Universidade Federal de Ouro Preto – UFOP
Instituto de Ciências Exatas e Biológicas – ICEB
Departamento de Computação – DECOM
Disciplina: Algoritmos e Estruturas de Dados I – CIC102
Professor: David Menotti (menottid@gmail.com)

Trabalho Prático 2

Listas / Filas / Pilhas

Valor: 1,2 pontos (12% da nota total) **Data de entrega: 22/10/2008**
Documentação em Latex: 0,1 pontos extra

Este trabalho prático é divido em 3 partes principais. A primeira parte está relacionada com Listas duplamente encadeadas, enquanto que a segunda e terceira estão direcionadas à pilhas e filas, respectivamente.

1. Matrizes Esparsas

(Utilização de Listas por meio de Estruturas Auto-Referencias ([3] apud [2]))

Objetivos

Consiste em concretizar os conceitos de Listas implementadas por encadeamento através de uma aplicação: Matrizes esparsas.

Descrição

Matrizes esparsas são matrizes nas quais a maioria das posições é preenchida por zeros. Para essas matrizes, podemos economizar um espaço significativo de memória se apenas os termos diferentes de zero forem armazenados. As operações usais sobre essas matrizes (somar, multiplicar, inverter, pivotar) também podem ser feitas em tempo muito menor se não armazenarmos as posições que contêm zeros.

Uma maneira eficiente de presentar estruturas com tamanho variável e/ou desconhecido é com o emprego de alocação encadeada, utilizando listas. Vamos usar essa representação para armazenar as matrizes esparsas. Cad coluna da matriz será representada por uma **lista linear circular** com uma **célula cabeça**. Da mesma maneira, cada linha da matriz também será representada por uma lista linear circular com uma célula cabeça. Cada célula da estrutura, além das células cabeça, representará os termos diferentes de zero da matriz e deverá ser como no código abaixo:

```
typedef struct Celula {
  Celula direita, abaixo;
  int linha, coluna;
  double valor;
} Celula;
```

O campo abaixo deve ser usado para referenciar o elemento diferente de zero na mesma coluna. O campo direita deve ser usadao para referenciar o próximo elemento diferente de zero na mesma linha. Dada uma matriz A, para um valor A(i,j) diferente de zero, deverá haver uma célula com o campo valor contendo A(i,j), o campo linha contendo i e o campo coluna contendo j. Essa célula deverá pertencer a lista ciruclar da linha i e também deverá pertencer à lista circular da coluna j. Ou seja, cada célula pertencerá a duas listas ao mesmo tempo. Para diferenciar as células cabeça, coloque -1 nos campos linha e coluna dessas células.

Considere a seguinte matriz esparsa:

$$A = \begin{pmatrix} 50 & 0 & 0 & 0 \\ 10 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 \\ -30 & 0 & -60 & 5 \end{pmatrix}$$

A representação da matriz A pode ser vista na Figura 1. Com essa representação, uma matriz esparsa *m* x *n* com *r* elementos diferentes de zero gastará ($m + n + r$) células. É bem verdade que cada célula ocupa vários bytes na memória; no entanto, o total de memória usado será menor do que as *m* x *n* posições necessárias para representar a matriz toda, desde que *r* seja suficientemente pequeno.

