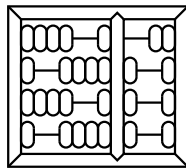


Estruturas de Dados e Técnicas de Programação

Tomasz Kowaltowski

Instituto de Computação
Universidade Estadual de Campinas



www.ic.unicamp.br/~tomasz

Copyright © 2010-2014 Tomasz Kowaltowski

Instituto de Computação

Universidade Estadual de Campinas

Algumas transparências foram adaptadas da apostila *Estruturas de Dados e Técnicas de Programação* de autoria de Cláudio L. Lucchesi e Tomasz Kowaltowski.

Estas transparências somente podem ser copiadas para uso pessoal dos docentes e alunos das disciplinas oferecidas pelo Instituto de Computação da UNICAMP.

Avaliação

Provas e tarefas de laboratório:

Provas:

$$P = (2P_1 + 3P_2 + 5P_3)/10$$

- ▶ P_1 : 23 de outubro
- ▶ P_2 : 27 de novembro
- ▶ P_3 : 18 de dezembro

Estas datas estão sujeitas à confirmação, dependendo dos cancelamentos imprevistos de aulas.

Tarefas: (de 10 a 15 tarefas)

$$L = \frac{\sum_{i=1}^n t_i}{n}$$

onde n é o número de tarefas de laboratório e t_i ($0 \leq t_i \leq 10$) é a nota obtida na tarefa; poderá haver tarefas especiais com nota máxima maior que 10.

Aproveitamento e média final:

Aproveitamento:

$$A = \begin{cases} (7P + 3L)/10 & \text{se } P \geq 5 \text{ e } L \geq 5 \\ \min(P, L) & \text{caso contrário.} \end{cases}$$

Média final:

$$F = \begin{cases} (A + E)/2 & \text{se o aluno fez o exame final} \\ A & \text{caso contrário} \end{cases}$$

onde E é a nota obtida no exame.

Exame final: 15 de janeiro de 2015

Não terão direito ao exame final alunos com média A inferior a 2.5 ou frequência inferior a 75%.

Correção das tarefas de laboratório

- ▶ Nota zero se não produzir resultados corretos em todos os testes (SuSy).
- ▶ Nota zero se não seguir as exigências do enunciado.
- ▶ Nota inicial 7,0 se produzir resultados corretos em todos os testes (exceto caso anterior).
- ▶ Pontos adicionais e descontos:
 - ▶ Correção do algoritmo: -7,0 a 0,0
 - ▶ Eficiência: -2,0 a +1,0
 - ▶ Estruturação de código, facilidade de leitura, indentação, etc: -3,0 a +3,0
 - ▶ Comentários: -1,0 a +1,0
- ▶ Nota máxima: 10,0 (exceto casos anunciados).

Observações:

- ▶ Todas as tarefas de laboratório são individuais.
- ▶ Será estabelecido um limite para o número de submissões de cada tarefa de laboratório (em geral, 10 submissões) – testar antes de submeter!
- ▶ A submissão de uma tarefa de laboratório poderá ser considerada rejeitada se não seguir estritamente as exigências do enunciado, mesmo que produza resultados corretos nos testes.
- ▶ Qualquer tentativa de fraude nas provas ou nas tarefas de laboratório implicará em aproveitamento zero no semestre para todos os envolvidos, sem prejuízo de outras sanções.
- ▶ As transgressões às regras de uso dos sistemas computacionais implicarão em aproveitamento zero no semestre para todos os envolvidos, sem prejuízo de outras sanções.
- ▶ Não haverá provas substitutivas.
- ▶ Todas as provas e o exame final serão realizados sem consulta.

Informações complementares:

- ▶ **Material**
 - ▶ Apostila, transparências, manuais etc:
<http://www.ic.unicamp.br/~tomasz/mc202>
 - ▶ Livros: Biblioteca do IMECC
 - ▶ Gravações das aulas de 2012 (2.o semestre):
<http://weblectures.ic.unicamp.br:8080>
(⇒ Disciplinas ⇒ MC202EF-2012s2).
- ▶ **Sistema SuSy:**
 - ▶ Entrega de tarefas pela Internet
 - ▶ Logins, e-mails e senhas do IC
 - ▶ Tarefa 00: apenas para verificação
 - ▶ Tarefa 01: já está disponível

Introdução


Pré-requisito e objetivos


- ▶ Pré-requisito: curso básico de programação em C (MC102)
- ▶ Objetivos:
 - ▶ Programação em (relativamente) baixo nível
 - ▶ Técnicas de programação e estruturação de dados
 - ▶ Preparação para:
 - ▶ Análise de algoritmos
 - ▶ Programação de sistemas
 - ▶ Programação em geral
 - ▶ Bancos de dados
 - ▶ Engenharia de software
 - ▶ Sistemas embarcados
 - ▶ ...


Programa


- ▶ Introdução à análise de algoritmos
- ▶ Estruturação elementar de dados: matrizes, registros, apontadores
- ▶ Estruturas lineares: pilhas, filas, filas duplas
- ▶ Recursão e retrocesso
- ▶ Árvores binárias: representação, percursos
- ▶ Árvores gerais: representação, percursos
- ▶ Aplicação de árvores:
 - árvores de busca, árvores AVL, árvores B,
 - árvores rubro-negras, filas de prioridade, árvores digitais
- ▶ Listas generalizadas
- ▶ Espalhamento
- ▶ Processamento de cadeias de caracteres
- ▶ Gerenciamento de memória
- ▶ Algoritmos de ordenação
- ▶ Algoritmos em grafos
- ▶ Tipos abstratos de dados e orientação a objetos


Bibliografia

 A. V. Aho, J. E. Hopcroft, and J. Ullman.
Data Structures and Algorithms.
Addison-Wesley, 1983.

 A. Drozdek.
Estrutura de Dados e Algoritmos em C++.
Thomson, 2002.


 J. L. Szwarcfiter e L. Markenzon.
Estruturas de Dados e seus Algoritmos.
LTC Editora, 1994.


 C. L. Lucchesi e T. Kowaltowski.
Estruturas de Dados e Técnicas de Programação.
Instituto de Computação – UNICAMP, 2003.


 P. Feofiloff.
Algoritmos em Linguagem C.
Elsevier Editora Ltda., 2009.


 G. H. Gonnet.
Handbook of Algorithms and Data Structures.
Addison-Wesley, 1984.


 E. Horowitz and S. Sahni.
Fundamentals of Data Structures in Pascal.
Computer Science Press, 1984.

 D. E. Knuth.
The Art of Computer Programming, volume I: Fundamental Algorithms.
Addison-Wesley, 1978.


 E. M. Reingold and W. J. Hanson.
Data Structures.
Little Brown and Company, 1983.

 R. Sedgewick.
Algorithms in C.
Addison-Wesley, 1990.

 D. F. Stubbs and N. W. Webre.
Data Structures with Abstract Data Types and Pascal.
Brooks/Cole, 1985.

 A. M. Tenenbaum, Y. Langsam, and M. J. Augenstein.
Data Structures using C.
Prentice-Hall, 1990.

 N. Wirth.
Algorithms + Data Structures = Programs.
Prentice-Hall, 1976.

 N. Ziviani.
Projeto de Algoritmos (2a. ed.)
Thomson, 2004.

Noções de Análise de Algoritmos

Escolha da estrutura: exemplo

Importância da escolha de estrutura de dados – acesso ao k -ésimo elemento numa sequência:

```
...
x = a[k];
...
```

(a) vetor

```
...
p = a; i = 0;
while (i < k) {
    p = p->prox;
    i++;
}
x = p->info;
...
```

(b) lista ligada

(a) Número de operações constante.

(b) Número de operações proporcional a k .

Exemplo de análise de trechos de programas

```
...
x = a+b;
...
```

(a)

```
...
for (i=0; i<n; i++)
    x = a+b;
...
```

(b)

```
...
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        x = a+b;
...
```

(c)

	a	b	c
análise simples (1)	1	n	n^2
análise detalhada (2)	2	$5n + 2$	$5n^2 + 5n + 2$

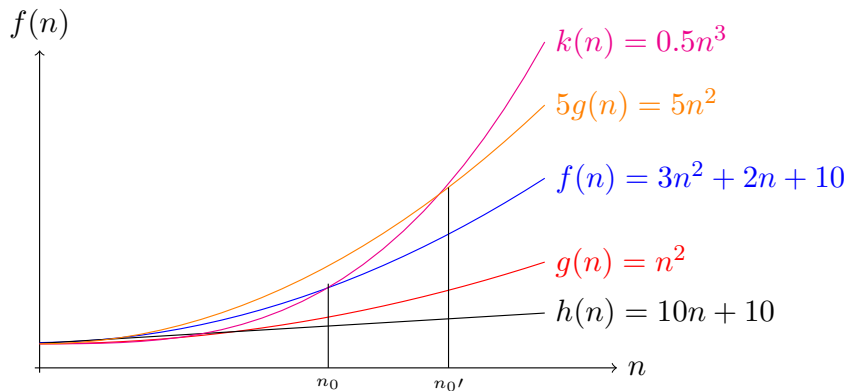
(1): atribuições (2): atribuições, operações aritméticas e comparações

As duas análises produzem resultados proporcionais para valores crescentes de n .

Notação $O()$

$f(n) = O(g(n))$ se existem n_0 e k com $f(n) \leq k * g(n)$ para todo $n > n_0$.

Em outras palavras, o crescimento de $g(n)$ *domina* o crescimento de $f(n)$ acima de n_0 .

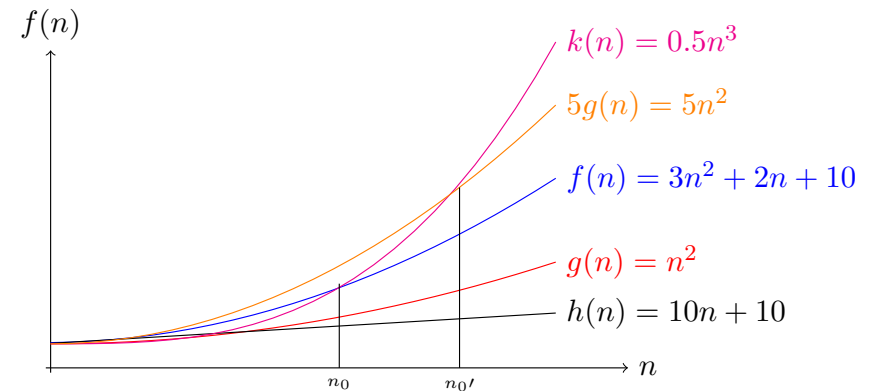


$h(n) = O(g(n))$, $g(n) = O(f(n))$, $f(n) = O(g(n))$, $f(n) = O(k(n))$

Notação $\Omega()$

$f(n) = \Omega(g(n))$ se $g(n) = O(f(n))$.

Em outras palavras, o crescimento de $f(n)$ *domina* o crescimento $g(n)$ acima de n_0 .

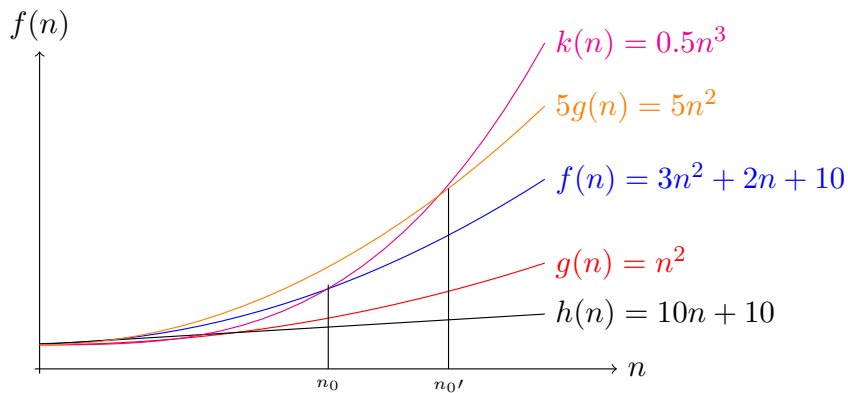


$g(n) = \Omega(h(n))$, $f(n) = \Omega(g(n))$, $g(n) = \Omega(f(n))$, $k(n) = \Omega(f(n))$

Notação $\Theta()$

$f(n) = \Theta(g(n))$ se $f(n) = O(g(n))$ e $g(n) = \Omega(f(n))$.

Em outras palavras, os crescimentos de $f(n)$ e de $g(n)$ acima de n_0 são aproximadamente proporcionais.



$f(n) = \Theta(g(n))$, $g(n) = \Theta(f(n))$, $h(n) = \Theta(n)$

Exemplos de notação $O()$

Na prática, inclusive nestas transparências, $O()$ é frequentemente usado em lugar de $\Theta()$, apesar de denotar o limite superior para o crescimento de uma função.

$$c = O(1) \quad \text{para qualquer constante } c$$

$$2 = O(1)$$

$$5n + 2 = O(n)$$

$$5n^2 + 5n + 2 = O(n^2)$$

$$n^2 = O(n^3)$$

$$n^k = O(n^{k+1}), \quad k \geq 0$$

$$\log_a n = O(\log_b n), \quad a, b > 0$$

$$\log_2 n = O(\log_{10} n)$$

Exemplo de análise de um procedimento de ordenação

```
void Ordena(int v[], int n) {
    int i, k, m, t;
    for (i=0; i<n-1; i++) {
        m = i;
        for (k=i+1; k<n; k++)
            if (v[k]<v[m]) m = k;
        t = v[i]; v[i] = v[m]; v[m] = t;
    }
} /* Ordena */
```

O número de comparações de elementos de v , para cada valor de i , é $n - i - 1$. O número total de comparações será:

$$\sum_{i=0}^{n-2} (n - i - 1) = \sum_{k=n-1}^1 k = \frac{n^2}{2} - \frac{n}{2}$$

ou seja, o número de comparações é da ordem de $O(n^2)$ (mais exatamente $\Theta(n^2)$).

Crescimento de algumas funções

n	$\log_2 n$	$n \log_2 n$	n	n^2	n^3	2^n
1	0	0	1	1	1	2
2	1	2	2	4	8	4
4	2	8	4	16	64	16
8	3	24	8	64	512	256
16	4	64	16	256	4.096	65.536
32	5	160	32	1.024	32.768	4.294.967.296
64	6	384	64	4.096	262.144	$\approx 18 \times 10^{18}$
128	7	896	128	16.384	2.097.152	$\approx 34 \times 10^{37}$

Terminologia:

$O(\log n)$: crescimento logarítmico

$O(n \log n)$: crescimento $n \log n$

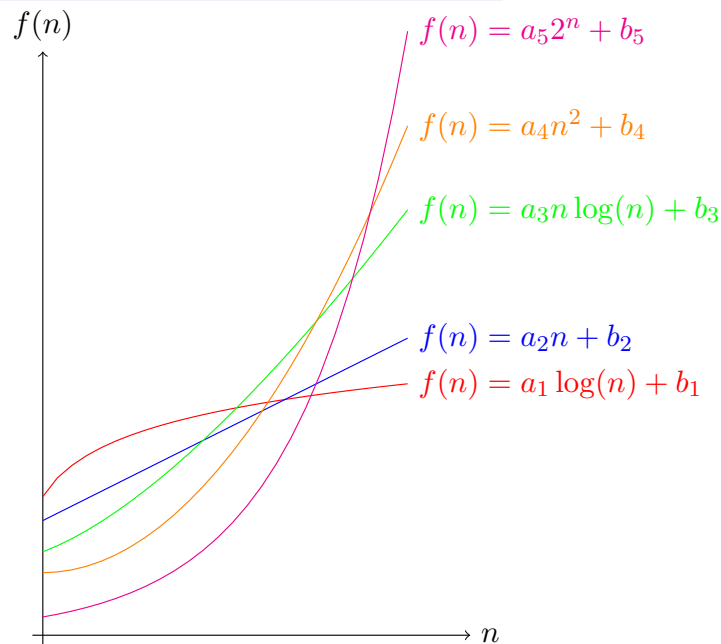
$O(n)$: crescimento linear

$O(n^2)$: crescimento quadrático

$O(n^3)$: crescimento cúbico

$O(2^n)$: crescimento exponencial

Comparação de crescimento das funções



Exemplo

- Suponhamos duas máquinas:
 - M_R : muito rápida
 - M_L : muito lenta (100 vezes)
- As máquinas executam programas de busca num vetor ordenado usando algoritmos distintos:
 - M_R : busca sequencial (n comparações no pior caso)
 - M_L : busca binária ($\log_2 n$ comparações no pior caso)
- A tabela da transparência seguinte poderia ser obtida através de medidas experimentais de tempo de execução para vetores de diversos tamanhos, usando alguma unidade de tempo conveniente.

Exemplo (cont.)

n	$\log_2 n$	M_R (busca sequencial – $O(n)$)	M_L (busca binária – $O(\log n)$)
16	4	16	400
32	5	32	500
64	6	64	600
128	7	128	700
256	8	256	800
512	9	512	900
1024	10	1024	1000
2048	11	2048	1100
4096	12	4096	1200
...
2^{20}	20	1.048.576	2000
2^{21}	21	2.097.152	2100
...
2^{30}	30	1.073.741.824	3000

Supondo que a unidade seja 1ms (isto é, 10^{-3} segundo), 1.048.576ms correspondem a 17 minutos e 1.073.741.824ms equivalem a cerca de 5 horas. No caso da máquina lenta, mas com busca binária, seriam 2 e 3 segundos, respectivamente.

Execução de programas

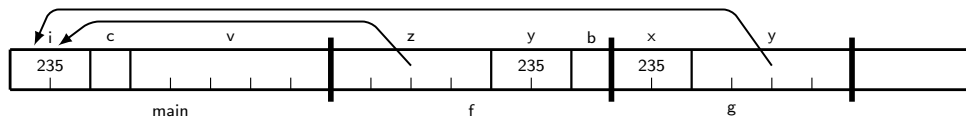
Exemplo de funções simples

```
void g(int x, int *y) {
    *y = x;
} /* g */

void f(int *z) {
    int y; char b;
    y = 235;
    g(y, z);
} /* f */
```

```
int main() {
    int i; char c;
    char v[5];
    f(&i);
    return 0;
} /* main */
```

Pilha de execução (supõe inteiros de dois bytes):



Obs.: Na realidade, os inteiros são armazenados sob a forma binária.

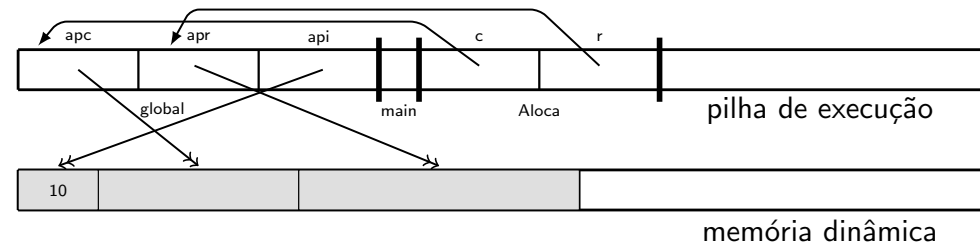
Exemplo de funções com alocação dinâmica

```
typedef char Cadeia[5];
typedef Cadeia *ApCadeia;
typedef struct {
    Cadeia nome;
    int idade;
} Reg, *ApReg;

void Aloca(ApCadeia *c, ApReg *r) {
    *c = malloc(sizeof(Cadeia));
    *r = malloc(sizeof(Reg));
} /* Aloca */
```

```
ApCadeia apc;
ApReg apr;
int *api;

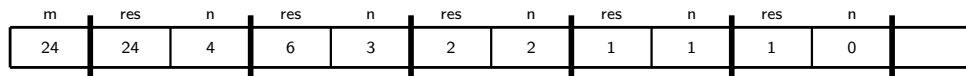
int main() {
    Aloca(&apc, &apr); free(apc);
    free(apr); free(api);
    return 0;
} /* main */
```



Exemplo de função recursiva

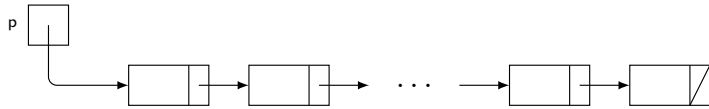
```
int main() {
    int m;
    m = fat(4);
    return 0;
} /* main */
```

```
int fat(int n) {
    if (n==0)
        return 1;
    else
        return n*fat(n-1);
} /* fat */
```



Estruturas ligadas

Listas ligadas simples

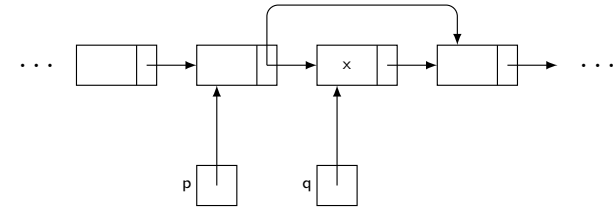


Declarações (equivalentes):

```
typedef
    struct RegLista *Lista;
typedef
    struct RegLista {
        T info;
        Lista prox;
    } RegLista;
```

```
typedef
    struct RegLista {
        T info;
        struct RegLista *prox;
    } RegLista, *Lista;
```

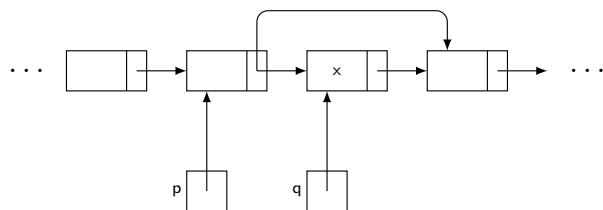
Inserção e remoção com passagem por valor



```
void Insere(Lista p, T x) {
    Lista q =
        malloc(sizeof(RegLista));
    q->info = x;
    q->prox = p->prox;
    p->prox = q;
}
```

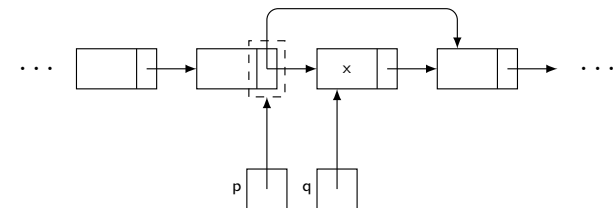
```
void Remove(Lista p, T *x) {
    Lista q = p->prox;
    *x = q->info;
    p->prox = q->prox;
    free(q);
}
```

Inserção e remoção com passagem por valor (cont.)



- ▶ O argumento p é o apontador para o predecessor do nó a ser inserido ou removido.
- ▶ A função 'Remove' não pode remover um nó que é o único da lista.
- ▶ A função 'Insere' não pode inserir um nó no início da lista, inclusive se ela for vazia.

Inserção e remoção com passagem por referência

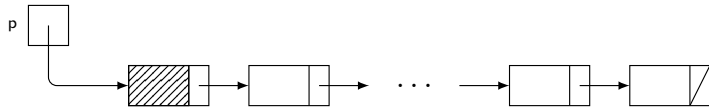


```
void Insere(Lista *p, T x) {
    Lista q =
        malloc(sizeof(RegLista));
    q->info = x;
    q->prox = *p;
    *p = q;
}
```

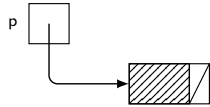
```
void Remove(Lista *p, T *x) {
    Lista q = *p;
    *x = q->info;
    *p = q->prox;
    free(q);
}
```

Esta convenção elimina os problemas da passagem por valor. Note-se que as variáveis p e q têm tipos diferentes.

Lista simples com nó cabeça

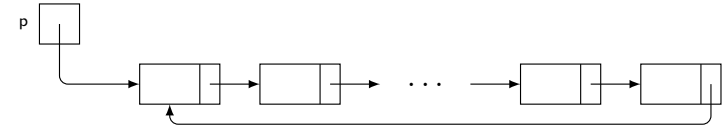


Lista vazia:



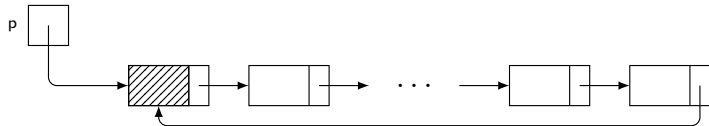
Esta convenção permite o uso de passagem por valor nas funções básicas. O campo de informação do nó cabeça pode ser aproveitado para guardar alguma informação adicional (por exemplo, o comprimento da lista).

Lista simples circular

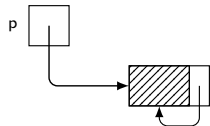


Problema: lista vazia?

Lista circular com nó cabeça



Lista vazia:

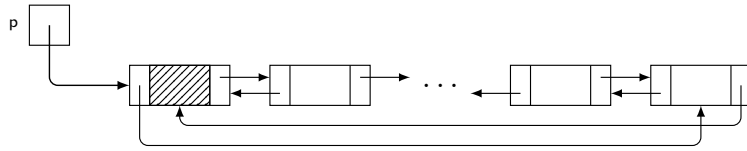


Busca em lista circular com nó cabeça – sentinelas

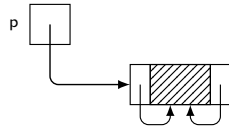
```
Lista BuscaCircular(
    Lista p, T x) {
    /* Busca sem sentinela */
    Lista q = p;
    do {
        q = q->prox;
    } while ((q!=p) &&
            (q->info!=x));
    if (q==p)
        return NULL;
    else
        return q;
}
```

```
Lista BuscaCircular(
    Lista p, T x) {
    /* Busca com sentinela */
    Lista q = p;
    q->info = x;
    do {
        q = q->prox;
    } while (q->info!=x);
    if (q==p)
        return NULL;
    else
        return q;
}
```

Lista duplamente ligada com nó cabeça



Lista vazia:



- ▶ É possível percorrer os elementos nas duas direções, a partir de qualquer lugar da lista.
- ▶ É possível remover o elemento apontado.

Operações sobre listas duplamente ligadas

```
typedef
struct RegListaDupla {
    T info;
    struct RegListaDupla *esq,*dir;
} RegListaDupla, *ListaDupla;
```

```
void InsereDuplaEsq(
    ListaDupla p, T x) {
    ListaDupla q =
        malloc(sizeof(RegListaDupla));
    q->info = x;
    q->esq = p->esq;
    q->dir = p;
    p->esq->dir = q;
    p->esq = q;
}
```

```
void RemoveDupla(
    ListaDupla p, T *x) {
    p->esq->dir = p->dir;
    p->dir->esq = p->esq;
    *x = p->info;
    free(p);
}
```

A função 'RemoveDupla' supõe que há pelo menos um elemento na lista.

Exemplo: operações com polinômios

Seja um polinômio de grau n :

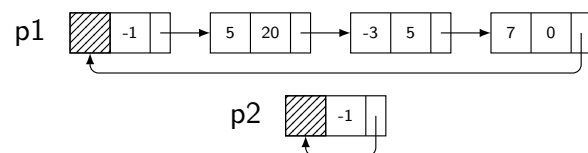
$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x^1 + a_0 x^0$$

onde $a_n \neq 0$, exceto possivelmente no caso $n = 0$.

Representação ligada omite os termos não nulos. Por exemplo, os polinômios:

$$P_1(x) = 5x^{20} - 3x^5 + 7 \quad \text{e} \quad P_2(x) = 0;$$

podem ser representados por:



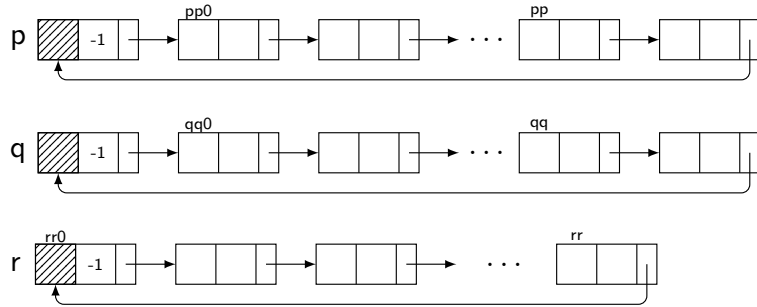
Por convenção, o expoente do nó cabeça é -1.

Exemplo de função: impressão

```
typedef struct AuxPol {
    int expo;
    float coef;
    struct AuxPol *prox;
} Termo, *Polinomio;

void ImprimePolinomio(Polinomio p) {
    if (p->prox==p) {
        printf("Polinômio nulo\n");
        return;
    }
    p = p->prox;
    while (p->expo != -1) {
        printf("(%2d,%5.1f) ",
            p->expo, p->coef);
        p = p->prox;
    }
    printf("\n");
}
```

Soma de polinômios: paradigma de intercalação



As variáveis pp e qq representam os termos correntes dos polinômios dentro da malha de repetição e a variável rr aponta para o último termo já calculado da soma; pp0, qq0 e rr0 são os valores iniciais das variáveis pp, qq e rr.

A implementação das operações é um exercício. Note-se que o produto de dois polinômios pode ser calculado como uma sequência de somas de produtos de um polinômio por um termo.

Matrizes esparsas

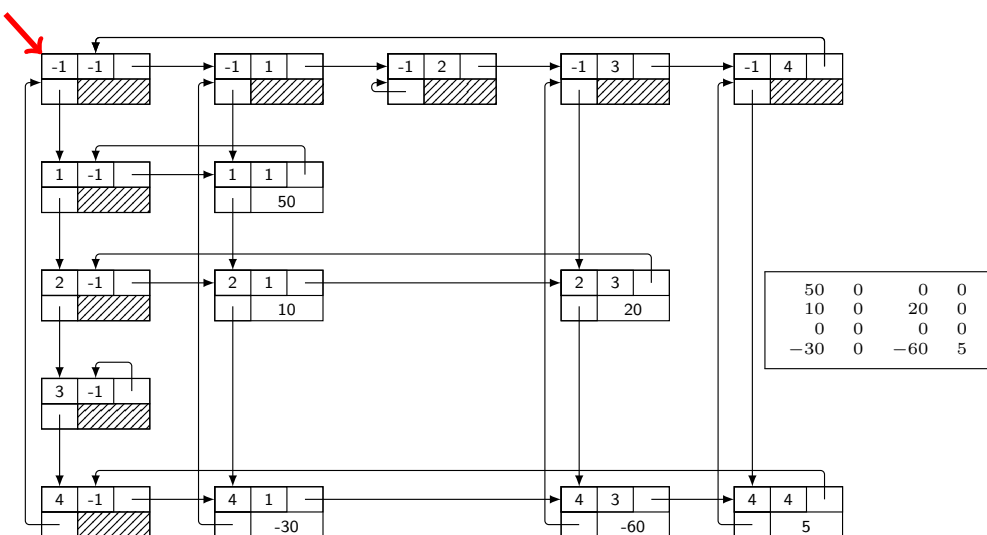
Exemplo:

$$\begin{vmatrix} 50 & 0 & 0 & 0 \\ 10 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 \\ -30 & 0 & -60 & 5 \end{vmatrix}$$

Dada uma matriz $n \times n$, quando o número de elementos não nulos é uma percentagem pequena de n^2 (não é o caso do exemplo!), pode ser conveniente representar a matriz por meio de uma estrutura de listas ortogonais.

Suporemos, neste exemplo, que as linhas e as colunas são numeradas a partir de 1.

Matrizes esparsas: listas ortogonais



O acesso à matriz é feito a partir do nó cabeça das listas das cabeças das linhas e das colunas (super-cabeça!).

Operações sobre matrizes esparsas

Alguns exemplos:

```
typedef
struct RegEsparsa {
    int linha, coluna;
    double valor;
    struct RegEsparsa *direita, *abaixo;
} RegEsparsa, *Matriz;

void InicializaMatriz(Matriz *a, int m, int n);
void LiberaMatriz(Matriz a);
double ElementoMatriz(Matriz a, int i, int j);
void AtribuiMatriz(Matriz a, int i, int j, double x);
void SomaMatrizes(Matriz a, Matriz b, Matriz *c);
void MultiplicaMatrizes(Matriz a, Matriz b, Matriz *c);
```

É importante notar os casos em que a passagem do argumento do tipo 'Matriz' é feita por referência. (Nas duas últimas operações, a variável 'c' recebe o resultado.)

Estruturas lineares

Estruturas lineares em geral

Operações típicas:

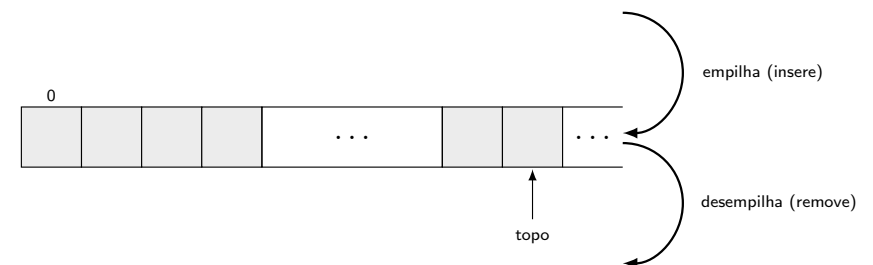
- ▶ selecionar e modificar o k -ésimo elemento;
- ▶ inserir um novo elemento entre as posições k e $k + 1$;
- ▶ remover o k -ésimo elemento;
- ▶ concatenar duas sequências;
- ▶ desdobrar uma sequência;
- ▶ copiar uma sequência;
- ▶ determinar o tamanho de uma sequência;
- ▶ buscar um elemento que satisfaz uma propriedade;
- ▶ ordenar uma sequência;
- ▶ aplicar um procedimento a todos os elementos de uma sequência;
- ▶ ...

Implementação: Depende do uso.

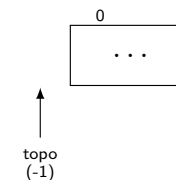
Estruturas lineares particulares

- ▶ Pilha (*stack*): inserção e remoção na mesma extremidade da estrutura (topo da pilha)
- ▶ Fila (*queue*): inserção numa extremidade (fim da fila) e remoção na outra extremidade (início da fila)
- ▶ Fila dupla (*double ended queue, dequeue*): inserção e remoção em ambas extremidades da estrutura

Pilha: implementação sequencial

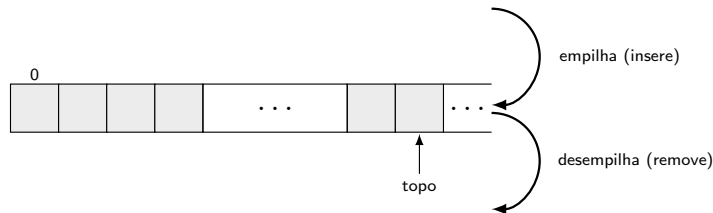


Pilha vazia:



Inicialmente: $\text{topo} = -1$ (outra convenção usada: $\text{topo} = 0$).

Pilha: implementação sequencial (cont.)

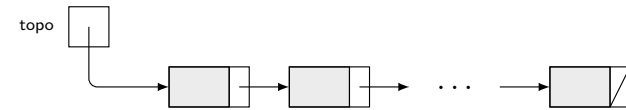


```
typedef struct {
    int topo;
    T elementos[TAM_MAX];
} Pilha;

void Empilha(Pilha *p, T x) {
    if ((*p).topo == (TAM_MAX - 1))
        TrataErro("Pilha cheia");
    (*p).topo++;
    ((*p).elementos)[(*p).topo] = x;
}
```

Exercício: a função “Desempilha”.

Pilha: implementação ligada

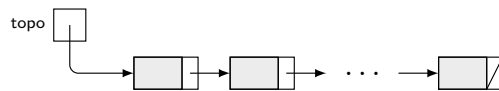


Pilha vazia:



(Uma lista ligada simples.)

Pilha: implementação ligada (cont.)

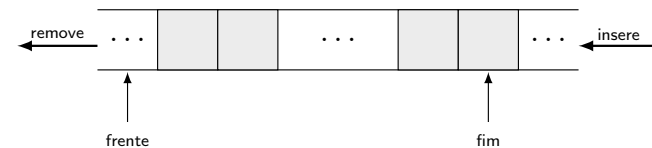


```
typedef struct ElemPilha {
    T info;
    struct ElemPilha *prox;
} ElemPilha, *Pilha;

void Empilha(Pilha *p, T x) {
    Pilha q =
        malloc(sizeof(ElemPilha));
    if (q == NULL)
        TrataErro("Falta memória");
    q->info = x;
    q->Prox = *p;
    *p = q;
}
```

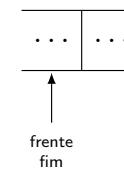
Exercício: a função “Desempilha”.

Fila: implementação sequencial



Convenção: *frente* precede o primeiro elemento da fila; consequentemente, o tamanho da fila é dado por $fim - frente$.

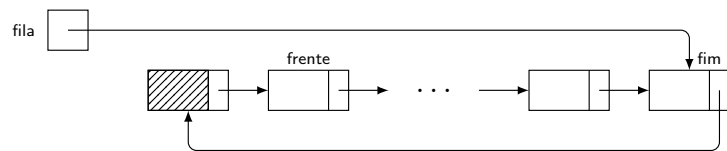
Fila vazia:



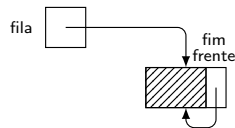
Condição de fila vazia: $frente == fim$.

Inicialmente: $frente = fim = -1$.

Fila: implementação ligada circular

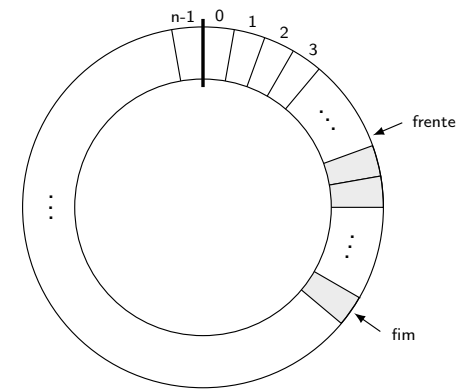


Fila vazia:



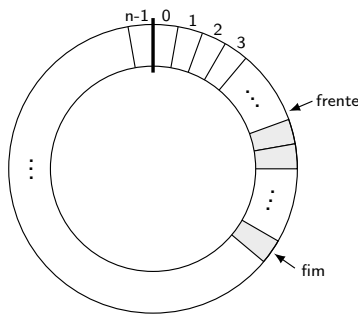
A fila pode ser representada por uma única variável (*fila*) ou um par de variáveis (*frente* e *fim*).

Fila: implementação sequencial circular



Convenção: *frente* precede o primeiro elemento da fila; consequentemente, o tamanho da fila é dado por $(fim - frente + n) \% n$.

Fila: implementação sequencial circular (cont.)



Condições:

- ▶ Inicial: $frente == fim == 0$ (ou qualquer outro valor)
- ▶ Fila vazia: $frente == fim$
- ▶ Fila cheia: $frente == fim$ (a mesma condição!)
- ▶ Solução 1: sacrificar uma posição do vetor; a condição de fila cheia fica: $frente == (fim + 1) \% n$.
- ▶ Solução 2: uma variável adicional inteira com o tamanho da fila ou booleana indicando se a fila está vazia.

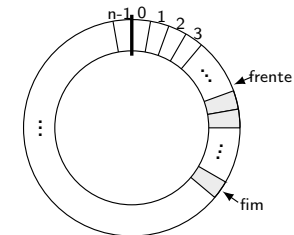
Fila: implementação sequencial circular (cont.)

```
#define TAM_MAX_FILA 1000
```

```
typedef struct {  
    int frente, fim;  
    T elementos[TAM_MAX_FILA];  
} Fila;
```

```
void InsereFila(Fila *f, T x) {  
    if ((*f).frente == ((*f).fim + 1) % TAM_MAX_FILA) {  
        TrataErro("Fila cheia");  
    }  
    (*f).fim = ((*f).fim + 1) % TAM_MAX_FILA;  
    (*f).elementos[(*f).fim] = x;  
}
```

Exercício: a função "RemoveFila".



Aplicações de pilhas

Aplicações de pilhas

- ▶ Processamento de linguagens parentéticas:
 - ▶ linguagens de programação
 - ▶ XML
 - ▶ HTML
 - ▶ \LaTeX
- ▶ Implementação da recursão
- ▶ Percurso de estruturas hierárquicas (árvores)
- ▶ Avaliação expressões em notação pós-fixa (código executável de programas)
- ▶ Transformação entre notações

Exemplo de aplicação simples

Balanceamento de parênteses:

Correto	Incorreto
ϵ	(
())
[()]	[]
[] () [() []]	() () [
((([[[[]]])))))) (

- ▶ No caso de um único tipo de parênteses, bastaria uma simples contagem (exercício).
- ▶ O caso de mais de um tipos de parênteses? Vários contadores?

Balanceamento de parênteses (cont.)

<i>Pilha</i>	<i>Resto da sequência</i>
Vazia	([([([()]]])
([([([()]])
([([([()]])
([([([()]])
([([] [()]])
([([([()]])
([([([()]])
([([([()]])
([([([([]])
([([([([()])
([([([([([])
()
Vazia	€

Notações para expressões aritméticas

- *Infixa (tradicional)*:
 - um operador unário precede o operando
 - um operador binário separa os dois operandos
 - parênteses indicam prioridades
- *Pré-fixa*: os operadores precedem os operandos (notação polonesa – usada em lógica matemática: Jan Łukasiewicz)
- *Pós-fixa*: os operadores seguem os operandos (notação polonesa reversa – código executável)

Exemplos:

<i>infixa</i>	<i>pós-fixa</i>	<i>pré-fixa</i>
a	a	a
$a + b$	$ab+$	$+ab$
$a + b * c$	$abc * +$	$+a * bc$
$(a + b) * c$	$ab + c*$	$* + abc$

Exemplo: avaliação de expressões sob forma pós-fixa

Notação infixa: $(3 + 5) * 2 - (10 - 3) / 2$

Notação pós-fixa: $3\ 5 + 2 * 10\ 3 - 2 / -$

Estados consecutivos da pilha:

<i>Pilha</i>	<i>Entrada</i>
Vazia	$3\ 5 + 2 * 10\ 3 - 2 / -$
3	$5 + 2 * 10\ 3 - 2 / -$
3 5	$+2 * 10\ 3 - 2 / -$
8	$2 * 10\ 3 - 2 / -$
8 2	$*10\ 3 - 2 / -$
16	$10\ 3 - 2 / -$
16 10	$3 - 2 / -$
16 10 3	$-2 / -$
16 7	$2 / -$
16 7 2	$/ -$
16 3	$-$
13	Vazia

Analogia com código executável.

Exemplo: transformação de notação infixa para pós-fixa

Notação tradicional: $ab + cd^e / f - gh$

Entrada infixa: $a * b + c * d \wedge e / f - g * h$

Saída pós-fixa: $a\ b * c\ d\ e \wedge * f / + g\ h * -$

- As variáveis são copiadas diretamente para a saída.
- Os operadores precisam ser lembrados numa pilha.
- Um operador é copiado da pilha para a saída somente quando aparece na entrada um operador de prioridade menor ou igual.

Transformação de notação infixa para pós-fixa (cont.)

