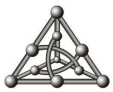
# Algoritmos e Programação de Computadores II

Fábio Henrique Viduani Martinez

Faculdade de Computação – UFMS



2011

# SUMÁRIO

AULA 1

# RECURSÃO

Recursão é um conceito fundamental em computação. Sua compreensão nos permite construir programas elegantes, curtos e poderosos. Muitas vezes, o equivalente não-recursivo é muito mais difícil de escrever, ler e compreender que a solução recursiva, devido em especial à estrutura recursiva intrínseca do problema. Por outro lado, uma solução recursiva tem como principal desvantagem maior consumo de memória, na maioria das vezes muito maior que uma função não-recursiva equivalente.

Na linguagem C, uma função recursiva é aquela que contém em seu corpo uma ou mais chamadas a si mesma. Naturalmente, uma função deve possuir pelo menos uma chamada proveniente de uma outra função externa. Uma função não-recursiva, em contrapartida, é aquela para qual todas as suas chamadas são externas. Nesta aula construiremos funções recursivas para soluções de problemas.

Este texto é baseado nas referências [2, 7].

## 1.1 Definição

De acordo com [2], em muitos problemas computacionais encontramos a seguinte propriedade: cada entrada do problema contém uma entrada menor do mesmo problema. Dessa forma, dizemos que esses problemas têm uma **estrutura recursiva**. Para resolver um problema como esse, usamos em geral a seguinte estratégia:

```
se a entrada do problema é pequena então
   resolva-a diretamente;
senão,
   reduza-a a uma entrada menor do mesmo problema,
   aplique este método à entrada menor
   e volte à entrada original.
```

A aplicação dessa estratégia produz um **algoritmo recursivo**, que implementado na linguagem C torna-se um **programa recursivo** ou um programa que contém uma ou mais **funções recursivas**.

Uma **função recursiva** é aquela que possui, em seu corpo, uma ou mais chamadas a si mesma. Uma chamada de uma função a si mesma é dita uma **chamada recursiva**. Como sabemos, uma função deve possuir ao menos uma chamada proveniente de uma outra função, externa a ela. Se a função só possui chamadas externas a si, então é chamada de função não-recursiva. No semestre anterior, construimos apenas funções não-recursivas.

Em geral, a toda função recursiva corresponde uma outra não-recursiva que executa exatamente a mesma computação. Como alguns problemas possuem estrutura recursiva natural, funções recursivas são facilmente construídas a partir de suas definições. Além disso, a demonstração da correção de um algoritmo recursivo é facilitada pela relação direta entre sua estrutura e a indução matemática. Outras vezes, no entanto, a implementação de um algoritmo recursivo demanda um gasto maior de memória, já que durante seu processo de execução muitas informações devem ser guardadas na sua pilha de execução.

## 1.2 Exemplos

Um exemplo clássico de uma função recursiva é aquela que computa o fatorial de um número inteiro $n \geqslant 0$. A idéia da solução através de uma função recursiva é baseada na fórmula mostrada a seguir:

$$n! = \begin{cases} 1\,, & \text{se } n \leqslant 1\,, \\ n \times (n-1)!\,, & \text{caso contrário}\,. \end{cases}$$

A função recursiva `fat`, que calcula o fatorial de um dado número inteiro não negativo $n$, é mostrada a seguir:

```
/* Recebe um número inteiro n >= 0 e devolve o fatorial de n */
int fat(int n)
{
   int result;

   if (n <= 1)
      result = 1;
   else
      result = n * fat(n-1);

   return result;
}
```

A sentença `return` pode aparecer em qualquer ponto do corpo de uma função e por isso podemos escrever a função `fat` como a seguir:

```
/* Recebe um número inteiro n >= 0 e devolve o fatorial de n */
int fat(int n)
{
   if (n <= 1)
      return 1;
   else
      return n * fat(n-1);
}
```

Geralmente, preferimos a primeira versão implementada acima, onde existe apenas uma sentença com a palavra reservada `return` posicionada no final do corpo da função `fat`. Essa maneira de escrever funções recursivas evita confusão e nos permite seguir o fluxo de execução

dessas funções mais naturalmente. No entanto, a segunda versão da função `fat` apresentada acima é equivalente à primeira e isso significa que também é válida. Além disso, essa segunda solução é mais compacta e usa menos memória, já que evita o uso de uma variável. Por tudo isso, essa segunda forma de escrever funções recursivas é muito usada por programadores(as) mais experientes.

A execução da função `fat` se dá da seguinte forma. Imagine que uma chamada `fat(3)` foi realizada. Então, temos ilustrativamente a seguinte situação:

`fat(3)`
- `fat(2)`
  - `fat(1)`
    - devolve 1
  - devolve $2 \times 1 = 2 \times$ `fat(1)`
- devolve $3 \times 2 = 3 \times$ `fat(2)`

Repare nas indentações que ilustram as chamadas recursivas à função.

Vamos ver um próximo exemplo. Considere o problema de determinar um valor máximo de um vetor $v$ com $n$ elementos. O tamanho de uma entrada do problema é $n \geqslant 1$. Se $n = 1$ então $v[0]$ é o único elemento do vetor e portanto $v[0]$ é máximo. Se $n > 1$ então o valor que procuramos é o maior dentre o máximo do vetor $v[0..n-2]$ e o valor armazenado em $v[n-1]$. Dessa forma, a entrada $v[0..n-1]$ do problema fica reduzida à entrada $v[0..n-2]$. A função `maximo` a seguir implementa essa idéia.

```
/* Recebe um número inteiro n > 0 e um vetor v de números in-
   teiros com n elementos e devolve um elemento máximo de v */
int maximo(int n, int v[MAX])
{
   int aux;

   if (n == 1)
      return v[0];
   else {
      aux = maximo(n-1, v);
      if (aux > v[n-1])
         return aux;
      else
         return v[n-1];
   }
}
```

Segundo P. Feofiloff [2], para verificar que uma função recursiva está correta, devemos seguir o seguinte roteiro, que significa mostrar por indução a correção de um algoritmo ou programa:

**Passo 1:** escreva *o que* a função deve fazer;

**Passo 2:** verifique se a função de fato faz o que deveria fazer quando a entrada é pequena;

**Passo 3:** imagine que a entrada é grande e suponha que a função fará a coisa certa para entradas menores; sob essa hipótese, verifique que a função faz o que dela se espera.

Isso posto, vamos provar então a seguinte propriedade sobre a função `maximo` descrita acima.

**Proposição 1.1.** *A função* `maximo` *encontra um maior elemento em um vetor $v$ com $n \geqslant 1$ números inteiros.*

*Demonstração.* Vamos provar a afirmação usando indução na quantidade $n$ de elementos do vetor $v$.

Se $n = 1$ o vetor $v$ contém exatamente um elemento, um número inteiro, armazenado em $v[0]$. A função `maximo` devolve, neste caso, o valor armazenado em $v[0]$, que é o maior elemento no vetor $v$.

Suponha que para qualquer valor inteiro positivo $m$ menor que $n$, a chamada externa à função `maximo(`$m$`, v)` devolva corretamente o valor de um maior elemento no vetor $v$ contendo $m$ elementos.

Suponha agora que temos um vetor $v$ contendo $n > 1$ números inteiros. Suponha que fazemos a chamada externa `maximo(`$n$`, v)`. Como $n > 1$, o programa executa a sentença descrita abaixo:

```
aux = maximo(n-1, v);
```

Então, por hipótese de indução, sabemos que a função `maximo` devolve um maior elemento no vetor $v$ contendo $n - 1$ elementos. Esse elemento, por conta da sentença acima, é armazenado então na variável `aux`. A estrutura condicional que se segue na função `maximo` compara o valor armazenado em `aux` com o valor armazenado em `v[n-1]`, o último elemento do vetor $v$. Um maior valor entre esses dois valores será então devolvido pela função. Dessa forma, a função `maximo` devolve corretamente o valor de um maior elemento em um vetor com $n$ números inteiros. □

## Exercícios

1.1 A $n$-ésima potência de um número $x$, denotada por $x^n$, pode ser computada recursivamente observando a seguinte a fórmula:

$$x^n = \begin{cases} 1\,, & \text{se } n = 0\,, \\ x \cdot x^{n-1}\,, & \text{se } n > 1\,. \end{cases}$$

Considere neste exercício que $x$ e $n$ são números inteiros.

(a) Escreva uma função não-recursiva com a seguinte interface:

```
int pot(int x, int n)
```

que receba dois números inteiros $x$ e $n$ e calcule e devolva $x^n$.

(b) Escreva uma função recursiva com a seguinte interface:

```
int potR(int x, int n)
```

que receba dois números inteiros $x$ e $n$ e calcule e devolva $x^n$.

(c) Escreva um programa que receba dois números inteiros $x$ e $n$, com $n \geqslant 0$, e devolva $x^n$. Use as funções em (a) e (b) para mostrar os dois resultados.

Programa 1.1: Solução do exercício 1.1.

```
#include <stdio.h>

/* Recebe um dois números inteiros x e n e devolve x a n-ésima potência */
int pot(int x, int n)
{
   int i, result;

   result = 1;
   for (i = 1; i <= n; i++)
      result = result * x;
   return result;
}

/* Recebe um dois números inteiros x e n e devolve x a n-ésima potência */
int potR(int x, int n)
{
   if (n == 0)
      return 1;
   else
      return x * potR(x, n-1);
}

/* Recebe dois números inteiros x e n e imprime x a n-ésima potên-
   cia chamando duas funções: uma não-recursiva e uma recursiva */
int main(void)
{
   int x, n;

   scanf("%d%d", &x, &n);
   printf("Não-resursiva: %d^%d = %d\n", x, n, pot(x, n));
   printf("Resursiva    : %d^%d = %d\n", x, n, potR(x, n));

   return 0;
}
```

1.2 O que faz a função abaixo?

```
void imprime_alguma_coisa(int n)
{
   if (n != 0) {
      imprime_alguma_coisa(n / 2);
      printf("%c", '0' + n % 2);
   }
}
```

Escreva um programa para testar a função `imprime_alguma_coisa`.

1.3 (a) Escreva uma função recursiva que receba dois números inteiros positivos e devolva o máximo divisor comum entre eles usando o algoritmo de Euclides.

(b) Escreva um programa que receba dois números inteiros e calcule o máximo divisor comum entre eles. Use a função do item (a).

1.4 (a) Escreva uma função recursiva com a seguinte interface:

```
float soma(int n, float v[MAX])
```

que receba um número inteiro $n > 0$ e um vetor $v$ de números com ponto flutuante com $n$ elementos, e calcule e devolva a soma desses números.

(b) Usando a função do item anterior, escreva um programa que receba um número inteiro $n$, com $n \geqslant 1$, e mais $n$ números reais e calcule a soma desses números.

1.5 (a) Escreva uma função recursiva com a seguinte interface:

```
int soma_digitos(int n)
```

que receba um número inteiro positivo $n$ e devolva a soma de seus dígitos.

(b) Escreva um programa que receba um número inteiro $n$ e imprima a soma de seus dígitos. Use a função do item (a).

1.6 A **seqüência de Fibonacci** é uma seqüência de números inteiros positivos dada pela seguinte fórmula:

$$\begin{cases} F_1 &= 1 \\ F_2 &= 1 \\ F_i &= F_{i-1} + F_{i-2}\,, \quad \text{para } i \geqslant 3. \end{cases}$$

(a) Escreva uma função recursiva com a seguinte interface:

```
int Fib(int i)
```

que receba um número inteiro positivo $i$ e devolva o $i$-ésimo termo da seqüência de Fibonacci, isto é, $F_i$.

(b) Escreva um programa que receba um número inteiro $i \geqslant 1$ e imprima o termo $F_i$ da seqüência de Fibonacci. Use a função do item (a).

1.7 O **piso** de um número inteiro positivo $x$ é o único inteiro $i$ tal que $i \leqslant x < i + 1$. O piso de $x$ é denotado por $\lfloor x \rfloor$.

Segue uma amostra de valores da função $\lfloor \log_2 n \rfloor$:

| $n$ | 15 | 16 | 31 | 32 | 63 | 64 | 127 | 128 | 255 | 256 | 511 | 512 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\lfloor \log_2 n \rfloor$ | 3 | 4 | 4 | 5 | 5 | 6 | 6 | 7 | 7 | 8 | 8 | 9 |

(a) Escreva uma função recursiva com a seguinte interface:

```
int piso_log2(int n)
```

que receba um número inteiro positivo $n$ e devolva $\lfloor \log_2 n \rfloor$.

(b) Escreva um programa que receba um número inteiro $n \geqslant 1$ e imprima $\lfloor \log_2 n \rfloor$. Use a função do item (a).

1.8 Considere o seguinte processo para gerar uma seqüência de números. Comece com um inteiro $n$. Se $n$ é par, divida por 2. Se $n$ é ímpar, multiplique por 3 e some 1. Repita esse processo com o novo valor de $n$, terminando quando $n = 1$. Por exemplo, a seqüência de números a seguir é gerada para $n = 22$:

22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

É conjecturado que esse processo termina com $n = 1$ para todo inteiro $n > 0$. Para uma entrada $n$, o **comprimento do ciclo de** $n$ é o número de elementos gerados na seqüência. No exemplo acima, o comprimento do ciclo de 22 é 16.

(a) Escreva uma função não-recursiva com a seguinte interface:

```
int ciclo(int n)
```

que receba um número inteiro positivo $n$, mostre a seqüência gerada pelo processo descrito acima na saída e devolva o comprimento do ciclo de $n$.

(b) Escreva uma versão recursiva da função do item (a) com a seguinte interface:

```
int cicloR(int n)
```

que receba um número inteiro positivo $n$, mostre a seqüência gerada pelo processo descrito acima na saída e devolva o comprimento do ciclo de $n$.

(c) Escreva um programa que receba um número inteiro $n \geqslant 1$ e determine a seqüência gerada por esse processo e também o comprimento do ciclo de $n$. Use as funções em (a) e (b) para testar.

1.9 Podemos calcular a potência $x^n$ de uma maneira mais eficiente. Observe primeiro que se $n$ é uma potência de 2 então $x^n$ pode ser computada usando seqüências de quadrados. Por exemplo, $x^4$ é o quadrado de $x^2$ e assim $x^4$ pode ser computado usando somente duas multiplicações ao invés de três. Esta técnica pode ser usada mesmo quando $n$ não é uma potência de 2, usando a seguinte fórmula:

$$x^n = \begin{cases} 1\,, & \text{se } n = 0, \\ (x^{n/2})^2\,, & \text{se } n \text{ é par}, \\ x \cdot x^{n-1}\,, & \text{se } n \text{ é ímpar}. \end{cases} \tag{1.1}$$

(a) Escreva uma função com interface

```
int potencia(int x, int n)
```

que receba dois números inteiros $x$ e $n$ e calcule e devolva $x^n$ usando a fórmula (1.1).

(b) Escreva um programa que receba dois números inteiros $a$ e $b$ e imprima o valor de $a^b$.

AULA 2

# EFICIÊNCIA DE ALGORITMOS E PROGRAMAS

Medir a eficiência de um algoritmo ou programa significa tentar predizer os recursos necessários para seu funcionamento. O recurso que temos mais interesse neste momento é o tempo de execução embora a memória, a comunicação e o uso de portas lógicas também podem ser de interesse. Na análise de algoritmos e/ou programas alternativos para solução de um mesmo problema, aqueles mais eficientes de acordo com algum desses critérios são em geral escolhidos como melhores. Nesta aula faremos uma discussão inicial sobre eficiência de algoritmos e programas tendo como base as referências [1, 2, 8].

## 2.1 Algoritmos e programas

Nesta aula, usaremos os termos algoritmo e programa como sinônimos. Dessa forma, podemos dizer que um **algoritmo** ou **programa**, como já sabemos, é uma seqüência bem definida de passos descritos em uma linguagem de programação específica que transforma um conjunto de valores, chamado de **entrada**, e produz um conjunto de valores, chamado de **saída**. Assim, um algoritmo ou programa é também uma ferramenta para solucionar um **problema computacional** bem definido.

Suponha que temos um problema computacional bem definido, conhecido como o problema da busca, que certamente já nos deparamos antes. A descrição mais formal do problema é dada a seguir:

> Dado um número inteiro $n$, com $1 \leqslant n \leqslant 100$, um conjunto $C$ de $n$ números inteiros e um número inteiro $x$, verificar se $x$ encontra-se no conjunto $C$.

O problema da busca é um problema computacional básico que surge em diversas aplicações práticas. A busca é uma operação básica em computação e, por isso, vários bons programas que a realizam foram desenvolvidos. A escolha do melhor programa para uma dada aplicação depende de alguns fatores como a quantidade de elementos no conjunto $C$ e da complexidade da estrutura em que $C$ está armazenado.

Se para qualquer entrada um programa pára com a resposta correta, então dizemos que o mesmo está **correto**. Assim, um programa correto **soluciona** o problema computacional associado. Um programa incorreto pode sequer parar, para alguma entrada, ou pode parar mas com uma resposta indesejada.

O programa 2.1 implementa uma busca de um número inteiro $x$ em um conjunto de $n$ números inteiros $C$ armazenado na memória como um vetor. A busca se dá seqüencialmente no vetor $C$, da esquerda para direita, até que um elemento com índice $i$ no vetor $C$ contenha o valor $x$, quando o programa responde que o elemento foi encontrado, mostrando sua posição. Caso contrário, o vetor $C$ é todo percorrido e o programa informa que o elemento $x$ não se encontra no vetor $C$.

Programa 2.1: Busca de $x$ em $C$.

```
#include <stdio.h>

#define MAX 100

/* Recebe um número inteiro n, com 1 <= n <= 100, um conjunto C de n
   números inteiros e um número inteiro x, e verifica se x está em C */
int main(void)
{
   int n, i, C[MAX], x;

   printf("Informe n: ");
   scanf("%d", &n);
   for (i = 0; i < n; i++) {
      printf("Informe um elemento: ");
      scanf("%d", &C[i]);
   }
   printf("Informe x: ");
   scanf("%d", &x);

   for (i = 0; i < n && C[i] != x; i++)
      ;

   if (i < n)
      printf("%d está na posição %d de C\n", x, i);
   else
      printf("%d não pertence ao conjunto C\n", x);

   return 0;
}
```

## 2.2 Análise de algoritmos e programas

Antes de analisar um algoritmo ou programa, devemos conhecer o modelo da tecnologia de computação usada na máquina em que implementamos o programa, para estabelecer os custos associados aos recursos que o programa usa. Na análise de algoritmos e programas, consideramos regularmente um modelo de computação genérico chamado de **máquina de acesso aleatório** (do inglês *random acess machine – RAM*) com um processador. Nesse modelo, as instruções são executadas uma após outra, sem concorrência. Modelos de computação paralela e distribuída, que usam concorrência de instruções, são modelos investigativos que vêm se tornando realidade recentemente. No entanto, nosso modelo para análise de algoritmos e programas não leva em conta essas premissas.

A análise de um programa pode ser uma tarefa desafiadora envolvendo diversas ferramentas matemáticas tais como combinatória discreta, teoria das probabilidades, álgebra e etc. Como o comportamento de um programa pode ser diferente para cada entrada possível, precisamos de uma maneira que nos possibilite resumir esse comportamento em fórmulas matemáticas simples e de fácil compreensão.

Mesmo com a convenção de um modelo fixo de computação para analisar um programa, ainda precisamos fazer muitas escolhas para decidir como expressar nossa análise. Um dos principais objetivos é encontrar uma forma de expressão abstrata que é simples de escrever e manipular, que mostra as características mais importantes das necessidades do programa e exclui os detalhes mais tediosos.

Não é difícil notar que a análise do programa 2.1 depende do número de elementos fornecidos na entrada. Isso significa que procurar um elemento $x$ em um conjunto $C$ com milhares de elementos certamente gasta mais tempo que procurá-lo em um conjunto $C$ com apenas três elementos. Além disso, é importante também notar que o programa 2.1 gasta diferentes quantidades de tempo para buscar um elemento em conjuntos de mesmo tamanho, dependendo de como os elementos nesses conjuntos estão dispostos, isto é, da ordem como são fornecidos na entrada. Como é fácil perceber também, em geral o tempo gasto por um programa cresce com o tamanho da entrada e assim é comum descrever o tempo de execução de um programa como uma função do tamanho de sua entrada.

Por **tamanho da entrada** queremos dizer, quase sempre, o número de itens na entrada. Por exemplo, o vetor $C$ com $n$ números inteiros onde a busca de um elemento será realizada tem $n$ itens ou elementos. Em outros casos, como na multiplicação de dois números inteiros, a melhor medida para o tamanho da entrada é o número de bits necessários para representar essa entrada na base binária. O **tempo de execução** de um programa sobre uma entrada particular é o número de operações primitivas, ou passos, executados por ele. Quanto mais independente da máquina é a definição de um passo, mais conveniente para análise de tempo dos algoritmos e programas.

Considere então que uma certa quantidade constante de tempo é necessária para executar cada linha de um programa. Uma linha pode gastar uma quantidade de tempo diferente de outra linha, mas consideramos que cada execução da $i$-ésima linha gasta tempo $c_i$, onde $c_i$ é uma constante positiva.

Iniciamos a análise do programa 2.1 destacando que o aspecto mais importante para sua análise é o tempo gasto com a busca do elemento $x$ no vetor $C$ contendo $n$ números inteiros, descrita entre a entrada de dados (leitura) e a saída (impressão dos resultados). As linhas restantes contêm diretivas de pré-processador, cabeçalho da função `main`, a própria entrada de dados, a própria saída e também a sentença `return` que finaliza a função `main`. É fato que a entrada de dados e a saída são, em geral, inerentes aos problemas computacionais e, além disso, sem elas não há sentido em se falar de processamento. Isso quer dizer que, como a entrada e a saída são inerentes ao programa, o que de fato damos mais importância na análise de um programa é no seu tempo gasto no processamento, isto é, na transformação dos dados de entrada nos dados de saída. Isto posto, vamos verificar a seguir o custo de cada linha no programa 2.1 e também o número de vezes que cada linha é executada. A tabela a seguir ilustra esses elementos.

| | *Custo* | *Vezes* |
|---|---|---|
| `#include <stdio.h>` | $c_1$ | 1 |
| `#define MAX 100` | $c_2$ | 1 |
| `int main(void)` | $c_3$ | 1 |
| `{` | 0 | 1 |
| `  int n, i, C[MAX], x;` | $c_4$ | 1 |
| `  printf("Informe n:  ");` | $c_5$ | 1 |
| `  scanf("%d", &n);` | $c_6$ | 1 |
| `  for (i = 0; i < n; i++) {` | $c_7$ | $n+1$ |
| `    printf("Informe um elemento:  ");` | $c_8$ | $n$ |
| `    scanf("%d", &C[i]);` | $c_9$ | $n$ |
| `  }` | 0 | $n$ |
| `  printf("Informe x:  ");` | $c_{10}$ | 1 |
| `  scanf("%d", &x);` | $c_{11}$ | 1 |
| `  for (i = 0; i < n && C[i] != x; i++)` | $c_{12}$ | $t_i$ |
| `    ;` | 0 | $t_i - 1$ |
| `  if (i < n)` | $c_{13}$ | 1 |
| `    printf("%d está na posição %d de C\n", x, i);` | $c_{14}$ | 1 |
| `  else` | $c_{15}$ | 1 |
| `    printf("%d não pertence ao conjunto C\n", x);` | $c_{16}$ | 1 |
| `  return 0;` | $c_{17}$ | 1 |
| `}` | 0 | 1 |

O tempo de execução do programa 2.1 é dado pela soma dos tempos para cada sentença executada. Uma sentença que gasta $c_i$ passos e é executada $n$ vezes contribui com $c_i n$ no tempo de execução do programa. Para computar $T(n)$, o tempo de execução do programa 2.1, devemos somar os produtos das colunas ***Custo*** e ***Vezes***, obtendo:

$$T(n) = c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7(n+1) + c_8 n + c_9 n + \\ c_{10} + c_{11} + c_{12} t_i + c_{13} + c_{14} + c_{15} + c_{16} + c_{17} \, .$$

Observe que, mesmo para entradas de um mesmo tamanho, o tempo de execução de um programa pode depender de qual entrada desse tamanho é fornecida. Por exemplo, no programa 2.1, o melhor caso ocorre se o elemento $x$ encontra-se na primeira posição do vetor $C$. Assim, $t_i = 1$ e o tempo de execução do melhor caso é dado por:

$$\begin{aligned} T(n) &= c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7(n+1) + c_8 n + c_9 n + \\ &\quad c_{10} + c_{11} + c_{12} + c_{13} + c_{14} + c_{15} + c_{16} + c_{17} \\ &= (c_7 + c_8 + c_9)n + \\ &\quad (c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_{10} + c_{11} + c_{12} + c_{13} + c_{14} + c_{15} + c_{16} + c_{17}) \, . \end{aligned}$$

Esse tempo de execução pode ser expresso como uma função linear $an + b$ para constantes $a$ e $b$, que dependem dos custos $c_i$ das sentenças do programa. Assim, o tempo de execução é dado por uma função linear em $n$. No entanto, observe que as constantes $c_7, c_8$ e $c_9$ que multiplicam $n$ na fórmula de $T(n)$ são aquelas que tratam somente da entrada de dados. Evidentemente que para armazenar $n$ números inteiros em um vetor devemos gastar tempo $an$, para alguma constante positiva $a$. Dessa forma, se consideramos apenas a constante $c_{12}$ na fórmula, que trata propriamente da busca, o tempo de execução de melhor caso, quando $t_i = 1$, é

dado por:

$$T(n) = c_{12}t_i = c_{12} .$$

Por outro lado, se o elemento $x$ não se encontra no conjunto $C$, temos então o pior caso do programa. Além de comparar $i$ com $n$, comparamos também o elemento $x$ com o elemento $C[i]$ para cada $i$, $0 \leqslant i \leqslant n-1$. Uma última comparação ainda é realizada quando $i$ atinge o valor $n$. Assim, $t_i = n+1$ e o tempo de execução de pior caso é dado por:

$$T(n) = c_{12}t_i = c_{12}(n+1) = c_{12}n + c_{12} .$$

Esse tempo de execução de pior caso pode ser expresso como uma função linear $an+b$ para constantes $a$ e $b$ que dependem somente da constante $c_{12}$, responsável pelo trecho do programa que realiza o processamento.

Na análise do programa 2.1, estabelecemos os tempos de execução do melhor caso, quando encontramos o elemento procurado logo na primeira posição do vetor que representa o conjunto, e do pior caso, quando não encontramos o elemento no vetor. No entanto, estamos em geral interessados no **tempo de execução de pior caso** de um programa, isto é, o maior tempo de execução para qualquer entrada de tamanho $n$.

Como o tempo de execução de pior caso de um programa é um limitante superior para seu tempo de execução para qualquer entrada, temos então uma garantia que o programa nunca vai gastar mais tempo que esse estabelecido. Além disso, o pior caso ocorre muito freqüentemente nos programas em geral, como no caso do problema da busca.

### 2.2.1 Ordem de crescimento de funções matemáticas

Acabamos de usar algumas convenções que simplificam a análise do programa 2.1. A primeira abstração que fizemos foi ignorar o custo real de uma sentença do programa, usando as constantes $c_i$ para representar esses custos. Depois, observamos que mesmo essas constantes nos dão mais detalhes do que realmente precisamos: o tempo de execução de pior caso do programa 2.1 é $an+b$, para constantes $a$ e $b$ que dependem dos custos $c_i$ das sentenças. Dessa forma, ignoramos não apenas o custo real das sentenças mas também os custos abstratos $c_i$.

Na direção de realizar mais uma simplificação, estamos interessados na **taxa de crescimento**, ou **ordem de crescimento**, da função que descreve o tempo de execução de um algoritmo ou programa. Portanto, consideramos apenas o 'maior' termo da fórmula, como por exemplo $an$, já que os termos menores são relativamente insignificantes quando $n$ é um número grande. Também ignoramos o coeficiente constante do maior termo, já que fatores constantes são menos significativos que a taxa de crescimento no cálculo da eficiência computacional para entradas grandes. Dessa forma, dizemos que o programa 2.1, por exemplo, tem tempo de execução de pior caso $O(n)$.

Usualmente, consideramos um programa mais eficiente que outro se seu tempo de execução de pior caso tem ordem de crescimento menor. Essa avaliação pode ser errônea para pequenas entradas mas, para entradas suficientemente grandes, um programa que tem tempo de execução de pior caso $O(n)$ executará mais rapidamente no pior caso que um programa que tem tempo de execução de pior caso $O(n^2)$.

Quando olhamos para entradas cujos tamanhos são grandes o suficiente para fazer com que somente a taxa de crescimento da função que descreve o tempo de execução de um programa

seja relevante, estamos estudando na verdade a **eficiência assintótica** de um algoritmo ou programa. Isto é, concentramo-nos em saber como o tempo de execução de um programa cresce com o tamanho da entrada *no limite*, quando o tamanho da entrada cresce ilimitadamente. Usualmente, um programa que é assintoticamente mais eficiente será a melhor escolha para todas as entradas, excluindo talvez algumas entradas pequenas.

As notações que usamos para descrever o tempo de execução assintótico de um programa são definidas em termos de funções matemáticas cujos domínios são o conjunto dos números naturais $\mathbb{N} = \{0, 1, 2, \ldots\}$. Essas notações são convenientes para descrever o tempo de execução de pior caso $T(n)$ que é usualmente definido sobre entradas de tamanhos inteiros.

No início desta seção estabelecemos que o tempo de execução de pior caso do programa 2.1 é $T(n) = O(n)$. Vamos definir formalmente o que essa notação significa. Para uma dada função $g(n)$, denotamos por $O(g(n))$ o *conjunto de funções*

$$O(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leqslant f(n) \leqslant cg(n) \text{ para todo } n \geqslant n_0\}.$$

A função $f(n)$ pertence ao conjunto $O(g(n))$ se existe uma constante positiva $c$ tal que $f(n)$ "não seja maior que" $cg(n)$, para $n$ suficientemente grande. Dessa forma, usamos a notação $O$ para fornecer um limitante assintótico superior sobre uma função, dentro de um fator constante. Apesar de $O(g(n))$ ser um conjunto, escrevemos "$f(n) = O(g(n))$" para indicar que $f(n)$ é um elemento de $O(g(n))$, isto é, que $f(n) \in O(g(n))$.

A figura 2.1 mostra a intuição por trás da notação $O$. Para todos os valores de $n$ à direita de $n_0$, o valor da função $f(n)$ está sobre ou abaixo do valor da função $g(n)$.

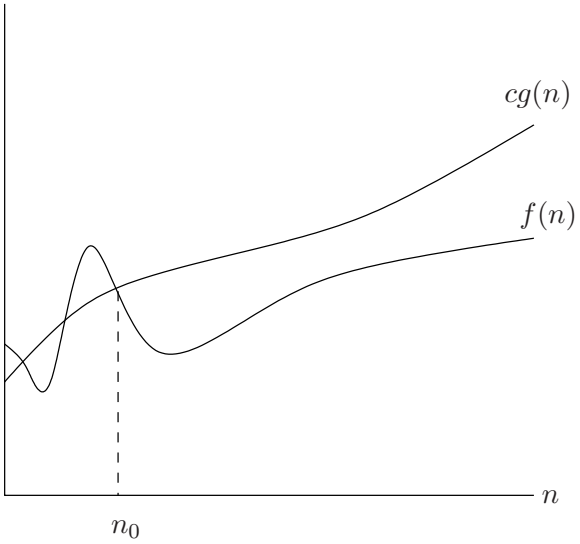



Figura 2.1: $f(n) = O(g(n))$.

Dessa forma, podemos dizer, por exemplo, que $4n + 1 = O(n)$. Isso porque existem constantes positivas $c$ e $n_0$ tais que

$$4n + 1 \leqslant cn$$

para todo $n \geqslant n_0$. Tomando, por exemplo, $c = 5$ temos

$$\begin{aligned} 4n + 1 &\leqslant 5n \\ 1 &\leqslant n\,, \end{aligned}$$

ou seja, para $n_0 = 1$, a desigualdade $4n + 1 \leqslant 5n$ é satisfeita para todo $n \geqslant n_0$. Certamente, existem outras escolhas para as constantes $c$ e $n_0$, mas o mais importante é que existe alguma escolha. Observe que as constantes dependem da função $4n + 1$. Uma função diferente que pertence a $O(n)$ provavelmente necessita de outras constantes.

A definição de $O(g(n))$ requer que toda função pertencente a $O(g(n))$ seja assintoticamente não-negativa, isto é, que $f(n)$ seja não-negativa sempre que $n$ seja suficientemente grande. Conseqüentemente, a própria função $g(n)$ deve ser assintoticamente não-negativa, caso contrário o conjunto $O(g(n))$ é vazio. Portanto, vamos considerar que toda função usada na notação $O$ é assintoticamente não-negativa.

A ordem de crescimento do tempo de execução de um programa ou algoritmo pode ser denotada através de outras notações assintóticas, tais como as notações $\Theta$, $\Omega$, $o$ e $\omega$. Essas notações são específicas para análise do tempo de execução de um programa através de outros pontos de vista. Nesta aula, ficaremos restritos apenas à notação $O$. Em uma disciplina mais avançada, como Análise de Algoritmos, esse estudo se aprofunda e atenção especial é dada a este assunto.

## 2.3 Análise da ordenação por trocas sucessivas

Vamos fazer a análise agora não de um programa, mas de uma função que já conhecemos bem e que soluciona o problema da ordenação. A função `trocas_sucessivas` faz a ordenação de um vetor $v$ de $n$ números inteiros fornecidos como parâmetros usando o método das trocas sucessivas ou o método da bolha.

```
/* Recebe um número inteiro n > 0 e um vetor v com n números intei-
   ros e rearranja o vetor v em ordem crescente de seus elementos */
void trocas_sucessivas(int n, int v[MAX])
{
   int i, j, aux;

   for (i = n-1; i > 0; i--)
      for (j = 0; j < i; j++)
         if (v[j] > v[j+1]) {
            aux = v[j];
            v[j] = v[j+1];
            v[j+1] = aux;
         }
}
```

Conforme já fizemos antes, a análise linha a linha da função `trocas_sucessivas` é descrita abaixo.

| | *Custo* | *Vezes* |
|---|---|---|
| `void trocas_sucessivas(int` $n$`, int` $v$`[MAX]` | $c_1$ | $1$ |
| `{` | $0$ | $1$ |
| `  int i, j, aux;` | $c_2$ | $1$ |
| `  for (i =` $n$`-1; i > 0; i--)` | $c_3$ | $n$ |
| `    for (j = 0; j < i; j++)` | $c_4$ | $\sum_{i=1}^{n-1}(i+1)$ |
| `      if (`$v$`[j] >` $v$`[j+1]) {` | $c_5$ | $\sum_{i=1}^{n-1} i$ |
| `        aux =` $v$`[j];` | $c_6$ | $\sum_{i=1}^{n-1} t_i$ |
| `        `$v$`[j] =` $v$`[j+1];` | $c_7$ | $\sum_{i=1}^{n-1} t_i$ |
| `        `$v$`[j+1] = aux;` | $c_8$ | $\sum_{i=1}^{n-1} t_i$ |
| `      }` | $0$ | $\sum_{i=1}^{n-1} i$ |
| `}` | $0$ | $1$ |

Podemos ir um passo adiante na simplificação da análise de um algoritmo ou programa se consideramos que cada linha tem custo de processamento 1. Na prática, isso não é bem verdade já que há sentenças mais custosas que outras. Por exemplo, o custo para executar uma sentença que contém a multiplicação de dois números de ponto flutuante de precisão dupla é maior que o custo de comparar dois números inteiros. No entanto, ainda assim vamos considerar que o custo para processar qualquer linha de um programa é constante e, mais ainda, que tem custo 1. Dessa forma, a coluna com rótulo ***Vezes*** nas análises acima representam então o custo de execução de cada linha do programa.

O melhor caso para a função `trocas_sucessivas` ocorre quando a seqüência de entrada com $n$ números inteiros é fornecida em ordem crescente. Observe que, nesse caso, uma troca nunca ocorrerá. Ou seja, $t_i = 0$ para todo $i$. Então, o tempo de execução no melhor caso é dado pela seguinte expressão:

$$\begin{aligned}
T(n) &= n + \sum_{i=1}^{n-1}(i+1) + \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} t_i + \sum_{i=1}^{n-1} t_i + \sum_{i=1}^{n-1} t_i \\
&= n + 2\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 + 3\sum_{i=1}^{n-1} t_i \\
&= n + 2\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 + 3\sum_{i=1}^{n-1} 0 \\
&= n + 2\,\frac{n(n-1)}{2} + n - 1 \\
&= n^2 + n - 1\,.
\end{aligned}$$

Então, sabemos que o tempo de execução da função `trocas_sucessivas` é dado pela expressão $T(n) = n^2 + n - 1$. Para mostrar que $T(n) = O(n^2)$ devemos encontrar duas constantes positivas $c$ e $n_0$ e mostrar que

$$n^2 + n - 1 \leqslant cn^2\,,$$

para todo $n \geqslant n_0$. Então, escolhendo $c = 2$ temos:

$$\begin{aligned}
n^2 + n - 1 &\leqslant 2n^2 \\
n - 1 &\leqslant n^2
\end{aligned}$$

ou seja,

$$n^2 - n + 1 \geqslant 0\,.$$

Observe então que a inequação $n^2 - n + 1 \geqslant 0$ é sempre válida para todo $n \geqslant 1$. Assim, escolhendo $c = 2$ e $n_0 = 1$, temos que

$$n^2 + n - 1 \leqslant cn^2$$

para todo $n \geqslant n_0$, onde $c = 2$ e $n_0 = 1$. Portanto, $T(n) = O(n^2)$, ou seja, o tempo de execução do melhor caso da função `trocas_sucessivas` é quadrático no tamanho da entrada.

Para definir a entrada que determina o pior caso para a função `trocas_sucessivas` devemos notar que quando o vetor contém os $n$ elementos em ordem decrescente, o maior número possível de trocas é realizado. Assim, $t_i = i$ para todo $i$ e o tempo de execução de pior caso é dado pela seguinte expressão:

$$\begin{aligned}
T(n) &= n + \sum_{i=1}^{n-1}(i+1) + \sum_{i=1}^{n-1} i + 3\sum_{i=1}^{n-1} t_i \\
&= n + \sum_{i=1}^{n-1}(i+1) + \sum_{i=1}^{n-1} i + 3\sum_{i=1}^{n-1} i \\
&= n + 5\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 \\
&= n + 5\,\frac{n(n-1)}{2} + n - 1 \\
&= \frac{5}{2}\,n^2 - \frac{5}{2}\,n + 2n - 1 \\
&= \frac{5}{2}\,n^2 - \frac{1}{2}\,n - 1\,.
\end{aligned}$$

Agora, para mostrar que o tempo de execução de pior caso $T(n) = O(n^2)$, escolhemos $c = 5/2$ e temos então que

$$\begin{aligned}
\frac{5}{2}\,n^2 - \frac{1}{2}\,n - 1 &\leqslant \frac{5}{2}\,n^2 \\
-\frac{1}{2}\,n - 1 &\leqslant 0
\end{aligned}$$

ou seja,

$$\frac{1}{2}\,n + 1 \geqslant 0$$

e, assim, a inequação $(1/2)n + 1 \geqslant 0$ para todo $n \geqslant 1$. Logo, escolhendo $c = 5/2$ e $n_0 = 1$ temos que

$$\frac{5}{2}\,n^2 - \frac{1}{2}\,n - 1 \leqslant cn^2$$

para todo $n \geqslant n_0$, onde $c = 5/2$ e $n_0 = 1$. Assim, $T(n) = O(n^2)$, ou seja, o tempo de execução do pior caso da função `trocas_sucessivas` é quadrático no tamanho da entrada. Note também que ambos os tempos de execução de melhor e de pior caso da função `trocas_sucessivas` têm o mesmo desempenho assintótico.

## 2.4 Moral da história

Esta seção é basicamente uma cópia traduzida da seção correspondente do livro de Cormen et. al [1] e descreve um breve resumo – a moral da história – sobre eficiência de algortimos e programas.

Um bom algoritmo ou programa é como uma faca afiada: faz o que é suposto fazer com a menor quantidade de esforço aplicada. Usar um programa errado para resolver um problema é como tentar cortar um bife com uma chave de fenda: podemos eventualmente obter um resultado digerível, mas gastaremos muito mais energia que o necessário e o resultado não deverá ser muito agradável esteticamente.

Programas alternativos projetados para resolver um mesmo problema em geral diferem dramaticamente em eficiência. Essas diferenças podem ser muito mais significativas que a diferença de tempos de execução em um supercomputador e em um computador pessoal. Como um exemplo, suponha que temos um supercomputador executando o programa que implementa o método da bolha de ordenação, com tempo de execução de pior caso $O(n^2)$, contra um pequeno computador pessoal executando o método da intercalação. Esse último método de ordenação tem tempo de execução de pior caso $O(n \log n)$ para qualquer entrada de tamanho $n$ e será visto na aula 6. Suponha que cada um desses programas deve ordenar um vetor de um milhão de números. Suponha também que o supercomputador é capaz de executar 100 milhões de operações por segundo enquanto que o computador pessoal é capaz de executar apenas um milhão de operações por segundo. Para tornar a diferença ainda mais dramática, suponha que o mais habilidoso dos programadores do mundo codificou o método da bolha na linguagem de máquina do supercomputador e o programa usa $2n^2$ operações para ordenar $n$ números inteiros. Por outro lado, o método da intercalação foi programado por um programador mediano usando uma linguagem de programação de alto nível com um compilador ineficiente e o código resultante gasta $50n \log n$ operações para ordenar os $n$ números. Assim, para ordenar um milhão de números o supercomputador gasta

$$\frac{2 \cdot (10^6)^2 \text{ operações}}{10^8 \text{ operações/segundo}} = 20.000 \text{ segundos} \approx 5{,}56 \text{ horas},$$

enquanto que o computador pessoal gasta

$$\frac{50 \cdot 10^6 \log 10^6 \text{ operações}}{10^6 \text{ operações/segundo}} \approx 1.000 \text{ segundos} \approx 16{,}67 \text{ minutos}.$$

Usando um programa cujo tempo de execução tem menor taxa de crescimento, mesmo com um compilador pior, o computador pessoal é 20 vezes mais rápido que o supercomputador!

Esse exemplo mostra que os algoritmos e os programas, assim como os computadores, são uma **tecnologia**. O desempenho total do sistema depende da escolha de algoritmos e programas eficientes tanto quanto da escolha de computadores rápidos. Assim como rápidos avanços estão sendo feitos em outras tecnologias computacionais, eles estão sendo feitos em algoritmos e programas também.

## Exercícios

2.1 Qual o menor valor de $n$ tal que um programa com tempo de execução $100n^2$ é mais rápido que um programa cujo tempo de execução é $2^n$, supondo que os programas foram implementados no mesmo computador?

2.2 Suponha que estamos comparando as implementações dos métodos de ordenação por trocas sucessivas e por intercalação em um mesmo computador. Para entradas de tamanho $n$, o método das trocas sucessivas gasta $8n^2$ passos enquanto que o método da intercalação gasta $64n \log n$ passos. Para quais valores de $n$ o método das trocas sucessivas é melhor que o método da intercalação?

2.3 Expresse a função $n^3/1000 - 100n^2 - 100n + 3$ na notação $O$.

2.4 Para cada função $f(n)$ e tempo $t$ na tabela abaixo determine o maior tamanho $n$ de um problema que pode ser resolvido em tempo $t$, considerando que o programa soluciona o problema em $f(n)$ microssegundos.

| | 1 segundo | 1 minuto | 1 hora | 1 dia | 1 mês | 1 ano | 1 século |
|---|---|---|---|---|---|---|---|
| $\log n$ | | | | | | | |
| $\sqrt{n}$ | | | | | | | |
| $n$ | | | | | | | |
| $n \log n$ | | | | | | | |
| $n^2$ | | | | | | | |
| $n^3$ | | | | | | | |
| $2^n$ | | | | | | | |
| $n!$ | | | | | | | |

2.5 É verdade que $2^{n+1} = O(2^n)$? E é verdade que $2^{2n} = O(2^n)$?

2.6 Suponha que você tenha algoritmos com os cinco tempos de execução listados abaixo. Quão mais lento cada um dos algoritmos fica quando você (*i*) duplica o tamanho da entrada, ou (*ii*) incrementa em uma unidade o tamanho da entrada?

(a) $n^2$

(b) $n^3$

(c) $100n^2$

(d) $n \log_2 n$

(e) $2^n$

2.7 Suponha que você tenha algoritmos com os seis tempos de execução listados abaixo. Suponha que você tenha um computador capaz de executar $10^{10}$ operações por segundo e você precisa computar um resultado em no máximo uma hora de computação. Para cada um dos algoritmos, qual é o maior tamanho da entrada $n$ para o qual você poderia receber um resultado em uma hora?

(a) $n^2$

(b) $n^3$

(c) $100n^2$

(d) $n \log_2 n$

(e) $2^n$

(f) $2^{2^n}$

2.8 Rearranje a seguinte lista de funções em ordem crescente de taxa de crescimento. Isto é, se a função $g(n)$ sucede imediatamente a função $f(n)$ na sua lista, então é verdade que $f(n) = O(g(n))$.

$$f_1(n) = n^{2.5}$$
$$f_2(n) = \sqrt{2n}$$
$$f_3(n) = n + 10$$
$$f_4(n) = 10^n$$
$$f_5(n) = 100^n$$
$$f_6(n) = n^2 \log_2 n$$

2.9 Considere o problema de computar o valor de um polinômio em um ponto. Dados $n$ coeficientes $a_0, a_1, \ldots, a_{n-1}$ e um número real $x$, queremos computar $\sum_{i=0}^{n-1} a_i x^i$.

(a) Escreva um programa simples com tempo de execução de pior caso $O(n^2)$ para solucionar este problema.

(b) Escreva um programa com tempo de execução de pior caso $O(n)$ para solucionar este problema usando o método chamado de regra de Horner para reescrever o polinômio:

$$\sum_{i=1}^{n-1} a_i x^i = (\cdots (a_{n-1}x + a_{n-2})x + \cdots + a_1)x + a_0 \, .$$

2.10 Seja $A[0..n-1]$ um vetor de $n$ números inteiros distintos dois a dois. Se $i < j$ e $A[i] > A[j]$ então o par $(i, j)$ é chamado uma **inversão** de $A$.

(a) Liste as cinco inversões do vetor $A = \langle 2, 3, 8, 6, 1 \rangle$.

(b) Qual vetor com elementos no conjunto $\{1, 2, \ldots, n\}$ tem a maior quantidade de inversões? Quantas são?

(c) Escreva um programa que determine o número de inversões em qualquer permutação de $n$ elementos em tempo de execução de pior caso $O(n \log n)$.

AULA 3

# CORREÇÃO DE ALGORITMOS E PROGRAMAS

É fato que estamos muito interessados em construir algoritmos e programas eficientes, conforme vimos na aula 2. No entanto, de nada vale um algoritmo eficiente mas incorreto. Por correto, queremos dizer que o algoritmo sempre pára com a resposta correta para todo entrada. Devemos, assim, ser capazes de mostrar que nossos algoritmos são eficientes e corretos.

Há duas estratégias básicas para mostrar que um algoritmo, programa ou função está correto, dependendo de como foi descrito. Se é recursivo, então a indução matemática é usada imediatamente para mostrar sua correção. Por outro lado, se não-recursivo, então contém um ou mais processos iterativos, que são controlados por estruturas de repetição. Processos iterativos podem ser então documentados com invariantes, que nos ajudam a entender os motivos pelos quais o algoritmo, programa ou função programa está correto. Nesta aula veremos como mostrar que uma função, recursiva ou não, está correta.

Esta aula é inspirada em [1, 2].

## 3.1 Correção de funções recursivas

Mostrar a correção de uma função recursiva é um processo quase que imediato. Usando indução matemátca como ferramenta, a correção de uma função recursiva é dada naturalmente, visto que sua estrutura intrínseca nos fornece muitas informações úteis para uma prova de correção. Na aula 1, mostramos a correção da função recursiva `maximo` usando indução.

Vamos mostrar agora que a função `potR` descrita no exercício 1.1 está correta. Reproduzimos novamente a função `potR` a seguir.

```
/* Recebe um dois números inteiros x e n e devolve x a n-ésima potência */
int potR(int x, int n)
{
   if (n == 0)
      return 1;
   else
      return x * potR(x, n-1);
}
```

**Proposição 3.1.** *A função* `potR` *recebe dois números inteiros $x$ e $n$ e devolve corretamente o valor de $x^n$.*

*Demonstração.*

Vamos mostrar a proposição por indução em $n$.

Se $n = 0$ então a função devolve $1 = x^0 = x^n$.

Suponha que a função esteja correta para todo valor $k$, com $0 < k < n$. Ou seja, a função `potR` com parâmetros $x$ e $k$ devolve corretamente o valor $x^k$ para todo $k$, com $0 < k < n$.

Agora, vamos mostrar que a função está correta para $n > 0$. Como $n > 0$ então a última linha do corpo da função é executada:

```
return x * potR(x, n-1);
```

Então, como $n-1 < n$, por hipótese de indução, a chamada `potR(x, n-1)` nesta linha devolve corretamente o valor $x^{n-1}$. Logo, a chamada de `potR(x, n)` devolve $x \cdot x^{n-1} = x^n$. □

## 3.2 Correção de funções não-recursivas e invariantes

Funções não-recursivas, em geral, possuem uma ou mais estruturas de repetição. Essas estruturas, usadas para processar um conjunto de informações de entrada e obter um outro conjunto de informações de saída, também são conhecidas como processos iterativos da função. Como veremos daqui por diante, mostrar a correção de uma função não-recursiva é um trabalho mais árduo do que de uma função recursiva. Isso porque devemos, neste caso, extrair informações úteis desta função que explicam o funcionamento do processo iterativo e que nos permitam usar indução matemática para, por fim, mostrar que o processo está correto, isto é, que a função está correta. Essas informações são denominadas invariantes de um processo iterativo.

### 3.2.1 Definição

Um **invariante** de um processo iterativo é uma relação entre os valores das variáveis envolvidas neste processo que vale no início de cada iteração do mesmo. Os invariantes explicam o funcionamento do processo iterativo e permitem provar por indução que ele tem o efeito desejado.

Devemos provar três elementos sobre um invariante de um processo iterativo:

**Inicialização:** é verdadeiro antes da primeira iteração da estrutura de repetição;

**Manutenção:** se é verdadeiro antes do início de uma iteração da estrutura de repetição, então permanece verdadeiro antes da próxima iteração;

**Término:** quando a estrutura de repetição termina, o invariante nos dá uma propriedade útil que nos ajuda a mostrar que o algoritmo ou programa está correto.

Quando as duas primeiras propriedades são satisfeitas, o invariante é verdadeiro antes de toda iteração da estrutura de repetição. Como usamos invariantes para mostrar a correção de um algoritmo e/ou programa, a terceira propriedade é a mais importante, é aquela que permite mostrar de fato a sua correção.

Dessa forma, os invariantes explicam o funcionamento dos processos iterativos e permitem provar, por indução, que esses processos têm o efeito desejado.

### 3.2.2 Exemplos

Nesta seção veremos exemplos do uso dos invariantes para mostrar a correção de programas. O primeiro exemplo, dado no programa 3.1, é bem simples e o programa contém apenas variáveis do tipo inteiro e, obviamente, uma estrutura de repetição. O segundo exemplo, apresentado no programa 3.2, é um programa que usa um vetor no processo iterativo para solução do problema.

Considere então o programa 3.1, que recebe um número inteiro $n > 0$ e uma seqüência de $n$ números inteiros, e mostra a soma desses $n$ números inteiros. O programa 3.1 é simples e não usa um vetor para solucionar esse problema.

Programa 3.1: Soma $n$ inteiros fornecidos pelo(a) usuário(a).

```
#include <stdio.h>

/* Recebe um número inteiro n > 0 e uma seqüência de n números
   inteiros e mostra o resultado da soma desses números */
int main(void)
{
   int n, i, num, soma;

   printf("Informe n: ");
   scanf("%d", &n);

   soma = 0;
   for (i = 1; i <= n; i++) {
      /* variável soma contém o somatório dos
         primeiros i-1 números fornecidos */
      printf("Informe um número: ");
      scanf("%d", &num);
      soma = soma + num;
   }

   printf("Soma dos %d números é %d\n", n, soma);

   return 0;
}
```

É importante destacar o *comentário* descrito nas duas linhas seguintes à estrutura de repetição `for` do programa 3.1: este é o *invariante* desse processo iterativo. E como Feofiloff destaca em [2], o enunciado de um invariante é, provavelmente, o único tipo de comentário que vale a pena inserir no corpo de um algoritmo, programa ou função.

Então, podemos provar a seguinte proposição.

**Proposição 3.2.** *O programa 3.1 computa corretamente a soma de $n \geqslant 0$ números inteiros fornecidos pelo(a) usuário(a).*

*Demonstração.*

Por conveniência na demonstração, usaremos o modo matemático para expressar as variáveis do programa. Dessa forma, denotaremos $i$ no lugar de `i`, *soma* no lugar de `soma` e *num* ao invés de `num`. Quando nos referirmos ao $i$-ésimo número inteiro da seqüência de números inteiros fornecida pelo(a) usuário(a), que é armazenado na variável `num`, usaremos por conveniência a notação $num_i$.

Provar que o programa 3.1 está correto significa mostrar que para qualquer valor de $n$ e qualquer seqüência de $n$ números, a variável *soma* conterá, ao final do processo iterativo, o valor

$$soma = \sum_{i=1}^{n} num_i \, .$$

Vamos agora mostrar que o invariante vale no início da primeira iteração do processo iterativo. Como a variável *soma* contém o valor 0 (zero) e $i$ contém 1, é verdade que a variável *soma* contém a soma dos $i - 1$ primeiros números fornecidos pelo(a) usuário(a).

Suponha agora que o invariante valha no início da $i$-ésima iteração, com $1 < i < n$.

Vamos mostrar que o invariante vale no início da última iteração, quando $i$ contém o valor $n$. Por hipótese de indução, a variável *soma* contém o valor

$$\alpha = \sum_{i=1}^{n-1} num_i \, .$$

Dessa forma, no decorrer da $n$-ésima iteração, o(a) usuário(a) deve informar um número que será armazenado na variável $num_n$ e, então, a variável *soma* conterá o valor

$$\begin{aligned} soma &= \alpha + num_n \\ &= \left( \sum_{i=1}^{n-1} num_i \right) + num_n \\ &= \sum_{i=1}^{n} num_i \, . \end{aligned}$$

Portanto, isso mostra que o programa 3.1 de fato realiza a soma dos $n$ números inteiros fornecidos pelo(a) usuário(a). □

O próximo exemplo é dado pelo seguinte problema: dado um vetor com $n$ números inteiros fornecidos pelo(a) usuário(a), encontrar um valor máximo armazenado nesse vetor. O programa 3.2 é bem simples e se propõe a solucionar esse problema.

Programa 3.2: Mostra um maior valor em um vetor com $n$ números inteiros.

```
#include <stdio.h>

#define MAX 100

/* Recebe um número inteiro n > 0 e uma seqüência de n
   números inteiros e mostra um maior valor da seqüência */
int main(void)
{
   int n, vet[MAX], i, max;

   scanf("%d", &n);
   for (i = 0; i < n; i++)
      scanf("%d", &vet[i]);

   max = vet[0];
   for (i = 1; i < n; i++) {
      /* max é um maior elemento em vet[0..i-1] */
      if (vet[i] > max)
         max = vet[i];
   }
   printf("%d\n", max);

   return 0;
}
```

Vamos mostrar agora a correção do programa 3.2.

**Proposição 3.3.** *O programa 3.2 encontra um elemento máximo de um conjunto de $n$ números fornecidos pelo(a) usuário(a).*

*Demonstração.*

Novamente, por conveniência na demonstração usaremos o modo matemático para expressar as variáveis do programa: trocaremos `i` por $i$, `vet` por *vet* e `max` por *max*.

Provar que o programa 3.2 está correto significa mostrar que para qualquer valor de $n$ e qualquer seqüência de $n$ números fornecidos pelo(a) usuário(a) e armazenados em um vetor *vet*, a variável *max* conterá, ao final do processo iterativo, o valor do elemento máximo em $vet[0..n-1]$.

Vamos mostrar que o invariante vale no início da primeira iteração do processo iterativo. Como *max* contém o valor armazenado em $vet[0]$ e, em seguida, a variável $i$ é inicializada com o valor 1, então é verdade que a variável *max* contém o elemento máximo em $vet[0..i-1]$.

Suponha agora que o invariante valha no início da $i$-ésima iteração, com $1 < i < n$.

Vamos mostrar que o invariante vale no início da última iteração, quando $i$ contém o valor $n-1$. Por hipótese de indução, no início desta iteração a variável *max* contém o valor o elemento máximo de $vet[0..n-2]$. Então, no decorrer dessa iteração, o valor $vet[n-1]$ é comparado com *max* e dois casos devem ser avaliados:

(*i*) $vet[n-1] > max$

Isso significa que o valor $vet[n-1]$ é maior que qualquer valor armazenado em $vet[0..n-2]$. Assim, na linha 15 a variável *max* é atualizada com $vet[n-1]$ e portanto a variável *max* conterá, ao final desta última iteração, o elemento máximo da seqüência em $vet[0..n-1]$.

(*ii*) $vet[n-1] \leqslant max$

Isso significa que existe pelo menos um valor em $vet[0..n-2]$ que é maior ou igual a $vet[n-1]$. Por hipótese de indução, esse valor está armazenado em *max*. Assim, ao final desta última iteração, a variável *max* conterá o elemento máximo da seqüência em $vet[0..n-1]$.

Portanto, isso mostra que o programa 3.2 de fato encontra o elemento máximo em uma seqüência de $n$ números inteiros armazenados em um vetor. □

## Exercícios

3.1 O programa 3.3 recebe um número inteiro $n > 0$, uma seqüência de $n$ números inteiros, um número inteiro $x$ e verifica se $x$ pertence à seqüência de números.

Programa 3.3: Verifica se $x$ pertence à uma seqüência de $n$ números.

```
#include <stdio.h>

#define MAX 100

int main(void)
{
   int n, C[MAX], i, x;

   scanf("%d", &n);
   for (i = 0; i < n; i++)
      scanf("%d", &C[i]);
   scanf("%d", &x);

   i = 0;
   while (i < n && C[i] != x)
      /* x não pertence à C[0..i] */
      i++;

   if (i < n)
      printf("%d é o %d-ésimo elemento do vetor\n", x, i);
   else
      printf("%d não se encontra no vetor\n", x);

   return 0;
}
```

Mostre que o programa 3.3 está correto.

3.2 Escreva uma função com a seguinte interface:

```
void inverte(int n, int v[MAX])
```

que receba um número inteiro $n > 0$ e uma seqüência de $n$ números inteiros armazenados no vetor, e devolva o vetor a seqüência de números invertida. Mostre que sua função está correta.

3.3 Mostre que sua solução para o exercício 1.6 está correta.

3.4 Mostre que sua solução para o exercício 1.7 está correta.

3.5 Mostre que sua solução para o exercício 1.8 está correta.

AULA 4

# BUSCA

Como temos visto ao longo de muitas das aulas anteriores, a busca de um elemento em um conjunto é uma operação básica em Computação. A maneira como esse conjunto é armazenado na memória do computador permite que algumas estratégias possam ser usadas para realizar a tarefa da busca. Na aula de hoje revemos os métodos de busca que vimos usando corriqueiramente até o momento como uma subtarefa realizada na solução de diversos problemas práticos. A busca será fixada, como antes, com os dados envolvidos como sendo números inteiros e o conjunto de números inteiros onde a busca se dará é armazenado em um vetor. Além de rever essa estratégia, chamada de busca seqüencial, vemos ainda uma estratégia nova e muito eficiente de busca em um vetor ordenado, chamada de busca binária. Esta aula está completamente baseada no livro de P. Feofiloff [2], capítulos 3 e 7.

## 4.1 Busca seqüencial

Vamos fixar o conjunto e o elemento onde a busca se dará como sendo constituídos de números inteiros, observando que o problema da busca não se modifica essencialmente se esse tipo de dados for alterado. Assim, dado um número inteiro $n \geqslant 0$, um vetor de números inteiros $v[0..n-1]$ e um número inteiro $x$, considere o problema de encontrar um índice $k$ tal que $v[k] = x$. O problema faz sentido com qualquer $n \geqslant 0$. Observe que se $n = 0$ então o vetor é vazio e portanto essa entrada do problema não tem solução.

Como em [2], adotamos a convenção de devolver $-1$ caso o elemento $x$ não pertença ao vetor $v$. A convenção é satisfatória já que $-1$ não pertence ao conjunto $\{0, \ldots, n-1\}$ de índices válidos do vetor $v$. Dessa forma, basta percorrer o vetor do fim para o começo, como mostra a função `busca_sequencial` a seguir.

```
/* Recebe um número inteiro n >= 0, um vetor v[0..n-1] com n nú-
   meros inteiros e um número inteiro x e devolve k no intervalo
   [0, n-1] tal que v[k] == x. Se tal k não existe, devolve -1. */
int busca_sequencial(int n, int v[MAX], int x)
{
   int k;

   for (k = n - 1; k >= 0 && v[k] != x; k--)
      ;

   return k;
}
```

Observe como a função é eficiente e elegante, funcionando corretamente mesmo quando o vetor está vazio, isto é, quando $n$ vale 0.

