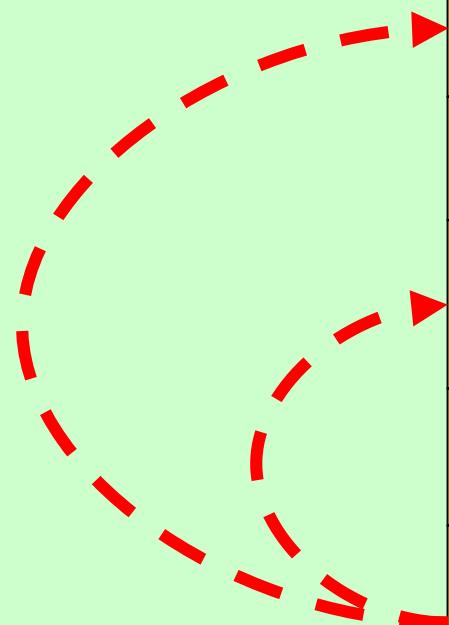


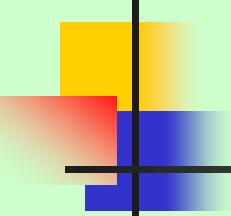
# Estrutura de Dados & Algoritmos

Prof. Edilberto Strauss, PhD

Março / 2018

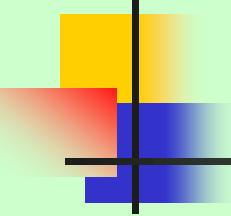
# PROBLEMA: solução





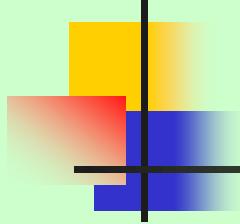
# Introdução

- Ciência da Computação=> estudo de algoritmos focando-se 4 principais áreas:
  - Hardware: garantir a execução eficiente dos algoritmos;
  - Linguagens: sintaxes e semânticas para linguagem de máquina;
  - Fundamentos dos algoritmos: modelagem computacional eficiente (numero de operações compatível com a complexidade natural do problema ?)
  - Análise do algoritmos: otimização do algoritmo (otimização do tempo de processamento e do espaço de memória requerido)



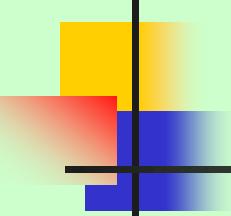
# Como devemos resolver alguns problemas ?????? Será que as soluções são realmente eficientes ???

1. Como poderemos achar uma celebridade em uma multidão, onde todos conhecem a celebridade (que não conhece ninguém da multidão) ???
2. Como podemos saber qual o candidato vence por maioria absoluta ( $50\% + 1$  voto) uma eleição????
3. Como faz nosso celular para achar o nome de um amigo armazenado na agenda ?????
4. Como podemos enviar uma mensagem de forma que ninguém saiba quem enviou, e qual o conteúdo da mensagem. Claro que somente o receptor final, terá acesso ao conteúdo da mensagem e quem a enviou ????
5. Como devemos fazer a otimização da distribuição de jornais em um bairro com várias bancas e somente 3 pontos de distribuição localizados de forma aleatória no bairro ???
6. De uma forma emergencial, como aumentar a produção em uma fábrica com mais de uma linha de montagem, sabendo que em cada linha de montagem existem os mesmos pontos de fabricação, os quais algumas linhas de montagem possuem máquinas mais modernas (e rápidas) e máquinas mais antigas (não tão rápidas, porém ainda eficientes) ????
7. Como devemos otimizar a execução de tarefas, que de forma dinâmica, estão sendo suas prioridades de execução alteradas ????
8. Como selecionar seus 5 melhores funcionários (baseado em um valor de produtividade individual) para premia-los com um final de semana em um resort ???



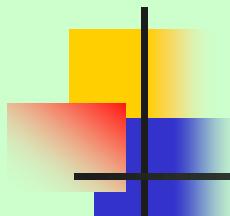
# Introdução

- Algoritmos: um conjunto finito de instruções destinado a obter a solução de um problema;
- Características:
  - INPUT: informações requeridas ( $0 \rightarrow n$ );
  - OUTPUT: resultado final ( $1 \rightarrow n$ );
  - Definição: toda a instrução deverá ser clara e não ambígua;
  - Eficiência: cada instrução deverá ser simples e por si só eficiente(execução individual da instrução);
  - Finitude: todo algoritmo terminará apos executar um numero finito K de operações ( $K \ll \infty$ ) – não existênciade loops;



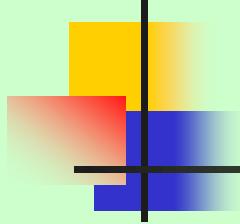
# Algoritmos

- O algoritmo tem a função de manipular e transformar dados (informações);
- Logo, estudar algoritmos também engloba estudos para o tratamento de dados, englobando:
  - Hardware (armazenagem, acessibilidade e segurança do dado);
  - Linguagens de manipulação (ex: SQL);
  - Refinamento de informações a partir de dados brutos (data mining);
  - Estruturas de representação da informação(ex: grafos para modelos 3D de objetos, etc);



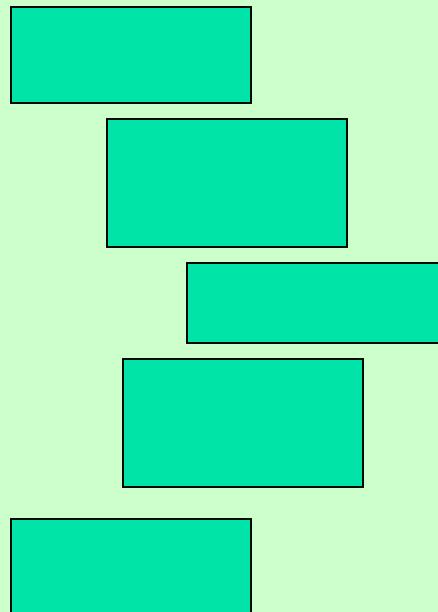
# Programa x Algoritmo

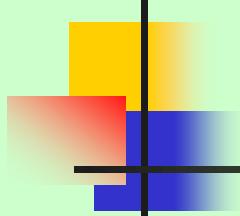
- Programa diferente de algoritmo:
  - em nível de abstração;
  - Algoritmo garante a finitude e o programa NÃO (ocorrência de loops não finitos);
- Formas de escrever algoritmos:
  - Linguagem natural;
  - Modelo gráfica (fluxogramas);



# Algoritmos

- Modelo de escrita para algoritmos  
(próximo a programação ...)
  - Usar estrutura de blocos endentados:





# Algoritmos

- ....continuação
  - Usar declarações básicas:

If -----

xxxxxx

else

xxxxxx

endIf

While -----

xxxxxx

endWhile;

For -----

xxxxxx

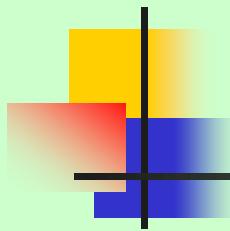
endFor;

Procedure()

begin

xxxxxx

endProcedure;



# Algoritmos

- ....continuação

- Usar variáveis simples ou complexas:

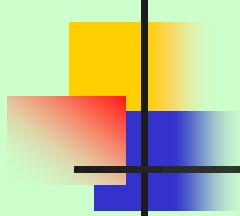
- Int, float, char, logic, vetor[n], matrix2D[x][y], matrix\_3D[x][y][z], matrix\_nD [x][y][z]...[t];
    - Set define structure{

```
Int matricula;  
Char nome[50];
```

```
} ESTUDANTE;
```

- Usar referência de registros (PONTEIROS):

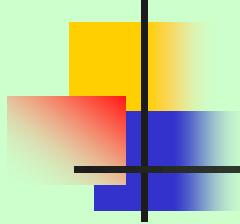
- Pt->informação (referencia ao endereço de memória descrito por pt)



# Algoritmos

- Ex1: Faça a inversão de uma seqüência de elementos armazenados em um vetor  $S[i]$  onde  $0 < i < (n-1)$ .
  - Input =  $S = [ A \ B \ C \ D \ E ]$
  - Output =  $S_{\text{invert}} = [ E \ D \ C \ B \ A ]$
  - ALGORITMO:

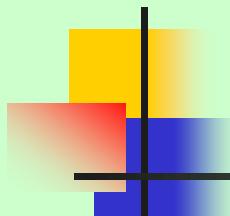
```
For i = 1 to maior_int(n/2)
    temp=S[i];
    S[i]=S[n-i+1];
    S[n-i+1]=temp;
endfor;
```



# Algoritmos

- Ex2: desenvolva um algoritmo para o cálculo do fatorial de um numero n.

- Algoritmo 1:  
Fat[0]=1;  
For i = 1 to n  
    fat[i]=i\*fat[i-1];  
endFor;
- Algoritmo 2:  
Fat = n;  
i = n-1;  
while i > 0  
    fat = fat \* i;  
    i = i-1  
endWhile;



# Recursividade

## ■ Recursividade:

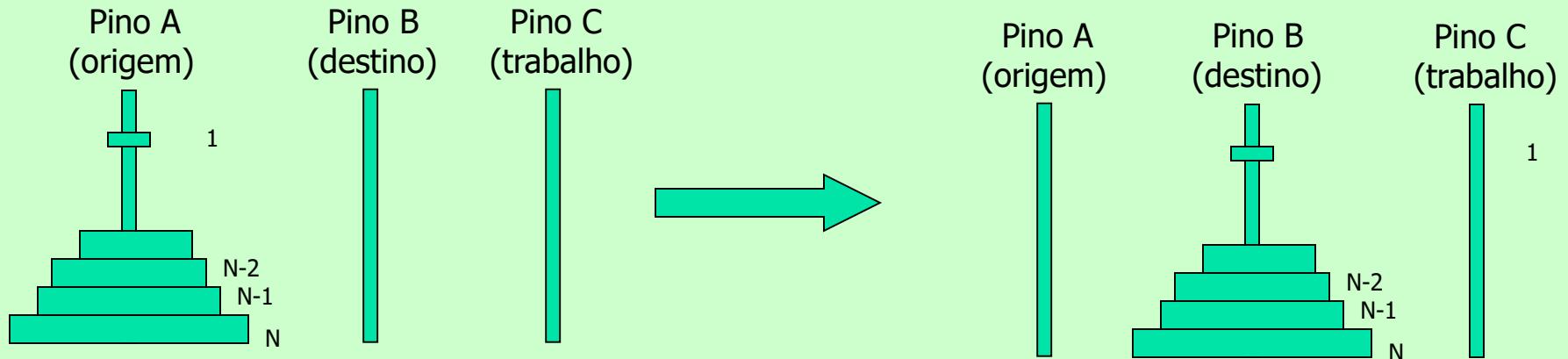
- Importante recurso computacional para o desenvolvimento de algoritmos;
- Possibilita que uma função (ou procedimento) possua uma, ou mais, chamadas a si mesmo.
- Recursividade garante a concisão mas não garante ganho em eficiência computacional

Ex: factorial\_recursivo(n)

```
begin;
    if n = 0 then
        return(1);
    else
        return(i * factorial_recursivo(n-1) );
    endIf
end_Fatorial_recursivo;
```

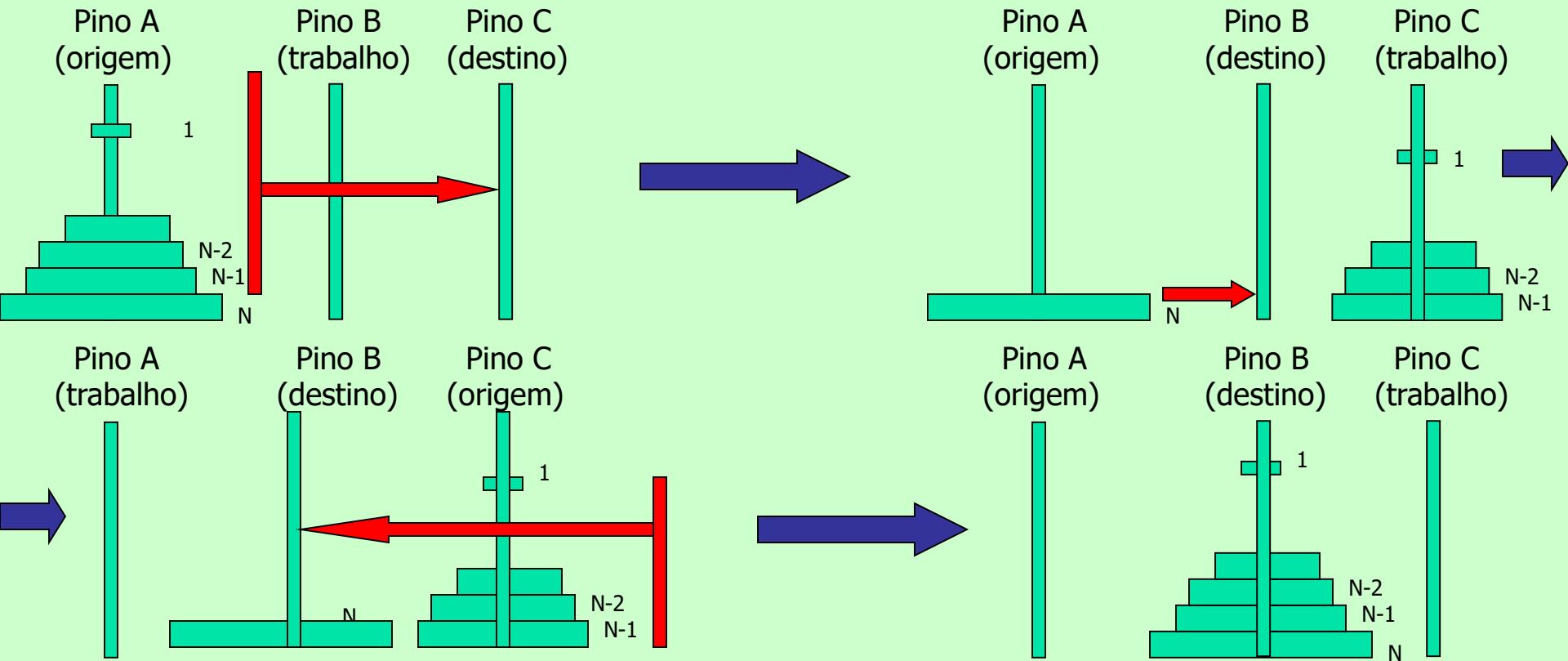
# Hanoi

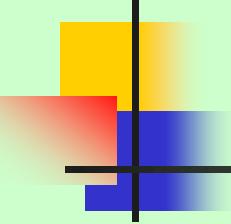
- Ex2: Problema da Torre de Hanoi: Consiste de 3 pinos (A B e C) respectivamente descritos como origem, destino e trabalho. Inicialmente, tem-se  $n$  discos empilhados no pino A (origem), em ordem decrescente de tamanho, de baixo para cima. O objetivo é empilhar todos os  $n$  discos no pino B (destino), respeitando-se as seguintes restrições:
  - (1) apenas um disco pode ser movido de cada vez;
  - (2) qualquer disco JAMAIS pode ser colocado sobre outro disco com diâmetro menor.



# Hanoi

- Solução recursiva proposta:
  - (1) Resolver o problema pra  $n-1$  discos do topo do pino A, adotando-se C como destino e B como trabalho;
  - (2) mover o  $n$ -éssimo pino (o maior) da origem (A) para destino (B);
  - (3) Resolver o problema para os  $n-1$  discos de C, supondo-se C como origem, A como trabalho e B como destino





# Hanoi

- Função Hanoi ( nº de discos,  
Pino de origem,  
Pino de destino,  
Pino de trabalho)

- Algoritmo:

```
Hanoi(n,A,B,C)
```

```
Begin
```

```
    if n > 0 then
```

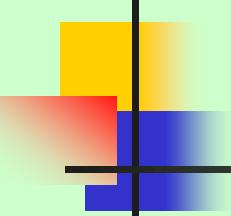
```
        Hanoi (n-1, A, C, B); /* ANTES: origem = A , destino = B, trabalho = C  
                                DEPOIS: origem = A , destino = C, trabalho = B */
```

```
        Move_Disco( origem -> destino); /* A -> B */
```

```
        Hanoi (n-1, C, B, A); /* ANTES: origem = A , destino = C, trabalho = B  
                                DEPOIS: origem = C , destino = B, trabalho = A */
```

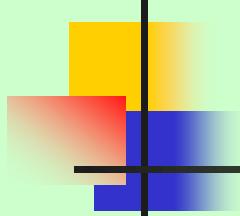
```
    endIF
```

```
endHanoi;
```



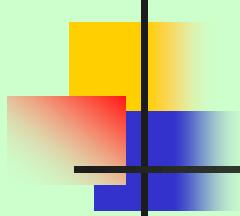
# Complexidade Algorítmica

- Complexidade de Algoritmos:
  - Avaliar eficiência do algoritmo é avaliar seu tempo de execução computacional;
  - O tempo computacional é uma grandeza empírica pois depende de parâmetros tipo o hardware, inputs,etc..
  - Tempo computacional pode ser definido a partir de métodos analítico objetivando calcular a ordem de grandeza do tempo de execução;
  - Método analítico = determinação de uma expressão matemática que descreve o comportamento do tempo de execução requerido por um algoritmo;
  - O tempo de execução passa a ser definido de forma independente do compilador, linguagem, hardware, etc



# Complexidade Algorítmica

- Considerações gerais para o cálculo da complexidade de um algoritmo:
  - (1)  $n$  = quantidade de dados (muito grande !!!);
  - (2) não serão consideradas as constantes de multiplicação e de adição:  
Ex:  $2 * n = n$   
 $n - 56 = n$
  - (4) Algoritmo manipula dados => o tempo de execução deverá ser expresso em função da quantidade de dados  $n$ ;
  - (5) A execução de um algoritmos deverá ser dividida em passo ou blocos;
  - (6) Cada passo (ou bloco) possui um numero fixo de operações básicas (instruções) com tempo de execução constante;
  - (7) Operação ou passo dominante é aquela que mais freqüentemente é executada no algoritmo;
  - (8) A função que avalia o tempo de execução de um algoritmo é a função matemática que fornece o numero de vezes que o bloco dominante é executado para um determinado numero  $n$  de dados;



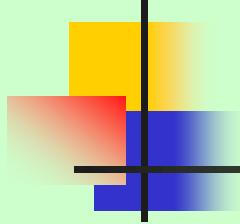
# Complexidade Algorítmica

- Ex1: Faça a inversão de uma seqüência de elementos armazenados em um vetor  $S[i]$  onde  $0 < i < (n-1)$ .
  - Input =  $S = [ A \ B \ C \ D \ E ]$
  - Output =  $S_{\text{invert}} = [ E \ D \ C \ B \ A ]$
  - ALGORITMO:

```
For i = 1 to maior_int(n/2)
    temp=S[i];
    S[i]=S[n-i+1];
    S[n-i+1]=temp;
endfor;
```

bloco dominante

Tempo de execução =  $T(n/2)$



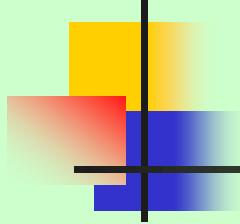
# Complexidade Algorítmica

Ex: Soma de 2 matrizes de dimensões n x m

```
soma (M1[n][m], M2[n][m])
begin
    Matrix Msol[n][m];
    int i, j;
    for i = to n
        for j = 1 to m
            Msol[i][j] = M1[i][j] + M2[i][j];
    endFor;
    endFor;
endSoma;
```

**bloco dominante** →

**Tempo de execução = T(n \* m)**



# Complexidade Algorítmica

Ex: Multiplicação de 2 matrizes de dimensões ( $n \times p$ ) e ( $p \times m$ )

```
Multiplica (M1[n][p], M2[p][m])
begin
    Matrix Msol[n][m];
    int i, j, k;
    for i = to n
        for j = 1 to m
            for k = 1 to p
                Msol[i][j] = Msol[i][j] + M1[i][k] + M2[k][j];
            endFor;
        endFor;
    endFor;
endMultiplica;
```

Bloco dominante →

$M_{sol}[i][j] = M_{sol}[i][j] + M1[i][k] + M2[k][j];$

Tempo de execução =  $T(p * m * n)$

# Complexidade Algorítmica

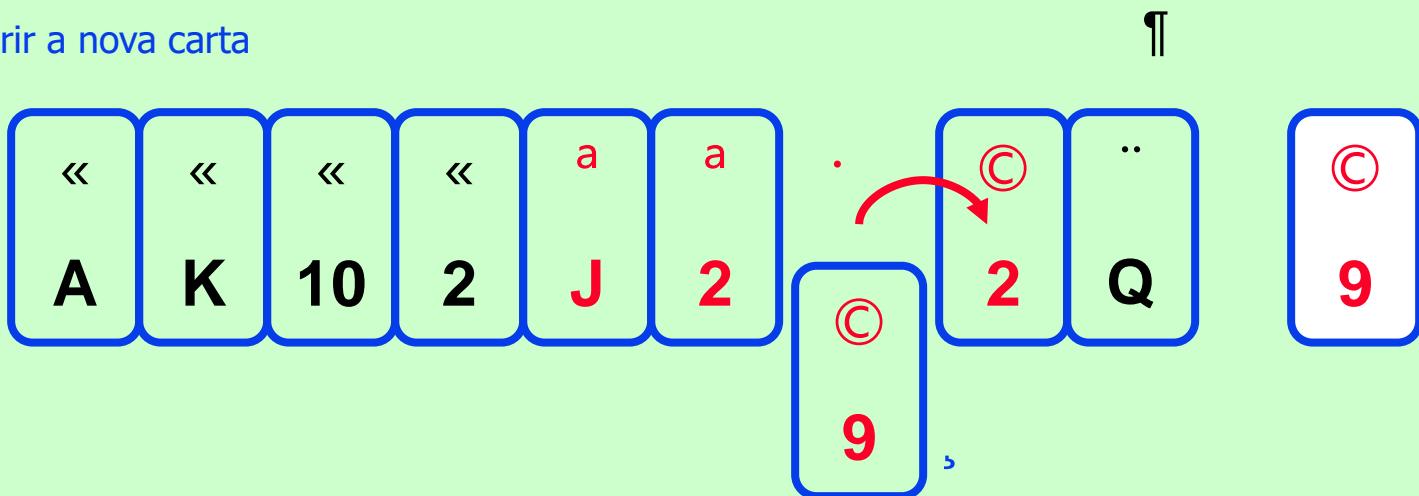
- EX: Ordenação por inserção (usada para pequeno numero de dados e similar a ordenação executada em jogos de baralho):

Input [65, 2, 190, 200, 4, 67, 0, 55]

Output[0, 2 , 4, 55, 65, 67, 190, 200]

Em jogo de cartas ...

- A primeira carta está ordenado
- Para as seguintes,
  - Varrer do final para o início até a primeira carta maior que a nova,
  - Mover as menores uma casa acima
  - inserir a nova carta



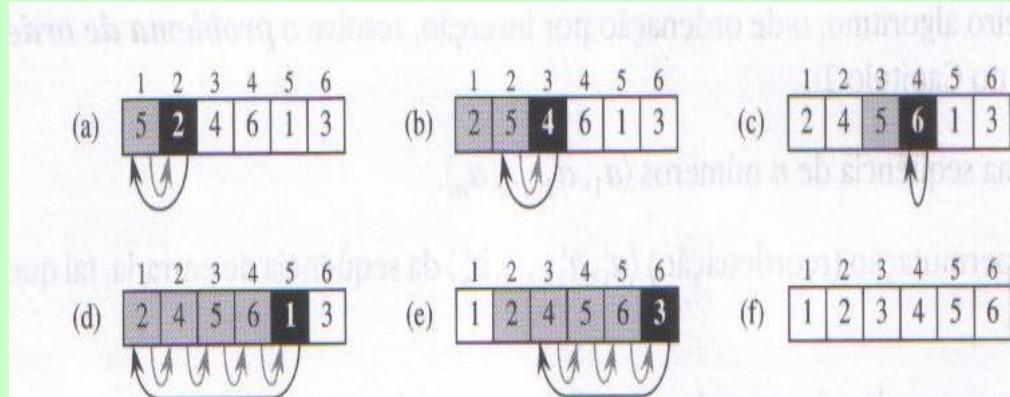
# Complexidade Algorítmica - Sort

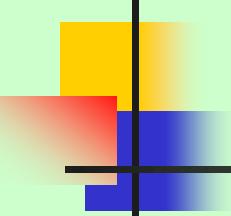
Insert\_Sort(A[n])

```
begin
    for i= 2 to n      /*especifica a variação do outer*/
        temp = A[i];
        j = i -1;
        while j > 0 and A[j] > temp
            A[j+1]=A[j];
            j = j -1;
        endwhile
        A[j+1] = temp
    endFor;
endInsert_Sort;
```

$T(n)$  → | ←  $T(n^2)$  || →  $T(n^2 + n)$

/\*especifica a variação do inner\*





# Complexidade Algorítmica - Sort

- Ex: Ordenação => algoritmo de Bubble\_Sort(A[n]);

```
BubbleSort(A[n])
begin
    for i = 1 to n
        for j = n to i
            if A[j] < A[j-1] then
                Temp= A[j-1];
                A[j-1] = A[j];
                A[j] = temp;
            endIf;
        endFor;
    endFor;
endBubble_Sort
```

$T(n^2)$

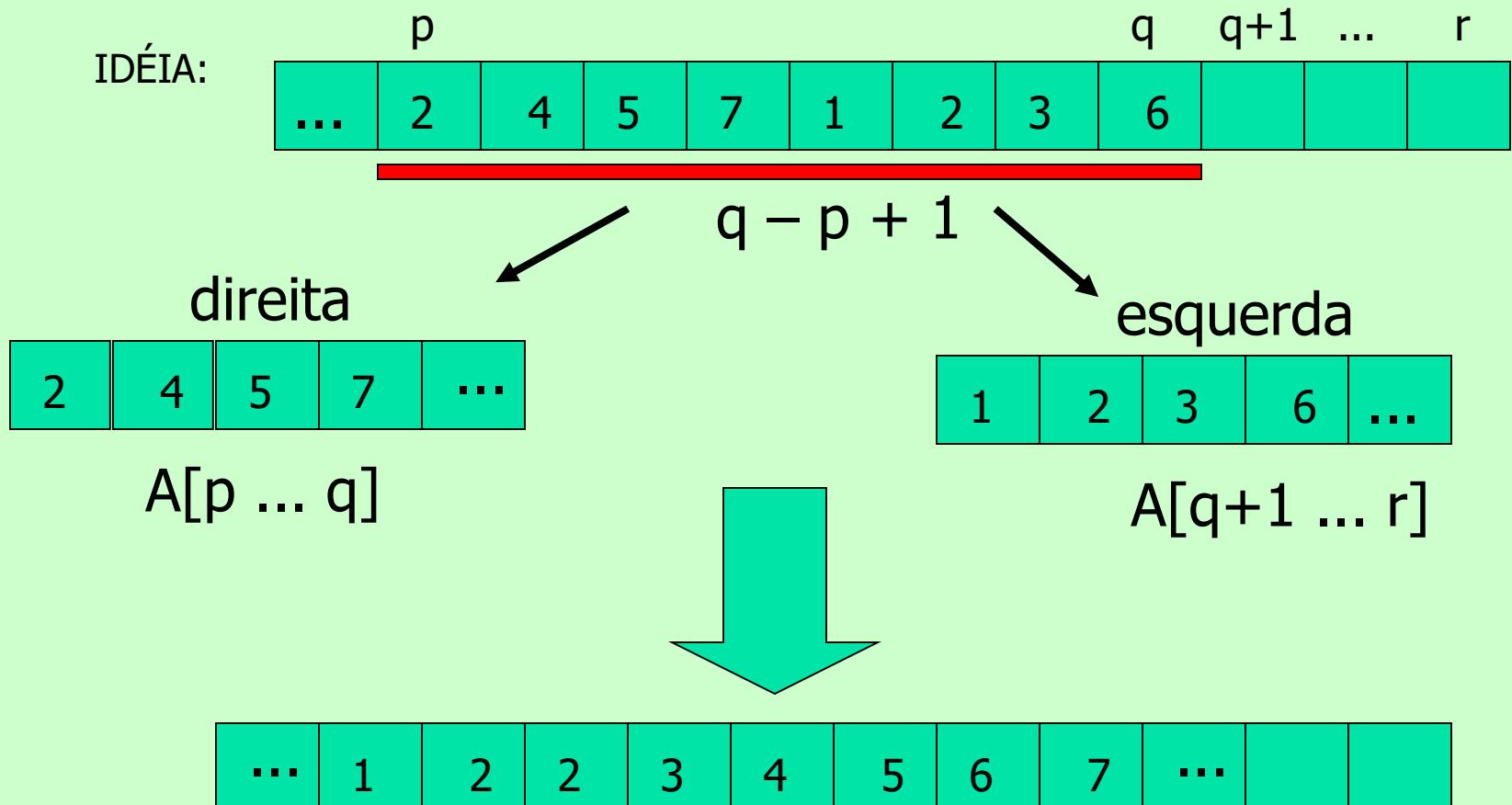


• **Do primeiro elemento**

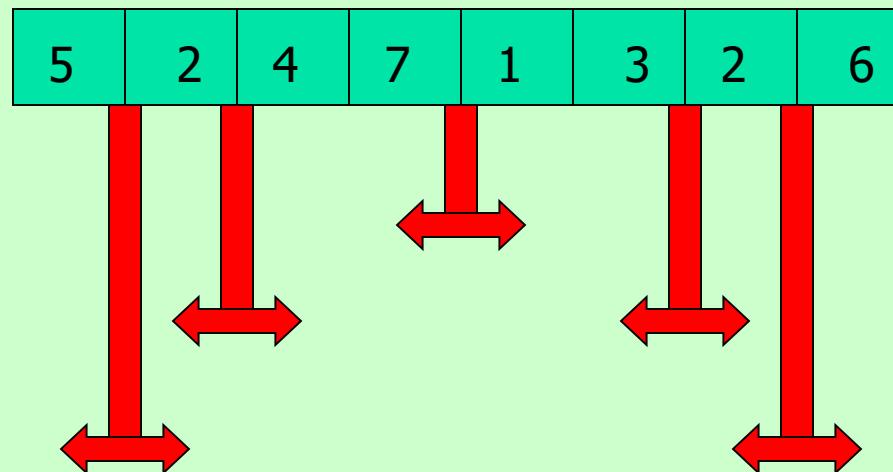
- Trocar os pares se estão fora de ordem
  - A última deve ser a maior
- Repetir da primeira até a n-1
- Parar quando existir apenas um elemento para verificar

# Complexidade Algorítmica - Sort

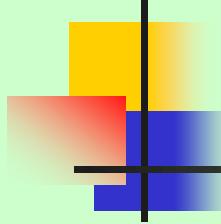
- EX: MergeSort : dividir + conquistar + combinar



# Complexidade Algorítmica - Sort



```
Algoritmo: Merge_Sort (A, p, r)
begin
    if (p < r) then
        q = menor_inteiro( (p +r) / 2 );
        Merge_Sort(A, p, q);
        Merge_Sort(A, q+1, r);
        MERGE (A, p, q, r);
    endIf
endMerge_Sort;
```



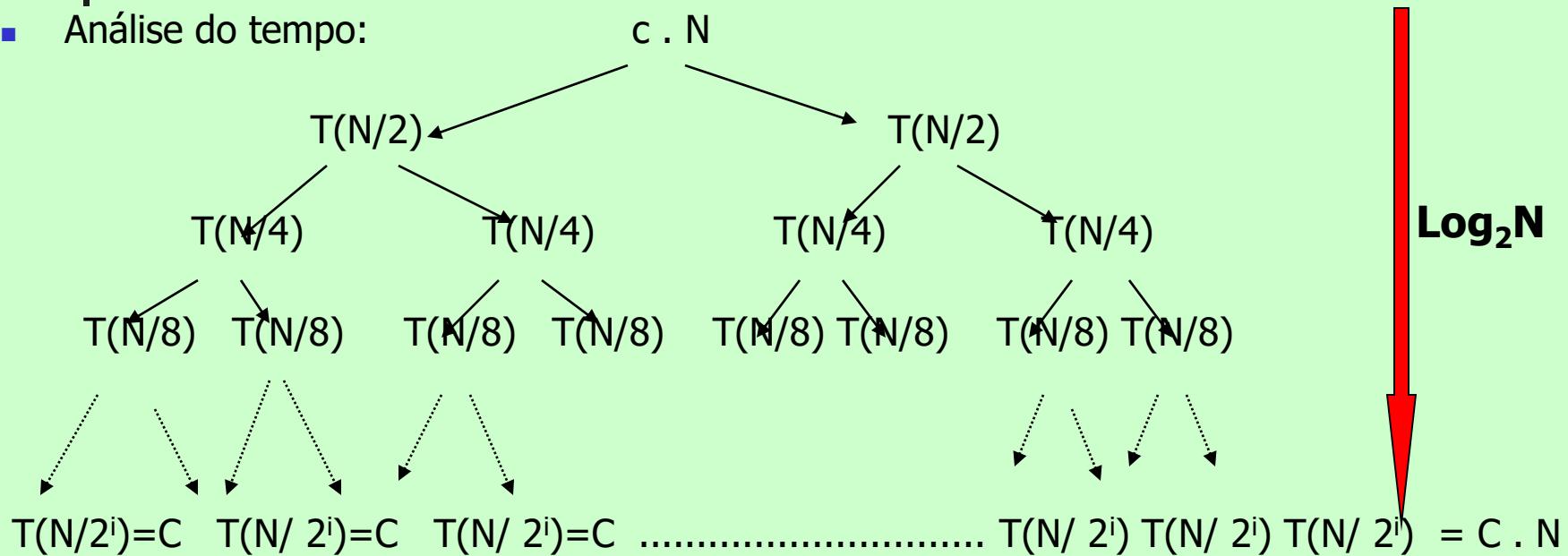
# Complexidade Algorítmica - Sort

**Algoritmo** MERGE (A, p, q, r)

```
begin
    n1= q - p + 1;
    n2 = r - q;
    CRIAR dir[1....n2+1];
    CRIAR esq[1....n1+1];
    for i = 1 to n1
        esq[i] = A[p+i-1];
    endfor;
    for i = 1 to n1
        dir[i] = A[q+i];
    endfor;
    esq[n1+1] = dir[n2+1] = 100000000000;
    i = j = 1;
    for k = p to r
        if esq[i] <= dir[j] then
            A[k] = esq[i];
            i++;
        else
            A[k] = dir[j];
            j++;
        endif
    endFor
end_MERGE;
```

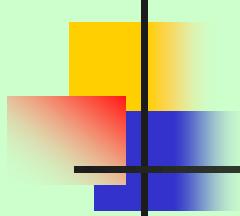
# Complexidade Algorítmica - Sort

- Análise do tempo:



LOGO: tempo =  $N \cdot \log_2 N + N$  , pois:

- custo de cada nó no nível  $i = N / 2^i$
- número de nós no nível  $i = 2^i$
- número de níveis =  $\log_2 N + 1$



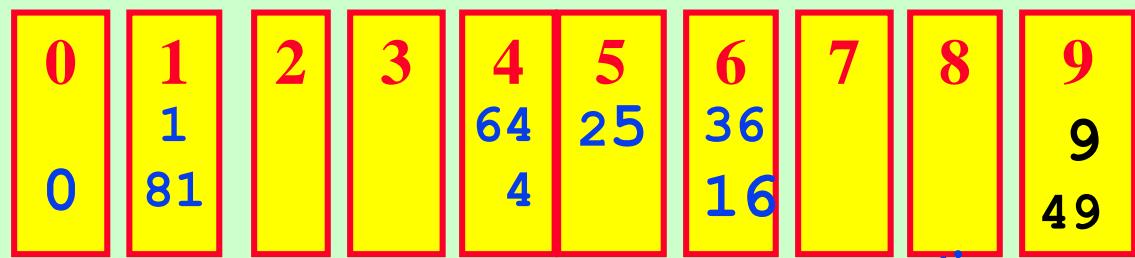
# Complexidade Algorítmica - Sort

- Ex: Ordenação – Bin
  - Todas as chaves em um faixa estreita
    - Existem  $m$  chaves possíveis
  - Não existem chaves duplicatas
- Ordenação Bin
  - Alocar um bin para cada valor da chave       $T(m)$ 
    - Geralmente uma entrada em uma lista
  - Para cada item,                                   $n$  vezes
    - Extrair a chave                                   $T(1)$
    - Computar a posição     $T(1)$
    - Colocar no bin                                   $T(1) \times n \subset T(n)$
  - Então:  $T(n) + T(m) = T(n+m) = T(n)$     se     $n \gg m$

# Complexidade Algorítmica - Sort

- Ordenação Bin em fases
    - Fase 1 - Ordenar pelo digito mais significativo

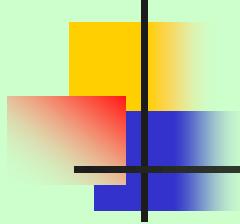
**Exemplo:** 36  
9 0 25 1 49  
64 16 81 4



- Fase 2 - Ordenar pelo digito mais significativo

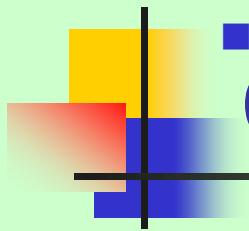
Qual é o espaço  
necessário em  
cada fase?  
*n* itens  
*m* bins





# Complexidade Algorítmica - Sort

- Inserção       $T(n^2)$
- Bubble       $T(n^2)$
- Merge       $T(n \log n)$       ***na maioria dos casos!***
  - Bom desempenho - o tempo não pode ser crítico
- Bin       $T(n)$       ***conjuntos pequenos !!***
  - Troca tempo por espaço de memória
    - (Não existe banquete grátis)

- 
- Uma vez compreendido a definição da função tempo em um algoritmo, pode-se analisar a complexidade do tempo:

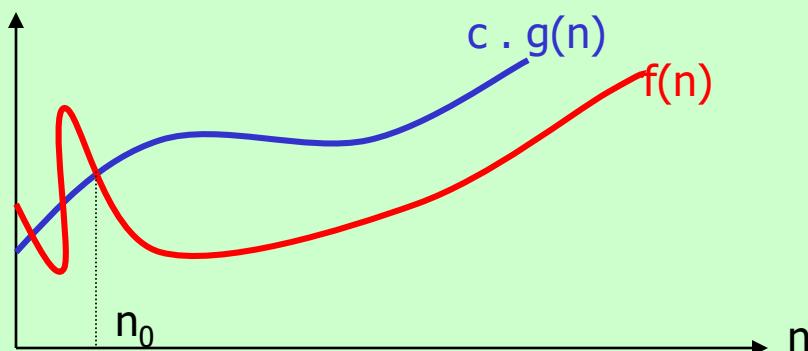
- Seja um algoritmo A, suas complexidades podem ser:
  - Complexidade no pior caso (máximo tempo)
  - Complexidade no melhor caso (mínimo tempo)
  - Complexidade de caso médio (tempo médio a partir de um modelo probabilístico, como o  $\sum_{(1 \leq i \leq m)} p_i \cdot T_i$ , onde  $p_i$  é a probabilidade de ocorrer  $T_i$ ;

Então: Complexidade do algoritmo é a complexidade no pior caso !!!!

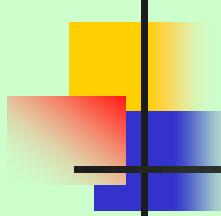
# Complexidade Algorítmica

## ■ Notação O

- Seja  $f$  e  $g$  funções reais positivas,  $n$  a variável dependente inteira:
- ENTÃO:  $f(n) = O(g(n))$  se existir uma constante  $c > 0$ , e um valor inteiro  $n_0 > 0$ , tal que:
  - $n > n_0 \Rightarrow f(n) \leq c \cdot g(n)$
  - OBS:  $f(n) = O(g(n))$  está indicando que a função  $f(n)$  cresce (ou decresce) conforme a função  $g(n)$  também cresce (ou decresce).
  - Ou seja, a função  $g(n)$  descreve o limite superior para os valores assintóticos da função  $f(n)$ :



$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq c$$



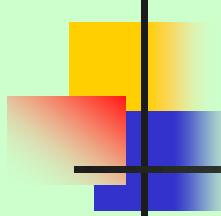
# Complexidade Algorítmica

## Propriedades da Notação O:

- Fatores Constantes podem ser ignorados
  - $\forall k > 0$ ,  $kf$  é  $O(f)$
- Potências maiores crescem mais rápido
  - $n^r$  é  $O(n^s)$  se  $0 \leq r \leq s$

OBS: O termo que cresce mais rápido domina uma soma

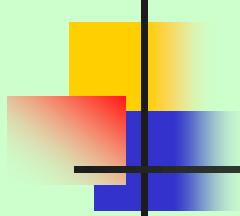
  - Se  $f$  é  $O(g)$ , então  $f + g$  é  $O(g)$   
*i.e.*  $an^4 + bn^3$  é  $O(n^4)$
- A taxa de crescimento Polinomial é determinada pelo termo de mais alto grau
  - Se  $f$  é um polinômio de grau  $d$  então  $f$  é  $O(n^d)$



# Complexidade Algorítmica

## Propriedades da Notação O:

- $f$  é  $O(g)$  é transitiva
  - Se  $f$  é  $O(g)$  e  $g$  é  $O(h)$  então  $f$  é  $O(h)$
- Produto de limites superiores é o limite superior para o produto
  - Se  $f$  é  $O(g)$  e  $h$  é  $O(r)$  então  $fh$  é  $O(gr)$
- Funções Exponenciais crescem mais rápido que potências
  - $n^k$  é  $O(b^n)$   $\forall b > 1$  e  $k \geq 0$   
*ie*  $n^{20}$  é  $O(1.05^n)$
- Logaritmos crescem mais devagar que potências
  - $\log_b n$  é  $O(n^k)$   $\forall b > 1$  e  $k > 0$   
*ie*  $\log_2 n$  é  $O(n^{0.5})$

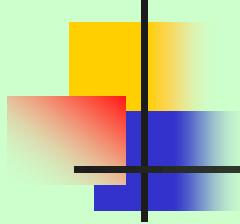


# Complexidade Algorítmica

## Propriedades da Notação O:

- Todos logaritmos crescem na mesma taxa
  - $\log_b n$  é  $O(\log_d n)$   $\forall b, d > 1$
- A soma da primeira  $n^{r^{\text{th}}}$  potência cresce como a potência  $(r+1)^{\text{th}}$ 
  - $\sum k^r$  é  $\Theta(n^{r+1})$

i.e. 
$$\sum_{k=1}^n i = \frac{n(n+1)}{2} \Rightarrow O(n^2)$$



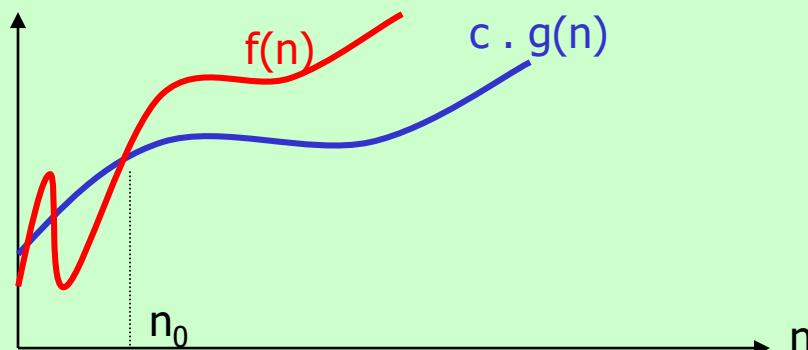
# Complexidade Algorítmica

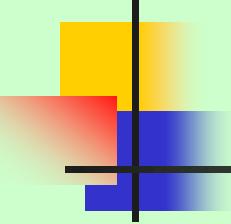
- Exs válidos:
  - $f(n) = n^2 - 1 \Rightarrow f = O(n^2) \Rightarrow g(n) = n^2;$
  - $f(n) = n^2 - 1 \Rightarrow f = O(n^3) \Rightarrow g(n) = n^3;$
  - $f(n) = n^3 - 1 \Rightarrow f = O(n^3) \Rightarrow g(n) = n^3;$
  - $f(n) = 403 \Rightarrow f = O(1) \Rightarrow g(n) = 1;$
  - $f(n) = 2 \cdot \log n - 1 \Rightarrow f = O(\log n) \Rightarrow g(n) = \log n;$
  - $f(n) = 2 \cdot n \cdot \log n - 1 \Rightarrow f = O(n \cdot \log n) \Rightarrow g(n) = n \cdot \log n;$
  - $f(n) = 2 \cdot n \cdot \log n - 1 \Rightarrow f = O(n^2) \Rightarrow g(n) = n^2;$
- Exs NÃO válidos:
  - $f(n) = n^2 - 1 \Rightarrow f = O(n) \Rightarrow g(n) = n;$  (**ERRO !!!**)
  - $f(n) = n^2 - 1 \Rightarrow f = O(n \cdot \log n) \Rightarrow g(n) = n \cdot \log n;$  (**ERRO !!!**)
  - $f(n) = n^3 - 1 \Rightarrow f = O(n^2) \Rightarrow g(n) = n^2;$  (**ERRO !!!**)
  - $f(n) = 2 \cdot n \cdot \log n - 1 \Rightarrow f = O(\log n) \Rightarrow g(n) = \log n;$  (**ERRO !!!**)
  - $f(n) = 2 \cdot n \cdot \log n - 1 \Rightarrow f = O(n) \Rightarrow g(n) = n;$  (**ERRO !!!**)

# Complexidade Algorítmica

## ■ Notação ' $\Omega$ '

- Seja  $f$  e  $g$  funções reais positivas,  $n$  a variável dependente inteira:
- ENTÃO:  $f(n) = \Omega(g(n))$  se existir uma constante  $c > 0$ , e um valor inteiro  $n_0 > 0$ , tal que:
  - $n > n_0 \Rightarrow f(n) \geq c \cdot g(n)$
  - OBS:  $f(n) = \Omega(g(n))$  esta indicando que a função  $g(n)$  cresce (ou decresce) conforme a função  $f(n)$  também cresce (ou decresce).
  - Ou seja, a função  $g(n)$  descreve o limite inferior para os valores assintóticos da função  $f(n)$ :



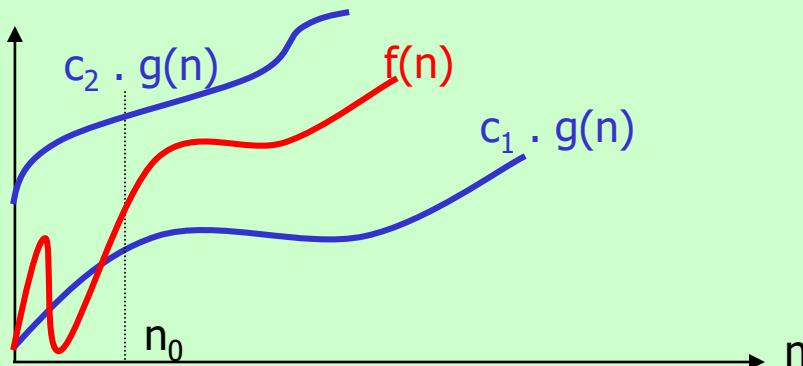


# Complexidade Algorítmica

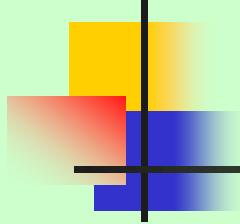
- Exs válidos:
  - $f(n) = n^2 - 1 \Rightarrow f = \Omega(n^2) \Rightarrow g(n) = n^2;$
  - $f(n) = n^2 - 1 \Rightarrow f = \Omega(n) \Rightarrow g(n) = n;$
  - $f(n) = n^3 - 1 \Rightarrow f = \Omega(1) \Rightarrow g(n) = 1;$
  - $f(n) = 2 \cdot \log n - 1 \Rightarrow f = \Omega(1) \Rightarrow g(n) = 1;$
  - $f(n) = 2 \cdot n \cdot \log n - 1 \Rightarrow f = \Omega(n) \Rightarrow g(n) = n;$
  - $f(n) = 2 \cdot n \cdot \log n - 1 \Rightarrow f = \Omega(\log n) \Rightarrow g(n) = \log n;$
- Exs NÃO válidos:
  - $f(n) = n^2 - 1 \Rightarrow f = \Omega(n^3) \Rightarrow g(n) = n^3; \text{ (ERRO !!!)}$
  - $f(n) = n^3 - 1 \Rightarrow f = \Omega(n^4) \Rightarrow g(n) = n^4; \text{ (ERRO !!!)}$
  - $f(n) = 2 \cdot n \cdot \log n - 1 \Rightarrow f = \Omega(n^2) \Rightarrow g(n) = n^2; \text{ (ERRO !!!)}$
  - $f(n) = 2 \cdot n \cdot \log n - 1 \Rightarrow f = \Omega(n^3) \Rightarrow g(n) = n^3; \text{ (ERRO !!!)}$

# Complexidade Algorítmica

- Notação  $\theta$ 
  - Seja  $f$  e  $g$  funções reais positivas,  $n$  a variável dependente inteira:
  - ENTÃO:  $f(n) = \theta(g(n))$  se existir duas constantes  $c_1$  e  $c_2 > 0$ , e um valor inteiro  $n_0 > 0$ , tal que:
    - $n > n_0 \Rightarrow f(n) \geq O(g(n))$  e  $g(n) \leq O(f(n))$ ;
    - $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ ;
    - Indica que  $f(n)$  está no espaço descrito pela intercessão do valores  $O(g(n))$  e  $\Omega(g(n))$ , que chamamos de limite assintótico restrito;

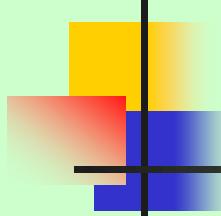


OBS: note que  $g(n)$  e  $f(n)$  possuem a mesma ordem de grandeza assintótica !!!



