

Introdução a Complexidade de Algoritmos

Estrutura de Dados

Prof. Anselmo C. de Paiva

Departamento de Informática – Núcleo de Computação Aplicada NCA-UFMA

Introdução

- ▶ Problema Importante:
 - ▶ escolha das estruturas de dados adequadas para uma dada aplicação.
- ▶ Exemplo:
 - ▶ Considere o problema de ordenar n números.
 - ▶ algoritmo simples: baseado na comparação de cada número com todos os outros n^2 comparações.
 - ▶ Para $n=10$, estamos falando de umas 100 comparações.
 - ▶ Para ordenar um milhão de números 10^{12} comparações
 - ▶ Algoritmo melhor - 20×10^6 comparações
 - ▶ Considerando um milhão de comparações por segundo:
 - um milhão de segundos (100 dias)
 - 20 segundos

Introdução

- ▶ Opção:
 - ▶ escolher a implementação mais adequada através de medidas reais de desempenho das várias implementações.
 - ▶ Problemas:
 - escolha dos casos para teste, porque uma forma de implementação pode ser boa para alguns casos e ruim para outros, e pode ser difícil decidir quais são os melhores e os piores casos.
 - levar em consideração todos os casos é impensável.
- ▶ Exemplo:
 - ▶ Estudar o desempenho dos dois algoritmos de ordenação.
 - ▶ números são inteiros de 16 bits - 65000 valores possíveis para cada um dos 10^6
 - ▶ 6.5×10^{10} casos.

Algoritmos e Complexidade

- ▶ Eficiência de um algoritmo
 - ▶ Complexidade de tempo: quanto “tempo” é necessário para computar o resultado para uma instância do problema de tamanho n
 - ▶ Pior caso: Considera-se a instância que faz o algoritmo funcionar mais lentamente
 - ▶ Caso médio: Considera-se todas as possíveis instâncias e mede-se o tempo médio
- ▶ Eficiência de uma estrutura de dados
 - ▶ Complexidade de espaço: quanto “espaço de memória/disco” é preciso para armazenar a estrutura (pior caso e caso médio)
- ▶ Complexidade de espaço e tempo estão freqüentemente relacionadas

Tempo de execução de um programa

- ▶ Questões:
 - ▶ em que máquina deve ser realizada a execução
 - ▶ que outros programas estão em execução nessa máquina ao mesmo tempo
 - ▶ que informação está disponível nos caches de memória e de disco por ocasião da execução
- ▶ Levam à procura da avaliação da forma de variação do tempo de execução de um algoritmo com o “tamanho do problema”, em geral representado por uma variável n .



Algoritmos e Complexidade

- ▶ Eficiência medida objetivamente depende de:
 - ▶ Como o programador implementou o algoritmo/ED
 - ▶ Características do computador usado para fazer experimentos:
 - ▶ Velocidade da CPU
 - ▶ Capacidade e velocidade de acesso à memória primária / secundária
 - ▶ Etc
 - ▶ Linguagem / Compilador / Sistema Operacional / etc
- ▶ Portanto, a medição formal de complexidade tem que ser subjetiva, porém matematicamente consistente
 - ▶ Complexidade assintótica

Complexidade Assintótica

- ▶ Tempo / espaço medidos em número de “passos” do algoritmo / “palavras” de memória ao invés de segundos ou bytes
- ▶ Análise do algoritmo / e.d. permite estimar uma função que depende do tamanho da entrada / número de dados armazenados (n).
 - ▶ Ex.: $T(n) = 13n^3 + 2n^2 + 6n \log n$
- ▶ Percebe-se que à medida que n aumenta, o termo cúbico começa a dominar
- ▶ A constante que multiplica o termo cúbico tem relativamente a mesma importância que a velocidade da CPU / memória
- ▶ Diz-se que $T(n) \in O(n^3)$. Big-oh
- ▶ Ozao - KNUTH
- ▶ $T(n)$ é $O(f(n))$. $F(n) = n^3$ •

Definição de O(f(n))

- ▶ $f(n)$ função de números naturais
- ▶ Definição:
 - ▶ Dizemos que uma função $T(n)$ é $O(f(n))$ se existem constantes n_0 e $c > 0$ tais que para todo $n > n_0$, $T(n) < c f(n)$.
- ▶ Exemplo: mostrar que $T(n)=5n^2 + 5n - 6$ é $O(n^2)$.
 - ▶ escolher valores apropriados de c e de n_0 .
 - ▶ Tomando $c=6$, podemos descobrir um valor adequado de n_0 escrevendo
 - ▶ $5n^2 + 5n - 6 < 6n^2 \Rightarrow 5n^2 - 6n^2 + 5n < 6 \Rightarrow -n^2 + 5n < 6 \Rightarrow n^2 - 5n + 6 > 0$
 - ▶ $n^2 - 5n + 6 > 0$
 - ▶ obtendo-se, assim $n > 3$
 - ▶ Como $5n^2 + 5n - 6 < 6n^2$ para $n > 3$, $T(n)$ é $O(n^2)$.



Complexidade Assintótica

- ▶ Definição:
 - ▶ $T(n) \in O(f(n))$ se existem constantes c e n_0 tais que
$$T(n) \leq c f(n) \text{ para todo } n \geq n_0$$
- ▶ Alternativamente,
 - ▶ $T(n) \in O(f(n))$ se $\lim_{n \rightarrow \infty} T(n) / f(n)$ é constante (mas não infinito)

