Morse Decoder – Trabalho Prático III

Estrutura de Dados

Luiz Felipe Couto Gontijo
Universidade Federal de Minas Gerais

Introdução:

Objetivo:

O objetivo do trabalho é decodificar mensagens escritas em código Morse a partir da aplicação da estrutura de dados árvore binária. A entrada caracteriza-se pela mensagem em Morse sendo que cada palavra é separada por uma barra (*/)* e cada letras separada por um espaço. A saída do programa é a mensagem decodificada e, caso aplicado o parâmetro opcional *–a,* a impressão da árvore criada em pré-ordem.

Para compilar:

Para compilar o programa basta entrar na pasta em que encontrar o arquivo *main.cpp* e executar os comandos via terminal:

make

./main.cpp <file.in> <file.out>

Em que o arquivo *file.in* é aquele que possui o código Morse a ser decodificado e *file.out* é o arquivo em que será escrita a saída do programa com a mensagem traduzida.

Pode-se também compilar com o parâmetro *–a:*

./main.cpp –a <file.in> <file.out>

Nesse caso a saída do programa será a mesma que anteriormente, porém adicionando-se ao final do arquivo *file.out* o encaminhamento em pré-ordem da árvore binária criada.

Ou ainda há a possibilidade de compilar apenas com:

./main.cpp

Em que é possível traduzir uma linha por vez. Caso queira sair do programa digite *exit* e caso queira o encaminhamento pré-ordem digite *–a* .

Visão Geral:

A estrutura do programa criado pode ser dividida em 3 partes:

- Leitura do arquivo texto *morse.txt,* que indicará as representações de cada caractere em Morse.
- Construção da árvore de acordo com os caracteres e suas respectivas representações.
- Busca na árvore binária e impressão de cada letra que, ao final, formará todo o texto decodificado.

Implementação:

O programa foi feito em C++, e para realizar o trabalho foram criadas dois tipos de estruturas de dados:

- Uma lista simplesmente encadeada denominada *MorseList,* que tem por objetivo organizar as informações retiradas do arquivo texto m*orse.txt* além de auxiliar na construção da árvore. Cada elemento da MorseList é uma *struct* do tipo *code*, que possui o nome do caractere lido(char *name*) , sua respectiva representação em *morse* (string *morse*) e, é claro, um ponteiro para o próximo elemento da lista.
- Uma árvore denominada *Tree*, onde cada um de seus nós (*struct nodes*) possui um elemento do tipo *code* denominado de *data*, além dos ponteiros para os nós tanto da esquerda quanto da direita. A classe *Tree* também possui um elemento essencial que é o ponteiro para sua *raíz*.

Etapas:

Leitura do arquivo texto e preenchimento da MorseList:

Nessa primeira etapa do programa é iniciada a lista do tipo *MorseList* e feita a leitura do *morse.txt* a partir da função *PopulateMorseList* criada no próprio *main*. Tal função lê linha por linha do arquivo e ao mesmo tempo preenche a lista criada com o nome do caractere e sua representação em *morse*. Portanto a

lista é preenchida na mesma ordem em que os elementos aparecem no arquivo texto. Segue um exemplo de preenchimento da lista:

Construção da árvore:

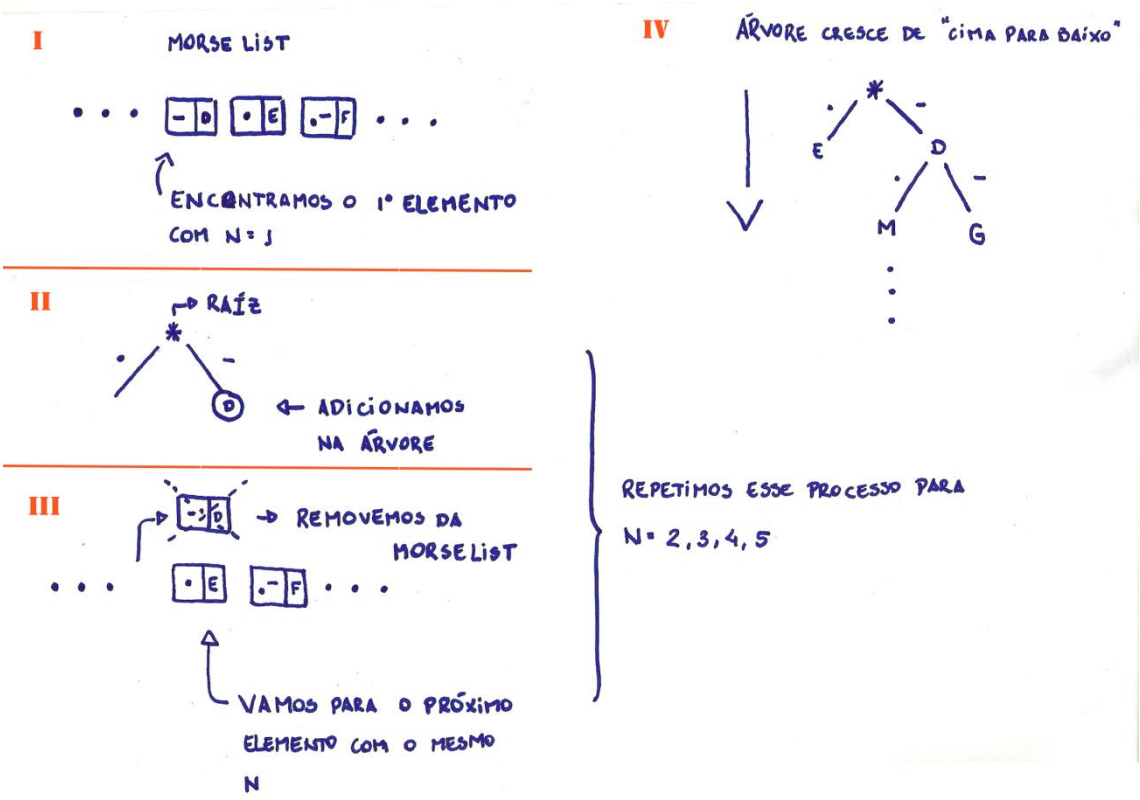
Para construir a árvore, primeiro iniciamos um objeto do tipo *Tree* que cria a arvore em si e também adiciona uma raíz (que não possui nenhum dado válido). Chamamos então a função *PopulateTree* (definida na própria classe da árvore) que recebe como argumento a *MorseList* criada e preenchida anteriormente. Dentro da função *PopulateTree* preenchemos cada nó da árvore da seguinte maneira:

