

Bacharelado em Ciência da Computação
GBC034 Algoritmos e Estruturas de Dados 2

Introdução a Complexidade de Algoritmos

Material baseado nos slides dos professores Nivio Ziviani (Projeto de Algoritmos) e André Backes (Linguagem C Descomplicada)

Profa. Maria Camila Nardini Barioni

camila.barioni@ufu.br

Bloco B - sala 1B137

2º semestre de 2023

Roteiro

- 
- ◆ Porque analisar a complexidade de algoritmos?
 - ◆ Função de Complexidade
 - ◆ Melhor Caso, Pior Caso e Caso Médio
 - ◆ A Notação O
 - ◆ Comparação de Programas
 - ◆ Sugestões de Leitura

Algoritmos

- ◆ Como resolver um **problema** no computador?
 - Precisamos descrevê-lo de uma forma clara e precisa
- ◆ Precisamos escrever o seu **algoritmo**
 - Um **algoritmo** é uma sequência simples e objetiva de **instruções**
 - Cada **instrução** é uma informação que indica ao computador uma ação básica a ser executada

Algoritmos

◆ Vários algoritmos para um mesmo problema

- Os algoritmos se diferenciam uns dos outros pela maneira como eles utilizam os recursos do computador

◆ Os algoritmos dependem

- Principalmente do **tempo** que demora pra ser executado
- Da quantidade de **memória** do computador

Porque analisar a complexidade de algoritmos?

- ◆ Área de pesquisa cujo foco são os algoritmos
- ◆ Permite analisar aspectos dos algoritmos
 - Eficiência: cumpre com o objetivo
 - Desempenho: rapidez com que é solucionado
 - Robustez: pode ser utilizado independente do tamanho do problema
- ◆ Algoritmos diferentes mas capazes de resolver o mesmo problema não necessariamente o fazem como o mesmo desempenho
 - Exemplo: ordenação de números

Porque analisar a complexidade de algoritmos?

- ◆ As diferenças de eficiência podem ser
 - **Irrelevantes** para um pequeno número de elementos processados
 - **Crescer proporcionalmente** com o número de elementos processados
- ◆ Dependendo do tamanho dos dados e da eficiência, um programa poderia executar
 - Instantaneamente
 - De um dia para o outro
 - Por séculos

Porque analisar a complexidade de algoritmos?

◆ Complexidade computacional

- Medida criada para comparar a eficiência dos algoritmos
- Indica o **custo** ao se aplicar um algoritmo

custo = memória + tempo

- ◆ **memória:** quanto de espaço o algoritmo vai consumir
- ◆ **tempo:** a duração de sua execução

Porque analisar a complexidade de algoritmos?

◆ Importante

- O custo pode estar associado a outros recursos computacionais, além da memória
 - ◆ Exemplo: tráfego de rede
- No entanto, para a maior parte dos problemas o custo está relacionado ao tempo de execução em função do **tamanho da entrada** a ser processada

Tipos de problemas na análise de algoritmos

1. Análise de **um algoritmo particular**

- ◆ Qual é o custo de usar um dado algoritmo para resolver um problema específico?
- ◆ Características que devem ser investigadas:
 - análise do número de vezes que cada parte do algoritmo deve ser executada
 - estudo da quantidade de memória necessária

Tipos de problemas na análise de algoritmos

2. Análise de **uma classe de algoritmos**

- Qual é o algoritmo de menor custo possível para resolver um problema particular?
- Toda uma família de algoritmos é investigada
- Procura-se identificar um que seja o melhor possível
- Coloca-se **limites** para a complexidade computacional dos algoritmos pertencentes à classe

Custo de um algoritmo

- ◆ Determinando o menor custo possível para resolver problemas de uma dada classe, temos a medida da dificuldade inerente para resolver o problema
- ◆ Quando o custo de um algoritmo é igual ao menor custo possível, o algoritmo é **ótimo** para a medida de custo considerada
- ◆ Podem existir vários algoritmos para resolver o mesmo problema
- ◆ Se a mesma medida de custo é aplicada a diferentes algoritmos, então é possível compará-los e escolher o mais adequado

Custo de um algoritmo

◆ Podemos utilizar duas abordagens

- **Análise empírica**

- ◆ Comparação entre os programas

- **Análise matemática**

- ◆ Estudo das propriedades do algoritmo

Medida do custo pela execução do programa – Método empírico

◆ Definição

- Avalia o custo (ou complexidade) de um algoritmo a partir da avaliação da execução do mesmo quando implementado
- Análise pela execução de seu programa correspondente

Medida do custo pela execução do programa – Método empírico

- ◆ Tais medidas são bastante inadequadas e os resultados jamais devem ser generalizados:
 - os resultados são **dependentes do compilador** que pode favorecer algumas construções em detrimento de outras
 - os resultados **dependem do hardware**
 - quando **grandes quantidades de memória** são utilizadas, as medidas de tempo podem depender deste aspecto

Medida do custo pela execução do programa – Método empírico

- ◆ Apesar disso, há argumentos a favor de se obterem medidas reais de tempo
 - Ex.: quando há vários algoritmos distintos para resolver um mesmo tipo de problema, todos com um custo de execução dentro de uma mesma ordem de grandeza
 - Assim, são considerados tanto os custos reais das operações como os custos não aparentes, tais como alocação de memória, indexação, carga, dentre outros

Medida do custo por meio de um modelo matemático – Método analítico

- ◆ Usa um modelo matemático baseado em um computador idealizado
- ◆ Deve ser especificado o conjunto de operações e seus custos de execuções
- ◆ É mais usual ignorar o custo de algumas das operações e considerar apenas as operações mais significativas **(operações dominantes)**
- ◆ Ex.: algoritmos de ordenação. Consideramos o número de comparações entre os elementos do conjunto a ser ordenado e ignoramos as operações aritméticas, de atribuição e manipulações de índices, caso existam

Função de complexidade

- ◆ Para medir o custo de execução de um algoritmo é comum definir uma função de custo ou **função de complexidade** f
- ◆ $f(n)$ é a medida do tempo necessário para executar um algoritmo para um problema de tamanho n
- ◆ Função de **complexidade de tempo**: $f(n)$ mede o tempo necessário para executar um algoritmo em um problema de tamanho n
- ◆ Função de **complexidade de espaço**: $f(n)$ mede a memória necessária para executar um algoritmo em um problema de tamanho n

Função de complexidade

- ◆ Utilizaremos f para denotar uma função de complexidade de tempo daqui para a frente
- ◆ A complexidade de tempo na realidade não representa tempo diretamente, mas o número de vezes que determinada operação considerada relevante é executada

Exemplo – Maior Elemento

- ◆ Considere o algoritmo para procurar o maior valor presente em um array **A** contendo **n** elementos

```
int M = A[0];  
  
for(i = 0; i < n; i++) {  
    if(A[i] >= M) {  
        M = A[i];  
    }  
}
```

Exemplo – Maior Elemento

◆ Quantas **instruções simples** ele executa?

- **Instruções simples:** instruções que podem ser executadas diretamente pela CPU (ou algo muito perto disso)
 - ◆ atribuição de um valor a uma variável
 - ◆ acesso ao valor de um determinado elemento do array
 - ◆ comparação de dois valores
 - ◆ incremento de um valor
 - ◆ operações aritméticas básicas, como adição e multiplicação

Exemplo – Maior Elemento

◆ Instruções simples

- Todas possuem o mesmo custo
- Comandos de seleção (como o comando **if**) possuem custo zero
 - ◆ Não contam como instruções

```
int M = A[0];  
  
for(i = 0; i < n; i++) {  
    if(A[i] >= M) {  
        M = A[i];  
    }  
}
```

Exemplo – Maior Elemento

- ◆ Custo da inicialização de **M**: 1 instrução
 - Apenas uma operação de atribuição

```
int M = A[0];  
  
for(i = 0; i < n; i++) {  
    if(A[i] >= M) {  
        M = A[i];  
    }  
}
```

Exemplo – Maior Elemento

- ◆ Custo de inicialização do laço **for**: 2 instruções
 - Uma operação de atribuição e uma comparação

```
int M = A[0];  
  
for(i = 0; i < n; i++) {  
    if(A[i] >= M) {  
        M = A[i];  
    }  
}
```

Exemplo – Maior Elemento

◆ Custo de execução do laço **for**: $2n$ instruções

- Uma operação de incremento e uma comparação executadas **n** vezes

```
int M = A[0];  
  
for(i = 0; i < n; i++) {  
    if(A[i] >= M) {  
        M = A[i];  
    }  
}
```

Exemplo – Maior Elemento

- ◆ Ignorando os comandos contidos dentro do laço **for**, temos que o algoritmo irá executar **$3+2n$** instruções
 - 3 instruções antes de iniciar o laço **for**
 - 2 instruções ao final de cada laço **for**

```
int M = A[0];  
  
for(i = 0; i < n; i++) {  
    if(A[i] >= M) {  
        M = A[i];  
    }  
}
```

Exemplo – Maior Elemento

- ◆ Assim, considerando um **laço vazio**, podemos definir uma função matemática que representa o custo do algoritmo em relação ao tamanho do array de entrada
 - $f(n) = 2n + 3$

```
int M = A[0];  
  
for(i = 0; i < n; i++) {  
    if(A[i] >= M) {  
        M = A[i];  
    }  
}
```

Exemplo – Maior Elemento

- ◆ As instruções vistas anteriormente eram sempre executadas. Porém, as instruções dentro do **for** podem ou não ser executadas
 - Comando de seleção: 1 instrução
 - ◆ Sempre executada
 - Atribuição: 1 instrução
 - ◆ Depende do resultado do comando de seleção

```
int M = A[0];  
  
for(i = 0; i < n; i++) {  
    if(A[i] >= M) {  
        M = A[i];  
    }  
}
```

Exemplo – Maior Elemento

- ◆ Antes, bastava saber o tamanho do array, **n**, para definir a função de custo **f(n)**
- ◆ Agora, temos que considerar também o conteúdo do array
- ◆ Tome como exemplo os dois arrays abaixo

```
int A1[4] = {1, 2, 3, 4};  
int A2[4] = {4, 3, 2, 1};
```

Exemplo – Maior Elemento

- ◆ O array **A1** irá executar mais instruções do que o array **A2**
 - Array **A1**: o comando **if** é sempre **verdadeiro**
 - Array **A2**: o comando **if** é sempre **falso**
- ◆ Devemos considerar o pior caso possível
 - Maior número de instruções é executado

```
int A1[4] = {1, 2, 3, 4};  
int A2[4] = {4, 3, 2, 1};
```

Exemplo – Maior Elemento

- ◆ Neste exemplo, o **pior caso** ocorre quando o array possui valores em ordem crescente
 - Valor de **M** é sempre substituído: $2n$ instruções
 - ◆ Maior número de instruções
 - A função custo será, no **pior caso**,
 - ◆ $f(n) = 3 + 2n + 2n$ ou
 - ◆ $f(n) = 4n + 3$

```
int M = A[0];  
  
for(i = 0; i < n; i++) {  
    if(A[i] >= M) {  
        M = A[i];  
    }  
}
```

Tamanho da entrada de dados

- ◆ A medida do custo de execução de um algoritmo depende principalmente do tamanho da entrada dos dados
- ◆ É comum considerar o tempo de execução de um programa como uma função do tamanho da entrada
- ◆ Para alguns algoritmos, o custo de execução é uma função da entrada particular dos dados, não apenas do tamanho da entrada
- ◆ No caso da função Max do programa do exemplo, o custo é uniforme sobre todos os problemas de tamanho n
- ◆ Já para um algoritmo de ordenação isso não ocorre: se os dados de entrada já estiverem quase ordenados, então o algoritmo pode ter que trabalhar menos

Exercício – Contando instruções

- ◆ Considere o algoritmo para procurar o maior valor presente em uma matriz **A** contendo **n x n** elementos

```
M = A[0][0];  
  
for(i=0; i<n; i++)  
    for(j=0; j<n; j++)  
        if(M < A[i][j])  
            M = A[i][j];
```

	j	
	0	1
i	0	10 20 30
	1	40 50 60
	2	70 80 90

- ◆ Quantas **instruções simples** ele executa?

Melhor Caso, Pior Caso e Caso Médio

- ◆ Com isso, define-se a notação de complexidade de tempo por:
 - ◆ Seja A um algoritmo
 - ◆ $\{E_1, \dots, E_m\}$ o conjunto de todas as entradas possíveis de A
 - ◆ Seja t_i o número de passos efetuados por A quando a entrada for E_i
 - ◆ **Melhor caso:** $\min_{E_i \in E} \{t_i\}$ (menor tempo de execução sobre todas as entradas de tamanho n)
 - ◆ **Pior caso:** $\max_{E_i \in E} \{t_i\}$ (maior tempo de execução sobre todas as entradas de tamanho n)
 - Se f é uma função de complexidade baseada na análise de pior caso, o custo de aplicar o algoritmo nunca é maior do que $f(n)$

Melhor Caso, Pior Caso e Caso Médio

- ◆ **Caso médio (ou caso esperado):** $\sum_{1 \leq i \leq m} p_i t_i$ (média dos tempos de execução de todas as entradas de tamanho n)
 - Na análise do caso esperado, supõe-se uma distribuição de probabilidades sobre o conjunto de entradas de tamanho n e o custo médio é obtido com base nessa distribuição
 - A análise do caso médio é geralmente muito mais difícil de obter do que as análises do melhor e do pior caso
 - É comum supor uma distribuição de probabilidades em que todas as entradas possíveis são igualmente prováveis
 - Na prática isso nem sempre é verdade

Exemplo – Registros de um Arquivo

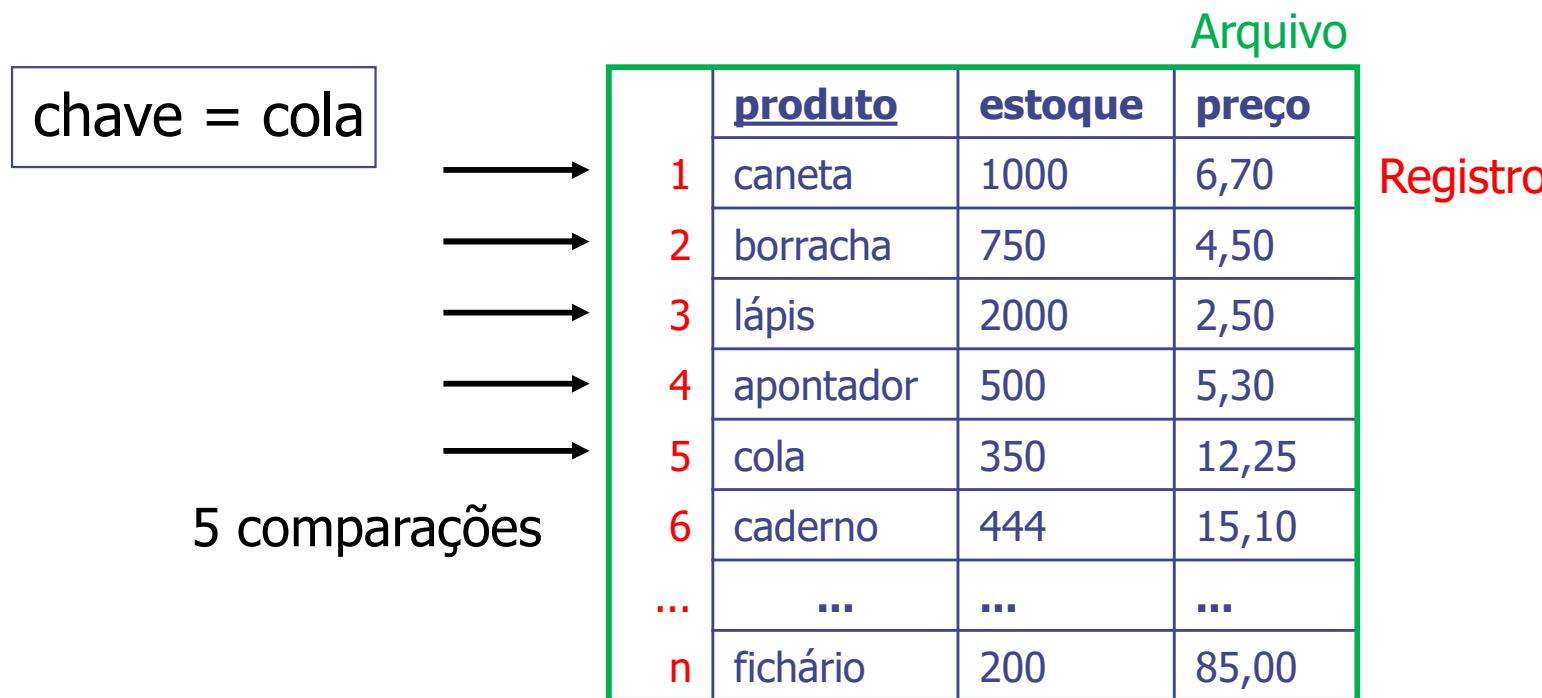
- ◆ Considere o problema de acessar os **registros** de um arquivo
- ◆ Cada registro contém uma **chave** única que é utilizada para recuperar registros do arquivo

Arquivo			
	<u>produto</u>	estoque	preço
1	caneta	1000	6,70
2	borracha	750	4,50
3	lápis	2000	2,50
4	apontador	500	5,30
5	cola	350	12,25
6	caderno	444	15,10
...
n	fichário	200	85,00

Registro

Exemplo – Registros de um Arquivo

- ◆ O problema: dada uma chave qualquer, localize o registro que contenha esta chave
- ◆ O algoritmo de pesquisa mais simples é o que faz a **pesquisa seqüencial**



Exemplo – Registros de um Arquivo

- ◆ Seja f uma função de complexidade tal que $f(n)$ é o número de registros consultados no arquivo (número de vezes que a chave de consulta é comparada com a chave de cada registro)
 - ◆ **melhor caso:** $f(n) = 1$ (registro procurado é o primeiro consultado);
 - ◆ **pior caso:** $f(n) = n$ (registro procurado é o último consultado ou não está presente no arquivo)
 - ◆ **caso médio:** $f(n) = (n + 1) / 2$

Exemplo – Registros de um Arquivo

- ◆ No estudo do caso médio, vamos considerar que toda pesquisa recupera um registro
- ◆ Se p_i for a probabilidade de que o i -ésimo registro seja procurado, e considerando que para recuperar o i -ésimo registro são necessárias i comparações, então

$$f(n) = 1 * p_1 + 2 * p_2 + 3 * p_3 + \dots + n * p_n$$

- ◆ Para calcular $f(n)$ basta conhecer a distribuição de probabilidades p_i
- ◆ Se cada registro tiver a mesma probabilidade de ser acessado que todos os outros, então

$$p_i = 1/n; 1 \leq i \leq n$$

Exemplo – Registros de um Arquivo

- ◆ Neste caso

$$f(n) = \frac{1}{n} \cdot (1 + 2 + 3 + \dots + n)$$

- ◆ A soma $(1 + 2 + 3 + \dots + n)$ é uma série aritmética $\frac{n.(n+1)}{2}$

- ◆ Assim

$$f(n) = \frac{1}{n} \cdot \frac{n.(n+1)}{2} = \frac{(n+1)}{2}$$

- ◆ A análise do caso esperado revela que uma pesquisa com sucesso examina aproximadamente metade dos registros

Exemplo – Maior e Menor Elemento (1)

Considere o problema de encontrar o maior e o menor elemento de um vetor de inteiros $A[0::n-1]$; $n \geq 1$

```
void MaxMin1 (int *A, int n, int *Max, int *Min){  
    int i;  
    Max = A[0]; Min = A[0];  
    for (i = 1; i < n; i++){  
        if (A[i] > Max) Max = A[i];  
        if (A[i] < Min) Min = A[i];  
    }  
}
```

EXEMPLO

$n = 5$

A	10	6	23	3	1
	0	1	2	3	4

Exemplo – Maior e Menor Elemento (1)

Considere o problema de encontrar o maior e o menor elemento de um vetor de inteiros $A[0::n-1]$; $n \geq 1$

```
void MaxMin1 (int *A, int n, int *Max, int *Min){  
    int i;  
    Max = A[0]; Min = A[0];  
    for (i = 1; i < n; i++){  
        if (A[i] > Max) Max = A[i];  
        if (A[i] < Min) Min = A[i];  
    }  
}
```



EXEMPLO

$n = 5$

A	10	6	23	3	1
	0	1	2	3	4

$\text{Max} = 10$

$\text{Min} = 10$

Exemplo – Maior e Menor Elemento (1)

Considere o problema de encontrar o maior e o menor elemento de um vetor de inteiros $A[0::n-1]$; $n \geq 1$

```
void MaxMin1 (int *A, int n, int *Max, int *Min){  
    int i;  
    Max = A[0]; Min = A[0];  
    for (i = 1; i < n; i++){  
        if (A[i] > Max) Max = A[i];  
        if (A[i] < Min) Min = A[i];  
    }  
}
```

EXEMPLO

$n = 5$

A	10	6	23	3	1
	0	1	2	3	4
		↑			i

$\text{Max} = 10$

$\text{Min} = 10$

Exemplo – Maior e Menor Elemento (1)

Considere o problema de encontrar o maior e o menor elemento de um vetor de inteiros $A[0::n-1]$; $n \geq 1$

```
void MaxMin1 (int *A, int n, int *Max, int *Min){  
    int i;  
    Max = A[0]; Min = A[0];  
    for (i = 1; i < n; i++){  
        if (A[i] > Max) Max = A[i];  
        if (A[i] < Min) Min = A[i];  
    }  
}
```

EXEMPLO

$n = 5$

A	10	6	23	3	1
	0	1	2	3	4
		↑			i

$\text{Max} = 10$

$\text{Min} = 6$

Exemplo – Maior e Menor Elemento (1)

Considere o problema de encontrar o maior e o menor elemento de um vetor de inteiros $A[0::n-1]$; $n \geq 1$

```
void MaxMin1 (int *A, int n, int *Max, int *Min){  
    int i;  
    Max = A[0]; Min = A[0];  
    for (i = 1; i < n; i++){  
        if (A[i] > Max) Max = A[i];  
        if (A[i] < Min) Min = A[i];  
    }  
}
```

EXEMPLO

$n = 5$

$\text{Max} = 23$

A	10	6	23	3	1
	0	1	2	3	4
			↑		i

$\text{Min} = 6$

Exemplo – Maior e Menor Elemento (1)

Considere o problema de encontrar o maior e o menor elemento de um vetor de inteiros $A[0::n-1]$; $n \geq 1$

```
void MaxMin1 (int *A, int n, int *Max, int *Min){  
    int i;  
    Max = A[0]; Min = A[0];  
    for (i = 1; i < n; i++){  
        if (A[i] > Max) Max = A[i];  
        if (A[i] < Min) Min = A[i];  
    }  
}
```

EXEMPLO

$n = 5$

$\text{Max} = 23$

A	10	6	23	3	1
	0	1	2	3	4
				↑	i

$\text{Min} = 3$

Exemplo – Maior e Menor Elemento (1)

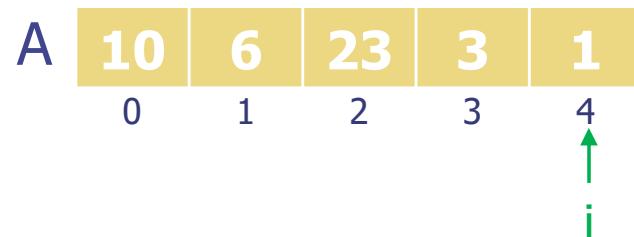
Considere o problema de encontrar o maior e o menor elemento de um vetor de inteiros $A[0::n-1]$; $n \geq 1$

```
void MaxMin1 (int *A, int n, int *Max, int *Min){  
    int i;  
    Max = A[0]; Min = A[0];  
    for (i = 1; i < n; i++){  
        if (A[i] > Max) Max = A[i];  
        if (A[i] < Min) Min = A[i];  
    }  
}
```

EXEMPLO

$n = 5$

$\text{Max} = 23$



$\text{Min} = 1$

Exemplo – Maior e Menor Elemento (1)

Considere o problema de encontrar o maior e o menor elemento de um vetor de inteiros $A[0::n-1]$; $n \geq 1$

```
void MaxMin1 (int *A, int n, int *Max, int *Min) {
    int i;
    Max = A[0]; Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > Max) Max = A[i];
        if (A[i] < Min) Min = A[i];
    }
}
```

- ◆ Seja f uma função de complexidade tal que **$f(n)$ é o número de comparações** entre os elementos de A , se A contiver n elementos
- ◆ Logo $f(n) = 2(n - 1)$, para $n > 0$, para o melhor caso, pior caso e caso médio

Exemplo – Maior e Menor Elemento (2)

Considere o problema de encontrar o maior e o menor elemento de um vetor de inteiros $A[0::n-1]$; $n \geq 1$

```
void MaxMin2 (int *A, int n, int *Max, int *Min) {
    int i;
    Max = A[0]; Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > Max) Max = A[i];
        else if (A[i] < Min) Min = A[i];
    }
}
```

- ◆ MaxMin1 pode ser facilmente melhorado: a comparação $A[i] < Min$ só é necessária quando a comparação $A[i] > Max$ dá falso

Exemplo – Maior e Menor Elemento (2)

- ◆ Para a nova implementação temos:
 - **melhor caso:** $f(n) = ?$
 - **pior caso:** $f(n) = ?$
 - **caso médio:** $f(n) = ?$

Exemplo – Maior e Menor Elemento (2)

- ◆ Para a nova implementação temos:
 - **melhor caso:** $f(n) = n - 1$ (quando os elementos estão em ordem crescente)
 - **pior caso:** $f(n) = 2(n - 1)$ (quando os elementos estão em ordem decrescente)
 - **caso médio:** $f(n) = (3n/2) - 3/2$
 - No caso médio, $A[i]$ é maior do que Max a metade das vezes
 - Logo $f(n) = n - 1 + (n - 1)/2 = (3n/2) - 3/2$, para $n > 0$

Exemplo – Maior e Menor Elemento (2)

Considere o problema de encontrar o maior e o menor elemento de um vetor de inteiros $A[0::n-1]$; $n \geq 1$

```
void MaxMin2 (int *A, int n, int *Max, int *Min) {
    int i;
    Max = A[0]; Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > Max) Max = A[i];
        else if (A[i] < Min) Min = A[i];
    }
}
```

Detalhando o caso médio...

Metade das vezes o primeiro *if* é falso $\rightarrow 2(n-1)/2$

Exemplo – Maior e Menor Elemento (2)

Considere o problema de encontrar o maior e o menor elemento de um vetor de inteiros $A[0::n-1]$; $n \geq 1$

```
void MaxMin2 (int *A, int n, int *Max, int *Min) {
    int i;
    Max = A[0]; Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > Max) Max = A[i];
        else if (A[i] < Min) Min = A[i];
    }
}
```

Detalhando o caso médio...

Metade das vezes o primeiro *if* é falso $\rightarrow 2(n-1)/2$

Metade das vezes o primeiro *if* é verdadeiro $\rightarrow (n-1)/2$

Exemplo – Maior e Menor Elemento (2)

Considere o problema de encontrar o maior e o menor elemento de um vetor de inteiros $A[0::n-1]$; $n \geq 1$

```
void MaxMin2 (int *A, int n, int *Max, int *Min) {
    int i;
    Max = A[0]; Min = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > Max) Max = A[i];
        else if (A[i] < Min) Min = A[i];
    }
}
```

Detalhando o caso médio...

Logo $f(n) = n - 1 + (n - 1)/2 = (3n/2) - 3/2$, para $n > 0$

Comportamento Assintótico de Funções

- ◆ O parâmetro n fornece uma medida da dificuldade para se resolver o problema
- ◆ Para valores suficientemente pequenos de n , qualquer algoritmo custa pouco para ser executado, mesmo os ineficientes
- ◆ A escolha do algoritmo não é um problema crítico para problemas de tamanho pequeno

Comportamento Assintótico de Funções

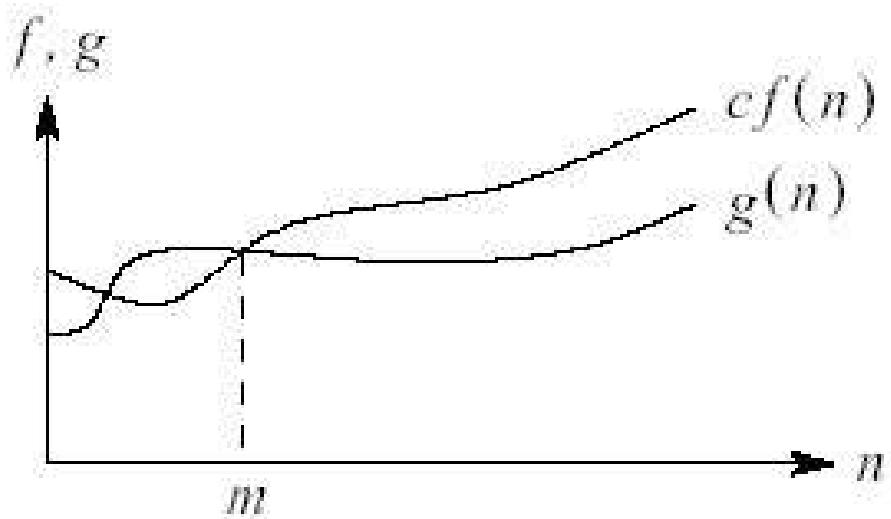
- ◆ Logo, a análise de algoritmos é realizada para valores grandes de n
- ◆ Estuda-se o comportamento assintótico das funções de custo (comportamento de suas funções de custo para valores grandes de n)
- ◆ O comportamento assintótico de $f(n)$ representa o limite do comportamento do custo quando n cresce

A Notação O

- ◆ Para a análise do comportamento de algoritmos existe toda uma terminologia própria
- ◆ **A mais importante delas e que é usada na prática é chamada de Ordem de Complexidade ou Notação-O**
- ◆ Escrevemos $g(n) = O(f(n))$ para expressar que $f(n)$ domina assintoticamente $g(n)$. Lê-se $g(n)$ é da ordem no máximo $f(n)$
- ◆ Exemplo: quando dizemos que o tempo de execução $T(n)$ de um programa é $O(n^2)$, significa que existem constantes c e m tais que, para valores de $n \geq m$, $T(n) \leq cn^2$

A Notação O

- ◆ Exemplo gráfico de dominação assintótica que ilustra a notação O



- ◆ Definição: Uma função $g(n)$ é $O(f(n))$ se existem duas constantes positivas c e m tais que $g(n) \leq c f(n)$, para todo $n \geq m$

A Notação O

- ◆ A análise de um algoritmo geralmente conta com apenas algumas operações elementares
- ◆ A medida de custo ou medida de complexidade relata o crescimento assintótico da operação considerada
- ◆ Tempo (ou espaço) é contabilizado em número de passos do algoritmo (unidade de armazenamento)
- ◆ Análise do algoritmo determina uma função que depende do tamanho da entrada n

$$10n^3 + 4n - 10$$

- ◆ à medida que n aumenta, o termo cúbico começa a dominar
- ◆ a constante do termo cúbico tem relativamente a mesma importância que a velocidade da CPU

A Notação O

◆ Complexidade

- desprezar constantes aditivas ou multiplicativas
 - número de passos $3n$ será aproximado para n
 - interesse assintótico - termos de menor grau podem ser desprezados: $n^2 + n$ será aproximado para n^2
 - $6n^3 + 4n - 9$ será aproximado para n^3

A Notação O

- ◆ A função atua como um limite superior assintótico da função f

◆ $f = n^2 - 1$	→	$f = O(n^2)$
◆ $f = n^3 - 1$	→	$f = O(n^3)$
◆ $f = 403$	→	$f = O(1)$
◆ $f = 5 + 2\log n$	→	$f = O(\log n)$
◆ $f = 3n + 5\log n + 2$	→	$f = O(n)$
◆ $f = 5 \cdot 2^n + 5n^{10}$	→	$f = O(2^n)$

A Notação O

- ◆ Usando a notação O para os exemplos

- ◆ Maior elemento

- $f(n) = 4n + 3 \rightarrow f = O(n)$

- ◆ Registros de um arquivo

- $f(n) = n \rightarrow f = O(n)$

- ◆ MaxMin1

- $f(n) = 2(n - 1) \rightarrow f = O(n)$

- ◆ MaxMin2

- $f(n) = 2(n - 1) \rightarrow f = O(n)$



NESSA DISCIPLINA VAMOS
UTILIZAR A **NOTAÇÃO O**

Comparação de Programas

- ◆ Podemos avaliar programas comparando as funções de complexidade, negligenciando as constantes de proporcionalidade
- ◆ Um programa com tempo de execução $O(n)$ é melhor que outro com tempo $O(n^2)$
- ◆ Porém, as constantes de proporcionalidade podem alterar esta consideração

Comparação de Programas

◆ Exemplo: um programa leva **$100n$** unidades de tempo para ser executado e outro leva **$2n^2$**

◆ Qual dos dois programas é melhor?

◆ depende do tamanho do problema

n	49	50	51	100
u. tempo	4802	5000	5202	20000

◆ Para **$n < 50$** , o programa com tempo **$2n^2$** é melhor do que o que possui tempo **$100n$**

n	49	50	51	100
u. tempo	4900	5000	5100	10000

◆ Para problemas com **entrada de dados pequena** e preferível usar o programa cujo tempo de execução é **$O(n^2)$**

◆ Entretanto, **quando n cresce**, o programa com tempo de execução **$O(n^2)$** leva **muito mais tempo** que o programa **$O(n)$**

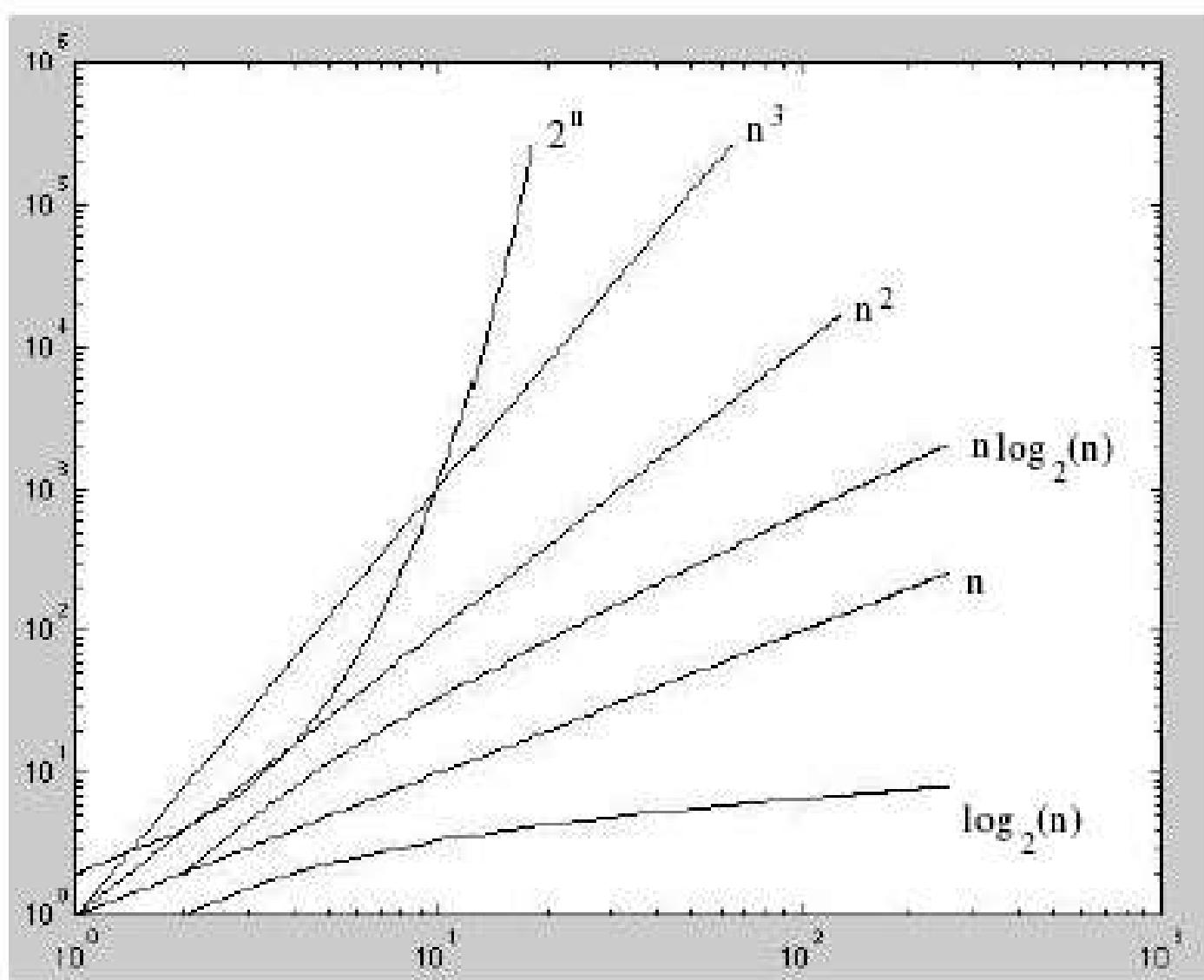
Principais classes de problemas

- ◆ Os tempos computacionais mais comuns de serem encontrados em análise de algoritmos são

$$O(1) < O(\log(n)) < O(n) < O(n\log(n)) < O(n^2) < O(n^3) < O(2^n)$$

$$O(1) < O(\log(n)) < O(n) < O(n\log(n)) < O(n^2) < O(n^3) < O(2^n)$$

A figura mostra, em escala logarítmica, como as ordens de magnitude dessas funções crescem com o valor de n



Principais classes de problemas

- ◆ $f(n) = O(1)$

- Algoritmos de complexidade $O(1)$ são ditos de **complexidade constante**
- Uso do algoritmo independe de n
- As instruções do algoritmo são executadas um número fixo de vezes

- ◆ $f(n) = O(\log n)$

- Um algoritmo de complexidade $O(\log n)$ é dito ter **complexidade logarítmica**
- Típico em algoritmos que transformam um problema em outros menores
- Pode-se considerar o tempo de execução como menor que uma constante grande
- Quando n é mil, $\log_2 n \sim 10$, quando n é 1 milhão, $\log_2 n \sim 20$

Principais classes de problemas

◆ $f(n) = O(n)$

- Um algoritmo de complexidade $O(n)$ é dito ter **complexidade linear**
- Em geral, um pequeno trabalho é realizado sobre cada elemento de entrada
- É a melhor situação possível para um algoritmo que tem que processar/produzir n elementos de entrada/saída
- Cada vez que n dobra de tamanho, o tempo de execução dobra

◆ $f(n) = O(n \log n)$

- Típico em algoritmos que quebram um problema em outros menores, resolvem cada um deles independentemente e ajuntando as soluções depois
- Quando n é 1 milhão, $n \log_2 n$ é cerca de 20 milhões
- Quando n é 2 milhões, $n \log_2 n$ é cerca de 42 milhões, pouco mais do que o dobro

Principais classes de problemas

◆ $f(n) = O(n^2)$

- Um algoritmo de complexidade $O(n^2)$ é dito ter **complexidade quadrática**
- Ocorrem quando os itens de dados são processados aos pares, muitas vezes em um anel dentro de outro
- Quando n é mil, o número de operações é da ordem de 1 milhão
- Sempre que n dobra, o tempo de execução é multiplicado por 4
- Úteis para resolver problemas de tamanhos relativamente pequenos

◆ $f(n) = O(n^3)$

- Um algoritmo de complexidade $O(n^3)$ é dito ter **complexidade cúbica**
- Úteis apenas para resolver pequenos problemas
- Quando n é 100, o número de operações é da ordem de 1 milhão
- Sempre que n dobra, o tempo de execução fica multiplicado por 8

Principais classes de problemas

◆ $f(n) = O(2^n)$

- Um algoritmo de complexidade $O(2^n)$ é dito ter **complexidade exponencial**
- Geralmente não são úteis sob o ponto de vista prático
- Quando n é 20, o tempo de execução é cerca de 1 milhão. Quando n dobra, o tempo fica elevado ao quadrado

◆ $f(n) = O(n!)$

- Um algoritmo de complexidade $O(n!)$ é dito ter **complexidade exponencial**, apesar de $O(n!)$ ter comportamento muito pior do que $O(2^n)$
- $n = 20 !$ $20! = 2432902008176640000$, um número com 19 dígitos
- $n = 40 !$ um número com 48 dígitos

Comparações de funções de complexidade

Função de custo	Tamanho n					
	10	20	30	40	50	60
n	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
n^2	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,035 s	0,0036 s
n^3	0,001 s	0,008 s	0,027 s	0,64 s	0,125 s	0,316 s
n^5	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
2^n	0,001 s	1 s	17,9 min	12,7 dias	35,7 anos	366 séc.
3^n	0,059 s	58 min	6,5 anos	3855 séc.	10^8 séc.	10^{13} séc.

Influência do aumento de velocidade dos computadores

Função de custo de tempo	Computador atual	Computador 100 vezes mais rápido	Computador 1.000 vezes mais rápido
n	t_1	$100 t_1$	$1000 t_1$
n^2	t_2	$10 t_2$	$31,6 t_2$
n^3	t_3	$4,6 t_3$	<u>$10 t_3$</u>
2^n	t_4	$t_4 + 6,6$	$t_4 + 10$

Sugestões de Leitura

- ◆ Capítulo 1 do livro do Nivio Ziviani: Projeto de Algoritmos
- ◆ Capítulos 2, 3 e 4 do livro do Thomas H. Cormen et al.: Algoritmos – Teoria e Prática
- ◆ Texto sobre Análise de Algoritmos do Professor Luis Gustavo Nonato ICMC/USP disponível no MS Teams

Material Complementar

◆ Vídeo Aulas

- Aula 99: Análise de Algoritmos:
youtu.be/iZK5WwJF1PE
- Aula 100: Análise de Algoritmos – Contando Instruções:
youtu.be/wfINJuryvTTQ
- Aula 101: Análise de Algoritmos – Comportamento Assintótico:
youtu.be/SClFMUpBiaw
- Aula 102: Análise de Algoritmos – Notação Grande-O:
youtu.be/Q7nwypDgTS8
- Aula 103: Análise de Algoritmos – Tipos de Análise Assintótica:
youtu.be/9RgC2dxi4W8
- Aula 104: Análise de Algoritmos – Classes de Problemas:
youtu.be/8RYvWMOMnXw

Bacharelado em Ciência da Computação
GBC034 Algoritmos e Estruturas de Dados 2

Recursividade

Profa. Maria Camila Nardini Barioni

camila.barioni@ufu.br

Bloco B - sala 1B137

2º semestre de 2023

Roteiro

- ◆ Introdução
- ◆ Exemplos
- ◆ Programação dinâmica
- ◆ Visão geral da Análise de Algoritmos Recursivos
- ◆ Exercícios

Recursividade

- ◆ Recursividade consiste na definição de um objeto em termos de (um caso mais simples de) si mesmo
- ◆ O poder da recursividade está na possibilidade de se definir um conjunto infinito de objetos por um conjunto finito de comandos
- ◆ Um algoritmo recursivo correto não gera uma seqüência infinita de chamadas a si mesmo
 - Deve haver uma “saída” da seqüência de chamadas recursivas

Recursividade

- ◆ A recursão é uma forma interessante de resolver problemas, pois o divide em problemas menores de mesma natureza

- ◆ Um processo recursivo consiste de duas partes:
 - O **caso trivial**, cuja solução é conhecida
 - Um **método geral** que reduz o problema a um ou mais problemas menores de mesma natureza

Recursividade

- ◆ Limitações de memória, processamento e armazenamento obrigam que este número seja finito
- Um programa recursivo pode executar um número teoricamente infinito de cálculos sem declarações explícitas de repetições

Recursividade Direta



```
se <condição>
```

```
    função(x) = função(y)
```

```
caso contrário
```

```
    função(x) = <valor>
```

```
se <condição>
```

```
    função (x) = <valor>
```

```
caso contrário
```

```
    função (x) = função(y)
```

- ◆ Para que haja a recursão direta, uma função / procedimento deve chamar a si mesmo

Recursividade Indireta

- ◆ Uma função A faz uma chamada à função B que, por sua vez, contém uma chamada direta ou indireta à função A
- ◆ Pode-se dizer que A e B são recursivas

- ◆ Exemplo:

```
void funcA (int a);
```

```
void funcB (int b);
```

Recursividade Indireta

```
void funcA (int a) {  
    if (a < 20) {  
        a++;  
        funcB (a); ←  
    }  
    printf( "a=%d", a);  
}
```

```
void funcB (int b) {  
    if (b < 20) {  
        b=b+2;  
        funcA (b); ←  
    }  
    printf ("b=%d",b);  
}
```

Recursividade Indireta

```
10  
void funcA (int a) {  
    if (a < 20) {  
        a++;  
        funcB (a);  
    }  
    printf( "a=%d", a);  
}  
  
void funcB (int b) {  
    if (b < 20) {  
        b=b+2;  
        funcA (b);  
    }  
    printf ("b=%d",b);  
}
```

Chamada de funcA (10)

funcA(10)

funcB(11)

1

2

3

4

5

6

7

8

9

10

11

12

13

Recursividade Indireta

```
void funcA (int a) {  
    if (a < 20) {  
        a++;  
        funcB (a);  
    }  
    printf( "a=%d", a);  
}  
  
void funcB (int b) {  
    if (b < 20) {  
        b=b+2;  
        funcA (b);  
    }  
    printf ("b=%d",b);  
}
```

Chamada de funcA (10)	
1	funcA(10)
2	funcB(11)
3	funcA(13)
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
0	

Recursividade Indireta

```
13  
void funcA (int a) {  
    if (a < 20) {  
        a++;  
        funcB (a);  
    }  
    printf( "a=%d", a);  
}  
  
void funcB (int b) {  
    if (b < 20) {  
        b=b+2;  
        funcA (b);  
    }  
    printf ("b=%d",b);  
}
```

Chamada de funcA (10)	
1	funcA(10)
2	funcB(11)
3	funcA(13)
4	funcB(14)
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

Recursividade Indireta

```
void funcA (int a) {  
    if (a < 20) {  
        a++;  
        funcB (a);  
    }  
    printf( "a=%d", a);  
}  
  
void funcB (int b) {  
    if (b < 20) {  
        b=b+2;  
        funcA (b);  
    }  
    printf ("b=%d",b);  
}
```

	Chamada de funcA (10)
1	funcA(10)
2	funcB(11)
3	funcA(13)
4	funcB(14)
5	funcA(16)
6	
7	
8	
9	
10	
11	
12	
13	
14	

Recursividade Indireta

16

```
void funcA (int a) {  
    if (a < 20) {  
        a++;  
        funcB (a);  
    }  
    printf( "a=%d", a);  
}  
  
void funcB (int b) {  
    if (b < 20) {  
        b=b+2;  
        funcA (b);  
    }  
    printf ("b=%d",b);  
}
```

	Chamada de funcA (10)
1	funcA(10)
2	funcB(11)
3	funcA(13)
4	funcB(14)
5	funcA(16)
6	funcB(17)
7	
8	
9	
10	
11	
12	
13	
14	

Recursividade Indireta

```
void funcA (int a) {  
    if (a < 20) {  
        a++;  
        funcB (a);  
    }  
    printf( "a=%d", a);  
}  
  
void funcB (int b) {  
    if (b < 20) {  
        b=b+2;  
        funcA (b);  
    }  
    printf ("b=%d",b);  
}
```

Chamada de funcA (10)

funcA(10)

funcB(11)

funcA(13)

funcB(14)

funcA(16)

funcB(17)

funcA(19)

Recursividade Indireta

19

```
void funcA (int a) {  
    if (a < 20) {  
        a++;  
        funcB (a);  
    }  
    printf( "a=%d", a);  
}  
  
void funcB (int b) {  
    if (b < 20) {  
        b=b+2;  
        funcA (b);  
    }  
    printf ("b=%d",b);  
}
```

	Chamada de funcA (10)
1	funcA(10)
2	funcB(11)
3	funcA(13)
4	funcB(14)
5	funcA(16)
6	funcB(17)
7	funcA(19)
8	funcB(20)
9	
10	
11	
12	
13	
14	

Recursividade Indireta

```
void funcA (int a) {  
    if (a < 20) {  
        a++;  
        funcB (a);  
    }  
    printf( "a=%d", a);  
}  
  
20  
void funcB (int b) {  
    if (b < 20) {  
        b=b+2;  
        funcA (b);  
    }  
    printf ("b=%d",b); ←—  
}
```

	Chamada de funcA (10)
1	funcA(10)
2	funcB(11)
3	funcA(13)
4	funcB(14)
5	funcA(16)
6	funcB(17)
7	funcA(19)
8	funcB(20)
9	b=20
10	
11	
12	
13	
6	

Recursividade Indireta

19

```
void funcA (int a) {  
    if (a < 20) {  
        a++;  
        funcB (a);  
    }  
    printf( "a=%d", a); ←—————  
}  
  
void funcB (int b) {  
    if (b < 20) {  
        b=b+2;  
        funcA (b);  
    }  
    printf ("b=%d",b);  
}
```

	Chamada de funcA (10)
1	funcA(10)
2	funcB(11)
3	funcA(13)
4	funcB(14)
5	funcA(16)
6	funcB(17)
7	funcA(19)
8	funcB(20)
9	b=20
10	a=20
11	
12	
13	

Recursividade Indireta

```
void funcA (int a) {  
    if (a < 20) {  
        a++;  
        funcB (a);  
    }  
    printf( "a=%d", a);  
}  
  
17  
void funcB (int b) {  
    if (b < 20) {  
        b=b+2;  
        funcA (b);  
    }  
    printf ("b=%d",b); ←—  
}
```

	Chamada de funcA (10)
1	funcA(10)
2	funcB(11)
3	funcA(13)
4	funcB(14)
5	funcA(16)
6	funcB(17)
7	funcA(19)
8	funcB(20)
9	b=20
10	a=20
11	b=19
12	
13	
8	

Recursividade Indireta

16

```
void funcA (int a) {
    if (a < 20) {
        a++;
        funcB (a);
    }
    printf( "a=%d", a); ←
}

void funcB (int b) {
    if (b < 20) {
        b=b+2;
        funcA (b);
    }
    printf ("b=%d",b);
}
```

Chamada de funcA (10)	
1	funcA(10)
2	funcB(11)
3	funcA(13)
4	funcB(14)
5	funcA(16)
6	funcB(17)
7	funcA(19)
8	funcB(20)
9	b=20
10	a=20
11	b=19
12	a=17
13	

Recursividade Indireta

```
void funcA (int a) {  
    if (a < 20) {  
        a++;  
        funcB (a);  
    }  
    printf( "a=%d", a);  
}  
  
14  
void funcB (int b) {  
    if (b < 20) {  
        b=b+2;  
        funcA (b);  
    }  
    printf ("b=%d",b); ←—  
}
```

	Chamada de funcA (10)
1	funcA(10)
2	funcB(11)
3	funcA(13)
4	funcB(14)
5	funcA(16)
6	funcB(17)
7	funcA(19)
8	funcB(20)
9	b=20
10	a=20
11	b=19
12	a=17
13	b=16

0	

Recursividade

- ◆ Quando uma função é chamada, o endereço de retorno, as variáveis locais, os parâmetros e o valor retornado pela função são guardados **na pilha de execução**
 - Cada chamada recursiva aloca memória para as variáveis locais e parâmetros

Recursividade

```
int Soma (int n)
{
    if (n ==1 ) return 1;
    else {
        int s = Soma(n-1) ;
        return s + n;
    }
}

int main( )
{
    printf("%d",Soma(4));
    return 0;
}
```

Exemplo:

$$\text{Soma}(4) = 4 + 3 + 2 + 1$$

$$\text{Soma}(3) = 3 + 2 + 1$$

$$\text{Soma}(2) = 2 + 1$$

$$\text{Soma}(1) = 1$$

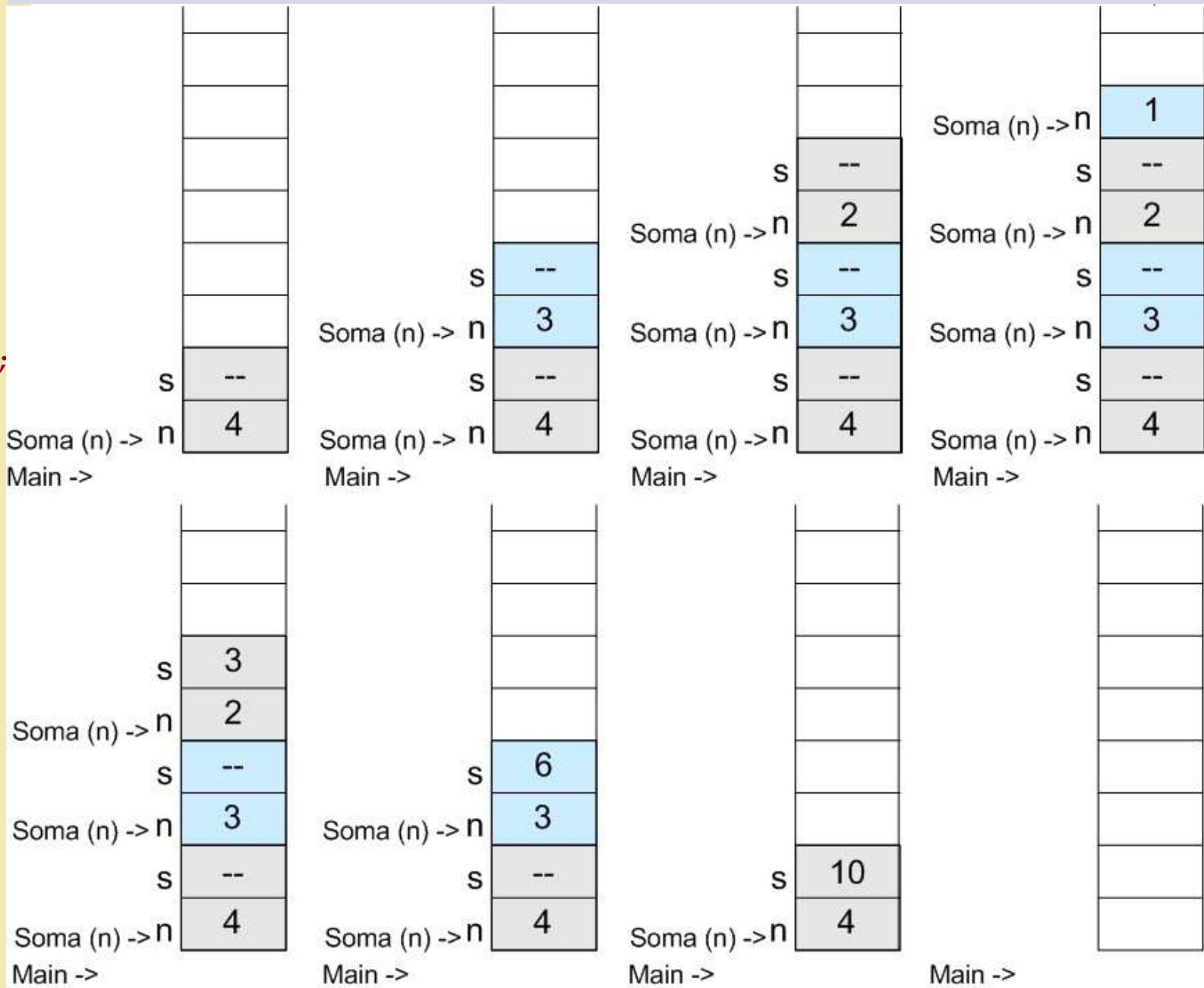
```

int Soma (int n)
{
    if (n ==1 )
        return 1;
    else {
        int s = Soma(n-1);
        return s + n;
    }
}

int main( )
{
    printf("%d",Soma(4));
    return 0;
}

```

Ex.: Esquema representativo da memória na execução passo a passo



Fatorial

Definição matemática:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n.(n-1)! & \text{se } n > 0 \end{cases}$$

$$0! = 1$$

$$1! = 1 \cdot 0! = 1 \cdot 1 = 1$$

$$2! = 2 \cdot 1! = 2 \cdot 1 \cdot 0! = 2$$

$$3! = 3 \cdot 2! = 3 \cdot 2 \cdot 1! \dots = 6$$

$$4! = 4 \cdot 3! = 4 \cdot 3 \cdot 2! \dots = 24$$

Fatorial

◆ Recursivo

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

obs1: critério de parada

obs2: parâmetro de fatorial sempre muda

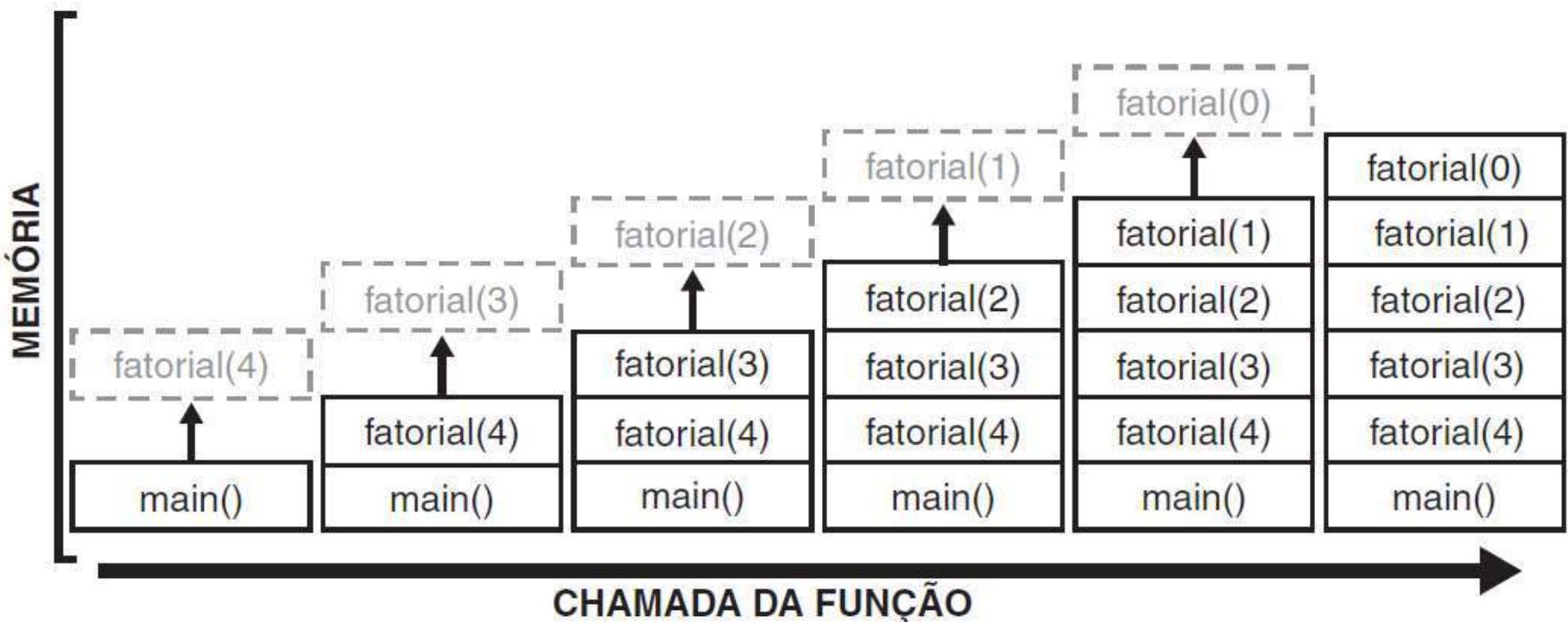
◆ Iterativo

```
int factorial (int n) {  
    if (n == 0)  
        return 1;  
    else{  
        int i;  
        int f = 1;  
        for(i = 1; i <= n; i++)  
            f = f * i;  
        return f;  
    }  
}
```

Recursão

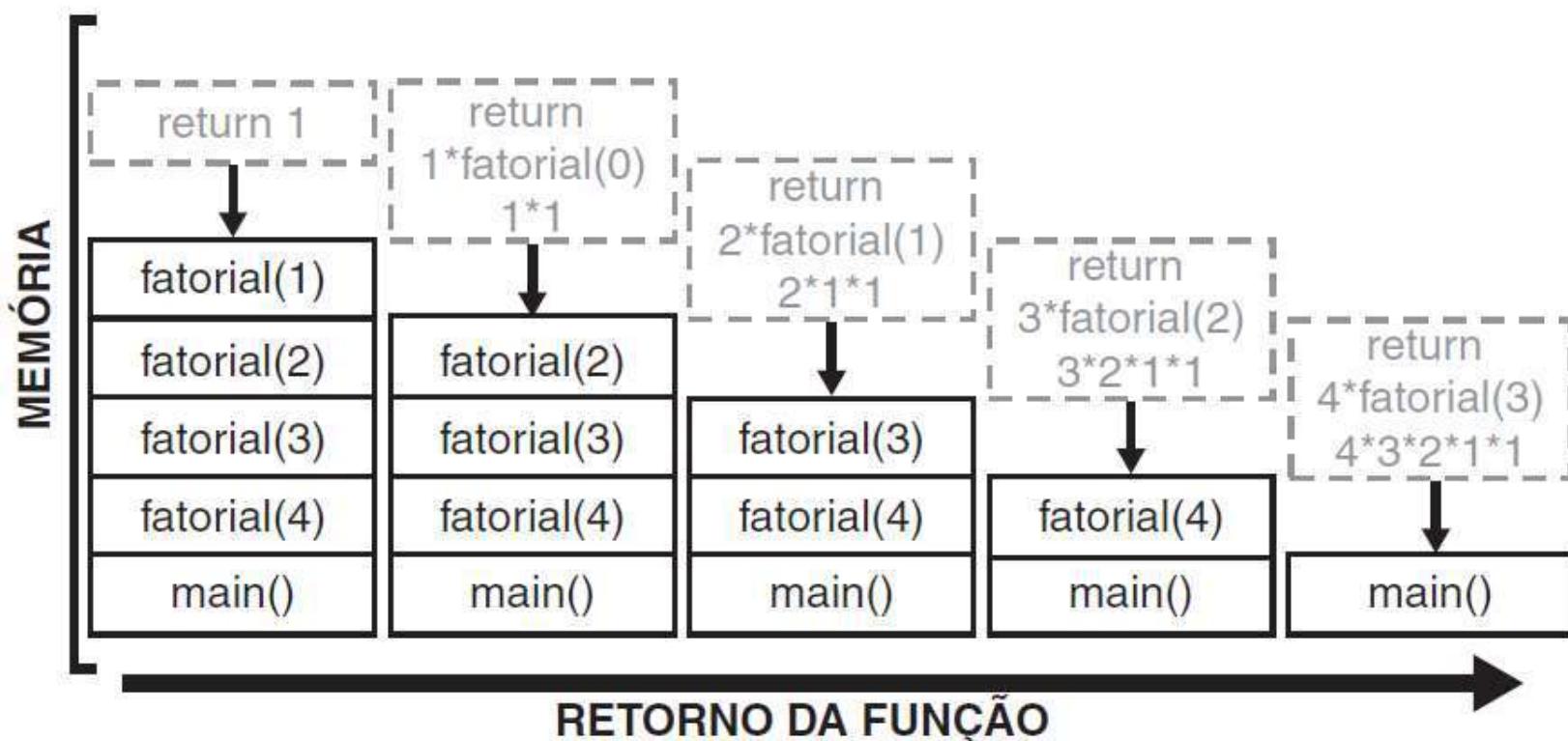
- ◆ O que acontece na chamada da função factorial com um valor como $n = 4$?

```
int x = factorial(4);
```



Recursão

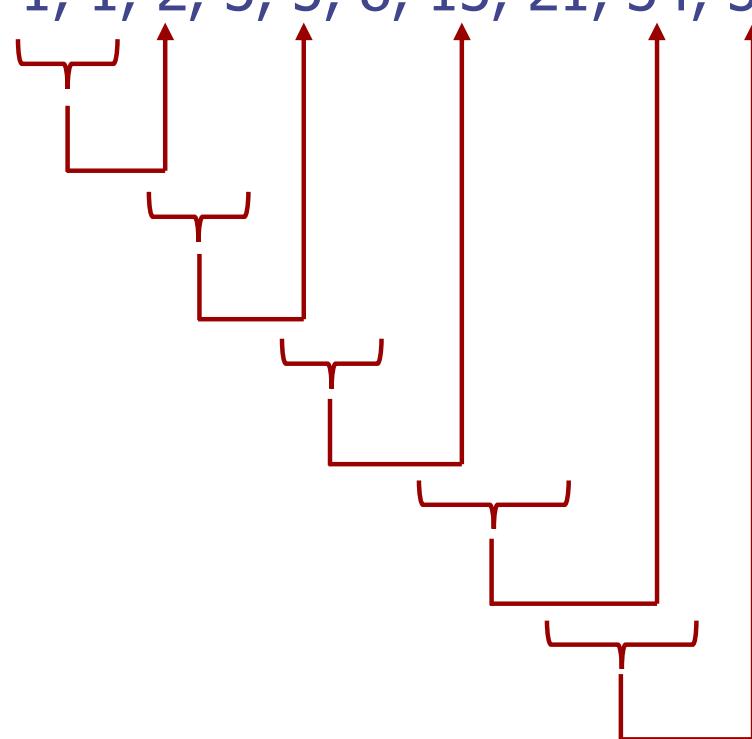
- ◆ Uma vez que chegamos ao caso-base, é hora de fazer o caminho de volta da recursão



Fibonacci

◆ Clássico da recursão

- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...



Fibonacci

◆ Definição Matemática

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

$f(0)$	$= 1$
$f(1)$	$= 1$
$f(2) = f(1) + f(0)$	$= 2$
$f(3) = f(2) + f(1)$	$= 3$
$f(4) = f(3) + f(2)$	$= 5$
$f(5) = f(4) + f(3)$	$= 8$
$f(6) = f(5) + f(4)$	$= 13$

Fibonacci

- ◆ Certos algoritmos são mais eficientes quando feitos de maneira recursiva
 - Contudo, quando usada incorretamente, a recursividade tende a consumir muita memória e ser lenta
 - Uma parcela da memória é reservada cada vez que o computador faz chamada a uma função
 - Compare as duas implementações da série de Fibonacci, uma realizada de forma iterativa e outra de forma recursiva: qual é mais eficiente?

Fibonacci

◆ Fibonacci recursivo:

```
int fibrec ( int n) {  
    if (n == 0 || n == 1)  
        return (1);  
    else  
        return (fibrec(n-1) + fibrec(n-2));  
}
```

Fibonacci

◆ Fibonacci iterativo:

```
int fibit ( int n) {  
    int i, n1 = 1, n2 = 1, F = 1;  
    for (i = 1; i < n; i++) {  
        F = n1 + n2;  
        n1 = n2;  
        n2 = F;  
    }  
    return (F);  
}
```

Fibonacci

- ◆ A versão recursiva é ineficiente
 - recalcula o mesmo valor várias vezes
 - Exemplo

$$f(5) = f(4) + f(3)$$



$$f(4) = f(3) + f(2)$$

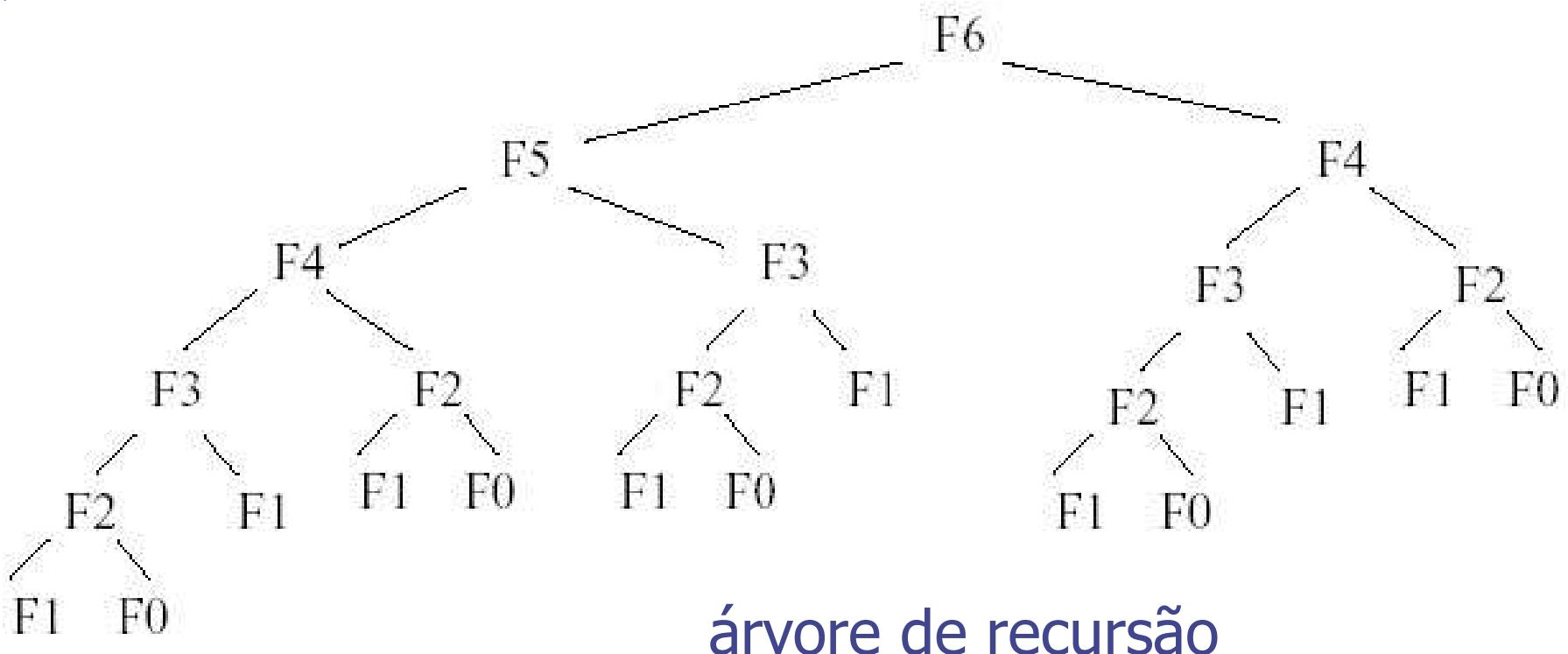
$f(3)$ vai ser calculado 2 vezes

$f(2)$ vai ser calculado 3 vezes

$f(1)$ vai ser calculado 5 vezes

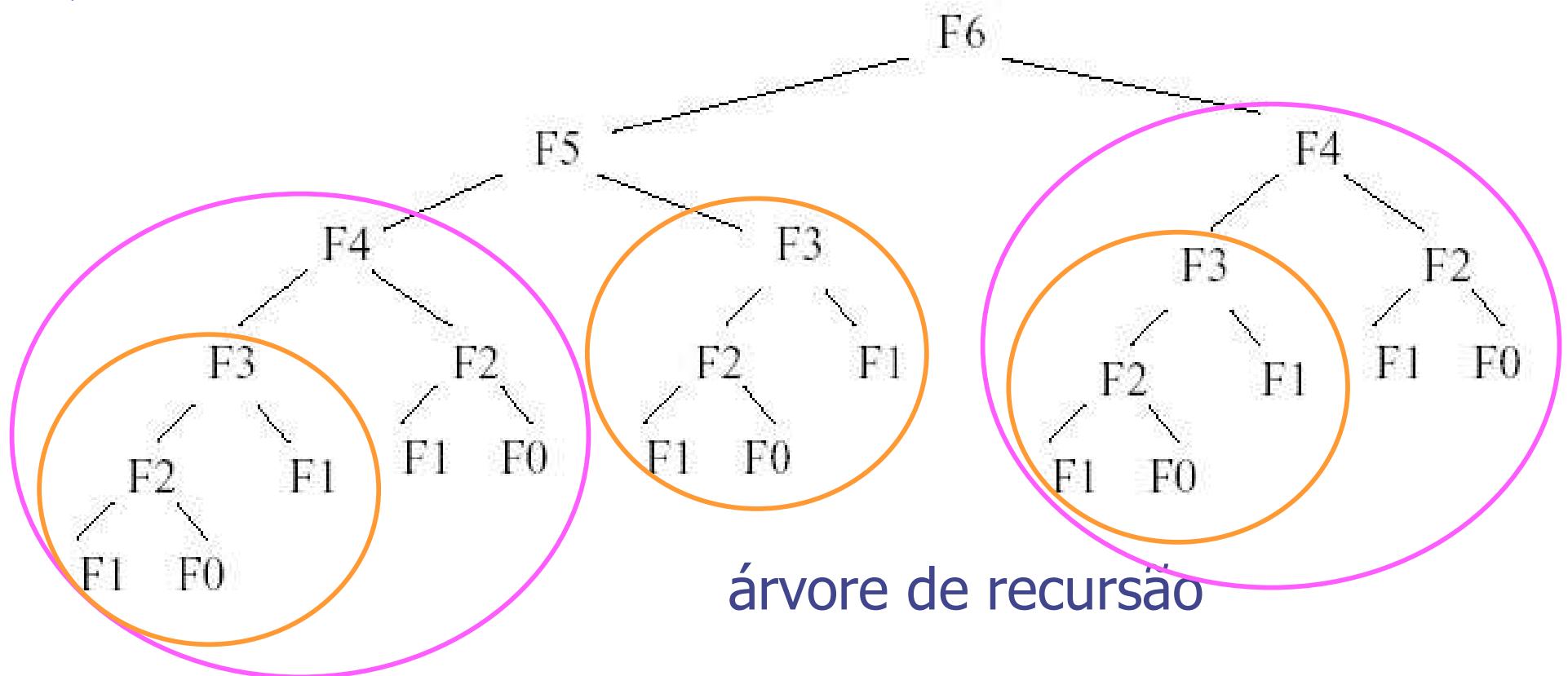
Fibonacci

- ◆ A versão recursiva é ineficiente
 - recalcula o mesmo valor várias vezes



Fibonacci

- ◆ A versão recursiva é ineficiente
 - recalcula o mesmo valor várias vezes



Fibonacci

◆ Comparação das funções FibRec e FibIter

n	10	20	30	50	100
FibRec	8 ms	1s	2 min	21 dias	10^9 anos
FibIter	1/6 ms	1/3 ms	½ ms	¾ ms	1,5 ms

Fonte: Brassard e Bradley, 1996. Citado em Projeto de Algoritmos, Nivio Ziviane.

Fibonacci

◆ Concluindo

- Quando não usar recursividade?
- Programas com característica de recursividade de cauda
 - ◆ são facilmente transformáveis em uma versão não recursiva
- Deve-se evitar o uso de recursividade quando existir uma solução óbvia por iteração

Exercício

Considere um vetor de inteiros. Escreva uma versão recursiva para achar o elemento máximo do vetor

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

int main ( ) {
    int A[5] = {5, 4, 7, 2, 1};
    printf("%d", Max(A, 5));
    return 0;
}
```

Mais sobre recursividade

- ◆ Alguns problemas podem parecer não serem recursivos
- ◆ Exemplo:
 - Encontrar o **Máximo Divisor Comum** entre dois inteiros
 - ◆ Pode ser resolvido recursivamente?

Mais sobre recursividade

◆ Cálculo do MDC pelo processo das divisões sucessivas

- Nesse processo efetuamos várias divisões até chegar a uma divisão exata. O divisor desta divisão é o mdc

Mais sobre recursividade

- ◆ Acompanhe o cálculo do mdc de 48 e 30
 - Regra prática:
 - 1º) dividimos o número maior pelo número menor
 $\rightarrow 48 / 30 = 1$ (com resto 18)
 - 2º) dividimos o divisor 30, que é divisor da divisão anterior, por 18, que é o resto da divisão anterior, e assim sucessivamente
 - ◆ $30 / 18 = 1$ (com resto 12)
 - ◆ $18 / 12 = 1$ (com resto 6)
 - ◆ $12 / 6 = 2$ (com resto zero - divisão exata)
 - 3º) O divisor da divisão exata é 6. Então mdc de 48 e 30 = 6

Mais sobre recursividade

◆ MDC iterativo:

```
int mdc_it(int a, int b) {  
    int resto;  
  
    while (b != 0) {  
  
        resto = a % b;  
        a = b;  
        b = resto;  
    }  
    return a;  
}
```

Mais sobre recursividade

MDC recursivo

- ◆ Máximo Divisor Comum de A e B
 - Caso base?
 - Chamadas recursivas?

Mais sobre recursividade

MDC recursivo

- ◆ Máximo Divisor Comum de A e B
 - Caso base?
 - ◆ MDC de A e B é A se B = 0
 - Chamadas recursivas?
 - ◆ MDC de A e B é MDC(B, resto da divisão de A por B)

Mais sobre recursividade

◆ MDC recursivo:

```
int mdc_rec(int a, int b) {  
  
    if (b == 0)  
        return a;  
  
    else  
        return mdc_rec(b, a % b);  
}
```

- caso base: condição em que facilmente se resolve o problema

Problemas Recursivos

- ◆ É importante considerar problemas de eficiência no momento da codificação para decidir por um algoritmo recursivo ou não
 - A cada chamada a função cria um novo registro de ativação, com variáveis e objetos locais, há um consumo de memória além do que qualquer chamada de função gasta um certo tempo para operações de controle interno:
 - salvamento de contexto,
 - alocação de memória,
 - passagem e recuperação de parâmetros, etc

Programação Dinâmica

◆ Recursividade

- é uma técnica muito útil quando o problema a ser resolvido puder ser dividido em sub-problemas a um custo não muito grande e os tamanhos dos subproblemas possam ser mantidos pequenos
- esse não é o caso quando um problema de tamanho n resulta em n subproblemas de tamanho $n-1$
- nesse último caso a técnica de programação dinâmica pode levar a um algoritmo mais eficiente

Programação Dinâmica

- ◆ A programação dinâmica calcula a solução para todos os subproblemas, partindo dos subproblemas menores para os maiores, armazenando os resultados em uma tabela
- ◆ A vantagem está no fato de que uma vez que um subproblema é resolvido, a resposta é guardada em uma tabela e nunca mais é recalculado
- ◆ A programação dinâmica pode ser definida como “recursão com o apoio de uma tabela”

Programação Dinâmica

- ◆ Para que este paradigma da programação possa ser aplicado, é preciso que o problema tenha estrutura recursiva: a solução de toda instância do problema deve "conter" soluções de subinstâncias da instância
- ◆ O algoritmo recursivo é tipicamente ineficiente porque refaz, muitas vezes, a solução de cada subinstância

Programação Dinâmica

- ◆ Uma vez escrito o algoritmo recursivo é fácil construir uma tabela para armazenar as solução das subinstâncias e assim evitar que elas sejam recalculadas
- ◆ A tabela é a base de um algoritmo de programação dinâmica.

Programação Dinâmica

◆ Exemplo:

- Sequência de Fibonacci
 - ◆ com memorização

Relembrando...

Sequência de Fibonacci

- ◆ Fibonacci recursivo:

```
int fibrec ( int n) {  
if (n == 0 || n == 1)  
    return (1);  
else  
    return (fibrec(n-1) + fibrec(n-2)) ;  
}
```

Relembrando...

Sequência de Fibonacci

◆ Fibonacci iterativo:

```
int fibit ( int n) {  
    int i, n1 = 1, n2 = 1, F = 1;  
    for (i = 1; i < n; i++) {  
        F = n1 + n2;  
        n1 = n2;  
        n2 = F;  
    }  
    return (F);  
}
```

Sequência de Fibonacci

- ◆ Fibonacci programação dinâmica:

```
#include <stdio.h>

long long f[40];
int pos = 2;

long long fb(int n) {
    f[0] = 1; f[1] = 1;
    int i;
    for (i = pos; i <= n; i++)
        f[i] = f[i-2] + f[i-1];
    if (n > 1)
        pos = n; // guarda até onde já calculou
    return f[n];}
```

Sequência de Fibonacci

- ◆ Fibonacci programação dinâmica:

...

```
int main(int argc, char** argv) {  
    int i;  
    for (i = 0; i < 40; i++)  
        printf("%lld ", fb(i));  
}
```

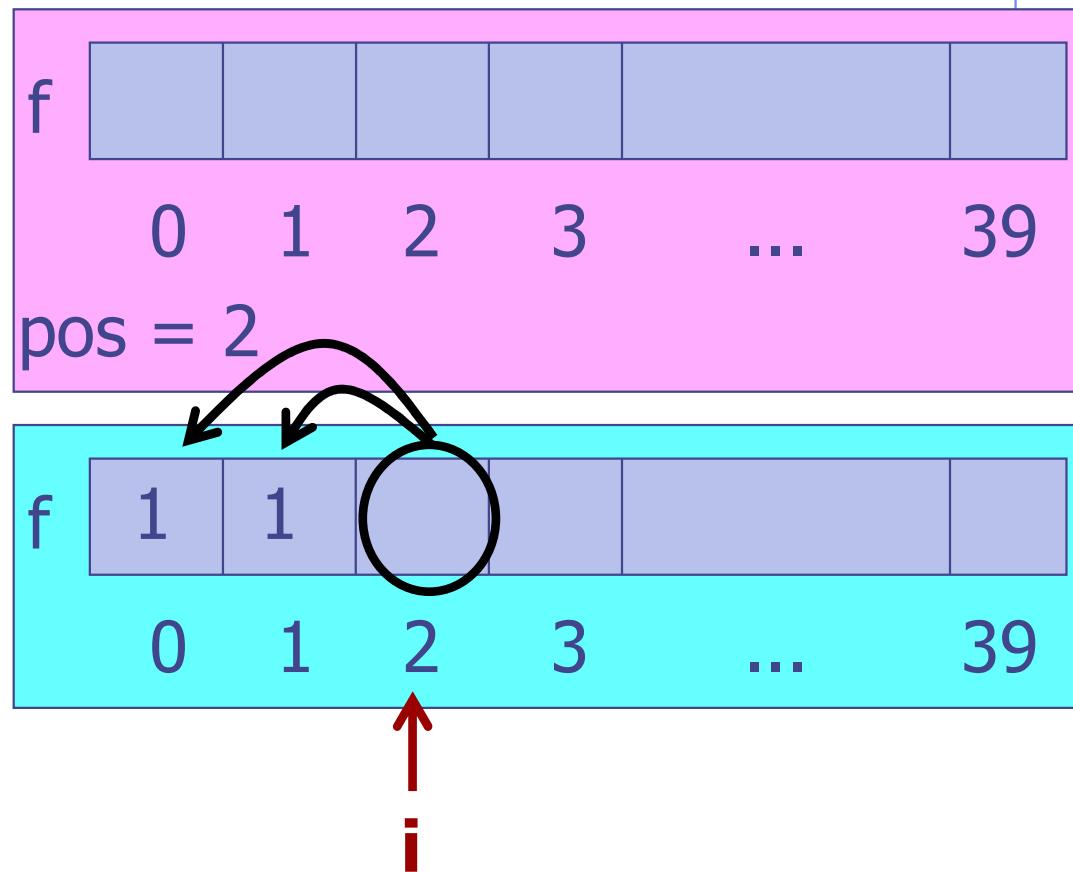
Sequência de Fibonacci

$n = 2$

```
#include <stdio.h>

long long f[40];
int pos = 2;

long long fb(int n) {
    f[0] = 1; f[1] = 1;
    int i;
    for (i = pos; i <= n; i++)
        f[i] = f[i-2] + f[i-1];
    if (n > 1)
        pos = n;
    return f[n];
}
```



$pos = 2 \rightarrow$ guarda a última
posição calculada

Programação dinâmica

◆ Aspectos do algoritmo:

- Uma vez que calculamos os números de Fibonacci, começando com os menores e armazenando os resultados, teremos já calculado os valores de F_{i-1} e F_{i-2} quando computarmos o valor de F_i

Visão geral da Análise de Algoritmos Recursivos

- ◆ Quando um algoritmo possui uma chamada recursiva, o seu tempo de execução pode ser descrito por uma recorrência
- ◆ Uma recorrência é uma equação ou desigualdade que descreve uma função em termos de seus valores sobre entidades menores
- ◆ Muitos algoritmos se baseiam em recorrência
- ◆ Exemplo do **cálculo do fatorial**

Definição Matemática	Implementação	Relação de recorrência
$N! = N * (N-1)!$ $0! = 1$	<pre>int factorial(int n) { if (n == 0) return 1; else return n * factorial(n-1); }</pre>	$T(n) = T(n-1) + 1, n > 1$ $T(1) = 0$

Visão geral da Análise de Algoritmos Recursivos

- ◆ Complexidade da recorrência
 - Uma recursão usualmente não utiliza estruturas de repetição, apenas comandos condicionais, atribuições, etc.
 - Podemos erroneamente imaginar que essas funções possuem complexidade **$O(1)$**
 - Saber a complexidade da recursão envolve resolver a sua relação de recorrência

Visão geral da Análise de Algoritmos Recursivos

- ◆ Exemplo de derivação da equação de recorrência:

```
1. int busca_binaria(int A[],int x, int esq, int dir)
2. {
3.     int mid;
4.
5.     if (esq > dir)
6.         return(-1);
7.     mid = (esq+dir)/2; /*cálculo do índice mediano do array*/
8.     if (x == A[mid])
9.         return mid;
10.    else if (x < A[mid])
11.        busca_binaria(A,x,esq,mid-1);
12.    else
13.        busca_binaria(A,x,mid+1,dir);
14. }
```

Busca binária

Visão geral da Análise de Algoritmos Recursivos

- ◆ Exemplo de derivação da equação de recorrência:
 - Analisando o algoritmo de busca binária
 - A cada chamada recursiva nas linhas 11 e 13 o conjunto de dados que deve ser analisado diminui pela metade
 - Se $T(n)$ é o tempo gasto para executar o algoritmo em um array de tamanho n , temos que $T(n) = T(n/2)+c_2$ onde c_2 é o tempo constante gasto para executar as linhas 5, 7, 8 10 e 11 (ou 13)
 - Se $n = 1$ a chamada recursiva é feita apenas uma vez logo podemos considerar que $T(1) = c_1$
 - Logo, o tempo gasto é descrito pela função de recorrência

$$T(n) = \begin{cases} c_1 & , \text{ se } n = 1 \\ T\left(\frac{n}{2}\right) + c_2, & \text{se } n > 1 \end{cases} \quad (1)$$

Visão geral da Análise de Algoritmos Recursivos

- ◆ Para encontrarmos um limite para $T(n)$ no pior caso é necessário resolver a recorrência
- ◆ Existem três métodos bem conhecidos para a resolução de recorrências
 - Método da substituição
 - Método iterativo
 - Método mestre
- ◆ O estudo de tais métodos foge ao escopo da disciplina
 - Para maiores detalhes veja (Cormen et al., Algoritmos – Teoria e Prática, 2002)

Visão geral da Análise de Algoritmos Recursivos

- ◆ Exemplo de resolução de equação de recorrência
 - ◆ Método iterativo: A ideia é expandir a recorrência e expressá-la como uma soma de termos dependentes de n e das condições iniciais
 - ◆ Considerando n como uma potência de 2, a recorrência (1) pode ser expressa como:
$$\begin{aligned} T(n) &= T(n/2) + c2 \\ &= (T(n/4) + c2) + c2 = T(n/4) + 2c2 \\ &= (T(n/8) + c2) + 2c2 = T(n/8) + 3c2 \end{aligned}$$
 - ◆ Cada linha da expressão acima é da forma $T(n/2^k) + k * c2$ e a iteração irá continuar até que $2^k = n$ ou $k = \log(n)$ (supondo log na base 2). Aplicando na função $T(n)$:

$$\begin{aligned} T(n) &= T(n/2^{\log(n)}) + k * c2 = T(n/n) + k * c2 \\ &= T(1) + k * c2 = c1 + \log(n) * c2, \\ &\text{portanto } T(n) = O(\log(n)) \end{aligned}$$

Para praticar

- 1) Implemente uma função recursiva para calcular o valor de 2^n .
- 2) Construa um algoritmo que calcule o fatorial de um número utilizando programação dinâmica. Compare o resultado obtido com os algoritmos iterativo e recursivo apresentados em aula.

Referências

- ◆ ZIVIANI, N. Projeto de Algoritmos. 2a edição, Editora Thomson.
- ◆ CORMEN, THOMAS H. et. al. Algoritmos: Teoria e Prática. Editora Campus, 2002.
- ◆ Slides do Prof. André Backes. Linguagem C Descomplicada – Portal de vídeo-aulas para estudo de programação.

Material Complementar

◆ Vídeo Aulas

- Aula 51: Recursão pt. 1 - Definição:
<http://youtu.be/T2gTc5u-i1o>
- Aula 52: Recursão pt. 2 - Funcionamento:
<http://youtu.be/FH5ICr-RVWE>
- Aula 53: Recursão pt. 3 - Cuidados:
<http://youtu.be/o3MPTEc3LD8>
- Aula 54: Recursão pt. 4 – Soma de 1 até N:
<http://youtu.be/YEeYk9uEqEI>
- Aula 122 – Relações de Recorrência:
<http://youtu.be/QeLYRyW5T94>
- Resolvendo relações de recorrência – Fatorial
<https://www.youtube.com/watch?v=iOZMQHW30VA&t=326s>

Sugestão de estudo

- ◆ Lista de exercícios sobre recursão
 - Disponível na Equipe no MS Teams

Ordenação

Profa. Maria Camila Nardini Barioni

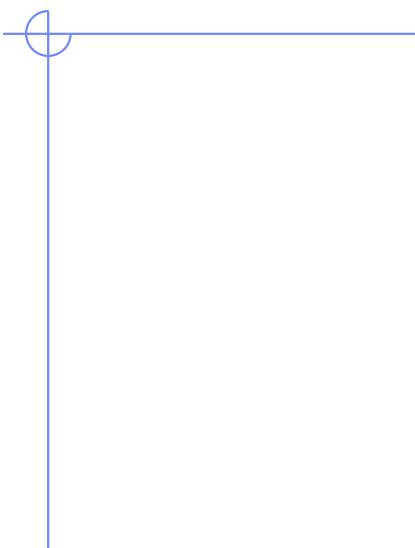
camila.barioni@ufu.br

Bloco B - sala 1B137

2º semestre de 2023

Roteiro

- ◆ Motivação
- ◆ O problema de Ordenação
- ◆ Relembrando Complexidade
- ◆ Métodos de Ordenação
 - ◆ Parte 1
 - ◆ Método da Bolha (Bubble Sort)
 - ◆ Método da Seleção (Selection Sort)
 - ◆ Parte2
 - ◆ Método da Inserção (Insertion Sort)
 - ◆ Shell Sort



PARTE 1

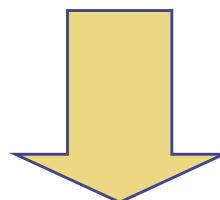
Motivação

- ◆ Processo bastante utilizado na computação de uma estrutura de dados (ED)
- ◆ Dados ordenados garantem melhor desempenho em pesquisa e listagem dos dados de uma ED
- ◆ Compromisso (trade-off)
 - A complexidade da ordenação da ED não deve exceder a complexidade da computação na ED sem a ordenação

O problema da Ordenação

- ◆ Dado um conjunto com elementos que podem ser comparados, colocar os elementos em ordem crescente ou decrescente

0	1	2	3	4	5	6	7	8
60	30	10	20	40	90	70	80	50



0	1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80	90

Relembrando Complexidade

- ◆ De maneira simples “complexidade” pode ser entendida como o número de comparações necessárias para realizar uma operação
 - Todos os métodos complexos têm comandos de repetição e a complexidade é calculada sobre a **operação mais custosa** (em termos de tempo de execução) que é **repetida várias vezes**
- ◆ A complexidade define o desempenho de um método
- ◆ Com N elementos a complexidade pode ser da ordem de N, N^2 , $N \log_2(N)$, etc
 - Notação: $O(N) = k N$ (k é uma constante)
 - Ou seja, um método $O(N)$ faz “em torno de” ou “da ordem de” N comparações

Relembrando Complexidade: $O(N^2)$

- ◆ Algoritmos com dois laços aninhados, cada um com n repetições, executam n^2 o conteúdo do laço interno
 - Supondo que o conteúdo é executado em tempo $O(1)$, então a complexidade é $O(N^2)$
- ◆ Algoritmos com dois laços aninhados, onde o externo é repetido n vezes e o interno é repetido n vezes da primeira repetição do externo, $n-1$ na segunda, ..., fazem $n + n-1 + n-2 + \dots + 1$ repetições =
 - A complexidade também é $O(N^2)$

Considerações

- ◆ Dados estão mantidos em um vetor ou outra ED
- ◆ Elemento do vetor
 - Objeto que possui um atributo **chave** que deve ser mantido ordenado
 - Chave: nome, número de matrícula, idade, nota, etc.
- ◆ As chaves dos elementos são **comparadas** e se necessário elas são **trocadas** de posição
- ◆ Um método *troca(x,y)* realiza a troca dos elementos presentes nas posições **x** e **y** do vetor

Considerações

◆ Os algoritmos de ordenação podem ser classificados em dois grupos

■ Ordenação interna

- ◆ O conjunto de dados a ser ordenado cabe todo na memória principal (RAM)
- ◆ Qualquer elemento pode ser imediatamente acessado

Considerações

◆ Os algoritmos de ordenação podem ser classificados em dois grupos

■ Ordenação externa

- ◆ O conjunto de dados a ser ordenado não cabe na memória principal
- ◆ Os dados estão armazenados em memória secundária (por exemplo, um arquivo)
- ◆ Os elementos são acessados sequencialmente ou em grandes blocos

Características de Ordenação

- ◆ Algoritmo “**no lugar**” (in-place)

- Não usam memória adicional (ou então a memória adicional é desprezível em relação a N)

- ◆ Algoritmo **estável**

- Preserva a ordem relativa que os elementos com mesmo valor de chave tinham antes da ordenação

Características de Ordenação

◆ Exemplo

- Dados não ordenados
 - ◆ **5a**, 2, **5b**, 3, 4, 1
 - ◆ **5a** e **5b** são o mesmo número
- Dados ordenados
 - ◆ 1, 2, 3, 4, 5a, 5b: ordenação **estável**
 - ◆ 1, 2, 3, 4, 5b, 5a: ordenação **não-estável**

Métodos de Ordenação

◆ Ordenação por troca

- *BubbleSort* (método da bolha)
- *QuickSort* (método da troca e partição)

◆ Ordenação por inserção

- *InsertionSort* (método da inserção direta)
- *ShellSort* (método otimizado da inserção)

◆ Ordenação por seleção

- *SelectionSort* (método da seleção direta)
- *HeapSort* (método da seleção em árvore)

◆ Outros métodos

- *MergeSort* (método da intercalação)
- *BucketSort* (método da distribuição de chave)

Métodos de Ordenação

- ◊ Podem ser classificados em
 - Métodos simples
 - Métodos eficientes

Métodos Simples

◆ Exemplos

- *BubbleSort, InsertionSort, SelectionSort*

◆ Características

- Fácil implementação
- Alta complexidade computacional: $O(N^2)$
- Adequados para lidar com pequenos arquivos

◆ Dividem o vetor em segmento ordenado e segmento não ordenado

- No início o segmento não ordenado é o vetor inteiro
- No final, o segmento ordenado é o vetor inteiro

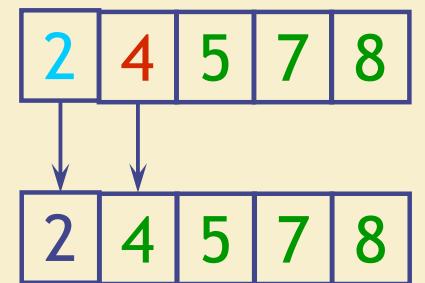
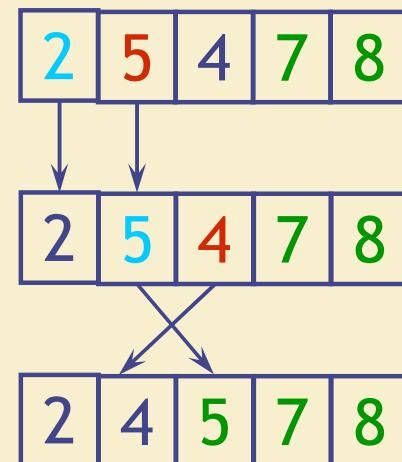
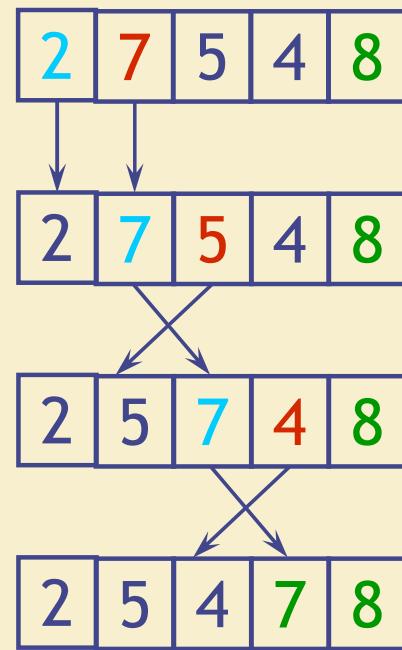
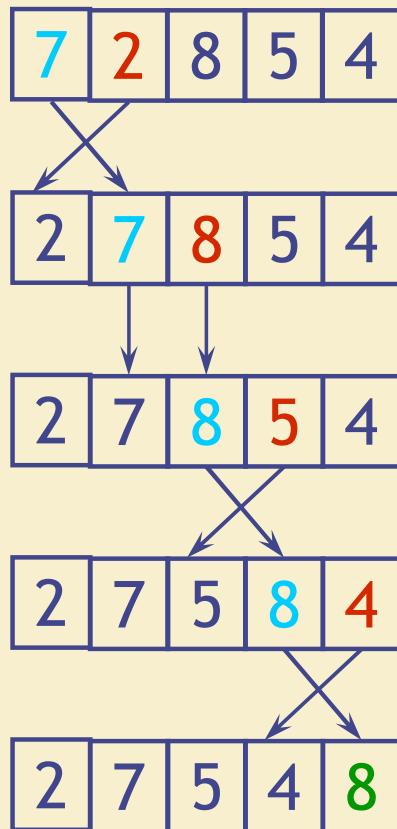
Método da Bolha (BubbleSort)

- ◆ É um método simples de troca
 - Ordena por meio de sucessivas trocas entre pares de elementos do vetor
- ◆ Características
 - Realiza varreduras no vetor, trocando **pares adjacentes** de elementos sempre que o próximo elemento for menor que o anterior
 - Após uma varredura, o maior elemento está corretamente posicionado no vetor
 - ◆ após a i -ésima varredura, os i maiores elementos estão ordenados
 - A cada varredura o maior elemento é inserido no segmento ordenado (arrastado) e removido do segmento desordenado

Método da Bolha (BubbleSort)

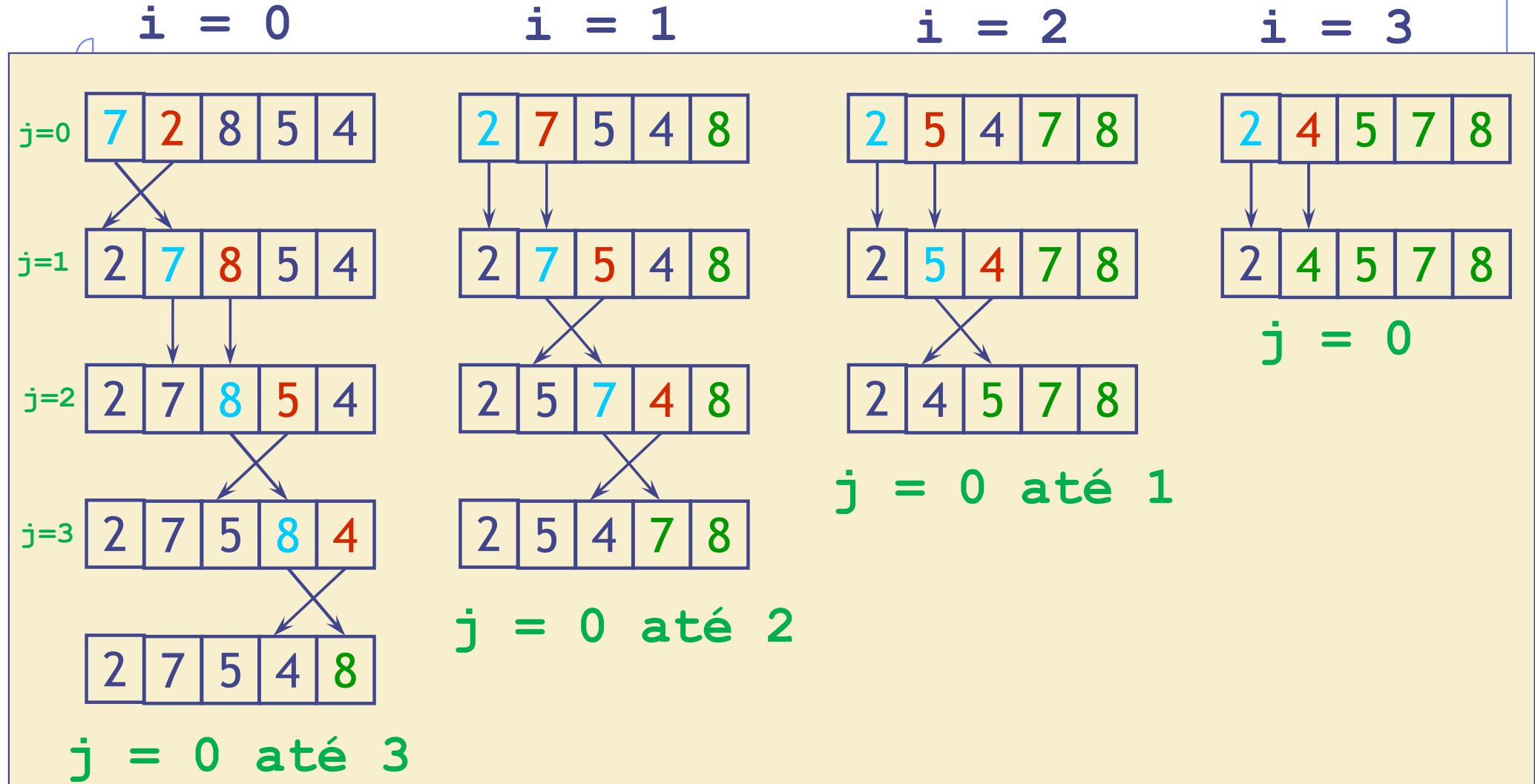
- ◆ É o mais antigo método de ordenação e o mais ineficiente (exige mais comparações e trocas)
- ◆ Faz com que os maiores valores sejam empurrados para o final do vetor em trocas sucessivas
 - Isso pode ser visualizado como uma “bolha” que percorre o vetor
- ◆ Vantagem
 - Simples de implementar
- ◆ Desvantagem
 - Desempenho: Complexidade $O(N^2)$ no melhor e no pior caso
 - No entanto, existem aprimoramentos com melhor desempenho

Exemplo do BubbleSort



Pensando no algoritmo...

$\text{tam} = 5$



Comparando posições $j+1$ e j

Condição de parada do 2º laço: $\text{tam} - 1 - i$

BubbleSort - Trocas

- ◆ Link – Animações:

- <https://visualgo.net/en/sorting>
- <https://yongdanielliang.github.io/animation/web/BubbleSortNew.html>

BubbleSort em C

```
#include <stdio.h>

void bubbleSort(int vetor[], int tam) {
    int temp, i, j;

    for (i = 0; i < tam ; i++) {
        for (j = 0; j < tam - 1 - i; j++) {
            if (vetor[j+1] < vetor[j]) {
                temp = vetor[j];
                vetor[j] = vetor[j+1];
                vetor[j+1] = temp;
            }
        }
    }
}
```

BubbleSort em C

```
int main (void) {  
    int i;  
    int v[8]= {25, 48, 37, 12, 57, 86, 33, 92};  
  
    bubbleSort(v,8);  
    printf("vetor ordenado: ");  
  
    for(i=0;i<8;i++)  
        printf("%d ", v[i]);  
  
    printf("\n");  
    return 0;  
}
```

BubbleSort - Comparações

- ◆ Quantas **operações de comparação** são necessárias para ordenar um conjunto de n elementos?
- ◆ A comparação `vetor[j+1] < vetor[j]` é executada $n-1$ vezes quando $i = 0$, $n-2$ vezes quando $i = 1$, $n-3$ vezes quando $i = 3$, ...
- ◆ O número total de comparações é: $O(n^2)$

BubbleSort - Trocas

- ◆ Quantas **operações de troca** de posição são necessárias para ordenar um conjunto de n elementos?
- ◆ No pior caso, a troca entre `vetor[j]` e `vetor[j+1]` é executada o mesmo número de vezes que a comparação
- ◆ O número total de atribuições é: $O(n^2)$

BubbleSort

Exercício

- ◆ Como o código apresentado poderia ser alterado para evitar que o processo continue mesmo depois de o vetor estar ordenado?

```
#include <stdio.h>

void bubbleSort(int vetor[], int tam) {
    int temp, i, j;

    for (i = 0; i < tam ; i++) {
        for (j = 0; j< tam - 1 - i; j++) {
            if (vetor[j+1] < vetor[j]) {
                temp = vetor[j];
                vetor[j] = vetor[j+1];
                vetor[j+1] = temp;
            }
        }
    }
}
```

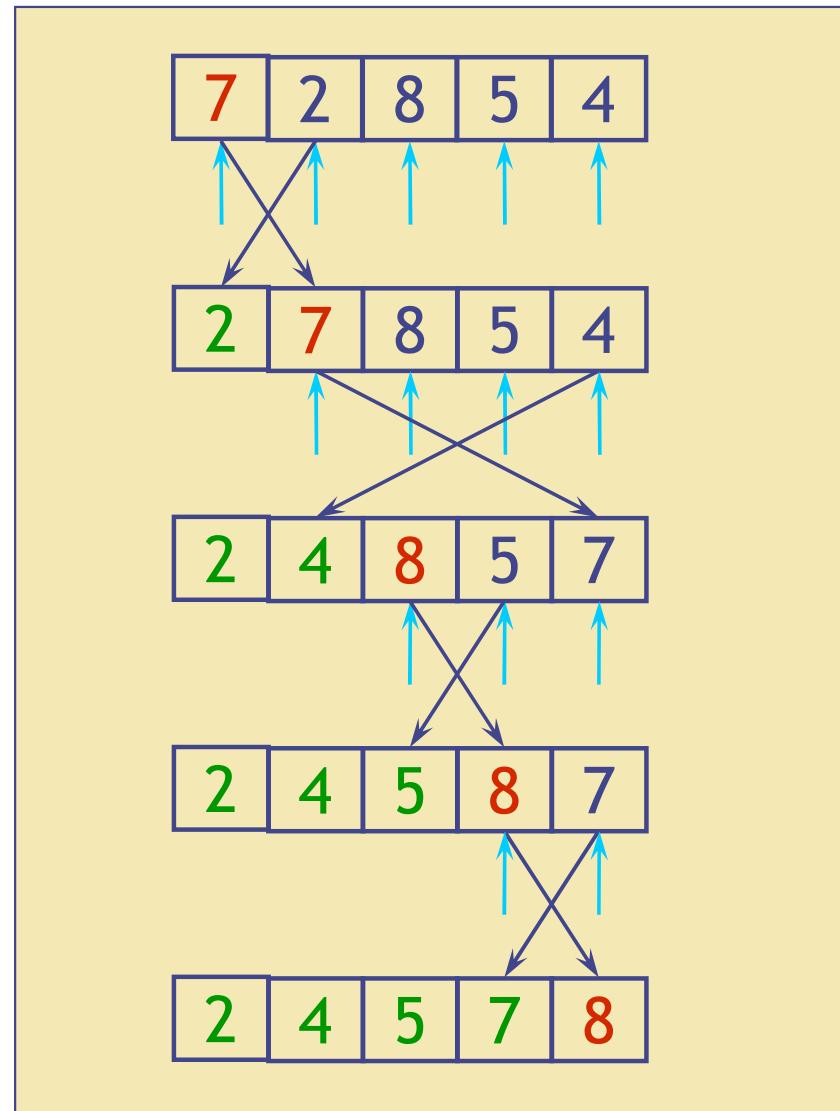
Método da Seleção (SelectionSort)

- ◆ Em cada passo (do início ao final)
 - Encontra o elemento de menor valor (da chave) do segmento não ordenado
 - Substitui pelo primeiro elemento desse segmento
 - Diminui o tamanho do segmento não ordenado e repete o passo
 - Alternativamente, pode encontrar o maior valor e substituir pelo último elemento
- ◆ A cada varredura o menor elemento é inserido no segmento ordenado (trocado) e removido do segmento desordenado

Método da Seleção (SelectionSort)

- ◆ Os elementos são adicionados ao segmento ordenado na ordem correta
- ◆ Vantagem
 - Simples de implementar
 - Ótimo desempenho quanto ao número de movimentos de registros
 - Algoritmo a ser considerado para arquivos com registros muito grandes
- ◆ Desvantagem
 - Baixo desempenho: Complexidade $O(N^2)$ no melhor e no pior caso
 - O algoritmo não é estável

Exemplo do SelectionSort



SelectionSort -

- ◆ Link – Animação:
 - <https://visualgo.net/en/sorting>

SelectionSort em C

```
void selectionSort(int vetor [], int n) {  
    int temp;  
    int menor, i, j;  
  
    for (i = 0; i < n - 1; i++) {  
        menor = i;  
        for (j = i + 1; j < n; j++) {  
            if (vetor[j] < vetor[menor]) {  
                menor = j;  
            }  
        }  
        temp = vetor[i];  
        vetor[i] = vetor[menor];  
        vetor[menor] = temp;  
    }  
}
```

SelectionSort - Comparações

- ◆ Quantas **operações de comparação** são necessárias para ordenar um conjunto de n elementos?
- ◆ A comparação `vetor[j] < vetor[menor]` é executada n-1 vezes quando $i = 0$, n-2 vezes quando $i = 1, \dots, 1$ vez quando $i = n-2$
- ◆ O número total de comparações é:
 - $n(n-1) = O(n^2)$

SelectionSort - Trocas

- ◆ Quantas **operações de troca** de posição são necessárias para ordenar um conjunto de n elementos?
- ◆ No pior caso, a troca entre `vetor[i]` e `vetor[menor]` é executada n-1 vezes
- ◆ Cada troca custa 3 atribuições
- ◆ O número total de atribuições é:
 - $3(n-1) = O(n)$

Comparando os dois métodos

- ◆ BubbleSort vs SelectionSort
- ◆ Link:
 - <https://www.toptal.com/developers/sorting-algorithms>

Exercícios

- ◆ Dado o vetor $X = [7, 5, 4, 2, 1, 8, 6, 10]$, ordená-lo em ordem crescente, detalhadamente, pelo método da bolha.

- ◆ Dado o vetor $X = [5, 4, 2, 6, 8, 3, 7, 1]$, ordená-lo em ordem crescente, detalhadamente, pelo método da seleção.

Bacharelado em Ciência da Computação
GBC034 Algoritmos e Estruturas de Dados 2

Ordenação

Profa. Maria Camila Nardini Barioni

camila.barioni@ufu.br

Bloco B - sala 1B137

2º semestre de 2023

Avisos

- ◆ Informações sobre a **monitoria da disciplina**
 - ◆ **Monitor:** Mateus Eurípedes
 - ◆ **Contato:** mateus.soares@ufu.br
 - ◆ **Horário da monitoria presencial:** Terças-feiras das 09:00 às 10:30 e Quartas-feiras das 13:00 às 14:50 no Lab01 (agendar por e-mail ou Teams)
 - ◆ **Atendimento pelo Teams:** Entrar em contato para agendar chamada quartas e sextas após 20:30h e sábados entre 14h e 16h

Roteiro

- ◆ Motivação
- ◆ O problema de Ordenação
- ◆ Relembrando Complexidade
- ◆ Métodos de Ordenação
 - ◆ Parte 1
 - ◆ Método da Bolha (Bubble Sort)
 - ◆ Método da Seleção (Selection Sort)
 - ◆ Parte 2
 - ◆ Método da Inserção (Insertion Sort)
 - ◆ Shell Sort

PARTE 2

Método da Inserção (InsertionSort)

- ◆ Baseia-se no conceito de inserção de elemento em fila ordenada
- ◆ Pega um elemento no segmento não ordenado e insere no lugar devido no conjunto ordenado
- ◆ Não há “troca” de elementos
 - Quando se acha a posição de um elemento, vários outros podem vir a ser movidos

Método da Inserção (InsertionSort)

◆ Vantagens

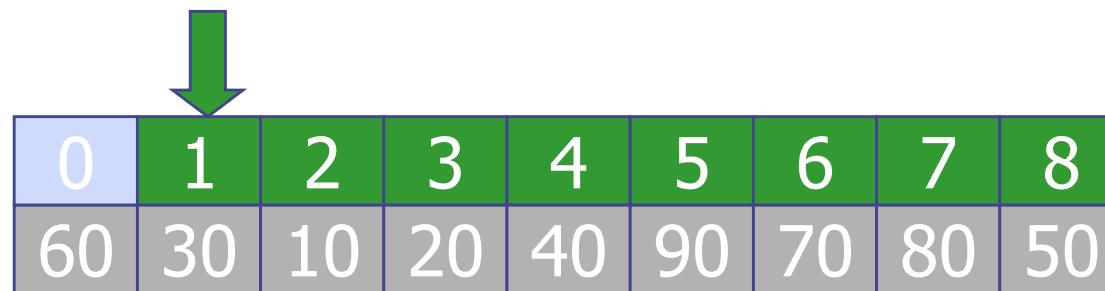
- Simples de implementar
- Método **estável**
- Para arquivos já ordenados, descobre a um custo $O(N)$ que cada item já está no lugar

◆ Desvantagens

- Baixo desempenho: Complexidade $O(N^2)$ no pior caso
- Ineficiente com conjuntos grandes de elementos

InsertionSort

- ◆ Segmento ordenado (elemento 0) e segmento desordenado (1 a 8)



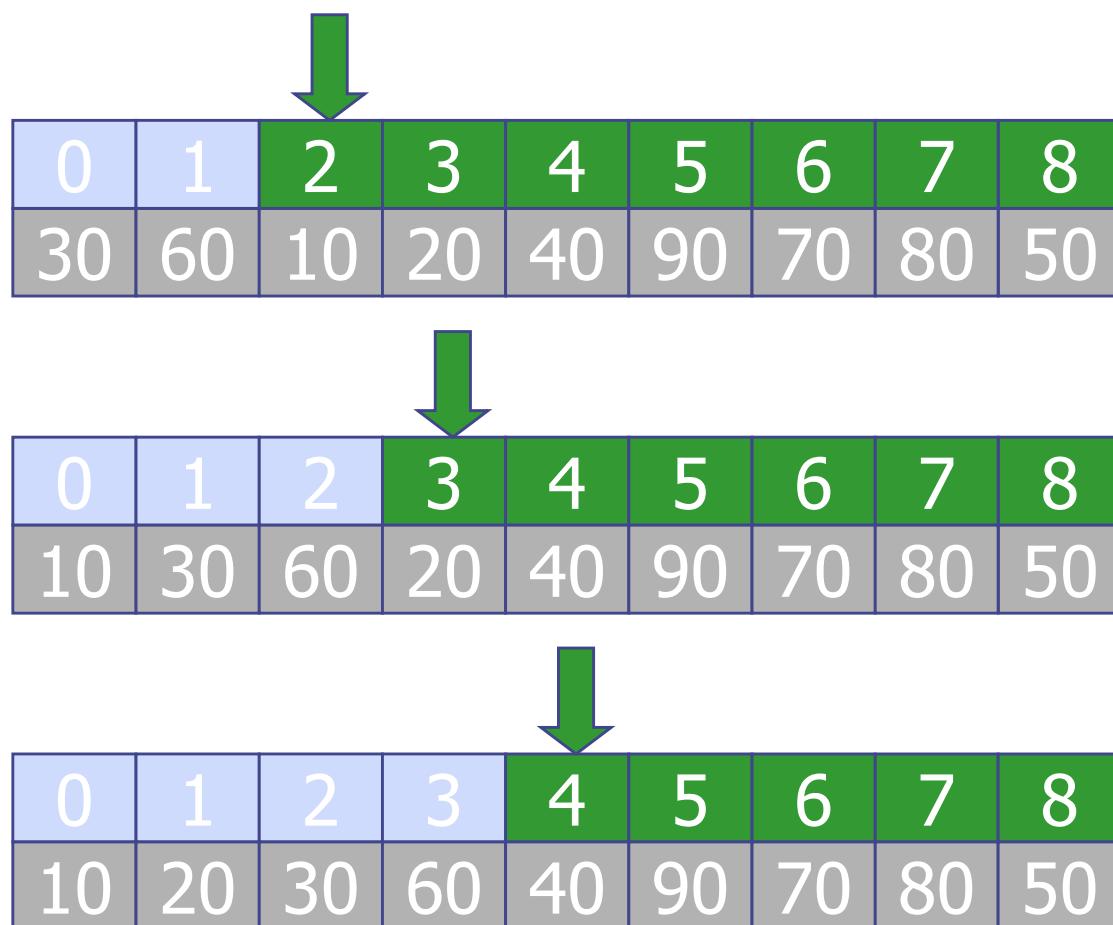
0	1	2	3	4	5	6	7	8
60	30	10	20	40	90	70	80	50

- Inserir o primeiro elemento do segmento desordenado no lugar correto

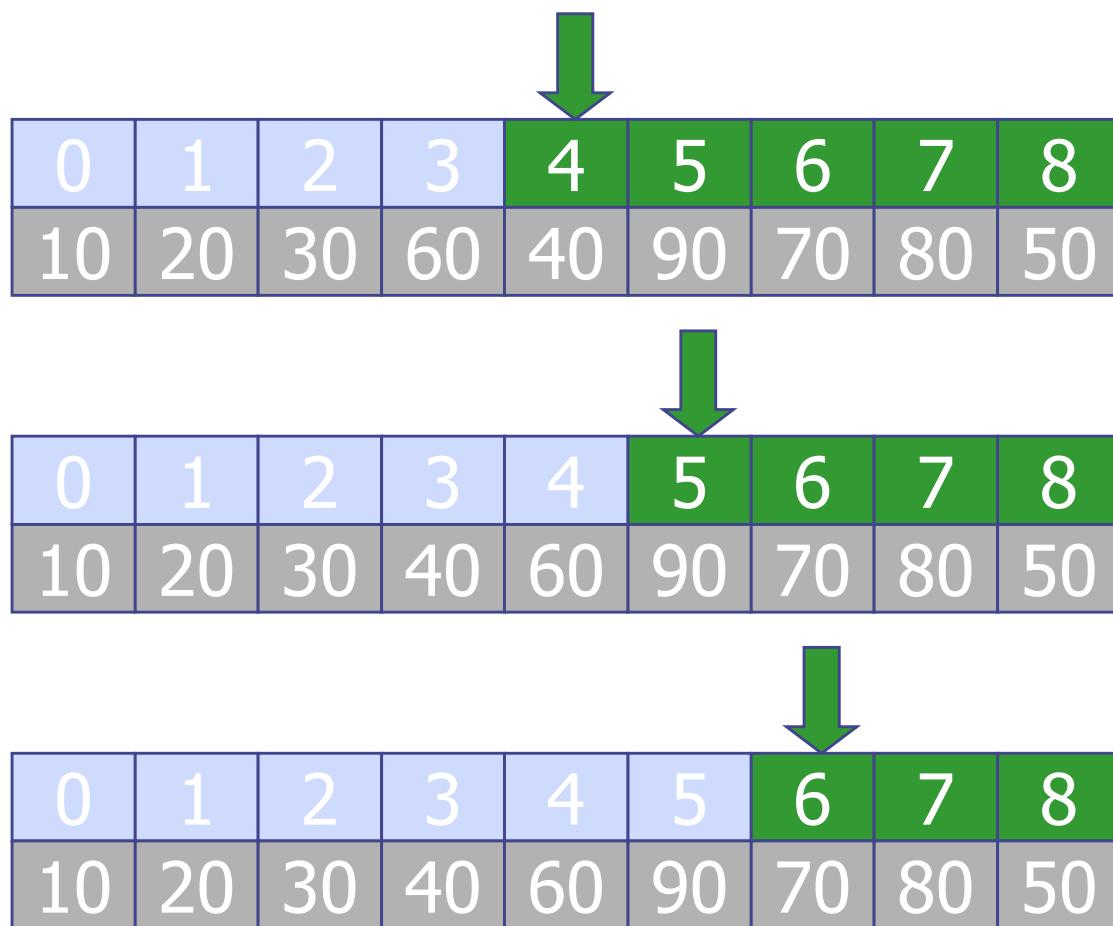


0	1	2	3	4	5	6	7	8
30	60	10	20	40	90	70	80	50

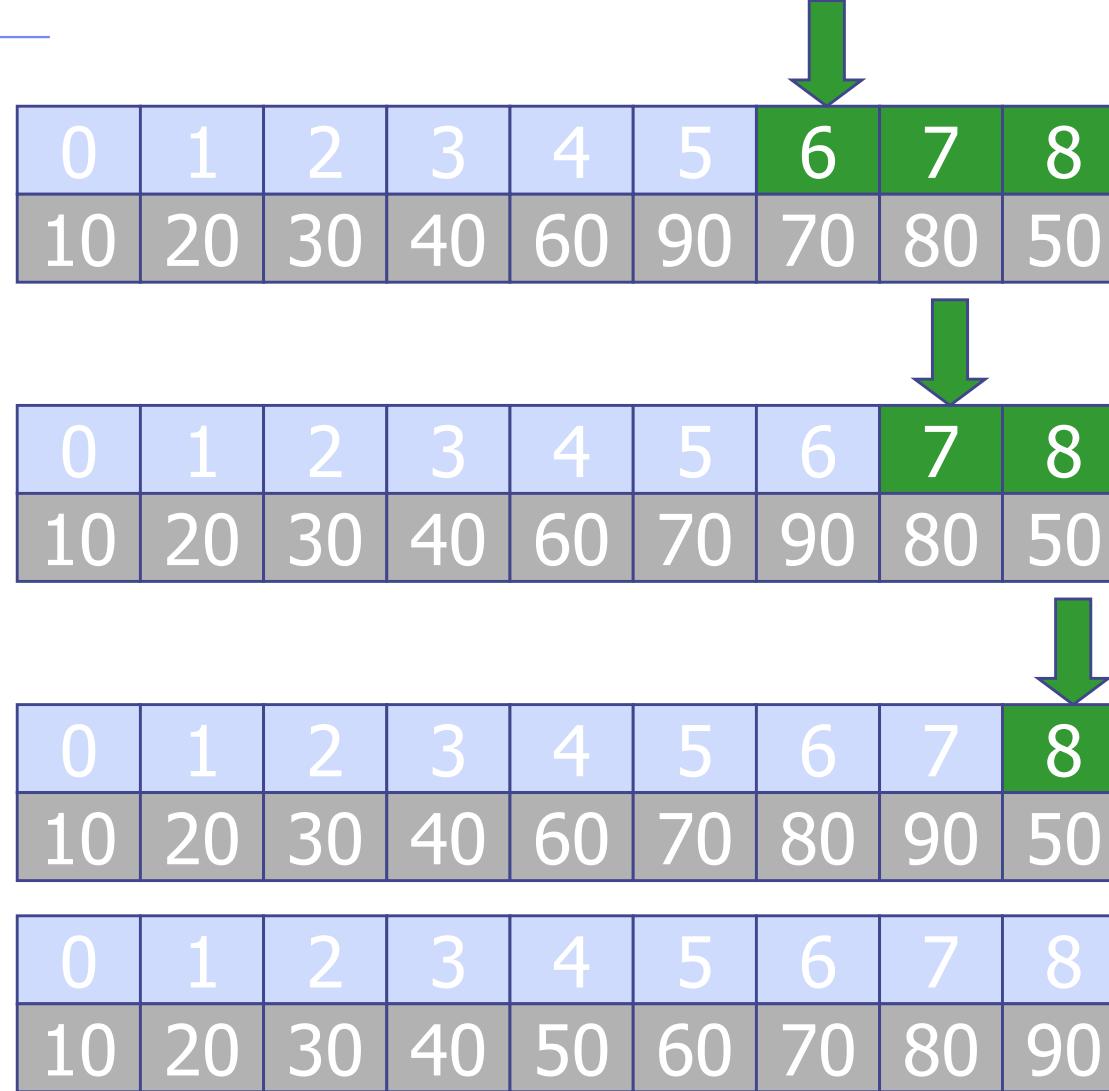
InsertionSort



InsertionSort



InsertionSort



InsertionSort em C

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;

    for (i=1 ; i < n ; i++) {
        tmp = vet[i];
        //coloca o item no lugar apropriado na sequência destino
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--)
            vet[j+1] = vet[j];
        vet[j+1] = tmp;
    }
}
```

InsertionSort em C

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;

    for (i=1 ; i < n ; i++) {
        tmp = vet[i];
        //coloca o item no lugar apropriado na sequência destino
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--)
            vet[j+1] = vet[j];
        vet[j+1] = tmp;
    }
}
```

Neste processo de alternar comparações e movimentos, duas condições podem causar a terminação do processo

InsertionSort em C

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;

    for (i=1 ; i < n ; i++) {
        tmp = vet[i];
        //coloca o item no lugar apropriado na sequência destino
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--)
            vet[j+1] = vet[j];
        vet[j+1] = tmp;
    }
}
```

1) Final da sequência destino é atingido à esquerda

InsertionSort em C

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;

    for (i=1 ; i < n ; i++) {
        tmp = vet[i];
        //coloca o item no lugar apropriado na sequência destino
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--)
            vet[j+1] = vet[j];
        vet[j+1] = tmp;
    }
}
```

- 2) Chave menor que a chave em consideração é encontrada

Detalhando o algoritmo

Exemplo

$i = 1$



0	1	2	3	4	5	6	7	8
60	30	10	20	40	90	70	80	50

- $\text{tmp} = \text{vet}[1] = 30$
- $j = i - 1 = 0$
- $\text{vet}[0] = 60 \rightarrow 60 > 30$

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) { ←
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--) ←
            vet[j+1] = vet[j];
        vet[j+1] = tmp;
    }
}
```

Detalhando o algoritmo

◆ Exemplo

$i = 1$



0	1	2	3	4	5	6	7	8
60	60	10	20	40	90	70	80	50

- $j = 0$ e $tmp = 30$
- $vet[j+1] = vet[j] \rightarrow vet[1] = 60$
- $j-- \rightarrow j = -1 \rightarrow$ final da sequência destino é atingido à esquerda

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) {
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--)
            vet[j+1] = vet[j];
        vet[j+1] = tmp;
    }
}
```

Detalhando o algoritmo

◆ Exemplo

$i = 1$



0	1	2	3	4	5	6	7	8
30	60	10	20	40	90	70	80	50

- $j = -1$ e $tmp = 30$
- $\text{vet}[j+1] = tmp \rightarrow \text{vet}[0] = 30$

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) {
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--)
            vet[j+1] = vet[j];
        vet[j+1] = tmp; ←
    }
}
```

Detalhando o algoritmo

◆ Exemplo

$i = 2$

0	1	2	3	4	5	6	7	8
30	60	10	20	40	90	70	80	50

- $\text{tmp} = \text{vet}[2] = 10$
- $j = i - 1 = 1$
- $\text{vet}[1] = 60 \rightarrow 60 > 10$

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) { ←
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--) ←
            vet[j+1] = vet[j];
        vet[j+1] = tmp;
    }
}
```

Detalhando o algoritmo

◆ Exemplo

$i = 2$

0	1	2	3	4	5	6	7	8
30	60	60	20	40	90	70	80	50

- $j = 1$ e $tmp = 10$
- $\text{vet}[j+1] = \text{vet}[j] \rightarrow \text{vet}[2] = 60$
- $j-- \rightarrow j = 0 \ \&\& \ \text{vet}[0] > \text{tmp}$

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) {
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--)
            vet[j+1] = vet[j];
        vet[j+1] = tmp;
    }
}
```

Detalhando o algoritmo

◆ Exemplo

$i = 2$

0	1	2	3	4	5	6	7	8
30	30	60	20	40	90	70	80	50

- $j = 0$ e $tmp = 10$
- $\text{vet}[j+1] = \text{vet}[j] \rightarrow \text{vet}[1] = 30$
- $j-- \rightarrow j = -1 \rightarrow$ final da sequência destino é atingido à esquerda

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) {
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--) {
            vet[j+1] = vet[j];
        }
        vet[j+1] = tmp;
    }
}
```

Detalhando o algoritmo

◆ Exemplo

$i = 2$

0	1	2	3	4	5	6	7	8
10	30	60	20	40	90	70	80	50

- $j = -1$ e $tmp = 10$
- $\text{vet}[j+1] = tmp \rightarrow \text{vet}[0] = 10$

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) {
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--)
            vet[j+1] = vet[j];
        vet[j+1] = tmp; ←
    }
}
```

Detalhando o algoritmo

Exemplo

$i = 3$



0	1	2	3	4	5	6	7	8
10	30	60	20	40	90	70	80	50

- $\text{tmp} = \text{vet}[3] = 20$
- $j = i - 1 = 2$
- $\text{vet}[2] = 60 \rightarrow 60 > 20$

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) { ←
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--) ←
            vet[j+1] = vet[j];
        vet[j+1] = tmp;
    }
}
```

Detalhando o algoritmo

◆ Exemplo

$i = 3$



0	1	2	3	4	5	6	7	8
10	30	60	60	40	90	70	80	50

- $j = 2$ e $\text{tmp} = 20$
- $\text{vet}[j+1] = \text{vet}[j] \rightarrow \text{vet}[3] = 60$
- $j-- \rightarrow j = 1 \ \&\& \ \text{vet}[1] > \text{tmp}$

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) {
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--)
            vet[j+1] = vet[j];
        vet[j+1] = tmp;
    }
}
```

Detalhando o algoritmo

◆ Exemplo

$i = 3$



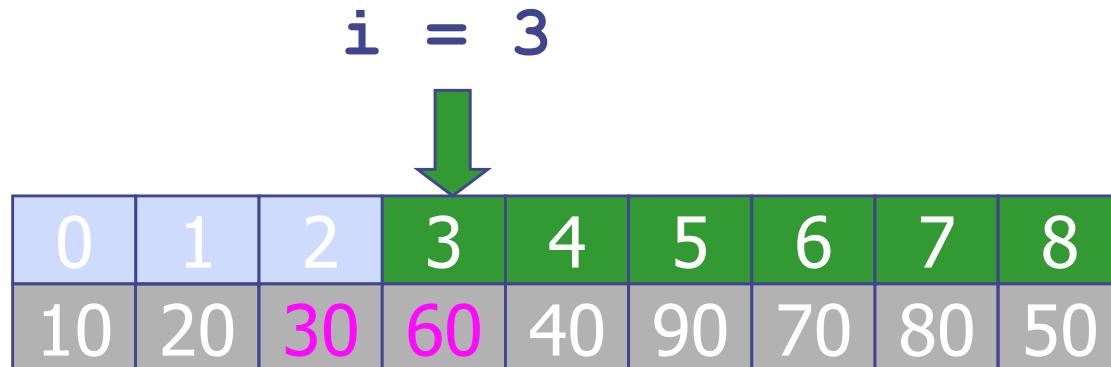
0	1	2	3	4	5	6	7	8
10	30	30	60	40	90	70	80	50

- $j = 1$ e $\text{tmp} = 20$
- $\text{vet}[j+1] = \text{vet}[j] \rightarrow \text{vet}[2] = 30$
- $j-- \rightarrow j = 0$ mas $\text{vet}[0] < \text{tmp}$

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) {
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--)
            vet[j+1] = vet[j];
        vet[j+1] = tmp;
    }
}
```

Detalhando o algoritmo

◆ Exemplo



- $j = 0$ e $\text{tmp} = 20$
- $\text{vet}[j+1] = \text{tmp} \rightarrow \text{vet}[1] = 20$

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) {
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--)
            vet[j+1] = vet[j];
        vet[j+1] = tmp; ←
    }
}
```

Detalhando o algoritmo

◆ Exemplo

$i = 4$



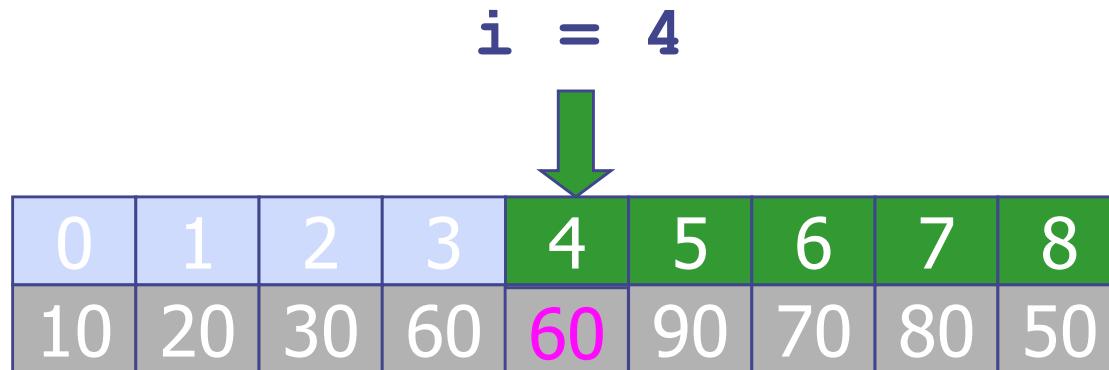
0	1	2	3	4	5	6	7	8
10	20	30	60	40	90	70	80	50

- $\text{tmp} = \text{vet}[4] = 40$
- $j = i - 1 = 3$
- $\text{vet}[3] = 60 \rightarrow 60 > 40$

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) { ←
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--) ←
            vet[j+1] = vet[j];
        vet[j+1] = tmp;
    }
}
```

Detalhando o algoritmo

◆ Exemplo

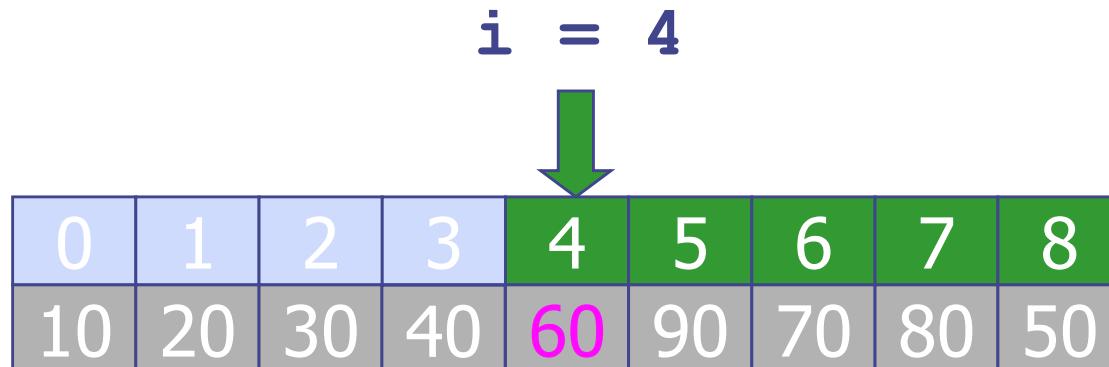


- $j = 3$ e $tmp = 40$
- $vet[j+1] = vet[j] \rightarrow vet[4] = 60$
- $j-- \rightarrow j = 2$ mas $vet[2] < tmp$

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) {
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--)
            vet[j+1] = vet[j];
        vet[j+1] = tmp;
    }
}
```

Detalhando o algoritmo

◆ Exemplo

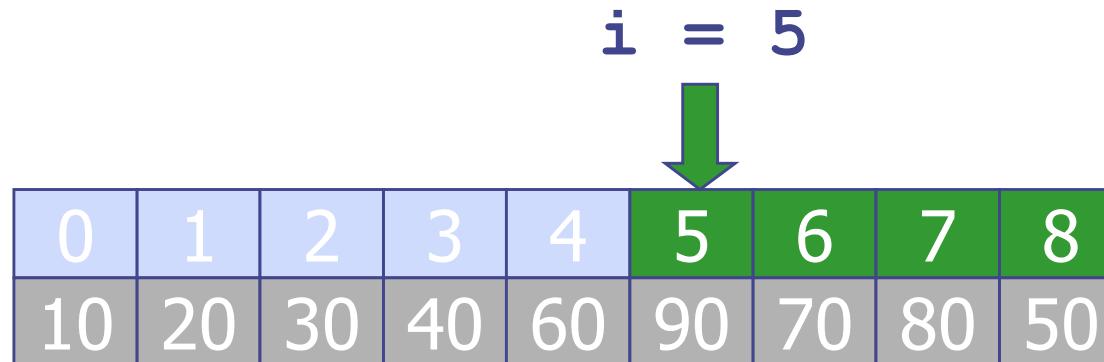


- $j = 2$ e $\text{tmp} = 40$
- $\text{vet}[j+1] = \text{tmp} \rightarrow \text{vet}[3] = 40$

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) {
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--)
            vet[j+1] = vet[j];
        vet[j+1] = tmp; ←
    }
}
```

Detalhando o algoritmo

Exemplo



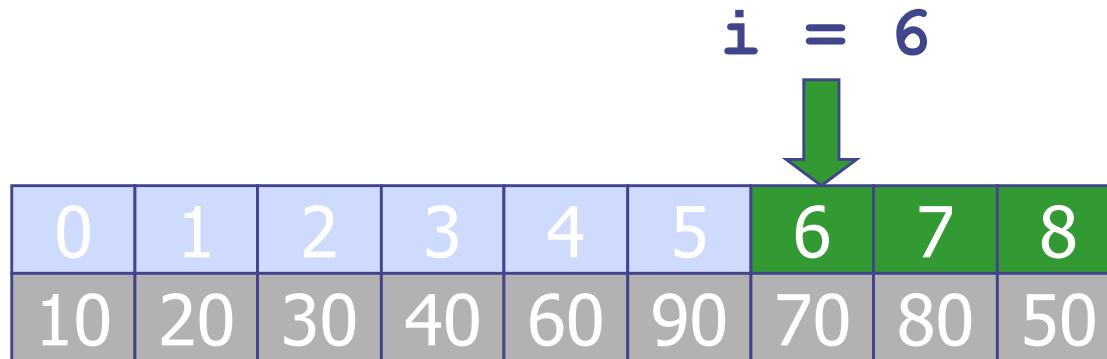
- $\text{tmp} = \text{vet}[5] = 90$
- $j = i - 1 = 4$
- $\text{vet}[4] = 60 \rightarrow 60 \text{ não é maior que } 90$

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) { ←
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--) ←
            vet[j+1] = vet[j];
        vet[j+1] = tmp;
    }
}
```

Detalhando o algoritmo

Exemplo

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) { ←
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--) ←
            vet[j+1] = vet[j];
        vet[j+1] = tmp;
    }
}
```

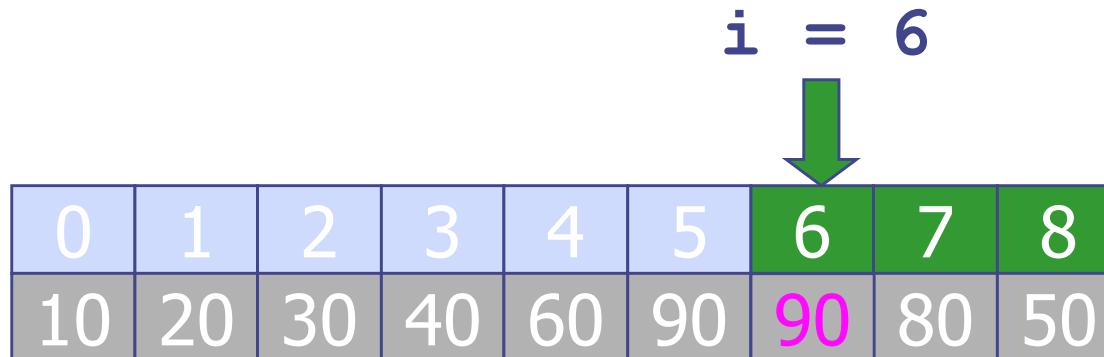


- $\text{tmp} = \text{vet}[6] = 70$
- $j = i - 1 = 5$
- $\text{vet}[5] = 90 \rightarrow 90 > 70$

Detalhando o algoritmo

◆ Exemplo

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) {
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--) {
            vet[j+1] = vet[j];
        }
        vet[j+1] = tmp;
    }
}
```

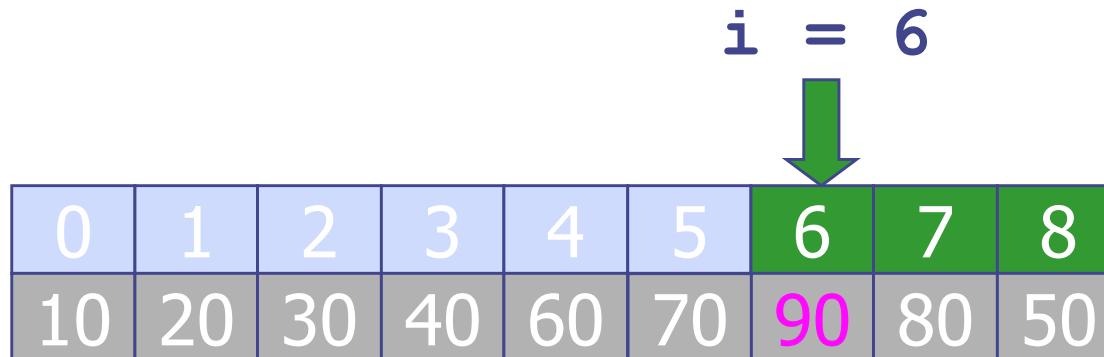


- $j = 5$ e $tmp = 70$
- $vet[j+1] = vet[j] \rightarrow vet[6] = 90$
- $j-- \rightarrow j = 4$ mas $vet[4] < tmp$

Detalhando o algoritmo

◆ Exemplo

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) {
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--)
            vet[j+1] = vet[j];
        vet[j+1] = tmp; ←
    }
}
```

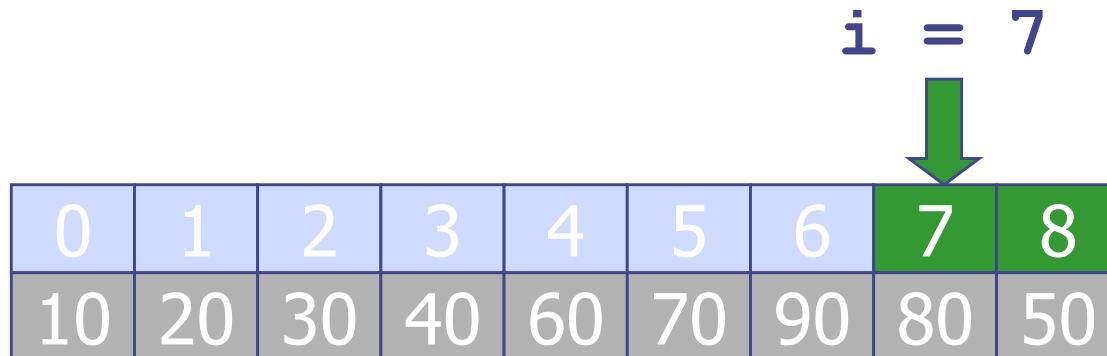


- $j = 4$ e $tmp = 70$
- $vet[j+1] = tmp \rightarrow vet[5] = 70$

Detalhando o algoritmo

◆ Exemplo

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) { ←
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--) ←
            vet[j+1] = vet[j];
        vet[j+1] = tmp;
    }
}
```

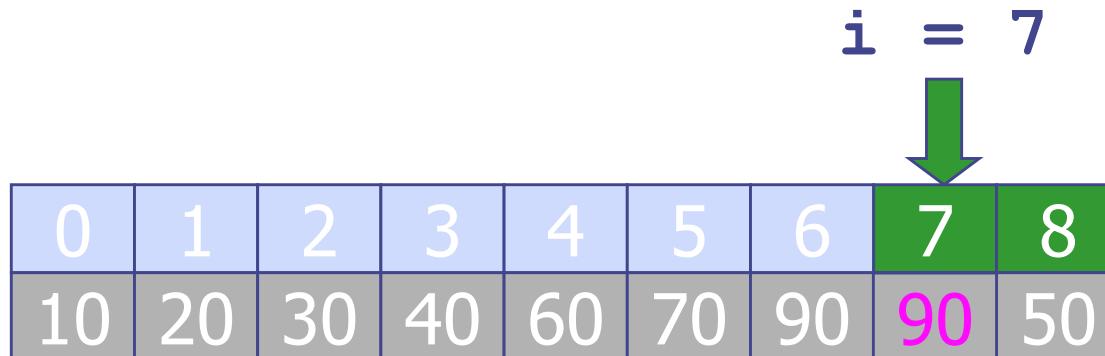


- $\text{tmp} = \text{vet}[7] = 80$
- $j = i - 1 = 6$
- $\text{vet}[6] = 90 \rightarrow 90 > 80$

Detalhando o algoritmo

◆ Exemplo

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) {
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--)
            vet[j+1] = vet[j];
        vet[j+1] = tmp;
    }
}
```

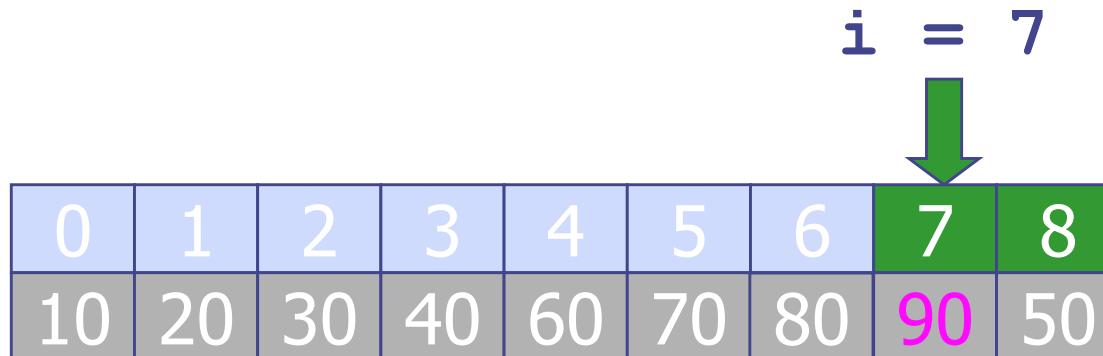


- $j = 6$ e $tmp = 80$
- $vet[j+1] = vet[j] \rightarrow vet[7] = 90$
- $j-- \rightarrow j = 5$ mas $vet[5] < tmp$

Detalhando o algoritmo

◆ Exemplo

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) {
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--)
            vet[j+1] = vet[j];
        vet[j+1] = tmp; ←
    }
}
```



- $j = 5$ e $tmp = 80$
- $vet[j+1] = tmp \rightarrow vet[6] = 80$

Detalhando o algoritmo

◆ Exemplo

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) { ←
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--) ←
            vet[j+1] = vet[j];
        vet[j+1] = tmp;
    }
}
```

$i = 8$



0	1	2	3	4	5	6	7	8
10	20	30	40	60	70	80	90	50

- $\text{tmp} = \text{vet}[8] = 50$
- $j = i - 1 = 7$
- $\text{vet}[7] = 90 \rightarrow 90 > 50$

Detalhando o algoritmo

◆ Exemplo

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) {
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--)
            vet[j+1] = vet[j];
        vet[j+1] = tmp;
    }
}
```

0	1	2	3	4	5	6	7	8
10	20	30	40	60	70	80	90	90

i = 8

- $j = 7$ e $tmp = 50$
- $vet[j+1] = vet[j] \rightarrow vet[8] = 90$
- $j-- \rightarrow j = 6 \ \&\& \ vet[6] > tmp$

Detalhando o algoritmo

◆ Exemplo

0	1	2	3	4	5	6	7	8
10	20	30	40	60	70	80	80	90

i = 8



- j = 6 e tmp = 50
- vet[j+1] = vet[j] → vet[7] = 80
- j-- → j = 5 && vet[5] > tmp

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) {
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--)
            vet[j+1] = vet[j];
        vet[j+1] = tmp;
    }
}
```

Detalhando o algoritmo

◆ Exemplo

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) {
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--) {
            vet[j+1] = vet[j];
        }
        vet[j+1] = tmp;
    }
}
```

$i = 8$

0	1	2	3	4	5	6	7	8
10	20	30	40	60	70	70	80	90

- $j = 5$ e $tmp = 50$
- $vet[j+1] = vet[j] \rightarrow vet[6] = 70$
- $j-- \rightarrow j = 4 \&\& vet[4] > tmp$

Detalhando o algoritmo

◆ Exemplo

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) {
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--) {
            vet[j+1] = vet[j];
        }
        vet[j+1] = tmp;
    }
}
```

$i = 8$



0	1	2	3	4	5	6	7	8
10	20	30	40	60	60	70	80	90

- $j = 4$ e $tmp = 50$
- $vet[j+1] = vet[j] \rightarrow vet[5] = 60$
- $j-- \rightarrow j = 3$ mas $vet[3] < tmp$

Detalhando o algoritmo

◆ Exemplo

0	1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80	90

$i = 8$



- $j = 3$ e $tmp = 50$
- $\text{vet}[j+1] = tmp \rightarrow \text{vet}[4] = 50$

```
void insertionSort (int vet[], int n)
{
    int i,j, tmp;
    for (i=1 ; i < n ; i++) {
        tmp = vet[i];
        for (j = i-1 ; j >= 0 && vet[j] > tmp ; j--)
            vet[j+1] = vet[j];
        vet[j+1] = tmp; ←
    }
}
```

InsertionSort -

- ◆ Link – Animação:
 - <https://visualgo.net/en/sorting>

InsertionSort - Comparações

- ◆ No melhor caso, a comparação do laço interno é executada uma só vez (`vet[j] > atual`) quando o elemento atual é maior que o último
 - $(n-1) * 1 = O(n)$
- ◆ No pior caso, são efetuadas o máximo de comparações no laço interno (enquanto $j \geq 0$), isto é, primeiro 1, depois 2,, até $n-1$
 - $(n-1) + (n-2) + .. + 1 = n(n-1) = O(n^2)$
- ◆ No caso médio, o número de comparações também é $O(n^2)$

InsertionSort - Movimentações

- ◆ Quantas **operações de movimentação de posição** são necessárias para ordenar um conjunto de n elementos?
- ◆ A movimentação de `vetor[j]` para `vet[j+1]` é executada tantas vezes quanto a comparação
- ◆ No melhor caso, ao ser feita uma comparação, o corpo do laço não é executado nenhuma vez
- ◆ No pior caso (e no caso médio) o número total de atribuições é:
 - $O(n^2)$

ShellSort

- ◆ Inventado por Donald Shell em 1959
 - Foi o primeiro algoritmo de ordenação a quebrar a barreira do $O(n^2)$
- ◆ Basicamente é um aprimoramento do algoritmo InsertionSort
- ◆ Qual é o problema do InsertionSort?
 - O método da inserção troca itens adjacentes quando está procurando o ponto de inserção na sequência destino
 - Se o menor item estiver na posição mais à direita no vetor, então o número de comparações e movimentações é igual a $n - 1$ para encontrar o seu ponto de inserção

ShellSort

- ◆ Em vez de comparar sempre elementos adjacentes, compara elementos distantes
- ◆ A ideia é poder mover os elementos de forma mais rápida até a posição correta
- ◆ As comparações e movimentos tem um incremento que inicia alto e vai diminuindo até 1
- ◆ Os itens separados de h posições são rearranjados
- ◆ Todo h -ésimo item leva a uma sequência ordenada
- ◆ Tal sequência é dita estar h -ordenada

ShellSort

◆ Vantagens

- Mais eficiente que Bubble, Selection e InsertionSort
- Algoritmo mais simples que MergeSort, QuickSort
- Opção para arquivos de tamanho moderado

◆ Desvantagens

- Algoritmo mais complexo que Bubble, Selection, etc
- Não tão eficiente quanto MergeSort, QuickSort
- Tempo de execução sensível à ordem inicial dos dados
- **Não é estável**

Método de ShellSort

♦ Exemplo 1:

- Vetor: [25,57,48,37,12,92,86,33] -
- Seqüência de Distâncias: - {5,3,1} **Segunda iteração**

■ Primeira iteração

(x[0],x[5])

(x[0],x[3],x[6])

(x[1],x[6])

(x[1],x[4],x[7])

(x[2],x[7])

(x[2],x[5])

(x[3])

Terceira iteração

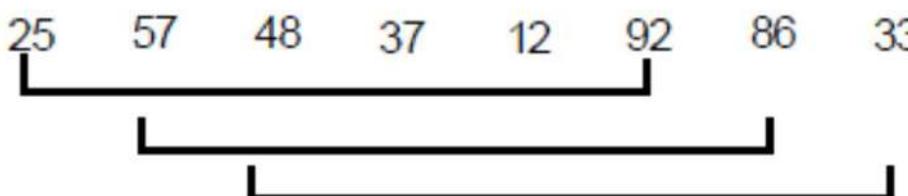
(x[4])

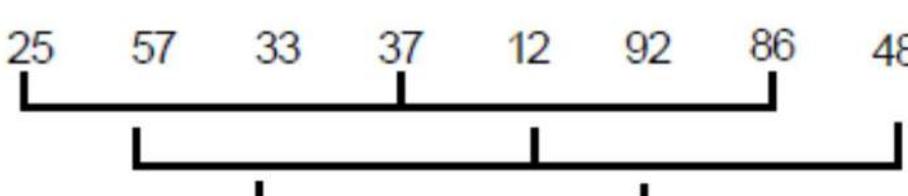
x[0], x[1], x[2],
x[3],x[4],x[5], x[6],x[7]

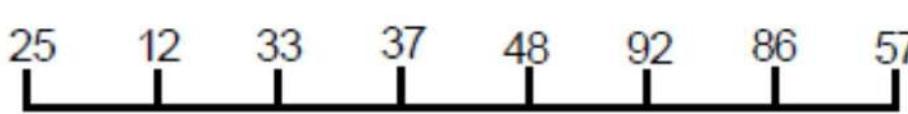
Método de ShellSort

- Exemplo 1:

Vetor original 0 1 2 3 4 5 6 7
 25 57 48 37 12 92 86 33

Passagem 1 $h = 5$


Passagem 2 $h = 3$


Passagem 3 $h = 1$


Vetor ordenado 12 25 33 37 48 57 86 92

Método de ShellSort

♦ Exemplo 2:

	1	2	3	4	5	6
Chaves iniciais:	O	R	D	E	N	A
$h = 4$	N	A	D	E	O	R
$h = 2$	D	A	N	E	O	R
$h = 1$	A	D	E	N	O	R

Projeto de Algoritmos: com Implementações em Pascal e C – 3ª edição revista e ampliada. Nivio Ziviani

Método de ShellSort

- ◆ Exemplo:

	1	2	3	4	5	6
Chaves iniciais:	<u>O</u>	R	D	E	<u>N</u>	A
→ h = 4	N	A	D	E	O	R
h = 2	D	A	N	E	O	R
h = 1	A	D	E	N	O	R

Projeto de Algoritmos: com Implementações em Pascal e C – 3ª edição revista e ampliada. Nivio Ziviani

Método de ShellSort

- ◆ Exemplo:

	1	2	3	4	5	6
Chaves iniciais:	O	R	D	E	N	A
→ h = 4	N	A	D	E	O	R
h = 2	D	A	N	E	O	R
h = 1	A	D	E	N	O	R

Projeto de Algoritmos: com Implementações em Pascal e C – 3ª edição revista e ampliada. Nivio Ziviani

Método de ShellSort

- ◆ Exemplo:

	1	2	3	4	5	6
Chaves iniciais:	O	R	D	E	N	A
h = 4	<u>N</u>	A	<u>D</u>	E	<u>O</u>	R
→ h = 2	D	A	N	E	O	R
h = 1	A	D	E	N	O	R

Projeto de Algoritmos: com Implementações em Pascal e C – 3ª edição revista e ampliada. Nivio Ziviani

Método de ShellSort

- ◆ Exemplo:

	1	2	3	4	5	6
Chaves iniciais:	O	R	D	E	N	A
$h = 4$	N	<u>A</u>	D	<u>E</u>	O	<u>R</u>
→ $h = 2$	D	A	N	E	O	R
$h = 1$	A	D	E	N	O	R

Projeto de Algoritmos: com Implementações em Pascal e C – 3ª edição revista e ampliada. Nivio Ziviani

Método de ShellSort

- ◆ Exemplo:

	1	2	3	4	5	6
Chaves iniciais:	O	R	D	E	N	A
$h = 4$	N	A	D	E	O	R
$h = 2$	D	A	N	E	O	R
→ $h = 1$	A	D	E	N	O	R

Projeto de Algoritmos: com Implementações em Pascal e C – 3ª edição revista e ampliada. Nivio Ziviani

- Quando $h = 1$, ShellSort é igual ao algoritmo de inserção
 - ◆ Porém, nenhum item precisa se mover para posições muito distantes

ShellSort

- ◆ Várias sequências para h têm sido experimentadas
- ◆ Knuth (1973, p. 95) mostrou experimentalmente que a escolha do incremento para h , mostrada a seguir, é difícil de ser batida por mais de 20% em eficiência no tempo de execução:
 - Para $s=1$: $h(s) = 1$
 - Para $s > 1$: $h(s) = 3h(s - 1) + 1$
- ◆ A sequência para h corresponde a 1, 4, 13, 40, 121, 364, 1.093, 3.280, ...

Método de ShellSort

