Estrutura de Dados com Algoritmos e C

Escrever um livro não é uma tarefa fácil, nunca foi. Toma tempo, exige pesquisa e dedicação. Como as editoras não desejam mais publicar este título, nada mais natural que tentar comercializa-lo na forma de arquivo (semelhante a um ebook). Mas, esta atividade também toma tempo e exige dedicação.

Pensando assim, resolvi liberar este livro para consulta pública, se você acha que este livro te ajudou e quiser colaborar comigo, passe numa lotérica, e deposite o valor que achar que deve.

Terminou de pagar a conta de luz, telefone ou água e sobrou troco, você pode depositar na minha conta. Eu acredito que nós dois podemos sair ganhando, você porque teve acesso a um bom material que te ajudou e eu como incentivo a continuar escrevendo livros. Caso de certo, talvez os próximos livros nem sejam publicados por uma editora e estejam liberados diretamente para sua consulta.

Qualquer valor depositado será direcionado para a conta poupança do meu filho, para quando ele estiver na maioridade ter recursos para começar um negócio próprio, financiar seus estudos, etc.

Dados para depósito:

Marcos Aurelio Pchek Laureano.
Banco: 104 - Caixa Econômica Federal
Agência: 1628
Operação: 001
Conta: 6012-2

Curitiba, 13 de março de 2012.

Estrutura de Dados com Algoritmos e C

Marcos Laureano

Editor: Sergio Martins de Oliveira
Diretora Editorial: Rosa Maria Oliveira de Queiroz
Assistente de Produção: Marina dos Anjos Martins de Oliveira
Revisão de Texto: Maria Helena A. M. Oliveira
Editoração Eletrônica: Abreu's System LTDA.
Capa: Use Design

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

BRASPORT Livros e Multimídia Ltda.
Rua Pardal Mallet, 23 – Tijuca
20270-280 Rio de Janeiro-RJ
Tels. Fax: (21) 2568.1415/2568.1507/2569.0212/2565.8257
e-mails: brasport@brasport.com.br
vendas@brasport.com.br
editorial@brasport.com.br
site: **www.brasport.com.br**

Filial
Av. Paulista, 807 – conj. 915
01311-100 – São Paulo-SP
Tel. Fax (11): 3287.1752
e-mail: filialsp@brasport.com.br

Aos meus genitores Luiz Otávio (in memoriam) e Natália.

Agradecimentos

Agradeço à Brasport por mais esta oportunidade de publicar um livro sobre um tema em que vários autores já trabalharam (é claro que este livro tem um diferencial em relação aos demais).

Aos meus colegas professores e alunos que ajudaram a evolução deste material nos últimos anos.

E para uma linda flor, que, além de alegrar os meus dias, corrigiu os assassinatos à língua portuguesa.

Sobre o autor

Marcos Laureano é tecnólogo em Processamento de Dados pela ESEEI, Pós-graduado em Administração pela FAE Business School e Mestre em Informática Aplicada pela Pontifícia Universidade Católica do Paraná. Doutorando em Informática Aplicada pela Pontifícia Universidade Católica do Paraná.

Trabalha com programação em C no ambiente Unix (AIX/HP-UX) desde 1997 e Linux desde 2000, sendo especialista em segurança de sistemas operacionais.

É professor de graduação e pós-graduação, tendo lecionado em várias instituições nos últimos anos.

É autor de vários guias de utilização/configuração de aplicativos para os ambientes Unix e Linux. Possui artigos e livros publicados sobre programação, sistemas operacionais e segurança de sistemas. Dentre eles, **Programando em C para Linux, Unix e Windows**, também publicado pela Brasport.

Atualmente leciona disciplinas relacionadas à segurança, programação, redes de computadores e sistemas operacionais em cursos de graduação e pós-graduação do Centro Universitário Franciscano (UNIFAE) e atua como consultor na área de projetos de desenvolvimento e segurança de sistemas.

O autor pode ser contactado pelo e-mail **marcos@laureano.eti.br** ou através de sua página **www.laureano.eti.br**, onde está disponível vasto material sobre programação, segurança de sistemas e sistemas operacionais.

Sobre o Livro

O crescimento dos cursos tecnológicos específicos e com curta duração (2 a 3 anos) gerou uma demanda de livros que tratam diretamente o assunto de maneira clara e eficiente.

Este livro é indicado aos cursos tecnológicos, para estudantes e profissionais que precisem dominar os conceitos e os algoritmos de forma rápida e precisa.

O material foi preparado com a experiência do autor em lecionar a disciplina, somada à sua experiência profissional. Outros professores e coordenadores de cursos foram consultados; com isto, este material tem os assuntos pertinentes à área e pode ser adotado tranqüilamente em cursos de 40 ou 80 horas de Estrutura de Dados ou Programação de Computadores.

Os algoritmos podem ser aplicados e convertidos para qualquer linguagem de programação, e os programas em C são simples e objetivos, facilitando o entendimento dos estudantes e profissionais que não dominam totalmente esta linguagem.

Sumário

Lista de Programas

Lista de Tabelas

Lista de Algoritmos

Lista de Figuras

1. Estrutura de Dados

"Não existe vitória sem sacrifício!"
Filme Transformers

Um computador é uma máquina que manipula informações. O estudo da ciência da computação inclui o exame da organização, manipulação e utilização destas informações num computador. Conseqüentemente, é muito importante entender os conceitos de organização e manipulação de informações.

A automatização de tarefas é um aspecto marcante da sociedade moderna, e na ciência da computação houve um processo de desenvolvimento simultâneo e interativo de máquinas (hardware) e dos elementos que gerenciam a execução automática (software) de uma tarefa.

Nesta grande evolução do mundo computacional, um fator de relevante importância é a forma de armazenar as informações, já que, informática é a ciência da informação. Então de nada adiantaria o grande desenvolvimento do hardware e do software se a forma de armazenamento e tratamento da informação não acompanhasse esse desenvolvimento. Por isso a importância das estruturas de dados, que nada mais são do que formas otimizadas de armazenamento e tratamento das informações eletronicamente.

As estruturas de dados, na maioria dos casos, baseiam-se nos tipos de armazenamento vistos dia a dia, ou seja, nada mais são do que a transformação de uma forma de armazenamento já conhecida e utilizada no mundo real adaptada para o mundo computacional. Por isso, cada tipo de estrutura de dados possui vantagens e desvantagens e cada uma delas tem sua área de atuação (massa de dados) otimizada.

Os dados manipulados por um algoritmo podem possuir natureza distinta, isto é, podem ser números, letras, frases etc. Dependendo da natureza de um dado, algumas operações podem ou não fazer sentido quando aplicadas a eles. Por exemplo, não faz sentido falar em somar duas letras - algumas linguagens de programação permitem que ocorra a soma dos valores ASCII correspondentes de cada letra.

Para poder distinguir dados de naturezas distintas e saber quais operações podem ser realizadas com eles, os algoritmos lidam com o conceito de tipo de dados. O tipo de um dado define o conjunto de valores que uma variável pode assumir, bem como o conjunto de todas as operações que podem atuar sobre qualquer valor daquela variável. Por exemplo, uma variável do tipo inteiro pode assumir o conjunto de todos os números e de todas as operações que podem ser aplicadas a estes números.

Os tipos de dados manipulados por um algoritmo podem ser classificados em dois grupos: atômicos e complexos ou compostos. Os tipos atômicos são aqueles cujos elementos do conjunto de valores são indivisíveis, por exemplo: o tipo inteiro, real, caractere e lógico. Por outro lado, os tipos complexos são aqueles cujos elementos do conjunto de valores podem ser decompostos em partes mais simples. Se um tipo de dado pode ser decomposto, então o tipo de dado é dito estruturado, e a organização de cada componente e as relações entre eles constituem a disciplina de Estrutura de Dados.

1.1 Dados Homogêneos

Uma estrutura de dados, que utiliza somente um tipo de dado, em sua definição é conhecida como *dados homogêneos*. Variáveis compostas homogêneas correspondem a posições de memória, identificadas por um mesmo nome, individualizado por índices e cujo conteúdo é composto do mesmo tipo. Sendo os vetores (também conhecidos como estruturas de dados unidimensionais) e as matrizes (estruturas de dados bidimensionais) os representantes dos dados homogêneos.

1.1.1 Vetor

O vetor é uma estrutura de dados linear que necessita de somente um índice para que seus elementos sejam endereçados. É utilizado para armazenar uma lista de valores do mesmo tipo, ou seja, o tipo vetor permite armazenar mais de um valor em uma mesma variável. Um dado vetor é definido como tendo um

número fixo de células idênticas (seu conteúdo é dividido em posições). Cada célula armazena um e somente um dos valores de dados do vetor. Cada uma das células de um vetor possui seu próprio endereço, ou índice, através do qual pode ser referenciada. Nessa estrutura todos os elementos são do mesmo tipo, e cada um pode receber um valor diferente [3, 21, 4].

Algumas características do tipo vetor([10]):

- Alocação estática (deve-se conhecer as dimensões da estrutura no momento da declaração)
- Estrutura homogênea
- Alocação seqüencial (bytes contíguos)
- Inserção/Exclusão
  - Realocação dos elementos
  - Posição de memória não liberada

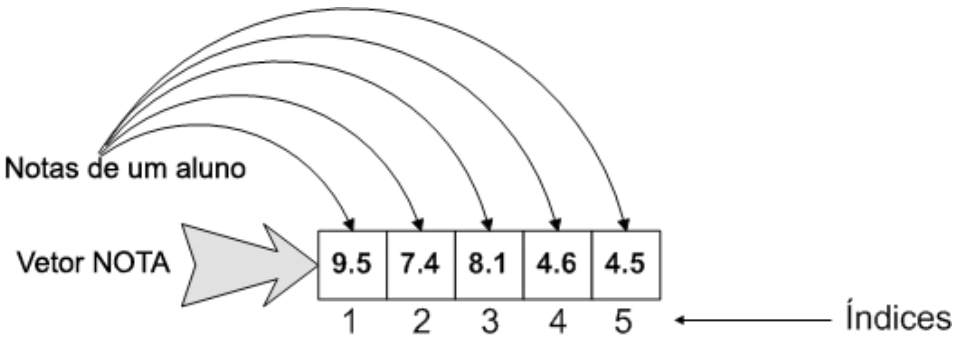

Figura 1.1: Exemplo de Vetor

A partir do endereço do primeiro elemento é possível determinar a localização dos demais elementos do vetor. Isso é possível porque os elementos do vetor estão dispostos na memória um ao lado do outro e cada elemento tem o seu tamanho fixo (algoritmo 1.1 [10]).

A partir do algoritmo 1.1 é possível deduzir a fórmula genérica para cálculo de posição na memória de um elemento qualquer. Sendo n o elemento, a fórmula se dá por Pos_n = *endereço Inicial + ((n - 1) * tamanho do tipo do elemento)*.

Algoritmo 1.1: Cálculo da posição de índices de um vetor na memória

```
1 begin
2 |   Endereço Elemento 1 ⟵ endereço Inicial
3 |   Endereço Elemento 2 ⟵ endereço Inicial + tamanho Elemento
4 |   Endereço Elemento 3 ⟵ endereço Inicial + (2 * tamanho Elemento)
5 |   Endereço Elemento 4 ⟵ endereço Inicial + (3 * tamanho Elemento)
6 end
```

A figura 1.1 mostra um vetor de notas de alunos, a referência **`NOTA[4]`** indica o valor **4.6** que se encontra na coluna indicada pelo índice **4**.

A definição de um vetor em C se dá pela sintaxe:

```
tipo_do_dado nome_do_vetor[ tamanho_do_vetor ]
```

O programa 1.1 contém um exemplo de declaração de um vetor na linguagem C. A figura 1.2 apresenta a representação deste vetor na memória do computador (lembrando que um vetor é guardado na memória de forma seqüencial).

Programa 1.1: Declaração de vetor em C

```
1  int i[3];
   i[0]=21;
   i[1]=22;
   i[2]=24;

6  char c[4];
   c[0]='a';
   c[1]='b';
   c[2]='c';
   c[3]='d';
```

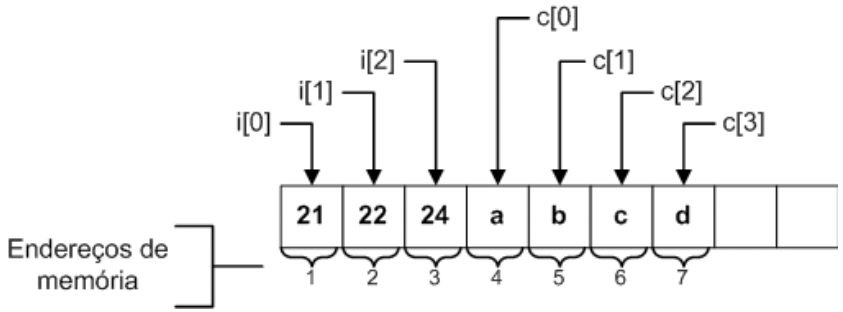

Figura 1.2: Representação de um vetor na memória

Programa 1.2: Exemplo de uso de vetores

```
/* programa_vetor_01.c */

#include <stdio.h>

#define TAMANHO 5

int main (void)
{
    int iIndice;
    int iValorA;
    int iSoma;
    int aVetor [TAMANHO];
    float fMedia;

    for (iIndice = 0; iIndice < TAMANHO; iIndice++)
    {
        printf("Entre com o valor %d:", iIndice + 1);
        scanf("%d", &iValorA);
        aVetor[iIndice] = iValorA;
    }

    iSoma = 0;
    for (iIndice=0; iIndice < TAMANHO; iIndice++)
    {
    iSoma += aVetor[iIndice];
    }
    fMedia = (float) iSoma/TAMANHO;
    printf ("Media : %f\n", fMedia);
    return 0;
    }
```

Lembrete: Caso seja colocada num programa a instrução `a[2]++` está sendo dito que a posição 2 do vetor `a` será incrementada.

1.1.2 Matriz

Uma matriz é um arranjo bidimensional ou multidimensional de alocação estática e seqüencial. A matriz é uma estrutura de dados que necessita de um índice para referenciar a linha e outro para referenciar a coluna para que seus elementos sejam endereçados. Da mesma forma que um vetor, uma matriz é definida com um tamanho fixo, todos os elementos são do mesmo tipo, cada célula contém somente um valor e os tamanhos dos valores são os mesmos (em C, um `char` ocupa 1 *byte* e um `int` `4` *bytes*) [3, 21, 4].

Os elementos ocupam posições contíguas na memória. A alocação dos elementos da matriz na memória pode ser feita colocando os elementos linha-por-linha ou coluna-por-coluna.

Para uma matriz de `2x2`— (figura 1.3) o algoritmo para calcular as posições da memória é listado em 1.2 [10].

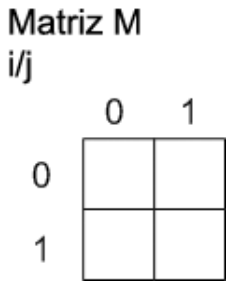

Figura 1.3: Matriz 2x2 - Cálculo de posição na memória

No algoritmo 1.2, sendo *C* a quantidade de colunas por linhas, *i* o número da linha e *j* a posição do elemento dentro linha, é possível definir a fórmula genérica para acesso na memória, onde Pos_{ij} = *endereço inicial + ((i-1) * C * tamanho do tipo do elemento) + ((j-1) * tamanho do tipo do elemento)*. A figura 1.4 demonstra a aplicação da fórmula.

A matriz `LETRAS` (figura 1.5) é composta de 18 elementos (3 linhas e 6 colunas), a referência a `MATRIZ[3][3]` (onde o primeiro 3 indica a linha e o segundo 3 indica a coluna) retorna o elemento 'N'; no caso de `MATRIZ[2][5]` (segunda linha e terceira coluna) irá retornar o elemento 'E'. Como é uma matriz de strings (linguagem C), a chamada a `MATRIZ[3]` irá reproduzir o valor "DONALD".

Algoritmo 1.2: Cálculo da posição de índices de uma matriz na memória

```
1 begin
2 |   M00 ⟵ endereço Inicial
3 |   M01 ⟵ endereço Inicial + 1 * tamanho Elemento
4 |   M10 ⟵ endereço Inicial + i * C * tamanho Elemento
5 |   M11 ⟵ endereço Inicial + i * C * tamanho Elemento + j * tamanho Elemento
6 end
```

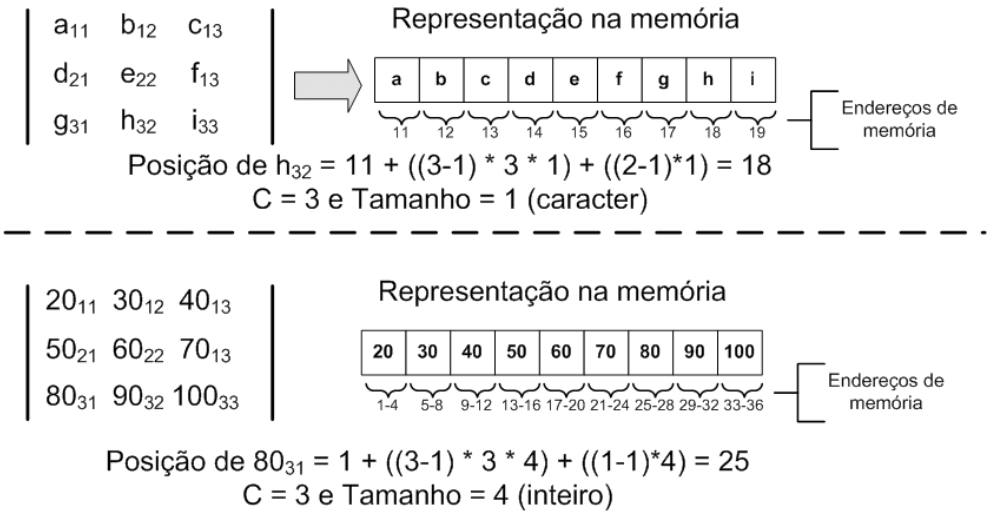

Figura 1.4: Cálculo de posição na memória

Uma matriz consiste de dois ou mais vetores definidos por um conjunto de elementos. Cada dimensão de uma matriz é um vetor (figura 1.6). O primeiro conjunto (dimensão) é considerado o primeiro vetor, o segundo conjunto o segundo vetor e assim sucessivamente [4].

A definição de uma matriz em C se dá pela sintaxe:

```
tipo_do_dado nome_da_matriz[ quantidade_linhas ] [ quantidade_colunas ]
```

O programa 1.3 apresenta um exemplo de declaração e utilização de matrizes bidimensionais na linguagem C.

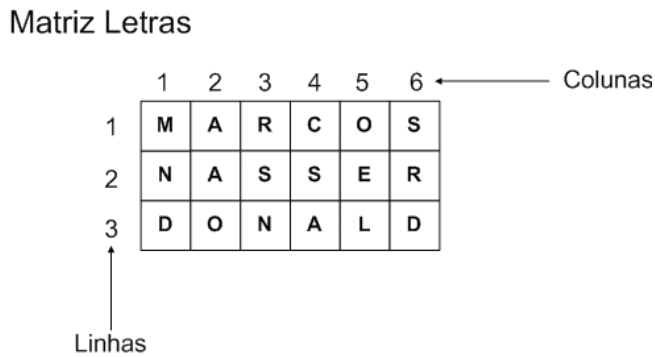

Figura 1.5: Exemplo de Matriz

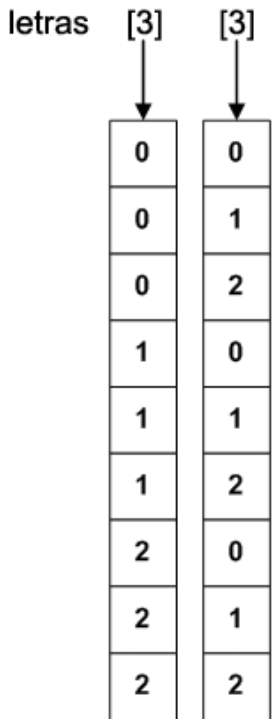

Figura 1.6: Uma matriz de duas dimensões vista como dois vetores

A linguagem C permite ainda trabalhar com matrizes de várias dimensões (matrizes n-dimensionais), embora o seu uso fique mais restrito em aplicações científicas face à sua pouca praticidade de uso.

A definição de uma matriz de várias dimensões em C se dá pela sintaxe:

```
    tipo_do_dado nome_da_matriz[tamanho_dimensão_1] [tamanho_
dimensão_2]
    [tamanho_dimensão_3] ... [tamanho_dimensão_n]
```

Programa 1.3: Exemplo de uso de matrizes

```
/* programa_matriz_01.c */

#include <stdio.h>

#define DIMENSAO 2

int main (void)
{
    int iLinha, iColuna;
    int iDeterminante;
    int iValorA;
    int aMatriz [DIMENSAO][DIMENSAO


    /* Uma regra que se pode sempre levar em consideração:
       para cada dimensão de uma matriz, sempre haverá um laço
       (normalmente um for). Se houver duas dimensões, então haverá dois laços. */
    for (iLinha=0; iLinha < DIMENSAO; iLinha++)
    {
        for (iColuna=0; iColuna < DIMENSAO; iColuna++)
        {
            printf ("Entre item %d %d:", iLinha + 1, iColuna + 1);
            scanf ("%d", &iValorA);
            matriz [iLinha][iColuna] = iValorA;
        }
    }
    iDeterminante = aMatriz[0][0] * aMatriz [1][1] -
    aMatriz[0][1] * aMatriz [1][0];
    printf ("Determinante : %d\n", iDeterminante);

    return 0;
}
```

O programa 1.4 é um exemplo de definição e utilização de matrizes multi-dimensionais na linguagem C.

Programa 1.4: Exemplo de uso de matrizes com várias dimensões

```
/* programa_matriz_02.c */

#include <stdio.h>
#define DIM_1 2
#define DIM_2 5
#define DIM_3 3
#define DIM_4 4

int main (void)
{
 int i,j,k,l;
 int aMatriz [DIM_1][DIM_2][DIM_3][DIM_4];

/* Código para zerar uma matriz de quatro dimensões */
for (i=0; i < DIM_1; i++)
{
    for (j=0; j < DIM_2; j++)
    {
        for (k=0; k < DIM_3; k++)
        {
            for (l=0; l < DIM_4; l++)
            {
                aMatriz [i][j][k][l] = i+j+k+l;
            }
        }
    }
 }

    /* Uma regra que se pode sempre levar em consideração: para cada
       dimensão de uma matriz, sempre haverá um laço (normalmente um for).
       Se houver quatro dimensoes então haverá quatro laços */
    For (i=0; i < DIM_1; i++)
    {
        for (j=0; j < DIM_2; j++)
        {
            for (k=0; k < DIM_3; k++)
            {
                for (l=0; l < DIM_4; l++)
                {
                    printf("\nValor para matriz em [%d] [%d] [%d] [%d] = %d",
                                i,j,k,l, aMatriz[i][j][k][l]);
                }
            }
        }
    }

    return 0;
}
```

1.1.3 Ponteiros

A linguagem C implementa o conceito de ponteiro. O ponteiro é um tipo de dado como int, char ou `float`. A diferença do ponteiro em relação aos outros tipos de dados é que uma variável que seja ponteiro guardará um endereço de memória [5, 7].

Por meio deste endereço pode-se acessar a informação, dizendo que a *variável ponteiro* aponta para uma posição de memória. O maior problema em relação ao ponteiro é entender quando se está trabalhando com o seu valor, ou seja, o endereço, e quando se está trabalhando com a informação apontada por ele.

Operador & e *

O primeiro operador de ponteiro é `&`. Ele é um operador unário que devolve o endereço na memória de seu operando. Por exemplo: `m = &count`; põe o endereço na memória da variável `count` em `m`. Esse endereço é a posição interna da variável na memória do computador e não tem nenhuma relação com o valor de `count`. O operador `&` tem como significado o *endereço de*. O segundo operador é *, que é o complemento de `&`. O * é um operador unário que devolve o valor da variável localizada no endereço que o segue. Por exemplo, se `m` contém o endereço da variável `count`: `q = *m`; coloca o valor de count em `q`. O operador * tem como significado *no endereço de*.

Lembrete: Cuide-se para não confundir o operador de ponteiros (*) como multiplicação na utilização de ponteiros e vice-versa.

A declaração de uma variável ponteiro é dada pela colocação de um asterisco (*) na frente de uma variável de qualquer tipo. Na linguagem C, é possível definir ponteiros para os tipos básicos ou estruturas. A definição de um ponteiro não reserva espaço de memória para o seu valor e sim para o seu conteúdo. Antes de utilizar um ponteiro, o mesmo deve ser inicializado, ou seja, deve ser colocado um endereço de memória válido para ser acessado posteriormente.

Um ponteiro pode ser utilizado de duas maneiras distintas. Uma maneira é trabalhar com o endereço armazenado no ponteiro e outro modo é trabalhar com a área de memória apontada pelo ponteiro. Quando se quiser trabalhar com o endereço armazenado no ponteiro, utiliza-se o seu nome sem o asterisco na frente. Sendo assim qualquer operação realizada será feita no *endereço do ponteiro*.

Como, na maioria dos casos, se deseja trabalhar com a memória apontada pelo ponteiro, alterando ou acessando este valor, deve-se colocar um asterisco

antes do nome do ponteiro. Sendo assim, qualquer operação realizada será feita no endereço de memória *apontado* pelo ponteiro. O programa 1.5 demostra a utilização de ponteiros para acesso à memória.

Programa 1.5: Exemplo de uso de ponteiros

```
1  /* programa_matriz_02.c */

   #include <stdio.h>

   int main (void)
   {
6     int *piValor; /* ponteiro para inteiro */

      int iVariavel = 27121975

      piValor = &iVariavel; /* pegando o endereço de memória da variável */
11

      printf ("Endereco: %d\n", piValor);
      printf ("Valor : %d\n", *piValor);

      *piValor = 18011982;
16    printf ("Valor alterado: %d\n", iVariavel);
      printf ("Endereco : %d\n", piValor);

      return 0;
   }
```

Passando variáveis para funções por referência

O ponteiro é utilizado para passar variáveis por referência, ou seja, variáveis que podem ter seu conteúdo alterado por funções e mantêm este valor após o término da função.

Na declaração de uma função, deve-se utilizar o asterisco antes do nome do parâmetro, indicando que está sendo mudado o valor naquele endereço passado como parâmetro. No programa 1.6 é visto um exemplo de uma variável sendo alterada por uma função (figura 1.7).

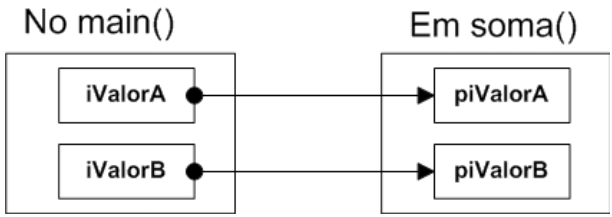

Figura 1.7: Chamada de função com ponteiros

Programa 1.6: Ponteiros como referência em funções

```
/* programa_ponteiro_02.c */

#include <stdio.h>

void soma (int, int, int *);

int main (void)
{
    int iValorA;
    int iValorB;
    int iResultado;

    printf ("Entre com os valores:");

    scanf ("%d %d", &iValorA, &iValorB);

    printf("Endereco de iResultado = %d\n", &iResultado);

    soma (iValorA, iValorB, &iResultado);/* está sendo passado o endereço de memória
                                            da variável, qualquer alteração estará
                                            sendo realizada na memória */
    printf ("Soma : %d\n", iResultado);

    return 0;
}

void soma (int piValorA, int piValorB, int * piResultado)
{
    printf("Endereco de piResultado = %d\n", piResultado);
    /* o valor está sendo colocado diretamente na memória */
    *piResultado = piValorA + piValorB;
    return;
}
```

Aritmética com ponteiros

Com uma variável do tipo ponteiro, é possível realizar operações de soma e subtração. Será somada ou diminuída no ponteiro a quantidade de endereços de memória relativos ao tipo do ponteiro. Por exemplo, um ponteiro para `int` ocupa 4 *bytes*, uma operação de soma neste ponteiro irá acrescentar 4 posições (unidades) na memória. O programa 1.7 é visto como os endereços de memória são manipulados através de aritmética de ponteiros, e a diferença entre o tipo `char` - que ocupa 1 *byte* de memória - e o tipo `int` - que ocupa 4 *bytes*.