Um exemplo de uma chamada à função `busca_sequencial` é apresentado abaixo:

```
i = busca_sequencial(v, n, x);
if (i >= 0)
   printf("Encontrei %d!\n", v[i]);
```

A função `busca_sequencial` pode ser escrita em versão recursiva. A idéia do código é simples: se $n = 0$ então o vetor é vazio e portanto $x$ não está em $v[0..n-1]$; se $n > 0$ então $x$ está em $v[0..n-1]$ se e somente se $x = v[n-1]$ ou $x$ está no vetor $v[0..n-2]$. A versão recursiva é então mostrada abaixo:

```
/* Recebe um número inteiro n >= 0, um vetor de números in-
   teiros v[0..n-1] e um número x e devolve k tal que 0 <= k
   < n e v[k] == x. Se tal k não existe, devolve -1. */
int busca_sequencial_R(int n, int v[MAX], int x)
{
   if (n == 0)
      return -1;
   else
      if (x == v[n - 1])
         return n - 1;
      else
         return busca_sequencial_R(n - 1, v, x);
}
```

A correção da função `busca_sequencial` é mostrada de forma semelhante àquela do exercício 3.1. Seu tempo de execução é linear no tamanho da entrada, isto é, é proporcional a $n$, ou ainda, é $O(n)$, como mostrado na aula 2. A função `busca_sequencial_R` tem correção e análise de eficiência equivalentes.

Vale a pena consultar o capítulo 3, páginas 12 e 13, do livro de P. Feofiloff [2], para verificar uma série de "maus exemplos" para solução do problema da busca, dentre deselegantes, ineficientes e até incorretos.

## 4.2 Busca em um vetor ordenado

Como em [2], adotamos as seguintes definições. Dizemos que um vetor de números inteiros $v[0..n-1]$ é **crescente** se $v[0] \leqslant v[1] \leqslant \cdots \leqslant v[n-1]$ e **decrescente** se $v[0] \geqslant v[1] \geqslant \cdots \geqslant v[n-1]$. Dizemos ainda que o vetor é **ordenado** se é crescente ou decrescente. Nesta seção, vamos focar no problema da busca de um elemento $x$ em um vetor ordenado $v[0..n-1]$.

No caso de vetores ordenados, uma pergunta equivalente mas ligeiramente diferente é formulada: em qual posição do vetor $v$ o elemento $x$ deveria estar? Dessa forma, o problema da busca pode ser reformulado da seguinte maneira: dado um número inteiro $n \geqslant 0$, um vetor de

números inteiros crescente $v[0..n-1]$ e um número inteiro $x$, encontrar um índice $k$ tal que

$$v[k-1] < x \leqslant v[k] \,. \tag{4.1}$$

Encontrar um índice $k$ como o da equação (4.1) significa então praticamente resolver o problema da busca, bastando comparar $x$ com $v[k]$.

Observe ainda que qualquer valor de $k$ no intervalo $[0, n]$ pode ser uma solução do problema da busca. Nos dois extremos do intervalo, a condição (4.1) deve ser interpretada adequadamente. Isto é, se $k = 0$, a condição se reduz apenas a $x \leqslant v[0]$, pois $v[-1]$ não faz sentido. Se $k = n$, a condição se reduz somente a $v[n-1] < x$, pois $v[n]$ não faz sentido. Tudo se passa como se o vetor $v$ tivesse um componente imaginário $v[-1]$ com valor $-\infty$ e um componente imaginário $v[n]$ com valor $+\infty$. Também, para simplificar um pouco o raciocínio, suporemos que $n \geqslant 1$.

Isso posto, observe então que uma busca seqüencial, recursiva ou não, como as funções `busca_sequencial` e `busca_sequencial_R` da seção 4.1, pode ser executada para resolver o problema. Vejamos então uma solução um pouco diferente na função `busca_ordenada`.

```
/* Recebe um número inteiro n > 0, um vetor de números in-
   teiros crescente v[0..n-1] e um número inteiro x e devol-
   ve um índice k em [0, n] tal que v[k-1] < x <= v[k] */
int busca_ordenada(int n, int v[MAX], int x)
{
   int k;

   for (k = 0; k < n && v[k] < x; k++)
      ;

   return k;
}
```

Uma chamada à função `buscaOrd` é mostrada a seguir:

```
i = busca_ordenada(n, v, x);
```

Se aplicamos a estratégia de busca seqüencial em um vetor ordenado, então certamente obtemos uma resposta correta, porém ineficiente do ponto de vista de seu consumo de tempo. Isso porque, no pior caso, a busca seqüencial realiza a comparação do elemento $x$ com cada um dos elementos do vetor $v$ de entrada. Ou seja, o problema da busca é resolvido em tempo proporcional ao número de elementos do vetor de entrada $v$, deixando de explorar sua propriedade especial de se encontrar ordenado. O tempo de execução de pior caso da função `busca_ordenada` permanece proporcional a $n$, o mesmo que das funções da seção 4.1.

Com uma busca binária, podemos fazer o mesmo trabalho de forma bem mais eficiente. A busca binária se baseia no método que usamos de modo automático para encontrar uma palavra no dicionário: abrimos o dicionário ao meio e comparamos a primeira palavra desta página com a palavra buscada. Se a primeira palavra é "menor" que a palavra buscada, jogamos fora a primeira metade do dicionário e repetimos a mesma estratégia considerando apenas a metade

restante. Se, ao contrário, a primeira palavra é "maior" que a palavra buscada, jogamos fora a segunda metade do dicionário e repetimos o processo. A função `busca_binaria` abaixo implementa essa idéia.

```
/* Recebe um número inteiro n > 0, um vetor de números in-
   teiros crescente v[0..n-1] e um número inteiro x e devol-
   ve um índice k em [0, n] tal que v[k-1] < x <= v[k] */
int busca_binaria(int n, int v[MAX], int x)
{
   int esq, dir, meio;

   esq = -1;
   dir = n;
   while (esq < dir - 1) {
      meio = (esq + dir) / 2;
      if (v[meio] < x)
         esq = meio;
      else
         dir = meio;
   }

   return dir;
}
```

Um exemplo de chamada à função `busca_binaria` é mostrado abaixo:

```
k = busca_binaria(n, v, x);
```

Para provar a correção da função `busca_binaria`, basta verificar o seguinte invariante:

> no início de cada repetição `while`, imediatamente antes da comparação de `esq` com `dir - 1`, vale a relação `v[esq] < x <= v[dir]`.

Com esse invariante em mãos, podemos usar a estratégia que aprendemos na aula 3 para mostrar finalmente que essa função está de fato correta.

Quantas iterações a função `busca_binaria` executa? O total de iterações revela o valor aproximado que representa o consumo de tempo dessa função em um dado vetor de entrada. Observe que em cada iteração, o tamanho do vetor $v$ é dado por `dir - esq - 1`. No início da primeira iteração, o tamanho do vetor é $n$. No início da segunda iteração, o tamanho do vetor é aproximadamente $n/2$. No início da terceira, aproximadamente $n/4$. No início da $(k+1)$-ésima, aproximadamente $n/2^k$. Quando $k > \log_2 n$, temos $n/2^k < 1$ e a execução da função termina. Assim, o número de iterações é aproximadamente $\log_2 n$. O consumo de tempo da função é proporcional ao número de iterações e portanto proporcional a $\log_2 n$. Esse consumo de tempo cresce com $n$ muito mais lentamente que o consumo da busca seqüencial.

Uma solução recursiva para o problema da busca em um vetor ordenado é apresentada a seguir. Antes, é necessário reformular ligeiramente o problema. A função recursiva `busca_binaria_R` procura o elemento $x$ no vetor crescente `v[esq..dir]`, supondo que o

valor $x$ está entre os extremos `v[esq]` e `v[dir]`.

```
/* Recebe dois números inteiros esq e dir, um vetor de números
   inteiros crescente v[esq..dir] e um número inteiro x tais
   que v[esq] < x <= v[dir] e devolve um índice k em
   [esq+1, dir] tal que v[k-1] < x <= v[k] */
int busca_binaria_R(int esq, int dir, int v[MAX], int x)
{
   int meio;

   if (esq == dir - 1)
      return dir;
   else {
      meio = (esq + dir) / 2;
      if (v[meio] < x)
         return busca_binaria_R(meio, dir, v, x);
      else
         return busca_binaria_R(esq, meio, v, x);
   }
}
```

Uma chamada da função `busca_binaria_R` pode ser realizada da seguinte forma:

```
k = busca_binaria_R(-1, n, v, x);
```

Quando a função `busca_binaria_R` é chamada com argumentos $(-1, n, v, x)$, ela chama a si mesma cerca de $\lfloor \log_2 n \rfloor$ vezes. Este número de chamadas é a profundidade da recursão e determina o tempo de execução da função.

## Exercícios

4.1 Tome uma decisão de projeto diferente daquela da seção 4.1: se $x$ não estiver em $v[0..n-1]$, a função deve devolver $n$. Escreva a versão correspondente da função `busca`. Para evitar o grande número de comparações de $k$ com $n$, coloque uma "sentinela" em $v[n]$.

```
/* Recebe um número inteiro n >= 0, um vetor de números intei-
   ros v[0..n-1] e um número inteiro x e devolve k no intervalo
   [0, n-1] tal que v[k] == x. Se tal k não existe, devolve n */
int busca_sequencial_sentinela(int n, int v[MAX+1], int x)
{
   int k;

   v[n] = x;
   for (k = 0; v[k] != x; k++)
      ;

   return k;
}
```

4.2 Considere o problema de determinar o valor de um elemento máximo de um vetor $v[0..n-1]$. Considere a função `maximo` abaixo.

```
int maximo(int n, int v[MAX])
{
   int i, x;

   x = v[0];
   for (i = 1; i < n; i++)
      if (x < v[i])
         x = v[i];

   return x;
}
```

(a) A função `maximo` acima resolve o problema?

(b) Faz sentido trocar `x = v[0]` por `x = 0`?

(c) Faz sentido trocar `x = v[0]` por `x = INT_MIN`[1]?

(d) Faz sentido trocar `x < v[i]` por `x <= v[i]`?

4.3 O autor da função abaixo afirma que ela decide se $x$ está no vetor $v[0..n-1]$. Critique seu código.

```
int buscaR2(int n, int v[MAX], int x)
{
   if (v[n-1] == x)
      return 1;
   else
      return buscaR2(n-1, v, x);
}
```

4.4 A operação de remoção consiste de retirar do vetor $v[0..n-1]$ o elemento que tem índice $k$ e fazer com que o vetor resultante tenha índices $0, 1, \ldots, n-2$. Essa operação só faz sentido se $0 \leqslant k < n$.

(a) Escreva uma função não-recursiva com a seguinte interface:

```
int remove(int n, int v[MAX], int k)
```

que remove o elemento de índice $k$ do vetor $v[0..n-1]$ e devolve o novo valor de $n$, supondo que $0 \leqslant k < n$.

(b) Escreva uma função recursiva para a remoção com a seguinte interface:

```
int removeR(int n, int v[MAX], int k)
```

[1] `INT_MIN` é o valor do menor número do tipo `int`.

4.5 A operação de inserção consiste em introduzir um novo elemento $y$ entre a posição de índice $k-1$ e a posição de índice $k$ no vetor $v[0..n-1]$, com $0 \leqslant k \leqslant n$.

(a) Escreva uma função não-recursiva com a seguinte interface:

```
int insere(int n, int v[MAX], int k, int y)
```

que insere o elemento $y$ entre as posições $k-1$ e $k$ do vetor $v[0..n-1]$ e devolve o novo valor de $n$, supondo que $0 \leqslant k \leqslant n$.

(b) Escreva uma função recursiva para a inserção com a seguinte interface:

```
int insereR(int n, int v[MAX], int k, int x)
```

4.6 Na busca binária, suponha que $v[i] = i$ para todo $i$.

(a) Execute a função `busca_binaria` com $n = 9$ e $x = 3$;

(b) Execute a função `busca_binaria` com $n = 14$ e $x = 7$;

(c) Execute a função `busca_binaria` com $n = 15$ e $x = 7$.

4.7 Execute a função `busca_binaria` com $n = 16$. Quais os possíveis valores de $m$ na primeira iteração? Quais os possíveis valores de $m$ na segunda iteração? Na terceira? Na quarta?

4.8 Confira a validade da seguinte afirmação: quando $n+1$ é uma potência de 2, o valor da expressão `(esq + dir)` é divisível por 2 em todas as iterações da função `busca_binaria`, quaisquer que sejam $v$ e $x$.

4.9 Responda as seguintes perguntas sobre a função `busca_binaria`.

(a) Que acontece se a sentença `while (esq < dir - 1)` for substituída pela sentença `while (esq < dir)`?

(b) Que acontece se a sentença `if (v[meio] < x)` for substituída pela sentença `if (v[meio] <= x)`?

(c) Que acontece se `esq = meio` for substituído por `esq = meio + 1` ou então por `esq = meio - 1`?

(d) Que acontece se `dir = meio` for substituído por `dir = meio + 1` ou então por `dir = meio - 1`?

4.10 Se $t$ segundos são necessários para fazer uma busca binária em um vetor com $n$ elementos, quantos segundos serão necessários para fazer uma busca em $n^2$ elementos?

4.11 Escreva uma versão da busca binária para resolver o seguinte problema: dado um inteiro $x$ e um vetor decrescente $v[0..n-1]$, encontrar $k$ tal que $v[k-1] > x \geqslant v[k]$.

4.12 Suponha que cada elemento do vetor $v[0..n-1]$ é uma cadeia de caracteres (ou seja, temos uma matriz de caracteres). Suponha também que o vetor está em ordem lexicográfica. Escreva uma função eficiente, baseada na busca binária, que receba uma cadeia de caracteres $x$ e devolva um índice $k$ tal que $x$ é igual a $v[k]$. Se tal $k$ não existe, a função deve devolver $-1$.

4.13 Suponha que cada elemento do vetor $v[0..n-1]$ é um registro com dois campos: o nome do(a) estudante e o número do(a) estudante. Suponha que o vetor está em ordem crescente de números. Escreva uma função de busca binária que receba o número de um(a) estudante e devolva seu nome. Se o número não estiver no vetor, a função deve devolver a cadeia de caracteres vazia.

4.14 Escreva uma função que receba um vetor crescente $v[0..n-1]$ de números inteiros e devolva um índice $i$ entre 0 e $n-1$ tal que $v[i] = i$. Se tal $i$ não existe, a função deve devolver $-1$. A sua função não deve fazer mais que $\lfloor \log_2 n \rfloor$ comparações envolvendo os elementos de $v$.

AULA 5

# ORDENAÇÃO: MÉTODOS ELEMENTARES

Além da busca, a ordenação é outra operação elementar em computação. Nesta aula revisaremos os métodos de ordenação elementares que já tivemos contato em aulas anteriores: o método das trocas sucessivas, da seleção e da inserção. Esses métodos foram projetados a partir de idéias simples e têm, como característica comum, tempo de execução de pior caso quadrático no tamanho da entrada. A aula é baseada no livro de P. Feofiloff [2].

## 5.1 Problema da ordenação

Lembrando da aula 4, um vetor $v[0..n-1]$ é **crescente** se $v[0] \leqslant v[1] \leqslant \cdots \leqslant v[n-1]$. O problema da ordenação de um vetor consiste em rearranjar, ou permutar, os elementos de um vetor $v[0..n-1]$ de tal forma que se torne crescente. Nesta aula nós discutimos três funções simples para solução desse problema. Nas aulas 6 e 7 examinaremos funções mais sofisticadas e eficientes.

## 5.2 Método das trocas sucessivas

O método das trocas sucessivas, popularmente conhecido como método da bolha, é um método simples de ordenação que, a cada passo, posiciona o maior elemento de um subconjunto de elementos do vetor de entrada na sua localização correta neste vetor. A função `trocas_sucessivas` implementa esse método.

```
/* Recebe um número inteiro n >= 0 e um vetor v de números inteiros
   com n elementos e rearranja o vetor v de modo que fique crescente */
void trocas_sucessivas(int n, int v[MAX])
{
   int i, j;

   for (i = n - 1; i > 0; i--)
      for (j = 0; j < i; j++)
         if (v[j] > v[j+1])
            troca(&v[j], &v[j+1]);

}
```

Um exemplo de chamada da função `trocas_sucessivas` é dado a seguir:

```
trocas_sucessivas(n, v);
```

Para entender a função `trocas_sucessivas` basta observar que no início de cada repetição do `for` externo vale que:

- o vetor $v[0..n-1]$ é uma permutação do vetor original,
- o vetor $v[i+1..n-1]$ é crescente e
- $v[j] \leqslant v[i+1]$ para $j = 0, 1, \ldots, i$.

Além disso, o consumo de tempo da função `trocas_sucesivas` é proporcional ao número de execuções da comparação "$v[j] > v[j+1]$", que é proporcional a $n^2$ no pior caso.

## 5.3 Método da seleção

O método de ordenação por seleção é baseado na idéia de escolher um menor elemento do vetor, depois um segundo menor elemento e assim por diante. A função `selecao` implementa esse método.

```
/* Recebe um número inteiro n >= 0 e um vetor v de números inteiros
   com n elementos e rearranja o vetor v de modo que fique crescente */
void selecao(int n, int v[MAX])
{
   int i, j, min;

   for (i = 0; i < n - 1; i++) {
      min = i;
      for (j = i+1; j < n; j++)
         if (v[j] < v[min])
            min = j;
      troca(&v[i], &v[min]);
   }

}
```

Um exemplo de chamada da função `selecao` é dado a seguir:

```
selecao(n, v);
```

Para entender como e por que o a função `selecao` funciona, basta observar que no início de cada repetição do `for` externo valem os seguintes invariantes:

- o vetor $v[0..n-1]$ é uma permutação do vetor original,
- o vetor $v[0..i-1]$ está em ordem crescente e

- $v[i-1] \leqslant v[j]$ para $j = i, i+1, \ldots, n-1$.

O terceiro invariante pode ser assim interpretado: $v[0..i-1]$ contém todos os elementos "pequenos" do vetor original e $v[i..n-1]$ contém todos os elementos "grandes". Os três invariantes garantem que no início de cada iteração os elementos $v[0], \ldots, v[i-1]$ já estão em suas posições definitivas.

Uma análise semelhante à que fizemos para a função `trocas_sucessivas` mostra que o método da inserção implementado pela função `selecao` consome $n^2$ unidades de tempo no pior caso.

## 5.4 Método da inserção

O método da ordenação por inserção é muito popular. É freqüentemente usado quando alguém joga baralho e quer manter as cartas de sua mão em ordem. A função `insercao` implementa esse método.

```
/* Recebe um número inteiro n >= 0 e um vetor v de números inteiros
   com n elementos e rearranja o vetor v de modo que fique crescente */
void insercao(int n, int v[MAX])
{
   int i, j, x;

   for (i = 1; i < n; i++) {
      x = v[i];
      for (j = i - 1; j >= 0 && v[j] > x; j--)
         v[j+1] = v[j];
      v[j+1] = x;
   }

}
```

Um exemplo de chamada da função `insercao` é dado a seguir:

```
insercao(n, v);
```

Para entender a função `insercao` basta observar que no início de cada repetição do `for` externo, valem os seguintes invariantes:

- o vetor $v[0..n-1]$ é uma permutação do vetor original e
- o vetor $v[0..i-1]$ é crescente.

O consumo de tempo da função `insercao` é proporcional ao número de execuções da comparação "$v[j] > x$", que é proporcional a $n^2$.

## Exercícios

5.1 Escreva uma função que verifique se um dado vetor $v[0..n-1]$ é crescente.

```
int verifica_ordem(int n, int v[MAX])
{

   for (i = 0; i < n - 1; i++)
      if (v[i] > v[i+1])
         return 0;
   return 1;
}
```

5.2 Que acontece se trocarmos a realação `i > 0` pela relação `i >= 0` no código da função `trocas_sucessivas`? Que acontece se trocarmos `j < i` por `j <= i`?

5.3 Troque a relação `v[j] > v[j + 1]` pela relação `v[j] >= v[j + 1]` no código da função `trocas_sucessivas`. A nova função continua produzindo uma ordenação crescente de $v[0..n-1]$?

5.4 Escreva uma versão recursiva do método de ordenação por trocas sucessivas.

5.5 Que acontece se trocarmos `i = 0` por `i = 1` no código da função `selecao`? Que acontece se trocarmos `i < n-1` por `i < n`?

5.6 Troque `v[j] < v[min]` por `v[j] <= v[min]` no código da função `selecao`. A nova função continua produzindo uma ordenação crescente de $v[0..n-1]$?

5.7 Escreva uma versão recursiva do método de ordenação por seleção.

5.8 No código da função `insercao`, troque `v[j] > x` por `v[j] >= x`. A nova função continua produzindo uma ordenação crescente de $v[0..n-1]$?

5.9 No código da função `insercao`, que acontece se trocarmos `i = 1` por `i = 0`? Que acontece se trocarmos `v[j+1] = x` por `v[j] = x`?

5.10 Escreva uma versão recursiva do método de ordenação por inserção.

5.11 Escreva uma função que rearranje um vetor $v[0..n-1]$ de modo que ele fique em ordem estritamente crescente.

5.12 Escreva uma função que permute os elementos de um vetor $v[0..n-1]$ de modo que eles fiquem em ordem decrescente.

AULA 6

# ORDENAÇÃO POR INTERCALAÇÃO

Na aula 5 revimos os métodos de ordenação mais básicos, que são todos iterativos, simples e têm tempo de execução de pior caso proporcional a $n^2$, onde $n$ é o tamanho da entrada. Métodos mais eficientes de ordenação são baseados em recursão, técnica introduzida na aula 1. Nesta aula, estudamos o método da ordenação por intercalação, conhecido como *mergesort*.

A ordenação por intercalação, que veremos nesta aula, e a ordenação por separação, que veremos na aula 7, são métodos eficientes baseados na técnica recursiva chamada **dividir para conquistar**, onde quebramos o problema em vários subproblemas de menor tamanho que são similares ao problema original, resolvemos esses subproblemas recursivamente e então combinamos essas soluções para produzir uma solução para o problema original. Esa aula é baseada no livro de P. Feofiloff [2] e no livro de Cormen et. al [1].

## 6.1 Dividir para conquistar

A técnica de dividir para conquistar é uma técnica geral de construção de algoritmos e programas, tendo a recursão como base, que envolve três passos em cada nível da recursão:

**Dividir** o problema em um número de subproblemas;

**Conquistar** os subproblemas solucionando-os recursivamente. No entanto, se os tamanhos dos subproblemas são suficientemente pequenos, resolva os subproblemas de uma maneira simples;

**Combinar** as soluções dos subproblemas na solução do problema original.

Como mencionamos na aula 4, o algoritmo da busca binária é um método de busca que usa a técnica de dividir para conquistar na solução do problema da busca. O método de ordenação por intercalação, que veremos nesta aula, e o método da ordenação por separação, que veremos na aula 7, também são algoritmos baseados nessa técnica.

## 6.2 Problema da intercalação

Antes de apresentar o método da ordenação por intercalação, precisamos resolver um problema anterior, que auxilia esse método, chamado de problema da intercalação. O problema da intercalação pode ser descrito de forma mais geral como a seguir: dados dois conjuntos crescentes $A$ e $B$, com $m$ e $n$ elementos respectivamente, obter um conjunto crescente $C$ a partir

de $A$ e $B$. Variantes sutis desse problema geral podem ser descritas como no caso em que se permite ou não elementos iguais nos dois conjuntos de entrada, isto é, conjuntos de entrada $A$ e $B$ tais que $A \cap B \neq \emptyset$ ou $A \cap B = \emptyset$.

O problema da intercalação que queremos resolver aqui é mais específico e pode ser assim descrito: dados dois vetores crescentes $v[p..q-1]$ e $v[q..r-1]$, rearranjar $v[p..r-1]$ em ordem crescente. Isso significa que queremos de alguma forma intercalar os vetores $v[0..q-1]$ e $v[q..r-1]$. Nesse caso, à primeira vista parece que os vetores de entrada podem ter elementos em comum. Entretanto, este não é o caso, já que estamos considerando o mesmo conjunto inicial de elementos armazenados no vetor $v$. Uma maneira fácil de resolver o problema da intercalação é usar um dos métodos de ordenação da aula 5 tendo como entrada o vetor $v[p..r-1]$. Essa solução, no entanto, tem consumo de tempo de pior caso proporcional ao quadrado do número de elementos do vetor e é ineficiente por desconsiderar as características dos vetores $v[p..q-1]$ e $v[q..r-1]$. Uma solução mais eficiente, que usa um vetor auxiliar, é mostrada a seguir.

```
/* Recebe os vetores crescentes v[p..q-1] e v[q..r-1]
   e rearranja v[p..r-1] em ordem crescente */
void intercala(int p, int q, int r, int v[MAX])
{
   int i, j, k, w[MAX];

   i = p;
   j = q;
   k = 0;
   while (i < q && j < r) {
      if (v[i] < v[j]) {
         w[k] = v[i];
         i++;
      }
      else {
         w[k] = v[j];
         j++;
      }
      k++;
   }
   while (i < q) {
      w[k] = v[i];
      i++;
      k++;
   }
   while (j < r) {
      w[k] = v[j];
      j++;
      k++;
   }
   for (i = p; i < r; i++)
      v[i] = w[i-p];
}
```

A função `intercala` tem tempo de execução de pior caso proporcional ao número de comparações entre os elementos do vetor, isto é, $r - p$. Assim, podemos dizer que o consumo de tempo no pior caso da função `intercala` é proporcional ao número de elementos do vetor de entrada.

## 6.3 Ordenação por intercalação

Com o problema da intercalação resolvido, podemos agora descrever uma função que implementa o método da ordenação por intercalação. Nesse método, dividimos ao meio um vetor $v$ com $r - p$ elementos, ordenamos recursivamente essas duas metades de $v$ e então as intercalamos. A função `mergesort` a seguir é recursiva e a base da recursão ocorre quando $p \geqslant r - 1$, quando não é necessário qualquer processamento.

```
/* Recebe um vetor v[p..r-1] e o rearranja em ordem crescente */
void mergesort(int p, int r, int v[MAX])
{
   int q;

   if (p < r - 1) {
      q = (p + r) / 2;
      mergesort(p, q, v);
      mergesort(q, r, v);
      intercala(p, q, r, v);
   }
}
```

Como a expressão $(p + q)/2$ da função `mergesort` é do tipo inteiro, observe que seu resultado é, na verdade, avaliado como $\lfloor \frac{p+q}{2} \rfloor$.

Observe também que para ordenar um vetor $v[0..n-1]$ basta chamar a função `mergesort` com os seguintes argumentos:

```
mergesort(0, n, v);
```

Vejamos um exemplo de execução da função `mergesort` na figura 6.1, para um vetor de entrada $v[0..7] = \{4, 6, 7, 3, 5, 1, 2, 8\}$ e chamada

```
mergesort(0, 8, v);
```

Observe que as chamadas recursivas são realizadas até a linha divisória imaginária ilustrada na figura, quando $p \geqslant r - 1$. A partir desse ponto, a cada volta de um nível de recursão, uma intercalação é realizada. No final, uma última intercalação é realizada e o vetor original torna-se então um vetor crescente com os mesmos elementos de entrada.

Qual o desempenho da função `mergesort` quando queremos ordenar um vetor $v[0..n-1]$? Suponha, para efeito de simplificação, que $n$ é uma potência de 2. Se esse não é o caso, podemos examinar duas potências de 2 consecutivas, justamente aquelas tais que $2^{k-1} < n \leqslant 2^k$, para algum $k \geqslant 0$. Observe então que o número de elementos do vetor é diminuído a aproximadamente metade a cada chamada da função `mergesort`. Ou seja, o número aproximado de chamadas é proporcional a $\log_2 n$. Na primeira vez, o problema original é reduzido a dois subproblemas onde é necessário ordenar os vetores $v[0..\frac{n}{2} - 1]$ e $v[\frac{n}{2}..n - 1]$. Na segunda vez, cada

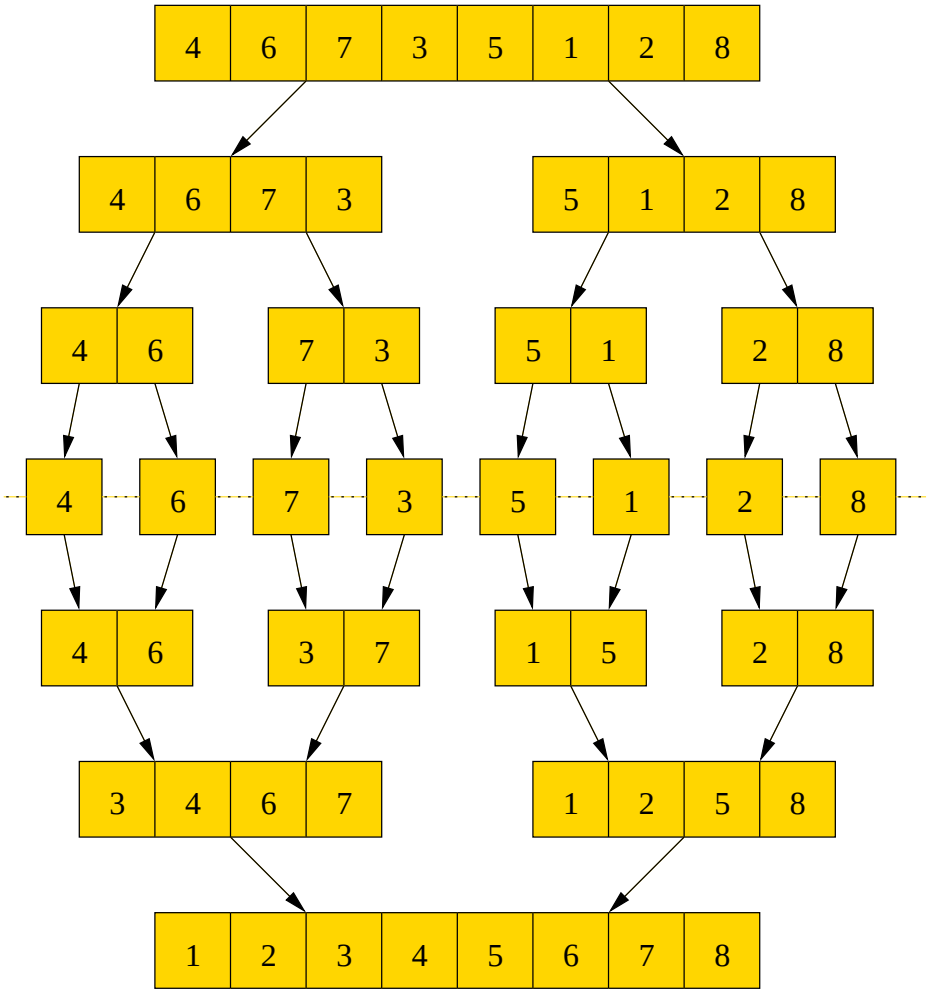



Figura 6.1: Exemplo de execução da ordenação por intercalação.

um dos subproblemas são ainda divididos em mais dois subproblemas cada, gerando quatro subproblemas no total, onde é necessário ordenar os vetores $v[0..\frac{n}{4}-1]$, $v[\frac{n}{4}..\frac{n}{2}-1]$, $v[\frac{n}{2}..\frac{3n}{4}-1]$ e $v[\frac{3n}{4}..n-1]$. E assim por diante. Além disso, como já vimos, o tempo total que a função `intercala` gasta é proporcional ao número de elementos do vetor $v$, isto é, $r-p$. Portanto, a função `mergesort` consome tempo proporcional a $n \log_2 n$.

## Exercícios

6.1 Simule detalhadamente a execução da função `mergesort` sobre o vetor de entrada $v[0..7] = \{3, 41, 52, 26, 38, 57, 9, 49\}$.

6.2 A função `intercala` está correta nos casos extremos $p = q$ e $q = r$?

6.3 Um algoritmo de intercalação é **estável** se não altera a posição relativa dos elementos que têm um mesmo valor. Por exemplo, se o vetor tiver dois elementos de valor 222, um algoritmo de intercalação estável manterá o primeiro 222 antes do segundo. A função

`intercala` é estável? Se a comparação `v[i] < v[j]` for trocada por `v[i] <= v[j]` a função fica estável?

6.4 O que acontece se trocarmos `(p + r)/2` por `(p + r - 1)/2` no código da função `mergesort`? Que acontece se trocarmos `(p + r)/2` por `(p + r + 1)/2`?

6.5 Escreva uma versão da ordenação por intercalação que rearranje um vetor $v[p..r-1]$ em ordem decrescente.

6.6 Escreva uma função eficiente que receba um conjunto $S$ de $n$ números reais e um número real $x$ e determine se existe um par de elementos em $S$ cuja soma é exatamente $x$.

6.7 Agora que você aprendeu o método da ordenação por intercalação, o problema a seguir, que já vimos na aula 2, exercício 2.10, fica bem mais fácil de ser resolvido.

Seja $A$ um vetor de $n$ números inteiros distintos. Se $i < j$ e $A[i] > A[j]$ então o par $(i, j)$ é chamado uma **inversão** de $A$.

(a) Liste as cinco inversões do vetor $\{2, 3, 8, 6, 1\}$.

(b) Qual vetor com elementos do conjunto $\{1, 2, \ldots, n\}$ tem o maior número de inversões? Quantas são?

(c) Qual a relação entre o tempo de execução da ordenação por inserção e o número de inversões em um vetor de entrada? Justifique sua resposta.

(d) Modificando a ordenação por intercalação, escreva uma função eficiente, com tempo de execução $O(n \log n)$, que determine o número de inversões em uma permutação de $n$ elementos.

6.8 Escreva um programa para comparar experimentalmente o desempenho da função `mergesort` com o das funções `trocas_sucessivas`, `selecao` e `insercao` da aula 5. Use um vetor com números (pseudo-)aleatórios para fazer os testes.

6.9 Veja animações dos métodos de ordenação que já vimos nas seguintes páginas:

- Sort Animation de R. Mohammadi;
- Sorting Algorithms de J. Harrison;
- Sorting Algorithms de P. Morin;
- Sorting Algorithms Animations de D. R. Martin.

AULA 7

# ORDENAÇÃO POR SEPARAÇÃO

O método de ordenação por separação resolve o problema da ordenação descrito na aula 5, onde se deve rearranjar um vetor $v[0..n-1]$ de modo que se torne crescente. Em geral, este método é muito mais rápido que os métodos elementares vistos na aula 5, no entanto pode ser lento, tanto quanto os métodos elementares para algumas entradas do problema.

O método da ordenação por separação é recursivo e também se baseia na estratégia de dividir para conquistar. O método é comumente conhecido como *quisksort*. É freqüentemente usado na prática para ordenação já que é rápido "na média" e apenas para algumas entradas especiais o método é lento como os métodos elementares de ordenação.

Esta aula é baseada nos livros de P. Feofiloff [2] e Cormen et. al [1].

## 7.1 Problema da separação

O problema da separação e sua solução computacional são o ponto-chave do método da ordenação por separação. Informalmente, no problema da separação queremos rearranjar um vetor $v[p..r]$ de modo que os elementos pequenos fiquem todos do lado esquerdo e os grandes fiquem todos do lado direito de $v$. Note que, dessa descrição, nosso desejo é que os dois lados tenham aproximadamente o mesmo número de elementos, salvo alguns casos, que podem ser menos equilibrados. Ademais, é importante que a separação não seja degenerada, isto é, que deixe um dos lados vazio. Seguindo [2], a dificuldade está em construir uma função que resolva o problema de maneira rápida e não use um vetor auxiliar.

O problema da separação que estamos interessados pode então ser formulado da seguinte maneira:

> rearranjar o vetor $v[p..r]$ de modo que tenhamos
>
> $$v[p..q] \leqslant v[q+1..r]$$
>
> para algum $q$ em $p..r-1$. Note que a expressão $v[p..q] \leqslant v[q+1..r]$ significa que $v[i] \leqslant v[j]$ para todo $i$, $p \leqslant i \leqslant q$, e todo $j$, $q < j \leqslant r$.

A função `separa` abaixo soluciona o problema da separação de forma eficiente. No início, um elemento de referência $x$, chamado de **pivô**, é escolhido e, a partir de então, o significado de *pequeno* e *grande* passa a ser associado a esse pivô: os elementos do vetor que forem maiores que $x$ serão considerados grandes e os demais serão considerados pequenos.

```
/* Recebe um par de números inteiros p e r, com p <= r e um vetor v[p..r]
   de números inteiros e rearranja seus elementos e devolve um número in-
   teiro j em p..r tal que v[p..j-1] <= v[j] < v[j+1..r] */
int separa(int p, int r, int v[MAX])
{
   int x, i, j;

   x = v[p];
   i = p - 1;
   j = r + 1;
   while (1) {
      do {
         j--;
      } while (v[j] > x);
      do {
         i++;
      } while (v[i] < x);
      if (i < j)
         troca(&v[i], &v[j]);
      else
         return j;
   }

}
```

Um exemplo de execução da função **separa** é apresentado na figura 7.1.

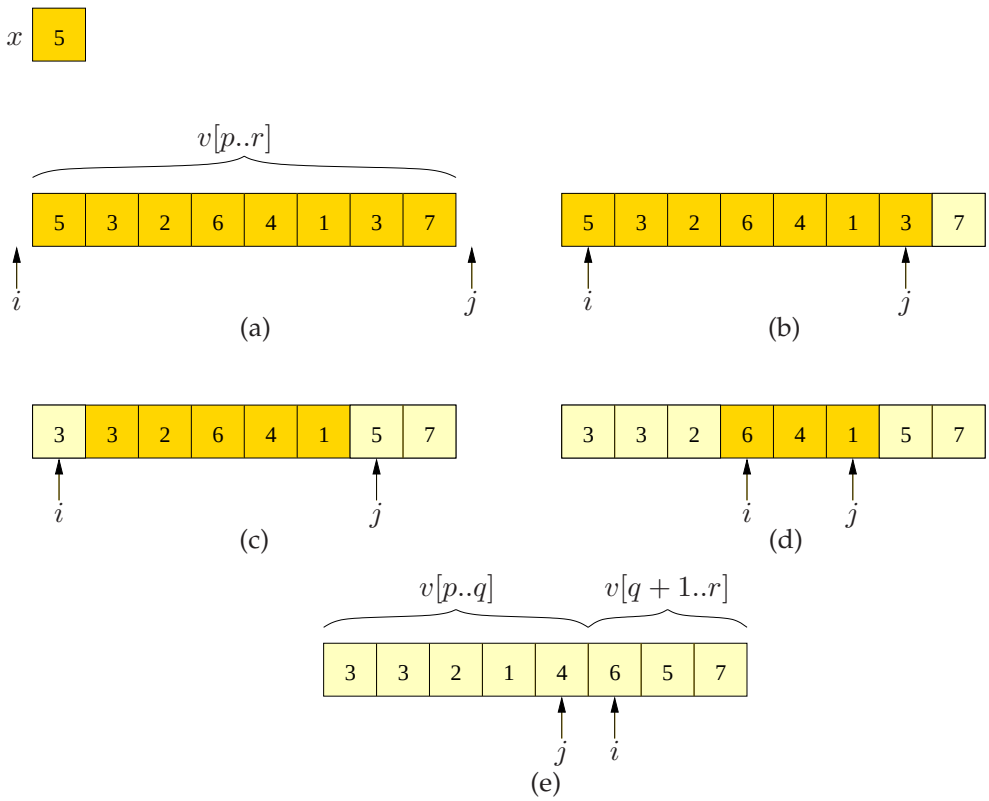



Figura 7.1: Uma execução da função **separa**.

O corpo da estrutura de repetição `while` da função `separa` é repetido até que $i \geqslant j$ e, neste ponto, o vetor $v[p..r]$ acabou de ser separado em $v[p..q]$ e $v[q+1..r]$, com $p \leqslant q < r$, tal que nenhum elemento de $v[p..q]$ é maior que um elemento de $v[q+1..r]$. O valor $q = j$ é devolvido então pela função `separa`.

Em outras palavras, podemos dizer que no início de cada iteração valem os seguintes invariantes:

- $v[p..r]$ é uma permutação do vetor original,
- $v[p..i] \leqslant x \leqslant v[j..r]$ e
- $p \leqslant i \leqslant j \leqslant r$.

O número de iterações que a função realiza é proporcional a $r - p + 1$, isto é, proporcional ao número de elementos do vetor.

## 7.2 Ordenação por separação

Com uma função que soluciona o problema da separação, podemos descrever agora o método da ordenação por separação. Esse método também usa a estratégia de dividir para conquistar e o faz de maneira semelhante ao método da ordenação por intercalação, mas com chamadas "invertidas".

```
/* Recebe um vetor v[p..r-1] e o rearranja em ordem crescente */
void quicksort(int p, int r, int v[MAX])
{
   int q;

   if (p < r) {
      q = separa(p, r, v);
      quicksort(p, q, v);
      quicksort(q+1, r, v);
   }
}
```

Uma chamada da função `quicksort` para ordenação de um vetor $v[0..n-1]$ deve ser feita como a seguir:

```
quicksort(0, n-1, v);
```

Observe ainda que a função `quicksort` está correta mesmo quando $p > r$, isto é, quando o vetor está vazio.

O consumo de tempo do método de ordenação por separação é proporcional ao número de comparações realizadas entre os elementos do vetor. Se o índice devolvido pela função `separa` sempre tiver valor mais ou menos médio de $p$ e $r$, isto é, próximo a $\lfloor (p + r)/2 \rfloor$,

então o número de comparações será aproximadamente $n \log_2 n$. Caso contrário, o número de comparações será da ordem de $n^2$. Observe que isso ocorre, por exemplo, quando o vetor já estiver ordenado ou quase-ordenado. Portanto, o consumo de tempo de pior caso da ordenação por separação não é melhor que o dos métodos elementares vistos na aula 5. Felizmente, o pior caso para a ordenação por separação é raro. Dessa forma, o consumo de tempo *médio* da função `quicksort` é proporcional a $n \log_2 n$.

## Exercícios

7.1 Ilustre a operação da função `separa` sobre o vetor $v$ que contém os elementos do conjunto $\{13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21\}$.

7.2 Qual o valor de $q$ a função `separa` devolve quando todos os elementos no vetor $v[p..r]$ têm o mesmo valor?

7.3 A função `separa` produz o resultado correto quando $p = r$?

7.4 Escreva uma função que rearranje um vetor $v[p..r]$ de números inteiros de modo que os elementos negativos e nulos fiquem à esquerda e os positivos fiquem à direita. Em outras palavras, rearranje o vetor de modo que tenhamos $v[p..q-1] \leqslant 0$ e $v[q..r] > 0$ para algum $q$ em $p..r+1$. Procure escrever uma função eficiente que não use um vetor auxiliar.

7.5 Digamos que um vetor $v[p..r]$ está **arrumado** se existe $q$ em $p..r$ que satisfaz

$$v[p..q-1] \leqslant v[q] < v[q+1..r] \,.$$

Escreva uma função que decida se $v[p..r]$ está arrumado. Em caso afirmativo, sua função deve devolver o valor de $q$.

7.6 Que acontece se trocarmos a expressão `if (p < r)` pela expressão `if (p != r)` no corpo da função `quicksort`?

7.7 Compare as funções `quicksort` e `mergesort`. Discuta as semelhanças e diferenças.

7.8 Como você modificaria a função `quicksort` para ordenar elementos em ordem decrescente?

7.9 Os bancos freqüentemente gravam transações sobre uma conta corrente na ordem das datas das transações, mas muitas pessoas preferem receber seus extratos bancários em listagens ordenadas pelo número do cheque emitido. As pessoas em geral emitem cheques na ordem da numeração dos cheques, e comerciantes usualmente descontam estes cheques com rapidez razoável. O problema de converter uma lista ordenada por data de transação em uma lista ordenada por número de cheques é portanto o problema de ordenar uma entrada quase já ordenada. Argumente que, neste caso, o método de ordenação por inserção provavelmente se comportará melhor do que o método de ordenação por separação.

7.10 Forneça um argumento cuidadoso para mostrar que a função `separa` é correta. Prove o seguinte:

(a) Os índices $i$ e $j$ nunca referenciam um elemento de $v$ fora do intervalo $[p..r]$;

(b) O índice $j$ não é igual a $r$ quando `separa` termina (ou seja, a partição é sempre não trivial);

(c) Todo elemento de $v[p..j]$ é menor ou igual a todo elemento de $v[j+1..r]$ quando a função `separa` termina.

7.11 Escreva uma versão da função `quicksort` que coloque um vetor de cadeias de caracteres em ordem lexicográfica.

7.12 Considere a seguinte solução do problema da separação, devido a N. Lomuto. Para separar $v[p..r]$, esta versão incrementa duas regiões, $v[p..i]$ e $v[i+1..j]$, tal que todo elemento na primeira região é menor ou igual a $x = v[r]$ e todo elemento na segunda região é maior que $x$.

```
/* Recebe um par de números inteiros p e r, com p <= r e um vetor v[p..r]
   de números inteiros e rearranja seus elementos e devolve um número in-
   teiro i em p..r tal que v[p..i] <= v[r] e v[r] < v[i+1..r] */
int separa_Lomuto(int p, int r, int v[MAX])
{
   int x, i, j;

   x = v[r];
   i = p - 1;
   for (j = p; j <= r; j++)
      if (v[j] <= x) {
         i++;
         troca(&v[i], &v[j]);
      }
   if (i < r)
      return i;
   else
      return i - 1;
}
```

(a) Ilustre a operação da função `separa_Lomuto` sobre o vetor $v$ que contém os elementos $\{13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21\}$.

(b) Argumente que `separa_Lomuto` está correta.

(c) Qual o número máximo de vezes que um elemento pode ser movido pelas funções `separa` e `separa_Lomuto`? Justifique sua resposta.

(d) Argumente que `separa_Lomuto`, assim como `separa`, tem tempo de execução proporcional a $n$ sobre um vetor de $n = r - p + 1$ elementos.

(e) Como a troca da função `separa` pela função `separa_Lomuto` afeta o tempo de execução do método da ordenação por separação quando todos os valores de entrada são iguais?

7.13 A função `quicksort` contém duas chamadas recursivas para ela própria. Depois da chamada da `separa`, o sub-vetor esquerdo é ordenado recursivamente e então o sub-vetor direito é ordenado recursivamente. A segunda chamada recursiva no corpo da função `quicksort` não é realmente necessária; ela pode ser evitada usando uma estrutura de controle iterativa. Essa técnica, chamada **recursão de cauda**, é fornecida automaticamente

por bons compiladores. Considere a seguinte versão da ordenação por separação , que simula a recursão de cauda.

```
/* Recebe um vetor v[p..r-1] e o rearranja em ordem crescente */
void quicksort2(int p, int r, int v[MAX])
{
   while (p < r) {
      q = separa(p, r, v);
      quicksort2(p, q, v);
      p = q + 1;
   }
}
```

Ilustre a operação da função `quicksort2` sobre o vetor $v$ que contém os elementos $\{21, 7, 5, 11, 6, 42, 13, 2\}$. Use a chamada `quicksort2(0, n-1, v)`. Argumente que a função `quicksort2` ordena corretamente o vetor $v$.

7.14 Um famoso programador propôs o seguinte método de ordenação de um vetor $v[0..n-1]$ de números inteiros:

```
/* Recebe um vetor v[i..j] e o rearranja em ordem crescente */
void silly_sort(int i, int j, int v[MAX])
{
   int k;

   if (v[i] > v[j])
      troca(&v[i], &v[j]);
   if (i + 1 < j) {
      k = (j - i + 1) / 3;
      silly_sort(i, j-k, v);
      silly_sort(i+k, j, v);
      silly_sort(i, j-k, v);
   }
}
```

Ilustre a operação desse novo método de ordenação sobre o vetor $v$ que contém os elementos $\{21, 7, 5, 11, 6, 42, 13, 2\}$. Use a chamada `silly_sort(0, n-1, v)`. Argumente que a função `silly_sort` ordena corretamente o vetor $v$.

7.15 Veja animações dos métodos de ordenação que já vimos nas seguintes páginas:

- Sort Animation de R. Mohammadi;
- Sorting Algorithms de J. Harrison;
- Sorting Algorithms de P. Morin;
- Sorting Algorithms Animations de D. R. Martin.

7.16 Familiarize-se com a função `qsort` da biblioteca `stdlib` da linguagem C.

AULA 8

# LISTAS DE PRIORIDADES

Nesta aula vamos estudar uma nova estrutura de dados chamada de lista de prioridades. Em uma lista como esta, as relações entre os elementos do conunto que compõe a lista se dão através das suas prioridades. Perguntas como qual o elemento com a maior prioridade e operações de inserção e remoção de elementos deste conjunto, ou alteração de prioridades de elementos da lista, são tarefas associadas a estruturas como esta. Listas de prioridades são estruturas simples e eficientes para solução de diversos problemas práticos como, por exemplo, o escalonamento de processos em um computador.

Esta aula é baseada especialmente nas referências [1, 2, 13].

## 8.1 *Heaps*

Antes de estudar as listas de prioridades, precisamos estudar com cuidado uma outra estrutura de dados, que é base para aquelas, chamada *heap*[1]. Um heap nada mais é que uma estrutura de dados armazenada em um vetor. Vale observar que cada célula do vetor pode conter um registro com vários campos, mas o campo mais importante é aquele que armazena um número, a sua **prioridade** ou **chave**. Como os outros dados associados à prioridade são supérfluos para o funcionamento de um heap, optamos por trabalhar apenas com um vetor de números inteiros para abrigá-lo, onde cada célula sua contém uma prioridade. Dessa forma, um heap é uma coleção de elementos identificados por suas prioridades armazenadas em um vetor numérico $S$ satisfazendo a seguinte propriedade:

$$S[\lfloor (i-1)/2 \rfloor] \geqslant S[i]\,, \tag{8.1}$$

para todo $i \geqslant 1$. Um vetor $S$ com a propriedade (8.1) é chamado um **max-heap**. Do mesmo modo, a propriedade (8.1) é chamada **propriedade max-heap**. O vetor $S$ apresentado na figura 8.1 é um max-heap.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $S$ | 26 | 18 | 14 | 16 | 8 | 9 | 11 | 4 | 12 | 6 |

Figura 8.1: Um max-heap com 10 elementos.

[1] A tradução de *heap* do inglês é monte, pilha, amontoado, montão. O termo *heap*, sem tradução, será usado daqui por diante, já que se encontra bem estabelecido na área.

Se o vetor $S$ tem a propriedade

$$S[\lfloor (i-1)/2 \rfloor] \leqslant S[i]\,, \tag{8.2}$$

para todo $i \geqslant 1$, então $S$ é chamado **min-heap** e a propriedade (8.2) é chamada **propriedade min-heap**. Max-heaps e min-heaps são semelhantes mas, para nossas aplicações neste momento, estamos interessados apenas em max-heaps.

Um max-heap pode também ser visto como uma **árvore binária**[2] com seu último nível preenchido da esquerda para direita. Uma ilustração do max-heap da figura 8.1 é mostrada na figura 8.2.

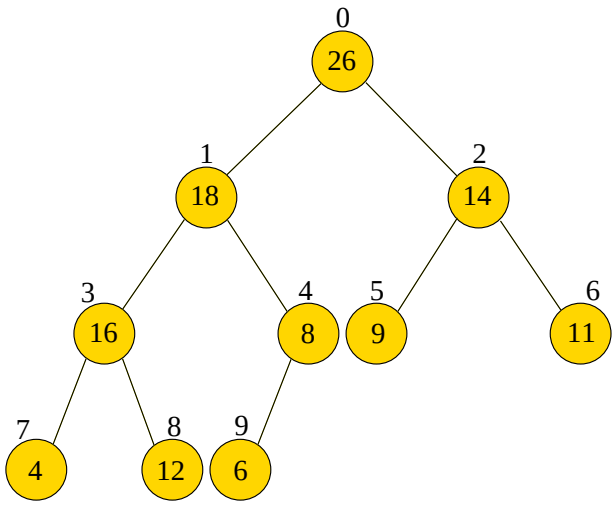



Figura 8.2: Representação do max-heap da figura 8.1 como uma árvore binária, com seu último nível preenchido da esquerda para diereita.

Observe que a visualização de um max-heap como uma árvore binária nos permite verificar a propriedade (8.1) facilmente. Em particular, note que o conteúdo de um nó da árvore é maior ou igual aos conteúdos dos nós que são seus filhos. Por conta dessa semelhança, podemos descrever operações básicas que nos permitem percorrer o max-heap facilmente. As operações são implementadas como funções.

```
/* Recebe um índice i em um max-heap e devolve o "pai" de i */
int pai(int i)
{
   if (i == 0)
      return 0;
   else
      return (i - 1) / 2;
}
```

[2] Uma árvore para os computólogos é um objeto bem diferente daquele visto pelos biólogos. Sua raiz está posicionada "no ar" e suas folhas estão posicionadas "no chão". A árvore é binária porque todo nó tem, no máximo, dois filhos. A definição formal de uma árvore binária foge ao escopo de Algoritmos e Programação II e será vista mais adiante.

```
/* Recebe um índice i em um max-heap e devolve o "filho esquerdo" de i */
int esquerdo(int i)
{
   return 2 * (i + 1) - 1;
}
```

```
/* Recebe um índice i em um max-heap e devolve o "filho direito" de i */
int direito(int i)
{
   return 2 * (i + 1);
}
```

Dadas as operações implementadas nas três funções acima, a propriedade (8.1) pode então ser reescrita da seguinte forma:

$$S[\,\texttt{pai(}i\texttt{)}\,] \geqslant S[i]\,, \tag{8.3}$$

para todo $i$, com $i \geqslant 0$.

Se olhamos para um max-heap como uma árvore binária, definimos a **altura** de um nó no max-heap como o número de linhas ou arestas no caminho mais longo do nó a uma folha. A **altura do max-heap** é a altura da sua raiz. Como um max-heap de $n$ elementos pode ser visto como uma árvore binária, sua altura é proporcional a $\log_2 n$. Como veremos a seguir, as operações básicas sobre heaps têm tempo de execução proporcional à altura da árvore binária, ou seja, $O(\log_2 n)$.

### 8.1.1 Manutenção da propriedade max-heap

Seja um vetor de números inteiros $S$ com $n > 0$ elementos. Ainda que o vetor $S$ não seja um max-heap, vamos enxergá-lo do mesmo modo como fizemos anteriormente, como uma árvore binária com o último nível preenchido da esquerda para direita. Nesta seção, queremos resolver o seguinte problema:

> Seja um vetor de números inteiros $S$ com $n > 0$ elementos e um índice $i$. Se $S$ é visto como uma árvore binária, estabeleça a propriedade max-heap (8.3) para a sub-árvore de $S$ com raiz $S[i]$, supondo que as sub-árvores esquerda e direita do nó $i$ de $S$ são max-heaps.

A função `desce` recebe um conjunto $S$ de números inteiros com $n > 0$ elementos e um índice $i$. Se enxergamos $S$ como uma árvore binária com seu último nível preenchido da esquerda para direita, então a função `desce` verifica a propriedade max-heap para a sub-árvore de $S$ com raiz $S[i]$, dado que as sub-árvores com raiz $S[\,\texttt{esquerdo(}i\texttt{)}\,]$ e $S[\,\texttt{direito(}i\texttt{)}\,]$ em $S$ são max-heaps. Se a propriedade não é satisfeita para $S[i]$, a função `desce` troca $S[i]$ com o filho que possui maior prioridade. Assim, a propriedade max-heap é estabelecida para o nó de índice $i$. No entanto, essa troca pode ocasionar a violação da propriedade max-heap para um dos filhos do nó de índice $i$. Então, esse processo é repetido recursivamente até que a propriedade max-heap não seja mais violada. Dessa forma, a função `desce` rearranja $S$ adequadamente de maneira que a sub-árvore com raiz $S[i]$ torne-se um max-heap, "descendo" o elemento $S[i]$ para a sua posição correta em $S$.

```
/* Recebe um número inteiro n > 0, um vetor S de números in-
   teiros com n elementos e um índice i e estabelece a pro-
   priedade max-heap para a sub-árvore de S com raiz S[i] */
void desce(int n, int S[MAX], int i)
{
   int e, d, maior;

   e = esquerdo(i);
   d = direito(i);
   if (e < n && S[e] > S[i])
      maior = e;
   else
      maior = i;
   if (d < n && S[d] > S[maior])
      maior = d;
   if (maior != i) {
      troca(&S[i], &S[maior]);
      desce(n, S, maior);
   }
}
```

A figura 8.3 ilustra um exemplo de execução da função **desce**.

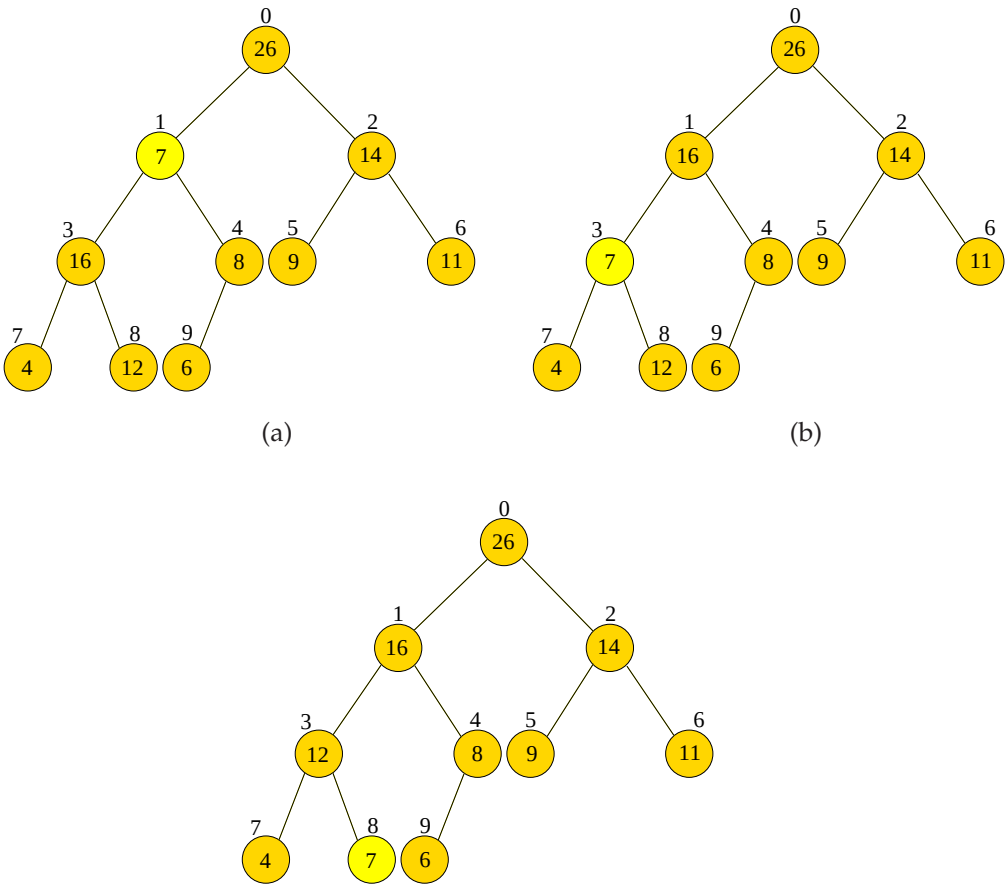



Figura 8.3: Um exemplo de execução da função **desce** com argumentos $(10, S, 1)$.

O tempo de execução de pior caso da função **desce** é proporcional à altura da árvore binária correspondente à $S$, isto é, proporcional à $\log_2 n$.

### 8.1.2 Construção de um max-heap

Seja um vetor $S$ de números inteiros com $n > 0$ elementos. Usando a função **desce**, é fácil transformar o vetor $S$ em um max-heap. Se enxergamos $S$ como uma árvore binária, basta então percorrer as sub-árvores de $S$ com alturas crescentes, usando a função **desce** para garantir a propriedade max-heap para cada uma de suas raízes. Podemos excluir as folhas de $S$, já que toda árvore com altura 0 é um max-heap. A função **constroi_max_heap** é apresentada a seguir.

```
/* Recebe um número inteiro n > 0 e um vetor de números
   inteiros S com n elementos e rearranja o vetor S de
   modo que o novo vetor S possua a propriedade max-heap */
void constroi_max_heap(int n, int S[MAX])
{
   int i;

   for (i = n/2 - 1; i >= 0; i--)
      desce(n, S, i);

}
```

Para mostrar que a função **constroi_max_heap** está correta, temos de usar o seguinte invariante:

> No início de cada iteração da estrutura de repetição da função, cada nó $i+1, i+2, \ldots, n-1$ é raiz de um max-heap.

Dado o invariante acima, é fácil mostrar que a função **constroi_max_heap** está correta, usando indução matemática.