```
void Shellsort(int vetor [], int n) {  
    int i, j, h = 1;  
    int aux;  
    do { h = h * 3 + 1; } while (h < n); // determina o intervalo  
    do { h /= 3;  
        for(i = h ; i < n ; i++) { // insere o elemento na posição  
            aux = vetor[i];           // correta no vetor  
            j = i;  
            while (vetor[j - h] > aux) {  
                vetor[j] = vetor[j - h]; j -= h;  
                if (j < h) break;  
            }  
            vetor[j] = aux;  
        }  
    } while (h != 1);  
}
```

Método de ShellSort

Exemplo de execução do algoritmo

$n = 8$, h será definido como 13 inicialmente

$$h = h/3 = 13/3 = 4$$

0	1	2	3	4	5	6	7	
25	57	48	37	12	92	86	33	aux = 12



Método de ShellSort

Exemplo de execução do algoritmo

$n = 8$, h será definido como 13 inicialmente

$$h = h/3 = 13/3 = 4$$

0	1	2	3	4	5	6	7	
12	57	48	37	25	92	86	33	aux = 92



Método de ShellSort

Exemplo de execução do algoritmo

$n = 8$, h será definido como 13 inicialmente

$$h = h/3 = 13/3 = 4$$

0	1	2	3	4	5	6	7	
12	57	48	37	25	92	86	33	aux = 86



Método de ShellSort

Exemplo de execução do algoritmo

$n = 8$, h será definido como 13 inicialmente

$$h = h/3 = 13/3 = 4$$

0	1	2	3	4	5	6	7	
12	57	48	37	25	92	86	33	aux = 33



Método de ShellSort

Exemplo de execução do algoritmo

$n = 8$, h será definido como 13 inicialmente

$$h = h/3 = 4/3 = 1$$

0	1	2	3	4	5	6	7	
12	57	48	33	25	92	86	37	aux = 57



Método de ShellSort

Exemplo de execução do algoritmo

$n = 8$, h será definido como 13 inicialmente

$$h = h/3 = 4/3 = 1$$

0	1	2	3	4	5	6	7	
12	57	48	33	25	92	86	37	aux = 48
								

0	1	2	3	4	5	6	7	
12	48	57	33	25	92	86	37	aux = 48
								

Método de ShellSort

Exemplo de execução do algoritmo

$n = 8$, h será definido como 13 inicialmente

$$h = h/3 = 4/3 = 1$$

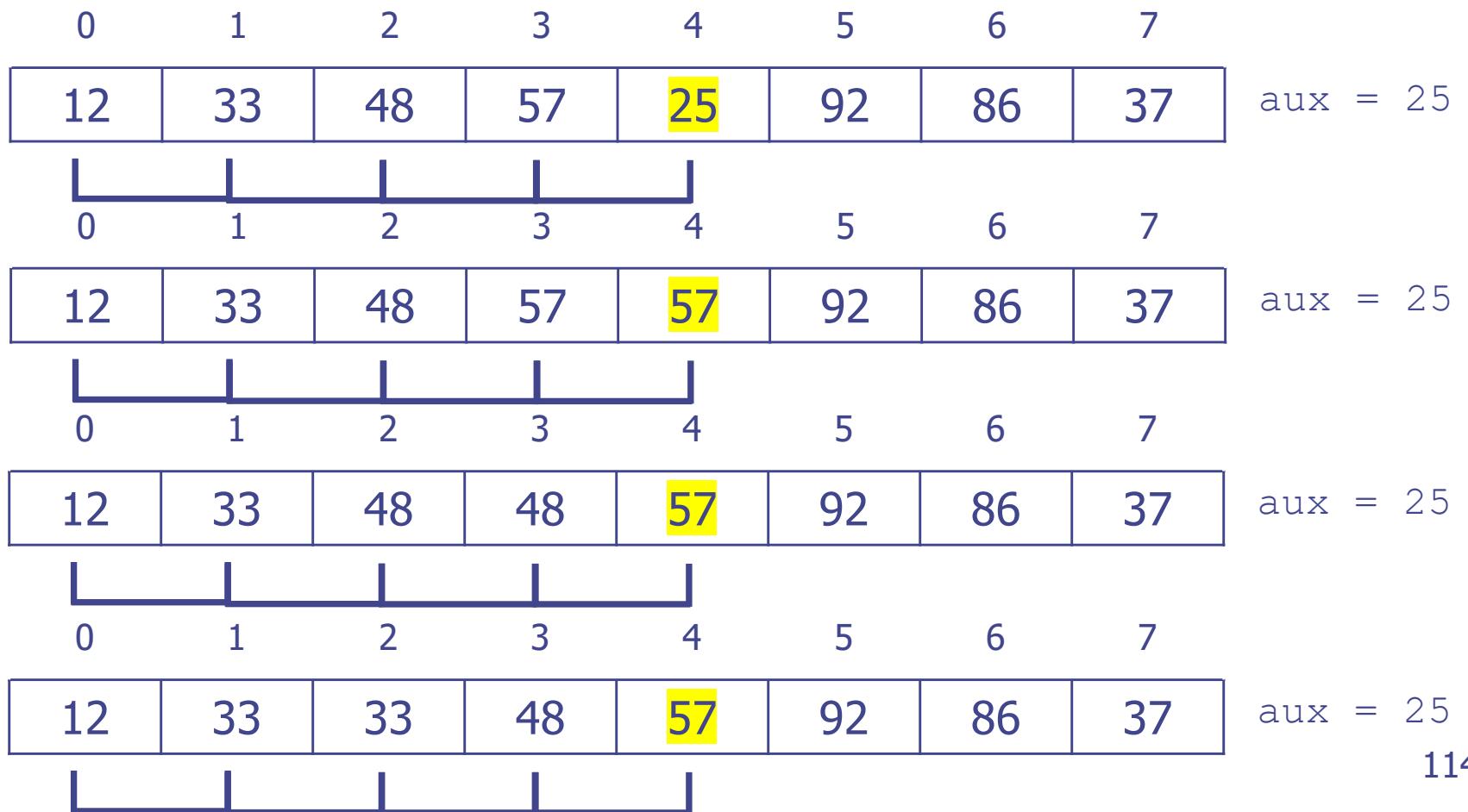
0	1	2	3	4	5	6	7	
12	48	57	33	25	92	86	37	aux = 33
0	1	2	3	4	5	6	7	
12	48	57	57	25	92	86	37	aux = 33
0	1	2	3	4	5	6	7	
12	48	48	57	25	92	86	37	aux = 33
0	1	2	3	4	5	6	7	
12	33	48	57	25	92	86	37	aux = 33

Método de ShellSort

Exemplo de execução do algoritmo

$n = 8$, h será definido como 13 inicialmente

$$h = h/3 = 4/3 = 1$$



Método de ShellSort

Exemplo de execução do algoritmo

$n = 8$, h será definido como 13 inicialmente

$$h = h/3 = 4/3 = 1$$

0	1	2	3	4	5	6	7	
12	25	33	48	57	92	86	37	aux = 25

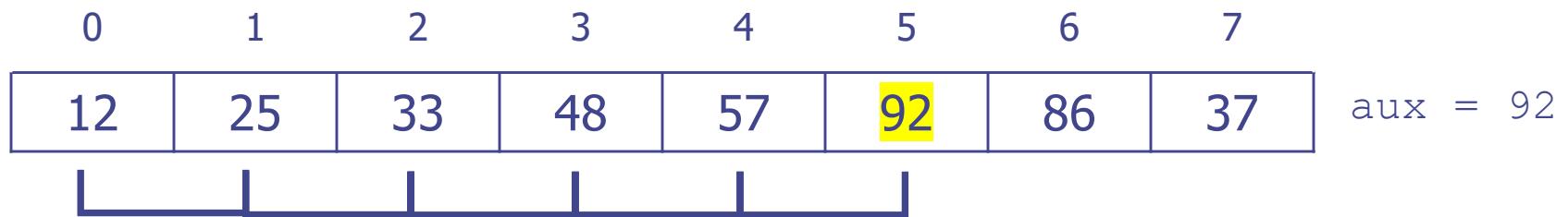


Método de ShellSort

Exemplo de execução do algoritmo

$n = 8$, h será definido como 13 inicialmente

$$h = h/3 = 4/3 = 1$$

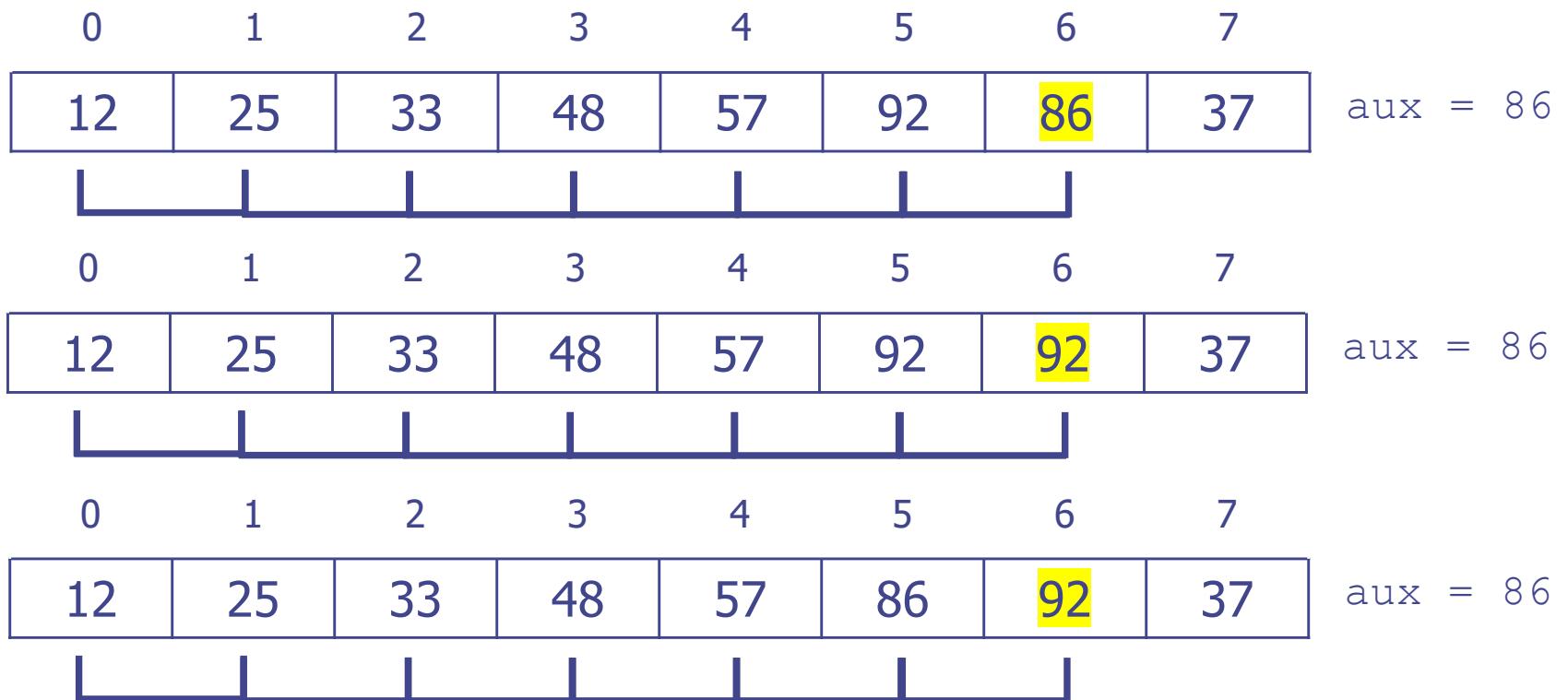


Método de ShellSort

Exemplo de execução do algoritmo

$n = 8$, h será definido como 13 inicialmente

$$h = h/3 = 4/3 = 1$$

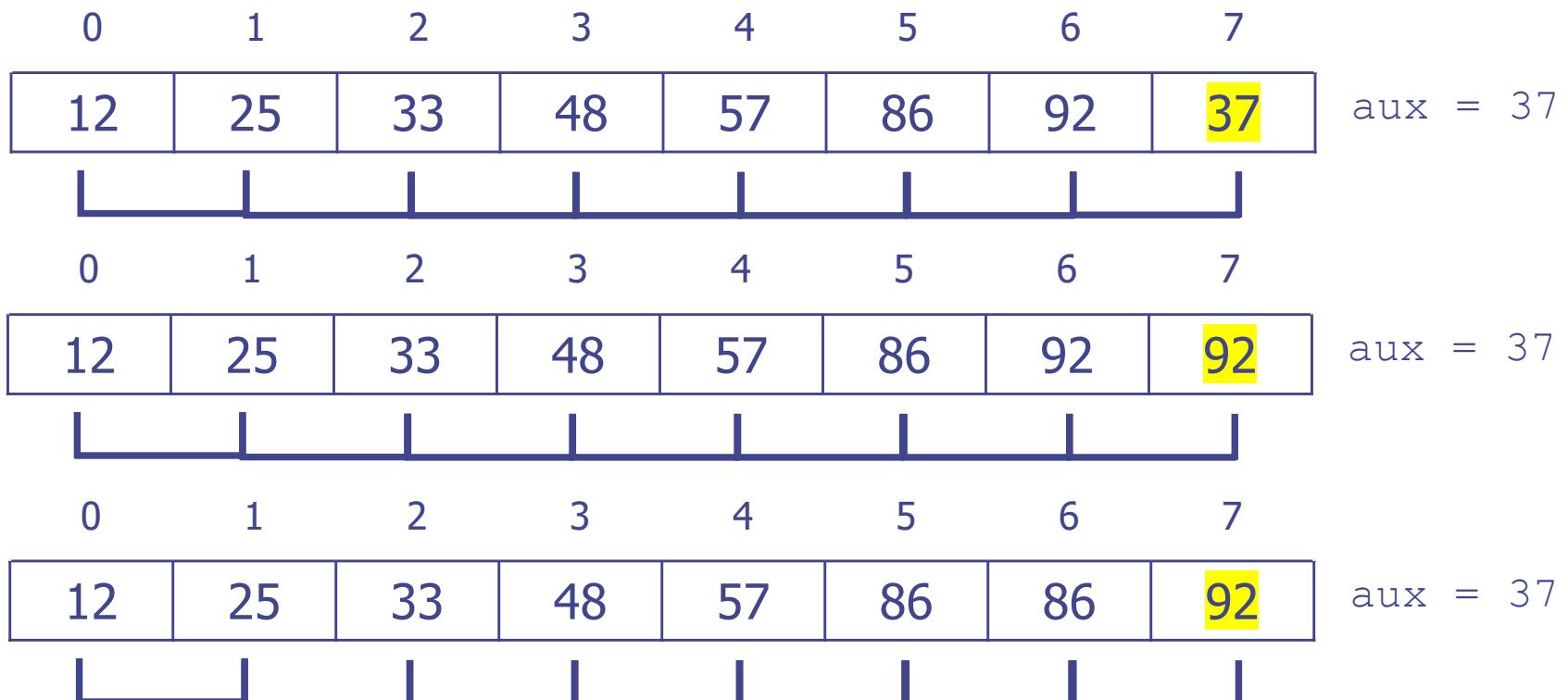


Método de ShellSort

Exemplo de execução do algoritmo

$n = 8$, h será definido como 13 inicialmente

$$h = h/3 = 4/3 = 1$$



Método de ShellSort

Exemplo de execução do algoritmo

$n = 8$, h será definido como 13 inicialmente

$$h = h/3 = 4/3 = 1$$

0	1	2	3	4	5	6	7	
12	25	33	48	57	57	86	92	aux = 37
								
0	1	2	3	4	5	6	7	
12	25	33	48	48	57	86	92	aux = 37
								
0	1	2	3	4	5	6	7	
12	25	33	37	48	57	86	92	aux = 37
								

Método de ShellSort

Exemplo de execução do algoritmo

$n = 8$, h será definido como 13 inicialmente

$$h = h/3 = 4/3 = 1$$

0	1	2	3	4	5	6	7	
12	25	33	48	57	57	86	92	aux = 37
0	1	2	3	4	5	6	7	
12	25	33	48	48	57	86	92	aux = 37
0	1	2	3	4	5	6	7	
12	25	33	37	48	57	86	92	aux = 37
0	1	2	3	4	5	6	7	
12	25	33	37	48	57	86	92	aux = 37
Vetor ordenado								
12	25	33	37	48	57	86	92	

ShellSort - Análise

- ◆ A última varredura do ShellSort é sempre com incremento 1
 - Isto é o mesmo que o InsertionSort
- ◆ Ou seja, o ShellSort faz umas varreduras anteriores com incrementos maiores para não ter que mover elementos tantas vezes quando o incremento for 1
- ◆ Não existe prova matemática de que o ShellSort é melhor que o InsertionSort, mas por dados empíricos acredita-se que ele tenha complexidade entre $n^{1.25}$ a $1.6n^{1.25}$

Comparação de Métodos

- ◆ Ordenação de 100.000 elementos tipo double, gerados aleatoriamente em máquina Pentium Core 2, 2.8 GHz
- ◆ Primeira execução
 - BubbleSort: 58.86 segundos
 - SelectionSort: 17.89 segundos
 - InsertionSort: 14.16 segundos
 - ShellSort: 0.047 segundos
- ◆ Segunda execução
 - BubbleSort: 60.44 segundos
 - SelectionSort: 18.00 segundos
 - InsertionSort: 14.46 segundos
 - ShellSort: 0.047 segundos

Comparando os dois métodos

◆ InsertionSort vs ShellSort

◆ Link:

- <https://www.toptal.com/developers/sorting-algorithms>

Exercícios

- ◆ Dado o vetor $X = [5, 8, 2, 1, 8]$, ordená-lo em ordem crescente, detalhadamente, pelo método da inserção.

- ◆ Dado o vetor $X = [5, 4, 2, 6, 8, 3, 7, 1]$, ordená-lo em ordem crescente, detalhadamente, pelo método shellsort.

Referências

- ◆ ZIVIANI, N. Projeto de Algoritmos. 2a edição, Editora Thomson.
- ◆ CORMEN, THOMAS H. et. al. Algoritmos: Teoria e Prática. Editora Campus, 2002.
- ◆ Prof. André Backes. Slides sobre Algoritmos de Ordenação. Disponíveis em
<https://programacaodescomplicada.wordpress.com/complementar/>

Para Praticar

◆ Exercício de proficiência do ShellSort

- <https://opendsa-server.cs.vt.edu/OpenDSA/AV/Sorting/shellsortPRO.html>

Prática 1 Ordenação

Profa. Maria Camila Nardini Barioni

camila.barioni@ufu.br

Bloco B - sala 1B137

2º semestre de 2023

Informações gerais

- Esta prática tem por objetivo reforçar o conhecimento do(a) aluno(a) com relação aos algoritmos de ordenação simples estudados em sala

- Data da entrega: 26/01/2024
- Grupo de até 3 (dois) alunos(as). Incluir nome e matrícula como comentário no código
- Linguagem de programação a ser usada: C
- O que deve ser entregue: Link para o repositório do código (Replit.com) e documento .pdf com as tabelas solicitadas na descrição da atividade
- Essa prática vale 01 ponto

Exercícios - Prática

1. Dados os métodos de ordenação estudados para organizar vetores de números inteiros, o objetivo do exercício é construir uma tabela de referência contendo o tempo obtido para cada método de ordenação, considerando entrada de dados aleatória (usar geração aleatória de números inteiros) e vetor de tamanho 10.000.

Exemplo

Método	Tempo
BubbleSort	0.0020
InsertionSort	0.0018
SelectionSort	0.0026
ShellSort	0.0001

Exercícios - Prática

- ◆ Modifique o código abaixo:

```
#define N 8
int main (void) {
    int i;
    int v[N]= {25, 48, 37, 12, 57, 86, 33, 92};
    bubbleSort(v,8);
    printf("vetor ordenado:");
    for(i=0, i<N;i++)
        printf("%d", v[i]);
    printf("\n");
    return 0;
}
```

- Inserir código para medir o tempo de execução
- Considerar alocação dinâmica
- Atribuir valores aleatórios ao vetor considerando tamanho 10.000

Exercícios - Prática

- ◆ Exemplo de como calcular o tempo de execução de uma função

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <time.h>
4
5 int main(){
6     time_t ini;
7     time_t fim;
8     double tempo_total;
9     int vetor[TAM];
10
11     ini=clock();
12     QuickSort(vetor,0,TAM-1);
13     fim=clock();
14
15     tempo_total=(double)(fim-ini)/CLOCKS_PER_SEC;
16
17     printf("\nTempo total =%f\n",tempo_total);
18 }
19
```

Exercícios - Prática

- ◆ Exemplo de alocação dinâmica para array com 10 posições em C

```
int v[10];  
  
int *v = malloc(10*sizeof(int));
```

- ◆ Exemplo de geração de números aleatórios entre 0 e 499

```
int aux = rand()%500;
```

Bacharelado em Ciência da Computação
GBC034 Algoritmos e Estruturas de Dados 2

Ordenação

Profa. Maria Camila Nardini Barioni

camila.barioni@ufu.br

Bloco B - sala 1B137

2º semestre de 2023

Avisos

- 
- ◆ A aula de sexta-feira, 02/02, será no laboratório 4

Roteiro

- ◆ Métodos de Ordenação Eficientes
 - ◆ Método da Intercalação (MergeSort)
 - ◆ Método da troca e partição (QuickSort)

Dividir para Conquistar

- ◆ Alguns algoritmos de ordenação usam a técnica “dividir para conquistar” (*divide and conquer*)
- ◆ Ideia básica: é mais fácil ordenar listas pequenas do que grandes
- ◆ Etapas
 - Dividir a lista em duas sub-listas
 - Resolver o problema recursivamente em cada uma
 - Combinar as sub-listas ordenadas para formar a lista ordenada
- ◆ Exemplos: MergeSort e QuickSort

Dividir para Conquistar

algoritmo ordena (lista)

 entrada: lista não ordenada

 saída: lista ordenada

 se a lista tem tamanho maior que 1 então

 dividir a lista em sub-listas inferior e superior

 ordena (lista-inferior)

 ordena (lista-superior)

 lista = combina (lista-inferior, lista-superior)

 fim se

 retorna lista

fim do algoritmo

MergeSort

- ◆ Desenvolvido por John von Neumann
- ◆ Seja uma lista A de n elementos:
 - **Dividir** A em 2 sub-listas de tamanho $\approx n/2$
 - **Ordenar** as sub-listas chamando MergeSort recursivamente, até que elas tenham somente 1 elemento
 - **Combinar** as sub-listas ordenadas formando uma única lista ordenada, intercalando os elementos
- ◆ É um algoritmo **estável**
- ◆ Não é **in-place**

MergeSort

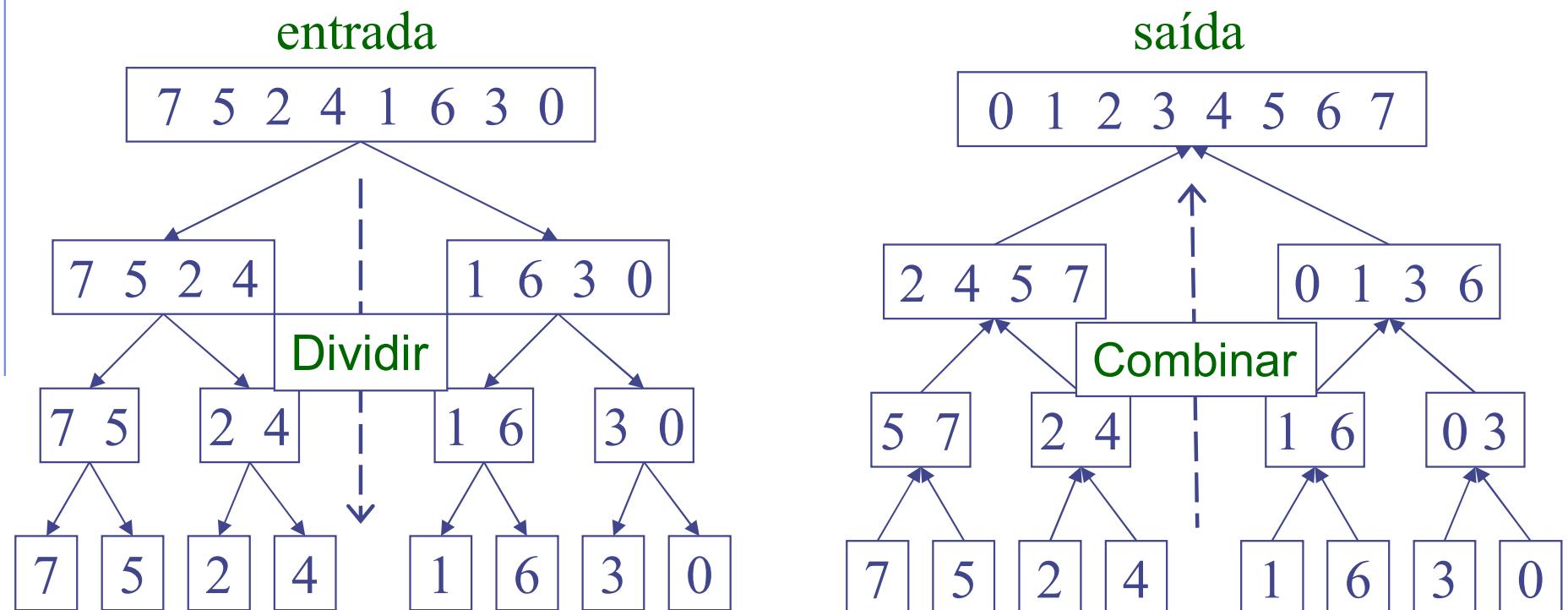
◆ Vantagens

- Extremamente rápido
 - ◆ Complexidade de $O(N \log_2 N)$ em todos os casos

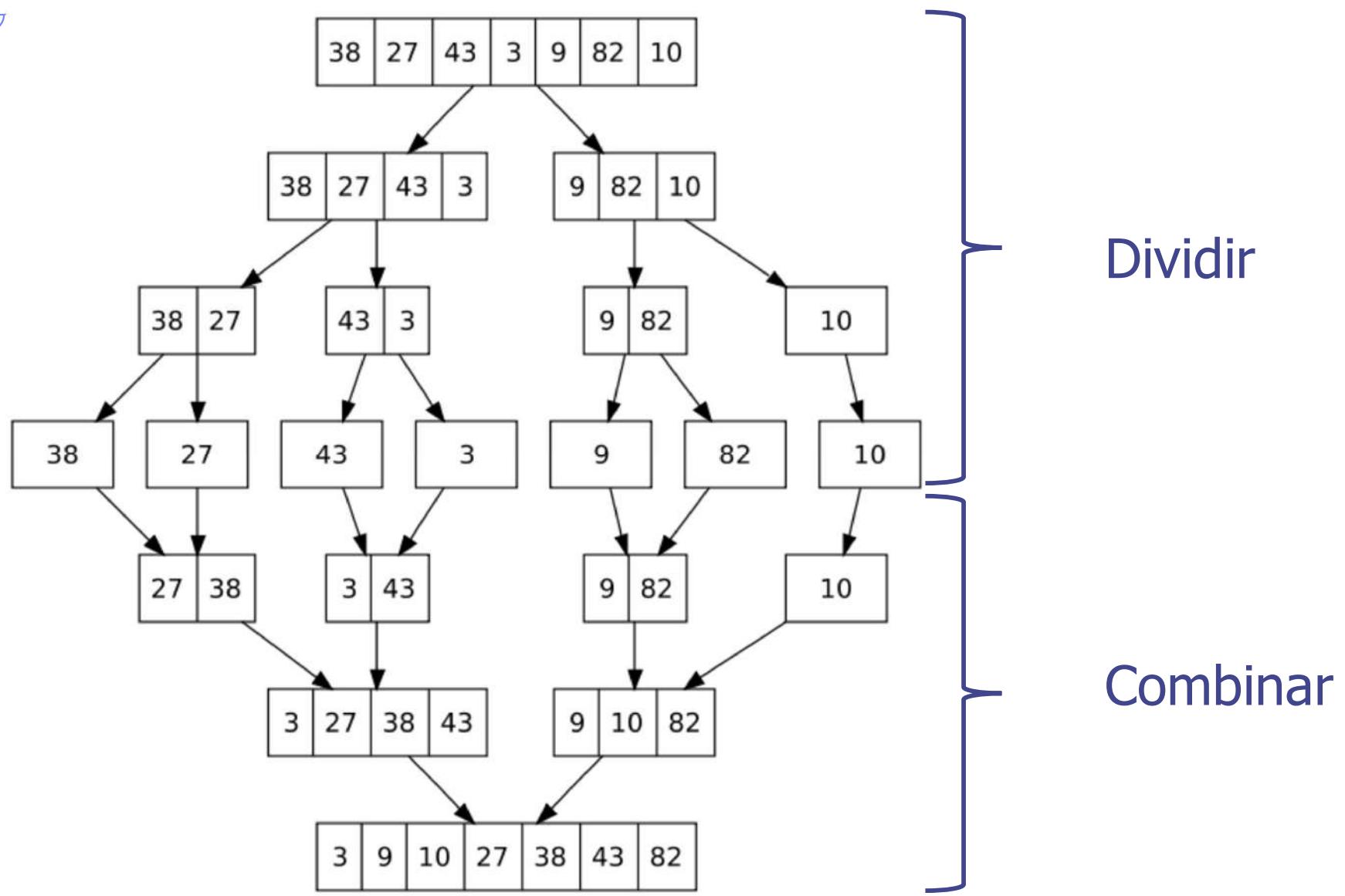
◆ Desvantagens

- Requer o dobro de memória, porque o vetor precisa ser copiado para outro vetor auxiliar
- No melhor caso, a complexidade também é $(N \log N)$

MergeSort - Exemplo



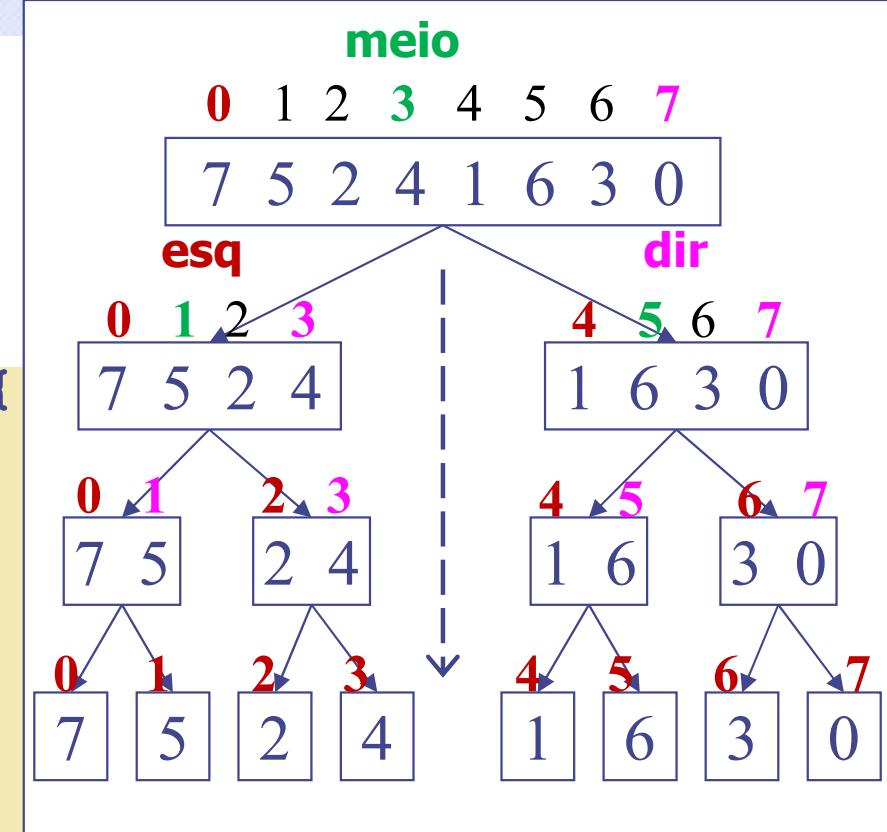
MergeSort - Exemplo



MergeSort em C

```
void mergesort(int vetor[], int n) {  
    int aux[n];  
  
    m_sort(vetor, aux, 0, n-1);  
}
```

```
void m_sort(int vetor[], int aux[], int esq, int dir) {  
    int meio;  
    if (dir > esq) {  
        meio = (dir + esq) / 2;  
        m_sort(vetor, aux, esq, meio);           // Primeira metade  
        m_sort(vetor, aux, meio+1, dir);         // Segunda metade  
        merge(vetor, aux, esq, meio+1, dir);     // Combina as metades  
    }  
}
```



MergeSort em C

```
void merge(int vetor[], int aux[], int esq, int meio, int dir) {  
    int i, esq_fim, n;  
  
    esq_fim = meio-1;      // Posição final do primeiro vetor  
    i = esq;                // Posição inicial do primeiro vetor  
    n = dir - esq + 1;    // Tamanho  
  
    while (esq <= esq_fim && meio <= dir) {  
        if (vetor[esq] <= vetor[meio]) // Seleciona o menor  
            aux[i++] = vetor[esq++];  
        else  
            aux[i++] = vetor[meio++];  
    }  
    // Copia o restante  
    while (esq <= esq_fim) aux[i++] = vetor[esq++];  
  
    while (meio <= dir) aux[i++] = vetor[meio++];  
  
    for (i = 0; i < n; i++) { // Copia: vetor auxiliar para final  
        vetor[dir] = aux[dir];  
        dir--;  
    } }  
}
```

MergeSort em C

```

void mergesort(int vetor[], int n) {
    int aux[n];

    m_sort(vetor, aux, 0, n-1);
}

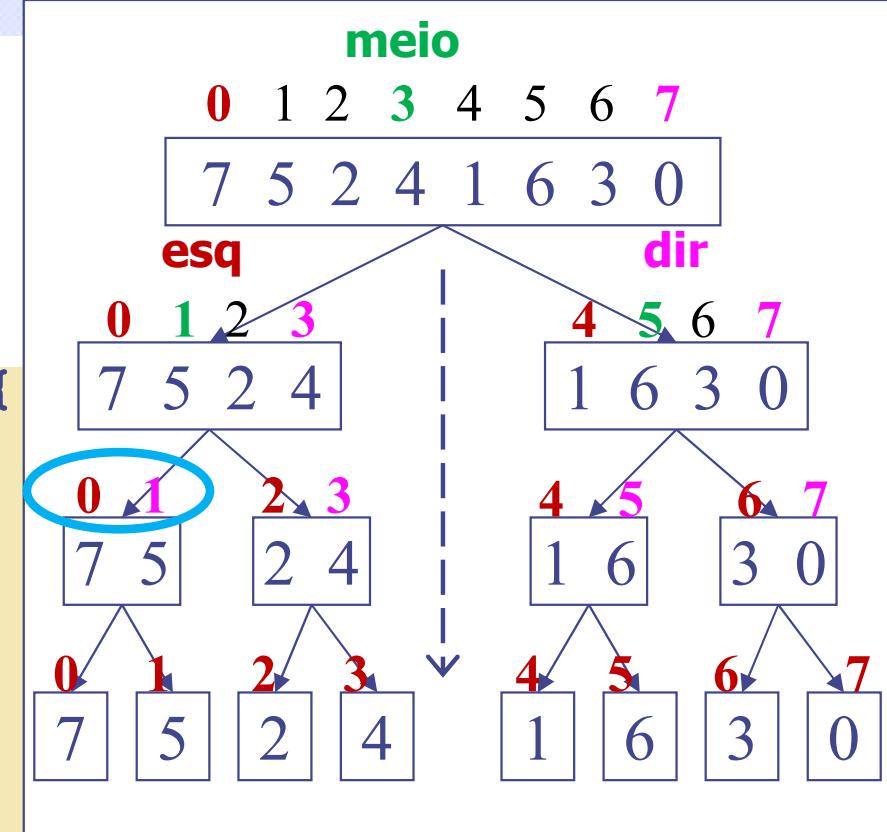
```

```

void m_sort(int vetor[], int aux[], int esq, int dir) {
    int meio;
    if (dir > esq) {
        meio = (dir + esq) / 2;
        m_sort(vetor, aux, esq, meio);           // Primeira metade
        m_sort(vetor, aux, meio+1, dir);         // Segunda metade
        merge(vetor, aux, esq, meio+1, dir);     // Combina as metades
    }
}

```

0 1 1



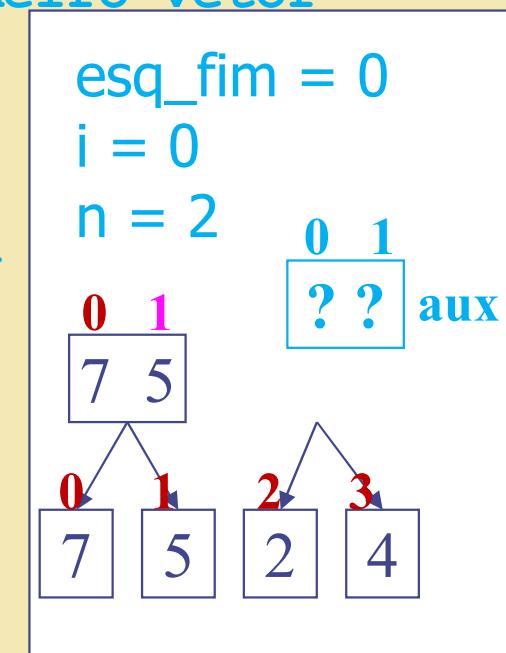
MergeSort em C

0

1

1

```
void merge(int vetor[], int aux[], int esq, int meio, int dir) {  
    int i, esq_fim, n;  
  
    esq_fim = meio-1; // Posição final do primeiro vetor  
    i = esq;           // Posição inicial do primeiro vetor  
    n = dir - esq + 1; // Tamanho  
  
    while (esq <= esq_fim && meio <= dir) {  
        if (vetor[esq] <= vetor[meio]) // Seleciona  
            aux[i++] = vetor[esq++];  
        else  
            aux[i++] = vetor[meio++];  
    }  
    // Copia o restante  
    while (esq <= esq_fim) aux[i++] = vetor[esq++];  
  
    while (meio <= dir) aux[i++] = vetor[meio++];  
  
    for (i = 0; i < n; i++) { // Copia: vetor auxiliar para final  
        vetor[dir] = aux[dir];  
        dir--;  
    } }  
}
```



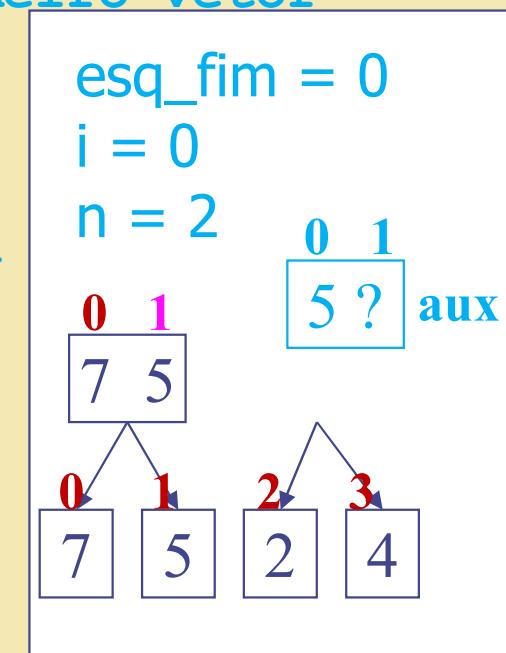
MergeSort em C

0

1

1

```
void merge(int vetor[], int aux[], int esq, int meio, int dir) {  
    int i, esq_fim, n;  
  
    esq_fim = meio-1;      // Posição final do primeiro vetor  
    i = esq;                // Posição inicial do primeiro vetor  
    n = dir - esq + 1;    // Tamanho  
  
    while (esq <= esq_fim && meio <= dir) {  
        if (vetor[esq] <= vetor[meio]) // Seleciona  
            aux[i++] = vetor[esq++];  
        else  
            aux[i++] = vetor[meio++];  
    }  
    // Copia o restante  
    while (esq <= esq_fim) aux[i++] = vetor[esq++];  
  
    while (meio <= dir) aux[i++] = vetor[meio++];  
  
    for (i = 0; i < n; i++) { // Copia: vetor auxiliar para final  
        vetor[dir] = aux[dir];  
        dir--;  
    } }
```



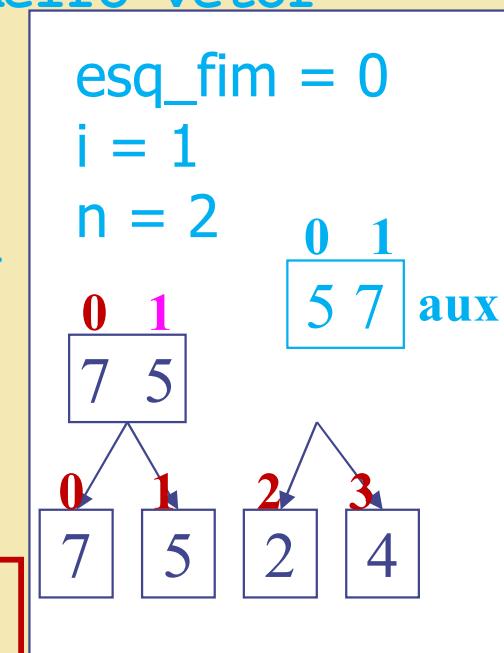
MergeSort em C

0

1

1

```
void merge(int vetor[], int aux[], int esq, int meio, int dir) {  
    int i, esq_fim, n;  
  
    esq_fim = meio-1;    // Posição final do primeiro vetor  
    i = esq;             // Posição inicial do primeiro vetor  
    n = dir - esq + 1;  // Tamanho  
  
    while (esq <= esq_fim && meio <= dir) {  
        if (vetor[esq] <= vetor[meio]) // Seleciona  
            aux[i++] = vetor[esq++];  
        else  
            aux[i++] = vetor[meio++];  
    }  
    // Copia o restante  
    while (esq <= esq_fim) aux[i++] = vetor[esq++];  
    while (meio <= dir) aux[i++] = vetor[meio++];  
  
    for (i = 0; i < n; i++) { // Copia: vetor auxiliar para final  
        vetor[dir] = aux[dir];  
        dir--;  
    } }
```



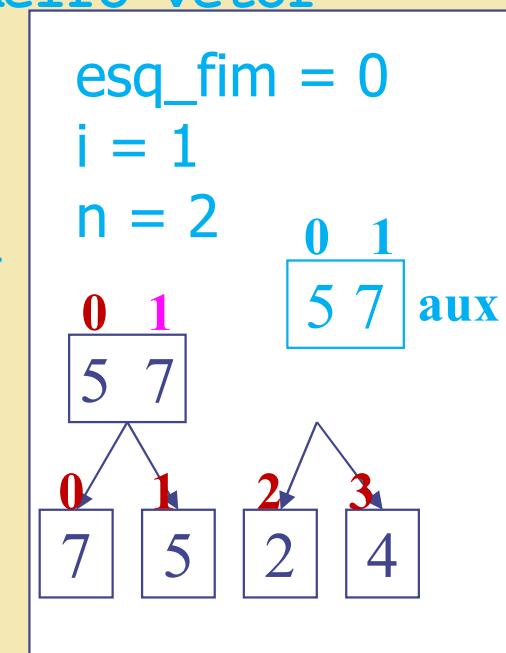
MergeSort em C

0

1

1

```
void merge(int vetor[], int aux[], int esq, int meio, int dir) {  
    int i, esq_fim, n;  
  
    esq_fim = meio-1; // Posição final do primeiro vetor  
    i = esq;           // Posição inicial do primeiro vetor  
    n = dir - esq + 1; // Tamanho  
  
    while (esq <= esq_fim && meio <= dir) {  
        if (vetor[esq] <= vetor[meio]) // Seleciona  
            aux[i++] = vetor[esq++];  
        else  
            aux[i++] = vetor[meio++];  
    }  
    // Copia o restante  
    while (esq <= esq_fim) aux[i++] = vetor[esq++];  
  
    while (meio <= dir) aux[i++] = vetor[meio++];  
  
    for (i = 0; i < n; i++) { // Copia: vetor auxiliar para final  
        vetor[dir] = aux[dir];  
        dir--;  
    } }  
}
```



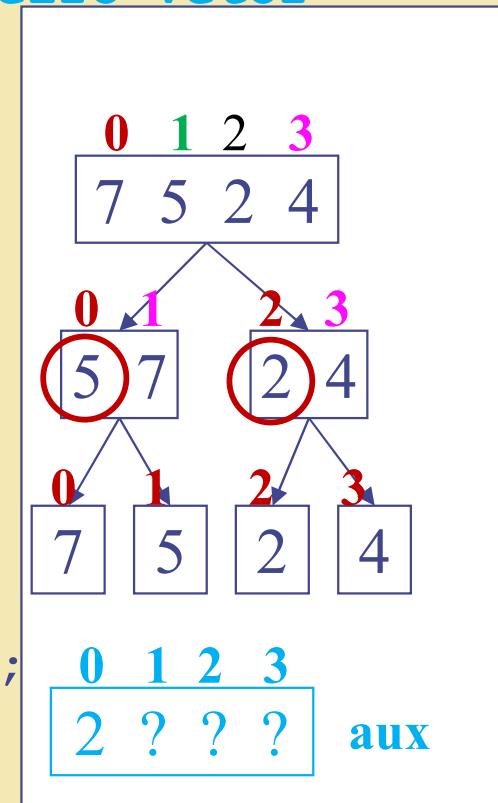
MergeSort em C

0

2

3

```
void merge(int vetor[], int aux[], int esq, int meio, int dir) {  
    int i, esq_fim, n;  
  
    esq_fim = meio-1;      // Posição final do primeiro vetor  
    i = esq;                // Posição inicial do primeiro vetor  
    n = dir - esq + 1;    // Tamanho  
  
    while (esq <= esq_fim && meio <= dir) {  
        if (vetor[esq] <= vetor[meio]) // Seleciona  
            aux[i++] = vetor[esq++];  
        else  
            aux[i++] = vetor[meio++];  
    }  
    // Copia o restante  
    while (esq <= esq_fim) aux[i++] = vetor[esq++];  
    while (meio <= dir) aux[i++] = vetor[meio++];  
  
    for (i = 0; i < n; i++) { // Copia: vetor auxiliar para final  
        vetor[dir] = aux[dir];  
        dir--;  
    } }
```



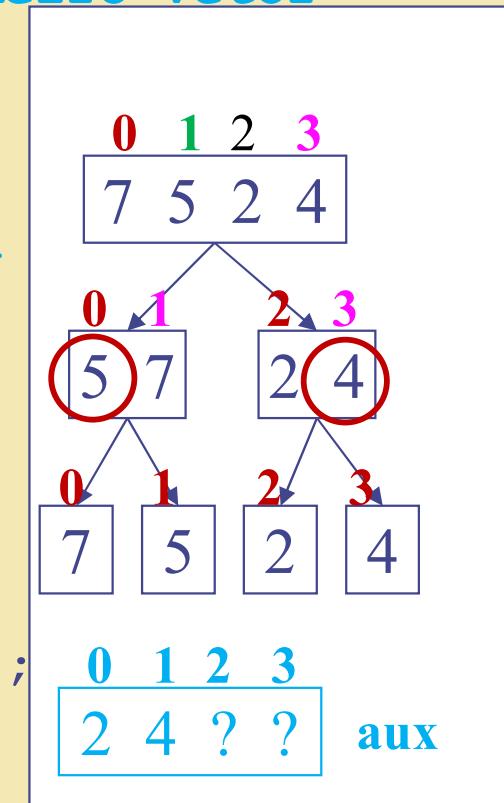
MergeSort em C

0

2

3

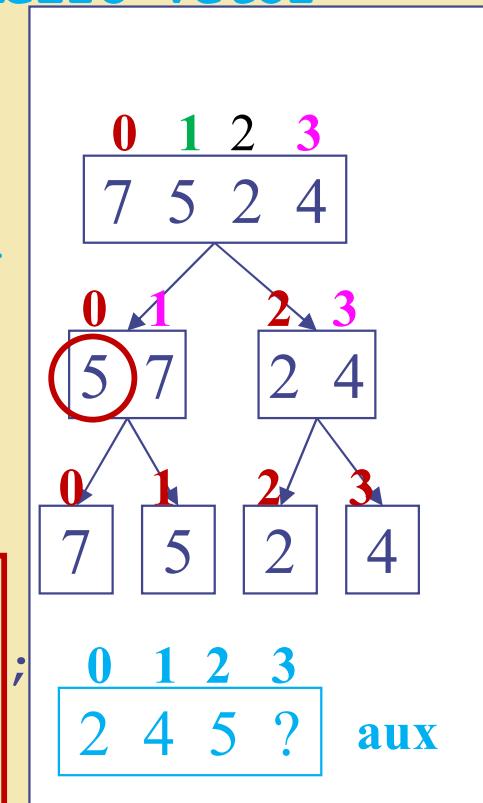
```
void merge(int vetor[], int aux[], int esq, int meio, int dir) {  
    int i, esq_fim, n;  
  
    esq_fim = meio-1;    // Posição final do primeiro vetor  
    i = esq;              // Posição inicial do primeiro vetor  
    n = dir - esq + 1;   // Tamanho  
  
    while (esq <= esq_fim && meio <= dir) {  
        if (vetor[esq] <= vetor[meio]) // Seleciona  
            aux[i++] = vetor[esq++];  
        else  
            aux[i++] = vetor[meio++];  
    }  
    // Copia o restante  
    while (esq <= esq_fim) aux[i++] = vetor[esq++];  
    while (meio <= dir) aux[i++] = vetor[meio++];  
  
    for (i = 0; i < n; i++) { // Copia: vetor auxiliar para final  
        vetor[dir] = aux[dir];  
        dir--;  
    } }
```



MergeSort em C

0 2 3

```
void merge(int vetor[], int aux[], int esq, int meio, int dir) {  
    int i, esq_fim, n;  
  
    esq_fim = meio-1; // Posição final do primeiro vetor  
    i = esq;           // Posição inicial do primeiro vetor  
    n = dir - esq + 1; // Tamanho  
  
    while (esq <= esq_fim && meio <= dir) {  
        if (vetor[esq] <= vetor[meio]) // Seleciona  
            aux[i++] = vetor[esq++];  
        else  
            aux[i++] = vetor[meio++];  
    }  
    // Copia o restante  
    while (esq <= esq_fim) aux[i++] = vetor[esq++];  
    while (meio <= dir) aux[i++] = vetor[meio++];  
  
    for (i = 0; i < n; i++) { // Copia: vetor auxiliar para final  
        vetor[dir] = aux[dir];  
        dir--;  
    } }
```



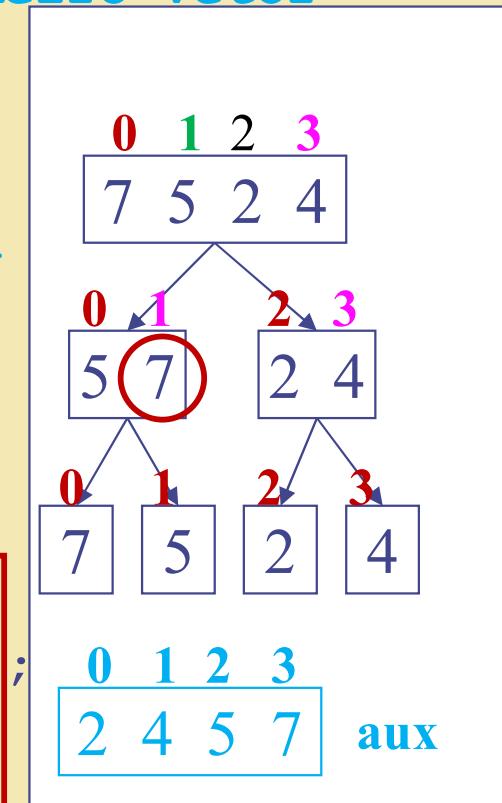
MergeSort em C

0

2

3

```
void merge(int vetor[], int aux[], int esq, int meio, int dir) {  
    int i, esq_fim, n;  
  
    esq_fim = meio-1; // Posição final do primeiro vetor  
    i = esq;           // Posição inicial do primeiro vetor  
    n = dir - esq + 1; // Tamanho  
  
    while (esq <= esq_fim && meio <= dir) {  
        if (vetor[esq] <= vetor[meio]) // Seleciona  
            aux[i++] = vetor[esq++];  
        else  
            aux[i++] = vetor[meio++];  
    }  
    // Copia o restante  
    while (esq <= esq_fim) aux[i++] = vetor[esq++];  
    while (meio <= dir) aux[i++] = vetor[meio++];  
  
    for (i = 0; i < n; i++) { // Copia: vetor auxiliar para final  
        vetor[dir] = aux[dir];  
        dir--;  
    } }
```



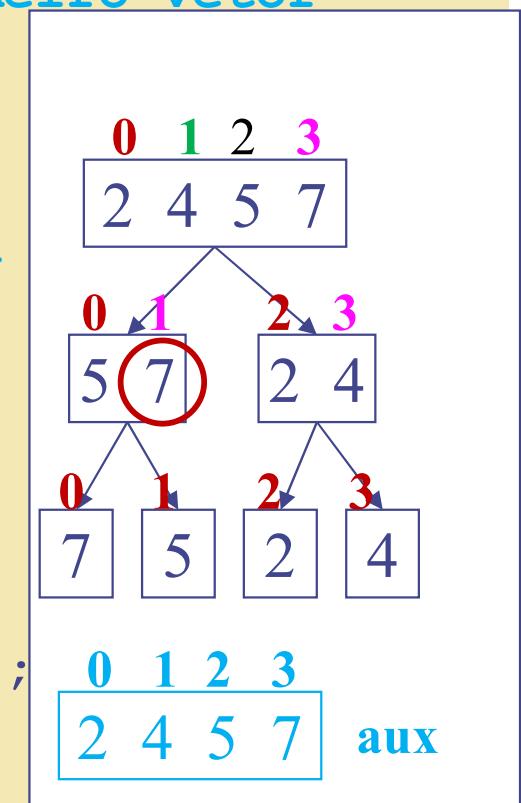
MergeSort em C

0

2

3

```
void merge(int vetor[], int aux[], int esq, int meio, int dir) {  
    int i, esq_fim, n;  
  
    esq_fim = meio-1; // Posição final do primeiro vetor  
    i = esq;           // Posição inicial do primeiro vetor  
    n = dir - esq + 1; // Tamanho  
  
    while (esq <= esq_fim && meio <= dir) {  
        if (vetor[esq] <= vetor[meio]) // Seleciona  
            aux[i++] = vetor[esq++];  
        else  
            aux[i++] = vetor[meio++];  
    }  
    // Copia o restante  
    while (esq <= esq_fim) aux[i++] = vetor[esq++];  
    while (meio <= dir) aux[i++] = vetor[meio++];  
  
    for (i = 0; i < n; i++) // Copia: vetor auxiliar para final  
        vetor[dir] = aux[dir];  
        dir--;  
    } }
```



MergeSort

◆ Link:

- <https://visualgo.net/en/sorting>

MergeSort

Análise da Complexidade

◆ Trecho de código relevante

```
void m_sort(int vetor[], int aux[], int esq, int dir) {  
    int meio;  
    if (dir > esq) {  
        meio = (dir + esq) / 2;  
        m_sort(vetor, aux, esq, meio);           // Primeira metade  
        m_sort(vetor, aux, meio+1, dir);         // Segunda metade  
        merge(vetor, aux, esq, meio+1, dir);     // Combina as metades  
    }  
}
```

◆ Tempo de execução → merge

- combina dois vetores de tamanhos m_1 e $m_2 \rightarrow n = m_1 + m_2$

MergeSort

Análise da Complexidade

◆ Trecho de código relevante

```
void m_sort(int vetor[], int aux[], int esq, int dir) {  
    int meio;  
    if (dir > esq) {  
        meio = (dir + esq) / 2;  
        m_sort(vetor, aux, esq, meio); // Primeira metade  
        m_sort(vetor, aux, meio+1, dir); // Segunda metade  
        merge(vetor, aux, esq, meio+1, dir); // Combina as metades  
    }  
}
```

n/2 elementos

◆ Tempo de execução → merge

- combina dois vetores de tamanhos m_1 e $m_2 \rightarrow n = m_1 + m_2$

MergeSort

Análise da Complexidade

◆ Expressão de recorrência

$$T(n) = \boxed{2T(n/2)} + \boxed{n}$$



chamadas
recursivas combinação

◆ A solução para essa recorrência

- $T(n) = O(n \log n)$
- qualquer que seja o vetor de entrada, o algoritmo trabalhará da mesma maneira

QuickSort

- ◆ É considerado o algoritmo de ordenação de propósito geral mais rápido que existe
- ◆ Seja uma lista A de n elementos:
 - **Dividir** A em 2 sub-listas tal que todos os elementos de uma das sub-listas sejam menores do que da outra
 - ◆ Escolher um elemento para ser o “pivô”
 - ◆ Sub-lista menor: todos os elementos \leq pivô
 - ◆ Sub-lista maior: todos os elementos $>$ pivô
 - **Ordenar** as sub-listas chamando QuickSort recursivamente, até que elas tenham somente 1 elemento
 - **Juntar (concatenar)** as duas sub-listas novamente

QuickSort

◆ Vantagens

- Extremamente rápido
 - ◆ Complexidade $O(N \log N)$ melhor caso e caso médio
 - ◆ Em alguns casos, ainda mais rápido
- Não requer memória adicional (**in-place**)

◆ Desvantagens

- Complexo
- **Não** é um algoritmo **estável**

QuickSort

◆ Desvantagens

- Como escolher o pivô?
 - ◆ Existem várias abordagens diferentes
 - ◆ No pior caso o pivô divide o array de N em dois: uma partição com N-1 elementos e outra com 1 elemento
 - ◆ Particionamento não é balanceado
 - ◆ Quando isso acontece a cada nível da recursão, temos o tempo de execução de $O(N^2)$

QuickSort

◆ Desvantagens

- No caso de um particionamento não balanceado, o insertion sort acaba sendo mais eficiente que o quicksort
- O pior caso do quicksort ocorre quando o array já está ordenado, uma situação onde a complexidade é $O(N)$ no insertion sort

Implementação Ingênua

- ◆ Usa memória adicional em cada chamada recursiva
- ◆ Aloca espaço para duas sub-listas: I (inferior) e S (superior)
- ◆ Fase da divisão
 - Se o elemento > pivô, insere em S
 - Senão, insere em I
- ◆ Fase da recursão
 - Simplesmente chamar o QuickSort para I e S
- ◆ Fase da conquista
 - Retirar os elementos de I e inserir novamente em L
 - Retirar os elementos de S e inserir novamente em L

Implementação Eficiente

- ◆ Usar o espaço de memória da própria lista para gerar as sub-listas
 - Usando um vetor isto é fácil
- ◆ Fase dividir
 - Procurar elemento maior que o pivô iniciando no limite inferior do vetor de maneira crescente (onde deveriam estar os menores)
 - Procurar elemento menor que o pivô iniciando no limite superior do vetor de maneira decrescente (onde deveriam estar os maiores)
 - Trocar esses dois elementos, se existirem

QuickSort – Exemplo

1ª execução do laço

Vetor de 0 a 9

$$\text{pivô} = (0+9)/2 = 4$$

0	1	2	3	4	5	6	7	8	9
5	8	3	1	6	2	4	9	7	5

↑
pivô

5 <= 6 → V → para
j = 9

0	1	2	3	4	5	6	7	8	9
5	8	3	1	6	2	4	9	7	5

↑
pivô

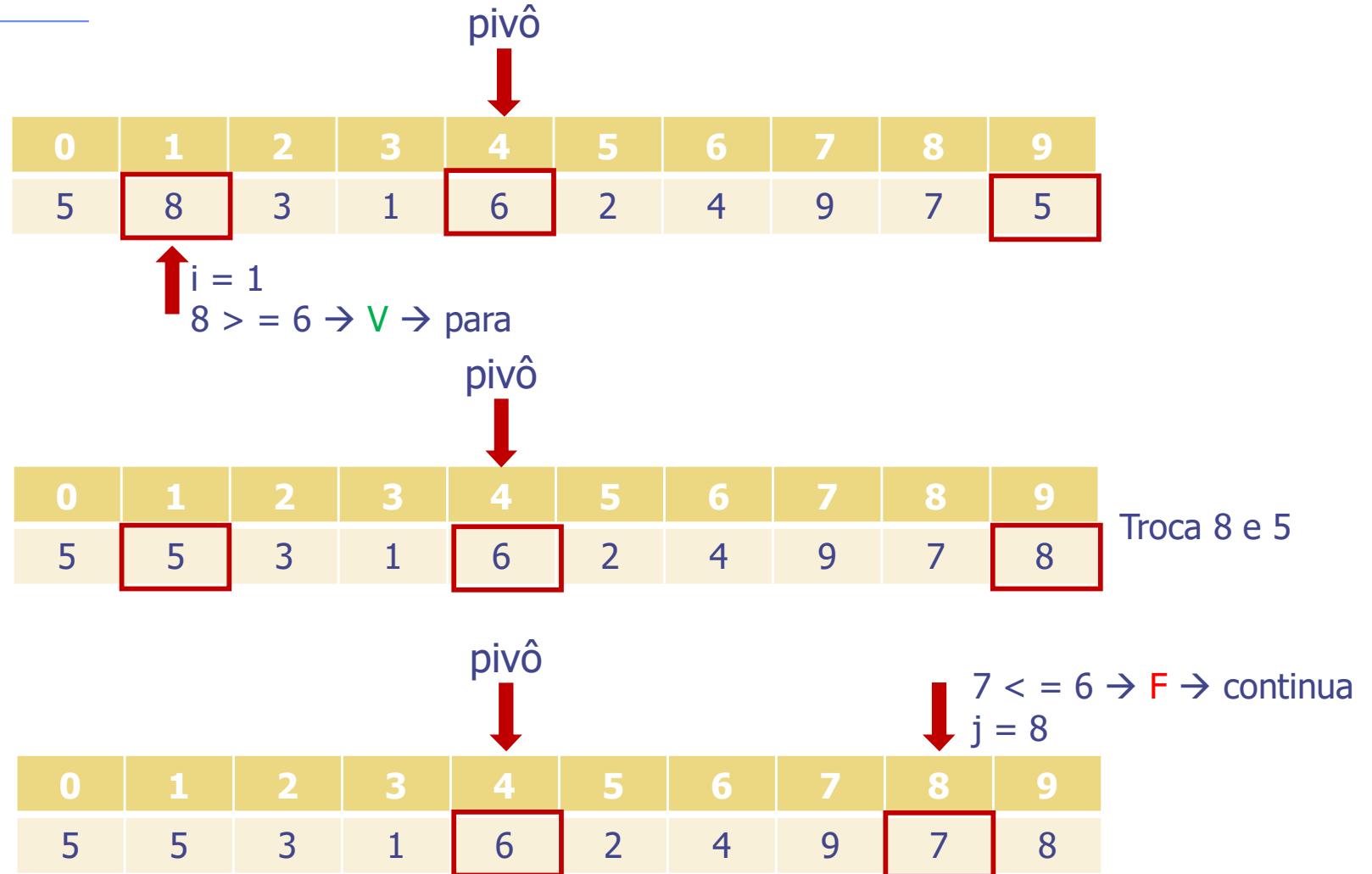
**procura por
um elemento
menor**

0	1	2	3	4	5	6	7	8	9
5	8	3	1	6	2	4	9	7	5

↑ i = 0
5 >= 6 → F → continua
↑
pivô

**procura por
um elemento
maior**

QuickSort – Exemplo



QuickSort – Exemplo

Diagram illustrating the first step of QuickSort on the array [5, 5, 3, 1, 6, 2, 4, 9, 7, 8, 9]. The pivot element is 6, highlighted with a red box. A red arrow labeled "pivô" points to the pivot. A red arrow labeled "j = 7" points to the element 9 at index 7. Text above indicates $9 \leq 6 \rightarrow F \rightarrow \text{continua}$.

0	1	2	3	4	5	6	7	8	9
5	5	3	1	6	2	4	9	7	8

Diagram illustrating the second step of QuickSort. The pivot element is 6, highlighted with a red box. A red arrow labeled "pivô" points to the pivot. A red arrow labeled "j = 6" points to the element 4 at index 6. Text above indicates $4 \leq 6 \rightarrow V \rightarrow \text{para}$.

0	1	2	3	4	5	6	7	8	9
5	5	3	1	6	2	4	9	7	8

Diagram illustrating the third step of QuickSort. The pivot element is 6, highlighted with a red box. A red arrow labeled "pivô" points to the pivot. A red arrow labeled "i = 2" points to the element 3 at index 2. Text below indicates $3 \geq 6 \rightarrow F \rightarrow \text{continua}$.

0	1	2	3	4	5	6	7	8	9
5	5	3	1	6	2	4	9	7	8

QuickSort – Exemplo

0	1	2	3	4	5	6	7	8	9
5	5	3	1	6	2	4	9	7	8

$i = 3$
 $1 >= 6 \rightarrow F \rightarrow$ continua

0	1	2	3	4	5	6	7	8	9
5	5	3	1	6	2	4	9	7	8

$i = 4$
 $6 >= 6 \rightarrow V \rightarrow$ para

0	1	2	3	4	5	6	7	8	9
5	5	3	1	4	2	6	9	7	8

Troca 6 e 4

QuickSort – Exemplo

2ª execução do laço

Vetor de 0 a 5
 $\text{pivô} = (0+5)/2 = 2$

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

pivô

$2 \leq 3 \rightarrow V \rightarrow \text{para}$
j = 5

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

pivô

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

i = 0

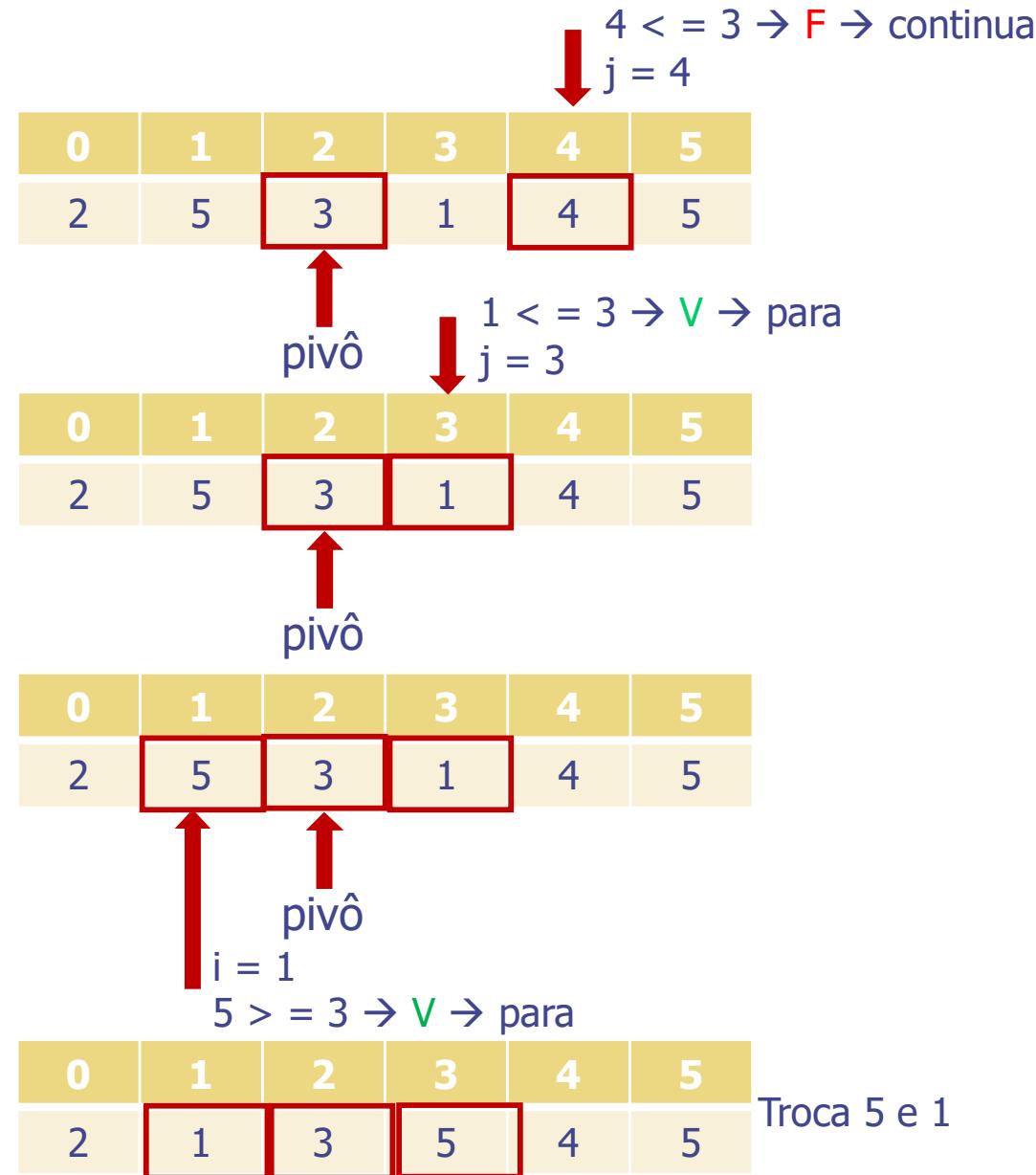
pivô

$5 \geq 3 \rightarrow V \rightarrow \text{para}$

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

Troca 5 e 2

QuickSort – Exemplo



QuickSort – Exemplo

3ª execução do laço

Vetor de 0 a 2

$$\text{pivô} = (0+2)/2 = 1$$

0	1	2
2	1	3

pivô

$$3 \leq 1 \rightarrow \text{F} \rightarrow \text{continua}$$

0	1	2
2	1	3

pivô

$$1 \leq 1 \rightarrow \text{V} \rightarrow \text{para}$$

0	1	2
2	1	3

pivô

0	1	2
2	1	3

pivô

i = 0

$$2 \geq 1 \rightarrow \text{V} \rightarrow \text{para}$$

0	1	2
1	2	3

Troca 2 e 1

pivô

QuickSort – Exemplo

4ª execução do laço

Vetor de 1 a 2

$$\text{pivô} = (1+2)/2 = 1$$

1	2
2	3

pivô

$3 \leq 2 \rightarrow \text{F} \rightarrow \text{continua}$

1	2
2	3

pivô

$2 \leq 2 \rightarrow \text{V} \rightarrow \text{para}$

1	2
2	3

pivô

1	2
2	3

i = 1

$2 \geq 2 \rightarrow \text{V} \rightarrow \text{para}$

1	2
2	3

pivô

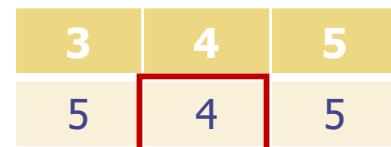
$i < j \rightarrow 1 < 1 \rightarrow \text{F} \rightarrow \text{não troca}$

QuickSort – Exemplo

5ª execução do laço

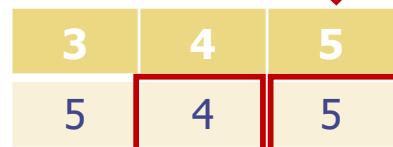
Vetor de 3 a 5

$$\text{pivô} = (3+5)/2 = 4$$



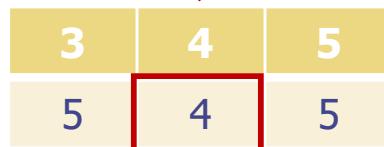
pivô

$5 \leq 4 \rightarrow \text{F}$ → continua
j = 5

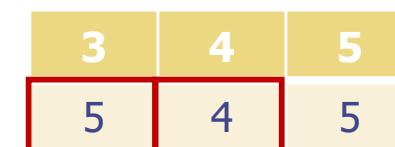


pivô

$4 \leq 4 \rightarrow \text{V}$ → para
j = 4

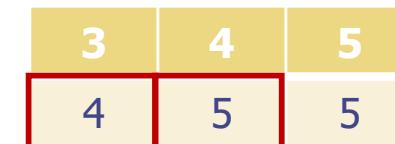


pivô



pivô

i = 3
 $5 \geq 4 \rightarrow \text{V}$ → para



pivô

Troca 5 e 4

QuickSort – Exemplo

6ª execução do laço

Vetor de 4 a 5

$$\text{pivô} = (4+5)/2 = 4$$

4	5
5	5

pivô

4	5
5	5

pivô

Troca 5 e 5

4	5
5	5

pivô

4	5
5	5

pivô

$i = 4$

$5 >= 5 \rightarrow V \rightarrow \text{para}$

QuickSort – Exemplo

6	7	8	9
6	9	7	8

pivô

$8 \leq 9 \rightarrow V \rightarrow$ para
 $j = 9$

6	7	8	9
6	9	7	8

pivô

6	7	8	9
6	9	7	8

pivô

$i = 6$

$6 \geq 9 \rightarrow F \rightarrow$ continua

7ª execução do laço

Vetor de 6 a 9
 $\text{pivô} = (6+9)/2 = 7$

6	7	8	9
6	9	7	8

pivô

$i = 7$
 $9 \geq 9 \rightarrow V \rightarrow$ para

6	7	8	9
6	8	7	9

pivô

Troca 9 e 8

QuickSort – Exemplo

8ª execução do laço

Vetor de 6 a 8

$$\text{pivô} = (6+8)/2 = 7$$

6	7	8
6	8	7

pivô

$$7 \leq 8 \rightarrow V \rightarrow \text{para}$$

6	7	8
6	8	7

pivô

6	7	8
6	8	7

pivô

i = 6

$6 \geq 8 \rightarrow F \rightarrow \text{continua}$

6	7	8
6	8	7

pivô

i = 7

$8 \geq 8 \rightarrow V \rightarrow \text{para}$

6	7	8
6	7	8

Troca 8 e 7

pivô

QuickSort – Exemplo

9ª execução do laço

Vetor de 6 a 7

$$\text{pivô} = (6+7)/2 = 6$$

6	7
6	7

pivô

$7 \leq 6 \rightarrow F \rightarrow$ continua

6	7
6	7

pivô

$6 \leq 6 \rightarrow V \rightarrow$ para
 $j = 6$

6	7
6	7

pivô

6	7
6	7

pivô

$i = 6$
 $6 \geq 6 \rightarrow V \rightarrow$ para

6	7
6	7

pivô

$i < j \rightarrow 6 < 6 \rightarrow F \rightarrow$ não troca

Vetor ordenado

0	1	2	3	4	5	6	7	8	9
1	2	3	4	5	5	6	7	8	9

QuickSort em C

```
void quicksort (int vetor[], int n) {  
    q_sort (vetor, 0, n-1);  
}  
  
void q_sort (int vetor[], int esq, int dir) {  
    int meio;  
  
    if (esq < dir) {  
        meio = particao (vetor, esq, dir);  
        q_sort (vetor, esq, meio);  
        q_sort (vetor, meio + 1, dir);  
    }  
}
```

QuickSort em C

```
int particao (int vet[], int esq, int dir)
{
    int i, j, pivo, temp ;
    pivo = vet[(esq + dir) / 2]; //define pivo
    i = esq - 1;
    j = dir + 1;
    while (i < j) {
        do{ i++;} while (vet[i] < pivo); // procura maior
        do{ j--;} while (vet[j] > pivo); // procura menor
        if (i < j) { //troca
            temp = vet[i];
            vet[i] = vet[j];
            vet[j] = temp;
        }
    }
    return (j);
}
```

QuickSort

◆ Link:

- <https://visualgo.net/en/sorting>

◆ Comparando MergeSort vc QuickSort

- <https://www.toptal.com/developers/sorting-algorithms>

Exercícios

- ◆ Dado o vetor $X = [5, 3, 1, 9, 7, 2, 4, 5, 8, 3]$, ordená-lo em ordem crescente, detalhadamente, pelo método mergesort.

- ◆ Dado o vetor $X = [17, 12, 6, 19, 23, 8, 5, 10]$, ordená-lo em ordem crescente, detalhadamente, pelo método quicksort.

Bibliografia

- ◆ Projeto de Algoritmos, Nivio Ziviani.
- ◆ Estruturas de Dados, Ana Fernanda Ascencio e Graziela Santos de Araújo.
- ◆ Algoritmos: Teoria e Prática, Thomas H. Cormen et al.
- ◆ Prof. André Backes. Slides sobre Algoritmos de Ordenação. Disponíveis em
<https://programacaodescomplicada.wordpress.com/complementar/>

Bacharelado em Ciência da Computação
GBC034 Algoritmos e Estruturas de Dados 2

Prática 2 Ordenação

Profa. Maria Camila Nardini Barioni

camila.barioni@ufu.br

Bloco B - sala 1B137

2º semestre de 2023

Informações gerais

- Esta prática tem por objetivo reforçar o conhecimento do(a) aluno(a) com relação aos algoritmos de ordenação simples estudados em sala

- Data da entrega: 02/02/2024
- Grupo de até 3 (dois) alunos(as). Incluir nome e matrícula como comentário no código
- Linguagem de programação a ser usada: C
- O que deve ser entregue: Link para o repositório do código (Replit.com) e documento .pdf com as tabelas solicitadas na descrição da atividade
- Essa prática vale 01 ponto

Exercícios - Prática

1. Dados os métodos de ordenação estudados para organizar vetores de números inteiros, o objetivo do exercício é construir uma tabela de referência contendo o tempo obtido para cada método de ordenação, considerando entrada de dados aleatória (usar geração aleatória de números inteiros) e vetor de tamanho 10.000.

Exemplo

Método	Tempo
BubbleSort	0.0020
ShellSort	0.0001
MergeSort	0.0001

Exercícios - Prática

- ◆ Modifique o código abaixo:

```
#define N 8
int main (void) {
    int i;
    int v[N]= {25, 48, 37, 12, 57, 86, 33, 92};
    bubbleSort(v,8);
    printf("vetor ordenado:");
    for(i=0, i<N;i++)
        printf("%d", v[i]);
    printf("\n");
    return 0;
}
```

- Inserir código para medir o tempo de execução
- Considerar alocação dinâmica
- Atribuir valores aleatórios ao vetor considerando tamanho 10.000

Exercícios - Prática

- ◆ Exemplo de como calcular o tempo de execução de uma função

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <time.h>
4
5 int main(){
6     time_t ini;
7     time_t fim;
8     double tempo_total;
9     int vetor[TAM];
10
11     ini=clock();
12     QuickSort(vetor,0,TAM-1);
13     fim=clock();
14
15     tempo_total=(double)(fim-ini)/CLOCKS_PER_SEC;
16
17     printf("\nTempo total =%f\n",tempo_total);
18 }
19
```

Exercícios - Prática

- ◆ Exemplo de alocação dinâmica para array com 10 posições em C

```
int v[10];  
  
int *v = malloc(10*sizeof(int));
```

- ◆ Exemplo de geração de números aleatórios entre 0 e 499

```
int aux = rand()%500;
```

Exercícios - Prática

- 
2. Incrementar a tabela com os tempos para vetores de números inteiros com os tamanhos 50.000 e 100.000, ainda considerando entrada de dados aleatória.

Exercícios - Prática

3. Incluir na tabela de referência o novo método ABC, conforme o algoritmo abaixo:

```
ABC-SORT (A, i, j)
1   if A[i] > A[j]
2       then trocar A[i] e A[j]
3   if i + 1 >= j
4       then return
5   k ← (j-i+1)/3    // divisão inteira (arredondar para baixo)
6   ABC-SORT (A, i, j-k)
7   ABC-SORT (A, i+k, j)
8   ABC-SORT (A, i, j-k)
```

Exercícios - Prática

- Mais detalhes sobre esse método de ordenação em:
 - https://pt.wikipedia.org/wiki/Stooge_sort
- Teste seus conhecimentos sobre o método StoogeSort
 - <https://www.sanfoundry.com/stooge-sort-multiple-choice-questions-answers-mcqs/>

Exercícios - Prática

4. Construir uma segunda tabela de referência contendo o **número médio de comparações** e o **número médio de trocas** considerando entrada aleatória e vetor de tamanho 10.000.
- Para esse experimento, utilize 10 vetores distintos.

Bacharelado em Ciência da Computação
GBC034 Algoritmos e Estruturas de Dados 2

Ordenação

Profa. Maria Camila Nardini Barioni

camila.barioni@ufu.br

Bloco B - sala 1B137

2º semestre de 2023

Avisos

- ◆ 09/02: a aula será no Lab 04

Roteiro

- ◆ Métodos de Ordenação Eficientes
 - ◆ Parte 2
 - ◆ Método da seleção em árvore (HeapSort)
 - ◆ Método da distribuição de chave (BucketSort)

PARTE 2

HeapSort

- ◆ Método de ordenação que é sempre $O(n \log n)$, para todos os casos
- ◆ Usa uma estrutura chamada heap, cujo tempo de inserção e remoção é $O(\log n)$ e que sempre possui o maior valor na raiz
- ◆ Método
 - Inserir todos os elementos do vetor na heap
 - Remover o elemento que está no nó raiz da heap e adicionar novamente no vetor
- ◆ A heap pode ser construída no próprio vetor

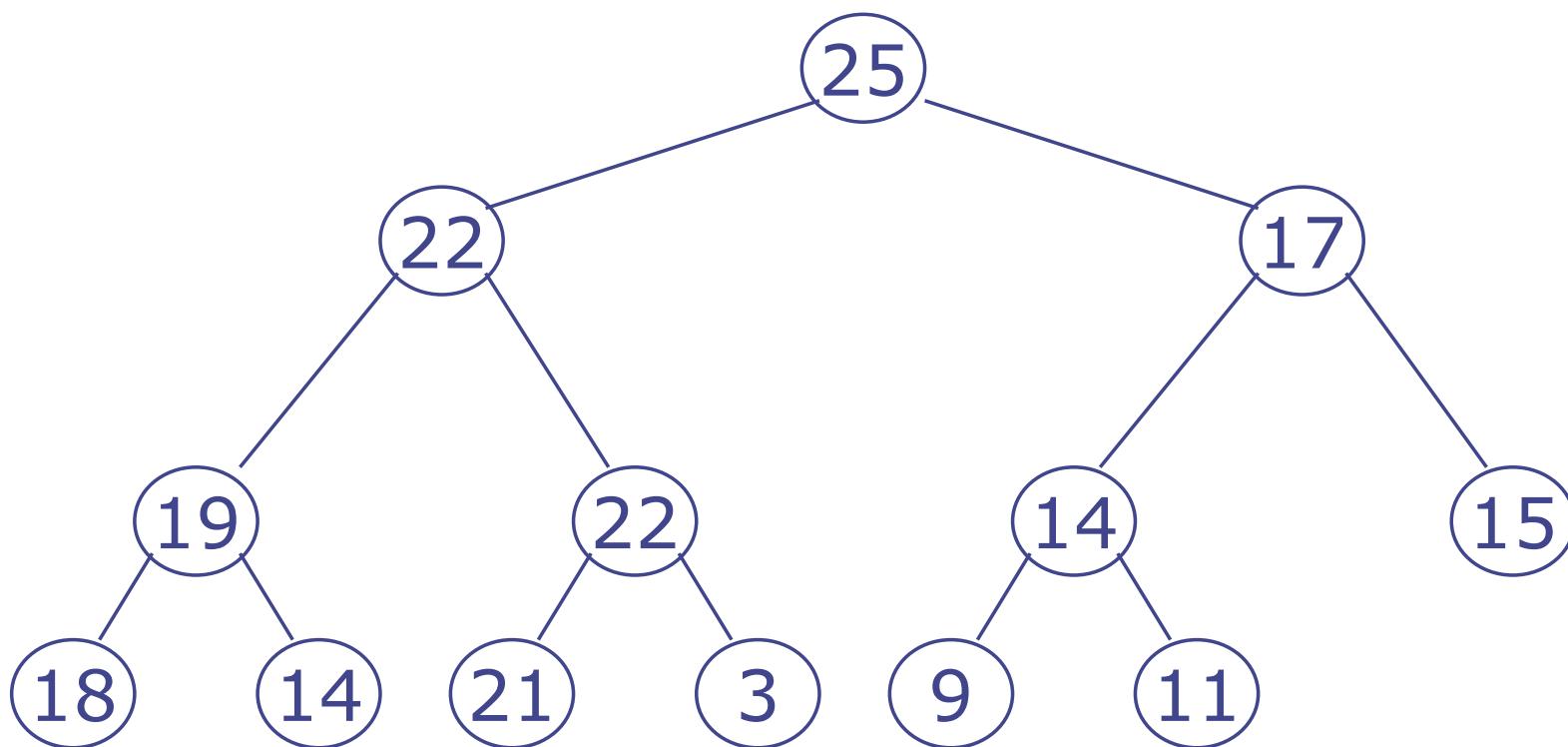
Heap

- ◆ Heap (binário) é um array que pode ser visto como uma árvore binária que satisfaz uma restrição
 - Em qualquer nó v (exceto a raiz) a chave armazenada em v é \leq (ou \geq) chave armazenada no nó pai de v
 - Garante que o maior (ou menor) elemento está sempre na raiz
- ◆ Além disso, a heap deve ter altura mínima
 - A árvore é completa ou “justificada à esquerda”
 - ◆ Quando no último nível de uma árvore todos os nós estão mais a esquerda
 - Garante que as operações tem um custo de $\log(N)$ e a heap pode ser representada como um vetor

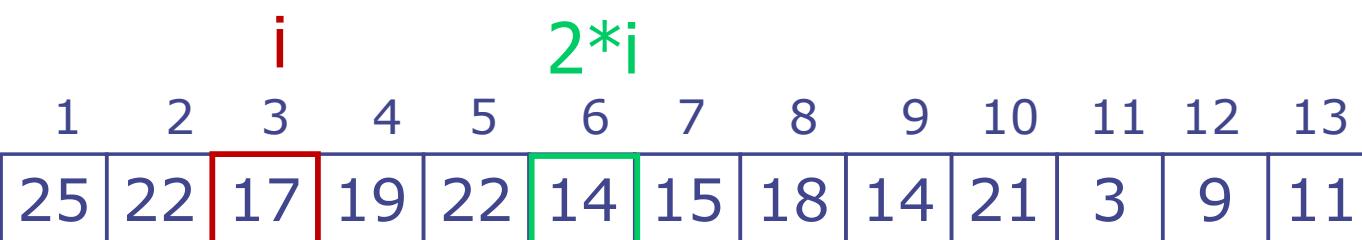
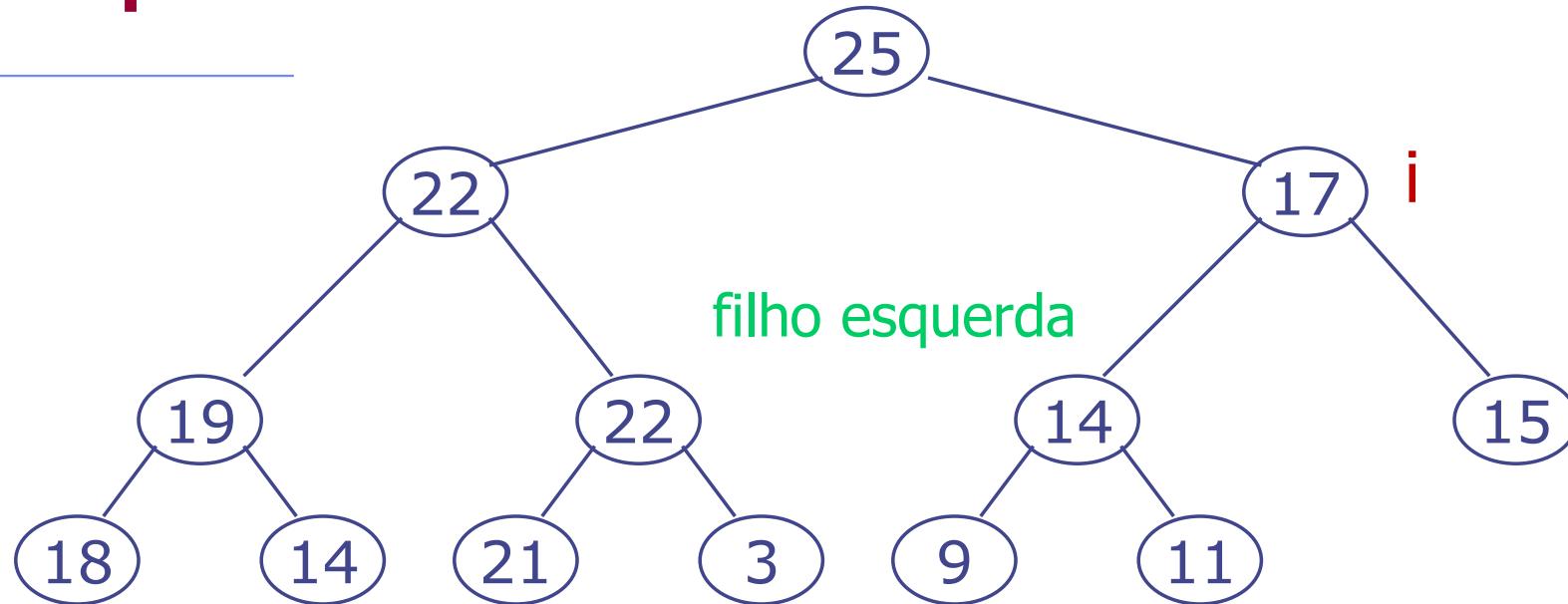
Heap

- ◆ Um array A que representa um heap tem dois atributos
 - $A.\text{comprimento}$: número de elementos do array
 - $A.\text{tamanho-do-heap}$: número de elementos do heap armazenado em A
 - ♦ $A.\text{tamanho-do-heap} \leq A.\text{comprimento}$
- ◆ A raiz da árvore é $A[1]$
- ◆ Dado o índice i de um nó, os índices de seu pai, do filho a esquerda, e do filho a direita podem ser calculados da seguinte forma
 - $\text{parent}(i) = \lfloor i/2 \rfloor$
 - $\text{left}(i) = 2i$
 - $\text{right} = 2i + 1$

Heap - Exemplo

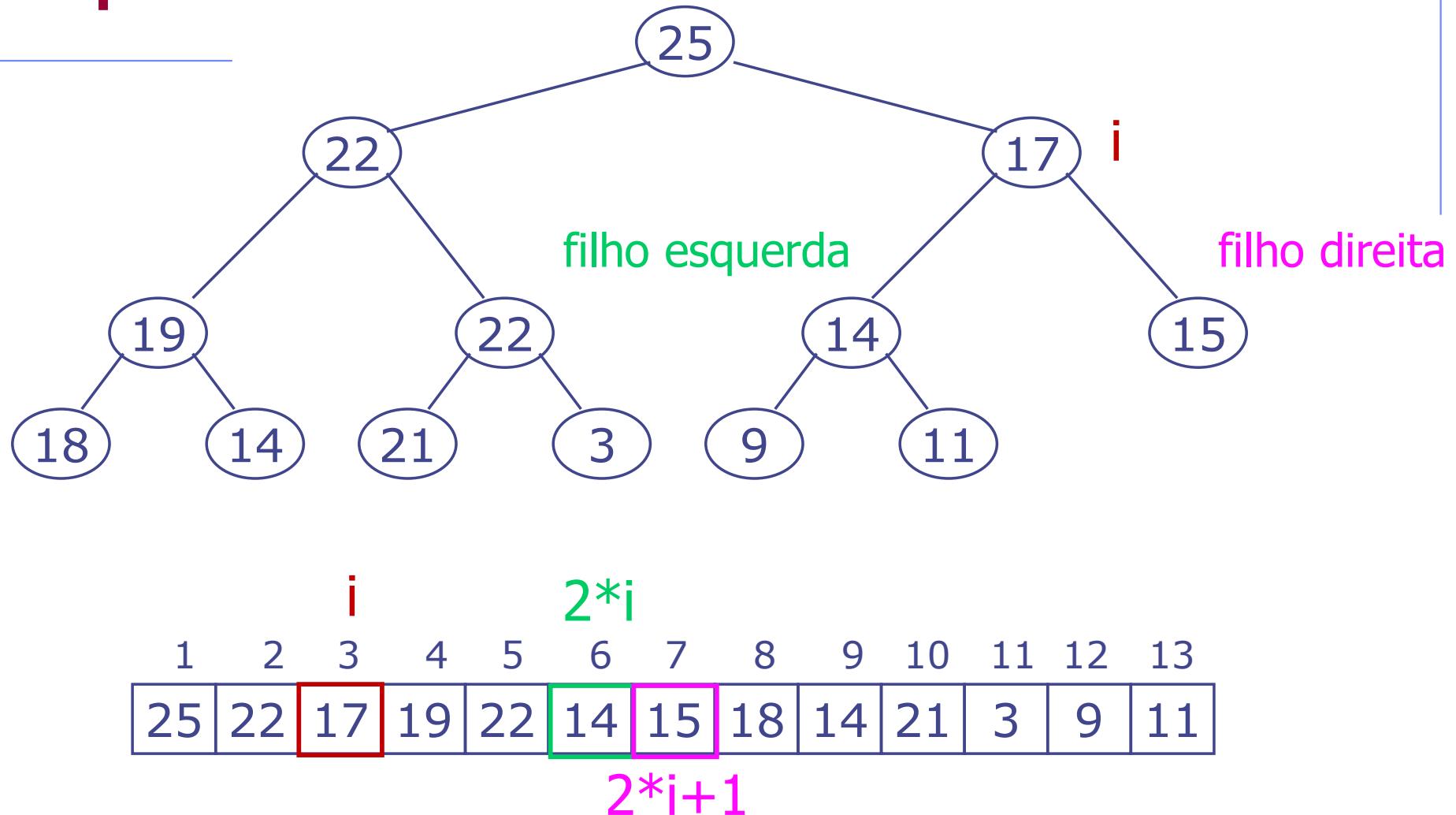


Mapeando em um Vetor



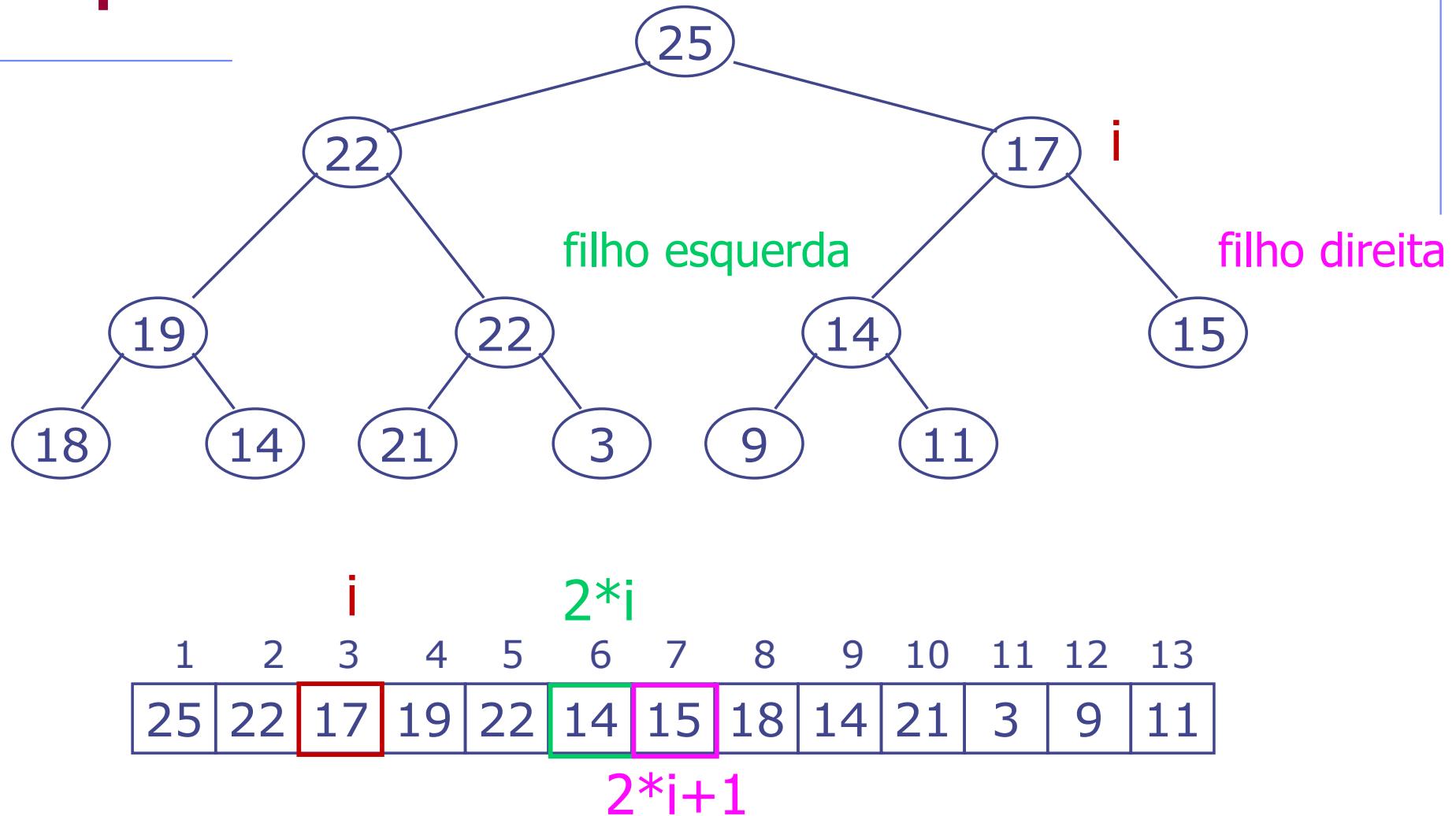
- O filho da esquerda do índice i está no índice $2*i$

Mapeando em um Vetor



- O filho da esquerda do índice i está no índice $2*i$
- O filho da direita do índice i está no índice $2*i+1$

Mapeando em um Vetor



- Exemplo: os filhos do nó 3 (17) são 6 (14) e 7 (15)

A estrutura de dados heap

- ◆ Existem dois tipos de heap
 - heap máximo
 - heap mínimo
- ◆ Em ambos os tipos, os valores nos nós satisfazem uma **propriedade de heap**
 - Em um heap máximo, a **propriedade de heap máximo** é que, para todo nó i diferente da raiz $A[\text{parente}(i)] \geq A[i]$
 - Em um heap mínimo, a **propriedade de heap mínimo** é que, para todo nó i diferente da raiz $A[\text{parente}(i)] \leq A[i]$
- ◆ Visualizando o heap como uma árvore, definimos
 - a **altura** de um nó como o número de arestas no caminho descendente simples mais longo deste nó até uma folha
 - a **altura do heap** como a altura de sua raiz
 - a altura de um heap é $\Theta(\lg n)$, tendo em vista que um heap de n elementos é baseado em uma árvore binária completa

Operações sobre o heap

◆ Algumas operações sobre heap

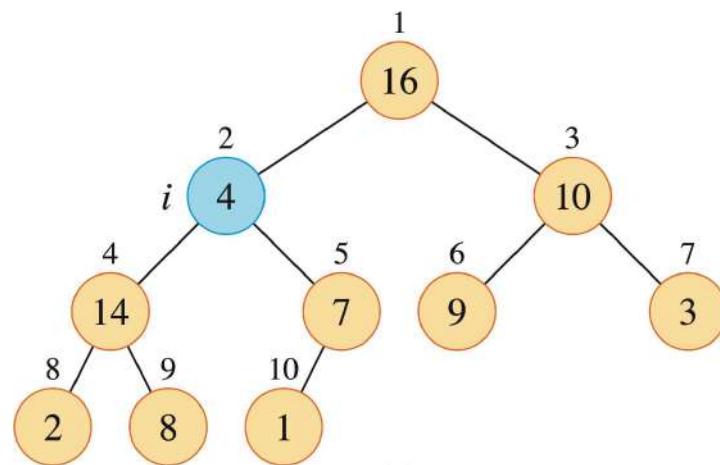
- `max-heapify`, executado no tempo $O(\lg n)$, é a chave para manter a propriedade de heap máximo
- `build-max-heap`, executado em tempo linear, produz um heap a partir de uma array de entrada não ordenado
- `heapsort`, executado no tempo $O(n \lg n)$, ordena um array localmente

Manutenção da propriedade de heap

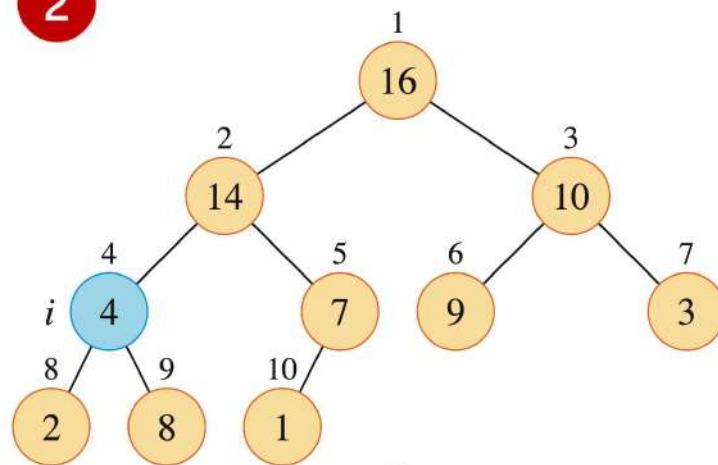
- ◆ A função `max-heapify` recebe como parâmetro um array A e um índice i
- ◆ As árvores binárias com raízes em `left(i)` e `right(i)` são heaps máximos
- ◆ $A[i]$ pode ser menor que seus filhos
- ◆ A função `max-heapify` deixa que o valor $A[i]$ "flutue para baixo", de maneira que a subárvore com raiz no índice i se torne um heap

Exemplo

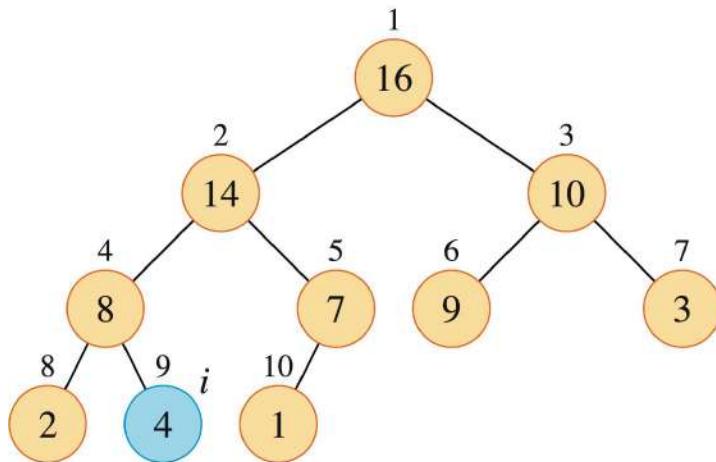
1



2



3



Algoritmo

```
max-heapify(A, i)
1 l = left(i)
2 r = right(i)
3 // seleciona o maior entre, pai, filho-esquerdo e filho-direito
4 if l <= A.tamanho-do-heap e A[l] > A[i] then
5     maior = l
6 else
7     maior = i
8 if r <= A.tamanho-do-heap e A[r] > A[maior] then
9     maior = r
10 if maior != i then
11     troca(A[i], A[maior])
12     max-heapify(A, maior)
```

Análise do max-heapify

- ◆ Tempo $\Theta(1)$ para corrigir os relacionamentos entre os elementos $A[i]$, $A[\text{left}(i)]$ e $A[\text{right}(i)]$
- ◆ Tempo para executar max-heapify em uma subárvore com raiz em um dos filhos do nó i
- ◆ As subárvore de cada filho têm tamanho máximo igual a $2n/3$ – o pior caso ocorre quando a última linha da árvore está exatamente metade cheia
- ◆ O tempo total é $T(n) \leq T(2n/3) + \Theta(1)$
- ◆ Pelo caso 2 do teorema mestre $T(n) = O(\lg n)$

A construção de um heap

- ◆ O procedimento `max-heapify` pode ser usado de baixo para cima para converter um array $A[1..n]$ em um heap máximo
- ◆ Os elementos no subarray $A[(\lfloor i/2 \rfloor + 1) .. n]$ são folhas, e cada um é um heap máximo
- ◆ O procedimento `build-max-heap` percorre os nós restantes da árvore e executa `max-heapify` sobre cada um

`build-max-heap (A)`

```
1 A.tamanho-do-heap = A.comprimento
2 // Para cada nó interno (não folha)
3 for i = piso(A.comprimento / 2) downto 1
4 // chama max-heapify, seguindo o sentido folha-raiz
5     max-heapify(A, i)
```

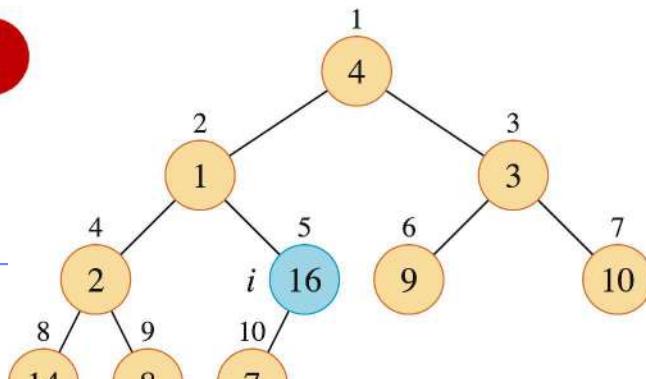
A construção de um heap

Exemplo

- ◆ Construindo um max-heap chamando build-max-heap(A, 10) no seguinte array não ordenado A[1:10] resulta no seguinte exemplo de heap
 - A.tamanho-do-heap é 10
 - i inicia em 5
 - max-heapify é aplicado nas subárvores cujas raízes estão nos nós (em ordem): A[5], A[4], A[3], A[2], A[1]

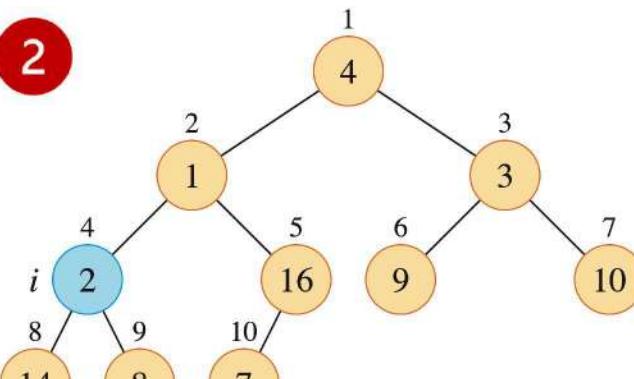
<i>A</i>	4	1	3	2	16	9	10	14	8	7
----------	---	---	---	---	----	---	----	----	---	---

1



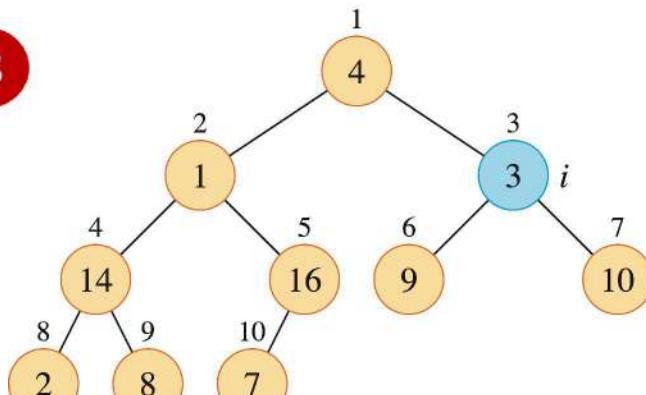
(a)

2



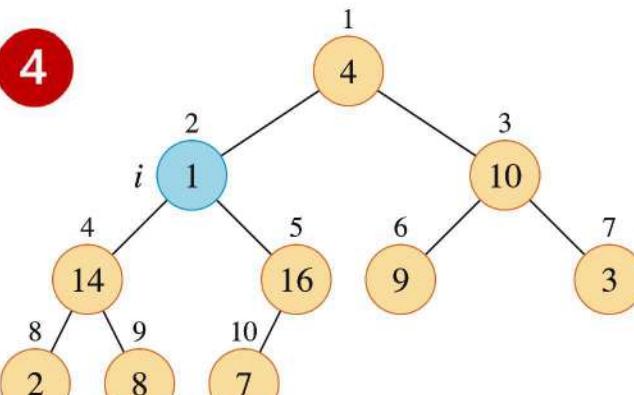
(b)

3



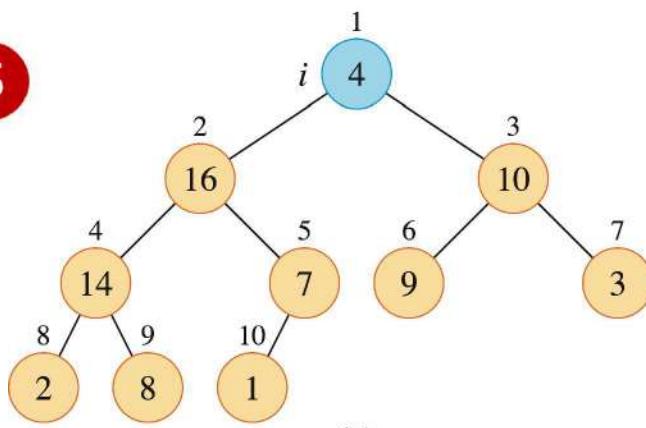
(c)

4



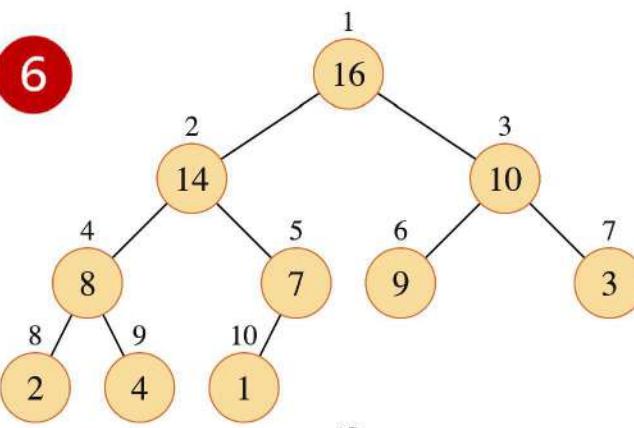
(d)

5



(e)

6



(f)

Análise do build-max-heap

◆ Limite superior simples

- Cada chamada de `max-heapify` custa $O(\lg n)$ e existem $O(n)$ chamadas, portanto, o tempo de execução é $O(n \lg n)$

◆ Limite restrito

- O tempo de execução do `max-heapify` varia com a altura da árvore, a altura da maioria dos nós é pequena
- Pode ser demonstrado que o tempo de execução é $O(n)$

O algoritmo heapsort

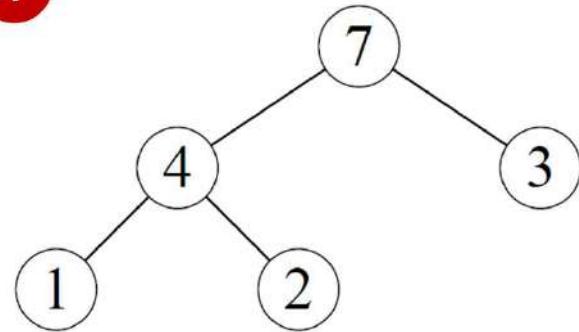
- ◆ Construir um heap, usando a função build-max-heap
- ◆ Trocar o elemento $A[1]$ com $A[n]$, e atualiza o tamanho do heap para $n-1$
- ◆ Corrigir o heap com a função max-heapify e repetir o processo

heapsort(A)

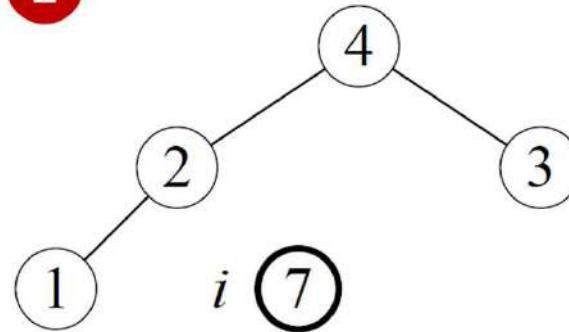
```
1 build-max-heap(A)      // Cria o heap
2 for i = A.comprimento downto 2
3     troca(A[1], A[i]) // Coloca o maior no final do vetor
4     A.tamanho-do-heap = A.tamanho-do-heap - 1
5     // refaz heap, desconsiderando o elemento que foi
6     // para o final até que o vetor esteja ordenado
7     max-heapify(A, 1)
```

Ordenando um heap exemplo

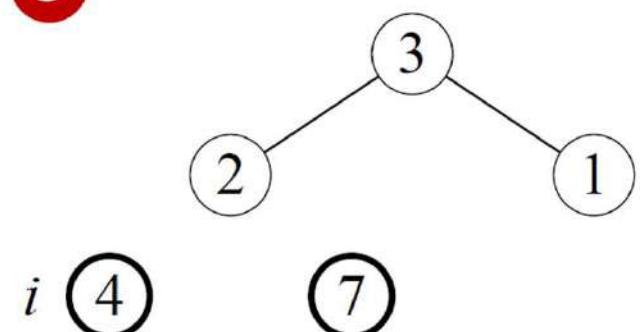
1



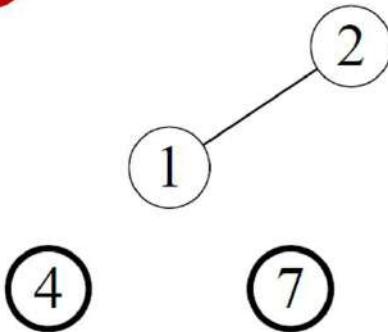
2



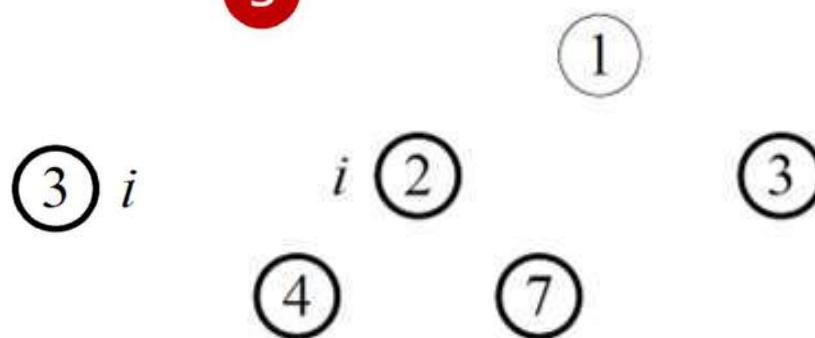
3



4



5



A

1	2	3	4	7
---	---	---	---	---

Nós com linha em negrito não estão mais no heap

Análise do heapsort

- ◆ A chamada `build-max-heap` demora $O(n)$
- ◆ O procedimento `max-heapify` demora $O(\lg n)$ e é chamado $n-1$
- ◆ Portanto, o tempo de execução do heapsort é $O(n \lg n)$

Heapsort

◆ Vantagens

- Comportamento $O(n \log n)$, qualquer que seja a entrada
- É melhor que o ShellSort para grandes arquivos

◆ Desvantagens

- Não é estável
- Não é recomendado para arquivos com poucos registros
 - ◆ devido ao tempo necessário para construir o heap
 - ◆ mais complexo do que o QuickSort

HeapSort

- ◆ Link de videos com exemplos:
 - <https://www.youtube.com/watch?v=P3JaDW72Hdc>
 - <https://www.youtube.com/watch?v=ZbUbCe0WpBE>
- ◆ Video-aula HeapSort Prof. André Backes
 - <https://www.youtube.com/watch?v=zSYOMJ1E52A>

BucketSort

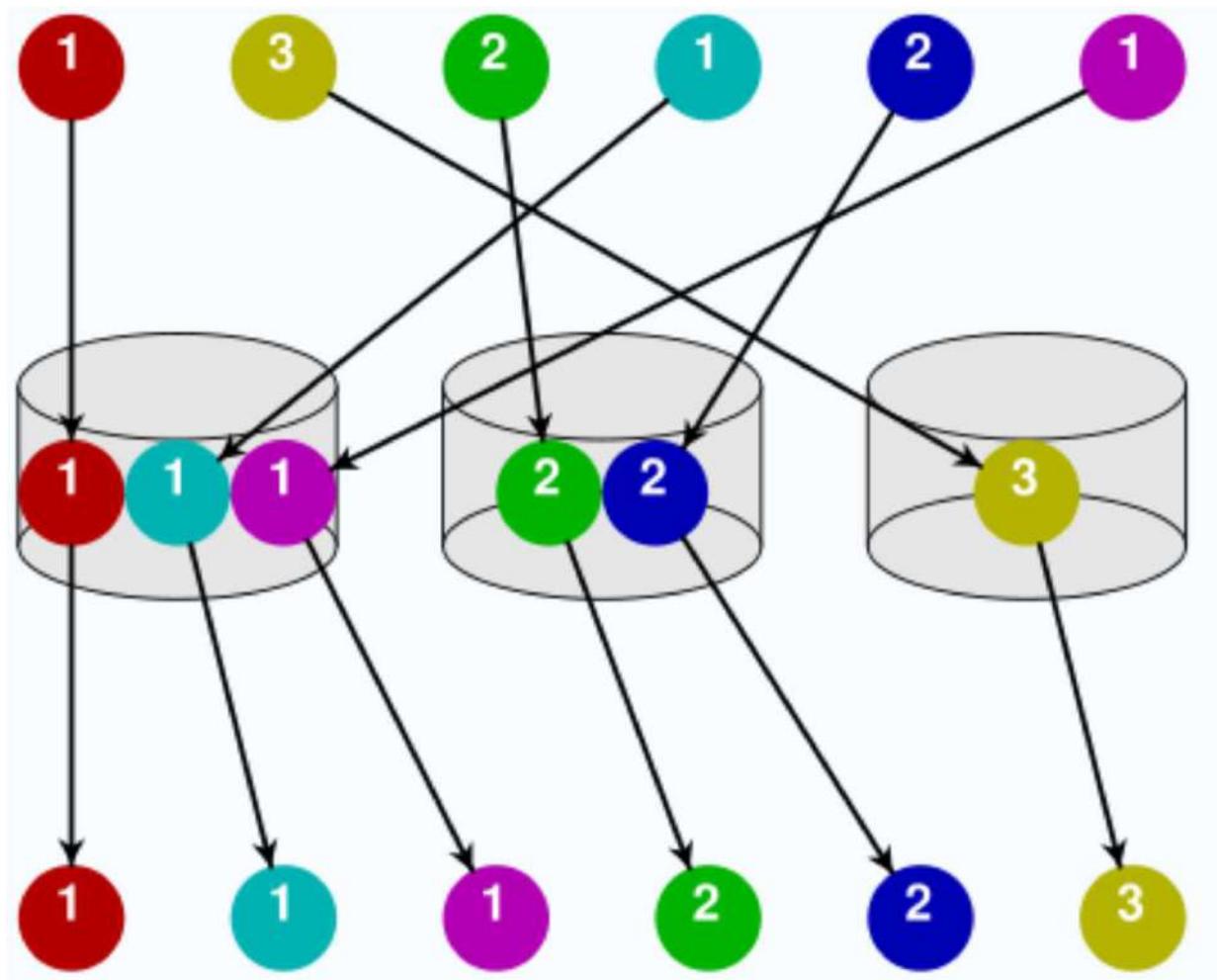
- ◆ BucketSort, ou BinSort, é um algoritmo de ordenação que funciona dividindo um vetor em um número finito de recipientes
- ◆ Cada recipiente é então ordenado individualmente, seja usando um algoritmo de ordenação diferente, ou usando o algoritmo BucketSort recursivamente
- ◆ O BucketSort tem complexidade linear ($O(n)$) quando o vetor a ser ordenado contém valores que são uniformemente distribuídos
- ◆ Ele consegue isso porque não usa comparações
- ◆ No entanto, não é tão genérico

BucketSort

◆ Funcionamento

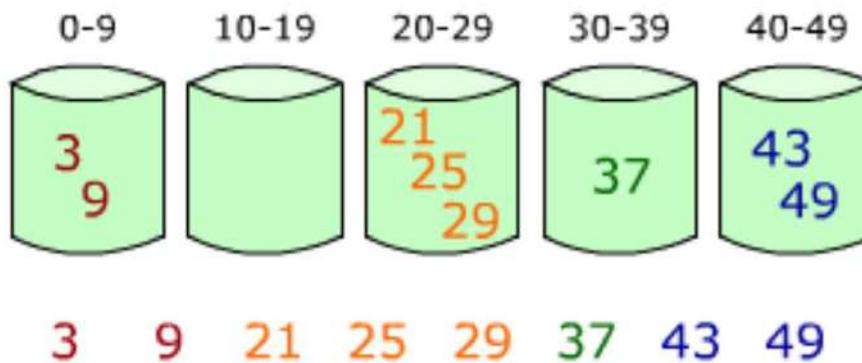
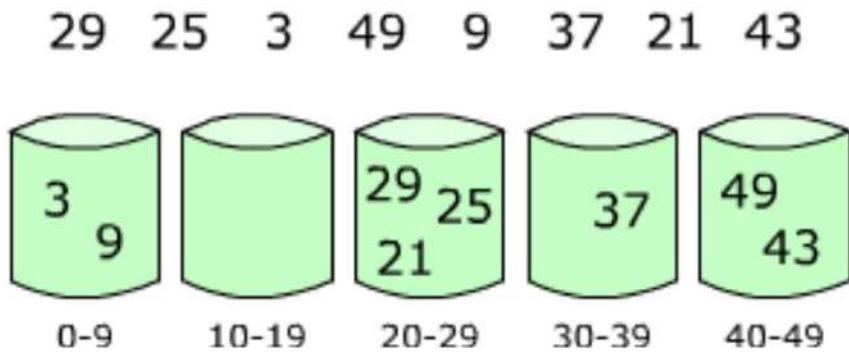
- Inicializar um vetor de "baldes", inicialmente vazios
- Ir para o vetor original, incluindo cada elemento em um balde
- Ordenar todos os baldes não vazios
- Colocar os elementos dos baldes que não estão vazios no vetor original

BucketSort - Visualização



BucketSort

Exemplo estratégia



BucketSort

◆ Algoritmo

```
#define TAM 5 // tamanho do balde
struct balde{
    int qtd;
    int valores[TAM];
};

void bucketSort(int *V, int N){
    int i, j, maior, menor, nroBaldes, pos;
    struct balde *bd;
    //acha maior e menor valor
    maior = menor = V[0];
    for(i = 1; i < N; i++) {
        if(V[i] > maior) maior = V[i];
        if(V[i] < menor) menor = V[i];
    }
    //Inicializa baldes
    nroBaldes = (maior - menor) / TAM + 1;
    bd = (struct balde *) malloc(nroBaldes * sizeof(struct balde));
    for(i = 0; i < nroBaldes; i++)
        bd[i].qtd = 0;
```

BucketSort

◆ Algoritmo

```
// Distribui os valores nos baldes
for(i = 0; i < N; i++) {
    pos = (V[i] - menor) / TAM;
    bd[pos].valores[bd[pos].qtd] = v[i];
    bd[pos].qtd++;
}
// Ordena baldes e coloca no array
pos = 0;
for(i = 0; i < nroBaldes; i++) {
    insertionSort(bd[i].valores, bd[i].qtd);
    for (j = 0; j < bd[i].qtd; j++) {
        V[pos] = bd[i].valores[j];
        pos++;
    }
}
free(bd);
```

BucketSort

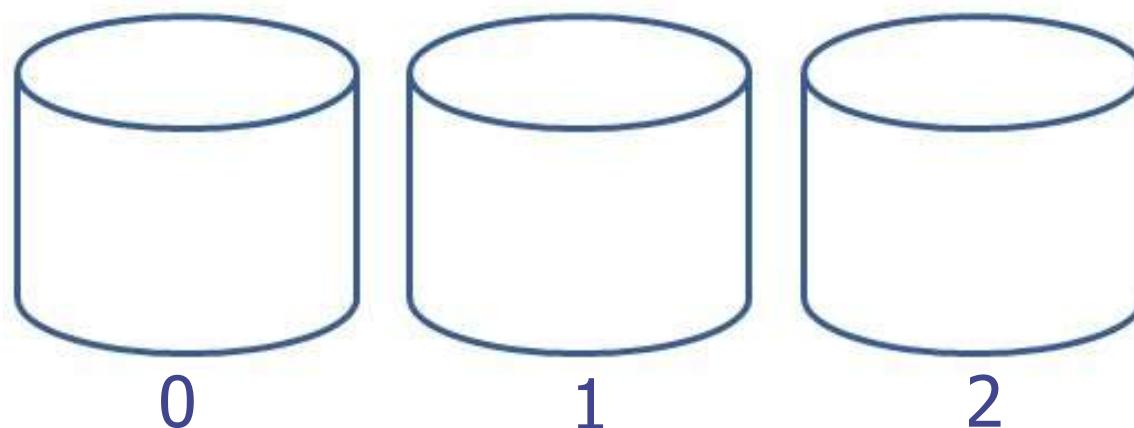
TAM = 9

$$(29 - 4)/TAM + 1 = 3 \text{ baldes}$$

◆ Passo a passo

Sem Ordenar

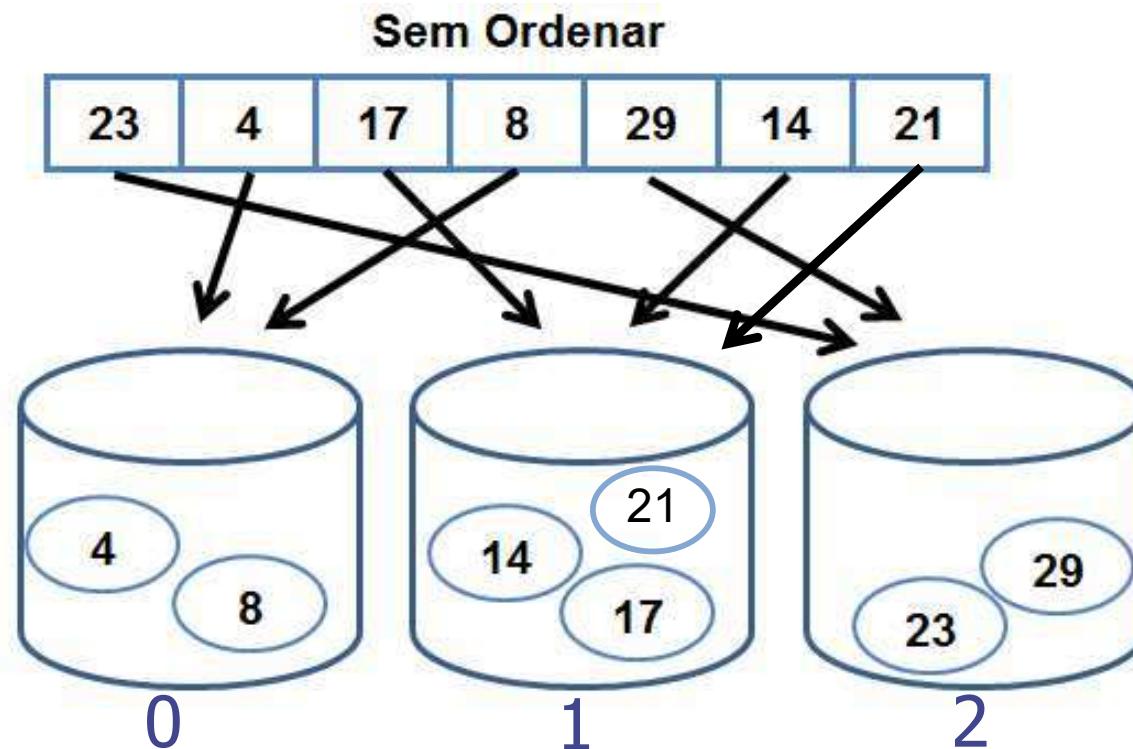
23	4	17	8	29	14	21
----	---	----	---	----	----	----



BucketSort

TAM = 9

◆ Passo a passo

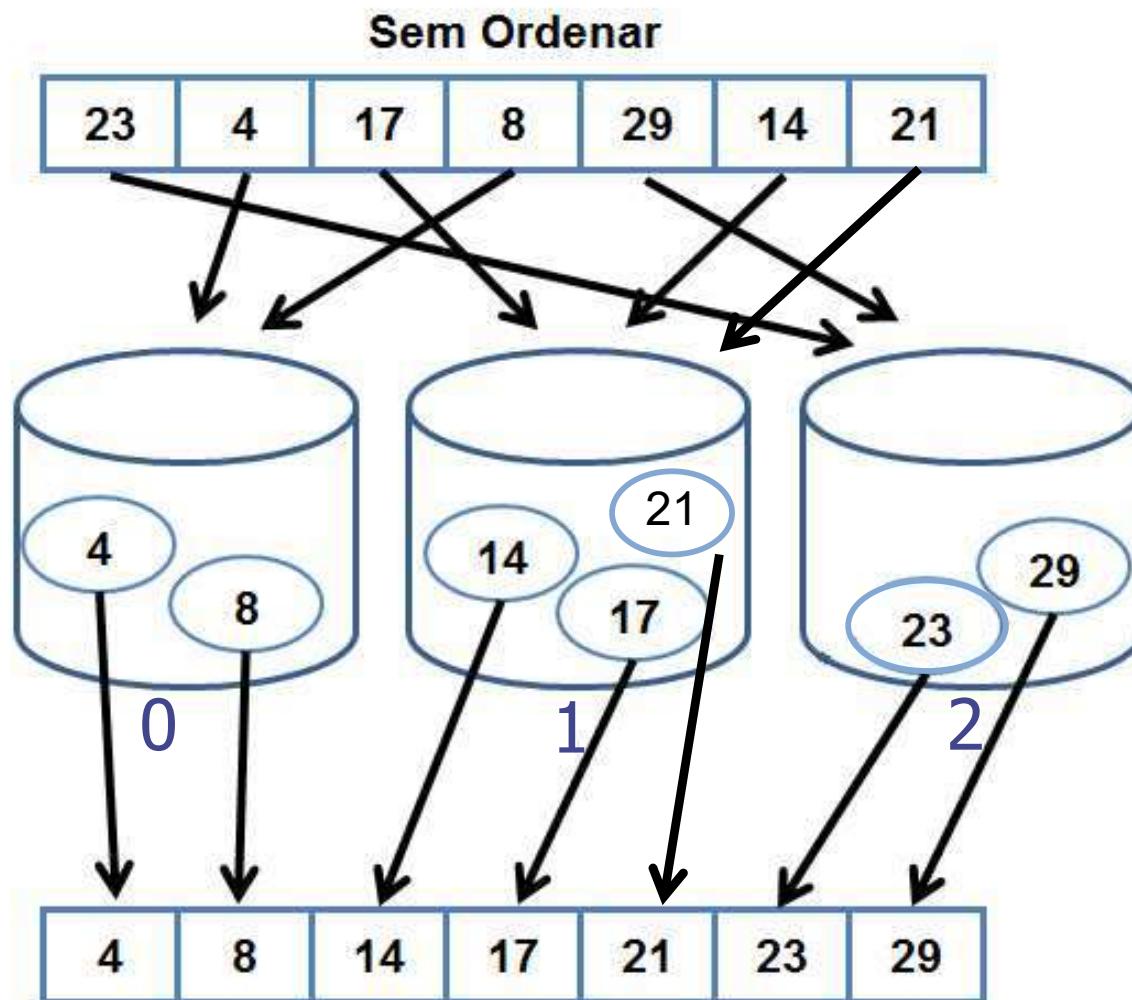


```
// trecho do código  
...  
for(i = 0; i < N; i++)  
    pos = (v[i] - menor) / TAM  
...
```

BucketSort

TAM = 9

◆ Passo a passo



BucketSort

◆ Vantagem

- Estável: não altera a ordem dos dados iguais
 - ◆ Exceto se usar um algoritmo não estável nos baldes
- Processamento simples

◆ Desvantagens

- Dados devem estar uniformemente distribuídos
- Não recomendado para grandes conjuntos de dados
- Ordena valores inteiros positivos (pode ser modificado para outros valores)

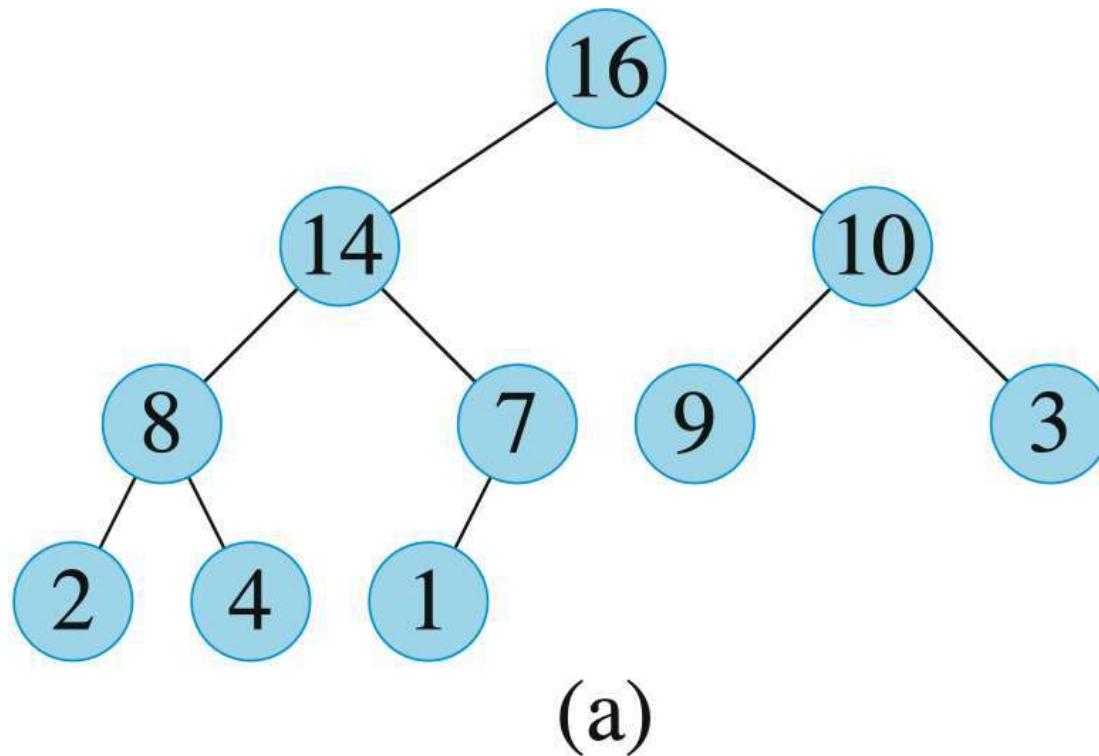
BucketSort

◆ Complexidade

- Considerando um array com **N** elementos e **K** baldes, o tempo de execução é
- **$O(N+K)$** , melhor caso: dados estão uniformemente distribuídos
- **$O(N^2)$** , pior caso: todos os elementos são colocados no mesmo balde

Exercício

- ◆ Ordenar usando o HeapSort a partir do **heap máximo**



(a)

Bibliografia

- ◆ Projeto de Algoritmos, Nivio Ziviani.
- ◆ Estruturas de Dados, Ana Fernanda Ascencio e Graziela Santos de Araújo.
- ◆ Algoritmos: Teoria e Prática, Thomas H. Cormen et al.
- ◆ Prof. André Backes. Slides sobre Algoritmos de Ordenação. Disponíveis em
<https://programacaodescomplicada.wordpress.com/complementar/>

Para estudar

- ◆ Lista de exercícios sobre ordenação disponível na equipe da disciplina no Teams

Prática 3 Ordenação

Profa. Maria Camila Nardini Barioni

camila.barioni@ufu.br

Bloco B - sala 1B137

2º semestre de 2023

Informações gerais

- Esta prática tem por objetivo reforçar o conhecimento do(a) aluno(a) com relação ao algoritmo de ordenação Heapsort estudado em sala

- Data da entrega: 09/02/2024
- Grupo de até 3 (dois) alunos(as). Incluir nome e matrícula como comentário no código
- Linguagem de programação a ser usada: C
- O que deve ser entregue: Link para o repositório do código (Replit.com) e documento .pdf com as respostas das questões descritas na atividade
- Essa prática vale 02 pontos

Exercícios

1. Responda às seguintes questões sobre o método de ordenação heapsort estudado em sala.
 - a) Suponha que todos os elementos de um **heap máximo** são distintos. Onde pode estar o segundo maior elemento? Onde não pode estar o segundo maior elemento? E o terceiro maior? Explique!
 - b) Um vetor em ordem crescente é um **heap máximo**? Um vetor em ordem decrescente é um **heap máximo**? Explique!
 - c) Mostre que o heapsort não é estável.

Exercícios

2. Implemente o algoritmo heapsort discutido em sala de aula para organizar vetores de números inteiros e compare o seu desempenho com o algoritmo shellsort. Para realizar essa comparação, construa uma tabela de referência contendo o tempo obtido para cada método de ordenação, considerando:

- a) três tipos de entrada de dados: aleatória, em ordem ascendente e em ordem descendente
- b) para cada tipo de entrada de dados, use pelo menos três tamanhos de entrada iniciando com vetor de tamanho 10.000.

Deve ser possível escolher o tipo de entrada de dados e o tamanho do vetor por meio de um menu de opções. Na comparação dos métodos, discuta as vantagens e desvantagens observadas. Você pode consultar os slides da aula teórica para apoiar a sua comparação.

Exercícios (esse vale 1 ponto extra)

3. Considere a função `max_heapify()` vista em aula que tem o objetivo de descer corrigindo a partir de uma dada posição. Faça uma função não-recursiva para realizar o mesmo objetivo.

A questão 3 pode ser entregue até 10/02 na tarefa do Teams "Exercício 3 Ordenação – Ponto extra"

Substitua a função `max_heapify()` pela sua versão não recursiva no código do algoritmo heapsort e apresente exemplos de execução no main.



Universidade
Federal de
Uberlândia

UFU – FACOM

MARIA ADRIANA VIDIGAL DE LIMA

QUICKSORT, PARTICIONAMENTO E MODIFICAÇÕES

QUICKSORT – ORDENAÇÃO RÁPIDA

Proposto por Charles A. R. Hoare em 1960 e publicado em 1962.

É o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações.

A ordenação baseia-se em dividir para conquistar. Um conjunto com n itens deve dividir-se em dois conjuntos menores que devem ser ordenados de forma independente.

A ordenação acontece no próprio vetor e o método não garante estabilidade de ordem para valores iguais.

ESTRATÉGIA PARA A ORDENAÇÃO

- Definição de um pivô.
- Colocam-se os elementos menores que o pivô à esquerda.
- Os elementos maiores que o pivô são acomodados à direita.
- O pivô é colocado na sua posição correta (ordenada).
- Os lados esquerdo e o direito são em seguida ordenados independentemente.

ESTRATÉGIA PARA A ORDENAÇÃO

Definição do PIVÔ:

Primeiro elemento do vetor

Ordenação baseada em:

- Escolher como pivô o primeiro elemento
- Particionar o vetor:
 - Trazer para a esquerda os elementos menores que o pivô e empurrar para a direita os maiores
 - Posicionar o pivô ao final dos menores.

```
int particao(int v[],int esq,int dir){  
    int i, pivo;  
  
    pivo = esq;  
    for(i=esq+1; i<=dir; i++)  
        if(v[i] < v[esq]) {  
            pivo = pivo + 1;  
            troca(v,pivo,i);  
        }  
    troca(v,esq,pivo);  
    return pivo;  
}
```

PASSOS DE ORDENAÇÃO

Definição do PIVÔ:

Primeiro elemento do vetor

Ordenação baseada em:

- Escolher como pivô o primeiro elemento
- Particionar o vetor:
 - Trazer para a esquerda os elementos menores que o pivô e empurrar para a direita os maiores
 - Posicionar o pivô ao final dos menores
- Ordenar os menores
- Ordenar os maiores

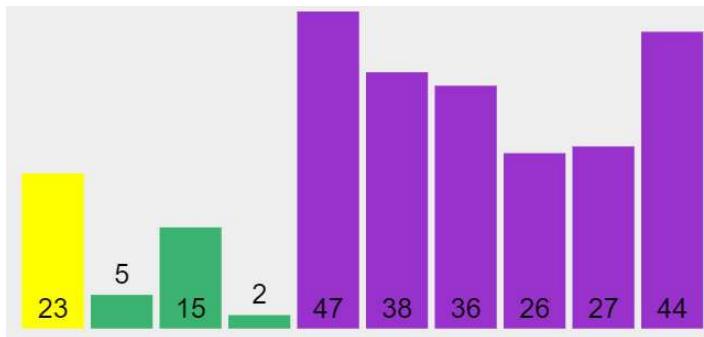
```
int particao(int v[], int esq, int dir){  
    int i, pivo;  
    pivo = esq;  
    for(i=esq+1; i<=dir; i++)  
        if(v[i] < v[esq]) {  
            pivo = pivo + 1;  
            troca(v, pivo, i);  
        }  
    troca(v, esq, pivo);  
    return pivo;  
}
```

```
void quicksort(int v[], int esq, int dir){  
    int i;  
    if(esq>=dir) return;  
    i = particao(v, esq, dir);  
    quicksort(v, esq, i-1);  
    quicksort(v, i+1, dir);  
}
```

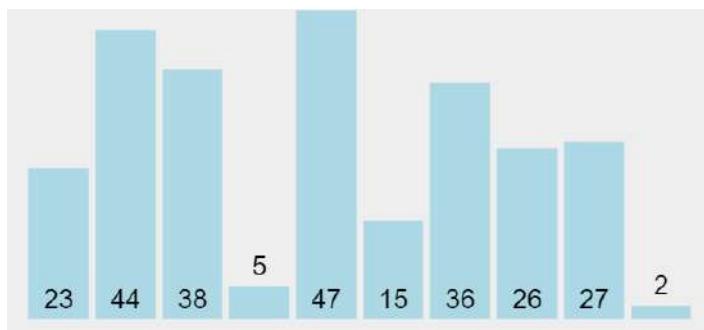


VISUALIZAÇÃO

Vetor inicial:



Pivô - Menores - Maiores

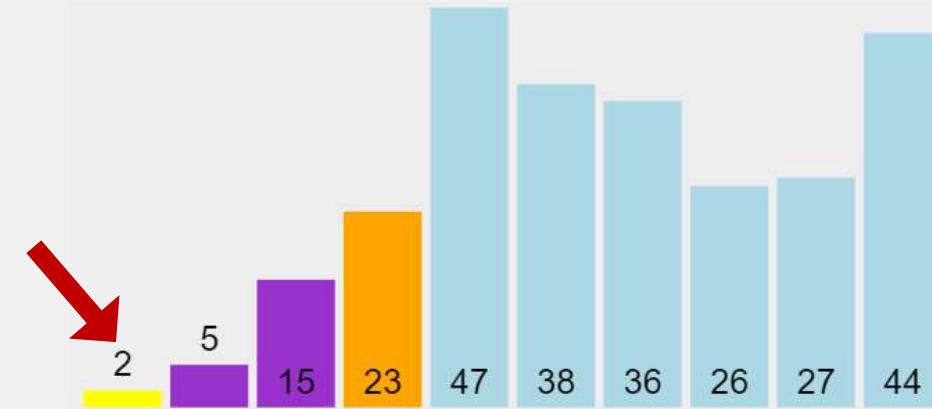


```
int particao(int v[], int esq, int dir){  
    int i, pivo;  
  
    pivo = esq;  
    for(i=esq+1; i<=dir; i++)  
        if(v[i] < v[esq]) {  
            pivo = pivo + 1;  
            troca(v, pivo, i);  
        }  
    troca(v, esq, pivo);  
    return pivo;  
}  
void quicksort(int v[], int esq, int dir){  
    int i;  
    if(esq>=dir) return;  
    i = particao(v, esq, dir);  
    quicksort(v, esq, i-1);  
    quicksort(v, i+1, dir);  
}
```

VISUALIZAÇÃO



Início da ordenação da esquerda: Pivô = 2



Início da ordenação da direita: Pivô = 47



TEMPO DE UMA PARTIÇÃO

A função *Partição* executa em tempo $O(n)$, sendo n o tamanho do vetor.

O laço for realiza uma única varredura no vetor completo para a separação dos menores e maiores.

A função troca tem tempo constante $O(1)$.

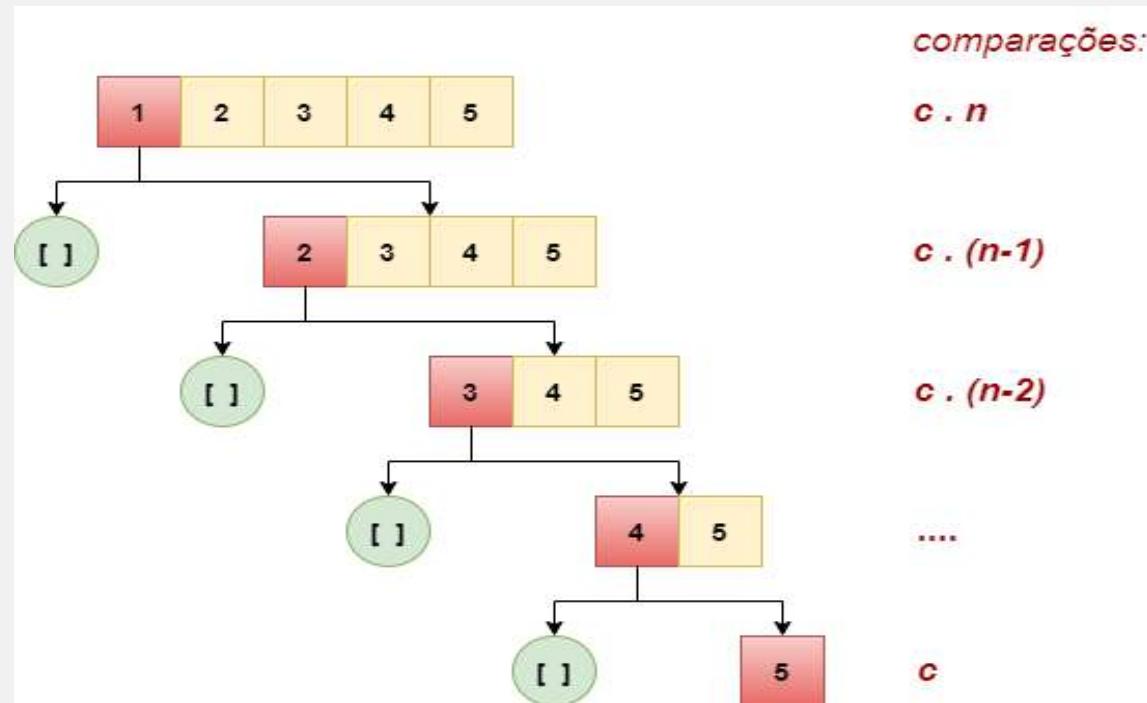
```
int particao(int v[],int esq,int dir){  
    int i, pivo;  
  
    pivo = esq;  
    for(i=esq+1; i<=dir; i++)  
        if(v[i] < v[esq]) {  
            pivo = pivo + 1;  
            troca(v,pivo,i);  
        }  
    troca(v,esq,pivo);  
    return pivo;  
}
```



PIOR CASO DO QUICKSORT

Uma entrada de dados ordenada representa o pior caso para o algoritmo *quicksort* quando a escolha do pivô se dá pelo primeiro elemento.

O tempo de execução do *quicksort* é quadrático neste caso:

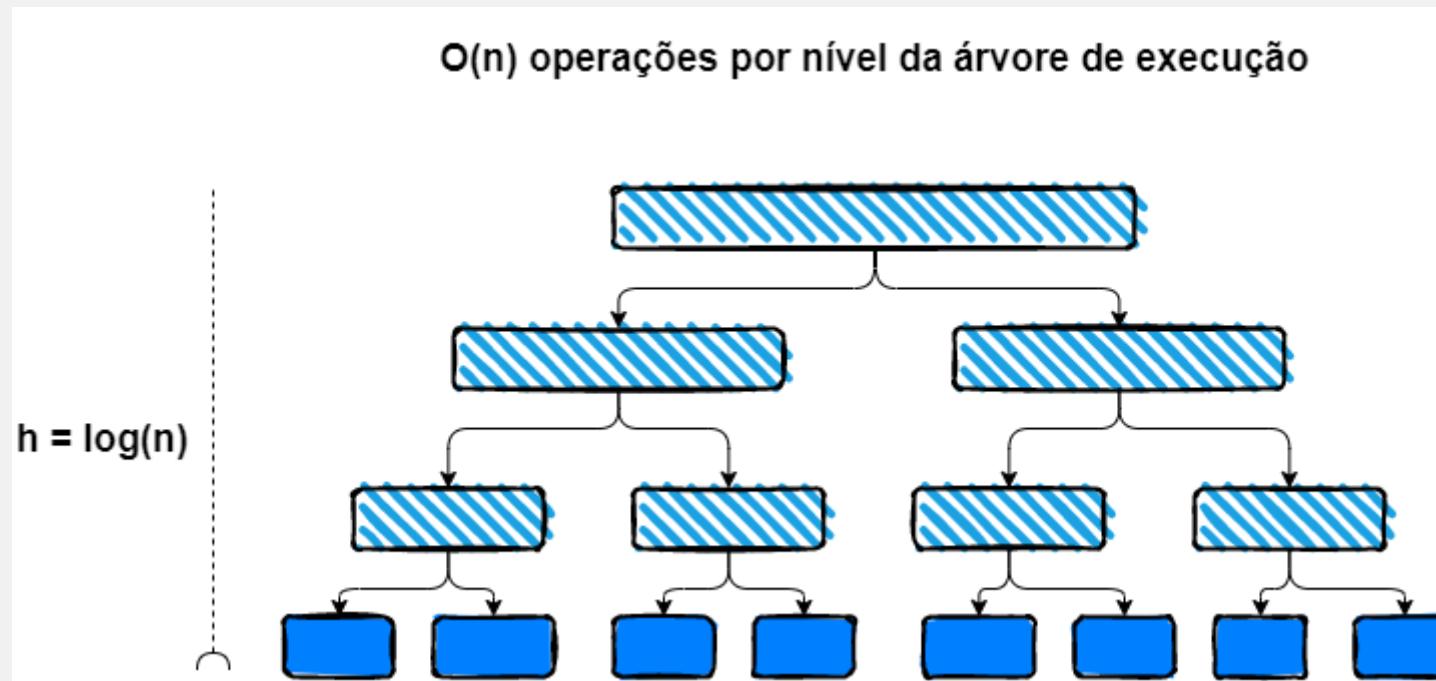


$$c \cdot n + c \cdot (n-1) + \dots + c = c \sum_{i=0}^{n-1} (n-i) = c \sum_{j=1}^n j = c \frac{n(n+1)}{2} = O(n^2)$$



CASO MÉDIO DO QUICKSORT

O tempo médio (esperado) do *quicksort* é $O(n \log n)$ quando o pivô particiona o vetor de forma equilibrada.



$$C(n) = 2 * C(n/2) + n = n \log n = O(n \log n)$$



QUICKSORT RECURSIVO: CARACTERÍSTICAS

Vantagens

Eficiente para ordenar arquivos.

Requer $O(n \log n)$ comparações em média (caso médio) para ordenar n itens.

Desvantagens

Tem um pior caso com $O(n^2)$ comparações.

O método não é estável.



MODIFICAÇÕES NO QUICKSORT

O pior caso pode ser evitado através da realização de pequenas modificações no algoritmo. Algumas opções são:

- Escolha aleatória do pivô
- Escolher três elementos quaisquer e usar a mediana dos três como pivô
- Antes de iniciar a ordenação, aplicar um algoritmo de embaralhamento, como o de Ficher-Yates ($O(n)$):

```
1 | n = tamanho do vetor
2 | para cada i entre n e 2
3 |   sorteie j como um número entre 1 e i
4 |   se i e j forem diferentes, troque os elementos i
      e j entre si
```



ALEATORIZAÇÃO DO PIVÔ

```
int pivo_aleatorio(int esq, int dir) {
    double r;
    r = (double) rand()/RAND_MAX;    // valor entre 0.01 e 0.99
    return (int)(esq + r*(dir-esq));
}
/* A função rand() gera um número pseudo-aleatório entre 0 e a
constante RAND_MAX. A constante RAND_MAX é 32762. */

void quicksort_aleatorizado(int *v, int esq, int dir) {
    int i;
    if (dir <= esq) return;
    troca(v, pivo_aleatorio(esq,dir), esq);
    i = particao(v, esq, dir);
    quicksort_aleatorizado(v, esq, i-1);
    quicksort_aleatorizado(v, i+1, dir);
}
```

O tempo de execução depende dos pivôs sorteados.

O tempo médio é $O(n \log n)$ – podendo ser rápido ou lento, mas não depende de como os elementos estão organizados no vetor.



MEDIANA DE TRÊS

Mediana:

Se o conjunto de informações for numérico e estiver organizado em ordem crescente ou decrescente, a sua **mediana** será o número que ocupa a posição central da lista.

Em lugar de fixar como pivô o elemento da esquerda, pode-se escolher o **elemento médio** de uma amostra de três elementos.
Por exemplo: o da esquerda, o do meio, e o da direita.

3 4 9 6 5 1 8

3 6 8

6 será escolhido como pivô



MEDIANA DE TRÊS - IMPLEMENTAÇÃO

1. Recuperar os elementos primeiro, do meio e último
2. Trocar o elemento do meio com o segundo elemento
3. Escolher como pivô a mediana entre os elementos: primeiro, segundo e último
4. O **pivô deve ser colocado na segunda posição**, o menor na primeira e maior deles ao final
5. Esta estratégia permite entrar na partição sem considerar os elementos maior e menor que o pivô.

3 4 9 6 5 1 8

3 6 9 4 5 1 8

6 será escolhido como pivô

3 6 9 4 5 1 8



Trecho pronto para iniciar a partição



MEDIANA DE TRÊS - IMPLEMENTAÇÃO

```
void quicksort_mediana_tres(int v[], int esq, int dir) {  
    int i;  
    if(dir <= esq) return;  
    troca(v, (esq+dir)/2, esq+1);  
    if(v[esq] > v[esq+1])  
        troca(v, esq, esq+1);  
    if(v[esq] > v[dir])  
        troca(v, esq, dir);  
    if(v[esq+1] > v[dir])  
        troca(v, esq+1, dir);  
    i = particao(v, esq+1, dir-1);  
    quicksort_mediana_tres(v, esq, i-1);  
    quicksort_mediana_tres(v, i+1, dir);  
}
```

3 4 9 6 5 1 8
3 6 9 4 5 1 8
3 6 9 4 5 1 8

$i = \text{particao}(v, 1, 5)$



MEDIANA DE TRÊS - RESUMO



O particionamento mediana-de-três consiste em selecionar três elementos como pré-pivôs, sejam estes por exemplo, os três da esquerda, os três da direita ou os três do centro, ou como mostrado anteriormente, esquerda-meio-direita.



Seleciona-se então a mediana desses três elementos e então esse elemento será eleito o pivô, e portanto, acomodado no início do vetor (ou trecho) a ser particionado.



Esta estratégia diminui a probabilidade de que o pivô seja o maior ou o menor elemento da sequência.



PIVÔ: ELEMENTO FUNDAMENTAL

- A escolha de um pivô adequado é uma atividade crítica para o bom funcionamento do *quicksort*. Se pudermos garantir que o pivô está próximo à mediana dos valores do vetor, então o quicksort é muito eficiente.
- Uma técnica que pode ser utilizada para aumentar a chance de encontrar bons pivôs é escolher aleatoriamente três elementos do vetor e usar a mediana desses três valores como pivô para a partição.
- Em sequências com muitos elementos repetidos, ainda é grande a chance de não encontrarmos bons pivôs aleatoriamente ou com a ajuda da mediana de três.



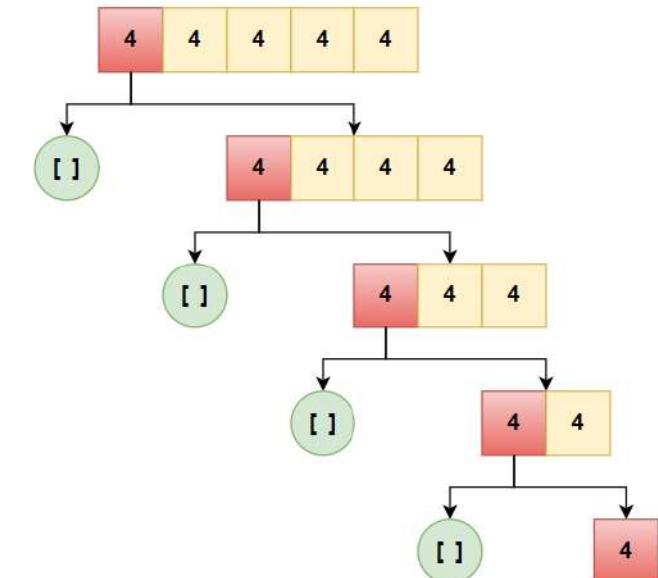
PROBLEMA DAS SEQUÊNCIAS COM MUITAS REPETIÇÕES

Seja o vetor $V = [4, 4, 4, 4, 4, 4, \dots, 4]$ contendo n elementos com valores iguais.

A função *partição*, mesmo precedida de embaralhamento, escolha prévia do pivô com mediana de três ou aleatória cria as seguintes partições:

- Uma vazia
- Outra com $n-1$ elementos

Ainda, cada chamada subsequente da função *partição* terá o mesmo comportamento. Este é o pior cenário para a função *quicksort*!



ESTRATÉGIA PARA SE ADEQUAR ÀS SEQUENCIAS COM MUITAS REPETIÇÕES

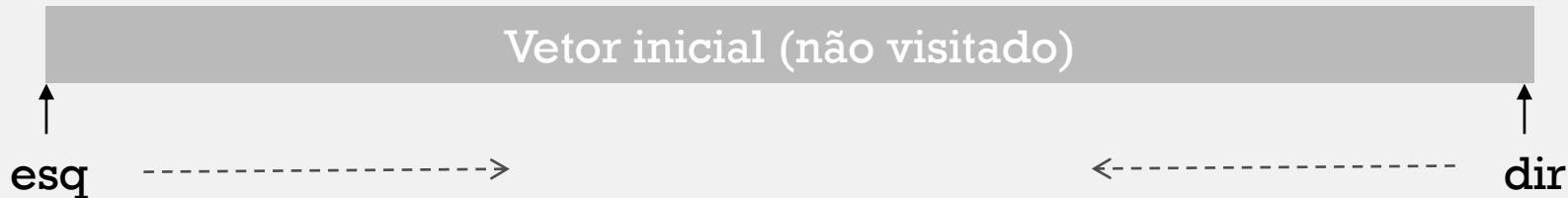
Jon Bentley e Douglas McIlroy co-produziram (em 1993) uma versão otimizada do *quicksort* para tratar entradas com muitos elementos repetidos utilizando um **particionamento em três vias**:



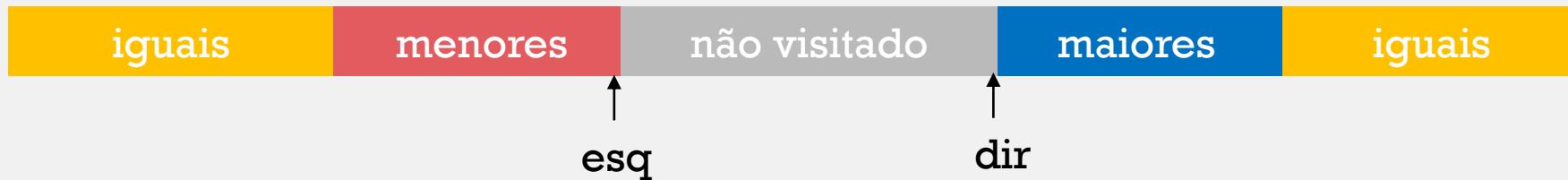
O objetivo é que em uma varredura, tenhamos um valor de particionamento (pivô) e que tanto o pivô quanto os elementos iguais a ele sejam corretamente posicionados (ao meio) enquanto os menores ficam à esquerda e os maiores à direita.



PROCESSO DE PARTICIONAMENTO EM 3 VIAS



A construção do particionamento se inicia com dois ponteiros: um em cada extremidade, e o caminhamento é feito em direção ao meio buscando separar os menores, maiores e iguais ao elemento pivô.

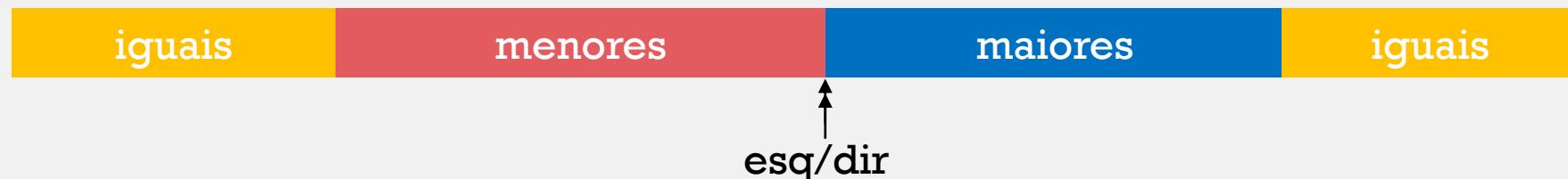


- Nos dois extremos estão as regiões que mantêm os elementos iguais ao pivô.
- A região ainda não visitada fica no meio e vai diminuindo a cada iteração.
- À esquerda dos não visitados estão os menores que o pivô.
- Ao lado direito dos não visitados está a região que mantém os elementos maiores que o pivô.



PROCESSO DE PARTICIONAMENTO EM 3 VIAS

Ao final da primeira visita completa, tem-se:



Em seguida, todos os elementos iguais, acomodados nas extremidades, são movidos para o centro (apontado por esq/dir):



Ao encerrar a visita e organizar as três vias (ou partições), a mesma abordagem será utilizada para particionar as regiões dos menores e maiores.



IMPLEMENTAÇÃO DA VISITA E CONSTRUÇÃO DAS TRÊS VIAS

- Definir pivô: o pivô será o elemento mais à direita.
- Apontar para a esquerda e caminhar até achar um elemento que não seja menor que o pivô
- Apontar para a direita-1 (não considerar o pivô) e caminhar até achar um elemento que não seja maior que o pivô.
- Parar se os ponteiros se cruzarem.
- Trocar os elementos nas posições de parada.
- Se os elementos trocados forem iguais ao pivô, enviá-los para as extremidades correspondentes.

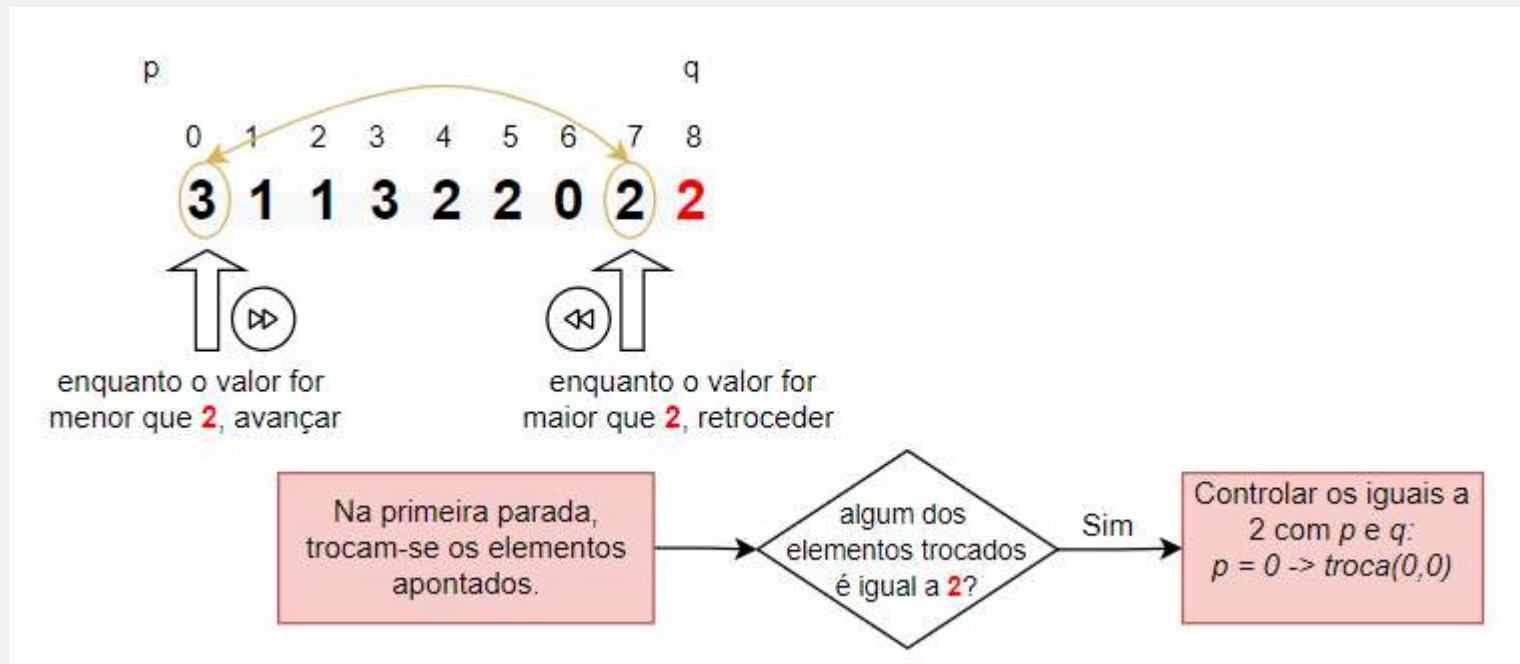


EXEMPLO:

Sequência Inicial: 3 1 1 3 2 2 0 2 2

Pivô : 2 (elemento mais à direita)

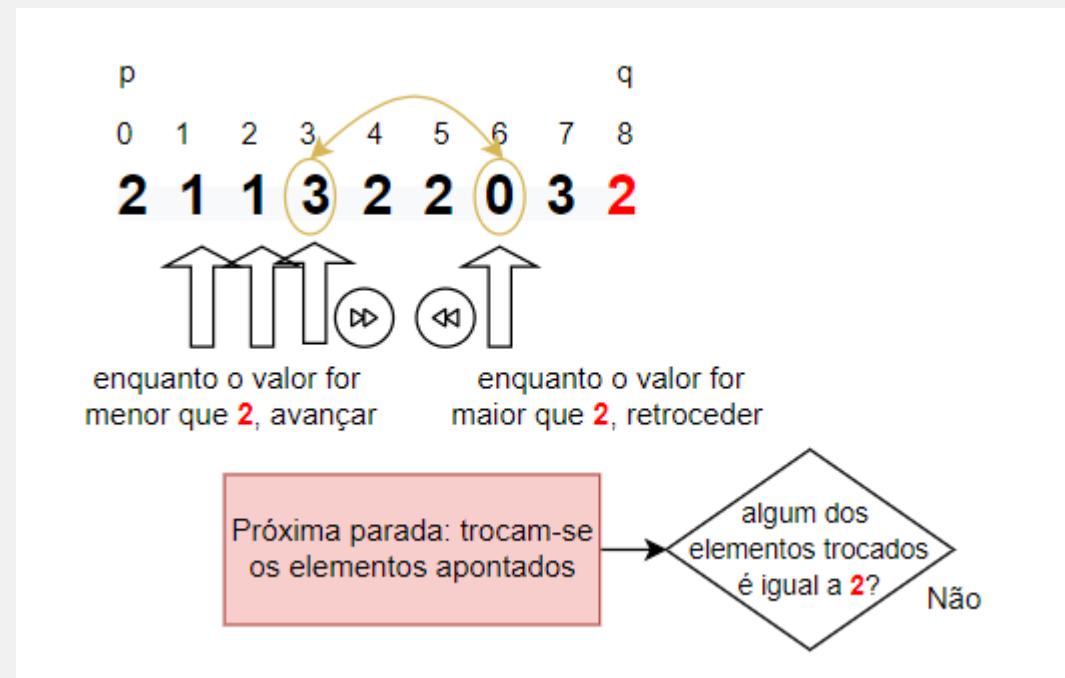
Início do percurso: primeira iteração



Resultado: 2 1 1 3 2 2 0 3 2

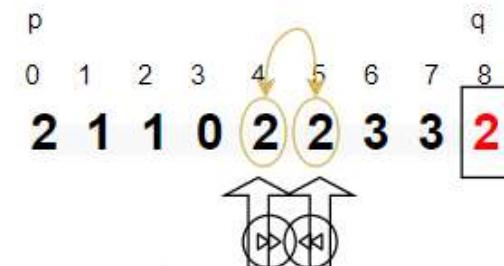
EXEMPLO:

Segunda iteração:

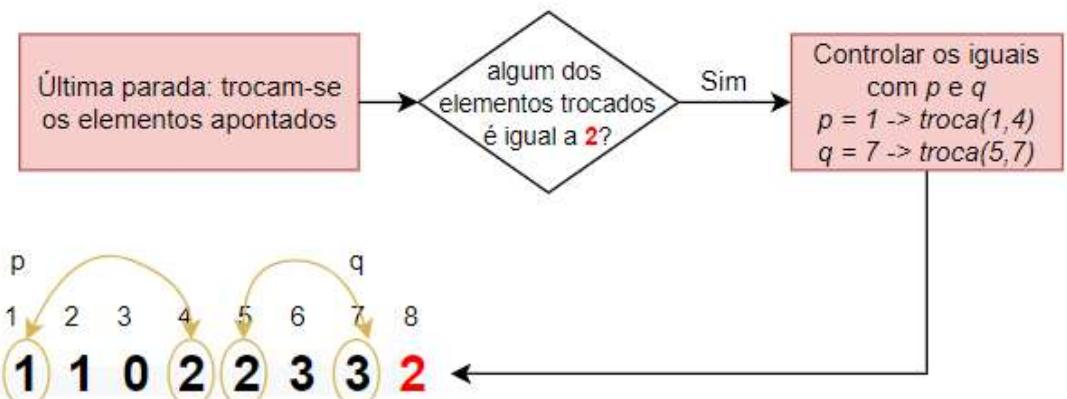


Resultado: 2 1 1 0 2 2 3 3 2

Terceira iteração:



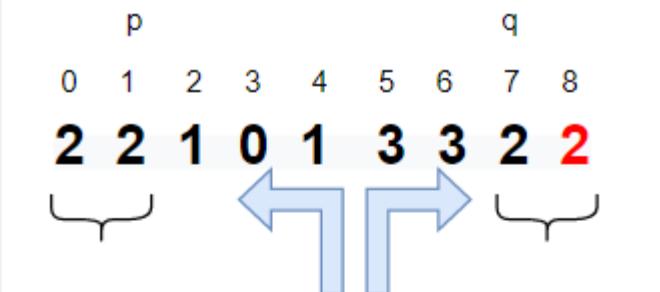
enquanto o valor for menor que 2, avançar



Resultado: 2 2 1 0 1 3 3 2 2

EXEMPLO:

Final do
percurso e
movimentação
do pivô



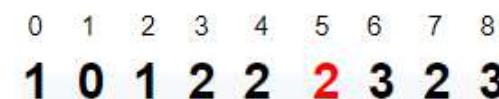
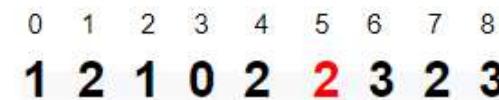
setas cruzadas:
iguais nas pontas
menores à esquerda
maiores à direita

Neste momento, trocam-se
as posições 5 e 8
(início dos maiores e pivô)



Movimentação de todos os iguais ao pivô
para o centro:

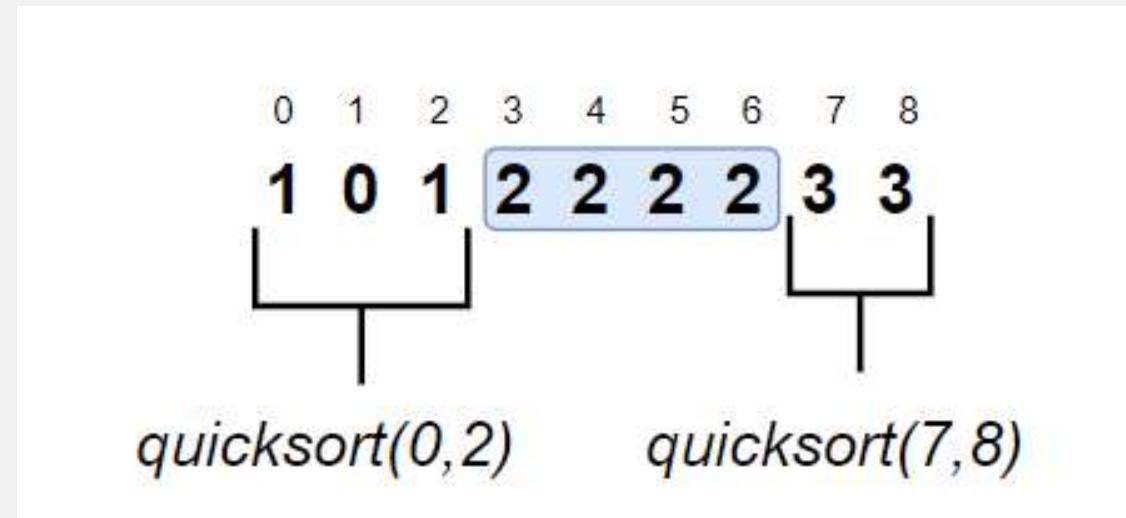
Em seguida, as extremidades
iniciando em 0 e 7 devem ser
trocadas para os meios



EXEMPLO:

Pivô e seus iguais já acomodados nas posições corretas para a primeira varredura ($\text{pivô} = 2$).

Chamadas recursivas na sequência para partições esquerda e direita.



IMPLEMENTAÇÃO:

Linha 5: Definição do pivô

Linhas 7 a 16: Varredura do vetor usando ponteiros para iniciar da esquerda (i) e da direita (j).

Linhas 14 e 15: Verifica se o elemento trocado é igual ao pivô. Se sim, levar para um dos extremos.

Linhas 20 e 21: Colocam os iguais ao pivô no meio.

