Data Structures and Algorithms 2
Course Project 2024

ICS2210

Malcolm Borg 3604H

Contents

Knuth shuffle

The knuth_shuffle function implements the Fisher-Yates (or Knuth) shuffle algorithm to randomly permute the elements of an array arr. It iterates backward through the list, swapping each element with a randomly selected one from a subset including itself and all preceding elements. The function returns the shuffled array.

AVL tree

Class `Node`

- Purpose: Represents a node within the AVL tree.
- Attributes:
    - `key`: The value or key contained in the node.
    - `left`: Reference to the left child node.
    - `right`: Reference to the right child node.
    - `height`: The height of the node in the tree.

Class `AVLTree`

- Purpose: Manages the AVL tree operations such as insertion, rotation, and traversal.
- Attributes:
    - `root`: The root node of the tree.
    - `steps`: A list tracking the number of steps taken during each insert operation.
    - `rotations`: A list tracking the number of rotations performed during each insert.

Method `__init__`

- Initializes an empty AVL tree with no root and empty lists for tracking steps and rotations during operations.

Method `insert`

- Inserts a new key into the AVL tree while maintaining its balanced property.
- Parameters:
    - `key`: The value to be inserted into the tree.

Method `_insert`

- Recursively inserts a new key into the subtree rooted at the given node, maintaining the AVL property through necessary rotations.
- Parameters:

    - `root`: The root node of the current subtree.
    - `key`: The key to be inserted.
    - `steps`: Counter for the number of steps taken during insertion.
    - `rotations`: Counter for the number of rotations performed.
- Returns: The new root of the subtree, steps, and rotations after insertion.

Method `leftRotate`

- Performs a left rotation on the subtree rooted at the given node to maintain the balance of the AVL tree.
- Parameters:
    - `z`: The root node of the subtree to be rotated.

Method `rightRotate`

- Performs a right rotation on the subtree rooted at the given node to maintain the balance of the AVL tree.
- Parameters:
    - `y`: The root node of the subtree to be rotated.

Method `getHeight`

- Returns the height of a node.
- Parameters:
    - `root`: The node whose height is to be calculated.

Method `getBalance`

- Calculates and returns the balance factor of a node as the difference in heights of its left and right subtrees.
- Parameters:
    - `root`: The node whose balance factor is to be calculated.

Method `preOrder`

- Prints the keys of the nodes in the AVL tree in preorder traversal (root, left, right).
- Parameters:
    - `root`: The starting node for the traversal. If not provided, starts from the root of the tree.

Method `count_leaves`

- Counts and returns the number of leaf nodes in the subtree rooted at the given node.
- Parameters:
    - `node`: The root node of the subtree for which leaf nodes are counted.

Method `printStatistics`

- Prints statistical data about the operations performed on the AVL tree, including steps and rotations for insertions, as well as the tree's height and the number of leaf nodes. Also includes statistical measures like minimum, maximum, mean, median, and standard deviation for both steps and rotations.

Red-Black tree

Class `Node`

- Purpose: Represents a node in the Red-Black Tree.
- Attributes:
    - `data`: The value or key contained in the node.
    - `color`: The color of the node, which can be "red" or "black".
    - `parent`: Reference to the parent node.
    - `left`: Reference to the left child node.
    - `right`: Reference to the right child node.

Class `RedBlackTree`

- Purpose: Manages operations such as insertion and balancing on the Red-Black Tree.
- Attributes:
    - `TNULL`: A sentinel node used to represent NULL children and the root's parent.
    - `root`: The root node of the tree.
    - `steps`: A list that tracks the number of steps taken during each insert operation.
    - `rotations`: A list that tracks the number of rotations performed during each insert.

Method `__init__`

- Initializes a Red-Black Tree with a sentinel node `TNULL` representing all leaf nodes and the tree's root initially set to `TNULL`.

Method `insert`

- Inserts a new key into the Red-Black Tree and maintains the required properties through balancing operations.
- Parameters:
    - `key`: The value to be inserted into the tree.

Method `left_rotate` and `right_rotate`

- Performs left and right rotations on nodes to maintain tree balance during insertions.
- Parameters:
    - `x`: The node around which the rotation is to be performed.

