

# Final Project - Inverted Index and Comparative Analysis

Arthur Rabello Oliveira<sup>1</sup>, Gabrielle Mascarello, Eliane Moreira, Nicolas Spaniol, Gabriel Carneiro

## Abstract

We present the implementation and comparative analysis of three fundamental data structures for indexing and searching textual documents: the Binary Search Tree (BST), the AVL Tree, and the Red-Black Tree (RBT). Each structure was implemented with its core operations, including insertion and search. Unit tests were developed to validate the correctness and performance of these implementations. We also provide a further comprehensive comparative study of the three trees based on their time complexity, balancing efficiency, and suitability for document indexing. The results demonstrate the trade-offs between implementation complexity and query performance, offering insights into the practical considerations for choosing appropriate search tree structures in information retrieval systems.

## Contents

1. Introduction .....	3
1.1. Motivation .....	3
1.2. Problem Statement .....	3
1.3. Overview of Search Trees .....	3
2. Data Structures Overview .....	3
2.1. Binary Search Tree (BST) .....	3
2.2. AVL Tree .....	3
2.3. Red-Black Tree (RBT) .....	3
2.4. Inverted Index .....	3
2.5. Comparison of Properties .....	3
3. Implementations .....	3
3.1. Binary Search Tree (BST) .....	3
3.1.1. Algorithms .....	3
3.1.2. Complexity Analysis .....	3
3.2. AVL Tree .....	3
3.2.1. Algorithms .....	3
3.2.2. Complexity Analysis .....	3
3.3. Red-Black Tree (RBT) .....	3
3.3.1. Algorithms .....	3
3.3.2. Complexity Analysis .....	3
3.4. Inverted Index .....	3
3.4.1. Algorithms .....	3
3.4.2. Complexity Analysis .....	3
4. Testing and Validation .....	3
4.1. Unit Testing Method .....	3
4.1.1. Binary Search Tree (BST) .....	3
4.1.2. AVL Tree .....	3
4.1.3. Red-Black Tree (RBT) .....	3
5. Comparative Analysis .....	3

---

<sup>1</sup>Escola de Matemática Aplicada, Fundação Getúlio Vargas (FGV/EMAp), email: [arthur.oliveira.1@fgv.edu.br](mailto:arthur.oliveira.1@fgv.edu.br)

5.1. The Experiment .....	4
5.2. Memory Usage .....	4
5.3. Time Complexity .....	4
6. Conclusion .....	4
6.1. Summary of Findings .....	4
7. Source code .....	4
8. Task Division (Required by the professor) .....	4
8.1. Arthur Rabello Oliveira .....	4
8.2. Gabrielle Mascarello .....	5
8.3. Eliane Moreira .....	5
8.4. Nicolas Spaniol .....	7
8.5. Gabriel Carneiro .....	7

# **1. Introduction**

## **1.1. Motivation**

## **1.2. Problem Statement**

## **1.3. Overview of Search Trees**

# **2. Data Structures Overview**

## **2.1. Binary Search Tree (BST)**

## **2.2. AVL Tree**

## **2.3. Red-Black Tree (RBT)**

## **2.4. Inverted Index**

## **2.5. Comparison of Properties**

# **3. Implementations**

## **3.1. Binary Search Tree (BST)**

### **3.1.1. Algorithms**

### **3.1.2. Complexity Analysis**

## **3.2. AVL Tree**

### **3.2.1. Algorithms**

### **3.2.2. Complexity Analysis**

## **3.3. Red-Black Tree (RBT)**

### **3.3.1. Algorithms**

### **3.3.2. Complexity Analysis**

## **3.4. Inverted Index**

### **3.4.1. Algorithms**

### **3.4.2. Complexity Analysis**

# **4. Testing and Validation**

## **4.1. Unit Testing Method**

### **4.1.1. Binary Search Tree (BST)**

### **4.1.2. AVL Tree**

### **4.1.3. Red-Black Tree (RBT)**

# **5. Comparative Analysis**

## 5.1. The Experiment

## 5.2. Memory Usage

## 5.3. Time Complexity

## 6. Conclusion

### 6.1. Summary of Findings

## 7. Source code

(repository)