```
3 void quicksort_tres_partes(int a[], int l, int r){  
4     int k, i = l-1, j = r, p = l-1, q = r;  
5     int v = a[r];  
6     if (r <= l) return;  
7     for (;;) {  
8         while (a[++i] < v);  
9         while (v < a[--j])  
10            if (j == l) break;  
11            if (i >= j) break;  
12            troca(a,i,j);  
13            if (a[i] == v) { p++; troca(a,p,i); }  
14            if (v == a[j]) { q--; troca(a,j,q); }  
15        }  
16        troca(a,i,r);  
17        j = i-1;  
18        i = i+1;  
19        for (k = l; k <= p; k++, j--) troca(a,k,j);  
20        for (k = r-1; k >= q; k--, i++) troca(a,i,k);  
21        quicksort_tres_partes(a, l, j);  
22        quicksort_tres_partes(a, i, r);  
23    }  
24 }
```

COMPARAÇÃO ENTRE ALGORITMOS: TESTES EXPERIMENTAIS

Implementar os algoritmos

- *Quicksort* básico
- *Quicksort* aleatorizado
- *Quicksort* com mediana de três
- *Quicksort* com partição em três vias

Criar quatro vetores:

- Sequência aleatória
- Sequência ordenada
- Sequência invertida
- Sequência com muitas repetições

Tamanho do vetor: 5000

	Aleatorio	Ordenado	Invertido	Repetidos
Básico	0,00094	0,04559	0,09331	0,00169
Aleatorizado	0,00088	0,00107	0,00088	0,00167
Mediana de 3	0,00053	0,00051	0,00052	0,00080
Partição em 3 vias	0,02063	0,02815	0,02658	0,00039

Tamanho do vetor: 50000

	Aleatorio	Ordenado	Invertido	Repetidos
Básico	0,01130	3,92205	7,97799	0,09105
Aleatorizado	0,01034	0,01066	0,01109	0,10022
Mediana de 3	0,00703	0,00640	0,00703	0,04711
Partição em 3 vias	0,51811	2,51951	2,48866	0,00364



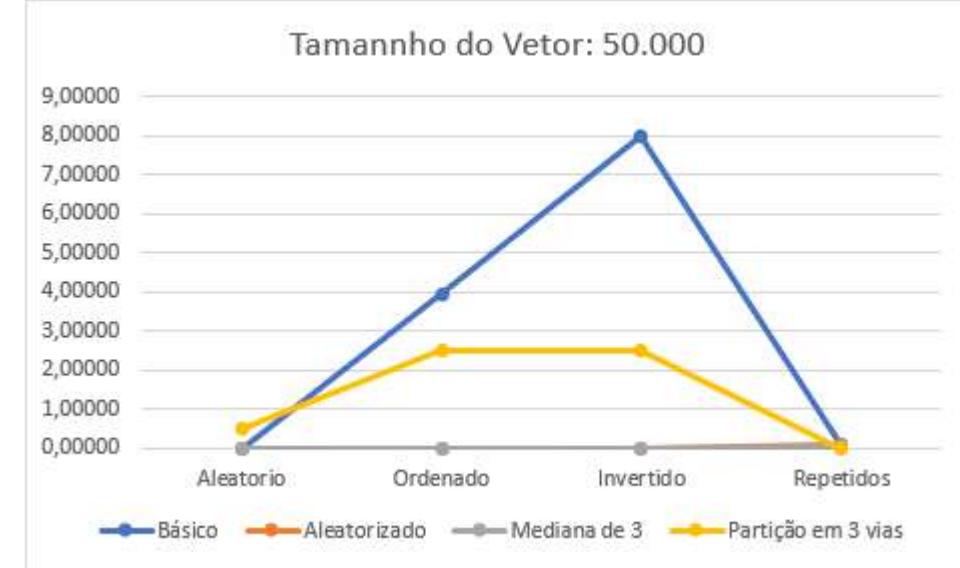
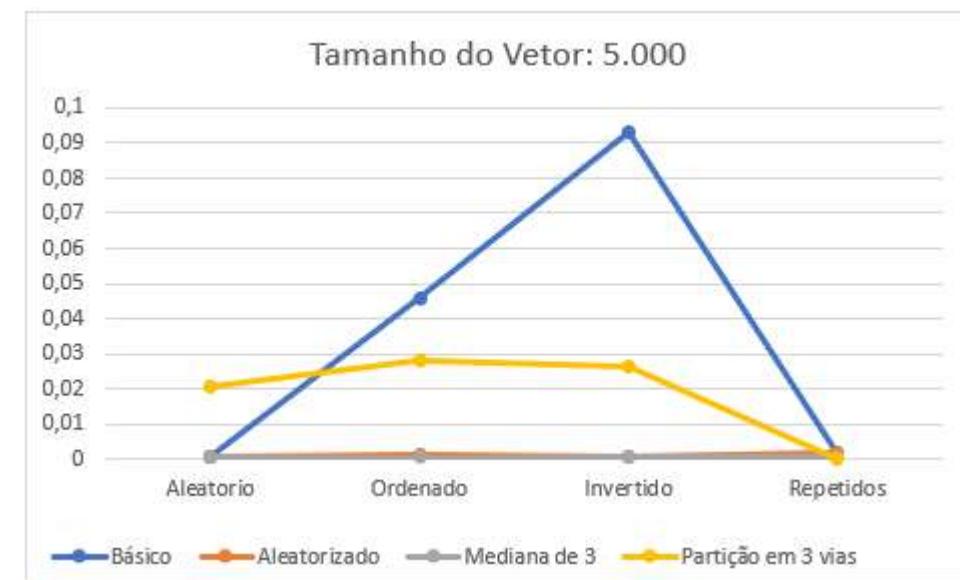
TESTES EXPERIMENTAIS

Tamanho do vetor: 5000

	Aleatorio	Ordenado	Invertido	Repetidos
Básico	0,00094	0,04559	0,09331	0,00169
Aleatorizado	0,00088	0,00107	0,00088	0,00167
Mediana de 3	0,00053	0,00051	0,00052	0,00080
Partição em 3 vias	0,02063	0,02815	0,02658	0,00039

Tamanho do vetor: 50000

	Aleatorio	Ordenado	Invertido	Repetidos
Básico	0,01130	3,92205	7,97799	0,09105
Aleatorizado	0,01034	0,01066	0,01109	0,10022
Mediana de 3	0,00703	0,00640	0,00703	0,04711
Partição em 3 vias	0,51811	2,51951	2,48866	0,00364



Bacharelado em Ciência da Computação
GBC034 Algoritmos e Estruturas de Dados 2

Grafos

Conceitos básicos

Material baseado nos slides do professor Nivio Ziviani
(Projeto de Algoritmos)

Profa. Maria Camila Nardini Barioni

camila.barioni@ufu.br

Bloco B - sala 1B137

2º semestre de 2023

Roteiro

- 
- ◆ Motivação
 - ◆ Aplicações
 - ◆ Conceitos Básicos

Por que estudar Grafos?

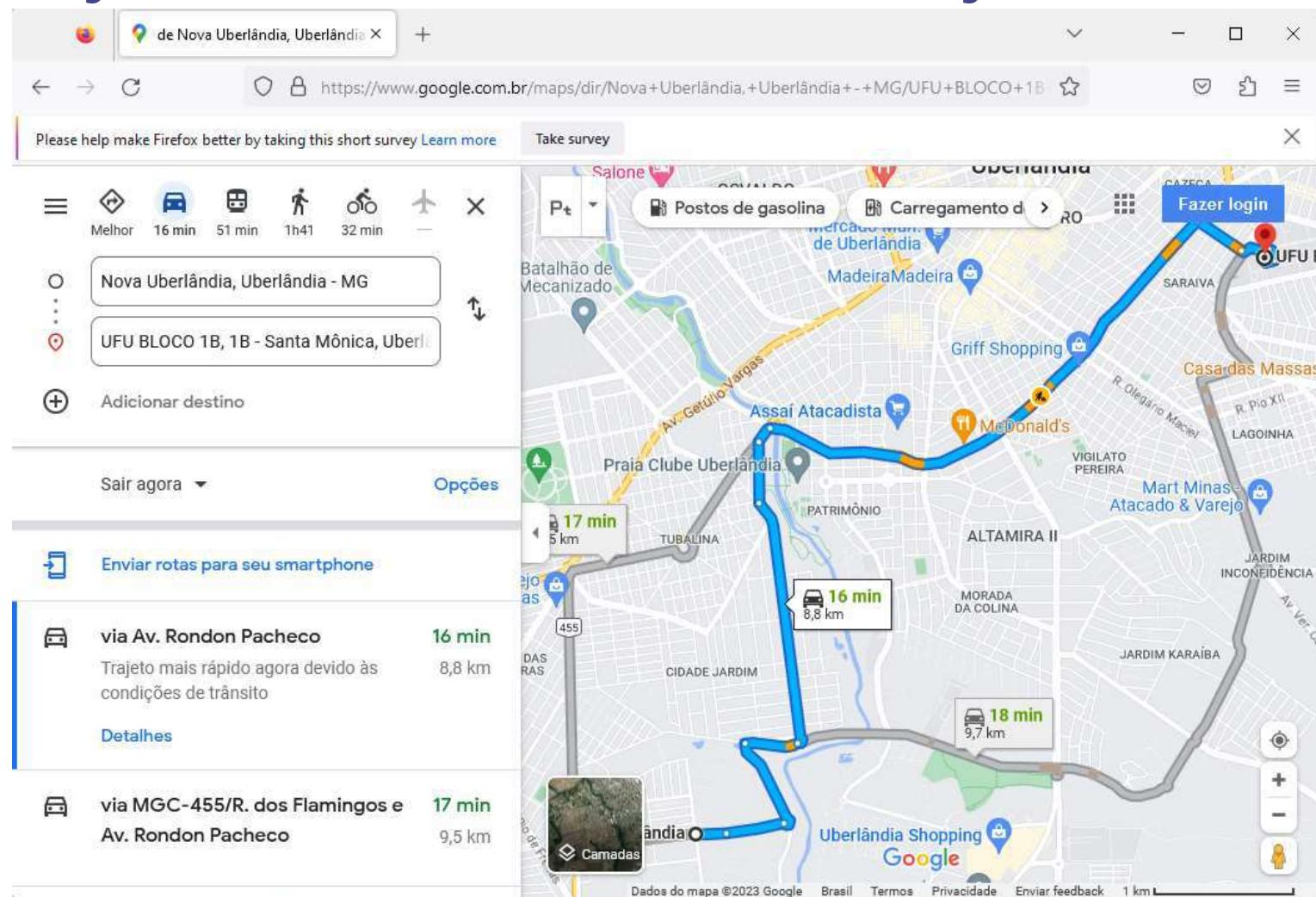
- ◆ Muitas aplicações em computação necessitam considerar conjunto de conexões entre pares de objetos:
 - Existe um caminho para ir de um objeto a outro seguindo as conexões?
 - Qual é a menor distância entre um objeto e outro objeto?
 - Quantos outros objetos podem ser alcançados a partir de um determinado objeto?
- ◆ Existe um tipo abstrato chamado grafo que é usado para modelar tais situações

Aplicações

- ◆ Alguns exemplos de problemas práticos que podem ser resolvidos através de uma modelagem em grafos

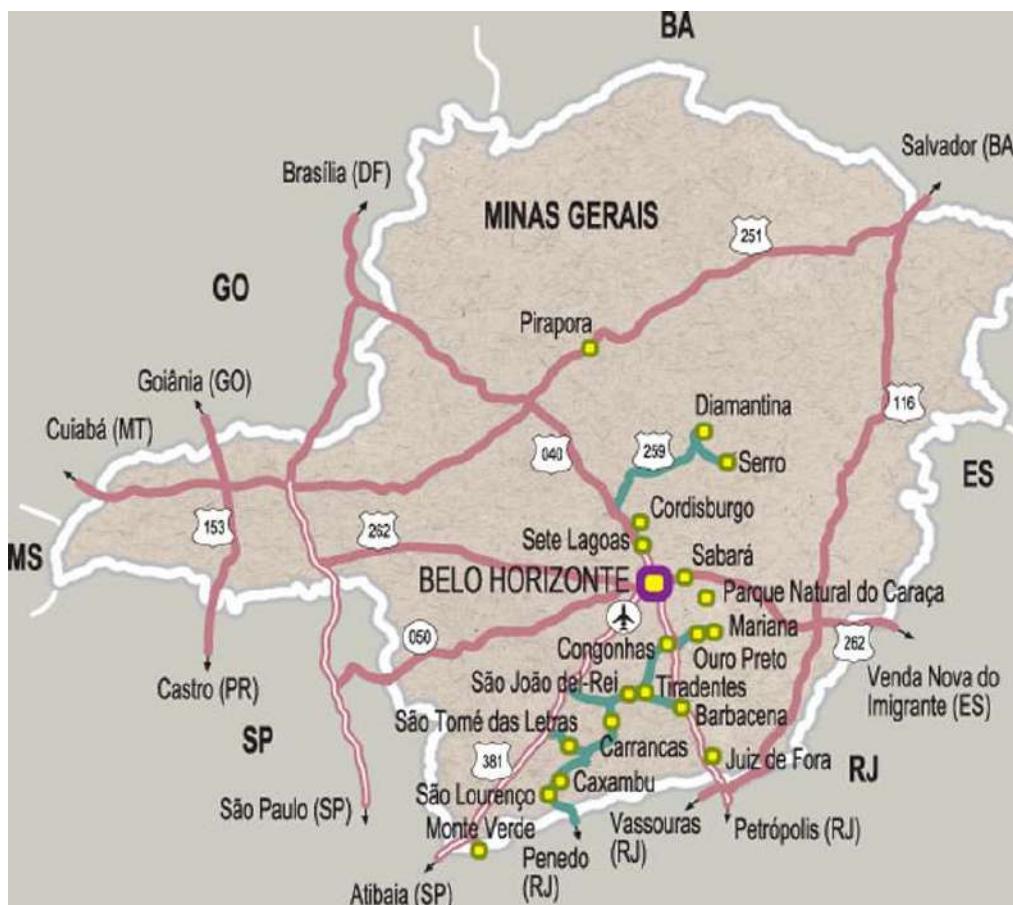
Aplicações

◆ Traçar rotas entre duas localizações



Aplicações

- ◆ Descobrir qual é o roteiro mais curto para visitar as principais cidades de uma região turística

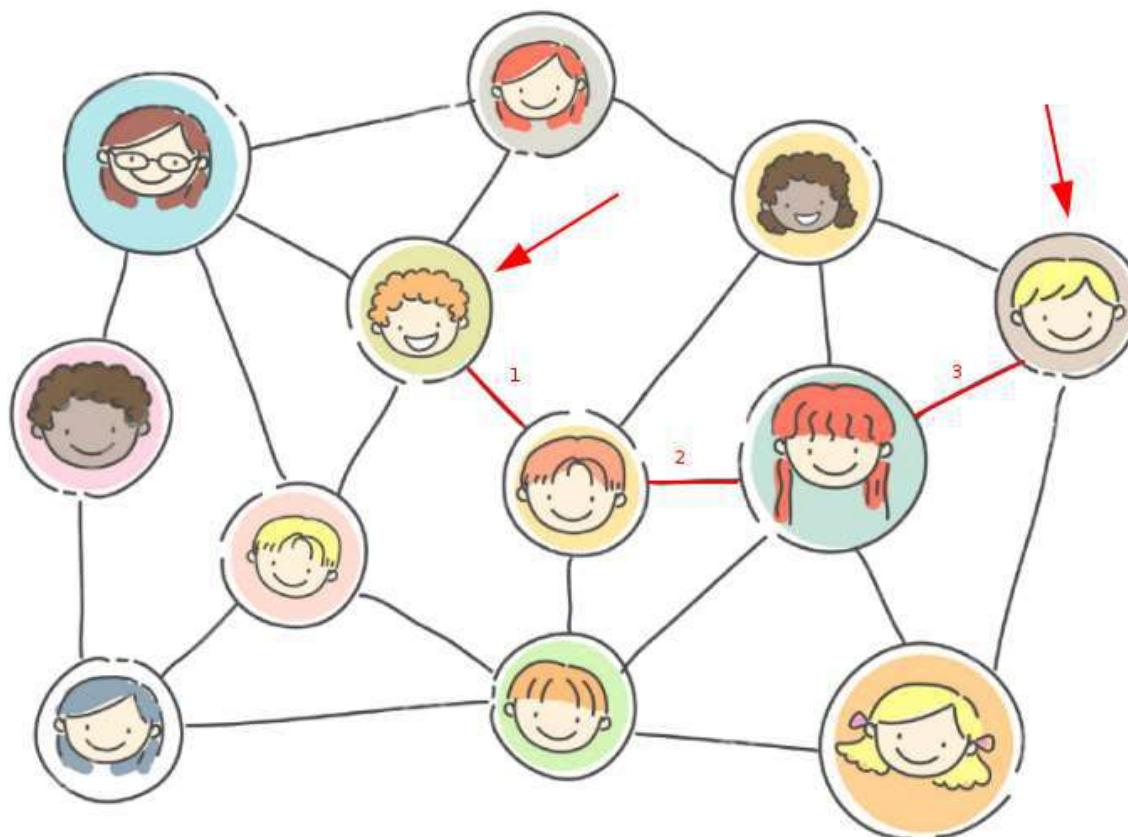


Malha rodoviária das principais rodovias de MG

Fonte: mapasblog.blogspot.com

Aplicações

- ◆ Descobrir qual é a "distância" entre dois membros de uma rede social



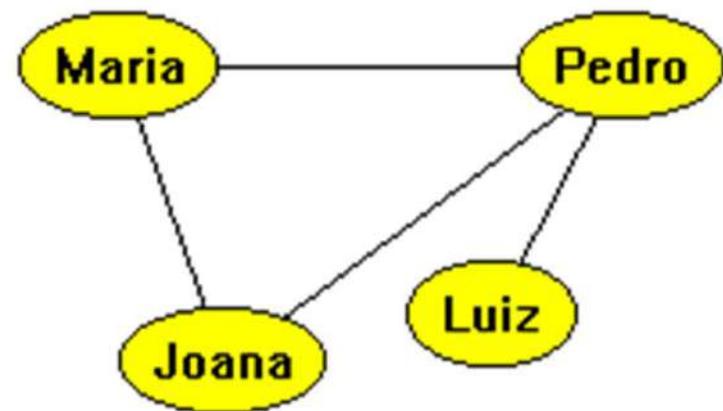
Representação de uma rede de pessoas utilizando grafos

Grafos: definição informal

- ◆ Grafos são uma forma de representar **elementos** de um conjunto e as **relações** entre esses elementos

- ◆ Exemplos :

- Grafo de Amizade



Maria, Pedro, Joana e Luiz são os elementos do conjunto
Uma **relação de amizade** representada no “grafo” acima é :
Maria é amiga de Pedro e Joana, mas não é amiga de Luiz

Modelagem em grafos

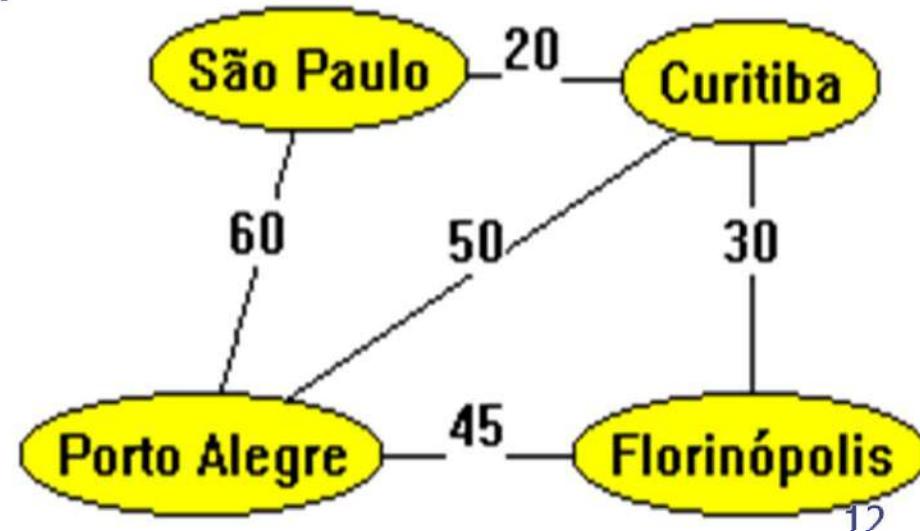
- ◆ Estamos interessados em modelar objetos/elementos e a relação entre eles
- ◆ Quem são eles nas aplicações mencionadas?
- ◆ Representação do modelo:
 - Forma matemática
 - Forma geométrica (gráfica)

Modelagem em grafos

Exemplo

◆ Problemas de Transporte

- Uma das mais acessíveis aplicações da teoria dos grafos é em problemas que envolvem **transportes de carga ou pessoas**
- Neste tipo de problema, os pontos de parada, embarque e desembarque podem ser representados por **vértices** e as estradas entre os pontos por **arestas**



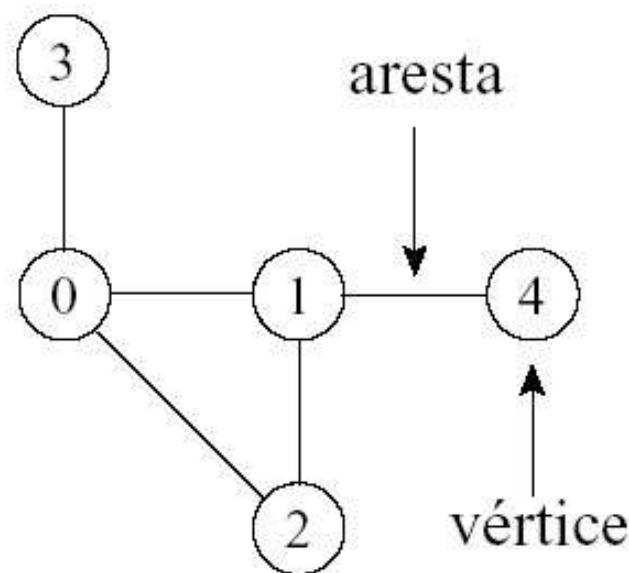
Modelagem em grafos

Exemplo

- ◆ Podemos então formular questões que podem ser resolvidas com os algoritmos em grafos :
 - **Problema de Conectividade :** dadas as direções das vias, é possível ir da cidade A até a cidade B, sem andar na contramão ?
 - **Problema de Fluxo Máximo:** dada a capacidade de fluxo em cada via, quanta mercadoria podemos mandar de uma cidade A a uma cidade B ?
 - **Problema de Menor Caminho:** Dados os comprimentos de cada via, qual o percurso mais rápido/barato para sair de uma cidade A e chegar a uma cidade B ?

Conceitos Básicos

- ◆ **Grafo:** conjunto de vértices e arestas.
- ◆ **Vértice:** objeto simples que pode ter nome e outros atributos.
- ◆ **Aresta:** conexão entre dois vértices.

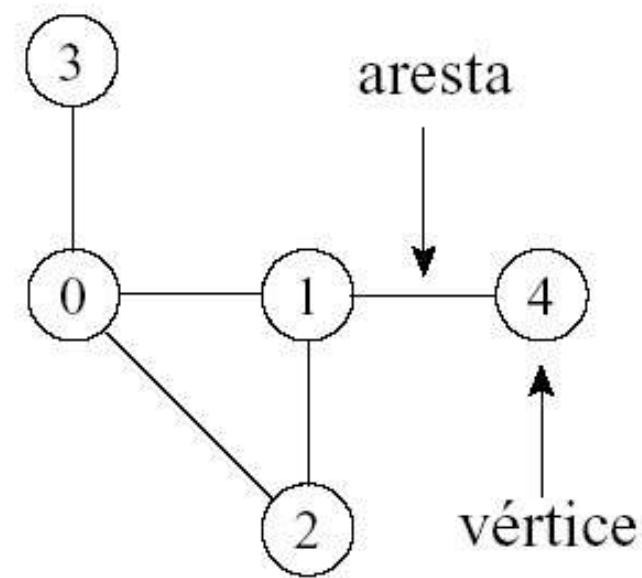


Conceitos Básicos

- ◆ Notação: $G = (V, A)$
 - G : grafo
 - V : conjunto de vértices
 - A : conjunto de arestas
- ◆ Uma aresta é representada por $a = (u, v)$, e sempre interliga dois vértices quaisquer u e v de V .
- ◆ Dois vértices ligados por uma aresta são denominados **adjacentes**.

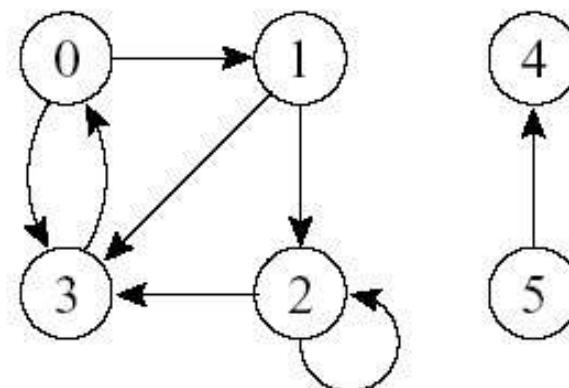
Conceitos Básicos

- ◆ A **ordem** de um grafo G é dada pela cardinalidade do conjunto de vértices $|V|$, ou seja, pelo número de vértices de G
- ◆ O **número de arestas** de um grafo é dado por $|A|$ de G . Assim, para o grafo do exemplo abaixo, temos:
 - $|V| = 5$
 - $|A| = 5$

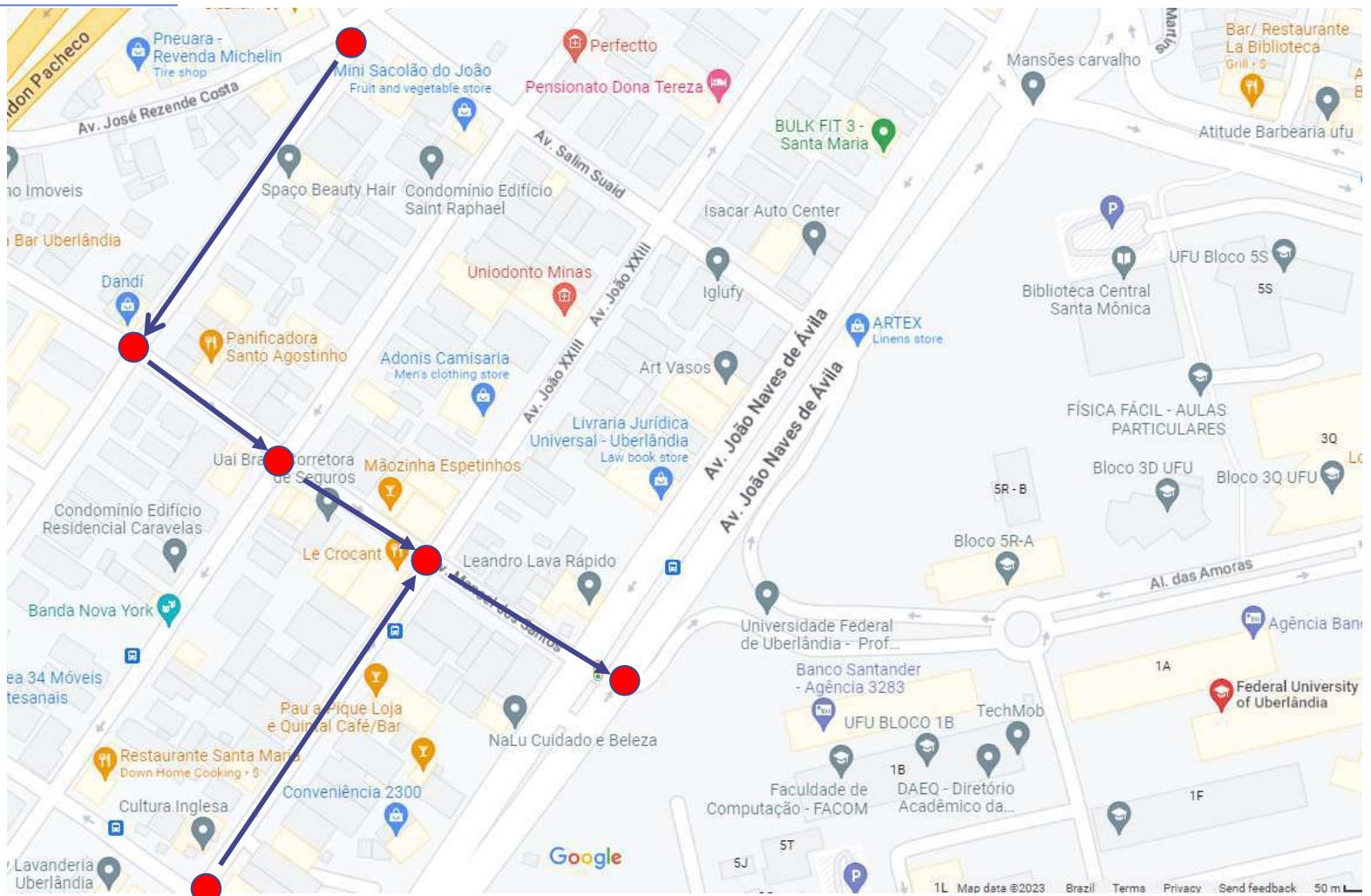


Grafos Direcionados

- ◆ Um **grafo direcionado** G é um par (V, A) , onde V é um conjunto finito de vértices e A é uma relação binária em V .
 - Uma aresta (u, v) sai do vértice u e entra no vértice v . O vértice v é adjacente ao vértice u .
 - Podem existir arestas de um vértice para ele mesmo, chamadas de *self-loops*.

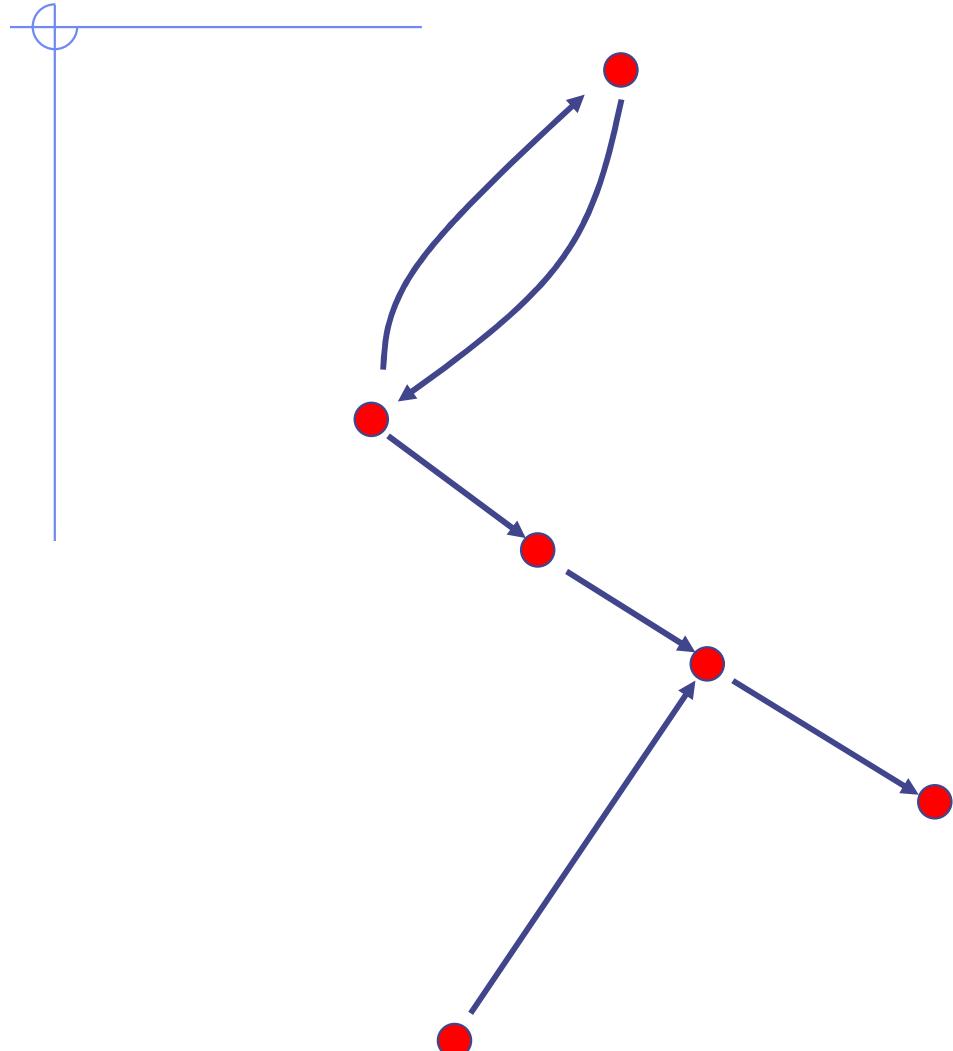


Grafos Direcionados



Exemplo: Traçando trajetos até a UFU

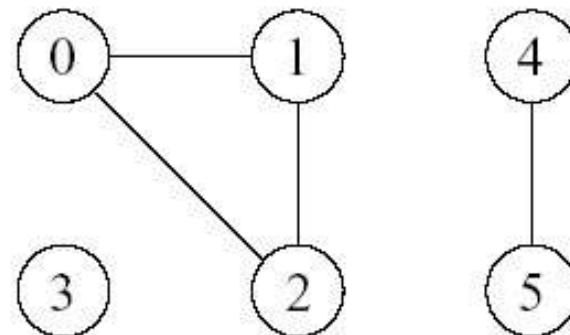
Grafos Direcionados



Exemplo: Grafo representando um trecho de mão dupla

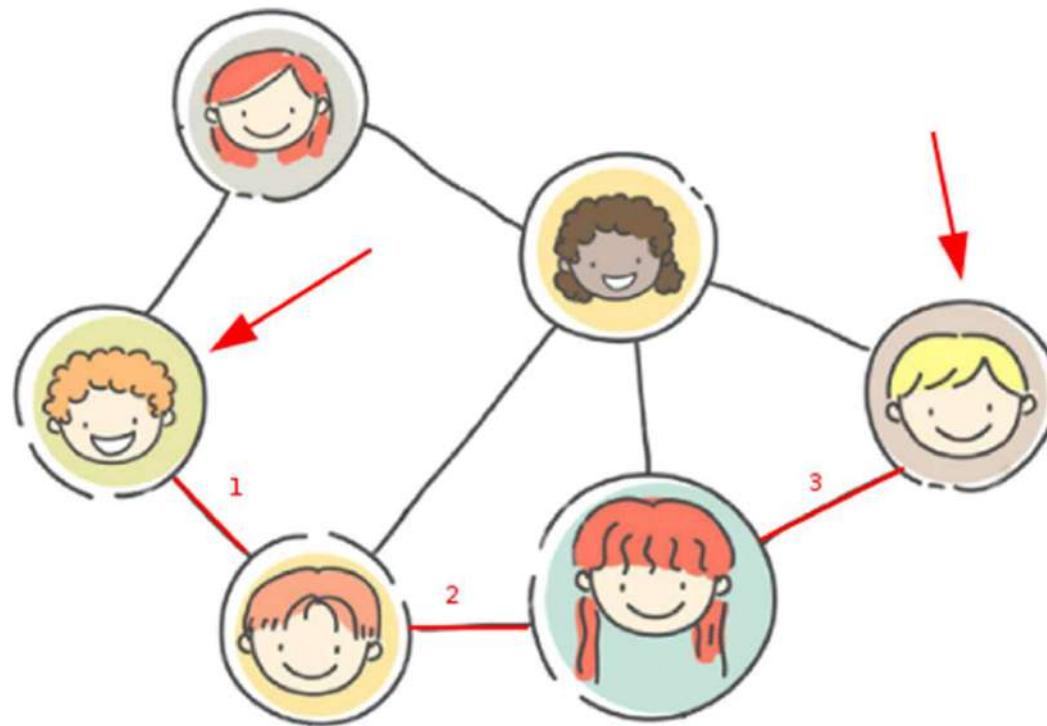
Grafos Não-Direcionados

- ◆ Um **grafo não direcionado** G é um par (V, A) , onde o conjunto de arestas A é constituído de pares de vértices não ordenados.
 - As arestas (u, v) e (v, u) são consideradas como uma única aresta.
 - A relação de adjacência é simétrica.
 - *Self-loops* não são permitidos.



Grafos Não-Direcionados

- Exemplo: Para calcular a distância entre dois membros de uma rede social não é preciso saber a direção



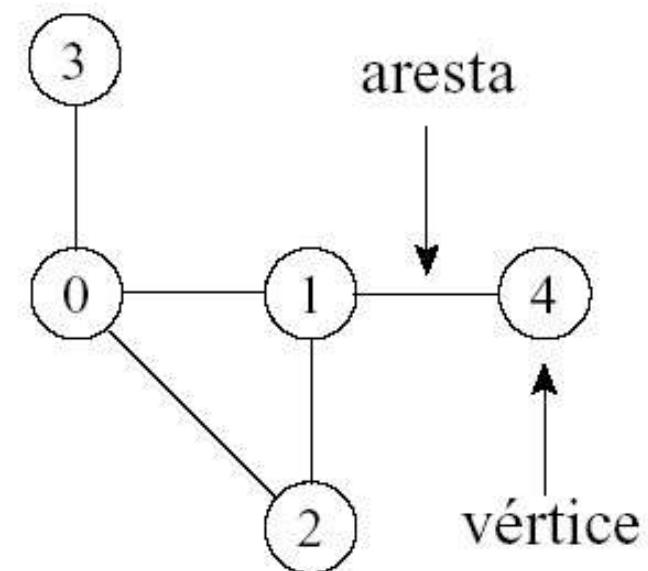
Representação de uma rede de pessoas utilizando grafos

Vértices Adjacentes

◆ Diz-se que os vértices u e v são **adjacentes** (ou **vizinhos**) quando estes forem os extremos de uma mesma aresta $a = (u, v)$

◆ Assim:

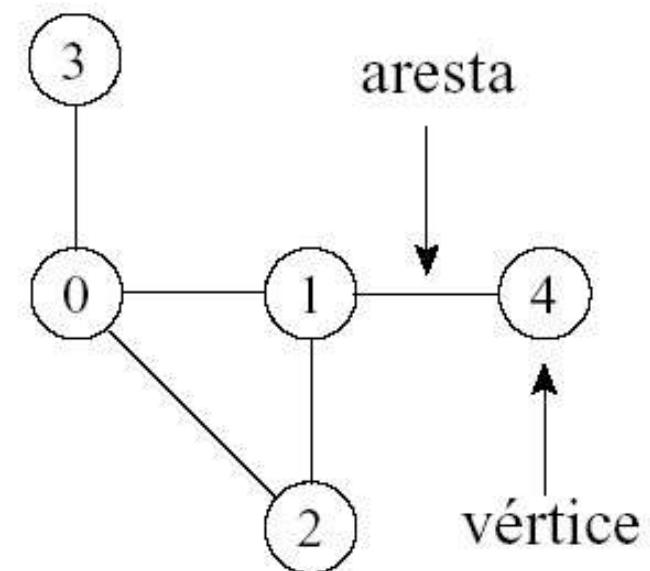
- V_1 é adjacente a V_4
- V_4 é adjacente a V_1
- V_2 NÃO é adjacente a V_4



Arestas Adjacentes

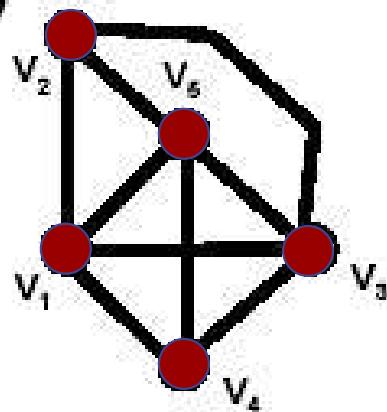
- ◆ Diz-se que duas arestas *são adjacentes (ou vizinhas)* quando estas possuírem um mesmo extremo, ou vértice

- ◆ Assim:
 - (V_0, V_1) é adjacente a (V_1, V_4)
 - (V_0, V_1) é adjacente a (V_0, V_2)

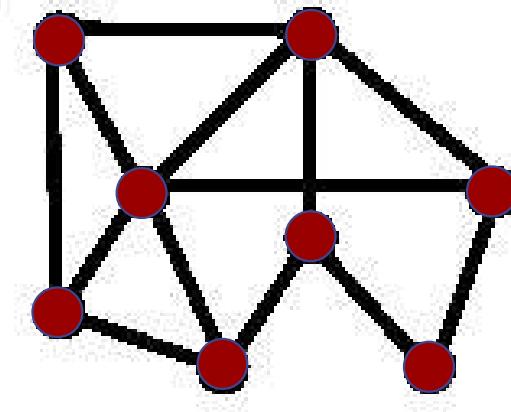


Exercícios de Fixação

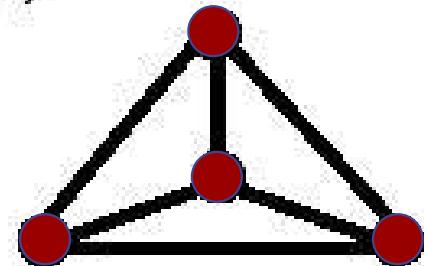
(a)



(b)



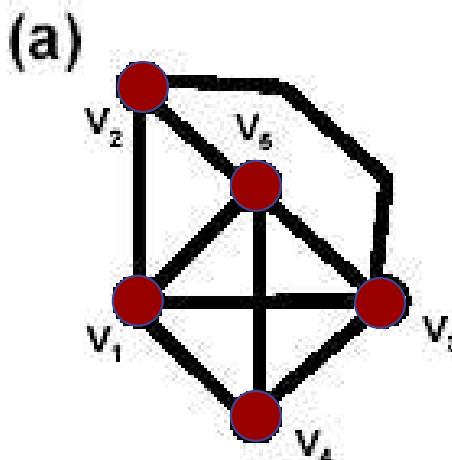
(c)



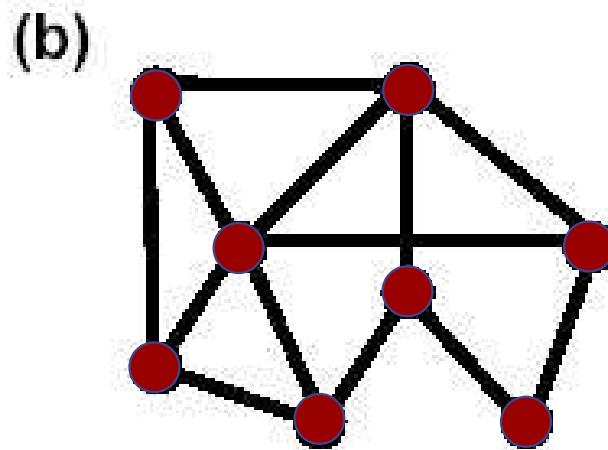
- ◆ Qual a ordem e o número de arestas de cada grafo?
- ◆ No grafo (a), quais vértices são adjacentes a V_3 ? E quais arestas são adjacentes a (V_3, V_5) ?

Exercícios de Fixação

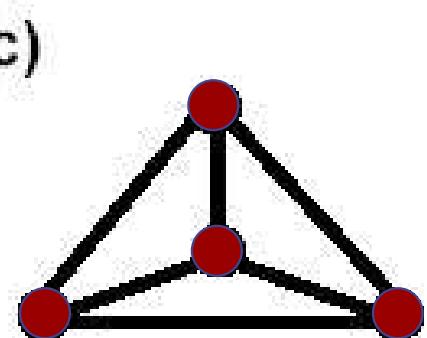
ordem = 5
nro arestas = 9



ordem = 8
nro arestas = 13



ordem = 4
nro arestas = 6



- ◆ Qual a ordem e o número de arestas de cada grafo?
- ◆ No grafo (a), quais vértices são adjacentes a V3? E quais arestas são adjacentes a (V3,V5)?

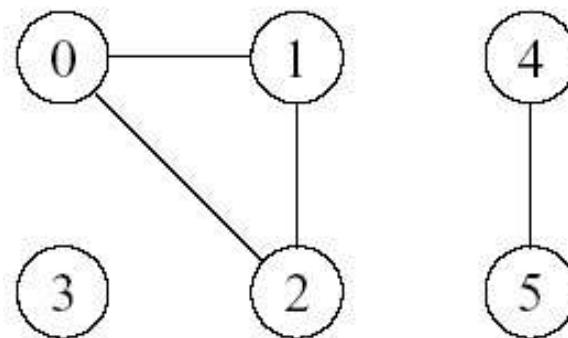
vértices adj. = V5, V2, V1 e V4

arestas adj. = (V3,V2), (V3,V1), (V3,V4), (V5,V2), (V5,V1), (V5,V4)

Grau de um vértice

◆ Em grafos não direcionados:

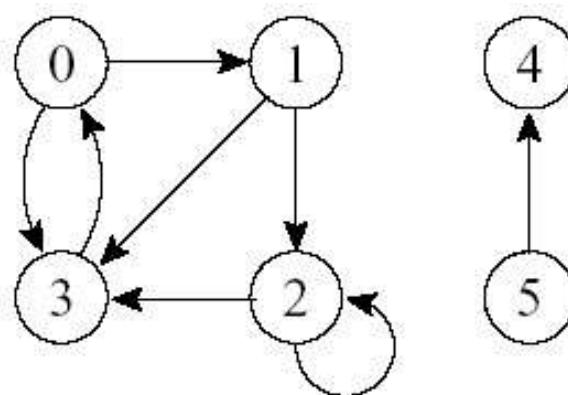
- O grau de um vértice é o número de arestas que incidem nele.
- Um vértice de grau zero é dito **isolado** ou **não conectado**.
- Ex.: O vértice 1 tem grau 2 e o vértice 3 é isolado.



Grau de um vértice

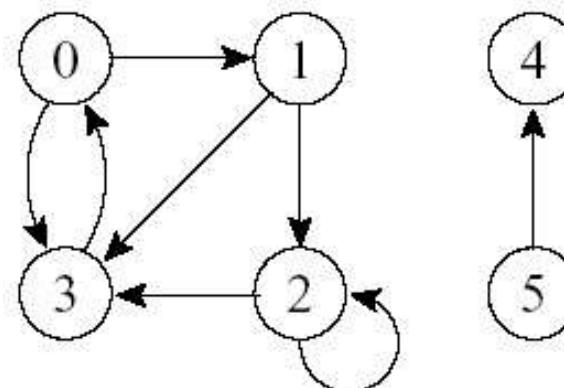
◆ Em grafos direcionados

- O grau de um vértice é o número de arestas que saem dele (*out-degree*) mais o número de arestas que chegam nele (*in-degree*).
- Ex.: O vértice 2 tem *in-degree* 2, *out-degree* 2 e grau 4.

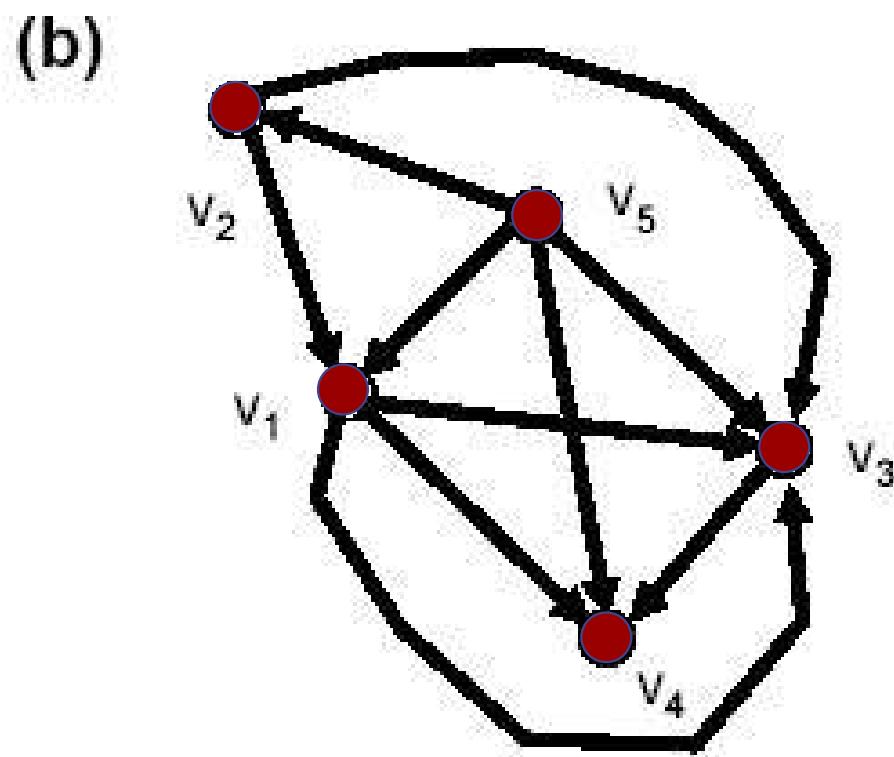
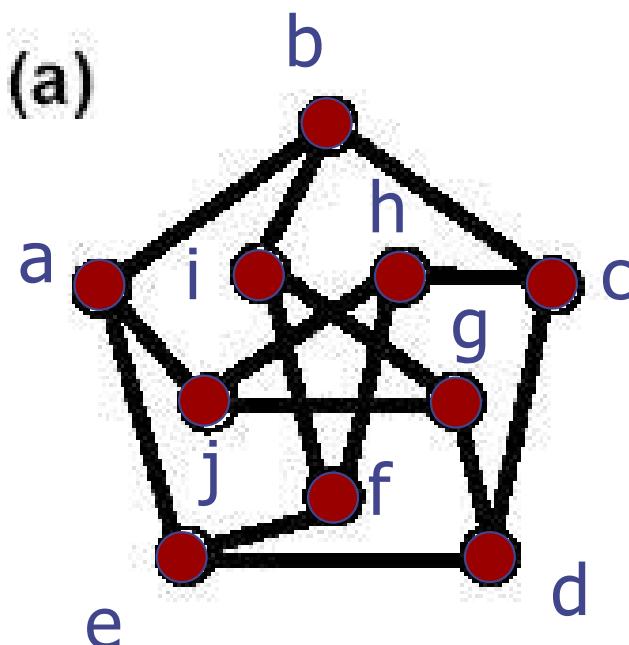


Grau de um vértice

- ◆ Um vértice com grau de saída nulo, ou seja, *out-degree* = 0, é chamado de **sumidouro** (ou **sorvedouro**)
- ◆ Um vértice com grau de entrada nulo, ou seja, *in-degree* = 0, é chamado de **fonte**
- ◆ Diz-se que um grafo é **regular** se todos os seus vértices tiverem o mesmo grau
 - Ex.: o vértice 4 é um sumidouro e o vértice 5 é uma fonte

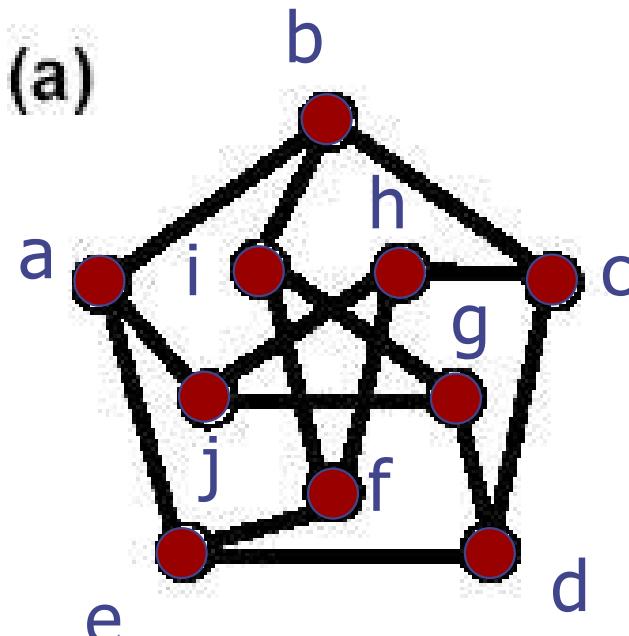


Exercícios de Fixação

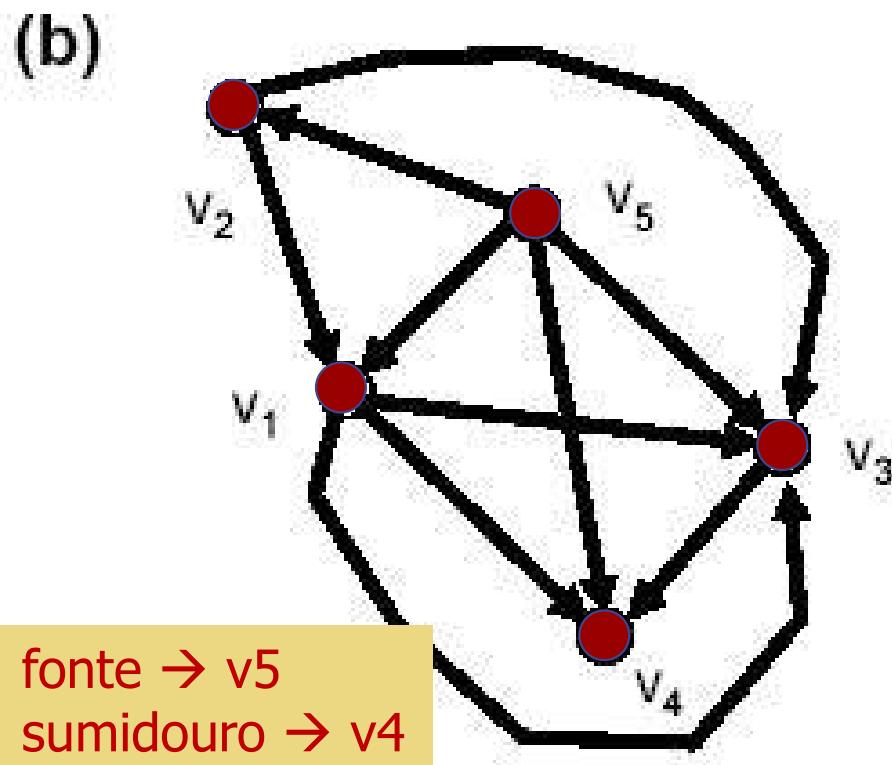


- ◆ O grafo (a) é regular? Por quê?
 - ◆ Existe alguma fonte ou sumidouro no grafo (b)?

Exercícios de Fixação



sim. resolução
no quadro.



fonte → v5
sumidouro → v4

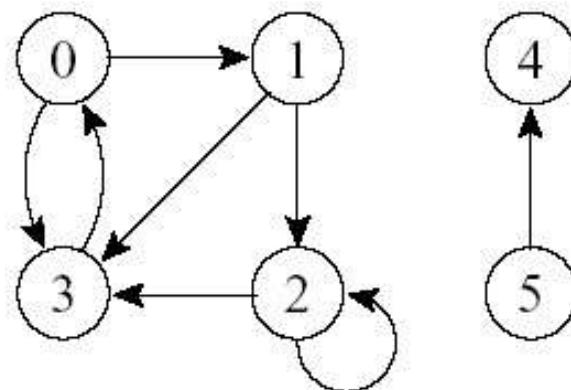
- ◆ O grafo (a) é regular? Por quê?
- ◆ Existe alguma fonte ou sumidouro no grafo (b)?

Caminho entre vértices

- ◆ Um caminho de **comprimento** k de um vértice x a um vértice y em um grafo $G = (V, A)$ é uma sequência de vértices $(v_0, v_1, v_2, \dots, v_k)$ tal que $x = v_0$ e $y = v_k$, e $(v_{i-1}, v_i) \in A$ para $i = 1, 2, \dots, k$.
- ◆ O comprimento de um caminho é o número de arestas nele, isto é, o caminho contém os vértices $v_0, v_1, v_2, \dots, v_k$ e as arestas $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$.
- ◆ Se existir um caminho c de x a y então y é alcançável a partir de x via c .

Caminho entre vértices

- ◆ Um caminho é **simples** se todos os vértices do caminho são distintos.
- ◆ Ex.: O caminho $(0, 1, 2, 3)$ é simples e tem comprimento 3. O caminho $(1, 3, 0, 3)$ não é simples.



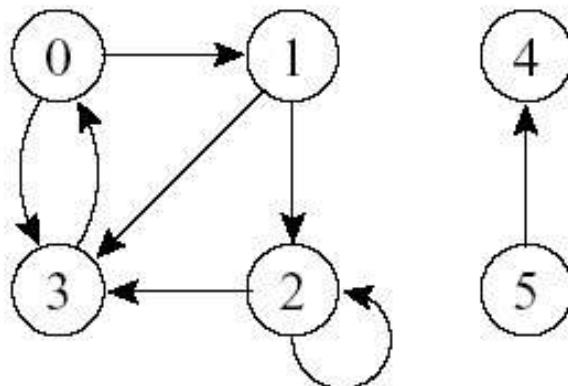
Ciclos

◆ Em um grafo direcionado:

- Um caminho (v_0, v_1, \dots, v_k) forma um ciclo se $v_0 = v_k$ e o caminho contém pelo menos uma aresta.
- O ciclo é simples se os vértices v_1, v_2, \dots, v_k são distintos.
- O *self-loop* é um ciclo de tamanho 1.
- Dois caminhos (v_0, v_1, \dots, v_k) e $(v'_0, v'_1, \dots, v'_k)$ formam o mesmo ciclo se existir um inteiro j tal que $v'_i = v_{(i+j) \bmod k}$ para $i = 0, 1, \dots, k-1$.

Ciclos

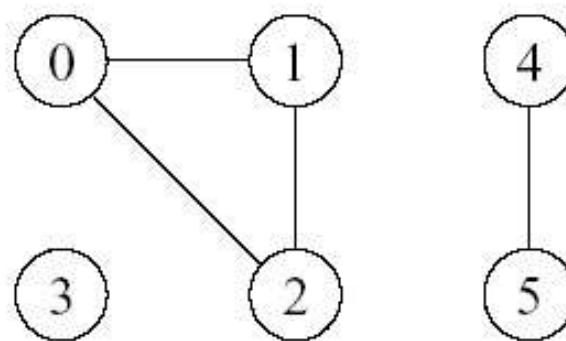
- Ex.: O caminho $(0, 1, 2, 3, 0)$ forma um ciclo. O caminho $(0, 1, 3, 0)$ forma o mesmo ciclo que os caminhos $(1, 3, 0, 1)$ e $(3, 0, 1, 3)$.



Ciclos

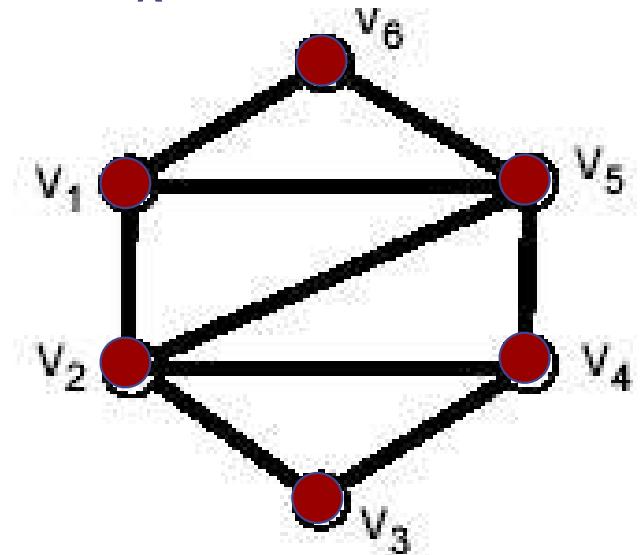
◆ Em um grafo não direcionado:

- Um caminho (v_0, v_1, \dots, v_k) forma um ciclo se $v_0 = v_k$ e o caminho contém pelo menos três arestas.
- O ciclo é simples se os vértices v_1, v_2, \dots, v_k são distintos.
- Ex.: O caminho $(0, 1, 2, 0)$ é um ciclo.



Caminho Hamiltoniano

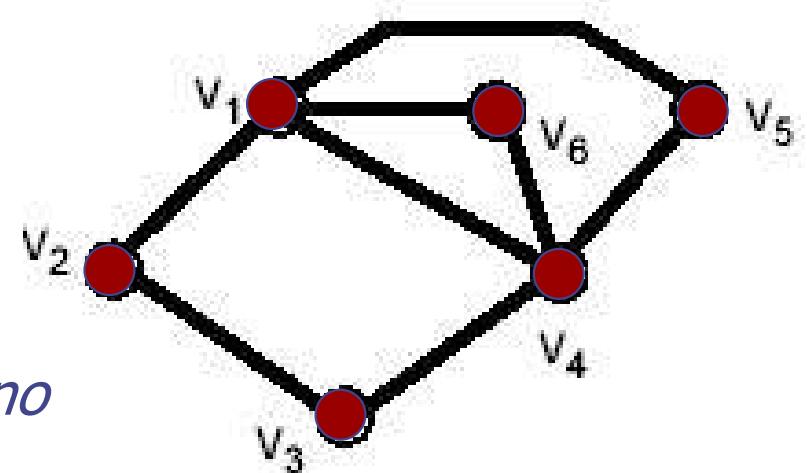
- ◆ É aquele que contém cada vértice do grafo exatamente uma vez.
- ◆ Um ciclo $V_1, V_2, \dots, V_k, V_{k+1}$ é hamiltoniano quando o caminho V_1, V_2, \dots, V_k for um caminho hamiltoniano.



- $V1, V6, V5, V2, V3, V4$ é hamiltoniano
- $V6, V5, V4, V3, V2, V1, V6$ é um ciclo hamiltoniano

Caminho Euleriano

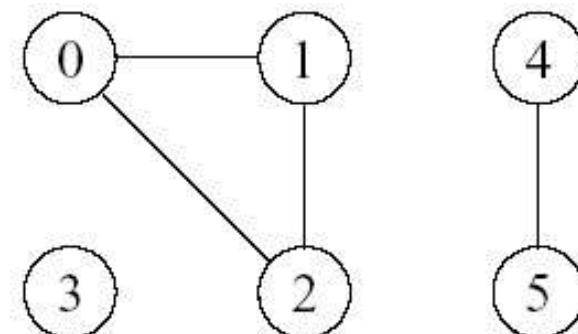
- ◆ É aquele que contém cada aresta do grafo exatamente uma vez.
- ◆ Um grafo é **Euleriano** se há um ciclo em G que contenha todas as suas arestas.



- ◆ $V_1, V_6, V_4, V_1, V_2, V_3, V_4, V_5, V_1$ é euleriano
- ◆ Portanto, este grafo é euleriano

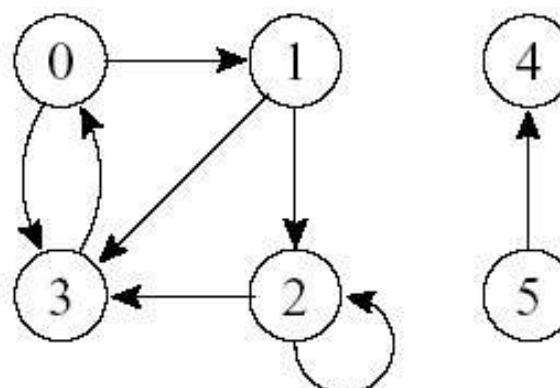
Componentes Conectados

- ◆ **Um grafo não direcionado** é conectado se cada par de vértices está conectado por um caminho.
- ◆ Os componentes conectados são as porções conectadas de um grafo.
- ◆ Um grafo não direcionado é conectado se ele tem exatamente um componente conectado.
- ◆ Ex.: Os componentes são: $\{0, 1, 2\}$, $\{4, 5\}$ e $\{3\}$.



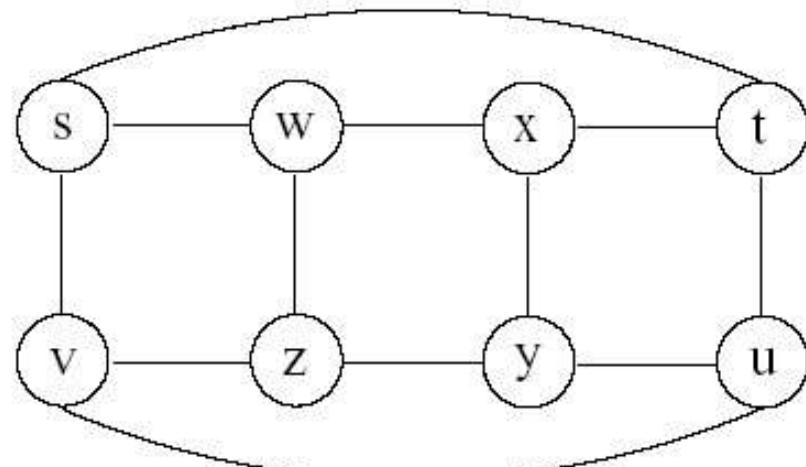
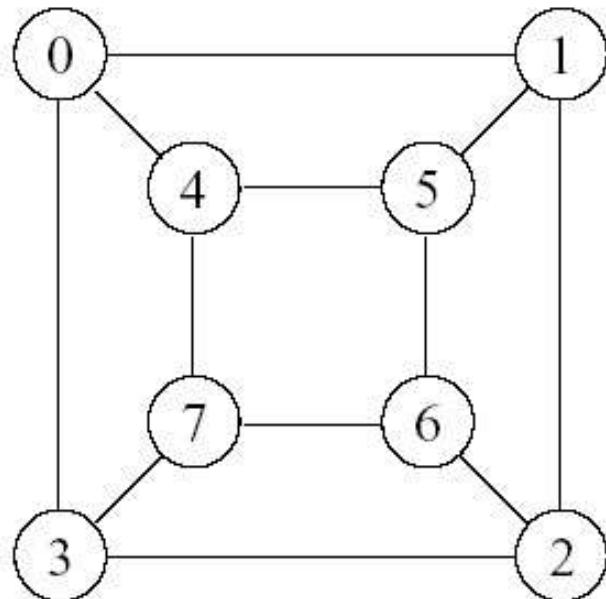
Componentes Fortemente Conectados

- ◆ Um grafo direcionado $G = (V, A)$ é **fortemente conectado se cada** dois vértices quaisquer são alcançáveis a partir um do outro.
- ◆ Os **componentes fortemente conectados de um grafo direcionado** são conjuntos de vértices sob a relação “são mutuamente alcançáveis”.
- ◆ Um **grafo direcionado fortemente conectado tem apenas um** componente fortemente conectado.
- ◆ Ex.: $\{0, 1, 2, 3\}$, $\{4\}$ e $\{5\}$ são os componentes fortemente conectados, o $\{4, 5\}$ não é pois o vértice 5 não é alcançável a partir do vértice 4.



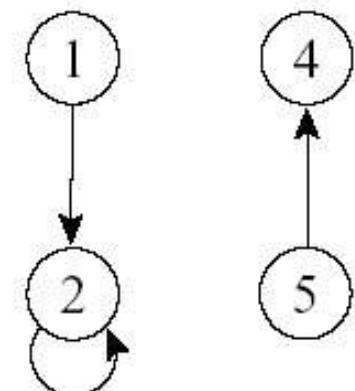
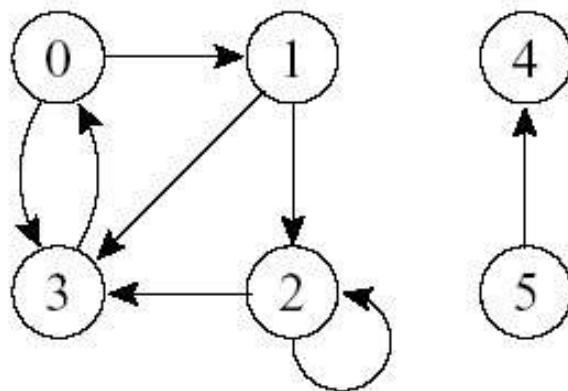
Grafos Isomorfos

- ◆ $G = (V, A)$ e $G' = (V', A')$ são isomorfos se existir uma bijeção $f: V \rightarrow V'$ tal que $(u, v) \in A$ se e somente se $(f(u), f(v)) \in A'$.
- ◆ Em outras palavras, é possível re-rotular os vértices de G para serem rótulos de G' mantendo as arestas correspondentes em G e G' .



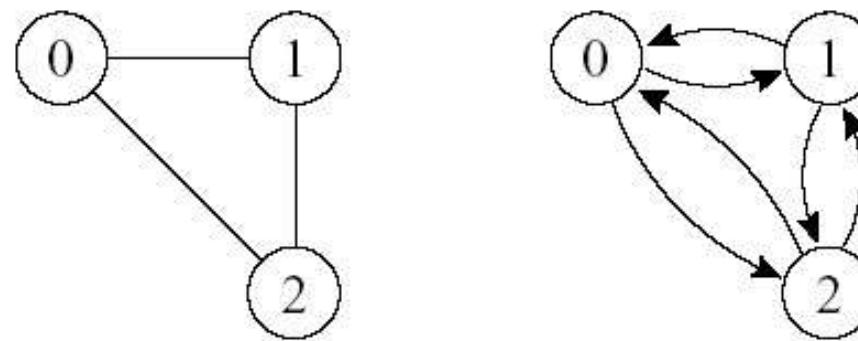
Subgrafos

- ◆ Um grafo $G' = (V', A')$ é um subgrafo de $G = (V, A)$ se $V' \subseteq V$ e $A' \subseteq A$.
- ◆ Dado um conjunto $V' \subseteq V$, o subgrafo induzido por V' é o grafo $G' = (V', A')$, onde $A' = \{(u, v) \in A \mid u, v \in V'\}$.
- ◆ Ex.: Subgrafo induzido pelo conjunto de vértices $\{1, 2, 4, 5\}$.



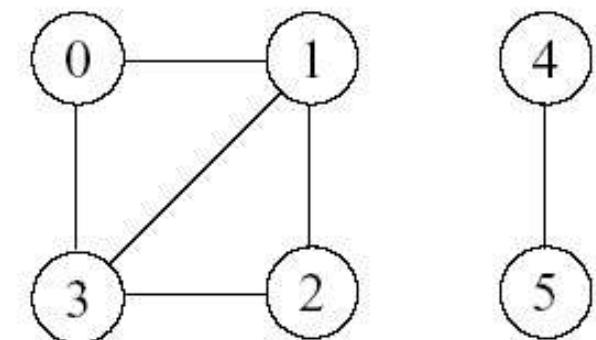
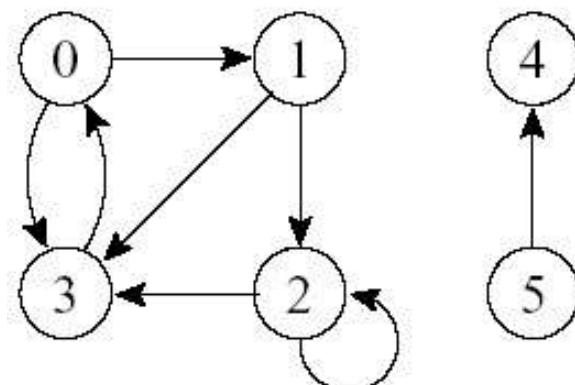
Versão direcionada de um grafo não direcionado

- ◆ A versão direcionada de um grafo não direcionado $G = (V, A)$ é um grafo direcionado $G' = (V', A')$ onde $(u, v) \in A'$ se e somente se $(u, v) \in A$.
- ◆ Cada aresta não direcionada (u, v) em G é substituída por duas arestas direcionadas (u, v) e (v, u)
- ◆ Em um grafo direcionado, um **vizinho de um vértice u é qualquer** vértice adjacente a u na versão não direcionada de G .



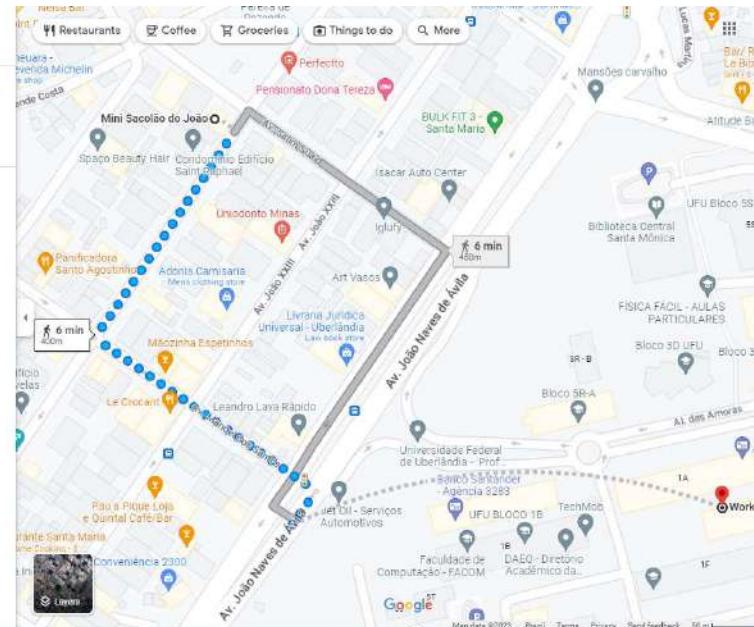
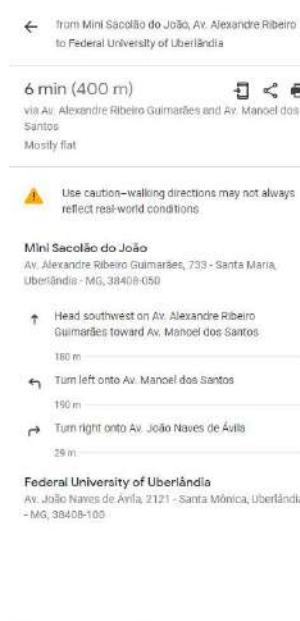
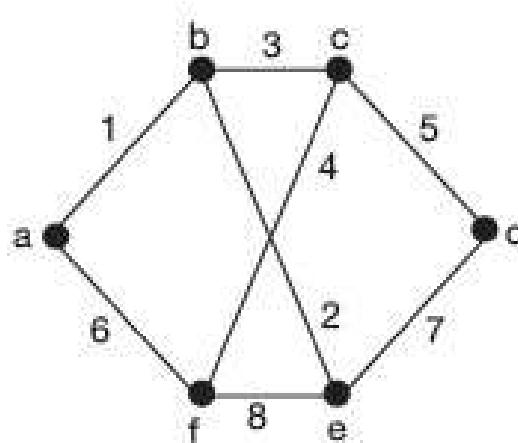
Versão não direcionada de um grafo direcionado

- ◆ A versão não direcionada de um grafo direcionado $G = (V, A)$ é um grafo não direcionado $G' = (V', A')$ onde $(u, v) \in A'$ se e somente se $u \neq v$ e $(u, v) \in A$ ou $(v, u) \in A$.
- ◆ A versão não direcionada contém as arestas de G sem a direção e sem os *self-loops*.
- ◆ Em um grafo não direcionado, u e v são vizinhos se eles são adjacentes.



Outras classificações de grafos

- ❖ **Grafo ponderado:** possui pesos associados às arestas.
 - Esses pesos podem representar custos ou distâncias, por exemplo.



Exemplo: Informação de trajeto, distância e tempo no Google Maps

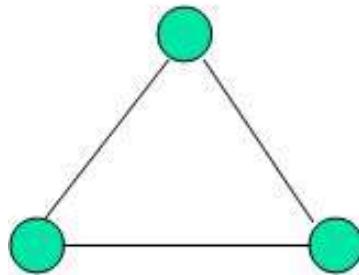
Grafos Completos

- ◆ Um grafo completo é um grafo não direcionado no qual todos os pares de vértices são adjacentes.
- ◆ K_n é um grafo completo com n vértices.
- ◆ Possui $(|V|^2 - |V|)/2 = |V|(|V| - 1)/2$ arestas, pois do total de $|V|^2$ pares possíveis de vértices devemos subtrair $|V|$ *self-loops* e dividir por 2 (cada aresta ligando dois vértices é contada duas vezes).

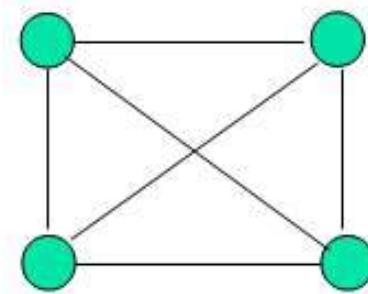
Grafos Completos

Exemplo

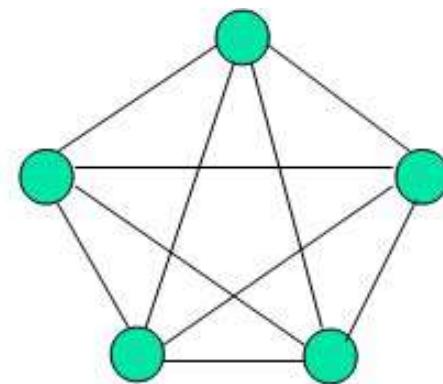
- ◆ Os grafos K_3 , K_4 e K_5



K_3



K_4



K_5

- ◆ possuem os seguintes números de arestas:

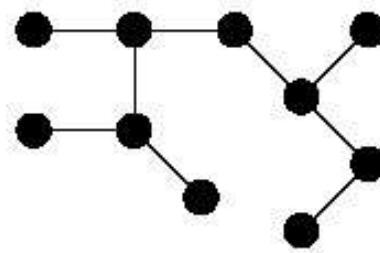
Grafo	#arestas
K_3	3
K_4	6
K_5	10

Árvores

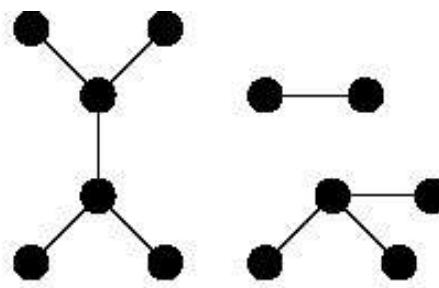
- ◆ **Árvore livre:** grafo não direcionado acíclico e conectado. É comum dizer apenas que o grafo é uma árvore omitindo o “livre”.
- ◆ **Floresta:** grafo não direcionado acíclico, podendo ou não ser conectado.
- ◆ **Árvore geradora** de um grafo conectado $G = (V, A)$: subgrafo que contém todos os vértices de G e forma uma árvore.
- ◆ **Floresta geradora** de um grafo $G = (V, A)$: subgrafo que contém todos os vértices de G e forma uma floresta.

Árvores – Exemplos

Árvore livre



Floresta

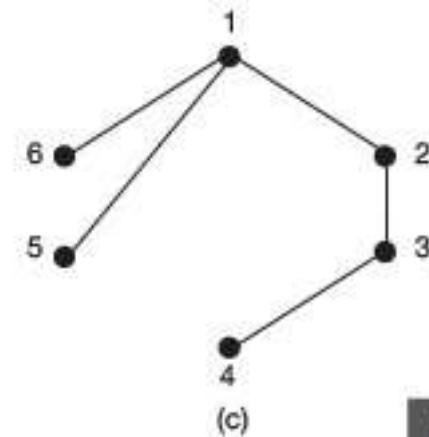
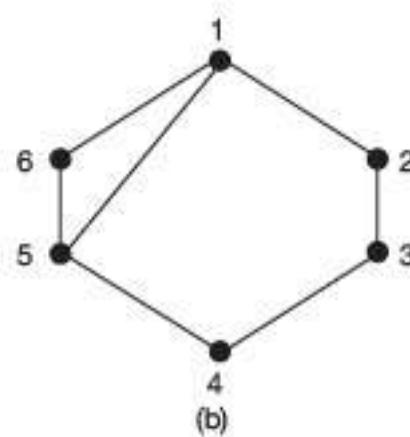
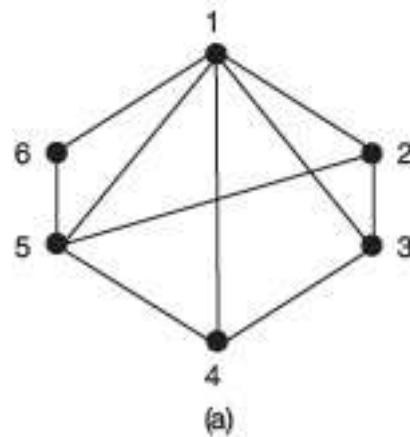


(a)

(b)

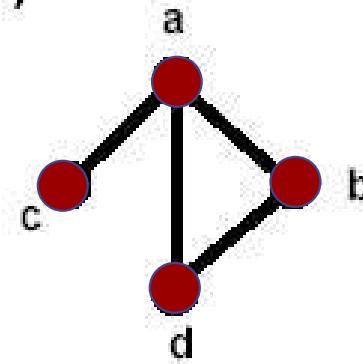
Árvores – Exemplos

- ◆ (a) Grafo G; (b) Subgrafo gerador; (c) Árvore geradora

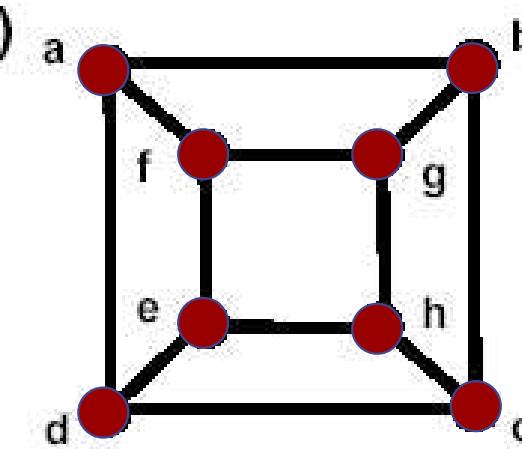


Exercícios de Fixação

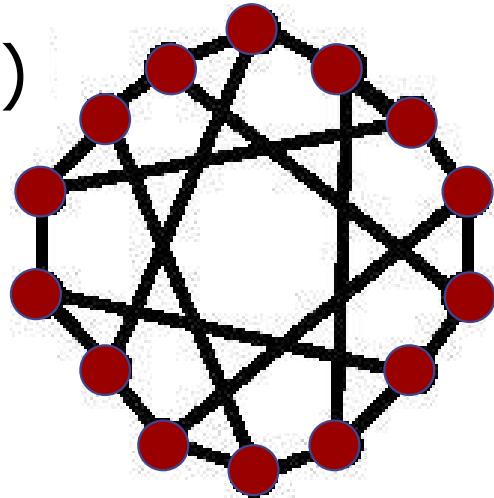
(a)



(b)



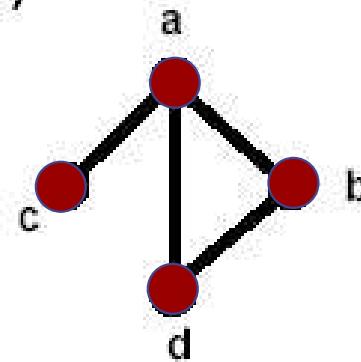
c)



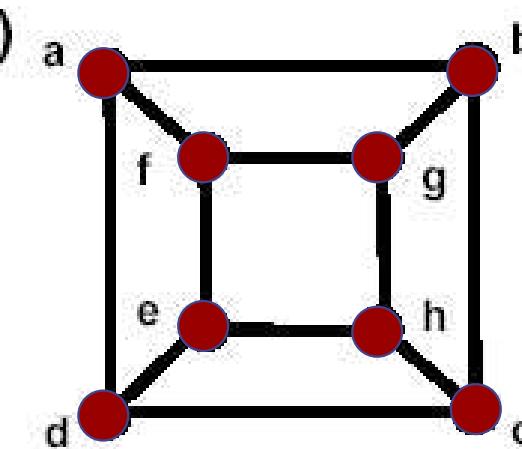
- ◆ Qual dos grafos acima são cílicos?
- ◆ Indique os grafos que são conectados.
- ◆ Indique os grafos que são completos.

Exercícios de Fixação

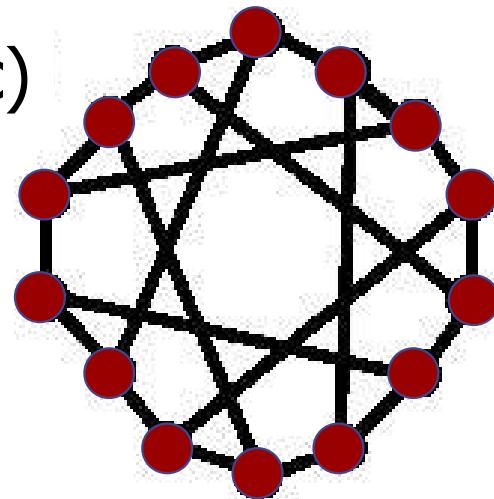
(a)



(b)



c)



◆ Qual dos grafos acima são cílicos? **a, b e c**

◆ Indique os grafos que são conectados. **a, b e c**

◆ Indique os grafos que são completos. **nenhum**

$$K_4 = 6 \quad K_8 = 28 \quad K_{14} = 91$$

Bibliografia

- 
- Aho, A.V.; Hopcroft, J.E., Ullman, J.D. Data Structures and Algorithms, Addison-Wesley, 1987.
 - Tenenbaum, A.M.; Langsam, Y.; Augenstein, M.J. Data Structures Using C, Prentice-Hall, New Jersey, 1990.
 - Ziviani, N. Projeto de Algoritmos, Thomson Learning, 2005.
 - Ascencio, A. F. G.; Araújo, G. S. Estruturas de Dados, Pearson, 2010.

Material complementar

Ferramenta para visualização de grafos
<https://graphonline.ru/en/>

Bacharelado em Ciência da Computação
GBC034 Algoritmos e Estruturas de Dados 2

Grafos

Representação

Material baseado nos slides do professor Nivio Ziviani
(Projeto de Algoritmos)

Profa. Maria Camila Nardini Barioni

camila.barioni@ufu.br

Bloco B - sala 1B137

2º semestre de 2023

Roteiro

- ◆ Representação de grafos
- ◆ Tipo Abstrato de Dados Grafo
- ◆ Algoritmos para representação de Grafos

Relembrando

◆ Definição de grafo

- Um grafo G é formado pelo par de conjuntos V e A , sendo V o conjunto de vértices de G , e A o conjunto de arestas de G que interligam dois vértices quaisquer de V

◆ Notação: $G = (V, A)$

- $|V|$: quantidade de vértices de G
- $|A|$: quantidade de arestas de G

◆ Ordem de um grafo

- Número de vértices que ele possui
 - ◆ $ordem(G) = |V|$

Representação de grafos

- ◆ Um grafo pode ser representado adequadamente por meio de
 - matrizes de adjacência
 - listas de adjacência

Matriz de Adjacência

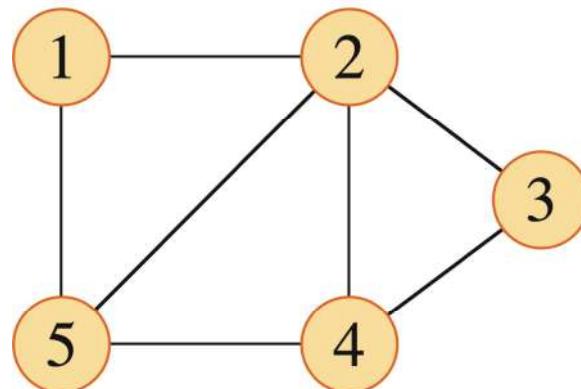
- ◆ A matriz de adjacência de um grafo $G = (V, A)$ de ordem n é uma matriz M de bits formada por n linhas e n colunas, onde

$$\blacksquare m_{ij} = \begin{cases} 1, & \text{se } (i, j) \in A \\ 0, & \text{se } (i, j) \notin A \end{cases}$$

- ◆ Uma matriz de adjacência é **simétrica** para **grafos não direcionados**

Matriz de Adjacência: Exemplo

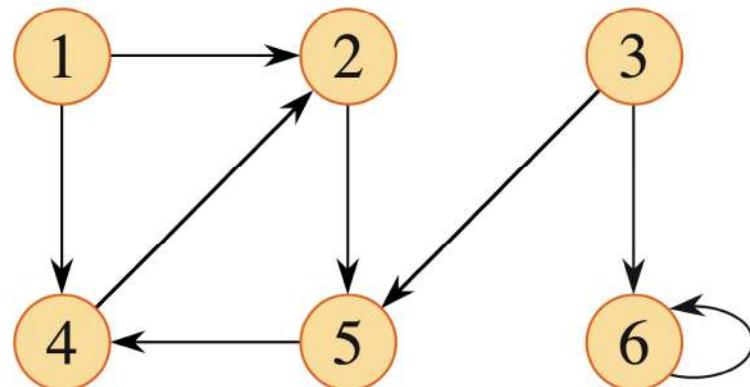
- ◆ Matriz de adjacência para um grafo não direcionado



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

Matriz de Adjacência: Exemplo

- ◆ Matriz de adjacência para um grafo direcionado



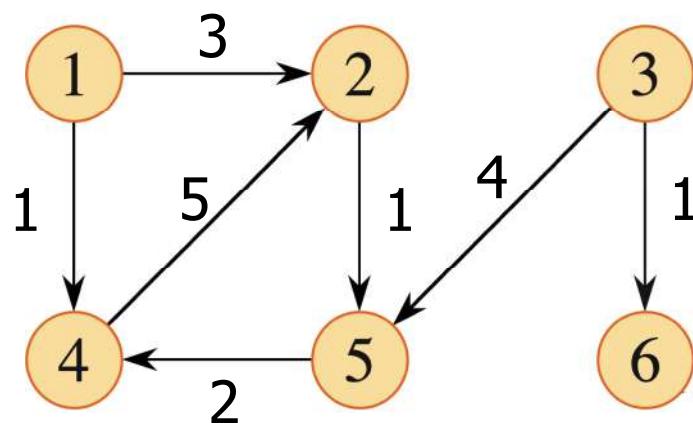
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Matriz de Adjacência

- ◆ Para grafos ponderados m_{ij} contém o rótulo ou peso associado com a aresta, neste caso, a matriz não é de *bits*
- ◆ Se não existir uma aresta de i para j então é necessário utilizar um valor que não possa ser usado como rótulo ou peso

Matriz de Adjacência: Exemplo

◆ Matriz de adjacência para um grafo direcionado ponderado



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

Matriz de Adjacência

- ◆ A construção da matriz e sua manipulação são operações simples
- ◆ É fácil determinar se $(i, j) \in A$
- ◆ É fácil encontrar os vértices adjacentes a um determinado vértice i
- ◆ Quando o grafo é não orientado, a matriz é simétrica (mais econômica)
- ◆ A inserção de novas arestas é fácil
- ◆ A inserção de novos vértices é muito difícil

Matriz de Adjacência: Análise

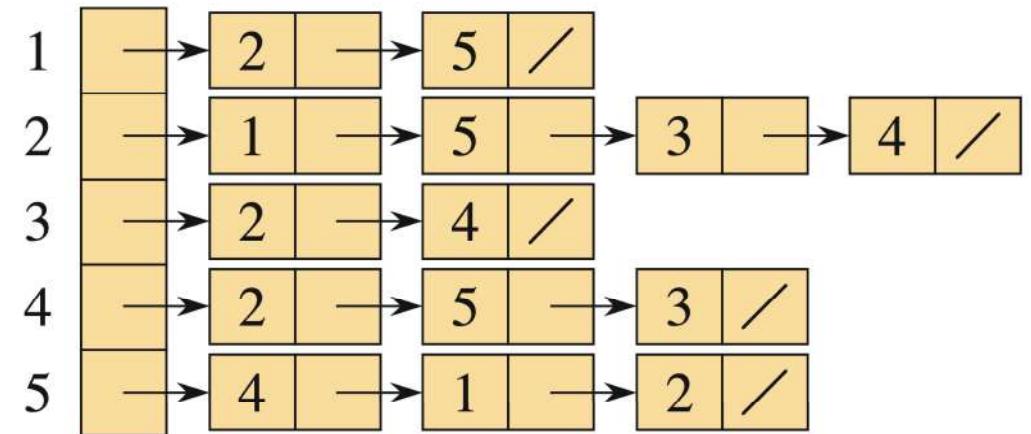
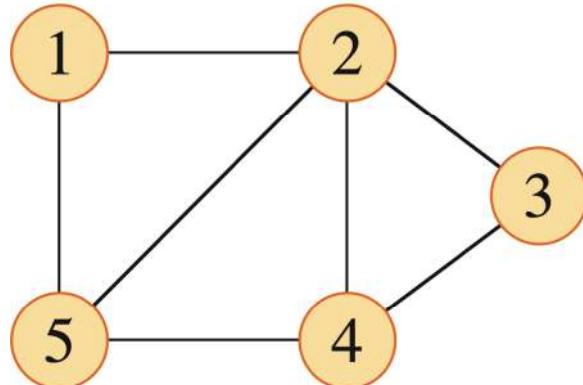
- ◆ Deve ser utilizada para grafos **densos**, onde $|A|$ é próximo de $|V|^2$
- ◆ O tempo necessário para acessar um elemento é independente de $|V|$ ou $|A|$
- ◆ É muito útil para algoritmos em que necessitamos saber com rapidez se existe uma aresta ligando dois vértices
- ◆ A maior desvantagem é que a matriz necessita $(|V|^2)$ de espaço. Ler ou examinar a matriz tem complexidade de tempo $O(|V|^2)$

Listas de Adjacência Usando Apontadores

- ◆ A representação de um grafo $G = (V, A)$ por listas de adjacência consiste de um array Adj com $n = |V|$ entradas, uma para cada vértice em V
- ◆ Para cada $u \in V$, $Adj[u]$ contém uma lista encadeada com todos os vértices adjacentes a u em G

Listas de Adjacência Usando Apontadores - Exemplo

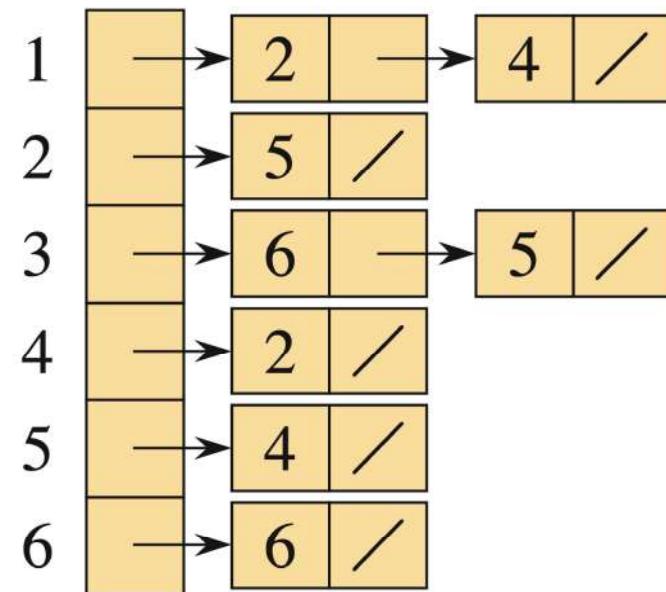
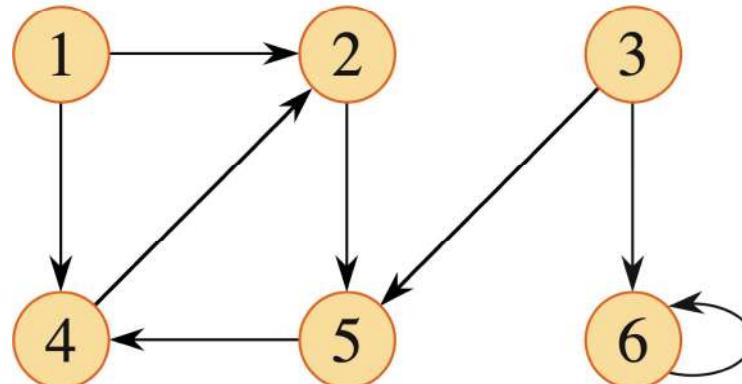
- ◆ Grafo não direcionado
 - Para cada vértice u é associada a lista de vértices v tais que $\{u, v\} \in A$



Listas de Adjacência Usando Apontadores - Exemplo

◆ Grafo direcionado

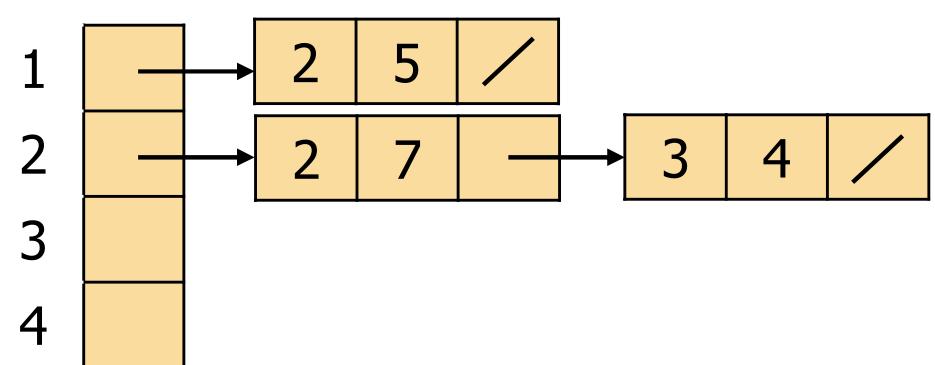
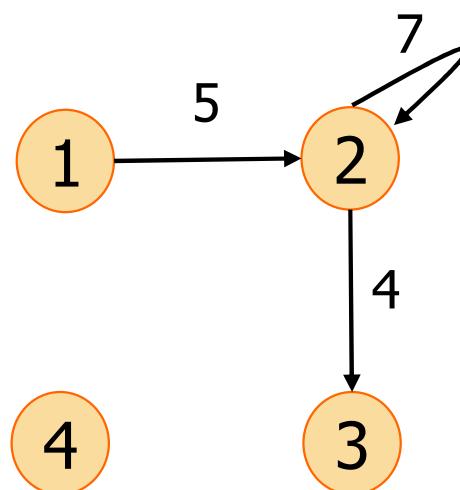
- Para cada vértice u é associada a lista de vértices v tais que $(u, v) \in A$



Listas de Adjacência Usando Apontadores - Exemplo

◆ Grafo ponderado

- Para cada vértice u é associada a lista de vértices v com seus pesos correspondentes tais que $(u, v, k) \in A$



Listas de Adjacência

- ◆ A implementação de listas de adjacência pode ser realizada por meio das duas estruturas de dados usuais para representar listas lineares
 - Apontadores
 - Posições contíguas de memória
- ◆ Nessa disciplina, vamos considerar o uso de apontadores

Listas de Adjacência Usando Apontadores: Análise

- ◆ Os vértices de uma lista de adjacência são em geral armazenados em uma ordem arbitrária
- ◆ Possui uma complexidade de espaço $O(|V| + |A|)$
- ◆ Indicada para grafos **esparsos**, onde $|A|$ é muito menor do que $|V|^2$
- ◆ É compacta e usualmente utilizada na maioria das aplicações
- ◆ A principal desvantagem é que ela pode ter tempo $O(|V|)$ para determinar se existe uma aresta entre o vértice i e o vértice j , pois podem existir $O(|V|)$ vértices na lista de adjacentes do vértice i

Comparação

Aspecto	Melhor
Rapidez para saber se (x,y) está no grafo	Matriz de adjacência
Rapidez para determinar o grau de um vértice	Listas de adjacência
Memória menor em grafos pequenos	$\text{Lista} = (A + V)$ $\text{Matriz} = (V ^2)$
Memória menor em grafos grandes	Matriz de adjacência
Melhor na maioria dos problemas	Lista de adjacência
Rapidez para percorrer o grafo	$\text{Lista} = (A + V)$ $\text{Matriz} = ((V ^2))$

Tipo Abstrato de Dados

- ◆ Importante considerar os algoritmos em grafos como **tipos abstratos de dados**
- ◆ Conjunto de operações associado a uma estrutura de dados
- ◆ Independência de implementação para as operações

Operações do TAD

1. *FGVazio(Grafo)*: Cria um grafo vazio
2. *InsereAresta(V1, V2, Peso, Grafo)*: Insere uma aresta no grafo
3. *ExisteAresta(V1, V2, Grafo)*: Verifica se existe uma determinada aresta
4. Obtem a lista de vértices adjacentes a um determinado vértice (tratada a seguir)
5. *RetiraAresta(V1, V2, Peso, Grafo)*: Retira uma aresta do grafo
6. *LiberaGrafo(Grafo)*: Liberar o espaço ocupado por um grafo

Operações do TAD

7. *ImprimeGrafo(Grafo)*: Imprime um grafo

8. *GrafoTransposto(Grafo,GrafoT)*: Obtém o transposto de um grafo direcionado

9. *RetiraMin(A)*: Obtém a aresta de menor peso de um grafo com peso nas arestas

Operação “Obter Lista de Adjacentes”

1. *ListaAdjVazia(v, Grafo)*: retorna true se a lista de adjacentes de v está vazia
2. *PrimeiroListaAdj(v, Grafo)*: retorna o endereço do primeiro vértice na lista de adjacentes de v
3. *ProxAdj(v, Grafo, u, Peso, Aux, FimListaAdj)*: retorna o vértice u (apontado por *Aux*) da lista de adjacentes de v , bem como o peso da aresta (v, u) . Ao retornar, *Aux* aponta para o próximo vértice da lista de adjacentes de v , e *FimListaAdj* retorna *true* se o final da lista de adjacentes foi encontrado

Operação “Obter Lista de Adjacentes”

- ◆ É comum encontrar um pseudo comando do tipo:
 - For $u \in \text{ListaAdjacentes}(v)$ do { faz algo com u }
- ◆ O trecho de programa abaixo apresenta um possível refinamento do pseudo comando acima

```
if (!ListaAdjVazia(v, Grafo))  
{ Aux = PrimeiroListaAdj(v, Grafo);  
FimListaAdj = FALSE;  
while (!FimListaAdj) {  
    ProxAdj (&v, Grafo, &u, &Peso, &Aux,  
            &FimListaAdj);  
    printf ("%2d (%d)", Adj, Peso); }  
}
```

Representação de grafos

◆ Algoritmos

- Matriz de adjacência
- Lista de adjacência (usando apontadores)

Matriz de Adjacência: Estrutura de Dados

- ◆ A inserção de um novo vértice ou retirada de um vértice já existente pode ser realizada com custo constante

```
#define MAXNUMVERTICES 100
#define MAXNUMARESTAS 4500

typedef int TipoValorVertice;
typedef int TipoPeso;
typedef struct TipoGrafo {
    TipoPeso Mat[MAXNUMVERTICES][MAXNUMVERTICES];
    int NumVertices;
    int NumArestas;
} TipoGrafo;
typedef int TipoApontador;
```

Matriz de Adjacência: Operadores

```
void FGVazio(TipoGrafo *Grafo) {  
    short i , j ;  
    for (i = 0; i < Grafo->NumVertices; i++) {  
        for (j = 0; j < Grafo->NumVertices; j ++)  
            Grafo->Mat[ i ] [ j ] = 0;  
    }  
}
```

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

Visão geral da Matriz de Adjacência com 5 vértices

Matriz de Adjacência: Operadores

```
void InsereAresta(TipoValorVertice *V1,
                    TipoValorVertice *V2, TipoPeso *Peso,
                    TipoGrafo *Grafo) {
    Grafo->Mat[*V1] [*V2] = *Peso;
}

short ExisteAresta(TipoValorVertice Vertice1,
                    TipoValorVertice Vertice2,
                    TipoGrafo *Grafo) {
    return (Grafo->Mat[Vertice1] [Vertice2] > 0);
}
```

Matriz de Adjacência: Operadores

```
/* Operadores para obter a lista de adjacentes */

short ListaAdjVazia(TipoValorVertice *Vertice ,
                      TipoGrafo *Grafo) {

    TipoApontador Aux = 0;

    short ListaVazia = TRUE;

    while (Aux < Grafo->NumVertices && ListaVazia)

        if (Grafo->Mat[*Vertice] [Aux] > 0)

            ListaVazia = FALSE;

        else Aux++;

    return (ListaVazia == TRUE) ;

}
```

Matriz de Adjacência: Operadores

```
/* Operadores para obter a lista de adjacentes */

TipoApontador PrimeiroListaAdj (TipoValorVertice
                                *Vertice , TipoGrafo *Grafo) {

    TipoValorVertice Result;
    TipoApontador Aux = 0;
    short ListaVazia = TRUE;

    while (Aux < Grafo->NumVertices && ListaVazia) {
        if (Grafo->Mat[*Vertice ][Aux] > 0) {
            Result = Aux; ListaVazia = FALSE;
        }
        else Aux++;
    }

    if (Aux == Grafo->NumVertices)
        printf( "Erro : Lista adjacencia vazia
                (PrimeiroListaAdj ) \n" );
}

return Result ;
```

Matriz de Adjacência: Operadores

```
/* Operadores para obter a lista de adjacentes */
void ProxAdj (TipoValorVertice *Vertice,
                TipoGrafo *Grafo, TipoValorVertice *Adj ,
                TipoPeso *Peso, TipoApontador *Prox,
                short *FimListaAdj) {

    /* Retorna Adj apontado por Prox */
    *Adj = *Prox;
    *Peso = Grafo->Mat[*Vertice] [*Prox];
    (*Prox)++;
    while (*Prox < Grafo->NumVertices &&
            Grafo->Mat[*Vertice] [*Prox] == 0)
        (*Prox)++;
    if (*Prox == Grafo->NumVertices)
        *FimListaAdj = TRUE;
}
```

Vertice = 1
Prox = 0

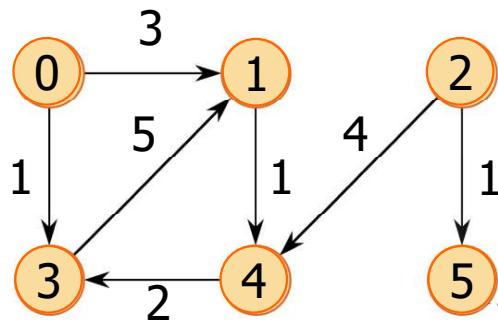
	0	1	2	3	4
0	0	0	0	0	0
1	1	0	0	1	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0

Matriz de Adjacência: Operadores

◆ Trecho de programa para imprimir lista de adjacentes

...

```
if (!ListaAdjVazia(&v, &Grafo)) {  
    Aux = PrimeiroListaAdj(&v, &Grafo);  
    FimListaAdj = FALSE;  
    while (!FimListaAdj) {  
        ProxAdj(&v, &Grafo, &u, &Peso, &Aux,  
                &FimListaAdj);  
        printf("%2d (%d)", Adj, Peso); }  
}
```



```
Lista adjacentes de: 1  
4 (1)
```

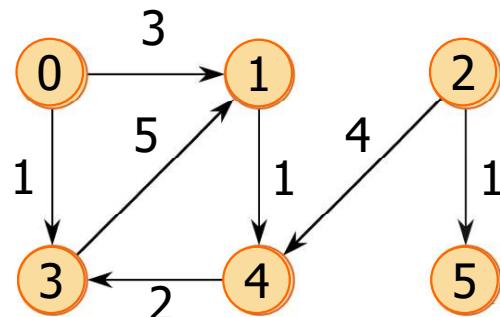
Matriz de Adjacência: Operadores

```
void RetiraAresta(TipoValorVertice *V1,
                    TipoValorVertice *V2,
                    TipoPeso *Peso, TipoGrafo *Grafo) {
    if (Grafo->Mat[*V1] [*V2] == 0)
        printf( "Aresta nao existe \n" ) ;
    else { *Peso = Grafo->Mat[*V1] [*V2] ;
            Grafo->Mat[*V1] [*V2] = 0; }
}
```

```
void LiberaGrafo(TipoGrafo *Grafo) {
    /* Nao faz nada no caso de matrizes de adjacencia */
}
```

Matriz de Adjacência: Operadores

```
void ImprimeGrafo(TipoGrafo *Grafo) {  
    short i , j;  
    printf ( " " );  
    for ( i = 0; i < Grafo->NumVertices; i++)  
        printf ( "%3d" , i );  
    printf ( " \n" ) ;  
    for ( i = 0; i < Grafo->NumVertices; i++)  
    { printf ( "%3d" , i ) ;  
        for ( j = 0; j < Grafo->NumVertices; j++)  
            printf ( "%3d" , Grafo->Mat[ i ] [ j ] ) ;  
        printf ( " \n" ); } }
```



Imprimindo o grafo						
0	1	2	3	4	5	
0	0	3	0	1	0	0
1	0	0	0	0	0	1
2	0	0	0	0	4	1
3	0	5	0	0	0	0
4	0	0	0	2	0	0
5	0	0	0	0	0	0

Listas de Adjacência Usando Apontadores

```
#define MAXNUMVERTICES 100
#define MAXNUMARESTAS 4500

typedef int TipoValorVertice;
typedef int TipoPeso;
typedef struct TipoItem {
    TipoValorVertice Vertice;
    TipoPeso Peso;
} TipoItem;

typedef struct TipoCelula *TipoApontador;
struct TipoCelula {
    TipoItem Item;
    TipoApontador Prox;
} TipoCelula;
```

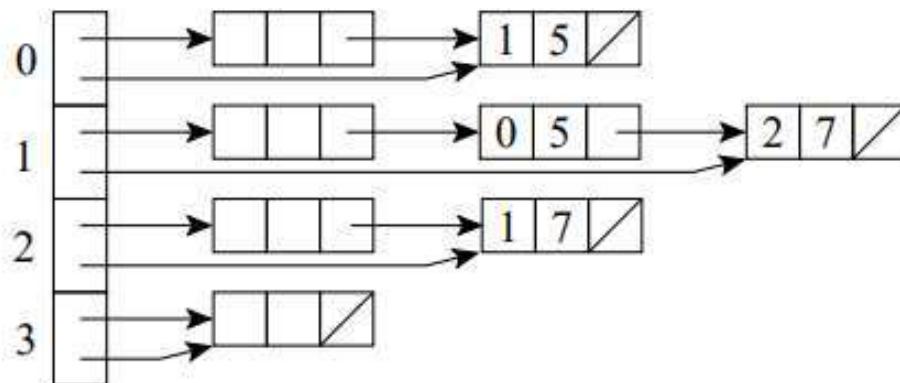
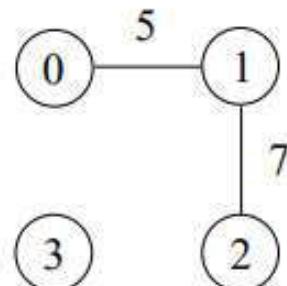
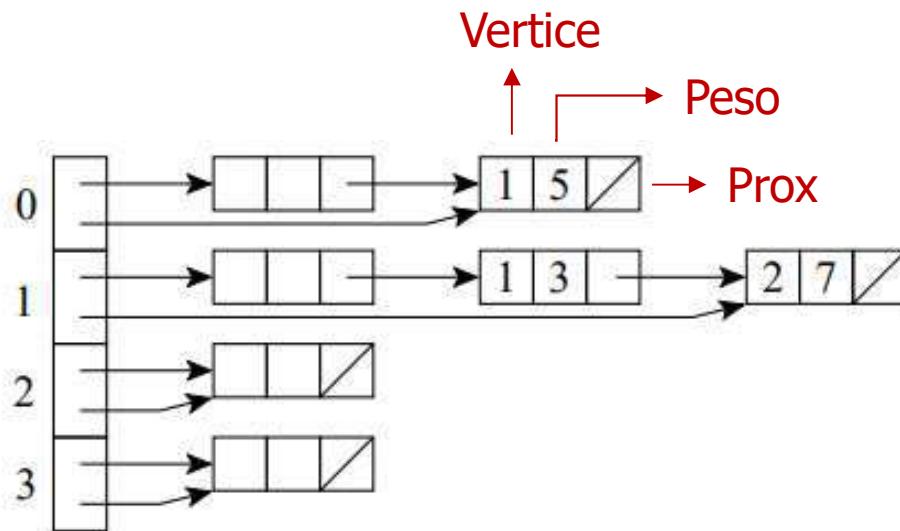
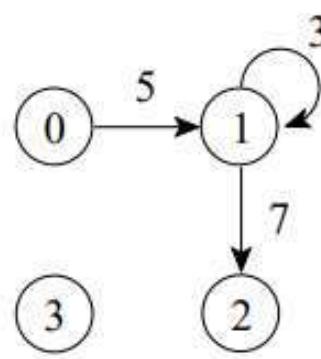
Listas de Adjacência Usando Apontadores

```
typedef struct TipoLista {  
    TipoApontador Primeiro , Ultimo;  
} TipoLista;
```

```
typedef struct TipoGrafo {  
    TipoLista Adj [MAXNUMVERTICES] ;  
    int NumVertices;  
    int NumArestas;  
} TipoGrafo;
```

- ◆ No uso de apontadores a lista é constituída de células, onde cada célula contém um item da lista e um apontador para a célula seguinte

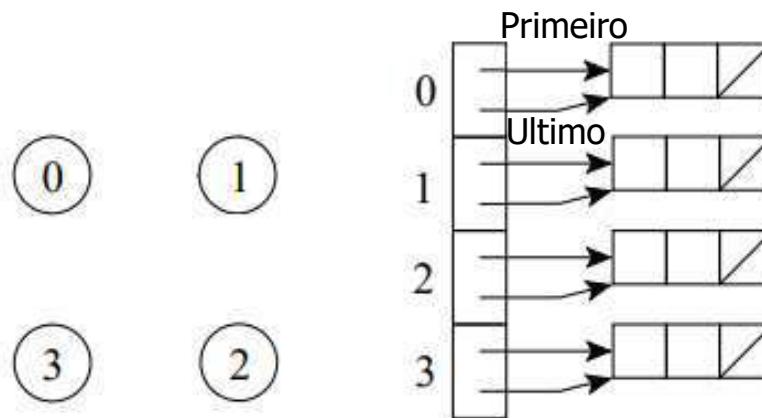
Listas de Adjacência Usando Apontadores



Visão geral da Lista de Adjacência para dois grafos exemplo

Listas de Adjacência Usando Apontadores: Operadores

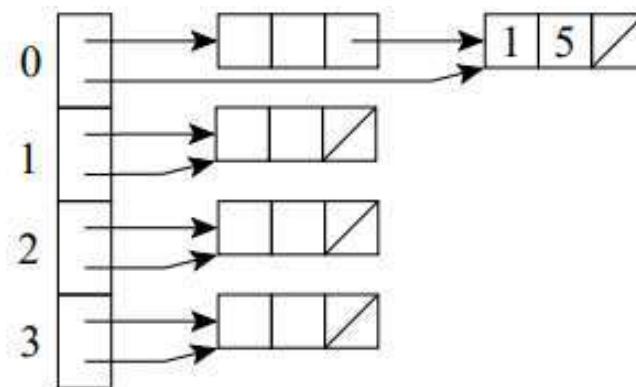
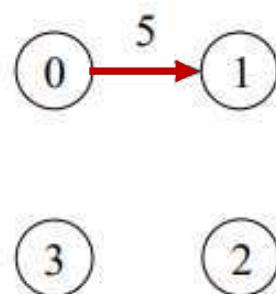
```
/*--Entram aqui os operadores FLVazia, Vazia,  
    Insere, Retira e Imprime--*/  
/*--do TAD Lista de Apontadores do Programa 3.4 --*/  
  
void FGVazio(TipoGrafo *Grafo) {  
    long i;  
    for ( i = 0; i < Grafo->NumVertices; i ++)  
        FLVazia(&Grafo->Adj [ i ] ) ;  
}
```



Exemplo: para cada vértice iniciar a lista de adjacência

Listas de Adjacência Usando Apontadores: Operadores

```
void InsereAresta(TipoValorVertice *V1,  
                    TipoValorVertice *V2,  
                    TipoPeso *Peso, TipoGrafo *Grafo) {  
    TipoItem x;  
    x.Vertice = *V2;  
    x.Peso = *Peso;  
    Insere(&x, &Grafo->Adj[*V1] ) ;  
}
```



Listas de Adjacência Usando Apontadores: Operadores

```
short ExisteAresta(TipoValorVertice Vertice1,  
                    TipoValorVertice Vertice2 ,  
                    TipoGrafo *Grafo) {  
  
    TipoApontador Aux;  
  
    short EncontrouAresta = FALSE;  
  
    Aux = Grafo->Adj [Vertice1].Primeiro->Prox;  
  
    while (Aux != NULL && EncontrouAresta == FALSE) {  
  
        if (Vertice2 == Aux->Item.Vertice )  
            EncontrouAresta = TRUE;  
  
        Aux = Aux->Prox;  
    }  
  
    return EncontrouAresta;  
}
```

Listas de Adjacência Usando Apontadores: Operadores

```
/* Operadores para obter a lista de adjacentes */
short ListaAdjVazia(TipoValorVertice *Vertice ,
TipoGrafo *Grafo) {
    return (Grafo->Adj [*Vertice].Primeiro ==
            Grafo->Adj [*Vertice] .Ultimo);
}
```

```
TipoApontador PrimeiroListaAdj (TipoValorVertice
*Vertice , TipoGrafo *Grafo) {
    return (Grafo->Adj [*Vertice].Primeiro->Prox );
}
```

Listas de Adjacência Usando Apontadores: Operadores

```
/* Operadores para obter a lista de adjacentes */
void ProxAdj(TipoValorVertice *Vertice ,
                TipoGrafo *Grafo,
                TipoValorVertice *Adj, TipoPeso *Peso,
                TipoApontador *Prox,
                short *FimListaAdj) {

    /* Retorna Adj e Peso do Item apontado por Prox */
    *Adj = (*Prox)->Item.Vertice ;
    *Peso = (*Prox)->Item.Peso;
    *Prox = (Prox)->Prox;
    if (*Prox == NULL)
        *FimListaAdj = TRUE;
}
```

Listas de Adjacência Usando Apontadores: Operadores

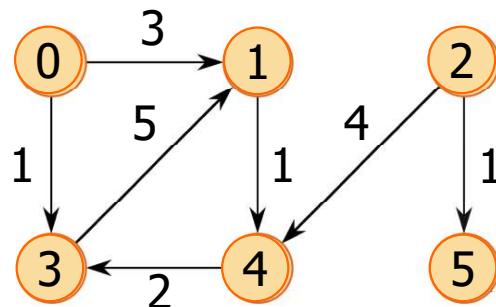
```
void LiberaGrafo(TipoGrafo *Grafo) {  
    TipoApontador AuxAnterior , Aux;  
    for ( i = 0; i < Grafo->NumVertices; i++)  
    { Aux = Grafo->Adj[i].Primeiro->Prox;  
        free(Grafo->Adj[i].Primeiro );  
        /*Libera celula cabeca*/  
        Grafo->Adj[i].Primeiro=NULL;  
        while (Aux != NULL) {  
            AuxAnterior = Aux;  
            Aux = Aux->Prox;  
            free(AuxAnterior);  
        }  
    }  
    Grafo->NumVertices = 0; }
```

Listas de Adjacência Usando Apontadores: Operadores

```
void ImprimeGrafo(TipoGrafo *Grafo) {  
    int i ;  
    TipoApontador Aux;  
    for (i = 0; i < Grafo->NumVertices; i++) {  
        printf ("Vertice%2d: " , i );  
        if (!Vazia(Grafo->Adj[i])) {  
            Aux = Grafo->Adj[i].Primeiro->Prox;  
            while (Aux != NULL) {  
                printf ("%3d (%d) " , Aux->Item.Vertice,  
                        Aux->Item.Peso) ;  
                Aux = Aux->Prox;  
            }  
        }  
        putchar( ' \n ' ) ;    } }
```

Listas de Adjacência Usando Apontadores: Operadores

Exemplo de impressão



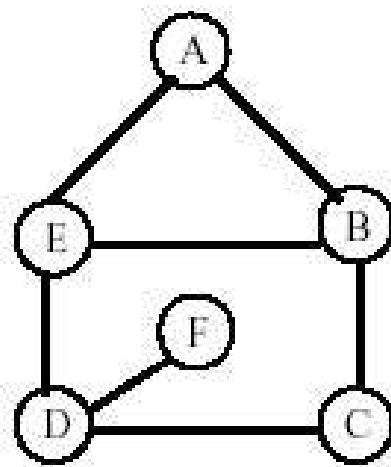
```
Vertice 0: 1 (3) 3 (1)
Vertice 1: 4 (1)
Vertice 2: 4 (4) 5 (1)
Vertice 3: 1 (5)
Vertice 4: 3 (2)
Vertice 5:
```

Exemplo de execução

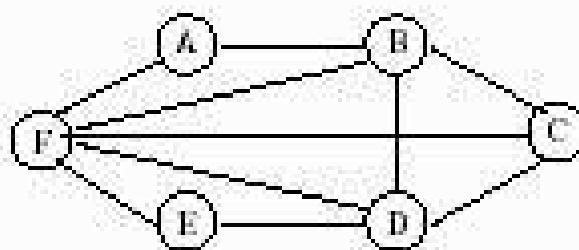
- ◆ Executar códigos exemplo no Replit
 - <https://replit.com/@mcamilab/Exemplo-TAD-Grafos-Matriz-Adjacencia>
 - <https://replit.com/@mcamilab/Exemplo-TAD-Grafos-Lista-Adjacencia>

Exercícios

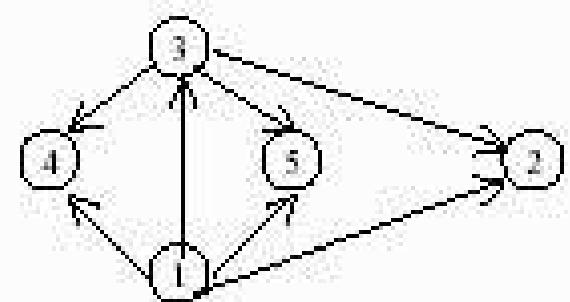
Determine a matriz de adjacência e a lista de adjacência para os grafos abaixo.



a)



b)



c)

Bibliografia

Ziviani, N. Projeto de Algoritmos, Thomson Learning, 2005.

Ebook da edição de 2018 disponível na biblioteca
<https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788522126590>

Ascencio, A. F. G.; Araújo, G. S. Estruturas de Dados, Pearson, 2010.

Bacharelado em Ciência da Computação
GBC034 Algoritmos e Estruturas de Dados 2

Grafos

Busca

Material baseado nos slides do professor Nivio Ziviani
(Projeto de Algoritmos)

Profa. Maria Camila Nardini Barioni

camila.barioni@ufu.br

Bloco B - sala 1B137

2º semestre de 2023

Roteiro

- ◆ Busca em Profundidade
- ◆ Busca em Largura

Busca em profundidade

- ◆ A busca em profundidade, (do inglês *depth-first search*), é um algoritmo para caminhar no grafo
- ◆ A estratégia é buscar o mais profundo no grafo sempre que possível
- ◆ As arestas são exploradas a partir do vértice v mais recentemente descoberto que ainda possui arestas não exploradas saindo dele

Busca em profundidade

- ◆ Quando todas as arestas adjacentes a v tiverem sido exploradas a busca anda para trás (do inglês *backtrack*) para explorar vértices que saem do vértice do qual v foi descoberto
- ◆ O algoritmo é a **base para muitos outros algoritmos importantes**, tais como verificação de grafos acíclicos, ordenação topológica e componentes fortemente conectados

Busca em profundidade

- ◆ Para acompanhar o progresso do algoritmo cada vértice é colorido de branco, cinza ou preto
- ◆ Todos os vértices são inicializados branco
- ◆ Quando um vértice é descoberto pela primeira vez ele torna-se cinza, e é tornado preto quando sua lista de adjacentes tenha sido completamente examinada
- ◆ $d[v]$: tempo de descoberta
- ◆ $t[v]$: tempo de término do exame da lista de adjacentes de v
- ◆ Estes registros são inteiros entre 1 e $2|V|$ pois existe um evento de descoberta e um evento de término para cada um dos $|V|$ vértices

Busca em profundidade

Implementação

```
void BuscaEmProfundidade(TipoGrafo *Grafo)
{ TipoValorVertice x;
  TipoValorTempo Tempo;
  TipoValorTempo d[MAXNUMVERTICES + 1], t[MAXNUMVERTICES + 1];
  TipoCor Cor[MAXNUMVERTICES+1];
  short Antecessor [MAXNUMVERTICES+1];
  Tempo = 0; for (x = 0; x <= Grafo->NumVertices - 1; x++) {
    Cor[x] = branco;
    Antecessor[x] = -1;
  }
for (x = 0; x <= Grafo->NumVertices - 1; x++) {
  if (Cor[x] == branco)
    VisitaDfs(x, Grafo, &Tempo, d, t, Cor, Antecessor);
}
}

verifica cada vértice e quando
encontra um vértice branco
visita esse vértice
```

usada para registrar os tempos de descoberta e término

inicializa os vértices de branco e os antecessores para -1

Busca em profundidade

Implementação

```
void BuscaEmProfundidade(TipoGrafo *Grafo)
{ TipoValorVertice x;
  TipoValorTempo Tempo;
  TipoValorTempo d[MAXNUMVERTICES + 1], t[MAXNUMVERTICES + 1];
  Tip...  
show...  
Tempo  
for...  
}
  for (x = 0; x <= Grafo->NumVertices - 1; x++) {
    if (Cor[x] == branco)
      VisitaDfs(x, Grafo, &Tempo, d, t, Cor, Antecessor);
  }
}
```

Toda vez que $VisitaDfs(u)$ é chamado, o vértice u se torna a raiz de uma nova **árvore de busca em profundidade**, e o conjunto de árvores forma uma **floresta** de árvores de busca

verifica cada vértice e quando encontra um vértice branco visita esse vértice

Busca em profundidade

Implementação

```
void VisitaDfs(TipoValorVertice u, TipoGrafo *Grafo,
                TipoValorTempo* Tempo, TipoValorTempo* d,
                TipoValorTempo* t, TipoCor* Cor, short* Antecessor)
{ char FimListaAdj; TipoValorAresta Peso; TipoApontador Aux;
  TipoValorVertice v; Cor[u] = cinza; (*Tempo)++; d[u] = (*Tempo);
  printf("Visita%2d Tempo descoberta:%2d cinza\n", u, d[u]); getchar();
  if (!ListaAdjVazia(&u, Grafo))
  { Aux = PrimeiroListaAdj(&u, Grafo);
    FimListaAdj = FALSE;
    while (!FimListaAdj) {
      ProxAdj(&u, &v, &Peso, &Aux, &FimListaAdj);
      if (Cor[v] == branco) {
        Antecessor[v] = u;
        VisitaDfs(v, Grafo, Tempo, d, t, Cor, Antecessor);
      }
    }
  }
  Cor[u] = preto; (*Tempo)++; t[u] = (*Tempo);
  printf("Visita%2d Tempo termino:%2d preto\n", u, t[u]); getchar();
}
```

Busca em profundidade

Implementação

u é tornado cinza

```
void VisitaDfs(TipoValorVertice u, TipoGrafo *Grafo,
                 TipoValorTempo* Tempo, TipoValorTempo* d,
                 TipoValorTempo* t, TipoCor* Cor, short* Antecessor)
{ char FimListaAdj; TipoValorAresta Peso; TipoApontador Aux;
  TipoValorVertice v; Cor[u] = cinza; (*Tempo)++; d[u] = (*Tempo);
  printf("Visita%2d Tempo descoberta:%2d cinza\n", u, d[u]); getchar();
  if (!ListaAdjVazia(&u, Grafo))
  { Aux = PrimeiroListaAdj(&u, Grafo);
    FimListaAdj = FALSE;
    while (!FimListaAdj) {
      ProxAdj(&u, &v, &Peso, &Aux, &FimListaAdj);
      if (Cor[v] == branco) {
        Antecessor[v] = u;
        VisitaDfs(v, Grafo, Tempo, d, t, Cor, Antecessor);
      }
    }
  }
  Cor[u] = preto; (*Tempo)++; t[u] = (*Tempo);
  printf("Visita%2d Tempo termino:%2d preto\n", u, t[u]); getchar();
}
```

Busca em profundidade

Implementação

A variável *Tempo* é incrementada

```
void VisitaDfs(TipoValorVertice u, TipoGrafo *Grafo,
                 TipoValorTempo* Tempo, TipoValorTempo* d,
                 TipoValorTempo* t, TipoCor* Cor, short* Antecessor)
{ char FimListaAdj; TipoValorAresta Peso; TipoApontador Aux;
  TipoValorVertice v; Cor[u] = cinza; (*Tempo)++; d[u] = (*Tempo);
  printf("Visita%2d Tempo descoberta:%2d cinza\n", u, d[u]); getchar();
  if (!ListaAdjVazia(&u, Grafo))
  { Aux = PrimeiroListaAdj(&u, Grafo);
    FimListaAdj = FALSE;
    while (!FimListaAdj) {
      ProxAdj(&u, &v, &Peso, &Aux, &FimListaAdj);
      if (Cor[v] == branco) {
        Antecessor[v] = u;
        VisitaDfs(v, Grafo, Tempo, d, t, Cor, Antecessor);
      }
    }
  Cor[u] = preto; (*Tempo)++; t[u] = (*Tempo);
  printf("Visita%2d Tempo termino:%2d preto\n", u, t[u]); getchar();
}
```

Busca em profundidade

Implementação

```
void VisitaDfs(TipoValorVertice u, TipoGrafo *Grafo
                TipoValorTempo* Tempo, TipoValorTempo* d,
                TipoValorTempo* t, TipoCor* Cor, short* Antecessor)
{ char FimListaAdj; TipoValorAresta Peso; TipoApontador Aux;
  TipoValorVertice v; Cor[u] = cinza; (*Tempo)++; d[u] = (*Tempo);
  printf("Visita%2d Tempo descoberta:%2d cinza\n", u, d[u]); getchar();
  if (!ListaAdjVazia(&u, Grafo))
  { Aux = PrimeiroListaAdj(&u, Grafo);
    FimListaAdj = FALSE;
    while (!FimListaAdj) {
      ProxAdj(&u, &v, &Peso, &Aux, &FimListaAdj);
      if (Cor[v] == branco) {
        Antecessor[v] = u;
        VisitaDfs(v, Grafo, Tempo, d, t, Cor, Antecessor);
      }
    }
  Cor[u] = preto; (*Tempo)++; t[u] = (*Tempo);
  printf("Visita%2d Tempo termino:%2d preto\n", u, t[u]); getchar();
}
```

→ E o novo valor de *Tempo* é registrado como o tempo de descoberta $d[u]$

Busca em profundidade

Implementação

```
void VisitaDfs(TipoValorVertice u, TipoGrafo *Grafo,
                TipoValorTempo* Tempo, TipoValorTempo* d,
                TipoValorTempo* t, TipoCor* Cor, short* Antecessor)
{ char FimListaAdj; TipoValorAresta Peso; TipoApontador Aux;
  TipoValorVertice v; Cor[u] = cinza; (*Tempo)++; d[u] = (*Tempo);
  printf("Visita%2d Tempo descoberta:%2d cinza\n", u, d[u]); getchar();
  if (!ListaAdjVazia(&u, Grafo))
  { Aux = PrimeiroListaAdj(&u, Grafo);
    FimListaAdj = FALSE;
    while (!FimListaAdj) {
      ProxAdj(&u, &v, &Peso, &Aux, &FimListaAdj);
      if (Cor[v] == branco) {
        Antecessor[v] = u;
        VisitaDfs(v, Grafo, Tempo, d, t, Cor, Antecessor);
      }
    }
  }
  Cor[u] = preto; (*Tempo)++; t[u] = (*Tempo);
  printf("Visita%2d Tempo termino:%2d preto\n", u, t[u]); getchar();
}
```

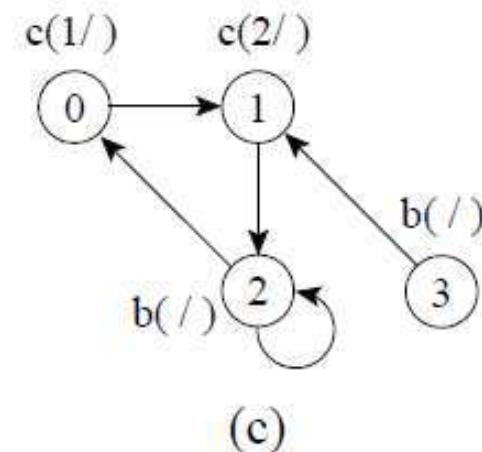
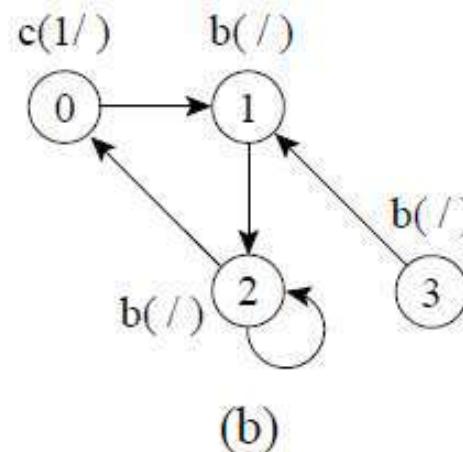
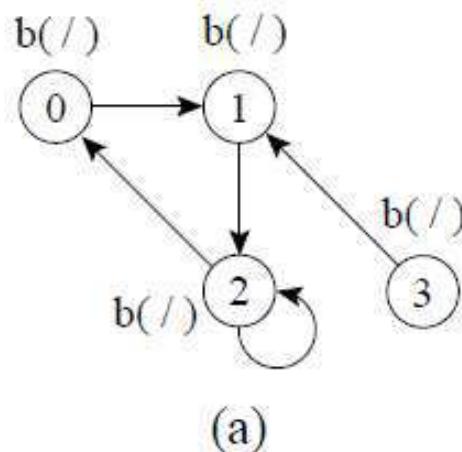
Examina lista de vértices adjacentes visitando-os recursivamente se eles forem brancos

Busca em profundidade

Implementação

