Trabalho Prático 3
As mensagens secretas da Aliança Rebelde

Euller Saez Lage Silva

2019054501

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

eullersaez@ufmg.br

1. Introdução

O problema proposto se consistia em auxiliar a Aliança Rebelde de um universo paralelo da saga Star Wars no intuito de que seus membros pudessem trocar mensagens criptografadas entre si e finalmente derrotar o imperador. Assim, neste sistema computacional desenvolvido eles deveriam poder inserir palavras no sistema, substituir alguma já inserida por uma outra, encriptar uma mensagem composta de algumas palavras e também desencriptar uma mensagem que seria composta por posições específicas de palavras já inseridas no sistema.

2. Implementação

O programa foi desenvolvido na linguagem C++ por meio do Visual Studio Code para Windows da Microsoft Corporation, utilizando o Windows Subsystem for Linux (WSL) e compilado pelo compilador G++ da GNU Compiler Collection.

2.1 Estrutura de dados

A implementação do programa se deu pela escolha da estrutura de dados Árvore Binária de Pesquisa, uma vez que a própria especificação do trabalho requeria a utilização de árvore binárias. Além disso, o fato de ações como a de inserir novas palavras levar em consideração a sua ordem alfabética também faziam com que esse tipo de estrutura de dados fosse a mais conveniente.

2.2 Classes

Foram utilizadas duas classes principais para a implementação do trabalho e inclusive ambas foram estritamente baseadas nas implementações vistas em aula, fazendo as devidas modificações necessárias para o funcionamento adequado do trabalho. A primeira delas, base para a segunda, era a classe No, que possuia um atributo chave do tipo string, e outros dois ponteiros para esq e dir, ambos do tipo No, além de um construtor vazio.

A outra classe era Arvore, que possuía um atributo raiz ponteiro do tipo No. Além disso, possuía um construtor vazio e mais alguns métodos úteis. O primeiro deles era *insere* que recebia uma string e conforme comentado no código, chamava *insere_recursivo* no nó raiz com a mesma string, e este por sua vez achava recursivamente a posição onde um novo nó deveria ser criado com a chave igual a string recebida, levando em consideração a regra de ordenação das árvores binárias de pesquisa. O método *remove* que recebia uma string a ser

removida da árvore, chamava o método *remove_recursivo* na raiz passando a mesma string como parâmetro e basicamente ficava responsável por procurar na árvore pelo nó cuja chave era igual a string a ser removida e, quando o encontrava, caso não tivesse pelo menos um dos filhos (da esquerda ou da direita), era substituído pelo nó filho existente (ou por nulo quando nenhum era presente). Caso possuísse dois filhos, era substituído pelo seu antecessor com o auxílio do método *antecessor,* que achava qual nó era o antecessor adequado e fazia a devida remoção. O método *substitui* recebia duas strings, chamava *remove* para a primeira e *insere* para a segunda. O método *encripta* recebia uma string, criava um iterador ponteiro do tipo inteiro e o passava por referência junto com a string e o nó raiz da árvore para *pre_ordem_encripta,* que ficava responsável por percorrer recursivamente a árvore pelo caminhamento pré-ordem, iterando a cada nó a posição do iterador. Caso a chave do nó fosse igual a string recebida, imprimia-se na tela a posição do iterador, ou seja, a posição do nó na árvore. De modo muito similar, o método *desencripta* recebia um inteiro, criava um iterador ponteiro do tipo inteiro e o passava por referência junto com o nó raiz da árvore e o inteiro recebido para *pre_ordem_desencripta*, que da mesma forma que *pre_ordem_encripta,* percorria a árvore pelo caminhamento pré-ordem. No entanto, agora comparava-se o inteiro com a posição do iterador. Caso fosse igual, imprimia-se a chave do nó naquela posição.

3. Instruções de compilação e execução

3.1 Compilação

Para compilar o programa primeiro é necessário abrir o diretório euller_silva/src no terminal do linux e digitar o comando “make”.

3.2 Execução

Para executar o programa, após compilado, ainda dentro do mesmo diretório euller_silva/src no terminal do linux, basta digitar o comando “./tp3”.

3.3 Limpeza de arquivos temporários

Para limpar todos os arquivos temporários gerados durante a compilação, ainda dentro do mesmo diretório euller_silva/src no terminal do linux, basta digitar o comando “make clean”.

4. Análise de complexidade

4.1 Tempo

A complexidade de tempo do programa como um todo irá depender de quais métodos serão chamados na função main. Assim, teremos w vezes o custo de se chamar *insere*, x vezes o custo de se chamar *substitui*, y vezes o custo de se chamar *encripta* e z vezes o custo de se chamar *desencripta*.

