Documentação TP3 ED - Descobrindo os segredos de Arendelle

Autor: Matheus Ibrahim Fonseca Magalhães
Matrícula: 2018046912

1. Introdução

A tarefa apresentada determinava que era necessário criar um sistema capaz de traduzir uma variação de código morse para uma mensagem no alfabeto romano, utilizando como base para a tradução uma árvore Trie. Para isso, foi estabelecida uma relação entre cada símbolo do alfabeto(as 26 letras e os 10 algarismos, sem acentos ou caracteres especiais) e sua conversão para a variação de código morse utilizada. O código morse também possuía um espaço em branco entre cada letra e uma barra entre cada palavra, já que símbolos diferentes podem ser compostos por quantidades diferentes de elementos('.' e '-').

Portanto, o programa foi dividido em duas partes: a primeira consiste na construção da árvore, com base em um arquivo .txt que definiu a representação de cada símbolo no código morse, e a segunda na conversão de mensagens, recebendo uma sequência indefinida de caracteres em código morse e traduzindo-os, realizando, para isso, uma pesquisa na árvore até receber o código completo de um nó e retornando sua chave(que pode ser uma letra ou número).

O programa também aceita um parâmetro opcional "-a", que, caso passado, deverá imprimir a árvore criada em caminhamento pré-ordem.

2. Implementação

O programa foi implementado na linguagem de programação C++ e compilado em uma máquina virtual que simula um ambiente Linux com o compilador g++.

2.1) Arquivos criados

Foram criados três arquivos para a implementação do programa: o primeiro, o arquivo *main.cpp*, simplesmente chama as duas funções responsáveis por cada uma das etapas(criação da árvore e tradução da mensagem), além de possuir a lógica que verifica se o parâmetro opcional foi passado, e, caso tenha sido, imprime a árvore.

Segundo, o arquivo *tree.hpp*, que possui a implementação da estrutura *Node*, que define os dados existentes a cada nó da árvore, e o contrato que deve ser implementado pela árvore, ou seja, seus atributos e funções.

E, por último, o arquivo *tree.cpp*, que implementa o contrato estabelecido em *tree.hpp*, incluindo as funções responsáveis por praticamente toda a lógica do programa.

2.2) Fluxo do programa

O programa na função *main*, como já explicado, chama duas funções. A primeira, chamada *createTree*, recebe como argumento uma string, que define o nome do arquivo que a função deve usar como base para criar a Trie. Quando a função é chamada, ela abre o arquivo utilizando uma variável do tipo *ifstream*(utilizada para transcrever dados de um arquivo para outras variável), e realiza uma de duas operações: se a entrada for um dígito do alfabeto, ela atribui o símbolo à chave do nó atual e atribui o caminho do nó atual(valor em código morse) ao código do nó, definindo assim, uma forma mais simples de encontrar o valor em morse de cada símbolo, já que o próprio nó armazena esse valor(útil para a impressão da árvore). Se a entrada for um '.' ou um '-', a função chama uma função auxiliar *insertNode*, que recebe um caracter em morse e um nó, e percorre a árvore, podendo criar um novo nó ou simplesmente retornar um nó já existente, dependendo do caminho que está sendo percorrido já tiver sido ou não criado. Realizando essas duas ações para cada valor no arquivo, é possível criar a árvore corretamente, com cada nó possuindo seu valor no alfabeto romano e em código morse.

A segunda chamada de *main* é para a função da árvore *translate*, que não recebe argumentos. A função lê da entrada padrão uma quantidade indefinida de strings em código morse, no padrão explicado no item 1. desse documento, e opera dentro de um laço *for* que itera sobre cada caracter da string digitada. Dentro desse laço, uma das três operações serão realizadas: caso a entrada seja um ponto ou um traço, esse valor é concatenado a uma variável string auxiliar *letterCode*, que representa o valor de um símbolo em morse. Caso a entrada seja um espaço em branco, quer dizer que um símbolo acabou de ser digitado, e a função verifica se o código não está vazio para chamar uma função auxiliar *decodeNode*, que possui como parâmetro uma string, no caso a string que armazena o código digitado(*letterCode*) Essa função simplesmente percorre a árvore com os caracteres do símbolo em morse até encontrar o nó desejado e retorna seu valor(o símbolo em alfabeto romano). Por fim, a última opção de entrada é uma barra, que indica que uma palavra foi digitada. Nesse caso, a função só imprime um espaço. Depois disso, a função faz uma última chamada da função de decodificação, pois a última letra da mensagem não possui o identificador do espaço em branco que seu código terminou. Dessa forma, toda a mensagem é traduzida para o alfabeto romano.

Por último, o programa verifica se o parâmetro opcional “-a” foi passado, e, se sim, chama a função *preOrder*, que imprime a árvore em caminhamento pré-ordem.