```
void VisitaDfs(TipoValorVertice u, TipoGrafo *Grafo,
                TipoValorTempo* Tempo, TipoValorTempo* d,
                TipoValorTempo* t, TipoCor* Cor, short* Antecessor)
{ char FimListaAdj; TipoValorAresta Peso; TipoApontador Aux;
  TipoValorVertice v; Cor[u] = cinza; (*Tempo)++; d[u] = (*Tempo);
  printf("Visita%2d Tempo descoberta:%2d cinza\n", u, d[u]); getchar();
  if (!ListaAdjVazia(&u, Grafo))
  {
    Quando VisitaDfs retorna, cada vértice  $u$  possui
    um tempo de descoberta  $d[u]$  e um tempo de
    término  $t[u]$ 
    Antecessor[v] = u;
    VisitaDfs(v, Grafo, Tempo, d, t, Cor, Antecessor);
  }
}
Cor[u] = preto; (*Tempo)++; t[u] = (*Tempo);
printf("Visita%2d Tempo termino:%2d preto\n", u, t[u]); getchar();
}
```

Exemplo

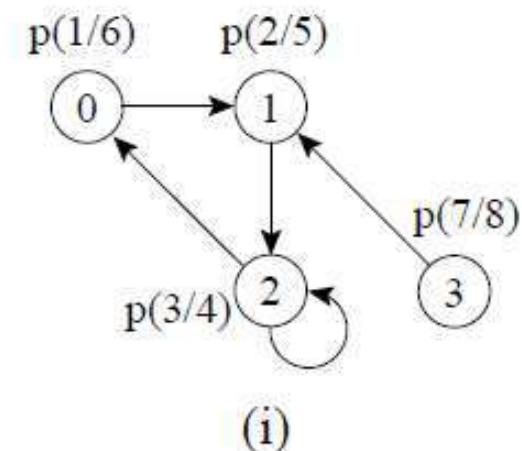
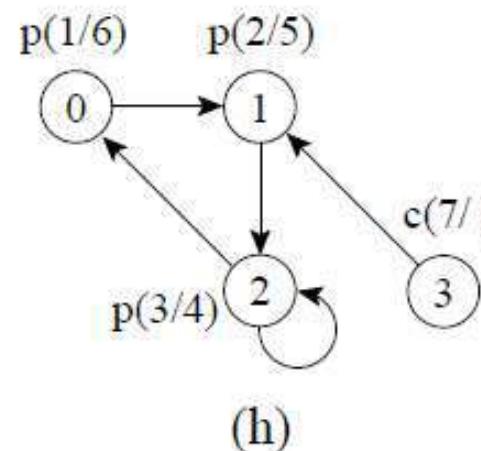
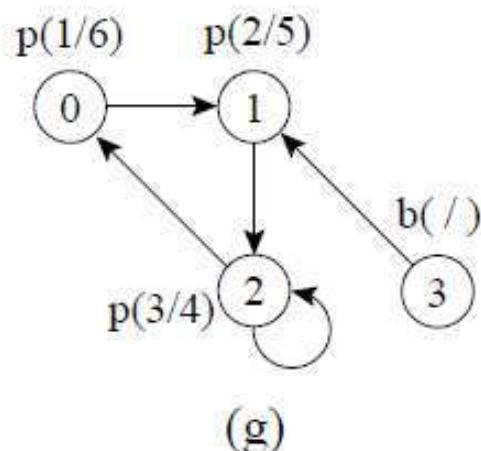


Ilustra o progresso da busca em profundidade.
Ao lado de cada vértice é mostrada a cor branca, cinza ou preta (b, c ou p), e entre parênteses
tempo-de-descoberta/tempo-de-término

(d)

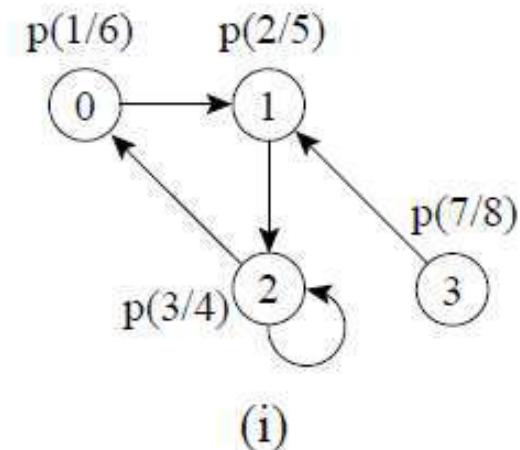
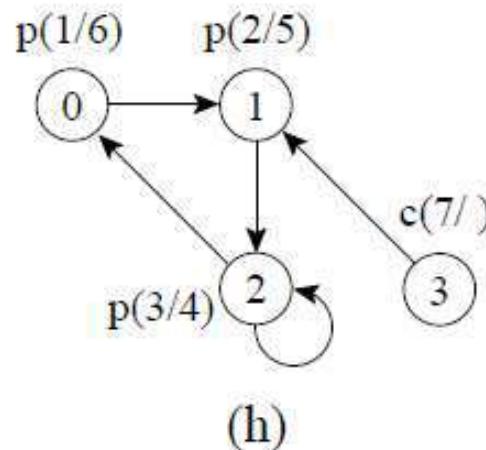
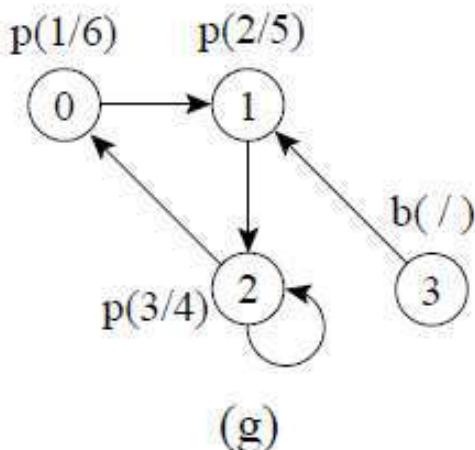
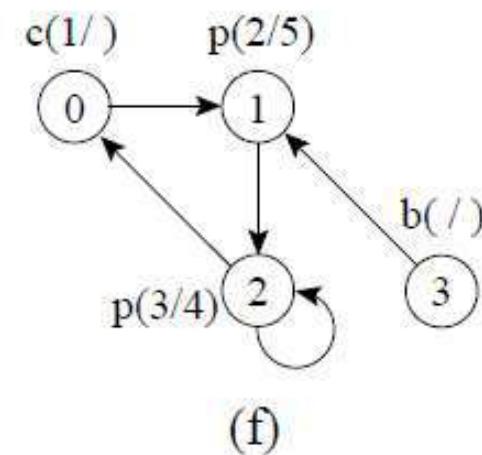
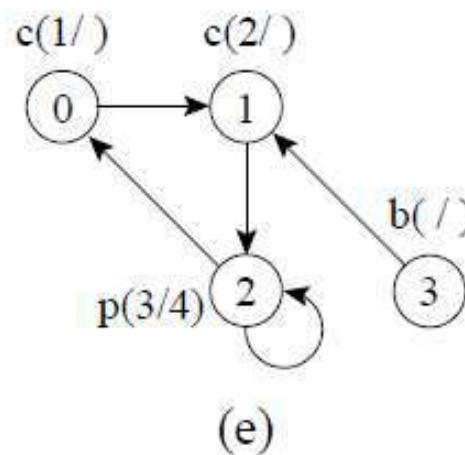
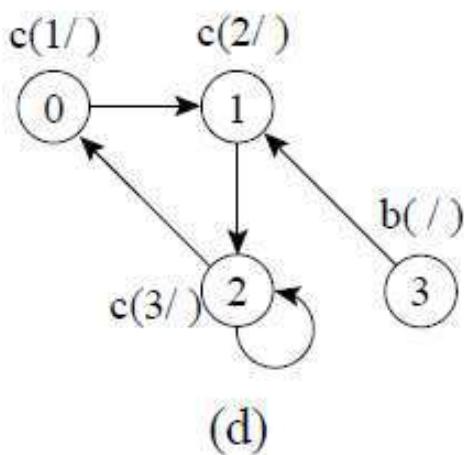
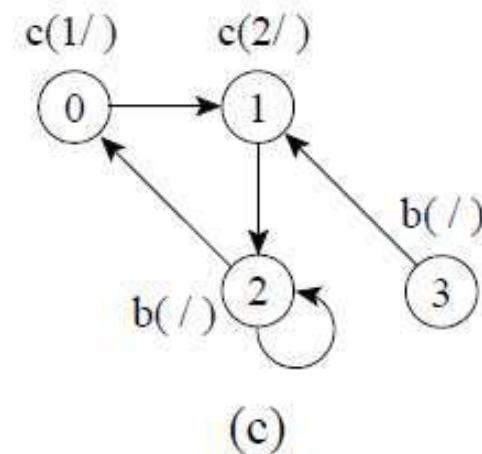
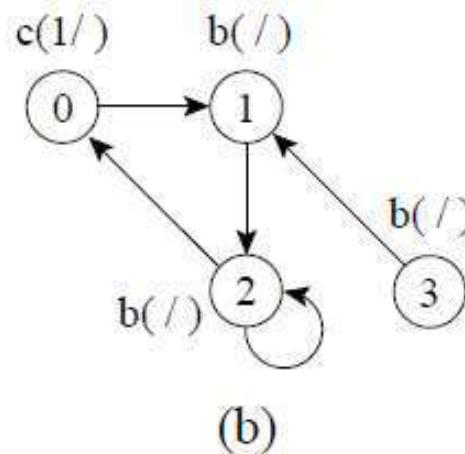
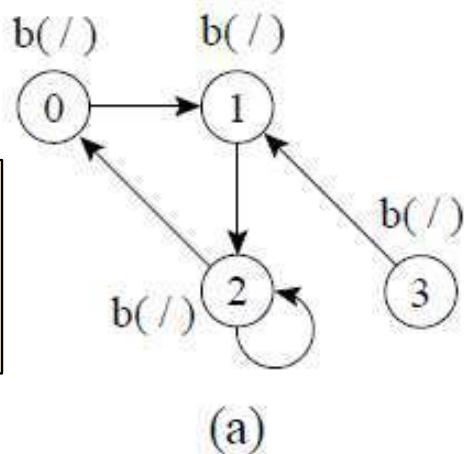
(e)

(f)

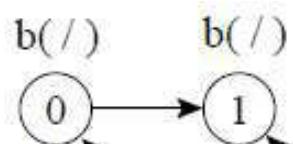


Exemplo

inicializa os
vértices de
branco

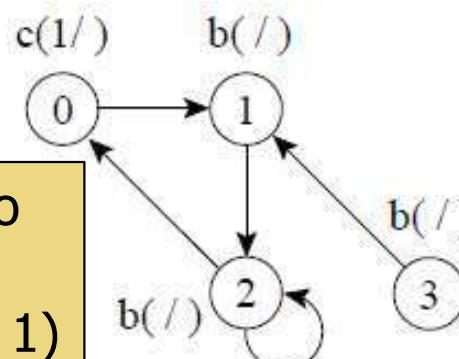


Exemplo

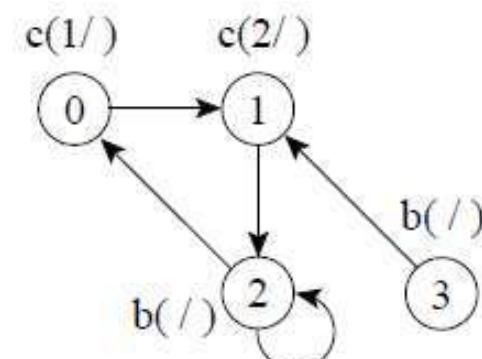


Vértice 0 é tornado
cinza $d[0] = 1$
Visita adj. (vértice 1)

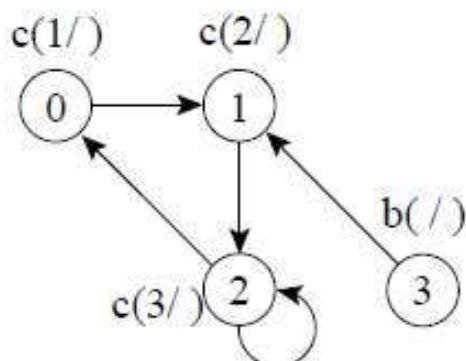
(a)



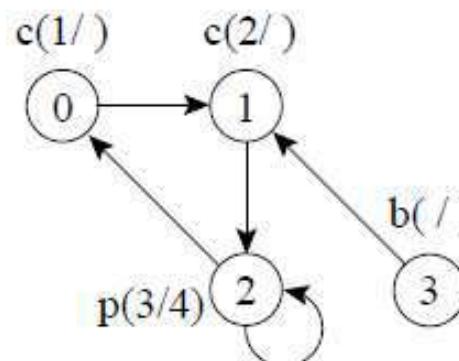
(b)



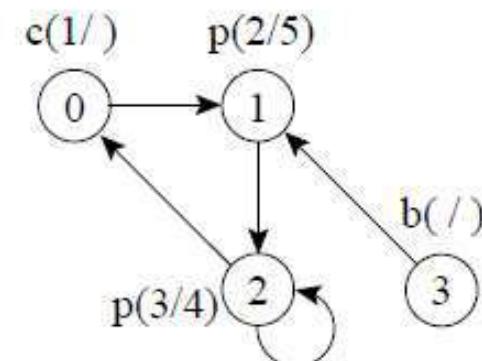
(c)



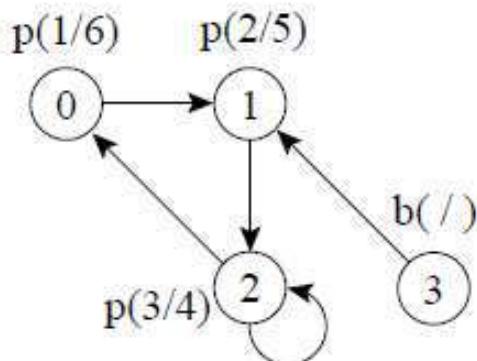
(d)



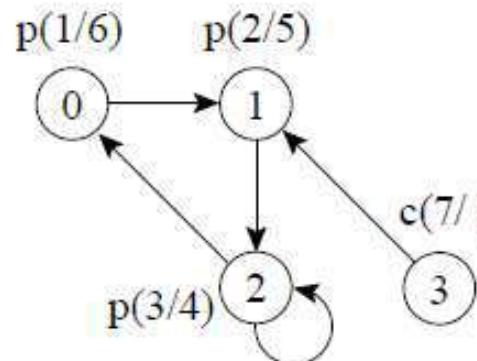
(e)



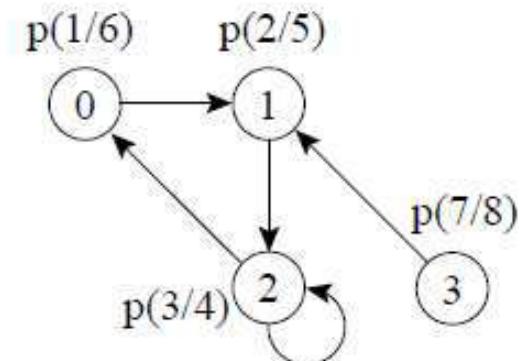
(f)



(g)

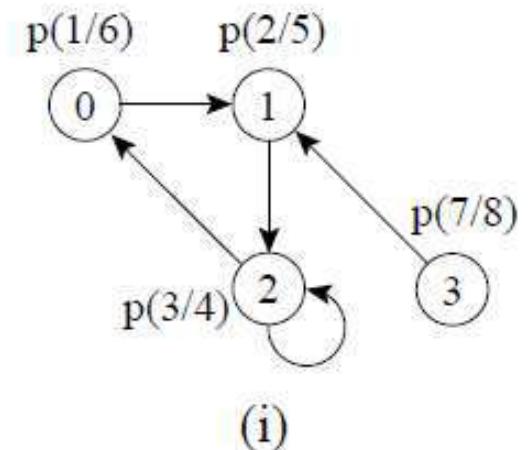
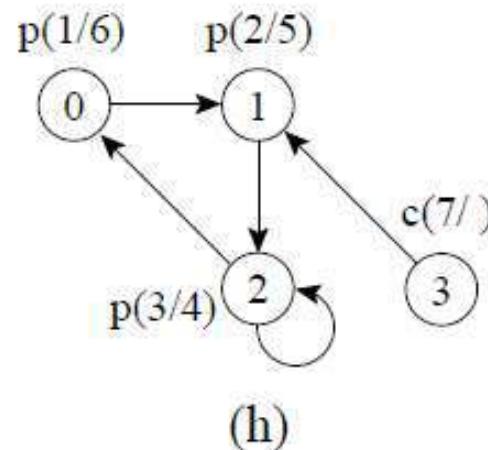
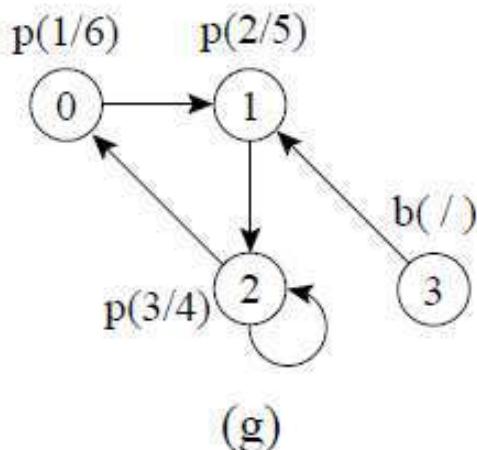
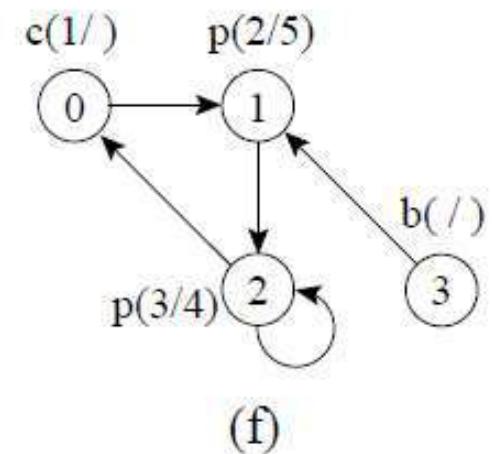
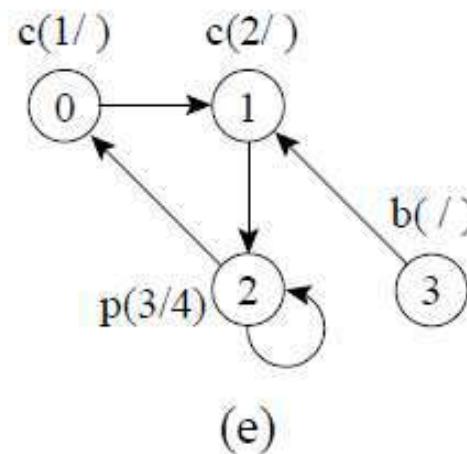
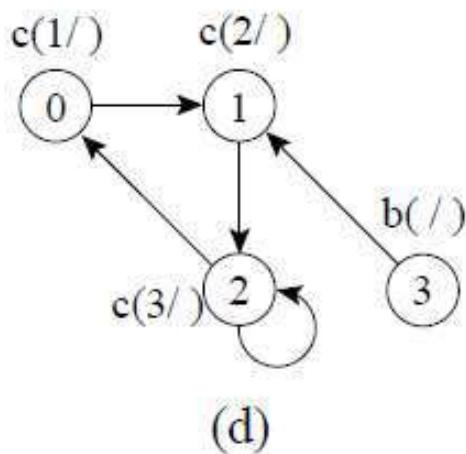
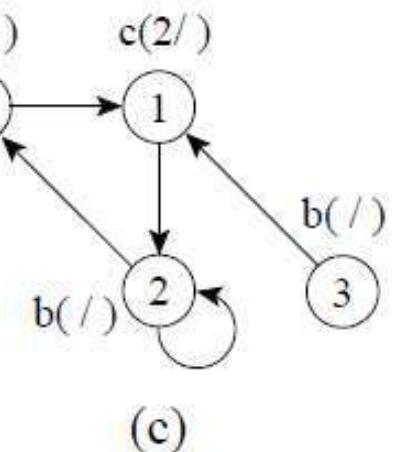
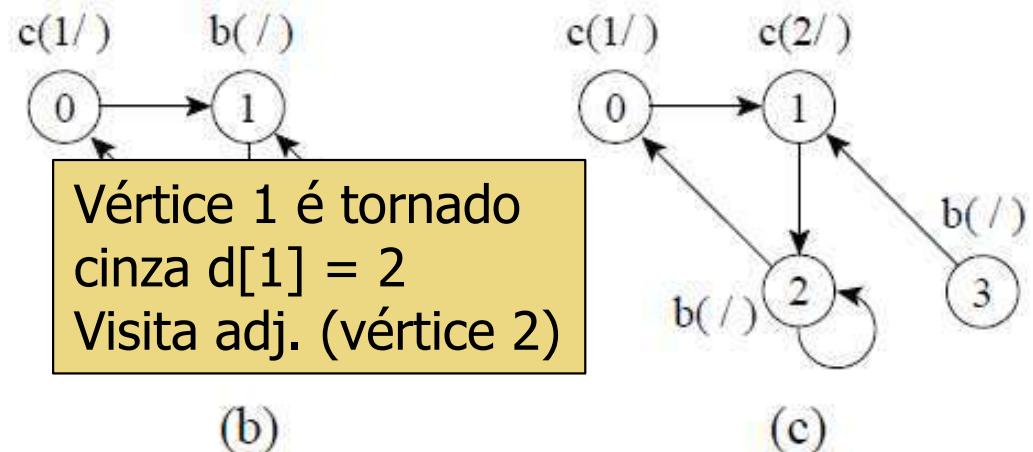
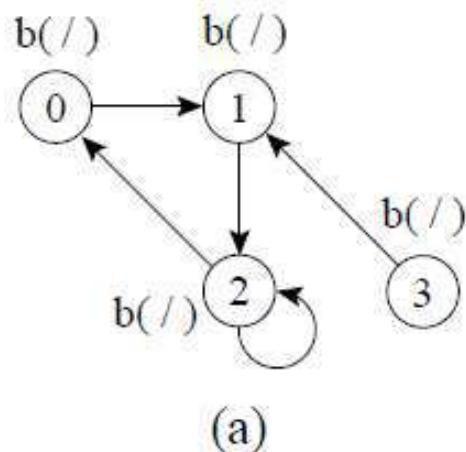


(h)

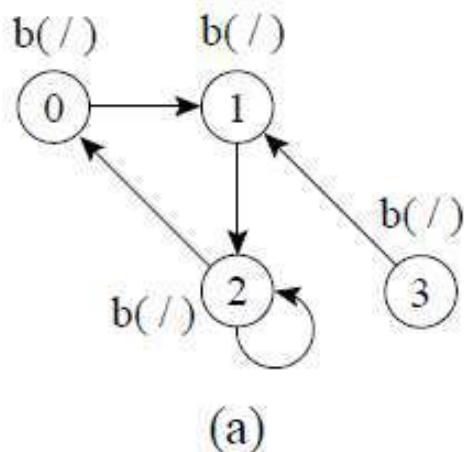


(i)

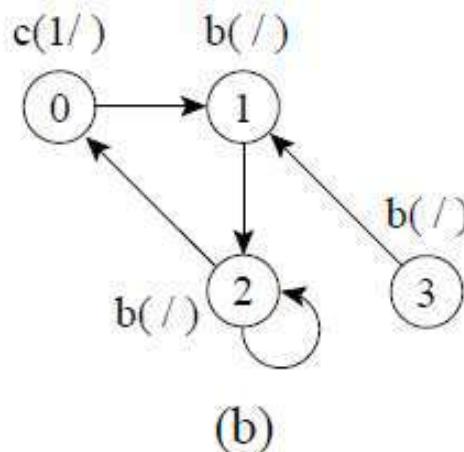
Exemplo



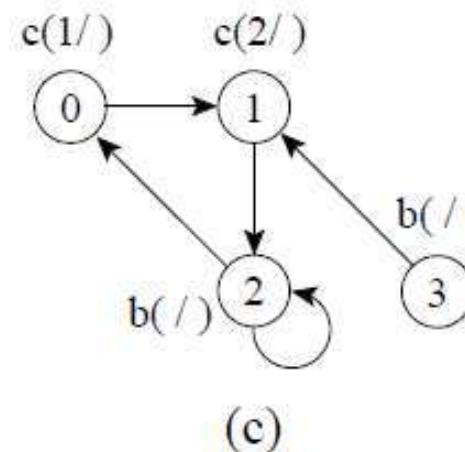
Exemplo



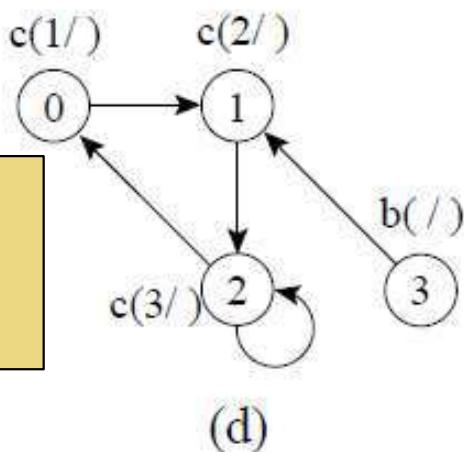
(a)



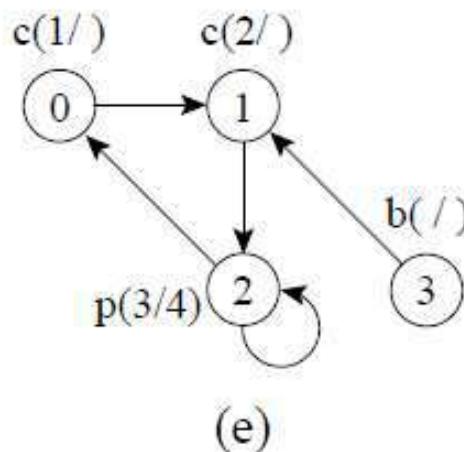
(b)



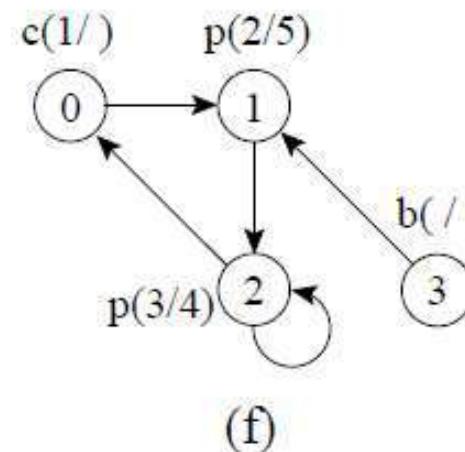
(c)



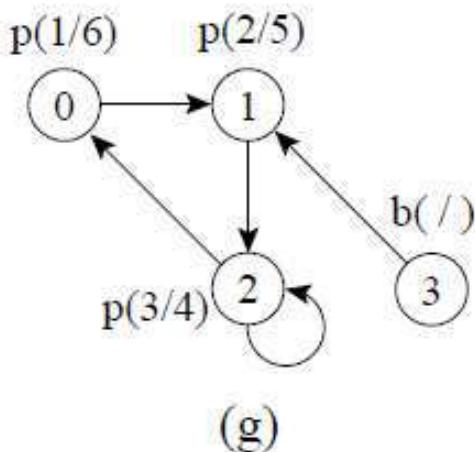
(d)



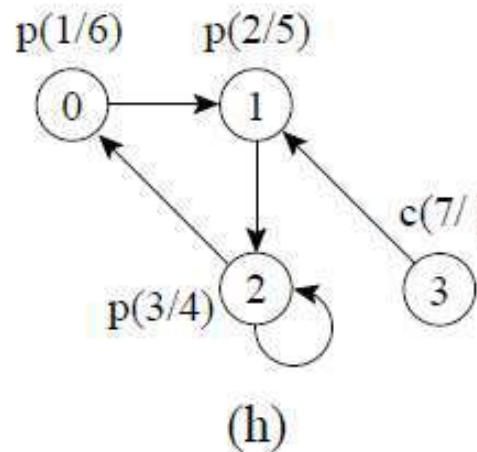
(e)



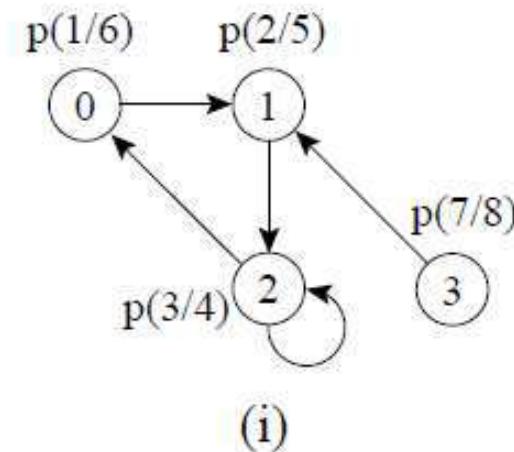
(f)



(g)



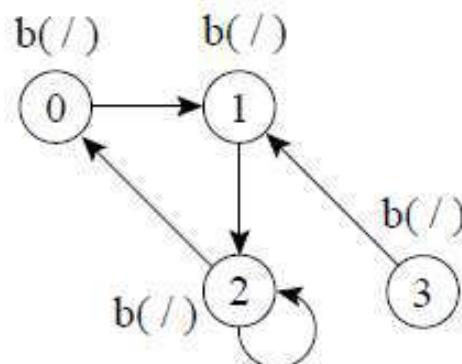
(h)



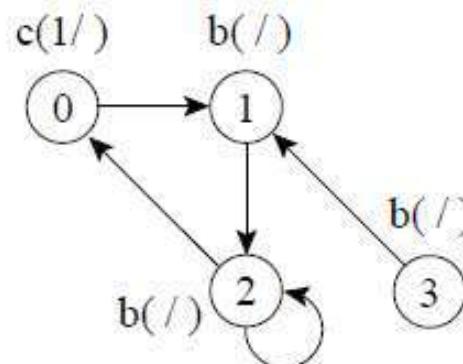
(i)

Vértice 2 é
tornado cinza
 $d[2] = 3$

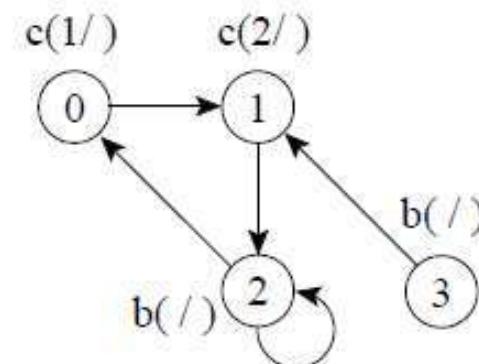
Exemplo



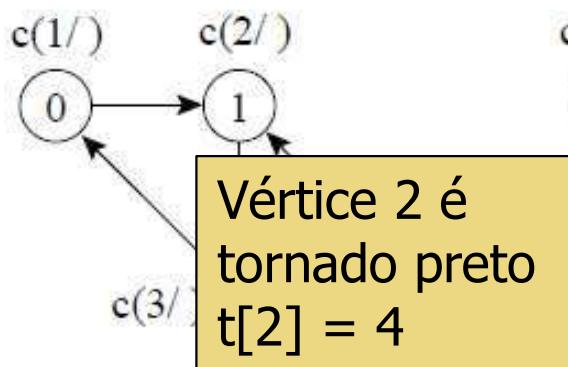
(a)



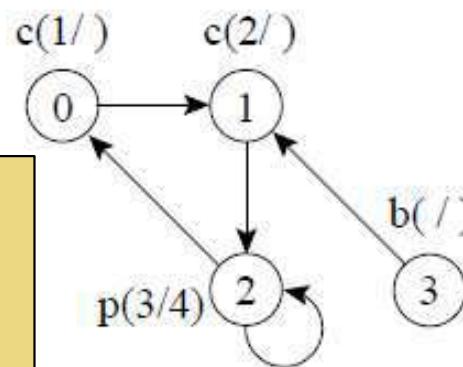
(b)



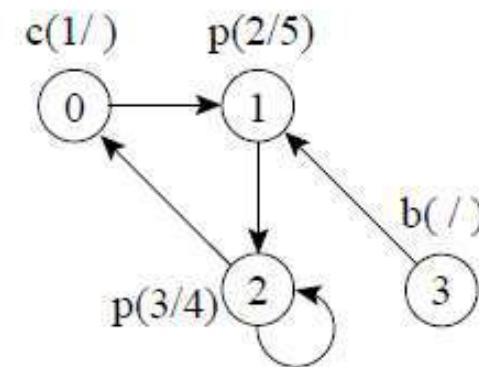
(c)



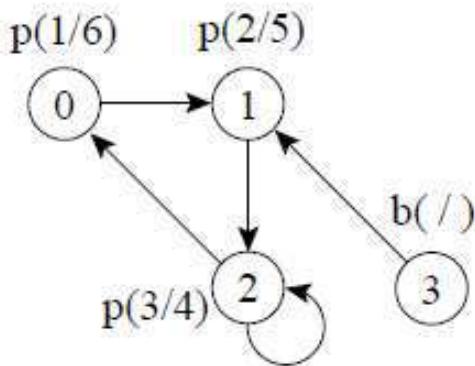
(d)



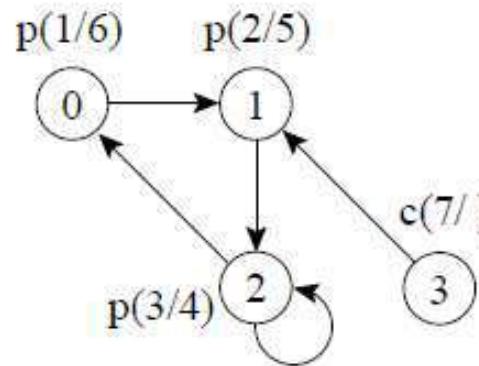
(e)



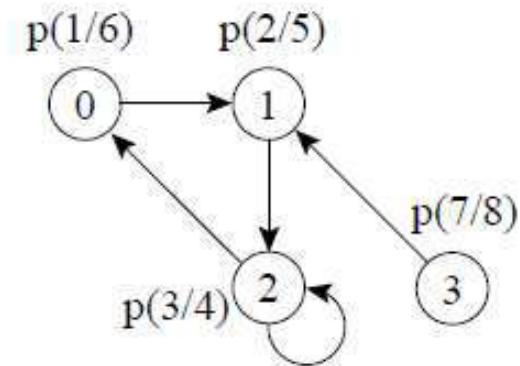
(f)



(g)

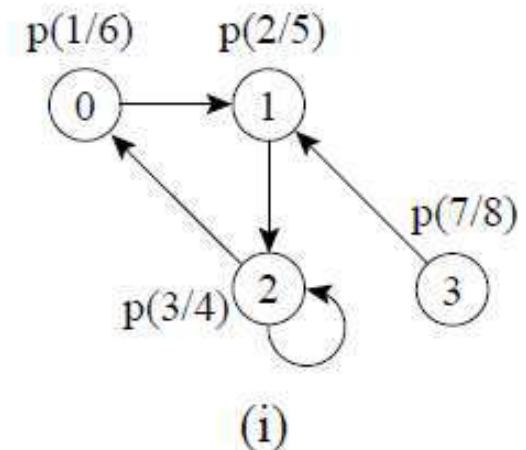
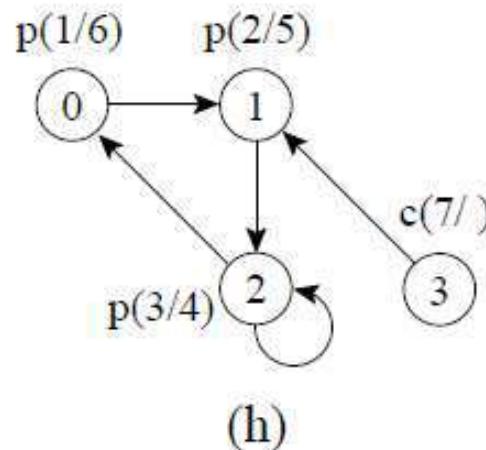
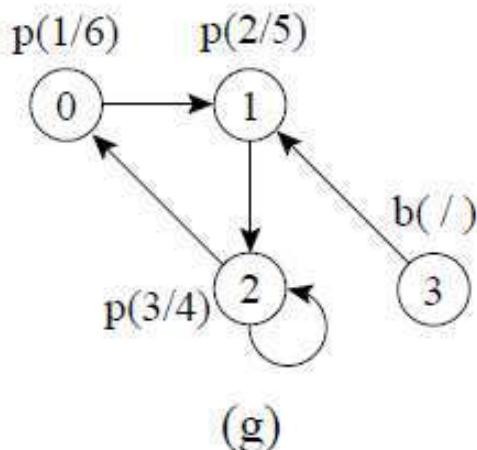
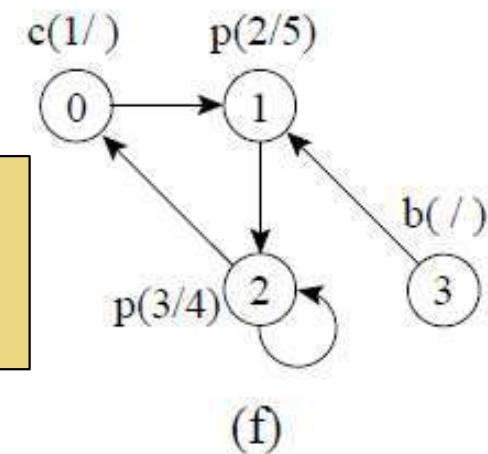
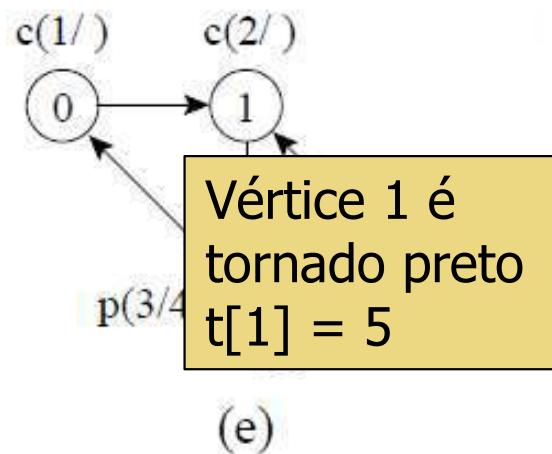
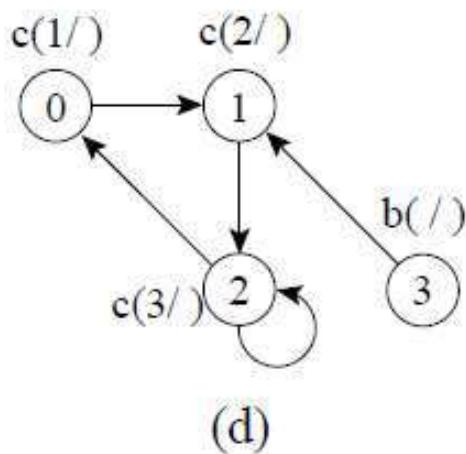
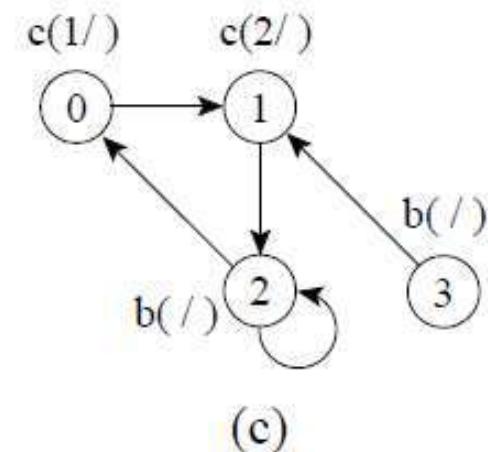
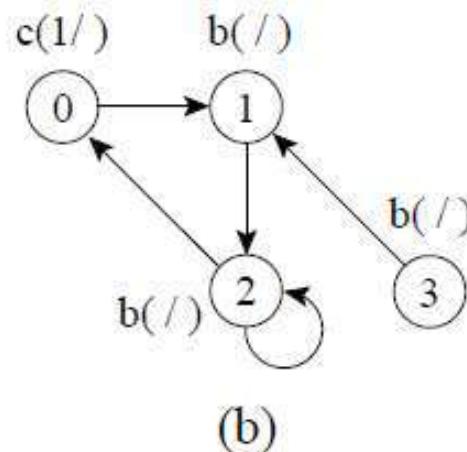
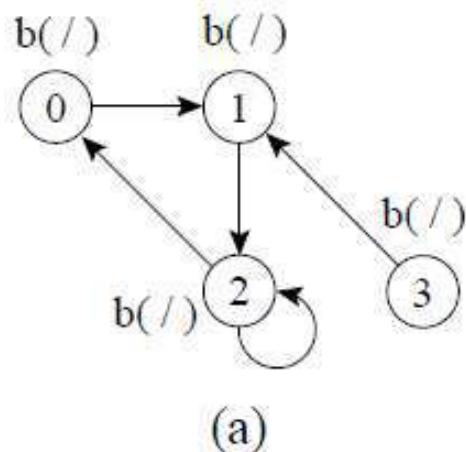


(h)

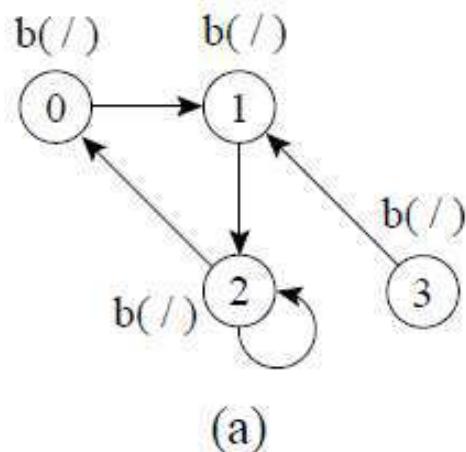


(i)

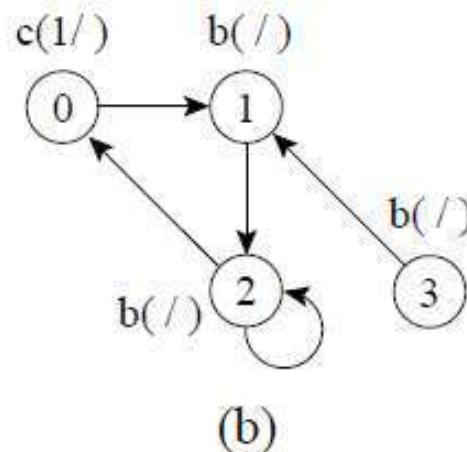
Exemplo



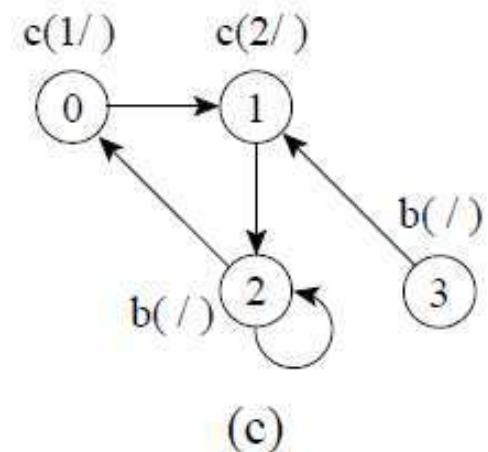
Exemplo



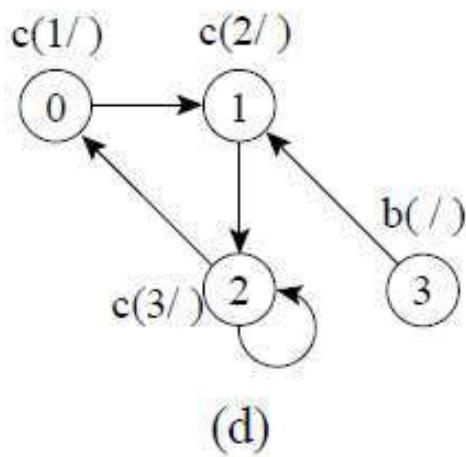
(a)



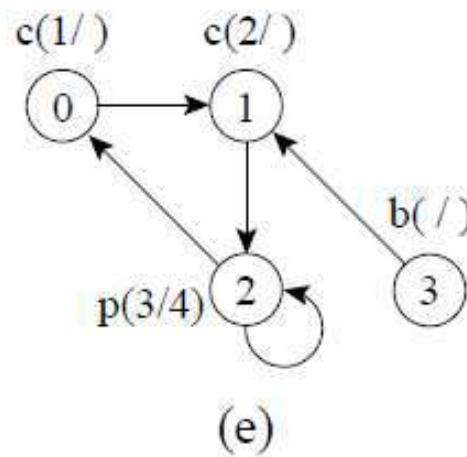
(b)



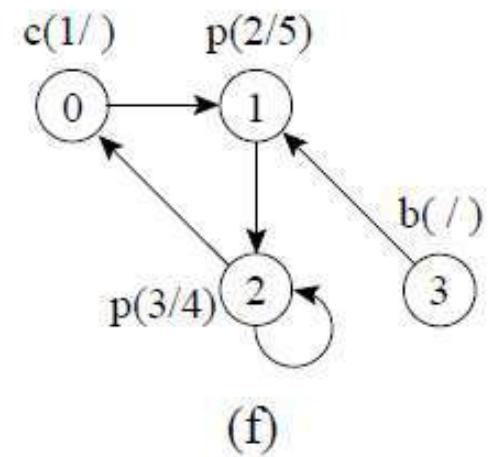
(c)



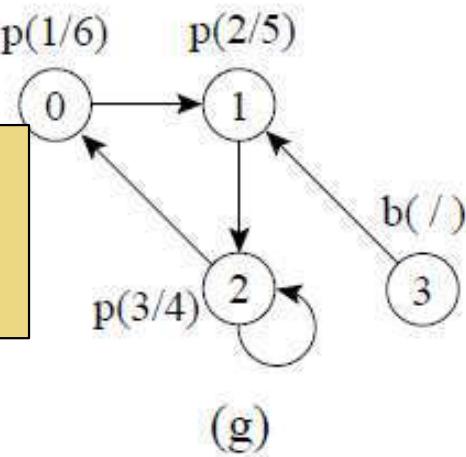
(d)



(e)

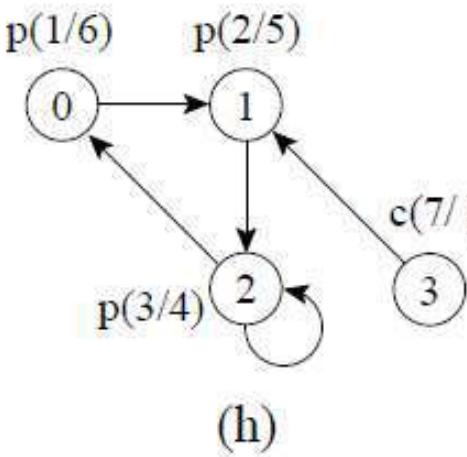


(f)

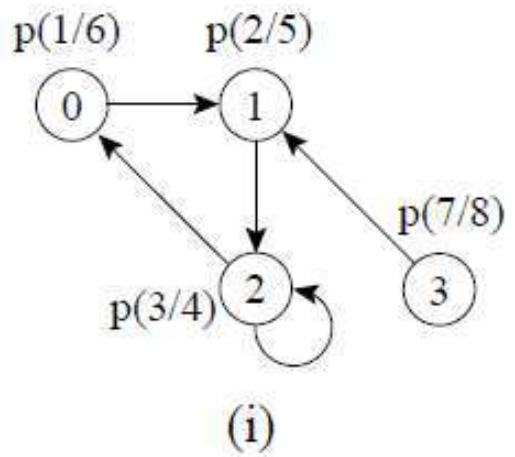


Vértice 0 é
tornado preto
 $t[0] = 6$

(g)

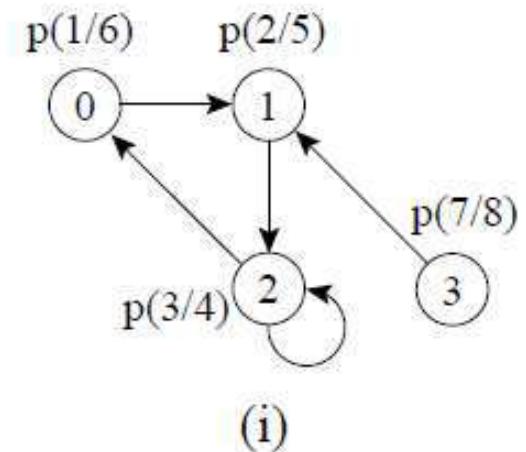
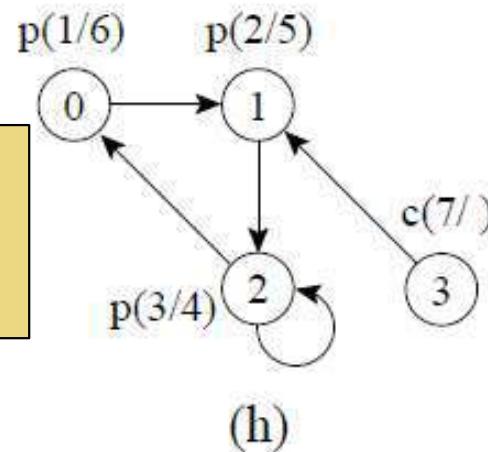
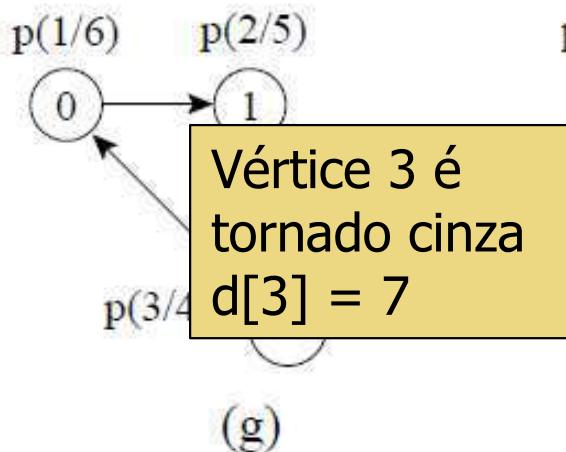
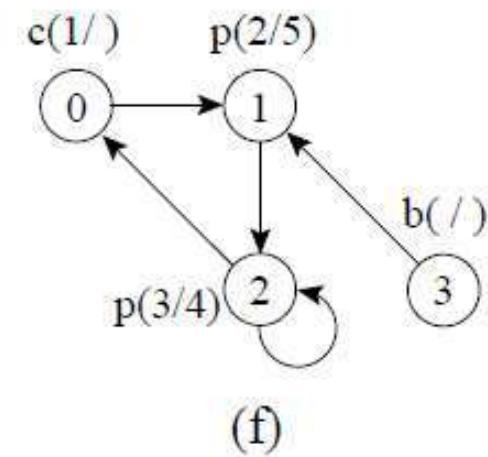
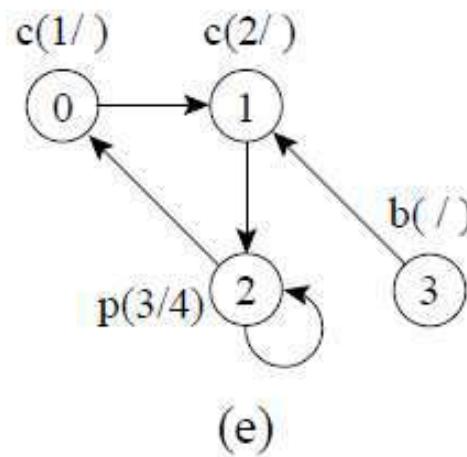
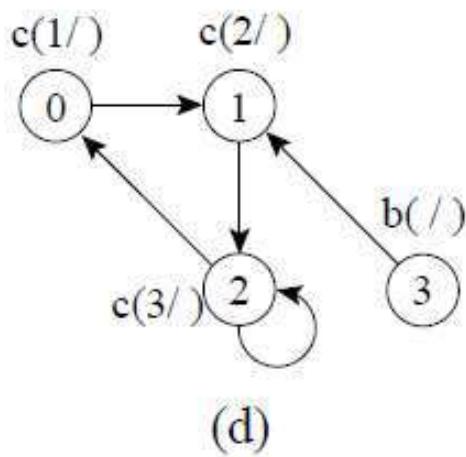
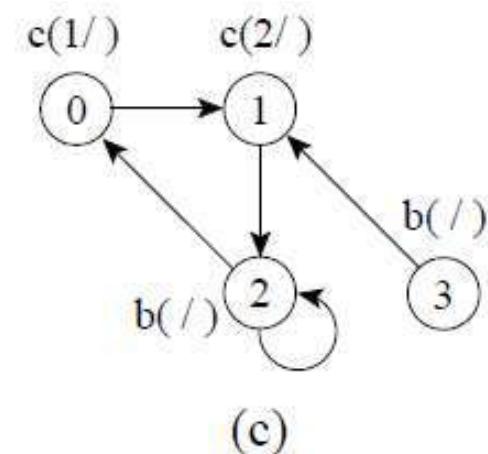
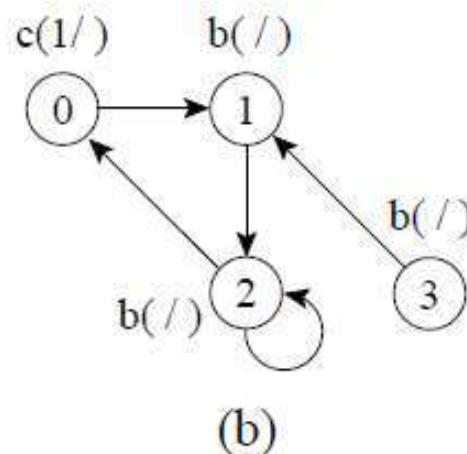
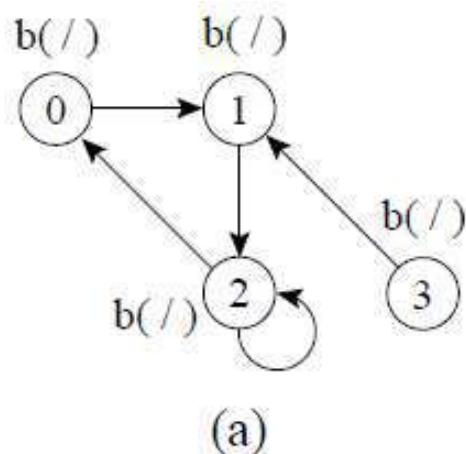


(h)

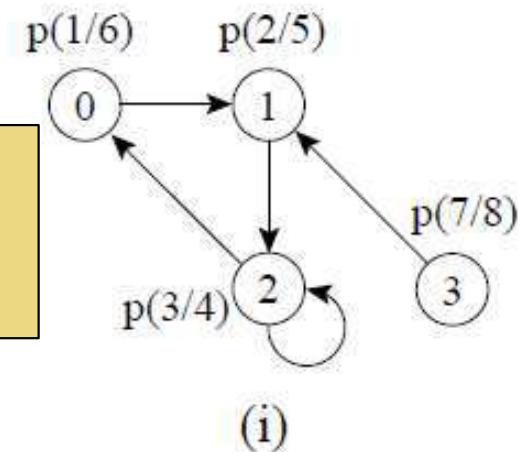
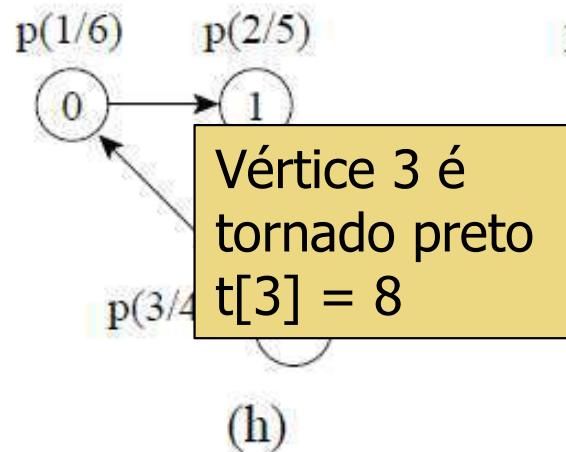
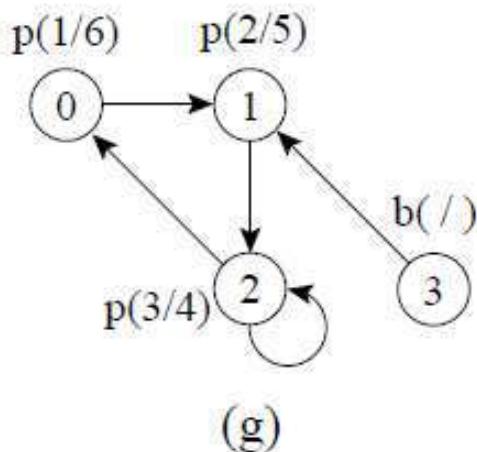
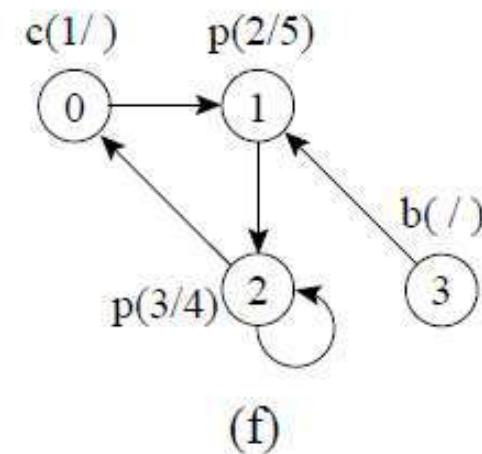
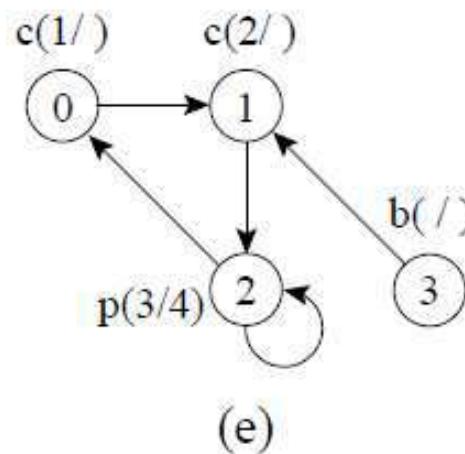
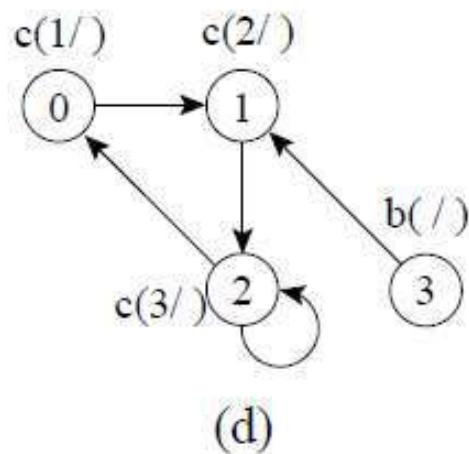
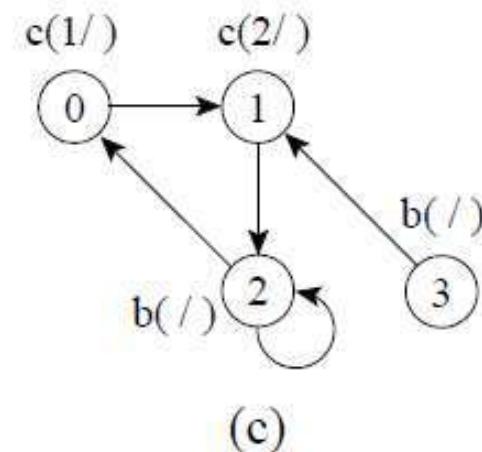
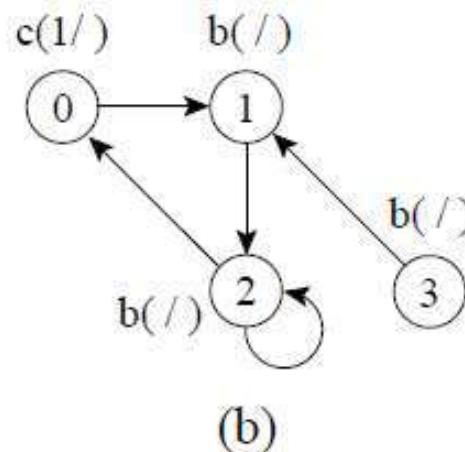
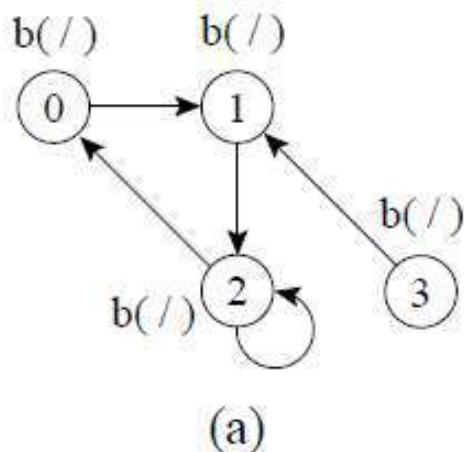


(i)

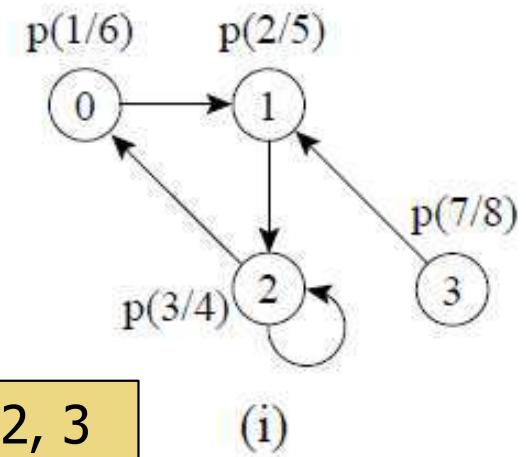
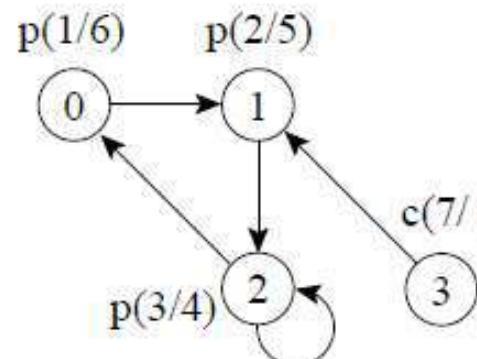
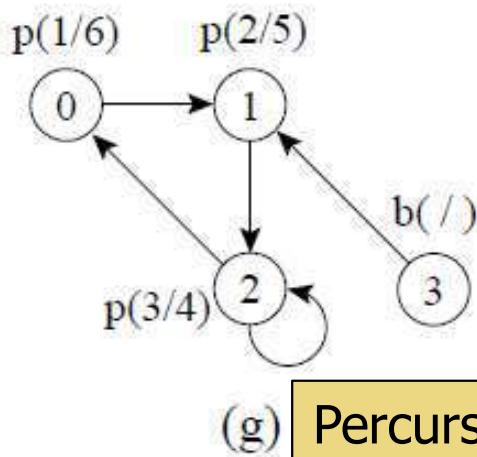
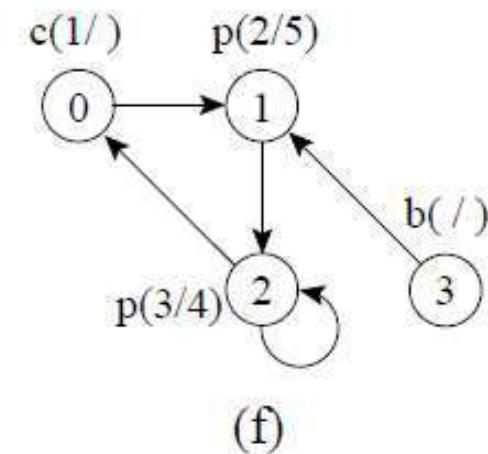
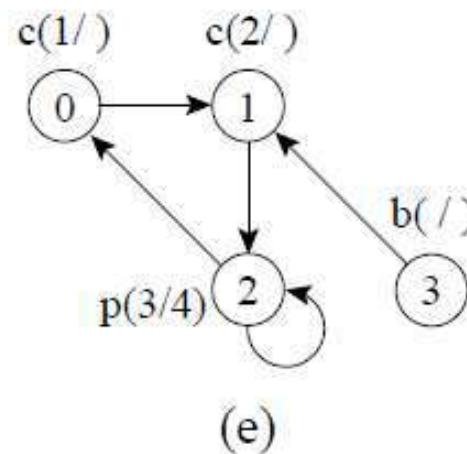
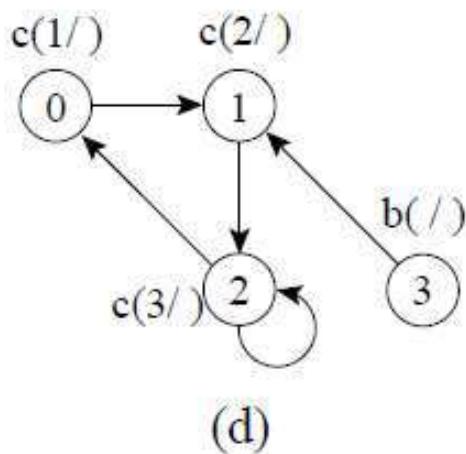
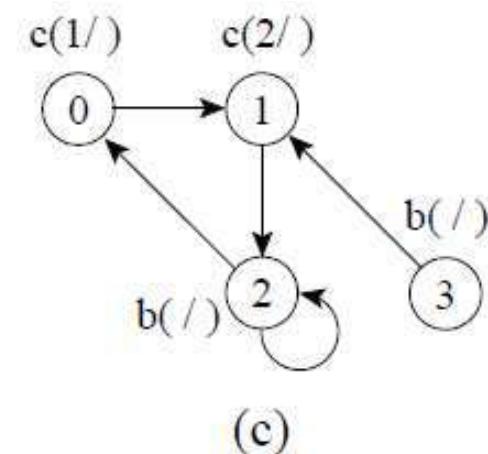
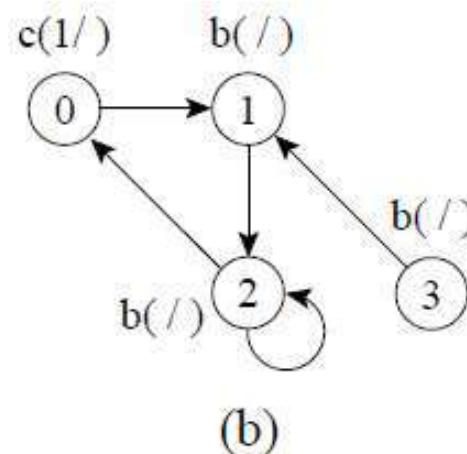
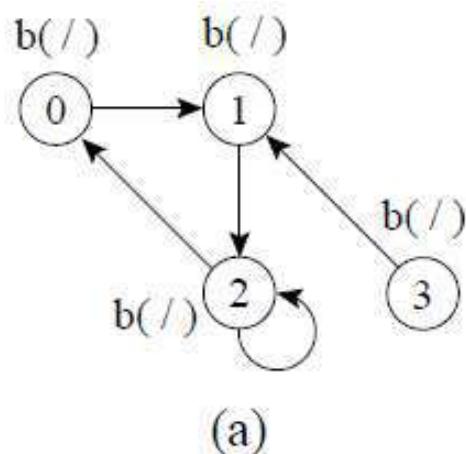
Exemplo



Exemplo



Exemplo



Percorso em profundidade: 0, 1, 2, 3

Busca em profundidade

Análise

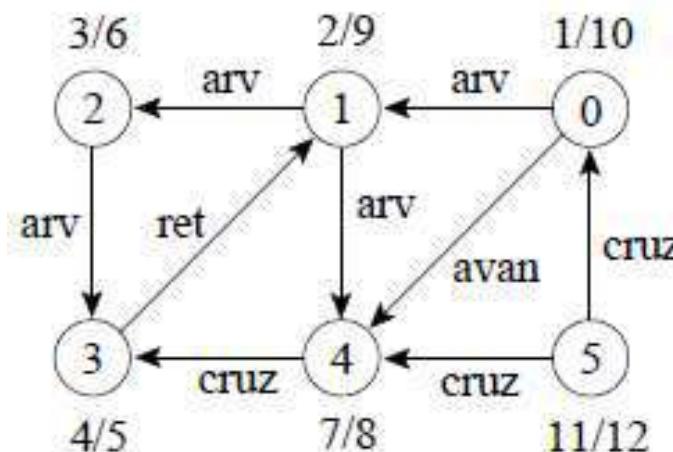
- ◆ Os dois anéis da *BuscaEmProfundidade* têm custo $O(|V|)$ cada um, a menos da chamada do procedimento *VisitaDfs*(u) no segundo anel
- ◆ O procedimento *VisitaDfs* é chamado exatamente uma vez para cada vértice $u \in V$, desde que *VisitaDfs* é chamado apenas para vértices brancos e a primeira ação é pintar o vértice de cinza
- ◆ Durante a execução de *VisitaDfs*(u) o anel principal é executado $|Adj[u]|$ vezes
- ◆ Desde que $\sum_{u \in V} |Adj[u]| = O(|A|)$, o tempo total de execução de *VisitaDfs* é $O(|A|)$
- ◆ Logo, a complexidade total da *BuscaEmProfundidade* é $O(|V| + |A|)$

Classificação de arestas

- ◆ Classificação de arestas pode ser útil para derivar outros algoritmos
 - **Arestas de árvore:** são arestas de uma árvore de busca em profundidade. A aresta (u, v) é uma aresta de árvore se v foi descoberto pela primeira vez ao percorrer a aresta (u, v)
 - **Arestas de retorno:** conectam um vértice u com um antecessor v em uma árvore de busca em profundidade (inclui *self-loops*)
 - **Arestas de avanço:** não pertencem à árvore de busca em profundidade mas conectam um vértice a um descendente que pertence à árvore de busca em profundidade
 - **Arestas de cruzamento:** podem conectar vértices na mesma árvore de busca em profundidade, ou em duas árvores diferentes

Classificação de arestas

- ♦ Na busca em profundidade cada aresta pode ser classificada pela cor do vértice que é alcançado pela primeira vez:
 - Branco indica uma aresta de árvore
 - Cinza indica uma aresta de retorno
 - Preto indica uma aresta de avanço quando u é descoberto antes de v ou uma aresta de cruzamento caso contrário



Teste para verificar se Grafo é acíclico

- ◆ A busca em profundidade pode ser usada para verificar se um grafo é acíclico ou contém um ou mais ciclos
- ◆ Se uma aresta de retorno é encontrada durante a busca em profundidade em G , então o grafo tem ciclo
- ◆ Um grafo direcionado G é acíclico se e somente se a busca em profundidade em G não apresentar arestas de retorno
- ◆ O algoritmo *BuscaEmProfundidade* pode ser alterado para descobrir arestas de retorno. Para isso, basta verificar se um vértice v adjacente a um vértice u apresenta a cor cinza na primeira vez que a aresta (u, v) é percorrida
- ◆ O algoritmo tem custo linear no número de vértices e de arestas de um grafo $G = (V, A)$ que pode ser utilizado para verificar se G é acíclico

Busca em Largura

- ◆ A busca em largura (do inglês *breadth-first search*) expande a fronteira entre vértices descobertos e não descobertos uniformemente através da largura da fronteira
- ◆ O algoritmo descobre todos os vértices a uma distância k do vértice origem antes de descobrir qualquer vértice a uma distância $k + 1$
- ◆ O grafo $G(V, A)$ pode ser direcionado ou não direcionado

Busca em Largura

- ◆ Cada vértice é colorido de branco, cinza ou preto
- ◆ Todos os vértices são inicializados branco
- ◆ Quando um vértice é descoberto pela primeira vez ele torna-se cinza
- ◆ Vértices cinza e preto já foram descobertos, mas são distinguidos para assegurar que a busca ocorra em largura
- ◆ Se $(u, v) \in A$ e o vértice u é preto, então o vértice v tem que ser cinza ou preto
- ◆ Vértices cinza podem ter alguns vértices adjacentes brancos, e eles representam a fronteira entre vértices descobertos e não descobertos

Busca em Largura

Implementação

```
void BuscaEmLargura(TipoGrafo *Grafo)
{ TipoValorVertice x;
  int Dist[MAXNUMVERTICES + 1];
  TipoCor Cor[MAXNUMVERTICES + 1];
  int Antecessor[MAXNUMVERTICES + 1];
  for (x = 0; x <= Grafo->NumVertices - 1; x++) {
    Cor[x] = branco; Dist[x] = INFINITO;
    Antecessor[x] = -1;
  }
  for (x = 0; x <= Grafo->NumVertices - 1; x++) {
    if (Cor[x] == branco)
      VisitaBfs(x, Grafo, Dist, Cor, Antecessor);
  }
}
```

Busca em Largura

Implementação

```
void BuscaEmLargura(TipoGrafo *Grafo)
{ TipoValorVertice x;
  int Dist[MAXNUMVERTICES + 1];
  TipoValorVertice Colore os vértices de branco[MVERTICES + 1];
  int Cor[XNUMVERTICES + 1];
  for (x = 0; x <= Grafo->NumVertices - 1; x++) {
    Cor[x] = branco; Dist[x] = INFINITO;
    Antecessor[x] = -1;
  }
  for (x = 0; x <= Grafo->NumVertices - 1; x++) {
    if (Cor[x] == branco)
      VisitaBfs(x, Grafo, Dist, Cor, Antecessor);
  }
}
```

Colore os vértices de branco

Cor[x] = branco;

Busca em Largura

Implementação

```
void BuscaEmLargura(TipoGrafo *Grafo)
{ TipoValorVertice x;
  int Dist[MAXNUMVERTICES + 1];
  TipoCor Cor[MAXNUMVE]
  int Antecessor[MAXNU
    for (x = 0; x <= Grafo->NumVertices - 1; x++) {
      Cor[x] = branco; Dist[x] = INFINITO;
      Antecessor[x] = -1;
    }
    for (x = 0; x <= Grafo->NumVertices - 1; x++) {
      if (Cor[x] == branco)
        VisitaBfs(x, Grafo, Dist, Cor, Antecessor);
    }
}
```

Inicializa $d[u]$ do vértice de origem até o vértice u para o valor infinito

Busca em Largura

Implementação

```
void BuscaEmLargura(TipoGrafo *Grafo)
{ TipoValorVertice x;
  int Dist[MAXNUMVERTICES + 1];
  TipoCor Cor[MAXNUMVERTICES + 1];
  int Inicializa os antecessores para -1
  for (x = 0; x <= Grafo->NumVertices - 1; x++) {
    Cor[x] = branco; Dist[x] = INFINITO;
    Antecessor[x] = -1;
  }
  for (x = 0; x <= Grafo->NumVertices - 1; x++) {
    if (Cor[x] == branco)
      VisitaBfs(x, Grafo, Dist, Cor, Antecessor);
  }
}
```

Busca em Largura

Implementação

```
void BuscaEmLargura(TipoGrafo *Grafo)
{ TipoValorVertice x;
  int Dist[MAXNUMVERTICES + 1];
  TipoCor Cor[MAXNUMVERTICES + 1];
  int Antecessor[MAXNUMVERTICES + 1];
  for (x = 0; x <= Grafo->NumVertices - 1; x++) {
    Cor[x] = branco; Dist[x] = INFINITO;
    verifica cada vértice e quando
    encontra um vértice branco
    visita esse vértice
    }
  for (x = 0; x <= Grafo->NumVertices - 1; x++) {
    if (Cor[x] == branco)
      VisitaBfs(x, Grafo, Dist, Cor, Antecessor);
  }
}
```

verifica cada vértice e quando
encontra um vértice branco
visita esse vértice

Busca em Largura

Implementação

```
void BuscaEmLargura(TipoGrafo *Grafo)
{ TipoValorVertice x;
  int Dist[MAXNUMVERTICES + 1];
  T
  i
  f
  veri
    encontra um vértice branco
    visita esse vértice
  }
  = -1;
}

for (x = 0; x <= Grafo->NumVertices - 1; x++) {
  if (Cor[x] == branco)
    VisitaBfs(x, Grafo, Dist, Cor, Antecessor);
}
}
```

Toda vez que $VisitaBfs(u)$ é chamado, o vértice u se torna a raiz de uma nova **árvore de busca em largura**, e o conjunto de árvores forma uma **floresta** de árvores de busca

Busca em Largura

Implementação

```
/** Entram aqui os operadores FFVazia, Vazia, Enfileira e Desenfileira do */
/** do Programa 3.18 ou do Programa 3.20, dependendo da implementação */
/** da busca em largura usar arranjos ou apontadores, respectivamente */
void VisitaBfs(TipoValorVertice u, TipoGrafo *Grafo,
                int *Dist, TipoCor *Cor, int *Antecessor)
{ TipoValorVertice v; TipoApontador Aux; short FimListaAdj;
  TipoPeso Peso; TipoItem Item; TipoFila Fila;
  Cor[u] = cinza; Dist[u] = 0;
  FFVazia(&Fila);
  Item.Vertice = u; Item.Peso = 0;
  Enfileira(Item, &Fila);
  printf("Visita origem%2d cor: cinza F:", u);
  ImprimeFila(Fila); getchar();
```

Busca em Largura

Implementação

```
/** Entram aqui os operadores FFVazia, Vazia, Enfileira e Desenfileira do */
/** do Programa 3.18 cu do Programa 3.20 dependendo da implementação */
/** da busca em largura */ u é tornado cinza pontadores, respectivamente */
void VisitaBfs(TipoValorVertice u, TipoGrafo *Grafo,
                int *Dist, TipoCor *Cor, int *Antecessor)
{ TipoValorVertice v; TipoApontador Aux; short FimListaAdj;
  TipoPeso Peso; TipoItem Item; TipoFila Fila;
  Cor[u] = cinza; Dist[u] = 0;
  FFVazia(&Fila);
  Item.Vertice = u; Item.Peso = 0;
  Enfileira(Item, &Fila);
  printf("Visita origem%2d cor: cinza F:", u);
  ImprimeFila(Fila); getchar();
```

Busca em Largura

Implementação

```
/* Entram aqui os operadores FFVazia, Vazia, Enfileira e Desenfileira do */
/* do Programa 3.18 ou do Programa 3.20 dependendo da implementação */
/* da busca em largura */ u é tornado cinza pontadores, respectivamente */

void VisitaBfs(TipoValorVertice u, TipoGrafo *Grafo,
{ TipoValorVertice Item;
  TipoFila Fila;
  Cor cor[MaxVertices];
  cor[u] = Cinza; cor[VerticeOrigem] = Branca;
  FFVazia(&Fila);
  Item.Vertice = u; Item.Peso = 0;
  Enfileira(Item, &Fila);
  printf("Visita origem%2d cor: cinza F:", u);
  ImprimeFila(Fila); getchar();
```

O algoritmo usa uma fila do tipo "primeiro-que-chega, primeiro-atendido" para gerenciar o conjunto de vértices cinza

Busca em Largura

Implementação

```
/** Entram aqui os operadores FFVazia, Vazia, Enfileira e Desenfileira do */
/** do Programa 3.18 ou do Projeto 3.19. Consultar o código da implementação */
/** da busca em largura usar a estrutura de dados que for mais conveniente, respectivamente */
void VisitaBfs(TipoValorVertice u, tipoGrafo *Grafo,
                int *Dist, TipoCor *Cor, int *Antecessor)
{ TipoValorVertice v; TipoApontador Aux; short FimListaAdj;
  TipoPeso Peso; TipoItem Item; TipoFila Fila;
  Cor[u] = cinza; Dist[u] = 0;
  FFVazia(&Fila);
  Item.Vertice = u; Item.Peso = 0;
  Enfileira(Item, &Fila);
  printf("Visita origem%2d cor: cinza F:", u);
  ImprimeFila(Fila); getchar();
```

$d[u]$ é inicializado com zero

Busca em Largura

Implementação

```
/** Entram aqui os operadores FFVazia, Vazia, Enfileira e Desenfileira do */
/** do Programa 3.18 ou do Programa 3.20, dependendo da implementação */
/** da busca em largura usar arranjos ou apontadores */
void VisitaBfs(TipoValorVertice u,
                int *Dist, TipoCor *Cor, int *Antecessor)
{ TipoValorVertice v; TipoApontador Aux; short FimListaAdj;
  TipoPeso Peso; TipoItem Item; TipoFila Fila;
  Cor[u] = cinza; Dist[u] = 0;
  FFVazia(&Fila);
  Item.Vertice = u; Item.Peso = 0;
  Enfileira(Item, &Fila);
  printf("Visita origem%2d cor: cinza F:", u);
  ImprimeFila(Fila); getchar();
```

Fila é inicializada com o vértice de origem

Busca em Largura

Implementação

```
while (!FilaVazia(Fila)) { ————— Enquanto houver vértices cinzas
    Desenfileira(&Fila, &Item);
    u = Item.Vertice;
    if (!ListaAdjVazia(&u, Grafo)) {
        Aux = PrimeiroListaAdj(&u, Grafo);
        FimListaAdj = FALSE;
        while (FimListaAdj == FALSE) {
            ProxAdj(&u, &v, &Peso, &Aux, &FimListaAdj);
            if (Cor[v] != branco) continue;
            Cor[v] = cinza; Dist[v] = Dist[u] + 1;
            Antecessor[v] = u;
            Item.Vertice = v; Item.Peso = Peso;
            Enfileira(Item, &Fila);
        }
        Cor[u] = preto;
        printf("Visita%2d Dist%2d cor: preto F:", u, Dist[u]);
        ImprimeFila(Fila); getchar(); }
    }
```

Busca em Largura

Implementação

```
while (!FilaVazia(Fila)) {  
    Desenfileira(&Fila, &Item); → O vértice do início da fila é  
    u = Item.Vertice;           desenfileirado  
    if (!ListaAdjVazia(&u, Grafo)) {  
        Aux = PrimeiroListaAdj(&u, Grafo);  
        FimListaAdj = FALSE;  
        while (FimListaAdj == FALSE) {  
            ProxAdj(&u, &v, &Peso, &Aux, &FimListaAdj);  
            if (Cor[v] != branco) continue;  
            Cor[v] = cinza; Dist[v] = Dist[u] + 1;  
            Antecessor[v] = u;  
            Item.Vertice = v; Item.Peso = Peso;  
            Enfileira(Item, &Fila);  
        } }  
    Cor[u] = preto;  
    printf("Visita%2d Dist%2d cor: preto F:", u, Dist[u]);  
    ImprimeFila(Fila); getchar(); } }
```

Busca em Largura

Implementação

```
while (!FilaVazia(Fila)) {  
    Desenfileira(&Fila, &Item);  
    u = Item.Vertice;  
    if (!ListaAdjVazia(&u, Grafo)) { → Examina a lista de vértices  
        Aux = PrimeiroListaAdj(&u, Grafo); adjacentes  
        FimListaAdj = FALSE;  
        while (FimListaAdj == FALSE) {  
            ProxAdj(&u, &v, &Peso, &Aux, &FimListaAdj);  
            if (Cor[v] != branco) continue;  
            Cor[v] = cinza; Dist[v] = Dist[u] + 1;  
            Antecessor[v] = u;  
            Item.Vertice = v; Item.Peso = Peso;  
            Enfileira(Item, &Fila);  
        } }  
    Cor[u] = preto;  
    printf("Visita%2d Dist%2d cor: preto F:", u, Dist[u]);  
    ImprimeFila(Fila); getchar(); } }
```

Busca em Largura

Implementação

```
while (!FilaVazia(Fila)) {  
    Desenfileira(&Fila, &Item);  
    u = Item.Vertice;  
    if (!ListaAdjVazia(&u, Grafo)) {  
        Aux = PrimeiroListaAdj(&u, Grafo);  
        FimListaAdj = FALSE;  
        while (FimListaAdj == FALSE) {  
            ProxAdj(&u, &v, &Peso, &Aux, &FimListaAdj);  
            if (Cor[v] != branco) continue; → Visita se o adjacente  
for branco  
            Cor[v] = cinza; Dist[v] = Dist[u] + 1;  
            Antecessor[v] = u;  
            Item.Vertice = v; Item.Peso = Peso;  
            Enfileira(Item, &Fila);  
        } }  
    Cor[u] = preto;  
    printf("Visita%2d Dist%2d cor: preto F:", u, Dist[u]);  
    ImprimeFila(Fila); getchar(); } }
```

Busca em Largura

Implementação

```
while (!FilaVazia(Fila)) {  
    Desenfileira(&Fila, &Item);  
    u = Item.Vertice;  
    if (!ListaAdjVazia(&u, Grafo)) {  
        Aux = PrimeiroListaAdj(&u, Grafo);  
        FimListaAdj = FALSE;  
        while (FimListaAdj == FALSE) {  
            ProxAdj(&u, &v, &Peso, &Aux, &FimListaAdj);  
            if (Cor[v] != branco) continue;  
            Cor[v] = cinza; Dist[v] = Dist[u] + 1;  
            Antecessor[v] = u;  
            Item.Vertice = v; Item.Peso = Peso;  
            Enfileira(Item, &Fila);  
        } }  
    Cor[u] = preto;  
    printf("Visita%2d Dist%2d cor: preto F:", u, Dist[u]);  
    ImprimeFila(Fila); getchar(); } }
```

é tornado cinza, a distância de v a u é registrada e u é atribuído a antecessor de v e o novo vértice cinza é enfileirado em $Fila$

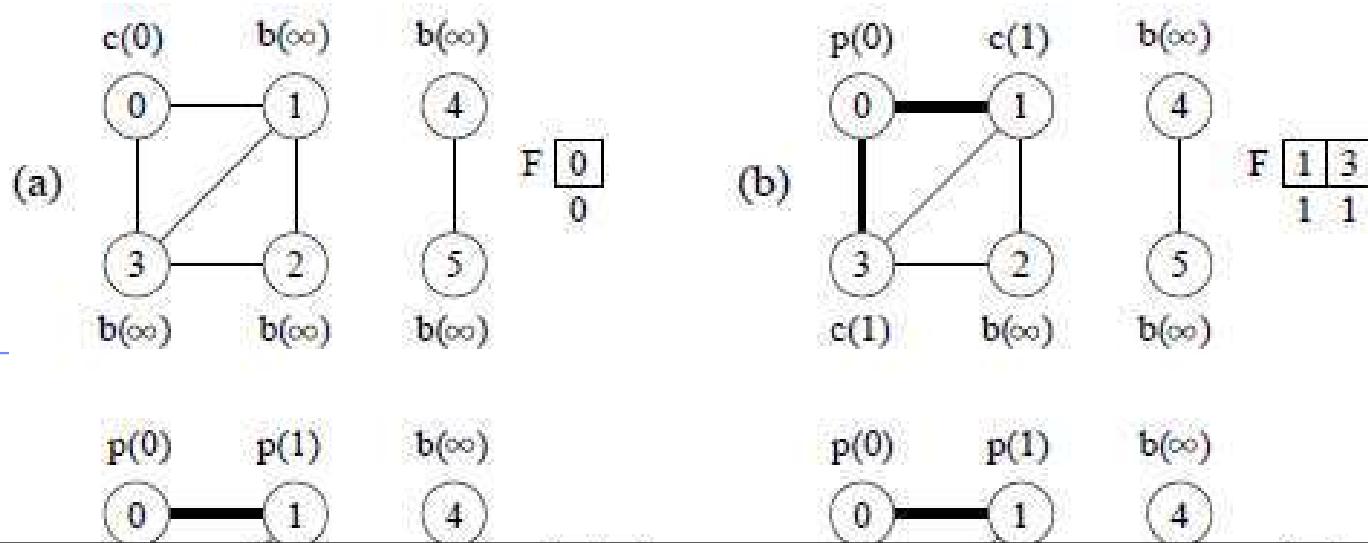
Busca em Largura

Implementação

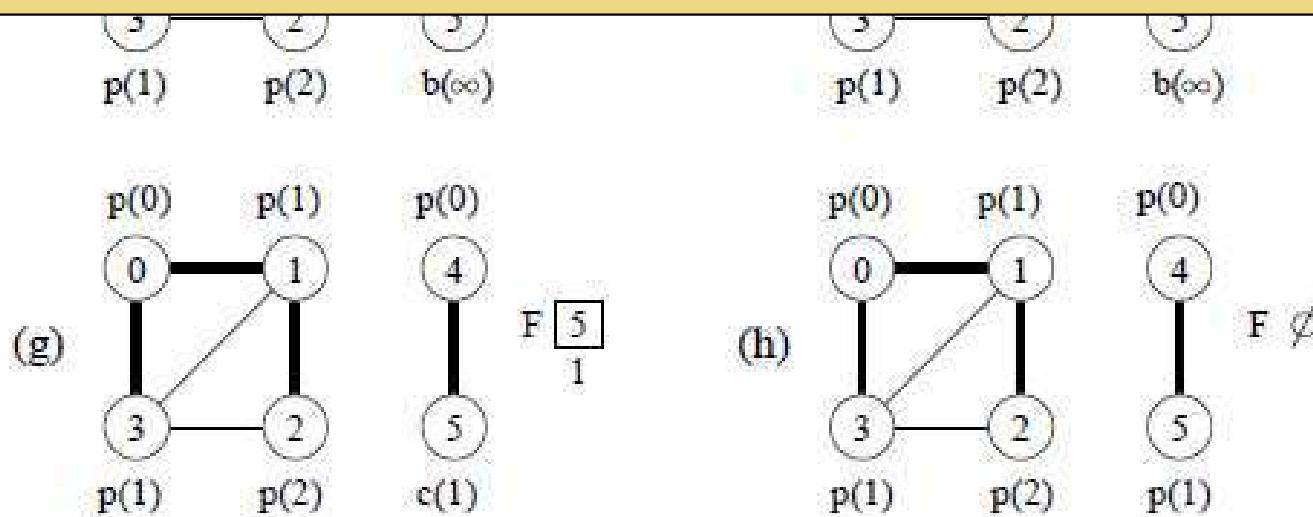
```
while (!FilaVazia(Fila)) {  
    Desenfileira(&Fila, &Item);  
    u = Item.Vertice;  
    if (!ListaAdjVazia(&u, Grafo)) {  
        Aux = PrimeiroListaAdj(&u, Grafo);  
        FimListaAdj = FALSE;  
        while (FimListaAdj == FALSE) {  
            ProxAdj(&u, &v, &Peso, &Aux, &FimListaAdj);  
            if (Cor[v] != branco) continue;  
            Cor[v] = cinza; Dist[v] = Dist[u] + 1;  
            Antecessor[v] = u;  
            Item.Vertice = v; Item.Peso = Peso;  
            Enfileira(Item, &Fila);  
        } }  
    Cor[u] = preto;   
    printf("Visita%2d Dist%2d cor: preto F:", u, Dist[u]);  
    ImprimeFila(Fila); getchar(); } }
```

Depois que toda a lista de adjacentes de u é percorrida, u é pintado de preto

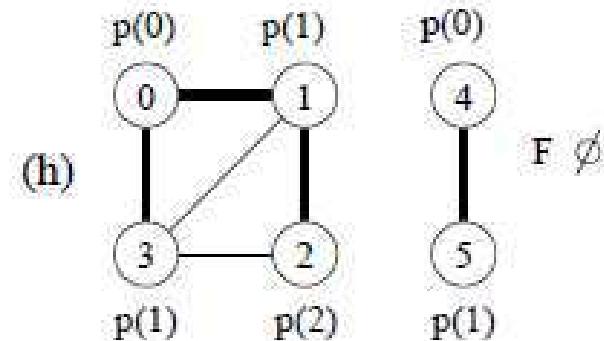
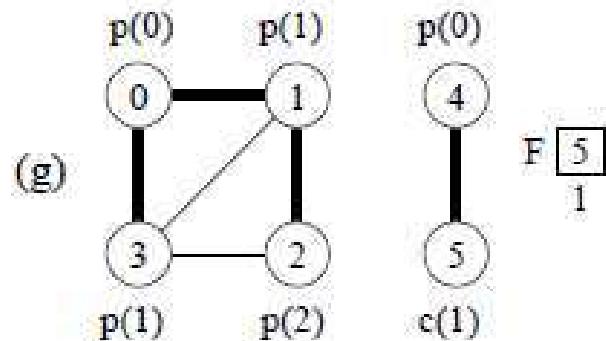
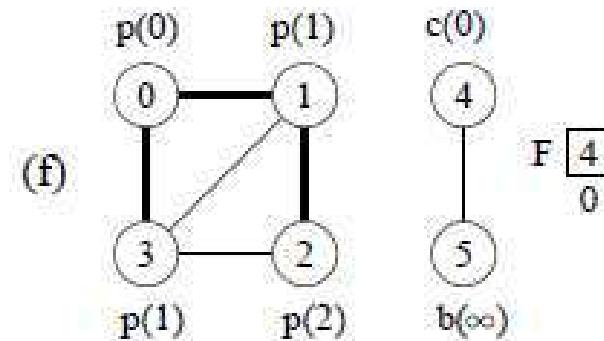
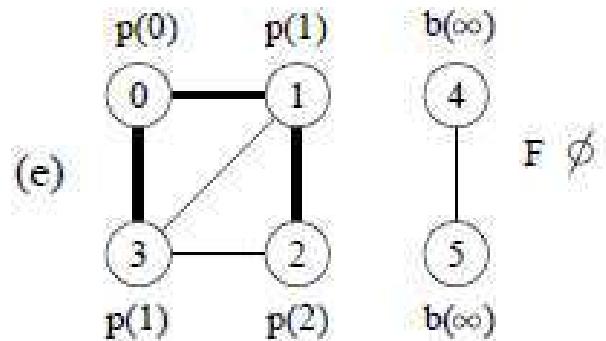
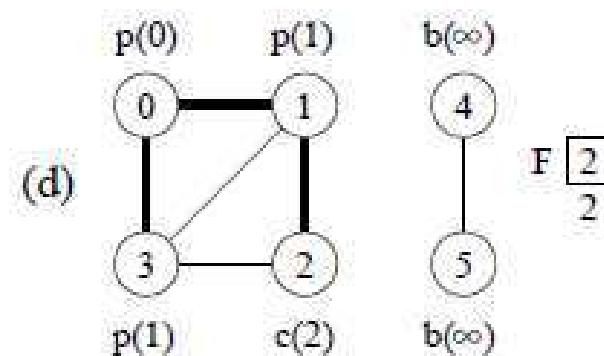
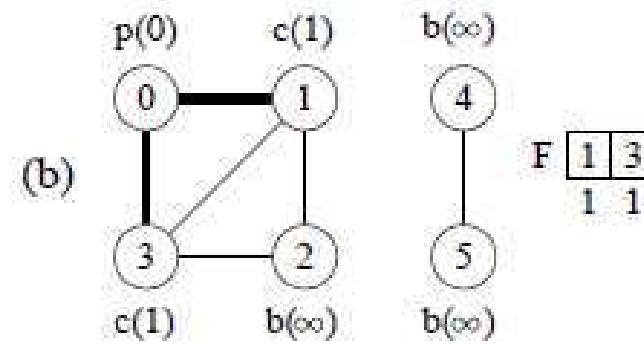
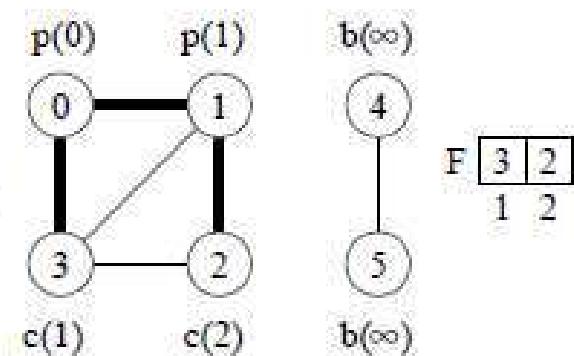
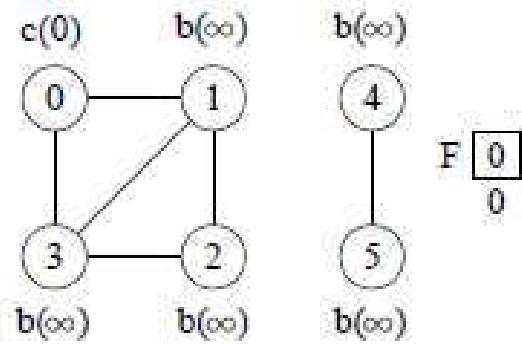
Exemplo



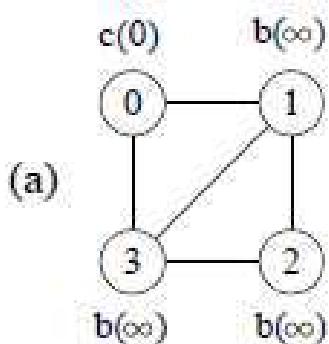
Ilustra o progresso da busca em largura. Arestras de árvore são mostradas em negrito. Ao lado de cada vértice é mostrada a cor branca, cinza ou preta (b, c ou p), e entre parênteses a distância $d[u]$. A fila F é mostrada ao final de cada iteração do anel while.



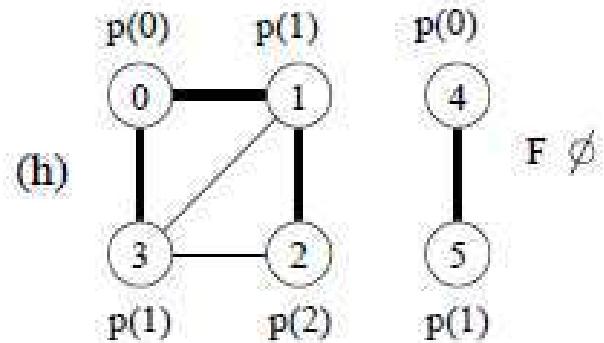
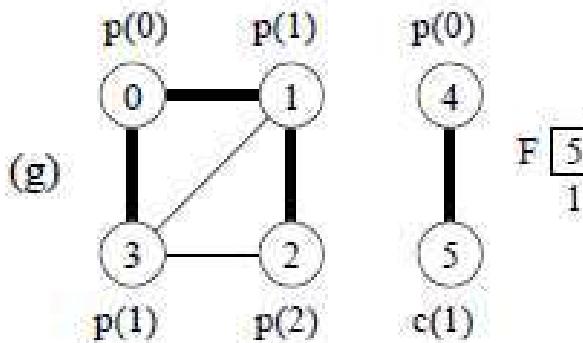
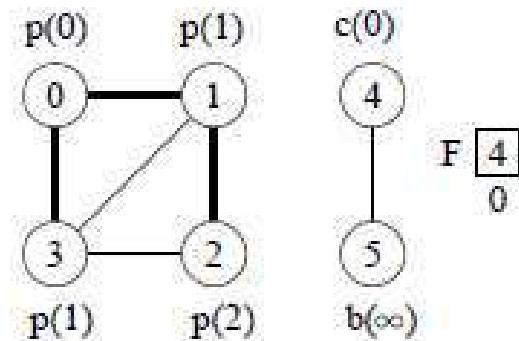
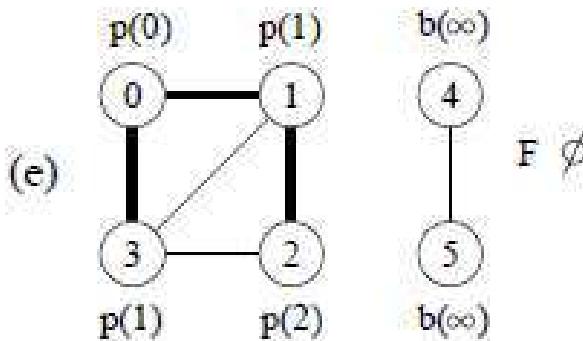
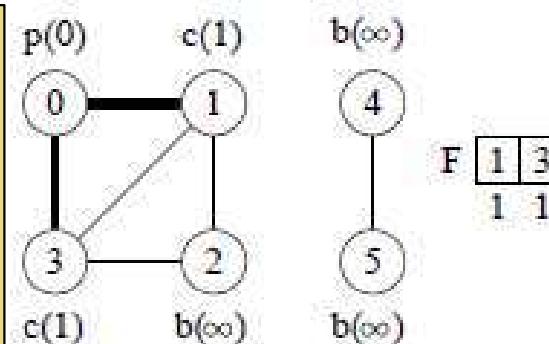
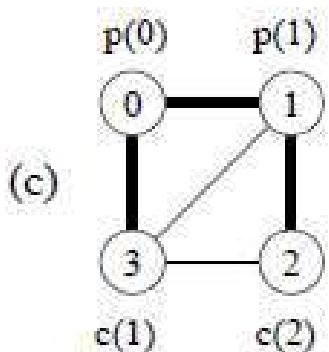
- 1) inicializa os vértices de branco
 2) Inicializa $d[u]$ do vértice de origem até o vértice u para o valor infinito
 3) Colore 0 de cinza e $d[0] = 0$
 4) Enfileira 0



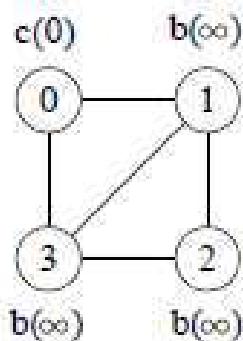
Exemplo



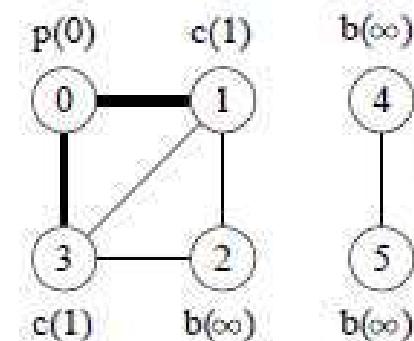
- 1) Desenfileira 0
- 2) Examina a lista de vértices adj. visita se branco
- 3) Colore de cinza vértices 1 e 3
- 4) $d[1] = 1$ e $d[3] = 1$
- 5) Enfileira 1 e 3
- 6) Colore 0 de preto



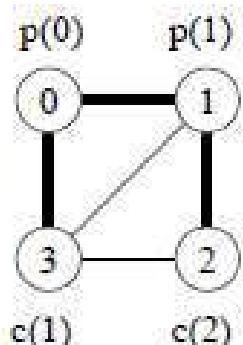
Exemplo



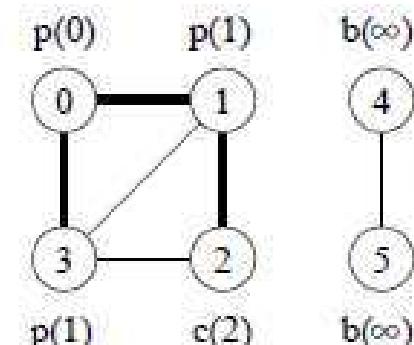
$F \boxed{0}$
0



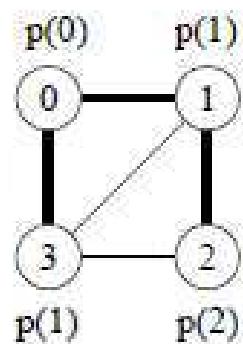
$F \boxed{1 \ 3}$
1 1



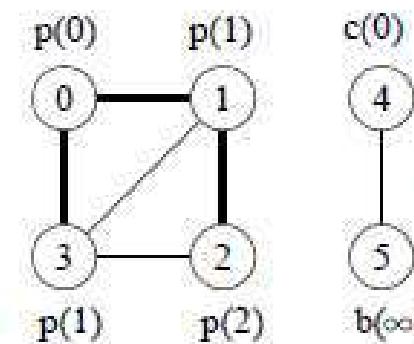
$F \boxed{3 \ 2}$
1 2



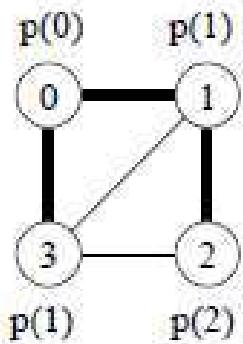
$F \boxed{2}$
2



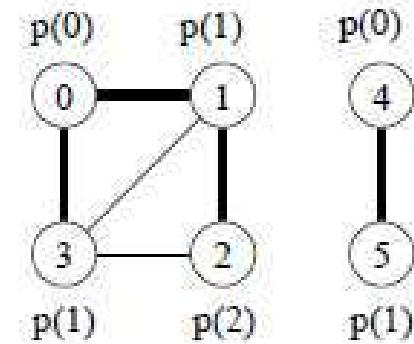
$F \emptyset$



$F \boxed{4}$
0

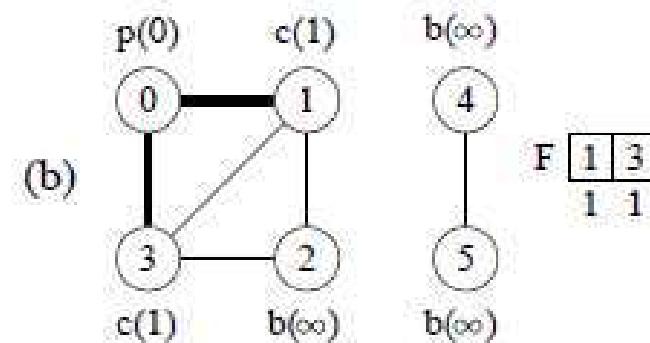
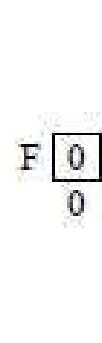
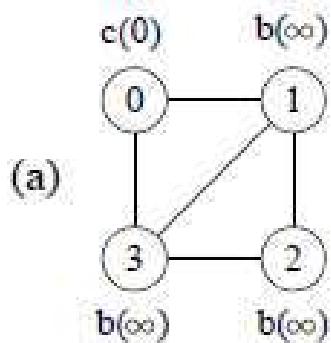


$F \boxed{5}$
1

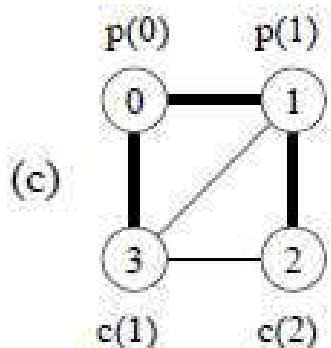


$F \emptyset$

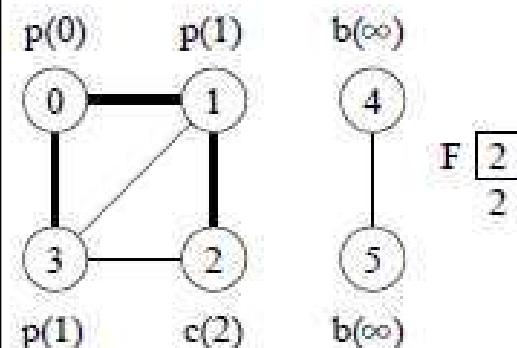
Exemplo



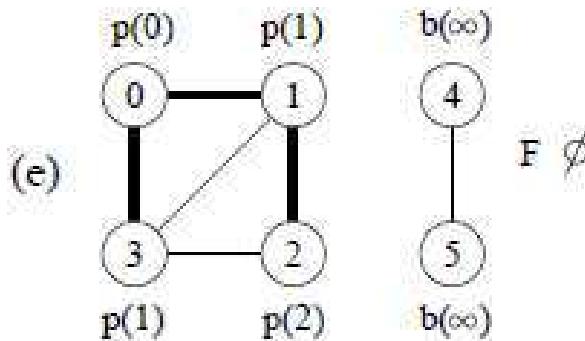
F [1 3]
1 1



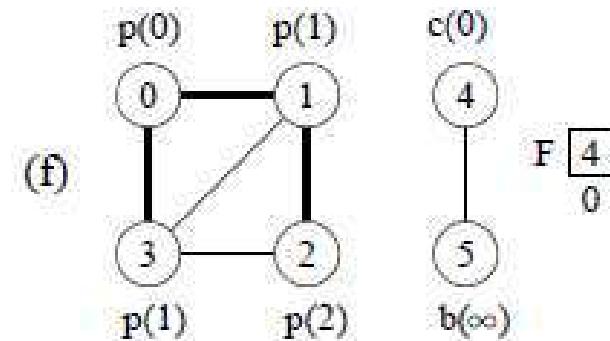
- 1) Desenfileira 3
2) Examina a lista de vértices adj. visita se branco
3) Colore 3 de preto



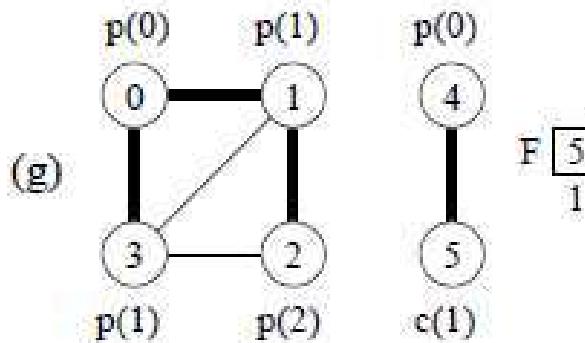
F [2]
2



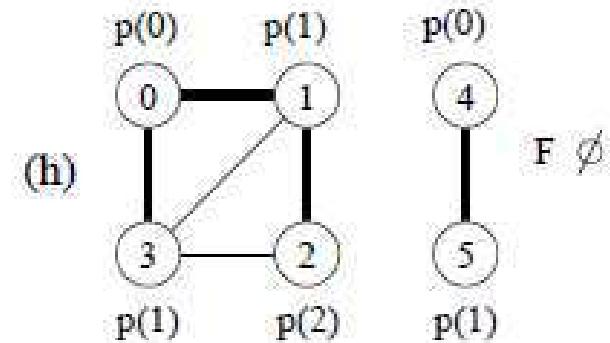
F \emptyset



F [4]
0

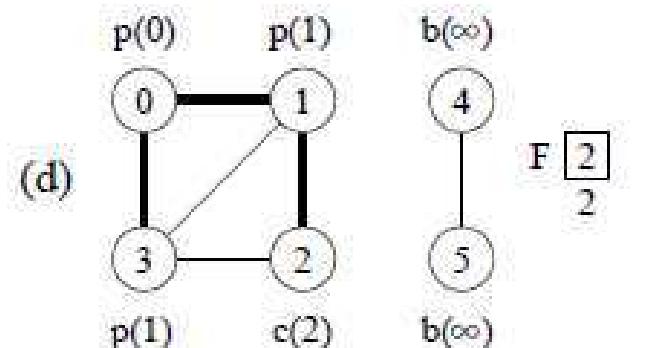
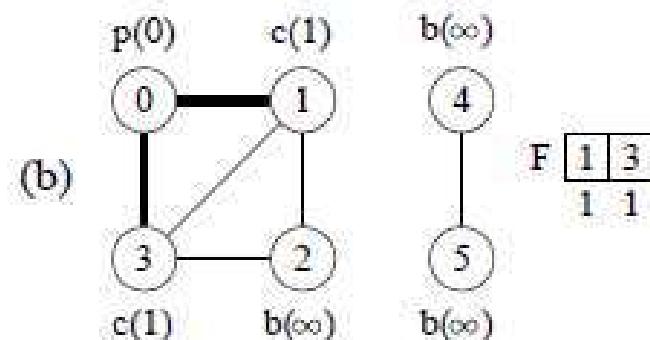
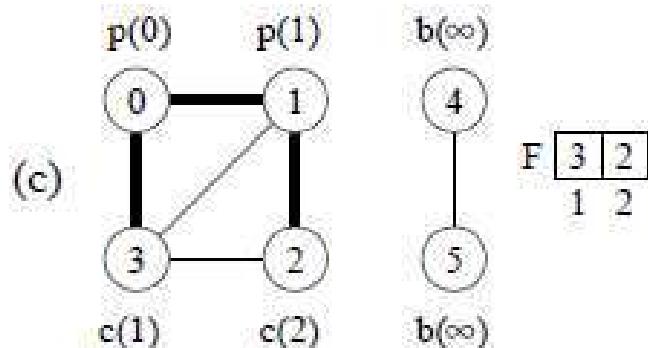
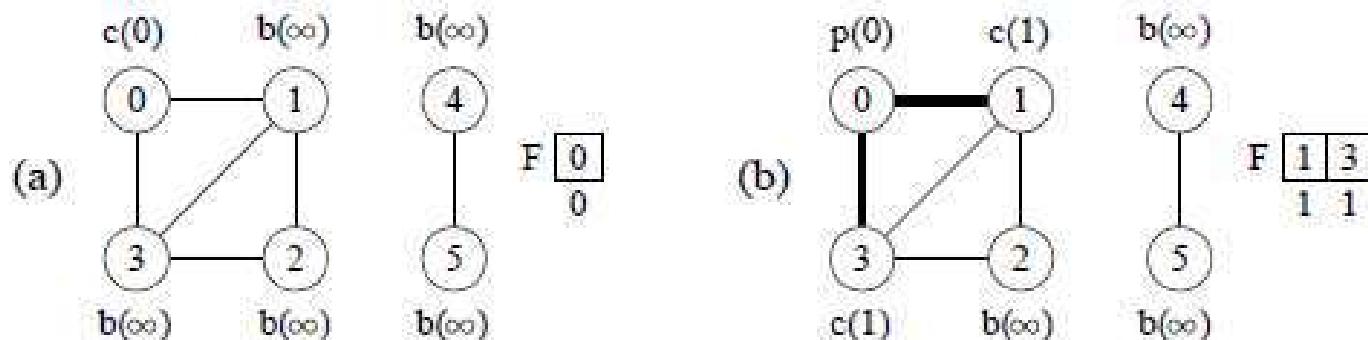


F [5]
1

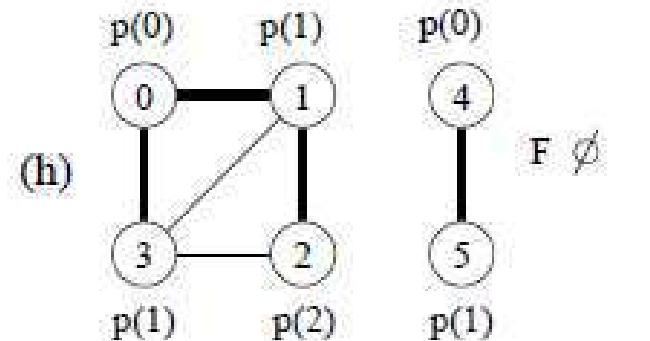
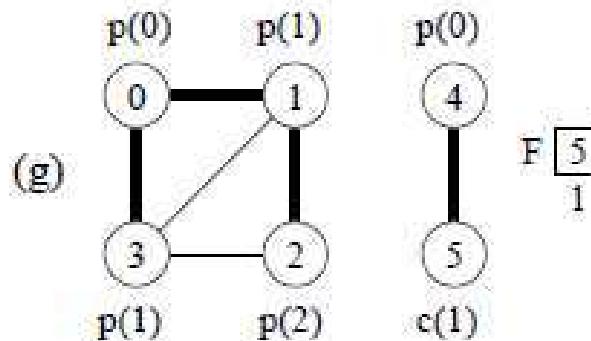
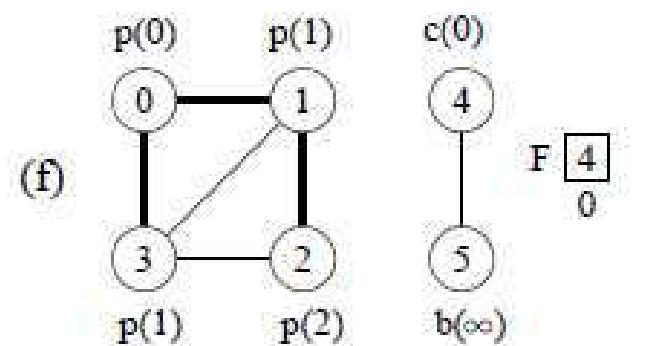
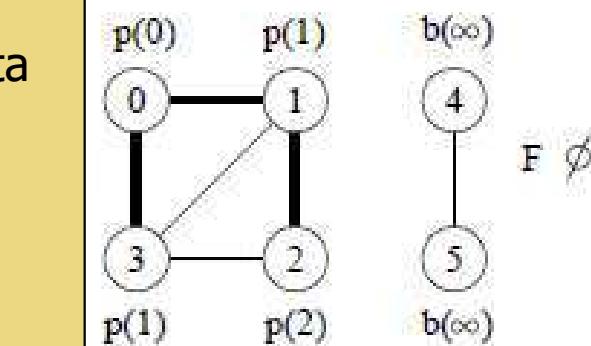


F \emptyset

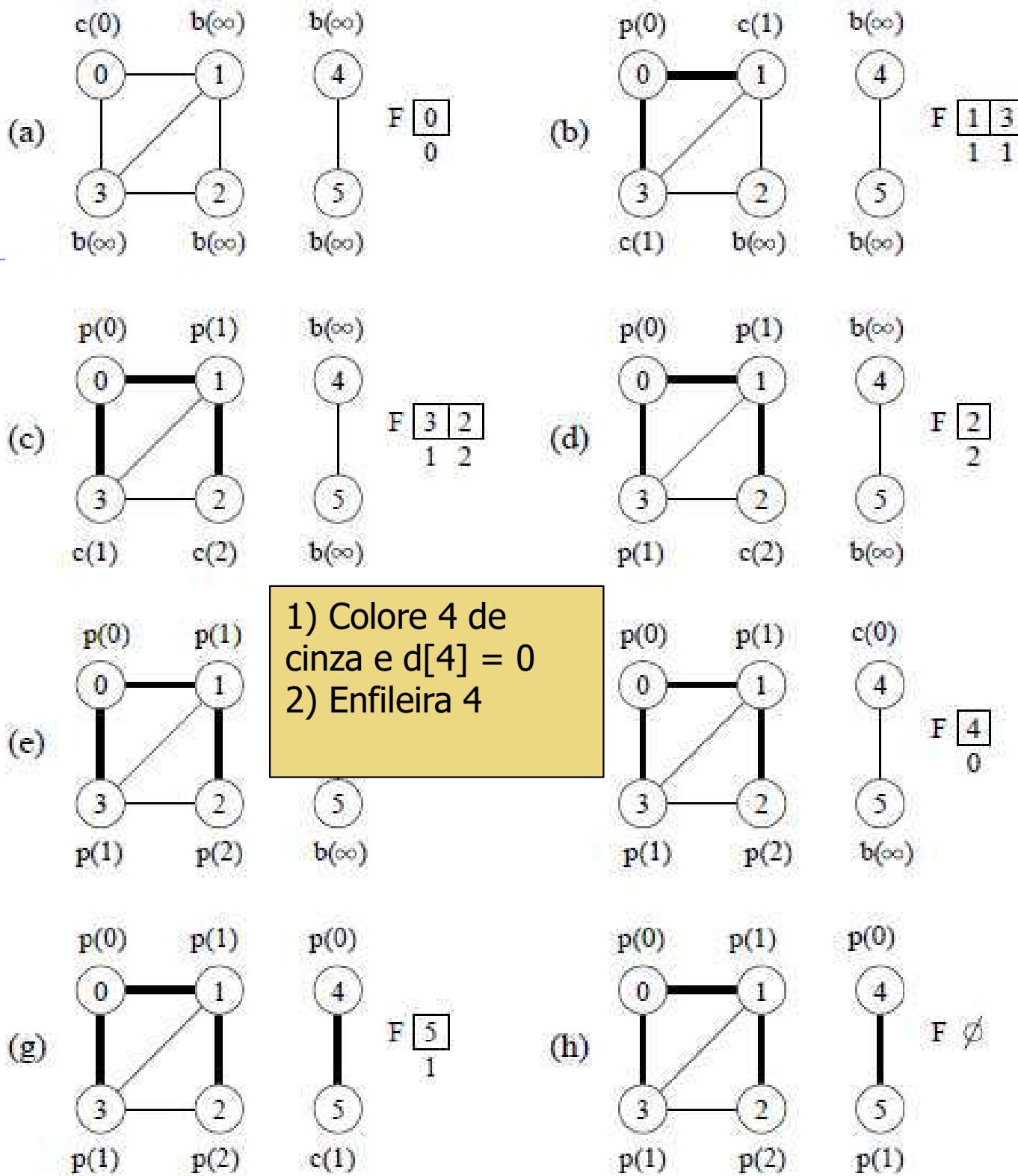
Exemplo



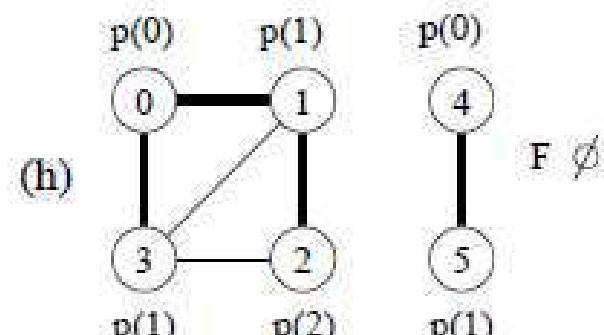
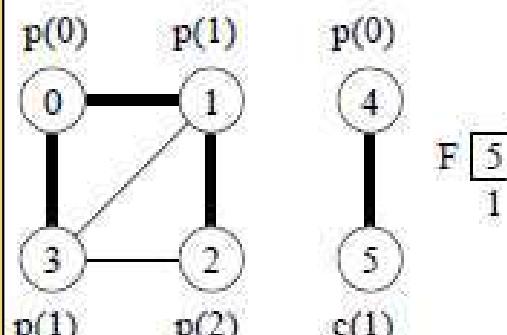
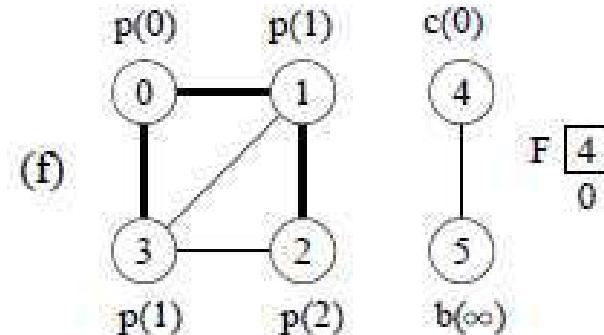
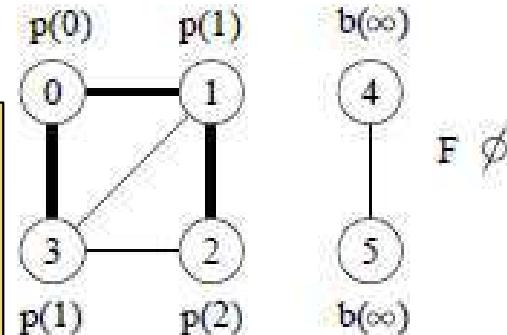
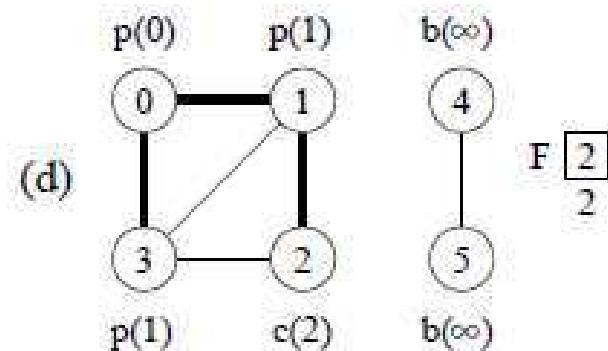
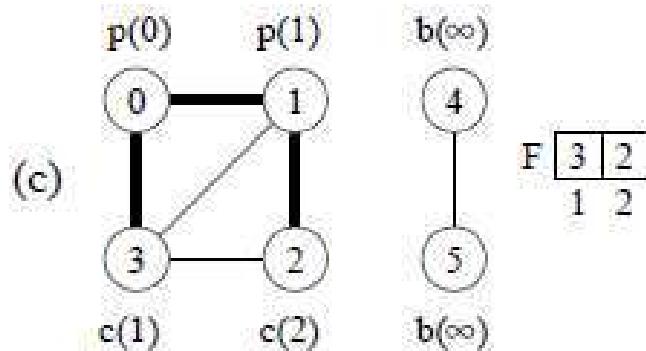
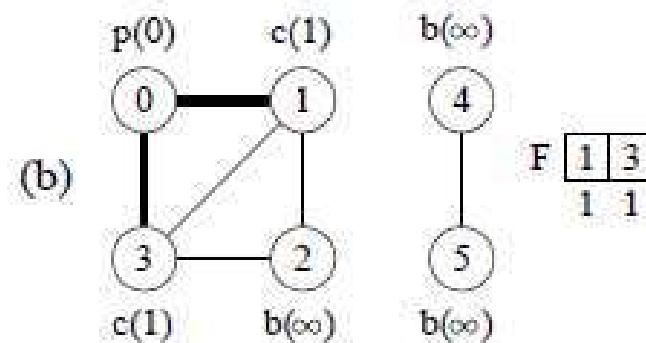
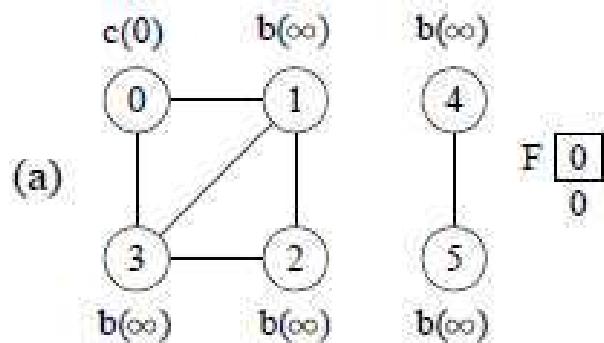
- 1) Desenfileira 2
- 2) Examina a lista de vértices adj. visita se branco
- 3) Colore 2 de preto



Exemplo

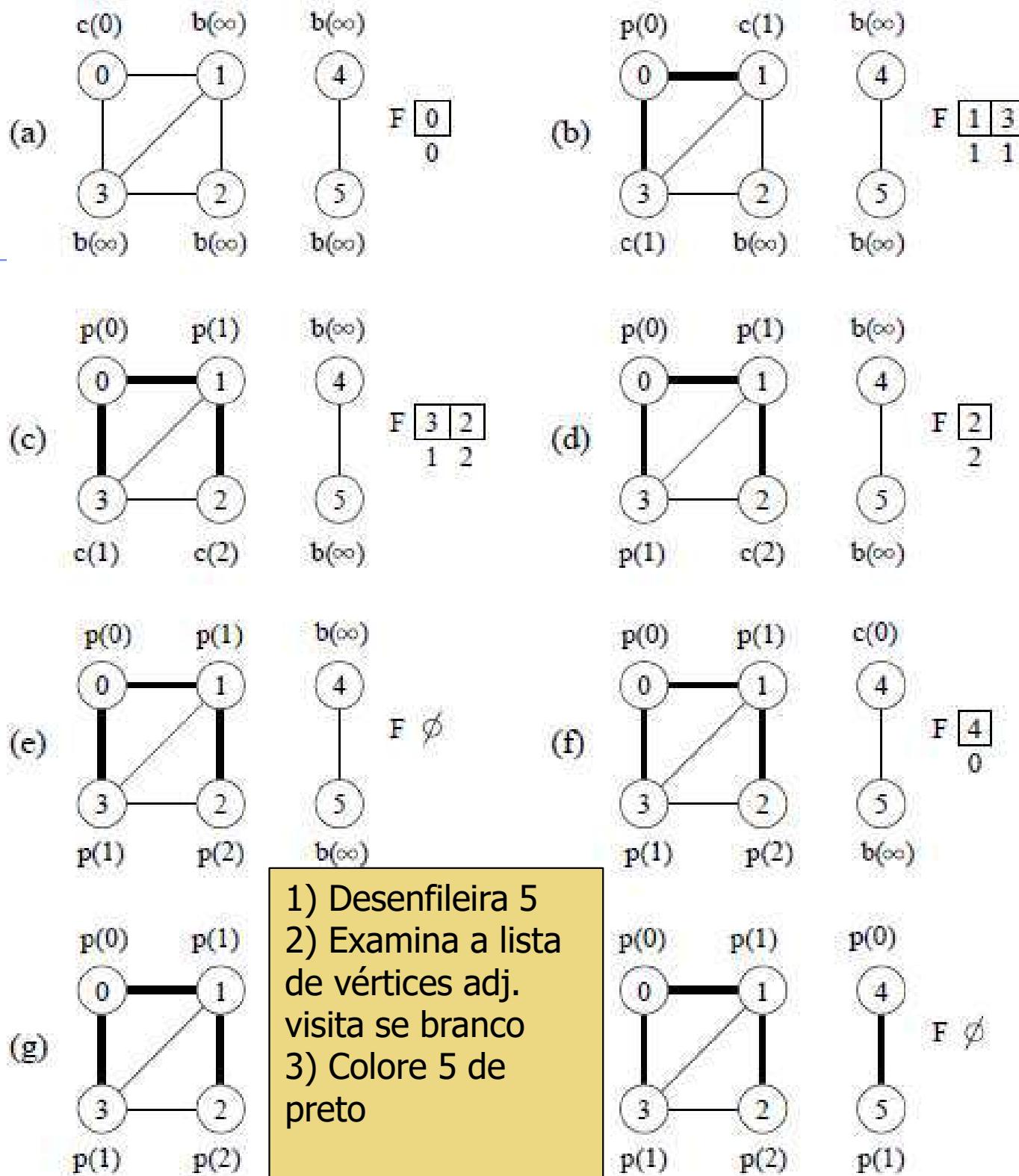


Exemplo

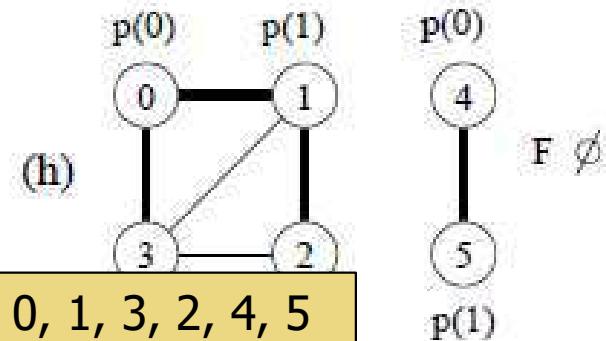
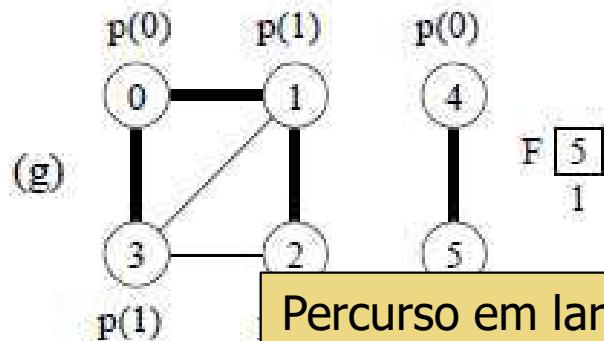
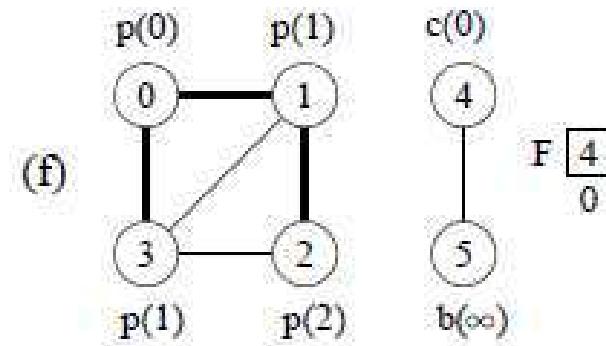
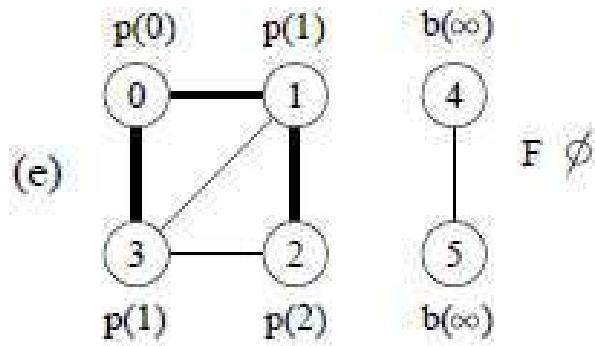
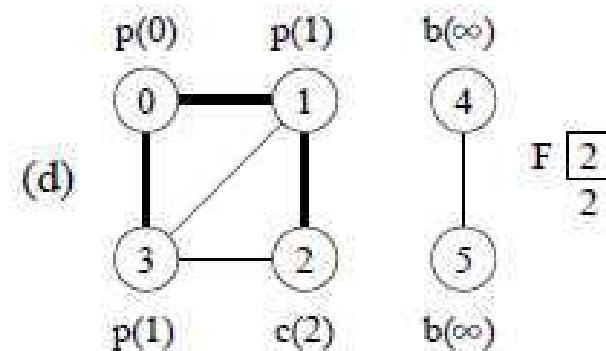
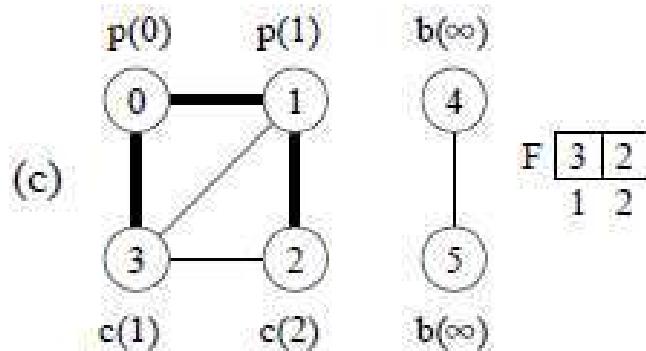
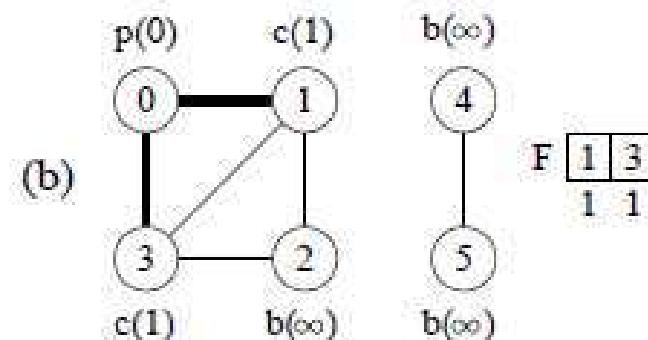
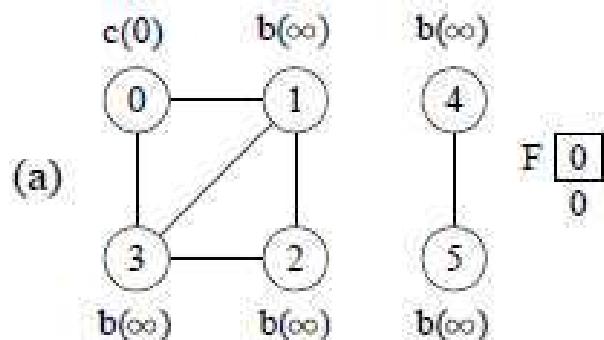


- 1) Desenfileira 4
- 2) Examina a lista de vértices adj. visita se branco
- visita se branco
- 3) Colore de cinza
- vértice 5
- 4) $d[5] = 1$
- 5) Enfileira 5
- 6) Colore 4 de preto

Exemplo



Exemplo



Percorso em largura: 0, 1, 3, 2, 4, 5

Busca em largura

Análise

- ◆ O custo de inicialização do primeiro anel em *BuscaEmLargura* é $O(|V|)$ cada um
- ◆ O custo do segundo anel é também $O(|V|)$
- ◆ *VisitaBfs*: enfileirar e desenfileirar têm custo $O(1)$, logo, o custo total com a fila é $O(|V|)$
- ◆ Cada lista de adjacentes é percorrida no máximo uma vez, quando o vértice é desenfileirado
- ◆ Desde que a soma de todas as listas de adjacentes é $O(|A|)$, o tempo total gasto com as listas de adjacentes é $O(|A|)$
- ◆ Complexidade total: é $O(|V| + |A|)$

Caminhos mais curtos

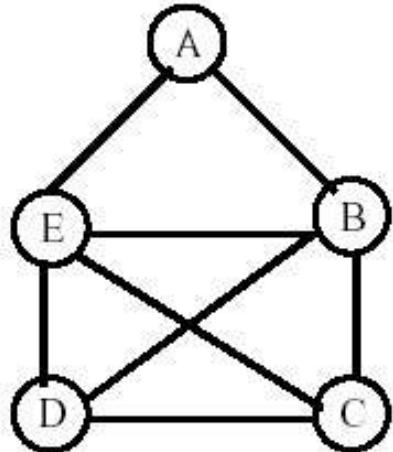
- ◆ A busca em largura obtém o caminho mais curto de u até v
- ◆ O procedimento *VisitaBfs* constrói uma árvore de busca em largura que é armazenada na variável *Antecessor*
- ◆ O programa a seguir imprime os vértices do caminho mais curto entre o vértice origem e outro vértice qualquer do grafo

```
void ImprimeCaminho(TipoValorVertice Origem, TipoValorVertice v,
                      TipoGrafo *Grafo, int *Dist, TipoCor *Cor,
                      int *Antecessor)
{ if (Origem == v) { printf("%d ", Origem); return; }
  if (Antecessor[v] == -1)
    printf("Nao existe caminho de %d ate %d", Origem, v);
  else { ImprimeCaminho(Origem, Antecessor[v],
                        Grafo, Dist, Cor, Antecessor);
         printf("%d ", v);
      }
}
```

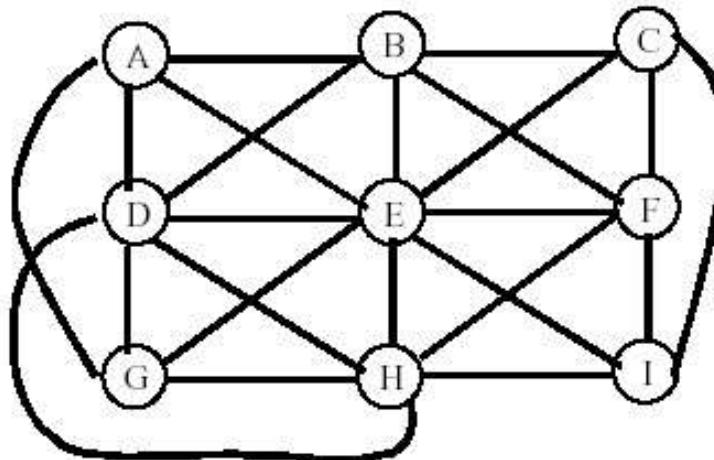
Exercícios

- ◆ Para cada um dos grafos indique o resultado (em termos dos vértices) dos percursos em profundidade e em largura iniciando no vértice A e supondo que a lista de adjacência esteja em ordem alfabética.
- ◆ Ilustre o progresso da busca. Ao lado de cada vértice indique a cor branca, cinza ou preta (b, c ou p), e entre parênteses tempo-de-descoberta/tempo-de-término. Indique também a fila F ao final de cada iteração do anel while para a busca em largura.

Exercícios



a)



b)

Para estudar

- ◆ Lista de exercícios sobre grafos disponível na equipe da disciplina no Teams

Bibliografia

Ziviani, N. Projeto de Algoritmos, Thomson Learning, 2005.

Ebook da edição de 2018 disponível na biblioteca

<https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788522126590>

Bacharelado em Ciência da Computação
GBC034 Algoritmos e Estruturas de Dados 2

Grafos

Caminhos mais curtos

Material baseado nos slides do professor Nivio Ziviani
(Projeto de Algoritmos)

Profa. Maria Camila Nardini Barioni

camila.barioni@ufu.br

Bloco B - sala 1B137

2º semestre de 2023

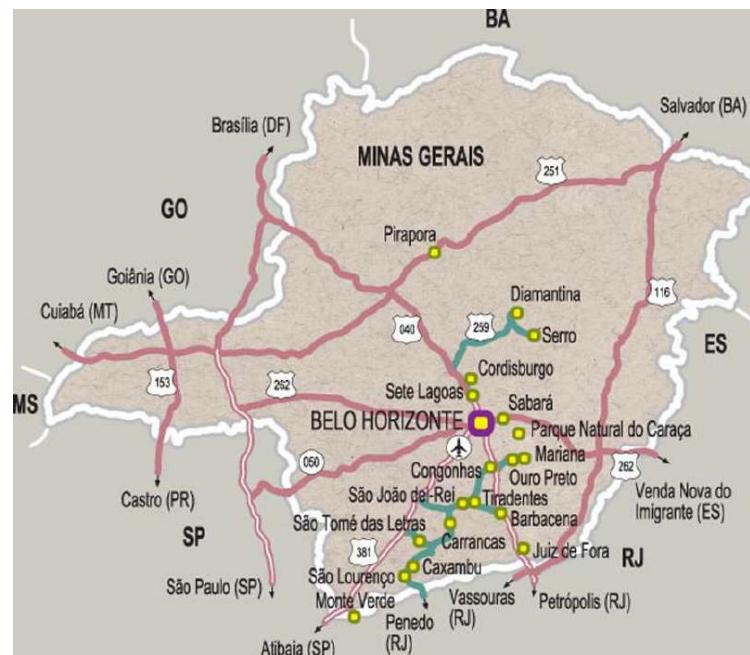
Roteiro

- 
- ◆ Definição do Problema
 - ◆ Algoritmo de Dijkstra
 - ◆ Exercício

Caminhos mais curtos

Aplicações

- ◆ Um motorista procura o caminho mais curto entre Diamantina e Ouro Preto
- ◆ Dado um mapa com as distâncias entre cada par de interseções adjacentes, como obter o caminho mais curto?



Malha rodoviária das principais rodovias de MG

Fonte: mapasblog.blogspot.com

Caminhos mais curtos

Aplicações

◆ Modelagem:

- $G = (V, A)$: grafo direcionado ponderado, mapa rodoviário
- V : interseções
- A : segmentos de estrada entre interseções
- $p(u, v)$: peso de cada aresta, distância entre interseções

◆ Peso de um caminho: $p(c) = \sum_{i=1}^k p(vi_{-1}, vi)$

◆ Caminho mais curto:

$$\delta(u, v) = \begin{cases} \min\{p(c) : \text{do caminho de } u \text{ a } v\}, & \text{se existir um caminho de } u \text{ a } v \\ \infty, & \text{caso contrário} \end{cases}$$

◆ Caminho mais curto do vértice u ao vértice v : qualquer caminho c com peso $p(c) = \delta(u, v)$

Caminhos mais curtos

Aplicações

◆ Modelagem:

- $G = (V, A)$: grafo direcionado ponderado, mapa rodoviário

O peso das arestas pode ser interpretado como outras métricas, por exemplo: tempo, custo, ou outra quantidade acumulada através do caminho que se deseja minimizar

◆ Peso de um caminho: $p(c) = \sum_{i=1}^n p(v_{i-1}, v_i)$

◆ Caminho mais curto:

$$\delta(u, v) = \begin{cases} \min\{p(c): \text{do caminho de } u \text{ a } v\}, & \text{se existir um caminho de } u \text{ a } v \\ \infty, & \text{caso contrário} \end{cases}$$

◆ Caminho mais curto do vértice u ao vértice v : qualquer caminho c com peso $p(c) = \delta(u, v)$

Definição do Problema

- ◆ O procedimento para realizar a busca em largura em um grafo $G = (V, A)$ obtém a distância do vértice de origem $u \in V$ para cada vértice alcançável $v \in V$
 - Obtém **o caminho mais curto** de u até v , onde os pesos das arestas são todos iguais
- ◆ **Caminhos mais curtos a partir de uma origem:** dado um grafo ponderado $G = (V, A)$, desejamos obter o caminho mais curto a partir de um dado vértice origem $s \in V$ até cada $v \in V$

Definição do Problema

- ◆ Muitos problemas podem ser resolvidos pelo algoritmo para o problema origem única:
 - **Caminhos mais curtos com destino único:** reduzido ao problema origem única invertendo a direção de cada aresta do grafo
 - **Caminhos mais curtos entre um par de vértices:** o algoritmo para origem única é a melhor opção conhecida
 - **Caminhos mais curtos entre todos os pares de vértices:** resolvido aplicando o algoritmo origem única $|V|$ vezes, uma vez para cada vértice origem

Definição do Problema

- ◆ Um caminho mais curto em um grafo $G = (V, A)$ não pode conter ciclos
- ◆ A remoção do ciclo de um caminho produz um caminho com mesmos vértices origem e destino e um caminho de menor peso
- ◆ Um caminho acíclico em G contém no máximo $|V|$ vértices e $|V| - 1$ arestas

Definição do Problema

- ◆ A representação de caminhos mais curtos pode ser realizada pela variável *Antecessor*
- ◆ Para cada vértice $v \in V$ o *Antecessor*[v] é um outro vértice $u \in V$ ou *nil*(-1)
- ◆ O algoritmo atribui a *Antecessor* os rótulos de vértices de uma cadeia de antecessores com origem em v e que anda para trás ao longo de um caminho mais curto até o vértice origem s

Definição do Problema

- ◆ Dado um vértice v no qual $\text{Antecessor}[v] \neq \text{nil}$, o procedimento *ImprimeCaminho* pode imprimir o caminho mais curto de s até v
- ◆ Os valores em $\text{Antecessor}[v]$, em um passo intermediário, não indicam necessariamente caminhos mais curtos
- ◆ Entretanto, ao final do processamento, Antecessor contém uma árvore de caminhos mais curtos definidos em termos dos pesos de cada aresta de G , ao invés do número de arestas
- ◆ Caminhos mais curtos não são necessariamente únicos

Árvores de caminhos mais curtos

- ◆ Uma **árvore de caminhos mais curtos** com raiz em $u \in V$ é um subgrafo direcionado $G' = (V', A')$, onde $V' \subseteq V$ e $A' \subseteq A$, tal que:
 1. V' é o conjunto de vértices alcançáveis a partir de $s \in G$,
 2. G' forma uma árvore de raiz s ,
 3. para todos os vértices $v \in V'$, o caminho simples de s até v é um caminho mais curto de s até v em G

Algoritmo de Dijkstra

- ◆ Mantém um conjunto S de vértices cujos caminhos mais curtos até um vértice origem já são conhecidos
- ◆ Produz uma árvore de caminhos mais curtos de um vértice origem s para todos os vértices que são alcançáveis a partir de s
- ◆ Utiliza a técnica de relaxamento:
 - Para cada vértice $v \in V$ o atributo $p[v]$ é um limite superior do peso de um caminho mais curto do vértice origem s até v
 - O vetor $p[v]$ contém uma estimativa de um caminho mais curto
- ◆ O primeiro passo do algoritmo é inicializar os antecessores e as estimativas de caminhos mais curtos:
 - $\text{Antecessor}[v] = \text{nil}$ para todo vértice $v \in V$
 - $p[u] = 0$, para o vértice origem s , e
 - $p[v] = \infty$ para $v \in V - \{s\}$

Relaxamento

- ◆ O **relaxamento** de uma aresta (u, v) consiste em verificar se é possível melhorar o melhor caminho até v obtido até o momento se passarmos por u
- ◆ Se isto acontecer, $p[v]$ e $\text{Antecessor}[v]$ devem ser atualizados

```
if (p[v] > p[u] + peso da aresta (u,v)) {  
    p[v] = p[u] + peso da aresta (u,v);  
    Antecessor [v] = u;  
}
```

O passo de relaxamento pode decrementar o valor da estimativa de caminho mais curto e atualizar o antecessor de v

Algoritmo de Dijkstra: 1º Refinamento

```
void Dijkstra (Grafo, Raiz)
{
    1. for (v=0; v < Grafo.NumVertices; v++)
    2.     p[v] = Infinito;
    3.     Antecessor [v] = -1;
    4. p[Raiz] = 0;           →
    5. Constroi heap no vetor A;
    6. S = Ø;
    7. while (heap > 1)
    8.     u = RetiraMin(A) ;
    9.     S = S + u;
    10.    for (v ∈ ListaAdjacentes[u] )
    11.        if (p[v] > p[u] + peso da aresta(u,v) )
    12.            p[v] = p[u] + peso da aresta(u,v) ;
    13.            Antecessor [v] = u;
}
```

Inicialização dos antecessores e das estimativas de caminhos mais curtos

Inicializa o vértice raiz

Algoritmo de Dijkstra: 1º Refinamento

```
void Dijkstra (Grafo, Raiz)
{
    1. for (v=0; v < Grafo.NumVertices; v++)
    2.     p[v] = Infinito;
    3.     Antecessor [v] = -1;
    4. p[Raiz] = 0;
    5. Constroi heap no vetor A; ——————> Sobre todos os vértices do grafo
    6. S = Ø; ——————> Inicializa o conjunto solução
    7. while (heap > 1)
    8.     u = RetiraMin(A) ;
    9.     S = S + u;
    10.    for (v ∈ ListaAdjacentes[u] )
    11.        if (p[v] > p[u] + peso da aresta(u,v) )
    12.            p[v] = p[u] + peso da aresta(u,v) ;
    13.            Antecessor [v] = u;
}
```

Sobre todos os vértices do grafo

Inicializa o conjunto solução

Algoritmo de Dijkstra: 1º Refinamento

```
void Dijkstra (Grafo, Raiz)
{
    1. for (v=0; v < Grafo.NumVertices; v++)
    2.     p[v] = Infinito;
    3.     Antecessor [v] = -1;
    4. p[Raiz] = 0;
    5. Constroi heap no vetor A;
    6. S = Ø;
    7. while (heap > 1)
    8.     u = RetiraMin(A) ;
    9.     S = S + u;
    10.    for (v ∈ ListaAdjacentes[u] )
    11.        if (p[v] > p[u] + peso da aresta(u,v) )
    12.            p[v] = p[u] + peso da aresta(u,v) ;
    13.            Antecessor [v] = u;
}
```

Executa enquanto heap for diferente de zero

Algoritmo de Dijkstra: 1º Refinamento

```
void Dijkstra (Grafo, Raiz)
{
    1. for (v=0; v < Grafo.NumVertices; v++)
    2.     p[v] = Infinito;
    3.     Antecessor [v] = -1;
    4. p[Raiz] = 0;
    5. Constroi heap no vetor A;
    6. S = Ø;
    7. while (heap > 1)
    8.     u = RetiraMin(A) ;
    9.     S = S + u;
    10.    for (v ∈ ListaAdjacentes[u] )
    11.        if (p[v] > p[u] + peso da aresta(u,v) )
    12.            p[v] = p[u] + peso da aresta(u,v) ;
    13.            Antecessor [v] = u;
}
```

- Executa enquanto heap for diferente de zero
- O número de elementos do heap é igual a $V - S$

Algoritmo de Dijkstra: 1º Refinamento

```
void Dijkstra (Grafo, Raiz)
{
    1. for (v=0; v < Grafo.NumVertices; v++)
    2.     p[v] = Infinito;
    3.     Antecessor [v] = -1;
    4. p[Raiz] = 0;
    5. Constroi heap no vetor A;
    6. S = Ø;
    7. while (heap > 1)
    8.     u = RetiraMin(A) ;
    9.     S = S + u;
    10.    for (v ∈ ListaAdjacentes[u] )
    11.        if (p[v] > p[u] + peso da aresta(u,v) )
    12.            p[v] = p[u] + peso da aresta(u,v) ;
    13.            Antecessor [v] = u;
}
```

A cada iteração um vértice u é
retirado da heap e adicionado a S

Algoritmo de Dijkstra: 1º Refinamento

```
void Dijkstra (Grafo, Raiz)
{
    1. for (v=0; v < Grafo.NumVertices; v++)
    2.     p[v] = Infinito;
    3.     Antecessor [v] = -1;
    4. p[Raiz] = 0;
    5. Constroi heap no vetor A;
    6. S = Ø;
    7. while (heap > 1)
    8.     u = RetiraMin(A) ;
    9.     S = S + u;
    10.    for (v ∈ ListaAdjacentes[u] )
    11.        if (p[v] > p[u] + peso da aresta(u,v) )
    12.            p[v] = p[u] + peso da aresta(u,v) ;
    13.            Antecessor [v] = u;
}
```

u contém o caminho mais curto estimado até o momento

Algoritmo de Dijkstra: 1º Refinamento

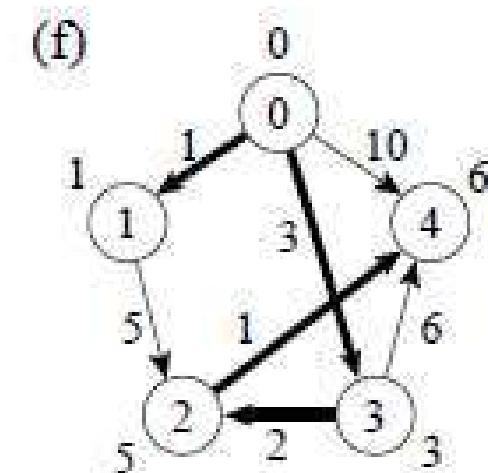
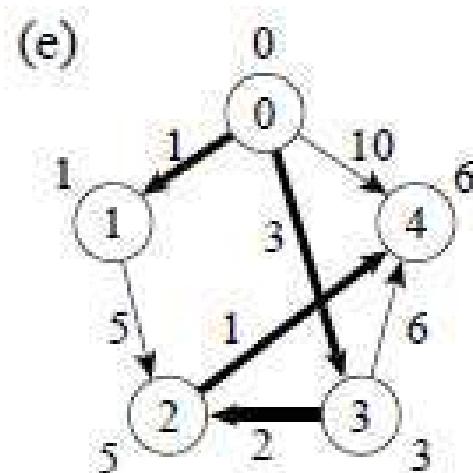
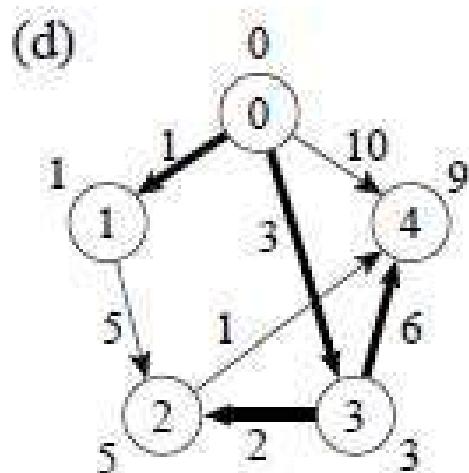
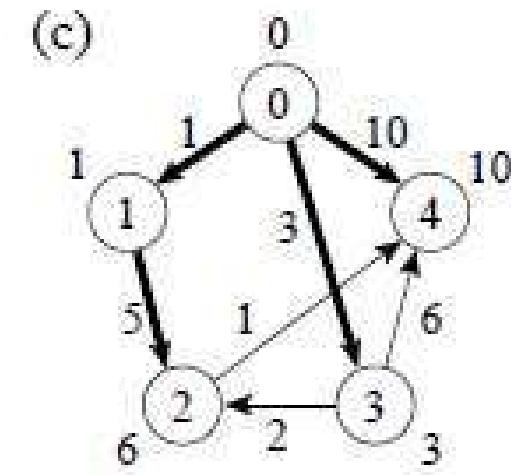
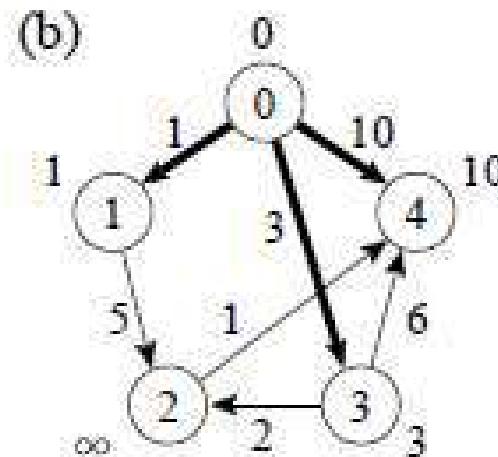
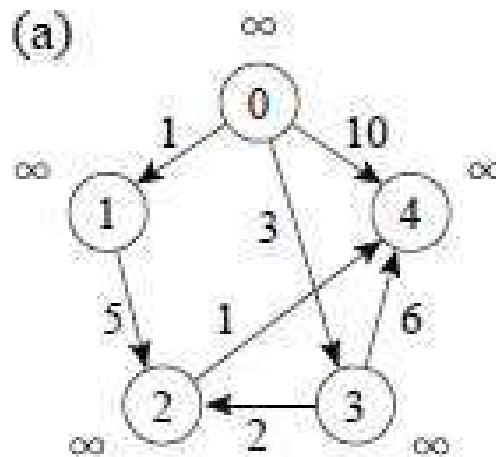
```
void Dijkstra (Grafo, Raiz)
{
    1. for (v=0; v < Grafo.NumVertices; v++)
    2.     p[v] = Infinito;
    3.     Antecessor [v] = -1;
    4. p[Raiz] = 0;
    5. Constroi heap no vetor A;
    6. S = Ø;
    7. while (heap > 1)
    8.     u = RetiraMin(A) ;
    9.     S = S + u;
    10.    for (v ∈ ListaAdjacentes[u] )
    11.        if (p[v] > p[u] + peso da aresta(u,v) )
    12.            p[v] = p[u] + peso da aresta(u,v) ;
    13.            Antecessor [v] = u;
}
```

A operação de relaxamento é realizada sobre cada aresta (u, v) adjacente a u

Algoritmo de Dijkstra: 1º Refinamento

- ◆ Invariante: o número de elementos do heap é igual a $V - S$ no início do anel **while**
- ◆ A cada iteração do **while**, um vértice u é extraído do heap e adicionado ao conjunto S , mantendo assim o invariante
- ◆ *RetiraMin* obtém o vértice u com o caminho mais curto estimado até o momento e adiciona ao conjunto S
- ◆ No anel da linha 10, a operação de relaxamento é realizada sobre cada aresta (u, v) adjacente ao vértice u

Algoritmo de Dijkstra: 1º Refinamento



Algoritmo de Dijkstra: 1º Refinamento

(a)

∞

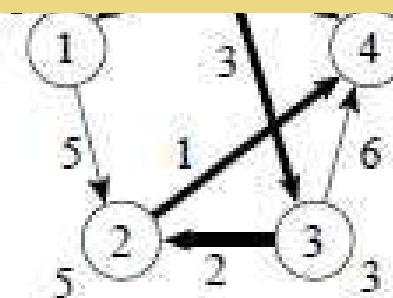
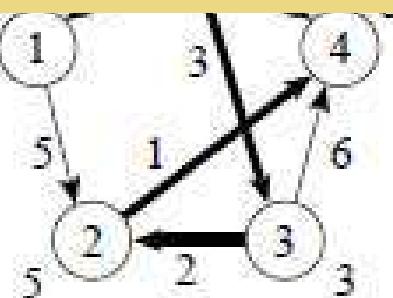
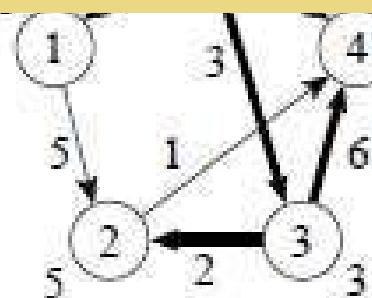
(b)

0

(c)

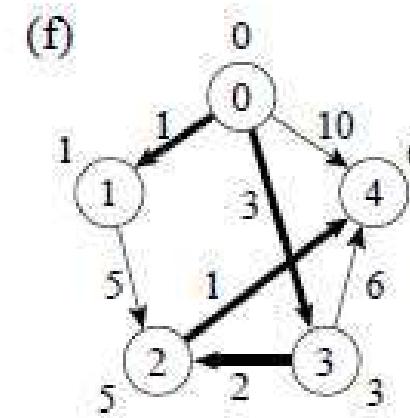
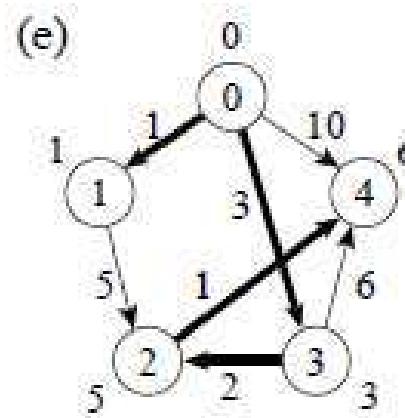
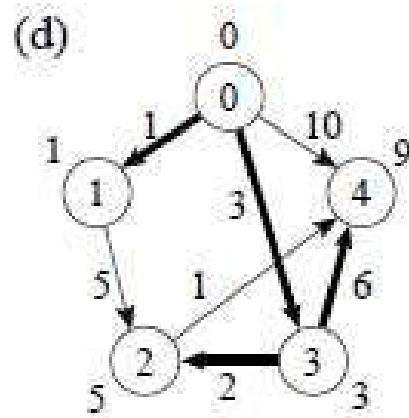
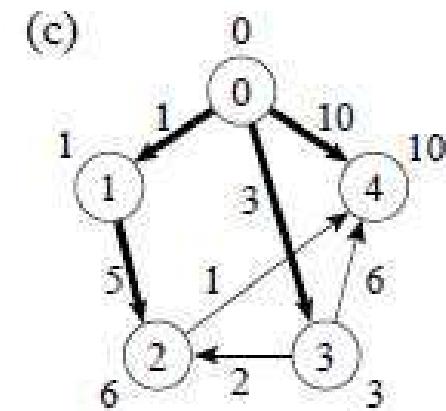
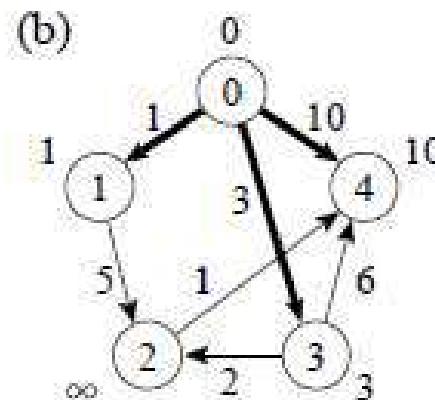
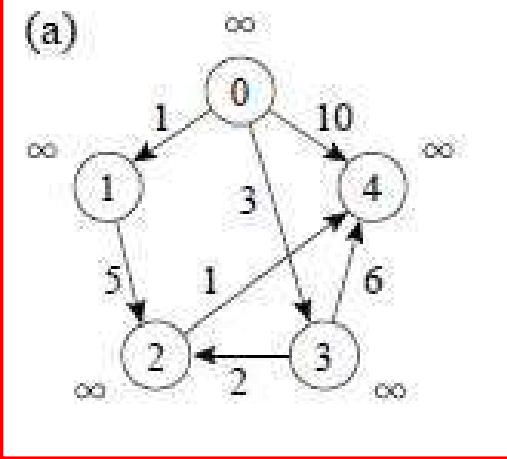
0

- A árvore começa pelo vértice 0
- A cada passo, um vértice é adicionado à árvore S
- Arestras em **negrito** pertencem à árvore de caminhos mais curtos sendo construída
- Esta estratégia é gulosa
 - A árvore é aumentada a cada passo com uma aresta que contribui com o mínimo possível para o custo total de cada caminho



Algoritmo de Dijkstra: 1º Refinamento

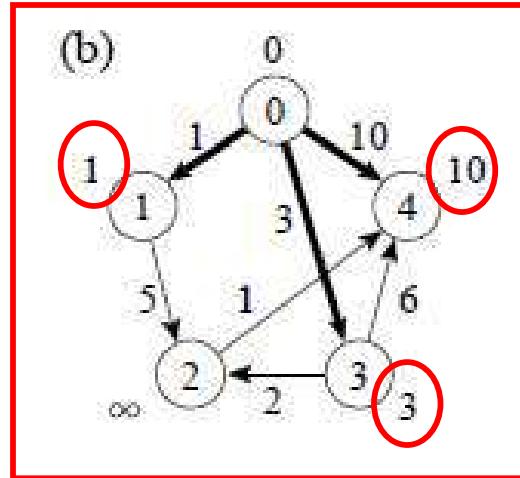
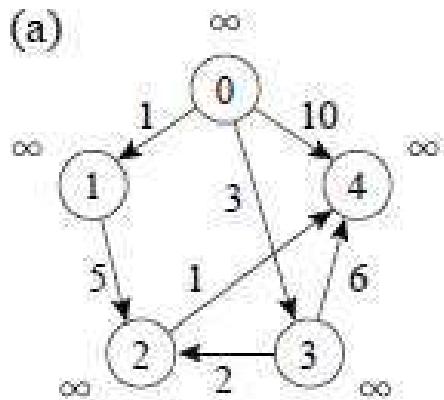
estimativas de caminhos mais curtos inicializadas com infinito



Iteração	S	$d[0]$	$d[1]$	$d[2]$	$d[3]$	$d[4]$
(a)	\emptyset	∞	∞	∞	∞	∞
(b)	{0}	0	1	∞	3	10
(c)	{0, 1}	0	1	6	3	10

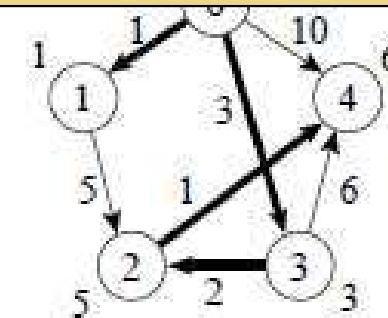
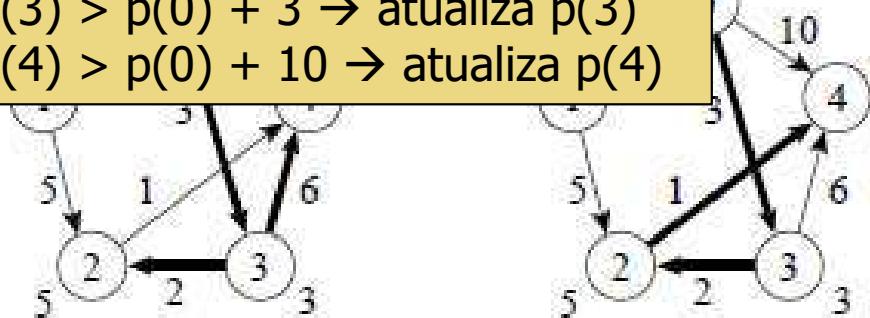


Algoritmo de Dijkstra: 1º Refinamento



- O vértice 0 é retirado da heap e adicionado a S
- $p(0) = 0$
- A operação de relaxamento é realizada sobre cada aresta (u, v) adjacente ao vértice 0

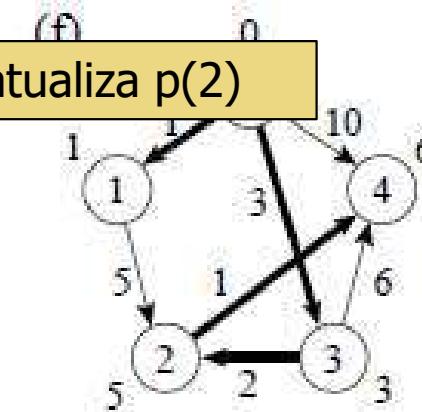
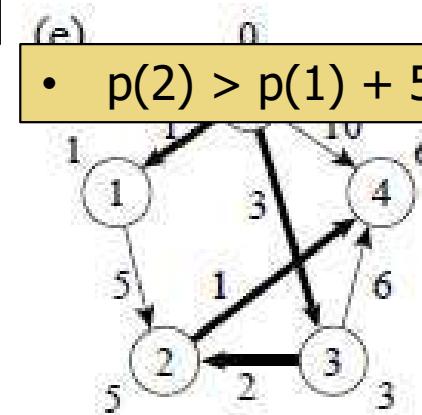
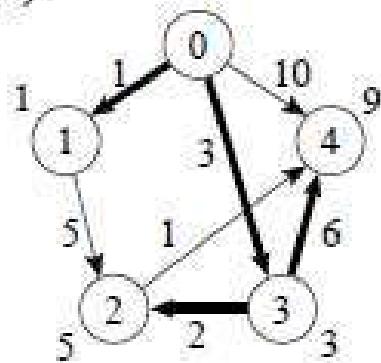
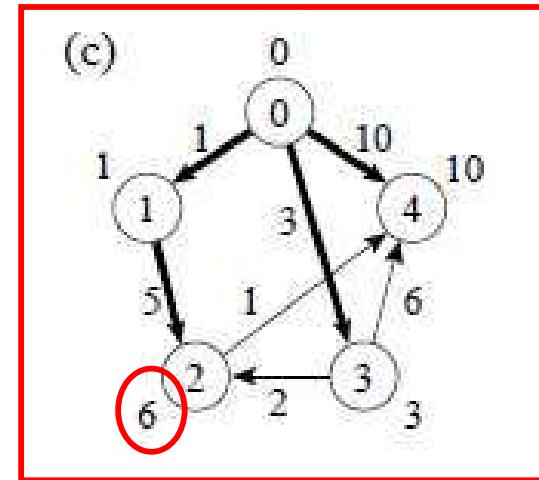
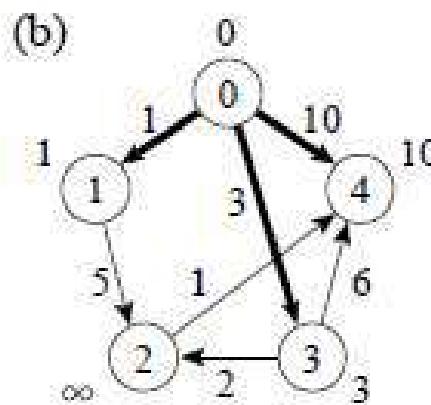
- $p(1) > p(0) + 1 \rightarrow$ atualiza $p(1)$
- $p(3) > p(0) + 3 \rightarrow$ atualiza $p(3)$
- $p(4) > p(0) + 10 \rightarrow$ atualiza $p(4)$



Iteração	S	$d[0]$	$d[1]$	$d[2]$	$d[3]$	$d[4]$
(a)	\emptyset	∞	∞	∞	∞	∞
(b)	{0}	0	1	∞	3	10
(c)	{0, 1}	0	1	6	3	10

Algoritmo de Dijkstra: 1º Refinamento

- O vértice 1 é retirado da heap e adicionado a S
- A operação de relaxamento é realizada sobre cada aresta (u, v) adjacente ao vértice 1



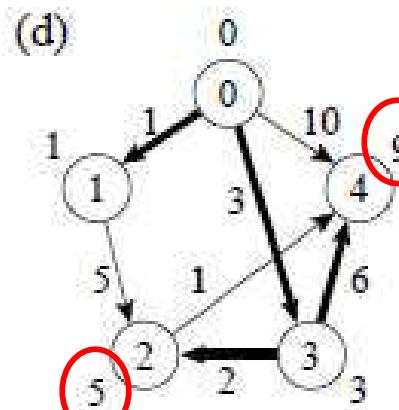
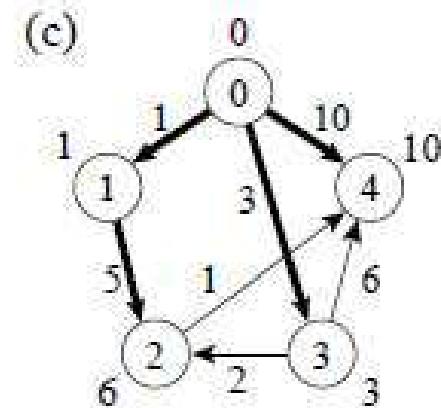
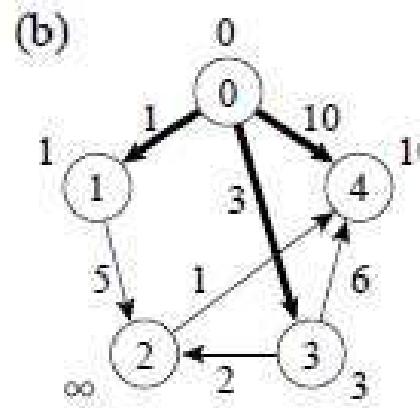
- $p(2) > p(1) + 5 \rightarrow \text{atualiza } p(2)$

Iteração	S	$d[0]$	$d[1]$	$d[2]$	$d[3]$	$d[4]$
(a)	\emptyset	∞	∞	∞	∞	∞
(b)	{0}	0	1	∞	3	10
(c)	{0, 1}	0	1	6	3	10

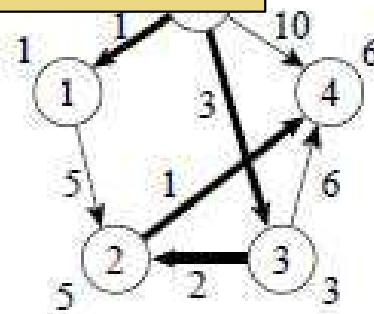
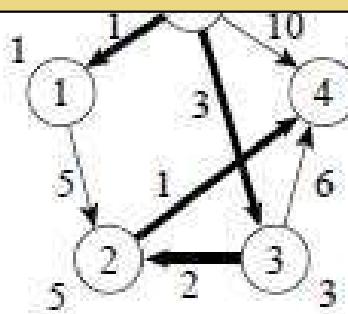


Algoritmo de Dijkstra: 1º Refinamento

- O vértice 3 é retirado da heap e adicionado a S
- A operação de relaxamento é realizada sobre cada aresta (u, v) adjacente ao vértice 3



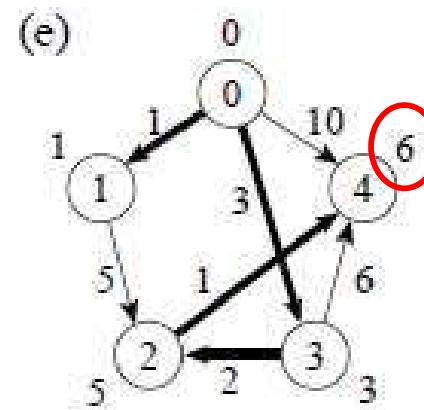
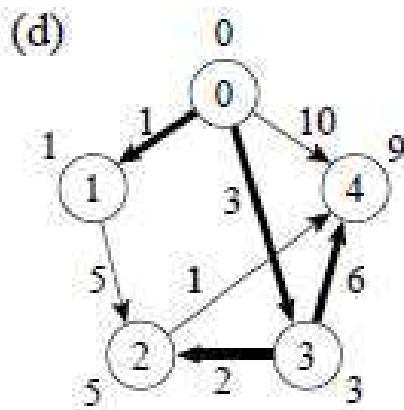
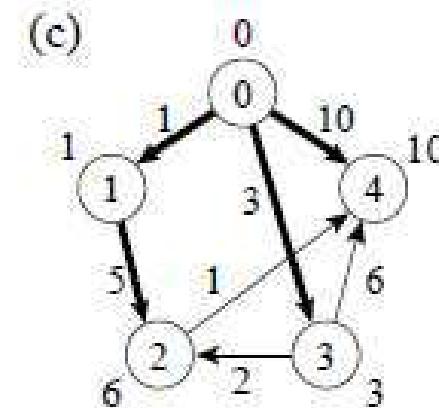
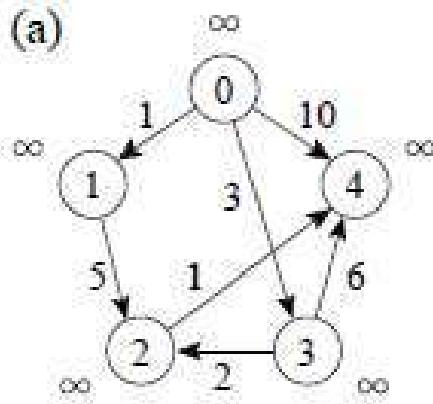
- $p(2) > p(3) + 2 \rightarrow$ atualiza $p(2)$
- $p(4) > p(3) + 6 \rightarrow$ atualiza $p(4)$



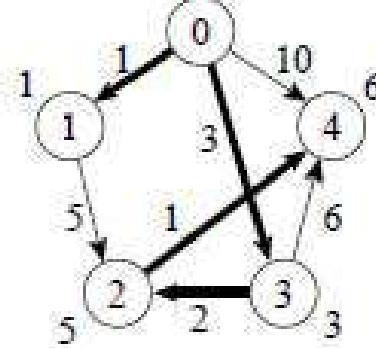
Iteração	S	$d[0]$	$d[1]$	$d[2]$	$d[3]$	$d[4]$
(d)	{0, 1, 3}	0	1	5	3	9
(e)	{0, 1, 3, 2}	0	1	5	3	6
(f)	{0, 1, 3, 2, 4}	0	1	5	3	6

Algoritmo de Dijkstra

- O vértice 2 é retirado da heap e adicionado a S
- A operação de relaxamento é realizada sobre cada aresta (u, v) adjacente ao vértice 2



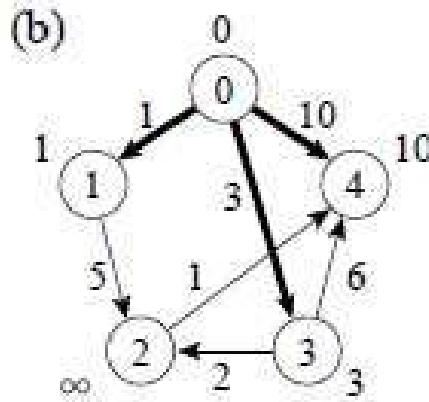
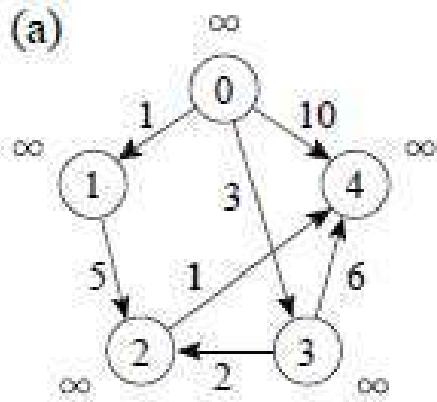
$p(4) > p(2) + 1 \rightarrow$ atualiza $p(2)$



Iteração	S	$d[0]$	$d[1]$	$d[2]$	$d[3]$	$d[4]$
(d)	{0, 1, 3}	0	1	5	3	9
(e)	{0, 1, 3, 2}	0	1	5	3	6
(f)	{0, 1, 3, 2, 4}	0	1	5	3	6

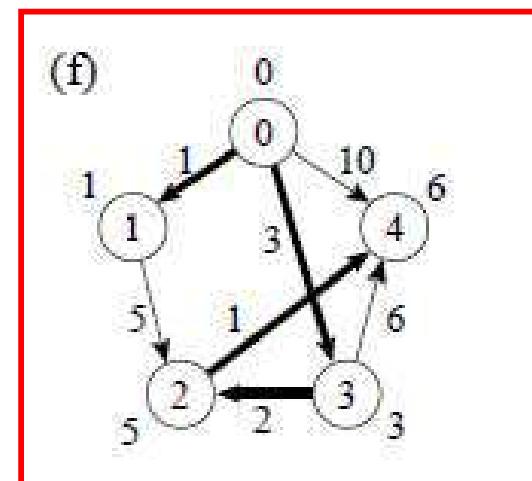
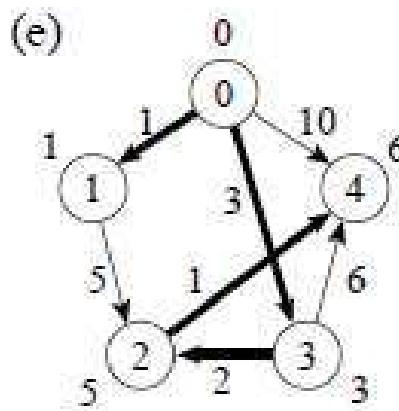
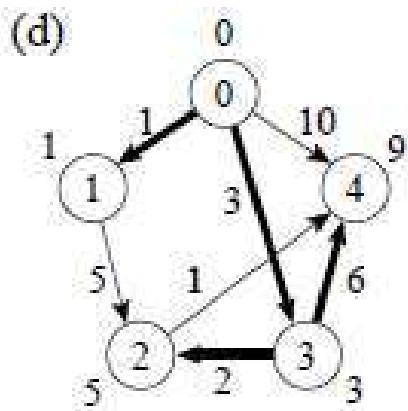


Algoritmo de Dijkstra: 1º Refinamento



(c)

- O vértice 4 é retirado da heap e adicionado a S
- Lista de adjacentes vazia



Iteração	S	$d[0]$	$d[1]$	$d[2]$	$d[3]$	$d[4]$
(d)	{0, 1, 3}	0	1	5	3	9
(e)	{0, 1, 3, 2}	0	1	5	3	6
(f)	{0, 1, 3, 2, 4}	0	1	5	3	6

Algoritmo de Dijkstra

- ◆ Para realizar de forma eficiente a seleção de uma nova aresta, todos os vértices que não estão na árvore de caminhos mais curtos residem no heap A baseada no campo p
- ◆ Para cada vértice v , $p[v]$ é o caminho mais curto obtido até o momento, de v até o vértice raiz
- ◆ O heap mantém os vértices, mas a condição do heap é mantida pelo caminho mais curto estimado até o momento através do arranjo $p[v]$, o heap é indireto
- ◆ O arranjo $Pos[v]$ fornece a posição do vértice v dentro do heap A , permitindo assim que o vértice v possa ser acessado a um custo $O(1)$ para a operação $DiminuiChaveInd$

Algoritmo de Dijkstra

