Trabalho Prático 3

Decifrando os Segredos de Arendelle

Paula Mara Ribeiro

2018047048

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

paulamribeiro@ufmg.br

1. Introdução

O problema proposto consiste em implementar um mecanismo de descriptografia a partir de uma estrutura de árvore. Assim, primeiro é lido um arquivo que define os caracteres e os códigos correspondentes e é montada uma árvore do tipo trie para armazenar essas informações. Com a árvore montada, é possível descriptografar as mensagens recebidas na entrada padrão.

2. Implementação

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

Primeiramente, o programa lê o arquivo *morse.txt* na raiz do projeto. Ele deve conter a codificação de cada caracter a ser utilizado. Depois, o programa fica disponível para receber as entradas a serem descriptografadas. Quando a entrada de parâmetros é finalizada, se o parâmetro *-a* foi utilizado na execução do programa, a árvore montada será impressa em pré-ordem.

2.1 O código

A **main** é responsável por executar as principais operações do código. Assim, ela obtém a árvore construída a partir da função *carregarArvore* do Orquestrador, depois, ela obtém as entradas e as descriptografa com a função *descriptografar* da Criptografia e, por fim, ela imprime a árvore em pré-ordem caso o parâmetro *-a* tenha sido utilizado.

A **árvore** possui um destrutor padrão e um construtor que inicia a raiz com um novo No que tem apontamentos nulos. A função *pesquisar* percorre a árvore a partir da raiz até que o código sendo pesquisado termine e o item seja encontrado, ou até que seja encontrado um item nulo.

A função *lerPreOrdem* chama a função privada *itemPreOrdem* para o nó da raiz. A função *itemPreOrdem* é recursiva e tem condição de parada no No nulo.

Caso a letra seja inválida, aquele No nao é impresso na leitura em pré-ordem. Depois, a função é chamada novamente para as subárvores da esquerda e direita.

A função de *inserir* pesquisa, a partir da função *pesquisar*, o item. Caso ele exista na árvore, o atributo correspondente a letra é igualado a letra do item a ser inserido e a função é finalizada. Caso não exista, então a árvore é percorrida inserindo os itens intermediários. Assim, itera-se sobre o código e procura-se o item até a i-ésima posição do código a ser inserido. Para encontrar esse item, utiliza-se a função *copiarString*, que cria uma nova string até o tamanho determinado. Caso o item intermediário não exista, ele é inserido e o item atual é atualizado para apontar para ele. Depois, o item atual é atualizado para apontar para o item do código até a i-ésima posição. Por fim, após terminar a iteração, o item atual será o item com o código que deveria ser inserido, então atualiza-se a letra desse item para a que deve ser inserida.

O **No** possui destrutor padrão e construtor que inicializa os atributos como nulos e a letra como vazia.

O **Orquestrador** possui a função *carregarArvore*, responsável por retornar a árvore com os códigos preenchidos. Ela valida e abre o arquivo *morse.txt* e depois lê caractere por caractere até o fim do arquivo. Um novo No é inicializado e o primeiro caractere lido corresponde a letra do código. O próximo é um espaço vazio. Depois, lê-se até encontrar uma quebra de linha, e esses caracteres são armazenados numa variável auxiliar, que ao ser finalizada é passada para o No criado e é inserido na árvore. Ao chegar ao fim do arquivo, ele é fechado e a árvore retornada.

A **Criptografia** possui a função *descriptografar*, que recebe a árvore com a codificação e é responsável por ler a entrada, descriptografá-la e exibi-la. Para isso, ela utiliza três variáveis auxiliares, a *contadorLetras*, a *contadorCaracteresCodigo* e a *anterior*. Os caracteres são lidos enquanto houver entrada. Se o caractere lido for *'/'*, então delimita-se a palavra. Assim, na variável resposta insere-se um espaço vazio e atualiza-se o *contadorLetras* e reinicia-se o *contadorCaracteresCodigo*. Caso o caractere seja diferente de espaço ou nova linha, então ele é uma entrada válida para descriptografia e é guardado na variável *codigo* e a *contadorCaracteresCodigo* é atualizada. Por fim, se a entrada for um espaço vazio e a anterior não for uma barra, isso quer dizer a entrada do código foi finalizada e deve-se realizar a descriptografia, então o *codigo* é delimitado, a resposta recebe o caractere descriptografado do código e é delimitada, e o *contadorCaracteresCodigo* é reiniciado. Se o caractere recebido é uma quebra de linha, então aquela entrada foi finalizada e deve ser impressa e as variáveis de *resposta* e as auxiliares são reiniciadas. A variável anterior é atualizada a cada iteração. Ao terminar de ler a entrada, é preciso computar o último código recebido e exibir a resposta.