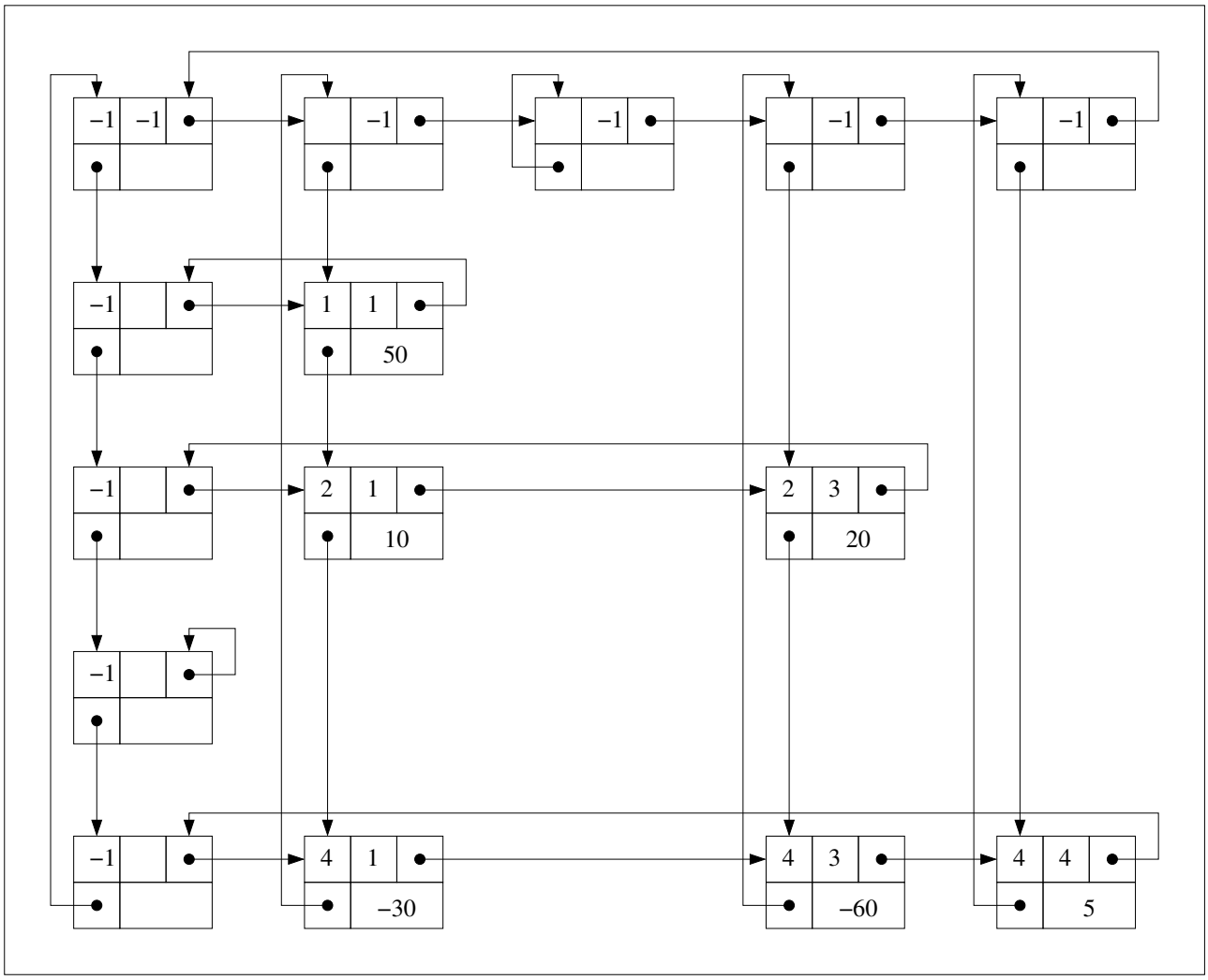

Figura 1. Exemplo de Matriz Esparsa.

Dada a representação de listas duplamente encadeadas, o trabalho consiste em desenvolver em C/C++ um tipo abstrato de dados Matriz com as seguintes operações, conforme esta especificação:

a) void imprimeMatriz()

Esse método imprime (uma linha da matriz por linha na saída) a matriz A, inclusive os elementos iguais a zero.

b) void leMatriz()

Esse método lê, de algum arquivo de entrada, os elementos diferentes de zero de uma matriz e monta a estrutura especificada anteriormente. Considere que a entrada conssite dos valores de m e n (número de linhas e de colunas da matriz) seguidos de triplas (*i*, *j*, *valor*) para os elementos diferentes de zero da matriz. Por exemplo, para a matriz anterior, a entrada seria:

```
4,   4
1,   1,   50.0
2,   1,   10.0
2,   3,   20.0
4,   1,  -30.0
4,   3,  -60.0
4,   4,   -5.0
```

c) TMatriz somaMatriz(TMatriz A, TMatriz B)

Esse método recebe como parâmetros as matrizes A e B, devolvendo uma matriz C que é a soma de A com B.

d) TMatriz multiplicaMatriz(TMatriz A, TMatriz B)

Esse método recebe como parâmetros as matrizes A e B, devolvendo uma matriz C que é o produto de A por B.