O tempo de execução da função **constroi_max_heap** é calculado observando que cada chamada da função **desce** na estrutura de repetição da função **constroi_max_heap** tem tempo de execução proporcional a $\log_2 n$, onde $n$ é o número de elementos no vetor $S$. Devemos notar ainda que um número proporcional a $n$ chamadas à função **desce** são realizadas nessa estrutura de repetição. Assim, o tempo de execução da função **constroi_max_heap** é proporcional a $n \log_2 n$. Apesar de correto, esse limitante superior para o tempo de execução da função **constroi_max_heap** não é muito justo. Isso significa que é correto afirmar que a função $T(n)$ que descreve o tempo da função é limitada superiormente pela função $cn \log_2 n$ para todo $n \geqslant n_0$, onde $c$ e $n_0$ são constantes positivas. Porém, existe outra função, que também limita superiormente a função $T(n)$, que é "menor" que $n \log_2 n$. Podemos mostrar que um limitante superior melhor pode ser obtido observando a variação nas alturas dos nós da árvore max-heap nas chamadas da função **desce**. Essa análise mais apurada fornece um tempo de execução de pior caso proporcional a $n$ para a função **constroi_max_heap**. Mais detalhes sobre essa análise podem ser encontrados no livro de Cormen et. al [1].

A figura 8.4 ilustra um exemplo de execução da função **constroi_max_heap**.

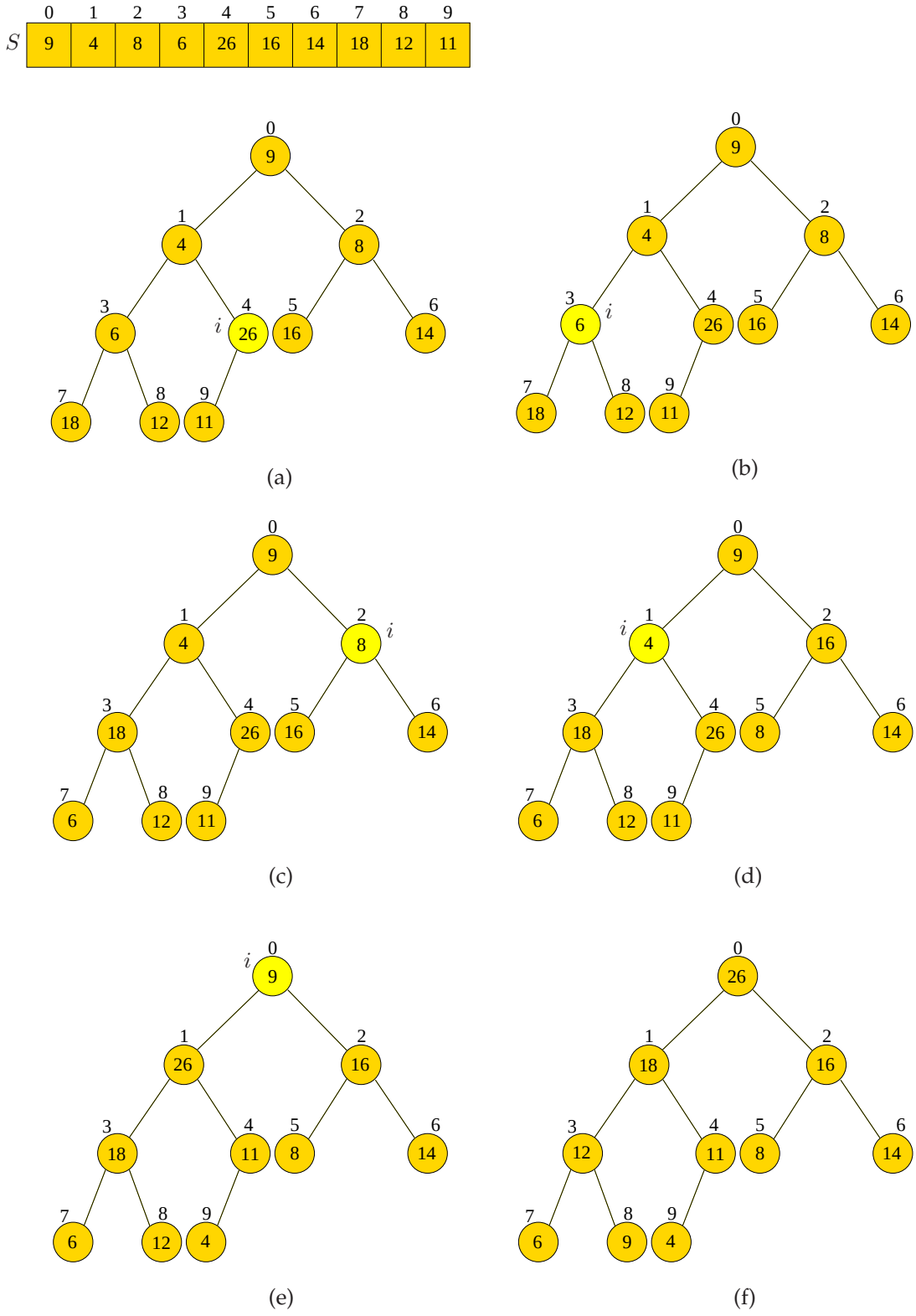



Figura 8.4: Exemplo de execução da função `constroi_max_heap` tendo como entrada o vetor $S = \{9, 4, 8, 6, 26, 16, 14, 18, 12, 11\}$.

Funções muito semelhantes às funções `desce` e `constroi_max_heap` podem ser construídas para obtermos um min-heap. Veja o exercício 8.7.

### 8.1.3 Alteração de uma prioridade em um max-heap

Vamos retomar o vetor $S$ de números inteiros com $n > 0$ elementos. Suponha que $S$ seja um max-heap. Suponha, no entanto, que a prioridade $S[i]$ seja modificada. Uma operação como esta pode resultar em um novo vetor que deixa de satisfazer a propriedade max-heap justamente na posição $i$ de $S$. A figura 8.5 mostra os três casos possíveis quando uma alteração de prioridade é realizada em um max-heap $S$.

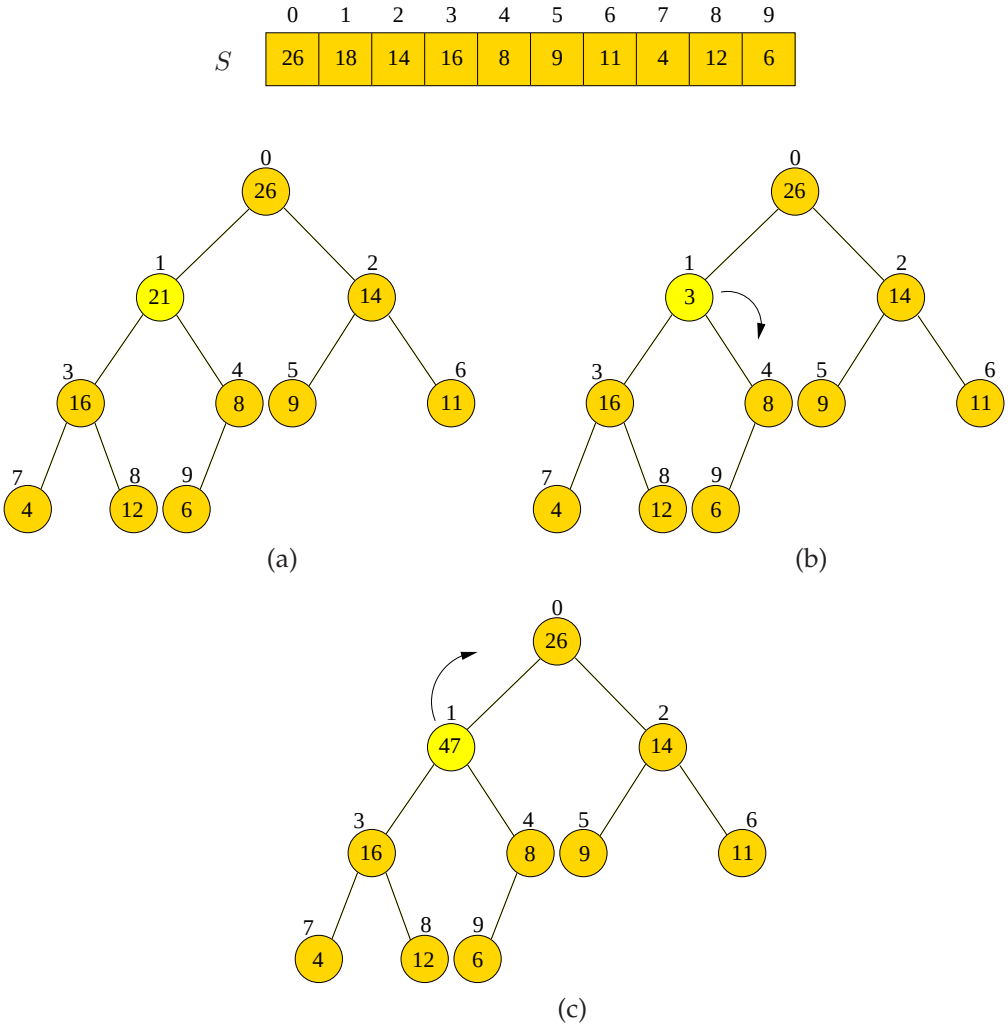



Figura 8.5: Alteração na prioridade $S[1]$ de um max-heap. (a) Alteração mantém a propriedade max-heap. (b) Alteração viola a propriedade max-heap e a prioridade deve "descer". (c) Alteração viola a propriedade max-heap e a prioridade deve "subir".

Como podemos perceber na figura 8.5(a), se a alteração mantém a propriedade max-heap, então não é necessário fazer nada. Observe também que se a prioridade alterada tem valor menor que de pelo menos um de seus filhos, então essa nova prioridade deve "descer" na árvore. Esse caso é mostrado na figura 8.5(b) e é resolvido facilmente usando a função `desce`

descrita na seção 8.1.1. Por fim, se alteração em um nó resulta em uma prioridade maior que a prioridade do seu nó pai, então essa nova prioridade deve "subir" na árvore, como mostra a figura 8.5(c). Nesse caso, precisamos de uma função eficiente que resolva esse problema. A função **sobe** apresentada a seguir soluciona esse problema.

```
/* Recebe um número inteiro n > 0, um vetor S de nú-
   meros inteiros com n elementos e um índice i e es-
   tabelece a propriedade max-heap para a árvore S */
void sobe(int n, int S[MAX], int i)
{

   while (S[pai(i)] < S[i]) {
      troca(&S[i], &S[pai(i)]);
      i = pai(i);
   }

}
```

A figura 8.6 ilustra um exemplo de execução da função **sobe**.

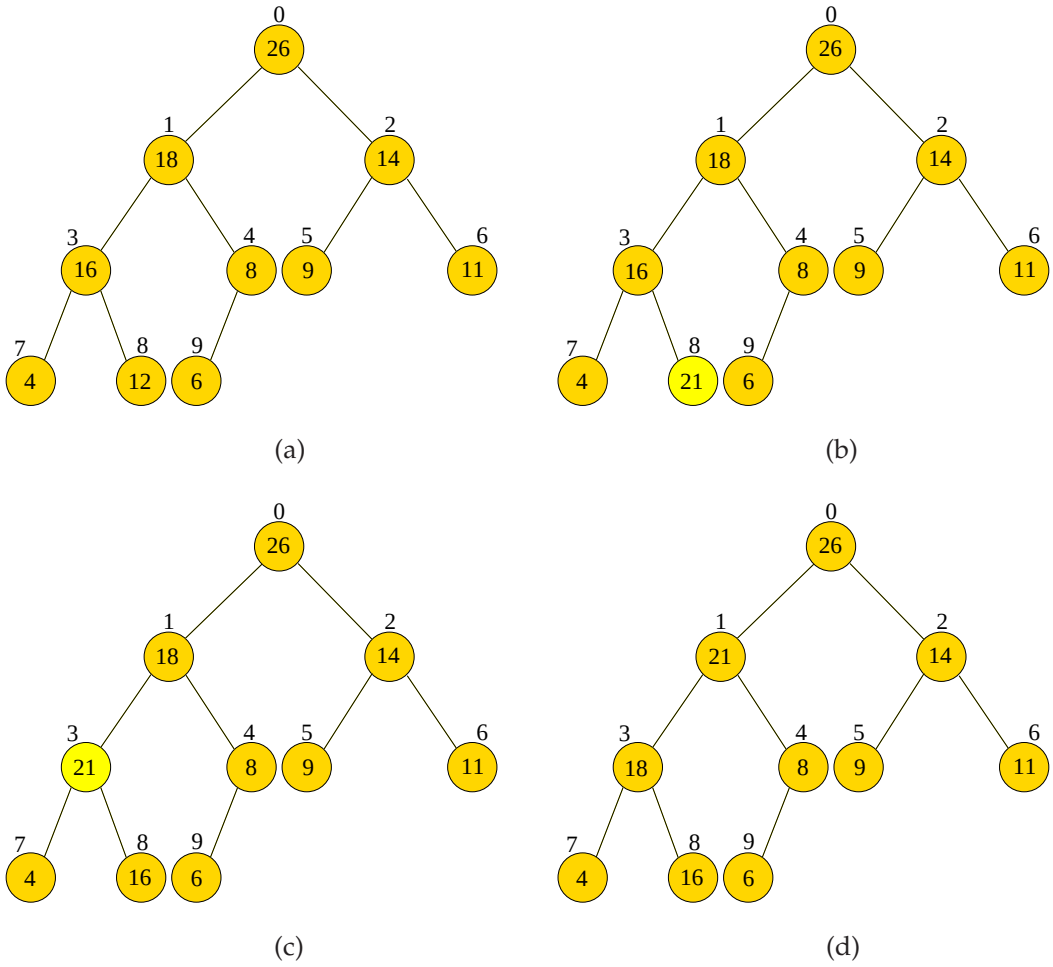



Figura 8.6: Um exemplo de execução da função **sobe** com argumentos $(10, S, 8)$.

O tempo de execução de pior caso da função `sobe` é proporcional à altura da árvore binária correspondente à $S$, isto é, proporcional à $\log_2 n$.

## 8.2 Listas de prioridades

Uma lista de prioridades é uma estrutura de dados que ocorre muito freqüentemente em diversas aplicações. Por exemplo, no escalonamento de tarefas em um computador ou em um simulador de eventos, as listas de prioridades são empregadas com muito sucesso. No primeiro exemplo, devemos manter uma lista dos processos a serem executados pelo computador com suas respectivas prioridades, selecionando sempre aquele de mais alta prioridade assim que um processador se torna ocioso ou disponível. No segundo exemplo, uma lista de eventos a serem simulados são colocados em uma lista de prioridades, onde seus tempos de ocorrência são as prioridades na lista, e os eventos devem ser simulados na ordem de seus tempos de ocorrência.

Do mesmo modo como com os heaps, existem dois tipos de listas de prioridades: listas de max-prioridades e listas de min-prioridades. Os exemplos que acabamos de descrever são associados a uma lista de max-prioridades e uma lista de min-prioridades, respectivamente. Assim como antes, focaremos atenção nas listas de max-prioridades, deixando as outras para os exercícios.

Uma **lista de max-prioridades** é uma estrutura de dados para manutenção de um conjunto de elementos $S$, onde a cada elemento está associada uma prioridade. As seguintes operações são associadas a uma lista de max-prioridades:

(1) inserção de um elemento no conjunto $S$;

(2) consulta da maior prioridade em $S$;

(3) remoção do elemento de maior prioridade em $S$;

(4) aumento da prioridade de um elemento de $S$.

É fácil ver que um max-heap pode ser usado para implementar uma lista de max-prioridades. A função `consulta_maxima` implementa a operação (2) em tempo de execução constante, isto é, $O(1)$.

```
/* Recebe uma lista de max-prioridades S e devolve a maior prioridade em S */
int consulta_maxima(int S[MAX])
{
   return S[0];
}
```

A função `extrai_maxima` implementa a operação (3) usando a função `desce`, devolvendo também o valor de maior prioridade na lista de max-prioridades. Se a lista é vazia, a função devolve um valor especial para indicar que a remoção não ocorreu.

```
/* Recebe um número inteiro n > 0 e uma lista de max-priorida-
   des S e remove e devolve o valor da maior prioridade em S */
int extrai_maxima(int *n, int S[MAX])
{
   int maior;

   if (*n > 0) {
      maior = S[0];
      S[0] = S[*n - 1];
      *n = *n - 1;
      desce(*n, S, 0);
      return maior;
   }
   else
      return -∞;
}
```

O tempo de execução da função **`extrai_maxima`** é na verdade o tempo gasto pela função **`desce`**. Portanto, seu tempo de execução é proporcional a $\log_2 n$.

A função **`aumenta_prioridade`** implementa a operação (4), recebendo uma lista de max-prioridades, uma nova prioridade e um índice e devolve a lista de max-prioridades com a prioridade alterada.

```
/* Recebe um número inteiro n > 0, uma lista de max-priorida-
   des S, um índice i e uma prioridade p e devolve a lista de
   max-prioridades com a prioridade na posição i modificada */
void aumenta_prioridade(int n, int S[MAX], int i, int p)
{
   if (p < S[i])
      printf("ERRO: nova prioridade é menor que da célula\n");
   else {
      S[i] = p;
      sobe(n, S, i);
   }
}
```

O tempo de execução da função **`aumenta_prioridade`** é o tempo gasto pela chamada à função **`sobe`** e, portanto, é proporcional a $\log_2 n$.

Por fim, a operação (1) é implementada pela função **`insere_lista`** que recebe uma lista de max-prioridades e uma nova prioridade e insere essa prioridade na lista de max-prioridades.

```
/* Recebe um número inteiro n > 0, uma lista de max-prioridades S e uma prio-
   ridade p e devolve a lista de max-prioridades com a nova prioridade */
void insere_lista(int *n, int S[MAX], int p)
{
   S[*n] = p;
   *n = *n + 1;
   sobe(*n, S, *n - 1);
}
```

O tempo de execução da função **insere_lista** é o tempo gasto pela chamada à função **sobe** e, portanto, é proporcional a $\log_2 n$.

## 8.3 Ordenação usando um max-heap

Um max-heap pode ser naturalmente usado para descrever um algoritmo de ordenação eficiente. Esse algoritmo é conhecido como *heapsort* e tem o mesmo tempo de execução de pior caso da ordenação por intercalação e do caso médio da ordenação por separação.

```
/* Recebe um número inteiro n > 0 e um vetor S de números inteiros com
   n elementos e rearranja S em ordem crescente usando um max-heap */
void heapsort(int n, int S[MAX])
{
   int i;

   constroi_max_heap(n, S);
   for (i = n - 1; i > 0; i--) {
      troca(&S[0], &S[i]);
      n--;
      desce(n, S, 0);
   }
}
```

Podemos mostrar que a função **heapsort** está correta usando o seguinte invariante do processo iterativo:

> No início de cada iteração da estrutura de repetição da função **heapsort**, o vetor $S[0..i]$ é um max-heap contendo os $i$ menores elementos de $S[0..n-1]$ e o vetor $S[i+1..n-1]$ contém os $n-i$ maiores elementos de $S[0..n-1]$ em ordem crescente.

O tempo de execução da função **heapsort** é proporcional a $n \log_2 n$. Note que a chamada da função **constroi_max_heap** gasta tempo proporcional a $n$ e cada uma das $n-1$ chamadas à função **desce** gasta tempo proporcional a $\log_2 n$.

## Exercícios

Os exercícios foram extraídos do livro de Cormen et. al [1].

8.1 A seqüência $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12\rangle$ é um max-heap?

8.2 Qual são os números mínimo e máximo de elementos em um max-heap de altura $h$?

8.3 Mostre que em qualquer sub-árvore de um max-heap, a raiz da sub-árvore contém a maior prioridade de todas as que ocorrem naquela sub-árvore.

8.4 Em um max-heap, onde pode estar armazenado o elemento de menor prioridade, considerando que todos os elementos são distintos?

8.5 Um vetor em ordem crescente é um min-heap?

8.6 Use como base a figura 8.3 e ilustre a execução da função `desce(14, S, 2)` sobre o vetor $S = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.

8.7 Suponha que você deseja manter um min-heap. Escreva uma função equivalente à função `desce` para um max-heap, que mantém a propriedade min-heap (8.2).

8.8 Qual o efeito de chamar `desce(`$n$`, S, i)` quando a prioridade $S[i]$ é maior que as prioridades de seus filhos?

8.9 Qual o efeito de chamar `desce(`$n$`, S, i)` para $i \geqslant n/2$?

8.10 O código da função `desce` é muito eficiente em termos de fatores constantes, exceto possivelmente pela chamada recursiva que pode fazer com que alguns compiladores produzam um código ineficiente. Escreva uma função não-recursiva eficiente equivalente à função `desce`.

8.11 Use como base a figura 8.4 e ilustre a operação da função `constroi_max_heap` sobre o vetor $S = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

8.12 Por que fazemos com que a estrutura de repetição da função `constroi_max_heap` controlada por $i$ seja decrescente de $n/2 - 1$ até 0 ao invés de crescente de 0 até $n/2 - 1$?

8.13 Use como exemplo a figura 8.3 e ilustre a operação da função `extrai_maximo` sobre o vetor $S = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

8.14 Ilustre a operação da função `insere_lista(12, S, 9)` sobre a lista de prioridades $S = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

8.15 Escreva códigos eficientes e corretos para as funções que implementam as operações `consulta_minimo`, `extrai_minimo`, `diminui_prioridade` e `insere_lista_min`. Essas funções devem implementar uma lista de min-prioridades com um min-heap.

8.16 A operação `remove_lista(&`$n$`, S, i)` remove a prioridade do nó $i$ de uma lista de max-prioridades. Escreva uma função eficiente para `remove_lista`.

8.17 Ilustre a execução da função `heapsort` sobre o vetor $S = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

AULA 9

# INTRODUÇÃO AOS PONTEIROS

Ponteiros ou apontadores, do inglês *pointers*, são certamente uma das características mais destacáveis da linguagem de programação C. Os ponteiros agregam poder e flexibilidade à linguagem de maneira a diferenciá-la de outras linguagens de programação de alto nível, permitindo a representação de estruturas de dados complexas, a modificação de valores passados como argumentos a funções, alocação dinâmica de espaços na memória, entre outros destaques. Nesta aula iniciaremos o contato com esses elementos.

Esta aula é baseada especialmente nas referências [2, 7].

## 9.1 Variáveis ponteiros

Para bem compreender os ponteiros, precisamos inicialmente compreender a idéia de indireção. Conceitualmente, um ponteiro permite acesso indireto a um valor armazenado em algum ponto da memória. Esse acesso é realizado justamente porque um **ponteiro** é uma variável que armazena um valor especial, que é um endereço de memória, e por isso nos permite acessar indiretamente o valor armazenado nesse endereço.

A memória de um computador pode ser vista como constituída de muitas posições (ou compartimentos ou células), dispostas continuamente, cada qual podendo armazenar um valor, na base binária. Ou seja, a memória nada mais é do que um grande vetor que pode armazenar valores na base binária e que, por sua vez, esses valores podem ser interpretados como valores de diversos tipos. Os índices desse grande vetor, numerados seqüencialmente a partir de 0 (zero), são chamados de **endereços de memória**.

Quando escrevemos um programa em uma linguagem de programação de alto nível, o nome ou identificador de uma variável é associado diretamente a um índice desse vetor, isto é, a um endereço da memória. A tradução do nome da variável para um endereço de memória, e vice-versa, é feita de forma automática e transparente pelo compilador da linguagem de programação. Essa é uma característica marcante de uma linguagem de programação de alto nível, que se diferencia das linguagens de programação de baixo nível, já que não há necessidade de um(a) programador(a) preocupar-se com os endereços de memória quando armazena/recupera dados na memória.

Em geral, a memória de um computador é dividida em bytes, com cada byte sendo capaz de armazenar 8 bits de informação. Cada byte tem um único endereço que o distingue de outros bytes da memória. Se existem $n$ bytes na memória, podemos pensar nos endereços como números de intervalo de 0 a $n - 1$, como mostra a figura 9.1.

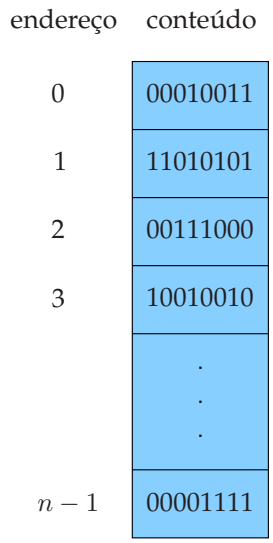



Figura 9.1: Uma ilustração da memória e de seus endereços.

Um programa executável é constituído por trechos de código e dados, ou seja, por instruções de máquina que correspondem às sentenças no programa original na linguagem C e também de variáveis. Cada variável do programa ocupa um ou mais bytes na memória. O endereço do primeiro byte de uma variável é dito ser o endereço da variável. Na figura 9.2, a variável $i$ ocupa os bytes dos endereços 2000 e 2001. Logo, o endereço da variável $i$ é 2000.

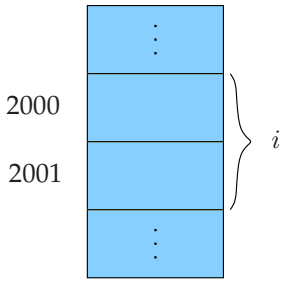



Figura 9.2: Endereço da variável $i$.

Os ponteiros têm um papel importante em toda essa história. Apesar de os endereços serem representados por números, como ilustrado nas figuras 9.1 e 9.2, o intervalo de valores que esses objetos podem assumir é diferente do intervalo que os números inteiros, por exemplo, podem assumir. Isso significa, entre outras coisas, que não podemos armazenar endereços em variáveis do tipo inteiro. Endereços são armazenados em variáveis especiais, chamadas de variáveis ponteiros.

Quando armazenamos o endereço de uma variável $i$ em uma variável ponteiro $p$, dizemos que $p$ **aponta para** $i$. Em outras palavras, um ponteiro nada mais é que um endereço e uma variável ponteiro é uma variável que pode armazenar endereços.

Ao invés de mostrar endereços como números, usaremos uma notação simplificada de tal forma que, para indicar que uma variável ponteiro $p$ armazena o endereço de uma variável $i$, mostraremos o conteúdo de $p$ – um endereço – como uma flecha orientada na direção de $i$, como mostra a figura 9.3.

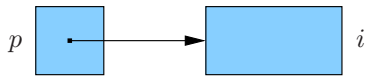



Figura 9.3: Representação de uma variável ponteiro.

Uma variável ponteiro pode ser declarada da mesma maneira que uma variável qualquer de qualquer tipo, como sempre temos feito, mas com um asterisco precedendo seu identificador. Por exemplo,

```
int *p;
```

Essa declaração indica que $p$ é uma variável ponteiro capaz de apontar para objetos do tipo `int`. A linguagem C obriga que toda variável ponteiro aponte apenas para objetos de um tipo particular, chamado de **tipo referenciado**.

Diante do exposto até este ponto, daqui para frente não mais distinguiremos os termos 'ponteiro' e 'variável ponteiro', ficando então subentendido o seu valor (conteúdo) e a própria variável.

## 9.2 Operadores de endereçamento e de indireção

A linguagem C possui dois operadores para uso específico com ponteiros. Para obter o endereço de uma variável, usamos o **operador de endereçamento (ou de endereço) &**. Se $v$ é uma variável, então `&`$v$ é seu endereço na memória. Para ter acesso ao objeto que um ponteiro aponta, temos de usar o **operador de indireção** `*`. Se $p$ é um ponteiro, então `*`$p$ representa o objeto para o qual $p$ aponta no momento.

A declaração de uma variável ponteiro reserva um espaço na memória para um ponteiro mas não a faz apontar para um objeto. Assim, é crucial inicializar um ponteiro antes de usá-lo. Uma forma de inicializar um ponteiro é atribuir-lhe o endereço de alguma variável usando o operador `&`:

```
int i, *p;

p = &i;
```

Atribuir o endereço da variável $i$ para a variável $p$ faz com que $p$ aponte para $i$, como ilustra a figura 9.4.

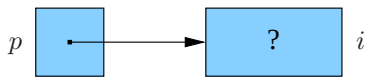



Figura 9.4: Variável ponteiro $p$ contendo o endereço da variável $i$.

É possível inicializar uma variável ponteiro no momento de sua declaração, como abaixo:

```
int i, *p = &i;
```

Uma vez que uma variável ponteiro aponta para um objeto, podemos usar o operador de indireção `*` para acessar o valor armazenado no objeto. Se $p$ aponta para $i$, por exemplo, podemos imprimir o valor de $i$ de forma indireta, como segue:

```
printf("%d\n", *p);
```

Observe que a função `printf` mostrará o valor de $i$ e não o seu endereço. Observe também que aplicar o operador `&` a uma variável produz um ponteiro para a variável e aplicar o operador `*` para um ponteiro retoma o valor original da variável:

```
j = *&i;
```

Na verdade, a atribuição acima é idêntica à seguinte:

```
j = i;
```

Enquanto dizemos que $p$ **aponta para** $i$, dizemos também que `*`$p$ é um **apelido** para $i$. Ademais, não apenas `*`$p$ tem o mesmo valor que $i$, mas alterar o valor de `*`$p$ altera também o valor de $i$.

Uma observação importante que auxilia a escrever e ler programas com variáveis ponteiros é sempre "traduzir" os operadores unários de endereço `&` e de indireção `*` para *endereço da variável* e *conteúdo da variável apontada por*, respectivamente. Sempre que usamos um desses operadores no sentido de estabelecer indireção e apontar para valores, é importante traduzi-los desta forma para fins de clareza.

Observe que no programa 9.1, após a declaração das variáveis $c$ e $p$, temos a inicialização do ponteiro $p$, que recebe o endereço da variável $c$, sendo essas duas variáveis do mesmo tipo `char`. Também ocorre a inicialização da variável $c$. É importante sempre destacar que o valor, ou conteúdo, de um ponteiro na linguagem C não tem significado até que contenha, ou aponte, para algum endereço válido.

A primeira chamada da função `printf` no programa 9.1 mostra o endereço onde a variável $c$ se localiza na memória e seu conteúdo, inicializado com o caractere `'a'` na linha anterior. Note que um endereço pode ser impresso pela função `printf` usando o especificador de tipo `%p`. Em seguida, um outra chamada à função `printf` é realizada, mostrando o endereço onde a varíavel $p$ se localiza na memória, o conteúdo da variável $p$ e o conteúdo da variável apontada por $p$. Como a variável $p$ aponta para a variável $c$, o valor apresentado na saída é também aquele armazenado na variável $c$, isto é, o caractere `'a'`. Na segunda vez que ocorrem as mesmas chamadas às funções `printf`, elas são precedidas pela alteração do conteúdo da variável $c$ e, como a variável $p$ mantém-se apontando para a variável $c$, o caractere `'/'` seja

Programa 9.1: Um exemplo do uso de ponteiros.

```
#include <stdio.h>

int main(void)
{
   char c, *p;

   p = &c;
   c = 'a';
   printf("&c = %p    c = %c\n", &c, c);
   printf("&p = %p    p = %p    *p = %c\n\n", &p, p, *p);

   c = '/';
   printf("&c = %p    c = %c\n", &c, c);
   printf("&p = %p    p = %p    *p = %c\n\n", &p, p, *p);

   *p = 'Z';
   printf("&c = %p    c = %c\n", &c, c);
   printf("&p = %p    p = %p    *p = %c\n\n", &p, p, *p);

   return 0;
}
```

apresentado na saída quando solicitamos a impressão de `*p`. É importante notar que, a menos que o conteúdo da variável $p$ seja modificado, a expressão `*p` sempre acessa o conteúdo da variável $c$. Por fim, o último conjunto de chamadas à função `printf` é precedido de uma atribuição que modifica o conteúdo da variável apontada por $p$ e conseqüentemente da variável $c$. Ou seja, a atribuição a seguir:

```
*p = 'Z';
```

faz também com que a variável $c$ receba o caractere `'Z'`.

A execução do programa 9.1 em um computador com processador de 64 bits tem o seguinte resultado na saída:

```
&c = 0x7fffffffc76f    c = a
&p = 0x7fffffffc760    p = 0x7fffffffc76f    *p = a

&c = 0x7fffffffc76f    c = /
&p = 0x7fffffffc760    p = 0x7fffffffc76f    *p = /

&c = 0x7fffffffc76f    c = Z
&p = 0x7fffffffc760    p = 0x7fffffffc76f    *p = Z
```

Observe que um endereço de memória é impresso na base hexadecimal com o especificador de tipo `%p`. A figura 9.5 ilustra o início da execução do programa 9.1.

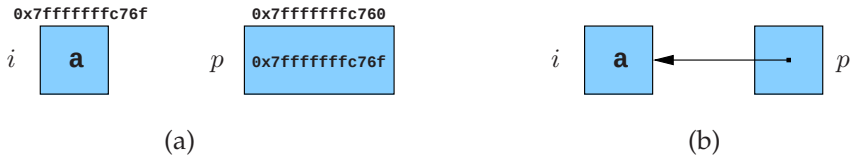



Figura 9.5: Ilustração das variáveis do programa 9.1 após as inicializações das variáveis. (a) Representação com endereços. (b) Representação esquemática.

A linguagem C permite ainda que o operador de atribuição copie ponteiros, supondo que possuam o mesmo tipo. Suponha que a seguinte declaração tenha sido feita:

```
int i, j, *p, *q;
```

Então, a sentença:

```
p = &i;
```

é um exemplo de atribuição de um ponteiro, onde o endereço de $i$ é copiado em $p$. Um outro exemplo de atribuição de ponteiro é dado a seguir:

```
q = p;
```

Essa sentença copia o conteúdo de $p$, o endereço de $i$, para $q$, fazendo com que $q$ aponte para o mesmo lugar que $p$ aponta, como podemos visualizar na figura 9.6.

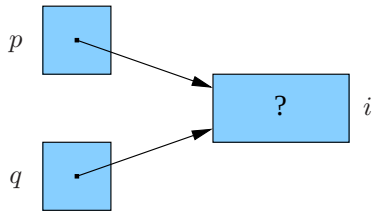



Figura 9.6: Dois ponteiros para um mesmo endereço.

Ambos os ponteiros $p$ e $q$ apontam para $i$ e, assim, podemos modificar o conteúdo de $i$ indiretamente através da atribuição de valores para `*`$p$ e `*`$q$.

## 9.3 Ponteiros em expressões

Ponteiros podem ser usados em expressões aritméticas de mesmo tipo que seus tipos referenciados. Por exemplo, um ponteiro para uma variável do tipo inteiro pode ser usado em uma

expressão aritmética envolvendo números do tipo inteiro, supondo o uso correto do operador de indireção `*`.

É importante observar também que os operadores `&` e `*`, por serem operadores unários, têm precedência sobre os operadores binários das expressões aritméticas em que se envolvem. Por exemplo, em uma expressão aritmética envolvendo números do tipo inteiro, os operadores binários `+`, `-`, `*` e `/` têm menor prioridade que o operador unário de indireção `*`.

Vejamos a seguir, como um exemplo simples, o programa 9.2 que usa um ponteiro como operando em uma expressão aritmética.

Programa 9.2: Outro exemplo do uso de ponteiros.

```
#include <stdio.h>

int main(void)
{
   int i, j, *ptr1, *ptr2;

   ptr1 = &i;
   i = 5;
   j = 2 * *ptr1 + 3;
   ptr2 = ptr1;

   printf("i = %d, &i = %p\n\n", i, &i);
   printf("j = %d, &j = %p\n\n", j, &j);
   printf("&ptr1 = %p, ptr1 = %p, *ptr1 = %d\n", &ptr1, ptr1, *ptr1);
   printf("&ptr2 = %p, ptr2 = %p, *ptr2 = %d\n\n", &ptr2, ptr2, *ptr2);

   return 0;
}
```

A execução do programa 9.2 tem a seguinte saída:

```
i =  5, &i = 0x7fffffffc55c
j = 13, &j = 0x7fffffffc558

&ptr1 = 0x7fffffffc550, ptr1 = 0x7fffffffc55c, *ptr1 = 5
&ptr2 = 0x7fffffffc548, ptr2 = 0x7fffffffc55c, *ptr2 = 5
```

## Exercícios

9.1 Se $i$ é uma variável e $p$ é uma variável ponteiro que aponta para $i$, quais das seguintes expressões são apelidos para $i$?

(a) `*`$p$

(b) `&`$p$

(c) `*&`$p$

(d) `&*`$p$

(e) `*`$i$

(f) `&`$i$

(g) `*&`$i$

(h) `&*`$i$

9.2 Se $i$ é uma variável do tipo `int` e $p$ e $q$ são ponteiros para `int`, quais das seguintes atribuições são corretas?

(a) `` $p$ `=` $i$`;` ``

(b) `*`$p$ `= &`$i$`;`

(c) `&`$p$ `=` $q$`;`

(d) $p$ `= &`$q$`;`

(e) $p$ `= *&`$q$`;`

(f) $p$ `=` $q$`;`

(g) $p$ `= *`$q$`;`

(h) `*`$p$ `=` $q$`;`

(i) `*`$p$ `= *`$q$`;`

9.3 Entenda o que o programa 9.3 faz, simulando sua execução passo a passo. Depois disso, implemente-o.

Programa 9.3: Programa do exercício 9.3.

```
#include <stdio.h>

int main(void)
{
   int a, b, *ptr1, *ptr2;

   ptr1 = &a;
   ptr2 = &b;
   a = 1;
   (*ptr1)++;
   b = a + *ptr1;
   *ptr2 = *ptr1 * *ptr2;

   printf("a=%d, b=%d, *ptr1=%d, *ptr2=%d\n", a, b, *ptr1, *ptr2);

   return 0;
}
```

9.4 Entenda o que o programa 9.4 faz, simulando sua execução passo a passo. Depois disso, implemente-o.

9.5 Entenda o que o programa 9.5 faz, simulando sua execução passo a passo. Depois disso, implemente-o.

Programa 9.4: Programa do exercício 9.4.

```
#include <stdio.h>

int main(void)
{
   int a, b, c, *ptr;

   a = 3;
   b = 7;
   printf("a=%d, b=%d\n", a, b);

   ptr = &a;
   c = *ptr;
   ptr = &b;
   a = *ptr;
   ptr = &c;
   b = *ptr;
   printf("a=%d, b=%d\n", a, b);

   return 0;
}
```

Programa 9.5: Programa do exercício 9.5.

```
#include <stdio.h>

int main(void)
{
   int i, j, *p, *q;

   p = &i;
   q = p;
   *p = 1;
   printf("i=%d, *p=%d, *q=%d\n", i, *p, *q);

   q = &j;
   i = 6;
   *q = *p;
   printf("i=%d, j=%d, *p=%d, *q=%d\n", i, j, *p, *q);

   return 0;
}
```

AULA 10

# PROGRAMAS EXTENSOS

Nesta aula aprenderemos como dividir e distribuir nossas funções em vários arquivos. Esta possibilidade é uma característica importante da linguagem C, permitindo que o(a) programador(a) possa ter sua própria biblioteca de funções e possa usá-la em conjunto com um ou mais programas.

Os programas que escrevemos até o momento são bem simples e, por isso mesmo pequenos, com poucas linhas de código. No entanto, programas pequenos são uma exceção. À medida que os problemas têm maior complexidade, os programas para solucioná-los têm, em geral, proporcionalmente mais linhas de código. Por exemplo, a versão 2.6.25 do núcleo do sistema operacional LINUX, de abril de 2008, tem mais de nove milhões de linhas de código na linguagem C e seria impraticável mantê-las todas no mesmo arquivo[1]. Nesta aula veremos que um programa na linguagem C consiste de vários arquivos-fontes e também de alguns arquivos-cabeçalhos. aprenderemos a dividir nossos programas em múltiplos arquivos.

Esta aula é baseada na referência [7].

## 10.1 Arquivos-fontes

Até a última aula, sempre consideramos que um programa na linguagem C consiste de um único arquivo. Na verdade, um programa pode ser dividido em qualquer número de **arquivos-fontes** que, por convenção, têm a extensão `.c`. Cada arquivo-fonte contém uma parte do programa, em geral definições de funções e variáveis. Um dos arquivos-fontes de um programa deve necessariamente conter uma função `main`, que é o ponto de início do programa. Quando dividimos um programa em arquivos, faz sentido colocar funções relacionadas e variáveis em um mesmo arquivo-fonte. A divisão de um programa em arquivos-fontes múltiplos tem vantagens significativas:

- agrupar funções relacionadas e variáveis em um único arquivo ajuda a deixar clara a estrutura do programa;
- cada arquivo-fonte pode ser compilado separadamente, com uma grande economia de tempo se o programa é grande é é modificado muitas vezes;
- funções são mais facilmente re-usadas em outros programas quando agrupadas em arquivos-fontes separados.

[1] O sistema operacional LINUX, por ser livre e de código aberto, permite que você possa consultar seu código fonte, além de modificá-lo a seu gosto.

## 10.2 Arquivos-cabeçalhos

Quando abordamos, em aulas anteriores, a biblioteca padrão e o pré-processador da linguagem C, estudamos com algum detalhe os arquivos-cabeçalhos. Quando dividimos um programa em vários arquivos-fontes, como uma função definida em um arquivo pode chamar uma outra função definida em outro arquivo? Como dois arquivos podem compartilhar a definição de uma mesma macro ou a definição de um tipo? A diretiva `#include` nos ajuda a responder a essas perguntas, permitindo que essas informações possam ser compartilhadas entre arquivos-fontes.

Muitos programas grandes contêm definições de macros e definições de tipos que necessitam ser compartilhadas por vários arquivos-fontes. Essas definições devem ser mantidas em arquivos-cabeçalhos.

Por exemplo, suponha que estamos escrevendo um programa que usa macros com nomes `LOGIC`, `VERDADEIRO` e `FALSO`. Ao invés de repetir essas macros em cada arquivo-fonte do programa que necessita delas, faz mais sentido colocar as definições em um arquivo-cabeçalho com um nome como `logico.h` tendo as seguintes linhas:

```
#define VERDADEIRO 1
#define FALSO      0
#define LOGIC      int
```

Qualquer arquivo-fonte que necessite dessas definições deve conter simplesmente a linha a seguir:

```
#include "logico.h"
```

Definições de tipos também são comuns em arquivos-cabeçalhos. Por exemplo, ao invés de definir a macro `LOGIC` acima, podemos usar `typedef` para criar um tipo `logic`. Assim, o arquivo `logico.h` terá as seguintes linhas:

```
#define VERDADEIRO 1
#define FALSO      0
typedef logic int;
```

A figura 10.1 mostra um exemplo de dois arquivos-fontes que incluem o arquivo-cabeçalho `logico.h`.

Colocar definições de macros e tipos em um arquivo-cabeçalho tem algumas vantagens. Primeiro, economizamos tempo por não ter de copiar as definições nos arquivos-fontes onde são necessárias. Segundo, o programa torna-se muito mais fácil de modificar, já que a modificação da definição de uma macro ou de um tipo necessita ser feita em um único arquivo-cabeçalho. E terceiro, não temos de nos preocupar com inconsistências em conseqüência de arquivos-fontes contendo definições diferentes da mesma macro ou tipo.

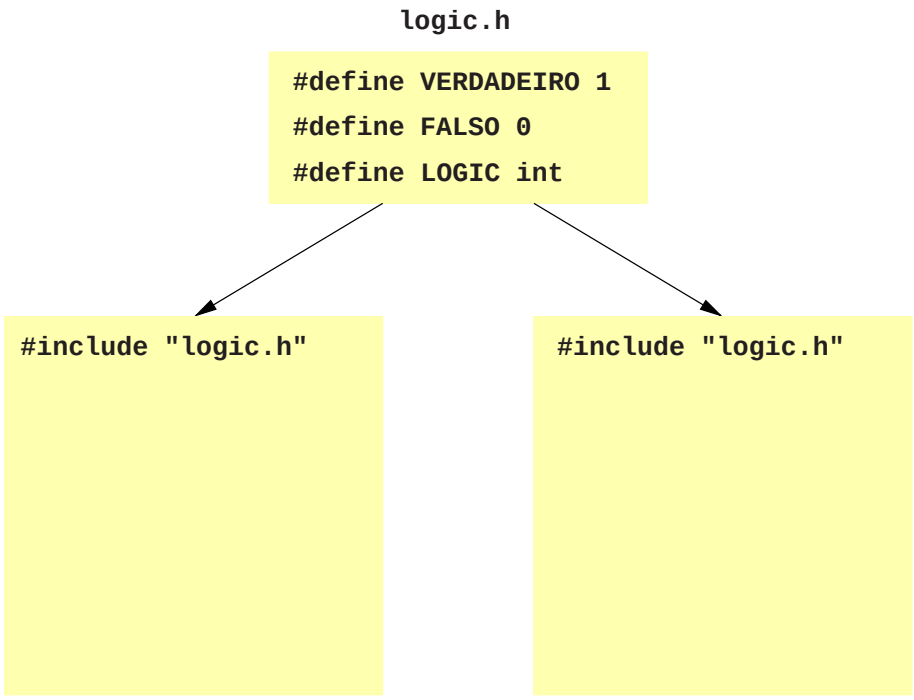



Figura 10.1: Inclusão de um arquivo-cabeçalho em dois arquivos-fontes.

Suponha agora que um arquivo-fonte contém uma chamada a uma função `soma` que está definida em um outro arquivo-fonte, com nome `calculos.c`. Chamar a função `soma` sem sua declaração pode ocasionar erros de execução. Quando chamamos uma função que está definida em outro arquivo, é sempre importante ter certeza que o compilador viu sua declaração, isto é, seu protótipo, antes dessa chamada.

Nosso primeiro impulso é declarar a função `soma` no arquivo-fonte onde ela foi chamada. Isso resolve o problema, mas pode criar imensos problemas de gerenciamento. Suponha que a função é chamada em cinqüenta arquivos-fontes diferentes. Como podemos assegurar que os protótipos das funções `soma` são idênticos em todos esses arquivos? Como podemos garantir que todos esses protótipos correspondem à definição de `soma` em `calculos.c`? Se `soma` deve ser modificada posteriormente, como podemos encontrar todos os arquivos-fontes onde ela é usada? A solução é evidente: coloque o protótipo da função `soma` em um arquivo-cabeçalho e inclua então esse arquivo-cabeçalho em todos os lugares onde `soma` é chamada. Como `soma` é definida em `calculos.c`, um nome natural para esse arquivo-cabeçalho é `calculos.h`. Além de incluir `calculos.h` nos arquivos-fontes onde `soma` é chamada, precisamos incluí-lo em `calculos.c` também, permitindo que o compilador verifique que o protótipo de `soma` em `calculos.h` corresponde à sua definição em `calculos.c`. É regra sempre incluir o arquivo-cabeçalho que declara uma função em um arquivo-fonte que contém a definição dessa função. Não fazê-lo pode ocasionar erros difíceis de encontrar. Se `calculos.c` contém outras funções, muitas delas devem ser declaradas no mesmo arquivo-cabeçalho onde foi declarada a função `soma`. Mesmo porque, as outras funções em `calculos.c` são de alguma forma relacionadas com `soma`. Ou seja, qualquer arquivo que contenha uma chamada à `soma` provavelmente necessita de alguma das outras funções em `calculos.c`. Funções cuja intenção seja usá-las apenas como suporte dentro de `calculos.c` não devem ser declaradas em um arquivo-cabeçalho.

Para ilustrar o uso de protótipos de funções em arquivos-cabeçalhos, vamos supor que queremos manter diversas definições de funções relacionadas a cálculos geométricos em um arquivo-fonte `geometricas.c`. As funções são as seguintes:

```
double perimetroQuadrado(double lado)
{
   return 4 * lado;
}

double perimetroTriangulo(double lado1, double lado2, double lado3)
{
   return lado1 + lado2 + lado3;
}

double perimetroCirculo(double raio)
{
   return 2 * PI * raio;
}

double areaQuadrado(double lado)
{
   return lado * lado;
}

double areaTriangulo(double base, double altura)
{
   return base * altura / 2;
}

double areaCirculo(double raio)
{
   return PI * raio * raio;
}

double volumeCubo(double lado)
{
   return lado * lado * lado;
}

double volumeTetraedro(double lado, double altura)
{
   return (double)1/3 * areaTriangulo(lado, lado * sqrt(altura) / 2) * altura;
}

double volumeEsfera(double raio)
{
   return (double)4/3 * PI * raio * raio;
}
```

Os protótipos dessas funções, além de uma definição de uma macro, serão mantidos em um arquivo-cabeçalho com nome `geometricas.h`:

```
double perimetroQuadrado(double lado);
double perimetroCirculo(double raio);
double perimetroTriangulo(double lado1, double lado2, double lado3);
double areaQuadrado(double lado);
double areaCirculo(double raio);
double areaTriangulo(double base, double altura);
double volumeCubo(double lado);
double volumeEsfera(double raio);
double volumeTetraedro(double lado, double altura);
```

Além desses protótipos, a macro `PI` também deve ser definida neste arquivo. Então, um arquivo-fonte `calc.c` que calcula medidas de figuras geométricas e que contém a função `main` pode ser construído. A figura 10.2 ilustra essa divisão.

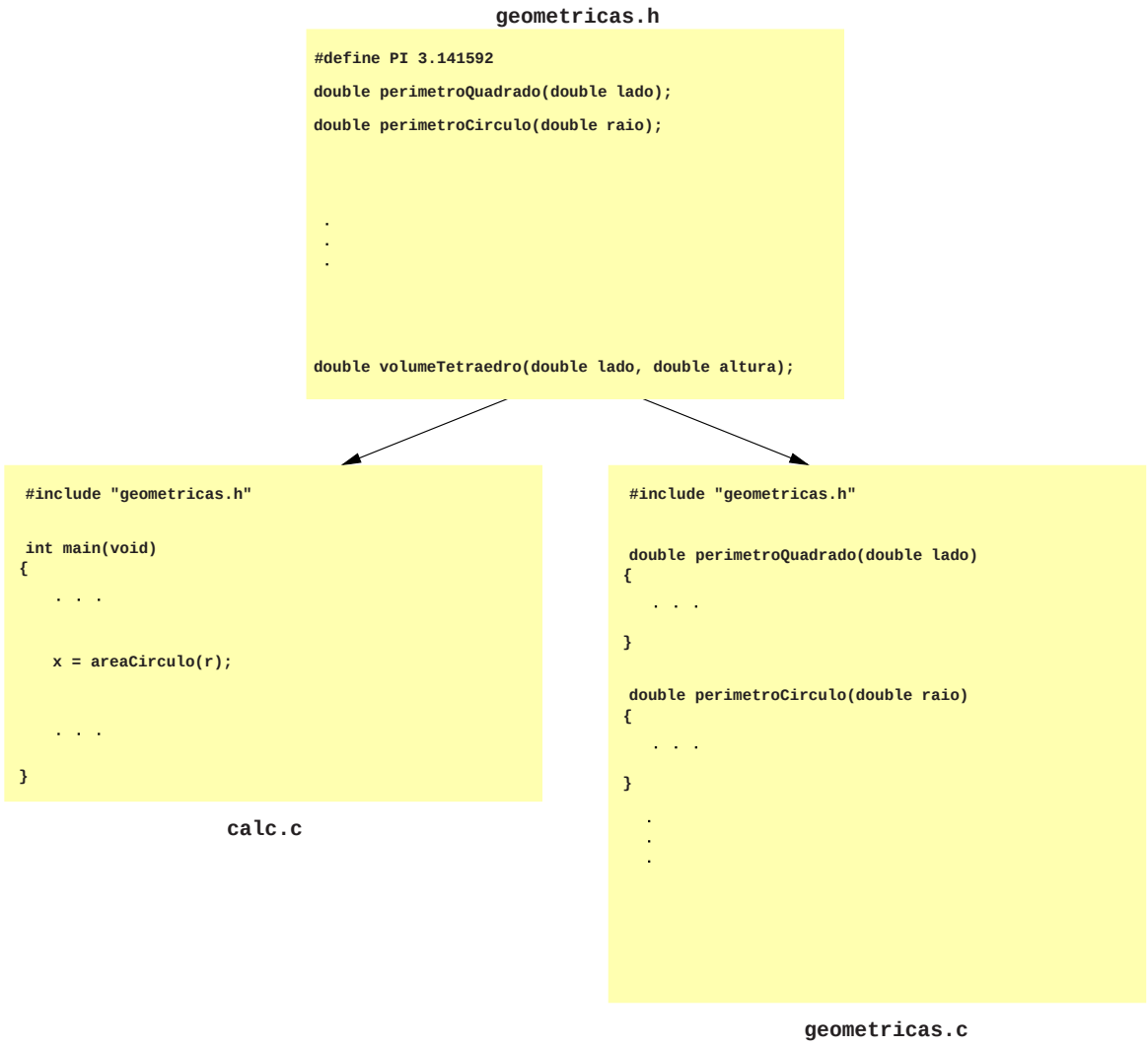



Figura 10.2: Relação entre protótipos, arquivo-fonte e arquivo-cabeçalho.

## 10.3 Divisão de programas em arquivos

Como vimos na seção anterior, podemos dividir nossos programas em diversos arquivos que possuem uma conexão lógica entre si. Usaremos agora o que conhecemos sobre arquivos-cabeçalhos e arquivos-fontes para descrever uma técnica simples de dividir um programa em arquivos. Consideramos aqui que o programa já foi projetado, isto é, já decidimos quais funções o programa necessita e como arranjá-las em grupos logicamente relacionados.

Cada conjunto de funções relacionadas é sempre colocado em um arquivo-fonte em separado. `geometricas.c` da seção anterior é um exemplo de arquivo-fonte desse tipo. Além disso, criamos um arquivo-cabeçalho com o mesmo nome do arquivo-fonte, mas com extensão `.h`. O arquivo-cabeçalho `geometricas.h` da seção anterior é um outro exemplo. Nesse arquivo-cabeçalho, colocamos os protótipos das funções incluídas no arquivo-fonte, lembrando que as funções que são projetadas somente para dar suporte às funções do arquivo-fonte não devem ter seus protótipos descritos no arquivo-cabeçalho. Então, devemos incluir o arquivo-cabeçalho em cada arquivo-fonte que necessite chamar uma função definida no arquivo-fonte. Além disso, incluímos o arquivo-cabeçalho no próprio arquivo-fonte para que o compilador possa verificar que os protótipos das funções no arquivo-cabeçalho são consistentes com as definições do arquivo-fonte. Mais uma vez, esse é o caso do exemplo da seção anterior, com arquivo-fonte `geometricas.c` e arquivo-cabeçalho `geometricas.h`. Veja novamente a figura 10.2.

A função principal `main` deve ser incluída em um arquivo-fonte que tem um nome representando o nome do programa. Por exemplo, se queremos que o programa seja conhecido como `calc` então a função `main` deve estar em um arquivo-fonte com nome `calc.c`. É possível que existam outras funções no mesmo arquivo-fonte onde `main` se encontra, funções essas que não são chamadas em outros arquivos-fontes do programa.

Para gerar um arquivo-executável de um programa dividido em múltiplos arquivos-fontes, os mesmos passos básicos que usamos para um programa em um único arquivo-fonte são necessários:

- **compilação:** cada arquivo-fonte do programa deve ser compilado separadamente; para cada arquivo-fonte, o compilador gera um arquivo contendo código objeto, que têm extensão `.o`;
- **ligação:** o ligador combina os arquivos-objetos criados na fase compilação, juntamente com código das funções da biblioteca, para produzir um arquivo-executável.

Muitos compiladores nos permitem construir um programa em um único passo. Com o compilador GCC, usamos o seguinte comando para construir o programa `calc` da seção anterior:

```
gcc -o calc calc.c geometricas.c
```

Os dois arquivos-fontes são primeiro compilados em arquivos-objetos. Esse arquivos-objetos são automaticamente passados para o ligador que os combina em um único arquivo. A opção `-o` especifica que queremos que nosso arquivo-executável tenha o nome `calc`.

Digitar os nomes de todos os arquivos-fontes na linha de comando de uma janela de um terminal logo torna-se uma tarefa tediosa. Além disso, podemos desperdiçar uma quantidade de tempo quando reconstruímos um programa se sempre recompilamos todos os arquivos-fontes, não apenas aqueles que são afetados pelas nossas modificações mais recentes.

Para facilitar a construção de grandes programas, o conceito de *makefiles* foi proposto nos primórdios da criação do sistema operacional UNIX. Um *makefile* é um arquivo que contém informação necessária para construir um programa. Um *makefile* não apenas lista os arquivos que fazem parte do programa, mas também descreve as dependências entre os arquivos. Por exemplo, da seção anterior, como `calc.c` inclui o arquivo `geometricas.h`, dizemos que `calc.c` 'depende' de `geometricas.h`, já que uma mudança em `geometricas.h` fará com que seja necessária a recompilação de `calc.c`.

Abaixo, listamos um *makefile* para o programa `calc`.

```
calc: calc.o geometricas.o
            gcc -o calc calc.o geometricas.o -lm
calc.o: calc.c geometricas.h
            gcc -c calc.c -lm
geometricas.o: geometricas.c geometricas.h
            gcc -c geometricas.c -lm
```

No arquivo acima, existem 3 grupos de linhas. Cada grupo é conhecido como um **regra**. A primeira linha em cada regra fornece um arquivo-**alvo**, seguido pelos arquivos dos quais ele depende. A segunda linha é um **comando** a ser executado se o alvo deve ser reconstruído devido a uma alteração em um de seus arquivos de dependência.

Na primeira regra, `calc` é o alvo:

```
calc: calc.o geometricas.o
            gcc -o calc calc.o geometricas.o -lm
```

A primeira linha dessa regra estabelece que `calc` depende dos arquivos `calc.o` e `geometricas.o`. Se qualquer um desses dois arquivos foi modificado desde da última construção do programa, então `calc` precisa ser reconstruído. O comando na próxima linha indica como a reconstrução deve ser feita, usando o GCC para ligar os dois arquivos-objetos.

Na segunda regra, `calc.o` é o alvo:

```
calc.o: calc.c geometricas.h
            gcc -c calc.c -lm
```

A primeira linha indica que `calc.o` necessita ser reconstruído se ocorrer uma alteração em `calc.c` ou `geometricas.h`. A próxima linha mostra como atualizar `calc.o` através da recompilação de `calc.c`. A opção `-c` informa o compilador para compilar `calc.c` em um arquivo-objeto, sem ligá-lo.

Tendo criado um *makefile* para um programa, podemos usar o utilitário `make` para construir ou reconstruir o programa. verificando a data e a hora associada com cada arquivo do programa, `make` determina quais arquivos estão desatualizados. Então, ele invoca os comandos necessários para reconstruir o programa.

Algumas dicas para criar *makefiles* seguem abaixo:

- cada comando em um *makefile* deve ser precedido por um caractere de tabulação horizontal `TAB`;
- um *makefile* é armazenado em um arquivo com nome `Makefile`; quando o utilitário `make` é usado, ele automaticamente verifica o conteúdo do diretório atual buscando por esse arquivo;
- use

  ```
  make alvo
  ```

  onde `alvo` é um dos alvos listados no *makefile*; se nenhum alvo é especificado, `make` construirá o alvo da primeira regra.

O utilitário `make` é complicado o suficiente para existirem dezenas de livros e manuais que nos ensinam a usá-lo. Com as informações desta aula, temos as informações básicas necessárias para usá-lo na construção de programas extensos divididos em diversos arquivos. Mais informações sobre o utilitário `make` devem ser buscadas no manual do GNU/Make.

## Exercícios

10.1 (a) Escreva uma função com a seguinte interface:

```
void preenche_aleatorio(int n, int v[MAX])
```

que receba um número inteiro $n$, com $0 < 0 \leqslant 100$, e um vetor $v$ de números inteiros e gere $n$ números inteiros aleatórios armazenando-os em $v$. Use a função `rand` da biblioteca `stdlib`.

(b) Crie um arquivo-fonte com todas as funções de ordenação que vimos nas aulas 5, 6, 7 e 8. Crie também um arquivo-cabeçalho correspondente.

(c) Escreva um programa que receba um inteiro $n$, com $1 \leqslant n \leqslant 10000$, gere um seqüência de $n$ números aleatórios e execute os métodos de ordenação que conhecemos sobre esse vetor, medindo seu tempo de execução. Use as funções `clock` e `difftime` da biblioteca `time`.

(d) Crie um *makefile* para compilar e ligar seu programa.

AULA 11

# PONTEIROS E FUNÇÕES

Nesta aula revemos ponteiros e funções na linguagem C. Até o momento, aprendemos algumas "regras" de como construir funções que têm parâmetros de entrada e saída ou argumentos passados por referência. Esses argumentos/parâmetros, como veremos daqui por diante, são na verdade ponteiros. O endereço de uma variável é passado como argumento para uma função. O parâmetro correspondente que recebe o endereço é então um ponteiro. Qualquer alteração realizada no conteúdo do parâmetro tem reflexos externos à função, no argumento correspondente. Esta aula é baseada especialmente na referência [7].

## 11.1 Parâmetros de entrada e saída?

A abstração que fizemos antes para compreendermos o que são parâmetros de entrada e saída na linguagem C será revista agora e revelará algumas novidades. Veja o programa 11.1.