O custo de *insere* é o custo de se chamar *insere_recursivo*. Quando a árvore é binária degenerada, é necessário percorrer recursivamente todos os nós, e, portanto o custo de se percorrer todos os nós sequencialmente é linear, caindo em seu pior caso $O(n)$. No melhor

caso, a árvore será balanceada e o número de nós a ser percorrido será da ordem da altura da árvore, ou seja um custo logarítmico *O(*log*)*.

O custo de *substitui* é o custo de se chamar *insere* mais o custo de se chamar *remove*. Então, na análise de *remove,* ao chamar *remove_recursivo*, em seu pior caso também terá que percorrer sequencialmente uma árvore degenerada até achar o nó a ser removido, o que terá custo linear *O(*n*)*. No seu melhor caso, percorrendo uma árvore balanceada e sendo o nó a ser removido um nó com no máximo um filho, o método *antecessor* não é chamado e portanto o número de nós percorridos será da ordem da altura da árvore, ou seja um custo logarítmico *O(*log*)*. Desse modo, no melhor caso de *substitui*, teremos

$$O(\log n) + O(\log n) = O(\log n)$$

Enquanto no seu pior caso

$$O(n) + O(n) = O(n)$$

Quanto a *encripta* e *desencripta,* ambos chamam *pre_ordem_encripta* e *pre_ordem_desencripta* que têm funcionamento praticamente idêntico. O custo deles não é melhor ou pior com nenhuma disposição diferente dos dados, isso porque em qualquer caso sempre é preciso fazer todo o caminhamento pré-ordem em toda a árvore. Assim, sempre terão custo linear $O(n)$.

Logo, no melhor caso do programa teremos w chamadas de *insere* em seu melhor caso, x chamadas de *substitui* em seu melhor caso, y chamadas de *encripta* e z chamadas de *desencripta*

$$w\ O(\log n) + x\ O(\log n) + y\ O(n) + z\ O(n) = O(n)$$

Enquanto no seu pior caso, teremos w chamadas de *insere* em seu pior caso, x chamadas de *substitui* em seu pior caso, y chamadas de *encripta* e z chamadas de *desencripta*

$$w\ O(n) + x\ O(n) + y\ O(n) + z\ O(n) = O(n)$$

4.2 Espaço

Vamos considerar que cada nó ocupa uma unidade de espaço no programa.

No caso de *insere*, a cada etapa da pilha de execução 1 nó é armazenado. Assim, o seu custo de espaço irá depender de qual caso irá cair e, conforme analisado anteriormente, teremos que no melhor caso seu custo de espaço será de

$$1O(\log n) = O(\log n)$$

Enquanto em seu pior caso

$$1O(n) = O(n)$$

No caso de *substitui*, assim como explicado anteriormente, teremos o custo de espaço de *insere* e de *remove*, e, de forma muito similar a explicação de custo de espaço de *insere*, teremos que armazenar dois nós (auxiliar + atual) a cada chamada recursiva. Assim, em seu melhor caso, *substitui*

$$1O(\log n) + 2O(\log n) = O(\log n)$$

Enquanto em seu pior caso

$$1O(n) + 2O(n) = O(n)$$

No caso de *encripta* e *desencripta*, ao chamar *pre_ordem_encripta* e *pre_ordem_desencripta,* como sempre percorrem toda a árvore, em cada chamada recursiva irão armazenar um nó, e, portanto terão sempre custo linear $O(n)$.

5. Conclusão

Após o término do trabalho, ficou bem evidente a importância de se estudar diferentes estruturas de dados para a resolução de diferentes problemas.

Em geral, a utilização de outras estruturas tornaria a resolução do trabalho extremamente mais complexa e talvez até inviável. Algumas propriedades das Árvores Binárias de Pesquisa, por exemplo, foram muito úteis na implementação do código. Manter no nó à esquerda apenas chaves menores e chaves maiores apenas nos nós à direita em cada subárvore fez com que o processo de inserção e remoção como um todo fossem mais facilitados e, apesar de muitas vezes não serem tão baratos como numa implementação de lista encadeada manipulada por ponteiros, as árvores binárias de pesquisa têm esse grande diferencial da organização que diminui bastante a dificuldade de se percorrer a estrutura após montada.

Dessa forma, reforça-se a necessidade de verificar bem cada aplicação e determinar de modo a levar questionamentos como a importância de sempre se manter os dados organizados ou não, além dos custos envolvidos nesse processo em consideração na hora de se escolher qual estrutura será utilizada.

References

CHAIMOWICZ, L. , PRATES, R. Slides da disciplina Árvores - Estrutura de Dados. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais (UFMG). 2020

CHAIMOWICZ, L. , PRATES, R. Slides da disciplina Árvore Binária de Pesquisa, Pesquisa em Memória Primária - Estrutura de Dados. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais (UFMG). 2020