ICS2210 - Course Project 2024

Data Structures & Algorithms 2

Name: **Andrea Filiberto Lucas**
Course: ICS2207 - Artificial Intelligence (AI)
ID No: 0279704L

Statement of completion

Item	Completed (Yes/No/Partial)
Created first array of integers	Yes
Knuth shuffle	Yes
Inserted in AVL tree	Yes
AVL tree insertion statistics	Yes
Inserted in Red-Black tree	Yes
Red-Black tree insertion statistics	Yes
Inserted in Skip List	Yes
Skip List insertion statistics	Yes
Discussion comparing data structures	Yes

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own work, of work of another person, whether or not such work has been published" (Regulations Governing Conduct at Examinations, 1997, Regulation 1 (viii), University of Malta).

I ~~/ We*~~, the undersigned, declare that the [**assignment** ~~/ Assigned Practical Task report / Final Year Project report~~] submitted is **my** ~~/ our*~~ work, except where acknowledged and referenced.

I ~~/ We*~~ understand that the penalties for making a false declaration may include, but are not limited to, loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

* Delete as appropriate.

(N.B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Andrea Filiberto Lucas
Student Name


Signature

ICS2210
Course Code

ICS2210 - Course Project 2024
Title of work submitted

06/05/2024
Date

ics2210-course-project-2024

May 6, 2024

1 Practical Component

1.1 Importing Necessary Libraries

The below code snippet starts by `importing the libraries` required for random number generation, statistical analysis, system operations, and timing functions (used for `ALL data structures`).

[1]:
```
import random
import statistics
import sys
import time
import math
```

1.2 Knuth Shuffle Function

The `knuth_shuffle function` employs the Knuth shuffle algorithm, which is designed to randomly rearrange the elements of an array. This algorithm is essential for shuffling input arrays prior to the construction of data structure, thus increasing robustness.

[2]:
```
def knuth_shuffle(array):
    """Implementation of Knuth shuffle algorithm."""
    n = len(array)
    for i in range(n - 1, 0, -1):
        j = random.randint(0, i)
        array[i], array[j] = array[j], array[i]
    return array
```

1.3 Other Aesthetic Functions

1.3.1 Clear Terminal Function

The function `clear_terminal` clears the terminal screen. It creates a clean and organised display environment when the notebook is executed. This function improves the user experience by removing clutter and ensuring clear output.

[3]:
```
def clear_terminal():
    """Clear the terminal."""
    print("\033[2J\033[H")
```

1.3.2 Print Box Function

The `print_box` function formatted and printed content within a box. This function improves readability and comprehension of statistical results by presenting them in an organised manner.

[4]:
```
def print_box(title, content):
    """Print content inside a box."""
    print(f"+{'-' * (len(title) + 2)}+")
    print(f"| {title} |")
    print(f"+{'-' * (len(title) + 2)}+")
    print(content)
    print(f"+{'-' * (len(title) + 2)}+")
```

1.4 AVL Trees

An AVL tree is a self-balancing binary search tree in which the heights of each node's two child subtrees differ by no more than one, ensuring logarithmic time complexity for insertion, deletion, and search operations.

1.4.1 AVL Node Class

The `AVLNode class` is a fundamental component of the AVL tree that represents individual nodes. Each node in this structure contains essential elements such as a key value, pointers to left and right children, and a height attribute. This class is critical in the construction of the AVL tree because it serves as the fundamental building block on which the entire tree structure is based.

[5]:
```
class AVLNode:
    """Node class for AVL Tree."""
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1
```

1.4.2 AVL Tree Class

At its core, the `AVLTree class` provides a framework for implementing and analysing AVL trees, which are a type of self-balanced binary search tree. This class provides a set of functionalities required for AVL tree management, including methods for adding nodes to the tree, performing rotation operations to maintain balance, calculating the tree's height, and evaluating balance factors at each node. Furthermore, the class allows you to extract detailed statistics about the tree, including important metrics like minimum, maximum, mean, standard deviation, and median values for insertion steps and rotation counts. By encapsulating these features, the AVLTree class enables the seamless construction and in-depth analysis of AVL trees, which is critical for efficient data organisation and retrieval tasks.