Para inserir e retirar células das listas que formam a matriz, crie métodos especiais para esse fim. Por exemplo

void insere(int i, int j, double v)

para inserir o valor *v* na linha *i*, coluna *j* da matriz A será útil tanto na função *leMatriz* quando na função *somaMatriz*.

As matrizes a serem lidas para testas as funções são:

a) a mesma matriz mostrada no enunciado deste problema e ilustrada pela Figura 1.

b) $\begin{pmatrix} 50 & 30 & 0 & 0 \\ 10 & 0 & -20 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -5 \end{pmatrix}$

c) $\begin{pmatrix} 3 & 0 & 0 \\ 0 & -1 & 0 \end{pmatrix}$

É obrigatório o uso de alocação dinâmica de memória para implementar as listas de adjacência que representam as matrizes. A análise de complexidade deve ser feita em função de *m*, *n* (dimensões da matriz) e *r* (número de elementos diferentes de zero).

As funções deverão ser testados utilizando-se um programa main **similiar** ao apresentado abaixo:

```
void main(void)
{
  TMatriz A, B,C;

  A.leMatriz(); A.imprimeMatriz();
  B.leMatriz(); B.imprimeMatriz();
  C = somaMatriz(A,B); C.imprimeMatriz();
  B.leMatriz();
  A.imprimeMatriz();B.imprimeMatriz();
  C = somaMatriz(A,B); C.imprimeMatriz();
  C = multiplicaMatriz(A,B); C.imprimeMatriz();
  C = multiplicaMatriz(B,B);
  A.imprimeMatriz(); B.imprimeMatriz(); C.imprimeMatriz();
  …
}
```

2. Avaliador de expressoes aritméticas (forma infixa)

(extraído e modificado de [1])

Objetivos

Consiste em concretizar os conceitos de pilhas de um relevante tópico de ciência da computação: avaliador de expressões aritméticas usando vários tipos de pilha.

Descrição

Considere a soma de *A* mais *B*. Imaginamos a aplicação do ***operador*** "+" sobre os ***operandos*** *A* e *B*, e escrevemos a soma como *A* + *B*. Essa representação particular é chamada ***infixa***. Existem duas notações alternativas para expressar a soma de *A* e *B* usando os símbolos *A*, *B* e +. São elas:

+ A B prefixa
A B + posfixa

Os prefixos "pre", "pos" e "in" referem-se à posição relativa do operador em relação aos dois operandos. Na notação prefixa, o operador precede os dois operandos; na notação posfixa, o operador é introduzido depois dos dois operandos e; na notação infixa, o operador aparece entre os dois operandos. Na realidade, as notações prefixa e posfixa não são tão incômodas de usar como possam parecer a princípio. Por exemplo, uma função em C para retornar a soma dos dois argumetnos, *A* e *B*, é chamada por *add*(*A*,*B*). O operador *add* precede os operandos *A* e *B*.

Examinemos agora alguns exemplos adicionais. A avaliação da expressão A + B * C, conforme escrita em notação infixa, requer o conhecimento de qual das duas operações, + ou *, deve ser efetuada em primeiro lugar. No caso de + e *, "sabemos" que a multiplicação deve ser efetuada antes da adição (na ausência de parênteses que indiquem o contrário). Sendo assim, interpretamos A + B * C como A + (B * C), a menos que especificado de outra forma. Dizemos, então, que a multiplicação tem precedência sobre a adição. Suponha que queiramos reescrever A + B * C em notação posfixa. Aplicando as regras da precedência, converteremos primeiro a parte da expressão que é avaliada em primeiro lugar, ou seja a multiplicação. Fazendo essa conversão em estágios, obteremos:

A + (B * C)	Parênteses para obter ênfase
A + (B C *)	Converte a multiplicação
A (B C *) +	Converte a adição
A B C * +	Forma posfixa

As únicas regras a lembrar durante o processo de conversão é que as operações com a precedência mais alta são convertidas em primeiro lugar e que, depois de uma parte da expressão ter sido convertida para posfixa, ela deve ser tratada como um único operando. Examine o mesmo exemplo com a precedência de operadores invertida pela inserção deliberada de parênteses.