<i>Saída</i>	<i>Pilha</i>	<i>Entrada</i>
		$a * b + c * d \wedge e / f - g * h$
a		$* b + c * d \wedge e / f - g * h$
a	*	$b + c * d \wedge e / f - g * h$
ab	*	$+ c * d \wedge e / f - g * h$
$ab*$		$+ c * d \wedge e / f - g * h$
$ab*$	+	$c * d \wedge e / f - g * h$
$ab * c$	+	$* d \wedge e / f - g * h$
$ab * c$	++	$d \wedge e / f - g * h$
$ab * cd$	++	$\wedge e / f - g * h$
$ab * cd$	++ ^	$e / f - g * h$
$ab * cde$	++ ^	$/ f - g * h$
$ab * cde \wedge$	++	$/ f - g * h$

(continua)

Transformação de notação infixa para pós-fixa (cont.)

Saída	Pilha	Entrada
$ab * cde \wedge *$	+	$/f - g * h$
$ab * cde \wedge *$	+ /	$f - g * h$
$ab * cde \wedge * f$	+ /	$- g * h$
$ab * cde \wedge * f /$	+	$- g * h$
$ab * cde \wedge * f / +$		$- g * h$
$ab * cde \wedge * f / +$	-	$g * h$
$ab * cde \wedge * f / + g$	-	$* h$
$ab * cde \wedge * f / + g$	- *	h
$ab * cde \wedge * f / + gh$	- *	
$ab * cde \wedge * f / + gh *$	-	
$ab * cde \wedge * f / + gh * -$		

A transformação é análoga à tradução feita por um compilador.

Transformação de notação infixa para pós-fixa (cont.)

Generalização para expressões com parênteses

- ▶ Uma expressão contida entre parênteses deve ser transformada de maneira independente do seu contexto.
- ▶ A transformação da expressão envolvente deve ser suspensa, criando-se uma nova pilha para a expressão entre parênteses.
- ▶ Terminada a transformação da expressão entre parênteses, deve ser retomada a transformação da expressão envolvente.
- ▶ Dentro de uma expressão entre parênteses pode haver outras análogas resultando em criação de novas pilhas.
- ▶ Não é necessário criar fisicamente novas pilhas: basta empilhar na mesma pilha única o símbolo de entrada '(' até que ele case com o símbolo ')' correspondente.

Notação pós-fixa e operadores unários

Problema – expressões distintas produzindo resultados iguais:

$$\begin{array}{ll} a - (-b) & ab -- \\ -(a - b) & ab -- \end{array}$$

Solução – adotar símbolos diferentes para os operadores unários (por exemplo, '~' em lugar de '-' e '&' em lugar de '+':

$$\begin{array}{ll} a - (-b) & ab \sim - \\ -(a - b) & ab - \sim \\ a - (+b) & ab \& - \end{array}$$

Os algoritmos de avaliação de expressões e de transformação de notação infixa para a pós-fixa podem ser facilmente adaptados.

Recursão e retrocesso

Considerações gerais

- ▶ Clareza
- ▶ Consequência direta das definições
- ▶ Programação mais simples
- ▶ Pode causar sacrifício de eficiência:
 - ▶ em termos de constante de proporcionalidade
 - ▶ versão iterativa
 - ▶ pilha explícita
 - ▶ em termos assintóticos; por exemplo, $O(2^n)$ em lugar de $O(n)$
- ▶ Implementação do retrocesso
- ▶ Algoritmos de busca exaustiva

Exemplo 1: função fatorial

```
int fatorial(int n) {  
    if (n==0)  
        return 1;  
    else  
        return n*fatorial(n-1);  
}
```

```
int fatorial(int n) {  
    int i, f=1;  
    for (i=1; i<=n; i++)  
        f = f*i;  
    return f;  
}
```

Eficiência:

- ▶ Operações (multiplicações): ambas as funções $O(n)$
- ▶ Diferença: constante de proporcionalidade
- ▶ Espaço: recursiva – $O(n)$, iterativa – $O(1)$ (constante)

Exemplo 2: números de Fibonacci

Definição: $f(0) = 0$, $f(1) = 1$, $f(n) = f(n-1) + f(n-2)$, $n \geq 1$
Alguns valores: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

```
int fibo(int n) { // recursivo  
    if (n<=1)  
        return n;  
    else  
        return fibo(n-1)+fibo(n-2);  
}
```

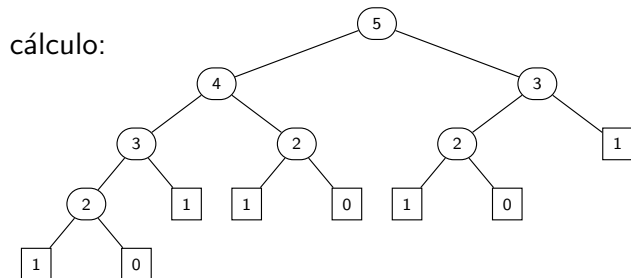
```
int fibo(int n) { // iterativo  
    int f1=0, f2=1, f3, i;  
    for (i=1; i<=n; i++) {  
        f3 = f1+f2;  
        f1 = f2; f2 = f3;  
    }  
    return f1;  
}
```

Eficiência:

- ▶ Operações (somas): recursiva – $O(1.6^n)$, iterativa – $O(n)$
- ▶ Espaço: recursiva – $O(n)$, iterativa – $O(1)$ (constante)
- ▶ Por exemplo, para $n = 100$:
recursiva: $\approx 10^{20}$ somas, iterativa: 100 somas

Exemplo 2: números de Fibonacci (cont.)

- ▶ Árvore de cálculo:



- ▶ Seja $f(n)$ o número de somas para calcular $\text{fibo}(n)$:

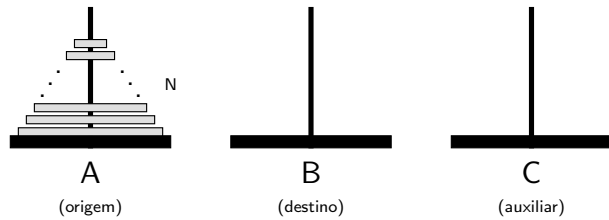
$$f(n) = \begin{cases} 0 & \text{se } n=0 \text{ ou } n=1 \\ f(n-1) + f(n-2) + 1 & \text{se } n > 1 \end{cases}$$

$f(n)$: 0, 0, 1, 2, 4, 7, 12, 20, 33, 54, ...

$\text{fibo}(n)$: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

- ▶ Prova-se por indução que: $f(n) = \text{fibo}(n+1) - 1$ ($n \geq 0$).
- ▶ Usando o fato que: $\text{fibo}(n) = \frac{\Phi^n - \phi^n}{\sqrt{5}}$, com $\Phi \approx 1.61$ e $\phi \approx -0.61$, obtém-se $f(n) = O(1.6^n)$.

Exemplo 3: Torres de Hanoi (ou Torres de Brama)



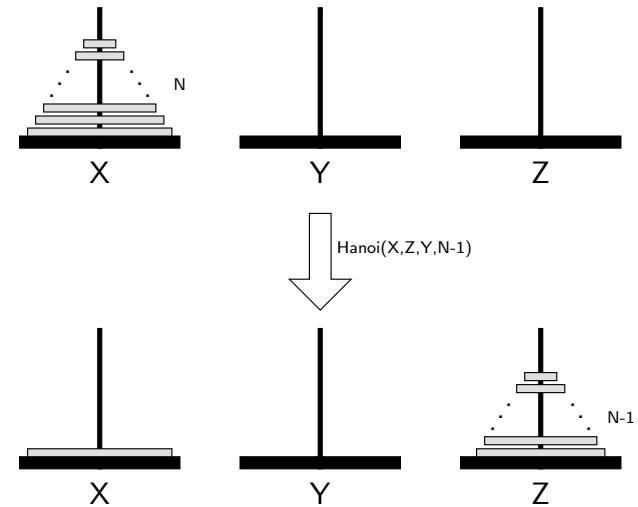
Objetivo: transferir os N discos da torre A para a torre B, usando a torre C como auxiliar.

Regras:

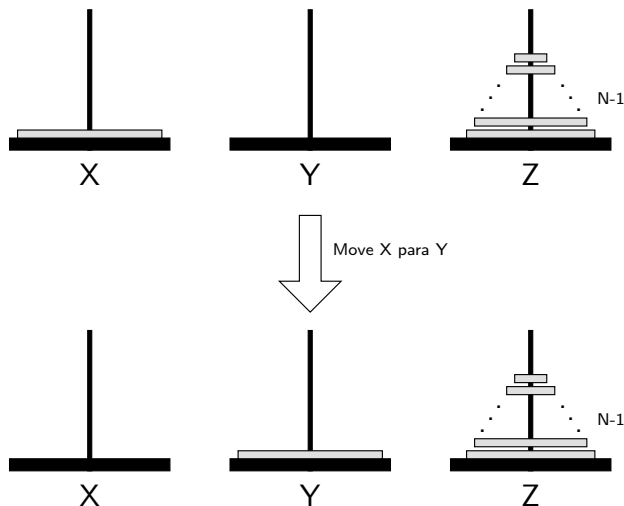
- ▶ um disco de cada vez
- ▶ disco de diâmetro maior não pode ficar em cima de um disco de diâmetro menor

Solução recursiva: função $Hanoi(org,dest,aux,n)$.

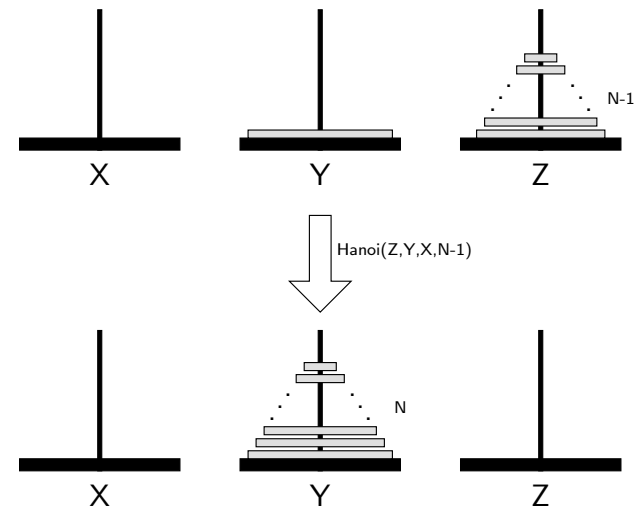
Torres de Hanoi: transferência recursiva de $N-1$ discos



Torres de Hanoi: movimento do maior disco



Torres de Hanoi: transferência recursiva final de $N-1$ discos



Torres de Hanoi: função *Hanoi*

```
void Hanoi(char org, char dest, char aux, int n) {  
    if (n>0) {  
        Hanoi(org, aux, dest, n-1);  
        printf("Mova de %c para %c\n", org, dest);  
        Hanoi(aux, dest, org, n-1);  
    }  
}
```

- ▶ Chamada inicial: *Hanoi*('A','B','C',64).
- ▶ Número de movimentos: $2^N - 1$ (prova por indução).
- ▶ Este é o número mínimo: $2^{64} - 1 = 18.446.744.073.709.551.615$.
- ▶ Supondo um movimento por segundo, levaria cerca de 585 bilhões de anos.
- ▶ Existem várias soluções iterativas mas ainda exigem $2^N - 1$ movimentos.

Torres de Hanoi: exemplos de saída

N=1:

Mova de A para B

N=2:

Mova de A para C
Mova de A para B
Mova de C para B

N=3:

Mova de A para B
Mova de A para C
Mova de B para C
Mova de A para B
Mova de C para A
Mova de C para B
Mova de A para B

Torres de Hanoi: exemplos de saída (cont.)

N=4

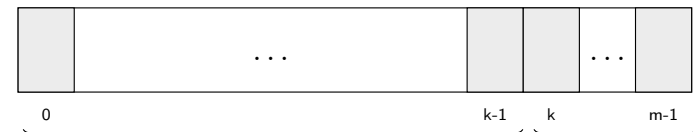
Mova de A para C	Mova de C para B
Mova de A para B	Mova de C para A
Mova de C para B	Mova de B para A
Mova de A para C	Mova de C para B
Mova de B para A	Mova de A para C
Mova de B para C	Mova de A para B
Mova de A para C	Mova de C para B
Mova de A para B	

Exemplo de animação:

<http://users.encs.concordia.ca/~twang/WangApr01/RootWang.html>

Exemplo 4: geração de permutações

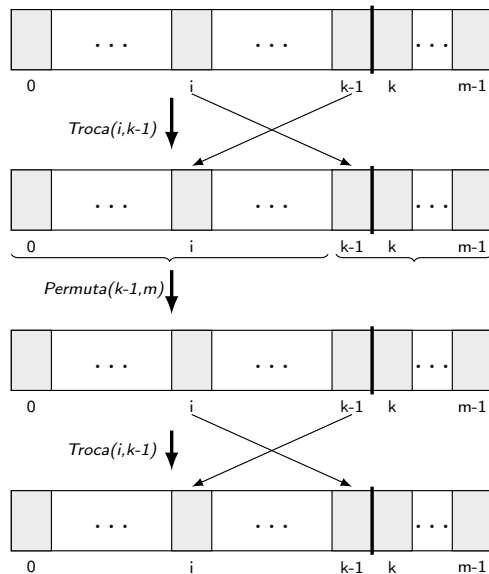
Problema: Gerar todas as permutações dos m elementos de um vetor.



- ▶ Suponha uma função *Permuta*(k,m) que gera (imprime) todas as permutações dos elementos de 0 a $k-1$, seguidas dos elementos de k a $m-1$.
- ▶ A chamada inicial *Permuta*(m,m) resolveria o problema.
- ▶ A solução consistirá em trocar o elemento de índice $k-1$ consecutivamente com todos os elementos que o precedem e aplicar a função recursivamente.

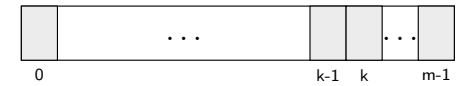
Geração das permutações (cont.)

Passo recursivo: $i=k-1, \dots, 0$



Geração das permutações (cont.)

Função *Permuta*:



```
void Permuta(int k, int m) {  
    if (k==0)  
        Imprime(m);  
    else {  
        int i;  
        for (i=k-1; i>=0; i--) {  
            Troca(i, k-1);  
            Permuta(k-1, m);  
            Troca(i, k-1);  
        }  
    }  
}
```

- ▶ A função *Imprime* imprime os m elementos do vetor.
- ▶ Chamada inicial: *Permuta*(m, m).
- ▶ Uma sequência de n elementos tem $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$ permutações (aproximação de Stirling).
- ▶ Exemplos: $10! = 3.628.800$ $15! = 1.307.674.368.000$.

Geração das permutações (cont.)

Saída de *Permuta*(3,3)

```
1 2 3  
2 1 3  
1 3 2  
3 1 2  
3 2 1  
2 3 1
```

Desafio: imprimir em ordem lexicográfica:

```
1 2 3  
1 3 2  
2 1 3  
2 3 1  
3 1 2  
3 2 1
```

Exemplos de retrocesso

Exemplo 1: movimentos do cavalo

Movimentos possíveis do cavalo no jogo de xadrez:

	-2	-1	0	1	2
-2		2		1	
-1	3				0
0					
1	4				7
2		5		6	

Movimentos do cavalo (cont.)

Problema: achar um percurso entre duas posições dadas.

Exemplo de solução: percurso da posição (0,0) até (4,4) (existem 27.419 soluções):

	0	1	2	3	4
0	1	4	9	12	
1	10	13	6	3	
2	5	2	11	8	
3		7	14		
4					15

Movimentos do cavalo (cont.)

Um percurso da posição (0,0) até (4,4) cobrindo todas as posições:

	0	1	2	3	4
0	1	12	17	6	23
1	18	7	22	11	16
2	13	2	19	24	5
3	8	21	4	15	10
4	3	14	9	20	25

Obs.: Não existe solução para o tabuleiro da transparência anterior (provar!).

Movimentos do cavalo (cont.)

Tipos de solução:


1. Achar uma solução
2. Achar uma solução que cobre todas as posições livres
3. Enumerar todas as soluções

Observação: Esta não é a melhor maneira de resolver este problema mas ilustra bem o mecanismo geral de retrocesso.

Movimentos do cavalo (cont.)

```
#define TAM.MAX 20
#define NUM.MOV 8
typedef enum {false, true} Boolean;
int tab[TAM.MAX][TAM.MAX];
int dl[NUM.MOV] = { -1, -2, -2, -1, 1, 2, 2, 1 };
int dc[NUM.MOV] = { 2, 1, -1, -2, -2, -1, 1, 2 };

void ImprimeTab(int tam) {
    int i, j;
    for (i=0; i<tam; i++) {
        for (j=0; j<tam; j++) {
            printf("%5d", tab[i][j]);
            printf("\n");
        }
    }
}
```

	-2	-1	0	1	2
-2		2		1	
-1	3				0
0					
1	4				7
2		5		6	

Movimentos do cavalo: achar uma solução

```
Boolean TentaMovimento(int tam, int num, int lin,
                        int col, int ld, int cd) {
    int k, lp, cp;
    Boolean res = false;
    if ((0<=lin) && (lin<tam) && (0<=col) &&
        (col<tam) && (tab[lin][col]==0)) {
        tab[lin][col] = num;
        if ((lin==ld) && (col==cd)) {
            res = true; ImprimeTab(tam);
        } else { k = 0;
            do { lp = lin+dl[k]; cp = col+dc[k];
                res = TentaMovimento(tam, num+1, lp, cp, ld, cd);
                k++;
            } while ((!res) && (k<NUM.MOV));
        }
        tab[lin][col] = 0;
    }
    return res;
}
```

Chamada inicial: *TentaMovimento(tam,1,lo,co,ld,cd)*

Movimentos do cavalo: exemplo de entrada e saída

	0	1	2	3	4
0	1	4	9	12	
1	10	13	6	3	
2	5	2	11	8	
3		7	14		
4					15

Entrada

Saída

5	1	4	9	12	-1
0 0	10	13	6	3	0
4 4	5	2	11	8	0
0 4	0	7	14	0	-1
3 4	-1	0	0	0	15
4 0					
-1 -1					

Número de tentativas: 121

Movimentos do cavalo: achar uma solução completa

```
Boolean TentaMovimento(int tam, int num, int lin,
                        int col, int ld, int cd, int noc) {
    int k, lp, cp;
    Boolean res = false;
    if ((0<=lin) && (lin<tam) && (0<=col) &&
        (col<tam) && (tab[lin][col]==0)) {
        tab[lin][col] = num;
        if ((lin==ld) && (col==cd) && ((noc+num)==tam*tam)) {
            res = true; ImprimeTab(tam);
        } else { k = 0;
            do { lp = lin+dl[k]; cp = col+dc[k];
                res =
                    TentaMovimento(tam, num+1, lp, cp, ld, cd, noc);
                k++;
            } while ((!res) && (k<NUM.MOV));
        }
        tab[lin][col] = 0;
    }
    return res;
}
```

Chamada inicial: *TentaMovimento(tam,1,lo,co,ld,cd,ocupadas)*

Movimentos do cavalo: achar todas as soluções

```
void TentaMovimento(int tam, int num, int lin,
                   int col, int ld, int cd) {
    int k, lp, cp;
    if ((0<=lin) && (lin<tam) && (0<=col) &&
        (col<tam) && (tab[lin][col]==0)) {
        tab[lin][col] = num;
        if ((lin==ld) && (col==cd)) {
            ImprimeTab(tam);
        } else {
            k = 0;
            do { lp = lin+dl[k]; cp = col+dc[k];
                TentaMovimento(tam,num+1,lp,cp,ld,cd);
                k++;
            } while (k<NUM.MOV);
        }
        tab[lin][col] = 0;
    }
}
```

Chamada inicial: *TentaMovimento(tam,1,lo,co,ld,cd)*

Exemplo 2: distância de edição

RECURSÃO E RETROCESSO

RCURÇÃO ER RTROCESOS

^	^	^	^	^	^
I	S	R	I	I	R

- ▶ Operações elementares:
 - ▶ **A**: avanço (subentendido)
 - ▶ **I**: inserção
 - ▶ **S**: substituição
 - ▶ **R**: remoção
- ▶ Cada operação recebe um custo (avanço, em geral, zero)
- ▶ Problema: achar uma sequência de operações que torna as cadeias iguais ao custo total mínimo.

Distância de edição: função *Distancia*

```
int Distancia(char *teste, char *correta) {
    int dIns, dRem, dSub;
    if (((*teste)==NUL_CHAR) && ((*correta)==NUL_CHAR))
        return 0;
    dIns = dRem = dSub = INT_MAX;
    if (((*teste)!=NUL_CHAR) && ((*correta)!=NUL_CHAR) &&
        ((*teste)==(*correta)))
        return Distancia(teste+1,correta+1);
    if (((*teste)!=NUL_CHAR) && ((*correta)!=NUL_CHAR))
        dSub = custoSub+Distancia(teste+1,correta+1);
    if ((*teste)!=NUL_CHAR)
        dRem = custoRem+Distancia(teste+1,correta);
    if ((*correta)!=NUL_CHAR)
        dIns = custoIns+Distancia(teste,correta+1);
    return min(dIns, min(dRem, dSub));
}
```

Distância de edição: desafios

- ▶ Melhorar o desempenho do algoritmo: o algoritmo é exponencial não sendo viável, sob esta forma, em aplicações práticas
- ▶ Imprimir o número de operações de cada tipo (avanço, inserção, remoção e substituição) para obter a solução
- ▶ Imprimir a sequência de operações para obter a solução

Eliminação da recursão

Exemplo: Torres de Hanoi

```
void Hanoi(char org, char dest, char aux, int n) {
    if (n>0) {
        Hanoi(org, aux, dest, n-1); // (1)
        printf("Mova de %c para %c\n", org, dest);
        Hanoi(aux, dest, org, n-1); // (2)
    }
}
```

Simulação:

- ▶ Ideia básica: simular a recursão usando uma pilha conveniente.
- ▶ Para cada chamada recursiva:
 - ▶ salva os valores dos parâmetros e das variáveis locais
 - ▶ salva uma indicação do ponto de chamada: (1) ou (2).
 - ▶ calcula novos valores dos parâmetros
 - ▶ volta a simular uma chamada
- ▶ Para cada retorno
 - ▶ restaura os valores salvos
 - ▶ continua simulação no ponto seguinte ao de chamada
- ▶ Pilha vazia indica término da simulação.

Exemplo: Torres de Hanoi (cont.)

```
typedef enum {chamada1, chamada2} Chamadas;
typedef enum {entrada, saida, retorno} Acoes;
void Hanoi(char org, char dest, char aux, int n) {
    Pilha f; Chamadas ch; Acoes acao=entrada; char t;
    InicializaPilha(&f);
    do { switch (acao) {
        case (entrada) {
            if (n>0) {
                Empilha(f, org, dest, aux, n, chamada1);
                t = dest; dest = aux; aux = t; n = n-1;
            } else acao = retorno;
            break;
        }
        case (retorno) {
            if (PilhaVazia(f)) acao = saida; break;
            else {
                Desempilha(f, &org, &dest, &aux, &n, &ch);
                switch (ch) {
                    case (chamada1) {
                        printf("Mova de %c para %c\n", org, dest);
                        Empilha(f, org, dest, aux, n, chamada2);
                        t = aux; aux = org; org = t; n = n-1;
                        acao = entrada; break;
                    }
                    case (chamada2) {
                        break;
                    }
                }
            }
        }
    } while (acao!=saida);
}
```

```
void Hanoi(char org, char dest, char aux, int n) {
    if (n>0) {
        Hanoi(org, aux, dest, n-1); // (1)
        printf("Mova de %c para %c\n", org, dest);
        Hanoi(aux, dest, org, n-1); // (2)
    }
}
```

Esquema geral de função recursiva

```
void Exemplo(T1 x1, T2 x2, ...) {
    S1 y1; S2 y2; ...;
    Ci; /* Comandos iniciais */
    if (E(...)) {
        C0; /* Caso base */
    } else { /* Chamadas recursivas */
        C1; Exemplo(e11, e12, ...);
        C2; Exemplo(e21, e22, ...);
        C3; Exemplo(e31, e32, ...);
        ...;
        Cm; Exemplo(em1, em2, ...);
        Cf;
    }
}
```

Os símbolos C_i , C_0 , C_1 , ..., C_m e C_f representam sequências, possivelmente vazias, de comandos.

Esquema de eliminação da recursão

```
typedef enum {chamada1, chamada2, chamada3, ...} Chamadas;
typedef enum {entrada, saida, retorno} Acoes;
```

```
void Exemplo(T1 x1, T2 x2, ...) {
    S1 y1; S2 y2; ...; /* variáveis locais originais */
    T1 t1, T2 t2, ...; /* variáveis temporárias */
    Pilha f; Chamadas ch; Acoes acao;
    InicializaPilha(&f); acao = entrada;
    do {
        switch (acao) {
            case (entrada): ... break;
            case (retorno): ... break;
        }
    } while (acao!=saida);
}
```

```
void Exemplo(T1 x1, T2 x2, ...) {
    S1 y1; S2 y2; ...;
    Ci; /* Comandos iniciais */
    if (E(...)) {
        C0; /* Caso base */
    } else {
        /* Chamadas recursivas */
        C1; Exemplo(e11,e12,...);
        C2; Exemplo(e21,e22,...);
        C3; Exemplo(e31,e32,...);
        ...;
        Cm; Exemplo(em1,em2,...);
        Cf;
    }
}
```

Esquema de eliminação da recursão (cont.)

```
case (entrada):
    Ci; /* Comandos iniciais */
    if (E(...)) {
        C0; acao = retorno; /* Caso base */
    } else { /* Primeira chamada recursiva */
        C1; Empilha(f,x1,x2,...,y1,y2,...,chamada1);
        t1 = e11; t2 = e12; ...;
        x1 = t1; x2 = t2; ...; /* Recalcula argumentos */
    }
    break;
```

```
void Exemplo(T1 x1, T2 x2, ...) {
    S1 y1; S2 y2; ...;
    Ci; /* Comandos iniciais */
    if (E(...)) {
        C0; /* Caso base */
    } else {
        /* Chamadas recursivas */
        C1; Exemplo(e11,e12,...);
        C2; Exemplo(e21,e22,...);
        C3; Exemplo(e31,e32,...);
        ...;
        Cm; Exemplo(em1,em2,...);
        Cf;
    }
}
```

Esquema de eliminação da recursão (cont.)

```
case (retorno):
    if (PilhaVazia(f)) acao = saida;
    else {
        Desempilha(f,&x1,&x2,...,&y1,&y2,...,&ch);
        switch (ch) {
            case (chamada1):
                C2; Empilha(f,x1,x2,...,y1,y2,...,chamada2);
                t1 = e21; t2 = e22; ...;
                x1 = t1; x2 = t2; ...;
                acao = entrada; break;
            case (chamada2):
                C3; Empilha(f,x1,x2,...,y1,y2,...,chamada3);
                t1 = e31; t2 = e32; ...;
                x1 = t1; x2 = t2; ...;
                acao = entrada; break;
            ...;
            case (chamadam):
                Cf; break;
        } /* switch (ch) */
    }
    break;
```

```
void Exemplo(T1 x1, T2 x2, ...) {
    S1 y1; S2 y2; ...;
    Ci; /* Comandos iniciais */
    if (E(...)) {
        C0; /* Caso base */
    } else {
        /* Chamadas recursivas */
        C1; Exemplo(e11,e12,...);
        C2; Exemplo(e21,e22,...);
        C3; Exemplo(e31,e32,...);
        ...;
        Cm; Exemplo(em1,em2,...);
        Cf;
    }
}
```

Exemplo 1: função fatorial

```
int fatorial(int n) {
    if (n==0)
        return 1;
    else
        return n*fatorial(n-1);
}
```

Função fatorial (cont.)

```
typedef enum {chamada1} Chamadas;
typedef enum {entrada, saida, retorno} Acoes;
```

```
int fatorial(int n) {
    int res, t1;
    Pilha f; Chamadas ch; Acoes acao;
    InicializaPilha(&f); acao = entrada;
    do {
        switch (acao) {
            case (entrada): ... break;
            case (retorno): ... break;
        }
    } while (acao!=saida);
    return res;
} /* fatorial */
```

```
int fatorial(int n) {
    if (n==0)
        return 1;
    else
        return n*fatorial(n-1);
}
```

Função fatorial (cont.)

```
case (entrada):
    if (n==0) {
        res = 1; acao = retorno;
    } else {
        Empilha(f,n,chamada1);
        t1 = n; n = t1-1;
    }
    break;
```

```
int fatorial(int n) {
    if (n==0)
        return 1;
    else
        return n*fatorial(n-1);
}
```

Função fatorial (cont.)

```
case (retorno):
    if (PilhaVazia(f)) acao = saida;
    else {
        Desempilha(f,&n,&ch);
        switch (ch) {
            case (chamada1):
                res = n*res;
                break;
        }
    }
    break;
```

```
int fatorial(int n) {
    if (n==0)
        return 1;
    else
        return n*fatorial(n-1);
}
```

Obs.: Note como neste caso a variável *res* é usada para guardar o resultado da função.

Exemplo 2: função *Hanoi*

```
void Hanoi(char org, char dest, char aux, int n) {
    if (!(n>0))
        ;
    else {
        Hanoi(org, aux, dest, n-1);
        printf("Mova de %c para %c\n", org, dest);
        Hanoi(aux, dest, org, n-1);
    }
}
```

Função *Hanoi* (cont.)

```
typedef enum {chamada1, chamada2} Chamadas;
typedef enum {entrada, saida, retorno} Acoes;
void Hanoi(char org, char dest, char aux, int n) {
    char t1; char t2; char t3; int t4;
    Pilha f; Chamadas ch; Acoes acao;
    InicializaPilha(&f); acao = entrada;
    do {
        switch (acao) {
            case (entrada): ...; break;
            case (retorno): ...; break;
        }
    } while (acao!=saida);
}
```

```
void Hanoi(char org, char dest, char aux, int n) {
    if (!(n>0))
        ;
    else {
        Hanoi(org, aux, dest, n-1);
        printf("Mova_de_%c_para_%c\n", org, dest);
        Hanoi(aux, dest, org, n-1);
    }
}
```

Função *Hanoi* (cont.)

```
case (entrada):
    if (!(n>0)) {
        acao = retorno;
    } else {
        Empilha(f, org, dest, aux, n, chamada1);
        t1 = org; t2 = aux; t3 = dest; t4 = n-1;
        org = t1; dest = t2; aux = t3; n = t4;
    }
    break;
```

```
void Hanoi(char org, char dest, char aux, int n) {
    if (!(n>0))
        ;
    else {
        Hanoi(org, aux, dest, n-1);
        printf("Mova_de_%c_para_%c\n", org, dest);
        Hanoi(aux, dest, org, n-1);
    }
}
```

Função *Hanoi* (cont.)

```
case (retorno):
    if (PilhaVazia(f))
        acao = saida;
    else {
        Desempilha(f, &org, &dest, &aux, &n, &ch);
        switch (ch) {
            case (chamada1):
                printf("Mova_de_%c_para_%c\n", org, dest);
                Empilha(f, org, dest, aux, n, chamada2);
                t1 = aux; t2 = dest; t3 = org; t4 = n-1;
                org = t1; dest = t2; aux = t3; n = t4;
                acao = entrada;
                break;
            case (chamada2):
                break;
        }
    }
    break;
```

```
void Hanoi(char org, char dest, char aux, int n) {
    if (!(n>0))
        ;
    else {
        Hanoi(org, aux, dest, n-1);
        printf("Mova_de_%c_para_%c\n", org, dest);
        Hanoi(aux, dest, org, n-1);
    }
}
```

Exemplo de eliminação da recursão caudal

Aplicável quando a última ação dentro do corpo da função é uma chamada recursiva: reaproveita o mesmo registro de ativação da função, mudando os valores dos argumentos.

```
void Hanoi(char org, char dest, char aux, int n) {
    if (n>0) {
        Hanoi(org, aux, dest, n-1);
        printf("Mova_de_%c_para_%c\n", org, dest);
        Hanoi(aux, dest, org, n-1);
    }
}
```

A chamada recursiva é substituída por recálculo dos valores dos parâmetros e reexecução da função:

```
void Hanoi(char org, char dest, char aux, int n) {
    char t;
    while (n>0) {
        Hanoi(org, aux, dest, n-1);
        printf("Mova_de_%c_para_%c\n", org, dest);
        t = org; org = aux; aux = t; n = n-1;
    }
}
```

Recursão mútua: Análise sintática

Exemplo simples de recursão mútua

```
int g(int n);

int f(int n) {
    if (n==0)
        return 0;
    else
        return g(n-1);
}

int g(int n) {
    if (n==0)
        return 1;
    else
        return f(n-1);
}
```

A declaração do cabeçalho da função g é exigida pelo compilador para permitir a verificação de tipos num único passo.

Análise de expressões

Expressões com operadores binários '+', '-', '*', '/' e parênteses '(' e ')':

$$e = t_1 \oplus t_2 \oplus \dots \oplus t_n, \quad n \geq 1 \quad (' \oplus ' \text{ denota } '+' \text{ ou } '-')$$

$$t = f_1 \otimes f_2 \otimes \dots \otimes f_n, \quad n \geq 1 \quad (' \otimes ' \text{ denota } '*' \text{ ou } '/')$$

$$f = x \quad \text{ou} \quad f = (e) \quad (x \text{ denota uma variável simples})$$

Estas definições incluem as prioridades dos operadores, mas não as associatividades.

Programa de tradução de infix para pós-fixa:

- ▶ Declarações iniciais:

```
char entrada[TAM_MAX];
char *pe;
void Expressao();
void Termo();
void Fator();
void Erro();
```

Hipótese: As funções Expressao, Termo e Fator transformam a subexpressão que tem início em *pe e avançam o valor de pe para o primeiro caractere após a expressão.

- ▶ Função principal:

```
void InPos() {
    pe = &entrada[0];
    Expressao();
    if ((*pe) != '\0')
        Erro();
}
```

Fator

$$f = x \quad \text{ou} \quad f = (e)$$

```
void Fator() {
    char corrente = *pe;
    switch (corrente) {
        case 'a': case 'b': ...: case 'z':
            Sai(corrente); pe++; break;
        case '(':
            pe++;
            Expressao();
            if ((*pe)==' ')
                pe++;
            else
                Erro();
            break;
        default:
            Erro();
    }
}
```

Termo

$$t = f_1 \otimes f_2 \otimes \cdots \otimes f_n, \quad n \geq 1$$

```
void Termo() {
    char op;
    Fator();
    do {
        op = *pe;
        if ((op=='*') || (op=='/')) {
            pe++;
            Fator();
            Sai(op);
        } else
            break; /* do */
    } while (true);
}
```

Implícita a associação à esquerda.

Expressão

$$e = t_1 \oplus t_2 \oplus \cdots \oplus t_n, \quad n \geq 1$$

```
void Expressao() {
    char op;
    Termo();
    do {
        op = *pe;
        if ((op=='+') || (op=='-')) {
            pe++;
            Termo();
            Sai(op);
        } else
            break; /* do */
    } while (true);
}
```

Implícita a associação à esquerda.

Operador de exponenciação

Fator redefinido:

$$f = p_1 \wedge p_2 \wedge \cdots \wedge p_n, \quad n \geq 1$$

Primário:

$$p = x \quad \text{ou} \quad p = (e)$$

Associatividade? Solução – redefinir o fator:

$$f = p \quad \text{ou} \quad f = p \wedge f$$

Fator redefinido

$$f = p \quad \text{ou} \quad f = p \wedge f$$

```
void Fator() {
    Primario();
    if ((*pe)=='^') {
        pe++;
        Fator();
        Sai('^');
    }
}
```

Primário

$$p = x \quad \text{ou} \quad p = (e)$$

```
void Primario() {
    corrente = *pe;
    switch (corrente) {
        case 'a': case 'b': ...: case 'z':
            Sai(corrente); pe++; break;
        case '(':
            pe++;
            Expressao();
            if ((*pe)==' ')
                pe++;
            else
                Erro();
            break;
        default:
            Erro();
    }
}
```

Analogia para expressões e termos

$$\begin{array}{lll} e = t & \text{ou} & e = e \oplus t \\ t = f & \text{ou} & t = t \otimes f \end{array}$$

Problemas:

- ▶ como distinguir as alternativas
- ▶ repetição infinita no segundo caso (recursão esquerda):

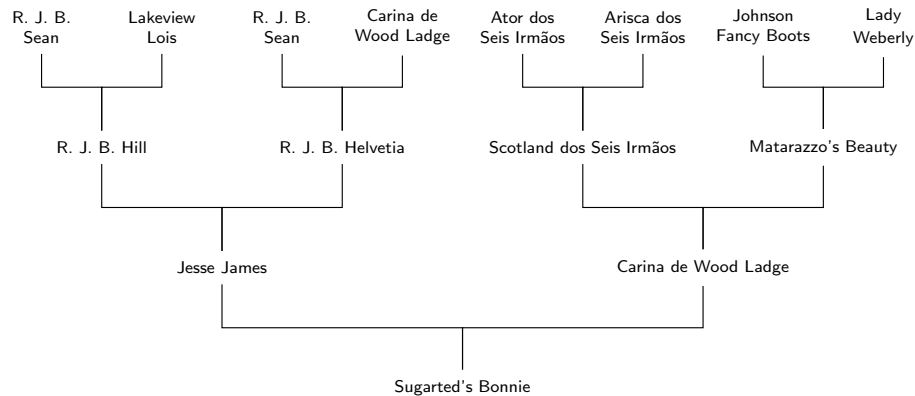
```
void Expressao() {
    ...;
    if (???)
        Termo();
    else
        Expressao();
    ...
}
```

Solução:

- ▶ manter a iteração (comando **do ... while**)
- ▶ interpretar a associação esquerda por programa

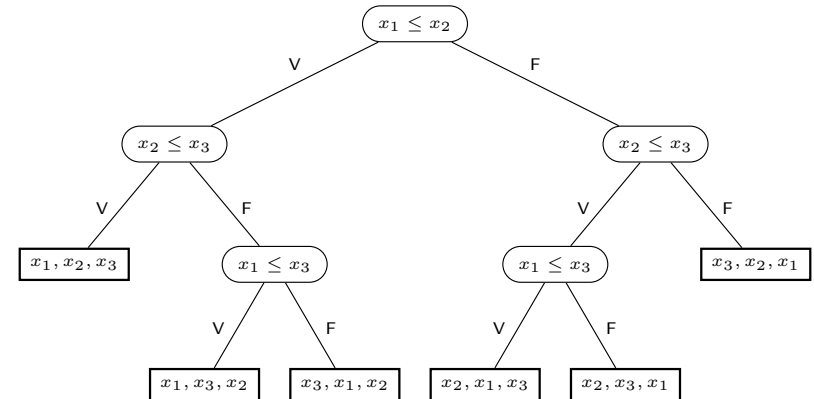
Árvores binárias

Exemplo de árvore binária 1: antepassados (*pedigree*)



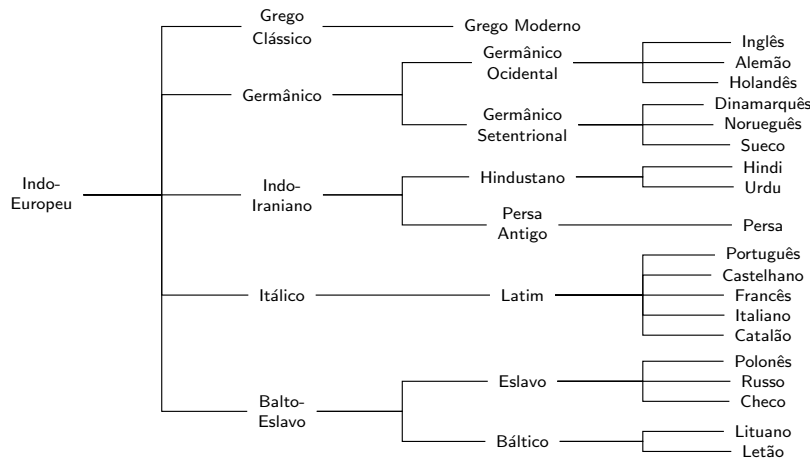
- ▶ Alguns nomes são repetidos: eles devem ser tratados como instâncias separadas.
- ▶ Pela própria natureza da árvore, cada elemento tem dois predecessores: *árvore binária*.

Exemplo de árvore binária 2: árvore de decisão



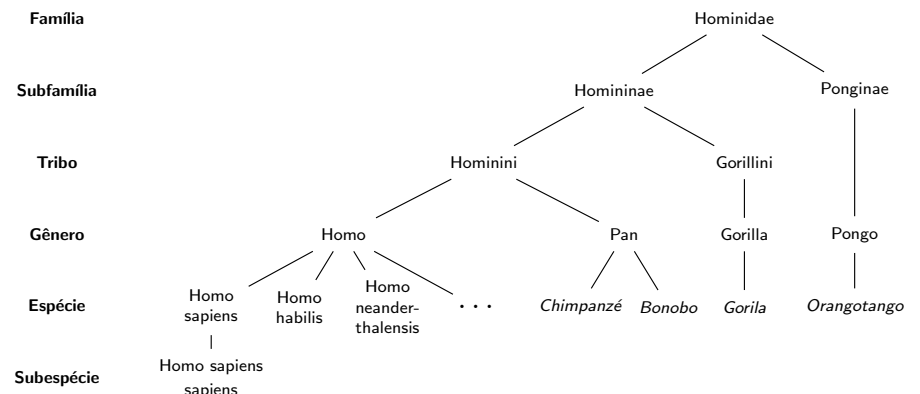
- ▶ A árvore representa as decisões tomadas por um algoritmo de ordenação de três elementos usando operações de comparação; V: verdadeiro, F: falso.
- ▶ Devido à natureza das comparações, a árvore é binária.

Exemplo de árvore geral 1: descendentes



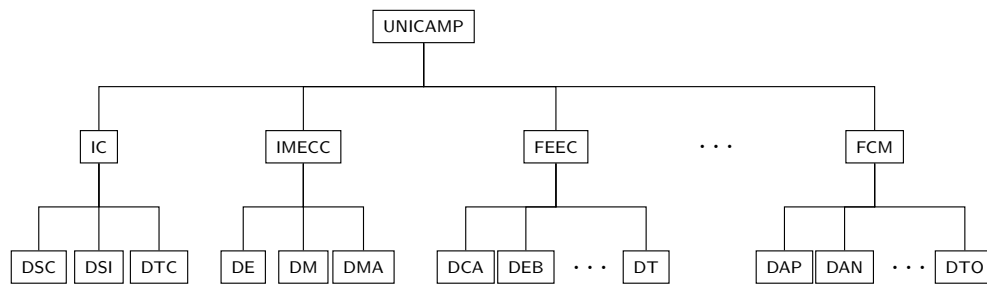
- ▶ A árvore é incompleta e não necessariamente exata.
- ▶ Cada elemento pode ter um número variável de sucessores: *árvore geral*.

Exemplo de árvore geral 2: descendentes



- ▶ Árvore da família *Hominidae* determinada por comparação de DNA de várias espécies (incompleta).
- ▶ Cada elemento pode ter um número variável de sucessores: *árvore geral*.

Exemplo de árvore geral 3: organograma



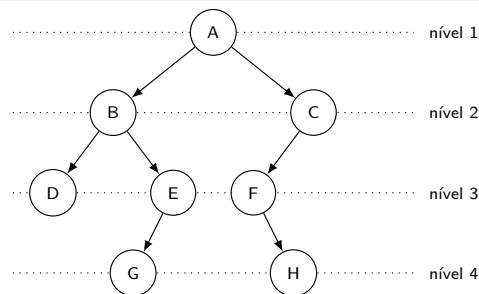
Obs.: A UNICAMP tem 23 unidades acadêmicas. Algumas unidades têm mais de 10 departamentos.