3. Instruções de compilação e execução

Para rodar o programa criado, basta executar o arquivo *make* anexo. É preciso que o arquivo com a definição do código exista e seja nomeado *morse.txt* na raiz da projeto.

4. Análise de complexidade

As funções da **No** são O(1).

O **No** ocupa o espaço referente a dois char e dois No.

Os construtores da **Arvore** também são O(1). A função *itemPreOrdem* e O(n), pois passa por todos os nós das subárvores do nó inserido uma única vez. Assim, a função *lerPreOrdem* é também O(n), pois ela apenas chama a função *lerPreOrdem* para a raiz. A função *copiarString* é O(t), sendo t o tamanho da string que deve-se copiar. A função *pesquisar* é O(k), sendo k o tamanho do código a ser pesquisado.

Para montar a **Arvore**, ela ocupa o espaço de um No (raiz) + n No, sendo n o número de itens da árvore. A função *copiarString* gasta um vetor de char do tamanho t definido. As outras funções não ocupam novas posições de memória.

A função *inserir*, no melhor caso, encontra o nó já inserido e apenas atualiza a letra, sendo assim O(k). No pior caso, o item não é encontrado, portanto a é preciso iterar sobre todos os caracteres do código e pesquisar até a i-ésima posição. Assim, executa-se a *copiarString* com custo O(i) e a *pesquisar* com um código de também tamanho *i* e então são executadas operações O(1). Portanto, cada iteração tem complexidade 2i. Ela é executada k vezes, sendo k o tamanho do código, portanto 2ki. Como i é menor ou igual ao tamanho k do código, a complexidade pode também ser definida como O(k).

A função de *inserir* apenas ocupa as posições de memória necessárias para inserir na árvore. No final, o tamanho ocupado pela árvore é n + 1, sendo n o tamanho de itens.

No **Orquestrador**, o melhor caso pode ser definido como o arquivo não existindo, em que ele será O(1). Caso contrário, ele itera para cada caractere do arquivo. Apesar de ter whiles aninhados, o while interno colabora para a finalização do while externo. Assim, a operação mais custosa que ele executa é o *inserir* na árvore, definida como O(k), sendo k o tamanho do código. Como o *inserir* é executado no while externo, que representa a quantidade de entradas (ou linhas) no arquivo, a complexidade da função *carregarArvore* é O(pk), sendo k o tamanho do maior código a ser inserido e p o número de linhas no arquivo.

Na função *carregarArvore*, utiliza-se durante a execução um espaço de memória para guardar um tipo FILE. Depois, utiliza-se uma posição de char, um vetor de char de até 10 caracteres para salvar o código e um inteiro. Como ela é usada para inserir o item, ela também ocupa a memória para esse novo registro.

Na **Criptografia**, o melhor caso é não receber entrada, pois dessa forma são realizadas apenas o pesquisar para um código vazio e a resposta é exibida, sendo O(k). No pior caso, todo caractere recebido seria um espaço, pois dessa forma a pesquisa na árvore seria executada em cada execução e ela é a operação mais custosa da função. Então, ela seria O(qk), sendo q o tamanho da entrada.

Na criptografia, utiliza-se 2 inteiros contadores, um char e dois vetores de char durante a execução.

Por fim, a aplicação pode ser definida como O(qk), pois em geral, o tamanho da entrada a ser descriptografada dominará assintoticamente o O(pk), que representa o custo para montar a árvore de códigos.

Em questões de memória, assintoticamente o maior custo será o de guardar a árvore, sendo então de n No.

5. Conclusão

Ao realizar esse trabalho, foi possível identificar a importância de escolher a estrutura de dados correta para a resolução de um problema. Nesse caso, a trie era a mais indicado pela necessidade de recuperar a informação a partir de um código.

6. Bibliografia

Cplusplus.com. (2019). C++ Tutorials. [online] Disponível em: <http://www.cplusplus.com>

CHAIMOWICZ, Luiz; PRATES, Raquel. Árvore de Pesquisa Binária.

FEOFILOFF, Paulo. Tries (árvores digitais). [online] Disponível em: <https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/tries.html>