Final Report

Development of an Application for Simulating and Measuring the Performance of Self-Balancing Trees (AVL and Red-Black) in Java and C++

Student: Bîrdău Cătălina-Elena

Specialization: Information Systems for e-Business (ISB)

Group: ISB2

Course: Modelling and Performance Evaluation

Professor: Mihai Mocanu

CONTENTS

1. Introduction

The purpose of this project is to develop an application to simulate and evaluate the performance of two prominent self-balancing tree data structures: AVL Trees and Red-Black Trees. These structures are crucial in ensuring efficient data storage and retrieval in various computer science domains. The project will examine their performance across three key operations:

- Insertion
- Deletion
- Search

Additionally, the application will compare the behavior of these data structures in Java and C++, offering insights into the performance characteristics and implementation complexity of these languages.

The implementation and code for this project are available on GitHub: https://github.com/birdaucatalina/self-balancing-trees-performance

2. Tools and Technologies Used

For the development and evaluation of the application, the following tools and technologies were utilized:

1. **Java 17 (LTS):**
   - A high-level, object-oriented language chosen for implementing AVL and Red-Black Trees due to its stability (LTS version) and strong performance benchmarking capabilities.
2. **Eclipse IDE 2022:**
   - A widely used IDE for Java development, providing features like code completion, debugging tools, and project management for efficient development.
3. **C++14:**
   - A performance-oriented programming language chosen for its low-level memory management and speed, used to implement the same data structures for comparison with Java.
4. **Visual Studio 2022:**
   - A powerful IDE for C++ development, used for writing, compiling, and debugging C++ code, providing an optimized environment for performance testing.

3. Implementation Details of AVL and Red-Black Trees

3.1 AVL Trees

An **AVL Tree** is a self-balancing binary search tree (BST). Each node in the tree maintains a balance factor, which is the difference between the heights of the left and right subtrees. The balance factor must always be in the range [-1, 0, 1] to ensure the tree remains balanced.

Operations:

1. **Insertion:**
   - Inserts the new key in a manner similar to a standard BST.
   - After insertion, the heights of the nodes are updated, and the balance factor is checked. If the balance factor violates the AVL property, **rotations** (single or double) are performed to restore balance.
   - Code Explanation:

```java
// Recursive method to insert a key and balance the tree

 public NodeAVL insert(NodeAVL node, int key) {
     if (node == null) {
         return new NodeAVL(key);
     }

     if (key < node.getKey()) {
         node.setLeft(insert(node.getLeft(), key));
     } else if (key > node.getKey()) {
         node.setRight(insert(node.getRight(), key));
     } else {
         // Duplicate keys are not allowed in AVL tree
         return node;
     }

     node.setHeight(1 + Math.max(height(node.getLeft()),
height(node.getRight())));

     int balance = getBalance(node);

     // Balance the tree
     if (balance > 1 && key < node.getLeft().getKey()) {
         return rotateRight(node);
     }
     if (balance < -1 && key > node.getRight().getKey()) {
         return rotateLeft(node);
     }
     if (balance > 1 && key > node.getLeft().getKey()) {
         node.setLeft(rotateLeft(node.getLeft()));
         return rotateRight(node);
     }
     if (balance < -1 && key < node.getRight().getKey()) {
         node.setRight(rotateRight(node.getRight()));
         return rotateLeft(node);
     }
     return node;

 }
```

   - Rotations:
     - Left Rotation: Balances a node when the right subtree is heavier.
     - Right Rotation: Balances a node when the left subtree is heavier.

2. **Deletion:**
   - Follows the standard BST deletion procedure: find the node, replace it with its successor/predecessor, and remove it.
   - After deletion, the heights of nodes are updated, and the tree is rebalanced by performing rotations where needed.
   - Complexity: O (log n) due to the height-balanced nature.
3. **Search:**
   - Operates like a standard BST: traverse left or right depending on the key.
   - Complexity: O (log n) since the tree remains balanced after every operation.

3.2 Red-Black Trees

A **Red-Black Tree** is another self-balancing BST but with different balancing rules compared to AVL Trees. It uses **coloring** (red and black) and a set of rules to ensure balance.