(A + B) *C	Forma infixa
(A B +) * C	Converte a adição
(A B +) C *	Converte a multiplicação
A B + C *	Forma posfixa

Nesse exemplo, a adiação é convertida antes da multiplicação por causa dos parênteses. Ao passar de (A + B) * C para (A B +) * C, A e B são os operandos e + é o operador. Ao passar de (A B +) * C para (A B +) C *, (A B +) e C são operandos e * é o operador. As regras para converter da forma infixa para a posfixa são simples, desde que você conheça as regras de precedência.

Consideramos cinco operações binárias: adição, substração, multiplicação, divisão e exponenciação. As quatro primeiras estão disponíveis em C e são indicadas pelos conhecidos operadores +, -, * e /. A quinta oepração, exponenciação, é representada pelo operador $. O valor da expressão A $ B é A elevado à potência B, de maneira que 3 $ 2 é 9. Veja a seguir a ordem de precedência (da superior para a inferior) para esses operadores binários:

Exponenciação
Multiplicação / Divisão
Adição / Substração

Quando operadores sem parênteses e da mesma ordem de precedência são avaliados, pressupõe-se a ordem da esquerda para a direita, exceto no caso da exponenciação, em que a ordem é supostamente da direita para a esquerda. Sendo assim, A + B + C significa (A + B) + C, enquanto A $ B $ C significa A $ (B $ C). Usando parênteses, podemos ignorar a precedência padrão.

Apresentamos os seguintes exemplos adicionais de conversão da forma infixa para a posfixa. Procure entender cada um dos exemplos (e fazê-los por contra própria) antes de prosseguir com o restante deste trabalho.

Forma Infixa	Forma Posfixa
A + B	A B +
A + B - C	A B + C -
(A + B) * (C - D)	A B + C D - *
A $ B * C - D + E / F / (G + H)	AB $ C * D - E F / G H + / +
((A + B) * C - (D - E)) $ (F + G)	A B + C * D E - - F G + $
A - B / (C * D $ E)	A B C D E $ * / -

As regras de precedência para converter uma expressão da forma infixa apra a prefixa são idênticas. A única mudança da conversão posfixa é que o operador é posicionado antes dos operadnos, em vez de depois deles. Apresentamos as formas prefixas das expressões anteriores.

Universidade Federal de Ouro Preto – UFOP
Instituto de Ciências Exatas e Biológicas – ICEB
Departamento de Computação – DECOM
Disciplina: Algoritmos e Estruturas de Dados I – CIC102
Professor: David Menotti (menottid@gmail.com)

Forma Infixa	Forma Prefixa
A + B	+ A B
A + B - C	- + A B C
(A + B) * (C - D)	* + A B - C D
A $ B * C - D + E / F / (G + H)	+ - * $ A B C D / / E F + G H
((A + B) * C - (D - E)) $ (F + G)	$ - + A B C – D E + F G
A - B / (C * D $ E)	- A / B * C $ D E

Observe que a forma prefixa de uma expressão complexa não representa a imagem espelhada da forma posfixa, como podemos notar no segundo exemplo apresentado anteriormente, A + B – C. De agora em diante, consideraremos somente as transformações posfixas e deixaremos para o leitor, como exercício, a maior parte do trablaho envolvendo a forma prefixa.

Uma questão imediatamente óbvia sobre a forma posfixa de uma expressão é a ausência de parênteses. Examine as duas expressões, A + (B * C) e (A + B) * C. Embora os parênteses em uma das expressões sejam supérfluos (por convenção, A + B * C = A + (B * C)), os parênteses na segunda expressão são necessários para evitar confusão com a primeira. As formas posfixas dessas expressões são:

Forma Infixa	Forma Posfixa
A + (B * C)	A B C * +
(A + B) * C	A B + C *

Não existem parênteses nas duas expressões transformadas. A ordem dos operadores nas expressões posfixas determina a verdadeira ordem das operações, ao avaliar a expressão, tornando desnecessário o uso de parênteses.

Ao passar da forma infixa para a posfixa, abrimos mão da possibilidade de observar rapidamente os operandos associados a um determinado operador. Entretanto, obtemos uma forma não-ambígua da expressão original sem o uso dos incômodos parênteses. Na verdade, a forma posfixa da expressão orginal poderia parecer mais simples, não fosse o fato de que ela parece difícil de avaliar. Por exemplo, como saberemos que, se A = 3, B = 4 e C = 5, nos exemplos anteriores, então 3 4 5 * + equivalerá a 23 e 3 4 + 5 * equivalerá a 35?