Programa 1.7: Aritmética com ponteiros

```
/* programa_ponteiro_03.c */

#include <stdio.h>

int main (void)
{
    int   *piValor;
    int   iValor;
    char *pcValor;
    char cValor;

    piValor = &iValor;
    pcValor = &cValor;

    printf ("Endereco de piValor = %d\n", piValor);
    printf ("Endereco de pcValor = %d\n", pcValor);

    piValor++; /* somando uma unidade (4 bytes) na memória */
    pcValor++; /* somando uma unidade (1 byte) na memória */

    printf ("Endereco de piValor = %d\n", piValor);
    printf ("Endereco de pcValor = %d\n", pcValor);

    return 0;
}
```

Vetores e matrizes como ponteiros em C

Vetores nada mais são do que ponteiros com alocação estática de memória, logo, todo vetor na linguagem C é um ponteiro, tal que o acesso aos índices do vetor podem ser realizados através de aritmética de ponteiros. Observe a figura 1.8:

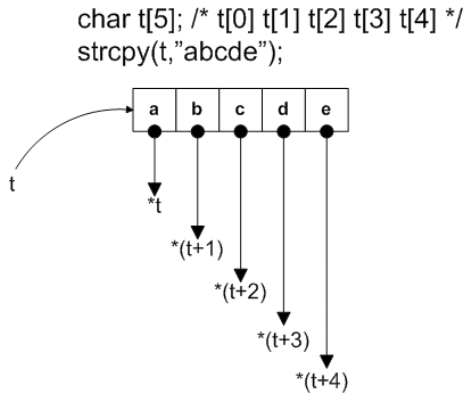

Figura 1.8: Vetor como Ponteiro em C

Existem 2 formas para se obter o endereço (ponteiro) do início do vetor ou matriz:

- `&t[0];`
- `t`

Sendo a segunda opção (t) a melhor, por ser mais simples. O endereço (ponteiro) para o primeiro elemento do vetor ou matriz é dado pelo nome do vetor sem colchetes.

O programa 1.8 mostra a utilização de um vetor com aritmética de ponteiros.

Programa 1.8: Vetor como ponteiro

```
/* programa_vetor_02.c */

#include <stdio.h>
#include <string.h>

int main(void)
{
    char t[5];

    strcpy(t,"abcde");

    printf("\n%ld %c", t, *t);
    printf("\n%ld %c", t+1, *(t+1));
    printf("\n%ld %c", t+2, *(t+2));
```

```
    printf("\n%ld %c", t+3, *(t+3));
    printf("\n%ld %c", t+4, *(t+4));
    return 0;
}
```

1.2 Dados Heterogêneos

Uma estrutura de dados é chamada de heterogênea quando envolve a utilização de mais de um tipo básico de dado (inteiro ou caractere, por exemplo) para representar uma estrutura de dados. Normalmente, este tipo de dado é chamado de registro.

Um registro é uma estrutura de dados que agrupa dados de tipos distintos ou, mais raramente, do mesmo tipo. Um registro de dados é composto por certo número de campos de dados, que são itens de dados individuais. Registros são conjuntos de dados logicamente relacionados, mas de tipos diferentes (numéricos, lógicos, caractere etc) [3, 21, 4].

O conceito de registro visa facilitar o agrupamento de variáveis que não são do mesmo tipo, mas que guardam estreita relação lógica. Registros correspondem a conjuntos de posições de memória conhecidos por um mesmo nome e individualizados por identificadores associados a cada conjunto de posições. O registro é um caso mais geral de variável composta na qual os elementos do conjunto não precisam ser, necessariamente, homogêneos ou do mesmo tipo. O registro é constituído por componentes. Cada tipo de dado armazenado em um registro é chamado de campo.

A figura 1.9 é um exemplo de registro de um funcionário: composto de nome, departamento, data de nascimento e salário.

Na variável composta homogênea, a individualização de um elemento é feita através de seus índices, já no registro cada componente é individualizado pela explicitação de seu identificador.

A linguagem C utiliza as estruturas para representar um registro. Com a estrutura definida pode-se fazer atribuição de variáveis do mesmo tipo de maneira simplificada. Veja o programa exemplo 1.9 (representação da figura 1.9):

Figura 1.9: Registro de funcionário

Programa 1.9: Exemplo de estrutura

```
struct Funcionario
{
    char nome [40];
    struct
    {
        int dia;
        int mes;
        int ano;
    } dataNasc;
    char departamento[10];
    float salario;
};
```

Para se fazer o acesso de um único campo deve-se utilizar o nome da estrutura seguido de um ponto e do nome do campo desejado da estrutura. A linguagem C também permite que seja criado um vetor de estruturas (programa 1.10).

Programa 1.10: Exemplo de uso de estruturas com vetores

```
/* programa_estrutura_01.c */
#include <stdio.h>


struct DADO
{
    char sNome[40];
    int iIdade;
};
```

```
int main(void)
{
    struct DADO sDados[5];

    /* A estrutura é dividida em duas partes por um ponto (.). Tem-se o nome da
       estrutura à esquerda e o nome do campo à direita. Neste exemplo,
       como está sendo manipulado um vetor de estruturas, também tem
       índice para cada linha do vetor. */
    for(iIndice=0;iIndice<5;iIndice++)
    {
        printf("\nEntre com o Nome ->" );
        scanf("%s", &sDados[iIndice].sNome );
        printf("Entre com a Idade ->" );
        scanf("%d", &sDados[iIndice].iIdade );
    }

    for(iIndice=0;iIndice<5;iIndice++)
    {
        printf("\n%s tem %d anos", sDados[iIndice].sNome, sDados[iIndice].iIdade);
    }
    return;
}
```

Lembrete: Estruturas são utilizadas para referenciar múltiplos tipos de dados.

1.3 Exercícios

1. Fazer um programa em C que implemente o algoritmo 1.1 para acessar elementos de vetores via ponteiros. Crie uma função:

   ```
   imprime_array_elemento(int *aArray, int iElemento);
   ```

2. Fazer um programa em C que implemente o algoritmo 1.2 para acessar elementos de uma matriz via ponteiros. Considerando uma matriz de 2x2:

   - Crie uma função:

     ```
     imprime_matriz_elemento_estatica(int paMatriz[][2],
                                      int piLinha,
                                     int piColuna);
     ```

   - Após entender o capítulo 2, crie uma função:

     ```
     imprime_matriz_elemento_dinamica(int ** paMatriz,
                                      int piLinha,
                                     int piColuna);
     ```

2. Uso de Memória

"Nada traz de volta os bons e velhos
tempos quanto uma memória fraca."
Franklin P. Adams

2.1 Alocação de memória Estática x Dinâmica

A alocação estática ocorre em tempo de compilação, ou seja, no momento em que se define uma variável ou estrutura é necessário que se definam seu tipo e tamanho. Nesse tipo de alocação, ao se colocar o programa em execução, a memória necessária para utilizar as variáveis e estruturas estáticas precisa ser reservada e deve ficar disponível até o término do programa (rotina ou função).

A alocação dinâmica ocorre em tempo de execução, ou seja, as variáveis e estruturas são declaradas sem a necessidade de se definir seu tamanho, pois nenhuma memória será reservada ao colocar o programa em execução. Durante a execução do programa, no momento em que uma variável ou parte de uma estrutura precise ser utilizada, sua memória será reservada e, no momento em que não for mais necessária, deve ser liberada. Isso é feito com o auxílio de comandos ou funções que permitem, por meio do programa, reservar e/ou liberar memória.

Pode-se dizer que os vetores e matrizes são estruturas estáticas e, por esse motivo, devemos definir seu número de posições. Algumas linguagens permitem criar vetores dinâmicos por meio do uso de ponteiros e sua memória é reservada durante a execução do programa.

Mesmo vetores dinâmicos devem ter o seu tamanho conhecido no momento da alocação da memória, pois um vetor deve ter toda sua memória alocada

antes da sua utilização. Estruturas encadeadas dinâmicas possuem tamanho variável, pois diferentemente de vetores dinâmicos, sua memória é reservada por elemento e não para toda a estrutura.

2.2 Alocação dinâmica de memória

Várias linguagens de programação possibilitam manipular dinamicamente a memória das suas estruturas de dados. Algumas linguagens como o Java possibilitam que uma estrutura de dados cresça ou diminua quase que sem interferência do programador. Outras linguagens como o C exigem que o trabalho de alocação de memória seja feito antecipadamente pelo programador.

Na linguagem C, uma matriz na prática é um vetor de ponteiros, onde cada coluna é um ponteiro para uma linha. Na figura 2.1 pode ser visualizada a representação de uma matriz como um vetor com ponteiros.

2.3 Funções para alocação de memória

Na linguagem C, a alocação dinâmica de memória pode ser realizada com apenas quatro chamadas a funções:

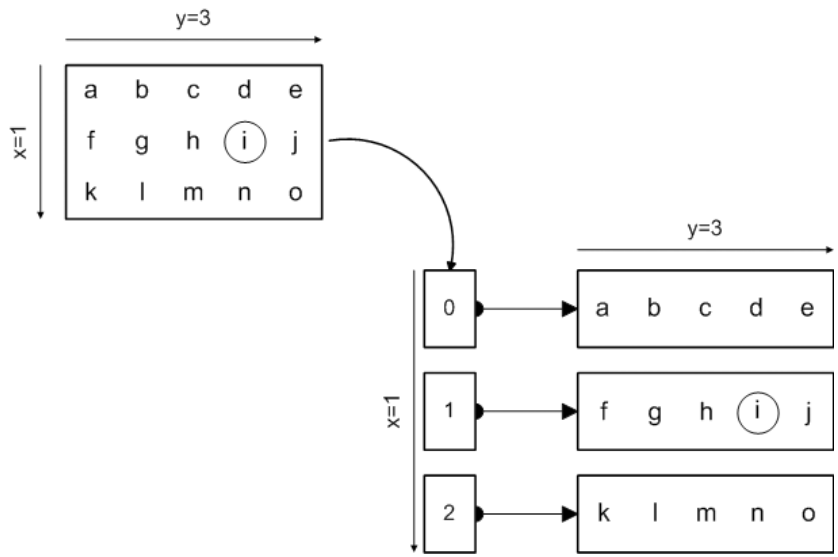

Figura 2.1: Matriz como vetor de ponteiros

- `void * malloc(int qty_bytes_alloc);`
- `void * calloc(int qty, int size);`
- `void * realloc(void * pointer, int new_size);`
- `free( void * pointer);`

A função **malloc** permite que seja feita a alocação de uma nova área de memória para uma estrutura. A função **calloc** tem a mesma funcionalidade de **malloc**, exceto que devem ser fornecidos o tamanho da área e a quantidade de elementos. A função **realloc** permite que uma área previamente alocada seja aumentada ou diminuída e a função **free** libera uma área alocada previamente com a função **malloc**, **calloc** ou **realloc**.

2.3.1 Função malloc

É a função **malloc** que realiza a alocação de memória. Deve-se informar para a função a quantidade de bytes para alocação. A função irá retornar, se existir memória suficiente, um endereço que deve ser colocado em uma variável do tipo ponteiro.

Como a função retorna um ponteiro para o tipo **void**, deve-se utilizar o *typecast*, transformando este endereço para o tipo de ponteiro desejado.

2.3.2 Função calloc

Em vez de se alocar uma quantidade de *bytes* através da função **malloc**, pode-se usar a função **calloc** e especificar a quantidade de bloco de um determinado tamanho. Funcionalmente a alocação irá ocorrer de maneira idêntica.

A única diferença entre o **malloc** e o **calloc** é que a última função, além de alocar o espaço, também inicializa o mesmo com zeros.

2.3.3 Função realloc

Às vezes é necessário expandir uma área alocada. Para isto deve-se usar a função **realloc**. Deve-se passar para ela o ponteiro retornado pelo **malloc** e a indicação do novo tamanho. A realocação de memória pode resultar na troca de blocos na memória.

2.3.4 Função free

Quando não se deseja mais uma área alocada, deve-se liberá-la através da função **free**. Deve ser passado para a função o endereço, que se deseja liberar, que foi devolvido quando a alocação da memória ocorreu.

2.4 Utilizando as funções para alocação de memória

Um vetor nada mais é do que um ponteiro com alocação estática de memória. A declaração `int aVetor[10]`; é equivalente:

Programa 2.1: Declaração de vetor como ponteiro

```
1   int *aVetor;
     aVetor = (int *) malloc(10 * sizeof(int *));
```

Quando se quer criar estruturas com dois índices (matrizes), três índices (tijolos) etc. A declaração da matriz seria algo como `int aMatriz[2][3];`, utilizando ponteiros para declarar uma matriz deve-se usar:

Programa 2.2: Declaração de matriz como ponteiro

```
    int **aMatriz;
    aMatriz = (int **) malloc( 2 * sizeof(int *)); /* 2 linhas */
3   for( i=0; i<2; i++ )
    {
    aMatriz[i] = (int *) malloc( 3 * sizeof(int)); /* 3 colunas */
    }
```

A notação `aMatriz[i][j]` pode ser utilizada com matrizes alocadas dinamicamente equivalente a `*(aMatriz[i]+j)` ou `*(*(aMatriz+i)+j)`. Ou seja, pega-se o endereço de `aMatriz` e encontra-se o endereço da i-ésima linha (`i*sizeof(int *)`) posições à frente (`aMatriz[i]` é equivalente a `*(aMatriz+i)`). Este endereço pode ser interpretado como um vetor, ao qual somam-se `j*sizeof(int)` posições para encontrar o elemento `aMatriz[i][j]`.

O programa 2.3 utiliza as funções **malloc** e **realloc** para criar e aumentar o tamanho de um vetor dinamicamente (em tempo de execução). No caso de algum erro, as funções retornam um ponteiro nulo (`NULL`) para indicar erro de alocação de memória.

Programa 2.3: Exemplo de uso do malloc e realloc

```
/* programa_memoria_01.c */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *p;
    int i,k, n;
    printf("\nDigite a quantidade de numeros que serao digitados ->");
    scanf("%d", &i);

    /* A função malloc reserva espaço suficiente para um vetor de inteiros.
       Caso sejam digitados 5 elementos, serão reservados 20 bytes, pois cada
       inteiro ocupa 4 bytes na memória */
    p = (int *)(malloc(i*sizeof(int)));
    if( p == NULL )
    {
        printf("\nErro de alocacao de memoria");
        exit(1);
    }

    for( k=0;k<i;k++)
    {
        printf("Digite o numero para o indice %d ->", k);
        scanf("%d", &p[k]);
    }

    for( k=0;k<i;k++)
    {
        printf("O numero do indice %d eh %d\n", k, p[k]);
    }

    printf("\nDigite quantos elementos quer adicionar ao vetor ->");
    scanf("%d", &n

    /* A função realloc aumenta ou diminui o tamanho do vetor dinamicamente. Ela recebe o
       ponteiro para o vetor anterior e retorna o novo espaço alocado. */
    p = (int *)(realloc(p,(i+n)*sizeof(int)));
    if( p == NULL )
    {
        printf("\nErro de re-alocacao de memoria");
        exit(1);
    }
    for(;k<(n+i);k++)
    }
```

```
    {
        printf("Digite o numero para o indice %d ->", k);
        scanf("%d", &p[k]);
    }
    for( k=0;k<(i+n);k++)
    {
        printf("O numero do indice %d eh %d\n", k, p[k]);
    }
    free(p);

    return 0;
}
```

O programa 2.4 utiliza a função **calloc** para criar uma matriz em tempo de execução. De forma idêntica a **malloc**, a função **calloc** retorna um ponteiro nulo (`NULL`) no caso de erro de alocação de memória.

Programa 2.4: Exemplo de uso do calloc

```
/* programa_memoria_02.c */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    /* A declaração de uma matriz de 2 dimensões exige um ponteiro para ponteiro. */
    int **p;
    int i,j,k,x;
    printf("\nDigite as dimensoes da matriz ->");
    scanf("%d %d", &i, &j);

    /* Alocação da primeira dimensão. Internamente, a função calloc fará uma multiplicação da
       quantidade de elementos pelo tamanho de cada elemento para saber quanto de memória
       deve ser alocada. */
    p = calloc(i,sizeof(int));
    if( p == NULL )
    {
        printf("\nErro de alocacao de memoria");
        exit(1);
    }
    for( k=0;k<i;k++)
    {
        /* Alocação das linhas de cada coluna (segunda dimensão) */
        p[k] = calloc(j,sizeof(int));
        if( p[k] == NULL )
        {
```

```
            printf("\nErro de alocacao de memoria");
            exit(1);
        }
    }

    for( k=0;k<i;k++)
    {
        for(x=0;x<j;x++)
        {
            printf("Digite o numero para o indice %d,%d ->", k,x);
            scanf("%d", &p[k][x]);
        }
    }

    for( k=0;k<i;k++)
    {
        for(x=0;x<j;x++)
        {
            printf("O numero do indice %d,%d eh %d\n", k,x, p[k][x]);
        }
    }

    printf("\nLiberando memoria alocada");
    for( k=0;k<i;k++)
    {
        free(p[k]); /* Primeiro deve ser liberada a memória para linha da matriz... */
    }
    free(p); /* ... para depois liberar a memória do vetor que continha as linhas. */
}
```

2.5 Alocação de memória e estruturas em C

A estrutura de dados **struct** também pode ser alocada dinamicamente com as funções malloc ou calloc. O programa 2.5 trabalha com uma estrutura de dados **struct** alocada dinamicamente.

Programa 2.5: Exemplo de uso de estruturas com ponteiros

```
/* programa_memoria_03.c */
#include <stdio.h>
#include <stdlib.h>

struct ST_DADOS
{
    char nome[40];
```

```
    float salario;

    /* estrutura dentro de uma estrutura */
    struct nascimento
    {
        int ano;
        int mes;
        int dia;
    } dt_nascimento;
};

int main(void)
{

    /* ponteiro para a estrutura */
    struct ST_DADOS * p;

    /* alocação de memória para o ponteiro da estrutura */
    p = (struct ST_DADOS *) malloc(sizeof(struct ST_DADOS));

    /* string (vetor de caracteres) é um ponteiro, por isto a ausência do & */
    printf("\nEntre com o nome ->");
    scanf("%s", p->nome);

    printf("Entre com o salario ->");
    scanf("%f", &p->salario);

    /* O -> é chamado de pointer member (apontador de membro). Ele é usado para
       referenciar um campo da estrutura no lugar do ponto (.) */
    printf("Entre com o nascimento ->");
    scanf("%d%d%d", &p->dt_nascimento.dia,
                    &p->dt_nascimento.mes,
                    &p->dt_nascimento.ano);

    printf("\n===== Dados digitados ====");
    printf("\nNome = %s", p->nome);
    printf("\nSalario = %f", p->salario);
    printf("\nNascimento = %d/%d/%d\n", p->dt_nascimento.dia,
                                        p->dt_nascimento.mes,
                                        p->dt_nascimento.ano);
    free(p);
}
```

2.6 Ponteiros para ponteiros – mistério ou não

O uso de ponteiros confunde os iniciantes nas disciplinas de programação. Quando vêem ponteiros para ponteiros, até profissionais mais experientes têm dificuldades [7].

A referência para uma variável `int iVariavel` é obtida atráves do uso do `&` no caso `funcao(&iVariavel)`, ou seja, é passado o endereço da memória da variável em uso. A variável é recebida como um ponteiro na função (`void funcao(int * piVariavel)`).

Quando se quer alocar um vetor ou uma matriz dentro de uma função, deve-se passar a referência do ponteiro (por causa da alocação dinâmica). Ou seja, uma variável declarada como ponteiro (`int * piVariavel`) vai ter sua referência passada para uma função (`funcao(&piVariavel)`) e recebida como um ponteiro na função em questão. Como regra pode ser adotada a adição de um * sempre no início de uma variável, logo, a função seria declarada como `void funcao(int **piVariavel)`.

Considere a declaração de um ponteiro para inteiro (`int *piVariavel`), esta declaração gera uma alocação estática de memória para o ponteiro (vamos considerar o endereço 1 na memória (a)). Os segmentos 2 e 3 da memória estão sendo utilizados por outras variáveis de tal forma que o comando `*piVariavel = (int *) malloc(sizeof(int))` retorna o endereço 4 da memória (b), isto significa que o valor 4 vai ser armazenado (c) dentro do ponteiro `*piVariavel` (endereço 1). A instrução `*piVariavel = 20` (d) irá colocar o valor 20 no endereço apontado por `*piVariavel` (o endereço apontado é 4). A figura 2.2 exemplifica a operação.

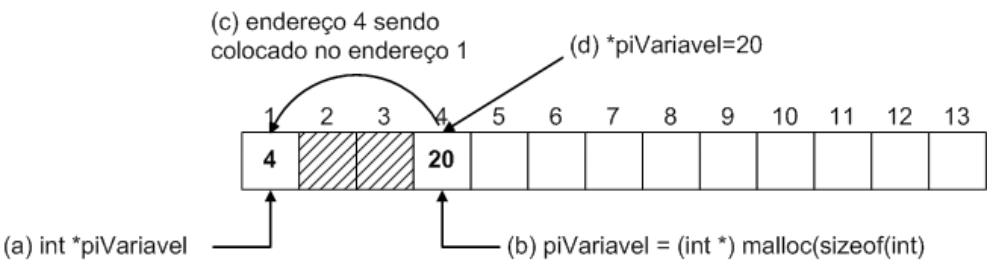

Figura 2.2: Exemplo de ponteiro na memória

A tabela 2.1 demonstra as operações com ponteiro (pegando o endereço de memória do ponteiro, o endereço armazenado no ponteiro e o conteúdo colocado no endereço indicado pelo ponteiro).

Tabela 2.1: Operações com ponteiros

Operação	Resultado
`printf("%d\n",&piVariavel)`	1
`printf("%d\n",piVariavel)`	4
`printf("%d\n",*piVariavel)`	20

A passagem de um ponteiro por referência a uma função gera um ponteiro para ponteiro. Considere a declaração de uma função `int funcao(int **piParametro)` (programa 2.6), esta declaração gera uma declaração estática de memória para o ponteiro `**piParametro` (vamos considerar o endereço de memória 6 (a)). A chamada a função `funcao(&piVariavel))` será interpretada como passando *endereço de* `piVariavel` para o ponteiro `piParametro` (b). Desta forma é criado um ponteiro para o ponteiro (c). A figura 2.3 (continuação da figura 2.2) exemplifica a operação.

Programa 2.6: Ponteiro para ponteiro

```
#include <stdio.h>
#include <stdlib.h>
int funcao(int **piParametro)
{
    printf("%d\n",&piParametro);
    printf("%d\n",piParametro);
    printf("%d\n",*piParametro);
    printf("%d\n",**piParametro);
    return 0;
}

int main( void )
{
    int *piVariavel;
    *piVariavel = (int *) malloc(sizeof(int);
    *piVariavel = 20;
    printf("%d\n",&piVariavel);
    printf("%d\n",piVariavel);
    printf("%d\n",*piVariavel);


    funcao( &piVariavel );
```

```
    return 0;
}
```

A tabela 2.2 demonstra as operações com ponteiro (pegando o endereço de memória do ponteiro, o endereço armazenado no ponteiro e o conteúdo colocado no endereço indicado pelo ponteiro).

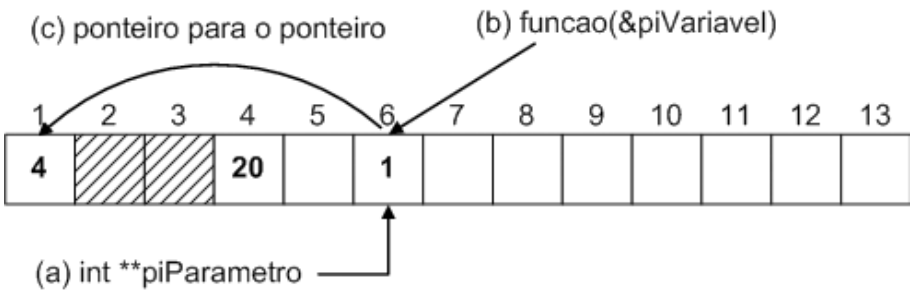

Figura 2.3: Exemplo de ponteiro para ponteiro na memória

Tabela 2.2: Operações com ponteiros de ponteiros

Operação	Resultado	Significado
`printf("%d\n",&piParametro)`	6	Endereço de piParametro
`printf("%d\n",piParametro)`	1	Conteúdo de piParametro
`printf("%d\n",*piParametro)`	4	Conteúdo do endereço apontado por piParametro (piVariavel)
`printf("%d\n",**piParametro)`	20	Valor do endereço apontado por piParametro (*piVariavel)

2.7 Mais alocação de vetores e matrizes como ponteiros

No programa 2.7 são utilizadas várias funções para a alocação de memória e manipulação de uma matriz; nestes casos, uma matriz deve ser declarada utilizando o formato de ponteiro. Ao alocar dinamicamente uma matriz utilizando o formato de ponteiros, o ponteiro que representa uma matriz pode ser utilizado no formato `matriz[linha][coluna]` para referenciar uma posição da ma-

triz (mesmo formato de utilização caso a matriz tivesse sido declarada estaticamente no programa).

Programa 2.7: Exemplo de uso de alocação de matrizes

```
1  1/* programa_memoria_04.c */
   #include <stdio.h>
   #include <stdlib.h>

   int ** aloca(int i, int j);
6  void libera(int **p, int i, int j);
   void leitura(int **p, int i, int j);
   void imprime(int **p, int i, int j);

   int main(void)
11 {
       int **p;
       int **p1;

       p = aloca(3,2);
16     leitura(p, 3, 2);

       p1 = aloca(2,3);
       leitura(p1,2,3);

21     imprime(p,3,2);
       imprime(p1,2,3);

       libera(p,3,2);
       libera(p1,2,3);
26
       return 0;
   }

   /* 2 asteriscos (*) indicam que será retornada uma matriz */
31 int ** aloca(int i, int j)
   {
       /* ponteiro de ponteiro. Indica que será alocada uma matriz de 2 dimensões */
       int **p;
       int x;
36     p = calloc(i,sizeof(int)); /* alocação de linhas... */
       if( p == NULL )
       {
           printf("\nErro de alocacao");
           exit(-1);
```

```
41      }
        for(x=0;x<i;x++)
        {
            p[x]=calloc(j,sizeof(int)); /* ... e alocação de colunas */
            if( p[x] == NULL )
46          {
                printf("\nErro de alocacao");
                exit(-1);
            }
        }
51      return p;
}

/* 2 asteriscos (*) indicam que a função recebe uma matriz */
void leitura(int **p, int i, int j)
{
    int x,y;
    for(x=0;x<i;x++)
    {
        for(y=0;y<j;y++)
        {
61          printf("Entre com o elemento %d,%d ->", x, y);
            scanf("%d", &p[x][y]); /* uso da matriz no formato tradicional */
        }
    }
66  }


/* 2 asteriscos (*) indicam que a função recebe uma matriz */
71  void imprime(int **p, int i, int j)
{
    int x,y;
    for(x=0;x<i;x++)
    {
        for(y=0;y<j;y++)
        {
76          printf("\nElemento %d,%d = %d", x, y, p[x][y]);
        }
    }
}
81

/* 2 asteriscos (*) indicam que a função recebe uma matriz */
void libera(int **p, int i, int j)
{
    int x;
86  for(x=0;x<i;x++)
    {
        free(p[x]); /* libera coluna a coluna */
```

```
    }
    free(p); /* libera as linhas */
}
```

Finalmente, o programa 2.8 realiza a alocação de matrizes de uma forma diferente. No programa 2.7 a função **aloca** retorna o ponteiro para uma matriz alocada, no programa 2.8 a função **aloca** recebe um ponteiro já definido e deve alocar a memória solicitada pelo programador. Sempre que uma função precisa alocar memória para um ponteiro recebido, esta função deve receber o ponteiro do ponteiro. A notação de uso de ponteiro para ponteiro normalmente confunde os profissionais menos experientes. Pode-se tomar como regra a utilização do ponteiro precedido de um * e entre parênteses para cada dimensão que se deseja manipular. A partir da segunda dimensão é possível utilizar a notação de coluna para manipulação da matriz.

Programa 2.8: Exemplo de uso de alocação de matrizes dentro de funções

```
/* programa_memoria_05.c */
#include <stdio.h>
#include <stdlib.h>

void aloca(int ***p, int x, int y);

/* a função recebe um ponteiro para uma matriz */
void aloca(int ***p, int x, int y)
{
    int i;
    *p = (int **)malloc(sizeof(int) * x);
    if( *p == NULL )
    {
        printf("\nErro de alocacao de memoria!");
        exit(1);
    }

    for( i = 0; i<y; i++)
    {
        (*p)[i] = (int *) malloc(sizeof(int) * y);
        if( (*p)[i] == NULL )
        {
            printf("\nErro de alocacao de memoria!");
            exit(1);
        }
    }
    return;
    }
```

```
int main(void)
{
    int **p; /* declaração de uma matriz com duas dimensões */
33  int i,k;

    aloca(&p,4,5); /* passando para a função o endereço de memória do ponteiro */

    for( i=0;i<4;i++)
38  {
        for( k=0;k<5;k++)
        {
            p[i][k] = i + k;
        }
43  }

    for( i=0;i<4;i++)
    {
        for( k=0;k<5;k++)
48      {
            printf("%d ", p[i][k]); /* referência aos elementos através de linha e coluna */
        }
        printf("\n");
        }
53  return 0;
    }
```

2.7.1 Controle de agenda com ponteiros de estruturas e vetores

O programa 2.9 contém várias formas de manipulação e alocação de ponteiros com vetores. O objetivo do programa é criar um cadastro simples de agenda (em memória) com inclusão de novas entradas e alteração, consulta, exclusão e pesquisa (ordenada) de entradas já existentes.