## 8. Task Division (Required by the professor)

### 8.1. Arthur Rabello Oliveira

Implemented and documented the following functions:

```
1  BinaryTree* createTree(){ //artu
2      BinaryTree* newBinaryTree = new BinaryTree{nullptr};
3      return newBinaryTree;
4  }
5
6  SearchResult search(BinaryTree* binary_tree, const std::string& word) { //artu
7      auto start_time = std::chrono::high_resolution_clock::now(); //start
      measuring time
8
9      if (binary_tree == nullptr || binary_tree->root == nullptr) {
10         auto end_time = std::chrono::high_resolution_clock::now(); //done
            lol
11         double duration =
            std::chrono::duration_cast<std::chrono::microseconds>(end_time -
            start_time).count() / 1000.0;
12         return {0, {}, duration, 0};
13
14     } else {
15         Node* current_node = binary_tree->root;
16         int number_of_comparisons = 0;
17
18         while (current_node != nullptr) {
19             number_of_comparisons++;
20
21             int compareResult = strcmp(word.c_str(), current_node->
                word.c_str());
22
23             if (compareResult == 0) { //found!
24                 auto end_time = std::chrono::high_resolution_clock::now();
```

```

25         double duration =
            std::chrono::duration_cast<std::chrono::microseconds>(end_time
            - start_time).count() / 1000.0;
26         return {1, current_node->documentIds, duration,
            number_of_comparisons};
27
28     } else if (compareResult < 0) {
29         current_node = current_node->left; //go left because word is
            smaller
30     } else {
31         current_node = current_node->right; //go right because word
            is bigger
32     }
33 }
34
35 //if word not found
36 auto end_time = std::chrono::high_resolution_clock::now();
37 double duration =
    std::chrono::duration_cast<std::chrono::microseconds>(end_time -
    start_time).count() / 1000.0;
38 return {0, {}, duration, number_of_comparisons};
39 }
40 }

```

## 8.2. Gabrielle Mascarello

## 8.3. Eliane Moreira

Implemented and documented the following functions:

```

1  InsertResult insert(BinaryTree* binary_tree, const std::string& word, int
    documentId){ //eliane
2      InsertResult result;
3      int comparisons = 0;
4      auto start_time = std::chrono::high_resolution_clock::now();
5
6      Node* newNode = nullptr;
7
8      if(binary_tree->root == nullptr){
9          newNode = createNode(word, {documentId});
10         binary_tree->root = newNode;
11     } else {
12         Node* current = binary_tree->root;
13         Node* parent = nullptr;
14
15         while (current != nullptr){
16             parent = current;
17             comparisons++;
18

```

```

19         if(word == current->word){
20             //checks if documentId has already been added
21             bool found = false;
22             for(size_t i = 0; i < current->documentIds.size(); i++){
23                 if (current->documentIds[i] == documentId) {
24                     found = true;
25                     break;
26                 }
27             }
28
29             if (found == false) {
30                 current->documentIds.push_back(documentId);
31             }
32
33             auto end_time = std::chrono::high_resolution_clock::now();
34             double duration =
35                 std::chrono::duration_cast<std::chrono::microseconds>(end_time -
36                     start_time).count() / 1000.0;
37
38             result.numComparisons = comparisons;
39             result.executionTime = duration;
40             return result;
41
42         } else if(word < current->word){
43             current = current->left;
44         } else {
45             current = current->right;
46         }
47     }
48
49     newNode = createNode(word, {documentId});
50     newNode->parent = parent;
51
52     if(word < parent->word){
53         parent->left = newNode;
54     } else {
55         parent->right = newNode;
56     }
57
58     auto end_time = std::chrono::high_resolution_clock::now();
59     double duration =
60         std::chrono::duration_cast<std::chrono::microseconds>(end_time -
61             start_time).count() / 1000.0;
62
63     result.numComparisons = comparisons;

```

```

61     result.executionTime = duration;
62     return result;
63 }

```

## 8.4. Nicolas Spaniol

## 8.5. Gabriel Carneiro

Implemented and documented the following functions:

```

Node* createNode(std::string word, std::vector<int>documentIds, int color
1  = 0) { //sets for 0 if it the tree doesnt support red-black, gabriel
    carneiro
2
3     Node* newNode = new Node;
4     newNode->word = word;
5     newNode->documentIds = documentIds;
6     newNode->parent = nullptr;
7     newNode->left = nullptr;
8     newNode->right = nullptr;
9     newNode->height = 1; //height of a new node is 1
10    newNode->isRed = color; //0 for red, 1 for black
11    return newNode;
12 }
13
14 void deleteBinaryTree(BinaryTree* binary_tree){ //gabriel carneiro
15     Node* root = binary_tree->root;
16
17     if(root != nullptr){
18         Node* leftNode = root->left;
19         BinaryTree* leftSubTree = createTree();
20         leftSubTree->root = leftNode;
21
22         Node* rightNode = root->right;
23         BinaryTree* rightSubTree = createTree();
24         rightSubTree->root = rightNode;
25
26         delete root;
27
28         deleteBinaryTree(leftSubTree);
29         deleteBinaryTree(rightSubTree);
30
31         delete binary_tree;
32     }
33 }

```