Programa 11.1: Exemplo de parâmetros de entrada e saída e ponteiros.

```
#include <stdio.h>

/* Recebe dois valores e devolve os mesmos valores com conteúdos trocados */
void troca(int *a, int *b)
{
   int aux;

   aux = *a;
   *a = *b;
   *b = aux;
}

/* Recebe dois valores e troca seus conteúdos */
int main(void)
{
   int x, y;

   scanf("%d%d", &x, &y);
   printf("Antes da troca : x = %d e y = %d\n", x, y);
   troca(&x, &y);
   printf("Depois da troca: x = %d e y = %d\n", x, y);

   return 0;
}
```

Agora que entendemos os conceitos básicos que envolvem os ponteiros, podemos olhar o programa 11.1 e compreender o que está acontecendo, especialmente no que se refere ao uso de ponteiros como argumentos de funções. Suponha que alguém está executando esse programa. A execução inicia na primeira linha da função `main` e seu efeito é ilustrado na figura 11.1.


Figura 11.1: Execução da linha 11.

Em seguida, suponha que o(a) usuário(a) do programa informe dois valores quaisquer do tipo inteiro como, por exemplo, 3 e 8. Isso se reflete na memória como na figura 11.2.


Figura 11.2: Execução da linha 13.

Na linha seguinte da função `main`, a execução da função `printf` permite que o(a) usuário(a) verifique na saída os conteúdos das variáveis que acabou de informar. Na próxima linha do programa, a função `troca` é chamada com os endereços das variáveis `x` e `y` como argumentos. Isto é, esses endereços são copiados nos argumentos correspondentes que compõem a interface da função. O fluxo de execução do programa é então desviado para o trecho de código da função `troca`. Os parâmetros da função `troca` são dois ponteiros para valores do tipo inteiro com identificadores `a` e `b`. Esses dois parâmetros recebem os endereços das variáveis `x` e `y` da função `main`, respectivamente, como se pode observar na figura 11.3.

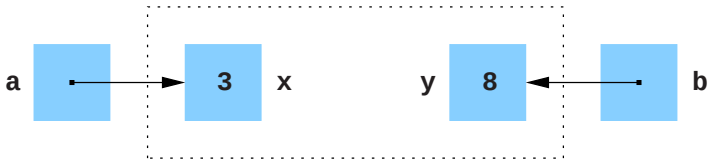



Figura 11.3: Execução da linha 15.

A seguir, na primeira linha do corpo da função `troca`, ocorre a declaração da variável `aux` e um espaço identificado por `aux` é reservado na memória principal, como podemos ver na figura 11.4.

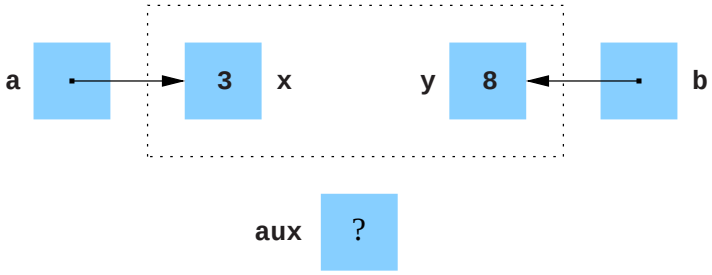



Figura 11.4: Execução da linha 4.

Depois da declaração da variável `aux`, a execução faz com que o *conteúdo da variável apontada por* `a` seja armazenado na variável `aux`. Veja a figura 11.5.

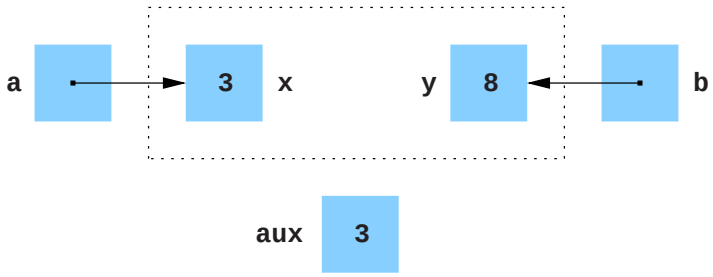



Figura 11.5: Execução da linha 5.

Na execução da linha seguinte, o *conteúdo da variável apontada por* `a` recebe o *conteúdo da variável apontada por* `b`, como mostra a figura 11.6.

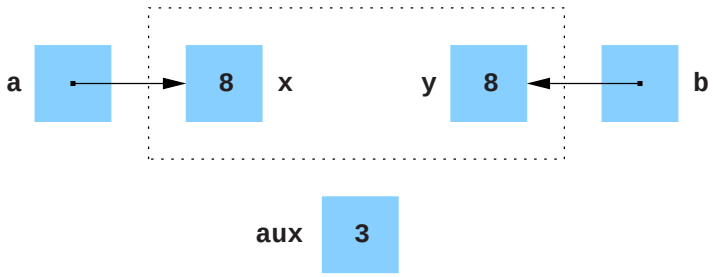



Figura 11.6: Execução da linha 6.

Por fim, na execução da próxima linha, o *conteúdo da variável apontada por* `b` recebe o conteúdo da variável `aux`, conforme a figura 11.7.

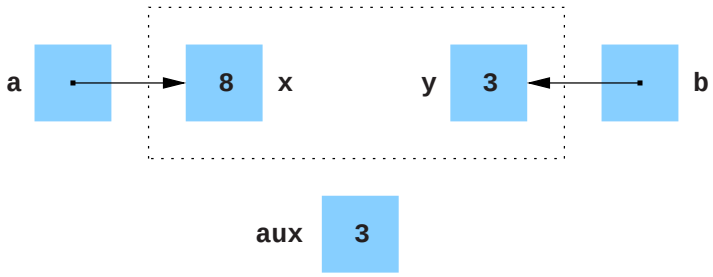



Figura 11.7: Execução da linha 7.

Após o término da função `troca`, seus parâmetros e variáveis locais são destruídos e o fluxo de execução volta para a função `main`, no ponto logo após onde foi feita a chamada da função `troca`. A memória neste momento encontra-se no estado ilustrado na figura 11.8.


Figura 11.8: Estado da memória após o término da função `troca`.

Na linha seguinte da função `main` é a chamada da função `printf`, que mostra os conteúdo das variáveis `x` e `y`, que foram trocados, conforme já constatado e ilustrado na figura 11.8. O programa então salta para a próxima linha e chega ao fim de sua execução.

Esse exemplo destaca que são realizadas cópias do valores dos argumentos – que nesse caso são endereços das variáveis da função `main` – para os parâmetros respectivos da função `troca`. No corpo dessa função, sempre que usamos o operador de indireção para acessar algum valor, estamos na verdade acessando o conteúdo da variável correspondente dentro da função `main`, que chamou a função `troca`. Isso ocorre com as variáveis `x` e `y` da função `main`, quando copiamos seus endereços nos parâmetros `a` e `b` da função `troca`.

Cópia? Como assim cópia? Nós aprendemos que parâmetros passados desta mesma forma são parâmetros de entrada e saída, ou seja, são parâmetros passados por referência e não por cópia. Esse exemplo mostra uma característica muito importante da linguagem C, que ficou dissimulada nas aulas anteriores: *só há passagem de argumentos por cópia na linguagem C*, ou ainda, *não há passagem de argumentos por referência na linguagem C*. O que fazemos de fato é simular a passagem de um argumento por referência usando ponteiros. Assim, passando (por cópia) o endereço de uma variável como argumento para uma função, o parâmetro correspondente deve ser um ponteiro e, mais que isso, um ponteiro para a variável correspondente cujo endereço foi passado como argumento. Dessa forma, qualquer modificação indireta realizada no corpo dessa função usando esse ponteiro será realizada na verdade no conteúdo da variável apontada pelo parâmetro, que é simplesmente o conteúdo da variável passada como argumento na chamada da função.

Não há nada de errado com o que aprendemos nas aulas anteriores sobre argumentos de entrada e saída, isto é, passagem de argumentos por referência. No entanto, vale ressaltar que passagem de argumentos por referência é um tópico conceitual quando falamos da linguagem de programação C. O correto é repetir sempre que *só há passagem de argumentos por cópia na linguagem C*.

## 11.2 Devolução de ponteiros

Além de passar ponteiros como argumentos para funções também podemos fazê-las devolver ponteiros. Funções como essas são comuns, por exemplo, quando tratamos de cadeias de caracteres conforme veremos na aula 14.

A função abaixo recebe dois ponteiros para números inteiros e devolve um ponteiro para um maior dos números inteiros, isto é, devolve o endereço onde se encontra um maior dos números.

```
/* Recebe dois valores e devolve o endereço de um maior */
int *max(int *a, int *b)
{
   if (*a > *b)
      return a;
   else
      return b;
}
```

Quando chamamos a função `max`, passamos ponteiros para duas variáveis do tipo `int` e armazenamos o resultado em uma variável ponteiro:

```
int i, j, *p;
⋮
p = max(&i, &j);
```

Na execução da função `max`, temos que `*a` é um apelido para `i` e `*b` é um apelido para `j`. Se `i` tem valor maior que `j`, então `max` devolve o endereço de `i`. Caso contrário, `max` devolve o endereço de `j`. Depois da chamada, `p` aponta para `i` ou para `j`.

Não é possível que uma função devolva o endereço de uma variável local sua, já que ao final de sua execução, essa variável será destruída.

## Exercícios

11.1 (a) Escreva uma função com a seguinte interface:

```
void min_max(int n, int v[MAX], int *max, int *min)
```

que receba um número inteiro $n$, com $1 \leqslant n \leqslant 100$, e um vetor $v$ com $n > 0$ números inteiros e devolva um maior e um menor dos elementos desse vetor.

(b) Escreva um programa que receba $n > 0$ números inteiros, armazene-os em um vetor e, usando a função do item (a), mostre na saída um maior e um menor elemento desse conjunto.

Simule no papel a execução de seu programa antes de implementá-lo.

11.2 (a) Escreva uma função com a seguinte interface:

```
void dois_maiores(int n, int v[MAX], int *p_maior, int *s_maior)
```

que receba um número intero $n$, com $1 \leqslant n \leqslant 100$, e um vetor $v$ com $n > 0$ números inteiros e devolva um maior e um segundo maior elementos desse vetor.

(b) Escreva um programa que receba $n > 0$ números inteiros, armazene-os em um vetor e, usando a função do item (a), mostre na saída um maior e um segundo maior elemento desse conjunto.

Simule no papel a execução de seu programa antes de implementá-lo.

11.3 (a) Escreva uma função com a seguinte interface:

```
void soma_prod(int a, int b, int *soma, int *prod)
```

que receba dois números inteiros $a$ e $b$ e devolva a soma e o produto destes dois números.

(b) Escreva um programa que receba $n$ números inteiros, com $n > 0$ par, calcule a soma e o produto deste conjunto usando a função do item (a) e determine quantos deles são maiores que esta soma e quantos são maiores que o produto. Observe que os números na entrada podem ser negativos.

Simule no papel a execução de seu programa antes de implementá-lo.

11.4 (a) Escreva uma função com a seguinte interface:

```
int *maximo(int n, int v[MAX])
```

que receba um número intero $n$, com $1 \leqslant n \leqslant 100$, e um vetor $v$ de $n$ números inteiros e devolva o endereço do elemento de $v$ onde reside um maior elemento de $v$.

(b) Escreva um programa que receba $n > 0$ números inteiros, armazene-os em um vetor e, usando a função do item (a), mostre na saída um maior elemento desse conjunto.

Simule no papel a execução de seu programa antes de implementá-lo.

AULA 12

# PONTEIROS E VETORES

Nas aulas 9 e 11 aprendemos o que são os ponteiros e também como são usados como argumentos/parâmetros de funções e devolvidos de funções. Nesta aula veremos outra aplicação para os ponteiros. A linguagem C nos permite usar expressões aritméticas de adição e subtração com ponteiros que apontam para elementos de vetores. Essa é uma forma alternativa de trabalhar com vetores e seus índices. Para nos tornarmos melhores programadores na linguagem C é necessário conhecer bem essa relação íntima entre ponteiros e vetores. Além disso, o uso de ponteiros para trabalhar com vetores é vantajoso em termos de eficiência do programa executável resultante. Esta aula é baseada nas referências [2, 7].

## 12.1 Aritmética com ponteiros

Nas aulas 9 e 11 vimos que ponteiros podem apontar para elementos de um vetor. Suponha, por exemplo, que temos declaradas as seguintes variáveis:

```
int v[10], *p;
```

Podemos fazer o ponteiro $p$ apontar para o primeiro elemento do vetor $v$ fazendo a seguinte atribuição, como mostra a figura 12.1:

```
p = &v[0];
```

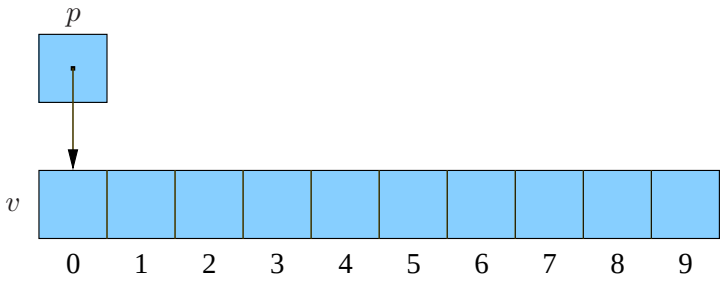



Figura 12.1: `p = &v[0];`

Podemos acessar o primeiro compartimento de $v$ através de $p$, como ilustrado na figura 12.2.

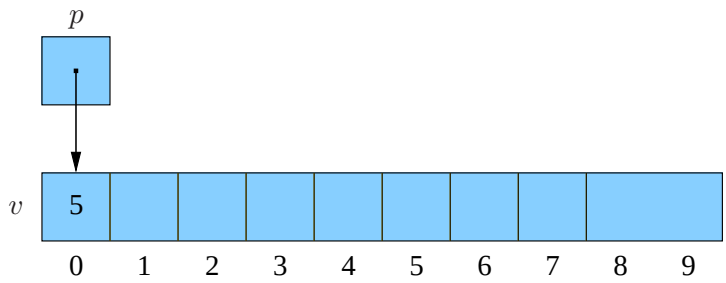



Figura 12.2: `*p = 5;`

Podemos ainda executar **aritmética com ponteiros** ou **aritmética com endereços** sobre $p$ e assim acessamos outros elementos do vetor $v$. A linguagem C possibilita três formas de aritmética com ponteiros: (*i*) adicionar um número inteiro a um ponteiro; (*ii*) subtrair um número inteiro de um ponteiro; e (*iii*) subtrair um ponteiro de outro ponteiro.

Vamos olhar para cada uma dessas operações. Suponha que temos declaradas as seguintes variáveis:

```
int v[10], *p, *q, i;
```

Adicionar um inteiro $j$ a um ponteiro $p$ fornece um ponteiro para o elemento posicionado $j$ posições após $p$. Mais precisamente, se $p$ aponta para o elemento `v[i]`, então $p + j$ aponta para `v[i+j]`. A figura 12.3 ilustra essa idéia.

Do mesmo modo, se $p$ aponta para o elemento `v[i]`, então $p - j$ aponta para `v[i-j]`, como ilustrado na figura 12.4.

Ainda, quando um ponteiro é subtraído de outro, o resultado é a distância, medida em elementos do vetor, entre os ponteiros. Dessa forma, se $p$ aponta para `v[i]` e $q$ aponta para `v[j]`, então $p - q$ é igual a $i - j$. A figura 12.5 ilustra essa situação.

Podemos comparar variáveis ponteiros entre si usando os operadores relacionais usuais (`<`, `<=`, `>`, `>=`, `==` e `!=`). Usar os operadores relacionais para comparar dois ponteiros que apontam para um mesmo vetor é uma ótima idéia. O resultado da comparação depende das posições relativas dos dois elementos do vetor. Por exemplo, depois das atribuições dadas a seguir:

```
p = &v[5];
q = &v[1];
```

o resultado da comparação `p <= q` é falso e o resultado de `p >= q` é verdadeiro.

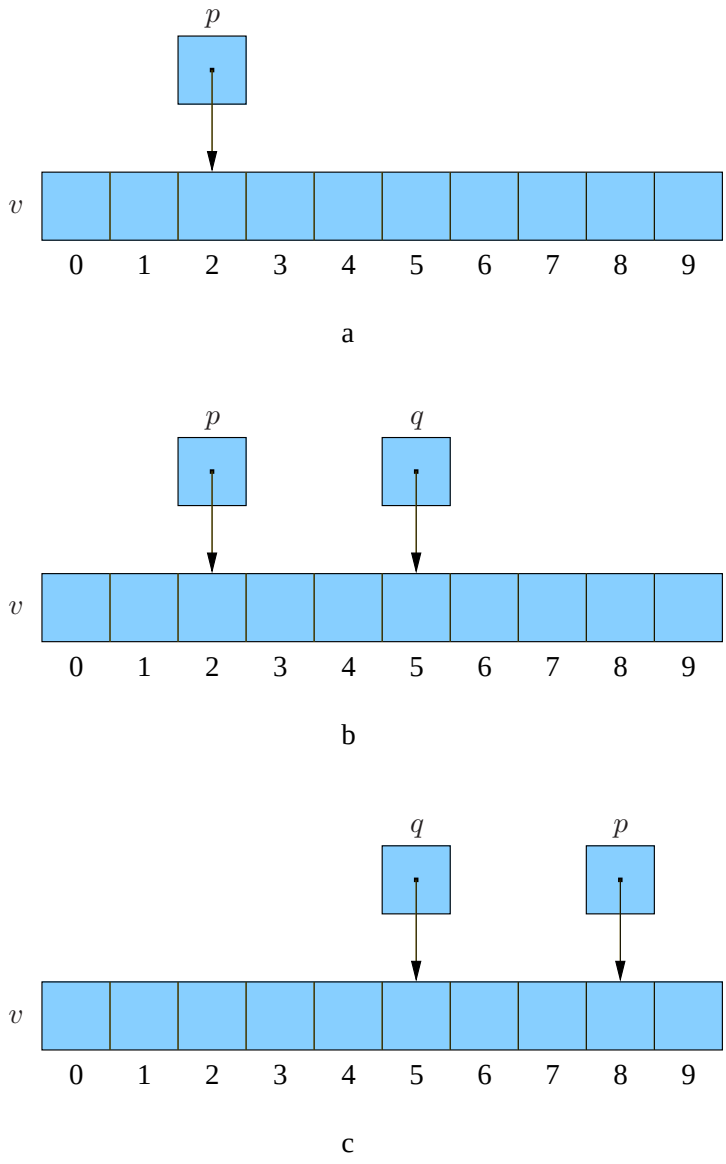



Figura 12.3: (a) `p = &v[2];` (b) `q = p + 3;` (c) `p = p + 6;`

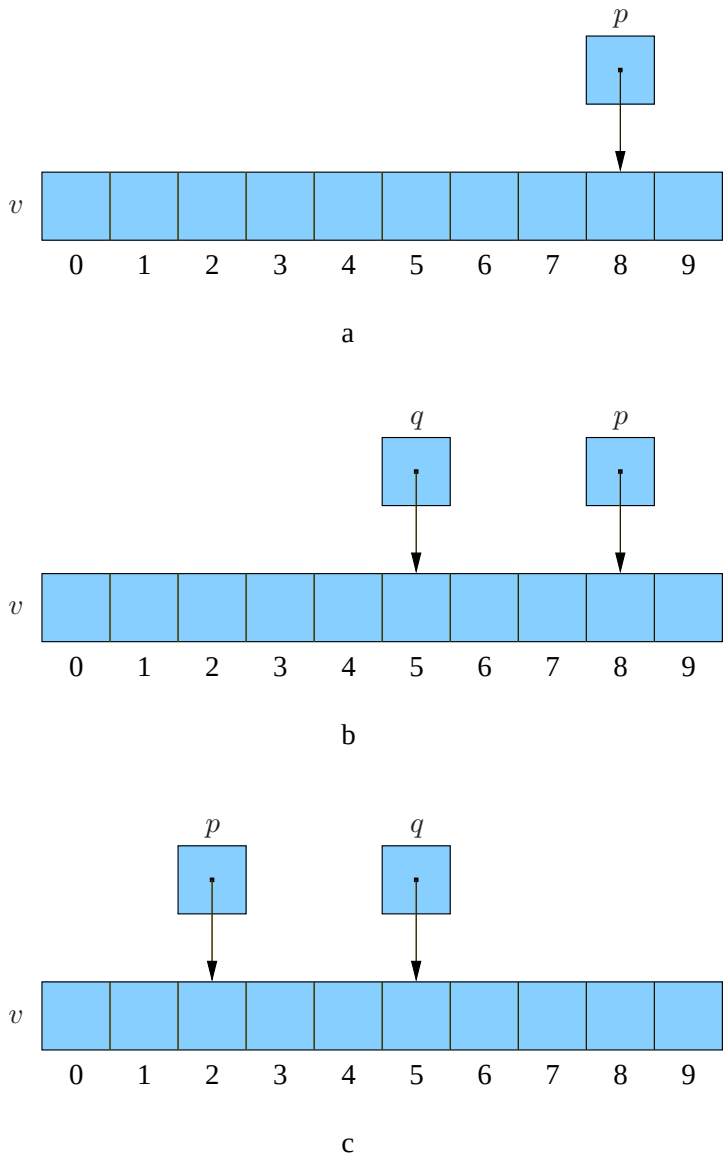



Figura 12.4: (a) `p = &v[8];` (b) `q = p - 3;` (c) `p = p - 6;`

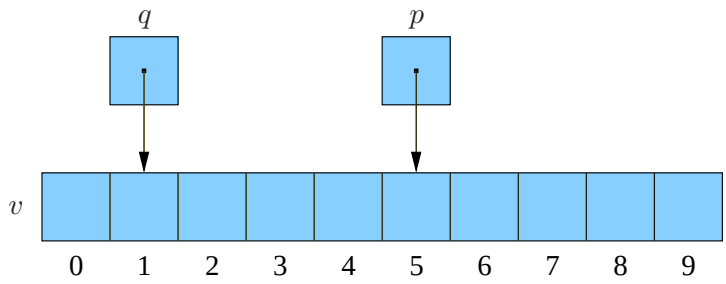



Figura 12.5: `p = &v[5];` e `q = &v[1];` A expressão $p - q$ tem valor 4 e a expressão $q - p$ tem valor $-4$.

## 12.2 Uso de ponteiros para processamento de vetores

Usando aritmética de ponteiros podemos visitar os elementos de um vetor através da atribuição de um ponteiro para seu início e do seu incremento em cada passo, como mostrado no trecho de código abaixo:

```
⋮
#define DIM 100

int main(void)
{
   int v[DIM], soma, *p;
   ⋮
   soma = 0;
   for (p = &v[0]; p < &v[DIM]; p++)
      soma = soma + *p;
   ⋮
```

A condição `p < &v[DIM]` na estrutura de repetição `for` necessita de atenção especial. Apesar de estranho, é possível aplicar o operador de endereço para `v[DIM]`, mesmo sabendo que este elemento não existe no vetor $v$. Usar `v[DIM]` dessa maneira é perfeitamente seguro, já que a sentença `for` não tenta examinar o seu valor. O corpo da estrutura de repetição `for` será executado com $p$ igual a `&v[0]`, `&v[1]`, ..., `&v[DIM-1]`, mas quando $p$ é igual a `&v[DIM]` a estrutura de repetição termina.

Como já vimos, podemos também combinar o operador de indireção `*` com operadores de incremento `++` ou decremento `--` em sentenças que processam elementos de um vetor. Considere inicialmente o caso em que queremos armazenar um valor em um vetor e então avançar para o próximo elemento. Usando um índice, podemos fazer diretamente:

```
v[i++] = j;
```

Se $p$ está apontando para o $(i+1)$-ésimo elemento de um vetor, a sentença correspondente usando esse ponteiro é:

```
*p++ = j;
```

Devido à precedência do operador `++` sobre o operador `*`, o compilador enxerga essa sentença como

```
*(p++) = j;
```

O valor da expressão `*p++` é o valor de `*p`, antes do incremento. Depois que esse valor é devolvido, a sentença incrementa $p$.

A expressão `*p++` não é a única combinação possível dos operadores `*` e `++`. Podemos escrever `(*p)++` para incrementar o valor de `*p`. Nesse caso, o valor devolvido pela expressão é também `*p`, antes do incremento. Em seguida, a sentença incrementa `*p`. Ainda, podemos escrever `*++p` ou ainda `++*p`. O primeiro caso incrementa $p$ e o valor da expressão é `*p`, depois do incremento. O segundo incrementa `*p` e o valor da expressão é `*p`, depois do incremento.

O trecho de código acima, que realiza a soma dos elementos do vetor $v$ usando aritmética com ponteiros, pode então ser reescrito como a seguir, usando uma combinação dos operadores `*` e `++`.

```
⋮
soma = 0;
p = &v[0];
while (p < &v[DIM])
   soma = soma + *p++;
⋮
```

## 12.3 Uso do identificador de um vetor como ponteiro

Ponteiros e vetores estão intimamente relacionados. Como vimos nas seções anteriores, usamos aritmética de ponteiros para trabalhar com vetores. Mas essa não é a única relação entre eles. Outra relação importante entre ponteiros e vetores fornecida pela linguagem C é que o identificador de um vetor pode ser usado como um ponteiro para o primeiro elemento do vetor. Essa relação simplifica a aritmética com ponteiros e estabelece ganho de versatilidade em ambos, ponteiros e vetores.

Por exemplo, suponha que temos o vetor $v$ declarado como abaixo:

```
int v[10];
```

Usando $v$ como um ponteiro para o primeiro elemento do vetor, podemos modificar o conteúdo de `v[0]` da seguinte forma:

```
*v = 7;
```

Podemos também modificar o conteúdo de `v[1]` através do ponteiro `v + 1`:

```
*(v + 1) = 12;
```

Em geral, $v + i$ é o mesmo que `&v[i]`, e `*(v + i)` é equivalente a `v[i]`. Em outras palavras, índices de vetores podem ser vistos como uma forma de aritmética de ponteiros.

O fato que o identificador de um vetor pode servir como um ponteiro facilita nossa programação de estruturas de repetição que percorrem vetores. Considere a estrutura de repetição do exemplo dado na seção anterior:

```
soma = 0;
for (p = &v[0]; p < &v[DIM]; p++)
   soma = soma + *p;
```

Para simplificar essa estrutura de repetição, podemos substituir `&v[0]` por $v$ e `&v[DIM]` por `v + DIM`, como mostra o trecho de código abaixo:

```
soma = 0;
for (p = v; p < v + DIM; p++)
   soma = soma + *p;
```

Apesar de podermos usar o identificador de um vetor como um ponteiro, não é possível atribuir-lhe um novo valor. A tentativa de fazê-lo apontar para qualquer outro lugar é um erro, como mostra o trecho de código abaixo:

```
while (*v != 0)
   v++;
```

O programa 12.1 mostra um exemplo do uso desses conceitos, realizando a impressão dos elementos de um vetor na ordem inversa da qual forma lidos.

Programa 12.1: Imprime os elementos na ordem inversa da de leitura.

```
#include <stdio.h>

#define N 10

int main(void)
{
   int v[N], *p;

   printf("Informe %d números: ", N);
   for (p = v; p < v + N; p++)
      scanf("%d", p);

   printf("Em ordem inversa: ");
   for (p = v + N - 1; p >= v; p--)
      printf("%d ", *p);
   printf("\n");

   return 0;
}
```

Outro uso do identificador de um vetor como um ponteiro é quando um vetor é um argumento em uma chamada de função. Nesse caso, o vetor é sempre tratado como um ponteiro. Considere a seguinte função que recebe um vetor de $n$ números inteiros e devolve um maior elemento nesse vetor.

```
/* Recebe um número inteiro n > 0 e um vetor v com n
   números inteiros e devolve um maior elemento em v */
int max(int n, int v[MAX])
{
   int i, maior;

   maior = v[0];
   for (i = 1; i < n; i++)
      if (v[i] > maior)
         maior = v[i];

   return maior;
}
```

Suponha que chamamos a função `max` da seguinte forma:

```
M = max(N, u);
```

Essa chamada faz com que o endereço do primeiro compartimento do vetor $u$ seja atribuído à $v$. O vetor $u$ não é de fato copiado.

Para indicar que queremos que um parâmetro que é um vetor não seja modificado, podemos incluir a palavra reservada `const` precedendo a sua declaração.

Quando uma variável simples é passada para uma função, isto é, quando é um argumento de uma função, seu valor é copiado no parâmetro correspondente. Então, qualquer alteração no parâmetro correspondente não afeta a variável. Em contraste, um vetor usado como um argumento não está protegido contra alterações, já que não ocorre uma cópia do vetor todo. Desse modo, o tempo necessário para passar um vetor a uma função independe de seu tamanho. Não há perda por passar vetores grandes, já que nenhuma cópia do vetor é realizada. Além disso, um *parâmetro* que é um vetor pode ser declarado como um ponteiro. Por exemplo, a função `max` descrita acima pode ser declarada como a seguir:

```
/* Recebe um número inteiro n > 0 e um ponteiro v para um ve-
   tor com n números inteiros e devolve um maior elemento em v */
int max(int n, int *v)
{
   int i, maior;

   maior = v[0];
   for (i = 1; i < n; i++)
      if (v[i] > maior)
         maior = v[i];

   return maior;
}
```

Neste caso, declarar o parâmetro $v$ como sendo um ponteiro é equivalente a declarar $v$ como sendo um vetor. O compilador trata ambas as declarações como idênticas.

Apesar de a declaração de um *parâmetro* como um vetor ser equivalente à declaração do mesmo *parâmetro* como um ponteiro, o mesmo não vale para uma *variável*. A declaração a seguir:

```
int v[10];
```

faz com que o compilador reserve espaço para 10 números inteiros. Por outro lado, a declaração abaixo:

```
int *v;
```

faz o compilador reservar espaço para uma variável ponteiro. Nesse último caso, $v$ não é um vetor e tentar usá-lo como tal pode causar resultados desastrosos. Por exemplo, a atribuição:

```
*v = 7;
```

armazena o valor 7 onde $v$ está apontando. Como não sabemos para onde $v$ está apontando, o resultado da execução dessa linha de código é imprevisível.

Do mesmo modo, podemos usar uma variável ponteiro, que aponta para uma posição de um vetor, como um vetor. O trecho de código a seguir ilustra essa afirmação.

```
⋮
#define DIM 100

int main(void)
{
   int v[DIM], soma, *p;
   ⋮
   soma = 0;
   p = v;
   for (i = 0; i < DIM; i++)
      soma = soma + p[i];
```

O compilador trata a referência `p[i]` como `*(p + i)`, que é uma forma possível de usar aritmética com ponteiros, como vimos anteriormente. Essa possibilidade de uso, que parece um tanto estranha à primeira vista, é muito útil em alocação dinâmica de memória, como veremos em uma próxima aula.

## Exercícios

12.1 Suponha que as declarações e atribuições simultâneas tenham sido realizadas nas variáveis listadas abaixo:

```
int v[] = {5, 15, 34, 54, 14, 2, 52, 72};
int *p = &v[1], *q = &v[5];
```

(a) Qual o valor de `*(p + 3)`?

(b) Qual o valor de `*(q - 3)`?

(c) Qual o valor de `q - p`?

(d) A expressão `p < q` tem valor verdadeiro ou falso?

(e) A expressão `*p < *q` tem valor verdadeiro ou falso?

12.2 Qual o conteúdo do vetor $v$ após a execução do seguinte trecho de código?

```
⋮
#define N 10

int main(void)
{
   int v[N] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   int *p = &v[0], *q = &v[N - 1], temp;

   while (p < q) {
      temp = *p;
      *p++ = *q;
      *q-- = temp;
   }
   ⋮
```

12.3 Suponha que $v$ é um vetor e $p$ é um ponteiro. Considere que a atribuição `p = v;` foi realizada previamente. Quais das expressões abaixo não são permitidas? Das restantes, quais têm valor verdadeiro?

(a) `p == v[0]`

(b) `p == &v[0]`

(c) `*p == v[0]`

(d) `p[0] == v[0]`

12.4 Escreva um programa que leia uma mensagem e a imprima em ordem reversa. Use a função `getchar` para ler caractere por caractere, armazenando-os em um vetor. Pare quando encontrar um caractere de mudança de linha `'\n'`. Faça o programa de forma a usar um ponteiro, ao invés de um índice como um número inteiro, para controlar a posição corrente no vetor.

AULA 13

# PONTEIROS E MATRIZES

Na aula 12, trabalhamos com ponteiros e vetores. Nesta aula veremos as relações entre ponteiros e matrizes. Neste caso, é necessário estender o conceito de indireção (simples) para indireção dupla. Veremos também como estender esse conceito para indireção múltipla e suas relações com variáveis compostas homogêneas multi-dimensionais.

Esta aula é baseada na referência [7].

## 13.1 Ponteiros para elementos de uma matriz

Sabemos que a linguagem C armazena matrizes, que são objetos representados de forma bi-dimensional, como uma seqüência contínua de compartimentos de memória, com demarcações de onde começa e onde termina cada uma de suas linhas. Ou seja, uma matriz é armazenada como um vetor na memória, com marcas especiais em determinados pontos regularmente espaçados: os elementos da linha 0 vêm primeiro, seguidos pelos elementos da linha 1, da linha 2, e assim por diante. Uma matriz com $n$ linhas tem a aparência ilustrada como na figura 13.1.

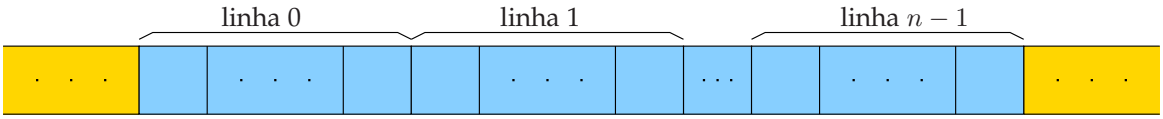



Figura 13.1: Representação da alocação de espaço na memória para uma matriz.

Essa representação pode nos ajudar a trabalhar com ponteiros e matrizes. Se fazemos um ponteiro $p$ apontar para a primeira célula de uma matriz, isto é, o elemento na posição (0,0), podemos então visitar todos os elementos da matriz incrementando o ponteiro $p$ repetidamente.

Por exemplo, suponha que queremos inicializar todos os elementos de uma matriz de números inteiros com 0. Suponha que temos a declaração da seguinte matriz:

```
int A[LINHAS][COLUNAS];
```

A forma que aprendemos inicializar uma matriz pode ser aplicada à matriz $A$ declarada acima e então o seguinte trecho de código realiza a inicialização desejada:

```
int i, j;
⋮
for (i = 0; i < LINHAS; i++)
   for (j = 0; j < COLUNAS; j++)
      A[i][j] = 0;
```

Mas se vemos a matriz $A$ da forma como é armazenada na memória, isto é, como um vetor unidimensional, podemos trocar o par de estruturas de repetição por uma única estrutura de repetição:

```
int *p;
⋮
for (p = &A[0][0]; p <= &A[LINHAS-1][COLUNAS-1]; p++)
   *p = 0;
```

A estrutura de repetição acima inicia com $p$ apontando para `A[0][0]`. Os incrementos sucessivos de $p$ fazem-no apontar para `A[0][1]`, `A[0][2]`, e assim por diante. Quando $p$ atinge `A[0][COLUNAS-1]`, o último elemento da linha 0, o próximo incremento faz com que $p$ aponte para `A[1][0]`, o primeiro elemento da linha 1. O processo continua até que $p$ ultrapasse o elemento `A[LINHAS-1][COLUNAS-1]`, o último elemento da matriz.

## 13.2 Processamento das linhas de uma matriz

Podemos processar uma linha de uma matriz – ou seja, percorrê-la, visitar os conteúdos dos compartimentos, usar seus valores, modificá-los, etc – também usando ponteiros. Por exemplo, para visitar os elementos da linha $i$ de uma matriz $A$ podemos usar um ponteiro $p$ apontar para o elemento da linha $i$ e da coluna 0 da matriz $A$, como mostrado abaixo:

```
p = &A[i][0];
```

ou poderíamos fazer simplesmente

```
p = A[i];
```

já que a expressão `A[i]` é um ponteiro para o primeiro elemento da linha $i$.

A justificativa para essa afirmação vem da aritmética com ponteiros. Lembre-se que para um vetor $A$, a expressão `A[i]` é equivalente a `*(A + i)`. Assim, `&A[i][0]` é o mesmo que `&(*(A[i] + 0))`, que é equivalente a `&*A[i]` e que, por fim, é equivalente a `A[i]`. No trecho de código a seguir usamos essa simplificação para inicializar com zeros a linha $i$ da matriz $A$:

```
int A[LINHAS][COLUNAS], *p, i;
⋮
for (p = A[i]; p < A[i] + COLUNAS; p++)
   *p = 0;
```

Como `A[i]` é um ponteiro para a linha $i$ da matriz $A$, podemos passar `A[i]` para um função que espera receber um vetor como argumento. Em outras palavras, um função que foi projetada para trabalhar com um vetor também pode trabalhar com uma linha de uma matriz. Dessa forma, a função `max` da aula 12 pode ser chamada com a linha $i$ da matriz $A$ como argumento:

```
M = max(COLUNAS, A[i]);
```

## 13.3 Processamento das colunas de uma matriz

O processamento dos elementos de uma coluna de uma matriz não é tão simples como o processamento dos elementos de uma linha, já que a matriz é armazenada linha por linha na memória. A seguir, mostramos uma estrutura de repetição que inicializa com zeros a coluna $j$ da matriz $A$:

```
int A[LINHAS][COLUNAS], (*p)[COLUNAS], j;
⋮
for (p = &A[0]; p < A[LINHAS]; p++)
   (*p)[j] = 0;
```

Nesse exemplo, declaramos $p$ como um ponteiro para um vetor de dimensão `COLUNAS`, cujos elementos são números inteiros. Os parênteses envolvendo `*p` são necessários, já que sem eles o compilador trataria $p$ como um vetor de ponteiros em vez de um ponteiro para um vetor. A expressão `p++` avança $p$ para a próxima linha. Na expressão `(*p)[j]`, `*p` representa uma linha inteira de $A$ e assim `(*p)[j]` seleciona o elemento na coluna $j$ da linha. Os parênteses são essenciais na expressão `(*p)[i]`, já que, sem eles, o compilador interpretaria a expressão como `*(p[i])`.

## 13.4 Identificadores de matrizes como ponteiros

Assim como o identificador de um vetor pode ser usado como um ponteiro, o identificador de uma matriz também pode e, na verdade, de qualquer o identificador de qualquer variável composta homogênea pode. No entanto, alguns cuidados são necessários quando ultrapassamos a barreira de duas ou mais dimensões.

Considere a declaração da matriz a seguir:

```
int A[LINHAS][COLUNAS];
```

Neste caso, $A$ *não* é um ponteiro para `A[0][0]`. Ao contrário, é um ponteiro para `A[0]`. Isso faz mais sentido se olharmos sob o ponto de vista da linguagem C, que considera $A$ não como uma matriz bidimensional, mas como um vetor. Quando usado como um ponteiro, $A$ tem tipo `int (*)[COLUNAS]`, um ponteiro para um vetor de números inteiros de tamanho `COLUNAS`.

Saber que $A$ aponta para `A[0]` é útil para simplificar estruturas de repetição que processam elementos de uma matriz. Por exemplo, ao invés de escrever:

```
for (p = &A[0]; p < &A[LINHAS]; p++)
   (*p)[j] = 0;
```

para inicializar a coluna $j$ da matriz $A$, podemos escrever

```
for (p = A; p < A + LINHAS; p++)
   (*p)[j] = 0;
```

Com essa idéia, podemos fazer o compilador acreditar que uma variável composta homogênea multi-dimensional é unidimensional, isto é, é um vetor. Por exemplo, podemos passar a matriz $A$ como argumento para a função `max` da aula 12 da seguinte forma:

```
M = max(LINHAS*COLUNAS, A[0]);
```

já que `A[0]` aponta para o elemento na linha 0 e coluna 0 e tem tipo `int *` e assim essa chamada será executada corretamente.

## Exercícios

13.1 Escreva uma função que preencha uma matriz quadrada de dimensão $n$ com a matriz identidade $I_n$. Use um único ponteiro que percorra a matriz.

13.2 Reescreva a função abaixo usando aritmética de ponteiros em vez de índices de matrizes. Em outras palavras, elimine as variáveis $i$ e $j$ e todos os `[]`. Use também uma única estrutura de repetição.

```
int soma_matriz(int n, const int A[DIM][DIM])
{
   int i, j, soma = 0;

   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         soma = soma + A[i][j];

   return soma;
}
```

AULA 14

# PONTEIROS E CADEIAS

Nas aulas 12 e 13 estudamos formas de trabalhar com variáveis compostas homogêneas e ponteiros para seus elementos. No entanto, é importante ainda estudar a relação entre ponteiros e as cadeias de caracteres que, como já vimos, são vetores especiais que contêm caracteres. Nesta aula, baseada na referência [7], aprenderemos algumas particularidades de ponteiros para elementos de cadeias de caracteres na linguagem C, além de estudar a relação entre ponteiros e constantes que são cadeias de caracteres.

## 14.1 Literais e ponteiros

Devemos relembrar que uma **literal** é uma seqüência de caracteres envolvida por aspas duplas. Um exemplo de uma literal é apresentado a seguir[1]:

```
"O usuário médio de computador possui o cérebro de um macaco-aranha"
```

Nosso primeiro contato com literais foi ainda em Algoritmos e Programação I. Literais ocorrem com freqüência na chamada das funções `printf` e `scanf`. Mas quando chamamos uma dessas funções e fornecemos uma literal como argumento, o que de fato estamos passando? Em essência, a linguagem C trata literais como cadeias de caracteres. Quando o compilador encontra uma literal de comprimento $n$ em um programa, ele reserva um espaço de $n+1$ bytes na memória. Essa área de memória conterá os caracteres da literal mais o caracter nulo que indica o final da cadeia. O caracter nulo é um byte cujos bits são todos zeros e é representado pela seqüência de caracteres `\0`. Por exemplo, a literal `"abc"` é armazenada como uma cadeia de quatro caracteres, como mostra a figura 14.1.

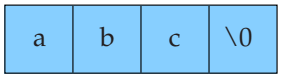



Figura 14.1: Representação de uma literal na memória.

Literais podem ser vazias. A literal `""` é armazenada como um único caractere nulo.

Como uma literal é armazenada em um vetor, o compilador a enxerga como um ponteiro do tipo `char *`. As funções `printf` e `scanf`, por exemplo, esperam um valor do tipo `char *` como primeiro argumento. Se, por exemplo, fazemos a seguinte chamada:

[1] Frase de Bill Gates, dono da Micro$oft, em uma entrevista para *Computer Magazine*.

```
printf("abc");
```

o endereço da literal `"abc"` é passado como argumento para a função `printf`, isto é, o endereço de onde se encontra o caractere `a` na memória.

Em geral, podemos usar uma literal sempre que a linguagem C permita o uso de um ponteiro do tipo `char *`. Por exemplo, uma literal pode ocorrer do lado direito de uma atribuição, como mostrado a seguir:

```
char *p;
p = "abc";
```

Essa atribuição não copia os caracteres de `"abc"`, mas faz o ponteiro $p$ apontar para o primeiro caractere da literal, como mostra a figura 14.2.

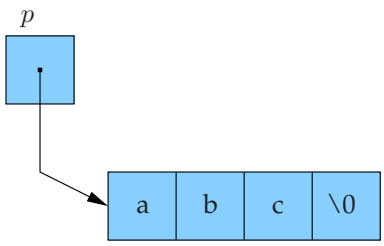



Figura 14.2: Representação da atribuição de uma literal a um ponteiro.

Observe ainda que não é permitido alterar uma literal durante a execução de um programa. Isso significa que a tentativa de modificar uma literal pode causar um comportamento indefinido do programa.

Relembrando, uma variável cadeia de caracteres é um vetor do tipo `char` que necessariamente deve reservar espaço para o caractere nulo. Quando declaramos um vetor de caracteres que será usado para armazenar cadeias de caracteres, devemos sempre declarar esse vetor com uma posição a mais que a mais longa das cadeias de caracteres possíveis, já que por convenção da linguagem C, toda cadeia de caracteres é finalizada com um caractere nulo. Veja, por exemplo, a declaração a seguir:

```
char cadeia[TAM+1];
```

onde `TAM` é uma macro definida com o tamanho da cadeia de caracteres mais longa que pode ser armazenada na variável `cadeia`.

Lembrando ainda, podemos inicializar uma cadeia de caracteres no momento de sua declaração, como mostra o exemplo abaixo:

```
char data[13] = "7 de outubro";
```

O compilador então coloca os caracteres de `"7 de outubro"` no vetor `data` e adiciona o caractere nulo ao final para que `data` possa ser usada como uma cadeia de caracteres. A figura 14.3 ilustra essa situação.


Figura 14.3: Declaração e inicialização de uma cadeia de caracteres.

Apesar de `"7 de outubro"` se parecer com uma literal, a linguagem C de fato a vê como uma abreviação para um inicializador de um vetor. Ou seja, a declaração e inicialização acima é enxergada pelo compilador como abaixo:

```
char data[13] = {'7',' ','d','e',' ','o','u','t','u','b','r','o','\0'};
```

No caso em que o inicializador é menor que o tamanho definido para a cadeia de caracteres, o compilador preencherá as posições finais restantes da cadeia com o caractere nulo. Por outro lado, é sempre importante garantir que o inicializador tenha menor comprimento que o tamanho do vetor declarado. Também, podemos omitir o tamanho do vetor em uma declaração e inicialização simultâneas, caso em que o vetor terá o tamanho equivalente ao comprimento do inicializador mais uma unidade, que equivale ao caractere nulo.

Agora, vamos comparar a declaração abaixo:

```
char data[] = "7 de outubro";
```

que declara um vetor `data`, que é uma cadeia de caracteres, com a declaração a seguir:

```
char *data = "7 de outubro";
```

que declara `data` como um ponteiro. Devido à relação estrita entre vetores e ponteiros que vimos na aula 12, podemos usar as duas versões da declaração de `data` como uma cadeia de caracteres. Em particular, qualquer função que receba um vetor/cadeia de caracteres ou um ponteiro para caracteres aceita qualquer uma das versões da declaração da variável `data` apresentada acima.

No entanto, devemos ter cuidado para não cometer o erro de acreditar que as duas versões da declaração de `data` são equivalentes e intercambiáveis. Existem diferenças significativas entre as duas, que destacamos abaixo:

- na versão em que a variável é declarada como um vetor, os caracteres armazenados em `data` podem ser modificados, como fazemos com elementos de qualquer vetor; na versão em que a variável é declarada como um ponteiro, `data` aponta para uma literal que, como já vimos, não pode ser modificada;

- na versão com vetor, `data` é um identificador de um vetor; na versão com ponteiro, `data` é uma variável que pode, inclusive, apontar para outras cadeias de caracteres durante a execução do programa.

Se precisamos que uma cadeia de caracteres seja modificada, é nossa responsabilidade declarar um vetor de caracteres no qual será armazenada essa cadeia. Declarar um ponteiro não é suficiente, neste caso. Por exemplo, a declaração abaixo:

```
char *p;
```

faz com que o compilador reserve espaço suficiente para uma variável ponteiro. Infelizmente, o compilador não reserva espaço para uma cadeia de caracteres, mesmo porque, não há indicação alguma de um possível comprimento da cadeia de caracteres que queremos armazenar. Antes de usarmos $p$ como uma cadeia de caracteres, temos de fazê-la apontar para um vetor de caracteres. Uma possibilidade é fazer $p$ apontar para uma variável que é uma cadeia de caracteres, como mostramos a seguir:

```
char cadeia[TAM+1], *p;
p = cadeia;
```

Com essa atribuição, $p$ aponta para o primeiro caractere de `cadeia` e assim podemos usar $p$ como uma cadeia de caracteres. Outra possibilidade é fazer $p$ apontar para uma cadeia de caracteres dinamicamente alocada, como veremos na aula 16.

Ainda poderíamos discorrer sobre processos para leitura e escrita de cadeias de caracteres, sobre acesso aos caracteres de uma cadeia de caracteres e também sobre o uso das funções da biblioteca da linguagem C que trabalham especificamente com cadeias de caracteres, o que já fizemos nas aulas de Algoritmos e Programação I. Ainda veremos a seguir dois tópicos importantes sobre cadeias de caracteres: vetores de cadeias de caracteres e argumentos de linha de comando.

## 14.2 Vetores de cadeias de caracteres

Uma forma de armazenar em memória um vetor de cadeias de caracteres é através da criação de uma matriz de caracteres e então armazenar as cadeias de caracteres uma a uma. Por exemplo, podemos fazer como a seguir:

```
char planetas[][9] = {"Mercurio", "Venus", "Terra",
                      "Marte", "Jupiter", "Saturno",
                      "Urano", "Netuno", "Plutao"};
```

Observe que estamos omitindo o número de linhas da matriz, que é fornecido pelo inicializador, mas a linguagem C exige que o número de colunas seja especificado, conforme fizemos na declaração.

A figura 14.4 ilustra a declaração e inicialização da variável `planetas`. Observe que todas as cadeias cabem nas colunas da matriz e, também, que há um tanto de compartimentos desperdiçados na matriz, preenchidos com o caractere `\0`, já que nem todas as cadeias são compostas por 8 caracteres.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | M | e | r | c | u | r | i | o | \0 |
| 1 | V | e | n | u | s | \0 | \0 | \0 | \0 |
| 2 | T | e | r | r | a | \0 | \0 | \0 | \0 |
| 3 | M | a | r | t | e | \0 | \0 | \0 | \0 |
| 4 | J | u | p | i | t | e | r | \0 | \0 |
| 5 | S | a | t | u | r | n | o | \0 | \0 |
| 6 | U | r | a | n | o | \0 | \0 | \0 | \0 |
| 7 | N | e | t | u | n | o | \0 | \0 | \0 |
| 8 | P | l | u | t | a | o | \0 | \0 | \0 |

Figura 14.4: Matriz de cadeias de caracteres `planetas`.

A ineficiência de armazenamento aparente nesse exemplo é comum quando trabalhamos com cadeias de caracteres, já que coleções de cadeias de caracteres serão, em geral, um misto entre curtas e longas cadeias. Uma possível forma de sanar esse problema é usar um vetor cujos elementos são ponteiros para cadeias de caracteres, como podemos ver na declaração abaixo:

```
char *planetas[] = {"Mercurio", "Venus", "Terra",
                    "Marte", "Jupiter", "Saturno",
                    "Urano", "Netuno", "Plutao"};
```

Note que há poucas diferenças entre essa declaração e a declaração anterior da variável `planetas`: removemos um par de colchetes com um número no interior deles e colocamos um asterisco precedendo o identificador da variável. No entanto, o efeito dessa declaração na memória é muito diferente, como podemos ver na figura 14.5.

Cada elemento do vetor `planetas` é um ponteiro para uma cadeia de caracteres, terminada com um caractere nulo. Não há mais desperdício de compartimentos nas cadeias de caracteres, apesar de termos de alocar espaço para os ponteiros no vetor `planetas`. Para acessar

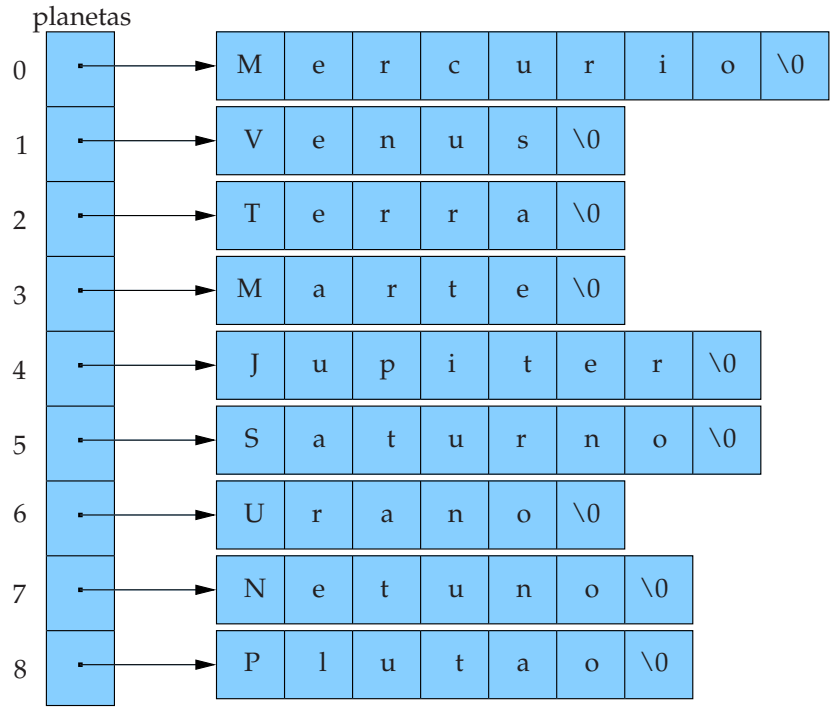



Figura 14.5: Vetor **planetas** de ponteiros para cadeias de caracteres.

um dos nomes dos planetas necessitamos apenas do índice do vetor. Para acessar um caractere do nome de um planeta devemos fazer da mesma forma como acessamos um elemento em uma matriz. Por exemplo, para buscar cadeias de caracteres no vetor **planetas** que iniciam com a letra M, podemos usar o seguinte trecho de código:

```
for (i = 0; i < 9; i++)
   if (planetas[i][0] == 'M')
      printf("%s começa com M\n", planetas[i]);
```

## 14.3 Argumentos na linha de comandos

Quando executamos um programa, em geral, devemos fornecer a ele alguma informação como, por exemplo, um nome de um arquivo, uma opção que modifica seu comportamento, etc. Por exemplo, considere o comando UNIX **ls**. Se executamos esse comando como abaixo:

```
prompt$ ls
```

em uma linha de comando, o resultado será uma listagem de nomes dos arquivos no diretório atual. Se digitamos o comando seguido de uma opção, como abaixo:

```
prompt$ ls -l
```

então o resultado é uma listagem detalhada[2] que nos mostra o tamanho de cada arquivo, seu proprietário, a data e hora em que houve a última modificação no arquivo e assim por diante. Para modificar ainda mais o comportamento do comando `ls` podemos especificar que ele mostre detalhes de apenas um arquivo, como mostrado abaixo:

```
prompt$ ls -l exerc1.c
```

Nesse caso, o comando `ls` mostrará informações detalhadas sobre o arquivo `exerc1.c`.

Informações em linha de comando estão disponíveis para todos os programas, não apenas para comandos do sistema operacional. Para ter acesso aos **argumentos de linha de comando**, chamados de **parâmetros do programa** na linguagem C padrão, devemos definir a função `main` como uma função com dois parâmetros que costumeiramente têm identificadores `argc` e `argv`. Isto é, devemos fazer como abaixo:

```
int main(int argc, char *argv[])
{
   ⋮
}
```

O parâmetro `argc`, abreviação de "contador de argumentos", é o número de argumentos de linha de comando, incluindo também o nome do programa. O parâmetro `argv`, abreviação de "vetor de argumentos", é um vetor de ponteiros para os argumentos da linha de comando, que são armazenados como cadeias de caracteres. Assim, `argv[0]` aponta para o nome do programa, enquanto que `argv[1]` até `argv[argc-1]` apontam para os argumentos da linha de comandos restantes. O vetor `argv` tem um elemento adicional `argv[argc]` que é sempre um ponteiro nulo, um ponteiro especial que aponta para nada, representado pela macro `NULL`.

Se um(a) usuário(a) digita a linha de comando abaixo:

```
prompt$ ls -l exerc1.c
```

então `argc` conterá o valor 3, `argv[0]` apontará para a cadeia de caracteres com o nome do programa, `argv[1]` apontará para a cadeia de caracteres `"-l"`, `argv[2]` apontará para a cadeia de caracteres `"exerc1.c"` e `argv[3]` apontará para nulo. A figura 14.6 ilustra essa situação. Observe que o nome do programa não foi listado porque pode incluir o nome do diretório ao qual o programa pertence ou ainda outras informações que dependem do sistema operacional.

[2] `l` do inglês *long*.

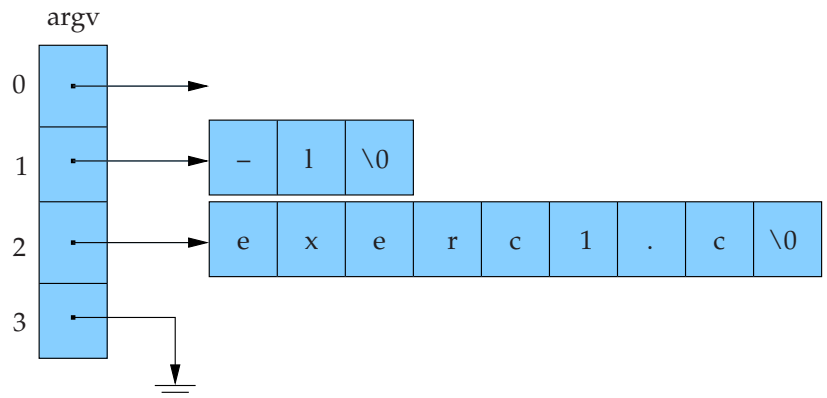



Figura 14.6: Representação de `argv`.

Como `argv` é um vetor de ponteiros, o acesso aos argumentos da linha de comandos é realizado, em geral, como mostrado na estrutura de repetição a seguir:

```
int i;
⋮
for (i = 1; i < argc; i++)
   printf("%s\n", argv[i]);
```

O programa 14.1 ilustra como acessar os argumentos de uma linha de comandos.

Programa 14.1: Verifica nomes de planetas.

```
#include <stdio.h>
#include <string.h>
#define NUM_PLANETAS 9

int main(int argc, char *argv[])
{
   char *planetas[] = {"Mercurio", "Venus", "Terra", "Marte", "Jupiter",
                       "Saturno", "Urano", "Netuno", "Plutao"};
   int i, j, k;

   for (i = 1; i < argc; i++) {
      for (j = 0; j < NUM_PLANETAS; j++)
         if (strcmp(argv[i], planetas[j]) == 0) {
            k = j;
            j = NUM_PLANETAS;
         }
      if (j == NUM_PLANETAS + 1)
         printf("%s é o planeta %d\n", argv[i], k);
      else
         printf("%s não é um planeta\n", argv[i]);
   }
   return 0;
}
```

Se o programa 14.1 tem o nome `planetas.c` e seu executável correspondente tem nome `planetas`, então podemos executar esse programa com uma seqüência de cadeias de caracteres, como mostramos no exemplo abaixo:

```
prompt$ ./planetas Jupiter venus Terra Joaquim
```

O resultado dessa execução é dado a seguir:

```
Jupiter é o planeta 5
venus não é um planeta
Terra é o planeta 3
Joaquim não é um planeta
```

## Exercícios

14.1 As chamadas de funções abaixo supostamente escrevem um caractere de mudança de linha na saída, mas algumas delas estão erradas. Identifique quais chamadas não funcionam e explique o porquê.

(a) `printf("%c", '\n');`
(b) `printf("%c", "\n");`
(c) `printf("%s", '\n');`
(d) `printf("%s", "\n");`
(e) `printf('\n');`
(f) `printf("\n");`
(g) `putchar('\n');`
(h) `putchar("\n");`

14.2 Suponha que declaramos um ponteiro $p$ como abaixo:

```
char *p = "abc";
```

Quais das chamadas abaixo estão corretas? Mostre a saída produzida por cada chamada correta e explique por que a(s) outra(s) não está(ão) correta(s).

(a) `putchar(p);`
(b) `putchar(*p);`
(c) `printf("%s", p);`
(d) `printf("%s", *p);`

14.3 Suponha que declaramos as seguintes variáveis:

```
char s[MAX+1];
int i, j;
```

Suponha também que a seguinte chamada foi executada:

```
scanf("%d%s%d", &i, s, &j);
```

Se o(a) usuário(a) digita a seguinte entrada:

```
12abc34 56def78
```

quais serão os valores de $i$, $j$ e $s$ depois dessa chamada?

14.4 A função abaixo supostamente cria uma cópia idêntica de uma cadeia de caracteres. O que há de errado com a função?

```
char *duplica(const char *p)
{
   char *q;

   strcpy(q, p);

   return q;
}
```

14.5 O que imprime na saída o programa abaixo?

```
#include <stdio.h>

int main(void)
{
   char s[] = "Dvmuvsb", *p;

   for (p = s; *p; p++)
      --*p;
   printf("%s\n", s);

   return 0;
}
```

14.6 (a) Escreva uma função com a seguinte interface:

```
void maiuscula(char cadeia[])
```

que receba uma cadeia de caracteres (terminada com um caractere nulo) contendo caracteres arbitrários e substitua os caracteres que são letras minúsculas nessa cadeia por letras maiúsculas. Use `cadeia` apenas como vetor, juntamente com os índices necessários.

(b) Escreva uma função com a seguinte interface:

```
void maiuscula(char *cadeia)
```

que receba uma cadeia de caracteres (terminada com um caractere nulo) contendo caracteres arbitrários e substitua os caracteres que são letras minúsculas nessa cadeia por letras maiúsculas. Use apenas ponteiros e aritmética com ponteiros.