Definição de árvore binária

Uma *árvore binária* é um conjunto de *nós* (unidades de informação) que:

- ▶ ou é vazio (*árvore binária vazia*)
- ▶ ou contém um nó especial denominado *raiz da árvore* e o resto do conjunto está particionado em duas árvores binárias disjuntas (possivelmente vazias), denominadas *subárvore esquerda* e *subárvore direita*.

Representação gráfica, convenções e conceitos



Raiz da árvore: *A*

Filho esquerdo de *A*: *B*

Pai de *F*: *C*

Descendentes de *B*: *B*, *D*, *E* e *G*

Folhas: *D*, *G* e *H*

Níveis: indicados na figura

Subárvores binárias vazias: 9

Filho direito de *A*: *C*

Irmão de *E*: *D*

Antepassados de *H*: *H*, *F*, *C* e *A*

Nós internos: todos exceto as folhas

Altura (profundidade) – nível máximo: 4

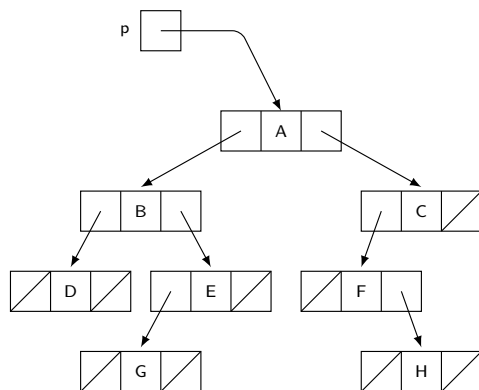
Subárvores binárias não vazias: 7

Obs.: Alguns autores começam a numeração dos níveis a partir de zero.

Fatos sobre árvores binárias

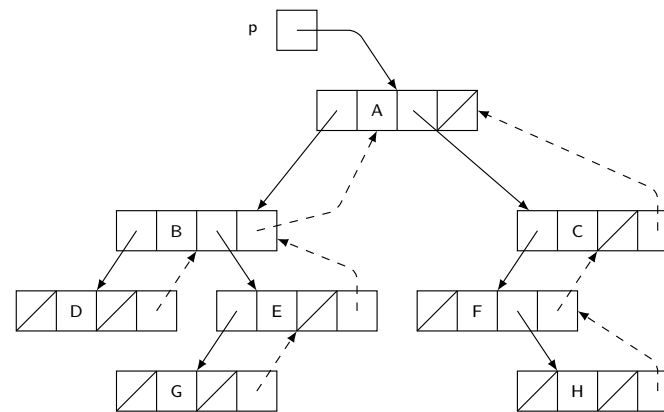
- ▶ Uma árvore binária de altura h tem:
 - ▶ no mínimo h nós
 - ▶ no máximo $2^h - 1$ nós
- ▶ Uma árvore binária com n nós tem:
 - ▶ altura máxima n
 - ▶ altura mínima $\lceil \log_2(n+1) \rceil$
 - ▶ subárvores vazias: $n+1$
 - ▶ subárvores não vazias: $n-1$ (se $n > 0$)

Representação ligada comum



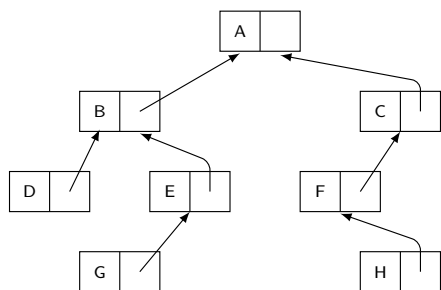
O acesso a todos os nós da árvore pode ser realizado através de um apontador para a raiz.

Representação ligada com três apontadores



O terceiro apontador possibilita descer e subir pela estrutura, analogamente às listas duplamente ligadas.

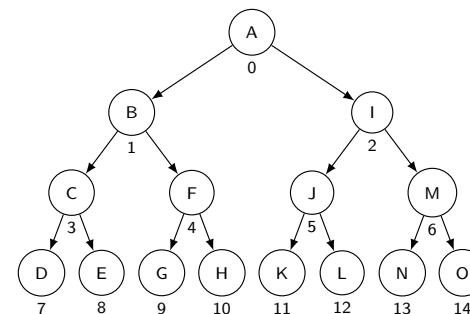
Representação com o campo *pai* apenas



Problemas:

- ▶ É necessário haver acesso (apontadores) pelo menos a todas as folhas.
- ▶ Não é possível distinguir entre os filhos esquerdos e direitos sem que haja alguma informação adicional.

Representação sequencial: árvores binárias completas

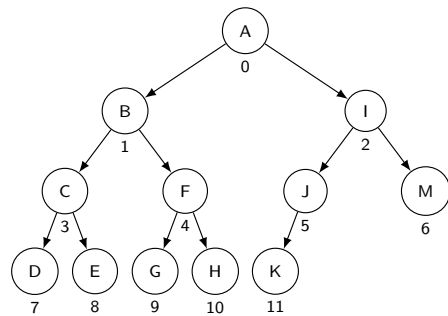


$$\text{Nó } n: \begin{cases} \text{filho esquerdo: } 2n + 1 \ (n \geq 0) \\ \text{filho direito: } 2n + 2 \ (n \geq 0) \\ \text{pai: } \lfloor (n - 1) / 2 \rfloor \ (n > 0) \end{cases}$$

A	B	I	C	F	J	M	D	E	G	H	K	L	N	O
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Esta representação não usa apontadores.

Representação sequencial: árvores binárias quase completas



A	B	I	C	F	J	M	D	E	G	H	K
0	1	2	3	4	5	6	7	8	9	10	11

Percursos em profundidade

► Pré-ordem:

Visitar a raiz

Percorrer a subárvore esquerda em pré-ordem

Percorrer a subárvore direita em pré-ordem

► Pós-ordem:

Percorrer a subárvore esquerda em pós-ordem

Percorrer a subárvore direita em pós-ordem

Visitar a raiz

► Inordem (ou ordem simétrica):

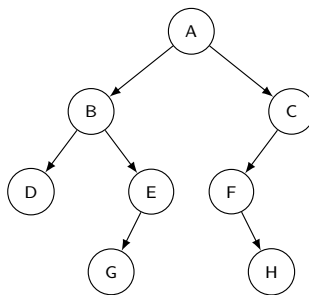
Percorrer a subárvore esquerda em inordem

Visitar a raiz

Percorrer a subárvore direita em inordem

Alguns autores utilizam denominações diferentes.

Exemplos de percurso em profundidade



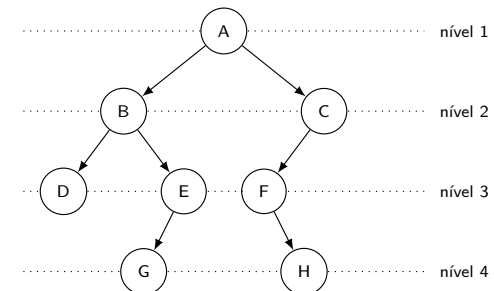
Pré-ordem: A,B,D,E,G,C,F,H

Pós-ordem: D,G,E,B,H,F,C,A

Inordem: D,B,G,E,A,F,H,C

Percurso em largura

Percurso por níveis, da esquerda para a direita:



Percurso: A,B,C,D,E,F,G,H

Implementação recursiva de percursos

```
typedef struct NoArvBin {  
    T info;  
    struct NoArvBin *esq, *dir;  
} NoArvBin, *ArvBin;
```

```
void InOrdem(ArvBin p) {  
    if (p!=NULL) {  
        InOrdem(p->esq);  
        Visita(p);  
        InOrdem(p->dir);  
    }  
} /* InOrdem */
```

```
void PreOrdem(ArvBin p) {  
    if (p!=NULL) {  
        Visita(p);  
        PreOrdem(p->esq);  
        PreOrdem(p->dir);  
    }  
} /* PreOrdem */
```

```
void PosOrdem(ArvBin p) {  
    if (p!=NULL) {  
        PosOrdem(p->esq);  
        PosOrdem(p->dir);  
        Visita(p);  
    }  
} /* PosOrdem */
```

Eliminação da recursão caudal

```
void PreOrdem(ArvBin p) {  
    if (p!=NULL) {  
        Visita(p);  
        PreOrdem(p->esq);  
        PreOrdem(p->dir);  
    }  
} /* PreOrdem */
```

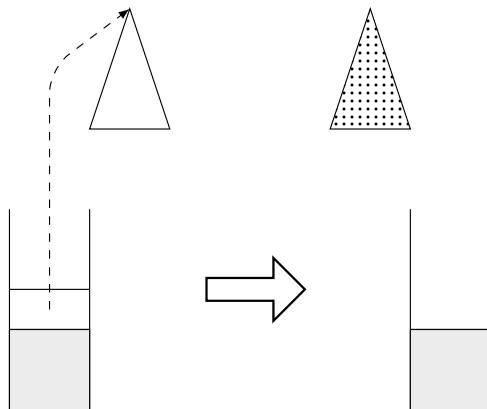
```
void PreOrdem(ArvBin p) {  
    while (p!=NULL) {  
        Visita(p);  
        PreOrdem(p->esq);  
        p = p->dir;  
    }  
} /* PreOrdem */
```

- Transformação análoga pode ser feita para a *inordem*.
- E a *pós-ordem*?

```
void PosOrdem(ArvBin p) {  
    if (p!=NULL) {  
        PosOrdem(p->esq);  
        PosOrdem(p->dir);  
        Visita(p);  
    }  
} /* PosOrdem */
```

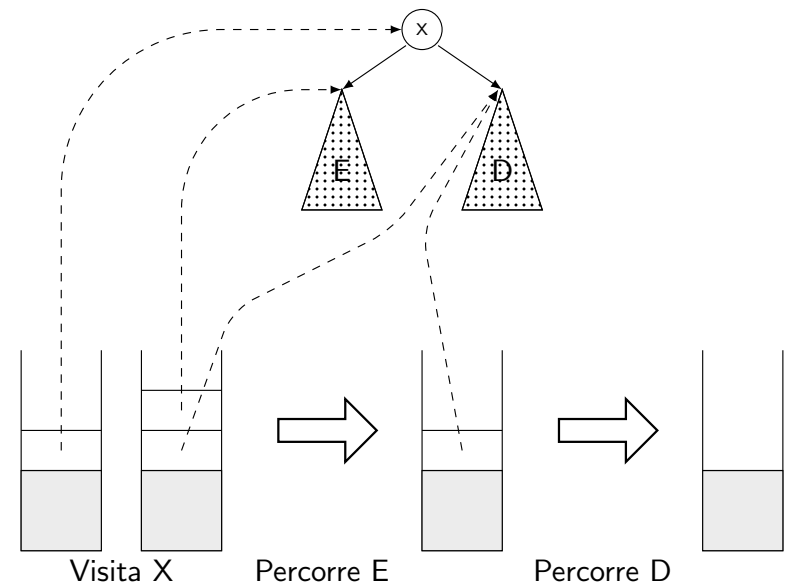
Não há recursão caudal!

Percurso em pré-ordem, usando uma pilha explícita



A figura indica a situação inicial e final do percurso de uma árvore arbitrária (pode ser vazia). Inicialmente, o apontador para a árvore deve estar no topo da pilha. Terminado o percurso, a pilha terá um elemento a menos.

Percurso em pré-ordem, usando uma pilha explícita (cont.)

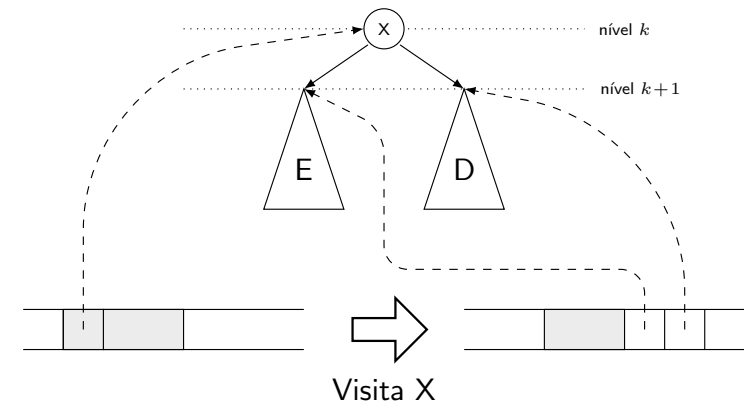


Percurso em pré-ordem, usando uma pilha (cont.)

```
void PreOrdem(ArvBin p) {
    Pilha pl;
    InicializaPilha(&pl);
    Empilha(&pl, p);
    do {
        Desempilha(&pl, &p);
        if (p!=NULL) {
            Visita(p);
            Empilha(&pl, p->dir);
            Empilha(&pl, p->esq);
        }
    } while (!PilhaVazia(pl));
} /* PreOrdem */
```

Percurso em largura, usando uma fila

Os nós da árvore a serem visitados são guardados numa fila.



Percurso em largura, usando uma fila (cont.)

```
void Largura(ArvBin p) {
    Fila fl;
    InicializaFila(&fl);
    InsereFila(&fl, p);
    do {
        RemoveFila(&fl, &p);
        if (p!=NULL) {
            Visita(p);
            InsereFila(&fl, p->esq);
            InsereFila(&fl, p->dir);
        }
    } while (!FilaVazia(fl));
} /* Largura */
```

Comparação dos percursos em pré-ordem e em largura

```
void PreOrdem(ArvBin p) {
    Pilha pl;
    InicializaPilha(&pl);
    Empilha(&pl, p);
    do {
        Desempilha(&pl, &p);
        if (p!=NULL) {
            Visita(p);
            Empilha(&pl, p->dir);
            Empilha(&pl, p->esq);
        }
    } while (!PilhaVazia(pl));
} /* PreOrdem */
```

```
void Largura(ArvBin p) {
    Fila fl;
    InicializaFila(&fl);
    InsereFila(&fl, p);
    do {
        RemoveFila(&fl, &p);
        if (p!=NULL) {
            Visita(p);
            InsereFila(&fl, p->esq);
            InsereFila(&fl, p->dir);
        }
    } while (!FilaVazia(fl));
} /* Largura */
```

Quase idênticos, exceto a troca de esquerda pela direita!

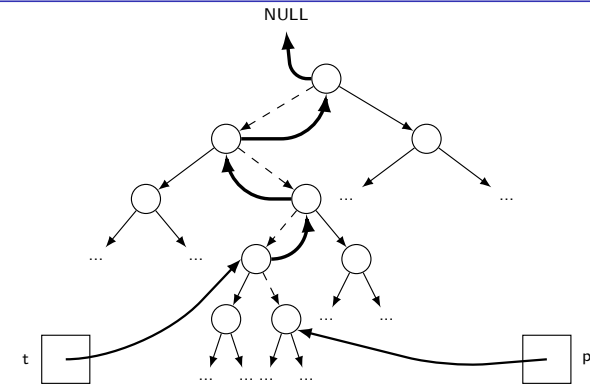
Preordem com pilha otimizada

Evita Empilha/Desempilha de um mesmo nó:

```
void PreOrdem(ArvBin p) {
    Pilha pl;
    Boolean fim = false;
    InicializaPilha(&pl);
    do {
        if (p!=NULL) {
            Visita(p);
            if (p->dir!=NULL)
                Empilha(pl, p->dir);
            p = p->esq;
        } else if (PilhaVazia(pl))
            fim = true
        else
            Desempilha(pl, &p);
    } while (!fim);
} /* PreOrdem */
```

```
void PreOrdem(ArvBin p) {
    Pilha pl;
    InicializaPilha(&pl);
    Empilha(&pl, p);
    do {
        Desempilha(&pl, &p);
        if (p!=NULL) {
            Visita(p);
            Empilha(&pl, p->dir);
            Empilha(&pl, p->esq);
        } while (!PilhaVazia(pl));
    } /* PreOrdem */
```

Pré-ordem com pilha embutida: Deutsch, Schorr e Waite



- ▶ A variável p aponta para a subárvore a ser percorrida.
- ▶ A variável t aponta para o topo de uma pilha formada pelos nós que levam ao nó p (apontadores invertidos; t aponta para o pai de p).
- ▶ Cada nó deverá conter uma marca indicando qual dos dois apontadores está invertido.
- ▶ A função seguinte implementa os três percursos em profundidade.

Pré-ordem com pilha embutida (cont.)

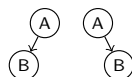
```
void DSW(ArvBin p) {
    ArvBin t = NULL; ArvBin q; Boolean sobe;
    do {
        while (p!=NULL) { /* à esquerda */
            PreVisita(p); p->marca = MarcaEsq; q = p->esq;
            p->esq = t; t = p; p = q;
        }
        sobe = true;
        while (sobe && (t!=NULL)) {
            switch (t->marca) {
                case MarcaEsq: /* à direita */
                    InVisita(t); sobe = false; t->marca = MarcaDir;
                    q = p; p = t->dir; t->dir = t->esq; t->esq = q;
                    break;
                case MarcaDir: /* sobe */
                    PosVisita(t); q = t->dir; t->dir = p; p = t; t = q;
                    break;
            }
        }
    } while (t!=NULL);
}
```

Desafios:

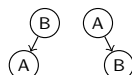
- ▶ melhorar a pré-ordem com pilha otimizada
- ▶ inordem com pilha otimizada
- ▶ pós-ordem com pilha otimizada

Reconstrução de árvores binárias

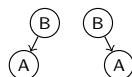
Pré-ordem AB :



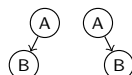
Inordem AB :



Pós-ordem AB :



Pré-ordem AB e pós-ordem BA :



Conclusão: uma única ordem e a combinação de pré- e pós-ordens não determinam a árvore de maneira única.

Reconstrução de árvores binárias (cont.)

Verifica-se facilmente que a pré-ordem (ou a pós-ordem) combinada com a inordem determinam, de maneira única, a forma da árvore. No caso da pré-ordem e inordem, pode-se seguir o seguinte algoritmo:

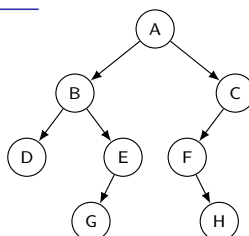
- ▶ a partir da pré-ordem, determine a raiz da árvore
- ▶ dada a raiz, ela separa a inordem em inordens das suas subárvores esquerda e direita
- ▶ a partir da pré-ordem, são determinadas as pré-ordens das subárvores que têm os mesmos comprimentos das respectivas inordens
- ▶ recursivamente são reconstruídas as subárvores

O caso da pós-ordem é análogo.

Representações externas de árvores binárias

- ▶ percursos canônicos: inordem e pré- (ou pós-) ordem (já visto):

$DBGEAFHC$
 $ABDEGCFH$



- ▶ percurso canônico com indicadores de subárvores (pré-ordem):

$A_{11}B_{11}D_{00}E_{10}G_{00}C_{10}F_{01}H_{00}$

O índice 0 indica a ausência e 1 indica a existência de filho esquerdo ou direito.

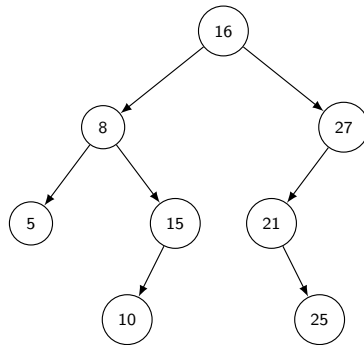
- ▶ descrição parentética (inordem):

$(((((D()))B(((G()))E()))A(((F((H()))C()))))$

$()$ representa uma árvore vazia; $(\alpha X \beta)$ representa uma árvore de raiz X e subárvores descritas pelas cadeias α e β .

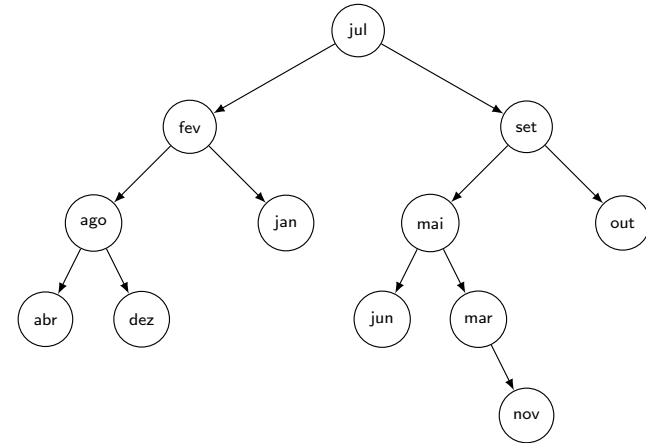
Árvores binárias de busca

Exemplo de árvore de busca: números



Para qualquer nó da árvore, os elementos da sua subárvore esquerda (direita) são menores ou iguais (maiores ou iguais) do que o elemento do nó.

Exemplo de árvore de busca: nomes



Para qualquer nó da árvore, os elementos da sua subárvore esquerda (direita) precedem (seguem) em ordem alfabética o elemento do nó.

Busca em árvore de busca

Versão recursiva:

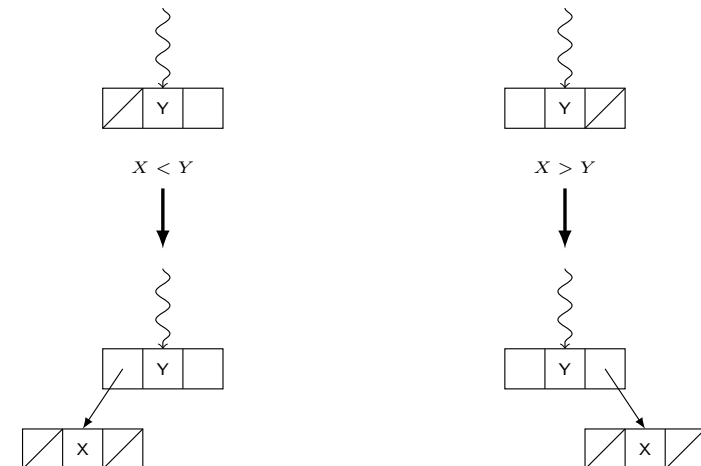
```
Boolean Busca(ArvBin p, T x) {  
    if (p==NULL)  
        return false;  
    if (x==p->info)  
        return true;  
    if (x<p->info)  
        return Busca(p->esq, x);  
    else  
        return Busca(p->dir, x);  
} /* Busca */
```

Número de operações: no pior caso, proporcional à altura da árvore.

Exercício: escrever a versão iterativa da função (as duas chamadas recursivas são caudais).

Inserção em árvore de busca

A inserção de um valor X cria uma nova folha em lugar de uma subárvore vazia. O ponto de inserção é determinado pelo percurso da árvore usando a propriedade de árvores de busca.



Inserção recursiva

```
Boolean InsereArvBusca(ArvBin *p, T x) {  
    /* Versão recursiva */  
    if ((*p)==NULL) {  
        *p = malloc(sizeof(NoArvBin));  
        (*p)->esq = (*p)->dir = NULL;  
        (*p)->info = x;  
        return true;  
    } else {  
        T info = (*p)->info;  
        if (x<info)  
            return InsereArvBusca(&((*p)->esq), x);  
        else if (x>info)  
            return InsereArvBusca(&((*p)->dir), x);  
        else  
            return false;  
    }  
}
```

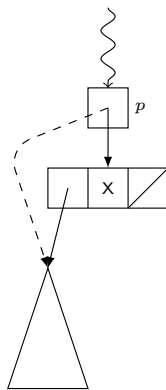
- Note-se o uso de passagem de parâmetro p por referência.
- Esta versão apresenta somente recursão caudal que pode ser facilmente eliminada.

Inserção iterativa

```
Boolean InsereArvBusca(ArvBin *p, T x) {  
    /* Versão iterativa */  
    T info;  
    while ((*p)!=NULL) {  
        info = (*p)->info;  
        if (x<info)  
            p = &((*p)->esq);  
        else if (x>info)  
            p = &((*p)->dir);  
        else  
            return false;  
    }  
    *p = malloc(sizeof(NoArvBin));  
    (*p)->esq = (*p)->dir = NULL;  
    (*p)->info = x;  
    return true;  
}
```

Remoção em árvore de busca

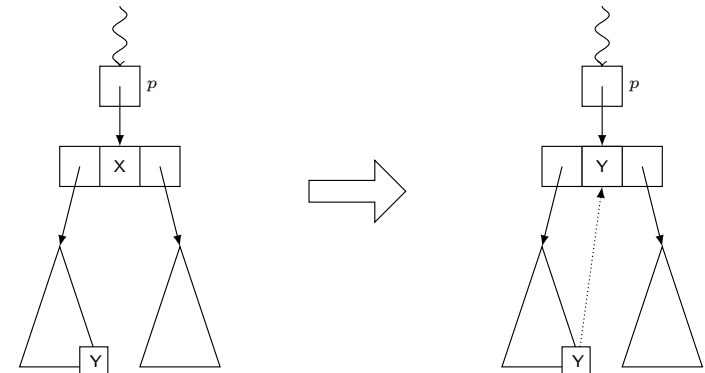
Caso 1: pelo menos uma das subárvores é vazia:



- p é o endereço do campo ou da variável que contém o apontador para o nó com a informação X .
- O caso de ambas as subárvores vazias também está coberto.
- O caso de subárvore esquerda vazia mas direita não vazia é análogo.
- O nó com a informação X pode ser liberado.

Remoção em árvore de busca (cont.)

Caso 2: as duas subárvores são não vazias



- Substituir a informação X por Y – o valor máximo contido na subárvore esquerda (ou mínimo na subárvore direita).
- Remover o nó que originalmente continha Y (sua subárvore direita é vazia – aplica-se o caso 1).
- Implementação: exercício.

Inserções e remoções em árvores binárias de busca

- ▶ Problema: a altura da árvore pode crescer muito já que numa árvore com n nós:
 - ▶ altura máxima n
 - ▶ altura mínima $\lceil \log_2(n+1) \rceil$
- ▶ Se $n \approx 1.000$:
 - ▶ altura máxima 1.000
 - ▶ altura mínima 10
- ▶ Se $n \approx 1.000.000$:
 - ▶ altura máxima 1.000.000
 - ▶ altura mínima 20
- ▶ O pior caso ocorre, por exemplo, quando a inserção é feita em ordem (crescente ou decrescente)

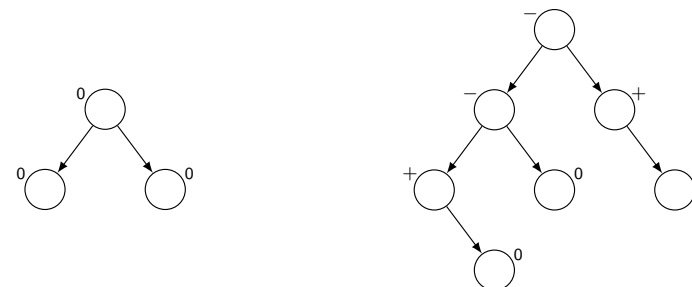
Balanceamento de árvores

- ▶ Algoritmos óbvios de inserção e remoção não garantem balanceamento
- ▶ Balanceamento perfeito (altura mínima):
 - ▶ eficiência de busca: $O(\log n)$
 - ▶ eficiência de inserção: $O(n)$ – inaceitável
- ▶ Balanceamento aproximado:
 - ▶ árvores AVL – eficiência de busca, inserção e remoção: $O(\log n)$
 - ▶ árvores rubro-negras – eficiência de busca, inserção e remoção: $O(\log n)$

Árvores de altura balanceada (AVL)

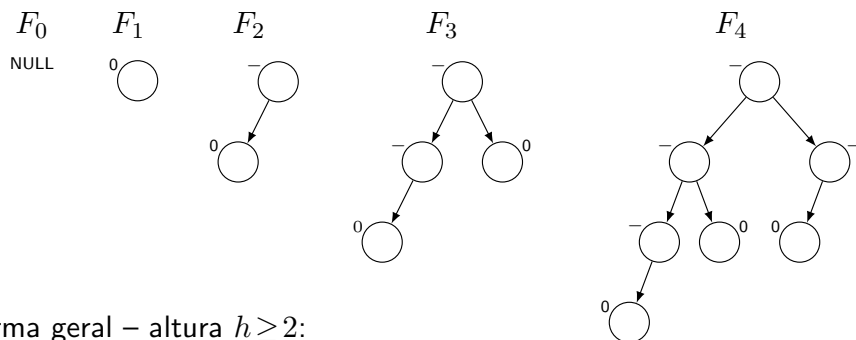
- ▶ Autores: G. M. Adel'son-Vel'skiĭ e E. M. Landis (1962)
- ▶ Uma árvore binária de busca é do tipo AVL se ela é vazia, ou então, se para todos os seus nós a diferença de alturas entre as subárvores esquerda e direita é no máximo 1, em valor absoluto.
- ▶ A diferença entre as alturas direita e esquerda é chamada *fator de balanceamento*.

Exemplos de árvores AVL

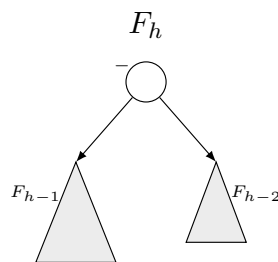


Note-se que a primeira árvore é de altura mínima enquanto que a segunda não é.

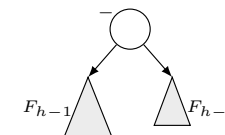
Pior caso de desbalanceamento: árvores de Fibonacci



Forma geral – altura $h \geq 2$:



Árvores de Fibonacci: propriedades



- ▶ Para uma dada altura h , a árvore contém o número mínimo de nós possível preservando ainda a propriedade AVL.
- ▶ Qualquer outra árvore AVL com o mesmo número de nós tem altura menor ou igual – este é o pior caso.
- ▶ Número de nós de F_h : $N(h) = N(h-1) + N(h-2) + 1$, $h \geq 2$
- ▶ Demonstra-se por indução: $N(h) = f_{h+2} - 1$, onde f_i é o i -ésimo número de Fibonacci.
- ▶ Usando a aproximação $f_i \approx ((1 + \sqrt{5})/2)^i / \sqrt{5}$ obtém-se: $h \approx 1,44 \log_2(n+2)$ (no máximo).
- ▶ Operação de busca usará $O(\log n)$ operações.
- ▶ A ser visto: inserção e remoção também usarão $O(\log n)$ operações.

Inserção em árvores AVL

A explicação a seguir supõe que a inserção é realizada por uma função recursiva cujo cabeçalho é:

```
Boolean BuscaInsere(ArvAVL *p, T x, Boolean *alt);
```

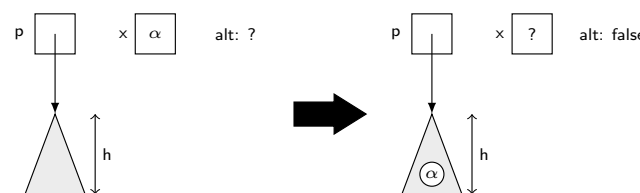
onde

- ▶ p : endereço da variável que contém o apontador para a árvore
- ▶ x : valor a ser inserido de algum tipo T conveniente
- ▶ alt : endereço da variável na qual é devolvida a informação que indica se a árvore aumentou ou não de altura
- ▶ se não houver aumento de altura numa chamada recursiva, o resto da árvore não sofre mudança
- ▶ conforme será visto, o aumento da altura será no máximo de um e pode acontecer somente numa árvore vazia ou então cuja raiz tem fator de balanceamento igual a zero; neste caso, o fator resultante será diferente de zero, exceto quando a árvore era vazia.

O valor devolvido pela função indica se a inserção foi efetivamente realizada ou se o elemento x já pertencia à árvore.

Inserção em árvores AVL (cont.)

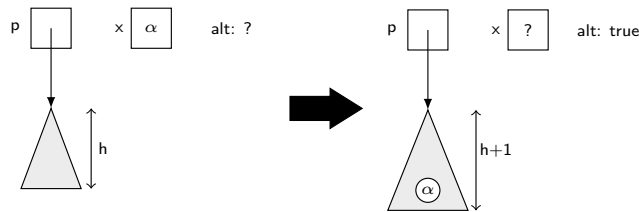
Explicação geral: caso de chamada recursiva sem aumento de altura



Neste caso, não haverá mais modificações na árvore.

Inserção em árvores AVL (cont.)

Explicação geral: caso de chamada recursiva com aumento de altura



Neste caso, haverá modificação no nó corrente com possível propagação para as chamadas anteriores.

Inserção em árvores AVL (cont.)

Caso 1: Inserção em árvore vazia:

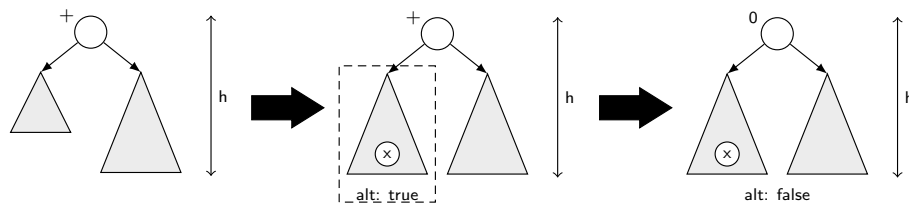


Neste caso a altura h aumenta. Este fato será propagado no retorno da função através de valor verdadeiro da variável *alt*.

Nos casos seguintes, será suposto sempre que a inserção foi realizada na subárvore esquerda; o caso da inserção do lado direito é análogo.

Inserção em árvores AVL (cont.)

Caso 2: Inserção do lado mais baixo:

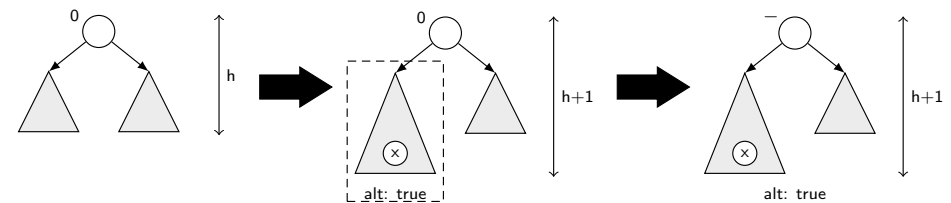


O conteúdo do retângulo representa o resultado da chamada recursiva. O fator de balanceamento será modificado somente se a árvore esquerda aumentou de tamanho.

Neste caso a altura permanece inalterada. Este fato será propagado no retorno da função através de valor falso da variável *alt*. Como consequência, o processo de inserção para (exceto os retornos).

Inserção em árvores AVL (cont.)

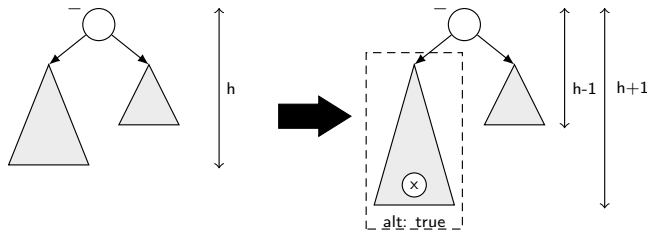
Caso 3: Inserção quando as duas alturas são iguais



Neste caso, se houve aumento de altura na chamada recursiva, a altura total também aumentará. Este fato será propagado no retorno da função através de valor verdadeiro da variável *alt*.

Inserção em árvores AVL (cont.)

Caso 4: Inserção do lado mais alto

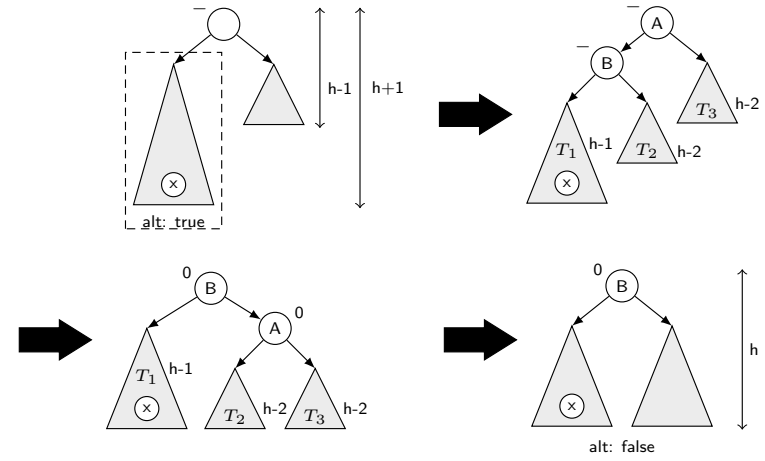


Neste caso, se houve aumento de altura na chamada recursiva, a árvore deixará de ser do tipo AVL. Haverá então dois subcasos, dependendo do lado da subárvore esquerda em que houve inserção. A identificação do subcaso será feita pelo valor do fator de balanceamento final da subárvore que aumentou de altura durante a chamada recursiva.

Nos dois casos haverá rearranjos convenientes mas locais da árvore.

Inserção em árvores AVL (cont.)

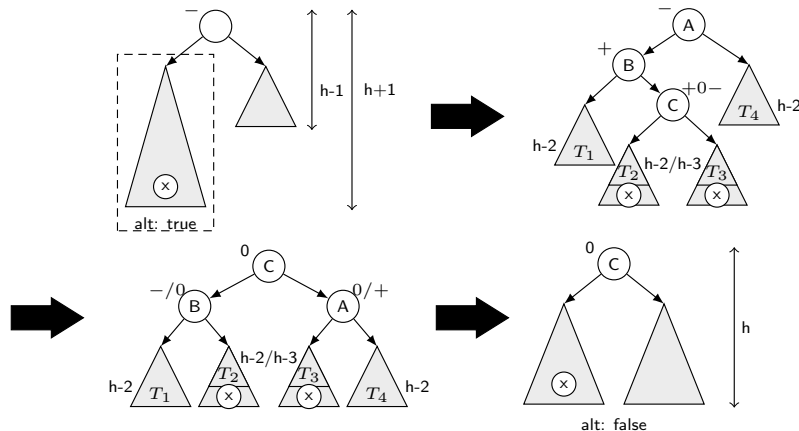
Caso 4a: inserção do lado esquerdo da subárvore (rotação LL)



Neste caso é realizada uma transformação denominada rotação simples LL (esquerda, esquerda). A altura final permanece inalterada e a variável *alt* recebe valor falso.

Inserção em árvores AVL (cont.)

Caso 4b: inserção do lado direito da subárvore (rotação LR)



Neste caso, a inserção pode ter sido realizada na subárvore esquerda ou direita do lado que cresceu, ou então no próprio nó C (quando $h=2$). Os fatores de balanceamento finais dependem disto, mas o da raiz será 0. A transformação é denominada rotação dupla LR (esquerda, direita). A altura final permanece inalterada e a variável *alt* recebe valor falso.

Função de inserção em árvores AVL

```
Boolean BuscaInsere(ArvAVL *p, T x, Boolean *alt) {
    /* Devolve 'true' ou 'false' conforme houve ou não inserção;
       se houve inserção, 'alt' indica se houve aumento da altura.
    */
    if (*p==NULL) {
        *p = malloc(sizeof(NoArvAVL));
        (*p)->esq = (*p)->dir = NULL; (*p)->info = x;
        (*p)->bal = zero; *alt = true;
        return true;
    } else {
        T info = (*p)->info;
        if (x==info)
            return false;
        else if (x<info) { /* desce à esquerda */
            Boolean res = BuscaInsere(&((*p)->esq), x, alt);
            if (!res)
                return false;
        }
    }
}
```

Função de inserção em árvores AVL (cont.)

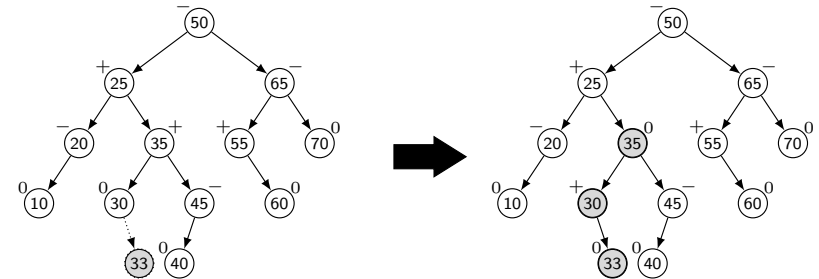
```

if (*alt) { /* aumento de altura */
    ArvAVL p1, p2;
    switch ((*p)->bal) {
        case mais: (*p)->bal = zero; *alt = false; break;
        case zero: (*p)->bal = menos; break;
        case menos:
            p1 = (*p)->esq;
            if (p1->bal==menos) {
                /* Rotação simples LL */
            } else {
                /* Rotação dupla LR */
            }
            p1->bal = zero; *alt = false;
            break;
    }
}
return true;
} else {
    /* desce à direita - análogo */
}
}

```

Exemplos de inserção em árvores AVL

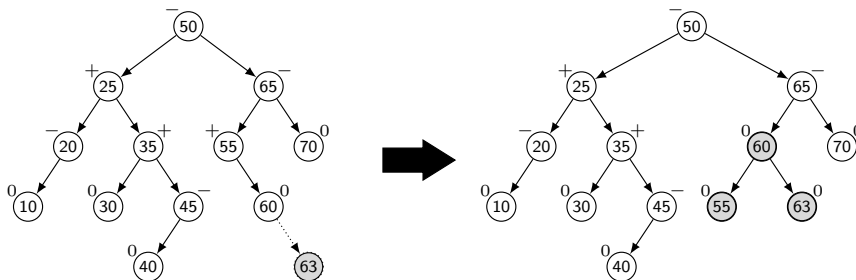
Inserção de 33:



Neste caso, houve uma inserção simples e a mudança de fatores de balanceamento afetou os nós marcados.

Exemplos de inserção em árvores AVL (cont.)

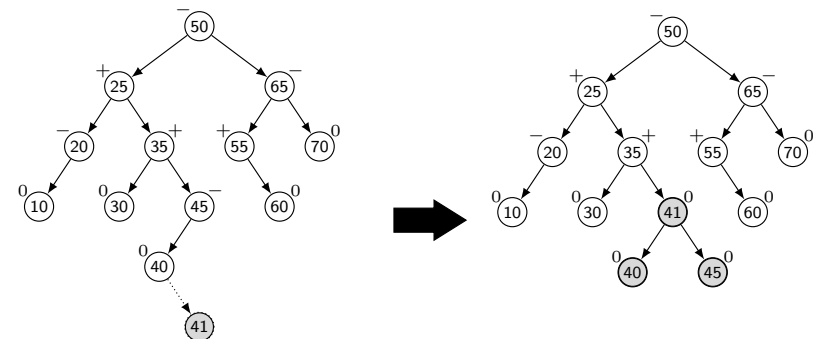
Inserção de 63:



Neste caso, a inserção causou uma rotação simples do tipo RR, afetando os nós marcados.

Exemplos de inserção em árvores AVL (cont.)

Inserção de 41:



Neste caso, a inserção causou uma rotação dupla do tipo LR, afetando os nós marcados.

Remoção em árvores AVL

1. Transformação em remoção de uma folha - três casos:

- ▶ o nó tem grau zero: já é uma folha
- ▶ o nó tem grau um: pela propriedade AVL, a sua única subárvore é necessariamente constituída por uma folha, cujo valor é copiado para o nó pai; o nó a ser eliminado é a folha da subárvore
- ▶ o nó tem grau dois: o seu valor é substituído pelo maior valor contido na sua subárvore esquerda (ou o menor valor contido na sua subárvore direita); o nó que continha o menor (ou maior) valor copiado tem necessariamente grau zero ou um, recaindo num dos casos anteriores.

