Estruturas de Dados — Notas de Aula (versão 0.9)

Estruturas de Dados — Notas de Aula

Flávio Velloso Laper

1 de fevereiro de 2005

Universidade Fumec / Face

Sumário

Introdução

Conceitos

- Um *algoritmo* é um padrão de comportamento expresso em termos de um conjunto finito de ações. Exemplo: para somar $a+b$, executamos o mesmo processo independentemente dos valores de a e b.
- *Estruturas de dados* são a forma de representação escolhida para a resolução de um determinado problema.
- *Programar* é estruturar dados e construir algoritmos. Um programa é uma formulação concreta de um algoritmo abstrato, baseada em uma estrutura específica de dados.
- Um *tipo de dados* caracteriza os valores que podem ser assumidos por uma variável, constante ou expressão, e as operações que podem ser efetuadas sobre elas.
- Tipos de dados *simples* são grupos de valores indivisíveis. Exemplo: inteiros, caracteres, lógicos, etc.
- Tipos de dados *estruturados* são coleções ou agregados de tipos diversos (simples e/ou estruturados).
- Um tipo de dados *primitivo* (simples ou estruturado) está diretamente disponível na linguagem de programação utilizada.
- Um tipo de dados *abstrato* é um modelo (matemático) que define um conjunto de dados e as operações que podem ser executadas sobre os mesmos. Exemplo: listas, árvores, etc.
- A implementação de um algoritmo exige que se encontre uma forma de representar o modelo (abstrato) através dos tipos primitivos disponibilizados pela linguagem.

Materiais

Bibliografia do curso:

1. CELES, W., CERQUEIRA, L., RANGEL, J.L. *Introdução a Estruturas de Dados*; Com técnicas de programação em C. Rio de Janeiro: Elsevier, 2004. 294p.
2. ZIVIANI, N. *Projeto de Algoritmos*; Com implementações em Pascal e C. 2.ed. São Paulo: Thomson, 2004. 552p.

3. KERNIGHAM, B., Ritchie, D. *C*; A Linguagem de Programação Padrão ANSI. Rio de Janeiro: Campus, 1990. 289p.

Ferramentas:

1. Linux: compilador *gcc* (http://gcc.gnu.org).
2. Windows: ambiente de desenvolvimento Dev-C++ (http://www.bloodshed.net/devcpp.html).

Procedimentos para criação de um projeto no Dev-C++

Uma boa prática de trabalho é criar um novo projeto para cada programa a desenvolver. Os passos a seguir mostram como fazer isso no ambiente de trabalho Dev-C++:

1. Iniciar o programa.
2. Selecionar Arquivo ▷ Novo ▷ Projeto (ou utilizar o atalho na barra de ferramentas).
3. Criar o novo projeto. Informar:
   - Console Application.
   - Projeto C (atenção: não trabalhe com a linguagem C++!).
   - Nome do projeto (escolha um nome significativo).
4. Salvar o projeto em uma pasta individual (não misture arquivos de projetos diferentes na mesma pasta!).
5. Editar e salvar o arquivo fonte principal (ex.: *main.c*) no diretório do projeto.

Para inserir um novo arquivo fonte no projeto:

1. Selecionar Arquivo ▷ Novo ▷ Arquivo Fonte.
2. Responder "Sim" para acrescentar o arquivo ao projeto.
3. Editar e salvar o novo arquivo no diretório do projeto.

Atalhos para comandos úteis:

- Salvar: Control-S.
- Compilar: Control-F9.
- Compilar e executar: F9.

1 Estrutura de Um Programa C e Função Principal

Leitura Recomendada

Celes cap. 1: Conceitos fundamentais.
Kernigham cap. 1: Uma introdução através de exemplos.

Notas

Primeiro programa em C:

```
1 /* Primeiro programa C */
2
3 #include <stdio.h>
4 int main(void){
5   printf("Hello World\n");
6   return 0;
7 }
```

Observações:

- Um programa C é *case-sensitive*.
- Linha 1: Comentários começam com "/*" e terminam com "*/" (podem abranger várias linhas).
- Linha 3: Uma diretiva de compilação é uma instrução para o compilador. Inicia-se com "#" como primeiro caracter da linha. Algumas diretivas:
  1. *include*: inclui o arquivo indicado (no exemplo, *stdio.h,* que é o arquivo com as definições das funções padronizadas de entrada e saída).
  2. *define*: define uma macro.
  3. *if, elif, else, endif*: compilação condicional.
- Linha 4: função *main*: função principal pela qual inicia-se a execução do programa. Um programa C é uma coleção de funções. Detalhes:
  1. *int*: tipo de valor retornado pela função (inteiro).
  2. *main* nome da função.
  3. Entre parênteses: parâmetros recebidos pela função (*void* indica que a função *main* não recebe parâmetro algum).

- Linhas 4 e 7: chaves delimitadoras de blocos. Delimitam o corpo da função.
- Linha 5: exemplo de uma instrução executável. Observações:
  1. *printf* é uma chamada de função pré-definida na biblioteca do compilador. Trata-se de uma função de saída de propósito geral, que enviará para a saída padrão a *string* que aparece como seu argumento (entre parênteses).
  2. *Strings* são delimitadas por aspas duplas.
  3. \n é o caracter *newline*.
  4. Sentenças são terminadas por um caracter ponto-e-vírgula[1].
- Linha 6: outra instrução executável. *return* interrompe a execução da função corrente e retorna o resultado indicado para a função chamadora. No caso da função *main*, o programa é encerrado. O valor retornado (zero) indica (por convenção) uma execução sem problemas.

Outras observações gerais:

- Um programa C pode estar em mais de um arquivo fonte (o exemplo só tem um arquivo: *hello.c*)[2].
- Cada arquivo é compilado separadamente para um arquivo objeto: *gcc -c hello.c*.
- Os arquivos objeto resultantes são ligados, juntamente com a biblioteca padrão, formando um arquivo executável: *gcc -o hello hello.o*.

[1] O caracter ponto-e-vírgula é um *terminador* de sentenças, e não um separador como em outras linguagens (por exemplo, Pascal).

[2] Na verdade, um programa C é constituído por diversos arquivos-fontes (.c) e cabeçalhos (.h). Arquivos de cabeçalho são descritos nas páginas 42 e 43.

2 Tipos de Dados, Operadores e Expressões

Leitura Recomendada

Celes cap. 2: Expressões.
Kernigham cap 2: Tipos, operadores e expressões.

Notas

Tipos

- Todas as unidades em C têm um tipo.
- Um tipo de dados define:
  1. Os valores que a variável pode assumir.
  2. As operações que podem ser executadas sobre ela.
  3. A maneira como são armazenadas em memória.
- Em C, todas as variáveis devem ser declaradas (no início de um bloco) através da especificação de seu tipo.
- Exemplos de declarações de variáveis:

  int x;

  double velocidade;

  char inicial , final ;
- Na declaração, pode-se atribuir um valor inicial:

  int x = 0;

  double velocidade = 10.0;
- Os tipos primitivos mais comuns estão na tabela a seguir:

Tipos	Constantes	Controle
int, signed int	5, 05, 0x5	d
unsigned, unsigned int	5, 5U	u
short, short int, signed short int	5	d
unsigned short, unsigned short int	5U	u

long, long int, signed long int	5, 5L	ld
unsigned long, unsigned long int	5, 5LU	lu
char, signed char	'a', '\n', 5	c
char, unsigned char	'a', '\n', 5	c
float	5.2e10	e, f, g
double	5.2e10	le, lf, lg
long double	5.2e10	Le, Lf, Lg

- Os caracteres de controle são usados para a formatação das funções de entrada e saída, por exemplo:
  - *printf*: escrita de dados na saída padrão (normalmente o monitor).
  - *scanf*: leitura de dados da entrada padrão (normalmente o teclado).
- O programa abaixo trás um exemplo de utilização dessas funções:

```
1  #include <stdio.h>
2
3  int soma(int a, int b){
4    int r = 0;
5    r = a+b;
6    return r;
7  }
8
9  int main(void){
10   int x, y, z;
11
12   printf("Favor informar dois numeros inteiros");
13   scanf("%d %d", &x, &y);
14   z = soma(x,y);
15   printf("A soma de %d com %d e %d\n", x, y, z);
16   return 0;
17 }
```

- Valores lógicos (*booleanos*) são simulados através da utilização de inteiros: zero representa *false* e qualquer valor diferente de zero representa *true*.

Operadores e Expressões

- Os operadores e as expressões aritméticas funcionam de forma análoga à das linguagens já conhecidas.
- Operadores aritméticos:
  1. Soma: +
  2. Subtração: −

3. Multiplicação: $*$
4. Divisão: /
5. Resto de divisão inteira: %

- Exemplo:

```
#include <math.h>
double a = 1, b = -5, c = 6, x1;
x1 = (-b + sqrt(b*b - 4*a*c)) / (2*a);
```

- Observações:

1. Os operadores + e − possuem as versões unária e binária.
2. A prioridade e a associatividade funcionam da maneira usual.
3. Parênteses são usados para alterar a prioridade dos operadores.
4. O operador % só pode ser utilizado com operandos inteiros (**int**, **long**, etc).
5. O funcionamento do operador de divisão (/) varia de acordo com o tipo dos operandos: se ambos operandos forem inteiros, efetua-se uma divisão inteira; caso contrário, efetua-se uma divisão de ponto flutuante. Por exemplo, a expressão (7 / 2) fornece 3 como resultado, enquanto que as expressões (7.0 / 2.0), (7.0 / 2) e (7 / 2.0) fornecem 3.5.

- O operador de atribuição é =. Este operador tem as seguintes características:

1. O valor atribuído é retornado como em uma função:
   int x = 5, y, z;
   z = 5 * (y = 2*x + 3); /* *y = 13, z = 65* */
2. O operador associa à direita: "a = b = c = 4;" colocará 4 em a, b, c.

- Outros operadores de atribuição: +=, −=, $*$=, /=, %=, etc. Em geral, todo operador binário tem um operador de atribuição correspondente, com o seguinte significado (exemplificado para a soma, mas válido para os demais): "x += (y+2);" é equivalente a "x = x + (y+2);".

- Os operadores de incremento (++) são operadores unários que incrementam de uma unidade o valor do operando:

  int a = 5;

  a++; /*pos−incremento: a vale 6 */

  ++a; /*pre−incremento: a vale 7 */

- A diferença entre as formas de pré e pós-incremento é o valor retornado para a expressão:

  – O operador de pré-incremento faz primeiramente o incremento e depois retorna o valor:
    int a=3, b;
    b = ++a; /*a e b valem 4 */

- O operador de pós-incremento retorna o valor antes do incremento:
  int a=3, b;
  b = a++; /*a vale 4, b vale 3 */

- Existem também operadores de pré e pós-decremento (−−) que funcionam de forma inteiramente análoga, porém subtraindo 1 do operando.

- Operadores relacionais (funcionam da maneira usual):

  1. Igualdade: ==
  2. Diferença: !=
  3. Menor: <
  4. Menor ou igual: <=
  5. Maior: >
  6. Maior ou igual: >=

- Deve-se tomar cuidado para não confundir o operador de atribuição (=) com o de igualdade (==):

```
1 /* Forma correta da comparacao */
2 if(idade == 40)
3   printf("Velho\n");
4
5 /* Forma incorreta: atribui 40 a idade e retorna true */
6 if(idade = 40)
7   printf("Velho\n");
```

- Os operadores lógicos são:

  1. *and*: &&
  2. *or*: ||
  3. *not*: !

- Os operadores *and* e *or* são operadores de curto-circuito: a análise da expressão lógica é interrompida assim que se puder determinar um resultado. O exemplo a seguir não irá gerar erro mesmo que o valor da variável i seja menor que 0:

  if(i >= 0) && sqrt(i) < 10)...

- A linguagem C possui também um operador ternário (*expr1* ? *expr2* : *expr3*) que recebe três expressões, a primeira delas (*expr1*) retornando um valor lógico, e retorna o valor da segunda (*expr2*) se a primeira for verdadeira, ou o valor da terceira (*expr3*), caso contrário. Exemplo:

```
1 int a, b, c;
2 /* A expressao abaixo... */
3 a = b > 0 ? c + 1 : c - 1;
```

```
4
5 /* ... e equivalente a seguinte instrucao: */
6 if(b > 0)
7   a = c+1;
8 else
9   a = c-1;
```

Promoções

- Valores de tipos diferentes em expressões aritméticas são automaticamente promovidos para os tipos maiores:

```
1 char c; short s; int i; long l; float f; double d;
2
3 ... l+i ...    /* Promocao para long */
4 ... f = i;     /* Promocao para float */
5 ... d*s ...    /* Promocao para double */
6 ... c+s+i ... /* Promocoes para short e int */
```

- Observações:

  1. Não há detecção de *overflow*.
  2. As promoções referem-se aos valores intermediários no cálculo das expressões (os tipos das variáveis não são alterados).
  3. Conversões para tipos menores podem provocar a perda de dados. Por exemplo:
     a) float → int: perda da parte fracionária.
     b) int → short: perda dos dígitos mais significativos.

3 Instruções de Controle, Vetores e Matrizes

Leitura Recomendada

Celes cap. 3 (Controle de fluxo), 5 (Vetores e alocação dinâmica), até a seção *Passagem de vetores para funções.*
Kernigham cap. 3,5: Fluxo de controle e Apontadores e vetores.

Notas

Instruções de controle de fluxo

- Instrução de desvio condicional: *if*. Sintaxe:

  if (<expressao>) <instrucao>

- Observações:

  1. A expressão entre parênteses deve retornar um resultado lógico. A instrução que se segue só será executada se o resultado for verdadeiro. Os parênteses são obrigatórios.
  2. A instrução pode ser simples (terminada por ponto-e-vírgula) ou composta (um bloco de instruções delimitado por chaves).

- Exemplos:

```
1 if(idade > 40)
2   printf("Velho\n");
3
4 if(nota > 60){
5   printf("Aprovado\n");
6   aprovados++;
7 }
```

- Existe também a forma com o ramo alternativo:

  if (<expressao>) <instrucao1> **else** <instrucao2>

- Observações:

  1. O ramo representado por *instrucao1* será executado se a expressão lógica for verdadeira; caso contrário, *instrucao2* será executada.

2. Qualquer dos ramos pode ser constituído por instruções simples ou compostas.

- Exemplo:

```
1  if(nota < 60)
2    printf("Reprovado\n");
3  else {
4    printf("Aprovado\n");
5    aprovados++;
6  }
```

- Instrução *switch*:

```
1  switch (<expressao>){
2  case <const1>: <instrucoes1>
3  case <const2>: <instrucoes2>
4  ...
5  case <constn>: <instrucoesn>
6  default: <instrucoes>
7  }
```

- Observações:

1. A expressão da linha 1 deve ser do tipo integral (**int**, **char**, etc). O valor da expressão será comparado com os valores das contantes *const1, const2*, etc., que devem ser do mesmo tipo. As instruções *instrucoes1, instrucoes2*, etc., serão executadas a partir do ramo em que a comparação for bem sucedida. Pode haver várias instruções (simples e/ou compostas) em cada ramo.
2. O ramo com a marcação *default* (linha 6) será executado se nenhuma comparação for bem sucedida (este ramo é opcional).
3. A execução das instruções dos ramos pode ser interrompida pela utilização de uma instrução *break* (obtendo um efeito análogo ao da instrução *case* do Pascal).

- Exemplo:

```
1  switch(dia){
2  case 0:  printf("Domingo\n");  break;
3  case 1:  printf("Segunda\n");  break;
4  case 2:  printf("Terca\n");    break;
5  case 3:  printf("Quarta\n");   break;
6  case 4:  printf("Quinta\n");   break;
7  case 5:  printf("Sexta\n");    break;
8  case 6:  printf("Sabado\n");   break;
9  default: printf("Invalido\n"); break;
10 }
```

- Instrução *while*:

while (<expressao>) <instrucao>

- Observações:

  1. Esta é uma instrução de laço: a instrução (simples ou composta) será executada enquanto a expressão (lógica) for verdadeira.
  2. O laço é de pré-teste: o primeiro teste é executado antes de qualquer execução da instrução.

- Exemplo:

```
1 int sum = 0;
2 int i = 1;
3 while (i < 11) {
4   sum += i;
5   i++;
6 }
```

- Instrução *do-while*:

  do <instrucao> **while** (<expressao>)

- Observações:

  1. Esta é uma instrução de laço: a instrução (simples ou composta) será executada enquanto a expressão (lógica) for verdadeira.
  2. O laço é de pós-teste: o primeiro teste é executado após a primeira execução da instrução.

- Exemplo:

```
1 int sum = 0;
2 int i = 1;
3 do {
4   sum += 1;
5   i++;
6 } while (i < 11);
```

- Instrução *for*:

  for (<instrucoes1>; <expressao>; <instrucoes2>) <instrucao>

- Observações:

  1. Os grupos *instrucoes1* e *instrucoes2* podem ser formados por zero ou mais instruções simples separadas por vírgulas.
  2. O grupo *instrucoes1* é executado uma única vez antes do início do laço.
  3. A seguir, a expressão (lógica) é testada. Caso seja falsa, o laço é interrompido (ou seja, o laço é de pré-teste). A expressão é opcional (em caso de falta, é interpretada como verdadeira).

4. O corpo do laço (*instrucao*) é executado. O mesmo é constituído por uma instrução simples ou composta.
5. O grupo *instrucoes2* é executado, e em seguida a expressão volta a ser testada.
6. A utilização mais simples é a seguinte: o grupo *instrucoes1* é o grupo de inicialização, a expressão testa o final do laço, e o grupo *instrucoes2* incrementa as variáveis de constrole.

- Exemplo:

```
1 int sum, i;
2 for (sum = 0, i = 1; i < 11; i++)
3   sum += i;
```

- Instruções *break* e *continue*:
  - *break*: interrompe a execução de um laço.
  - *continue*: interrompe uma iteração do laço.

Arrays

- Um *array* é uma estrutura de dados primitiva homogênea (ou seja, uma coleção de elementos do mesmo tipo).
- Exemplo de declaração (vetor de *double*):

  double lucro[365];
- O vetor acima tem 365 elementos; a indexação é de 0 a 364:

  lucro [0] = 1; d = lucro [5];
- Exemplo: cópia de *arrays*:

```
1 int a[10], b[10], i;
2 for (i=0; i < 10; i++)
3   b[i] = a[i];
```

- *Arrays* multidimensionais são *arrays* de *arrays*:

  int tabela [10][10];
- Utilização:

  tabela [1][2] = 99;
- O primeiro índice representa a "linha", o segundo a "coluna". A faixa indexável é de tabela [0][0] a tabela [9][9] .
- A seguinte sintaxe pode ser utilizada para a inicialização dos *arrays* juntamente com a declaração (o tamanho do vetor é obtido automaticamente pelo compilador pela contagem dos elementos entre chaves):

  int vet [] = {5,6,10,3,−2,4};

4 Funções

Leitura Recomendada

Celes cap. 4: Funções
Kernigham cap. 4: Funções e estrutura de um programa, seções 4.1 (Conceitos básicos) a 4.10 (Recursividade).

