Universidade de Brasília – Instituto de Ciências Exatas
Departamento de Ciências da Computação

Estruturas de Dados – Trabalho 2

Tradução de arquivos contendo código Morse

Lucas de Almeida Abreu Faria e Mateus Berardo de Souza Terra
Prof. Eduardo A. P. Alchieri
4 de Junho de 2018

1 Introdução

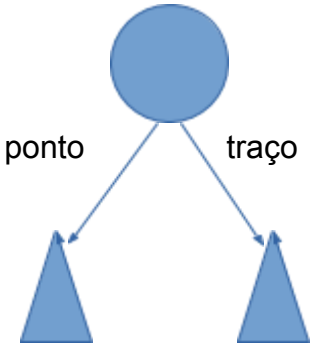
O trabalho 2 proposto para a matéria de Estruturas de Dados consiste na implementação de um tradutor de código morse utilizando uma árvore binária e uma lista para efeito de comparação de desempenho baseado nas estruturas de dados envolvidas.

Para tanto, o programa a ser desenvolvido receberá como entrada, por meio de arquivos, o código morse utilizando o formato do exemplo abaixo e uma mensagem como exemplificado após o código. Na mensagem, o caractere "*/*" marca a ocorrência de espaço entre as letras e o espaço marca a separação entre dois caracteres.

A .-.\n

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

O programa utilizará a referência para construir uma árvore binária que represente o código da seguinte forma: O nó raiz estará vazio e a cada ponto ou traço do código ele moverá o ponteiro para um dos filhos, sendo um o filho referente a ponto e o outro a traço.



Depois, utilizará esta árvore para processar a mensagem e mostrar a mensagem traduzida na tela. Além disso, o programa construirá uma lista encadeada e realizará a tradução da mensagem também, para efeito de comparação de desempenho da tradução utilizando ambos os métodos.

2 Implementação

Dividiremos a implementação do programa em tópicos, a saber:

I – Estruturas de Dados
II – Montagem da Árvore e Tradução
III – Montagem da Lista e Tradução
IV – Aferição de Desempenho

Além do detalhamento das implementações, faremos também uma rápida análise da complexidade utilizando a notação ***O***. Para a análise, consideraremos o custo de instruções da biblioteca padrão do **C** e declarações com custo igual a 1, bem como comparações.

I – Estruturas de Dados

As estruturas de dados aplicadas a resolução do problema foram árvores e listas encadeadas. A seguir detalharemos esses conceitos e sua implementação em nosso programa.

Lista Encadeada: "*Uma lista é uma estrutura de dados dinâmica. O número de nós de uma lista pode variar consideravelmente à medida que são inseridos e removidos elementos. A natureza dinâmica de uma lista pode ser comparada à natureza estática de um vetor, cujo tamanho permanece constante. (…) Cada item na lista é chamado nó e contém dois campos, um campo de informação e um campo do endereço seguinte. O campo de informação armazena o real elemento da lista. O campo do endereço seguinte contém o endereço do próximo nó na lista.*". Abaixo mostramos uma representação esquemática da lista e de sua disposição na memória.

Figura 1: Disposição de uma lista encadeada na memória

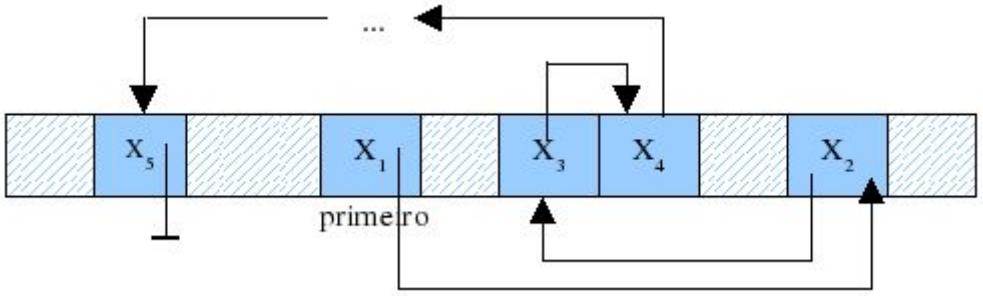
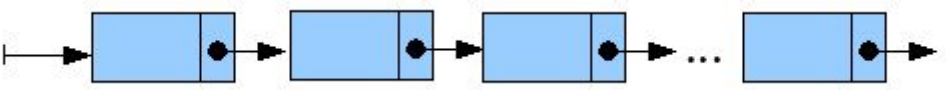