[6]:
```
class AVLTree:
    """AVL Tree implementation."""
    def __init__(self):
```

```
        self.root = None
        self.steps = []  # List to track steps taken during insertion
        self.rotations = []  # List to track rotations during insertion
        self.height = 0  # Initialize height attribute
        self.leaves = 0  # Initialize leaves attribute

    def calculate_height(self, node):
        """Calculate height of a node."""
        return node.height if node else 0

    def rightRotate(self, y):
        """Right rotation operation."""
        x = y.left
        T2 = x.right
        x.right = y
        y.left = T2
        y.height = max(self.calculate_height(y.left), self.calculate_height(y.
↪right)) + 1
        x.height = max(self.calculate_height(x.left), self.calculate_height(x.
↪right)) + 1
        return x

    def leftRotate(self, x):
        """Left rotation operation."""
        y = x.right
        T2 = y.left
        y.left = x
        x.right = T2
        x.height = max(self.calculate_height(x.left), self.calculate_height(x.
↪right)) + 1
        y.height = max(self.calculate_height(y.left), self.calculate_height(y.
↪right)) + 1
        return y

    def getBalance(self, node):
        """Get balance factor of a node."""
        return self.calculate_height(node.left) - self.calculate_height(node.
↪right)

    def insert(self, node, key):
        """Recursive function to insert a key into the tree."""
        if not node:
            return AVLNode(key)
        elif key < node.key:
            self.steps[-1] += 1
            node.left = self.insert(node.left, key)
        else:
```

```
            self.steps[-1] += 1
            node.right = self.insert(node.right, key)
        node.height = 1 + max(self.calculate_height(node.left), self.
↪calculate_height(node.right))
        balance = self.getBalance(node)
        if balance > 1 and key < node.left.key:
            self.rotations[-1] += 1
            return self.rightRotate(node)
        if balance < -1 and key > node.right.key:
            self.rotations[-1] += 1
            return self.leftRotate(node)
        if balance > 1 and key > node.left.key:
            node.left = self.leftRotate(node.left)
            self.rotations[-1] += 1
            return self.rightRotate(node)
        if balance < -1 and key < node.right.key:
            node.right = self.rightRotate(node.right)
            self.rotations[-1] += 1
            return self.leftRotate(node)
        return node

    def insertKey(self, key):
        """Function to insert a key into the tree."""
        self.steps.append(0)
        self.rotations.append(0)
        self.root = self.insert(self.root, key)
        self.calculate_height_and_leaves()

    def calculate_height_and_leaves(self):
        """Calculate height and number of leaves in the tree."""
        if self.root is None:
            self.height = 0
            self.leaves = 0
        else:
            self.height = self.calculate_tree_height(self.root)
            self.leaves = self.calculate_tree_leaves(self.root)

    def calculate_tree_height(self, node):
        """Recursive function to calculate height of the tree."""
        if node is None:
            return 0
        left_height = self.calculate_tree_height(node.left)
        right_height = self.calculate_tree_height(node.right)
        return max(left_height, right_height) + 1

    def calculate_tree_leaves(self, node):
        """Recursive function to calculate number of leaves in the tree."""
```

```
        if node is None:
            return 0
        if node.left is None and node.right is None:
            return 1
        return self.calculate_tree_leaves(node.left) + self.
↪calculate_tree_leaves(node.right)

    def get_statistics(self):
        """Compute statistics of the tree."""
        if not self.steps or not self.rotations:
            return None
        min_steps = min(self.steps)
        max_steps = max(self.steps)
        mean_steps = round(statistics.mean(self.steps), 3)
        std_steps = round(statistics.stdev(self.steps), 3) if len(self.steps) >␣
↪1 else 0
        median_steps = round(statistics.median(self.steps), 3)
        min_rotations = min(self.rotations)
        max_rotations = max(self.rotations)
        mean_rotations = round(statistics.mean(self.rotations), 3)
        std_rotations = round(statistics.stdev(self.rotations), 3) if len(self.
↪rotations) > 1 else 0
        median_rotations = round(statistics.median(self.rotations), 3)
        height = self.height
        leaves = self.leaves
        return {
            "Steps": {
                "Min": min_steps,
                "Max": max_steps,
                "Mean": mean_steps,
                "Standard Deviation (std)": std_steps,
                "Median": median_steps
            },
            "Rotations": {
                "Min": min_rotations,
                "Max": max_rotations,
                "Mean": mean_rotations,
                "Standard Deviation (std)": std_rotations,
                "Median": median_rotations
            },
            "Height": height,
            "Leaves": leaves
        }
```

