**Software Requirements Specification (SRS) Document**

**Self-Balancing Tree with Hash Table and Collision Resolution in C++**

**1. Introduction**

**1.1 Purpose**

The purpose of this project is to develop a **Self-Balancing Binary Search Tree (BST)** using either **AVL Tree** or **Red-Black Tree**, integrated with a **Hash Table** for efficient data storage. Collision resolution will be handled using **Separate Chaining** (via an AVL/Red-Black Tree) and **Linear Probing**.

**1.2 Scope**

This project implements:

* A **Self-Balancing BST** to ensure **O(log n)** insertions, deletions, and searches.
* A **Hash Table** with key-value storage, supporting insertion, deletion, and searching.
* **Collision Resolution Techniques**:
  + **Separate Chaining:** Uses an AVL or Red-Black Tree to handle collisions at each index.
  + **Linear Probing:** Probes the next available slot in case of a collision.
* **Dynamic Resizing**: Hash table resizes when the load factor exceeds a threshold (e.g., 75%).
* Implementation in **C++** using standard data structures and algorithms.

**1.3 Intended Audience**

This project is intended for:

* **Computer Science students** learning advanced data structures.
* **Developers** looking to integrate self-balancing trees with hash tables.
* **Researchers** exploring efficient data storage techniques.

**1.4 Definitions, Acronyms, and Abbreviations**

* **BST**: Binary Search Tree
* **AVL Tree**: A self-balancing BST maintaining a balance factor of -1, 0, or 1.
* **Red-Black Tree**: A self-balancing BST with strict color-based balancing rules.
* **Hash Table**: A data structure for fast key-based retrieval.
* **Separate Chaining**: A collision resolution method using linked structures (AVL/Red-Black Trees).
* **Linear Probing**: A collision resolution method that finds the next available slot sequentially.
* **O(1) Complexity**: Constant time operations (ideal case in hash tables).
* **O(log n) Complexity**: Logarithmic time operations (in self-balancing trees).

**2. Overall Description**

**2.1 Product Perspective**

This system is a standalone **data structure implementation** in C++ designed to optimize **key-value storage and retrieval**. The **hash table** improves search efficiency, while **self-balancing trees** ensure optimal storage within hash table buckets.

**2.2 Product Functions**

* **Insert elements** into the AVL/Red-Black Tree and Hash Table.
* **Search for elements** efficiently using both structures.
* **Delete elements** while maintaining balance.
* **Handle collisions** using either **Separate Chaining** or **Linear Probing**.
* **Dynamically resize** the hash table when the load factor exceeds the threshold.

**2.3 Constraints**

* Implemented in **C++**.
* Uses **AVL or Red-Black Trees** for balancing.
* Hash table operations should be **O(1) on average**.
* Tree operations should be **O(log n)**.
* Memory usage should be optimized to prevent excessive overhead.

**3. Specific Requirements**

**3.1 Functional Requirements**

**3.1.1 AVL Tree / Red-Black Tree**

* **FR1:** The system shall insert elements into the self-balancing BST while maintaining balance.
* **FR2:** The system shall delete elements while preserving tree balance.
* **FR3:** The system shall allow searching for elements in **O(log n)** time.
* **FR4:** The system shall perform rotations (left/right) for balancing.

**3.1.2 Hash Table**

* **FR5:** The system shall insert, search, and delete elements from the hash table.
* **FR6:** The system shall use a **hash function** to determine storage location.
* **FR7:** The system shall implement **dynamic resizing** when the load factor exceeds **75%**.

**3.1.3 Collision Resolution**

* **FR8:** The system shall support **Separate Chaining** using an AVL/Red-Black Tree.
* **FR9:** The system shall support **Linear Probing** for collision resolution.
* **FR10:** The system shall allow users to choose the collision resolution strategy.

**4. Non-Functional Requirements**

* **NFR1:** The system shall be implemented in **C++**.
* **NFR2:** The system shall execute insert, search, and delete operations **efficiently**.
* **NFR3:** The system shall ensure memory efficiency by minimizing unnecessary allocations.
* **NFR4:** The system shall be modular, allowing easy integration with other applications.
* **NFR5:** The system shall be tested with different dataset sizes to ensure reliability.

**5. System Design**

**5.1 Self-Balancing Tree (AVL / Red-Black Tree)**

* **Nodes** contain: **key**, **height/color**, **left child**, **right child**.
* **Rotations (left/right)** ensure balance after insertions and deletions.

**5.2 Hash Table**

* **Array-based** storage with indexing based on **hash function**.
* **Separate Chaining** stores collided elements in an **AVL/Red-Black Tree** at the same index.
* **Linear Probing** finds the next available slot sequentially.

**5.3 Collision Handling**

* **Separate Chaining** → Tree stores multiple elements in a single bucket.
* **Linear Probing** → Finds next open space in a circular manner.

**6. Example Use Case**

**6.1 Scenario: Storing Student IDs and Names**

1. **Insert student IDs** {12345, 23456, 34567} into the hash table.
2. **Search** for a student by ID (23456).
3. **Handle collisions** using **Separate Chaining (AVL/Red-Black Tree) or Linear Probing**.
4. **Delete** a student by ID.
5. **Resize** the hash table if load factor exceeds 75%.

**7. Expected Deliverables**

1. **Source Code** for AVL/Red-Black Tree and Hash Table with collision handling.
2. **Documentation** explaining design decisions and usage.
3. **Test Cases** to verify correctness and efficiency, including edge cases.

**8. Conclusion**

This project provides an **efficient data structure** combining **Self-Balancing Trees** and **Hash Tables** with **collision resolution**. The combination ensures **fast lookups (O(1))** while maintaining **efficient insertions and deletions (O(log n))**. By implementing both **Separate Chaining** and **Linear Probing**, this project demonstrates **real-world data handling efficiency** in **C++**.

**9.Advantages & Disadvantages of This Project**

**Advantages:**

1. **Fast Lookups:** Hash table provides **O(1) average search time** for quick access.
2. **Efficient Insertions & Deletions:** Self-balancing trees ensure **O(log n)** operations.
3. **Collision Handling Flexibility:** Users can choose between **Separate Chaining (trees)** or **Linear Probing** based on needs.
4. **Dynamic Resizing:** Prevents excessive collisions and maintains performance.
5. **Optimized Memory Usage:** Unlike linked lists in chaining, trees provide structured storage.

**Disadvantages:**

1. **Higher Complexity:** Implementing **AVL/Red-Black Trees** along with a hash table increases code complexity.
2. **Extra Memory Usage:** Trees in Separate Chaining require more memory than traditional linked lists.
3. **Performance Overhead:** Balancing operations in self-balancing trees may add extra computation.
4. **Load Factor Dependence:** Hash table efficiency depends on the **load factor and resizing strategy**.
5. **Slower Worst-Case Lookups:** If hash function distributes poorly, lookup times may degrade to **O(log n) (Separate Chaining) or O(n) (Linear Probing in a full table)**.

This structure balances **speed vs. efficiency**, making it great for optimized data storage but requiring careful design choices.