Assignment 2

Learning objectives

1. Implement a correct, robust AVL (self-balancing binary search) tree from scratch.
2. Understand and implement single (LL, RR) and double (LR, RL) rotations.
3. Apply AVL trees to a practical problem and measure empirical performance.
4. Compare AVL behavior against an unbalanced BST (and language built-ins when available).

Overview

In this assignment you will implement an AVL tree that supports insertion, deletion, search, and traversal. You will then use your AVL implementation in a small application: a **word-frequency manager** that stores words (strings) and their counts. Finally, you will run experiments comparing AVL against a simple unbalanced BST implementation and report the results.

Requirements (mandatory)

1. **AVL Implementation**
    - Implement an AVL tree class `AVLTree` with nodes containing a key and associated value (for the word-frequency manager the key is a string and value is an integer). Your implementation must include:
        - `insert(key, value)` — if the key exists, update the value (or allow an increment operation; choose one and document it).
        - `delete(key)` — remove a key if present.
        - `search(key)` — return associated value or `null`/`None`/`optional` if not present.
        - `in_order_traversal()` — return a list/array of (key, value) pairs in ascending key order.
        - Access to `height(node)` and `balance_factor(node)` (internally) and correct rotations: LL, RR, LR, RL.
    - The tree must maintain AVL balance invariants after every insertion and deletion.
2. **Unbalanced BST Implementation**
    - Implement a plain Binary Search Tree `BST` with the same public interface as `AVLTree` (insert, delete, search, traversal) for fair comparison. You may reuse parts of code but must not include balancing logic.
3. **Application: Word-Frequency Manager**
    - Read a text file (plain `.txt`) and insert all words into both data structures (AVL and BST) in the same order.
    - Normalize words by: lowercasing and removing punctuation (define your normalization clearly).
    - For each insertion, if the word already exists, increment its frequency count.

    - Provide functionality to:
        - Query frequency of a given word.
        - Output the top-*k* frequent words (you may compute this by traversing and sorting the (word, count) list — justify your complexity in the report).
        - Persist the in-order traversal to a file `out.txt` (one `word count` per line).
4. **Performance Experiments**
    - Design and run experiments to compare AVL vs BST on the following metrics:
        - **Average time per insertion** (measured in milliseconds or microseconds). Use at least three different input sizes (e.g., 10k, 50k, 200k words — adjust to machine capability).
        - **Average tree height** after all insertions.
        - **Time for `search` queries**: pick a set of random words (present and absent) and report mean search time.
    - Run each experiment multiple times and report averaged results. Present results as a table and (optionally) simple plots.
5. **Report** (Maximum 2 pages)
    - Present results (tables/plots) and analyze them: explain why AVL performed better/worse in different metrics.
    - Include complexity analysis (worst-case/average-case) for insertion, deletion, search, and obtaining top-k.

Deliverables

1. Source code (single zip or repository) including:
    - `AVLTree` implementation.
    - `BST` implementation.
    - Application scripts (word-frequency manager, experiment runner).
    - Unit tests.
2. `report.pdf`