2. Remoção propriamente dita.

3. O algoritmo de remoção será apresentado novamente como uma função recursiva que indica se houve diminuição da altura da árvore após a remoção. Serão estudados apenas os casos de remoção do lado esquerdo; os outros são análogos.

4. A implementação do algoritmo é um exercício.

Remoção em árvores AVL (cont.)

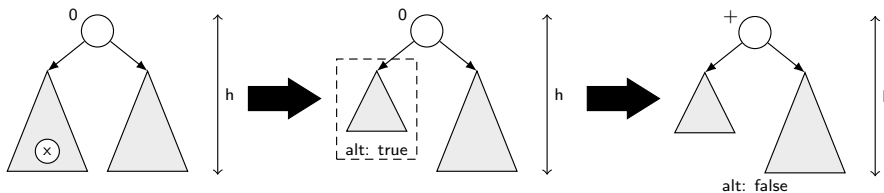
Caso 1: Remoção de uma folha



Neste caso a altura h diminui. Este fato será propagado no retorno da função através de valor verdadeiro da variável *alt*.

Remoção em árvores AVL (cont.)

Caso 2: Remoção quando as duas alturas são iguais

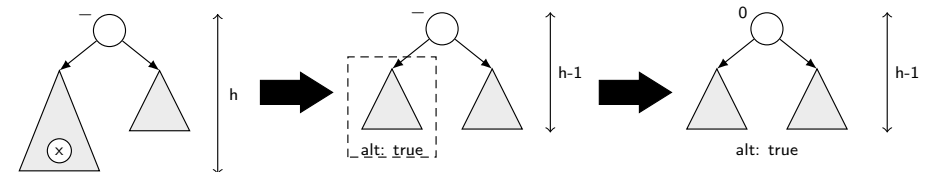


O conteúdo do retângulo representa o resultado da chamada recursiva. O fator de balanceamento será modificado somente se a árvore esquerda diminuiu de tamanho.

Neste caso, mesmo que haja diminuição de altura na chamada recursiva, a altura total permanece a mesma. Este fato será propagado no retorno da função através de valor falso da variável *alt*.

Remoção em árvores AVL (cont.)

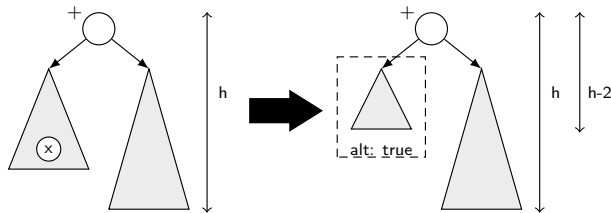
Caso 3: Remoção do lado mais alto



Neste caso, se a chamada recursiva indica diminuição da altura da subárvore, a altura final da árvore também diminui e o processo continua.

Remoção em árvores AVL (cont.)

Caso 4: Remoção do lado mais baixo

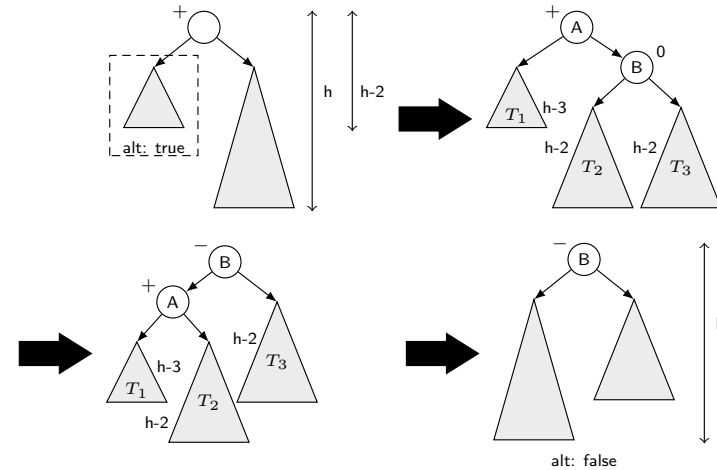


Caso a subárvore esquerda tenha sua altura diminuída, a árvore deixa de ser do tipo AVL. Há três subcasos, dependendo do fator de balanceamento do filho direito da raiz.

Note-se que, neste caso, tem-se $h \geq 3$.

Remoção em árvores AVL (cont.)

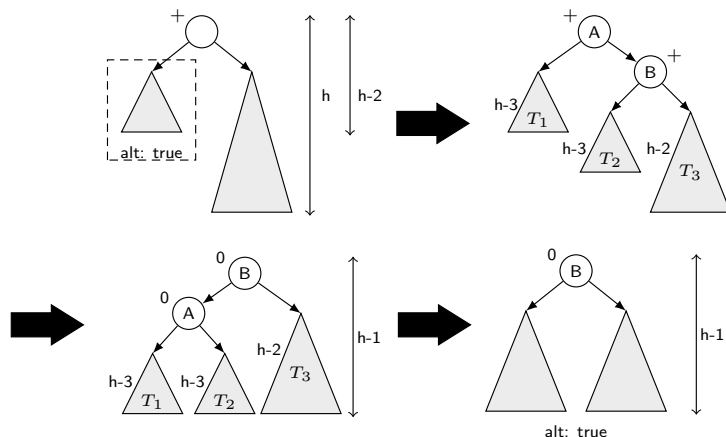
Caso 4a: Fator de balanceamento 0 (rotação RR)



Neste caso é realizada uma transformação denominada rotação simples RR (direita, direita). A altura final permanece inalterada e a variável *alt* recebe valor falso. O processo de ajuste da árvore para.

Remoção em árvores AVL (cont.)

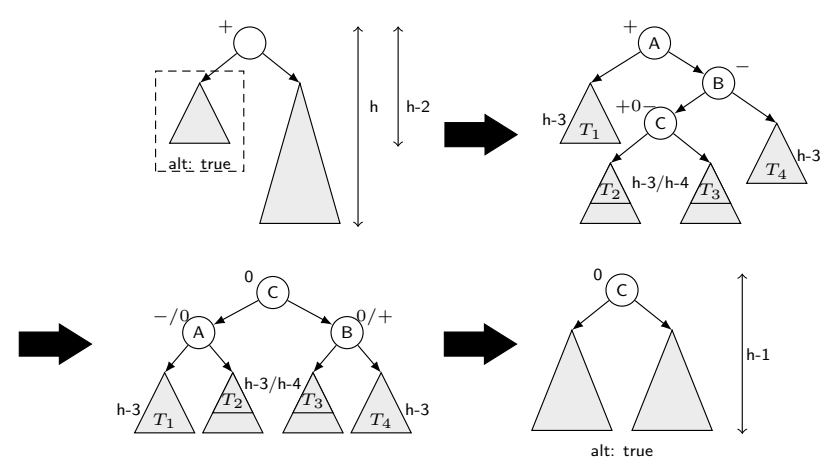
Caso 4b: Fator de balanceamento +1 (rotação RR)



Neste caso também é realizada a transformação denominada rotação simples RR (direita, direita). A altura final diminui e a variável *alt* recebe o valor verdadeiro. O processo de ajuste da árvore continua.

Remoção em árvores AVL (cont.)

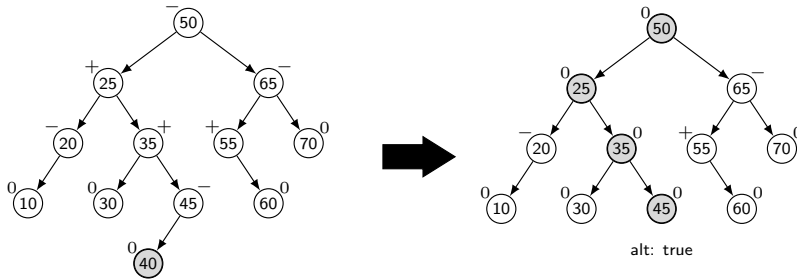
Caso 4c: Fator de balanceamento -1 (rotação RL)



Neste caso também é realizada uma transformação denominada rotação dupla RL (direita, esquerda). A altura final diminui e a variável *alt* recebe o valor verdadeiro. O processo de ajuste da árvore continua.

Exemplos de remoção em árvores AVL

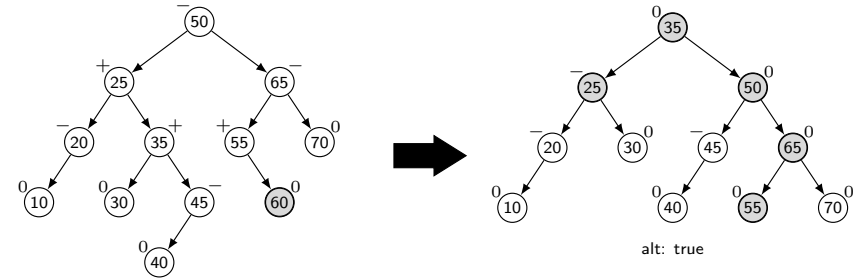
Remoção de 40:



Neste caso, houve uma remoção simples e a mudança de fatores de balanceamento afetou os nós marcados.

Exemplos de remoção em árvores AVL (cont.)

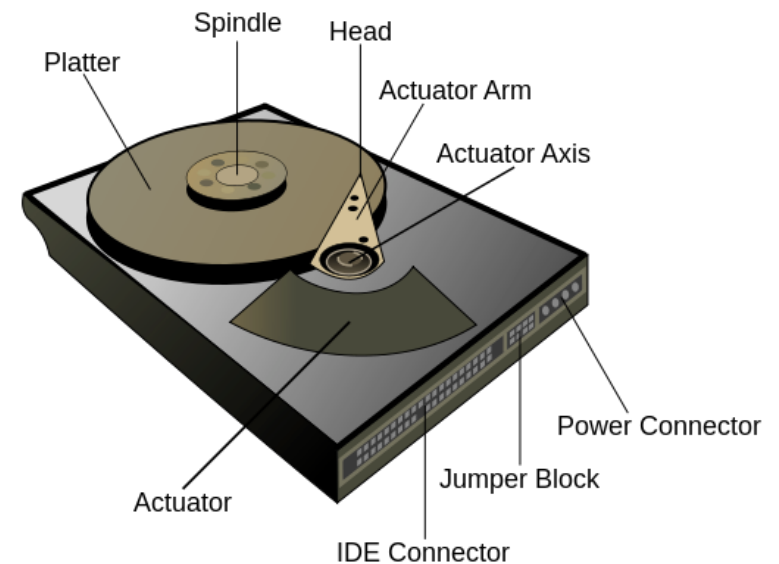
Remoção de 60:



Neste caso, a remoção causou, após a volta da chamada com a raiz original, uma rotação dupla do tipo LR, afetando os nós marcados.

Árvores do tipo B (B trees)

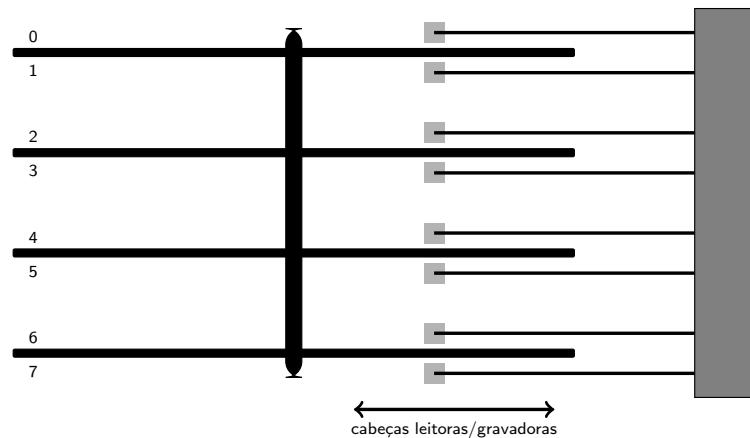
Discos magnéticos



Fonte: Wikipedia: Hard disk drive

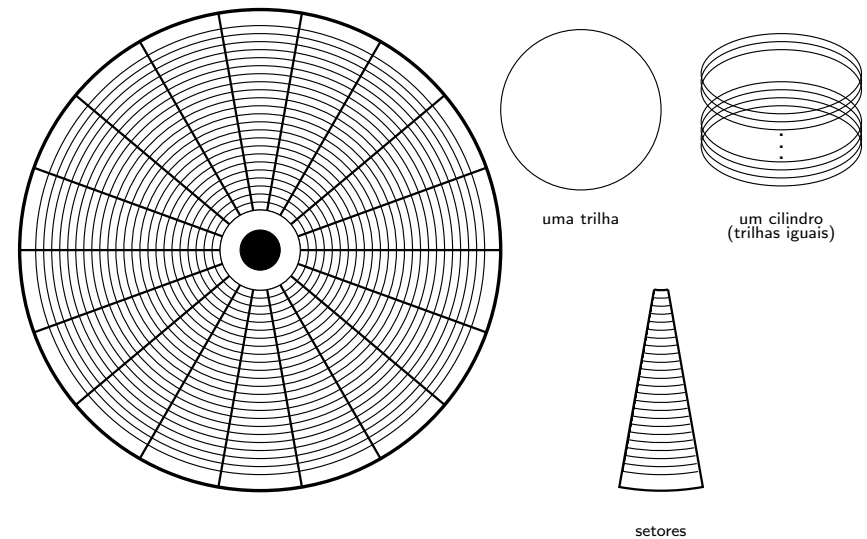
Discos magnéticos (cont.)

Esboço esquemático do corte vertical de uma unidade com quatro discos (oito superfícies):



Discos magnéticos (cont.)

Esboço esquemático de uma superfície de um disco:



Discos magnéticos (cont.)

Dados para um disco fictício de 150 *gigabytes*:

- ▶ 6 cabeças leitoras/gravadoras
- ▶ 30.000 trilhas (5.000 por superfície)
- ▶ 2000 setores por trilha
- ▶ 512 *bytes* por setor (unidade mínima de endereçamento)
- ▶ tempo médio de busca da trilha endereçada (*seek*): ΔS (10 milissegundos)
- ▶ tempo médio de latência – espera pelo setor endereçado: ΔL (5 milissegundos)
- ▶ tempo de transferência de dados: ΔT (100 megabytes/segundo)
- ▶ Estes tempos são várias ordens de grandeza maiores do que tempo de acesso à memória RAM (tipicamente 100.000 vezes).
- ▶ Número de acessos: altura da árvore – $\log_2 n$ não é mais aceitável
- ▶ Solução: $\log_k n$, com $k \gg 2$

Árvores B

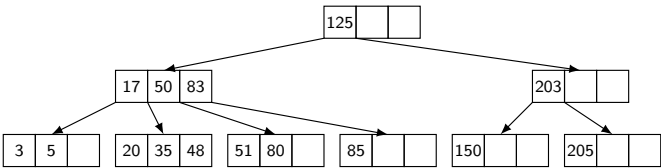
- ▶ Autores: Rudolf Bayer e Ed McCreight (1971)
- ▶ T é uma *árvore B* de ordem $b \geq 2$ se:
 1. todas as folhas de T têm o mesmo nível;
 2. cada nó interno tem um número variável r de registros de informação e $r+1$ de filhos, onde
$$\begin{cases} \lfloor b/2 \rfloor \leq r \leq b & \text{se nó} \neq \text{raiz} \\ 1 \leq r \leq b & \text{se nó} = \text{raiz} \end{cases}$$
 3. cada folha tem um número variável r de registros obedecendo à mesma restrição do item anterior;
 4. os campos de informação contidos nos registros obedecem à propriedade de árvores de busca.
- ▶ Alguns autores definem de maneira diferente o conceito de ordem: $b = 2m$, $m \geq 1$ (assim, b é sempre par).
- ▶ Pode-se provar que a altura máxima h de uma árvore B de ordem b que contém n registros é dada por:

$$\log_{\lfloor b/2 \rfloor} (n+1)/2.$$

Exemplo de árvore B de ordem 3 (árvore 2-3-4)

Neste caso, $b=3$ e cada nó tem no mínimo um e no máximo três registros de informação (dois, três ou quatro apontadores).

$$\lfloor b/2 \rfloor \leq r \leq b$$

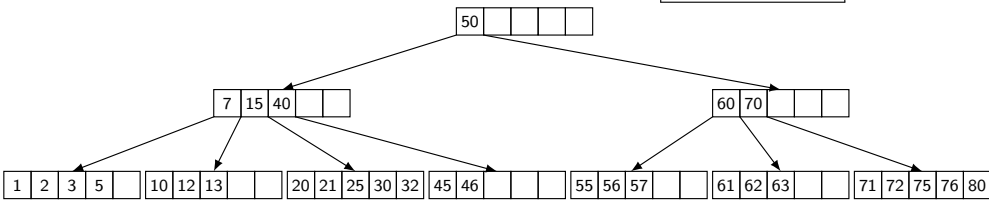


Árvores 2-3-4 são frequentemente usadas na memória, através da implementação denominada *árvores rubro-negras* a serem vistas mais adiante.

Exemplo de árvore B de ordem 5

Neste caso, $b=5$ e cada nó contém no mínimo dois e no máximo cinco registros de informação; somente a raiz pode conter um registro.

$$\lfloor b/2 \rfloor \leq r \leq b$$



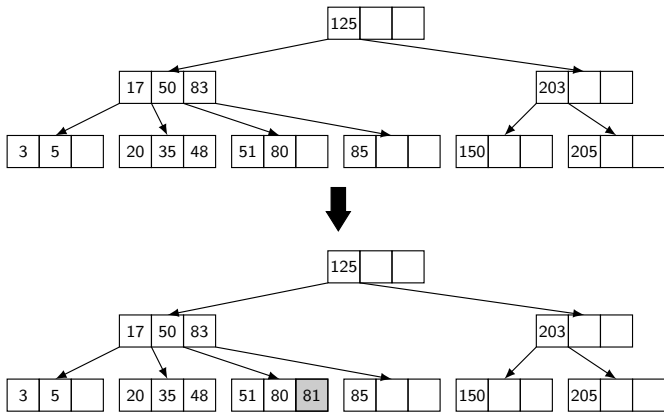
Números mínimos e máximos de registros

Árvore B de ordem 255:

nível	mínimo		máximo	
	nós	registros	nós	registros
1	1	1	1	1×255
2	2	2×127	256^1	$256^1 \times 255$
3	2×128^1	$2 \times 128^1 \times 127$	256^2	$256^2 \times 255$
4	2×128^2	$2 \times 128^2 \times 127$	256^3	$256^3 \times 255$
5	2×128^3	$2 \times 128^3 \times 127$	256^4	$256^4 \times 255$
Total	4.227.331	536.870.911	4.311.810.305	1.099.511.627.775

Exemplos de inserção

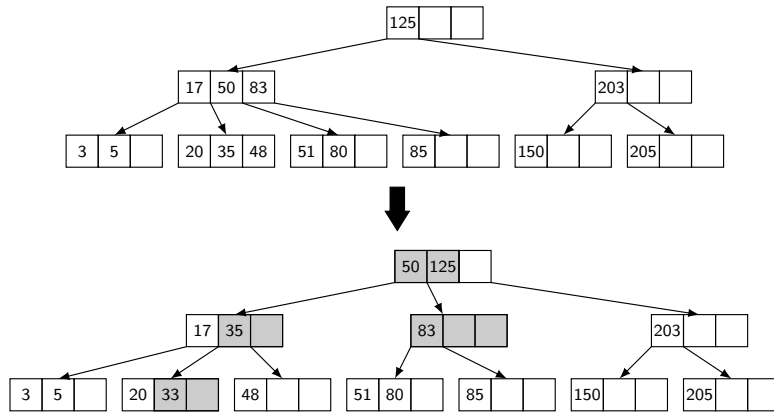
Inserção de 81:



Neste caso, foi feita inserção numa folha com espaço disponível. Houve h leituras e uma gravação (h é a altura da árvore). O processo não se propaga.

Exemplos de inserção (cont.)

Inserção de 33:



A capacidade de uma folha seria excedida e foi feita uma quebra que propagou-se para cima. Haveria no máximo h leituras e $2h+1$ gravações (se a raiz também fosse quebrada).

Representação de árvores B

```
#define ORDEM 255

typedef struct NoArvB * ArvB;

typedef struct NoArvB {
    int numregs;
    ArvB filhos[ORDEM+1];
    T info[ORDEM];
} NoArvBin;
```

Esta representação será usada para simular árvores B na memória RAM. Normalmente, árvores B são implementadas em memórias externas como discos. O endereçamento em discos é usado em lugar de apontadores comuns.

Inserção em árvores B

A explicação a seguir supõe que a inserção é realizada por uma função recursiva auxiliar cujo cabeçalho é:

```
Boolean InsereRecArvB(ArvB *p, ArvB *s, T *x, Boolean *prop);
```

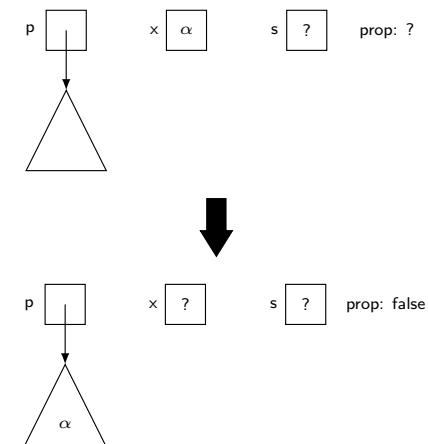
onde

- ▶ p : endereço da variável que contém o apontador para a árvore
- ▶ $prop$: endereço da variável na qual é devolvida a informação que indica se houve propagação de inserção no retorno
- ▶ x : endereço de uma variável
 - ▶ numa chamada: contém o valor a ser inserido de algum tipo T conveniente
 - ▶ no retorno, se houver propagação: contém o valor a ser propagado que separa os valores das árvores apontadas por p e por s
- ▶ s : endereço da variável que contém o apontador para a árvore propagada à direita de x (se houver)
- ▶ se não houver propagação numa chamada recursiva, o resto da árvore não sofre mudança

O valor devolvido pela função indica se a inserção foi efetivamente realizada ou se o elemento x já pertencia à árvore.

Inserção em árvores B (cont.)

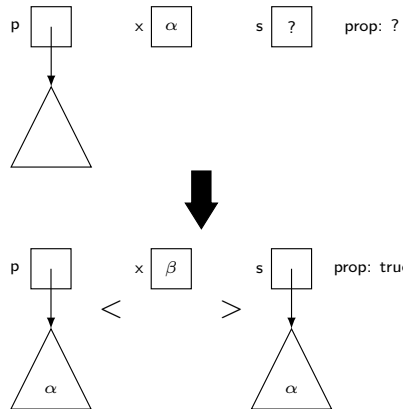
Explicação geral: caso de chamada recursiva sem propagação no retorno



Neste caso, não haverá mais modificações na árvore.

Inserção em árvores B (cont.)

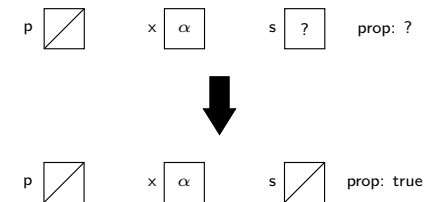
Explicação geral: caso de chamada recursiva com propagação no retorno



Neste caso, a modificação deverá ser propagada para cima. O valor α da variável x foi inserido numa das duas árvores (ou $\beta = \alpha$). Se p apontava para a raiz da árvore, será necessário criar uma nova raiz, com um único valor β , e subárvores apontadas por p e s .

Inserção em árvores B (cont.)

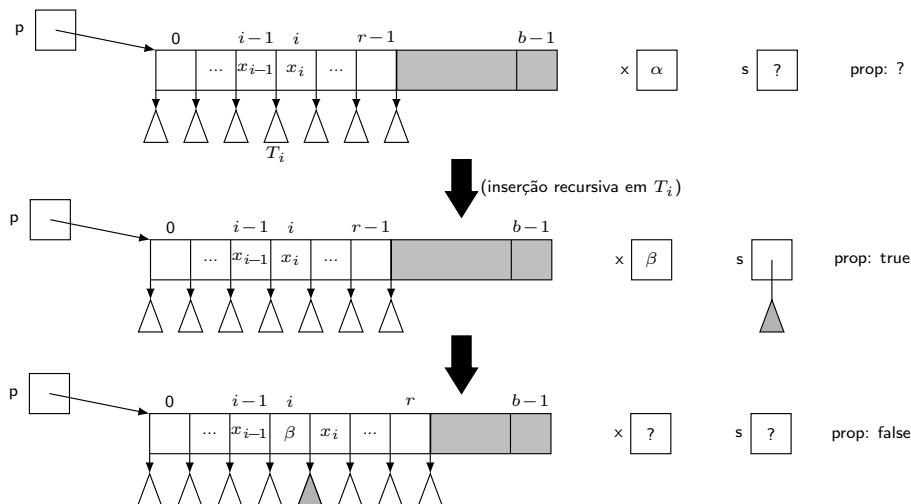
Caso 1: árvore vazia



Neste caso, são adotados valores das variáveis x , s e $prop$ de maneira a recair no caso geral de propagação.

Inserção em árvores B (cont.)

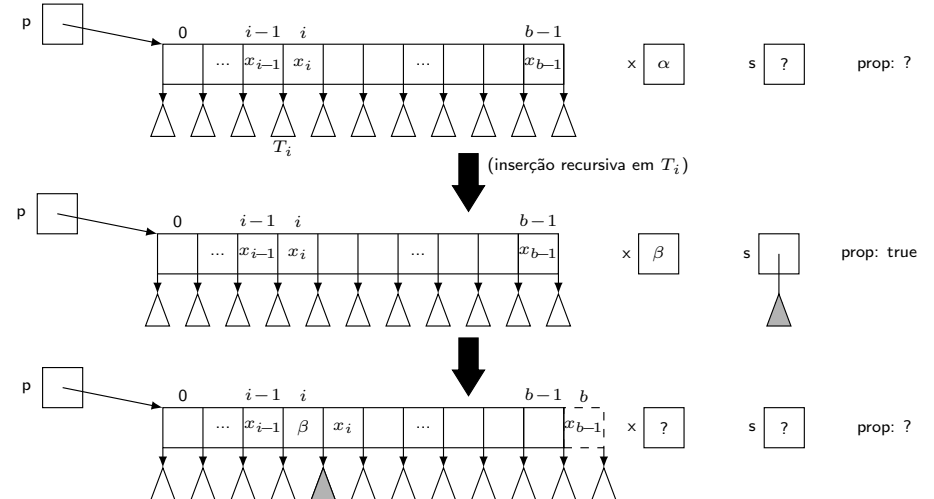
Caso 2: inserção com espaço disponível ($r < b$)



Neste caso, o valor propagado após a chamada recursiva é absorvido no nó corrente e a propagação para.

Inserção em árvores B (cont.)

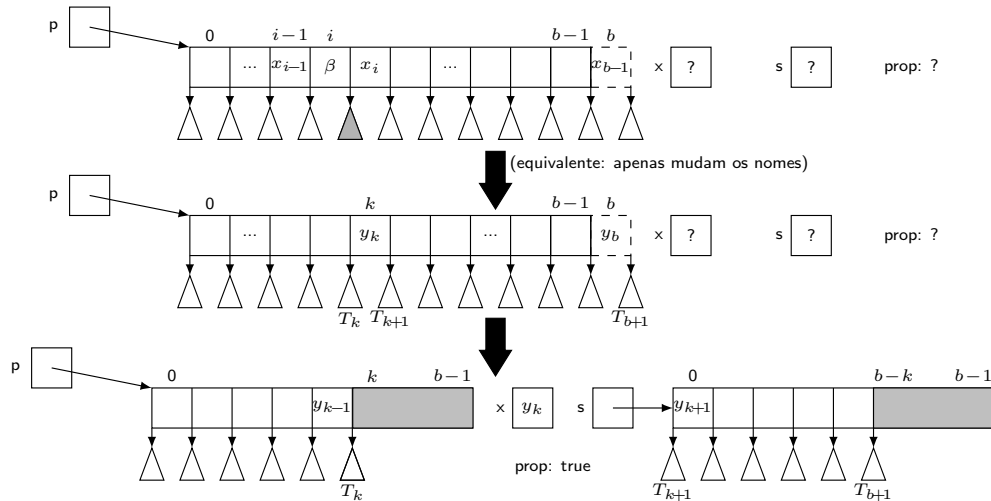
Caso 3: inserção sem espaço disponível ($r = b$)



Neste caso, o valor propagado após a chamada recursiva não pode ser absorvido pois o nó teria que ser aumentado além da capacidade máxima; continua com quebra do nó (o espaço extra é apenas conceitual).

Inserção em árvores B (cont.)

Caso 3: inserção sem espaço disponível – quebra do nó



O nó corrente é quebrado em dois; o primeiro (nó original apontado por p) retém $k = \lceil b/2 \rceil$ primeiros registros; o k -ésimo elemento e um novo nó com $b-k$ registros restantes são propagados de volta.

Função de inserção auxiliar (esboço)

```
Boolean InsereRecArvB(ArvB *p, ArvB *s, T *x, Boolean *prop) {
    int i; Boolean inseriu;
    if (*p == NULL) {
        *prop = true; *s = NULL; return true;
    }
    i = IndiceArvB(p, x); // localiza o ponto de inserção
    if ((i < ((*p)->numregs)) && (x == ((*p)->info)[i])) {
        *prop = false; return false;
    }
    inseriu = InsereRecArvB(&((*p)->filhos[i]), s, x, prop);
    if (*prop) {
        InserInfoArvB(p, s, x, i); // insere 's' e 'x' no nó
        ((*p)->numregs)++;
        if (((*p)->numregs <= ORDEM))
            *prop = false;
        else {
            QuebraNoArvB(p, s, x); *prop = true; // quebra
        }
    }
    return inseriu;
}
```

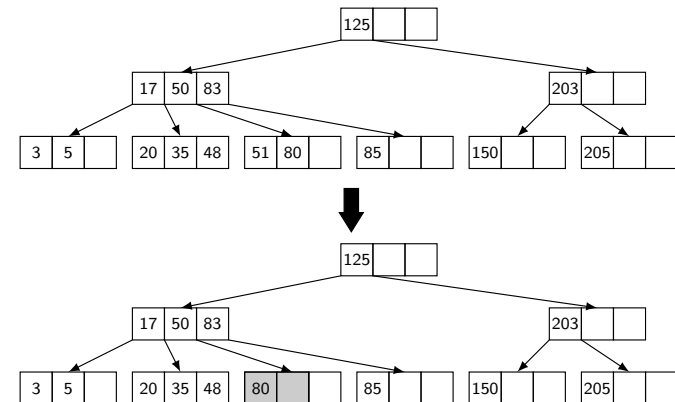
Função de inserção inicial

```
Boolean InsereArvB(ArvB *p, T x) {
    /* Devolve 'false' se o valor de 'x' já ocorre na árvore 'p' */
    Boolean prop;
    ArvB q, s;
    Boolean inseriu = InsereRecArvB(p, &s, &x, &prop);
    if (prop) {
        q = (ArvB) malloc(sizeof(NóArvB));
        q->numregs = 1;
        (q->filhos)[0] = *p; (q->filhos)[1] = s;
        (q->info)[0] = x;
        *p = q;
    }
    return inseriu;
}
```

O eventual aumento da altura da árvore se dará sempre nesta função.

Exemplos de remoção

Remoção de 51:

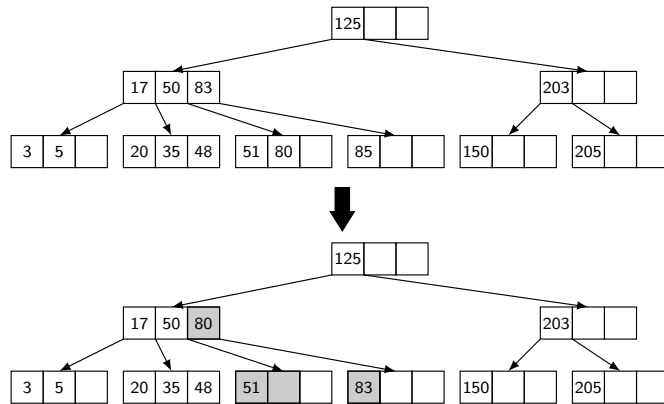


Neste caso, foi feita remoção numa folha com número de registros acima do mínimo. Houve h leituras e uma gravação (h é a altura da árvore). O processo não se propaga.

Em todos os casos, a remoção deverá iniciar-se numa folha. Se necessário, um elemento de um nó interno deverá ser substituído convenientemente.

Exemplos de remoção (cont.)

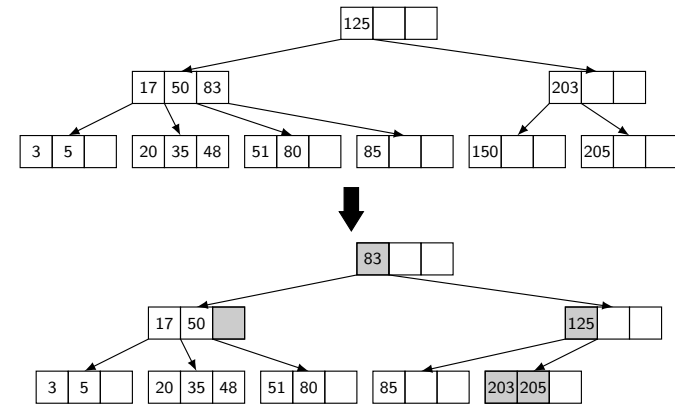
Remoção de 85:



Neste caso, foi feita remoção numa folha com número mínimo de registros, e foi feito um “empréstimo” de um nó irmão imediato com sobra de registros. O empréstimo passa pelo nó pai a fim de manter a propriedade de árvore de busca. Haveria no máximo $h + 2$ leituras e três gravações. O processo não se propaga.

Exemplos de remoção (cont.)

Remoção de 150:



Neste caso, foi feita remoção numa folha com número mínimo de registros e cujos irmãos também estão no mínimo. Foi feita então uma junção de dois nós e incluído o valor do nó pai que os separa. A remoção deste valor do nó pai seguirá o mesmo esquema de remoções, e poderá se propagar até a raiz. Haveria no máximo $3h - 2$ leituras e $2h - 1$ gravações.

Observações

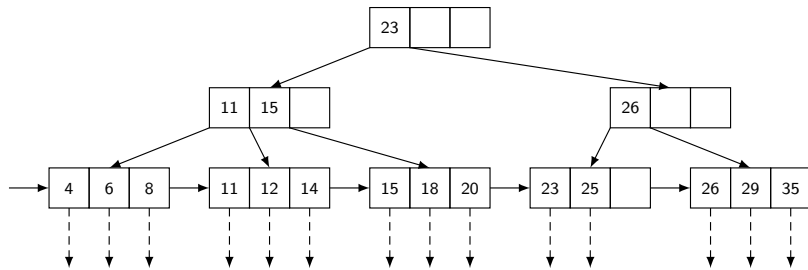
- ▶ Existem outros variantes para os algoritmos – v. literatura.
- ▶ Verifica-se facilmente que tanto no caso de quebras (inserção) como no caso de junções (remoção), os nós resultantes preservam as propriedades de árvores B.
- ▶ O número de leituras e gravações é sempre proporcional à altura da árvore.
- ▶ O nó raiz da árvore é normalmente guardado na memória, diminuindo o número de acessos ao disco.
- ▶ De acordo com a definição das árvores B, a utilização mínima do espaço dos nós é de cerca de 50%; pode-se provar que a utilização média é de cerca de 69%.
- ▶ Usando técnicas probabilísticas, pode-se mostrar que as operações mais complicadas são muito infrequentes.
- ▶ A remoção pode ser implementada de maneira análoga à inserção e será deixada para exercício.
- ▶ Uma árvore B inicial pode ser construída por inserções sucessivas o que seria muito ineficiente; na prática, utiliza-se um algoritmo direto.

Variantes de árvores B

- ▶ Árvores B*: o número de registros ocupados de um nó é no mínimo $\frac{2}{3}$ da sua capacidade.
- ▶ Árvores B⁺:
 - ▶ nós internos com chaves apenas para orientar o percurso
 - ▶ pares (*chave, valor*) apenas nas folhas
 - ▶ regra de descida:
 - ▶ subárvore esquerda: menor
 - ▶ subárvore direita: maior ou igual
 - ▶ apontadores em lugar de valores tornando mais eficiente a movimentação dos registros durante inserções e remoções
 - ▶ ligações facilitando percurso em ordem de chaves

Variantes de árvores B (cont.)

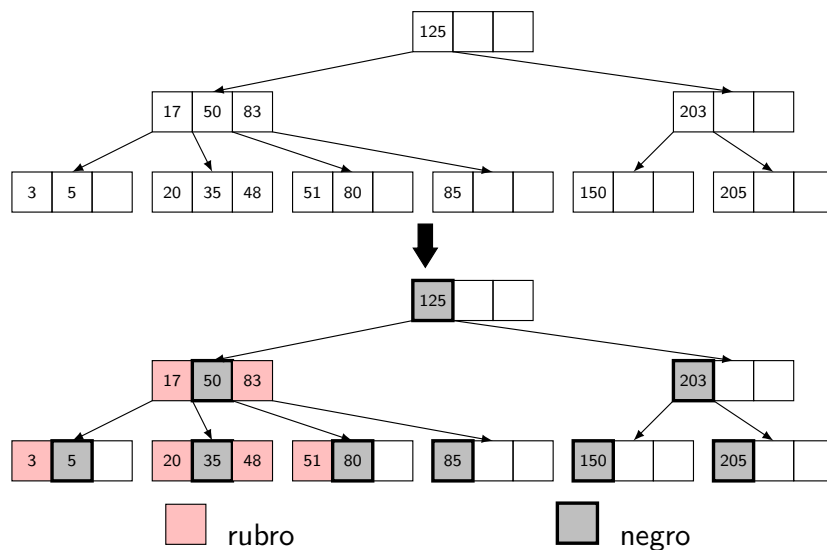
Exemplo de árvore B⁺ de ordem 3:



Setas tracejadas indicam apontadores para os valores da informação. A lista ligada das folhas permite percurso simples e eficiente em ordem de chaves.

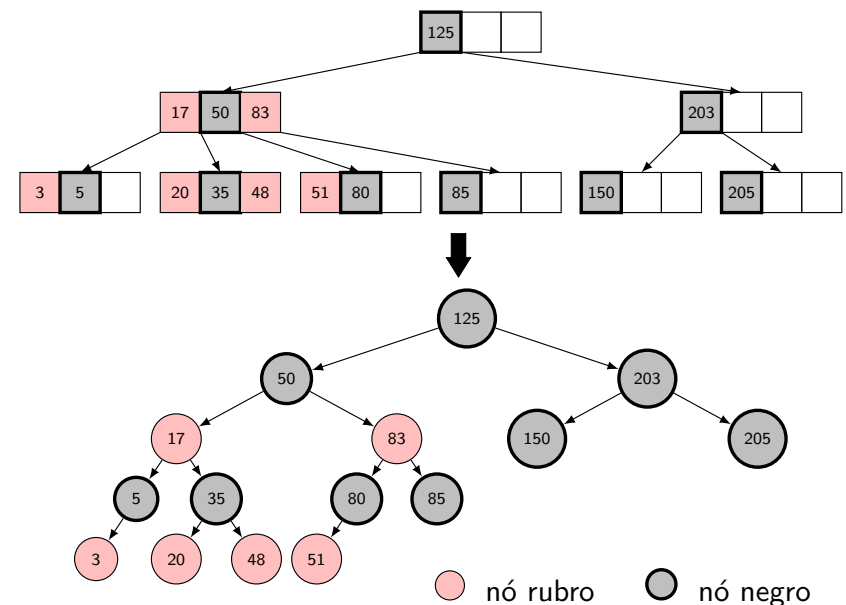
Árvores rubro-negras (Red-black trees)

Exemplo de árvore B de ordem 3 (ou árvore 2-3-4)

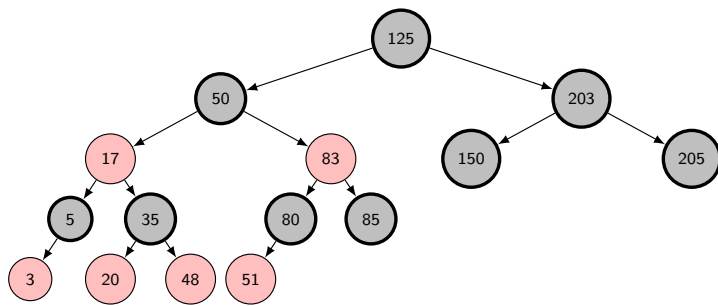


Deve-se notar a coloração em cada um dos três casos: um valor, dois valores e três valores. No caso de dois valores, poderia ser contrária.

Transformação para árvore binária



Árvores rubro-negras



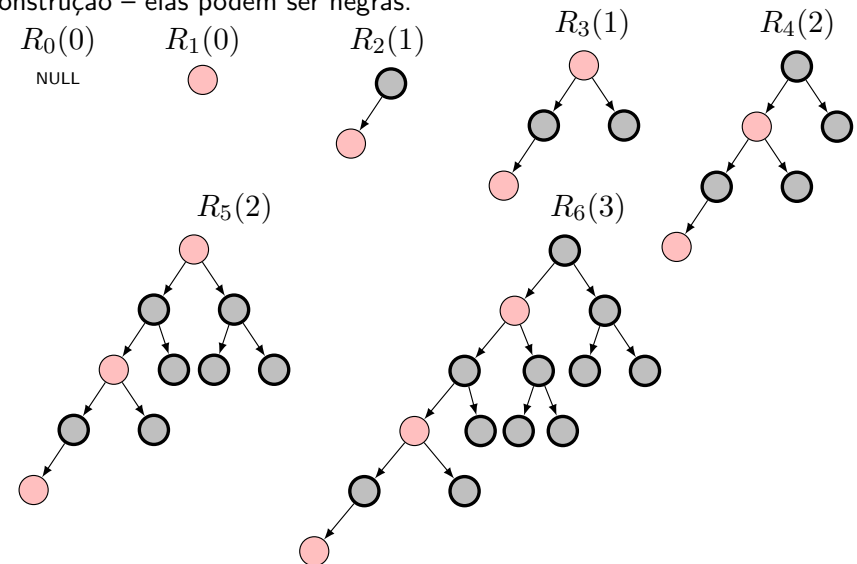
Definição:

- ▶ cada nó recebe uma cor: rubra ou negra
- ▶ a raiz recebe a cor negra
- ▶ os filhos de um nó rubro são sempre negros
- ▶ o número de nós negros em qualquer caminho a partir de um nó v até uma subárvore vazia é sempre o mesmo (*altura negra do nó v*).

Consequência: A altura de uma árvore rubro-negra é, no máximo, o dobro da altura negra da sua raiz (no exemplo 5 e 3, respectivamente).