Programa 2.9: Exemplo completo de uso de vetor (ponteiros) de estruturas

```
 /* agenda.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
5

/* função getch */
#ifdef DOS
#include <conio.h>
#else
```

```
#include <curses.h>
#endif

typedef struct agenda
{
    char nome[40];
    char email[40];
    int telefone;
} AGENDA;

void aloca( AGENDA **pAgenda, int *piEntradas );
void consulta( AGENDA *pAgenda, int iEntradas);

void qs_ordena(AGENDA pAgenda[], int left, int right );
void ordena( AGENDA pAgenda[], int iEntradas );

void excluir(AGENDA **pAgenda, int *piEntradas);
void pesquisar(AGENDA *pAgenda, int iEntradas);
void alterar(AGENDA *pAgenda, int iEntradas);

int main(void)
{
    AGENDA * pAgenda;

    int iEntradas, op;
    iEntradas=0;

    pAgenda = (AGENDA *) malloc(sizeof(AGENDA)); /* alocando espaço para a posição 0
                                                    do vetor */
    if( pAgenda == NULL )
    {
       printf("\nErro de alocacao de memoria.");
       exit(1);
    }

    do
    {
       fflush(stdin);
       printf("\n1 - Inclusao");
       printf("\n2 - Alteracao");
       printf("\n3 - Consulta");
       printf("\n4 - Excluir");
       printf("\n5 - Pesquisar");
       printf("\n9 - Sair");
       printf("\nEntre com uma opcao -> ");
       scanf("%d", &op);
```

```
        if( op == 1 )
        {
        /* farei aqui para ilustrar algumas formas de manipular ponteiros */
        fflush(stdin);

        /* alocação de ponteiros em funções requer trabalhar com ponteiros
              para ponteiros */
        aloca(&pAgenda, &iEntradas);

        printf("*** Inclusao ***");
        printf("\nEntre com o Nome:");
        /* forma 1 - endereço ponteiro inicial + x posições na memória
                    quando se trabalhar com o endereço, deve-se usar -> */
        gets((pAgenda+iEntradas)->nome);
        fflush(stdin);

        printf("Entre com o email:");
        /* forma 2 - endereço ponteiro inicial + x posições na memória
                    quando se trabalhar com ponteiro (conteúdo do endereço ou *),
                    deve-se usar o . (ponto) */
        gets((*(pAgenda+iEntradas)).email);
        fflush(stdin);

        printf("Entre com o telefone:");
        /* forma 3 - trabalhando como vetor */
        scanf("%d", &pAgenda[iEntradas].telefone);
        fflush(stdin);

        iEntradas++;
    }
    else if( op == 2)
    {
        alterar(pAgenda, iEntradas);
    }
    else if( op == 3 )
    {
        /* se o vetor de estruturas vai ser somente lido
           não é preciso passar ponteiro para ponteiro */
        ordena(pAgenda, iEntradas);
        consulta(pAgenda, iEntradas);
    }
    else if( op == 4)
    {
    ordena(pAgenda, iEntradas);
    excluir(&pAgenda, &iEntradas);
    }
    else if( op == 5
```

```
        {
            ordena(pAgenda, iEntradas);
            pesquisar(pAgenda,iEntradas);
        }
    } while(op!=9);
}

void consulta(AGENDA *pAgenda, int iEntradas)
{
    int i;
    for(i=0;i<iEntradas;i++)
    {
        printf("\n\nRegistro %d", i);
        printf("\n\tNome: %s", pAgenda[i].nome );
        printf("\n\tEmails: %s", pAgenda[i].email );
        printf("\n\tTelefone: %d", pAgenda[i].telefone );
        getch();
    }
}

void alterar(AGENDA *pAgenda, int iEntradas)
{
    char op;
    int i=0;
    char nome[40];
    printf("\n\tDigite o Nome:");
    fflush(stdin);
    gets(nome);

    for(i=0; i < iEntradas && strncmp( pAgenda[i].nome, nome, strlen(nome))!=0;i++);
    if( i>= iEntradas )
    {
        printf("\nRegistro nao encontrado");
    }
    else
    {
        printf("\n\tRegistro %d", i);
        printf("\n\tNome : %s", pAgenda[i].nome );
        printf("\n\tEmail : %s", pAgenda[i].email );
        printf("\n\tFone : %d", pAgenda[i].telefone );
        printf("\n\tConfirma a alteracao ?");
        op = getch();
        if( op == 'S' || op == 's' )
        {
            fflush(stdin);
            printf("\nEntre com o Nome:");
            /* forma 1 - endereço ponteiro inicial + x posições na memória
                        quando se trabalhar com o endereço, deve-se usar -> */
```

```
            gets((pAgenda+i)->nome);
            fflush(stdin);

            printf("Entre com o email:");
            /* forma 2 - endereço ponteiro inicial + x posições na memória
                        quando se trabalhar com ponteiro (conteúdo do endereço ou *),
                       deve-se usar o . (ponto) */
            gets((*(pAgenda+i)).email);
            fflush(stdin);

            printf("Entre com o telefone:");
            /* forma 3 - trabalhando como vetor */
            scanf("%d", &pAgenda[i].telefone);
            fflush(stdin);
        }
    }
}

void excluir(AGENDA **pAgenda, int *piEntradas)
{
    char op;
    int i=0;
    char nome[40];
    printf("\n\tDigite o Nome:");
    fflush(stdin);
    gets(nome);
    /* Uso a sintaxe (*pAgenda)[i].nome pelo fato de ser ponteiro de ponteiro.
       Os parênteses neste caso servem para "fixar" a primeira posição da memória, pois
       a linguagem C tende a trabalhar com ponteiros de ponteiros como se fossem
       matrizes (que na prática são ponteiros para ponteiros) */
    for(i=0; i < *piEntradas && strncmp( (*pAgenda)[i].nome, nome, strlen(nome))!=0;i++);
    if( i>= *piEntradas )
    {
        printf("\nRegistro não encontrado");
    }
    else
    {
        fflush(stdin);
        printf("\n\tRegistro %d", i);
        printf("\n\tNome : %s", (*pAgenda)[i].nome );
        printf("\n\tEmail : %s", (*pAgenda)[i].email );
        printf("\n\tFone : %d", (*pAgenda)[i].telefone );
        printf("\n\tConfirma a exclusao ?");
        op = getch();
        if( op == 'S' || op == 's' )
        {
            /* copio o último elemento para o elemento corrente */
            (*pAgenda)[i] = (*pAgenda)[(*piEntradas)-1];
```

```
            (*piEntradas)--;
            /* excluo o último elemento com realoc */
            aloca(pAgenda, piEntradas);
        }
    }
}

void aloca( AGENDA **pAgenda, int *piEntradas )
{
    (*pAgenda) = (AGENDA *)(realloc(*pAgenda,
                (*piEntradas+1)*sizeof(AGENDA)));
    if( *pAgenda == NULL )
    {
        printf("\nErro de re-alocacao de memoria");
        exit(1);
    }
}

void pesquisar(AGENDA *pAgenda, int iEntradas)
{
    char op;
    int i=0;
    char nome[40];
    printf("\n\tDigite o Nome:");
    fflush(stdin);
    gets(nome);

    for(i=0; i < iEntradas && strncmp( pAgenda[i].nome, nome, strlen(nome))!=0;i++);
    if( i>= iEntradas )
    {
        printf("\nRegistro nao encontrado");
    }
    else
    {
        do
        {
            fflush(stdin);
            printf("\n\n\tRegistro %d", i);
            printf("\n\tNome : %s", pAgenda[i].nome );
            printf("\n\tEmail : %s", pAgenda[i].email );
            printf("\n\tFone : %d", pAgenda[i].telefone );
            printf("\n\tProximo ?" );
            op = getch();
            i++;
            if(i>=iEntradas)
            {
                i = 0;
            }
```

```
        } while( op == 'S' || op == 's');
    }
}

void ordena( AGENDA pAgenda[], int iEntradas )
{
    qs_ordena(pAgenda, 0, iEntradas-1 );
}

void qs_ordena(AGENDA pAgenda[], int left, int right )
{
    register int i, j;
    char * x;
    AGENDA t;

    i = left;
    j = right;
    x = pAgenda[(left+right)/2].nome;

    do
    {
        while(strcmp(pAgenda[i].nome,x)<0 && i<right) i++;
        while(strcmp(pAgenda[j].nome,x)>0 && j>left) j --;
        if( i<=j )
        {
            t = pAgenda[i];
            pAgenda[i]=pAgenda[j];
            pAgenda[j]=t;
            i++;
            j--;
        }
    } while( i<=j );
    if( left < j ) qs_ordena(pAgenda, left, i);
    if( i<right) qs_ordena(pAgenda, i, right );
}
```

3. Pilha

"A tecnologia é dominada por aqueles que
gerenciam o que não entendem."
Arthur Bloch

Uma pilha é um conjunto ordenado de itens, no qual novos itens podem ser inseridos e a partir do qual podem ser eliminados itens de uma extremidade, chamada topo da pilha. Também é chamada de lista linear, onde todas as inserções e eliminações são feitas em apenas uma das extremidades, chamada topo. A figura 3.1 mostra a representação de uma pilha.

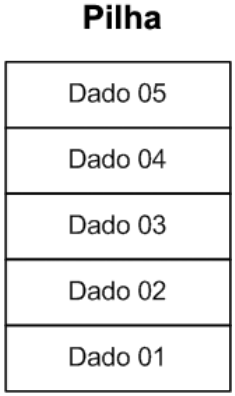

Figura 3.1: Exemplo de Pilha

A estrutura de dados do tipo pilha tem como característica que a última informação a entrar é a primeira a sair (LIFO - *last in first out*). A estrutura em pilha tem os seguintes métodos ou funções:

- *push* - coloca uma informação na pilha (empilha).
- *pop* - retira uma informação da pilha (desempilha).
- *size* - retorna o tamanho da pilha.
- *stackpop* - retorna o elemento superior da pilha sem removê-lo (equivalente às operações de pop e um push).
- *empty* - verifica se a pilha está vazia.

A aplicação da estrutura de pilhas é mais freqüente em compiladores e sistemas operacionais, que a utilizam para controle de dados, alocação de variáveis na memória etc.

O problema no uso de pilhas é controlar o final da pilha. Isto pode ser feito de várias formas, sendo a mais indicada criar um método para verificar se existem mais dados na pilha para serem retirados.

Tomando a pilha da figura 3.2 como exemplo, a operação **push(H)** irá acrescentar um novo elemento ao topo da pilha sendo, em seguida, executado um conjunto de operações sobre a pilha:

- 2 - push(I) - Coloca o elemento I no topo da Pilha
- 3 - pop() - Retorna o elemento I
- 4 - pop() - Retorna o elemento H
- 5 - pop() - Retorna o elemento F
- 6 - pop() - Retorna o elemento E
- 7 - pop() - Retorna o elemento D
- 8 - push(D) - Coloca o elemento D no topo da Pilha
- 9 - push(E) - Coloca o elemento E no topo da Pilha

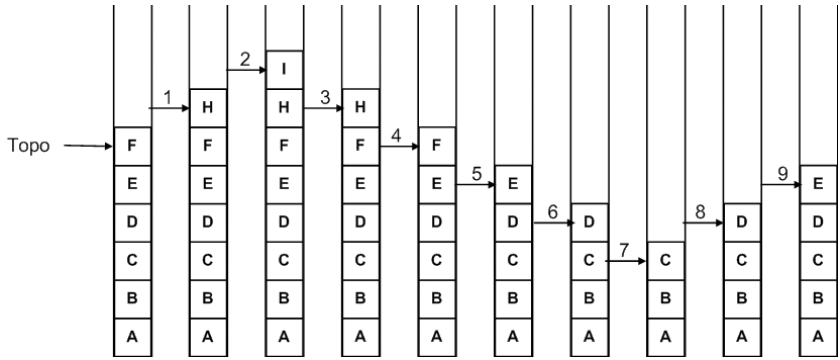

Figura 3.2: Operações em uma pilha

3.1 Representação das Operações com Pseudo-código

As operações de pilha podem ser representadas com algumas linhas de pseudo-código. Os algoritmos **empty** (3.1), **push** (3.2), **pop** (3.3), **stackpop** (3.4) e **size** (3.5) demonstram as operações numa pilha. Estes códigos podem ser adaptados para qualquer linguagem de programação [9].

Algoritmo 3.1: Verificação se a pilha está vazia (função EMPTY(S))

```
   Input: A variável que contém a pilha (S)
   Output: Verdadeiro ou falso
1  begin
2  |  if topo de S = 0 then
3  |  |  return true
4  |  else
5  |  |  return false
6  |  endif
7  end
```

3.2 Pilhas em C

Antes de programar a solução de um problema que usa uma pilha, é necessário determinar como representar uma pilha usando as estruturas de dados existentes na linguagem de programação. Uma pilha é um conjunto ordenado de itens, e a linguagem C já contém um tipo de dado que representa um conjunto ordenado de itens: o vetor. Então, sempre que for necessário utilizar a estrutura

de pilhas para resolver um problema pode-se utilizar o vetor para armazenar esta pilha. Mas a pilha é uma estrutura dinâmica e pode crescer infinitamente, enquanto um vetor na linguagem C tem um tamanho fixo; contudo, pode-se definir este vetor com um tamanho suficientemente grande para conter esta pilha.

Algoritmo 3.2: Colocar um item na pilha (função PUSH(S,x))

```
  Input: Pilha (S) e o item a ser incluído na pilha (x)
  Output: Não tem retorno
1 begin
2 |  topo de S⟵ topo de S + 1
3 |  S(topo de S)⟵ x
4 |  return
5 end
```

Algoritmo 3.3: Retirada de um item da pilha (função POP(S))

```
  Input: Pilha (S)
  Output: Item que está no topo da pilha
1 begin
2 |  if EMPTY(S) then
3 |  |  Erro de pilha vazia
4 |  else
5 |  |  topo de S⟵ topo de S - 1
6 |  |  return S(topo de S + 1)
7 |  endif
8 end
```

Algoritmo 3.4: Pega o item do topo da pilha mas não desempilha (função STACKPOP(S))

```
  Input: Pilha (S)
  Output: Item que está no topo da pilha
1 begin
2 |  if EMPTY(S) then
3 |  |  Erro de pilha vazia
4 |  else
5 |  |  return S(topo de S)
6 |  endif
7 end
```

Algoritmo 3.5: Tamanho da pilha (função SIZE(S))

```
Input: A variável que contém a pilha (S)
Output: Quantidade de itens da pilha (topo de S)
1 begin
2 |  return topo de S
3 end
```

O programa 3.1 apresenta um exemplo de programa em C para manipulação de pilhas.

Programa 3.1: Exemplo de manipulação de pilha

```
/* programa_pilha_01.c */

#include <stdio.h>

void push(int valor);
int pop(void);
int size(void);
int stacktop(void);

int pilha[20];
int pos=0;

void push(int valor)
{
    pilha[pos]=valor;
    /* Empilha um novo elemento. Não é verificada a capacidade
       máxima da pilha.*/
    pos++;
    return;
}

int pop()
{
    /* Retorna o elemento do topo da filha. Não é verificado
       o final da pilha. */
    return (pilha[--pos]);
}

int size()
{
    return pos; /* retorna o topo da pilha */
}
```

```
33
   int stacktop() /* retorna o topo da pilha sem desempilhar */
   {
    return pilha[pos];
   }
38
   int main(int argc, char ** argv )
   {
   printf("\nColocados dados na pilha");
   push(1);
43 push(2);
   push(3);

   printf("\nTamanho da pilha %d", size());

48 printf("\nPegando dado da pilha: %d", pop());
   printf("\nPegando dado da pilha: %d", pop());
   printf("\nPegando dado da pilha: %d", pop());

   printf("\nTamanho da pilha %d", size());
53 return 0;
   }
```

Uma pilha em C pode ser declarada como uma estrutura contendo dois objetos: um vetor para armazenar os elementos da pilha e um inteiro para indicar a posição atual do topo da pilha (programa 3.2).

Programa 3.2: Exemplo de manipulação de pilha com estrutura

```
   /* programa_pilha_02.c */

   #include <stdio.h>
   #include <stdlib.h>
5  #define TAMANHO_PILHA 100

   /* Estrutura que irá conter a pilha de informações */
   struct pilha
   {
10     int topo;
       int itens[TAMANHO_PILHA];
   };

   int empty(struct pilha *p)
15 {
       if( p->topo == -1
```

```
    {
        return 1;
    }
    return 0;
}

int pop(struct pilha *p)
{
    if( empty(p) )
    {
        printf("\nPilha vazia");
        exit(1);
    }
    /* retorna o item da pilha atual e diminui a posição da pilha */
    return (p->itens[p->topo--]);
}

void push(struct pilha *p, int e)
{
    if( p->topo == (TAMANHO_PILHA - 1))
    {
        printf("\nEstouro da pilha");
        exit(1);
    }
    /* após verificar se não haveria estouro na capacidade da pilha,
       é criada uma nova posição na pilha e o elemento é armazenado */
    p->itens[++(p->topo)] = e;
    return;
}

int size(struct pilha *p)
{
    /* sempre lembrando que na linguagem C o índice de um
       vetor começa na posição 0 */
    return p->topo+1;
}

int stackpop(struct pilha *p)
{
    return p->itens[p->topo];
}

int main(void)
{
    struct pilha x;
    x.topo = -1;

    push(&x,1);
```

```
    push(&x,2);
    push(&x,3);

    printf("\nTamanho da pilha %d", size(&x));
    printf("\nElemento do topo da fila %d", stackpop(&x));

    printf("\n%d", pop(&x));
    printf("\n%d", pop(&x));
    printf("\n%d", pop(&x));
    printf("\n%d", pop(&x));
    return 0;
    }
```

3.3 Exercícios

1. Dada uma pilha P, construir uma função que inverte a ordem dos elementos dessa pilha, utilizando apenas uma estrutura auxiliar. Definir adequadamente a estrutura auxiliar e prever a possibilidade da pilha estar vazia.

2. Construir uma função que troca de lugar o elemento que está no topo da pilha com o que está na base da pilha. Usar apenas uma pilha como auxiliar.

3. Dada uma pilha contendo números inteiros quaisquer, construir uma função que coloca os pares na base da pilha e os ímpares no topo da pilha. Usar duas pilhas como auxiliares.

4. Fila

"A sutileza do pensamento consiste em
descobrir a semelhança das coisas diferentes
e a diferença das coisas semelhantes."
Charles de Montesquieu

Uma fila é um conjunto ordenado de itens a partir do qual se podem eliminar itens numa extremidade - início da fila - e no qual se podem inserir itens na outra extremidade - final da fila.

Ela é uma *prima* próxima da pilha, pois os itens são inseridos e removidos de acordo com o princípio de *que o primeiro que entra é o primeiro que sai - first in, first out* (FIFO).

O conceito de fila existe no mundo real, vide exemplos como filas de banco, pedágios, restaurantes etc. As operações básicas de uma fila são:

- *insert* ou *enqueue* - insere itens numa fila (ao final).
- *remove* ou *dequeue* - retira itens de uma fila (primeiro item).
- *empty* - verifica se a fila está vazia.
- *size* - retorna o tamanho da fila.
- *front* - retorna o próximo item da fila sem retirar o mesmo da fila.

A operação **insert** ou **enqueue** sempre pode ser executada, uma vez que teoricamente uma fila não tem limite. A operação **remove** ou **dequeue** só pode ser aplicado se a fila não estiver vazia, causando um erro de *underflow* ou fila vazia se esta operação for realizada nesta situação.

Tomando a fila da figura 4.1 como exemplo, o item a apresenta a fila no seu estado inicial e é executado o conjunto de operações:

- *dequeue()* - Retorna o item A (a fila resultante é representada pelo item B)
- *enqueue(F)* - O item F é armazenado ao final da fila (a fila resultante é representada pelo item C)
- *dequeue()* - Retirado o item B da fila
- *enqueue(G)* - Colocado o item G ao final da fila (item D)

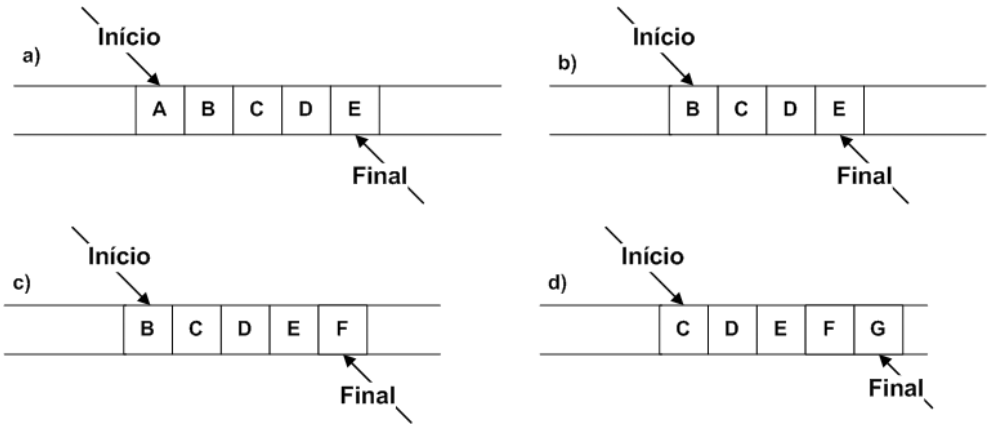

Figura 4.1: Operações numa fila

4.1 Representação de Filas com Pseudo-códigos

As operações em fila podem ser representados com os seguintes blocos de código: a operação **enqueue** (4.1), **dequeue** (4.2), **empty** (4.3), **size** (4.4) e **front** (4.5). Estes códigos podem ser adaptados para qualquer linguagem de programação [9].

Algoritmo 4.1: Inclusão de dados na fila (ENQUEUE(Q,x))

```
Input: A variável que contém a fila (Q) e o elemento a ser colocado na fila (x)
Output: Sem retorno
1  begin
2      Q[ fim de Q]⟵ x
3      if final de Q = comprimento de Q then
4          fim de Q⟵ 1
5          return
6      else
7          Q[ fim de Q]⟵ fim de Q + 1
8          return
9      endif
10 end
```

Algoritmo 4.2: Retirada de dados na fila (DEQUEUE(Q))

```
Input: A variável que contém a fila (Q)
Output: O elemento representado por na fila Q
1  begin
2      x ⟵ Q[ inicio de Q]
3      if início de Q = comprimento de Q then
4          Inicio de Q⟵ 1
5      else
6          Inicio de Q⟵ Inicio de Q + 1
7      endif
8      return x
9  end
```

Algoritmo 4.3: Verificação se a fila está vazia (função EMPTY(Q))

```
Input: A variável que contém a fila (Q)
Output: Verdadeiro ou falso
1  begin
2      if Inicio de Q = Fim de Q then
3          return true
4      else
5          return false
6      endif
7  end
```

Algoritmo 4.4: Tamanho da fila (função SIZE(Q))

```
   Input: A variável que contém a fila (Q)
   Output: Quantidade de itens da fila
1  begin
2  |  return fim de Q
3  end
```

Algoritmo 4.5: Próximo elemento da fila (função FRONT(Q))

```
   Input: A variável que contém a fila (Q)
   Output: Próximo elemento da fila (mas sem retirar da fila)
1  begin
2  |  x ⟵ Q[ inicio de Q ]
3  |  return x
4  end
```

4.2 Filas em C

A exemplo do que ocorre com estrutura em pilha, antes de programar a solução de um problema que usa uma fila, é necessário determinar como representar uma fila usando as estruturas de dados existentes na linguagem de programação. Novamente na linguagem C podemos usar um vetor. Mas a fila é uma estrutura dinâmica e pode crescer infinitamente, enquanto que um vetor na linguagem C tem um tamanho fixo. Contudo, pode-se definir este vetor com um tamanho suficientemente grande para conter a fila. No programa 4.1 é apresentado um exemplo para manipulação de filas.

Programa 4.1: Exemplo de manipulação de fila em C

```
   /* programa_fila_01.c */
   #include <stdio.h>
3  #include <stdlib.h>

   #define TAMANHO_MAXIMO 100

   struct queue
8  {
       int itens[TAMANHO_MAXIMO];
       int front,rear;
   };

13 int empty(struct queue * pq)
```

```
{
    /* se o início da fila for igual ao final da fila, a fila está vazia */
    if( pq->front == pq->rear )
    {
        return 1;
    }
    return 0;
}
void enqueue(struct queue * pq, int x)
{
    if( pq->rear + 1 >= TAMANHO_MAXIMO )
    {
        printf("\nEstouro da capacidade da fila");
        exit(1);
    }
    pq->itens[ pq->rear++ ] = x;
    return;
}

int size(struct queue * pq)
{
    return (pq->rear + 1);
}

int front(struct queue * pq)
{
    /* o primeiro elemento sempre está no início do vetor */
    return pq->itens[0];
}

int dequeue(struct queue * pq)
{
    int x, i;
    if( empty(pq) )
    {
        printf("\nFila vazia");
        exit(1);
    }
    /* Salva o primeiro elemento e refaz o arranjo dos itens, puxando o segundo elemento
       para o primeiro, o terceiro para o segundo e assim sucessivamente. */
    x = pq->itens[0];
    for( i=0; i < pq->rear; i++)
    {
        pq->itens[i] = pq->itens[i+1];
    }
    pq->rear--;
    return x;
}
```

```
int main(void)
{
    struct queue q;
    q.front = 0; q.rear = 0;
    enqueue(&q,1);
    enqueue(&q,2);
    enqueue(&q,3);
    enqueue(&q,4);

    printf("\nFila vazia %d", empty(&q));
    printf("\nTamanho da fila %d", size(&q));
    printf("\nProximo da fila %d", front(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nProximo da fila %d", front(&q));
    printf("\nTirando da fila %d", dequeue(&q));

    printf("\nFila vazia %d", empty(&q));

    printf("\n");
}
```

No programa 4.1, o vetor foi definido para comportar apenas 100 elementos, caso fosse inserido um 101º elemento, haveria o estouro da pilha mesmo após várias operações de **dequeue**. Para resolver este problema, na operação **dequeue** foi implementada uma técnica de redistribuição dos elementos na fila, de tal forma que nunca se chegue a estourar a fila caso haja várias operações de inserção ou remoção (exceto se realmente houver 100 elementos da fila e houve uma tentativa de inserção de um novo elemento). O programa 4.2 é o trecho que implementa a técnica comentada:

Programa 4.2: Reajuste da fila

```
x = pq->itens[0];
for( i=0; i < pq->rear; i++)
{
    pq->itens[i] = pq->itens[i+1];
}
pq->rear--;
```

Esta técnica é ineficiente, pois cada eliminação da fila envolve deslocar cada elemento restante na fila. Se uma fila contiver 1000 ou 2000 elementos, cada elemento retirado da fila provocará o deslocamento de todos os demais elementos. A operação de remoção de um item na fila deveria logicamente trabalhar somen-

te com aquele elemento, permanecendo os demais elementos em suas posições originais.

A solução para o problema é definir o vetor como um círculo, em vez de uma linha reta. Neste caso, os elementos são inseridos como numa fila reta, e a remoção de um elemento da fila não altera os demais elementos da fila. Com o conceito de fila circular, ao chegar ao final da fila, o ponteiro de controle da fila vai imediatamente para o início da fila novamente (se este estiver vago). As seguintes operações exemplificam a explicação (acompanhar o desenvolvimento da fila na figura 4.2), sendo o caso 1 o estado inicial da fila:

1. Estado inicial
2. *enqueue(D)* - O item D é armazenado ao final da fila
3. *enqueue(E)* - O item D é armazenado ao final da fila
4. *dequeue()* - Retirado o item A da fila
5. • *enqueue(F)* - O item F é armazenado ao final da fila
   • *enqueue(G)* - O item G é armazenado ao final da fila
6. *dequeue()* - Retirado o item B da fila
7. *enqueue(H)* - O item H é armazenado ao final da fila. Neste momento, o ponteiro da fila chegou ao final do vetor que contém a implementação da fila.
8. • *dequeue()* - Retirado o item C da fila
   • *enqueue(I)* - O item I é armazenado ao final da fila (mas no início do vetor)
9. *enqueue(K)* - O item K é armazenado ao final da fila (mas na segunda posição do vetor)

O programa 4.3 mostra a declaração da estrutura para uma fila circular.