# Complexidade Algorítmica

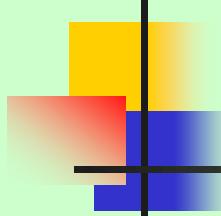
- Ex válido:
  - $f(n) = \frac{1}{2} \cdot n^2 - 3 \cdot n$ ;  $g(n) = n^2$ ;
  - $f(n) = \theta(g(n))$  ?????
  - Análise:  $c_1 \cdot n^2 < \frac{1}{2} \cdot n^2 - 3 \cdot n < c_2 \cdot n^2 = c_1 < \frac{1}{2} - 3 / n < c_2$ ; para todo  $n > 0$  ;
  - **LOGO : É VÁLIDO !!!!!**
- Ex NÃO válido:
  - $f(n) = n^3$ ;  $g(n) = n^2$ ;
  - $f(n) = \theta(g(n))$  ?????
  - Análise:  $c_1 \cdot n^2 < n^3 < c_2 \cdot n^2 = c_1 < n < c_2$ ;  $n$  válido somente entre  $c_1$  e  $c_2$  !!!!!**LOGO : NÃO VÁLIDO !!!!!**



# Complexidade Algorítmica

## ■ Notação o

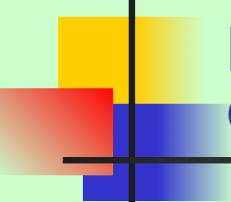
- Seja  $f$  e  $g$  funções reais positivas,  $n$  a variável dependente inteira:
- ENTÃO:  $f(n) = o(g(n))$  se existir uma constante  $c > 0$ , e um valor inteiro  $n_0 > 0$ , tal que:
  - $n > n_0 \Rightarrow f(n) < c \cdot g(n)$
  - Ou seja, a função  $g(n)$  descreve o limite superior **RESTRITO** para os valores assintóticos da função  $f(n)$ , ou seja:
$$\lim_{n \rightarrow \infty} \left( \frac{f(n)}{g(n)} \right) = 0$$
- Ex:  $f(n) = 2n^2$      $O(n^2)$  VÁLIDO  
                         $o(n^2)$  NÃO VÁLIDO
- Ex:  $f(n) = 2n$      $O(n^2)$  VÁLIDO  
                         $o(n^2)$  VÁLIDO



# Complexidade Algorítmica

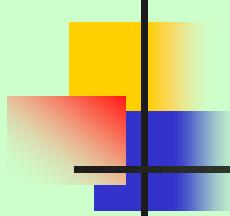
## ■ Notação w

- Seja  $f$  e  $g$  funções reais positivas,  $n$  a variável dependente inteira:
- ENTÃO:  $f(n) = o(g(n))$  se existir uma constante  $c > 0$ , e um valor inteiro  $n_0 > 0$ , tal que:
  - $n > n_0 \Rightarrow f(n) < c \cdot g(n)$
  - Ou seja, a função  $g(n)$  descreve o limite inferior RESTRITO para os valores assintóticos da função  $f(n)$ , ou seja:
$$\lim_{n \rightarrow \infty} \left( \frac{f(n)}{g(n)} \right) = \infty$$
  - Ex:  $f(n) = 2n^2 \rightarrow \Omega(n^2)$  VÁLIDO  
 $w(n^2)$  NÃO VÁLIDO  
 $\Omega(n)$  VÁLIDO  
 $w(n)$  VÁLIDO



# Como devemos resolver alguns problemas ?????? Será que as soluções são realmente eficientes ???

1. Como poderemos achar uma celebridade em uma multidão, onde todos conhecem a celebridade (que não conhece ninguém da multidão) ???
2. Como podemos saber qual o candidato vence por maioria absoluta (50% + 1 voto) uma eleição ???
3. Como faz nosso celular para achar o nome de um amigo armazenado na agenda ????
4. Como podemos enviar uma mensagem de forma que ninguém saiba quem enviou, e qual o conteúdo da mensagem. Claro que somente o receptor final, terá acesso ao conteúdo da mensagem e quem a enviou ???
5. Como devemos fazer a otimização da distribuição de jornais em um bairro com várias bancas e somente 3 pontos de distribuição localizados de forma aleatória no bairro ???
6. De uma forma emergencial, como aumentar a produção em uma fábrica com mais de uma linha de montagem, sabendo que em cada linha de montagem existem os mesmos pontos de fabricação, os quais algumas linhas de montagem possuem máquinas mais modernas (e rápidas) e máquinas mais antigas (não tão rápidas, porém ainda eficientes) ???
7. Como devemos otimizar a execução de tarefas, que de forma dinâmica, estão sendo suas prioridades de execução alteradas ???
8. Como selecionar seus 5 melhores funcionários (baseado em um valor de produtividade individual) para premia-los com um final de semana em um resort ???



# Complexidade Algorítmica

## ■ Dicas para a análise de algoritmos:

- Seqüência Simples de operações

$s_1; s_2; \dots; s_k$

- $O(1)$  desde que  $k$  seja constante

- Loops Simples

`for(i=0; i<n; i++) { s; }`

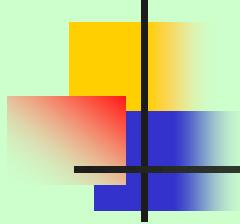
onde  $s$  é  $O(1)$

- Complexidade de Tempo é  $n$   $O(1)$  ou  $O(n)$

- Loops Aninhados

```
for(i=0; i<n; i++)
    for(j=0; j<n; j++) { s; }
```

- Complexidade é  $n$   $O(n)$  ou  $O(n^2)$



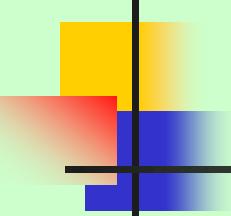
# Complexidade Algorítmica

- **Dicas para a análise de algoritmos:**

- O índice de um Loop não varia linearmente

```
h = 1;  
while ( h <= n ) {  
    s;  
    h = 2 * h;  
}
```

- $h$  assume valores  $1, 2, 4, \dots$  até ele exceder  $n$
- Existem  $1 + \log_2 n$  iterações
- Complexidade  $O(\log n)$



# Complexidade Algorítmica

## ■ Dicas para a análise de algoritmos:

- O índice de um Loop depende do índice de um loop externo

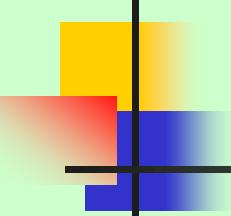
```
for (j=0; j<n; j++)  
    for (k=0; k<j; k++) {  
        s;  
    }
```

- loop interno executado
  - 1, 2, 3, ...., n **vezes**

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

- Complexidade  $O(n^2)$

Distinguir este caso - onde o contador da iteração cresce (decresce) por uma constante  $\in O(n^k)$  do anterior - onde ele muda por um fator  $\in O(\log n)$



# Complexidade Algorítmica

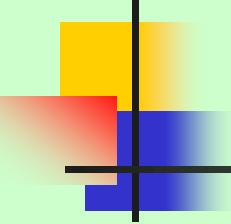
## ■ **Dicas para a análise de algoritmos:**

### ■ Cálculo da complexidade em algoritmos recursivos:

- 1a: determinar o numero total de chamadas ao processo recursivo (rec)
  - 2a: calcule a complexidade correspondente a uma única chamada , ou seja, não considerando as chamadas recursivas; (complex);
  - 3a: Complexidade = rec . Complex
- 
- Ex: Cálculo da fatorial recursivo: rec = n ; complex = O(1); **Complexidade = n. O(1) = O(n)**

```
fatorial_recursivo(n)
begin;
    if n = 0 then

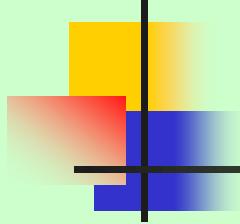
        return(1);
    else
        return(i * factorial_recursivo(n-1) );
    endIf
end_Fatorial_recursivo;
```



# Complexidade Algorítmica

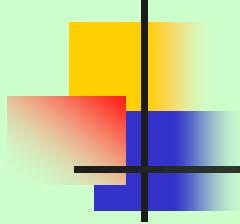
- EX: Torre de Hanoi: rec =  $2^n$ ; complex= O(1); **Complexidade =  $2^n \cdot O(1)$**   
**=  $O(2^n)$**

```
Hanoi(n,A,B,C)
Begin
    if n > 0 then
        Hanoi (n-1, A, C, B); /* ANTES: origem = A , destino = B, trabalho = C
                                    DEPOIS: origem = A , destino = C, trabalho = B */
        Move_Disco( origem -> destino); /* A -> B */
        Hanoi (n-1, C, B, A); /* ANTES: origem = A , destino = C, trabalho = B
                                    DEPOIS: origem = C , destino = B, trabalho = A */
    endIF
endHanoi;
```



# Complexidade Algorítmica

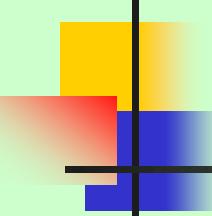
- Cálculo de caso médio (modelo probabilístico):
  - Suponha um algoritmo que, dados 2 matrizes A e B, tem-se:
    - Se  $x = 1$ , faça  $A \cdot B$ ; ( Complexidade  $O(n^3)$  )
    - Se  $x = 0$ , faça  $A + B$ ; ( Complexidade  $O(n^2)$  )
  - Complexidade =  $O( q \cdot n^3 + (1-q) \cdot n^2 )$ ;
    - $q$  = probabilidade de ocorrer  $x = 1$ ;
    - $1 - q$  = probabilidade de ocorrer  $x = 0$ ;



# Complexidade Algorítmica

---

- RESUMO:
  - Cada algoritmo tem sua complexidade independente de outros algoritmos já propostos para resolver um determinado problema. Ou seja, a complexidade de um algoritmos (solução computacional) descreve o numero de passos executados (tempo computacional – grandeza empírica) na resolução de um determinado problema em conformidade com um modelo computacional proposto.



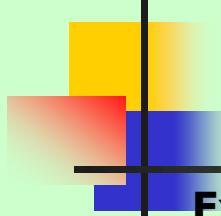
# Complexidade Algorítmica

- Então, algoritmo dito **ótimo** é:

- O algoritmo que, dentre os possíveis algoritmos propostos para a resolução de um mesmo problema, apresenta a MENOR complexidade:

Ex: Dado um problema  $\underline{P}$ , onde  $g(n)$  é a função de limite inferior (definido pela análise direta sobre o problema );

- Logo: ' $\Omega(g(n))$ ' é a complexidade de pior caso de qualquer algoritmo que resolve o problema  $\underline{P}$ . Ou seja, todo e qualquer algoritmo que resolve  $P$  executa ao menos ' $\Omega(g(n))$ ';
- ENTÃO: Se existir um algoritmo  $\underline{A}$  de complexidade  $O(g(n))$ , então  $\underline{A}$  é dito um **algoritmo ótimo** para o problema  $\underline{P}$ .



# Complexidade Algorítmica

## Exemplos:

(1) Inversão de um vetor:

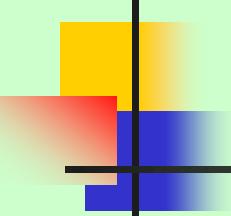
- (1) Limite inferior natural do problema = ' $\Omega(n)$ '
- (2) Complexidade do algoritmo proposto =  $O(n)$ ;
- (3) Logo, o algoritmo proposto é **ÓTIMO**;

(2) Soma de matrizes:

- (1) Limite inferior natural do problema = ' $\Omega(n^2)$ '
- (2) Complexidade do algoritmo proposto =  $O(n^2)$ ;
- (3) Logo, o algoritmo proposto é **ÓTIMO**;

(3) Produto de matrizes:

- (1) Limite inferior natural do problema = ' $\Omega(n^2)$ '
- (2) Complexidade do algoritmo proposto =  $O(n^3)$ ;
- (3) Logo, o algoritmo proposto é **NÃO ÓTIMO**;



# Complexidade Algorítmica

**OBS: O cálculo do limite inferior natural é NÃO TRIVIAL;**

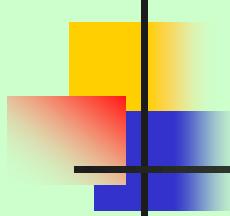
**A busca de algoritmos ótimos é uma das áreas mais importantes de pesquisa na Ciência da Computação (Análise de Algoritmos).**

**OBS: problema da ordenação:**

- Limite inferior natural trivial = ' $\Omega(n)$ '
- Limite inferior matematicamente comprovado = ' $\Omega(n \cdot \log n)$ '
- Proposta algorítmica (MERGE-SORT) =  $O(n \log n)$

**(1) é dito ÓTIMO (comparando-se ao limite inferior matemático)**

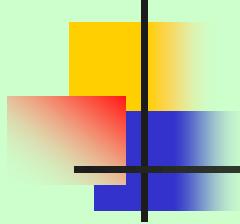
**(2) é dito NÃO ÓTIMO (comparando-se ao limite inferior NATURAL)**



# Complexidade Algorítmica

- Exemplos NÃO ÓTIMO:

- (1) SOMA ARITMÉTICA:  $1+2+3+4+\dots+n$       for i = 1 to n  
 $O(n)$  → soma = soma + i;  
Não ótimo !!  
endFor;
- (2) SOMA dos quadrados:  $1^2+2^2+3^2+4^2+\dots+n^2$     for i = 1 to n  
 $O(n)$  → soma = soma + i\*i;  
Não ótimo !!  
endFor;
- (3) série Geométrica:  $1+2+4+8+\dots+2^n$   
 $O(n^2)$  → soma = 0;  
for i = 1 to n  
exp=1;  
for j = 1 to i  
exp=exp\*2;  
endFor;  
soma = soma + exp;  
endFor;



# Complexidade Algorítmica

- Exemplos ÓTIMO:

- (1) SOMA ARITMÉTICA:  $1+2+3+4+\dots+n$

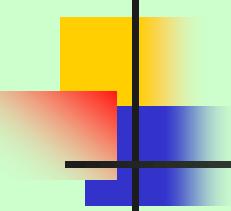
$$= [n.(n+1)]/2 \longrightarrow O(1)$$

- (2) SOMA dos quadrados:  $1^2+2^2+3^2+4^2+\dots+n^2$

$$= 2^{n+1} - 1 \longrightarrow O(1)$$

- (3) série Geométrica:  $1+2+4+8+\dots+2^n$

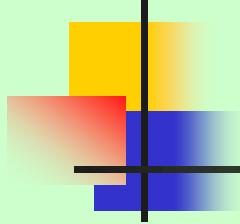
$$= [n.(n+1).(2n+1)]/6 \longrightarrow O(1)$$



# Análise Probabilística

- **Análise Probabilística e Algoritmos Aleatórios:**

- Problema da Contratação de um funcionário (**sempre o melhor !!!**);
  - Analisar cada candidato individualmente, com custo \$X por entrevista;
  - A decisão de contratar é feita logo após a entrevista, gerando um custo \$Y, onde \$Y >> \$X;
  - O objetivo é selecionar o melhor candidato, gerando a demissão do funcionário atual (sempre pior do que o novo encontrado). Mesmo que isso leve a um aumento do custo geral da contratação;
  - Algoritmo CONTRATAÇÃO()
    - begin
    - melhor\_cand = 0; /\*candidato fictício \*/
    - for i to n /\* n = numero de candidatos \*/
    - Do entrevista candidato\_i ;
    - custo = custo + \$X;
    - if candidato\_i MELHOR melhor\_cand then
    - melhor\_cand = i;
    - contratar i;
    - custo = custo + \$y
    - enfIF
    - endFOR
    - endCONTRATAÇÃO;



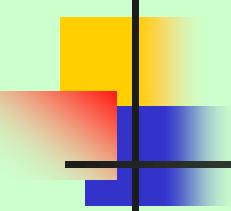
# Análise Probabilística

## ■ Análise do custo da contratação:

- Pior caso será quando os candidatos estiverem em ordem CRESCENTE de qualificação para o cargo:
  - CUSTO = O(\$Y. n)
- SERIA JUSTO ???:

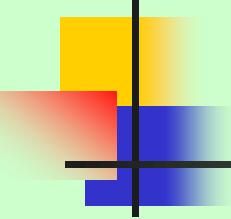
Probabilidade MUITO  
PEQUENA de ocorrência !!!!

USAR A ANÁLISE PROBABILÍSTICA PARA GARANTIR AO CÁLCULO DO CUSTO MÉDIO !!!



# Análise Probabilística

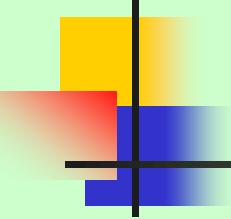
- Análise Probabilística:
  - É o uso da probabilidade na análise do problema;
  - Baseia-se no uso do conhecimento prévio (ou suposições) sobre a distribuição das possíveis ocorrências;
  - O resultado esperado no cálculo do tempo de execução ( ou média do tempo de execução) sobre TODAS as possíveis entradas (seqüências de dados);
- Em nosso problema:
  - devemos garantir que o candidato enviado para a entrevista é em ordem ALEATÓRIA, ou seja, podemos ter  $n!$  perturbações aleatórias com a mesma probabilidade de ocorrência para todas as  $n!$  perturbações;
  - Para que seja verdade, temos que garantir a seleção aleatória dos candidatos para a etapa de entrevista, ou seja, uma RANDOM\_SELECTION (seleção randômica);
  - Com isso, afirma-se que a probabilidade do candidato  $i$  ser designado para a entrevista é  $1/i$ , logo no total teremos:
    - $\sum_{i=1}^n (1/i) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n} = \text{Log } n$
    - **Significa que, apesar de entrevistarmos  $n$  candidatos, somente teremos  $\text{Log } n$  contratações em média !!!!!!**



# Análise Probabilística

- Algoritmo CONTRATAÇÃO\_ALEATÓRIA()  
begin  
    Permutação\_Aleatória(lista\_de\_candidatos);  
    melhor\_cand = 0; /\*candidato fictício \*/  
    for i to n /\* n = numero de candidatos \*/  
        Do entrevista candidato\_i ;  
        custo = custo +\$X;  
        if candidato\_i MELHOR melhor\_cand then  
            melhor\_cand = i;  
            contratar i;  
            custo = custo +\$Y;  
        enfIF  
    endFOR  
endCONTRATAÇÃO\_ ALEATÓRIA;

Função que garante  
a permutação aleatória  
de TODOS os possíveis  
candidatos !!!



# Análise Probabilística

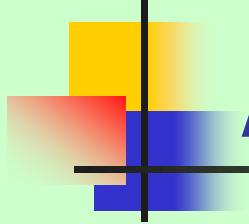
- Algoritmos de Permutação aleatória:

- **Permute\_By\_Sorting( Lista )**

```
begin
  n = comprimento de Lista;
  for i = 1 to n
    Do P[i] = RANDOM(1, n3);
  endFOR
  Ordenar Lista, usando valores de P como chaves de ordenação;
  return (Lista)
end_Permute_By_Sorting;
```

- **Randomize\_in\_Place( Lista )**

```
begin
  n = comprimento de Lista;
  for i = 1 to n
    Do troca Lista[i] ↔ Lista[ RANDOM(i, n) ];
  endFOR
  return (Lista)
end_Randomize_in_Place;
```

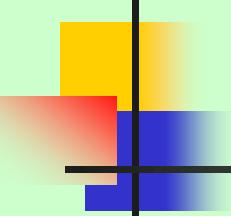


# Análise Probabilística

- **Permute\_By\_Cyclic( Lista )**

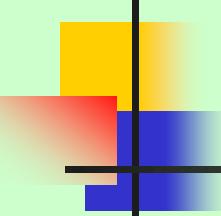
```
begin
desloca = RANDOM(1, n);
n = comprimento de Lista;
for i = 1 to n
    novo_desloca = i + desloca;
    if novo_desloca > n then
        novo_desloca = novo_desloca - n;
    endIF
    Lista_nova [novo_desloca] = Lista[ i ];
endFOR;
return( Lista_nova );
end_Permute_By_Cyclic;
```

**OBS: GERA PERMUTAÇÕES UNIFORMES E ALEATÓRIAS !!**



# Modelagem Orientada a Objetos

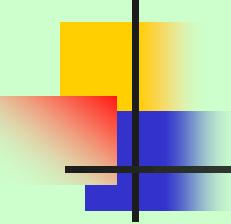
- **Paradigma:** “o mundo pode ser discretizado em objetos (indivíduos), que incorporam informações e comportamentos, permitindo assim que cada indivíduo (objeto) seja capaz de sofrer e promover ações , propiciando uma dinâmica evolutiva ao mundo externo”.
- **Aspectos:**
  - **Identidade:** discretizar objetos;
  - **Classificação:** agrupamento de objetos similares;
  - **Polimorfismo:** operações similares com diferentes ações;
  - **Herança:** compartilhar características entre objetos relacionados;



# Modelagem Orientada a Objetos

## ■ Considerações importantes:

- **Abstração:** focar o essencial, principalmente os aspectos de herança, ignorando-se propriedades transitórias;
- **Encapsulamento** (esconder informações): garante o aumento da aplicabilidade dos objetos (informações associadas);
  - Camada externa do objeto: todos tem acesso;
  - Camada interna do objeto: detalhes internos do objeto;
- **Combinar dados e comportamentos:** gera uma abordagem unificada.
  - Ex: processamento de mamografia (imagem estática) x de cine-ecocardiograma (imagem dinâmica);
- **Sharing:** objeto pode ser usado em várias aplicações diferentes, permitindo múltiplas visões de um mesmo objeto;
  - Ex: armazenamento e display de mamografia (imagem estática) x de cine-ecocardiograma (imagem dinâmica);
- **Não procedural:** descreve-se o que o objeto é (foca a informação), e nunca descrever de como é usado (não focar funcionalidade);
- **Sinergia:** o objeto poderá ser único ou em grupo, a partir de uma integração (garante a reusabilidade);

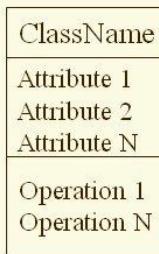


# Modelagem Orientada a Objetos

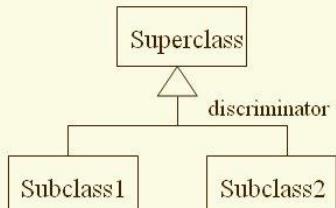
- Modelagem OO:

**Classe:** grupo de objetos com **propriedades** (atributos) similares, **comportamentos** (operações) comuns, **relacionamentos** comuns com outros objetos e com **semântica** comum;

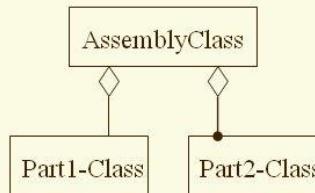
**Instancias:** é o objeto individual, com suas propriedades, comportamentos e relacionamentos individualmente caracterizados;



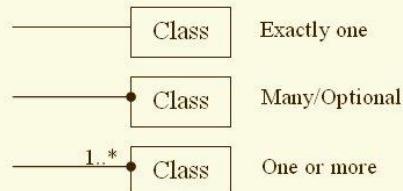
Basic class model includes name, attributes, & operations



Inheritance



– Aggregation

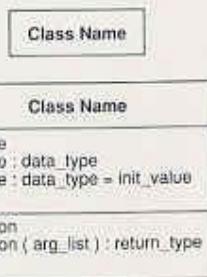


– Multiplicity of Associations

# Modelagem Orientada a Objetos

## Object Model Notation Basic Concepts

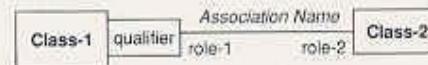
### Class:



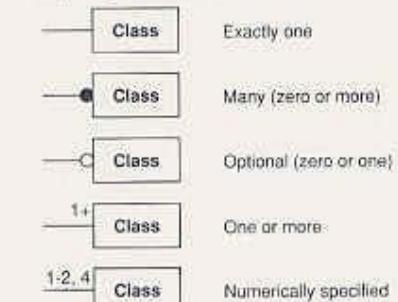
### Association:



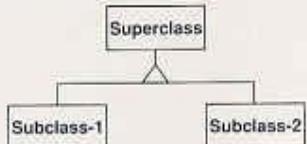
### Qualified Association:



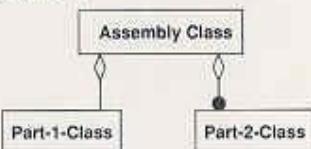
### Multiplicity of Associations:



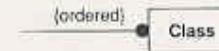
### Generalization (Inheritance):



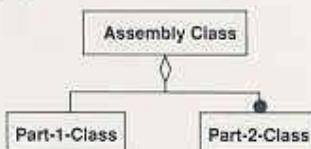
### Aggregation:



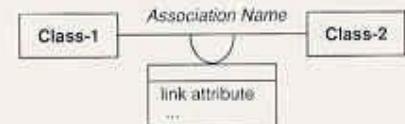
### Ordering:



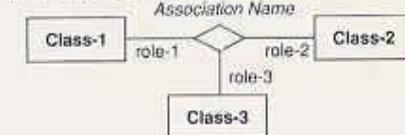
### Aggregation (alternate form):



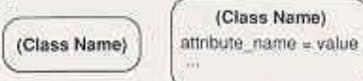
### Link Attribute:



### Ternary Association:



### Object Instances:



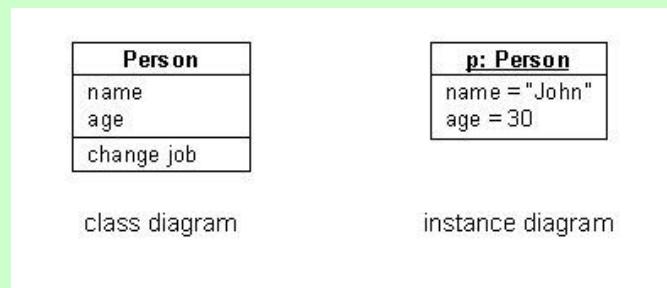
### Instantiation Relationship:



# Modelagem Orientada a Objetos

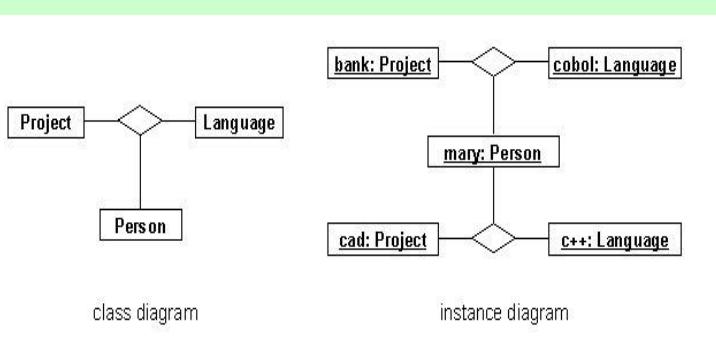
## ■ Bases do modelo OO:

### ■ Classes e Instâncias:

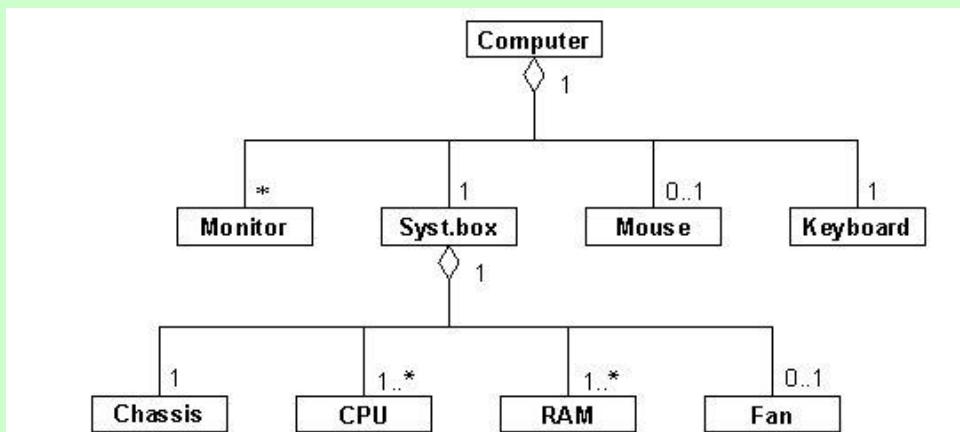


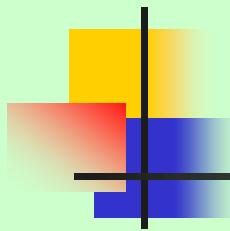
### ■ Associações:

definição de uso, ter, etc



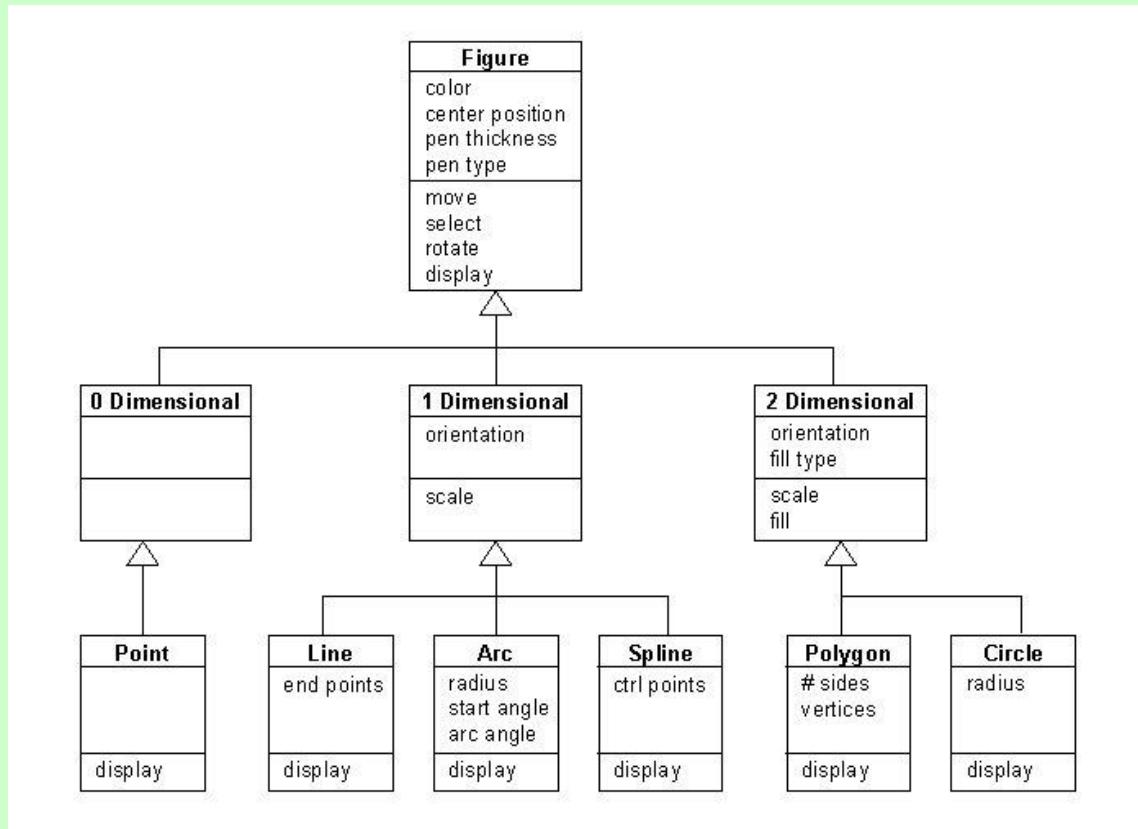
### ■ Agregações: relações que definem componentização



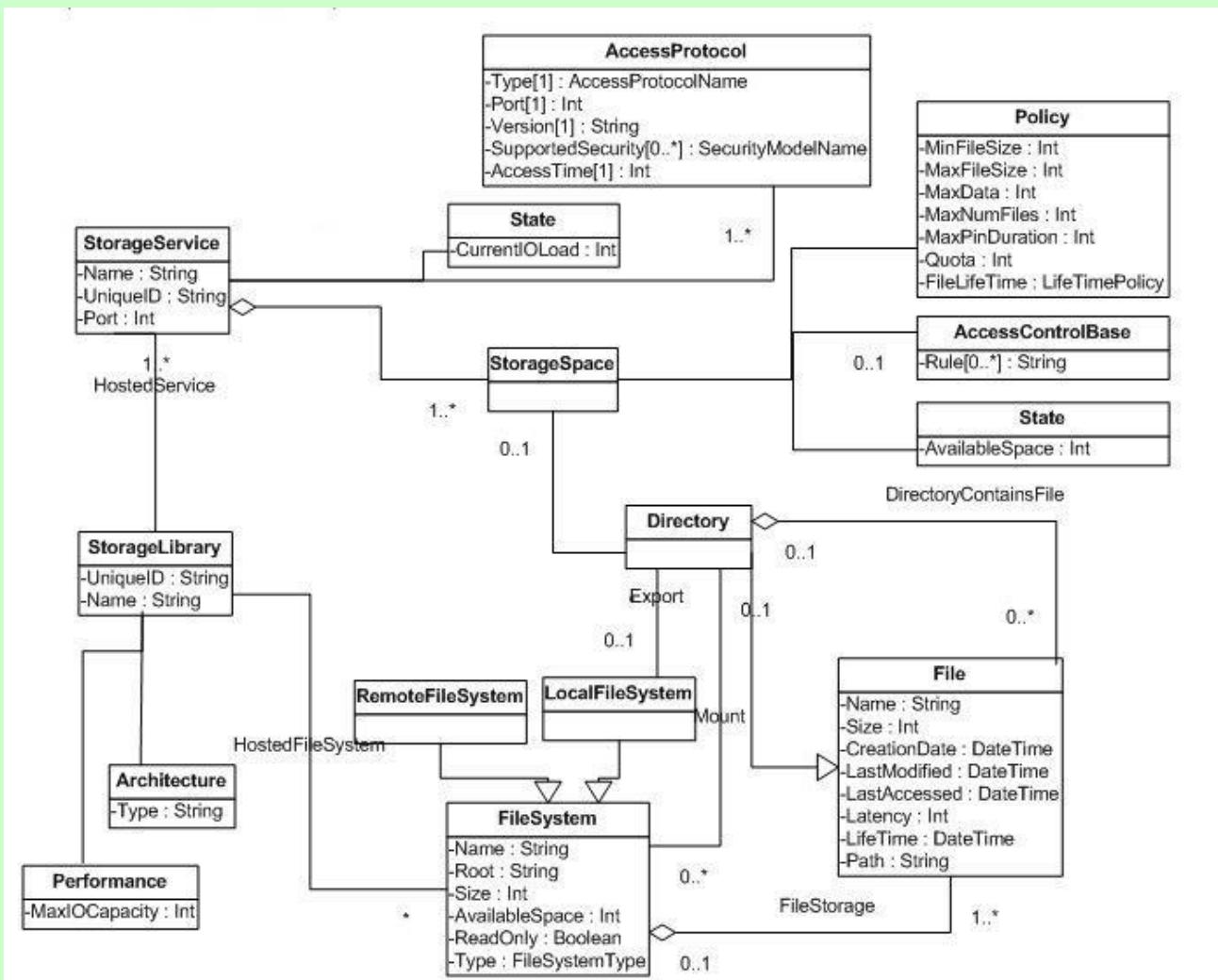


# Modelagem Orientada a Objetos

- **Generalização:** definição de herança;



# Modelagem Orientada a Objetos



# Modelagem Orientada a Objetos

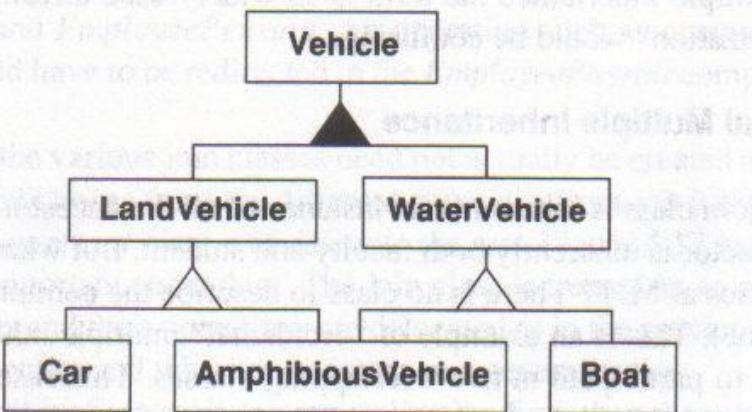
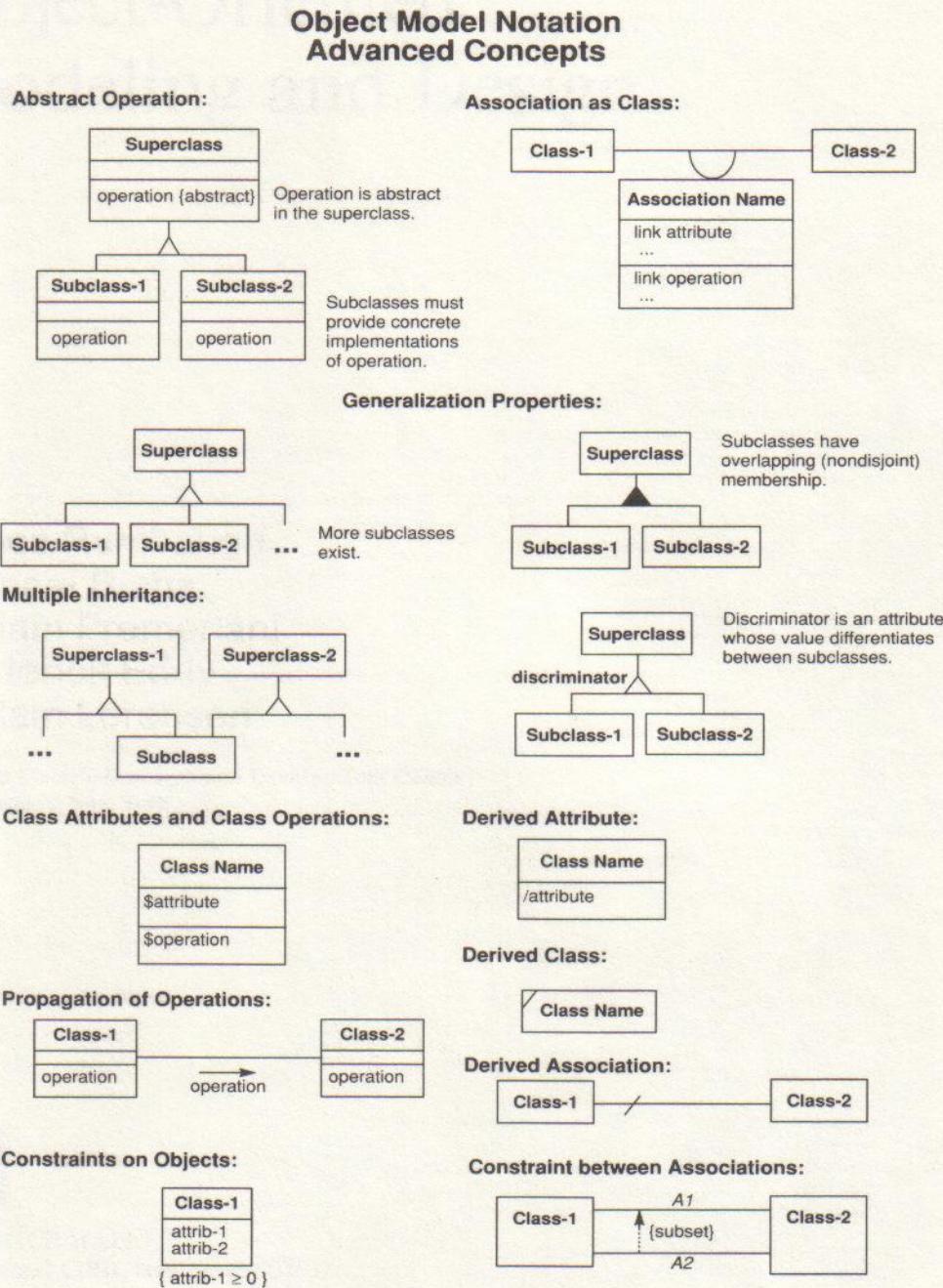
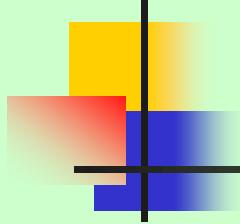


Figure 4.8 Multiple inheritance from overlapping classes





# Modelagem Orientada a Objetos

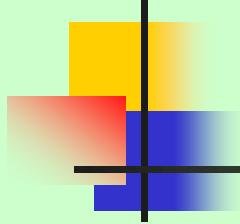
- Metodologia OMT (Object Modelling Technique – James Rumbaugh):

1. Análise:

1. Reconhecer o problema;
2. Modelagem da situação atual do mundo;
3. O que o sistema deverá fazer ??? (funcionalidades básicas);
4. Como fazer ???