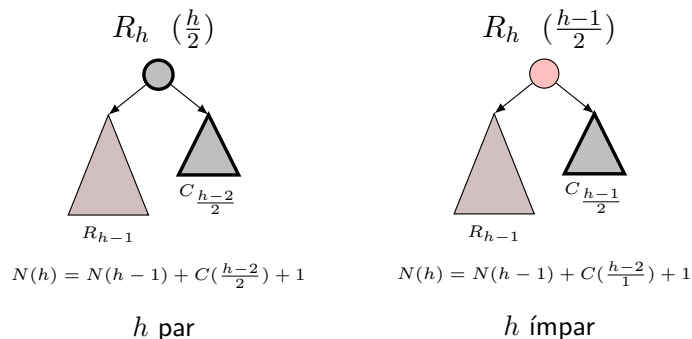
Árvores rubro-negras minimais

R_h denota uma árvore de altura h com o mínimo de nós. Alturas negras estão entre parênteses; casos ímpares têm raízes rubras apenas para fins desta construção – elas podem ser negras.



Árvores rubro-negras minimais (cont)

Casos gerais:



C_k denota uma árvore completa de altura k contendo apenas nós negros ($2^k - 1$ nós).

Árvores rubro-negras minimais (cont.)

Demonstra-se que:

- ▶ Numa árvore rubro-negra, qualquer subárvore de altura h , contém pelo menos $2^b - 1$ nós, onde b é a altura negra da subárvore (prova por indução).
- ▶ O número de nós n de uma árvore rubro-negra: $n \geq 2^{h/2} - 1$.
- ▶ A altura h de uma árvore rubro-negra: $h \leq 2 \log_2(n + 1)$.

Conclui-se que, para um n grande, a altura máxima da árvore rubro-negra com n nós é $h \approx 2 \log_2 n$.

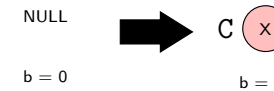
Inserção em árvores rubro-negras

- ▶ Inicialmente, um nó é inserido como uma folha de cor rubra, seguindo o algoritmo comum de inserção.
- ▶ Em seguida, há uma subida na direção da raiz da árvore enquanto o nó corrente é rubro, realizando transformações que preservam as propriedades de árvores rubro-negras, sem mudar a altura negra da subárvore: há vários casos possíveis.
- ▶ Caso a raiz final da árvore tenha ficado rubra, sua cor é mudada para a cor negra, aumentando em uma unidade a altura negra da árvore inteira.

Nas transparências seguintes, o nó corrente será indicado por C, seu nó pai por P, seu nó avô por A e seu nó tio por T. A letra b indicará a altura negra de uma árvore.

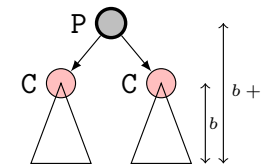
Inserção em árvores rubro-negras (cont.)

Caso 1: inserção em árvore vazia



Inicia-se o processo de subida.

Caso 2: o nó pai P é negro

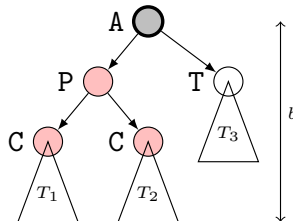


O nó corrente C pode ser filho esquerdo ou direito de P – o processo para.

Inserção em árvores rubro-negras (cont.)

Caso 3: o nó pai P é rubro

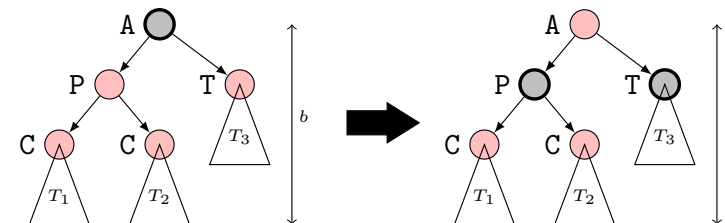
Neste caso, deve existir o nó avô A negro; caso contrário P não poderia ser rubro:



A ação a ser tomada depende da cor do nó tio T (se ele existe) e da posição do nó corrente C (filho esquerdo ou filho direito), produzindo três casos.

Inserção em árvores rubro-negras (cont.)

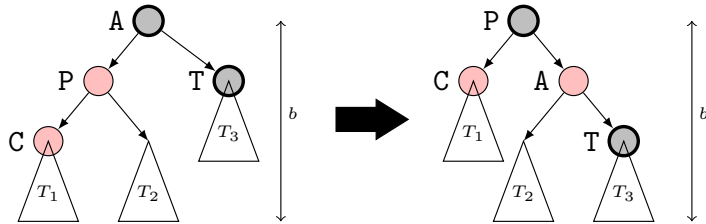
Caso 3a: o nó pai P e o nó tio T (existe!) são ambos rubros (a posição esquerda ou direita de C não é relevante)



Neste caso foi realizada uma mudança de cor dos nós A, P e T, preservando as propriedades de árvores rubro-negras, incluindo a altura negra. O nó A torna-se o novo nó corrente e o processo continua.

Inserção em árvores rubro-negras (cont.)

Caso 3b: o nó pai P é rubro, o nó tio T, se existir, é negro e o nó corrente C é filho esquerdo de P

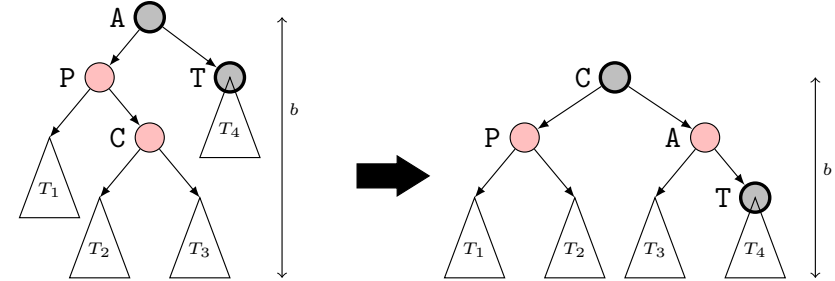


Neste caso foi realizada uma rotação simples com a troca de cores de P e de A que preserva as propriedades citadas. Como a nova raiz é de cor negra, o processo para.

A mesma transformação pode ser usada quando o nó T não existe (isto é, as árvores T_1 , T_2 e T_3 são vazias).

Inserção em árvores rubro-negras (cont.)

Caso 3c: o nó pai P é rubro, o nó tio T, se existir, é negro e o nó corrente C é filho direito de P

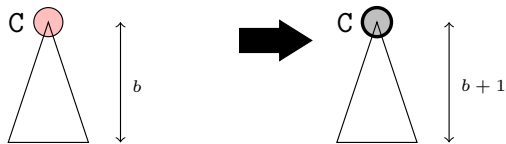


Neste caso foi realizada uma rotação dupla, com a troca de cores entre os nós A e C, preservando as propriedades citadas. Como a nova raiz é de cor negra, o processo para.

A mesma transformação pode ser usada quando o nó T não existe (isto é, as árvores T_1 , T_2 , T_3 e T_4 são vazias).

Inserção em árvores rubro-negras (cont.)

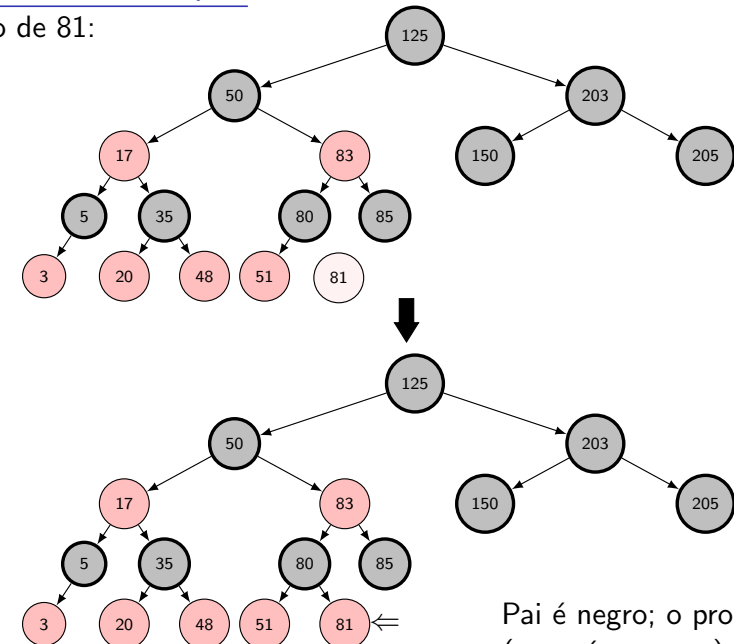
Caso 4: o nó corrente C é a raiz da árvore final



Neste caso foi apenas trocada a cor da raiz. Com isto, a altura negra da árvore cresceu de um. Esta é única situação em que a altura negra aumenta. Deve-se notar a analogia com as árvores B que também crescem pela raiz.

Exemplos de inserção

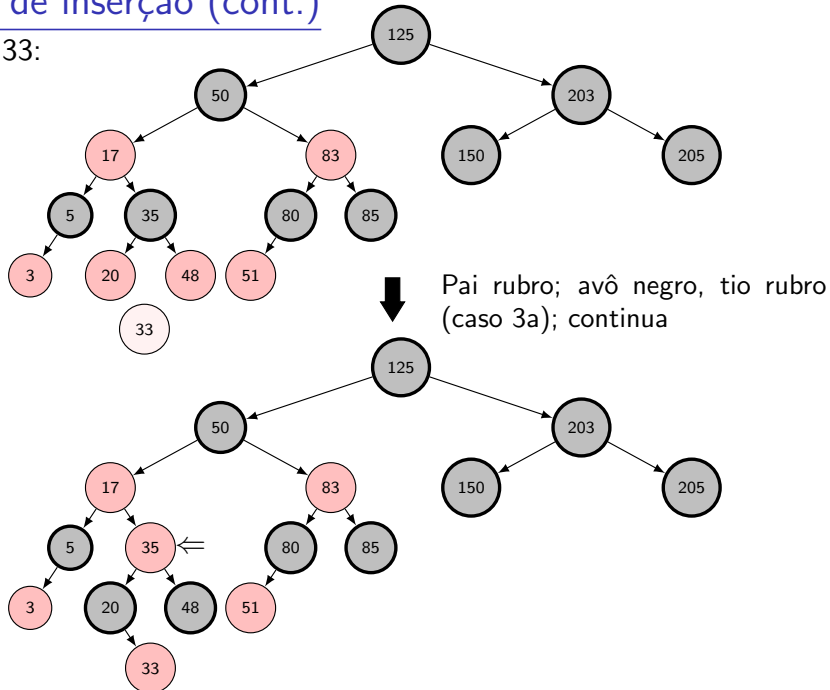
Inserção de 81:



Pai é negro; o processo para (⇐: nó corrente)

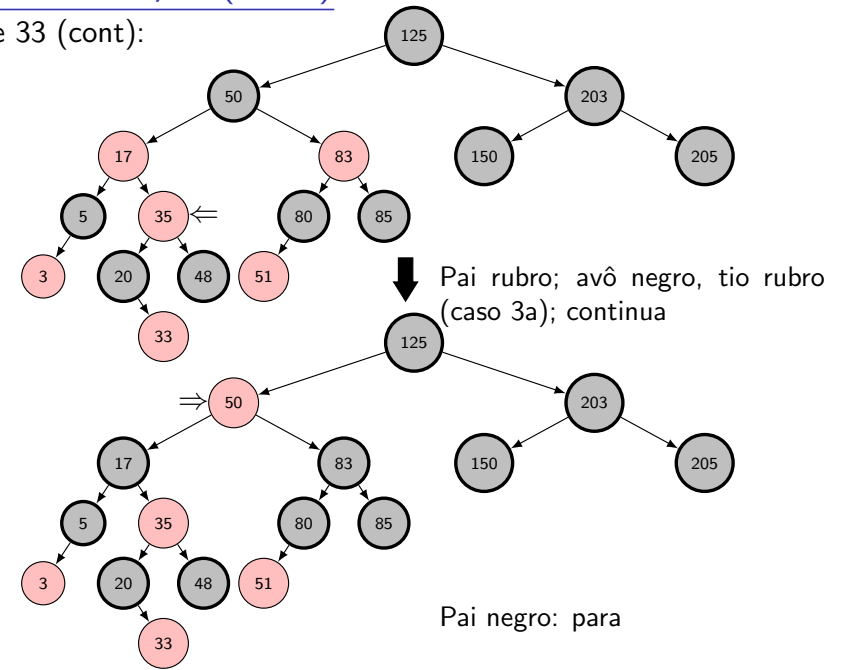
Exemplos de inserção (cont.)

Inserção de 33:



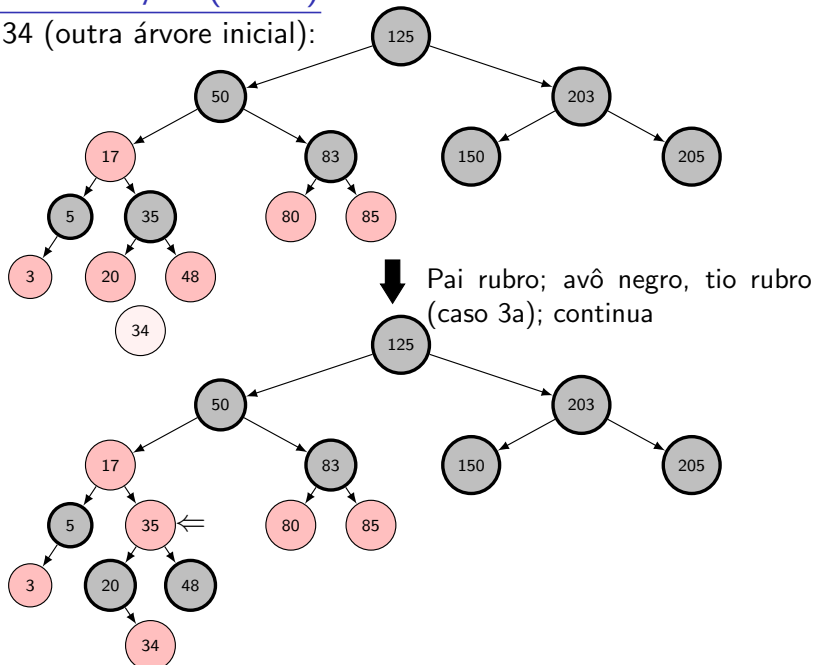
Exemplos de inserção (cont.)

Inserção de 33 (cont):



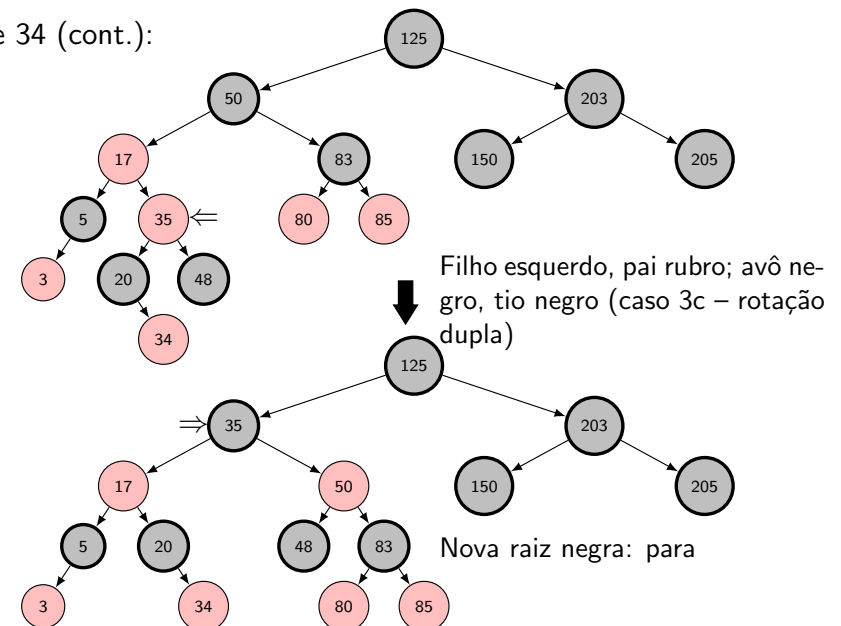
Exemplos de inserção (cont.)

Inserção de 34 (outra árvore inicial):



Exemplos de inserção (cont.)

Inserção de 34 (cont.):



Comparação de árvores rubro-negras com AVL

- ▶ Árvores AVL têm altura máxima (e média) menor do que rubro-negras:

$$h \approx 1,44 \log_2 n \quad h \approx 2 \log_2 n$$

- ▶ Consequentemente, busca em árvores AVL é mais rápida.
- ▶ Operações de inserção e de remoção em árvores rubro-negras são mais rápidas, mas também $O(\log_2 n)$.
- ▶ Árvores AVL são melhores para árvores que sofrem poucas mudanças.
- ▶ Árvores rubro-negras são melhores para árvores muito dinâmicas.
- ▶ Cada nó de uma árvore AVL precisa de um campo para o fator de balanceamento (dois bits).
- ▶ Cada nó de uma árvore rubro-negra precisa de um campo para a cor (um bit).

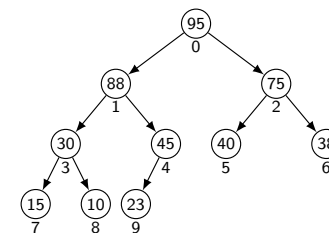
O algoritmo de busca é idêntico para todas as árvores binárias de busca. O algoritmo de remoção para árvores rubro-negras é um pouco mais complicado; sua descrição pode ser encontrada na literatura.

Filas de prioridade (Priority queues)

Definição e propriedades

- ▶ Uma fila de prioridade é uma árvore binária com as propriedades:
 - ▶ a árvore é *completa* ou *quase completa*;
 - ▶ em cada nó da árvore, o valor da chave é maior do que valores das chaves dos filhos (e consequentemente, de todos os descendentes).
- ▶ Uma fila de prioridade **não é** uma árvore de busca!
- ▶ A determinação do elemento máximo de uma fila de prioridade pode ser feita em tempo constante (está na raiz).
- ▶ As operações de inserção e de remoção podem ser realizadas em tempo proporcional à altura ($O(\log n)$).
- ▶ Filas de prioridade podem ser implementadas eficientemente de maneira sequencial.
- ▶ Em algumas aplicações é conveniente utilizar filas de prioridade em que um elemento é menor ou igual a todos os seus descendentes (mínimo na raiz).

Exemplo

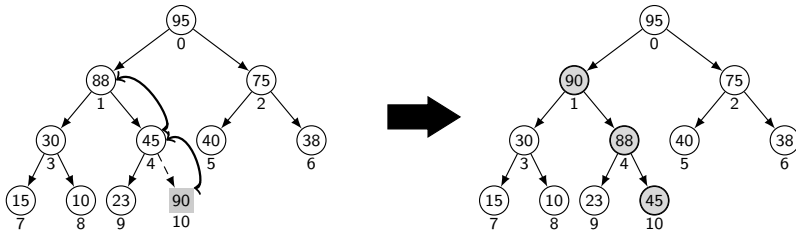


Implementação sequencial (*heap*):

95	88	75	30	45	40	38	15	10	23
0	1	2	3	4	5	6	7	8	9

Operação de subida

Supondo que, exceto pelo último elemento, a árvore é uma fila de prioridade, a operação torna a árvore inteira uma fila válida.

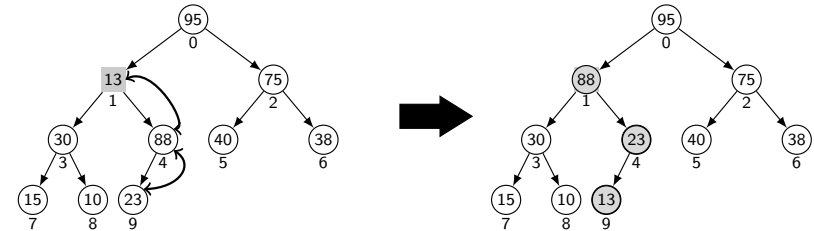


Setas duplas indicam as operações de troca a serem realizadas.

Obviamente, o número de operações de troca executadas é, no máximo, igual à altura da árvore original ($\log_2 n$).

Operação de descida

Supondo que, exceto por um único valor que não é maior ou igual do que seus descendentes, a árvore é uma fila de prioridade, a operação torna a árvore inteira uma fila válida.



Setas duplas indicam as operações de troca a serem realizadas.

Obviamente, o número de operações de troca executadas é menor do que a altura da árvore original ($\log_2 n$).

Implementação das operações

```
#define TAM.MAX 50
```

```
typedef struct {  
    T vetor[TAM.MAX];  
    int tam;  
} Heap;
```

```
void Sobe(Heap *h, int m) {  
    int j = (m-1)/2;  
    T x = (*h).vetor[m];  
  
    while ((m>0) && ((*h).vetor[j]<x)) {  
        (*h).vetor[m] = (*h).vetor[j];  
        m = j;  
        j = (j-1)/2;  
    }  
    (*h).vetor[m] = x;  
} /* Sobe */
```

Note-se que as operações de troca foram otimizadas com a utilização da variável temporária x .

Implementação das operações (cont.)

```
void Desce(Heap *h, int m) {  
    int k = 2*m+1;  
    T x = (*h).vetor[m];  
    while (k<(*h).tam) {  
        if ((k<((*h).tam)-1) && ((*h).vetor[k]<(*h).vetor[k+1]))  
            k++;  
        if (x<(*h).vetor[k]) {  
            (*h).vetor[m] = (*h).vetor[k];  
            m = k;  
            k = 2*k+1;  
        } else  
            break;  
    }  
    (*h).vetor[m] = x;  
} /* Desce */
```

Também neste caso, as operações de troca foram otimizadas.

Construção inicial

Dado um vetor com elementos em ordem arbitrária, deve-se transformá-lo numa fila de prioridade:

```
void ConstroiHeap1(Heap *h) {
    int i;
    for (i=1; i<(*h).tam; i++)
        Sobe(h, i);
} /* ConstroiHeap1 */

void ConstroiHeap2(Heap *h) {
    int i;
    for (i=((*h).tam-2)/2; i>=0; i--)
        Desce(h, i);
} /* ConstroiHeap2 */
```

Verifica-se facilmente que a eficiência da função *ConstroiHeap1* é $O(n \log n)$. Pode-se demonstrar, também, que a eficiência de *ConstroiHeap2* é $O(n)$ (linear).

Inserção e remoção

```
void InsereHeap(Heap *h, T x) {
    (*h).vetor[(*h).tam] = x;
    ((*h).tam)++;
    Sobe(h, ((*h).tam)-1);
} /* InsereHeap */

void RemoveHeap(Heap *h, T *x) {
    *x = (*h).vetor[0];
    ((*h).tam)--;
    (*h).vetor[0] =
        (*h).vetor[(*h).tam];
    Desce(h, 0);
} /* RemoveHeap */
```

Note-se que a função *RemoveHeap* remove e devolve na variável x o elemento máximo da fila. Obviamente, as duas funções realizam no máximo $O(\log n)$ operações.

Algoritmo de ordenação *Heapsort*

O algoritmo constrói um *heap* inicial. Em seguida, remove um a um o elemento máximo e o coloca na posição final do vetor.

```
void HeapSort(Heap *h) {
    int i, n = (*h).tam;
    /* constrói heap */
    for (i=(n-2)/2; i>=0; i--)
        Desce(h, i);
    /* ordena */
    for (i=n-1; i>0; i--) {
        T t = (*h).vetor[0];
        (*h).vetor[0] = (*h).vetor[i];
        (*h).vetor[i] = t;
        (*h).tam--;
        Desce(h, 0);
    }
    (*h).tam = n;
} /* HeapSort */
```

Número de operações: $O(n \log n)$ (um dos algoritmos ótimos).

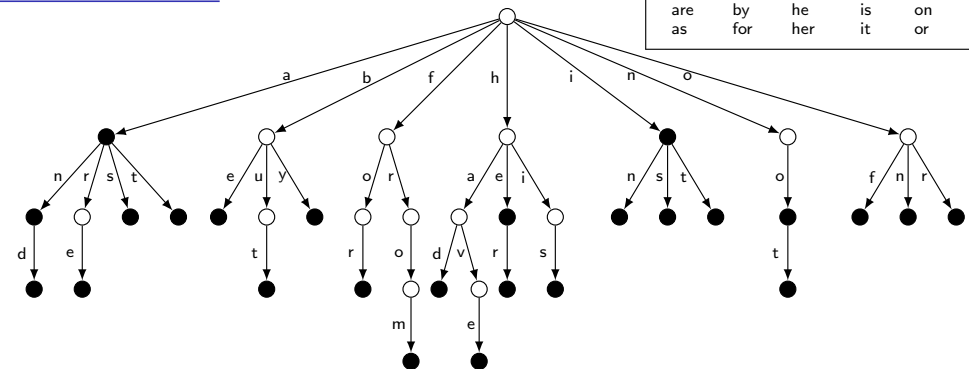
Árvores digitais (Tries)

Conjuntos de cadeias de caracteres

Exemplo: 25 palavras em inglês

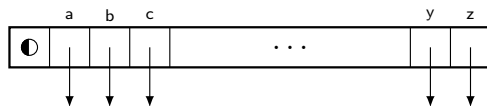
a	at	from	his	no
an	be	had	i	not
and	but	have	in	of
are	by	he	is	on
as	for	her	it	or

Árvore digital



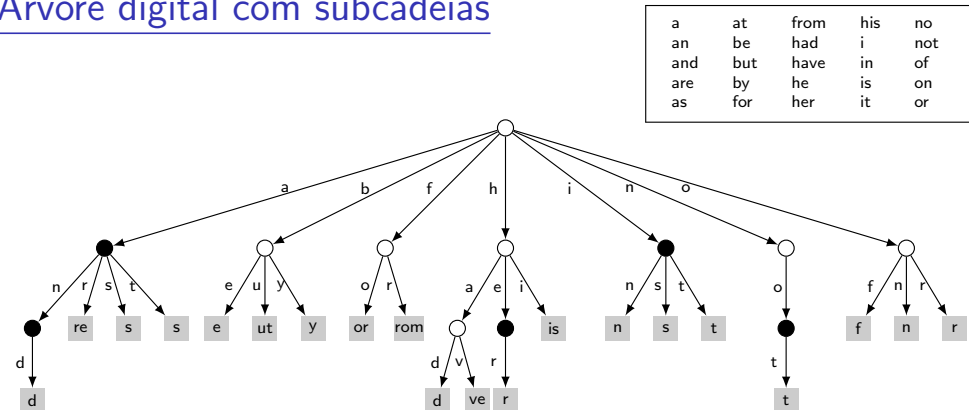
- ▶ Arestas são rotuladas com as letras das palavras.
- ▶ Nós cheios indicam o fim de uma cadeia.
- ▶ São fatorados os prefixos comuns das cadeias.
- ▶ Números para o exemplo:
 - ▶ 39 nós
 - ▶ 20 folhas
 - ▶ 19 nós internos (não folhas)
 - ▶ 25 nós cheios (25 palavras)

Implementação de árvores digitais



- ▶ Algoritmos de busca, inserção e remoção óbvios (exercício).
- ▶ Uso de memória:
 - ▶ $39 \text{ nós} \times 26 = 1014$ campos apontadores
 - ▶ 38 campos são não nulos
- ▶ Eliminando apontadores das folhas:
 - ▶ $19 \text{ nós internos} \times 26 = 494$ campos apontadores
 - ▶ 38 campos são não nulos
- ▶ Existem alternativas mais econômicas para representar os nós:
 - ▶ subcadeias finais
 - ▶ rótulos explícitos das subárvores

Árvore digital com subcadeias

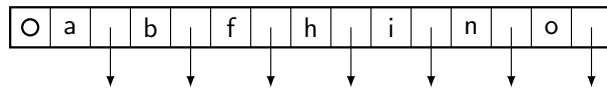


- ▶ Uso de memória:
 - ▶ $12 \text{ nós} \times 26 = 312$ campos apontadores
 - ▶ 31 campos são não nulos

Árvore digital com rótulos explícitos

a	at	from	his	no
an	be	had	i	not
and	but	have	in	of
are	by	he	is	on
as	for	her	it	or

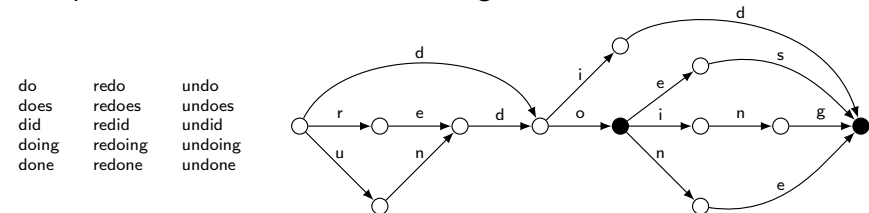
- ▶ Cada nó da árvore incluirá apenas as subárvores não vazias e seus rótulos; a raiz do exemplo ficaria:



- ▶ Uso de busca da letra em lugar da indexação direta (geralmente poucos elementos)
- ▶ Uma árvore vazia: NULL
- ▶ Uma árvore contendo apenas a cadeia vazia:
- ▶ Uso de memória no caso do exemplo: exercício
- ▶ Problema: tamanho dinâmico dos nós – implementação da inserção e da remoção mais complicada

Autômato finito minimal acíclico

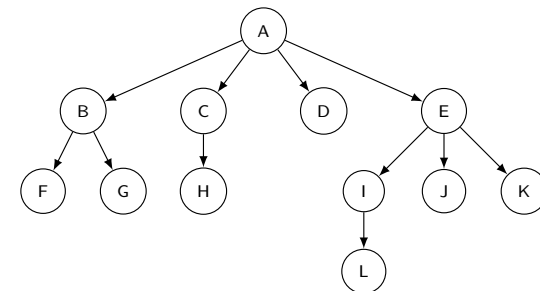
Exemplo: as 15 formas dos verbos ingleses: *do*, *redo* e *undo*



- ▶ São fatorados tanto os prefixos quanto os sufixos comuns das cadeias.
- ▶ Algoritmo de busca igual ao de árvores digitais.
- ▶ Algoritmos de inserção e de remoção muito mais complicados.
- ▶ Uso de memória:
 - ▶ 11 nós \times 26 = 286 campos apontadores (nós internos)
 - ▶ 16 campos são não nulos
- ▶ Se fosse árvore digital:
 - ▶ 26 nós \times 26 = 676 campos apontadores (nós internos)
 - ▶ 37 campos seriam não nulos
- ▶ As estruturas para um exemplo análogo em português seriam maiores (mais de 50 formas verbais) mas resultariam em muito mais economia.

Árvores gerais

Exemplo de árvore geral



- ▶ árvores gerais nunca são vazias
- ▶ as subárvores são ordenadas: primeira, segunda, etc
- ▶ o número de subárvores pode ser qualquer, inclusive zero
- ▶ conceitos naturais: grau, filhos, pai, descendente, altura, etc

Representação de árvores gerais

```
#define GRAU_MAX 10

typedef struct
    NoArvGeral *ArvGeral;
typedef struct NoArvGeral {
    T info;
    int grau;
    ArvGeral filhos[GRAU_MAX];
} NoArvGeral;

...

p = malloc(sizeof(NoArvGeral));
...

typedef struct
    NoArvGeral *ArvGeral;
typedef struct NoArvGeral {
    T info;
    int grau;
    ArvGeral filhos[1];
} NoArvGeral;

...

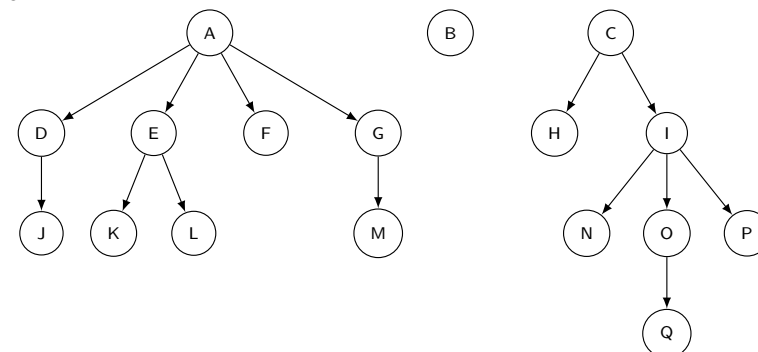
p = malloc(sizeof(NoArvGeral)+
    (grau-1)*sizeof(ArvGeral));
...
```

A segunda alternativa permite economia de memória e evita a imposição de um limite no grau do nó. Uma vez alocado, o nó fica com tamanho limitado.

Florestas

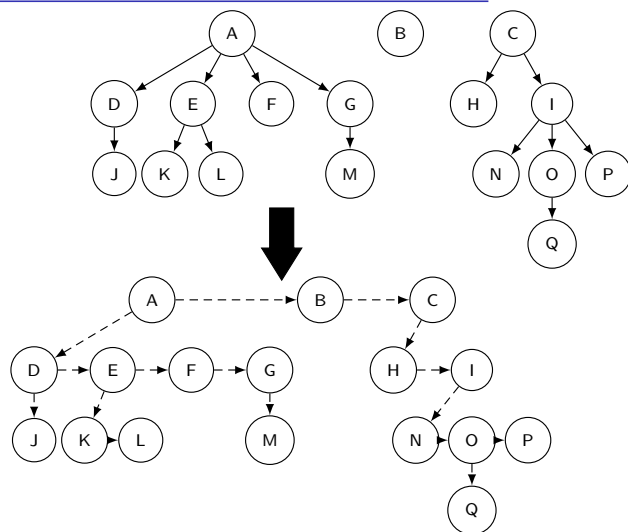
Uma *floresta* é uma sequência, possivelmente vazia, de árvores gerais.

Exemplo:



Note-se que as subárvores de um nó de uma árvore geral constituem uma floresta.

Floresta representada como árvore binária



- ▶ o campo esquerdo aponta para a raiz da primeira subárvore original
- ▶ o campo direito aponta para o nó irmão seguinte
- ▶ as raízes das árvores da floresta são consideradas irmãos.

Floresta representada como árvore binária (cont.)

- ▶ A árvore binária $\mathbf{B}(F)$ que representa uma floresta $F = (T_1, T_2, \dots, T_m)$ é definida por:
 - ▶ árvore binária vazia se F é uma floresta vazia ($m = 0$);
 - ▶ árvore binária cuja raiz contém a mesma informação da raiz de T_1 ; cuja subárvore esquerda é dada por $\mathbf{B}((T_{11}, T_{12}, \dots, T_{1m_1}))$ onde $(T_{11}, T_{12}, \dots, T_{1m_1})$ é a floresta das subárvores de T_1 ; e cuja subárvore direita é dada por $\mathbf{B}((T_2, \dots, T_m))$.
- ▶ Conclui-se facilmente que toda floresta tem uma única representação binária.
- ▶ A implementação de árvores binárias é mais simples.
- ▶ Exercício: definir a transformação contrária $\mathbf{F}(T)$ que obtém a floresta a partir da árvore binária T que a representa.
- ▶ Exercício: verificar se toda árvore binária representa uma floresta.

Percursos em profundidade de florestas

Os percursos de uma floresta $F = (T_1, T_2, \dots, T_m)$ são definidos por (F_1 denota a floresta de subárvores de T_1):

- ▶ Pré-ordem de florestas:

Visitar a raiz de T_1

Percorrer a floresta F_1 em pré-ordem de florestas

Percorrer a floresta (T_2, \dots, T_m) em pré-ordem de florestas

- ▶ Pós-ordem de florestas:

Percorrer a floresta F_1 em pós-ordem de florestas

Percorrer a floresta (T_2, \dots, T_m) em pós-ordem de florestas

Visitar a raiz de T_1

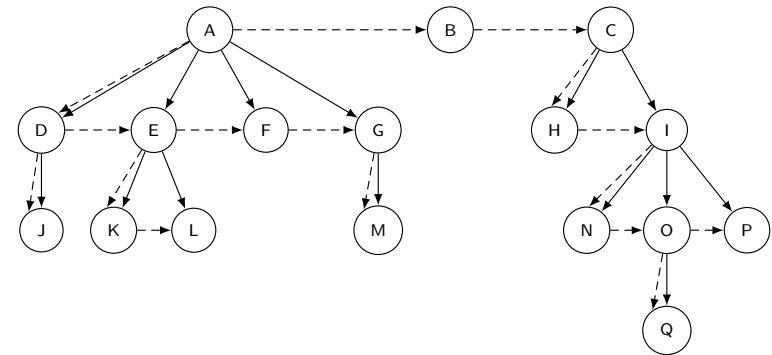
- ▶ Inordem de florestas:

Percorrer a floresta F_1 em inordem de florestas

Visitar a raiz de T_1

Percorrer a floresta (T_2, \dots, T_m) em inordem de florestas

Percursos em profundidade de florestas (cont.)



Pré-ordem: A,D,J,E,K,L,F,G,M,B,C,H,I,N,O,Q,P

Pós-ordem: J,L,K,M,G,F,E,D,Q,P,O,N,I,H,C,B,A

Inordem: J,D,K,L,E,F,M,G,A,B,H,N,Q,O,P,I,C

Percursos em profundidade de florestas (cont.)

Propriedades:

- ▶ percurso de uma floresta F produz o mesmo resultado que o percurso (binário) correspondente da árvore $\mathbf{B}(F)$.
- ▶ pré-ordem de florestas é semelhante à pré-ordem de árvores binárias
- ▶ inordem de florestas é semelhante à pós-ordem de árvores binárias
- ▶ pós-ordem de florestas não tem uma interpretação natural

Desafio:

Elabore um algoritmo para percurso em largura de árvores gerais sob representação binária.

Listas generalizadas

Conceito e exemplos

- Um *átomo* é qualquer informação; por exemplo: um inteiro, uma cadeia de caracteres, um registro com informações, etc.
- Uma *lista generalizada* é uma sequência:

$$(\alpha_1, \alpha_2, \dots, \alpha_n)$$

onde α_i denota um átomo ou uma lista generalizada (definição recursiva).

- Exemplos de listas (de inteiros):

A: ((4,7),(4,7,(8)))

B: ((1,4),(7,8))

C: (3,B,B)

D: (5,8,D)

E: ()

- Comprimentos das listas: A, B, C, D e E têm, respectivamente, 2, 2, 3, 3 e 0 elementos.
- A definição de átomo poderia ser estendida para outros tipos de valores.

Expansão de listas

A: ((4,7),(4,7,(8)))
B: ((1,4),(7,8))
C: (3,B,B)
D: (5,8,D)
E: ()

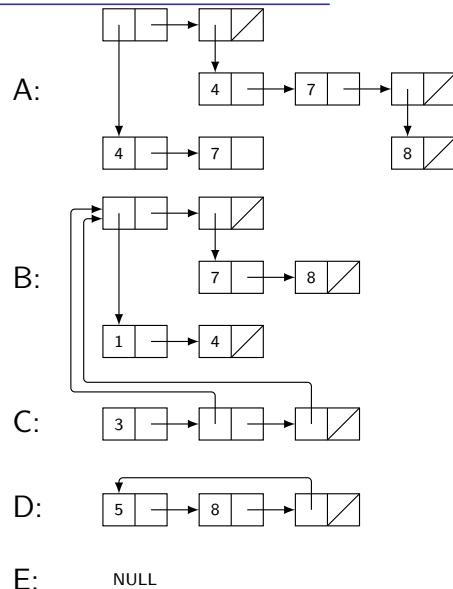
As listas C e D podem ser expandidas com as definições correspondentes:

C: (3,((1,4),(7,8)),((1,4),(7,8)))

D: (5,8,(5,8,(5,8,...)))

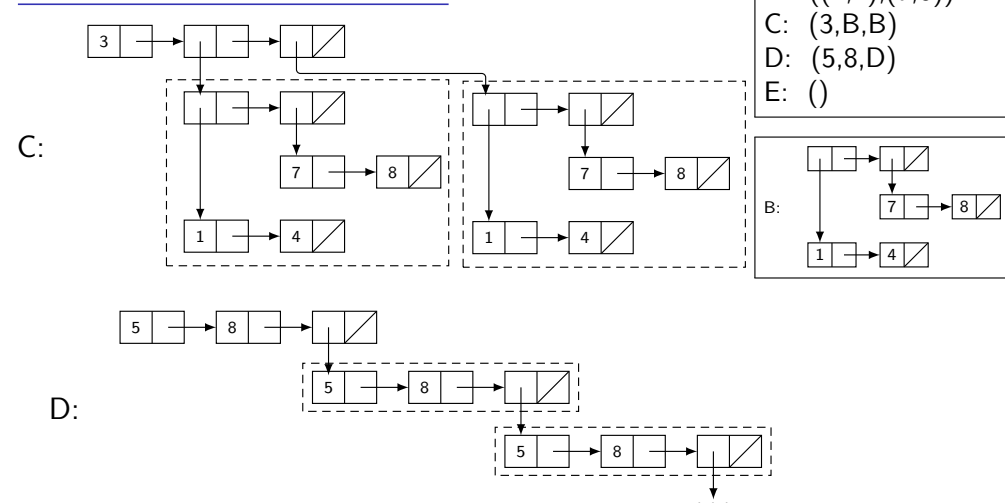
A lista D tem três elementos, mas inclui um número infinito de inteiros, por ser recursiva.

Implementação compartilhada



Há somente uma cópia da lista B; a lista D tem uma referência circular (recursiva), mas não é uma lista circular!

Implementação com cópia



- Há duas cópias da lista B.
- Não é possível completar a expansão da lista D.
- As representações das listas A, B e E não mudam.

Representação de listas generalizadas

```
typedef struct RegListaGen *ListaGen;

typedef struct RegListaGen {
    ListaGen prox;
    Boolean eAtomo;
    union {
        int atomo;           /* 'eAtomo' verdadeiro */
        ListaGen lista;      /* 'eAtomo' falso */
    } info;
} RegListaGen;
```

Exemplo de manipulação

Função de contagem de átomos:

```
int ContaAtomos(ListaGen p) {
    int s = 0;
    while (p!=NULL) {
        if (p->eAtomo)
            s++;
        else
            s += ContaAtomos(p->info.lista);
        p = p->prox;
    }
    return s;
} /* ContaAtomos */
```

Problemas com compartilhamento:

- ▶ contagem repetida (caso da lista *C*); pode ser intencional
- ▶ repetição infinita (caso da lista *D*)

Representação alternativa

```
typedef struct RegListaGen *ListaGen;

typedef struct RegListaGen {
    Boolean visitado; /* inicialmente falso */
    ListaGen prox;
    Boolean eAtomo;
    union {
        int atomo;           /* 'eAtomo' verdadeiro */
        ListaGen lista;      /* 'eAtomo' falso */
    } info;
} RegListaGen;
```

Exemplo de manipulação

Função geral de contagem de átomos:

```
int ContaAtomos(ListaGen p) {
    int s = 0;
    while ((p!=NULL) && !(p->visitado)) {
        p->visitado = true;
        if (p->eAtomo)
            s++;
        else
            s += ContaAtomos(p->info.lista);
        p = p->prox;
    }
    return s;
} /* ContaAtomos */
```

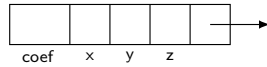
Problema: restauração dos valores do campo *visitado* para o próximo percurso.

