Universidade Federal do Ceará - Campus Quixadá
Estrutura de Dados Avançada
Prof. Atílio Gomes Luiz

Kauan Pablo de Sousa Silva - 556027

17 de setembro, 2024

Contagem de Frequências Usando Estruturas de Dados

Repositório do Projeto no GitHub, contém o código completo da implementação, uma descrição detalhada dos requisitos do trabalho, e a documentação com instruções passo a passo para compilar e executar o projeto corretamente.

01. Introdução

O objetivo deste trabalho é o desenvolvimento de uma aplicação em C++ para calcular a frequência de palavras em um texto e imprimir essas palavras em ordem alfabética, com suas respectivas frequências. O projeto deve receber um arquivo de texto, converter todas as letras para minúsculas, remover caracteres especiais, e em seguida utilizar uma das estruturas de dados implementadas para armazenar as palavras e contar suas frequências.

As estruturas de dados implementadas incluem duas árvores (AVL e Rubro-Negra) e duas tabelas hash (encadeamento externo e endereçamento aberto). Também é necessário fazer uma análise de desempenho entre essas estruturas de dados com base no tempo de execução e no número de comparações realizadas.

02. Estruturas de Dados Implementadas

As estruturas de dados são do tipo genérico e podem ser utilizadas para armazenar pares de chaves e valores. As principais operações usadas no projeto seguem um padrão comum entre as estruturas de dados implementadas, entre elas estão:

- `insert(key, value)`: insere um par chave-valor na estrutura de dados.
- `remove(key)`: remove um par chave-valor da estrutura de dados. (Lança uma exceção se a chave não for encontrada)
- `search(key)`: retorna uma referência ao valor associado a uma chave na estrutura de dados. (Lança uma exceção se a chave não for encontrada)
- `att(key, value)`: atualiza o valor associado a uma chave na estrutura de dados. (Lança uma exceção se a chave não for encontrada)
- `size()`: retorna o número de pares chave-valor na estrutura de dados.
- `empty()`: retorna verdadeiro se a estrutura de dados estiver vazia, falso caso contrário.
- `begin()`: retorna um iterador para o primeiro par chave-valor na estrutura de dados.
- `end()`: retorna um iterador para o último par chave-valor na estrutura de dados.

O iterador das estruturas percorre os elementos em ordem crescente de acordo com o valor da chave.

02.1 Árvore AVL

Árvore AVL é um árvore binária de busca balanceada onde, para cada nó, a diferença entre as alturas de suas subárvores esquerda e direita é no máximo 1. Esse balanceio permite que as operações de busca, inserção e remoção de elementos tenham complexidade de tempo de $O(\log n)$.

02.2 Árvore Rubro-Negra

Árvore Rubro-Negra é uma árvore binária de busca balanceada que segue cinco propriedades:

1. Cada nó é vermelho ou preto.
2. A raiz é preta.
3. Cada folha (NIL) é preta.
4. Se um nó é vermelho, então seus filhos são pretos.
5. Para cada nó, todos os caminhos de um nó para folhas descendentes contêm o mesmo número de nós pretos.

Essas propriedades garantem que a árvore esteja balanceada e que as operações de busca, inserção e remoção de elementos tenham complexidade de tempo de $O(\log n)$. A inserção e remoção de elementos em uma árvore Rubro-Negra costumam ser mais eficientes que em uma árvore AVL, pois a árvore Rubro-Negra é menos rígida em relação ao balanceamento necessitando de menos rotações para manter suas propriedades. Em relação a busca, a árvore AVL é mais eficiente para procurar chaves por manter um balanceamento mais rígido, mesmo que a complexidade também seja $O(\log n)$.