2. Projeto do Sistema:

1. Definir arquitetura conceitual;
2. Especificar subsistemas;
3. Escolher a estratégica logística;



# Modelagem Orientada a Objetos

- Metodologia OMT (Object Modelling Technique – James Rumbaugh):
  - 3. Projeto do Objeto:
    - 1. Construir o modelo do objeto baseado na Análise (etapa 1);
    - 2. Evitar detalhes implementacionais;
    - 3. Especificar o domínio de aplicação;
    - 4. Especificar o domínio computacional;
  - 4. Implementação:
    - 1. Traduzir em linguagem de programação as classes de objetos e seus relacionamentos analisados;
    - 2. Especificar estruturação de dados;
    - 3. Especificar ferramentas computacionais e aplicativos;

# Modelagem Orientada a Objetos

## ■ Tipos de Modelos OMT:

### ■ Modelo do Objeto:

- Descreve a estrutura estática do objeto e seus relacionamentos;
- Baseia-se no diagrama de objetos (objetos + relacionamentos);

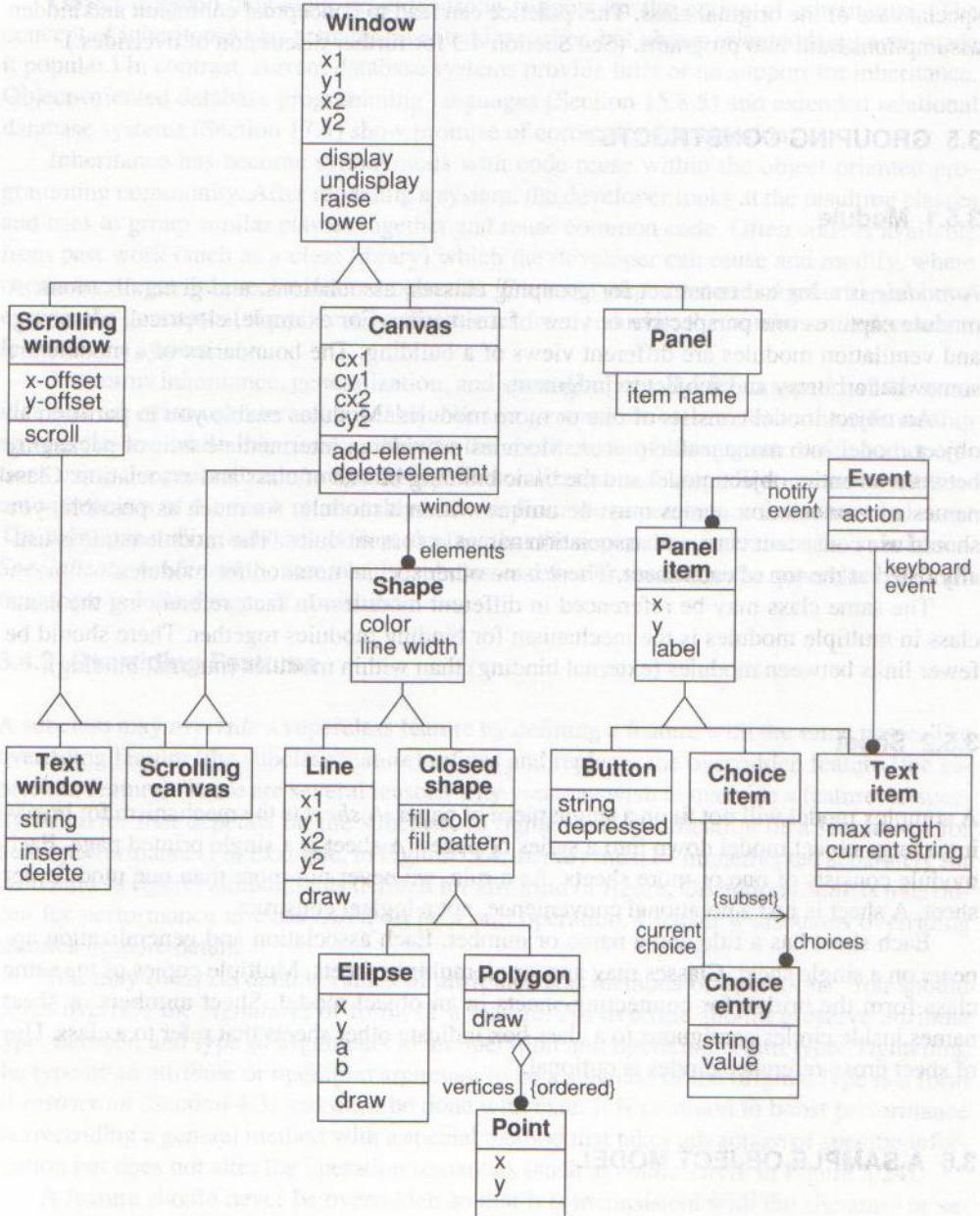


Figure 3.25 Object model of windowing system

# Modelagem Orientada a Objetos

- Tipos de Modelos OMT:

- Modelo Dinâmico:

- Descreve os aspectos que se alteram com o tempo;
    - Baseia-se no diagrama de estados (estados + transições + eventos);

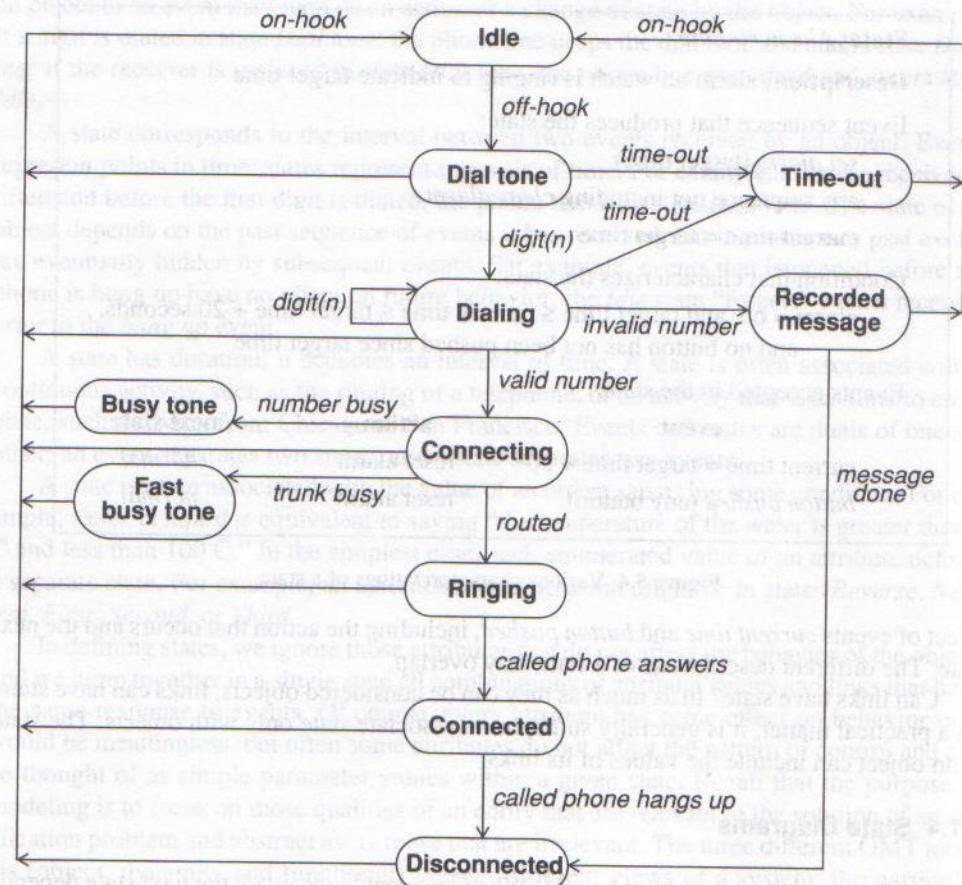


Figure 5.5 State diagram for phone line

# Modelagem Orientada a Objetos

- Tipos de Modelos OMT:

- Modelo Funcional:

- Descreve as transformações dos valores associados aos dados;
    - Baseia-se no diagrama de fluxo de dados (processos + fluxo de informações);

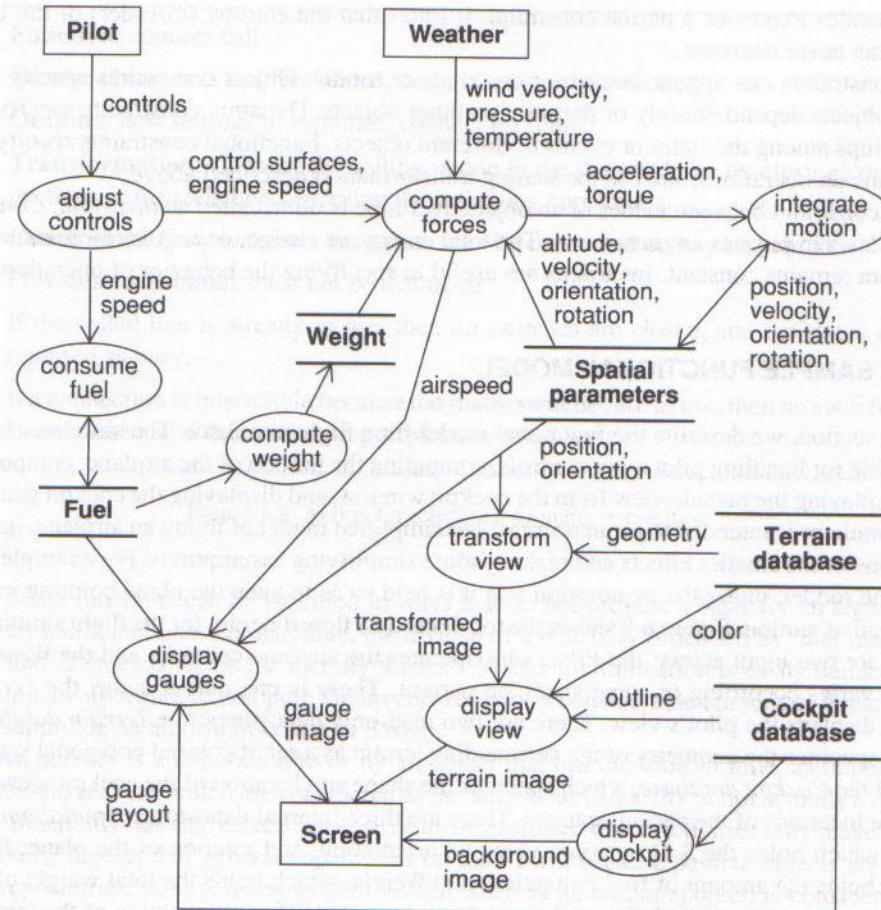
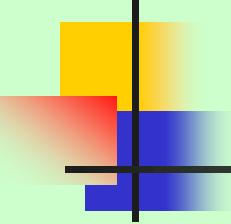


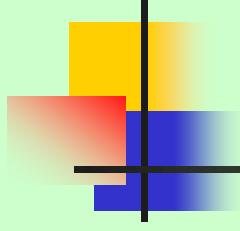
Figure 6.9 Functional model of flight simulator



# Modelagem Orientada a Objetos

- **O que é um Objeto implementacionalmente?**
  - Um objeto é um componente de programa que sabe como agir sobre certas ações e interagir com outras porções de programa
- **Exemplo:** Medieval video game;
- **Análise (conhecer o problema):**
  - **2 classes:** player e monster;
  - **Player atributos:** health, strength, agility, weapon e armor;
  - **Player ações:** move through a maze; attack a monster e pick up treasure;
  - **Monster atributos:** health, skin thickness, claws tail spikes;
  - **Monster ações:** move, attack player with claws, attack player with tail;

<p>Player Object::: data:   health   strength   agility   type of weapon   type of armor actions:   move   attack monster   get treasure END;</p>	<p>Monster Object:: data:   health   skin thickness   claws   tail spikes actions:   move   attack with claws   attack with tail END;</p>
---	---



# Modelagem Orientada a Objetos

- Instâncias:

Player Instance #1:

data:

```
health = 16  
strength = 12  
agility = 14  
type of weapon = "mace"  
type of armor = "leather"  
END;
```

Monster Instance #1:

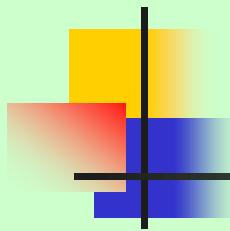
data:

```
health = 21  
skin thickness = 20  
claws = "sharp"  
tail spikes = "razor sharp"  
END;
```

Monster Instance#2:

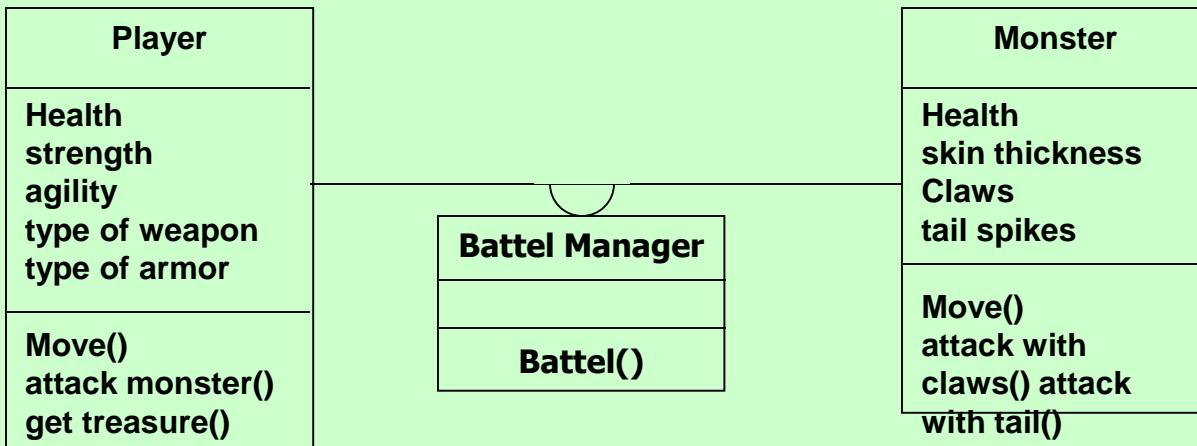
data:

```
health = 9  
skin thickness = 5  
claws = "dull"  
tail spikes = "quite dull"  
END;
```



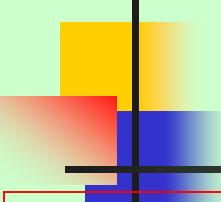
# Modelagem Orientada a Objetos

- Modelo OO (conceitual):



```
Class Battel_Manager:
{
    void Battle();
};

...
...
Battle_Manager:: Battle( Player play1,
                         Monster mont1)
{
    turn = PLAYER;
    while (( play1.health > 0) AND
           (mont1. health > 0))
    {
        if (turn == PLAYER)
        {
            player attack's monster;
            turn = MONSTER;
        }
        else
        {
            monster attack's player
            turn = PLAYER;
        }
    }
} // END FUNCTION
```



# Modelagem Orientada a Objetos

## Declarar Classe:

```
class Player
{
    int health;
    int strength;
    int agility;
    void move();
    void attackMonster();
    void getTreasure();
};

...
void Player::move()
{
    //function body goes here
}
void Player::getTreasure()
{
    health++;
}
```

## Instanciar Classe:

```
Player blueHat;
Player redHat;
Player greenHat;
Player yellowHat;
```

## Usar uma função de uma classe:

```
greenHat.attackMonster();
greenHat.move();
greenHat.getTreasure();
```

# Modelagem Orientada a Objetos

## Construtor:

```
Player::Player()  
{  
    strength = 10;  
    agility = 10;  
    health = 10;  
}
```

## Construtor:

```
Player::Player( int s , int a)  
{  
    strength = s;  
    agility = a;  
    health = 10;  
}
```

## Destrutor:

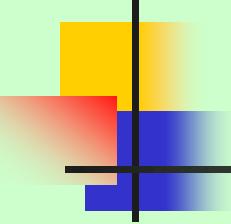
```
Player::~Player()  
{  
    strength = 0;  
    agility = 0;  
    health = 0;  
}
```

## Declarar Classe:

```
class Player  
{  
    int health;  
    int strength;  
    int agility;  
    void move();  
    void attackMonster();  
    void getTreasure();  
    Player();  
    Player (int s , int a);  
    ~Player();  
};  
...  
void Player::move()  
{  
    //function body goes here  
}  
void Player::getTreasure()  
{  
    health++;  
}
```

## Instanciar Classe:

```
Player blueHat (2, 6);  
Player redHat;  
Player greenHat (10, 11);  
Player yellowHat;
```



# Modelagem Orientada a Objetos

## Proteger Informações privativas:

```
class Player
{
    Private:
        int health;
        int strength;
        int agility;
    Public:
        void move();
        void attackMonster();
        void getTreasure();
};
```

## Função In-Line:

```
class Player
{
    int health;
    int strength;
    int agility;
    void move();
    void attackMonster();
    void getTreasure() { health++; } ← Evitar !!
};

...
void Player::move()
{
    //function body goes here
}
```

# Modelagem Orientada a Objetos

## Modelo Dinâmico

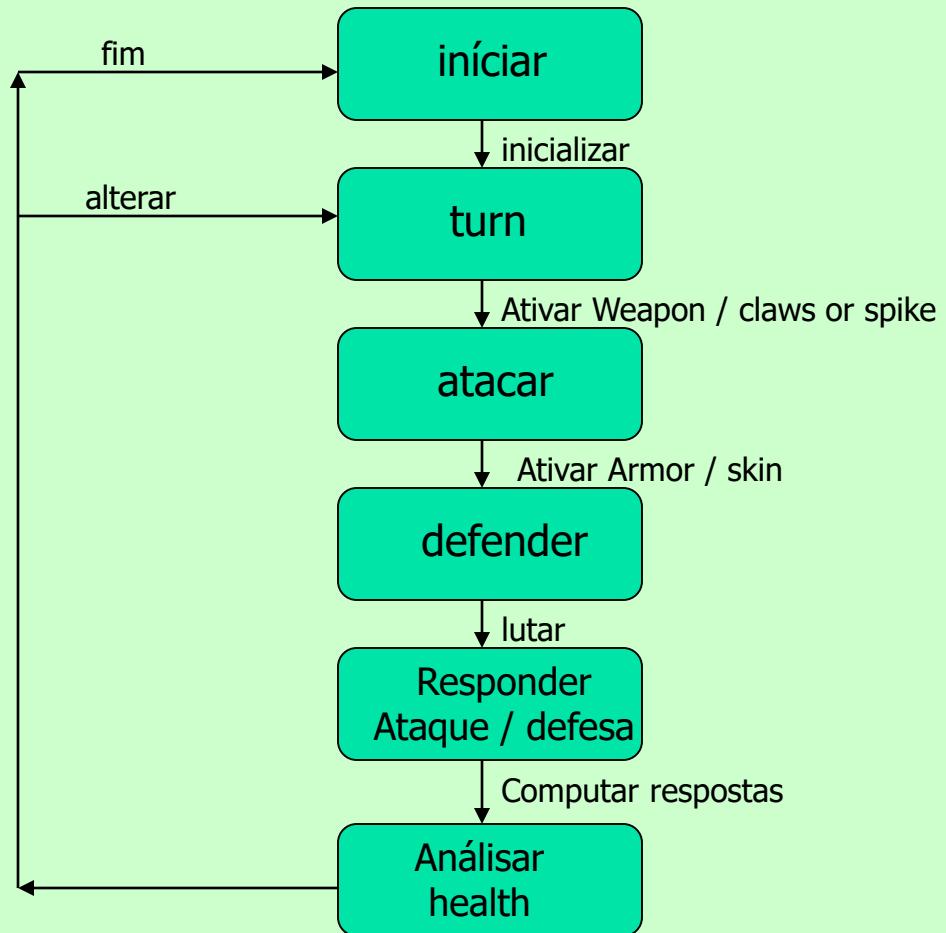
(diagrama de estados - nomenclatura):

estados

Transição:

Evento:

Evento 1



# Modelagem Orientada a Objetos

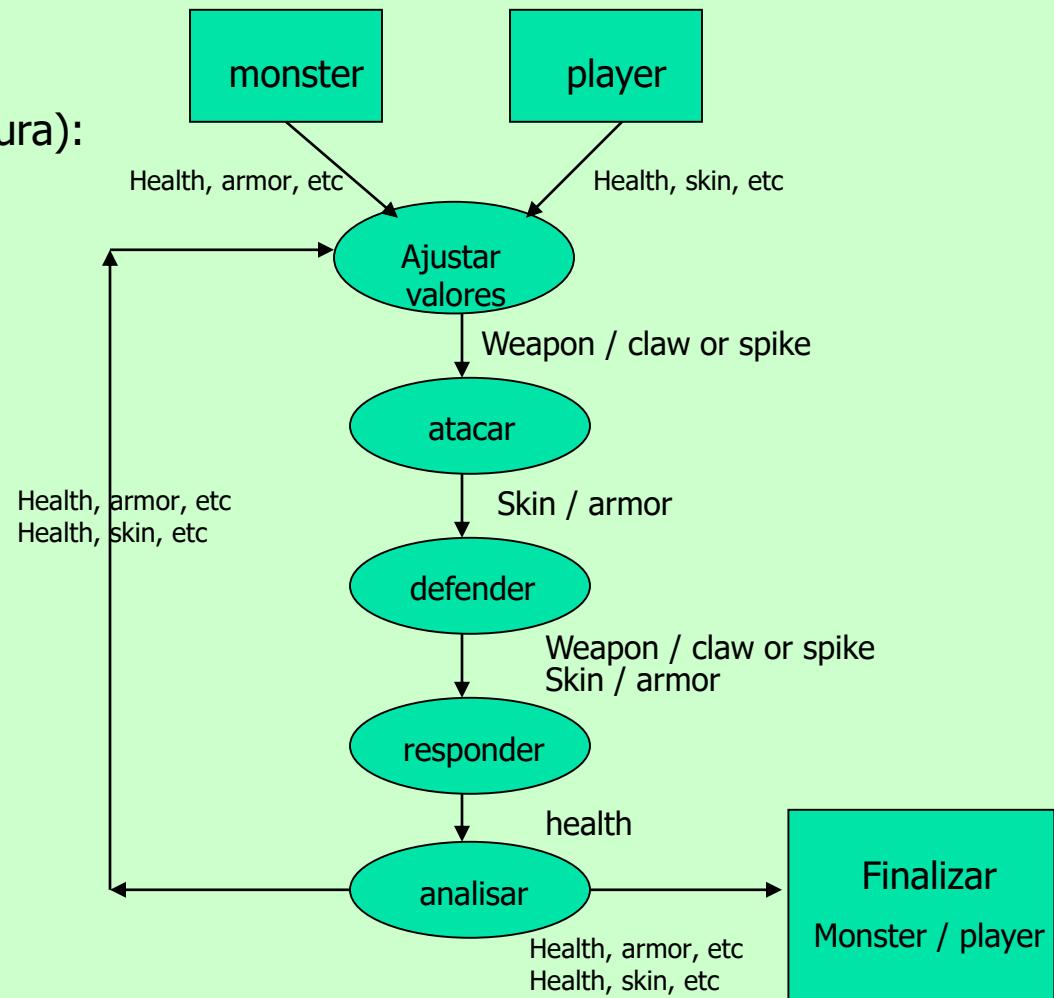
## Modelo Funcional

(diagrama de fluxo de dados - nomenclatura):



Fluxo de informações:

Dados: Informação 1



```
// classes example
#include <iostream.h>

class CRectangle {
    int x, y;
public:
    void set_values (int,int);
    int area (void) {return (x*y);}
};

void CRectangle::set_values (int a,
int b) {
    x = a;
    y = b;
}

int main () {
    CRectangle rect;
    rect.set_values (3,4);
    cout << "area: " << rect.area();
}
```

```
area: 12
```

```
// class example
#include <iostream.h>

class CRectangle {
    int x, y;
public:
    void set_values (int,int);
    int area (void) {return (x*y);}
};

void CRectangle::set_values (int a,
int b) {
    x = a;
    y = b;
}

int main () {
```

```
rect area: 12
rectb area: 30
```

```
CRectangle rect, rectb;
rect.set_values (3,4);
rectb.set_values (5,6);
cout << "rect area: " <<
rect.area() << endl;
cout << "rectb area: " <<
rectb.area() << endl;
}
```

## Constructors and destructors

```
// classes example
#include <iostream.h>

class CRectangle {
    int width, height;
public:
    CRectangle (int,int);
    int area (void) {return
(width*height);}
};

CRectangle::CRectangle (int a, int
b) {
    width = a;
    height = b;
}

int main () {
    CRectangle rect (3,4);
    CRectangle rectb (5,6);
    cout << "rect area: " <<
rect.area() << endl;
    cout << "rectb area: " <<
rectb.area() << endl;
}
```

```
rect area: 12
rectb area: 30
```

```
// overloading class constructors
#include <iostream.h>

class CRectangle {
    int width, height;
public:
    CRectangle ();
    CRectangle (int,int);
    int area (void) {return
(width*height);}
};

CRectangle::CRectangle () {
    width = 5;
    height = 5;
}

CRectangle::CRectangle (int a, int
b) {
    width = a;
    height = b;
}

int main () {
    CRectangle rect (3,4);
    CRectangle rectb;
    cout << "rect area: " <<
rect.area() << endl;
    cout << "rectb area: " <<
rectb.area() << endl;
}
```

```
rect area: 12
rectb area: 25
```

```
// example on constructors and  
// destructors  
#include <iostream.h>  
  
class CRectangle {  
    int *width, *height;  
public:  
    CRectangle (int,int);  
    ~CRectangle ();  
    int area (void) {return (*width  
    * *height);} };  
  
CRectangle::CRectangle (int a, int  
b) {  
    width = new int;  
    height = new int;  
    *width = a;  
    *height = b;  
}  
  
CRectangle::~CRectangle () {  
    delete width;  
    delete height;  
}  
  
int main () {  
    CRectangle rect (3,4), rectb  
(5,6);  
    cout << "rect area: " <<  
rect.area() << endl;  
    cout << "rectb area: " <<  
rectb.area() << endl;  
    return 0;  
}
```

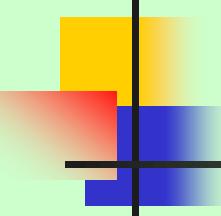
```
rect area: 12  
rectb area: 30
```

# Pointers to classes

```
CRectangle * prect;  
"  
// pointer to classes example  
#include <iostream.h>  
  
class CRRectangle {  
    int width, height;  
public:  
    void set_values (int, int);  
    int area (void) {return (width  
* height);} };  
  
void CRRectangle::set_values (int a,  
int b) {  
    width = a;  
    height = b; }  
  
int main () {  
    CRRectangle a, *b, *c;  
    CRRectangle * d = new CRRectangle[2];  
    b= new CRRectangle;  
    c= &a;  
    a.set_values (1,2);  
    b->set_values (3,4);  
    d->set_values (5,6);  
    d[1].set_values (7,8);  
    cout << "a area: " << a.area() << endl;  
    cout << "*b area: " << b->area() << endl;  
    cout << "*c area: " << c->area() << endl;  
    cout << "d[0] area: " << d[0].area() << endl;  
    cout << "d[1] area: " << d[1].area() << endl;  
    return 0; }
```

```
a area: 2  
*b area: 12  
*c area: 2  
d[0] area: 30  
d[1] area: 56
```

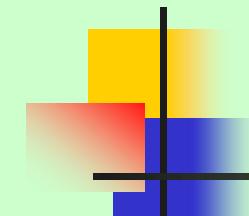
- \*x can be read: pointed by x
- &x can be read: address of x
- x.y can be read: member y of object x
- (\*x).y can be read: member y of object pointed by x
- x->y can be read: member y of object pointed by x (equivalent to the previous one)
- x[0] can be read: first object pointed by x
- x[1] can be read: second object pointed by x
- x[n] can be read: (n+1)<sup>th</sup> object pointed by x



# Estrutura de Dados

- **Tipos de Dados:**

- Linguagem de programação = linguagem descritiva usada para definir o modelo físico de um algoritmo;
- A linguagem de programação faz uso da especificação de dados, ou seja, a declaração do tipo de dado atribuído a uma variável necessária em um algoritmo proposto (a solução abstrata de um problema);
- Declarar o tipo de uma variável consistem em delimitar um domínio de valores válidos, além da especificação de operações que, quando aplicadas à variável, gera um resultado também dito válido (dentro do domínio previamente definido);



# Estrutura de Dados

- Para toda a linguagem de programação os tipos de dados podem ser clarificados como:
  - Primitivos: tipos de dados básicos ou pré-definidos;
    - Exs: Int, Float, Real, Char, Logic(&, or)
  - Construtivos: faz uso de mecanismos especiais para a criação de novos tipos de dados
    - Exs: vetor (array)=> dado homogêneo => Vet[23, 44, 66, 87] = matriz 1D

Matriz => dado homogêneo n dimensional => M<sub>1D</sub>[x], M<sub>2D</sub>[x][y], M<sub>3D</sub>[x][y][z], M<sub>4D</sub>[x][y][z][t];

registro=> dado heterogêneo=> set define structure{

Char nome[1...30]

Int Matricula } Funcionário;

seqüência => dado homogêneo=> A='casa branca';

referência(pointers); =>PTR\_Funcionario func;

func->nome = 'joão';

func->matricula=12345;

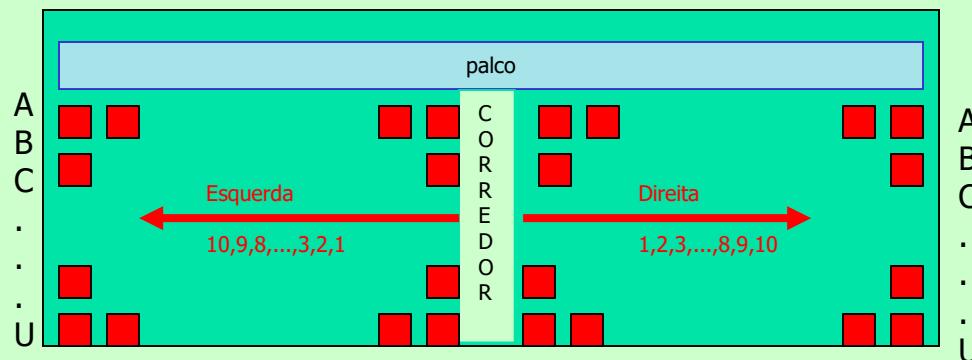
mat = func.matricula /\* A = 12345 \*/

enumeração=> Enum {janeiro=1, fevereiro=2, ....,dezembro=12} mês;

mês=dezembro =12;

# Estrutura de Dados

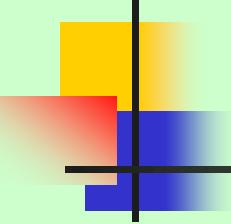
- Exercício: Sabendo-se que um teatro possui 400 lugares distribuídos em um numero igual de filas na transversal como na longitudinal do teatro.
- Descreva um algoritmo que seja capaz de identificar a exata localização de um assento do teatro por um código de localização impresso no bilhete como por exemplo:
  - Fila A esquerda10
  - Fila J direita 5
  - Ou seja, a seção transversal do teatro é identificada por letras seqüenciais (A,B,C,D,...), e a seção longitudinal por números respeitando-se os lado direito ou esquerdo de um corredor central ao teatro.



# Estrutura de Dados

- Logic TEATRO[20][20]; /\* lugar ocupado =1; lugar não ocupado = 0 \*/
- Enum{ A = 1, B=2,C=3, .....T=19, U=20} LONGITUDINAL;
- Enum{esquerda = -1 , direita =1}LADO;
- Set define Structure{ LADO local;  
                        Int      numero;    } TRANSVERSAL;
- Set define Structure{ LONGITUDINAL long;  
                           TRANSVERSAL  trans;    } BILHETE;
- Int corredor;
- LOGIC achado;
  
- LOGIC Localizar\_Assento( BILHETE ticket )  
begin;  
    if (bilhete->trans.local == -1) then corredor=9;  
    if (bilhete->trans.local == +1) then corredor=10;  
    return( TEATRO [bilhete.long] [corredor + bilhete->trans.local \*  
   bilhete->trans.numero - bilhete->trans.local ];  
endProcedure;

O(1) →  
 $\Omega(1)$   
Ótimo !!



# Estrutura de Dados

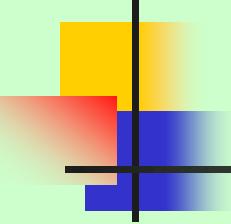
- Descreva um algoritmo que seja capaz analisar a taxa de ocupação para ao mesmo teatro:

```
float Ocupação ()  
begin;  
    ocupação = 0;  
    for i = 1 to 20 /* A,B,C,...,T,U */  
        for J = 1 to 10 /* numeração de assentos */  
  
            local == -1; /* verifica esquerdo */  
            corredor=9;  
            if ( TEATRO [i] [corredor + local * j - local ] == TRUE ) then    ocupação++;  
  
            local == +1; /* verifica direito */  
            corredor=10;  
            if ( TEATRO [i] [corredor + local * j - local ] == TRUE ) then    ocupação++;  
            endFOR;  
        endFOR;  
    return( ocupação/400 );  
endProcedure;
```

$O(n^2)$

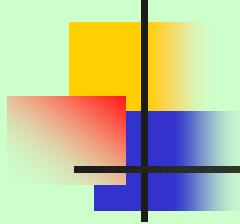
$\Omega(n^2)$

Ótimo !!!



# Estrutura de Dados - Seqüências

- Seqüências:
  - Muitas vezes temos que manipular dados tais como nomes, frases e textos, onde não se tem a previsão exata da quantidade de informação;
  - Para tal, podemos fazer uso de uma estrutura de dados chamada CADEIA ;
  - ex: CADEIA :: seq char;  
palavra :: CADEIA  
palavra = 'emigrar';  
palavra= 'casa';  
palavra = ``;



# Estrutura de Dados - Seqüências

- **Operações válidas:**

1. Concatenação entre 2 CADEIAS gerando uma nova CADEIA (CONC):

EX: A = 'EMI' ; B = 'GRAR' , C = CONC( A , B ) = 'EMIGRAR'

2. Atribuição do 1º item da CADEIA (PRIN):

EX: A = 'EMIGRAR' ; B = PRIN( A ) = 'E'

3. Eliminação do 1º item da CADEIA, ou seja, o resto de uma CADEIA (CONT):

EX: A = 'EMIGRAR' ; B = CONT( A ) = 'MIGRAR'

# Estrutura de Dados - Seqüências

- Exercício: Calcule o comprimento de uma cadeia de informações:

```
int Comprimento_CADEIA ( cad)
```

```
begin
```

```
    i = 0;
```

```
    while cad != ``
```

```
        cad = CONT( cad);
```

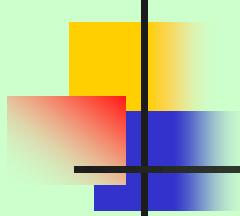
```
        i++;
```

```
    endWHILE;
```

```
    return( i );
```

```
endComprimento_CADEIA;
```

O(n)  
 $\Omega(n)$   
Ótimo !!



# Estrutura de Dados - Seqüências

- **Exercício:** Descreva um algoritmo que extraia uma parte de uma CADEIA dada, ou seja uma SUBCADEIA. Use as seguintes informações:
  1. P é a posição de início da SUBCADEIA;
  2. M é o comprimento da SUBCADEIA

# Estrutura de Dados - Seqüências

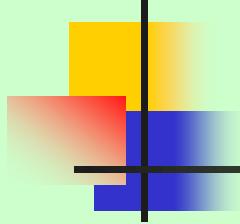
**Exercício:** Descreva um algoritmo que extraia uma parte de uma CADEIA dada, ou seja uma SUBCADEIA. Use as seguintes informações:

1. P é a posição de início da SUBCADEIA;
2. M é o comprimento da SUBCADEIA

```
CADEIA Extrair_CADEIA ( CADEIA cad, INT p , INT m )
    begin
        CADEIA extração;
        extração = '';
        for i = 1 to (p-1)
            cad = CONT(cad);
        endFOR;
        for i = 1 to m
            extração = CONC( extração , PRIN( cad ) );
            cad = CONT(cad);
        endFOR
        return(extração);
    end_Extrair_CADEIA;
```

$$O(p + m) = O(n)$$

$\Omega(n)$   
Ótimo !!



# Estrutura de Dados - Seqüências

- **Exercício:** Descreva um algoritmo que, dado duas CADEIAS cad1 e cad2, forneça quantas vezes cad2 ocorre em cad1, ou seja, quantas vezes cad2 é SUBCADEIA em cad1:

# Estrutura de Dados - Seqüências

- **Exercício:** Descreva um algoritmo que, dado duas CADEIAS cad1 e cad2, forneça quantas vezes cad2 ocorre em cad1, ou seja, quantas vezes cad2 é SUBCADEIA em cad1:

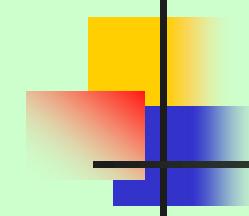
```
INT Verificação_Subcadeias (CADEIA cad1, cad2 )
begin
CADEIA original, apoio;
INT subcadeia = 0;
original = cad2;
apoio = cad1;
while cad1 != ''
    while ( PRIN ( cad1 ) = PRIN ( cad2 ) )
        cad1= CONT(cad1);
        cad2= CONT(cad2);
    endWHILE;
    if (cad2 = '' ) then
        subcadeia++;
    endIF
    cad2 = original;
    cad1 = CONT ( apoio );
    apoio = cad1;
endWHILE;
return(subcadeia);
end_ Verificação_Subcadeias;
```

$O(p)$   
Tamanho  
de cad2

$O(m)$   
Tamanho  
De cad1

$O(p * m) = O(n^2)$   
 $\Omega(n)$   
Não Ótimo !!  
(melhor caso)

$\Omega(n^2)$   
Ótimo !!  
(pior caso)



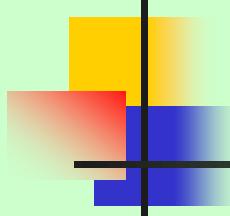
# Estrutura de Dados - array

## ■ **Array ou Arranjos:**

- Consiste em um meio de estruturação de dados;
- NÂO é somente um conjunto consecutivo de localizações na memória, apesar de a nível físico (implementacional) um array ser definido a partir do uso de memória contínua, ou quase contínua;
- Um array é um conjunto de pares de índices e de seus respectivos valores, ou seja , matematicamente um array pode ser definido como um mapeamento entre índices e valores associados;
- Logo: array = { {índice 1 , valor 1},  
                 {índice 2 , valor 2},  
                 {índice 3 , valor 3},  
                 ...         ...  
                 {índice n , valor n} };

ex: vetor

1	2	3	4	5	6	7
S	T	R	A	U	S	S



## Estrutura de Dados - array

- Exemplo: um array também pode ser usado para representar funções matemáticas e assim garantindo operações entre elas:

$$f(x) = 3x^2 + 2x + 4 + x^4 = \{ \{0, 4\}, \{1,2\}, \{2,3\}, \{3,0\}, \{4,1\} \}$$

0	1	2	3	4	5	6
4	2	3	0	1	0	0

$$f(x) = x^6 + 2x + 6 + 3x^5 = \{ \{0, 6\}, \{1,2\}, \{2,0\}, \{3,0\}, \{4,0\} \}, \{5,3\}, \{6,1\} \}$$

0	1	2	3	4	5	6
6	2	0	0	0	3	1

# Estrutura de Dados - array

- Desenvolva o algoritmo que calcule  $f(x) \cdot g(x)$ , onde  
 $f(x) = x^p + x^{p-1} + x^{p-2} + \dots + x^0$   
 $g(x) = x^m + x^{m-1} + x^{m-2} + \dots + x^0$

MULTIPLICA (M1[p] ,M2[n])

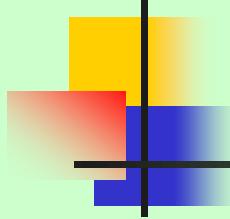
```
begin
  INT Mresp[p+m+1];
  for i = 0 to p
    for j = 0 to m
      Mresp[i+j] = Mresp[i+j] + M1[i] * M2[j];
    endFOR;
  endFOR;
end_MULTIPLICA;
```

$O(p * m) = O(n^2)$

$\Omega(p+m) = \Omega(n)$

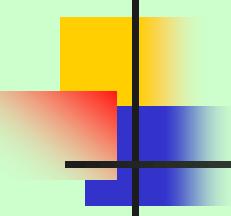
Não Ótimo !!

(mundo matemático)



# Estrutura de Dados - array

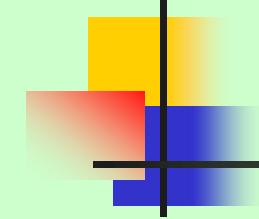
- Na ciência da computação, falar de array consiste em também se especificar as operações pertinentes a essa estrutura de dados, ou seja, é preciso garantir operações capazes de armazenar (store) e de recuperar (retrieve) os valores (informações) associados aos índices;
- **AXIOMA:** para se recuperar o  $i$ -ésimo item de um array  $\mathbf{A}$ , quando  $x$  (informação) já foi armazenado no índice  $i$ . Isso equivale a verificar se os índices  $i$  e  $j$  são iguais, e se for assim, use  $x$  (recuperação com sucesso !!!). Caso contrário, procure o  $i$ -ésimo valor do resto do array  $\mathbf{A}$ .
- **OBS:** note que o AXIOMA garante que elementos em um array respeitam a ordenação, garantindo que índices sejam QUALQUER numero ordenado. Ou seja, os índices NÃO obrigatoriamente serão sempre números inteiro positivos.



# Estrutura de Dados - Listas

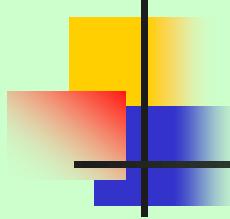
- Listas Lineares

- É uma estrutura de dados que permite representar um conjunto de dados de forma a preservar a relação de ordenação linear entre a informação.
- Consiste em um conjunto de NOS ligados de forma a respeitar um ordem, onde cada NÓ contem um dado primitivo ou composto;
- Em uma lista com n elementos ( $n > 0$ ), a posição relativa de cada elemento é refletida na própria estrutura de dados:
  - $N>0 \Rightarrow X_1 = \text{PRIMEIRO NÓ}$   
 $\Rightarrow X_k = \text{nó precedido pelo nó de ordem } k-1, \text{ e predecessor do nó } k+1;$   
 $\Rightarrow X_n = \text{último nó};$
  - $N=0 \Rightarrow \text{lista vazia};$



# Estrutura de Dados - Listas

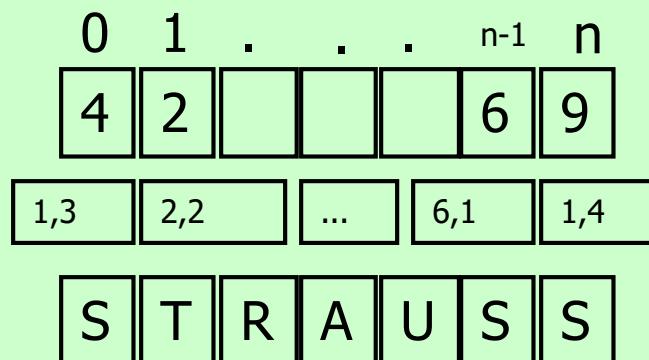
- Tipos de operações:
  1. Acessar o k-éssimo nó da lista para recuperação e/ou armazenagem de dado;
  2. Inserir um nó, anterior ou posterior, ao k-éssimo nó da lista;
  3. Remover o k-éssimo nó;
  4. Concatenar 2 listas;
  5. Determinar o tamanho da lista (numero total de nós);
  6. Localizar o nó (sua ordem) que contem um determinado valor de dado;



# Estrutura de Dados - Listas

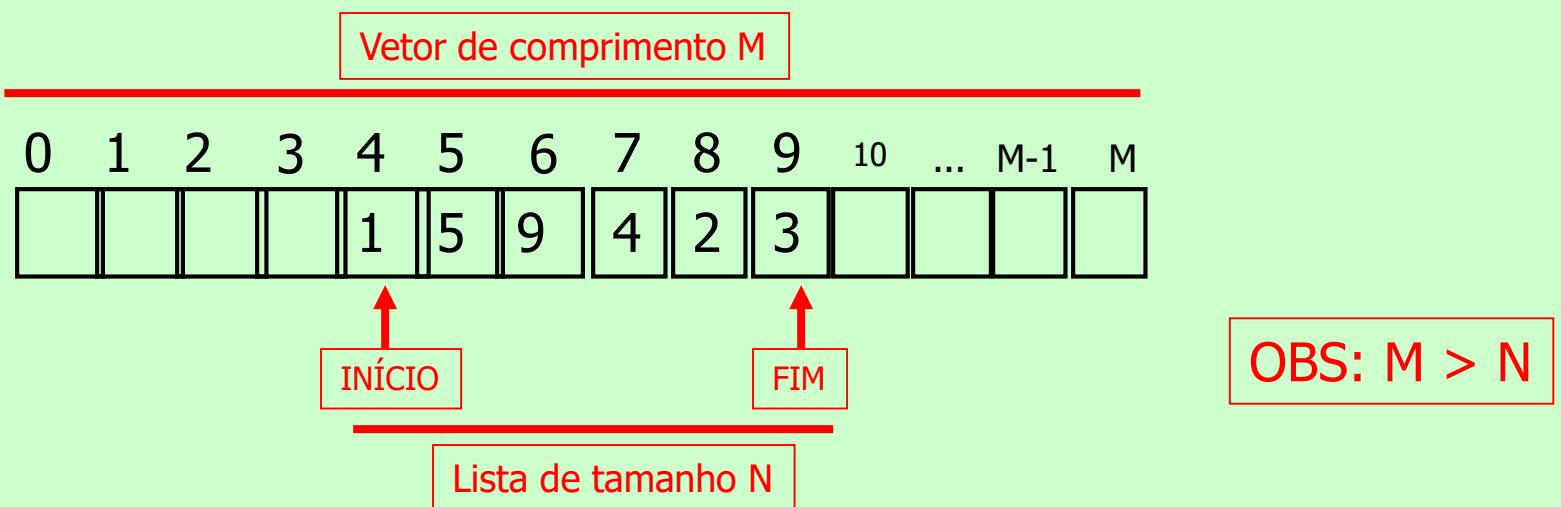
- Formas de Representação:

- **Por continuidade:** explora a sequenciabilidade da memória do computador, de forma que os nós de uma lista são armazenados em endereços contínuos, ou igualmente distanciados um do outro:
  - Ex: vetor => Int    vetor[1...n]  
=> Float vetor[1...n]  
=> Char vetor[1...n]

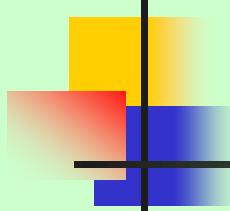


# Estrutura de Dados - Listas

- OBS: OUTRA FORMA DE REPRESENTAÇÃO ONDE A REMOÇÃO/INSERÇÃO PODERÃO SER OTIMIZADAS POR POSSUIREM ESPAÇOS NO INÍCIO E NO FIM DO VETOR QUE DESCREVE A LISTA



```
Set define structure { Int inicio, fim;  
                      Int vetor[1...M] } LISTA;
```



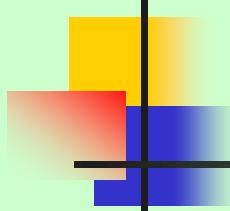
# Estrutura de Dados - Listas

- **Acessar o k-éssimo nó da lista:**

```
ACESSAR-Késsimo(Lista list, Int k)
begin
if (k < list.inicio) or (k > list.fim)
    printf('K esta fora do domínio');
else
    printf('valor = %d', list.vetor[k]);
endif;
end_ACESSAR-Késsimo;
```

- **Alterar o k-éssimo nó da lista:**

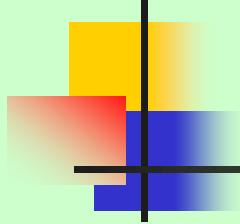
```
ALTERAR-Késsimo(Lista list, Int k, Int valor)
begin
if (k < list.inicio) or (k > list.fim)
    printf('K esta fora do domínio');
else
    list.vetor[k] = valor;
endif;
end_ALTERAR-Késsimo;
```



# Estrutura de Dados - Listas

- **Inserir novo nó, antes do k-éssimo nó da lista:**

```
Inserir_Antes-Késsimo(Lista list, Int k, Int valor)
begin
  if (k < list.inicio) or (k > list.fim)
    printf('K esta fora do domínio');
  else
    for i = list.fim to k step -1
      list.vetor[i+1] = list.vetor[i];
    endfor;
    list.vetor[k] = valor;
  endif;
end_ Inserir_Antes -Késsimo;
```



# Estrutura de Dados - Listas

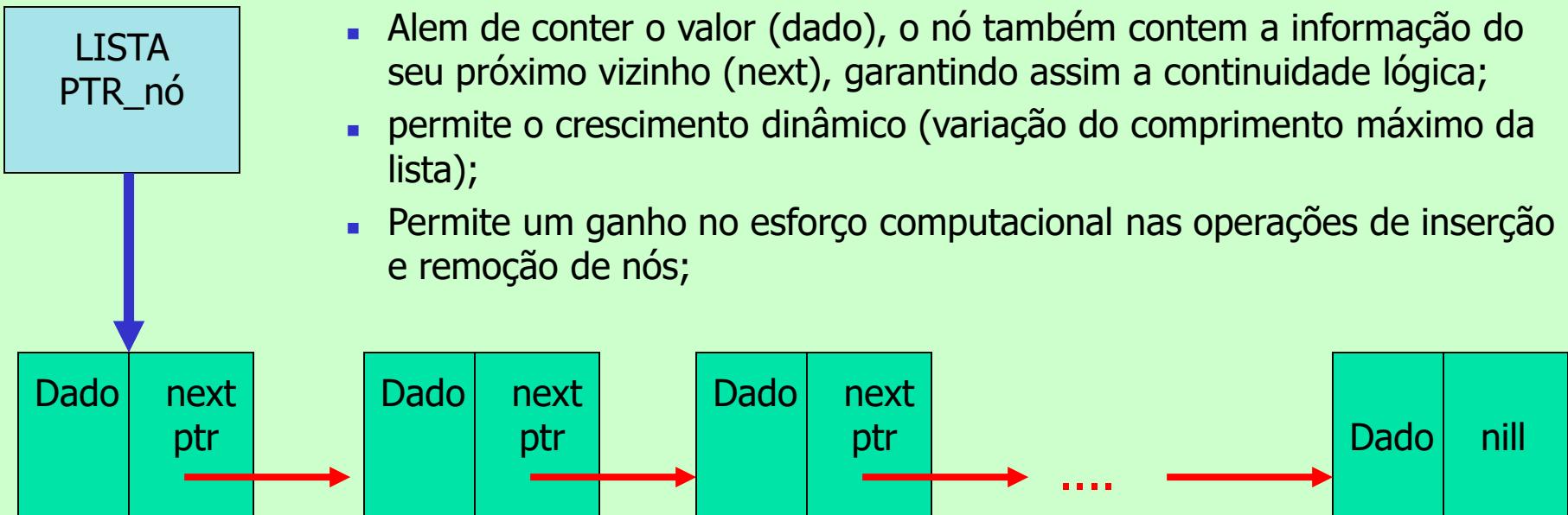
- **Remover o k-éssimo nó da lista:**

```
Remover-Késsimo(Lista list, Int k)
begin
    if (k < list.inicio) or (k > list.fim)
        printf('K esta fora do domínio');
    else
        for i = K to list.fim-1
            list.vetor[i] = list.vetor[i +1];
        endfor;
    endif;
end_ Remover -Késsimo;
```

# Estrutura de Dados - Listas

## ■ Por encadeamento:

- Os nós da lista estão LIGADOS entre si, indicando a relação de ordem entre os nós;
- Além de conter o valor (dado), o nó também contém a informação do seu próximo vizinho (next), garantindo assim a continuidade lógica;
- permite o crescimento dinâmico (variação do comprimento máximo da lista);
- Permite um ganho no esforço computacional nas operações de inserção e remoção de nós;



# Estrutura de Dados - Listas

- Implementacionalmente: Set define structure {

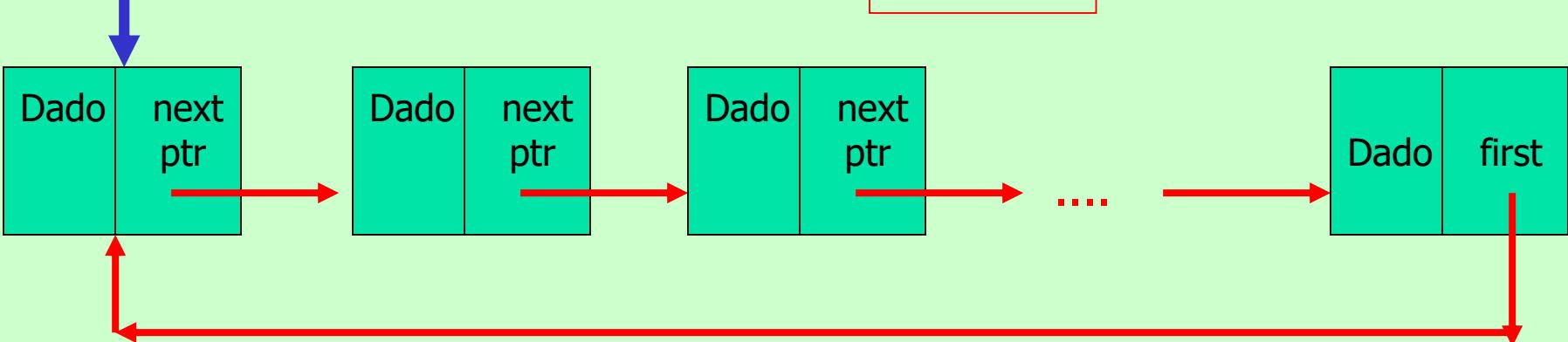
```
    Info dados;  
    PTR_nó next; } Nó;
```