Key Properties:

1. Each node is either red or black.
2. The root node is always black.
3. Every path from a node to its descendant leaves contains the same number of black nodes.
4. Red nodes cannot have red children (no consecutive red nodes).

Operations:

1. **Insertion**:
   - Insert the new key like a BST.
   - Fix violations of the Red-Black properties by applying rotations and recoloring.
2. **Deletion**:
   - Removes a node like in BST, but additional fixes (recoloring and rotations) are applied to maintain Red-Black properties.
   - Balancing is less strict than AVL, so fewer rotations are typically needed.
   - Complexity: O (log n).
3. **Search**:
   - Similar to AVL, search is performed using the BST traversal rules.
   - Complexity: O(log n) because the tree remains balanced, though it may not be as tightly balanced as AVL.

3.3 Performance Measurement

Time Measurement

1. **In Java**:
    - The System.nanoTime() method was used to measure the execution time of operations in nanoseconds:

```
long startTime = System.nanoTime();
// Operation (e.g., insertion, deletion, search)
long endTime = System.nanoTime();

System.out.println("Execution time: " + (endTime - startTime) + "
ns");
```

2. **In C++**:
    - The <chrono> library was used for precise time measurement:

```
auto start = std::chrono::high_resolution_clock::now();
// Operation
auto end = std::chrono::high_resolution_clock::now();
std::cout << "Execution time: "
    << std::chrono::duration_cast<std::chrono::nanoseconds>(end -
start).count()
    << " ns" << std::endl;
```

Memory Measurement

1. **In Java**:
    - The Runtime class was used to calculate memory usage:

```
long memoryBefore = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
// Operation …
long memoryAfter = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();

System.out.println("Memory used: " + (memoryAfter - memoryBefore)
+ " bytes");
```

2. **In C++**:
    - Memory usage was measured using the GetProcessMemoryInfo function on Windows:

```
void printMemoryUsage(const string& message) {
    PROCESS_MEMORY_COUNTERS pmc;
    if (GetProcessMemoryInfo(GetCurrentProcess(), &pmc, sizeof(pmc))) {
        cout << message << " Memory used: " << pmc.WorkingSetSize / 1024 <<
" KB" << endl;
    }
```

```
                else {
                    cerr << "Failed to get memory info" << endl;
                }
            }
```

3.4. Random Data Generator

To generate datasets for experiments, a custom **Java program** was developed. The program outputs random integers into a `.txt` file with configurable parameters:

- `numElements`: Number of integers to generate.
- `bound`: Upper limit for random values.

Code Implementation:

```
public class RandomGenerator {

      public static void main(String[] args) {
            String inputFileName = "random_output.txt";
            int numElements = 10000000;
            int bound = 1000000000;

        try {
                  saveRandomData(inputFileName, numElements, bound);
            } catch (IOException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }
      }

      // Save random integers to a file in resources
      public static void saveRandomData(String filename, long size, long
bound) throws IOException {
        try (PrintWriter pw = new PrintWriter(new FileWriter("src/" +
filename))) {
            Random random = new Random();
            for (long i = 0; i < size; i++) {
                pw.print(random.nextLong(bound) + " ");
            }
        }
        System.out.println("Random data saved to " + filename);
    }

}
```

4. Experimental Data Results

1. Simulation 1 Results

Input Configurations:

- int numElements = 10000000;
- int bound = 1000000000;
- int searchKey = 133857562;
- int deleteKey = 298200287;

Language	Tree Type	Insertion Time (µs)	Search Time (µs)	Delete Time (µs)	Memory Before Insert (KB)	Memory After Insert (KB)	Memory Change (KB)
Java	AVL Tree	9,257,248.10	4.2	16.3	795,650	940,032	144,382
Java	Red-Black Tree	8,650,777.40	6.1	9.001	1,209,550	1,468,620	259,070
C++	AVL Tree	24,382,687.60	0.2	4.7	52,372	986,632	934,260
C++	Red-Black Tree	11,540,076.70	0.3	1.8	52,376	1,148,236	1,095,860

Observations:

- Insertion Time: Java AVL insertion is significantly faster than C++ AVL but slower than C++ Red-Black.
- Search Time: C++ trees outperform Java trees, with Red-Black being slightly slower than AVL in C++.
- Delete Time: Deletion in C++ is much faster than in Java, especially for Red-Black Trees.
- Memory Usage: Java consumes more memory overall, but the increase during insertion is more moderate compared to C++. C++ Red-Black Trees have the highest memory usage after insertion.