Method `fix_insert`

- Fixes the Red-Black Tree properties after insertion if they are violated.
- Parameters:
    - `k`: The newly inserted node that may have caused a violation.

Method `count_leaves`

- Recursively counts the number of leaf nodes in the subtree rooted at a given node.
- Parameters:
    - `node`: The root node of the subtree.

Method `get_number_of_leaves`

- Returns the total number of leaf nodes in the entire Red-Black Tree.

Method `compute_height`

- Recursively computes the height of the subtree rooted at the given node.
- Parameters:
    - `node`: The root node of the subtree.

Method `get_tree_height`

- Returns the height of the entire Red-Black Tree.

Method `printStatistics`

- Prints statistical data about the operations performed on the Red-Black Tree, including details on steps and rotations, tree height, and the number of leaf nodes. Statistical measures such as minimum, maximum, mean, median, and standard deviation for steps and rotations are also displayed.

Skip List tree

Class `Node`

- Purpose: Represents an individual node within the skip list.
- Attributes:
    - `key`: The key associated with the node, used for ordering.
    - `value`: The value stored in the node.
    - `forward`: A list of pointers (or references) to nodes that represent the next elements at various levels of the list.

Class `SkipList`

- Purpose: Manages the entire structure of the skip list, facilitating operations like insertion and providing statistical insights.
- Attributes:
    - `MAX_LVL`: The maximum level of the skip list.
    - `P`: The probability factor used to decide the random level of new nodes.
    - `header`: The header node of the skip list, acting as an entry point.
    - `level`: The current maximum level of nodes within the skip list.
    - `steps`: A list that accumulates the number of steps (or node visits) required to reach the insertion point for each operation.
    - `promotions`: A list that tracks how many levels each new node is inserted into.

Method `__init__`

- Initializes the skip list with specified maximum levels and probability.
- Parameters:
    - `max_lvl`: The maximum possible level for nodes.
    - `P`: The probability threshold for incrementing levels during node insertion.

Method `create_node`

- Creates and returns a new `Node` object with specified level, key, and value.
- Parameters:
    - `lvl`: The level of the node.
    - `key`: The key of the node.
    - `value`: The value associated with the node.

Method `random_level`

- Determines the level for a new node based on the probability `P`, ensuring that higher levels are progressively less likely.

Method `insert`

- Inserts a new key-value pair into the skip list, adjusting the structure as necessary to maintain the properties of skip lists.
- Parameters:
    - `key`: The key for the new node.
    - `value`: The value for the new node.
- Process:
    - It traverses from the highest level down to the base level to find the correct insertion point, tracking the number of steps.
    - Inserts the new node at the appropriate level determined by `random_level`.
    - Updates forward pointers and levels, recording promotions.

Method `printStatistics`

- Prints various statistics about the skip list operations, such as minimum, maximum, mean, standard deviation, and median of the steps to the insertion point and promotions. Also reports the current maximum level of the skip list.
- This method aids in analyzing the efficiency and behavior of the skip list over time based on the inserted elements.

Statistics

Steps

Tree	minimum	maximum	mean	sd	median
AVL	11	15	13.425	0.9159..	14
Red Black	10	16	13.897	0.9630..	14
Skip List	170	217	183.8674…	9.2630…	181.0

Rotations

(Promotions for skip list)

Tree	minimum	maximum	mean	sd	median
AVL	0	1	0.475	0.4996…	0
Red Black	0	2	0.598	0.7931…	0
Skip List	1	6	2.0105…	1.3392…	2.0

Height

AVL tree: 15
Red Black tree: 16

Number of Leaves

AVL tree: 2584
Red Black tree: 2579

Max Level

Skip List: 5

Evaluation Of Statistics

Both the AVL and Red-Black trees are performing efficiently in terms of balancing and maintaining a reasonable height, which ensures quick search times. The AVL tree, with fewer rotations and a slightly shorter height, is a bit more tightly balanced than the Red-Black tree, suggesting it might be faster for read-heavy applications. The Red-Black tree, with its ability to handle slightly more rotations efficiently, indicates a better performance where more frequent updates are involved.

The Skip List shows a wider range in steps to the insertion point, reflecting its probabilistic nature, which can lead to variable performance but offers benefits in environments needing high concurrency. The statistics demonstrate that each structure is well-optimized for its design goals, with clear distinctions in their suitability for different types of applications.