1.4.3 AVL Tree Construction Function

The `construct_avl_tree` function aids in the generation of AVL trees from input arrays. It iteratively traverses the array, inserting each element into the AVL tree. Throughout the insertion process, the function meticulously monitors and records the number of steps and rotations completed, which are critical metrics for future statistical analysis.

[7]:
```
def construct_avl_tree(array, part_name):
    """Construct the AVL tree from an array."""
    print(f"\033[1;32mConstructing AVL Tree ({part_name}):\033[0m")
    for i, num in enumerate(array):
        avl_tree.insertKey(num)
        progress = (i + 1) / len(array)
        sys.stdout.write("\r\033[1;32m[{:<50}] {:.2f}%\033[0m".format("=" *␣
 ↪int(progress * 50), progress * 100))
        sys.stdout.flush()
    print("\n")
```

1.4.4 Main Execution Block - AVL Trees

The `main script creates and analyses AVL trees` in two phases: First, a sorted array is shuffled and used to construct an AVL tree labelled `"Part 1"`, followed by the generation of a randomised array and the use of another AVL tree labelled `"Part 2"`. After each phase, the terminal is cleared to ensure clarity. Once both trees have been built, a `confirmation message appears`, followed by a `presentation of statistical analysis` results such as step counts, rotation counts, tree height, and the number of leaves, which provide insights into the AVL tree's structural properties and performance.

[8]:
```
if __name__ == "__main__":
    clear_terminal()
    array1 = list(range(1, 5001))
    array1 = knuth_shuffle(array1)

    avl_tree = AVLTree()
    construct_avl_tree(array1, "Part 1")
    time.sleep(1)

    clear_terminal()

    array2 = [random.randint(0, 100000) for _ in range(1000)]
    avl_tree.steps = []
    avl_tree.rotations = []
    construct_avl_tree(array2, "Part 2")
    time.sleep(1)

    clear_terminal()

    print("\033[1;32mAVL Tree Construction Complete\033[0m\n")
```

```
    time.sleep(1)

    statistics = avl_tree.get_statistics()
    if statistics:
        print_box("Steps", "\n".join([f"{key}: {value}" for key, value in␣
↪statistics["Steps"].items()]))
        print("\n")
        time.sleep(1)
        print_box("Rotations", "\n".join([f"{key}: {value}" for key, value in␣
↪statistics["Rotations"].items()]))
        print("\n")
        time.sleep(1)
        print(f"\033[1;32mHeight: {statistics['Height']}\033[0m")
        print(f"\033[1;32mLeaves: {statistics['Leaves']}\033[0m")
```

```
Constructing AVL Tree (Part 1):
[==================================================] 100.00%


Constructing AVL Tree (Part 2):
[==================================================] 100.00%


AVL Tree Construction Complete

+-------+
| Steps |
+-------+
Min: 10
Max: 15
Mean: 13.412
Standard Deviation (std): 0.901
Median: 13.0
+-------+


+-----------+
| Rotations |
+-----------+
Min: 0
Max: 2
Mean: 0.466
Standard Deviation (std): 0.501
Median: 0.0
+-----------+
```

```
Height: 15
Leaves: 2569
```

1.5 Red-Black Trees

A red-black tree is another self-balancing binary search tree in which each node is assigned a colour (red or black) based on predefined rules. These rules keep the tree balanced, allowing for efficient operations such as insertion, deletion, and search with logarithmic time complexity.