2. Simulation 2 Results

Input Configurations:

- int numElements = 1000000;
- int bound = 100000000;
- int searchKey = 62349935;
- int deleteKey = 28547108;

Language	Tree Type	Insertion Time (µs)	Search Time (µs)	Delete Time (µs)	Memory Before Insert (KB)	Memory After Insert (KB)	Memory Change (KB)
Java	AVL Tree	618,285.600	5.000	16.600	79,912	114,476	34,564
Java	Red-Black Tree	415,296.100	6.000	12.800	194,939	219,136	24,197
C++	AVL Tree	1,571,690.300	0.700	4.000	8,052	101,888	93,836
C++	Red-Black Tree	796,058.200	0.600	1.200	8,056	118,008	109,952

Observations:

- Insertion Time: Java Red-Black Tree has the fastest insertion, while C++ AVL Tree takes the longest.
- Search Time: C++ implementations (both AVL and Red-Black) are significantly faster in searching.
- Delete Time: C++ Red-Black Tree shows superior performance for deletion, whereas Java AVL Tree is the slowest.
- Memory Usage: Java Red-Black Tree uses the most memory, while C++ AVL Tree shows the lowest memory usage post-insertion.

3. Simulation 3 Results

Input Configurations:

- int numElements = 500000;
- int bound = 50000000;
- int searchKey = 35877928;
- int deleteKey = 34117044;

Language	Tree Type	Insertion Time (µs)	Search Time (µs)	Delete Time (µs)	Memory Before Insert (KB)	Memory After Insert (KB)	Memory Change (KB)
Java	AVL Tree	250,067.700	4.400	14.700	38,621	49,603	10,982
Java	Red-Black Tree	172,526.300	6.200	8.100	91,839	107,608	15,769
C++	AVL Tree	825,792.600	0.200	3.700	8,036	53,748	45,712
C++	Red-Black Tree	431,577.200	0.200	0.900	6,688	61,808	55,120

Observations:

- Insertion Time: Java Red-Black Tree performs the fastest insertion, while C++ AVL Tree takes the longest.
- Search Time: Both C++ implementations excel in search time, with AVL and Red-Black achieving the same performance.
- Delete Time: C++ Red-Black Tree is the fastest in deletion, while Java AVL Tree takes the longest.
- Memory Usage: C++ Red-Black Tree shows the largest increase in memory usage post-insertion, whereas Java AVL Tree has the smallest memory footprint.

4. Simulation 4 Results

Input Configurations:

- int numElements = 10000;
- int bound = 100000;
- int searchKey = 13032;
- int deleteKey = 13032;

