Relatório do Projeto Final EDA

Linyker Vinícius Gomes Barbosa - 556280

30 de agosto de 2024

1 Introdução

O projeto tem como objetivo realizar a contagem de frequência das palavras contidas em um arquivo de texto, a fim de avaliar o desempenho de quatro diferentes estruturas de dados: Árvore AVL, Árvore Rubro-Negra, Tabela Hash com endereçamento aberto e Tabela Hash com encadeamento exterior.

Para realizar a contagem de frequência, o texto será processado de maneira a ignorar espaços em branco e sinais de pontuação, desconsiderando a diferença entre letras maiúsculas e minúsculas. O programa gera uma tabela de frequências das palavras, organizando-as em ordem alfabética. Além da contagem de frequência, o programa realiza a contabilização do número de comparações de chaves necessárias para a inserção e busca em cada estrutura, bem como o tempo total de processamento para construir a tabela de frequências, permitindo a análise aprofundada do desempenho de cada estrutura de dados.

2 Estruturas de dados

2.1 AVLTree.h

A Árvore AVL é uma árvore binária de busca balanceada que mantém o balanceamento através de rotações, garantindo que a altura da árvore seja mantida em $O(\log n)$, onde n é o número de nós. No contexto deste projeto, foi implementada para armazenar os nós como pares de chave/valor, sendo a chave a palavra, acompanhada da contagem de sua frequência como valor.

2.2 RBTree.h

A Árvore Rubro-Negra é outra árvore binária de busca balanceada, mas com regras de balanceamento diferentes da árvore AVL. Sua principal característica é a distinção entre nós vermelhos e pretos, o que permite uma

altura $O(\log n)$ de forma mais permissiva em relação ao balanceamento. Assim como na AVL, as palavras foram inseridas como chaves e as frequências como valores. A Árvore Rubro-Negra também é conhecida por exigir menos rotações do que a AVL, podendo se rebalancear em no máximo 3 rotações, resultando em um desempenho superior em certos cenários.

2.3 Hash2.h

A Tabela Hash com endereçamento aberto é uma técnica de resolução de colisões em que, ao ocorrer uma colisão (ou seja, quando duas chaves diferentes mapeiam para o mesmo índice da tabela), a próxima posição disponível na tabela é utilizada. O tratamento de colisões por endereçamento aberto torna a inserção e busca eficientes, especialmente quando a tabela não está muito cheia, e utiliza menos memória, pois ocupa apenas o tamanho do vetor de hash. No projeto, essa estrutura foi utilizada para armazenar as palavras e suas frequências, resolvendo colisões através da técnica de sondagem linear.

2.4 Hash.h

A Tabela Hash com encadeamento exterior é outra técnica para tratar colisões. Nesse caso, cada índice da tabela aponta para uma lista encadeada que armazena todas as chaves que colidiram naquele índice. Essa abordagem é interessante quando a tabela contém muitas colisões, pois mantém a eficiência de inserção e busca sem a necessidade de sondagem. No projeto, foi utilizada para armazenar as palavras e frequências, com listas encadeadas.

3 Implementação

3.1 Dict

O arquivo `Dict.h` define uma classe template chamada `Dict`, que pode ser parametrizada para utilizar as estruturas de dados `AVLTree`, `RBTree`, `HashTable` e `Hash2Table`. A classe encapsula as funcionalidades dessas estruturas para manipulação de pares chave-valor, oferecendo operações como inserção (`add`), remoção (`remove`), busca (`find`), atualização (`update`) e impressão (`print`).

Além disso, a classe inclui métodos para verificar a existência de uma chave (`contains`), limpar a estrutura de dados (`clear`), obter o tamanho (`size`) e contabilizar o número de comparações realizadas (`comparisons`). Essa implementação generalizada permite a troca flexível entre diferentes

estruturas de dados, tornando o gerenciamento de dicionários mais eficiente e adaptável a diferentes cenários de uso.

```
template <typename EDType>
class Dict
{
private:
    EDType _dict;

public:
    void add(icu::UnicodeString key, unsigned int value = 1)
    {
        try
        {
            _dict[key] += value;
        }
        catch (std::out_of_range &e)
        {
            _dict.insert(key, value);
        }
    }

    void remove(icu::UnicodeString key)
    {
        try
        {
            _dict[key] -= 1;
            if (_dict[key] <= 0)
                _dict.remove(key);
        }
        catch (std::out_of_range &e)
        {
            std::cerr << "Key not found" << std::endl;
        }
    }

    void update(icu::UnicodeString key, unsigned int value)
    {
        _dict.update(key, value);
    }
```

```
    int find(icu::UnicodeString key)
    {
        return _dict.find(key);
    }

    void clear()
    {
        _dict.clear();
    }

    bool contains(icu::UnicodeString key)
    {
        return _dict.contains(key);
    }

    size_t size()
    {
        return _dict.size();
    }

    size_t comparisons()
    {
        return _dict.comparisons();
    }

    void print()
    {
        _dict.print();
    }
};
```

