# Estrutura de Dados Avançadas

## 1. Introdução

O presente trabalho tem como objetivo implementar e comparar diferentes estruturas de dados (árvores AVL, árvores rubro-negras e tabelas hash) na tarefa de processamento de textos, com foco na contagem e análise de frequência de palavras. O sistema funciona como uma espécie de dicionário, onde cada item representa uma palavra (uma string), acompanhada de sua respectiva frequência (um número inteiro).

Para construir esse "dicionário", foi necessário, primeiramente, desenvolver uma forma de extrair as informações de um arquivo `entrada.txt` genérico, desconsiderando espaços em branco e sinais de pontuação. Em seguida, as estruturas de dados genéricos foram adaptadas para atender aos requisitos da aplicação, utilizando-se das operações fundamentais de cada componente. Por fim, definiu-se a forma de escrita da saída, gerada em um arquivo `saida.txt`, buscando-se minimizar o consumo de memória e a complexidade computacional das operações.

Para finalizar, uma análise sobre o desempenho de cada estrutura foi feita, com testes utilizando os diversos livros fornecidos na descrição do trabalho.

## 2. Descrição dos Arquivos

O programa foi modularizado nos seguintes componentes:

- `main.cpp` - Arquivo principal do projeto. Contém a função `main()` que inicializa o programa, faz a leitura da entrada, executa os comandos e coordena o uso das estruturas de dados.
- `comandos.cpp` - Pequeno descritivo sobre o processo de compilação, detalhando o fluxo de informação.
- `TextProcessor.h` - Realiza a normalização de strings, ou seja, prepara o texto para processamento eliminando ruídos como pontuação e letras maiúsculas.
- `AVLTree.h` - Implementa a estrutura de Árvore AVL, uma árvore binária de busca auto-balanceada.
- `RedBlackTree.h` - Contém a implementação de uma Árvore Rubro-Negra, outra forma de árvore balanceada.
- `openHash.h` - Implementa uma variação da tabela hash com hashing aberto (open addressing), técnica usada para resolver colisões sem listas encadeadas.
- `hash.h` - Implementar uma tabela hash com encadeamento externo (chaining).
- `entrada.txt` - Arquivo de texto usado como entrada para o programa. Contém o conteúdo a ser processado e analisado.
- `saida.txt` - Arquivo de saída gerado pelo programa. Contém o resultado da análise, como palavras e suas respectivas frequências.

## 3. Funcionamento do Programa

Fluxo básico:

1. Leitura do arquivo de entrada (`entrada.txt` ou `ptbr.srt`).
2. Processamento do texto (remoção de pontuação, conversão para minúsculas, separação de palavras).
3. Armazenamento em diferentes estruturas de dados.
4. Geração de saída (`saida.txt`) com resultados como frequência de palavras, tempo de execução e outras métricas no terminal.

## 4. Estruturas de Dados Utilizadas

Uma explicação mais detalhada de cada estrutura utilizada e os componentes da `main.cpp`.

**AVLTree.h**

**Estrutura da árvore:**

```
struct Node {
    K key;
    V value;
    Node *left, *right;
    int height;
    Node(const K& k, const V& v) : key(k), value(v), left(nullptr), right(nullptr), height(1) {}
};
Node* root;
int _size;
int comparisons;
int rotations;
```

A `Struct Node` representa cada elemento da árvore e armazena um par genérico (chave, valor), sem especificar o tipo das variáveis. Em seguida, temos as variáveis da classe: `root`, que aponta para a raiz da árvore; `_size`, que representa a quantidade de palavras armazenadas (tamanho da árvore); `comparisons`, que contabiliza o número de comparações feitas durante a construção da árvore; e `rotations`, que indica quantas rotações foram necessárias para manter a árvore balanceada.

