Trabalho Prático 3

Decifrando os Segredos de Arendelle

Lucas Paulo Martins Mariz

2018055016

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

lucaspaulom@hotmail.com

Abstract: *The following article describes in a precise, succinct and qualitative way the implementation and solution of the problem proposed as a practical work of the Data Structure discipline. In short, the text stems from the implementation of binary trees and their use in data storage and retrieval.*

Resumo: *O seguinte artigo descreve de forma precisa, suscinta e qualitativa a implementação e solução do problema proposto como trabalho prático da disciplina Estrutura de Dados. Em suma, o texto decorre sobre a implementação de árvores binárias e seu uso no armazenamento e recuperação de dados.*

1. Introdução

O problema proposto por esse trabalho consiste basicamente em decifrar mensagens codificadas em Morse. Para tal, o desafio reside em utilizar estruturas de dados eficientes para armazenar e recuperar as informações necessárias para a decodificação de mensagens.

Com isso em mente, foi utilizado uma variação da Trie Binária com o auxílio de algumas regras de inserção e pesquisa, fato que facilitou consideravelmente na conclusão do trabalho.

2. Implementação

O programa foi desenvolvido na linguagem C, compilado pelo GCC versão 7.4.0 em um sistema operacional Ubuntu 18.04. Todo o código foi executado em meu notebook cujas especificações principais estão descritas abaixo:

- Intel Core i5-7200U CPU 2.5GHZ x 4

- Nvidia G-Force GTX 840M
- 8GB RAM
- SSD Kingston A400 240GB
- Ubuntu 18.04.2 LTS

Sendo assim, o trabalho consistiu em duas etapas principais:

- Montagem da árvore;
- Recuperação da informação.

Inicialmente, foi definida uma estrutura do registro, o qual deveria ser capaz de armazenar o código Morse e a letra correspondente. Essas informações são recuperadas de um arquivo ("morse.txt") que se encontra no diretório do trabalho. Logo após, foi definida a estrutura Arvore que contém apontadores para a direita e esquerda, além do registro.

```
typedef struct {
    char chave[10];
    char letra;
} Chave;

typedef struct no *Apontador;

typedef struct no {
    Apontador esq;
    Apontador dir;
    Chave registro;
} No;

typedef Apontador Arvore;
```

Figura 1: Estrutura da Árvore.

A organização da informação é definida com base no caractere Morse. Caso seja '.' o dado irá residir na subárvore da esquerda do nó atual. Caso seja '-' o dado irá residir na subárvore da direita do nó atual. A raiz da árvore permanece vazia. Dessa forma, a árvore resultante será semelhante a imagem da Figura 2.

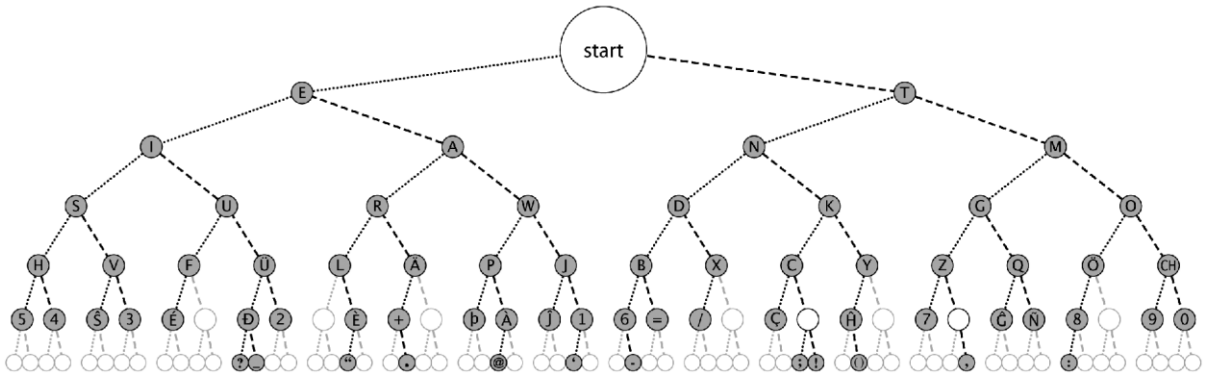

Figura 2: Montagem da árvore

A segunda parte do trabalho consistiu na decodificação das mensagens inseridas pelo usuário utilizando a árvore montada anteriormente. O método de pesquisa funciona de forma similar a inserção, ou seja, começando da raiz da árvore, caso o caractere atual seja '.' a pesquisa continua na subárvore filha esquerda do nó atual. Por outro lado, caso o caractere seja '-' a pesquisa procederá na subárvore filha da direita do nó atual. Quando todos os caracteres do código inserido forem lidos, verificamos se o nó existe, retornando o resultado da pesquisa ou um ponteiro nulo.