Programa 4.3: Declaração de estrutura circular

```
#define TAMANHO_MAXIMO 100

struct queue
{
    int itens[TAMANHO_MAXIMO];
    int front,rear;
};
struct queue q;
q.front = q.rear = -1
```

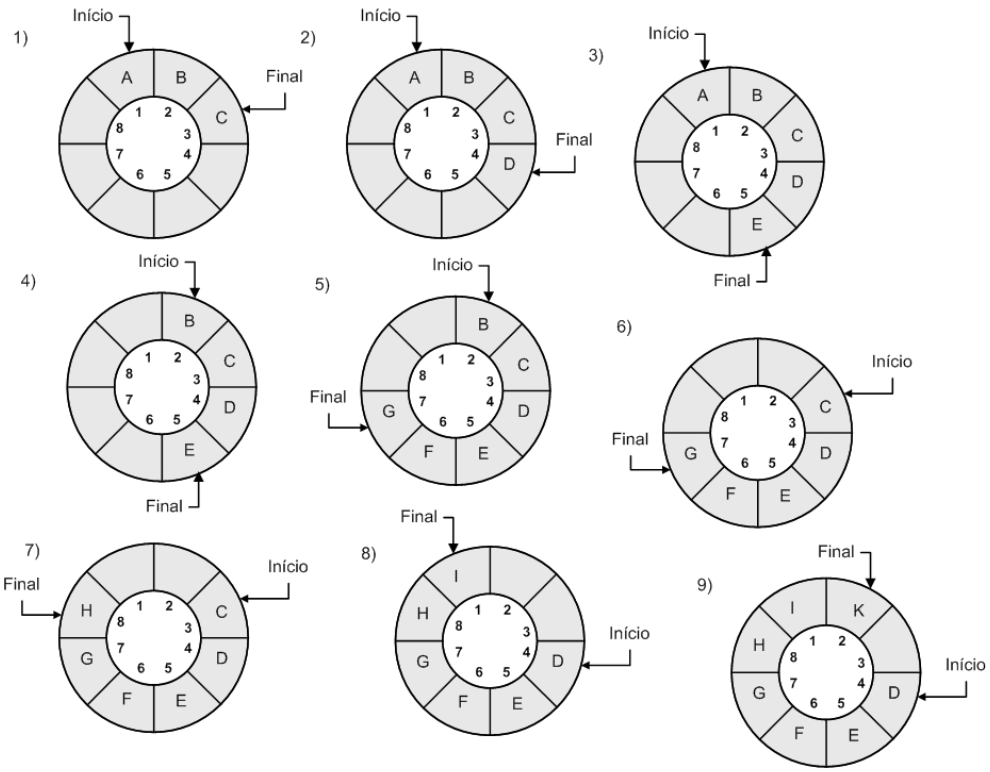

Figura 4.2: Operações numa fila circular

Desta forma, as funções de manipulação de fila (**empty**, **enqueue**, **dequeue**, **size** e **front**) devem sofrer modificações para refletir a nova condição de fila circular (programa 4.4):

Programa 4.4: Manipulação de fila circular em C

```
/* programa_fila_02.c */
#include <stdio.h>
#include <stdlib.h>

#define TAMANHO_MAXIMO 10

struct queue
{
int itens[TAMANHO_MAXIMO];
int front,rear;
};
```

```
int empty(struct queue * pq)
{
    if( pq->front == pq->rear )
    {
        return 1;
    }
    return 0;
}

void enqueue(struct queue * pq, int x)
{
    /* Inversão das posições dos ponteiros. Se o final do vetor já foi
    alcançado, então retorna-se ao início do vetor */
if( pq->rear == TAMANHO_MAXIMO-1)
{
    pq->rear = 0;
}
else
{
    pq->rear++;
}
if( pq->rear == pq->front )
{
    printf("\nEstouro da fila");
    exit(1);
}
    pq->itens[pq->rear] = x;
    return;
}

int size(struct queue * pq)
{
    /* se o final da fila ainda não alcançou o final do vetor... */
    if( pq->front <= pq->rear)
    {
        return pq->rear - pq->front; /* ... então o tamanho da fila é o final
                                        da fila menos o início da fila... */
}


    /* ... se não, quer dizer que o ponteiro de final da fila já alcançou o final do vetor
       e foi reposicionado para o início do vetor, então o tamanho da fila é a quantidade
       de itens que ainda restam até chegar ao final do vetor somando itens que estão
       no início do vetor */
    return pq->rear + pq->front;
}

int front(struct queue * pq)
{
```

```
    return pq->itens[pq->front+1];
}

int dequeue(struct queue * pq)
{
    int x, i;
    if( empty(pq) )
    {
        printf("\nFila vazia");
        exit(1);
    }

    /* Inversão das posições dos ponteiros. Se o final do vetor já foi alcançado,
      então retorna-se ao início do vetor */
    if( pq->front == TAMANHO_MAXIMO - 1)
    {
        pq->front = 0;
    }
    else
    {
        pq->front++;
    }
    return (pq->itens[ pq->front ]);
}


int main(void)
{
    struct queue q;
    q.front = -1;
    q.rear = - 1;
    enqueue(&q,1);
    enqueue(&q,2);
    enqueue(&q,3);
    enqueue(&q,4);


    printf("\nTamanho da fila %d", size(&q));
    printf("\nProximo da fila %d", front(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nProximo da fila %d", front(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nTamanho da fila %d", size(&q));
    enqueue(&q,5);
    enqueue(&q,6);
    enqueue(&q,7);
    enqueue(&q,8);
```

```
    enqueue(&q,9);
    printf("\nTamanho da fila %d", size(&q));
    printf("\nProximo da fila %d", front(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nProximo da fila %d", front(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nTamanho da fila %d", size(&q));

    enqueue(&q,10);
    enqueue(&q,11);
    enqueue(&q,12);
    enqueue(&q,13);
    enqueue(&q,14);
    enqueue(&q,15);
    enqueue(&q,16);
    enqueue(&q,17);
    enqueue(&q,18);
    printf("\nTamanho da fila %d", size(&q));
    printf("\nProximo da fila %d", front(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nProximo da fila %d", front(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nTamanho da fila %d", size(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nTamanho da fila %d", size(&q));

    printf("\nFila vazia %d", empty(&q));

    enqueue(&q,20);
    enqueue(&q,21);
    enqueue(&q,22);
    enqueue(&q,23);
    enqueue(&q,24);
    enqueue(&q,25);
    printf("\nTamanho da fila %d", size(&q));
    printf("\nProximo da fila %d", front(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nProximo da fila %d", front(&q));
    printf("\nTirando da fila %d", dequeue(&q));
```

```
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nTamanho da fila %d", size(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nTamanho da fila %d", size(&q));
    printf("\nTirando da fila %d", dequeue(&q));
    printf("\nTamanho da fila %d", size(&q));

    printf("\nFila vazia %d", empty(&q));


    printf("\n");
    return 0;
}
```

4.3 Exercícios

1. Se um vetor armazenando uma fila não é considerado circular, o texto sugere que cada operação **dequeue** deve deslocar para baixo todo elemento restante de uma fila. Um método alternativo é adiar o deslocamento até que ***rear*** seja igual ao último índice do vetor. Quando essa situação ocorre e faz-se uma tentativa de inserir um elemento na fila, a fila inteira é deslocada para baixo, de modo que o primeiro elemento da fila fique na posição 0 do vetor. Quais são as vantagens desse método sobre um deslocamento em cada operação **dequeue**? Quais as desvantagens? Reescreva as rotinas **dequeue**, **queue** e **size** usando esse método.

2. Faça um programa para controlar uma fila de pilhas.

5. Recursividade

"E não sabendo que era impossível, foi lá e fez."
Jean Cocteau

Recursão é o processo de definir algo em termos de si mesmo e é, algumas vezes, chamado de definição circular. Assim, pode-se dizer que o conceito de algo recursivo está dentro de si, que por sua vez está dentro de si e assim sucessivamente, infinitamente.

O exemplo a seguir define o ancestral de uma pessoa:

- Os pais de uma pessoa são seus ancestrais (caso base);
- Os pais de qualquer ancestral são também ancestrais da pessoa inicialmente considerada (passo recursivo).

Definições como estas são normalmente encontradas na matemática. O grande apelo que o conceito da recursão traz é a possibilidade de dar uma definição finita para um conjunto que pode ser infinito [15]. Um exemplo aritmético:

- O primeiro número natural é zero.
- O sucessor de um número natural é um número natural.

Na computação o conceito de recursividade é amplamente utilizado, mas difere da recursividade típica por apresentar uma condição que provoca o fim do ciclo recursivo. Essa condição deve existir, pois, devido às limitações técnicas que o computador apresenta, a recursividade é impedida de continuar eternamente.

5.1. Função para cálculo de Fatorial

Na linguagem C, as funções podem chamar a si mesmas. A função é recursiva se um comando no corpo da função a chama. Para uma linguagem de computador ser recursiva, uma função deve poder chamar a si mesma. Um exemplo simples é a função fatorial, que calcula o fatorial de um inteiro. O fatorial de um número N é o produto de todos os números inteiros entre `1` e `N`. Por exemplo, 3 fatorial (ou `3!`) é `1 * 2 *3 = 6`. O programa 5.1 apresenta uma versão iterativa para cálculo do fatorial de um número.

Programa 5.1: Fatorial (versão iterativa)

```
1  int fatorialc ( int n )
   {
       int t, f;
       f = 1;
       for ( t = 1; t<=n; t++ )
6          f = f * t
       return f;
   }
```

Mas multiplicar `n` pelo produto de todos os inteiros a partir de `n-1` até 1 resulta no produto de todos os inteiros de `n` a 1. Portanto, é possível dizer que fatorial:

- 0! = 1
- 1! = 1 * 0!
- 2! = 2 * 1!
- 3! = 3 * 2!
- 4! = 4 * 3!

Logo o fatorial de um número também pode ser definido recursivamente (ou por recorrência) através das seguintes regras (representação matemática) [15, 1]:

- n! = 1, se n = 0
- n! = n * (n-1)! , se n > 0

O programa 5.2 mostra a versão recursiva do programa fatorial.

Programa 5.2: Fatorial (versão recursiva)

```
int fatorialr( int n)
{
    int t, f;
    /* condição de parada */
    if( n == 1 || n == 0)
    {
        return 1;
    }
    f = fatorialr(n-1)*n; /* chamada da função */
    return f;
}
```

A versão não-recursiva de fatorial deve ser clara. Ela usa um laço que é executado de 1 a `n` e multiplica progressivamente cada número pelo produto móvel.

A operação de fatorial recursiva é um pouco mais complexa. Quando `fatorialr` é chamada com um argumento de 1, a função devolve 1. Caso contrário, ela devolve o produto de `fatorialr(n-1)*n`. Para avaliar essa expressão, `fatorialr` é chamada com `n-1`. Isso acontece até que `n` se iguale a 1 e as chamadas à função comecem a retornar.

Calculando o fatorial de 2, a primeira chamada a `fatorialr` provoca uma segunda chamada com o argumento 1. Essa chamada retorna 1, que é, então, multiplicado por 2 (o valor original e `n`). A resposta então é 2.

Para melhor entendimento, é interessante ver como o programa é executado internamente no computador. No caso do programa iterativo (programa 5.1) é necessário duas variáveis `f` e `t` para armazenar os diversos passos do processamento. Por exemplo, ao calcular fatorial de 6, o computador vai passar sucessivamente pelos seguintes passos (tabela 5.1).

Tabela 5.1: Cálculo de fatorial de 6

t	f
1	1
2	2
3	6
4	24
5	120
6	720

No programa recursivo (5.2) nada disto acontece. Para calcular o fatorial de 6, o computador tem de calcular primeiro o fatorial de 5 e só depois é que faz a multiplicação de 6 pelo resultado (120). Por sua vez, para calcular o fatorial de 5, vai ter de calcular o fatorial de 4. Resumindo, aquilo que acontece internamente é uma expansão seguida de uma contração:

```
fatorialr(6)
6 * fatorialr(5)
6 * 5 * fatorialr(4)
6 * 5 * 4 * fatorialr(3)
6 * 5 * 4 * 3 * fatorialr(2)
6 * 5 * 4 * 3 * 2 * fatorialr(1)
6 * 5 * 4 * 3 * 2 * 1
6 * 5 * 4 * 3 * 2
6 * 5 * 4 * 6
6 * 5 * 24
6 * 120
720
```

Quando uma função chama a si mesma, novos parâmetros e variáveis locais são alocados na pilha e o código da função é executado com essas novas variáveis. Uma chamada recursiva não faz uma nova cópia da função; apenas os argumentos são novos. Quando cada função recursiva retorna, as variáveis locais e os parâmetros são removidos da pilha e a execução recomeça do ponto da chamada à função dentro da função.

5.2 Número triangular

Pitágoras, matemático e filósofo grego, demonstrou várias propriedades matemáticas, entre elas a propriedade dos números triangulares. Um número triangular é um número natural que pode ser representado na forma de triângulo equilátero. Para encontrar o *n-ésimo* número triangular a partir do anterior basta somar-lhe *n* unidades. Os primeiros números triangulares são 1, 3, 6, 10, 15, 21, 28. O *n-ésimo* termo pode ser descoberto pela fórmula a seguir:

$$T_n = \sum_{k=1}^{n} k = 1 + 2 + 3 + \ldots + (n-2) + (n-1) + n = \frac{n(n+1)}{2} [17]$$

Estes números são chamados de triangulares pois podem ser visualizados como objetos dispostos na forma de um triângulo (figura 5.1).

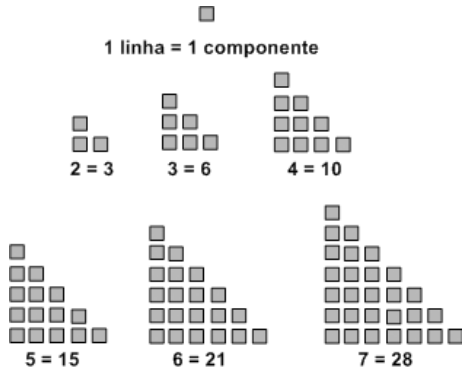

Figura 5.1: Números triangulares

Supondo que se esteja buscando o quinto elemento (dado pelo número 15), como descobrir este elemento? Basta distribuir entre as linhas e colunas conforme a figura 5.2 (5+4+3+2+1 = 15).

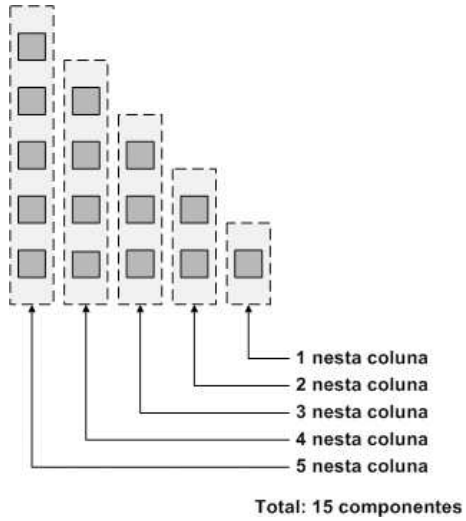

Figura 5.2: Descobrindo o quinto elemento triangular

Este é um processo repetitivo, dado por um programa simples (programa 5.3).

Programa 5.3: Descobrindo o número triangular (iterativo)

```
int triangulo(int n)
{
  int iTotal = 0;
  while( n > 0 )
  {
    iTotal += n;
    n--;
  }
  return iTotal;
}
```

Este é um processo recursivo [8] (figura 5.3) , pois:

1. Primeira coluna tem *n* elementos.
2. Soma-se a próxima coluna com *n*-1 elementos até que reste apenas 1 elemento.

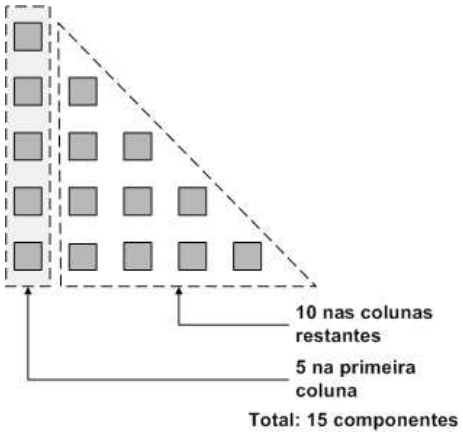

Figura 5.3: Descobrindo o quinto elemento triangular de forma recursiva

O programa 5.4 implementa a solução recursiva do problema. A figura 5.4 demostra o que ocorre a cada chamada da função *triangulo*, nela pode ser observado o retorno de cada execução da função.

Programa 5.4: Descobrindo o número triangular (recursivo)

```
int triangulo(int n)
{
   if( n == 1 )
```

```
    {
5       return n;
    }
    return n + triangulo(n-1);
}
```

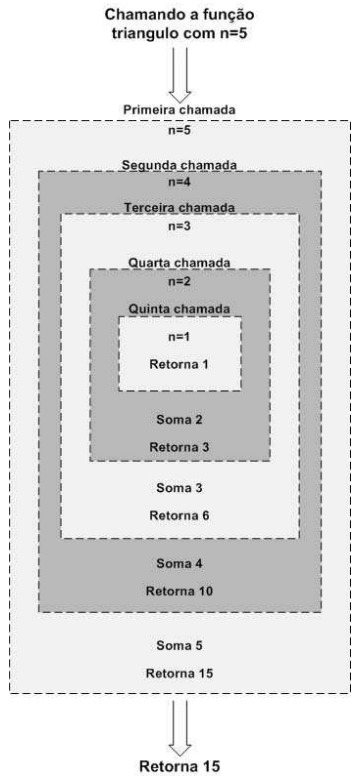

Figura 5.4: O que ocorre a cada chamada

5.3 Números de Fibonacci

Fibonacci (matemático da Renascença italiana) estabeleceu uma série curiosa de números para modelar o número de casais de coelhos em sucessivas gerações. Assumindo que nas primeiras duas gerações só existe um casal de coelhos, a seqüência de Fibonacci é a seqüência de inteiros: 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

No programa 5.5 é mostrada uma versão iterativa para calcular o n-ésimo termo da seqüência de Fibonacci.

Programa 5.5: Cálculo do n-ésimo termo de Fibonacci (versão iterativa)

```
int fibc(int n)
{
    int l,h, x, i;
    if( n <= 2)
        return 1;
    l = 0;
    h = 1;
    for(i=2; i<= n; i++)
    {
        /* Cálculo do próximo número da seqüência. */
        x = l;
        l = h;
        h = x + l;
    }
    return h;
}
```

O n-ésimo número é definido como sendo a soma dos dois números anteriores. Logo, fazendo a definição recursiva:

- fib(n) = n se n <= 2
- fib(n) = fib(n-2) + fib(n-1) se n > 2

A sua determinação recursiva impõe o cálculo direto do valor para dois elementos de base (a primeira e a segunda geração). No programa 5.6 é mostrada a versão recursiva para calcular o n-ésimo termo da seqüência de Fibonacci.

Programa 5.6: Cálculo do n-ésimo termo de Fibonacci (versão recursiva)

```
int fibr( int n )
{
    if( n <= 2)
    {
    return 1;
    }
    /* chama a si próprio 2 vezes!!! */
    return fibr(n-1) + fibr(n-2);
}
```

Esta solução (programa 5.6) é muito mais simples de programar do que a versão iterativa (programa 5.5). Contudo, esta versão é ineficiente, pois cada vez que a função `fibr` é chamada, a dimensão do problema reduz-se apenas uma unidade (de n para n-1), mas são feitas duas chamadas recursivas. Isto dá origem a uma explosão combinatorial e o computador acaba por ter de calcular o mesmo termo várias vezes.

Para calcular `fibr(5)` é necessário calcular `fibr(4)` e `fibr(3)`. Conseqüentemente, para calcular `fibr(4)` é preciso calcular `fibr(3)` e `fibr(2)`. E assim sucessivamente. Este tipo de processamento é inadequado, já que o computador é obrigado a fazer trabalho desnecessário. No exemplo, usando o programa 5.6, para calcular `fibr(5)` foi preciso calcular `fibr(4)` 1 vez, `fibr(3)` 2 vezes, `fibr(2)` 3 vezes e `fibr(1)` 2 vezes. No programa iterativo (programa 5.5), apenas era necessário calcular `fibc(5)`, `fibc(4)`, `fibc(3)`, `fibc(2)` e `fibc(1)` 1 vez. A figura 5.5 demonstra como ficaria a chamada do programa 5.6 para cálculo do sétimo termo.

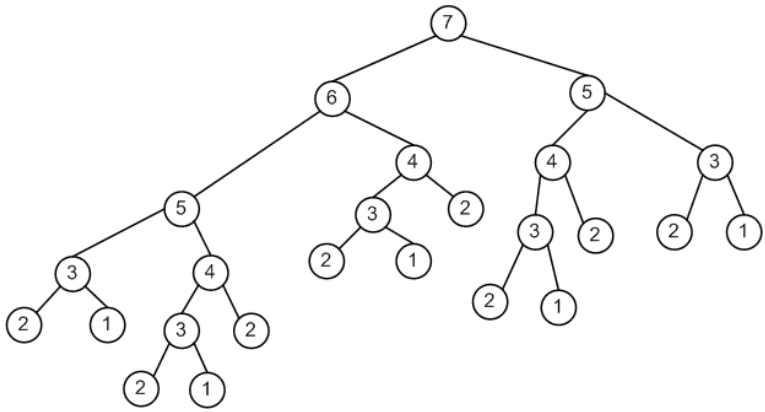

Figura 5.5: Cálculo de Fibonacci recursivo para o sétimo termo

5.4 Algoritmo de Euclides

O algoritmo de Euclides busca encontrar o máximo divisor comum (MDC) entre dois números inteiros diferentes de zero. O procedimento é simples:

1. Chame o primeiro número de *m* e o segundo número de *n*;
2. Divida *m* por *n* e chame o resto de *r*;
3. Se *r* for igual a zero, então o MDC é *n* e o procedimento termina, se não o procedimento continua;
4. Atribua *n* para *m* e *r* para *n*;
5. Recomece o procedimento do segundo passo.

Estes passos podem ser descritos conforme o algoritmo 5.1.

No programa 5.7 é vista uma versão iterativa do algoritmo de Euclides para cálculo do MDC.

Algoritmo 5.1: Algoritmo de Euclides

```
Input: M e N
Output: MDC calculado
1 begin
2     r ⟵ resto da divisão m por n
3     while r ≠ 0 do
4         m ⟵ n
5         n ⟵ r
6         r ⟵ resto da divisão m por n
7     endw
8     return n;
9 end
```

Programa 5.7: Cálculo do MDC iterativo

```
1  #include <stdio.h>

   int mdc(int m, int n)
   {
       int r;
6      while( m % n != 0)
       {
           r = m % n;
           m = n;
           n = r;
11     }
       return n;
   }

   int main(void)
16 {
       printf("60,36 = %d\n", mdc(60,36));
       printf("36,24 = %d\n", mdc(36,24));
       return 0;
   }
```

O programa 5.8 implementa o algoritmo de Euclides de forma recursiva.

Programa 5.8: Cálculo do MDC recursivo

```
   #include <stdio.h>
   int mdc(int m, int n)
5  {
       if( n == 0)
```

```
    {
        return m;
    }
    return mdc(n, m % n);
}

int main(void)
{
    printf("60,36 = %d\n", mdc(60,36));
    printf("36,24 = %d\n", mdc(36,24));
    return 0;
}
```

O MDC entre dois números *m* e *n* é também um divisor da sua diferença, *m-n*. Por exemplo: o MDC de 60 e 36 é 12, que divide 24 = 60-36. Por outro lado, o MDC dos dois números *m* e *n* é ainda o MDC do menor número (*n*) com a diferença (*m-n*). Se houvesse um divisor comum maior, ele seria igualmente divisor de *n*, contrariamente à hipótese. Portanto, é possível determinar o MDC de dois números através da determinação do MDC de números cada vez menores. O programa termina quando os números forem iguais e, neste caso, o MDC é este número. Exemplo: 60-36 = 24; 36-24 = 12; 24-12 = 12; 12 = 12. No programa 5.9 é vista uma versão do programa MDC utilizando os passos descritos (*m* sempre tem que ser maior que *n*).

Programa 5.9: Cálculo do MDC recursivo

```
#include <stdio.h>
int mdc(int m, int n)
{
    if( m == n )
    {
        return m;
    }
    if( m-n >= n)
    {
        return mdc(m-n, n);
    }
    return mdc(n, m-n);
}

int main(void)
{
    printf("60,36 = %d\n", mdc(60,36));
    printf("36,24 = %d\n", mdc(36,24));
```

```
  return 0;
}
```

5.5 Torres de Hanoi

No grande templo de Benares, embaixo da cúpula que marca o centro do mundo, repousa uma placa de latão onde estão presas três agulhas de diamante, cada uma com 50 cm de altura e com espessura do corpo de uma abelha. Em uma dessas agulhas, durante a criação, Deus colocou sessenta e quatro discos de ouro puro, com o disco maior repousando sobre a placa de latão e os outros diminuindo cada vez mais ate o topo. Essa é a torre de Brahma. Dia e noite, sem parar, os sacerdotes transferem os discos de uma agulha de diamante para outra de acordo com as leis fixas e imutáveis de Brahma, que exigem que o sacerdote em vigília não mova mais de um disco por vez e que ele coloque este disco em uma agulha de modo que não haja nenhum disco menor embaixo dele. Quando os sessenta e quatro discos tiverem sido assim transferidos da agulha em que a criação de Deus as colocou para uma das outras agulhas, a torre, o templo e os brâmanes virarão pó, e com um trovejar, o mundo desaparecerá.

Várias são as histórias que contam a origem da Torre de Hanoi. A história anterior foi utilizada pelo matemático francês Édouard Lucas em 1883 como inspiração para o jogo criado por ele [18].

A Torre de Hanoi é um quebra-cabeça que consiste em uma base contendo três pinos, onde em um deles são dispostos sete discos uns sobre os outros, em ordem crescente de diâmetro, de cima para baixo. O problema consiste em passar todos os discos de um pino para outro qualquer, usando um dos pinos como auxiliar, de maneira que um disco maior nunca fique em cima de outro menor em nenhuma situação. O número de discos pode variar sendo que o mais simples contém apenas três (figura 5.6).

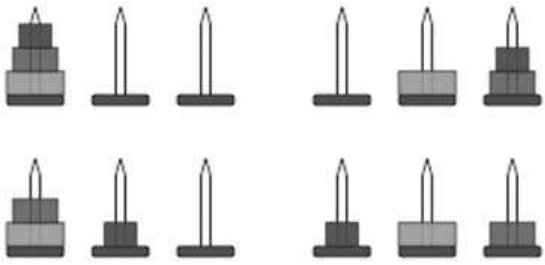

Figura 5.6: Torre de Hanoi

A solução para o problema da Torre de Hanoi com recursividade baseia-se no seguinte:

1. A única operação possível de ser executada é mover um disco de um pino para outro;
2. Uma torre com (N) discos, em um pino, pode ser reduzida ao disco de baixo e a torre de cima com (N-1) discos;
3. A solução consiste em transferir a torre com (N-1) discos do pino origem para o pino auxiliar, mover o disco de baixo do pino origem para o pino destino e transferir a torre com (N-1) discos do pino auxiliar para o pino destino. Como a transferência da torre de cima não é uma operação possível de ser executada, ela deverá ser reduzida sucessivamente até transformar-se em um movimento de disco.

O algoritmo 5.2 demostra os passos necessários para desenvolver a função recursiva. Os passos podem ser observados na figura 5.7.

Algoritmo 5.2: Passar n peças de uma torre (A) para outra (C)

```
1 begin
2 |   Passar n-1 peças da torre inicial (A) para a torre livre (B)
3 |   Mover a última peça, para a torre final (C)
4 |   Passar n-1 peças da torre B para a torre (C)
5 end
```

Baseado no algoritmo 5.2 é possível determinar o número de movimentos necessários e determinar os movimentos necessários.

