**Приложение**

**Код реализации**

import random

class Node:

def \_\_init\_\_(self, key, weight):

self.key = key

self.weight = weight

self.total\_weight = weight

self.color = 1

self.left = None

self.right = None

self.parent = None

class RedBlackTree:

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

self.TNULL = Node(0, 0)

self.TNULL.color = 0

self.TNULL.total\_weight = 0

self.TNULL.left = None

self.TNULL.right = None

self.root = self.TNULL

def update\_total\_weight(self, node):

if node:

node.total\_weight = (

node.weight

+ (node.left.total\_weight if node.left else 0)

+ (node.right.total\_weight if node.right else 0)

)

def left\_rotate(self, x):

y = x.right

x.right = y.left

if y.left != self.TNULL:

y.left.parent = x

y.parent = x.parent

if x.parent is None:

self.root = y

elif x == x.parent.left:

x.parent.left = y

else:

x.parent.right = y

y.left = x

x.parent = y

self.update\_total\_weight(x)

self.update\_total\_weight(y)

def right\_rotate(self, x):

y = x.left

x.left = y.right

if y.right != self.TNULL:

y.right.parent = x

y.parent = x.parent

if x.parent is None:

self.root = y

elif x == x.parent.right:

x.parent.right = y

else:

x.parent.left = y

y.right = x

x.parent = y

self.update\_total\_weight(x)

self.update\_total\_weight(y)

def insert\_fix(self, k):

while k.parent.color == 1:

if k.parent == k.parent.parent.right:

u = k.parent.parent.left

if u.color == 1:

u.color = 0

k.parent.color = 0

k.parent.parent.color = 1

k = k.parent.parent

else:

if k == k.parent.left:

k = k.parent

self.right\_rotate(k)

k.parent.color = 0

k.parent.parent.color = 1

self.left\_rotate(k.parent.parent)

else:

u = k.parent.parent.right

if u.color == 1:

u.color = 0

k.parent.color = 0

k.parent.parent.color = 1

k = k.parent.parent

else:

if k == k.parent.right:

k = k.parent

self.left\_rotate(k)

k.parent.color = 0

k.parent.parent.color = 1

self.right\_rotate(k.parent.parent)

if k == self.root:

break

self.root.color = 0

def add(self, key, weight):

node = Node(key, weight)

node.parent = None

node.left = self.TNULL

node.right = self.TNULL

node.color = 1

y = None

x = self.root

while x != self.TNULL:

y = x

if node.key < x.key:

x = x.left

elif node.key == x.key:

x.weight = weight

self.update\_total\_weight(x)

return

else:

x = x.right

node.parent = y

if y is None:

self.root = node

elif node.key < y.key:

y.left = node

else:

y.right = node

current = node

while current.parent:

self.update\_total\_weight(current)

current = current.parent

else:

self.update\_total\_weight(current)

if node.parent is None:

node.color = 0

return

if node.parent.parent is None:

return

self.insert\_fix(node)

def delete(self, key):

self.delete\_node\_helper(self.root, key)

def delete\_node\_helper(self, node, key):

z = self.TNULL

while node != self.TNULL:

if node.key == key:

z = node

if node.key <= key:

node = node.right

else:

node = node.left

if z == self.TNULL:

return

y = z

y\_original\_color = y.color

if z.left == self.TNULL:

x = z.right

self.rb\_transplant(z, z.right)

elif z.right == self.TNULL:

x = z.left

self.rb\_transplant(z, z.left)

else:

y = self.minimum(z.right)

y\_original\_color = y.color

x = y.right

if y.parent == z:

x.parent = y

else:

self.rb\_transplant(y, y.right)

y.right = z.right

y.right.parent = y

self.rb\_transplant(z, y)

y.left = z.left

y.left.parent = y

y.color = z.color

self.update\_total\_weight(y)

current = x.parent

while current:

self.update\_total\_weight(current)

current = current.parent

if y\_original\_color == 0:

self.delete\_fix(x)

def rb\_transplant(self, u, v):

if u.parent == None:

self.root = v

elif u == u.parent.left:

u.parent.left = v

else:

u.parent.right = v

v.parent = u.parent

def delete\_fix(self, x):

while x != self.root and x.color == 0:

if x == x.parent.left:

s = x.parent.right

if s.color == 1:

s.color = 0

x.parent.color = 1

self.left\_rotate(x.parent)

s = x.parent.right

if s.left.color == 0 and s.right.color == 0:

s.color = 1

x = x.parent

else:

if s.right.color == 0:

s.left.color = 0

s.color = 1

self.right\_rotate(s)