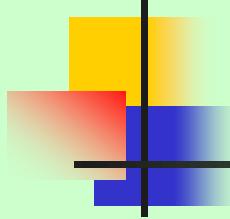
```
Set define structure {  
    PTR_nó nó; } LISTA;
```

```
PTR_LISTA lista;  
PTR_nó nó;  
alloc(nó);  
lista->nó = nó; /* lista vazia */  
nó.dado = valores;  
nó.next = null
```

No.next = no

LISTA\_CIRCULAR  
PTR\_nó





# Estrutura de Dados - Listas

- **Inclusão de novo nó:**

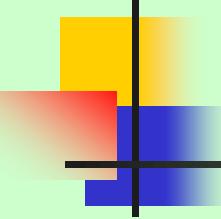
INCLUSÃO(LISTA lista)

```
begin
    PTR_NO novo;
    alloc(novo);
    novo->next = lista.no;
    lista.no = novo;
end;
```

- **Remoção de nó (último inserido):**

Remoção(LISTA lista)

```
begin
    PTR_NO remove;
    remove= lista->no;
    lista.no = lista.no->next;
    free(remove);
end;
```



# Estrutura de Dados - Listas

- **Comprimento de uma lista:**

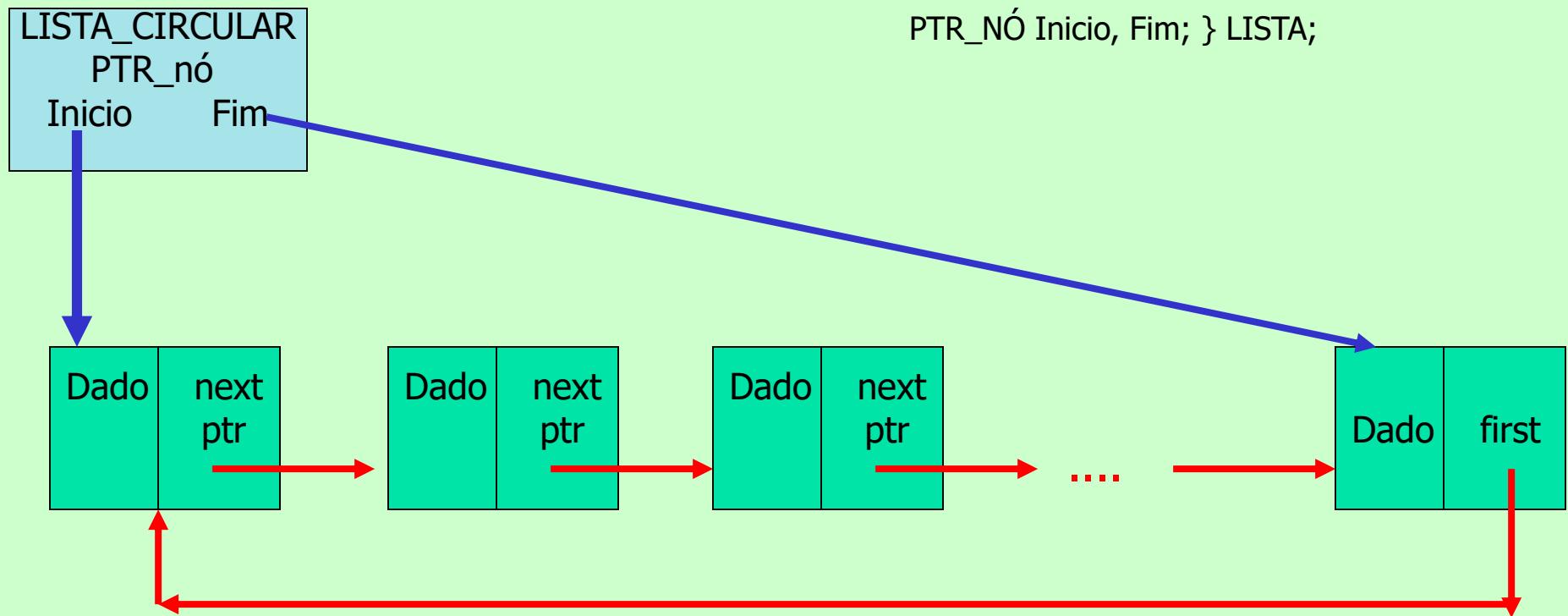
```
COMPRIMENTO(LISTA lista)
begin
    PTR_NO apoio;
    apoio=lista->no;
    cont = 1;
    while (apoio->next != nill) /* while (apoio->next != lista.no) */
        apoio = apoio->next;
        cont++;
    enWHILE
    print(cont);
end;
```

- **Remoção do último nó de uma lista (primeiro que foi inserido):**

```
Remoção do Último (LISTA lista)
begin
    PTR_NO apoio1, apoio2;
    apoio1=apoio2=lista->no;
    while (apoio1->next != nill) /* while (apoio1->next != lista.no) */
        apoio2 = apoio1;
        apoio1 = apoio1->next;
    enWHILE
    apoio2->next=apoio1->next;
    free(apoio1);
end;
```

# Estrutura de Dados - Listas

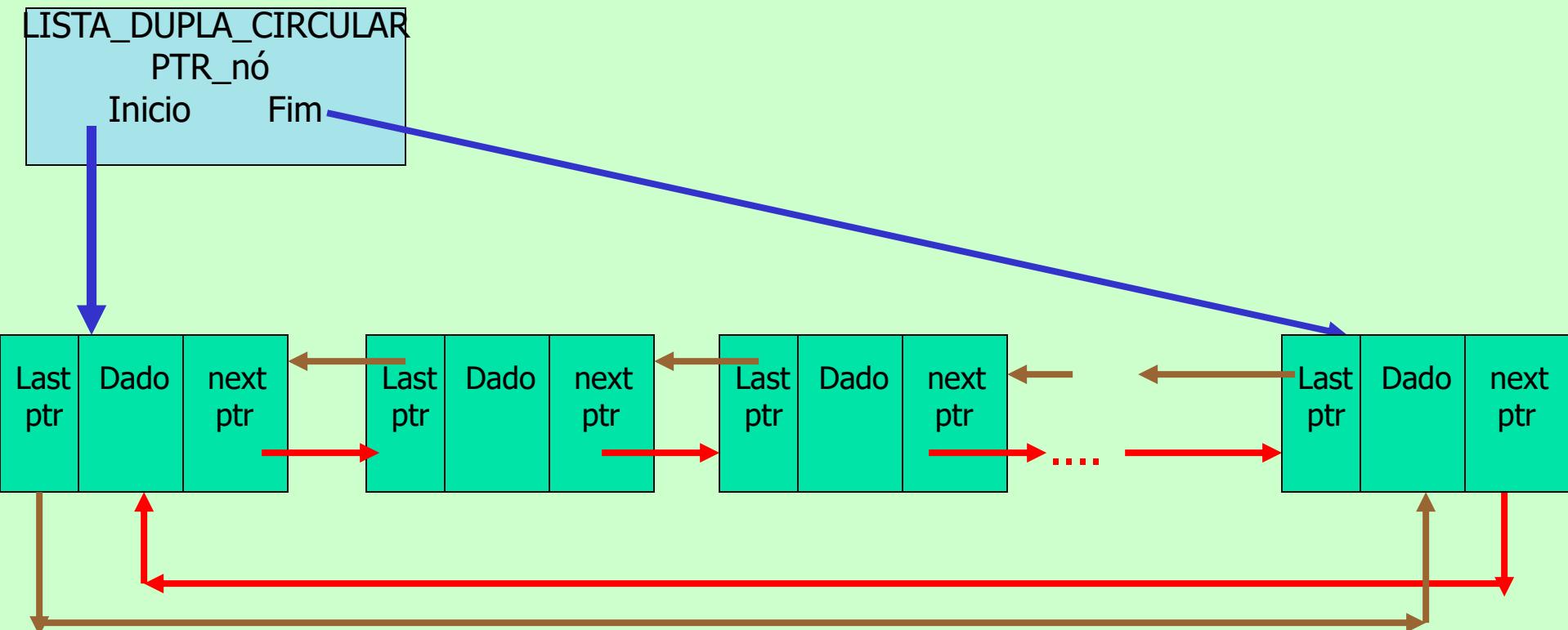
- OBS: Otimização de inserir ou deletar no início ou no fim da lista:

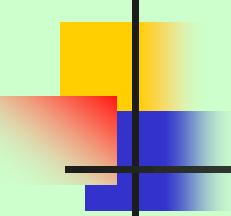


# Estrutura de Dados - Listas

## ■ Por encadeamento (duplo):

- Lista duplamente encadeada descreve que cada nó passa a ter referencias ao seus vizinhos anterior (LAST) e posterior (NEXT);

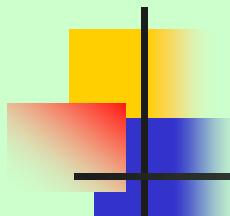




# Estrutura de Dados – Pilhas, Filas e Deques

- PILHAS , FILAS e DEQUES:

- São estruturas de dados do tipo linear, onde são impostas restrições para a inserção e a retirada de dados;
- Critérios:
  - **LIFO** (Last In First Out) => caracterização de uma **pilha**;
  - **FIFO** (First In First Out) => caracterização de uma **fila**;
  - **DEQUE** (Double Ended Queue) => caracteriza uma fila com duas extremidades, ou seja, os acessos para retirada e inclusão ocorrem em ambas as extremidades (inicio e fim da estrutura)



# Estrutura de Dados – Pilhas, Filas e Deques

- Realizações de PILHAS:

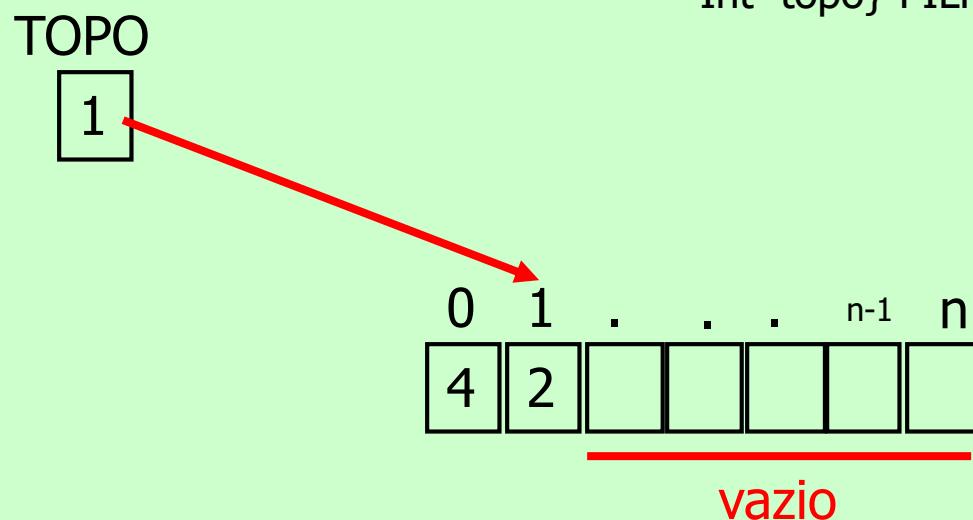
- **Por seqüência de dados:**

- Para tal, podemos fazer uso de uma estrutura de dados seqüência já apresentada;
    - ex: PILHA :: seq char;  
palavra :: CADEIA  
palavra = 'emigrar';  
palavra= 'casa';  
palavra = ' ';

# Estrutura de Dados – Pilhas, Filas e Deques

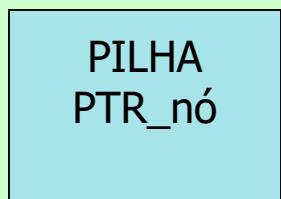
- **Por vetor:**

- Set define Structure {  
    Int vetor[1...n];  
    Int topo} PILHA;

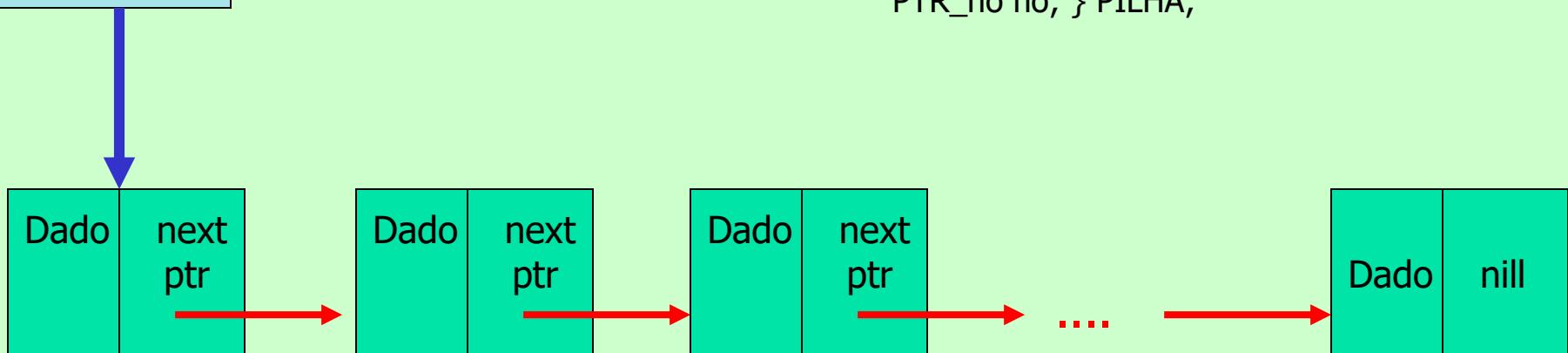


# Estrutura de Dados – Pilhas, Filas e Deques

- Por lista linear simplesmente encadeada:



```
Set define structure {  
    Info dados;  
    PTR_nó next; } Nó;  
Set define structure {  
    PTR_nó nó; } PILHA;
```



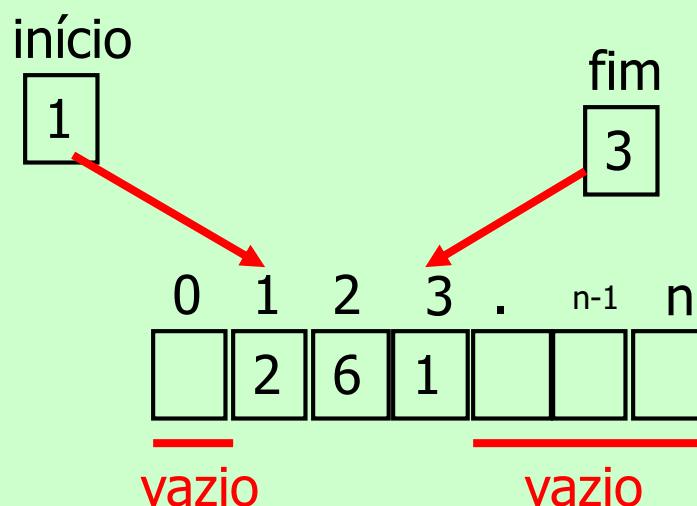
# Estrutura de Dados – Pilhas, Filas e Deques

- Realizações de FILAS:

- **Por vetor:**

- Set define Structure {

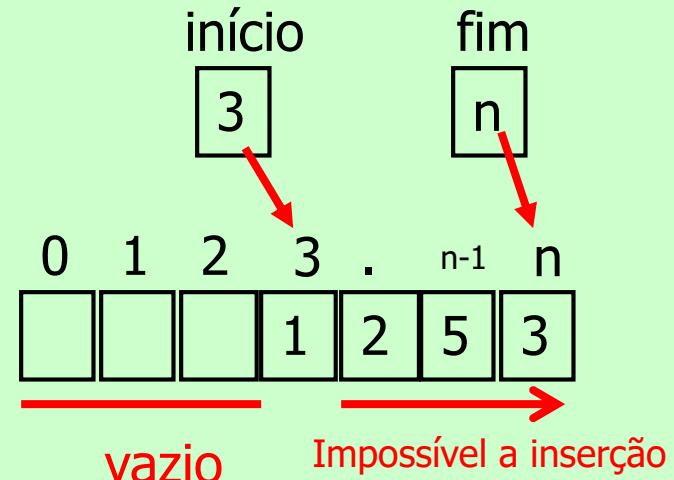
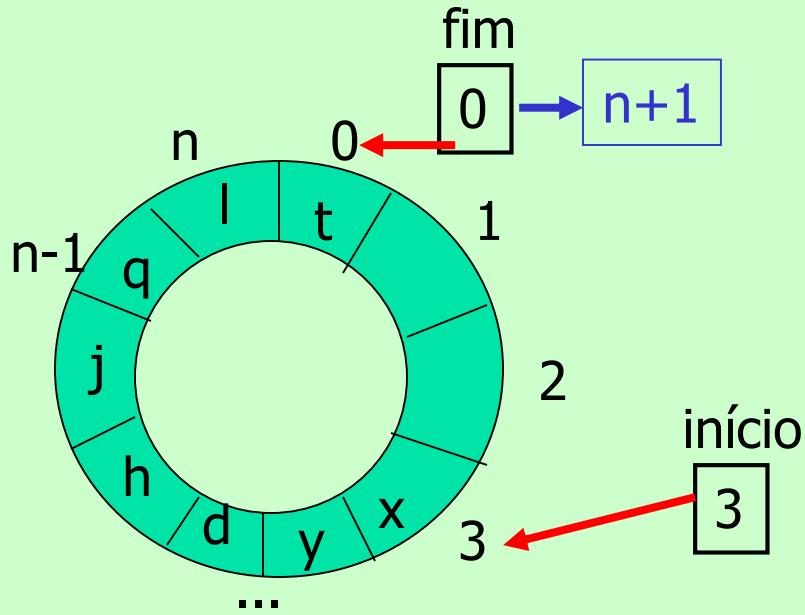
```
    Int vetor[1...n];  
    Int inicio, fim} FILA; /* inicio = retirada  
                                fim = inclusão */
```



# Estrutura de Dados – Pilhas, Filas e Deques

- Problema de overflow:

Solução: representação circular da fila

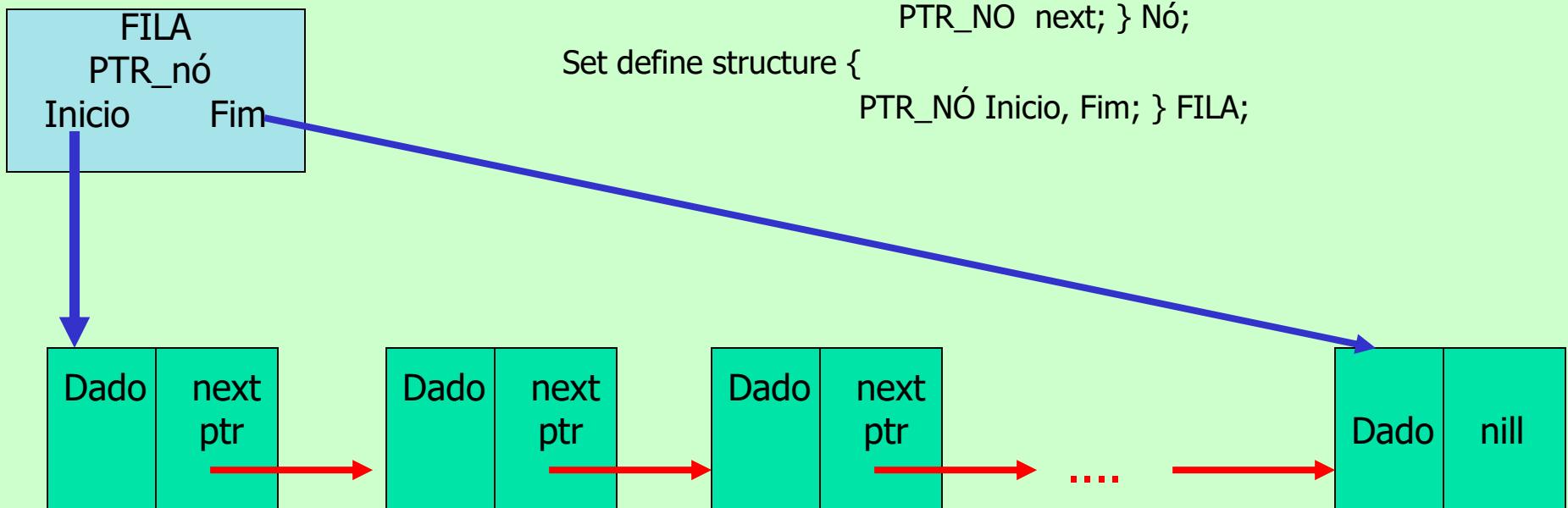


OBS:

1. Comprimento do vetor =  $p = n+1$ ;
2. Usar a função MOD:  
ex:  $p=10 \Rightarrow x \text{ MOD } 10$   
 $0 \text{ MOD } 10 = 0$   
 $10 \text{ MOD } 10 = 0$   
 $11 \text{ MOD } 10 = 1$

# Estrutura de Dados – Pilhas, Filas e Deques

- **Fila por lista linear**

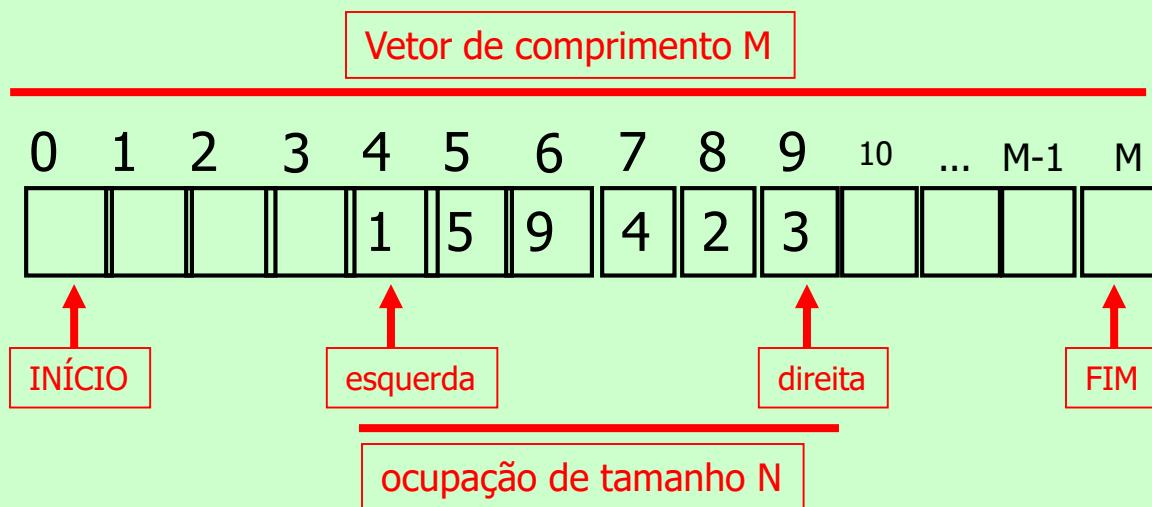


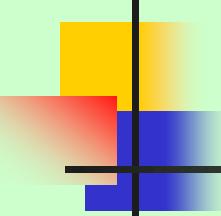
# Estrutura de Dados – Pilhas, Filas e Deques

- Representação do DEQUE:

- Por vetor:

```
Set define Structure {  
    Int direita, esquerda;  
    Int inicio, fim;  
    Elementos vetor[1...M]  
} DEQUE
```





# Estrutura de Dados – Pilhas, Filas e Deques

- Inicializar um DEQUE:

```
DEQUE deq  
deq.direita = m/2;  
deq.esquerda = m/2 +1;  
deq.inicio= 0;  
deq.fim=m ;
```

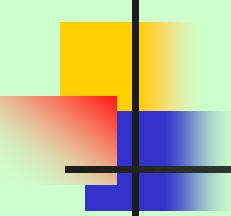
- Inserir à esquerda em um DEQUE(Dado)

```
begin_Procedure;  
if deq.esquerda = deq.inicio then  
print("erro: overflow");  
else  
deq.esquerda = deq.esquerda - 1  
deq.vetor[deq.esquerda] = dado;  
endif;  
end_Procedure;
```

- Inserir à direita em um DEQUE(Dado)

```
begin_Procedure;  
if deq.direita = deq.fim then  
print("erro: overflow");  
else  
deq.direita = deq.direita + 1  
deq.vetor[deq.direita] = dado;  
endif;  
end_Procedure;
```

OBS: podemos definir um DEQUE circular usando a função MOD



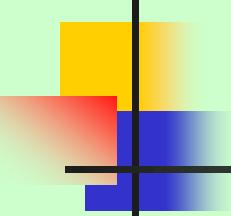
# Estrutura de Dados – Pilhas, Filas e Deques

- Eliminar à esquerda em um DEQUE()

```
begin_Procedure;  
if deq.direita < deq.esqueda then  
    print("erro: DEQUE vazio !!");  
else  
    deq.esquerda = deq.esquerda + 1  
endif;  
end_Procedure;
```

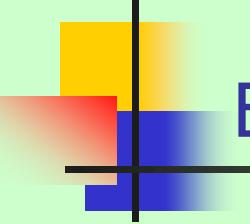
- Eliminar à direita em um DEQUE()

```
begin_Procedure;  
if deq.direita < deq.esqueda then  
    print("erro: DEQUE vazio !!");  
else  
    deq.direita = deq.direita - 1  
endif;  
end_Procedure;
```



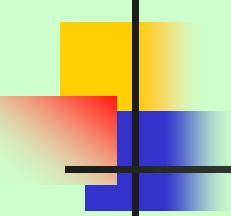
# Estrutura de Dados – Pilhas, Filas e Deques

- Uma aplicação importante de PILHAS em funções matemáticas:
  - INFIXA: A+B
  - PREFIXA: +AB
  - POSFIXA: AB+
  
  - Ex: INFIXA para POSFIXA:
    - A + B \* C
    - A + (B C \*)
    - A B C \* +
  - Ex: INFIXA para PREFIXA:
    - A + B \* C
    - A + (\* B C)
    - + A \* B C
  
- Regras de precedência para conversão :
  1. exponenciação
  2. Multiplicação e divisão
  3. Adição e subtração



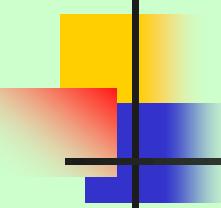
# Estrutura de Dados – Pilhas, Filas e Deques

FORMA INFIXA	FORMA POSFIXA	FORMA PREFIXA
$A+B$	$AB+$	$+AB$
$A+B-C$	$AB+C-$	$-+ABC$
$(A+B)*(C-D)$	$AB+CD-*$	$*+AB-CD$
$A\$B*C-D+E/F/(G+H)$	$AB\$C*D-EF/GH+/+$	$+-$ABCD//EF+GH$
$((A+B)*C-(D-E))\$(F+G)$	$AB+C*DE- -FG+\$$	$$-*+ABC-DE+FG$
$A-B/(C*D\$E)$	$ABCDE$*/-$	$-A/B*C\$DE$



# Estrutura de Dados – Pilhas, Filas e Deques

- **Como avaliar uma expressão POSFIXA ???**
- **REGRA:**
  - Cada operador refere-se aos dois operando anteriores. Note que o operando pode ser o próprio resultado !!!
- **SOLUÇÃO:**
  - Cada vez que se lê um operando, nós o introduzimos em uma pilha. Ao atingirmos um operador, seus operandos são os 2 elementos no topo da pilha; Note que os 2 elementos são retirados, e o resultado é incluído no topo da pilha para ser usado por um próximo operador;



# Estrutura de Dados – Pilhas, Filas e Deques

```
Avalia_POSIXA( PILHA exp)
begin_Procedure
    PILHA pilha_operando = " "; /* inializar com pilha vazia */
    While (exp != "")
        caracter=LER_PILHA(exp);
        if (caracter = operando) then
            INCLUIR_PILHA(caracter, pilha_operando);
        else
            op1 = LER_PILHA(pilha_ operando);
            op2 = LER_PILHA(pilha_ operando);
            valor = CALCULAR( op2, op1, caracter);
            INCLUIR_PILHA(valor, pilha_operando);
        endif;
    endwhile;
end_Procedure;
```

Exemplo:

6 2 3 + - 3 8 2 / + \* 2 \$ 3 + = 52

# Estrutura de Dados – Pilhas, Filas e Deques

## ■ Conversão de infixa para pósfixa:

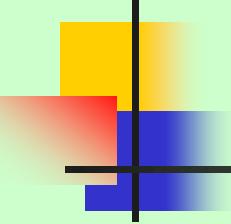
- Fazer uso de uma função PRECED(op1,op2) retornando:
  - =1 se op1 tem precedência sobre op2;
  - =0 se op2 tem precedência sobre op1;
  - Ex: PRECED( \* , + ) = 1;  
PRECED( + , + ) = 1;  
PRECED( + , \* ) = 0;  
PRECED( \$ , \$ ) = 0;

Exemplos:

A\*B+C\*D

A+B\*C\$D\$E

```
In_to_POS (PILHA exp)
begin
  PILHA pilha_operadores = "";
  PILHA pilha_pos = "";
  while ( exp != "" ) do
    caracter = LER_PILHA(exp);
    if (caracter = operando) then
      INCLUIR_PILHA(caracter, pilha_pos);
    else
      while (pilha_operadores != "") &&
        (PRECED( LER_PILHA(pilha_operadores) , caracter) ) do
        topo = LER_PILHA(pilha_operadores);
        INCLUIR_PILHA( topo, pilha_pos)
      endwhile;
      INCLUIR_PILHA(caracter, pilha_operadores);
    endif;
  endwhile;
  while ( pilha_operadores != "" ) do
    topo = LER_PILHA(pilha_operadores);
    INCLUIR_PILHA( topo, pilha_pos);
  endwhile;
end_Procedure;
```

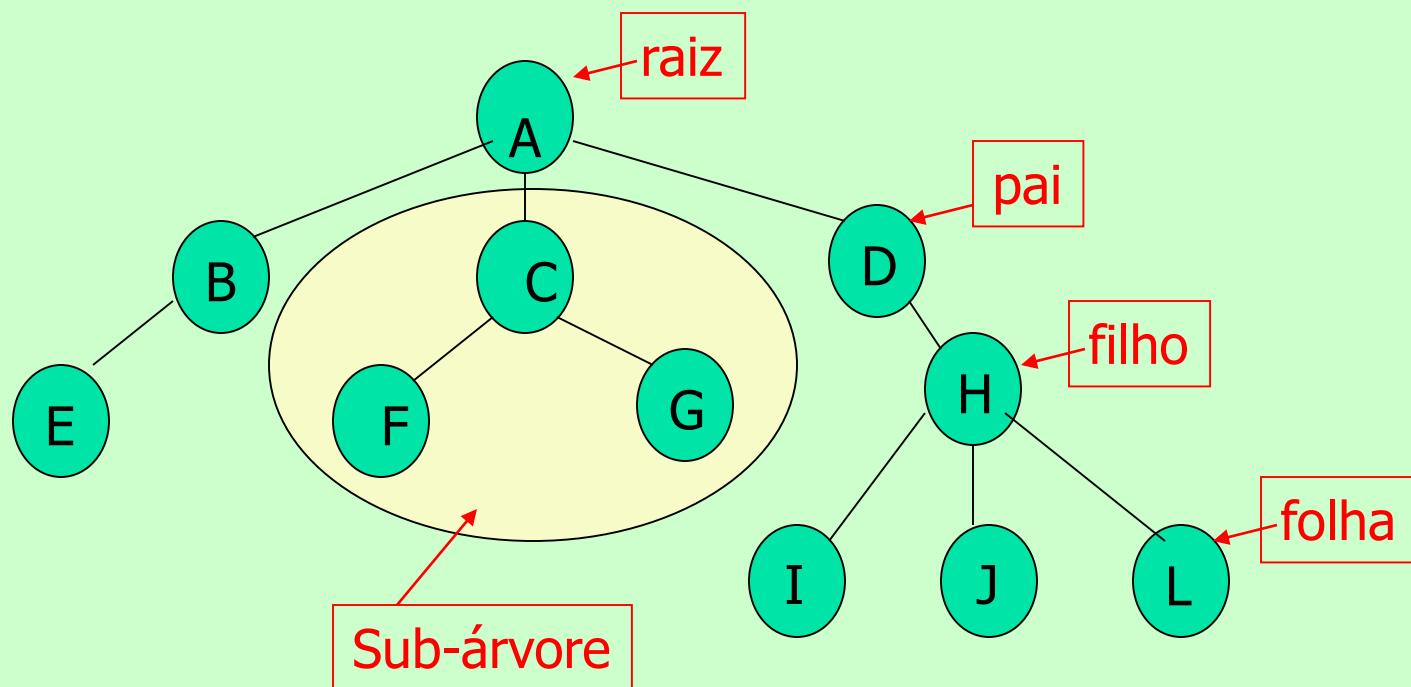


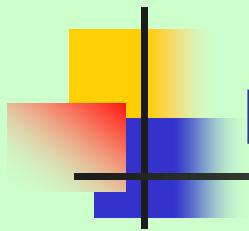
# Estrutura de Dados – Árvores

- Arvores:
  - São estruturas de dados que caracterizam uma relação hierárquica , ou de decomposição, entre dados;
  - Tem-se sempre um conjunto de dados hierarquicamente subordinado a outro;
  - Matematicamente, uma árvore é um conjunto finito **T** com um, ou mais, nós da forma que:
    1. Existe um nó denominado RAIZ;
    2. Os demais nós formam conjuntos disjuntos  $s_1, s_2, s_3, \dots, s_{n-1}, s_n$ , onde cada um desses conjuntos é uma árvore, ou seja uma subárvore de **T**.
    3. Todo o nó de uma árvore é raiz de uma subárvore;

# Estrutura de Dados – Árvores

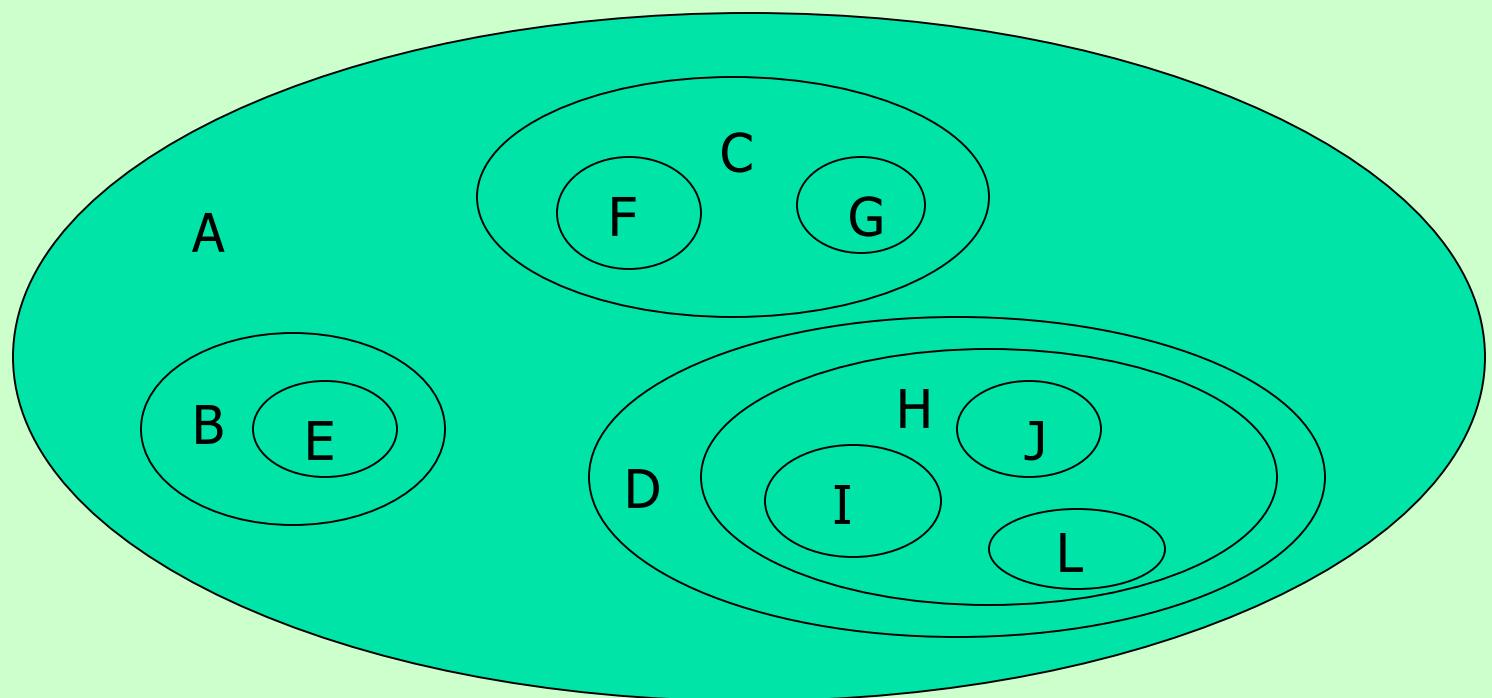
- Representações:
  - Esquemática por hierarquia:

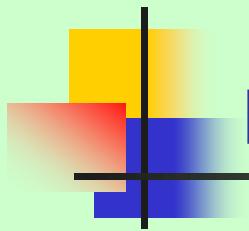




# Estrutura de Dados – Árvores

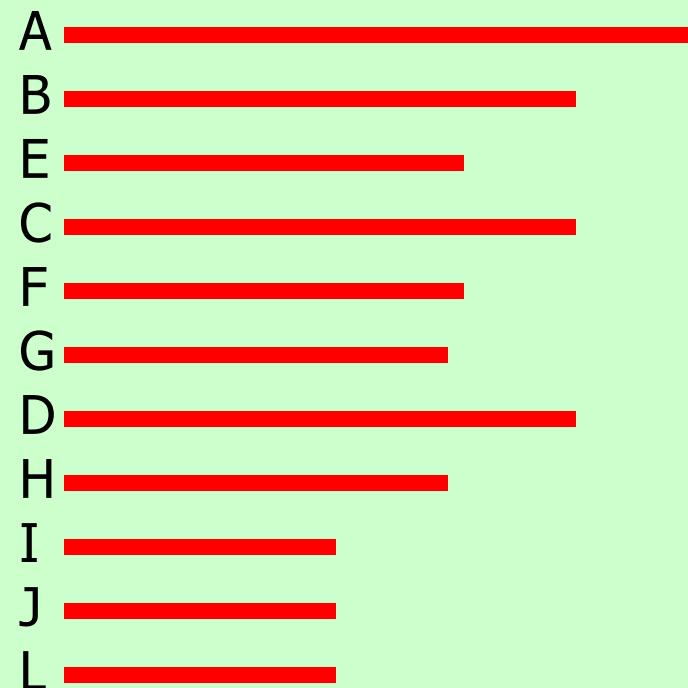
- Por inclusão (baseada na teoria de conjuntos)





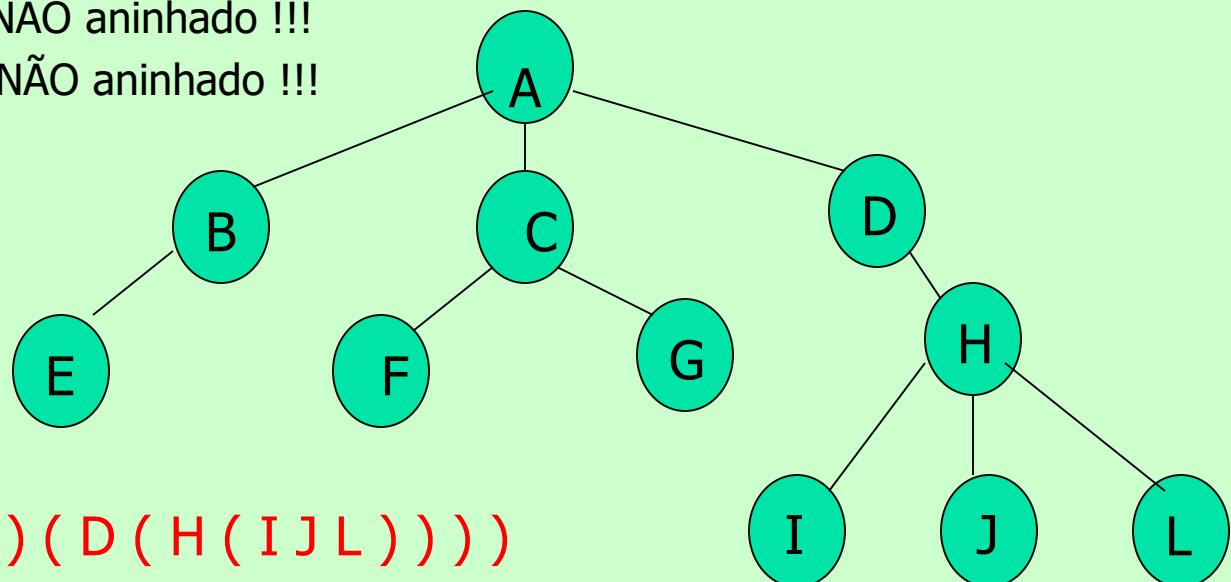
# Estrutura de Dados – Árvores

- Por diagrama de barras



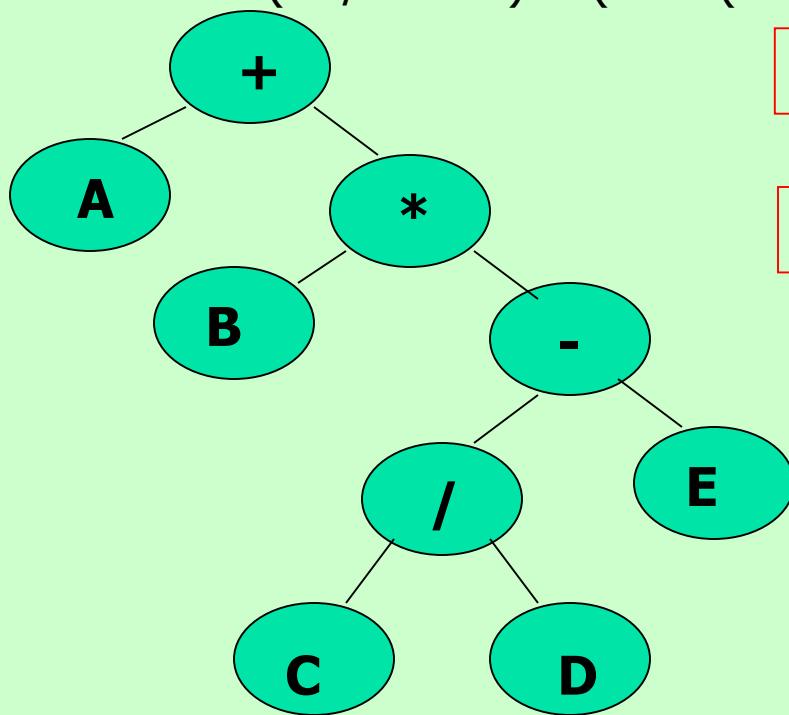
# Estrutura de Dados – Árvores

- Por parênteses aninhados:
  - Seqüência aninhada = cada sub-seqüência tem maior número de " ( " do que " ) ";
  - Ex: ( ( ) ( ) ) = aninhado !!  
      ( ( ) )( ) = NÃO aninhado !!!  
      ( ) )( ) = NÃO aninhado !!!