O número de movimentos necessário é simples de determinar:

```
hanoi_count(n) = hanoi_count(n-1) + 1 + hanoi_count(n-1)
```

Neste caso, é possível evitar dupla recursividade (como ocorre com Fibonacci) de uma forma simples:

```
hanoi_count(n) = 2 * hanoi_count(n-1) + 1
```

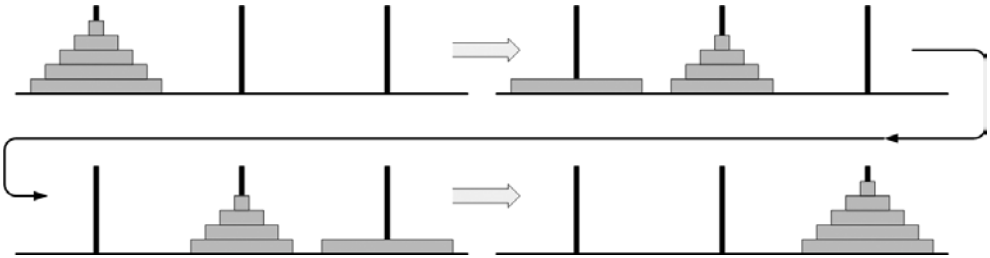

Figura 5.7: Movimentos conforme algoritmo

Para conseguir transferir todos os discos da primeira estaca à terceira é $2^n - 1$, sendo n o número de discos, portanto:

- Para solucionar um hanoi de 3 discos, são necessários 23 - 1 movimentos = 7 movimentos.
- Para solucionar um hanoi de 7 discos, são necessários 127 movimentos (27 - 1).
- Para solucionar um hanoi de 15 discos, são necessários 32767 movimentos (215 - 1).
- Para solucionar um hanoi de 64 discos, como diz a lenda, são necessários 18446744073709551615 movimentos (264 - 1) ou 585 bilhões de anos (considerando um movimento por segundo).

No programa 5.10 é vista a solução para o problema da Torre de Hanoi seguindo as definições recursivas.

Programa 5.10: Torre de Hanoi recursivo

```
/* programa_recursividade_hanoi.c */

#include <stdio.h>

void hanoi (int discos , char origem, char destino, char ajuda);

void hanoi (int discos, char origem, char destino, char ajuda)
{
    if( discos == 1)
    {
       printf("\t Mova o disco %d de %c para %c \n",discos,origem, destino);
    }
    else
```

```
    {
        hanoi(discos-1,origem,ajuda,destino);
        printf("\t Mova o disco %d de %c para %c \n",discos,origem,destino);
        hanoi(discos-1,ajuda,destino,origem);
    }
    return;
}

int main (void)
{
    int total_discos;
    printf("Informe o numero de discos:");
    scanf("%d",&total_discos);
    hanoi(total_discos,'A','B','C');
    printf("\n");
    return 0;
}
```

5.6 Curiosidades com Recursividade

O programa 5.11 utiliza de recursão para imprimir a fase da lua (cheia, minguante, crescente ou nova). Este programa foi um dos concorrentes do 15th International Obfuscated C Code Contest[1].

Programa 5.11: Natori - Imprimindo as fases da lua

```
/* programa_recursividade_curiosidade_01.c */

#include <stdio.h>
#include <math.h>
double l;

main(_,o,O)
{
    return putchar((_--+22&&_+44&&main(_,-43,_),_&&o) ?
        ( main(-43,++o,O),
        ( l=(o+21)/sqrt(3-O*22-O*O),l*l<4 &&
            (fabs(((time(0)-607728)%2551443)/405859.-4.7 +
                acos(l/2))<1.57))["  #"]))
        :10 );
}
```

1. Disponível em http://www.iocc.org/years.html

O programa 5.12 participou do mesmo concurso[2] e usa recursividade em cima de ponteiros. O código irá gerar outro código que, quando compilado, irá gerar outro código e assim sucessivamente.

Programa 5.12: Dhyanh - Saitou, aku, soku e zan

```
/* programa_recursividade_curiosidade_01.c */

                    #define/**/X
                  char*d="X0[!4cM,!"
5               "4cK‘*!4cJc(!4cHg&!4c$j"
              "8f’!&~]9e)!’|:d+!)rAc-!*m*"
            ":d/!4c(b4e0!1r2e2!/t0e4!-y-c6!"
          "+|,c6!)f$b(h*c6!(d’b(i)d5!(b*a’‘&c"
          ")c5!’b+‘&b’c)c4!&b-_$c’d*c3!&a.h’d+"
10      "d1!%a/g’e+e0!%b-g(d.d/!&c*h’d1d-!(d%g)"
       "d4d+!*l,d7d)!,h-d;c’!.b0c>d%!A‘Dc$![7)35E"
      "!’1cA,,!2kE‘*!-s@d(!(k(f//g&!)f.e5’f(!+a+)"
      "f%2g*!?f5f,!=f-*e/!<d6e1!9e0’f3!6f)-g5!4d*b"
      "+e6!0f%k)d7!+~^’c7!)z/d-+!’n%a0(d5!%c1a+/d4"
15    "!2)c9e2!9b;e1!8b>e/! 7cAd-!5fAe+!7fBe(!"
     "8hBd&!:iAd$![7S,Q0!1 bF 7!1b?’_6!1c,8b4"
     "!2b*a,*d3!2n4f2!${4 f.    ’!%y4e5!&f%"
    "d-^-d7!4c+b)d9!4c-a ’d     :!/i(’‘&d"
    ";!+l’a+d<!)l*b(d=!’ m-    a &d>!&d’"
20   "‘0_&c?!$dAc@!$cBc@!$ b     <   ^&d$‘"
   ":!$d9_&l++^$!%f3a’ n1        _      $ !&"
  "f/c(o/_%!(f+c)q*c %!       *     f &d+"
  "f$s&!-n,d)n(!0i- c-       k)     ! 3d"
  "/b0h*!H‘7a,![7*  i]        5     4  71"
25 "[=ohr&o*t*q*‘*d  *v       *r      ; 02"
 "7*~=h./}tcrsth  &t         :       r  9b"
"].,b-725-.t--//  #r       [        <  t8-"
"752793? <.~;b   ].t--+r   /        #   53"
"7-r[/9~X .v90   <6/<.v;-52/={      k  goh"
30 "./}q; u vto   hr ‘.i*$engt$        $   ,b"
";$/   =t ;v;   6   =‘it.‘;7=‘      :   ,b-"
"725  = / o‘.  .d     ;b]‘--[/+    55/   }o"
"‘.d  :  - ?5  /        }o‘.’   v/i]q    - "
"-[;  5 2 =‘ it          .       o;53-    . "
35 "v96  <7 /    =o        :          d     =o"
"--/i ]q--    [;       h.          /     = "
"i]q--[ ;v    9h       ./          <     - "
```

2. http://www.iocc.org/2000/dhyang.hint

```
"52={cj u     c&`       i  t      . o     ; "
"?4=o:d=      o--       / i       ]q      - "
"-[;54={ cj    uc&       i]q       -       -"
"[;76=i]q[;6  =vsr      u.i       /       ={"
"=),BihY_gha  ,)\0      "         ,         o [
 3217];int i,       r,w,f        ,          b        ,x ,
 p;n(){return       r <X         X          X        X X
 768?d[X(143+    X  r++          +          *d       ) %
  768]:r>2659    ?   59:         (          x        = d
  [(r++-768)%   X 947            +        768]       ) ?
  x^(p?6:0):(p = 34              X         X         X )
  ;}s(){for(x= n    ();          (         x^        ( p
?6:0))==32;x= n   ()             )   ;return x       ; }
void/**/main X   ()              {         r         = p
=0;w=sprintf (X     X           X         X X       X o
,"char*d="); for                 (      f=1;f <      * d
+143;)if(33-( b=d                [        f++ X     ] )
){if(b<93){if   X(!              p          )      o
 [w++]=34;for X(i                =         35      +
   (p?0:1);i<b;   i++            )          o
   [w++]=s();o[   w++                       ]
    =p?s():34;}    else                     X
     {for(i=92;     i<b;                    i
      ++)o[w++]=   32;}                     }
          else o      [w++                  ]
                       =10;o                [
                         w]=0               ;
                           puts(o);}
```

5.7 Cuidados com Recursividade

Ao escrever funções recursivas, deve-se ter um comando `if` em algum lugar para forçar a função a retornar sem que a chamada recursiva seja executada. Se não existir, a função nunca retornará quando chamada (equivalente a um loop infinito). Omitir o comando `if` é um erro comum quando se escrevem funções recursivas.

Isto garante que o programa recursivo não gere uma seqüência infinita de chamadas a si mesmo. Portanto, todo programa deve ter uma condição de parada não recursiva. Nos exemplos vistos, as condições de paradas não recursivas eram:

- Fatorial: 0! = 1
- Números Triangulares: n = 1
- Seqüência de Fibonacci: fib(1) = 1 e fib(2) = 1

- Euclides: n = 0
- Hanoi: discos = 1

Sem essa saída não recursiva, nenhuma função recursiva poderá ser computada. Ou seja, todo programa recursivo deve ter uma condição de parada não recursiva.

5.8 Vantagens

A maioria das funções recursivas não minimiza significativamente o tamanho do código ou melhora a utilização da memória. Além disso, as versões recursivas da maioria das rotinas podem ser executadas um pouco mais lentamente que suas equivalentes iterativas devido às repetidas chamadas à função. De fato, muitas chamadas recursivas a uma função podem provocar um estouro da pilha. Como o armazenamento para os parâmetros da função e variáveis locais está na pilha e cada nova chamada cria uma nova cópia dessas variáveis, a pilha pode provavelmente escrever sobre outra memória de dados ou de programa. Contudo, não é necessário se preocupar com isso, a menos que uma função recursiva seja executada de forma desenfreada. A principal vantagem das funções recursivas é a possibilidade de utilizá-las para criar versões mais claras e simples de vários algoritmos, embora uma solução não recursiva envolvendo outras estruturas (pilhas, filas etc.) seja mais difícil de desenvolver e mais propensa a erros. Dessa forma, ocorre um conflito entre a eficiência da máquina e a do programador. O custo da programação está aumentando e o custo da computação está diminuindo. Isto leva a um cenário que não vale a pena para um programador demandar muito tempo para elaborar programas iterativos quando soluções recursivas podem ser escritas mais rapidamente. Somente deve-se evitar o uso de soluções recursivas que utilizam recursão múltipla (Fibonacci, por exemplo).

5.9 Exercícios

1. Determine o que a seguinte função recursiva em C calcula. Escreva uma função iterativa para atingir o mesmo objetivo:

```
int func(int n)
{
    if( n == 0)
        return 0;
    return (n+ func(n-1));
}
```

2. Imagine vet como um vetor de inteiros. Apresente programas iterativos e recursivos para calcular:

   a) o elemento máximo do vetor;

   b) o elemento mínimo do vetor;

   c) a soma dos elementos do vetor;

   d) o produto dos elementos do vetor;

   e) a média dos elementos do vetor.

3. Uma cadeia *s* de caracteres é palíndrome se a leitura de *s* é igual da esquerda para a direita e da direita para a esquerda. Por exemplo, as palavras seres, arara e ama são palíndromes, assim como a sentença "A torre da derrota". Faça um programa que fique lendo palavras do usuário e imprima uma mensagem dizendo se as palavras são palíndromes ou não. O seu programa deve ter uma função recursiva com o seguinte protótipo: `int palindrome( char * palavra, int first, int last);`. Esta função recebe como parâmetros a palavra que está sendo testada se é palíndrome ou não e os índices que apontam para o primeiro e último caracteres da palavra. Talvez seja mais fácil fazer uma função com o seguinte protótipo: `int checaPalindrome(char * palavra, int last);`. Esta função recebe a palavra a ser verificada e o tamanho da palavra.

4. A função de Ackermann [2, 19] é definida recursivamente nos números não negativos como segue:

   a) a(m,n) = n + 1 Se m = 0,

   b) a(m,n) = a(m-1,1) Se m <> 0 e n = 0,

   c) a(m,n) = a(m-1, a(m,n-1)) Se m <> 0 e n <> 0

Faça um procedimento recursivo para computar a função de Ackermann. Observação: Esta função cresce muito rápido, assim ela deve ser impressa para valores pequenos de m e n.

6. Lista

"Existem três tipos de pessoas: as que
deixam acontecer, as que fazem acontecer
e as que perguntam o que aconteceu."
Provérbio escocês

Uma lista é um estrutura de dados similar a um pilha ou fila. Uma pilha ou fila tem um tamanho fixo na memória, e o acesso a um elemento da estrutura destrói o item selecionado (operação **dequeue** e **pop**). Na lista, cada elemento contém um elo (endereço) para o próximo elemento da lista. Desta forma a lista pode ser acessada de maneira randômica, podendo ser retirados, acrescidos ou inseridos elementos no meio da lista dinamicamente.

Uma lista pode ter uma entre várias formas. Ela pode ser simplesmente ligada ou duplamente ligada, pode ser ordenada ou não, e pode ser circular ou não. Se uma lista é ***simplesmente ligada***, omitimos o ponteiro anterior em cada elemento. Se uma lista é **ordenada**, a ordem linear da lista corresponde à ordem linear de chaves armazenadas em elementos da lista; o elemento minímo é o início da lista e o elemento máximo é o fim. Se a lista é **não ordenada**, os elementos podem aparecer em qualquer ordem. Em uma *lista circular*, o ponteiro anterior do início da lista aponta para o fim, e o ponteiro próximo do fim da lista aponta para o início. Desse modo, a lista pode ser vista como um anel de elementos [9].

Listas encadeadas podem ser singularmente (ou simples) ou duplamente encadeadas (ligadas). Uma lista simplesmente encadeada contém um elo (endereço) com o próximo item da lista. Uma lista duplamente encadeada contém elos (endereços) com o elemento anterior e com o elemento posterior a ele. O uso de cada tipo de lista depende da sua aplicação.

Normalmente a representação de uma lista simplesmente encadeada é dada por um endereço (nó) que contém o próximo elemento (figura 6.1).

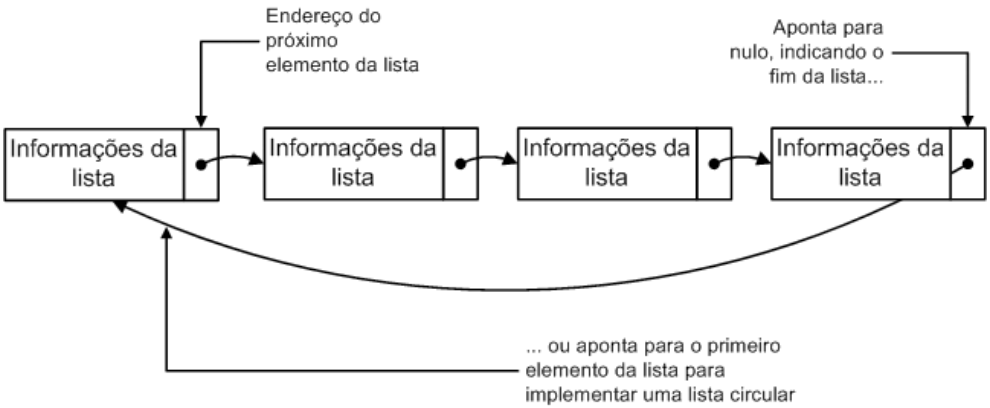

Figura 6.1: Exemplo de lista simplesmente encadeada

O último elemento da lista pode ter o endereço para um endereço nulo (caracterizando o final da lista) ou o endereço para o primeiro item lista (caracterizando uma lista circular).

Uma lista sem endereços (também conhecida como nós) é chamada de lista vazia ou lista nula. Uma observação importante: a lista simplesmente encadeada por ser utilizada para representar uma pilha ou fila de dados dinamicamente.

Listas duplamente encadeadas consistem em dados e endereços (nós) para o próximo item e para o item precedente (figura 6.2). O fato de haver dois nós tem duas vantagens principais: a lista pode ser lida em ambas as direções, o que simplifica o gerenciamento da lista. E, no caso de alguma falha de equipamento, onde uma das extremidades seja perdida, a lista pode ser reconstruída a partir da outra extremidade.

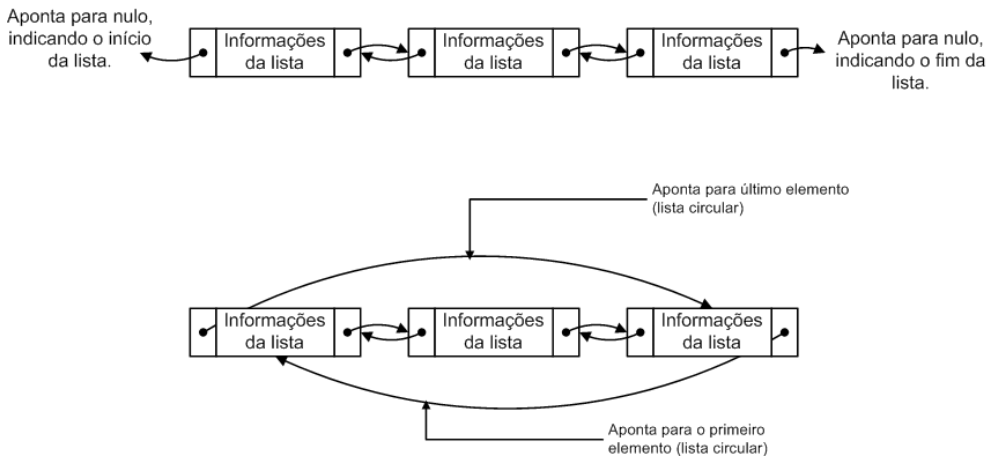

Figura 6.2: Exemplo de lista duplamente encadeada

As operações para manipulação de uma lista são:

- *insert* - insere num determinado ponto uma informação na lista.
- *add* - adiciona ao final da lista uma informação.
- *delete* - deleta um nó da lista.
- *info* - retorna uma informação da lista atual.
- *next* - desloca o ponteiro atual da lista caminhando para o final.
- *last* - desloca o ponteiro atual da lista caminhando para o início da lista.

A figura 6.3 ilustra a inserção de um novo elemento entre os elementos 1 e 2. A figura 6.4 ilustra a remoção de um elemento.

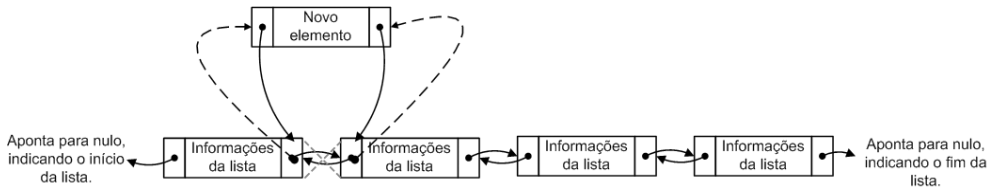

Figura 6.3: Inclusão de novo elemento

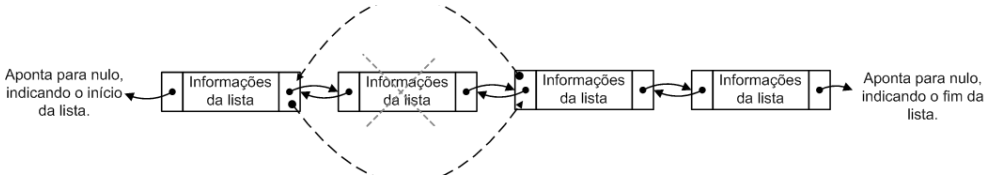

Figura 6.4: Exclusão de elemento

Nos algoritmos 6.1 e 6.2 são implementadas as operações *insert* e *delete* conforme visto nas figuras 6.3 e 6.4.

Algoritmo 6.1: Inserção numa lista duplamente encadeada

```
   Input: Lista L e elemento X
 1 begin
 2 |   Sendo N novo elemento, A elemento anterior e P o próximo elemento
 3 |   N = Alocar espaço para o novo elemento
 4 |   if É inclusão no fim da lista then
 5 |   |   L[N (campo próximo)] ⟵ NULO
 6 |   |   L[N (campo anterior)] ⟵ endereço de L[A]
 7 |   |   L[A (campo próximo)] ⟵ endereço de L[N]
 8 |   else
 9 |   |   L[N (campo próximo)] ⟵ endereço de L[P]
10 |   |   L[N (campo anterior)] ⟵ endereço de L[A]
11 |   |   L[A (campo próximo)] ⟵ endereço de L[N]
12 |   |   L[P (campo anterior)] ⟵ endereço de L[N]
13 |   endif
14 |   L[N] = X
15 |   return
16 end
```

6.1 Vetores ou alocação dinâmica?

Vetores podem ser utilizados para implementação de listas, exceto por um problema. Um vetor é considerado pelas linguagens de programação como uma única variável, mesmo um vetor que cresce dinamicamente. Por ser considerada, uma única variável, a ocupação em memória é contígua (lado a lado), portanto, não é possível alocar novos elementos nos *buracos* da memória, o que pode limitar o crescimento da lista.

A figura 6.5 apresenta a problemática do crescimento de um vetor. Considerando 90 posições de memória, um vetor ocupa os espaços compreendidos entre as posições 01 e 12. Este vetor poderá crescer até o tamanho máximo de 15 posições, que é o maior espaço livre e contíguo disponível.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40	41	42	43	44	45
46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70	71	72	73	74	75
76	77	78	79	80	81	82	83	84	85	86	87	88	89	90

Figura 6.5: Problemática do crescimento do vetor

Se em vez de um vetor, a lista for implementada com estruturas de dados (*structs* em C), é possível crescer até o tamanho máximo disponível na memória. Conforme a figura 6.6, após a lista reservar os segmentos 13,14 e 15 ela pode continuar crescendo a partir da ocupação do segmento 17,18 e 19 que eram os próximos segmentos de memória disponíveis.

Algoritmo 6.2: Remoção numa lista duplamente encadeada

```
   Input: Lista L e elemento X
1  begin
2  |  Considerando A elemento anterior e P o próximo elemento
3  |  if É remoção no fim da lista then
4  |  |  remover da lista L o elemento X
5  |  |  L[A (campo próximo)] ⟵ NULO
6  |  else
7  |  |  L[A (campo próximo)] ⟵ endereço de L[P]
8  |  |  L[P (campo anterior)] ⟵ endereço de L[A]
9  |  endif
10 |  return
11 end
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40	41	42	43	44	45
46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70	71	72	73	74	75
76	77	78	79	80	81	82	83	84	85	86	87	88	89	90

Figura 6.6: Crescimento de uma lista sem utilizar vetor

6.2 Listas em C

Na linguagem C, os vetores podem ser utilizados para representar uma lista, mas a lista também pode ser implementada através de estruturas com alocação dinâmica de memória.

Na implementação de listas em C normalmente são utilizadas estruturas que, além de conter as informações necessárias - códigos, nomes etc. -, terão campos adicionais para controle da própria lista. Os campos adicionais de controle de lista são ponteiros para a própria lista (isto é possível através da capacidade da linguagem de definição recursiva). No programa 6.1 é demonstrado como manipular uma lista simples. Na manipulação de lista simples é exigida a criação de um ponteiro para manter o início da lista.

Programa 6.1: Exemplo de manipulação de lista simples em C

```
/* programa_lista_01.c */
#include <stdio.h>
#include <stdlib.h>

struct LISTA
{
    char string[41];
    int numero;
    struct LISTA * NEXT;
};
```

```
int main(void)
{
    int i;
    struct LISTA *lista;
    struct LISTA *inicio;

    lista = calloc(1,sizeof(struct LISTA));
    if( lista == NULL )
    {
        printf("\nErro de alocacao de memoria!");
        exit(-1);
    }
    lista->NEXT = NULL;

    /* guardando o início da lista */
    inicio = lista;
    for(i=0;i<25;i++)
    {
        lista->numero = i;
        sprintf(lista->string, "Numero %02d", i);

        /* aloca o próximo elemento da lista */
        lista->NEXT = calloc(1,sizeof(struct LISTA));
        if( lista->NEXT == NULL )
        {
            printf("\nErro de alocacao de memoria!");
            exit(-1);
        }

        /* posiciona no próximo elemento */
        lista = lista->NEXT;
        lista->NEXT = NULL;
    }

    /* volta para o início da lista */
    lista = inicio;
    while(lista->NEXT != NULL )
    {
    printf("\nNumero = %d, String = %s", lista->numero, lista->string);

    /* caminha elemento a elemento da lista */
    lista = lista->NEXT;
    }

    lista = inicio;
    while( lista->NEXT != NULL )
    {
```

```
        struct LISTA *next; /* mantém referência do próximo elemento */
        next = lista->NEXT;

        /* libera o espaço do endereço atual e "limpa" o endereço
64        atribuindo NULL */
        free(lista);
        lista = NULL;
        lista = next;
    }
69
    return 0;
}
```

O programa 6.2 apresenta um programa simples para manipulação de uma lista encadeada. Além de incluir um campo adicional na estrutura para manter o endereço do elemento anterior, o tratamento com relação ao início e final de fila é realizado da forma *correta* (observe que o programa 6.1 sempre aloca um elemento a mais que não é utilizado).

Programa 6.2: Exemplo de manipulação de lista encadeada em C

```
 /* programa_lista_02.c */
#include <stdio.h>

4 #include <stdlib.h>
struct LISTA
{
    char string[41];
    int numero;
9   struct LISTA * NEXT;
    struct LISTA * LAST;
};

int main(void)
14 {
    int i;
    struct LISTA *lista;

    lista = calloc(1,sizeof(struct LISTA));
19  if( lista == NULL )
    {
        printf("\nErro de alocacao de memoria!");
        exit(1);
    }
24
```

```
    lista->NEXT = NULL;
    lista->LAST = NULL;

    for(i=0;i<25;i++)
29  {
        struct LISTA *atual;
        lista->numero = i;

        sprintf(lista->string, "Numero %02d", i
34
        /* aloca o próximo elemento da lista */
        lista->NEXT = calloc(1,sizeof(struct LISTA));
        if( lista->NEXT == NULL )
        {
39          printf("\nErro de alocacao de memoria!");
            exit(1);
        }

        /* pega o endereço do elemento atual */
44      atual = lista;
        lista = lista->NEXT;
        lista->NEXT = NULL;

        /* guarda o endereço do elemento anterior */
49      lista->LAST = atual;
    }

    lista = lista->LAST;
    free(lista->NEXT); /* descarta o último elemento alocado não utilizado */
54  lista->NEXT = NULL;

    while(1)
    {
        printf("\nNumero = %d, string = %s", lista->numero, lista->string);
59      if( lista->LAST == NULL )
        {
            break;
        }

64      /* caminha na lista do final para o início */
        lista = lista->LAST;
    }

    while(lista != NULL )
69  {
        struct LISTA *next;
        next = lista->NEXT;
```

```
        /* liberar o endereço */
        free(lista);
        lista = NULL;
        lista = next;
    }

    return 0;
}
```

O programa 6.3 contém um conjunto de funções que implementam todas as operações que podem ocorrer com uma lista (estas operações podem ser programadas de diferentes maneiras). O controle de início e fim da lista é realizado dentro do programa principal (`main`) de forma indireta (através do controle de elementos).

Programa 6.3: Funções para manipulação de listas

```
/* programa_lista_03.c */
#include <stdio.h>
#include <stdlib.h>

struct NOPTR
{
    int information;
    struct NOPTR *next;
    struct NOPTR *last;
};

void add(struct NOPTR **p, int i)
{
    struct NOPTR *aux;


     if( *p == NULL ) /* primeiro elemento da lista */
     {
         *p = (struct NOPTR *)malloc(sizeof(struct NOPTR));
         if( *p == NULL)
         {
             printf("\nErro de alocacao de memoria");
             exit(1);
         }
         (*p)->next = NULL;
         (*p)->last = NULL;
         (*p)->information = i;
     }
     else
```

```
    {
        aux = *p; /* demais elementos da lista */
        (*p)->next = (struct NOPTR *)malloc(sizeof(struct NOPTR));
        if( (*p)->next == NULL)
        {
            printf("\nErro de alocacao de memoria");
            exit(1);
        }
        *p = (*p)->next;
        (*p)->next = NULL;
        (*p)->last = aux;
        (*p)->information = i;
    }
    return;
}

void insert(struct NOPTR **p, int i)
{
    struct NOPTR *new;
    if( p == NULL)
    {
        printf("\nLista vazia, não pode ser inserido elementos");
        exit(1);
    }

    new = (struct NOPTR *)malloc(sizeof(struct NOPTR));
    if( new == NULL)
    {
        printf("\nErro de alocacao de memoria");
        exit(1);
    }
    new->information = i;
    new->last = *p;
    new->next = (*p)->next;

    /* controla inserção no final da lista */
    if( (*p)->next != NULL )
    {
        struct NOPTR *atual;
        atual = *p;
        *p = (*p)->next;
        (*p)->last = new;
        *p = atual;
    }
    (*p)->next = new;

    return;
}
```

```
void delete(struct NOPTR **p)
{
    struct NOPTR *last;
    struct NOPTR *next;

    /* salva os ponteiros para fazer o ajuste */
    last = (*p)->last;
    next = (*p)->next;
    free(p);
    *p = NULL;
    *p = last;

    (p)->next = next;