```
/** Entram aqui os operadores de uma das implementações de grafos, bem como o
operador Constroi da implementação de filas de prioridades, assim como os operadores
RefazInd, RetiraMinInd e DiminuiChaveInd do Programa Constroi **/
```

```
void Dijkstra(TipoGrafo *Grafo, TipoValorVertice *Raiz)
{ TipoPeso P[MAXNUMVERTICES + 1];
  TipoValorVertice Pos[MAXNUMVERTICES + 1];
  long Antecessor[MAXNUMVERTICES + 1];
  short Itensheap[MAXNUMVERTICES + 1];
  TipoVetor A;
  TipoValorVertice u, v;
  TipoItem temp;
  for (u = 0; u <= Grafo->NumVertices; u++) {
    /*Constroi o heap com todos os valores igual a INFINITO*/
    Antecessor[u] = -1; P[u] = INFINITO;
    A[u+1].Chave = u; /*Heap a ser construido*/
    Itensheap[u] = TRUE; Pos[u] = u + 1;
  }
  n = Grafo->NumVertices;
  P[*Raiz] = 0;
  Constroi(A, P, Pos);
```

Algoritmo de Dijkstra

```
while (n >= 1) { /*enquanto heap nao vazio*/
    temp = RetiraMinInd(A, P, Pos);
    u = temp.Chave; Itensheap[u] = FALSE;
    if (!ListaAdjVazia(&u, Grafo))
    { Aux = PrimeiroListaAdj(&u, Grafo); FimListaAdj = FALSE;
        while (!FimListaAdj)
        { ProxAdj(&u, Grafo, &v, &Peso, &Aux, &FimListaAdj);
            if (P[v] > (P[u] + Peso))
            { P[v] = P[u] + Peso; Antecessor[v] = u;
                DiminuiChaveInd(Pos[v], P[v], A, P, Pos);
                printf("Caminho: v[%d] v[%ld] d[%d]",
                       v, Antecessor[v], P[v]);
                scanf("%*[^\n]");
                getchar();
            }
        }
    }
}
```

Análise do algoritmo de Dijkstra

- ◆ O desempenho depende da forma como a fila de prioridades é implementada
- ◆ Se a fila de prioridades é implementada como um heap
 - A inicialização de A é realizada com um custo $O(|V|)$
 - O corpo do anel while mais externo é executado $|V|$ vezes e considerando que a operação que refaz o heap tem custo $O(\log |V|)$, o custo da operação de retirada do item com menor peso é $O(|V| \log |V|)$
 - O while mais interno para percorrer a lista de adjacentes é executado $O(|A|)$ vezes e a operação DiminuiChave sobre o heap A tem custo $O(\log |V|)$
 - O tempo total para executar o algoritmo Dijkstra é $O(|V| \log |V|) + O(|A| \log |V|) = O(|A| \log |V|)$

Porque o algoritmo de Dijkstra funciona

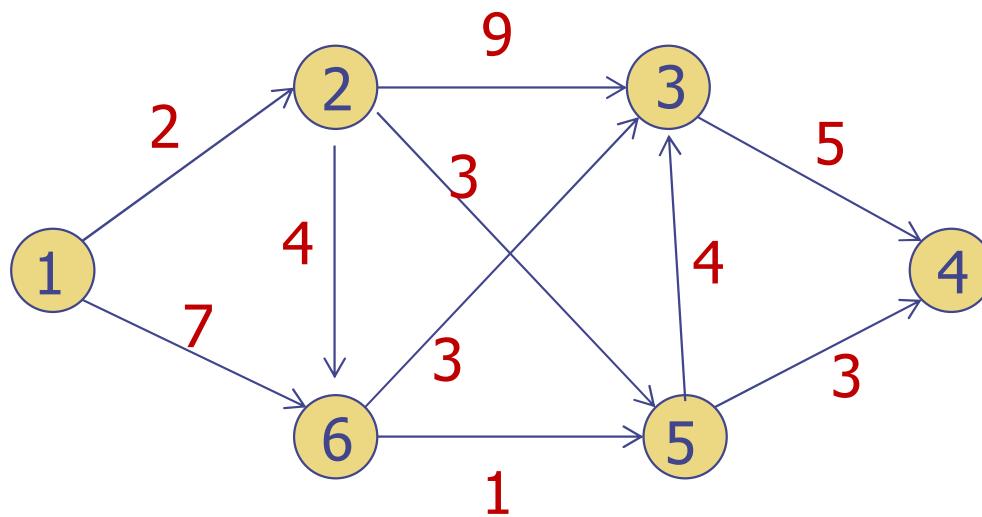
- ◆ O algoritmo usa uma estratégia gulosa: sempre escolher o vértice mais leve (ou o mais perto) em $V - S$ para adicionar ao conjunto solução S
- ◆ O algoritmo de Dijkstra sempre obtém os caminhos mais curtos, pois cada vez que um vértice é adicionado ao conjunto S temos que $p[u] = (\text{Raiz}, u)$

Exercícios (em duplas)

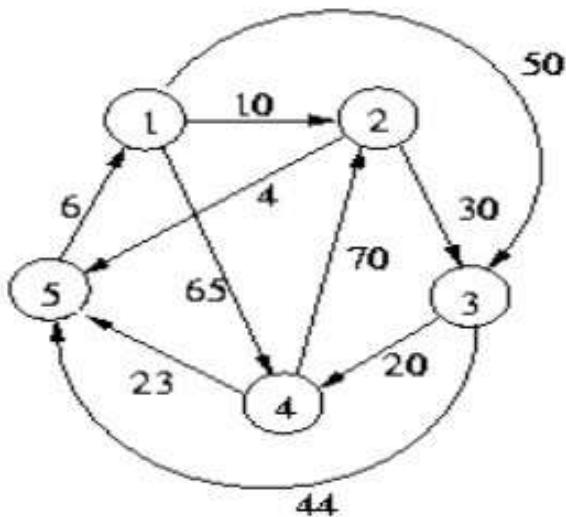
- ◆ Considerando os Grafos a seguir, mostre o quadro dos passos da execução do algoritmo Dijkstra a partir do vértice 1, colocando em **negrito** as arestas que pertencem à árvore de caminhos mais curtos sendo construída

Exercícios (em duplas)

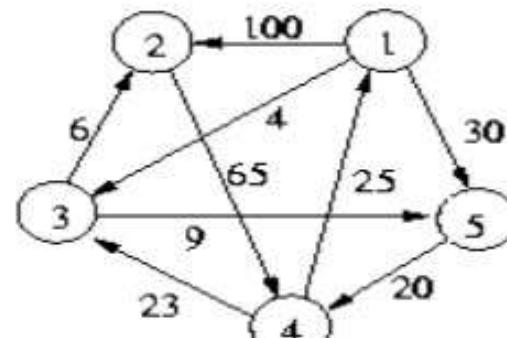
a)



Exercícios adicionais para praticar



b)



c)

Bibliografia



Ziviani, N. Projeto de Algoritmos, Thomson Learning, 2005.

Ebook da edição de 2018 disponível na biblioteca
<https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788522126590>

Matéria Prova 1

◆ Matéria

- Complexidade de Algoritmos
- Recursividade
- Ordenação
- Grafos
 - ◆ Introdução
 - ◆ Representação
 - ◆ Algoritmos de busca

◆ Data: 05/03 (em sala de aula)

Bacharelado em Ciência da Computação
GBC034 Algoritmos e Estruturas de Dados 2

Grafos

Aplicações de busca

Material baseado nos slides do professor Nivio Ziviani
(Projeto de Algoritmos)

Profa. Maria Camila Nardini Barioni

camila.barioni@ufu.br

Bloco B - sala 1B137

2º semestre de 2023

Roteiro

- 
- ◆ Relembrando busca em profundidade
 - ◆ Ordenação topológica
 - ◆ Componentes fortemente conectados
 - ◆ Exercícios

Relembrando busca em profundidade

- ◆ A busca em profundidade, (do inglês *depth-first search*), é um algoritmo para caminhar no grafo
- ◆ A estratégia é buscar o mais profundo no grafo sempre que possível
- ◆ As arestas são exploradas a partir do vértice v mais recentemente descoberto que ainda possui arestas não exploradas saindo dele

Relembrando busca em profundidade

- ◆ Quando todas as arestas adjacentes a v tiverem sido exploradas a busca anda para trás (do inglês *backtrack*) para explorar vértices que saem do vértice do qual v foi descoberto
- ◆ O algoritmo é a base para muitos outros algoritmos importantes, tais como verificação de grafos acíclicos, **ordenação topológica** e **componentes fortemente conectados**

Relembrando busca em profundidade

- ◆ Para acompanhar o progresso do algoritmo cada vértice é colorido de branco, cinza ou preto
- ◆ Todos os vértices são inicializados branco
- ◆ Quando um vértice é descoberto pela primeira vez ele torna-se cinza, e é tornado preto quando sua lista de adjacentes tenha sido completamente examinada
- ◆ $d[v]$: tempo de descoberta
- ◆ $t[v]$: tempo de término do exame da lista de adjacentes de v
- ◆ Estes registros são inteiros entre 1 e $2|V|$ pois existe um evento de descoberta e um evento de término para cada um dos $|V|$ vértices

Relembrando busca em profundidade

```
void BuscaEmProfundidade(TipoGrafo *Grafo)
{ TipoValorVertice x;
  TipoValorTempo Tempo;
  TipoValorTempo d[MAXNUMVERTICES + 1], t[MAXNUMVERTICES + 1];
  TipoCor Cor[MAXNUMVERTICES+1];
  short Antecessor[MAXNUMVERTICES+1];
  Tempo = 0; —————>
  for (x = 0; x <= Grafo->NumVertices - 1; x++) {
    Cor[x] = branco;
    Antecessor[x] = -1;
  }
  for (x = 0; x <= Grafo->NumVertices - 1; x++) {
    if (Cor[x] == branco)
      VisitaDfs(x, Grafo, &Tempo, d, t, Cor, Antecessor);
  }
}
```

verifica cada vértice e quando encontra um vértice branco visita esse vértice

usada para registrar os tempos de descoberta e término

inicializa os vértices de branco e os antecessores para -1

Relembrando busca em profundidade

```
void BuscaEmProfundidade(TipoGrafo *Grafo)
{ TipoValorVertice x;
  TipoValorTempo Tempo;
  TipoValorTempo d[MAXNUMVERTICES + 1], t[MAXNUMVERTICES + 1];
  Tip...
show...
Temp...
for...
}
for (x = 0; x <= Grafo->NumVertices - 1; x++) {
    if (Cor[x] == branco)
        VisitaDfs(x, Grafo, &Tempo, d, t, Cor, Antecessor);
}
```

Toda vez que $VisitaDfs(u)$ é chamado, o vértice u se torna a raiz de uma nova **árvore de busca em profundidade**, e o conjunto de árvores forma uma **floresta** de árvores de busca

verifica cada vértice e quando encontra um vértice branco visita esse vértice

Relembrando busca em profundidade

```
void VisitaDfs(TipoValorVertice u, TipoGrafo *Grafo,
                TipoValorTempo* Tempo, TipoValorTempo* d,
                TipoValorTempo* t, TipoCor* Cor, short* Antecessor)
{ char FimListaAdj; TipoValorAresta Peso; TipoApontador Aux;
  TipoValorVertice v; Cor[u] = cinza; (*Tempo)++; d[u] = (*Tempo);
  printf("Visita%2d Tempo descoberta:%2d cinza\n", u, d[u]); getchar();
  if (!ListaAdjVazia(&u, Grafo))
  { Aux = PrimeiroListaAdj(&u, Grafo);
    FimListaAdj = FALSE;
    while (!FimListaAdj) {
      ProxAdj(&u, &v, &Peso, &Aux, &FimListaAdj);
      if (Cor[v] == branco) {
        Antecessor[v] = u;
        VisitaDfs(v, Grafo, Tempo, d, t, Cor, Antecessor);
      }
    }
  Cor[u] = preto; (*Tempo)++; t[u] = (*Tempo);
  printf("Visita%2d Tempo termino:%2d preto\n", u, t[u]); getchar();
}
```

Relembrando busca em profundidade

```
void VisitaDfs(TipoValorVertice u, TipoGrafo *Grafo,
                TipoValorTempo* Tempo, TipoValorTempo* d,
                TipoValorTempo* t, TipoCor* Cor, short* Antecessor)
{ char FimListaAdj; TipoValorAresta Peso; TipoApontador Aux;
  TipoValorVertice v; Cor[u] = cinza; (*Tempo)++; d[u] = (*Tempo);
  printf("Visita%2d Tempo descoberta:%2d cinza\n", u, d[u]); getchar();
  if (!ListaAdjVazia(&u, Grafo))
  { Aux = PrimeiroListaAdj(&u, Grafo);
    FimListaAdj = FALSE;
    while (!FimListaAdj) {
      ProxAdj(&u, &v, &Peso, &Aux, &FimListaAdj);
      if (Cor[v] == branco) {
        Antecessor[v] = u;
        VisitaDfs(v, Grafo, Tempo, d, t, Cor, Antecessor);
      }
    }
    Cor[u] = preto; (*Tempo)++; t[u] = (*Tempo);
    printf("Visita%2d Tempo termino:%2d preto\n", u, t[u]); getchar();
  }
}
```

Relembrando busca em profundidade

A variável *Tempo* é incrementada

```
void VisitaDfs(TipoValorVertice u, TipoGrafo *Grafo,
                 TipoValorTempo* Tempo, TipoValorTempo* d,
                 TipoValorTempo* t, TipoCor* Cor, short* Antecessor)
{ char FimListaAdj; TipoValorAresta Peso; TipoApontador Aux;
  TipoValorVertice v; Cor[u] = cinza; (*Tempo)++; d[u] = (*Tempo);
  printf("Visita%2d Tempo descoberta:%2d cinza\n", u, d[u]); getchar();
  if (!ListaAdjVazia(&u, Grafo))
  { Aux = PrimeiroListaAdj(&u, Grafo);
    FimListaAdj = FALSE;
    while (!FimListaAdj) {
      ProxAdj(&u, &v, &Peso, &Aux, &FimListaAdj);
      if (Cor[v] == branco) {
        Antecessor[v] = u;
        VisitaDfs(v, Grafo, Tempo, d, t, Cor, Antecessor);
      }
    }
  Cor[u] = preto; (*Tempo)++; t[u] = (*Tempo);
  printf("Visita%2d Tempo termino:%2d preto\n", u, t[u]); getchar();
}
```

Relembrando busca em pr

```
void VisitaDfs(TipoValorVertice u, TipoGrafo *Grafo,
                TipoValorTempo* Tempo, TipoValorTempo* d,
                TipoValorTempo* t, TipoCor* Cor, short* Antecessor)
{ char FimListaAdj; TipoValorAresta Peso; TipoApontador Aux;
  TipoValorVertice v; Cor[u] = cinza; (*Tempo)++; d[u] = (*Tempo);
  printf("Visita%2d Tempo descoberta:%2d cinza\n", u, d[u]); getchar();
  if (!ListaAdjVazia(&u, Grafo))
  { Aux = PrimeiroListaAdj(&u, Grafo);
    FimListaAdj = FALSE;
    while (!FimListaAdj) {
      ProxAdj(&u, &v, &Peso, &Aux, &FimListaAdj);
      if (Cor[v] == branco) {
        Antecessor[v] = u;
        VisitaDfs(v, Grafo, Tempo, d, t, Cor, Antecessor);
      }
    }
  Cor[u] = preto; (*Tempo)++; t[u] = (*Tempo);
  printf("Visita%2d Tempo termino:%2d preto\n", u, t[u]); getchar();
}
```

E o novo valor de *Tempo* é registrado como o tempo de descoberta $d[u]$

Relembrando busca em profundidade

```
void VisitaDfs(TipoValorVertice u, TipoGrafo *Grafo,
                TipoValorTempo* Tempo, TipoValorTempo* d,
                TipoValorTempo* t, TipoCor* Cor, short* Antecessor)
{ char FimListaAdj; TipoValorAresta Peso; TipoApontador Aux;
  TipoValorVertice v; Cor[u] = cinza; (*Tempo)++; d[u] = (*Tempo);
  printf("Visita%2d Tempo descoberta:%2d cinza\n", u, d[u]); getchar();
  if (!ListaAdjVazia(&u, Grafo))
  { Aux = PrimeiroListaAdj(&u, Grafo);
    FimListaAdj = FALSE;
    while (!FimListaAdj) {
      ProxAdj(&u, &v, &Peso, &Aux, &FimListaAdj);
      if (Cor[v] == branco) {
        Antecessor[v] = u;
        VisitaDfs(v, Grafo, Tempo, d, t, Cor, Antecessor);
      }
    }
  }
  Cor[u] = preto; (*Tempo)++; t[u] = (*Tempo);
  printf("Visita%2d Tempo termino:%2d preto\n", u, t[u]); getchar();
}
```

Examina lista de vértices adjacentes visitando-os recursivamente se eles forem brancos

Relembrando busca em profundidade

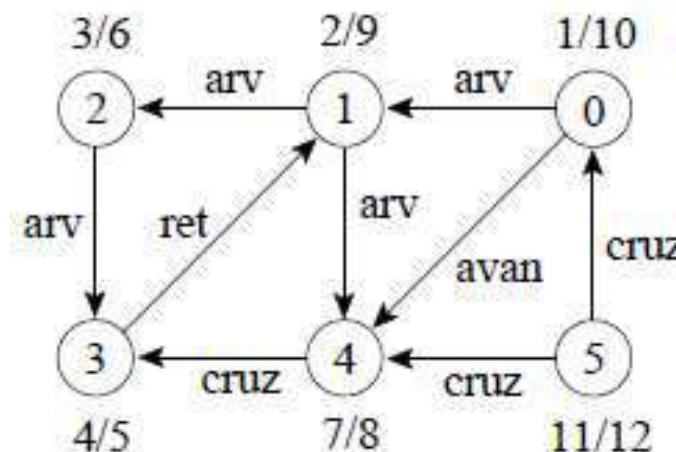
```
void VisitaDfs(TipoValorVertice u, TipoGrafo *Grafo,
                TipoValorTempo* Tempo, TipoValorTempo* d,
                TipoValorTempo* t, TipoCor* Cor, short* Antecessor)
{ char FimListaAdj; TipoValorAresta Peso; TipoApontador Aux;
  TipoValorVertice v; Cor[u] = cinza; (*Tempo)++; d[u] = (*Tempo);
  printf("Visita%2d Tempo descoberta:%2d cinza\n", u, d[u]); getchar();
  if (!ListaAdjVazia(&u, Grafo))
  {
    Quando VisitaDfs retorna, cada vértice  $u$  possui
    um tempo de descoberta  $d[u]$  e um tempo de
    término  $t[u]$ 
    Antecessor[v] = u;
    VisitaDfs(v, Grafo, Tempo, d, t, Cor, Antecessor);
  }
}
Cor[u] = preto; (*Tempo)++; t[u] = (*Tempo);
printf("Visita%2d Tempo termino:%2d preto\n", u, t[u]); getchar();
}
```

Classificação de arestas

- ◆ Classificação de arestas pode ser útil para derivar outros algoritmos
 - **Arestas de árvore:** são arestas de uma árvore de busca em profundidade. A aresta (u, v) é uma aresta de árvore se v foi descoberto pela primeira vez ao percorrer a aresta (u, v)
 - **Arestas de retorno:** conectam um vértice u com um antecessor v em uma árvore de busca em profundidade (inclui *self-loops*)
 - **Arestas de avanço:** não pertencem à árvore de busca em profundidade mas conectam um vértice a um descendente que pertence à árvore de busca em profundidade
 - **Arestas de cruzamento:** podem conectar vértices na mesma árvore de busca em profundidade, ou em duas árvores diferentes

Classificação de arestas

- ♦ Na busca em profundidade cada aresta pode ser classificada pela cor do vértice que é alcançado pela primeira vez:
 - Branco indica uma aresta de árvore
 - Cinza indica uma aresta de retorno
 - Preto indica uma aresta de avanço quando u é descoberto antes de v ou uma aresta de cruzamento caso contrário

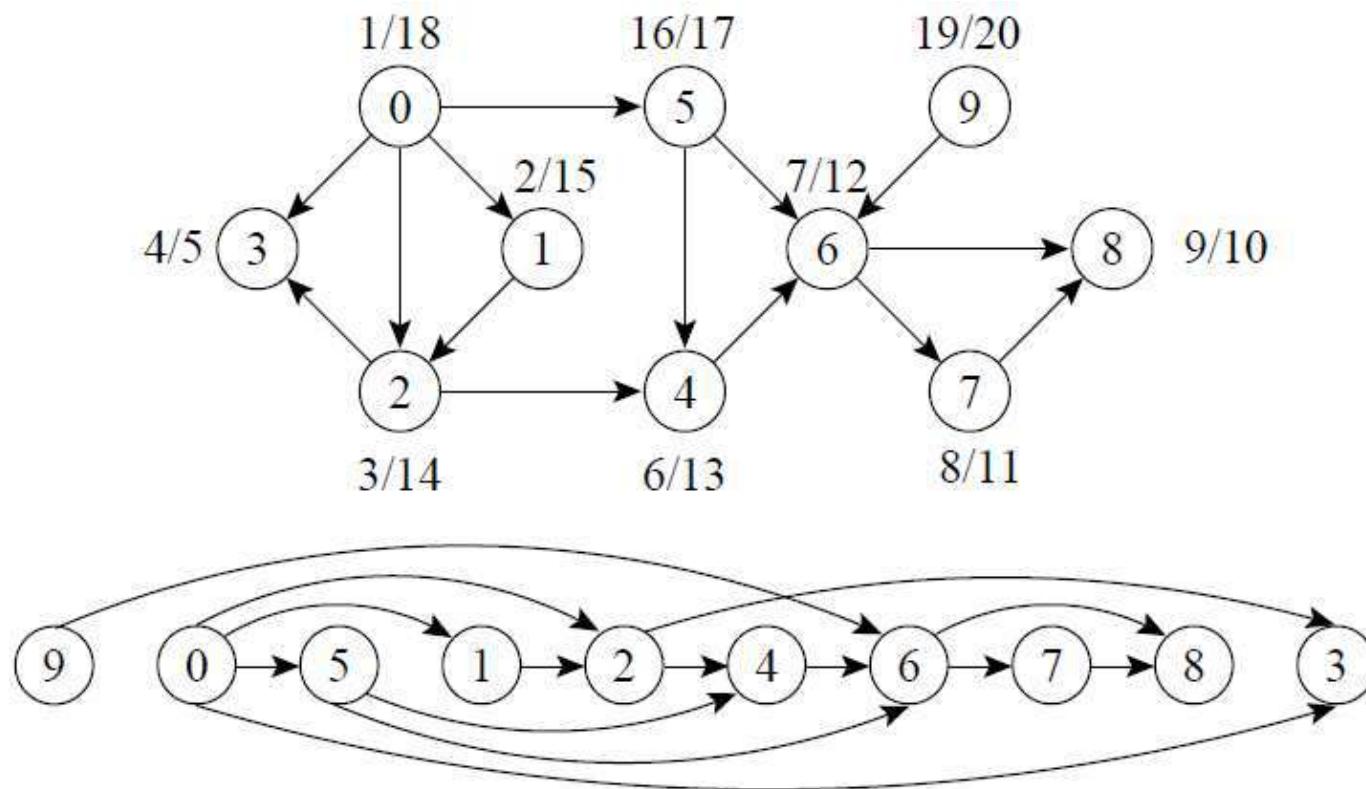


Ordenação topológica

- ◆ Ordenação linear de todos os vértices, tal que se G contém uma aresta (u, v) então u aparece antes de v
- ◆ Pode ser vista como uma ordenação de seus vértices ao longo de uma linha horizontal de tal forma que todas as arestas estão direcionadas da esquerda para a direita
- ◆ Pode ser feita usando a **busca em profundidade**

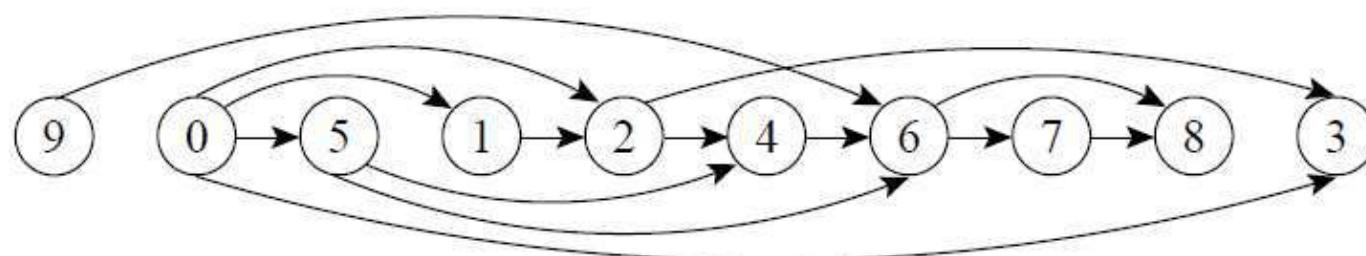
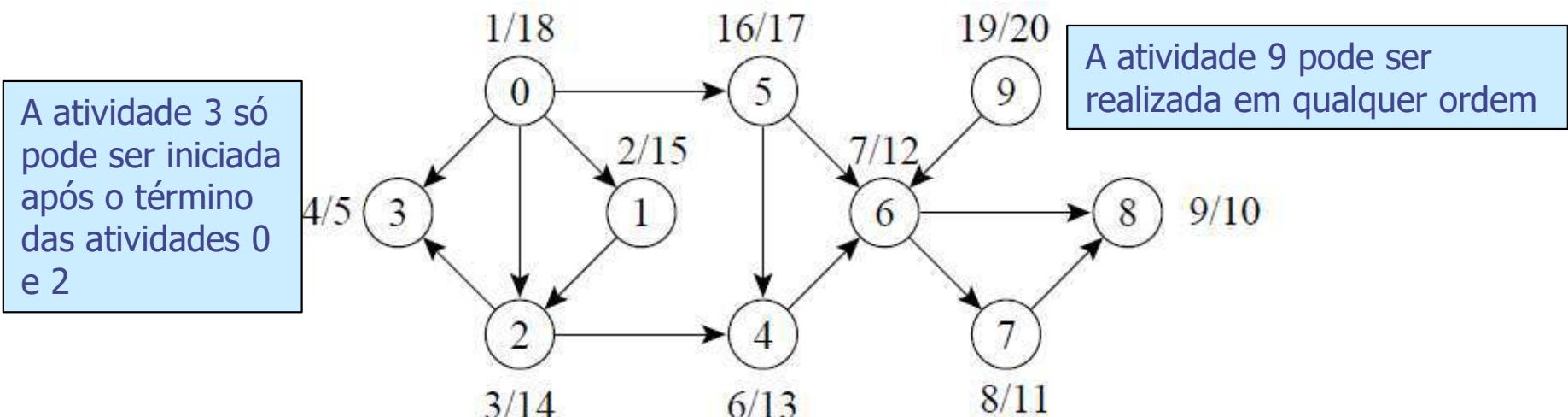
Ordenação topológica

- ◆ Os grafos direcionados acíclicos são usados para indicar precedências entre eventos
- ◆ Uma aresta direcionada (u, v) indica que a atividade u tem que ser realizada antes da atividade v



Ordenação topológica

- ◆ Os grafos direcionados acíclicos são usados para indicar precedências entre eventos
- ◆ Uma aresta direcionada (u, v) indica que a atividade u tem que ser realizada antes da atividade v



Ordenação topológica

- ◆ Algoritmo para ordenar topologicamente um grafo direcionado acíclico $G = (V, A)$:
 1. Chama *BuscaEmProfundidade*(G) para obter os tempos de término $t[u]$ para cada vértice u
 2. Ao término de cada vértice insira-o na frente de uma lista linear encadeada
 3. Retorna a lista encadeada de vértices
- ◆ A Custo $O(|V| + |A|)$, uma vez que a busca em profundidade tem complexidade de tempo $O(|V| + |A|)$ e o custo para inserir cada um dos $|V|$ vértices na frente da lista linear encadeada custa $O(1)$

Ordenação topológica

Implementação

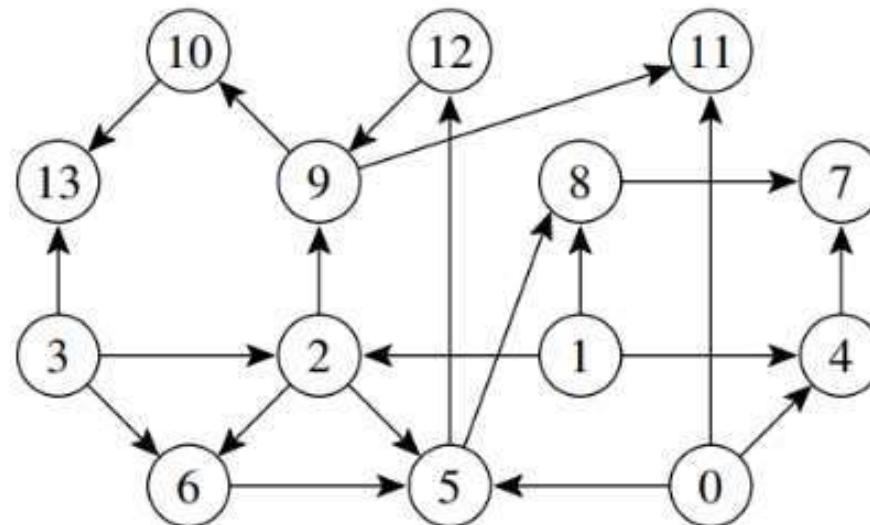
- ◆ Basta inserir uma chamada para o procedimento *InsLista* no procedimento *BuscaDfs*, logo após o momento em que o tempo de término $t[u]$ é obtido e o vértice é pintado de preto
- ◆ Ao final, basta retornar a lista obtida (ou imprimi-la usando o procedimento **Imprime** do Programa 3.4)

```
void InsLista (TipoItem *Item, TipoLista *Lista)
{ /*--Insere antes do primeiro item da lista --*/
    TipoApontador Aux;
    Aux = Lista->Primeiro->Prox;
    Lista->Primeiro->Prox = (TipoApontador)malloc(sizeof(
        tipoCelula));
    Lista->Primeiro->Prox->Item = Item;
    Lista->Primeiro->Prox->Prox = Aux;
}
```

Ordenação topológica

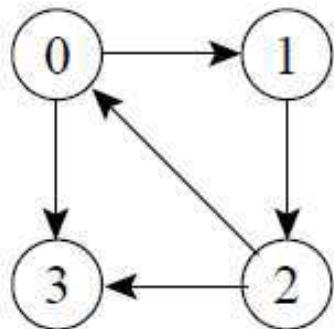
Exercício

- ♦ Mostre a ordem topológica quando o algoritmo executa sobre o grafo direcionado acíclico abaixo.



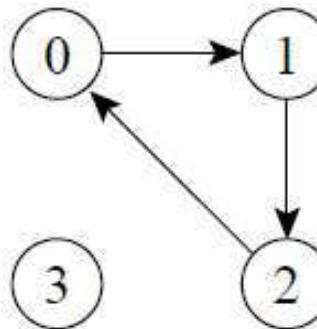
Componentes fortemente conectados

- ◆ Um componente fortemente conectado de $G = (V, A)$ é um conjunto maximal de vértices $C \subseteq V$ tal que para todo par de vértices u e v em C , u e v são mutuamente alcançáveis
- ◆ Podemos partitionar V em conjuntos V_i , $1 \leq i \leq r$, tal que vértices u e v são equivalentes se e somente se existe um caminho de u a v e um caminho de v a u



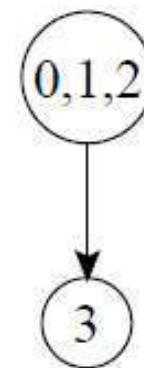
(a)

Grafo direcionado



(b)

Componentes
fortemente
conectados



(c)

Grafo
reduzido
acíclico

Componentes fortemente conectados

Algoritmo

- ◆ Usa o **transposto** de G , definido $G^T = (V, A^T)$, onde $A^T = \{(u, v) : (v, u) \in A\}$, isto é, A^T consiste das arestas de G com suas direções invertidas
- ◆ G e G^T possuem os mesmos componentes fortemente conectados, isto é, u e v são mutuamente alcançáveis a partir de cada um em G se e somente se u e v são mutuamente alcançáveis a partir de cada um em G^T

Componentes fortemente conectados

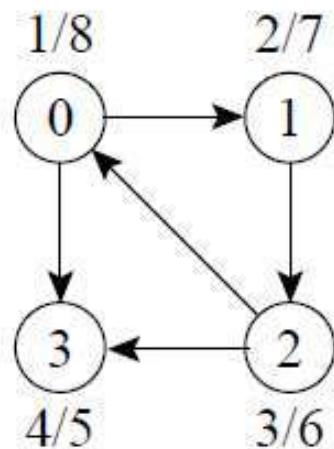
Algoritmo

1. Chama $\text{BuscaEmProfundidade}(G)$ para obter os tempos de término $t[u]$ para cada vértice u
2. Obtem G^T
3. Chama $\text{BuscaEmProfundidade}(G^T)$, realizando **a busca a partir do vértice de maior $t[u]$** obtido na linha 1. Inicie uma nova busca em profundidade a partir do vértice de maior $t[u]$ entre os vértices restantes se houver
4. Retorne os vértices de cada árvore da floresta obtida como um componente fortemente conectado separado

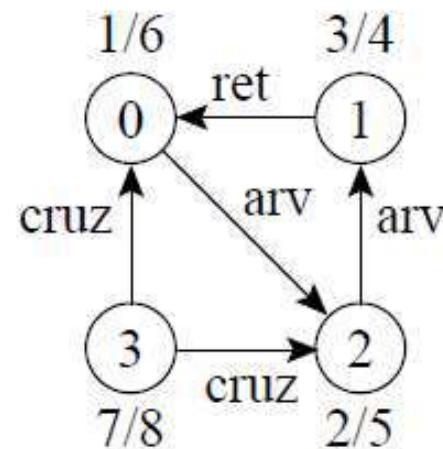
Componentes fortemente conectados

Exemplo

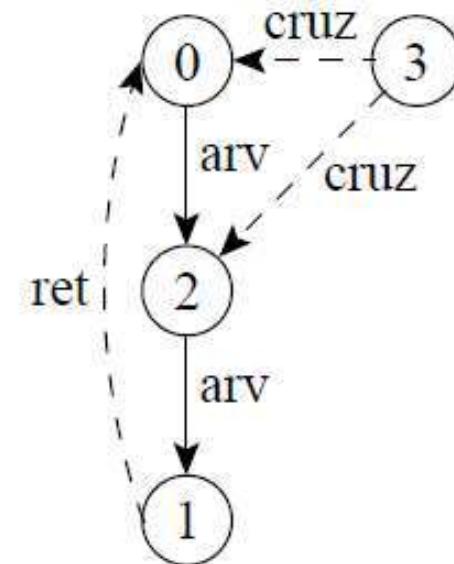
- ◆ A parte (b) apresenta o resultado da busca em profundidade sobre o grafo transposto obtido, mostrando os tempos de término e a classificação das arestas
- ◆ A busca em profundidade em G^T resulta na floresta de árvores mostrada na parte (c)



(a)



(b)



(c)

Componentes fortemente conectados

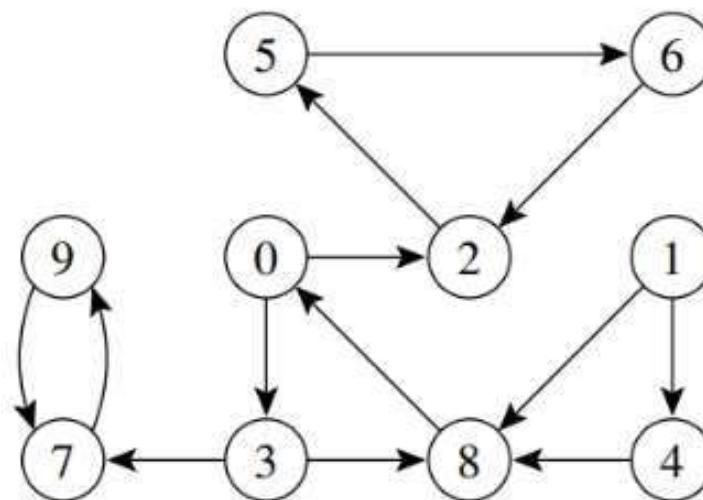
Análise

- ◆ Utiliza o algoritmo para busca em profundidade duas vezes, uma em G e outra em G^T
- ◆ Logo, a complexidade total é $O(|V| + |A|)$

Componentes fortemente conectados

Exercício

- ♦ Mostre como o procedimento para obter os componentes fortemente conectados funciona para o grafo abaixo. Assuma que os vértices são processados em ordem alfabética e que as listas de adjacência também estão em ordem alfabética.



Bibliografia



Ziviani, N. Projeto de Algoritmos, Thomson Learning, 2005.

Ebook da edição de 2018 disponível na biblioteca

<https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788522126590>

Bacharelado em Ciência da Computação
GBC034 Algoritmos e Estruturas de Dados 2

Métodos de Pesquisa em memória primária

Profa. Maria Camila Nardini Barioni

camila.barioni@ufu.br

Bloco B - sala 1B137

2º semestre de 2023

Roteiro das próximas aulas

◆ Hoje

- Motivação
- Conceitos Básicos
- Pesquisa sequencial
- Pesquisa sequencial ordenada
- Pesquisa binária

◆ Sexta-feira 15/03

- Aula prática no laboratório

Motivação

- ◆ A **operação de busca (pesquisa)** é algo rotineiro em computação
 - Ato de procurar por um elemento em um conjunto de dados armazenado em um repositório ou base de dados
- ◆ Nesta aula examinaremos métodos para pesquisar grandes quantidades de dados para encontrar determinada informação
- ◆ Veremos que certos métodos de organizar dados tornam o processo de busca mais eficiente

Conceitos Básicos

- ◆ A informação é dividida em **registros**
- ◆ Cada registro possui uma **chave** para ser usada na pesquisa
 - ◆ Valor armazenado em um vetor
 - ◆ Campo de uma struct
 - ◆ etc

The screenshot shows a web browser displaying the UFU Library Catalog at <https://acervo.bibliotecas.ufu.br>. The search term 'nivio ziviani' was entered in the search bar. The results page displays one item:

Material	Livros Impressos
Nº de chamada	661.3.06 Z82p 3.ed.
Ent. princ.	Ziviani, Nívio, 1946-
Título	Projeto de algoritmos : com implementações em Pascal e C / 3. ed., rev. e ampl.
Ano	c2011
Assuntos	Estruturas de dados (Computação)
Acervo	BCMON - SANTA MÔNICA - 20 exemplares BSMTC - MONTE CARMELO - 12 exemplares

On the left, there is a sidebar with filters for Material (Livros Impressos, eBooks Assinatura), Edição (2. ed., rev. e ampl, 3 (1), 3. ed., rev. e ampl), and Ano (2004, 2007, 2012, 2018). On the right, there are buttons for Selecionar, Detalhes, Exemplares, Reservar, and Referência.

Conceitos Básicos

- ◆ A informação é dividida em **registros**
- ◆ Cada registro possui uma **chave** para ser usada na pesquisa

◆ **Objetivo da pesquisa:**

- Encontrar uma ou mais ocorrências de registros com chaves iguais à **chave de pesquisa**
- **Pesquisa com sucesso X Pesquisa sem sucesso**

The screenshot shows a library catalog interface with a search result for the book "Projeto de Algoritmos".

Search Results:

- Educação:** 2. ed., rev. e ampl (1), 3 (1), 3. ed., rev. e ampl (1)
- Ano:** 2004 (1), 2007 (1), 2012 (1), 2018 (1)
- Material:** Livros Impressos
- Nº de chamada:** 661.3.06 Z82p 3.ed.
- Ent. princ.:** Ziviani, Nívio, 1946-
- Título:** Projeto de algoritmos : com implementações em Pascal e C / 3. ed., rev. e ampl
- Ano:** c2011
- Assuntos:** Estruturas de dados (Computação)
- Acervo:** BCOMN - SANTA MÔNICA - 20 exemplares
BSMTC - MONTE CARMELO - 12 exemplares

Rating: ★★★★☆ Seja o primeiro a avaliar [Tweet](#)

Actions: Selecionar, Detalhes, Exemplares, Reservar, Referência

Conceitos Básicos

◆ Conjunto de registros → tabela ou arquivo

◆ **Tabela:**

- associada a entidades de vida curta, criadas na memória interna durante a execução de um programa

◆ **Arquivo:**

- geralmente associado a entidades de vida mais longa, armazenadas em memória externa

◆ **Distinção não é rígida:**

- **tabela:** arquivo de índices
- **arquivo:** tabela de valores de funções

Escolha do Método de Pesquisa mais Adequado

- ◆ **Depende principalmente:**
 1. Quantidade dos dados envolvidos
 2. Arquivo estar sujeito a inserções e retiradas frequentes
- ◆ *Se conteúdo do arquivo é estável é importante minimizar o tempo de pesquisa, sem preocupação com o tempo necessário para estruturar o arquivo*

Métodos de Pesquisa → TAD

- ◆ É importante considerar os algoritmos de pesquisa como **tipos abstratos de dados**, com um conjunto de operações associado a uma estrutura de dados, de tal forma que haja uma independência de implementação para as operações
- ◆ **Operações mais comuns:**
 1. Inicializar a estrutura de dados
 2. Pesquisar um ou mais registros com determinada chave
 3. Inserir um novo registro
 4. Retirar um registro específico
 5. Ordenar um arquivo para obter todos os registros em ordem de acordo com a chave
 6. Ajuntar dois arquivos para formar um arquivo maior

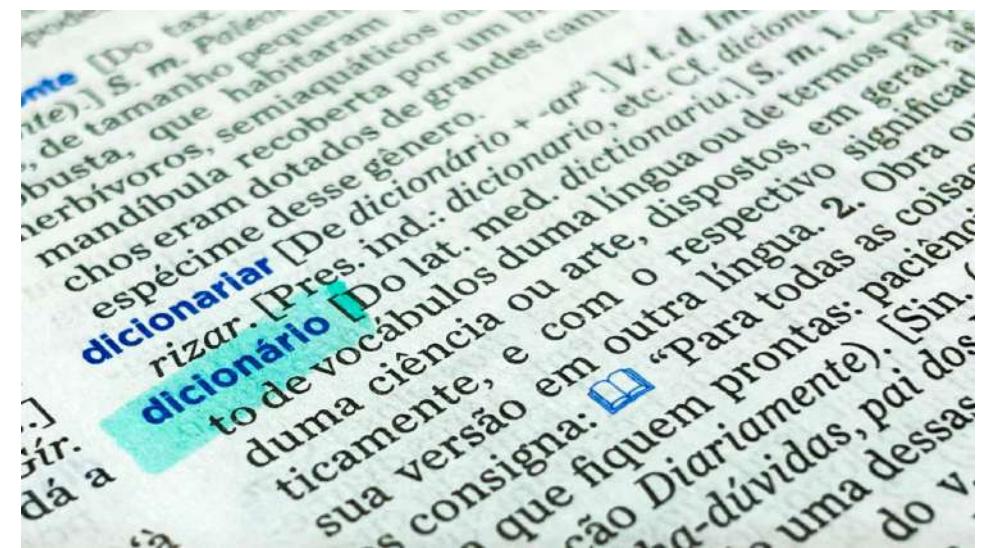
Dicionário

- ◆ Nome comumente utilizado para descrever uma estrutura de dados para pesquisa
- ◆ **Dicionário** é um **tipo abstrato de dados** com as operações:
 1. Inicializa
 2. Pesquisa
 3. Insere
 4. Retira

Dicionário

◆ Analogia com um dicionário da língua portuguesa:

- Chaves ↔ palavras
- Registros ↔ entradas associadas com cada palavra:
 - ◆ pronúncia
 - ◆ definição
 - ◆ sinônimos
 - ◆ outras informações



Métodos de Pesquisa

- ◆ Existem vários tipos de busca
- ◆ Sua utilização depende de como são estes dados
 - Os dados estão estruturados ?
 - ◆ Vetor, lista, árvore, etc.?
 - ◆ Existe também a busca em dados não estruturados
 - Os dados estão ordenados?
 - Existem valores duplicados?

Métodos de Pesquisa

◆ **Pesquisa sequencial (ou linear)**

- Aplica-se tanto a dados ordenados quanto a dados não ordenados

◆ **Pesquisa binária**

- Aplica-se somente a dados ordenados
 - ◆ Vetores, Árvores
- Problema: a ordenação freqüentemente leva muito mais tempo do que a busca

◆ **Pesquisa direta com hashing**

- Acesso à chave em tempo $O(1)$

Métodos de Pesquisa

◆ Tipos de busca abordados

- Dados armazenados em um vetor
- Dados ordenados ou não

◆ Métodos

- Pesquisa Sequencial ou Linear
- Pesquisa Sequencial Ordenada
- Pesquisa Binária

Pesquisa Sequencial

- ◆ **Método de pesquisa mais simples** que usa um laço (loop) para buscar dados em um vetor ou lista
- ◆ O algoritmo percorre o vetor que contém os dados desde a sua primeira posição até a última
 - Assume que os dados não estão ordenados, por isso a necessidade de percorrer o vetor do seu início até o seu fim
- ◆ É um método **muito custoso**, porque no pior caso todos os elementos devem ser pesquisados

Pesquisa Sequencial

◆ Funcionamento

- Para cada posição do vetor, o algoritmo compara se a posição atual do vetor é igual a uma **chave** de busca (elemento a ser procurado)
 - ◆ Se os valores forem iguais, a busca termina
 - ◆ Caso contrário, a busca continua com a próxima posição do vetor

◆ Exemplo: Buscar o elemento 8 no vetor $V = \{5, 3, 1, 8, 2\}$

chave = 8

i=0	5	3	1	8	2	v[0] ≠ chave, continuar
i=1	5	3	1	8	2	v[1] ≠ chave, continuar
i=2	5	3	1	8	2	v[2] ≠ chave, continuar
i=3	5	3	1	8	2	v[3] = chave, parar

Pesquisa Sequencial em vetor

Algoritmo

```
1 int busca_sequencial(int v[], int n, int chave_busca){  
2     int i;  
3     for(i = 0; i<n; i++){  
4         if( v[i] == chave_busca) {  
5             return i;  
6         }  
7     }  
8     return -1;  
9 }
```

Pesquisa Sequencial em lista

Algoritmo

```
1 t_elem *pesquisaLista (t_elem *prim, int chave_busca) {  
2     t_elem *aux;  
3     for (aux = prim; aux != NULL; aux = aux->prox) {  
4         if(aux->campo == chave_busca) {  
5             return aux;  
6         }  
7     }  
8     return NULL;  
9 }
```

Pesquisa Sequencial

◆ Porque é custoso?

- **Pior caso:** Compara a chave com todos os elementos, requerendo n comparações
 - ◆ Complexidade: $T(n) = n \rightarrow O(n)$
- **Melhor caso:** encontra logo na primeira comparação
 - ◆ Complexidade: $T(n) = 1 \rightarrow O(1)$
- **Caso médio:** na média, encontra com $n/2$ comparações
 - ◆ Complexidade: $T(n) = n/2 \rightarrow O(n)$

Pesquisa Sequencial Ordenada

- ◆ Em diversas aplicações reais, precisamos de algoritmos de busca mais eficientes
- ◆ Como melhorar a eficiência do algoritmo de busca mostrado?
- ◆ Se os elementos estiverem armazenados em uma ordem aleatória no vetor, não é possível melhorar o algoritmo de busca, pois é necessário verificar todos os elementos
- ◆ Se os **elementos estiverem armazenados em ordem crescente**, pode-se concluir que um elemento não está presente no vetor assim que acharmos um elemento maior: se o elemento que buscamos estivesse presente ele precederia um elemento maior na ordem do vetor

Pesquisa Sequencial Ordenada

◆ Funcionamento

- Assume que os dados estão ordenados
- Se a **chave** de busca (elemento procurado) for menor do que o valor em uma determinada posição do vetor, temos a certeza de que ela não estará no restante do vetor
- Isso evita a necessidade de percorrer o vetor do seu início até o seu fim

◆ Exemplo: Buscar o elemento 4 no vetor $V = \{1, 3, 5, 7, 9\}$

chave = 4

i=0	1	3	5	7	9	v[0] < chave, continuar
i=1	1	3	5	7	9	v[1] < chave, continuar
i=2	1	3	5	7	9	v[2] > chave, parar (chave não encontrada)

Pesquisa Sequencial Ordenada em vetor - Algoritmo

```
1 int busca_sequencial_ordenada(int v[], int n, int chave_busca) {  
2     int i;  
3     for(i = 0; i<n; i++) {  
4         if(v[i] == chave_busca) {  
5             return i; // chave_busca encontrada  
6         }  
7         else if(v[i] > chave_busca) {  
8             return -1; // interrompe busca  
9         }  
10    }  
11    return -1; // chave_busca não encontrada  
12}
```

Pesquisa Sequencial Ordenada

◆ Análise

- **Pior caso:** a chave não foi encontrada, é maior que o último elemento do vetor
 - ◆ Complexidade: $T(n) = n \rightarrow O(n)$
- **Melhor caso:** encontra logo na primeira comparação
 - ◆ Complexidade: $T(n) = 1 \rightarrow O(1)$
- No caso da chave procurada não pertencer ao vetor, esse segundo algoritmo apresenta um desempenho ligeiramente superior ao primeiro, mas a ordem dessa versão do algoritmo continua sendo linear

Pesquisa Sequencial Ordenada

◆ Desvantagens

- Ordenar um vetor também tem um custo
 - ◆ Esse custo é superior ao custo da busca sequencial no seu pior caso
- Se for para fazer a busca de um único elemento, não compensa ordenar o vetor
 - ◆ Porém, se mais de um elemento for recuperado do vetor, o esforço de ordenar o vetor pode compensar

Pesquisa Binária (com dados ordenados)

- ◆ Se os elementos do vetor estiverem ordenados, podemos aplicar um algoritmo mais eficiente para realizar a busca
- ◆ A **Busca Sequencial Ordenada** é uma estratégia de busca extremamente simples
 - ◆ Ela percorre todo o vetor linearmente
 - ◆ Não utiliza adequadamente a ordenação dos dados
- ◆ Uma estratégia de busca mais sofisticada é a **Busca Binária**
 - ◆ Muito mais eficiente do que a Busca Sequencial Ordenada

Pesquisa Binária (com dados ordenados)



- ◆ É uma estratégia baseada na **ideia de dividir para conquistar**
- ◆ Tem um mecanismo semelhante à procurar uma palavra no dicionário. Deve-se abrir o dicionário ao meio e:
 - comparar a palavra procurada com a que está na página do meio e então:
 - se a palavra foi encontrada, encerra-se a busca
 - se a palavra procurada for menor (na ordem alfabética), repetir a busca na primeira metade do dicionário
 - se a palavra procurada for maior (na ordem alfabética), repetir a busca na segunda metade do dicionário
 - este processo se repete enquanto a palavra não for encontrada e enquanto existirem trechos do dicionário a serem consultados

Pesquisa Binária (com dados ordenados)



- ◆ É uma estratégia baseada na **ideia de dividir para conquistar**
- ◆ Tem um mecanismo semelhante à procurar uma palavra no dicionário. Deve-se abrir o dicionário ao meio e:
 Ideia geral é ignorar a metade dos dados a cada teste
 Ou seja, dividir o tamanho do problema pela metade
 - se a palavra procurada for menor (na ordem alfabética), repetir a busca na primeira metade do dicionário
 - se a palavra procurada for maior (na ordem alfabética), repetir a busca na segunda metade do dicionário
- este processo se repete enquanto a palavra não for encontrada e enquanto existirem trechos do dicionário a serem consultados

Pesquisa Binária

◆ Funcionamento

- A cada passo, esse algoritmo analisa o valor do meio do vetor
 - se o elemento procurado for menor que o elemento do meio, se o elemento estiver presente no vetor, ele estará na primeira parte do vetor
 - se for maior, estará na segunda parte do vetor
 - se for igual, o elemento é encontrado no vetor
- O procedimento é repetido considerando apenas a parte que condiz com o elemento procurado: o elemento procurado é comparado com o elemento armazenado no meio dessa parte
- Esse procedimento é continuamente repetido, subdividindo a parte de interesse até encontrar o elemento ou chegar ao fim

Pesquisa Binária

◆ Detalhando o funcionamento

- Dado um vetor V e uma chave de busca
 - ◆ Iniciar limite inferior = 0 e limite superior = $n - 1$
 - ◆ Comparar a chave de busca com o elemento do meio do vetor
 - ◆ Se a chave de busca for maior, deslocar o limite inferior para meio + 1
 - ◆ Se a chave de busca for menor, deslocar o limite superior para meio - 1
 - ◆ Se a chave de busca for igual, encontrou a chave (retorna True ou a posição do elemento)
 - ◆ Repetir enquanto o limite inferior for menor que o limite superior
 - ◆ Ao final, se não encontrou, retorna False ou -1

Pesquisa Binária - exemplo

$$n = 8$$

chave = 86

Passo 1

The diagram illustrates a binary search operation on an array of 8 elements. The array is labeled "vetor" and contains the values 10, 35, 43, 51, 77, 86, 91, and 99. Above the array, indices 0 and n-1 are shown in red. Below the array, pointers "inf" (information), "meio" (middle), and "sup" (superior) are shown in green, indicating the search range.

0								n-1
vetor	10	35	43	51	77	86	91	99
inf								sup
meio								

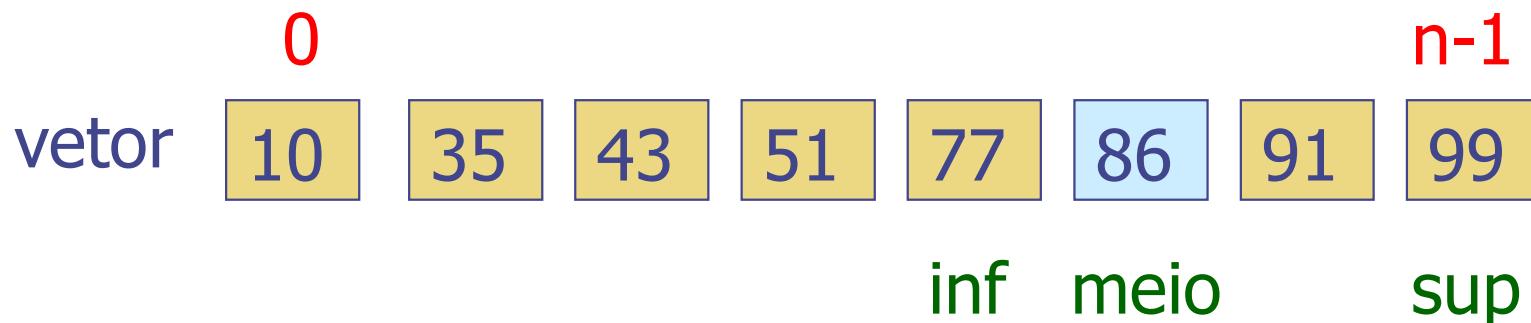
chave > vetor [meio]

Pesquisa Binária - ilustração

n = 8

chave = 86

Passo 2



Pesquisa Binária - Algoritmo

```
1 int busca_binaria(int v[], int n, int chave_busca) {  
2     int inf = 0, sup = n-1, meio;  
3     while(inf <= sup){  
4         meio = (inf+sup)/2;  
5         if (v[meio] == chave_busca)  
6             return meio;  
7         else if(v[meio] > chave_busca)  
8             sup = meio-1;  
9         else  
10            inf = meio+1;  
11    }  
12    return -1;  
13 }
```

Eficiência da Pesquisa Binária

- ◆ O desempenho desse algoritmo é muito superior ao da busca linear
- ◆ Novamente, o pior caso acontece quando o elemento procurado não está no vetor
- ◆ Quantas vezes precisamos repetir o procedimento de subdivisão para concluir que o elemento não está presente no vetor?
 - Inicialmente, todo o vetor é considerado; a cada repetição, a parte considerada na busca é dividida à metade. A tabela a seguir mostra o tamanho do vetor ao fim de cada repetição do laço do algoritmo

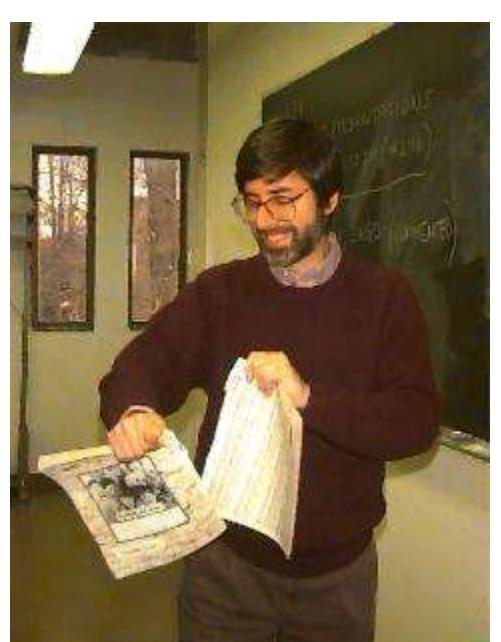
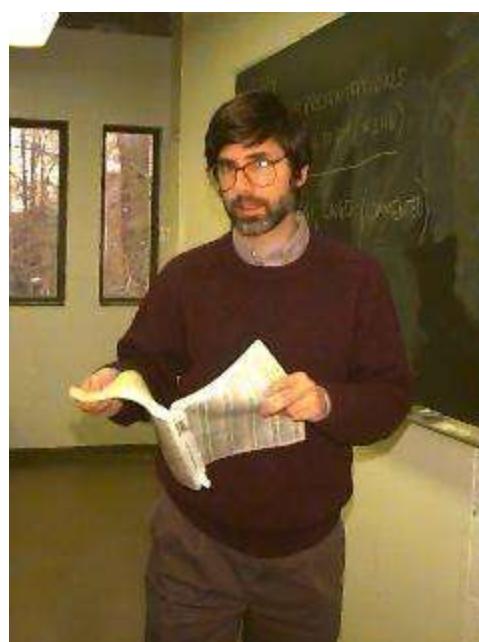
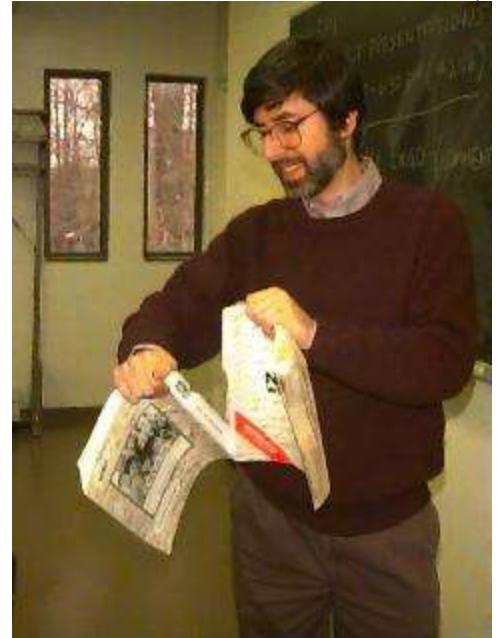
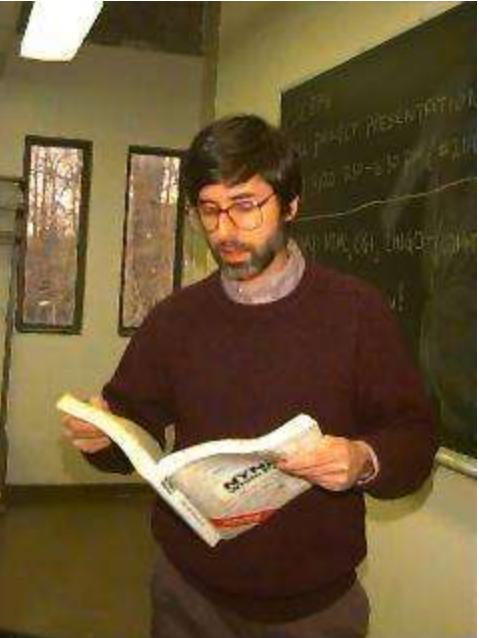
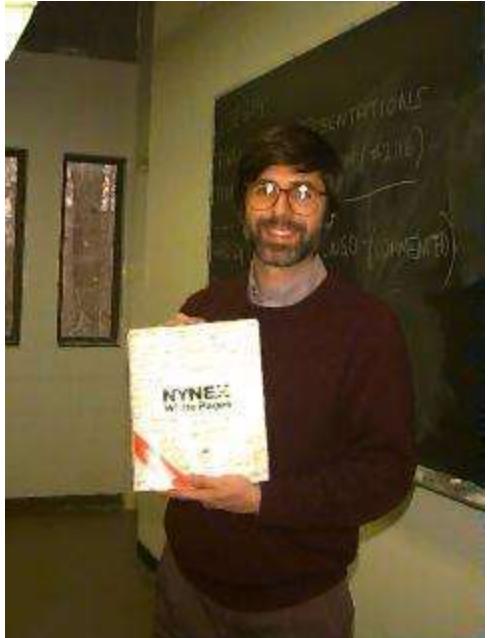
Repetição	Tamanho do problema
0	n
1	$n/2$
2	$n/4$
3	$n/8$
...	...
$\log n$	1

Eficiência da Pesquisa Binária

- ◆ Considerando um vetor com n elementos, o tempo de execução é:
 - **Pior Caso:** A chave não foi encontrada: $O(\log_2 n)$
 - **Caso Médio:** $O(\log_2 n)$
 - **Melhor Caso:** A chave está no meio do vetor: $O(1)$
- ◆ Para se ter uma ideia da sua vantagem, considere vetores contendo diferentes quantidades de elementos, no pior caso

Num elementos	Num comparações		Tempo em computador que faz 10^9 comparações por segundo	
	Sequencial	Busca bin	Sequencial	Busca bin
1.000	1.000	10	1 micro segundo	alguns nano segundos
1.000.000	1.000.000	20	1 mili segundo	alguns nano segundos
1.000.000.000	1.000.000.000	30	1 segundo	alguns nano segundos
1.000.000.000.000	1.000.000.000.000	40	1000 s = 16 minutos	alguns nano segundos

Pesquisa Binária - Animação



Busca em vetor de struct

◆ A busca em um vetor de inteiros é uma tarefa simples

- Na prática, trabalhamos com dados um pouco mais complexos, como estruturas
- Mais dados para manipular

```
11 struct aluno{  
12     int matricula;  
13     char nome[30];  
14     float n1, n2, n3;  
15 };
```

Busca em vetor de struct

- ◆ Como fazer a busca quando o que temos é um vetor de struct?

```
struct aluno v[6];
```

matricula; nome [30]; n1, n2, n3;					
---	---	---	---	---	---

v[0]

v[1]

v[2]

v[3]

v[4]

v[5]

Busca em vetor de struct

◆ Relembrando

◆ A busca é baseada em uma chave

- A chave de busca é o **campo** do item utilizado para comparação
 - ◆ Valor armazenado em um vetor de inteiros
 - ◆ **Campo de uma struct**
 - ◆ etc
- É por meio dela que sabemos se dado elemento é o que buscamos
 - ◆ No caso do item estar presente no conjunto de elementos, seus dados são retornados para o usuário

Busca em vetor de struct

- ◆ Ou seja, devemos modificar o algoritmo para que a comparação das chaves seja feita utilizando um determinado campo da **struct**
- ◆ Exemplo
 - Vamos modificar a **busca linear**
 - ◆ Essa modificação vale para os outros tipos de busca

```
13
14 int buscaLinear(int *v, int N, int elem) {
15     int i;
16     for(i = 0; i < N; i++) {
17         if(elem == v[i])
18             return i; //elemento encontrado
19     }
20     return -1; //elemento não encontrado
21 }
```

Busca em vetor de struct

◆ Duas novas buscas

- Busca por **matricula**
- Busca por **nome**

```
28 int buscaLinearMatricula(struct aluno *V, int N, int elem) {  
29     int i;  
30     for(i = 0; i < N; i++) {  
31         if(elem == V[i].matricula)  
32             return i; //elemento encontrado  
33     }  
34     return -1; //elemento não encontrado  
35 }  
36  
37  
38 int buscaLinearNome(struct aluno *V, int N, char* elem) {  
39     int i;  
40     for(i = 0; i < N; i++) {  
41         if(strcmp(elem, V[i].nome) == 0)  
42             return i; //elemento encontrado  
43     }  
44     return -1; //elemento não encontrado  
45 }
```

Bibliografia

- ◆ Ziviani, N. Projeto de Algoritmos, Thomson Learning, 2005.
- ◆ Tenenbaum, A.M.; Langsam, Y.; Augenstein, M.J. Estruturas de Dados usando C, Makron Books, 1995.
- ◆ Celes, W.; Cerqueira, R.; Rangel, J.L. Introdução a Estruturas de Dados com Técnicas de Programação em C. 2a. ed. Elsevier, 2016.
- ◆ Backes, A. Linguagem C Descomplicada – Estruturas de Dados. Material Complementar.
- ◆ Slides Profa. Maria Adriana Vidigal de Lima.

Material extra

- ◆ Backes, A. Programação Descomplicada
 - Estruturas de Dados.
 - Vídeo-aula 45: <http://youtu.be/ptvnLzqcJuA>
 - Vídeo-aula 46: <http://youtu.be/zxwCSxbntKA>

Agendamento Apresentações Projeto

- ◆ 07/04 – Entrega do projeto final
- ◆ Agendar as apresentações dos projetos até 19/03
 - O agendamento deve ser feito por meio de envio de mensagem privada no MS Teams informando dia/horário e os nomes de todos os integrantes do grupo

Agenda – Apresentações projetos

Data/Horário	Grupo
13/04 – 08:50h	
13/04 – 09:05h	
13/04 – 09:20h	
13/04 – 09:35h	
13/04 – 09:50h	
13/04 – 10:05h	
13/04 – 10:20h	
13/04 – 10:35h	

- Todas as apresentações serão feitas no Campus Santa Mônica Lab03
- Cada grupo deve chegar no laboratório 10 minutos antes do horário agendado para a apresentação para logar no computador e deixar tudo preparado para a sua apresentação.

Agenda – Apresentações projetos

Data/Horário	Grupo
13/04 – 10:50h	
13/04 – 11:05h	
13/04 – 11:20h	
13/04 – 11:35h	
13/04 – 11:50h	
13/04 – 12:05h	

- Todas as apresentações serão feitas no Campus Santa Mônica Lab03
- Cada grupo deve chegar no laboratório 10 minutos antes do horário agendado para a apresentação para logar no computador e deixar tudo preparado para a sua apresentação.

Bacharelado em Ciência da Computação
GBC034 Algoritmos e Estruturas de Dados 2

Árvore

Profa. Maria Camila Nardini Barioni

camila.barioni@ufu.br

Bloco B - sala 1B137

2º semestre de 2023

Motivação

- ◆ Diversas aplicações necessitam de estruturas mais complexas que as estudadas até agora
- ◆ Ex.: como visualizar o conjunto de diretórios apenas com listas, pilhas e filas?
 - Sistemas Operacionais (arquivos), Linguagens, etc.
- ◆ Árvores solucionam esse problema, e podem ser usadas para modelar diversos outros
- ◆ Também existem algoritmos eficientes para o tratamento de árvores
 - Algumas árvores podem definir uma ordenação implícita, o que facilita a execução do algoritmo, com um tratamento condicional simples

Conceito

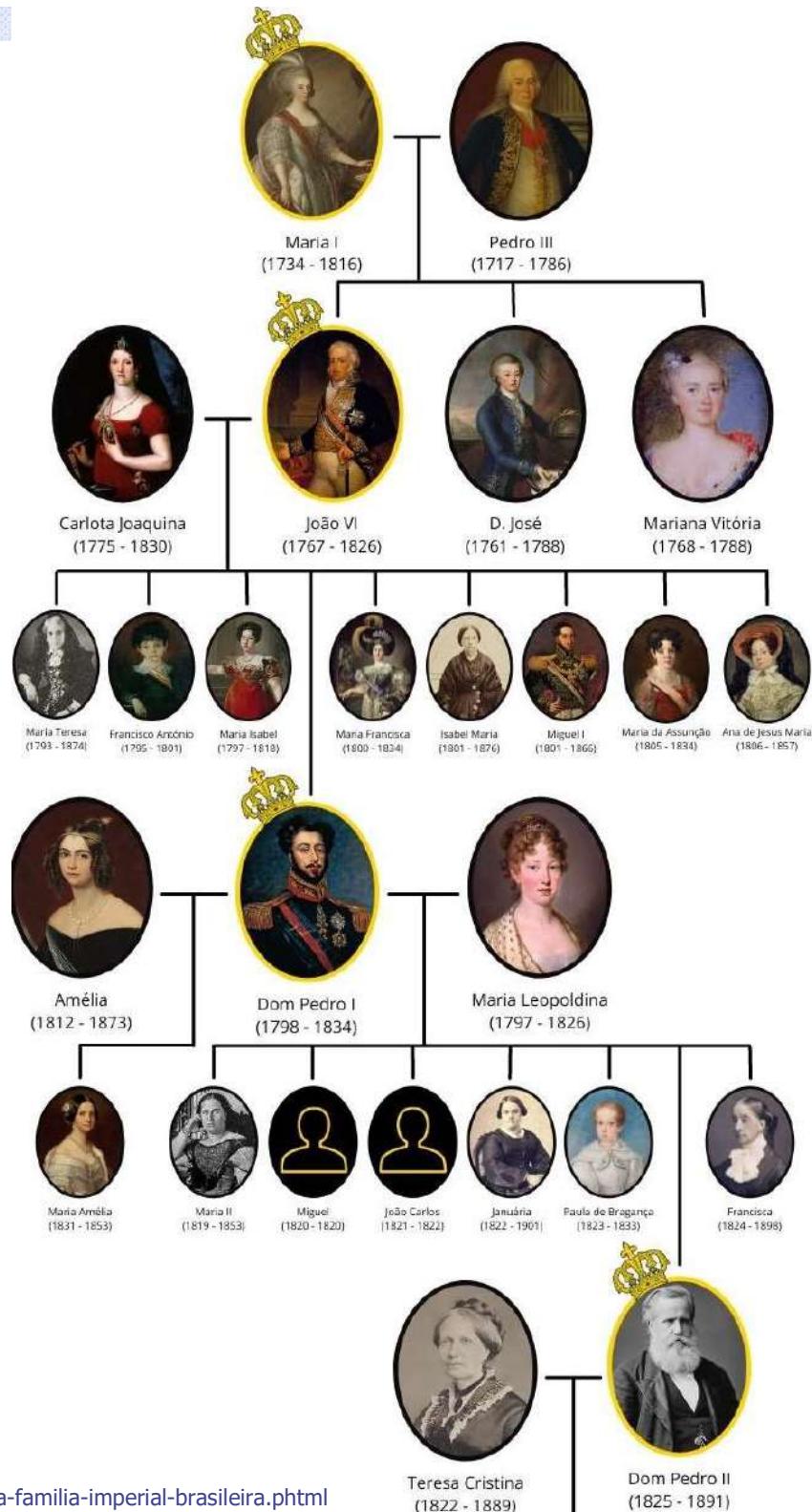
◆ As árvores são estruturas de dados capazes de representar o relacionamento hierárquico entre diversas informações

◆ Exemplos

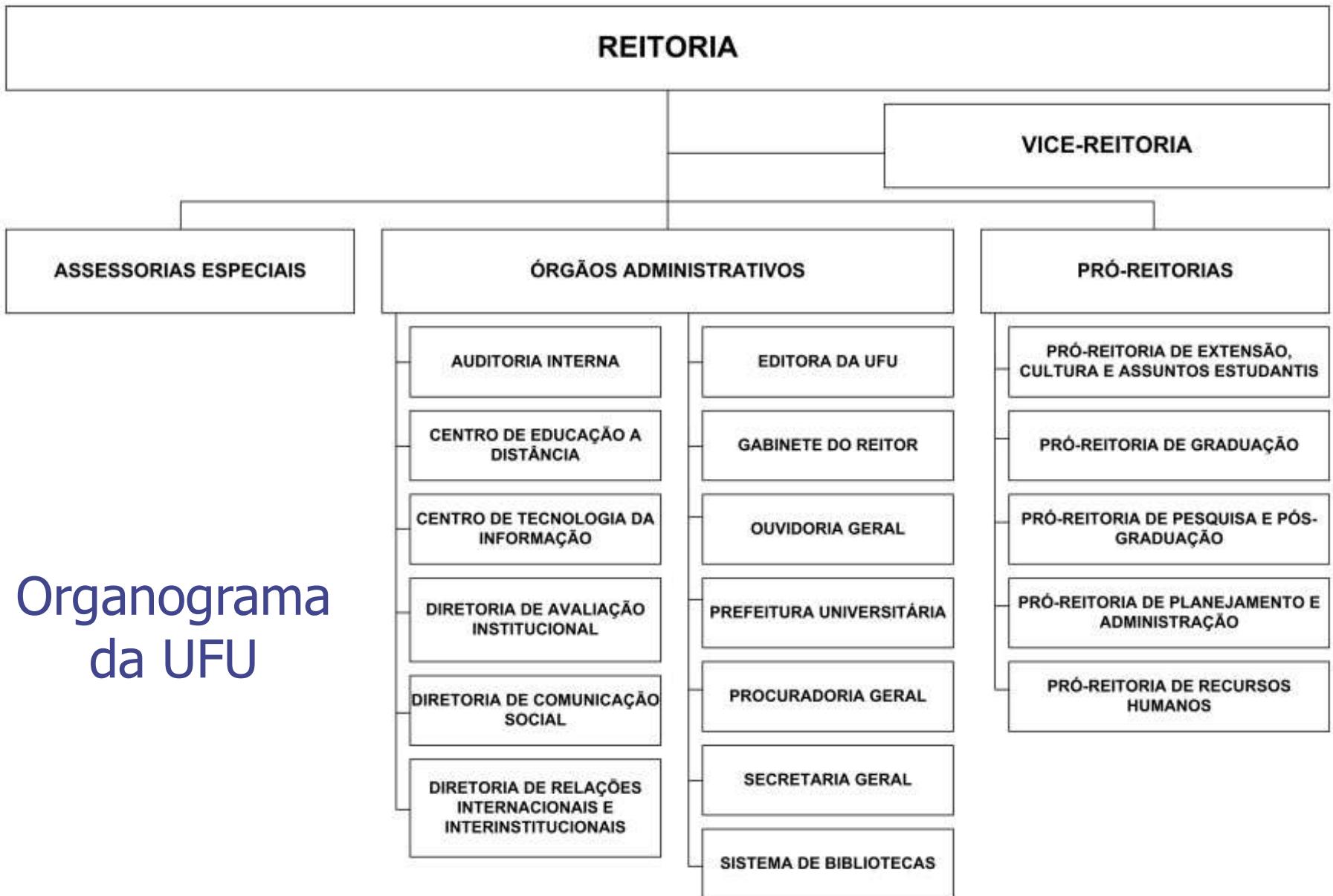
- Árvore genealógica de uma família
- Organograma de uma empresa
- Árvore estrutural de um povo

Exemplo

Árvore genealógica da Família Imperial Brasileira

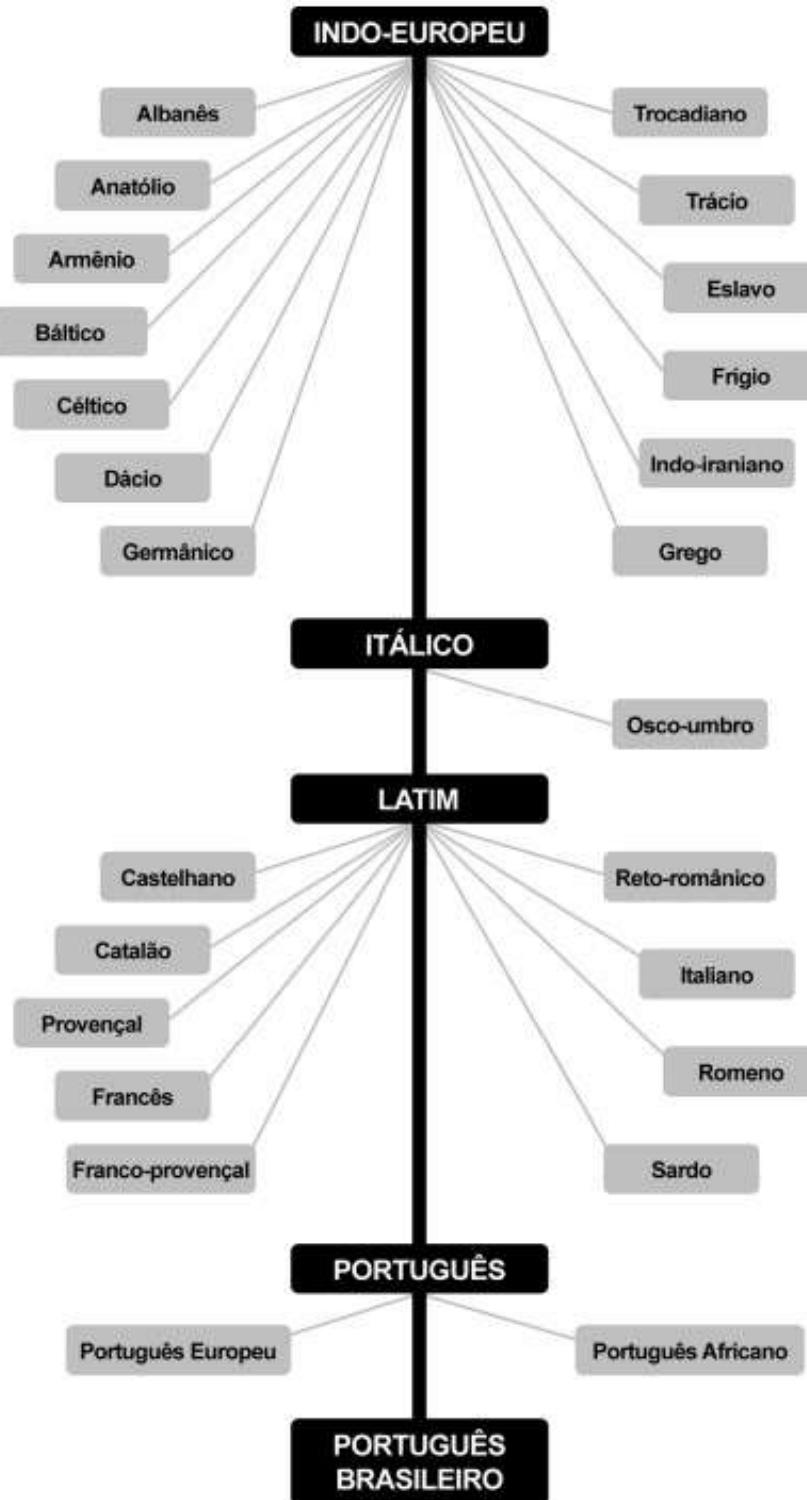


Exemplo



Exemplo

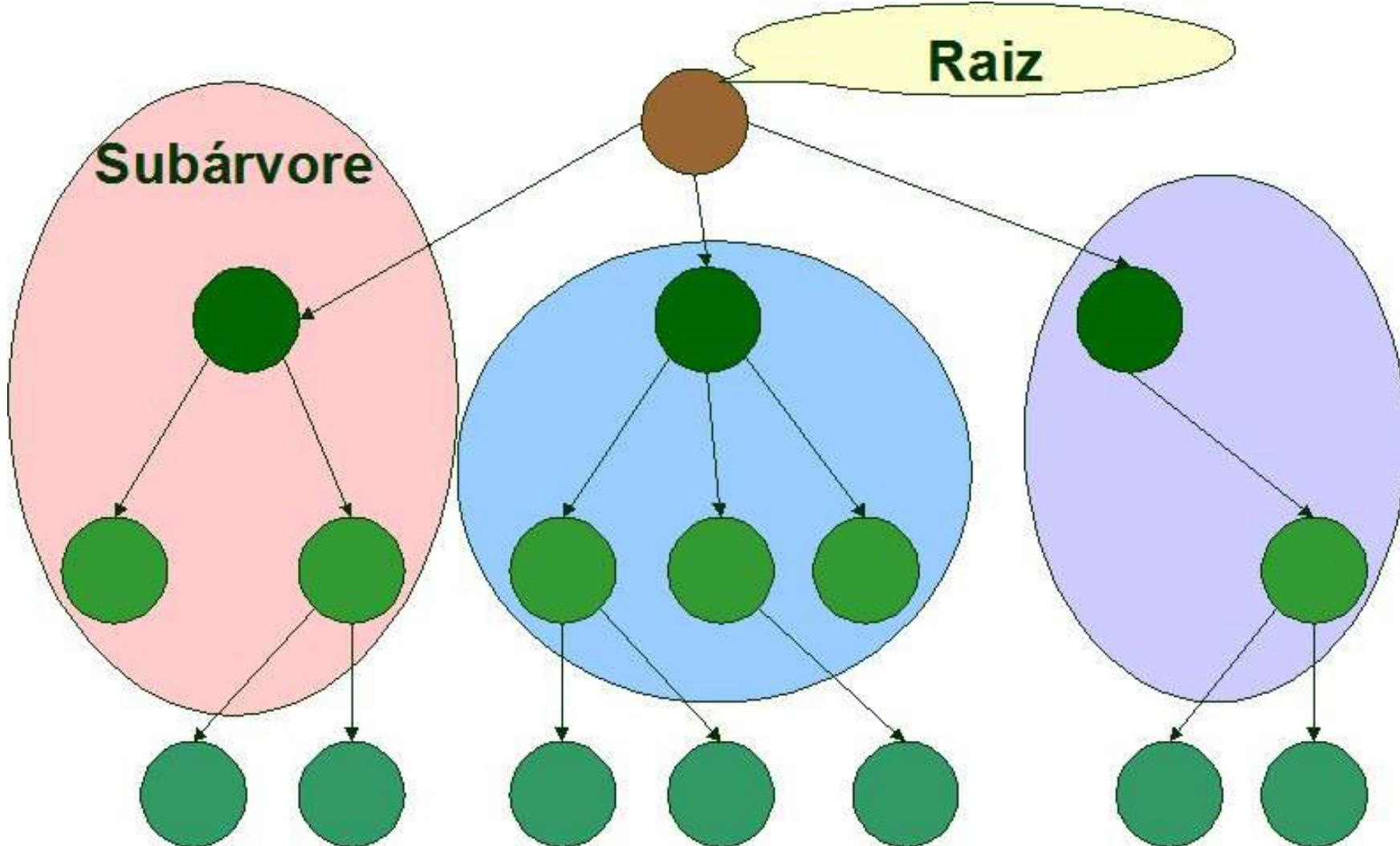
Origem da Língua Portuguesa



Definições

- ◆ Uma árvore T é um conjunto finito de elementos, denominados **nós** ou vértices, e um conjunto de linhas dirigidas, chamadas de **ramificações**, que conectam os nós, tais que:
 - $T = \emptyset$ é a árvore dita vazia ou
 - Existe um nó especialmente designado denominado raiz de T
 - Os nós restantes constituem um único conjunto vazio ou estão desdobrados em N conjuntos separados ($N \geq 1$) T_1, T_2, \dots, T_N , em que cada um dos conjuntos se constitui uma árvore (denominadas de subárvores da raiz)

Exemplo

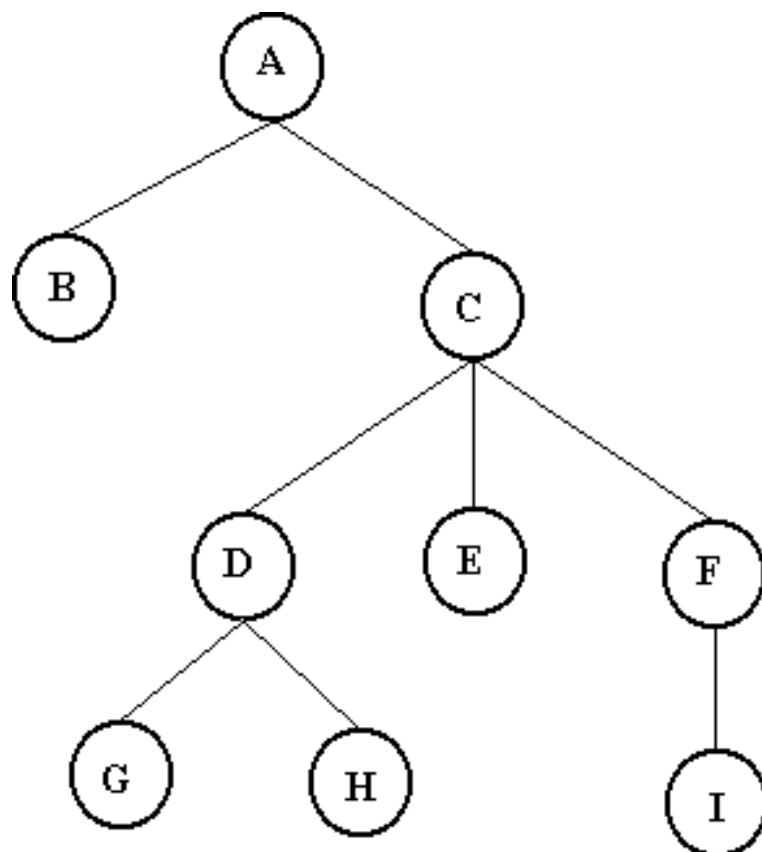


Definições: Sub-árvores

- ◆ **Notação:** Se v é um nó de T , então a notação T_v indica a subárvore de T com raiz em v
- ◆ Seja a árvore $T_A = \{A, B, C, D, E, F, G, H, I\}$
- ◆ A árvore T possui duas subárvores: T_B e T_C onde $T_B = \{B\}$ e $T_C = \{C, D, E, F, G, H, I\}$
- ◆ A subárvore T_C possui 3 subárvores: T_D , T_F e T_E onde $T_D = \{D, G, H\}$ $T_F = \{F, I\}$ $T_E = \{E\}$
- ◆ As subárvores T_B , T_E , T_G , T_H , T_I possuem apenas o nó raiz e nenhuma subárvore

Árvores: Representação

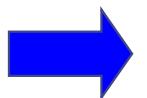
◆ Hierárquica



Alinhamento dos nós

A
B
C
D
E
F
I

G
H

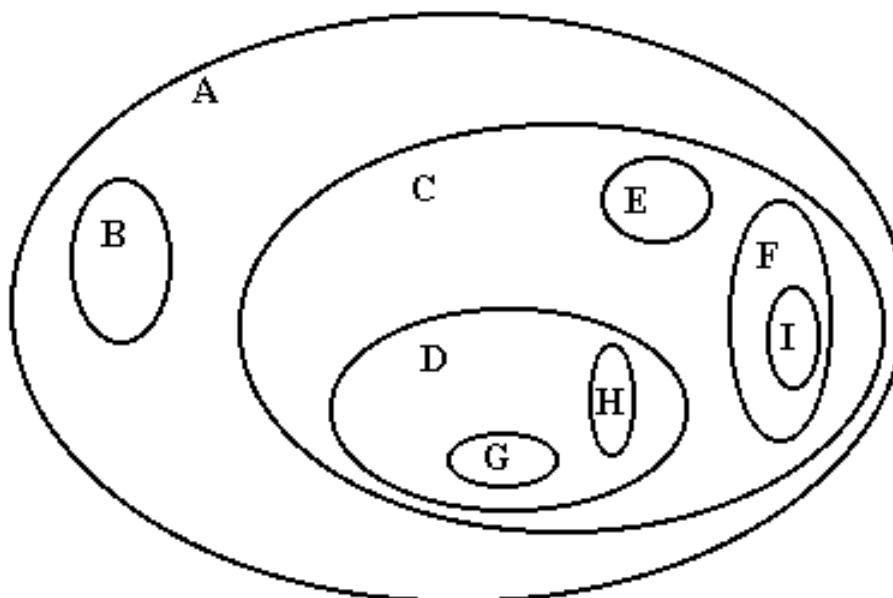


Árvores: Representação

- ◆ Parênteses aninhados

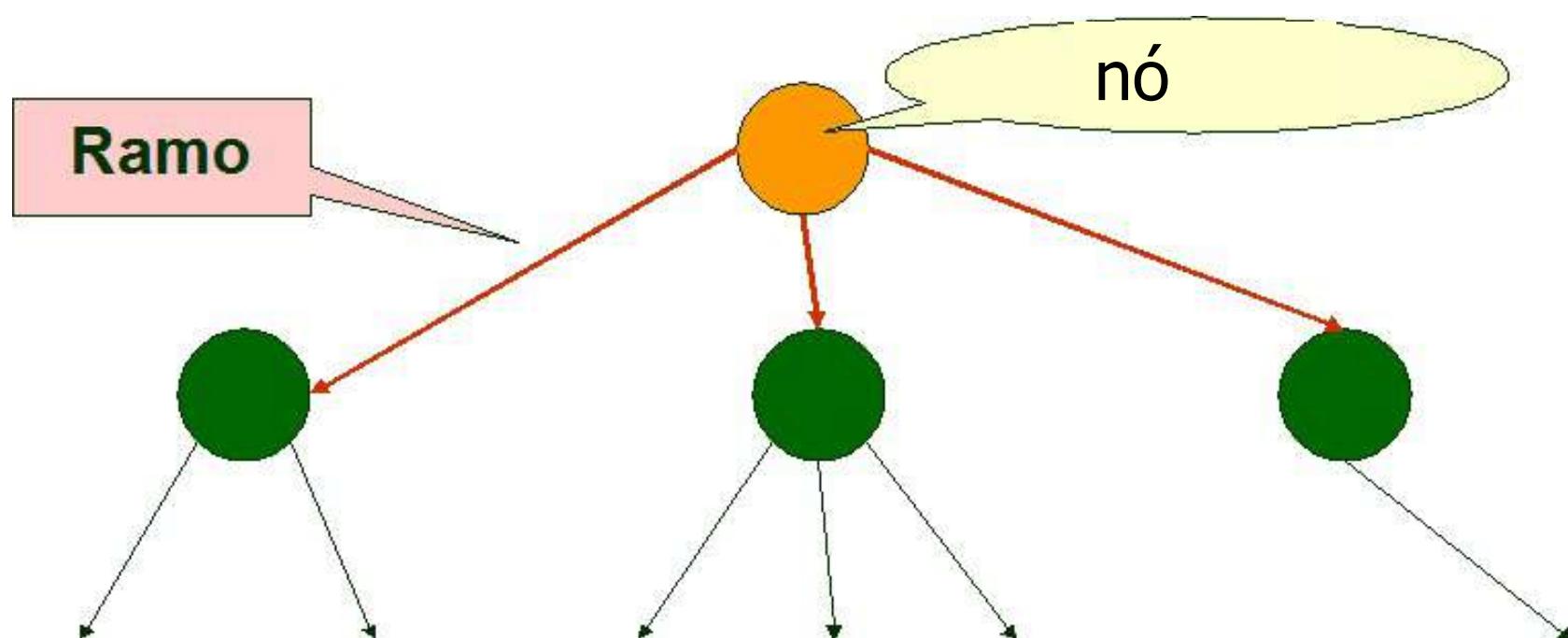
(A (B) (C (D (G) (H)) (E) (F (I))))

- ◆ Diagramas de inclusão



Nó

- ◆ Cada nó representa o fator da informação e mais os ramos que os ligam a outros nós (nós-filhos)



Genealogia das árvores

Seja v o nó raiz da subárvore T_v de T

◆ Nós filhos

- os nós $w_1, w_2, \dots w_j$, raízes das subárvores de T_v , são chamados filhos de v

◆ Pais, tios, irmãos e avô

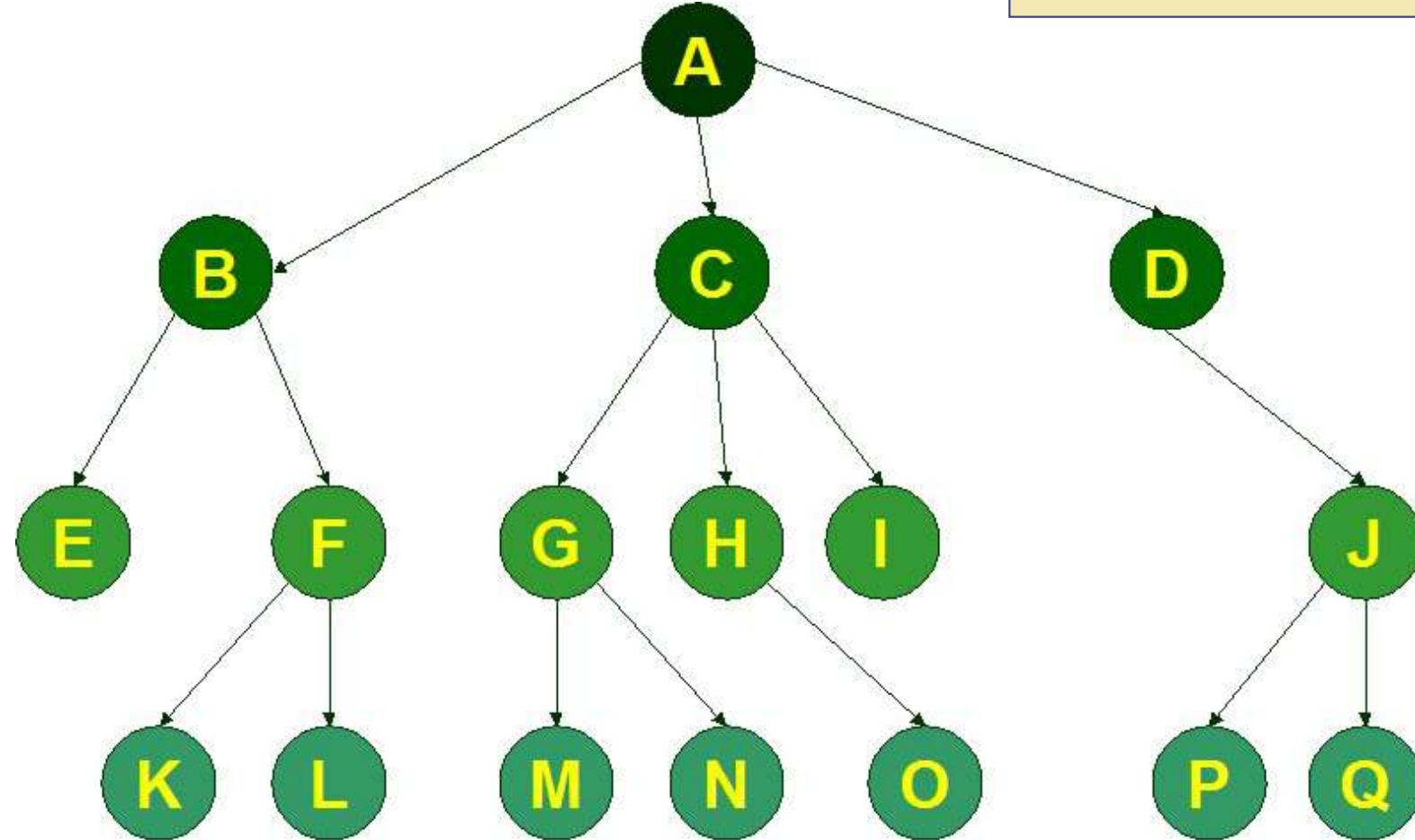
- O nó v é chamado pai de $w_1, w_2, \dots w_j$. Os nós $w_1, w_2, \dots w_j$ são irmãos. Se z é filho de w_1 então w_2 é tio de z e v é avô de z

◆ Nό descendente e ancestral

- Se x pertence à sub-árvore T_v , então x é descendente de v e v é ancestral, ou antecessor, de x

Genealogia das árvores

Nó Pai



Pai

A é de B, C e D

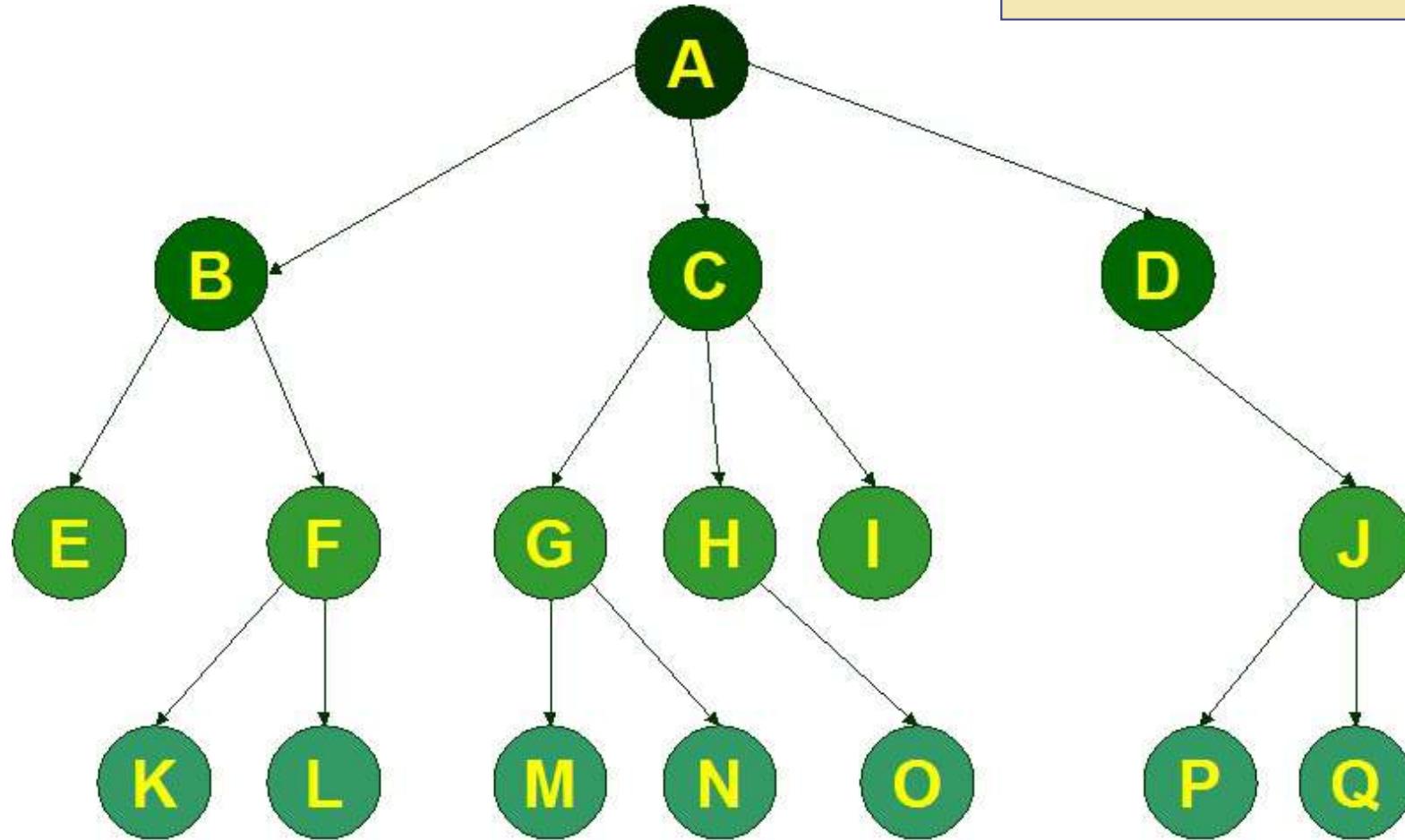
B é de E e F

C é de G, H e I

D é de J

Genealogia das árvores

Nó Filho



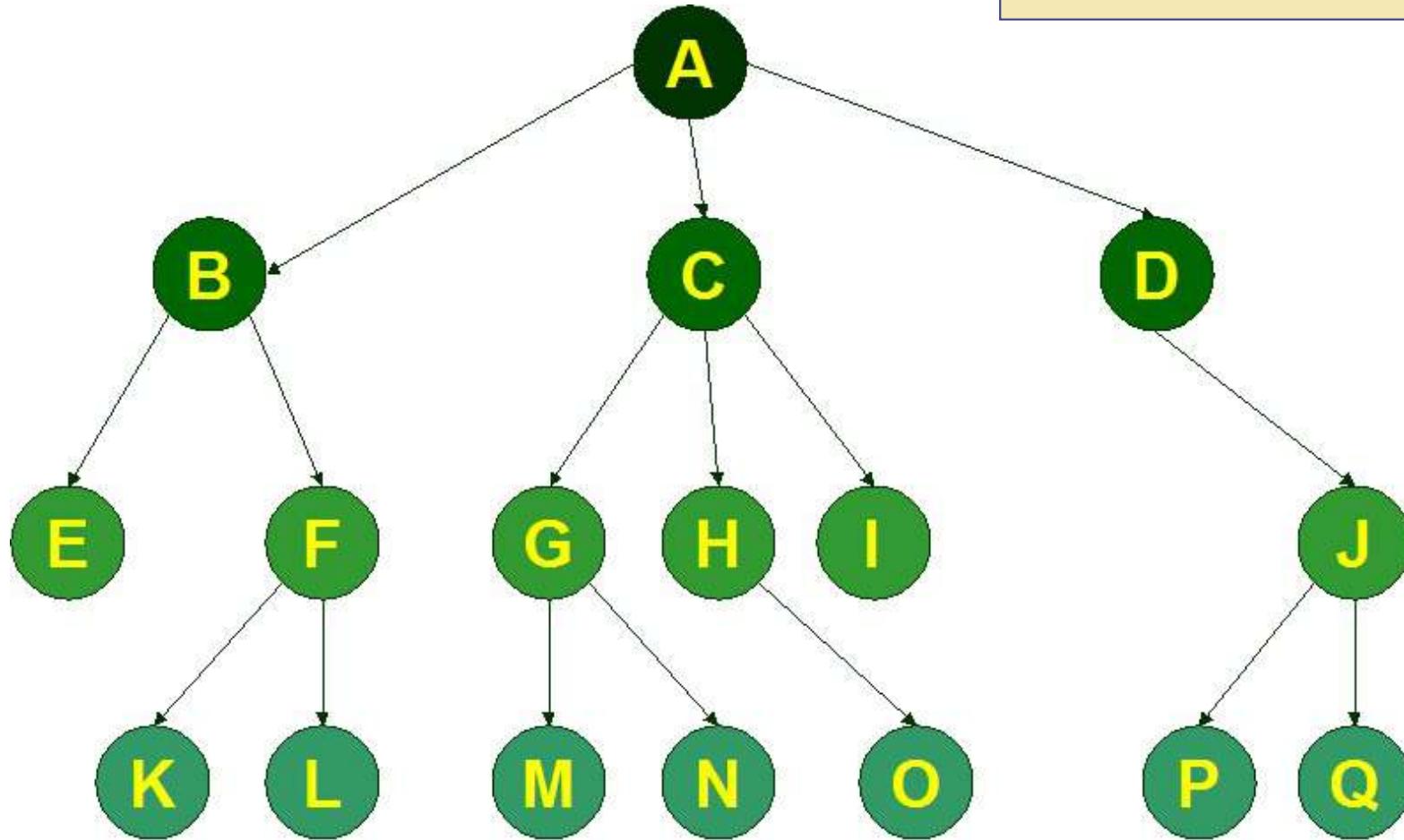
Filho

B, C e D são de A
E e F são de B
G, H e I são de C
J é de D

Genealogia das árvores

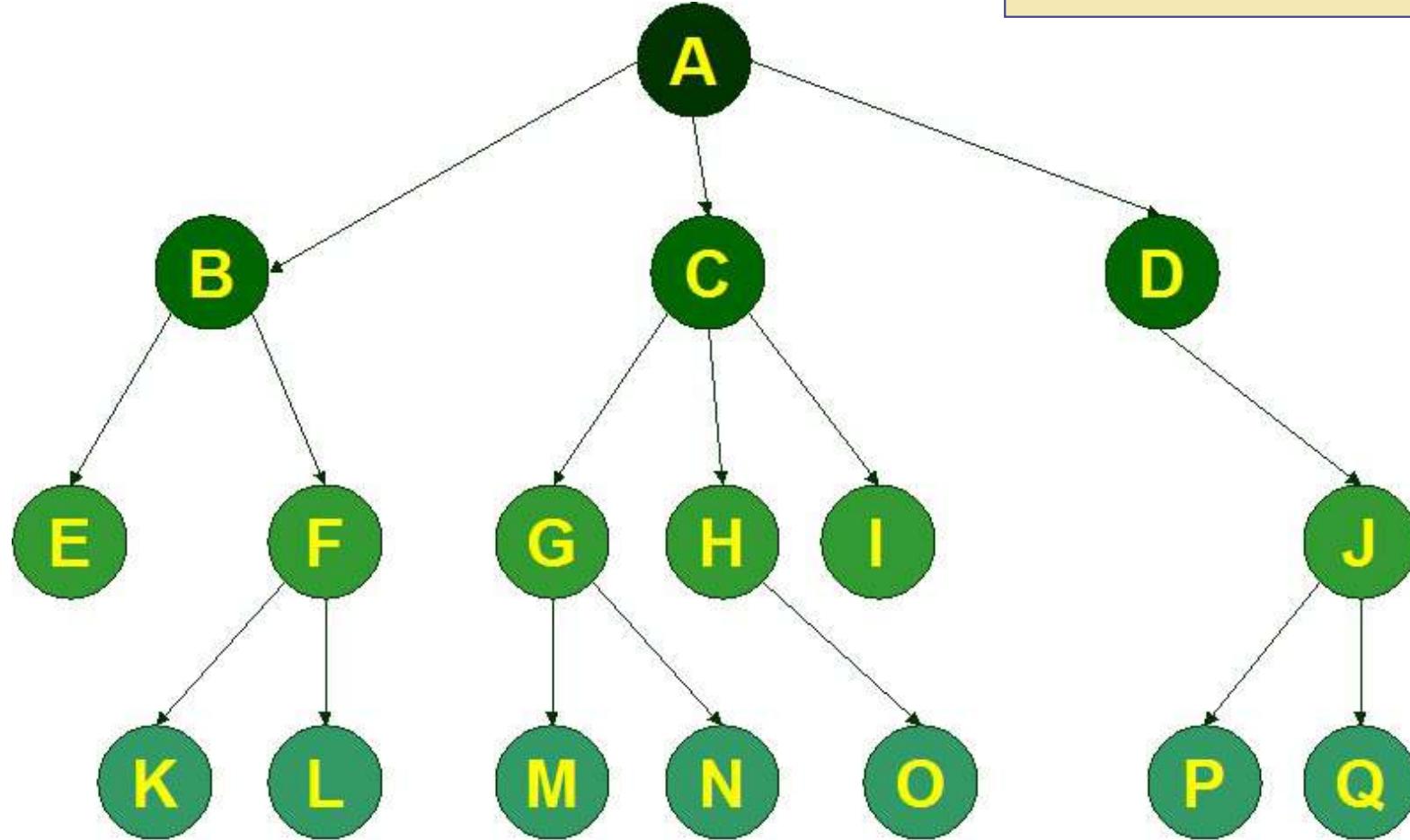
Nó Irmão

Irmão
B, C e D
E e F; G, H e I;
J não tem irmãos
K e L; M e N; P e Q



Genealogia das árvores

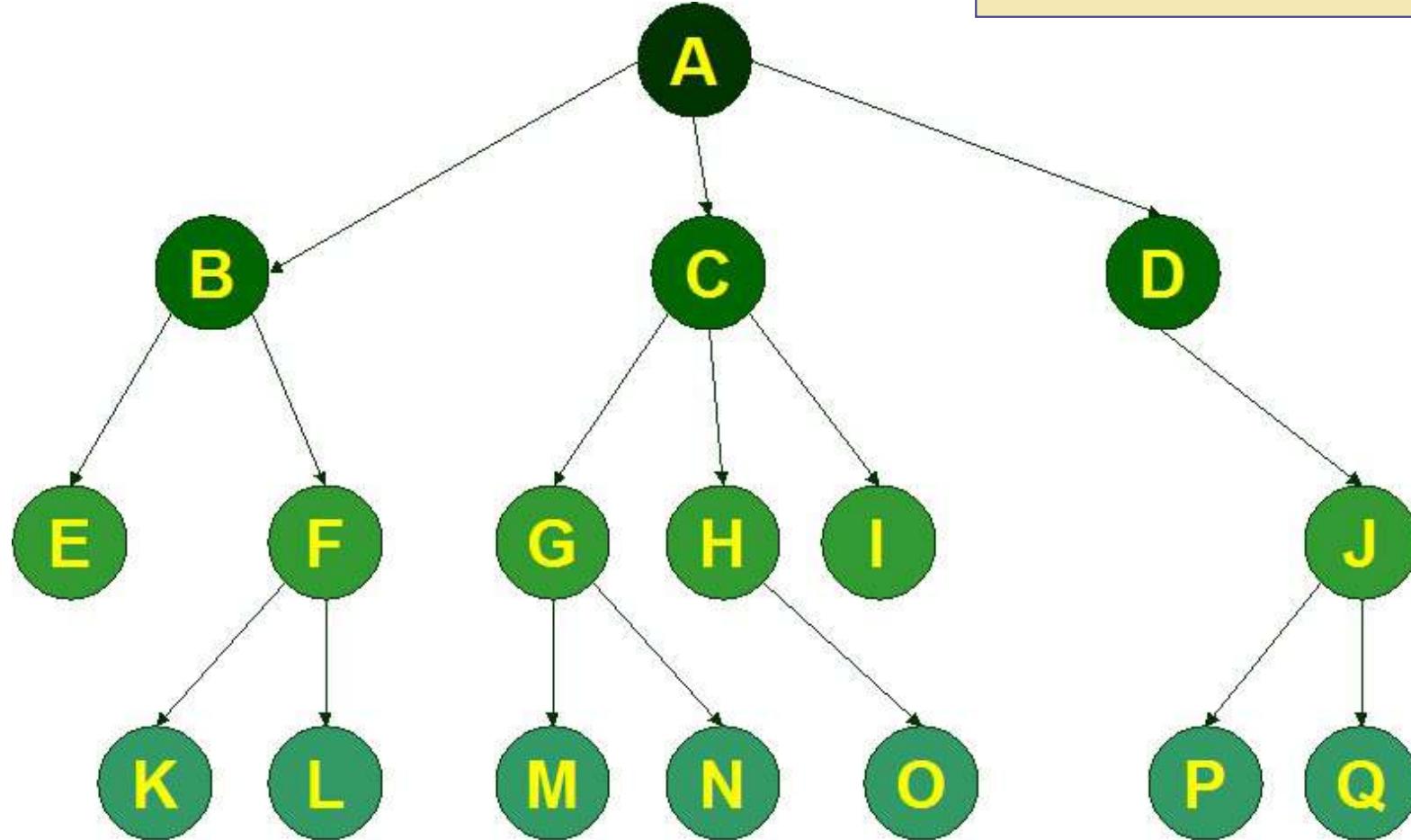
Nós Ancestrais



Ancestrais
de B é apenas A
de G são C e A
de K são F, B e A
de P são J, D e A

Genealogia das árvores

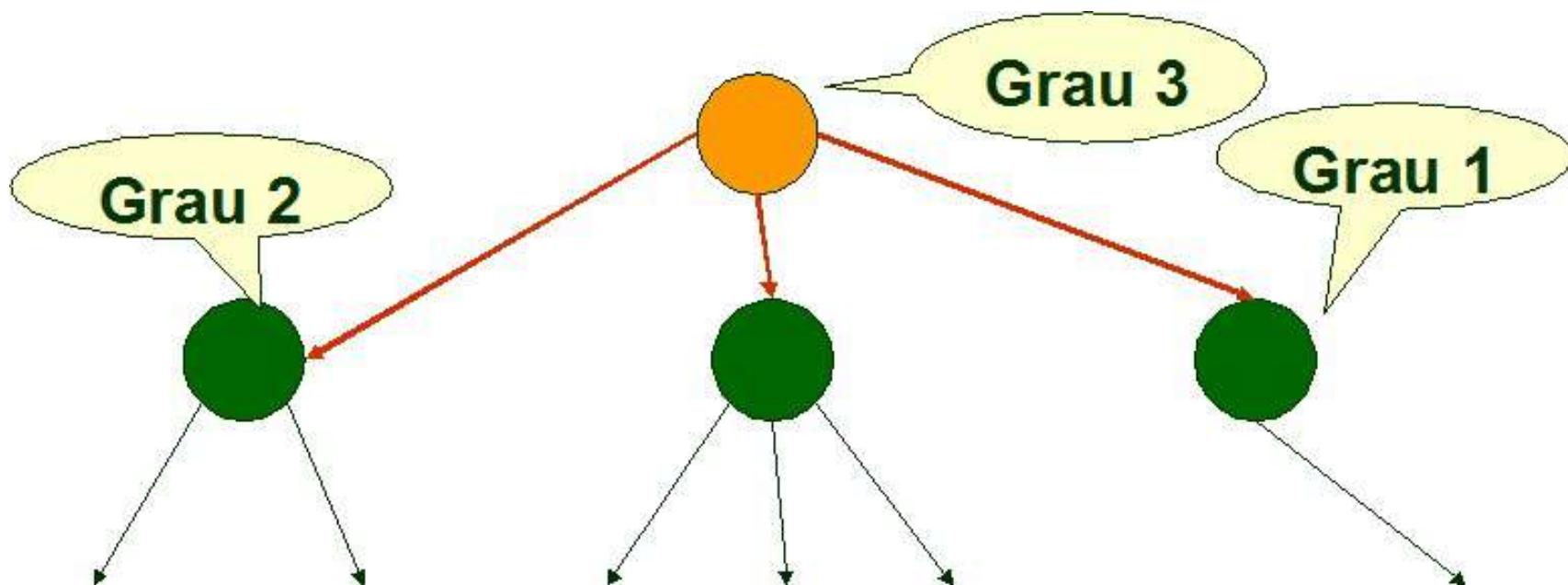
Nós Descendentes



Descendentes
de A são todos os nós
de B são E, F, K e L
de C são G, H, I, M, N e O
de D são J, P e Q

Grau ou ordem de um nó

- ◆ É o número de sub-árvore desse nó



Tipos de Nó

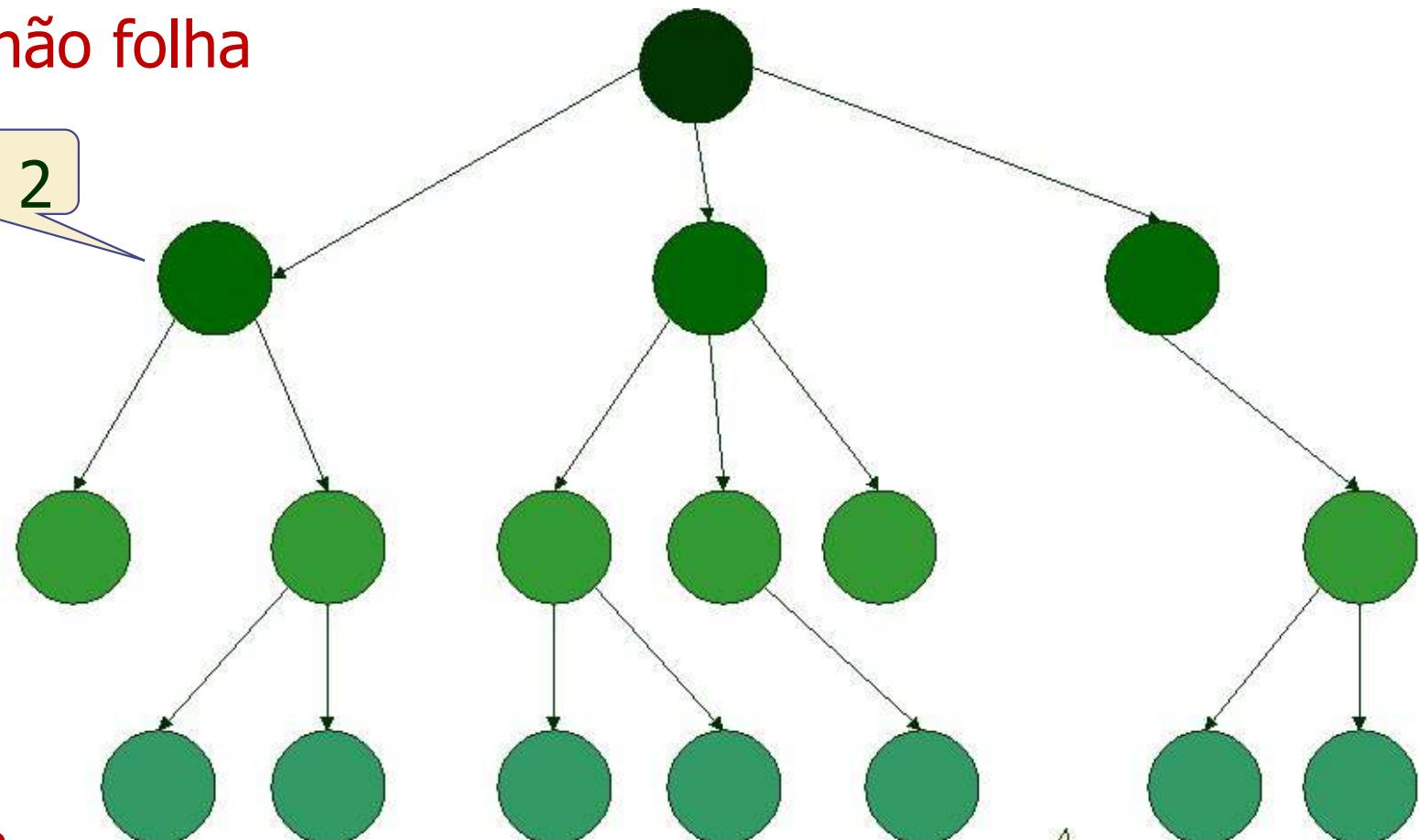
Nós folha e não folha

- ◆ O nó cujo grau é igual a ZERO é denominado de nó-folha (ou nó externo)
 - Não possui descendentes
- ◆ Os nós intermediários são, por consequência, denominados de nós não-folhas (ou nós internos)

Exemplo

Nó não folha

Grau 2



Nó-folha

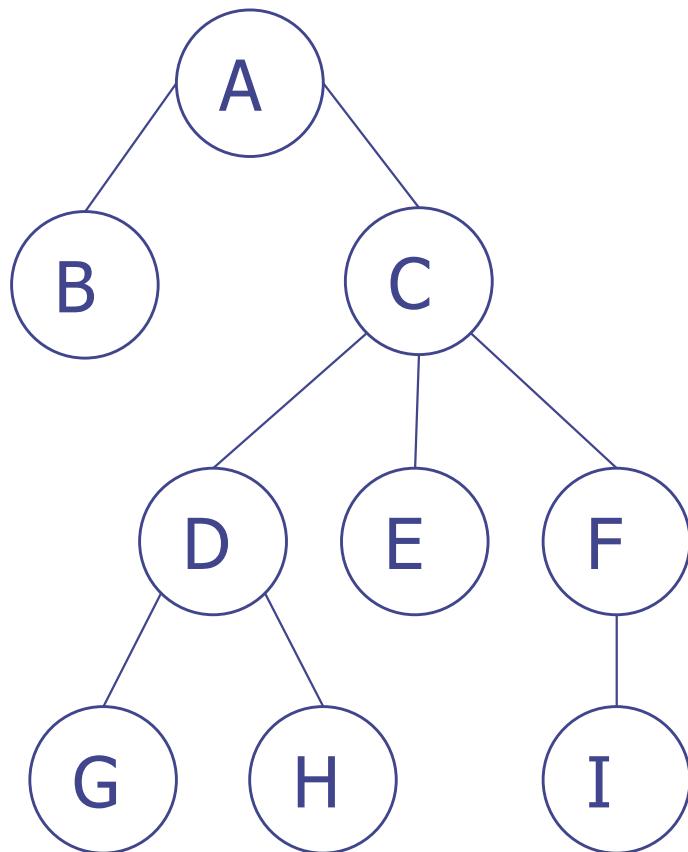
Grau 0

Grau 0

Nó-folha

Grau ou ordem da árvore

- ◆ É o maior grau de seus nós constituintes



- ◆ Graus de Saída

- $GS(A)=2$
- $GS(B)=0$
- $GS(C)=3$
- $GS(D)=2$
- $GS(E)=0$
- $GS(F)=1$
- $GS(G)=0$
- $GS(H)=0$
- $GS(I)=0$

Grau(T) = 3

Nós Internos

Folhas

Mais definições...

◆ Floresta

- Conjunto de zero ou mais árvores

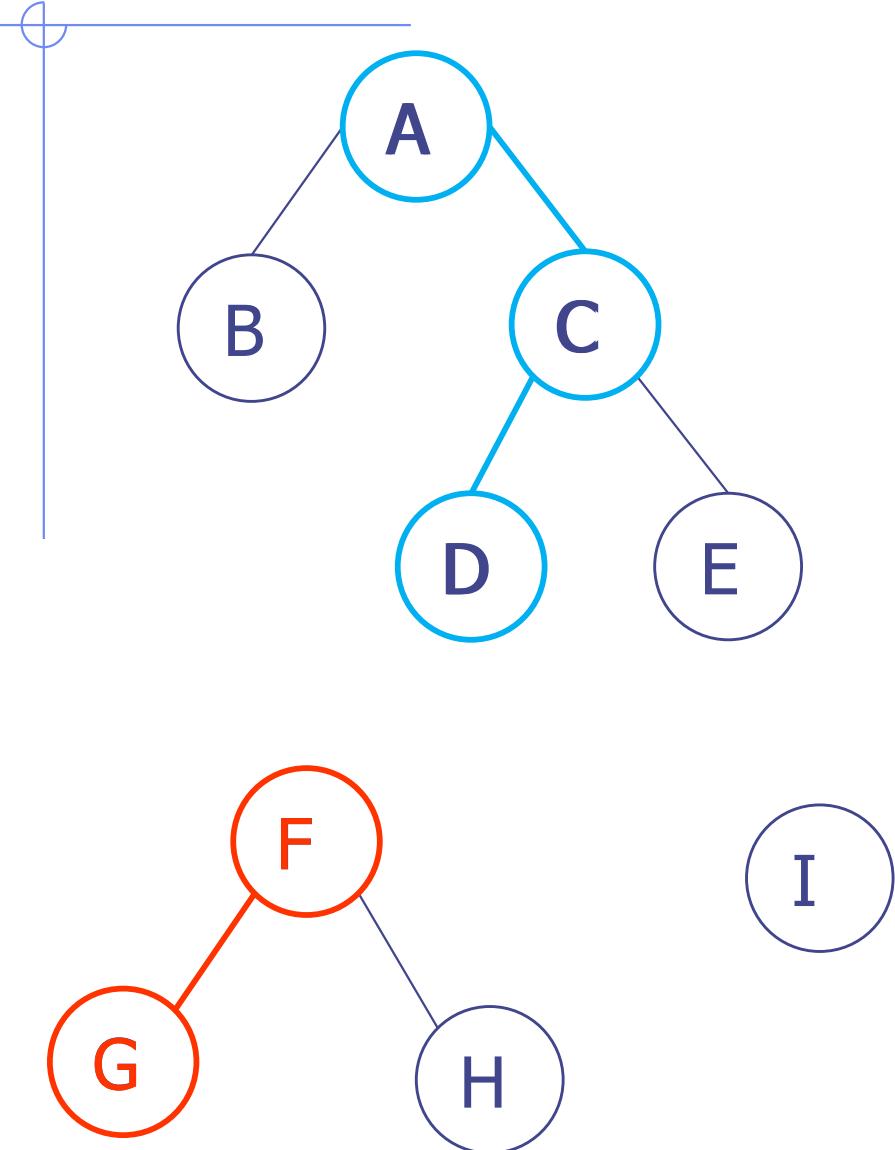
◆ Caminho na árvore

- Seqüência de nós distintos v_1, v_2, \dots, v_k , tal que existe sempre entre nós consecutivos (isto é, entre v_1 e v_2 , entre v_2 e v_3 , ..., $v_{(k-1)}$ e v_k) a relação "é filho de" ou "é pai de"

◆ Comprimento do caminho

- Um caminho que passa por v_k vértices é obtido pela seqüência de $k-1$ pares
- O valor $k-1$ é o comprimento do caminho

Mais definições...

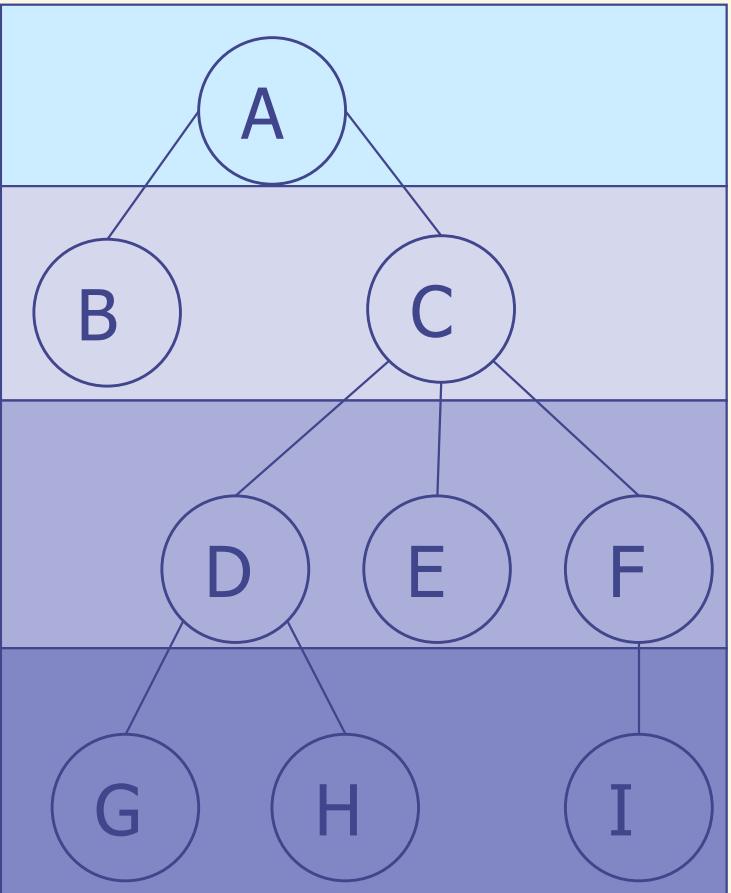


- ◆ Floresta $T = T_A \cup T_F \cup T_I$
 - $T_A = \{A, B, C, D, E\}$
 - $T_F = \{F, G, H\}$
 - $T_I = \{I\}$
- ◆ Caminhos:
 - Caminho1 = (A, C, D)
 - Caminho2 = (F, G)
- ◆ Comprimentos:
 - Comprimento (Caminho1) = 2
 - Comprimento (Caminho2) = 1

Mais definições...

- ◆ O **nível ou profundidade de um nó** é o número de nós do caminho da raiz até o nó (raiz tem nível zero)
 - Estando um nó no nível k , os seus filhos estarão no nível $(k + 1)$
- ◆ **Altura de um nó v**
 - É o comprimento do maior caminho do nó v até um de seus descendentes
- ◆ **Altura de uma árvore T ou $h(T)$**
 - É a altura do nó raiz
 - A altura da sub-árvore de raiz v é representada por $h(v)$

Mais definições...



$$h(T) = 3$$

NÍVEIS	
A	0
B, C	1
D, E, F	2
G, H, I	3

**ALTURAS EM
RELAÇÃO AO NÓ
FOLHA COM MAIOR
NÍVEL**

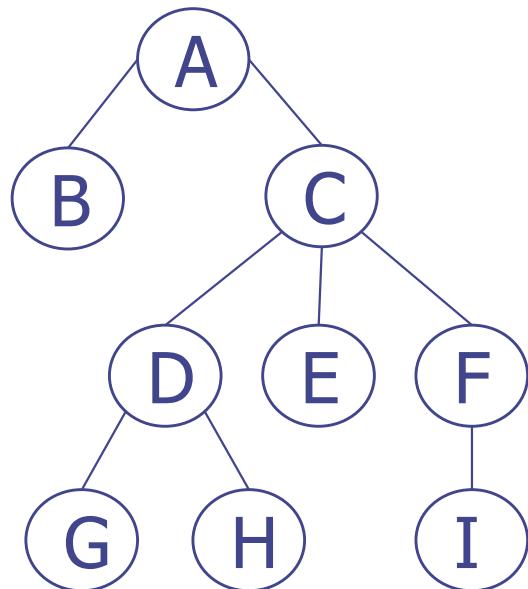
$h(A)$	3
$h(B)$	0
$h(C)$	2
$h(D)$	1
$h(E)$	0
$h(F)$	1
$h(G)$	0
$h(H)$	0
$h(I)$	0

Mais definições...

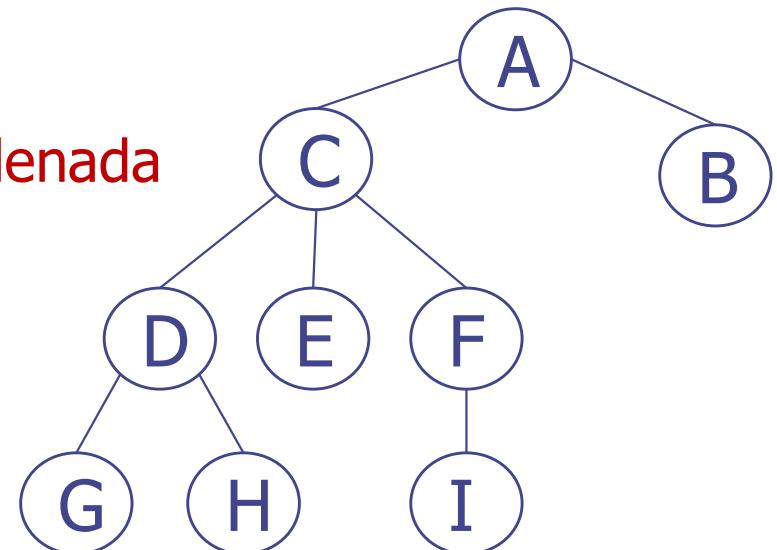
◆ Árvore Ordenada

- Os filhos de cada nó estão ordenados (assume-se ordenação da esquerda para a direita)

ordenada



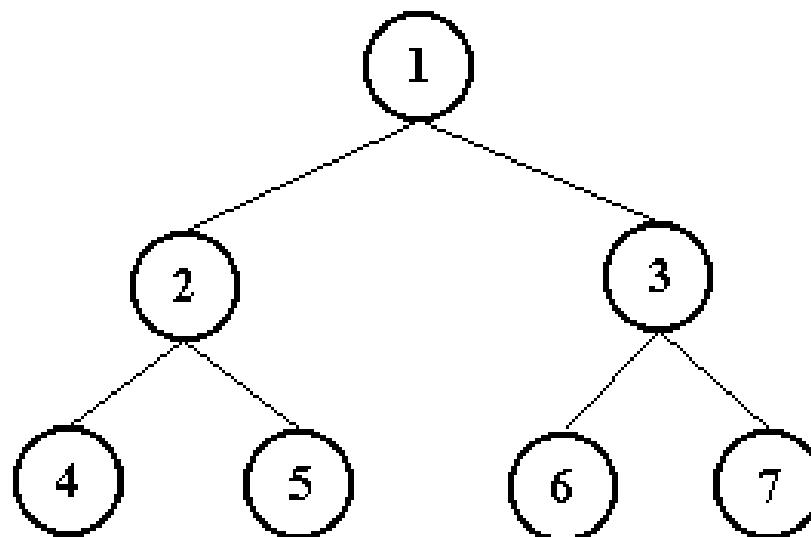
desordenada



Mais definições...

♦ Árvore Cheia

- Uma árvore de grau d é uma árvore cheia se possui o número máximo de nós, isto é, todos os nós tem número máximo de filhos exceto as folhas, e todas as folhas estão na mesma altura



Árvore
cheia de
grau 2

Operações sobre árvores (TAD)

◆ Dados

- árvore A

◆ Operações

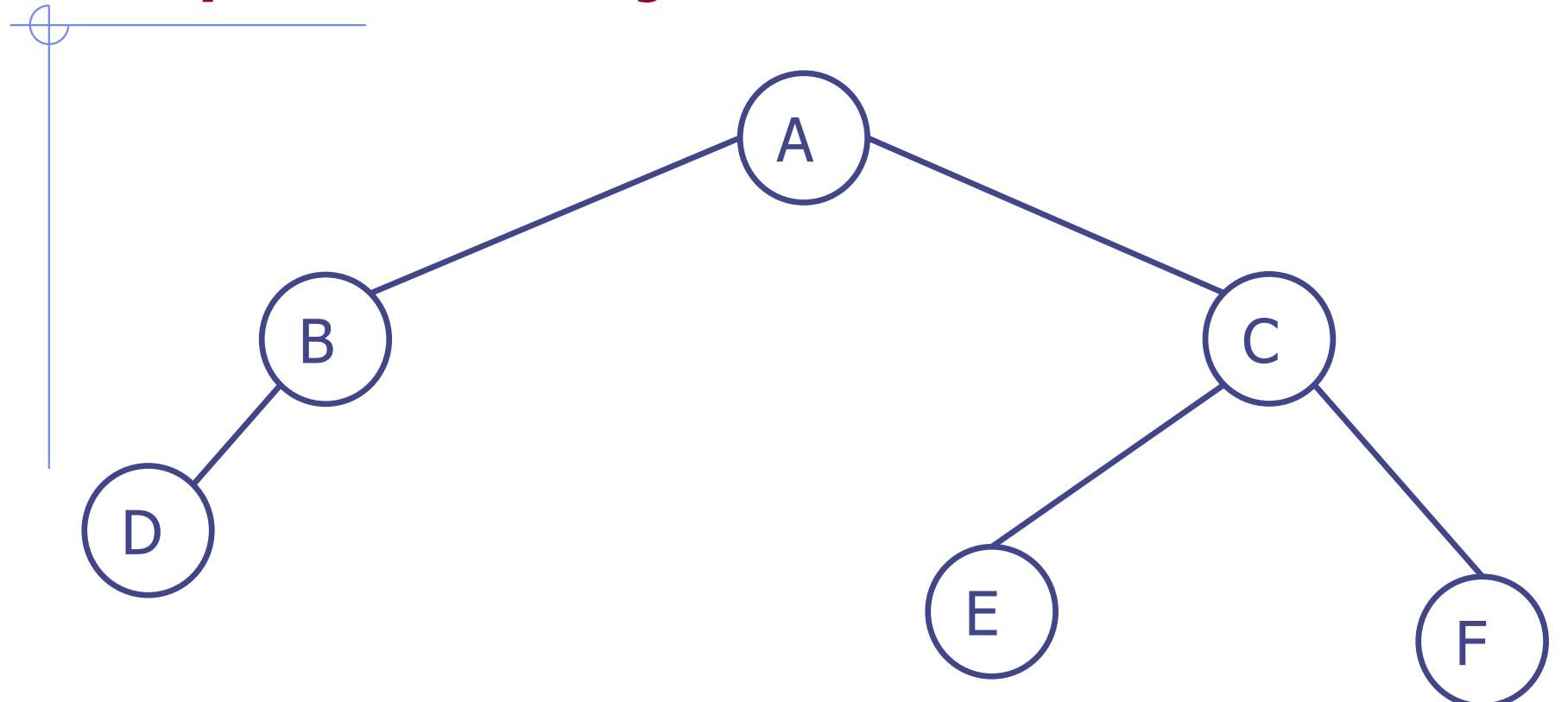
- criação da árvore
- inserção de um novo nó
 - ◆ raiz
 - ◆ folha
 - ◆ posição intermediária
- exclusão de um determinado nó
- acesso a um nó
- determinar forma de percorrer a árvore
- destruição da árvore

Representação de Árvores

- ◆ Podemos utilizar o mesmo conceito utilizado em listas ligadas e, por meio de ponteiros, representar o relacionamento existente entre um nó e seus nós-filhos

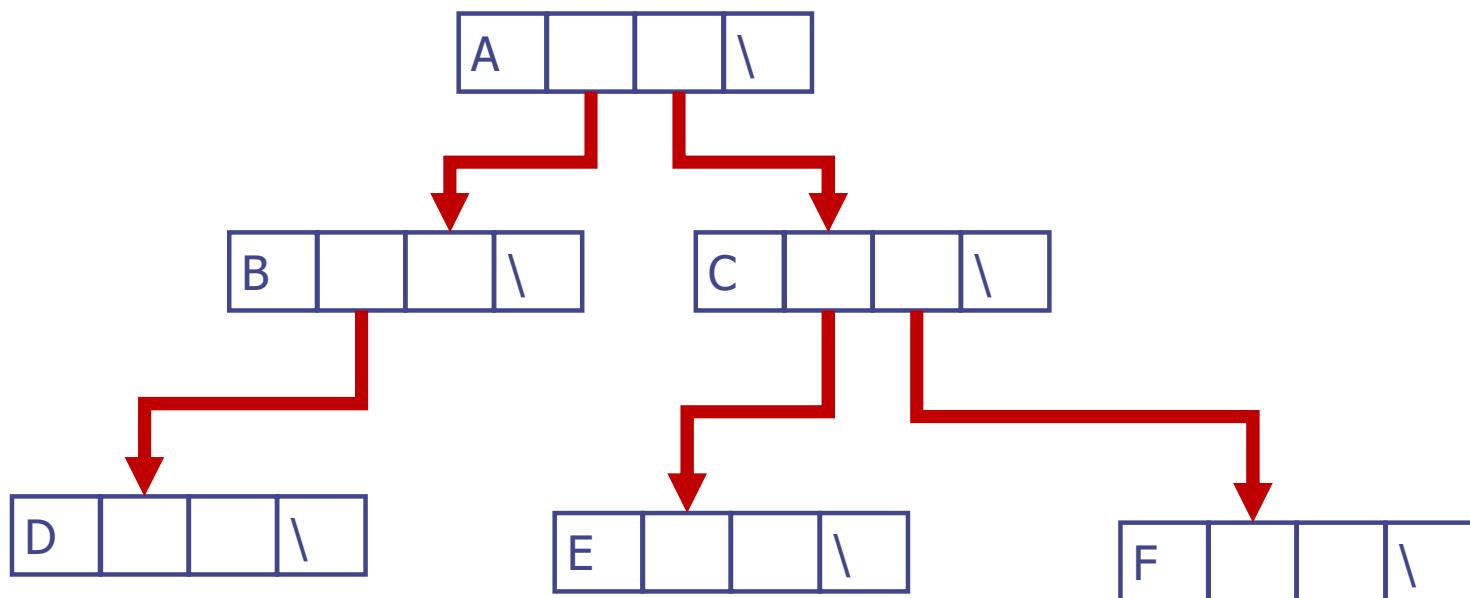


Representação de Árvores



ENCADEAMENTO

Representação de Árvores

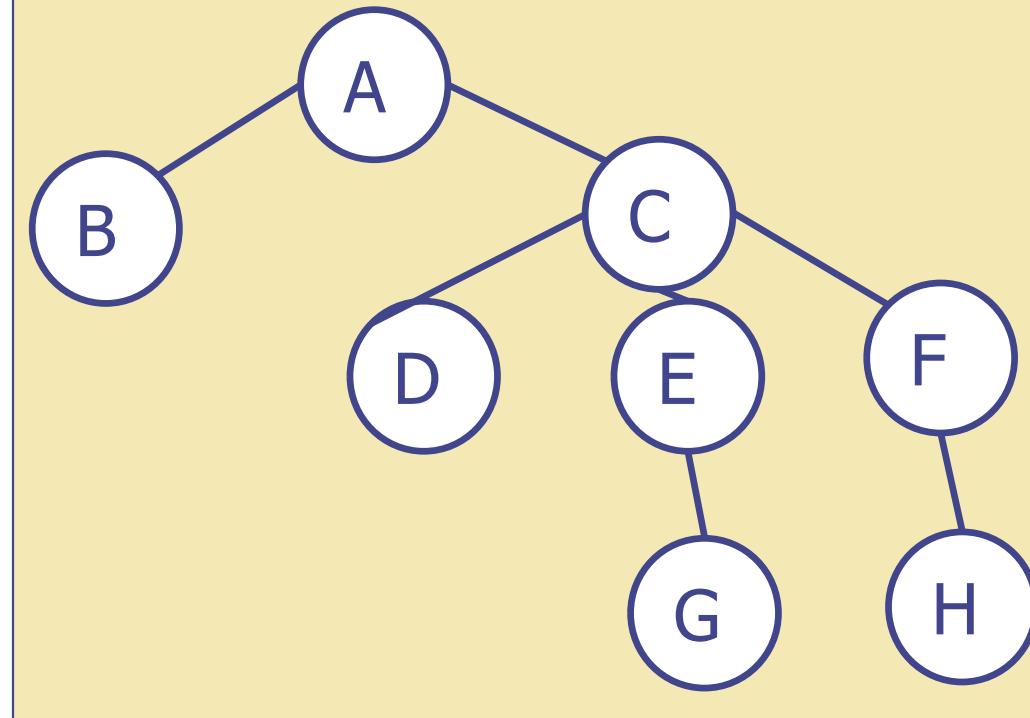


ENCADEAMENTO

Exercícios

◆ Para a árvore:

1. Quantas subárvore contém T_A ?
2. Quais os nós são folhas?
3. Quais nós são internos?
4. Qual o grau de cada nó?
5. Qual o grau da árvore?
6. Dê o nível e altura do nó F?



Bibliografia

- ◆ Ziviani, N. Projeto de Algoritmos, Thomson Learning, 2005.

- ◆ Tenenbaum, A.M.; Langsam, Y.; Augenstein, M.J. Estruturas de Dados usando C, Makron Books, 1995.

Árvores Binárias de Pesquisa

Profa. Maria Camila Nardini Barioni

camila.barioni@ufu.br

Bloco B - sala 1B137

2º trimestre de 2023

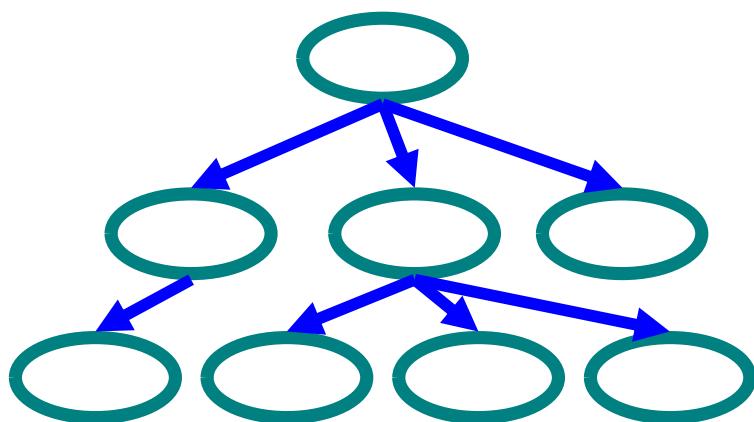
Roteiro

- ◆ Definições
- ◆ Tipos de árvores binárias
- ◆ Representação
- ◆ Implementação em C
- ◆ Árvore binária de pesquisa

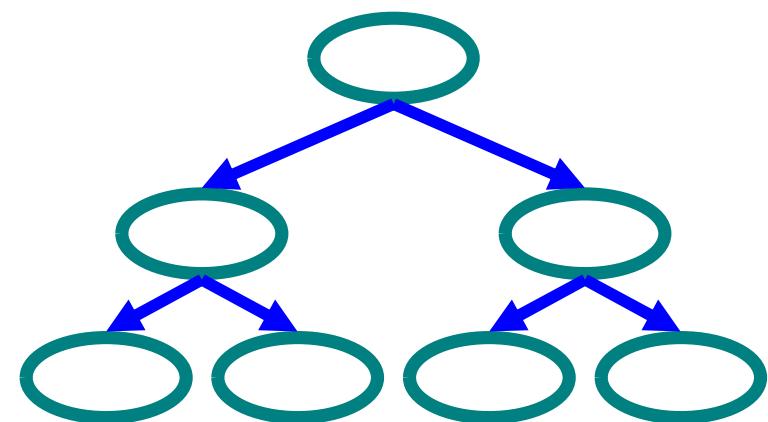
Árvores e Árvores Binárias

♦ Árvore Binária

- O grau de cada nó deve ser menor ou igual a dois
- Subárvores de um nó: esquerda e direita
 - ◆ Se o grau de um nó for 1, deve ser especificado se a sua subárvore é a da esquerda ou da direita



Árvore



Árvore Binária

Árvores Binárias: Definição

- ◆ Conjunto finito de elementos que é ou vazio ou composto de três conjuntos disjuntos
 - o primeiro contém um único elemento, a **raiz**
 - os outros dois subconjuntos são árvores binárias
 - ◆ as subárvore da **esquerda** e da **direita**
 - ◆ as subárvore da esquerda ou da direita podem estar vazias

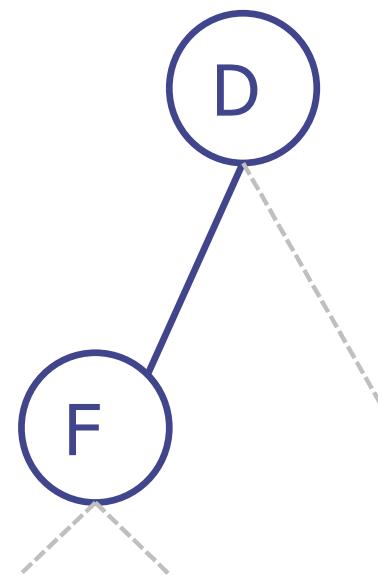
Árvores Binárias: Definição

- ◆ Considerando que os dois filhos de cada nó interno são ordenados:
 - o filho da esquerda e
 - o filho da direita
 - Cada nó interno tem que ter filho da direita ou da esquerda, sendo que um ou ambos podem ser nós externos
- ◆ Uma árvore binária é uma árvore ordenada, na qual cada nó tem 0, 1, ou 2 filhos
 - cada filho corresponde a uma árvore binária

Árvores Binárias

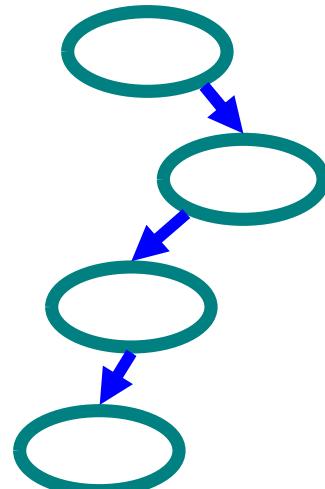
O número de subárvores a esquerda e a direita vazias em uma árvore binária com n nós é de $n+1$

- se $n = 1$ então 2 subárvores vazias
- se $n = 2$ então 3 subárvores vazias

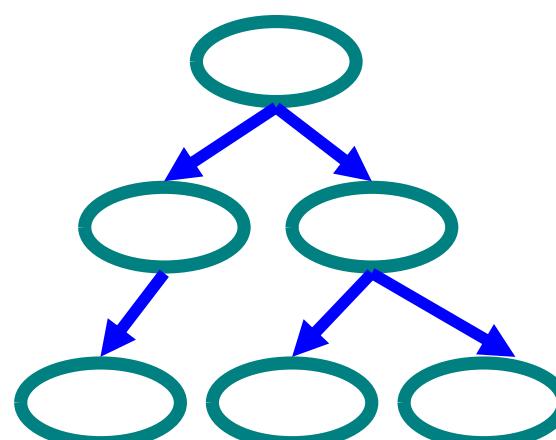


Tipos de Árvores Binárias

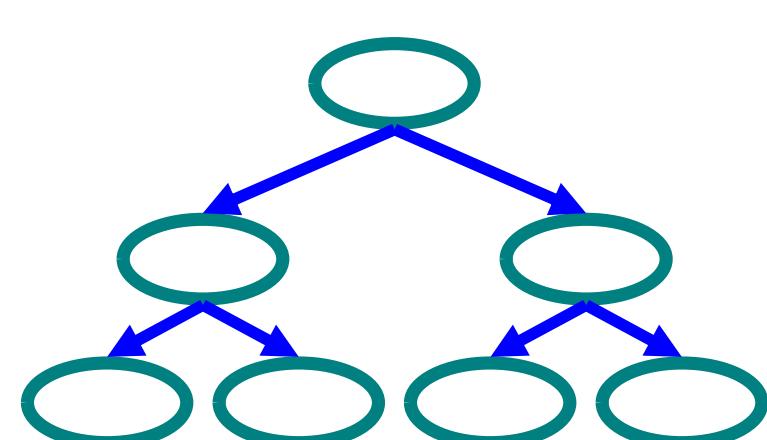
- ◆ Degenerada – nós com grau de saída 1
- ◆ Completa – todos os nós com grau de saída 2
- ◆ Balanceada – A “maioria” dos nós com grau de saída 2
 - Ideias intuitivas (existem definições mais formais)



Árvore binária
degenerada

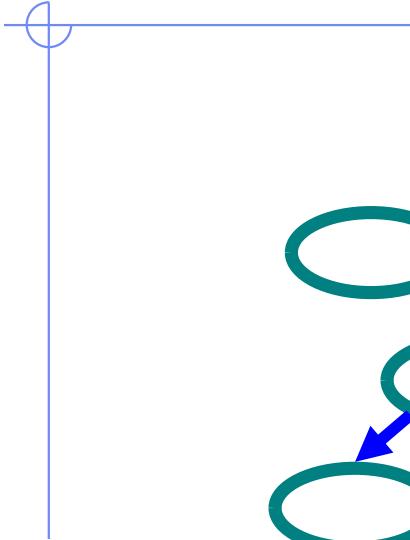


Árvore binária
balanceada



Árvore binária
completa

Árvore Binária de altura máxima



Árvore binária
degenerada

- ◆ Para árvores com n nós:
 - altura máxima: cada nó não folha só possui um filho - ziguezague
 - sua altura é $n-1$

Árvore Binária - altura mínima

- ◆ nível 0 – (somente a raiz) contém um nó
- ◆ nível 1 – contém no **máximo** 2 nós
- ◆ nível 2 – contém no **máximo** 4 nós
-
- ◆ no nível L - pode conter no **máximo** 2^L nós

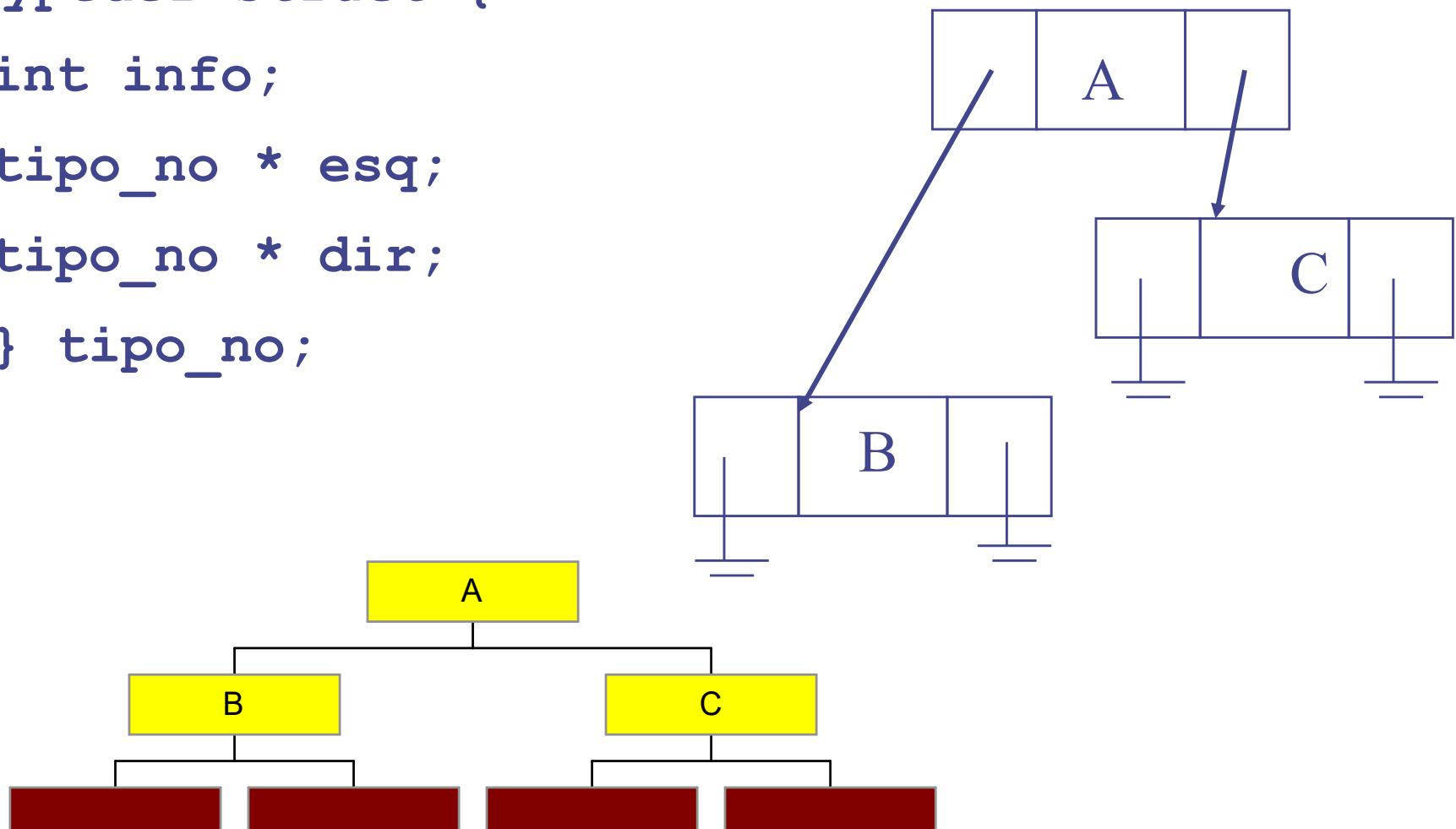
⇒ árvore binária **cheia** de altura d tem exatamente 2^L nós em cada nível $0 \leq L \leq d$

Árvore Binária - Vantagens

- ◆ Otimização de alocação de memória
- ◆ Mais fáceis de manipular
- ◆ Implementação de operações é muito simplificada

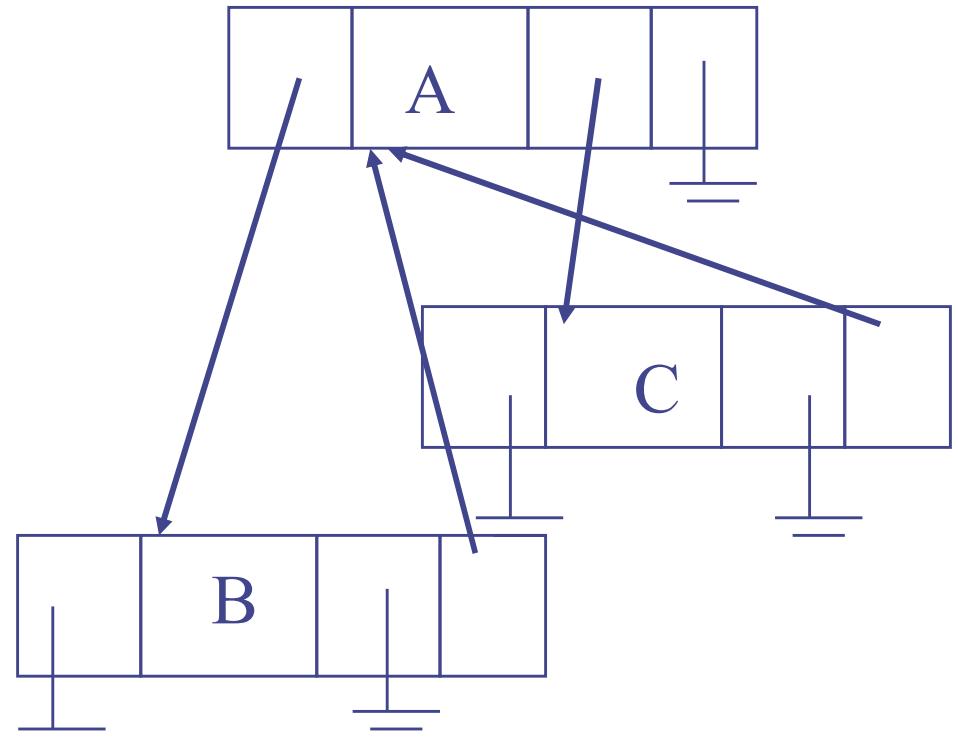
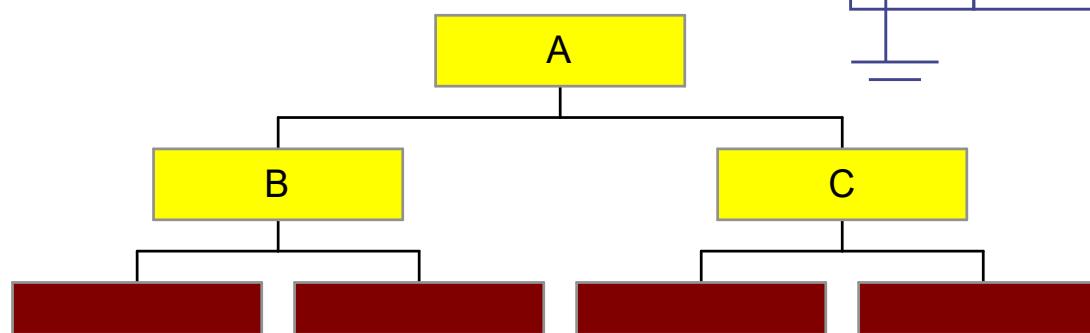
Representando Árvores Binárias

