Trabalho Prático 3 - Decifrando os Segredos de Arendelle

(Estudo de Árvore e Pesquisa Binária)

Caio Guedes de Azevedo Mota

2018054990

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brasil

caioguedes@ufmg.br

1. Introdução

O intuito deste trabalho prático é praticar os conceitos de Árvores Binárias e de Pesquisa Binária, implementando um programa que cria um dicionário de código morse, o armazena em uma árvore binária e traduz mensagens buscando os caracteres armazenados no dicionário.

O programa funciona de maneira bem direta. Há uma estrutura de dados principal, chamada no código de *Tree* (a *Árvore* que será o dicionário em questão), e essa essencialmente é uma conexão de outras estruturas chamadas de *Nodes* (os *Nós* da árvore). Cada nó possui um campo para o dígito (caractere original), sua tradução em código morse, e ponteiros para o nó filho à esquerda e direita. A árvore, inicialmente, é apenas um nó cabeça sem filhos.

O funcionamento do programa, de maneira geral, é o seguinte:

- ➔ Cria uma árvore, e a enche de nós vazios até os nós folha terem profundidade 5 (os caracteres desejados só vão até o 5º símbolo consecutivo em morse);
- ➔ Lê o arquivo *morse.txt* para obter informações e armazená-las no dicionário, cada dígito em seu respectivo nó na árvore;
- ➔ Percorre a entrada de dados, armazenando os caracteres em morse, traduzindo-os em letras e os imprimindo na tela
- ➔ Caso o parâmetro opcional `-a` tenha sido inserido, imprime os caracteres armazenados no dicionário em pré-ordem (raiz da subárvore, esquerda da subárvore, direita da subárvore)

Mais detalhes serão discutidos na seção de Implementação.

2. Implementação

O formato de entrada e saída de dados se dá basicamente pela própria linha de comando, mas também pode ser feito via arquivos de entrada e saída ao rodar o programa, de maneira descrita na seção Instruções de Compilação e Execução.

O primeiro problema a implementar a solução foi o de ler os caracteres do arquivo *morse.txt* e armazená-los em uma árvore. Caso os nós fossem armazenados sem que a árvore tivesse nós prévios e ela fosse sendo criada dinamicamente, dada a ordem de caracteres do arquivo para leitura, os caracteres teriam de ser rearranjados para que estivessem na ordem sempre que forem inseridos na ordem errada.

O problema foi contornado fazendo com que a árvore estivesse preenchida inicialmente por nós vazios (como descrito na seção de Introdução). Assim, ao inserir um caractere, ele já poderia fazer o caminho completo independente se caractere antecessor à ele esteja lá ou não. Assim que um caractere inserido tivesse seu lugar encontrado, basta mudar o dígito e o código morse do nó presente para o caractere em questão.

O código para a implementação da árvore vazia (*EmptyTree*) e para inserir informações nos nós da árvore (*InsertNode*) pode ser examinado à seguir:

```
void Tree::EmptyTree(Node *aux, const int &level) {
    // Ao criar a árvore vazia, apenas insere nós vazios
    // até o sexto nível (para cabeça como nível 1,
    // level = nível - 1)
    if(level <= 5) {
        if(aux->GetLeft() == nullptr) {
            aux->SetLeft(new Node());
            // Chamada recursiva para criar os nós vazios do próximo nível
            EmptyTree(aux->GetLeft(), level+1);
        }
        if(aux->GetRight() == nullptr) {
            aux->SetRight(new Node());
            // Chamada recursiva para criar os nós vazios do próximo nível
            EmptyTree(aux->GetRight(), level+1);
        }
    }
}

void Tree::EmptyTree() {EmptyTree(this->GetHead(), 0);}

void Tree::InsertNode(const char &digit, const std::string &morse) {
    // Caminha a árvore, iniciando pela cabeça
    Node *aux = this->GetHead();
    int i;
    for(i = 0; i < morse.size(); i++) {
        // Caso o caractere seja . move para a esquerda
        if(morse[i] == '.') {
            aux = aux->GetLeft();
        }
        // Caso seja - move para a direita
        else if(morse[i] == '-') {
            aux = aux->GetRight();
        }
    }

    // Insere os dados no nó final obtido pelo caminhamento
    aux->SetMorse(morse);
    aux->SetDigit(digit);
}
```

O segundo e principal problema foi, dado o dicionário pronto, como passar pela mensagem em morse e obter a tradução devida caminhando pela árvore. Primeiro, foi feita uma função *SearchFor*, que busca um dígito na árvore dado o seu equivalente em código morse, bem parecido com a função *InsereNode*:

```
char Tree::SearchFor(const std::string &morse) {
    // Inicia o caminho pelo nó cabeça
    Node *aux = this->GetHead();
    int i;
    // Itera sobre todos os caracteres presentes no código morse desejado
    for(i = 0; i < morse.size(); i++) {
        // Caso o caractere do índice i seja . vai para esquerda
        if(morse[i] == '.')
            aux = aux->GetLeft();
        // Caso seja - vai para direita
        else if(morse[i] == '-')
            aux = aux->GetRight();
    }
    // Retorna o digito correspondente
    return aux->GetDigit();
}
```