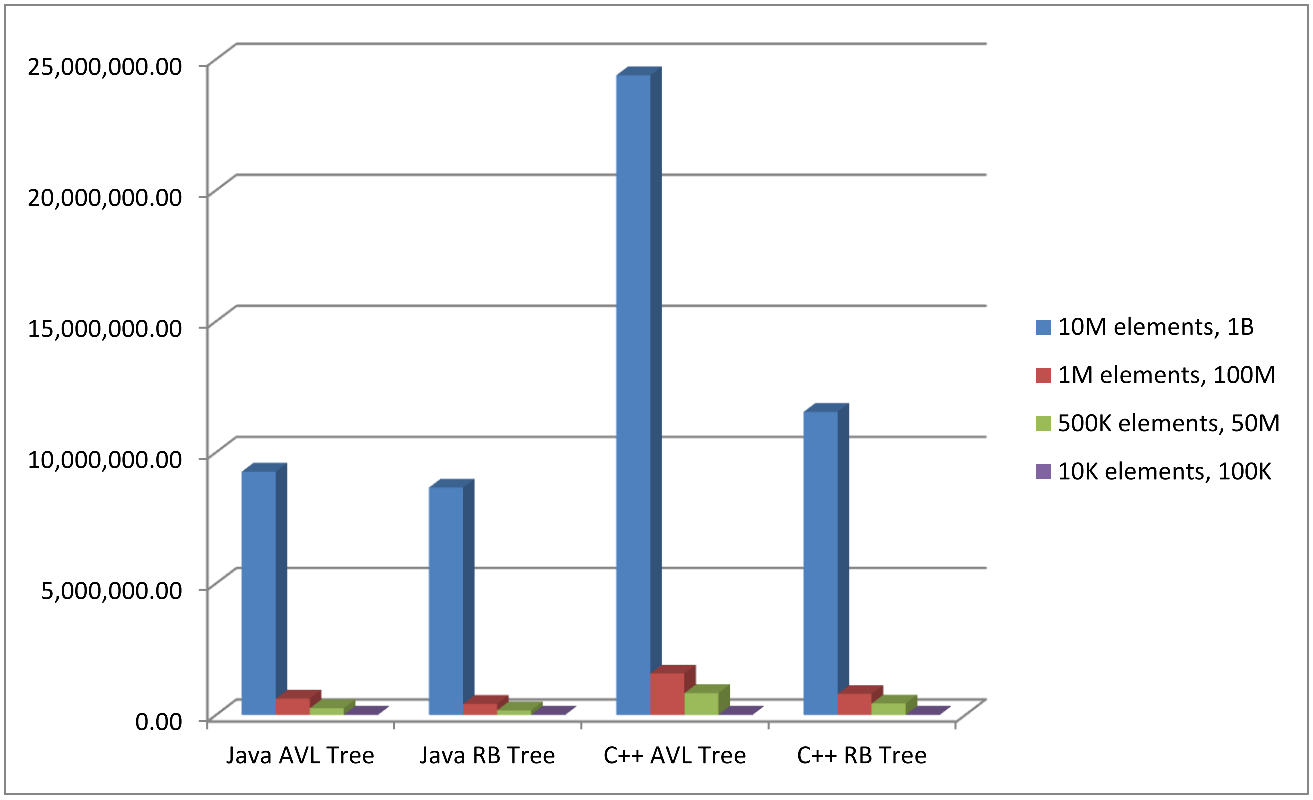
Language	Tree Type	Insertion Time (µs)	Search Time (µs)	Delete Time (µs)	Memory Before Insert (KB)	Memory After Insert (KB)	Memory Change (KB)
Java	AVL Tree	9,454.200	5.200	16.600	3,004	3,455	451
Java	Red-Black Tree	4,504.600	6.400	8.300	4,096	4,548	452
C++	AVL Tree	8,792.400	0.500	4.400	4,056	5,076	1,020
C++	Red-Black Tree	3,826.400	1.600	1.200	4,056	5,356	1,300

Observations:

- Insertion Time: C++ Red-Black Tree performs the fastest insertion, while Java AVL Tree takes the longest.
- Search Time: C++ AVL Tree has the fastest search performance, while Java Red-Black Tree takes the longest.
- Delete Time: C++ Red-Black Tree deletes the key most efficiently, whereas Java AVL Tree requires the most time.
- Memory Usage: Memory change is minimal for Java implementations compared to C++ trees. C++ Red-Black Tree exhibits the largest increase in memory usage.

