Database Indexing and Range Queries Using BBST

Project Overview

In this project, you will design and implement an in-memory database index using a Balanced Binary Search Tree (BBST). The goal is to efficiently manage a dynamic set of records, supporting fast insertion, deletion, and retrieval operations—including range queries. Additionally, you will explore augmenting the tree nodes with aggregate data (such as count or sum) to extend the index's capabilities.

Project Objectives

- **Data Structure Implementation:** Build a BBST (e.g., AVL Tree, Red-Black Tree) to store and manage records.
- **Efficient Operations:** Ensure that insertion, deletion, and range queries are performed in O(log n) time.
- **Data Augmentation:** Augment nodes with additional aggregate information to support extended queries.
- **Practical Application:** Demonstrate how efficient indexing can improve database operations, focusing on quick lookups and range retrieval.

Background

Database indexes are essential for optimizing query performance in modern databases. By using a BBST, the index maintains a sorted order of keys, which is crucial for efficient search operations. Moreover, augmenting the tree with aggregate data allows the system to quickly answer range queries—such as finding the total count or sum of records within a given interval.

Problem Statement

Design an in-memory database index with the following functionalities:

1. **Insertion:** Add a new record with a key and associated value.
2. **Deletion:** Remove a record by its key.
3. **Range Query:** Retrieve all records (or aggregate data) within a specified key range.
4. **Augmented Queries (Extension):** Optionally, support queries that return aggregated results

(e.g., the total number of records or sum of values) within a key range.

Functional Requirements

- **Insert Operation:**
  - **Input:** A key-value pair.
  - **Operation:** Insert the pair into the BBST while maintaining tree balance and updating aggregate data.
  - **Complexity:** O(log n)
- **Delete Operation:**
  - **Input:** A key.
  - **Operation:** Remove the corresponding record from the tree, ensuring the tree remains balanced and aggregate data is updated.
  - **Complexity:** O(log n)
- **Range Query Operation:**
  - **Input:** A lower and upper bound for the key range.
  - **Operation:** Retrieve and return all records with keys within the specified range.
  - **Complexity:** O(log n + k), where k is the number of records in the range.
- **Aggregate Queries (Optional Extension):**
  - **Input:** A key range.
  - **Operation:** Return aggregated information (e.g., count, sum) of records in that range.
  - **Note:** You must design your node structure to store and update aggregate values during insertions and deletions.

Technical Specifications

- **Data Structure:**
  Implement the index using a BBST. You may choose to implement an AVL tree, a Red-Black tree, or another self-balancing tree variant.
- **Programming Language:**
  You may implement the project in any language of your choice (e.g., C++, Java, Python), but ensure that your code is well-documented and follows best coding practices.
- **Interface:**
  Design a clear API for your index that includes functions for insertion, deletion, and querying.

Ensure the interface is user-friendly and tested with various cases.

Input and Output Format

For the purpose of testing your implementation, your program should accept a sequence of operations through standard input and produce outputs on standard output.

Input Format

- **First Line:**
  An integer `T` indicating the number of operations to be processed.
- **Next T Lines:**
  Each line contains one of the following operations:
  - **Insert Operation:**
    Format: `I key value`
    Example: `I 20 200`
    (This command inserts a record with key `20` and value `200`.)
  - **Delete Operation:**
    Format: `D key`
    Example: `D 20`
    (This command deletes the record with key `20`.)
  - **Range Query Operation:**
    Format: `R low high`
    Example: `R 10 30`
    (This command queries and retrieves all records with keys between `10` and `30` (inclusive).)
  - **Aggregate Query Operation (Optional Extension):**
    Format: `A low high`
    Example: `A 10 30`
    (This command computes and returns an aggregate result—for instance, the sum of values—of records with keys between `10` and `30` (inclusive).)

Output Format

- For each **Range Query** (`R`) operation, output the list of records in ascending order of keys. Each record should be formatted as `key:value`, and records should be separated by a space.
  If no records exist in the specified range, you may output `EMPTY` or leave the line blank.
- For each **Aggregate Query** (`A`) operation (if implemented), output a single value

representing the aggregate result (e.g., the sum or count) for the records in the range.

- **Note:** Insert and Delete operations do not require any output unless you decide to provide confirmation messages.

Sample Input

```
7
I 20 200
I 10 100
I 30 300
R 10 30
D 20
R 10 30
A 10 30
```

Sample Output

```
10:100 20:200 30:300
10:100 30:300
400
```

Explanation:

- After the first three insertions, a range query from 10 to 30 returns all three records.
- After deleting the record with key 20, the subsequent range query returns only the records with keys 10 and 30.
- The aggregate query (assuming it sums the values) returns `100 + 300 = 400` .