Exemplo de aplicação

Manipulação de polinômios em múltiplas variáveis:

$$P(x, y, z) = x^{10}y^3z^2 + 2x^8y^3z^2 + 3x^8y^2z^2 + x^4y^4z - 6x^3y^4z + 2yz$$

Representação possível para cada termo:



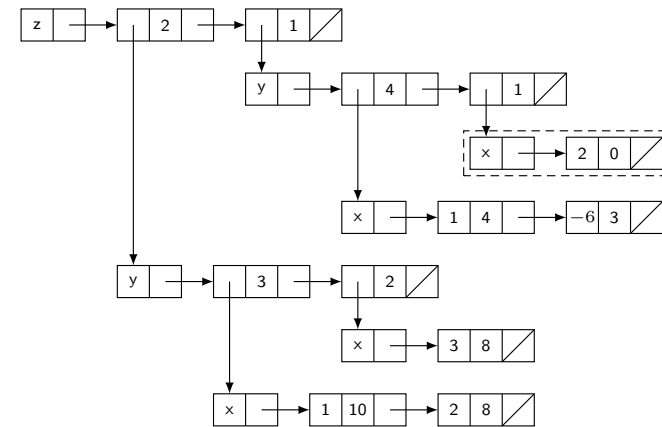
- Problema: muito inflexível, somente para polinômios em três variáveis.
- Alternativa: um polinômio em $k \geq 1$ variáveis pode ser considerado um polinômio em uma variável, com coeficientes que são polinômios em $k-1$ variáveis, etc:

$$P(x, y, z) = ((x^{10} + 2x^8)y^3 + 3x^8y^2)z^2 + ((x^4 - 6x^3)y^4 + 2x^0y)z$$

Representação de polinômios

$$((x^{10} + 2x^8)y^3 + 3x^8y^2)z^2 + ((x^4 - 6x^3)y^4 + 2x^0y)z$$

Alternativa 1: representação uniforme em todos os níveis:

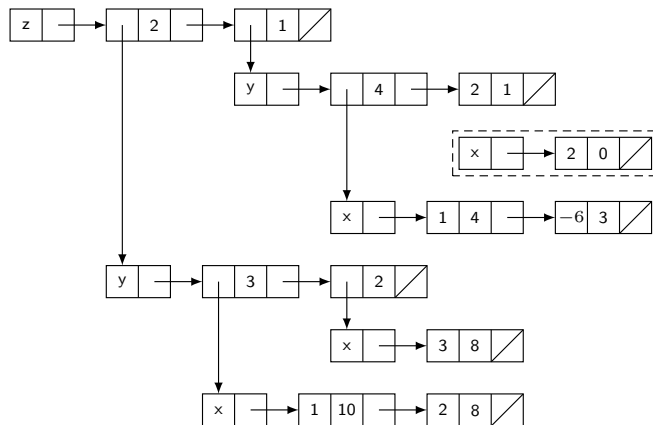


Note-se que o termo $2x^0y$ é representado de maneira completa (retângulo tracejado). Esta representação torna os algoritmos mais simples.

Representação de polinômios (cont.)

$$((x^{10} + 2x^8)y^3 + 3x^8y^2)z^2 + ((x^4 - 6x^3)y^4 + 2x^0y)z$$

Alternativa 2: representação que elimina polinômios “degenerados”



Note-se que o termo $2x^0y$ é representado como $2y$. Esta representação economiza memória (retângulo tracejado indica o nó eliminado).

Declaração de tipo

```
typedef struct Termo *ApTermo;
typedef ApTermo Polinomio;
typedef struct Termo {
    Polinomio prox;
    Boolean eCabeca;
    union {
        char variavel; /* se é cabeça */
        struct {
            int expoente;
            Boolean coefInteiro;
            union {
                int coefInt;
                Polinomio coefPolin;
            } coef;
        } termo;
    } no;
} Termo;
```

Exercício: escrever as funções de soma e de multiplicação para polinômios em múltiplas variáveis.

LISP: uma linguagem para processamento de listas

- ▶ Programas são expressos como listas.
- ▶ Dados são átomos e listas.
- ▶ Aplicações:
 - ▶ inteligência artificial
 - ▶ *scripts* para Emacs
 - ▶ *scripts* para AutoCAD
 - ▶ ...

LISP (cont.)

Exemplo 1: função fatorial

```
(defun fatorial (n)
  (cond (leq n 1)
        1
        (mult n (fatorial (minus n 1)))
  )
)
```

- ▶ A expressão: (fatorial 5) produz: 120.
- ▶ Deve-se notar o uso de notação pré-fixa.
- ▶ As implementações comuns de LISP permitem o uso de símbolos de operações como \leq , $+$ e $*$ em lugar de átomos.

LISP (cont.)

Exemplo 2: concatenação e inversão de listas

```
(defun concat (p q)
  (cond (null p)
        q
        (cons (car p) (concat (cdr p) q))
  )
)
(defun invert (p)
  (cond (null p)
        nil
        (concat (invert (cdr p)) ((car p)))
  )
)
```

- ▶ A expressão: (invert '(A B C D)) produz D C B A.
- ▶ A expressão (car L) devolve o primeiro elemento da lista L; a expressão (cdr L) devolve a lista L sem o primeiro elemento.
- ▶ A operação (cons x L) devolve a lista L com o elemento x inserido na frente da lista, isto é: $(\text{cons } \alpha (\alpha_1 \alpha_2 \dots)) = (\alpha \alpha_1 \alpha_2 \dots)$.
- ▶ **Obs.:** A função invert é muito ineficiente (quadrática!).

Espalhamento (Hashing ou scattering)

Tabelas de espalhamento

Exemplo de tabela com $b=7$ linhas e $s=3$ colunas:

$f('joão') \rightarrow$

	1	2	3
0			
1			
2			
3	joão		
4			
5			
6			

Supõe-se, neste caso, que:

- ▶ a função de espalhamento f produz resultados entre 0 e 6
- ▶ $f('joão') = 3$
- ▶ existem no máximo três valores (s) a serem inseridos que produzem o mesmo valor da função f .

Tabelas de espalhamento (cont.)

Exemplo de tabela com $b=26$ linhas e $s=2$ colunas:

	1	2
0	antônio	áttila
1		
2	carlos	célio
3	douglas	
4	ernesto	estêvão
5		
...		
24		
25	zoroastro	

Foi usada uma função (muito ingênua!) de espalhamento: índice da primeira letra (a: 0, b: 1, ...), desconsiderando os acentos etc.

Virtudes e problemas

- ▶ Virtudes
 - ▶ simplicidade
 - ▶ busca muito rápida (se a função de espalhamento for eficiente)
- ▶ Problemas
 - ▶ escolha da função de espalhamento
 - ▶ tratamento de colisões
 - ▶ tratamento de estouro da tabela

Construção de funções de espalhamento

- ▶ Propriedades desejáveis:
 - ▶ eficiência de cálculo
 - ▶ bom espalhamento
- ▶ Técnicas:
 - ▶ espalhamento mínimo perfeito
 - ▶ espalhamento pseudo-aleatório: combinação de várias técnicas

Construção de funções de espalhamento (cont.)

Divisão

- ▶ O nome, tratado como um número na base 26, é dividido por um número p relativamente primo $f(x) = x \bmod p$; p será adotado como o número de linhas da tabela.
- ▶ Para $p = 51$ teríamos:

$$\begin{aligned} f(\text{carlos}) &= (((((2 \times 26 + 0) \times 26 + 17) \times 26 + 11) \\ &\quad \times 26 + 14) \times 26 + 18) \bmod 51 \\ &= 24.069.362 \bmod 51 = 14 \end{aligned}$$

- ▶ Na realidade, o cálculo pode ser simplificado, com a operação **mod** aplicada a cada passo.

Construção de funções de espalhamento (cont.)

Seleção de algarismos e meio-do-quadrado

- ▶ O nome é tratado como uma sequência de algarismos ou de *bytes* ou de *bits*, e uma subsequência é selecionada para representar o índice.
- ▶ Por exemplo, suponhamos que todos os nomes são representados como a sequência de dígitos $x = d_0 d_1 \dots d_{11}$ em alguma base conveniente; uma escolha seria $f(x) = d_3 d_5 d_9$.
- ▶ Exemplo: a representação de 'carlos' poderia ser 020017111418. Supusemos que cada letra é indicada por dois dígitos que indicam a posição no alfabeto, ou seja 00 para 'a', 01 para 'b', etc. Teríamos então $f(\text{carlos}) = 074$.
- ▶ Frequentemente, antes de fazer a seleção, é calculado o quadrado do identificador (tratado como número); é o método "meio-do-quadrado" (*mid-square*).

Construção de funções de espalhamento (cont.)

Dobramento (*folding*)

- ▶ O nome é tratado como uma sequência de algarismos ou de *bytes* ou de *bits*, e algumas subsequências são combinadas por operações convenientes para produzir um índice.
- ▶ Por exemplo, suponhamos que todos os nomes são representados como uma sequência de *bits* $x = b_0 b_1 b_2 b_3 b_4 \dots$; uma escolha seria:

$$f(x) = b_0 b_1 b_2 b_3 b_4 b_5 \oplus b_6 b_7 b_8 b_9 b_{10} b_{11} \oplus \dots$$

onde \oplus denota a operação de *ou exclusivo bit a bit*.

- ▶ Exemplo: a representação de 'carlos' usada anteriormente poderia ser (com cinco *bits* para cada número):
00010 00000 10001 01011 01110 10010
produzindo a sequência de bits: 000100000010001010110111010010 e o resultado:

$$\begin{aligned} f(000100000010001010110111010010) &= \\ 000100 \oplus 000010 \oplus 001010 \oplus 110111 \oplus 010010 &= 101001 = 41_{10} \end{aligned}$$

Tratamento de colisões: endereçamento aberto

- ▶ Busca sistemática de outras entradas disponíveis na tabela:
 - ▶ reespalhamento linear
 - ▶ reespalhamento quadrático
 - ▶ reespalhamento duplo
- ▶ Em todos os casos, os algoritmos de busca, inserção e remoção deverão ser coerentes.
- ▶ Exemplos usam: antônio, carlos, douglas, célio, armando, zoroastro, átila, alfredo (nesta ordem).

Reespalhamento linear

antônio, carlos, douglas, célio, armando, zoroastro, átila, alfredo

(a:0, c:2, d:3, z:25)

Usando $(f(x) + i) \bmod b, (i = 0, 1, \dots)$, procura a primeira posição livre.

0	antônio
1	armando
2	carlos
3	douglas
4	célio
5	átila
6	alfredo
7	
...	...
25	zoroastro

Reespalhamento quadrático

antônio, carlos, douglas, célio, armando, zoroastro, átila, alfredo

(a:0, c:2, d:3, z:25)

Usando $(f(x) + i^2) \bmod b, (i = 0, 1, \dots)$, procura a primeira posição livre.

0	antônio
1	armando
2	carlos
3	douglas
4	átila
5	
6	célio
7	
8	
9	alfredo
...	...
25	zoroastro

Reespalhamento duplo

antônio, carlos, douglas, célio, armando, zoroastro, átila, alfredo

(a:0, c:2, d:3, e:4, l:11, r:17, t:19, z:25)


- ▶ Usando $(f(x) + i \times g(x)) \bmod b, (i = 0, 1, \dots)$ procura a primeira posição livre.
- ▶ $g(x)$ é a função de reespalhamento; por exemplo, $g(x) = (c \bmod 3) + 1$ onde c é a segunda letra (também muito ingênua!).

0	antônio
1	
2	carlos
3	douglas
4	célio
5	
6	armando
7	
8	átila
9	alfredo
...	...
25	zoroastro

Remoção

- ▶ Lápides (*tombstones*):
 - ▶ entradas que indicam posições ocupadas para fins de busca, mas livres para fins de inserção.
 - ▶ podem ser usadas com qualquer esquema de espalhamento
- ▶ Exemplo: remoção da entrada armando (tabela com reespalhamento linear):

0	antônio
1	armando
2	carlos
3	douglas
4	célio
5	átila
6	alfredo
7	
...	...
25	zoroastro



0	antônio
1	++++++
2	carlos
3	douglas
4	célio
5	átila
6	alfredo
7	
...	...
25	zoroastro

Eficiência com endereçamento aberto

- ▶ Número médio de comparações para encontrar um elemento:

$$C(n) = (2 - \alpha)/(2 - 2\alpha)$$

onde:

- ▶ $\alpha = n/b$ (fator de carga, $0 \leq \alpha \leq 1$)
- ▶ n é o número de entradas
- ▶ b é o tamanho da tabela
- ▶ Note-se que $C(n) \rightarrow \infty$ para $n \rightarrow b$ (i. é., $\alpha \rightarrow 1$).
- ▶ Exemplo de tabela de tamanho 1000:

n	100	200	300	400	500	600	700	800	900	950	975
$C(n)$	1,06	1,13	1,21	1,33	1,50	1,75	2,17	3,00	5,50	10,5	20,5

- ▶ Pode-se demonstrar que o número médio de comparações numa árvore binária de 511 nós, perfeitamente balanceada (isto é, de altura igual a 9), é de cerca de 8.

Estouro da tabela com endereçamento aberto

- ▶ Problemas:

- ▶ À medida que cresce o fator de carga, a eficiência de busca cai.
- ▶ Quando a tabela fica cheia, não é possível inserir mais elementos.

- ▶ Solução:

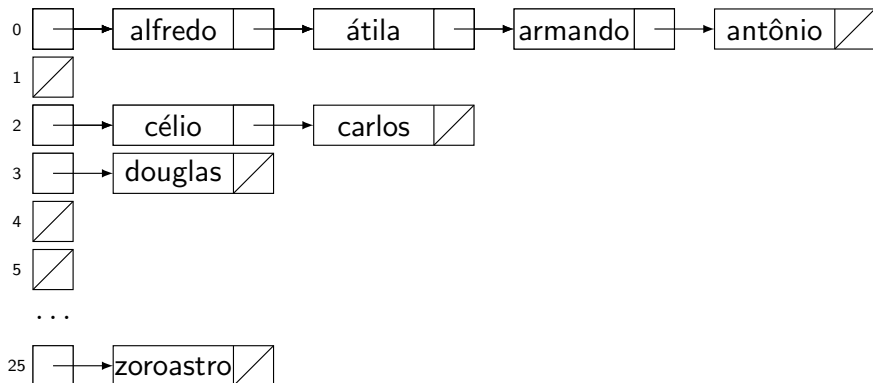
- ▶ Criar uma nova tabela maior (por exemplo, o dobro do tamanho).
- ▶ Redefinir a(s) função(ões) de espalhamento.
- ▶ Reespalhar os elementos na nova tabela.

Tratamento de colisões: listas ligadas

Técnica de encadeamento (*chaining*): utiliza listas ligadas para manter entradas com o mesmo valor da função de espalhamento.

Exemplo: (a:0, c:2, d:3, z:25)

antônio, carlos, douglas, célio, armando, zoroastro, átila, alfredo



Os apontadores aos inícios das listas constituem um vetor indexado pela função de espalhamento. As listas poderiam ser ordenadas.

Eficiência com encadeamento

- ▶ Número médio de comparações para encontrar um elemento:

$$C(n) = 1 + \alpha/2$$

onde:

- ▶ $\alpha = n/b$ (fator de carga, $\alpha > 0$)
- ▶ n é o número de entradas
- ▶ b é o tamanho da tabela

- ▶ Exemplo de tabela com tamanho 1000:

n	100	200	400	500	1000	2000	4000	8000
$C(n)$	1,05	1,10	1,20	1,25	1,50	2,00	3,00	5,00

- ▶ Em princípio, uma tabela com encadeamento não está sujeita a estouros (exceto da memória total).
- ▶ Em caso de aumento excessivo do fator de carga, pode-se adotar uma solução semelhante: aumento do tamanho da tabela.

Compressão de textos

(Codificação de Huffman)

Compressão de textos

- ▶ Objetivos:
 - ▶ economia de espaço
 - ▶ velocidade de transmissão
- ▶ Representação normal: um *byte* (8 *bits*) por caractere (alfabetos “comuns”)
- ▶ Português: padrão ISO-8859-1
- ▶ Outras representações: UTF-8 (um ou dois *bytes* por letra)
- ▶ Compressão por contagem (*run-length encoding*)
- ▶ Codificação de Huffman
- ▶ Algoritmos de codificação aritmética (Lempel-Ziv): zip, gzip, winzip, etc.)

Codificação de Huffman

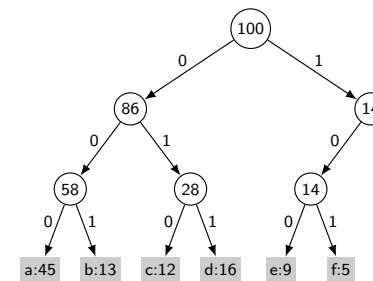
- ▶ Explora frequências de ocorrência de caracteres
- ▶ Exemplo de alfabeto: $\mathcal{A} = \{a, b, c, d, e, f\}$

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>
frequência de cada letra	45	13	12	16	9	5
codificação fixa usando 3 <i>bits</i>	000	001	010	011	100	101
codificação de tamanho variável	0	101	100	111	1101	1100

- ▶ Para um arquivo de 100.000 caracteres:
 - ▶ codificação fixa: 300.000 bits
 - ▶ codificação variável: 224.000 bits (economia de 25%)

Árvores binárias de codificação

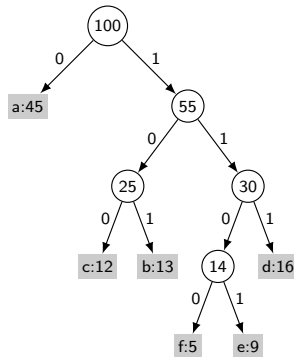
Codificação fixa a: 000 b: 001 c: 010 d: 011 e: 100 f: 101



- ▶ Os rótulos das arestas da raiz até uma folha compõem o código da letra correspondente (0: subárvore esquerda, 1: subárvore direita).
- ▶ Para obter o código de uma letra, é necessário percorrer a árvore partindo da folha correspondente até a raiz.
- ▶ Cada nó inclui a frequência acumulada da árvore da qual é raiz.
- ▶ Exemplo de codificação: $abc = 000||001||010 = 000001010$.

Árvores binárias de codificação (cont.)

Codificação variável a: 0 b: 101 c: 100 d: 111 e: 1101 f: 1100



- ▶ O código de uma letra não pode constituir um prefixo de uma outra letra.
- ▶ Exemplo de codificação: $abc = 0||101||100 = 0101100$.

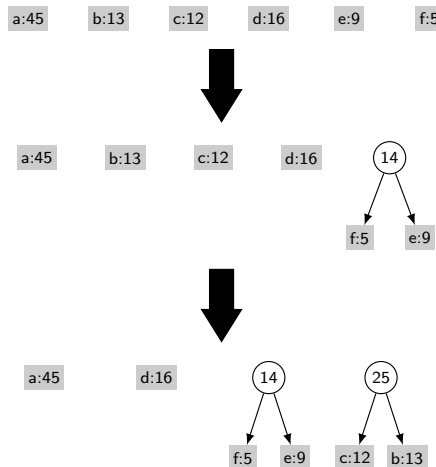
Construção da árvore de Huffman

▶ Algoritmo (guloso):

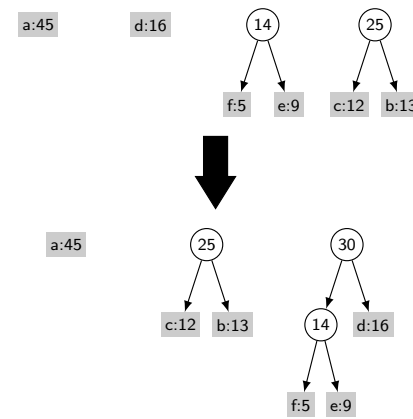
1. Construa uma floresta de folhas, cada uma correspondendo a um caractere, com a respectiva frequência como seu peso.
 2. Enquanto a floresta tiver mais de uma árvore, repita:
 - ▶ procure na floresta duas árvores t_1 e t_2 de menor peso
 - ▶ construa uma nova árvore binária t , com subárvores t_1 e t_2 , e com peso que é a soma dos pesos das duas subárvores
 - ▶ remova t_1 e t_2 da floresta, e insira t .
- ▶ A solução não é única (pode haver várias escolhas de peso mínimo), mas todos os resultados são equivalentes quanto à eficiência de compressão.
- ▶ Se o alfabeto for razoavelmente grande, pode-se utilizar uma fila de prioridade para selecionar, em cada passo, duas árvores de menor peso.

Construção da árvore de Huffman (cont.)

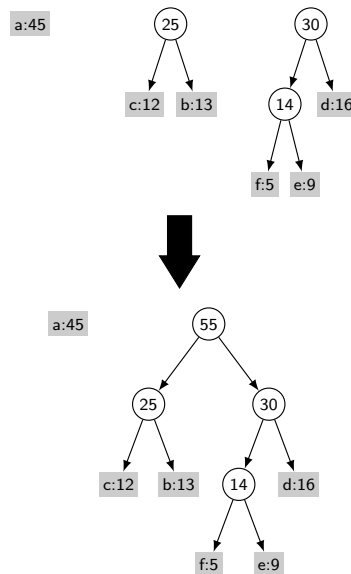
Exemplo:



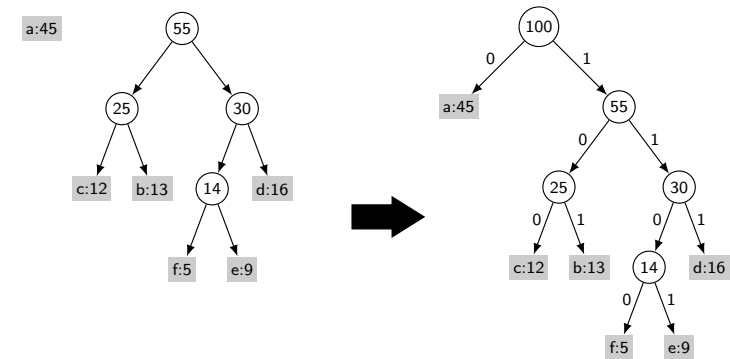
Construção da árvore de Huffman (cont.)



Construção da árvore de Huffman (cont.)



Construção da árvore de Huffman (cont.)



Observações

- ▶ Adotadas certas hipóteses, demonstra-se a otimalidade de compressão
- ▶ Algoritmo de compressão: para cada letra, deve acessar a folha correspondente da árvore e reconstruir o caminho à raiz – pode ser preprocessado:

a: 0 , b: 101, c: 100, d: 111, e: 1101, f: 1100

- ▶ Algoritmo de descompressão: percorre a árvore a partir da raiz seguindo o caminho indicado por *bits* da codificação
- ▶ Variantes:
 - ▶ árvore fixa, por exemplo, uma para cada língua
 - ▶ árvore por texto (acompanha o arquivo)
 - ▶ árvores dinâmicas (Faller, Gallager e Knuth).

Abstração de Dados e Objetos

Tipos abstratos de dados

- ▶ Um *tipo abstrato de dados* (TAD) é constituído por um conjunto de valores e um conjunto de operações sobre estes valores.
- ▶ Os valores possuem uma representação e podem ser muito simples (inteiros, *bytes*, ...) ou bastante complexos (pilhas, árvores, ...).
- ▶ Exemplo de especificação de um TAD *Figura* através de declarações em C:

```
typedef void * Figura;  
Figura Retangulo(float alt, float larg);  
Figura Circulo(float raio);  
Figura Quadrado(float lado);  
float Area(Figura fig);  
void Transladar(Figura fig, float dx, float dy);  
void Desenhar(Figura fig);
```

- ▶ A especificação de um tipo como “**void ***” é uma técnica comum em C para “esconder” a implementação.
- ▶ Normalmente, estas declarações, chamadas às vezes de *interface* ou *API* (*Application Programming Interface*) estariam num arquivo denominado *figuras.h*.

Tipos abstratos de dados (cont.)

- ▶ Usando a especificação, é possível escrever programas que utilizam o TAD, mesmo sem completar a sua implementação.
- ▶ Exemplo de utilização do TAD *Figura*:

```
#include "figuras.h"  
int main() {  
    Figura c = Circulo(10.0);  
    Figura r = Retangulo(10.0, 20.0);  
    Figura q = Quadrado(50.0);  
    Transladar(r, 5.0, 8.0);  
    Desenhar(q);  
    printf("%f\n", Area(c));  
    printf("%f\n", Area(r));  
    printf("%f\n", Area(q));  
    return 0;  
} /* main */
```

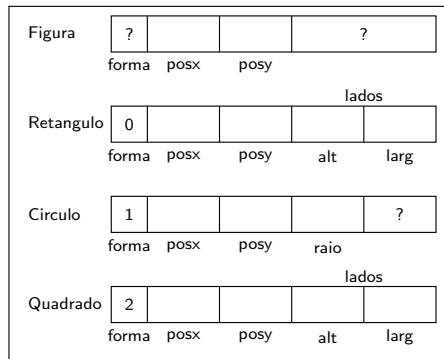
```
typedef void * Figura;  
Figura Retangulo(float alt, float larg);  
Figura Circulo(float raio);  
Figura Quadrado(float lado);  
float Area(Figura fig);  
void Transladar(Figura fig, float dx, float dy);  
void Desenhar(Figura fig);
```

- ▶ As funções *Circulo*, *Retangulo* e *Quadrado* devem construir e devolver as representações dos valores correspondentes (construtores).
- ▶ A função *main* poderia estar num arquivo denominado *main.c*.

Tipos abstratos de dados (cont.)

- ▶ A implementação de um TAD depende de vários fatores, mas deve seguir sempre a sua especificação.
- ▶ Exemplo de declarações “naturais” para implementar o TAD *Figura*:

```
typedef enum { RETANGULO, CIRCULO, QUADRADO } FormaFigura;  
typedef struct {  
    FormaFigura forma;  
    float posx, posy;  
    union {  
        struct {  
            float alt, larg;  
        } lados;  
        float raio;  
    } dados;  
} ImplRegFigura, *ImplFigura;
```



- ▶ Deve-se notar a diferença entre os tipos *Figura* (*void **) e *ImplFigura*.
- ▶ Normalmente, estas declarações (e as seguintes) estariam num arquivo *figuras.c*.

Tipos abstratos de dados (cont.)

Declarações dos construtores:

```
Figura Retangulo(float alt, float larg) {  
    ImplFigura f =  
        malloc(sizeof(ImplRegFigura));  
    f->forma = RETANGULO;  
    f->posx = 0.0;  
    f->posy = 0.0;  
    f->dados.lados.alt = alt;  
    f->dados.lados.larg = larg;  
    return (Figura)f;  
} /* Retangulo */
```

```
Figura Circulo(float raio) {  
    ImplFigura f =  
        malloc(sizeof(ImplRegFigura));  
    f->forma = CIRCULO;  
    f->posx = 0.0;  
    f->posy = 0.0;  
    f->dados.raio = raio;  
    return (Figura)f;  
} /* Circulo */
```

```
Figura Quadrado(float lado) {  
    ImplFigura f =  
        Retangulo(lado, lado);  
    f->forma = QUADRADO;  
    return (Figura)f;  
} /* Quadrado */
```


Tipos abstratos de dados (cont.)

Declarações das funções:

```
float Area(Figura fig) {
    ImplFigura f=(ImplFigura)fig;
    switch (f->forma) {
        case RETANGULO:
        case QUADRADO:
            return (f->dados.lados.alt)*
                (f->dados.lados.larg);
        case CIRCULO:
            return PI*(f->dados.raio)*
                (f->dados.raio);
        default:
            exit(1); /* Impossível */
    }
} /* Area */

void Transladar(Figura fig ,
                float dx,
                float dy) {
    ImplFigura f=(ImplFigura)fig;
    f->posx += dx;
    f->posy += dy;
} /* Transladar */

void Desenhar(Figura f) {
    /* Não foi implementada */
} /* Desenhar */
```

Tipos abstratos de dados (cont.)

- ▶ Nesta implementação do TAD *Figura*, a estrutura de dados que implementa o tipo e as funções são implementadas separadamente.
- ▶ É possível mudar a implementação de maneira que as funções passem fazer parte da própria estrutura de dados – uma característica de objetos; neste caso são denominados *métodos*.
- ▶ Nesta nova implementação do exemplo, por simplicidade, a técnica será aplicada somente à função *Area*, mas poderia ser estendida às outras funções.
- ▶ Trata-se de uma nova implementação da mesma interface; consequentemente os arquivos *figuras.h* (repetido abaixo) e *main.c* permanecem iguais.

```
typedef void * Figura;
Figura Retangulo(float alt, float larg);
Figura Circulo(float raio);
Figura Quadrado(float lado);
float Area(Figura fig);
void Transladar(Figura fig, float dx, float dy);
void Desenhar(Figura fig);
```

Tipos abstratos de dados (cont.)

Declaração de *Figura* com um método:

```
typedef float funcArea(Figura); /* tipo função */
typedef enum { RETANGULO, CIRCULO, QUADRADO } FormaFigura;

typedef struct {
    FormaFigura forma;
    float posx, posy;
    funcArea *Area; /* apontador para função */
    union {
        struct {float alt, larg; } lados;
        float raio;
    } dados;
} ImplRegFigura, * ImplFigura;
```

Tipos abstratos de dados (cont.)

Funções do tipo *funcArea*:

```
float AreaRetangulo(Figura fig) {
    ImplFigura f = (ImplFigura)fig;
    return (f->dados.lados.alt)*(f->dados.lados.larg);
} /* AreaRetangulo */

float AreaCirculo(Figura fig) {
    ImplFigura f = (ImplFigura)fig;
    return PI*(f->dados.raio)*(f->dados.raio);
} /* AreaCirculo */
```

Tipos abstratos de dados (cont.)

Declarações dos construtores:

```
Figura Retangulo(float alt,
                float larg) {
    ImplFigura f =
        malloc(sizeof(ImplRegFigura));
    f->forma = RETANGULO;
    f->posx = 0.0;
    f->posy = 0.0;
    f->Area = AreaRetangulo;
    f->dados.lados.alt = alt;
    f->dados.lados.larg = larg;
    return (Figura)f;
} /* Retangulo */

Figura Circulo(float raio) {
    ImplFigura f =
        malloc(sizeof(ImplRegFigura));
    f->forma = CIRCULO;
    f->posx = 0.0;
    f->posy = 0.0;
    f->Area = AreaCirculo;
    f->dados.raio = raio;
    return (Figura)f;
} /* Circulo */

Figura Quadrado(float lado) {
    ImplFigura f =
        Retangulo(lado, lado);
    f->forma = QUADRADO;
    return (Figura)f;
} /* Quadrado */
```

Tipos abstratos de dados (cont.)

Declarações das funções:

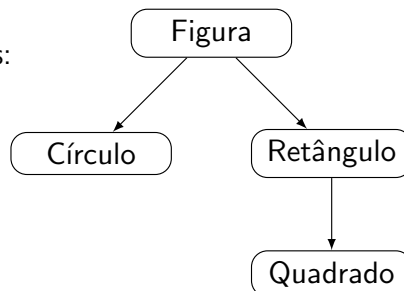
```
float Area(Figura fig) {
    return ((ImplFigura)fig)->Area(fig);
}

void Transladar(Figura fig, float dx, float dy) {
    ImplFigura f = (ImplFigura)fig;
    f->posx += dx;
    f->posy += dy;
} /* Transladar */

void Desenhar(Figura fig) {
    /* Não foi implementada */
} /* Desenhar */
```

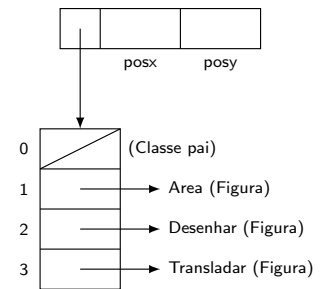
Objetos

- ▶ O exemplo anterior demonstra que é possível simular, dentro de algumas limitações, a implementação de objetos numa linguagem que não incorpora este conceito.
- ▶ Há vários aspectos que ficam a cargo do próprio programador, especialmente a consistência de tipos, fonte comum de erros.
- ▶ O exemplo anterior será transformado de maneira a ilustrar a implementação de objetos numa linguagem que possui este conceito.
- ▶ Será usada uma linguagem fictícia, uma extensão simples de C.
- ▶ Não serão tratados vários aspectos como por exemplo polimorfismo, visibilidade etc.
- ▶ Exemplo de hierarquia das classes:



Objetos (cont.)

```
class Figura {
    float posx, posy;
    /* não existe construtor */
    float Area() virtual;
    void Desenhar() virtual;
    float Transladar(float dx, dy) {
        this->posx += dx;
        this->posy += dy;
    }
} /* Figura */
```

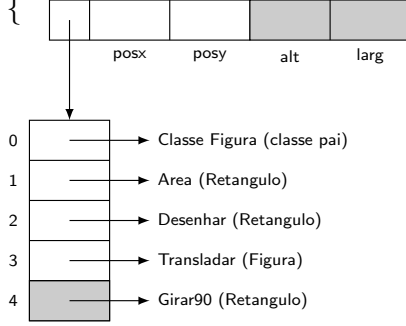


Todos os objetos de uma classe apontam para a mesma tabela de métodos. Pode haver mais informações. Neste exemplo, todas as funções foram transformadas em métodos.

A classe *Figura* possui métodos virtuais que deverão ser definidos nas classes derivadas. Consequentemente, a classe é abstrata e não tem construtor.

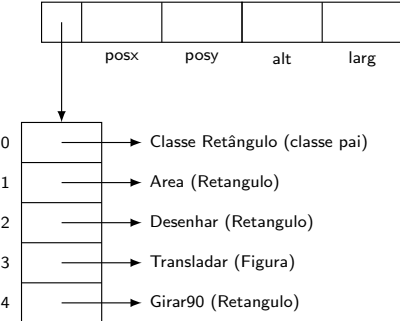
Objetos (cont.)

```
class Retangulo extends Figura {  
    float alt, larg;  
    Retangulo(float a, float l) {  
        this.alt = a;  
        this.larg = l;  
        this.posx = 0.0;  
        this.posy = 0.0;  
    }  
    float Area() {  
        return alt*larg;  
    }  
    void Desenhar() { ... } /* omitido */;  
    Retangulo Girar90() {  
        return new Retangulo(this.larg, this.alt);  
    }  
} /* Retangulo */
```



Objetos (cont.)

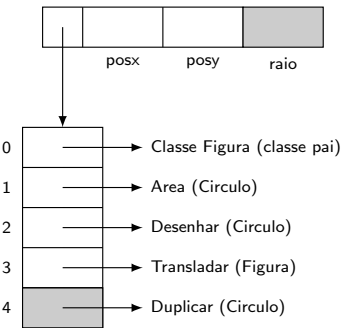
```
class Quadrado extends Retangulo {  
    Quadrado(float l) {  
        super(l, l);  
    }  
} /* Quadrado */
```



Somente o construtor é diferente da classe *Retangulo*.

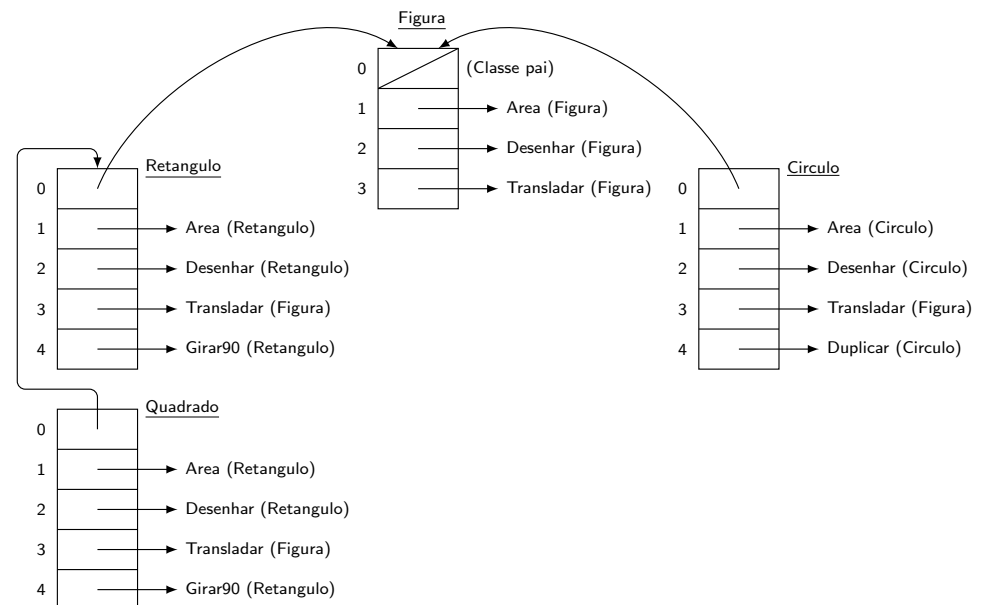
Objetos (cont.)

```
class Circulo extends Figura {  
    float raio;  
    Circulo(float r) {  
        this.posx = 0.0;  
        this.posy = 0.0;  
        this.raio = r;  
    }  
    float Area() {  
        return PI*sqr(raio);  
    }  
    void Desenhar() { ... } /* omitido */;  
    void Duplicar() {  
        this.raio = 2.0*this.raio;  
    }  
} /* Circulo */
```



Objetos (cont.)

Representação de todas as classes:



Objetos (cont.)

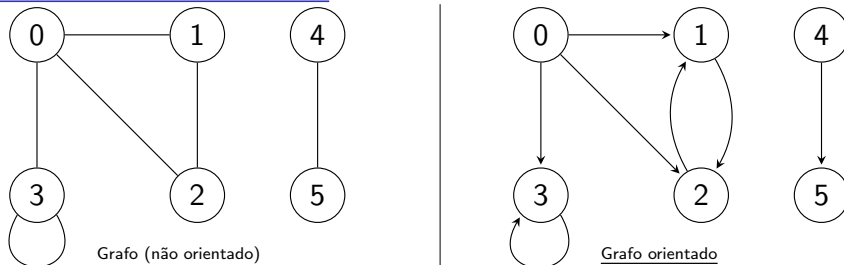
Exemplo de uso dos objetos:

```
int main() {  
    Figura f = new Circulo(10.0);  
    Retangulo r =  
        new Retangulo(10.0,20.0);  
    Quadrado q = new Quadrado(50.0);  
    Circulo c = new Circulo(30.0);  
    printf("%f\n", f.Area());  
    printf("%f\n", r.Area());  
    c.Desenhar();  
    f = q;  
    f.Transladar(5.0,8.0);  
    f.Desenhar();  
    printf("%f\n", f.Area());  
  
    /* comandos inválidos */  
    f.Duplicar();  
    f.Girar90();  
    c.Girar90();  
    c = r;  
    q = f;  
}  
/* main */
```

Os comandos inválidos seriam detectados pelo compilador.

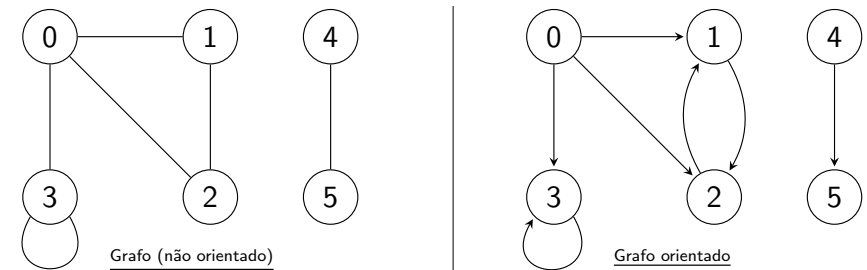
Algoritmos em grafos

Exemplos e definições



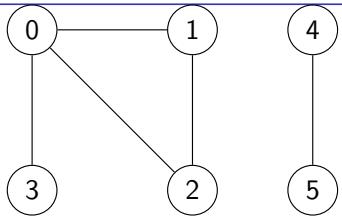
- ▶ Um *grafo* G é um par (V, E) onde V é o conjunto de *vértices* e E é o conjunto de arestas.
- ▶ Num grafo não orientado, E é um conjunto de pares não ordenados de vértices (u, v) .
- ▶ Num grafo orientado, E é um conjunto de pares ordenados de vértices (u, v) (u para v).
- ▶ Alguns autores não incluem a possibilidade de arestas da forma (u, u) (as arestas $(3, 3)$ nos dois exemplos).
- ▶ Outros autores incluem a possibilidade de arestas múltiplas (as arestas $(1, 2)$ e $(2, 1)$ do grafo orientado não são múltiplas).

Exemplos e definições (cont.)

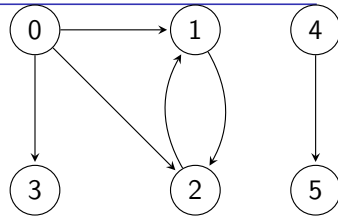


- ▶ Nos dois casos: $V = \{0, 1, 2, 3, 4, 5\}$.
- ▶ Grafo não orientado: $E = \{(0, 1), (0, 2), (0, 3), (1, 2), (3, 3), (4, 5)\}$ (6 arestas – ordem dos vértices em cada par é irrelevante).
- ▶ Grafo orientado: $E = \{(0, 1), (0, 2), (0, 3), (1, 2), (2, 1), (3, 3), (4, 5)\}$ (7 arestas).

Representação de grafos: matrizes de adjacências



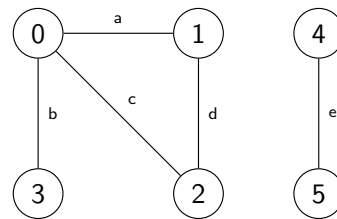
	0	1	2	3	4	5
0	0	1	1	1	0	0
1	1	0	1	0	0	0
2	1	1	0	0	0	0
3	1	0	0	0	0	0
4	0	0	0	0	0	1
5	0	0	0	0	1	0



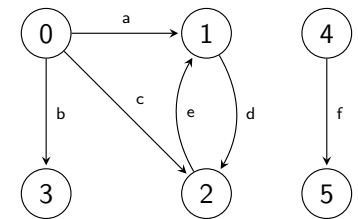
	0	1	2	3	4	5
0	0	1	1	1	0	0
1	0	0	1	0	0	0
2	0	1	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	1
5	0	0	0	0	0	0

A matriz é quadrada. $A_{ij} = 1$ se o par (i, j) é uma aresta do grafo. No caso do grafo não orientado $A_{ij} = A_{ji}$ e a matriz é simétrica (informação redundante).

Representação de grafos: matrizes de incidências



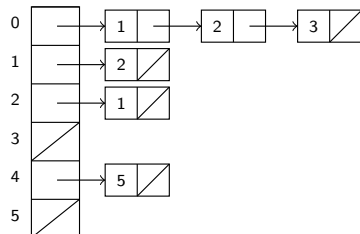
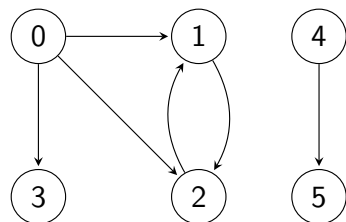
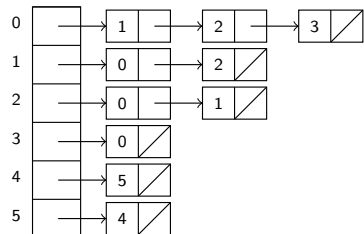
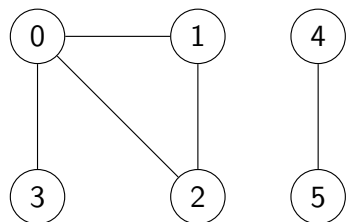
	a	b	c	d	e
0	1	1	1	0	0
1	1	0	0	1	0
2	0	0	1	1	0
3	0	1	0	0	0
4	0	0	0	0	1
5	0	0	0	0	1



	a	b	c	d	e	f
0	-1	-1	-1	0	0	0
1	+1	0	0	-1	+1	0
2	0	0	+1	+1	-1	0
3	0	+1	0	0	0	0
4	0	0	0	0	0	-1
5	0	0	0	0	0	+1

A matriz é retangular e as arestas recebem rótulos. No caso não orientado, $A_{ip} = 1$ se a aresta p tem uma extremidade i . No caso orientado, $A_{ip} = -1$ ou $A_{ip} = +1$, conforme a aresta p tenha origem ou destino i .

