Fundação Getúlio Vargas

Escola de Matemática Aplicada

Relatório do Projeto Final de Estrutura de Dados

Eficiência de diferentes árvores binárias de busca em um índice invertido

Antonio Batista

Bernardo Quintella

Dilmar Castanheiro

João Vitor Ferreira

Roger Vinícius Augusto

Sumário

1 Introdução

Dado um conjunto de documentos, um índice invertido é uma ferramenta de busca que visa associar dados, sejam eles palavras, números ou outros, aos seus documentos de origem. O projeto tem como objetivo implementar do zero três árvores binárias de busca muito comuns como ferramentas internas de um índice invertido. As árvores implementadas foram a árvore binária de busca (BST), a árvore Adelson-Velsky e Landis (AVL) e a árvore rubro-negra (RBT). Nosso palpite inicial era que a BST levaria menos tempo para inserção, mas muito mais tempo para busca pois é muito suscetível ao desbalanceamento. Suspeitamos também que a AVL seria a mais rápida para busca e mais demorada para inserção, pois as rotações levam um tempo considerável. Já a BRT seria intermediária, mas com desempenho de busca bem superior à BST.

2 Implementação

2.1 Data

data.h e data.cpp: arquivos usados para implementação de funções para leitura dos arquivos com palavras a serem indexadas.

2.1.1 Estruturas (structs)

```
1 struct ProcessResult {
2     int totalWords;
3     double executionTime;
4 };
```

2.1.2 Funções

`struct ProcessResult`

Estrutura que armazena os resultados do processamento de arquivos.

- `totalWords`: Total de palavras processadas (incluindo repetições).
- `executionTime`: Tempo total de processamento em segundos.

`std::string processWord(const std::string& word)`

Processa uma palavra, removendo caracteres não alfanuméricos e convertendo para minúsculas.

- `word`: Palavra a ser processada.

`ProcessResult processFiles(const std::string& directory, int numFiles, const std::functi`

Processa múltiplos arquivos de texto em um diretório.

- `directory`: Caminho do diretório contendo os arquivos.
- `numFiles`: Número de arquivos a processar.
- `insertCallback`: Função de callback para inserção de palavras.
- `tree`: Árvore a ser construída após o processamento.

2.2 Tree Utils

tree_utils.h e tree_utils.cpp: arquivos usados para implementação de estruturas e funções comuns às três árvores.

2.2.1 Estruturas (structs)

```
struct Node {
    std::string word;
    std::vector<int> documentIds;
    Node* parent;
    Node* left;
    Node* right;
    int height;    // usado na AVL
    int isRed;     // usado na RBT // 0 para preto, 1 para vermelho
}

struct BinaryTree {
    Node* root;
    Node* NIL;  // usado na RBT (Opcional)
};

struct InsertResult {
    int numComparisons;
    double executionTime;
};

struct SearchResult {
    int found;
    std::vector<int> documentIds;
    double executionTime;
    int numComparisons;
};
```

2.2.2 Funções

`void printIndex(BinaryTree* tree)`

Imprime as palavras armazenadas em tree, em ordem lexicográfica, seguidas dos índices dos arquivos aos quais elas pertencem.

- `tree` Ponteiro para a árvore.

`void printTreeAux(Node* node, const std::string& prefix, bool isLeft)`

Função interna para formatar a impressão das palavras armazenadas em tree em formato de árvore.

- `node` Ponteiro para o nó raiz.
- `prefix` Prefixo atual. Passar por referência.
- `isLeft` True se o nó raiz está na esquerda de seu nó pai. False caso contrário.

`void printTree(BinaryTree* tree)`

Imprime as palavras armazenadas em tree em formato de árvore.

- `tree` Ponteiro para a árvore.

`int max(int a, int b)`

Calcula o valor máximo entre dois inteiros e retorna-o.

- `a` Primeiro inteiro a ser comparado.
- `b` Segundo inteiro a ser comparado.