A maioria das funções não foram alteradas de uma AVL genérica normal, mas precisou ter uma pequena alteração na função `Insert`, logo abaixo:

```
Node* insert(Node* node, const K& key, const V& value) {
    if (!node) {
        _size++;
        return new Node(key, value);
    }
    comparisons++;
    if (key < node->key)
        node->left = insert(node->left, key, value);
    else if (key > node->key)
        node->right = insert(node->right, key, value);
    else{
        node->value++;
        return node; // desnecessario na aplicação, na main já fazemos a verificação se existe ou nã
    }
    return balance(node);
}
```

A pequena alteração se encontra quando a palavra já foi inserida na árvore e precisamos apenas aumentar sua frequência. Em uma AVL normalmente ignoramos quando existe inserção de um nó que já existe na árvore, entretanto isso aumentaria o custo computacional do programa, já que precisamos atualizar manualmente o valor de `V value` (e antes verificar se a palavra já foi adicionada anteriormente). Com a adição de um simples "`node->value++;`" diminuímos complexidade no código da `main`:

Tratando as frequências manualmente:

```
while (inFile >> palavra) {
    palavra = TextProcessor::normalize(palavra);
    if (palavra.empty()) continue;

    if (dicionario.exists(palavra)) {
        int freq = dicionario.get(palavra);
        dicionario.update(palavra, freq + 1);
    } else {
        dicionario.insert(palavra, 1);
    }
}
```

Versão otimizada:

```
while (inFile >> palavra) {
    palavra = TextProcessor::normalize(palavra);
    if (palavra.empty()) continue;
    dicionario.insert(palavra, 1);
}
```

Para contar o número de rotações adicionamos "`rotations++;`", nas funções `Node* rotateLeft(Node* x)` e `Node* rotateRight(Node* x)`, assim contabilizamos de forma bem simples cada vez que acontece uma rotação na árvore.

Para escrever os dados da árvore em um `saida.txt`, realizamos três formas distintas para diversas aplicações: escrita direta no arquivo por meio de uma travessia em ordem na árvore, uma variante da primeira abordagem mas printando apenas palavras que começam com determinada letra, e a cópia dos valores de cada nó para um vetor auxiliar que pode ordenar e exibir a árvore de diversas formas. Vale lembrar que a inserção acontece respeitando o critério de `Key` (palavra), logo em uma travessia em ordem somente funcionará quando você pedir a ordem alfabética do conjunto, o que é mais convencional em um dicionário.

```
if (!node) return;
printInOrder(node->left, out);
out << node->key << " " << node->value << "\n";
printInOrder(node->right, out);
}
void printChar(Node* node, std::ostream& out, char initial) {
    if (!node) return;

    printChar(node->left, out, initial);

    if (!node->key.empty() && node->key[0] == initial)
        out << node->key << " " << node->value << "\n";

    printChar(node->right, out, initial);
}
void toVector(Node* node, std::vector<std::pair<K, V>>& vec) {
    if (!node) return;
    toVector(node->left, vec);
    vec.push_back({node->key, node->value});
    toVector(node->right, vec);
}
```

As demais funções são triviais em uma AVL genérica, não precisando de longa descrição, na verdade nem são usadas durante essa aplicação.

### RedBlackTree

```
template <typename K, typename V>
class RedBlackTree {
private:
    enum Color { RED, BLACK };

    struct Node {
        K key;
        V value;
        Color color;
        Node *left, *right, *parent;

        Node(const K& k, const V& v) : key(k), value(v), color(RED), left(nullptr), right(nullptr),
    };

    Node* root;
    int _size;
```

```
    int comparisons;
    int rotations;
```

A classe `RedBlackTree` implementa uma árvore rubro-negra genérica, utilizada para armazenar pares de chave e valor. Cada elemento da árvore é representado por um nó, definido pela `struct Node`, que contém a chave, o valor, a cor (vermelho ou preto), e ponteiros para os filhos esquerdo e direito, além do pai. A cor é um elemento fundamental para manter a árvore balanceada automaticamente, seguindo as regras das árvores rubro-negras.

