>();
40 }
41
42
43 @Override
44 public boolean containsKey(T key) {
45     if(key == null) {
46         throw new RuntimeException(null);
47     }
48     return RBTREE.contains(key);
49 }
50
51 @Override
52 public boolean containsValue(Object value) {
53     if(value == null) {
54         throw new RuntimeException(null);
55     }
56     return RBTREE.containsValue(value);
57 }
58
59 @Override
60 public Set<Map.Entry<T, V>> entrySet() {
61     return RBTREE.setMap();
62 }
63
64 @Override
65 public Map.Entry<T, V> firstEntry() {
66     return RBTREE.Minimum();
67 }
68
69 @Override
70 public T firstKey() {
71     Entry<T, V> e = RBTREE.Minimum();
72     if(e == null) {
73         return null;
74     }
75     return (T) e.getKey();
76 }
77
78 @Override
79 public Map.Entry<T, V> floorEntry(T key) {
80     if(key == null) {
81         throw new RuntimeException(null);
82     }
83     return RBTREE.floorEntry(key);
84 }
85
86 @Override
87 public T floorKey(T key) {
88     if(key == null) {
89         throw new RuntimeException(null);
90     }
91     Entry<T, V> e = RBTREE.floorEntry(key);
92     return (T) e.getKey();
93 }
94
95
```

```

*RedBlackTree.java  TreeMap.java
104 @Override
105 public ArrayList<Map.Entry<T, V>> headMap(T toKey) {
106     if(toKey == null) {
107         throw new RuntimeException(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 RuntimeException(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     return RBTree.Maximum() ;
139 }
140
141

```

```

*RedBlackTree.java  TreeMap.java
136 @Override
137 public Map.Entry<T, V> lastEntry() {
138     return RBTree.Maximum() ;
139 }
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 RuntimeException(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 RuntimeException(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
212 @Override

```