3.3.39

private Node moveRedLeft(Node h) {

flipColors(h);

if (isRed(h.right.left)) {

h.right = rotateRight(h.right);

h = rotateLeft(h);

}

return h;

}

public void deleteMin() {

if (!isRed(root.left) && !isRed(root.right))

root.color = RED;

root = deleteMin(root);

if (!isEmpty()) root.color = BLACK;

}

private Node balance(Node h) {

if (isRed(h.right)) h = rotateLeft(h);

if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);

if (isRed(h.left) && isRed(h.right)) flipColors(h);

return h;

}

private void flipColors(Node h) {

h.color = !h.color;

h.left.color = !h.left.color;

h.right.color = !h.right.color;

}

3.3.40

private Node moveRedRight(Node h) {

flipColors(h);

if (!isRed(h.left.left)) h = rotateRight(h);

return h;

}

public void deleteMax() {

if (!isRed(root.left) && !isRed(root.right)) root.color = RED;

root = deleteMax(root);

if (!isEmpty()) root.color = BLACK;

}

private Node deleteMax(Node h) {

if (isRed(h.left)) h = rotateRight(h);

if (h.right == null) return null;

if (!isRed(h.right) && !isRed(h.right.left)) h = moveRedRight(h);

h.right = deleteMax(h.right);

return balance(h);

}

private Node balance(Node h) {

if (isRed(h.right)) h = rotateLeft(h);

if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);

if (isRed(h.left) && isRed(h.right)) flipColors(h);

return h;

}

private void flipColors(Node h) {

h.color = !h.color;

h.left.color = !h.left.color;

h.right.color = !h.right.color;

}

3.3.41

public void delete(Key key) {

if (!isRed(root.left) && !isRed(root.right)) {

root.color = RED;

}

root = delete(root, key);

if (!isEmpty()) root.color = BLACK;

}

private Node delete(Node h, Key key) {

if (key.compareTo(h.key) < 0) {

if (!isRed(h.left) && !isRed(h.left.left)) h = moveRedLeft(h);

h.left = delete(h.left, key);

} else {

if (isRed(h.left)) h = rotateRight(h);

if (key.compareTo(h.key) == 0 && h.right == null) return null;

if (!isRed(h.right) && !isRed(h.right.left)) h = moveRedRight(h);

if (key.compareTo(h.key) == 0) {

h.val = get(h.right, min(h.right).key);

h.key = min(h.right).key;

h.right = deleteMin(h.right);

} else {

h.right = delete(h.right, key);

}

}

return balance(h);

}

private Node moveRedLeft(Node h) {

flipColors(h);

if (isRed(h.right.left)) {

h.right = rotateRight(h.right);

h = rotateLeft(h);

}

return h;

}

private Node moveRedRight(Node h) {

flipColors(h);

if (!isRed(h.left.left)) h = rotateRight(h);

return h;

}

private Node balance(Node h) {

if (isRed(h.right)) h = rotateLeft(h);

if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);

if (isRed(h.left) && isRed(h.right)) flipColors(h);

return h;

}

private Node deleteMin(Node h) {

if (h.left == null) return null;

if (!isRed(h.left) && !isRed(h.left.left)) h = moveRedLeft(h);

h.left = deleteMin(h.left);

return balance(h);

}

private Node min(Node h) {

if (h.left == null) return h;

return min(h.left);

}

private void flipColors(Node h) {

h.color = !h.color;

h.left.color = !h.left.color;

h.right.color = !h.right.color;

}

private boolean isRed(Node x) {

if (x == null) return false;

return x.color == RED;

}

private Node rotateLeft(Node h) {

Node x = h.right;

h.right = x.left;

x.left = h;

x.color = h.color;

h.color = RED;

return x;

}

private Node rotateRight(Node h) {

Node x = h.left;

h.left = x.right;

x.right = h;

x.color = h.color;

h.color = RED;

return x;

}

3.3.42

import random

# Red-Black Tree Node Definition

class Node:

def \_\_init\_\_(self, key, color='red', left=None, right=None, parent=None):

self.key = key

self.color = color

self.left = left

self.right = right

self.parent = parent

class RedBlackTree:

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

self.NIL\_LEAF = Node(None, 'black') # Sentinel NIL leaf node

self.root = self.NIL\_LEAF

# Insert a new key into the tree

def insert(self, key):

new\_node = Node(key, color='red', left=self.NIL\_LEAF, right=self.NIL\_LEAF)

parent = None

current = self.root

while current != self.NIL\_LEAF:

parent = current

if new\_node.key < current.key:

current = current.left

else:

current = current.right

new\_node.parent = parent

if not parent:

self.root = new\_node

elif new\_node.key < parent.key:

parent.left = new\_node

else:

parent.right = new\_node

self.fix\_insert(new\_node)

# Fix the tree after insertion to maintain Red-Black properties

def fix\_insert(self, node):

while node != self.root and node.parent.color == 'red':

if node.parent == node.parent.parent.left:

uncle = node.parent.parent.right

if uncle.color == 'red':

node.parent.color = 'black'

uncle.color = 'black'

node.parent.parent.color = 'red'

node = node.parent.parent

else:

if node == node.parent.right:

node = node.parent

self.left\_rotate(node)

node.parent.color = 'black'

node.parent.parent.color = 'red'

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

else:

uncle = node.parent.parent.left

if uncle.color == 'red':

node.parent.color = 'black'

uncle.color = 'black'

node.parent.parent.color = 'red'

node = node.parent.parent

else:

if node == node.parent.left:

node = node.parent

self.right\_rotate(node)

node.parent.color = 'black'

node.parent.parent.color = 'red'

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

self.root.color = 'black'

def left\_rotate(self, x):

y = x.right

x.right = y.left

if y.left != self.NIL\_LEAF:

y.left.parent = x

y.parent = x.parent

if not x.parent:

self.root = y

elif x == x.parent.left:

x.parent.left = y

else:

x.parent.right = y

y.left = x

x.parent = y

def right\_rotate(self, y):

x = y.left

y.left = x.right

if x.right != self.NIL\_LEAF:

x.right.parent = y

x.parent = y.parent

if not y.parent:

self.root = x

elif y == y.parent.right:

y.parent.right = x

else:

y.parent.left = x

x.right = y

y.parent = x

# Count red nodes

def count\_red\_nodes(self, node=None):

if node is None:

node = self.root

if node == self.NIL\_LEAF:

return 0

count = 1 if node.color == 'red' else 0

count += self.count\_red\_nodes(node.left)

count += self.count\_red\_nodes(node.right)

return count

# Get total number of nodes

def count\_total\_nodes(self, node=None):

if node is None:

node = self.root

if node == self.NIL\_LEAF:

return 0

count = 1

count += self.count\_total\_nodes(node.left)

count += self.count\_total\_nodes(node.right)

return count

# Perform trials

def perform\_trials(num\_trials, num\_keys):

percentages = []

for \_ in range(num\_trials):

tree = RedBlackTree()

for \_ in range(num\_keys):

tree.insert(random.randint(0, 1000000)) # Insert random keys

red\_node\_count = tree.count\_red\_nodes()

total\_node\_count = tree.count\_total\_nodes()

percentage\_red\_nodes = (red\_node\_count / total\_node\_count) \* 100

percentages.append(percentage\_red\_nodes)

return percentages

# Run experiments for N = 10^4, 10^5, 10^6

num\_trials = 100

for N in [10\*\*4, 10\*\*5, 10\*\*6]:

percentages = perform\_trials(num\_trials, N)

avg\_percentage = sum(percentages) / len(percentages)

print(f"Average percentage of red nodes for N={N}: {avg\_percentage:.2f}%")

3.3.44

import random

import math

import numpy as np

import matplotlib.pyplot as plt

class Node:

def \_\_init\_\_(self, key, color='red', left=None, right=None, parent=None, depth=0):

self.key = key

self.color = color

self.left = left

self.right = right

self.parent = parent

self.depth = depth # Track the depth of each node

class RedBlackTree:

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

self.NIL\_LEAF = Node(None, 'black') # Sentinel NIL leaf node

self.root = self.NIL\_LEAF

def insert(self, key):

new\_node = Node(key, color='red', left=self.NIL\_LEAF, right=self.NIL\_LEAF)

parent = None

current = self.root

depth = 0 # Depth starts at 0 for the root

while current != self.NIL\_LEAF:

parent = current

depth += 1 # Increment depth for each level down

if new\_node.key < current.key:

current = current.left

else:

current = current.right

new\_node.parent = parent

new\_node.depth = depth

if not parent:

self.root = new\_node

elif new\_node.key < parent.key:

parent.left = new\_node

else:

parent.right = new\_node

self.fix\_insert(new\_node)

def fix\_insert(self, node):

while node != self.root and node.parent.color == 'red':

if node.parent == node.parent.parent.left:

uncle = node.parent.parent.right

if uncle.color == 'red':

node.parent.color = 'black'

uncle.color = 'black'

node.parent.parent.color = 'red'

node = node.parent.parent

else:

if node == node.parent.right:

node = node.parent

self.left\_rotate(node)

node.parent.color = 'black'

node.parent.parent.color = 'red'