Overall Evaluation

When deciding between AVL trees, Red-Black trees, and Skip Lists for real-world applications, the choice largely hinges on the specific needs of the project. If the priority is to have the fastest possible search times because of a lot of read operations, AVL trees are ideal because of their tight balancing, which guarantees quick searches. However, if you're dealing with an environment where both updates and searches are frequent, Red-Black trees are a better fit. They strike a good balance between quick insertions and deletions and efficient search times, making them perfect for such systems.

On the other hand, if your application requires high scalability and concurrent operations, Skip Lists are the way to go. They handle multiple simultaneous updates efficiently, which is great for real-time data processing or distributed systems. Each data structure has its strengths, so the best choice depends on understanding the specific trade-offs that align with the demands of your application.

```
# Create an array of 5,000 integers whose values start at 1 and end at
# 5,000

arr = [0] * 5000
for i in range(0, 5000):
   arr[i] = i + 1

# Implement the Knuth shuffle algorithm to randomise the order of the
# elements in the array. The implementation should yours; don't use any
# inbuilt array shuffling function from your programming language.

import random
def knuth_shuffle(arr):
   n = len(arr)
   for i in range(n-1, 0, -1):
       j = random.randint(0, i)
       arr[i], arr[j] = arr[j], arr[i]
   return arr

arr = knuth_shuffle(arr)

# Insert all the 5,000 integers from the array into an AVL tree, a RedBlack tree, and
a Skip List. The implementations of these three data
# structures should be your own.

import avl
import rbt
import sl

avl_tree = avl.AVLTree()
rb_tree = rbt.RedBlackTree()
sl_e = sl.SkipList(5, 0.5)

for i in range(0, 5000):
   avl_tree.insert(arr[i])
   rb_tree.insert(arr[i])
   sl_e.insert(arr[i], arr[i])

# Create another array containing 1,000 random integers in the range
# [0…100,000]. This array may contain duplicates.
arr = [0] * 1000
```

```
for i in range(0, 1000):
   arr[i] = random.randint(0, 100000)

# Insert all the elements from this second array into the trees. When
# inserting, keep track of statistics

# reset the stats to start fresh
avl_tree.reset_stats()
rb_tree.reset_stats()
sl_e.reset_stats()

for i in arr:
   avl_tree.insert(i)
   rb_tree.insert(i)
   sl_e.insert(i, i)

# Print the statistics
avl_tree.printStatistics()
rb_tree.printStatistics()
sl_e.printStatistics()
```

```
import statistics


class Node:
   def __init__(self, key):
       self.key = key
       self.left = None
       self.right = None
       self.height = 1

class AVLTree:
   def __init__(self):
       self.root = None
       self.steps = []
       self.rotations = []

   def insert(self, key):
       if not self.root:
           self.steps.append(0)
           self.root = Node(key)
       else:
           self.root, steps, rotations = self._insert(self.root, key, 0, 0)
```

```
            self.steps.append(steps)  # Append steps taken for this insertion
            self.rotations.append(rotations)  # Append rotations taken for this
insertion

    def _insert(self, root, key, steps=0, rotations=0):
        if not root:
            return Node(key), steps, rotations
        elif key < root.key:
            root.left, steps, rotations = self._insert(root.left, key, steps+1,
rotations)
        else:
            root.right, steps, rotations = self._insert(root.right, key, steps+1,
rotations)

        # Update the height of the ancestor node
        root.height = 1 + max(self.getHeight(root.left),
                              self.getHeight(root.right))

        # balance the tree after insertion
        balance = self.getBalance(root)

        if balance > 1 and key < root.left.key:
            return self.rightRotate(root), steps, rotations+1

        if balance < -1 and key > root.right.key:
            return self.leftRotate(root), steps, rotations+1

        if balance > 1 and key > root.left.key:
            root.left = self.leftRotate(root.left)
            return self.rightRotate(root), steps, rotations+1

        if balance < -1 and key < root.right.key:
            root.right = self.rightRotate(root.right)
            return self.leftRotate(root), steps, rotations+1

        if root == self.root:
            self.steps.append(steps)

        return root, steps, rotations

    def leftRotate(self, z):
        y = z.right
```

```
        T2 = y.left
        y.left = z
        z.right = T2
        z.height = 1 + max(self.getHeight(z.left), self.getHeight(z.right))
        y.height = 1 + max(self.getHeight(y.left), self.getHeight(y.right))
        return y

    def rightRotate(self, y):
        x = y.left
        T2 = x.right
        x.right = y
