Documentação - Trabalho Prático 3 – Estruturas de Dados

Mateus Latrova Stephanin - 2019006981

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

mateusls@ufmg.br

1. Introdução

Esta documentação lida com a terceira parte do problema "Extração Z" explicado na especificação do trabalho. Como os piratas estão atrapalhando a comunicação entre as bases e a central de controle, a solução desse problema dependeu da automatização da codificação e decodificação de mensagens.

Para tal, foram elaborados algoritmos para cada uma das situações possíveis: recebimento do alfabeto utilizado para as mensagens, codificação da mensagem a ser enviada e decodificação da mensagem recebida. Dessa forma, como foi necessário caminhar pela árvore de transliteração em cada um deles e, também, percorrer a string da mensagem em questão, foi utilizada uma estratégia que mesclou iteração e recursão.

Além disso, esta documentação tem como objetivo explicar e analisar clara e sucintamente como foram utilizados recursos, ideias e ferramentas para a solução desse problema. A fim de resolvê-lo, foi seguida uma estratégia de planejamento e implementação do código considerando cada entidade apresentada como uma classe(paradigma orientado a objetos), tendo, cada uma, seu papel e suas comunicações bem definidos dentro do programa.

2. Instruções de compilação e execução

A compilação dos arquivos deve ser realizada por meio da execução do comando "make" pelo terminal dentro da pasta "projeto". Após compilados os arquivos, a execução do programa gerado deve ser feita exatamente conforme explicado na seção 3 - Entregáveis - da especificação do trabalho prático.

3. Classes implementadas

A seguir, está uma lista com todas as classes/templates de classes implementados juntamente com uma breve explicação a respeito de seus atributos e métodos mais relevantes:

Node<T>: similar à estrutura TipoCelula vista em aula, porém, neste caso, o TipoItem foi substituído por um tipo genérico T a ser instanciado no momento da criação de um objeto. Seus atributos incluem um objeto do tipo T e um ponteiro para outro objeto do tipo Node<T>, pois a finalidade aqui é construir listas, filas ou pilhas encadeadas.

LinkedStack<T>: uma pilha encadeada de objetos do tipo genérico T. Sua função aqui é, basicamente, proporcionar a realização de caminhamentos por nível nos objetos do tipo BinaryTree<T> caso fosse necessário.

TreeNode<T>: Similar ao tipo Node<T>, porém com uma única diferença: possui dois ponteiros para objetos do tipo TreeNode<T>. A ideia por trás disso é possibilitar a construção de uma árvore binária, a qual será descrita a seguir.

BinaryTree<T>: árvore binária de pesquisa cujos nós possuem um objeto do tipo T. Seu único atributo é um ponteiro para a raiz da árvore, a partir da qual pode-se alcançar qualquer nó desejado. Seu uso principal no projeto foi a construção da árvore de transliteração, a qual foi utilizada para codificar e decodificar mensagens. O único método novo utilizado dessa classe foi o **searchAndEncode()**. Ele foi usado como um método auxiliar para o processo de codificação(método encodeMessage() da classe ControlCenter descrita a seguir). Recebendo o valor(caracter) a ser buscado na árvore, à medida que ele desce nela fazendo a busca, ele codifica o valor buscado adicionando os números aleatórios usados para tal(conforme a especificação) numa string que será o resultado da codificação. Além disso, foi utilizado o método **insert()**, que corresponde ao método de inserção visto em aula.

ControlCenter: classe que representa o centro de controle da missão "Extração Z". Possui a árvore de transliteração como atributo e é capaz de coletar as mensagens e os comandos recebidos na entrada, podendo codificá-las ou decodificá-las conforme eles pedirem por meio de seus métodos. São eles:

1) **getAndExecuteCommands():** processa a entrada e, mediante os comandos dados, faz a chamada do método correspondente por meio de um comando switch. Os três métodos auxiliares usados por ele estão descritos logo abaixo.

2) **loadAlphabet():** este método é chamado quando se identifica um comando do tipo "A"(de alfabeto). Ele insere, na mesma ordem em que foram recebidos, os caracteres do comando na árvore de transliteração. Para isso, faz a chamada do método insert() da árvore num laço simples que percorre todos os caracteres da string do alfabeto recebido.