Figura 2: Representação esquemática de uma lista simplesmente encadeada



Para que fosse possível criar a lista que seria usada na hora de traduzir a mensagem, seus elementos contém o símbolo na linguagem não codificada, o código associado a ele e um ponteiro para o elemento seguinte, segundo a estrutura a seguir:

```
typedef struct elemento{
    char simbolo;
    char codigo[10];
    struct elemento *prox;
} t_elemento;
```

A lista também é uma *struct*, que possui dois ponteiros, um para seu primeiro elemento e outro para o seu último elemento.

```
typedef struct lista{
    t_elemento *inicio;
    t_elemento *fim;
} t_lista;
```

As funções implementadas foram as seguintes:

*t_lista * novaLista()* → Retorna um ponteiro para lista com os campos início e fim nulos. Não recebe nenhum argumento. Cria uma nova lista vazia. Possui uma complexidade ***O(1)***, pois não tem laços de repetição;

void insereInicio(t_lista * lista, char simbolo, char * codigo) → Não possui retorno. Recebe como argumentos um ponteiro para a lista que receberá o novo elemento, o símbolo não codificado e uma string com seu respectivo código. Insere um novo elemento no início da lista. Possui uma complexidade **O(1)**, pois não possui laços de repetição;

*char procura(char * codigo, t_lista * l)* → Recebe como argumentos o código de algum caractere e a lista em que a procura será realizada. Percorre a lista comparando o código recebido como argumento com o código armazenado em cada um dos elementos. Assim que encontra uma correspondência, retorna o caractere não codificado referente àquele código. Possui complexidade ***O(n)***, em que ***n*** é o número de elementos da lista, visto que no pior caso possível (código correspondente estar no último elemento), a lista é percorrida por completo.

```
char procura(char * codigo, t_lista * l){
    int naoAchou = 1;
    t_elemento * atual = l->inicio;
    while(naoAchou){
        if(strcmp(atual->codigo, codigo) != 0)
            atual = atual->prox;
        else
            naoAchou = 0;
    }
    return atual->simbolo;
}
```

Árvore: "*Uma árvore binária é um conjunto finito de elementos que está vazio ou é particionado em três subconjuntos*

1) *Raiz da Árvore - elemento inicial (único);*
2) *Subárvore da Esquerda - se vista isoladamente compõe uma outra árvore;*
3) *Subárvore da Direita - se vista isoladamente compõe uma outra árvore.*

A árvore pode não ter nenhum elemento (árvore vazia). A definição é recursiva e, devido a isso, muitas operações sobre árvores binárias utilizam recursão.

As árvores onde cada nó que não seja folha numa árvore binária tem subárvores esquerda e direita não vazias são conhecidas como árvores estritamente binárias." (LAUREANO, 2008). Mostramos na introdução a representação de uma árvore na memória, vamos descrever agora as funções implementadas para a criação da Árvore e as estruturas necessárias para tal:

```
typedef struct no{
    char caractere;
    struct no* ponto;
    struct no* traco;
} t_no;
```

A estrutura de cada nó contém o ponteiro para a sub-árvore esquerda e para a sub-árvore direita, cada uma representando um '.' ou um '-' a mais no código. O dado armazenado em cada nó é a letra representada pelo código gerado pelo caminho acumulado até ela.

As funções implementadas foram as seguintes:

t_no criarNo(char dado)* → Retorna o endereço de um novo nó. Recebe o caractere a ser armazenado no nó como argumento. Cria um nó com o caractere fornecido como argumento e retorna o endereço deste nó. Possui complexidade **O(1)**, pois possui custo constante.

void inserirNo(t_no raiz, char *codigo, char letra)* → Não possui retorno. Recebe o endereço do nó raiz, o ponteiro para a sequência de caracteres(não necessariamente todos) que representa o código morse e a letra. Recursivamente tenta inserir o nó na sub-árvore relativa ao primeiro caractere do código até que não haja mais pontos ou traços, quando a letra será inserida na sub-árvore correta. Possui custo **O(n)**, pois, no pior caso, percorre toda a árvore em sequência, como uma lista.