    /* controla remoção no final da lista */
    if( next != NULL )
    {
        *p = (*p)->next;
        (*p)->last = last;
    }
    return;
}

struct NOPTR * last(struct NOPTR *p)
{
    if( p->last != NULL )
    {
    return p->last;
    }
    return NULL;
}

struct NOPTR * next(struct NOPTR *p)
{
    if( p->next != NULL )
    {
        return p->next;
    }
    return NULL;
}

int info(struct NOPTR *p)
{
     return p->information;
}
```

```
int main(void)
{
    int i;
    struct NOPTR *p;
    p = NULL;
    for(i=0;i<10;i++)
    {
        add(&p,i);
    }

    printf("\nImprimindo do final para o inicio");
    for(i=0;i<9;i++)
    {
        printf("\nInformation = %d", info(p));
        p = last(p);
    }
    printf("\nInformation = %d", info(p));

    printf("\nImprimindo do inicio para o final");
    printf("\nInformation = %d", info(p));
    for(i=0;i<9;i++)
    {
        p = next(p);
        printf("\nInformation = %d", info(p));
    }

    printf("\nVoltando 1 elementos e inserindo um novo elemento");
    p = last(p);
    insert(&p,99);

    printf("\nInformation atual = %d", info(p));
    p = next(p);
    printf("\nInformation new = %d", info(p));
    p = next(p);
    printf("\nInformation atual = %d", info(p));
    printf("\nImprimindo do final para o inicio");
    for(i=0;i<10;i++)
    {
        printf("\nInformation = %d", info(p));
        p = last(p);
    }
    printf("\nInformation = %d", info(p));

    printf("\nInserindo 1 elemento e imprimindo do inicio para o final");
    insert(&p,88);
    printf("\nInformation = %d", info(p));
    for(i=0;i<11;i++)
    {
```

```
        p = next(p);
        printf("\nInformation = %d", info(p));
    }

    printf("\nVoltando 1 elemento e removendo");
    p = last(p);
    printf("\nInformation = %d", info(p));
    delete(&p);
    printf("\nInformation = %d", info(p));

    printf("\nImprimindo do final para o inicio");
    for(i=0;i<10;i++)
    {
        printf("\nInformation = %d", info(p));
        p = last(p);
    }
    printf("\nInformation = %d", info(p));

    printf("\nImprimindo do inicio para o final");
    printf("\nInformation = %d", info(p));
    for(i=0;i<10;i++)
    {
        p = next(p);
        printf("\nInformation = %d", info(p));
    }


    delete(&p);
    printf("\nImprimindo do final para o inicio");
    for(i=0;i<9;i++)
    {
        printf("\nInformation = %d", info(p));
        p = last(p);
    }
    printf("\nInformation = %d", info(p));
    return;
}
```

6.3 Exercícios

1. Conforme comentado, a lista pode ser utilizada para implementar uma fila ou uma pilha com crescimento dinâmico, então:

   a) Crie um programa para manipulação de pilhas utilizando listas

   b) Crie um programa para manipulação de filas utilizando listas.

   Observação: Lembrar que a pilha cresce e diminui ao final, e a fila cresce ao final e diminui à frente.

2. Criar uma lista circular com 20 elementos, fazer um programa para tratar esta lista.

3. Alterar o programa 6.3 para que as operações `next` e `last` controlem corretamente o início e o fim da fila.

4. Alterar o programa 6.3 para a que a função `insert` utilize a função `add` caso a inserção seja no final da lista.

7. Pesquisa

"A arte de programar consiste na arte de
organizar e dominar a complexidade."
Edsger Dijkstra

Bancos de dados existem para que, de tempos em tempos, um usuário possa localizar o dado de um registro simplesmente digitando sua chave. Há apenas um método para se encontrar informações em um arquivo (matriz) desordenado e um outro para um arquivo (matriz) ordenado.

Encontrar informações em uma matriz desordenada requer uma pesquisa seqüencial começando no primeiro elemento e parando quando o elemento procurado, ou o final da matriz, é encontrado. Esse método deve ser usado em dados desordenados, podendo ser aplicado também a dados ordenados. Se os dados foram ordenados, pode-se utilizar uma pesquisa binária, o que ajuda a localizar o dado mais rapidamente.

7.1 Pesquisa Seqüencial

Este é o método mais simples de pesquisa, e consiste em uma varredura serial da tabela (vetor, matriz ou arquivo), durante a qual o argumento de pesquisa é comparado com a chave de cada entrada até ser encontrada uma igual, ou ser atingido o final da tabela, caso a chave procurada não exista. A pesquisa seqüencial é fácil de ser codificada. O algoritmo 7.1 apresenta a implementação em pseudo-código em uma pesquisa seqüencial genérica. A função de pesquisa pode ser implementada por duas formas:

- Retornar o próprio elemento encontrado;
- Retornar o índice do elemento (no caso de um vetor)

A figura 7.1 demostra um processo de pesquisa seqüencial em um arquivo qualquer. No caso, este arquivo deveria ter um campo chave para ser utilizado na pesquisa ([6]).

Algoritmo 7.1: Pesquisa seqüencial

```
   Input: A variável que contém o vetor (V) e a chave (K)
   Output: -1 se não encontrou o elemento ou o próprio elemento
1  begin
2  |  forall the elementos de V do
3  |  |  if elemento = K then
4  |  |  |  return elemento
5  |  |  endif
6  |  endfall
7  |  return-1;
8  end
```

O desempenho deste algoritmo é bastante modesto, já que o número médio de comparações para a localização de uma entrada arbitrária é dado por $N_c = \frac{n+1}{2}$

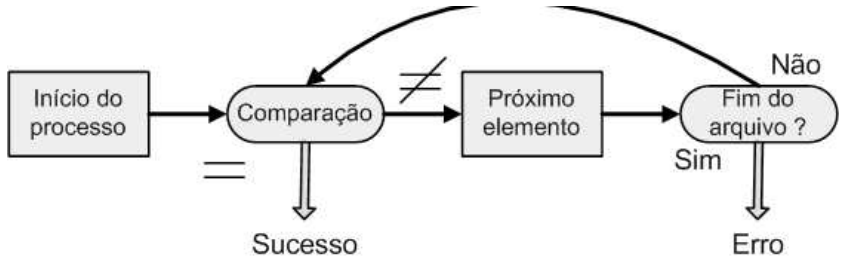

Figura 7.1: Processo de pesquisa seqüencial

A função mostrada no programa 7.1 faz uma pesquisa em um vetor de caracteres de comprimento conhecido até que seja encontrado, a partir de uma chave específica, o elemento procurado:

Programa 7.1: Função para pesquisa seqüencial

```
int pesquisa_sequencial( char * item, int contador, char chave )
{
    register int t;
    /* procura do 1º elemento do vetor até o último */
    for( t = 0; t<contador; t++)
    {
        if( chave == item[t] )
        {
            return t; /* se encontrar, retorna o índice */
        }
    }
    return -1; /* não encontrou a chave */
}
```

Esta função devolve o índice da entrada encontrada se existir alguma; caso contrário, irá devolver `-1`.

É comum o fato de algumas entradas serem mais solicitadas do que outras. Desta forma, se a tabela contiver nas suas primeiras posições as entradas mais solicitadas, o número médio de comparações será menor que se a lista estivesse distribuida aleatoriamente. Observe a tabela 7.1, onde há cinco entradas e suas freqüências de pesquisa.

Tabela 7.1: Entradas e freqüências para cálculo de comparações médias

Entrada	Freqüência
A	0,5
B	0,3
C	0,15
D	0,05
E	0,03

O número médio de comparações para a localização de uma entrada, considerando que elas aparecem na seqüência dada, será dado por: $N_c = 1 * 0,5 + 2 * 0,3 + 3 * 0,15 + 4 * 0,05 + 5 * 0,03 = 1,9$ comparações.

Caso as entradas estivessem distribuídas aleatoriamente, o número médio de comparações seria dado por $N_c = \frac{n+1}{2} = \frac{5+1}{2} = 3$ comparações.

Como não é possível conhecer antecipadamente a distribuição das entradas e suas freqüências de acesso, durante o processo de pesquisa é possível mover as entradas mais solicitadas para o início da tabela. Uma estratégia consiste em mover a entrada para o início da tabela cada vez que ela for solicitada [16, 14]. O pseudo-código 7.2 apresenta esta possibilidade.

Algoritmo 7.2: Pesquisa seqüencial com ajuste de freqüência

```
   Input: A variável que contém o vetor (V) e a chave (K)
   Output: -1 se não encontrou o elemento ou o próprio elemento
 1 begin
 2 |  forall the elementos de V do
 3 |  |  if elemento = K then
 4 |  |  |  empurrar os elementos de V em direção ao final de V
 5 |  |  |  mover o elemento para o início de V
 6 |  |  |  return elemento
 7 |  |  endif
 8 |  endfall
 9 |  return-1;
10 end
```

7.2 Pesquisa Binária

Se o dado a ser encontrado se apresentar de forma ordenada, pode ser utilizado um método muito superior para encontrar o elemento procurado. Esse método é a pesquisa binária que utiliza a abordagem *dividir e conquistar*. Ele primeiro verifica o elemento central, se esse elemento é maior que a chave, ele testa o elemento central da primeira metade; caso contrário, ele testa o elemento central da segunda metade. Esse procedimento é repetido até que o elemento seja encontrado ou que não haja mais elementos a testar (veja o algoritmo 7.3).

Por exemplo, para encontrar o número 4 na matriz 1 2 3 4 5 6 7 8 9, uma pesquisa binária primeiro testa o elemento médio, nesse caso 5. Visto que é maior que 4, a pesquisa continua com a primeira metade ou 1 2 3 4 5. O elemento central agora é 3, que é menor que 4, então a primeira metade é descartada. A pesquisa continua com 4 5. Nesse momento o elemento é encontrado. O desempenho deste algoritmo é dado pela expressão $\log_2 n$ sendo o **n** o número de elementos da tabela. A figura 7.2 exemplifica a busca binária em uma tabela ordenada; neste caso, está sendo pesquisado o elemento 34.

Algoritmo 7.3: Pesquisa binária

```
   Input: A variável que contém o vetor (V) e a chave (K)
   Output: -1 se não encontrou o elemento ou o próprio elemento
 1 begin
 2 |   baixo ⟵ 1
 3 |   alto ⟵ quantidade de elementos de V
 4 |   while baixo ≤ alto do
 5 |   |   meio ⟵ (baixo+alto)/2
 6 |   |   if K < V[ meio ] then
 7 |   |   |   alto ⟵ meio - 1
 8 |   |   else if K > V[ meio ] then
 9 |   |   |   baixo ⟵ meio + 1
10 |   |   else
11 |   |   |   return elemento em V[ meio ]
12 |   |   endif
13 |   endw
14 |   return-1;
15 end
```

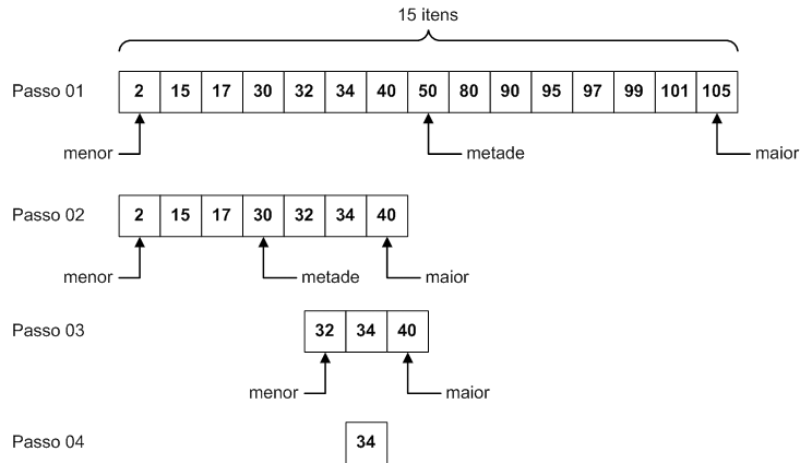

Figura 7.2: Busca binária

No programa 7.2 é demonstrada uma pesquisa binária para um vetor de caracteres.

Programa 7.2: Função para pesquisa binária

```
int binário( char * item, int contador, char chave)
{
    int baixo, alto, meio;
    baixo = 0;
    alto = contador - 1;
    while( baixo <= alto )
    {
        /* dividir para conquistar */
        meio = (baixo+alto)/2;

        /* procura nas metades até encontrar o elemento */
        if( chave<item[meio])
            alto = meio - 1; /* elemento na metade inferior */
        else if( chave>item[meio])
            baixo = meio + 1; /* elemento na metade superior */
        else
            return meio; /* elemento encontrado */
    }
    return -1;
}
```

O programa 7.2 pode ser adaptado para realizar pesquisas em qualquer tipo de dados (vetores de inteiros ou estruturas, por exemplo).

7.3 Exercícios

1. Calcule o número médio de comparações necessário para localizar uma entrada em tabelas com 15, 127, 32.767 e 35.215 entradas, para a pesquisa seqüencial e para a pesquisa binária.

2. Construa um programa que gere e preencha randomicamente vetores com 15, 127, 32.767 e 35.215 itens e comprove os cálculos obtidos na questão anterior.

8. Ordenação

"A fórmula para o sucesso é: **A=X+Y+Z**,
onde **A** é sucesso, **X** é trabalho, **Y** é lazer
e **Z** é boca fechada."
Albert Einstein

Ordenação é o processo de arranjar um conjunto de informações semelhantes em uma ordem crescente ou decrescente. Especificamente, dada uma lista ordenada `i` de `n` elementos, então: $i_1 <= i_2 <= ... <= I_n$ [13].

Algoritmo de ordenação em ciência da computação é um algoritmo que coloca os elementos de uma dada sequência em uma certa ordem - em outras palavras, efetua sua ordenação completa ou parcial. As ordens mais usadas são a numérica e a lexicográfica.

Existem várias razões para se ordenar uma sequência, uma delas é a possibilidade se acessar seus dados de modo mais eficiente.

8.1 BubbleSort

O algoritmo de ordenação *BubbleSort* é um método simples de ordenação por troca. Sua popularidade vem do seu nome fácil e de sua simplicidade. Porém, é uma das piores ordenações já concebidas. Ela envolve repetidas comparações e, se necessário, a troca de dois elementos adjacentes.

Inicialmente percorre-se a lista da esquerda para a direita, *comparando pares de elementos consecutivos*, trocando de lugar os que estão fora de ordem [12]. A tabela 8.1 exemplifica o método *BubbleSort*.

Tabela 8.1: BubbleSort - primeira varredura

troca	**L[1]**	**L[2]**	**L[3]**	**L[4]**	**L[5]**
1 com 2	10	9	7	13	5
2 com 3	9	**10**	7	13	5
4 com 5	9	7	**10**	13	5
fim da varredura	9	7	10	5	**13**

Após a primeira varredura (tabela 8.1), o maior elemento encontra-se alocado em sua posição definitiva na lista ordenada. Logo, a ordenação pode continuar no restante da lista sem considerar o último elemento (tabela 8.2).

Na segunda varredura, o segundo maior elemento encontra-se na sua posição definitiva e o restante da ordenação é realizada considerando apenas os 3 últimos elementos (7, 9 e 5). Logo são necessárias **`elementos - 1`** varreduras, pois cada varredura leva um elemento para sua posição definitiva.

Tabela 8.2: BubbleSort - segunda varredura

troca	**L[1]**	**L[2]**	**L[3]**	**L[4]**	**L[5]**
troca 1 com 2	9	7	10	5	13
troca 3 com 4	7	**9**	10	5	13
fim da varredura	7	9	5	**10**	13

O algoritmo 8.1 mostra o funcionamento do algoritmo de ordenação *BubbleSort*.

Algoritmo 8.1: Ordenação Bubble

```
   Input: A tabela (B) ser ordenada
   Output: A tabela (B) ordenada
 1 begin
 2 |  for i=comprimento de B - 1 to 1 do
 3 |  |  for j = 0 to i-1 do
 4 |  |  |  if B[j] > B[j+1] then
 5 |  |  |  |  troque B[j] com B[j+1]
 6 |  |  |  endif
 7 |  |  endfor
 8 |  endfor
 9 |  return B
10 end
```

No melhor caso, o algoritmo executa $\frac{(n^2)}{2}$ operações relevantes. No pior caso, são feitas $2n^2$ operações e no caso médio, são feitas $\frac{(5n^2)}{2}$ operações [13]. Por ser um algoritmo de ordem quadrática, não é recomendado para programas que precisem de velocidade e operem com quantidade elevada de dados.

A ordenação Bubble é dirigida por dois laços (programa 8.1). Dados que existem `iQtdElementos` elementos na matriz, o laço mais externo faz a matriz ser varrida `iQtdElementos-1` vezes. Isso garante, na pior hipótese, que todo elemento estará na posição correta quando a função terminar. O laço mais interno faz as comparações e as trocas.

Programa 8.1: Função BubbleSort

```
/* programa_bubble_01.c */
#include <stdio.h>
#include <stdlib.h>

void bubble( int piItem[], int iQtdElementos )
{
    register int i,j;
    register int iAux;
    for(i=1;i<iQtdElementos;i++)
    {
        for(j=iQtdElementos-1;j>=i;j--)
        {
            if(piItem[j-1] > piItem[j])
            {
```

```
                iAux = piItem[j-1];
                piItem[j-1] = piItem[j];
                piItem[j] = iAux;
            }
        }
    }
    return;
}

int main(void)
{
    int iContador;
    int aBubble[] = { 10, 9, 7, 13, 5};

    bubble(aBubble, 5);


    printf("Ordenado:");
    for(iContador = 0; iContador < 5; iContador++)
    {
        printf(" %d", aBubble[iContador] );
    }

    printf("\n");

    return 0;
    }
```

O programa, na forma como é apresentado, sempre varre do início ao fim da tabela, mesmo que não ocorram mais trocas. Neste caso, o programa pode ser melhorado para detectar que não houve nenhuma troca e conseqüentemente a tabela já está ordenada (programa 8.2).

Programa 8.2: Função BubbleSort melhorado

```
/* programa_bubble_02.c */
#include <stdio.h>
#include <stdlib.h>

void bubble( int piItem[], int iQtdElementos )
{
    register int i,j;
    register int iAux;
    _Bool bTroca;
    for(i=1;i<iQtdElementos;i++)
    {
```

```
        bTroca = 0; /* falso */
        for(j=iQtdElementos-1;j>=i;j--)
        {
            if(piItem[j-1] > piItem[j])
            {
                iAux = piItem[j-1];
                piItem[j-1] = piItem[j];
                piItem[j] = iAux;
                bTroca = 1; /* verdadeiro */
            }
        }

        if( !bTroca )
        {
            return;
        }
    }
    return;
}

int main(void)
{
    int iContador;
    int aBubble[] = { 10, 9, 7, 13, 5};

    bubble(aBubble, 5);

    printf("Ordenado:");
    for(iContador = 0; iContador < 5; iContador++)
    {
        printf(" %d ", aBubble[iContador] );
    }

    printf("\n");

    return 0;
    }
```

8.2 Ordenação por Seleção

A ordenação por seleção consiste em trocar o menor elemento (ou maior) de uma lista com o elemento posicionado no início da lista, depois o segundo menor elemento para a segunda posição e assim sucessivamente com os (n - 1) elementos restantes, até os últimos dois elementos. A complexidade deste algoritmo é (n - 1) + (n - 2) + … + 2 + 1 = $\frac{n(n-1)}{2}$ = n^2 comparações.

Na tabela 8.3 são vistos os passos para a ordenação de uma seqüencia de 5 inteiros. A figura 8.1 apresenta a ordenação por seleção do menor valor para o maior valor.

Tabela 8.3: Seleção - o que ocorre em cada passo

inicial	13	7	5	1	4
passo 1	**1**	7	5	**13**	4
passo 2	1	**4**	5	13	**7**
passo 3	1	4	5	13	7
passo 4	1	4	5	**7**	**13**

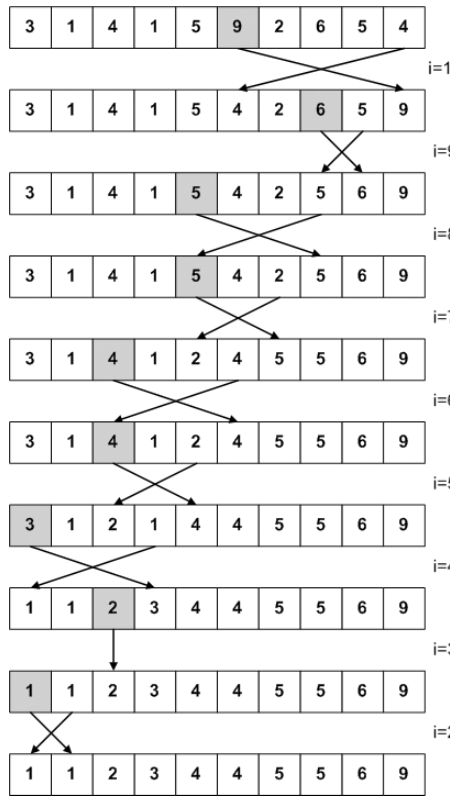

Figura 8.1: Exemplo de Ordenação por Seleção com números inteiros

Algoritmo 8.2: Ordenação por Seleção

```
   Input: A tabela (S) ser ordenada
   Output: A tabela (S) ordenada
1  begin
2      for i=1 to comprimento de S do
3          Posição ⟵ posição do menor elemento de S[i ... comprimento de S]
4          troque S[i] com S[Posição]
5      endfor
6      return S
7  end
```

O programa 8.3 implementa o algoritmo de ordenação por inserção, utilizando dados da tabela 8.3.

Programa 8.3: Função Select

```
   /* programa_selecao_01.c */
2  #include <stdio.h>

   void selection(int piItem[],int iQtdElementos)
   {
       register int i,j, iMinimo, iAux;
7      for( i=0; i<iQtdElementos-1; i++)
       {
           iMinimo=i;
           for( j=i+1; j<iQtdElementos; j++)
           {
12             if (piItem[j] < piItem[iMinimo])
                   iMinimo=j;
               }
           }
17         iAux = piItem[i];
           piItem[i] = piItem[iMinimo];
           piItem[iMinimo] = iAux;
       }
       return;
22 }

   int main(void)
   {
       int iContador;
27     int aSelect[] = { 13,7,5,1,4 };

       selection(aSelect, 5);
```

```
    printf("Ordenado:");
32  for(iContador = 0; iContador < 5; iContador++)
    {
        printf(" %d ", aSelect[iContador] );
    }

37  printf("\n");

    return 0;
}
```

8.3 Ordenação por Inserção

A ordenação por inserção é um algoritmo simples e indicado para listas pequenas de valores a serem ordenados.

Inicialmente, ela ordena os dois primeiros membros da lista, em seguida o algoritmo insere o terceiro membro na sua posição ordenada com relação aos dois primeiros membros. Na sequência, é inserido o quarto elemento na lista dos três primeiros elementos e o processo continua até que toda a lista esteja ordenada.

O algoritmo de inserção funciona da mesma maneira com que muitas pessoas ordenam cartas em um jogo de baralho como o pôquer. Uma das características deste algoritmo é o menor número de trocas e comparações se a lista estiver ordenada (parcialmente).

A figura 8.2 [11] demonstra o processo de ordenação utilizando como exemplo 10 números inteiros.

O número de comparações é n^2 no pior caso e no melhor caso $2(n - 1)$ comparações [13], no caso médio o número de comparações é $\frac{n^2}{4}$.

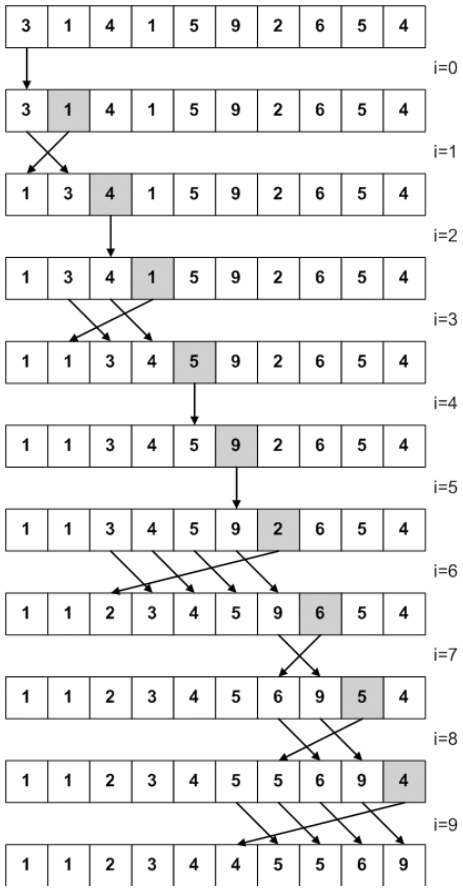

Figura 8.2: Exemplo de Ordenação por Inserção com números inteiros

Na tabela 8.4 é visto como ocorre a ordenação por inserção. Primeiro os elementos 13 e 7 são ordenados (passo 1). Na sequência, o elemento 5 é colocado no início da lista (passo 2); depois o elemento 1 é movido para o início da lista (passo 3) e, finalmente, o elemento 4 é inserido entre os elementos 1 e 5. A figura 8.3 ilustra todo o processo.

A figura 8.4 (inspirada em [6]) e o algoritmo 8.3 demonstram os passos para a ordenação por inserção e uma implementação em C pode ser vista no programa 8.4.

Tabela 8.4: Inserção - o que ocorre em cada passo

inicial	13	7	5	1	4
passo 1	**7**	**13**	5	1	4
passo 2	**5**	7	13	1	4
passo 3	**1**	5	7	13	4
passo 4	1	**4**	5	7	13

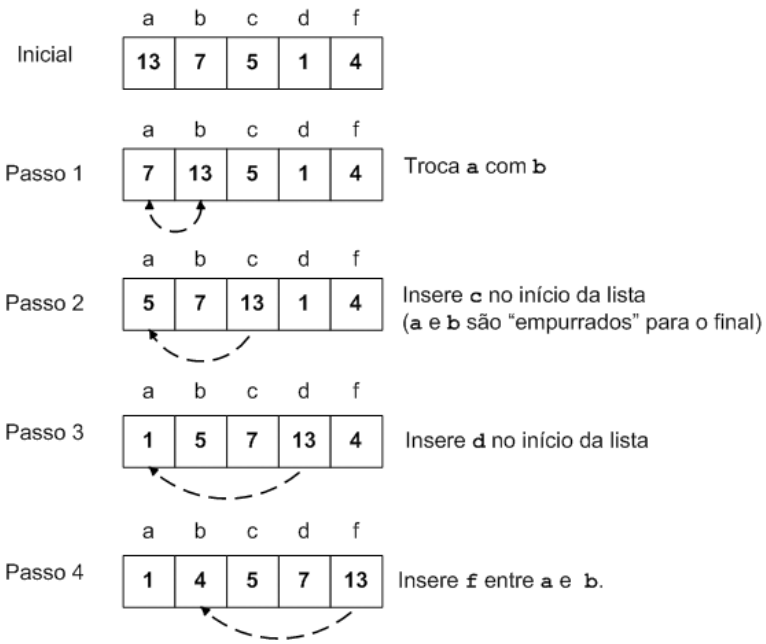

Figura 8.3: Seqüência de ordenação por inserção

Programa 8.4: Função Insert

```
/* programa_insercao_01.c */
#include <stdio.h>

void insert(int piItem[], int iQtdElementos)
{
    register int i,j, iAux;
    for( i=1; i<iQtdElementos; i++)
    {
        iAux = piItem[i];
```

```
        for( j=i-1; j>=0 && iAux < piItem[j]; j--)
        {
            piItem[j+1]=piItem[j];
        }
14      piItem[j+1]=iAux;
    }
    return;
}

19  int main(void)
{
    int iContador;
    int aInsert[] = { 13,7,5,1,4 };

24  insert(aInsert, 5);

    printf("Ordenado:");
    for(iContador = 0; iContador < 5; iContador++)
    {
29  printf(" %d ", aInsert[iContador] );
    }

    printf("\n");

34  return 0;
}
```

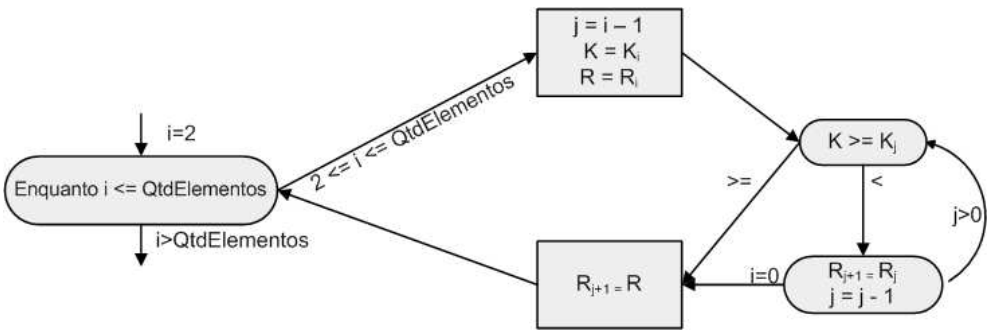

Figura 8.4: Algoritmo da ordenação por inserção

Algoritmo 8.3: Ordenação por Inserção