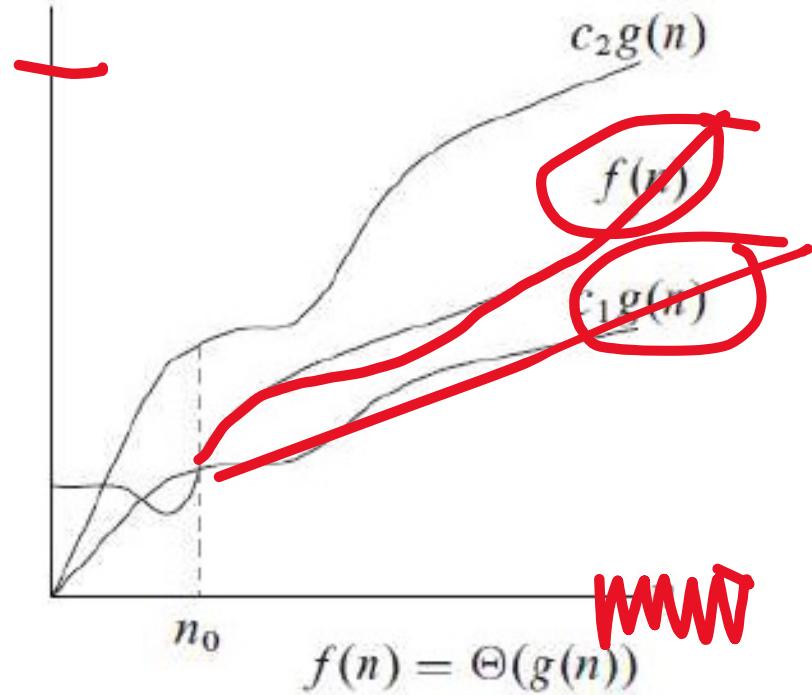
$f(n) \in O(g(n))$

$f(n) < g(n)$

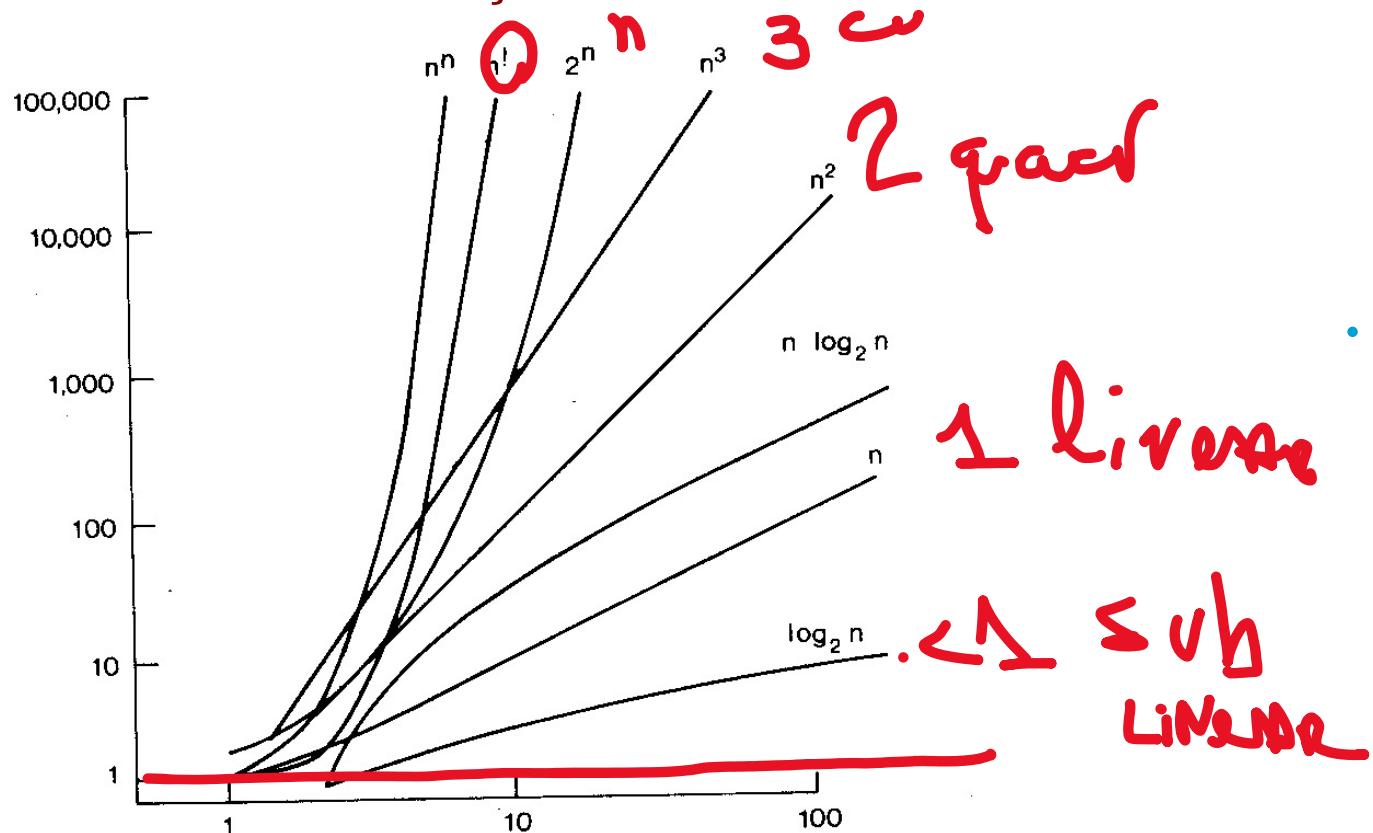
$T(n) \in O(n^2)$

•

Dizemos que uma função $f(n)$ é $O(g(n))$ se existem constantes n_0 e $c_i > 0$ tais que para todo $n > n_0$, $T(n) < c f(n)$.



Ordem de Crescimento de Funções



Ordem dos Algoritmos

- ▶ “ordem” dos algoritmos é informação adequada para escolha de um algoritmo
- ▶ Problema:
 - ▶ Vários algoritmos de mesma ordem estão disponíveis.
- ▶ Exemplo: Para resolver um problema, estão disponíveis dois algoritmos:
 - Algoritmo A, com tempo $f(n) = 10000 n$.
 - Algoritmo B, com tempo $g(n) = n^2$.
 - ▶ A é $O(n)$, e B é $O(n^2)$
 - ▶ algoritmo A é melhor que algoritmo B
 - ▶ $f(n) < g(n)$, para $n > 10000$
 - ▶ para valores grandes de n, o algoritmo A é mais rápido.
 - ▶ Entretanto, $g(n) > f(n)$, para $n < 10000$
 - ▶ para problemas pequenos, o algoritmo B seria preferível.