2.3) Decisões de implementação

Para tornar o código mais dinâmico e de simples compreensão, algumas decisões foram tomadas baseadas na interpretação da especificação do trabalho.

a) Arquivo base para criar a árvore

A árvore deveria ser criada com base no arquivo “morse.txt”, disponibilizado na página do curso. Contudo, a especificação também definia que, se a variação do código morse fosse outra(símbolos no alfabeto com diferentes valores em morse), o programa deveria ser capaz de se adaptar. Por isso, ao invés de definir dentro da implementação da função o arquivo a ser usado(o que dificultaria uma eventual mudança), o nome do arquivo(e sua extensão) são um parâmetro da função, permitindo maior flexibilidade para realizar mudanças no código.

b) Uso da linguagem C++

Pela maior facilidade de se manipular strings e pela possibilidade de usar técnicas de programação orientada a objetos, o que torna mais simples criar operações para a estrutura de dados criada, a linguagem C++ foi escolhida ao invés da linguagem C.

c) Encapsulamento

Como foi mencionado no item 2.2),o programa possui vários métodos auxiliares. O usuário do sistema não precisa utilizar nenhum desse métodos para realizar as operações requisitadas, e, por isso, todos foram criado como *private* no contrato definido no arquivo *tree.hpp*. Dessa forma, existem apenas cinco métodos públicos: o construtor, um método *getRoot*, que retorna a raiz da árvore(utilizado para imprimir a árvore) e as três funções já explicadas que devem ser chamadas por main: *createTree*, para criar a árvore, *translate*, para traduzir mensagens, e *preOrder*, para imprimir a árvore.

Não foi criada uma função *setRoot*, pois a célula raiz deve ser vazia(chave e código vazios, apenas aponta para esquerda e direita) e é inicializada automaticamente no construtor. Também não foram criados métodos para acessar ou modificar diretamente valores de outros nós, pois ambas as operações são realizadas automaticamente dentro dos outros métodos.

d) Métodos

O entendimento foi que a árvore deveria ser capaz de realizar todas as operações necessárias(criação, tradução e impressão), já que como a árvore é quem possui os dados, deve ser ela que possui as operações que o

manipulam de forma correta, sem necessidade da função main realizar nenhuma operação, apenas chamar as funções já definidas na árvore, o que permite uma interface mais enxuta e um entendimento mais simples do código principal da função, permitindo flexibilidade ao usuário do sistema sem correr grandes riscos de erros acidentais que afetariam o funcionamento do programa.

3. Instruções de compilação e execução

A pasta enviada contém, dentro da pasta src, um arquivo chamado *main*, que pode ser executado no ambiente Linux com o comando ./main. Também está incluso na pasta um arquivo makefile, se por algum motivo for necessário compilar novamente o programa. Se houver necessidade, o makefile também é capaz de gerar arquivos executáveis para o sistema operacional Windows.

4. Análise de complexidade

O custo da função que cria a árvore é o próprio custo mais o custo da função auxiliar que cria novos nós ou retorna eles, se já existentes. O custo da criação de um nó é constante(5 atribuições), então o custo da função auxiliar é logarítmico, já que basicamente percorre uma árvore binária balanceada. A função *createTree* possui também um laço while, o que leva a um custo linear n no número de símbolos existentes no arquivo base. Portanto, seu custo geral é O(n logn).

Para a função que traduz uma mensagem, seu custo é *n logn*, já que a decodificação de um caracter é logarítmica e uma mensagem possui *n* caracteres. Portanto, seu custo total é O(n logn).

A função de impressão tem o custo de visitar cada nó, ou seja, um custo linear no número de elementos da árvore.

Portanto, definindo:

n → quantidade de elementos da árvore

m → quantidade de caracteres da mensagem a serem traduzidos

o custo total do programa é $O(n^2 \log n \, m \log m)$.

Uma observação válida é que essa avaliação está unificando todas as mensagens que o programa recebe(um programa pode receber qualquer quantidade de mensagens), pois uma análise que não as unificasse precisaria levar em conta a quantidade de caracteres de cada mensagem. Por isso, como uma simplificação, foi considerado que existe apenas uma mensagem com m caracteres(o custo geral do programa não é diferente com a outra análise, já que a árvore de pesquisa é a mesma, mas a fórmula do cálculo se tornaria mais confusa).

5. Conclusão

O trabalho, como já mencionado algumas vezes nesse documento, consistia em duas partes: a criação da árvore Trie e a decodificação do código. A criação da árvore se mostrou mais desafiadora, pois o arquivo base utilizado apresentava o símbolo do alfabeto antes do seu código em morse, enquanto na criação é preciso estar no nó certo antes de atribuir o símbolo como a chave do nó, o que tornou o algoritmo um pouco mais confuso. Além disso, a criação da função que pode tanto se movimentar quanto criar novos nós na árvore, enquanto possui um resultado final extremamente simplificado, levou a uma boa confusão.

Ao final, foi possível perceber que o código era muito mais simples do que a primeira vista. Uma revisão da implementação inicial(mesmo que esta cumprisse com os requisitos) levou ao corte de diversas funções e variáveis, que podiam ser substituídas por trechos mais curtos e simples de código, além de diversas mudanças na modularização do código.

Finalmente, o resultado final do código aparenta ser um código eficiente para a resolução do problema, com uma modularização que permite uma boa interface para o usuário e implementado de forma satisfatoriamente compacta.

6. Bibliografia

August Council. Disponível em:
< http://www.augustcouncil.com/~tgibson/tutorial/iotips.html > Acesso em 27/06/2019

Geeks for Geeks. Disponível em:
< https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/ > Acesso em 27/06/2019

cplusplus.com. Disponível em:
< http://www.cplusplus.com/reference/ > Acesso em 27/06/2019