II – Montagem da Árvore e Tradução

Para a montagem da lista, o programa abre o arquivo que contém as definições do código morse e lê linha por linha do arquivo chamando a função *inserirNo* com o código e a letra como argumentos. A implementação pode ser vista abaixo:

```
t_no* montarArvore(){
    t_no *raiz = criarNo(' ');
    FILE *arvore = fopen( "./morse.txt", "r" );
    char codigo[9], letra;
    while(!feof(arvore)){
        fscanf(arvore, "%c %s\n", &letra, codigo);
        inserirNo(raiz, codigo, letra);
    }
    fclose(arvore);
    return raiz;
}
```

Para a tradução, foi utilizada uma função recursiva que percorre a árvore utilizando o código como caminho para localizar a letra. Caso não seja encontrado o caractere, a função retorna " ", que foi útil para o problema proposto, uma vez que '/' era o único caractere não contido no código mas presente na mensagem, o qual representava o espaço. Tal função tem complexidade **O(n)**, contudo pode ter complexidade **O(log n)** no melhor dos casos.

```
char traduzir(t_no *raiz, char *codigo){
    if(*codigo == '\0')
        return raiz->caractere;
    else if (*codigo != '/'){
        if(*codigo=='.')
            return traduzir(raiz->ponto, codigo+1);
        if(*codigo=='-')
            return traduzir(raiz->traco, codigo+1);
    }
    return ' ';
}
```

III – Montagem da Lista e Tradução

A fim de implementar a decodificação do Código Morse por meio de listas, foram criados os seguintes algoritmos:

*t_lista * dicionario()* → Retorna um ponteiro para a lista contendo a definição do Código Morse. Lê o arquivo *morse.txt* linha a linha até que o retorno da função *fscanf* acuse o fim do arquivo. Insere na lista o símbolo não codificado e seu respectivo código após ler cada linha. Possui uma complexidade ***O(n)***, em que ***n*** é o número de linhas do arquivo lido, e consequentemente o número de vezes que o laço while será executado;

```
t_lista * dicionario(){
    int lido = 1;
    char simbolo, codigo[10];
    t_lista * lista = novaLista();
    FILE * morse = fopen("morse.txt", "r");
    while(lido != -1){
        lido = fscanf(morse, "%c %s\n", &simbolo, codigo);
        if(lido != -1)
            insereInicio(lista, simbolo, codigo);
    }
    fclose(morse);
    return lista;
}
```

Após criada a lista, parte-se para a tradução na mensagem contida no arquivo *mensagem.txt*:

*void decodificaLista(t_lista * tradutor)* → Recebe como argumento a lista que contém a definição do código morse, não possui retorno. Abre o arquivo *mensagem.txt*, lê cada conjunto de caracteres até encontrar um espaço. Caso a string lida seja igual a "/", imprime um espaço em branco na saída padrão. Caso contrário, significa que a string lida é um caractere codificado, então a função *procura* é chamada para que seja achado o caractere correspondente ao código. Imprime-se então esse caractere na saída padrão. Ao final, o arquivo *mensagem.txt* é devidamente fechado. Possui uma complexidade $O(n^2)$, pois laço *while* é repetido ***n*** vezes, em que ***n*** é o número de códigos que o arquivo possui, e, para cada código, a lista que contém a definição do código morse é percorrida até que seja encontrada uma correspondência.

```
void decodificaLista(t_lista * tradutor){
        int lido = 1;
        char codigo[10];
        FILE * mensagem = fopen("mensagem.txt", "r");
        while(lido != -1){
                lido = fscanf(mensagem, "%s ", codigo);
                if(lido != -1){
                        if(strcmp(codigo, "/") != 0)
                                putchar(procura(codigo, tradutor));
                        else
                                putchar(' ');
                }
        }
        fclose(mensagem);
}
```