```
◆ typedef struct {  
    int info;  
    tipo_no * esq;  
    tipo_no * dir;  
} tipo_no;
```



Representando Árvores Binárias

- ```
typedef struct {
 char info;
 tipo_no * esq;
 tipo_no * dir;
 tipo_no * pai;
} tipo_no
```



# Estrutura do dicionário para Árvore Binária em C

```
1. typedef int TipoChave;
2. typedef struct Registro{
3. TipoChave Chave;
4. /*outros componentes */
5. }Registro;
6. typedef struct No *Apontador;
7. typedef struct No {
8. Registro Reg;
9. Apontador Esq, Dir;
10. }No;
11.typedef Apontador TipoDicionario;
```

# Árvore Binária em C

```
1. void Inicializa (Apontador *Dicionario) {
2. *Dicionario = NULL;
3. }
4.
5. int Vazio (Apontador *Dicionario) {
6. return (Dicionario == NULL);
7. }
```

# Percorso de uma Árvore

- ◆ Percorso em uma árvore é o ato de percorrer todos os nós da árvore, com o objetivo de consultar ou alterar alguma informação
- ◆ A árvore deve ser percorrida de forma sistemática de tal modo que cada nó seja “visitado” apenas uma vez
  - O termo “visita” significa realizar qualquer operação com o conteúdo do nó, como consultar ou alterar algum campo
- ◆ Métodos de percurso em árvores
  - Pré-fixado, Central (em ordem), Pós-fixado

# Percorso de uma Árvore

## ◆ Percorso Pré-fixado

1. visitar raiz
2. percorrer subárvore da esquerda
3. percorrer subárvore da direita

## ◆ Percorso Central (em ordem)

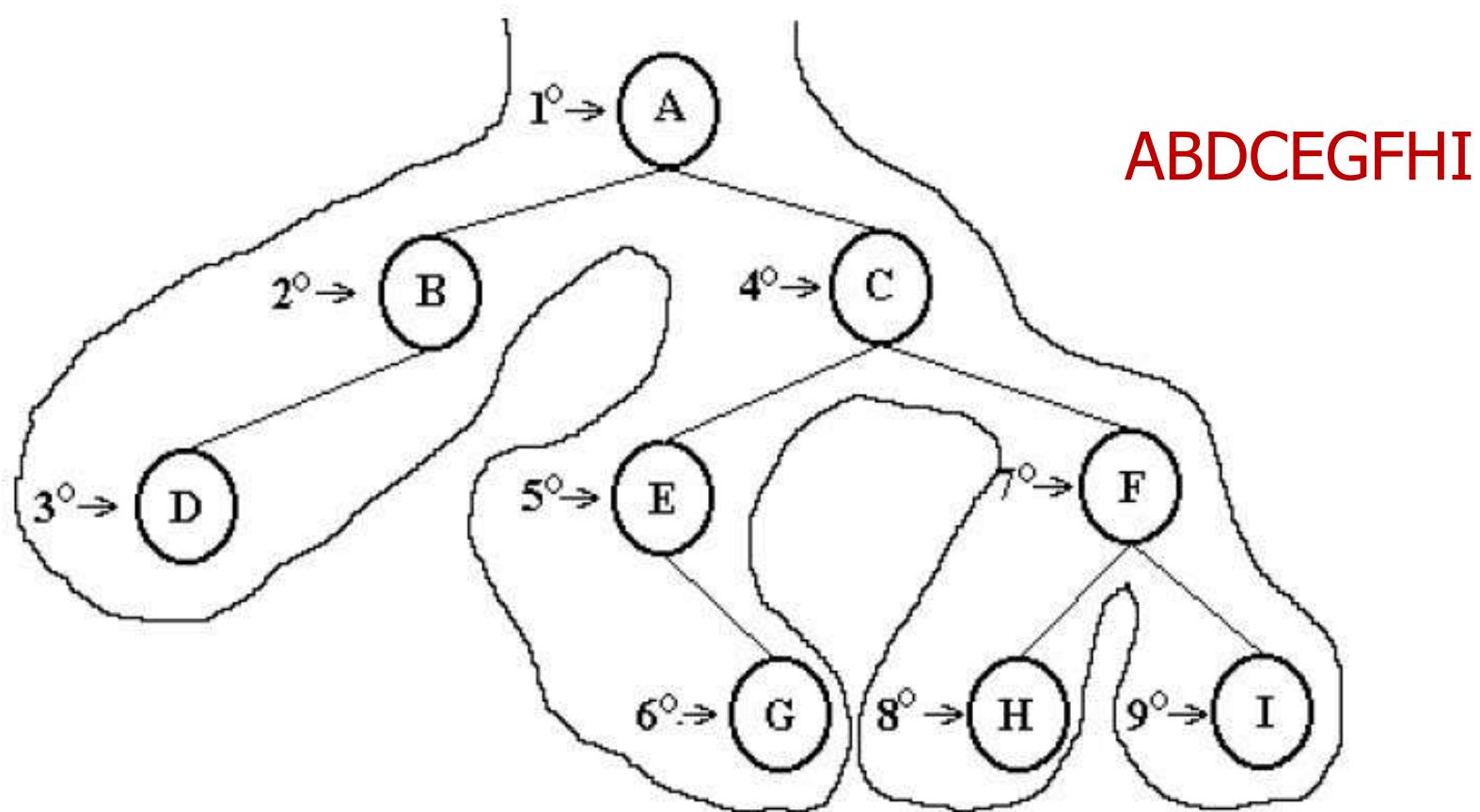
1. percorrer subárvore da esquerda
2. visitar raiz
3. percorrer subárvore da direita

## ◆ Percorso Pós-fixado

1. percorrer subárvore da esquerda
2. percorrer subárvore da direita
3. visitar raiz

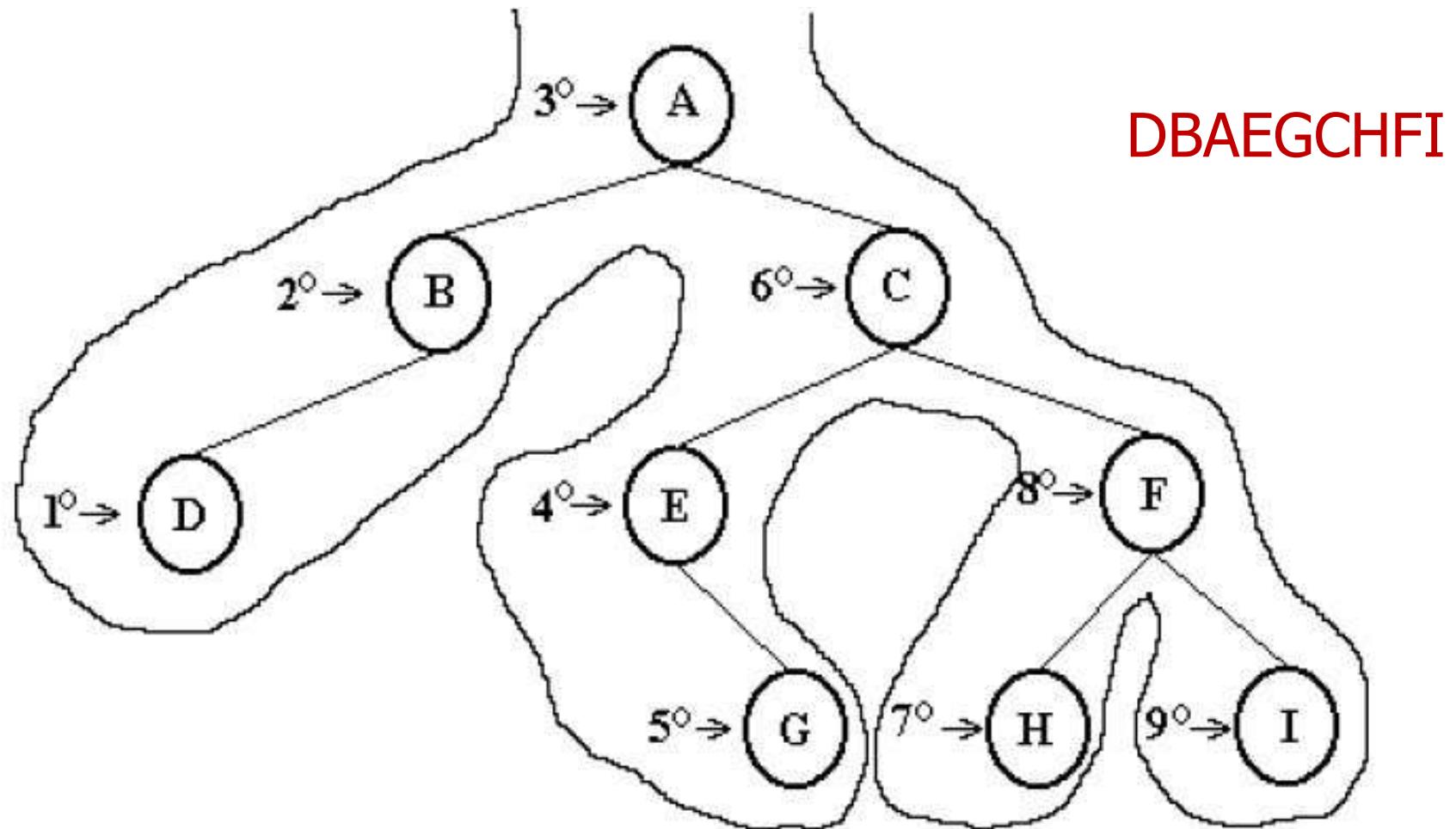
# Percorso de uma Árvore

## ◆ Percurso Pré-fixado



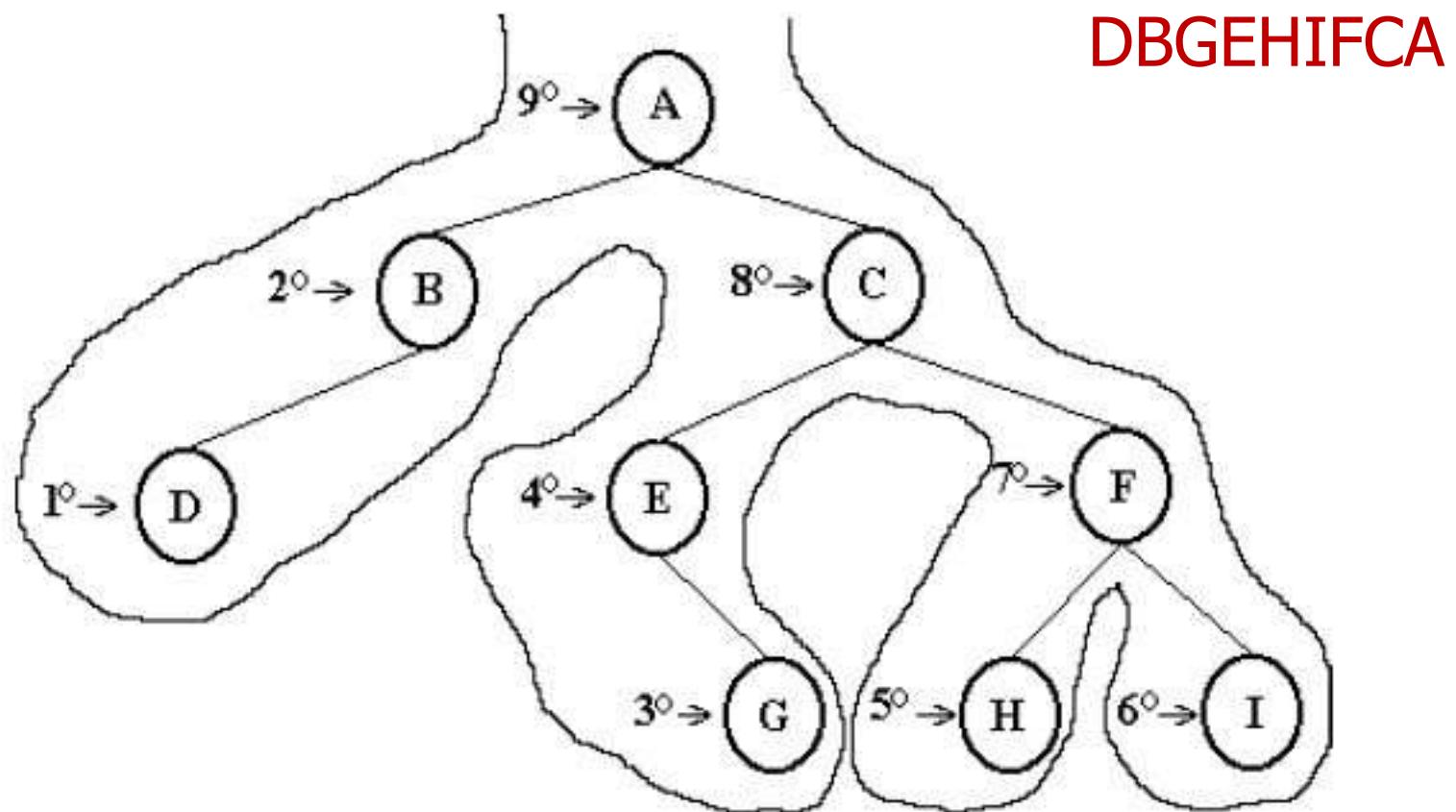
# Percorso de uma Árvore

## ◆ Percurso Central



# Percorso de uma Árvore

## ◆ Percorso Pós-Ordem



# Percorso Central em C

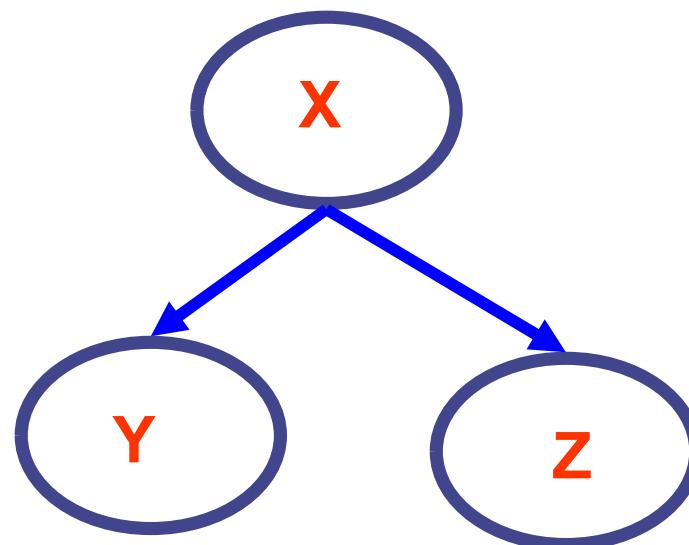
```
1. void Central(Apontador p) {
2. if (p == NULL) return;
3. Central (p->Esq) ;
4. printf ("%d\n", p->Reg.Chave) ;
5. Central (p->Dir) ;
6. }
```

# Árvore Binária de Pesquisa

- ◆ Utilizada para armazenar chaves e depois recuperá-los
- ◆ Alternativas: Tabela hash e Lista linear
- ◆ Hash
  - Excelente para acesso direto
  - Pobre para acesso sequencial
- ◆ Lista
  - Excelente para acesso sequencial
  - Pobre para acesso direto
- ◆ Árvore binária
  - Bom compromisso entre acesso direto e sequencial

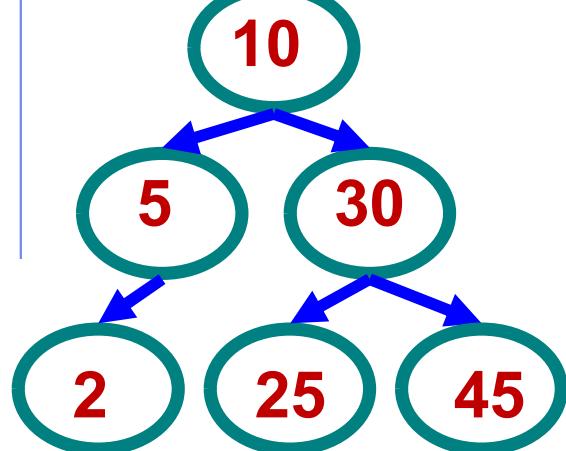
# Árvore Binária de Pesquisa

- ◆ Propriedade chave: valor de um nó
  - Valores menores: sempre na subárvore da esquerda
  - Valores maiores: sempre na subárvore da direita
  - Exemplo:
    - ◆  $X > Y$
    - ◆  $X < Z$

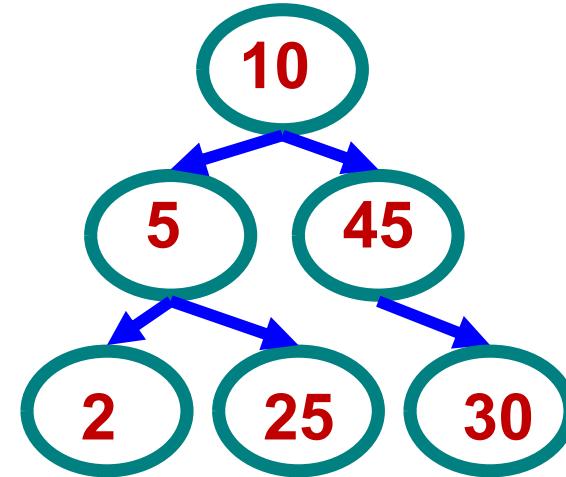
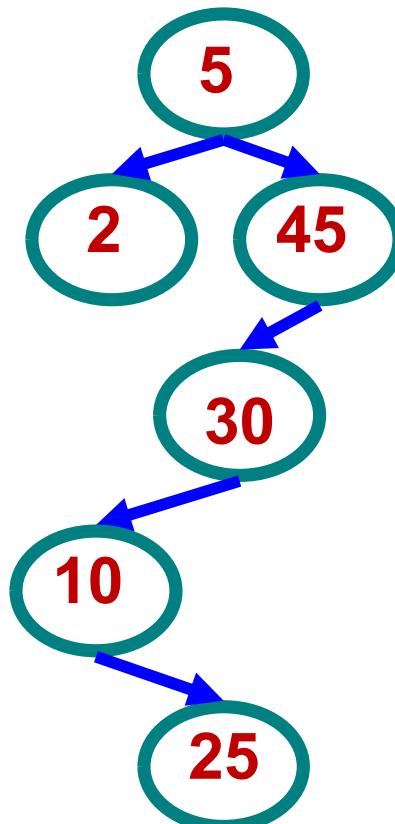


# Árvore Binária de Pesquisa

## Exemplos



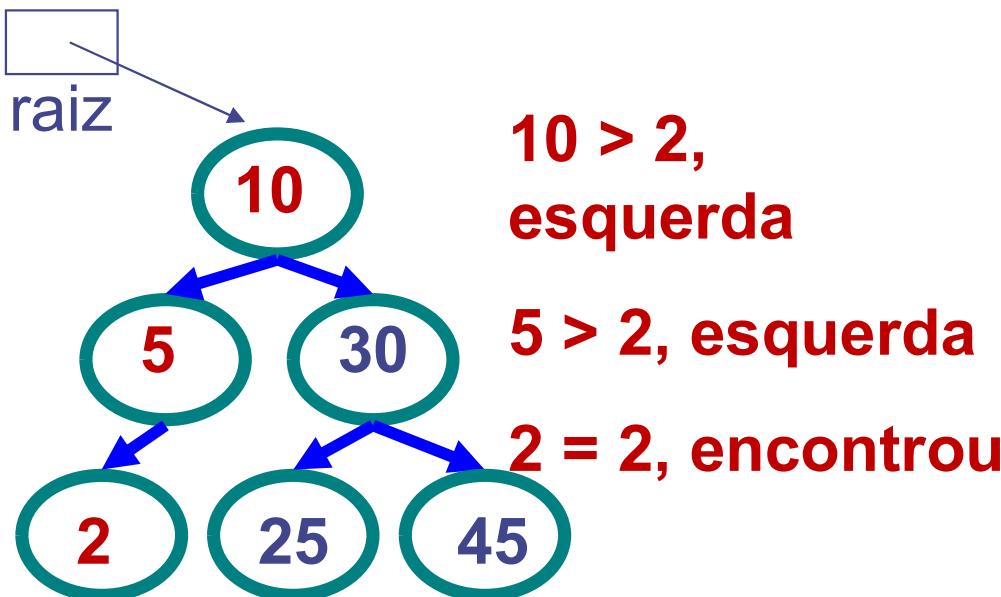
É uma árvore  
binária de pesquisa



Não é uma árvore  
binária de pesquisa

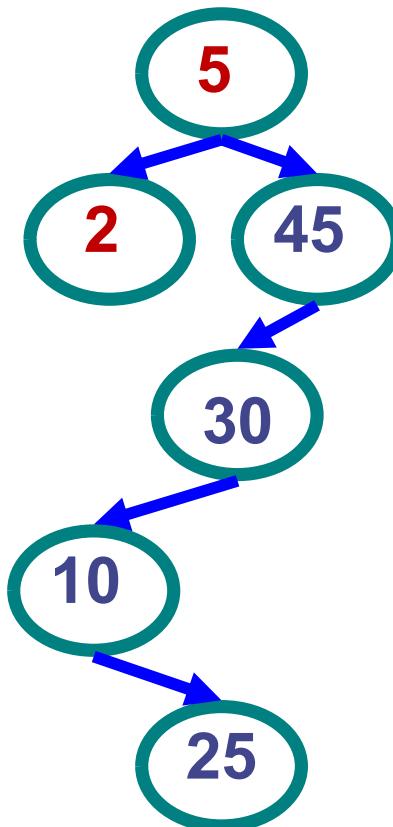
# Exemplos de Pesquisas Binárias

◆ pesquisar ( raiz, 2 )



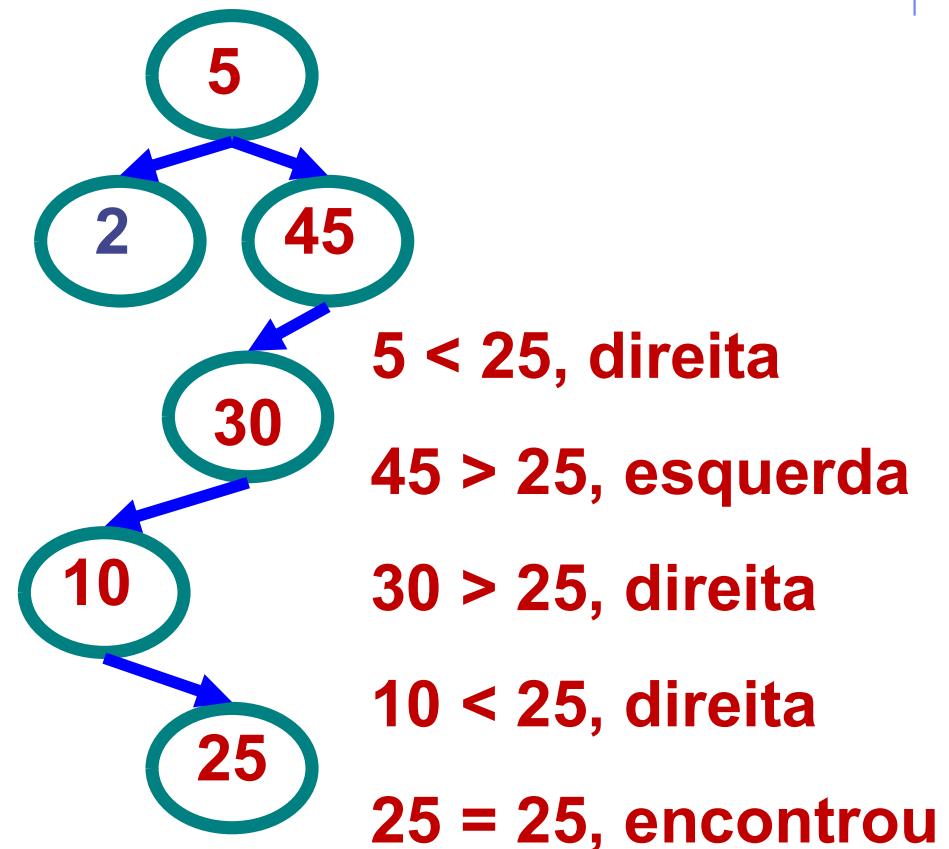
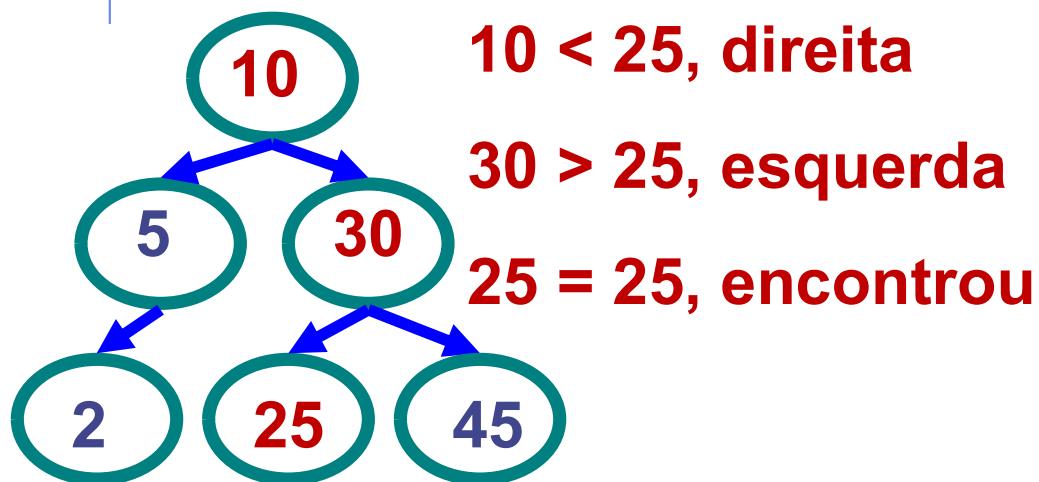
$5 > 2$ , esquerda

$2 = 2$ , encontrou



# Exemplos de Pesquisas Binárias

♦ pesquisar ( raiz, 25 )



# Árvore Binária em C: Pesquisa

```
1. void Pesquisa(Registro *x, Apontador *p) {
2. if (*p == NULL) {
3. printf("Erro: Registro nao esta na arvore \n");
4. return;
5. }
6. if (x->Chave < (*p)->Reg.Chave) {
7. Pesquisa(x, &(*p)->Esq);
8. return;
9. }
10. if (x->Chave > (*p)->Reg.Chave)
11. Pesquisa(x, &(*p)->Dir);
12. else *x = (*p)->Reg;
13. }
```

# Propriedades das Árvores Binárias

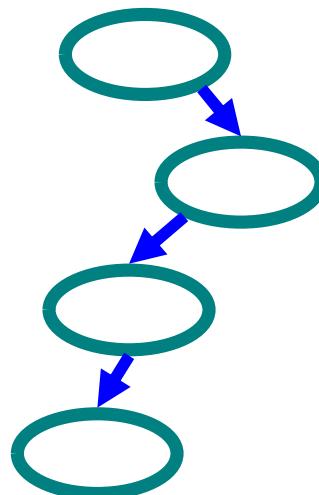
## ◆ Tempo de pesquisa

- Proporcional à altura da árvore
- Árvore binária balanceada
  - ◆ Tempo  $O(\log N)$
- Árvore binária degenerada
  - ◆ Tempo  $O(N)$
  - ◆ Semelhante a percorrer uma lista encadeada ou um vetor

# Propriedades das Árvores Binárias

## ◊ Degenerada

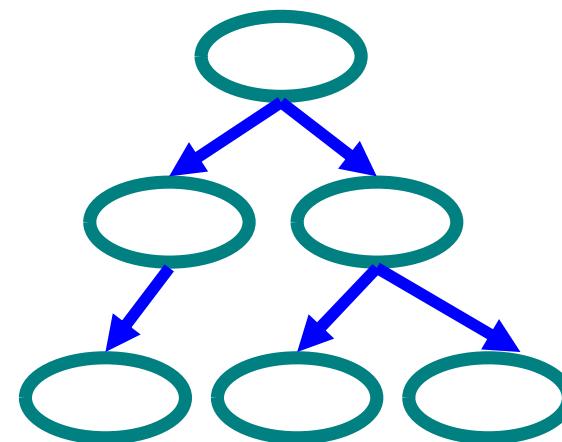
- Altura =  $O(N)$  para  $N$  nós
- Semelhante à lista encadeada



**Árvore binária  
degenerada**

## ◊ Balanceada

- Altura =  $O(\log N)$  para  $N$  nós
- Útil para pesquisas



**Árvore binária  
balanceada**

# Construção da Árvore Binária

- ◆ Como construir e manter árvores binárias?
  - Operações de inserção e remoção
- ◆ Manutenção da propriedade das chaves deve ser preservada sempre (invariante)
  - Valores menores: sempre na sub-árvore da esquerda
  - Valores maiores: sempre na sub-árvore da direita

# Operação de Inserção

## ◆ Algoritmo

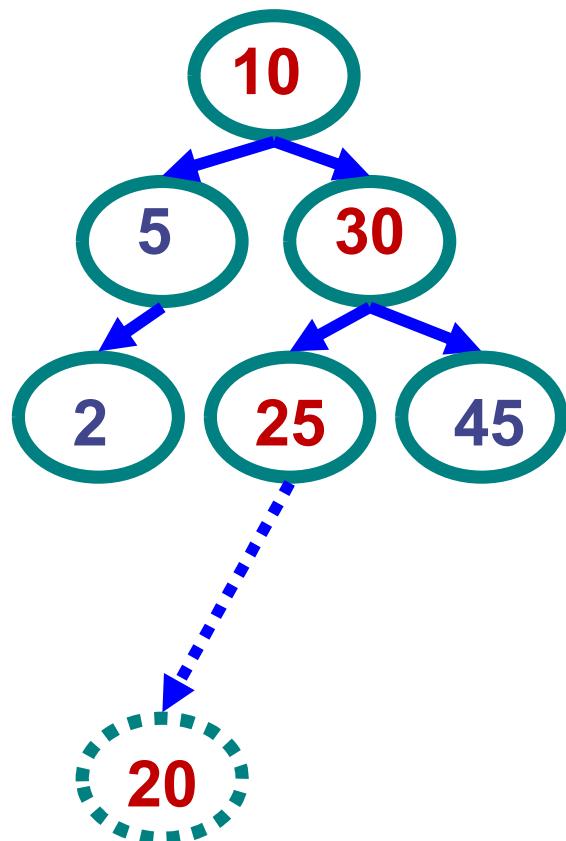
1. Realiza busca pelo valor X
2. Busca termina no nó Y (se X não está na árvore)
3. Se  $X < Y$ , inserir nova folha X como uma subárvore da esquerda de Y
4. Se  $X > Y$ , inserir nova folha X como uma subárvore da direita de Y

## ◆ Observações

- O ( $\log N$ ) – somente para árvores平衡adas
- Atenção: inserções podem desbalancear a árvore

# Inserção - Exemplo

## ◆ Inserir (20)



**10 < 20, direita**

**30 > 20, esquerda**

**25 > 20, esquerda**

**Inserir 20 na  
esquerda**

# Árvore Binária em C: Inserir

```
1. void Insere(Registro x, Apontador *p) {
2. if (*p == NULL) {
3. *p = (Apontador)malloc(sizeof(No));
4. (*p)->Reg = x;
5. (*p)->Esq = NULL;
6. (*p)->Dir = NULL;
7. return;
8. }
9. if (x.Chave < (*p)->Reg.Chave) {
10. Insere(x, &(*p)->Esq);
11. return;
12. }
13. if (x.Chave > (*p)->Reg.Chave)
14. Insere(x, &(*p)->Dir);
15. else printf("Erro: Registro ja existe na arvore \n");
16. }
```

# Operação de Remoção

## ◆ Algoritmo

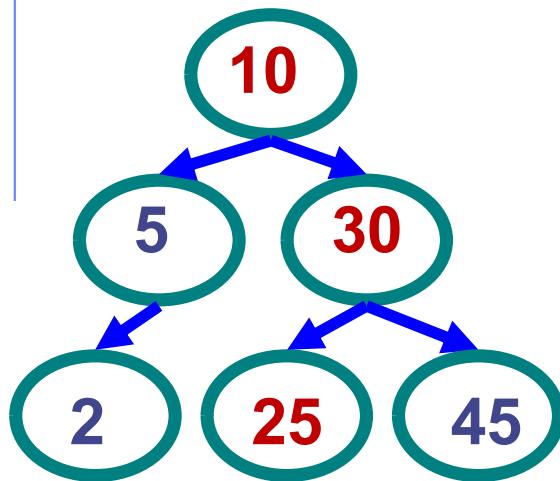
- Realizar busca por valor X
- Se X for uma folha, remover X
- Senão // precisa remover nó interno
  - ◆ Trocar X com o maior valor Y na sub-árvore da esquerda  
OU
  - ◆ Trocar X com o menor valor Z na sub-árvore da direita

## ◆ Observação

- O ( $\log N$ ) – somente para árvores平衡adas
- Atenção: remoções podem desbalancear a árvore

# Remoção – Exemplo (nó folha)

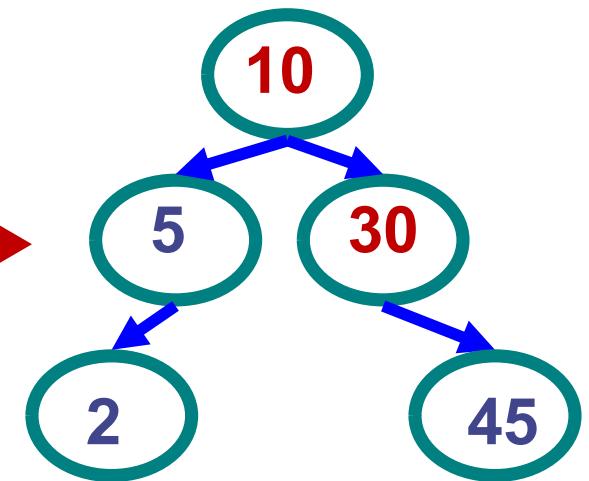
## ◆ Remover (25)



$10 < 25$ , direita

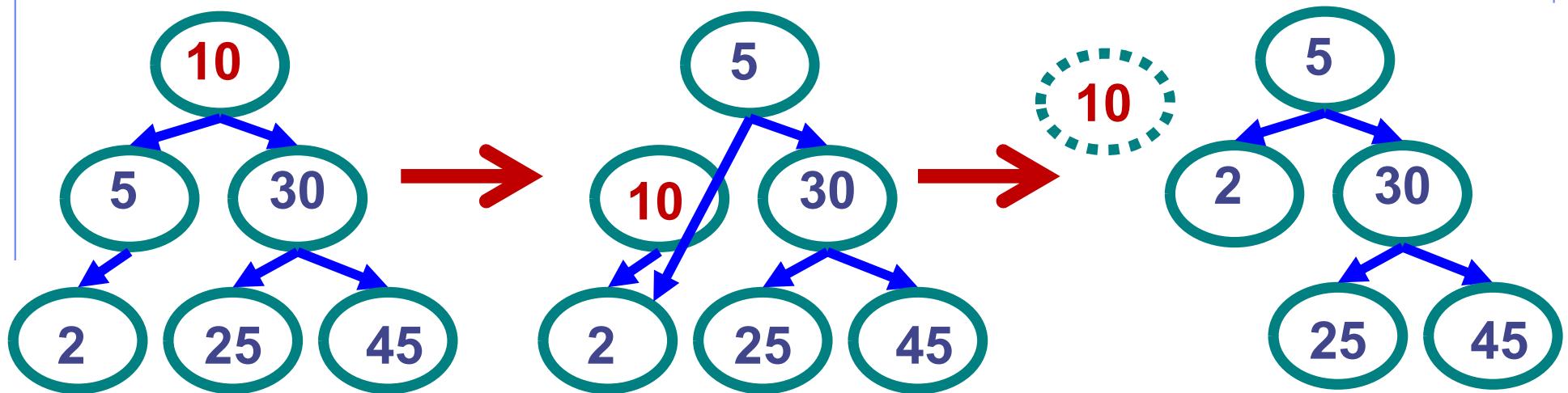
$30 > 25$ , esquerda

$25 = 25$ , remover



# Remoção – Exemplo (nó interno)

## ♦ Remover (10)



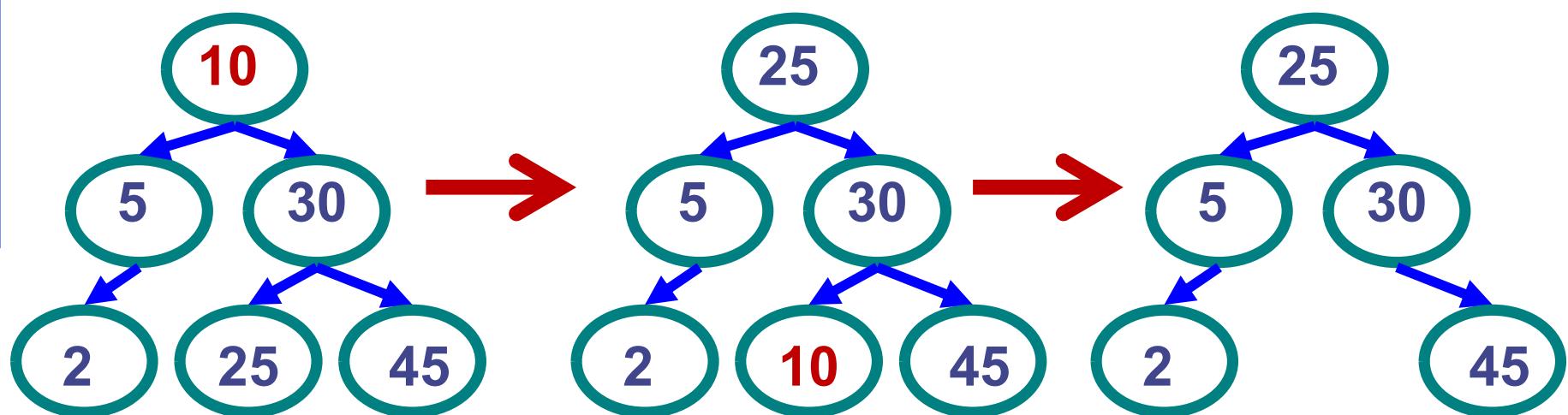
Trocar 10 com  
o maior valor  
da subárvore  
da esquerda

Reconectar a  
subárvore da  
esquerda no  
antigo 10

Remover nó

# Remoção – Exemplo (nó interno)

## ♦ Remover (10)



Trocar 10 com  
o menor valor  
da subárvore  
da direita

Remover nó

Árvore  
resultante

# Árvore Binária em C: Funções para retirar x da árvore

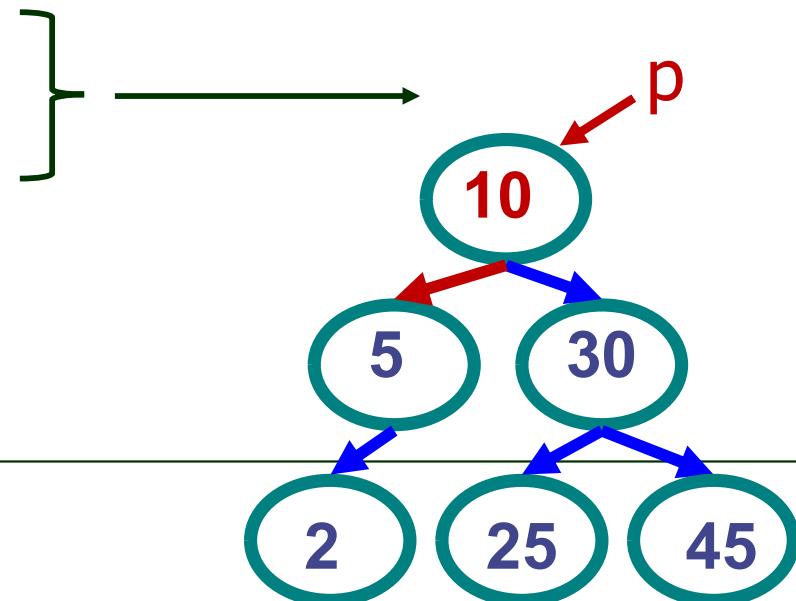
```
1. void Retira(Registro x, Apontador *p) {
2. Apontador Aux;
3. if (*p == NULL){
4. printf("Erro: Registro nao esta na arvore \n");
5. return;
6. }
7. if(x.Chave < (*p)->Reg.Chave){ Retira(x, &(*p)->Esq); return; }
8. if(x.Chave > (*p)->Reg.Chave){ Retira(x, &(*p)->Dir); return; }
9. if ((*p)->Esq != NULL){
10. Antecessor(*p, &(*p)->Esq);
11. return;
12. }
13. Aux = *p; *p = (*p)->Dir;
14. free(Aux);
15. }
```

}

Procura pelo maior valor  
da subárvore da esquerda  
(caso não vazia)

# Árvore Binária em C: Funções para retirar x da árvore

```
1. void Retira(Registro x, Apontador *p) {
2. Apontador Aux;
3. if (*p == NULL){
4. printf("Erro: Registro nao esta na arvore \n");
5. return;
6. }
7. if(x.Chave < (*p)->Reg.Chave){ Retira(x, &(*p)->Esq); return; }
8. if(x.Chave > (*p)->Reg.Chave){ Retira(x, &(*p)->Dir); return; }
9. if ((*p)->Esq != NULL){
10. Antecessor(*p, &(*p)->Esq); }
11. return;
12. }
13. Aux = *p; *p = (*p)->Dir;
14. free(Aux);
15. }
```



# Árvore Binária em C: Funções para retirar x da árvore

```
1. void Antecessor(Apontador q, Apontador *r) {
2. if ((*r)->Dir != NULL) {
3. Antecessor(q, &(*r)->Dir);
4. return;
5. }
6. q->Reg = (*r)->Reg;
7. q = *r;
8. *r = (*r)->Esq;
9. free(q);
10.}
```

# Árvore Binária em C: Funções para retirar x da árvore

```
1. void Antecessor(Apontador q, Apontador *r) {
2. if ((*r)->Dir != NULL) {
3. Antecessor(q, &(*r)->Dir);
4. return;
5. }
6. q->Reg = (*r)->Reg;
7. q = *r;
8. *r = (*r)->Esq;
9. free(q);
10.}
```

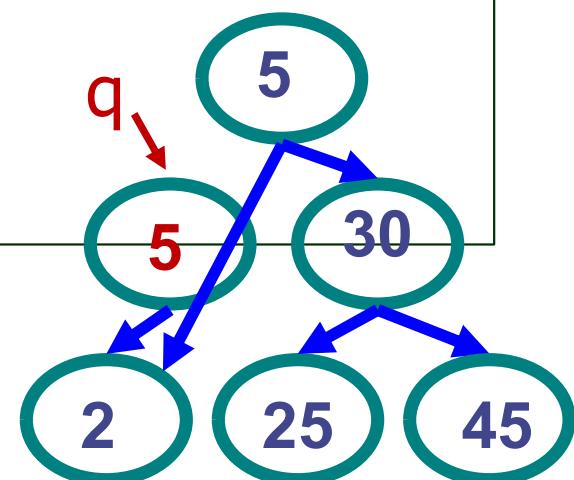
Procura pelo maior valor da subárvore da esquerda

# Árvore Binária em C:

## Funções para retirar x da árvore

```
1. void Antecessor(Apontador q, Apontador *r) {
2. if ((*r)->Dir != NULL) {
3. Antecessor(q, &(*r)->Dir);
4. return;
5. }
6. q->Reg = (*r)->Reg; }
7. q = *r;
8. *r = (*r)->Esq;
9. free(q);
10. }
```

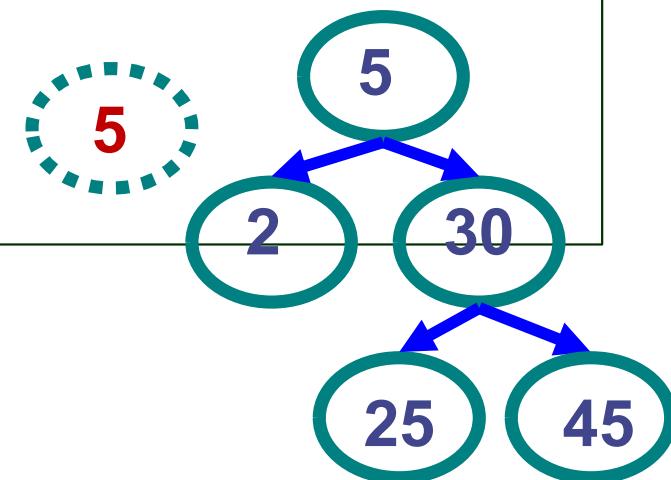
} Reconectar a subárvore da esquerda



# Árvore Binária em C: Funções para retirar x da árvore

```
1. void Antecessor(Apontador q, Apontador *r) {
2. if ((*r)->Dir != NULL) {
3. Antecessor(q, &(*r)->Dir);
4. return;
5. }
6. q->Reg = (*r)->Reg;
7. q = *r;
8. *r = (*r)->Esq;
9. free(q);
10. }
```

Remover o nó



# Árvore Binária em C: Criar

```
1. int main(int argc, char *argv[]) {
2. TipoDicionario Dicionario;
3. Registro x;
4. Inicializa(&Dicionario);
5. scanf(" %d", &x.Chave);
6. while(x.Chave > 0) {
7. Insere(x, &Dicionario);
8. scanf(" %d", &x.Chave);
9. }
10. }
```

# Bibliografia

- ◆ Ziviani, N. Projeto de Algoritmos, Thomson Learning, 2005.
  
- ◆ Tenenbaum, A.M.; Langsam, Y.; Augenstein, M.J. Estruturas de Dados usando C, Makron Books, 1995.

# Bacharelado em Ciência da Computação

GBC034 Algoritmos e Estruturas de Dados 2

Profa. Maria Camila Nardini Barioni

## PRÁTICA BUSCA BINÁRIA EM VETOR

### 1. Informações Básicas

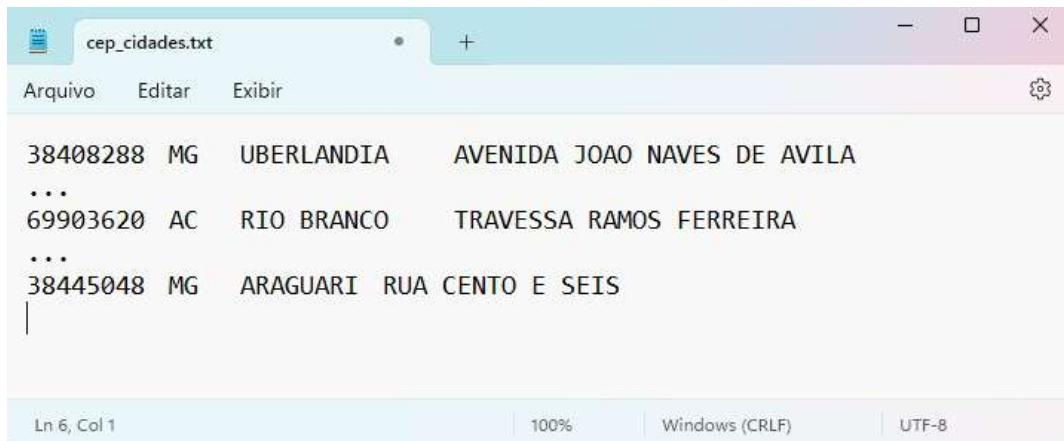
Esta prática tem por objetivo reforçar o conhecimento do(a) aluno(a) com relação aos algoritmos de pesquisa estudados em sala.

- Data da entrega: 15/03/2023 (até o final da aula)
- Grupo de até 3 (três) alunos(as)
- Linguagem de programação a ser usada: C
- O que deve ser entregue: Link para o repositório do código (Replit.com)
- **Essa prática vale 01 ponto.**

### 2. Descrição

Suponha um arquivo contendo entradas desordenadas para um conjunto de CEPs de cidades brasileiras seguindo o formato: CEP – sigla do estado – nome da cidade – endereço.

Exemplo do arquivo (.txt) contendo as entradas para o conjunto de CEPs:



The screenshot shows a Windows Notepad window with the title bar 'cep\_cidades.txt'. The menu bar includes 'Arquivo', 'Editar', and 'Exibir'. The window content displays the following text:

```
38408288 MG UBERLANDIA AVENIDA JOAO NAVES DE AVILA
...
69903620 AC RIO BRANCO TRAVESSA RAMOS FERREIRA
...
38445048 MG ARAGUARI RUA CENTO E SEIS
```

At the bottom of the window, status bars show 'Ln 6, Col 1', '100%', 'Windows (CRLF)', and 'UTF-8'.

O exercício prático consiste na implementação de um programa em linguagem C capaz de:

1. Ler o arquivo texto, e preencher um vetor contendo estruturas de dados baseada em um TAD.
2. Ordenar o vetor pelo CEP utilizando um método de ordenação de bom desempenho.
3. Permitir pesquisa por CEP baseada em busca binária, implementada de forma recursiva. Para cada CEP encontrado, deve-se exibir na tela a respectiva entrada completa: CEP, sigla do estado, nome da cidade e endereço.

# Árvores Binárias com Balanceamento: AVL

Profa. Maria Camila Nardini Barioni

[camila.barioni@ufu.br](mailto:camila.barioni@ufu.br)

Bloco B - sala 1B137

2º semestre de 2023

# Próximas aulas

| Data  | Dia da semana                  | Atividade                                                   |
|-------|--------------------------------|-------------------------------------------------------------|
| 02/04 | Terça                          | Árvores Rubro Negras                                        |
| 05/04 | Sexta                          | Aula disponibilizada para a finalização do trabalho prático |
| 07/04 | Domingo                        | Prazo final para a entrega do trabalho prático              |
| 09/04 | Terça                          | Aula de Exercícios sobre árvores AVL e Rubro Negras         |
| 12/04 | Sexta                          | Hash                                                        |
| 13/04 | Sábado                         | Apresentação dos Projetos                                   |
| 16/04 | Terça                          | Prova 2                                                     |
| 19/04 | Sexta                          | Exercícios lab Hash                                         |
| 23/04 | Terça                          | Prova de recuperação de aprendizagem.                       |
| 25/04 | Quinta<br>(reposição<br>sexta) | Vista final.                                                |

# Roteiro

## ◆ Definições

## ◆ Árvores AVL

- Fator de balanceamento
- Tipos de desbalanceamento
- Inserção
- Remoção

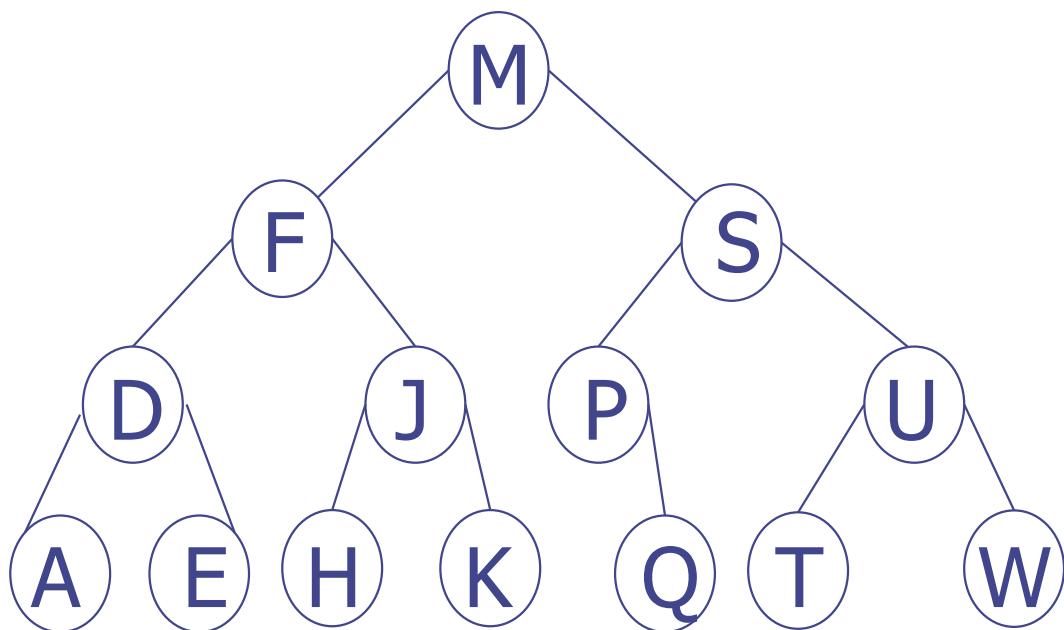
# Árvores Binárias Balanceadas

- ◆ Uma das vantagens no uso de árvores é que o processo de busca na mesma é muito mais rápido do que em listas encadeadas
- ◆ Entretanto, esse argumento nem sempre é válido, pois há uma dependência da estrutura da árvore:
  - Exemplo: Dadas duas AB de pesquisa inicialmente vazias, insira (e desenhe) os seguintes elementos
    - Árvore1: M, F, S, D, J, P, U, A, E, H, Q, T, W, K.
    - Árvore2: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z.
  - Qual é a pior?
  - Número de comparações necessárias?

# Árvores Binárias Balanceadas

## ◆ Exemplo:

- Árvore1: M, F, S, D, J, P, U, A, E, H, Q, T, W, K

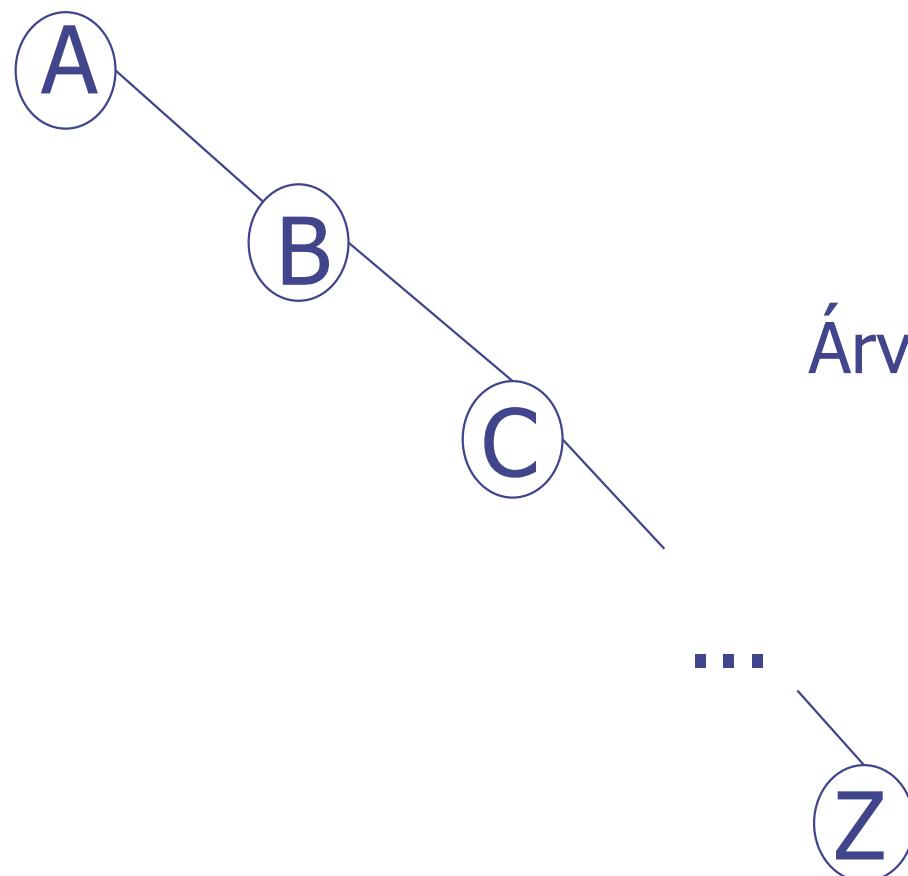


Árvore balanceada

# Árvores Binárias Balanceadas

## ◆ Exemplo:

- Árvore2: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z



Árvore degenerada

# Árvores Binárias Balanceadas

- ◆ A desvantagem de uma árvore binária de pesquisa é que a sua altura pode atingir  $N$  (onde  $N$  é o número de nós)
  - As operações de inserção, remoção e pesquisa podem ser  $O(N)$  no pior caso
- ◆ Portanto, é preciso uma árvore com altura menor
- ◆ Uma árvore binária com  $N$  nós tem altura de pelo menos  $O(\log N)$
- ◆ Então, o objetivo é manter a altura da árvore binária de pesquisa em  $O(\log N)$
- ◆ Árvores que exigem tal característica são chamadas árvores de pesquisa binárias平衡adas
  - Exemplos: árvores AVL e árvores rubro-negras

# Árvores AVL

- ◆ AVL = Adelson-Velskii e Landis
  - Inventada em 1962 por 2 matemáticos russos
- ◆ AVL é uma árvore de pesquisa
  - Balanceada
  - Altura das subárvore da esquerda e da direita de um nó difere em no máximo 1 (um)
  - As subárvore também são árvores AVL
- ◆ Portanto, as operações de inserção, remoção e pesquisa ocorrem em tempo  $O(\log N)$

# Árvores AVL

- ◆ Para definir o balanceamento é utilizado um cálculo específico:

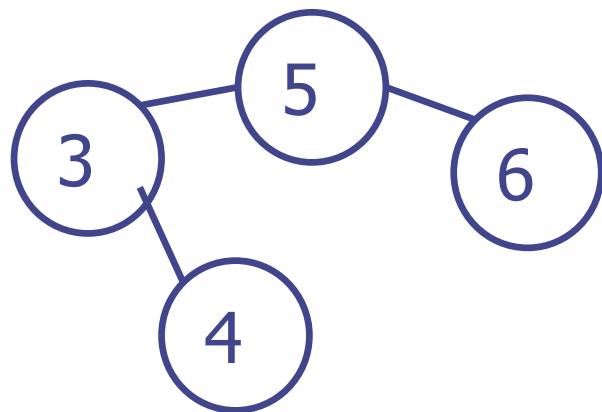
$$FB(v) = hE(v) - hD(v) \text{ onde:}$$

- $FB(v)$  é o fator de平衡amento para o nó  $v$
- $hE(v)$  é a altura da subárvore da esquerda
- $hD(v)$  é a altura da subárvore da direita

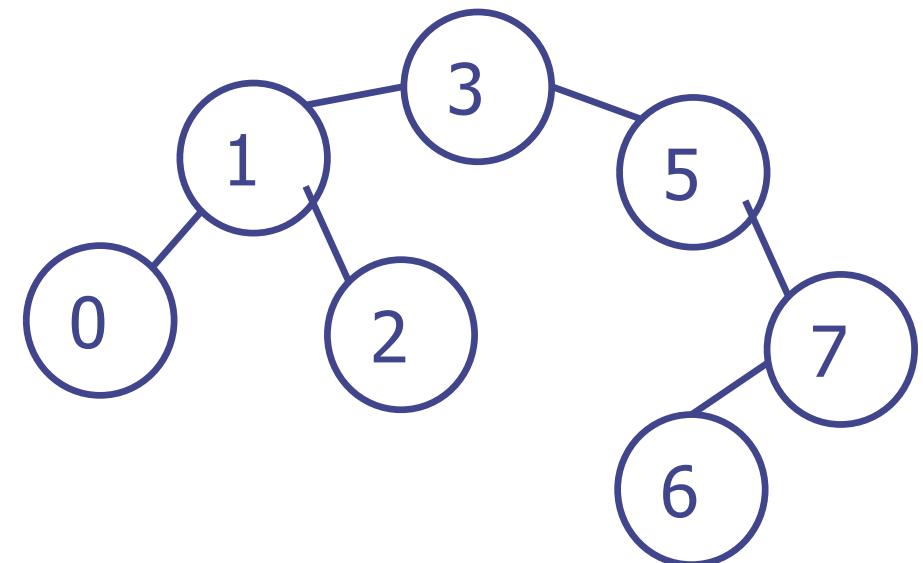
# Fator de balanceamento

- ◆ Todos os nós de uma árvore AVL tem um fator de balanceamento
  - $-1$  = subárvore direita é mais alta (a subárvore direita é mais alta que a esquerda em  $-1$ );
  - $0$  = subárvore esquerda e direita com mesma altura
  - $+1$  = subárvore esquerda é mais alta (a subárvore esquerda é mais alta que a direita em  $1$ )
- ◆ Se a árvore não estiver balanceada é necessário fazer seu rebalanceamento por meio de rotações
  - Rotação simples ou rotação dupla

# Fator de balanceamento - Exercício - análise do fator de balanceamento

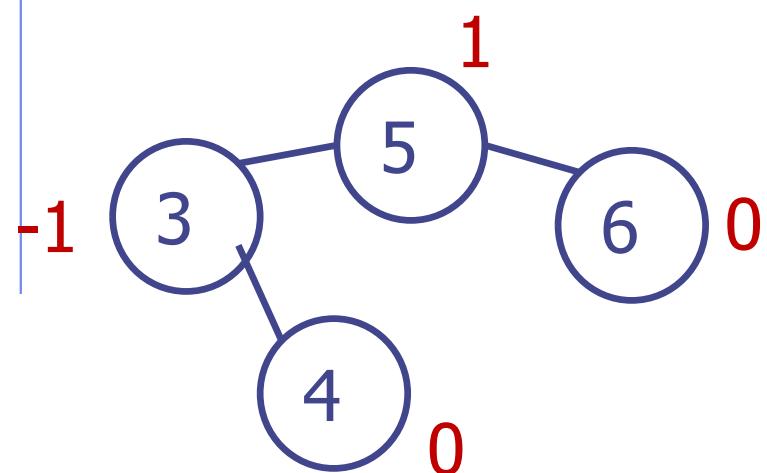


A

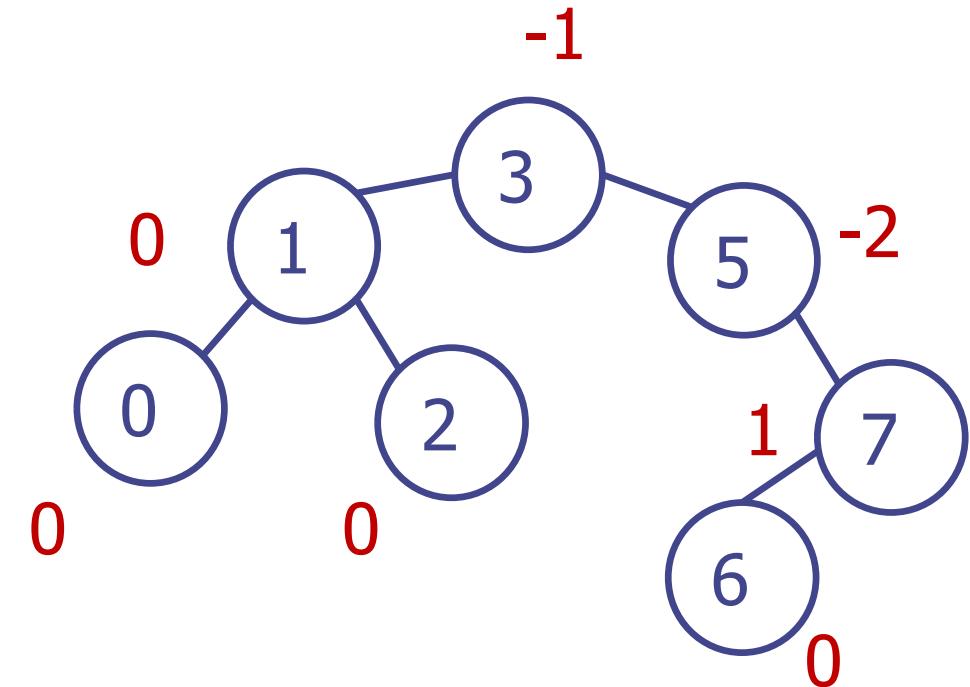


B

# Fator de balanceamento - Exercício - análise do fator de balanceamento



A



B

Não é AVL

# Causas de Desbalanceamentos

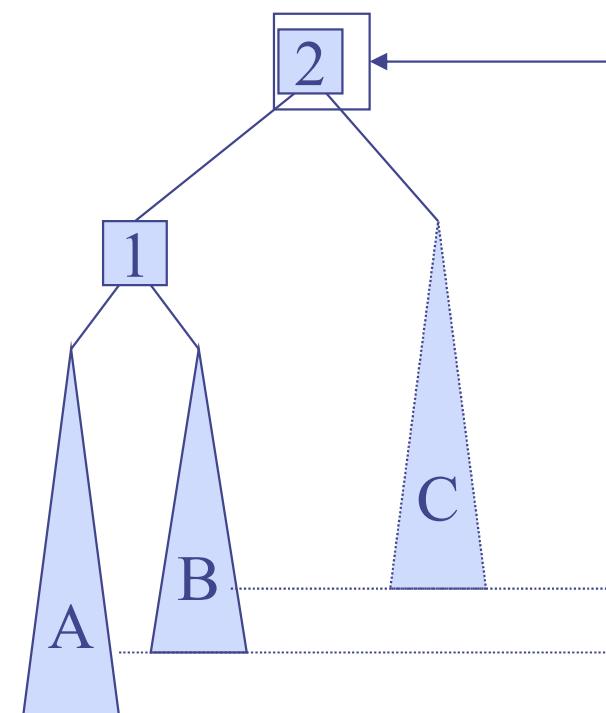
- ◆ Supondo que a árvore esteja balanceada
- ◆ Inserção
  - A menos que as chaves sempre apareçam na ordem correta, um desbalanceamento irá ocorrer
- ◆ Remoção
  - Remover um nó pode desbalancear a árvore
- ◆ Dois tipos de desbalanceamentos
  - Esquerdo-esquerdo (ou direito-direito)
  - Esquerdo-direito (ou direito-esquerdo)
  - (Esquerdos e direitos são simétricos)

# Condições envolvendo a altura das subárvore

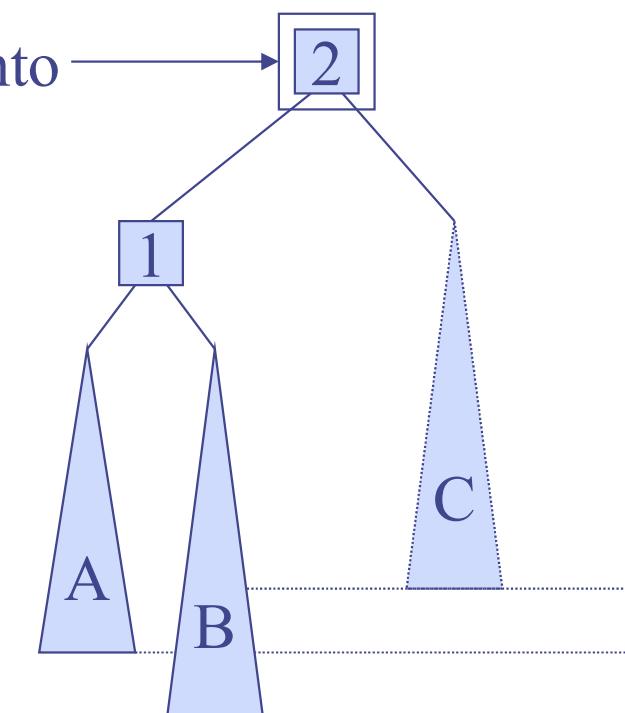
- ◆ Seja uma árvore T com subárvore E (esquerda) e D (direita), com alturas  $h_E$  e  $h_D$ , respectivamente. Se um novo elemento é inserido em E, três casos podem ocorrer:
  - Se  $h_E = h_D \rightarrow$  E e D passarão a ter alturas diferentes
  - Se  $h_E < h_D \rightarrow$  E e D passarão a ter alturas iguais
  - Se  $h_E > h_D \rightarrow$  E passará a ser maior que D em 2. O rebalanceamento é necessário

# Tipos de Desbalanceamentos

- Esquerdo-esquerdo



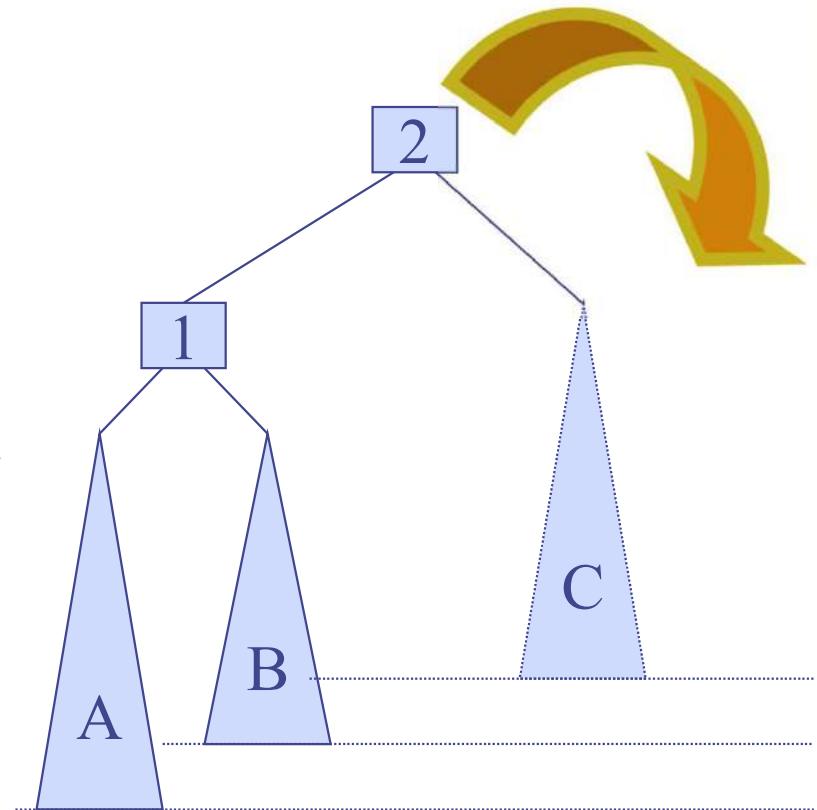
- Esquerdo-direito



# Desbalanceamento Esquerdo-Esquerdo

(e direito-direito, por simetria)

- ◆ Seja  $q$  o nó que foi inserido na subárvore esquerda de 2
- ◆  $B$  e  $C$  têm a mesma altura
- ◆  $A$  é um nível mais alta
- ◆ Então, fazer:
  - 1 a nova raiz
  - 2 sua subárvore da direita
  - $B$  e  $C$  subárvores de 2

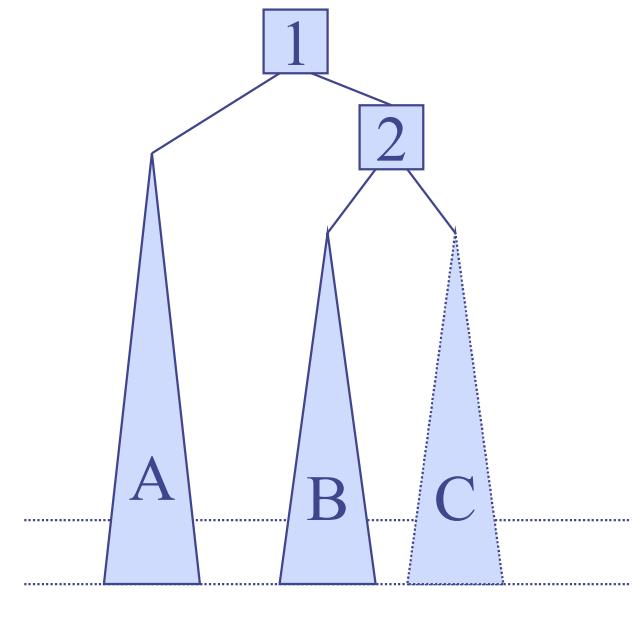


Rotação simples  
à direita (ou  
Rotação EE)

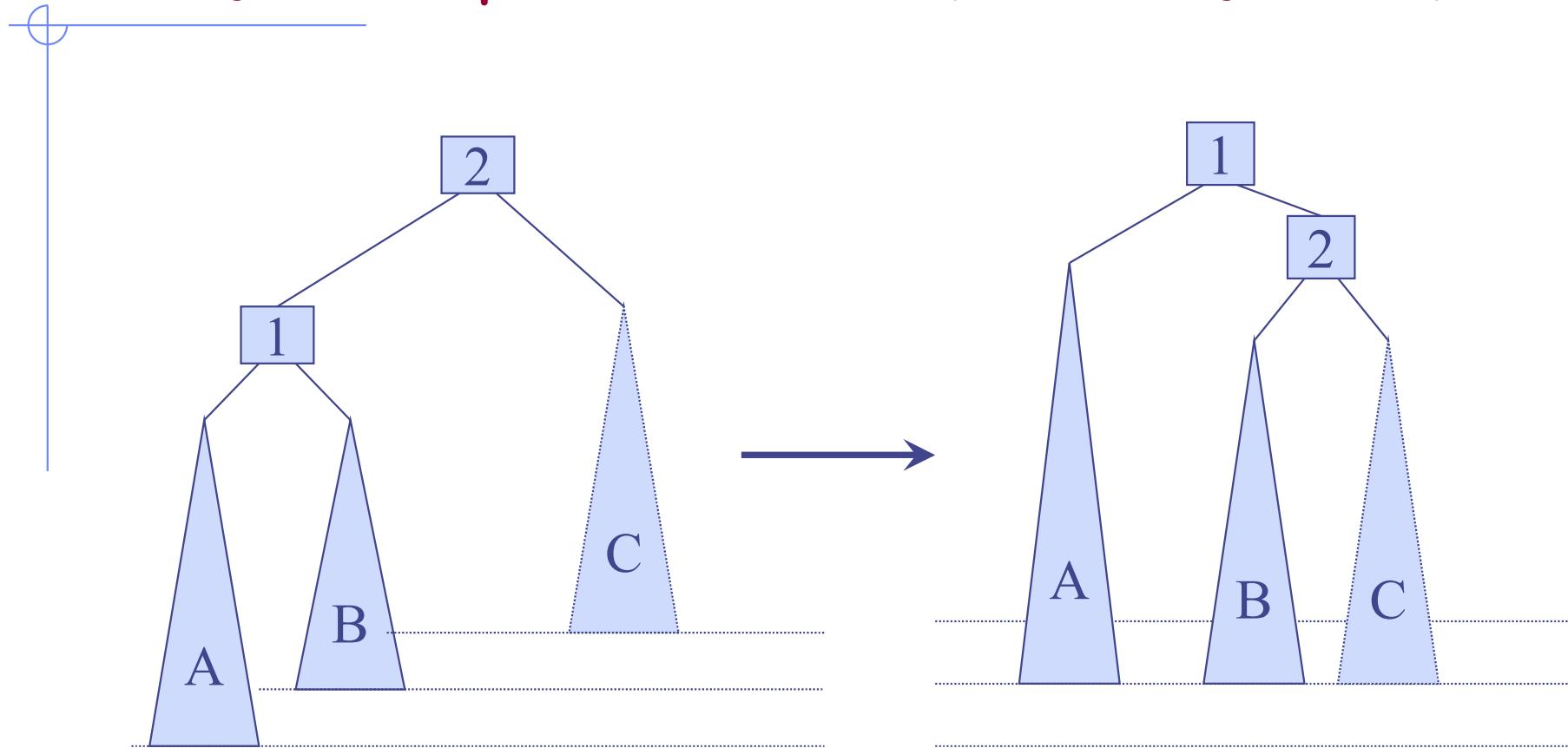
# Desbalanceamento Esquerdo-Esquerdo

(e direito-direito, por simetria)

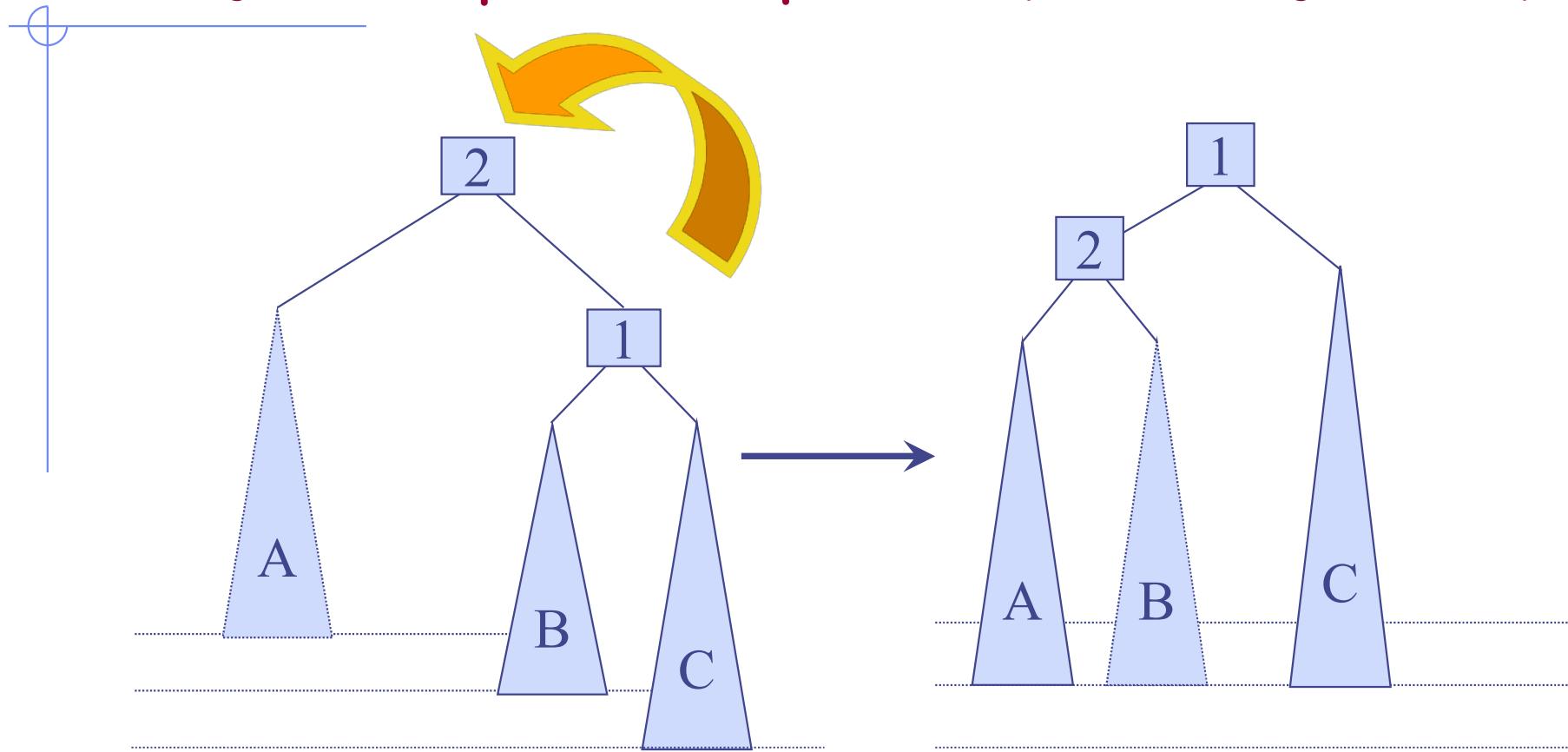
- ◆ B e C têm a mesma altura
- ◆ A é um nível mais alta
- ◆ Então, fazer:
  - 1 a nova raiz
  - 2 sua subárvore da direita
  - B e C subárvore de 2
- ◆ Resultado
  - Uma árvore AVL, ou seja, estrutura balanceada



## Rotação Simples à direita (ou Rotação EE)



## Rotação Simples à esquerda (ou Rotação DD)

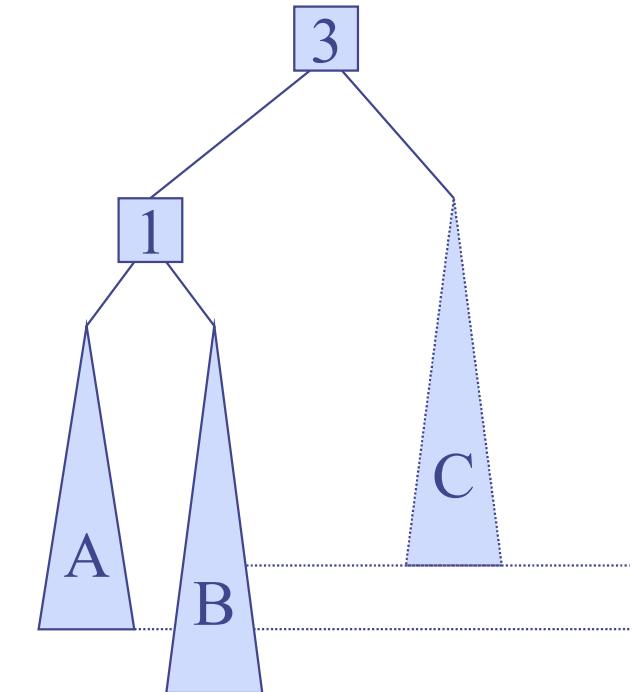


Desbalanceamento  
direito-direito

# Desbalanceamento Esquerdo-Direito

(e direito-esquerdo, por simetria)

- ◆ Não é possível usar a primeira estratégia de balanceamento esquerdo-esquerdo, porque agora é a subárvore do meio que é profunda demais
- ◆ Então é melhor observar mais de perto o que existe dentro de B



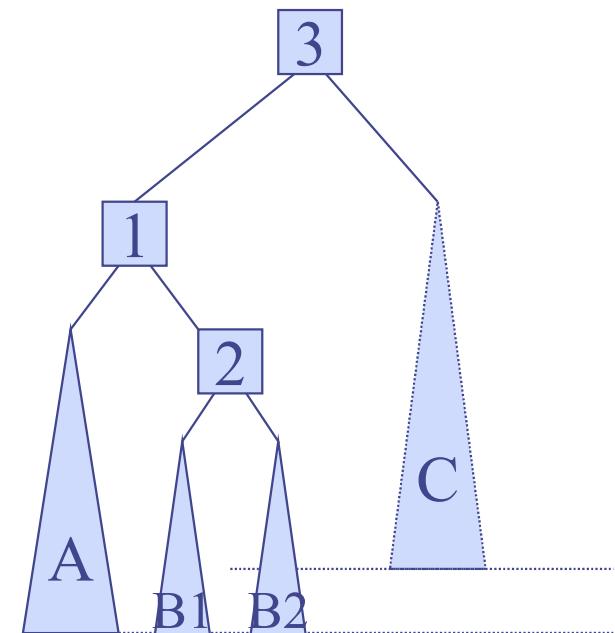
1 a nova raiz  
3 sua sub-árvore da direita  
B e C sub-árvore de 3

**Não**

# Desbalanceamento Esquerdo-Direito

(e direito-esquerdo, por simetria)

- ◆ B tem duas subárvore, cada uma com pelo menos um nó
- ◆ Não se sabe qual delas está alta demais
- ◆ Então, podemos determinar que ambas estão 0.5 nível abaixo da subárvore A

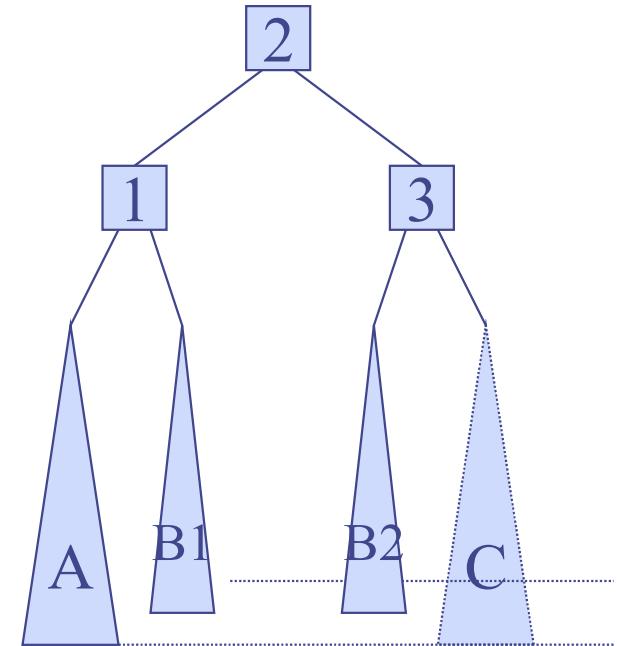


Rotação dupla à direita (ou Rotação ED)

# Desbalanceamento Esquerdo-Direito

(e direito-esquerdo, por simetria)

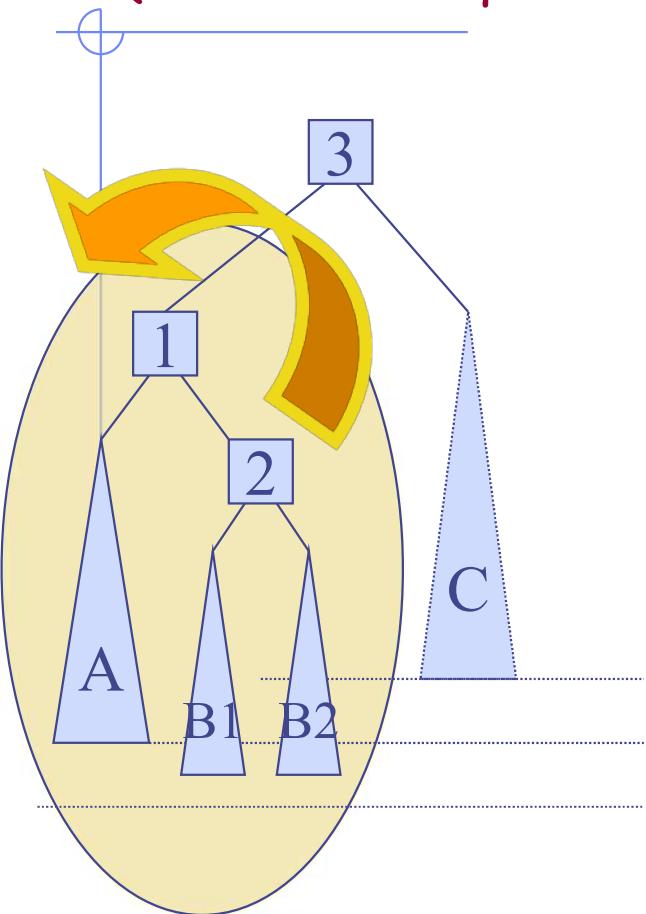
- ◆ Como nem 3 nem 1 servem como raiz, então fazer 2 a raiz da árvore
- ◆ Rearranciar as subárvoreas na ordem correta
- ◆ Independente da altura de B1 e B2, a árvore será uma AVL válida



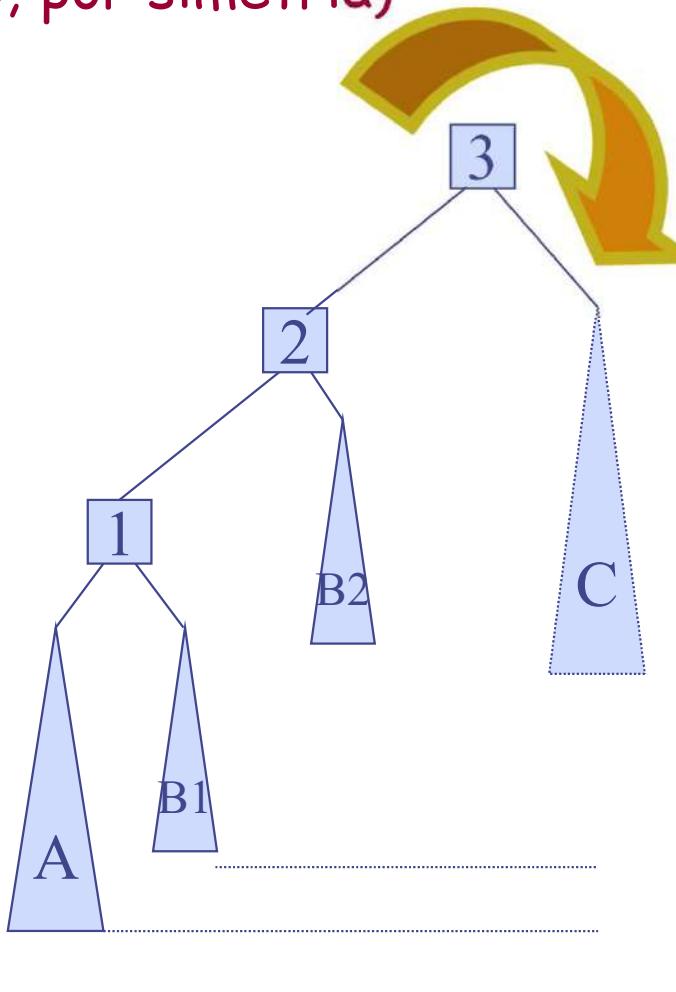
- Uma rotação simples à esquerda
- Uma rotação simples à direita

# Desbalanceamento Esquerdo-Direito

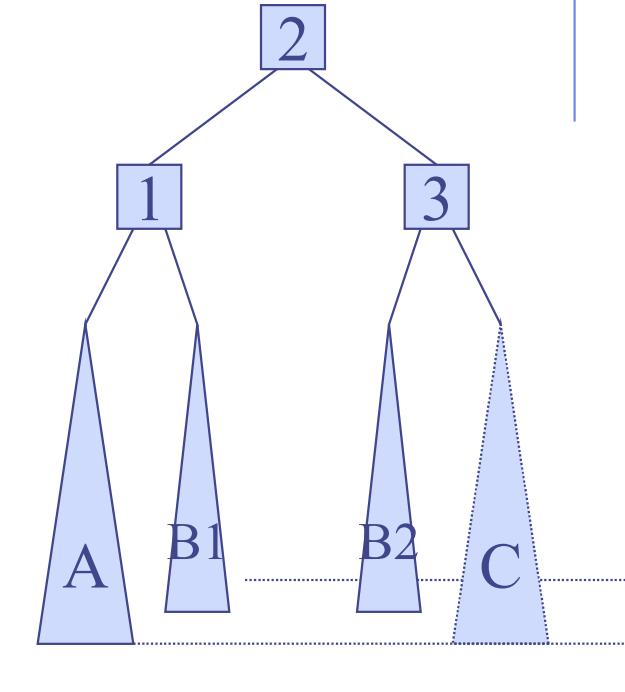
(e direito-esquerdo, por simetria)



Uma rotação simples à esquerda



Uma rotação simples à direita



# Inserção - Passos

- ◆ Para inserir um novo valor  $v$  na árvore, devem ser considerados os casos/ações:
  - Se a raiz é igual a NULL: criar o nó e inserir  $v$ .
  - Se  $v$  é menor do que a raiz: caminhar para a subárvore esquerda e reiniciar a inserção.
  - Se  $v$  é maior que a raiz: caminhar para a subárvore direita e reiniciar a inserção.
- ◆ A aplicação recursiva do método permite alcançar uma posição nula na árvore, local em que o novo valor  $v$  será inserido.

# Inserção - Passos

- ◆ Primeiro, inserir a nova chave como uma nova folha, igual a uma árvore binária de pesquisa normal
- ◆ Após a inserção de um novo valor na árvore AVL, deve-se seguir o caminho da nova folha até a raiz
  - Para cada nó  $x$  encontrado, verificar se o fator de balanceamento de  $x$  é no máximo 1
  - Se sim, prosseguir para o nó pai
  - Se não, rebalancear a árvore usando uma rotação simples ou dupla
- ◆ Um único rebalanceamento é suficiente
  - Ou seja, feito um rebalanceamento em  $x$ , não é necessário seguir para os ancestrais de  $x$

# Inserção

- ◆ Assumir  $x$  como sendo o nó onde o fator de desbalanceamento é maior que 1
- ◆ Assumir que a altura de  $x$  é  $h+2$
- ◆ Existem 4 casos para a inserção
  - Altura de esquerda( $x$ ) é  $h+2$  (altura de direita( $x$ ) é  $h$ )
    - ◆ Altura de esquerda(esquerda( $x$ )) é  $h+1 \Rightarrow$  rotação simples da subárvore da esquerda
    - ◆ Altura de direita(esquerda( $x$ )) é  $h+1 \Rightarrow$  rotação dupla da subárvore da esquerda
  - Altura de direita( $x$ ) é  $h+2$  (altura de esquerda( $x$ ) é  $h$ )
    - ◆ Altura de direita(direita( $x$ )) é  $h+1 \Rightarrow$  rotação simples da subárvore da direita
    - ◆ Altura de esquerda(direita( $x$ )) é  $h+1 \Rightarrow$  rotação dupla da subárvore da direita

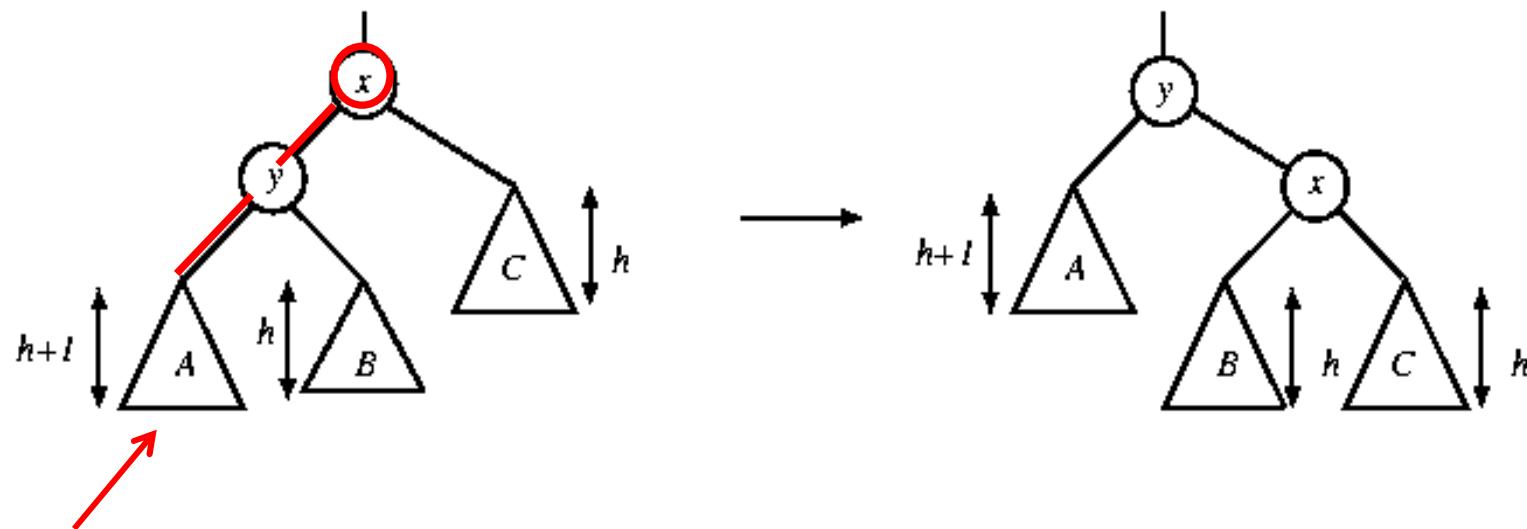
# Análise do código em C para inserção

- ◆ O fator de balanceamento do nó deve ser amazenado explicitamente:

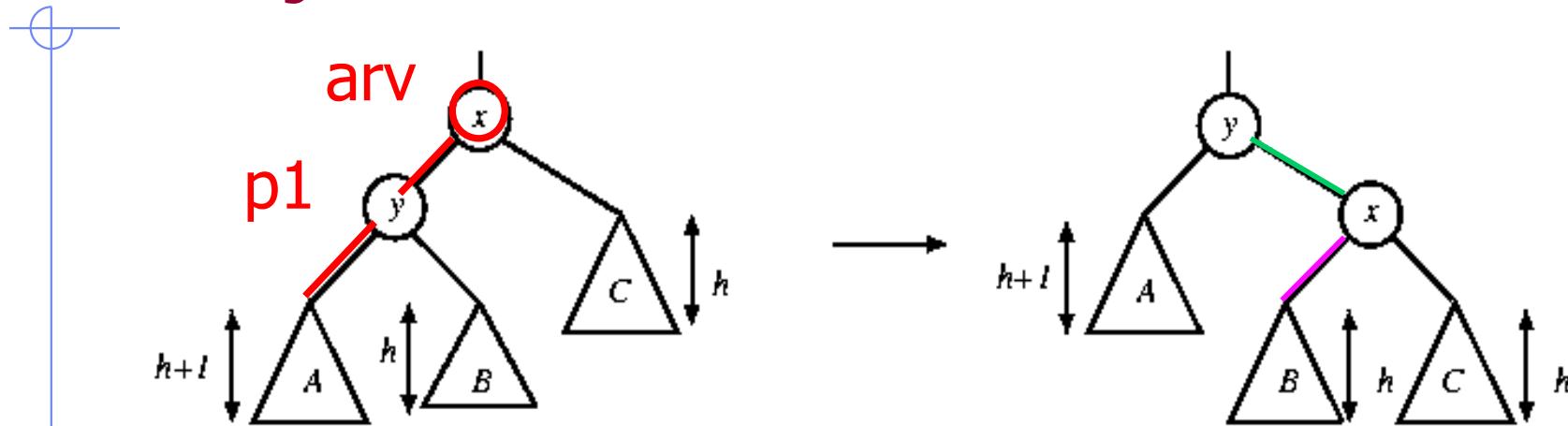
```
struct NO {
 int info;
 int fator;
 struct NO *fesq, *fdir;
};
```

# Inserção - Rotação Simples à direita (Rotação EE)

- ◆ A nova chave é inserida na subárvore A
- ◆ A propriedade da AVL é violada em x
  - Altura de esquerda(x) é  $h+2$
  - Altura de direita(x) é  $h$



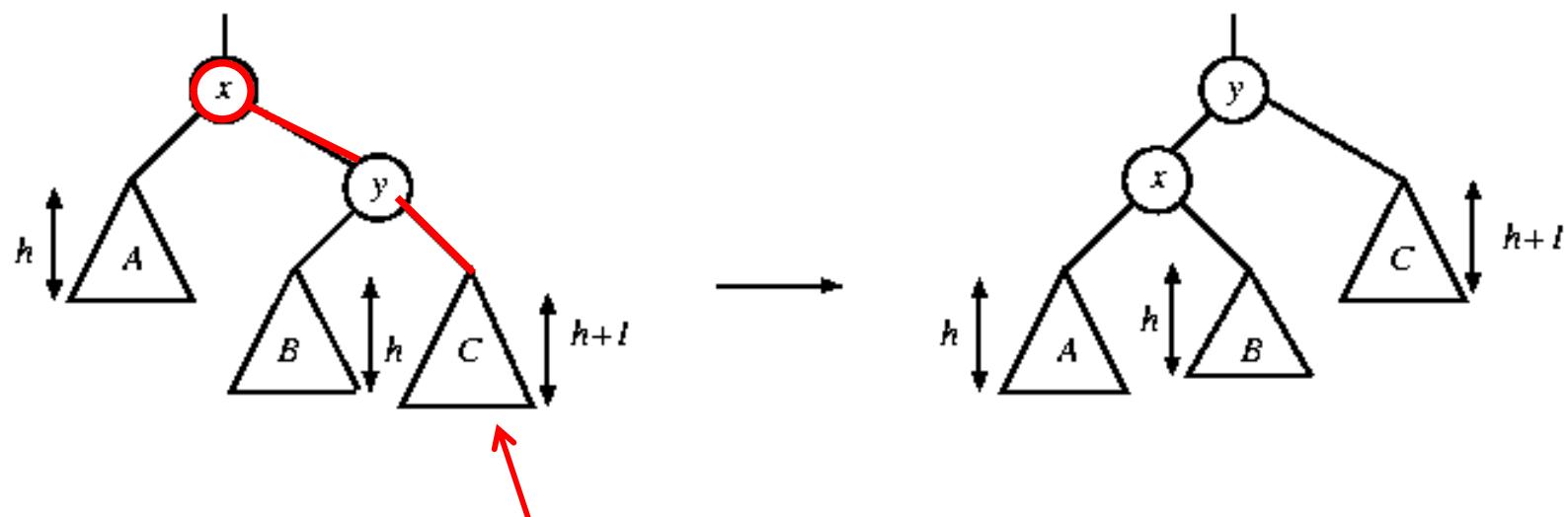
# Análise do código em C para inserção Rotação EE



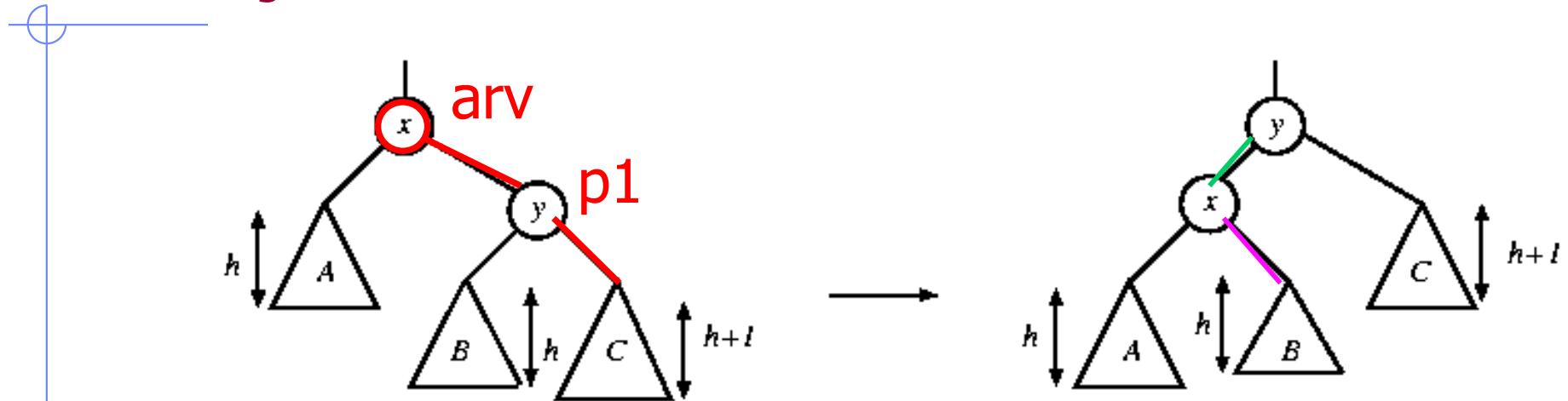
```
p1 = (*arv)->fesq;
(*arv)->fesq = p1->fdir;
p1->fdir = (*arv);
(*arv) = p1;
```

# Inserção - Rotação Simples à esquerda (Rotação DD)

- ◆ A nova chave é inserida na subárvore C
- ◆ A propriedade da AVL é violada em x
  - Altura de esquerda(x) é h
  - Altura de direita(x) é h+2



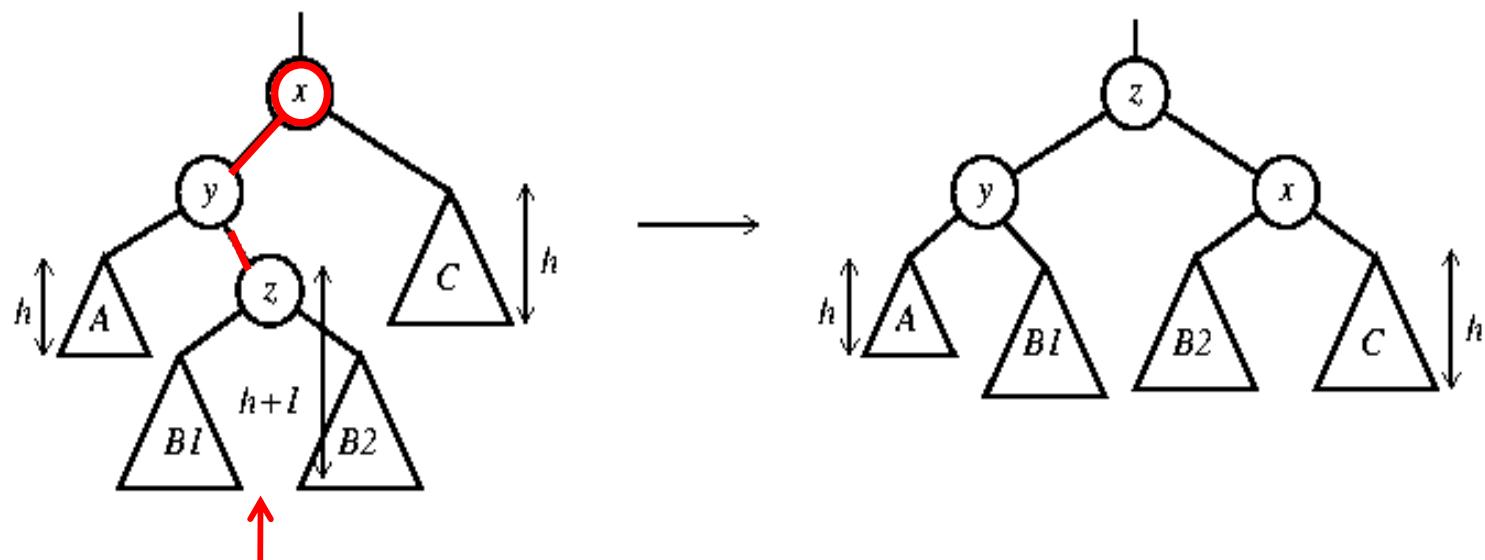
# Análise do código em C para inserção Rotação DD



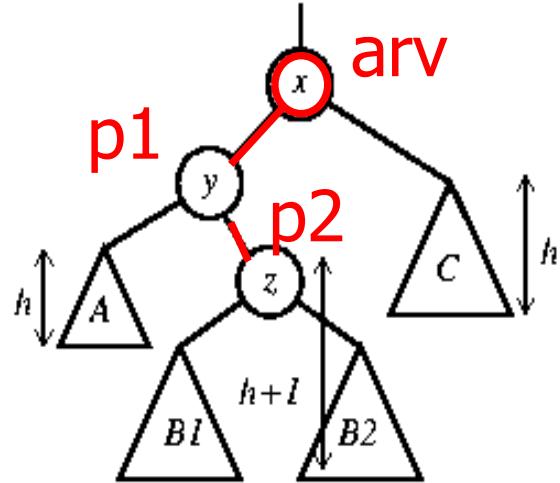
```
p1 = (*arv)->fdir;
(*arv)->fdir = p1->fesq;
p1->fesq = (*arv);
(*arv) = p1;
```

# Inserção - Rotação Dupla à direita (Rotação ED)

- ◆ A nova chave é inserida na subárvore  $B1$  ou  $B2$
- ◆ A propriedade da AVL é violada em  $x$ 
  - Altura de esquerda( $x$ ) é  $h+2$
  - Altura de direita( $x$ ) é  $h$

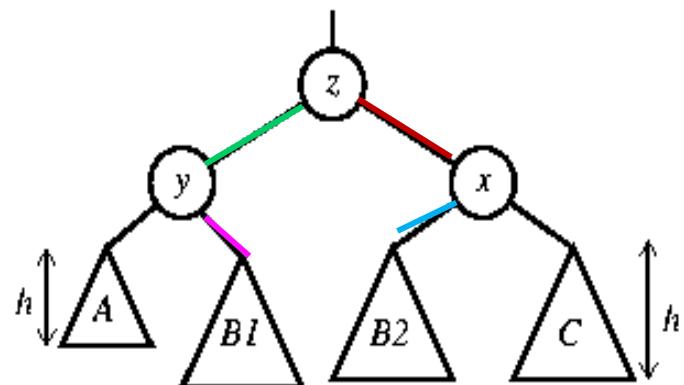


# Análise do código em C para inserção Rotação ED



```
p1 = (*arv)->fesq;
p2 = p1->fdir;
p1->fdir = p2->fesq; p2->fesq = p1;
(*arv)->fesq = p2->fdir; p2->fdir = (*arv);
(*arv) = p2;
```

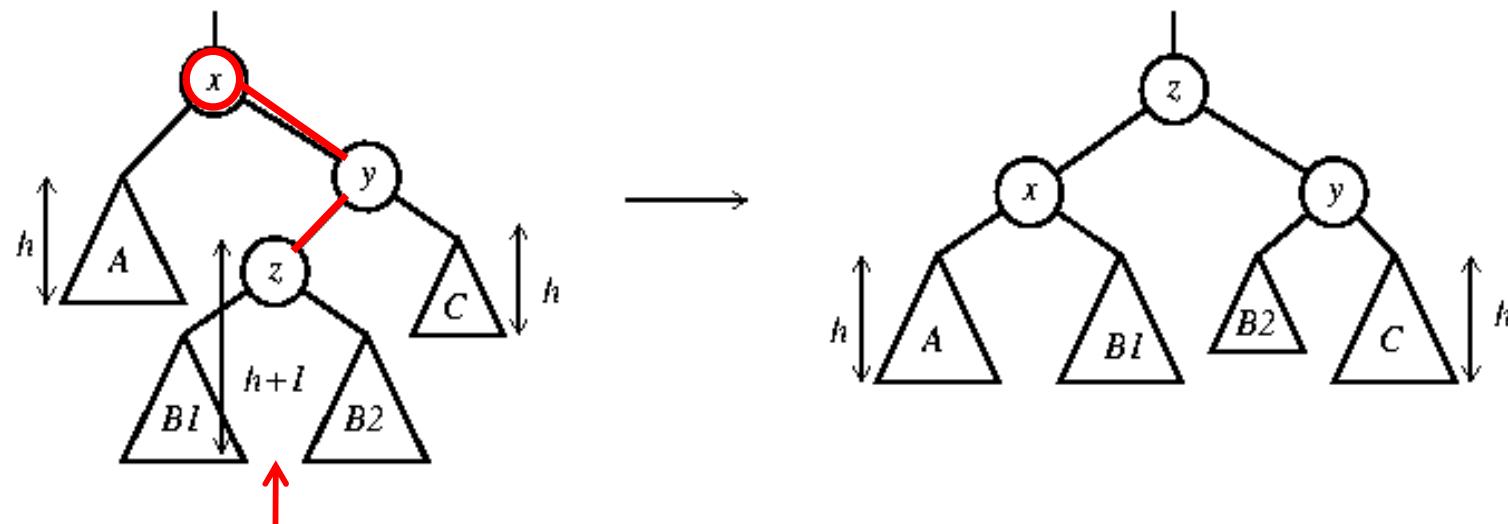
ou



```
RotacaoEsquerda(&(*arv)->esq);
RotacaoDireita(arv);
```

# Inserção - Rotação Dupla à esquerda (Rotação DE)

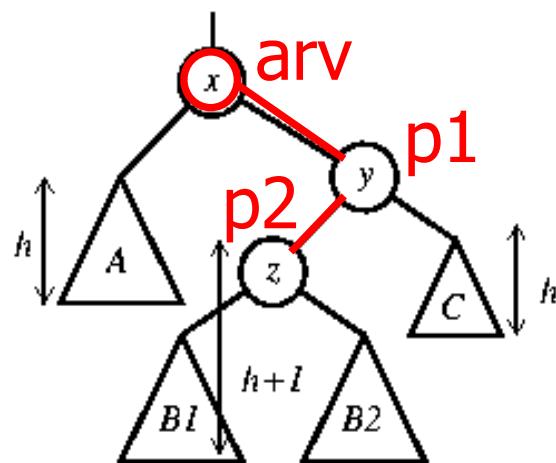
- ◆ A nova chave é inserida na subárvore B1 ou B2
- ◆ A propriedade da AVL é violada em x
  - Altura de direita(x) é  $h+2$
  - Altura de esquerda(x) é  $h$



# Análise do código em C para inserção

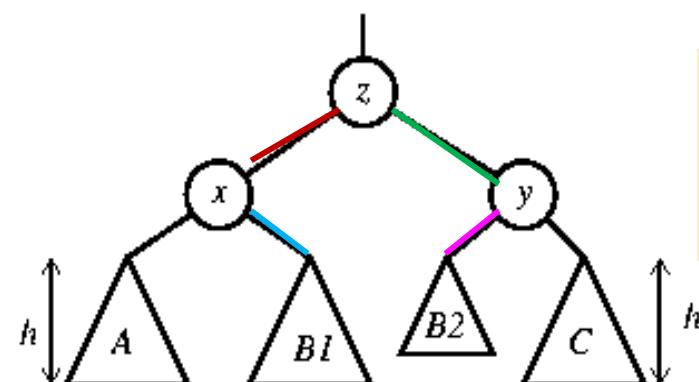
## Rotação DE

←



```
p1 = (*arv)->fdir;
p2 = p1->fesq;
p1->fesq = p2->fdir; p2->fdir = p1;
(*arv)->fdir = p2->fesq;
p2->fesq = (*arv);
(*arv) = p2;
```

ou



```
RotacaoDireita(&(*arv)->dir);
RotacaoEsquerda(arv);
```

# Exemplo de Inserção

Inserir 3,2,1,4,5,6,7, 16,15,14

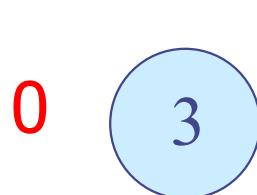


Fig 1

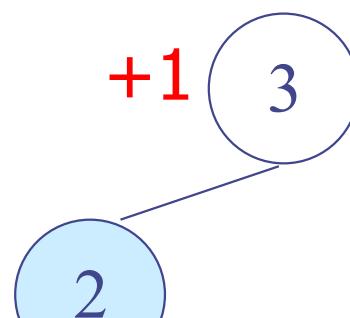


Fig 2

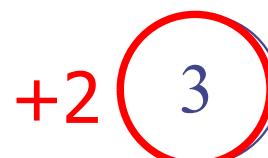


Fig 4

Rotação simples

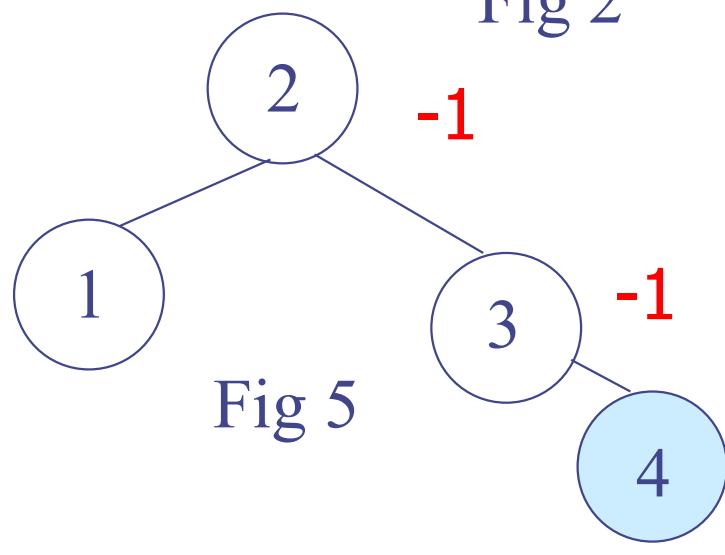


Fig 5



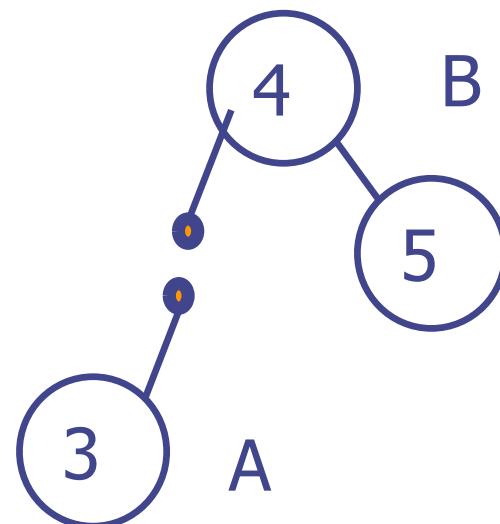
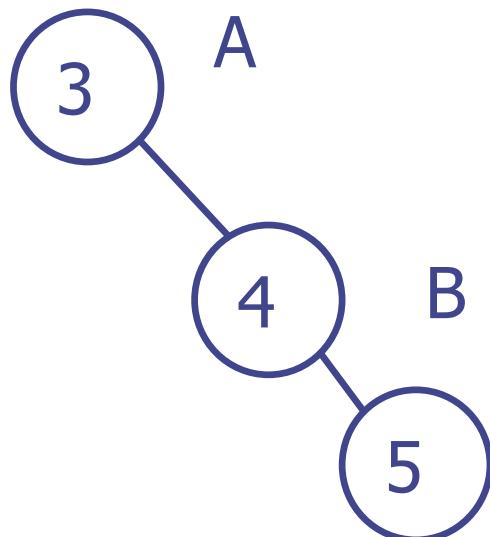
Fig 6

Rotação simples

# Exemplo de Inserção - Análise

Inserir 3,2,1,4,5,6,7, 16,15,14

**A->dir = B->esq;**  
**B->esq = A;**



# Exemplo de Inserção - Análise

Inserir 3,2,1,4,5,6,7, 16,15,14

```
A->dir = B->esq;
B->esq = A;
```

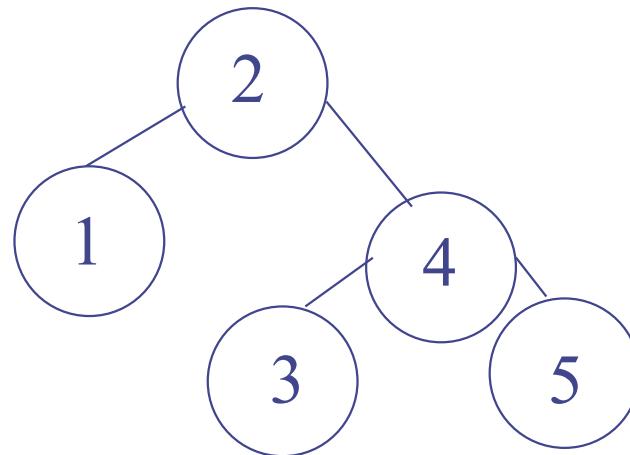


Fig 7

Animação

<https://cmps-people.ok.ubc.ca/y lucet/DS/AVLtree.html>

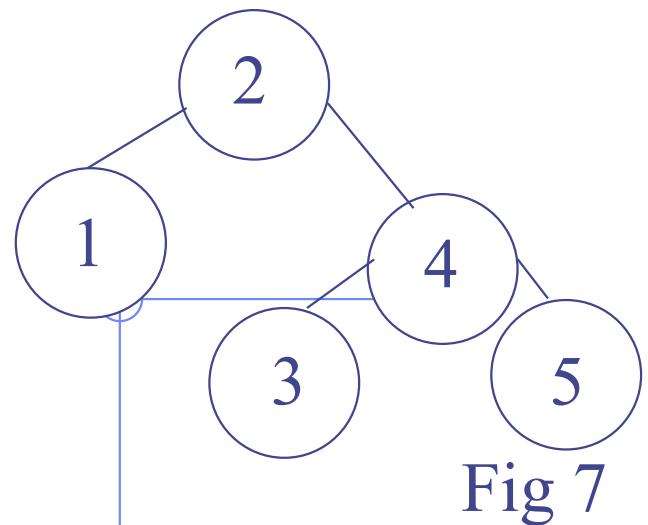
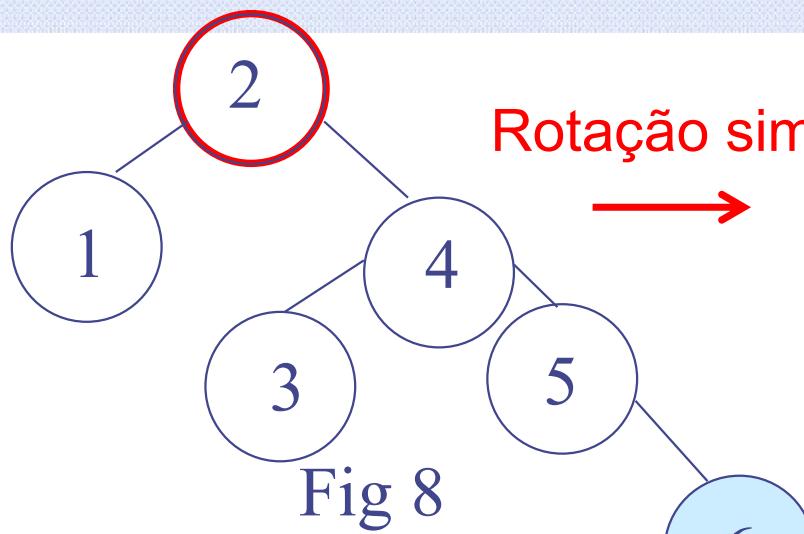


Fig 7



Rotação simples



Fig 8

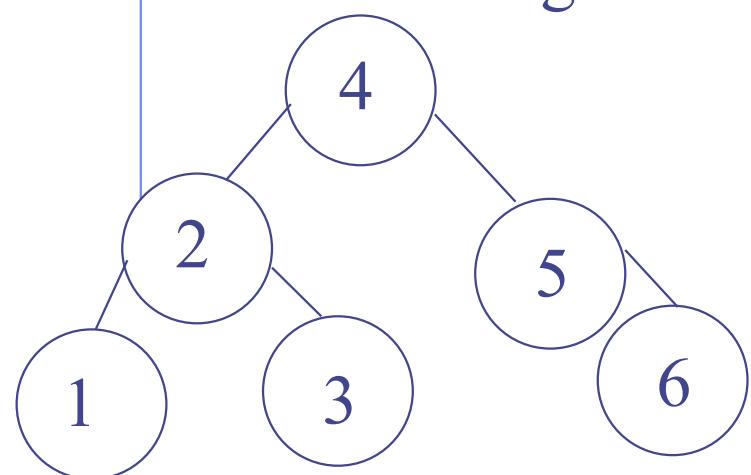


Fig 9

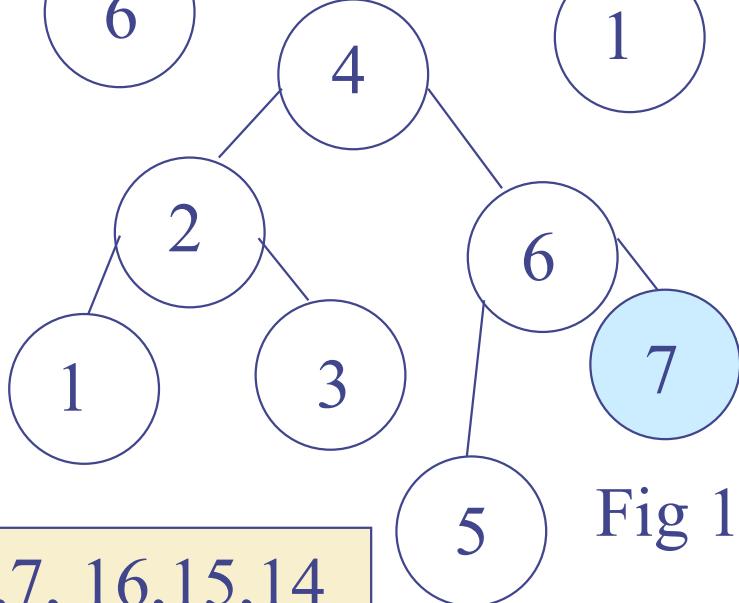


Fig 11

Rotação simples



Fig 10

Inserir 3,2,1,4,5,6,7, 16,15,14

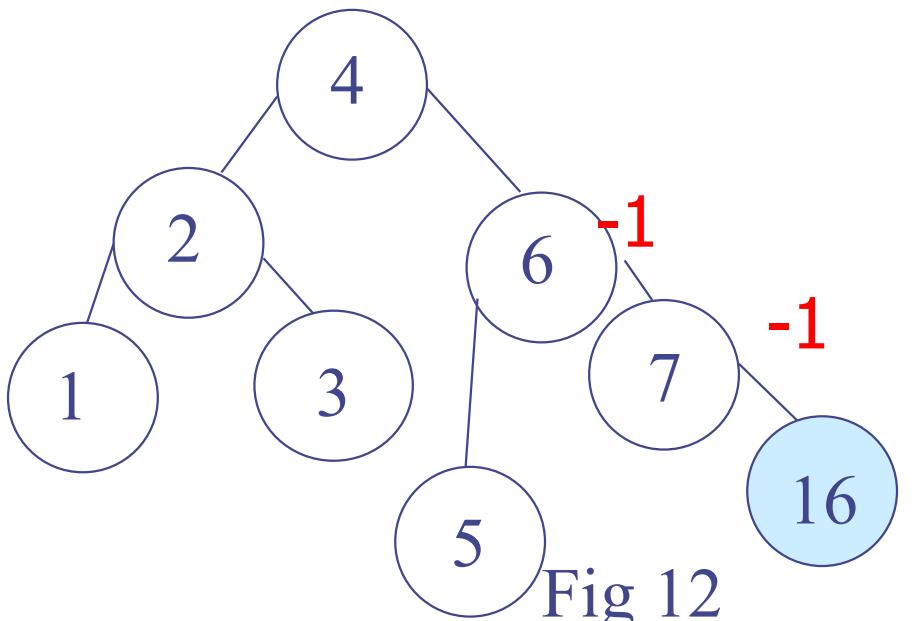


Fig 12

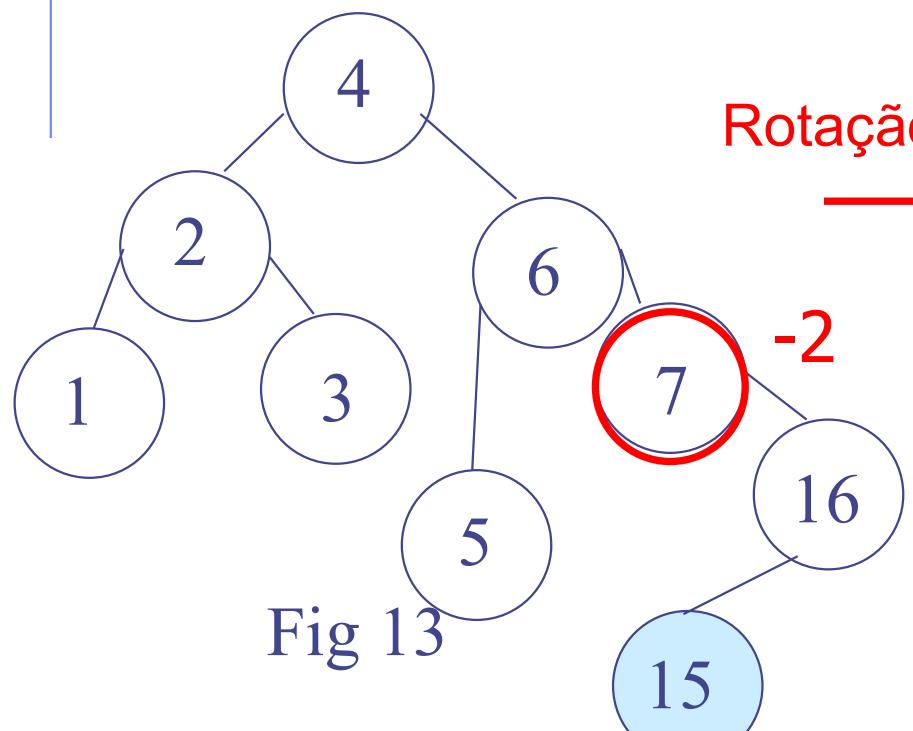


Fig 13

Rotação dupla

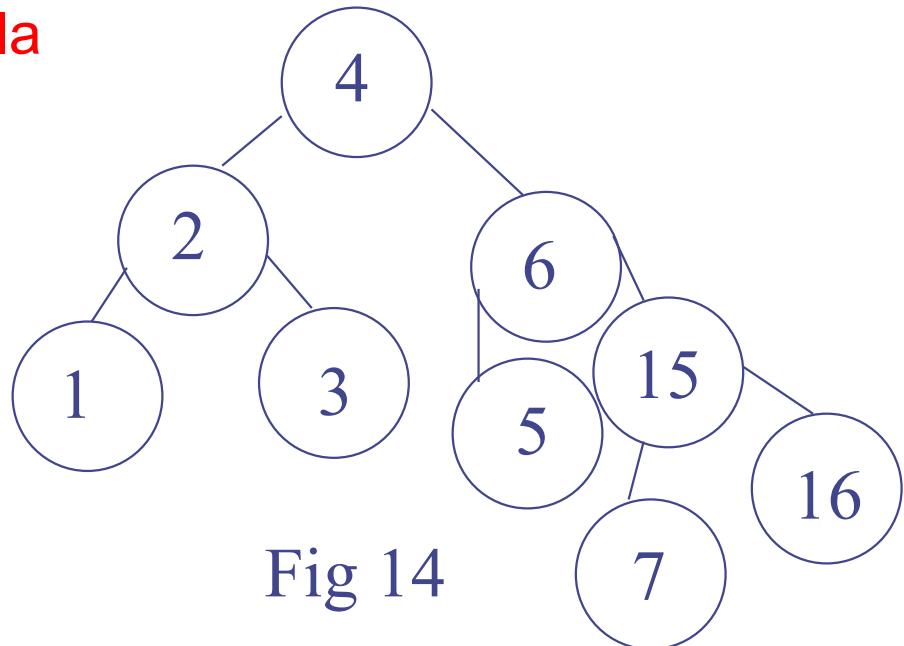


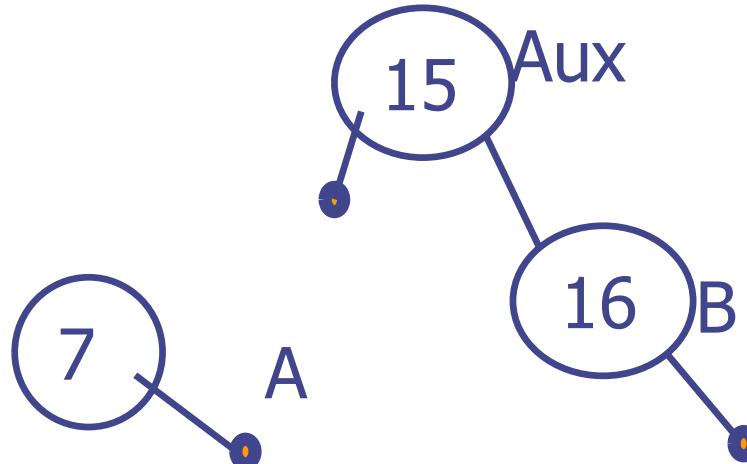
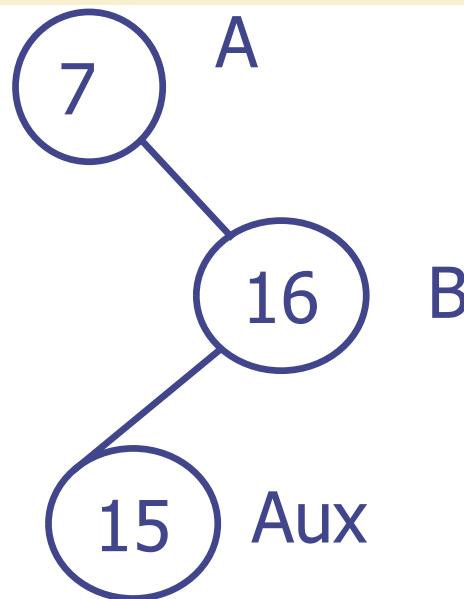
Fig 14

Inserir 3,2,1,4,5,6,7, 16,15,14

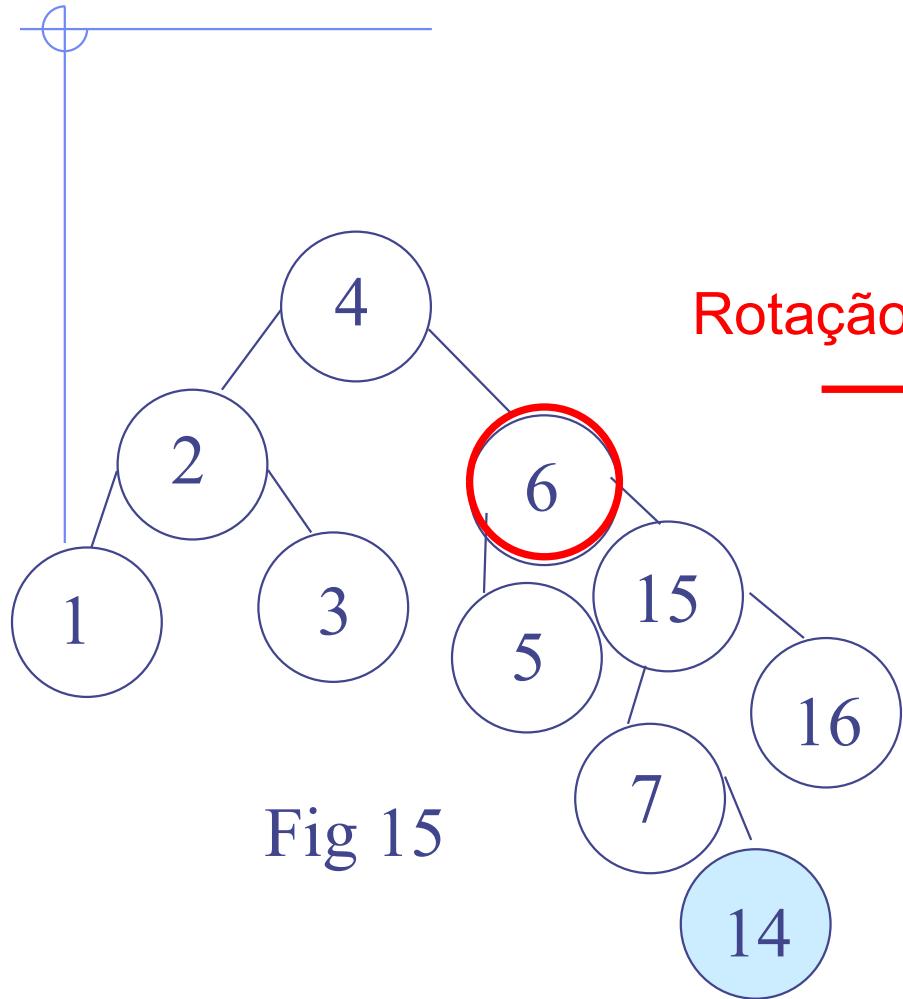
# Exemplo de Inserção - Análise

Inserir 3,2,1,4,5,6,7, 16,15,14

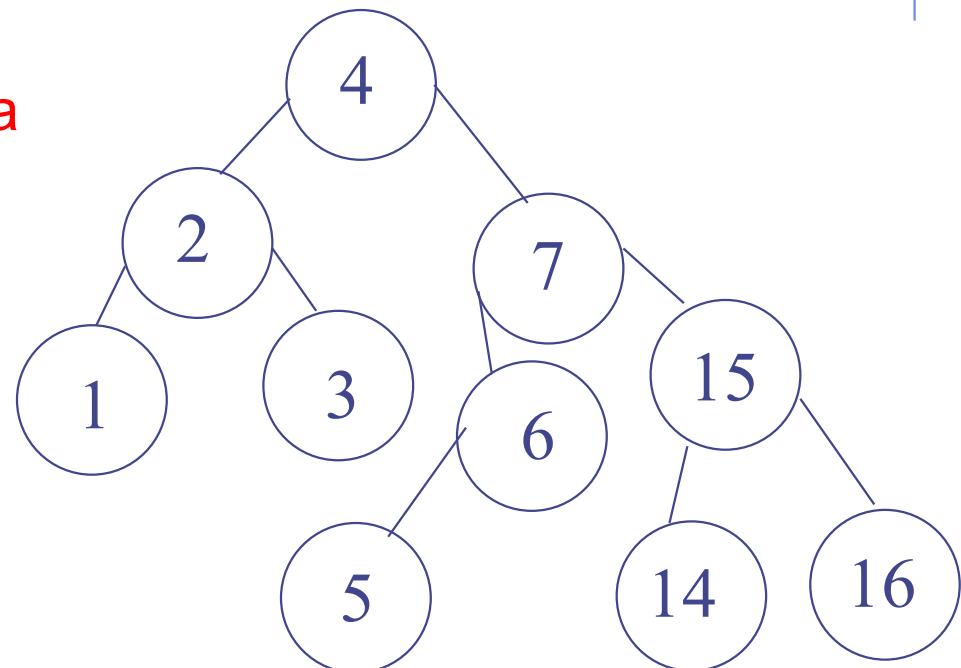
```
Aux = B->esq;
// rotação simples à direita (B - Aux)
B->esq = Aux->dir;
Aux->dir = B;
// rotação simples à esquerda (A - Aux)
A->dir = Aux->esq;
Aux->esq = A;
```



Inserir 3,2,1,4,5,6,7, 16,15,14



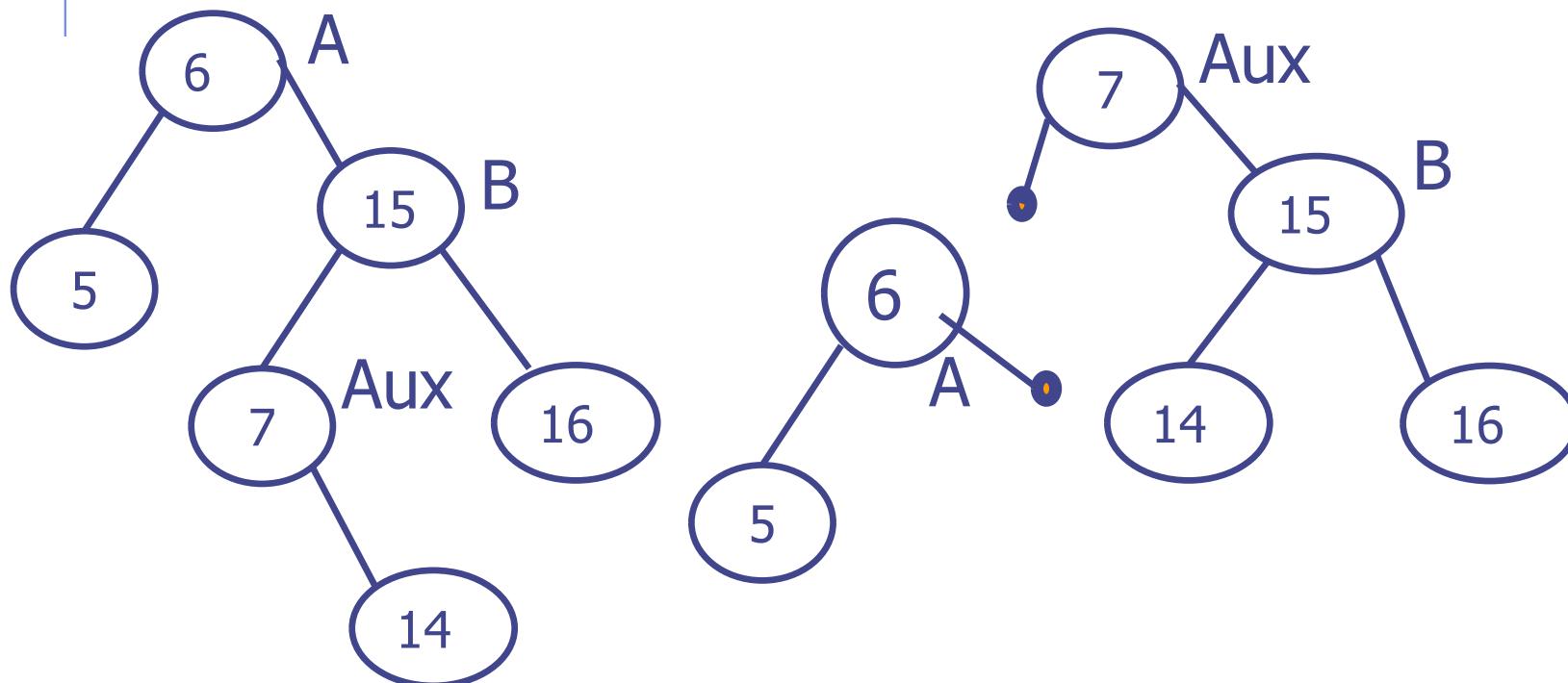
Rotação dupla  
→



# Exemplo de Inserção - Análise

Inserir 3,2,1,4,5,6,7, 16,15,14