14.7 (a) Escreva uma função que receba uma cadeia de caracteres e devolva o número total de caracteres que ela possui.

(b) Escreva uma função que receba uma cadeia de caracteres e devolva o número de vogais que ela possui.

(c) Escreva uma função que receba uma cadeia de caracteres e devolva o número de consoantes que ela possui.

(d) Escreva um programa que receba diversas cadeias de caracteres e faça a média do número de vogais, de consoantes e de símbolos de pontuação que elas possuem.

Use apenas ponteiros nas funções em (a), (b) e (c).

14.8 Escreva um programa que encontra a maior e a menor palavra de uma seqüência de palavras informadas pelo(a) usuário(a). O programa deve terminar se uma palavra de quatro letras for fornecida na entrada. Considere que nenhuma palavra tem mais que 20 letras.

Um exemplo de entrada e saída do programa pode ser assim visualizado:

```
Informe uma palavra: laranja
Informe uma palavra: melao
Informe uma palavra: tomate
Informe uma palavra: cereja
Informe uma palavra: uva
Informe uma palavra: banana
Informe uma palavra: maca

Maior palavra: laranja
Menor Palavra: uva
```

14.9 Escreva um programa com nome `reverso.c` que mostra os argumentos da linha de comandos em ordem inversa. Por exemplo, executando o programa da seguinte forma:

```
prompt$ ./reverso garfo e faca
```

deve produzir a seguinte saída:

```
faca e garfo
```

14.10 Escreva um programa com nome `soma.c` que soma todos os argumentos informados na linha de comandos, considerando que todos eles são números inteiros. Por exemplo, executando o programa da seguinte forma:

```
prompt$ ./soma 81 25 2
```

deve produzir a seguinte saída:

```
108
```

AULA 15

# PONTEIROS E REGISTROS

Nesta aula trabalharemos com ponteiros e registros. Primeiro, veremos como declarar e usar ponteiros para registros. Essas tarefas são equivalentes as que já fizemos quando usamos ponteiros para números inteiros, por exemplo. Além disso, vamos adicionar também ponteiros como campos de registros. É muito comum usar registros contendo ponteiros em estruturas de dados poderosas, como listas lineares e árvores, para solução de problemas. Esta aula é baseada nas referências [2, 7].

## 15.1 Ponteiros para registros

Suponha que definimos uma etiqueta de registro `data` como a seguir:

```
struct data {
   int dia;
   int mes;
   int ano;
};
```

A partir dessa definição, podemos declarar variáveis do tipo `struct data`, como abaixo:

```
struct data hoje;
```

E então, assim como fizemos com ponteiros para inteiros, caracteres e números de ponto flutuante, podemos declarar um ponteiro para o registro `data` da seguinte forma:

```
struct data *p;
```

Podemos, a partir dessa declaração, fazer uma atribuição à variável $p$ como a seguir:

```
p = &hoje;
```

Além disso, podemos atribuir valores aos campos do registro de forma indireta, como fazemos abaixo:

```
(*p).dia = 11;
```

Essa atribuição tem o efeito de armazenar o número inteiro 11 no campo `dia` da variável `hoje`, indiretamente através do ponteiro $p$ no entanto. Nessa atribuição, os parênteses envolvendo `*p` são necessários porque o operador `.`, de seleção de campo de um registro, tem maior prioridade que o operador `*` de indireção. É importante relembrar também que essa forma de acesso indireto aos campos de um registro pode ser substituída, e tem o mesmo efeito, pelo operador `->` como mostramos no exemplo abaixo:

```
p->dia = 11;
```

O programa 15.1 ilustra o uso de ponteiros para registros.

Programa 15.1: Uso de um ponteiro para um registro.

```
#include <stdio.h>

struct data {
   int dia;
   int mes;
   int ano;
};

int main(void)
{
   struct data hoje, *p;

   p = &hoje;
   p->dia = 13;
   p->mes = 10;
   p->ano = 2010;
   printf("A data de hoje é %d/%d/%d\n", hoje.dia, hoje.mes, hoje.ano);

   return 0;
}
```

No programa 15.1, há a declaração de duas variáveis: um registro com identificador `hoje` e um ponteiro para registros com identificador $p$. Na primeira atribuição, $p$ recebe o endereço da variável `hoje`. Observe que a variável `hoje` é do tipo `struct data`, isto é, a variável `hoje` é do mesmo tipo da variável $p$ e, portanto, essa atribuição é válida. Em seguida, valores do tipo inteiro são armazenados na variável `hoje`, mas de forma indireta, com uso do ponteiro $p$. Por fim, os valores atribuídos são impressos na saída. A figura 15.1 mostra as variáveis `hoje` e $p$ depois das atribuições realizadas durante a execução do programa 15.1.

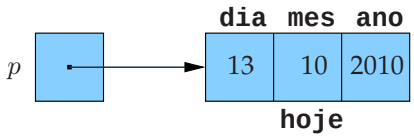



Figura 15.1: Representação do ponteiro $p$ e do registro `hoje`.

## 15.2 Registros contendo ponteiros

Podemos também usar ponteiros como campos de registros. Por exemplo, podemos definir uma etiqueta de registro como abaixo:

```
struct reg_pts {
   int *pt1;
   int *pt2;
};
```

A partir dessa definição, podemos declarar variáveis (registros) do tipo `struct reg_pts` como a seguir:

```
struct reg_pts bloco;
```

Em seguida, a variável `bloco` pode ser usada como sempre fizemos. Note apenas que `bloco` não é um ponteiro, mas um registro que contém dois campos que são ponteiros. Veja o programa 15.2, que mostra o uso dessa variável.

Observe atentamente a diferença entre `(*`$p$`).dia` e `*reg.pt1`. No primeiro caso, $p$ é um ponteiro para um registro e o acesso indireto a um campo do registro, via esse ponteiro, tem de ser feito com a sintaxe `(*`$p$`).dia`, isto é, o conteúdo do endereço contido em $p$ é um registro e, portanto, a seleção do campo é descrita fora dos parênteses. No segundo caso, `reg` é um registro – e não um ponteiro para um registro – e como contém campos que são ponteiros, o acesso ao conteúdo dos campos é realizado através do operador de indireção `*`. Assim, `*reg.pt1` significa que queremos acessar o conteúdo do endereço apontado por `reg.pt1`. Como o operador de seleção de campo `.` de um registro tem prioridade pelo operador de indireção `*`, não há necessidade de parênteses, embora pudéssemos usá-los da forma `*(reg.pt1)`. A figura 15.2 ilustra essa situação.

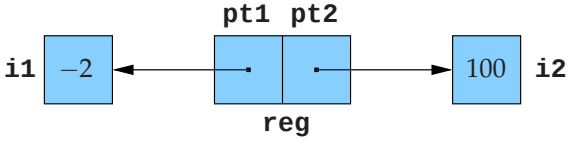



Figura 15.2: Representação do registro `reg` contendo dois campos ponteiros.

Programa 15.2: Uso de um registro que contém campos que são ponteiros.

```
#include <stdio.h>

struct pts_int {
   int *pt1;
   int *pt2;
};

int main(void)
{
   int i1, i2;
   struct pts_int reg;

   i2 = 100;
   reg.pt1 = &i1;
   reg.pt2 = &i2;
   *reg.pt1 = -2;
   printf("i1 = %d, *reg.pt1 = %d\n", i1, *reg.pt1);
   printf("i2 = %d, *reg.pt2 = %d\n", i2, *reg.pt2);

   return 0;
}
```

## Exercícios

15.1 Qual a saída do programa descrito abaixo?

```
#include <stdio.h>

struct dois_valores {
   int vi;
   float vf;
};

int main(void)
{
   struct dois_valores reg1 = {53, 7.112}, reg2, *p = &reg1;

   reg2.vi = (*p).vf;
   reg2.vf = (*p).vi;
   printf("1: %d %f\n2: %d %f\n", reg1.vi, reg1.vf, reg2.vi, reg2.vf);

   return 0;
}
```

15.2 Simule a execução do programa descrito abaixo.

```
#include <stdio.h>

struct pts {
   char *c;
   int *i;
   float *f;
};

int main(void)
{
   char caractere;
   int inteiro;
   float real;
   struct pts reg;

   reg.c = &caractere;
   reg.i = &inteiro;
   reg.f = &real;
   scanf("%c%d%f", reg.c, reg.i, reg.f);
   printf("%c\n%d\n%f\n", caractere, inteiro, real);

   return 0;
}
```

15.3 Simule a execução do programa descrito abaixo.

```
#include <stdio.h>

struct celula {
   int valor;
   struct celula *prox;
};

int main(void)
{
   struct celula reg1, reg2, *p;

   scanf("%d%d", &reg1.valor, &reg2.valor);
   reg1.prox = &reg2;
   reg2.prox = NULL;
   for (p = &reg1; p != NULL; p = p->prox)
      printf("%d ", p->valor);
   printf("\n");

   return 0;
}
```

AULA 16

# USO AVANÇADO DE PONTEIROS

Nas aulas 9 a 15 vimos formas importantes de uso de ponteiros: como parâmetros de funções simulando passagem por referência e como elementos da linguagem C que podem acessar indiretamente outros compartimentos de memória, seja uma variável, uma célula de um vetor ou de uma matriz ou ainda um campo de um registro, usando, inclusive uma aritmética específica para tanto.

Nesta aula, baseada nas referências [2, 7], veremos outros usos para ponteiros: como auxiliares na alocação dinâmica de espaços de memória, como ponteiros para funções e como ponteiros para outros ponteiros.

## 16.1 Ponteiros para ponteiros

Como vimos até aqui, um ponteiro é uma variável cujo conteúdo é um endereço. Dessa forma, podemos acessar o conteúdo de uma posição de memória através de um ponteiro de forma indireta. Temos visto exemplos de ponteiros e suas aplicações em algumas das aulas anteriores e também na seção anterior.

Por outro lado, imagine por um momento que uma variável de um tipo básico qualquer contém um endereço de uma posição de memória que, por sua vez, ela própria também contém um endereço de uma outra posição de memória. Então, essa variável é definida como um **ponteiro para um ponteiro**, isto é, um ponteiro para um compartimento de memória que contém um ponteiro. Ponteiros para ponteiros têm diversas aplicações na linguagem C, especialmente no uso de matrizes, como veremos adiante na seção 16.2. Ponteiros para ponteiros são mais complicados de entender e exigem maturidade para sua manipulação.

O conceito de indireção dupla é então introduzido neste caso: uma variável que contém um endereço de uma posição de memória, isto é, um ponteiro, que, por sua vez, contém um endereço para uma outra posição de memória, ou seja, um outro ponteiro. Certamente, podemos estender indireções com a multiplicidade que desejarmos como, por exemplo, indireção dupla, indireção tripla, indireção quádrupla, etc. No entanto, a compreensão de um programa fica gradualmente mais difícil à medida que indireções múltiplas vão sendo utilizadas. Entender bem um programa com ponteiros para ponteiros, ou ponteiros para ponteiros para ponteiros, e assim por diante, é bastante complicado e devemos usar esses recursos de forma criteriosa. Algumas estruturas de dados, porém, exigem que indireções múltiplas sejam usadas e, portanto, é necessário estudá-las.

Considere agora o programa 16.1.

Programa 16.1: Um exemplo de indireção dupla.

```
#include <stdio.h>

int main(void)
{
   int x, y, *pt1, *pt2, **ptpt1, **ptpt2;

   x = 1;
   y = 4;
   printf("x=%d y=%d\n", x, y);
   pt1 = &x;
   pt2 = &y;
   printf("*pt1=%d *pt2=%d\n", *pt1, *pt2);
   ptpt1 = &pt1;
   ptpt2 = &pt2;
   printf("**ptpt1=%d **ptpt2=%d\n", **ptpt1, **ptpt2);

   return 0;
}
```

Inicialmente, o programa 16.1 faz a declaração de seis variáveis do tipo inteiro: `x` e `y`, que armazenam valores desse tipo; `pt1` e `pt2`, que são ponteiros; e `ptpt1` e `ptpt2` que são ponteiros para ponteiros. O símbolo `**` antes do identificador das variáveis `ptpt1` e `ptpt2` significa que as variáveis são ponteiros para ponteiros, ou seja, podem armazenar um endereço onde se encontra um outro endereço onde, por sua vez, encontra-se um valor, um número inteiro nesse caso.

Após essas declarações, o programa segue com atribuições de valores às variáveis `x` e `y` e com atribuições dos endereços das variáveis `x` e `y` para os ponteiros `pt1` e `pt2`, respectivamente, como já vimos em outros exemplos.

Note então que as duas atribuições abaixo:

```
   ptpt1 = &pt1;
   ptpt2 = &pt2;
```

fazem das variáveis `ptpt1` e `ptpt2` ponteiros para ponteiros. Ou seja, `ptpt1` contém o endereço da variável `pt1` e `ptpt2` contém o endereço da variável `pt2`. Por sua vez, a variável `pt1` contém o endereço da variável `x` e a variável `pt2` contém o endereço da variável `y`, o que caracteriza as variáveis `ptpt1` e `ptpt2` declaradas no programa 16.1 como ponteiros para ponteiros.

Observe finalmente que para acessar o conteúdo do endereço apontado pelo endereço apontado por `ptpt1` temos de usar o símbolo de indireção dupla `**`, como pode ser verificado na última chamada à função `printf` do programa.

Veja a figura 16.1 que ilustra indireção dupla no contexto do programa 16.1.

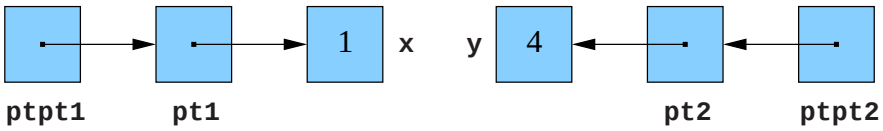



Figura 16.1: Exemplo esquemático de indireção dupla.

## 16.2 Alocação dinâmica de memória

As estruturas de armazenamento de informações na memória principal da linguagem C têm, em geral, tamanho fixo. Por exemplo, uma vez que um programa foi compilado, a quantidade de elementos de um vetor ou de uma matriz é fixa. Isso significa que, para alterar a capacidade de armazenamento de uma estrutura de tamanho fixo, é necessário alterar seu tamanho no arquivo-fonte e compilar esse programa novamente.

Felizmente, a linguagem C permite **alocação dinâmica de memória**, que é a habilidade de reservar espaços na memória principal durante a execução de um programa. Usando alocação dinâmica, podemos projetar estruturas de armazenamento que crescem ou diminuem quando necessário durante a execução do programa. Apesar de disponível para qualquer tipo de dados, a alocação dinâmica é usada em geral com variáveis compostas homogêneas e heterogêneas.

Aprendemos até aqui a declarar uma variável composta homogênea especificando um identificador e sua(s) dimensão(ões).

Por exemplo, veja as declarações a seguir:

```
int vet[100];
float mat[40][60];
```

Nesse caso, temos a declaração de um vetor com identificador `vet` e 100 posições de memória que podem armazenar números inteiros e uma matriz com identificador `mat` de 40 linhas e 60 colunas que são compartimentos de memória que podem armazenar números de ponto flutuante. Todos os compartimentos dessas variáveis compostas homogêneas ficam disponíveis para uso durante a execução do programa.

Em diversas aplicações, para que os dados de entrada sejam armazenados em variáveis compostas homogêneas com dimensão(ões) adequadas, é necessário saber antes essa(s) dimensão(ões), o que é então solicitado a um(a) usuário(a) do programa logo de início. Por exemplo, o(a) usuário(a) da aplicação pode querer informar a quantidade de elementos que será armazenada no vetor `vet`, um valor que será mantido no programa para verificação do limite de armazenamento e que não deve ultrapassar o limite máximo de 100 elementos. Note que a previsão de limitante máximo deve sempre ser especificada também. Do mesmo modo, o(a) usuário(a) pode querer informar, antes de usar essa estrutura, quantas linhas e quantas colunas da matriz `mat` serão usadas, sem ultrapassar o limite máximo de 40 linhas e 60 colunas. Se o(a) usuário(a), por exemplo, usar apenas 10 compartimentos do vetor `vet` ou apenas $3 \times 3$ compartimentos da matriz `mat` durante a execução do programa, os compartimentos restantes não serão usados, embora tenham sido alocados na memória durante suas declarações, ocupando espaço desnecessário na memória do sistema computacional.

Essa alocação que acabamos de descrever, e que conhecemos bem de muito tempo, é chamada de **alocação estática de memória**, o que significa que, no início da execução do programa, quando encontramos uma declaração como essa, ocorre a reserva na memória principal de um número fixo de compartimentos correspondentes ao número especificado na declaração. Esse espaço é fixo e não pode ser alterado durante a execução do programa de forma alguma. Ou seja, não há possibilidade de realocar espaço na memória, nem diminuindo-o nem aumentando-o.

Alocação estática de variáveis e compartimentos não-usados disponíveis na memória podem não ter impacto significativo em programas pequenos, que fazem pouco uso da memória principal, como a grande maioria de nossos programas que desenvolvemos até aqui. No entanto, devemos sempre ter em mente que a memória é um recurso limitado e que em programas maiores e que armazenam muitas informações na memória principal temos, de alguma forma, de usá-la de maneira eficiente, economizando compartimentos sempre que possível. Essa diretriz, infelizmente, não pode ser atingida com uso de alocação estática de compartimentos de memória.

Nesse sentido, se pudéssemos declarar, por exemplo, variáveis compostas homogêneas – vetores e matrizes em particular – com o número exato de compartimentos que serão de fato usados durante a execução do programa, então é evidente que não haveria esse desperdício mencionado.

No exemplo anterior das declarações das variáveis `vet` e `mat`, economizaríamos espaço significativo na memória principal, caso necessitássemos usar apenas 10 compartimentos no vetor `vet` e 9 compartimentos na matriz `mat`, por exemplo, já que ambas foram declaradas com muitos compartimentos mais. No entanto, em uma outra execução subseqüente do mesmo programa que declara essas variáveis, poderiam ser necessárias capacidades diferentes, bem maiores, para ambas as variáveis. Dessa forma, fixar um valor específico baseado na execução do programa torna-se inviável e o que melhor podemos fazer quando usamos alocação estática de memória é prever um limitante máximo para esssas quantidades, conforme vimos fazendo até aqui.

Felizmente para nós, programadores e programadoras da linguagem C, é possível alocar dinamicamente um ou mais blocos de memória na linguagem C. Isso significa que é possível alocar compartimentos de memória durante a execução do programa, com base nas demandas do(a) usuário(a).

**Alocação dinâmica de memória** significa que um programa solicita ao sistema computacional, durante a sua execução, blocos da memória principal que estejam disponíveis para uso naquele momento. Caso haja espaço suficiente disponível na memória principal, a solicitação é atendida e o espaço solicitado fica reservado na memória para aquele uso específico. Caso contrário, isto é, caso não haja possibilidade de atender a solicitação de espaço, uma mensagem de erro em tempo de execução é emitida para o(a) usuário(a) do programa. O(a) programador(a) tem de estar preparado para esse tipo de situação, incluindo testes de verificação de situações como essa no arquivo-fonte, informando também ao(à) usuário(a) do programa sobre a situação de impossibilidade de uso do espaço solicitado de memória. O(a) programador(a) deve solicitar ainda alguma decisão do(a) usuário(a) quando dessa circunstância e, provavelmente, encerrar a execução do programa.

Vejamos um exemplo no programa 16.2 que faz alocação dinâmica de memória para alocação de um vetor.

Programa 16.2: Um exemplo de alocação dinâmica de memória.

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

int main(void)
{
   int i, n, *vetor, *pt;

   printf("Informe a dimensão do vetor: ");
   scanf("%d", &n);

   vetor = (int *) malloc(n * sizeof(int));
   if (vetor != NULL) {
      for (i = 0; i < n; i++) {
         printf("Informe o elemento %d: ", i);
         scanf("%d", (vetor + i));
      }

      printf("\nVetor         : ");
      for (pt = vetor; pt < (vetor + n); pt++)
         printf("%d ", *pt);

      printf("\nVetor invertido: ");
      for (i = n - 1; i >= 0; i--)
         printf("%d ", vetor[i]);
      printf("\n");

      free(vetor);
   }
   else
      printf("Impossível alocar o espaço requisitado\n");

   return 0;
}
```

Na segunda linha do programa 16.2, incluímos o arquivo-cabeçalho `stdlib.h`, que contém a declaração da função `malloc`, usada nesse programa. A função `malloc` tem sua interface apresentada a seguir:

```
void *malloc(size_t tamanho)
```

A função `malloc` reserva uma certa quantidade específica de memória e devolve um ponteiro do tipo `void`. No programa acima, reservamos $n$ compartimentos contínuos que podem armazenar números inteiros, o que se reflete na expressão `n * sizeof(int)`. Essa expressão, na verdade, reflete o número de bytes que serão alocados continuamente na memória, que depende do sistema computacional onde o programa é compilado. O operador unário `sizeof` devolve como resultado o número de bytes dados pelo seu operando, que pode ser um tipo de dados ou uma expressão. Nesse caso, nas máquinas que usamos no laboratório, a expressão `sizeof(int)` devolve 4 bytes. Esse número é multiplicado por $n$, o número de compartimentos que desejamos para armazenar números inteiros. O endereço da primeira posição de

memória onde encontram-se esses compartimentos é devolvido pela função `malloc`. Essa função devolve um ponteiro do tipo `void`. Por isso, usamos o modificador de tipo `(int *)` para indicar que o endereço devolvido é de fato um ponteiro para um número inteiro. Por fim, esse endereço é armazenado em `vetor`, que foi declarado como um ponteiro para números inteiros. A partir daí, podemos usar `vetor` da forma como preferirmos, como um ponteiro ou como um vetor.

O programa 16.3 é um exemplo de alocação dinâmica de memória de uma matriz de números inteiros. Esse programa é um pouco mais complicado que o programa 16.2, devido ao uso distinto que faz da função `malloc`.

Programa 16.3: Um exemplo de alocação dinâmica de uma matriz.

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

int main(void)
{
   int i, j, m, n, **matriz, **pt;

   printf("Informe a dimensão da matriz: ");
   scanf("%d%d", &m, &n);
   matriz = (int **) malloc(m * sizeof(int *));
   if (matriz == NULL) {
      printf("Não há espaço suficiente na memória\n");
      return 0;
   }

   for (pt = matriz, i = 0; i < m; i++, pt++) {
      *pt = (int *) malloc(n * sizeof(int));
      if (*pt == NULL) {
         printf("Não há espaço suficiente na memória\n");
         return 0;
      }
   }

   for (i = 0; i < m; i++)
      for (j = 0; j < n; j++) {
         printf("Informe o elemento (%d,%d): ", i, j);
         scanf("%d", &matriz[i][j]);
      }
   printf("\nMatriz:\n");
   pt = matriz;
   for (i = 0; i < m; i++) {
      for (j = 0; j < n; j++)
         printf("%d ", *(*(pt+i)+j));
      printf("\n");
   } printf("\n");

   for (i = 0; i < m; i++)
      free(matriz[i]);
   free(matriz);

   return 0;
}
```

Observe por fim que as linhas em que ocorrem a alocação dinâmica de memória no programa 16.3 podem ser substituídas de forma equivalente pelas linhas a seguir:

```
matriz = (int **) malloc(m * sizeof(int *));
for (i = 0; i < m; i++)
   matriz[i] = (int *) malloc(n * sizeof(int));
```

Além da função `malloc` existem duas outras funções para alocação de memória na `stdlib.h`: `calloc` e `realloc`, mas a primeira é mais freqüentemente usada que essas outras duas. Essas funções solicitam blocos de memória de um espaço de armazenamento conhecido também como ***heap*** ou ainda **lista de espaços disponíveis**. A chamada freqüente dessas funções pode exaurir o *heap* do sistema, fazendo com que essas funções devolvam um ponteiro nulo. Pior ainda, um programa pode alocar blocos de memória e perdê-los de algum modo, gastando espaço desnecessário. Considere o exemplo a seguir:

```
p = malloc(...);
q = malloc(...);
p = q;
```

Após as duas primeiras sentenças serem executadas, $p$ aponta para um bloco de memória e $q$ aponta para um outro. No entanto, após a atribuição de $q$ para $p$ na última sentença, as duas variáveis apontam para o mesmo bloco de memória, o segundo bloco, sem que nenhuma delas aponte para o primeiro. Além disso, não poderemos mais acessar o primeiro bloco, que ficará perdido na memória, ocupando espaço desnecessário. Esse bloco é chamado de **lixo**.

A função `free` é usada para ajudar os programadores da linguagem C a resolver o problema de geração de lixo na memória durante a execução de programas. Essa função tem a seguinte interface na `stdlib.h`:

```
void free(void *pt)
```

Para usar a função `free` corretamente, devemos lhe passar um ponteiro para um bloco de memória que não mais necessitamos, como fazemos abaixo:

```
p = malloc(...);
q = malloc(...);
free(p);
p = q;
```

A chamada à função `free` devolve o bloco de memória apontado por $p$ para o *heap*, que fica disponível para uso em chamadas subseqüentes das funções de alocação de memória.

O argumento da função `free` deve ser um ponteiro que foi previamente devolvido por uma função de alocação de memória.

## 16.3 Ponteiros para funções

Como vimos até este ponto, ponteiros podem conter endereços de variáveis de tipos básicos, de elementos de vetores e matrizes, de campos de registros ou de registros inteiros. Um ponteiro pode também apontar para outro ponteiro. Um ponteiro pode ainda ser usado para alocação dinâmica de memória. No entanto, a linguagem C não requer que ponteiros contenham apenas endereços de dados. É possível, em um programa, ter ponteiros para funções, já que as funções ocupam posições de memória e, por isso, possuem um endereço na memória, assim como todas as variáveis.

Podemos usar ponteiros para funções assim como usamos ponteiros para variáveis. Em particular, passar um ponteiro para uma função como um argumento de outra função é bastante comum em programas da linguagem C.

Suponha que estamos escrevendo a função `integral` que integra uma função matemática `f` entre os pontos `a` e `b`. Gostaríamos de fazer a função `integral` tão geral quanto possível, passando a função `f` como um argumento seu. Isso é possível na linguagem C pela definição de `f` como um ponteiro para uma função. Considerando que queremos integrar funções que têm um parâmetro do tipo `double` e que devolvem um valor do tipo `double`, uma possível interface da função `integral` é apresentada a seguir:

```
double integral(double (*f)(double), double a, double b)
```

Os parênteses em torno de `*f` indicam que `f` é um ponteiro para uma função, não uma função que devolve um ponteiro.

Também é permitido definir `f` como se fosse uma função:

```
double integral(double f(double), double a, double b)
```

Do ponto de vista do compilador, não há diferença alguma entre as interfaces apresentadas acima.

Quando chamamos a função `integral` devemos fornecer um nome de uma função como primeiro argumento. Por exemplo, a chamada a seguir integra a função seno de $0$ a $\pi/2$:

```
result = integral(sin, 0.0, PI / 2);
```

O argumento `sin` é o nome/identificador da função seno, que foi incluída no programa através da biblioteca `math.h`.

Observe que não há parênteses após o identificador da função `sin`. Quando o nome de uma função não é seguido por parênteses, o compilador produz um ponteiro para a função em vez de gerar código para uma chamada da função. No exemplo acima, não há uma chamada à função `sin`. Ao invés disso, estamos passando para a função `integral` um ponteiro para

a função `sin`. Podemos pensar em ponteiros para funções como pensamos com ponteiros para vetores e matrizes. Relembrando, se, por exemplo, $v$ é o identificador de um vetor, então `v[i]` representa um elemento do vetor enquanto que $v$ representa um ponteiro para o vetor, ou melhor, para o primeiro elemento do vetor. Da forma similar, se `f` é o identificador de uma função, a linguagem C trata `f(x)` como uma chamada da função, mas trata `f` como um ponteiro para a função.

Dentro do corpo da função `integral` podemos chamar a função apontada por `f` da seguinte forma:

```
y = (*f)(x);
```

Nessa chamada, `*f` representa a função apontada por `f` e `x` é o argumento dessa chamada. Assim, durante a execução da chamada `integral(sin, 0.0, PI / 2)`, cada chamada de `*f` é, na verdade, uma chamada de `sin`.

Também podemos armazenar ponteiros para funções em variáveis ou usá-los como elementos de um vetor, de uma matriz, de um campo de um registro. Podemos ainda escrever funções que devolvem ponteiros para funções. Como um exemplo, declaramos abaixo uma variável que pode armazenar um ponteiro para uma função:

```
void (*ptf)(int);
```

O ponteiro `ptf` pode apontar para qualquer função que tenha um único parâmetro do tipo `int` e que devolva um valor do tipo `void`.

Se `f` é uma função com essas características, podemos fazer `ptf` apontar para `f` da seguinte forma:

```
ptf = f;
```

Observe que não há operador de endereçamento antes de `f`.

Uma vez que `ptf` aponta para `f`, podemos chamar `f` indiretamente através de `ptf` como a seguir:

```
(*ptf)(i);
```

O programa 16.4 imprime uma tabela mostrando os valores das funções seno, cosseno e tangente no intervalo e incremento escolhidos pelo(a) usuário(a). O programa usa as funções `sin`, `cos` e `tan` de `math.h`.

Programa 16.4: Um exemplo de ponteiro para função.

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

/* Recebe um ponteiro para uma função trigonométrica, um inter-
   valo de valores reais e um incremento real e imprime o valor
   (real) da integral da função trigométrica neste intervalo */
void tabela(double (*f)(double), double a, double b, double incr)
{
   int i, num_intervalos;
   double x;

   num_intervalos = ceil((b - a) / incr);
   for (i = 0; i <= num_intervalos; i++) {
      x = a + i * incr;
      printf("%11.6f %11.6f\n", x, (*f)(x));
   }
}

/* Imprime a integral de funções trigonométricas, dados
   um intervalo de números reais e um incremento real */
int main(void)
{
   double inicio, fim, incremento;

   printf("Informe um intervalo [a, b]: ");
   scanf("%lf%lf", &inicio, &fim);
   printf("Informe o incremento: ");
   scanf("%lf", &incremento);

   printf("\n      x         cos(x)"
          "\n    ------    ------\n");
   tabela(cos, inicio, fim, incremento);
   printf("\n      x         sen(x)"
          "\n    ------    ------\n");
   tabela(sin, inicio, fim, incremento);
   printf("\n      x         tan(x)"
          "\n    ------    ------\n");
   tabela(tan, inicio, fim, incremento);

   return 0;
}
```

## Exercícios

16.1 Dados dois vetores $x$ e $y$, ambos com $n$ elementos, $1 \leqslant n \leqslant 100$, determinar o produto escalar desses vetores. Use alocação dinâmica de memória.

16.2 Dizemos que uma seqüência de $n$ elementos, com $n$ par, é **balanceada** se as seguintes somas são todas iguais:

a soma do maior elemento com o menor elemento;
a soma do segundo maior elemento com o segundo menor elemento;
a soma do terceiro maior elemento com o terceiro menor elemento;
e assim por diante ...

Exemplo:

$2\ \ 12\ \ 3\ \ 6\ \ 16\ \ 15$ é uma seqüência balanceada, pois $16 + 2 = 15 + 3 = 12 + 6$.

Dados $n$ ($n$ par e $0 \leqslant n \leqslant 100$) e uma seqüência de $n$ números inteiros, verificar se essa seqüência é balanceada. Use alocação dinâmica de memória.

16.3 Dada uma cadeia de caracteres com no máximo 100 caracteres, contar a quantidade de letras minúsculas, letras maiúsculas, dígitos, espaços e símbolos de pontuação que essa cadeia possui. Use alocação dinâmica de memória.

16.4 Dada uma matriz de números reais $A$ com $m$ linhas e $n$ colunas, $1 \leqslant m, n \leqslant 100$, e um vetor de números reais $v$ com $n$ elementos, determinar o produto de $A$ por $v$. Use alocação dinâmica de memória.

16.5 Dizemos que uma matriz quadrada de números inteiros distintos é um **quadrado mágico** se a soma dos elementos de cada linha, a soma dos elementos de cada coluna e a soma dos elementos da diagonal principal e secundária são todas iguais.

Exemplo:

A matriz

$$\begin{pmatrix} 8 & 0 & 7 \\ 4 & 5 & 6 \\ 3 & 10 & 2 \end{pmatrix}$$

é um quadrado mágico.

Dada uma matriz quadrada de números inteiros $A_{n \times n}$, com $1 \leqslant n \leqslant 100$, verificar se $A$ é um quadrado mágico. Use alocação dinâmica de memória.

16.6 Simule a execução do programa a seguir.

```
#include <stdio.h>

int f1(int (*f)(int))
{
   int n = 0;

   while ((*f)(n))
      n++;
   return n;
}

int f2(int i)
{
   return i * i + i - 12;
}

int main(void)
{
   printf("Resposta: %d\n", f1(f2));
   return 0;
}
```

16.7 Escreva uma função com a seguinte interface:

```
int soma(int (*f)(int), int inicio, int fim)
```

Uma chamada `soma(g, i, j)` deve devolver `g(i) + ... + g(j)`.

AULA 17

# ARQUIVOS

Nos programas que fizemos até aqui, a entrada e a saída de dados sempre ocorreram em uma janela de terminal ou console do sistema operacional. Na linguagem C, não existem palavras-chaves definidas para tratamento de operações de entrada e saída. Essas tarefas são realizadas através de funções. Pouco usamos outras funções de entrada e saída da linguagem C além das funções `scanf` e `printf`, localizadas na biblioteca padrão de entrada e saída, com arquivo-cabeçalho `stdio.h`. Isso significa que o fluxo de dados de entrada e saída dos programas que fizemos sempre passa pela memória principal. Nesta aula, baseada na referência [7], aprenderemos funções que realizam tarefas de entrada e saída armazenados em um dispositivo de memória secundária em arquivos.

## 17.1 Seqüências de caracteres

Na linguagem C, o termo **seqüência de caracteres**, do inglês *stream*, significa qualquer fonte de entrada ou qualquer destinação para saída de informações. Os programas que produzimos até aqui sempre obtiveram toda sua entrada a partir de uma seqüência de caracteres, em geral associada ao teclado, e escreveram sua saída em outra seqüência de caracteres, associada com o monitor.

Programas maiores podem necessitar de seqüências de caracteres adicionais, associadas a arquivos armazenados em uma variedade de meios físicos tais como discos e memórias ou ainda portas de rede e impressoras. As funções de entrada e saída mantidas em `stdio.h` trabalham do mesmo modo com todas as seqüências de caracteres, mesmo aquelas que não representam arquivos físicos.

O acesso a uma seqüência de caracteres na linguagem C se dá através de um **ponteiro de arquivo**, cujo tipo é `FILE *`, declarado em `stdio.h`. Algumas seqüências de caracteres são representadas por ponteiros de arquivos com nomes padronizados, como veremos a seguir. Podemos ainda declarar ponteiros de arquivos conforme nossas necessidades, como fazemos abaixo:

```
FILE *pt1, *pt2;
```

Veremos mais sobre arquivos na linguagem C e as funções de suporte associadas a eles a partir da seção 17.3. Em particular, veremos arquivos do sistema com nomes padronizados na seção 17.3.5.

A biblioteca representada pelo arquivo-cabeçalho `stdio.h` suporta dois tipos de arquivos: texto e binário. Os bytes em um **arquivo-texto** representam caracteres, fazendo que seja possível examinar e editar o seu conteúdo. O arquivo-fonte de um programa na linguagem C, por exemplo, é armazenado em um arquivo-texto. Por outro lado, os bytes em um **arquivo-binário** não representam necessariamente caracteres. Grupos de bytes podem representar outros tipos de dados tais como inteiros e números com ponto flutuante. Um programa executável, por exemplo, é armazenado em um arquivo-binário.

Arquivos-texto possuem duas principais características que os diferem dos arquivos-binários: são divididos em linhas e podem conter um marcador especial de fim de arquivo. Cada linha do arquivo-texto normalmente termina com um ou dois caracteres especiais, dependendo do sistema operacional.

## 17.2 Redirecionamento de entrada e saída

Como já fizemos nos trabalhos da disciplina e em algumas aulas, a leitura e escrita em arquivos podem ser facilmente executadas nos sistemas operacionais em geral. Como percebemos, nenhum comando especial teve de ser adicionado aos nossos programas na linguagem C para que a leitura e a escrita fossem executadas de/para arquivos. O que fizemos até aqui foi redirecionar a entrada e/ou a saída de dados do programa. Como um exemplo simples, vejamos o programa 17.1, onde um número inteiro na base decimal é fornecido como entrada e na saída é apresentado o mesmo número na base binária.

Programa 17.1: Conversão de um número inteiro na base decimal para base binária.

```
#include <stdio.h>

int main(void)
{
   int pot10, numdec, numbin;

   scanf("%d", &numdec);
   pot10 = 1;
   numbin = 0;
   while (numdec > 0) {
      numbin = numbin + (numdec % 2) * pot10;
      numdec = numdec / 2;
      pot10 = pot10 * 10;
   }
   printf("%d\n", numbin);

   return 0;
}
```

Supondo que o programa 17.1 tenha sido armazenado no arquivo-fonte `decbin.c` e o programa executável equivalente tenha sido criado após a compilação com o nome `decbin`, então, se queremos que a saída do programa executável `decbin` seja armazenada no arquivo `resultado`, podemos digitar em uma linha de terminal o seguinte:

```
prompt$ ./decbin > resultado
```

Esse comando instrui o sistema operacional a executar o programa `decbin` redirecionando sua saída, que normalmente seria apresentada no terminal, para um arquivo com nome `resultado`. Dessa forma, qualquer informação a ser apresentada por uma função de saída, como `printf`, não será mostrada no terminal, mas será escrita no arquivo `resultado`, conforme especificado na linha de comandos do terminal.

Por outro lado, podemos redirecionar a entrada de um programa executável, de tal forma que chamadas a funções que realizam entrada de dados, não mais solicitem essas informações ao usuário a partir do terminal, mas as obtenha a partir de um arquivo. Por exemplo, o programa 17.1 usa a função `scanf` para ler um número inteiro a partir de um terminal. Podemos redirecionar a entrada do programa `decbin` quando está sendo executado, fazendo que essa entrada seja realizada a partir de um arquivo. Por exemplo, se temos um arquivo com nome `numero` que contém um número inteiro, podemos digitar o seguinte em uma linha de terminal:

```
prompt$ ./decbin < numero
```

Com esse redirecionamento, o programa 17.1, que solicita um número a ser informado pelo usuário, não espera até que um número seja digitado. Ao contrário, pelo redirecionamento, a entrada do programa é tomada do arquivo `numero`. Ou seja, a chamada à função `scanf` tem o efeito de ler um valor do arquivo `numero` e não do terminal, embora a função `scanf` não "saiba" disso.

Podemos redirecionar a entrada e a saída de um programa simultaneamente da seguinte maneira:

```
prompt$ ./decbin < numero > resultado
```

Observe então que esse comando faz com que o programa `decbin` seja executado tomando a entrada de dados a partir do arquivo `numero` e escrevendo a saída de dados no arquivo `resultado`.

O redirecionamento de entrada e saída é uma ferramenta útil, já que, podemos manter um arquivo de entradas e realizar diversos testes sobre um programa executável a partir desse arquivo de entradas. Além disso, se temos, por exemplo, um arquivo alvo que contém soluções correspondentes às entradas, podemos comparar esse arquivo de soluções com as saídas de nosso programa, que também podem ser armazenadas em um arquivo diferente. Utilitários para comparação de conteúdos de arquivos disponíveis no sistema operacional Linux, tais como `diff` e `cmp`, são usados para atingir esse objetivo. Por exemplo,

```
prompt$ diff resultado solucao
```

## 17.3 Funções de entrada e saída da linguagem C

Até esta aula, excluindo o redirecionamento de entrada e saída que vimos na seção 17.2, tínhamos sempre armazenado quaisquer informações na memória principal do nosso sistema computacional. Como já mencionado, uma grande quantidade de problemas pode ser resolvida com as operações de entrada e saída que conhecemos e com o redirecionamento de entrada e saída que acabamos de aprender na seção 17.2. Entretanto, existem problemas onde há necessidade de obter, ou armazenar, dados de/para dois ou mais arquivos. Os arquivos são mantidos em um dispositivo do sistema computacional conhecido como memória secundária, que pode ser implementado como um disco rígido, um disquete, um disco compacto (CD), um disco versátil digital (DVD), um cartão de memória, um disco removível (*USB flash memory*), entre outros. A linguagem C tem um conjunto de funções específicas para tratamento de arquivos que se localiza na biblioteca padrão de entrada e saída, cujo arquivo-cabeçalho é `stdio.h`. Na verdade, essa biblioteca contém todas as funções que fazem tratamento de qualquer tipo de entrada e saída de um programa, tanto da memória principal quanto secundária. Como exemplo, as funções `printf`, `scanf`, `putchar` e `getchar`, que conhecemos bem, encontram-se nessa biblioteca.

### 17.3.1 Funções de abertura e fechamento

Para que se possa realizar qualquer operação sobre um arquivo é necessário, antes de tudo, de **abrir** esse arquivo. Como um programa pode ter de trabalhar com diversos arquivos, então todos eles deverão estar abertos durante a execução desse programa. Dessa forma, há necessidade de identificação de cada arquivo, o que é implementado na linguagem C com o uso de um **ponteiro para arquivo**.

A função `fopen` da biblioteca padrão de entrada e saída da linguagem C é uma função que realiza a abertura de um arquivo no sistema. A função recebe como parâmetros duas cadeias de caracteres: a primeira é o identificador/nome do arquivo a ser aberto e a segunda determina o modo no qual o arquivo será aberto. O nome do arquivo deve constar no sistema de arquivos do sistema operacional. A função `fopen` devolve um ponteiro único para o arquivo que pode ser usado para identificar esse arquivo a partir desse ponto do programa. Esse ponteiro é posicionado no início ou no final do arquivo, dependendo do modo como o arquivo foi aberto. Se o arquivo não puder ser aberto por algum motivo, a função devolve o ponteiro com valor `NULL`. Um ponteiro para um arquivo deve ser declarado com um tipo pré-definido `FILE`, também incluso no arquivo-cabeçalho `stdio.h`. A interface da função `fopen` é então descrita da seguinte forma:

```
FILE *fopen(char *nome, char *modo)
```

As opções para a cadeia de caracteres `modo`, parâmetro da função `fopen`, são descritas na tabela a seguir:

| modo | Descrição |
|---|---|
| `r` | modo de leitura de texto |
| `w` | modo de escrita de texto† |
| `a` | modo de adicionar texto‡ |
| `r+` | modo de leitura e escrita de texto |
| `w+` | modo de leitura e escrita de texto† |
| `a+` | modo de leitura e escrita de texto‡ |
| `rb` | modo de leitura em binário |
| `wb` | modo de escrita em binário† |
| `ab` | modo de adicionar em binário‡ |
| `r+b` ou `rb+` | modo de leitura e escrita em binário |
| `w+b` ou `wb+` | modo de leitura e escrita em binário† |
| `a+b` ou `ab+` | modo de leitura e escrita em binário‡ |

onde:

† trunca o arquivo existente com tamanho 0 ou cria novo arquivo;

‡ abre ou cria o arquivo e posiciona o ponteiro no final do arquivo.

Algumas observações sobre os modos de abertura de arquivos se fazem necessárias. Primeiro, observe que se o arquivo não existe e é aberto com o modo de leitura ( `r` ) então a abertura falha. Também, se o arquivo é aberto com o modo de adicionar ( `a` ), então todas as operações de escrita ocorrem o final do arquivo, desconsiderando a posição atual do ponteiro do arquivo. Por fim, se o arquivo é aberto no modo de atualização ( `+` ) então a operação de escrita não pode ser imediatamente seguida pela operação de leitura, e vice-versa, a menos que uma operação de reposicionamento do ponteiro do arquivo seja executada, tal como uma chamada a qualquer uma das funções `fseek` , `fsetpos` , `rewind` ou `fflush` .

Por exemplo, o comando de atribuição a seguir

```
ptarq = fopen("entrada", "r");
```

tem o efeito de abrir um arquivo com nome `entrada` no modo de leitura. A chamada à função `fopen` devolve um identificador para o arquivo aberto que é atribuído ao ponteiro `ptarq` do tipo `FILE` . Então, esse ponteiro é posicionado no primeiro caracter do arquivo. A declaração prévia do ponteiro `ptarq` deve ser feita da seguinte forma:

```
FILE *ptarq;
```

A função `fclose` faz o oposto que a função `fopen` faz, ou seja, informa o sistema que o programador não necessita mais usar o arquivo. Quando um arquivo é fechado, o sistema realiza algumas tarefas importantes, especialmente a escrita de quaisquer dados que o sistema possa ter mantido na memória principal para o arquivo na memória secundária, e então dissocia o identificador do arquivo. Depois de fechado, não podemos realizar tarefas de leitura ou escrita no arquivo, a menos que seja reaberto. A função `fclose` tem a seguinte interface:

```
int fclose(FILE *ptarq)
```

Se a operação de fechamento do arquivo apontado por `ptarq` obtém sucesso, a função `fclose` devolve o valor 0 (zero). Caso contrário, o valor `EOF` é devolvido.

### 17.3.2 Funções de entrada e saída

A função `fgetc` da biblioteca padrão de entrada e saída da linguagem C permite que um único caracter seja lido de um arquivo. A interface dessa função é apresentada a seguir:

```
int fgetc(FILE *ptarq)
```

A função `fgetc` lê o próximo caracter do arquivo apontado por `ptarq`, avançando esse ponteiro em uma posição. Se a leitura é realizada com sucesso, o caracter lido é devolvido pela função. Note, no entanto, que a função, ao invés de especificar o valor de devolução como sendo do tipo `unsigned char`, especifica-o como sendo do tipo `int`. Isso se deve ao fato de que a leitura pode falhar e, nesse caso, o valor devolvido é o valor armazenado na constante simbólica `EOF`, definida no arquivo-cabeçalho `stdio.h`. O valor correspondente à constante simbólica `EOF` é obviamente um valor diferente do valor de qualquer caracter e, portanto, um valor negativo. Do mesmo modo, se o fim do arquivo é encontrado, a função `fgetc` também devolve `EOF`.

A função `fputc` da biblioteca padrão de entrada e saída da linguagem C permite que um único caracter seja escrito em um arquivo. A interface dessa função é apresentada a seguir:

```
int fputc(int caracter, FILE *ptarq)
```

Se a função `fputc` tem sucesso, o ponteiro `ptarq` é incrementado e o caracter escrito é devolvido. Caso contrário, isto é, se ocorre um erro, o valor `EOF` é devolvido.

Existe outro par de funções de leitura e escrita em arquivos com identificadores `fscanf` e `fprintf`. As interfaces dessas funções são apresentadas a seguir:

```
int fscanf(FILE *ptarq, char *formato, ...)
```

e

```
int fprintf(FILE *ptarq, char *formato, ...)
```

Essas duas funções são semelhantes às respectivas funções `scanf` e `printf` que conhecemos bem, a menos de um parâmetro a mais que é informado, justamente o primeiro, que é

o ponteiro para o arquivo que ser quer realizar as operações de entrada e saída formatadas. Dessa forma, o exemplo de chamada a seguir

```
fprintf(ptarq, "O número %d é primo\n", numero);
```

realiza a escrita no arquivo apontado por `ptarq` da mensagem entre aspas duplas, substituindo o valor numérico correspondente armazenado na variável `numero`. O número de caracteres escritos no arquivo é devolvido pela função `fprintf`. Se um erro ocorrer, então o valor `-1` é devolvido.

Do mesmo modo,

```
fscanf(ptarq, "%d", &numero);
```

realiza a leitura de um valor que será armazenado na variável `numero` a partir de um arquivo identificado pelo ponteiro `ptarq`. Se a leitura for realizada com sucesso, o número de valores lidos pela função `fscanf` é devolvido. Caso contrário, isto é, se houver falha na leitura, o valor `EOF` é devolvido.

Há outras funções para entrada e saída de dados a partir de arquivos, como as funções `fread` e `fwrite`, que não serão cobertas nesta aula. O leitor interessado deve procurar as referências bibliográficas do curso.

### 17.3.3 Funções de controle

Existem diversas funções de controle que dão suporte a operações sobre os arquivos. Dentre as mais usadas, listamos uma função que descarrega o espaço de armazenamento temporário da memória principal para a memória secundária e as funções que tratam do posicionamento do ponteiro do arquivo.

A função `fflush` faz a descarga de qualquer informação associada ao arquivo que esteja armazenada na memória principal para o dispositivo de memória secundária associado. A função `fflush` tem a seguinte interface:

```
int fflush(FILE *ptarq)
```

onde `ptarq` é o ponteiro para um arquivo. Se o ponteiro `ptarq` contém um valor nulo, então todos espaços de armazenamento temporários na memória principal de todos os arquivos abertos são descarregados nos dispositivos de memória secundária. Se a descarga é realizada com sucesso, a função devolve o valor 0 (zero). Caso contrário, a função devolve o valor `EOF`.

Sobre as funções que tratam do posicionamento do ponteiro de um arquivo, existe uma função específica da biblioteca padrão da linguagem C que realiza um teste de final de arquivo. Essa função tem identificador `feof` e a seguinte interface:

```
int feof(FILE *ptarq)
```

O argumento da função `feof` é um ponteiro para um arquivo do tipo `FILE`. A função devolve um valor inteiro diferente de 0 (zero) se o ponteiro `ptarq` está posicionado no final do arquivo. Caso contrário, a função devolve o valor 0 (zero).

A função `fgetpos` determina a posição atual do ponteiro do arquivo e tem a seguinte interface:

```
int fgetpos(FILE *ptarq, fpos_t *pos)
```

onde `ptarq` é o ponteiro associado a um arquivo e `pos` é uma variável que, após a execução dessa função, conterá o valor da posição atual do ponteiro do arquivo. Observe que `fpos_t` é um novo tipo de dado, definido no arquivo-cabeçalho `stdio.h`, adequado para armazenamento de uma posição qualquer de um arquivo. Se a obtenção dessa posição for realizada com sucesso, a função devolve 0 (zero). Caso contrário, a função devolve um valor diferente de 0 (zero).

A função `fsetpos` posiciona o ponteiro de um arquivo em alguma posição escolhida e tem a seguinte interface:

```
int fsetpos(FILE *ptarq, fpos_t *pos)
```

onde `ptarq` é o ponteiro para um arquivo e `pos` é uma variável que contém a posição para onde o ponteiro do arquivo será deslocada. Se a determinação dessa posição for realizada com sucesso, a função devolve 0 (zero). Caso contrário, a função devolve um valor diferente de 0 (zero).

A função `ftell` determina a posição atual de um ponteiro em um dado arquivo. Sua interface é apresentada a seguir:

```
long int ftell(FILE *ptarq)
```

A função `ftell` devolve a posição atual no arquivo apontado por `ptarq`. Se o arquivo é binário, então o valor é o número de *bytes* a partir do início do arquivo. Se o arquivo é de texto, então esse valor pode ser usado pela função `fseek`, como veremos a seguir. Se há sucesso na sua execução, a função devolve a posição atual no arquivo. Caso contrário, a função devolve o valor `-1L`.

A função `fseek` posiciona o ponteiro de um arquivo para uma posição determinada por um deslocamento. A interface da função é dada a seguir:

```
int fseek(FILE *ptarq, long int desloca, int a_partir)
```

O argumento `ptarq` é o ponteiro para um arquivo. O argumento `desloca` é o número de *bytes* a serem saltados a partir do conteúdo do argumento `apartir`. Esse conteúdo pode ser um dos seguintes valores pré-definidos no arquivo-cabeçalho `stdio.h`:

| | |
|---|---|
| `SEEK_SET` | A partir do início do arquivo |
| `SEEK_CUR` | A partir da posição atual |
| `SEEK_END` | A partir do fim do arquivo |

Em um arquivo de texto, o contéudo de `apartir` dever ser `SEEK_SET` e o conteúdo de `desloca` deve ser 0 (zero) ou um valor devolvido pela função `ftell`. Se a função é executada com sucesso, o valor 0 (zero) é devolvido. Caso contrário, um valor diferente de 0 (zero) é devolvido.

A função `rewind` faz com que o ponteiro de um arquivo seja posicionado para o início desse arquivo. A interface dessa função é a seguinte:

```
void rewind(FILE *ptarq)
```

### 17.3.4 Funções sobre arquivos

Há funções na linguagem C que permitem que um programador remova um arquivo do disco ou troque o nome de um arquivo. A função `remove` tem a seguinte interface:

```
int remove(char *nome)
```

A função `remove` elimina um arquivo, com nome armazenado na cadeia de caracteres `nome`, do sistema de arquivos do sistema computacional. O arquivo não deve estar aberto no programa. Se a remoção é realizada, a função devolve o valor 0 (zero). Caso contrário, um valor diferente de 0 (zero) é devolvido.

A função `rename` tem a seguinte interface:

```
int rename(char *antigo, char *novo)
```

A função `rename` faz com que o arquivo com nome armazenado na cadeia de caracteres `antigo` tenha seu nome trocado pelo nome armazenado na cadeia de caracteres `novo`. Se a função realiza a tarefa de troca de nome, então `rename` devolve o valor 0 (zero). Caso contrário, um valor diferente de 0 (zero) é devolvido e o arquivo ainda pode ser identificado por seu nome antigo.

### 17.3.5 Arquivos do sistema

Sempre que um programa na linguagem C é executado, três arquivos ou seqüências de caracteres são automaticamente abertos pelo sistema, identificados pelos ponteiros `stdin`,

`stdout` e `stderr`, definidos no arquivo-cabeçalho `stdio.h` da biblioteca padrão de entrada e saída. Em geral, `stdin` está associado ao teclado e `stdout` e `stderr` estão associados ao monitor. Esses ponteiros são todos do tipo `FILE`.

O ponteiro `stdin` identifica a entrada padrão do programa e é normalmente associado ao teclado. Todas as funções de entrada definidas na linguagem C que executam entrada de dados e não têm um ponteiro do tipo `FILE` como um argumento tomam a entrada a partir do arquivo apontado por `stdin`. Assim, ambas as chamadas a seguir:

```
scanf("%d", &numero);
```

e

```
fscanf(stdin, "%d", &numero);
```

são equivalentes e lêem um número do tipo inteiro da entrada padrão, que é normalmente o terminal.

Do mesmo modo, o ponteiro `stdout` se refere à saída padrão, que também é associada ao terminal. Assim, as chamadas a seguir:

```
printf("Programar é bacana!\n");
```

e

```
fprintf(stdout, "Programa é bacana!\n");
```

são equivalentes e imprimem a mensagem acima entre as aspas duplas na saída padrão, que é normalmente o terminal.

O ponteiro `stderr` se refere ao arquivo padrão de erro, onde muitas das mensagens de erro produzidas pelo sistema são armazenadas e também é normalmente associado ao terminal. Uma justificativa para existência de tal arquivo é, por exemplo, quando as saídas todas do programa são direcionadas para um arquivo. Assim, as saídas do programa são escritas em um arquivo e as mensagens de erro são escritas na saída padrão, isto é, no terminal. Ainda há a possibilidade de escrever nossas próprias mensagens de erro no arquivo apontado por `stderr`.

## 17.4 Exemplos

Nessa seção apresentaremos dois exemplos que usam algumas das funções de entrada e saída em arquivos que aprendemos nesta aula.

O primeiro exemplo, apresentado no programa 17.2, é bem simples e realiza a cópia do conteúdo de um arquivo para outro arquivo.

Programa 17.2: Um exemplo de cópia de um arquivo.

```
#include <stdio.h>

#define MAX 100

int main(void)
{
   char nome_base[MAX+1], nome_copia[MAX+1];
   int c;
   FILE *ptbase, *ptcopia;

   printf("Informe o nome do arquivo a ser copiado: ");
   scanf("%s", nome_base);

   printf("Informe o nome do arquivo resultante: ");
   scanf("%s", nome_copia);

   ptbase = fopen(nome_base, "r");
   if (ptbase != NULL) {
      ptcopia = fopen(nome_copia, "w");
      if (ptcopia != NULL) {
         c = fgetc(ptbase);
         while (c != EOF) {
            fputc(c, ptcopia);
            c = fgetc(ptbase);
         }
         fclose(ptbase);
         fclose(ptcopia);
         printf("Arquivo copiado\n");
      }
      else {
         printf("Impossível abrir o arquivo %s para escrita\n", nome_copia);
      }
   }
   else {
      printf("Impossível abrir o arquivo %s para leitura\n", nome_base);
   }

   return 0;
}
```

O segundo exemplo, apresentado no programa 17.3, mescla o conteúdo de dois arquivos texto em um arquivo resultante. Neste exemplo, cada par de palavras do arquivo resultante é composto por uma palavra de um arquivo e uma palavra do outro arquivo. Se um dos arquivos contiver menos palavras que o outro arquivo, esse outro arquivo é descarregado no arquivo resultante.

Programa 17.3: Um segundo exemplo de uso de arquivos.