Caso médio e pior caso

- ▶ Algoritmo para ordenar os elementos de um vetor de tamanho n
 - ▶ Pode gastar menos tempo quando o vetor já se encontra ordenado, ou quase ordenado,
 - ▶ Pode gastar mais tempo quando os elementos de vetor se encontram “embaralhados”
- ▶ Opções:
 - ▶ Analisar o algoritmo e determinar o tempo de execução no pior caso
 - ▶ Realizar a análise levando em consideração todos os casos, juntamente com suas probabilidades de ocorrência, e obter um tempo de execução médio

Caso médio e pior caso

- ▶ Considere o problema de inserir elementos em uma lista encadeada ordenada, com n elementos.
 - ▶ Pior caso: Elemento precisa ser inserido no fim da lista
 - ▶ percorre toda a lista ($k_1 n$)
 - ▶ tempo constante k_2 da inserção propriamente dita.
 - ▶ $T(n) = k_1 n + k_2$, ou seja, $O(n)$.
 - ▶ Caso médio: elementos a serem inseridos se distribuem uniformemente entre os já existentes.
 - ▶ Em média, será necessário percorrer meia lista, para realizar a inserção
 - ▶ $T(n) = k_1 (n/2) + k_2$, ou seja, $O(n)$

Tempo de execução em algoritmos

- ▶ Tempo gasto por um algoritmo
 - ▶ soma os tempos gastos por suas componentes.
- ▶ Exemplo: Avaliar o tempo consumido pela execução de

```
▶ for (i=0; i<n; i++) {  
    ▶ a=10;  
    ▶ }
```

- ▶ tempo correspondente à atribuição $i=0$ 
- ▶ n vezes o tempo consumido pela atribuição de 10 para a variável a 
- ▶ n vezes o tempo consumido pelo teste $i < n$ 
- ▶ n vezes o tempo consumido pelo incremento $i++$ 
- ▶ $T(n) = 1 + n + n + n = 1 + 3n$
 - ▶ da forma $k_1 + k_2n$
 - ▶ tempo de execução é $O(n)$.

```
for (i=0; i<n; i++) {  
    for (j=0; j<n; j++) {  
        a=10;  
    }  
}
```

execução de uma operação
toma uma unidade de tempo;

Tempo de execução em algoritmos

- ▶ Tempo gasto por um algoritmo
 - ▶ soma os tempos gastos por suas componentes.
- ▶ Exemplo: Avaliar o tempo consumido pela execução de

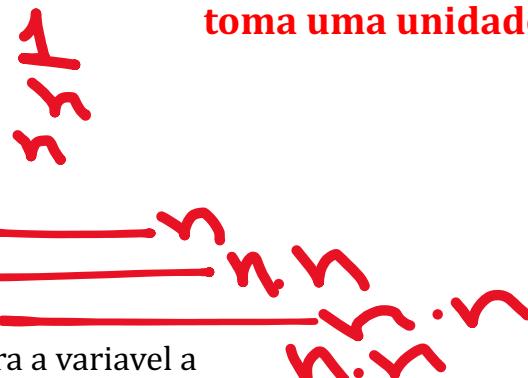
```
▶ for (i=0; i<n; i++) {  
    for (j=0; j<n; j++){  
        a=10;  
    }  
}
```

- ▶ 1 vez o tempo correspondente à atribuição $i=0$
- ▶ n vezes o tempo consumido pelo teste $i < n$
- ▶ n vezes o tempo consumido pelo incremento $i++$
- ▶ N vezes a execução do for interno

- ▶ { 1 vez o tempo correspondente à atribuição $j=0$
 ▶ n vezes o tempo consumido pelo teste $j < n$
 ▶ n vezes o tempo consumido pelo incremento $j++$
 ▶ n vezes o tempo consumido pela atribuição de 10 para a variável a

- ▶ $T(n) = 1 + n + n + n + n \cdot n + n \cdot n + n \cdot n = 1 + 3n + 3n^2$
 - ▶ da forma $k_1 + k_2n + k_3n^2$
 - ▶ tempo de execução é $O(n^2)$.

execução de uma operação
toma uma unidade de tempo;



Classes de Comportamento Assintótico

- ▶ determinam a complexidade inerente do algoritmo
- ▶ Comportamento assintótico é medido quando o tamanho da entrada (n) tende a infinito
 - ▶ constantes são ignoradas e apenas o componente mais significativo da função de complexidade é considerado
- ▶ Dois algoritmos da mesma classe de comportamento assintótico são ditos equivalentes



Principais Classes de Problemas

- ▶ $f(n) = O(1)$
 - ▶ Complexidade constante.
 - ▶ Tempo de execução independe de n .
 - ▶ instruções do algoritmo são executadas um número fixo de vezes.
- ▶ $f(n) = O(\log n)$.
 - ▶ Complexidade logarítmica.
 - ▶ Típico em algoritmos dividir pra conquistar
 - ▶ transformam um problema em outros menores
 - ▶ Quando n é mil, $\log_2 n \approx 10$, quando n é 1 milhão, $\log_2 n \approx 20$.
 - ▶ Para dobrar o valor de $\log n$ temos de considerar o quadrado de n .
 - ▶ A base do logaritmo muda pouco estes valores: quando n é 1 milhão, o $\log_2 n$ é 20 e o $\log_{10} n$ é 6.
- ▶ $f(n) = O(n)$
 - ▶ Complexidade linear.
 - ▶ Pequeno trabalho realizado sobre cada elemento de entrada.
 - ▶ Melhor situação para algoritmo que processa/produz n elementos de entrada/saída.
 - ▶ n dobrou de tamanho, o tempo de execução dobrou.
- ▶ $f(n) = O(n \log n)$
 - ▶ Típico em algoritmos dividir pra conquistar
 - ▶ transformam um problema em outros menores
 - ▶ n é 1 milhão, $n \log_2 n$ é cerca de 20 milhões.
 - ▶ n é 2 milhões, $n \log_2 n$ é cerca de 42 milhões,
 - ▶ pouco mais do que o dobro.

▶ $f(n) = O(n^2)$

- ▶ complexidade quadrática.

~~• itens de dados são processados aos pares~~

- ▶ n é mil, o número de operações é da ordem de 1 milhão.
- ▶ n dobra, o tempo de execução é multiplicado por 4.
- ▶ Úteis para resolver problemas de tamanhos relativamente pequenos.

▶ $f(n) = O(n^3)$

~~• Complexidade cúbica.~~

- ▶ Úteis apenas para resolver pequenos problemas.
- ▶ n é 100, o número de operações é da ordem de 1 milhão.
- ▶ n dobra, o tempo de execução fica multiplicado por 8.

▶ $f(n) = O(2^n)$

~~• Complexidade exponencial.~~

- ▶ Geralmente não são úteis sob o ponto de vista prático.
- ▶ Ocorrem na solução de problemas quando se usa força bruta para resolvê-los.
- ▶ n é 20, o tempo de execução é cerca de 1 milhão.
- ▶ n dobra, o tempo fica elevado ao quadrado.