1.5.1 Red-Black Node Class

The `RBNode class` defines the structure of a Red-Black Tree node, which includes attributes such as a key value, left and right child pointers, a parent pointer, and a colour indicator initialised with "RED". This class serves as the foundation for creating Red-Black Trees, providing the elements required to maintain the tree's balance and adhere to the Red-Black Tree characteristics.

[9]:
```python
class RBNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.parent = None
        self.color = "RED"
```

1.5.2 Red-Black Tree Class

The `RedBlackTree class` provides the foundation for implementing and analysing Red-Black Trees, another type of self-balancing binary search tree. This class contains the essential functionalities required for managing Red-Black Trees, such as methods for inserting nodes into the tree and performing rotation operations to maintain balance. It also makes it easier to calculate the tree's height and find balance factors at each node. Furthermore, the class allows for the extraction of detailed statistics about the tree, including important metrics like minimum, maximum, mean, standard deviation, and median values for insertion steps and rotation counts.

[10]:
```python
class RedBlackTree:
    def __init__(self):
        self.root = None
        self.steps = []
        self.rotations = []
        self.height = 0
        self.leaves = 0

    def left_rotate(self, x):
        y = x.right
        x.right = y.left
        if y.left is not None:
            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.rotations[-1] += 1


    def right_rotate(self, y):
        x = y.left
        y.left = x.right
        if x.right is not None:
            x.right.parent = y
        x.parent = y.parent
        if y.parent is None:
            self.root = x
        elif y == y.parent.left:
            y.parent.left = x
        else:
            y.parent.right = x
        x.right = y
        y.parent = x
        self.rotations[-1] += 1


    def insert_fixup(self, z):
        while z.parent is not None and z.parent.color == "RED":
            if z.parent == z.parent.parent.left:
                y = z.parent.parent.right
                if y is not None and y.color == "RED":
                    z.parent.color = "BLACK"
                    y.color = "BLACK"
                    z.parent.parent.color = "RED"
                    z = z.parent.parent
                else:
                    if z == z.parent.right:
                        z = z.parent
                        self.left_rotate(z)
                    z.parent.color = "BLACK"
                    z.parent.parent.color = "RED"
                    self.right_rotate(z.parent.parent)
            else:
                y = z.parent.parent.left
```

```
            if y is not None and y.color == "RED":
                z.parent.color = "BLACK"
                y.color = "BLACK"
                z.parent.parent.color = "RED"
                z = z.parent.parent
            else:
                if z == z.parent.left:
                    z = z.parent
                    self.right_rotate(z)
                z.parent.color = "BLACK"
                z.parent.parent.color = "RED"
                self.left_rotate(z.parent.parent)
        self.root.color = "BLACK"

    def insert_key(self, key):
        self.steps.append(0)
        self.rotations.append(0)
        new_node = RBNode(key)
        y = None
        x = self.root
        while x is not None:
            y = x
            self.steps[-1] += 1
            if new_node.key < x.key:
                x = x.left
            else:
                x = x.right
        new_node.parent = y
        if y is None:
            self.root = new_node
        elif new_node.key < y.key:
            y.left = new_node
        else:
            y.right = new_node
        self.insert_fixup(new_node)

    def tree_height(self, node):
        if node is None:
            return 0
        left_height = self.tree_height(node.left)
        right_height = self.tree_height(node.right)
        return max(left_height, right_height) + 1

    def count_leaves(self, node):
        if node is None:
            return 0
        if node.left is None and node.right is None:
```

```
            return 1
        return self.count_leaves(node.left) + self.count_leaves(node.right)
```

1.5.3 Statistics - Red-Black Trees

The `print_statistics` function formats and prints key metrics such as insertion steps, rotations, tree height, and leaf count in structured boxes, improving readability and comprehension. Furthermore, the `calculate_statistics` function computes the following statistics: minimum, maximum, mean, standard deviation, and median values for both insertion steps and rotations.