Representação de grafos: listas de adjacências



Para cada vértice, a lista ligada i contém os vértices adjacentes. No caso não orientado há redundância porque cada aresta é representada duas vezes. No caso orientado, são representadas arestas com origem em i .

Representação de grafos: observações

Seja $G = (V, E)$ um grafo e sejam $n = |V|$ (número de vértices) e $m = |E|$ (número de arestas) de G .

- A matriz de adjacências tem n^2 elementos. No caso de grafos esparsos, com número de arestas muito menor que n^2 , a matriz usa muita memória e os algoritmos tendem a ser ineficientes.
- A matriz de incidências tem $n \times m$ elementos, com apenas dois elementos não nulos em cada coluna; também há uso de muita memória e os algoritmos tendem a ser ineficientes.
- As listas de adjacências ocupam memória da ordem de $O(n + m)$ e resultam, em geral, em algoritmos mais eficientes.
- A redundância das listas de adjacências para grafos não orientados é compensada pela facilidade de acesso à informação.

Aplicações e algoritmos básicos

▶ Aplicações:

- ▶ Modelagem de relações em vários campos da ciência
- ▶ Modelos de redes de computadores
- ▶ Modelagem de circuitos elétricos
- ▶ Modelos de mapas geográficos (teorema das quatro cores!)
- ▶ Otimização em vários contextos (ex. problema do caixeiro viajante)
- ▶

▶ Alguns algoritmos básicos:

- ▶ Detalhes dependem da aplicação
- ▶ Busca em profundidade
- ▶ Busca em largura
- ▶ Caminhos de custo mínimo (Dijkstra)
- ▶ Os algoritmos serão esboçados em pseudo-C
- ▶ Exercício: transformar os esboços em funções completas utilizando as listas de adjacências para representar os grafos

Busca em profundidade

(Depth first search – DFS)

```
void DFS(Grafo G, int r, void (*visita)(Grafo,int)) {  
    if (!G.visitado(r)) {  
        G.visitado(r) = true;  
        visita(G,r);  
        for (v in G.Adj(r))  
            DFS(G,v,visita);  
    } /* DFS */  
}
```

- ▶ O grafo $G = (V, E)$ pode ser orientado ou não.
- ▶ O primeiro argumento r pode ser qualquer vértice do grafo.
- ▶ Se o grafo não for conexo (i. é., se houver vértices não alcançáveis a partir do inicial), haverá vértices não visitados.
- ▶ A função $G.Adj(u)$ devolve os vértices adjacentes a u .
- ▶ O número de operações é dado por $O(|E| + |V|)$.
- ▶ Exercício: modifique o algoritmo para forçar a visita a todos os vértices.
- ▶ Exercício: esboce uma implementação usando uma pilha explícita em lugar de recursão.

Busca em largura

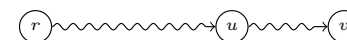
(Breadth first search – BFS)

```
void BFS(Grafo G, int r, void (*visita)(Grafo,int)) {  
    Fila f;    int u,v;  
    InicializaFila(f);    InsereFila(f,r);  
    while (!FilaVazia(Q)) {  
        RemoveFila(Q,&v);  
        if (!G.visitado(v)) {  
            G.visitado(v) = true;  
            for (u in G.Adj(v))  
                if (!G.visitado(u))    InsereFila(Q,u);  
        }  
    }  
} /* BFS */
```

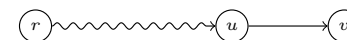
- ▶ O grafo $G = (V, E)$ pode ser orientado ou não.
- ▶ O argumento r pode ser qualquer vértice do grafo.
- ▶ Se o grafo não for conexo (i. é., se houver vértices não alcançáveis a partir de r), haverá vértices não visitados.
- ▶ O número de operações é dado por $O(|E| + |V|)$.
- ▶ Exercício: modifique o algoritmo para forçar a visita a todos os vértices.

Algoritmo de Dijkstra (1956)

- ▶ A função *Dijkstra* a seguir determina os caminhos de custo (distância) mínimo de um vértice dado (*origem*) a todos os outros vértices do grafo, a partir dos custos associados às arestas.
- ▶ A ideia básica do algoritmo: se o caminho mais curto da origem r até um vértice v passa pelo vértice u , então o caminho mais curto de r a u é a parte inicial deste caminho:



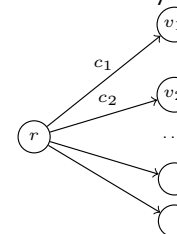
Em particular:



▶ Inicialmente:



Primeira iteração:



v_i 's são os vértices adjacentes a r , e c_i 's são os pesos das arestas correspondentes

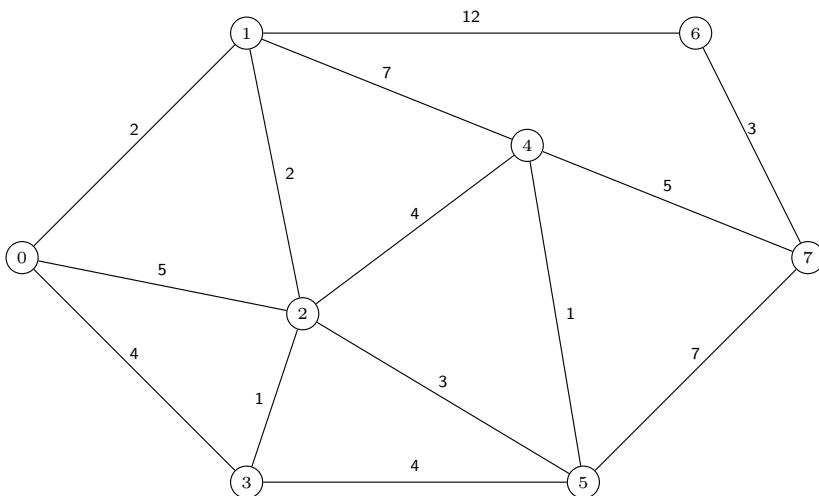
Algoritmo de Dijkstra (cont.)

```
void Dijkstra(Grafo G, int r, int custo[], int pred[]) {
    Conjunto Q;    int v;    int d;
    for (v in G.V) {
        custo[v] = INT_MAX; // infinito
        pred[v] = -1;      // indefinido
    }
    custo[r] = 0;
    Q = G.V;
    while (!vazio(Q)) {
        u = vértice em Q com valor custo[u] mínimo;
        if (custo[u] == INT_MAX)
            break; // vértice não alcançável
        Remover(Q, u);
        for (v in G.Adj(u)) {
            d = custo[u] + G.peso(u, v);
            if (d < custo[v]) { // alternativa com custo menor
                custo[v] = d;  pred[v] = u;
            }
        }
    }
} /* Dijkstra */
```

Algoritmo de Dijkstra (cont.)

- ▶ Os resultados são devolvidos através dos argumentos *custo* e *pred*.
- ▶ O grafo $G = (V, E)$ pode ser orientado ou não.
- ▶ A função $G.Adj(u)$ devolve os vértices adjacentes a u .
- ▶ A função $G.peso(u, v)$ devolve o peso (custo, distância) da aresta (u, v) .
- ▶ Os pesos têm que ser não negativos.
- ▶ Se for utilizada uma fila de prioridades para achar o vértice em Q com valor $custo[u]$ mínimo, o número de operações é dado, no pior caso, por $O((|E| + |V|) \log |V|)$; se for utilizado um vetor, será $O(|V|^2)$.
- ▶ Se for utilizada uma fila de prioridades, além da função de remoção, será necessária a função *Sobe* pois os elementos que ainda estão na fila podem ter seu peso diminuído migrando, portanto, na direção da raiz.

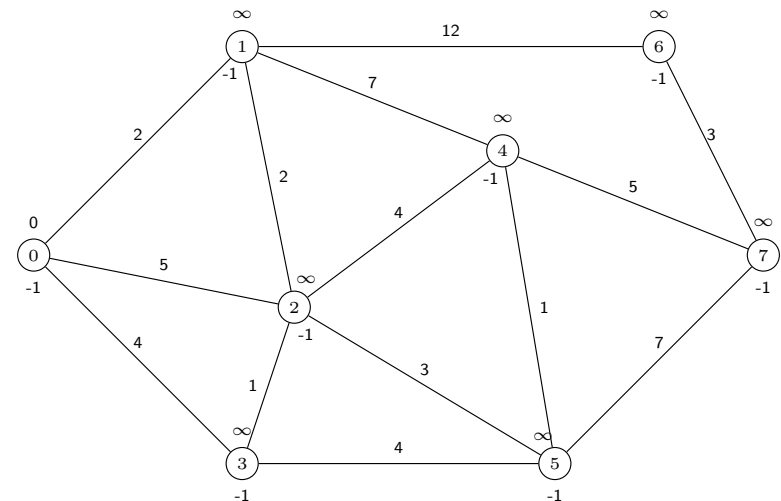
Exemplo do algoritmo de Dijkstra



O vértice inicial é $r=0$.

Exemplo do algoritmo de Dijkstra (cont.)

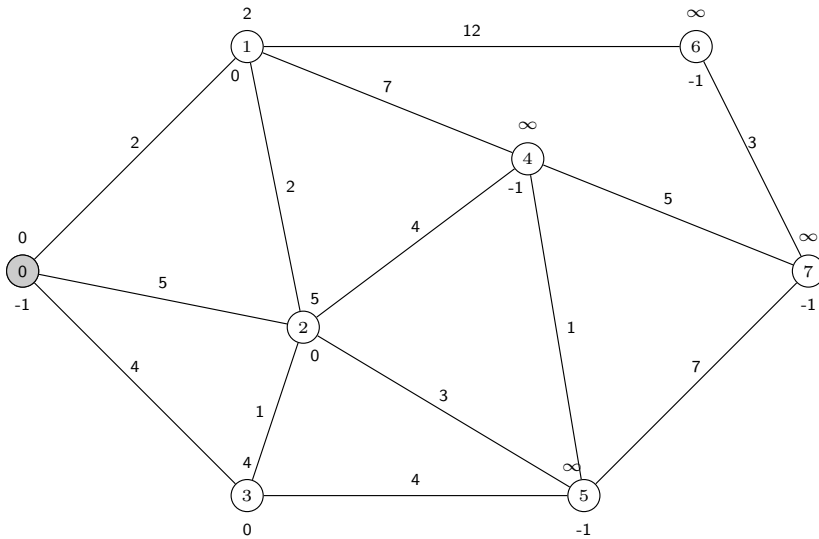
Após a inicialização ($Q = \{0, 1, 2, 3, 4, 5, 6, 7\}$):



Os números acima de cada vértice indicam a distância mínima; os números abaixo indicam o predecessor no caminho da distância mínima.

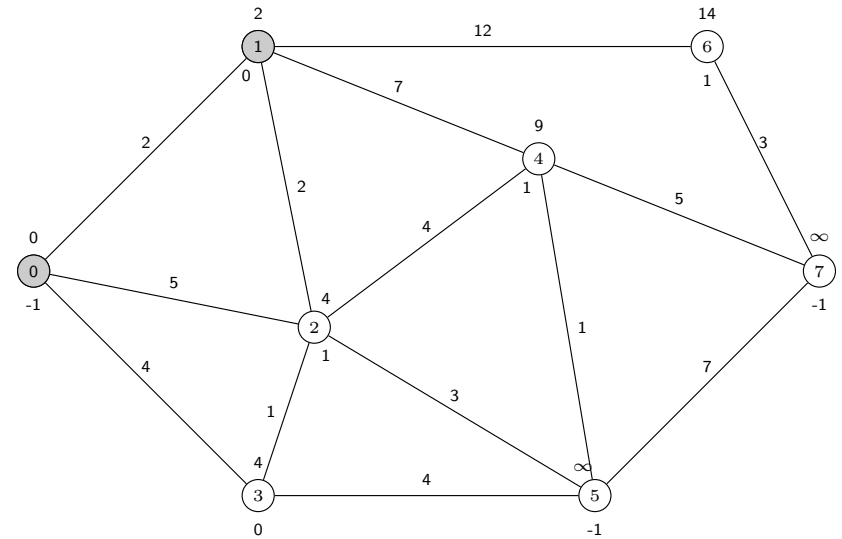
Exemplo do algoritmo de Dijkstra (cont.)

Após a primeira iteração ($u=0$, $Q=\{1, 2, 3, 4, 5, 6, 7\}$):



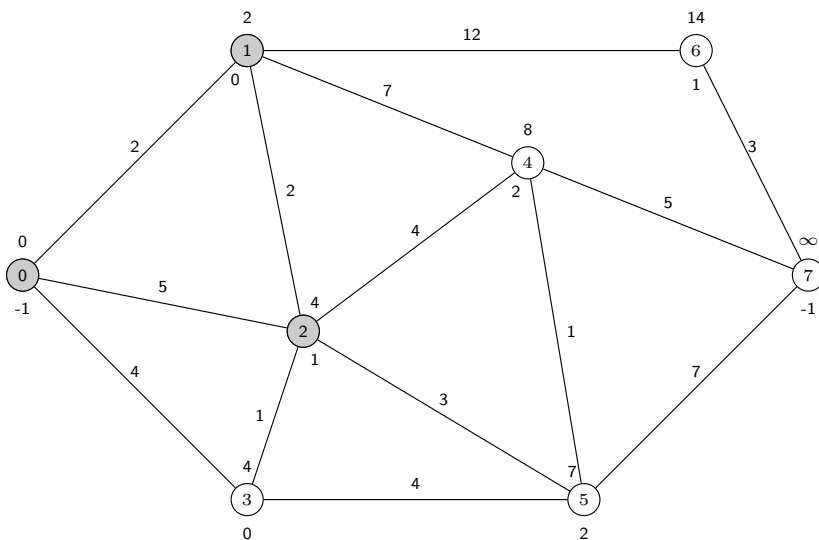
Exemplo do algoritmo de Dijkstra (cont.)

Após a segunda iteração ($u=1$, $Q=\{2, 3, 4, 5, 6, 7\}$):



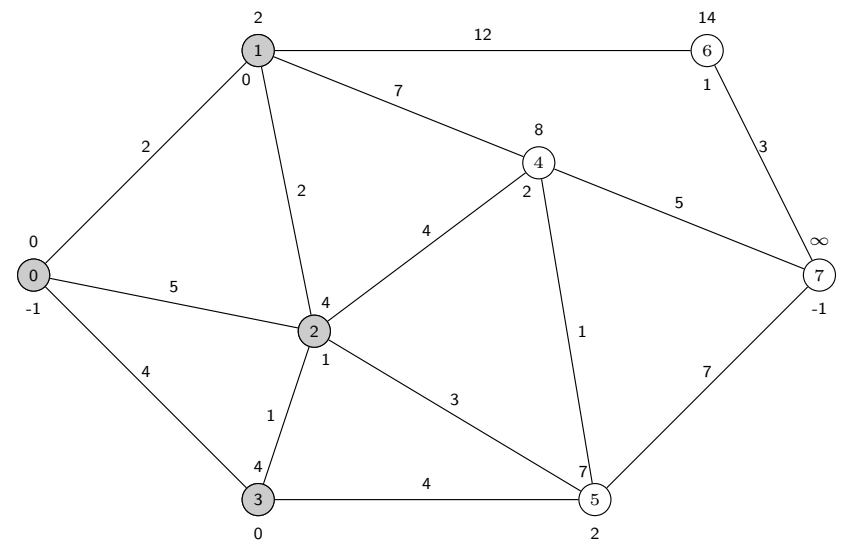
Exemplo do algoritmo de Dijkstra (cont.)

Após a terceira iteração ($u=2$, $Q=\{3, 4, 5, 6, 7\}$):



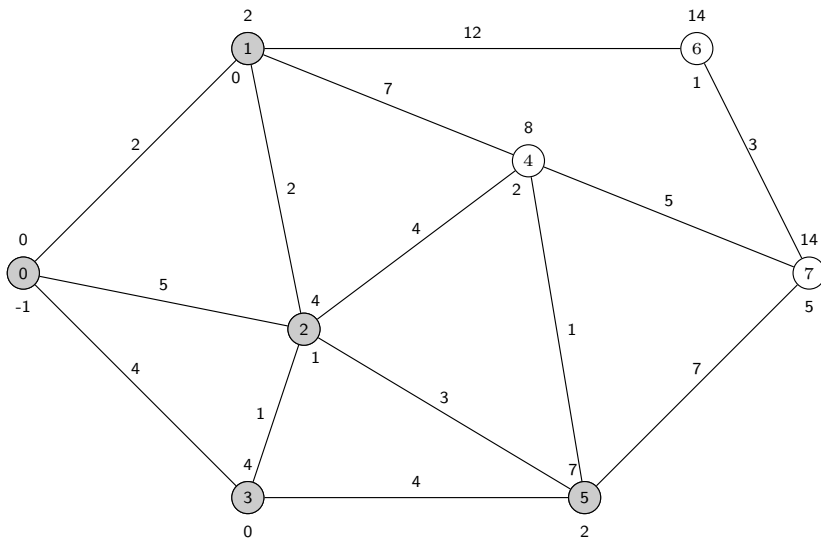
Exemplo do algoritmo de Dijkstra (cont.)

Após a quarta iteração ($u=3$, $Q=\{4, 5, 6, 7\}$):



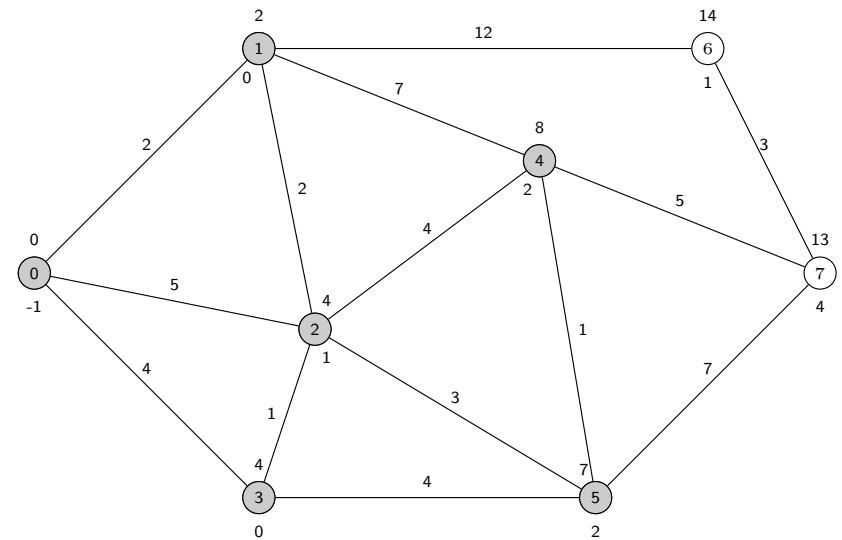
Exemplo do algoritmo de Dijkstra (cont.)

Após a quinta iteração ($u=5$, $Q=\{4, 6, 7\}$):



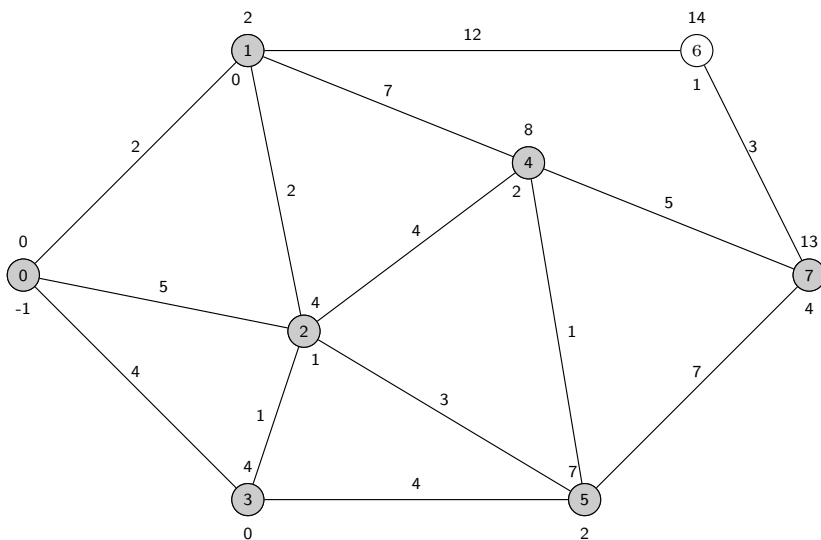
Exemplo do algoritmo de Dijkstra (cont.)

Após a sexta iteração ($u=4$, $Q=\{6, 7\}$):



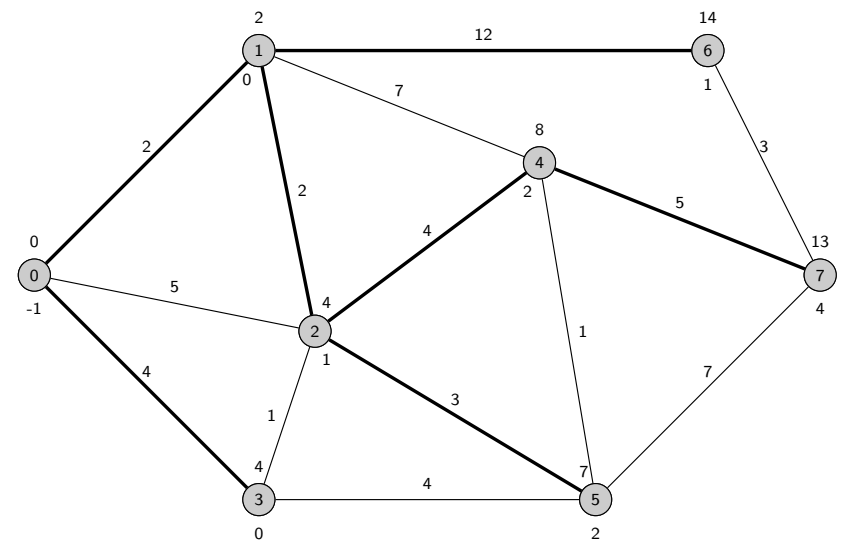
Exemplo do algoritmo de Dijkstra (cont.)

Após a sétima iteração ($u=7$, $Q=\{6\}$):



Exemplo do algoritmo de Dijkstra (cont.)

Após a oitava iteração ($u=6$, $Q=\emptyset$):



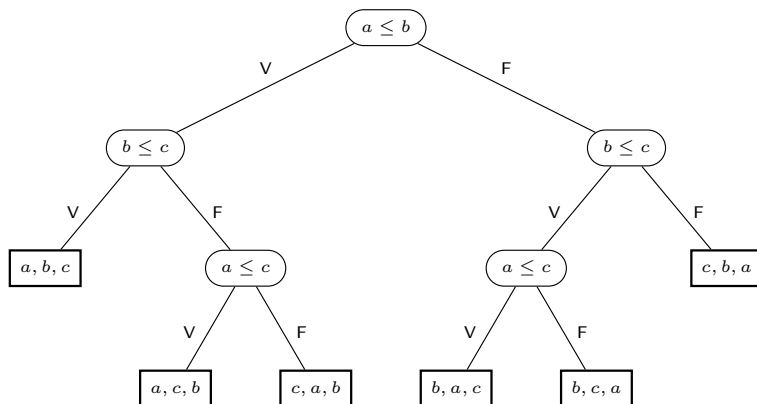
Algoritmos de ordenação

Generalidades

- ▶ Ordenação interna e externa
- ▶ Ordenação ótima por comparações: $O(n \log n)$
- ▶ Algoritmos por comparação:
 - ▶ transposição (*bubblesort*, *quicksort*)
 - ▶ inserção (inserção simples, *shellsort*)
 - ▶ seleção (seleção simples, *heapsort*)
 - ▶ intercalação (iterativo, recursivo)
- ▶ Outros algoritmos: distribuição (*radix sort*)
- ▶ Ordenação estável: mantém a ordem relativa dos registros de com chaves iguais.

Ordenação ótima por comparações

Árvore de decisão para ordenar três valores a , b e c :



- ▶ A árvore tem $3! = 6$ folhas (permutações de 3 elementos).
- ▶ A altura h da árvore é 4.
- ▶ Portanto, o número mínimo de comparações, no pior caso, é $3(h-1)$.

Ordenação ótima por comparações (cont.)

Caso geral de n elementos:

- ▶ A árvore de decisão deverá ter $n!$ folhas (número de permutações de n elementos).
- ▶ Uma árvore de altura h tem no máximo 2^{h-1} folhas.
- ▶ Deve-se ter, portanto:

$$\begin{aligned} 2^{h-1} &\geq n! \\ \Rightarrow h &\geq \lceil \log_2(n!) \rceil + 1 \end{aligned}$$

- ▶ Pode-se demonstrar que:

$$\log_2(n!) = O(n \log_2 n)$$

- ▶ Portanto, o número mínimo de comparações, no pior caso, é $O(n \log_2 n)$.
- ▶ Assim, não existe nenhum algoritmo de ordenação mais eficiente que utiliza apenas comparações de elementos.

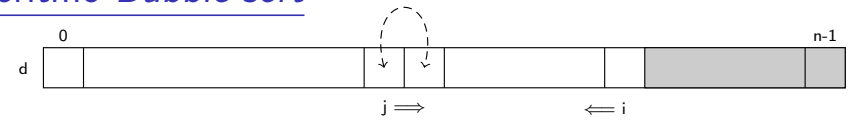
Algoritmos: declarações comuns

```
typedef struct Vetor {  
    int n;  
    int dados[1];  
} Vetor, *ApVetor;
```

```
void troca(int *x, int *y) {  
    int t = *x;  
    *x = *y;  
    *y = t;  
} /* troca */
```

- ▶ Os algoritmos de ordenação serão apresentados como funções em linguagem C que ordenam um vetor de dados que faz parte de uma estrutura denominada *Vetor*.
- ▶ Nesta declaração, n denota o tamanho verdadeiro do vetor *dados* que dependerá do parâmetro passado à função *malloc* quando o vetor for alocado.
- ▶ Vários algoritmos fazem trocas de valores entre elementos do vetor indicadas por chamadas da função *troca*.

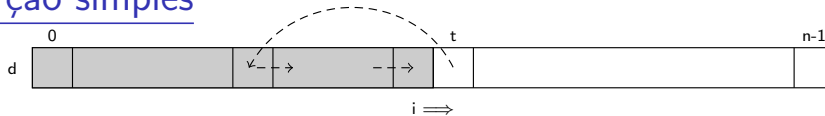
Algoritmo *Bubble sort*



- ▶ Os elementos entre $i+1$ e $n-1$ já estão ordenados.
- ▶ Os elementos $d[j]$ (abaixo de i) são “empurrados” se necessário; o maior deles acaba em seu lugar final.
- ▶ Verifica-se facilmente que o número de comparações executado por este algoritmo é sempre $(n^2 - n)/2$ (da ordem de $O(n^2)$).
- ▶ É um algoritmo estável.

```
void bubbleSort(ApVetor v) {  
    /*Exemplo de transposição */  
    int n = v->n, i, j;  
    int *d = (v->dados);  
    for (i=n-1; i>0; i--)  
        for (j=0; j<i; j++)  
            if (d[j]>d[j+1])  
                troca(&d[j], &d[j+1]);  
} /* bubbleSort */
```

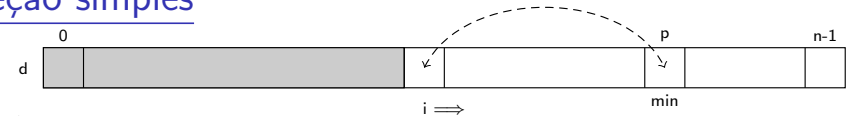
Inserção simples



- ▶ Os elementos entre 0 e i já estão ordenados.
- ▶ Os elementos maiores do que $d[i+1]$ são “empurrados” à direita e $d[i+1]$ inserido no seu lugar.
- ▶ No pior caso, $O(n^2)$ comparações; no melhor caso, $O(n)$.
- ▶ Um bom algoritmo se os dados já estão parcialmente ordenados.
- ▶ É um algoritmo estável.

```
void insercao(ApVetor v) {  
    int n = v->n, i, j, t;  
    int *d = (v->dados);  
    for (i=0; i<n-1; i++) {  
        t = d[i+1]; j = i;  
        while ((j>=0) && (t<d[j])) {  
            d[j+1] = d[j]; j--;  
        }  
        d[j+1] = t;  
    }  
} /* insercao */
```

Seleção simples



- ▶ Os elementos entre 0 e $i-1$ já estão ordenados e em suas posições finais.
- ▶ O elemento mínimo entre as posições i e $n-1$ ($d[p]$) troca de lugar com o elemento $d[i]$.
- ▶ O número de comparações é sempre $(n^2 - n)/2$ (da ordem de $O(n^2)$).
- ▶ Esta implementação não é estável.

```
void selecao(ApVetor v) {  
    int n = v->n, i, j, p;  
    int *d = (v->dados);  
    for (i=0; i<n-1; i++) {  
        p = i;  
        for (j=i+1; j<n; j++)  
            if (d[j]<d[p])  
                p = j;  
        troca(&d[i], &d[p]);  
    }  
} /* selecao */
```

Algoritmo *Heapsort*

- ▶ Em primeiro lugar, constrói um *heap* de tamanho n com elemento máximo na raiz (posição 0 do vetor).
- ▶ Troca o elemento na posição 0 com a posição $i = n-1$, e reconstrói o *heap* de tamanho $n-1$.
- ▶ Repete o passo anterior para $i = n-2, n-3, \dots, 1$.
- ▶ O número de comparações é da ordem $O(n \log n)$.
- ▶ O algoritmo não é estável.

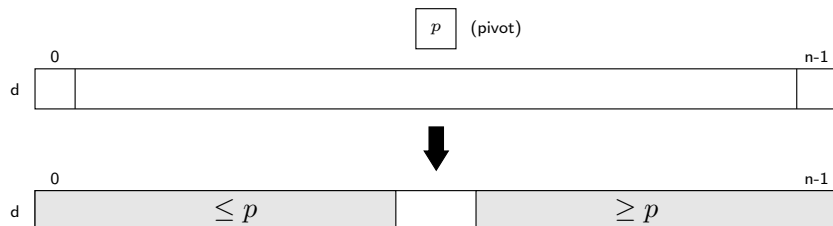
```
void HeapSort(ApVetor v) {
    int n = v->n, i, t;
    int *d = (v->dados);
    for (i=(n-2)/2; i>=0; i--)
        Desce(v, i, n);
    for (i=n-1; i>0; i--) {
        troca(&d[0], &d[i]);
        Desce(v, 0, i);
    }
} /* HeapSort */
```

Algoritmo *Heapsort* (cont.)

```
void Desce(ApVetor v, int m, int n) {
    int *d = (v->dados);
    int k = 2*m+1;
    int x = d[m];
    while (k<n) {
        if ((k<n-1) && (d[k]<d[k+1])) k = k+1;
        if (x<d[k]) {
            d[m] = d[k];
            m = k;
            k = 2*k+1;
        } else
            break;
    }
    d[m] = x;
}
```

Algoritmo *Quicksort*

- ▶ *Quicksort* foi idealizado por C. A. R. Hoare em 1962.
- ▶ É um algoritmo recursivo que ordena segmentos do vetor dado.
- ▶ Ideia geral:
 - ▶ particionar o vetor em duas partes, contendo elementos menores (ou iguais) e maiores (ou iguais) do que um valor escolhido (pivô):



- ▶ alguns valores, iguais ao pivô, podem ficar entre as duas partes a serem ordenadas, em seus lugares finais
- ▶ recursivamente ordenar as duas partes

Algoritmo *Quicksort* (cont.)

- ▶ A ordenação do vetor inteiro é realizada através da chamada de uma função auxiliar com argumentos que cobrem partes do vetor:

```
void quickSort(ApVetor v) {
    quickSortAux(v, 0, (v->n)-1);
} /* quickSort */
```

- ▶ A função auxiliar *quickSortAux* implementa de fato o algoritmo.

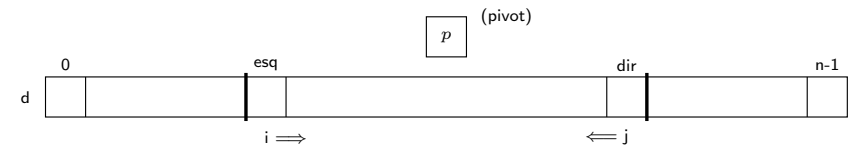
Algoritmo Quicksort (cont.)

```
void quickSortAux(ApVetor v, int esq, int dir) {
    /* supõe esq ≤ dir */
    int *d = (v->dados);
    int i = esq, j = dir;
    int pivot = d[(esq+dir)/2]; /* meio */
    do { /* particiona */
        while (d[i] < pivot) i++;
        while (d[j] > pivot) j--;
        if (i ≤ j) {
            troca(&d[i], &d[j]);
            i++; j--;
        }
    } while (i ≤ j);
    /* ordena */
    if (esq < j) quickSortAux(v, esq, j);
    if (dir > i) quickSortAux(v, i, dir); /* caudal */
} /* quickSortAux */
```

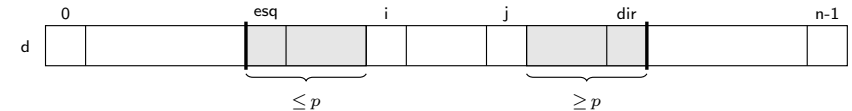
Algoritmo Quicksort (cont.)

```
do { /* particiona */
    while (d[i] < pivot) i++;
    while (d[j] > pivot) j--;
    if (i ≤ j) {
        troca(&d[i], &d[j]);
        i++; j--;
    }
} while (i ≤ j);
if (esq < j) quickSortAux(v, esq, j);
if (dir > i) quickSortAux(v, i, dir); /* caudal */
```

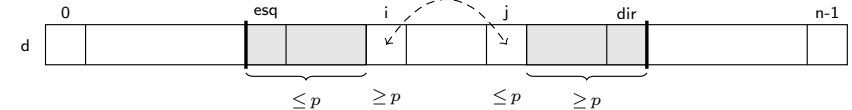
- ▶ Início do particionamento:



- ▶ Situação genérica antes de cada pivotação:



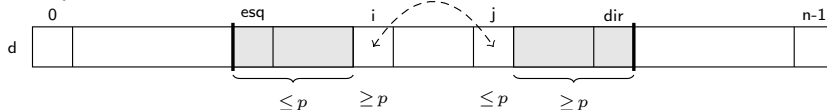
- ▶ Situação antes de uma troca:



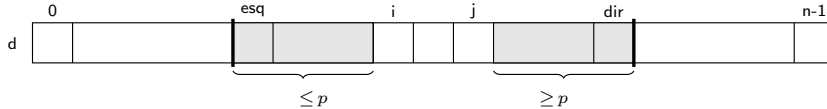
Algoritmo Quicksort (cont.)

```
do { /* particiona */
    while (d[i] < pivot) i++;
    while (d[j] > pivot) j--;
    if (i ≤ j) {
        troca(&d[i], &d[j]);
        i++; j--;
    }
} while (i ≤ j);
if (esq < j) quickSortAux(v, esq, j);
if (dir > i) quickSortAux(v, i, dir); /* caudal */
```

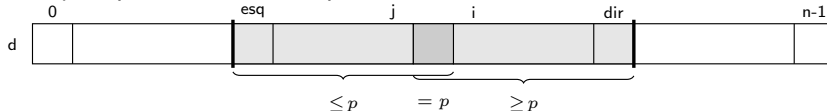
- ▶ Situação antes de uma troca:



- ▶ Situação após uma troca:



- ▶ Situação quando termina o particionamento:

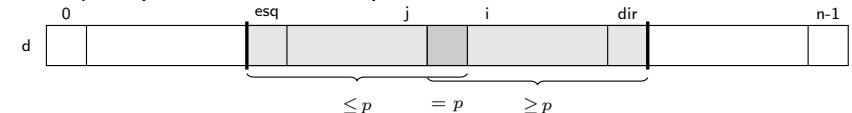


Pode haver dois, um ou nenhum elemento entre j e i . Se houver, eles são necessariamente iguais ao pivô e estão na sua posição final.

Algoritmo Quicksort (cont.)

```
do { /* particiona */
    while (d[i] < pivot) i++;
    while (d[j] > pivot) j--;
    if (i ≤ j) {
        troca(&d[i], &d[j]);
        i++; j--;
    }
} while (i ≤ j);
if (esq < j) quickSortAux(v, esq, j);
if (dir > i) quickSortAux(v, i, dir); /* caudal */
```

- ▶ Situação quando termina o particionamento:



- ▶ Situação após as chamadas recursivas – o segmento está ordenado:



Observações sobre o algoritmo *Quicksort*

- ▶ Escolha do pivô:
 - ▶ em princípio, o algoritmo funciona com qualquer valor do pivô, mas não pode ser menor ou maior do que todos os elementos
 - ▶ o ideal seria um pivô que particiona o segmento em duas partes de comprimentos iguais
 - ▶ algumas implementações utilizam a média de alguns poucos elementos
 - ▶ na implementação aqui exibida foi usado o valor do elemento do meio
- ▶ Eficiência:
 - ▶ no pior caso o algoritmo realiza da ordem de $O(n^2)$ operações
 - ▶ em média e no melhor caso são $O(n \log n)$ operações
 - ▶ na prática são quase sempre $O(n \log n)$ operações
 - ▶ é o algoritmo de ordenação interna mais utilizado e faz parte das bibliotecas de várias linguagens (por exemplo, *qsort* em C)
 - ▶ algumas implementações rearranjam o vetor de maneira pseudo-aleatória para aumentar a probabilidade de $O(n \log n)$
- ▶ Estabilidade:
 - ▶ sob a forma apresentada, o algoritmo não é estável
 - ▶ exercícios:
 - ▶ exibir um exemplo que demonstra a falta de estabilidade
 - ▶ modificar o algoritmo para que seja estável

Algoritmo *Quicksort* (cont.)

Exemplo de execução do *Quicksort* ($n=20$):
(' $<$ ' e ' $>$ ' delimitam o segmento corrente; ' $*$ ' marca o pivô)

Vetor original:

07 49 73 58 30 72 44 78 23 09 40 65 92 42 87 03 27 29 40 12

Pivot: 09

```
< * >
07 49 73 58 30 72 44 78 23 09 40 65 92 42 87 03 27 29 40 12
i j
07 03 73 58 30 72 44 78 23 09 40 65 92 42 87 49 27 29 40 12
i j
07 03 09 58 30 72 44 78 23 73 40 65 92 42 87 49 27 29 40 12
i j
07 03 09 58 30 72 44 78 23 73 40 65 92 42 87 49 27 29 40 12
j i
```

Pivot: 03

```
< * >
03 07 09 58 30 72 44 78 23 73 40 65 92 42 87 49 27 29 40 12
j i
```

Pivot: 07

```
< * >
03 07 09 58 30 72 44 78 23 73 40 65 92 42 87 49 27 29 40 12
j i
```

Algoritmo *Quicksort* (cont.)

Pivot: 65

```
< * >
03 07 09 58 30 12 44 78 23 73 40 65 92 42 87 49 27 29 40 72
i j
03 07 09 58 30 12 44 40 23 73 40 65 92 42 87 49 27 29 78 72
i j
03 07 09 58 30 12 44 40 23 29 40 65 92 42 87 49 27 73 78 72
i j
03 07 09 58 30 12 44 40 23 29 40 27 92 42 87 49 65 73 78 72
i j
03 07 09 58 30 12 44 40 23 29 40 27 49 42 87 92 65 73 78 72
i j
03 07 09 58 30 12 44 40 23 29 40 27 49 42 87 92 65 73 78 72
j i
```

Pivot: 23

```
< * >
03 07 09 23 30 12 44 40 58 29 40 27 49 42 87 92 65 73 78 72
i j
03 07 09 23 12 30 44 40 58 29 40 27 49 42 87 92 65 73 78 72
j i
```

Pivot: 23

```
< * >
03 07 09 12 23 30 44 40 58 29 40 27 49 42 87 92 65 73 78 72
j i
```

Algoritmo *Quicksort* (cont.)

Pivot: 29

```
< * >
03 07 09 12 23 27 44 40 58 29 40 30 49 42 87 92 65 73 78 72
i j
03 07 09 12 23 27 29 40 58 44 40 30 49 42 87 92 65 73 78 72
i j
03 07 09 12 23 27 29 40 58 44 40 30 49 42 87 92 65 73 78 72
j i
```

Pivot: 27

```
< * >
03 07 09 12 23 27 29 40 58 44 40 30 49 42 87 92 65 73 78 72
j i
```

Pivot: 40

```
< * >
03 07 09 12 23 27 29 30 58 44 40 49 42 87 92 65 73 78 72
i j
03 07 09 12 23 27 29 30 40 44 58 40 49 42 87 92 65 73 78 72
i=j
03 07 09 12 23 27 29 30 40 44 58 40 49 42 87 92 65 73 78 72
j i
```

Pivot: 30

```
< * >
03 07 09 12 23 27 29 30 40 44 58 40 49 42 87 92 65 73 78 72
j i
```

Algoritmo *Quicksort* (cont.)

Pivot: 40

									<	*	>								
03	07	09	12	23	27	29	30	40	i=j	40	44	49	42	87	92	65	73	78	72
03	07	09	12	23	27	29	30	40	j	40	44	49	42	87	92	65	73	78	72
										i									

Pivot: 44

										<	*		>						
03	07	09	12	23	27	29	30	40	40	42	44	49	58	87	92	65	73	78	72
											i	j							
03	07	09	12	23	27	29	30	40	40	42	44	49	58	87	92	65	73	78	72
										j		i							

Pivot: 49

03 07 09 12 23 27 29 30 40 40 42 44 ^{<*}
 j 49 58 87 92 65 73 72
 i

Pivot: 65

															<	*		>			
03	07	09	12	23	27	29	30	40	40	42	44	49	58	65	i=j	92	87	73	78	72	
03	07	09	12	23	27	29	30	40	40	42	44	49	58	65	j	i	92	87	73	78	72

Algoritmo *Quicksort* (cont.)