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

else:

uncle = node.parent.parent.left

if uncle.color == 'red':

node.parent.color = 'black'

uncle.color = 'black'

node.parent.parent.color = 'red'

node = node.parent.parent

else:

if node == node.parent.left:

node = node.parent

self.right\_rotate(node)

node.parent.color = 'black'

node.parent.parent.color = 'red'

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

self.root.color = 'black'

def left\_rotate(self, x):

y = x.right

x.right = y.left

if y.left != self.NIL\_LEAF:

y.left.parent = x

y.parent = x.parent

if not x.parent:

self.root = y

elif x == x.parent.left:

x.parent.left = y

else:

x.parent.right = y

y.left = x

x.parent = y

def right\_rotate(self, y):

x = y.left

y.left = x.right

if x.right != self.NIL\_LEAF:

x.right.parent = y

x.parent = y.parent

if not y.parent:

self.root = x

elif y == y.parent.right:

y.parent.right = x

else:

y.parent.left = x

x.right = y

y.parent = x

# Calculate the internal path length (sum of all node depths)

def internal\_path\_length(self, node=None):

if node is None:

node = self.root

if node == self.NIL\_LEAF:

return 0

return node.depth + self.internal\_path\_length(node.left) + self.internal\_path\_length(node.right)

# Perform trials and collect data

def perform\_trials(num\_trials, num\_keys):

average\_path\_lengths = []

for \_ in range(num\_trials):

tree = RedBlackTree()

for \_ in range(num\_keys):

tree.insert(random.randint(0, 1000000)) # Insert random keys

total\_path\_length = tree.internal\_path\_length()

average\_path\_length = total\_path\_length / num\_keys

average\_path\_lengths.append(average\_path\_length)

mean = np.mean(average\_path\_lengths)

std\_dev = np.std(average\_path\_lengths)

return mean, std\_dev

# Run experiments for N from 1 to 10,000

num\_trials = 1000

results = []

N\_values = range(1, 10001)

for N in N\_values:

mean, std\_dev = perform\_trials(num\_trials, N)

results.append((N, mean, std\_dev))

# Extract results for plotting

Ns, means, std\_devs = zip(\*results)

# Plotting results using a Tufte-style plot

plt.figure(figsize=(10, 6))

plt.errorbar(Ns, means, yerr=std\_devs, fmt='o', label="Empirical Average Path Length")

plt.plot(Ns, [math.log2(N) - 0.5 for N in Ns], label="lg(N) - 0.5", color="red")

plt.xlabel("N (Number of Nodes)")

plt.ylabel("Average Path Length")

plt.title("Average Path Length in Red-Black BST (Empirical vs Theoretical)")

plt.legend()

plt.grid(True, which="both", linestyle='--', linewidth=0.5)

plt.xscale("log")

plt.yscale("log")

plt.show()

import random

import math

import numpy as np

import matplotlib.pyplot as plt

class Node:

def \_\_init\_\_(self, key, color='red', left=None, right=None, parent=None, depth=0):

self.key = key

self.color = color

self.left = left

self.right = right

self.parent = parent

self.depth = depth # Track depth of each node

class RedBlackTree:

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

self.NIL\_LEAF = Node(None, 'black') # Sentinel NIL leaf node

self.root = self.NIL\_LEAF

self.rotations = 0 # Count rotations

self.node\_splits = 0 # Count node splits (color flips)

def insert(self, key):

new\_node = Node(key, color='red', left=self.NIL\_LEAF, right=self.NIL\_LEAF)

parent = None

current = self.root

depth = 0 # Depth starts at 0 for the root

while current != self.NIL\_LEAF:

parent = current

depth += 1 # Increment depth for each level down

if new\_node.key < current.key:

current = current.left

else:

current = current.right

new\_node.parent = parent

new\_node.depth = depth

if not parent:

self.root = new\_node

elif new\_node.key < parent.key:

parent.left = new\_node

else:

parent.right = new\_node

self.fix\_insert(new\_node)

def fix\_insert(self, node):

while node != self.root and node.parent.color == 'red':

if node.parent == node.parent.parent.left:

uncle = node.parent.parent.right

if uncle.color == 'red':