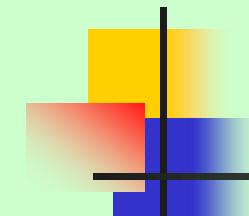
# Estrutura de Dados – Árvores

- OBS: sabendo-se que toda a expressão matemática pode ser escrita na forma de parênteses aninhados. Então, pode-se definir uma expressão matemática como uma árvore, onde cada nó corresponderá a um operador, e suas sub árvores corresponderão à operandos;
- Ex:  $A + B * ( C / D - E ) = ( A + ( B * ( ( C / D ) - E ) ) )$



PRÉ FIXADA: + A \* B - / C D E

PÓS FIXADA: A B C D / E - \* +



# Estrutura de Dados – Árvores

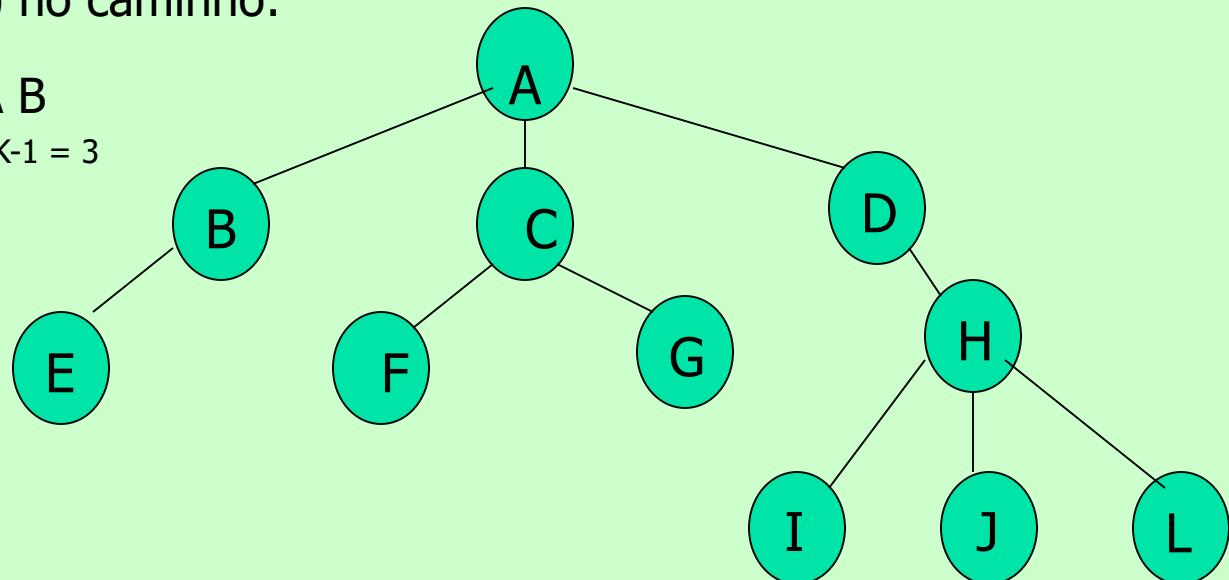
- Todas as características de uma árvore estão baseadas no relacionamento pai-filho:
  1. Cada nó é pai de vários filhos, exceto os nós-folhas que NÃO possuem filhos;
  2. Todo nó é filho de alguém, exceto o RAIZ que não tem pai (pai virtual – sistema);
  3. O grau do nó é o seu número de filhos;
  4. Toda a árvore com  $n$  nós, onde  $n > 1$ , possui no mínimo 1 filho e no máximo  $n-1$  filhos;
  5. Nó NÃO folha = nó interno;

# Estrutura de Dados – Árvores

6. Caminho da árvore é baseado na relação do tipo “é filho de” ou “é pai de”, sendo descrito por uma seqüência de nós (ou vértices);  
Essa seqüência de nós é especificada por dois vértices conhecidos, caracterizados por nó\_1 ALCANÇA no\_2;  
Logo, o comprimento de um caminho é sempre  $K-1$ , onde k é o número de nós (ou vértices) no caminho:

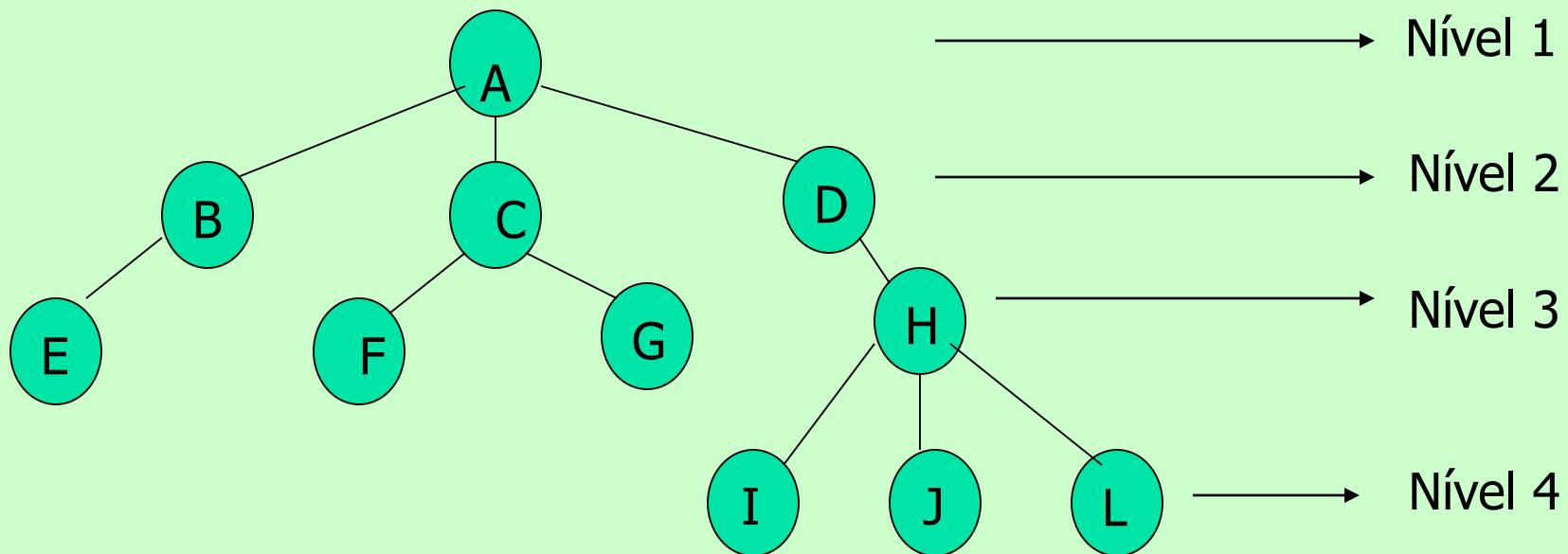
ex: **C alcança E = C A B**

comprimento do caminho ( $k=4$ ):  $K-1 = 3$



# Estrutura de Dados – Árvores

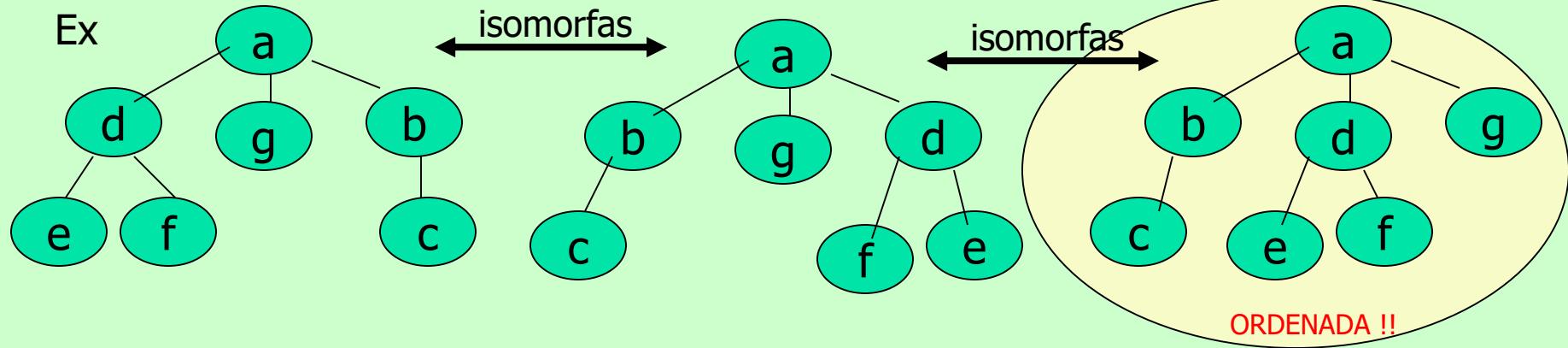
7. Nível de um nó é o número de nós do caminho da raiz até o nó. O nível da raiz é sempre 1;



# Estrutura de Dados – Árvores

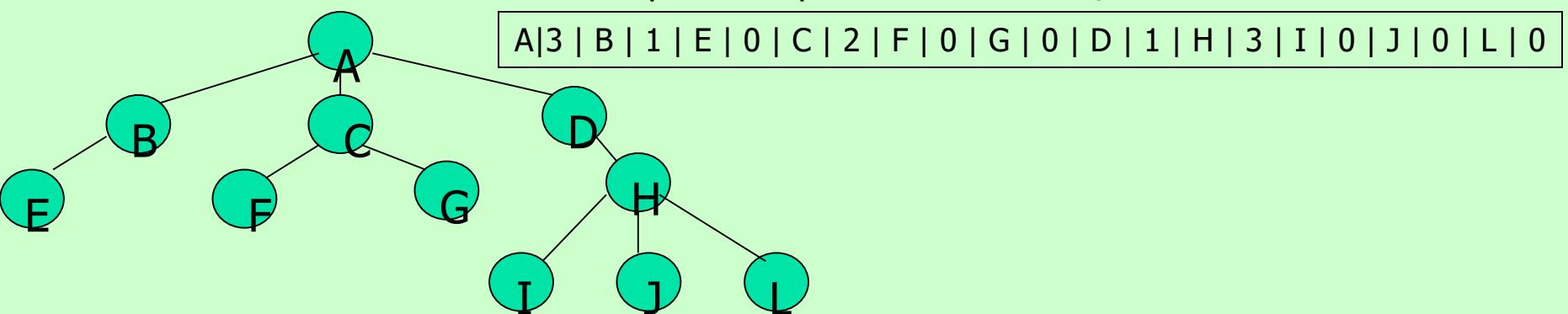
8. Altura do nó é o número de nós do maior caminho do nó até um de seus descendentes mais 1, pois toda a folha tem altura 1;  
Ex:  $\text{alt}( C ) = \text{caminho} + \text{folha} = 1 + 1 = 2;$   
 $\text{alt}( F ) = \text{caminho} + \text{folha} = 0 + 1 = 1;$   
 $\text{alt}( B ) = \text{caminho} + \text{folha} = 1 + 1 = 2;$   
 $\text{alt}( A ) = \text{caminho} + \text{folha} = 3 + 1 = 4;$
9. Arvore Ordenada possui os filhos de cada nó ordenados da esquerda para a direita (por convenção);
10. Arvores Isomorfas: duas arvores são ditas isomorfas se, aplicando-se a ordenação, tornam-se coincidentes a uma arvore ordenada;

Ex



# Estrutura de Dados – Árvores

- Alocação de árvores:
  - Por adjacência:
    - Os nós são representados seqüencialmente na memória, isso respeitando-se a ordem convencionada em que eles aparecem na arvore;

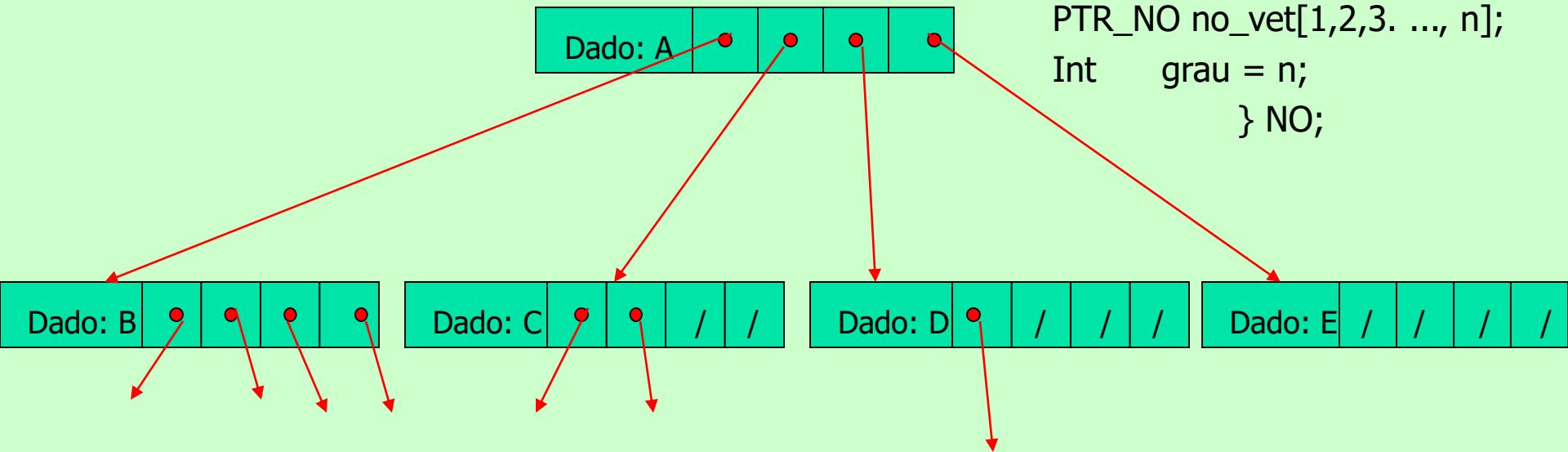


OBS: a alocação por seqüênciA NÃO é uma maneira conveniente de se representar uma arvore pois as operações de manipulação (inserir, alterar ou remover) são difíceis de serem trabalhadas !!

# Estrutura de Dados – Árvores

- Por Encadeamento:

```
Set define Structure{  
    DADO valor;  
    PTR_NO no_vet[1,2,3. ..., n];  
    Int     grau = n;  
} NO;
```



**PROBLEMA:** cada nó pode ter um número variável de referências a outros nós (sub-árvores);

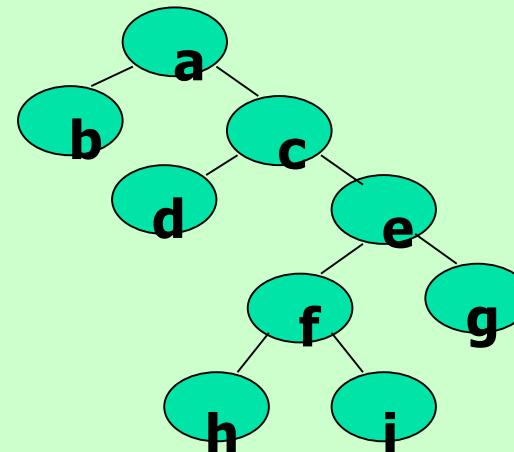
# Estrutura de Dados – Árvore Binária

- Arvore Binária:
  - É uma estrutura de dados do tipo arvore que se caracteriza por:
    1. Existe somente um nó raiz de T, e os demais nós são subdivididos em dois subconjuntos disjuntos ( $T_{esquerda}$  e  $T_{direira}$ ), ou seja, uma subarvore esquerda (Te) e outra subarvore direita (Td), onde ambas se mantêm como arvores binárias;
    2. Todo o pai SEMPRE possuirá 2 filhos (mesmo quando vazios): um filho esquerdo; e um filho direito;

OBS:

1. as arvores binárias não obedecem rigidamente as definições de arvores;
2. Considerações sobre o ISOMORFISMO não podem ser aplicadas devido as relações entre o pai e seus filhos esquerdo e direito;
3. Todas as arvores binárias com n nós, possuirão n+1 subarvores vazias;

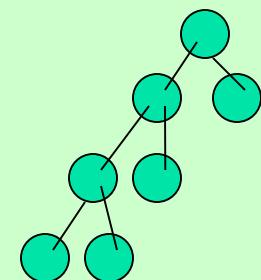
EX:



# Estrutura de Dados – Árvore Binária

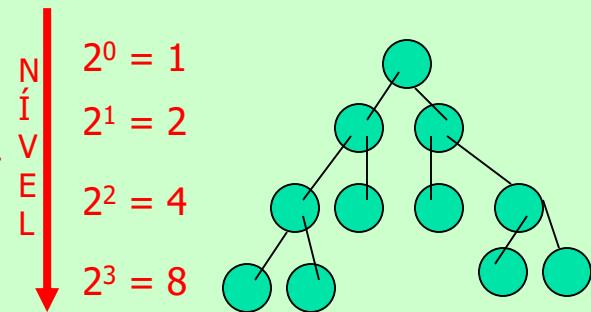
- Tipos especiais de árvores binárias:

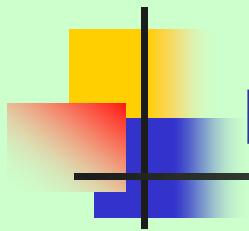
1. Árvore estritamente binária: cada nó possui 0 ou 2 filhos:



2. Árvore binária completa: se um nó possui subárvore vazia, este nó estará localizado no último ou penúltimo nível da árvore.

Note que a árvore completa tem altura mínima !!! ->

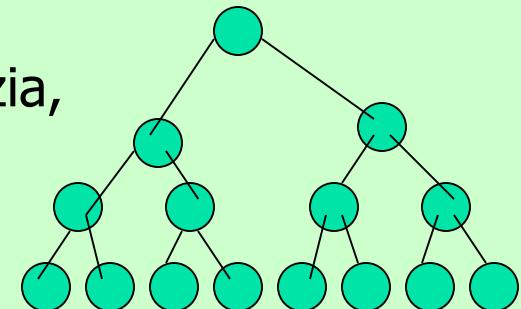




# Estrutura de Dados – Árvore Binária

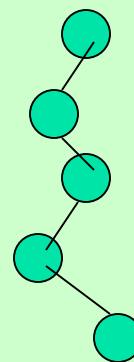
3. Árvore binária cheia: se um nó possui subárvore vazia, este nó estará localizado no último nível da árvore;

OBS: ruim para inserção / exclusão de nós !!



4. Árvore binária zig-zag: possui uma altura máxima e os nós internos possuem exatamente um subárvore vazia;

OBS: péssima para busca =>  $O(n)$  !!!



# Estrutura de Dados – Árvore Binária

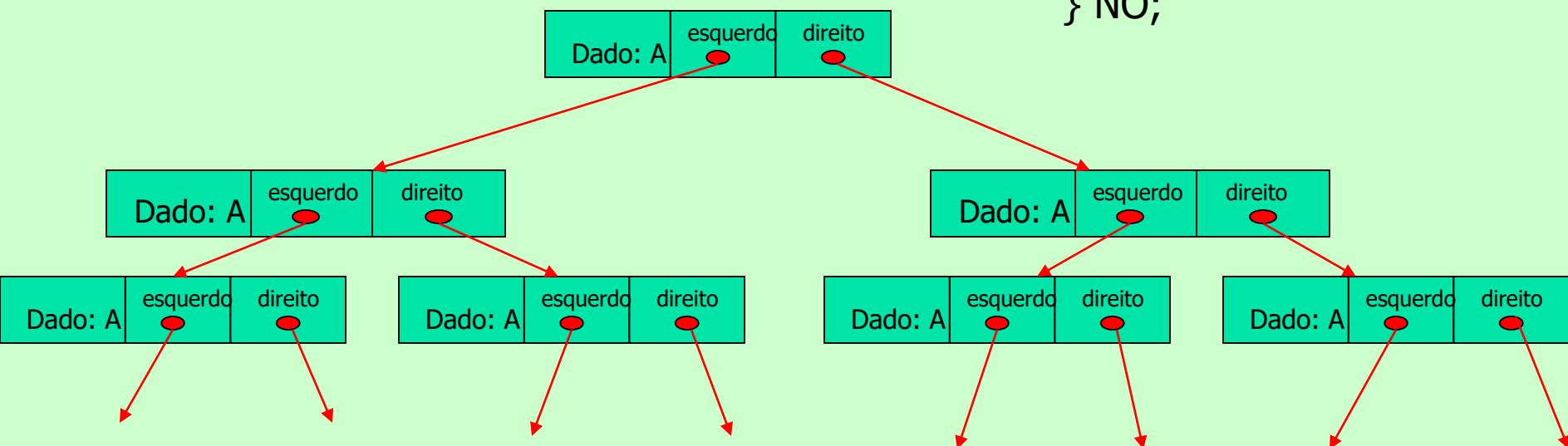
- Forma de representação:

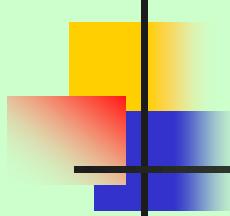
- Por encadeamento: Set define Structure{

```
DADO valor;
```

```
PTR_NO esquerdo, direito;
```

```
} NO;
```





# Estrutura de Dados – Árvore Binária

- Percursos em árvores binárias:

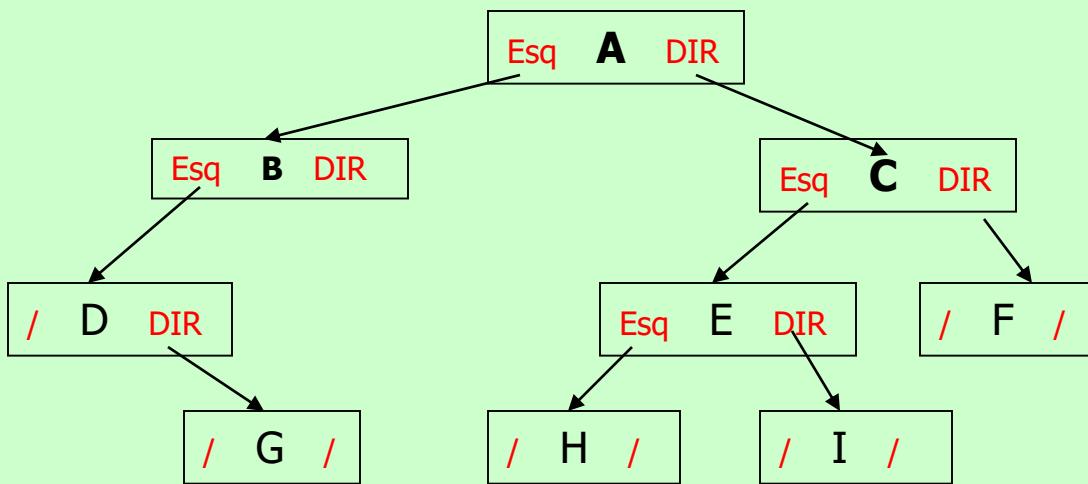
- Consiste em um procedimento de visita ÚNICA sistemática a cada nó de uma árvore, com o objetivo de promover alguma operação sobre a informação contida no nó;
- É uma operação básica relativa a manipulação de árvores;
- Durante o percurso, pode-se passar n vezes sobre o mesmo nó sem visitá-lo.
- Note que a visita a um nó é SEMPRE ÚNICA no processo de manipulação para preservar custo  $O(n)$ , onde n é o número total de nós da árvore;
- O algoritmo de percurso é visitar a raiz de cada subárvore **v** da árvore **T**, caracterizando um processo **RECURSIVO**;
- O percurso sobre uma árvore T pode ser definido por uma composição de percursos de suas subárvores de forma contínua;

# Estrutura de Dados – Árvore Binária

- Tipos de percursos:

1. Percurso em PRÉ-ORDEM:

- Visitar raiz;
- Percorrer recursivamente sua subarvore a esquerda em pré-ordem;
- Percorrer recursivamente sua subarvore a direita em pré-ordem;



```
Algoritmo PRE(PTR_NO no)
begin
    visita (no);
    if (no->esquerda != nill) then
        PRE(no->esquerda);
    if (no->direita != nill) then
        PRE(no->direita);
end_Procedure;
```

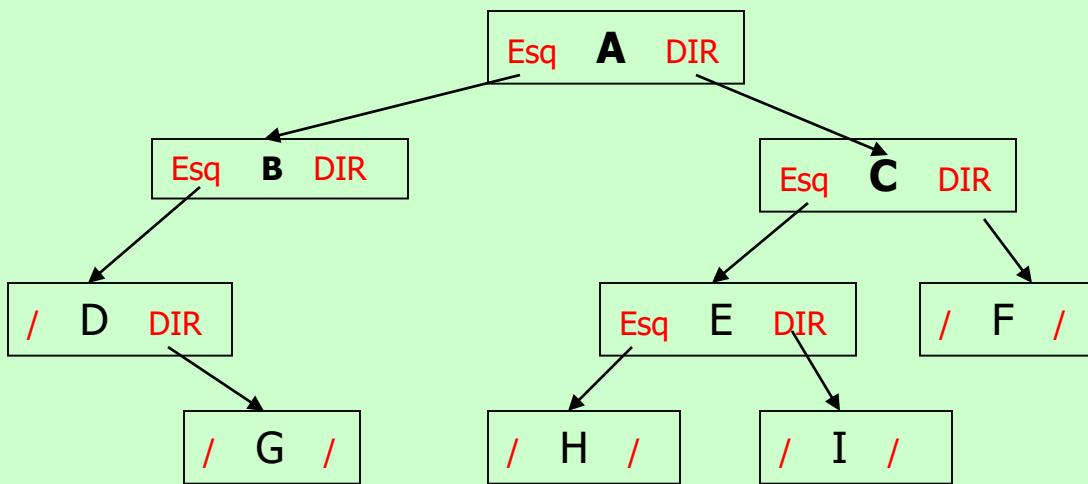
**Resultado:**  
**A B D G C E H I F**

# Estrutura de Dados – Árvore Binária

2.

## Percorso em PÓS-ORDEM:

- Percorrer recursivamente sua subarvore a esquerda em PÓS-ordem;
- Percorrer recursivamente sua subarvore a direita em PÓS-ordem;
- Visitar raiz;



```
Algoritmo POS(PTR_NO no)
begin
  if (no->esquerda != nill) then
    POS(no->esquerda);
  if (no->direita != nill) then
    POS(no->direita);
  visita (no);
end_Procedure;
```

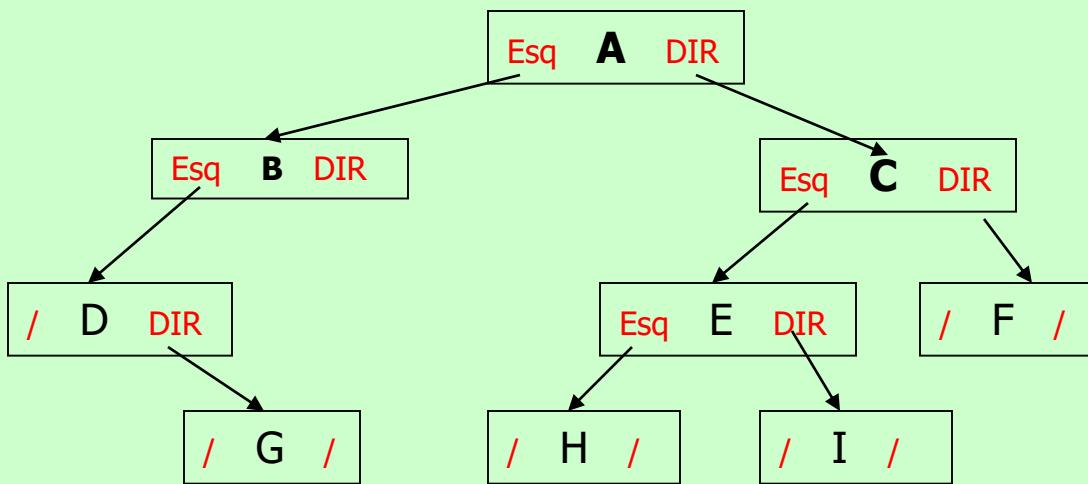
**Resultado:**

**G D B H I E F C A**

# Estrutura de Dados – Árvore Binária

## 3. Percurso em ORDEM-SIMÉTRICA:

- (a) Percorrer recursivamente sua subarvore a esquerda em ordem-SIMÉTRICA;
- (b) Visitar raiz;
- (c) Percorrer recursivamente sua subarvore a direita em ordem- SIMÉTRICA;

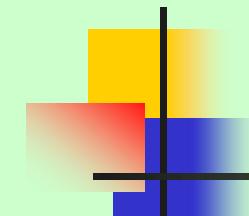


Algoritmo SIMETRICA(PTR\_NO no)  
begin

```
if (no->esquerda != nill) then  
    SIMETRICA(no->esquerda);  
    visita (no);  
if (no->direita != nill) then  
    SIMETRICA(no->direita);  
end_Procedure;
```

**Resultado:**

**D G B A H E I C F**



# Estrutura de Dados – Árvore Binária

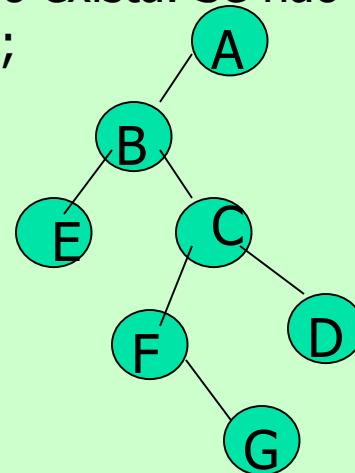
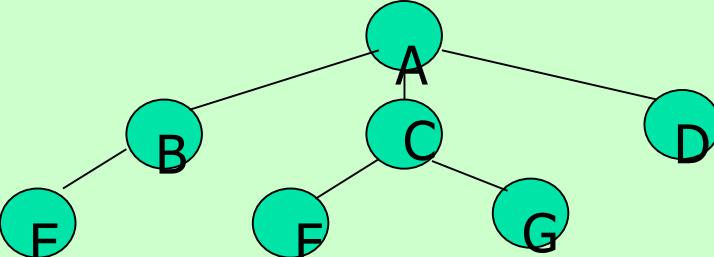
- **OBS:** O algoritmo de cálculo da altura de um nó faz uso do algoritmo de percurso em pós-ordem para controlar as variações de altura a partir do conhecimento das alturas dos descendentes esquerdo e direito.

## Visita (PTR\_NO no)

```
begin
    if (no->esquerda != nill) then
        h1 = no->esquerda.altura;
    else
        h1=0; /* no caso de folhas */
    end_IF;
    if (no->direita != nill) then
        h2 = no->direita.altura;
    else
        h2=0; /* no caso de folhas */
    end_IF;
    if ( h1 > h2) then
        no.altura = h1 + 1;
    else
        no.altura = h2 + 1;
    end_IF;
end_PROCEDURE;
```

# Estrutura de Dados – Árvore Binária

- Conversão de uma árvore genérica (Tg) em árvore binária (Tb):
  - A raiz de Tb é a raiz de Tg;
  - O filho esquerdo de um nó v em Tb, corresponde ao primeiro filho de v em Tg, caso exista. Se não existir, a subárvore esquerda de v em Tb será **VAZIA**;
  - O filho direito de um nó v em Tb, corresponde ao próximo irmão de v (próximo irmão a direita de v) em Tg, caso exista. Se não existir, a subárvore direita de v em Tb será **VAZIA**;

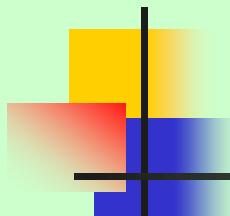


# Estrutura de Dados – Árvore Binária

**Converter\_Tg para Tb** (PTR\_NO\_BINARIO no\_b,  
PTR\_NO\_GENERICO no\_g)

```
begin
    PTR_NO_BINARIO no_b1;
    if (Tb->raiz = nill) then
        begin
            no_b.valor = no_g.valor;
            Tb->raiz=no_b;
        end_IF;
    For i = 1 to no_g.grau do
        if ( i = 1) then
            begin
                alloc(no_b1);
                no_b1.valor = no_g->no_vet[i].valor
                no_b->esquerdo = no_b1;
            end_IF;
        if ( i != 1) then
            begin
                alloc(no_b1);
                no_b1.valor = no_g->no_vet[i].valor
                no_b->direito = no_b1;
                no_b = no_b1;
            end_IF;
        Converte_Tg para Tb (no_b, no_g->no_vet[i]);
    end_FOR;
end_PROCEDURE;
```

```
Inicio_conversão ( ARVORE_GENERICA Tg);
begin
    ARVORE_BINARIA Tb;
    PTR_NO_GENERICO no_g;
    PTR_NO_BINARIO no_b;
    no_g =Tg->raiz;
    alloc(no_b);
    Converter_Tg para Tb (no_b, no_g);
End_Inicio_Conversão;
```



# Estrutura de Dados – Árvore Binária de Busca

- Arvore Binária de Busca:

- **OBJETIVO:**

- Identificar em um conjunto de elementos caracterizados por uma chave, aquele elemento que corresponde a chave procurada (uma busca !!!);

- **SOLUÇÃO:**

- Os elementos do conjunto serão previamente distribuídos pelos nós de uma arvore binária. Então a busca (localizar a chave procurada) consiste em definir um percurso orientado pela chave;

- **COMPUTACIONALMENTE:**

- Problema da Busca:

- Seja  $S=\{s_1, s_2, s_3, \dots, s_n\}$  um conjunto de chave onde  $s_1 < s_2 < s_3 < \dots < s_n$ ;
      - Seja  $x$  a informação desejada;
      - A busca consiste em saber de  $x \in S$ ;
      - Caso verdadeiro, localizar  $x$  em  $S$ , ou seja , determinar o índice  $j$  tal que  $s_j = x$ ;

# Estrutura de Dados – Árvore Binária de Busca

## Solução Computacional:

- Usar uma árvore binária rotulada ( $T$ ) com as seguintes características:
  1.  $T$  possui  $n$  nós. Cada nó  $v$  corresponde a uma chave  $s_j \in S$ , e possui como rótulo o valor  $r(v) = s_j$ ;
  2. Se  $v$  é um nó raiz de  $T$ , e  $v1$  pertence à subárvore esquerda de  $T$ . Logo,  $r(v1) < r(v)$ ;
  3. Analogamente, se  $v$  é um nó raiz de  $T$ , e  $v2$  pertence à subárvore direita de  $T$ . Logo,  $r(v2) > r(v)$ ;

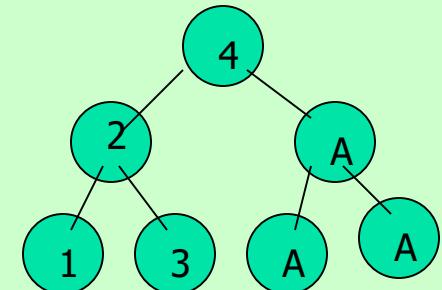
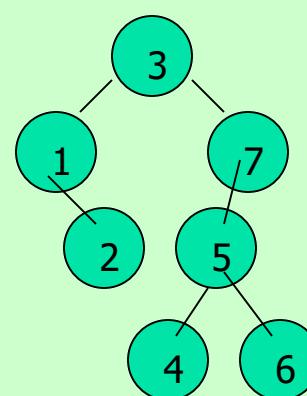
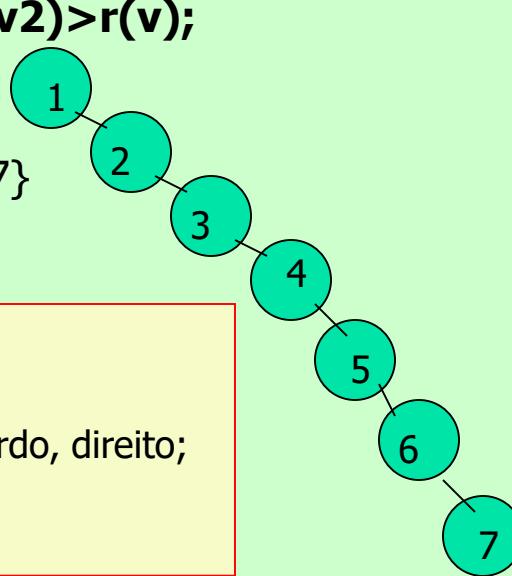
EX:  $S = \{ 1, 2, 3, 4, 5, 6, 7 \}$

Set define Structure{

    CHAVE rotulo;

    PTR\_NO esquerdo, direito;

} NO;



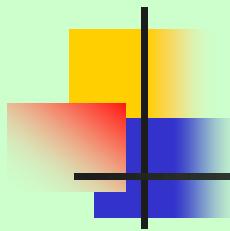
# Estrutura de Dados – Árvore Binária de Busca

```
Busca_Arvore_Binária ( CHAVE x, PTR_NO no, INT f)
begin
  if ( no = nill ) then return(f=0); /* não existe arvore */
  if ( x = no.rotulo) then return(f=1); /* busca com sucesso */
  if ( x < no.rotulo ) then
    begin
      if (no->esquerda = nill) then
        return(f=-1) /* busca sem sucesso */
      else
        Busca_Arvore_Binária(x, no->esquerda, f);
      end_IF;
    else
      if (no->direita = nill) then
        return(f=-1) /* busca sem sucesso */
      else
        Busca_Arvore_Binária(x, no->direita, f);
      end_IF;
    end_IF;
end_PROCEDURE;
```

Complexidade  $O(n)$ ,  
onde  $n$  é a altura da arvore !!

LOGO: é bom garantir sempre a  
Altura mínima !!!

Questão: arvore binária completa ???



# Estrutura de Dados – Árvore Binária de Busca

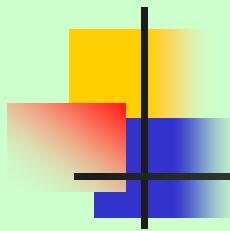
- Inserção de nós na árvore de Busca:

- Seja  $x$  o valor de uma chave que se deseja inserir na árvore de busca  $T$ .
- Para isso, primeiro verifica-se de  $x \in S$ , onde  $S = \{ s_1, s_2, \dots, s_n \}$ ;
- Em caso positivo,  $x$  não poderá ser inserido pois causaria duplicidade de informação;
- Em caso negativo, então  $x$  será o rótulo de um nó situado a esquerda ou a direita de algum nó folha da árvore  $T$ ;
- **Solução algorítmica:** fazer uso do algoritmo “Busca\_Arvore\_Binária”, adotando  $f$  como indicador da posição de inserção:
  - **$f = 2$**  => inserir novo nó a esquerda do ultimo nó visitado;
  - **$f = 3$**  => inserir novo nó a direita do ultimo nó visitado;

# Estrutura de Dados – Árvore Binária de Busca

```
Inserir_No ( CHAVE x)
begin
    no = T->raiz;
    Busca_Arvore_Binaria(x, no, f);
    if ( f = 1) then
        Printf( "x é existente !!!" );
    else
        alloc(no1)
        no1.rotulo = x;
        no1.valor=Get_Informação();
        no1->esquerda = nill;
        no1->direita = nill;
        if ( f = 0 ) then
            T->raiz= no1;
        else
            if ( f = 2 ) then no->esquerda = no1;
            if ( f = 3 ) then no-> direita = no1;
        endIF;
    endIF;
end_PROCEDURE;
```

```
Busca_Arvore_Binária ( CHAVE x, PTR_NO no, INT f)
begin
    if ( no = nill ) then return(f=0); /* não existe arvore */
    if ( x = no.rotulo) then return(f=1); /* busca com sucesso */
    if ( x < no.rotulo ) then
        begin
            if (no->esquerda = nill) then
                return(f= 2) /* busca sem sucesso – inserir nó à esquerda*/
            else
                Busca_Arvore_Binária(x, no->esquerda, f);
            end_IF;
        else
            if (no->direita = nill) then
                return(f=3) /* busca sem sucesso – inserir nó à direita */
            else
                Busca_Arvore_Binária(x, no->direita, f);
            end_IF;
        end_IF;
    end_PROCEDURE;
```

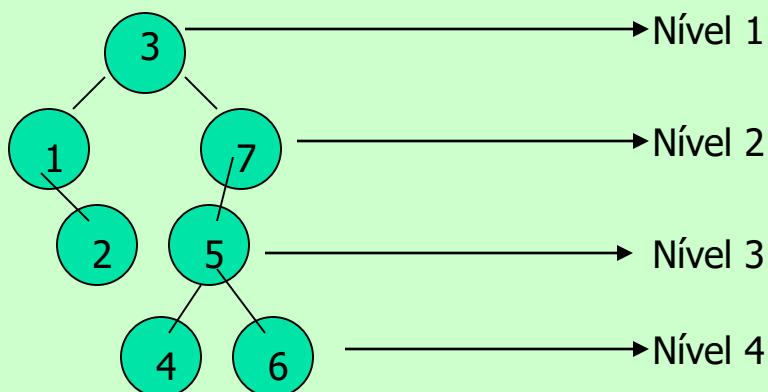


# Estrutura de Dados – Árvore Binária de Busca

- O algoritmo apresentado não garante a altura mínima. Ou seja, a altura poderá ser  $n$  (ex: arvore zig-zag) onde a complexidade no pior caso para a construção será  $O(n^2)$ ;
- Note que a construção da arvore depende da ordem de inclusão dos nós (já visto anteriormente !!);
- A arvore de busca cheia (menor altura !!) será construída garantindo-se uma boa ordenação das chaves dos nós. Note que sua construção será  $O(n \log n)$ ;

# Estrutura de Dados – Árvore Binária de Busca

- Parâmetros de avaliação de uma árvore binária de busca:
  - Caminho Interno: é o número total de comparações entre chaves para se localizar cada nó da arvore T (buscas com sucesso):
    - $I(T) = \sum_{1 \leq k \leq n} l_k$  , onde  $l_k$  descreve o nível de cada nó da arvore.



$$I(T) = 1+2+2+3+3+4+4 = 19$$

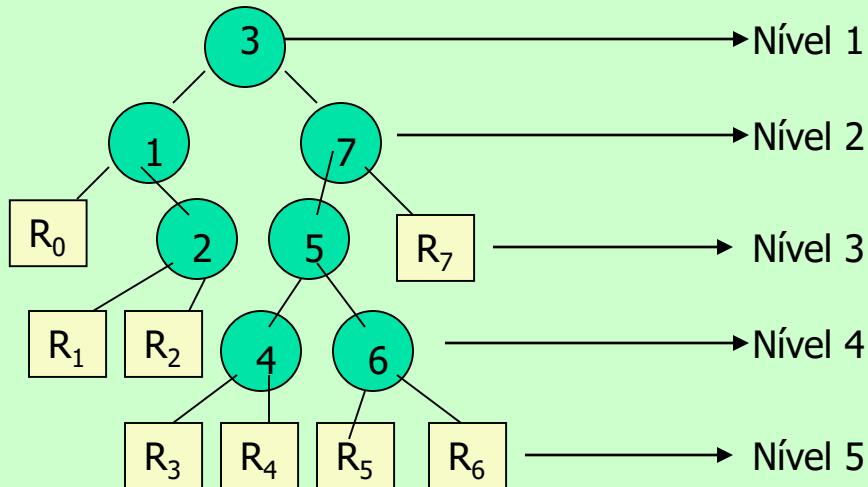
# Estrutura de Dados – Árvore Binária de Busca

- Caminho Externo: é o número total de comparações entre chaves para todos os intervalos de buscas sem sucesso;
  - Para a avaliação do caminho externo devemos representar os intervalos de nós referente as buscas sem sucesso:

$$R_0 = \{ x \in R, \text{ tal que } x < s_1 \}$$

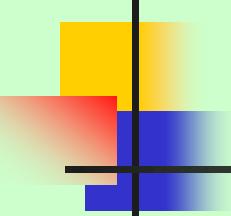
$$R_n = \{ x \in R, \text{ tal que } x > s_n \}$$

$$R_j = \{ x \in R, \text{ tal que } s_j < x < s_{j+1} \}, \text{ onde } j = 1, 2, 3, \dots, n-1.$$



$$E(T) = \sum_{0 \leq k \leq n} (l'_k - 1), \text{ onde } l'_k \text{ descreve o nível de cada nó da árvore.}$$

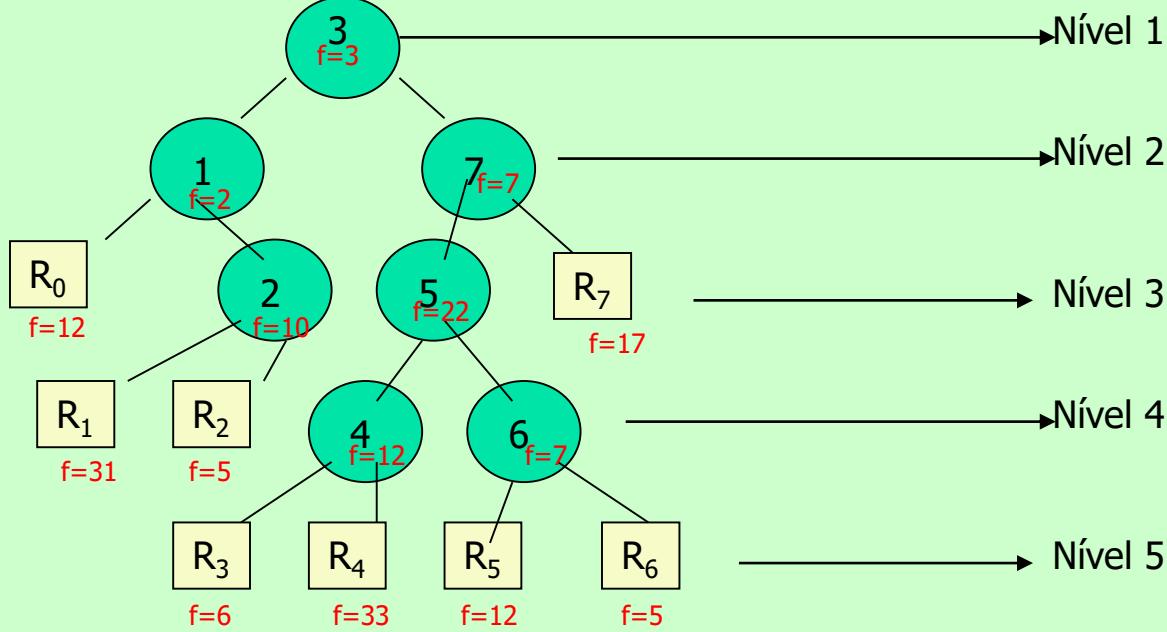
$$E(T) = (3-1) + (4-1) + (4-1) + (5-1) + (5-1) + (5-1) + (3-1) = 26$$



# Estrutura de Dados – Árvore Binária de Busca

- Busca com freqüência de acesso diferenciado:
  - PROBLEMA: algumas chaves apresentam freqüências de acesso distintas;
  - SOLUÇÃO: definir uma melhor árvore para resolver a busca com freqüência de acesso diferenciado:
    - Seja T uma árvore binária de busca;
    - Cada chave  $s_k$  possui uma freqüência de acesso  $f_k$ , e esta localizada na árvore T em um nível  $l_k$ , onde  $1 \leq k \leq n$  ;
    - A árvore T possui folhas, considerando as buscas sem sucesso ( $R_0, R_1, \dots, R_n$ ), onde cada  $R_k$  possui uma freqüência  $f'_k$ , e esta em um nível  $l'_k$  da árvore T, onde  $0 \leq k \leq n$  ;
  - LOGO:  $C(T) = I(T) + E(T) = \sum_{1 \leq k \leq n} f_k \cdot l_k + \sum_{0 \leq k \leq n} f'_k \cdot (l'_k - 1)$

# Estrutura de Dados – Árvore Binária de Busca



$$\begin{aligned} I(T) &= \\ &1.3 + 2.2 + 2.7 + 3.10 + 3.22 + 4 \\ &.12 + 4.7 \\ &= 223 \end{aligned}$$

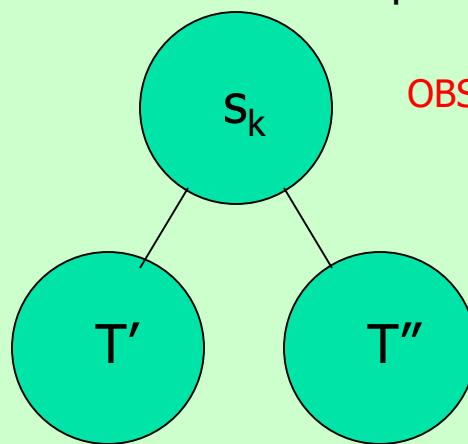
$$\begin{aligned} E(T) &= (3-1).12 + (4- \\ &1).31 + (4-1).5 + (5- \\ &1).6 + (5-1).33 + (5- \\ &1).12 + (5-1).5 + (3-1).17 \\ &= 372 \end{aligned}$$

LOGO: C(T) = I(T) + E(T) = 595

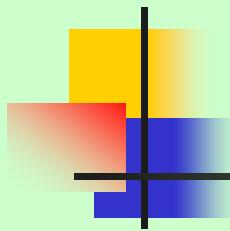
# Estrutura de Dados – Árvore Binária de Busca

- Árvore Ótima:

- É aquela de custo total mínimo;
- Logo, definir uma árvore ótima para um determinado problema consiste em especificar a árvore de menor custo. MAS COMO ????????
- SOLUÇÃO: usar a programação dinâmica, que consiste em se decompor o problema em dois (ou mais..) outros problemas menores, onde a decomposição especifica subconjuntos de S (conjunto de nós da árvore):
  - Seja  $s_k$  a raiz da árvore T.
  - Logo,  $T'$  e  $T''$  serão sub-árvore a esquerda e a direita, respectivamente:



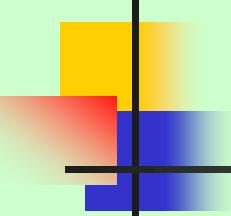
OBS: sub-árvore de árvores ótimas também são ótimas



# Estrutura de Dados – Árvore Binária de Busca

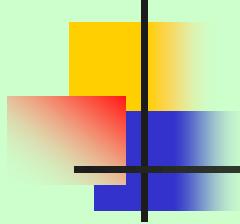
- Características de T':
  1. T' é uma árvore binária de busca;
  2. Seu nós internos são  $\{s_1, s_2, \dots, s_{k-1}\}$ ;
  3. Seus nós externos são  $\{R_0, R_1, \dots, R_{k-1}\}$ ;
- Características de T":
  1. T" é uma árvore binária de busca;
  2. Seu nós internos são  $\{s_{k+1}, s_{k+2}, \dots, s_n\}$ ;
  3. Seus nós externos são  $\{R_k, R_{k+1}, \dots, R_n\}$ ;

PROBLEMA: (1) como conhecer a raiz ( $s_k$ ) ????  
(2) Como determinar T' e T" ?????