IV – Aferição do Desempenho

Para comparar o desempenho dos dois métodos, utilizamos o método *clock_gettime* da biblioteca *time* para ler o tempo de início e de final de execução e dessa forma calcular o tempo necessário para tradução. Não foi comparado o tempo de geração das Estruturas, mas apenas para tradução da mensagem. Para que tivéssemos tempo de processamento suficiente para medir a diferença, utilizamos também um arquivo que continha uma mensagem estendida criado a partir do arquivo *mensagem.txt*.

```
   clock_gettime(CLOCK_REALTIME, &requestStart);
   mensagem = fopen( "./mensagem.txt", "r" );
   while(fscanf(mensagem, "%s ", codigo) != -1){
      putchar(traduzir(raiz, codigo));
   }
   fclose(mensagem);
   clock_gettime(CLOCK_REALTIME, &requestEnd);
   double accum = ( requestEnd.tv_sec - requestStart.tv_sec )+( requestEnd.tv_nsec -
requestStart.tv_nsec )*ONE_OVER_BILLION;
   printf( "%lf\n", accum );

   t_lista * tradutor = dicionario();
   clock_gettime(CLOCK_REALTIME, &requestStart);
         decodificaLista(tradutor);
         clock_gettime(CLOCK_REALTIME, &requestEnd);

   accum = ( requestEnd.tv_sec - requestStart.tv_sec )+( requestEnd.tv_nsec -
requestStart.tv_nsec )*ONE_OVER_BILLION;
   printf( "%lf\n", accum );
```

3 Testes

Durante o desenvolvimento, utilizamos alguns testes baseados em outros arquivos .txt que criamos, contendo apenas “A .-.” no arquivo para a criação da árvore e da lista que conteriam a definição do código morse, e “.-. .-. .-. / .-. .-. / .-.” no arquivo que deveria ser traduzido. Após esses, realizamos testes com os arquivos originais (*morse.txt e mensagem.txt)*, que serão transcritos abaixo:

Teste 1: Árvore

```
int main(){
   t_no *raiz = montarArvore();
   mensagem = fopen( "./mensagem.txt", "r" );
   while(fscanf(mensagem, "%s ", codigo) != -1)
        putchar(traduzir(raiz, codigo));
   fclose(mensagem);

   return 0;
}
```

Teste 2: Lista

```
int main(){
        t_lista * tradutor = dicionario();
        decodificaLista(tradutor);

   return 0;
}
```

4 Conclusão

O trabalho foi de grande utilidade para a compreensão e treinamento do uso de árvores e a comparação de seu desempenho em relação às listas encadeadas para a realização de uma mesma tarefa. O conhecimento adquirido durante as aulas foi suficiente para implementar corretamente as funções de criação e manipulação das árvores. Houveram algumas dificuldades na implementação da função de tradução usando árvores, devido ao cuidado que se deve ter ao tratar os espaços em branco e "/" contidos no arquivo *mensagem.txt*. Foi possível observar também que o desempenho do algoritmo que utiliza árvores é mais eficiente do que aquele usando listas, porque a busca nas árvores construídas só possui complexidade igual à das listas no pior caso possível, sendo mais rápida nos demais casos, chegando a custar cerca de metade do tempo necessário para realizar o mesmo procedimento. Para facilitar o desenvolvimento de código compartilhado, utilizamos o GitHub, e o trabalho está disponível no repositório https://github.com/lucasaafaria/Trabalhos-ED/tree/master/Trabalho2 .

5 Bibliografia

Tenenbaum, A. A; Langsan Y.; Augenstein M. Estruturas de Dados Usando C. Prentice-Hall, 1995.

FARIAS, R. Estrutura de Dados e Algoritmos. **http://www.cos.ufrj.br**, 2009. Disponível em: < http://www.cos.ufrj.br/~rfarias/cos121/aula_11.html >. Acesso em: 4 jun. 2018.

Laureano, M. Estrutura de Dados com Algoritmos e C. Brasport, 2008. Disponível em: < http://www.mlaureano.org/livro/livro_estrutura_conta.pdf > Acesso em 5 jun. 2018