Notas

Funções

- Um programa C é uma coleção de funções.
- Uma das funções deve se chamar *main* (a função inicial do programa).
- Uma função pode:
  - Receber parâmetros;
  - Declarar variáveis locais;
  - Conter instruções executáveis;
  - Retornar um valor.
- Uma função não pode declarar outra função.
- Sintaxe:

  `<retorno> <nome> (<parâmetros>) {<corpo>}`

  onde:

  1. `<retorno>` é o tipo do valor retornado pela função;
  2. `<nome>` é o nome da função.
  3. `<parâmetros>` é a lista de parâmetros recebida pela função. Consiste de uma série de `<tipo> <nome>` separados por vírgulas, ou *void* se a função não recebe parâmetro algum.
  4. `<corpo>` é o corpo da função, consistindo de declarações de variáveis locais e instruções executáveis.
- Exemplo:

```
1  #include <stdio.h>
2
3  int soma (int i, int j) {
4    return (i+j);
5  }
6
7  int main (void) {
8    int a = 2, b = 3, total;
9
10   total = soma(a,b);
11   printf("Soma: %d\n", total);
12   return 0;
13 }
```

- Observações:

  1. O programa é constituído de duas funções: *main* (linhas 7–13) e *soma* (linhas 3–5).
  2. A função *soma* recebe dois parâmetros inteiros i e j (linha 3) e retorna a soma dos mesmos (linha 4). Esses parâmetros são chamados de *parâmetros formais* e funcionam como variáveis locais à função *soma*.
  3. A função *main* declara variáveis locais (linha 8) e algumas instruções executáveis (linhas 10–12). Na linha 10 é feita uma chamada da função *soma*; os parâmetros a e b são chamados *parâmetros reais* e terão seus valores copiados para os parâmetros formais i e j, respectivamente.

- A passagem de parâmetros é sempre feita por valor (ou seja, os parâmetros formais recebem cópias do conteúdo dos parâmetros reais). Não existe passagem por referência, embora esta possa ser simulada com a utilização de ponteiros (ver p. 26).

- Pode-se utilizar também o qualificador *const* juntamente com um parâmetro. Isto indica que o mesmo não será modificado pelo corpo da função, permitindo que o compilador possa gerar um código mais eficiente. Exemplo: **int** funcao(**const int** param) {...}.

Retorno das funções

- Uma função pode retornar valores de qualquer tipo, exceto *arrays* ou outras funções.

- Como exemplo, a função abaixo retorna um caracter (uma resposta do tipo "sim ou não" recebida do usuário):

```
1  char simOuNao (void) {
2    char c;
3
4    do {
5      printf ("s/n?");
6      scanf ("%c", %c);
7    }while (c != 's' && c != 'n');
```

```
8
9     return c;
10 }
```

- A mesma pode ser chamada a seguinte forma:

  char ch; ch = simOuNao();

- Uma função pode retornar *void* (indicando que não retorna valor algum, sendo usada como um procedimento).

- Uma função termina:

  1. Ao encontrar a chave de fechamento.
  2. Ao executar uma instrução *return*. A expressão que segue *return* é o valor retornado pela função (cujo tipo deve corresponder ao declarado no cabeçalho).

- Na chamada da função, o valor retornado pela mesma pode ser ignorado.

Ordem de declaração

- Uma função deve ser definida antes de ser utilizada. A declaração da função serve como definição da mesma.

- Para que uma função possa ser declarada após o seu ponto de chamada, ou mesmo em um arquivo fonte diferente, a definição da mesma pode ser feita através da utilização de um *protótipo*.

- O protótipo especifica o nome da função, seu tipo de retorno e o tipo dos parâmetros recebidos. O protótipo deve aparecer antes de qualquer chamada da função. Exemplo:

```
1  #include <stdio.h>
2
3  int soma(int,int);
4
5  int main (void) {
6     int a = 2, b = 3, total;
7
8     total = soma(a,b);
9     printf("Soma: %d\n", total);
10    return 0;
11 }
12
13 int soma (int i, int j) {
14    return (i+j);
15 }
```

- A função *soma* é declarada nas linhas 13–15 e utilizada na linha 8. A definição é feita na linha 3 através de um protótipo.

Passagem de *arrays* como parâmetros

- *Arrays* podem ser passados como parâmetros para funções. Como exemplo, a função abaixo recebe um vetor de *float* e retorna o índice do menor elemento:

```
1  int menorElemento (float a[], int tamanho){
2    int i, menor = 0;
3
4    for (i = 1; i < tamanho; i++)
5      if (a[i] < a[menor])
6        menor = i;
7
8    return menor;
9  }
```

- Quando um vetor é passado como parâmetro de uma função, perde-se a informação sobre o tamanho do mesmo, de modo que um parâmetro extra deve ser passado contendo esse tamanho (linha 1).

- A chamada pode ser feita da seguinte maneira:

  int n; **float** lista [10]; n = menorElemento(lista,10);

- Uma forma melhor de informar o tamanho é utilizar o operador *sizeof*, que retorna o tamanho (em *bytes*) de seu argumento:

  n = menorElemento(lista, **sizeof**(lista)/**sizeof**(**float**));

Variáveis globais

- Variáveis globais podem ser declaradas no arquivo fonte fora do corpo de qualquer função.

- Variáveis globais existem durante todo o ciclo de vida do programa.

- Variáveis globais só são acessíveis a funções declaradas depois delas no mesmo arquivo fonte[1].

- Exemplo:

```
1  #include <stdio.h>
2  void func1 (void) {
3    printf ("Variavel global nao acessivel\n");
4  }
5
6  int g;
7
8  void func2 (void) {
9    g++;
10   printf ("Variavel global acessivel: %d\n", g);
11 }
```

[1] Este comportamento pode ser modificado.

```
12
13 int main (void) {
14   g = 5;
15   func1(); /* Imprime "Variavel global nao acessivel" */
16   func2(); /* Imprime "Variavel global acessivel: 6" */
17   return 0;
18 }
```

Estrutura de blocos

- Variáveis declaradas dentro de uma função:
  - São locais à função.
  - Só existem enquanto a função está sendo executada.
- Qualquer bloco ({ ... }) dentro de uma função pode declarar variáveis locais:

```
1  if (muitoEspaco) {
2    double matriz[100][100];
3    matriz[50][50] = 10.0;
4    ...
5  }
6  else {
7    float matrizPequena[10][10];
8    matrizPequena[5][5] = 5.0;
9    ...
10 }
```

- As variáveis declaradas dentro de um bloco:
  - Só existem dentro do mesmo.
  - Devem vir antes de qualquer instrução executável.
- Uma variável dentro de um bloco interno esconde a variável de mesmo nome no bloco externo:

```
1  int i;
2
3  i = 1;
4  printf("%d",i);   /* Imprime "1" */
5  {
6    int i = 2;
7    printf("%d", ++i); /* Imprime "3" */
8    i += 5;
9  }
10 printf("%d", ++i); /* Imprime "2" */
```

Recursão

- Funções podem chamar a si mesmas (direta ou indiretamente).
- Exemplo (série de Fibonacci):

```
1  #include <stdio.h>
2
3  int fib (int ind) {
4    if (ind == 1) return 0;
5    else if (ind ==2) return 1;
6    else return fib(ind-1) + fib(ind-2);
7  }
8
9  int main (void) {
10   int i;
11
12   scanf ("%d", &i);
13   printf ("fib(%d) = %d\n", i, fib(i))
14   return 0;
15 }
```

- A série de Fibonacci é uma série numérica cujo primeiro elemento é 0, o segundo é 1, e cada um dos elementos seguintes é a soma dos dois anteriores.
- A função *fib* acima (linhas 3–7) recebe um índice e calcula o elemento correspondente da série de Fibinacci. O cálculo é feito através de duas chamadas recursivas diretas (linha 6).
- Toda função recursiva deve ter uma forma de cortar a recursão para evitar um laço infinito do programa (linhas 4 e 5).
- O programa acima não representa uma forma eficiente de calcular elementos da série de Fibonacci. Não se deve empregar recursão quando existe uma solução não recursiva simples para o problema (a solução não recursiva é consideravelmente mais eficiente).
- Em geral, a recursão é empregada da seguinte forma:
  1. O problema é dividido em partes menores.
  2. Cada problema menor é resolvido recursivamente. Caso os problemas menores sejam suficientemente pequenos, eles são resolvidos de forma direta.
  3. A solução das partes menores é combinada para construir a solução do problema.
- Como exemplo, o algoritmo abaixo utiliza o método recursivo *MergeSort* para a classificação de um vetor:

  MergeSort (vetor V, inicio, fim): begin

  se inicio $<$ fim **então**

  meio $\leftarrow \left\lfloor \frac{\text{inicio+fim}}{2} \right\rfloor$

MergeSort(V, inicio, meio)

MergeSort(V, meio+1, fim)

Intercala(V, inicio, meio, fim)

fim se

end MergeSort

- Observações:

  1. A função recebe o vetor a ordenar e os índices dos elementos inicial e final do subvetor a tratar.
  2. O algoritmo divide o subvetor em duas metades e chama-se recursivamente para classificar cada uma.
  3. A função *intercala* toma dois subvetores previamente classificados e combina-os em um único (com o dobro do tamanho), intercalando os elementos na ordem correta.
  4. O algoritmo deve ser chamado a primeira vez informando os índices inicial e final do vetor a ser classificado.

Biblioteca padrão

- Os compiladores C vêm com uma grande biblioteca de funções que podem ser utilizadas pelo programador para desempenhar as mais diversas tarefas.
- Para utilizar uma função da biblioteca, é necessário incluir no arquivo fonte o seu cabeçalho.
- Exemplos de alguns cabeçalhos:
  - *ctype.h*: teste e manipulação de caracteres (funções *isdigit, isupper, toupper*, etc.)
  - *float.h*: constantes relacionadas com a representação de números em ponto flutuante (*FLT_MIN, DBL_MAX, LDBL_MAX*, etc.)
  - *limits.h*: constantes relacionadas com a representação de números inteiros (*INT_MIN, LONG_MAX, ULONG_MAX*, etc.)
  - *math.h*: funções matemáticas (*sin, cos, exp, log, pow, abs*, etc.)
  - *stdio.h*: funções de entrada e saída (*printf, scanf*, etc.)
  - *stdlib.h*: funções de propósito geral (*malloc, free, exit, rand*, etc.)
  - *string.h*: manipulação de *strings* e *arrays* (*strcpy, strcmp, strcat, memcpy*, etc).
  - *time.h*: manipulação de datas e horas (*localtime, time, clock*, etc.)

5 Ponteiros e Alocação Dinâmica de Memória

Leitura Recomendada

Celes p. 45–51 (Ponteiros de variáveis), cap. 5 e 6 (Vetores e alocação dinâmica e Matrizes).
Kernigham cap. 5: Apontadores e vetores.

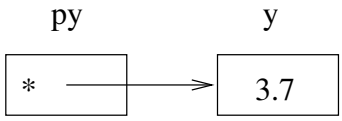
Notas

Ponteiros

- Um ponteiro é uma variável que contém o endereço de uma estrutura (variável simples, estrutura, função).
- Declaração:

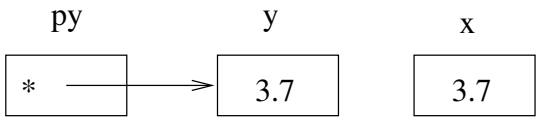
  float x,y, *py,*p;
- Atribuição de endereço — operador &:

  y = 3.7; py = &y;



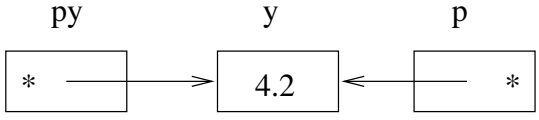
- De-referência (acesso ao elemento apontado via ponteiro) — operador *:

  x = *py



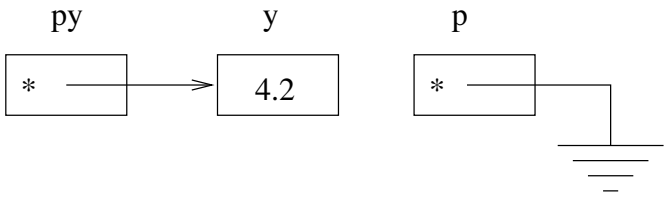
- Ponteiros podem criar *alias* para variáveis:

```
1 printf ("%f\n", *py);   /* imprime 3.7 */
2 *py = 4.2;
3 printf ("%f\n", y);   /* imprime 4.2 */
4 p = py;
5 printf ("%f\n", *p);   /* imprime 4.2 */
```



- Valor nulo do ponteiro: NULL (definido em stdio.h).

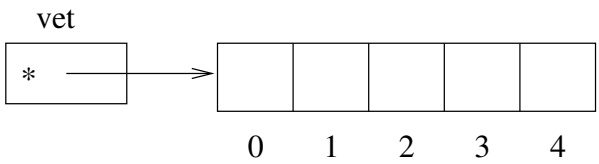
  p = NULL;



Arrays e ponteiros

- *Arrays* e ponteiros estão fortemente relacionados em C.
- O nome de um *array* é um ponteiro (constante) para o primeiro elemento do *array*:

  float vet [5];



- Como o nome do *array* é um ponteiro, as seguintes construções são equivalentes: ∗vet ⟷ vet [0].
- As seguintes operações são possíveis com ponteiros:

  1. Adição de ponteiro com inteiro: a soma de um ponteiro com um inteiro k tem o efeito de adiantar o ponteiro de k posições (ou seja, o tamanho do objeto apontado é levado em conta).
  2. Subtração de inteiro de ponteiro: a subtração de um inteiro$k$ de um ponteiro tem o efeito de recuar o ponteiro de k posições.
  3. Subtração de ponteiros: a subtração de dois ponteiros fornece o número de elementos entre os ponteiros.

- Assim, as seguintes construções são equivalentes: vet[j] ⟷ ∗(vet+j) ⟷ ∗(j+vet)
- Ponteiros podem ser usados para evitar indexação:

```
1  /* Este trecho de codigo troca dois elementos do vetor
2   * a utilizando indexacao
3   */
4  int a[tam], i, j, *pi, *pj, temp;
5  temp = a[i]; a[i] = a[j]; a[j] = temp;
6
7  /* Este trecho de codigo troca dois elementos do vetor
8   * a utilizando ponteiros
9   */
10 pi = &a[i]; pj = &a[j];
11 temp = *pi; *pi = *pj; *pj = temp;
```

- Isto pode se tornar relevante quando os elementos são referenciados dentro de um laço:

```
1  /* Copia do vetor a para o vetor b com indexacao */
2  void copia(int a[], int b[], int tam){
3    int i;
4
5    for(i=0; i<tam; i++)
6      b[i] = a[i];
7  }
8
9  /* Copia do vetor a para o vetor b com ponteiros */
10 void copia(int a[], int b[], int tam){
11   int i, *pi, *pj;
12
13   for(i=0, pi=a, pj=b; i<tam; i++, pi++, pj++)
14     *pj = *pi;
15 }
```

- Observações: alternativamente, o cabeçalho da função (linhas 2 ou 10) poderia ser escrito como:
  void copia(**int** *a, **int** *b, **int** tam)...

Ponteiros como parâmetros de funções

- Ponteiros fornecem um mecanismo que permite simular a passagem de parâmetros por referência.
- Para passar uma variável v por referência:
  1. Declarar o parâmetro formal como um ponteiro para o tipo de v.
  2. Passar como parâmetro atual o endereço de v.
  3. Dentro da função, utilizar sempre o operador de de-referência para referenciar v.
- Exemplo: função para calcular a soma de dois inteiros a e b e retornar o resultado no parâmetro s:

```
1 void soma (int a, int b, int *s) {
2   *s = a + b;
3 }
```

- A chamada dessa função poderia ser feita da seguinte forma:
  int i=1, j=2, k=0; soma (i, j, &k); /* *k agora contem 3* */
- Outro exemplo: função que encontra o maior e o menor valores de um vetor de inteiros:

```
1 void limites (float a[], int tam, float *min, float *max) {
2   int i;
3
```

```
4   *min = *max = a[0];
5     for (i = 1; i < tam; i++) {
6       if (a[i] < *min) *min = a[i];
7       if (a[i] > *max) *max = a[i];
8     }
9   }
```

- Utilização:

  float Min, Max, lista[10];

  limites(lista, 10, &Min, &Max);

Alocação dinâmica de memória

- Estruturas de dados estáticas têm seu tamanho determinado a tempo de compilação: variáveis simples, *arrays*, etc.
- Estruturas de dados dinâmicas têm seu tamanho determinado a tempo de execução e, portanto, precisam de mecanismos que façam alocação de memória durante a execução do programa.
- Memória pode ser alocada de uma região chamada *heap*.
- Memória alocada da *heap* deve ser devolvida quando não forem mais necessária.
- Funções para manipulação da *heap*: *malloc, calloc, realloc, free* (definidas em *stdlib.h*).
- A alocação de memória é feita com *malloc*: **void** *malloc(**int** tam); onde:
  - O parâmetro (*tam*) indica a quantidade de memória (em *bytes*) que deve ser alocada.
  - O valor retornado é um ponteiro para o início da área de memória alocada. Este ponteiro é de tipo genérico e necessita de sofrer uma conversão de tipo (*type cast*).
- Em caso de erro, malloc retorna NULL.
- A liberação de memória é feita com *free*: **void** free(**void** *ptr); onde o parâmetro (*ptr*) é o ponteiro para uma área que tenha sido alocada com *malloc*.
- Exemplo:

```
1 #include <stdlib.h>
2
3 /* Alocacao de um inteiro */
4 int *pi;
5 pi = (int*) malloc(sizeof(int));
6
7 /* Alocacao de um vetor de 10 caracteres */
8 char *vc;
9 vc = (char*) malloc(10);
```

```
10
11 /* Alocacao de um vetor de 10 inteiros */
12 int *vi;
13 vi = (int*) malloc(10 * sizeof(int));
14
15 /* Liberacao das areas alocadas */
16 free(pi); free(vc); free(vi);
```

- Observação: o operador **sizeof** (linhas 5 e 13) retorna o tamanho (em *bytes*) de uma estrutura qualquer (variável ou tipo).