Pivot: 73

[illegible]

Pivot: 72

03	07	09	12	23	27	29	30	40	40	42	44	49	58	65 j	<*> 72 i	73 i	87	78	92
----	----	----	----	----	----	----	----	----	----	----	----	----	----	---------	----------------	---------	----	----	----

Pivot: 78

03	07	09	12	23	27	29	30	40	40	42	44	49	58	65	72	73	< 78 j	* 87 i	> 92
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	--------------	--------------	---------

Pivot: 87

																	<*	>	
03	07	09	12	23	27	29	30	40	40	42	44	49	58	65	72	73	78 j	87 i	92 i

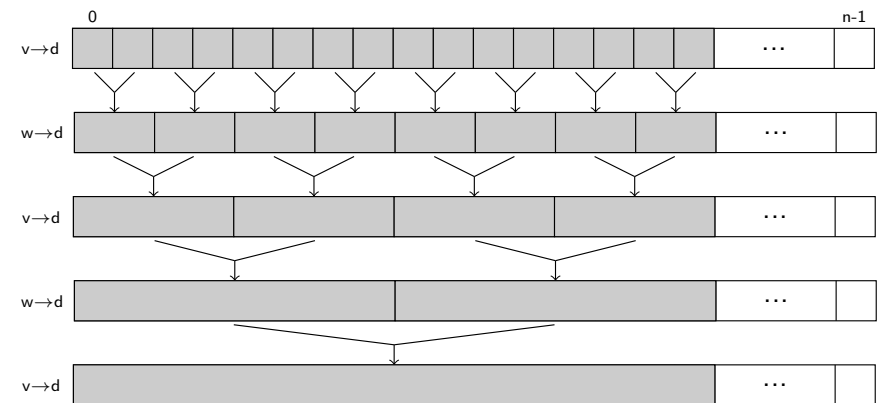
Resultado:

03 07 09 12 23 27 29 30 40 40 42 44 49 58 65 72 73 78 87 92

Intercalação iterativa (*Mergesort*)

- ▶ O algoritmo de intercalação iterativa foi provavelmente um dos primeiros algoritmos de ordenação interna propostos: John von Neumann (1945) que também provou que o algoritmo realiza $O(n \log n)$ operações.
- ▶ O algoritmo consiste em várias passagens pelo vetor, intercalando segmentos consecutivos de tamanhos 1, 2, 4, 8, ..., até completar o vetor.
- ▶ Utiliza um vetor auxiliar; os dois vetores são usados alternadamente para guardar os resultados da intercalação.
- ▶ Se necessário, há um passo adicional para copiar os resultados do vetor auxiliar para o original.
- ▶ O número de comparações deste algoritmo é da ordem de $O(n \log n)$ (ótimo).
- ▶ O algoritmo é estável.

Intercalação iterativa (cont.)



...

Quando n não é uma potência de 2, os últimos segmentos de cada estágio podem ficar mais curtos do que os outros.

Intercalação iterativa (cont.)

```
void
intercalalterativo(ApVetor v){
/* Ordena de 2 em 2, de 4 em 4,
... , por intercalação */
int n = v->n;
int td = 1;
int esq, dir, ld;
int tamanho =
sizeof(Vetor)+
sizeof(int)*(n-1);
Boolean par = false;
ApVetor w =
malloc(tamanho);
w->n = v->n;

while (td<n) {
esq = 0; par = !par;
do {
dir = esq+td; ld = dir+td;
if (dir>=n) { /* vazio */
dir = n; ld = n-1;
} else if (ld>n)
ld = n;
if (par)
interAux(v,w,esq,dir,ld);
else
interAux(w,v,esq,dir,ld);
esq = dir+td;
} while (esq<n);
td = 2*td;
}
if (par)
memcpy(v,w,tamanho);
free(w);
} /* intercalalterativo */
```

Intercalação iterativa (cont.)

```
void interAux(ApVetor v, ApVetor w,
int esq, int dir, int ld) {
/* Intercala v.dados[esq:dir-1] e
v.dados[dir:ld-1] em w.dados[esq:ld-1] */
int *dv = (v->dados), *dw = (w->dados);
int i = esq, j = dir, k = esq;
while ((i<dir)&&(j<ld)) {
if (dv[i]<=dv[j]) {
dw[k] = dv[i]; i++;
} else {
dw[k] = dv[j]; j++;
}
k++;
}
while (i<dir) { dw[k] = dv[i]; i++; k++; }
while (j<ld) { dw[k] = dv[j]; j++; k++; }
} /* interAux */
```

Intercalação iterativa (cont.)

Exemplo de execução ($n=20$):

Vetor original:

07 49 73 58 30 72 44 78 23 09 40 65 92 42 87 03 27 29 40 12

td=1:
| 07 49 | 58 73 | 30 72 | 44 78 | 09 23 | 40 65 | 42 92 | 03 87 | 27 29 | 12 40 |

td=2:
| 07 49 58 73 | 30 44 72 78 | 09 23 40 65 | 03 42 87 92 | 12 27 29 40 |

td=4:
| 07 30 44 49 58 72 73 78 | 03 09 23 40 42 65 87 92 | 12 27 29 40 |

td=8:
| 03 07 09 23 30 40 42 44 49 58 65 72 73 78 87 92 | 12 27 29 40 |

td=16:
| 03 07 09 12 23 27 29 30 40 40 42 44 49 58 65 72 73 78 87 92 |

Resultado:

03 07 09 12 23 27 29 30 40 40 42 44 49 58 65 72 73 78 87 92

Intercalação recursiva

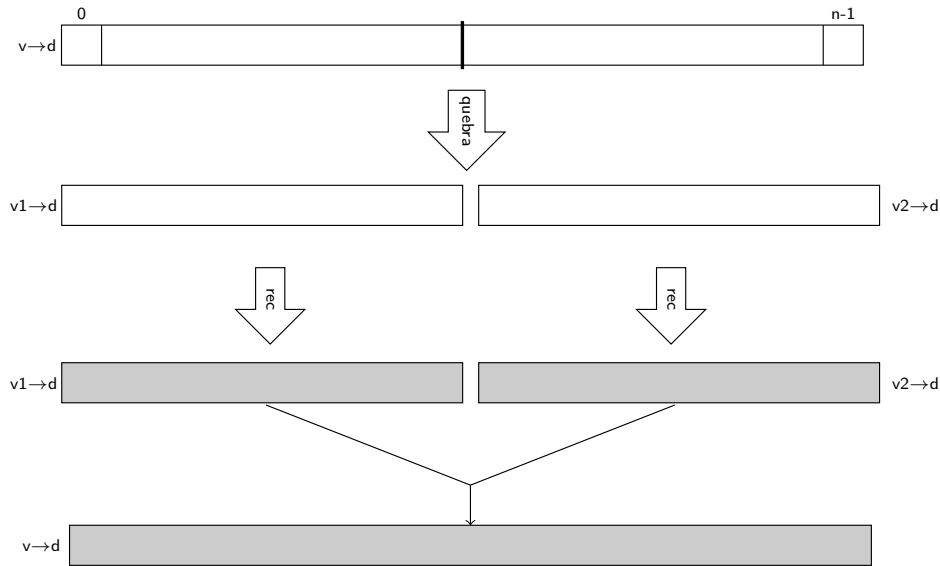
► Passos do algoritmo:

- quebrar o vetor v dado em dois vetores v_1 e v_2 , de tamanhos aproximadamente iguais
- se o tamanho de v_1 é maior que 1, ordená-lo recursivamente
- se o tamanho de v_2 é maior que 1, ordená-lo recursivamente
- intercalar os vetores v_1 e v_2 , deixando o resultado no vetor v original

► É fácil verificar que o número de comparações é da ordem de $O(n \log n)$ (ótimo).

► Se implementado corretamente, o algoritmo é estável.

Intercalação recursiva (cont.)



Intercalação recursiva (cont.)

```
void intercalaRecursivo(ApVetor v) {
    int n = v->n;
    if (n>1) {
        int *dv = v->dados;  ApVetor v1, v2;
        int i, nv1 = (int)(n/2), nv2 = n-nv1;
        v1 = (ApVetor) malloc(sizeof(Vetor)+sizeof(int)*(nv1-1));
        v2 = (ApVetor) malloc(sizeof(Vetor)+sizeof(int)*(nv2-1));
        v1->n = nv1;  v2->n = nv2;
        for (i=0; i<nv1; i++) (v1->dados)[i] = dv[i];
        for (i=0; i<nv2; i++) (v2->dados)[i] = dv[i+nv1];
        intercalaRecursivo(v1); intercalaRecursivo(v2);
        intercalaRecursivoAux(v1, v2, v);
        free(v1);  free(v2);
    }
} /* intercalaRecursivo */
```

Intercalação recursiva (cont.)

```
void intercalaRecursivoAux(ApVetor u, ApVetor v, ApVetor w) {
    /* Intercala os vetores u e v, deixando o resultado em w. */
    int i = 0, j = 0, k;
    int nu = u->n,  nv = v->n,  n = nu+nv;
    int *du = (u->dados),  *dv = (v->dados),
        *dw = (w->dados);
    for (k=0; k<n; k++) {
        if ((i<nu)&&(j<nv)) {
            if (du[i]<=dv[j]) { dw[k] = du[i]; i++; }
            else { dw[k] = dv[j]; j++; }
        } else {
            if (i<nu) { dw[k] = du[i]; i++; }
            else { dw[k] = dv[j]; j++; }
        }
    }
} /* intercalaRecursivoAux */
```

Comparação dos algoritmos de ordenação interna

Tempos em milissegundos, medidos numa máquina típica:

n	2^n	Bubble	Inserção	Seleção	Heap-sort	Quick-sort	Qsort	Interc. iter.	Interc. recur.	Interc. rec opt
6	64	0.02	0.00	0.01	0.00	0.00	0.01	0.00	0.02	0.01
7	128	0.11	0.03	0.04	0.02	0.01	0.01	0.01	0.04	0.01
8	256	0.40	0.09	0.17	0.03	0.03	0.04	0.03	0.13	0.04
9	512	1.65	0.37	0.59	0.07	0.06	0.08	0.06	0.18	0.07
10	1024	4.80	1.08	1.79	0.13	0.11	0.13	0.11	0.31	0.15
11	2048	19.27	4.28	7.05	0.27	0.23	0.30	0.24	0.64	0.27
12	4096	76.43	17.02	27.94	0.60	0.51	0.65	0.50	1.39	0.58
13	8192	307.52	67.88	111.24	1.30	1.09	1.41	1.08	2.86	1.26
14	16384	1228.45	273.24	446.97	2.82	2.29	3.01	2.27	6.06	2.64
15	32768	4895.07	1096.20	1782.97	6.10	4.90	6.42	4.85	12.42	5.58
16	65536	19591.07	4374.76	7128.58	13.22	10.21	13.67	10.29	26.44	11.73
17	131072	78355.01	17583.98	28522.31	28.51	21.36	29.19	21.95	54.57	24.94
18	262144	313686.39	70239.25	114134.72	61.37	44.95	61.85	45.95	114.27	52.14
19	524288	1258385.96	281594.75	457130.05	133.62	94.61	128.71	98.26	236.55	109.11

- ▶ A coluna *Quicksort* corresponde à versão que elimina a chamada recursiva caudal.
- ▶ A coluna *Qsort* corresponde à utilização da função *qsort* de biblioteca de C que aceita tipos mais gerais do que *int*.
- ▶ A última coluna corresponde a uma implementação otimizada do algoritmo de intercalação recursivo que evita alocações e liberações de espaço repetidas.
- ▶ *Quicksort* é o mais eficiente.

Observações sobre a comparação dos algoritmos

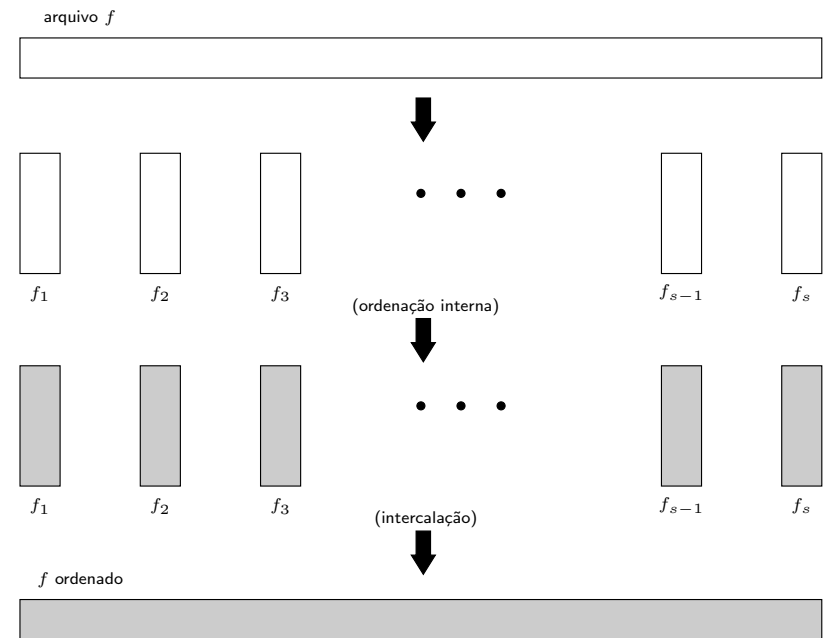
- ▶ As medidas de tempo apresentadas correspondem à ordenação de vetores de números inteiros.
- ▶ As diferenças entre os algoritmos podem tornar-se mais ou menos significativas dependendo de:
 - ▶ tamanho dos valores a serem ordenados (trocas e/ou cópias)
 - ▶ tempo de comparação entre os valores
 - ▶ uso de memória virtual
 - ▶ otimizações especiais para casos específicos; por exemplo, apontadores para registros grandes.

- ▶ Para aplicar a função `qsort` de C, foi usado:

```
int qcomp(const void *a, const void *b)
{
    return ( *(int*)a - *(int*)b );
}

qsort(v->dados, v->n, sizeof(int), qcomp);
```

Ordenação externa: intercalação balanceada múltipla



Ordenação externa: intercalação balanceada múltipla (cont.)

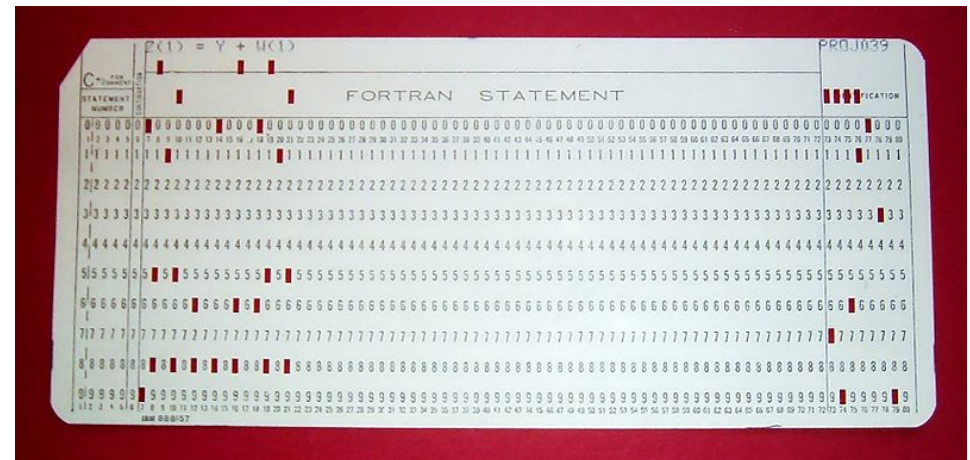
Passos do algoritmo, dado um arquivo f :

- ▶ $i = 1$
- ▶ Enquanto há dados em f :
 - ▶ Leia o máximo número de registros de f que cabem na memória num vetor v .
 - ▶ Ordene v usando um dos algoritmos de ordenação internos (por exemplo, *quicksort*).
 - ▶ Escreva o conteúdo de v em arquivo f_i .
 - ▶ $i = i + 1$
- ▶ Faça a intercalação múltipla dos arquivos $f_1, f_2, f_3, \dots, f_s$.

Obs.: Se o número s de arquivos f_i é razoavelmente grande (mais que 5 a 10), pode ser usada uma fila de prioridades de tamanho s .

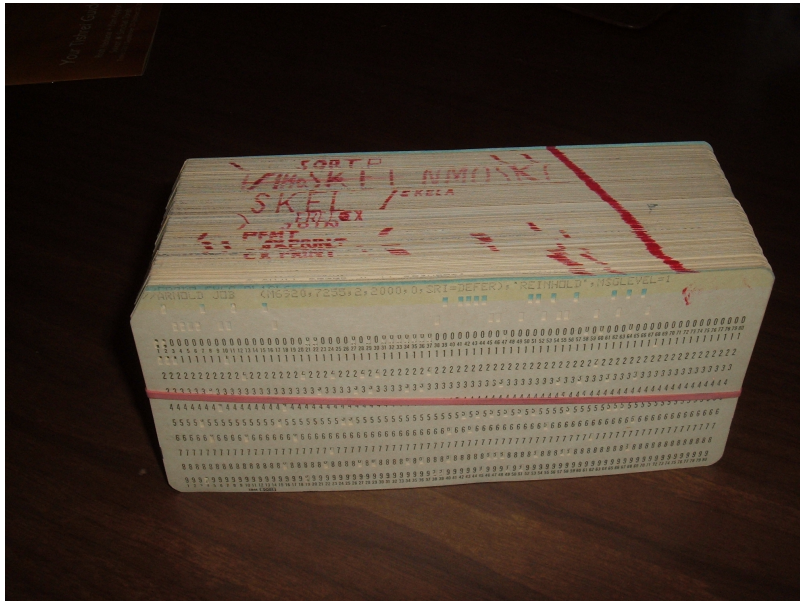
Ordenação digital ou por distribuição (*radix sort*)

O método precede a invenção de computadores digitais e foi usado para ordenar decks de cartões perfurados. Exemplo de um cartão:



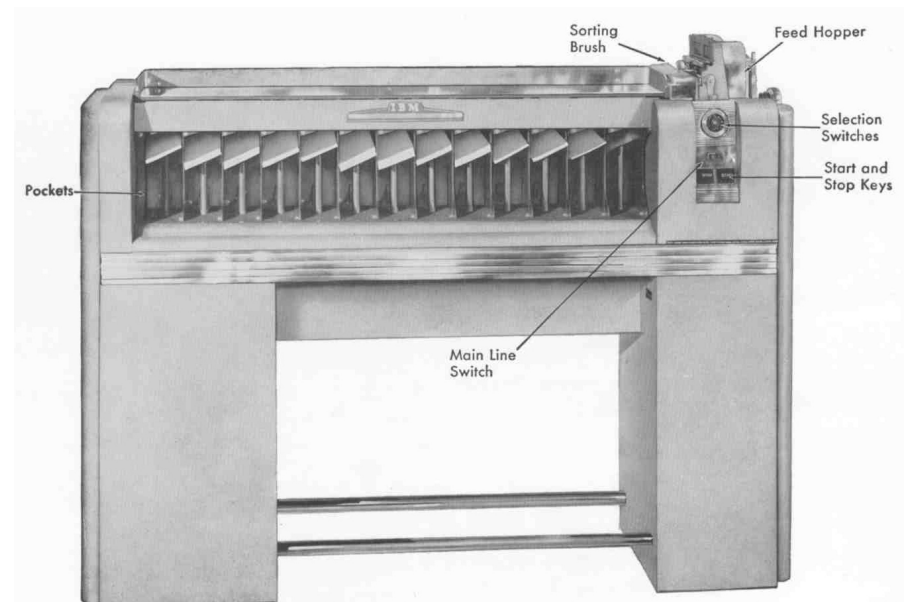
Ordenação digital ou por distribuição (cont.)

Exemplo de um deque de cartões:



Ordenação digital ou por distribuição (cont.)

Uma máquina classificadora de cartões perfurados (IBM-082):



Ordenação digital ou por distribuição (cont.)

- ▶ O algoritmo baseia-se em procedimentos que eram utilizados para ordenar cartões perfurados com máquinas classificadoras.
- ▶ A chave de cada registro de informação é tratada como uma cadeia de caracteres (ou um número numa base conveniente b) de comprimento m .
- ▶ Os registros são distribuídos em b sequências, conforme o último caractere da chave (mantendo a ordem relativa original).
- ▶ As b sequências são (conceitualmente) concatenadas em ordem crescente do caractere usado na distribuição.
- ▶ Os dois passos anteriores são repetidos para o penúltimo, o antepenúltimo, etc, caractere da chave.
- ▶ Após m distribuições, o vetor estará ordenado.
- ▶ O número de operações é no mínimo da ordem de $O(nm)$, dependendo da implementação.
- ▶ O algoritmo é estável.

Ordenação digital (cont.)

Exemplo ($n=20$, $m=2$):

Vetor original:

07 49 73 58 30 72 44 78 23 09 40 65 92 42 87 03 27 29 40 12

Distribuição pelo último dígito:

0:	1:	2:	3:	4:	5:	6:	7:	8:	9:					
30 40 40		72 92 42 12		73 23 03		44		65		07 87 27		58 78		49 09 29

Distribuição pelo penúltimo dígito:

0:	1:	2:	3:	4:	5:	6:	7:	8:	9:									
03 07 09		12		23 27 29		30		40 40 42 44 49		58		65		72 73 78		87		92

Resultado:

03 07 09 12 23 27 29 30 40 40 42 44 49 58 65 72 73 78 87 92

Gerenciamento de memória

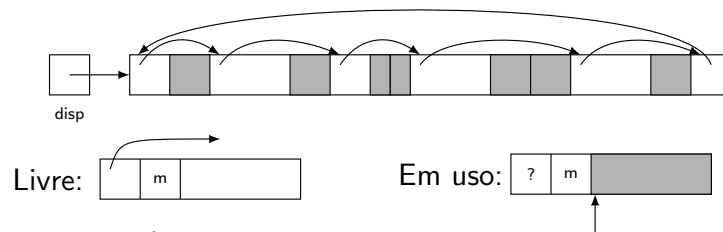
Gerenciamento de memória

Vários aspectos:

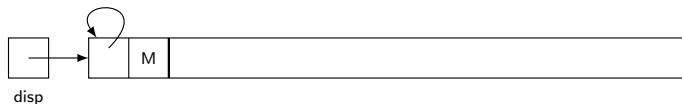
- ▶ alocação com e sem disciplina de pilha
- ▶ características da linguagem de programação
- ▶ registros de tamanho fixo ou variável
- ▶ gerenciamento explícito (*malloc* e *free*)
- ▶ gerenciamento implícito (coleta de lixo e contagem de referências)
- ▶ gerenciamento misto

Exemplo de gerenciamento explícito

Configuração genérica da memória dinâmica (*heap*):

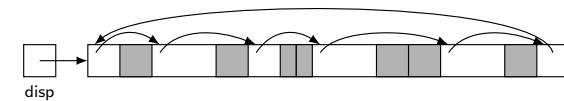


Configuração inicial:



- ▶ A variável *disp* (memória disponível) é global.
- ▶ A lista das áreas livres é ordenada pelo valor dos apontadores.
- ▶ *M* denota o tamanho da área livre inicial; *m* de uma área livre ou em uso.
- ▶ A função de alocação devolve o apontador para o primeiro *byte* livre da área.

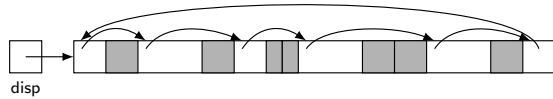
Gerenciamento explícito (cont.)



Uma versão muito simples de *malloc(n)* (*f* é o tamanho da parte fixa de cada área – apontador mais o campo de tamanho):

1. procure na lista *disp* o primeiro elemento (ou o elemento de tamanho mínimo) *p* com $\text{tamanho} \geq n + f$
2. remova *p* da lista *disp*
3. quebre a área apontada por *p* em duas partes: uma *p₁* de tamanho $n + f$ e outra *p₂* com o que sobrar (se houver sobra suficiente)
4. insira *p₂* na lista *disp* (se existir)
5. devolva *p₁ + f*

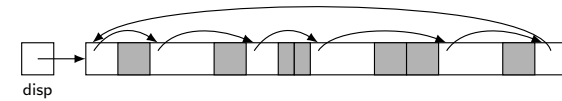
Gerenciamento explícito (cont.)



Uma versão muito simples de *free(p)*:

1. procure na lista *disp* o ponto de inserção para *p* (ordem crescente dos apontadores)
2. verifique se o predecessor e/ou sucessor de *p* neste ponto são adjacentes à área apontada por *p*
3. se for possível, junte a área liberada à predecessora e/ou à sucessora, modificando os campos de *tamanho*
4. atualize a lista

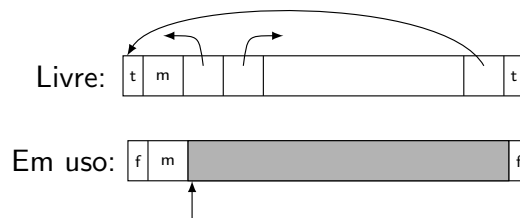
Gerenciamento explícito (cont.)



Problemas:

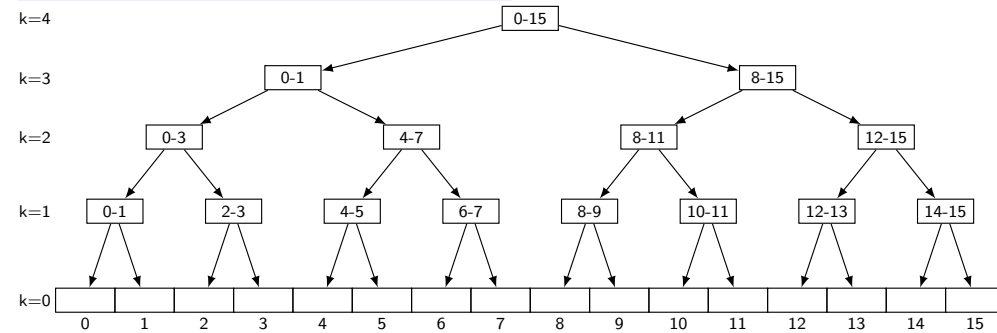
- O que fazer quando *malloc* não acha uma área de tamanho suficiente – requer outra área ao sistema operacional
- Fragmentação – após várias alocações e liberações, com tamanhos variáveis, haverá tendência a produzir muitas áreas livres pequenas
- Busca numa lista ligada pode ser ineficiente
- Algumas soluções:
 - blocos com *marcas de fronteira* (*boundary tags*)
 - sistema de *blocos conjugados* (*buddy system*)

Marcas de fronteira



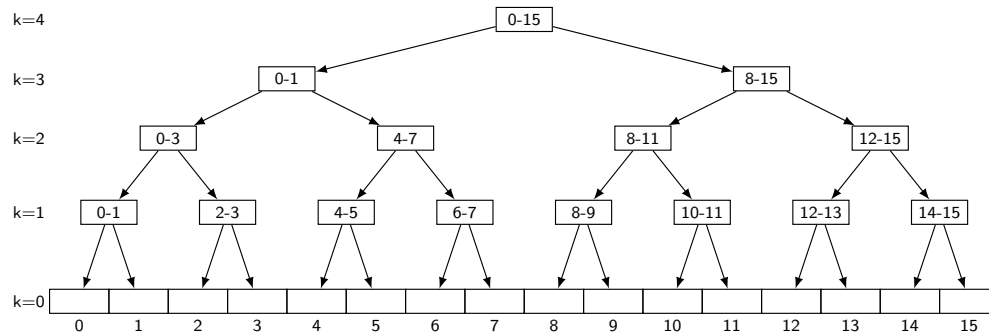
- As áreas livres constituem uma lista duplamente ligada.
- *m* denota o tamanho da área.
- *t* e *f* denotam os valores booleanos que indicam se a área é livre.
- A função de alocação devolve o apontador para o primeiro *byte* livre da área.
- Não é necessário fazer uma busca na lista para encontrar as áreas vizinhas.
- Exercício: esboçar a implementação das funções *malloc* e *free*.

Sistema de blocos conjugados



- Uma árvore binária imaginária em que cada nó representa um bloco de alocação.
- Cada folha da árvore representa um bloco mínimo de alocação.
- Cada nível *k* da árvore (a partir das folhas) representa a alocação de uma área constituída de 2^k blocos mínimos.
- Áreas conjugadas (irmãs) facilmente reconhecidos pelos índices sob forma binária.
- O exemplo exhibe a árvore para uma memória de $2^4 = 16$ blocos.

Sistema de blocos conjugados (cont.)



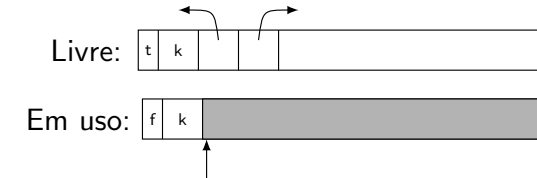
Dado o número do bloco inicial de uma área (em binário) de tamanho 2^k , o número da área conjugada é determinado complementando o k -ésimo *bit* (a partir da direita); exemplo de bloco 12 para $k=2$:

$$12_{10} = 1100_2 \implies 1000_2 = 8_{10}$$

Portanto, a área conjugada de quatro blocos tem início no bloco 8.

Sistema de blocos conjugados (cont.)

- Formato das áreas:



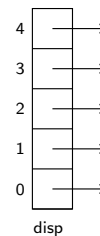
- t e f denotam os valores booleanos que indicam se a área é livre.
- k indica que o tamanho da área é 2^k .
- Para cada valor de k , existe uma lista duplamente ligada $disp[k]$ de blocos livres de tamanho 2^k .
- A função de alocação devolve o apontador para o primeiro *byte* livre da área.

Sistema de blocos conjugados (cont.)

Esboço da função $malloc(n)$ (f é o tamanho da parte fixa de cada área – marca de uso, tamanho):

1. procure um k mínimo tal que $2^k \geq n + f$, e a lista de blocos para k não está vazia; remova desta lista uma área p
2. se $2^{k-1} \geq n + f$, quebre a área p em duas (conjugadas), acerte os tamanhos e insira uma delas na lista de áreas para tamanho $k-1$
3. repita o passo anterior para $k-2, k-3, \dots$, enquanto possível
4. devolva o apontador $p + f$

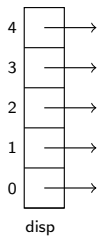
Note-se que o desperdício de memória de uma área pode chegar a quase 50%.



Sistema de blocos conjugados (cont.)

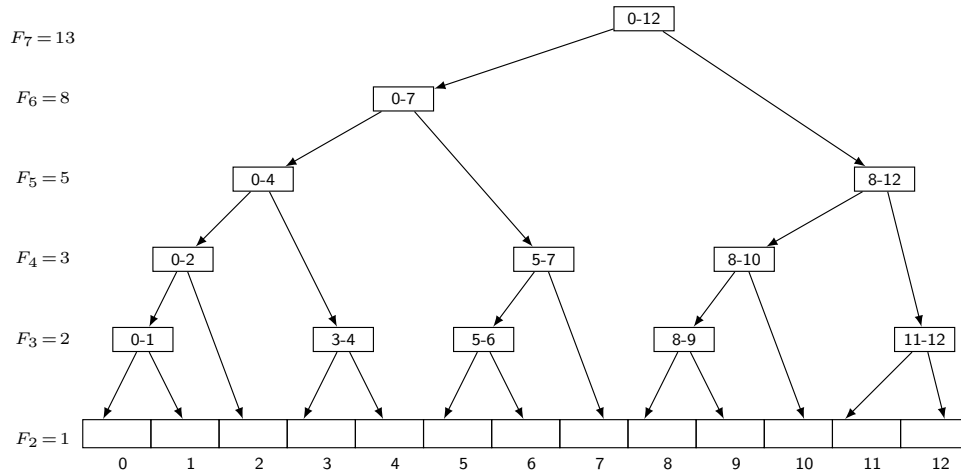
Esboço da função $free(p)$:

1. seja k o expoente correspondente ao tamanho de p
2. calcule o endereço da área q conjugada de p
3. se a área q está livre:
 - remova q da lista $disp[k]$
 - junte as áreas p e q para formar uma nova área livre p
 - faça $k = k+1$
 - volte ao passo inicial
4. se a área q não está livre (ou não existe – p já é a memória inteira), insira p na lista $disp[k]$



Sistema de blocos conjugados (cont.)

Uma outra alternativa é utilizar os números de Fibonacci:



- ▶ Esta solução diminui o desperdício de memória, mas torna mais complicados os algoritmos.
- ▶ Exercício: esboçar as funções *malloc* e *free* para esta alternativa.

Gerenciamento implícito

Coleta de lixo (*garbage collection*)

- ▶ Características:
 - ▶ operação de alocação implícita em algumas operações
 - ▶ não existe operação de liberação explícita ou é opcional
 - ▶ liberação de memória realizada, em geral, quando não há mais espaço para alocar
 - ▶ exemplos: Java, LISP, Prolog, Perl, Python, Modula-3, ...
 - ▶ contra-exemplos: C, Pascal, C++, ...
- ▶ Fases:
 - ▶ marcação
 - ▶ coleta ou compactação
 - ▶ caso haja compactação: cálculo de destino; atualização dos apontadores; cópia dos blocos

Marcação e coleta

Hipóteses

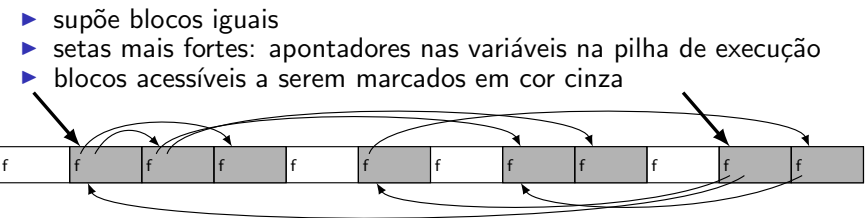
- ▶ Localização conhecida das variáveis apontadoras na pilha de execução.
- ▶ Localização conhecida de apontadores em cada bloco.
- ▶ Blocos de tamanho fixo ou com campos de comprimento.
- ▶ Marcas de utilização em cada bloco, inicialmente falsas.

Algoritmo:

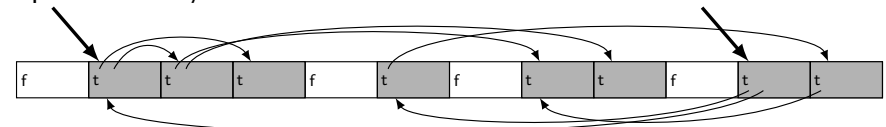
- ▶ Percurso análogo à pré-ordem e marcação a partir das variáveis na pilha.
- ▶ Percurso linear da memória para coleta de blocos livres e restauração das marcas.

Marcação e coleta (cont.)

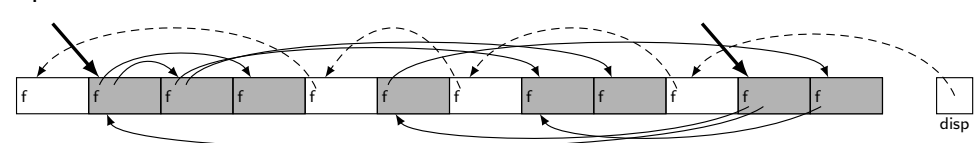
Exemplo de situação inicial:



Após a marcação:



Após a coleta:



Marcação e coleta (cont.)

- ▶ Hipóteses simplificadoras:
 - ▶ tamanho fixo de registros
 - ▶ localização conhecida dos apontadores (um vetor)

- ▶ Declarações:

```
typedef struct Bloco *ApBloco;
typedef struct Bloco {
    Boolean marca;
    ApBloco destino; /* se houver compactação */
    . . .
    int numAps;
    ApBloco aponts[ NUM_MAX_APONTS ];
} Bloco;

ApBloco disp; /* inicialmente todos os blocos */
Bloco memoria[ TAM_MEM_DIN ];
```

Marcação e coleta (cont.)

Função de marcação:

```
void Marcar(ApBloco p) {
    int i;
    if (p!=NULL) {
        if (!p->marca) {
            p->marca = true;
            for (i=0; i<p->numAps; i++)
                Marcar(p->aponts[i]);
        }
    }
} /* Marcar */
```

A função *Marcar* deve ser chamada para cada variável apontadora na pilha de execução.

Marcação e coleta (cont.)

Função de coleta:

```
void Coletar() {
    int i;
    disp = NULL;
    for (i=0; i<TAM_MEM_DIN; i++)
        if (memoria[i].marca) /* em uso */
            memoria[i].marca = false;
        else { /* insere na lista disponível */
            memoria[i].aponts[0] = disp;
            disp = &(memoria[i]);
        }
} /* Coletar */
```

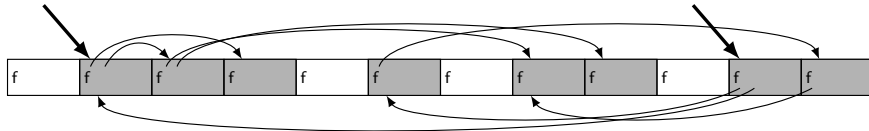
Marcação e compactação

- ▶ Hipóteses
 - ▶ Localização conhecida das variáveis apontadoras na pilha de execução.
 - ▶ Localização conhecida de apontadores em cada bloco.
 - ▶ Blocos de tamanho fixo ou com campos de comprimento.
 - ▶ Marcas de utilização em cada bloco, inicialmente falsas.
- ▶ Algoritmo:
 - ▶ Percurso análogo à pré-ordem e marcação a partir das variáveis na pilha.
 - ▶ Cálculo dos novos endereços dos blocos.
 - ▶ Atualização dos campos apontadores.
 - ▶ Compactação (cópia).

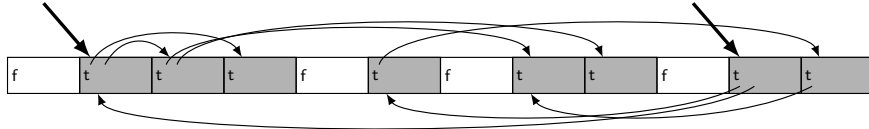
Marcação e compactação (cont.)

Exemplo de situação inicial:

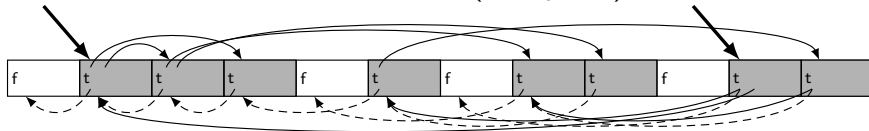
- ▶ supõe blocos iguais
- ▶ setas mais fortes: apontadores nas variáveis na pilha de execução
- ▶ blocos acessíveis a serem marcados em cor cinza



Após a marcação:

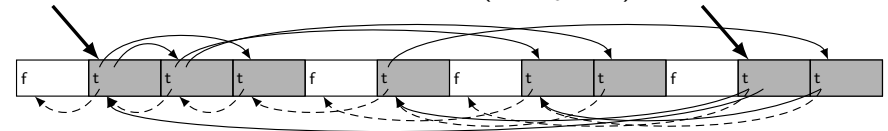


Após o cálculo dos novos endereços (tracejados):

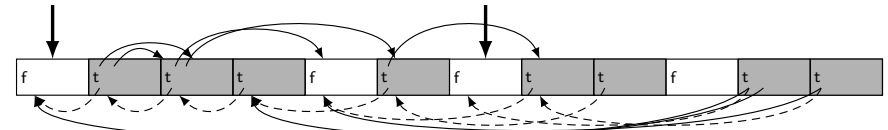


Marcação e compactação (cont.)

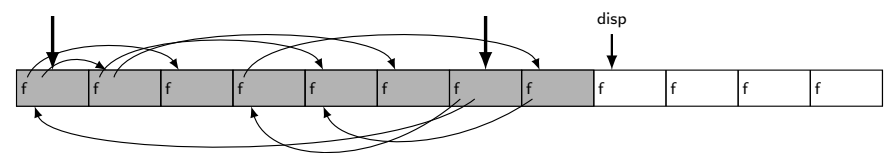
Após o cálculo dos novos endereços (tracejados):



Após a atualização dos apontadores, inclusive os da pilha:



Após a compactação:



A variável *disp* aponta para o início da área contígua liberada pela operação de compactação (não é necessário usar uma lista ligada).

Marcação e compactação (cont.)

- ▶ São adotadas as mesmas hipóteses do caso de marcação e coleta.
- ▶ A função *marcar* é a mesma.
- ▶ Função de cálculo dos novos endereços:

```
void CalcularDestino() {
    int i, j = 0;
    for (i=0; i<TAM_MEM_DIN; i++)
        if (memoria[i].marca) {
            memoria[i].destino = &(memoria[j]);
            j++;
        }
    disp = &(memoria[j]); /* primeiro bloco livre */
} /* CalcularDestino */
```

Marcação e compactação (cont.)

Função de atualização dos apontadores:

```
void Atualiza() {
    int i, j = 0;
    for (i=0; i<TAM_MEM_DIN; i++)
        if (memoria[i].marca)
            for (j=0; j<memoria[i].numAps; j++) {
                memoria[i].aponts[j] =
                    (memoria[i].aponts[j])->destino;
            }
} /* Atualiza */
```

Devem ser atualizadas também todas as variáveis apontadoras na pilha de execução.

Marcação e compactação (cont.)

Função de compactação:

```
void Move() {
    int i;
    for (i=0; i<TAM_MEM_DIN; i++)
        if (memoria[i].marca) {
            memoria[i].marca = false;
            *(memoria[i].destino) = memoria[i];
        }
} /* Move */
```

Adaptação para blocos de tamanho variável: introduzir o campo *tamanho* em cada bloco e adaptar as funções.

Contagem de referências

- ▶ As técnicas de coleta de lixo (marcação e coleta ou compactação) encontram e liberam toda a memória disponível de uma vez.
- ▶ O processo pode ser bastante demorado, com tempo de execução proporcional ao tamanho total da memória dinâmica.
- ▶ A execução da coleta de lixo interrompe o processo em curso; esta interrupção pode demorar mais do que seria aceitável em algumas aplicações.
- ▶ Dependendo da complexidade das estruturas de dados criadas pelo programa, a fase de marcação pode exigir memória adicional apreciável para manter a pilha de execução.
- ▶ A técnica de *contagem de referências* tem a característica de, em geral, distribuir o tempo de gerenciamento de memória ao longo da execução normal do programa.

Contagem de referências (cont.)

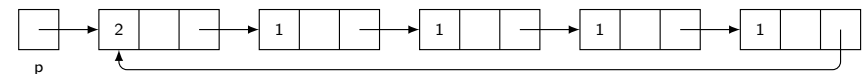
- ▶ Cada bloco alocado possui um campo inteiro *refs* contendo o número de variáveis (normais ou dinâmicas) que apontam para o bloco.
- ▶ Durante a alocação de um bloco, o seu campo *refs* recebe o valor inicial 1.
- ▶ Toda variável ou campo apontador, antes de ser atribuído, recebe o valor NULL.
- ▶ Todo comando de atribuição que envolve apontadores é implementado pela função *AtribuiApont(ApBloco *p, ApBloco q)*:

```
typedef struct Bloco *ApBloco;
typedef struct Bloco {
    int refs;
    . . .
    int numAps;
    ApBloco
    aponts[NUM_MAX_APONTS];
} Bloco;

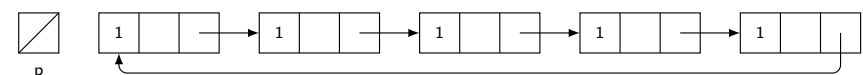
void AtribuiApont(ApBloco *p, ApBloco q) {
    if (q!=NULL) (q->refs)++;
    if ((*p)!=NULL) {
        ((*p)->refs)--;
        if (((*p)->refs)==0)
            DesalocaRefs(*p);
    }
    *p = q;
}
```

Contagem de referências (cont.)

- ▶ A função *DesalocaRefs(p)*:
 - ▶ desaloca o bloco apontado por *p*
 - ▶ decrementa os contadores dos blocos referidos em *p*
 - ▶ caso algum destes contadores torne-se nulo, a função é chamada recursivamente
- ▶ Problemas:
 - ▶ dependendo das estruturas de dados, o tempo de execução de comando de atribuição entre apontadores é imprevisível devido à recursividade da função *DesalocaRefs*
 - ▶ o método, como exposto, não funciona para estruturas com circularidades; exemplo:



após a atribuição *p = NULL*:



Os nós da lista não seriam liberados. Alguns sistemas utilizam um esquema misto: contagem de referências e coleta de lixo.