`int height(Node* node)`

Calcula a altura de um nó e retorna-a.

- `node` Ponteiro para o nó a ser descoberto a altura.

2.3 Árvore Binária de Busca (BST)

bst.h e bst.cpp: arquivos usados para implementação de funções específicas à BST.

2.3.1 Namespace BST: funções

`BinaryTree* create()`

Inicializa uma árvore binária e retorna-a.

`InsertResult insert(BinaryTree* tree, const std::string& word, int documentId)`

Insere uma palavra na árvore associada a um documentId. Se for inédita cria um novo nó, se já existir adiciona o documentId ao nó associado. Retorna um `InsertResult` com estatísticas sobre o desempenho.

- `tree`: Ponteiro para a BST.
- `word` Palavra a ser inserida. Passar por referência.
- `documentId` Índice do documento que contém a palavra.

`SearchResult search(BinaryTree* tree, const std::string& word)`

Realiza a busca de uma palavra na BST. Retorna um `SearchResult` que diz se a palvra foi achada, em quais documentos e estatísticas sobre o processo.

- `tree`: Ponteiro para a BST.
- `word` Palavra a ser inserida. Passar por referência.

`void destroyNode(Node* node)`

Função auxiliar da destroy.

- `node` Ponteiro para o node a ser destruído.

`void destroy(BinaryTree* tree)`

Libera toda a memória alocada pela BST.

- `tree` Ponteiro para a BST a ser destruída.

2.4 Árvore AVL

avl.h e avl.cpp: arquivos usados para implementação de funções específicas à AVL.

2.4.1 Namespace AVL: funções

`BinaryTree* create()`

Inicializa uma árvore binária AVL e retorna-a.

`int getBalance(Node* node)`

Calcula o fator de balanceamento de um nó.

- `node`: Ponteiro para o nó.

`Node* rightRotate(Node* y)`

Realiza a rotação para a direita de um nó. Retorna o ponteiro para o novo nó na posição de `y`.

- `y`: Ponteiro para o nó a ser rotacionado.

`Node* leftRotate(Node* x)`

Realiza a rotação para a esquerda de um nó. Retorna o ponteiro para o novo nó na posição de `x`.

- `x`: Ponteiro para o nó a ser rotacionado.

`Node* insertAVL(Node* node, const std::string& word, int documentId, int& numComparisons`

Função recursiva auxiliar da `insert`, responsável por manter a árvore balanceada. Retorna o novo nó após a inserção.

- `node`: Ponteiro para o nó raiz da sub-árvore onde a palavra será inserida.
- `word`: Palavra a ser inserida. Passar por referência.
- `documentId`: Índice do documento de origem da palavra.
- `numComparisons`: Ponteiro para o número de comparações realizadas. Passar por referência.

`InsertResult insert(BinaryTree* tree, const std::string& word, int documentId)`

Insere uma palavra na árvore associada a um documentId. Se for inédita, cria um novo nó; se já existir, adiciona o documentId ao nó associado. Retorna um `InsertResult` com estatísticas sobre o desempenho.

- `tree`: Ponteiro para a árvore binária AVL.
- `word`: Palavra a ser inserida. Passar por referência.
- `documentId`: Índice do documento que contém a palavra.

`SearchResult search(BinaryTree* tree, const std::string& word)`

Realiza a busca de uma palavra na árvore binária AVL. Retorna um `SearchResult` que diz se a palavra foi achada, em quais documentos e estatísticas sobre o processo.

- `tree`: Ponteiro para a árvore binária AVL.
- `word`: Palavra a ser inserida. Passar por referência.

`void destroyNode(Node* node)`

Função auxiliar da `destroy`.

- `node`: Ponteiro para o nó a ser destruído.

`void destroy(BinaryTree* tree)`

Libera toda a memória alocada pela árvore binária AVL.

- `tree`: Ponteiro para a árvore binária AVL a ser destruída.

