Trabalho Prático 3

As mensagens secretas da Aliança Rebelde

Gilliard Gabriel Rodrigues
Matrícula: 2019054609

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

gilliard2019@ufmg.br

1. Introdução

O problema proposto foi implementar um sistema de transmissão de mensagens encriptadas para a Aliança Rebelde, baseado em uma árvore binária.

Segundo as especificações da Aliança Rebelde, dado um idioma de entrada, as mensagens da Aliança Rebelde a serem transmitidas devem ser encriptadas seguindo a sua disposição numa árvore binária, de modo que uma mensagem de texto comum seja transformada num texto “ilegível” por quem não conhece a criptografia. De forma análoga, uma mensagem criptografada deve ser descriptografada também seguindo a disposição numa árvore binária.

Conforme explicado na especificação do trabalho, a encriptação de uma mensagem ocorre no mapeamento das palavras para a sua posição na árvore, considerando o caminhamento pré-ordem, da seguinte forma: seja “absha ut alehsa” uma mensagem. Suponha que, pela visitação pré-ordem, a palavra "absha" seja a de ordem 1, "ut" seja de ordem 4 e "alehsa", de ordem 3. Dessa forma, a mensagem encriptada será "1 4 3" e com ela seria possível percorrer o caminho reverso, chegando à mensagem em linguagem natural.

Levando em consideração que o idioma utilizado pela Aliança é muito dinâmico, além das operações de encriptação e desencriptação de mensagens, o sistema a ser implementado deve permitir a inserção e substituição de palavras à árvore, respeitando o funcionamento do TAD.

Com base nessas especificações, será apresentada nesta documentação uma implementação para o sistema de transmissão de mensagens encriptadas utilizando modificações do TAD árvore binária visto em aula.

2. Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection.

2.1 Estrutura de Dados

Como citado anteriormente, a estrutura de dados utilizada foi a árvore binária. Foram feitas algumas modificações na classe, como por exemplo a exclusão de alguns métodos que não seriam utilizados, como também a adição de métodos extras para encriptar e desencriptar mensagens.

2.2 Classes e métodos

A fim de seguir as boas práticas de programação e implementar um código modularizado, foram montadas três classes, divididas em .h e .cpp.

A primeira delas chama-se *TipoPalavra* e foi baseada na classe *TipoItem* apresentada em aula. Ela possui um atributo privado "*TipoChave* palavra", em que *TipoChave* é uma *string*, e possui também métodos públicos *GetPalavra()*, *SetPalavra()* e *Imprime()*, cujos nomes já explicam suas utilidades. Há também dois construtores, um vazio que inicializa a chave como "-1" (string, não inteiro), e um com parâmetro *TipoChave* p.

A segunda classe chama-se *TipoNo* e é basicamente como aquela vista em aula, possui os atributos privados *TipoPalavra* palavra, *TipoNo* *esq e *dir, representando a chave do nó e seus ponteiros para os nós filhos, além de um construtor, que inicializa palavra como "-1" e os ponteiros apontando para NULL. Esta classe é *friend class* da árvore binária.

A terceira e última classe representa a árvore binária propriamente dita, e chama-se *ArvoreBinariaDeCriptografia*, que possui cinco métodos públicos, seis métodos privados e um atributo *TipoNo* *raiz, também privado.

O primeiro método público é o *void Insere(TipoPalavra chave)*, que apenas serve para acessar o método privado *void InsereRecursivo(TipoNo *&p, TipoPalavra palavra)*, passando como parâmetros a raiz e a chave a ser inserida (*TipoPalavra palavra*), que insere na árvore uma palavra nova, seguindo a ideia de que se a palavra a ser inserida for anterior à palavra presente no nó atual, ela é inserida na subárvore da esquerda, caso contrário, ela é inserida na subárvore da direita (utilizando chamadas recursivas até encontrar o lugar ideal para a nova palavra).

O segundo método público é o *void Encripta(TipoChave mensagem)*, que basicamente inicializa um *int posicao* como zero e chama o método privado *void EncriptaRecursivo(TipoNo *p, TipoChave mensagem, int *posicao)*, passando como parâmetros a raiz, a i-ésima palavra da mensagem a ser encriptada (será mais detalhada posteriormente) e o endereço de memória do *int posicao* instanciado no método *Encripta*. O método chamado é basicamente um caminhamento pré-ordem modificado, onde ao invés de imprimir cada chave antes das duas chamadas recursivas, ele incrementa a variável *int posicao* e analisa com um *if* se a palavra do nó atual corresponde à i-ésima palavra da mensagem, e se sim, imprime a posição dela segundo o caminhamento pré-ordem.

