<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

"""

A classic (not left-leaning) Red-Black Tree implementation, supporting addition and deletion.

"""

# The possible Node colors

BLACK = 'BLACK'

RED = 'RED'

NIL = 'NIL'

class Node:

def \_\_init\_\_(self, value, color, parent, left=None, right=None):

self.value = value

self.color = color

self.parent = parent

self.left = left

self.right = right

def \_\_repr\_\_(self):

return '{color} {val} Node'.format(color=self.color, val=self.value)

def \_\_iter\_\_(self):

if self.left.color != NIL:

yield from self.left.\_\_iter\_\_()

yield self.value

if self.right.color != NIL:

yield from self.right.\_\_iter\_\_()

def \_\_eq\_\_(self, other):

if self.color == NIL and self.color == other.color:

return True

if self.parent is None or other.parent is None:

parents\_are\_same = self.parent is None and other.parent is None

else:

parents\_are\_same = self.parent.value == other.parent.value and self.parent.color == other.parent.color

return self.value == other.value and self.color == other.color and parents\_are\_same

def has\_children(self) -> bool:

""" Returns a boolean indicating if the node has children """

return bool(self.get\_children\_count())

def get\_children\_count(self) -> int:

""" Returns the number of NOT NIL children the node has """

if self.color == NIL:

return 0

return sum([int(self.left.color != NIL), int(self.right.color != NIL)])

class RedBlackTree:

# every node has null nodes as children initially, create one such object for easy management

NIL\_LEAF = Node(value=None, color=NIL, parent=None)

def \_\_init\_\_(self):

self.count = 0

self.root = None

self.ROTATIONS = {

# Used for deletion and uses the sibling's relationship with his parent as a guide to the rotation

'L': self.\_right\_rotation,

'R': self.\_left\_rotation

}

def \_\_iter\_\_(self):

if not self.root:

return list()

yield from self.root.\_\_iter\_\_()

def add(self, value):

if not self.root:

self.root = Node(value, color=BLACK, parent=None, left=self.NIL\_LEAF, right=self.NIL\_LEAF)

self.count += 1

return

parent, node\_dir = self.\_find\_parent(value)

if node\_dir is None:

return # value is in the tree

new\_node = Node(value=value, color=RED, parent=parent, left=self.NIL\_LEAF, right=self.NIL\_LEAF)

if node\_dir == 'L':

parent.left = new\_node

else:

parent.right = new\_node

self.\_try\_rebalance(new\_node)

self.count += 1

def remove(self, value):

"""

Try to get a node with 0 or 1 children.

Either the node we're given has 0 or 1 children or we get its successor.

"""

node\_to\_remove = self.find\_node(value)

if node\_to\_remove is None: # node is not in the tree

return

if node\_to\_remove.get\_children\_count() == 2:

# find the in-order successor and replace its value.

# then, remove the successor

successor = self.\_find\_in\_order\_successor(node\_to\_remove)

node\_to\_remove.value = successor.value # switch the value

node\_to\_remove = successor

# has 0 or 1 children!

self.\_remove(node\_to\_remove)

self.count -= 1

def contains(self, value) -> bool:

""" Returns a boolean indicating if the given value is present in the tree """

return bool(self.find\_node(value))

def ceil(self, value) -> int or None:

"""

Given a value, return the closest value that is equal or bigger than it,

returning None when no such exists

"""

if self.root is None: return None

last\_found\_val = None if self.root.value < value else self.root.value

def find\_ceil(node):

nonlocal last\_found\_val

if node == self.NIL\_LEAF:

return None

if node.value == value:

last\_found\_val = node.value

return node.value

elif node.value < value:

# go right

return find\_ceil(node.right)

else:

# this node is bigger, save its value and go left

last\_found\_val = node.value

return find\_ceil(node.left)

find\_ceil(self.root)

return last\_found\_val

def floor(self, value) -> int or None:

"""

Given a value, return the closest value that is equal or less than it,

returning None when no such exists

"""

if self.root is None: return None

last\_found\_val = None if self.root.value < value else self.root.value

def find\_floor(node):

nonlocal last\_found\_val

if node == self.NIL\_LEAF:

return None

if node.value == value:

last\_found\_val = node.value

return node.value

elif node.value < value:

# this node is smaller, save its value and go right, trying to find a cloer one

last\_found\_val = node.value

return find\_floor(node.right)

else:

return find\_floor(node.left)

find\_floor(self.root)

return last\_found\_val

def \_remove(self, node):

"""

Receives a node with 0 or 1 children (typically some sort of successor)

and removes it according to its color/children

:param node: Node with 0 or 1 children

"""

left\_child = node.left

right\_child = node.right

not\_nil\_child = left\_child if left\_child != self.NIL\_LEAF else right\_child

if node == self.root:

if not\_nil\_child != self.NIL\_LEAF:

# if we're removing the root and it has one valid child, simply make that child the root

self.root = not\_nil\_child

self.root.parent = None

self.root.color = BLACK

else:

self.root = None

elif node.color == RED:

if not node.has\_children():

# Red node with no children, the simplest remove

self.\_remove\_leaf(node)

else:

"""

Since the node is red he cannot have a child.

If he had a child, it'd need to be black, but that would mean that

the black height would be bigger on the one side and that would make our tree invalid

"""

raise Exception('Unexpected behavior')

else: # node is black!

if right\_child.has\_children() or left\_child.has\_children(): # sanity check

raise Exception('The red child of a black node with 0 or 1 children'

' cannot have children, otherwise the black height of the tree becomes invalid! ')

if not\_nil\_child.color == RED:

"""

Swap the values with the red child and remove it (basically un-link it)

Since we're a node with one child only, we can be sure that there are no nodes below the red child.

"""

node.value = not\_nil\_child.value

node.left = not\_nil\_child.left

node.right = not\_nil\_child.right

else: # BLACK child

# 6 cases :o

self.\_remove\_black\_node(node)

def \_remove\_leaf(self, leaf):

""" Simply removes a leaf node by making it's parent point to a NIL LEAF"""

if leaf.value >= leaf.parent.value:

# in those weird cases where they're equal due to the successor swap

leaf.parent.right = self.NIL\_LEAF

else:

leaf.parent.left = self.NIL\_LEAF

def \_remove\_black\_node(self, node):

"""

Loop through each case recursively until we reach a terminating case.

What we're left with is a leaf node which is ready to be deleted without consequences

"""

self.\_\_case\_1(node)

self.\_remove\_leaf(node)

def \_\_case\_1(self, node):

"""

Case 1 is when there's a double black node on the root

Because we're at the root, we can simply remove it

and reduce the black height of the whole tree.

\_\_|10B|\_\_ \_\_10B\_\_

/ \ ==> / \

9B 20B 9B 20B

"""

if self.root == node:

node.color = BLACK

return

self.\_\_case\_2(node)

def \_\_case\_2(self, node):

"""

Case 2 applies when

the parent is BLACK

the sibling is RED

the sibling's children are BLACK or NIL

It takes the sibling and rotates it

40B 60B

/ \ --CASE 2 ROTATE--> / \

|20B| 60R LEFT ROTATE 40R 80B

DBL BLACK IS 20----^ / \ SIBLING 60R / \

50B 80B |20B| 50B

(if the sibling's direction was left of it's parent, we would RIGHT ROTATE it)

Now the original node's parent is RED

and we can apply case 4 or case 6

"""

parent = node.parent

sibling, direction = self.\_get\_sibling(node)

if sibling.color == RED and parent.color == BLACK and sibling.left.color != RED and sibling.right.color != RED:

self.ROTATIONS[direction](node=None, parent=sibling, grandfather=parent)

parent.color = RED

sibling.color = BLACK

return self.\_\_case\_1(node)

self.\_\_case\_3(node)

def \_\_case\_3(self, node):

"""

Case 3 deletion is when:

the parent is BLACK

the sibling is BLACK

the sibling's children are BLACK

Then, we make the sibling red and

pass the double black node upwards

Parent is black

\_\_\_50B\_\_\_ Sibling is black \_\_\_50B\_\_\_

/ \ Sibling's children are black / \

30B 80B CASE 3 30B |80B| Continue with other cases

/ \ / \ ==> / \ / \

20B 35R 70B |90B|<---REMOVE 20B 35R 70R X

/ \ / \

34B 37B 34B 37B

"""

parent = node.parent

sibling, \_ = self.\_get\_sibling(node)

if (sibling.color == BLACK and parent.color == BLACK

and sibling.left.color != RED and sibling.right.color != RED):