```
   Input: A tabela (B) ser ordenada
   Output: A tabela (B) ordenada
 1 begin
 2 |  for  i = 2 to comprimento de B do
 3 |  |  valor ⟵ B[i]
 4 |  |  insira B[i] na sequência ordenada de B[1 ... i - 1]
 5 |  |  j ⟵ i - 1
 6 |  |  while  j > 0 E B[j] > valor do
 7 |  |  |  troque B[j + 1] com B[j]
 8 |  |  |  j ⟵ j - 1
 9 |  |  endw
10 |  |  B[j+1] ⟵ valor;
11 |  endfor
12 |  return B
13 end
```

8.4 QuickSort

O algoritmo *QuickSort* é do tipo *divisão e conquista*. Um algoritmo deste tipo resolve vários problemas *quebrando* um determinado problema em mais (e menores) *subproblemas* [11].

O algoritmo, publicado pelo professor C.A.R. Hoare em 1962, baseia-se na idéia simples de partir um vetor (ou lista a ser ordenada) em dois subvetores, de tal maneira que todos os elementos do primeiro vetor sejam menores ou iguais a todos os elementos do segundo vetor. Estabelecida a divisão, o problema estará resolvido, pois aplicando recursivamente a mesma técnica a cada um dos subvetores, o vetor estará ordenado ao se obter um subvetor de apenas 1 elemento.

Os passos para ordenar uma sequência S = {a_1; a_2; a_3; …; an}é dado por [11]:

1. Seleciona um elemento do conjunto *S*. O elemento selecionado (*p*) é chamado de *pivô*.
2. Retire *p* de *S* e particione os elementos restantes de *S* em 2 seqüências distintas, *L* e *G*.

3. A partição *L* deverá ter os elementos menores ou iguais ao elemento pivô *p*, enquanto que a partição *G* conterá os elementos maiores ou iguais a *p*.
4. Aplique novamente o algoritmo nas partições *L* e *G*.

Para organizar os itens, tais que os menores fiquem na primeira partição e os maiores na segunda partição, basta percorrer o vetor do início para o fim e do fim para o início simultaneamente, trocando os elementos. Ao encontrar-se no meio da lista, tem-se a certeza de que os menores estão na primeira partição e os maiores na segunda partição.

A figura 8.5 ilustra o que se passa quando se faz a partição de um vetor com a sequência de elementos *S = { 7, 1, 3, 9, 8, 4, 2, 7, 4, 2, 3, 5 }*. Neste caso o pivô é 4, pois é o valor do elemento que está na sexta posição (e 6 é igual a $\frac{1+12}{2}$). A escolha do elemento pivô é arbitrária, pegar o elemento médio é apenas uma das possíveis implementações no algoritmo [15]. Outro método para a escolha do pivô consiste em escolher três (ou mais) elementos randomicamente da lista, ordenar esta sublista e pegar o elemento médio [13].

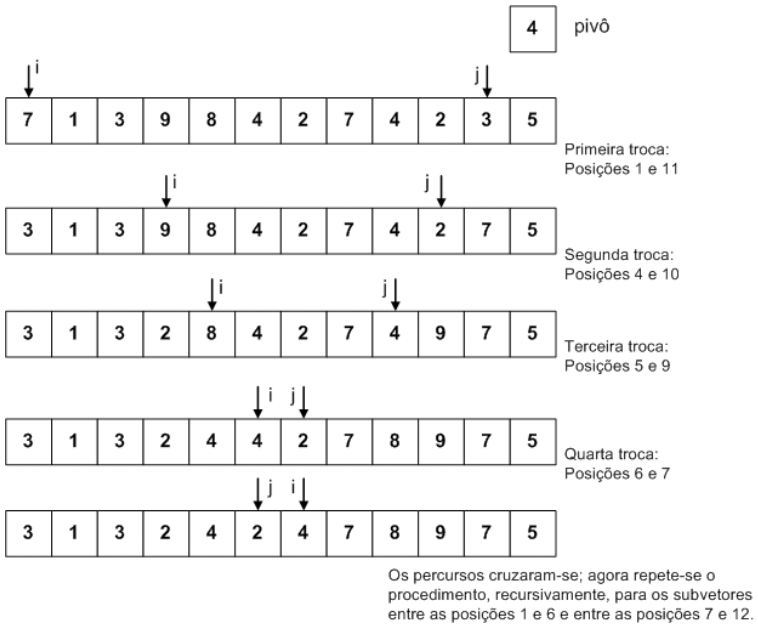

Figura 8.5: Ordenação QuickSort

O *QuickSort* pode ser implementando pelos algoritmos 8.4 e 8.5 [9]. O tempo de execução do algoritmo depende do fato de o particionamento ser balanceado ou não, e isso por sua vez depende de quais elementos são usados para particionar. Se o valor do pivô, para cada partição, for o maior valor, então o algoritmo se tornará numa ordenação lenta com um tempo de processamento n^2. A complexidade é dada por $n \log_2 n$ no melhor caso e caso médio. O pior caso é dado n^2 comparações [15, 9].

Algoritmo 8.4: QuickSort

```
   Input: Vetor V, índice inicial p, índice final r
   Output: Vetor V, ordenado
1  begin
2      if p < r then
3          q ⟵ Particiona(V, p, r)
4          QuickSort( V, p, q - 1 )
5          QuickSort( V, q+1, r)
6      endif
7      return V
8  end
```

O programa 8.5 é uma das várias implementações possíveis do algoritmo *QuickSort*. Esta versão é uma adaptação em C [13] do programa apresentado em [21]. Este programa poderá ser adaptado para qualquer conjunto de dados ou estruturas.

Programa 8.5: Ordenação QuickSort

```
   #include <stdio.h>

   void qs( char *item, int left, int right);

5  void qs( char *item, int left, int right)
   {
       int i,j;
       char x,y;
       i = left;
10     j = right;
       x = item [ (left+right)/2 ]; /* elemento pivo */

       /* partição das listas */
       do
15     {
```

```
        /* procura elementos maiores que o pivô na primeira parte*/
        while(item[i]<x && i<right)
        {
            i++;
        }

        /* procura elementos menores que o pivô na segunda parte */
        while(x<item[j] && j>left)
        {
            j--;
        }

        if(i<=j)
        {
            /* processo de troca (ordenação) */
            y = item[i];
            item[i] = item[j];
            item[j] = y;
            i++;
            j--;
        }
    } while(i<=j);

    /* chamada recursiva */
    if( left<j )
    {
        qs(item, left, j);
    }
    if( i<right )
    {
        qs(item, i, right);
    }
    return ;
}


int main(void)
{
    char aVetor[]="3490bn09685lnv 3-49580bgojfog39458=9ugkj n098=526yh";

    printf("\nAntes = [%s]", aVetor);

    /* na primeira chamada, os parâmetros iniciais são os extremos da matriz */
    qs(aVetor,0,strlen(aVetor)-1);

    printf("\nDepois = [%s]", aVetor);
    return 0;

}
```

Algoritmo 8.5: Particiona - Divisão do vetor

```
   Input: Vetor V, índice inicial p, índice final r
   Output: q, índice do ponto de partição
 1 begin
 2 |  x ⟵ V[r]
 3 |  i ⟵ p - 1
 4 |  for j = p to r - 1 do
 5 |  |  if V[j] ≤ x then
 6 |  |  |  i ⟵ i + 1
 7 |  |  |  troque V[i] com V[j]
 8 |  |  endif
 9 |  endfor
10 |  troque V[i + 1] com V[j]
11 |  return i + 1
12 end
```

8.5 MergeSort

Como o algoritmo *QuickSort*, o *MergeSort* é outro exemplo de algoritmo do tipo *divisão e conquista*, sendo é um algoritmo de ordenação por intercalação ou segmentação. A idéia básica é a facilidade de criar uma seqüência ordenada a partir de duas outras também ordenadas. Para isso, o algoritmo divide a seqüência original em pares de dados, ordena-as; depois as agrupa em sequências de quatro elementos, e assim por diante, até ter toda a seqüência dividida em apenas duas partes.

Então, os passos para o algoritmo são [11]:

1. Dividir uma seqüência em duas novas seqüências.
2. Ordenar, recursivamente, cada uma das seqüências (dividindo novamente, quando possível).
3. Combinar (*merge*) as subseqüências para obter o resultado final.

Nas figuras 8.6 [11] e 8.7 [20] podem ser vistos exemplos de ordenação utilizando os passos do algoritmo.

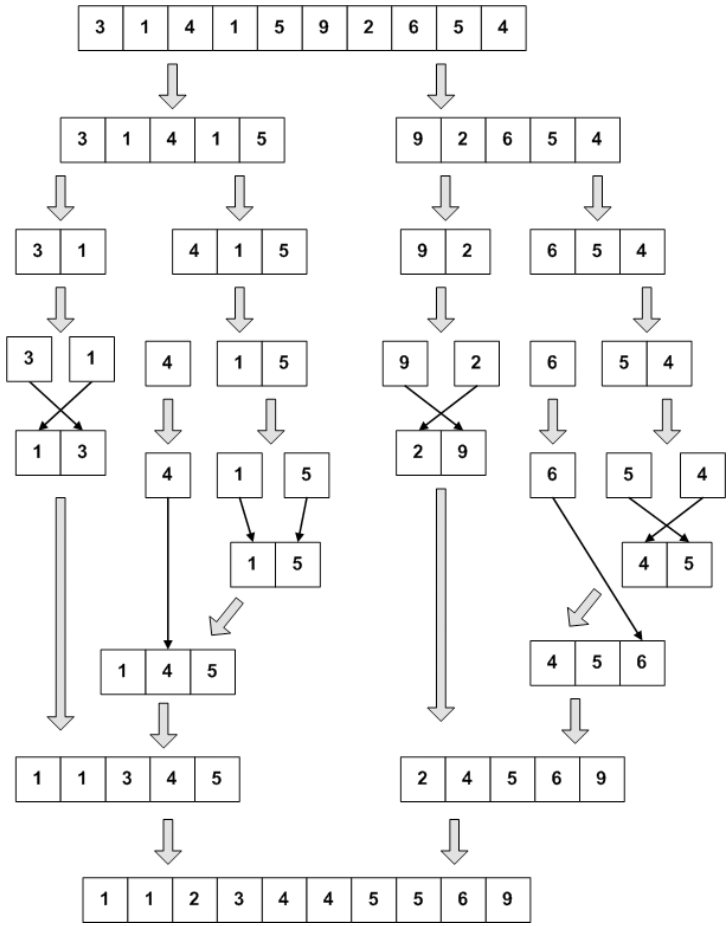

Figura 8.6: Ordenação MergeSort

A complexidade do algoritmo é dada por $n \log_2 n$ em todos os casos. A desvantagem deste algoritmo é precisar de uma lista (vetor) auxiliar para realizar a ordenação, ocasionando em gasto extra de memória, já que a lista auxiliar deve ter o mesmo tamanho da lista original.

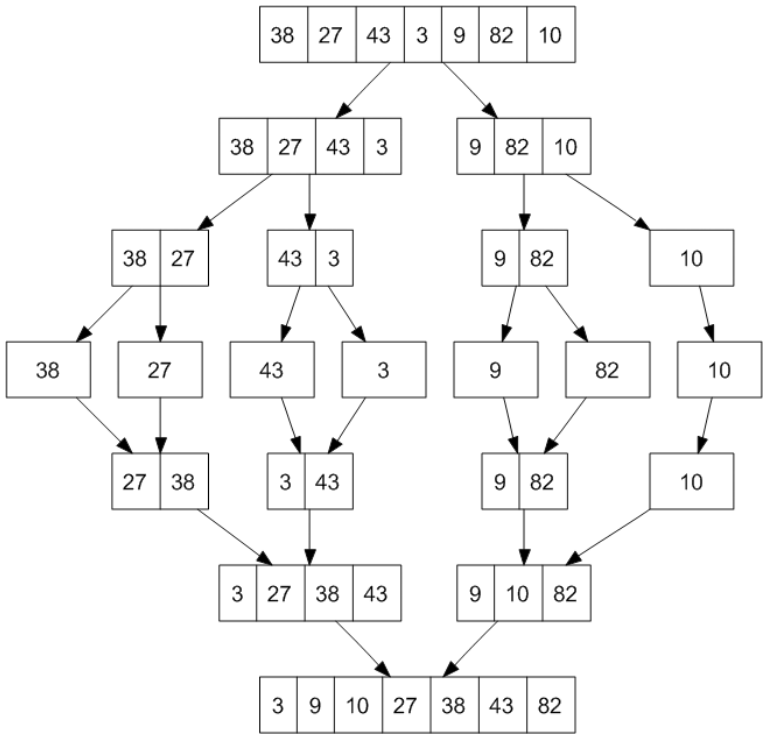

Figura 8.7: Ordenação MergeSort

O algoritmo *MergeSort* (algoritmo 8.6), a exemplo do *QuickSort*, pode fazer uso de um algoritmo auxiliar para realizar a intercalação dos itens (algoritmo 8.7). O programa 8.6 implementa estes algoritmos.

Programa 8.6: Ordenação MergeSort

```
/* programa_merge_01.c */
#include <stdio.h>
#include <stdlib.h>

4

void mergesort(int v[],int inicio,int fim) ;
void intercala(int v[], int inicio, int meio, int fim);
```

```
void mergesort(int v[],int inicio,int fim)
{
    int meio;
    if (inicio < fim)
    {
        meio = (inicio+fim)/2;
        mergesort(v,inicio,meio);
        mergesort(v,meio+1,fim);
        intercala(v, inicio, meio, fim);
    }
    return ;
}

void intercala(int v[], int inicio, int meio, int fim)
{
    /* intercalação no vetor temporário auxiliar */
    int i,j,k, *auxiliar;
    auxiliar = (int *) calloc(sizeof(int) , fim-inicio+1);
    i = inicio;
    j = meio+1;
    k = 0;
    while( i<=meio && j<=fim )
    {
        if( v[i] <= v[j] )
        {
            auxiliar[k] = v[i];
            i++;
        }
        else
        {
            auxiliar[k] = v[j];
            j++;
        }
        k++;
    }

    while( i <= meio )
    {
        auxiliar[k] = v[i];
        i++;
        k++;
    }

    while( j <= fim )
    {
        auxiliar[k] = v[j];
        j++;
        k++;
```

```
    }

    /* copia vetor intercalado para o vetor original */
    for( i = 0; i< (fim - inicio)+1; i++)
    {
        v[inicio + i] = auxiliar[i];
    }

    free(auxiliar);

    return;
}

int main(void)
{
    int iContador;
    int aMerge[] = { 9,1,11,13,17,19,21,1,3,89,24

    mergesort(aMerge, 0, 10);

    printf("Ordenado:");
    for(iContador = 0; iContador < 11; iContador++)
    {
        printf(" %d ", aMerge[iContador] );
    }

    printf("\n");

    return 0;
}
```

No algoritmo 8.8 é vista uma versão mais simplificada (a intercalação é realizada junto com o processo de ordenação) do *MergeSort*, no programa 8.7 é vista a implementação do algoritmo.

Programa 8.7: Ordenação MergeSort

```
/* programa_merge_02.c */
#include <stdio.h>
#include <stdlib.h>

void mergesort(int v[],int inicio,int fim)
{
    int i,j,k,meio,*auxiliar;
    if(inicio == fim)
    {
```

```
        return;
    }

    /* ordenação recursiva das duas metades */
    meio = (inicio+fim)/2;
    mergesort(v,inicio,meio);
    mergesort(v,meio+1,fim);

    /* intercalação no vetor temporário auxiliar */
    i = inicio;
    j = meio+1;
    k = 0;
    auxiliar = (int *) malloc(sizeof(int) * (fim-inicio+1));
    while(i<meio+1 || j<fim+1)
    {
        if( i == meio+1) /* i passou do final da primeira metade, pegar v[j] */
        {
            auxiliar[k] = v[j];
            j++; k++;
        }
        else if ( j == fim+1) /* j passou do final da segunda metade, pegar v[i] */
        {
            auxiliar[k] = v[i];
            i++; k++;
        }
        else if (v[i] < v[j]) /* v[i]<v[j], pegar v[i] */
        {
            auxiliar[k] = v[i];
            i++; k++;
        }
        else /* v[j]<=v[i], pegar v[j] */
        {
            auxiliar[k] = v[j];
            j++; k++;
        }
    }
    /* copia vetor intercalado para o vetor original */
    for( i=inicio; i<=fim; i++)
    {
        v[i] = auxiliar[i-inicio];
    }
    free(auxiliar);
    return ;
}
int main(void)
{
    int iContador;
    int aMerge[] = { 9,8,7,6,5,4,3,2,1 };
```

```
60      mergesort(aMerge, 0, 8);

        printf("Ordenado:");
        for(iContador = 0; iContador < 9; iContador++)
        {
65          printf(" %d ", aMerge[iContador] );
        }

        printf("\n");

70      return 0;
    }
```

Algoritmo 8.6: MergeSort

```
   Input: Vetor V, índice inicial inicio, índice final fim
   Output: Vetor V ordenado
1  begin
2      if inicio < fim then
3          meio ⟵ (inicio+fim)/2
4          MergeSort(V, inicio, meio)
5          MergeSort(V, meio + 1, fim)
6          Intercala(V, inicio, meio, fim)
7      endif
8      return
9  end
```

Algoritmo 8.7: Intercala

```
   Input: Vetor V, índice inicial inicio, índice final fim
   Output: q, índice do ponto de partição
 1 begin
 2 |  auxiliar ⟵ Alocação de espaço (fim-inicio + 1)
 3 |  i ⟵ inicio
 4 |  j ⟵ meio + 1
 5 |  k ⟵ 0
 6 |  while  i ≤ meio E j ≤ r do
 7 |  |  if  V[i] ≤ V[j] then
 8 |  |  |  auxiliar[k] ⟵ V[i]
 9 |  |  |  i ⟵ i + 1
10 |  |  else
11 |  |  |  auxiliar[k] ⟵ V[j]
12 |  |  |  j ⟵ j + 1
13 |  |  endif
14 |  |  k ⟵ k + 1
15 |  endw
16 |  while  i ≤ meio do
17 |  |  auxiliar[k] ⟵ V[i]
18 |  |  k ⟵ k + 1
19 |  |  i ⟵ i + 1
20 |  endw
21 |  while  j menor igual fim do
22 |  |  auxiliar[k] ⟵ V[j]
23 |  |  k ⟵ k + 1
24 |  |  j ⟵ j + 1
25 |  endw
26 |  for  i=0 to fim-inicio+1 do
27 |  |  V[inicio + i] ⟵ auxiliar[i]
28 |  endfor
29 |  Liberar espaço de auxiliar
30 |  return
31 end
```

Algoritmo 8.8: MergeSort

```
   Input: Vetor V, índice inicial inicio, índice final fim
   Output: Vetor V ordenado
 1 begin
 2     if inicio = fim then
 3         return V
 4     endif
 5     meio ⟵ (inicio+fim)/2
 6     Ordene V[a ... meio] por MergeSort
 7     Ordene V[meio + 1 ... fim] por MergeSort
 8     i ⟵ inicio
 9     j ⟵ meio + 1
10     k ⟵ 0;
11     auxiliar ⟵ Alocação de espaço (fim-inicio + 1)
12     while i < meio + 1 OU j < fim + 1 do
13         if (i ≤ meio E V[i] < V[j]) OU (j = fim + 1) then
14             auxiliar[k] ⟵ V[i]
15             i ⟵ i + 1
16             k ⟵ k + 1
17         else
18             auxiliar[k] ⟵ V[j]; j ⟵ j + 1
19             k ⟵ k + 1
20         endif
21     endw
22     for i=inicio to fim do
23         V[i] ⟵ auxiliar[i-inicio]
24     endfor
25     Liberar espaço de auxiliar
26     return V
27 end
```

8.6 Exercícios

1. *BubbleSort* - O programa 8.1 não reflete exatamente o algoritmo 8.1. Modifique o programa para que o mesmo reflita o algoritmo apresentado.

2. *BubbleSort* - Utilizando o programa do exercício anterior e tomando como base o programa 8.1, implemente a ordenação bolha oscilante [13] (também conhecido como ordenação bolha bidirecional).

3. *QuickSort* - Implemente em C os algoritmos *QuickSort* (8.4) e Particiona (8.5).

4. Fazer um programa que, utilizando ponteiros para um vetor de inteiros com 15 mil itens (gerados randomicamente), implemente todos os algoritmos de ordenação vistos. O programa deverá informar o tempo necessário para ordenar a lista nos dois sentidos (do maior para o menor e vice-versa).

5. Considere que a ordenação para n números leve t segundos. Calcule o tempo de ordenação para 10, 100, 1.000, 10.000 e 100.000 elementos para todos os algoritmos vistos.

9. Árvores Binárias

"Há três maneiras de fazer as coisas: a maneira errada, a maneira certa e uma maneira melhor."
Anônimo

Esquemas em árvores são utilizados para representar estruturas hierárquicas (árvores genealógicas, campeonatos de futebol ou organizações). Na ciência da computação, as árvores podem ser utilizadas para representar decisões, definições formais de linguagem ou mesmo para representar hierarquia entre elementos [1].

No contexto da programação e ciência da computação, é uma estrutura de dados que herda as características das topologias em árvore onde os dados estão dispostos de forma hierárquica (um conjunto de dados é hierarquicamente subordinado a outro [16]).

9.1 Analogia entre árvores

Uma árvore é composta por um elemento principal chamado raiz, que possui ligações para outros elementos, que são denominados galhos ou filhos. Estes galhos levam a outros elementos que também possuem outros galhos. O elemento que não possui galhos é conhecido como folha ou nó terminal. Observe a figura 9.1 [1].

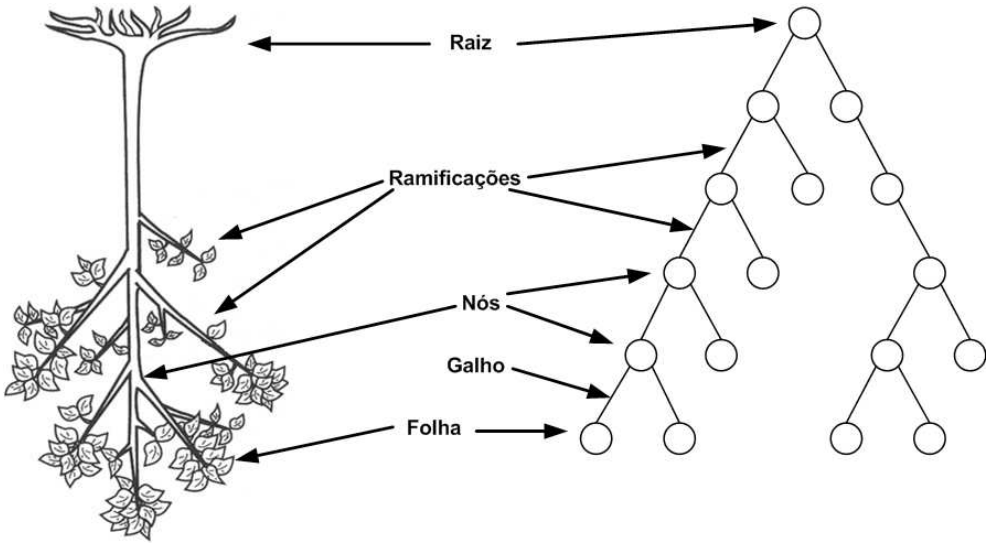

Figura 9.1: Analogia entre árvores

9.2 Árvore binária

Uma árvore binária é um conjunto finito de elementos que está vazio ou é particionado em três subconjuntos [15]:

- raiz da árvore - elemento inicial (único);
- subárvore da esquerda - se vista isoladamente compõe uma outra árvore;
- 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. Uma árvore estritamente binária com n folhas tem $2n$ - 1 nós.

A figura 9.2 apresenta um método convencional de representação de uma árvore. Nesta árvore, o elemento **A** é a raiz da árvore, a subárvore da esquerda é o elemento **B** e a da direita é representada pelo elemento **C**. Um nó sem filhos

é chamado de folha. Sendo **A** a raiz de uma árvore binária e **B** sua subárvore, é dito que **A** é pai de **B** e que **B** é filho de **A**.

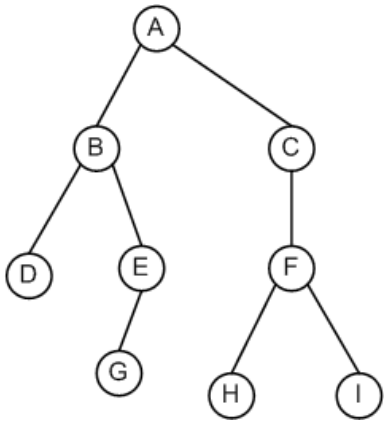

Figura 9.2: Representação de uma árvore

9.2.1 Relações

Outras relações (e conceitos) podem ser observados na figura 9.2:

- **B** e **C** são filhos de **A**.
- **B** e **C** são irmãos.
- **D** e **E** são irmãos.
- **H** e **I** são irmãos.
- T_A é a subárvore enraizada em **A**, portanto toda a árvore.
- T_F é a subárvore enraizada em **F**, que contém os nós **F**, **H** e **I**.
- Nós sem filhos são chamados de folhas, portanto os nós **D**, **G**, **H** e **I** são folhas.

Grau de saída de um nó

O número de filhos de um nó é chamado de grau de saída de um nó. Por exemplo, o nó **B** tem grau de saída 2 e o nó **C** grau 1.

Caminho

Um caminho da árvore é composto por uma seqüência de nós consecutivos (n_1,n_2,n_3,…,n_{k-1},n_k) tal que existe sempre a relação: n_j é pai de n_{j+1}. Os k nós formam um caminho de comprimento k - 1. O comprimento entre o nó **A** e o nó **H** é 3.

Nível do nó

O nível de um nó pode ser definido como o nó raiz de nível 0. Os outros nós têm um nível que é uma unidade a mais do que o nível do seu pai. Na árvore da figura 9.2 tem-se:

- Nível 0: **A**
- Nível 1: **B** e **C**
- Nível 2: **D**, **E** e **F**
- Nível 3: **G**, **H** e **I**

Altura de um nó

A altura de um nó é o comprimento do maior caminho do nó até alguns de seus descendentes. Descendentes do nó são todos os nós que podem ser alcançados caminhando-se para baixo a partir do nó. A altura de cada uma das folhas é 1. Desta maneira a altura de **A** é 4, a altura de **C** é 3 enquanto que de **E** e **F** é 2.

9.2.2 Árvore Binária Completa

Na figura 9.3 pode ser vista uma árvore completa de nível 3. Uma árvore completa é uma árvore *estritamente* binária na qual todas as folhas estão no mesmo nível k. Sendo k a profundidade da árvore, o número total de nós é 2^{k+1} - 1 e o número total de folhas é 2^k.

Embora uma árvore binária completa possua muitos nós (o máximo para cada profundidade), a distância da raiz a uma folha qualquer é relativamente pequena.

A árvore da figura 9.3 tem profundidade 3 com 15 nós (2^4 - 1) e 8 folhas (2^3).

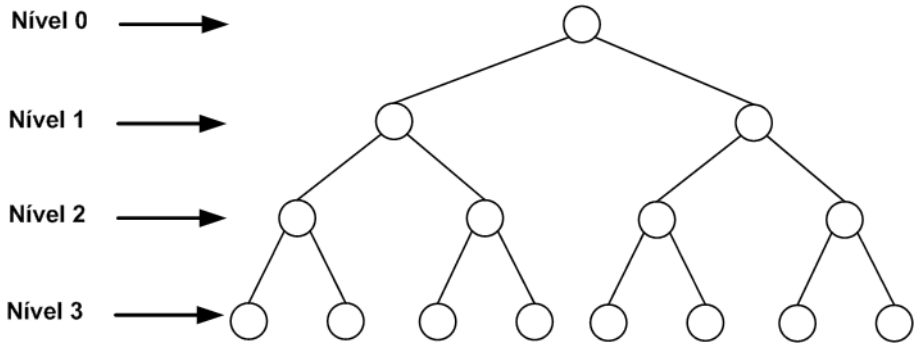

Figura 9.3: Árvore Binária completa de nível 3

9.3 Árvores de Busca Binária

Uma árvore de busca binária (*Binary search tree*) é uma árvore binária onde a informação que o nó esquerdo possui é menor ou igual à informação da chave. De forma análoga, a informação que o nó direito possui é maior ou igual à informação da chave. O objetivo de organizar dados em árvores de busca binária é facilitar a tarefa de procura de um determinado valor. A partir da raiz e de posse da informação a ser encontrada, é possível saber qual o caminho (galho) a ser percorrido até encontrar o nó desejado. Para tanto, basta verificar se o valor procurado é maior, menor ou igual ao nó que se está posicionando.

Deve-se observar que não existe uma única forma de organizar um conjunto de informações em uma árvore de busca binária, afinal, dependendo da escolha do nó raiz, obtêm-se árvores diferentes. Na figura 9.4 os valores ({ 2, 3, 5, 5, 7, 8}) são organizados em árvores de busca de duas maneiras diferentes.

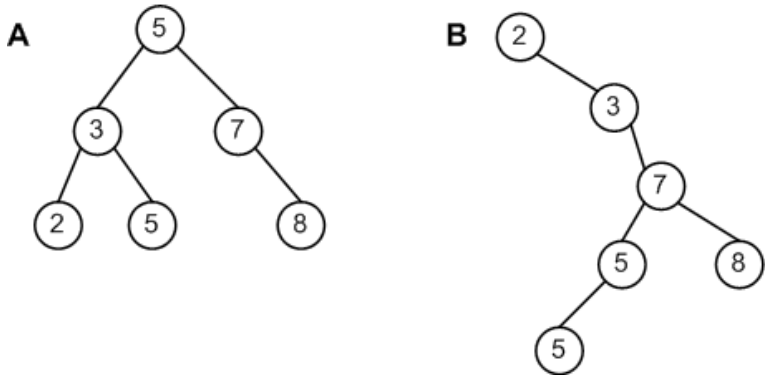

Figura 9.4: Árvore de busca binária - duas organizações diferentes

As duas árvores contêm exatamente os mesmos valores, porém possuem estruturas diferentes. Enquanto a árvore A está enraizada em um dos nós de valor 5, a árvore B está enraizada no nó de valor 2. Supondo que se está buscando o valor 8 nas árvores, as comparações seriam como se segue na tabela 9.1.

Tabela 9.1: Comparações para busca de um elemento

Árvore A	Árvore B
8 > 5	8 > 2
8 > 7	8 > 3
Encontrado!	8 > 7
	Encontrado!

Na árvore A, são realizadas menos comparações em relação à utilizada na árvore B. O melhor caso irá ocorrer quando a árvore estiver cheia, neste caso a procura de um item terá tempo proporcional a log *n*, enquanto o pior caso ocorrerá quando todos os nós da árvores apontarem somente para um dos lados, caso em que o tempo de processamento da procura será proporcional a *n*.

9.4 Operações em Árvores Binárias

9.4.1 Inserção

A inserção começa com uma busca procurando pelo valor na árvore. Se o elemento não existir na ávore, é alcançada a folha, e então inserido o valor nesta posição. Ou seja, é examinada a raiz e introduzido um novo nó na subárvore da esquerda, se o valor novo é menor do que a raiz, ou na subárvore da direita, se o valor novo for maior do que a raiz.

Os algoritmos 9.1 (versão iterativa) e 9.2 (versão recursiva) demonstram o processo de inclusão de um elemento na árvore.

Algoritmo 9.1: Inserir elemento na árvore - iterativo