        y.left = T2
        y.height = 1 + max(self.getHeight(y.left), self.getHeight(y.right))
        x.height = 1 + max(self.getHeight(x.left), self.getHeight(x.right))
        return x

    def getHeight(self, root):
        if not root:
            return 0
        return root.height

    def getBalance(self, root):
        if not root:
            return 0
        return self.getHeight(root.left) - self.getHeight(root.right)

    def preOrder(self, root=None):
        if root is None:
            root = self.root
        if root is not None:
            print("{0} ".format(root.key), end="")
            self.preOrder(root.left)
            self.preOrder(root.right)

    def count_leaves(self, node):
        # If the tree is empty, return 0
        if not node:
            return 0
        # If the node is a leaf node
        if not node.left and not node.right:
            return 1
        # Recursively count the leaves in both the left and right subtree
```

```
        return self.count_leaves(node.left) + self.count_leaves(node.right)

    def reset_stats(self):
        self.steps = []
        self.rotations = []

    def printStatistics(self):
        print("AVL Tree")
        print("Steps:")
        print("minimum: ", min(self.steps))
        print("maximum: ", max(self.steps))
        print("mean", sum(self.steps) / len(self.steps))
        print("standard deviation: ", statistics.stdev(self.steps))
        print("median: ", self.steps[len(self.steps) // 2])

        print("Rotations:")
        print("minimum: ", min(self.rotations))
        print("maximum: ", max(self.rotations))
        print("mean", sum(self.rotations) / len(self.rotations))
        print("standard deviation: ", statistics.stdev(self.rotations))
        print("median: ", self.rotations[len(self.rotations) // 2])

        print("Height of the tree: ", self.root.height)
        print("number of leaves: ", self.count_leaves(self.root))
        print("")
```

```
import statistics

class Node:
    def __init__(self, data, color="red"):
        self.data = data
        self.color = color
        self.parent = None
        self.left = None
        self.right = None

class RedBlackTree:
    def __init__(self):
        self.TNULL = Node(0, "black")  # Null nodes are black
        self.TNULL.left = None
        self.TNULL.right = None
        self.root = self.TNULL
        self.steps = []
```

```
        self.rotations = []

    def insert(self, key):
        node = Node(key)
        node.parent = None
        node.data = key
        node.left = self.TNULL
        node.right = self.TNULL
        node.color = "red"  # New nodes are red

        parent = None
        current = self.root

        curSteps = 0

        while current != self.TNULL:
            parent = current
            if node.data < current.data:
                current = current.left
            else:
                current = current.right
            curSteps += 1

        node.parent = parent
        if parent is None:
            self.root = node
        elif node.data < parent.data:
            parent.left = node
        else:
            parent.right = node

        if node.parent is None:
            node.color = "black"
            self.steps.append(curSteps)
            self.rotations.append(0)
            return

        if node.parent.parent is None:
            self.steps.append(curSteps)
            self.rotations.append(0)
            return
```

```
        # Fix the tree
        self.steps.append(curSteps)
        # rotations are added in the fix_insert method
        self.fix_insert(node)

    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

    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

    def fix_insert(self, k):
        curRotations = 0

        while k.parent.color == "red":
            if k.parent == k.parent.parent.right:
```

```
            u = k.parent.parent.left  # left uncle
            if u.color == "red": #red uncle
                u.color = "black"
                k.parent.color = "black"
                k.parent.parent.color = "red"
                k = k.parent.parent
            else:  # black uncle
                if k == k.parent.left:
                    k = k.parent
                    self.right_rotate(k)
                    curRotations += 1

                k.parent.color = "black"
                k.parent.parent.color = "red"
                self.left_rotate(k.parent.parent)
                curRotations += 1
        else:  # Mirror case: Parent is the left child of its parent
            u = k.parent.parent.right  # right uncle

            if u.color == "red":  # red uncle
                u.color = "black"
                k.parent.color = "black"
                k.parent.parent.color = "red"
                k = k.parent.parent
            else:  # black uncle
