CptS 223 – Advanced Data Structures in C++

PA 3: Implementing an AVL-Based Map and Performance Comparison

I. Learner Objectives

At the conclusion of this assignment, students should be able to:

- Implement a **self-balancing AVL tree** as a map (`avl_map<Key, Value>`)
- Compare the performance of their implementation with STL `std::map<Key, Value>`
- Parse and process **CSV files** in C++
- Conduct **performance benchmarking** using any library of their choice

---

II. Overview & Requirements

You should **implement an AVL tree-based map** (`avl_map<Key, Value>`) and compare its performance with `std::map`. The AVL tree will store **ZIP codes** as keys and **corresponding city/state and other information** as values.

You should download the **US ZIP codes dataset** from:
📌 https://simplemaps.com/data/us-zips

You should:

1. **Parse the CSV file** and populate both maps:
   - `avl_map<int, USCity>` (Custom AVL tree-based map)
   - `std::map<int, USCity>` (STL implementation)
2. **Extract all ZIP codes into a std::list**
3. **Randomly select 1000+ ZIP codes** from the list
4. **Measure lookup performance** for both maps
5. **Compare the performance** and summarize findings

III. Implementation Details

1. Implement `avl_map<Key, Value>` (50 pts)

Create a **self-balancing AVL tree** as a map in `avl_map.h`.
Your `avl_map` template class should:

- Store **key-value pairs** like `std::map<Key, Value>`
- Support **Insertion, Deletion, and Lookup** while maintaining AVL balancing
- Provide the following member functions with signatures similar to `std::map<Key, Value>`. Additionally, implement an **inner class iterator** that allows iteration over `avl_map`, and return an iterator from `find` just like `std::map`. You can refer to the code we wrote in class on **Week 4, Lecture 2** for an example of implementing an iterator:

```
void insert(const Key& key, const Value& value);   // Inserts key-value pair
void erase(const Key& key);                        // Removes key-value pair
iterator find(const Key& key);                     // Returns an iterator
```

- You may need to implement an **AVLNode class** to store key-value pairs
- Ensure **rebalancing** occurs after insertion/deletion

---

2. Parse the CSV file (25 pts)

- Download the **US ZIP codes dataset** (see Section II for the link)
- Extract and store **ZIP codes as keys** and **all other columns as strings**

Parsing Steps:

- Open the CSV file
- Read **ZIP code (first column) as an integer**
- Store all other columns as strings in a `USCity` class
- Create a `USCity` instance for each row and populate **both maps (`avl_map` and `std::map`)**
- While parsing, you can also populate a `std::list` with just the ZIP codes. We will use this list later for testing.

---

3. Benchmark Lookup Performance (15 pts)

Steps:

1. Randomly **select 1000+ ZIP codes** from the list
2. Use a for-loop to perform lookup operations in `avl_map` for the selected ZIP codes and compute elapsed time.
3. Use a for-loop to perform lookup operations in `std::map` for the selected ZIP codes and compute elapsed time.
4. Print the elapsed time of both lookup operations for comparison.

---

4. Performance Analysis & Summary (5 pts)

In a **README file**, analyze and confirm that:

- Your implementation supports **logarithmic insertion and lookup**
- Explain how AVL tree operations maintain logarithmic performance
- Compare avl_map performance with std::map and discuss trade-offs

IV. Points Breakdown

Task	Points
Implement `avl_map<Key, Value>`	50 pts
- Insert method	10 pts
- Erase method	10 pts
- Find method (returning iterator)	10 pts
- Implementing inner class iterator	10 pts
- Successful rebalancing (left/right rotations)	10 pts
Parse and Populate Cities Data from CSV	25 pts
- Implementing `USCity` class	10 pts
- Extracting and storing ZIP codes in a `std::list`	5 pts
- Populating `avl_map` and `std::map`	10 pts
Benchmark Performance of Lookups	15 pts
Performance Analysis & Summary	5 pts
Makefile & Modularization (Modularize code into multiple header and CPP files appropriately)	5 pts
Code Cleanliness (Proper Naming Convention & Clear Comments)	5 pts
Total	**100 pts**

V. Submission Guidelines

- **Submit all `.cpp` and `.h` files as a ZIP**
- Include a **README** with your observations
- Your project must compile and run on Linux/Mac (Ubuntu/WSL) using **`g++ -std=c++11`**
- **Include a Makefile** with `-g -Wall -std=c++11` flags