```
   Input: Árvore tree, elemento x
 1 begin
 2 |  no ⟵ NULO
 3 |  atual ⟵ raiz de tree
 4 |  while atual ≠ NULO do
 5 |  |  no ⟵ atual
 6 |  |  if x < atual then
 7 |  |  |  atual ⟵ esquerda de tree
 8 |  |  else
 9 |  |  |  atual ⟵ direita de tree
10 |  |  endif
11 |  endw
12 |  Árvore vazia, primeiro elemento é a raiz
13 |  if no = NULO then
14 |  |  raiz de tree ⟵ x
15 |  else
16 |  |  if x < no (campo chave) then
17 |  |  |  esquerda de no ⟵ x
18 |  |  else
19 |  |  |  direita de no ⟵ x
20 |  |  endif
21 |  endif
22 |  return
23 end
```

Algoritmo 9.2: Inserir elemento na árvore - recursivo

```
   Input: Árvore tree, elemento x
 1 begin
 2 |  Árvore vazia, primeiro elemento é a raiz
 3 |  if tree = NULO then tree ⟵ x
 4 |  else if x < tree (campo chave) then
 5 |  |  tree ⟵ esquerda de tree
 6 |  |  Inserir(tree,x)
 7 |  else if x > tree (campo chave) then
 8 |  |  tree ⟵ direita de tree
 9 |  |  Inserir(tree,x)
10 |  endif
11 |  return
12 end
```

Os algoritmos deixam claro o processo de inserção, o novo valor é primeiro comparado com o valor da raiz. Se seu valor for menor que a raiz, é comparado então com o valor do filho da esquerda da raiz. Se seu valor for maior, então compara-se com o filho da direita da raiz. Este processo continua até que se chegue a um nó folha, e então adiciona-se o filho à direita ou à esquerda, dependendo de seu valor ser maior ou menor que o valor da folha.

9.4.2 Pesquisa

Para a busca em uma árvore binária por um valor específico deve-se examinar a raiz. Se o valor for igual à raiz, o valor existe na árvore. Se o valor for menor do que a raiz, então deve-se buscar na subárvore da esquerda, e assim recursivamente em todos os nós da subárvore.

Similarmente, se o valor for maior que a raiz, então deve-se buscar na subárvore da direita. Até alcançar o nó-folha da árvore, encontrando-se ou não o valor requerido.

Esta operação efetua *log n* operações no caso médio e *n* no pior caso quando a árvore está desequilibrada; neste caso, a árvore é considerada uma árvore degenerada.

Os algoritmos 9.3 (versão iterativa) e 9.4 (versão recursiva) demonstram o processo de pesquisa de um elemento na árvore.

Algoritmo 9.3: Pesquisar elemento na árvore - iterativo

```
   Input: Árvore tree, elemento x
   Output: Verdadeiro ou falso
 1 begin
 2 |   while tree ≠ NULO E tree (campo chave) ≠ x do
 3 |   |   if x < tree (campo chave) then  tree ⟵ esquerda de tree
 4 |   |   else if  x > tree (campo chave) then
 5 |   |   |   tree ⟵ direita de tree
 6 |   |   else
 7 |   |   |   return verdadeiro
 8 |   |   endif
 9 |   endw
10 |   return falso
11 end
```

Algoritmo 9.4: Pesquisar elemento na árvore - recursivo

```
   Input: Árvore tree, elemento x
   Output: Verdadeiro ou falso
 1 begin
 2 |  if tree = NULO then return Falso
 3 |  else if x < tree (campo chave) then
 4 |  |  tree ⟵ esquerda de tree
 5 |  |  return Pesquisar(tree,x)
 6 |  else if x > tree (campo chave) then
 7 |  |  tree ⟵ direita de tree
 8 |  |  return Pesquisar(tree,x)
 9 |  endif
10 |  return Verdadeiro
11 end
```

9.4.3 Exclusão

O processo de exclusão de um nó é mais complexo que as operações anteriores. Para excluir um nó de uma árvore binária, deve-se considerar três casos distintos para realizar a exclusão [9].

Exclusão na folha

A exclusão de um nó que se encontra no fim da árvore, isto é, que seja uma folha, é o caso mais simples de exclusão. Basta remover o nó da árvore (figura 9.5).

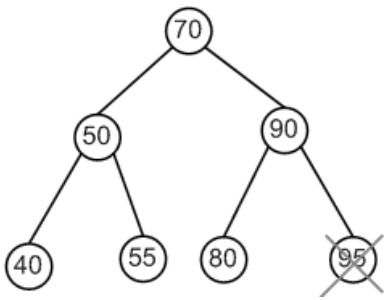

Figura 9.5: Exclusão de folha

Exclusão de nó com um filho

Caso o nó que será excluído tenha um único filho, o pai do nó (avô do filho) herda o filho. Isto é, o filho assume a posição do pai na árvore (figura 9.6).

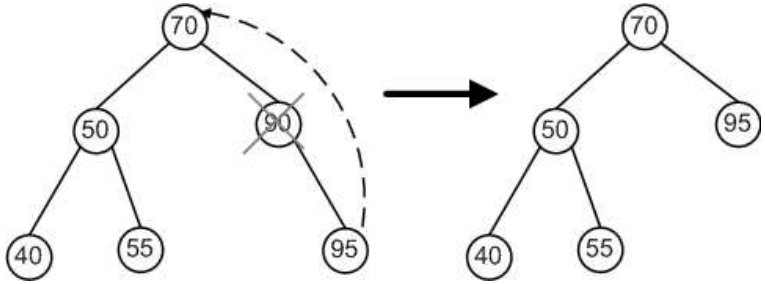

Figura 9.6: Exclusão de um nó com um filho

Exclusão de nó com dois filhos

Se o nó a ser excluído tiver dois filhos, o processo de exclusão poderá operar de duas maneiras diferentes:

- Substituir o valor do nó a ser retirado pelo valor sucessor (o nó mais à esquerda da subárvore direita).
- Substituir o valor do nó a ser retirado pelo valor antecessor (o nó mais à direita da subárvore esquerda).

Realizada a escolha, remove-se o nó sucessor (ou antecessor). A figura 9.7 exemplifica a operação. O nó com valor 50 será excluído e possui como sucessor o valor 55 e como antecessor imediato do 55 o nó com valor 53. Desta forma, o filho (53) do nó com valor 55 será promovido no lugar do nó a ser excluído (50), o nó 55 continuará em sua posição e o filho do nó 53 (no caso o nó com valor 54) será passado para o nó de valor 55. Estas operações podem ser vistas nos algoritmos 9.5 e 9.6.

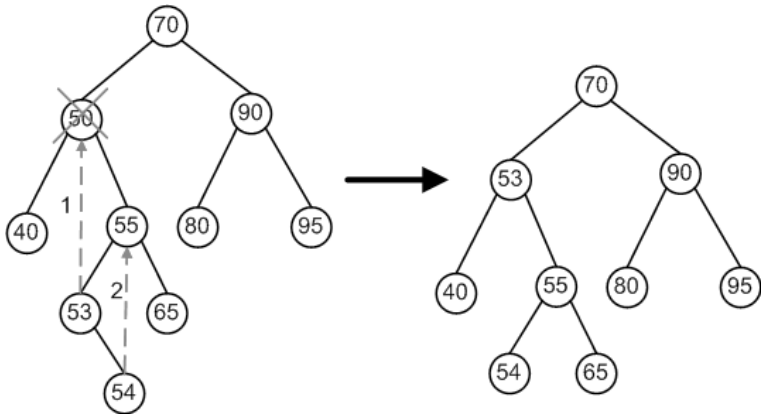

Figura 9.7: Exclusão de um nó com dois filhos

Algoritmo 9.5: Exclusão na árvore

```
   Input: Árvore tree, elemento x
   Output: Verdadeiro ou falso
 1 begin
 2 |  if tree = NULO then return Falso
 3 |  else if x < tree (campo chave) then
 4 |  |  tree ⟵ esquerda de tree
 5 |  |  return Excluir(tree,x)
 6 |  else if x > tree (campo chave) then
 7 |  |  tree ⟵ direita de tree
 8 |  |  return Excluir(tree,x)
 9 |  endif
10 |  else
11 |  |  auxiliar ⟵ tree
12 |  |  if direita de auxiliar = NULO then tree ⟵ esquerda de auxiliar
13 |  |  else if esquerda de auxiliar = NULO then
14 |  |  |  tree ⟵ direita de auxiliar
15 |  |  else
16 |  |  |  Sucessor(auxiliar, esquerda de auxiliar)
17 |  |  endif
18 |  endif
19 |  return Verdadeiro
20 end
```

Algoritmo 9.6: Sucessor

```
   Input: Árvore q, Árvore r
 1 begin
 2 |  if direita de r ≠ NULO then
 3 |  |  Sucessor(q, direita de r)
 4 |  else
 5 |  |  q ⟵ r
 6 |  |  r ⟵ esquerda de r
 7 |  endif
 8 |  return
 9 end
```

9.4.4 Maior elemento

O maior elemento da árvore, nó com o maior valor, será encontrado sempre na folha mais à direita da árvore [9]. Para encontrar o maior valor, basta procurar a partir da raiz sempre na subárvore da direita (algorimo 9.7).

Algoritmo 9.7: Maior elemento da árvore

```
   Input: Árvore tree
   Output: Maior elemento
1  begin
2      while tree ≠ NULO do
3          maior ⟵ tree (campo informação)
4          tree ⟵ direita de tree
5      endw
6      return maior
7  end
```

9.4.5 Menor elemento

O menor elemento da árvore, nó com o menor valor, será encontrado sempre na folha mais à esquerda da árvore [9]. Para encontrar o menor valor, basta procurar a partir da raiz sempre na subárvore da esquerda (algorimo 9.8).

9.4.6 Percorrendo uma árvore

Uma operação comum é percorrer uma árvore binária, o que consiste em visitar todos os nós desta árvore segundo algum critério. Esse percurso, também chamado de *travessia* da árvore, pode ser feito de três formas [15, 16]:

- pré-ordem ou profundidade - os filhos de um nó são processados após o nó.
- pós-ordem - os filhos são processados antes do nó.
- em-ordem ou simétrica - em que se processa o filho à esquerda, o nó, e finalmente o filho à direita.

Algoritmo 9.8: Menor elemento da árvore

```
   Input: Árvore tree
   Output: Menor elemento
1  begin
2      while tree ≠ NULO do
3          menor ⟵ tree (campo informação)
4          tree ⟵ esquerda de tree
5      endw
6      return menor
7  end
```

A operação percorrer pode ser descrita nos algoritmos 9.9, 9.10 e 9.11.

Algoritmo 9.9: Operação Percorre - Pré-ordem

```
1 begin
2     Visita a raiz
3     Percorre a subárvore esquerda em pré-ordem
4     Percorre a subárvore direita em pré-ordem
5 end
```

Algoritmo 9.10: Operação Percorre - Pós-ordem

```
1 begin
2     Percorre a subárvore esquerda em pós-ordem
3     Percorre a subárvore direita em pós-ordem
4     Visita a raiz
5 end
```

Algoritmo 9.11: Operação Percorre - Em-ordem

```
1 begin
2     Percorre a subárvore esquerda em pós-ordem.
3     Percorre a subárvore direita em pós-ordem
4     Visita a raiz
5 end
```

9.5 Representações de árvores em C

Árvores binárias podem ser representadas como um vetor de filhos (programa 9.1 ou de forma dinâmica (programa 9.2).

Programa 9.1: Representação com vetor de filhos

```
#define FILHOS 4

typedef struct NO {
    int info;
    struct NO *pai;
    struct NO *filhos[FILHOS];
} NO;
```

Programa 9.2: Representação dinâmica

```
typedef struct NO {
    int info;
    struct NO *pai;
    struct NO *filho; /* 1º filho */
    struct NO *irmao; /* próximo irmão*/
} NO;
```

A representação no formato de vetores da figura 9.2 pode ser verificada na figura 9.8. Na figura fica clara a representação de árvores, com vetores, ter limitações para a quantidade de filhos.

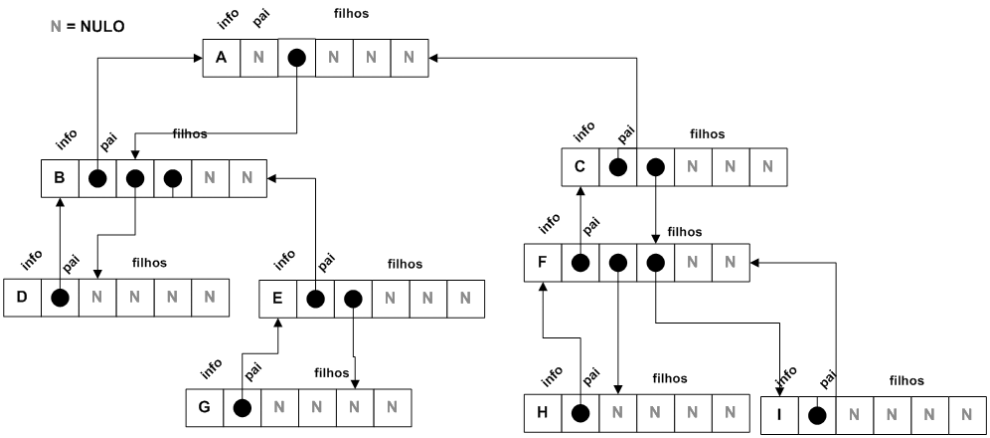

Figura 9.8: Representação com vetores

Se no programa 9.2 o campo *filho* for um ponteiro para subárvore esquerda e o campo *irmao* como o ponteiro para a raiz da subárvore direita, tem-se uma árvore binária como a representada na figura 9.9.

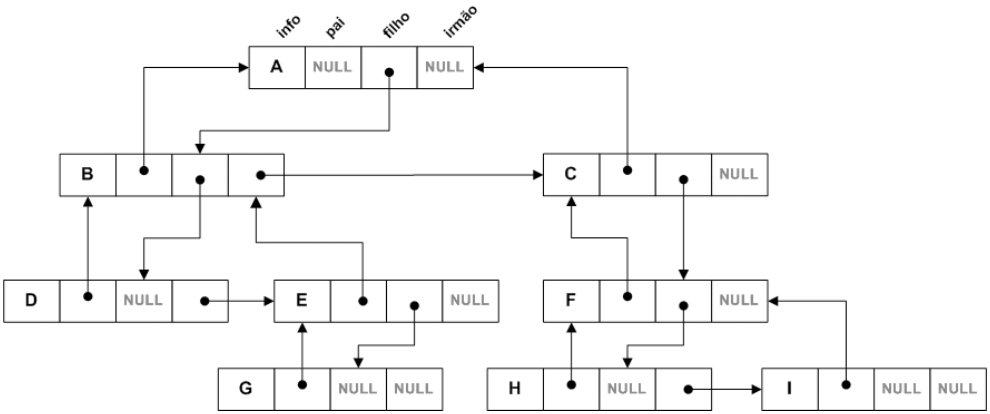

Figura 9.9: Representação dinâmica

Com o programa 9.3 tem-se a representação de uma árvore binária conforme a figura 9.3 vista anteriormente.

Programa 9.3: Representação dinâmica de uma árvore binária

```
typedef struct NO {
    int info;
    struct NO *esquerda;
struct NO *direita;
struct NO *pai;
} NO;
```

9.6 Implementação em C

No programa 9.4 podem ser vistas todas as operações e algoritmos conceituados anteriormente.

Programa 9.4: Implementação das operações

```
/* programa_arvore_01.c */

#include <stdio.h>
#include <stdlib.h>
```

```
typedef struct NO
{
int info;
struct NO *esquerda, *direita;
} node, *arvore;

arvore root = NULL;

arvore pesquisar(arvore, int);
int proxmaior(int);
void inserir(arvore *, int);
void imprimir(arvore, int);
void range(arvore, int, int);
int excluir_menor(void);
void excluir(arvore *, int);
void del(arvore *, arvore *);

void percorre_preordem(node *);
void percorre_posordem(node *);
void percorre_emordem(node *);
int maior(node * );
int menor(node * );

int main(void)
{
   int x, y, opcao;

   do
   {
      printf("\nEntre com a opcao");
      printf("\n ---1:inserir");
      printf("\n ---2:pesquisar");
      printf("\n ---3:excluir o menor");
      printf("\n ---4:excluir");
      printf("\n ---5:procurar o maior que");
      printf("\n ---6:imprimir a arvore");
      printf("\n ---7:mostrar nos do intervalo");
      printf("\n ---8:percorrer");
      printf("\n ---9:maior e menor");
      printf("\n ---10:sair do programa\n");
      printf("\n->");
      fflush(stdin);
      scanf("%d", &opcao);
      switch(opcao)
      {
         case 1:
         {
```

```
            printf("\n Informe o valor ->");
            scanf("%d", &x);
            inserir(&root, x);
            imprimir(root, 0);
            break;
        }
        case 2:
        {
            printf("\n Informe o valor ->");
            scanf("%d", &x);
            if(pesquisar(root, x) != NULL)
            {
                printf(" Encontrado\n");
            }
            else
            {
                printf(" Nao encontrado!\n");
            }
            break;
        }
        case 3:
        {
            printf(" Excluido o menor = %d\n", excluir_menor());
            imprimir(root, 0);
            break;
        }
        case 4:
        {
            printf("\n Informe o valor ->");
            scanf("%d", &x);
            excluir(&root, x);
            imprimir(root, 0);
            break;
        }
        case 5:
        {
            printf("\n Informe o valor ->");
            scanf("%d", &x);
            imprimir(root, 0);
            printf("\n --- proximo maior que = %d", proxmaior(x));
            break;
        }
        case 6:
        {
            imprimir(root, 0);
            break;
        }
        case 7:
        {
```

```
            printf("\n Informe [min, max]");
            scanf("%d %d",&x, &y);
            range(root, x, y);
            break;
        }
        case 8:
        {
            printf("\nPercorrendo em ordem ->");
            percorre_emordem(root);
            printf("\nPercorrendo em pre ordem ->");
            percorre_preordem(root);
            printf("\nPercorrendo em pos ordem ->");
            percorre_posordem(root);
            break;
        }
        case 9:
        {
            printf("\nMaior = %d", maior(root));
            printf("\nMenor = %d", menor(root));
            break;
        }
    }
} while(opcao!=10);
}

arvore pesquisar(arvore v, int chave)
{
    if( v == NULL)
    {
        return NULL;
    }
    if(v->info == chave)
    {
        return v;
    }
    else if(v->info < chave)
    {
        return pesquisar(v->direita, chave);
    }
    else
    {
        return pesquisar(v->esquerda, chave);
    }
}

/* maior no próximo elemento informado */
int proxmaior(int chave)
{
    arvore p=NULL, v;
```

```
    v = root;
    while( v != NULL && v->info != chave)
154 {
        if(v->info < chave)
        {
            v = v->direita;
        }
159     else
        {
            p = v;
            v = v->esquerda;
        }
164 }
    if(v == NULL)
    {
        printf("\n Elemento nao encontrado");
        return -1;
169 }
    if( v->direita != NULL )
    {
        v = v->direita;
        while(v->esquerda != NULL)
174     {
            v = v->esquerda;
        }
        return v->info;
    }
179 if(p != NULL)
    {
        return p->info;
    }
    else
184 {
        return -1;
    }
}

189 void inserir(arvore *p, int chave)
{
    if( *p == NULL )
    {
        *p = (arvore) malloc(sizeof(node));
194     (*p)->info = chave;
        (*p)->esquerda = NULL;
        (*p)->direita = NULL;
    }
    else if((*p)->info < chave)
199 {
```

```
        inserir(&((*p)->direita), chave);
    }
    else
    {
        inserir(&((*p)->esquerda), chave);
    }
    return;
}

void imprimir(arvore v, int nivel)
{
    int i;

    if( v != NULL )
    {
        imprimir(v->esquerda, nivel+1);
        for(i=0; i<nivel; i++)
        {
            printf("    ");
        }
        printf("%d\n", v->info);
        imprimir(v->direita, nivel+1);
    }
    return;
}


/* mostra os nós de um intervalo informado */
void range(arvore v, int x, int y)
{
    if( v == NULL )
    {
        return;
    }
    if(v->info >= x)
    {
        range(v->esquerda, x,y);
    }
    if(x <= v->info && v->info <= y)
    {
        printf("  %d", v->info);
    }
    if(v->info <= y)
    {
        range(v->direita, x,y);
    }
    return;
}
```

```
/* excluir o menor */
int excluir_menor(void)
{
    int menor;
    arvore p, v;
    if(root->esquerda == NULL)
    {
        menor = root->info;
        root = root->direita;
    }
    else
    {
        v = root;
        do
        {
            p = v;
            v = v->esquerda;
        } while(v->esquerda != NULL);
        menor = v->info;
        p->esquerda = v->direita;
    }
    return menor;
}

/* exclusão de elemento da árvore */
void excluir(arvore *p, int chave)
{
    arvore q;

    if(*p == NULL)
    {
        printf("\n Elemento nao existe!");
    }
    else if( chave < (*p)->info )
    {
        excluir(&((*p)->esquerda), chave);
    }
    else if( chave > (*p)->info )
    {
        excluir(&((*p)->direita), chave);
    }
    else
    {
        q = *p;
        if(q->direita == NULL)
        {
            *p = q->esquerda;
        }
```

```
        else if( q->esquerda == NULL)
        {
            *p = q->direita;
        }
        else
        {
            del(&q, &(q->esquerda));
        }
        free(q);
    }
    return;
}

/* procura sucessor para depois excluir */
void del(arvore *q, arvore *r)
{
    if((*r)->direita != NULL)
    {
        del(q, &((*r)->direita));
    }
    else
    {
        (*q)->info = (*r)->info;
        (*q) = *r;
        *r = (*r)->esquerda;
    }
    return;
}

/* percorrer uma árvore utilizando o algoritmo de pré-ordem */
void percorre_preordem(node * arvore)
{
    if( arvore == NULL )
    {
        return;
    }

    printf(" %d", arvore->info);
    percorre_preordem(arvore->esquerda);
    percorre_preordem(arvore->direita);

    return;
}

/* percorrer uma árvore utilizando o algoritmo de pós-ordem */
void percorre_posordem(node * arvore)
{
```

```
    if( arvore == NULL )
344 {
        return;
    }

    percorre_posordem(arvore->esquerda);
349 percorre_posordem(arvore->direita);
    printf(" %d", arvore->info);
    return;
}

354 /* percorrer uma árvore utilizando no modo em-ordem */
void percorre_emordem(node * arvore)
{
    if( arvore == NULL )
    {
359     return;
    }

    percorre_emordem(arvore->esquerda);
    printf(" %d", arvore->info);
364 percorre_emordem(arvore->direita);

    return;
}
369 /* pesquisa do maior elemento na árvore */
int maior(node * arvore )
{
    int maior;
    maior = arvore->info;
374

    while( arvore != NULL )
    {
        maior = arvore->info;
        arvore = arvore->direita;
379 }
    return maior;
}

/* pesquisa do menor elemento na árvore */
384 int menor(node * arvore)
{
    int menor;
    menor = arvore->info;

389 while( arvore != NULL )
```

```
    {
        menor = arvore->info;
        arvore = arvore->esquerda;
    }
394 return menor;
}
```

9.7 Exercício

1. Implemente os algoritmos iterativos, para manipulação de árvores, em C.

Referências Bibliográficas

[1] D. Baldwin and G. W. Scragg. *Algorithms and Data Structures: The Science of Computing*. Charles River Media, first edition, 2004.

[2] P. E. Black. *Dictionary of Algorithms and Data Structures*. U.S. National Institute of Standards and Technology, 2006. Ackermann's function in www.nist.gov/dads/HTML/ackermann.html.

[3] P. Deshpande and O. Kakde. *C and Data Structures*. Charles River Media, first edition, 2004.

[4] J. Keogh and K. Davidson. *Data Structures Demystified*. McGraw-Hill/Osborne, first edition, 2004.

[5] B. W. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[6] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 3. Addison-Wesley, third edition, 1997.

[7] A. Koening. *C Traps and Pitfalls*. Addison-Wesley, first edition, 1989.

[8] R. Lafore. *Teach Yourself Data Structures and Algorithms in 24 hours*. Samns Publishing, first edition, 1999.

[9] C. E. Leiserson, C. Stein, R. L. Rivest, and T. H. Cormen. *Introduction to Algorithms*. MIT Press and McGraw-Hill, second edition, 2001.

[10] F. Lorenzi, P. N. de Mattos, and T. P. de Carvalho. *Estrutura de dados*. Thomson, first edition, 2007.

[11] B. R. Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. John Wiley and Sons, first edition, 1998.

[12] D. D. Salvetti and L. M. Barbosa. *Algoritmos*. Pearson Makron Books, first edition, 1998.

[13] H. Schildt. *C The Complete Reference*. McGraw-Hill/Osborne, fourth edition, 2000.

[14] E. A. Schmitz and A. A. de Souza Teles. *Pascal e técnicas de programação*. LTC Editora, third edition, 1988.

[15] A. M. Tenenbaum, Y. Langsam, and M. J. Augenstein. *Estruturas de Dados Usando C*. Pearson, first edition, 1995.

[16] P. Veloso, C. dos Santos, P. Azeredo, and A. Furtado. *Estrutura de Dados*. Editora Campus, first edition, 1983. 16. Tiragem.

[17] Wikipedia. The free encyclopedia. http://en.wikipedia.org, 2007. Triangular Number.

[18] Wikipedia. The free encyclopedia. http://en.wikipedia.org, 2007. Tower of Hanoi.

[19] Wikipedia. The free encyclopedia. http://en.wikipedia.org, 2007. Ackermann function.

[20] Wikipedia. The free encyclopedia. http://en.wikipedia.org, 2007. Merge Sort.

[21] N. Wirth. Algorithms and Data Structures. Prentice Hall, first edition, 1985.

Índice Remissivo