```
Aux = B->esq;
// rotação simples à direita (B - Aux)
B->esq = Aux->dir;
Aux->dir = B;
// rotação simples à esquerda (A - Aux)
A->dir = Aux->esq;
Aux->esq = A;
```



# Remoção - Passos

- ◆ A remoção em árvores AVL é similar à remoção em uma árvore binária de busca. Entretanto, pode ser necessário o rebalanceamento da parte afetada pela remoção.
- ◆ Na remoção de um valor v, encontrado no nó p na árvore, devem ser considerados os casos/ações:
- ◆ Se p possui nenhum filho: remover p, rebalancear nós antecessores afetados, se necessário.
  - Se p possui um filho: subir este filho, remover p, rebalancear antecessores, se necessário.
  - Se p possui dois filhos:
    - Encontrar o menor na subárvore à direita e substituir seu valor em p;
    - Executar a remoção do menor na subárvore à direita de p (este nó possuirá um ou nenhum filho);
    - Após a remoção do menor à direita, rebalancear p, caso necessário.

# Remoção - Passos

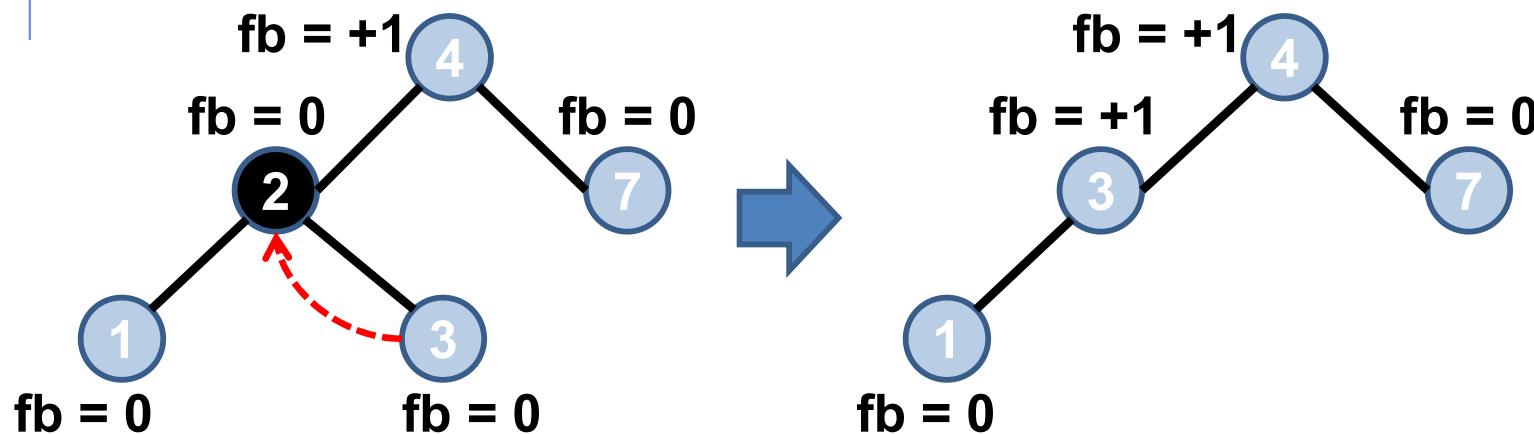
## ◆ Uma vez removido o nó

- Devemos voltar pelo caminho percorrido e calcular o fator de balanceamento de cada um dos nós visitados
- Aplicar a rotação necessária para restabelecer o balanceamento da árvore se o fator de balanceamento for +2 ou -2
  - ◆ Remover um nó da subárvore direita equivale a inserir um nó na subárvore esquerda

# Remoção - Exemplo

## ◆ Passo a passo

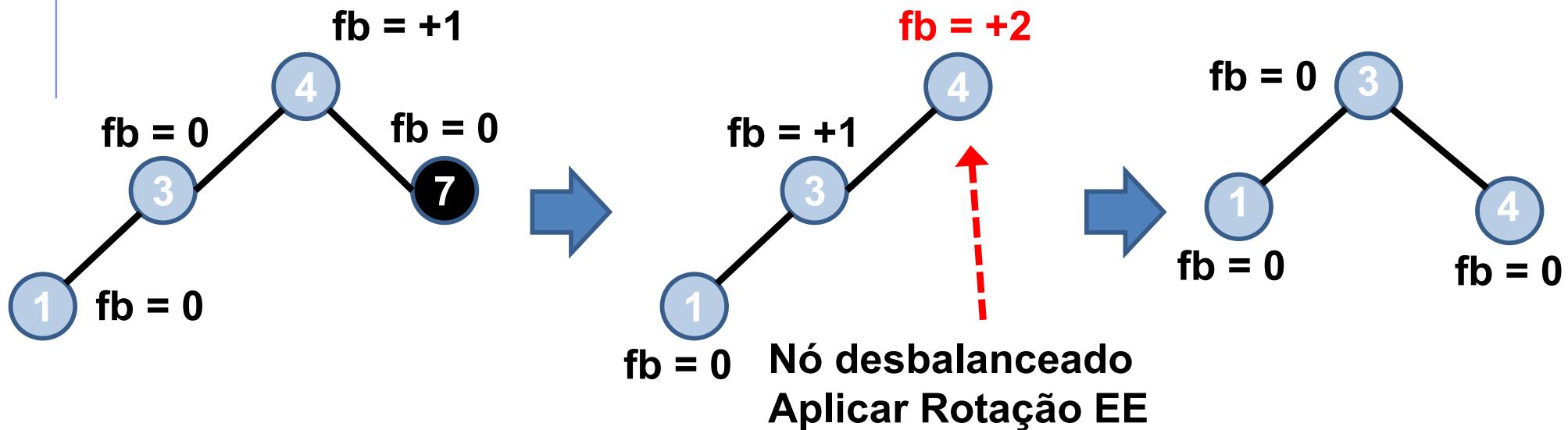
Remove valor: 2



# Remoção - Exemplo

## ◆ Passo a passo

Remove valor: 7



# Tempos de Execução para as operações da AVL

- ◆ Pesquisa é  $O(\log N)$ 
  - Altura de árvore é  $O(\log N)$  e não necessita rebalanceamento
- ◆ Inserção é  $O(\log N)$ 
  - Pesquisa inicial é  $O(\log N)$
  - Rebalanceamento é  $O(\log N)$  , devido à detecção de desbalanceamento
- ◆ Remoção é  $O(\log N)$ 
  - Pesquisa inicial é  $O(\log N)$
  - Rebalanceamento é  $O(\log N)$  , devido à detecção de desbalanceamento

# Exercícios

- ◆ Sejam as sequências de números inteiros abaixo. Para cada uma, crie uma árvore AVL fazendo a inserção de cada elemento e as rotações necessárias. Faça no caderno, sem utilizar computador.
  - (p1): 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
  - (p2): 5, 12, 7, 2, 15, 8, 28, 29, 45, 1, 56
- ◆ Considerando a árvore p2 criada na questão anterior, remova passo-a-passo os elementos 2 e 5

# Referência e Material extra

- ◆ Backes, A. Programação Descomplicada - Estruturas de Dados. Vídeo-aulas 79 a 84:
  - <https://programacaodescomplicada.wordpress.com/indice/estrutura-de-dados/>
- ◆ Animação AVL:
  - <https://cmps-people.ok.ubc.ca/y lucet/DS/AVLtree.html>
  - <https://visualgo.net>

# MC202 - Estruturas de Dados

## Árvores Rubro-Negras

Emilio Francesquini

[francesquini@ic.unicamp.br](mailto:francesquini@ic.unicamp.br)

Instituto de Computação - UNICAMP

Aulas 16,17 e 18 - maio de 2017



**UNICAMP**

# Disclaimer

- Esses slides foram preparados para um curso de Estrutura de Dados ministrado na UNICAMP
- Este material pode ser usado livremente desde que sejam mantidos os créditos dos autores e da instituição.
- Os exemplos apresentados aqui foram retirados do livro texto CLRS
- Alguns slides foram baseados nos slides preparados pela Profa. L. S. Assis



**UNICAMP**

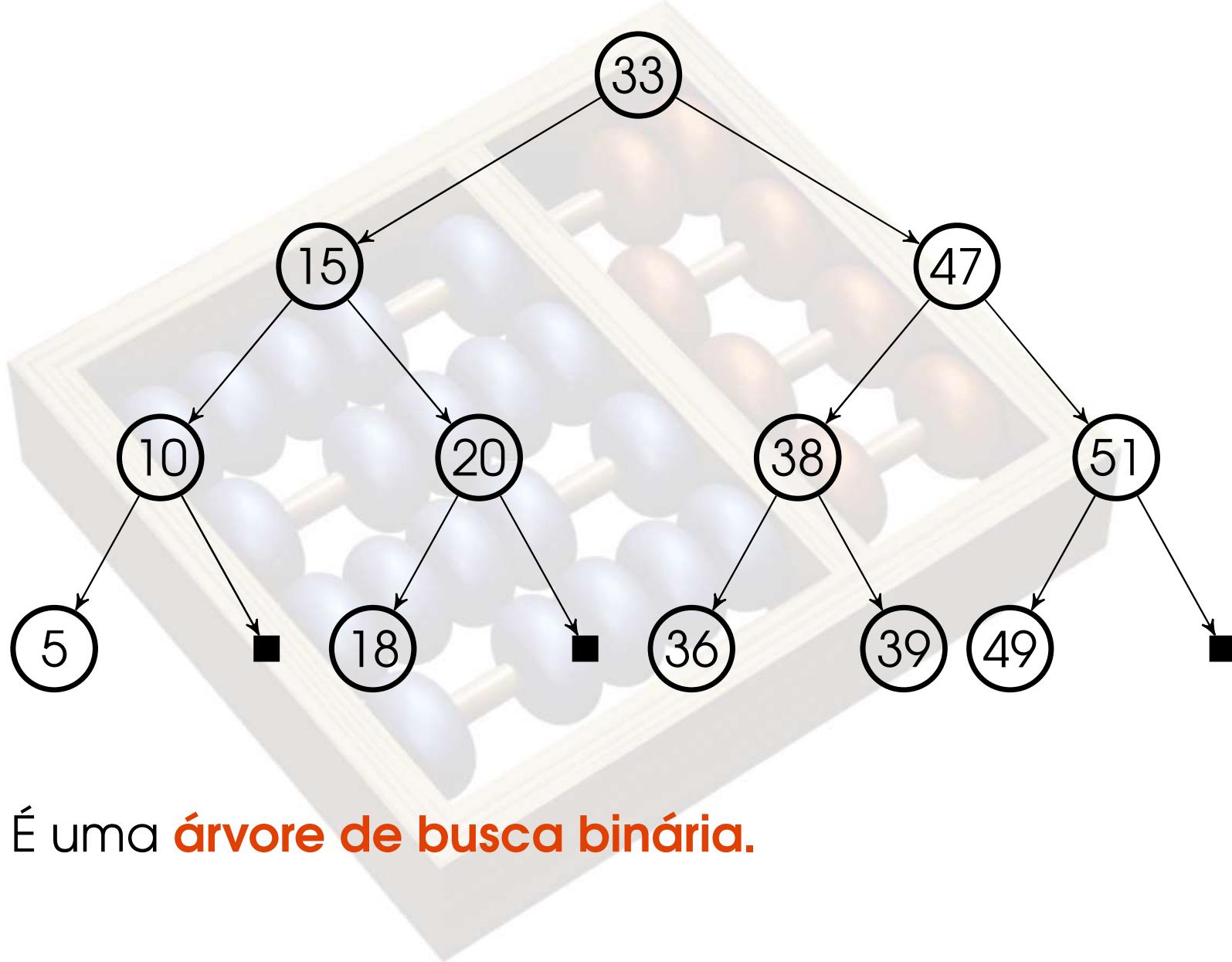
# Recap.: Árvores binárias de busca

- Uma árvore binária é chamada **árvore binária de busca** se para **cada nó p** vale que:
  - todo nó da sub-árvore **esquerda** de p tem chave **menor que a chave de p**;
  - todo nó da sub-árvore **direita** de p tem chave **maior que a chave de p**.



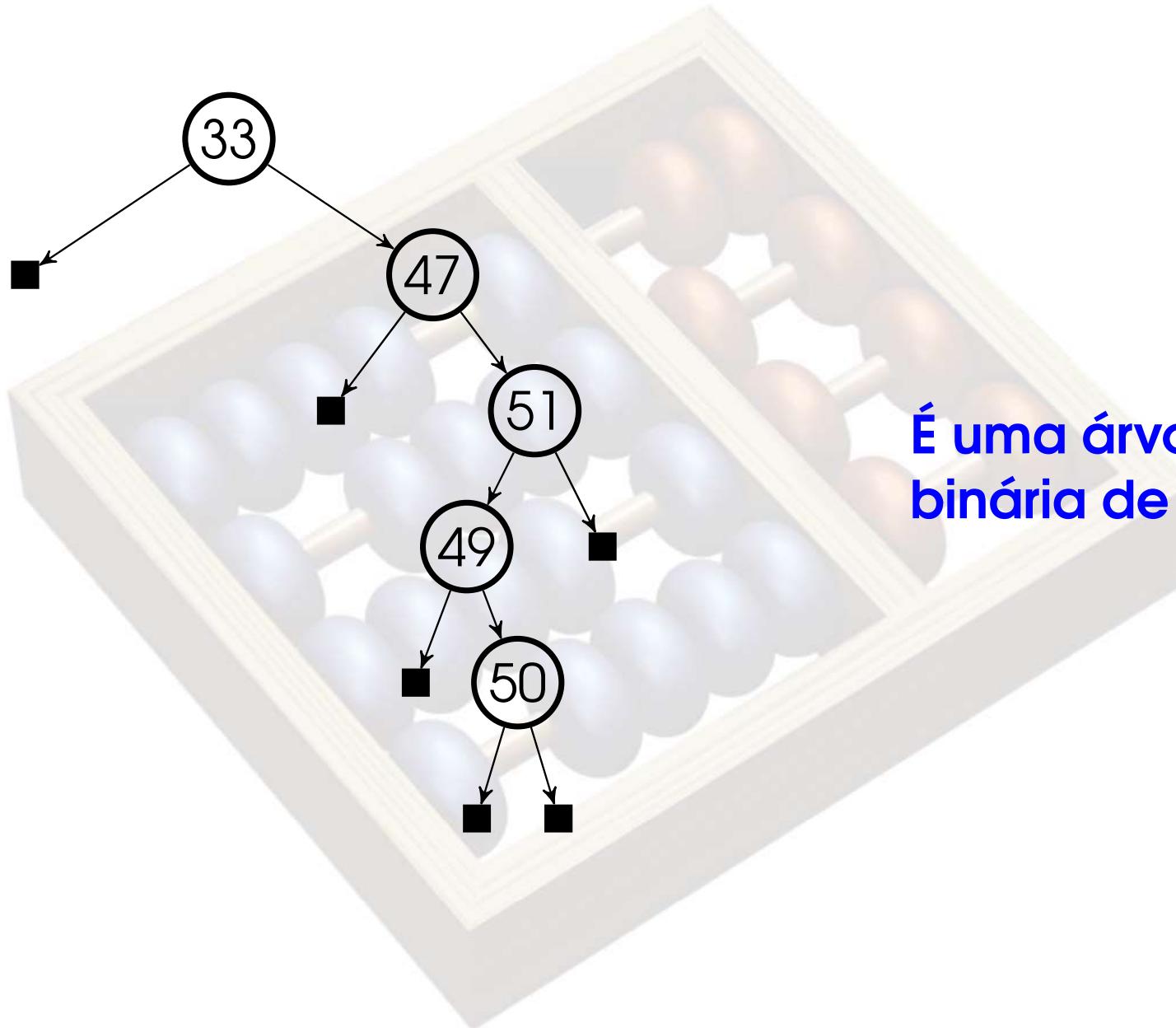
**UNICAMP**

# Recap.: Árvores binárias de busca



UNICAMP

# Recap.: Árvores binárias de busca

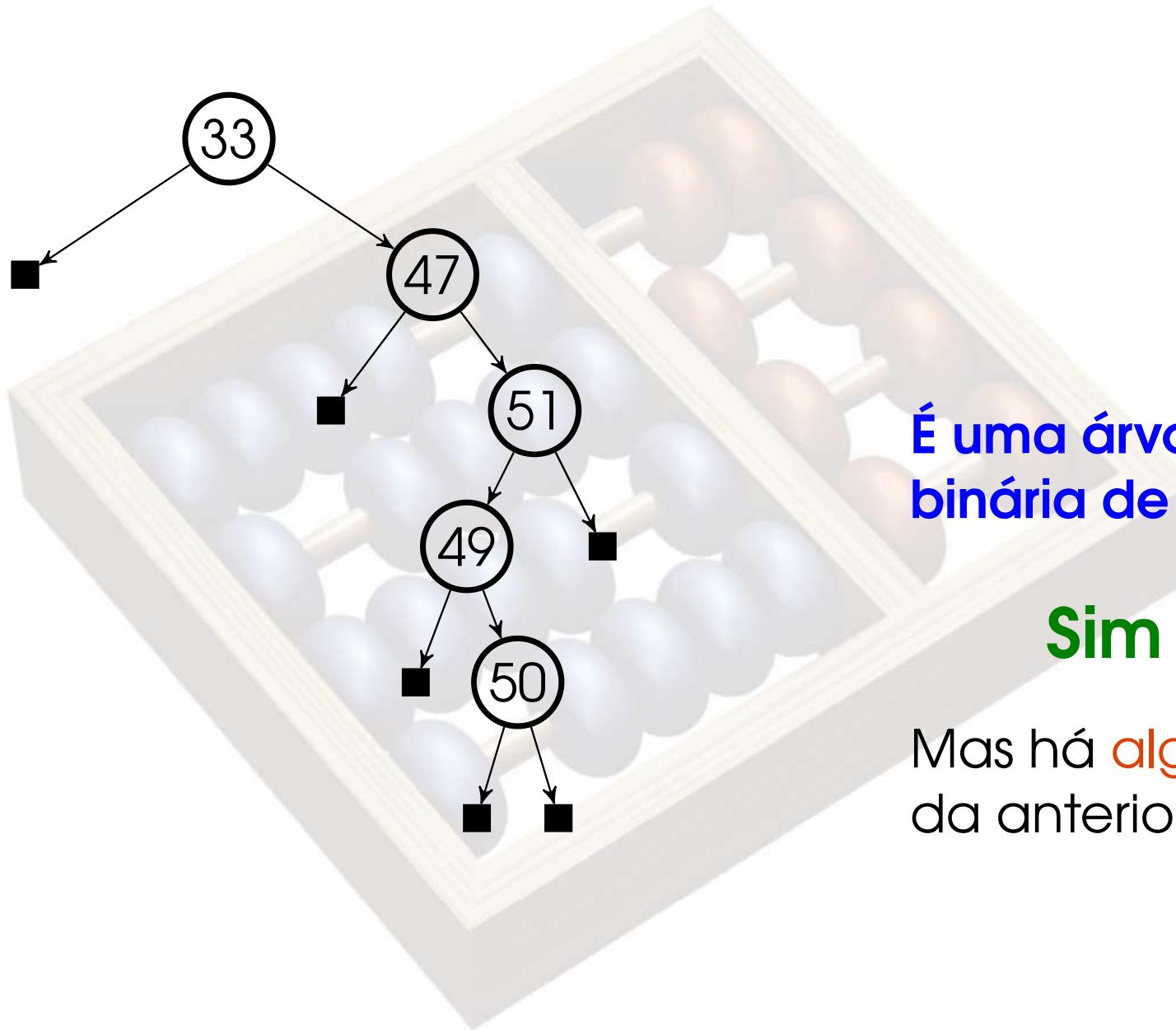


É uma árvore  
binária de busca?



UNICAMP

# Recap.: Árvores binárias de busca



É uma árvore  
binária de busca?

Sim

Mas há algo diferente  
da anterior...



UNICAMP

# Árvores Balanceadas de Busca

- Árvores de busca com garantias de altura máxima
- Em outras palavras, árvores balanceadas têm a garantia de que a sua **altura é no máximo  $O(\lg n)$**
- Árvores AVL e árvores **rubro-negras** são exemplos de árvores balanceadas de busca



**UNICAMP**

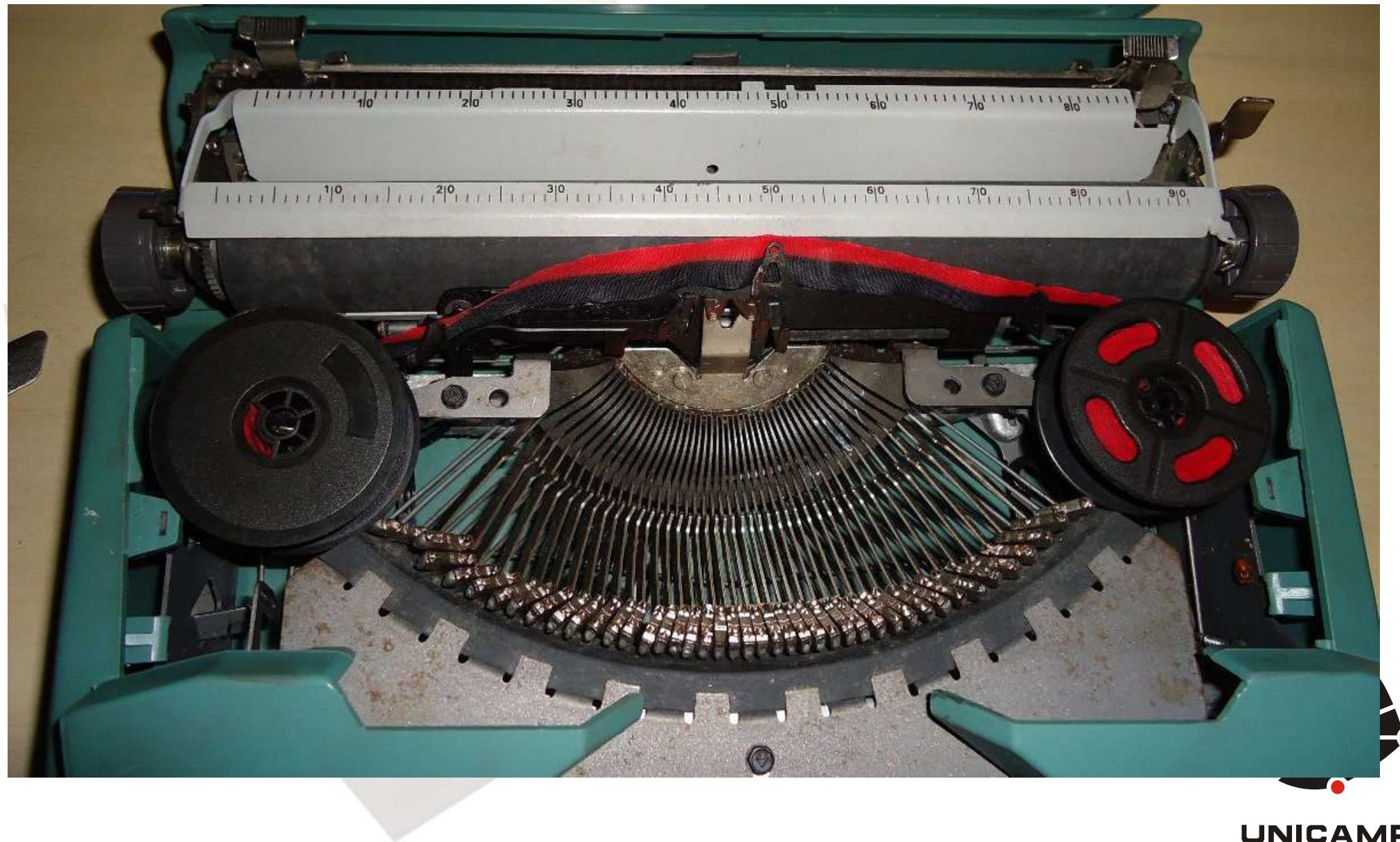
# Árvores Rubro-Negras

- **Rubro-Negras** são árvores de busca balanceadas
- Também chamadas de árvores vermelho-preto (do inglês *red-black trees*)
- Receberam este nome em um artigo de Guibas e Sedgewick em 1978
  - Supostamente pois impressora apenas era capaz de imprimir vermelho e preto...



UNICAMP

# Árvores Rubro-Negras



# Árvores Rubro-Negras

- Uma árvore rubro-negra é uma árvore de busca binária, logo segue todas as regras:
  - todo nó da sub-árvore **esquerda** de um nó p tem chave **menor que a chave de p**;
  - todo nó da sub-árvore **direita** de um nó p tem chave **maior que a chave de p**.
- Além disto, cada nó de uma árvore rubro negra tem os seguintes campos:
  - **cor** – Indica se o nó é vermelho ou preto
  - **chave** (ou valor) – Conteúdo do nó
  - **dir, esq** – Sub-árvores direita e esquerda
  - **pai** – Apontador para o pai do nó
- Se o filho ou o pai de um nó **não existir**, o campo é **preenchido com NULL**



UNICAMP

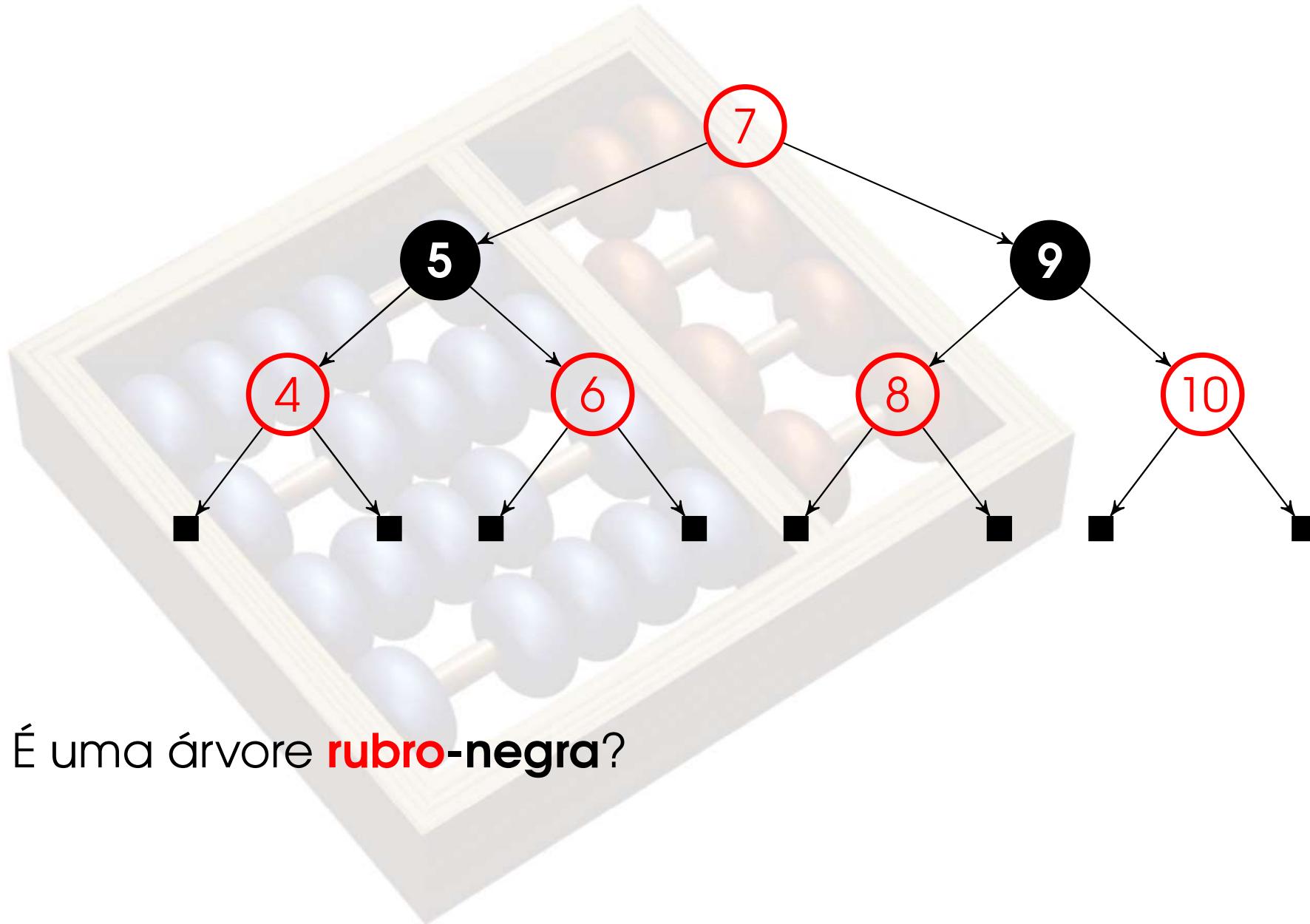
# Árvores Rubro-Negras

- Porém, uma árvore rubro-negra tem algumas **regras adicionais**:
  - Regra 1: Um nó é **vermelho** ou é **preto**
  - Regra 2: A raiz é **preta**
  - Regra 3: Toda **folha (NULL)** é **preta**
  - Regra 4: Se um nó é **vermelho** então ambos os seus filhos são **pretos**
  - Regra 5: Para cada nó  $p$ , **todos** os caminhos desde  $p$  até as folhas contêm o **mesmo número de nós pretos**



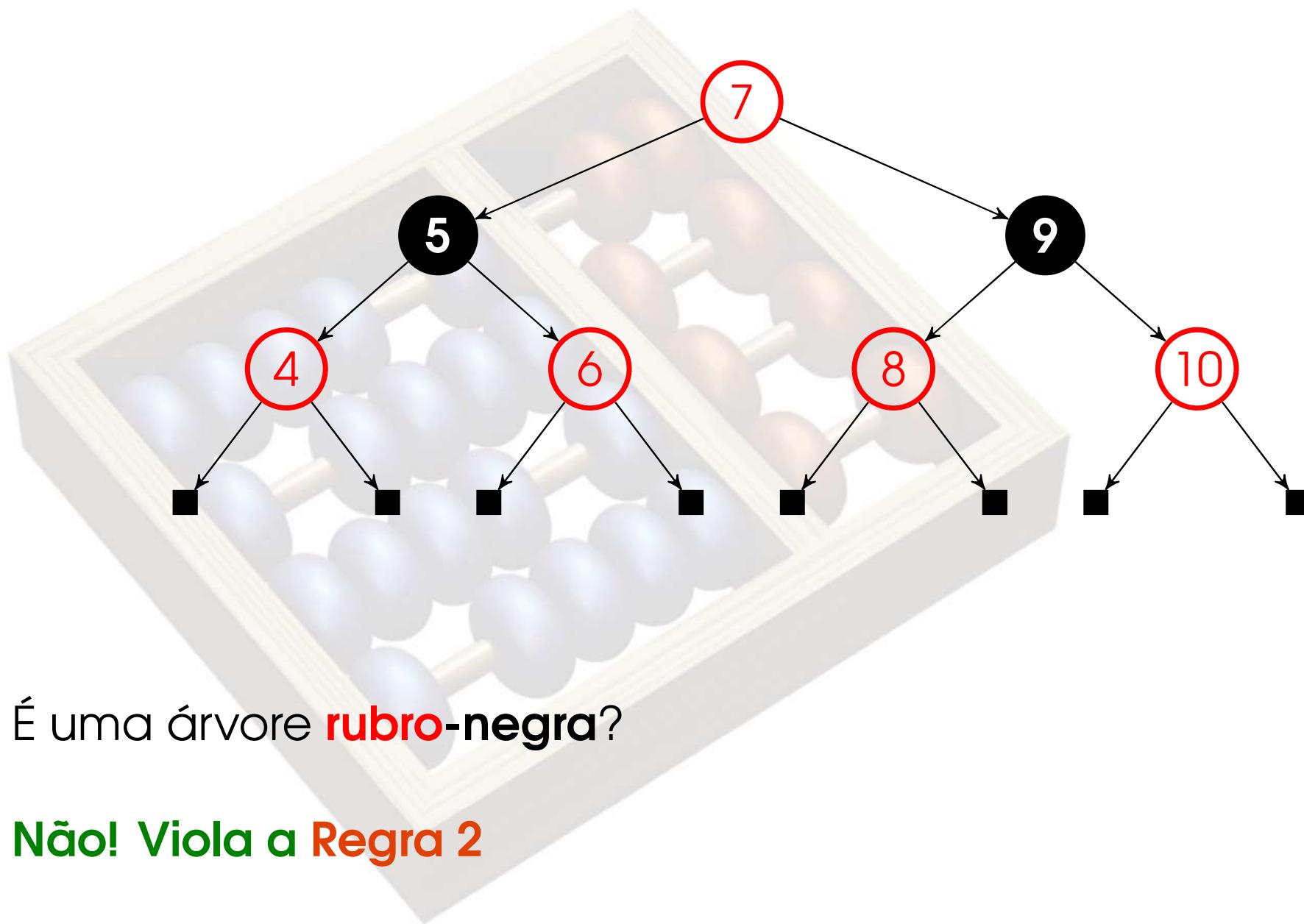
**UNICAMP**

# Árvores Rubro-Negras - Reconhecendo 1



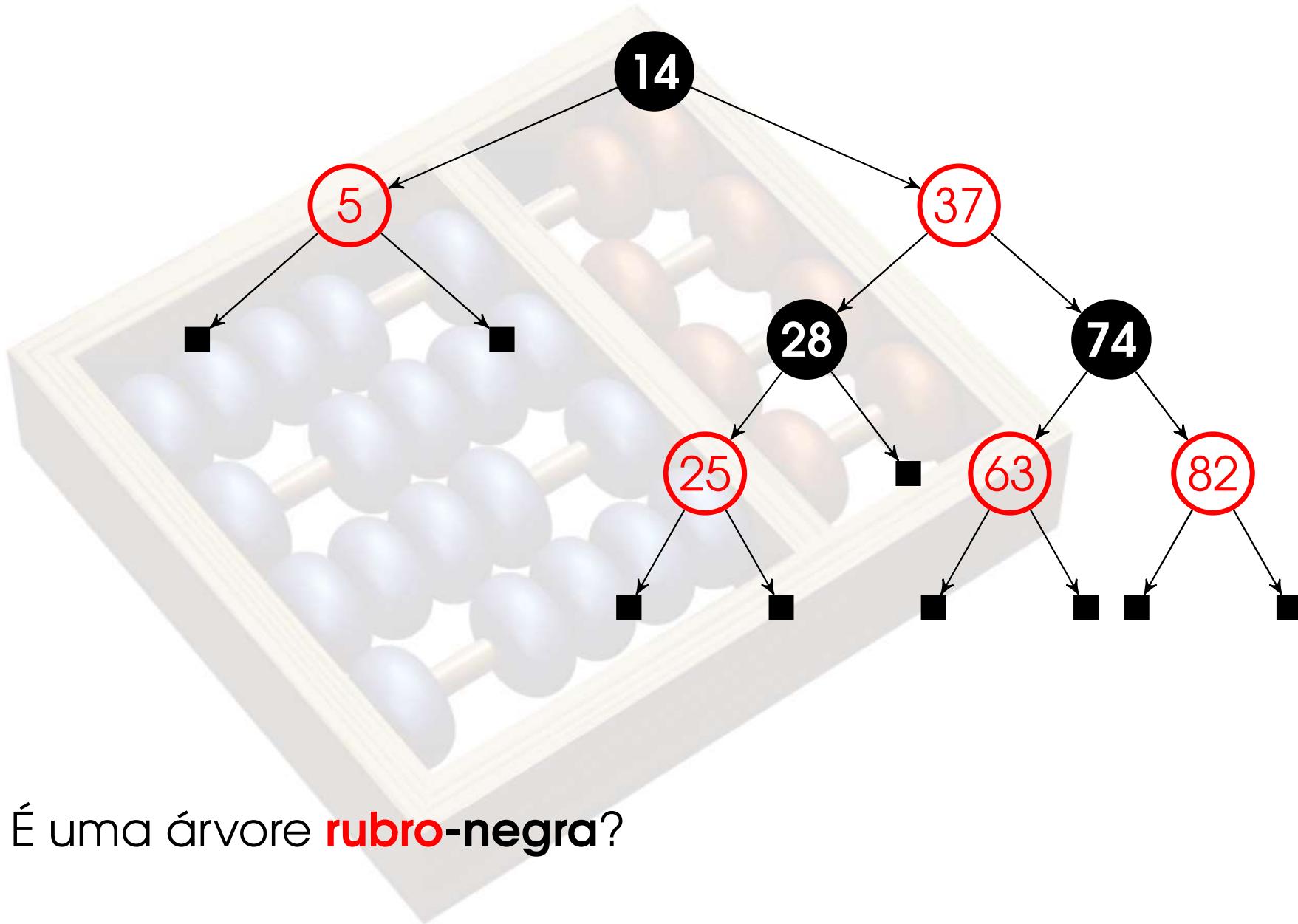
UNICAMP

# Árvores Rubro-Negras - Reconhecendo 1



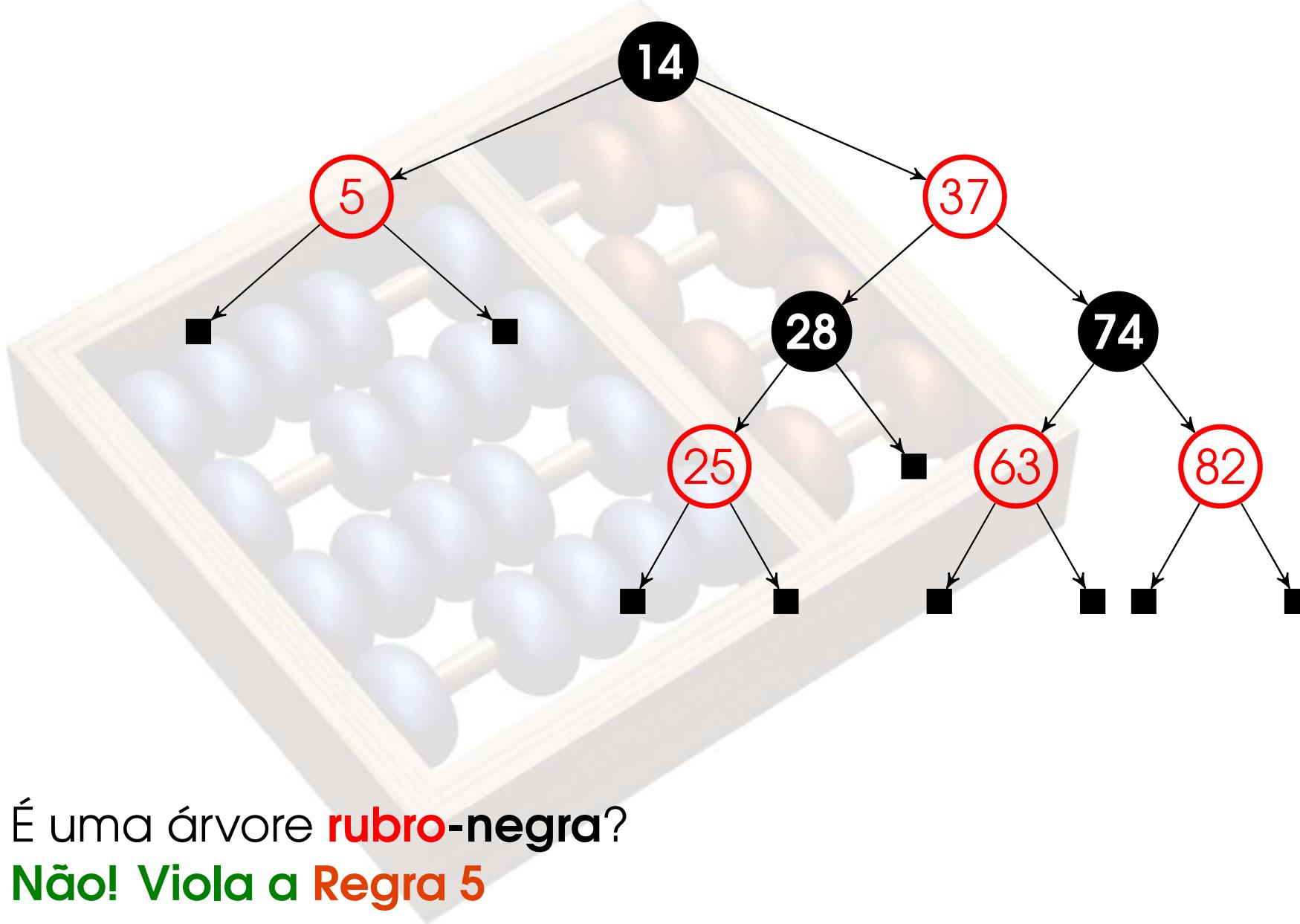
UNICAMP

# Árvores Rubro-Negras - Reconhecendo 2

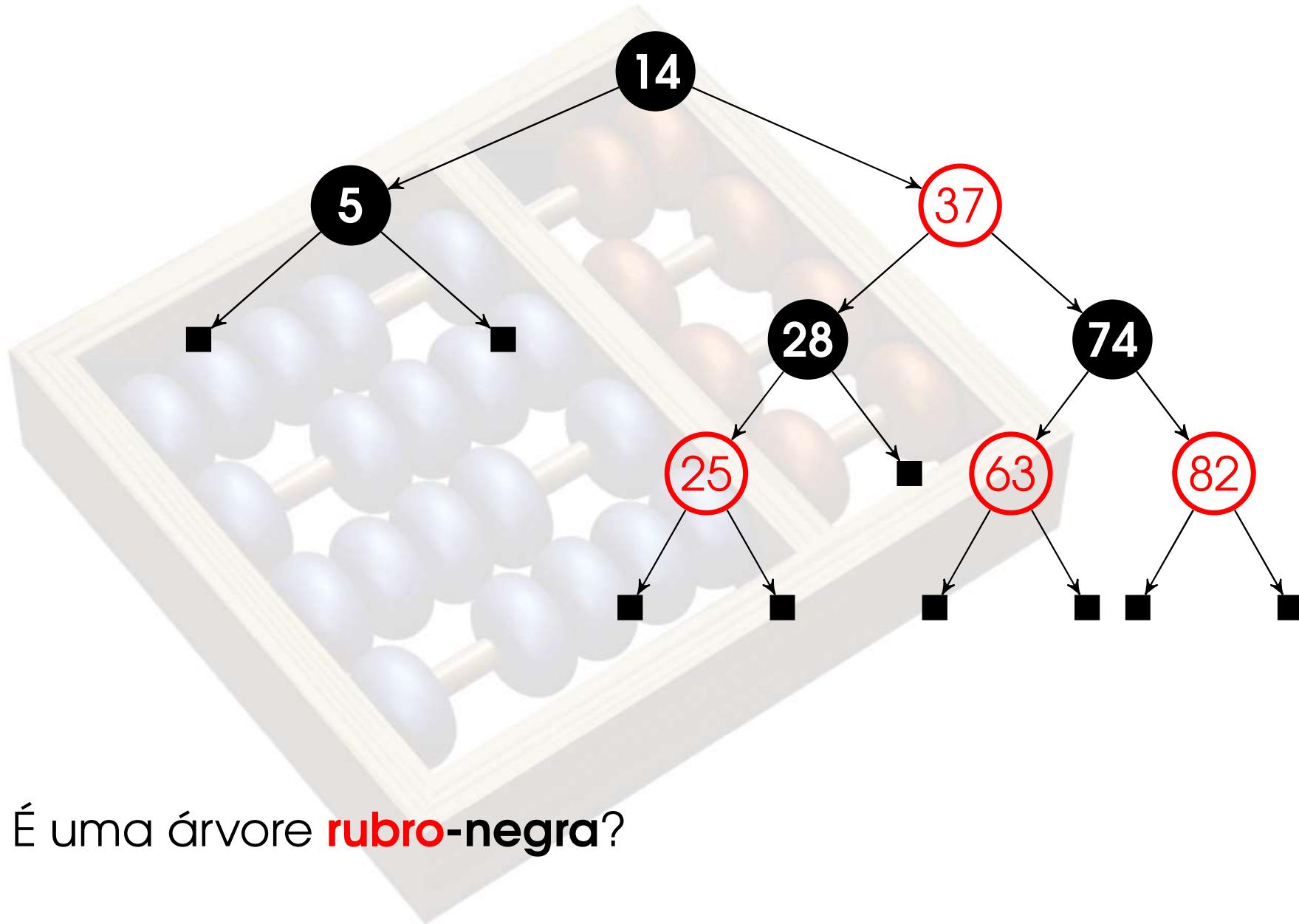


UNICAMP

# Árvores Rubro-Negras - Reconhecendo 2

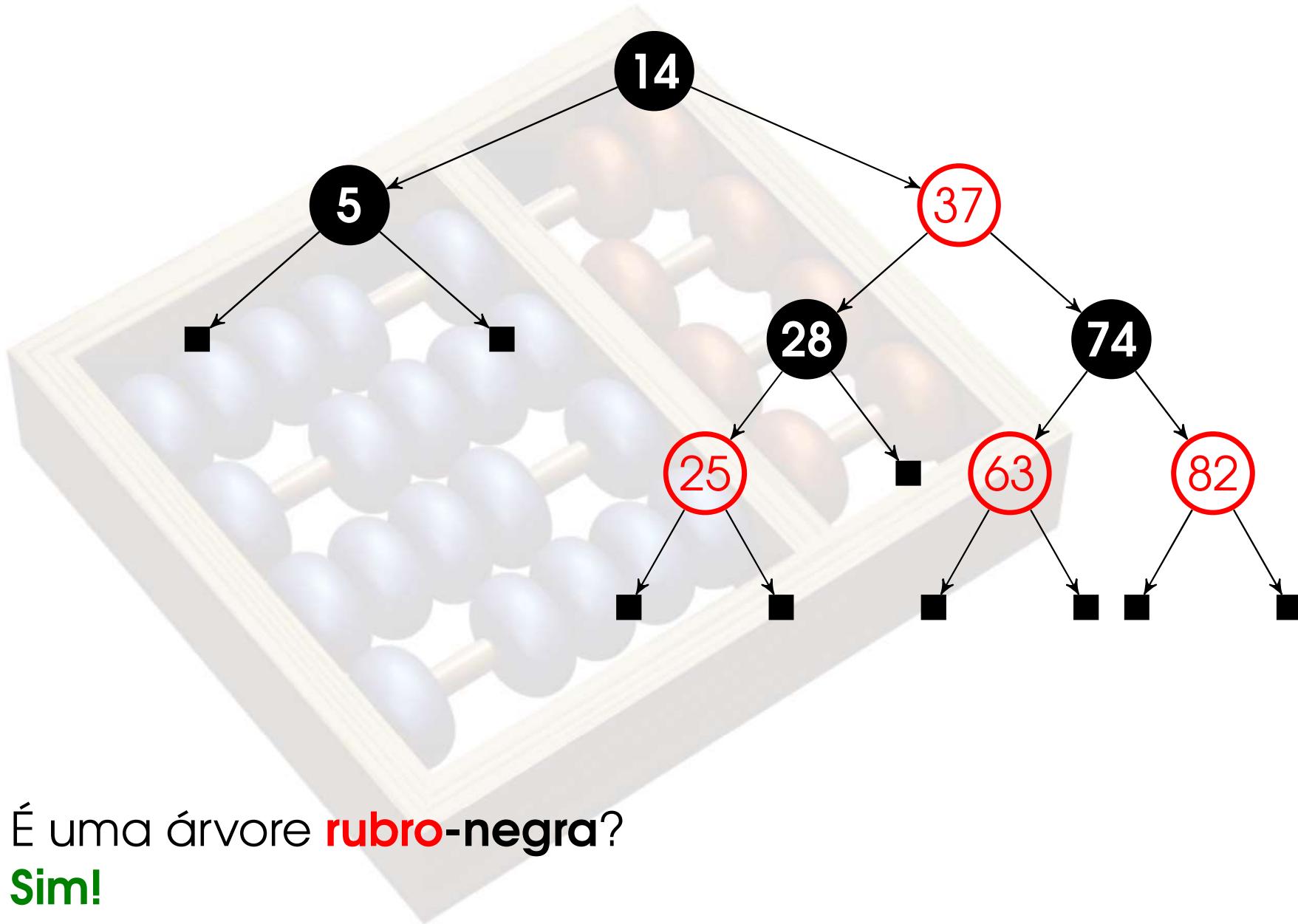


# Árvores Rubro-Negras - Reconhecendo 3



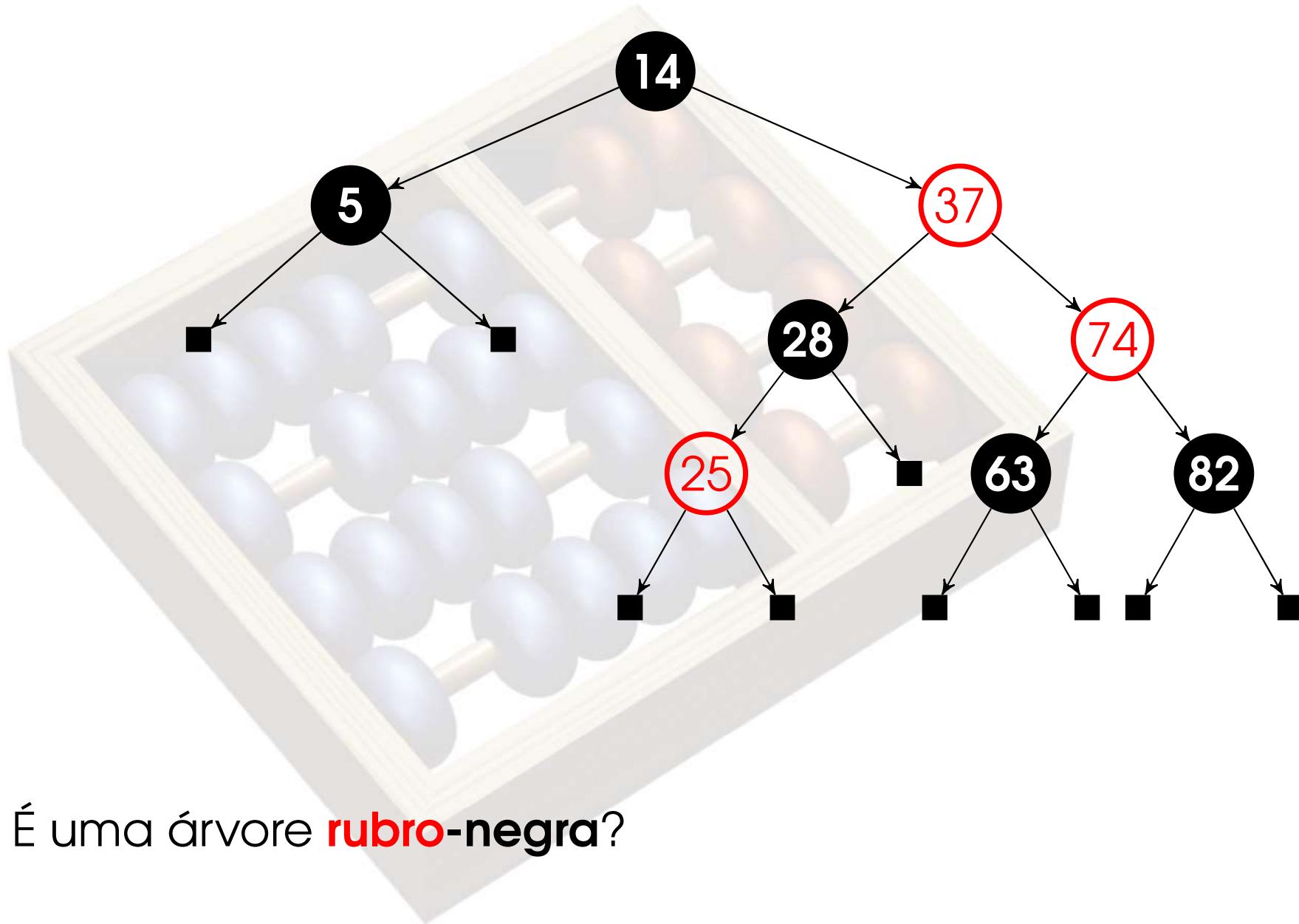
UNICAMP

# Árvores Rubro-Negras - Reconhecendo 3



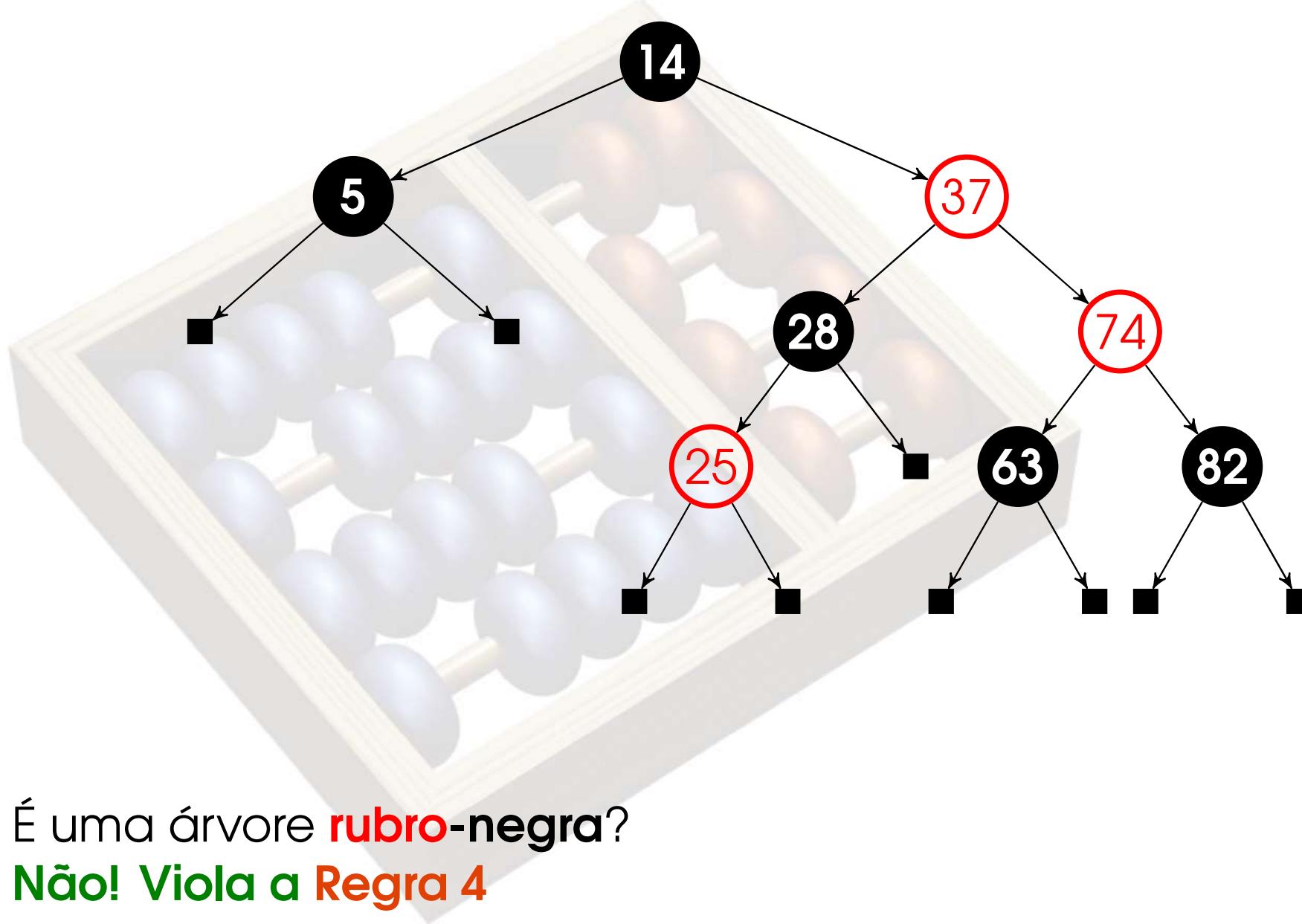
UNICAMP

# Árvores Rubro-Negras - Reconhecendo 4



UNICAMP

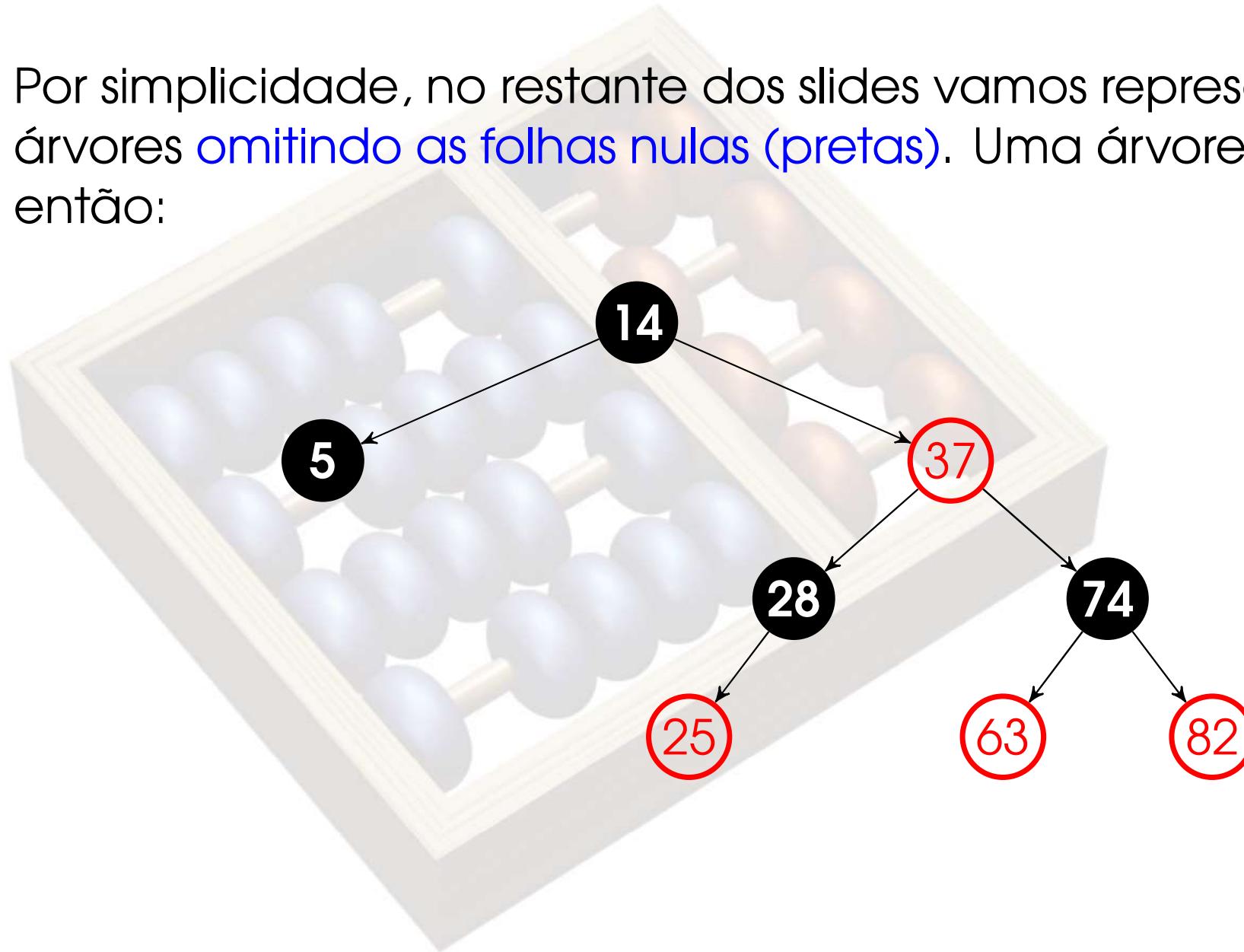
# Árvores Rubro-Negras - Reconhecendo 4



UNICAMP

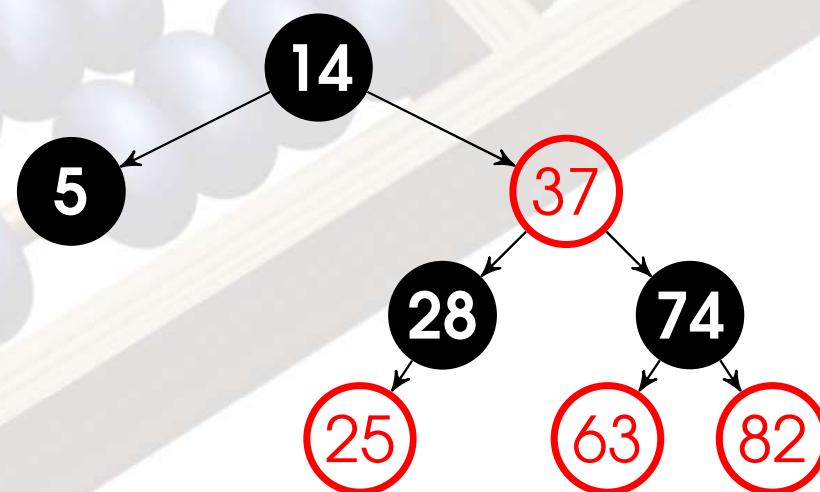
# Árvores Rubro-Negras - Representação

Por simplicidade, no restante dos slides vamos representar as árvores omitindo as folhas nulas (pretas). Uma árvore válida seria então:



# Árvores Rubro-Negras

- Ok! Já vimos todas as regras de uma árvore rubro-negra. Mas qual a razão de tudo isso? **Como isso nos ajuda?**
- **Restringindo** a maneira que os nós podem ser coloridos do caminho da raíz até qualquer uma das suas folhas, as árvores rubro-negras:
  - **Garantem** que nenhum dos caminhos será maior que 2x o comprimento de qualquer outro
  - **Garantem** que a árvore é aproximadamente **balanceada**

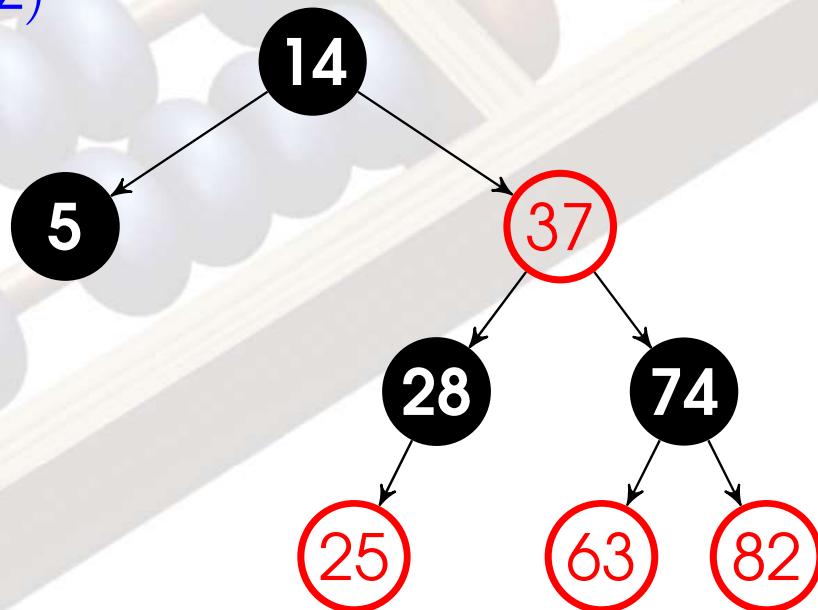


UNICAMP

# Altura de preto

A **altura de preto** de um nó  $n$  é definida como o número de nós pretos (sem incluir o próprio  $n$ ) visitados em qualquer **caminho** de  $n$  até as folhas.

- A altura de preto do nó  $n$  é denotada por  $H_p(n)$
- Pela **Regra 5**,  $H_p(n)$  é bem definida para todos os nós da árvore
- A **altura de preto da árvore rubro-negra** é definida como sendo a  $H_p(\text{raiz})$



# Lema 1 - Altura máxima

**Lema 1:** A altura máxima de uma árvore rubro-negra com  $n$  nós internos tem altura máxima de  $2\lg(n + 1)$

- A prova pode ser feita por indução utilizando a  $H_p$  dos nós da árvore. Veja o Lema 13.1 do CLRS para a prova completa.

**Corolário:** As operações de *Busca*, *Mínimo*, *Máximo*, *Sucessor* e *Predecessor* podem ser efetuadas em tempo  $\mathcal{O}(\lg(n))$



**UNICAMP**

# Operações em árvores rubro-negras

**Busca** – A busca que estamos acostumados **funciona sem modificações**

**Inserção e Remoção** – Mantém **apenas** as propriedades de árvores binárias de busca, mas **não mantém as propriedades rubro-negras**

Veja animação de operações em:  
<http://tommikaikkonen.github.io/rbtree>



**UNICAMP**

# Comparação entre estruturas de dados

| Tempo Assintótico (limitante superior) * |       |            |              |                   |            |                 |
|------------------------------------------|-------|------------|--------------|-------------------|------------|-----------------|
|                                          | Vetor | Vetor Ord. | Lista Ligada | Lista Ligada Ord. | Árv. Busca | Árv. Bal. Busca |
| Busca                                    | n     | $\lg(n)$   | n            | n                 | n          | $\lg(n)$        |
| Inserção                                 | 1     | n          | 1            | n                 | n          | $\lg(n)$        |
| Remoção                                  | n     | n          | 1            | 1                 | n          | $\lg(n)$        |

\* Esses são os tempos das operações nessas estruturas de dados tal qual elas foram apresentadas neste curso. Em alguns casos específicos é possível melhorar estes tempos. Ainda assim a árvore binária balanceada de busca continua sendo uma das estruturas mais eficientes dentre todas as que já vimos.



# Rotações

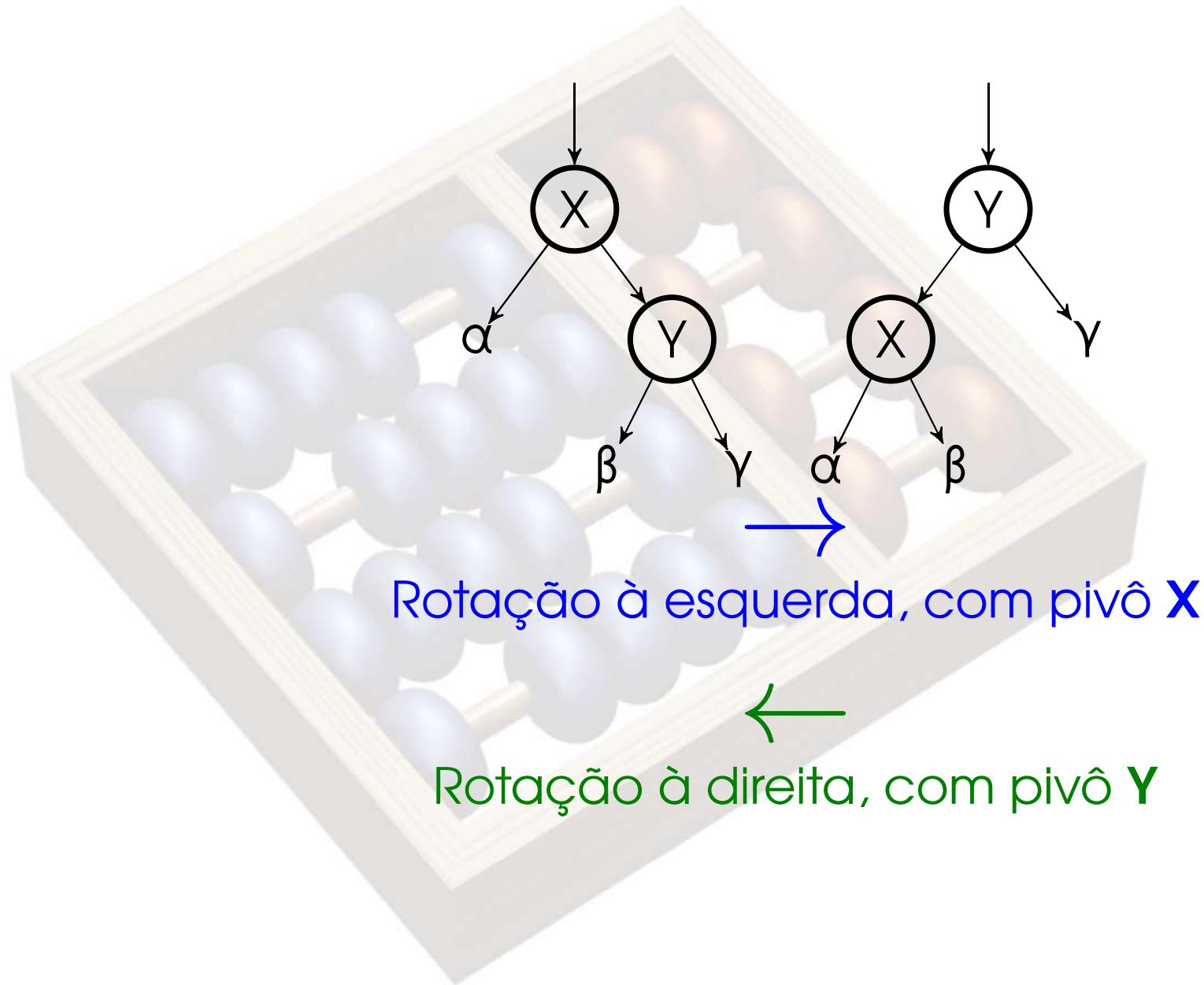
Antes de vermos como fazer inserções em uma árvore rubro-negra, é preciso apresentar o conceito de **rotações**.

- Usamos as **rotações para consertar** (parte do) o estrago feito pelas operações já conhecidas de inserção e remoção nas propriedades rubro-negras
- O resto do estrago é consertado utilizando **recoloração de nós**
- Rotações são operações **locais**: alteram um número **pequeno e constante** de ponteiros



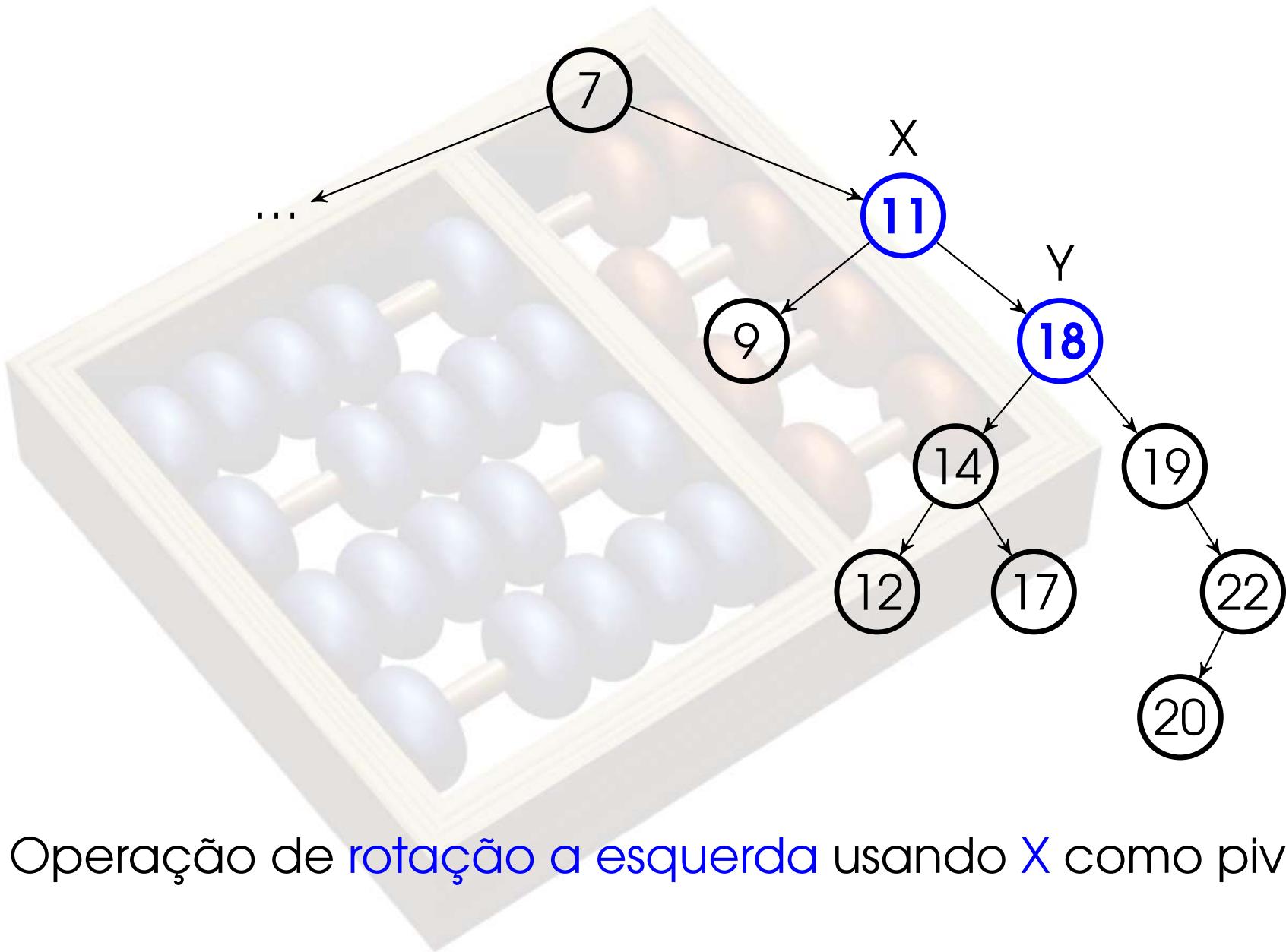
**UNICAMP**

# Rotações



UNICAMP

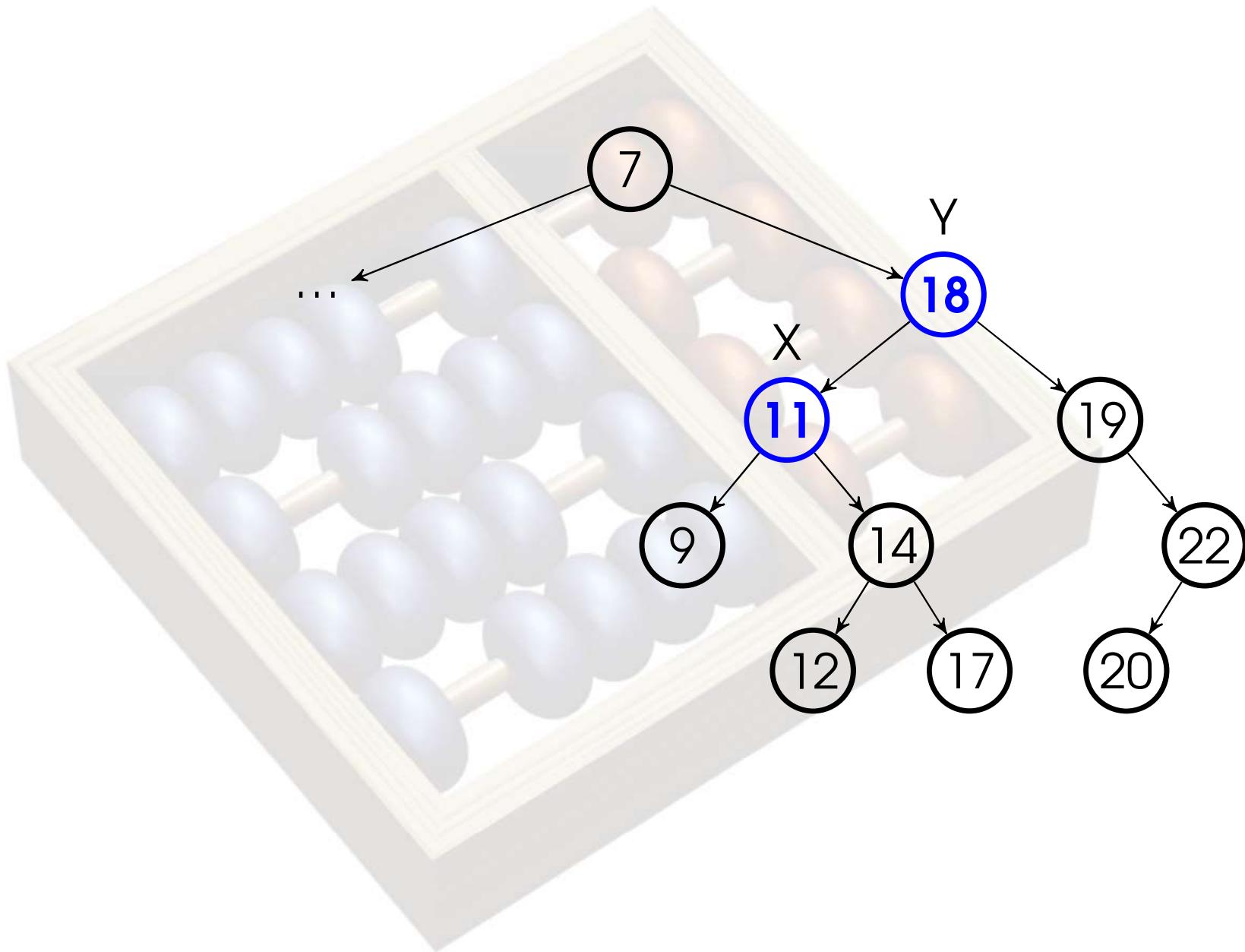
# Rotações



Operação de rotação a esquerda usando X como pivô...



# Rotações



**UNICAMP**

# Resumo

- Um nó que satisfaz as regras de uma árvore rubro-negra é denominado **equilibrado**, caso contrário é dito **desequilibrado**
- Em uma árvore rubro-negra **todos os nós estão equilibrados**
- Uma consequência direta das propriedades é que em qualquer caminho da raiz até uma folha **não existem dois nós vermelhos consecutivos**
- Cada vez que uma operação de modificação for realizada na árvore, o conjunto de propriedades é **verificado em busca de violações**
  - Caso alguma propriedade tenha sido quebrada, realizamos **rotações e ajustamos as cores** dos nós para que todas as propriedades continuem válidas



**UNICAMP**

# Inserções em árvores rubro-negras

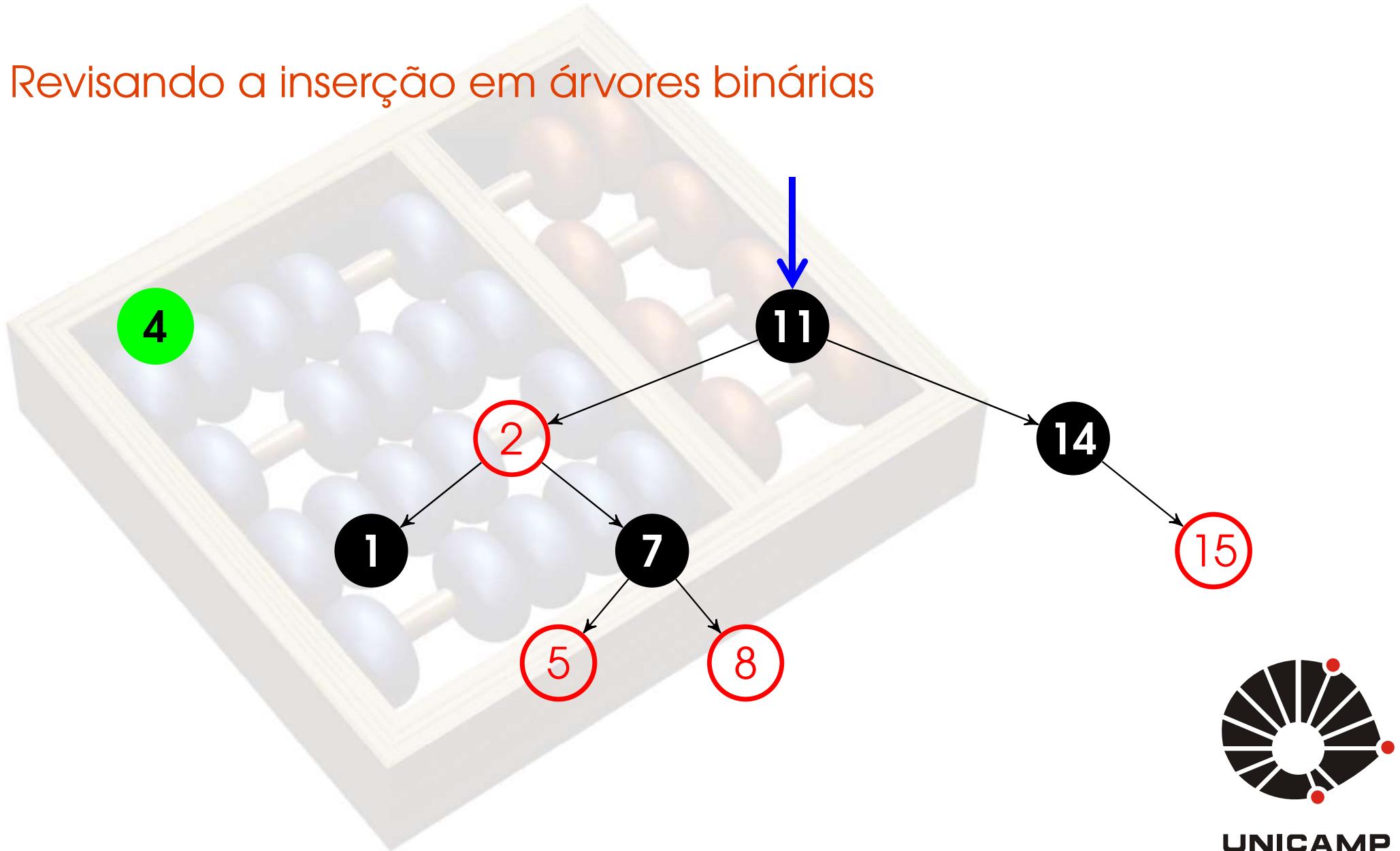
- Árvores rubro-negras são árvores de busca binária com algumas regras adicionais
  - Regra 1: Um nó é **vermelho** ou é **preto**
  - Regra 2: A raiz é **preta**
  - Regra 3: Toda **folha (NULL)** é **preta**
  - Regra 4: Se um nó é **vermelho** então ambos os seus filhos são **pretos**
  - Regra 5: Para cada nó  $p$ , **todos** os caminhos desde  $p$  até as folhas contêm o **mesmo número de nós pretos**
- Se utilizarmos o algoritmo que já conhecemos para a inserção em uma árvore rubro-negra, corremos o risco de quebrar algumas dessas regras. Mas quais?



**UNICAMP**

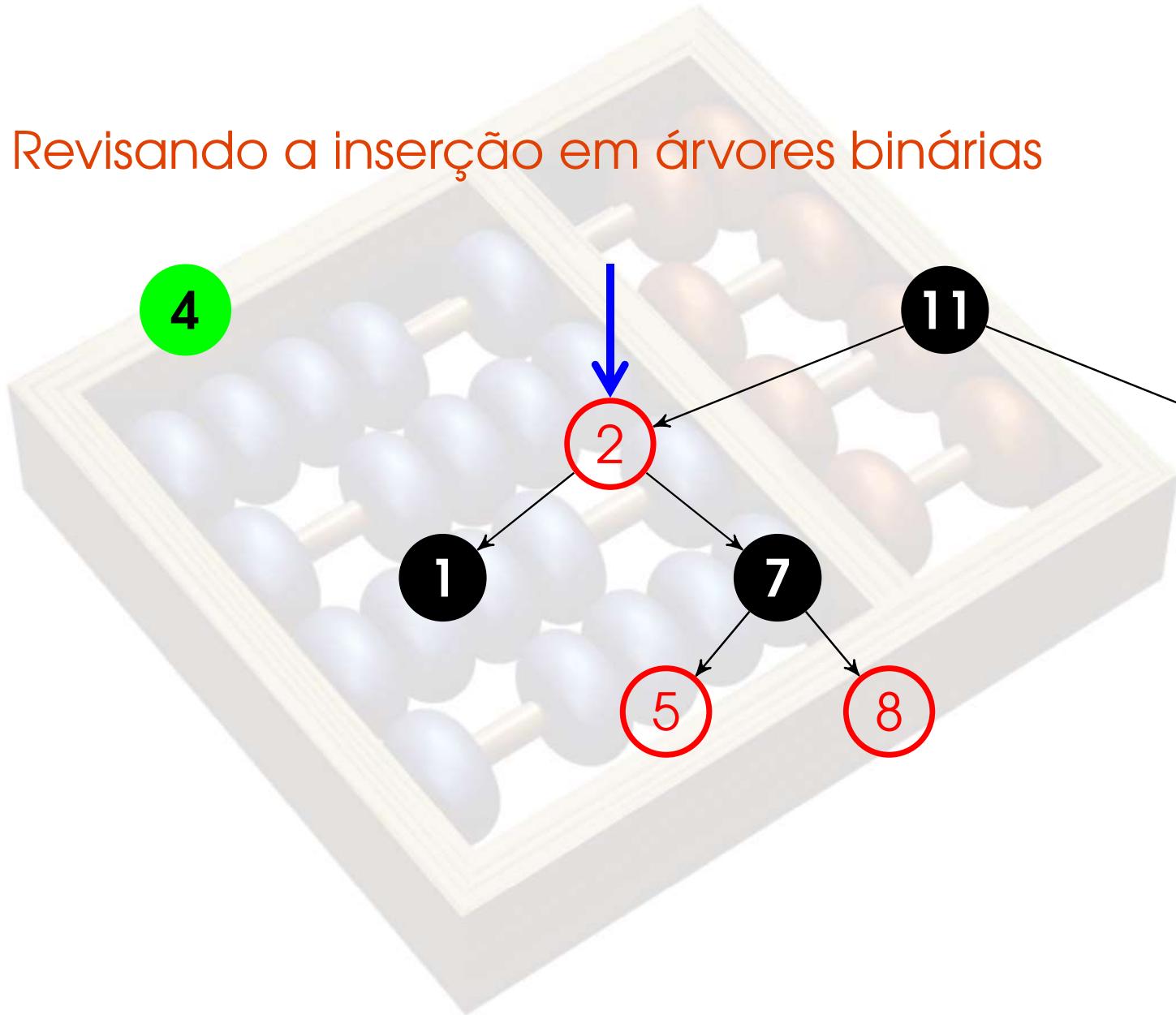
# Revisão Inserções em Árv. Bin. de Busca

Revisando a inserção em árvores binárias



# Revisão Inserções em Árv. Bin. de Busca

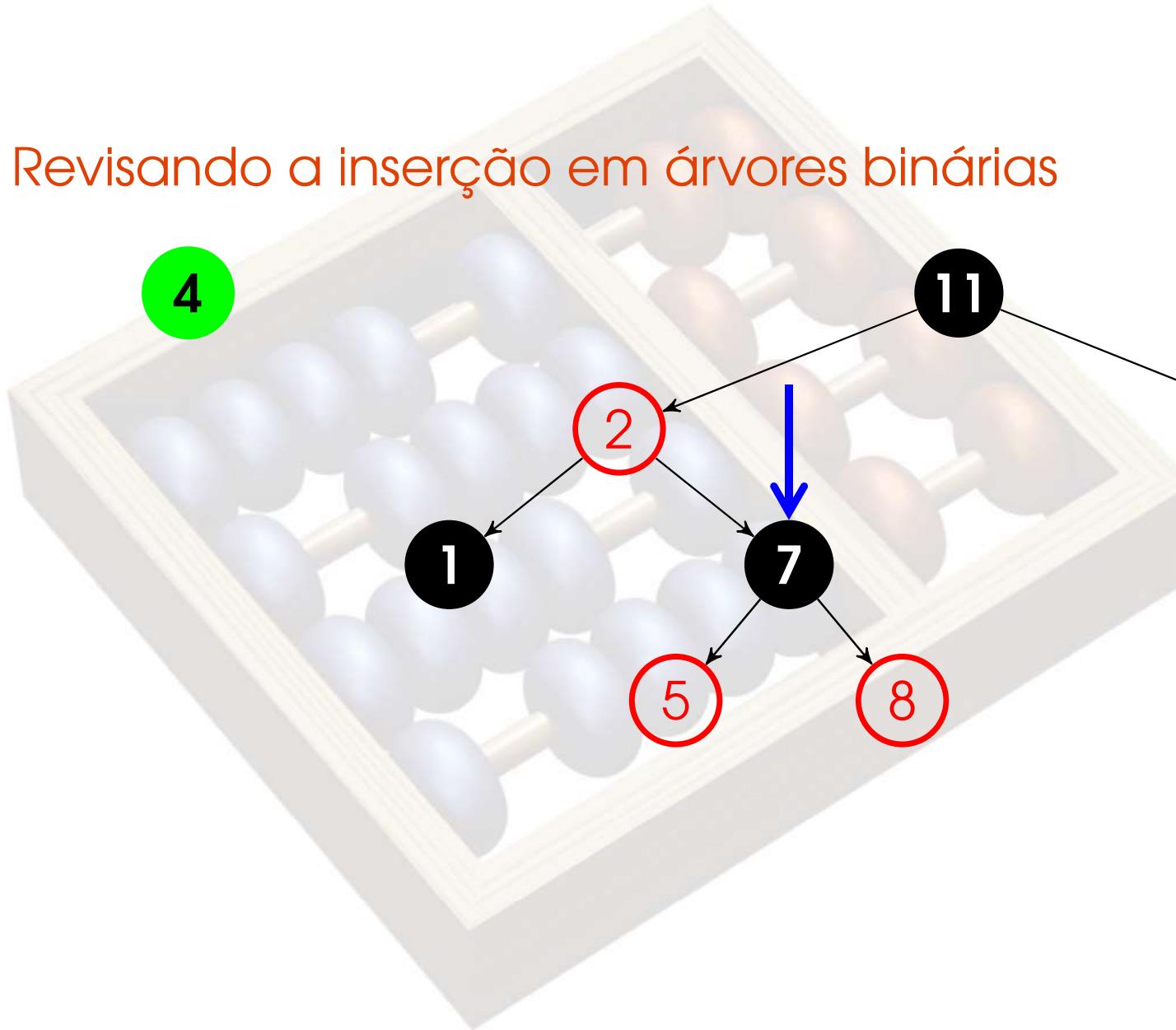
Revisando a inserção em árvores binárias



UNICAMP

# Revisão Inserções em Árv. Bin. de Busca

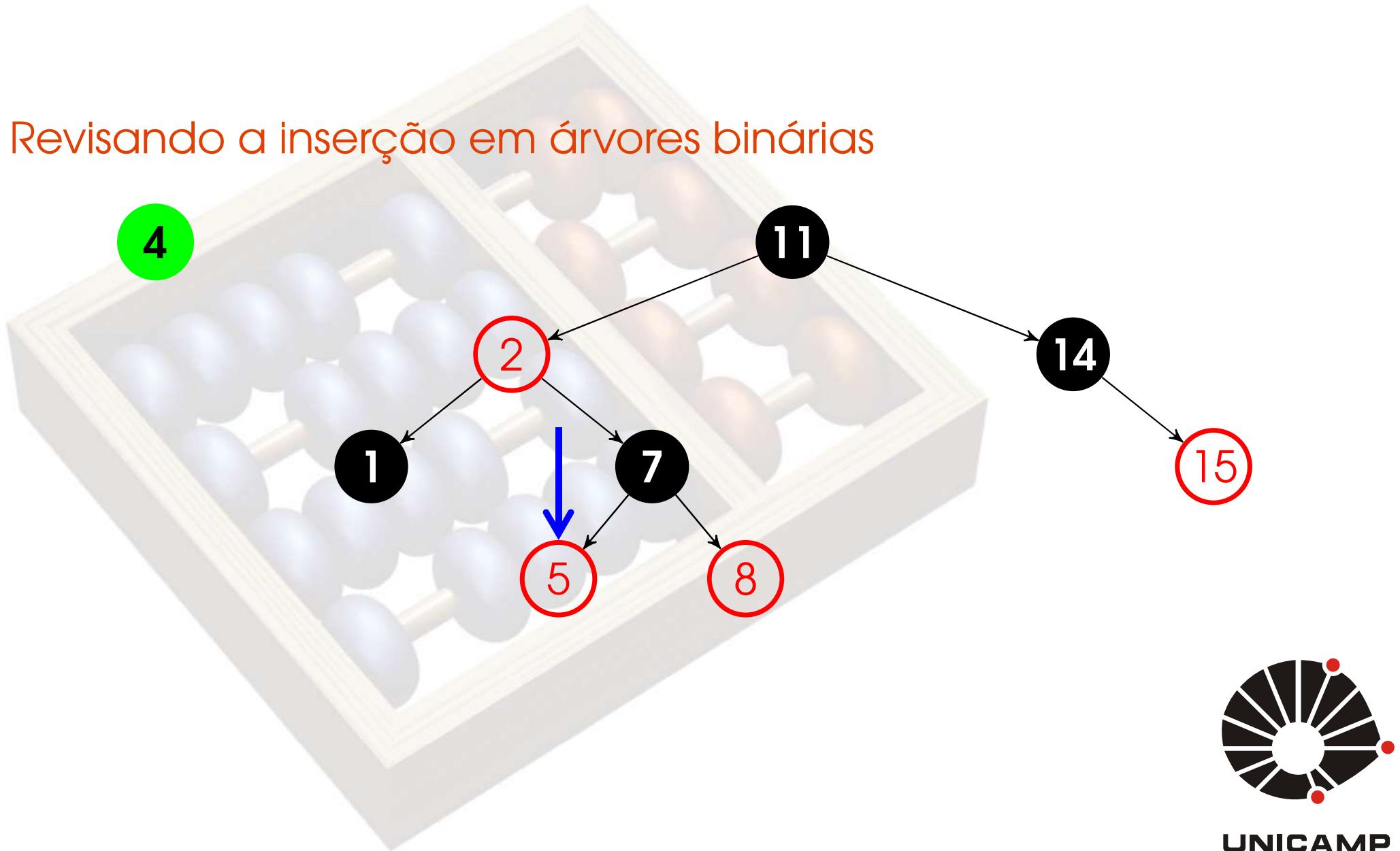
Revisando a inserção em árvores binárias



UNICAMP

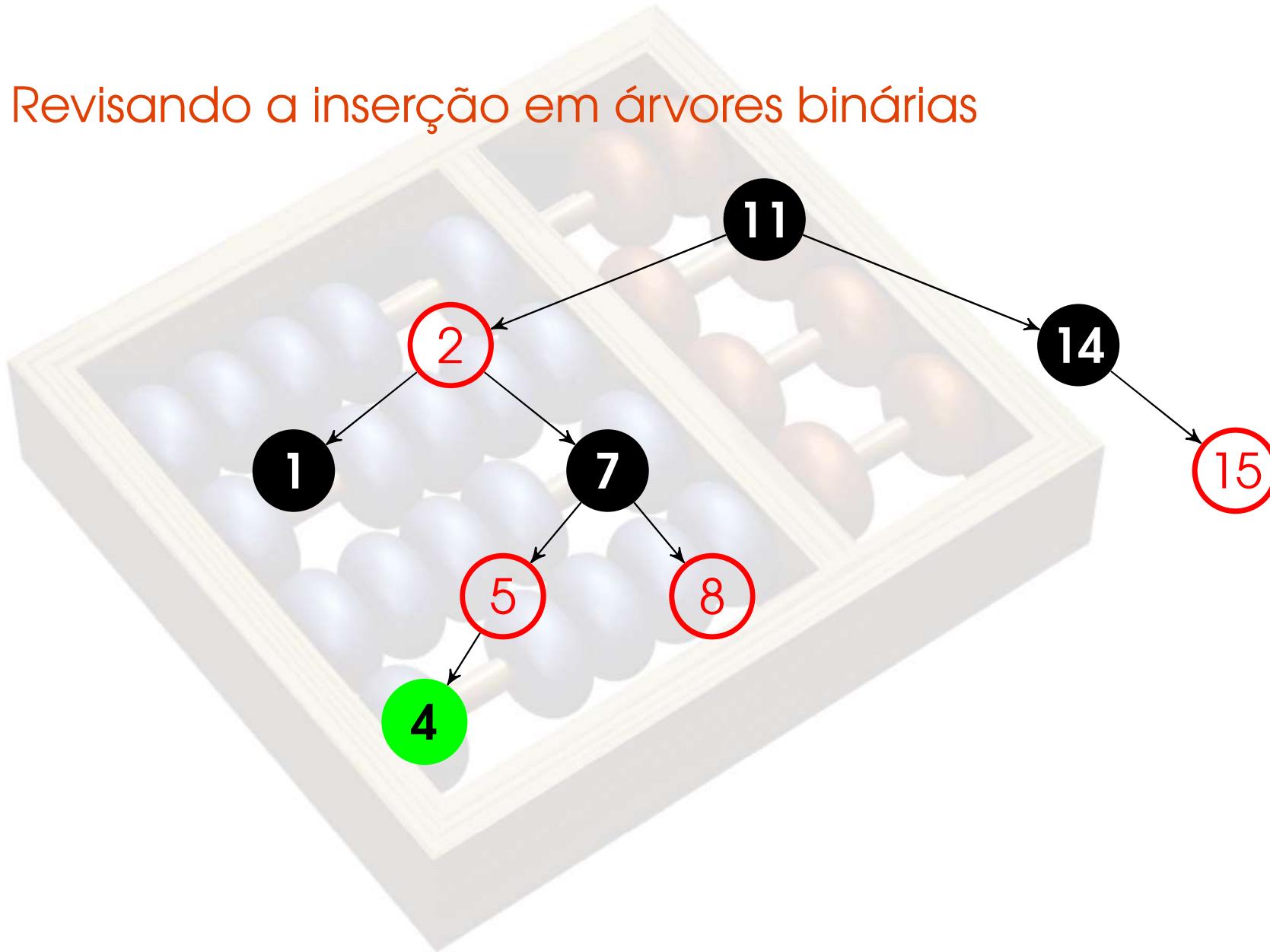
# Revisão Inserções em Árv. Bin. de Busca

Revisando a inserção em árvores binárias



# Revisão Inserções em Árv. Bin. de Busca

Revisando a inserção em árvores binárias



UNICAMP

# Revisão Inserções em Árv. Bin. de Busca

Revisando a inserção em árvores binárias



Quebra a **Regra 1** – Todo nó deve ser vermelho ou preto



UNICAMP

# Inserções em árvores rubro-negras

É preciso decidir, qual faz menos mal, colocar um nó vermelho ou um preto?

- O vermelho pode não quebrar nada
- O preto vai desequilibrar a altura de preto da raiz com certeza

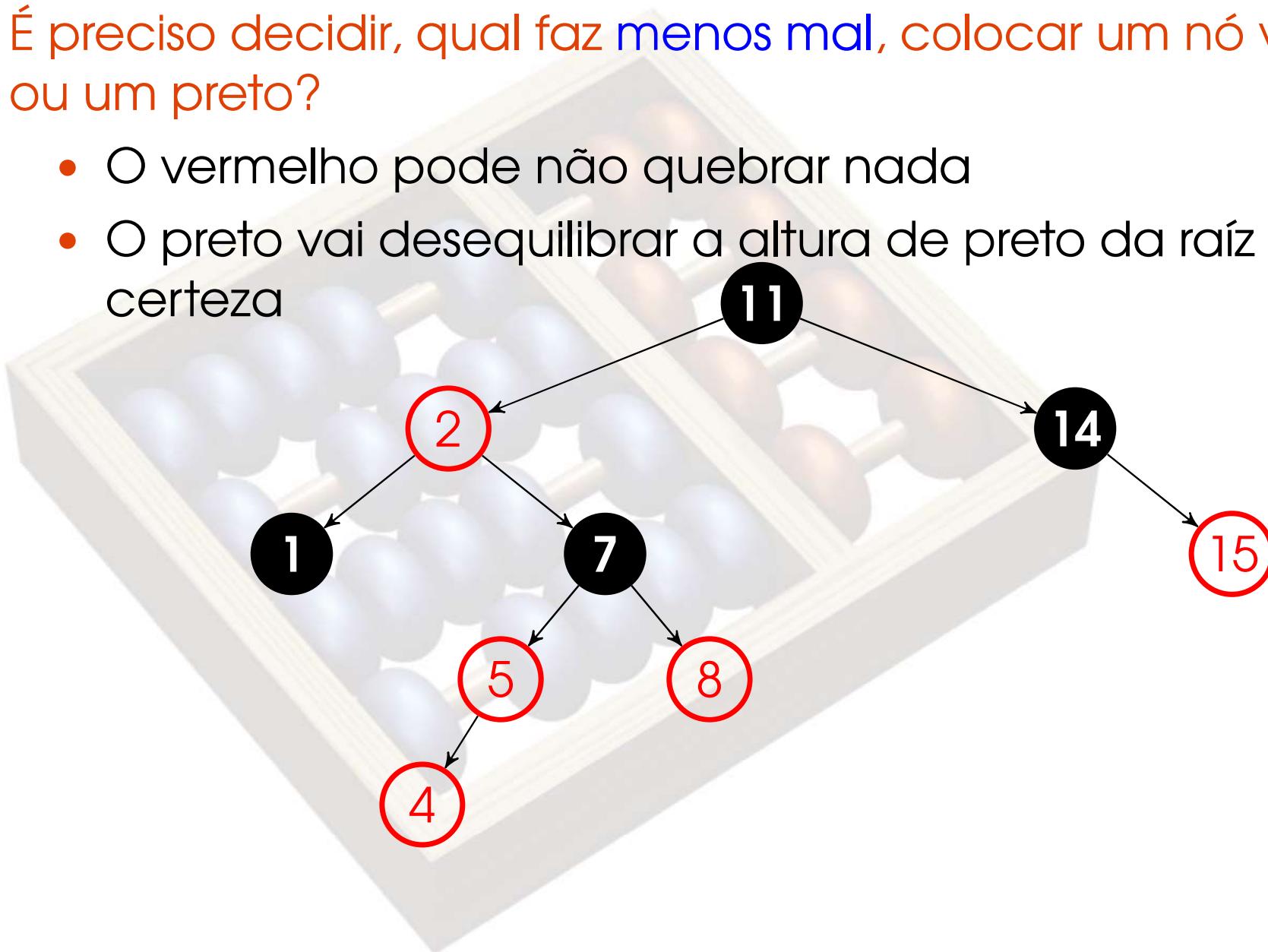


UNICAMP

# Inserções em árvores rubro-negras

É preciso decidir, qual faz menos mal, colocar um nó vermelho ou um preto?

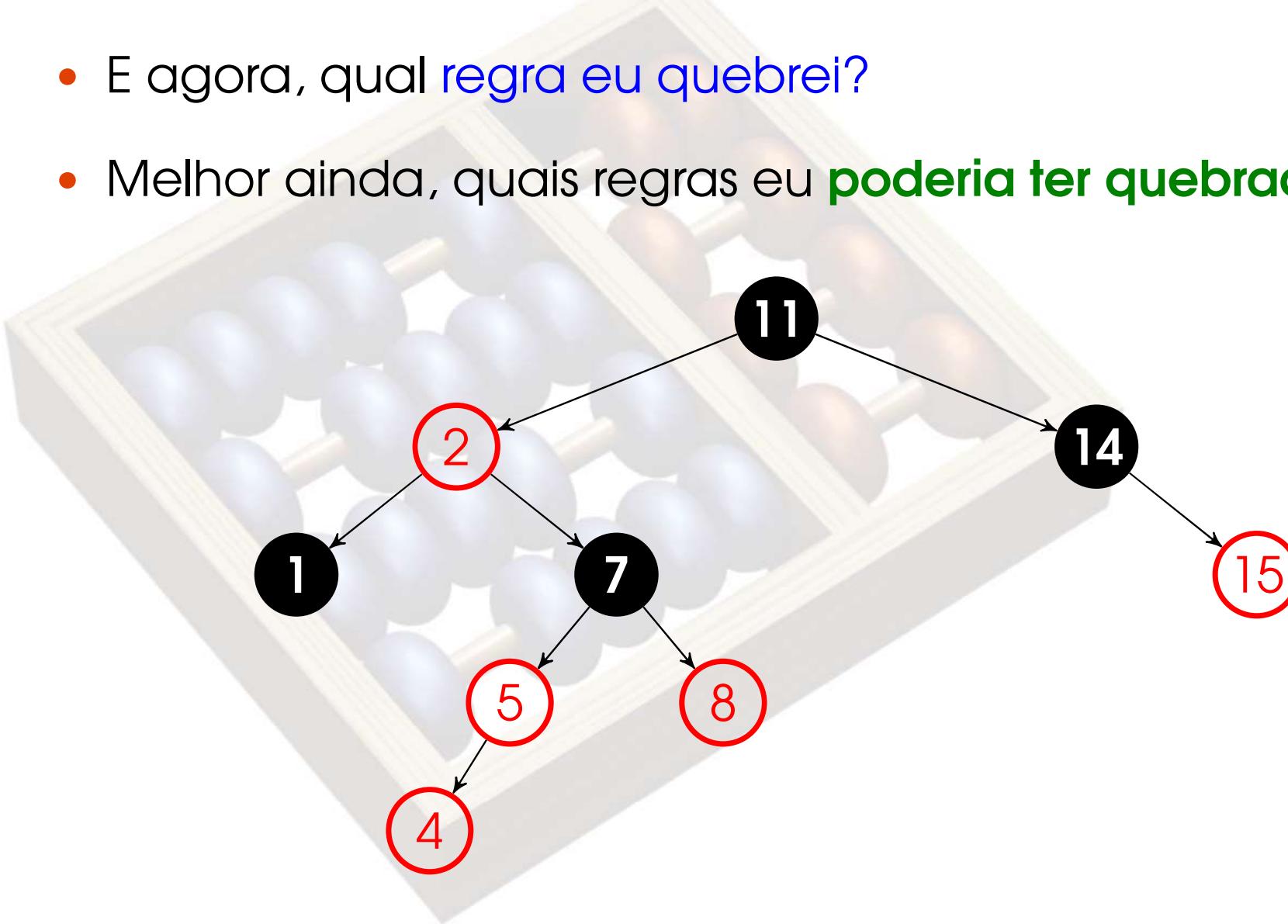
- O vermelho pode não quebrar nada
- O preto vai desequilibrar a altura de preto da raiz com certeza



UNICAMP

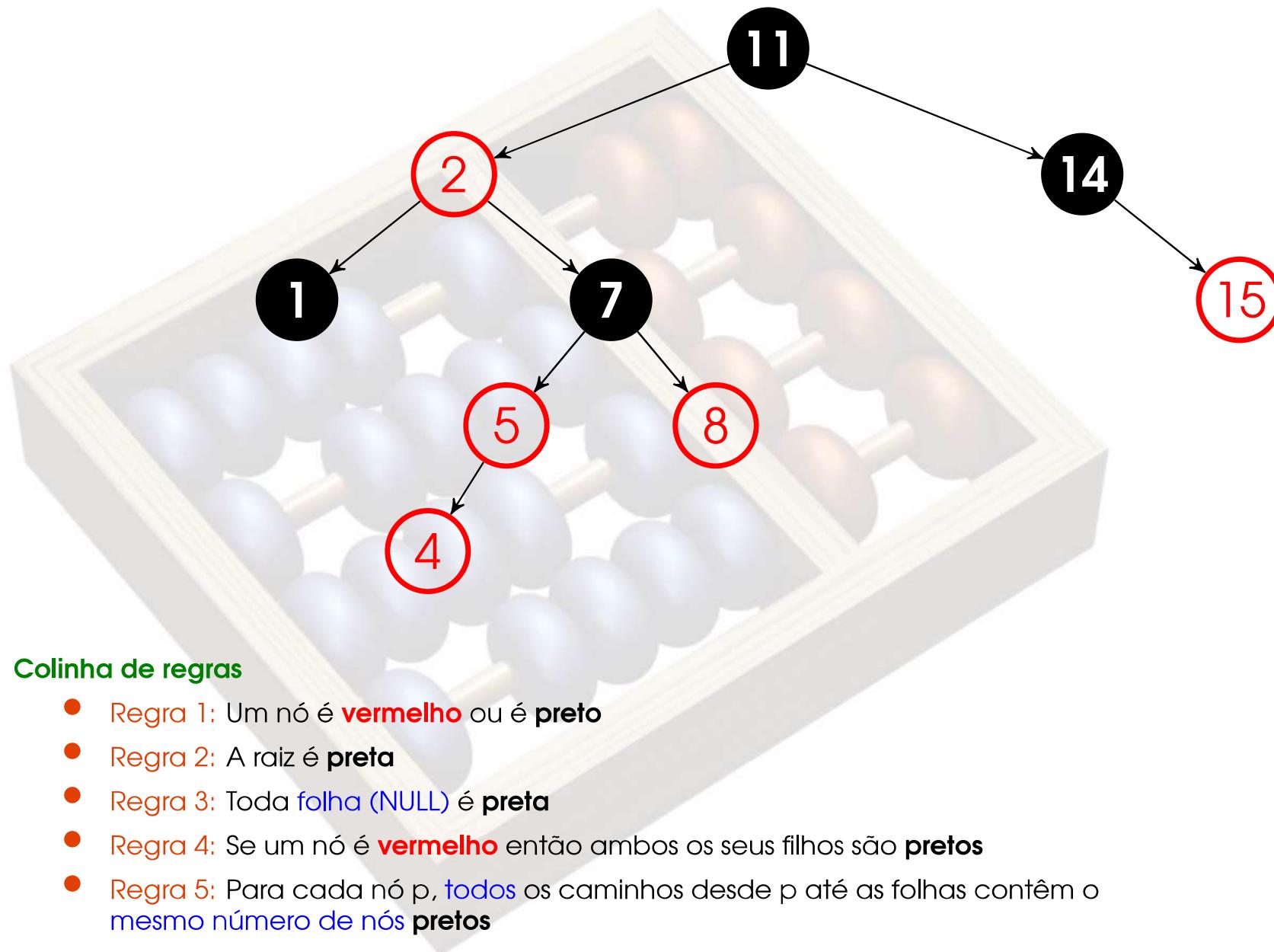
# Inserções em árvores rubro-negras

- Regra 1 resolvida, sempre insiro um nó com a **cor vermelha**
- E agora, qual **regra eu quebrei?**
- Melhor ainda, quais regras eu **poderia ter quebrado?**



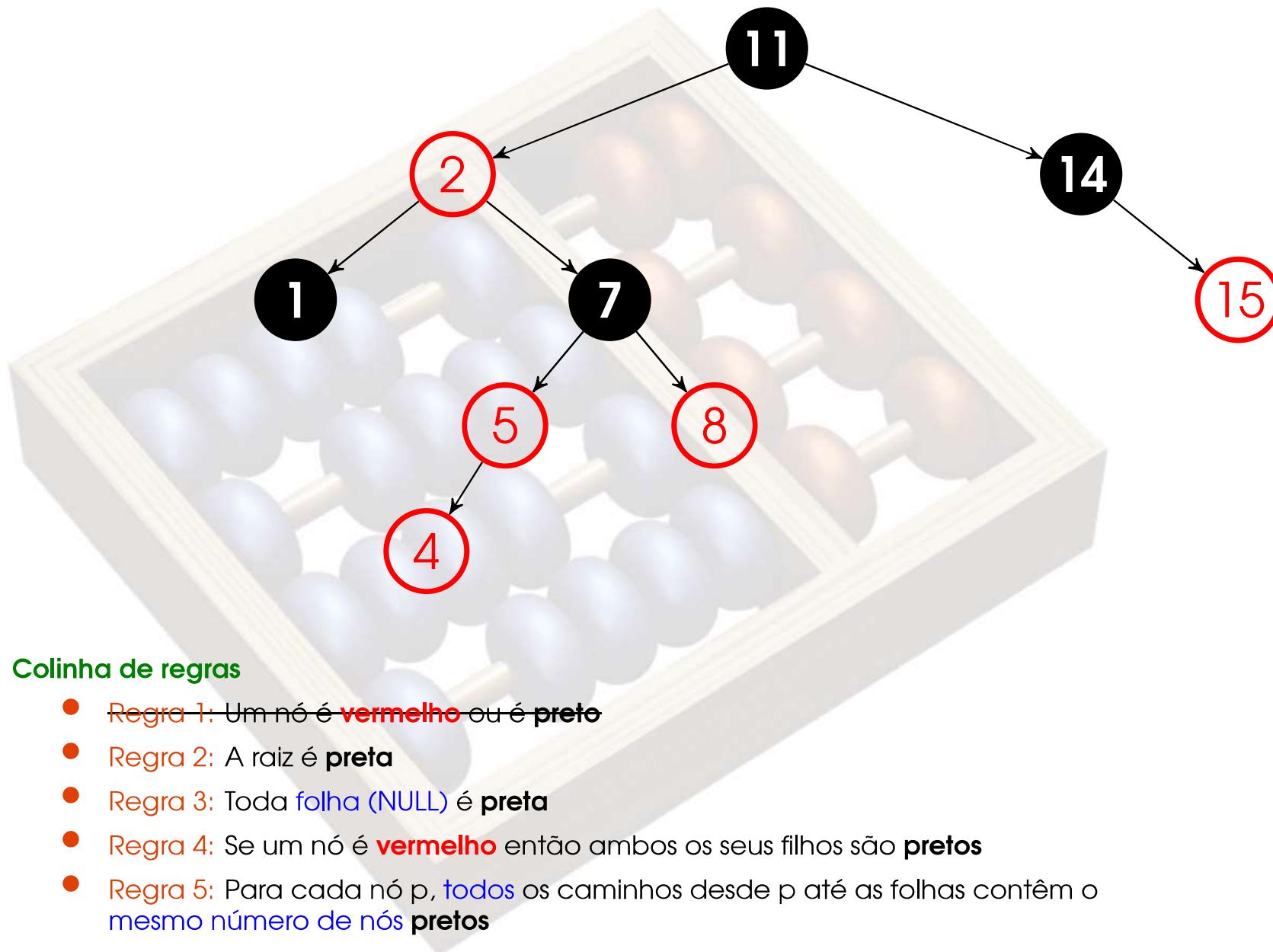
UNICAMP

# Avaliando quebras de Regras



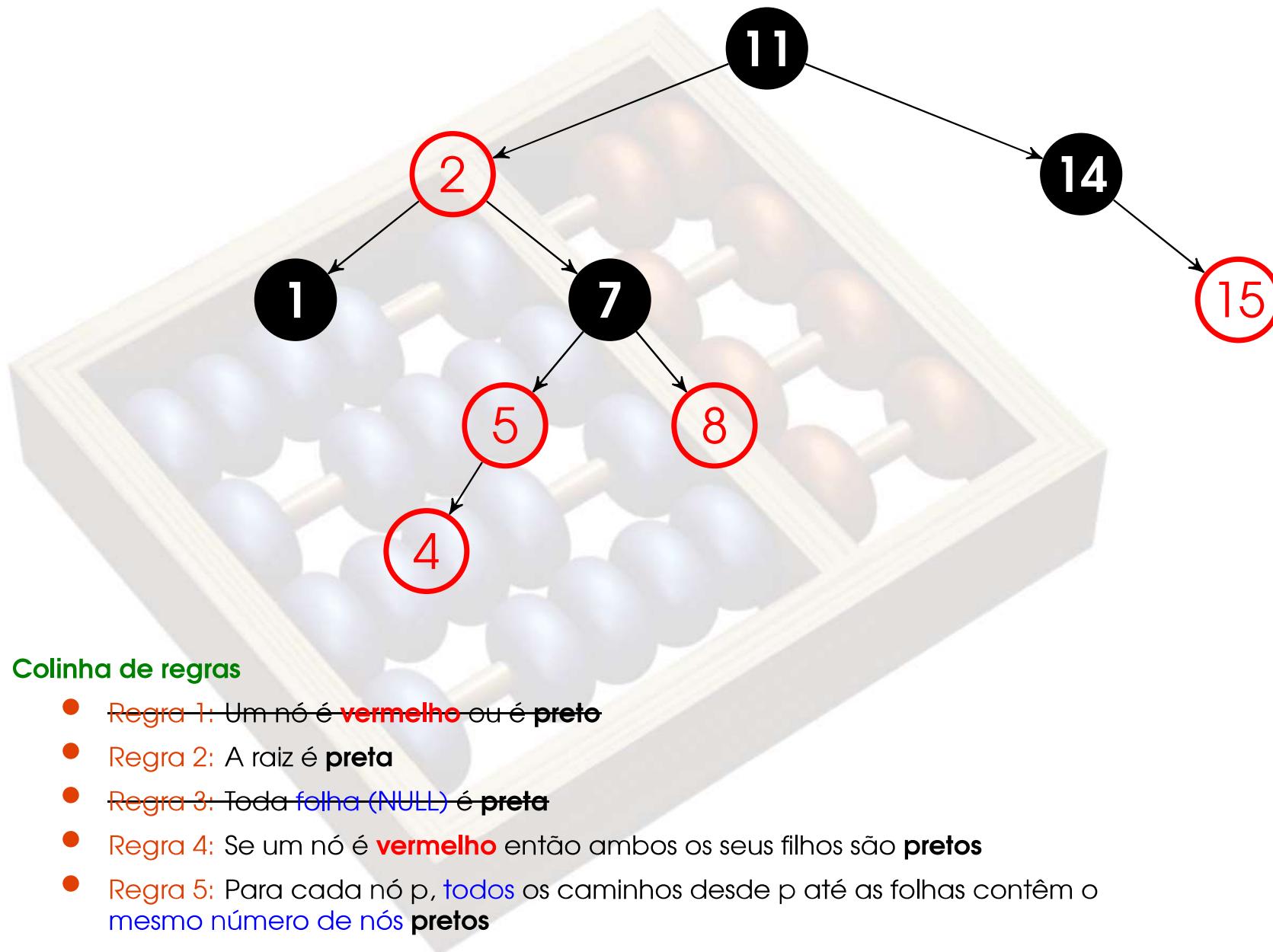
UNICAMP

# Avaliando quebras de Regras



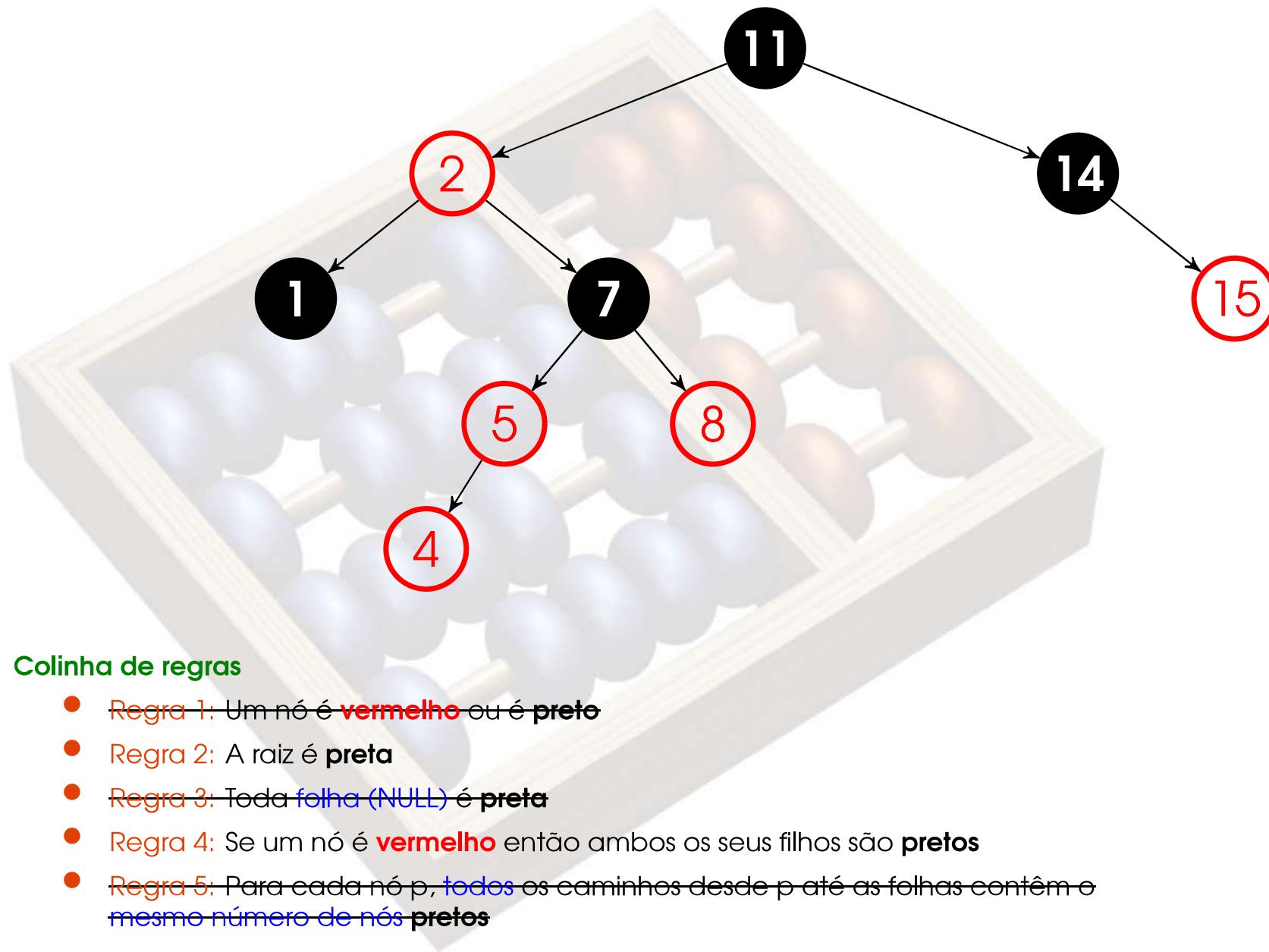
UNICAMP

# Avaliando quebras de Regras



UNICAMP

# Avaliando quebras de Regras



UNICAMP

# Inserções em árvores rubro-negras



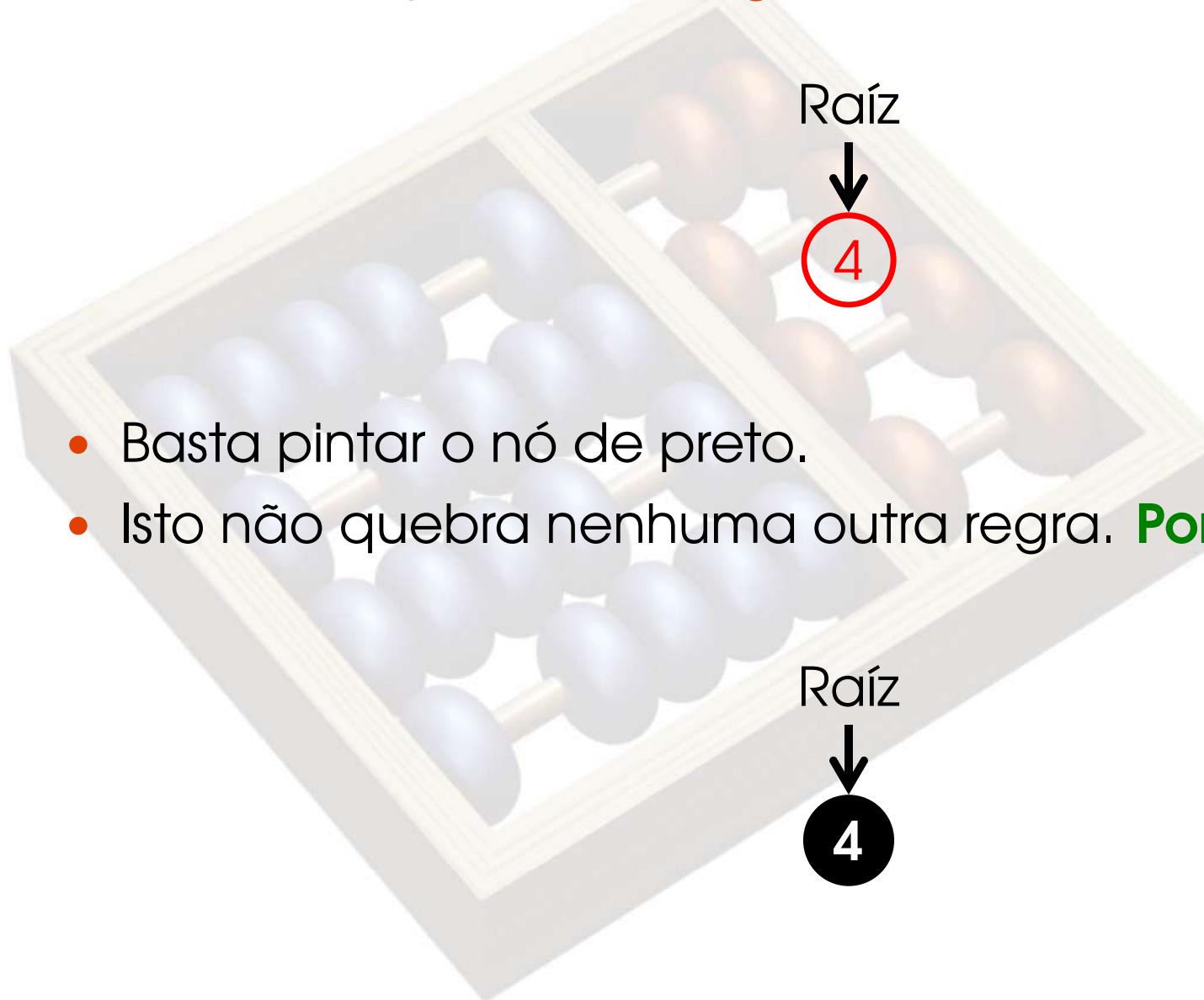
Ok! Sabemos que podemos **quebrar** apenas a **Regra 2** e a **Regra 4**. Vamos escrever o código.



UNICAMP

# Quebra na regra 2

Resolvendo a quebra na Regra 2

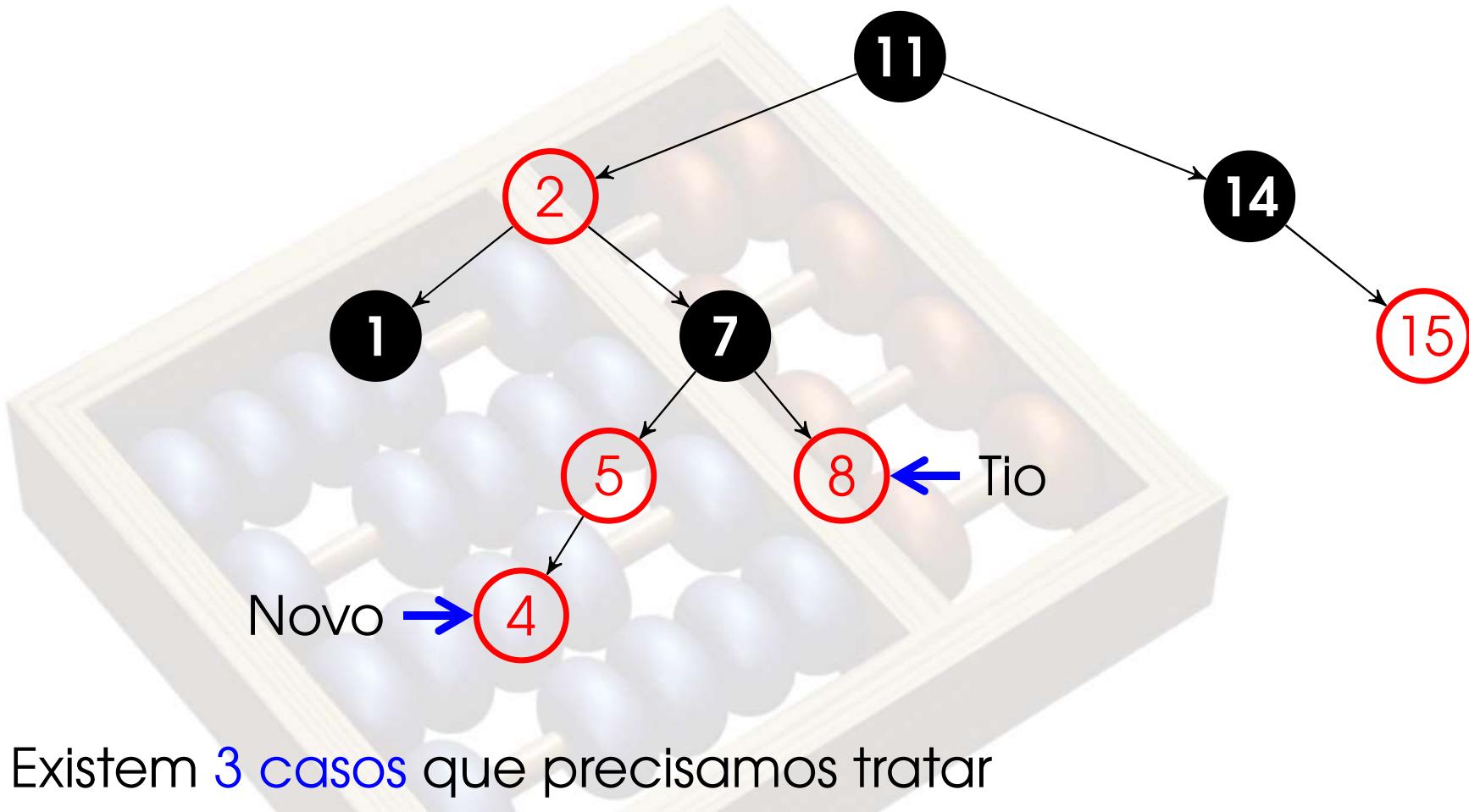


- Basta pintar o nó de preto.
- Isto não quebra nenhuma outra regra. **Por que?**



UNICAMP

# Inserções em árvores rubro-negras

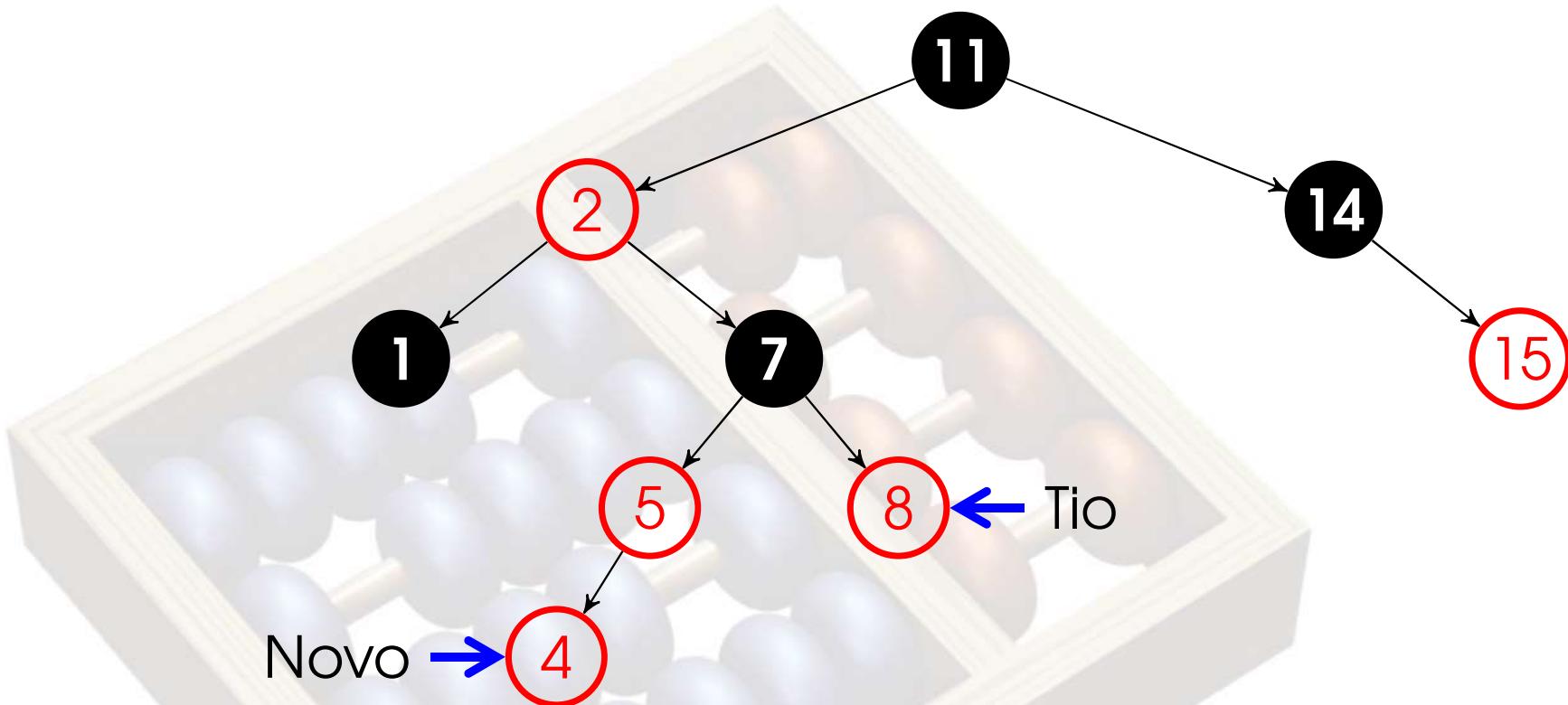


Existem 3 casos que precisamos tratar

- Caso 1 – O tio de novo é **vermelho**
- Caso 2 – O tio de novo é **preto** e novo é filho da direita
- Caso 3 – O tio de novo é **preto** e novo é filho da esquerda



# Inserções em árvores rubro-negras

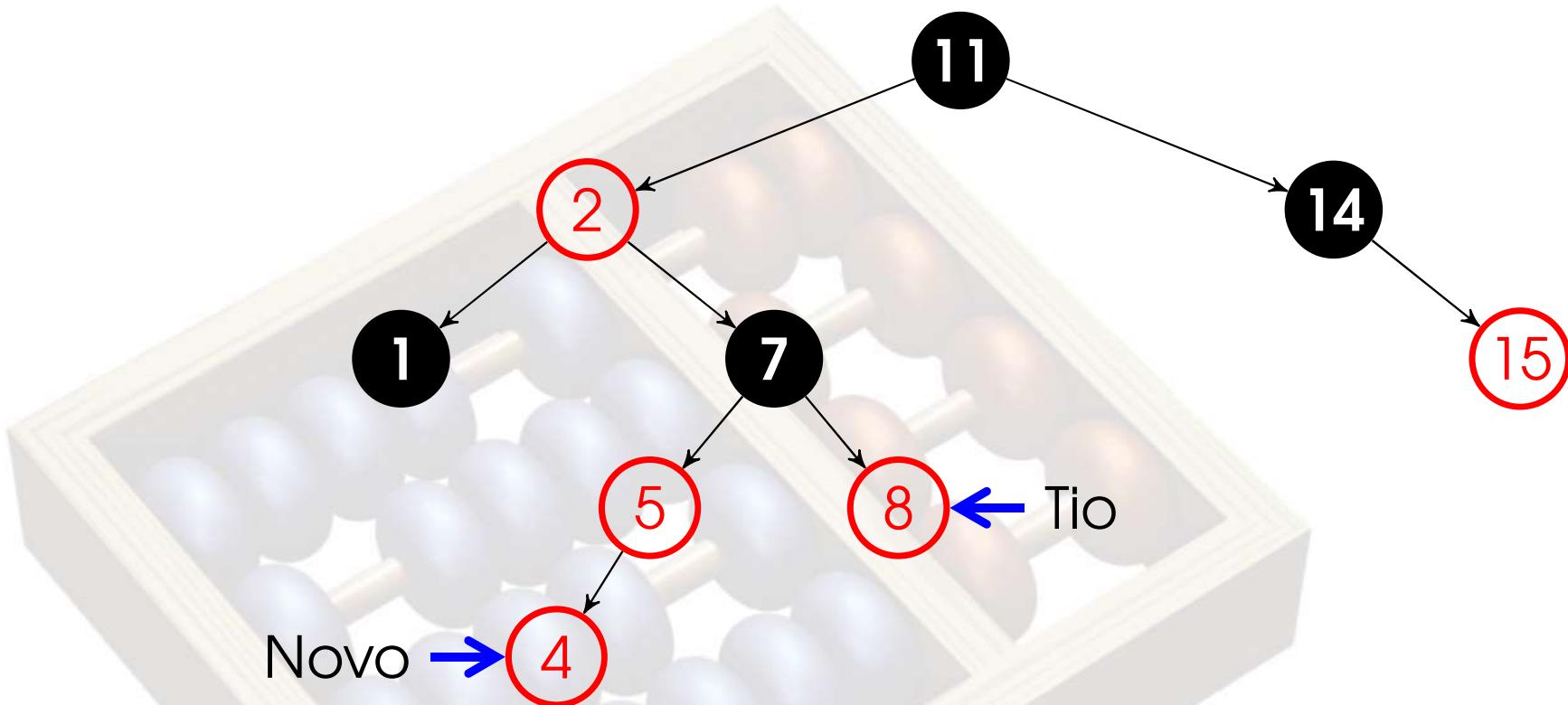


Existem 3 casos que precisamos tratar

- Caso 1 - O tio de novo é **vermelho**
- Caso 2 - O tio de novo é **preto** e novo é filho da direita
- Caso 3 - O tio de novo é **preto** e novo é filho da esquerda



# Inserções em árvores rubro-negras

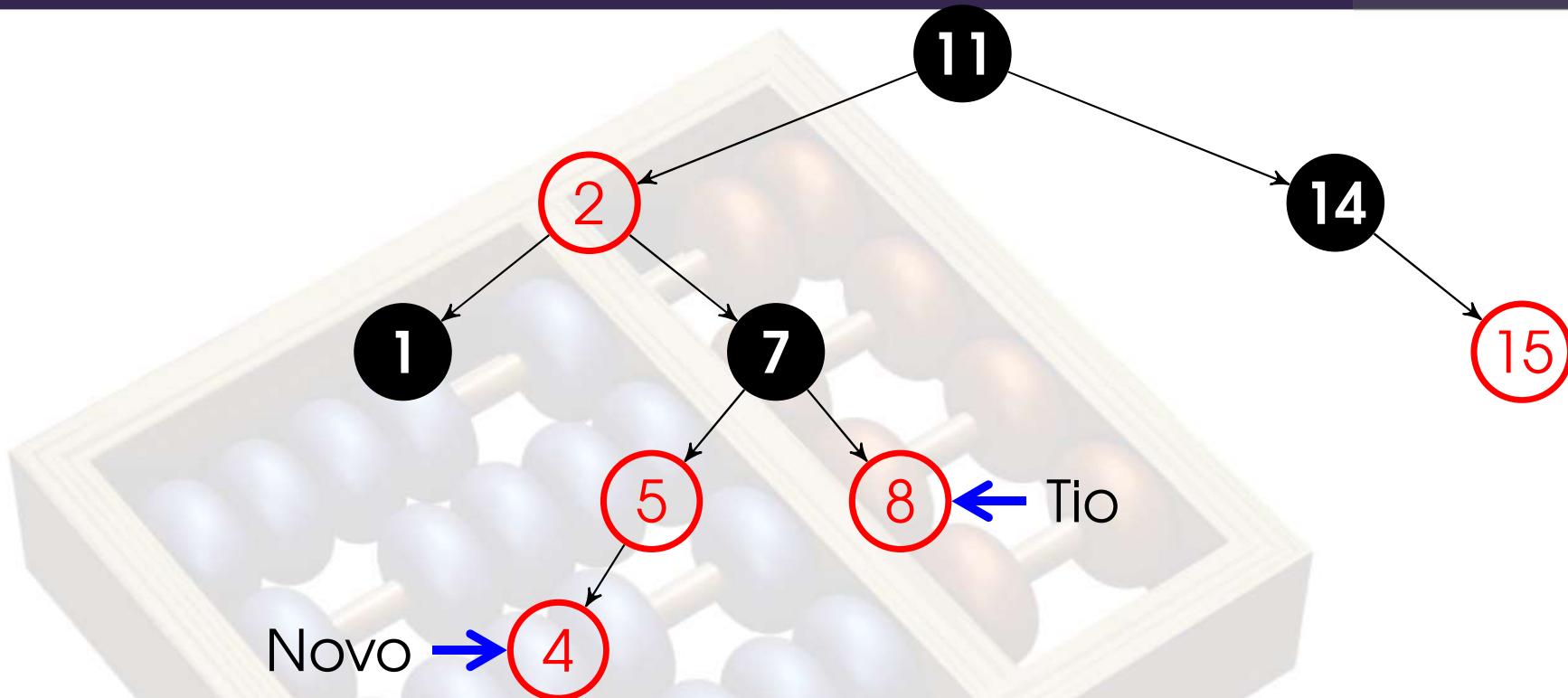


Existem 3 casos que precisamos tratar

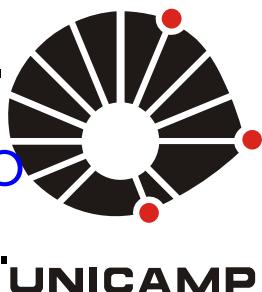
- Caso 1 - O tio de novo é **vermelho**
- Caso 2 - O tio de novo é **preto** e novo é filho da direita
- Caso 3 - O tio de novo é **preto** e novo é filho da esquerda



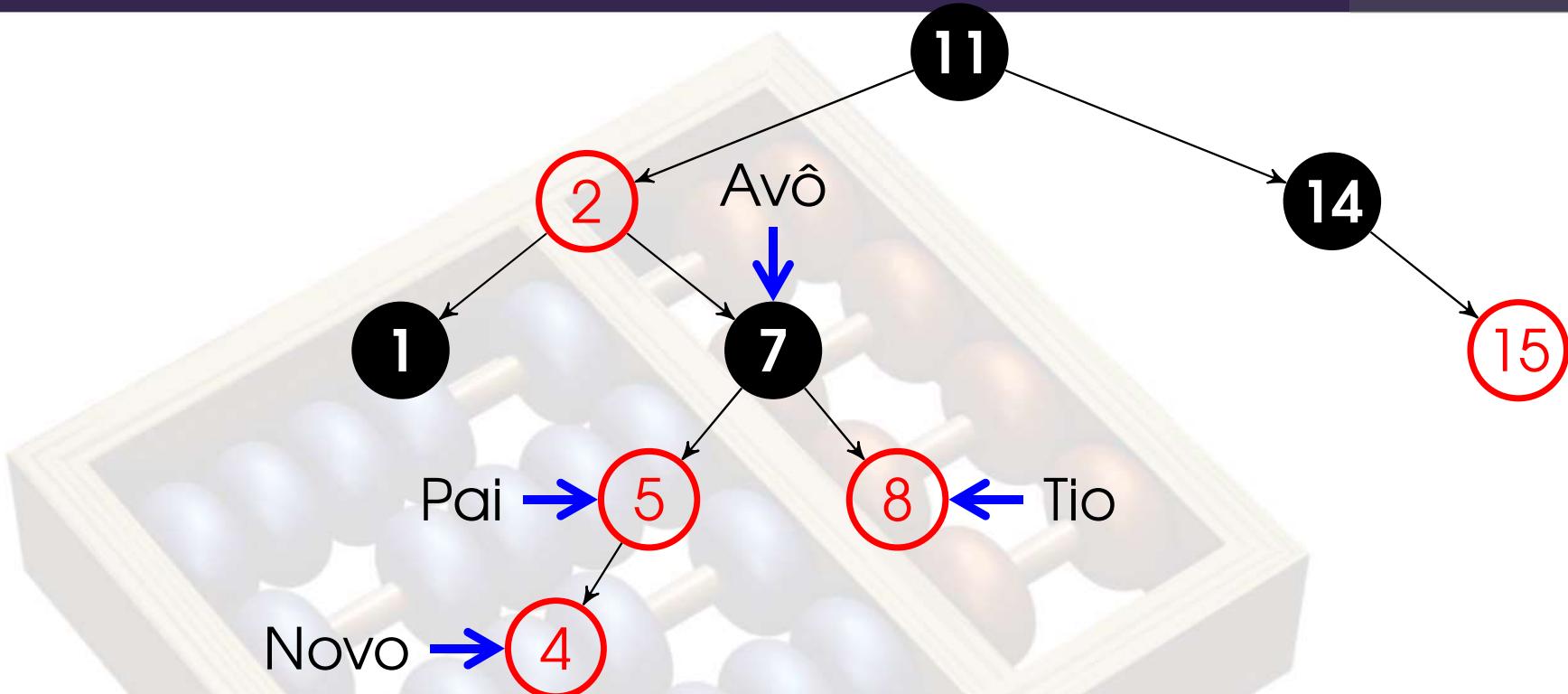
# Inserções em árvores rubro-negras



- O **pulo do gato** é que quando resolvemos um caso, ou **criamos um outro** em outro nó **mais alto** da árvore, ou fazemos algumas rotações que resolvem o problema.
- Como a árvore tem  $O(\lg n)$  níveis, precisamos fazer **no máximo  $O(\lg n)$  consertos** até bater na raiz e terminar.



# Inserções – Caso 1



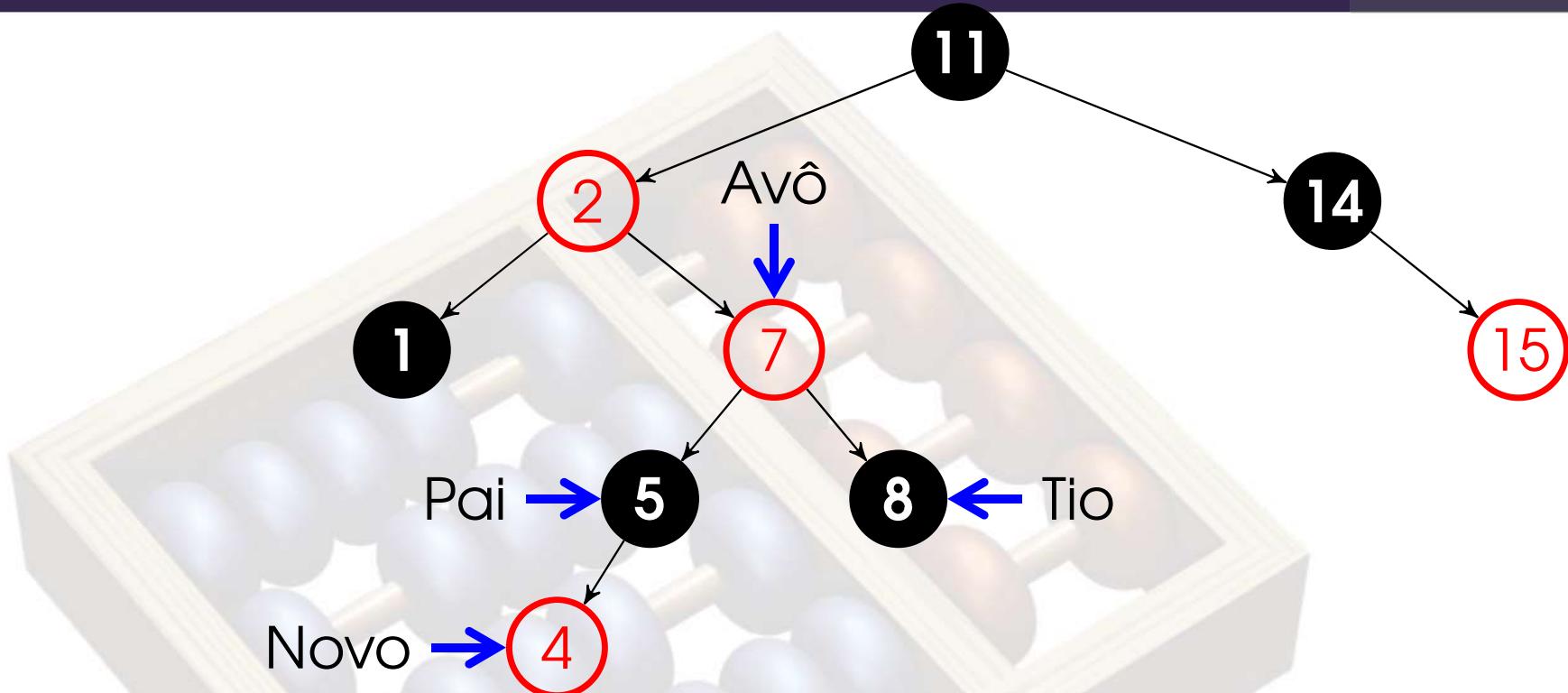
Caso 1 – O **tio** de **novo** é **vermelho**

- Se o **tio** é **vermelho** o **avô** obrigatoriamente é **preto** (PQ?)
- Troque a cor do **pai** e **tio** para **preto**
- Troque a cor do **avô** para **vermelho**
- Neste ponto, consertamos o problema no novo, mas possivelmente estragamos o **avô**



UNICAMP

# Inserções – Caso 1



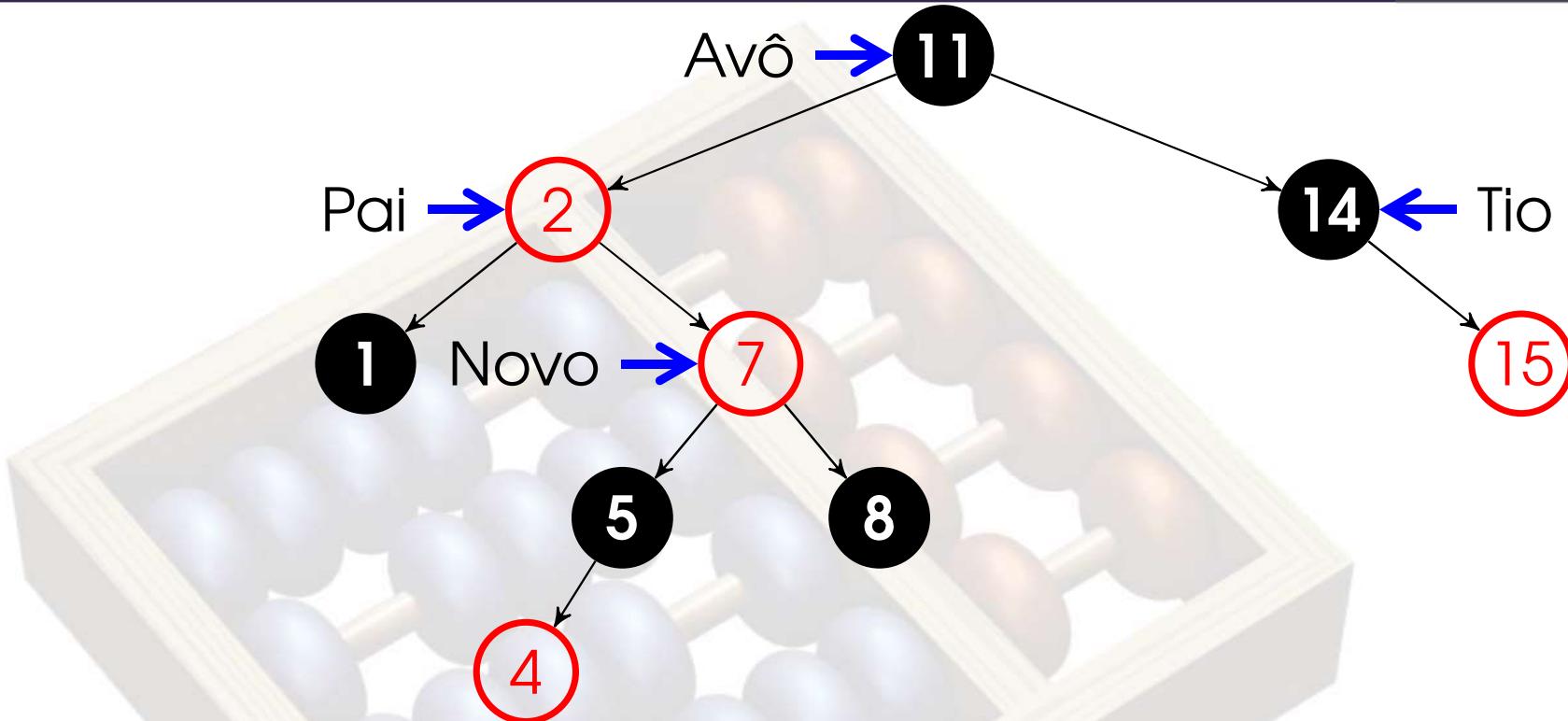
Caso 1 – O **tio** de **novo** é **vermelho**

- Se o **tio** é **vermelho** o **avô** obrigatoriamente é **preto** (PQ?)
- Troque a cor do **pai** e **tio** para **preto**
- Troque a cor do **avô** para **vermelho**
- Neste ponto, consertamos o problema no novo, mas possivelmente estragamos o **avô**



UNICAMP

# Inserções em árvores rubro-negras



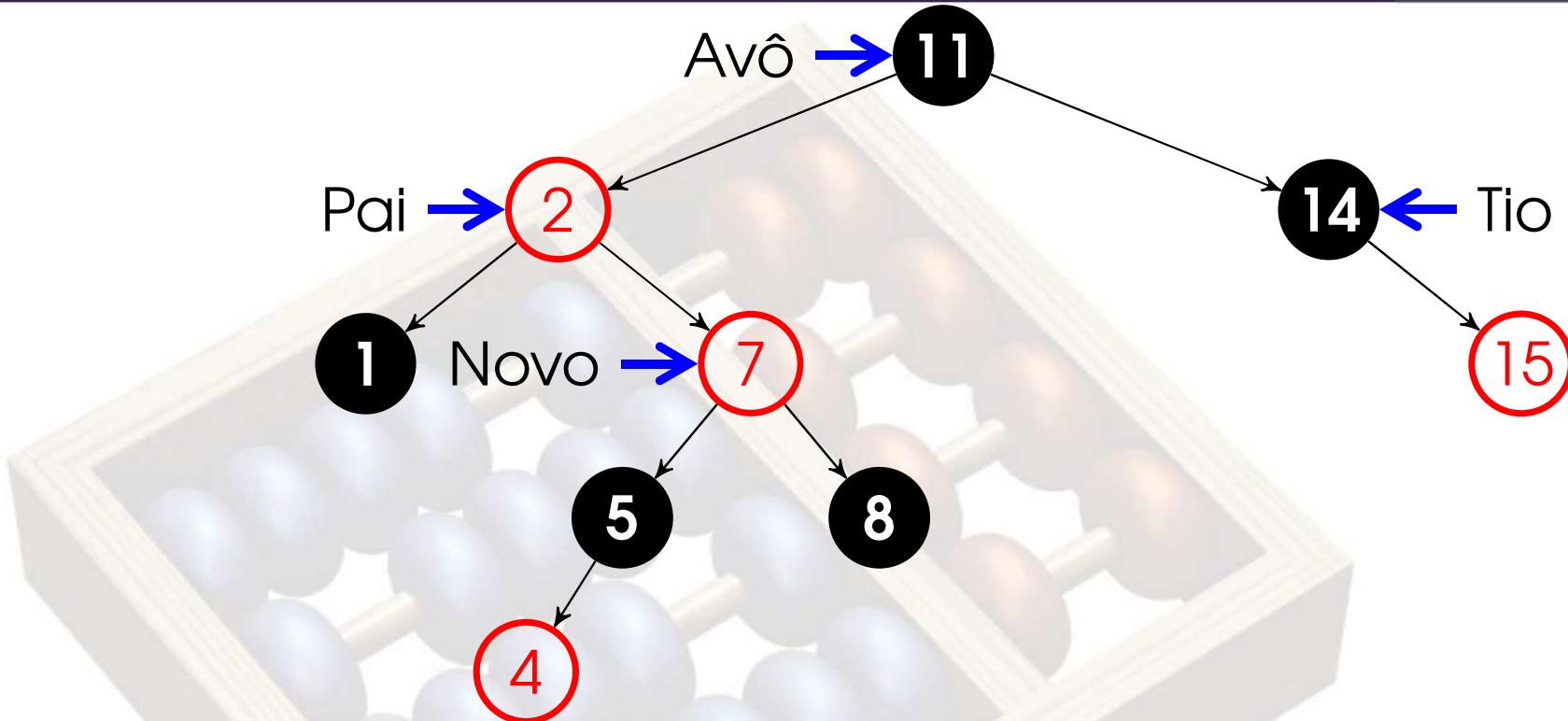
Continuamos o conserto, agora tomando como referência o antigo **avô**, o nó com o valor 7.

- Caso 1 – O tio de novo é **vermelho**
- Caso 2 – O tio de novo é **preto** e novo é filho da direita
- Caso 3 – O tio de novo é **preto** e novo é filho da esquerda



UNICAMP

# Inserções em árvores rubro-negras



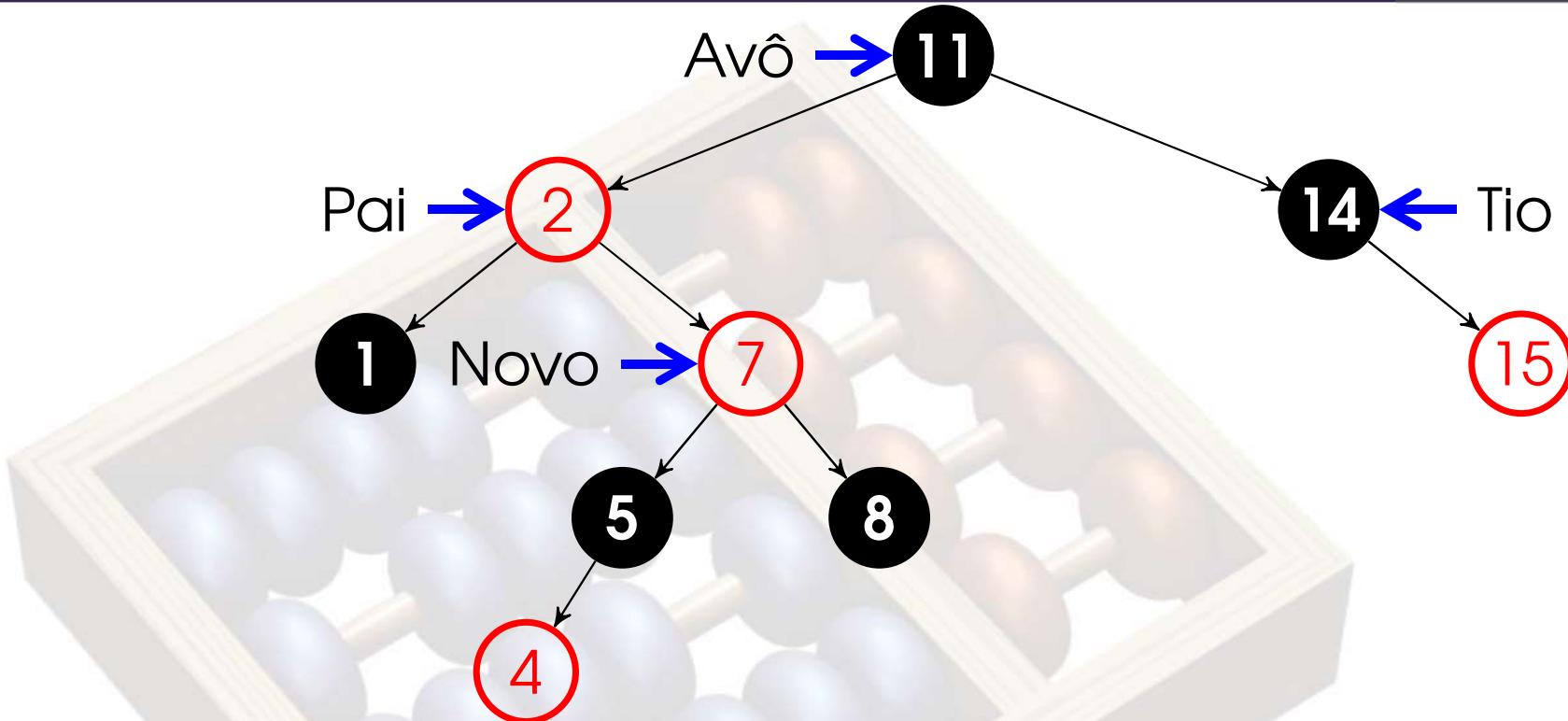
Continuamos o conserto, agora tomando como referência o antigo **avô**, o nó com o valor 7.

- Caso 1 – O tio de novo é **vermelho**
- Caso 2 – O tio de novo é **preto** e novo é filho da direita
- Caso 3 – O tio de novo é **preto** e novo é filho da esquerda



UNICAMP

# Inserções em árvores rubro-negras



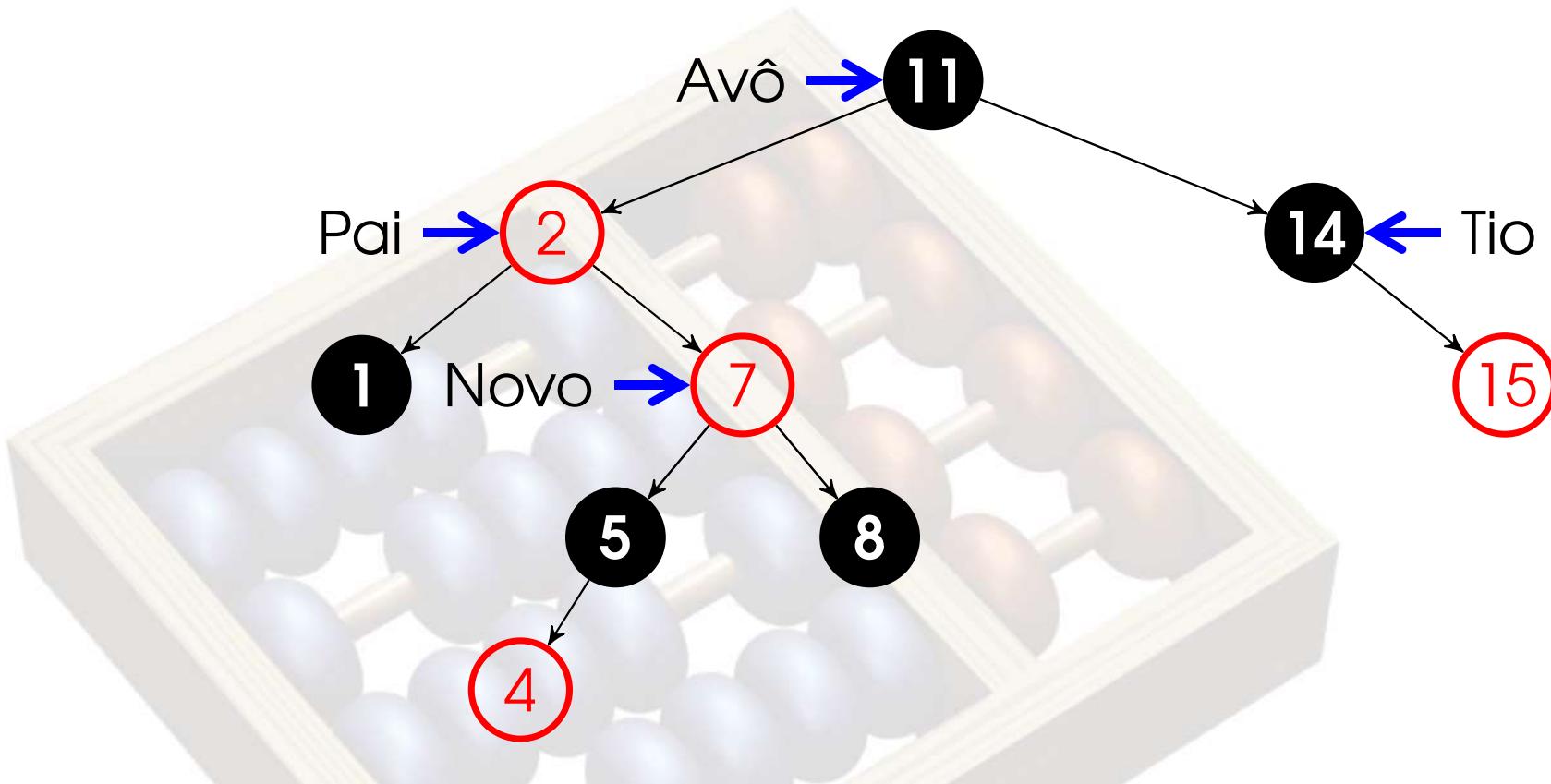
Continuamos o conserto, agora tomando como referência o antigo **avô**, o nó com o valor 7.

- Caso 1 – O tio de novo é **vermelho**
- Caso 2 – O tio de novo é **preto** e novo é filho da direita
- Caso 3 – O tio de novo é **preto** e novo é filho da esquerda



UNICAMP

# Inserções – Caso 2

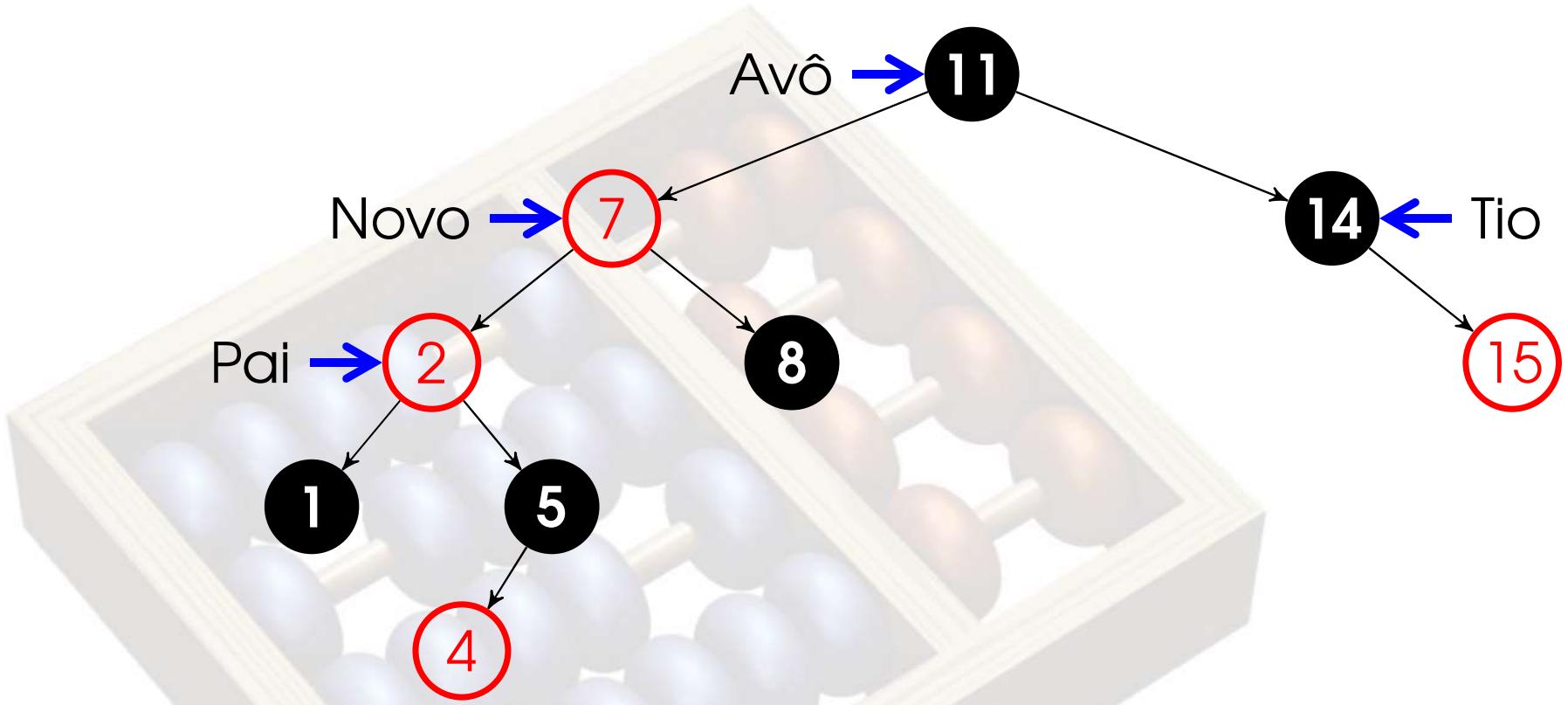


Caso 2 – O tio de novo é **preto** e novo é filho da direita

- Executa uma **rotação à esquerda** tendo como pivô o **pai**
- Neste ponto, consertamos o problema no novo, mas possivelmente estragamos o **pai**



# Inserções – Caso 2

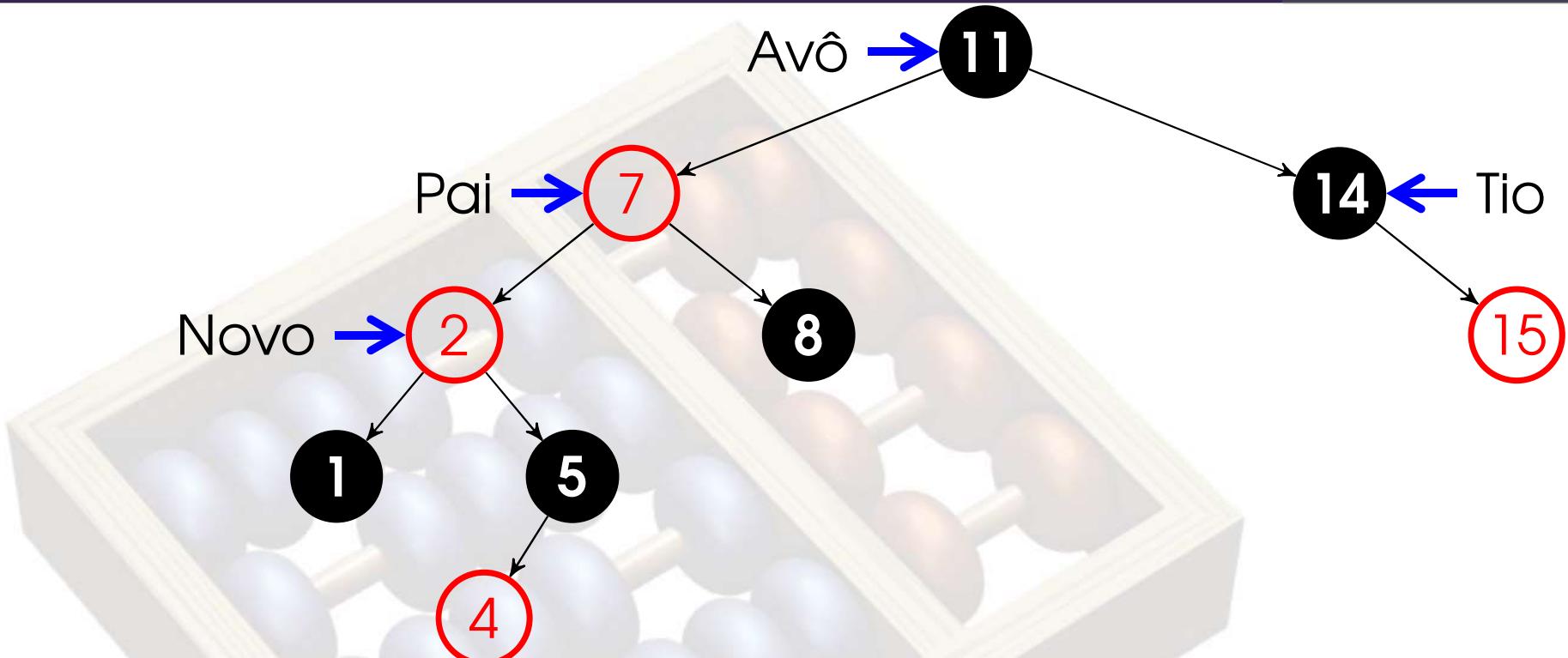


Caso 2 – O tio de novo é **preto** e novo é filho da direita

- Executa uma **rotação à esquerda** tendo como pivô o **pai**
- Neste ponto, consertamos o problema no novo, mas possivelmente estragamos o **pai**



# Inserções em árvores rubro-negras

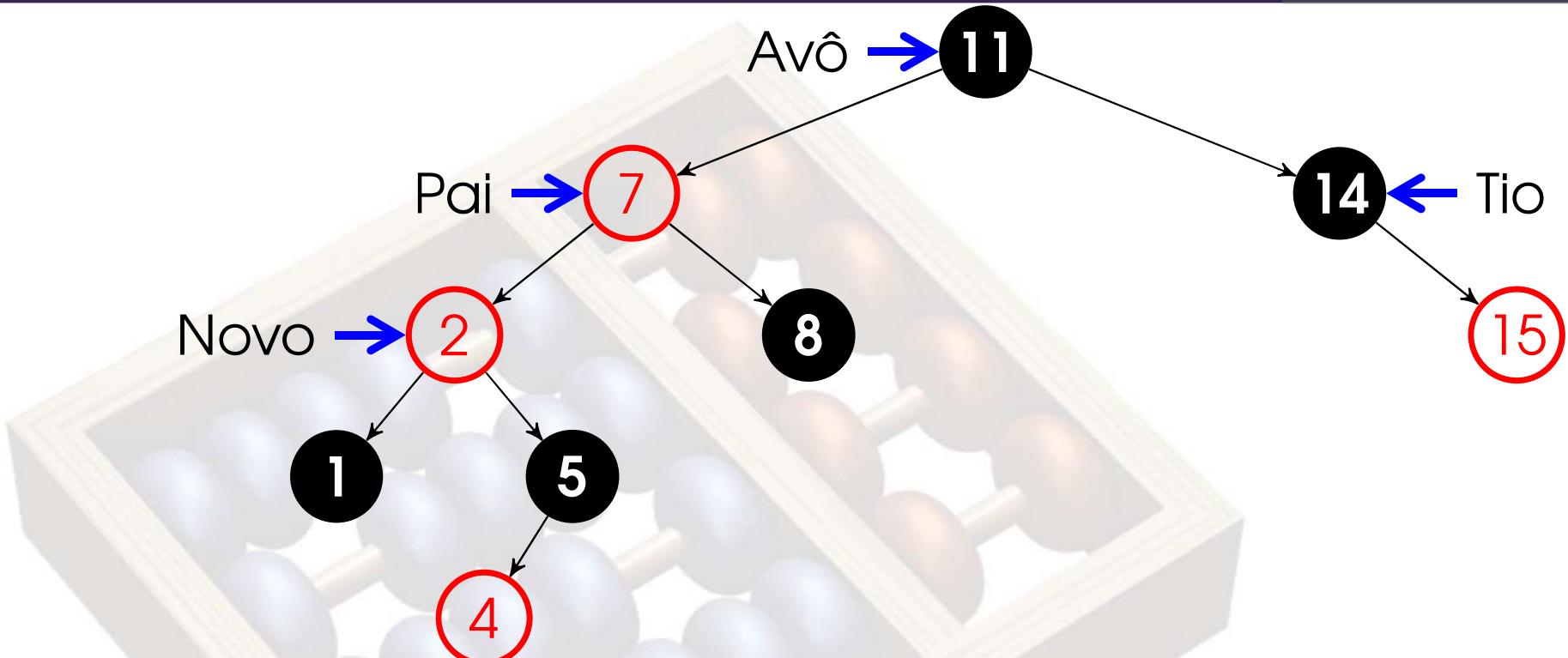


Continuamos o conserto, agora tomando como referência o antigo **pai**, o nó com o valor 2.

- Caso 1 – O tio de novo é **vermelho**
- Caso 2 – O tio de novo é **preto** e novo é filho da direita
- Caso 3 – O tio de novo é **preto** e novo é filho da esquerda



# Inserções em árvores rubro-negras

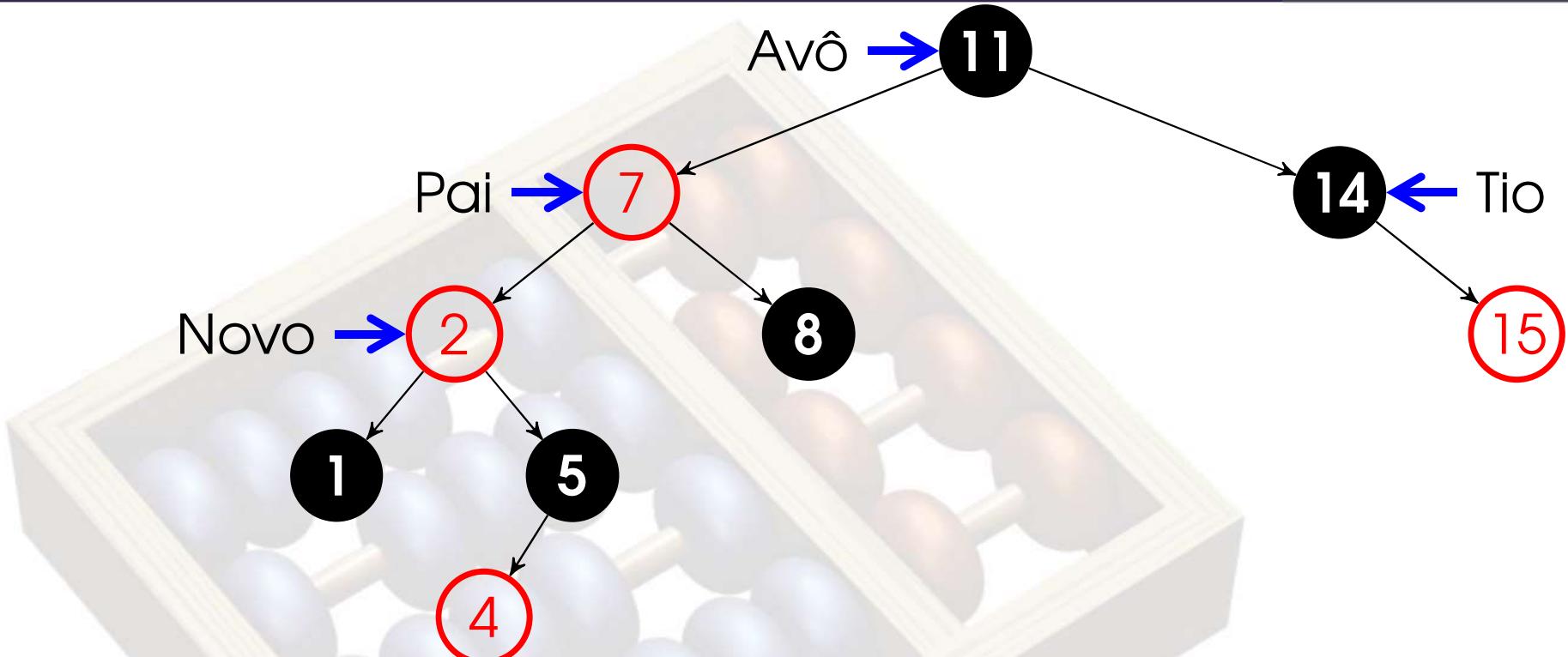


Continuamos o conserto, agora tomando como referência o antigo **pai**, o nó com o valor 2.

- Caso 1 – O tio de novo é **vermelho**
- Caso 2 – O tio de novo é **preto** e novo é filho da direita
- Caso 3 – O tio de novo é **preto** e novo é filho da esquerda



# Inserções em árvores rubro-negras



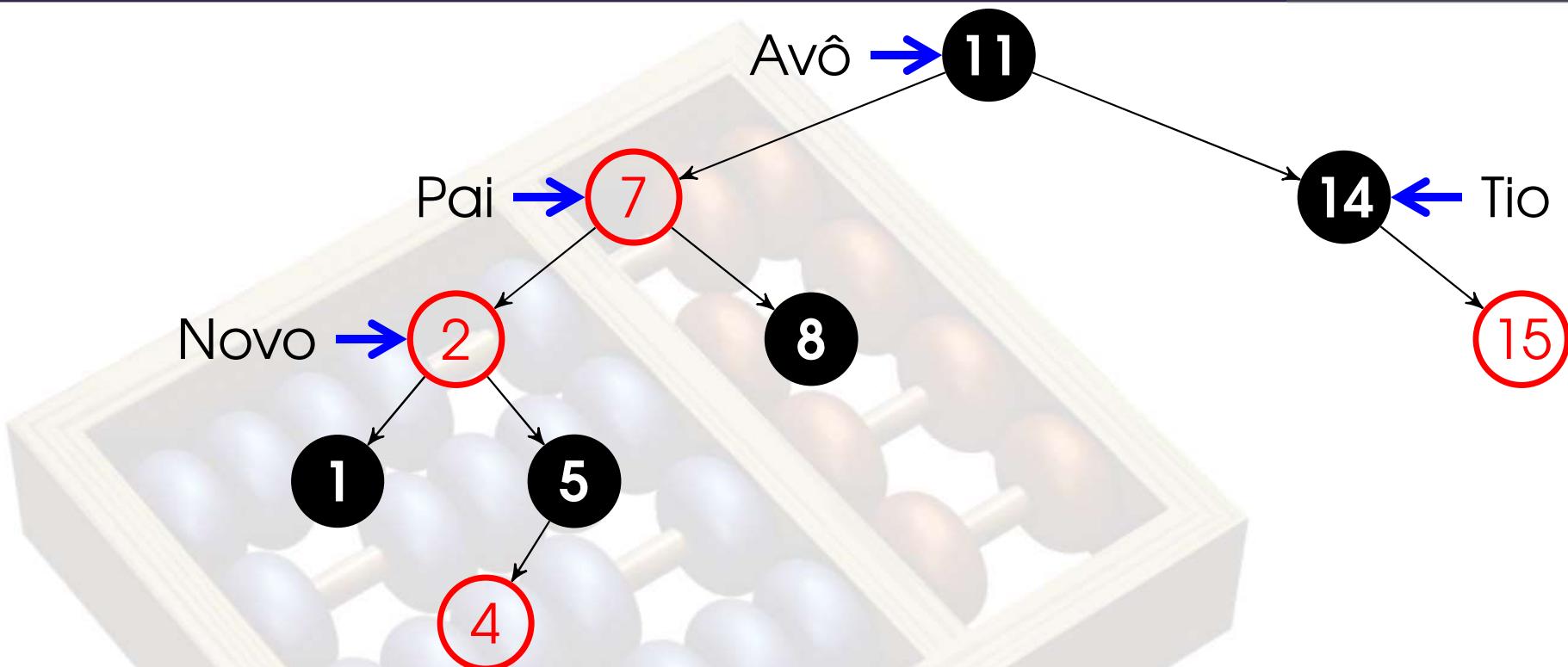
Continuamos o conserto, agora tomando como referência o antigo **pai**, o nó com o valor 2.

- Caso 1 – O tio de novo é **vermelho**
- Caso 2 – O tio de novo é **preto** e novo é filho da direita
- Caso 3 – O tio de novo é **preto** e novo é filho da esquerda



UNICAMP

# Inserções – Caso 3

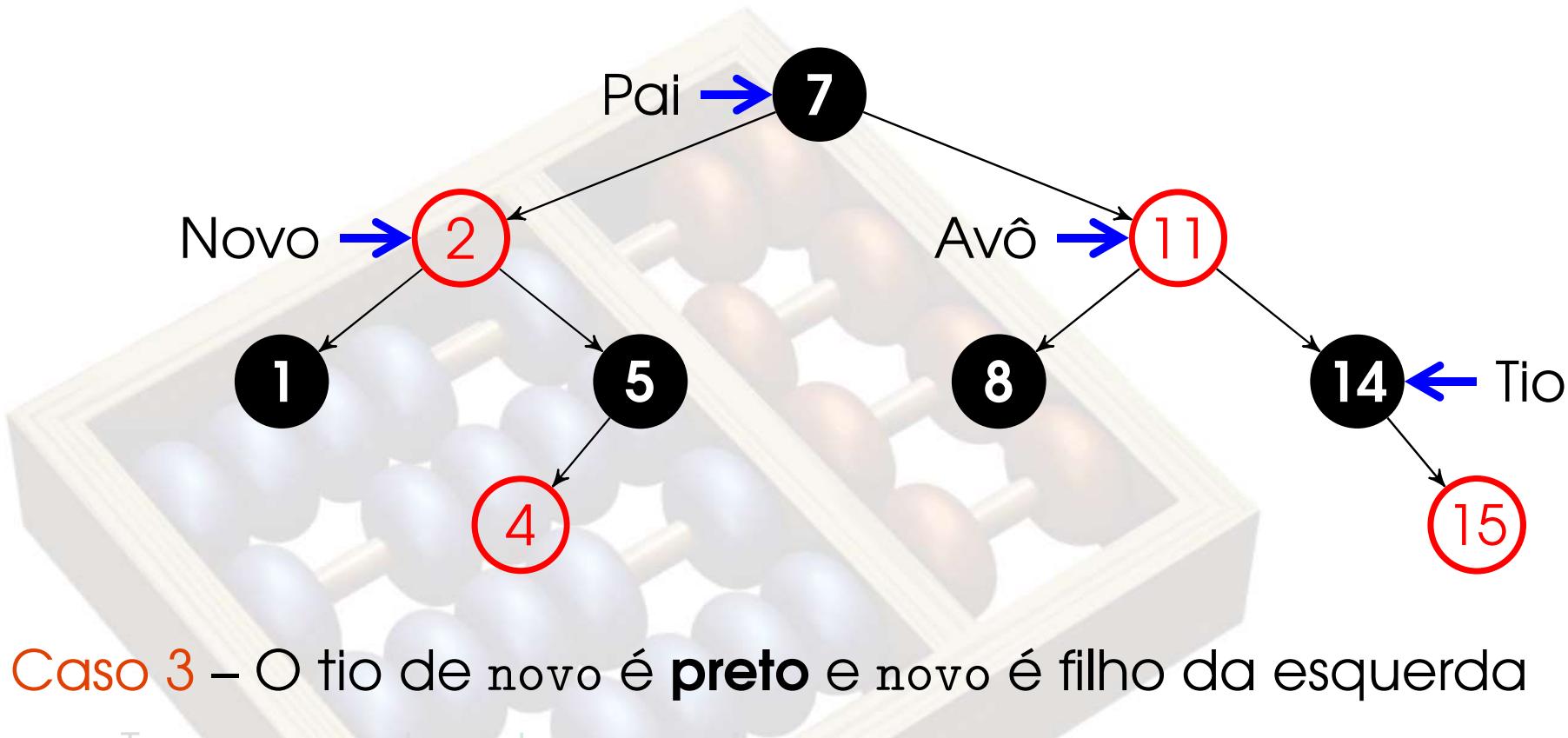


Caso 3 – O tio de novo é **preto** e novo é filho da esquerda

- Troca a cor do **pai** para **preto**
- Troca a cor do **avô** para **vermelho**
- Executa uma **rotação à direita** tendo como pivô o **avô**
- Neste ponto a árvore voltou a ser uma árvore rubro-negra válida



# Inserções – Caso 3



Caso 3 – O tio de novo é **preto** e novo é filho da esquerda

- Troca a cor do pai para preto
- Troca a cor do avô para **vermelho**
- Executa uma **rotação à direita** tendo como pivô o avô
- Neste ponto a árvore voltou a ser uma árvore rubro-negra válida



# Casos 2 e 3 – o tio do nó inserido é preto

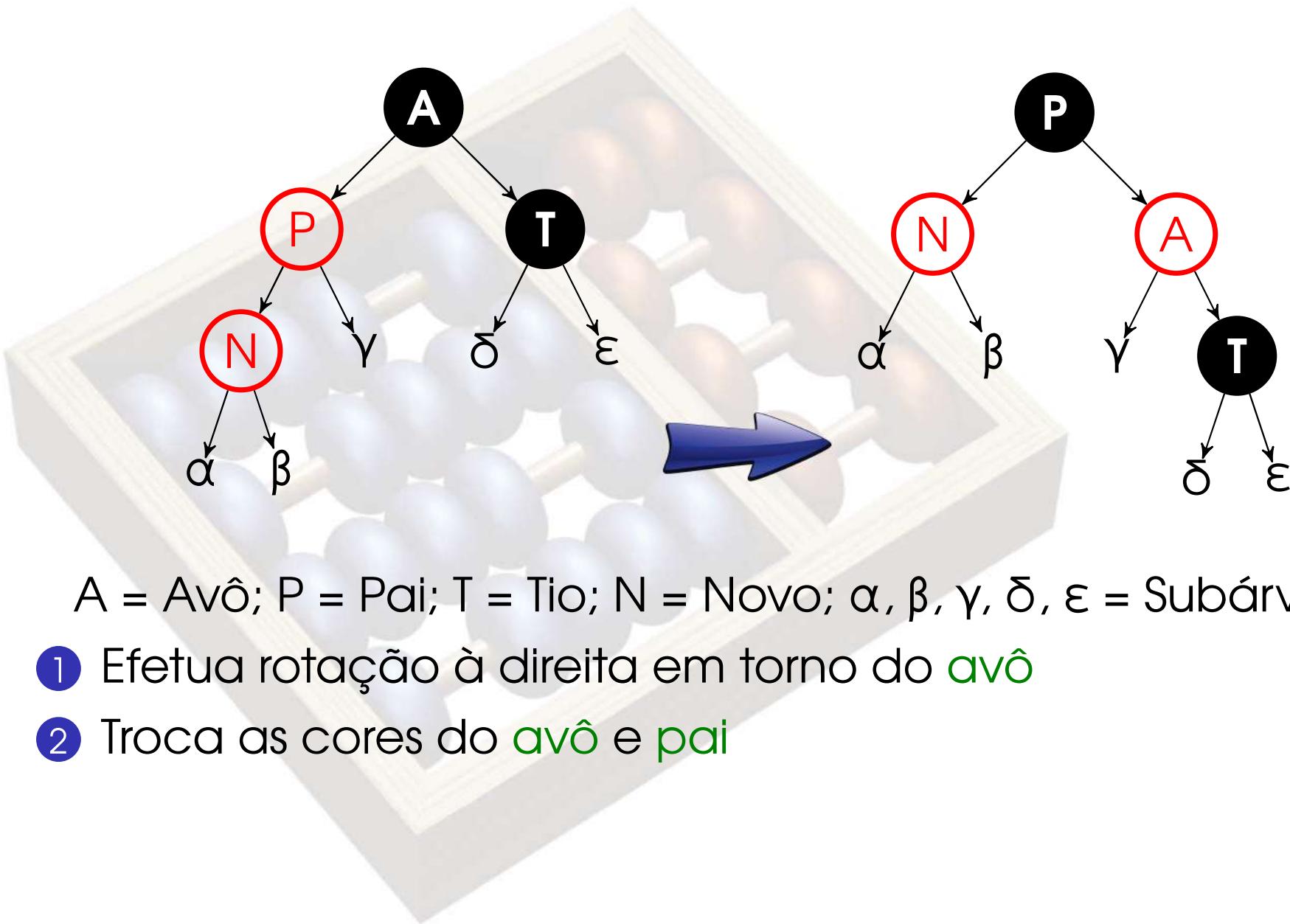
No Caso 2 e no Caso 3

- Apenas consideramos um lado da árvore, o caso onde o **pai** é **filho esquerdo** do **avô**.
- A resolução dos demais casos é simétrica
- Se o **tio** é preto, então existem 4 configurações possíveis:
  - 1 Configuração **Esq-Esq** (**pai** é filho esquerdo do **avô** e **novo** é filho esquerdo de **pai**)
  - 2 Configuração **Esq-Dir** (**pai** é filho esquerdo do **avô** e **novo** é filho direito de **pai**)
  - 3 Configuração **Dir-Dir** (espelho da 1)
  - 4 Configuração **Dir-Esq** (espelho da 2)

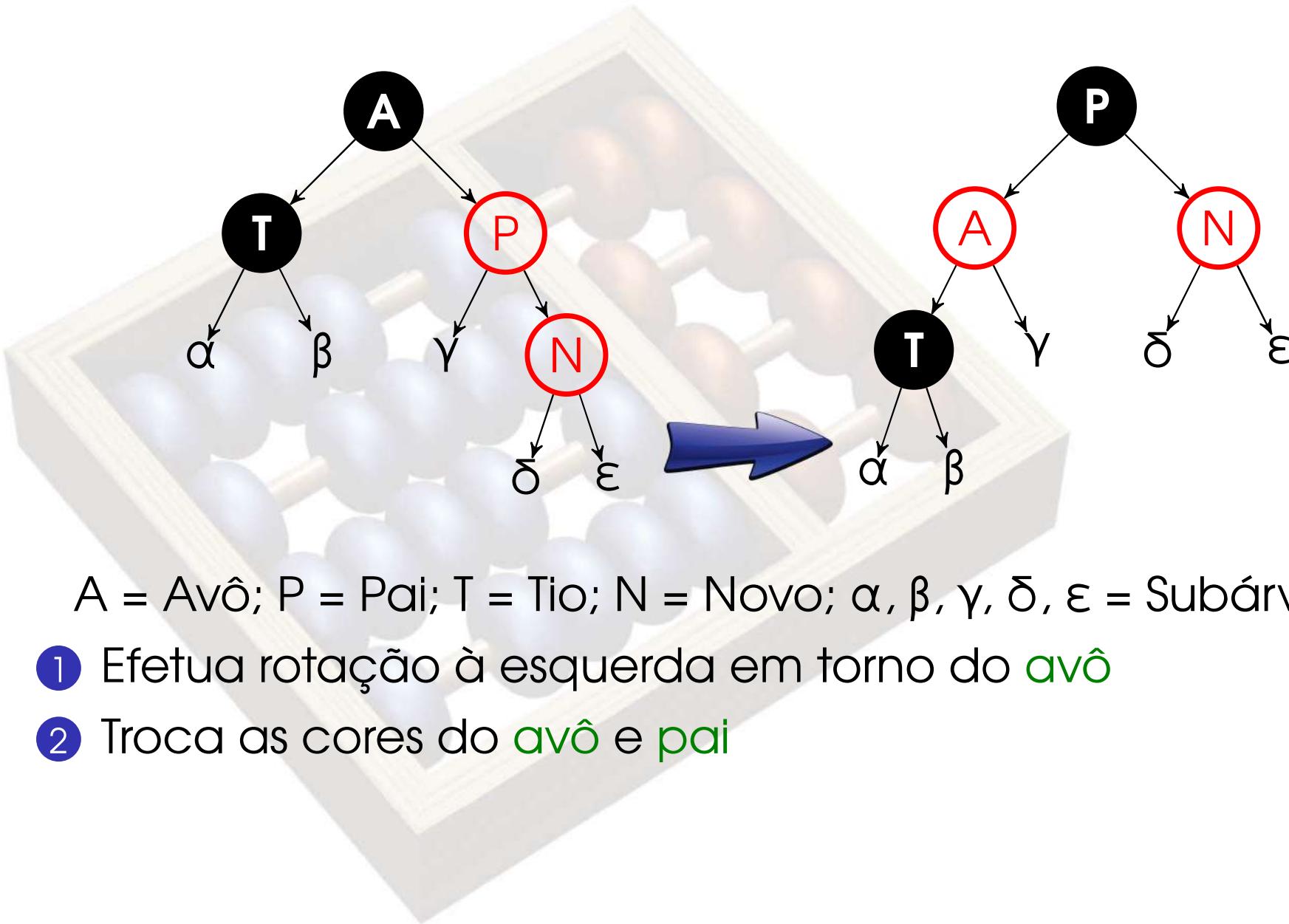


**UNICAMP**

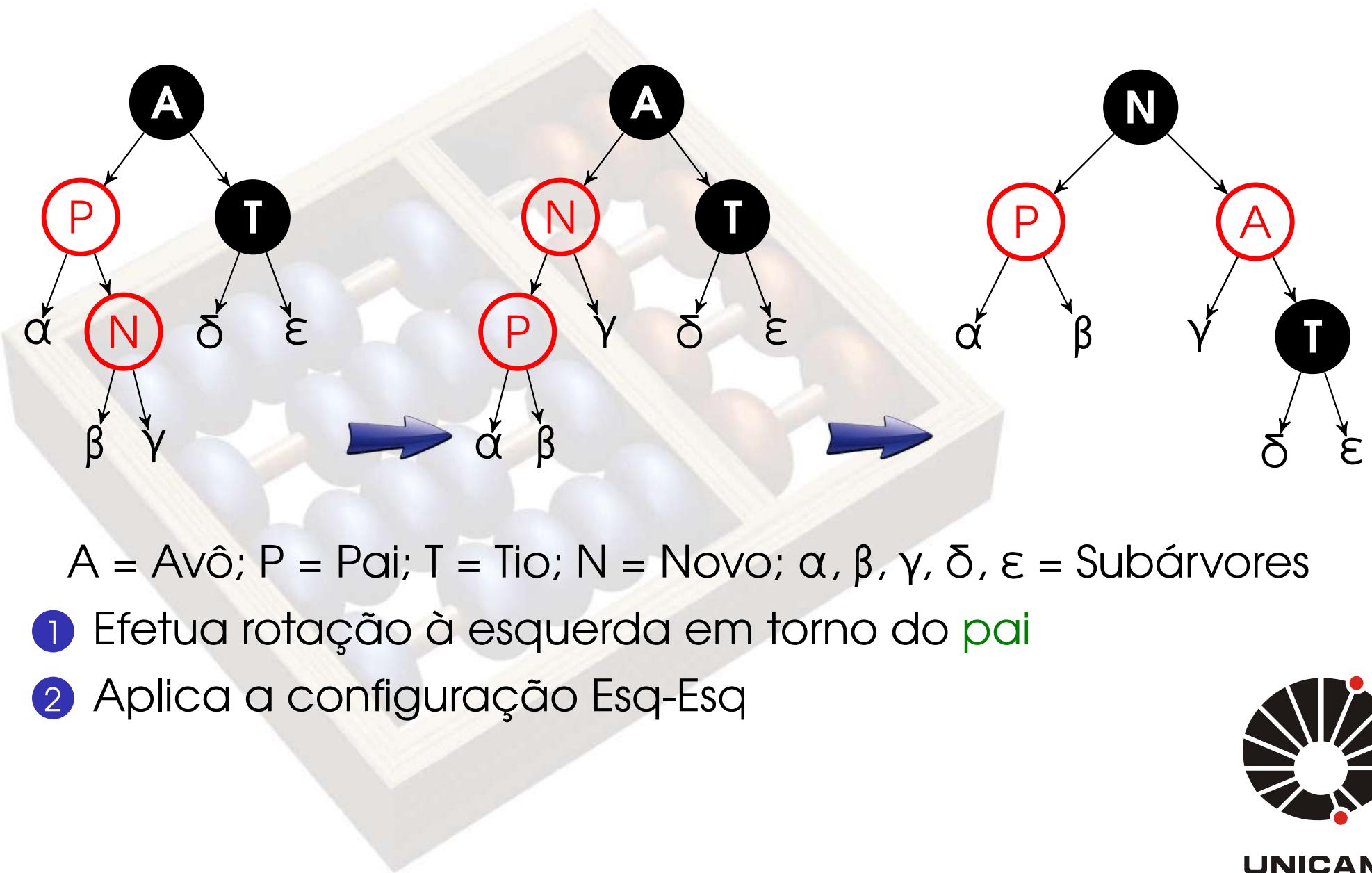
# Configuração Esq-Esq



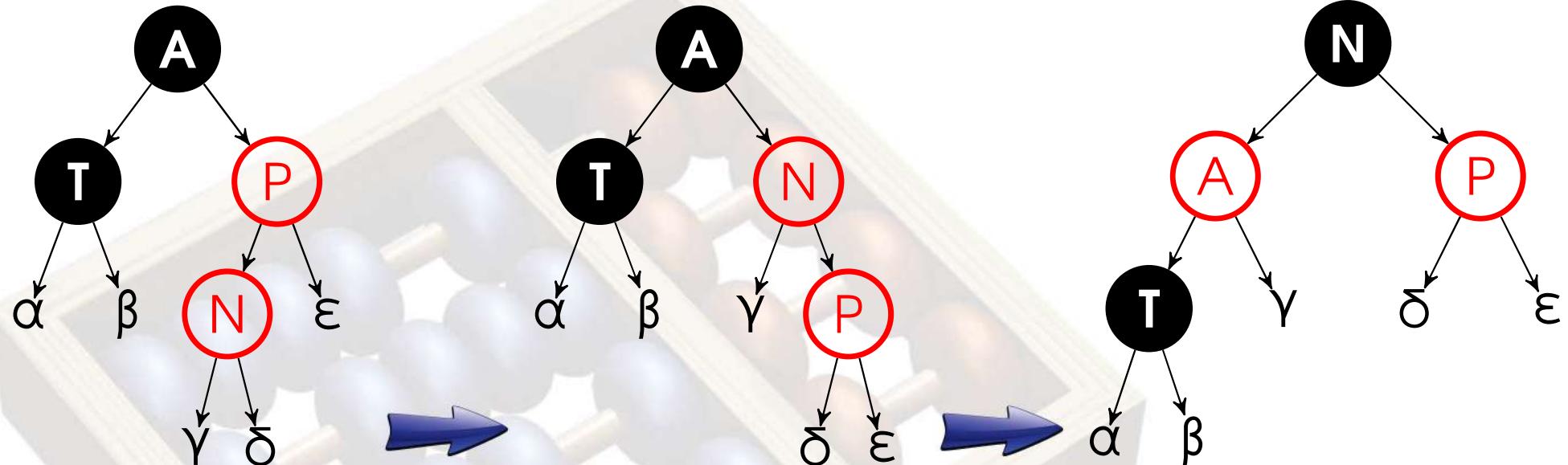
# Configuração Dir-Dir



# Configuração Esq-Dir



# Configuração Dir-Esq



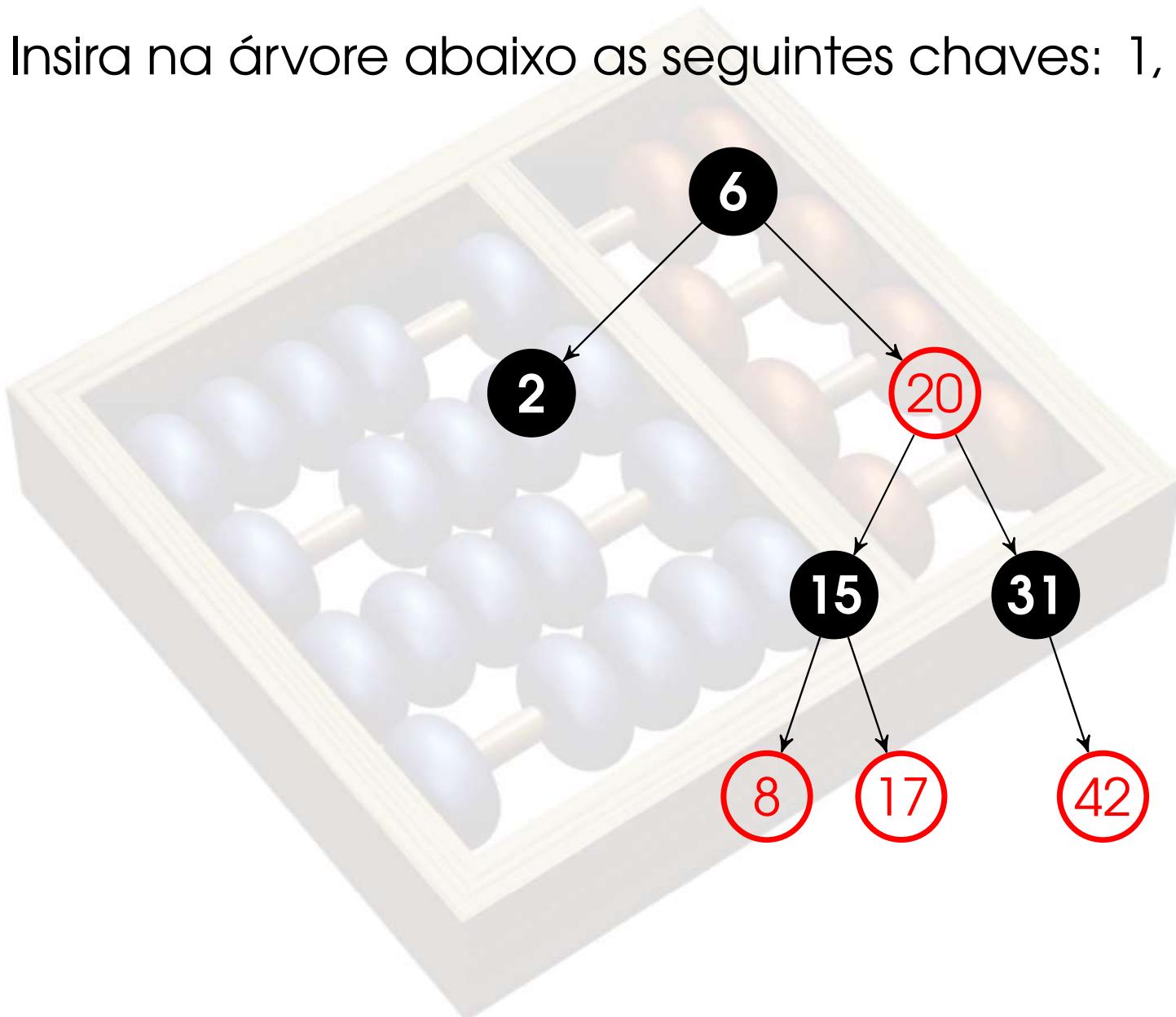
A = Avô; P = Pai; T = Tio; N = Novo;  $\alpha, \beta, \gamma, \delta, \varepsilon$  = Subárvore

- 1 Efetua rotação à direita em torno do **pai**
- 2 Aplica a configuração Dir-Dir



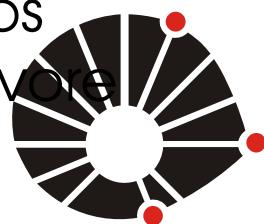
# Exercício

Insira na árvore abaixo as seguintes chaves: 1, 5, 19



# Remoções em árvores rubro-negras

- Assim como no caso da **inserção**, nós utilizaremos **rotações** e **recolorações** para manter as propriedades da árvore rubro negra
  - Contudo, a **remoção** de nós de uma árvore rubro-negra exige **um pouco mais de trabalho**
- Durante a **inserção**, baseamos as nossas operações de readaptação **na cor do tio**
  - Já durante a **remoção** nós nos baseamos na **cor do irmão** do nó para decidir qual caso aplicar.
- Durante a **inserção** o principal problema que enfrentamos era um duplo nó vermelho
  - Durante a **remoção**, se retirarmos um nó preto, estamos “estrugando” a **propriedade de altura de preto** da árvore



UNICAMP

# Revisão – Remoção em Árv. de Busca

**Problema:** dada uma árvore binária de busca  $r$  e uma chave  $k$ , remover o nó com chave  $k$  (se existir) de  $r$  de modo que árvore binária resultante continue sendo uma árvore binária de busca.

Como vimos nas aulas passadas, isto é **mais difícil do que a inserção**. Para resolver este problema tratamos a princípio a **remoção de uma raiz** e depois partimos para a **resolução de um problema mais geral**.



**UNICAMP**

# Revisão – Remoção em Árv. de Busca

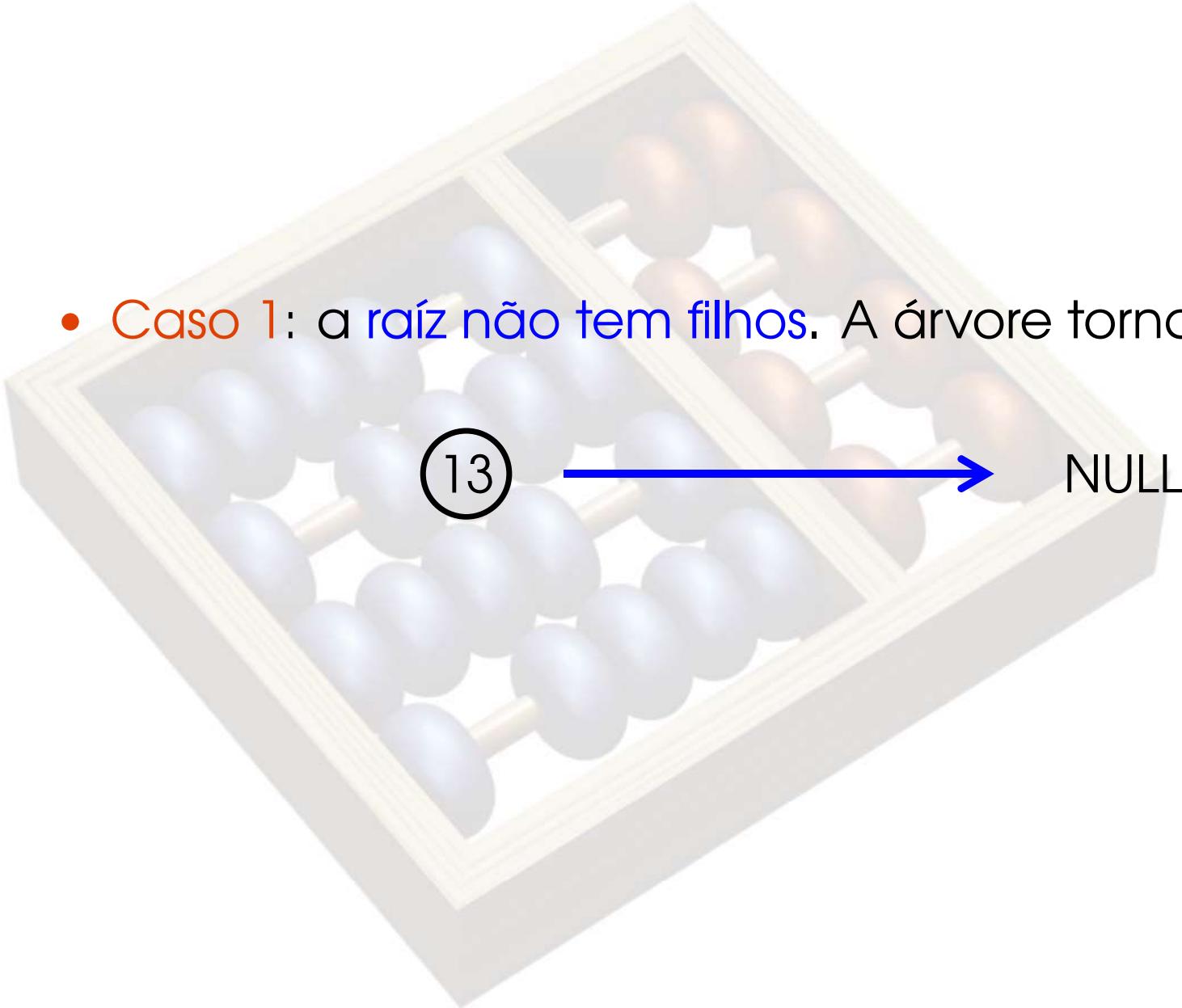
- Caso 1: a raiz não tem filhos. A árvore torna-se vazia.
- Caso 2: a raiz tem um único filho. Seu filho torna-se a nova raiz.
- Caso 3: a raiz tem dois filhos. Neste caso, tomamos o nó que sucede (ou precede) a raiz no percurso inordem (e-r-d) como a nova raiz.



UNICAMP

# Revisão – Remoção em Árv. de Busca

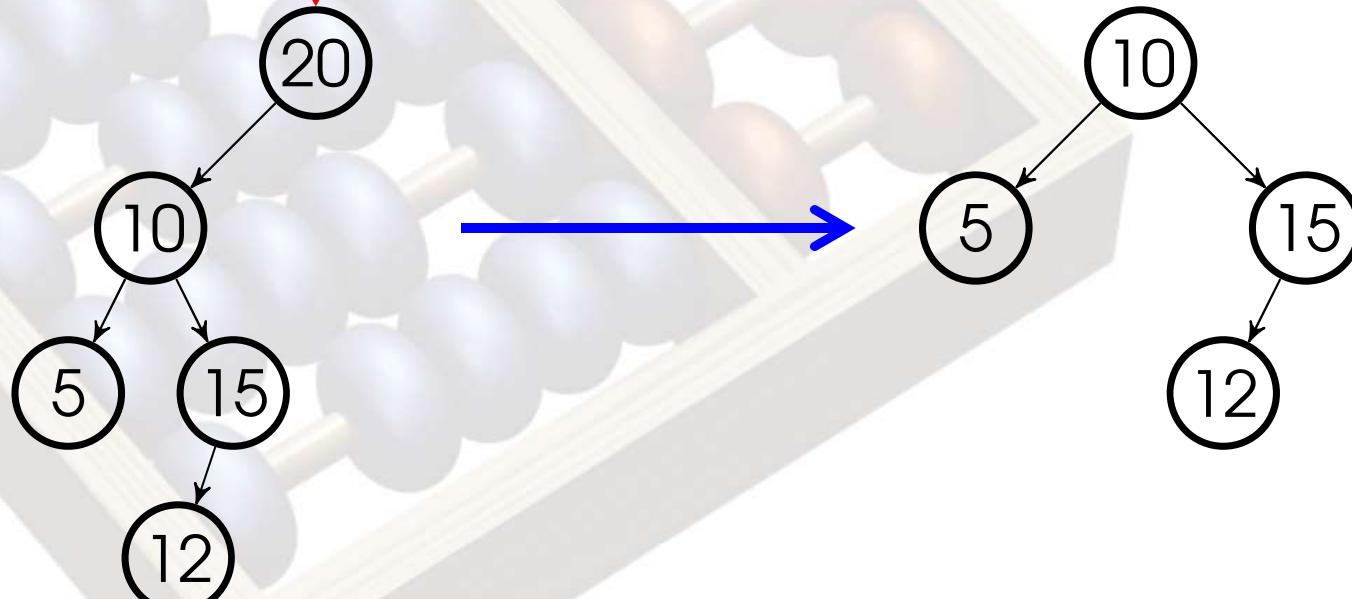
- Caso 1: a raiz não tem filhos. A árvore torna-se vazia.



UNICAMP

# Revisão – Remoção em Árv. de Busca

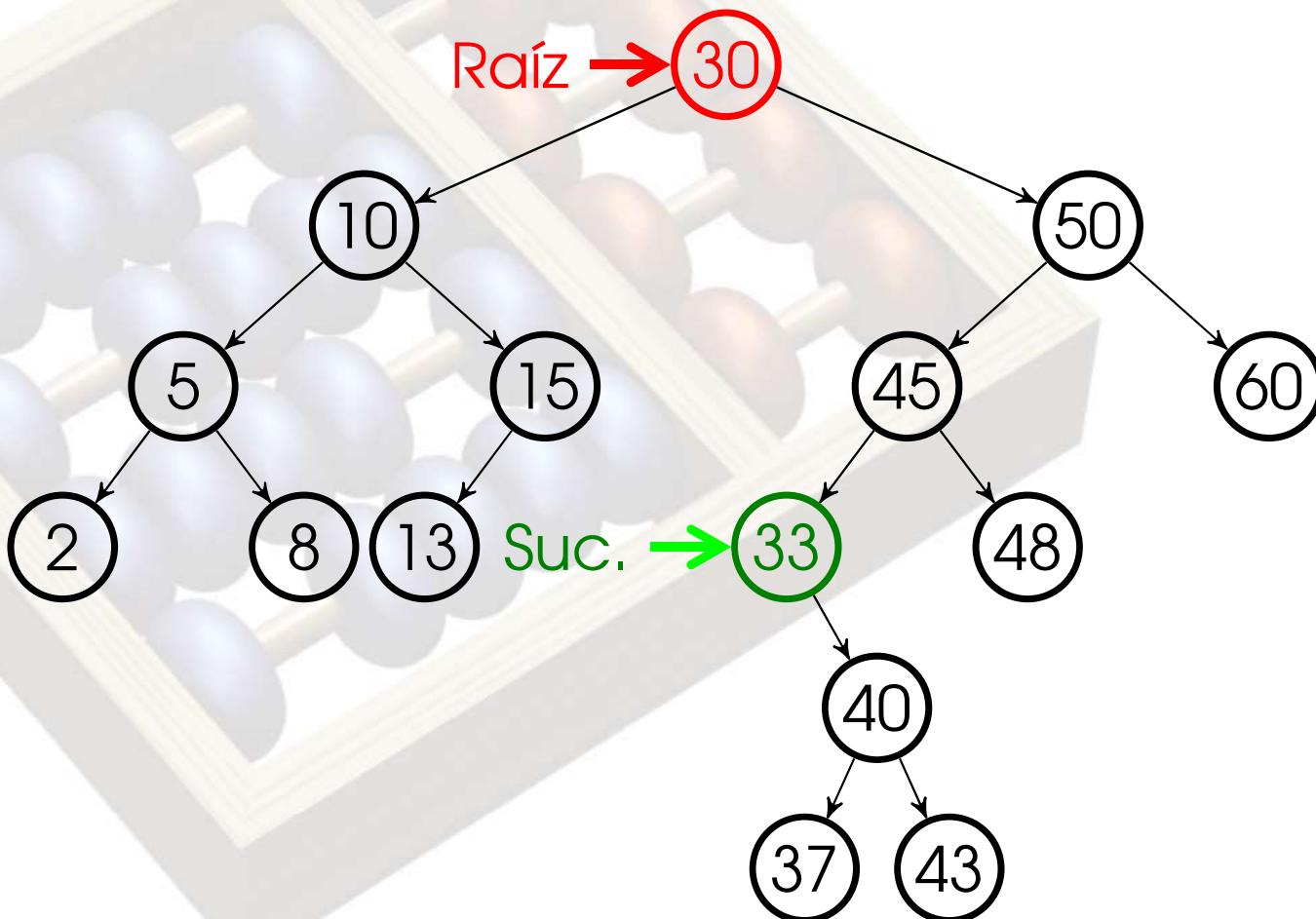
- Caso 2: a raíz tem um único filho. Seu filho torna-se a nova raiz.



UNICAMP

# Revisão – Remoção em Árv. de Busca

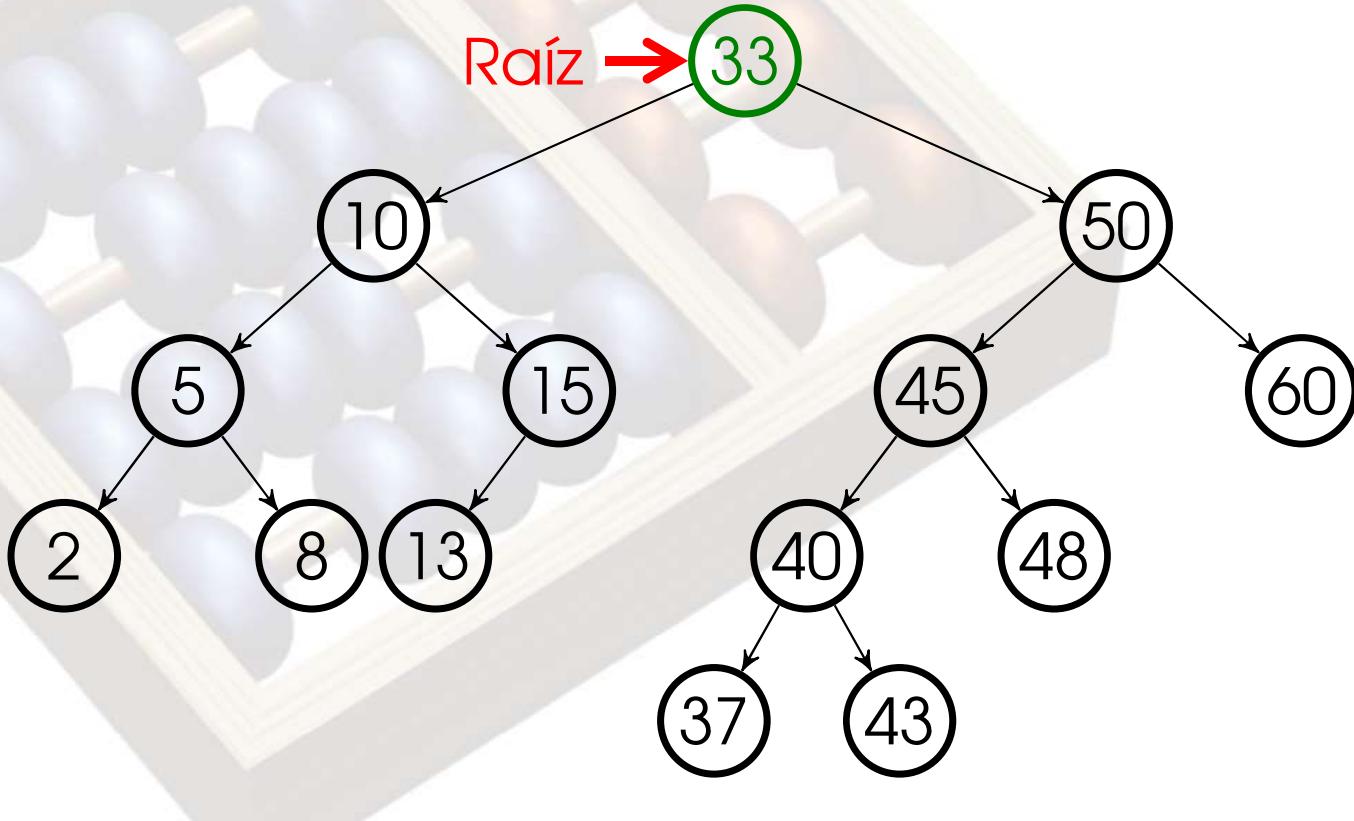
- Caso 3: a raiz tem dois filhos. Neste caso, tomamos o nó que sucede (ou precede) a raiz no percurso inordem (e-r-d) como a nova raiz.



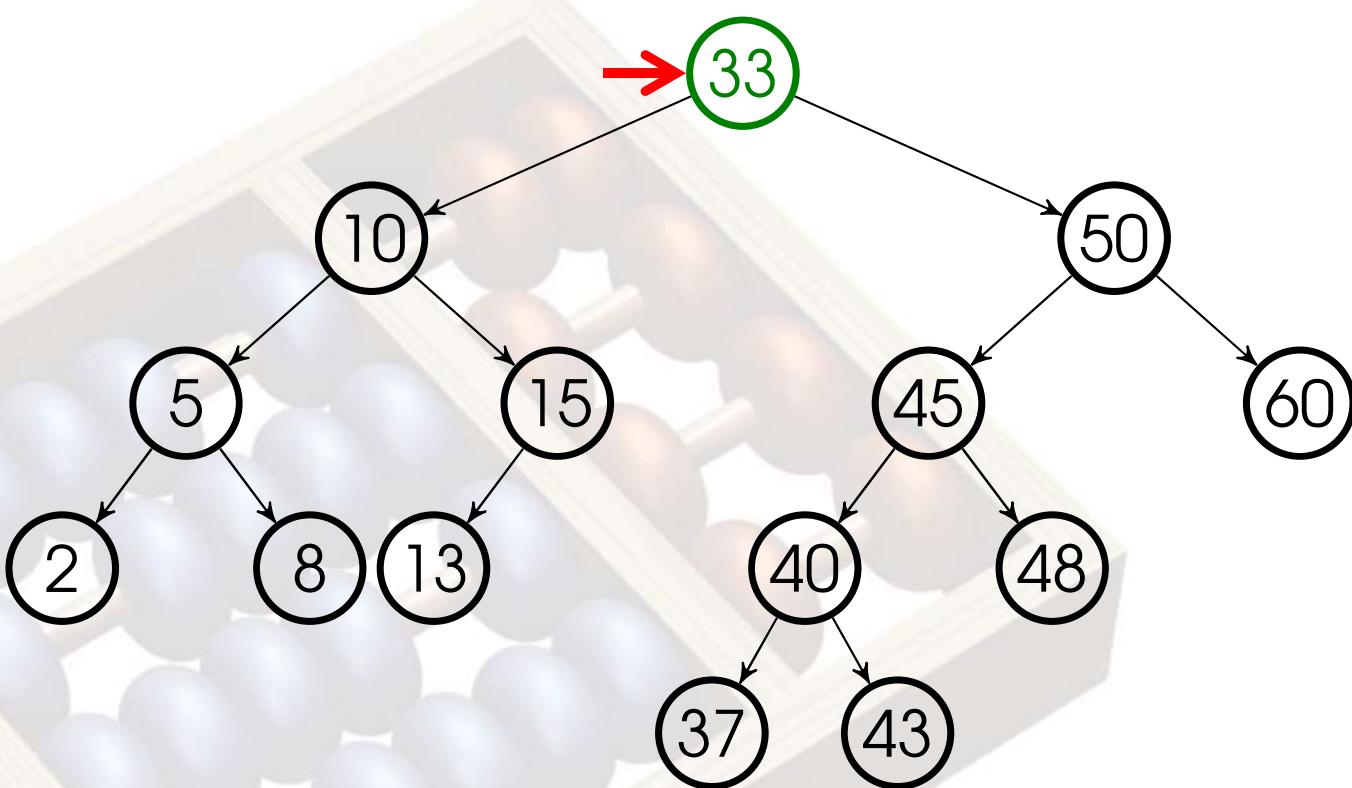
UNICAMP

# Revisão – Remoção em Árv. de Busca

- Caso 3: a raíz tem dois filhos. Neste caso, tomamos o nó que sucede (ou precede) a raíz no percurso inordem (e-r-d) como a nova raiz.



# Revisão – Remoção em Árv. de Busca



Outra maneira de interpretar o Caso 3 é a seguinte:  
copiamos a informação do nó sucessor para a raiz e  
removemos o nó copiado (ou seja, reduzimos  
ao Caso 1 ou Caso 2).



# Revisão – Remoção em Árv. de Busca

Consulte os slides da [Aula 12](#) para mais detalhes sobre a remoção de nós em árvores binárias de busca



**UNICAMP**

# Remoção em árvores Rubro-Negras

- Passos
  - Encontre o nó  $v$  a ser removido
  - Remova o nó  $v$  da árvore (use o algoritmo que acabamos de rever)
  - Conserte as propriedades da árvore rubro-negra
- Assim como as outras operações básicas em uma árvore rubro-negra com  $n$  nós, a remoção também tem complexidade de  $O(\lg n)$



UNICAMP

# Remoção ARN – Problemas

**Que problemas podemos criar** quando copiamos o valor do sucessor para a posição do nó a ser removido e removemos o sucessor (original/copiado)?

- **Remoção de um nó vermelho:** Não causa problemas no balanceamento.
  - Nenhuma altura preta mudou
  - Nenhum nó vermelho se tornou adjacente
  - Como o nó é vermelho, ele não era raiz e portanto a raiz permanece preta
- **Remoção de um nó preto:** Pode causar problema “**duplo preto**”



**UNICAMP**

# Remoção ARN – Consertando a árvore

- Se o nó removido for **preto**
  - 1 Se **suc** era raiz e um filho **vermelho** de **suc** se torna raiz quebramos a **Regra 2**
  - 2 Se tanto **x** quanto **suc->pai** (que agora também é **x->pai**) eram **vermelhos** então violamos a **Regra 4**
  - 3 A remoção de **suc** faz com que qualquer caminho que continha **suc** anteriormente ter um nó preto a menos. Desse modo quebramos a **Regra 5**

## Colinha de regras

- Regra 1: Um nó é **vermelho** ou é **preto**
- Regra 2: A raiz é **preta**
- Regra 3: Toda **folha (NULL)** é **preta**
- Regra 4: Se um nó é **vermelho** então ambos os seus filhos são **pretos**
- Regra 5: Para cada nó p, **todos** os caminhos desde p até as folhas contêm o mesmo número de nós **pretos**



**UNICAMP**

# Remoção ARN – Consertando a árvore

Assim como na inserção onde tratamos diferentes casos de violações, vamos tratar de **4 casos** diferentes para consertar a árvore após uma remoção

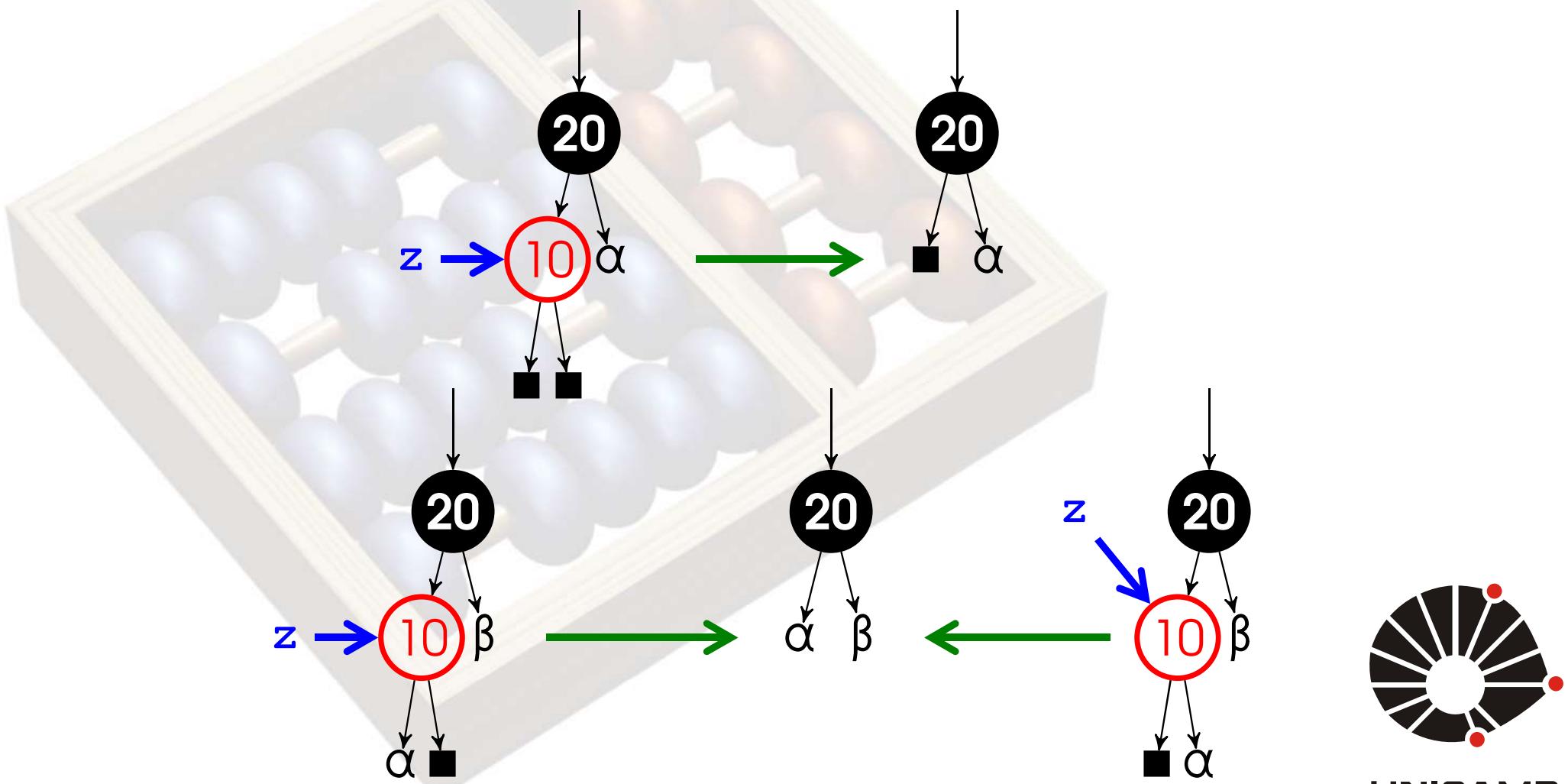
| Caso   | Nó a ser removido<br>z | Sucessor<br>suc |
|--------|------------------------|-----------------|
| Caso 1 | Vermelho               | Vermelho        |
| Caso 2 | Preto                  | Vermelho        |
| Caso 3 | Preto                  | Preto           |
| Caso 4 | Vermelho               | Preto           |



# Remoção ARN – Caso 1

**Caso 1** O nó a ser removido  $z$  é **vermelho** e possui 0 ou 1 filho

- 1 Remova o nó diretamente

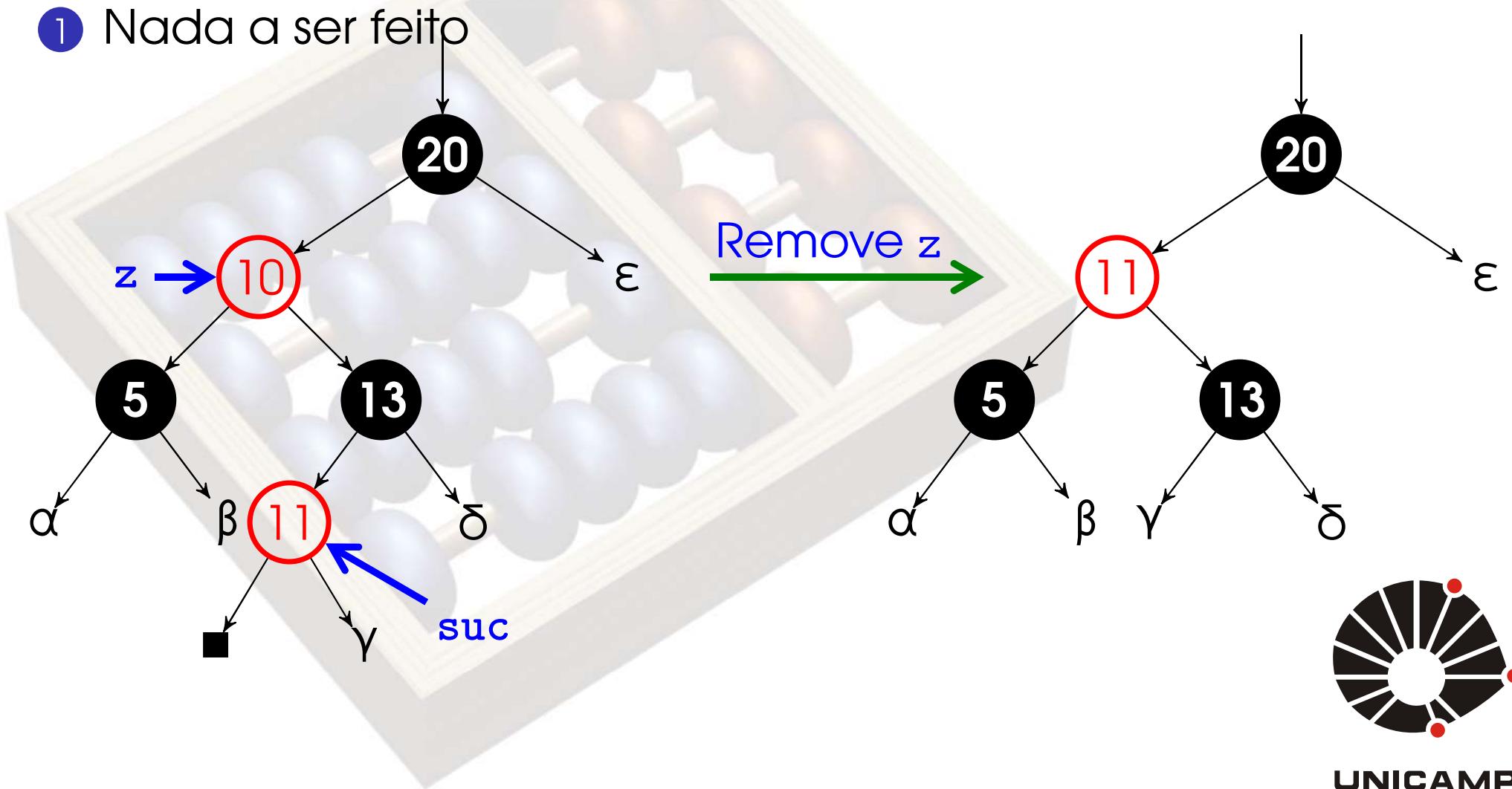


UNICAMP

# Remoção ARN – Caso 1

**Caso 1** O nó a ser removido  $z$  é **vermelho** e  $suc$ , sucessor de  $z$  também é **vermelho**

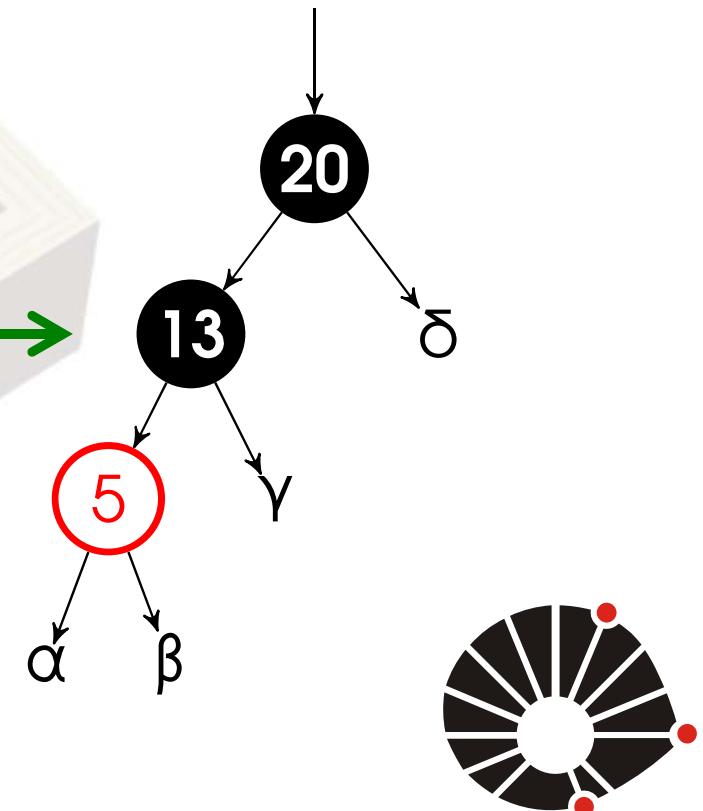
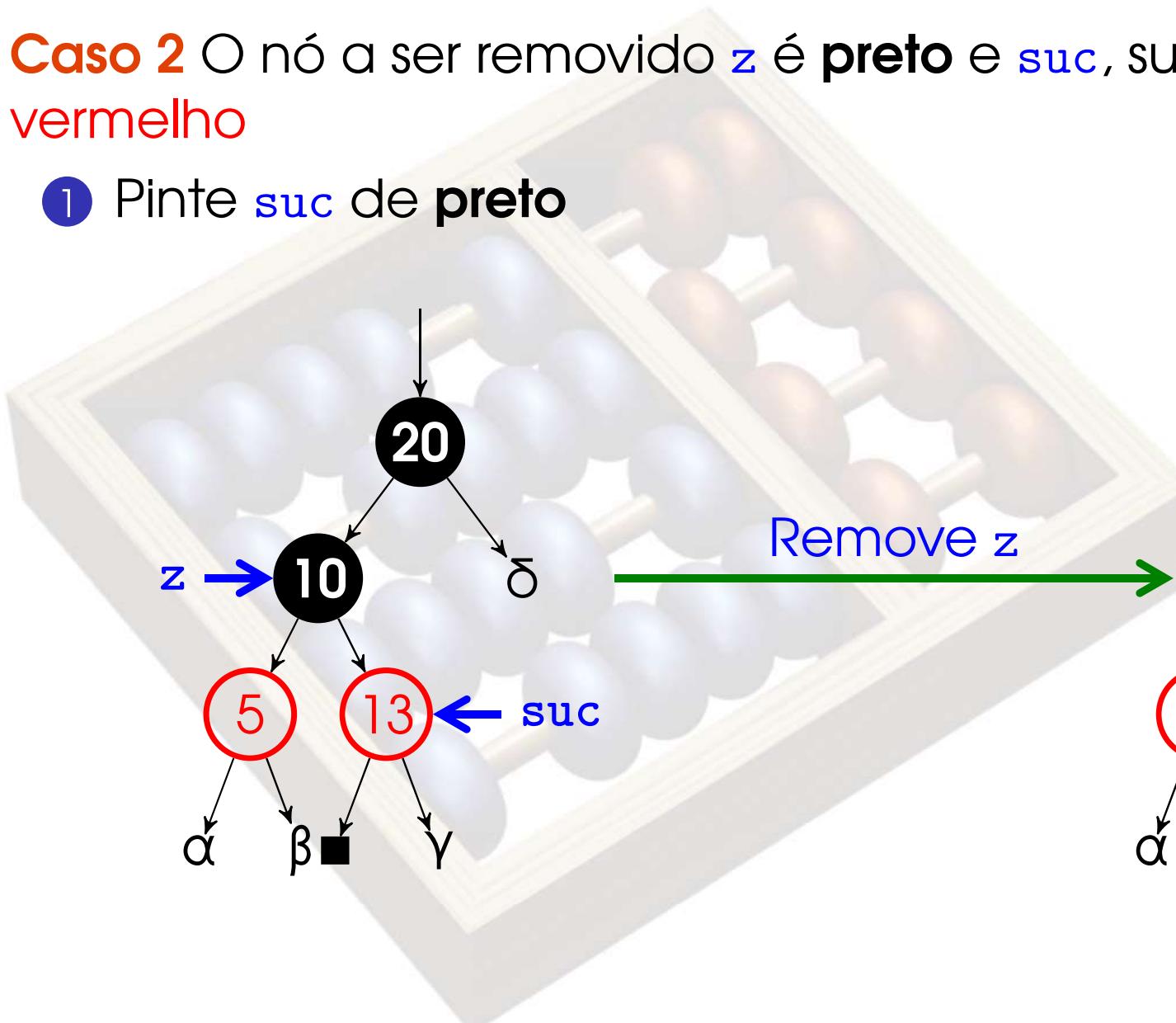
- 1 Nada a ser feito



# Remoção ARN – Caso 2

**Caso 2** O nó a ser removido  $z$  é **preto** e **suc**, sucessor de  $z$  é **vermelho**

- 1 Pinte **suc** de **preto**



# Remoção ARN – Caso 3

- O problema do duplo preto ocorre quando retiramos um nó **preto**
- O duplo preto nada mais é do que uma forma de **compensar** a falta do nó removido na altura de preto da árvore
- Existem 4 casos a tratar caso o nó **suc** a ser removido for **preto**



**UNICAMP**

# Remoção ARN – Caso 3

Usaremos a seguinte nomenclatura para os nós envolvidos no processo de remoção:

- $z$  – O nó a ser removido
- $suc$  – Onde  $suc = z$  se  $z$  possui 0 ou 1 filho. Caso contrário  $suc = sucessor(z)$
- $x$  – O filho de  $suc$  antes de qualquer modificação, ou  $NULL$  caso  $suc$  não tenha filhos
- $w$  – o tio de  $x$  (irmão de  $suc$ ) antes da remoção de  $z$



UNICAMP

# Remoção ARN – Caso 3

- Além disso, durante o conserto da árvore, quando considerarmos o nó **x** vamos conta-lo como **preto duas vezes**
- A ideia é que o **x** possa receber sempre a contagem de preto do pai para manter a **Regra 5**.
- Nos casos em que **x** é vermelho (ainda assim o contamos como preto no cálculo da altura de preto), basta pintar **x** de **preto** para finalizar o conserto da árvore
- **x** aponta para um duplo preto, o que nos diz que o nó **w** existe (Pq?)



UNICAMP

# Remoção ARN – Caso 3

Resumo das variações do Caso 3: **z preto** e **suc preto**

- **Caso 3.1** – O irmão **w** de **x** é **vermelho**
- **Caso 3.2** – O irmão **w** de **x** é **preto**, e ambos os filhos de **w** são **pretos**
- **Caso 3.3** – O irmão **w** de **x** é **preto**, o filho esquerdo de **w** é **vermelho** e o filho da direita de **w** é **preto**
- **Caso 3.4** – O irmão **w** de **x** é **preto**, e o filho direito de **w** é **vermelho**



**UNICAMP**

# Remoção ARN – Caso 3.1

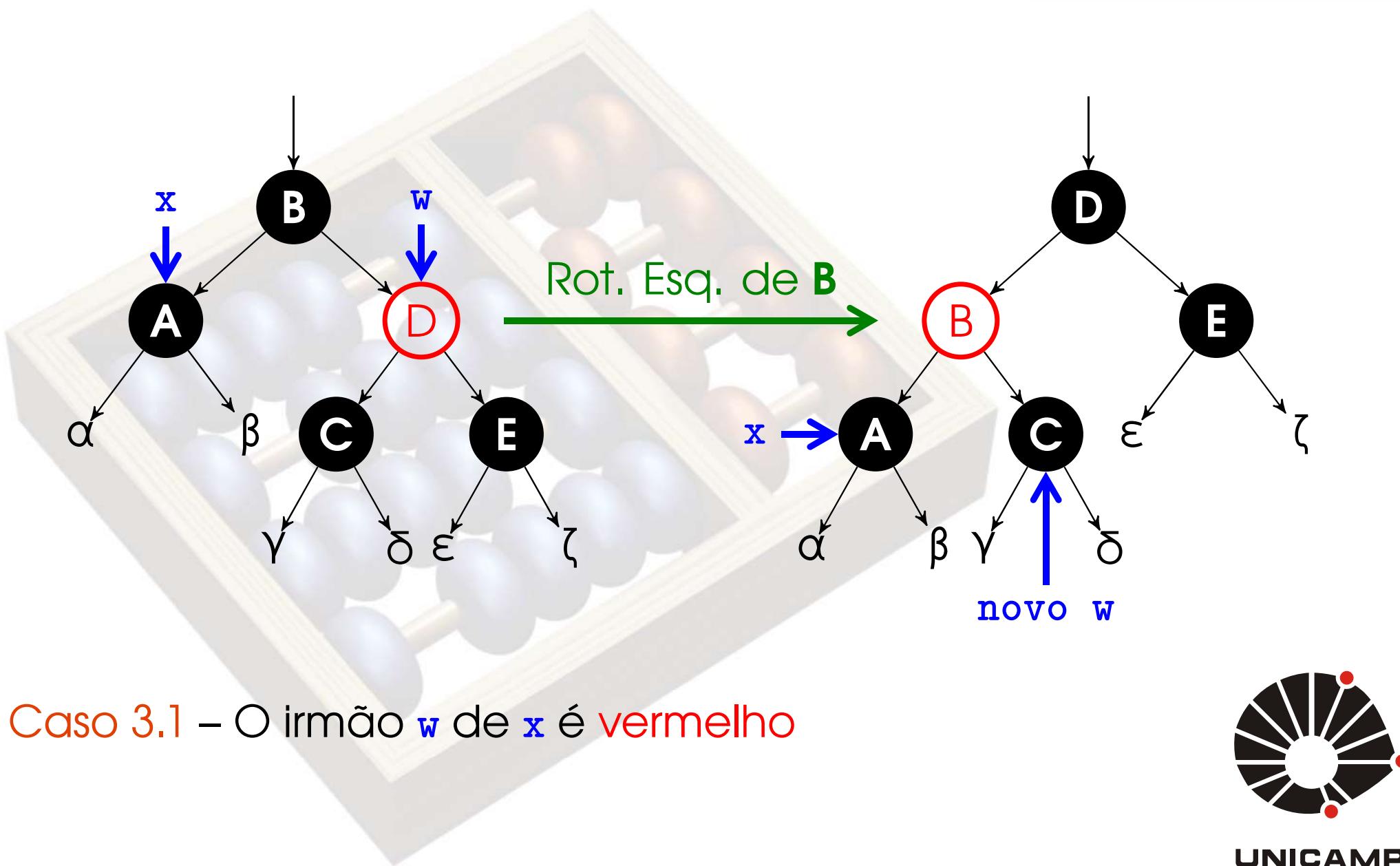
Caso 3.1 – O irmão  $w$  de  $x$  é vermelho

- Claramente o pai deles é vermelho
- Como  $w$  é vermelho, ambos os filhos são pretos. Logo:
  - Trocamos as cores de  $w$  e  $x.pai$
  - Efetuamos uma rotação à esquerda tendo como pivô  $x.pai$
- Essas alterações não violam nenhuma regra da árvore rubro-negra
  - Mas transformam o Caso 3.1 no Caso 3.2, Caso 3.3 ou Caso 3.4



UNICAMP

# Remoção ARN – Caso 3.1



# Remoção ARN – Caso 3.2

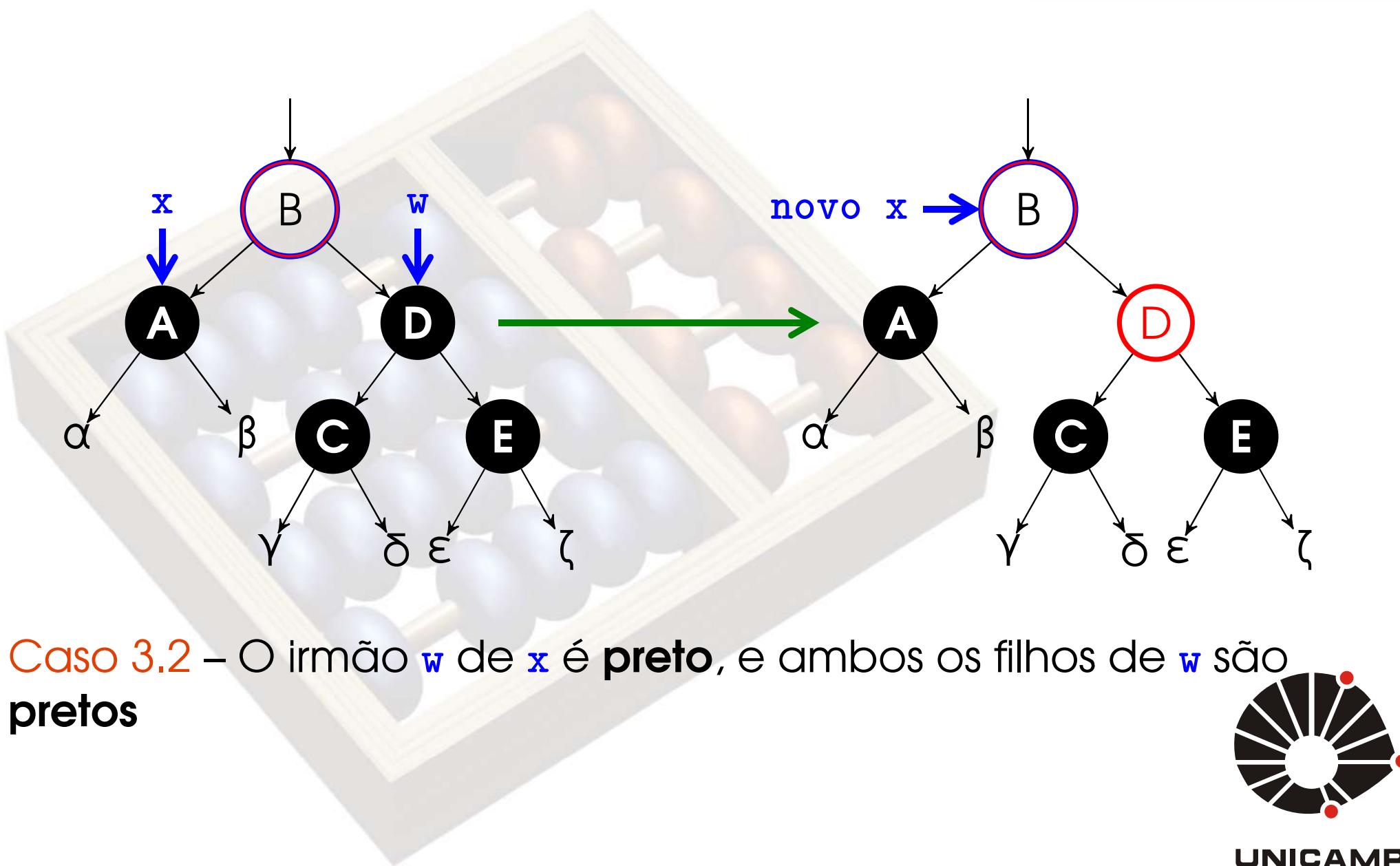
Caso 3.2 – O irmão **w** de **x** é **preto**, e ambos os filhos de **w** são **pretos**

- Retiramos um preto de **x** e um de **w** deixando **x** com apenas um preto e deixando **w** **vermelho**
- Para compensar a remoção de um preto de **x** e de **w**, adicionamos um preto extra no **x.pai** que era originalmente **preto** ou **vermelho**
- Jogamos a bomba para cima, tratando o **x.pai** como sendo o **novo x**



**UNICAMP**

# Remoção ARN – Caso 3.2



UNICAMP

# Remoção ARN – Caso 3.3

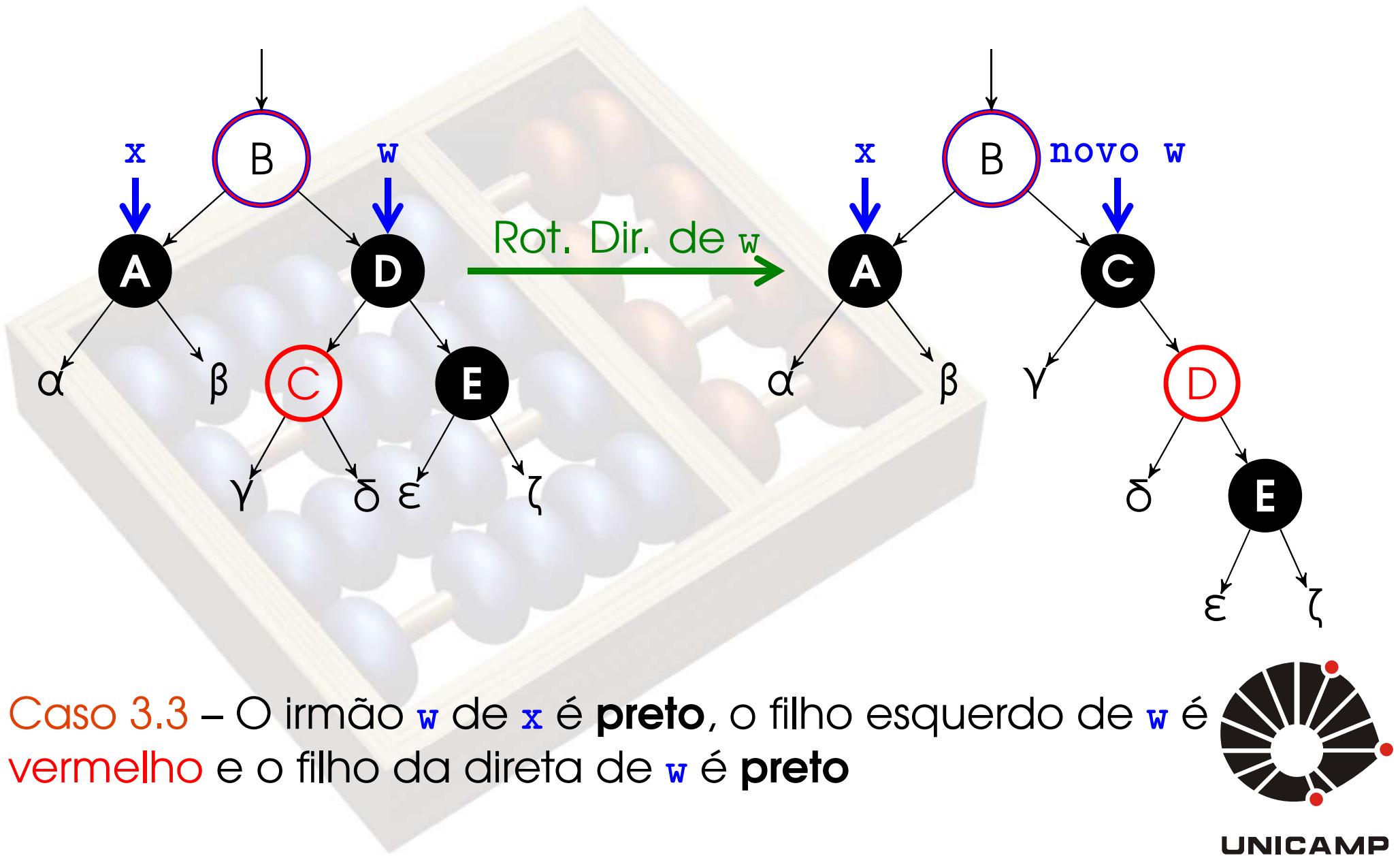
Caso 3.3 – O irmão  $w$  de  $x$  é **preto**, o filho esquerdo de  $w$  é **vermelho** e o filho da direita de  $w$  é **preto**

- Trocamos as cores de  $w$  e de seu filho esquerdo
- Rotaciona árvore à direita usando como pivô  $w$
- Essas operações não introduzem violações
- Neste ponto o novo irmão  $w$  de  $x$  é um nó **preto** com um filho da direita **vermelho**. Estamos, portanto, no **Caso 3.4**.



**UNICAMP**

# Remoção ARN – Caso 3.3



UNICAMP

# Remoção ARN – Caso 3.4

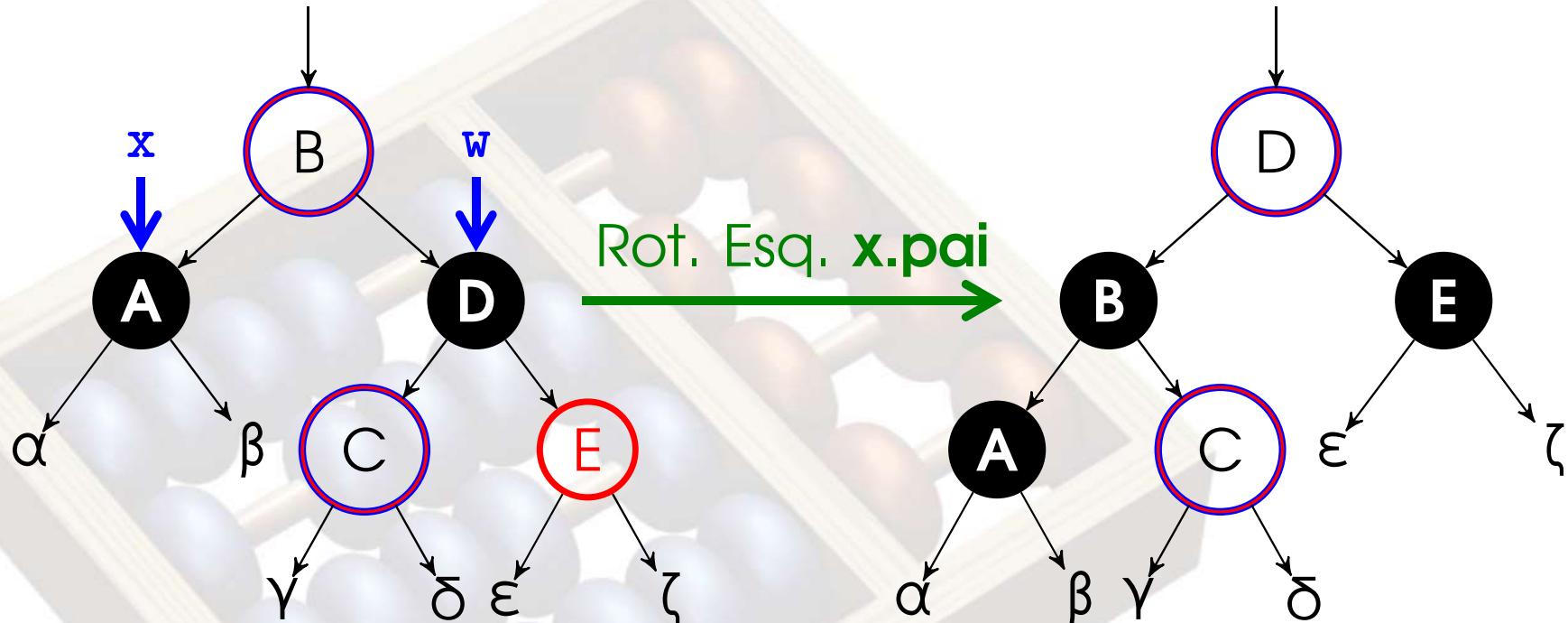
Caso 3.4 – O irmão  $w$  de  $x$  é preto, e o filho direito de  $w$  é vermelho

- Rotaciona árvore à esquerda usando como pivô  $x.pai$
- $w$  é pintado com a cor de  $x.pai$
- Pinta o  $x.pai$  de preto
- Pinta o filho direito de  $w$  de preto



UNICAMP

# Remoção ARN – Caso 3.4



Caso 3.4 – O irmão  $w$  de  $x$  é **preto**, e o filho direito de  $w$  é **vermelho**



# Remoção ARN – Caso 4

**Caso 4** O nó a ser removido  $z$  é vermelho e  $x$ , irmão de  $suc$ , é preto

- 1 Pinte  $x$  de vermelho
- 2 Resolva usando o Caso 3



UNICAMP

# AVL vs. ARN

- **AVL**

- Primeira árvore binária de busca com balanceamento proposta por Adelson-Velskii e Landis em 1962
- Altura entre  $\log(n+1)$  e  $1.44404 * \log(n+2) - 0.328$ , ou seja,  $O(\lg n)$

- **ARN**

- Proposta por Guibas e Sedgewick em 1978.
- Altura até  $2 * \log(n + 1)$ , logo,  $O(\lg n)$



**UNICAMP**

# AVL vs. ARN

- Na teoria ambas têm a mesma complexidade de inserção, remoção e busca ( $O(\lg n)$ )
- Na prática a árvore AVL é mais rápida para buscas e mais lenta para inserção e remoção
- As árvores **AVL** são mais rigidamente balanceadas do que as árvores rubro-negras, o que permite uma operação de busca mais rápida mas também compromete o desempenho das operações de modificação
- Por **exemplo**, uma operação de remoção pode exigir  $O(\lg n)$  rotações na AVL no pior caso, enquanto uma árvore rubro-negra se arrumaria com apenas 3



**UNICAMP**

# AVL vs. ARN

Quando usar AVL ou ARN?

- Se a aplicação da árvore realiza de forma mais intensa operações de busca, é mais apropriado o uso de uma árvore AVL
- Se operações de modificação são mais comuns, o melhor é usar uma árvore rubro-negra
- Via de regra árvores rubro negras são mais utilizadas em bibliotecas



**UNICAMP**

Bacharelado em Ciência da Computação  
**GBC034 Algoritmos e Estruturas de Dados 2**

# Hashing

Profa. Maria Camila Nardini Barioni

[camila.barioni@ufu.br](mailto:camila.barioni@ufu.br)

Bloco B - sala 1B137

2º trimestre de 2023

# Roteiro

- ◆ Relembrando
- ◆ Hashing - Definições
- ◆ Tratamento de Colisão
- ◆ Comparações: Estruturas e Métodos de Pesquisa
- ◆ Exercícios

# Relembrando...

## ◆ Pesquisa sequencial (ou linear)

- Aplica-se tanto a dados ordenados quanto a dados não ordenados
- É um método muito custoso, porque no pior caso todos os elementos devem ser pesquisados

## ◆ Pesquisa binária

- Aplica-se somente a dados ordenados
- Arrays, Árvores
- **Problema:** a **ordenação** frequentemente leva muito mais tempo do que a **busca**

# Relembrando...

- ◆ Foram estudadas diferentes estruturas e algoritmos para pesquisa
  - Baseados na **comparação da chave** de pesquisa com as chaves armazenadas
  - Para obter algoritmos eficientes
    - ◆ Considerou-se o armazenamento de elementos ordenados ( **$O(N \log N)$**  no melhor caso)
    - ◆ Algoritmos eficientes → esforço computacional  **$O(\log N)$**

# Aula de hoje...

- ◆ Serão apresentadas as estruturas de dados conhecidas como **Tabelas de Dispersão (Hash Tables)** → métodos de pesquisa **Transformação de Chave (Hashing)**
  - Os elementos armazenados em uma tabela são diretamente endereçados a partir de uma transformação aritmética sobre a chave de pesquisa
  - Caso bem projetadas, podem ser usadas para buscar um elemento em ordem constante: **O(1)**

# Hashing

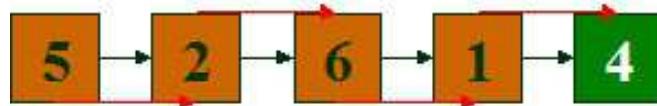
◆ De acordo com *Webster's New World Dictionary*

- Hash → (i) fazer picadinho de carne e vegetais para cozinhar; (ii) fazer bagunça
  - ◆ Nome apropriado para o método

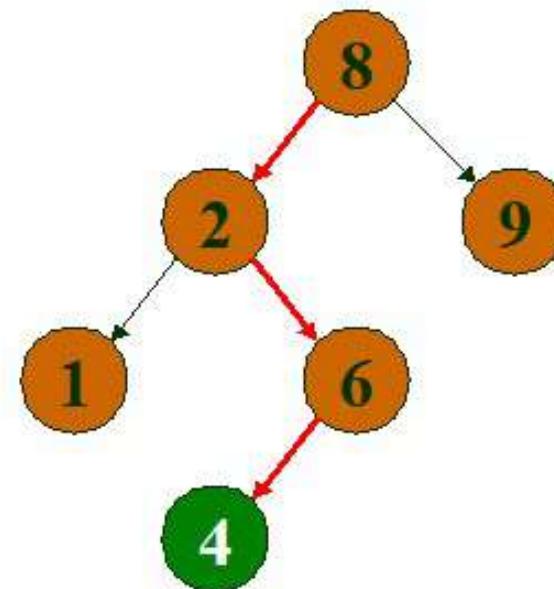
# Por que usar Hashing?

- ◆ Estruturas de busca seqüencial levam tempo até encontrar o elemento desejado

Ex: Arrays e listas

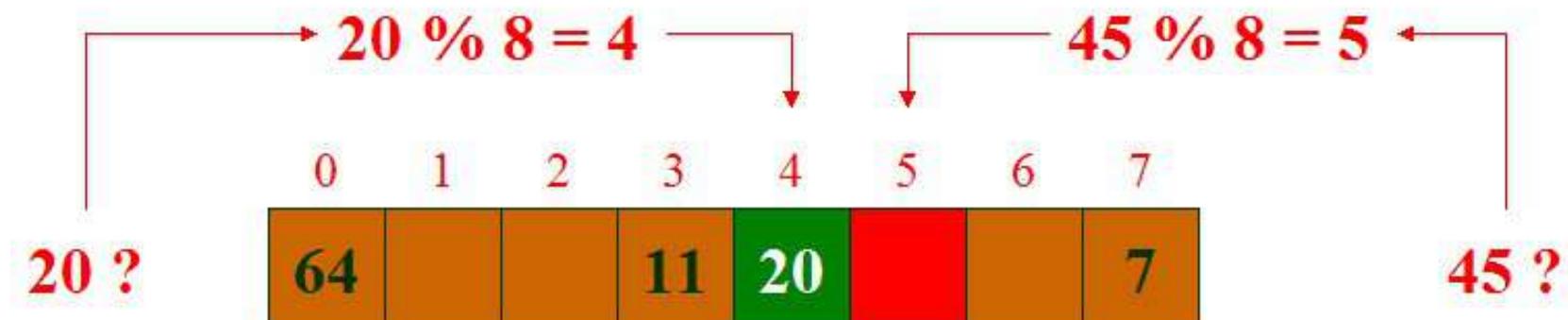


Ex: Árvores



# Por que usar Hashing?

- ◆ Em algumas aplicações, é necessário obter o valor com poucas comparações, logo, é preciso saber a posição em que o elemento se encontra, **sem precisar varrer todas as chaves**
- ◆ A estrutura com tal propriedade é chamada de tabela hash



# Apresentando a ideia geral

- ◆ Problema: Armazenar os dados dos alunos de uma disciplina
  - Identificação → número de matrícula (`mat`)
    - Chave de busca
    - Oito dígitos sendo o último usado para controle
  - Acesso direto → usar `mat` como índice de um vetor `vet`
    - Acesso aos dados do aluno → `vet[mat]`
    - Problema: o preço pago para ter esse acesso direto é muito grande

# Apresentando a ideia geral

## ◆ Estrutura para representar as informações dos alunos