# Estrutura de Dados – Árvore Binária de Busca

- Solução trivial:
  1. tentar todas as possibilidades para escolher o melhor  $s_k$ . Note que serão  $n$  tentativas !!!!
  2. A determinação de  $T'$  e  $T''$ , deve-se fazer uso da recursividade:
    - Para construir  $T'$ :
      1. Escolhe-se uma raiz com chave entre  $S_1, S_2, \dots, S_{k-1}$ ;
      2. Construir as duas novas sub-arvores  $T'_{\text{nova}}$  e  $T''_{\text{nova}}$ ;
      3. Repetir o processo até ser obtida uma solução trivial para a árvore ótima, ou seja, quando a sub-árvore for vazia;
    - Para construir  $T''$ :
      1. Escolhe-se uma raiz com chave entre  $S_{k+1}, S_{k+2}, \dots, S_n$ ;
      2. Construir as duas novas sub-arvores  $T'_{\text{nova}}$  e  $T''_{\text{nova}}$ ;
      3. Repetir o processo até ser obtida uma solução trivial para a árvore ótima, ou seja, quando a sub-árvore for vazia;



# Estrutura de Dados – Árvore Binária de Busca

- Logo, tem-se:
  - $T(i,j)$  = árvore correspondente ao conjunto de chaves  $s_{i+1}, s_{i+2}, \dots, s_j$ , onde  $0 \leq i \leq j \leq n$  ;
  - $F(i,j)$  = somatório de todas as freqüências correspondentes a  $T(i,j)$ , onde:

$$F(i, j) = \sum_{i < k \leq j} f_k + \sum_{i \leq k \leq j} f'_k$$

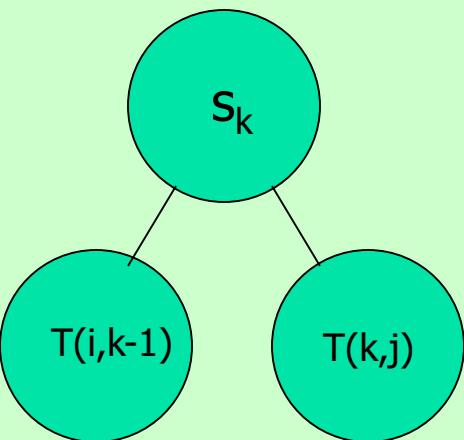
- A relação dos custos da árvore  $T$  com as sub-arvores  $T'$  e  $T''$  é:

$$C(T) = C(T') + C(T'') + F(i, j)$$

$$C(T(i, j)) = C(T(i, k-1)) + C(T(k, j)) + F(i, j)$$

# Estrutura de Dados – Árvore Binária de Busca

- Comprovação:



**C(T(i,k-1)), contando com mais 1 nível**

$$C(T(i, j)) = f_k \cdot 1 + \sum_{i < q < k} f_q \cdot (l_q + 1) + \sum_{i \leq q < k} f'_q \cdot (l'_q - 1 + 1) +$$

**C(T(k,j)), + 1 nível**

$$\sum_{k < q \leq j} f_q \cdot (l_q + 1) + \sum_{k \leq q \leq j} f_q \cdot (l'_q - 1 + 1) =$$

$$= f_k \cdot 1 + \sum_{i < q < k} f_q \cdot 1 + \sum_{i < q \leq k} f_q \cdot l_q + \sum_{i \leq q < k} f'_q \cdot (l'_q - 1) + \sum_{i \leq q < k} f'_q \cdot 1 +$$

$$\sum_{k < q \leq j} f_q \cdot 1 + \sum_{k < q \leq j} f_q \cdot l_q + \sum_{k \leq q \leq j} f'_q \cdot (l'_q - 1) + \sum_{k \leq q \leq j} f'_q \cdot 1 =$$

$$= C(T(i, k - 1)) + C(T(k, j)) + F(i, j)$$

# Estrutura de Dados – Árvore Binária de Busca

```
Algoritmo_Arvore_Ótima(Int f[1...n], f'[0...n])
begin
    For j = 0 to n do
        C[j,j]=0;
        F[j,j]=f'[j];
    end_FOR;
    For d = 1 to n do
        For i = 0 to (n-d) do
            j = i + d;
            F[ i , j ] = F[i, j-1] + f[j] + f'[j];
            C[ i , j ]=Custo_Mínimo( i , j ) + F[ i , j ];
        end_FOR;
    end_FOR;
end_PROCEDURE;
```

```
Custo_Mínimo( Int i , j )
begin
    min=0;
    K[ i , j ] = 0;
    For k = i+1 to j do
        custo = C[ i , k-1 ] + C[ k , j ];
        if ( custo < min ) then
            min = custo;
            K[ i , j ] = k;
        end_IF;
    end_FOR;
    return( min );
end_PROCEDURE;
```

## Observações:

- 1- K, C e F são matrizes  $(n+1) \times (n+1)$ ;**
- 2- f[1...n] e f'[0...n] são valores de freqüência conhecidos;**
- 3- Custo da árvore ótima = C [ 0, n ];**

# Estrutura de Dados – Árvore Binária de Busca

- EXEMPLO:

j	0	1	2	3	4
f	-	10	1	3	2
f'	2	1	1	1	1

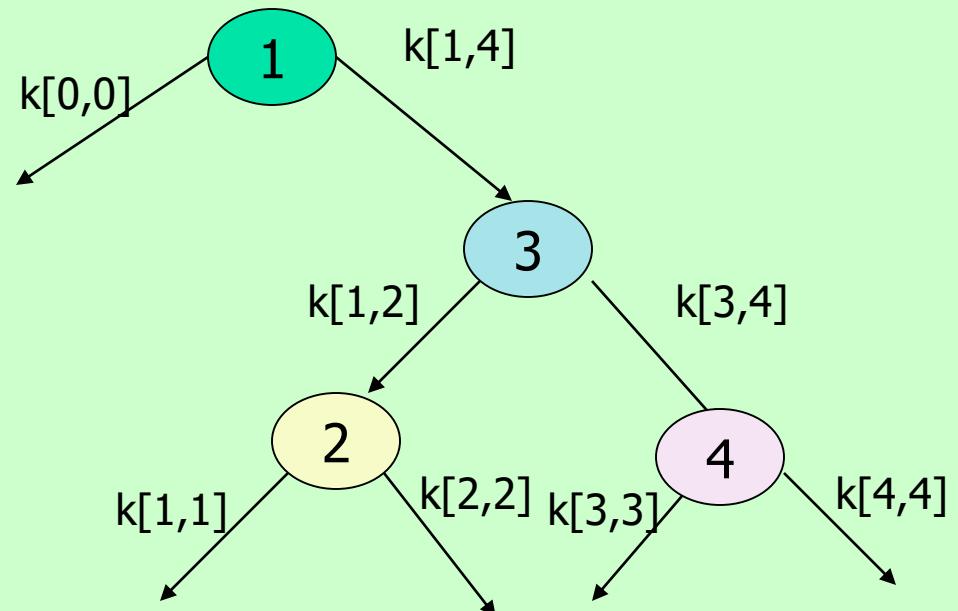
Matriz F				
2	13	15	19	22
	1	3	7	10
		1	5	8
			1	4
				1

Matriz C				
0	13	18	29	39
	0	3	10	17
		0	5	12
			0	4
				0

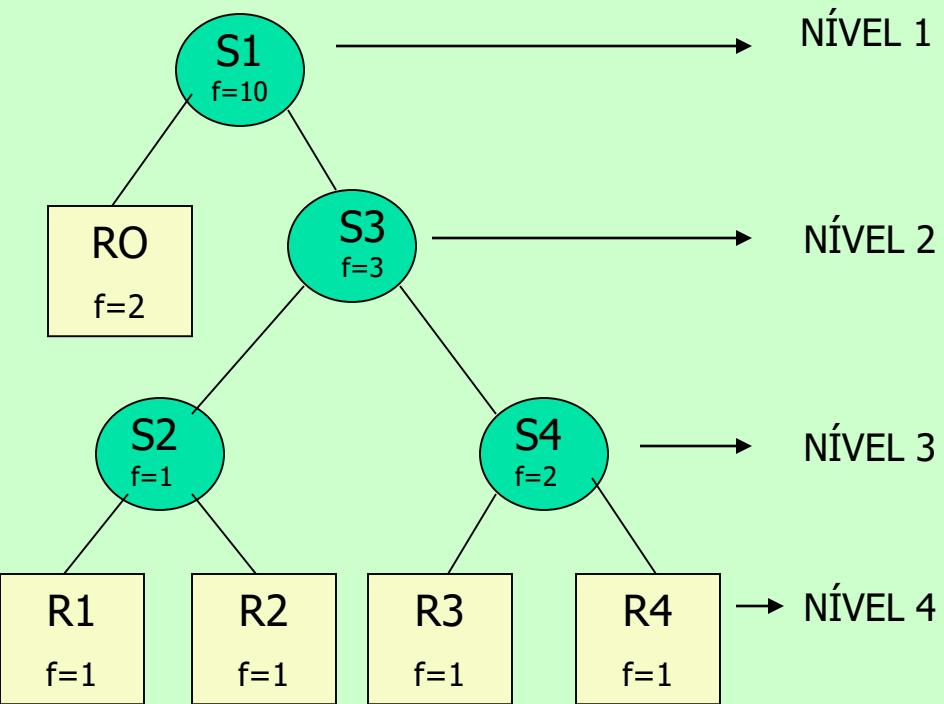
Matriz K				
	1	1	1	1
		2	3	3
			3	3
				4

# Estrutura de Dados – Árvore Binária de Busca

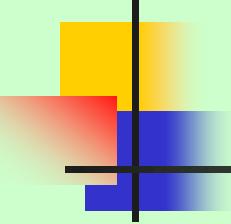
Matriz K				
-	1	1	1	1
-	-	2	3	3
-	-	-	3	3
-	-	-	-	4
-	-	-	-	-



# Estrutura de Dados – Árvore Binária de Busca

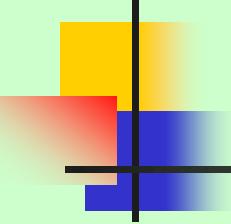


$$\begin{aligned}C(T) &= \sum_{1 \leq k \leq n} f_k \cdot l_k + \sum_{0 \leq k \leq n} f'_k \cdot (l'_k - 1) = \\&= 10 \cdot (1) + 3 \cdot (2) + 1 \cdot (3) + 2 \cdot (3) + 2 \cdot (2-1) + \\&\quad 1 \cdot (4-1) + 1 \cdot (4-1) + 1 \cdot (4-1) + 1 \cdot (4-1) = \\&= 39\end{aligned}$$



# Estrutura de Dados – Árvore de Partilha

- Arvore binária de partilha:
  - QUESTÃO: como escolher a raiz ??
    1. Escolher a chave de maior freqüência de acesso=> otimiza o tempo de busca mas provoca provável desbalanceamento;
    2. Escolhe ra chave de valor médio=> gera árvores balanceadas mas não optimiza o tempo de busca;
  - Conclusão:  
Unir as metodologias (1) e (2), mesmo sendo contraditórias;
  - Solução:  
Arvore binária de partilha !!!



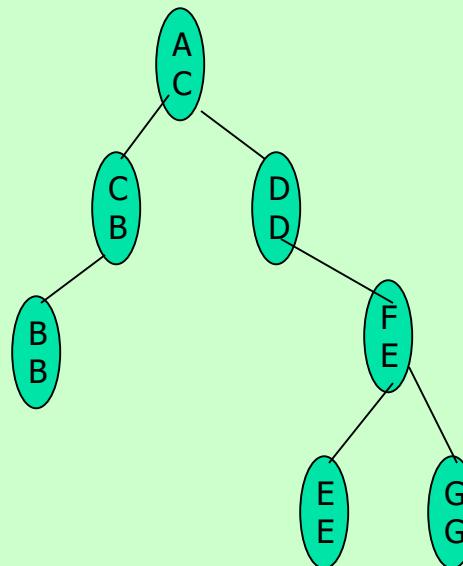
# Estrutura de Dados – Árvore de Partilha

- Consiste em associar a cada nó da árvore duas chaves em vez de uma:
  1. **Chave REAL:** define o objeto de busca;
  2. **Chave de PARTILHA:** define as chaves que compõem as subárvores esquerda e direita do nó considerado;
- Na verdade, consiste em se ter duas chaves objetivando desvincular a chave armazenada em cada nó (REAL) daquela usada para determinar a seqüência de busca (PARTILHA) à direita ou à esquerda;

# Estrutura de Dados – Árvore de Partilha

```
Busca_Partilha(CHAVE x, PTR_NO no)
begin
    if ( no = nill) then
        printf( "x é chave inexistente !! ");
    else
        if ( x = no.real ) then
            printf( "chave x encontrada !! ");
        else
            if ( x ≤ no.partilha ) then
                no=no->esquerda;
                Busca_Partilha(x , no);
            else
                no=no->direita;
                Busca_Partilha(x , no);
            end_IF;
        end_IF;
    end_IF;
end PROCEDURE;
```

```
Set define Structure{
    CHAVE real, partilha;
    PTR_NO_partilha esquerda, direito;
} NO_PARTILHA;
```



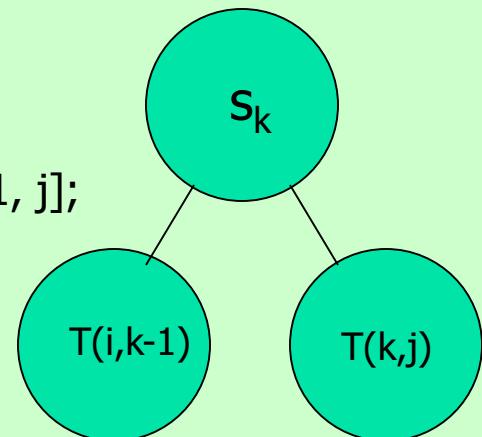
# Estrutura de Dados – Árvore de Partilha

- Criação de uma árvore de partilha:

- $\text{Alcance}(T(i, j)) = [i+1, i+2, \dots, j-1, j] = [i+1, j];$

- Para uma árvore **T** de raiz **s<sub>k</sub>**:
    1.  $\text{Alcance}(T) = [i+1, j];$
    2.  $\text{Alcance}(T') = [i+1, k-1];$
    3.  $\text{Alcance}(T'') = [k+1, j];$

- Idéia:
    1. Definir o alcance de T;
    2. REAL = chave de maior freqüência de acesso dentro do intervalo do alcance;
    3. PARTILHA = chave da mediana de freqüência;
    4. A partir da PARTILHA, criar T' e T'';
    5. Repetir procedimento para T' e T'';

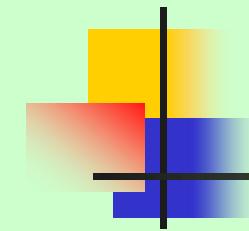


# Estrutura de Dados – Árvore de Partilha

```
Partilha_Otima (PTR_NO_PARTILHA no,
                 CHAVE rótulos[1...n], Int freqüências[1...n],
                 INT alc_inicio, alc_fim )
begin
  PTR_NO_PARTILHA no_esq, no_dir;
  if ( no = nill) then /*verifica o raiz */
    alloc(no);
    arvore_partilha.raiz = no;
    alc_inicio=1;
    alc_fim = n;
  end_IF;
  maior_freq = 0;
  for i = alc_inicio to alc_fim do /*seleção maior freqüência */
    if (maior_freq < freqüências[i] ) then
      maior_freq = freqüências[i];
      maior = i;
    end_IF;
  end_FOR;
  freqüências[ maior ] = -1;
  media=0;
  cont=0;
  for i = alc_inicio to alc_fim do /*seleção freqüência média */
    if (frequências[ i ] != -1 ) then
      media = media + freqüência[i];
      cont++;
    end_FOR;
  media = media / cont;
```

```
distanca = 1000;
for i = alc_inicio to alc_fim do /*seleção próximo media */
  if (frequências[ i ] != -1 ) then
    if ( modulo(freqüências[i] – media) < distancia ) then
      distancia = modulo(freqüências[i] – media);
      media=i;
    end_IF;
  end_IF;
end_FOR;
no.real = rótulos[ maior ];
no.partilha = rótulos[ media ];
Alloc(no_esq);
Alloc(no_dir);
no.esquerda = no_esq;
no.direita = no_dir;
For i = alc_inicio to media do /*novo inicio esquerda */
  if (freqüências[i] != -1) then
    inicio_esq = i;
    i = media+1;
  end_IF;
end_FOR;
For i = media+1 to alc_fim do /*novo inicio direita */
  if (freqüências[i] != -1) then
    inicio_dir = i;
    i = alc_fim+1;
  end_IF;
end_FOR;
```

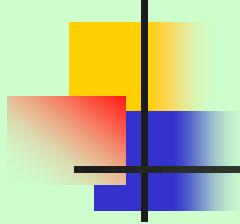
```
fim_esq = media;
fim_dir = alc_fim;
Partilha_Otima (no.esquerda,
                 rótulos[1...n],
                 freqüências[1...n],
                 inicio_esq,
                 fim_esq );
Partilha_Otima (no.direita,
                 rótulos[1...n],
                 freqüências[1...n],
                 inicio_dir,
                 fim_dir );
End PROCEDURE;
```



# Estrutura de Dados – Árvores Balanceadas

- Árvores Balanceadas:

1. Árvore ótima é a de menor custo de acesso;
2. Processos dinâmicos de inserção / deleção provocam desbalanceamento;
3. a redefinição de uma árvore ótima tem alto custo computacional;
4. Logo, árvores balanceadas são estruturas que operam de forma dinâmica, alterando sua estrutura (dados) de forma a garantir um balanceamento quase constante, o que garante um custo de acesso  $O(\log n)$  – árvores ótimas;

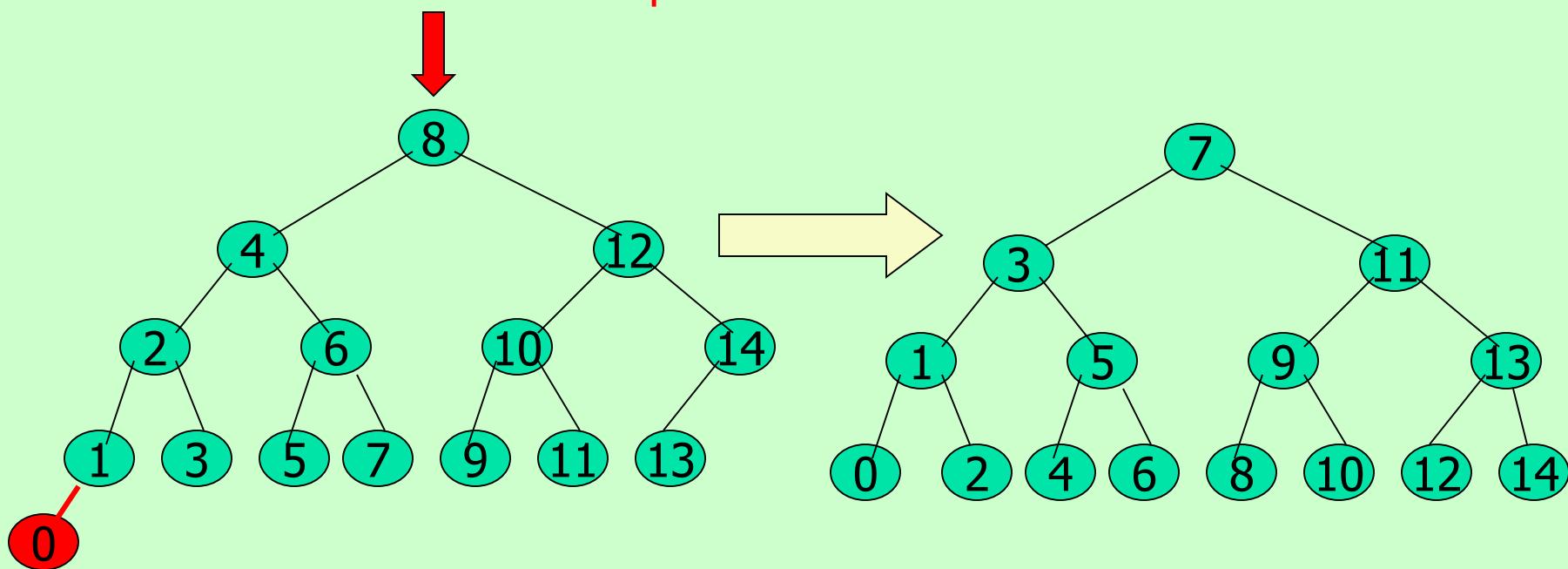


# Estrutura de Dados – Árvores Balanceadas

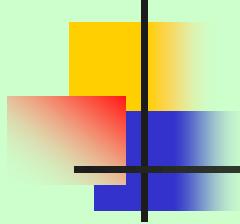
- Conceito de Balanceamento:
  1. Árvore completa minimiza o número de comparações durante a busca;
  2. Árvore completa no comportamento dinâmico é desaconselhável pois no pior caso, pode gerar uma lista linear.
- Idéia:
  - Especificar um algoritmo que ao se alterar dinamicamente uma árvore, a mesma se tornaria completa novamente, tendo que ser um procedimento ' $\Omega(n)$ ' – n é o numero de nós;
  - Note que restabelecer o balanceamento completo com custo ' $\Omega(n)$ ' significa passar por todos os nós uma única vez;

# Estrutura de Dados – Árvores Balanceadas

Desbalanceada e não-completa



LOGO: a árvore completa (busca) não é recomendada para estruturas dinâmicas !!



## Estrutura de Dados – Árvores Balanceadas

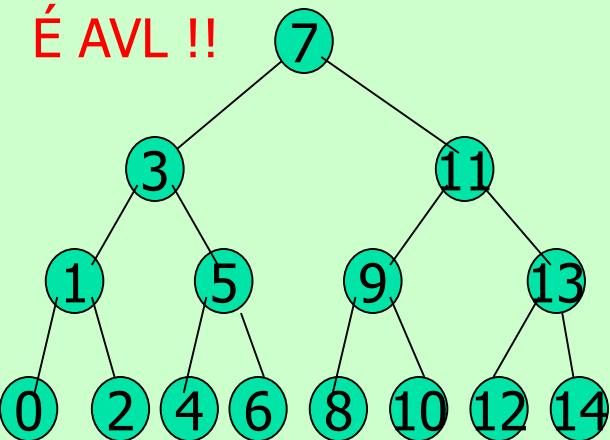
- Solução: usar uma árvore que sua altura seja da mesma ordem de grandeza que a de uma árvore completa e com o mesmo número de nós. Ou seja, altura  $O(\log n)$ , onde  $n$  é o número de nós.
- Então, deve-se usar uma árvore balanceada do tipo AVL, graduada, rubro-negra ou árvore-B.

# Estrutura de Dados – Árvores Balanceadas

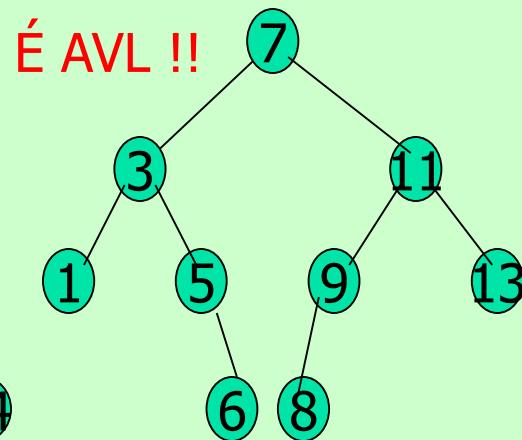
- Árvore AVL:

- Uma árvore T é dita AVL quando TODOS os nós de T forem ditos regulados;
- Um nó é dito regulado se a diferença entre as alturas das subárvore esquerda e direita for no máximo 1, em módulo;
- Note que toda árvore completa é dita AVL, mas nem toda AVL é dita uma árvore completa;
- Exs:

É AVL !!

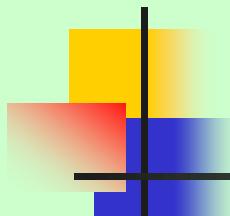


É AVL !!



NÃO AVL !!

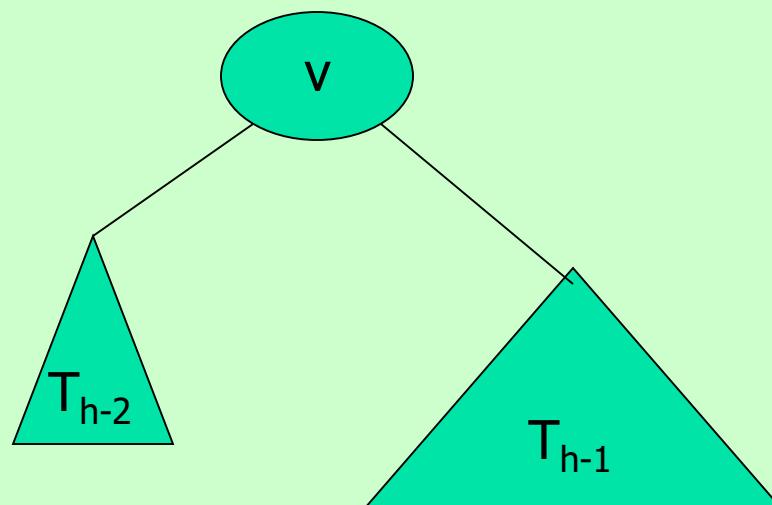


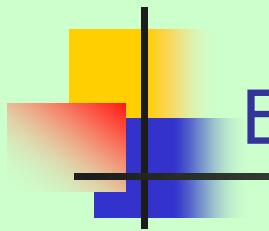


# Estrutura de Dados – Árvores Balanceadas

- **Como manter o balanceamento após a inclusão de um novo nó ???**

- **Resposta:** A cada inclusão, se existir um nó desregulado, aplicar operações de transformação para restabelecer a regulagem.
- Representação esquemática: existirá um lado com maior altura (em 1 unidade para ser AVL).

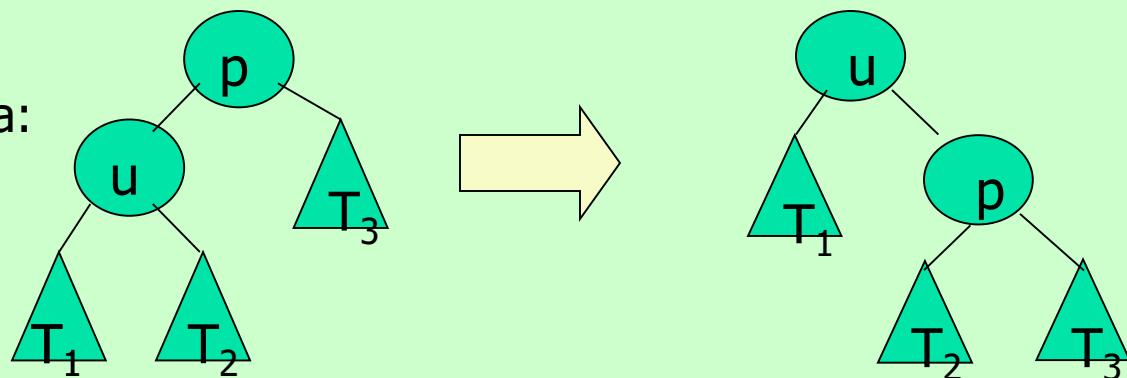




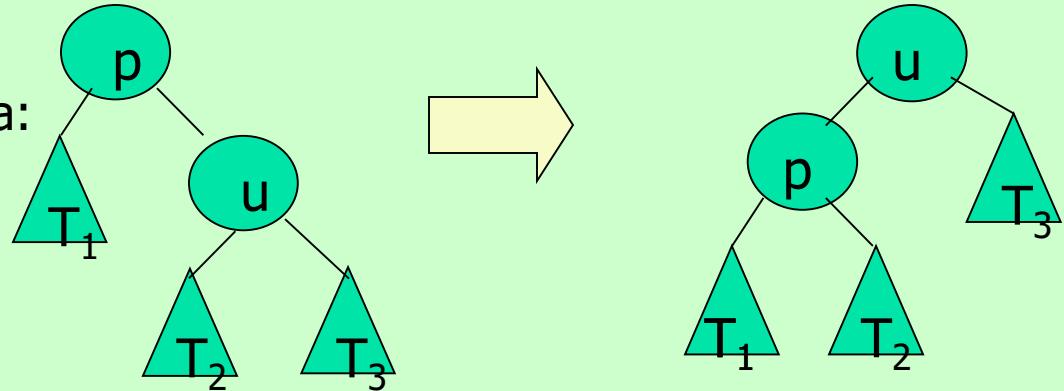
# Estrutura de Dados – Árvores Balanceadas

- Tipos de rotações válidas:

- Rotação à direita:



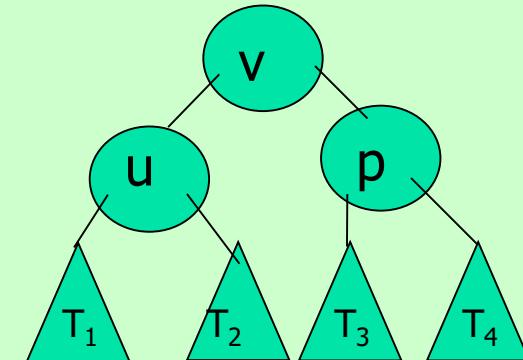
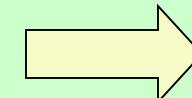
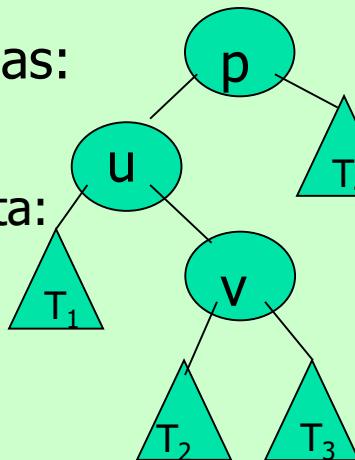
- Rotação à esquerda:



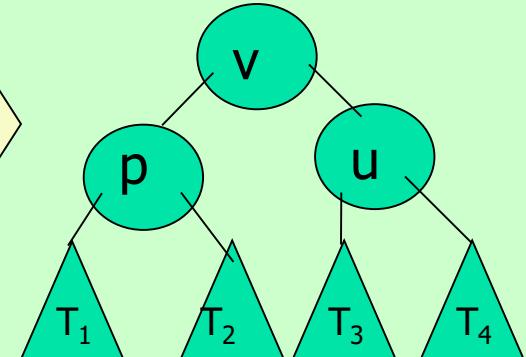
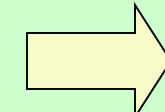
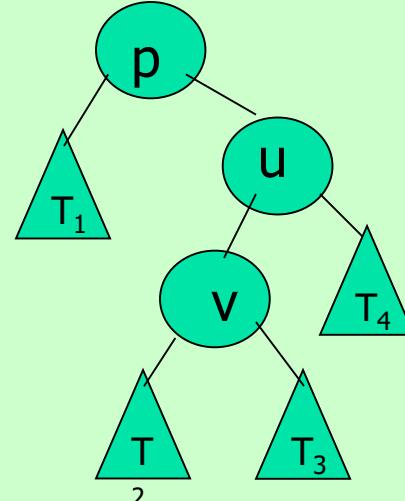
# Estrutura de Dados – Árvores Balanceadas

- Tipos de rotações válidas:

- Rotação dupla à direita:



- Rotação dupla à esquerda:

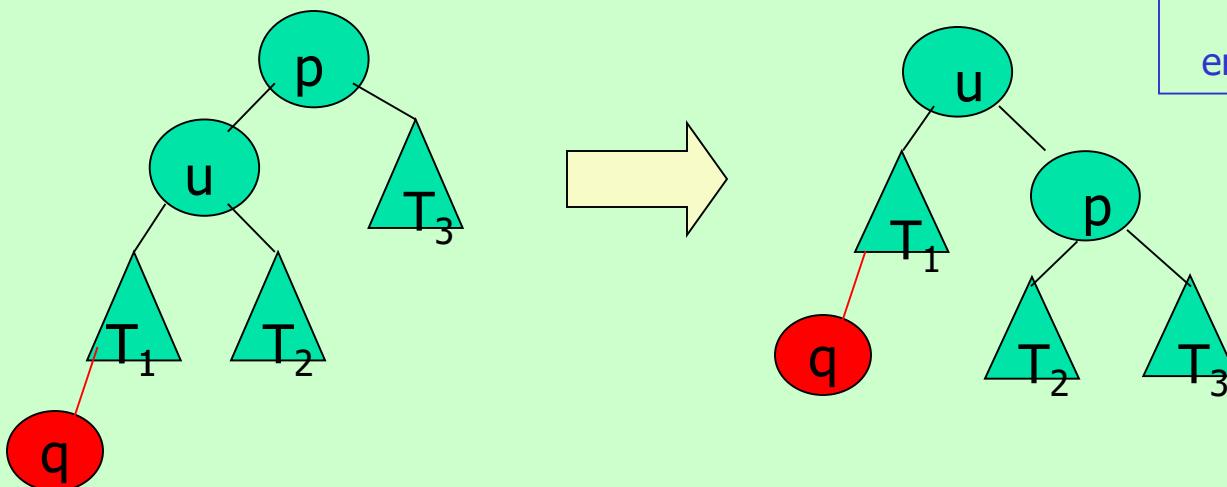


# Estrutura de Dados – Árvores Balanceadas

- 1º caso:  $h_{\text{esquedo}}(p) > h_{\text{direito}}(p)$ :

- Subcaso 1.a->  $h_{\text{esquedo}}(u) > h_{\text{direito}}(u)$ , onde u é filho esquerdo de **p**:

- $h(T_1) - h(T_2) = 1$ ;
    - $h(T_1) = h(T_3)$ ;
    - Então: aplicar rotação simples à DIREITA:



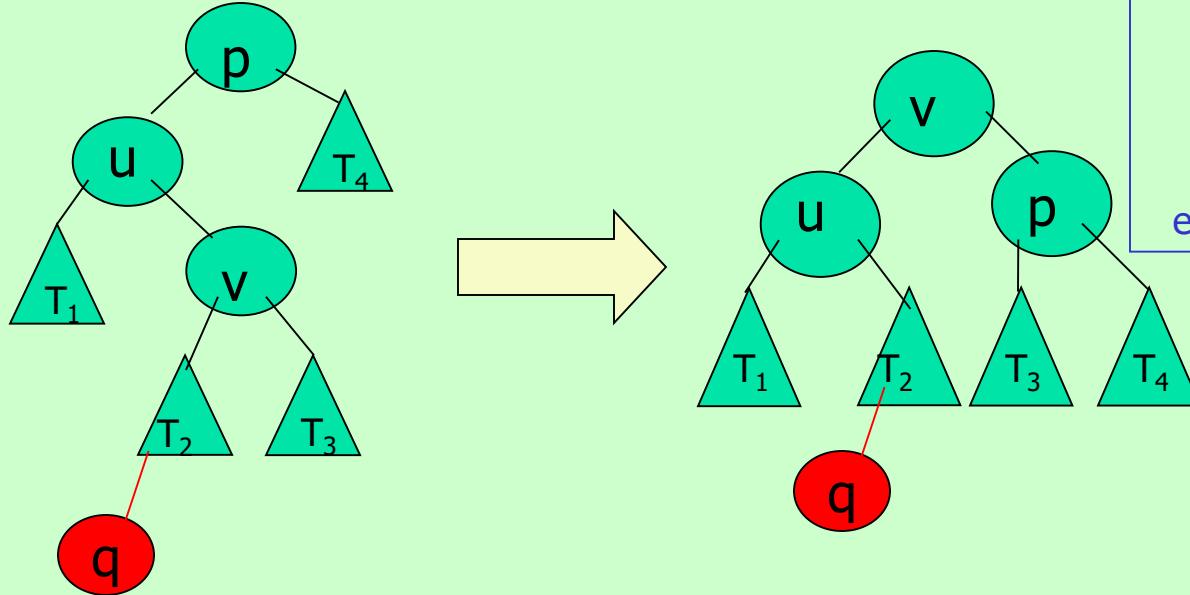
```
Rot_Simple_Direita(PTR_NO p)
begin
    u = p->esquerda;
    p->esquerda = u->direita;
    u->direita = p;
endProcedure;
```

# Estrutura de Dados – Árvores Balanceadas

- 1º caso:  $h_{\text{esquedo}}(p) > h_{\text{direito}}(p)$ :

- Subcaso 1.b->  $h_{\text{direito}}(u) > h_{\text{esquerdo}}(u)$ , onde **u** é filho esquerdo de **p**:

- $|| h(T_2) - h(T_3) || \leq 1$ ;
    - $\text{Max} \{h(T_2), h(T_3)\} = h(T_1) = h(T_4) - 1$  ;
    - Então: aplicar rotação Dupla à DIREITA:



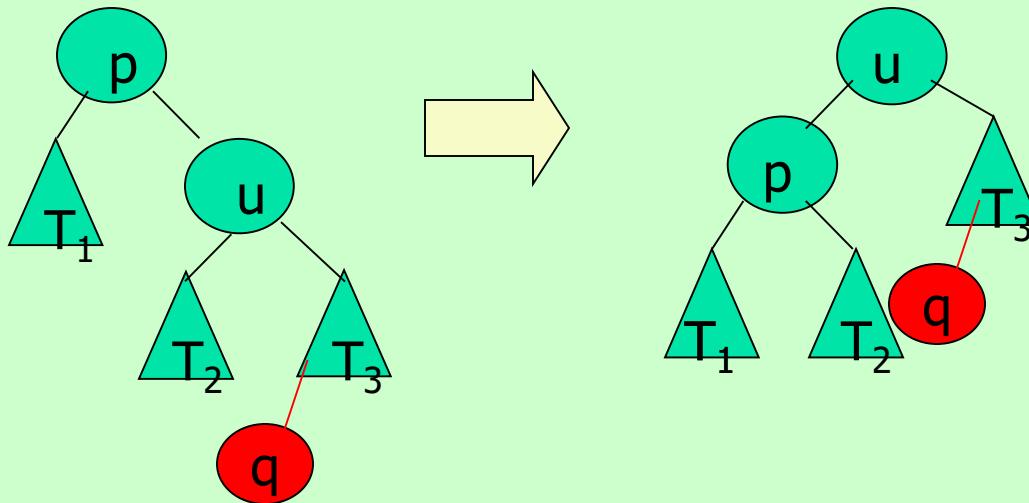
```
Rot_Dupla_Direita(PTR_NO p)
begin
    u = p->esquerda;
    v = u->direita;
    u->direita = v->esquerda;
    p->esquerda = v->direita;
    v->esquerda = u;
    v->direita = p;
endProcedure;
```

# Estrutura de Dados – Árvores Balanceadas

- 2º caso:  $h_{\text{direito}}(p) > h_{\text{esquedo}}(p)$ :

- Subcaso 2.a->  $h_{\text{direito}}(u) > h_{\text{esquedo}}(u)$ , onde u é filho direito de **p**:

- $h(T_3) - h(T_2) = 1$ ;
    - $h(T_2) = h(T_1)$ ;
    - Então: aplicar rotação simples à ESQUERDA:



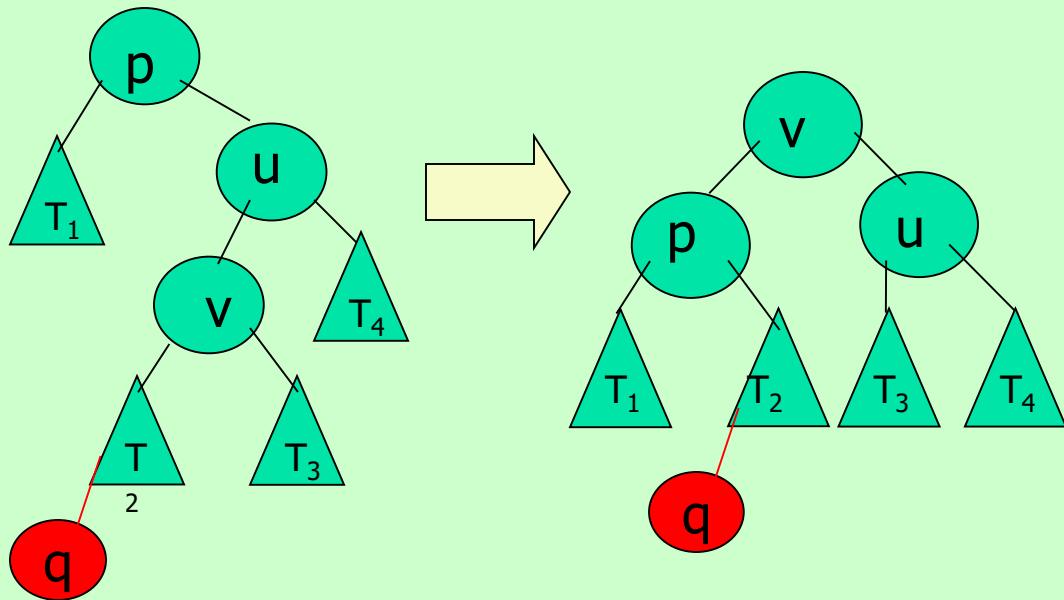
```
Rot_Simple_Esquerda(PTR_NO p)
begin
    u = p->direita;
    p->direita = u->esquerda;
    u->esquerda = p;
endProcedure;
```

# Estrutura de Dados – Árvores Balanceadas

- 2º caso:  $h_{\text{direito}}(p) > h_{\text{esquedo}}(p)$ :

- Subcaso 2.b->  $h_{\text{esquerdo}}(u) > h_{\text{direito}}(u)$ , onde u é filho direito de p:**

- $|| h(T_2) - h(T_3) || \leq 1$ ;
    - $\text{Max} \{h(T_2), h(T_3)\} = h(T_1)-1 = h(T_4)$  ;
    - Então: aplicar rotação Dupla à ESQUERDA:



```
Rot_Dupla_Esquerda (PTR_NO p)
begin
    u = p->direita;
    v = u->esquerda;
    u->esquerda = v->direita;
    p->direita = v->esquerda;
    v->esquerda = p;
    v->direita = u;
endProcedure;
```

# Estrutura de Dados – Árvores Balanceadas

```
Transforma_em_AVL(ARVORE_BINARIA_Busca Tb;)
begin
    Percuso_POS(tb->raiz);
end_Procedure;
```

```
Percuso_POS(PTR_NO no)
begin
    if (no->esquerda != nill) then
        POS(no->esquerda);
    if (no->direita != nill) then
        POS(no->direita);
    visita (no);
end_Procedure;
```

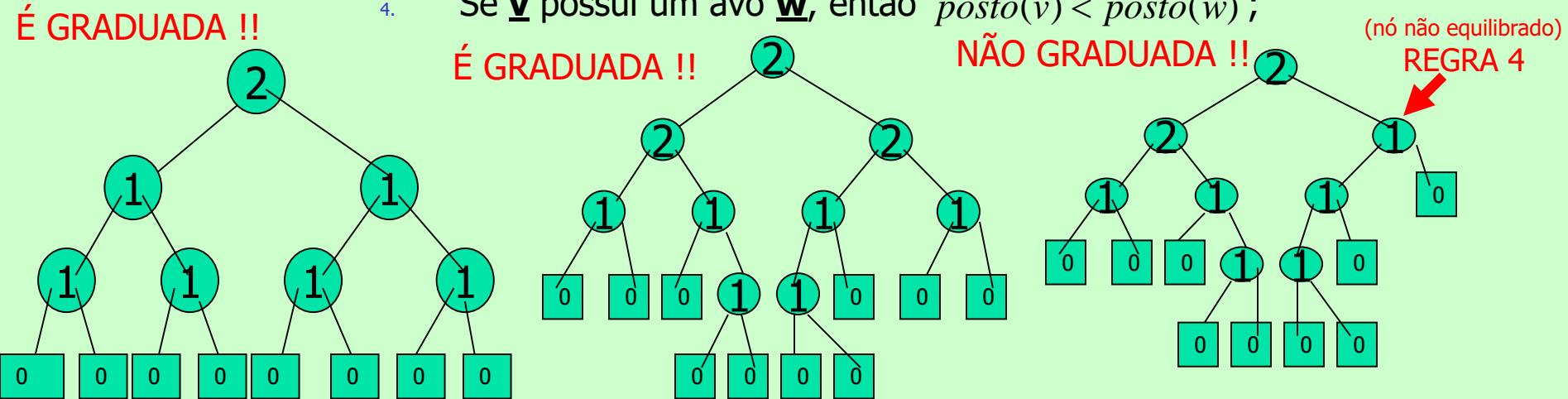
```
Visita (PTR_NO no)
begin
    if (no->esquerda != nill) then
        h1 = no->esquerda.altura;
    else
        h1=0; /* no caso de folhas */
    end_IF;
    if (no->direita != nill) then
        h2 = no->direita.altura;
    else
        h2=0; /* no caso de folhas */
    end_IF;
    if (modulo(h1 - h2) > 1) then
        Definir_Rotação(no);
        visita(no);
    else
        if ( h1 > h2) then
            no.altura = h1 + 1;
        else
            no.altura = h2 + 1;
        end_IF;
    end_Procedure;
```

```
Definir_Rotação(PTR_NO no)
begin
    if ( no->esquerda.h > no->direita.h) then
        if (no->esquerda->esquerda > no->esquerda->direita)
            then
                Rotação_Simples_Direita(no);
        else
            Rotação_Dupla_Direita(no);
        end_IF;
    else
        if (no->direita->direita > no->direita->esquerda ) then
            Rotação_Simples_Esquerda(no);
        else
            Rotação_Dupla_Esquerda(no);
        end_IF;
    end_IF;
end_Procedure;
```

# Estrutura de Dados – Árvores Balanceadas

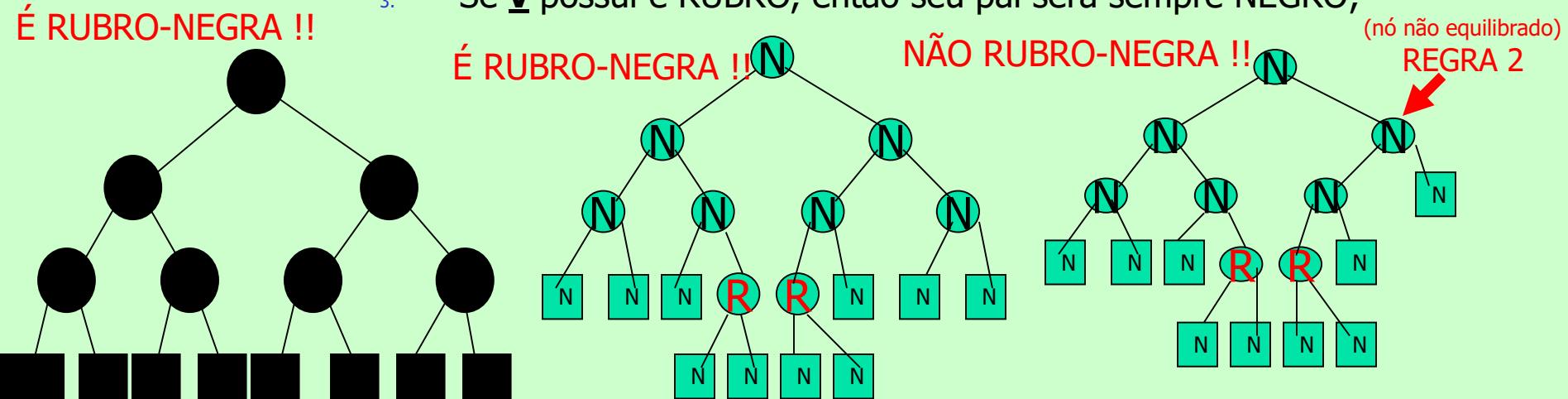
- Árvore GRADUADA:

- Uma árvore T é dita **graduada** quando o posto (rotulo de um numero inteiro) de TODOS os nó de T respeitar as seguintes heurísticas:
  1. Se no nó v é dito nó externo, então  $\text{posto}(v) = 0$ ;
  2. Se v é pai de nó externo, então  $\text{posto}(v) = 1$ ;
  3. Se v possui um pai w, então  $\text{posto}(v) \leq \text{posto}(w) \leq \text{posto}(v)+1$  ;
  4. Se v possui um avô w, então  $\text{posto}(v) < \text{posto}(w)$  ;

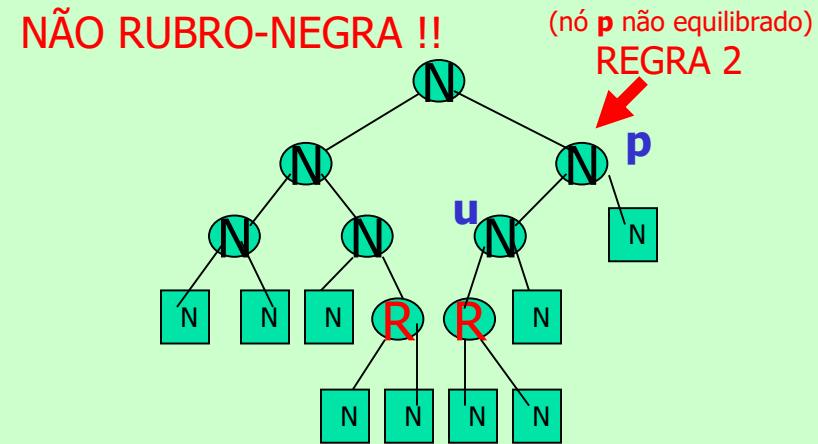
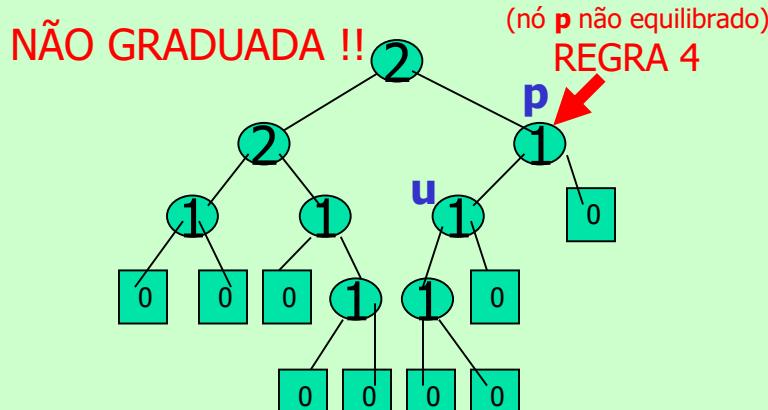