Vetores dinâmicos

- Ponteiros podem ser utilizados para fazer alocação dinâmica de vetores.
- O tipo declarado deve ser um ponteiro para o tipo que será contido pelo vetor.
- A alocação é feita normalmente com a função *malloc*.
- Após a alocação, o vetor pode ser acessado da mesma forma que os vetores estáticos (através de indexação ou utilizando outros ponteiros).
- Ao final da utilização, o espaço alocado deve ser liberado com a função *free*.
- Exemplo: alocação e inicialização de um vetor de *double*:

```
1  #include <stdlib.h>
2
3  double *vetor;
4  int tam=10, i;
5  double d;
6
7  /* Alocacao */
8  vetor = (double*) malloc (tam * sizeof(double));
9
10 /* Inicializacao */
11
12 for (i=0; i<tam; i++)
13
14   vetor[i] = 0.0;
15
16 /* Utilizacao */
17
18 vetor[5] = 1.0; d = vetor[6];
19
20 /* Liberacao */
21 free(vetor);
```

Ponteiros para ponteiros

- Um vetor de inteiros é um ponteiro para inteiros:

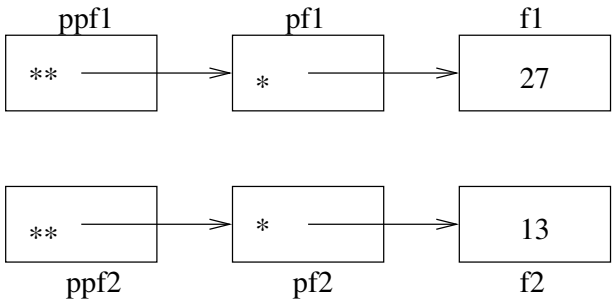
$$\textbf{int } a[] \longleftrightarrow \textbf{int } *a\ .$$

- Analogamente, um vetor de ponteiros para inteiros é um ponteiro para ponteiros para inteiros:

$$\textbf{int } *m[] \longleftrightarrow \textbf{int } **m\ .$$

- Exemplos:

```
1  float f1=27, f2=13, *pf1, *pf2, **ppf1, **ppf2;
2
3  pf1 = &f1;
4  pf2 = &f2;
5  printf("%f %f", *pf1, *pf2); /* Mostra "27 13" */
6
7  ppf1 = &pf1;
8  ppf2 = &pf2;
9  printf("%f %f", **ppf1, **ppf2);  /* Idem */
```



Matrizes Dinâmicas

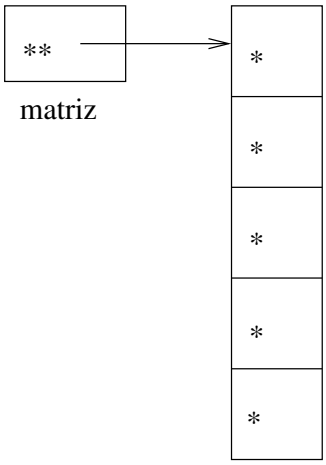
- Uma matriz é, na verdade, um vetor de vetores.
- Ou seja: uma matriz é um ponteiro para um vetor de ponteiros, cada um dos quais referencia um vetor comum.
- Assim, a variável matriz é um ponteiro para ponteiros (para o tipo de elementos da matriz).
- Exemplo: o trecho de código abaixo aloca dinamicamente uma matriz quadrada de inteiros, cuja dimensão é dada pela variável *tamanho*:

```
1  #include <stdlib.h>
2
3  int **matriz;
```

```
4  int tamanho = 5, i;
5
6  matriz = (int**) malloc (tamanho * sizeof(int*));
7
8  for (i=0; i < tamanho; i++)
9      matriz[i] = (int*) malloc (tamanho * sizeof(int));
```

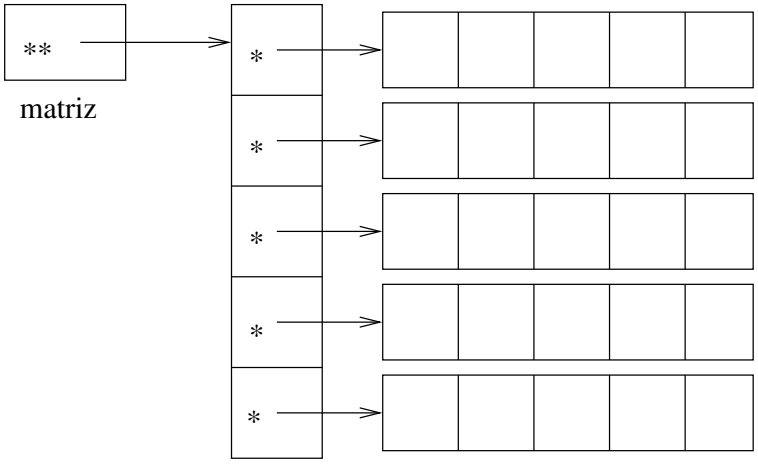
- Observações:

  1. A linha 3 declara a variável *matriz* através da qual a matriz será referenciada.
  2. Na linha 6 é feita a alocação dinâmica do vetor de ponteiros para as linhas da matriz. Após sua execução, a seguinte estrutura terá sido alocada (considerando que a variável *tamanho* contém o valor 5):



  Observar que cada elemento *matriz[i]* é do tipo ponteiro para inteiro e, portanto, pode servir para a alocação dinâmica de um vetor de inteiros tal como visto acima. Tais vetores correspondem às linhas da matriz.

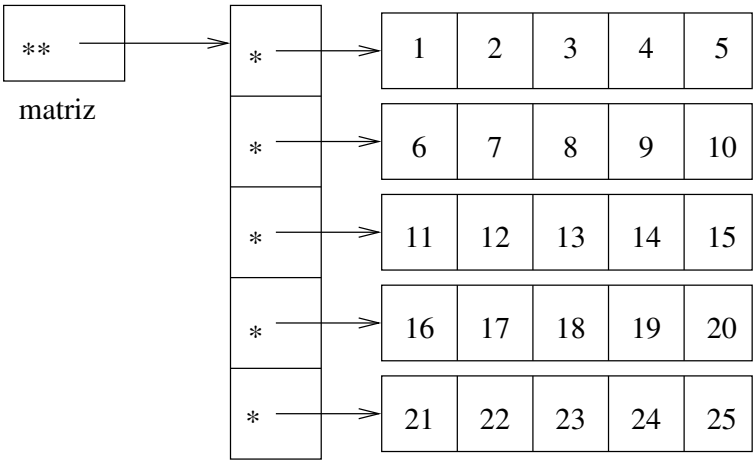
  3. As linhas 8–9 fazem a alocação dinâmica dos vetores mencionados no item anterior, completando a montagem da matriz. Após a sua execução, a estrutura a seguir terá sido alocada:



- Após a sua criação, a matriz pode ser utilizada da maneira usual, como se tivesse sido alocada estaticamente. O trecho abaixo mostra a inicialização de todos os elementos da matriz alocada acima com valores crescentes:

```
int inicio = 1, j;

for (i=0; i < tamanho; i++)
    for (j=0; j < tamanho; j++)
        matriz[i][j] = inicio++;
```

O resultado é:



- Após a utilização, a memória alocada deve ser liberada começando pelas linhas de dados (para que não se percam as referências necessárias para a chamada da função *free*):

```
for (i=0; i < tamanho; i++)
    free (matriz[i]);
free (matriz);
```

- Como cada linha da matriz é alocada por uma instrução separada, nada impede que as mesmas tenham tamanhos diferentes. As instruções abaixo alocam e manipulam uma matriz triangular:

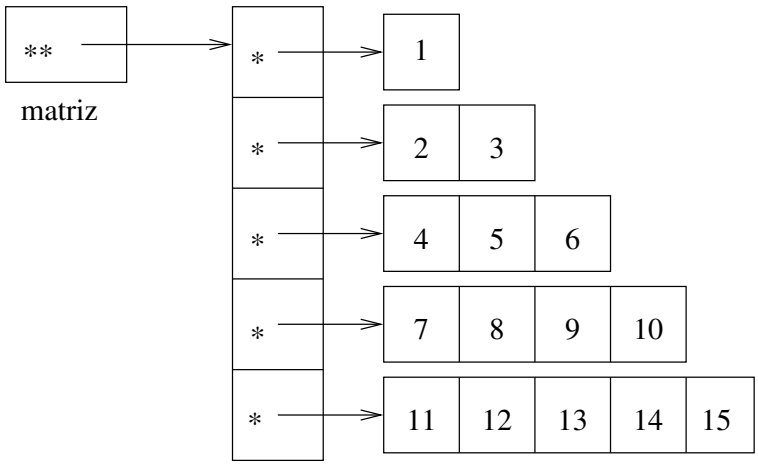
```
#include <stdlib.h>

int **matriz;
int tamanho = 5, inicio = 1, i, j;

matriz = (int**) malloc (tamanho * sizeof(int*));
for (i=0; i < tamanho; i++)
    matriz[i] = (int*) malloc ((i+1) * sizeof(int));

for (i=0; i < tamanho; i++)
    for (j=0; j <= i; j++)
        matriz[i][j] = inicio++;
```

```
13
14 for (i=0; i < tamanho; i++)
15     free (matriz[i]);
16 free (matriz);
```

- Observações:

  1. A alocação é feita nas linhas 6–8. A única diferença para o caso de alocação da matriz quadrada acima é a quantidade de elementos requisitados para cada linha (expressão $(i + 1)$ na linha 8).
  2. A inicialização é feita nas linhas 10–12. Reparar no controle de fim de laço para a variável j na linha 11.
  3. As linhas 14–16 fazem a liberação de memória da forma usual.

- A estrutura manipulada, antes de sua liberação, é a seguinte:



6 *Strings* e Arquivos

Leitura Recomendada

Celes cap. 7 (Cadeias de caracteres) e cap; 15 (Arquivos).
Kernigham cap. 7: Entrada e saída.

Notas

Strings

- Uma *string* é uma seqüência de caracteres terminada por *null* (caracter '\0').
- Constantes *string* são envolvidas por aspas duplas:

  "Estrutura␣de␣dados"

  Neste caso, o *null* é providenciado automaticamente pelo sistema.
- Varáveis *string* são vetores de caracteres:

  char nome[51];

  Observação: deve-se lembrar de reservar espaço para o *null*. O nome do exemplo acima pode ter, no máximo, cinqüenta caracteres.
- O cabeçalho <string.h> contém a definição de diversas funções de manipulação de *strings*. Exemplos:
  - **char** *strcpy (**char** *origem, **const char** *destino); : copia *origem* para *destino*. Retorna um ponteiro para *destino*.
  - **char** *strncpy (**char** *destino, **const char** *origem, size_t tamanho); : análoga a *strcpy*, porém copia no máximo *tamanho* caracteres. Observação: o tipo *size_t* é um tipo integral utilizado para indicar o tamanho de estruturas em *C*, e é dependente da plataforma utilizada.
  - **char** *strcat (**char** *destino, **const char** *origem); : concatena *origem* ao final de *destino*. Retorna um ponteiro para *destino*.
  - size_t strlen (**const char** *s); : retorna o tamanho (em caracteres) de *s* (o *null* final não é considerado). Ver acima observação sobre *size_t*).
  - **int** strcmp (**const char** *s1, **const char** *s2); : faz uma comparação lexicográfica de *s1* com *s2* retornando -1 ($s1 < s2$), 0 ($s1 = s2$) ou 1 ($s1 > s2$).

- A utilização de *strings* juntamente com as funções *scanf* e *printf* necessita da *string* de controle "%s". Por exemplo:

  scanf ("%s", nome);
  printf ("Nome:␣%s\n", nome);

Argumentos Para Programas

- Os argumentos passados para um programa quando o mesmo é executado são recebidos como parâmetros da função *main* da seguinte forma:

  int main (**int** argc, **char** *argv[]);

  ou

  int main (**int** argc, **char** **argv);

  onde *argv* é um vetor de *strings* cujo primeiro elemento (*argv[0]*) é o nome do programa e os demais (*argv[1]*, etc.) são os argumentos fornecidos. O tamanho do vetor é dado pelo parâmetro *argc*. O programa abaixo lista todos os argumentos fornecidos para um programa:

```
1  /* Programa args.c */
2  #include <stdio.h>
3
4  int main (int argc, char **argv) {
5      int i;
6      for (i = 0; i < argc; i++)
7          printf("argv[%d] = %s\n", i, argv[i]);
8      return 0;
9  }
```

- Quando executado através do comando:

  `args teste 1 2 3`

  a saída padrão exibirá:

```
argv[0] = args
argv[1] = teste
argv[2] = 1
argv[3] = 2
argv[4] = 3
```

Operações Simples Com Arquivos

- As notas abaixo são um simples resumo das operações que serão mais úteis para trabalhar com as estruturas de dados. Para maiores detalhes, consultar a bibliografia indicada.

- Observação: as funções e estruturas mostradas abaixo estão definidas no cabeçalho <stdio.h>.

- Variáveis do tipo "arquivo": FILE∗ . Exemplo:

  FILE ∗entrada;

- Abertura de arquivos: função *fopen*:

  FILE ∗fopen (**const char** ∗nome, **const char** ∗modo);

  onde *nome* é uma *string* com o caminho do arquivo a ser aberto e *modo* é uma *string* que indica o modo de abertura. Os modos mais comuns são "r" (*read*), "w" (*write*) e "a" (*append*). O valor retornado deve ser armazenado em uma variável tipo FILE∗ que será usada para todas as manipulações subseqüentes. Exemplo:

  entrada = fopen(nomearquivo,modo);

- Fechamento: função *fclose*:

  int fclose (FILE ∗f);

  O valor retornado indica o sucesso (= 0) ou fracasso (≠ 0) da operação. Exemplo:

  fclose (entrada);

- Verificação de final de arquivo: função *feof*:

  int feof (FILE ∗f);

  O inteiro retornado deve ser interpretado como um valor lógico que indica se o final de arquivo foi ou não atingido.

- Para a utilização das funções a seguir, o arquivo deve ter sido aberto no modo de leitura (*read*).

- Leitura de caracteres: função *fgetc*:

  int fgetc (FILE ∗f);

  Retorna o próximo caracter de f ou *EOF* se o final do arquivo tiver sido alcançado. Exemplo:

  ```
  int ch;
  ch = fgetc (entrada);
  if (ch == EOF) ...
  ```

- Leitura de *strings*: função *fgets*:

  char ∗fgets (**char** ∗s, **int** num, FILE ∗f);

  A função lê $num - 1$ caracteres de f e os coloca em s, providenciando automaticamente o *null* de encerramento da *string*. A leitura é interrompida se um final de arquivo ou um final de linha ("\n") forem encontrados (neste ultimo caso, a quebra de linha fará parte de s). Devolve um ponteiro para s. Exemplo:

  ```
  char linha[80];
  fgets (linha, sizeof(linha), entrada);
  ```

- Leitura formatada: função *fscanf:*

  int fscanf (FILE *f, **const char** *formato, ...);

  Atua de forma análoga à da função *scanf* (ver p. 9), fazendo a leitura de *f* e não da entrada padrão. Retorna o número de itens lidos ou *EOF*. Exemplo:

  ```
  int i; double d;
  fscanf (entrada, "%d %lg", i, d);
  ```

- Para a utilização das funções a seguir, o arquivo deve ter sido aberto em modo de escrita (*write* ou *append*), como no trecho de código a seguir:

  ```
  FILE *saida;
  saida = fopen ("nomearquivo","w");
  ```

- Gravação de caracteres: função *fputc*:

  int fputc (**int** ch, FILE *f);

  A função grava o caracter *ch* no arquivo *f*. Retorna o caracter gravado ou *EOF* na ocorrência de um erro. Exemplo:

  ```
  fputc ('a',saida);
  ```

- Gravação de *strings*: função *fputs*:

  int fputs (**const char** *s, FILE *f);

  Grava *s* em *f* (o *null* final não é gravado). Retorna *EOF* em caso de erro. Exemplo:

  ```
  fputs ("Teste", saida);
  ```

- Gravação formatada: função *fprintf*:

  int fprintf (FILE *f, **const char** *formato, ...);

  Atua de forma análoga à função *printf* (ver p. 9), fazendo a gravação em *f* e não na saída padrão. Retorna o número de caracteres gravados ou um valor negativo em caso de erro. Exemplo:

  ```
  int i = 5;
  fprintf ("i = %d\n", i);
  ```

7 Estruturas, Uniões e Declaração de Novos Tipos

Leitura Recomendada

Celes cap. 8: Tipos estruturados.
Kernigham cap. 6: Estruturas.

Notas

Criação de Tipos

- *typedef*: a instrução *typedef* permite dar nomes a novos tipos de dados e renomear os antigos. Exemplos:
  - **typedef float** real; : define *real* como um novo nome para *float*.
  - **typedef int** vetor[3]; : define *vetor* como um *array* de três inteiros.
- Com as definições acima, podem ser declaradas variáveis tais como "real x;" ou "vetor v;". Esta última poderia ser normalmente indexada: v[0], v[1], v[2].
- *Enumerações*: definem novos tipos através da especificação de todos os valores possíveis:
  enum boolean {FALSE, TRUE};
  enum diaSemana {DOMINGO, SEGUNDA, TERCA, QUARTA, QUINTA, SEXTA, SABADO};
- Enumerações podem ser usadas juntamente com *typedef*:
  typedef enum {DOMINGO,..., SABADO} diaSemana;
- Variáveis do novo tipo podem ser normalmente utilizadas:
  diaSemana d; d = SEGUNDA; **if** (d == TERCA)...