02.3 Tabela Hash com Encadeamento Externo

A Tabela Hash com Encadeamento Externo utiliza uma função de hash para mapear chaves em índices de uma tabela. Quando acontece uma colisão (duas chaves são mapeadas para o mesmo lugar) a tabela armazena em uma lista que contém todos os pares chave-valor que colidiram naquele índice. As operações de inserção, busca e remoção têm complexidade de tempo de $O(1)$ em média, mas podem chegar a $O(n)$ no pior caso, quanto melhor a função de hash, menor a chance de colisões ocorrerem e maior é a eficiência da tabela. O iterador da tabela percorre uma cópia dos elementos que estão em um vector ordenado.

02.4 Tabela Hash com Endereçamento Aberto (Hashing Duplo)

A Tabela Hash com Endereçamento Aberto implementada utiliza a estratégia de Hashing Duplo para resolver colisões. A função hash é calculada utilizando duas funções de hash, uma principal e outra secundária, quando ocorre uma colisão um deslocamento é feito ao chamar a função de hash com o número de colisões ocorridas até encontrar um índice vazio. As operações de inserção, busca e remoção têm complexidade de tempo de $O(1)$ em média, mas podem chegar a $O(n)$ no pior caso. Apesar de ser um pouco mais complexa de implementar em relação a tabela com encadeamento externo, a tabela com endereçamento aberto utiliza menos memória pelo fato de não precisar armazenar ponteiros para listas encadeadas. O iterador da tabela percorre uma cópia dos elementos que estão em um vector ordenado.

03. Sistema Implementado

A classe `dictionary` gerencia quatro estruturas de dados e utiliza um tipo genérico `type`, que deve assumir uma das quatro estruturas implementadas. A seguir estão os principais métodos da classe.

- `void insert(const icu::UnicodeString& word);`

Insere uma palavra no dicionário. A função tenta atribuir à variável `auto` o valor retornado por `search(key)`. Se a chave já existir, o valor é incrementado para refletir uma nova ocorrência da palavra. Caso o `search` lance uma exceção (indicando que a chave não existe), um novo par (key, 1) é inserido na estrutura de dados.

- `void insert_text(icu::UnicodeString text);`

  Recebe um texto (idealmente já formatado para o trabalho) e o divide em palavras, inserindo cada uma no dicionário usando o método `insert`. Palavras separadas por hífen são consideradas como uma só.

- `void remove(const icu::UnicodeString& word);`

  Remove uma palavra do dicionário. (Lança uma exceção se a palavra não for encontrada)

- `void clear();`

  Limpa o dicionário, removendo todas as palavras e suas respectivas frequências.

- `unsigned int size();`

  Retorna a quantidade de palavras diferentes presentes no dicionário.

- `bool empty();`

  Verifica se o dicionário está vazio, retornando `true` se não contiver palavras e `false` caso contrário.

- `bool contains(const icu::UnicodeString& word);`

  Verifica se uma palavra está presente no dicionário.

- `void att(const icu::UnicodeString& word, int att);`

  Atualiza a frequência de uma palavra no dicionário. (Lança uma exceção se a palavra não for encontrada)

- `int search(const icu::UnicodeString& word);`

  Retorna a frequência de uma palavra no dicionário. (Lança uma exceção se a palavra não for encontrada)

- `icu::UnicodeString list();`

  Utiliza o iterador das estruturas de dados para percorrer as estruturas e adicionar cada par de chave-valor a uma variável do tipo `icu::UnicodeString`. As palavras e suas frequências são listadas em ordem crescente, conforme implementado pelos iteradores das quatro estruturas de dados.

- `void print();`

  Imprime o dicionário, mostrando palavras e suas frequências no console.

- `void save(const std::string& filename, std::chrono::milliseconds duration);`

  Salva o conteúdo do dicionário em um arquivo, incluindo o tamanho do dicionário, o número de comparações realizadas e o tempo gasto para montar a tabela.

- `unsigned int comparisons();`

  Retorna o número de comparações realizadas durante as operações no dicionário.