▶ $f(n) = O(n!)$

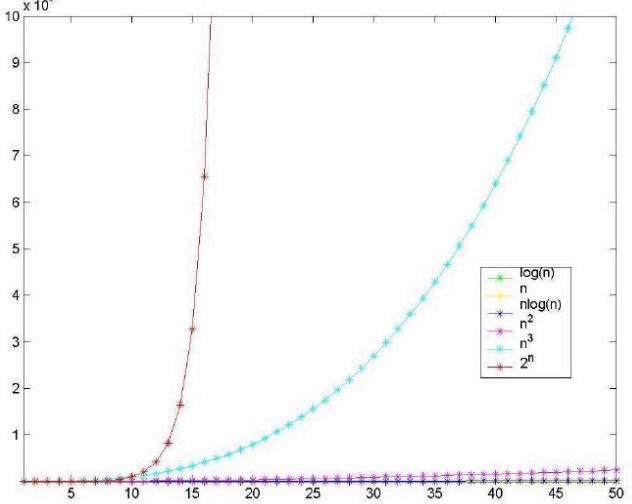
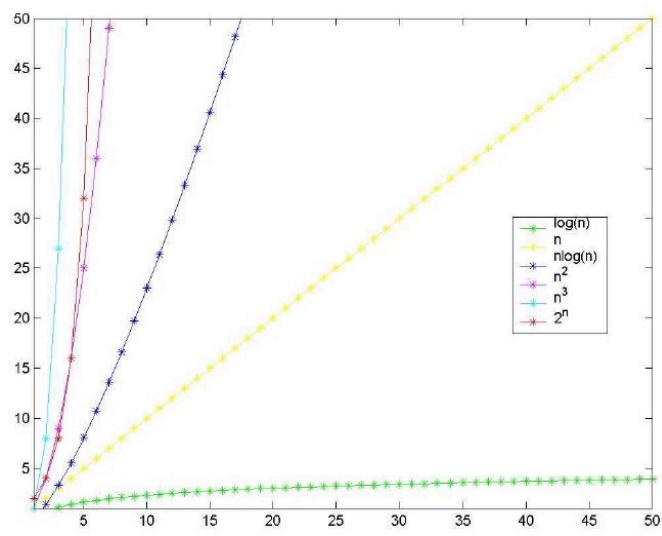
~~• Complexidade exponencial~~

- ▶ ~~$O(n!)$ é muito pior do que $O(2^n)$.~~
- ▶ força bruta para solução do problema.
- ▶ $n = 20 \Rightarrow 20! = 2432902008176640000$, um número com 19 dígitos.
- ▶ $n = 40 \Rightarrow$ um número com 48 dígitos

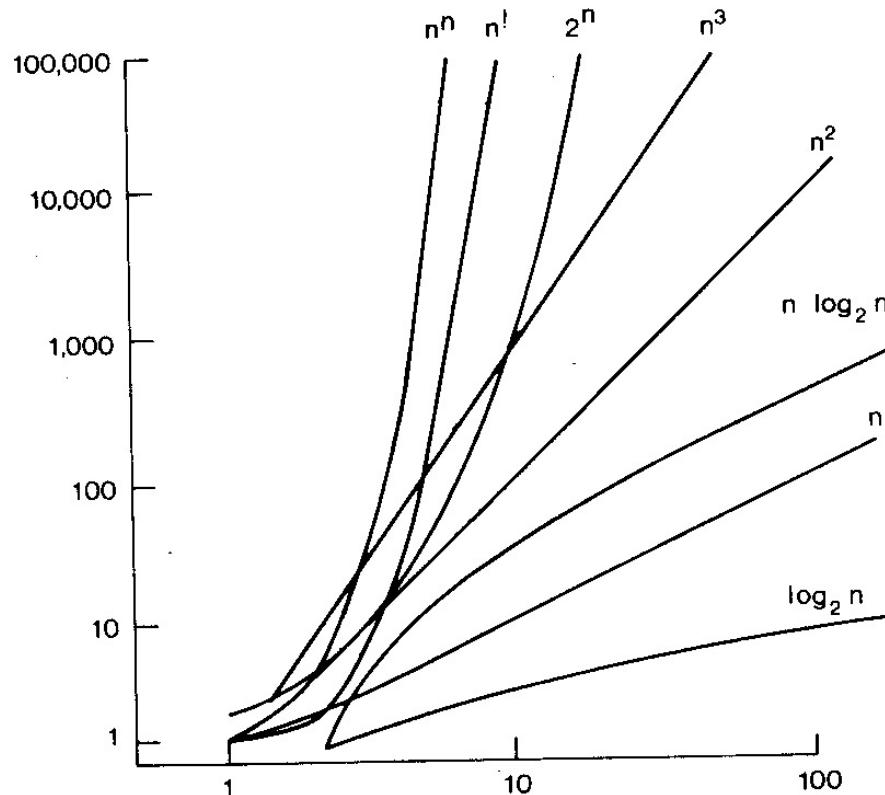
Comparação

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	$4,6 t_3$	$10 t_3$
2^n	t_4	$t_4 + 6,6$	$t_4 + 10$

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.



Ordem de Crescimento de Funções



EXERCÍCIOS

7. Calcule a soma dos 20 primeiros termos da P.A. (5, 8, 11, 14...)

- a) 680 b) 670 c) 640 d) 650 e) 620

EXERCÍCIOS (PA)

SOMA DOS N TERMOS

$$S_n = \frac{(a_1 + a_n)n}{2}$$

- ▶ Sum++;
- ▶ }
- ▶ }

$$\cancel{n} \cancel{t_n} \cancel{(a_1 + a_n)} h$$

$$\underline{\underline{1 + 2n + n + 2n^2 + n^2}}$$

$$\underline{\underline{O(n^2)}}$$

- ▶ sum = 0;
- ▶ for (i = 1; i ≤ n; i++) {
 - ▶ for (j = 1; j ≤ i; j++) {
 - ▶ Sum++;
- ▶ }
- ▶ }

1

$$1 + n + n$$

$$(O(n^2))$$

$$\underline{\underline{1 + 2 + 3 + \dots + n}}$$

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

EXERCÍCIOS

7. Calcule a soma dos 20 primeiros termos da P.A. (5, 8, 11, 14...)
a) 680 b) 670 c) 640 d) 650 e) 620

Exemplos

```
▶ sum= 0;  
▶ for (i = 1; i ≤ n; i + +) {  
    ▶ for (j = 1; j ≤ n; j + +) {  
        ▶ Sum+;  
    ▶ }  
▶ }
```

EXERCÍCIOS (PA)

SOMA DOS N TERMOS

▶ sum= 0;

▶ for (i = 1; i ≤ n; i + +) {

▶ for (j = 1; j ≤ i; j + +) {

▶ Sum+;

▶ }

▶ }

$$S_n = \frac{(a_1 + a_n)n}{2}$$

Arvores

Estrutura de Dados

Prof. Anselmo C. de Paiva

Departamento de Informática – Núcleo de Computação Aplicada NCA-UFMA

Definições

- ▶ Árvore: conjunto finito T de elementos denominados nós ou vértices, tq
 - ▶ existe um nó especial denominado raiz da árvore
 - ▶ Os nós restantes são particionados em $m > 0$ conjuntos disjuntos T_1, \dots, T_m , cada um sendo uma árvore
 - ▶ T_1, \dots, T_m são denominados subárvores da raiz.
- ▶ Um conjunto de árvores é denominado floresta.