[11]:
```
import statistics # Re-written (error fix)

def print_statistics(statistics):
    print_box("Steps", "\n".join([f"{key}: {value}" for key, value in␣
 ↪statistics["Steps"].items()]))
    print("\n")
    time.sleep(1)
    print_box("Rotations", "\n".join([f"{key}: {value}" for key, value in␣
 ↪statistics["Rotations"].items()]))
    print("\n")
    time.sleep(1)
    print(f"\033[1;32mHeight: {statistics['Height']}\033[0m")
    print(f"\033[1;32mLeaves: {statistics['Leaves']}\033[0m")

def calculate_statistics(tree):
    min_steps = min(tree.steps)
    max_steps = max(tree.steps)
    mean_steps = round(statistics.mean(tree.steps), 3)
    std_steps = round(statistics.stdev(tree.steps), 3) if len(tree.steps) > 1␣
 ↪else 0
    median_steps = round(statistics.median(tree.steps), 3)

    min_rotations = min(tree.rotations)
    max_rotations = max(tree.rotations)
    mean_rotations = round(statistics.mean(tree.rotations), 3)
    std_rotations = round(statistics.stdev(tree.rotations), 3) if len(tree.
 ↪rotations) > 1 else 0
    median_rotations = round(statistics.median(tree.rotations), 3)

    height = tree.tree_height(tree.root)
    leaves = tree.count_leaves(tree.root)

    return {
        "Steps": {
            "Min": min_steps,
            "Max": max_steps,
            "Mean": mean_steps,
```

```
            "Standard Deviation (std)": std_steps,
            "Median": median_steps
        },
        "Rotations": {
            "Min": min_rotations,
            "Max": max_rotations,
            "Mean": mean_rotations,
            "Standard Deviation (std)": std_rotations,
            "Median": median_rotations
        },
        "Height": height,
        "Leaves": leaves
    }
```

1.5.4 Red-Black Tree Construction Function

The `construct_red_black_tree` function plays an important role in constructing Red-Black Trees from input arrays, allowing for easier creation and progress monitoring. When invoked, it starts the construction process and displays a progress bar indicating completion status. As each element from the input array is inserted into the tree, the progress bar updates in real time, providing feedback on the insertion process. This function allows for the systematic construction of Red-Black Trees, which improves their reliability and efficiency in data organisation and retrieval tasks.

[12]:
```
def construct_red_black_tree(array, part_name):
    print(f"\033[1;32mConstructing Red-Black Tree ({part_name}):\033[0m")
    rb_tree = RedBlackTree()
    for i, num in enumerate(array):
        rb_tree.insert_key(num)
        progress = (i + 1) / len(array)
        sys.stdout.write("\r\033[1;32m[{:<50}] {:.2f}%\033[0m".format("=" *␣
↪int(progress * 50), progress * 100))
        sys.stdout.flush()
        time.sleep(0.0001) # aesthetic delay (clarity)
    print("\n")
    return rb_tree
```

1.5.5 Main Execution Block - Red-Black Trees

The `main function for the Red-Black Trees` begins with clearing the terminal for clarity, followed by preparing and shuffling a sorted array of integers from 1 to 5000, resulting in the creation of a Red-Black Tree labelled `"Part 1"`. After a brief pause, the terminal is then cleared to proceed to the next phase, in which a randomised array of 1000 integers between 0 and 100000 is generated. This array is used to create another Red-Black Tree labelled `"Part 2"`. A `confirmation message` is displayed after both phases of tree construction are completed, indicating that the construction was successful. The constructed trees are then `statistically analysed`, with key metrics such as insertion steps, rotations, tree height, and the number of leaves presented in a structured format to provide insights into their structural properties and performance characteris-

tics.