                if k == k.parent.right:
                    k = k.parent
                    self.left_rotate(k)
                    curRotations += 1
                k.parent.color = "black"
                k.parent.parent.color = "red"
                self.right_rotate(k.parent.parent)
                curRotations += 1

        if k == self.root:
            break
    self.rotations.append(curRotations)
    self.root.color = "black"

def count_leaves(self, node):

    if node == self.TNULL:
```

```
            return 0

        if node.left == self.TNULL and node.right == self.TNULL:
            return 1

        return self.count_leaves(node.left) + self.count_leaves(node.right)

    def get_number_of_leaves(self):
        return self.count_leaves(self.root)

    def compute_height(self, node):
        if node == self.TNULL:
            return 0
        else:
            left_height = self.compute_height(node.left)
            right_height = self.compute_height(node.right)
            return max(left_height, right_height) + 1

    def get_tree_height(self):
        return self.compute_height(self.root)

    def reset_stats(self):
        self.steps = []
        self.rotations = []

    def printStatistics(self):
        print("Red Black Tree")
        print("Steps:")
        print("minimum: ", min(self.steps))
        print("maximum: ", max(self.steps))
        print("mean", sum(self.steps) / len(self.steps))
        print("standard deviation: ", statistics.stdev(self.steps))
        print("median: ", self.steps[len(self.steps) // 2])

        print("Rotations:")
        print("minimum: ", min(self.rotations))
        print("maximum: ", max(self.rotations))
        print("mean", sum(self.rotations) / len(self.rotations))
        print("standard deviation: ", statistics.stdev(self.rotations))
        print("median: ", self.rotations[len(self.rotations) // 2])

        print("Height of the tree: ", self.get_tree_height())
```

```
        print("number of leaves: ", self.get_number_of_leaves())
        print("")
```

```
import random
import statistics

class Node:
    def __init__(self, level, key, value):
        self.key = key
        self.value = value
        self.forward = [None] * (level + 1)

class SkipList:
    def __init__(self, max_lvl, P):
        self.MAX_LVL = max_lvl
        self.P = P
        self.header = self.create_node(self.MAX_LVL, None, None)
        self.level = 0
        self.steps = []  # List to track the number of steps to insertion point
        self.promotions = []  # List to track promotions

    def create_node(self, lvl, key, value):
        return Node(lvl, key, value)

    def random_level(self):
        lvl = 0
        while random.random() < self.P and lvl < self.MAX_LVL:
            lvl += 1
        return lvl

    def insert(self, key, value):
        update = [None] * (self.MAX_LVL + 1)
        current = self.header
        step_count = 0  # Initialize step count for this insertion

        for i in range(self.level, -1, -1):
            while current.forward[i] and current.forward[i].key < key:
                current = current.forward[i]
                step_count += 1  # Increment step count for each node traversed
            update[i] = current

        current = current.forward[0]
```

```
        if current is None or current.key != key:
            rlevel = self.random_level()
            if rlevel > self.level:
                for i in range(self.level + 1, rlevel + 1):
                    update[i] = self.header
                self.level = rlevel

            n = self.create_node(rlevel, key, value)
            promotions_this_time = rlevel + 1  # Record the number of levels the node
is inserted into

            for i in range(rlevel + 1):
                n.forward[i] = update[i].forward[i]
                update[i].forward[i] = n

            self.steps.append(step_count)  # Record steps for this insertion
            self.promotions.append(promotions_this_time)  # Record promotions for this
insertion

    def reset_stats(self):
        self.steps = []
        self.promotions = []

    def printStatistics(self):
        print("Skip List Statistics")
        print("Steps to Insertion Point:")
        print("Minimum: ", min(self.steps))
        print("Maximum: ", max(self.steps))
        print("Mean: ", sum(self.steps) / len(self.steps))
        print("Standard Deviation: ", statistics.stdev(self.steps))
        print("Median: ", statistics.median(self.steps))
        print("Promotions:")
        print("Minimum: ", min(self.promotions))
        print("Maximum: ", max(self.promotions))
        print("Mean: ", sum(self.promotions) / len(self.promotions))
        print("Standard Deviation: ", statistics.stdev(self.promotions))
        print("Median: ", statistics.median(self.promotions))
        print("Current Max Level: ", self.level)
```