Estruturas

- *Estruturas* fornecem um meio de agrupar itens de informações de tipos diferentes (são estruturas heterogêneas, diferentemente dos vetores que são estruturas homogêneas). Exemplo:

```
1 struct Pessoa {
2   char nome[40];
3   int idade;
```

```
4     double salario;
5   }
```

- Variáveis podem ser declaradas da seguinte forma: **struct** Pessoa funcionario; .
- O acesso aos campos da estrutura é feito através do *operador-ponto*:

  funcionario.idade = 25;

  strcpy (funcionario.nome, "Jose da Silva");
- Utilização juntamente com *typedef*:

```
1  typedef struct Pessoa_tag {
2    char nome [40];
3    int idade;
4    double salario;
5  } Pessoa;
6
7  Pessoa aluno, professor; double s;
8  Pessoa departamento [40];
9
10 s = departamento[5].salario;
```

- Observações:

  1. Linhas 1–5: declaram um tipo *Pessoa* (que consiste de uma estrutura).
  2. Linha 1: *Pessoa_tag* é um identificador alternativo (não obrigatório) que pode ser utilizado para que a estrutura se auto-referencie.
  3. Linha 8: declara um vetor de quarenta pessoas.
  4. Linha 10: acessa o salário da sexta pessoa do vetor.
- Estruturas podem ser aninhadas:

```
1  typedef struct {
2    Pessoa p;
3    int quantidade;
4  } Nota;
5
6  Nota n; Nota an [10]; int i; double s; int q;
7
8  i = n.p.idade;
9  q = an[3].quantidade;
10 s = an[5].p.salario;
```

- Observações:

  1. Linhas 1–4: declaram o tipo *Nota* que possui como atributo um campo do tipo *Pessoa*.

2. Linha 8: obtém a idade da pessoa da nota *n*.
3. Linha 9: obtém a quantidade da quarta nota do vetor *an*.
4. Linha 10: obtém o salário da pessoa da sexta nota do vetor *an*.

Uniões

- *Uniões* têm sintaxe semelhante à das estruturas, com a palavra reservada *union* substituindo *struct*.
- Todos os membros de uma união ocupam a mesma área de memória.
- Ou seja: apenas um campo existe em um dado instante.
- Exemplo:

```
1  union Valor {
2    int iVal;
3    float fVal;
4    double dVal;
5  } v;
6
7  v.iVal = 5;
8  v.dVal = 135.7;
```

- Observação: a atribuição da linha 8 irá sobrepor o valor atribuído na linha 7.
- O programador deve controlar o tipo de valor armazenado em um dado instante.
- Em geral, uniões são usadas em conjunto com as estruturas, com um tipo enumerável para controlar o valor armazenado:

```
1  typedef struct {
2    enum {INT, FLOAT, DOUBLE} tipo;
3    union {
4      int iVal; float fVal; double dVal;
5    } val;
6  } Valor;
7
8  Valor v; v.tipo = INT; v.val.iVal = 5;
9
10 switch(v.tipo){
11   case INT: v.val.iVal...
12   case FLOAT: v.val.fVal...
13   case DOUBLE: v.val.dVal...
14 }
```

Alocação Dinâmica de Estruturas

- Estruturas pode ser alocadas dinamicamente com a função *malloc*.
- Para isso, deve-se utilizar um ponteiro para o tipo da estrutura.
- O acesso aos campos da estrutura a partir do ponteiro deve ser feito com o *operador-seta*. Exemplo:

```
1  typedef struct {
2    int id;
3    char nome[30];
4    int idade;
5  } Pessoa;
6
7  Pessoa *p;
8
9  p = (Pessoa*) malloc (sizeof(Pessoa));
10 p->id = 1; p->idade = 20;
11 strcpy (p->nome, "Jose da Silva");
12 free(p);
```

- Observações:

  1. Linha 7: declara um ponteiro p para a estrutura definida nas linhas 1–5.
  2. Linha 9: aloca dinamicamente uma estrutura do tipo *Pessoa*. A alocação é feita de forma inteiramente análoga à dos tipos primitivos.
  3. Linhas 10 e 11: demonstram o acesso aos campos da estrutura através do operador-seta. Formas análogas utilizando os operadores de de-referência e ponto seriam:
     (*p).id = 1; (*p).idade = 20;
     strcpy ((*p).nome, "Jose da Silva");
  4. Linha 12: libera a memória alocada.

- A alocação dinâmica de vetores e matrizes de estruturas é feita de forma análoga à dos tipos primitivos. Por exemplo, a alocação de um vetor de *Pessoa* é:

  Pessoa *vp;

  vp = (Pessoa*) malloc (5 * **sizeof**(Pessoa));

  vp[0].id = 2; vp[0].idade = 30; ...

8 Diretivas de Compilação

Leitura Recomendada

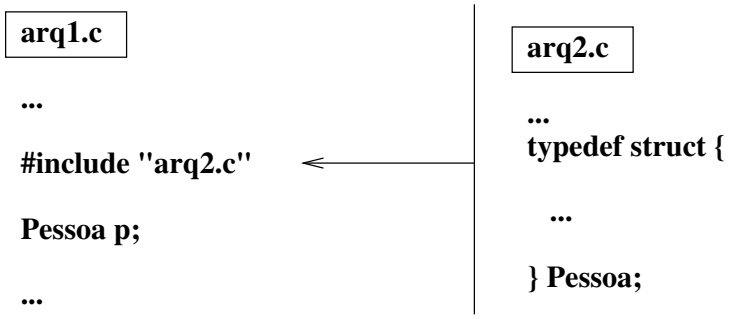
Celes p. 55–57 (Pré-processador e macros).
Kernigham cap. 4: Funções e estrutura de um programa, seção 4.11 (O pré-processador C).

Notas

- Diretivas de compilação são instruções para o pré-processador, que manipula o *texto* de um programa antes da compilação propriamente dita.
- Uma diretiva é identificada por uma linha cujo primeiro caracter não branco é '#'.
- Diretivas terminam no final da linha, a não ser que o último caracter da mesma seja '\'. Neste caso, a diretiva prossegue na linha seguinte.

Diretiva "include"

- Uma diretiva *include* instrui o compilador a incluir em um determinado ponto de um programa o conteúdo de outro arquivo. O caminho do arquivo a incluir deve ser indicado após a expressão *include*. Por exemplo, na figura abaixo, o texto do arquivo "arq2.c" será incluído na posição indicada no arquivo "arq1.c"[1], permitindo que o tipo *Pessoa* seja usada neste último.



- A diretiva *include* tem duas formas:
  - #**include** "arquivo" : a procura do arquivo é feita nos diretórios do usuário.
  - #**include** <arquivo> : a procura do arquivo é feita nos diretórios do sistema.
- Em geral, *include* é utilizado com *arquivos cabeçalho* (extensão ".h").

[1] O efeito é análogo ao de uma operação "copiar e colar" em um editor de textos.

- Um arquivo cabeçalho é utilizado por diferentes módulos de um programa e usualmente contém:
  - Definições de tipos.
  - Definições de constantes (macros, ver a seguir).
  - Protótipos de funções.
- Geralmente, nenhum arquivo cabeçalho contém declarações de variáveis ou corpo de funções.

Diretiva "define"

- Uma diretiva *define* pode ser utilizada para definir uma *macro* (que atua como uma *string* de substituição). Por exemplo:

  #**define** PI 3.1415927
- Toda vez que a *string* "PI" for encontrada no corpo do arquivo, a mesma será substituída pela *string* "3.1415927".
- O principal uso das macros é na definição de constantes:

  double area, raio;

  area = PI * raio * raio;
- Macros podem receber parâmetros entre parênteses (imediatamente após o nome da macro). Por exemplo, a macro a seguir define o máximo de dois valores numéricos:

```
1 #define MAX(x,y)        ((x) > (y) ? (x) : (y))
2
3 int j, a, b, c, d, x;
4
5 j = MAX(a,b);
6 ...
7 j = MAX(c+d, 3*x);
```

- Uma macro *não* é uma função; o que ocorre é uma substituição de *strings*. Por exemplo, após o pré-processamento, o texto real da linha 5 será:

  j = ((a) > (b) ? (a) : (b));

  e o da linha 7:

  j = ((c+d) > (3*x) ? (c+d) : (3*x));
- Uma macro pode ser removida com a diretiva *undef*:

  #**undef** MAX
- Nas linhas seguintes no arquivo fonte, a macro "MAX" não mais existirá.
- Observação: a utilização de letras maiúsculas para o nome das macros é apenas uma convenção.

Diretiva "if"

- A diretiva *if* permite fazer *compilação condicional* [2]:

```
1  #define MODELO     PEQUENO
2  #if MODELO == PEQUENO
3      #define MaxX       10
4      #define MaxY       5
5  #elif MODELO == MEDIO
6      #define MaxX       100
7      #define MaxY       50
8  #else
9      #define MaxX       1000
10     #define MaxY       500
11 #endif
12
13 int array [MaxX][MaxY];
```

- Supõe-se, no exemplo acima, que a macro "MODELO" poderia ser definida como "PEQUENO", "MEDIO" ou "GRANDE". Com a definição da linha 1, apenas as linhas 3 e 4 farão parte da compilação, definindo as dimensões da matriz da linha 13 como 10×5.

- Outras condições que podem ser usadas juntamente com *if* e *elif*:

  - #**if** defined(MODELO) : verifica se a macro "MODELO" está definida.
  - #**if** !defined(MODELO) : verifica se a macro "MODELO" não está definida.

- As técnicas de compilação condicional permitem construir arquivos de cabeçalho que possam ser incluídos diversas vezes pelo mesmo programa. Por exemplo, seja o arquivo *"cabecalho.h"* a seguir:

```
1 #if !defined (CABECALHO_H)
2 #define (CABECALHO_H)
3
4  ...
5  ...
6 #endif
```

- A técnica consiste em utilizar a compilação condicional para fazer com a macro da linha 2 (cuja existência é testada na linha 1) seja definida na primeira vez em que o arquivo for incluído. Tentativas de inclusão subseqüentes encontraram a macro já definida, de modo que a inclusão não será mais feita, evitando-se declarações repetidas do corpo do arquivo (linhas 3–5).

- Informações mais detalhadas e novas diretivas de compilação podem ser encontradas na bibliografia indicada.

[2] Observação: *elif* é uma contração de *else if*.

9 Listas Lineares

Leitura Recomendada

Ziviani seção 3.1: Listas Lineares.
Celes cap. 10: Listas Encadeadas.

Notas

- Uma lista linear é uma estrutura que corresponde a uma seqüência de zero ou mais itens $x_1, x_2, \ldots, x_n$ onde:

  1. x_i é um elemento de um determinado tipo.
  2. n é o tamanho da lista.

- A principal propriedade estrutural da lista envolve as posições relativas dos elementos; ou seja, o item x_i precede o item x_{i+1} e sucede o item x_{i-1}.

- Dito de outra forma: os itens ocupam posições determinadas na lista: x_i é o i-ésimo elemento da lista.

- As operações de *inserir*, *retirar* e *localizar* estão definidas para uma lista.

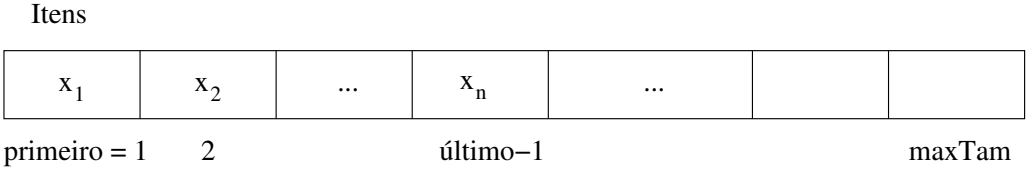
- A lista pode crescer ou diminuir em tempo de execução.

- Em geral, as listas são empregadas em aplicações em que não é possível prever a demanda de memória.

- Operações sobre uma lista:

  1. Criar a lista vazia.
  2. Inserir um novo elemento em qualquer posição.
  3. Retirar um elemento de qualquer posição.
  4. Localizar um elemento para leitura e/ou alteração.
  5. Percorrer todos os elementos aplicando sobre eles uma operação.
  6. Verificar as características da lista (por exemplo, número de elementos).

- Uma lista linear pode ser implementada através de um vetor:



onde:

1. *primeiro* é o índice do primeiro elemento do vetor;
2. *último* é o índice da próxima posição disponível;
3. *maxTam* é a capacidade máxima do vetor.

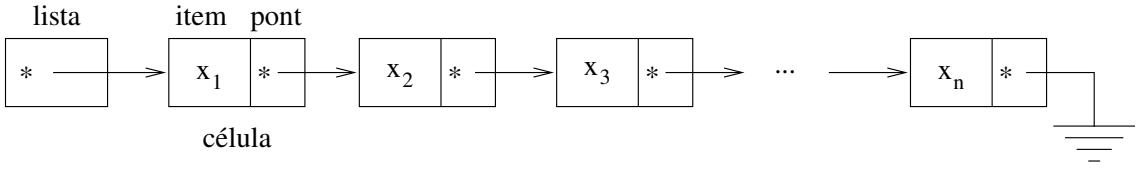
- Neste caso, a propriedade estrutural da lista é dada pelo armazenamento contíguo dos itens na memória.
- Vantagens:

1. Facilidade de implementação.
2. Utilização eficiente de memória.
3. Operações eficientes: inserção e remoção no final da lista, localização dos elementos.

- Desvantagens:

1. Lista limitada a um tamanho máximo.
2. Operações ineficientes: inserção / remoção de outros elementos que não o último.

- A implementação das listas com a utilização de ponteiros dá origem às *listas ligadas*.
- Nessa implementação, cada elemento possui, além dos dados próprios, um ponteiro para o elemento seguinte:



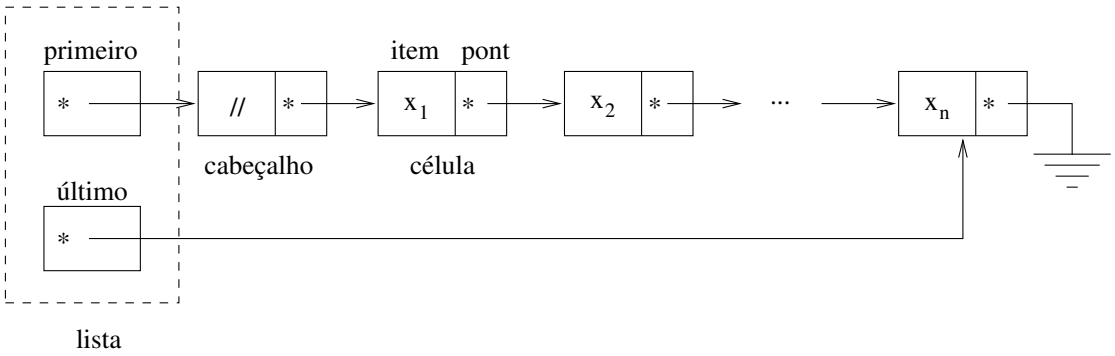
- A propriedade estrutural da lista é dada pelo encadeamento dos ponteiros (os elementos não estão, necessariamente, em posições contíguas de memória).
- Vantagens:

1. Inserções e retiradas são feitas sem a necessidade de deslocamento dos demais elementos.
2. A lista não tem um tamanho máximo.

- Desvantagens:

1. O acesso a elementos no meio da lista deve ser feito de forma seqüencial (e não aleatória).
2. O uso de ponteiros implica na necessidade de memória extra.

- *Implementação*: a implementação será feita utilizando a estrutura mostrada abaixo:



- A estrutura "lista" tem, na verdade, dois ponteiros: um para a primeira célula e outro para a última.

- A primeira célula é uma célula falsa (denominada "célula cabeçalho"), utilizada para simplificar as operações de inserção e remoção dos elementos na lista. O primeiro elemento (x_1) está, portanto, na segunda célula, e assim por diante.

- A listagem a seguir (que deve estar em um arquivo denominado *lista.h*) define os tipos de dados para a estrutura acima:

```
1  /*
2   * Definicao da estrutura de lista ligada
3   */
4  #if !defined(LISTA_H)
5  #define LISTA_H
6
7  /* Ponteiro para celula */
8  typedef struct CELULA_TAG *PONT;
9
10 /* Item que armazena os dados relevantes */
11 typedef struct {
12     int chave;
13     /* Outros componentes */
14 } ITEM;
15
16 /* Celula contendo um item e um ponteiro para
17  * a proxima celula
18  */
19 typedef struct CELULA_TAG {
20     ITEM item;
21     PONT prox;
22 } CELULA;
23
24 /* Lista: contem ponteiros para a primeira
25  * e a ultima celulas
26  */
```

```
27 typedef struct {
28    PONT primeiro, ultimo;
29 } LISTA;
30
31 /* Operacoes implementadas para a lista */
32 void cria    (LISTA*);
33 int  vazia   (LISTA);
34 int  insere  (ITEM, LISTA*);
35 int  retira  (int, LISTA*, ITEM*);
36 void imprime (LISTA);
37
38 #endif
```

- Observações:

  1. A estrutura que armazenará os dados (denominada *ITEM*) está definida nas linhas 10–14. Supõe-se que os mesmos são compostos por uma *chave* (linha 12) e por outros elementos (não mostrados)
  2. A estrutura *CÉLULA* (linhas 19–22) é o componente básico da lista. Contem um *item* (linha 20) e um ponteiro para a próxima célula (linha 21). O ponteiro para uma célula está definido na linha 8.
  3. A *LISTA* propriamente dita é uma estrutura (linhas 27–29) composta de um ponteiro para a primeira célula e outro para a última (linha 28).

- Existem diversas operações que podem ser realizadas sobre uma lista. Como exemplo, serão implementadas as seguintes:

  1. Criação da lista vazia.
  2. Verificação de lista vazia.
  3. Inserção de um item no final da lista.
  4. Remoção de um item com uma determinada chave.
  5. Impressão dos elementos da lista.