s = x.parent.right

s.color = x.parent.color

x.parent.color = 0

s.right.color = 0

self.left\_rotate(x.parent)

x = self.root

else:

s = x.parent.left

if s.color == 1:

s.color = 0

x.parent.color = 1

self.right\_rotate(x.parent)

s = x.parent.left

if s.right.color == 0 and s.left.color == 0:

s.color = 1

x = x.parent

else:

if s.left.color == 0:

s.right.color = 0

s.color = 1

self.left\_rotate(s)

s = x.parent.left

s.color = x.parent.color

x.parent.color = 0

s.left.color = 0

self.right\_rotate(x.parent)

x = self.root

x.color = 0

def get(self):

if self.root.total\_weight == 0:

return None

threshold = random.uniform(0, self.root.total\_weight)

current = self.root

while current != self.TNULL:

left\_weight = current.left.total\_weight if current.left != self.TNULL else 0

if threshold < left\_weight:

current = current.left

elif threshold < left\_weight + current.weight:

return current.key

else:

threshold -= (left\_weight + current.weight)

current = current.right

return None

def minimum(self, node):

while node.left != self.TNULL:

node = node.left

return node

**Код тестов**

import time

import tracemalloc

from random import uniform

import Struct

import Struct2

test\_tree = Struct.Tree()

test\_tree\_2 = Struct2.RedBlackTree()

def generate\_data(n,min\_weight,max\_weight):

return [(i,uniform(min\_weight,max\_weight)) for i in range(1,n)]

def test\_with\_generate\_date(test\_struct,n,m,min\_weigh,max\_weight):

tracemalloc.start()

g\_data = generate\_data(n,min\_weigh,max\_weight)

start\_time = time.time()

for data in g\_data:

test\_struct.add(\*data)

end\_time = time.time()

snapshot = tracemalloc.take\_snapshot()

print(snapshot.statistics('lineno')[0])

tracemalloc.stop()

print(f"Добавление {n} элементов:", end\_time-start\_time)

start\_time = time.time()

for i in range(m):

test\_struct.get()

end\_time = time.time()

print(f"Получение {m} элементов из дерева с {n} элементами:", end\_time - start\_time)

start\_time = time.time()

for i in range(m):

test\_struct.delete(i+1)

end\_time = time.time()

print(f"Удаление {m} элементов из дерева с {n} элементами:", end\_time - start\_time)

def static\_test(weights,struct,n):

results = [0] \* len(weights)

t = len(weights)\*10\*\*n

for \_ in range(t):

key = struct.get()

results[key] += 1

total\_weight = struct.root.total\_weight

deviation = 0

for i in range(1, 101):

deviation += abs(results[i]-(weights[i]/total\_weight)\*t)

deviation = deviation\*100/t

print(f"Отклонение: {deviation:.2f}% после {t} операций get")

def test\_with\_static\_date():

#test1

test\_struct = Struct2.RedBlackTree()

weights = [0]

for i in range(1, 101):

test\_struct.add(i, i)

weights.append(i)

print("Test1")

for i in range(1,4):static\_test(weights,test\_struct,i)

#test2

test\_struct\_2 = Struct2.RedBlackTree()

weights\_2 = [0]

for i in range(1, 101):

test\_struct\_2.add(i, i)

if i % 2:

test\_struct\_2.delete(i)

weights\_2.append(0)

else:weights\_2.append(i)

print("Test2")

for i in range(1,4):static\_test(weights\_2,test\_struct\_2,i)

#test3

test\_struct\_3 = Struct2.RedBlackTree()

weights\_3 = [0]

for i in range(1, 101):

test\_struct\_3.add(i, i)

test\_struct\_3.add(i,2\*i)

weights\_3.append(2\*i)

print("Test3")

for i in range(1,4):static\_test(weights\_3,test\_struct\_3,i)

#test4

test\_struct\_4 = Struct2.RedBlackTree()

weights\_4 = [0]

for i in range(1, 101):

test\_struct\_4.add(i, i)

test\_struct\_4.add(i,2\*i)

if i % 2:

test\_struct\_4.delete(i)

weights\_4.append(0)

else:weights\_4.append(2\*i)

print("Test4")

for i in range(1,4):static\_test(weights\_4,test\_struct\_4,i)

print("Struct\_2")

test\_with\_generate\_date(test\_tree\_2,10000,10000,0.001,10)

test\_with\_generate\_date(test\_tree\_2,100000,1000000,0.001,10)

test\_with\_generate\_date(test\_tree\_2,1000000,1000000,0.001,10)

test\_with\_static\_date()