O terceiro método público é o *void Desencripta(int mensagem_criptografada)*, que basicamente inicializa um *int posicao* como zero e chama o método privado *void DesencriptaRecursivo(TipoNo *p, int mensagem_criptografada, int *posicao)*, passando como parâmetros a raiz, o i-ésimo número da mensagem criptografada (será mais detalhado posteriormente) e o endereço de memória do *int posicao* instanciado no método *Desencripta*. O método chamado também é basicamente um caminhamento pré-ordem modificado, onde ao invés de imprimir cada chave antes das duas chamadas recursivas, ele incrementa a variável *int posicao* e analisa com um *if* se o i-ésimo número da mensagem criptografada (armazenado em *int mensagem_criptografada*) é igual à posição atual (armazenada no inteiro *posicao* que é incrementado a cada caminhada), e se sim, imprime a palavra presente no nó atual, segundo o caminhamento pré-ordem.

O quarto método público é o void *Remove(TipoChave chave)*, que serve apenas para chamar o método privado *RemoveRecursivo(TipoNo *&no, TipoChave chave)*, passando como parâmetros a raiz e a chave a ser removida da árvore. Esse método é basicamente igual ao visto em aula: se a palavra a ser removida é anterior à do nó atual, chama recursivamente o método para a subárvore da esquerda, caso contrário, chama para a subárvore da direita, após encontrar o nó procurado, se ele tem um filho da esquerda, esse nó filho fica no lugar do removido, se é da direita, ocorre de forma análoga, e caso o nó a ser removido tenha dois nós filhos, chama o método privado *Antecessor(TipoNo *q, TipoNo *&r)*, passando como parâmetros o nó atual e o seu filho da esquerda, e em seguinte esse método encontra qual é a palavra antecessora à que será removida, a fim de colocar em seu lugar.

O quinto método público é o void *Limpa()*, que serve apenas para chamar o método privado *LimpaRecursivo(TipoNo *p)*, passando como parâmetro a raiz, que serve para deletar todos os nós da árvore. Esses métodos são utilizados no destrutor do TAD.

2.3 Entradas e saídas

Conforme especificado pela Aliança Rebelde, a entrada e a saída são padrões do sistema (stdin e stdout, respectivamente). A entrada será composta por **N** linhas, tal que $0 < \mathbf{N} \leq 10000$. Em cada linha haverá uma operação a ser realizada pelo seu sistema, cujo primeiro caractere da linha indica qual é a operação correspondente, conforme segue:

• **i**: indica uma operação de inserção de palavra na ´arvore. Para esta operação, uma linha de entrada é constituída pelo caractere 'i' e uma string *s*, tal que o tamanho de $s \leq 15$ caracteres, sendo a palavra a ser inserida. Para esta operação, não deve ser impresso nada na saída do programa;

• **s**: indica uma operação de substituição de palavra na ´arvore. Para esta operação, uma linha de entrada é constituída pelo caractere 's'; uma string $s1$, tal que o tamanho de $s1 \leq 15$ caracteres, sendo a palavra a ser substituída; e uma string $s2$, tal que o tamanho de $s2 \leq 15$ caracteres, sendo a nova palavra a ser inserida na árvore. Para esta operação, não deve ser impresso nada na saída do programa;

• **e**: indica uma operação de encriptação de uma mensagem. Para esta operação, uma linha de entrada é constituída pelo caractere 'e'; um inteiro i, tal que $i \leq 20$, que indica o número de palavras que compõem a mensagem; e uma mensagem *M*, constituída por i palavras. Para esta operação, a mensagem encriptada deve ser impressa na saída do programa.

• **d**: indica uma operação de desencriptação de uma mensagem. Para esta operação, uma linha de entrada é constituída pelo caractere 'd'; um inteiro i, tal que $i \leq 20$, que indica o número de palavras que compõem a mensagem; e uma mensagem *M*, constituída por i palavras. Para esta operação, a mensagem desencriptada deve ser impressa na saída do programa.

O final da entrada é indicado por EOF (CTRL + D no terminal).

2.4 Estrutura do main

No *main()* do programa, ocorre a instanciação de um objeto do tipo *ArvoreBinariaDeCriptografia*, de um *TipoPalavra* e das demais variáveis necessárias.

Em seguida, tem um *while* que itera enquanto não chegar ao EOF, e dentro desse *while* há um *switch*, que possui quatro *cases* e um *default*.