  As operações que retornam um valor inteiro seguem a seguinte convenção: o valor 0 indica sucesso da operação, e -1 indica a ocorrência de algum erro. A única exceção a essa regra é a operação *vazia,* cujo retorno deve ser interpretado como um resultado booleano. Nas operações em que a lista pode ter a sua estrutura modificada, a mesma é recebida por referência; nas demais, por valor. A listagem a seguir (que deve estar em um arquivo denominado *lista.c*) mostra a implementação das operações:

```
1 /*
2  * Implementacao das operacoes da lista
3  */
4 #include <stdio.h>
5 #include <stdlib.h>
6
```

```c
#include "lista.h"

/* Cria a lista vazia
 * (inclui a celula cabecalho)
 */
void cria(LISTA *lista){
   lista->primeiro = (PONT) malloc(sizeof (CELULA));
   lista->ultimo = lista->primeiro;
   lista->primeiro->prox = NULL;
}

/* Verifica se a lista esta vazia
 * (compara os ponteiros dos elementos extremos)
 */
int vazia(LISTA lista){
   return(lista.primeiro == lista.ultimo);
}

/* Insere um elemento no final da lista */
int insere (ITEM x, LISTA *lista){
   lista->ultimo->prox = (PONT) malloc(sizeof(CELULA));
   lista->ultimo = lista->ultimo->prox;
   lista->ultimo->item = x;
   lista->ultimo->prox = NULL;
   return 0;
}

/* Procura um elemento cuja chave e especificada
 * e remove-o da lista.
 */
int retira(int elemento, LISTA *lista, ITEM *item){
   PONT p, q;

   p = lista->primeiro;
   while (p->prox != NULL) {
      if (p->prox->item.chave == elemento){
        q = p->prox;
        *item = q->item;
        p->prox = q->prox;
        if (p->prox == NULL)
           lista->ultimo = p;
        free (q);
        return 0;
      }
      p = p->prox;
   }

```

```
54      return -1;
55  }
56
57  /* Percorre toda a lista imprimindo seus elementos */
58  void imprime(LISTA lista) {
59      PONT aux;
60
61      aux = lista.primeiro->prox;
62      while (aux != NULL) {
63          printf ("%d\n", aux->item.chave);
64          aux = aux->prox;
65      }
66  }
```

- Observações:

  1. A operação *cria* (linhas 12–16) recebe uma referência para a lista a ser criada, e faz a criação da lista vazia (a célula cabeçalho é alocada e ambos os ponteiros apontam para ela).
  2. A operação *vazia* (linhas 21–23) recebe uma lista e verifica se a mesma está vazia (ou seja, se ambos os ponteiros apontam para a mesma célula).
  3. A operação *insere* (linhas 26–32) recebe um *item* e uma referência para uma *lista*. Uma nova célula para conter o item é alocada e inserida no final da lista.
  4. A operação *retira* (linhas 37–55) recebe uma chave de um *elemento* e uma referência para uma *lista*. Procura-se na lista um elemento com a chave correspondente. Se este for encontrado, é retirado da lista e o item correspondente é devolvido no parâmetro *item* recebido por referência; caso contrário, retorna-se uma indicação de erro. *Importante*: o ponteiro auxiliar utilizado para a pesquisa do elemento está sempre uma célula atrasado, de modo a se ter uma referência correta para a remoção da célula.
  5. A operação *imprime* (linhas 58–66) percorre todas as células da lista imprimindo os dados de seus itens.

- A listagem a seguir (arquivo *main.c*) traz um exemplo de um programa que utiliza as operações acima:

```
1   /*
2    * Programa de teste para a lista ligada
3    */
4   #include <stdio.h>
5   #include "lista.h"
6
7   int main (void){
8       FILE *arq;
9       LISTA lista;
10      ITEM item;
```

```
11      int i;
12
13      /* Cria a lista */
14      cria(&lista);
15
16      /* Le e insere os elementos */
17      arq = fopen("arq.txt","r");
18      while(fscanf(arq, "%d", &i) != EOF) {
19          item.chave = i;
20          if(insere(item, &lista) == -1)
21              printf ("Erro na insercao de %d\n", i);
22      }
23
24      /* Imprime a lista criada */
25      imprime(lista);
26
27      /* Solicita uma chave e tenta remover o elemento.
28       * Imprime a lista para mostrar o resultado.
29       * -1 encerra o programa.
30       */
31      while(!vazia(lista)) {
32          printf("Favor informar o item a retirar: ");
33          scanf ("%d", &i);
34
35          if(i == -1)
36              break;
37
38          if(retira(i, &lista, &item) == -1)
39              printf("Elemento nao existe na lista\n");
40          else{
41              printf("Elemento retirado. Lista:\n");
42              imprime(lista);
43          }
44      }
45
46      return 0;
47  }
```

- Observações:

  1. A lista declarada na linha 9 é criada na linha 14.
  2. Os dados da lista são lidos de um arquivo externo diretamente para uma estrutura *item* (declarada na linha 10) e inseridos na lista (linhas 17–22).
  3. A lista criada é impressa (linha 25).
  4. O programa passa a executar um laço que solicita do usuário a chave de um elemento a remover (linhas 31–44). Se o usuário informar -1, o programa é encerrado. Caso

contrário, o elemento é procurado e, se encontrado, removido. A lista resultante (após a remoção) é exibida.

10 Pilhas e Filas

Leitura Recomendada

Ziviani seções 3.2 (Pilhas) e 3.3 (Filas).
Celes cap. 11 (Pilhas) e 12 (Filas).

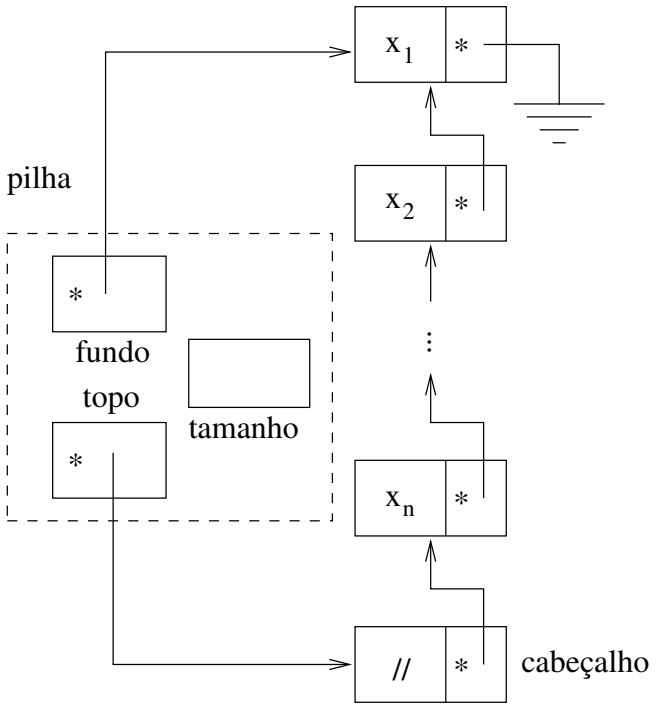
Notas

Pilhas

- Uma pilha é uma lista linear em que todas as inserções, retiradas e acessos são feitos em apenas um extremo, denominado *topo* da lista.
- Itens da pilha são colocados um "sobre" o outro, com o item mais recentemente inserido no topo e o menos recentemente no fundo.
- Propriedade fundamental: o último item a ser inserido é o primeiro a ser retirado (LIFO — *last in, first out*).
- Aplicação prática: caminhar por um conjunto de dados e guardar coisas para fazer posteriormente. Exemplo: processamento de estruturas aninhadas, onde as sub-estruturas mais internas tenham que ser processadas antes das mais externas:
  - Parênteses em expressões.
  - Chamadas de sub-programas.
- Operações do tipo abstrato *Pilha*:
  1. *cria (pilha)*: cria uma pilha vazia.
  2. *vazia (pilha)*: retorna *true* se a pilha está vazia, *false* caso contrário.
  3. *push (x, pilha)*: insere o item x no topo da pilha.
  4. *pop (pilha, x)*: retorna em x o item no topo da pilha, retirando-o da mesma.
  5. *examina (pilha, x)*: retorna em x o item no topo da pilha, sem retirá-lo da mesma.
  6. *tamanho (pilha)*: retorna o número de itens da pilha.

Implementação de pilhas com ponteiros

- Uma pilha pode ser implementada com uma estrutura semelhante à da lista ligada, com ponteiros para o início (topo) e final (fundo) da lista:



- Todas as inserções e remoções (empilhamento e desempilhamento) são feitas no final da lista, utilizando o ponteiro *topo*.
- A operação *examina* ("look") permite acessar o elemento no topo da pilha sem desempilhá-lo.
- É útil dispor de um campo para armazenar o tamanho da pilha (a quantidade de elementos empilhados) de modo a facilitar a operação *tamanho* (ver o campo *tamanho* na figura acima).
- A listagem a seguir (que deve estar em um arquivo denominado *pilha.h*) define os tipos da dados para a estrutura acima:

```
1  /*
2   * Definicao da estrutura de pilha
3   */
4  #if !defined(PILHA_H)
5  #define PILHA_H
6
7  /* Ponteiro para celula */
8  typedef struct CELULA_TAG *PONT;
9
10 /* Item que armazena os dados relevantes */
11 typedef struct {
12     int chave;
13     /* Outros componentes */
14 } ITEM;
15
```

```
16 /* Celula contendo um item e um ponteiro para
17  * a proxima celula
18  */
19 typedef struct CELULA_TAG {
20    ITEM item;
21    PONT prox;
22 } CELULA;
23
24 /* Pilha: contem ponteiros para o topo e o fundo
25  * da pilha e um contador de elementos
26  */
27 typedef struct {
28    PONT fundo, topo;
29    int tamanho;
30 } PILHA;
31
32 /* Operacoes implementadas para as pilhas */
33 void cria     (PILHA*);
34 int  vazia    (PILHA);
35 int  push     (ITEM, PILHA*);
36 int  pop      (PILHA*, ITEM*);
37 int  look     (PILHA*, ITEM*);
38 int  tamanho  (PILHA);
39 #endif
```

- Observações:

  1. A estrutura que armazena os dados (*ITEM*) está definida nas linhas 11–14.
  2. A estrutura *CÉLULA* está definida nas linhas 19–22. O ponteiro para uma célula está definido na linha 8.
  3. A *PILHA* é uma estrutura (linhas 27–30) composta de um ponteiro para o topo e um para o fundo da pilha (linha 28), e um campo *tamanho* (linha 29) para guardar a quantidade de elementos correntemente empilhados.

- As operações seguem as mesmas convenções mencionadas para as listas na p. 47 (obviamente, o valor retornado pela operação *tamanho* indica a quantidade de elementos empilhados). A listagem a seguir (que deve estar em um arquivo denominado *pilha.c*) mostra a implementação das operações:

```
1 /*
2  * Implementacao das operacoes da pilha
3  */
4 #include <stdlib.h>
5 #include "pilha.h"
6
7 /* Cria a pilha vazia
8  * (inclui a celula cabecalho)
```

```
9   */
10  void cria (PILHA *pilha){
11      pilha->topo = (PONT) malloc (sizeof (CELULA));
12      pilha->fundo = pilha->topo;
13      pilha->topo->prox = NULL;
14      pilha->tamanho = 0;
15  }
16
17  /* Verifica se a pilha esta vazia
18   * (compara os ponteiros dos elementos extremos)
19   */
20  int vazia (PILHA pilha){
21      return (pilha.topo == pilha.fundo);
22  }
23
24  /* Empilha um item */
25  int push (ITEM x, PILHA *pilha){
26      PONT aux;
27
28      aux = (PONT) malloc (sizeof (CELULA));
29      pilha->topo->item = x;
30      aux->prox = pilha->topo;
31      pilha->topo = aux;
32      pilha->tamanho++;
33      return 0;
34  }
35
36  /* Desempilha um item */
37  int pop (PILHA *pilha, ITEM *item){
38      PONT q;
39
40      if (vazia (*pilha))
41          return -1;
42
43      q = pilha->topo;
44      pilha->topo = q->prox;
45      *item = q->prox->item;
46      free (q);
47      pilha->tamanho--;
48      return 0;
49  }
50
51  /* Recupera (sem desempillhar) o item no topo da pilha */
52  int look (PILHA *pilha, ITEM *item){
53      if (vazia (*pilha))
54          return -1;
55
```

```
56      *item = pilha->topo->prox->item;
57      return 0;
58  }
59
60  /* Retorna a quantidade de itens da pilha */
61  int tamanho (PILHA pilha){
62      return pilha.tamanho;
63  }
```

- Observações:

  1. A operação *cria* (linhas 10–15) é inteiramente análoga à operação de mesmo nome para as listas ligadas (ver listagem da p. 47). Reparar que o tamanho da lista é corretamente inicializado (linha 14). Em todas as operações da pilha, o ponteiro *topo* fará as vezes de *primeiro* e *fundo* de *último*.
  2. A operação *vazia* (linhas 20–22) é análoga àquela de mesmo nome para as listas ligadas (ver listagem da p. 47).
  3. A operação *push* (linhas 25–34) insere uma nova célula no início da lista, que servirá como nova célula cabeçalho. O antigo cabeçalho é preenchido com o item recebido, e o tamanho é atualizado.
  4. A operação *pop* (linhas 37–49) retira o elemento no início da lista, retornando-o no parâmetro recebido. Na verdade, a célula retirada é o cabeçalho, sendo que aquela no início torna-se o novo cabeçalho. Antes da retirada, um teste é feito para verificar se a pilha contém elementos. Ao final da operação, o tamanho da pilha é atualizado.
  5. A operação *look* (linhas 52–58) apenas verifica se a pilha contém elementos, retornando no parâmetro recebido o primeiro deles.
  6. A operação *tamanho* (linhas 61–63) apenas retorna o atributo correspondente.

- A listagem a seguir (arquivo *main.c*) traz um exemplo de um programa que utiliza as operações acima:

```
1   /*
2    * Programa de teste para a pilha
3    */
4   #include <stdio.h>
5   #include "pilha.h"
6
7   int main (void){
8       FILE *arq;
9       PILHA pilha;
10      ITEM item;
11      int i;
12
13      /* Cria a pilha */
14      cria (&pilha);
```

```
15
16      /* Le e empilha os elementos */
17      arq = fopen ("arq.txt","r");
18      while (fscanf (arq, "%d", &i) != EOF) {
19          item.chave = i;
20          if (push (item, &pilha) == -1)
21              printf ("Erro no empilhamento de %d\n", i);
22      }
23
24      /* Retorna informacoes sobre a pilha
25       * (tamanho e elemento no topo).
26       */
27      printf ("Tamanho da pilha: %d\n", tamanho(pilha));
28      if(look(&pilha,&item))
29          printf("Erro no exame do topo da pilha\n");
30      printf ("Elemento no topo: %d\n", item.chave);
31
32      /* Desempilha e imprime todos os elementos */
33      printf ("Pilha: ");
34      while (!pop (&pilha, &item))
35          printf ("%d ", item.chave);
36      printf ("\n");
37
38      return 0;
39  }
```

- Observações:

  1. A pilha declarada na linha 9 é criada na linha 14.
  2. Os dados são lidos de um arquivo externo diretamente para uma estrutura *item* (declarada na linha 10) e empilhados (linhas 17–22).
  3. O tamanho da pilha e o elemento no topo são exibidos (linhas 27–30).
  4. O programa passa a executar um laço que desempilha e exibe todos os elementos (linhas 33–36).

Filas

- Uma fila é uma lista linear na qual as inserções são feitas em um extremo da lista (denominado *traseira*) e as retiradas e acessos em um outro extremo (denominado *frente*). Uma propriedade fundamental das filas é que o primeiro elemento a entrar é também o primeiro a sair (FIFO — *first in first out*). Filas são geralmente utilizadas em aplicações como simulações ou o controle da ordem de execução de tarefas em sistemas operacionais. As filas serão estudadas através de um exercício prático em laboratório.

11 Análise de Algoritmos

Leitura Recomendada

Ziviani cap. 1: Introdução.

Notas

Observação: as notas de aula dessa unidade são baseadas nas transparências do capítulo 1 de *Ziviani* (`http://www.dcc.ufmg.br/algoritmos/`).

Complexidade de Algoritmos

- A análise de algoritmos procura investigar o comportamento dos mesmos para levantar aquele que é o mais adequado para resolver um determinado problema.
- Existem dois tipos básicos de análise:
  1. *Análise de um algoritmo particular*: procura determinar o custo de um algoritmo específico para resolver um problema particular, através da análise de seu tempo de execução e/ou necessidade de memória.
  2. *Análise de uma classe de algoritmos*: procura determinar o menor custo possível para a resolução de um determinado problema. Investiga toda uma família de algoritmos e identifica aquele que é o melhor possível, colocando limites para a complexidade de todos os algoritmos daquela família.
- Este último tipo de análise permite determinar uma dificuldade inerente para a resolução de um problema. Se existe um algoritmo cujo custo corresponde a essa dificuldade, este é um *algoritmo ótimo* para o problema.

Função de Complexidade

- A análise de um algoritmo particular é feita através do estudo de seu custo de execução.
- Para medir o custo de execução define-se uma *função de complexidade* f:

  $$C = f(n)$$

  onde C é o custo e n representa o tamanho do problema a resolver.
- $f(n)$ pode medir o tempo necessário à execução do algoritmo para um problema de tamanho n (sendo chamada de *complexidade de tempo*) ou a quantidade de memória necessária para a execução (*complexidade de espaço*). Na seqüência, $f(n)$ irá referir-se sempre à complexidade de tempo.

- A complexidade de tempo mede, na realidade, não o tempo em segundos, mas a quantidade de vezes que cada instrução é executada. Em geral, limita-se a análise a uma instrução (ou grupo de instruções) considerada mais relevante.
- O parâmetro n indica o tamanho do problema a ser resolvido. Isto, em geral, relaciona-se com a quantidade de dados que devem ser manipulados (ou seja, com o tamanho da entrada de dados).
- Assim, o tempo de execução é função do tamanho da entrada de dados.
- Para alguns problemas, a determinação do tempo de execução depende não apenas do tamanho n, mas da forma como os dados de entrada estão organizados (por exemplo, o tempo de execução de um algoritmo de classificação pode variar, para entradas de mesmo tamanho, dependendo do fato de estar ou não a entrada previamente ordenada).
- Existem então três tipos de análise que podem ser feitas:

  1. *Análise de melhor caso*: determinação do menor tempo de execução para todas as entradas de tamanho n.
  2. *Análise de pior caso*: determinação do maior tempo de execução para todas as entradas de tamanho n.
  3. *Análise de caso médio*: média de todos os tempos de execução de todas as entradas de tamanho n. Depende da determinação de uma distribuição de probabilidades para o conjunto de entradas de tamanho n. Em geral, supõe-se que todas as entradas são igualmente prováveis.

- Exemplo: pesquisa seqüencial dos registros de um arquivo para encontrar um registro com uma determinada chave. Seleciona-se como operação relevante a leitura de um registro do arquivo e a comparação de sua chave com a chave desejada; ou seja, $f(n)$ representa o número de consultas necessárias para encontrar um registro com a chave dada (onde n é a quantidade de registros no arquivo).

  1. *Melhor caso*: o registro procurado é o primeiro a ser lido. Só é necessária uma consulta. Tem-se:
  $$f(n) = 1.$$
  2. *Pior caso*: o registro procurado não se encontra no arquivo. São necessárias n consultas. Tem-se:
  $$f(n) = n.$$
  3. *Caso médio*[1]: o registro procurado é o i-ésimo registro do arquivo. São necessárias i comparações. Sendo p_i a probabilidade de que isso ocorra, o custo da busca seria $i \times p_i$. A média para i variando de 1 a n é:
  $$f(n) = 1 \times p_1 + 2 \times p_2 + 3 \times p_3 + \cdots + n \times p_n$$
  Considerando $p_1 = p_2 = \cdots = p_n = 1/n$ vem:
  $$f(n) = \frac{1}{n}(1 + 2 + 3 + \cdots + n) = \frac{n+1}{2}$$
  Ou seja, aproximadamente metade dos registros serão consultados.