```
#include <stdio.h>

#define MAX 100

int main(void)
{
   char nome1[MAX+1], nome2[MAX+1], palavra1[MAX+1], palavra2[MAX+1];
   int i1, i2;
   FILE *pt1, *pt2, *ptr;

   printf("Informe o nome do primeiro arquivo: ");
   scanf("%s", nome1);
   printf("Informe o nome do segundo arquivo: ");
   scanf("%s", nome2);
   pt1 = fopen(nome1, "r");
   pt2 = fopen(nome2, "r");
   if (pt1 != NULL && pt2 != NULL) {
      ptr = fopen("mesclado", "w");
      if (ptr != NULL) {
         i1 = fscanf(pt1, "%s", palavra1);
         i2 = fscanf(pt2, "%s", palavra2);
         while (i1 != EOF && i2 != EOF) {
            fprintf(ptr, "%s %s ", palavra1, palavra2);
            i1 = fscanf(pt1, "%s", palavra1);
            i2 = fscanf(pt2, "%s", palavra2);
         }
         while (i1 != EOF) {
            fprintf(ptr, "%s ", palavra1);
            i1 = fscanf(pt1, "%s", palavra1);
         }
         while (i2 != EOF) {
            fprintf(ptr, "%s ", palavra2);
            i2 = fscanf(pt2, "%s", palavra2);
         }
         fprintf(ptr, "\n");
         fclose(pt1);
         fclose(pt2);
         fclose(ptr);
         printf("Arquivo mesclado\n");
      }
      else
         printf("Não é possível abrir um arquivo para escrita\n");
   }
   else
      printf("Não é possível abrir os arquivos para leitura\n");

   return 0;
}
```

## Exercícios

17.1 Escreva um programa que leia o conteúdo de um arquivo cujo nome é fornecido pelo usuário e copie seu conteúdo em um outro arquivo, trocando todas as letras minúsculas por letras maiúsculas.

17.2 Suponha que temos dois arquivos cujas linhas são ordenadas lexicograficamente. Por exemplo, esses arquivos podem conter nomes de pessoas, linha a linha, em ordem alfabética. Escreva um programa que leia o conteúdo desses dois arquivos, cujos nomes são fornecidos pelo usuário, e crie um novo arquivo resultante contendo todas as linhas dos dois arquivos ordenadas lexicograficamente. Por exemplo, se os arquivos contêm as linhas

| **Arquivo 1** | **Arquivo 2** |
|---|---|
| Antônio | Carlos |
| Berenice | Célia |
| Diana | Fábio |
| Solange | Henrique |
| Sônia | |
| Zuleica | |

o arquivo resultante deve ser

Antônio
Berenice
Carlos
Célia
Diana
Fábio
Henrique
Solange
Sônia
Zuleica

AULA 18

# LISTAS LINEARES

Listas lineares são as próximas estruturas de dados que aprendemos. Essas estruturas são muito usadas em diversas aplicações importantes para organização de informações na memória tais como representações alternativas para expressões aritméticas, armazenamento de argumentos de funções, compartilhamento de espaço de memória, entre outras. Nesta aula, baseada nas referências [2, 7, 13], aprenderemos a definição dessa estrutura, sua implementação em alocação dinâmica e suas operações básicas.

## 18.1 Definição

Informalmente, uma lista linear é uma estrutura de dados que armazena um conjunto de informações que são relacionadas entre si. Essa relação se expressa apenas pela ordem relativa entre os elementos. Por exemplo, nomes e telefones de uma agenda telefônica, as informações bancárias dos funcionários de uma empresa, as informações sobre processos em execução pelo sistema operacional, etc, são informações que podem ser armazenadas em uma lista linear. Cada informação contida na lista é na verdade um registro contendo os dados relacionados, que chamaremos daqui por diante de célula. Em geral, usamos um desses dados como uma chave para realizar diversas operações sobre essa lista, tais como busca de um elemento, inserção, remoção, etc. Já que os dados que acompanham a chave são irrelevantes e participam apenas das movimentações das células, podemos imaginar que uma lista linear é composta apenas pelas chaves dessas células e que essas chaves são representadas por números inteiros.

Agora formalmente, uma **lista linear** é um conjunto de $n \geqslant 0$ células $c_1, c_2, \ldots, c_n$ determinada pela ordem relativa desses elementos:

(*i*) se $n > 0$ então $c_1$ é a primeira célula;

(*ii*) a célula $c_i$ é precedida pela célula $c_{i-1}$, para todo $i$, $1 < i \leqslant n$.

As operações básicas sobre uma lista linear são as seguintes:

- busca;
- inclusão; e
- remoção.

Dependendo da aplicação, muitas outras operações também podem ser realizadas sobre essa estrutura como, por exemplo, alteração de um elemento, combinação de duas listas, ordenação da lista de acordo com as chaves, determinação do elemento de menor ou maior chave, determinação do tamanho ou número de elementos da lista, etc.

As listas lineares podem ser armazenadas na memória de duas maneiras distintas:

**Alocação estática ou seqüencial:** os elementos são armazenados em posições consecutivas de memória, com uso da vetores;

**Alocação dinâmica ou encadeada:** os elementos podem ser armazenados em posições não consecutivas de memória, com uso de ponteiros.

A aplicação, ou problema que queremos resolver, é que define o tipo de armazenamento a ser usado, dependendo das operações sobre a lista, do número de listas envolvidas e das características particulares das listas. Na aula 4 vimos especialmente as operações básicas sobre uma lista linear em alocação seqüencial, especialmente a operação de busca, sendo que as restantes foram vistas nos exercícios.

As células de uma lista linear em alocação encadeada encontram-se dispostas em posições aleatórias da memória e são ligadas por ponteiros que indicam a posição da próxima célula da lista. Assim, um campo é acrescentado a cada célula da lista indicando o endereço do próximo elemento da lista. Veja a figura 18.1 para um exemplo de uma célula de uma lista.

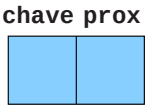



Figura 18.1: Representação de uma célula de uma lista linear em alocação encadeada.

A definição de uma célula de uma lista linear encadeada é então descrita como a seguir:

```
struct cel {
   int chave;
   struct cel *prox;
};
```

É boa prática de programação definir um novo tipo de dados para as células de uma lista linear em alocação encadeada:

```
typedef struct cel celula;
```

Uma célula `c` e um ponteiro `p` para uma célula podem ser declarados da seguinte forma:

```
   celula c;
   celula *p;
```

Se `c` é uma célula então `c.chave` é o conteúdo da célula e `c.prox` é o endereço da célula seguinte. Se `p` é o endereço de uma célula então `p->chave` é o conteúdo da célula apontada por `p` e `p->prox` é o endereço da célula seguinte. Se `p` é o endereço da última célula da lista então `p->prox` vale `NULL`.

Uma ilustração de uma lista linear em alocação encadeada é mostrada na figura 18.2.

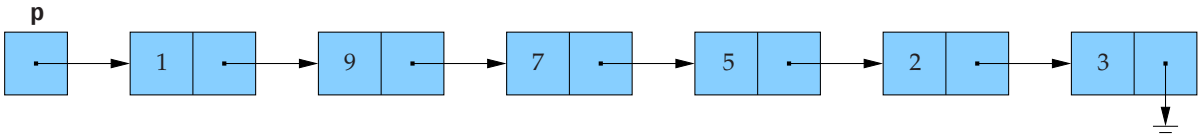



Figura 18.2: Representação de uma lista linear em alocação encadeada na memória.

Dizemos que o endereço de uma lista encadeada é o endereço de sua primeira célula. Se `p` é o endereço de uma lista, podemos dizer que "`p` é uma lista" ou ainda "considere a lista `p`". Por outro lado, quando dizemos "`p` é uma lista", queremos dizer que "`p` é o endereço da primeira célula de uma lista".

Uma lista linear pode ser vista de duas maneiras diferentes, dependendo do papel que sua primeira célula representa. Em uma lista linear **com cabeça**, a primeira célula serve apenas para marcar o início da lista e portanto o seu conteúdo é irrelevante. A primeira célula é a **cabeça** da lista. Em uma lista linear **sem cabeça** o conteúdo da primeira célula é tão relevante quanto o das demais.

Uma lista linear está vazia se não tem célula alguma. Para criar uma lista vazia `lista` com cabeça, basta escrever as seguintes sentenças:

```
celula c, *lista;
c.prox = NULL;
lista = &c;
```

ou ainda

```
celula *lista;
lista = (celula *) malloc(sizeof (celula));
lista->prox = NULL;
```

A figura 18.3 mostra a representação na memória de uma lista linear encadeada com cabeça e vazia.

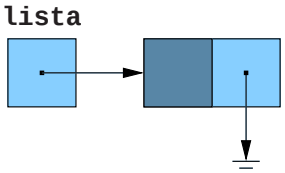



Figura 18.3: Uma lista linear encadeada com cabeça e vazia `lista`.

Para criar uma lista vazia `lista` sem cabeça, basta escrever as seguintes sentenças:

```
celula *lista;
lista = NULL;
```

A figura 18.4 mostra a representação na memória de uma lista linear encadeada sem cabeça vazia.

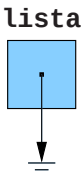



Figura 18.4: Uma lista linear encadeada sem cabeça e vazia `lista`.

Listas lineares com cabeça são mais fáceis de manipular do que aquelas sem cabeça. No entanto, as listas com cabeça têm sempre a desvantagem de manter uma célula a mais na memória.

Para imprimir o conteúdo de todas as células de uma lista linear podemos usar a seguinte função:

```
/* Recebe um ponteiro para uma lista linear encadeada
   e mostra o conteúdo de cada uma de suas células */
void imprime_lista(celula *lst)
{
   celula *p;

   for (p = lst; p != NULL; p = p->prox)
      printf("%d\n", p->chave);
}
```

Se `lista` é uma lista linear com cabeça, a chamada da função deve ser:

```
imprime_lista(lista->prox);
```

Se `lista` é uma lista linear sem cabeça, a chamada da função deve ser:

```
imprime_lista(lista);
```

A seguir vamos discutir e implementar as operações básicas de busca, inserção e remoção sobre listas lineares em alocação encadeada. Para isso, vamos dividir o restante do capítulo em listas sem cabeça e com cabeça.

## 18.2 Listas lineares com cabeça

Nesta seção tratamos das operações básicas nas listas lineares encadeadas com cabeça. Como mencionamos, as operações básicas são facilmente implementadas neste tipo de lista, apesar de sempre haver necessidade de manter uma célula a mais sem armazenamento de informações na lista.

### 18.2.1 Busca

O processo de busca de um elemento em uma lista linear em alocação encadeada com cabeça é muito simples. Basta verificar se o elemento é igual ao conteúdo de alguma célula da lista, como pode ser visto na descrição da função abaixo. A função `busca_C` recebe uma lista linear encadeada com cabeça apontada por `lst` e um número inteiro `x` e devolve o endereço de uma célula contendo `x`, caso `x` ocorra em `lst`. Caso contrário, a função devolve o ponteiro nulo `NULL`.

```
/* Recebe um número inteiro x e uma lista encadeada com cabeça lst e devol-
   ve o endereço da célula que contém x ou NULL se tal célula não existe */
celula *busca_C(int x, celula *lst)
{
   celula *p;

   p = lst->prox;

   while (p != NULL && p->chave != x)
      p = p->prox;

   return p;
}
```

É fácil notar que este código é muito simples e sempre devolve o resultado correto, mesmo quando a lista está vazia. Um exemplo de execução da função `busca_C` é mostrado na figura 18.5.

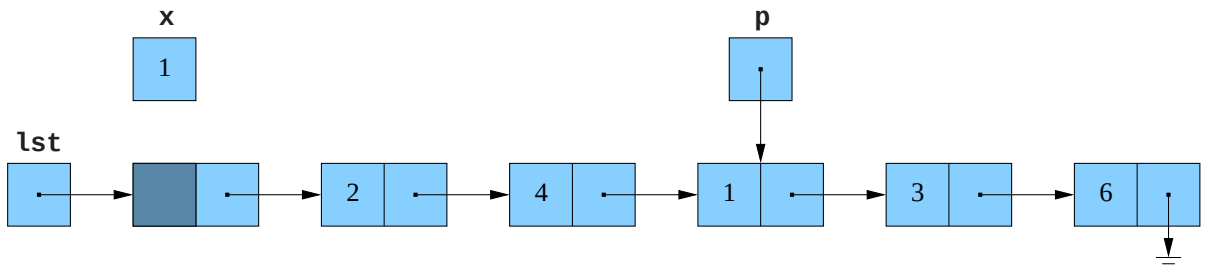



Figura 18.5: Resultado de uma chamada da função `busca_C` para uma dada lista linear e a chave 1.

Uma versão recursiva da função acima é apresentada a seguir.

```
/* Recebe um número inteiro x e uma lista encadeada com cabeça lst e devol-
   ve o endereço da célula que contém x ou NULL se tal célula não existe */
celula *buscaR_C(int x, celula *lst)
{
   if (lst->prox == NULL)
      return NULL;

   if (lst->prox->chave == x)
      return lst->prox;

   return buscaR_C(x, lst->prox);
}
```

### 18.2.2 Remoção

O problema nesta seção é a remoção de uma célula de uma lista linear encadeada. Parece simples remover tal célula apontada pelo ponteiro, mas esta idéia tem um defeito fundamental: a célula anterior a ela deve apontar para a célula seguinte, mas neste caso não temos o endereço da célula anterior. Dessa forma, no processo de remoção, é necessário apontar para a célula anterior àquela que queremos remover. Como uma lista linear encadeada com nó cabeça sempre tem uma célula anterior, a implementação da remoção é muito simples neste caso. A função `remove_C` recebe um ponteiro `p` para uma célula de uma lista linear encadeada e remove a célula seguinte, supondo que o ponteiro `p` aponta para uma célula e existe uma célula seguinte àquela apontada por `p`.

```
/* Recebe o endereço p de uma célula em uma lista encadeada e remo-
   ve a célula p->prox, supondo que p != NULL e p->prox != NULL */
void remove_C(celula *p)
{
   celula *lixo;

   lixo = p->prox;
   p->prox = lixo->prox;
   free(lixo);
}
```

A figura 18.6 ilustra a execução dessa função.

Observe que não há movimentação de dados na memória, como fizemos na aula 4 ao remover um elemento de um vetor.

### 18.2.3 Inserção

Nesta seção, queremos inserir uma nova célula em uma lista linear encadaeda com cabeça. Suponha que queremos inserir uma nova célula com chave `y` entre a célula apontada por `p` e a célula seguinte. A função `insere_C` abaixo recebe um número inteiro `y` e um ponteiro `p` para uma célula da lista e realiza a inserção de uma nova célula com chave `y` entre a célula apontada por `p` e a célula seguinte da lista.

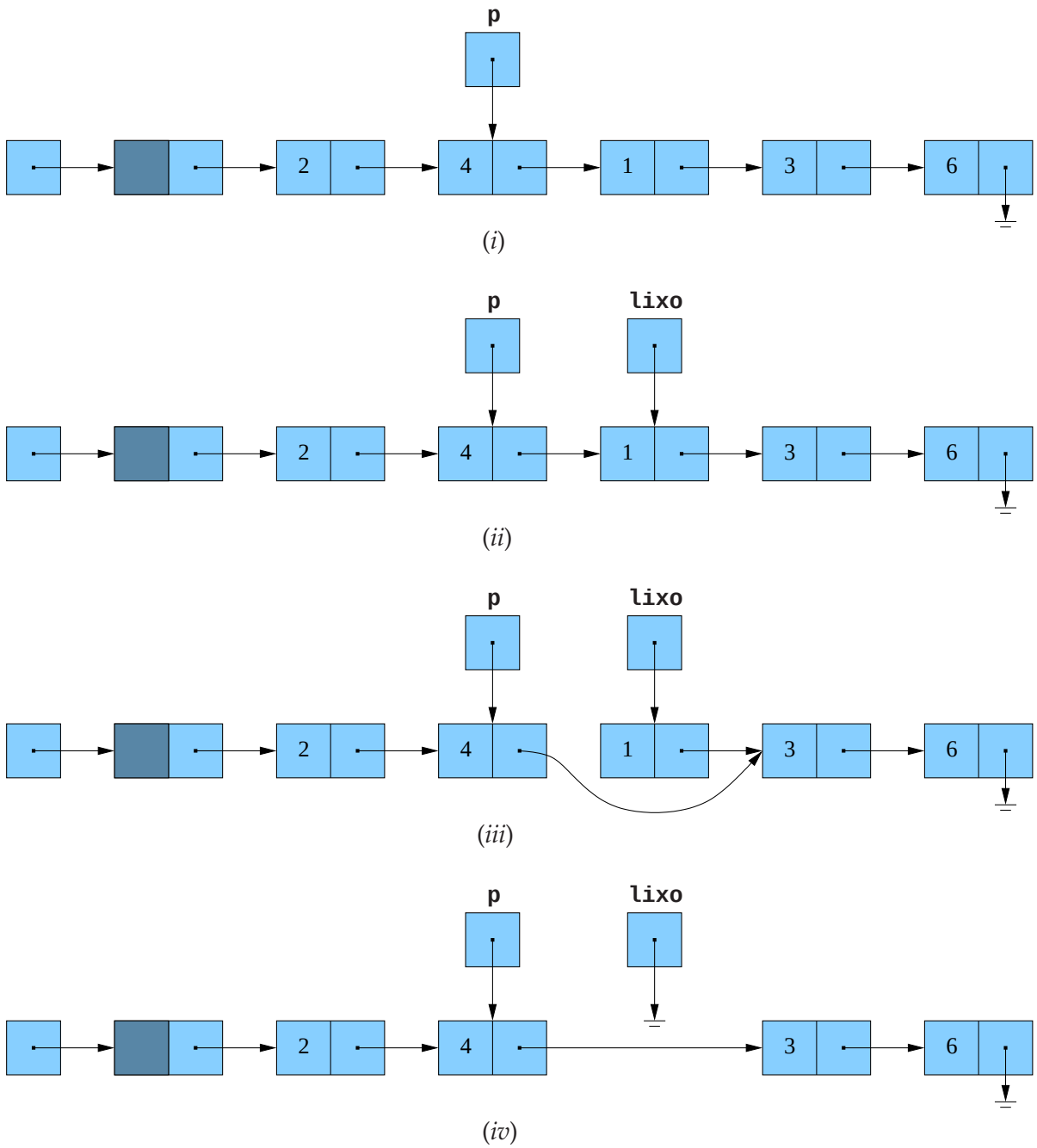



Figura 18.6: Resultado de uma chamada da função `remove_C` para uma célula apontada por `p` de uma dada lista linear. As figuras de (*i*) a (*iv*) representam a execução linha a linha da função.

```
/* Recebe um número inteiro y e o endereço p de uma célula de
   uma lista encadeada e insere uma nova célula com conteúdo
   y entre a célula p e a seguinte; supomos que p != NULL */
void insere_C(int y, celula *p)
{
   celula *nova;

   nova = (celula *) malloc(sizeof (celula));
   nova->chave = y;
   nova->prox = p->prox;
   p->prox = nova;
}
```

A figura 18.7 ilustra a execução dessa função.

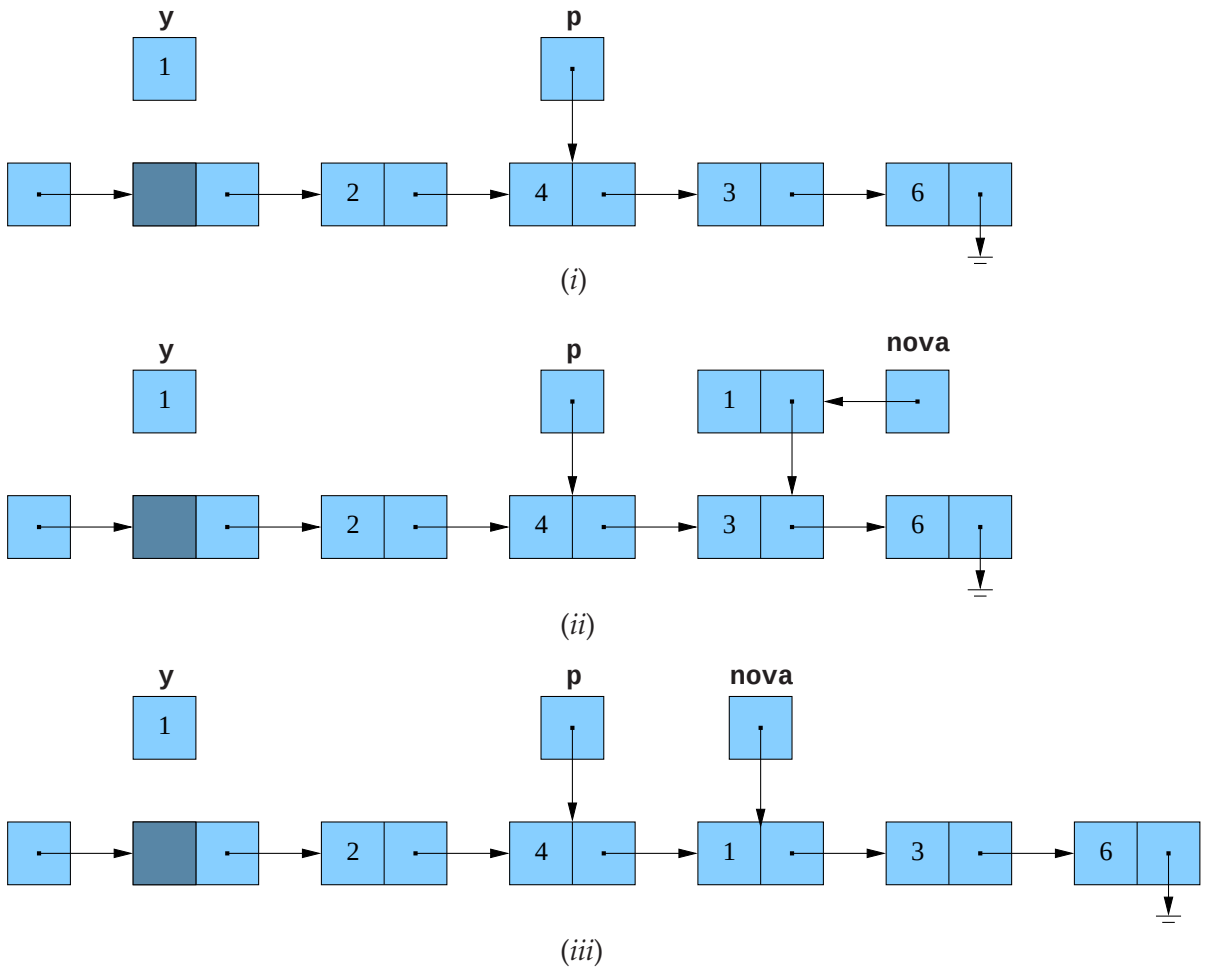



Figura 18.7: Resultado de uma chamada da função `insere_C` para uma nova chave 1 e a célula apontada por `p` de uma dada lista linear.

Observe que também não há movimentação de dados na memória na inserção realizada nessa função.

### 18.2.4 Busca com remoção ou inserção

Nesta seção, temos dois problemas a serem resolvidos. No primeiro, dado um número inteiro, nosso problema agora é aquele de remover a célula de uma lista linear encadaeda que contenha tal número. Se tal célula não existe, nada fazemos. A função `busca_remove_C` apresentada a seguir implementa esse processo. A figura 18.8 ilustra a execução dessa função.

```
/* Recebe um número inteiro x e uma lista encadeada com cabeça lst e remo-
   ve da lista a primeira célula que contiver x, se tal célula existir */
void busca_remove_C(int x, celula *lst)
{
   celula *p, *q;

   p = lst;
   q = lst->prox;
   while (q != NULL && q->chave != x) {
      p = q;
      q = q->prox;
   }
   if (q != NULL) {
      p->prox = q->prox;
      free(q);
   }
}
```

Agora, nosso problema é o de inserir uma nova célula com uma nova chave em uma lista linear encadaeda com cabeça, antes da primeira célula que contenha como chave um número inteiro fornecido. Se tal célula não existe, inserimos a nova chave no final da lista. A função `busca_remove_C` apresentada a seguir implementa esse processo. Um exemplo de execução dessa função é ilustrado na figura 18.8.

```
/* Recebe dois números inteiros y e x e uma lista encadeada
   com cabeça lst e insere uma nova célula com chave y nessa
   lista antes da primeira que contiver x; se nenhuma célula
   contiver x, a nova célula é inserida no final da lista */
void busca_insere_C(int y, int x, celula *lst)
{
   celula *p, *q, *nova;

   nova = (celula *) malloc(sizeof (celula));
   nova->chave = y;
   p = lst;
   q = lst->prox;
   while (q != NULL && q->chave != x) {
      p = q;
      q = q->prox;
   }
   nova->prox = q;
   p->prox = nova;
}
```

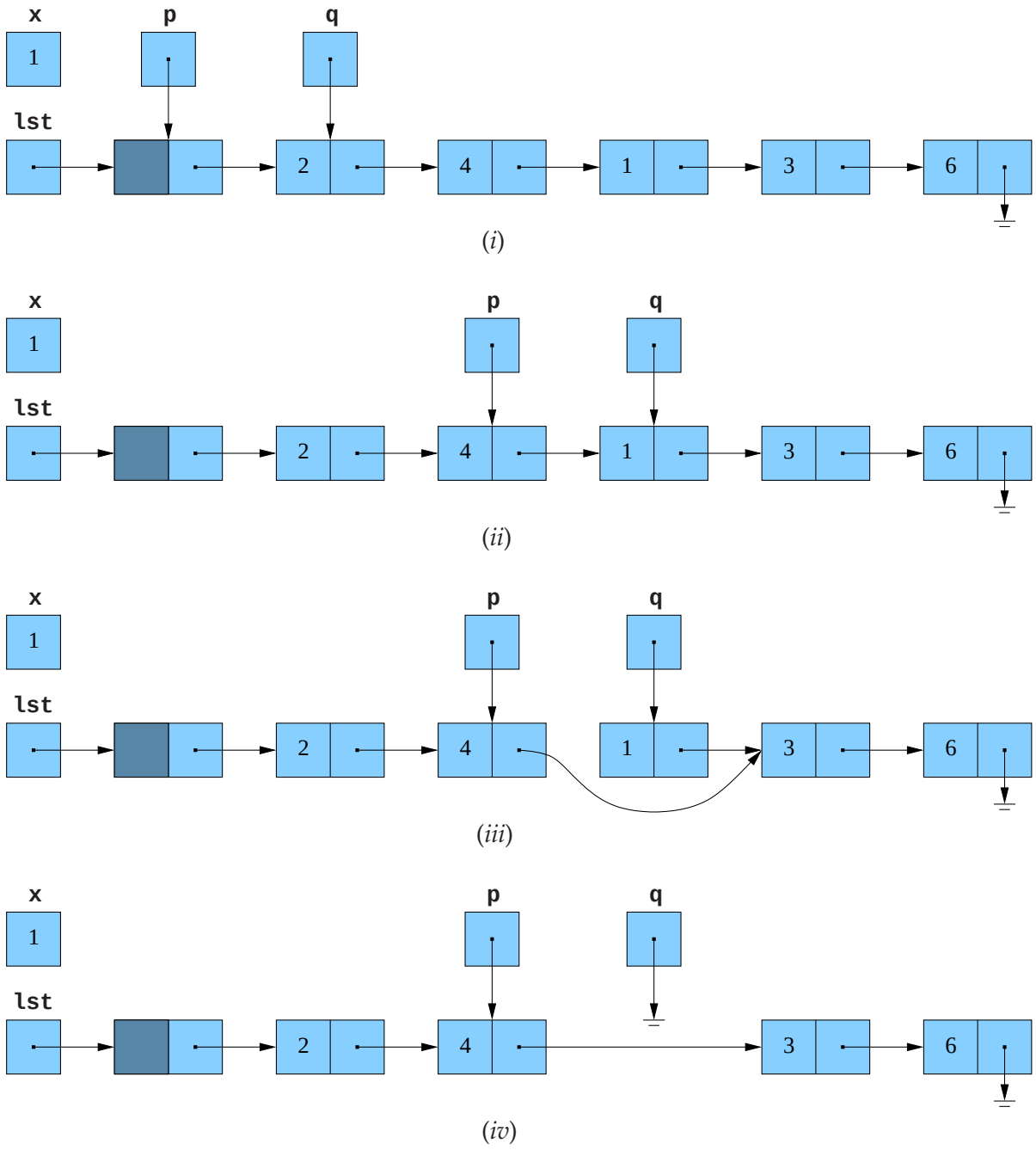



Figura 18.8: Resultado de uma chamada da função `busca_remove_C` para a chave 1 e uma lista `lst`. As figuras de (*i*) a (*iv*) representam a execução linha a linha da função.

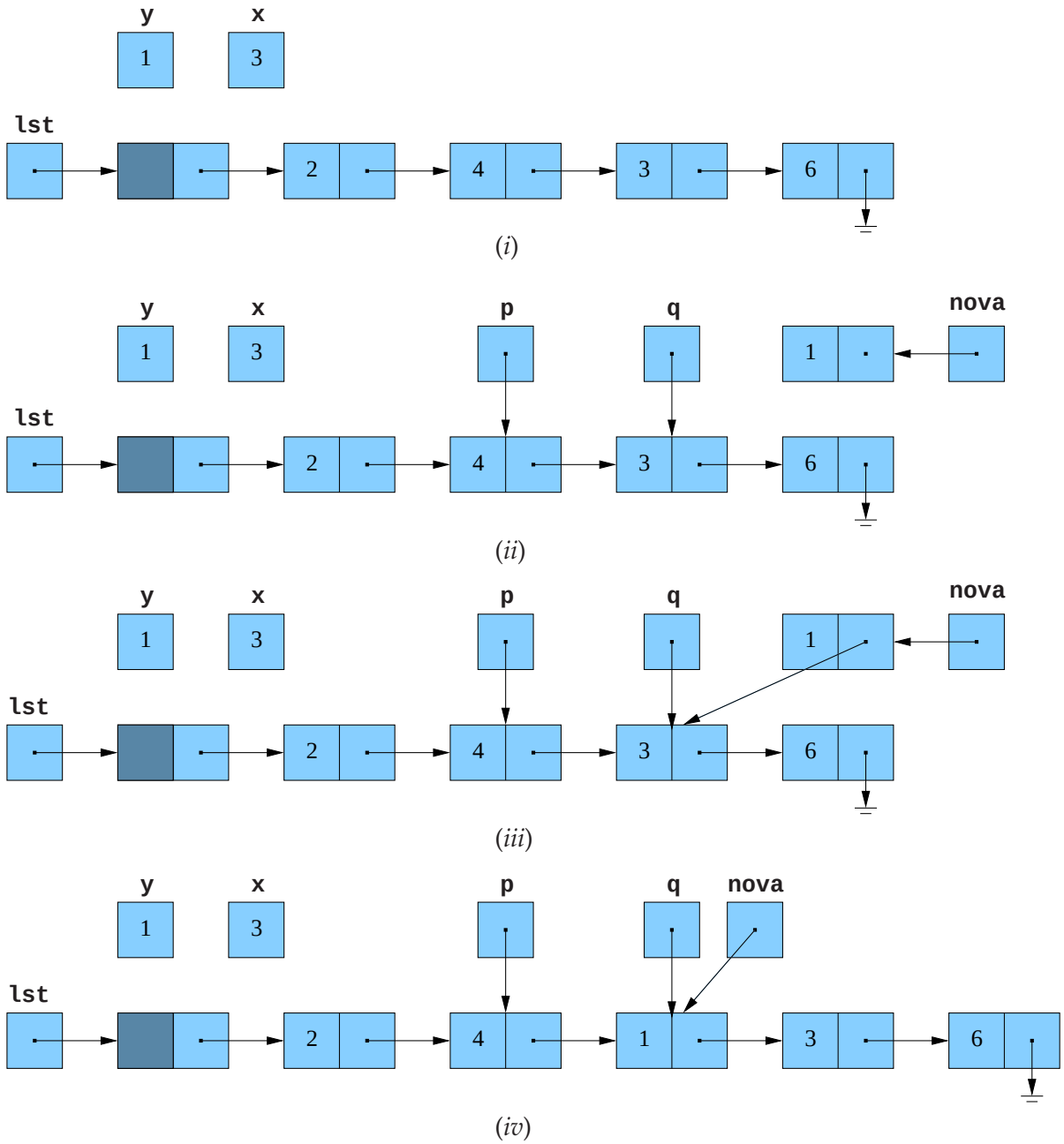



Figura 18.9: Resultado de uma chamada da função `busca_insere_C` para as chaves 1 e 3 e uma lista `lst`. As figuras de (*i*) a (*iv*) representam a execução linha a linha da função.

## 18.3 Listas lineares sem cabeça

Nesta seção tratamos das operações básicas nas listas lineares encadeadas sem cabeça. Como mencionamos, sempre economizamos uma célula em listas desse tipo, mas a implementação as operações básicas são um pouco mais complicadas.

### 18.3.1 Busca

A busca de um elemento em uma lista linear em alocação encadeada sem cabeça é muito simples e semelhante ao mesmo processo realizado sobre uma lista linear encadeada com cabeça. Seguem os códigos da função não-recursiva `busca_S` e da função recursiva `buscaR_S`.

```
/* Recebe um número inteiro x e uma lista encadeada sem cabeça lst e devol-
   ve o endereço da célula que contém x ou NULL se tal célula não existe */
celula *busca_S(int x, celula *lst)
{
   celula *p;

   p = lst;
   while (p != NULL && p->chave != x)
      p = p->prox;

   return p;
}
```

```
/* Recebe um número inteiro x e uma lista encadeada sem cabeça lst e devol-
   ve o endereço da célula que contém x ou NULL se tal célula não existe */
celula *buscaR_S(int x, celula *lst)
{
   if (lst == NULL)
      return NULL;

   if (lst->chave == x)
      return lst;

   return buscaR_S(x, lst->prox);
}
```

### 18.3.2 Busca com remoção ou inserção

Nesta seção apresentamos uma função que realiza uma busca seguida de remoção em uma lista linear encadeada sem cabeça.

Há muitas semelhanças entre a função de busca seguida de remoção em uma lista linear encadeada com cabeça e sem cabeça. No entanto, devemos reparar na diferença fundamental entre as duas implementações: uma remoção em uma lista linear encadeada sem cabeça pode modificar o ponteiro que identifica a lista quando removemos a primeira célula da lista. Dessa forma, a função `busca_remove_S` descrita abaixo tem como parâmetro que identifica a lista linear um ponteiro para um ponteiro.

```
/* Recebe um número inteiro x e uma lista encadeada com cabeça lst e remo-
   ve da lista a primeira célula que contiver x, se tal célula existir */
void busca_remove_S(int x, celula **lst)
{
   celula *p, *q;

   p = NULL;
   q = *lst;
   while (q != NULL && q->chave != x) {
      p = q;
      q = q->prox;
   }
   if (q != NULL)
      if (p != NULL) {
         p->prox = q->prox;
         free(q);
      }
      else {
         *lst = q->prox;
         free(q);
      }
}
```

Veja a figura 18.10 para um exemplo de busca seguida de remoção.

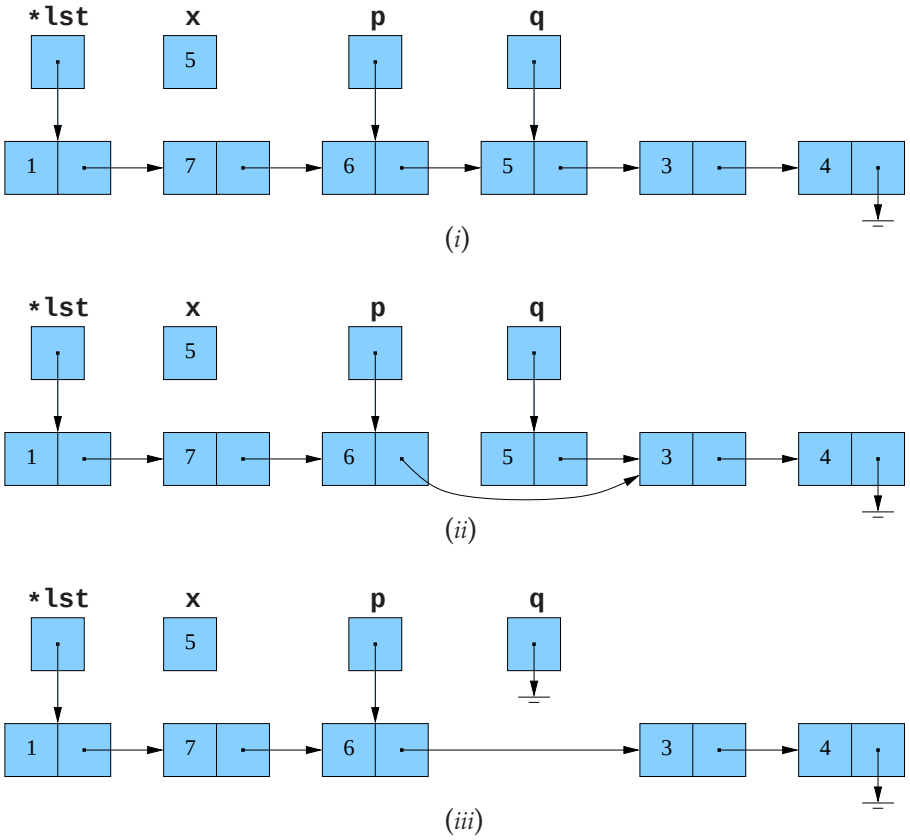



Figura 18.10: Remoção de uma chave em uma lista linear encadeada sem cabeça. (*i*) Após a busca do valor. (*ii*) Modificação do ponteiro `p->prox`. (*iii*) Liberação da memória.

Uma chamada à função `busca_remove_S` é ilustrada abaixo, para um número inteiro `x` e uma `lista`:

```
busca_remove_S(x, &lista);
```

Observe que os ponteiros `p` e `q` são variáveis locais à função `busca_remove_S`, que auxiliam tanto na busca como na remoção propriamente. Além disso, se `p == NULL` depois da execução da estrutura de repetição da função, então a célula que contém `x` que queremos remover, encontra-se na primeira posição da lista. Dessa forma, há necessidade de alterar o conteúdo do ponteiro `*lst` para o registro seguinte desta lista.

Agora, nosso problema é realizar uma bsuca seguida de uma inserção de uma nova célula em uma lista linear ecncadeada sem cabeça. Do mesmo modo, há muitas semelhanças entre a função de busca seguida de inserção em uma lista linear encadeada com cabeça e sem cabeça. De novo, devemos reparar na diferença fundamental entre as duas implementações: uma inserção em uma lista linear encadeada sem cabeça pode modificar o ponteiro que identifica a lista quando inserimos uma nova célula em uma lista vazia. Dessa forma, a função `busca_insere_S` descrita abaixo tem como parâmetro que identifica a lista linear um ponteiro para um ponteiro.

```
/* Recebe dois números inteiros y e x e uma lista encadeada
   com cabeça lst e insere uma nova célula com chave y nessa
   lista antes da primeira que contiver x; se nenhuma célula
   contiver x, a nova célula é inserida no final da lista */
void busca_insere_S(int y, int x, celula **lst)
{
   celula *p, *q, *nova;

   nova = (celula *) malloc(sizeof (celula));
   nova->chave = y;
   p = NULL;
   q = *lst;
   while (q != NULL && q->chave != x) {
      p = q;
      q = q->prox;
   }
   nova->prox = q;
   if (p != NULL)
      p->prox = nova;
   else
      *lst = nova;
}
```

Uma chamada à função `busca_insere_S` pode ser feita como abaixo, para números inteiros `y` e `x` e uma `lista`:

```
busca_insere_S(y, x, &lista);
```

## Exercícios

18.1 Se conhecemos apenas o ponteiro `p` para uma célula de uma lista linear em alocação encadeada, como na figura 18.11, e nada mais é conhecido, como podemos modificar a lista linear de modo que passe a conter apenas os valores $20, 4, 19, 47$, isto é, sem o conteúdo da célula apontada por `p`?

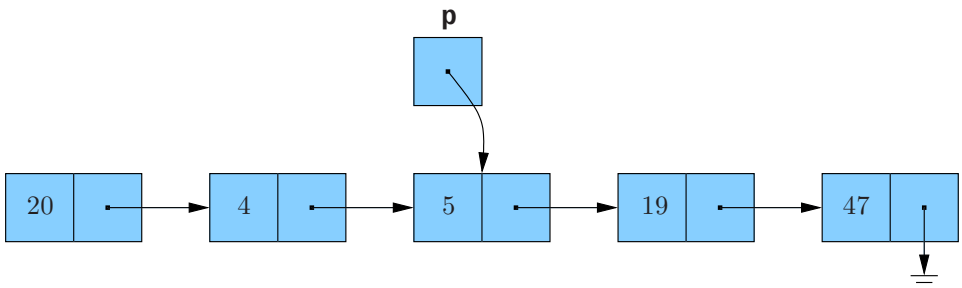



Figura 18.11: Uma lista linear encadeada com um ponteiro `p`.

18.2 O esquema apresentado na figura 18.12 permite percorrer uma lista linear encadeada nos dois sentidos, usando apenas o campo `prox` que contém o endereço do próximo elemento da lista. Usamos dois ponteiros `esq` e `dir`, que apontam para dois elementos vizinhos da lista. A idéia desse esquema é que à medida que os ponteiros `esq` e `dir` caminham na lista, os campos `prox` são invertidos de maneira a permitir o tráfego nos dois sentidos.

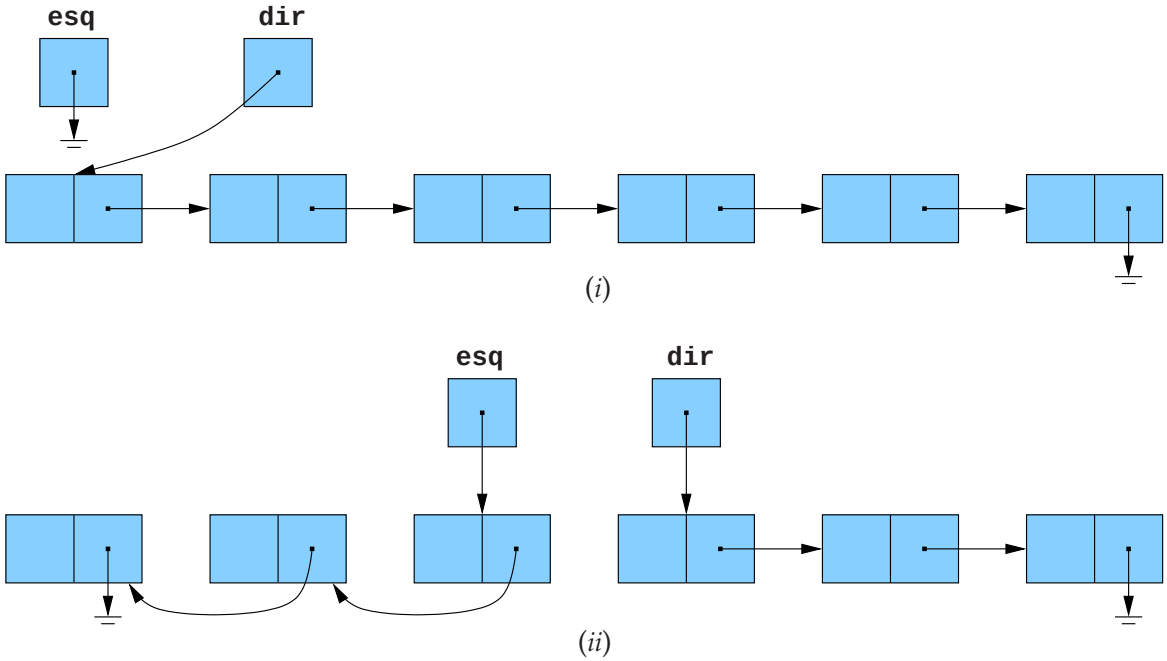



Figura 18.12: A figura (*ii*) foi obtida de (*i*) através da execução da função dada no exercício 18.2(a) por três vezes.

Escreva funções para:

(a) mover `esq` e `dir` para a direita de uma posição.

(b) mover `esq` e `dir` para a esquerda de uma posição.

18.3 Que acontece se trocarmos `while (p != NULL && p->chave != x)` por `while (p->chave != x && p != NULL` na função `busca_C`?

18.4 Escreva uma função que encontre uma célula cuja chave tem valor mínimo em uma lista linear encadeada. Considere listas com e sem cabeça e escreva versões não-recursivas e recursivas para a função.

18.5 Escreva uma função `busca_insere_fim` que receba um número inteiro `y`, uma lista linear encadeada `lst` e um ponteiro `f` para o fim da lista e realize a inserção desse valor no final da lista. Faça uma versões para listas lineares com cabeça e sem cabeça.

18.6 (a) Escreva duas funções: uma que copie um vetor para uma lista linear encadeada com cabeça; outra, que gfaça o mesmo para uma lista linear sem cabeça.

(b) Escreva duas funções: uma que copie uma lista linear encadeada com cabeça em um vetor; outra que copie uma lista linear encadeada sem cabeça em um vetor.

18.7 Escreva uma função que decida se duas listas dadas têm o mesmo conteúdo. Escreva duas versões: uma para listas lineares com cabeça e outra para listas lineares sem cabeça.

18.8 Escreva uma função que conte o número de células de uma lista linear encadeada.

18.9 Seja `lista` uma lista linear com seus conteúdos dispostos em ordem crescente. Escreva funções para realização das operações básicas de busca, inserção e remoção, respectivamente, em uma lista linear com essa característica. Escreva conjuntos de funções distintas para listas lineares com cabeça e sem cabeça. As operações de inserção e remoção devem manter a lista em ordem crescente.

18.10 Sejam duas listas lineares `lst1` e `lst2`, com seus conteúdos dispostos em ordem crescente. Escreva uma função `concatena` que receba `lst1` e `lst2` e construa uma lista `R` resultante da intercalação dessas duas listas, de tal forma que a lista construída também esteja ordenada. A função `concatena` deve destruir as listas `lst1` e `lst2` e deve devolver `R`. Escreva duas funções para os casos em que as listas lineares encadeadas são com cabeça e sem cabeça.

18.11 Seja `lst` uma lista linear encadeada composta por células contendo os valores $c_1, c_2, \ldots, c_n$, nessa ordem. Para cada item abaixo, escreva duas funções considerando que `lst` é com cabeça e sem cabeça.

(a) Escreva uma função `roda1` que receba uma lista e modifique e devolva essa lista de tal forma que a lista resultante contenha as chaves $c_2, c_3, \ldots, c_n, c_1$, nessa ordem.

(b) Escreva uma função `inverte` que receba uma lista e modifique e devolva essa lista de tal forma que a lista resultante contenha as chaves $c_n, c_{n-1}, \ldots, c_2, c_1$, nessa ordem.

(c) Escreva uma função `soma` que receba uma lista e modifique e devolva essa lista de tal forma que a lista resultante contenha as chaves $c_1 + c_n, c_2 + c_{n-1}, \ldots, c_{n/2} + c_{n/2+1}$, nessa ordem. Considere $n$ par.

18.12 Sejam $S_1$ e $S_2$ dois conjuntos disjuntos de números inteiros. Suponha que $S_1$ e $S_2$ estão implementados em duas listas lineares em alocação encadeada. Escreva uma função `uniao` que receba as listas representando os conjuntos $S_1$ e $S_2$ e devolva uma lista resultante que representa a união dos conjuntos, isto é, uma lista linear encadeada que representa o conjunto $S = S_1 \cup S_2$. Considere os casos em que as listas lineares encadeadas são com cabeça e sem cabeça.

18.13 Seja um polinômio $p(x) = a_0x^n + a_1x^{n-1} + \ldots + a_n$, com coeficientes de ponto flutuante. Represente $p(x)$ adequadamente por uma lista linear encadeada e escreva as seguintes funções. Para cada item a seguir, considere o caso em que a lista linear encadeada é com cabeça e sem cabeça.

(a) Escreva uma função `pponto` que receba uma lista `p` e um número de ponto flutuante `x0` e calcule e devolva $p(x_0)$.

(b) Escreva uma função `psoma` que receba as listas lineares `p` e `q`, que representam dois polinômios, e calcule e devolva o polinômio resultante da operação $p(x) + q(x)$.

(c) Escreva uma função `pprod` que receba as listas lineares `p` e `q`, que representam dois polinômios, e calcule e devolva o polinômio resultante da operação $p(x) \cdot q(x)$.

18.14 Escreva uma função que aplique a função `free` a todas as células de uma lista linear encadeada, supondo que todas as suas células foram alocadas com a funçao `malloc`. Faça versões considerando listas lineares encadeadas com cabeça e sem cabeça.

AULA 19

# PILHAS

Quando tratamos de listas lineares, o armazenamento seqüencial é usado, em geral, quando essas estruturas sofrem poucas modificações ao longo do tempo de execução em uma aplicação, com poucas inserções e remoções realizadas durante sua existência. Isso implica claramente em poucas movimentações de células. Por outro lado, a alocação encadeada é mais usada em situações inversas, ou seja, quando modificações nas listas lineares são mais freqüentes.

Caso os elementos a serem inseridos ou removidos de uma lista linear se localizem em posições especiais nessas estruturas, como por exemplo a primeira ou a última posição, então temos uma situação favorável para o uso de listas lineares em alocação seqüencial. Este é o caso da estrutura de dados denominada pilha. O que torna essa lista linear especial é a adoção de uma política bem definida de inserções e remoções, sempre em um dos extremos da lista. Além da alocação seqüencial, a implementação de uma pilha em alocação encadeada tem muitas aplicações importantes e também será discutida nesta aula.

Esta aula é baseada nas referências [2, 13].

## 19.1 Definição

Uma **pilha** é uma lista linear tal que as operações de inserção e remoção são realizadas em um único extremo dessa estrutura de dados.

O funcionamento dessa lista linear pode ser comparado a qualquer pilha de objetos que usamos com freqüência como, por exemplo, uma pilha de pratos de um restaurante. Em geral, os clientes do restaurante retiram pratos do início da pilha, isto é, do primeiro prato mais alto na pilha. Os funcionários colocam pratos limpos também nesse mesmo ponto da pilha. Seria estranho ter de movimentar pratos, por exemplo no meio ou no final da pilha, sempre que uma dessas dessas operações fosse realizada.

O indicador do extremo onde ocorrem as operações de inserção e remoção é chamado de **topo** da pilha. Essas duas operações são também chamadas de empilhamento e desempilhamento de elementos. Nenhuma outra operação é realizada sobre uma pilha, a não ser em casos específicos. Observe, por exemplo, que a operação de busca não foi mencionada e não faz parte do conjunto de operações básicas de uma pilha.

## 19.2 Operações básicas em alocação seqüencial

Uma pilha em alocação seqüencial é descrita através de duas variáveis: um vetor e um índice para o vetor. Supomos então que a pilha esteja armazenada no vetor `P[0..MAX-1]` e que a parte do vetor efetivamente ocupada pela pilha seja `P[0..t]`, onde `t` é o índice que define o topo da pilha. Os compartimentos desse vetor são convencionalmente do tipo `int`, mas podemos defini-los de qualquer tipo. Uma ilustração de uma pilha em alocação seqüencial é dada na figura 19.1.

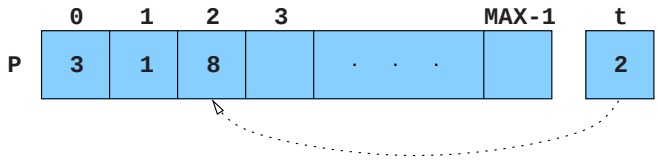
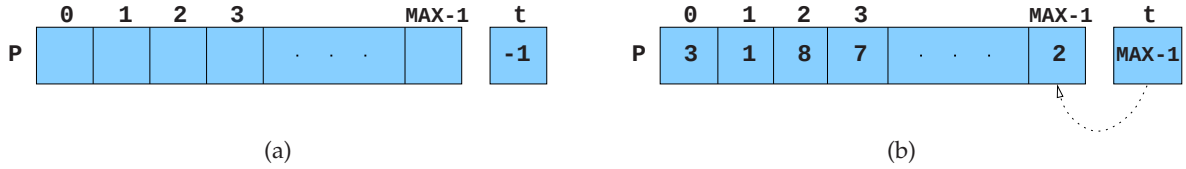



Figura 19.1: Representação de uma pilha `P` com topo `t` em alocação seqüencial.

Convencionamos que uma pilha está **vazia** se seu topo `t` vale $-1$ e **cheia** se seu topo `t` vale `MAX-1`. As representações de pilhas vazia e cheia são mostradas na figura 19.2.




(a) (b)

Figura 19.2: (a) Pilha vazia. (b) Pilha cheia.

A declaração de uma pilha e sua inicialização são mostradas abaixo:

```
int t, P[MAX];
t = -1;
```

Suponha agora que queremos inserir um elemento de valor 7 na pilha `P` da figura 19.1. Como resultado desta operação, esperamos que a pilha tenha a configuração mostrada na figura 19.3 após sua execução.

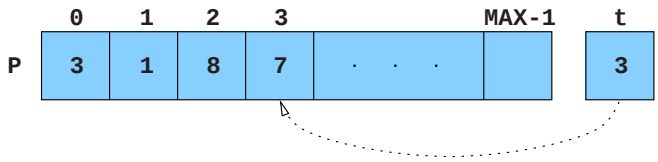



Figura 19.3: Inserção da chave 7 na pilha `P` da figura 19.1.

A operação de inserir um objeto em uma pilha, ou empilhar, é descrita na função `empilha_seq` a seguir.

```
/* Recebe um ponteiro t para o topo de uma pilha P
   e um elemento y e insere y no topo da pilha P */
void empilha_seq(int *t, int P[MAX], int y)
{
   if (*t != MAX - 1) {
      (*t)++;
      P[*t] = y;
   }
   else
      printf("Pilha cheia!\n");
}
```

Suponha agora que queremos remover um elemento de uma pilha. Observe que, assim como na inserção, a remoção também deve ser feita em um dos extremos da pilha, isto é, no topo da pilha. Assim, a remoção de um elemento da pilha `P` que tem sua configuração ilustrada na figura 19.1 tem como resultado a pilha com a configuração mostrada na figura 19.4 após a sua execução.

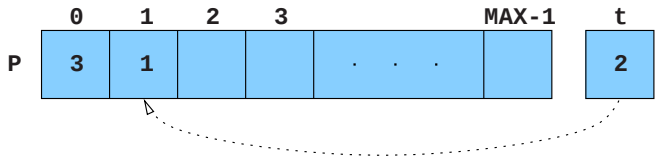



Figura 19.4: Remoção de um elemento da pilha `P` da figura 19.1.

A operação de remover, ou desempilhar, um objeto de uma pilha é descrita na função `desempilha_seq` a seguir.

```
/* Recebe um ponteiro t para o topo de uma pilha
   P e remove um elemento do topo da pilha P */
int desempilha_seq(int *t, int P[MAX])
{
   int r;

   if (*t != -1) {
      r = P[*t];
      (*t)--;
      } else {
         r = INT_MIN;
         printf("Pilha vazia!\n");
      }
      return r;

   }
```

Diversas são as aplicações que necessitam de uma ou mais pilhas nas suas soluções. Dentre elas, podemos citar a conversão de notação de expressões aritméticas (notação *prefixa*, *infixa* e *posfixa*), a implementação da pilha de execução de um programa no sistema operacional, a análise léxica de um programa realizada pelo compilador, etc. Algumas dessas aplicações serão vistas nos exercícios desta aula.

Como exemplo, suponha que queremos saber se uma seqüência de caracteres é da forma $a$Z$b$, onde $a$ é uma seqüência de caracteres e $b$ é o reverso da $a$. Observe que se $a$ = ABCDEFG e $b$ = GFEDCBA então a cadeia $a$Z$b$ satisfaz a propriedade. Suponh ainda que as seqüências $a$ e $b$ possuem a mesma quantidade de caracteres. A função **aZb_seq** abaixo verifica essa propriedade para uma seqüência de caracteres usando uma pilha em alocação seqüencial.

```
/* Recebe, via entrada padrão, uma seqüência de caracteres e ve-
   rifica se a mesma é da forma aZb, onde b é o reverso de a */
int aZb_seq(void)
{
   char c, P[MAX];
   int t;

   t = -1;
   c = getchar();
   while (c != 'Z') {
      t++;
      P[t] = c;
      c = getchar();
   }
   c = getchar();
   while (t != -1 && c == P[t]) {
      t--;
      c = getchar();
   }
   if (t == -1)
      return 1;
   while (t > -1) {
      c = getchar();
      t--;
   }
   return 0;
}
```

## 19.3 Operações básicas em alocação encadeada

Existem muitas aplicações práticas onde as pilhas implementadas em alocação encadeada são importantes e bastante usadas. Consideramos que as células da pilha são do tipo abaixo:

```
typedef struct cel {
   int chave;
   struct cel *prox;
} celula;
```

Uma pilha vazia com cabeça pode ser criada da seguinte forma:

```
   celula *t;
   t = (celula *) malloc(sizeof (celula));
   t->prox = NULL;
```

Uma pilha vazia sem cabeça pode ser criada da seguinte forma:

```
celula *t;
t = NULL;
```

Uma ilustração de uma pilha em alocação encadeada vazia é mostrada na figura 19.5.

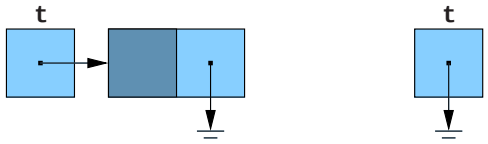



Figura 19.5: Pilha vazia em alocação encadeada (a) com cabeça e (b) sem cabeça.

Considere uma pilha em alocação encadeada com cabeça. A operação de empilhar uma nova chave `y` nessa pilha é descrita na função `empilha_enc_C` a seguir.

```
/* Recebe um elemento y e uma pilha t e insere y no topo de t */
void empilha_enc_C(int y, celula *t)
{
   celula *nova;

   nova = (celula *) malloc(sizeof (celula));
   nova->conteudo = y;
   nova->prox = t->prox;
   t->prox = nova;
}
```

A operação de desempilhar uma célula da pilha é descrita na função `desempilha_enc_C`.

```
/* Recebe uma pilha t e remove um elemento de seu topo */
int desempilha_enc_C(celula *t)
{
   int x;
   celula *p;

   if (t->prox != NULL) {
      p = t->prox;
      x = p->conteudo;
      t->prox = p->prox;
      free(p);
      return x;
   }
   else {
      printf("Pilha vazia!\n");
      return INT_MIN;
   }
}
```

A função **aZb_enc** é a versão que usa uma pilha em alocação encadeada para solucionar o mesmo problema que a função **aZb_seq** soluciona usando uma pilha em alocação seqüencial.

```
/* Recebe, via entrada padrão, uma seqüência de caracteres e ve-
   rifica se a mesma é da forma aZb, onde b é o reverso de a */
int aZb_enc(void)
{
   char c;
   celula *t, *p;

   t = (celula *) malloc(sizeof (celula));
   t->prox = NULL;
   c = getchar();
   while (c != 'Z') {
      p = (celula *) malloc(sizeof (celula));
      p->chave = c;
      p->prox = t->prox;
      t->prox = p;
      c = getchar();
   }
   c = getchar();
   while (t->prox != NULL && c == t->chave) {
      p = t->prox;
      t->prox = p->prox;
      free(p);
      c = getchar();
   }
   if (t->prox == NULL)
      return 1;
   while (t->prox != NULL) {
      p = t->prox;
      t->prox = p->prox;
      free(p);
      c = getchar();
   }
   return 0;
}
```

## Exercícios

19.1 Considere uma pilha em alocação encadeada sem cabeça. Escreva as funções para empilhar um elemento na pilha e desempilhar um elemento da pilha.

19.2 Uma palavra é um **palíndromo** se a seqüência de caracteres que a constitui é a mesma quer seja lida da esquerda para a direita ou da direita para a esquerda. Por exemplo, as palavras RADAR e MIRIM são palíndromos. Escreva um programa eficiente para reconhecer se uma dada palavra é palíndromo.

19.3 Um estacionamento possui um único corredor que permite dispor 10 carros. Existe somente uma única entrada/saída do estacionamento em um dos extremos do corredor. Se um cliente quer retirar um carro que não está próximo à saída, todos os carros impedindo sua passagem são retirados, o cliente retira seu carro e os outros carros são recolocados na mesma ordem que estavam originalmente. Escreva um algoritmo que processa o fluxo

de chegada/saída deste estacionamento. Cada entrada para o algoritmo contém uma letra `E` para entrada ou `S` para saída, e o número da placa do carro. Considere que os carros chegam e saem pela ordem especificada na entrada. O algoritmo deve imprimir uma mensagem sempre que um carro chega ou sai. Quando um carro chega, a mensagem deve especificar se existe ou não vaga para o carro no estacionamento. Se não existe vaga, o carro não entra no estacionamento e vai embora. Quando um carro sai do estacionamento, a mensagem deve incluir o número de vezes que o carro foi movimentado para fora da garagem, para permitir que outros carros pudessem sair.

19.4 Considere o problema de decidir se uma dada seqüência de parênteses e chaves é bem-formada. Por exemplo, a seqüência abaixo:

```
(  (  )  {  (  )  }  )
```

é bem-formada, enquanto que a seqüência

```
(  {  )  }
```

é malformada.

Suponha que a seqüência de parênteses e chaves está armazenada em uma cadeia de caracteres `s`. Escreva uma função `bem_formada` que receba a cadeia de caracteres `s` e devolva `1` se `s` contém uma seqüência bem-formada de parênteses e chaves e devolva `0` se a seqüência está malformada.

19.5 Como mencionado na seção 19.2, as expressões aritméticas podem ser representadas usando notações diferentes. Costumeiramente, trabalhamos com expressões aritméticas em notação *infixa*, isto é, na notação em que os operadores binários aparecem entre os operandos. Outra notação freqüentemente usada em compiladores é a notação *posfixa*, aquela em que os operadores aparecem depois dos operandos. Essa notação é mais econômica por dispensar o uso de parênteses para alteração da prioridade das operações. Exemplos de expressões nas duas notações são apresentados abaixo.

| notação *infixa* | notação *posfixa* |
|---|---|
| `(A + B * C)` | `A B C * +` |
| `(A * (B + C) / D + E)` | `A B C + * D / E -` |
| `(A + B * C / D * E - F)` | `A B C * D / E * + F -` |

Escreva uma função que receba um cadeia de caracteres contendo uma expressão aritmética em notação *infixa* e devolva uma cadeia de caracteres contendo a mesma expressão aritmética em notação *posfixa*. Considere que a cadeia de caracteres da expressão *infixa* contém apenas letras, parênteses e os símbolos `+`, `-`, `*` e `/`. Considere também que cada variável tem apenas uma letra e que a cadeia de caracteres *infixa* sempre está envolvida por um par de parênteses.

19.6 Suponha que exista um único vetor $M$ de céluas de um tipo pilha pré-definido, com um total de `MAX` posições. Este vetor fará o papel da memória do computador. Este vetor $M$ será compartilhado por duas pilhas em alocação seqüencial. Implemente eficientemente as operações de empilhamento e desempilhamento para as duas pilhas de modo que nenhuma das pilhas estoure sua capacidade de armazenamento, a menos que o total de elementos em ambas as pilhas seja `MAX`.

AULA 20

# FILAS

Uma fila é, assim como uma pilha, uma lista linear especial, onde há uma política de inserções e remoções bem definida. As únicas operações realizadas sobre as filas são exatamente inserção e remoção. Há dois extremos em uma fila e a operação de inserção é sempre realizada em um dos extremos e de remoção sempre no outro extremo. Em uma pilha, como vimos na aula 19, a inserção e a remoção são realizadas em um mesmo extremo. Nesta aula, baseada nas referências [2, 13], veremos uma definição formal para essas estruturas de dados e como implementar as operações básicas sobre elas, considerando sua implementação em alocação seqüencial e encadeada.

## 20.1 Definição

Uma **fila** é uma lista linear com dois extremos destacados e tal que as operações de inserção são realizadas em um dos extremos da lista e a remoção é realizada no outro extremo. O funcionamento dessa estrutura pode ser comparado a qualquer fila que usamos com freqüência como, por exemplo, uma fila de um banco. Em geral, as pessoas entram, ou são inseridas, no final da fila e as pessoas saem, ou são removidas, do início da fila.

Note que uma fila tem sempre um indicador de onde começa e de onde termina. Quando queremos inserir um elemento na fila, esta operação é realizada no final da estrutura. Quando queremos remover um elemento da fila, essa operação é realizada no início da estrutura. Se a fila é implementada em alocação seqüencial então esses extremos são índices de um vetor. Caso contrário, se for implementada em alocação encadeada, então esses extremos são ponteiros para a primeira e a última células da fila.

Assim como nas pilhas, duas operações básicas são realizadas sobre uma fila: inserção e remoção. Essas duas operações são também chamadas de enfileiramento e desenfileiramento de elementos. Observe que a operação de busca não foi mencionada e não faz parte do conjunto de operações básicas de uma fila.

## 20.2 Operações básicas em alocação seqüencial

A implementação mais simples de uma fila em alocação seqüencial é armazenada em um segmento `F[i..f-1]` de um vetor `F[0..MAX-1]`, com $0 \leqslant$ `i` $\leqslant$ `f` $\leqslant$ `MAX`. O primeiro elemento da fila está na posição `i` e o último na posição `f-1`. Uma ilustração de uma fila é mostrada na figura 20.1.

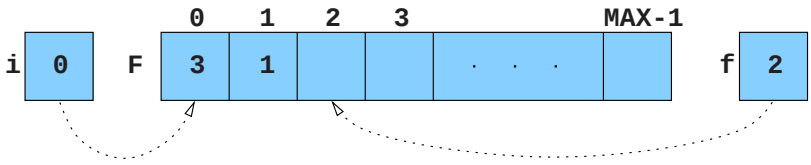
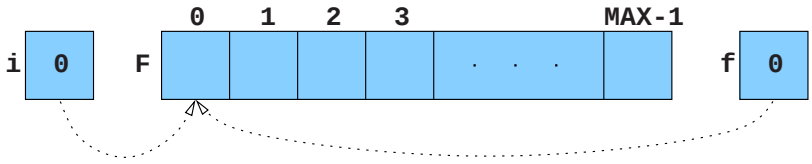
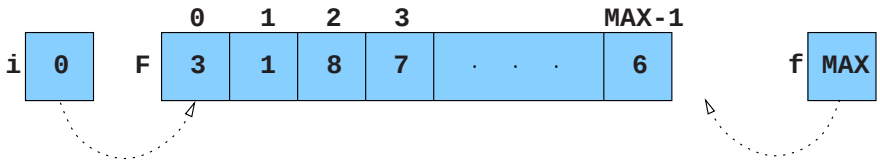



Figura 20.1: Representação de uma fila `F` em alocação seqüencial.

Convecionamos que uma fila está **vazia** se seus marcadores de início e fim são tais que `i` = `f` e **cheia** se seu marcador de fim `f` vale `MAX`. O problema de controle de espaços disponíveis na fila surge naturalmente numa implementação como esta. Representações de filas vazia e cheia em alocação seqüencial é mostrada na figura 20.2.




(a)




(b)

Figura 20.2: Representação de uma fila em alocação seqüencial (a) vazia e (b) cheia.

A declaração de uma fila e sua inicialização são mostradas abaixo:

```
int i, f, F[MAX];
i = 0;
f = 0;
```

Suponha que queremos inserir um elemento de chave 8 na fila `F` da figura 20.1. A inserção de uma nova chave em uma fila é sempre realizada no fim da estrutura. Como resultado desta operação, esperamos que a fila tenha a configuração mostrada na figura 20.3 após a execução dessa operação.

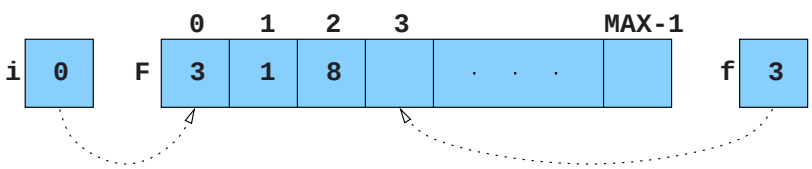



Figura 20.3: Inserção da chave 8 na fila `F` da figura 20.1.

A operação de inserir, ou enfileirar, uma chave em uma fila é dada pela função a seguir.