A classe também possui variáveis que auxiliam no controle da estrutura, como `root`, que aponta para a raiz da árvore, `_size`, que armazena a quantidade de elementos inseridos (ou palavras, na aplicação), `comparisons`, que contabiliza o número de comparações realizadas durante as operações, e `rotations`, que indica quantas rotações foram necessárias para garantir o balanceamento da árvore.

As funções foram feitas com o mesmo escopo de uma AVL, para poderem dividir trecho de código na `main`, assim as funções `printChar`, `ToVector` e `printInOrder` seguem a mesma estrutura, como o uso das variáveis `comparisons` e `rotations`. Na inserção seguimos a mesma ideia de otimização da AVL também:

```
void _insert(const K& key, const V& value) {
    Node* pt = new Node(key, value);

    Node* y = nullptr;
    Node* x = root;

    while (x != nullptr) {
        y = x;
        comparisons++;
        if (pt->key < x->key)
            x = x->left;
        else if (pt->key > x->key)
            x = x->right;
        else {
            x->value++;
            return;
        }
    }
    _size++;

    pt->parent = y;
    if (y == nullptr)
        root = pt;
    else if (pt->key < y->key)
        y->left = pt;
    else
        y->right = pt;

    fixInsert(root, pt);
}
```

### HASH Com Encadeamento

Aqui usamos o exemplo dado em sala de aula, e customizamos para o trabalho:

```
using namespace std;

template <typename Key, typename Value, typename Hash = std::hash<Key>>
class ChainedHashTable {
private:
    size_t m_number_of_elements;
    size_t m_table_size;
    float m_max_load_factor;
```

```
    std::vector<std::list<std::pair<Key, Value>>> m_table;
    Hash m_hashing;
    int comparasons;
    int collisions = 0;
```

A classe `ChainedHashTable` implementa uma tabela hash genérica com tratamento de colisões por encadeamento externo. Ela é capaz de armazenar pares de chave e valor de qualquer tipo, utilizando um vetor de listas como estrutura principal. Cada posição do vetor, chamada de **bucket**, contém uma lista de pares, o que permite armazenar múltiplos elementos em um mesmo índice no caso de colisões.

A classe possui alguns atributos importantes para o controle da estrutura e análise de desempenho. O atributo `m_number_of_elements` armazena a quantidade total de elementos inseridos na tabela, enquanto `m_table_size` indica o número de buckets disponíveis. O fator de carga máximo é controlado por `m_max_load_factor`, que define o limite antes de se realizar um redimensionamento da tabela (processo conhecido como **rehash**).

A tabela utiliza a função de hash definida por `m_hashing` para transformar as chaves em índices. Além disso, a classe mantém um contador de comparações (`comparisons`), usado para monitorar a eficiência das operações, e um contador de colisões (`collisions`), que registra quantas vezes múltiplos elementos foram mapeados para o mesmo índice. A função `insert` foi devidamente adaptada para catalogar a quantidade de comparações e a frequência das palavras, semelhante aos exemplos anteriores:

```
void insert(const Key& k) {
    if (load_factor() >= m_max_load_factor) {
    collisions++;
        rehash(2 * m_table_size);
    }

    size_t slot = hash_code(k);

        for (auto& p : m_table[slot]) {
        comparasons++;
        if (p.first == k) {
            ++p.second;
            return;
        }

    }
    m_table[slot].push_back(std::make_pair(k, 1));
    ++m_number_of_elements;
}
```

