CS6.302 - Software System Development

Lab Activity: The Performance Cost of Unbalanced Trees

Topic: BST vs. AVL Tree Performance Analysis

Scenario: The Network Anomaly Detector

You are a software engineer at a cybersecurity firm. Your team is building a high-speed anomaly detector that logs millions of network events. Each event has a unique, integer-based "event ID". You need to store these IDs in a way that allows for two critical operations:

1. **Fast Insertion**: New events arrive constantly and must be logged.
2. **Ultra-Fast Lookup**: A separate process must constantly query the system to see if a specific event ID has already been logged.

A junior developer implemented a standard Binary Search Tree (BST). In testing with random data, performance was great (O(log n)). However, when the system went live, it monitoring a specific subnet that was compromised. This subnet sent a massive flood of packets with sequential, increasing event IDs (e.g., 1001, 1002, 1003...).

The system's performance immediately collapsed. Your job is to demonstrate **why** this happened and implement a robust solution (an AVL Tree) that can handle this "worst-case" data.

Your Task: An Automated Performance Analysis

You will implement both a standard **BST** and a self-balancing **AVL Tree**. You will feed both trees a "killer" dataset of 50,000 sorted integers read from a file. Finally, you will time their search performance and generate a report.

This is an auto-graded assignment. Your final script must produce a file named `analysis.txt` in the exact format specified.

Step 1: Define the 'Node' Class

Create a file named `avl_vs_bst.py`. Use this provided `Node` class.

```python
from dataclasses import dataclass
from typing import Optional

@dataclass
class Node:
    key: int
    height: int = 1
    left: Optional['Node'] = None
    right: Optional['Node'] = None
```

Step 2: Implement the 'BST' Class

In the same file, implement a standard Binary Search Tree. You must fill in the logic for all the methods below. All methods must be recursive (where appropriate).

```python
class BST:
    def __init__(self):
        self.root: Optional[Node] = None

    def insert(self, key: int):
        self.root = self._insert(self.root, key)

    def _insert(self, root: Optional[Node], key: int) -> Node:
        # TODO: Implement recursive BST insertion

    def search(self, key: int) -> bool:
        return self._search(self.root, key)

    def _search(self, root: Optional[Node], key: int) -> bool:
        # TODO: Implement recursive BST search

    def get_height(self) -> int:
        return self._get_height(self.root)

    def _get_height(self, root: Optional[Node]) -> int:
        # TODO: Implement recursive height calculation

    def is_balanced(self) -> bool:
        return self._is_balanced(self.root)

    def _is_balanced(self, root: Optional[Node]) -> bool:
        # TODO: Implement recursive balance check.
```

Step 3: Implement the 'AVLTree' Class

Now, implement the AVL tree. It will be very similar to the BST, but with crucial differences in the `insert` method and new helper functions.

```python
class AVLTree:
    def __init__(self):
        self.root: Optional[Node] = None

    # TODO: Implement _get_node_height (you can copy from BST, but it's
    # simpler: return node.height if node else 0)
    def _get_node_height(self, node: Optional[Node]) -> int:

    # TODO: Implement get_balance
    def _get_balance(self, node: Optional[Node]) -> int:

    # TODO: Implement right_rotate
    def _right_rotate(self, y: Node) -> Node:

    # TODO: Implement left_rotate
    def _left_rotate(self, x: Node) -> Node:

    def insert(self, key: int):
        self.root = self._insert(self.root, key)

    def _insert(self, root: Optional[Node], key: int) -> Node:
        # TODO: Implement recursive AVL insertion

    def search(self, key: int) -> bool:
        return self._search(self.root, key)
```

```python

    # TODO: Implement _search (can copy from BST)
    def _search(self, root: Optional[Node], key: int) -> bool:
        # ... (Same as BST's search)

    def get_height(self) -> int:
        return self._get_node_height(self.root)

    def is_balanced(self) -> bool:
        return self._is_balanced(self.root)

    # TODO: Implement _is_balanced
    def _is_balanced(self, root: Optional[Node]) -> bool:
```