```
/* Recebe o índice f do fim da fila F e a chave y e insere y no fim de F */
void enfileira_seq(int *f, int F[MAX], int y)
{
   if (*f != MAX) {
      F[*f] = y;
      (*f)++;
   }
   else
      printf("Fila cheia!\n");
}
```

Suponha agora que queremos remover um elemento da fila `F` da figura 20.3. A remoção de um elemento ou de uma chave é sempre realizada no início da fila. Como resultado desta operação, esperamos que a fila tenha a configuração ilustrada na figura 20.3 após terminada sua execução.

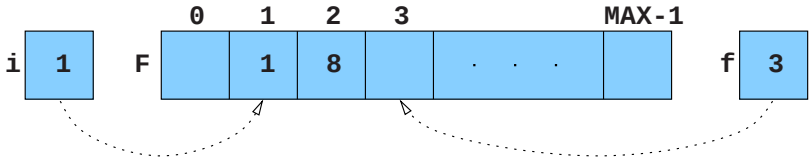



Figura 20.4: Remoção de um elemento da fila `F` da figura 20.3.

A operação de remover, ou desenfileirar, uma chave em uma fila é implementada na função abaixo.

```
/* Recebe os índices i e f da fila F e remove a chave do início i de F */
int desenfileira_seq(int *i, int f, int F[MAX])
{
   int x;

   if (*i != f) {
      x = F[*i];
      (*i)++;
   }
   else {
      x = INT_MIN;
      printf("Fila vazia!\n");
   }
   return x;
}
```

De acordo com as implementações das operações sobre a fila, observe que o índice `i` move-se somente quando da remoção de uma célula da fila e o índice `f` move-se somente quando da inserção de uma célula na fila. Esses incrementos fazem com que a fila "movimente-se" da esquerda para direita, o que pode ocasionar a falsa impressão de falta de capacidade de armazenamento na fila, como mostramos na figura 20.5.

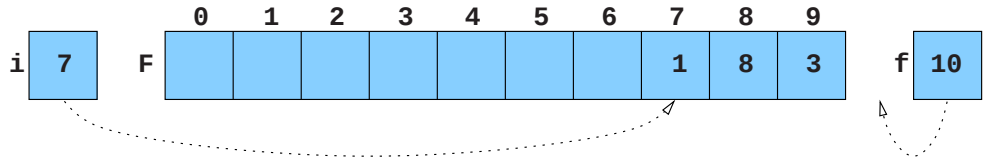



Figura 20.5: Representação de uma fila falsamente cheia.

Consideramos daqui por diante que as células são alocadas seqüencialmente como se estivessem em um círculo, onde o compartimento `F[MAX-1]` é seguido pelo compartimento `F[0]`. Assim, os elementos da fila estão dispostos no vetor `F[0..MAX-1]` em `F[i..f-1]` ou em `F[i..MAX-1]F[0..f-1]`. Temos também que $0 \leqslant$ `i` $<$ `MAX` e $0 \leqslant$ `f` $\leqslant$ `MAX`. Com essas suposições, dizemos que uma fila está **vazia** se `i` $=$ `INT_MIN` e `f` $=$ `INT_MAX` e está **cheia** se `f` $=$ `i`.

As operações básicas em uma fila circular são descritas a seguir.

```
/* Recebe os índice i e f da fila F e a chave y e insere y no fim de F */
void enfileira_seq_2(int *i, int *f, int F[MAX], int y)
{
   if (*f != *i) {
      if (*f == INT_MAX) {
         *i = 0;
         *f = 0;
      }
      F[*f] = y;
      *f = (*f + 1) % MAX;
   }
   else
      printf("Fila cheia!\n");
}
```

```
/* Recebe os índices i e f da fila F e remove a chave do início de F */
int desenfileira_seq_2(int *i, int *f, int F[MAX])
{
   int r;

   r = INT_MIN;
   if (*i != INT_MIN) {
      r = F[*i];
      *i = (*i + 1) % MAX;
      if (*i == *f) {
         *i = INT_MIN;
         *f = INT_MAX;
      }
   }
   else
      printf("Fila vazia!\n");
   return r;
}
```

## 20.3 Operações básicas em alocação encadeada

Uma fila tem sempre dois marcadores de onde começa e termina. A política de armazenamento de elementos nas células de uma fila estabelece que quando queremos inserir um elemento, sempre o fazemos no extremo identificado como o final dessa estrutura. Por outro lado, quando queremos remover um elemento, essa operação é sempre realizada no extremo identificado como início da estrutura. Em alocação encadeada, esses extremos são referenciados através de ponteiros para células que contêm o endereço das suas primeira e última células.

Assim como nas pilhas, apenas duas operações básicas são realizadas sobre as filas: inserção e remoção. Essas duas operações são também chamadas de enfileiramento e desefileiramento de elementos. Observe que a operação básica de busca não foi mencionada e também não faz parte do conjunto de operações básicas de uma fila.

Um exemplo de uma fila em alocação encadeada é dado na figura 20.6.

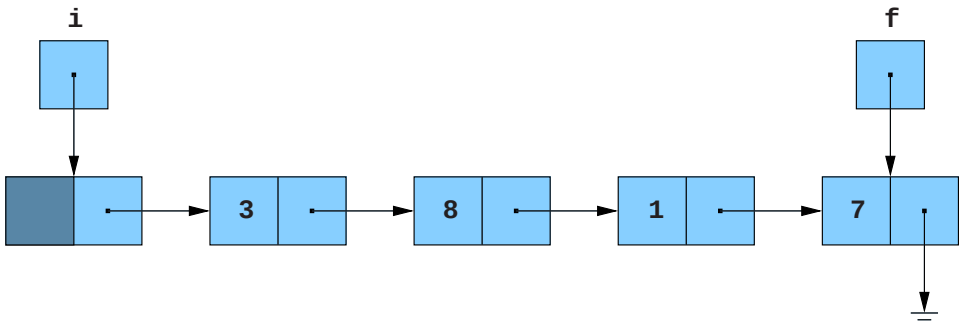



Figura 20.6: Representação de uma fila em alocação encadeada com cabeça.

Consideramos, como sempre fizemos, que as células da pilha são do tipo abaixo:

```
typedef struct cel {
   int chave;
   struct cel *prox;
} celula;
```

A inicialização de uma fila vazia em alocação encadeada com cabeça é dada a seguir:

```
   celula *i, *f;
   i = (celula *) malloc(sizeof (celula));
   i->prox = NULL;
   f = i;
```

e sem cabeça é dada a seguir:

```
   celula *i, *f;
   i = NULL;
   f = NULL;
```

Exemplos de filas vazias com cabeça e sem cabeça são mostrados na figura 20.7.

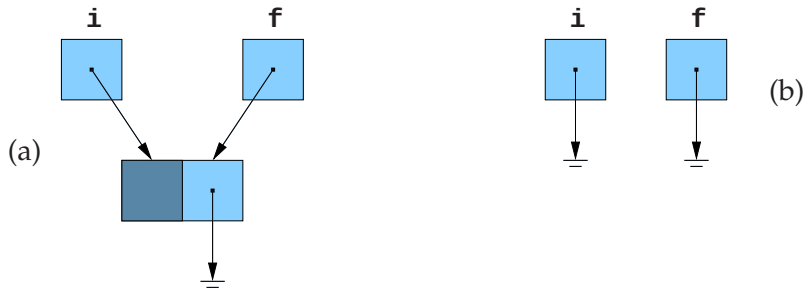



Figura 20.7: Representação de uma fila em alocação encadeada (a) com cabeça e (b) sem cabeça.

As operações básicas em uma fila com cabeça são dadas abaixo.

```
/* Recebe uma fila i e f e uma chave y e enfileira y na fila */
void enfileira_enc_C(celula *i, celula **f, int y)
{
   celula *nova;

   nova = (celula *) malloc(sizeof (celula));
   nova->chave = y;
   nova->prox = NULL;
   (*f)->prox = nova;
   *f = nova;
}
```

```
/* Recebe uma fila i e f e desenfileira um elemento da fila */
int desenfileira_enc_C(celula *i, celula **f)
{
   int x;
   celula *p;

   x = INT_MIN;
   p = i->prox;
   if (p != NULL) {
      x = p->chave;
      i->prox = p->prox;
      free(p);
      if (i->prox == NULL)
         *f = i;
   }
   else
      printf("Fila vazia!\n");
   return x;
}
```

## Exercícios

20.1 Implemente as funções de enfileiramento e desenfileiramento em uma fila em alocação encadaeda sem cabeça.

20.2 Um estacionamento possui um único corredor que permite dispor 10 carros. Os carros chegam pelo sul do estacionamento e saem pelo norte. Se um cliente quer retirar um carro que não está próximo do extremo norte, todos os carros impedindo sua passagem são retirados, o cliente retira seu carro e os outros carros são recolocados na mesma ordem que estavam originalmente. Sempre que um carro sai, todos os carros do sul são movidos para frente, de modo que as vagas fiquem disponíveis sempre no extremo sul do estacionamento. Escreva um algoritmo que processa o fluxo de chegada/saída deste estacionamento. Cada entrada para o algoritmo contém uma letra 'E' para entrada ou 'S' para saída, e o número da placa do carro. Considere que os carros chegam e saem pela ordem especificada na entrada. O algoritmo deve imprimir uma mensagem sempre que um carro chega ou sai. Quando um carro chega, a mensagem deve especificar se existe ou não vaga para o carro no estacionamento. Se não existe vaga, o carro deve esperar até que exista uma vaga, ou até que uma instrução fornecida pelo usuário indique que o carro deve partir sem que entre no estacionamento. Quando uma vaga torna-se disponível, outra mensagem deve ser impressa. Quando um carro sai do estacionamento, a mensagem deve incluir o número de vezes que o carro foi movimentado dentro da garagem, incluindo a saída mas não a chegada. Este número é 0 se o carro partiu da linha de espera, isto é, se o carro esperava uma vaga, mas partiu sem entrar no estacionamento.

20.3 Implemente uma fila usando duas pilhas.

20.4 Implemente uma pilha usando duas filas.

20.5 Suponha que temos $n$ cidades numeradas de 0 a $n-1$ e interligadas por estradas de mão única. As ligações entre as cidades são representadas por uma matriz $A$ definida da seguinte forma: $A[x][y]$ vale 1 se existe estrada da cidade $x$ para a cidade $y$ e vale 0 em caso contrário. A figura 20.8 ilustra um exemplo. A **distância** de uma cidade $o$ a uma cidade $x$ é o menor número de estradas que é preciso percorrer para ir de $o$ a $x$. O problema que queremos resolver é o seguinte: determinar a distância de uma dada cidade $o$ a cada uma das outras cidades da rede. As distâncias são armazenadas em um vetor $d$ de tal modo que $d[x]$ seja a distância de $o$ a $x$. Se for impossível chegar de $o$ a $x$, podemos dizer que $d[x]$ vale $\infty$. Usamos ainda $-1$ para representar $\infty$ uma vez que nenhuma distância "real" pode ter valor $-1$.

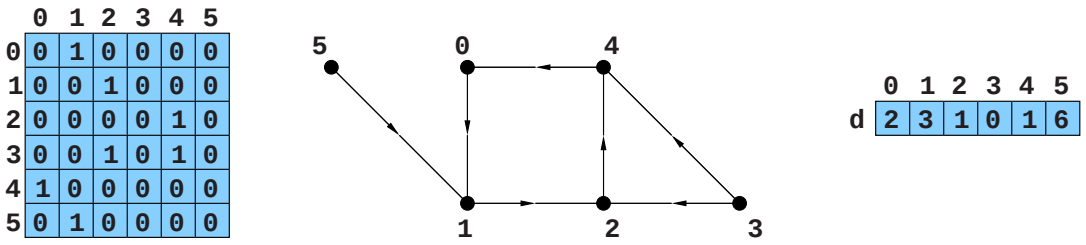



Figura 20.8: A matriz representa cidades $0, \ldots, 5$ interligadas por estradas de mão única. O vetor $d$ dá as distâncias da cidade 3 a cada uma das demais.

Solucione o problema das distâncias em uma rede usando uma fila em alocação seqüencial. Solucione o mesmo problema usando uma fila em alocação encadeada.

20.6 Imagine um tabuleiro quadrado 10–por–10. As casas "livres" são marcadas com 0 e as casas "bloqueadas" são marcadas com $-1$. As casas $(1,1)$ e $(10,10)$ estão livres. Ajude

uma formiga que está na casa $(1, 1)$ a chegar à casa $(10, 10)$. Em cada passo, a formiga só pode se deslocar para uma casa livre que esteja à direita, à esquerda, acima ou abaixo da casa em que está.

20.7 Um **deque** é uma lista linear que permite a inserção e a remoção de elementos em ambos os seus extremos. Escreva quatro funções para manipular um deque: uma que realiza a inserção de um novo elemento no início do deque, uma que realiza a inserção de um novo elemento no fim do deque, uma que realiza a remoção de um elemento no início do deque e uma que realiza a remoção de um elemento no fim do deque.

20.8 Suponha que exista um único vetor $M$ de células de um tipo fila pré-definido, com um total de **MAX** posições. Este vetor fará o papel da memória do computador. Este vetor $M$ será compartilhado por duas filas em alocação seqüencial. Implemente eficientemente as operações de enfileiramento e desenfileiramento para as duas filas de modo que nenhuma delas estoure sua capacidade de armazenamento, a menos que o total de elementos em ambas as filas seja **MAX**.

AULA 21

# LISTAS LINEARES CIRCULARES

Algumas operações básicas em listas lineares, em especial aquelas implementadas em alocação encadeada, não são muito eficientes. Um exemplo emblemático desse tipo de operação é a busca, mais eficiente e mais econômica em listas lineares em alocação seqüencial como pudemos perceber em aulas anteriores. As listas lineares circulares realizam algumas operações ainda mais eficientemente, já que possuem uma informação a mais que as listas lineares ordinárias.

Uma lista linear circular é, em geral, implementada em alocação encadeada e difere de uma lista linear por não conter um ponteiro para nulo em qualquer de suas células, o que não permite que se identifique um fim, ou mesmo um começo, da lista. Caso exista uma célula cabeça da lista, então é possível identificar facilmente um início e um fim da lista. Nesta aula, baseada nas referências [7, 13], veremos a implementação das operações básicas de busca, inserção e remoção em listas lineares circulares com cabeça em alocação encadeada.

## 21.1 Operações básicas em alocação encadeada

A adição de uma informação na estrutura de uma lista linear em alocação encadeada permite que operações sejam realizadas de forma mais eficiente. Quando a última célula da lista linear encadeada aponta para a primeira célula, temos uma **lista linear circular**. Dessa forma, não podemos identificar um fim da lista linear. No entanto, o que aparenta uma dificuldade é, na verdade, uma virtude desse tipo de estrutura, como veremos nas funções que implementam suas operações básicas de busca, inserção e remoção. Veja a figura 21.1 para um exemplo de uma lista linear circular com cabeça em alocação encadeada.

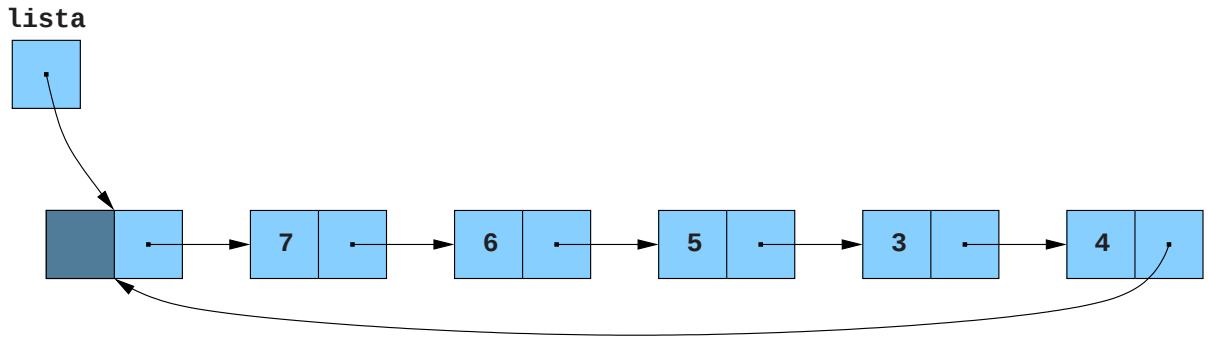



Figura 21.1: Representação de uma lista linear circular com cabeça em alocação encadeada.

A definição de um tipo célula da lista linear circular permanece a mesma:

```
typedef struct cel {
   int chave;
   struct cel *prox;
} celula;
```

A declaração e inicialização de uma lista linear circular com cabeça em alocação encadeada é realizada da seguinte forma:

```
   celula *lista;
   lista = (celula *) malloc(sizeof (celula));
   lista->prox = lista;
```

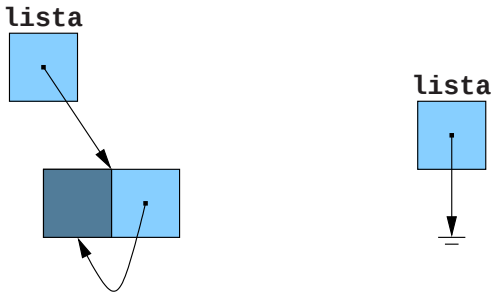
ou ainda:

```
   celula c, *lista;
   c.prox = &c;
   lista = &c;
```

e sem cabeça:

```
   celula *lista;
   lista = NULL;
```

A figura 21.2 mostra uma lista linear circular vazia (a) com cabeça e (b) sem cabeça.




(a) (b)

Figura 21.2: Representação de uma lista linear circular vazia (a) com cabeça e (b) sem cabeça.

A seguir vamos implementar as operações básicas de busca, inserção e remoção sobre listas lineares circulares com cabeça em alocação encadeada. A implementação das operações básicas em uma lista linear circular sem cabeça em alocação encadeada é deixada como exercício.

### 21.1.1 Busca

Imagine que queremos verificar se uma chave está presente em uma lista linear circular em alocação encadeada. A função `busca_circular` recebe um ponteiro `c` para a lista linear circular e um valor `x` e devolve uma célula contendo a chave procurada `x`, caso `x` ocorra na lista. Caso contrário, a função devolve `NULL`.

```
/* Recebe uma chave x e uma lista linear circular c e verifica se x consta em
   c, devolvendo a célula em que x se encontra ou NULL em caso contrário */
celula *busca_circular(int x, celula *c)
{
   celula *p, *q;

   p = c;
   p->chave = x;
   q = c->prox;
   while (q->chave != x)
      q = q->prox;
   if (q != c)
      return q;
   else
      return NULL;
}
```

Observe que a busca em uma lista linear circular em alocação encadeada é muito semelhante à busca em uma lista linear ordinária da aula 18. Em particular, seu tempo de execução de pior caso é proporcional ao número de células da lista, assim como é o tempo de execução de pior caso da busca em uma lista linear ordinária. Observe, no entanto, que as constantes são diferentes: enquanto na busca em uma lista linear ordinária a estrutura de repetição sempre realiza duas comparações, a estrutura de repetição da busca em uma lista circular realiza apenas uma comparação.

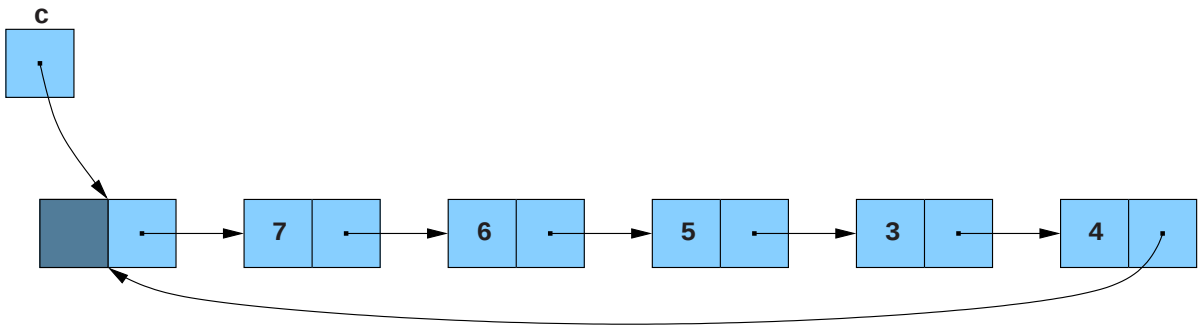
### 21.1.2 Inserção

A inserção em uma lista linear circular com cabeça em alocação encadeada é simples e semelhante à inserção em uma lista linear ordinária. Temos apenas de prestar atenção para que a inserção sempre se mantenha circular, o que se dá de modo natural se inserimos a célula que contém a nova chave sempre como a célula seguinte à cabeça.

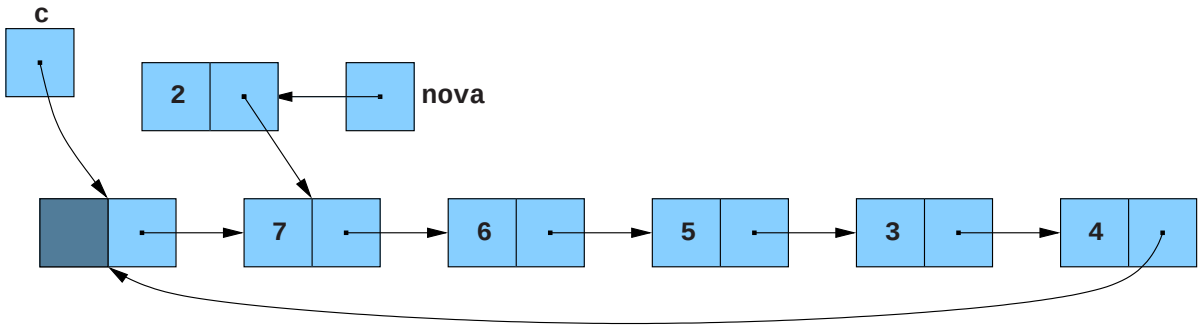
```
/* Recebe uma chave y e uma lista circular c e insere y no início da lista c */
void insere_circular(int y, celula *c)
{
   celula *nova;

   nova = (celula *) malloc(sizeof (celula));
   nova->chave = y;
   nova->prox = c->prox;
   c->prox = nova;
}
```

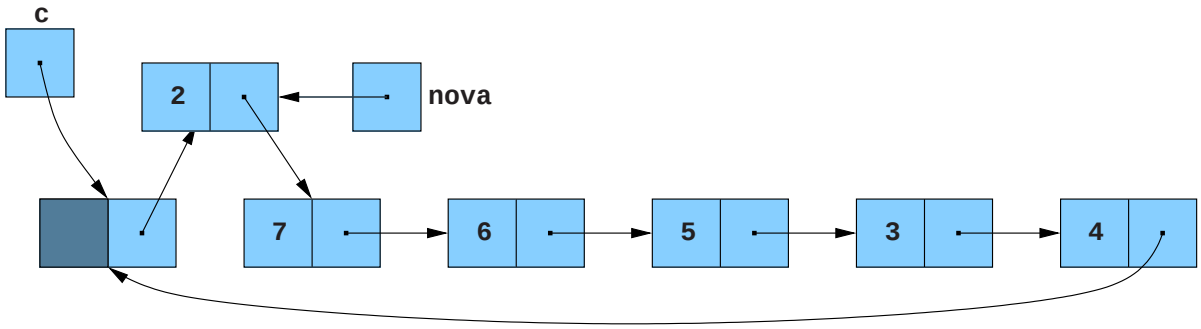
A figura 21.3 mostra a execução da função `insere_circular` para uma chave 2 e uma lista circular `c`.




(a)




(b)




(c)

Figura 21.3: Inserção da chave 2 em uma lista circular `c`.

### 21.1.3 Remoção

A função de remoção de um elemento em uma lista linear circular em alocação encadeada é também muito semelhante à remoção de um elemento realizada em uma lista linear ordinária. Buscamos a célula a ser removida como sendo aquela que possui a chave `x`, fornecida como parâmetro para a função.

```
/* Recebe uma chave x e uma lista linear circular c e
   remove a célula com chave x da lista c, caso exista */
void remove_circular(int x, celula *c)
{
   celula *p, *q;

   p = c;
   p->chave = x;
   q = p->prox;
   while (q->chave != x) {
      p = q;
      q = q->prox;
   }
   if (q != c) {
      p->prox = q->prox;
      free(q);
   }
}
```

A figura 21.4 mostra a execução de `remove_circular` para a chave 4 e a lista circular `c`.

## Exercícios

21.1 Escreva funções que implementem as operações básicas de busca, inserção e remoção sobre uma lista linear circular sem cabeça em alocação encadeada.

21.2 Escreva uma função para liberar todas as células de uma lista linear circular em alocação encadeada para a lista de espaços disponíveis da memória. Escreva duas versões dessa função, uma para uma lista linear circular com cabeça e outra para uma lista linear circular sem cabeça.

21.3 Suponha que queremos implementar uma lista linear circular em alocação encadeada de tal forma que as chaves de suas células sempre permaneçam em ordem crescente. Escreva as funções para as operações básicas de busca, inserção e remoção para listas lineares circulares em alocação encadeada com as chaves em ordem crescente. Escreva dois conjuntos de funções, considerando listas lineares circulares com cabeça e listas lineares circulares sem cabeça.

21.4 O **problema de Josephus** foi assim descrito através do relato de Flavius Josephus, um historiador judeu que viveu no primeiro século, sobre o cerco de Yodfat. Ele e mais 40 soldados aliados estavam encurralados em uma caverna rodeada por soldados romanos e, não havendo saída, optaram então por suas próprias mortes antes da captura. Decidiram que formariam um círculo e, a cada contagem de 3, um soldado seria morto pelo grupo. Josephus foi o soldado restante e decidiu então não se suicidar, deixando essa estória como legado. Podemos descrever um problema mais geral como segue. Imagine $n$ pessoas dispostas em círculo. Suponha que as pessoas são numeradas de 1 a $n$ no sentido horário e que um número inteiro $m$, com $1 \leqslant m < n$, seja fornecido. Começando com a pessoa de número 1, percorra o círculo no sentido horário e elimine cada $m$-ésima pessoa enquanto o círculo tiver duas ou mais pessoas. Escreva e teste uma função que resolva o problema, imprimindo na saída o número do sobrevivente.

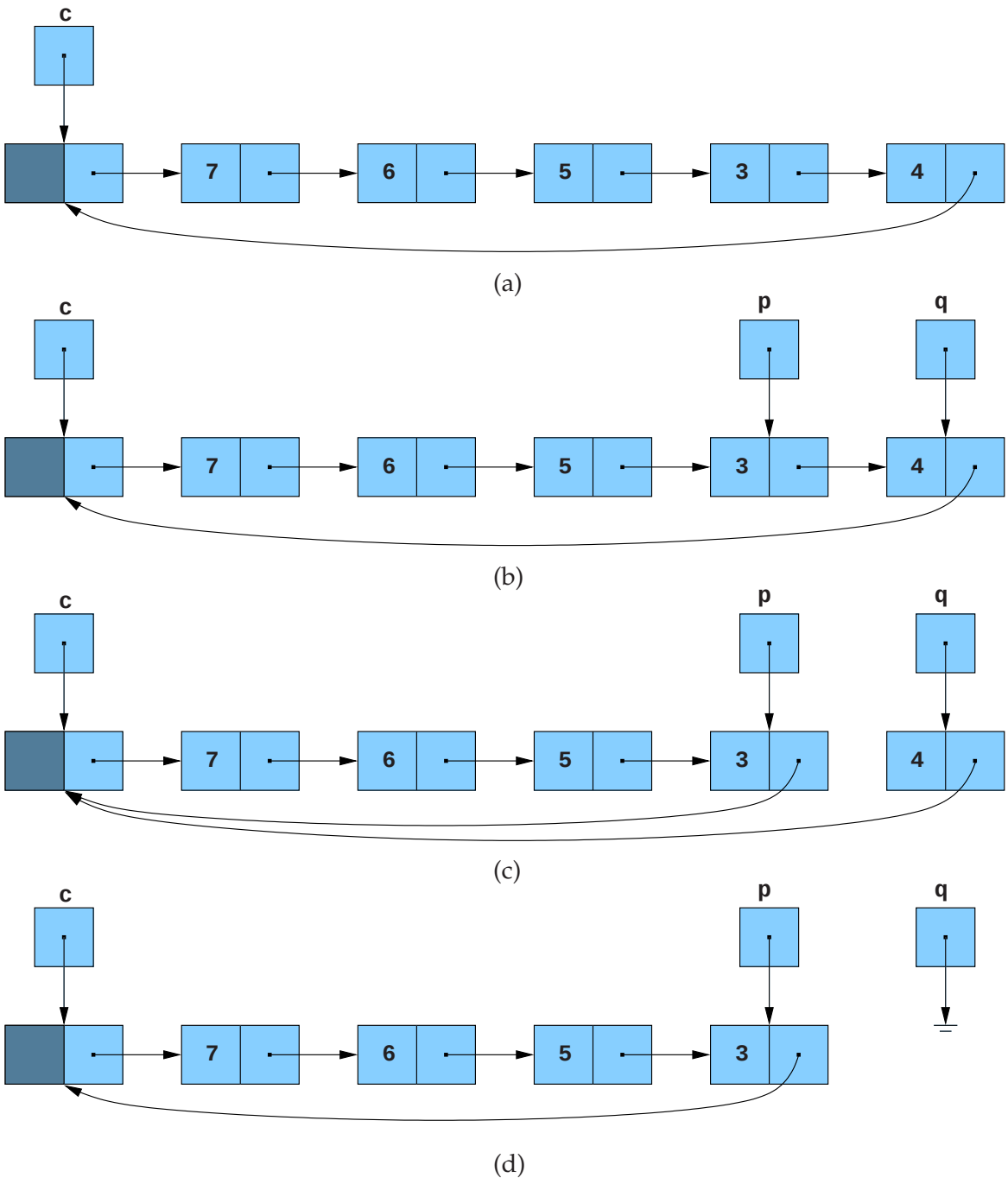



Figura 21.4: Remoção da chave 4 de uma lista circular `c`.

21.5 Um computador permite representar números inteiros até um valor máximo, que depende da sua organização e arquitetura interna. Suponha que seja necessário, em uma dada aplicação, o uso de números inteiros com precisão muito grande, digamos ilimitada. Uma forma de representar números inteiros com precisão ilimitada é mostrada na figura 21.5.

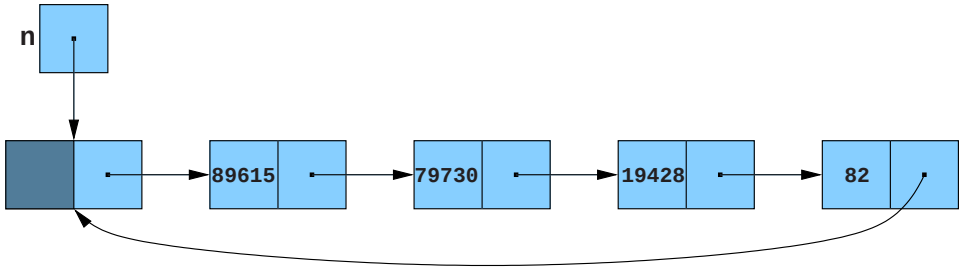



Figura 21.5: Uma lista linear circular com cabeça em alocação encadeada que armazena o número inteiro 82194287973089615.

O tipo célula, que define as células da lista linear circular, não sofre alterações na sua definição, sendo o mesmo o tipo `celula` previamente definido. Observe apenas que uma chave contém um número inteiro de até 5 algarismos.

Escreva uma função que receba duas listas lineares circulares contendo dois números inteiros de precisão ilimitada e devolva uma lista linear circular contendo o resultado da soma desses dois números.

AULA 22

# LISTAS LINEARES DUPLAMENTE ENCADEADAS

Quando trabalhamos com listas lineares encadeadas, muitas vezes precisamos de manter um ponteiro para uma célula e também para a célula anterior para poder realizar algumas operações sobre a lista, evitando percursos adicionais desnecessários. Esse é o caso, por exemplo, da busca e da inserção. No entanto, algumas vezes isso não é suficiente, já que podemos precisar percorrer a lista linear nos dois sentidos. Neste caso, para solucionar eficientemente esse problema, adicionamos um novo campo ponteiro nas células da lista, que aponta para a célula anterior da lista. O gasto de memória imposto pela adição deste novo campo se justifica pela economia em não ter de reprocessar a lista linear inteira nas operações de interesse. Esta aula é baseada nas referências [7, 13].

## 22.1 Definição

As células de uma lista linear duplamente encadeada são ligadas por ponteiros que indicam a posição da célula anterior e da próxima célula da lista. Assim, um outro campo é acrescentado à cada célula da lista indicando, além do endereço da próxima célula, o endereço da célula anterior da lista. Veja a figura 22.1.

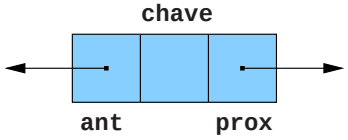



Figura 22.1: Representação de uma célula de uma lista linear duplamente encadeada.

A definição de uma célula de uma lista linear duplamente encadeada é então descrita como um tipo, como mostramos a seguir:

```
typedef struct cel {
   int chave;
   struct cel *ant;
   struct cel *prox;
} celula;
```

Uma representação gráfica de uma lista linear em alocação encadeada é mostrada na figura 22.2.

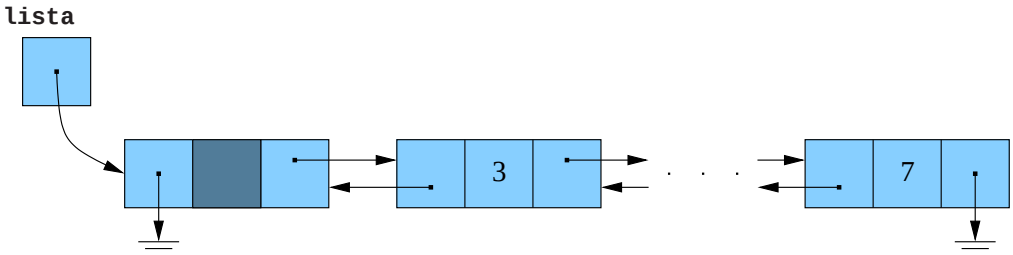



Figura 22.2: Representação de uma lista linear duplamente encadeada com cabeça **lista**. A primeira célula tem seu campo **ant** apontando para **NULL** e a última tem seu campo **prox** também apontando para **NULL**.

Uma lista linear duplamente encadeada com cabeça pode ser criada e inicializada da seguinte forma:

```
celula *lista;
lista = (celula *) malloc(sizeof (celula));
lista->ant = NULL;
lista->prox = NULL;
```

e sem cabeça como abaixo:

```
celula *lista;
lista = NULL;
```

A figura 22.3 ilustra a declaração e inicialização de uma lista linear duplamente encadeada (a) com cabeça e (b) sem cabeça.

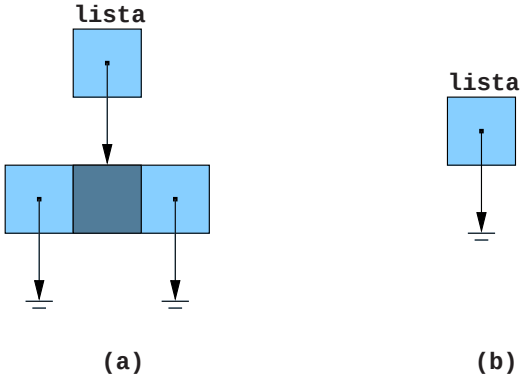



Figura 22.3: Lista linear duplamente encadeada vazia (a) com cabeça e (b) sem cabeça.

A seguir vamos discutir e implementar as operações básicas sobre listas lineares duplamente encadeadas com cabeça, deixando as liestas lineares sem cabeça para os exercícios.

## 22.2 Busca

A função `busca_dup_C` recebe uma lista linear duplamente encadeada `lst` e um valor `x` e devolve uma célula que contém a chave procurada `x`, caso `x` ocorra em `lst`. Caso contrário, a função devolve `NULL`.

```
/* Recebe uma chave x e um alista linear duplamente encadeada lst
   e devolve a célula que contém x em lst ou NULL caso contrário */
celula *busca_dup_C(int x, celula *lst)
{
   celula *p;

   p = lst->prox;
   while (p != NULL && p->chave != x)
      p = p->prox;

   return p;
}
```

Observe que a função `busca_dup_C` é praticamente uma cópia da função `busca_C`.

## 22.3 Busca seguida de remoção

No primeiro passo da busca seguida de remoção, uma busca de uma chave é realizada na lista. Se a busca tem sucesso, então o ponteiro para a célula que ser quere remover também dá acesso à célula posterior e anterior da lista linear duplamente encadeada. Isso é tudo o que precisamos saber para realizar o segundo passo, isto é, realizar a remoção satisfatoriamente. A função `busca_remove_dup_C` implementa essa idéia.

```
/* Recebe um número inteiro x e uma lista duplamente encadeada lst e remo-
   ve da lista a primeira célula que contiver x, se tal célula existir */
void busca_remove_dup_C(int x, celula *lst)
{
   celula *p;

   p = lst->prox;
   while (p != NULL && p->chave != x)
      p = p->prox;
   if (p != NULL) {
      p->ant->prox = p->prox;
      if (p->prox != NULL)
         p->prox->ant = p->ant;
      free(p);
   }
}
```

A função `busca_remove_dup_C` é muito semelhante à função `busca_remove_C`, mas não há necessidade do uso de dois ponteiros para duas células consectivas da lista. A figura 22.4

ilustra um exemplo de execução dessa função.

## 22.4 Busca seguida de inserção

A função `busca_insere_dup_C` a seguir realiza a busca de uma chave `x` e insere uma chave `y` antes da primeira ocorrência de `x` em uma lista linear duplamente encadeada `lst`. Caso `x` não ocorra na lista, a chave `y` é inserida no final de `lst`.

```
/* Recebe dois números inteiros y e x e uma lista duplamente
    encadeada lst e insere uma nova célula com chave y nessa
   lista antes da primeira que contiver x; se nenhuma célula
   contiver x, a nova célula é inserida no final da lista */
void busca_insere_dup_C(int y, int x, celula *lst)
{
   celula *p, *q, *nova;

   nova = (celula *) malloc(sizeof (celula));
   nova->chave = y;
   p = lst;
   q = lst->prox;
   while (q != NULL && q->chave != x) {
      p = q;
      q = q->prox;
   }
   nova->ant = p;
   nova->prox = q;
   if (p != NULL)
      p->prox = nova;
   if (q != NULL)
      q->ant = nova;
}
```

A função `busca_insere_dup_C` é também muito semelhante à função `busca_insere_C`, mas não há necessidade de manter dois ponteiros como na primeira função. Veja a figura 22.5.

## Exercícios

22.1 Escreva funções para implementar as operações básicas de busca, inserção e remoção em uma lista linear duplamente encadeada sem cabeça.

22.2 Escreva uma função que receba um ponteiro para o início de uma lista linear em alocação duplamente encadeada, um ponteiro para o fim dessa lista e uma chave, e realize a inserção dessa chave no final da lista.

22.3 Listas lineares duplamente encadeadas também podem ser listas circulares. Basta olhar para uma lista linear duplamente encadeada e fazer o ponteiro para a céula anterior da célula cabeça apontar para a última célula e a ponteiro para a próxima célula da última

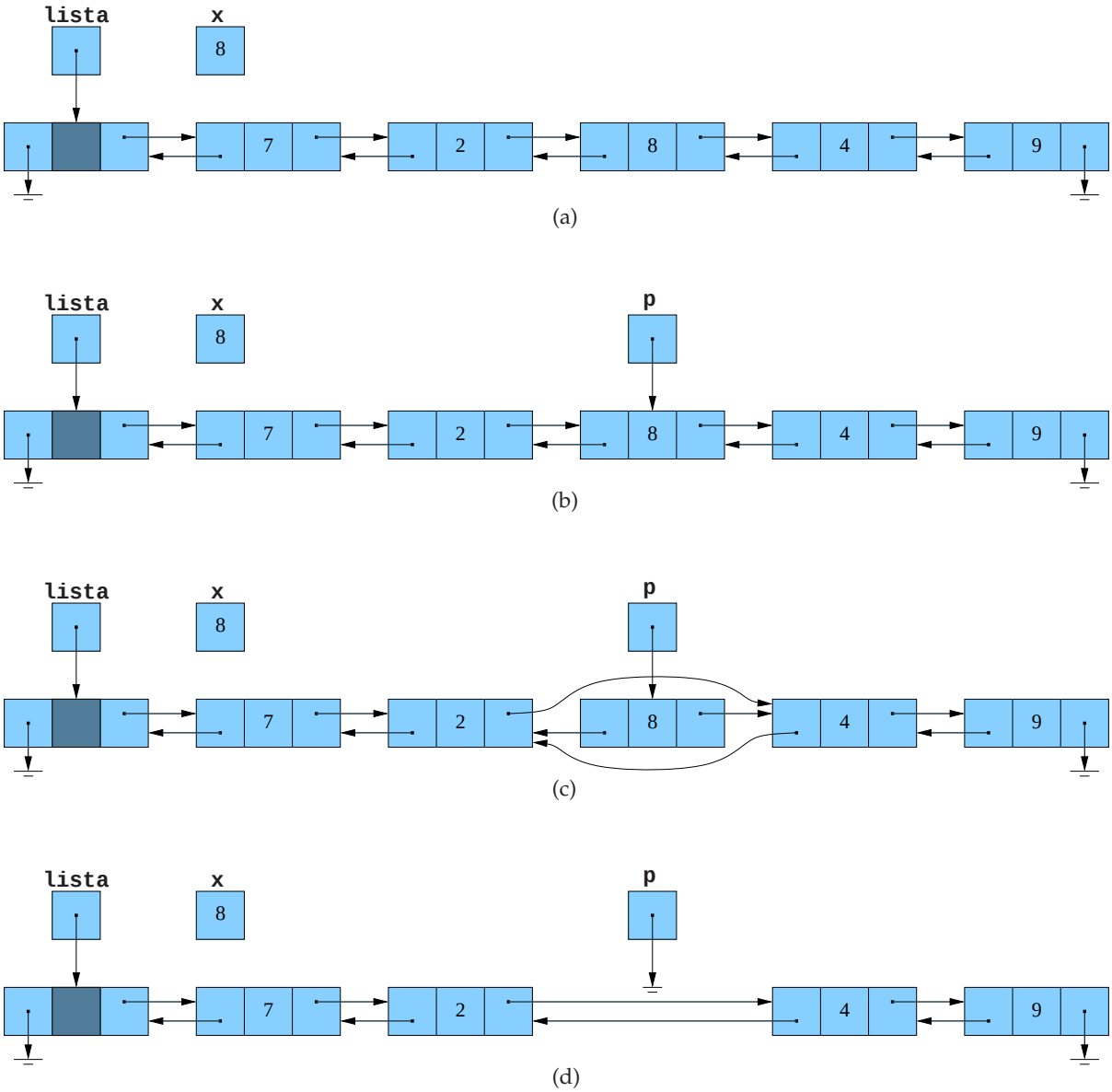



Figura 22.4: Remoção em uma lista linear duplamente encadeada com cabeça.

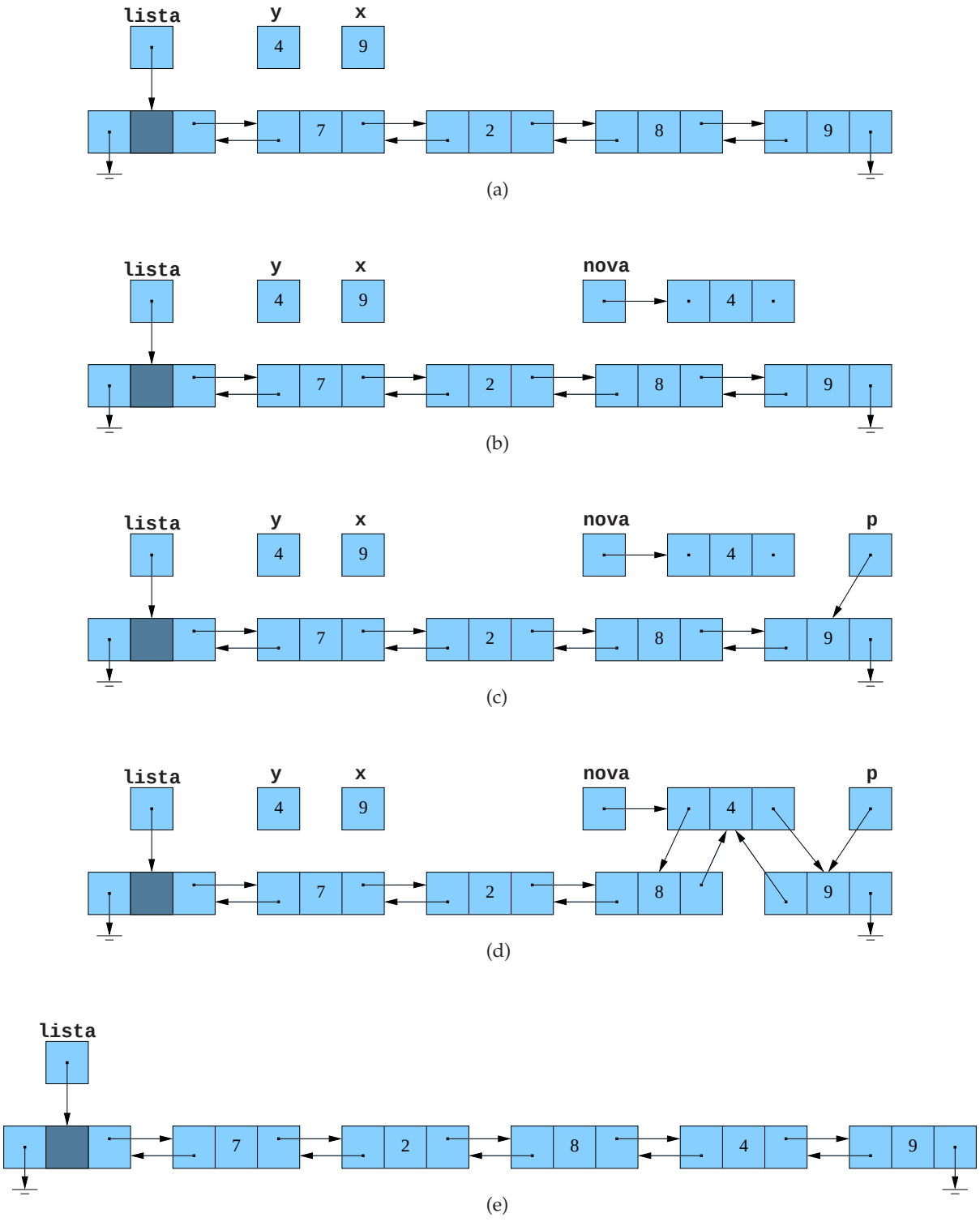



Figura 22.5: Inserção em uma lista linear duplamente encadeada com cabeça.

célula apontar para a célula cabeça. Escreva funções para implementar as operações básicas de busca, inserção e remoção em uma lista linear duplamente encadeada circular com cabeça.

22.4 Considere uma lista linear duplamente encadeada contendo as chaves $c_1, c_2, \ldots, c_n$, como na figura 22.6.

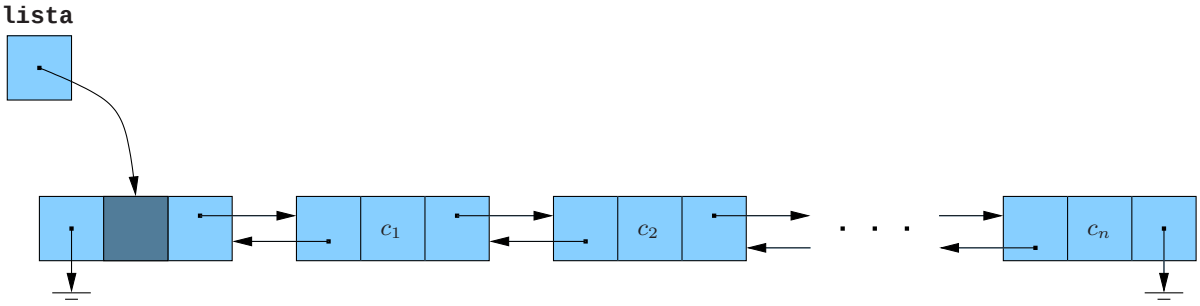



Figura 22.6: Uma lista linear duplamente encadeada.

Escreva uma função que altere os ponteiros `ant` e `prox` da lista, *sem* mover suas informações, tal que a lista fique invertida como na figura 22.7.

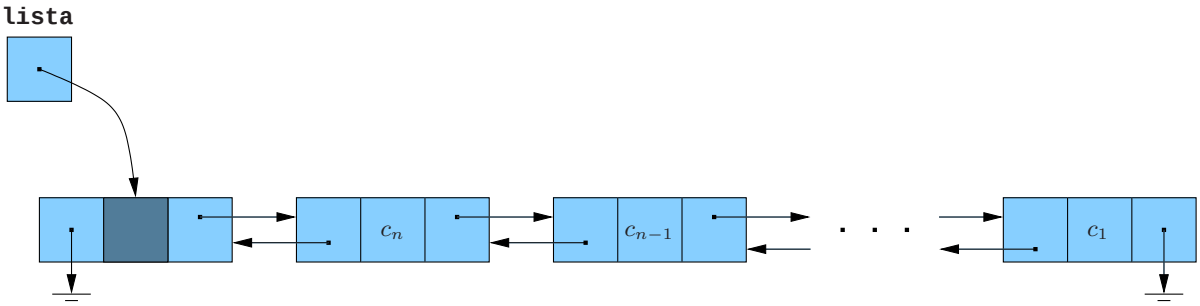



Figura 22.7: Inversão da lista linear duplamente encadeada da figura 22.6.

22.5 Suponha que você queira manter uma lista linear duplamente encadeada com cabeça com as chaves em ordem crescente. Escreva as funções de busca, inserção e remoção para essa lista.

AULA 23

# TABELAS DE ESPALHAMENTO

Donald E. Knuth, em seu livro *The Art of Computer Programming*, observou que o cientista da computação Hans P. Luhn parece ter sido o primeiro a usar o conceito de espalhamento[1] em janeiro de 1953. Também observou que Robert H. Morris, outro cientista da computação, usou-o em um artigo da *Communications of the ACM* de 1968, tornando o termo espalhamento formal e conhecido na academia. O termo com o sentido que enxergamos na Computação vem da analogia com a mesma palavra da língua inglesa, um termo não-técnico que significa "corte e misture" ou "pique e misture".

Muitos problemas podem ser resolvidos através do uso das operações básicas de busca, inserção e remoção sobre as suas entradas. Por exemplo, um compilador para uma linguagem de programação mantém uma tabela de símbolos em memória onde as chaves são cadeias de caracteres que correspondem às palavras-chaves da linguagem. Outro exemplo menos abstrato é o de armazenamento das correspondências em papel dos funcionários de uma empresa. Se a empresa é grande, em geral não é possível manter um compartimento para cada funcionário(a), sendo mais razoável manter, por exemplo, 26 compartimentos rotulados com as letras do alfabeto. As correspondências de um dado funcionário(a), cujo nome inicia com uma dada letra, estão armazenadas no compartimento rotulado com aquela letra. A implementação de uma tabela de espalhamento para tratar das operações básicas de busca, inserção e remoção é uma maneira eficiente de solucionar problemas como esses. De fato, uma tabela de espalhamento é uma generalização da noção de vetor, como veremos.

Nesta aula, baseada nas referências [1, 13], estudaremos essas estruturas de dados.

## 23.1 Tabelas de acesso direto

Suponha que temos uma aplicação em que cada informação seja identificada por uma chave e que as operações essenciais realizadas sobre essas informações sejam a busca, inserção e remoção. Podemos supor, sem perda de generalidade, que as chaves sejam apenas chaves numéricas, já que é sempre fácil associar um número inteiro a cada informação. Como as outras informações associadas às chaves são irrelevantes para as operações básicas necessárias à aplicação, podemos descartá-las e trabalhar apenas com as chaves. Suponha que todas as chaves possíveis na aplicação pertençam ao conjunto $\mathscr{C} = \{0, 1, \ldots, m-1\}$, onde $m$ é um número inteiro relativamente pequeno.

[1] Do inglês *hash* ou *hashing*. Optamos pela tradução "espalhamento" e "tabela de espalhamento" usada pelo professor Tomasz Kowaltowsky, do IC-UNICAMP. O professor Jayme Luiz Szwarcfiter, da COPPE-UFRJ, usa os termos "dispersão" e "tabela de dispersão".

Então, usamos uma **tabela de acesso direto** para representar esse conjunto de informações, que nada mais é que um vetor $T[0..m-1]$ do tipo inteiro, onde o índice de cada compartimento de $T$ corresponde a uma chave do conjunto $\mathscr{C}$. Em um dado momento durante a execução da aplicação, um determinado conjunto de chaves $C \subseteq \mathscr{C}$ está representado na tabela $T$. Veja a figura 23.1 para uma ilustração.

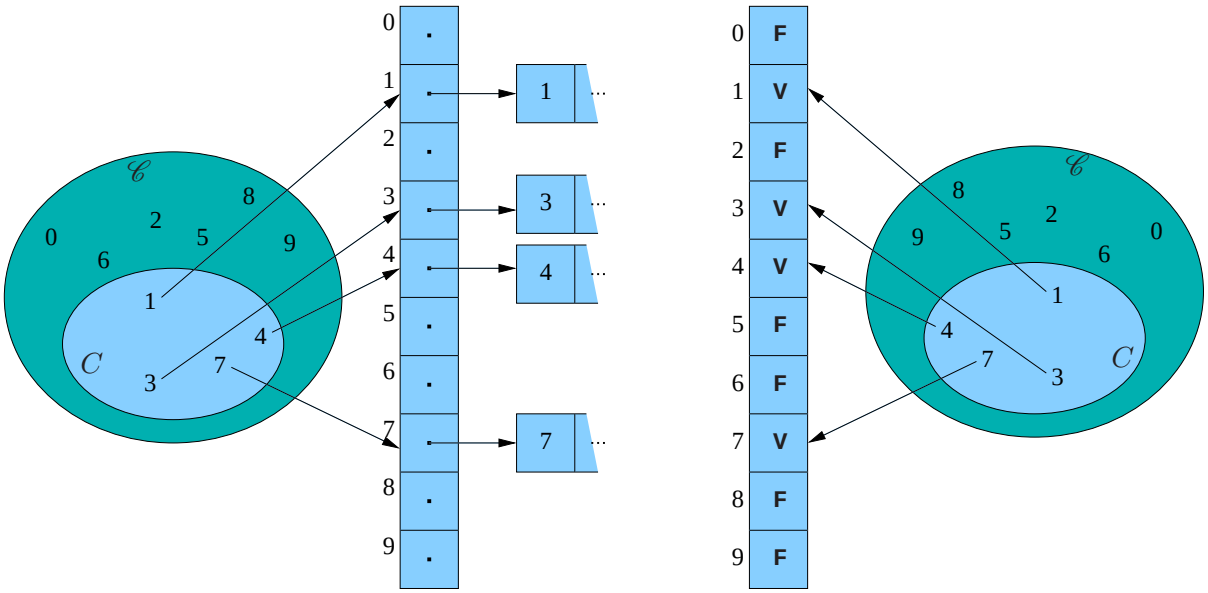



Figura 23.1: Dois exemplos de tabela de acesso direto. (a) Cada compartimento da tabela armazena um ponteiro para um registro contendo a chave e outras informações relevantes. (b) Cada compartimento da tabela armazena um bit indicando se a chave representada no índice do vetor está presente.

É fácil ver que as operações básicas de busca, remoção e inserção são realizadas em tempo constante em uma tabela de acesso direto, considerando que as chaves são todas distintas. O problema com esta estratégia é a impossibilidade evidente de alocação de espaço na memória quando o conjunto de todas as possíveis chaves $\mathscr{C}$ contém alguma chave que é um número inteiro muito grande. Ademais, se há poucas chaves representadas no conjunto $C$ em um dado instante da aplicação, a tabela de acesso direto terá, nesse instante, muito espaço alocado mas não usado.

## 23.2 Introdução às tabelas de espalhamento

Para tentar solucionar eficientemente o problema conseqüente do acesso direto, usamos uma tabela de espalhamento. Em uma tabela de acesso direto, a chave $k$ é armazenada no compartimento de índice $k$. Em uma tabela de espalhamento, uma chave $x$ é armazenada no compartimento $h(x)$, onde a função $h\colon \mathscr{C} \to \{0, 1, \ldots, m-1\}$ é chamada uma **função de espalhamento**. A função $h$ mapeia as chaves de $\mathscr{C}$ nos compartimentos de uma **tabela de espalhamento** $T[0..m-1]$. O principal objetivo de uma função de espalhamento é reduzir o intervalo de índices da tabela de espalhamento.

A figura 23.2 mostra um exemplo de uma tabela de espalhamento associada a uma função de espalhamento.

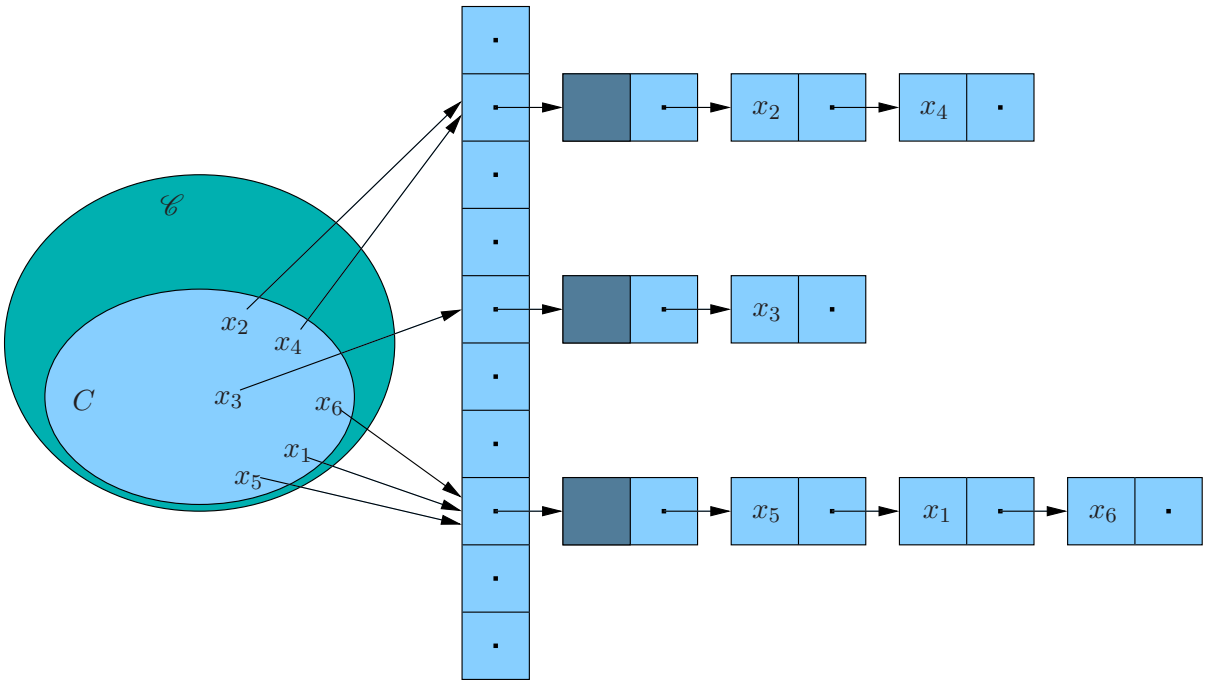



Figura 23.2: Exemplo de uma tabela de espalhamento $T$ associada a uma função de espalhamento $h$. Observe que ocorrem as colisões $h(x_5) = h(x_1) = h(x_6)$ e $h(x_2) = h(x_4)$ e que tais colisões são resolvidas com listas lineares encadeadas com cabeça.

A principal estratégia de uma tabela de espalhamento é fazer uso de uma boa função de espalhamento que permita distribuir bem as chaves pela tabela. Em geral, como $|\mathscr{C}| > m$, isto é, como o número de elementos do conjunto de chaves possíveis é maior que o tamanho da tabela $T$, duas ou mais chaves certamente terão o mesmo índice. Ou seja, devem existir pelo menos duas chaves, digamos $x_i$ e $x_j$, no conjunto das chaves possíveis $\mathscr{C}$ tais que $h(x_i) = h(x_j)$, como vimos na figura 23.2. Quando duas chaves $x_i$ e $x_j$ possuem essa propriedade, dizemos que há, ou pode haver, uma **colisão** entre $x_i$ e $x_j$ na tabela de espalhamento $T$. Uma maneira bastante evidente de solucionar o problema das colisões é manter uma lista linear encadeada com cabeça para cada subconjunto de colisões.

Observe que se a função de espalhamento é muito ruim, podemos ter um caso degenerado em que todas as chaves têm o mesmo índice, isto é, $h(x_1) = h(x_2) = \cdots = h(x_n)$ para toda chave $x_i \in C$, com $1 \leqslant i \leqslant n$ e $|C| = n$. Isso significa que temos, na verdade, uma tabela implementada como lista linear encadeada. Observe que esse caso é muito ruim especialmente porque todas as operações básicas, de busca, remoção e inserção, gastam tempo proporcional ao número de chaves contidas na lista, isto é, tempo proporcional a $n$. Veja a figura 23.3 para um exemplo desse caso.

A situação ideal quando projetamos uma tabela de espalhamento é usar uma função de espalhamento que distribua bem as chaves por todos os $m$ compartimentos da tabela $T$. Dessa forma, uma tal função maximiza o número de compartimentos usados em $T$ e minimiza os comprimentos das listas lineares encadeadas que armazenam as colisões. Um exemplo de uma situação como essa é ilustrado na figura 23.4.

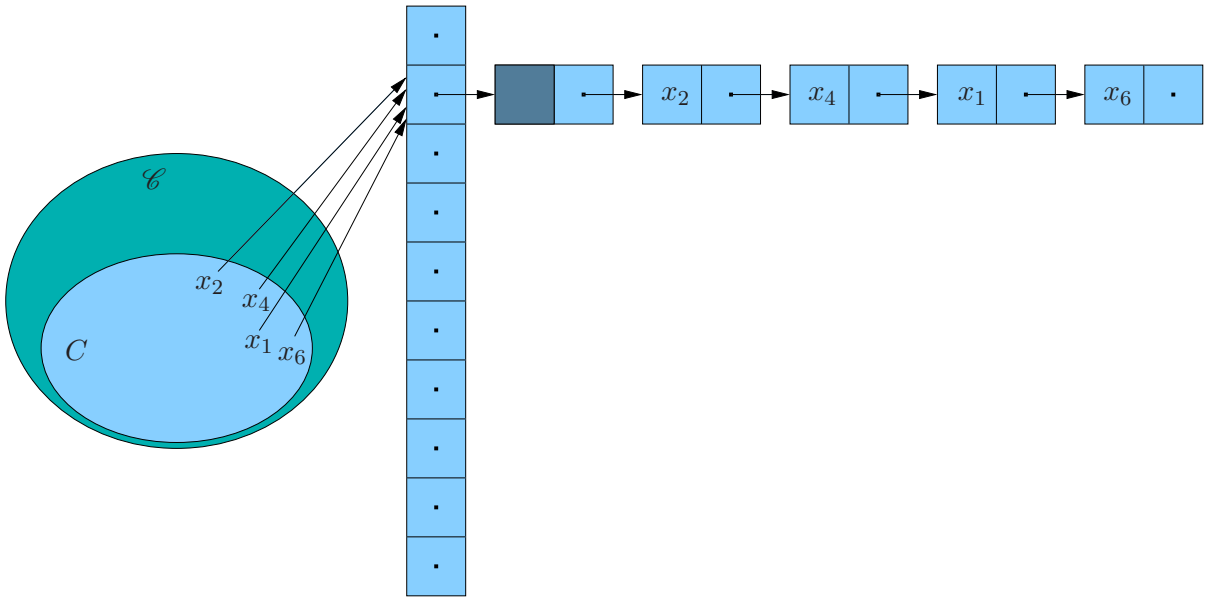



Figura 23.3: Exemplo de uma função de espalhamento "ruim", isto é, uma função que produz uma tabela de espalhamento degenerada.

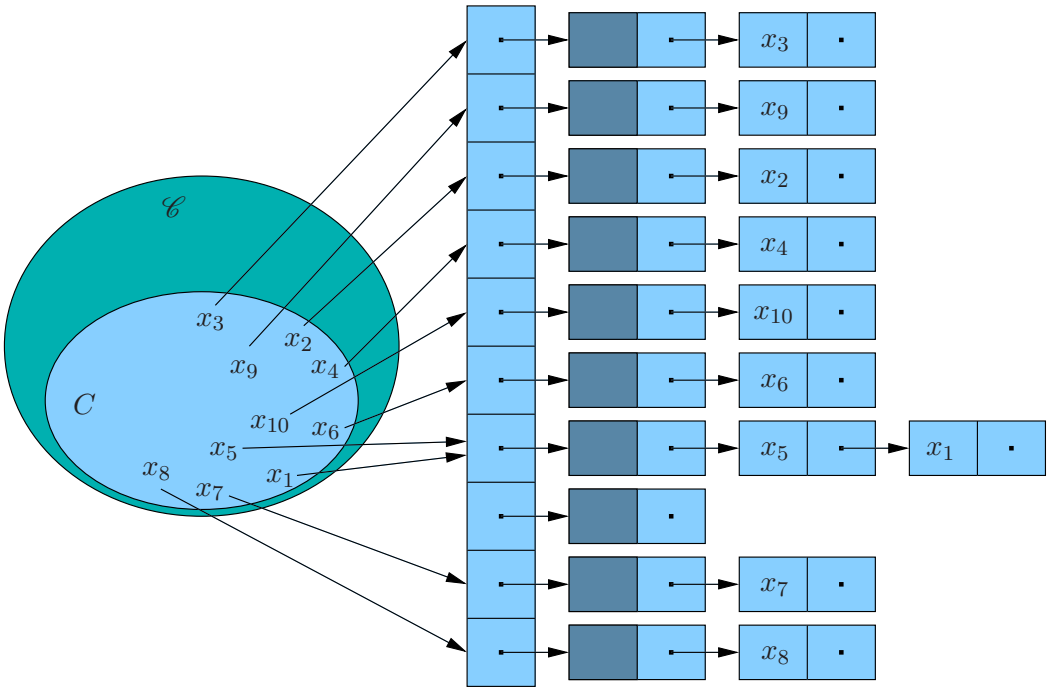



Figura 23.4: Exemplo de uma função de espalhamento próxima do ideal, que "espalha" bem as chaves pela tabela $T$, havendo poucas colisões e poucos compartimentos apontando para listas lineares vazias.

## 23.3 Tratamento de colisões com listas lineares encadeadas

Em uma tabela de espalhamento, o uso de listas lineares encadeadas é uma maneira intuitiva e eficiente de tratar colisões. Assim, as operações básicas de busca, remoção e inserção são operações de busca, remoção e inserção sobre listas lineares encadeadas, como vimos na aula 18. Apenas modificamos levemente a operação de inserção, que em uma tabela de espalhamento é sempre realizada no início da lista linear apropriada.

Se temos uma função de espalhamento, então a implementação de uma tabela de espalhamento com tratamento de colisões usando listas lineares encadeadas é um espelho da implementação de uma lista linear encadeada. Para analisar a eficiência dessa implementação, basta analisar o tempo de execução de cada uma das operações. Se considerarmos que todas elas necessitam de alguma forma da operação de busca, podemos fixar e analisar o tempo de execução apenas dessa operação.

O tempo de execução no pior caso da operação de busca é proporcional ao tamanho da lista linear encadeada associada. No pior caso, como já mencionamos, tal lista contém todas as $n$ chaves do conjunto $C$ e, assim, o tempo de execução de pior caso da operação de busca é proporcional a $n$. Dessa forma, à primeira vista parece que uma tabela de espalhamento é uma forma muito ruim de armazenar informações, já que o tempo de pior caso de cada uma das operações básicas é ruim.

No entanto, se estudamos o **tempo de execução no caso médio**, ou **tempo esperado de execução**, não o tempo de execução no pior caso, a situação se reverte em favor dessas estruturas. O tempo de execução no caso médio de uma busca em uma tabela de espalhamento é proporcional a $1 + \alpha$, onde $\alpha$ é chamado de **fator de carga** da tabela. Se a tabela de espalhamento $T$ tem tamanho $m$ e armazena $n$ chaves, então $\alpha$ é definido como $n/m$. Dessa forma, se a quantidade de chaves $n$ armazenada na tabela $T$ é proporcional ao tamanho $m$ de $T$, então as operações básicas são realizadas em tempo esperado constante.

### 23.3.1 Funções de espalhamento

Uma função de espalhamento deve, a princípio, satisfazer a suposição de que cada chave é igualmente provável de ser armazenada em qualquer um dos $m$ compartimentos da tabela de espalhamento $T$, independentemente de onde qualquer outra chave foi armazenada antes. Uma função de espalhamento que tem essa propriedade é chamada de **função de espalhamento simples e uniforme**. Essa suposição implica na necessidade de conhecimento da distribuição de probabilidades da qual as chaves foram obtidas, o que em geral não sabemos previamente.

O uso de heurísticas para projetar funções de espalhamento é bastante freqüente. Isso significa que não há garantia alguma de que uma tal função seja simples e uniforme, mas muitas delas são usadas na prática e funcionam bem na maioria dos casos. O método da divisão e o método da multiplicação são heurísticas que fornecem funções de espalhamento com desempenho prático satisfatório. Algumas aplicações, no entanto, exigem que as funções de espalhamento tenham garantias de que são simples e uniformes. O método universal de espalhamento gera uma classe de funções de espalhamento que satisfazem essa propriedade.

É comum que uma função de espalhamento tenha como domínio o conjunto dos números naturais. Se as chaves de uma aplicação não são números naturais, podemos facilmente

transformá-las em números naturais. Por exemplo, se as chaves são cadeias de caracteres, podemos somar o valor de cada caractere que compõe a cadeia, obtendo assim um número natural que representa esta chave.

Nesta aula, estudamos as heurísticas mencionadas acima, deixando que os leitores interessados no método universal de espalhamento consultem as referências desta aula.

### Método da divisão

O **método da divisão** projeta uma função de espalhamento mapeando o valor $x$ da chave em um dos $m$ compartimentos da tabela $T$. O método divide então $x$ por $m$ e toma o resto dessa divisão. Dessa forma, a função de espalhamento $h$ é dada por:

$$h(x) = x \bmod m \, .$$

Para que a função obtida seja o mais efetiva possível, devemos evitar certos valores de $m$ que podem tornar a função menos uniforme. Em geral, escolhemos um número primo grande não muito próximo a uma potência de 2.

Por exemplo, suponha que desejamos alocar uma tabela de espalhamento com colisões resolvidas por listas lineares encadeadas, que armazena aproximadamente $n = 1000$ cadeias de caracteres. Se não nos importa examinar em média 5 chaves em uma busca sem sucesso, então fazemos $m = 199$, já que $199$ é um número primo próximo a $1000/5$ e distante de uma potência de 2.

### Método da multiplicação

O **método da multiplicação** projeta funções de espalhamento da seguinte forma. Primeiro, multiplicamos a chave $x$ por uma constante de ponto flutuante $A$, com $0 < A < 1$, e tomamos a parte fracionária de $xA$. Depois, multiplicamos esse valor por $m$ e tomamos o piso do valor resultante. Ou seja, uma função de espalhamento $h$ é dada por:

$$h(x) = \lfloor m \left( xA - \lfloor xA \rfloor \right) \rfloor \, .$$

Diferentemente do método da divisão, o valor de $m$ no método da multiplicação não é o mais importante. Em geral, escolhemos $m = 2^p$ para algum número inteiro $p$, já que dessa forma podemos usar operações sobre bits para obter $h(x)$. No entanto, o método da multiplicação funciona melhor com alguns valores de $A$ do que com outros. Donald E. Knuth sugere que $A \approx (\sqrt{5} - 1)/2 = 0{,}6180339887\ldots$ seja um valor que forneça boas funções de espalhamento pelo método da multiplicação.

### Outros métodos

Algumas outras heurísticas para geração de funções de espalhamento podem ser citadas, como por exemplo, o método da dobra e o método da análise dos dígitos. Interessados devem procurar as referências desta aula. Além disso, o método universal escolhe ou gera funções aleatoriamente, de forma independente das chaves a serem armazenadas na tabela, garantindo assim um bom desempenho médio.

## 23.4 Endereçamento aberto

Em uma tabela de espalhamento com endereçamento aberto, as chaves são todas armazenadas na própria tabela. Por isso, muitas vezes uma tabela de espalhamento com colisões solucionadas por listas lineares encadeadas é chamada de tabela de espalhamento com endereçamento exterior.

Cada compartimento de uma tabela de espalhamento com endereçamento aberto ou contém uma chave ou um valor que indica que o compartimento está vazio. Na busca por uma chave, examinamos sistematicamente os compartimentos da tabela até que a chave desejada seja encontrada ou até que fique claro que a chave não consta na tabela. Ao invés de seguir ponteiros, como em uma tabela de espalhamento com endereçamento exterior, devemos computar a seqüência de compartimentos a serem examinados. Para realizar uma inserção em uma tabela de espalhamento com endereçamento aberto, examinamos sucessivamente a tabela até que um compartimento vazio seja encontrado. A seqüência de compartimentos examinada depende da chave a ser inserida e não da ordem dos índices $0, 1, \ldots, m-1$.

Em uma tabela com endereçamento aberto, a função de espalhamento tem domínio e contra-domínio definidos como:

$$h\colon \mathscr{C} \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}\ .$$

Além disso, é necessário que para toda chave $x$, a **seqüência de tentativas** ou **seqüência de exames** dada por $\langle h(x,0), h(x,1), \ldots, h(x,m-1)\rangle$ seja uma permutação de $\langle 0, 1, \ldots, m-1\rangle$.

Inicialmente, a tabela de espalhamento com endereçamento aberto $T$ é inicializada com $-1$ em cada um de seus compartimentos, indicando que todos estão vazios. Veja a figura 23.5.

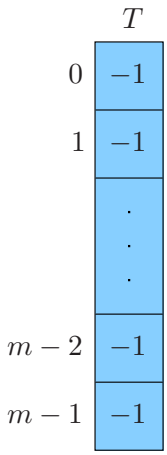



Figura 23.5: Uma tabela de espalhamento com endereçamento aberto vazia.

Na função `insere_aberto` a seguir, uma tabela de espalhamento $T$ com endereçamento aberto, contendo $m$ compartimentos, e uma chave $x$ são fornecidas como entrada. Consideramos que as chaves estão armazenadas diretamente na tabela de espalhamento $T$ e que um compartimento vazio de $T$ contém o valor $-1$. Após o processamento, a função devolve um número inteiro entre 0 e $m-1$ indicando que a inserção obteve sucesso ou o valor $m$ indicando o contrário.