[13]:
```python
if __name__ == "__main__":
    clear_terminal()
    array1 = list(range(1, 5001))
    array1 = knuth_shuffle(array1)

    rb_tree = construct_red_black_tree(array1, "Part 1")
    time.sleep(1)

    clear_terminal()

    array2 = [random.randint(0, 100000) for _ in range(1000)]
    rb_tree.steps = []
    rb_tree.rotations = []

    rb_tree = construct_red_black_tree(array2, "Part 2")
    time.sleep(1)

    clear_terminal()

    print("\033[1;32mRed-Black Tree Construction Complete\033[0m\n")

    statistics = calculate_statistics(rb_tree)
    print_statistics(statistics)
```

```
Constructing Red-Black Tree (Part 1):
[==================================================] 100.00%


Constructing Red-Black Tree (Part 2):
[==================================================] 100.00%


Red-Black Tree Construction Complete

+-------+
| Steps |
+-------+
Min: 0
Max: 12
Mean: 8.755
Standard Deviation (std): 1.655
Median: 9.0
+-------+
```

```
+-----------+
| Rotations |
+-----------+
Min: 0
Max: 2
Mean: 0.591
Standard Deviation (std): 0.803
Median: 0.0
+-----------+


Height: 12
Leaves: 426
```

1.6 Skip Lists

A skip list is a probabilistic data structure that looks like a hierarchical linked list with multiple levels and allows for fast search, insertion, and deletion operations. Each level represents a different "skip" and allows for faster traversal, providing an alternative to balanced trees with comparable performance characteristics.

1.6.1 Skip List Node Class

The `SkipListNode class` is the basic building block for creating Skip Lists, with attributes required for their functionality. Each node in this structure contains a key value and an array of pointers that represent forward references at different levels. The level of the node determines the number of forward references it has.

[14]:
```python
class SkipListNode:
    """Node class for Skip List."""
    def __init__(self, key, level):
        self.key = key
        self.forward = [None] * (level + 1)
```

1.6.2 Skip List Class

The `SkipList class` implements skip lists, a data structure known for its efficient search and retrieval operations. With its core functionalities centred on insertion and statistical analysis, the class provides a solid framework for managing skip lists. It ensures data integrity and balance by tracking insertion steps and promotions for insightful statistical analysis. The `get_statistics` method improves the class's utility by calculating key metrics like minimum, maximum, mean, and median values for insertion steps and promotions, as well as providing an overview of the skip list's level distribution.