[1] Considerando que toda pesquisa irá recuperar um registro.

Limite Inferior

- A análise de uma classe de algoritmos procura determinar um limite inferior para o custo de resolução de um determinado problema. Assim, é possível determinar se um algoritmo que resolve um problema é *ótimo* ou se é possível que exista um outro mais eficiente.

- Em geral, esta análise é mais complexa que a de algoritmos particulares.

- Exemplo[2]: sejam $2n$ números distintos distribuídos em dois vetores A e B de tamanho n, ordenados da seguinte forma:

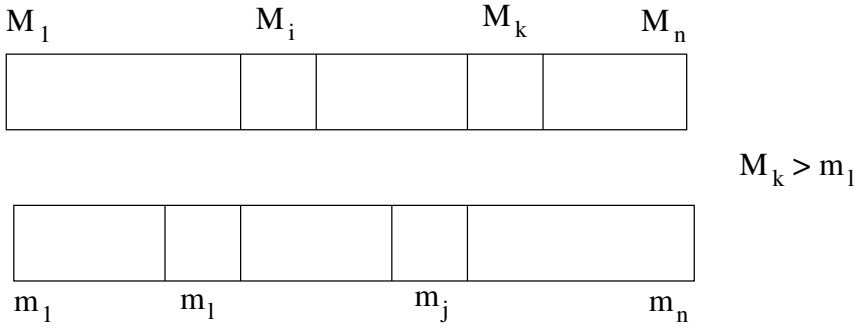
$$A_1 > A_2 > \cdots > A_n$$

$$B_1 > B_2 > \cdots > B_n$$

  Deseja-se encontrar um limite inferior para o número de comparações necessárias para resolver o seguinte problema: encontrar o n-ésimo maior número dentre os $2n$ elementos.

- O n-ésimo maior elemento é menor que $n-1$ elementos e maior que n elementos. Não faz sentido comparar elementos do mesmo vetor (já estão ordenados).

- Comparando o k-ésimo elemento de um vetor com o l-ésimo elemento de outro ($1 \leq k, l \leq n$), e chamando de M o vetor que vencer a comparação[3] (seja o de índice k) e de m o que perder (índice l), tem-se:



- Então: para $i = 1, \cdots, k$, M_i é maior que $(n-i)+n-(l-1) = 2n+1-(l+i)$ elementos. Para eliminar da busca o i-ésimo elemento de M (e todos os anteriores) é necessário que $2n+1-(l+i) > n$ ou:

$$i < n - l + 1.$$

- Da relação acima, para eliminar o k-ésimo elemento elemento de M (e todos os anteriores $1, 2, \ldots, k-1$), $k < n - l + 1$ ou:

$$k + l < n + 1 \tag{11.1}$$

  ou seja, $k + l \in [2, n]$. Neste caso seriam eliminados k elementos.

[2] *Ziviani* p. 31, exercício 11 (parcial).
[3] Ou seja, contiver o maior elemento daquela comparação.

- Da mesma forma, para $j = l, l+1, \ldots, n$, m_j é menor que $(j-1)+k = j+k-1$ elementos. Para eliminar o j-ésimo elemento de m (e todos os posteriores) é necessário que $j+k-1 > n-1$ ou:

$$j > n-k$$

- Da relação acima, para eliminar o l-ésimo elemento de m (e os posteriores $l+1$, $l+2$, $\ldots$, n), $l > n-k$ ou:

$$k+l > n \tag{11.2}$$

ou seja, $k+l \in [n+1,\, 2n]$. Neste caso seriam eliminados $n-(l-1)$ elementos.

- As condições 11.1 e 11.2 são incompatíveis. Deve-se abrir mão da eliminação de um dos elementos de um dos vetores. Eliminando o $(k-1)$-ésimo elemento de M (e os anteriores) vem $k-1 < n-l+1$ ou:

$$k+l < n+2 \tag{11.3}$$

ou seja, $k+l \in [2,\, n+1]$; seriam eliminados $k-1$ elementos.

- De 11.3 e 11.2: $k+l = n+1 \Rightarrow k-1+n-(l-1)$ elementos seriam eliminados. Chamando de E este número de elementos vem:

$$E = k-1+n-l+1 = n+k-l;$$

$$l = n-k+1 \Rightarrow E = n+k-(n-k+1) = 2k-1.$$

- No caso geral, não se pode tomar $k = n$ (com $l = 1$) para maximizar o número de elementos eliminados, porque não é possível determinar de antemão qual é o vetor M e qual é o m (uma escolha incorreta eliminaria apenas um elemento no lugar de $2n-1$). O mesmo raciocínio vale para $k = n-1$ (com $l = 2$), etc. O melhor a fazer é escolher $k = l = \lfloor (n+1)/2 \rfloor$, quando uma única comparação eliminaria metade dos $2n$ elementos.

- Os elementos restantes constituem dois novos vetores com metade dos elementos, e o processo pode ser repetido.

- Como cada comparação elimina metade dos elementos, após i comparações restariam $2n/2^i$. No final, deverá sobrar apenas um elemento e terão sido realizadas c comparações:

$$\frac{2n}{2^c} = 1 \Rightarrow c = \lceil \log 2n \rceil .$$

- Observação: o raciocínio quando se opta por eliminar o $(l+1)$-ésimo elemento de m é análogo:

$$(l+1) > n-k \Rightarrow k+l > n-1 \Rightarrow k+l \in [n, 2n] \tag{11.4}$$

com $n-l$ elementos eliminados. De 11.1 e 11.4: $k+l = n \Rightarrow k+(n-l)$ elementos eliminados. Vem: $E = n+(k-l)$; $k+l = n$ e $E = 2k$ (k máximo $= n-1$).

- A listagem a seguir mostra um programa que implementa o processo determinado acima:

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

int encontraNMaior(int *a, int *b, int tam){
  int inicioA = 0, finalA = tam-1;
  int inicioB = 0, finalB = tam-1;
  int k;

  while(tam > 1){
    k = (tam+1)/2;
    if(a[inicioA+k-1]>b[inicioB+k-1]){
      inicioA = finalA - k + 1;
      finalB = inicioB + k - 1;
    }
    else{
      finalA = inicioA + k - 1;
      inicioB = finalB - k + 1;
    }
    tam = finalA - inicioA + 1;
  }

  return a[inicioA]>b[inicioB] ? a[inicioA] : b[inicioB];
}

int main(int argc, char *argv[]){
  int A[] = {14,13,12,10,8,4,1};
  int B[] = {11,9,7,6,5,3,2};

  printf("N-esimo maior elemento = %d\n",
         encontraNMaior(A,B,sizeof(A)/sizeof(int)));
  return 0;
}
```

Notação O e comportamento assintótico.

- A escolha de um algoritmo ótimo (ou apenas mais eficiente) não é crítica quando o tamanho do problema a tratar é pequeno (ou seja, para valores pequenos de m).
- No estudo das funções de complexidade, é importante conhecer o seu funcionamento para grandes valores de n. Este comportamento é chamado de *assintótico*.
- O comportamento assintótico de uma função $f(n)$ representa o limite de de seu comportamento quando n cresce.
- Para o estudo do comportamento assintótico de funções, utiliza-se a noção de *dominação assintótica*: sejam duas funções $f(n)$ e $g(n)$. Diz-se que $f(n)$ domina assintoticamente

$g(n)$ se existem duas constantes positivas c e m tais que, para $n \geq m$ vale a seguinte relação:

$$|g(n)| \leq c \cdot |f(n)|.$$

- Se $f(n)$ domina assintoticamente $g(n)$, pode-se escrever $g(n) = O(f(n))$ e diz-se que $g(n)$ é da ordem de, no máximo, $f(n)$. Esta é a chamada *notação O*.
- Exemplos:
  1. $(n+1)^2 = O(n^2)$.
  2. $3n^3 + 2n^2 + n = O(n^3)$.
  3. $\log_5 n = O(\log n)$.
- É possível mostrar que as seguintes operações são válidas para a notação O:
  1. $f(n) = O(f(n))$.
  2. $c \times O(f(n)) = O(f(n))$ (para c constante).
  3. $O(f(n)) + O(f(n)) = O(f(n))$.
  4. $O(O(f(n))) = O(f(n))$.
  5. $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$.
  6. $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$.
  7. $f(n) \cdot O(g(n)) = O(f(n) \cdot g(n))$.
- Existem também as notações Ω, Θ e o para o estudo do comportamento assintótico das funções. Para maiores detalhes, consultar *Ziviani* cap 1.
- O estudo das funções de complexidade permite dividir as funções em classes de complexidade. As principais classes são as seguintes (em ordem crescente de custo):
  1. $f(n) = O(1)$:
     - Este é a classe dos algoritmos de complexidade constante, que independem do tamanho da entrada.
  2. $f(n) = O(\log n)$:
     - Esta é a classe dos algoritmos de complexidade logarítmica.
     - Em geral, refere-se a algoritmos que resolvem um determinado problema transformando-o em uma série de outros problemas menores (ver, por exemplo, o algoritmo da p. 61).
  3. $f(n) = O(n)$:
     - Esta é a classe dos algoritmos de complexidade linear.
     - Em geral, refere-se aos algoritmos que tratam os elementos de uma entrada fazendo um pequeno processamento e, possivelmente, gravando-os em uma saída.
  4. $f(n) = O(n \log n)$:

   - Algoritmos desta classe tratam um problema dividindo-o em problemas menores, resolvendo cada parte (por vezes, recursivamente) e depois ajuntando as soluções (ver, por exemplo, o algoritmo *mergesort* da p. 22).

5. $f(n) = O(n^2)$:
   - Esta é a classe dos algoritmos de complexidade quadrática.
   - Esta complexidade é típica de algoritmos que processam elementos aos pares, em geral com laços aninhados.
   - Algoritmos dessa classe são úteis para resolver problemas pequenos.
6. $f(n) = O(2^n)$:
   - Esta é a classe dos algoritmos de complexidade exponencial.
   - Em geral, refere-se a algoritmos que procuram resolver um problema pela abordagem de "força bruta".
   - Algoritmos desta classe têm pequena aplicação prática, e limitam-se a problemas de tamanho muito pequeno.
7. $f(n) = O(n!)$:
   - Esta é a classe dos algoritmos de complexidade fatorial.
   - Em geral, refere-se também a problemas que utilizam a abordagem de "força bruta".
   - A utilidade prática é ainda mais limitada que a dos algoritmos exponenciais.

- Dentre as classes acima, distinguem-se dois grupos mais importantes:

  1. Os algoritmos polinomiais, com complexidade na forma $O(p(n))$, onde $p(n)$ é um polinômio.
  2. Os algoritmos exponenciais, com complexidade na forma $O(c^n)$, onde c é uma constante inteira, $c > 1$.

- De forma geral, estes dois grupos permitem fazer algumas afirmações sobre a viabilidade do tratamento dos problemas através de algoritmos:

  - Se, para um determinado problema, existe um algoritmo polinomial capaz de solucioná-lo, então o mesmo é considerado bem resolvido.
  - Se tal algoritmo não existe (por exemplo, o único algoritmo existente pode ser exponencial), então o problema é considerado intratável.

- Algoritmos exponenciais podem ser úteis em algumas aplicações práticas, em geral para o tratamento de problemas com entradas de tamanho suficientemente pequeno.

- Para executar a análise de um algoritmo utilizando a notação O, as seguintes regras devem ser observadas (utilizando também as regras de operações da p. 63)

  1. Comandos simples (atribuição, leitura, escrita, etc.) são $O(1)$.
  2. Seqüências de comandos: soma dos tempos de execução dos comandos da seqüência.

3. Comando de decisão: soma dos tempos dos comandos do corpo do comando condicional, mais o tempo para analisar a condição (geralmente $O(1)$).
4. Laço: soma do tempo de execução do corpo do anel com o tempo de avaliação da condição de terminação (em geral, $O(1)$), multiplicado pelo número de iterações. Obs.: para laços aninhados, analisar primeiramente os mais internos.
5. Procedimentos não recursivos: analisar segundo as regras acima, começando pelos procedimentos que não chamam outros procedimentos. Para os procedimentos que chamam outros, considerar o custo de chamada previamente calculado.
6. Procedimentos recursivos: associa-se ao procedimento uma função de complexidade $f(n)$ desconhecida, onde n representa o tamanho dos argumentos do procedimento, e obtem-se uma relação de recorrência para $f(n)$ (para detalhes, consultar *Ziviani* cap. 1).

- Como exemplo, a análise do procedimento *MergeSort* (p. 22) é feita a seguir.
- Seja $T(n)$ a função de complexidade associada ao procedimento. Tem-se (onde a é constante e cn, c constante, é o custo linear associado à função *merge*):

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ T(n/2) &= 2T(n/4) + cn/2 \\ T(n/4) &= 2T(n/8) + cn/4 \\ &\cdots \\ T(n/2^i) &= 2T(n/2^{i+1}) + cn/2^i \\ &\cdots \\ T(1) &= a \end{aligned}$$

- Substituindo as linhas acima uma nas outras vem:

$$\begin{aligned} T(n) &= 2(2T(n/4) + cn/2) + cn \\ &= 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn \\ &= 8T(n/8) + 3cn \\ &= \cdots \\ &= 2^iT(n/2^i) + icn \end{aligned}$$

- Considerando $n = 2^k$, vem $k = \log n$ e:

$$\begin{aligned} T(n) &= 2^kT(1) + kcn \\ &= a2^k + kcn \\ &= an + cn\log n \end{aligned}$$

- Ou seja, $T(n) = an + cn\log n = O(n\log n)$.

12 Ordenação

Leitura Recomendada

Ziviani cap. 4: Ordenação.
Celes cap. 16: Ordenação.

Notas

- Ordenar é rearranjar um conjunto de objetos de forma ascendente ou descendente.
- Os algoritmos de ordenação trabalham sobre os registros de uma coleção.
- Uma parte do registro, chamada *chave*, é utilizada para controlar a ordenação:

```
1 typedef struct {
2     int chave;
3     /* Outros componentes do registro */
4 } REGISTRO;
```

- A chave (linha 2) pode ser de qualquer tipo que possua uma regra de ordenação bem definida (ou seja, para o qual as operações "igual", "menor que" e "maior que" façam sentido).
- Os métodos de ordenação podem ser classificados em métodos de *ordenação interna* e *ordenação externa*:
  - Ordenação interna: a coleção a ser ordenada cabe toda na memória principal (implicando que qualquer dos registros pode ser imediatamente acessado).
  - Ordenação externa: a coleção a ser ordenada está em memória secundária (implicando que o acesso aos registros deve ser feito de forma seqüencial).
- Os algoritmos de ordenação interna que serão estudados procuram utilizar a memória de forma eficiente, ou seja, os elementos de um vetor serão ordenados utilizando-se a área do próprio vetor (a menos de uma pequena área auxiliar).
- Classificação da ordenação interna:
  - Métodos simples: necessitam de um número de comparações da ordem de n^2, onde n é o tamanho da coleção a ordenar.
  - Métodos eficientes: necessitam de um número de comparações da ordem de $n \cdot \lg n$.
- Os algoritmos de ordenação podem ainda ser classificados como *estáveis* ou *instáveis*:

  - Estáveis: a posição relativa de itens de mesma chave é mantida na coleção resultante.
  - Instáveis: nada se pode dizer sobre a posição relativa de itens de mesma chave na coleção final.

Ordenação por Seleção

- A ordenação por seleção é um método simples que trabalha de acordo com o seguinte princípio: selecionar o menor item do vetor e trocá-lo com o item na primeira posição; repetir o mesmo processo para o vetor resultante com os $n-1$, $n-2$, etc., elementos restantes, até que só reste um elemento.
- O programa abaixo mostra uma implementação deste método:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <limits.h>
4
5  #define MAXTAM  20
6
7  typedef long CHAVE;
8  typedef struct ITEM_TAG {
9    CHAVE chave;
10   /* Outros componentes */
11 } ITEM;
12 typedef int INDICE;
13 typedef ITEM VETOR[MAXTAM];
14
15 void selecao(ITEM *a, int tam){
16   INDICE i,j,min;
17   ITEM x;
18
19   for(i=0; i < tam-1; i++){
20     min = i;
21     for(j=i; j < tam; j++)
22       if(a[j].chave < a[min].chave) min = j;
23     x = a[min]; a[min] = a[i]; a[i] = x;
24   }
25 }
26
27 double rand0a1(){
28   double res = (double)rand() / INT_MAX;
29   if(res > 1.0) res = 1.0;
30   return res;
31 }
32
33 void permuta(VETOR a, int tam){
34   ITEM b;
```

```
35    int i,j;
36
37       for(i=tam-1; i>=0; i--){
38       j = (i * rand0a1()) + 1;
39       b = a[i];
40       a[i] = a[j];
41       a[j] = b;
42    }
43 }
44
45 void imprime(VETOR a, int tam){
46    int i;
47
48    for(i=0; i<tam; i++) printf("%d ", a[i].chave);
49    printf("\n");
50 }
51
52 int main(int argc, char *argv[]){
53   VETOR v;
54    int n = sizeof(v)/sizeof(ITEM);
55    int i;
56
57    for(i=0; i<n; i++) v[i].chave = i;
58
59    permuta(v,n);
60
61    printf("Original\n");
62    imprime(v,n);
63
64    selecao(v,n);
65
66    printf("Ordenado\n");
67    imprime(v,n);
68
69    return 0;
70
71 }
```

- Observações:

  1. As linhas 5–13 definem os tipos de dados necessários. A coleção a ser ordenada é um vetor de itens (linha 13). Um item é uma estrutura (linhas 8–11) que possui uma chave (linha 9) e, possivelmente, outros componentes que não interferem no processo de ordenação (linha 10).
  2. O método está implementado na função *selecao* (linhas 15–25) e basicamente consiste de dois laços aninhados. O primeiro (linha 19) seleciona o sub-vetor a pesquisar

e o segundo (linhas 21–22) procura o menor elemento desse sub-vetor. O elemento encontrado é trocado com o primeiro item do sub-vetor (linha 23).