Considerando o conteúdo exposto acima, pede para criar um programa que seja capaz de avaliar uma expressão infixa. Para criar tal programa, sugere-se, a princípio, criar uma função que seja capaz de converter uma expressão da forma infixa para a forma posfixa. Uma vez que a expressão posfixa é obtida, pode-se facilmente implementar, através de pilhas, um programa para avaliar a expressão posfixa, que é, em essência, equivalente a expressão infixa original.

O programa a ser construído deverá, antes de avaliar a expressão infixa, verificar se a expressão é sintáticamente válida. Ou seja, se o número de operandos, operadores e parênteses é correto, além da sua posição dentro da expressão (que pode ser representada por uma cadeia de caracteres). Por exemplo, a seguinte expressão é inválida: (3 (+ 5)), pois embora tenhamos o número correto de parênteses, operandos e operadores, a posição do segundo parênteses de abertura é inválida. Considere que as expressões serão compostas por números de um único algarismo (ou seja, 0, 1, 2, ..., 8 e 9), as cinco operações apresentadas anteriores, e parenteses. O programa deve detectar automaticamente a ordem de precedência das operações (sugestão: através da comparação do elemento no topo da pilha e o operador lido). Ao final, o programa deverá apresentar o valor numérico da expressão avalida, por exemplo, para a expressão 3 + 5 * 14 / (4 + 3) o valor a ser apresentado será 13.

Universidade Federal de Ouro Preto – UFOP
Instituto de Ciências Exatas e Biológicas – ICEB
Departamento de Computação – DECOM
Disciplina: Algoritmos e Estruturas de Dados I – CIC102
Professor: David Menotti (menottid@gmail.com)

3. Filas, Simulação

(Utilização de Filas por meio de encadeamento ([4] apud [2]))

Objetivos

Consiste em concretizar os conceitos de Filas implementadas por encadeamento através de uma aplicação: Simulaçao de de aterrisagem e decolagem em um aeroporto.

Descrição

Suponha um aeroporto que possui três pistas, numeradas como 1, 2 e 3. Existem quatro "prateleiras" de espera para aterrisagem, duas para cada uma das pistas 1 e 2. Aeronaves que se aproximam do aeroporto devem integrar-se a uma das prateleiras (filas) de espera, sendo que essas filas devem procurar matner o mesmo tamanho. Assim que um avião entra em uma fila de aterissagem, ele recebe um número de identificação ID e outro número inteiro que indica a quantidade de unidades de tempo que o avião pode permanecer na fila antes que ele tenha de descer (do contrário, seu combustível termina e ele cai).

Existem também filas para decolagem, uma para cada pista. Os aviões que chegam nessas filas também recebem uma identificação ID. Essas filas devem procurar manter o mesmo tamanho.

A cada unidade de tempo, de zero a três aeronaves podem chegar nas filas de decolagem, e de zero a três aeronaves podem chegar nas prateleiras. A cada unidade de tempo, cada pista pode ser usada para umpouso ou uma decolagem. A pista 3 em geral só é usada para decolagens, a não ser que um dos aviões nas prateleiras fique sem combustível, quando então ela deve ser imediatamente usada para pouso. Se apensar uma aeronave está com falta de combustível, ela pousara na pista 3; se mais de um avião estiver nessa situação, as outras pistas poderão ser utilizadas (a cada unidade de tempo no máximo três aviões poderão estar nessa desagradável situação).

Utilize inteiros pares (ímpares) sucessivos para a ID dos aviões chegando nas filas de decolagem (aterrisagem). A cada unidade de tempo, assuma que os aviões entram nas filas antes que aterrisagens ou decolagens ocorram. Tente projetar um algoritmo que não permita o crescimento excessivo das filas de aterrisagem ou decolagem. Coloque os aviões sempre no final das filas que não devem ser reordenadas.

A saída do programa deverá indicar o que ocorre a cada unidade de tempo. Periodicamente imprima:

a) o conteúdo de cada fila;
b) o tempo médio de espera para decolagem;
c) o tempo médio de espera para aterrissagem;
d) o número de aviões que aterrissam sem reserva de combustível.