# Estrutura de Dados – Árvores Balanceadas

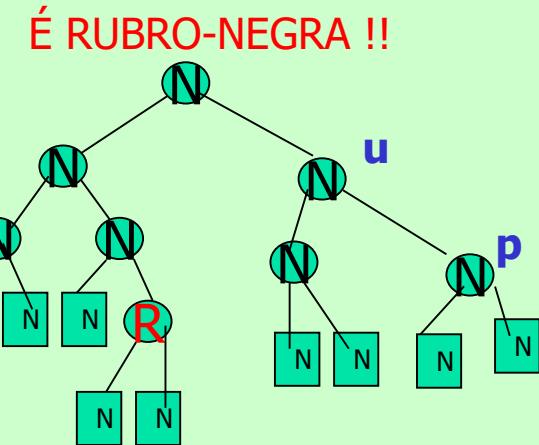
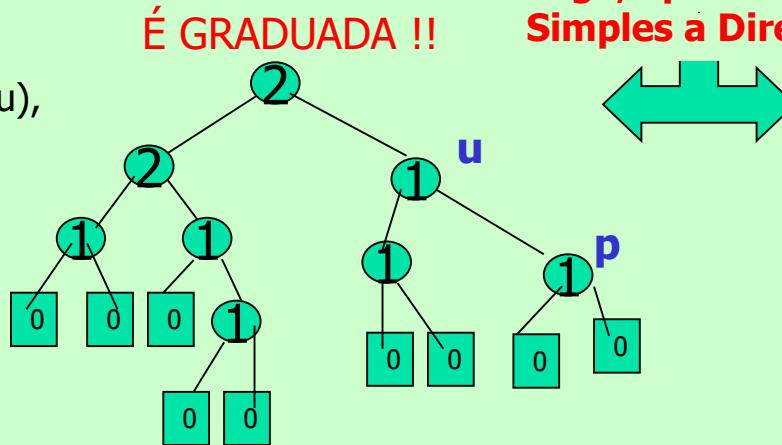
- Árvore RUBRO-NEGRA:
  - Uma árvore T é dita **RUBRO-NEGRA** quando o posto (rotulo restritivamente RUBRO ou NEGRO) de TODOS os nó de T respeitar as seguintes heurísticas:
    1. Se no nó **v** é dito nó externo, então  $\text{posto}(v) = \text{NEGRO}$ ;
    2. Os caminhos de um nó v para seus descendentes nós externos possuem idênticos números de nós NEGROS;
    3. Se **v** possui é RUBRO, então seu pai será sempre NEGRO;

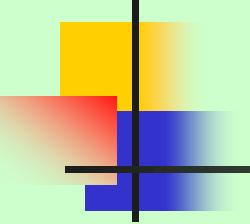


# Estrutura de Dados – Árvores Balanceadas



Logo, aplicar Rotação  
Simples a Direita





```

Posto_Filhos (PTR_NO no)
begin
  if (no->esquerda = nill) or
    (no->direita = nill) then
    posto = 1;
  if ( no.posto > posto) then
    Definir_Rotação(no);
    Visita(no);
  end_IF
else
  posto = maior ( no.posto,
                  no->esquerda.posto ,
                  no->direita.posto );
end_IF;
if ( modulo( posto - no->esquerda.posto ) > 1 ) or
  ( modulo( posto - no->direita.posto ) > 1 ) then
  Definir_Rotação(no);
  Visita(no);
else
  no.posto = posto;
end_IF;
Posto_Netos(no);
end_Procedure;

```

```

Posto_Netos (PTR_NO no)
begin
  if ( (no.posto > no->esquerda->esquerda.post ) and
      (no.posto > no->esquerda->direita.post ) and
      (no.posto > no->direita->esquerda.post ) and
      (no.posto > no->direita->direita.post ) )
    then
      return;
  else
    no.posto = no.posto + 1;
    Posto_Filhos(no);
  end_IF;
end_Procedure;

```

```

Visita (PTR_NO no)
begin
  no.posto = 0;
  if (no->esquerda != nill) then
    h1 = no->esquerda.altura;
  else
    h1=0; /* no caso de folhas */
  end_IF;
  if (no->direita != nill) then
    h2 = no->direita.altura;
  else
    h2=0; /* no caso de folhas */
  end_IF;
  Posto_Filhos(no);
  if ( h1 > h2) then
    no.altura = h1 + 1;
  else
    no.altura = h2 + 1;
  end_IF;
end_PROCEDURE;

```

```

Transforma_em_GRADUADA(ARVORE_BINARIA_Busca Tb;)
begin
  Percuso_POS(tb->raiz);
end_Procedure;

```

```

Percuso_POS(PTR_NO no)
begin
  if (no->esquerda != nill) then
    POS(no->esquerda);
  if (no->direita != nill) then
    POS(no->direita);
  visita (no);
end_Procedure;

```

# Estrutura de Dados – Árvores Balanceadas

```
Transforma_em_RUBRO-NEGRA (ARVORE_BINARIA_Busca Tb;)
begin
    Percuso_POS(tb->raiz);
end_Procedure;
```

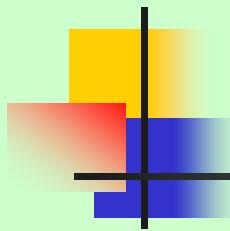
```
Percuso_POS(PTR_NO no)
begin
    if (no->esquerda != nill) then
        POS(no->esquerda);
    if (no->direita != nill) then
        POS(no->direita);
    visita (no);
end_Procedure;
```

```
Visita (PTR_NO no)
begin
    no.posto = 0;
    if (no->esquerda != nill) then
        h1 = no->esquerda.altura;
    else
        h1=0; /* no caso de folhas */
    end_IF;
    if (no->direita != nill) then
        h2 = no->direita.altura;
    else
        h2=0; /* no caso de folhas */
    end_IF;
Posto_RN(no);
if ( h1 > h2) then
    no.altura = h1 + 1;
else
    no.altura = h2 + 1;
end_IF;
end PROCEDURE;
```

# Estrutura de Dados – Árvores Balanceadas

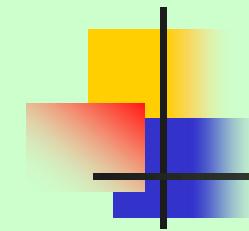
```
Posto_RN(PTR_NO no)
begin
  OK = SIM;
  if (no->esquerda = nill) or
    (no->direita = nill) then
    no.negros++;
  if ( modulo(no->esq.negros - no->dir.negros) >0 ) then
    if (no->esquerdo.negros > no->direito.negros) then
      OK = Transforma_em Rubros(no->esquerdo);
    end_IF;
    if (no->esquerdo.negros < no->direito.negros) then
      OK = Transforma_em Rubros(no->direito);
    end_IF;
  end_IF;
  if (no->esquerdo.negros > no->direito.negros) then
    OK = Transforma_em Rubros(no->esquerdo);
  end_IF;
  if (no->esquerdo.negros < no->direito.negros) then
    OK = Transforma_em Rubros(no->direito);
  end_IF;
  if (OK = NÃO) then
    Definir_Rotação(no);
    Percurso_POS(no);
  end_IF;
  no.posto = NEGRO;
  no.negro = no->esquerdo.negros + 1;
end_PROCEDURE;
```

```
LOGIC Transforma_em Rubros(PTR_NO no)
begin
  OK = SIM;
  if ( no = nill) then return (OK = NÃO);
  if ( no.posto = NEGRO) then
    no.posto = RUBRO;
    no.negros = no.negros - 1;
  else
    OK = Transforma_em Rubros(no->esquerdo);
    if ( OK = SIM ) then OK = Transforma_em Rubros(no->direito);
    if (OK = SIM) then no.negros--;
  endif;
  if (no.posto = RUBRO and no->esquerdo = RUBRO) then return (OK = NÃO);
  if (no.posto = RUBRO and no->direito = RUBRO) then return (OK = NÃO);
  if ( no->esquerdo.posto = RUBRO) then
    no->esquerdo.posto = NEGRO;
    no->esquerdo.negros++;
    OK =Transforma_em Rubros(no->esquerdo->esquerdo);
    OK = Transforma_em Rubros(no->esquerdo->direito);
  end_IF;
  if ( no->direito.posto = RUBRO) then
    no->direito.posto = NEGRO;
    no->direito.negros++;
    OK = Transforma_em Rubros(no->direito->esquerdo);
    OK = Transforma_em Rubros(no->direito->direito);
  end_IF;
  return(OK);
end_Procedure;
```



# Estrutura de Dados – Árvores Balanceadas

- Árvore-B
  - Problema:
    - Quando o número de chaves é alto, ficando impossível de se armazenar na memória principal. Com isso, os acessos serão feitos diretamente sobre a memória secundária, o que requer um alto custo (tempo) computacional.
  - SOLUÇÃO:
    - FAZER USO DE UMA ÁRVORE-B PARA MINIMIZAÇÃO DO TEMPO;

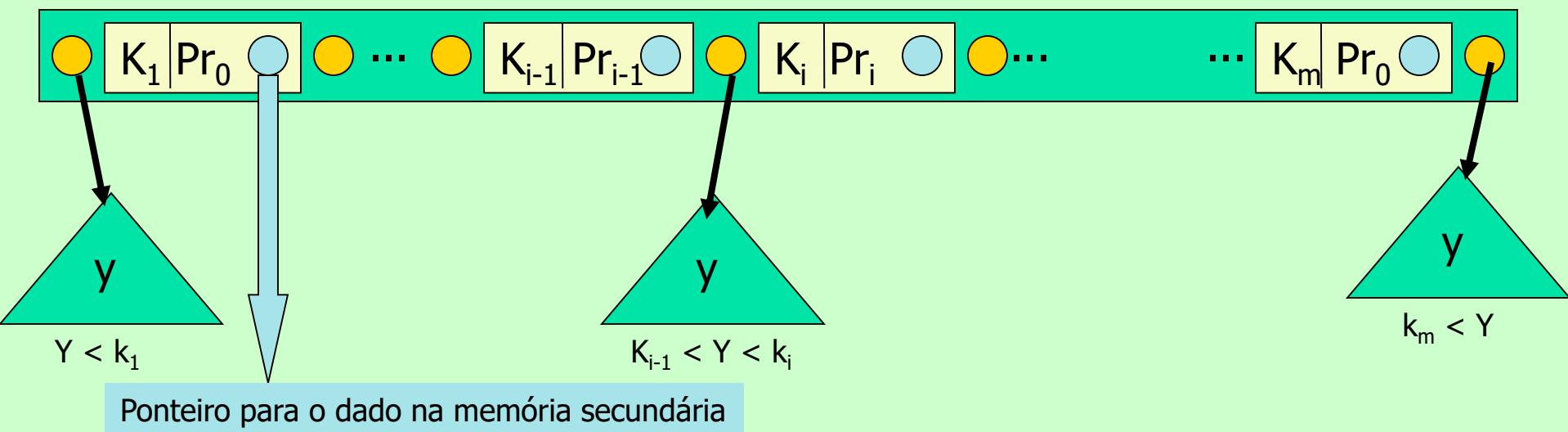


# Estrutura de Dados – Árvores Balanceadas

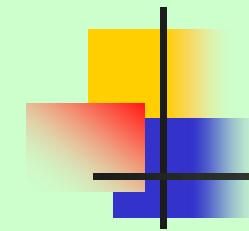
- Características:
  1. Cada nó armazena mais de uma chave;
  2. Acesso a memória secundária é otimizado devido ao uso de ponteiros. Logo, a árvore-B possui uma organização por ponteiros;
  3. Todas as folhas estão em um mesmo nível;
  4. A raiz é um nó que tem no mínimo 2 filhos;
  5. Cada nó interno (não-raiz e não-folha) possui no mínimo **d+1** filhos, onde **d** é a ordem da árvore;
  6. Cada nó tem no máximo  $2d+1$  filhos;

# Estrutura de Dados – Árvores Balanceadas

7. Um nó da árvore-B define uma página, que é onde ficam armazenados os diversos nós que estão na memória secundária:



- Características:**
1.  $m = \text{número de chaves ordenadas na pagina } P \Rightarrow (k_1 < k_2 < \dots < k_{m-1} < k_m)$ ;
  2.  $m+1 = \text{número de filhos de } P$ ;
  3. Página não-raiz:  $d-1 < m < 2d+1$ ;
  4. Página raiz:  $0 < m < 2d$ ;



# Estrutura de Dados – Árvores Balanceadas

8. Limites do número de páginas e de elementos, sendo  $n$  o número total de elementos da tabela ( $n > 0$ ),  $d$  a ordem ( $d > 0$ ),  $h$  a altura da árvore-B ( $h > 0$ ):

1. Número mínimo de páginas:  $p_{\min} = 1 + \frac{2}{d} [(d+1)^{h-1} - d];$

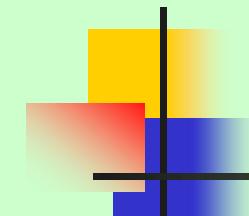
2. Número máximo de páginas:  $p_{\max} = \sum_{k=0}^{h-1} (2d+1)^k = \frac{1}{2d} [(2d+1)^h - 1];$

3. Número mínimo e máximo de elementos da tabela (arquivo secundário) dispostos na árvore-B:

$$\eta_{\min} = 2(d+1)^{h-1} - 1; \quad \eta_{\max} = (2d+1)^h - 1;$$

4. Variação da altura da árvore-B:

$$\log_{2d+1}(n+1) \leq h \leq \log_{d+1}\left(\frac{n+1}{2}\right)$$

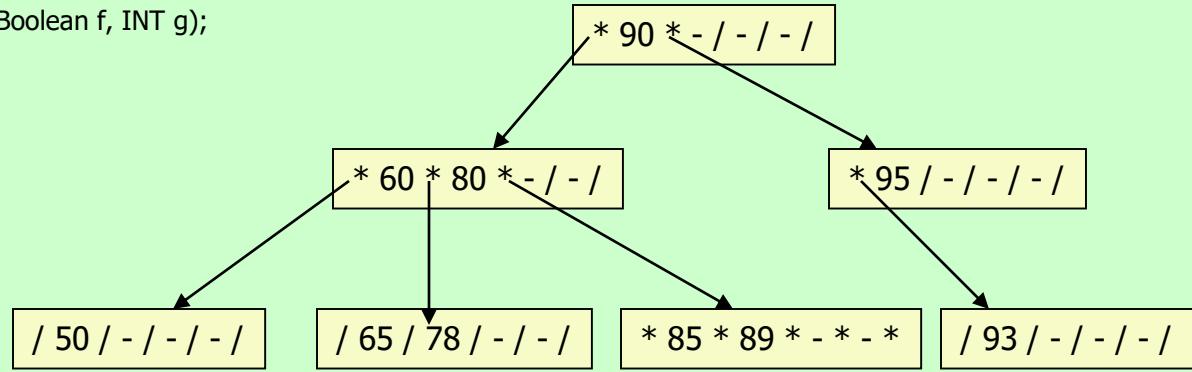


# Estrutura de Dados – Árvores Balanceadas

- Busca em árvore-B:
  - IDÉIA: buscar a chave x, usando um processo semelhante à busca em árvore binária de busca;
  - Base:
    - x = chave procurada;
    - pt = é o ponteiro resultado da busca;
    - f = flag = 1 = chave encontrada;  
= 0 = chave não encontrada;
    - g = posição da página apontada por pt que contem x;
    - m = número de chaves na página;
  - OBS:
    1. Se f=0, então pt aponta para folha e g informa a posição na página pt onde x será incluída;
    2. Set Define Structure{  
    Int m;  
    PTR\_SECUNDÁRIO chaves[1 ...m];  
    PTR\_PAGINA pags[0 ... m];  
} PAGINA;

# Estrutura de Dados – Árvores Balanceadas

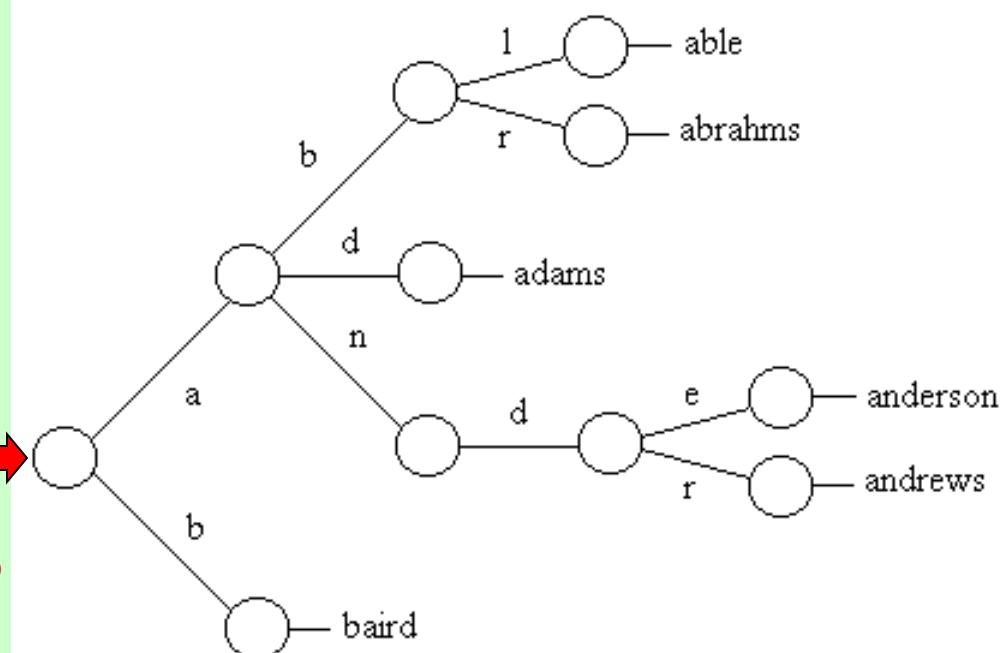
```
Busca_árvore-B ( PTR_SECINDÁRIO x, PTR_PAGINA pt, Boolean f, INT g);
begin
  PTR_PAGINA pt, p;
  INT i;
  p=pt; /* aponta para o raiz da árvore-B */
  f=0;
  while ( p != nill ) do
    g=i=1;
    pt=p;
    while ( i <= p.m ) do
      if ( x > p->chave[i] ) then
        g = i = i+1;
      else
        if ( x = p->chave[i] ) then
          p = nill;
          f = 1;
        else
          p = p->pags[i-1];
        end_IF
      i = m+2; /* força sair do segundo while */
    end_IF;
  end_WHILE;
  if ( i = m+1) then p = p->pags[m]; /* garante o acesso ao último filho da pagina */
end_WHILE;
end PROCEDURE;
```



# Estrutura de Dados – TRIES (Information reTRIEvel – recuperação da informação)

- TRIE = *radix searching tree => busca por prefixos;*
- é uma árvore de busca na qual o grau do nó, ou número máximo de filhos por nó, é igual ao número de símbolos do alfabeto.
- EX: construir uma trie para armazenar as chaves 'able', 'abrahms', 'adams', 'anderson', 'andrews', 'baird'

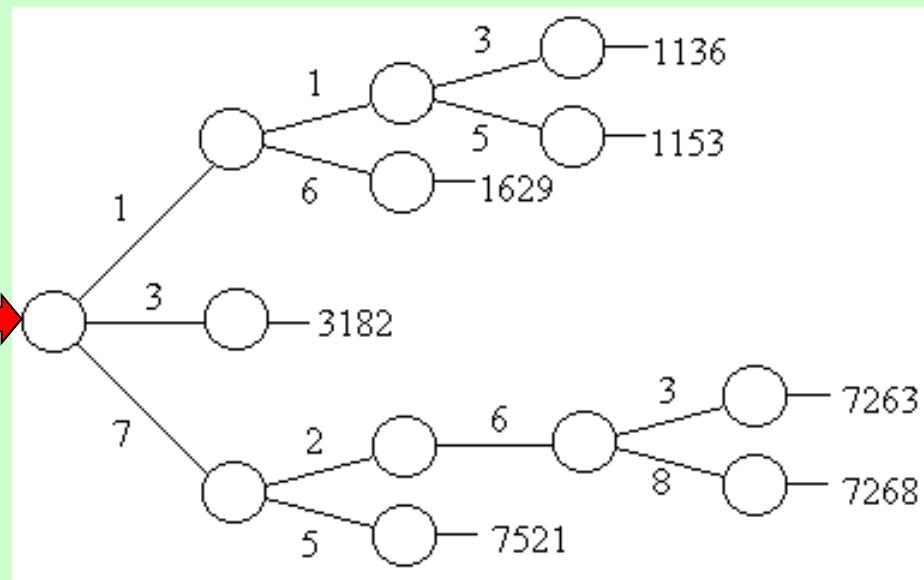
A idéia é que a busca prossegue letra por letra ao longo da chave



OBS: radix 26 = numero de letras do alfabeto

## Estrutura de Dados – TRIES (Information reTRIEvel – recuperação da informação)

- EX: construir uma trie para armazenar as chaves '1136', '1629', '1153', '3182', '7263', '7268', '7524'

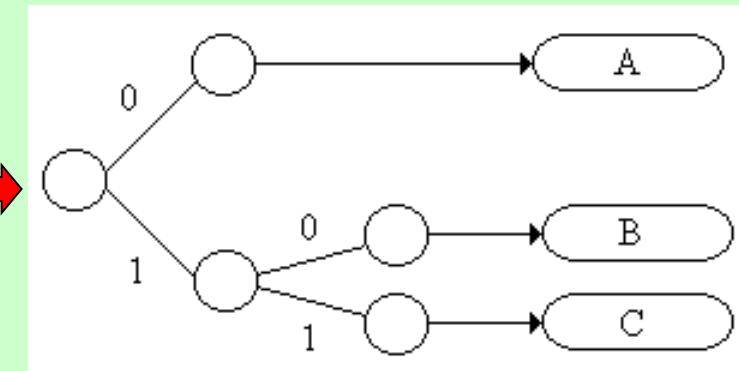


OBS: radix 10 = numero de dígitos

## Estrutura de Dados – TRIES (Information reTRIEvel – recuperação da informação)

- EX: construir uma trie para tomadas de decisões em jogos

A idéia é que a busca prossegue  
Por decisões de cada jogador  
ao longo da chave

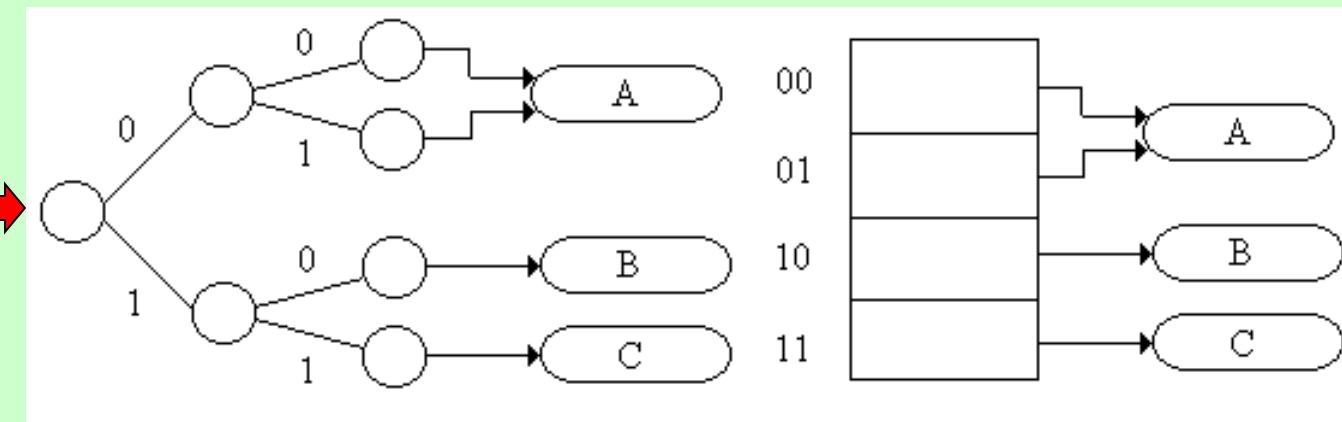


OBS: radix 2 = numero de decisões (sim/não);

## Estrutura de Dados – TRIES (Information reTRIEvel – recuperação da informação)

- EX: construir uma trie para o endereçamento de informações na memória (similar ao hash) – usar uma árvore cheia

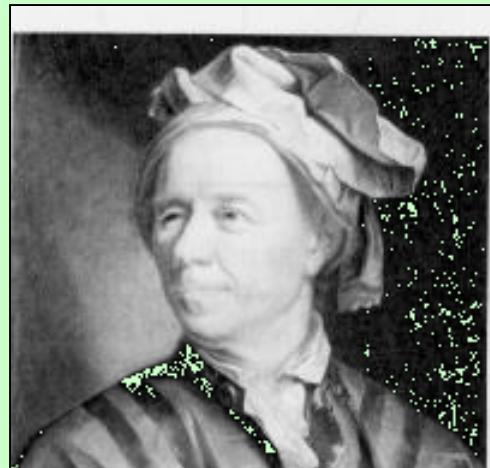
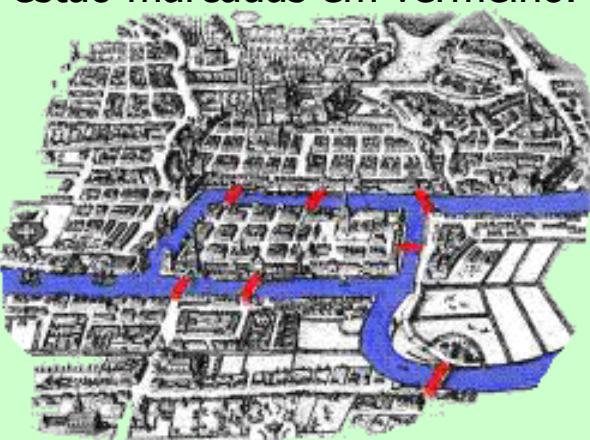
A idéia é que a busca Binário do endereçamento



OBS: radix 2 = código binário 1 / 0

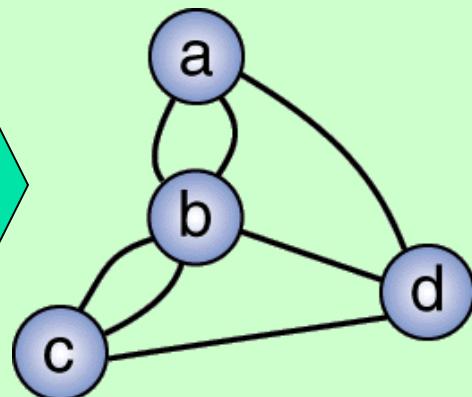
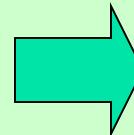
# Estrutura de Dados – Grafos

- Historia do grafo:
  - Primeiro modelo: Leonhard Euler
- Mapa de Konigsberg onde as pontes estão marcadas em vermelho.



**Problema sobre a cidade:**

*Walk around the city,  
crossing each bridge  
exactly once while ending  
where you started*

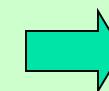
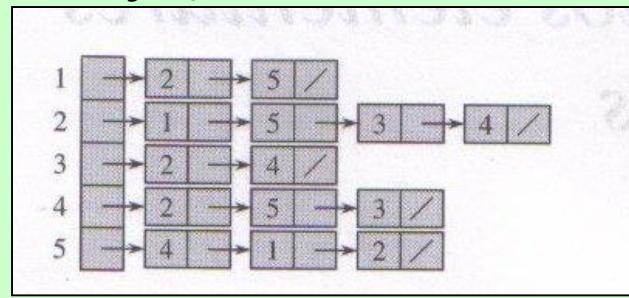
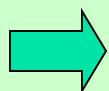
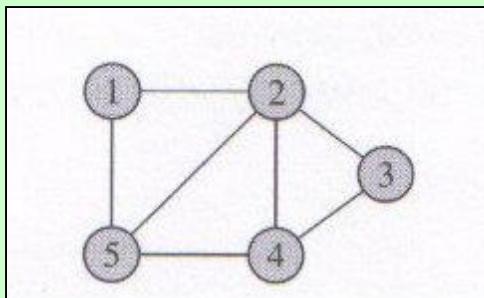


**Modelagem matemática:**  
primeira definição de um grafo

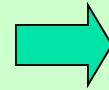
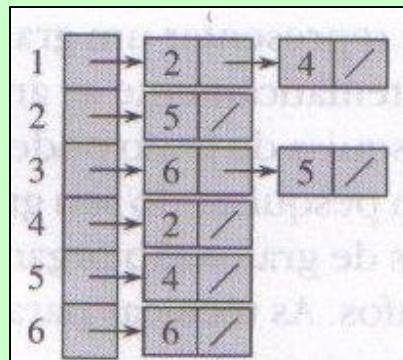
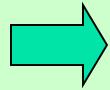
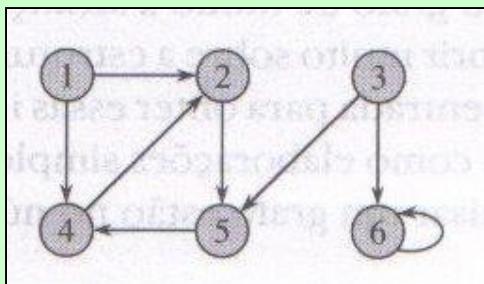
# Estrutura de Dados – Grafos

- Grafos:

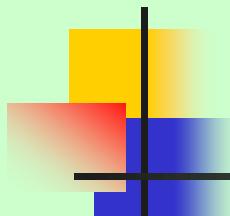
- $G=(V, E) = (\text{vértices}, \text{arestas})$ ;
- Estrutura de dados que descreve as adjacências de um nó (VERTICE) a partir da especificação das arestas vizinhas;
- Define-se vizinhanças, orientabilidade e análise topológica;



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



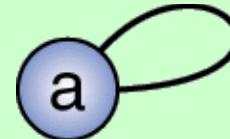
1	2	3	4	5	6
1	0	1	0	1	0
2	0	0	0	0	1
3	0	0	0	0	1
4	0	1	0	0	0
5	0	0	0	1	0
6	0	0	0	0	1



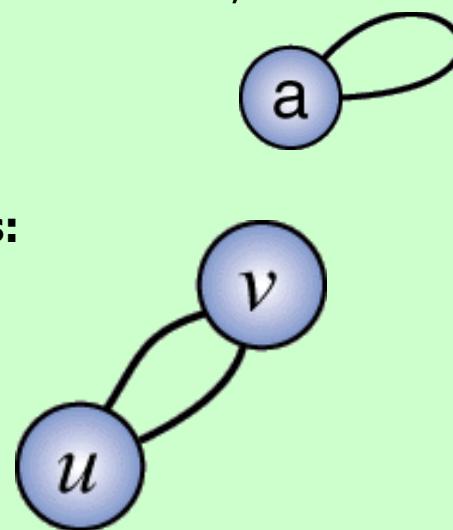
# Estrutura de Dados – Grafos

- Terminologia:

- **Grafo de Euler:** possui um caminho que passa por todas as arestas exatamente uma única vez, e termina no vértice de inicio;
- **Grau de um vértice** = numero de arestas vizinhas ao vértice;
  - Ex: grafo de Euler=> grau (a) = 3;
- **Loop** = aresta( $u,v$ ) onde  $v = u$ ;



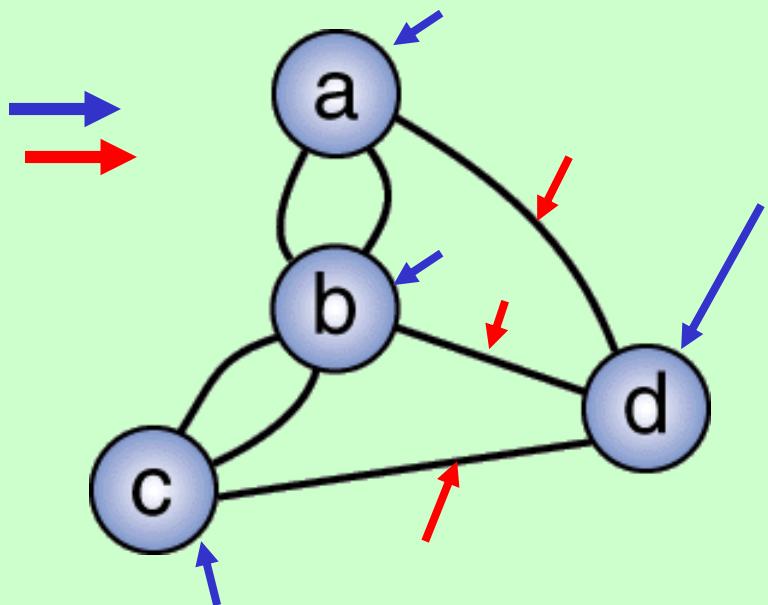
- **Múltiplas arestas:**



# Estrutura de Dados – Grafos

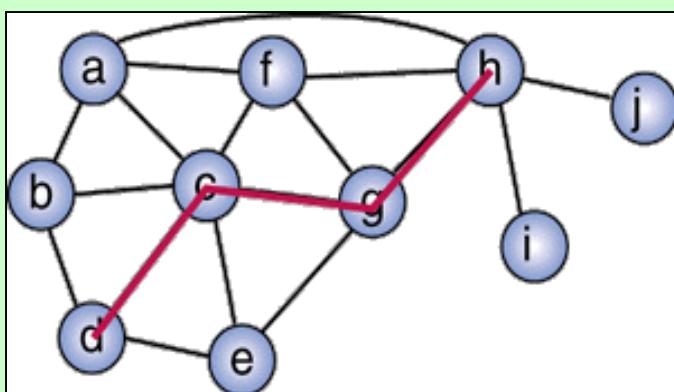
- **Adjacência:**

- De vértice (possuem arestas vizinhas);
- De arestas (possuem vértices vizinhos);



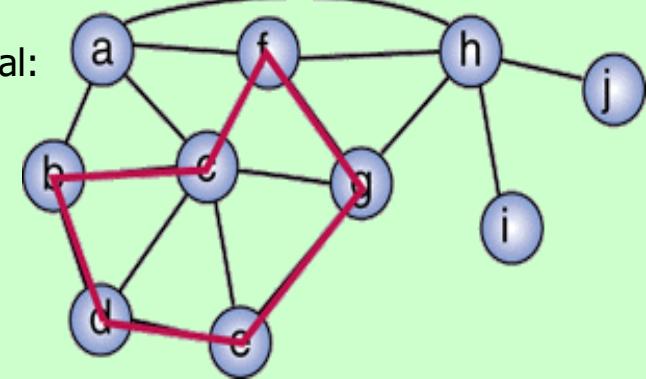
- **Caminho:**

- Seqüência de vértices: d c g h



**Ciclo:**

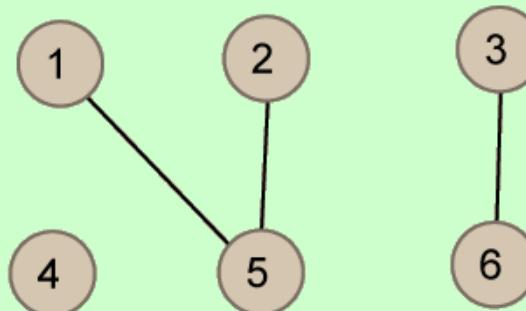
caminho com  $V_{\text{initial}} = V_{\text{final}}$ :



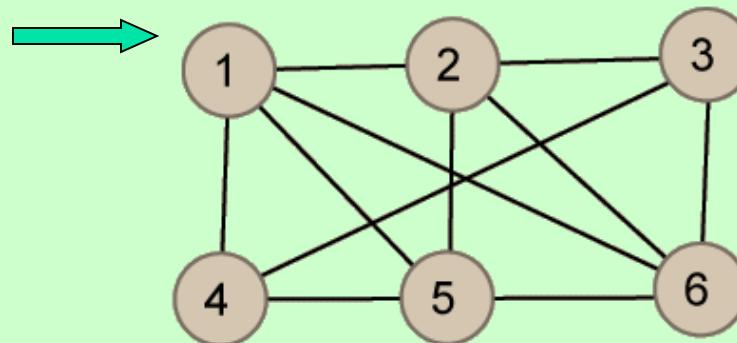
# Estrutura de Dados – Grafos

- Tipos de grafos:

- **Simples**: possui um numero finito de V e E;
- **Acíclico**: não formam ciclos;
- **Esparsos**:  $E \ll V^2$  

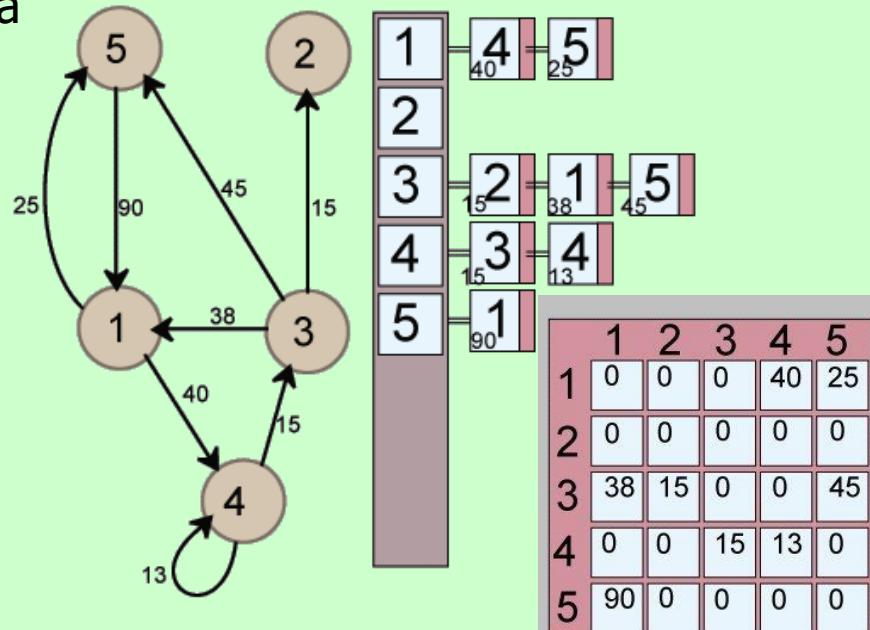
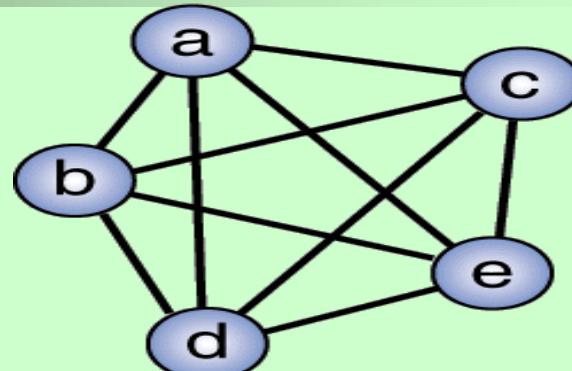


- **Denso**: quase todos os vértices são conectados por uma aresta



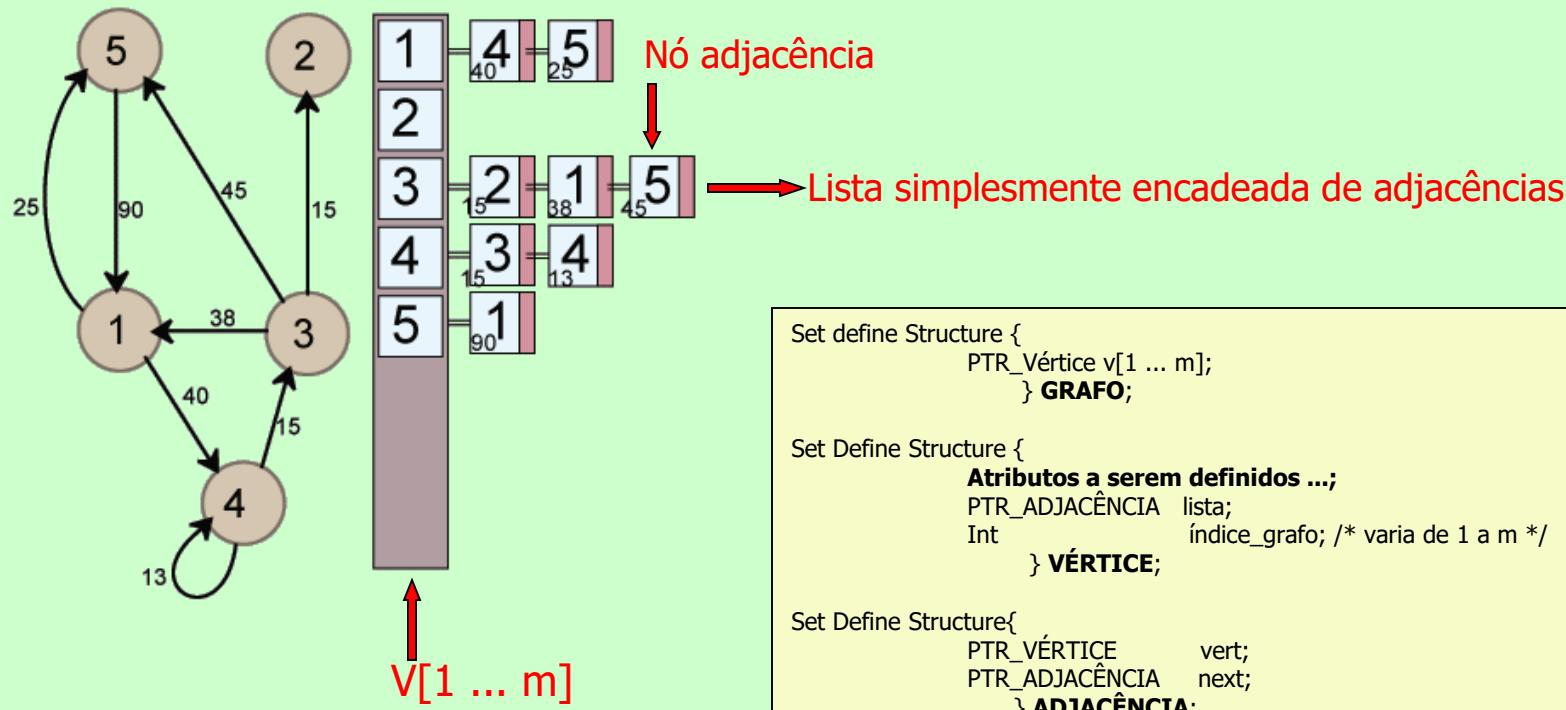
# Estrutura de Dados – Grafos

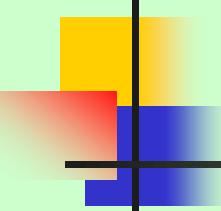
- **Completo:** para qualquer par de vértices, haverá uma aresta.
- **Grafos ponderados:** cada aresta tem um peso definido por uma função-peso  $W(u,v)$  para cada  $E(u,v)$ ;
- **Grafo Orientado:** existe uma orientação única aplicada a uma aresta;



# Estrutura de Dados – Grafos

- Representação:





# Estrutura de Dados – Grafos

- Pesquisa primeiro em extensão:

- **OBJETIVO:**

- Para um dado vértice, obtém-se entre suas arestas, o cálculo da distância (menor numero de arestas) até TODOS os vértices acessíveis. Obtém-se o caminho mínimo para todos os vértices de um grafo a partir de um vértice v qualquer pertencente a  $G(V,E)$ ;

- **Idéia:**

- 1. Definir uma cor ao vértice, onde :

-  -> descoberto pela primeira vez;

-  -> quando todos os vértice vizinhos já foram descobertos (são ao menos brancos);

-  -> quando nem todos os vértice vizinhos já foram descobertos (fase intermediária);

- 2. Cada vértice terá uma distância já calculada;
    - 3. Cada predecessor de um vértice u (seu pai) é armazenado em pai;
    - 4. Usar uma fila para controlar os vértices cinzas (fila\_cinza);

# Estrutura de Dados – Grafos

Busca\_Primeiro em Extensão ( Grafo G, INT vertex)

```
begin
    PTR_ADJACÊNCIA fila_cinza; /* fila referente aos vértices de cor cinza */
    PTR_ADJACÊNCIA adj; /* fila referente aos vértices de cor cinza */
    For i = 1 to m
        grafo. v[i].cor= BRANCO;
        grafo. v[i].distância = 10000000;
        grafo. v[i].pai= nill;
    end_FOR;
    grafo.v[vertex].cor = CINZA;
    grafo.v[vertex].distância = 0;
    grafo.v[vertex].pai = nill;
    alloc (adj);
    adj->vert = grafo.v[vertex];
    adj->next = nill;
    Inserir_na_Fila (fila_cinza ,adj); /* inserir primeiro nó-cinza da fila que estava vazia */
    While (fila_cinza != VAZIA)
        u = Tirar_elemento_FILA (fila_cinza );
        v = grafo. v[ u->vert.índice_grafo ].lista;
        while ( v != nill )
            if v->vert.cor = BRANCO then
                v->vert.cor = CINZA;
                v->vert.distância = u->vert.distância + 1;
                v->vert.pai = u->vert;
                alloc(adj);
                adj->vert = v->vert;
                Inserir_na_Fila (fila_cinza ,v);
            end_IF;
            v = v->next;
        end_WHILE;
        u->vert.cor = PRETO;
    end_WHILE;
end_PROCEDURE;
```

```
Set define Structure {
    PTR_Vértice v[1 ... m];
} GRAFO;
```

```
Set Define Structure {
    COLORAÇÃO          cor;
    Int                 distância;
    PTR_VÉRTICE         pai;
    PTR_ADJACÊNCIA     lista;
    Int                 índice_grafo; /* varia de 1 a m */
} VÉRTICE;
```

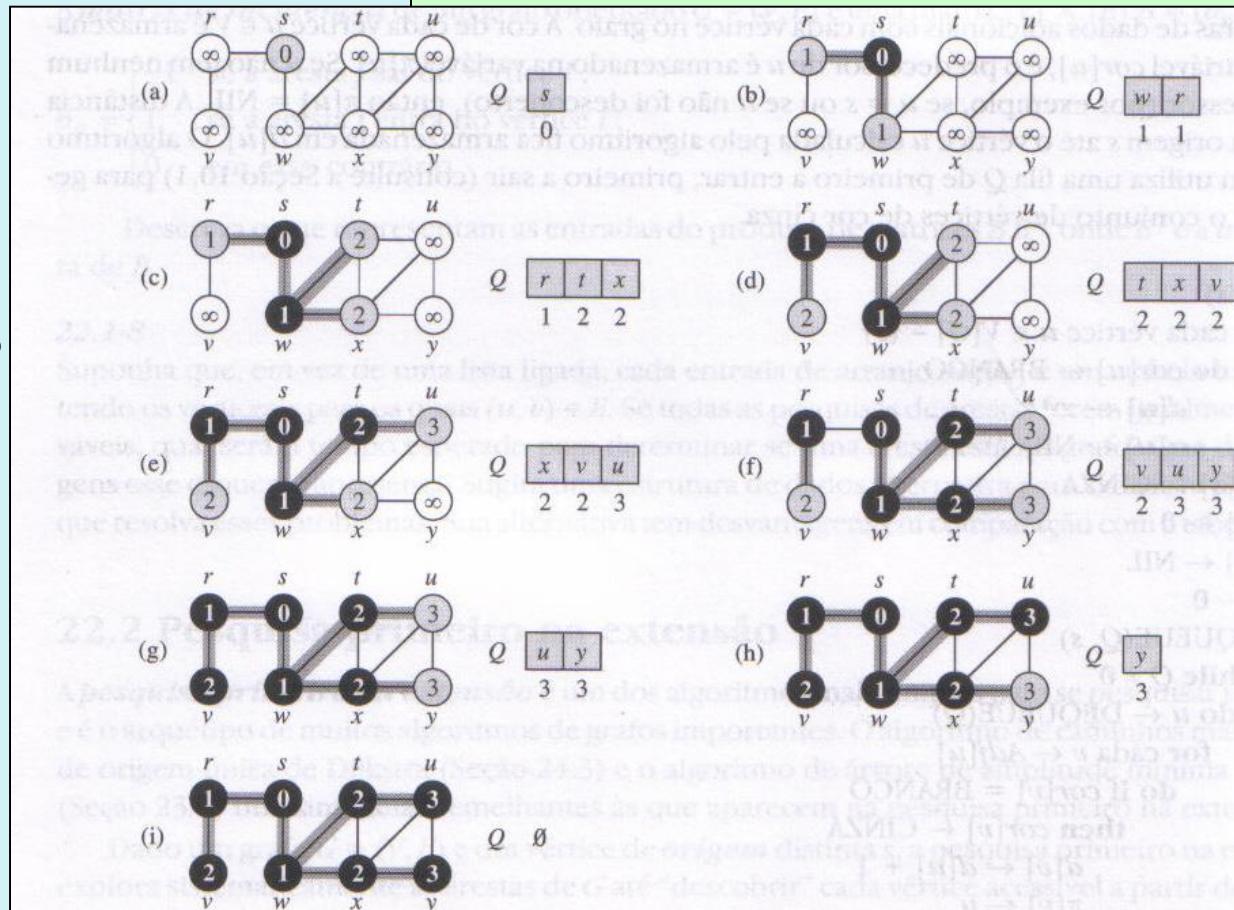
```
Set Define Structure{
    PTR_VÉRTICE         vert;
    PTR_ADJACÊNCIA     next;
} ADJACÊNCIA;
```

# Estrutura de Dados – Grafos

Busca\_Primero em Extensão ( Grafo G, INT vertex)

```

begin
PTR_ADJACÊNCIA fila_cinza; /* fila referente aos vértices de cor cinza */
PTR_ADJACÊNCIA adj; /* fila referente aos vértices de cor cinza */
For i = 1 to m
    grafo. v[i].cor= BRANCO;
    grafo. v[i].distância = 10000000;
    grafo. v[i].pai= nill;
end_FOR;
grafo.v[vertex].cor = CINZA;
grafo.v[vertex].distância = 0;
grafo.v[vertex].pai = nill;
alloc (adj);
adj->vert = grafo.v[vertex];
adj->next = nill;
Inserir_na_Fila (fila_cinza ,adj); /* inserir primeiro
While (fila_cinza != VAZIA)
    u = Tirar_elemento_FILA (fila_cinza );
    v = grafo. v[ u->vert.índice_grafo ].lista;
    while ( v != nill )
        if v->vert.cor = BRANCO then
            v->vert.cor = CINZA;
            v->vert.distância = u->vert.distância + 1;
            v->vert.pai = u->vert;
            alloc(adj);
            adj->vert = v->vert;
            Inserir_na_Fila (fila_cinza ,v);
        end_IF;
        v = v->next;
    end_WHILE;
    u->vert.cor = PRETO;
end_WHILE;
end PROCEDURE;
```



# Estrutura de Dados – Grafos

- Pesquisa primeiro em profundidade (Deep First Search - DFS):
  - **Objetivo:**
    - procurar “mais fundo” no grafo, sempre que possível;
  - **Idéia:**
    1. Explorar as arestas a partir do vértice **v** mais recentemente descoberto, e que ainda possui arestas inexploradas saindo dele;
    2. Após TODAS as arestas terem sido exploradas, a pesquisa regressa para explorar as arestas que deixam o vértice **v** descoberto. Isso se repete para TODOS os vértices acessíveis de **v**;
    3. Restando algum vértice ainda não explorado, inicia-se (1) para o novo vértice inexplorado.
  - **OBS:** o algoritmo faz uso de 2 carimbos de tempo (time-stamp):
    1. Carimbo D, que registra o tempo quando o vértice v é **descoberto** (assume cor cinza);
    2. Carimbo F, que registra o tempo quando a pesquisa **finalizou** (assume cor preta)
  - Então:
    1.  $D(v) < F(v)$ ;
    2. 
    3. 
    4. 