3) **decodeMessage():** invocado de acordo ao aparecimento do comando "D"(de decodificação). Faz a decodificação da mensagem codificada e imprime seu resultado na saída. Para tal, utiliza-se de dois laços aninhados. O laço externo é encarregado de percorrer todos os caracteres 'x' da mensagem codificada, ou seja, ele encontra o início da cifragem de cada letra da mensagem resultante. Achado esse início, parte-se para o laço interno, o qual usará os números pares e ímpares da mensagem cifrada para buscar a letra correspondente na árvore.

4) **encodeMessage():** por último, este método coincide com o comando "C"(de codificação). Utilizando-se de um laço que itera sobre as letras da string que contém a mensagem decodificada, ele chama o método searchAndEncode() da árvore de transliteração, codificando a mensagem letra por letra. Sendo que a encriptação de cada letra resulta numa string, cada string dessa vai sendo adicionada ao final da string que será impressa ao final do processo, isto é, a mensagem já codificada.

O programa principal foi responsável por criar o stream no qual seria aberto o arquivo de entrada – passado como parâmetro pelo terminal – além de fechá-lo ao final. Após aberto o arquivo, criou-se um objeto da classe ControlCenter e foi executado seu método getAndExecuteCommands(), que recebeu o stream do arquivo aberto como parâmetro. Como

descreveu-se acima, esse método processou a entrada e distribuiu todas as tarefas que deviam ser executadas aos objetos das devidas classes.

Obs.: a maioria das classes foram implementadas(exceto a classe ControlCenter) inteiramente no arquivo de cabeçalho devido ao uso de templates. Caso contrário, não seria possível realizar a compilação.

4. Análise de complexidade

Método insert(classe BinaryTree<T>):

Sendo n o número de elementos presentes na árvore em questão:

- **complexidade de tempo:** como foi visto em aula, o custo de tempo para a inserção de um elemento na árvore é O(log n) no melhor caso e O(n) no pior caso.
- **complexidade de espaço:** a inserção sempre alocará espaço no heap para apenas um novo nó. Porém, semelhantemente à complexidade de tempo, como são utilizadas chamadas recursivas nessa função, teremos que, no melhor caso(árvore balanceada), serão empilhadas O(log n) chamadas na memória e, no pior caso(árvore degenerada), serão empilhadas O(n) chamadas.

Método searchAndEncode(classe BinaryTree<T>):

Sendo n o número de elementos presentes na árvore em questão:

- **complexidade de tempo:** é conveniente considerar como operação de interesse a própria chamada da função, já que é recursiva e possui um custo de tempo constante em cada chamada. Dessa forma, pode-se concluir que, no melhor caso(quando o valor procurado está na raiz), o custo de tempo é O(1) e que, no pior caso(quando a árvore é degenerada e o elemento buscado é a folha), esse custo passa a ser O(n).
- **complexidade de espaço:** semelhantemente, no melhor caso, haverá um número constante de todas operações e, portanto, o custo de espaço é O(1). Enquanto que no pior caso, utilizando a mesma estratégia usada no método insert acima, já que a maneira que a recursão é feita neste método é praticamente idêntica, fica claro observar que o custo de espaço é O(n) tanto em relação à pilha de execução quanto em relação ao tamanho da string resultante.

Método loadAlphabet(classe ControlCenter**):**

Sendo n o tamanho da string que contém o alfabeto:

- **complexidade de tempo:** no pior caso, ou seja, quando a árvore resultante do carregamento do alfabeto for uma árvore degenerada, pode-se tratá-la como uma lista ligada. Dessa forma, tomando como operação de interesse a comparação do valor a ser inserido com o valor de cada nó, serão feitas $(1+n-1) \cdot n / 2 = O(n^2)$ comparações para a inserção. Já no melhor caso, isto é, quando a árvore resultante é balanceada, o número de

comparações feitas será o somatório de log de i na base 2, com i variando de 1 até n, o que resulta em $O(n \cdot \log n)$.

- **complexidade de espaço:** de maneira análoga, o custo de espaço é $O(n^2)$ no pior caso e $O(n \cdot \log n)$ no melhor caso, tomando como referência o número de chamadas empilhadas na pilha de execução do programa.