2.5 Árvore Rubro-Negra (RBT)

rbt.h e rbt.cpp: arquivos usados para implementação de funções específicas à RBT.

2.5.1 Namespace RBT: funções

`BinaryTree* create()`

Inicializa uma árvore binária BRT e retorna-a.

`Node* rightRotate(Node* y, Node *NIL)` Realiza a rotação para a direita de um nó. Retorna o ponteiro para o novo nó na posição de `y`.

- `y`: Ponteiro para o nó a ser rotacionado.
- `NIL`: Ponteiro para o NIL.

`Node* leftRotate(Node* x, Node *NIL)`

Realiza a rotação para a esquerda de um nó. Retorna o ponteiro para o novo nó na posição de `x`.

- `x`: Ponteiro para o nó a ser rotacionado.
- `NIL`: Ponteiro para o NIL.

`void fixInsert(Node *z, Node *NIL)`

Corrige as cores e faz as rotações se necessário.

- `z`: Ponteiro para o nó a ser corrigido.
- `NIL`: Ponteiro para o NIL.

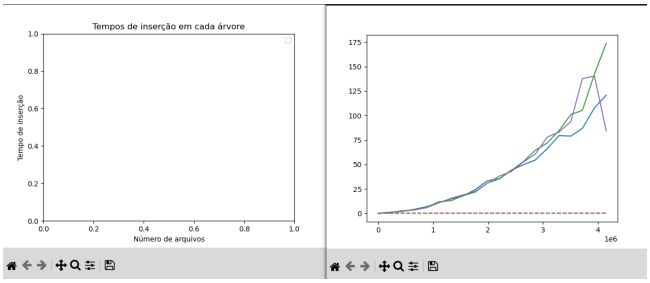

Figure 1: Enter Caption

`InsertResult insert(BinaryTree* tree, const std::string& word, int documentId)`

Insere uma palavra na árvore associada a um documentId. Se for inédita, cria um novo nó; se já existir, adiciona o documentId ao nó associado. Retorna um `InsertResult` com estatísticas sobre o desempenho.

- `tree`: Ponteiro para a árvore binária.
- `word`: Palavra a ser inserida. Passar por referência.
- `documentId`: Índice do documento que contém a palavra.

`SearchResult search(BinaryTree* tree, const std::string& word)`

Realiza a busca de uma palavra na árvore binária. Retorna um `SearchResult` que diz se a palavra foi achada, em quais documentos e estatísticas sobre o processo.

- `tree`: Ponteiro para a árvore binária.
- `word`: Palavra a ser inserida. Passar por referência.

`void destroyNode(Node* node)`

Função auxiliar da `destroy`.

- `node`: Ponteiro para o nó a ser destruído.

`void destroy(BinaryTree* tree)`

Libera toda a memória alocada pela árvore binária.

- `tree`: Ponteiro para a árvore binária a ser destruída.