Agora aqui está um resumo de como foi a execução do projeto, após executar o programa, o arquivo de texto e a estrutura de dado escolhida é passada pelos argumentos do terminal, o programa cria um dicionário utilizando de template a estrutura escolhida, e logo após chama uma função auxiliar pra executar as operações.

```
template <typename dict_type>
void process_and_save_dict(dict_type &dict, const string &filename,
                           const string &mode_structure) {
    // Processa o arquivo, removendo caracteres especiais e convertendo para minúsculas.
    UnicodeString file = read_file("in/" + filename);

    // Inicia a contagem do tempo e insere as palavras no dicionário
    auto start = high_resolution_clock::now();
    dict.insert_text(file);

    // Finaliza a contagem do tempo e calcula a duração
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(stop - start);

    // Printa algumas informações sobre o dicionário
    cout << "Tempo de execução: " << duration.count() << "ms" << endl;
    // cout << dict.comparisons() << " comparações" << endl;

    // Salva o dicionário no arquivo
    string out_filename = "out/" + mode_structure + "_" + filename;
    dict.save(out_filename, duration);
}
```

04. Dificuldades para Implementação

- **Manipulação de Strings Unicode**: A principal dificuldade encontrada durante o projeto foi a manipulação de strings Unicode. A biblioteca ICU, que contém a `ICU::UnicodeString`, foi utilizada para lidar com as strings, a própria documentação não é muito clara mas em compensação as funções da biblioteca em si são bem fáceis de usar, a parte mais complexa foi aprender a utilizar o `ICU::Collator` para comparar as strings em ordem lexicográfica, pois a biblioteca não possui uma função de comparação direta para esse caso, sendo necessário usar um objeto `Collator` dentro de uma função de comparação customizada.
- **Hashing Duplo**: A implementação da tabela hash com endereçamento aberto utilizando a técnica de hashing duplo foi a mais complexa de implementar, pois o cálculo da nova posição de uma chave quando a tabela vai ser redimensionada é um pouco mais complexo que o cálculo de uma função de hash comum, sendo necessário calcular a nova posição da chave a partir da posição original e do número de colisões que ocorreram até encontrar um índice vazio.

05. Testes Realizados

O programa foi testado com diferentes arquivos de texto para verificar o desempenho das estruturas de dados implementadas, dentre os teste realizados estão:

05.1 `biblia_sagrada_english.txt` (5.649.985 caracteres)

- **Árvore AVL:**
  - Tamanho do dicionário: 16388

  - Número de comparações: 31752346
  - Tempo de execução: 754 ms

- **Árvore Rubro-Negra:**
  - Tamanho do dicionário: 16388
  - Número de comparações: 22929095
  - Tempo de execução: 498 ms

- **Tabela Hash (Encadeamento Externo):**
  - Tamanho do dicionário: 16388
  - Número de comparações: 2583772
  - Tempo de execução: 176 ms

- **Tabela Hash (Endereçamento Aberto):**
  - Tamanho do dicionário: 16388
  - Número de comparações: 2876367
  - Tempo de execução: 182 ms

05.2 `a_riqueza_das_nacoes_english.txt` (2.467.546 caracteres)

- **Árvore AVL:**
  - Tamanho do dicionário: 10038
  - Número de comparações: 12102502
  - Tempo de execução: 295 ms

- **Árvore Rubro-Negra:**
  - Tamanho do dicionário: 10038
  - Número de comparações: 8439015
  - Tempo de execução: 196 ms

- **Tabela Hash (Encadeamento Externo):**
  - Tamanho do dicionário: 10038
  - Número de comparações: 1062040
  - Tempo de execução: 82 ms

- **Tabela Hash (Endereçamento Aberto):**
  - Tamanho do dicionário: 10038
  - Número de comparações: 1100226
  - Tempo de execução: 80 ms

05.3 `memorias_postumas_de_braz_cubas.txt` (387.816 caracteres)