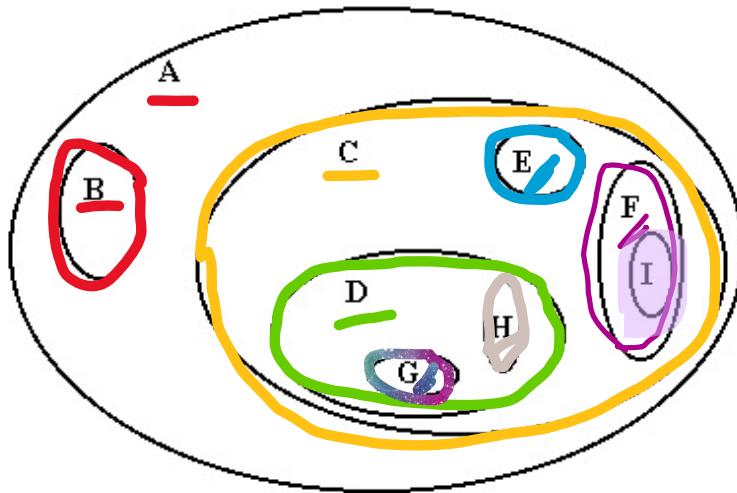
Motivação

- ▶ Diversas aplicações necessitam de estruturas mais complexas que as listas
- ▶ Inúmeros problemas podem ser modelados através de árvores
- ▶ Árvores admitem tratamento computacional eficiente quando comparadas às estruturas mais genéricas como os grafos

