<2T>

# Red-Black Tree Description

A **red-black tree** is a balanced binary search tree. All basic operations take O(log n) time in the worst case.

Properties:

1. Every node is either red or black
2. The root is black
3. Every leaf is black
4. If a node is red, then both its children are black
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Rotations must be done when inserting or deleting to preserve the properties of the tree.

Insertion: we insert a node into the tree like a binary search tree and color it red initially. Then we call an auxiliary procedure to recolor and rotate the nodes in the tree.

Deleting a node from a red-black tree is more complicated, and we need to define a transplant procedure that swaps two nodes. Deleting a node is structurally similar to deleting a node from a binary search tree. We need a dedicated fixup function to handle updating the coloring of the red-black tree.

# Red-Black Tree Pseudocode

LEFT-ROTATE(T, x)

1 y = x.right // set y

2 x.right = y.left // turn y’s left subtree into x’s right subtree

3 if y.left ≠ T.nil

4 y.left.p = x

5 y.p = x.p // link x’s parent to y

6 if x.p = T.nil

7 T.root = y

8 elseif x == x.p.left

9 x.p.left = y

10 else x.p.right = y

11 y.left = x // put x on y’s left

12 x.p = y

RB-INSERT(T, z)

1 y = T.nil

2 x = T.root

3 while x ≠ T.nil

4 y = x

5 if z.key < x.key

6 x = x.left

7 else x = x.right

8 z.p = y

9 if y == T.nil

10 T.root = z

11 elseif z.key < y.key

12 y.left = z

13 else y.right = z

14 z.left = T.nil

15 z.right = T.nil

16 z.color = RED

17 RB-INSERT-FIXUP(T, z)

RB-INSERT-FIXUP(T, z)

1 while z.p.color = RED

2 if z.p == z.p.p.left

3 y = z.p.p.right

4 if y.color == RED

5 z.p.color = BLACK

6 y.color = BLACK

7 z.p.p.color = RED

8 z = z.p.p

9 elseif z == z.p.right

10 z = z.p

11 LEFT-ROTATE(T, z)

12 z.p.color = BLACK

13 z.p.p.color = RED

14 RIGHT-ROTATE(T, z.p.p)

15 else (same as then clause with “right” and “left” exchanged)

16 T.root.color = BLACK

RB-TRANSPLANT(T, u, v)

1 if u.p == T.nil

2 T.root = v

3 elseif u == u.p.left

4 u.p.left = v

5 else u.p.right = v

6 v.p = u.p

RB-DELETE(T, z)

1 if u.p == T.nil

2 T.root = v

3 elseif u == u.p.left

4 u.p.left = v

5 else u.p.right = v

6 v.p = u.p

RB-DELETE (T, z)

1 y = z

2 y-original-color = y.color

3 if z.left == T.nil

4 x = z.right

5 RB-TRANSPLANT (T, z, z.right)

6 elseif z.right == T.nil

7 x = z.left

8 RB-TRANSPLANT(T, z, z.left)

9 else y == TREE-MINIMUM(z.right)

10 y-original-color = y.color

11 x = y/right

12 if y.p == z

13 x.p = y

14 else RB-TRANSPLANT(T, y, y.right)

15 y.right = z.right

16 y.right.p = y

17 RB-TRANSPLANT(T, z, y)

18 y.left = z.left

19 y.left.p = y

20 y.color = z.color

21 if y-original-color == BLACK

22 RB-DELETE-FIXUP(T, x)

RB-DELETE-FIXUP(T, x)

1 while x ≠ T.root and x.color == BLACK

2 if x == x.p. left

3 w = x.p.right

4 if w.color == RED

5 w.color = BLACK // case 1

6 x.p.color = RED // case 1

7 LEFT-ROTATE (T, x.p) // case 1

8 w = x.p.right // case 1

9 if w.left.color == BLACK and w.right.color == BLACK

10 w.color = RED // case 2

11 x = x.p // case 2

12 else if w.right.color == BLACK

13 w.left.color = BLACK // case 3

14 w.color = RED // case 3

15 RIGHT-ROTATE (T, w) // case 3

16 w = x.p.right // case 3

17 w.color = x.p.color // case 4

18 x.p.color = BLACK // case 4

19 w.right.color = BLACK // case 4

20 LEFT-ROTATE (T, x.p) // case 4

21 x = T.root // case 4

22 else (same as then clause with “right” and “left” exchanged)

23 x.color = BLACK

# Red-Black Tree Code

""""

Red-Black Tree implementation

Credit: http://code.activestate.com/recipes/576817-red-black-tree/

"""

class rbnode(object):

    """

    A node in a red black tree. See Cormen, Leiserson, Rivest, Stein 2nd edition pg 273.