Como a Hash não é inserida de forma naturalmente em ordem, como a AVL e a RBT, aqui a operação de escrita no arquivo e organização dos dados, é necessário de um vetor auxiliar. Diferente das árvores, que conseguimos juntar todas em um escopo só e criar na `main` uma função `Write()`, aqui precisamos encapsular dentro da própria classe esse método.

```
void write(ChainedHashTable<Key, Value>& dicionario, const std::string& arquivoSaida) {
    std::vector<std::pair<Key, Value>> vec;
    cout << "Deseja imprimir o dicionário completo ou palavras que começam com uma letra específica?
    cout << "Digite 'completo', 'freq' ou uma letra (ex: a): ";
    string escolha;
    cin >> escolha;
    std::ofstream outFile(arquivoSaida);

    if (escolha == "completo") {
        for (size_t i = 0; i < dicionario.bucket_count(); ++i) {
        auto& bucket = dicionario.bucket(i);
        for (const auto& p : bucket) {
            vec.push_back(p);
        }
    }
```

```
    std::sort(vec.begin(), vec.end(), [](const auto& a, const auto& b) {
    return a.first < b.first;  // ordena pela chave (string) em ordem crescente (alfabetica)
    });
    for (const auto& p : vec) {
        outFile << p.first << ": " << p.second << "\n";
    }
    } else if (escolha.size() == 1 && isalpha(escolha[0])) {
        for (size_t i = 0; i < dicionario.bucket_count(); ++i) {
        auto& bucket = dicionario.bucket(i);
        for (const auto& p : bucket) {
        if (!p.first.empty() && std::tolower(p.first[0]) == std::tolower(escolha[0]))
            vec.push_back(p);
        }
    }

    std::sort(vec.begin(), vec.end(), [](const auto& a, const auto& b) {
        return a.second > b.second;
    });
    for (const auto& p : vec) {
        outFile << p.first << ": " << p.second << "\n";
    }
    } else if(escolha == "freq") {

    for (size_t i = 0; i < dicionario.bucket_count(); ++i) {
        auto& bucket = dicionario.bucket(i);
        for (const auto& p : bucket) {
            vec.push_back(p);
        }
    }

    std::sort(vec.begin(), vec.end(), [](const auto& a, const auto& b) {
        return a.second > b.second;
    });
    for (const auto& p : vec) {
        outFile << p.first << ": " << p.second << "\n";
    }
    }
}
```

O método `write` é responsável por gerar um arquivo de saída contendo o conteúdo da tabela hash. Ele oferece três opções ao usuário: imprimir o dicionário completo ordenado alfabeticamente, listar apenas palavras que começam com uma letra específica, ou exibir as palavras em ordem decrescente de frequência. Após coletar os dados da tabela de acordo com a escolha, o método os organiza e escreve no arquivo especificado.

### OpenHash

```
template <typename Key, typename Value>
class HashOpen {
private:
    enum EntryState { EMPTY, OCCUPIED, DELETED };

    struct Entry {
        Key key;
        Value value;
        EntryState state;
```

```
        Entry() : state(EMPTY) {}
        Entry(const Key& k, const Value& v) : key(k), value(v), state(OCCUPIED) {}
    };

    std::vector<Entry> table;
    size_t m_size;
    size_t m_capacity;
    int  comparisons;
```

A classe `HashOpen` implementa uma tabela hash utilizando o método de endereçamento aberto para lidar com colisões. Nessa abordagem, todos os elementos são armazenados diretamente em um vetor fixo de entradas, sem uso de listas auxiliares. Cada entrada é representada por uma estrutura (`Entry`) que contém a chave, o valor associado e um estado, que indica se a posição está vazia, ocupada ou foi deletada.

Os principais atributos dessa classe incluem o vetor `table`, que armazena os dados, `m_size`, que indica a quantidade de elementos inseridos, e `m_capacity`, que representa a capacidade total da tabela. Há ainda o contador `comparisons`, que registra quantas comparações foram feitas durante as operações, servindo como métrica de desempenho.