3. As funções *rand01* (linhas 27–31), *permuta* (linhas 33–43) e *imprime* (linhas 45–50) são auxiliares. As duas primeiras tomam um vetor e embaralham seus elementos; a última imprime o conteúdo de um vetor.
4. O programa principal (linhas 52–71) declara um vetor de itens (linha 53), preenche o mesmo com chaves em ordem crescente (linha 57) e depois embaralha e imprime seus elementos (linha 59–62). Após a ordenação (linha 64), o vetor é novamente impresso (linha 66–67).

- Características da ordenação por seleção:

1. Análise: os números de comparações ($C(n)$) e movimentações ($M(n)$) de registros, para uma coleção de n itens, são:
   $C(n)= \frac{n^2}{2} - \frac{n}{2}$
   $M(n) = 3(n-1)$
   Ou seja, $C(n) = O(n^2)$ e $M(n) = O(n)$. A ordem inicial dos itens não interfere na análise (não há diferença entre pior caso, melhor caso ou caso médio).
2. Vantagens:
   a) É um método simples ($O(n^2)$), ideal para coleções com poucos registros.
   b) O método é linear ($O(n)$) no número de movimentações, devendo ser escolhido para coleções com registros muito grandes.
3. Desvantagens:
   a) Não é muito eficiente para coleções grandes.
   b) O método é *instável*.

Ordenação por Inserção

- A ordenação por inserção é um método simples que trabalha de acordo com o seguinte princípio: para cada item a partir do segundo, selecione-o e coloque-o no lugar correto no sub-vetor esquerdo (já ordenado), deslocando os elementos necessários para a direita.
- O programa abaixo mostra uma implementação desse método:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <limits.h>
4
5  #define MAXTAM  20
6
7  typedef long CHAVE;
8  typedef struct ITEM_TAG {
9    CHAVE chave;
10   /* Outros componentes */
11 } ITEM;
12 typedef int INDICE;
```

```
13 typedef ITEM VETOR[MAXTAM+1]; /* Espaco extra p/ sentinela */
14
15 void insercao(ITEM *a, int tam){
16   INDICE i,j;
17   ITEM x;
18
19   for(i=1; i < tam; i++){
20     x = a[i];
21     j = i-1;
22     a[0] = x;  /*sentinela*/
23     while(x.chave < a[j].chave){
24       a[j+1] = a[j];
25       j--;
26     }
27     a[j+1] = x;
28   }
29 }
30
31 double rand0a1(){
32   double res = (double)rand() / INT_MAX;
33   if(res > 1.0) res = 1.0;
34   return res;
35 }
36
37 void permuta(VETOR a, int tam){
38   ITEM b;
39   int i,j;
40
41     for(i=tam-1; i>=1; i--){
42       j = (i * rand0a1()) + 1;
43       b = a[i];
44       a[i] = a[j];
45       a[j] = b;
46     }
47 }
48
49 void imprime(VETOR a, int tam){
50   int i;
51
52   for(i=1; i<tam; i++) printf("%d ", a[i].chave);
53   printf("\n");
54 }
55
56 int main(int argc, char *argv[]){
57   VETOR v;
58   int n = sizeof(v)/sizeof(ITEM);
59   int i;
```

```
60
61      for(i=1; i<n; i++) v[i].chave = i;
62
63      permuta(v,n);
64
65      printf("Original\n");
66      imprime(v,n);
67
68      insercao(v,n);
69
70      printf("Ordenado\n");
71      imprime(v,n);
72
73      return 0;
74  }
```

- Observações:

  1. A definição de tipos (linhas 5–13) é análoga à da ordenação por seleção (p. 68). Reparar que o vetor tem uma posição extra (linha 13) para o elemento *sentinela* (ver a seguir).
  2. O método está implementado na função *insercao* (linhas 15–29) e consiste de dois laços aninhados. O primeiro (linha 19) obtém o elemento a inserir (linha 20) e o segundo (linhas 23–26) procura a posição de inserção, deslocando os elementos maiores para a direita. A inserção é feita na linha 27.
  3. O laço mais interno (linhas 23–26) pode ser terminado por duas condições:
     a) Um elemento menor que o pesquisado foi encontrado.
     b) Os elementos do sub-vetor esquerdo esgotaram-se (o final do mesmo foi atingido à esquerda). Isto indica que o elemento pesquisado é o menor até então.

     Para evitar o teste duplo na condição de terminação do laço (linha 23), utiliza-se a primeira posição do vetor (índice 0) para guardar uma cópia do elemento pesquisado (linha 22). Esta cópia é denominada *sentinela*. Assim, a condição de terminação é simplificada tal como mostrado na linha 23.
  4. As funções auxiliares *rand01* (linhas 31–35), *permuta* (linhas 37–47) e *imprime* (linhas 49–54) são análogas às do método de inserção (p. 69).
  5. A função principal (linhas 56–74) é análoga à do método de seleção (p. 69). Reparar, entretanto que os elementos da coleção utilizam o vetor a partir da segunda posição (linha 61), uma vez que a posição inicial é reservada para a sentinela. As funções *permuta* e *imprime* foram também alteradas em função disto (linhas 41 e 52, respectivamente).

- Características da ordenação por inserção:

  1. Análise: quantidade de comparações ($C(n)$) e movimentações ($M(n)$) de registros:
     a) Melhor caso: $C(n) = n - 1 = O(n)$; $M(n) = 3(n-1) = O(n)$.

b) Pior caso: $C(n) = \frac{n^2}{2} + \frac{n}{2} - 1 = O(n^2)$; $M(n) = \frac{n^2}{2} + \frac{5n}{2} - 3 = O(n^2)$.

c) Caso médio: $C(n) = \frac{n^2}{4} + \frac{3n}{4} - 1 = O(n^2)$; $M(n) = \frac{n^2}{4} + \frac{11n}{4} - 3 = O(n^2)$.

O melhor caso corresponde à situação em que os itens já estão previamente ordenados, e o pior àquela em que estão em ordem inversa.

2. Vantagens:
   a) O método é estável.
   b) É o melhor método simples ($O(n^2)$) no caso geral.
   c) É muito eficiente caso os elementos da coleção já estejam quase ordenados.
3. Desvantagens:
   a) Não é muito eficiente para coleções grandes.
   b) O número de movimentações é também quadrático ($O(n^2)$), não trazendo as vantagens do método de seleção para registros grandes.

Quicksort

- *Quicksort* é o algoritmo de ordenação interna mais rápido que se conhece para a maioria das situações. É um método eficiente, ou seja, o número de comparações efetuadas é $O(n \lg n)$.
- Princípio básico: para ordenar um conjunto de n elementos:
  1. Repartir a coleção em duas coleções menores.
  2. Ordenar independentemente cada uma das coleções resultantes.
  3. Combinar os resultados.
- O primeiro passo é denominado *particionamento,* e é a parte mais crítica do método. Ele pode ser descrito mais detalhadamente da seguinte forma: para particionar o vetor A[esq..dir], escolhe-se arbitrariamente um valor x denominado *pivô*. Ao final do processo, o vetor deverá estar dividido em:
  - Parte esquerda: todos os elementos têm chaves menores ou iguais a x.
  - Parte direita: todos os elementos têm chaves maiores ou iguais a x.
- Algoritmo do particionamento:
  1. Escolher arbitrariamente o pivô x.
  2. Percorrer o vetor a partir de *esq* até encontrar um item A[i] $\geq x$; percorrer o vetor a partir de *dir* até encontrar um item A[j] $\leq x$.
  3. Trocar A[i] com A[j].
  4. Continuar até que i e j se cruzem no vetor.
- Ao final: os itens A[esq], A[esq +1], ..., A[j] são todos menores ou iguais a x, e formam a "parte esquerda" mencionada acima. Da mesma forma, a "parte direita" é formada pelos itens A[i], A[i +1], ..., A[dir] (que são todos maiores ou iguais a x).

- Este procedimento é então repetido (em geral, recursivamente) para cada uma das partes, até que, ao final, o vetor esteja completamente ordenado.
- A escolha do pivô pode ser feita de diversas formas. As mais comuns são escolher o elemento central do vetor (utilizada abaixo) ou tomar a média aritmética entre os elementos inicial, central e final.
- O programa abaixo mostra uma implementação desse método:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <limits.h>
4
5  #define MAXTAM  20
6
7  typedef long CHAVE;
8  typedef struct ITEM_TAG {
9    CHAVE chave;
10   /* Outros componentes */
11 } ITEM;
12 typedef int INDICE;
13 typedef ITEM VETOR[MAXTAM];
14
15 void particao(INDICE esq, INDICE dir, INDICE *i, INDICE *j,
16                ITEM *a){
17   ITEM x,w;
18
19   *i = esq; *j = dir;
20   x = a[(*i + *j) / 2];  /*pivo*/
21   do{
22     while(x.chave > a[*i].chave) (*i)++;
23     while(x.chave < a[*j].chave) (*j)--;
24     if(*i <= *j) {
25       w = a[*i]; a[*i] = a[*j]; a[*j] = w;
26       (*i)++; (*j)--;
27     }
28   } while(*i <= *j);
29 }
30
31 void ordena(INDICE esq, INDICE dir, ITEM *a){
32   INDICE i,j;
33
34   particao(esq,dir,&i,&j,a);
35   if(esq < j) ordena(esq,j,a);
36   if(i < dir) ordena(i,dir,a);
37 }
38
39 void quicksort(ITEM *a, int tam){
```

```
40    ordena(0,tam-1,a);
41  }
42
43  double rand0a1(){
44    double res = (double)rand() / INT_MAX;
45    if(res > 1.0) res = 1.0;
46    return res;
47  }
48
49  void permuta(VETOR a, int tam){
50    ITEM b;
51    int i,j;
52
53    for(i=tam-1; i>=0; i--){
54      j = (i * rand0a1()) + 1;
55      b = a[i];
56      a[i] = a[j];
57      a[j] = b;
58    }
59  }
60
61  void imprime(VETOR a, int tam){
62    int i;
63
64    for(i=0; i<tam; i++) printf("%d ", a[i].chave);
65    printf("\n");
66  }
67
68  int main(int argc, char *argv[]){
69    VETOR v;
70    int n = sizeof(v)/sizeof(ITEM);
71    int i;
72
73    for(i=0; i<n; i++) v[i].chave = i;
74
75    permuta(v,n);
76
77    printf("Original\n");
78    imprime(v,n);
79
80    quicksort(v,n);
81
82    printf("Ordenado\n");
83    imprime(v,n);
84
85    return 0;
86  }
```

- Observações:

  1. A definição de tipos (linhas 5–13) é análoga à da ordenação por seleção (p. 68).
  2. O método está implementado nas funções *particao* (linhas 15–29), *ordena* (linhas 31–37) e *quicksort* (linhas 39–41).
  3. A função *particao* (linhas 15–29) implementa o algoritmo de particionamento descrito acima. Reparar na escolha do pivô na linha 20.
  4. A função *ordena* (linhas 31–37) chama a função partição (linha 34) e depois se chama recursivamente para ordenar as partes esquerda (linha 35) e direita (linha 36).
  5. A função *quicksort* (linhas 39–41) faz apenas a chamada inicial da função *ordena* (linha 40), passando para a mesma o vetor completo.
  6. As funções *rand01* (linhas 43–47), permuta (linhas 49–59) e imprime (linhas 61–66) são análogas às do método de inserção (p. 69).
  7. A função principal (linhas 68–86) é análoga à do método de seleção (p. 69).

- Características do *quicksort*:

  1. À medida que as partições se tornam muito pequenas, pode ser vantajoso ordená-las não por novos particionamentos, mas utilizando um dos métodos simples (por exemplo, seleção ou inserção).
  2. Análise: quantidade de comparações de registros ($C(n)$):
     a) Melhor caso: $C(n) = n \lg n - n + 1 = O(n \lg n)$ (os pivôs são escolhidos de modo que cada partição divide a coleção em duas partes iguais).
     b) Pior caso: $C(n) = O(n^2)$ (os pivôs são escolhidos como um dos extremos da coleção).
     c) Caso médio: $C(n) \approx 1.386n \lg n - 0.846n = O(n \lg n)$.
  3. Vantagens:
     a) É extremamente eficiente (o mais eficiente para situações não específicas).
     b) Necessita de pouca memória auxiliar.
  4. Desvantagens:
     a) A escolha do pivô é crítica. No pior caso, são necessárias $O(n^2)$ comparações.
     b) O método é instável.

13 Árvores Binárias

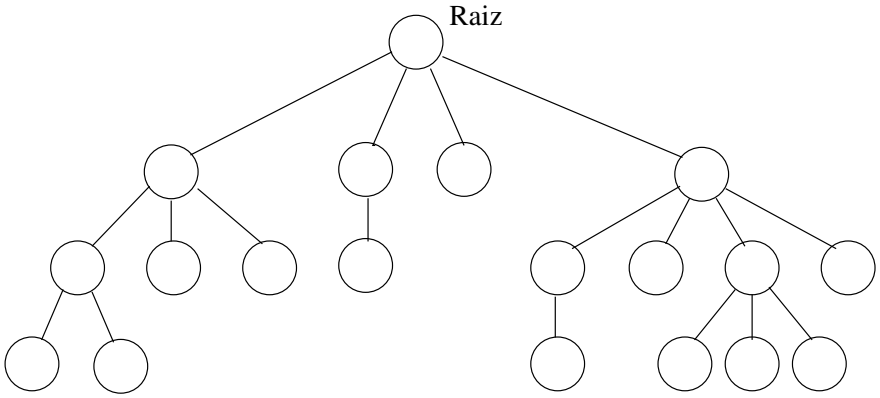
Leitura Recomendada

Ziviani seção 5.3.1: Árvores Binárias de Pesquisa Sem Balanceamento.
Celes cap. 13: Árvores.

Notas

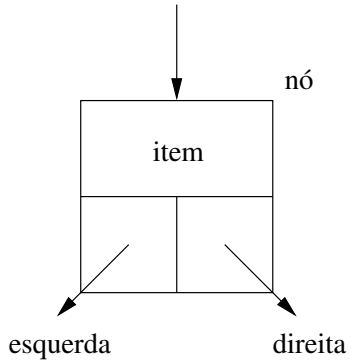
Definições

- Uma árvore é uma estrutura com um número finito de células (nós), consistindo de um nó denominado *raiz* mais 0 a n sub-árvores distintas. Exemplo de árvore com $n = 4$:

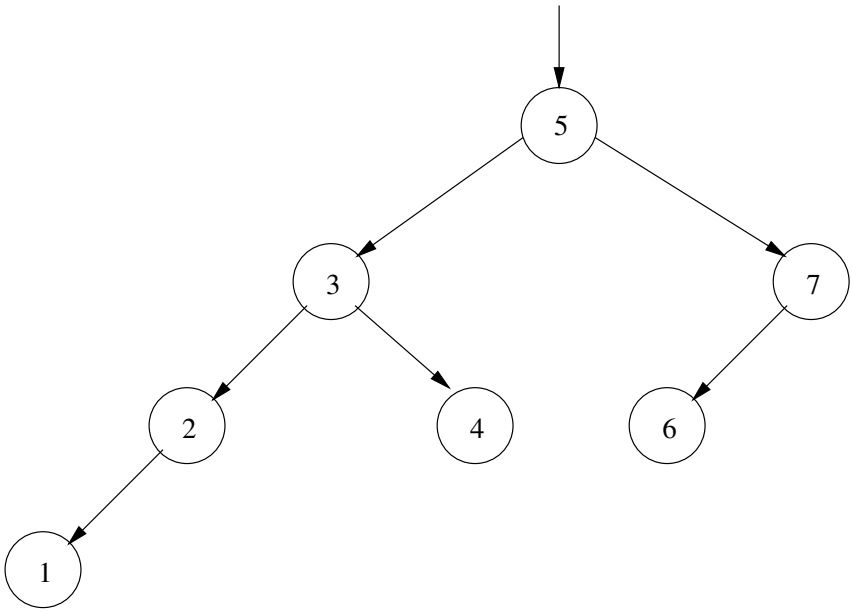


- Em uma *árvore binária*, tem-se $n = 2$ (ou seja, cada nó tem, no máximo, duas sub-árvores).
- Definições:
  - *Grau* de um nó: número de sub-árvores do nó. A árvore binária tem nós de grau 0, 1 e 2.
  - *Nó externo* ou *folha*: nó de grau 0.
  - *Nó interno*: nó de grau superior a 0.
  - *Nível*: O nível da raiz é 0. Se um nó está no nível i, as raízes de suas sub-árvores estão no nível $i + 1$.
  - A *altura* de um nó é o comprimento do caminho mais longo deste nó até uma folha (ou seja, o número de nós até a folha, contando todos os nós envolvidos).
  - A altura da árvore é a altura da raiz.
- As sub-árvores de um nó de uma árvore binária são denominadas sub-árvores *esquerda* e *direita*.

- Um nó da árvore binária contém:
  - Um *item* contendo os dados do nó. Este item em geral contém, dentre outras coisas, uma *chave* que é utilizada para realizar comparações com os itens dos demais nós.
  - Ponteiros para as sub-árvores esquerda e direita.



- Uma *árvore binária de pesquisa* é uma árvore binária onde, para cada nó, vale a seguinte propriedade: todos os itens com chaves menores estão na sub-árvore esquerda e todos os itens com chaves maiores estão na sub-árvore direita. Exemplo:



- Aplicações: uma árvore binária de pesquisa pode ser usada em aplicações que exijam estruturas de dados com as seguintes características:
  1. Acesso direto rápido (pesquisa eficiente).
  2. Acesso seqüencial rápido (percorrimento eficiente).
  3. Facilidade de inserções e remoções.

- Operações do tipo de dados abstrato *árvore*:
  1. *cria (árvore)*: cria uma *árvore* vazia.

2. *pesquisa (item, árvore)*: verifica se a *árvore* um determinado *item*. Se o mesmo for encontrado, retorna seu conteúdo.
3. *insere (item, árvore)* : insere um *item* em uma *árvore* se o mesmo já não existir.
4. *retira (item, árvore)*: retira um *item* de uma *árvore*, se o mesmo existir.
5. *percorre (árvore)*: percorre todos os elementos de uma *árvore*, executando sobre os mesmos algum tipo de operação.

- Observação: o número de operações para pesquisa em uma árvore binária é $O(\log n)$, em comparação com uma lista linear, onde esta operação é $O(n)$.