[15]:
```python
import statistics # Re-written (error fix)

class SkipList:
    """Skip List implementation."""
    def __init__(self, max_levels, p):
```

```
        self.max_levels = max_levels
        self.p = p
        self.header = self.create_node(float('-inf'), max_levels)
        self.level = 0
        self.steps = []  # List to track steps taken during insertion
        self.promotions = []  # List to track promotions during insertion

    def create_node(self, key, level):
        """Create a new node with the given key and level."""
        return SkipListNode(key, level)

    def random_level(self):
        """Generate a random level for a new node."""
        level = 0
        while random.random() < self.p and level < self.max_levels:
            level += 1
        return level

    def insert(self, key):
        """Insert a key into the skip list."""
        update = [None] * (self.max_levels + 1)
        x = self.header
        for i in range(self.level, -1, -1):
            while x.forward[i] and x.forward[i].key < key:
                x = x.forward[i]
            update[i] = x
        x = x.forward[0]
        if not x or x.key != key:
            new_level = self.random_level()
            if new_level > self.level:
                for i in range(self.level + 1, new_level + 1):
                    update[i] = self.header
                self.level = new_level
            x = self.create_node(key, new_level)
            for i in range(new_level + 1):
                x.forward[i] = update[i].forward[i]
                update[i].forward[i] = x
            self.steps.append(sum(1 for _ in update))
            self.promotions.append(new_level)
        else:
            self.steps.append(sum(1 for _ in update))

    def get_statistics(self):
        """Compute statistics of the skip list."""
        if not self.steps or not self.promotions:
            return None
        min_steps = min(self.steps)
```

```
        max_steps = max(self.steps)
        mean_steps = round(statistics.mean(self.steps), 3)
        std_steps = round(statistics.stdev(self.steps), 3) if len(self.steps) >␣
↪1 else 0
        median_steps = round(statistics.median(self.steps), 3)

        min_promotions = min(self.promotions)
        max_promotions = max(self.promotions)
        mean_promotions = round(statistics.mean(self.promotions), 3)
        std_promotions = round(statistics.stdev(self.promotions), 3) if␣
↪len(self.promotions) > 1 else 0
        median_promotions = round(statistics.median(self.promotions), 3)

        levels = self.level + 1

        return {
            "Steps": {
                "Min": min_steps,
                "Max": max_steps,
                "Mean": mean_steps,
                "Standard Deviation (std)": std_steps,
                "Median": median_steps
            },
            "Promotions": {
                "Min": min_promotions,
                "Max": max_promotions,
                "Mean": mean_promotions,
                "Standard Deviation (std)": std_promotions,
                "Median": median_promotions
            },
            "Levels": levels
        }
```

1.6.3 Skip List Construction Function

The `construct_skip_list` function allows the dynamic creation of skip lists from input arrays by leveraging critical parameters such as maximum levels and a probability parameter. The skip list's structure is automatically adjusted based on the array size, ensuring optimal performance and balance. Throughout the construction process, a progress bar provides real-time feedback on insertion progress. As each element is added to the skip list, the function systematically tracks and updates the skip list's structure, providing users with an efficient and effective tool for managing and analysing data within skip list structures.

[16]:
```
def construct_skip_list(array, part_name):
    """Construct the skip list from an array."""
    max_levels = int(math.log(len(array), 2))  # Calculate maximum levels based␣
↪on array size
```

```
    p = 0.5  # Probability parameter for skip list
    skip_list = SkipList(max_levels, p)
    print(f"\033[1;32mConstructing Skip List ({part_name}):\033[0m")
    for i, num in enumerate(array):
        skip_list.insert(num)
        progress = (i + 1) / len(array)
        sys.stdout.write("\r\033[1;32m[{:<50}] {:.2f}%\033[0m".format("=" *␣
↪int(progress * 50), progress * 100))
        sys.stdout.flush()
        time.sleep(0.0001) # aesthetic delay (clarity)
    print("\n")
    return skip_list
```

1.6.4 Main Execution Block - Skip List

The `main script` (for skip list) divides the construction and analysis of skip lists into two phases, each with the goal of evaluating the structure's performance under various input distribution scenarios. Initially, it shuffles a sorted array with the Knuth shuffle algorithm, creating a skip list labelled `"Part 1."` Then, a random array is generated to create another skip list labelled `"Part 2."` Following completion, both skip lists are `statistically analysed` to present key metrics such as step counts, promotion counts, and the number of levels. This script provides useful insights into skip list behaviour under various input scenarios, allowing us to better understand its efficiency and effectiveness in data organisation and retrieval tasks.