Para ler as mensagens, foi feito um loop no arquivo principal *main.cpp* que lê da entrada e passa por todos os caracteres de uma dada linha, fazendo a tradução correspondente em morse, imprimindo cada linha à medida que vai sendo traduzida. O funcionamento pode ser melhor descrito na figura a seguir, pelos comentários do código:

```
// Loop principal para ler as linhas da mensagem entrada
while(std::getline(std::cin, line)) {
    int index;
    // Variáveis para armazenar a mensagem de uma dada linha
    std::string line_message = "";
    // E o morse atual (enquanto percorre os caracteres da linha)
    std::string current_morse = "";

    // Percorre todos os caracteres da linha lida
    for(index = 0; index < line.size(); index++) {
        // Caso o caractere no índice seja . ou -, armazena-o no morse atual
        if(line[index] == '.' || line[index] == '-') {
            current_morse += line[index];
            // Caso seja o final da linha, termina de definir morse atual
            // e adiciona sua tradução na mensagem da linha
            if(line[index+1] == *line.end())
                line_message += t.SearchFor(current_morse);
        }
        // Caso o caractere no índice seja /, adiciona um espaço em branco
        // na mensagem da linha
        else if(line[index] == '/')
            line_message += ' ';
        // Caso seja outro caractere, termina de definir morse atual
        // e adiciona sua tradução na mensagem da linha
        else if(current_morse != "") {
            line_message += t.SearchFor(current_morse);
            // reinicia morse atual para passar pela próxima letra da mensagem
            current_morse = "";
        }

    }

    // Termina imprimindo a mensagem da linha a cada linha lida
    std::cout << line_message << std::endl;
}
```

3. Instruções de Compilação e Execução

O programa é compilado usando o **g++** (compilador GNU para C++). Para a compilação do programa, deve-se utilizar o Makefile incluído com os arquivos do trabalho. Com o diretório principal aberto no terminal, digitando o comando `make` o programa será compilado e estará pronto para execução.

Para a execução do programa, o comando padrão é:

```
./tp3 [-a] [< entrada.in > saída.out]
```

Sendo *entrada.in* e *saída.out* nomes genéricos para qualquer arquivo de entrada e saída opcionais, com os dados desejados, e -a um parâmetro opcional que imprime o dicionário armazenado, em pré-ordem. Para entrar com uma linha em morse para que o programa a traduza, ela deve ser escrita da seguinte maneira (mensagem exemplo):

```
--- / . .. / ... --- .. / --- / -- --- -.- ..
```

A letra desejada deverá ter os seus caracteres em morse juntos, seguida de um espaço em branco para a letra seguinte. Caso haja espaço na mensagem, ele deverá funcionar como um caractere em morse, mas sendo apenas uma barra para frente (/).

A saída imprimirá a linha seguinte, para o exemplo dado:

```
OI EU SOU O GOKU
```

O programa e os testes foram feitos em uma máquina com as seguintes especificações técnicas:

- **Processador:** Intel Core i7-7500U @ 2.7 Ghz
- **Memória RAM:** 8GB
- **Memória Secundária:** SSD Kingston A400 120GB
- **Sistema Operacional:** Linux (distribuição: Manjaro Linux 18.0.4)

4. Análise de Complexidade

4.1. Tempo

A complexidade de tempo do programa pode ser obtida analisando os laços realizados durante a execução.

O método *EmptyTree* de inicialização de árvore vazia tem complexidade constante *O(1)*, já que ele tem um número fixo de operações para sempre fazer uma árvore completa com profundidade máxima = 5.

Os métodos de inserção de nós e busca por nós (*InsertNode* e *SearchFor*) tem complexidade logarítmica *O(log n)*, com *n* sendo o tamanho da árvore, pois o caminhamento feito pela árvore é sempre dividido em 2 ao passar por um nó. Logo, são cerca de *log n* divisões até chegar no elemento procurado. Adicionalmente, o número de divisões será o tamanho do dígito desejado em código morse.

O método principal *main()* tem complexidade *O(n log n)*, pois suas estruturas, tanto a de leitura do arquivo *morse.txt* quando a de entrada da mensagem, percorrem cada elemento da entrada (iteração de complexidade linear) e, para cada iteração, chamam os métodos *InsertNode* ou *SearchFor* (complexidade logarítmica). O restante das operações do método principal são apenas operações de custo constante.

O método *PrintPreOrder* (usado para imprimir o dicionário em pré-ordem, teria complexidade linear *O(n)*, mas já que o dicionário tem tamanho fixo, esse custo sempre será constante.

No total, a complexidade do algoritmo para todos os processos é *O(n log n)*.

4.2. Espaço

A complexidade de espaço do programa é linear *O(1)*, pois a única estrutura que gasta espaço adicional com uma entrada de dados é o dicionário, no método *EmptyTree*, e a árvore do dicionário é criada com tamanho fixo (já que o máximo de caracteres em morse dos dígitos é sempre 5). O restante do processo não gasta memória adicional com base na entrada. Observação: o uso de espaço da pilha de recursão não foi considerado para a análise.

5. Conclusão

Ao final de tudo, o trabalho foi um exercício interessante dos conceitos iniciais apresentados na introdução, e mostra o poder e a simplicidade de algoritmos de busca binária, bem superiores à buscas sequenciais (principalmente ao se falar de arquivos com inúmeros registros).

A realização do trabalho foi simples, e o único problema maior foi procurar uma solução para contornar o problema da ordenação na árvore. Como explicado anteriormente, inserir os elementos desordenadamente geraria uma dificuldade desnecessária para reordená-los, e já que os locais de cada dígito na árvore eram predefinidos, bastava criar uma árvore com todos os nós vazios, e ir preenchendo-os com dados à medida que os dados de caracteres eram entrados.

6. Referências

Chaimowicz, L. e Prates, R. “Pesquisa Sequencial e Binária”, pdf disponível no Moodle da turma de Estruturas de Dados 2019/1 da UFMG. acesso em: junho/2019.

Autor Desconhecido. “Tree Traversals (Inorder, Preorder and Postorder)”, https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/ , acesso em junho/2019.