Os itens b e c devem ser calculados para os aviões que já decolaram ou pousaram respectivamente. A saída do programa deve ser auto-explicativa e fácil de entender.

A entrada poderia ser criada manualmente, mas o melhor é utilizar um gerador de números aleatórios. Para cada unidade de tempo, a entrada deve ter as seguintes informações:

a) número de aviões (zero a três) chegando nas filas de aterrissagem com respectivas reservas de combustível (de 1 a 20 em unidades de tempo);
b) número de aviões (zero a três) chegando nas filas de decolagem;

Universidade Federal de Ouro Preto – UFOP
Instituto de Ciências Exatas e Biológicas – ICEB
Departamento de Computação – DECOM
Disciplina: Algoritmos e Estruturas de Dados I – CIC102
Professor: David Menotti (menottid@gmail.com)

O que deve ser entregue para todas as 3 partes

- Código fonte do programa em C ou C++ (bem identada e comentada).
- Documentação do trabalho. Entre outras coisas, a documentação deve conter:
  1. Introdução: descrição do problema a ser resolvido e visão geral sobre o funcionamento do programa.
  2. Implementação: descrição sobre a implementação do programa. Deve ser detalhada a estrutura de dados utilizada (de preferência com diagramas ilustrativos), o funcionamento das principais funções e procedimentos utilizados, o formato de entrada e saída de dados, bem como decisões tomadas relativas aos casos e detalhes de especificação que porventura estejam omissos no enunciado.
  3. Estudo de Complexidade: estudo da complexidade do tempo de execução dos procedimentos implementados e do programa como um todo (notação O).
  4. Listagem de testes executados: os testes executados devem ser simplesmente apresentados.
  5. Conclusão: comentários gerais sobre o trabalho e as principais dificuldades encontradas em sua implementação.
  6. Bibliografia: bibliografia utilizada para o desenvolvimento do trabalho, incluindo sites da Internet se for o caso
  7. Em Latex: Caso o trabalho seja elaborado/escrito em latex, ganha-se 20% de pontos extra, ou seja 0,1 pontos.
  8. Formato: mandatoriamente em PDF (http://www.pdf995.com/).

Obs1: Consulte as dicas do Prof. Nívio Ziviani de como deve ser feita uma boa implementação e documentação de um trabalho prático: http://www.dcc.ufmg.br/~nivio/cursos/aed2/roteiro/

Obs2: Veja modelo de como fazer o trabalho em latex:
http://www.decom.ufop.br/prof/menotti/aedI/tps/modelo.zip
(caso alguém desenvolva um modelo similar em word, favor me enviar)

Como deve ser feita a entrega:

A entrega DEVE ser feita pelo Moodle na forma de um **único** arquivo zipado, contendo o código, os arquivos e a documentação. Como o trabalho é composto de 3 partes, crie um **diretório/pasta** para cada uma das partes. Também deve ser entregue a documentação impressa na próxima aula (teórica ou prática) após a data de entrega do trabalho.

Comentários Gerais:

- Comece a fazer este trabalho logo, enquanto o problema está fresco na memória e o prazo para terminá-lo está tão longe quanto jamais poderá estar;
- Clareza, identação e comentários no programa também vão valer pontos;
- O trabalho é individual (grupo de UM aluno);
- Trabalhos copiados (e FONTE) terão nota zero;
- Trabalhos entregue em atraso serão aceitos, todavia a nota atribuída ao trabalho será zero
- Evite discussões inócuas com o professor em tentar postergar a data de entrega do referido trabalho.

Referências:

[1] Aaron M. Tenenbaum, Yedidyah Langsam, Moshe J. Augenstein, *Estruturas de Dados Usando C*, Makron Books/Pearson Education, 1995.

[2] N. Ziviani, F.C. Botelho, Projeto de Algoritmos com implementações em Java e C++, Editora Thomson, 2006.

[3] F.C. Botelho, Comunicação Pessoal, Belo Horizonte, MG, Brasil,, 2003.

[4] J.N.C. Árabe, Comunicação Pessoal, Belo Horizonte, MG, Brasil, 1992