# Node split (color flip)

node.parent.color = 'black'

uncle.color = 'black'

node.parent.parent.color = 'red'

node = node.parent.parent

self.node\_splits += 1

else:

if node == node.parent.right:

node = node.parent

self.left\_rotate(node)

node.parent.color = 'black'

node.parent.parent.color = 'red'

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

else:

uncle = node.parent.parent.left

if uncle.color == 'red':

# Node split (color flip)

node.parent.color = 'black'

uncle.color = 'black'

node.parent.parent.color = 'red'

node = node.parent.parent

self.node\_splits += 1

else:

if node == node.parent.left:

node = node.parent

self.right\_rotate(node)

node.parent.color = 'black'

node.parent.parent.color = 'red'

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

self.root.color = 'black'

def left\_rotate(self, x):

y = x.right

x.right = y.left

if y.left != self.NIL\_LEAF:

y.left.parent = x

y.parent = x.parent

if not x.parent:

self.root = y

elif x == x.parent.left:

x.parent.left = y

else:

x.parent.right = y

y.left = x

x.parent = y

self.rotations += 1 # Count the rotation

def right\_rotate(self, y):

x = y.left

y.left = x.right

if x.right != self.NIL\_LEAF:

x.right.parent = y

x.parent = y.parent

if not y.parent:

self.root = x

elif y == y.parent.right:

y.parent.right = x

else:

y.parent.left = x

x.right = y

y.parent = x

self.rotations += 1 # Count the rotation

# Perform trials and collect data

def perform\_trials(num\_trials, num\_keys):

total\_rotations = 0

total\_splits = 0

for \_ in range(num\_trials):

tree = RedBlackTree()

for \_ in range(num\_keys):

tree.insert(random.randint(0, 1000000)) # Insert random keys

total\_rotations += tree.rotations

total\_splits += tree.node\_splits

avg\_rotations = total\_rotations / num\_trials

avg\_splits = total\_splits / num\_trials

return avg\_rotations, avg\_splits

# Run experiments for N from 1 to 10,000

num\_trials = 1000

results = []

N\_values = range(1, 10001)

for N in N\_values:

avg\_rotations, avg\_splits = perform\_trials(num\_trials, N)

results.append((N, avg\_rotations, avg\_splits))

# Extract results for plotting

Ns, avg\_rotations, avg\_splits = zip(\*results)

# Plotting results

plt.figure(figsize=(10, 6))

# Plot the number of rotations

plt.plot(Ns, avg\_rotations, label="Average Rotations", color="blue")

# Plot the number of node splits

plt.plot(Ns, avg\_splits, label="Average Node Splits", color="green")

plt.xlabel("N (Number of Nodes)")

plt.ylabel("Average Number of Operations")

plt.title("Average Number of Rotations and Node Splits in Red-Black BST")

plt.legend()

plt.grid(True, which="both", linestyle='--', linewidth=0.5)

plt.xscale("log")

plt.yscale("log")

plt.show()

3.3.46

class RedBlackTree:

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

self.NIL\_LEAF = Node(None, 'black')

self.root = self.NIL\_LEAF

# Insert and other methods remain the same...

def tree\_height(self, node=None):

if node is None:

node = self.root

if node == self.NIL\_LEAF:

return 0

return 1 + max(self.tree\_height(node.left), self.tree\_height(node.right))

# Perform trials and collect height data

def perform\_trials\_for\_height(num\_trials, num\_keys):

heights = []

for \_ in range(num\_trials):

tree = RedBlackTree()

for \_ in range(num\_keys):

tree.insert(random.randint(0, 1000000))

heights.append(tree.tree\_height())

return np.mean(heights)

# Run experiments for N from 1 to 10,000

num\_trials = 100

results = []

N\_values = range(1, 10001, 100) # Reduced steps for faster plotting

for N in N\_values:

avg\_height = perform\_trials\_for\_height(num\_trials, N)

results.append((N, avg\_height))

# Plotting the height

Ns, avg\_heights = zip(\*results)

plt.plot(Ns, avg\_heights, label="Average Height")

plt.xlabel("N (Number of Nodes)")

plt.ylabel("Average Tree Height")

plt.title("Average Height of Red-Black BST")

plt.xscale("log")

plt.grid(True)

plt.show()