    """

    def \_\_init\_\_(self, key):

        "Construct."

        self.\_key = key

        self.\_red = False

        self.\_left = None

        self.\_right = None

        self.\_p = None

    key = property(fget=lambda self: self.\_key, doc="The node's key")

    red = property(fget=lambda self: self.\_red, doc="Is the node red?")

    left = property(fget=lambda self: self.\_left, doc="The node's left child")

    right = property(fget=lambda self: self.\_right, doc="The node's right child")

    p = property(fget=lambda self: self.\_p, doc="The node's parent")

    def \_\_str\_\_(self):

        "String representation."

        return str(self.key)

    def \_\_repr\_\_(self):

        "String representation."

        return str(self.key)

class rbtree(object):

    """

    A red black tree. See Cormen, Leiserson, Rivest, Stein 2nd edition pg 273.

    """

    def \_\_init\_\_(self, create\_node=rbnode):

        "Construct."

        self.\_nil = create\_node(key=None)

        "Our nil node, used for all leaves."

        self.\_root = self.nil

        "The root of the tree."

        self.\_create\_node = create\_node

        "A callable that creates a node."

    root = property(fget=lambda self: self.\_root, doc="The tree's root node")

    nil = property(fget=lambda self: self.\_nil, doc="The tree's nil node")

    def search(self, key, x=None):

        """

        Search the subtree rooted at x (or the root if not given) iteratively for the key.

        @return: self.nil if it cannot find it.

        """

        if None == x:

            x = self.root

        while x != self.nil and key != x.key:

            if key < x.key:

                x = x.left

            else:

                x = x.right

        return x

    def minimum(self, x=None):

        """

        @return: The minimum value in the subtree rooted at x.

        """

        if None == x:

            x = self.root

        while x.left != self.nil:

            x = x.left

        return x

    def maximum(self, x=None):

        """

        @return: The maximum value in the subtree rooted at x.

        """

        if None == x:

            x = self.root

        while x.right != self.nil:

            x = x.right

        return x

    def insert\_key(self, key):

        "Insert the key into the tree."

        self.insert\_node(self.\_create\_node(key=key))

    def insert\_node(self, z):

        "Insert node z into the tree."

        y = self.nil

        x = self.root

        while x != self.nil:

            y = x

            if z.key < x.key:

                x = x.left

            else:

                x = x.right

        z.\_p = y

        if y == self.nil:

            self.\_root = z

        elif z.key < y.key:

            y.\_left = z

        else:

            y.\_right = z

        z.\_left = self.nil

        z.\_right = self.nil

        z.\_red = True

        self.\_insert\_fixup(z)

    def \_insert\_fixup(self, z):

        "Restore red-black properties after insert."

        while z.p.red:

            if z.p == z.p.p.left:

                y = z.p.p.right

                if y.red:

                    z.p.\_red = False

                    y.\_red = False

                    z.p.p.\_red = True

                    z = z.p.p

                else:

                    if z == z.p.right:

                        z = z.p

                        self.\_left\_rotate(z)

                    z.p.\_red = False

                    z.p.p.\_red = True

                    self.\_right\_rotate(z.p.p)

            else:

                y = z.p.p.left

                if y.red:

                    z.p.\_red = False

                    y.\_red = False

                    z.p.p.\_red = True

                    z = z.p.p

                else:

                    if z == z.p.left:

                        z = z.p

                        self.\_right\_rotate(z)

                    z.p.\_red = False

                    z.p.p.\_red = True

                    self.\_left\_rotate(z.p.p)

        self.root.\_red = False

    def \_left\_rotate(self, x):

        "Left rotate x."