Método decodeMessage(classe ControlCenter**):**

Sendo n o tamanho da mensagem codificada:

- **complexidade de tempo:** tomando como operação de interesse o número de comparações em relação aos contadores i e j feitas durante a execução dos dois laços aninhados, podemos observar que não há melhor/pior caso, já que o algoritmo faz com que todos os caracteres da string dada como entrada sejam percorridos sempre e que os dois laços possuem um custo interno constante. Portanto, pode-se concluir que serão feitas sempre $n = O(n)$ dessas comparações.
- **complexidade de espaço:** o custo de espaço será constante exceto para a string resultante do processo de decodificação. Portanto, sendo m o número de caracteres 'x' na mensagem codificada e, logo, o tamanho da string resultante, esse método terá complexidade de espaço O(m).

Método encodeMessage(classe ControlCenter**):**

Sendo n o número de elementos presentes na árvore de transliteração e m o tamanho da string que contém a mensagem a ser codificada:

- **complexidade de tempo:** todas as operações dentro desse método possuem custo constante exceto a chamada do método searchAndEncode(). Como essa chamada está dentro de um laço de m iterações, fica evidente que o custo de tempo deste método será $m \cdot O(1)$ (custo do searchAndEncode no melhor caso) = $O(m)$ no melhor caso e $m \cdot O(n)$ (custo do searchAndEncode no pior caso) = $O(m \cdot n)$ no pior caso.
- **complexidade de espaço:** semelhantemente ao raciocínio anterior, já que o espaço utilizado na função é constante exceto pelo espaço utilizado pela função searchAndEncode(), o custo de espaço será da mesma ordem que o custo de tempo.

Método getAndExecuteCommands(classe ControlCenter**):**

Este método chama três métodos acima de acordo com a disposição dos comandos no arquivo de entrada. O comando 'A' é sempre executado apenas uma vez e é o primeiro de todos. Após ele, a disposição é aleatória.

Dessa maneira, o melhor caso ocorrerá quando todos os comandos(exceto o primeiro) corresponderem à função de menor complexidade dentre as três e, semelhantemente, o pior, quando eles corresponderem à função de maior complexidade. Então, sendo p o número de linhas nesse arquivo, n o número de elementos presentes na árvore de transliteração, m o tamanho da

string que contém a mensagem(codificada/decodificada) e q o tamanho da string que contém o alfabeto a ser carregado:

- **complexidade de tempo:** como são feitas p iterações(uma para cada comando) e, em cada uma delas, é chamada uma função, a complexidade de tempo no melhor caso seria $O(q^2) + (p-1) \cdot O(m) = O(q^2) + O(p) \cdot O(m)$. Como q está limitado a 27(tamanho máximo do alfabeto incluindo o caracter de espaço), pode-se dizer que essa complexidade é da ordem de $O(p) \cdot O(m) = O(p \cdot m)$. Entretanto, no pior caso, pelas mesmas razões, a complexidade é da ordem de $O(p \cdot m \cdot n)$.
- **complexidade de espaço:** analogamente ao raciocínio anterior, a ordem do custo de espaço terá a mesma ordem do custo de tempo, tanto no melhor caso, quanto no pior caso.

5. Conclusão

Este trabalho lidou com o problema "Extração Z" e resolveu-o com uma abordagem clara, organizada e com a maior simplicidade possível. Procurou-se fazê-lo de forma que qualquer programador pudesse entender a solução apenas analisando o código. Foram colocados alguns comentários em pontos relevantes afim de sanar quaisquer dúvidas eventuais.

Durante a implementação da solução, houve algumas dificuldades, as quais se tornaram grandes ganhos em conhecimento para a realização de futuros projetos. Foi possível praticar muitos conceitos antes vistos, porém que não haviam ficado claros sem a elaboração de um projeto.

Dentre os aprendizados obtidos mediante as dificuldades, podem ser listados:

- Orientação a objeto no geral, mas, principalmente, a parte de polimorfismo utilizando templates.
- Implementação e funcionamento de árvores binárias.
- Cálculo do custo de métodos mais complexos, envolvendo a soma de logaritmos.
- Uso da recursão em problemas mais elaborados.
- Dependência circular(como resolver utilizando *forward declarations*);
- Prática de debugação;
- Uso do valgrind para verificação de *memory leaks*.

6. Bibliografia

- http://web.eecs.utk.edu/~bvanderz/teaching/cs365Sp12/notes/templates.html
- https://math.stackexchange.com/questions/805479/summation-with-ceilinged-logarithmic-function
- http://www.cplusplus.com/
- https://stackoverflow.com/
- https://www.geeksforgeeks.org/