Para variar um pouco a abordagem, resolvemos fazer a leitura e escrita da última classe inteira na `main.cpp`, lendo um arquivo de texto, normaliza as palavras (removendo pontuações e convertendo para minúsculas), e armazena a frequência de cada palavra em uma tabela hash com endereçamento aberto. Durante o processo, se contabiliza comparações e mede o tempo de execução. Após a leitura, todas as palavras são extraídas da tabela, ordenadas em ordem alfabética e escritas em um arquivo de saída.

```
HashOpen<string, int> dicionario;
string palavra;
auto inicio = chrono::high_resolution_clock::now();

while (inFile >> palavra) {
    palavra = TextProcessor::normalize(palavra);
    if (palavra.empty()) continue;

    int atual;
    if (dicionario.find(palavra, atual)) {
        dicionario.insert(palavra, atual + 1);
    } else {
        dicionario.insert(palavra, 1);
    }
}

auto fim = chrono::high_resolution_clock::now();
chrono::duration<double> duracao = fim - inicio;

auto vec = dicionario.getAllWords();

std::sort(vec.begin(), vec.end(), [](const auto& a, const auto& b) {
    return a.first < b.first;  // ordena alfabeticamente pela palavra
});

std::ofstream outFile(arquivoSaida);
for (const auto& p : vec) {
    outFile << p.first << " " << p.second << "\n";
}

std::cout << "Palavras: " << dicionario.size() << std::endl;
std::cout << "Comparações: " << dicionario.getComparisons() << std::endl;
std::cout << "Tempo: " << duracao.count() << " segundos" << std::endl;
```

**Main.cpp**

Na `main`, tentamos diversas abordagens de como interagir com estruturas. Primeiramente, pegamos os argumentos enviados no terminal, verificamos se estão corretos, se o arquivo de entrada é válido e, a partir daí, iniciamos o processamento:

```
cerr << "Uso: ./freq dictionary <estrutura> <entrada.txt> [saida.txt]" << endl;
return 1;
}
string estrutura = argv[2];
string arquivoEntrada = argv[3];
string arquivoSaida = (argc == 5) ? argv[4] : "saida.txt";
ifstream inFile(arquivoEntrada);
if (!inFile.is_open()) {
    cerr << "Erro ao abrir o arquivo de entrada." << endl;
    return 1;
}
```

**Artifícios Gerais:** Para medir o tempo utilizamos `high_resolution_clock` da biblioteca `<chrono>`, que é usada em C++ para capturar o momento atual com a maior resolução possível do relógio do sistema. Medindo o início e o final de cada operação para pegar o tempo total:

```
auto inicio = chrono::high_resolution_clock::now();
auto fim = chrono::high_resolution_clock::now()
chrono::duration<double> duracao = fim - inicio;
```

Para normalizar as palavras utilizamos um mapa que faz a conversão de palavras com acento para palavras normais, aceitas no c++. Isso pode gerar alguns problemas como “maca” e “maçã” se tornarem a mesma palavra, mas é a forma mais simples de implementar. Também converte tudo para minúsculo e desconsidera outros símbolos que não sejam letras.

```
class TextProcessor {
public:
    static std::string normalize(const std::string& input) {

        static const std::unordered_map<char, char> accent_map = {
            {'á', 'a'}, {'à', 'a'}, {'ã', 'a'}, {'â', 'a'}, {'ä', 'a'},
            {'é', 'e'}, {'è', 'e'}, {'ê', 'e'}, {'ë', 'e'},
            {'í', 'i'}, {'ì', 'i'}, {'î', 'i'}, {'ï', 'i'},
            {'ó', 'o'}, {'ò', 'o'}, {'õ', 'o'}, {'ô', 'o'}, {'ö', 'o'},
            {'ú', 'u'}, {'ù', 'u'}, {'û', 'u'}, {'ü', 'u'},
            {'ç', 'c'},
            {'Á', 'a'}, {'À', 'a'}, {'Ã', 'a'}, {'Â', 'a'}, {'Ä', 'a'},
            {'É', 'e'}, {'È', 'e'}, {'Ê', 'e'}, {'Ë', 'e'},
            {'Í', 'i'}, {'Ì', 'i'}, {'Î', 'i'}, {'Ï', 'i'},
            {'Ő', 'o'}, {'Ò', 'o'}, {'Õ', 'o'}, {'Ô', 'o'}, {'Ö', 'o'},
            {'Ú', 'u'}, {'Ù', 'u'}, {'Û', 'u'}, {'Ü', 'u'},
            {'Ç', 'c'}
        };

        std::string result;
        for (unsigned char c : input) {
            // se tem no mapa, substitui
            if (accent_map.count(c)) {
                result += accent_map.at(c);
            }
            else if (std::isalnum(c) || c == '-') {
                result += std::tolower(c);
            }
            // ignora outros caracteres
        }
```

```
        return result;
    }
};
```