• Encontramos o primeiro elemento da *MorseList* que possui tamanho de sua codificação em morse (que vamos chamar de *n*) igual a 1 (ou seja, sua codificação deve ser apenas um ponto ou uma barra).

• Partindo da raíz, se for um ponto vamos para a esquerda, e se for uma barra vamos para a direita. Como o tamanho *n* da string é igual a 1, sabemos que o procedimento acima é feito apenas uma vez, e já podemos adicionar esse nodo em sua posição correta.

• Removemos da *MorseList* o elemento adicionado anteriormente(pela chamada da função *RemoveElement*) .

• Verificamos se ainda há algum elemento com *n* = 1. Caso positivo, repetimos toda a operação acima. Caso contrário, passamos para elementos de *n* = 2.



Vale lembrar que a repetição do tópico dois acima ocorre *n* vezes, garantindo sempre que estamos construindo a árvore de "cima para baixo". Além disso, essa estratégia como um todo garante que não sejam criados nós antes do necessário, evitando problemas relacionados à alocação de memória.

Busca na árvore e impressão de cada letra:

Para imprimir os caracteres/palavras/frases, foi criada a função *Print* dentro do próprio main que é responsável por separar a entrada em cada uma de suas palavras, e mandar para a função da arvore que imprime essa palavra (*PrintWord*). Essa última separa a palavra em suas letras (ainda em *morse*) e chama finalmente a função de busca na árvore. A função de busca (*GetName*) simplesmente recebe a string e, recursivamente , itera sobre a árvore partindo da raíz. Caso o caractere *i* da string seja um ponto, vamos para a esquerda, caso seja uma barra, vamos para a direita, e assim sucessivamente, até que seja atingido o final da *string*. Basta, então, imprimir o caractere encontrado.

Análise de complexidade:

A análise de complexidade do problema pode ser feita a partir da avaliação de cada uma das etapas. A inserção na lista *MorseList* se dá na mesma ordem em que é feita a leitura do arquivo texto. Portanto, não temos comparações ou movimentações de registros (dentro da lista). Na etapa de inserir na arvore, é preciso encontrar os elementos de *MorseList* na ordem certa. Para tanto, temos um pior caso de busca igual a ***O(n)*** para o primeiro elemento. Porém, sempre removemos um elemento após inseri-lo, e, portanto, o segundo elemento a ser encontrado terá um pior caso ***O(n-1)***. E assim por diante. Isso diminui a complexidade, mas sua notação permanece **O(n)**. Para as buscas na árvore binária, temos sempre um pior caso igual a **O(log n)**.

Conclusão:

É possível concluir que o objetivo do trabalho foi atingindo. Pode-se notar que havia três partes distintas para o desenvolvimento: primeiramente era necessário organizar as informações contidas em *morse.txt,* depois era necessário construir a árvore e por último organizar a busca na árvore criada. O maior desafio foi de criar a árvore de uma forma estratégica que possibilitasse uma construção uniforme e evitasse ao mesmo tempo erros relacionados à alocação de memória.

Ao fim desse trabalho, é evidente a importância do planejamento do programa antes de sua execução. As estruturas de dados utilizadas (tanto a lista encadeada quanto a árvore binária) demandam funcionalidades específicas que devem ser implementadas não só de forma correta e funcionais, mas como também se aliando ao objetivo do programa.

Referências:

Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 3ª Edição, Cengage Learning, 2011.