Data Structures

Lab 2 – Red Black Tree

Prepared by: Eman Rafik

➢ Problem Statement:

- Implementing Red Black Tree with the following methods:

1. **getRoot:** return the root of the given Red black tree.
2. **isEmpty:** return whether the given tree is empty or not.
3. **clear:** Clear all keys in the given tree.
4. **search:** return the value associated with the given key or null if no value is found.
5. **contains:** return true if the tree contains the given key and false otherwise.
6. **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.
7. **delete:** Delete the node associated with the given key. Return true in case of success and false otherwise.

- Implementing an interface similar to Treemap java interface with the following functions:

1. **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.
2. **ceilingKey:** Returns the least key greater than or equal to the given key, or null if there is no such key.
3. **clear:** Removes all of the mappings from this map.
4. **containsKey:** Returns true if this map contains a mapping for the specified key.
5. **containsValue:** Returns true if this map maps one or more keys to the specified value.
6. **entrySet:** Returns a Set view of the mappings contained in this map in ascending key order.
7. **firstEntry:** Returns a key-value mapping associated with the least key in this map, or null if the map is empty.
8. **firstKey:** Returns the first (lowest) key currently in this map, or null if the map is empty.
9. **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.
10. **floorKey:** Returns the greatest key less than or equal to the given key, or null if there is no such key.
11. **get:** Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
12. **headMap:** Returns a view of the portion of this map whose keys are strictly less than toKey in ascending order.
13. **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.
14. **keySet:** Returns a Set view of the keys contained in this map.
15. **lastEntry:** Returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
16. **lastKey:** Returns the last (highest) key currently in this map.
17. **pollFirstElement:** Removes and returns a key-value mapping associated with the least key in this map, or null if the map is empty.
18. **pollLastEntry:** Removes and returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
19. **put:** Associates the specified value with the specified key in this map.

20. **putAll:** Copies all of the mappings from the specified map to this map.
21. **remove:** Removes the mapping for this key from this TreeMap if present.
22. **size:** Returns the number of key-value mappings in this map.
23. **values:** Returns a Collection view of the values contained in this map.

➢ Algorithms:

• Insert:

insert(key, value)

```
t = root
p = nullNode
while (!t.isNull()) {
        p = t
        if (key =  t.getKey)
                t. value=value
        else if (key < t.getKey)
                t = t.Left
        else
                t = t.Right
}
z.parent=p
z.key=key
z.value=value
if (p == nullNode)
        root = z;
else if (key <  p.key)
        p.Left=z
else
        p.Right=z
z.right=nullNode
z.left=nullNode
z.color=RED
insertFix(z);
```

```
insertFix(z)
        while (z.parent.color=RED) {
            if (z.parent == z.parent.parent.Left)
                y = z.parent.parent.Right
                if (y.color = RED)
                    y.color=BLACK
                    z.parent.color=BLACK
                    z.parent.parent.color=RED
                    z = z.getParent().getParent();
                else
                    if (z == z.parent.Right)
                        z = z.parent
                        rotateLeft(z);
                    z.parent.color=BLACK
                    z.parent.parent.color=RED
                    rotateRight(z.parent.parent)
            else if ((z.parent == z.parent.parent.right) {
                y = z.parent.parent.LeftChild
                if (y.color=RED)
                    y.color=BLACK
                    z.parent.color=BLACK
                    z.parent.parent.color=RED
                    z = z.parent.parent
                else
                    if (z == z.parent.Left)
                        z = z.parent
                        rotateRight(z)
                    z.parent.color=BLACK
                    z.parent.parent.color=RED
                    rotateLeft(z.parent.parent)
        root.color=BLACK
```

- **Delete:**

```
delete(T key)
        t = root
        while (key != t.Key) {
                if (key >  t.Key)
                        t = t.Right
                }else
                        t = t.Left
        }
        color = t.color
        if (t. Left.isNull()) {
                x = t. Right
                transplant(t.Right, t)
        else if (t.Right.isNull())
                x = t.LeftChild
                transplant(t.Left, t);
        else
                min = minimum(t.Right);
                x = min.RightChild
                color = min.color
                if (min.parent != t)
                        transplant(x, min)
                        min.Right=t.Right
                        min.Right.parent=min
                else
                        x.parent=min
                transplant(min, t);
                min. Left=t.LeftChild
                t.Left.parent=min
                min.color=t.color
                t = min;
        if (color=BLACK) {
                deleteFix(x);
        }
```

```
deleteFix(x)
        while (x != root && x.color=BLACK) {
            if (x == x.parent.Left)
                z = x.parent.RightChild
                if (z.color==RED)
                    z.color=BLACK
                    x.parent.color=RED
                    rotateLeft(x.parent)
                    z = x.parent.Right
                if (z.Left.color=BLACK && z.Right.color==BLACK)
                    z.color=RED
                    x = x.parent
                else
                    if (z. Left.color=RED && z.Right.olor=BLACK)
                        z.Left.color=BLACK
                        z.color=RED
                        rotateRight(z)
                        z = x.parent.Right
                    z.color=x.parent.color
                    x.parent.color=BLACK
                    z.Right.color=BLACK
                    rotateLeft(x.parent)
                    x = root
            else
                z = x.parent.RightChild
                if (z.color==RED) {
                    z.color=BLACK
                    x.parent.color=RED
                    rotateRight(x.parent)
                    z = x.parent.Left
                if (z.Left.color=BLACK && z.Right.color==BLACK)
                    z.color=RED
                        x = x.parent
                else
                    if (z. Right.color=RED && z.Left.olor=BLACK)
                        z.Right.color=BLACK
                        z.color=RED
                        rotateLeft(z)
                        z = x.parent.Left
                    z.color=x.parent.color
                    x.parent.color=BLACK
                    z.Left.color=BLACK
                    rotateRight(x.parent)
                    x = root

        }
        x.color=.BLACK
```