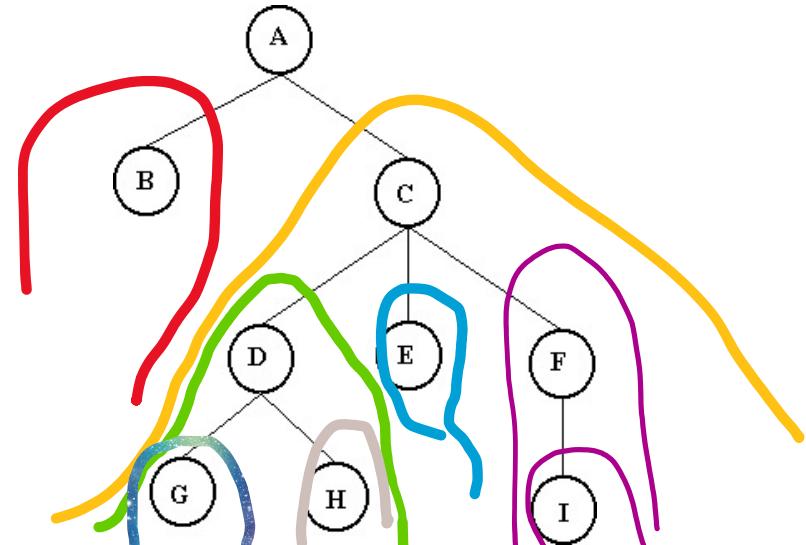
Representações Gráficas

- ▶ Representação por parênteses aninhados
- ▶ 

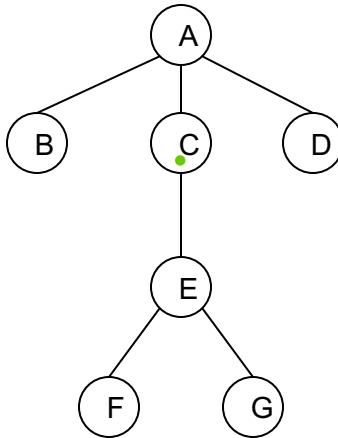
Diagrama de inclusão



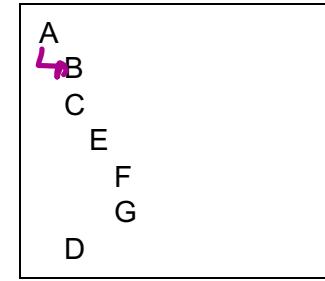
Repr. hierárquica



Árvores



Representação Gráfica



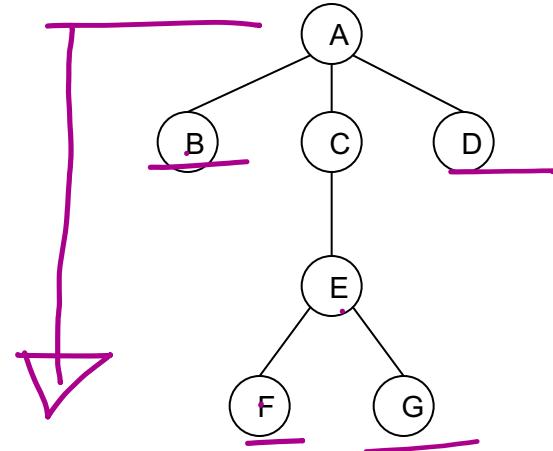
Representação Indentada

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

Representação com Parênteses

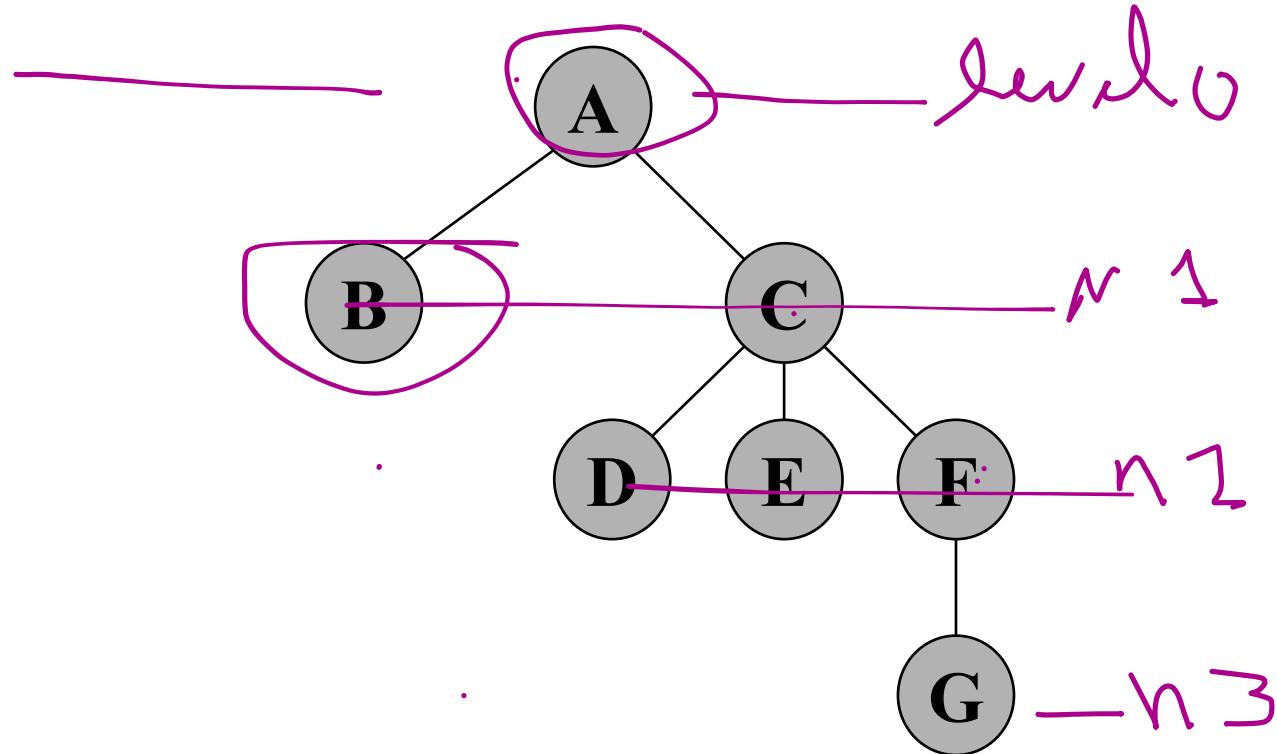
Terminologia

- ▶ Grau de um nó: número de subárvore do nó
- ▶ Folha: nó de grau zero.
- ▶ Profundidade de um nó:
 - ▶ A profundidade da raiz da árvore T é 0
 - ▶ A profundidade de outro nó n em T é 1 mais a altura de n na subárvore de raiz(T) que contém n.
- ▶ Altura de uma árvore:
 - ▶ maior profundidade de um nó na árvore



Exemplo

- ▶ A possui grau 2,
- ▶ C - 3, F - 1
- ▶ B, D, E, e G são folhas
- ▶ A altura da árvore é 3

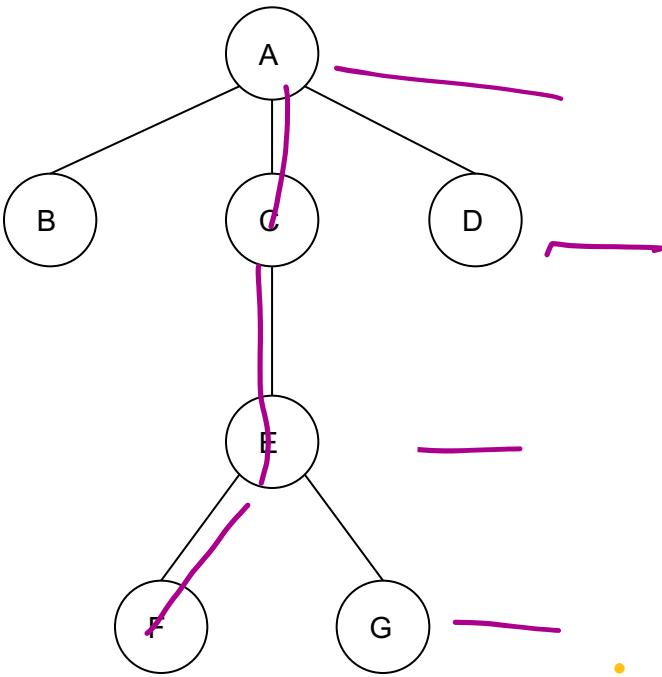


Mais Terminologia

- ▶ Seja n um nó e n_1, \dots, n_m as raízes de suas subárvores.
- ▶ n é o pai de n_1, \dots, n_m , e n_1, \dots, n_m são filhos de n .
- ▶ n_1, \dots, n_m são irmãos.
- ▶ n é o ancestral de todos os nós nas subárvores de n .
- ▶ Os nós nas subárvores de n são descendentes de n .



Árvores – Nomenclatura



- ▶ “A” é o pai de “B”, “C” e “D”
- ▶ “B”, “C” e “D” são filhos de “A”
- ▶ “B”, “C” e “D” são irmãos
- ▶ “A” é um ancestral de “G”
- ▶ “G” é um descendente de “A”
- ▶ “B”, “D”, “F” e “G” são nós folhas
- ▶ “A”, “C” e “E” são nós internos
- ▶ O grau do nó “A” é 3
- ▶ Comprimento do caminho de “C” a “G” é 2
- ▶ Nível de “A” é 0 e o de “G” é 3
- ▶ A altura da árvore é 3

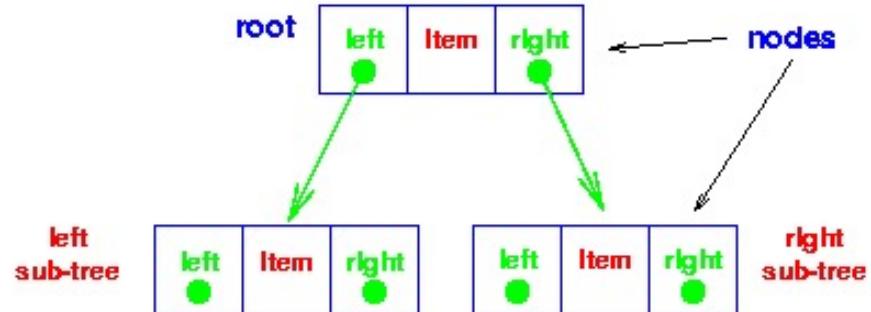
Arvores Binária

- ▶ Grau da arvore
 - ▶ Grau do nó de mais alto grau
- ▶ Grau de um nó: 0, 1, 2, 3,, n
- ▶ Arvore terciaria
- ▶ Arviore quaternária
- ▶ Arvores Binárias
 - ▶ Todo nó tem grau menor ou igual a 2
 - ▶ 0 1. 2