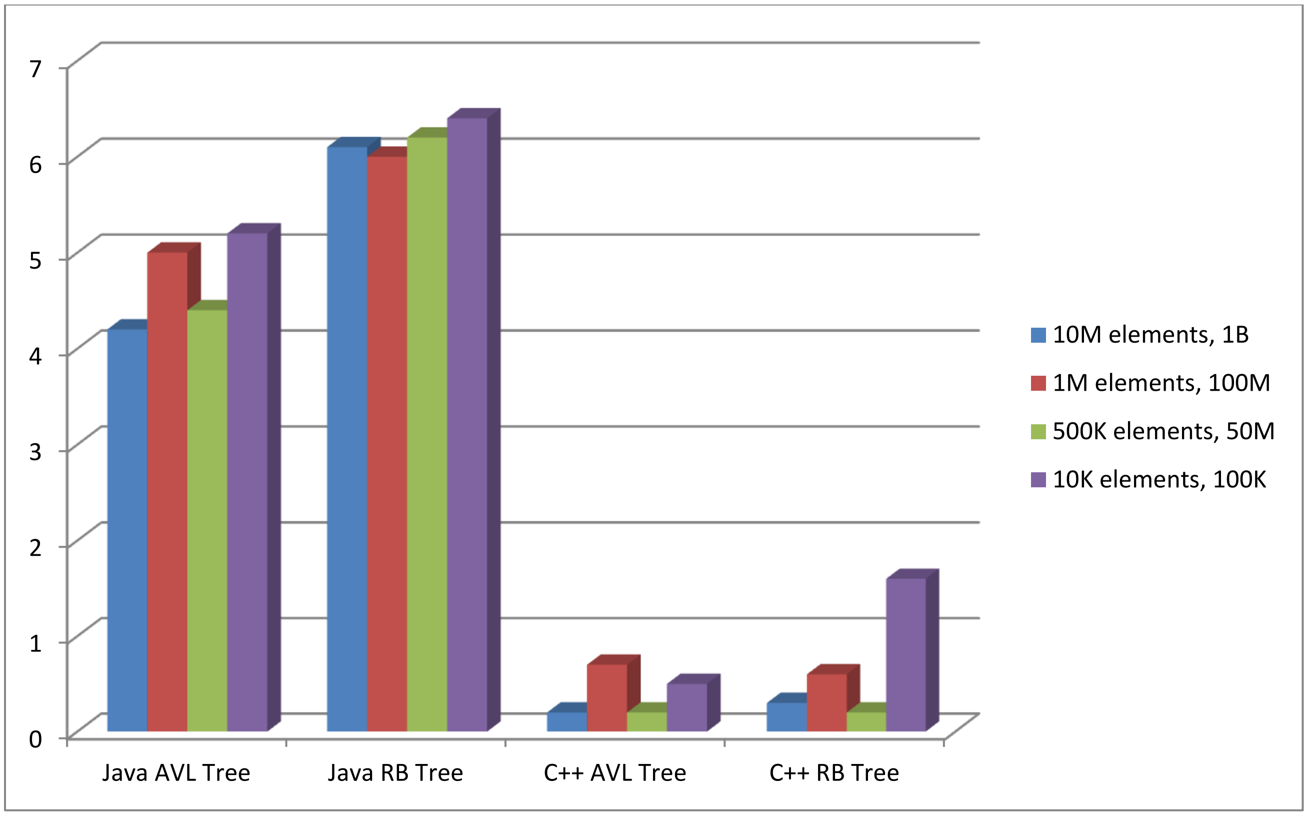
5. Comparisons

C++ vs. Java Insert Operation



Java vs. C++: For all dataset sizes, Java consistently shows better insertion performance compared to C++. The difference is particularly noticeable with larger datasets (e.g., 10M elements), where Java AVL Tree and RB Tree are faster than their C++ counterparts by a significant margin. This could be attributed to Java's optimized memory management and garbage collection for insertion-heavy operations.