# color the sibling red and forward the double black node upwards

# (call the cases again for the parent)

sibling.color = RED

return self.\_\_case\_1(parent) # start again

self.\_\_case\_4(node)

def \_\_case\_4(self, node):

"""

If the parent is red and the sibling is black with no red children,

simply swap their colors

DB-Double Black

\_\_10R\_\_ \_\_10B\_\_ The black height of the left subtree has been incremented

/ \ / \ And the one below stays the same

DB 15B ===> X 15R No consequences, we're done!

/ \ / \

12B 17B 12B 17B

"""

parent = node.parent

if parent.color == RED:

sibling, direction = self.\_get\_sibling(node)

if sibling.color == BLACK and sibling.left.color != RED and sibling.right.color != RED:

parent.color, sibling.color = sibling.color, parent.color # switch colors

return # Terminating

self.\_\_case\_5(node)

def \_\_case\_5(self, node):

"""

Case 5 is a rotation that changes the circumstances so that we can do a case 6

If the closer node is red and the outer BLACK or NIL, we do a left/right rotation, depending on the orientation

This will showcase when the CLOSER NODE's direction is RIGHT

\_\_\_50B\_\_\_ \_\_50B\_\_

/ \ / \

30B |80B| <-- Double black 35B |80B| Case 6 is now

/ \ / \ Closer node is red (35R) / \ / applicable here,

20B 35R 70R X Outer is black (20B) 30R 37B 70R so we redirect the node

/ \ So we do a LEFT ROTATION / \ to it :)

34B 37B on 35R (closer node) 20B 34B

"""

sibling, direction = self.\_get\_sibling(node)

closer\_node = sibling.right if direction == 'L' else sibling.left

outer\_node = sibling.left if direction == 'L' else sibling.right

if closer\_node.color == RED and outer\_node.color != RED and sibling.color == BLACK:

if direction == 'L':

self.\_left\_rotation(node=None, parent=closer\_node, grandfather=sibling)

else:

self.\_right\_rotation(node=None, parent=closer\_node, grandfather=sibling)

closer\_node.color = BLACK

sibling.color = RED

self.\_\_case\_6(node)

def \_\_case\_6(self, node):

"""

Case 6 requires

SIBLING to be BLACK

OUTER NODE to be RED

Then, does a right/left rotation on the sibling

This will showcase when the SIBLING's direction is LEFT

Double Black

\_\_50B\_\_ | \_\_35B\_\_

/ \ | / \

SIBLING--> 35B |80B| <- 30R 50R

/ \ / / \ / \

30R 37B 70R Outer node is RED 20B 34B 37B 80B

/ \ Closer node doesn't /

20B 34B matter 70R

Parent doesn't

matter

So we do a right rotation on 35B!

"""

sibling, direction = self.\_get\_sibling(node)

outer\_node = sibling.left if direction == 'L' else sibling.right

def \_\_case\_6\_rotation(direction):

parent\_color = sibling.parent.color

self.ROTATIONS[direction](node=None, parent=sibling, grandfather=sibling.parent)

# new parent is sibling

sibling.color = parent\_color

sibling.right.color = BLACK

sibling.left.color = BLACK

if sibling.color == BLACK and outer\_node.color == RED:

return \_\_case\_6\_rotation(direction) # terminating

raise Exception('We should have ended here, something is wrong')

def \_try\_rebalance(self, node):

"""

Given a red child node, determine if there is a need to rebalance (if the parent is red)

If there is, rebalance it

"""

parent = node.parent

value = node.value

if (parent is None # what the fuck? (should not happen)

or parent.parent is None # parent is the root

or (node.color != RED or parent.color != RED)): # no need to rebalance

return

grandfather = parent.parent

node\_dir = 'L' if parent.value > value else 'R'

parent\_dir = 'L' if grandfather.value > parent.value else 'R'

uncle = grandfather.right if parent\_dir == 'L' else grandfather.left

general\_direction = node\_dir + parent\_dir

if uncle == self.NIL\_LEAF or uncle.color == BLACK:

# rotate

if general\_direction == 'LL':

self.\_right\_rotation(node, parent, grandfather, to\_recolor=True)

elif general\_direction == 'RR':

self.\_left\_rotation(node, parent, grandfather, to\_recolor=True)

elif general\_direction == 'LR':

self.\_right\_rotation(node=None, parent=node, grandfather=parent)

# due to the prev rotation, our node is now the parent

self.\_left\_rotation(node=parent, parent=node, grandfather=grandfather, to\_recolor=True)

elif general\_direction == 'RL':

self.\_left\_rotation(node=None, parent=node, grandfather=parent)

# due to the prev rotation, our node is now the parent

self.\_right\_rotation(node=parent, parent=node, grandfather=grandfather, to\_recolor=True)

else:

raise Exception("{} is not a valid direction!".format(general\_direction))

else: # uncle is RED

self.\_recolor(grandfather)

def \_\_update\_parent(self, node, parent\_old\_child, new\_parent):

"""

Our node 'switches' places with the old child

Assigns a new parent to the node.

If the new\_parent is None, this means that our node becomes the root of the tree

"""

node.parent = new\_parent

if new\_parent:

# Determine the old child's position in order to put node there

if new\_parent.value > parent\_old\_child.value:

new\_parent.left = node

else:

new\_parent.right = node

else:

self.root = node

def \_right\_rotation(self, node, parent, grandfather, to\_recolor=False):

grand\_grandfather = grandfather.parent

self.\_\_update\_parent(node=parent, parent\_old\_child=grandfather, new\_parent=grand\_grandfather)

old\_right = parent.right

parent.right = grandfather

grandfather.parent = parent

grandfather.left = old\_right # save the old right values

old\_right.parent = grandfather

if to\_recolor:

parent.color = BLACK

node.color = RED

grandfather.color = RED

def \_left\_rotation(self, node, parent, grandfather, to\_recolor=False):

grand\_grandfather = grandfather.parent

self.\_\_update\_parent(node=parent, parent\_old\_child=grandfather, new\_parent=grand\_grandfather)

old\_left = parent.left

parent.left = grandfather

grandfather.parent = parent

grandfather.right = old\_left # save the old left values

old\_left.parent = grandfather

if to\_recolor:

parent.color = BLACK

node.color = RED

grandfather.color = RED

def \_recolor(self, grandfather):

grandfather.right.color = BLACK

grandfather.left.color = BLACK

if grandfather != self.root:

grandfather.color = RED

self.\_try\_rebalance(grandfather)

def \_find\_parent(self, value):

""" Finds a place for the value in our binary tree"""

def inner\_find(parent):

"""

Return the appropriate parent node for our new node as well as the side it should be on

"""

if value == parent.value:

return None, None

elif parent.value < value:

if parent.right.color == NIL: # no more to go

return parent, 'R'

return inner\_find(parent.right)

elif value < parent.value:

if parent.left.color == NIL: # no more to go

return parent, 'L'

return inner\_find(parent.left)

return inner\_find(self.root)

def find\_node(self, value):

def inner\_find(root):

if root is None or root == self.NIL\_LEAF:

return None

if value > root.value:

return inner\_find(root.right)

elif value < root.value:

return inner\_find(root.left)

else:

return root

found\_node = inner\_find(self.root)

return found\_node

def \_find\_in\_order\_successor(self, node):

right\_node = node.right

left\_node = right\_node.left

if left\_node == self.NIL\_LEAF:

return right\_node

while left\_node.left != self.NIL\_LEAF:

left\_node = left\_node.left

return left\_node

def \_get\_sibling(self, node):

"""

Returns the sibling of the node, as well as the side it is on

e.g

20 (A)

/ \

15(B) 25(C)

\_get\_sibling(25(C)) => 15(B), 'R'

"""

parent = node.parent

if node.value >= parent.value:

sibling = parent.left

direction = 'L'

else:

sibling = parent.right

direction = 'R'

return sibling, direction