Step 4: The Main Analysis Block

This is the most important part for grading. You **must** implement the main execution block to perform the analysis and write the `analysis.txt` file. You will be given `search_target.txt` and `sorted_sensor_data.txt`.

```python
import time

if __name__ == "__main__":

    data_filename = "sorted_sensor_data.txt"
    target_filename = "search_target.txt"
    output_filename = "analysis.txt"

    # --- Load Data ---
    print(f"Loading data from {data_filename}...")
    keys_to_insert = []
    try:
        with open(data_filename, 'r') as f:
            for line in f:
                keys_to_insert.append(int(line.strip()))

        with open(target_filename, 'r') as f:
            search_key = int(f.read().strip())

        print(f"Data loaded. {len(keys_to_insert)} keys.")
        print(f"Search target: {search_key}")

    except FileNotFoundError:
        print(f"Error: Data files not found.")
        exit()

    import sys
    # We must increase Python's recursion limit to allow the BST
    # to build its extremely deep, skewed tree (height = 50,000).
    # The default limit is ~1000.
    new_limit = len(keys_to_insert) + 10
    print(f"Setting recursion limit to {new_limit}...")
    sys.setrecursionlimit(new_limit)

    bst_tree = BST()
    avl_tree = AVLTree()

    # --- Time BST Insertion ---
    print("Inserting into BST...")
    bst_start_time = time.perf_counter()
    for key in keys_to_insert:
        bst_tree.insert(key)
```

```python
    bst_end_time = time.perf_counter()
    bst_insert_time = bst_end_time - bst_start_time
    print(f"BST insertion time: {bst_insert_time:.6f}s")

    # --- Time AVL Insertion ---
    print("Inserting into AVL...")
    avl_start_time = time.perf_counter()
    for key in keys_to_insert:
        avl_tree.insert(key)
    avl_end_time = time.perf_counter()
    avl_insert_time = avl_end_time - avl_start_time
    print(f"AVL insertion time: {avl_insert_time:.6f}s")

    # --- 1. Get Tree Heights ---
    bst_height = bst_tree.get_height()
    avl_height = avl_tree.get_height()

    # --- 2. Check Balance ---
    bst_balanced = bst_tree.is_balanced()
    avl_balanced = avl_tree.is_balanced()

    # --- 3. Time Search ---
    print(f"Searching for {search_key}...")

    # Time BST Search
    bst_search_start = time.perf_counter()
    bst_tree.search(search_key)
    bst_search_end = time.perf_counter()
    bst_search_time = bst_search_end - bst_search_start

    # Time AVL Search
    avl_search_start = time.perf_counter()
    avl_tree.search(search_key)
    avl_search_end = time.perf_counter()
    avl_search_time = avl_search_end - avl_search_start

    # --- 4. Calculate Ratio ---
    # Handle division by zero just in case
    search_ratio = 0.0
    if avl_search_time > 0:
        search_ratio = bst_search_time / avl_search_time

    # --- 5. Write Report ---
    print(f"Writing analysis to {output_filename}...")
    with open(output_filename, 'w') as report:
        report.write(f"BST_HEIGHT: {bst_height}\n")
        report.write(f"AVL_HEIGHT: {avl_height}\n")
        report.write(f"BST_IS_BALANCED: {bst_balanced}\n")
        report.write(f"AVL_IS_BALANCED: {avl_balanced}\n")
        report.write(f"BST_SEARCH_TIME: {bst_search_time}\n")
        report.write(f"AVL_SEARCH_TIME: {avl_search_time}\n")
        report.write(f"SEARCH_RATIO_BST_vs_AVL: {search_ratio}\n")

    print("Analysis complete.")
```

Submission

Submit your single Python script, `avl_vs_bst.py`, and the generated `analysis.txt` file.