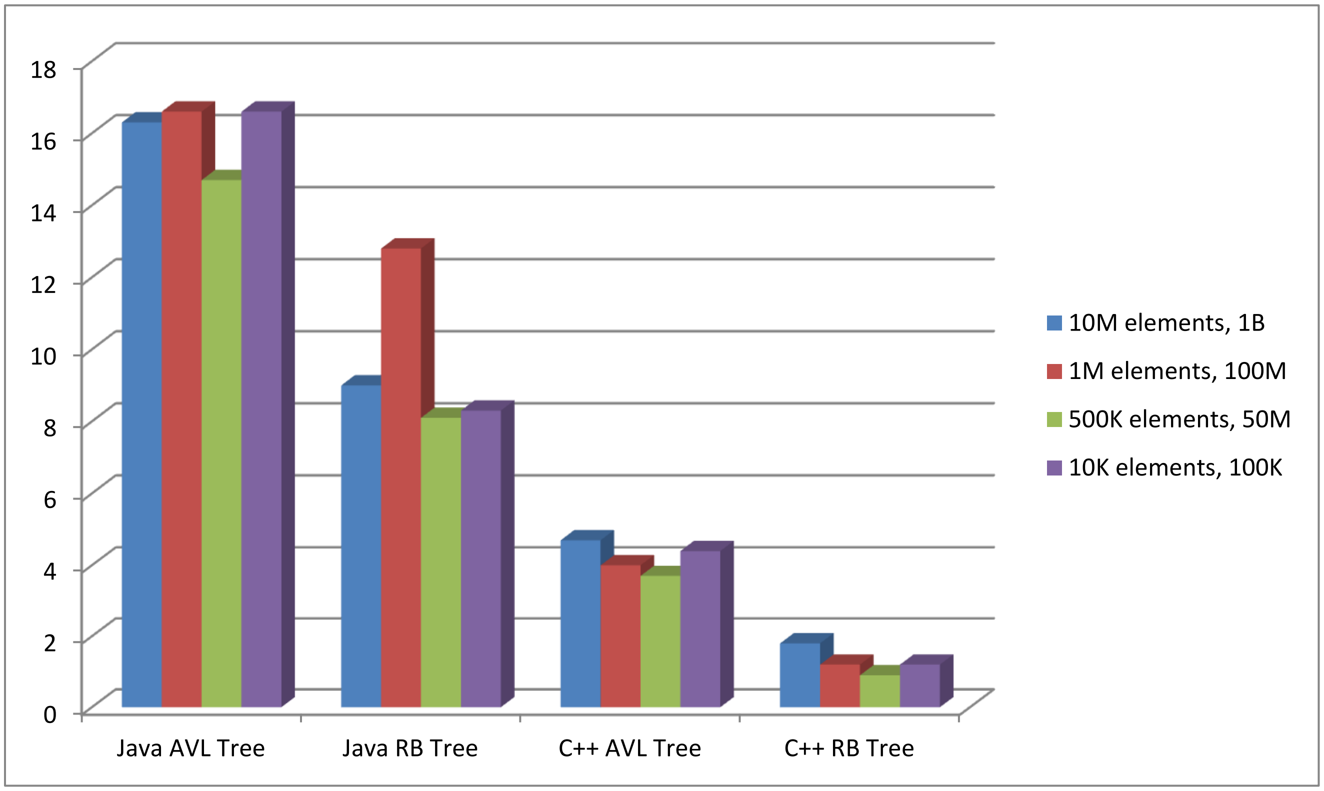
C++ vs. Java Search Operation



C++ vs. Java: C++ consistently outperforms Java in search operations, regardless of dataset size. The difference is especially stark for larger datasets, with C++ AVL and RB trees taking microseconds for searches, while Java trees take more time.

- **Example**: For 10M elements, C++ AVL Tree takes 0.2 microseconds, whereas Java AVL Tree takes 4.2 microseconds.
- C++ seems to be significantly more efficient for search operations, possibly due to the language's lower-level memory management and faster access to memory.

C++ vs. Java Delete Operation



C++ vs. Java: C++ also outperforms Java in deletion operations across all dataset sizes. The deletion times for C++ trees are notably lower compared to Java, which might be a result of the same memory management advantages that benefit search times as well.

- **Example**: For 10M elements, C++ AVL Tree takes 4.7 microseconds for deletion, while Java AVL Tree takes 16.3 microseconds.

6. Conclusion

While Java excels in insertion-heavy scenarios, C++ provides superior performance for search and deletion operations, as well as more efficient memory usage. This indicates that C++ is better suited for scenarios where fast searches and deletions are critical, and where memory constraints are a concern. Java, on the other hand, may be a better choice for applications where insertion is the primary operation and where ease of memory management (due to garbage collection) outweighs the performance benefits of lower-level memory control.

In summary, the choice between Java and C++ for implementing self-balancing trees largely depends on the specific needs of the application:

- Java: Better suited for insertion-heavy applications, offering ease of memory management and faster insertions.
- C++: Ideal for applications where search and deletion performance, as well as memory efficiency, are paramount. C++ provides superior performance for these operations due to its low-level memory handling and minimal runtime overhead.

These findings align with the general expectations based on the nature of both programming languages and their memory management models, with Java being optimized for ease of use and C++ for performance-intensive tasks.

7. References

1. https://www.geeksforgeeks.org/introduction-to-red-black-tree/, accessed in January, 2025
2. https://www.geeksforgeeks.org/introduction-to-avl-tree/ , accessed in January, 2025