

# Data Structures and Algorithms 2

## Course Project 2024

ICS2210

Malcolm Borg 3604H

# Contents

<b>Knuth shuffle</b>	<b>4</b>
<b>AVL tree</b>	<b>4</b>
Class Node	4
Class AVLTree	4
<b>Red-Black tree</b>	<b>6</b>
Class Node	6
Class RedBlackTree	6
<b>Skip List tree</b>	<b>8</b>
Class Node	8
Class SkipList	9
<b>Statistics</b>	<b>11</b>
Steps	11
Rotations	11
Height	11
Number of Leaves	11
Max Level	11
<b>Evaluation Of Statistics</b>	<b>12</b>
<b>Overall Evaluation</b>	<b>12</b>

# 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.