No *case 'i'* ocorre a operação de adição de uma nova palavra à árvore, uma palavra é lida do teclado, adicionada ao *TipoPalavra p* através do seu método *SetPalavra* e, em seguida, inserida em *ArvoreBinariaDeCriptografia arvore* através do seu método *Insere.*

No *case 's'* ocorre a operação de substituição de uma palavra na árvore, uma palavra é lida do teclado, passada como parâmetro para o método *Remove*, em seguida a segundo palavra é lida e adicionada ao *TipoPalavra p* através do método *SetPalavra* e, em seguida, inserida em *ArvoreBinariaDeCriptografia arvore* através do seu método *Insere.*

No *case 'e'* ocorre a operação de encriptação da mensagem. O número de palavras da mensagem é lido do teclado e assinalado à variável *num_palavras*, que por sua vez é utilizada como limite superior de um *for*. Dentro desse *for*, ocorre a leitura da i-ésima palavra da mensagem e a chamada do método *Encripta*. É por isso que o método criptografa uma palavra da mensagem por vez, como comentado anteriormente.

No *case 'd'* ocorre a operação de desencriptação da mensagem. O número de palavras da mensagem é lido do teclado e assinalado à variável *num_palavras*, que por sua vez é utilizada como limite superior de um *for*. Dentro desse *for*, ocorre a leitura do i-ésimo número da mensagem criptografada e a chamada do método *Desencripta*.

No *default* tem apenas o *break.*

2.5 Instruções de compilação e execução

Para compilar o trabalho, acesse o diretório do plano de dominação que deseja executar via terminal, e digite:

make

Em seguida, para executar o código, digite:

./tp3

Logo após, execute as operações conforme especificado na subseção anterior.

3. Análise de complexidade

3.1 Tempo

Para chegar a uma conclusão sobre o custo de complexidade do algoritmo, vamos analisar a função *main()* passo a passo. Há um *while* que itera até EOF (ou seja, até o usuário teclar Ctrl + D no terminal), dentro dele existe um switch com 4 cases. Sabe-se que as operações como atribuição às variáveis e exibir uma mensagem na tela têm custo constante, quanto às operações relacionadas aos métodos do TAD é preciso analisar cada caso:

Se o usuário tecla 'i', o programa executa uma operação de custo *O(log n)* no melhor caso e *O(n)* no pior caso, pois trata-se de uma inserção de um nó na árvore binária, e como visto em sala, a complexidade de tempo é essa.

Se o usuário tecla 's', o programa executa uma remoção e uma inserção, ambas essas operações tem melhor caso *O(log n)* e pior caso *O(n)*.

Se o usuário tecla ‘e’, o programa executa a encriptação de uma mensagem. A ordem de complexidade de tempo do método Encripta é a mesma de um caminhamento pré-ordem, ou seja: O(n), pois visita todos os nós da árvore. Seja k o número de palavras que compõem a mensagem a ser criptografada, o método Encripta será chamado k vezes no *main()*, então a complexidade de tempo será O(n*k).

O mesmo vale para o último caso, quando o usuário tecla ‘d’ para desencriptar uma mensagem. A ordem de complexidade de tempo do método Desencripta também é O(n) e será chamado k vezes no *main()*. Portanto, a ordem de complexidade de tempo nesse caso é O(n*k). Portanto, o melhor caso para o *switch* é O(log n) e o pior caso é O(n*k).

Independentemente do número de vezes em que o usuário escolher fazer as operações, esse número será sempre uma constante, logo, tem-se que o custo de complexidade do while no melhor caso é no máximo $O(log\ n) * c$, que é $O(log\ n)$. E no pior caso tem-se que o custo de complexidade do while será $O(n*k) * c$, que é $O(n*k)$.

Dessa forma, conclui-se que a ordem de complexidade de tempo do algoritmo é $O(log\ n)$ no melhor caso e $O(n*k)$ no pior caso.

3.2 Espaço

Como o TAD árvore binária utiliza alocação dinâmica, a ordem de complexidade de espaço do algoritmo é $O(n)$, pois depende da quantidade de nós que irão compor a árvore.

4. Conclusão

O ponto mais importante para conseguir implementar o sistema de transmissão de mensagens encriptadas para a Aliança Rebelde foi, sem dúvidas, saber como modificar o método de caminhamento pré-ordem de forma a conseguir encriptar e desencriptar uma mensagem, ou seja, colocar um contador que contasse as posições durante o caminhamento e um if que verificasse se a palavra do nó atual era igual à i-ésima palavra da mensagem (ou a posição igual ao i-ésimo número da mensagem criptografada). Fora isso, o restante era praticamente igual à implementação de uma árvore binária normal.

No decorrer desse trabalho, ficou evidente a importância de fazer testes para ter certeza de que o algoritmo estava encriptando e desencriptando as mensagens de forma correta. Por fim, foi dada uma atenção especial ao processo de refatoração e modularização do código para atender às boas práticas de programação e facilitar a identificação e correção de erros.

Por fim, foi possível aprender uma utilidade nova para as árvores binárias, complementando o aprendizado adquirido durante as aulas sobre o TAD.

Referências

CHAIMOWICZ, Luiz e PRATES, Raquel. Slides da disciplina Estrutura de Dados. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. 2020.