[17]:
```
if __name__ == "__main__":
    clear_terminal()
    array1 = list(range(1, 5001))
    array1 = knuth_shuffle(array1)

    skip_list = construct_skip_list(array1, "Part 1")
    time.sleep(1)

    clear_terminal()

    array2 = [random.randint(0, 100000) for _ in range(1000)]
    skip_list.steps = []
    skip_list.promotions = []

    skip_list = construct_skip_list(array2, "Part 2")
    time.sleep(1)

    clear_terminal()

    print("\033[1;32mSkip List Construction Complete\033[0m\n")

    statistics = skip_list.get_statistics()
    if statistics:
```

```
        print_box("Steps", "\n".join([f"{key}: {value}" for key, value in␣
↪statistics["Steps"].items()]))
        print("\n")
        time.sleep(1)
        print_box("Promotions", "\n".join([f"{key}: {value}" for key, value in␣
↪statistics["Promotions"].items()]))
        print("\n")
        time.sleep(1)
        print(f"\033[1;32mLevels: {statistics['Levels']}\033[0m")
```

```
Constructing Skip List (Part 1):
[==================================================] 100.00%


Constructing Skip List (Part 2):
[==================================================] 100.00%


Skip List Construction Complete

+-------+
| Steps |
+-------+
Min: 10
Max: 10
Mean: 10
Standard Deviation (std): 0.0
Median: 10.0
+-------+


+------------+
| Promotions |
+------------+
Min: 0
Max: 9
Mean: 1.084
Standard Deviation (std): 1.446
Median: 1
+------------+


Levels: 10
```

2 Evaluation

When assessing the performance and appropriateness of AVL Trees, Red-Black Trees, and Skip Lists, it is critical to take into account their distinct attributes and the specific needs of the use case. The statistics gathered from the data structures provide useful information about their efficiency, structural properties, and operational behavior.

Starting with AVL Trees, the data show that they have a balanced structure, resulting in relatively consistent insertion and retrieval times. The average number of steps required for insertion (13.412) and maximum height (15) indicate consistent performance with little variation in traversal depth. Furthermore, the low average number of rotations (0.466) indicates that AVL Trees effectively self-balance during insertion operations, resulting in minimal structural adjustments. However, AVL Trees have slightly higher average numbers of steps and rotations than Red-Black Trees and Skip Lists, which may affect performance in scenarios with strict latency requirements.

Similarly, Red-Black Trees have comparable performance to AVL Trees, having comparable average steps, rotations, and maximum height. This type of data structure does not require complex balancing rules, which simplifies their implementation and deems them appropriate for situations in which maintenance simplicity is essential. Despite slightly higher average steps and rotations than Skip Lists, Red-Black Trees maintain a balanced structure, allowing for efficient operations across a wide range of data sizes and distributions.

On the other hand Skip Lists, have a distinct structure that is designed for efficient search operations, especially in scenarios with high insertion (and deletion) rates. The consistent number of steps required for insertion (10 across different datasets) demonstrates Skip Lists' predictable performance. Furthermore, the low standard deviation in steps and promotions indicates stable operational behavior, making Skip Lists ideal for real-time applications that require predictable latency. However, Skip Lists have a higher average number of promotions (1.084), which may affect memory overhead and performance in memory-constrained environments.

Considering real-world scenarios, the choice between AVL Trees, Red-Black Trees, and Skip Lists is determined by the application's specific requirements and trade-offs. For applications that require balanced search and retrieval operations, such as database systems or balanced binary search trees, AVL Trees and Red-Black Trees provide optimal solutions with predictable performance characteristics. Red-Black Trees, with their simpler balancing rules, may be preferred for ease of implementation and maintenance in such cases.

Skip Lists, on the other hand, are an appealing option in situations involving frequent insertion and deletion operations, such as caching systems. Their efficient search and update operations, combined with consistent performance, make them ideal for high-throughput environments. However, they may be less efficient in scenarios that require strict memory constraints or when memory overhead is an issue due to their higher average number of promotions.

In conclusion, the various performance metrics demonstrated by the data structures highlight their respective strengths and limitations. Nonetheless, the most suitable choice is determined by the application's specific requirements. With a thorough understanding of each data structures and an in-depth understanding of the application's demands, developers can make well-informed choices to ensure maximum efficiency and efficacy in their real-world scenarios.