- **Árvore AVL:**
  - Tamanho do dicionário: 11398
  - Número de comparações: 2984820
  - Tempo de execução: 82 ms
- **Árvore Rubro-Negra:**
  - Tamanho do dicionário: 11398
  - Número de comparações: 1948424
  - Tempo de execução: 57 ms
- **Tabela Hash (Encadeamento Externo):**
  - Tamanho do dicionário: 11398
  - Número de comparações: 270190
  - Tempo de execução: 31 ms
- **Tabela Hash (Endereçamento Aberto):**
  - Tamanho do dicionário: 11398
  - Número de comparações: 274623
  - Tempo de execução: 32 ms

06. Conclusão

Este trabalho permitiu fazer uma ánalise bem detalhada sobre o desempenho de diferentes estruturas de dados e suas vantagens ou desvantagens em relação ao trabalho, além de aprender a utilizar a biblioteca ICU para manipulação de strings Unicode.

A árvore avl fez em média 1,45 vezes mais comparações em relação a rubro negra e executou em média 1,48 vezes mais devagar, isso se deve as operações de inserção e remoção em uma árvore avl que são operações mais custosas em relação a rubro negra, essa diferença seria ainda maior se não fosse considerado o tempo de pesquisa já que a árvore avl é um pouco mais eficiente que a rubro negra para pesquisar chaves(apesar de ambas terem complexidade de tempo de $O(\log n)$).

As tabelas hashs foram as mais eficientes para a implementação do trabalho, as duas tiveram tempos de execução e número de comparações muito próximos que variavam de acordo com o ambiente de execução, então a escolha entre uma ou outra depende mais do contexto, no geral escolheria a tabela hash com endereçamento abertp por utilizar menos mémoria ao não precisar armazenar ponteiros para listas encadeadas.

Em relação a rubro negra, as tabelas hash fizeram cerca de 8 vezes menos comparações e foram em média 2,3 vezes mais rápidas, entretanto o tempo de ordenação das chaves utilizada no iterator das tabelas hash não é considerado, então a diferença seria um pouco menor, mas mesmo assim no maior caso de teste (biblia_sagrada_english.txt) as tabelas hash aumentaram seu tempo de execução em aproximadamente apenas 60 ms, o que ainda é muito mais rápido que a rubro negra.

Em resumo, para projetos onde a inserção e remoção são mais importantes que a iteração ordenada das chaves, as tabelas hash são a melhor escolha, mas se a iteração ordenada das chaves for mais importante e for feita rotineiramente, a árvore rubro negra é a melhor escolha.

07. Referências

- Cplusplus.com. *C++ STL Iterators*. Disponível em: `https://www.cplusplus.com/reference/iterator/`.

- Unicode Consortium. *CharacterIterator*. Disponível em: `https://unicode-org.github.io/icu/userguide/strings/characteriterator.html`.
- Unicode Consortium. *ICU API Reference: UnicodeString*. Disponível em: `https://unicode-org.github.io/icu-docs/apidoc/released/icu4c/classicu_1_1UnicodeString.html`.
- Unicode Consortium. *ICU User Guide: Locale*. Disponível em: `https://unicode-org.github.io/icu/userguide/locale/`.
- Unicode Consortium. *ICU User Guide: Collation*. Disponível em: `https://unicode-org.github.io/icu/userguide/collation/`.
- GeeksforGeeks. *Inorder Tree Traversal without Recursion*. Disponível em: `https://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/`.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. *Algoritmos: Teoria e Prática*. 3ª edição, Editora Campus, 2012. ISBN: 978-8535236996.
  - Capítulo 11: Tabelas de Espalhamento.
  - Capítulo 13: Red-Black Trees.
- Jayme, M., Lilian, C. *Estruturas de Dados e seus Algoritmos*. 3ª edição, Editora LTC, 2010.
  - Capítulo 5: Árvores Balanceadas
  - Capítulo 8: Tabelas de Dispersão.