Implementação

- A estrutura dos tipos de dados necessários para a implementação de uma árvore binária de pesquisa está mostrada a seguir (arquivo *arvore.h*):

```
1  #if !defined(ARVORE_H)
2  #define ARVORE_H
3
4  typedef struct {
5     int chave;
6     /* Outros componentes */
7  } ITEM;
8
9  typedef struct NO_TAG *PONT;
10
11 typedef struct NO_TAG {
12    ITEM item;
13    PONT esq, dir;
14 } NO;
15
16 typedef PONT ARVORE;
17
18 void cria         (ARVORE*);
19 int  pesquisa     (ITEM*, PONT*);
20 int  insere       (ITEM, PONT*);
21 int  retira       (ITEM, PONT*);
22 void percorreIn   (PONT p);
23 void percorrePre  (PONT p);
24 void percorrePos  (PONT p);
25
26 #endif /* ARVORE_H */
```

- A implementação das operações é feita mais facilmente utilizando procedimentos recursivos, uma vez que a estrutura das árvores é definida recursivamente. Em geral, isto é feito da seguinte forma: se a árvore não for vazia,

  – atuar sobre o nó;

– atuar recursivamente sobre as sub-árvores.

A implementação das operações é mostrada na listagem abaixo (arquivo *arvore.c*):

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include "arvore.h"
4
5  void cria (ARVORE *arv){
6     *arv = NULL;
7  }
8
9  int pesquisa (ITEM *x, PONT *p){
10    if (*p == NULL)
11       return -1;
12
13    if (x->chave < (*p)->item.chave)
14       return pesquisa (x, &(*p)->esq);
15    if (x->chave > (*p)->item.chave)
16       return pesquisa (x, &(*p)->dir);
17
18    *x = (*p)->item;
19
20    return 0;
21 }
22
23 int insere (ITEM x, PONT *p){
24    if (*p == NULL) {
25       *p = (PONT) malloc (sizeof (NO));
26       (*p)->item = x;
27       (*p)->esq  = NULL;
28       (*p)->dir  = NULL;
29       return 0;
30    }
31
32    if (x.chave < (*p)->item.chave)
33       return insere (x, &(*p)->esq);
34    if (x.chave > (*p)->item.chave)
35       return insere (x, &(*p)->dir);
36
37    return -1;
38 }
39
40 void antecessor (PONT q, PONT *r){
41    if ((*r)->dir != NULL) {
42       antecessor (q, &(*r)->dir);
43       return;
44    }
```

```
45
46      q->item = (*r)->item;
47      q = *r;
48      *r = (*r)->esq;
49      free (q);
50  }
51
52  int retira (ITEM x, PONT *p){
53     PONT aux;
54
55      if (*p == NULL)
56          return -1;
57
58      if (x.chave < (*p)->item.chave)
59          return retira (x, &(*p)->esq);
60      if (x.chave > (*p)->item.chave)
61          return retira (x, &(*p)->dir);
62
63      if ((*p)->dir == NULL) {
64          aux = *p;
65          *p = (*p)->esq;
66          free (aux);
67          return 0;
68      }
69
70      if ((*p)->esq == NULL) {
71          aux = *p;
72          *p = (*p)->dir;
73          free (aux);
74          return 0;
75      }
76
77      antecessor (*p, &(*p)->esq);
78      return 0;
79  }
80
81  void percorreIn (PONT p){
82      if (p != NULL) {
83          percorreIn (p->esq);
84          printf ("%d\n", p->item.chave);
85          percorreIn (p->dir);
86      }
87  }
88
89  void percorrePre (PONT p){
90      if (p != NULL) {
91          printf ("%d\n", p->item.chave);
```

```
92          percorrePre (p->esq);
93          percorrePre (p->dir);
94      }
95  }
96
97  void percorrePos (PONT p){
98      if (p != NULL) {
99          percorrePos (p->esq);
100         percorrePos (p->dir);
101         printf ("%d\n", p->item.chave);
102     }
103 }
```

- Observações:

  1. A operação de criação da árvore vazia (linhas 5–7) é trivial, uma vez que o tipo de dados *ARVORE* (tal como definido na linha 16 da listagem anterior) é apenas um ponteiro para um nó.
  2. A operação de pesquisa (linhas 9–21) compara a chave do item procurado com a chave do item na raiz da sub-árvore, chamando-se recursivamente para continuar a busca na sub-árvore esquerda (linhas 13–14) ou na sub-árvore direita (linhas 15–16) se o mesmo não for encontrado. Esta função está dividida em três regiões muito claras:
     a) Linhas 10–11: região do "não encontrado": se a condição do *if* for satisfeita, o item procurado não está na árvore e a busca deve ser encerrada, com o retorno de uma condição de erro.
     b) Linhas 13–16: região de pesquisa: o item ainda não foi encontrado, e a busca continua na sub-árvore esquerda ou na direita.
     c) Linhas 18–20: região do "encontrado": o item procurado foi encontrado. O mesmo é devolvido e uma condição de sucesso é retornada.

     Estas regiões aparecem também nas operações de inserção e remoção (descritas a seguir).
  3. A operação de inserção (linhas 23–38) reproduz o procedimento de pesquisa, incluindo o novo elemento ao constatar que o mesmo não existe na árvore. Ou seja:
     a) A região do "não encontrado" (linhas 24–30) efetua a inserção e retorna uma condição de sucesso.
     b) A região de pesquisa (linhas 32–35) é análoga à da operação de pesquisa (linhas 13–16).
     c) A região do "encontrado" (linha 37) retorna uma condição de erro (o item a inserir já existe na árvore).
  4. A operação de remoção (linhas 52–79) reproduz o procedimento de pesquisa até encontrar o item a retirar:
     a) A região do "não encontrado" (linhas 55–56) retorna uma condição de erro (tentativa de remoção de elemento inexistente).

b) A região de pesquisa (linhas 58–61) é análoga à da operação de pesquisa (linhas 13–16).

c) A região do "encontrado" (linhas 63–78) remove o item desejado e retorna uma condição de sucesso (linha 78). Há dois casos a considerar:

i. O nó a ser retirado tem no máximo uma sub-árvore. Neste caso, o ponteiro do nó antecessor deve apontar para essa sub-árvore (que poderá estar também vazia, no caso da remoção de folha). Nas linhas 63–68 verifica-se que o nó a remover não tem a sub-árvore direita, e nas linhas 70–75 que o mesmo não tem a sub-árvore esquerda. Se nenhuma das condições dos *if*'s das linhas 63 e 70 for verificada, o nó a remover tem ambas as sub-árvores não vazias, caso tratado a seguir.

ii. O nó a ser retirado tem duas sub-árvores. O mesmo deverá ser trocado com o maior nó de sua sub-árvore esquerda (ou seja, o nó mais à direita de sua sub-árvore esquerda)[1] e este último será removido de forma análoga ao que foi descrito acima. A tarefa de remoção é delegada à função auxiliar *antecessor* (chamada na linha 77).

5. A função auxiliar *antecessor* (linhas 40–50) efetua o trabalho descrito acima:

a) Encontra o maior nó da sub-árvore esquerda do nó a remover através de uma busca recursiva (linhas 41–44).

b) Copia esse nó para o lugar do nó a remover (linha 46).

c) Remove esse nó (linha 49), preservando em seu antecessor a sua sub-árvore esquerda (linha 48).

6. As funções de percorrimento (linhas 81–103) percorrem todos os nós da árvore, aplicando sobre os mesmos uma determinada operação (no caso, imprimindo o valor de suas chaves). Este percorrimento pode ser feito em diferentes ordens, três das quais estão mostradas:

a) Percorrimento Central (linhas 81–87): percorre a sub-árvore esquerda (linha 83), aplica a operação sobre a raiz (linha 84) e percorre a sub-árvore direita (linha 85). Este percorrimento visita todos os nós em ordem crescente de chave.

b) Percorrimento em pré-ordem (linhas 89–95): aplica a operação sobre a raiz (linha 91), percorre a sub-árvore esquerda (linha 92) e a sub-árvore direita (linha 93).

c) Percorrimento em pós-ordem (linhas 97–103): percorre a sub-árvore esquerda (linha 99), a sub-árvore direita (linha 100) e aplica a operação sobre a raiz (linha 101).

- A listagem a seguir mostra um programa que exemplifica a utilização do tipo de dados *árvore* descrito acima (arquivo *main.c*):

```
1 #include <stdio.h>
2 #include "arvore.h"
3
4 int main (void){
```

[1] Esta escolha é arbitrária. Poder-se-ia trabalhar também com o menor nó da sub-árvore direita (o nó mais à esquerda da sub-árvore direita).

```
5     FILE *arq;
6     ARVORE arv;
7     ITEM item;
8     int i;
9
10    cria (&arv);
11    arq = fopen ("arq.txt","r");
12    while (fscanf (arq, "%d", &i) != EOF) {
13       item.chave = i;
14       if (insere (item, &arv) == -1)
15          printf ("Erro na insercao de %d\n", i);
16    }
17
18    printf ("Pre ordem:\n");
19    percorrePre (arv);
20
21    printf ("Pos ordem:\n");
22    percorrePos (arv);
23
24    printf ("In ordem:\n");
25    percorreIn (arv);
26
27    while (1) {
28       printf ("Favor informar o item a retirar: ");
29       scanf ("%d", &item.chave);
30
31       if (item.chave == -1)
32          break;
33
34       if (retira (item, &arv) == -1)
35          printf ("Elemento nao existe na arvore\n");
36       else {
37          printf ("Elemento retirado. Arvore:\n");
38          percorreIn (arv);
39       }
40    }
41
42    return 0;
43 }
```

14 Tabelas de Espalhamento

Leitura Recomendada

Ziviani seção 5.5: Transformação de Chave (*Hashing*).
Celes cap. 18: Tabelas de Dispersão.

Notas

Espalhamento (*hashing*) é um método de pesquisa em que os registros armazenados em uma tabela são *diretamente* acessados através de uma função aritmética aplicada à chave de pesquisa. Etapas do método:

1. Cálculo da função de espalhamento: transforma a chave de pesquisa em um endereço da tabela.
2. Resolução de colisões: uma colisão ocorre quando duas ou mais chaves são transformadas para o mesmo endereço; essas colisões devem ser tratadas de alguma forma.

Colisões

- Se a quantidade de chaves pudesse ser igual (ou menor) à quantidade de entradas na tabela, cada registro seria mapeado para a sua própria posição e não ocorreriam colisões.
- Se o número de chaves possíveis é maior que o tamanho da tabela, o mapeamento não pode ser de um para um, logo diferentes chaves são mapeadas para a mesma posição; ou seja, ocorrem colisões.

Função de Espalhamento

- A função de espalhamento (h) deve mapear chaves em inteiros no intervalo $[0, M-1]$, onde M é o tamanho da tabela.
- Características desejáveis:
  1. Fácil de calcular.
  2. Saídas igualmente prováveis para cada chave de entrada.
- Cálculo:
  - Transformar chaves não numéricas em inteiros:

$$\text{chave} \rightarrow k$$

Exemplo: transformar *strings* em inteiros somando os códigos ASCII dos caracteres da *string*.

- Calcular a função $k \longmapsto h(k)$. Exemplo: $h(k) = k \bmod M$.
- Observações:
  1. M deve ser primo.
  2. Devem ser evitados primos da forma $b^i \pm j$, onde b é a base do conjunto de caracteres (128 para ASCII) e i e j são inteiros pequenos.

Outros Exemplos de Funções de Espalhamento

- Os exemplos abaixo envolvem cálculos mais complexos que a função acima, porém têm a vantagem de não exigir que a tabela tenha um número primo de posições.
- Método do quadrado: após a transformação da chave em inteiro, o valor calculado é multiplicado por si mesmo e alguns algarismos (do início e/ou fim) do resultado são desprezados. Por exemplo, na tabela abaixo são desprezados os três primeiros e os três últimos algarismos:

Chave	$Chave^2$	*Endereço*
897	00804609	04
5315	28249225	49
6000	36000000	00
7800	60840000	40

- Método da multiplicação: este método utiliza a seguinte fórmula para o cálculo do endereço E a partir de uma chave k (já previamente convertida em inteiro):

$$E = \left\lfloor m \left(k\phi^{-1} - \left\lfloor k\phi^{-1} \right\rfloor \right) \right\rfloor$$

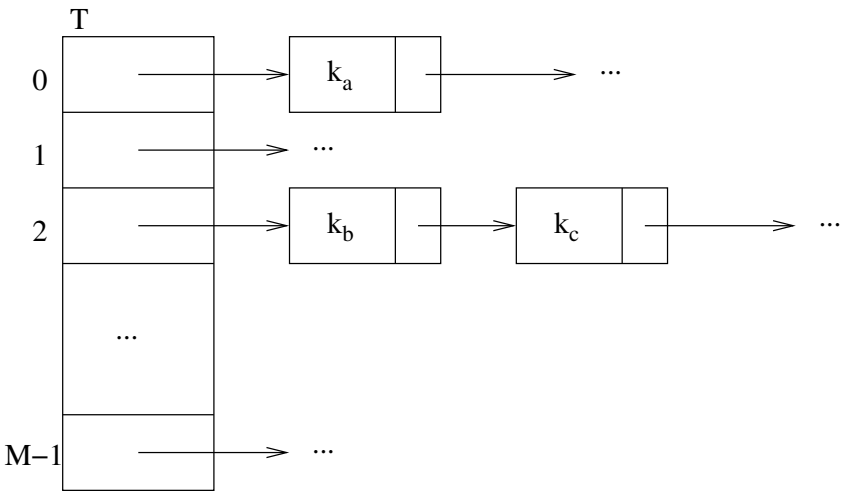
onde m é o tamanho da tabela e ϕ é a *razão áurea:* $\phi = \frac{\sqrt{5}-1}{2} = 0.6180399$. Para $m = 10$, a tabela abaixo traz um exemplo para os vinte primeiros valores de k:

k	$\lfloor 10 \left(k\phi^{-1} - \lfloor k\phi^{-1} \rfloor \right) \rfloor$
1	6
2	2
3	8
4	4
5	0
6	7
7	3
8	9
9	5
10	1
11	7
12	4

13	0
14	6
15	2
16	8
17	5
18	1
19	7
20	3

Resolução de Colisões

1. Listas encadeadas: cada endereço da tabela é o início de uma lista encadeada de itens:



   Análise: o custo das operações *insere, pesquisa* e *retira* é $O(1 + N/M)$ onde N é o número de registros na tabela e M é o tamanho da tabela.

2. Endereçamento aberto: os espaços vazios na tabela são utilizados para a resolução das colisões:

   - Todas os itens são armazenadas na própria tabela.
   - Quando ocorre uma colisão, utiliza-se uma seqüência de alternativas calculadas por $h_1(x)$, $h_2(x)$, etc.
   - Se todas as posições estão ocupadas, a tabela está cheia.
   - Exemplo: espalhamento linear: $h_j(x) = (h(x) + j) \bmod M$, $1 \leq j \leq M - 1$.
   - Deve haver um marcador especial para indicar posições da tabela de onde foram retirados registros.
   - Análise: custo da pesquisa com sucesso:

$$C(n) = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

   - α é chamado *fator de carga* da tabela: $\alpha = N/M$.

- No pior caso, $C(n) = O(n)$, no melhor caso e caso médio, $C(n) = O(1)$.
- A listagem abaixo mostra um exemplo de implementação de espalhamento com endereçamento aberto:

```
1  #include <stdio.h>
2  #include <string.h>
3  #define M          113
4  #define N          20
5  #define VAZIO      "____________________"
6  #define RETIRADO   "********************"
7
8  typedef char CHAVE [N+1];
9  typedef struct {
10    CHAVE chave;
11    int   quantidade;
12 } ITEM;
13
14 typedef ITEM DICIONARIO [M];
15
16 int h (CHAVE chave){
17    int i, soma;
18
19    soma = 0;
20    for (i = 0; i < N; i++)
21       soma += chave[i];
22    return (soma % M);
23 }
24
25 void inicializa (DICIONARIO t) {
26    int i;
27
28    for (i = 0; i < M; i++) {
29       strcpy (t[i].chave, VAZIO);
30    }
31 }
32
33 int pesquisa (CHAVE ch, DICIONARIO t){
34    int i, inicial;
35
36    inicial = h(ch);
37    i = 0;
38    while ((strncmp (t[(inicial+i) % M].chave, VAZIO, N))
39        && (strncmp (t[(inicial+i) % M].chave, ch   , N))
40        && (i < M)) i++;
41    if (!strncmp (t[(inicial + i) % M].chave, ch, N))
42       return ((inicial + i) % M);
43    else
```

```
44          return -1;
45  }
46
47  int insere (ITEM x, DICIONARIO t){
48      int i, inicial;
49
50      inicial = h(x.chave);
51      i = 0;
52      while ((strncmp (t[(inicial+i) % M].chave, VAZIO    , N))
53          && (strncmp (t[(inicial+i) % M].chave, RETIRADO, N))
54          && (i < M)) i++;
55      if (i < M) {
56          t[(inicial+i) % M] = x;
57          return ((inicial+i) % M);
58      }
59      else
60          return -1;
61  }
62
63  int retira (CHAVE ch, DICIONARIO t){
64      int i;
65
66      i = pesquisa (ch, t);
67      if (i != -1) {
68          memcpy (t[i].chave, RETIRADO, sizeof(CHAVE));
69          return 0;
70      }
71      else
72          return -1;
73  }
74
75  int main (void){
76      DICIONARIO dic;
77      CHAVE palavra;
78      ITEM  item;
79      int   rslt;
80
81      inicializa (dic);
82      for (;;) {
83          scanf ("%s", palavra);
84          if (!strcmp (palavra, "fim"))
85              break;
86          rslt = pesquisa (palavra, dic);
87          if (rslt == -1) {
88              memset (item.chave, '␣', N);
89              strncpy (item.chave, palavra, N);
90              item.quantidade = 0;
```

```
91              rslt = insere (item, dic);
92          }
93          if (rslt == -1)
94              printf ("Dicionario cheio\n");
95          else {
96              dic[rslt].quantidade++;
97              printf ("%s: %d\n", dic[rslt].chave,
98                                                  dic[rslt].quantidade);
99          }
100     }
101     return 0;
102 }
```

15 Árvores B

Leitura Recomendada

Ziviani seção 5.5: Transformação de Chave (*Hashing*).
Celes cap. 18: Tabelas de Dispersão.

Notas

- Uma árvore B é uma árvore n-ária com mais de um registro por nó. Pode ser vista como uma extensão de uma árvore binária de pesquisa.
- Em geral, árvores B são utilizadas para pesquisas em memória secundária, com cada nó sendo guardado em uma página no dispositivo de armazenamento.
- Cada nó de uma árvore B de ordem m pode armazenar até $2m$ registros. Estes registros são sempre guardados em ordem crescente da esquerda para a direita.