A entrada é formatada da seguinte forma:

--- ... / -. . - --- ... / --.- .. .

Cada letra é separada por um espaço em branco e cada palavra é separada por uma '/'. Para facilitar na manipulação dessas strings, foi utilizada a função *strtok* da biblioteca strings.h. Ela funciona de forma semelhante ao *split* do javascript, no qual pegamos uma parte da string de acordo com um caractere da nossa escolha. No caso do trabalho, foram pegas as substrings separadas pelo caractere espaço, montando assim a palavra decodificada letra por letra. Ao final da mensagem, a mesma é printada na tela e o processo se repete até o final do arquivo.

Além disso, caso o parâmetro -a seja passado na execução do programa, a árvore utilizada será impressa ao final da execução seguindo o caminhamento do tipo pré-ordem.

3. Instruções de Compilação e Execução

Como já mencionado anteriormente, recomenda-se que a execução do código ocorra em um sistema Ubuntu versão 14 ou superior contendo pelo menos a versão 7 do GNU C Compiler (GCC). Foi utilizado um makefile que se encarrega de todas as operações de compilação dos arquivos '.c'.

Para executar o programa utilizamos o seguinte comando:

./morse [-a] [< in.txt > out.txt]

O parâmetro ‘-a’ é passado caso o usuário queira ver a impressão da árvore utilizada.

4. Análise de Complexidade

4.1. Análise de tempo

Iniciaremos nossa análise com base no tempo gasto na montagem da árvore. Uma conclusão bem simples pode ser retirada acerca do processo de montagem da árvore: ela é sempre igual. Dependentemente da entrada e dos testes a serem realizados, a árvore é montada a partir dos dados contidos no arquivo *morse.txt*, fazendo com que o custo de sua montagem seja igual para toda execução. Dessa forma, assumiremos sua complexidade como uma constante, ou seja, O(1).

Por outro lado, outro importante fato a ser analisado se refere ao custo computacional da pesquisa na árvore. Pelos mesmos motivos citados anteriormente, pesquisar o caractere ‘.’, por exemplo, sempre terá um custo constante pois a árvore é imutável. Dessa forma, abstraindo para uma mensagem de n caracteres, teríamos um custo equivalente a O(n). A impressão realizada de cada caractere, por motivos semelhantes, terá um custo O(n) e a impressão da árvore tem um custo constante (O(1)).

Juntando todas essas informações, construímos a complexidade de tempo resultante do programa, que é limitada superiormente por **O(n).**

4.2. Análise de Espaço

Para armazenar a árvore, assim como na análise de tempo, temos um custo constante pois a mesma tem um tamanho fixo.

Análise do espaço ocupado para armazenar as mensagens decodificadas está diretamente relacionada a quantidade de caracteres da mensagem resultante. Em outras palavras, o custo para armazenar uma mensagem correspondente a n caracteres decodificados é da ordem de O(n).

Dessa forma, somando as informações relacionadas, temos que o programa tem uma complexidade de espaço igual a **O(n)**.

5. Conclusão

Durante o desenvolvimento do trabalho não foram encontradas tantas dificuldades devido ao fato de que toda a lógica a ser implementada foi

previamente estipulada e pensada, de forma a evitar que possíveis estruturas e pesquisas ineficientes fossem implementadas.

Sendo assim, a implementação e solução desse problema envolvendo a manipulação de árvores foi de suma importante para análise relacional para com métodos de busca sequencial. Diferentemente do trabalho anterior no qual os casos de testes eram bem maiores com o intuito de analisar profundamente a eficiência de cada algoritmo, este nos mostra o quão eficiente uma boa implementação de métodos de busca pode ser, mesmo em casos simples.

6. Bibliografia

Ziviani, N. Projeto de Algoritmos: com implementações em PASCAL e C. 2 ed. São Paulo: Thomson, 2004.

Chaimowicz, L. e Prates, R. Pesquisa Digital, disponível na turma ESTRUTURAS DE DADOS – METATURMA do Moodle UFMG 2019/1. Acesso em 1 Jun 2019.

Autor Desconhecido. C Reference. Disponível em <http://www.cplusplus.com/reference/cstring/strtok/>. Acesso em 1 Jun 2019.