        y = x.right

        x.\_right = y.left

        if y.left != self.nil:

            y.left.\_p = x

        y.\_p = x.p

        if x.p == self.nil:

            self.\_root = y

        elif x == x.p.left:

            x.p.\_left = y

        else:

            x.p.\_right = y

        y.\_left = x

        x.\_p = y

    def \_right\_rotate(self, y):

        "Left rotate y."

        x = y.left

        y.\_left = x.right

        if x.right != self.nil:

            x.right.\_p = y

        x.\_p = y.p

        if y.p == self.nil:

            self.\_root = x

        elif y == y.p.right:

            y.p.\_right = x

        else:

            y.p.\_left = x

        x.\_right = y

        y.\_p = x

    def check\_invariants(self):

        "@return: True iff satisfies all criteria to be red-black tree."

        def is\_red\_black\_node(node):

            "@return: num\_black"

            # check has \_left and \_right or neither

            if (node.left and not node.right) or (node.right and not node.left):

                return 0, False

            # check leaves are black

            if not node.left and not node.right and node.red:

                return 0, False

            # if node is red, check children are black

            if node.red and node.left and node.right:

                if node.left.red or node.right.red:

                    return 0, False

            # descend tree and check black counts are balanced

            if node.left and node.right:

                # check children's parents are correct

                if self.nil != node.left and node != node.left.p:

                    return 0, False

                if self.nil != node.right and node != node.right.p:

                    return 0, False

                # check children are ok

                left\_counts, left\_ok = is\_red\_black\_node(node.left)

                if not left\_ok:

                    return 0, False

                right\_counts, right\_ok = is\_red\_black\_node(node.right)

                if not right\_ok:

                    return 0, False

                # check children's counts are ok

                if left\_counts != right\_counts:

                    return 0, False

                return left\_counts, True

            else:

                return 0, True

        num\_black, is\_ok = is\_red\_black\_node(self.root)

        return is\_ok and not self.root.\_red

def write\_tree\_as\_dot(t, f, show\_nil=False):

    "Write the tree in the dot language format to f."

    def node\_id(node):

        return 'N%d' % id(node)

    def node\_color(node):

        if node.red:

            return "red"

        else:

            return "black"

    def visit\_node(node):

        "Visit a node."

        #print(node\_id(node)+"   [label=\""+node\_id(node)+"\", color=\""+node\_color(node)+"\"];", f)

        f.write(node\_id(node)+":[label=\""+str(node)+"\", color=\""+node\_color(node)+"\"];\n")

        if node.left:

            if node.left != t.nil or show\_nil:

                visit\_node(node.left)

                #print(node\_id(node) + "   -> "+ node\_id(node.left) +" ;", f)

                f.write(str(node) + "-> (L)"+ str(node.left) +" ;\n")

        if node.right:

            if node.right != t.nil or show\_nil:

                visit\_node(node.right)

                #print(node\_id(node) + "   -> "+ node\_id(node.right) +" ;", f)

                f.write(str(node) + "-> (R)"+ str(node.right) +" ;\n")

    #print("// Created by rbtree.write\_dot()", f)

    f.write("// Created by rbtree.write\_dot()\n")

    #print("digraph red\_black\_tree {", f)

    f.write("digraph red\_black\_tree {\n")

    visit\_node(t.root)

    #print("}", f)

    f.write("}")

def test\_tree(t, keys):

    "Insert keys one by one checking invariants and membership as we go."

    assert t.check\_invariants()

    for i, key in enumerate(keys):

        for key2 in keys[:i]:

            assert t.nil != t.search(key2)

        for key2 in keys[i:]:

            assert (t.nil == t.search(key2)) ^ (key2 in keys[:i])

        t.insert\_key(key)

        assert t.check\_invariants()

if '\_\_main\_\_' == \_\_name\_\_:

    import os, sys, numpy.random as R

    def write\_tree(t, filename):

        "Write the tree as an SVG file."

        f = open('%s.dot' % filename, 'w')

        write\_tree\_as\_dot(t, f)

        f.close()

        os.system('dot %s.dot -Tsvg -o %s.svg' % (filename, filename))

    # test the rbtree

    R.seed(2)

    size=10

    keys = R.randint(-50, 50, size=size)

    t = rbtree()

    test\_tree(t, keys)

    write\_tree(t, 'tree')