Final Project - Inverted Index and Comparative Analysis

Arthur Rabello Oliveira[1], Gabrielle Mascarello, Eliane Moreira, Nícolas 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]Escola de Matemática Aplicada, Fundação Getúlio Vargas (FGV/EMAp), email: arthur.oliveira.1@fgv.edu.br

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. Complextity Analysis

3.2. AVL Tree

3.2.1. Algorithms

3.2.2. Complextity Analysis

3.3. Red-Black Tree (RBT)

3.3.1. Algorithms

3.3.2. Complextity Analysis

3.4. Inverted Index

3.4.1. Algorithms

3.4.2. Complextity 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:

```cpp
BinaryTree* createTree(){ //artu
        BinaryTree* newBinaryTree = new BinaryTree{nullptr};
        return newBinaryTree;
    }

SearchResult search(BinaryTree* binary_tree, const std::string& word) { //artu
        auto start_time = std::chrono::high_resolution_clock::now(); //start measuring time

        if (binary_tree == nullptr || binary_tree->root == nullptr) {
            auto end_time = std::chrono::high_resolution_clock::now(); //done lol
            double duration = std::chrono::duration_cast<std::chrono::microseconds>(end_time - start_time).count() / 1000.0;
            return {0, {}, duration, 0};

        } else {
            Node* current_node = binary_tree->root;
            int number_of_comparisons = 0;

            while (current_node != nullptr) {
                number_of_comparisons++;

                int compareResult = strcmp(word.c_str(), current_node->word.c_str());

                if (compareResult == 0) { //found!
                    auto end_time = std::chrono::high_resolution_clock::now();
```

```
                    double duration =
                    std::chrono::duration_cast<std::chrono::microseconds>(end_tim
                    - start_time).count() / 1000.0;
                    return {1, current_node->documentIds, duration,
                    number_of_comparisons};

                } else if (compareResult < 0) {
                    current_node = current_node->left; //go left because word is
                    smaller
                } else {
                    current_node = current_node->right; //go right because word
                    is bigger
                }
            }

            //if word not found
            auto end_time = std::chrono::high_resolution_clock::now();
            double duration =
            std::chrono::duration_cast<std::chrono::microseconds>(end_time -
            start_time).count() / 1000.0;
            return {0, {}, duration, number_of_comparisons};
        }
    }
```

8.2. Gabrielle Mascarello

8.3. Eliane Moreira

Implemented and documented the following functions:

```cpp
InsertResult insert(BinaryTree* binary_tree, const std::string& word, int
documentId){ //eliane
        InsertResult result;
        int comparisons = 0;
        auto start_time = std::chrono::high_resolution_clock::now();

        Node* newNode = nullptr;

        if(binary_tree->root == nullptr){
            newNode = createNode(word, {documentId});
            binary_tree->root = newNode;
        } else {
            Node* current = binary_tree->root;
            Node* parent = nullptr;

            while (current != nullptr){
                parent = current;
                comparisons++;

```

```
                if(word == current->word){
                    //checks if documentId has already been added
                    bool found = false;
                    for(size_t i = 0; i < current->documentIds.size(); i++){
                        if (current->documentIds[i] == documentId) {
                            found = true;
                            break;
                        }
                    }

                    if (found == false) {
                        current->documentIds.push_back(documentId);
                    }

                    auto end_time = std::chrono::high_resolution_clock::now();
                    double duration =
                    std::chrono::duration_cast<std::chrono::microseconds>(end_tim
                    - start_time).count() / 1000.0;

                    result.numComparisons = comparisons;
                    result.executionTime = duration;
                    return result;

                } else if(word < current->word){
                    current = current->left;
                } else {
                    current = current->right;
                }
            }

            newNode = createNode(word, {documentId});
            newNode->parent = parent;

            if(word < parent->word){
                parent->left = newNode;
            } else {
                parent->right = newNode;
            }
        }

        auto end_time = std::chrono::high_resolution_clock::now();
        double duration =
        std::chrono::duration_cast<std::chrono::microseconds>(end_time -
        start_time).count() / 1000.0;

        result.numComparisons = comparisons;
```

```cpp
        result.executionTime = duration;
        return result;
    }
```

8.4. Nícolas Spaniol

8.5. Gabriel Carneiro

Implemented and documented the following functions:

```cpp
Node* createNode(std::string word, std::vector<int>documentIds, int color = 0) { //sets for 0 if it the tree doesnt support red-black, gabriel carneiro

        Node* newNode = new Node;
        newNode->word = word;
        newNode->documentIds = documentIds;
        newNode->parent = nullptr;
        newNode->left = nullptr;
        newNode->right = nullptr;
        newNode->height = 1; //height of a new node is 1
        newNode->isRed = color; //0 for red, 1 for black
        return newNode;
    }

    void deleteBinaryTree(BinaryTree* binary_tree){ //gabriel carneiro
        Node* root = binary_tree->root;

        if(root != nullptr){
            Node* leftNode = root->left;
            BinaryTree* leftSubTree = createTree();
            leftSubTree->root = leftNode;

            Node* rightNode = root->right;
            BinaryTree* rightSubTree = createTree();
            rightSubTree->root = rightNode;

            delete root;

            deleteBinaryTree(leftSubTree);
            deleteBinaryTree(rightSubTree);

            delete binary_tree;
        }
    }
```