```
/* Recebe uma tabela de espalhamento T de tamanho m  e uma chave x e
   insere a chave x na tabela T, devolvendo o índice da tabela onde a
   chave foi inserida, em caso de sucesso,  ou m em caso contrário */
int insere_aberto(int m, int T[MAX], int x)
{
   int i, j;

   i = 0;
   do {
      j = h(x, i);
      if (T[j] == -1) {
         T[j] = x;
         return j;
      }
      else
         i++;
   } while (i != m);

   return m;
}
```

A busca em uma tabela com endereçamento aberto é muito semelhante à inserção que vimos acima. A busca também termina se a chave foi encontrada na tabela ou quando encontra um compartimento vazio.

```
/* Recebe uma tabela de espalhamento T de tamanho m e uma chave
   x e busca a chave x na tabela T, devolvendo o índice da tabe-
   la onde a chave foi encontrada ou o valor m em caso contrário */
int busca_aberto(int m, int T[MAX], int x)
{
   int i, j;

   i = 0;
   do {
      j = h(x, i);
      if (T[j] == x)
         return j;
      else
         i++;
   } while (T[j] != -1 && i != m);

   return m;
}
```

A remoção de uma chave em um compartimento $i$ da tabela de espalhamento com endereçamento aberto $T$ não permite que o compartimento $i$ seja marcado com $-1$, já que isso tem implicação direta em seqüências de tentativas posteriores, afetando negativamente operações básicas subseqüentes, especialmente a inserção. Uma solução possível é fazer com que cada compartimento da tabela seja uma células com dois campos: uma chave e um estado. O estado de uma célula pode ser um dos três: **`VAZIA`**, **`OCUPADA`** ou **`REMOVIDA`**. Dessa forma, uma tabela de espalhamento com endereçamento aberto $T$ é um vetor do tipo **celula**, definido abaixo:

```
struct cel {
   int chave;
   int estado;
};

typedef struct cel celula;
```

Inicialmente, a tabela de espalhamento $T$ é inicializada com seus estados todos preenchidos com o **VAZIA**, como mostra a figura 23.6.

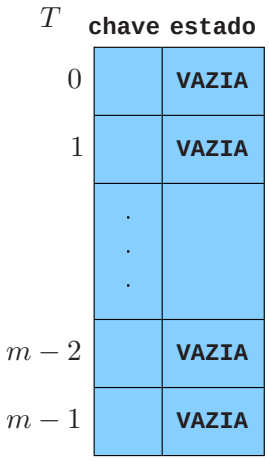



Figura 23.6: Uma tabela de espalhamento com endereçamento aberto vazia, onde cada compartimento contém uma chave e seu estado.

As três operações básicas sobre essa nova tabela de espalhamento com endereçamento aberto são implementadas nas funções a seguir.

```
/* Recebe uma tabela de espalhamento T de tamanho m e uma chave
   x e busca a chave x na tabela T, devolvendo o índice da tabe-
   la onde a chave foi encontrada ou o valor m em caso contrário */
int busca_aberto(int m, int T[MAX], int x)
{
   int i, j;

   i = 0;
   do {
      j = h(x, i);
      if (T[j].chave == x && T[j].estado == OCUPADA)
         return j;
      else
         i++;
   } while (T[j].estado != VAZIA && i != m);

   return m;
}
```

```
/* Recebe uma tabela de espalhamento T de tamanho m  e uma chave x e
   insere a chave x na tabela T, devolvendo o índice da tabela onde a
   chave foi inserida, em caso de sucesso,  ou m em caso contrário */
int insere_aberto(int m, int T[MAX], int x)
{
   int i, j;

   i = 0;
   do {
      j = h(x, i);
      if (T[j].estado != OCUPADA) {
         T[j].chave = x;
         T[j].estado = OCUPADA;
         return j;
      }
      else
         i++;
   } while (i != m);
   return m;
}
```

```
/* Recebe uma tabela de espalhamento T de tamanho m  e uma chave x e
   remove a chave x na tabela T, devolvendo o índice da tabela onde a
   chave foi inserida, em caso de sucesso,  ou m em caso contrário */
int remove_aberto(int m, int T[MAX], int x)
{
   int i, j;

   i = 0;
   do {
      j = h(x, i);
      if (T[j].estado != VAZIA) {
         if (T[j].chave == x && T[j].estado == OCUPADA) {
            T[j].estado = REMOVIDA;
            return j;
         }
         else
            i++;
      }
      else
         i = m;
   } while (i != m);
   return m;
}
```

A figura 23.7 mostra um exemplo de uma tabela de espalhamento $T$ com endereçamento aberto, alocada com 10 compartimentos e contendo 6 chaves: $41, 22, 104, 16, 37, 16$. A função de espalhamento usada $h$ é dada pelo método da tentativa linear, que veremos na seção 23.4.1 a seguir, e é definida da seguinte forma:

$$h(x, i) = \big(h'(x) + i\big) \bmod 10\,,$$

onde $h'$ é uma função de espalhamento ordinária dada pelo método da divisão apresentado na seção 23.3.1 e definida como:

$$h'(x) = x \bmod 10\,.$$

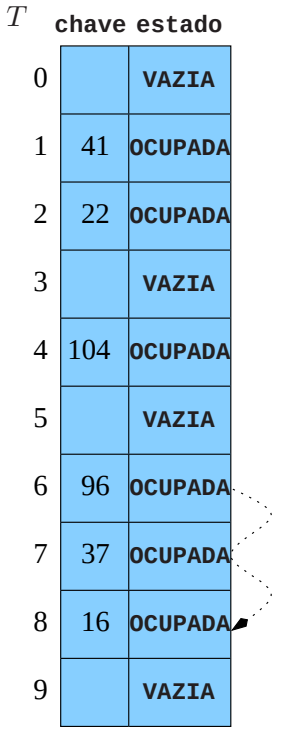



Figura 23.7: Um exemplo de uma tabela de espalhamento com endereçamento aberto. A busca da chave 16 usando a função de espalhamento $h$ segue a seqüência de tentativas ilustrada pela linha pontilhada.

O papel da função de espalhamento $h$ é gerar uma seqüência de compartimentos onde uma chave de interesse pode ocorrer. Observe que, para a função $h$ particular que vimos acima, se a remoção da chave 37 é realizada na tabela $T$, a busca subseqüente pela chave 16 ocorre corretamente.

Em uma tabela de espalhamento com endereçamento aberto temos sempre no máximo uma chave por compartimento. Isso significa que $n \leqslant m$, onde $n$ é o número de chaves armazenadas e $m$ o total de compartimentos da tabela. Portanto, o fator de carga $\alpha$ da tabela sempre satisfaz $\alpha \leqslant 1$. Se consideramos a hipótese que a tabela de espalhamento é uniforme, isto é, que a seqüência de tentativas $\langle h(x,0), h(x,1), \ldots, h(x,m-1)\rangle$ usada em uma operação básica é igualmente provável a qualquer permutação de $\langle 0, 1, \ldots, m-1\rangle$, então temos duas conseqüências importantes:

- o número esperado de tentativas em uma busca sem sucesso é no máximo

$$\frac{1}{1-\alpha}\,;$$

  isso significa, por exemplo, que se a tabela está preenchida pela metade, então o número médio de tentativas em uma busca sem sucesso é no máximo $1/(1-0.5) = 2$; se a tabela está 90% cheia, então o número médio de tentativas é no máximo $1/(1-0.9) = 10$;

- o número esperado de tentativas em uma busca com sucesso é no máximo

$$\left(\frac{1}{\alpha}\right) \ln\left(\frac{1}{1-\alpha}\right)\,;$$

isso significa, por exemplo, que se a tabela está preenchida pela metade, então o número médio de tentativas em uma busca sem sucesso é menor que $1.387$; se a tabela está 90% cheia, então o número médio de tentativas é menor que $2.559$.

### 23.4.1 Funções de espalhamento

Idealmente, supomos que cada chave em uma tabela de espalhamento com endereçamento aberto tenha qualquer uma das $m!$ permutações igualmente prováveis de $\langle 0, 1, \ldots, m-1\rangle$ como sua seqüência de tentativas. Essa suposição sobre a função de espalhamento é chamada de **espalhamento uniforme** e generaliza a suposição de espalhamento simples e uniforme que vimos acima. Funções de espalhamento uniforme são difíceis de implementar e na prática são usadas algumas boas aproximações.

As técnicas mais comuns para computar seqüências de tentativas em tabelas de espalhamento com endereçamento aberto são as seguintes: tentativa linear, tentativa quadrática e espalhamento duplo. Todas essas técnicas garantem que a seqüência de tentativas produzida $\langle h(x,0), h(x,1), \ldots, h(x,m-1)\rangle$ é uma permutação da seqüência $\langle 0, 1, \ldots, m-1\rangle$ para cada chave $x$. Nenhuma delas satisfaz as suposições de espalhamento uniforme, sendo que a técnica do espalhamento duplo é a que apresenta melhores resultados.

#### Tentativa linear

Dada uma função de espalhamento auxiliar $h' : \mathscr{C} \to \{0, 1, \ldots, m-1\}$, o **método da tentativa linear** é dado pela função

$$h(x,i) = \big(h'(x) + i\big) \bmod m\,,$$

para $i = 0, 1, \ldots, m-1$. Para uma chave $x$, o primeiro compartimento examinado é $T[h'(x)]$, que é o compartimento obtido pela função de espalhamento auxiliar. Em seguida, o compartimento $T[h'(x)+1]$ é examinado e assim por diante, até o compartimento $T[m-1]$. Depois disso, a seqüência de tentativas continua a partir de $T[0]$, seguindo para $T[1]$ e assim por diante, até $T[h'(x)-1]$.

É fácil de implementar funções de espalhamento usando o método da tentativa linear, mas tais funções sofrem de um problema conhecido como **agrupamento primário**, que provoca o aumento do tempo médio das operações básicas.

#### Tentativa quadrática

Uma função de espalhamento projetada pelo **método da tentativa quadrática** é ligeiramente diferente de uma função dada pelo método da busca linear, já que usa uma função da forma:

$$h(x,i) = \big(h'(x) + c_1 i + c_2 i^2\big) \bmod m\,,$$

onde $h'$ é uma função de espalhamento auxiliar, $c_1$ e $c_2$ são constantes auxiliares e $i = 0, 1, \ldots, m-1$. Para uma chave $x$, o primeiro compartimento examinado é $T[h'(x)]$. Em seguida, os compartimentos examinados são obtidos pelo deslocamento quadrático de valores que dependem de $i$. Este método é melhor que o anterior, mas a escolha dos valores $c_1, c_2$ e

$m$ é restrita. Além disso, o método sofre do problema de **agrupamento secundário**, já que se $h(x_1, 0) = h(x_2, 0)$ então $h(x_1, i) = h(x_2, i)$ para todo $i$.

#### Espalhamento duplo

O **método do espalhamento duplo** usa uma função de espalhamento da seguinte forma:

$$h(x, i) = \big(h_1(x) + ih_2(x)\big) \bmod m\,,$$

onde $h_1$ e $h_2$ são funções de espalhamento auxiliares. O compartimento inicialmente examinado é $T[h_1(x)]$. Os compartimentos examinados em seguida são obtidos do deslocamento dos compartimentos anteriores da quantidade de $h_2(x)$ módulo $m$.

A recomendação é que o valor $h_2(x)$ seja um primo relativo ao tamanho da tabela $m$. Podemos assegurar essa condição fazendo $m$ uma potência de 2 e projetando $h_2$ de tal forma que sempre produza um número ímpar. Outra forma é fazer $m$ primo e projetar $h_2$ de tal forma que sempre devolva um número inteiro positivo menor que $m$.

O método do espalhamento duplo é um dos melhores métodos disponíveis para o endereçamento aberto já que as permutações produzidas têm muitas das características de permutações aleatórias. O desempenho do método do espalhamento duplo é, assim, próximo ao desempenho do esquema ideal de espalhamento universal.

## Exercícios

23.1 Suponha que temos um conjunto de chaves $C$ armazenado em uma tabela de acesso direto $T$ de tamanho $m$. Escreva uma função que encontra a maior chave de $C$. Qual o tempo de execução de pior caso da sua função?

23.2 Suponha um conjunto de $n$ chaves formado pelos $n$ primeiros múltiplos de 7. Quantas colisões ocorrem mediante a aplicação das funções de espalhamento abaixo?

(a) $x \bmod 7$

(b) $x \bmod 14$

(c) $x \bmod 5$

23.3 Ilustre a inserção das chaves $5, 28, 19, 15, 20, 33, 12, 17, 10$ em uma tabela de espalhamento com colisões resolvidas por listas lineares encadeadas. Suponha que a tabela tenha 9 compartimentos e que a função de espalhamento seja $h(x) = x \bmod 9$.

23.4 Considere uma tabela de espalhamento de tamanho $m = 1000$ e uma função de espalhamento $h(x) = \lfloor m\,(Ax - \lfloor Ax \rfloor)\rfloor$ para $A = (\sqrt{5} - 1)/2$. Compute as posições para as quais as chaves $61, 62, 63, 64$ e $65$ são mapeadas.

23.5 Considere a inserção das chaves $10, 22, 31, 4, 15, 28, 17, 88, 59$ em uma tabela de espalhamento com endereçamento aberto de tamanho $m = 11$ com função de espalhamento auxiliar $h'(x) = x \bmod m$. Ilustre o resultado da inserção dessas chaves usando tentativa linear, tentativa quadrática com $c_1 = 1$ e $c_2 = 3$ e espalhamento duplo com $h_2(x) = 1 + (x \bmod (m - 1))$.

23.6 A tabela abaixo é composta das palavras-chaves da linguagem C padrão.

| | | | |
|---|---|---|---|
| `auto` | `double` | `int` | `long` |
| `break` | `else` | `long` | `switch` |
| `case` | `enum` | `register` | `typedef` |
| `char` | `extern` | `return` | `union` |
| `const` | `float` | `short` | `unsigned` |
| `continue` | `for` | `signed` | `void` |
| `default` | `goto` | `sizeof` | `volatile` |
| `do` | `if` | `static` | `while` |

Escreva um programa que leia um arquivo contendo um programa na linguagem C e identifique suas palavras-chaves.

23.7 Veja animações do funcionamento de tabelas de espalhamento nas páginas a seguir:

- Hashing Animation Tool, Catalyst Software, 2000
- Hash Table Animation, Woi Ang
- Hashing, Hang Thi Anh Pham, 2001

AULA 24

# OPERAÇÕES SOBRE BITS

A linguagem C possui muitas características de uma linguagem de programação de alto nível e isso significa que podemos escrever programas que são independentes das máquinas que irão executá-los. A maioria das aplicações possui naturalmente essa propriedade, isto é, podemos propor soluções computacionais – programas escritos na linguagem C – que não têm de se adaptar às máquinas em que serão executados. Por outro lado, alguns programas mais específicos, como compiladores, sistemas operacionais, cifradores, processadores gráficos e programas cujo tempo de execução e/ou uso do espaço de armazenamento são críticos, precisam executar operações no nível dos bits. Outra característica importante da linguagem C, que a diferencia de outras linguagens de programação alto nível, é que ela permite o uso de operações sobre bits específicos e sobre trechos de bits, e ainda permite o uso de estruturas de armazenamento para auxiliar nas operações de baixo nível.

Nesta aula, baseada na referência [7], tratamos com algum detalhe de cada um desses aspectos da linguagem C.

## 24.1 Operadores bit a bit

A linguagem C possui seis **operadores bit a bit**, que operam sobre dados do tipo inteiro e do tipo caractere no nível de seus bits. A tabela abaixo mostra esses operadores e seus significados.

| Operador | Significado |
|---|---|
| **~** | complemento |
| **&** | E |
| **^** | OU exclusivo |
| **\|** | OU inclusivo |
| **<<** | deslocamento à esquerda |
| **>>** | deslocamento à direita |

O operador de complemento **~** é o único operador unário dessa tabela, os restantes todos são binários. Os operadores **~**, **&**, **^** e **|** executam operações booleanas sobre os bits de seus operandos.

O operador **~** gera o complemento de seu operando, isto é, troca os bits 0 por 1 e os bits 1 por 0. O operador **&** executa um E booleano bit a bit nos seus operandos. Os operadores **^** e **|** executam um OU booleano bit a bit nos seus operandos, sendo que o operador **^** produz 0 caso os dois bits correspondentes dos operandos sejam 1 e, nesse mesmo caso, o operador **|** produz

1. Os operadores de deslocamento **<<** e **>>** são operadores binários que deslocam os bits de seu operando à esquerda um número de vezes representado pelo operando à direita. Por exemplo, `i << j` produz como resultado o valor de `i` deslocado de `j` posições à esquerda. Para cada bit que é deslocado à esquerda "para fora" do número, um bit 0 é adicionado à direita desse número. O mesmo vale inversamente para o operador de deslocamento à direita **>>**.

O programa 24.1 mostra um exemplo simples do uso de todos os operadores sobre bits.

Programa 24.1: Uso de operadores sobre bits.

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

/* Recebe um número inteiro n e devolve uma cadeia de carac-
   teres representando sua representação na base binária */
char *binario(unsigned short int n)
{
   char *b;
   int i;

   b = (char *) malloc(17 * sizeof (char));
   b[16] = '\0';
   for (i = 15; i >= 0; i--, n = n / 2)
      b[i] = '0' + n % 2;

   return b;
}

int main(void)
{
   unsigned short int i, j, k;

   i = 51;
   printf("     i = %5d (%s)\n", i, binario(i));
   j = 15;
   printf("     j = %5d (%s)\n\n", j, binario(j));

   k = ~i;
   printf("    ~i = %5d (%s)\n", k, binario(k));
   k = i & j;
   printf(" i & j = %5d (%s)\n", k, binario(k));
   k = i ^ j;
   printf(" i ^ j = %5d (%s)\n", k, binario(k));
   k = i | j;
   printf(" i | j = %5d (%s)\n", k, binario(k));
   k = i << 4;
   printf("i << 4 = %5d (%s)\n", k, binario(k));
   k = j >> 2;
   printf("j >> 2 = %5d (%s)\n", k, binario(k));

   return 0;
}
```

A execução do programa 24.1 provoca a seguinte saída:

```
     i =    51 (0000000000110011)
     j =    15 (0000000000001111)

    ~i = 65484 (1111111111001100)
 i & j =     3 (0000000000000011)
 i ^ j =    60 (0000000000111100)
 i | j =    63 (0000000000111111)
i << 4 =   816 (0000001100110000)
j >> 2 =     3 (0000000000000011)
```

Os operadores sobre bits têm precedências distintas uns sobre os outros, como mostra a tabela abaixo:

| Operador | Precedência |
|---|---|
| ~ | 1 (maior) |
| & << >> | 2 |
| ^ | 3 |
| \| | 4 (menor) |

Observe ainda que a precedência dos operadores sobre bits é menor que precedência dos operadores relacionais. Isso significa que um sentença como a seguir

```
if (status & 0x4000 != 0)
```

tem o significado a seguir

```
if (status & (0x4000 != 0))
```

o que muito provavelmente não é a intenção do(a) programador(a) neste caso. Dessa forma, para que a intenção se concretize, devemos escrever:

```
if ((status & 0x4000) != 0)
```

Muitas vezes em programação de baixo nível, queremos acessar bits específicos de uma seqüência de bits. Por exemplo, quando trabalhamos com computação gráfica, queremos colocar dois ou mais pixeis em um único byte. Podemos extrair e/ou modificar dados que são armazenados em um número pequeno de bits com o uso dos operadores sobre bits.

Suponha que `i` é uma variável do tipo `unsigned short int`, ou seja, um compartimento de memória de 16 bits que armazena um número inteiro. As operações mais freqüentes sobre um único bit que podem ser realizadas sobre a variável `i` são as seguintes:

- **Ligar um bit:** suponha que queremos ligar o bit 4 de `i`, isto é o quinto bit menos significativo, o quinto da direita para a esquerda. A forma mais fácil de ligar o bit 4 é executar um OU inclusivo entre `i` e a constante `0x0010`:

```
i = i | 0x0010;
```

Mais geralmente, se a posição do bit que queremos ligar está armazenada na variável `j`, podemos usar o operador de deslocamento à esquerda e executar a seguinte expressão, seguida da atribuição:

```
i = i | 1 << j;
```

- **Desligar um bit:** para desligar o bit 4 de `i`, devemos usar uma máscara com um bit 0 na posição 4 e o bit 1 em todas as outras posições:

```
i = 0x00ff; i = i & ~0x0010;
```

Usando a mesma idéia, podemos escrever uma sentença que desliga um bit cuja posição está armazenada em uma variável `j`:

```
i = i & ~(1 << j);
```

- **Testando um bit:** o trecho de código a seguir verifica se o bit 4 da variável `i` está ligado:

```
if (i & 0x0010)
```

Para testar se o bit `j` está ligado, temos de usar a seguinte sentença:

```
if (i & 1 << j)
```

Para que o trabalho com bits torne-se um pouco mais fácil, freqüentemente damos nomes às posições dos bits de interesse. Por exemplo, se os bits das posições 1, 2 e 4 correspondem às cores azul, verde e vermelho, podemos definir macros que representam essas três posições dos bits:

```
#define AZUL     1
#define VERDE    2
#define VERMELHO 4
```

Ligar, desligar e testar o bit `AZUL` pode ser feito como abaixo:

```
i = i | AZUL;
i = i & ~AZUL;
if (i & AZUL) ...
```

Podemos ainda ligar, desligar e testar vários bits ao mesmo tempo:

```
i = i | AZUL | VERDE;
i = i & ~(AZUL | VERMELHO);
if (i & (VERDE | VERMELHO)) ...
```

Trabalhar com um grupo de vários bits consecutivos, chamados de **trecho de bits**[1] é um pouco mais complicado que trabalhar com um único bit. As duas operações mais comuns sobre trechos de bits são as seguintes:

- **Modificar um trecho de bits:** para alterar um trecho de bits é necessário inicialmente limpá-lo usando a operação E bit a bit e posteriormente armazenar os novos bits no trecho de bits com a operação de OU inclusivo. A operação a seguir mostra como podemos armazenar o valor binário 101 nos bits 4–6 da variável `i`:

  ```
  i = i & ~0x0070 | 0x0050;
  ```

  Suponha ainda que a variável `j` contém o valor a ser armazenado nos bits 4–6 de `i`. Então, podemos executar a seguinte sentença:

  ```
  i = i & ~0x0070 | j « 4;
  ```

- **Recuperar um trecho de bits:** quando o trecho de bits atinge o bit 0 da variável, a recuperação de seu valor pode ser facilmente obtida da seguinte forma. Suponha que queremos recuperar os bits 0–2 da variável `i`. Então, podemos fazer:

  ```
  j = i & 0x0007;
  ```

  Se o trecho de bits não tem essa propriedade, então podemos deslocar o trecho de bits à direita para depois extraí-lo usando a operação E. Para extrair, por exemplo, os bits 4–6 de `i`, podemos usar a seguinte sentença:

  ```
  j = (i >> 4) & 0x0007;
  ```

[1] Tradução livre de *bit-field*.

## 24.2 Trechos de bits em registros

Além das técnicas que vimos na seção anterior para trabalhar com trechos de bits, a linguagem C fornece como alternativa a declaração de registros cujos campos são trechos de bits. Essa característica pode ser bem útil especialmente em casos em que a complexidade de programação envolvendo trechos de bits gera arquivos-fontes complicados e confusos.

Suponha que queremos armazenar uma data na memória. Como os números envolvidos são relativamente pequenos, podemos pensar em armazená-la em 16 bits, com 5 bits para o dia, 4 bits para o mês e 7 bits para o ano, como mostra a figura 24.1.

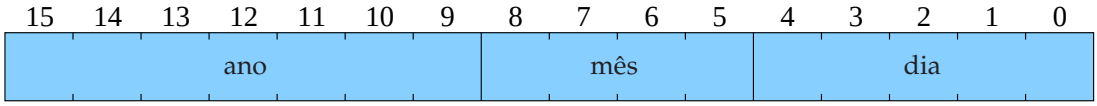



Figura 24.1: A forma de armazenamento de uma data.

Usando trechos de bits, podemos definir um registro com o mesmo formato:

```
struct data {
   unisgned int dia: 5;
   unisgned int mes: 4;
   unisgned int ano: 7;
};
```

O número após cada campo indica o comprimento em bits desse campo. O tipo de um trecho de bits pode ser um de dois possíveis: `int` (`signed int`) e `unsigned int`. O melhor é declarar todos os trechos de bits que são campos de um registro como `signed int` ou `unsigned int`.

Podemos usar os trechos de bits exatamente como usamos outros campos de um registro, como mostra o exemplo a seguir:

```
   struct data data_arquivo;

   data_arquivo.dia = 28;
   data_arquivo.mes = 12;
   data_arquivo.ano = 8;
```

Podemos considerar que um valor armazenado no campo `ano` seja relativo ao ano de 1980[2]. Ou seja a atribuição do valor `8` ao campo `ano`, como feita acima, tem significado 1988. Depois dessas atribuições, a variável `data_arquivo` tem a aparência mostrada na figura 24.2.

Poderíamos usar operadores sobre bits para obter o mesmo resultado, talvez até mais rapidamente. No entanto, escrever um programa legível é usualmente mais importante que ganhar alguns microssegundos.

[2] 1980 é o ano em que o mundo começou, segundo a Micro$oft. O registro `data` é a forma como o sistema operacional MS–DOS armazena a data que um arquivo foi criado ou modificado.

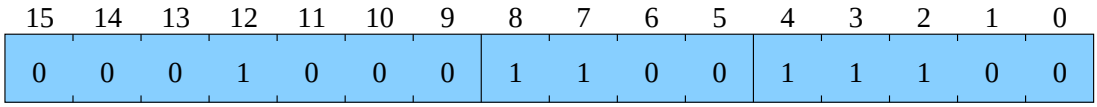



Figura 24.2: Variável `data_arquivo` com valores armazenados.

Trechos de bits têm uma restrição que não se aplica a outros campos de um registro. Como os trechos de bits não têm um endereço no sentido usual, a linguagem C não nos permite aplicar o operador de endereço `&` para um trecho de bits. Por conta disso, funções como `scanf` não podem armazenar valores diretamente em um campo de um registro que é um trecho de bits. É claro que podemos usar `scanf` para ler uma entrada em uma variável e então atribui seu valor a um campo de um registro que é um trecho de bits.

## Exercícios

24.1 Uma das formas mais simples de criptografar dados é usar a operação de OU exclusivo sobre cada caractere com uma chave secreta. Suponha, por exemplo, que a chave secreta é o caractere **&**. Supondo que usamos a tabela ASCII, se fizermos um OU exclusivo do caractere **z** com a chave, obtemos o caractere \:

| | | |
|---|---|---|
| | 00100110 | código binário ASCII para **&** |
| XOR | 01111010 | código binário ASCII para **z** |
| | 01011100 | código binário ASCII para \ |

Para decifrar uma mensagem, aplicamos a mesma idéia. Cifrando uma mensagem previamente cifrada, obtemos a mensagem original. Se executamos um OU exclusivo entre o caractere **&** e o caractere \, por exemplo, obtemos o caractere original **z**:

| | | |
|---|---|---|
| | 00100110 | código binário ASCII para **&** |
| XOR | 01011100 | código binário ASCII para \ |
| | 01111010 | código binário ASCII para **z** |

Escreva um programa que receba uma cadeia de caracteres e cifre essa mensagem usando a técnica previamente descrita.

A mensagem original pode ser digitada pelo(a) usuário ou lida a partir de um arquivo com redirecionamento de entrada. A mensagem cifrada pode ser vista na saída padrão (monitor) ou pode ser armazenada em um arquivo usando redirecionamento de saída.

AULA 25

# UNIÕES E ENUMERAÇÕES

Complementando o conteúdo apresentado até aqui sobre variáveis compostas, a linguagem C ainda oferece dois tipos delas: as uniões e as enumerações. Uma união é um registro que armazena apenas um dos valores de seus campos e, portanto, apresenta economia de espaço. Uma enumeração ajuda o(a) programador(a) a declarar e usar uma variável que representa intuitivamente uma seqüência de números naturais. Nesta aula, baseada especialmente na referência [7], veremos uma breve introdução sobre uniões e enumerações.

## 25.1 Uniões

Como os registros, as uniões também são compostas por um ou mais campos, definidos possivelmente por tipos diferentes de dados. No entanto, o compilador aloca espaço suficiente apenas para o maior dos campos de uma união, em termos de quantidade de bytes alocados. Esses campos compartilham então o mesmo espaço na memória. Dessa forma, a atribuição de um valor a um dos campos da união também altera os valores dos outros campos nessa mesma estrutura.

Vamos declarar a união `u` com dois campos, um do tipo caractere com sinal e outro do tipo inteiro com sinal, como abaixo:

```
union {
   char c;
   int i;
} u;
```

Observe também que a declaração de uma união é basicamente idêntica à declaração de um registro, a menos da palavra reservada `union` que substitui a palavra reservada `struct`. Por exemplo, podemos declarar o registro `r` com os mesmos dois campos da união `u` como a seguir:

```
struct {
   char c;
   int i;
} r;
```

A diferença entre a união `u` e o registro `r` concentra-se apenas no fato que os campos de `u` estão localizados no mesmo endereço de memória enquanto que os campos de `r` estão localizados em endereços diferentes. Considerando que um caractere ocupa um byte na memória e um número inteiro ocupa quatro bytes em algumas implementações vigentes de computadores, a figura 25.1 apresenta uma ilustração do que ocorre na memória na declaração das variáveis `u` e `r`.

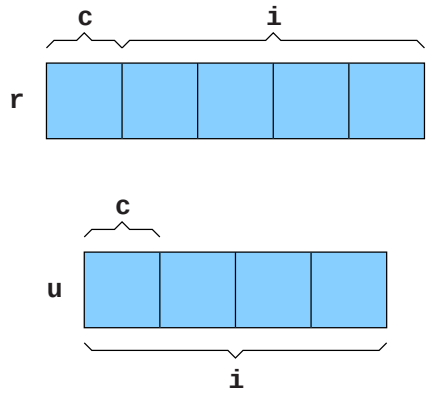



Figura 25.1: União e registro representados na memória.

No registro `r`, os campos `c` e `i` ocupam diferentes posições da memória. Assim, o total de memória necessário para armazenamento do registro `r` na memória é de cinco bytes. Na união `u`, os campos `c` e `i` compartilham a mesma posição da memória – isto é, os campos `c` e `i` de `u` têm o mesmo endereço de memória – e, portanto, o total de memória necessário para armazenamento de `u` é de quatro bytes.

Campos de uma união são acessados da mesma forma que os campos de um registro. Por exemplo, podemos fazer a atribuição a seguir:

```
u.c = 'a';
```

Também podemos fazer a atribuição de um número inteiro para o outro campo da união como apresentado abaixo:

```
u.i = 3894;
```

Como o compilador superpõe valores nos campos de uma união, é importante notar que a alteração de um dos campos implica na alteração de qualquer valor previamente armazenado nos outros campos. Portanto, se armazenamos, por exemplo, um valor no campo `u.c`, qualquer valor previamente armazenado no campo `u.i` será perdido. Do mesmo modo, alterar o valor do campo `u.i` altera o valor do campo `u.c`. Assim, podemos pensar na união `u` como um local na memória para armazenar um caractere no campo `u.c` *ou* um número no campo `u.i`, mas não ambos. Diferentemente, o registro `r` pode armazenar valores no campo `r.c` *e* no campo `r.i`.

Como o que ocorre com registros, uniões podem ser copiadas diretamente usando o comando de atribuição `=`, sem a necessidade de fazê-las campo a campo. Declarações e inicializações simultâneas também são feitas similarmente. No entanto, observe que somente o primeiro campo de uma união pode ter um valor atribuído no inicializador. Por exemplo, podemos fazer:

```
union {
   char c;
   int i;
} u = '\0';
```

Usamos uniões freqüentemente como uma forma de economizar espaço alocado desnecessariamente em registros. Ademais, usamos uniões quando queremos criar estruturas de armazenamento de dados que contenham uma mistura de diferentes tipos de dados, yambém visando economia de espaço alocado em memória.

Como um exemplo do segundo caso acima mencionado, suponha que temos uma coleção de números inteiros e de números de ponto flutuante que queremos armazenar em um único vetor. Como os elementos de um vetor têm de ser de um mesmo tipo, devemos usar uniões para implementar um vetor com essa característica. A declaração de um vetor como esse é dada a seguir:

```
union {
   int i;
   double d;
} vetor[100];
```

Cada compartimento do `vetor` acima declarado pode armazenar um valor do tipo `int` ou um valor do tipo `double`, o que possibilita armazenar uma mistura de valores do tipo `int` e `double` nos diferentes compartimentos do `vetor`. Por exemplo, para as atribuições abaixo atribuem um número inteiro e um número de ponto flutuante para as posições 0 e 1 do `vetor`, respectivamente:

```
   vetor[0].i = 5;
   vetor[1].d = 7.647;
```

Suponha agora que queremos resolver o seguinte problema: dada uma seqüência $n$ de números, de qualquer dos tipos inteiro ou real, com $1 \leqslant n \leqslant 100$, computar a adição de todos os números inteiros e o produto de todos os números reais, mostrando os respectivos resultados na saída padrão.

O programa 25.1 mostra um exemplo do uso de um vetor de registros onde um de seus campos é uma união.

Programa 25.1: Um exemplo de uso de registros e uniões.

```
#include <stdio.h>

#define TIPO_INT   0
#define TIPO_DOUBLE 1
#define MAX      100

typedef struct {
   int tipo;
   union {
      int i;
      double d;
   } u;
} mix_num;

/* Recebe uma seqüência de n numeros, inteiros e reais, e mos-
   tra a soma dos números inteiros e a soma dos números reais */
int main(void)
{
   int i, n, soma, inteiro, real;
   double produto;
   mix_num numero[MAX];

   printf("Informe n: ");
   scanf("%d", &n);
   for (i = 0; i < n; i++) {
      printf("Informe o tipo do número (0: inteiro, 1: real): ");
      scanf("%d", &numero[i].tipo);
      printf("Informe o número: ");
      if (numero[i].tipo == TIPO_INT)
         scanf("%d", &numero[i].u.i);
      else
         scanf("%lf", &numero[i].u.d);
   }

   soma = 0;
   produto = 1.0;
   inteiro = 0;
   real = 0;
   for (i = 0; i < n; i++)
      if (numero[i].tipo == TIPO_INT) {
         soma = soma + numero[i].u.i;
         inteiro = 1;
      }
      else {
         produto = produto * numero[i].u.d;
         real = 1;
      }

   if (inteiro)
      printf("Soma dos números inteiros: %d\n", soma);
   else
      printf("Não foram informados números inteiros na entrada\n");
   if (real)
      printf("Produto dos números reais: %g\n", produto);
   else
      printf("Não há números reais na entrada\n");
   return 0;
}
```

## 25.2 Enumerações

Muitas vezes precisamos de variáveis que conterão, durante a execução de um programa, somente um pequeno conjunto de valores. Por exemplo, variáveis lógicas ou booleanas deverão conter somente dois valores possíveis: "verdadeiro" e "falso". Uma variável que armazena os naipes das cartas de um baralho deverá conter apenas quatro valores potenciais: "paus", "copas", "espadas" e "ouros". Uma maneira natural de tratar esses valores é através da declaração de uma variável do tipo inteiro que pode armazenar valores que representam esses naipes. Por exemplo, 0 representa o naipe "paus", 1 o naipe "copas" e assim por diante. Assim, podemos por exemplo declarar uma variável naipe e atribuir valores a essa variável da seguinte forma:

```
int n;

n = 0;
```

Apesar de essa técnica funcionar, alguém que precisa compreender um programa que contém esse trecho de código é potencialmente incapaz de saber que a variável `n` pode assumir apenas quatro valores durante a execução do programa e, ademais, o significado do valor 0 não é imediatamente aparente.

Uma estratégia melhor que a anterior é o uso de macros para definir um "tipo" naipe e também para definir nomes para os diversos naipes existentes. Então, podemos fazer:

```
#define NAIPE   int
#define PAUS    0
#define COPAS   1
#define ESPADAS 2
#define OUROS   3
```

O exemplo anterior fica então bem mais fácil de ler:

```
NAIPE n;

n = PAUS;
```

Apesar de melhor, essa estratégia ainda possui restrições. Por exemplo, alguém que necessita compreender o programa não tem a visão imediata que as macros representam valores do mesmo "tipo". Além disso, se o número de valores possíveis é um pouco maior, a definição de uma macro para cada valor pode se tornar uma tarefa tediosa. Por fim, os identificadores das macros serão removidos pelo pré-processador e substituídos pelos seus valores respectivos e isso significa que não estarão disponíveis para a fase de depuração do programa. Depuração é uma atividade muito útil para programação a medida que nossos programas ficam maiores e mais complexos.

A linguagem C possui um tipo especial específico para variáveis que armazenam um conjunto pequeno de possíveis valores. Um **tipo enumerado** é um tipo cujos valores são listados ou enumerados pelo(a) programador(a) que deve criar um nome ou identificador, chamado de uma **constante da enumeração**, para cada um dos valores possíveis. O exemplo a seguir enumera os valores `PAUS`, `COPAS`, `ESPADAS` e `OUROS` que podem ser atribuídos às variáveis `n1` e `n2`:

```
enum PAUS, COPAS, ESPADAS, OUROS n1, n2;

n1 = PAUS;
n2 = n1;
```

Apesar de terem pouco em comum com registros e uniões, as enumerações são declaradas de modo semelhante. Entretanto, diferentemente dos campos dos registros e uniões, os nomes das constantes da enumeração devem ser diferentes de outros nomes de outras enumerações declaradas.

A linguagem C trata variáveis que são enumerações e constantes da enumeração como números inteiros. Por padrão, o compilador atribui os inteiros $0, 1, 2, \ldots$ para as constantes da enumeração na declaração de uma variável qualquer do tipo enumeração. No exemplo acima, `PAUS`, `COPAS`, `ESPADAS` e `OUROS` representam os valores 0, 1, 2 e 3, respectivamente.

Podemos escolher valores diferentes dos acima para as constantes da enumeração. Por exemplo, podemos fazer a seguinte declaração:

```
enum PAUS = 1, COPAS = 2, ESPADAS = 3, OUROS = 4 n;
```

Os valores das constantes da enumeração podem ser valores inteiros arbitrários, listados sem um ordem particular, como por exemplo:

```
enum FACOM = 17, CCET = 19, DEL = 11, DHT = 2, DEC = 21, DMT = 6 unidade;
```

Se nenhum valor é especificado para uma constante da enumeração, o valor atribuído à constante é um valor maior que o valor da constante imediatamente anterior. A primeira constante da enumeração tem o valor padrão 0 (zero). Na enumeração a seguir, a constante `PRETO` tem o valor 0, `CINZA_CLARO` tem o valor 7, `CINZA_ESCURO` tem o valor 8 e `BRANCO` tem o valor 15:

```
enum PRETO, CINZA_CLARO = 7, CINZA_ESCURO, BRANCO = 15 coresEGA;
```

Como as constantes de uma enumeração são, na verdade, números inteiros, a linguagem C permite que um(a) programador(a) use-as em expressões com inteiros. Por exemplo, o trecho de código a seguir é válido em um programa:

```
int i;
enum PAUS, COPAS, ESPADAS, OUROS n;

i = COPAS;
n = 0;
n++;
i = n + 2;
```

Apesar da conveniência de podermos usar uma constante de uma enumeração como um número inteiro, é perigoso usar um número inteiro como um valor de uma enumeração. Por exemplo, podemos acidentalmente armazenar o número 4, que não corresponde a qualquer naipe, na variável `n`.

Vejamos o programa 25.2 com um exemplo do uso de enumerações, onde contamos o número de cartas de cada naipe de um conjunto de cartas fornecido como entrada. Uma entrada é uma cadeia de caracteres, dada como "4pAp2c1o7e", onde "p" significa o naipe de paus, "c" copas, "e" espadas e "o" ouros.

## Exercícios

25.1 Escreva um programa que receba uma coleção de $n$ números, com $1 \leqslant n \leqslant 100$, que podem ser inteiros pequenos de 1 (um) byte, inteiros maiores de 4 (quatro) bytes ou números reais, e receba ainda um número $x$ e verifique se $x$ pertence a esse conjunto. Use uma forma de armazenamento que minimize a quantidade de memória utilizada. Use um vetor de registros tal que cada célula contém um campo que é uma enumeração, para indicar o tipo do número armazenado nessa célula, e um campo que é uma união, para armazenar um número.

25.2 Dados os números inteiros $m$ e $n$ e uma matriz $A$ de números inteiros, com $1 \leqslant m, n \leqslant 100$, calcular a quantidade de linhas que contêm o elemento 0 (zero) e a quantidade de colunas. Use uma variável indicadora de passagem, lógica, declarada como uma enumeração.

Programa 25.2: Uso de enumerações.

```
#include <stdio.h>

typedef struct {
   char valor[2];
   enum PAUS, COPAS, ESPADAS, OUROS naipe;
} carta;

/* Recebe uma cadeia de caracteres representando uma mão de
   cartas e conta o número de cartas de cada naipe nessa mão */
int main(void)
{
   char entrada[109];
   int i, j, conta[4] = 0;
   carta mao[52];

   printf("Informe uma mão: ");
   scanf("%s", entrada);
   for (i = 0, j = 0; entrada[i]; j++) {
      mao[j].valor[0] = entrada[i];
      i++;
      if (entrada[i] == '0') {
         mao[j].valor[1] = entrada[i];
         i++;
      }
      switch (entrada[i]) {
      case 'p':
         mao[j].naipe = PAUS;
         break;
      case 'c':
         mao[j].naipe = COPAS;
         break;
      case 'e':
         mao[j].naipe = ESPADAS;
         break;
      case 'o':
         mao[j].naipe = OUROS;
         break;
      default:
         break;
      }
      i++;
   }

   for (j--; j >= 0; j--)
      conta[mao[j].naipe]++;

   printf("Cartas:\n");
   printf(" %d de paus\n", conta[PAUS]);
   printf(" %d de copas\n", conta[COPAS]);
   printf(" %d de espadas\n", conta[ESPADAS]);
   printf(" %d de ouros\n", conta[OUROS]);

   return 0;
}
```

# REFERÊNCIAS BIBLIOGRÁFICAS


[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms. The MIT Press, 3rd edition, 2009. 2, 2.4, 3, 6, 7, 8, 8.1.2, 8.3, 23

[2] P. Feofiloff. Algoritmos em linguagem C. Editora Campus/Elsevier, 2009. 1, 1.1, 1.2, 2, 3, 3.2.2, 4, 4.1, 4.2, 5, 6, 7, 7.1, 8, 9, 12, 15, 16, 18, 19, 20

[3] Debugging with GDB: the GNU source-level debugger for GDB. http://sourceware.org/gdb/current/onlinedocs/gdb/.

[4] E. Huss. The C Library Reference Guide. http://www.acm.uiuc.edu/webmonkeys/book/c_guide/, 1997. último acesso em agosto de 2010.

[5] B. W. Kernighan and R. Pike. The Practice of Programming. Addison-Wesley Professional, 1999.

[6] B. W. Kernighan and D. M. Ritchie. C Programming Language. Prentice Hall, 2nd edition, 1988.

[7] K. N. King. C Programming – A Modern Approach. W. W. Norton & Company, Inc., 2nd edition, 2008. 1, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 21, 22, 24, 25

[8] J. Kleinberg and Éva Tardos. Algorithm Design. Pearson Education, Inc., 2006. 2

[9] S. G. Kochan. Unix Shell Programming. Sams Publishing, 3rd edition, 2003.

[10] Departamento de Ciência da Computação – IME/USP, listas de exercícios – Introdução à Computação. http://www.ime.usp.br/~macmulti/. último acesso em agosto de 2010.

[11] M. F. Siqueira. Algoritmos e Estruturas de Dados I. Notas de aula, 1998. (ex-professor do Departamento de Computação e Estatística da Universidade Federal de Mato Grosso do Sul (DCT/UFMS), atualmente professor do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte (DIMAp/UFRN)).

[12] S. S. Skiena and M. Revilla. Programming Challenges. Springer, 2003.

[13] J. L. Szwarcfiter and L. Markenzon. Estruturas de Dados e seus Algoritmos. Livros Técnicos e Científicos – LTC, 3 edition, 2010. 8, 18, 19, 20, 21, 22, 23

[14] Wikipedia – the Free Encyclopedia. http://en.wikipedia.org/wiki/Main_Page. último acesso em agosto de 2010.