Implementação

```
typedef struct _tnode_ {  
    void *data;  
    struct t_node *left;  
    struct t_node *right;  
}TNode;
```

```
Typedef struct _tree_ Tree ;
```



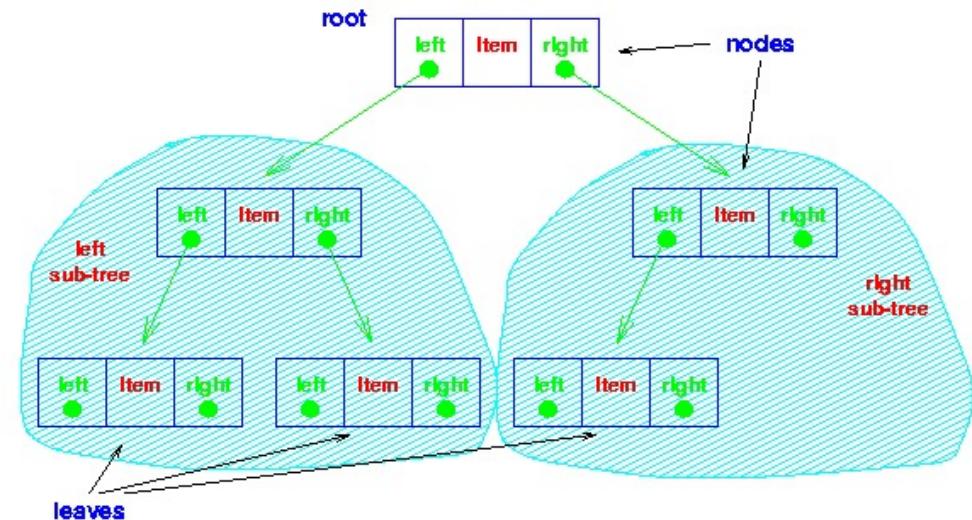
Árvores Binárias – Definição Recursiva

- São compostas de Nodes
- Duas subárvores (Left e Right)
- Ambas as subárvores são árvores binárias

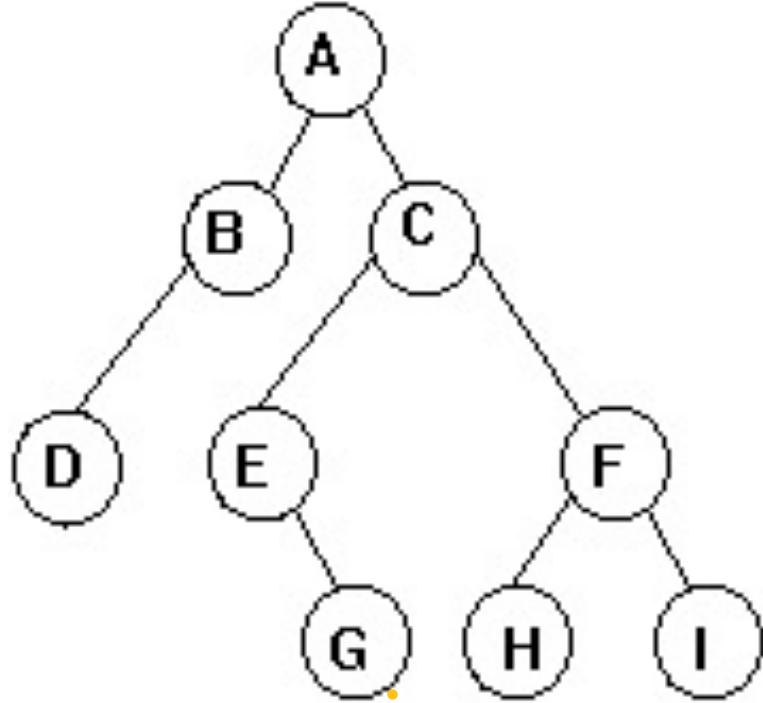
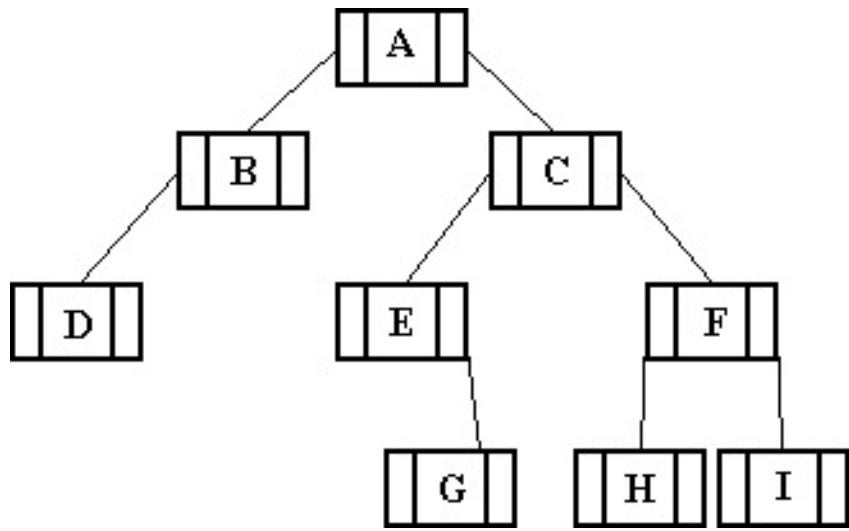
Cada subárvore é uma árvore binária

```
typedef struct _tnode_ {  
    void *item;  
    struct t_node *left;  
    struct t_node *right;  
}TNode;
```

Grau <= 2



Árvores Binárias – Exemplo



.