**IF AVL ou RBT** Se for o comando para uma árvore, o fluxo de execução é basicamente o mesmo, só muda a estrutura iniciada. Já mostramos como são inseridas as palavras, agora vamos detalhar como é a escrita na saída:

```
void write(Tree& dicionario, const string& arquivoSaida) {
    ofstream outFile(arquivoSaida);
    if (!outFile.is_open()) {
        cerr << "Erro ao abrir o arquivo de saída." << endl;
        return;
    }

    cout << "Deseja imprimir o dicionário completo ou palavras que começam com uma letra específica?
    cout << "Digite 'completo', 'freq' ou uma letra (ex: a): ";
    string escolha;
    cin >> escolha;
    auto inicio = std::chrono::high_resolution_clock::now();
    if (escolha == "completo") {
        dicionario.printInOrder(outFile);
    } else if (escolha.size() == 1 && isalpha(escolha[0])) {
        char letra = tolower(escolha[0]);
        outFile << "Palavras que começam com '" << letra << "':\n";
        dicionario.printChar(letra, outFile);
    } else if(escolha == "freq") {
        std::vector<std::pair<std::string, int>> vec;
        dicionario.toVector(vec);
        sort(vec.begin(), vec.end(),
            [](const pair<string, int>& a, const pair<string, int>& b) {
                return a.second > b.second;
            });

        outFile << "Palavras ordenadas por frequência:\n";
        for (const auto& p : vec) {
            outFile << p.first << ": " << p.second << "\n";
        }
    }
    else {
        std::cerr << "Opção inválida." << std::endl;
    }

    auto fim = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duracao = fim - inicio;
    cout << "Tempo para gerar a saída: " << duracao.count() << " segundos" << endl;
}
```

Dependendo da escolha do usuário, apenas chamamos os métodos já mostrados anteriormente: `printChar` e `printInOrder`, que aproveitam da ordenação natural no percurso em ordem e realizam a escrita sem o auxílio de outra estrutura. Mas a frequência torna inviável não usar um vetor auxiliar; como explicado anteriormente, esse vetor é preenchido por `Tovector`, ordenado em função da frequência (`Value`) e escrito linha a linha na saída.

## 0.1 Comparação de Desempenho

Por diversos conflitos quando eu tentava colocar as tabelas aqui, optei por fazer um doc para esse tópico com os dados obtidos :

https://docs.google.com/document/d/1GlNzfRa-by3BmRTmi8K6MUJwe4VKaRpc1hWOPt4RAoY/edit?usp=sharin

## 0.2 Considerações Finais

A Tabela Hash com Encadeamento é a estrutura que apresentou melhor desempenho geral, com menos comparações e poucas colisões, sendo ideal para buscas rápidas em conjuntos de dados grandes e estáveis.

A Árvore Rubro-Negra mostrou-se eficiente no balanceamento automático, realizando menos rotações que a AVL e mantendo tempos competitivos, o que a torna uma boa escolha para cenários onde os dados são frequentemente modificados.

A Árvore AVL apresentou desempenho similar à RBT, porém com maior número de rotações.

A Tabela Hash com Endereçamento Aberto precisa de ajustes para apresentar métricas completas, embora seu tempo de execução esteja em linha com as outras tabelas hash.

Em resumo, para buscas rápidas e estáticas, opte pela tabela hash com encadeamento. Para aplicações com muitas modificações e necessidade de dados ordenados, a Árvore Rubro-Negra é recomendada.