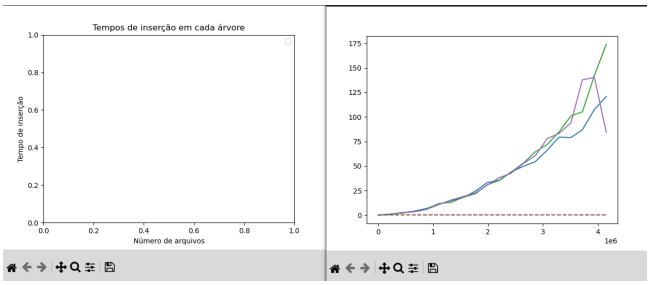

Figure 2: Enter Caption

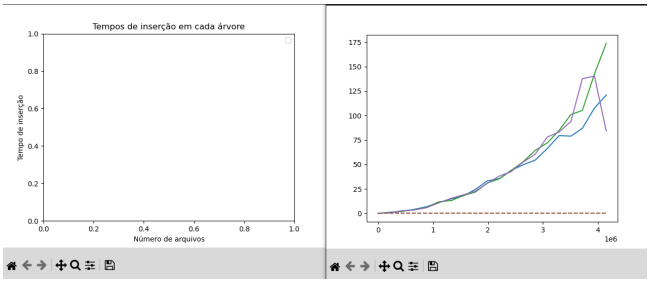

Figure 3: Enter Caption

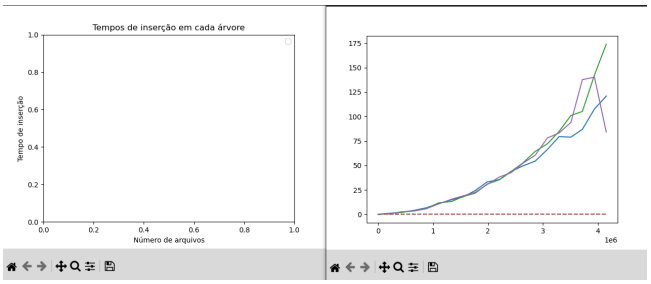

Figure 4: Enter Caption

3 Análise de Desempenho

4 Conclusão

De modo geral, embora o desempenho das diferentes árvores não apresente grandes variações, os dados indicam que a árvore AVL possui o melhor desempenho nas operações de busca, mas o pior nas inserções. A árvore Rubro-Negra (RBT) apresenta um desempenho intermediário, sendo a segunda melhor tanto em inserção quanto em busca. Por fim, a árvore binária de busca (BST) tem o melhor desempenho na inserção, mas o pior na busca.

5 Dificuldades

Durante o projeto, enfrentamos algumas dificuldades tanto na parte de programação quanto na organização do grupo.

Algumas funções, como a `printTree`, precisaram de adaptações e funções auxiliares para funcionar do jeito que a gente queria, sem mudar a interface principal. Além disso, implementar o balanceamento correto das árvores AVL e Rubro-Negra (RBT) deu um bom trabalho, principalmente para entender como funcionavam as rotações e garantir que tudo continuasse funcionando depois delas.

Na parte de organização, dividir as tarefas de forma justa entre todos e manter o repositório do GitHub atualizado foi mais difícil do que esperávamos. Em alguns momentos, tivemos problemas com conflitos de código, e foi necessário conversar bastante para resolver e alinhar as partes que cada um estava fazendo.

Também tivemos certa dificuldade em organizar os testes e garantir que todos os arquivos estivessem funcionando corretamente juntos. Às vezes, arrumar um erro em uma parte quebrava outra, o que exigiu bastante paciência e testes manuais.

Mesmo com essas dificuldades, conseguimos manter um bom ritmo e finalizar o projeto.[1]

6 Divisão de Tarefas

- **Antonio**: Responsável pelo desenvolvimento da interface de linha de comando (CLI) e dos arquivos `main_bst.cpp`, `main_avl.cpp` e `main_rbt.cpp`.
- **Bernardo**: Responsável pelo desenvolvimento dos arquivos `data.cpp`, `data.h`, `test_avl.cpp`, `test_rbt.cpp`, além de ter contribuído no arquivo `rbt.cpp`.
- **Dilmar**: Responsável pelo desenvolvimento dos arquivos `bst.cpp`, `bst.h`, `avl.cpp`, `avl.h`, `rbt.cpp`, `rbt.h` e das funções auxiliares no arquivo `tree_utils.cpp`.

[1] "What one programmer can do in one month, two programmers can do in two months." — *Folclore*

- **João Vitor**: Responsável pela estruturação do repositório no GitHub e pela elaboração do relatório. Desenvolveu os arquivos `tree_utils.h`, `tree_utils.cpp` e `graphics.py` (para geração de gráficos), além de documentar funções dos arquivos `tree_utils`, `bst` e `avl`.
- **Roger**: Responsável pelo desenvolvimento dos testes da BST, contribuições nos arquivos `main_bst.cpp` e `main_avl.cpp`, além de colaboração na elaboração deste relatório.