# Estrutura de Dados – Grafos

## Busca\_Primero em Profundidade ( Grafo G)

```
begin
PTR_ADJACÊNCIA fila_cinza; /* fila referente aos vértices de cor cinza */
PTR_ADJACÊNCIA adj; /* fila referente aos vértices de cor cinza */
For i = 1 to m
    grafo. v[i].cor= BRANCO;
    grafo. v[i].pai= nill;
end_FOR;
tempo = 0;
For i = 1 to m
    if grafo. v[i].cor= BRANCO then;
        DFP_Visita(grafo. v[i] , tempo);
    end_IF
end PROCEDURE;

DFP_Visita( PTR_VÉRTICE u, INT tempo)
begin
    u.cor = CINZA;
    tempo ++:
    u.D = tempo;
    v = grafo. v[ u->vert.índice_grafo ].lista;
    while ( v != nill )
        if v->vert.cor = BRANCO then
            v->vert.pai = u->vert;
            DFP_Visita ( v , tempo);
        end_IF;
        v = v->next;
    end_WHILE;
    u->vert.cor = PRETO;
    tempo ++;
    u->vert.F = tempo;
end PROCEDURE;
```

```
Set define Structure {
    PTR_Vértice v[1 ... m];
} GRAFO;
```

```
Set Define Structure {
    COLORAÇÃO      cor;
    Int             D; /* time stamp inicio */
    Int             F; /* time stamp fim */
    PTR_ADJACÊNCIA lista_adjacentes;
    Int             índice_grafo; /* varia de 1 a m */
} VÉRTICE;
```

```
Set Define Structure{
    PTR_VÉRTICE      vert;
    PTR_ADJACÊNCIA   next;
} ADJACÊNCIA;
```

# Estrutura de Dados – Grafos

## Busca\_Primero em Profundidade ( Grafo G)

```

begin
PTR_ADJACÊNCIA fila_cinza; /* fila referente aos vértices de cor cinza */
PTR_ADJACÊNCIA adj; /* fila referente aos vértices de cor cinza */
For i = 1 to m
    grafo. v[i].cor= BRANCO;
    grafo. v[i].pai= nill;
end_FOR;
tempo = 0;
For i = 1 to m
    if grafo. v[i].cor= BRANCO then;
        DFP_Visita(grafo. v[i] , tempo);
    end_IF
end PROCEDURE;
```

## DFP\_Visita( PTR\_VÉRTICE u, INT tempo)

```

begin
    u.cor = CINZA;
    tempo ++:
    u.D = tempo;
    v = grafo. v[ u->vert.índice_grafo ].lista;
    while ( v != nill )
        if v->vert.cor = BRANCO then
            v->vert.pai = u->vert;
            DFP_Visista ( v , tempo);
        end_IF;
        v = v->next;
    end WHILE;
    u->vert.cor = PRETO;
    tempo ++;
    u->vert.F = tempo;
end PROCEDURE;
```

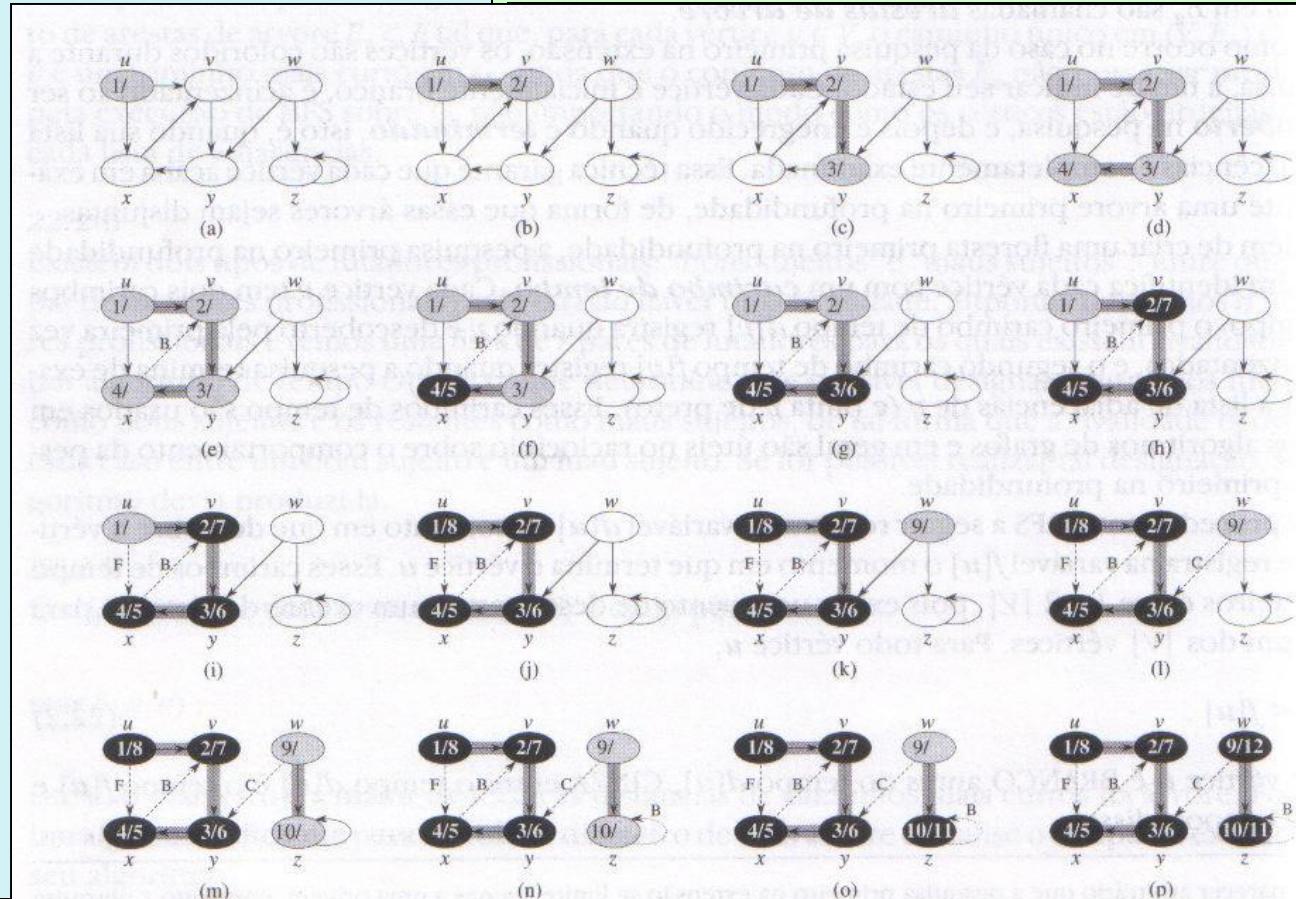
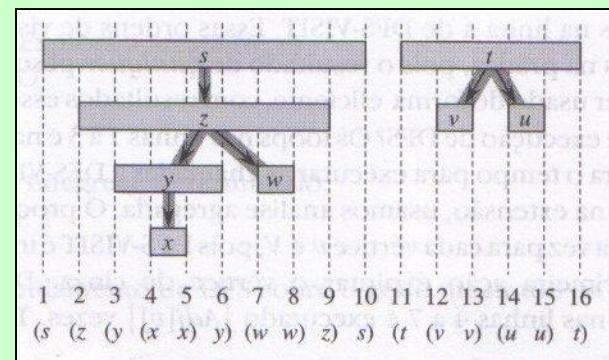
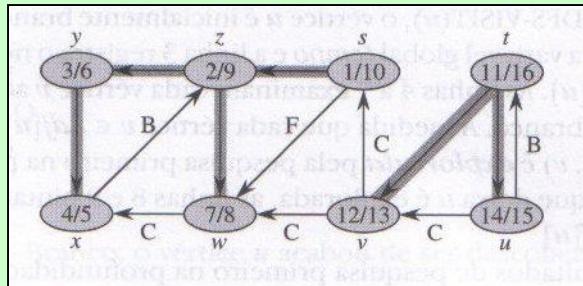


FIGURA 3.26 - O procedimento de busca DFP\_Visita é executado no grafo G da figura 1.

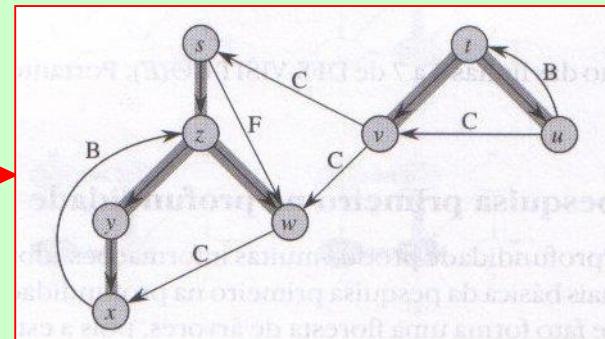
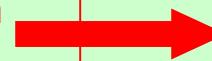
# Estrutura de Dados – Grafos

- Propriedade da Pesquisa Primeiro em Profundidade:

Os tempos especificados nos time-stamps D e F, descrevem intervalos que especificam árvores disjuntas (florestas);



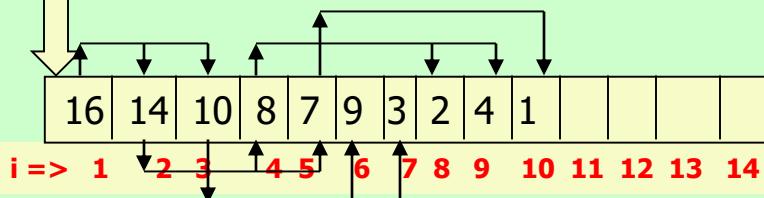
Redefinição gráfica do grafo,  
respeitando-se o esquema  
DFP para descendência



# Estrutura de Dados – Heap

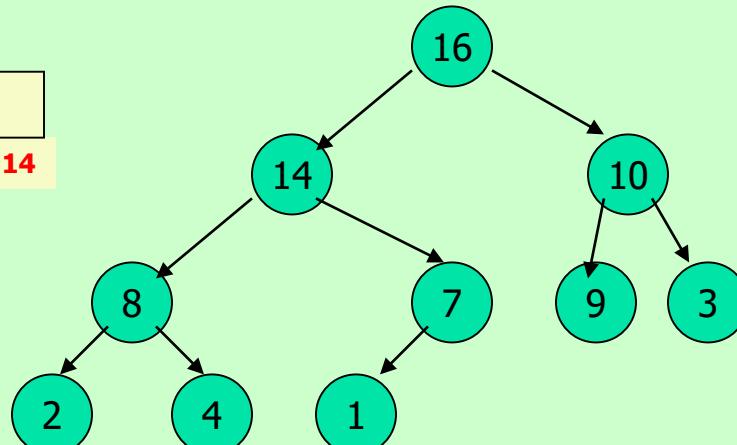
```
Set Define Structure{  
    Int comprimento,  
    Int tamanho;  
    Chave dado[1....m]  
} HEAP;
```

HEAP { comprimento = 14 ;  
tamanho = 10 }



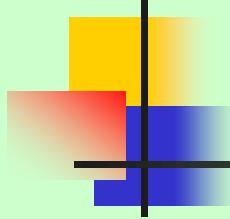
## HEAP (monte):

- Usado para criar uma lista de prioridades atribuída a cada elemento;
- É usado em processo de ordenação heap-sort => ordenação  $O(n \log n)$ ;
- Usa array como estrutura de dados, onde são informados o comprimento do array e tamanho do heap (ocupação do array);
- Pode ser visto como uma árvore binária praticamente completa:



OBS: para qualquer endereço **i** do array,  
sabe-se:

1. Pai(i) = maior-inteiro( $i/2$ );
2. Filho-esquerdo(i) =  $2i$ ;
3. Filho-direito(i) =  $2i + 1$ ;



# Estrutura de Dados – Heap

---

- Tipos de Heaps:

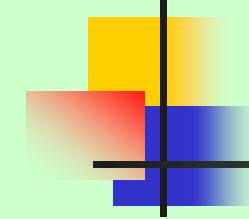
1. Heaps máximos (raiz é o mais alto elemento):

- $\text{heap.dado[ pai(i) ]} \geq \text{heap.dado[ i ]};$

2. Heaps mínimos (raiz é o mais baixo elemento) :

- $\text{heap.dado[ pai(i) ]} \leq \text{heap.dado[ i ]};$

OBS: para a ordenação(heap-sort) deve ser usado heaps máximos !!



# Estrutura de Dados – Heap

- Operação Heapify (transformar em heap):
  - Assegura que a propriedade heap não é violada;
  - Usado quando um dos elementos do heap é alterado;
  - Usado para construir um heap (converte uma array em um heap);
  - Usado para o merge de 2 heaps de mesmo tamanho, tendo um novo nó como o raiz para os 2 merging-heaps;

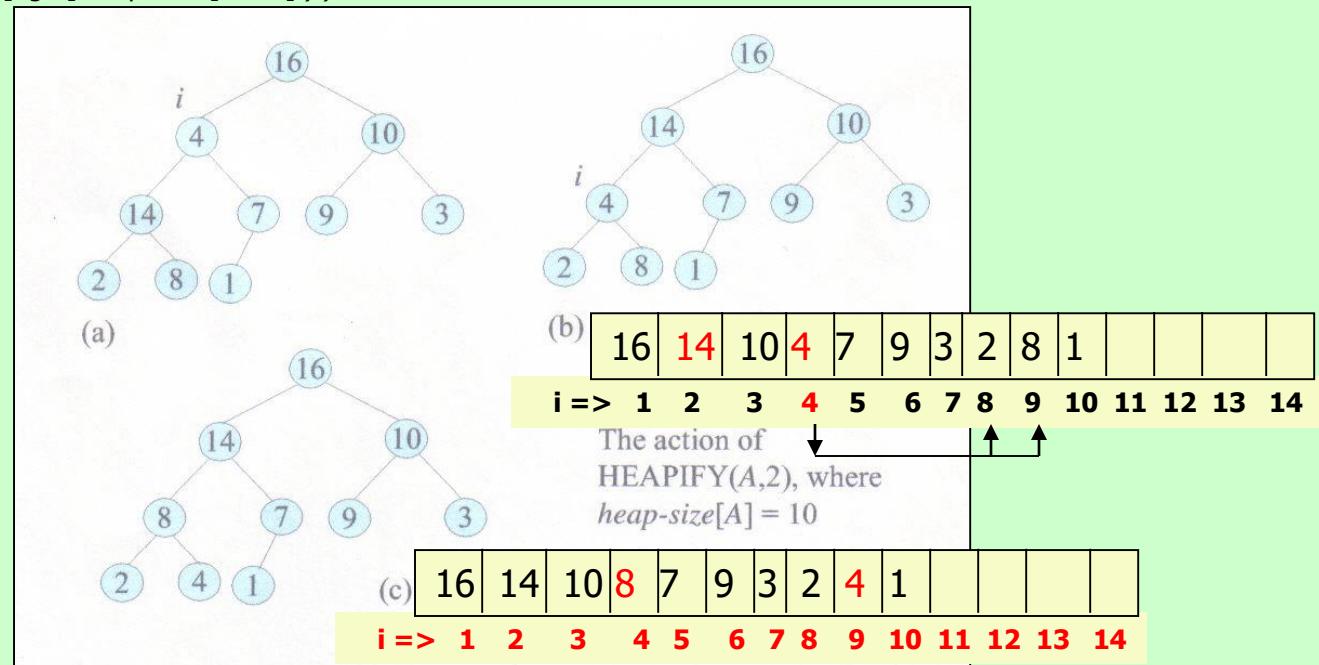
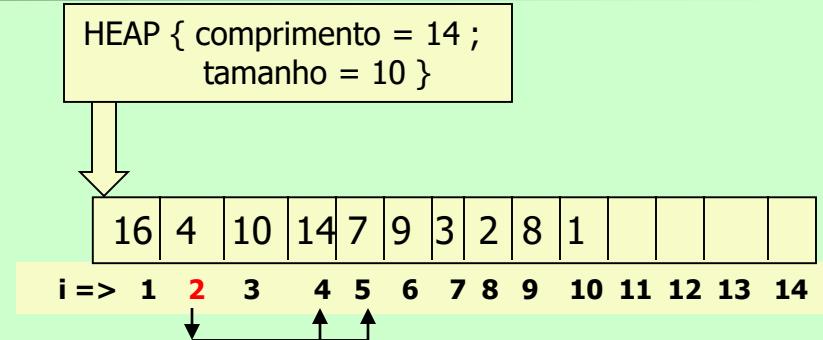
# Estrutura de Dados – Heap

```

Max_Heapify( Heap hp, Int i)
begin
    left = 2 * i;
    right = 2 * i + 1;
    if ( ( left <= hp.tamanho) and ( hp.dado[left] > hp.dado[i] ) ) then
        maior = left;
    else
        maior = i;
    end_IF;
    if ( ( right <= hp.tamanho) and ( hp.dado[right] > hp.dado[maior] ) ) then
        maior = right;
    end_IF;
    if ( maior != i ) then
        apoio = hp.dado[i];
        hp.dado[i] = hp.dado[maior];
        hp.dado[maior] = apoio;
        Max_Heapify( hp , maior );
    end_IF;
end_PROCEDURE;

```

HEAP { comprimento = 14 ;  
tamanho = 10 }

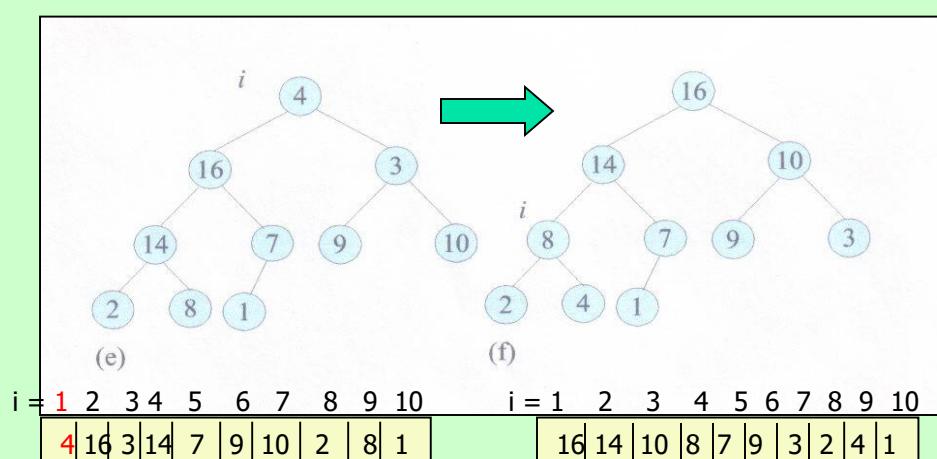
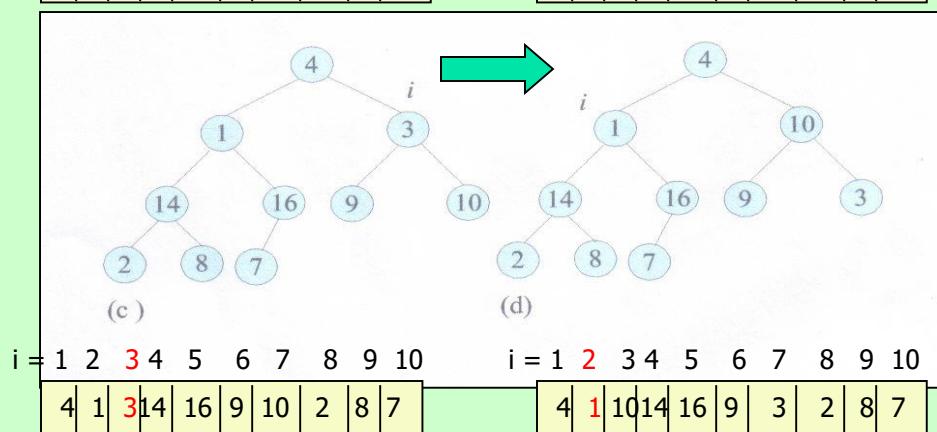
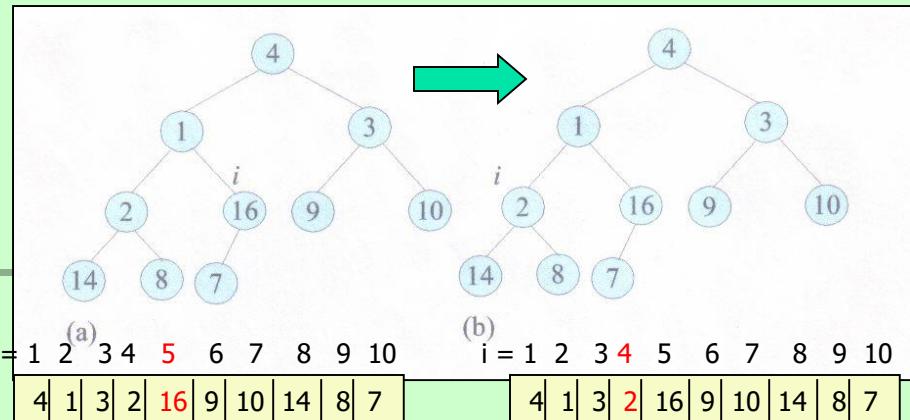


# Estrutura de Dados

```

PTR_Heap Construir_Max_Heap( Chave dado[1..m])
begin
    PTR_HEP hp;
    alloc(hp);
    comprimento = 0;
    while dado[comprimento] <> "" ) do
        hp.dado[comprimento] = dado[comprimento];
        comprimento++;
    end WHILE;
    comprimento++;
    hp.tamanho = comprimento;
    for i = Menor_Inteiro( hp.tamanho / 2) downto 1 do
        Max_Heapify( hp , i);
    end FOR;
    return(hp);
End PROCEDURE;

```



# Estrutura de Dados – Heap

**Heap\_Sort( Chave dado[1...m])**

```

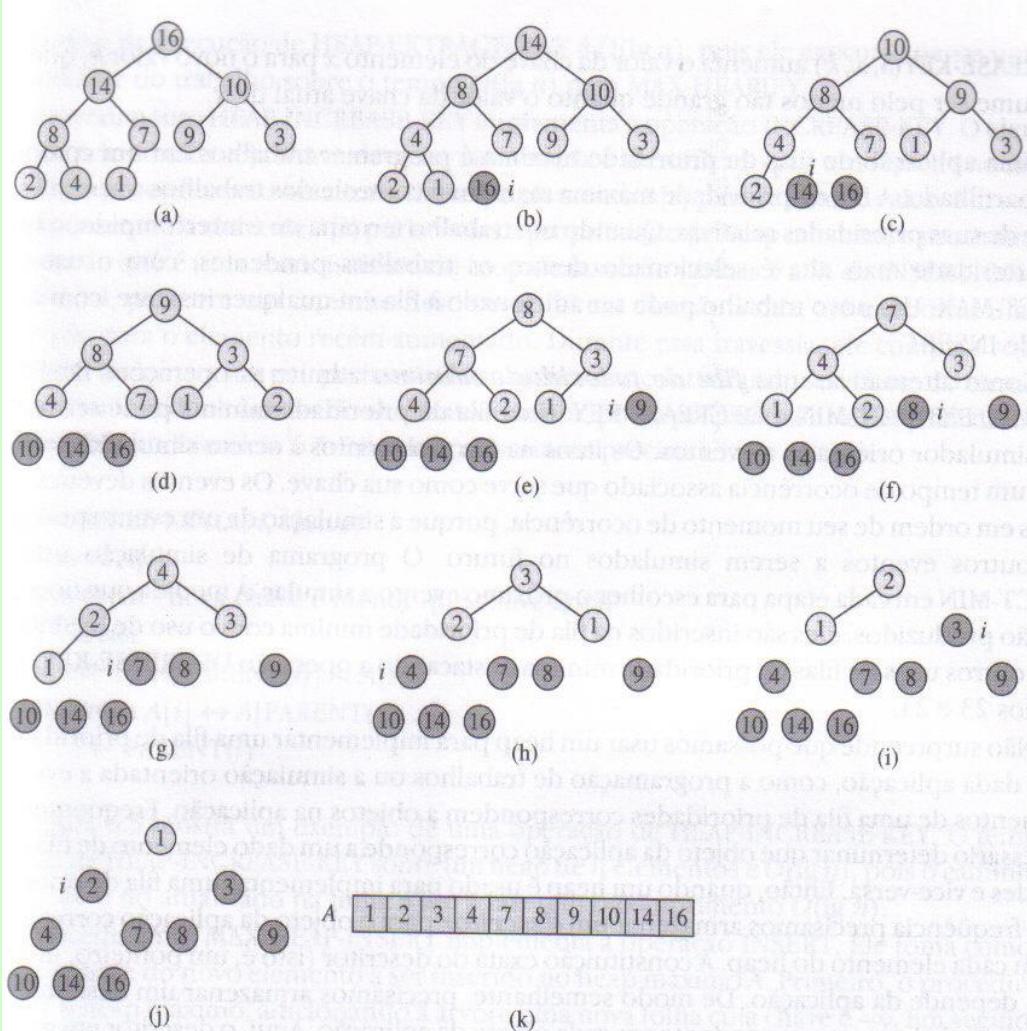
begin
    hp = Construir_Max_Heap(dado[1...m]);
    for i = hp.tamanho downto 2 do
        apoio = hp.dado[1];
        hp.dado[1] = hp.dado[i];
        hp.dado[i] = apoio;
        hp.tamanho --;
        Max_Heapify(hp,1)
    end_FOR
end PROCEDURE;
```

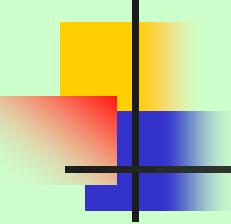
## ANÁLISE:

1. Construir\_Max\_heap() =  $O(n)$ ;
2.  $N-1$  Chamadas Max\_Heapfy() =
 
$$= (n-1) O(\log n);$$
3. **Heap\_Sort()** =  $O(n) + (n-1) O(\log n) =$ 

$$= O(n + (n-1) \log n) =$$

$$= O(n \log n);$$





# Estrutura de Dados – Fila de Prioridade

- Fila de Prioridade:
  - É uma estrutura de dados (um heap) para a manutenção de um conjunto **S** de elementos, onde cada elemento possui um valor associado chamado **chave**;
  - Um uso importante é para o controle de trabalhos em um computador compartilhado, onde cada trabalho possui um nível de prioridade diferenciado;
  - Operações:
    1.  $\text{INSERT}(S,x)$  = inserir nova chave **x** na lista **S**;
    2.  $\text{INCREASE\_KEY}(S,x,k)$  = aumenta o valor da prioridade de **x** para o novo valor **k**, presumindo-se que **k** é ao menos tão grande quanto **x**;
    3.  $\text{MAXIMUM}(S)$  = retorna o elemento de **S** com maior chave (maior prioridade);
    4.  $\text{EXTRACT\_MAX}(S)$  = remove e retorna o elemento de **S** com maior chave;

# Estrutura de Dados – Fila de Prioridade

```
Heap_INSERT_KEY(Heap hp , Chave ch)
```

```
begin
  hp.tamanho++;
  hp.dado[ hp.tamanho ] = -10000000;
  Heap_INCREASE_KEY( hp, hp.tamanho , ch);
end_PROCEDURE;
```

```
Heap_INCREASE_KEY ( Heap hp , Int i , Chave ch)
```

```
begin
  if chave < hp.dado[i] then
    Printf( ` erro: nova chave é menor que a atual !! ` );
  else
    hp.dado[ i ] = ch;
    pai = MENOR_INTEIRO (i / 2);
    while ( i > 1 ) and ( hp.dado [ pai ] < hp.dado [ i ] ) do
      apoio = hp.dado [ i ];
      hp.dado [ i ] = hp.dado [ pai ];
      hp.dado [ pai ] = apoio;
      i = pai;
      pai = MENOR_INTEIRO (i / 2);
    end_WHILE;
  end_IF;
end_PROCEDURE
```

```
Chave Heap_MAXIMUM( Heap hp)
```

```
begin
  return (hp.dado [ 1 ]);
end_PROCEDURE
```

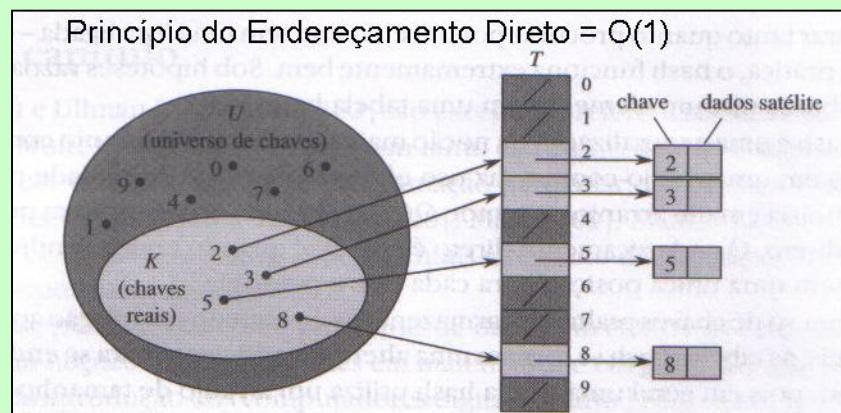
```
Chave Heap_EXTRACT_MAX( heap hp )
```

```
begin
  Chave max;
  if hp.tamanho < 1 then
    printf( `erro: heap underflow` );
  else
    max = hp.dado [ 1 ];
    hp.dado[ 1 ] = hp.dado [ hp.tamanho ];
    hp.tamanho--;
    MAX_HEAPIFY(hp,1);
  end_IF;
  return (max);
end_PROCEDURE;
```

# Estrutura de Dados – Tabela Hash

- Tabela Hash:

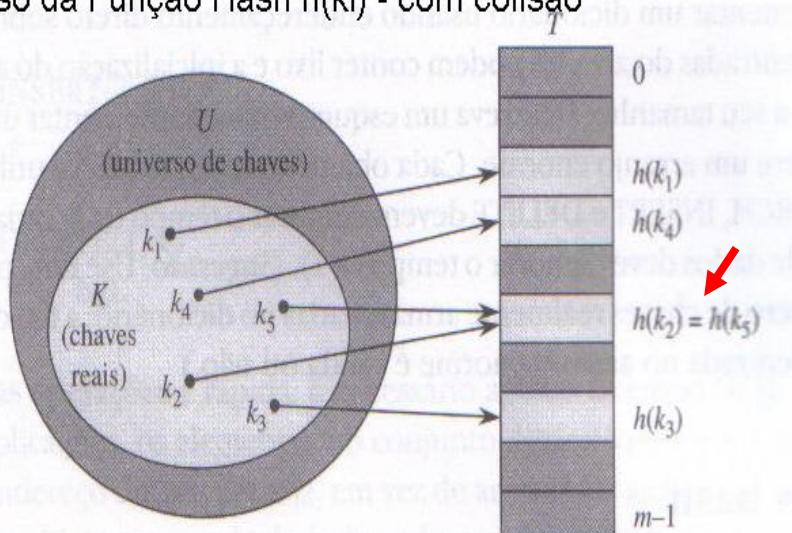
- Eficiente para a implementação de dicionários, ou seja, símbolos associados à significados;
- Exemplo de aplicação: compiladores;
- Análise da busca da informação:
  - Pior caso =  $O(n)$ ;
  - Caso médio =  $O(1)$ ;
- Consiste em uma generalização de uma array (arranjo) comum, ou seja, endereço mapeado sobre o conteúdo (processo de endereçamento direto) que leva a uma complexidade  $O(1)$ ;



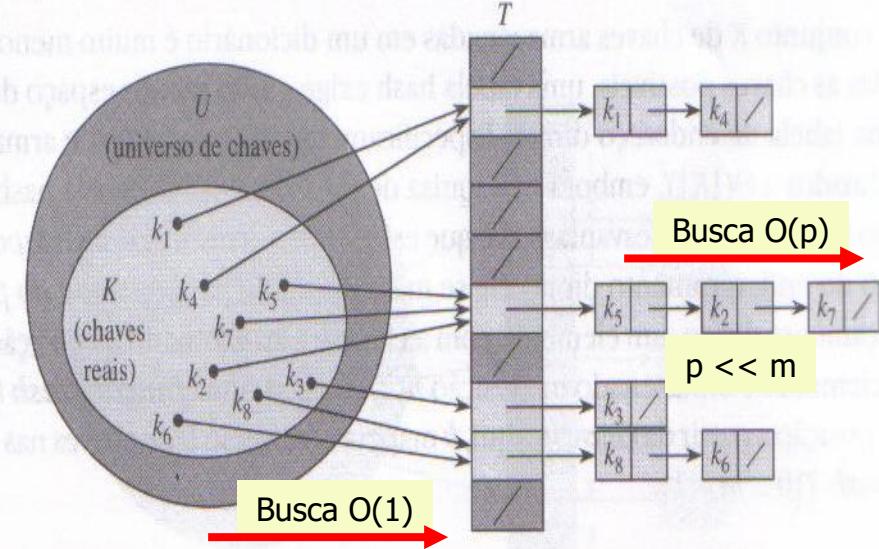
# Estrutura de Dados – Tabela Hash

- Tabela Hash:
  - Consiste no uso de uma função hash ( $h$ ) para o mapeamento da informação:

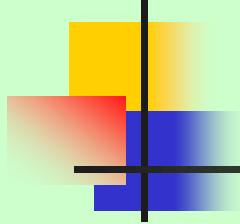
Uso da Função Hash  $h(k_i)$  - com colisão



Solução da colisão por encadeamento

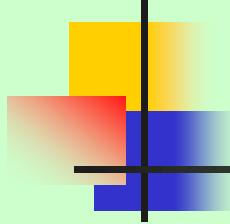


**SEGREDO DA EFICIÊNCIA:** estabelecer uma função Hash de boa qualidade !!!  
**(MINIMIZAR COLISÕES)**



# Estrutura de Dados – Tabela Hash

- Hash Bom:
  - É a função **h** que garante a busca  $O(1)$ , ou seja sem colisão;
  - Difícil de ser obtida devido a não se conhecer a distribuição de probabilidade segundo a qual as chaves são obtidas;
  - Então:
    - O prévio conhecimento da distribuição probabilística de ocorrências minimizará a ocorrência de colisões ( constitui-se em uma regra heurística !!! );
    - Logo, as informações com ocorrências similares **NÃO** devem ficar em um mesmo endereço, que geralmente são números naturais positivos ( $0, 1, 2, \dots, m$ );



# Estrutura de Dados – Tabela Hash

- Tipos de Função Hash:

1. Por divisão;
2. Por multiplicação;
3. Hash universal (hash aleatório);

1. Hash de Divisão:  $h(k) = k \text{ MOD } m$ ;

- Definição do  $m$ :
  1. Evitar potência de 2;
  2. Melhor usar um número primo, não próximo a uma potência de 2;

**Exemplo:** Armazenar  $n$  elementos, onde  $n = 2000$ ;

- bom nível de pesquisa é 3 elementos encadeados (previamente estabelecido !!);
- podemos adotar  $m = 701$ , pois é um primo próximo a 2000 divididos por 3;
- Então:  **$h(k) = k \text{ MOD } 701$** ;

# Estrutura de Dados – Tabela Hash

2. Hash de Multiplicação:  $h(k) = \text{MENOR\_INTEIRO}(m * (\underline{(k*A) \text{ MOD } (1)}))$ ;



=parte fracionária de K.A

- Definição do m:
  - É uma potência de 2 =  $2^p$ , sendo p um número inteiro;
- Definição do A (teoricamente  $0 < A < 1$ ):
  - Definido pelo valor de Knuth =  $(\sqrt{5} - 1)/2 = 0.6180339887$  ;

## Exemplo:

- $k = 123456$ ;
- $p = 14 \Rightarrow m = 2^{14} = 16384$ ;
- informação k se encaixa em uma palavra de 32 bits, adotando-se então  $w = 32$ ;
- (restrição imposta)  $A = s/2^w = 0.6180339887 = s/2^{32}$ , observe que  $0 < w < 2^w$ ;
- logo:  $s=2654435769$ ;
- Então:  $h(k) \cong 67 = \text{MENOR\_INTEIRO}(16384((k.0,618...)) \text{MOD}(1))$

# Estrutura de Dados – Tabela Hash

## 3. Hash Universal (Aleatório):

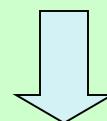
- Pior caso do hash (qualquer):
  - $O(n) = 1$  endereço com n colisões;
- O uso do Hash aleatório garante uma probabilidade mínima de ocorrência  $O(n)$  na busca;
- LOGO: deve-se selecionar uma função hash ao acaso !!!
- Para tal: necessidade de se projetar um conjunto de funções hash onde a cada uso, o mapeamento se dará aleatoriamente;
- ENTÃO:

$$h_{a,b}(k) = ((a \cdot k + b) \text{MOD}(p)) \text{MOD}(m);$$

$$a \in \{0, 1, 2, 3, \dots, p-1\};$$

$$b \in \{1, 2, 3, \dots, p-1\};$$

$$p = \text{numero\_primo} \Rightarrow p > m$$



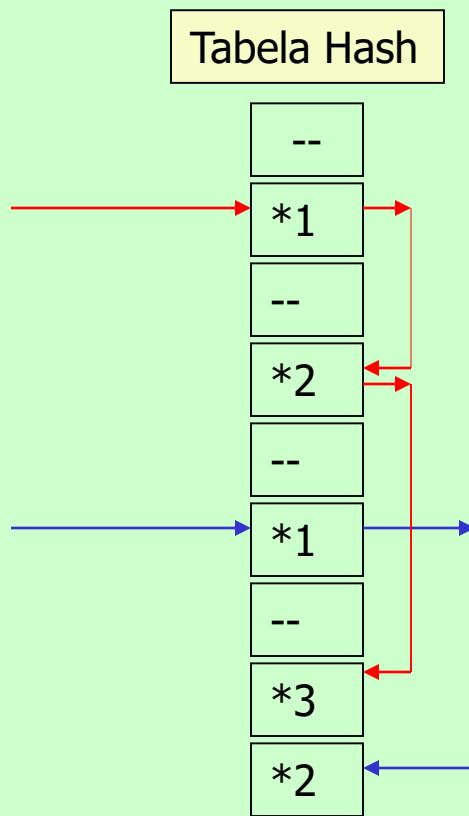
Exemplo: Sendo  $p = 17$  e  $m = 6$  tem-se:

$$h_{3,4}(8) = 5;$$

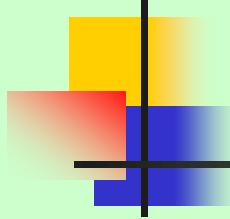
# Estrutura de Dados – Tabela Hash

- Hash de Endereçamento Aberto:

- Todos os elementos (informações) estão armazenadas na própria tabela hash;

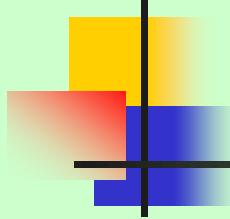


- Para cada chave  $k$ , tem-se uma seqüência de sondagem:  
Ex:  $h(k,0); h(k,1); h(k,2); \dots; h(k,m-2); h(k,m-1)$ ;



## Estrutura de Dados – Tabela Hash

- Hash com Sondagem Linear:
  - Hash: 
$$h(k, i) = ((h'(k) + i) MOD(m));$$
$$i = 0, 1, 2, 3, \dots, m - 1;$$
- **OBS:** sofre de agrupamento primário, ou seja, cria seqüências de informações;
- Hash com Sondagem Quadrática:
  - Hash: 
$$h(k, i) = ((h'(k) + c_1 \cdot i^1 + c_2 \cdot i^2) MOD(m));$$
$$i = 0, 1, 2, 3, \dots, m - 1;$$
$$c_1 \neq 0;$$
$$c_2 \neq 0;$$



# Estrutura de Dados – Tabela Hash

- Hash Duplo:
  - É melhor pois fornece perturbações aleatórias no endereços;

$$h(k, i) = ((h_1(k) + i \cdot h_2(k)) MOD(m));$$

$$h_1(k) = (k) MOD(m);$$

$$h_2(k) = 1 + ((k) MOD(m'));$$

$$i = 0, 1, 2, 3, \dots, m - 1;$$

*m = tamanho\_da\_tabela\_hash;*

*m = potencia\_de\_2\_ou\_um\_primo;*

*m' = um\_valor\_ligeiramente\_menor\_que\_m*

# Estrutura de Dados – Tabela Hash

$$K=79 \rightarrow i = 0 \Rightarrow h(79,0) = (1+0)MOD13 = 1$$

$$K=98 \rightarrow i = 0 \Rightarrow H(98,0) = ((7)+0.(1+10))MOD13 = 7MOD13 = 7$$

$$i = 1 \Rightarrow H(98,1) = ((98MOD13)+1.(1+(98MOD11))MOD13 = ((7)+1.(1+10))MOD13 = (7+1.11)MOD13 = 5$$

$$K=14 \rightarrow i = 0 \Rightarrow h(14,0) = (1+0.(1+(3)))MOD13 = 1$$

$$i = 1 \Rightarrow h(14,1) = ((14MOD13)+1.(1+(14MOD11))MOD13 = ((1+1.(1+3)))MOD13 = 5MOD13 = 5$$

$$i = 2 \Rightarrow h(14,2) = ((14MOD13)+2.(1+(14MOD11))MOD13 = h(14,2) = 9MOD13 = 9$$

Ex:  $m=13$ ;  
 $h_1(k) = k MOD 13$ ;  
 $h_2(k) = 1+(k MOD 11)$ ;

Tabela Hash

-	<b>79</b>	-	-	69	<b>98</b>	-	72	-	<b>14</b>	-	50	-
0	1	2	3	4	5	6	7	8	9	10	11	12

# Estrutura de Dados – Tabela Hash

```
Set Define Structure{  
    Chave dado[0....m-1]  
} Hash;
```

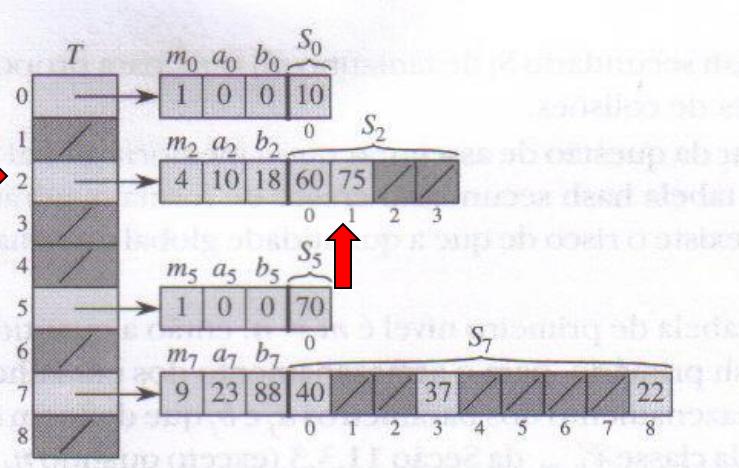
```
Int Hash_INSERT(Hash hs , Chave k)  
begin  
    i=0;  
    while i < tamanho(hs.dado) do  
        j = h( k , i ); /*função hash estabelecida */  
        if ( hs.dado[j] = nill ) then  
            hs.dado[j] = k;  
            return( j );  
        else  
            i ++;  
        end_IF;  
    end_WHILE;  
    printf( ` erro: tabela hash overflow ' );  
end PROCEDURE;
```

```
Int Hash_Search(Hash hs , Chave k)  
begin  
    i=0;  
    while ( i < tamanho(hs.dado) ) or ( hs.dado[j] != nill ) do  
        j = h( k , i ); /*função hash estabelecida */  
        if ( hs.dado[j] = k ) then  
            return( j );  
        else  
            i ++;  
        end_IF;  
    end_WHILE;  
    printf( ` erro: chave não encontrada ' );  
end PROCEDURE;
```

# Estrutura de Dados – Tabela Hash

## ■ Hash Perfeito:

- É aquele em que a busca é feita em  $O(1)$ ;
- Usa 2 níveis de hash universal:
  - 1º nível: aplicação do hash universal tradicional com valores de  $a$ ,  $b$ ,  $p$  e  $m$  previamente definidos;
  - 2º nível: ao invés de se ter uma lista de encadeada de nós, tem-se uma segunda tabela hash, com valores de  $a$ ,  $b$ , e  $m$  associados a cada nova tabela hash. O valor de  $p$  é o mesmo usado no hash do 1º nível;



Exemplo:  $h(k) = ((ak+b) \text{ MOD } p) \text{ mod } m$

1. Hash 1º nível:
  - $a_1=3; b_1=42; p_1=101; m_1=9;$
  - $H_1(75) = (3.75+42) \text{ MOD } 101 \text{ mod } 9 = 2;$

2. Hash 2º nível:
  - $a_2=10; b_2=18; p_2=101; m_2=4;$
  - $H_2(75) = (10.75+18) \text{ MOD } 101 \text{ mod } 4 = 1;$

**OBS:**  $m_j = (\text{número de colisões no endereço } j \text{ da tabela } T)^2$   
Isso garante busca  $O(1)$ , evitando colisões no 2º nível de hash !!!