```
struct aluno{
 int mat;
 char nome[81];
 char email[41];
 char turma; };
typedef struct aluno Aluno;
```

- Número de matrícula → número inteiro variando de 0 a 9999999

```
#define MAX 10000000
Aluno vet[MAX];
```

# Apresentando a ideia geral

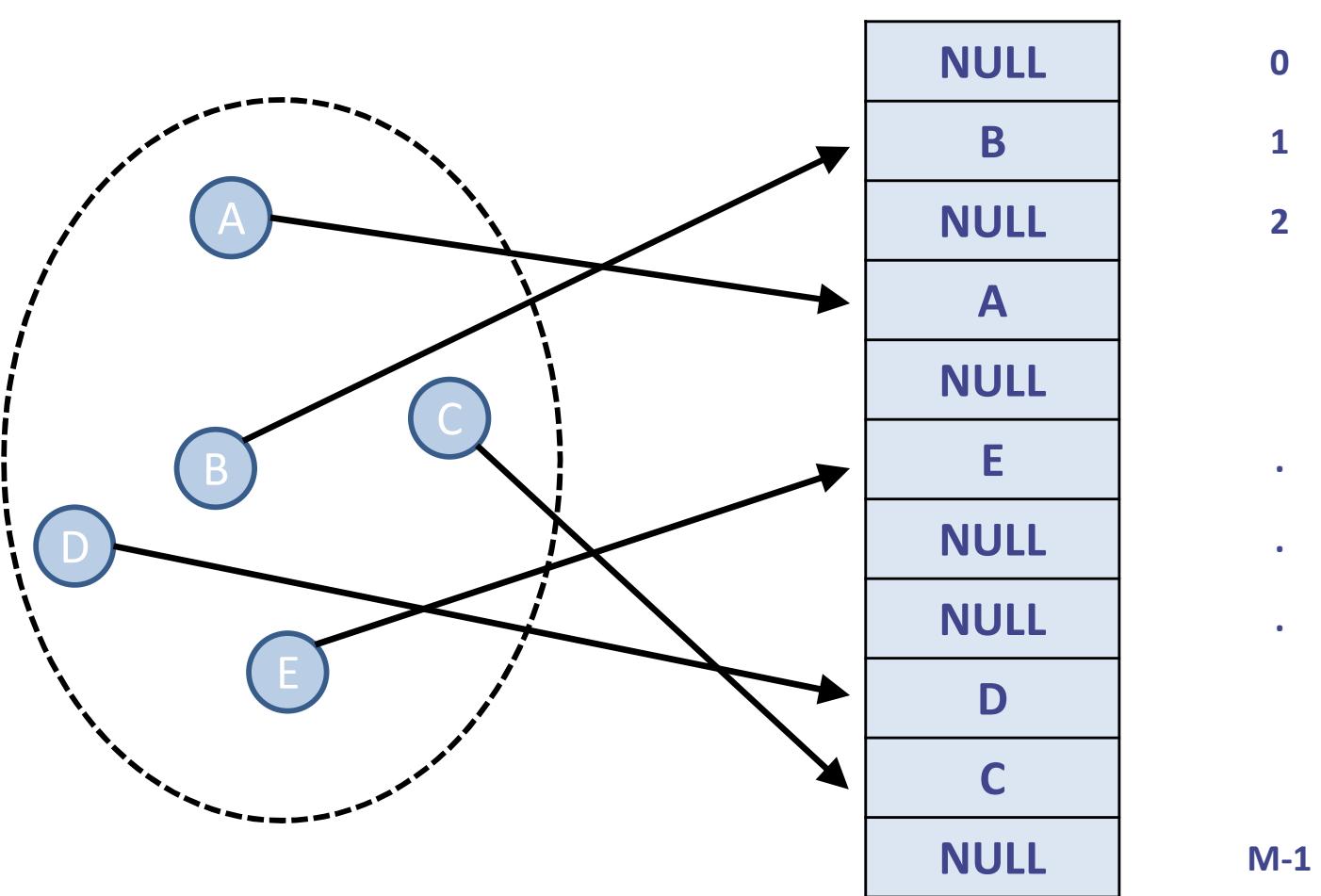
- ◆ Dessa forma, aluno com matrícula `mat` é acessado por: `vet[mat].nome`
  - Acesso rápido porém com uso de memória proibitivo
  - Cada aluno ocupa 127 bytes → 1.270.000.000 bytes (acima de 1 GB)
- ◆ Para amenizar o problema
  - vetor de ponteiros
  - Posições que não correspondem a alunos cadastrados → NULL
  - Aluno com matrícula `mat`: `vet[mat] -> nome`
  - Gasto menor de memória, mas ainda proibitivo
- ◆ Solução → **Tabela Hash**

# Tabela Hash

- ◆ Também conhecidas como **tabelas de dispersão, de indexação ou de espalhamento**
  - É uma generalização da ideia de array
- ◆ Ideia central
  - Utilizar uma função, chamada de **função de hashing**, para espalhar os elementos que queremos armazenar na tabela
  - Esse espalhamento faz com que os elementos fiquem dispersos de forma não ordenada dentro do array que define a tabela

# Tabela Hash

## ♦ Exemplo



# Tabela Hash

- ◆ Por que espalhar os elementos melhora a busca?
  - A tabela permite associar valores a chaves
    - ◆ **chave:** parte da informação que compõe o elemento a ser inserido ou buscado na tabela
    - ◆ **valor:** é a posição (índice) onde o elemento se encontra no array que define a tabela
  - Assim, a partir de uma chave podemos acessar de forma rápida uma determinada posição do array
    - ◆ Na média, essa operação tem custo  $O(1)$

# Tabela Hash

## ◆ Vantagens

- Alta eficiência na operação de busca
  - ◆ Caso médio é **O(1)** enquanto o da busca linear é **O(N)** e a da busca binária é **O(log<sub>2</sub> N)**
- Tempo de busca é praticamente independente do número de chaves armazenadas na tabela
- Implementação simples

# Tabela Hash

## ◆ Desvantagens

- Alto custo para recuperar os elementos da tabela ordenados pela chave
  - ◆ Nesse caso, é preciso ordenar a tabela
- O pior caso é **O(N)**, sendo **N** o tamanho da tabela
- Alto número de **colisões**

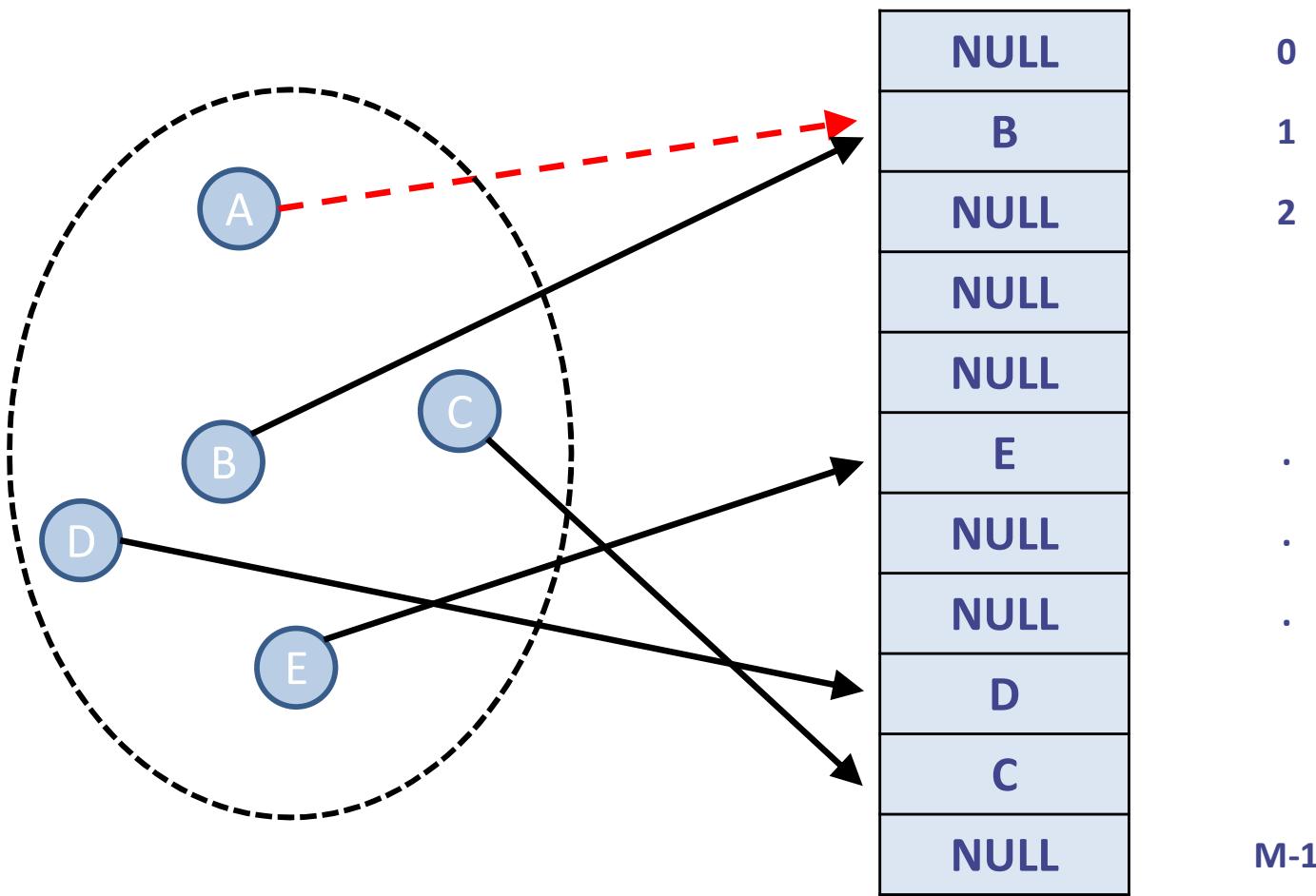
# Hashing → etapas de pesquisa

- ◆ Armazena um conjunto de elementos em um vetor de tamanho aproximadamente igual
  - Na verdade, o vetor deveria ser sempre maior que o número de elementos
- ◆ Duas etapas principais:
  - Computar o valor da **função de transformação (função hashing)**, a qual transforma a chave de pesquisa em um endereço da tabela
  - Considerando que duas ou mais chaves podem ser transformadas em um mesmo endereço de tabela → lidar com **colisões**

# Hashing → etapas de pesquisa

## ◆ Exemplo de colisão

- A tenta ocupar a posição onde B está



# Hashing → etapas de pesquisa

## ◆ Exemplos:

- Considerando chaves inteiras de 1 a n
  - ◆ Armazenar o registro **i** na posição **i** da tabela → acesso imediato
  - ◆ Tabela com capacidade **M = 97 chaves** para chaves com 4 dígitos → **N = 10.000 chaves possíveis** (podem ocorrer colisões)
- Paradoxo do aniversário (Feller, 1968)
  - ◆ Em um grupo de 23 ou mais pessoas existe uma chance maior do que 50% de que 2 pessoas façam aniversário no mesmo dia
  - ◆ Se for usada uma função hashing uniforme que enderece **23 chaves** randômicas em uma **tabela de tamanho 365** → 50% de chance de colisão

# Definições: Funções de Hashing

## ◆ Função de Hashing (Função de Transformação)

- Calcula a posição a partir de uma chave escolhida a partir dos dados manipulados
- Mapeia chaves em inteiros dentro do intervalo  $[0..M-1]$ , sendo  $M$  o tamanho da tabela



# Definições: Funções de Hashing

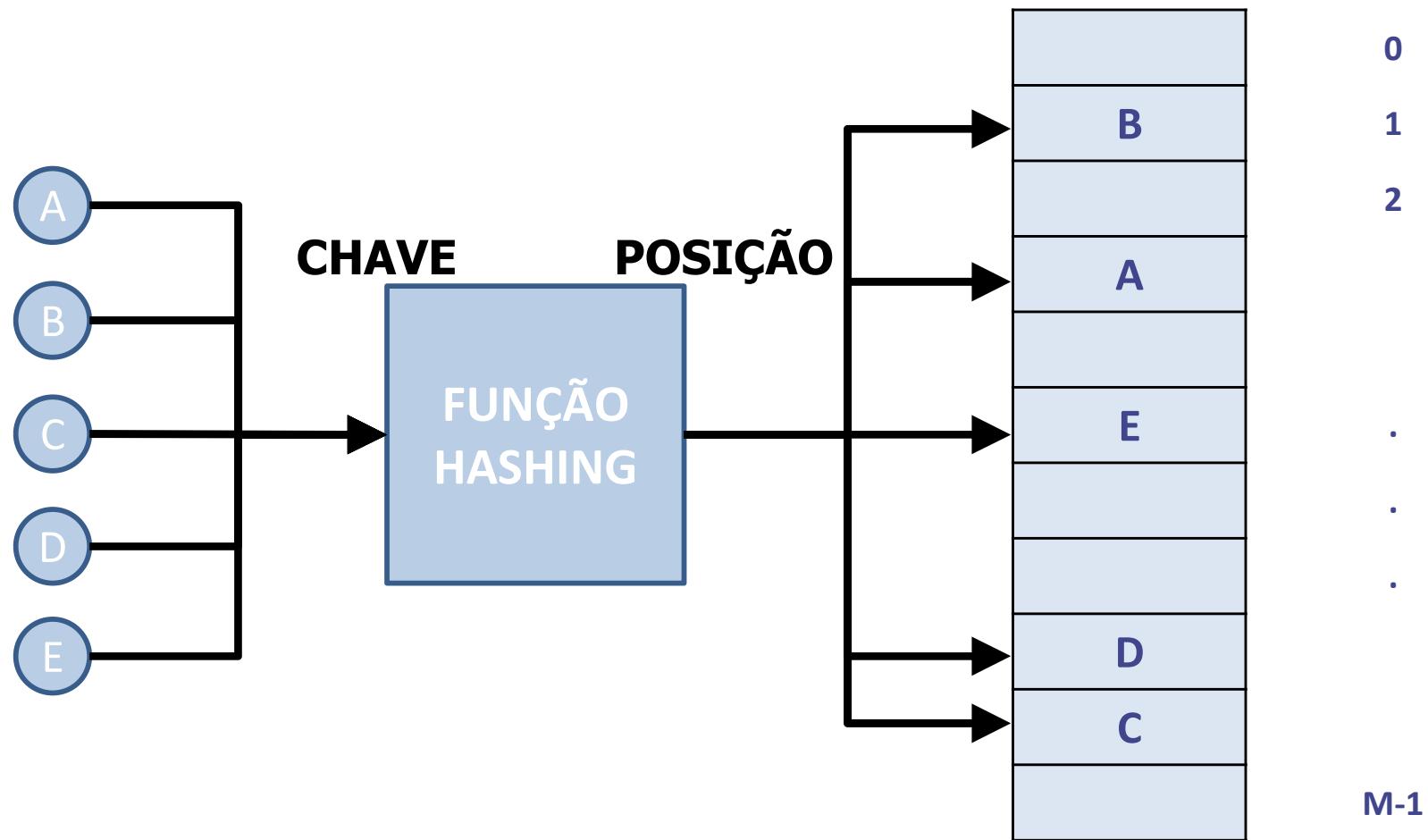
## ◆ Função de Hashing (Função de Transformação)

- É extremamente importante para o bom desempenho da tabela
- Ela é responsável por distribuir as informações de forma equilibrada pela tabela hash



# Definições: Funções de Hashing

## ◆ Exemplo de funcionamento



# Definições: Funções de Hashing

## ◆ Função ideal

- Simples de ser computada
- Garantir que valores diferentes produzam posições diferentes
- Gerar uma distribuição equilibrada dos dados na tabela
  - ◆ Para cada chave de entrada, qualquer uma das saídas possíveis é igualmente provável de ocorrer

# Definições: Funções de Hashing

- ◆ Sua implementação depende do conhecimento prévio da natureza e domínio da chave a ser utilizada
- ◆ Considerando que as transformações sobre as chaves são aritméticas → transformar chaves não numéricas em números

# Definições: Mapeamento

- ◆ Associação de cada objeto de um tipo a uma chave, permitindo a indexação
- ◆ Ex: String → Inteiro

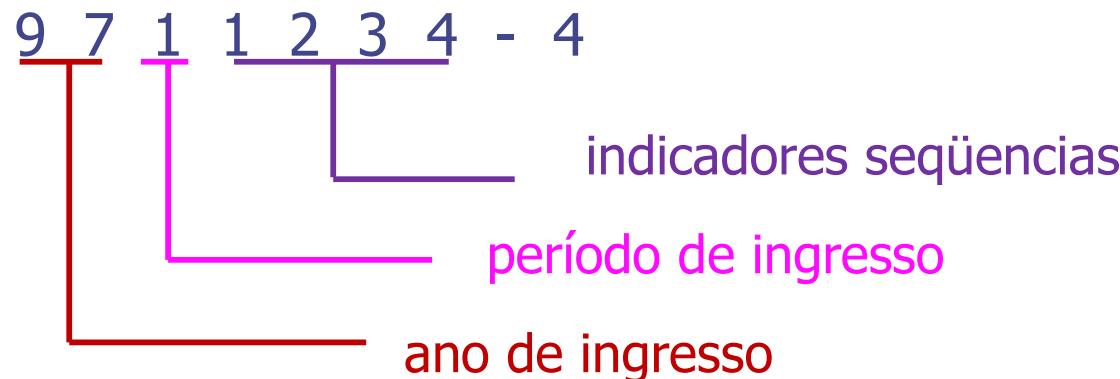


- ◆ Útil para construir estruturas de armazenamento com custo reduzido em tempo (acesso direto)

# Definições: Mapeamento

## ◆ Ex: dados dos alunos em uma disciplina

- Ideia central → identificar na chave de busca as partes significativas



- Os dígitos mais significativos são os dois últimos → em um universo de uma turma de alunos o dígito que representa a unidade varia mais do que o que representa o milhar
  - ◆ Podemos usar um número de matrícula parcial
  - ◆ Ex: 50 alunos na turma e chave de dois dígitos → `Aluno* vet[100]`
  - ◆ Problema: dois ou mais alunos podem apresentar a mesma chave parcial

# Definição: Função de transformação

- ◆ Várias funções de transformação têm sido estudadas
  - Um dos métodos que funciona muito bem é o resto da divisão por M
$$h(k) = k \text{ mod } M, \text{ sendo } k \text{ a chave e } M \text{ o tamanho da tabela}$$
  - É preciso ter cuidado na escolha de M

# Definição: escolha do tamanho da tabela

- ◆ Existem vários estudos (antigos) sobre a escolha do melhor tamanho para a tabela hash
- ◆ O objetivo é encontrar um tamanho que
  - Minimize a quantidade de posições não utilizadas
    - ◆ Isso porque se for utilizado um vetor muito grande, certamente vai ser difícil ocorrer colisão
    - ◆ No entanto, que tamanho deveria ter esse vetor?
  - Minimize a quantidade de colisões
    - ◆ O tamanho da tabela, a função de hash e a característica das chaves influenciam
- ◆ Estudos indicam que o tamanho da tabela  $M$  deve ser um número primo
- ◆ Estudos empíricos, mostram que não devemos permitir que a tabela tenha uma taxa de ocupação superior a 75%

# Definições: Colisões

- ◆ Devido ao fato de existirem mais chaves que posições, é comum que várias chaves sejam mapeadas na mesma posição. Quando isto ocorre, dizemos que houve uma colisão.
- ◆ Ex:  
 $45 \% 8 = 5$        $1256 \% 15 = 11$   
 $21 \% 8 = 5$        $356 \% 15 = 11$   
 $93 \% 8 = 5$        $506 \% 15 = 11$
- ◆ O que fazer quando mais de um elemento for inserido na mesma posição de uma tabela hash?

# Hashing Universal

- ◆ Função de hashing está sujeita ao problema de gerar posições iguais para chaves diferentes
  - Por se tratar de uma função determinística, ela pode ser manipulada de forma indesejada
  - Conhecendo a função de hashing, pode-se escolher as chaves de entrada de modo que todas colidam, diminuindo o desempenho da tabela na busca para **O(N)**

# Hashing Universal

- ◆ Hashing universal é uma estratégia que busca minimizar esse problema de colisões
  - Basicamente, devemos escolher aleatoriamente (em tempo de execução) a **função de hashing** que será utilizada
  - Para tanto, construímos um conjunto (ou família) de **funções de hashing**

# Hashing Universal

- ◆ Existem várias maneiras diferentes de construir uma família de funções de hashing
  - Uma família de funções pode ser facilmente obtida da seguinte forma:
    - ◆ Escolha um número primo **p**. Ele deve ser maior do que qualquer chave **k** a ser inserida
    - ◆ **p** também deve ser maior do que o tamanho da tabela, **M**
    - ◆ Escolha, aleatoriamente, dois números inteiros, **a** e **b**, de tal modo que  $0 < a \leq p$  e  $0 \leq b \leq p$

# Hashing Universal

◆ Dados os valores **p**, **a**, e **b**, definimos a função de hashing universal como sendo

- $$h(k)_{a,b} = ((ak + b) \% p) \% M$$

- Esse tipo de função de hashing universal permite que o tamanho da tabela, **M**, não seja necessariamente primo
- Além disso, como existem **p-1** valores diferentes para o valor de **a** e **p** valores possíveis para **b**, é possível gerar **p(p-1)** funções de hashing diferentes

# Hashing imperfeito e perfeito

- ◊ A depender do tamanho da tabela, **M**, e dos valores inseridos, uma função de hashing pode ser definida como
  - Hashing imperfeito
  - Hashing perfeito

# Hashing imperfeito e perfeito

## ◆ Hashing imperfeito

- Para duas chaves diferentes a saída da função de hashing é a mesma posição na tabela
- Ou seja, podem ocorrer colisões das chaves
  - ◆ A colisão de chaves não é algo exatamente ruim, é apenas algo indesejável pois diminui o desempenho do sistema
  - ◆ De modo geral, muitas tabelas hash fazem uso de alguma outra estrutura de dados para lidar com o problema da colisão, como veremos adiante

# Hashing imperfeito e perfeito

## ◆ Hashing perfeito

- Nunca ocorre colisão
  - ◆ Chaves diferentes irão sempre produzir posições diferentes
- No pior caso, as operações de busca e inserção são sempre executadas em tempo constante, **O(1)**
  - ◆ É utilizado onde a colisão não é tolerável
  - ◆ Trata-se de um tipo de aplicação muito específica, por exemplo, o **conjunto de palavras reservadas de uma linguagem de programação**. Nesse caso, conhecemos previamente o conteúdo a ser armazenado na tabela

# Tratamento de Colisões

## ◆ Mundo ideal

- Hashing perfeito
  - ◆ Função de hashing irá sempre fornecer posições diferentes para cada uma das chaves inseridas

## ◆ Mundo real

- Independente da função de hashing utilizada, a mesma vai retornar a mesma **posição** para duas **chaves** diferentes: **colisão!**

# Tratamento de Colisões

- ◊ A criação de uma tabela hash consiste de duas coisas
  - uma **função de hashing**
  - uma **abordagem para o tratamento de colisões**

# Tratamento de Colisões

- ◆ Uma escolha adequada do **tamanho da tabela** pode minimizar as colisões
  - Colisões ocorrerem porque temos mais chaves para armazenar do que o tamanho da tabela suporta
  - Não há espaço suficiente para todas as chaves

# Tratamento de Colisões

- ◆ Uma escolha adequada da **função de hashing** pode minimizar as colisões
  - Escolher uma função que produza um espalhamento uniforme das chaves reduz o número de colisões
    - ◆ Infelizmente, não se pode garantir que as funções de hashing possuam um bom potencial de espalhamento porque as colisões também são uniformemente distribuídas
    - ◆ Colisões são teoricamente inevitáveis

# Tratamento de Colisões

- ◊ Colisões são teoricamente inevitáveis. Por isso, devemos sempre ter uma abordagem para tratá-las
  - Existem diversas formas de se tratar a colisão

# Técnicas de tratamento de colisões

## ◆ Endereçamento aberto

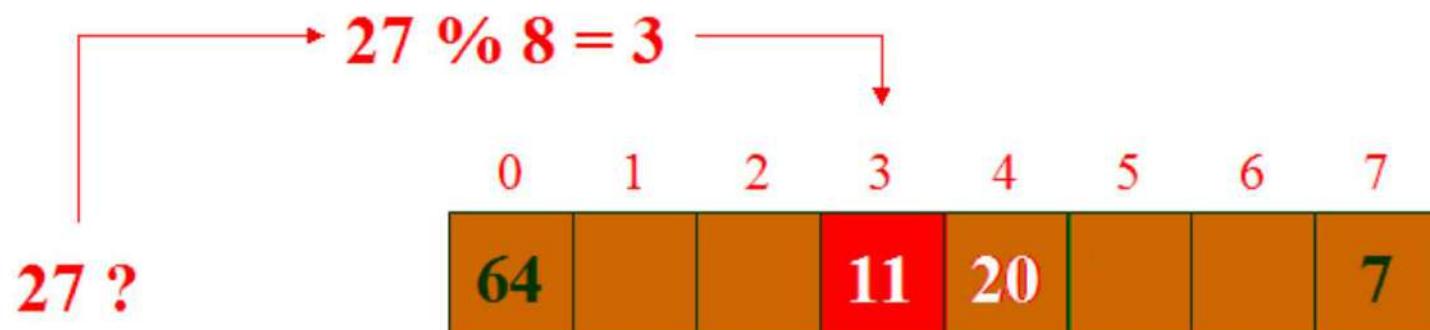
- Sondagem linear (linear probing)
  - ◆ Pega a primeira posição livre:  $F(k) = (k \bmod N) + 1$
- Sondagem quadrática
- Dispersão dupla (Hash duplo)
  - ◆ uma variação na forma de procurar uma posição livre
  - ◆ Exemplo:  $F'(k) = N - 2 - k \bmod (N - 2)$

## ◆ Lista encadeada

- Coloca os elementos em conflito em uma lista encadeada

# Endereçamento Aberto

- ◆ Os elementos são armazenados na própria tabela hash
  - Evita o uso de listas encadeadas
- ◆ No endereçamento aberto, quando uma nova chave é mapeada para uma posição já ocupada, uma nova posição é indicada para esta chave
- ◆ Exemplo:
  - Deve-se procurar por outra posição vaga (NULL) para o 27

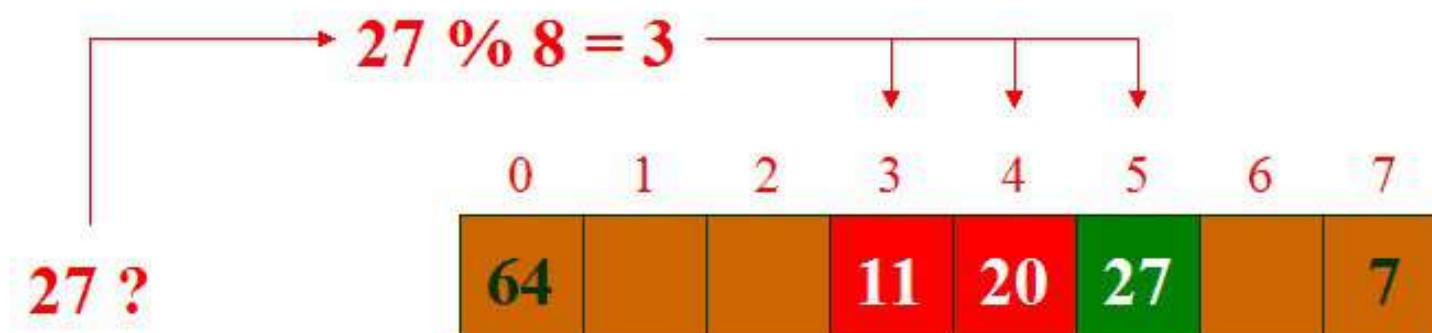


# Endereçamento Aberto

- ◆ Para a realização do cálculo da nova posição após a colisão, existem três estratégias muito utilizadas
  - Sondagem linear
  - Sondagem quadrática
  - Hash duplo

# Endereçamento Aberto: Sondagem Linear

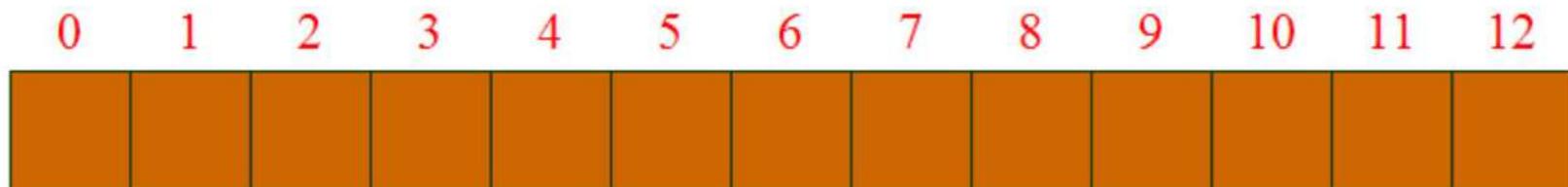
- ◆ Também conhecida como **procura linear**, **espalhamento linear** ou **rehash linear**
- ◆ Tenta espalhar os elementos de forma sequencial a partir da posição calculada utilizando a função de hashing
- ◆ Com sondagem linear, a nova posição é incrementada até que uma posição vazia seja encontrada:



# Endereçamento Aberto: Sondagem Linear

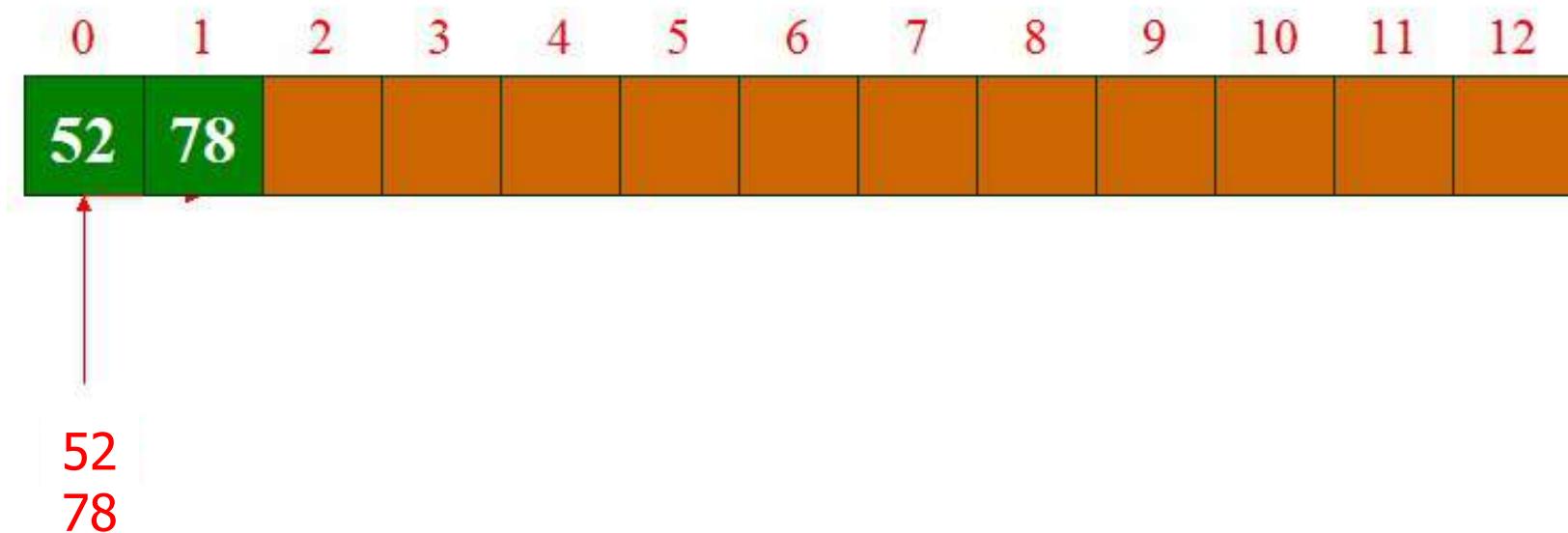
## ◆ Exercício 1

- Insira as chaves {52, 78, 48, 61, 81, 120, 79, 121, 92} em uma tabela T de tamanho 13 com hash linear e função:  **$h(k) = k \% 13$**



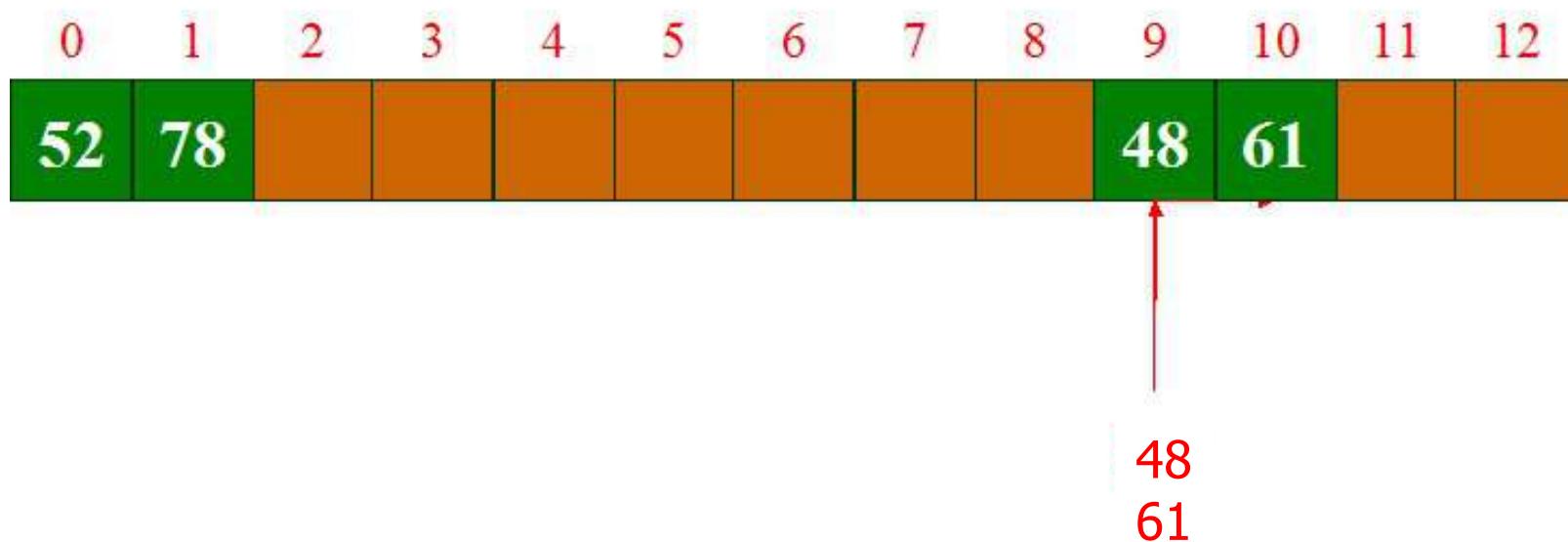
# Endereçamento Aberto: Sondagem Linear

- ◆ Valores: 52, 78, 48, 61, 81, 120, 79, 121, 92
- ◆ Função:  $h(k) = k \% 13$
- ◆ Tamanho da tabela: 13



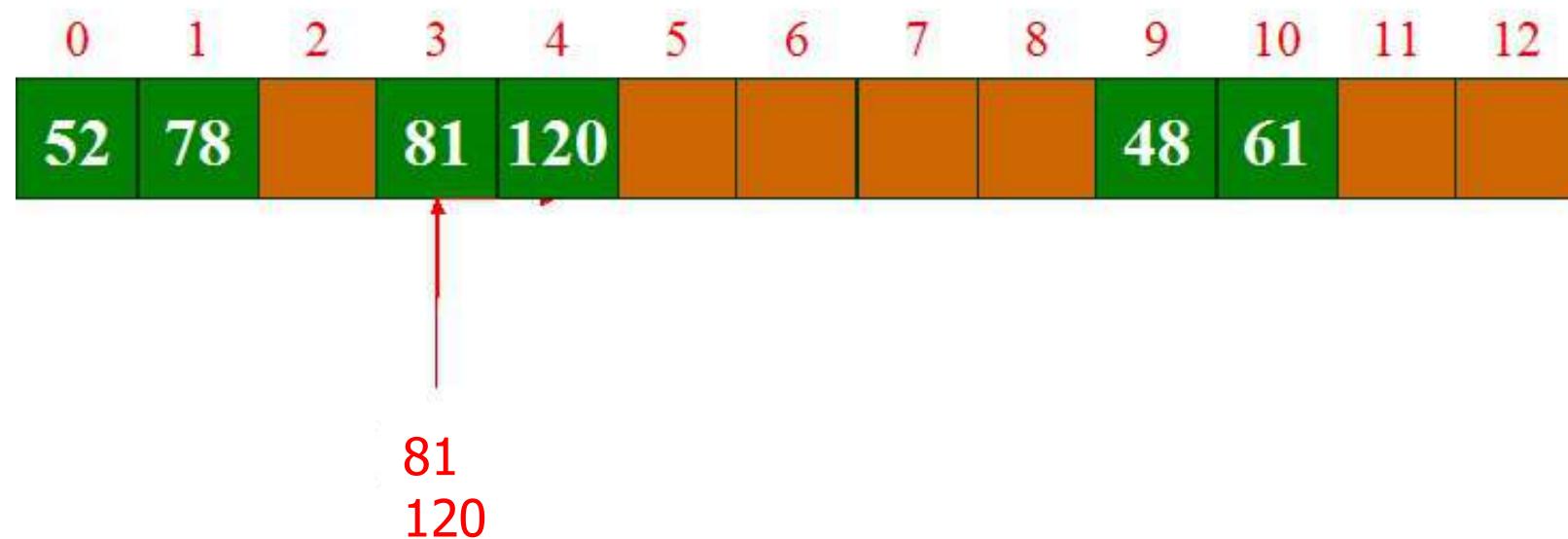
# Endereçamento Aberto: Sondagem Linear

- ◆ Valores: 52, 78, 48, 61, 81, 120, 79, 121, 92
- ◆ Função:  $h(k) = k \% 13$
- ◆ Tamanho da tabela: 13



# Endereçamento Aberto: Sondagem Linear

- ◆ Valores: 52, 78, 48, 61, 81, 120, 79, 121, 92
- ◆ Função:  $h(k) = k \% 13$
- ◆ Tamanho da tabela: 13



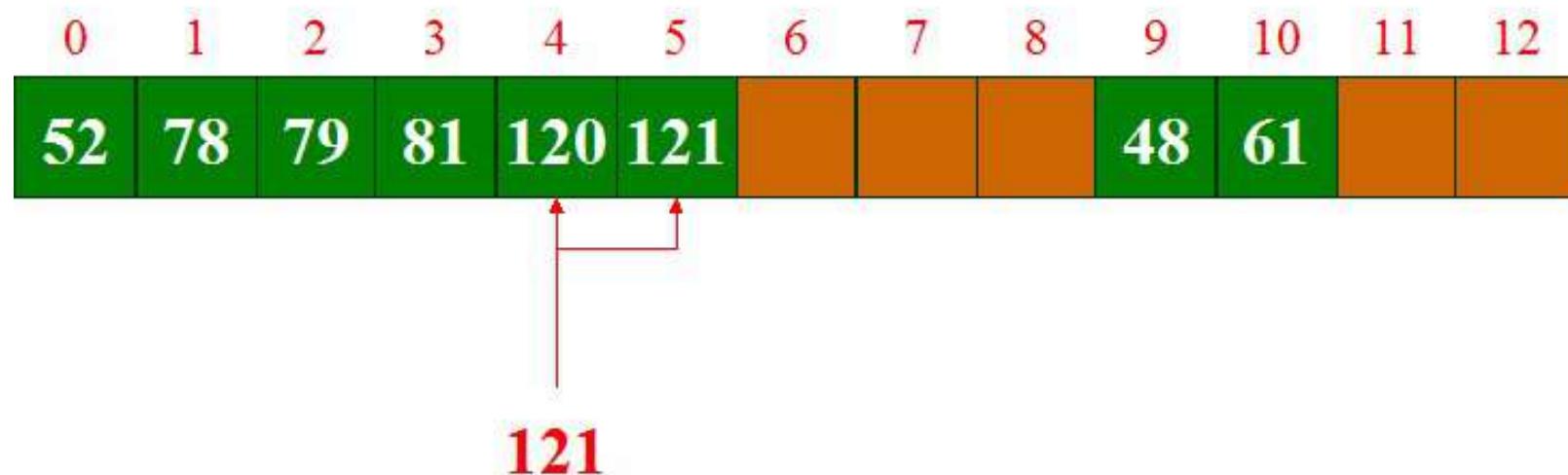
# Endereçamento Aberto: Sondagem Linear

- ◆ Valores: 52, 78, 48, 61, 81, 120, 79, 121, 92
- ◆ Função:  $h(k) = k \% 13$
- ◆ Tamanho da tabela: 13



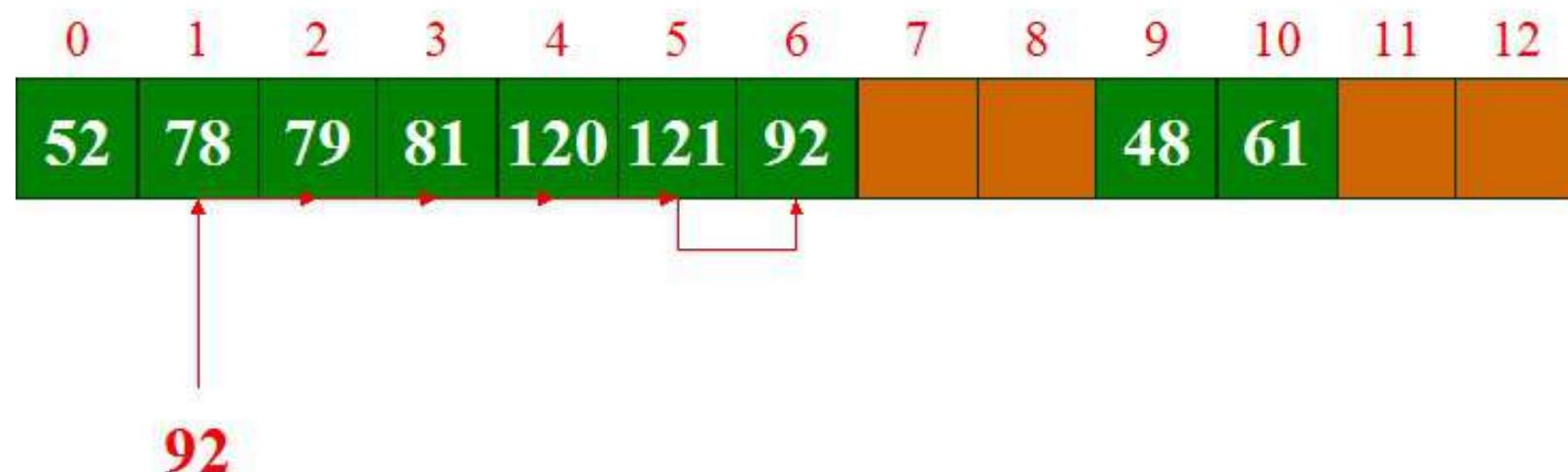
# Endereçamento Aberto: Sondagem Linear

- ◆ Valores: 52, 78, 48, 61, 81, 120, 79, 121, 92
- ◆ Função:  $h(k) = k \% 13$
- ◆ Tamanho da tabela: 13



# Endereçamento Aberto: Sondagem Linear

- ◆ Valores: 52, 78, 48, 61, 81, 120, 79, 121, 92
- ◆ Função:  $h(k) = k \% 13$
- ◆ Tamanho da tabela: 13



# Exemplo TAD Tabela Hash

◆ Inserção e busca com tratamento de colisão  
(endereçamento aberto e sondagem linear)

## ■ Busca

- ◆ Se uma chave  $x$  for mapeada pela função hashing para um determinado índice, procuramos a ocorrência do elemento a partir desse índice, até que o elemento seja encontrado ou uma posição vazia seja encontrada

# Exemplo TAD Tabela Hash

```
// Exemplo Aluno
struct aluno{
 int mat;
 char nome[81];
 char turma;
 char email[41];
};

typedef struct aluno Aluno;

#define N 127
typedef Aluno* Hash[N]; // vetor de ponteiros para a
 // estrutura aluno

int hash(int mat){ // função hashing
 return (mat%N);
}
```

# Exemplo TAD Tabela Hash

```
/* na operação de busca, procuramos a ocorrência
do elemento a partir do índice mapeado pela
função hash */

Aluno* hsh_busca (Hash tab, int mat) {
 int h = hash(mat);
 while (tab[h] != NULL) {
 if (tab[h]->mat == mat)
 return tab[h];
 h = (h + 1) % N;
 }
 return NULL;
}
```

# Exemplo TAD Tabela Hash

◆ Inserção e busca com tratamento de colisão  
(endereçamento aberto e sondagem linear)

## ■ Insere ou modifica

- ◆ Fazemos o mapeamento da chave de busca por meio da função hashing e verificamos se o elemento já existe na tabela
- ◆ Se existir, modificamos o seu conteúdo
- ◆ Se não existir, inserimos um novo na primeira posição livre encontrada na tabela, a partir do índice mapeado

# Exemplo TAD Tabela Hash

```
/* inserção */
Aluno* hsh_insere (Hash tab, int mat, char* n, char* e, char* t) {
 int h = hash(mat);
 while (tab[h] != NULL) {
 if (tab[h]->mat == mat)
 break;
 h = (h+1) % N;
 }
 if (tab[h] == NULL) { // não achou o elemento
 tab[h] = (Aluno*) malloc(sizeof(Aluno));
 tab[h]->mat = mat;
 }
 //atribui/modifica informação
 strcpy(tab[h]->nome, n);
 strcpy(tab[h]->email, e);
 tab[h]->turma = t;
 return tab[h];
}
```

# Endereçamento Aberto: Sondagem Linear

## ◆ Problemas:

- Apesar de bastante simples, sofre de um mal chamado **agrupamento**
  - ◆ Ocorre na medida que a tabela começa a ficar cheia
- Essa estratégia tende a concentrar os lugares ocupados na tabela
  - ◆ O ideal seria dispersar
- Deteriora o tempo de pesquisa

# Endereçamento Aberto: Sondagem Quadrática

- ◆ Também conhecida como **procura quadrática**, **espalhamento quadrático** ou **rehash quadrático**
- ◆ Tenta espalhar os elementos utilizando uma equação do segundo grau
- ◆ Exemplo
  - $\text{pos} + (c_1 * i) + (c_2 * i^2)$ 
    - ◆ pos é a posição obtida pela função de hashing
    - ◆ i é tentativa atual
    - ◆ c1 e c2 são os coeficientes da equação

# Endereçamento Aberto: Sondagem Quadrática

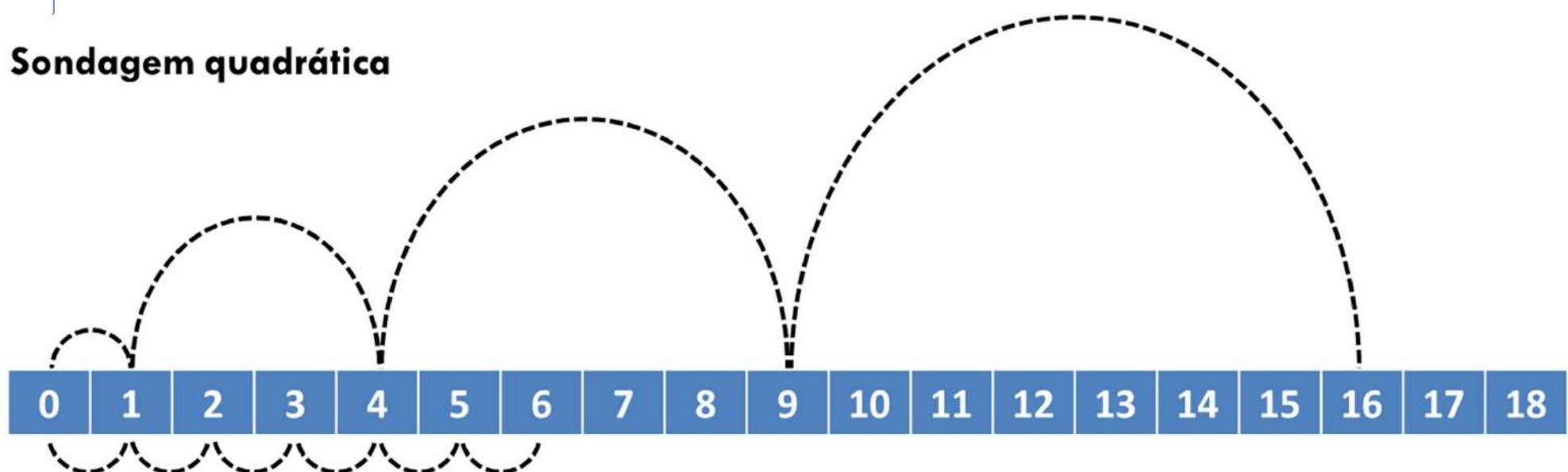
## ◆ Funcionamento

- Primeiro elemento ( $i = 0$ ) é colocado na **posição** obtida pela **função de hashing: pos**
- Segundo elemento (colisão) é colocado na posição **pos + (c<sub>1</sub> \* 1) + (c<sub>2</sub> \* 1<sup>2</sup>)**
- Terceiro elemento (nova colisão) é colocado na posição **pos + (c<sub>1</sub> \* 2) + (c<sub>2</sub> \* 2<sup>2</sup>)**

# Endereçamento Aberto: Sondagem Quadrática

## ◆ Exemplo ilustrativo

**Sondagem quadrática**



**Sondagem linear**

# Endereçamento Aberto: Sondagem Quadrática

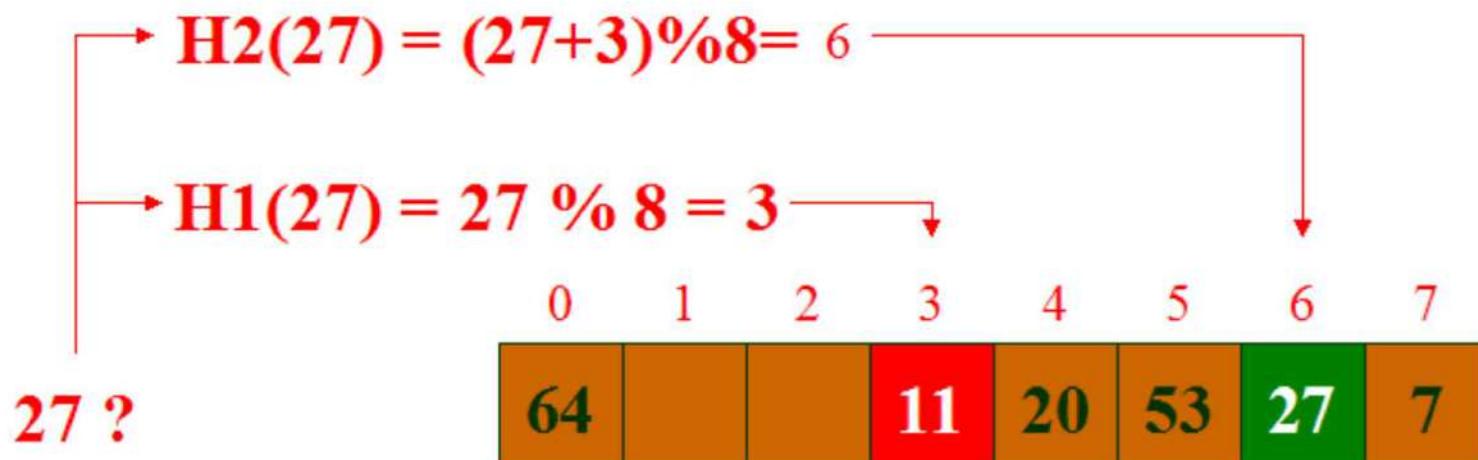
- ◆ Resolve o problema de agrupamento primário
- ◆ Porém, gera outro problema conhecido como **agrupamento secundário**
  - Todas as chaves que produzam a mesma posição inicial também produzem as mesmas posições na sondagem quadrática
  - Felizmente, a degradação produzida pelos agrupamentos secundários ainda é menor que a produzida pelos agrupamentos primários

# Endereçamento Aberto: Hash Duplo

◆ Outra política de endereçamento aberto é o chamado hash duplo: ao invés de incrementar a posição de 1, uma função auxiliar é utilizada para calcular o incremento. Esta função também leva em conta o valor da chave.

## ◆ Exemplo

- $H1 = \text{chave} \bmod M$
- $H2 = (\text{chave} + \text{offset}) \bmod M$



# Exemplo TAD Tabela Hash

- ◆ Inserção e busca com tratamento de colisão (endereçamento aberto e hash duplo)

- Uma possível segunda função hashing é

$$h'(x) = N - 2 - x\%(N/2)$$

- Onde  $x$  é a chave de busca e  $N$  a dimensão da tabela
  - De posse dessa segunda função, se houver colisão, procuramos uma posição livre na tabela com incrementos dados por  $h'(x)$

# Exemplo TAD Tabela Hash

- ◆ Inserção e busca com tratamento de colisão (endereçamento aberto e hash duplo)

- Uma possível segunda função hashing é

$$h'(x) = N - 2 - x\%(N/2)$$

- Onde  $x$  é a chave de busca e  $N$  a dimensão da tabela
  - Dois cuidados devem ser tomados na escolha dessa segunda função
    - ◆ Não pode retornar zero
      - pois isso não resultaria em incremento
    - ◆ Não deve retornar um número divisor da dimensão da tabela
      - Limita a procurar uma posição livre em um subconjunto restrito dos índices da tabela

# Exemplo TAD Tabela Hash

```
/* na operação de busca, procuramos a ocorrência do
 elemento a partir do índice mapeado pela função hash,
 considerando a segunda função de dispersão */

int hash2 (int mat) {
 return N - 2 - mat%(N-2);
}

Aluno* hsh_busca (Hash tab, int mat) {
 int h = hash(mat);
 int h2 = hash2(mat);
 while (tab[h] != NULL) {
 if (tab[h]->mat == mat)
 return tab[h];
 h = (h + h2) % N;
 }
 return NULL;
}
```

# Endereçamento Aberto

## ◆ Vantagens

- Maior número de posições na tabela para a mesma quantidade de memória usada no **encadeamento separado**
  - ◆ A memória utilizada para armazenar os ponteiros da lista encadeada no **encadeamento separado** pode ser aqui usada para aumentar o tamanho da tabela, diminuindo o número de colisões

# Endereçamento Aberto

## ◆ Vantagens

- Busca é realizada dentro da própria tabela
  - ◆ Recuperação mais rápida de elementos
- Voltada para aplicações com restrições de memória
- Ao invés de acessarmos ponteiros extras, calculamos a sequência de posições a serem armazenadas

# Endereçamento Aberto

## ◆ Desvantagens

- Maior esforço de processamento no cálculo das posições
- Esse esforço maior se deve ao fato de que, quando uma colisão ocorre, devemos calcular uma nova posição da tabela
  - ◆ Colisões sucessivas

# Lista encadeada

- ◆ Também conhecido como **separate chaining**
  - Não procura por posições vagas (valor NULL) dentro do array que define a tabela
  - Armazena dentro de cada posição do array o início de uma lista dinâmica encadeada
  - É dentro dessa lista que serão armazenadas as colisões (elementos com chaves iguais) para aquela posição do array

# Lista encadeada

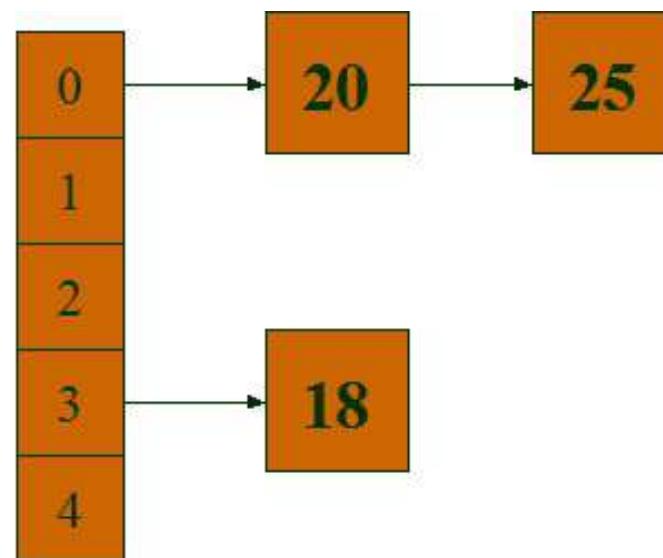
- ◆ Nesta estratégia, a posição de inserção não muda, logo, todos devem ser inseridos na mesma posição, por meio de uma lista ligada
- ◆ Característica principal:
  - A informação é armazenada em estruturas encadeadas fora da Tabela Hash

$$20 \% 5 = 0$$

$$18 \% 5 = 3$$

$$25 \% 5 = 0$$

*colisão com 20*



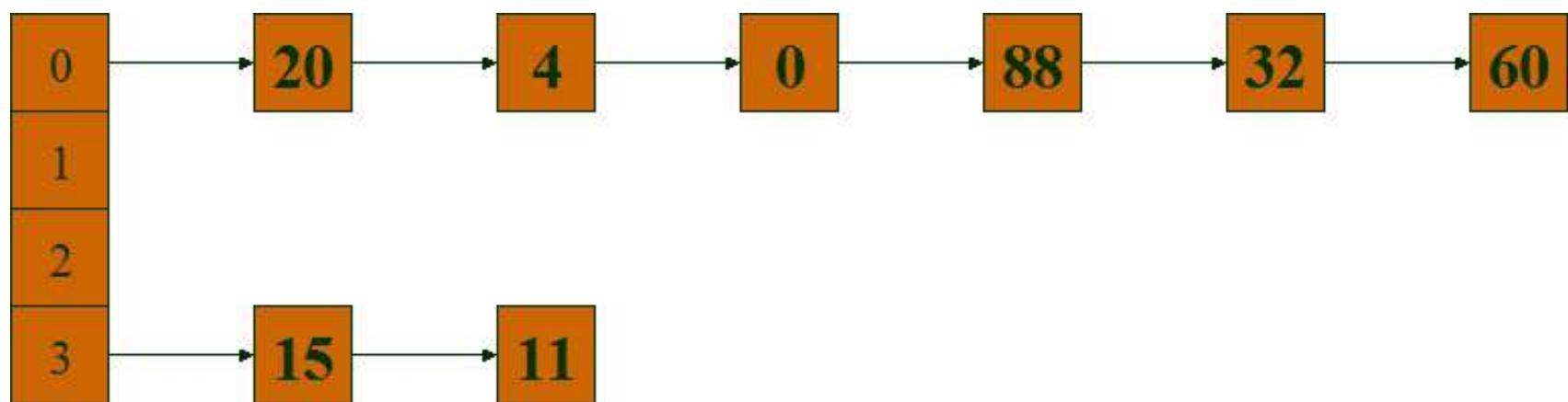
# Lista encadeada

## ◆ Características

- A lista dinâmica encadeada mantida em cada posição da tabela pode ser ordenada ou não
- Lista não ordenada
  - ◆ Inserção tem complexidade **O(1)** no pior caso: basta inserir o elemento no início da lista
  - ◆ Busca tem complexidade **O(M)** no pior caso: busca linear
- Desvantagem
  - ◆ Quantidade de memória consumida: gastamos mais memória para manter os ponteiros que ligam os diferentes elementos dentro de cada lista

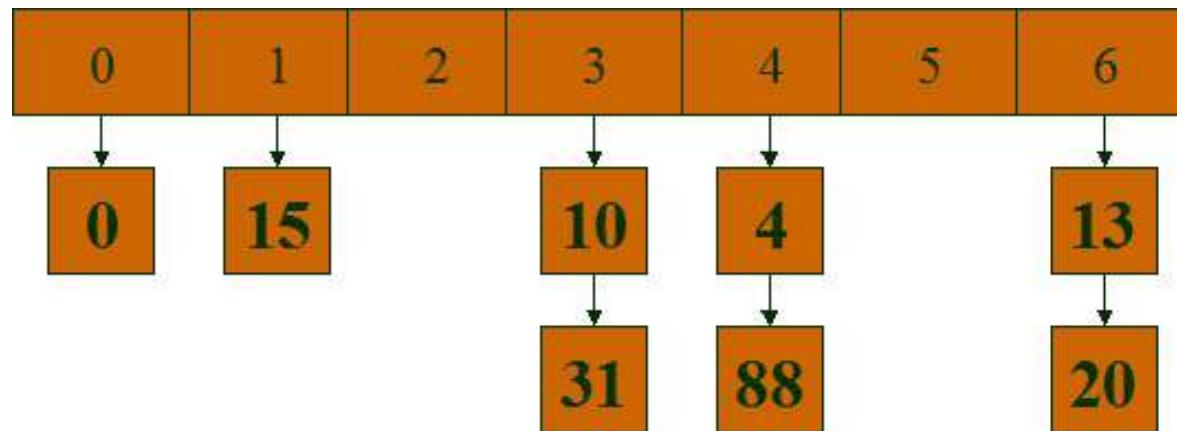
# Lista Encadeada

- ◆ A busca é feita do mesmo modo: calcula-se o valor da função hash para a chave, e a busca é feita na lista correspondente
- ◆ Se o tamanho das listas variar muito, a busca pode se tornar ineficiente, pois a busca nas listas é seqüencial:



# Lista Encadeada

- ◆ Por esta razão, a função hash deve distribuir as chaves entre as posições uniformemente:



- ◆ Se o tamanho da tabela for um número primo, há mais chances de ter uma melhor distribuição

# Exercício

- a) Desenhe o conteúdo da tabela hash resultante da inserção de registros com as chaves N I V O Z A P Q R S T U, nesta ordem, em uma tabela inicialmente vazia de tamanho 7 (sete), usando listas encadeadas. Use a função hash  $h(k) = k \bmod 7$  para a késima letra do alfabeto.
  
- a) Desenhe o conteúdo da tabela hash resultante da inserção de registros com as chaves N I V O Z A P Q R S T U, nesta ordem, em uma tabela inicialmente vazia de tamanho 13 (treze), usando endereçamento aberto e hash linear para resolver as colisões. Use a função  $h(k) = k \bmod 13$  para a k-ésima letra do alfabeto.

# Exemplo TAD Tabela Hash

- ◆ Inserção e busca com tratamento de colisão (lista encadeada)
- ◆ Com essa estratégia, cada elemento armazenado na tabela será um elemento de uma lista encadeada
- ◆ É preciso prever na estrutura da informação um ponteiro adicional para o próximo elemento da lista

# Exemplo TAD Tabela Hash

```
// Exemplo Aluno
struct aluno{
 int mat;
 char nome[81];
 char turma;
 char email[41];
 struct aluno* prox; // encadamento na lista de colisão
};
typedef struct aluno Aluno;
```

# Exemplo TAD Tabela Hash

```
/* na operação de busca, procuramos a ocorrência
do elemento na lista representada no índice
mapeado pela função hashing */

Aluno* hsh_busca (Hash tab, int mat) {
 int h = hash(mat);
 Aluno* a = tab[h];
 while (a != NULL) {
 if (a->mat == mat)
 return a;
 a = a->prox;
 }
 return NULL;
}
```

# Exemplo TAD Tabela Hash

```
/* inserção */
Aluno* hsh_insere (Hash tab, int mat, char* n, char* e, char* t) {
 int h = hash(mat);
 Aluno* a = tab[h];
 while (a != NULL) {
 if (a->mat == mat)
 break;
 a = a->prox;
 }
 if (a == NULL) { // não achou o elemento
 a = (Aluno*) malloc(sizeof(Aluno));
 a->mat = mat;
 a->prox = tab[h];
 tab[h] = a;
 }
 //atribui/modifica informação
 strcpy(a->nome,n);
 strcpy(a->email,e);
 a->turma = t;
 return a; }
```

# Considerações finais

- ◆ O aspecto negativo do método hashing está relacionado ao pior caso → **O(N)**
  - se a função não espalhar bem os registros → forma-se uma longa lista linear
- ◆ O melhor caso e o caso médio → **O(1)**
- ◆ **Vantagens**
  - Alta eficiência no custo de pesquisa
  - Simplicidade de implementação
- ◆ **Desvantagens**
  - Custo alto para a recuperação dos registros na ordem lexicográfica → ordenar
  - Pior caso **O(N)**

# Aplicações

- ◆ A tabela hash pode ser utilizada para
  - verificação de integridade de dados e autenticação de mensagens
    - ◆ os dados são enviados juntamente com o resultado da função de hashing
    - ◆ Quem receber os dados recaculta a função de hashing usando os dados recebidos e compara o resultado obtido com o que ele recebeu
    - ◆ Resultados diferentes: erro de transmissão
  - armazenamento de senhas com segurança
    - ◆ a senha não é armazenada no servidor, mas sim o resultado da função de hashing

# Aplicações

- ◆ A tabela hash pode ser utilizada para
  - implementação da tabela de símbolos dos compiladores
  - Criptografia
    - ◆ MD5 e família SHA (Secure Hash Algorithm)
  - busca de elementos em base de dados
    - ◆ estruturas de dados em memória, bancos de dados e **mecanismos de busca na Internet**

# Aplicações

## ◆ A tabela hash pode ser utilizada para

- Distribuidor de carga de uma máquina de busca:
  - ◆ Tarefa: identificar nó que contém site indexado
    - Como indexar 5 bilhões de domínios com complexidade de tempo  $O(1)$  ?
  - ◆ Curioso?
    - <https://doi.org/10.1016/j.is.2012.06.002>
    - em <https://www.periodicos.capes.gov.br> faça login com e-mail @ufu em Acesso Café e pesquise por 10.1016/j.is.2012.06.002

The screenshot shows a web browser window with the URL <https://www.periodicos.capes.gov.br/index.php?> in the address bar. The page is from the CAPES Periodicals portal. At the top, there are links for 'gov.br' (Ministério da Educação/CAPES), 'Órgãos do Governo', 'Acesso à Informação', 'Legisla', and navigation links for 'Sobre', 'Acervo', 'Treinamentos', and 'Informativo'. A red arrow points upwards from the 'Sobre' link towards the 'gov.br' logo. Another red arrow points downwards from the 'Acervo' link towards the search bar. The search bar contains the text 'Olá.' followed by the DOI '10.1016/j.is.2012.06.002'. To the right of the search bar is a magnifying glass icon. Below the search bar, a message reads 'Aqui você encontra conteúdo científico diversificado para deixar sua pesquisa ainda melhor.' On the right side, there is a 'Destques' section with a red banner and a 'GUIA' button.

# Estruturas de dados para pesquisa: Comparação

## ◆ Hash

- Excelente para acesso direto
- Pobre para acesso seqüencial

## ◆ Lista

- Excelente para acesso seqüencial
- Pobre para acesso direto

## ◆ Árvore binária

- Bom compromisso entre acesso direto e seqüencial

## ◆ E vetor ordenado?

- Mantê-lo ordenado tem custo  $O(N)$ !

# Métodos de Pesquisa: Comparação

| Método                                        | Melhor Caso | Caso Médio  | Pior Caso   |
|-----------------------------------------------|-------------|-------------|-------------|
| Pesquisa seqüencial<br>(dados não ordenados)  | $O(1)$      | $O(n)$      | $O(n)$      |
| Pesquisa seqüencial<br>(dados ordenados)      | $O(1)$      | $O(n)$      | $O(n)$      |
| Pesquisa binária<br>(dados ordenados)         | $O(1)$      | $O(\log n)$ | $O(\log n)$ |
| Árvore binária de pesquisa<br>(desbalanceada) | $O(1)$      | $O(n)$      | $O(n)$      |
| Árvore binária de pesquisa<br>(balanceada)    | $O(1)$      | $O(\log n)$ | $O(\log n)$ |
| Hashing                                       | $O(1)$      | $O(1)$      | $O(1)*$     |

(\*) na teoria seria  $O(n)$ , no caso de tamanho da tabela e função mal escolhidas

# Referências

- ◆ Capítulo 11 do Cormen. Algoritmos: Teoria e Prática.
- ◆ Seção 5.5 do Ziviani. Projeto de Algoritmos.
- ◆ Capítulo 18 do Celes. Introdução a Estruturas de Dados.
- ◆ Slides do Prof. André Backes. Linguagem C Descomplicada – Estrutura de Dados.

# Material Complementar

## ◆ Vídeo Aulas

- Aula 89 -98:

<https://programacaodescomplicada.wordpress.com/indice/estrutura-dados/>

- Lista de exercícios na equipe da disciplina no Teams

# Prova 2

## ◆ Matéria

- Grafos (Dijkstra, ordenação topológica e componentes fortemente conectados)
- Pesquisa (Busca)
- Árvores

## ◆ Data

- 16/04

# Bacharelado em Ciência da Computação

## GBC034 Algoritmos e Estruturas de Dados 2

### Trabalho 1 – Jogo de Labirinto

#### 1. Informações Básicas

Este trabalho tem por objetivo reforçar o conhecimento dos discentes com relação aos algoritmos para modelagem de problemas de grafos, árvores binárias e ordenação.

- Data da entrega: 07/04/2023
- Grupo de até 3 (três) alunos(as)
- Linguagem de programação a ser usada: C
- O que deve ser entregue: Apresentação em PDF e link para o repositório do código (Replit.com)

#### 2. Descrição

O objetivo do trabalho é projetar um jogo interativo de labirinto utilizando os conceitos de Grafos, Árvores Binárias e Ordenação. O labirinto é composto por cinco partes importantes:

- 1) **Área**, em que o jogador interage diretamente escolhendo para qual **sala** irá seguir com o objetivo de sair do local, **sem chance de retorno**. A escolha do **caminho** que o jogador seguirá pode resultar em uma **sala sem saídas** ou em uma sala com a **saída** da área atual. Caso o jogador caia em uma sala sem saída, deverá recomeçar o percurso da sala inicial daquela área (**derrota**). Caso o jogador encontre a saída da área atual, ele avança para outra área do mapa em busca da **Área Central** do labirinto (**avanço de fase**).
- 2) **Área Central**, diferente das demais salas contabiliza uma **pontuação** para cada avanço e permite que o jogador retroceda uma sala sacrificando um ponto. A **derrota** só ocorre caso o jogador encontre uma sala vazia sem pontos acumulados para usar o retorno.
- 3) **Dificuldade do percurso**, determinada pelo número de salas que a área inicial possui, e à medida que ocorre um avanço de fase, o número de salas da próxima área aumenta em uma unidade em relação à anterior.
- 4) **Mapa do labirinto**, composto por **caminhos de áreas** que levam a Área Central do labirinto. Cada **região** do mapa determina um conjunto de áreas, e o **progresso** para uma região nova é feito completando todas as áreas da região antecessora a ela. Todas as áreas levam ao fim do labirinto (Área Central) independente do caminho. A visão do percurso é mostrada após cada avanço de fase, e nela o jogador poderá ver por quantas áreas já passou.
- 5) **Sistema de Ranking**, baseado no tempo de cada partida e na pontuação feita na Área Central. O tempo é contado a partir do início do percurso e

# Bacharelado em Ciência da Computação

## GBC034 Algoritmos e Estruturas de Dados 2

### Trabalho 1 – Jogo de Labirinto

termina ao completar o labirinto, sendo que a cada avanço de fase é também contabilizado o tempo que foi gasto para completar uma área sem desconsiderar as derrotas (recomeços). O sistema de ranking armazena somente as últimas 10 vitórias, e organiza de forma crescente as informações do jogador (nome, tempo total, tempo gasto em cada área e pontuação feita na Área Central).

#### Detalhes de funcionamento:

- O grupo deve considerar:
  - O uso de grafo direcionado não-ponderado para cada **Área** (item 1), sendo que cada sala é um vértice e o caminho para outra sala é uma aresta;
  - O uso de grafo direcionado ponderado para a **Área Central** (item 2), sendo que cada sala é um vértice e o caminho para outra sala é uma aresta ponderada;
  - O uso de árvore binária para o **Mapa** do labirinto (item 4), sendo que cada área é um nó e a Área Central do labirinto é a raiz;
  - O uso de um vetor para armazenar as informações sobre os últimos 10 jogos que tenham obtido melhor pontuação, e organizá-los de forma crescente de acordo com um critério escolhido pelo usuário por meio de um algoritmo de ordenação. Além disso, o programa deve permitir que o usuário consiga acessar a exibição de ranking no menu principal de acordo com um critério escolhido (exemplo, ranking por tempo total ou por pontuação), limpar o histórico e apagar pontuações isoladas.

#### Observações:

- Cada grupo deve preparar um “manual” explicando como jogar o jogo;
- Deve ser entregue um cenário de configuração para o jogo (área, área central, mapa, dificuldade);
- O trabalho deve ser entregue na forma de uma apresentação em PDF que contenha a explicação das soluções adotadas pelo grupo para cada item da descrição do problema com ilustrações, uma visão geral do código com a indicação das referências usadas e casos de uso (exemplos de execução). No primeiro slide da apresentação devem constar os nomes e números de matrícula dos discentes e o link para o repositório em que o código está armazenado (Replit.com). A apresentação deverá estar em formato PDF e deve ser preparada levando em consideração que cada grupo terá 15 minutos para apresentar e demonstrar o trabalho;

# Bacharelado em Ciência da Computação

## GBC034 Algoritmos e Estruturas de Dados 2

### Trabalho 1 – Jogo de Labirinto

- O código deverá estar disponível em um repositório online e seu endereço indicado no documento da apresentação;
- Para o dia da apresentação, cada grupo deverá preparar um roteiro para demonstrar o jogo criado.

#### Critérios de avaliação:

- (a) Apresentação: objetividade, clareza, organização do texto/algoritmo/código, formatação/ilustração, roteiro de demonstração.
- (b) Programa: qualidade dos dados informados / qualidade da impressão visão do percurso / modularização / uso correto das estruturas de dados / legibilidade e consistência na codificação das funções de manipulação do grafo, da árvore e do algoritmo de ordenação/ documentação do código.