- **add**: Insere a palavra no dicionário, caso ela já exista apenas atualiza a frequencia dela.
- **remove**: diminui uma frequencia da palavra, caso chegue a zero remove a palavra do dicionário
- **update**: atualiza a frequencia da palavra
- **contains**: verifica se a palavra existe no dicionário
- **find**: retorna a frequencia da palavra

- **size**: retorna a quantia de palavras unicas no dicionário
- **clear**: deleta todas as palavras do dicionnário
- **comparisons**: retorna a quantia de comparações feitas no dicionário
- **print**: mostra todas as palavras com suas frequancias

4 Testes Realizados

Para o arquivo `biblia_sagrada_english.txt`, foram realizados testes utilizando quatro estruturas de dados distintas: Árvore AVL, Árvore Rubro-Negra, Tabela Hash de Endereçamento Aberto e Tabela Hash de Encadeamento Externo. Os resultados obtidos são apresentados abaixo:

4.1 biblia_sagrada_english.txt

- Árvore AVL
  - Número de palavras - 16.388
  - Tempo total de execução: 719ms.
  - Número de comparações: 16.373.306
- Árvore Rubro-negra
  - Número de palavras - 16.388
  - Tempo total de execução: 693ms.
  - Número de comparações: 15.428.509
- Tabela Hash de Endereçamento Aberto
  - Número de palavras - 16.388
  - Tempo total de execução: 467ms.
  - Número de comparações: 977.375
- Tabela Hash de Encadeamento Exterior
  - Número de palavras - 16.388
  - Tempo total de execução: 434ms.
  - Número de comparações: 1.274.778

4.2 memorias_postumas_de_braz_cubas.txt

- Árvore AVL
  - Número de palavras - 11.398
  - Tempo total de execução: 98ms.
  - Número de comparações: 1.414.445
- Árvore Rubro-negra
  - Número de palavras - 11.398
  - Tempo total de execução: 90ms.
  - Número de comparações: 1.454.805
- Tabela Hash de Endereçamento Aberto
  - Número de palavras - 11.398
  - Tempo total de execução: 86ms.
  - Número de comparações: 53.443
- Tabela Hash de Encadeamento Exterior
  - Número de palavras - 11.398
  - Tempo total de execução: 63ms.
  - Número de comparações: 82.581

4.3 o_manifesto_comunista_english.txt

- Árvore AVL
  - Número de palavras - 2.628
  - Tempo total de execução: 36ms.
  - Número de comparações: 266.936
- Árvore Rubro-negra
  - Número de palavras - 2.628
  - Tempo total de execução: 22ms.
  - Número de comparações: 252.115
- Tabela Hash de Endereçamento Aberto

  - Número de palavras - 2.628
  - Tempo total de execução: 9ms.
  - Número de comparações: 11.897

- Tabela Hash de Encadeamento Exterior
  - Número de palavras - 2.628
  - Tempo total de execução: 12ms.
  - Número de comparações: 18.851

5 Dificuldades Encontradas

O processamento de strings foi um desafio significativo. Para a realização de comparações com strings acentuadas, foi necessário o uso de uma biblioteca externa chamada Unicode.

6 Conclusão

O uso da biblioteca Unicode foi essencial para garantir a correta manipulação de strings com caracteres acentuados, o que adicionou uma camada extra de complexidade ao projeto, mas foi fundamental para o processamento correto dos dados.

Este trabalho permitiu uma análise aprofundada das estruturas de dados utilizadas, destacando as vantagens e desvantagens de cada uma em diferentes cenários. Com base nos resultados obtidos, as Tabelas Hash, especialmente com encadeamento exterior, podem ser consideradas as mais eficientes para a tarefa proposta, enquanto as árvores balanceadas são opções interessantes quando a ordem dos elementos e a manutenção do balanceamento são fatores cruciais.

References

[1] GeeksforGeeks, *AVL Tree - Data Structures*. Disponível em: `https://www.geeksforgeeks.org/data-structures/#6-avl-tree`.

[2] GeeksforGeeks, *Red-Black Tree - Data Structures*. Disponível em: `https://www.geeksforgeeks.org/data-structures/#9-redblack-tree`.

[3] University Academy, *Data Structures and Algorithms Video*. YouTube, 2021. Disponível em: https://www.youtube.com/watch?v=26Lf1t0Rywc.

[4] Unicode Documentation, *Unicode Library for C++*. Disponível em: https://unicode-org.github.io/icu/.