Altura de Árvores Binárias

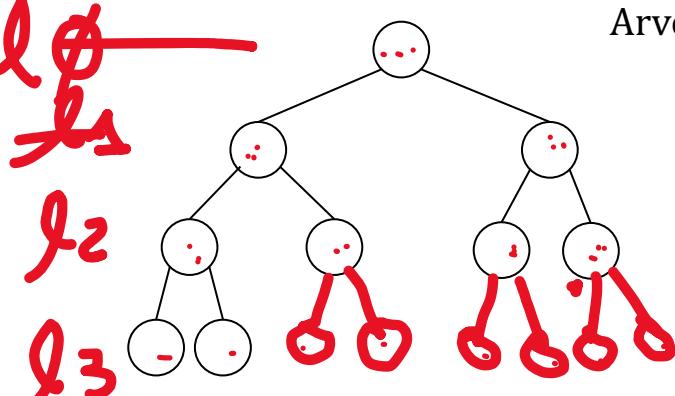
h

$$n = 2^h + 2^{h-1}$$

$$n = 2 \cdot 2^h - 1$$

- ▶ O processo de busca em árvores é normalmente feito a partir da raiz na direção de alguma de suas folhas
- ▶ Naturalmente, são de especial interesse as árvores com a menor altura possível
- ▶ Se uma árvore T com $n > 0$ nós é completa, então ela tem altura mínima. Para ver isso observe que mesmo que uma árvore mínima não seja completa é possível torná-la completa movendo folhas para níveis mais altos

$$\boxed{n = 2^{h+1} - 1}$$



Árvore binária completa

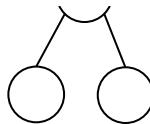
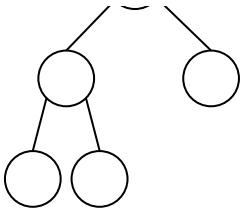
	n	antes de ℓ
0	$1 = 2^0$	0 $2^0 - 1$
1	$2 = 2^1$	1 $2^1 - 1$
2	$4 = 2^2$	3 $2^2 - 1$
3	$8 = 2^3$	7 $2^3 - 1$
4	$16 = 2^4$	15 $2^4 - 1$

$$\log_a(b \cdot c) = \log_a b + \log_a c$$

$$\log_a\left(\frac{b}{c}\right) = \log_a b - \log_a c$$

Arvore binaria completa de n nos

altura?



$$n = 2^{h+1} - 1$$

$$N = 2^{h+1} + 1$$

$$N-1 = 2 \cdot 2^h$$

$$(n-1)/2 = 2^h$$

$$H = \log((n-1)/2)$$

$$H = \log(n-1) - \log 2$$

$$H = \log(n-1) - 1$$

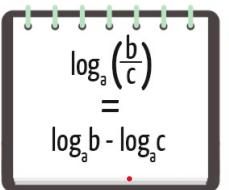
$$n+1 = 2 \cdot 2^h$$

$$2^h = \frac{n+1}{2}$$

$$h = \log_2 \frac{n+1}{2}$$

$$h = \log(n+1) - \log_2 2$$

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



$$\log_a \left(\frac{b}{c} \right) = \log_a b - \log_a c$$

Altura de Árvores Binárias

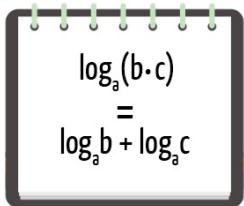
- ▶ A altura mínima de uma árvore binária com $n > 0$ nós é

$$h = 1 + \lfloor \log_2 n \rfloor$$
- ▶ Prova-se por indução. Seja T uma árvore completa de altura h
 - ▶ Vale para o caso base ($n=1$)
 - ▶ Seja T' uma árvore cheia obtida a partir de T pela remoção de k folhas do último nível
 - ▶ Então T' tem $n' = n - k$ nós
 - ▶ Como T' é uma árvore cheia,

$$n' = 1 + 2 + \dots + 2^{h-2} = 2h-1 - 1$$
 e

$$h = 1 + \log_2(n'+1)$$
 - ▶ Sabemos que $1 \leq k \leq n'+1$ e portanto

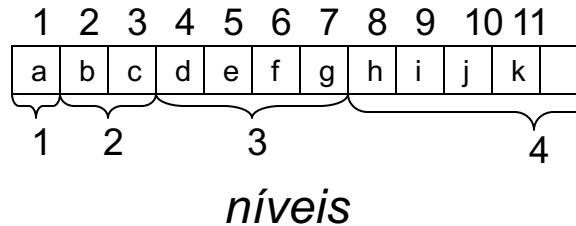
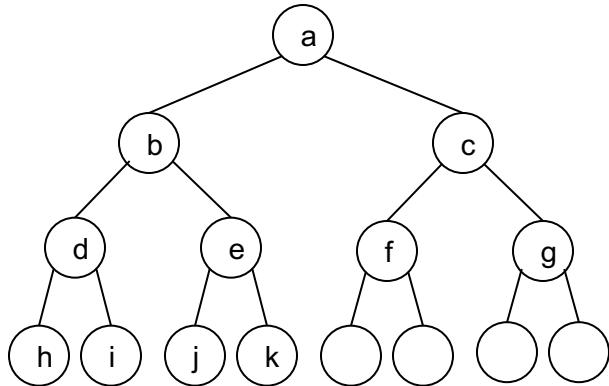
$$\log_2(n'+1) = \lfloor \log_2(n'+k) \rfloor = \lfloor \log_2 n \rfloor$$



$$\log_a(b \cdot c) = \log_a b + \log_a c$$

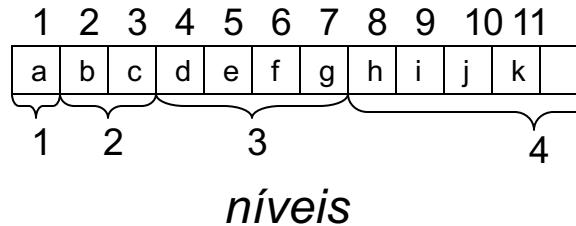
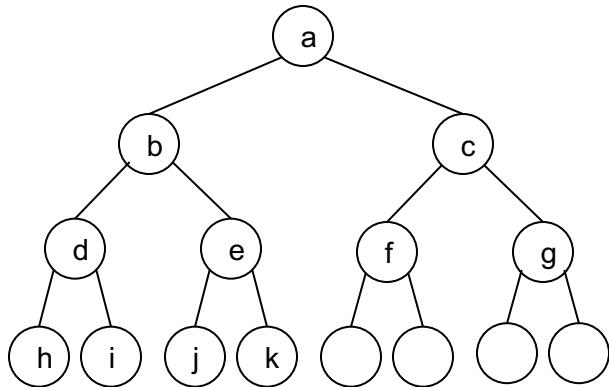
Implementando Árvores Binárias com Arrays

- ▶ Assim como listas, árvores binárias podem ser implementadas utilizando-se o armazenamento contíguo proporcionado por arrays
- ▶ A idéia é armazenar níveis sucessivos da árvore seqüencialmente no array



Implementando Árvores Binárias com Arrays

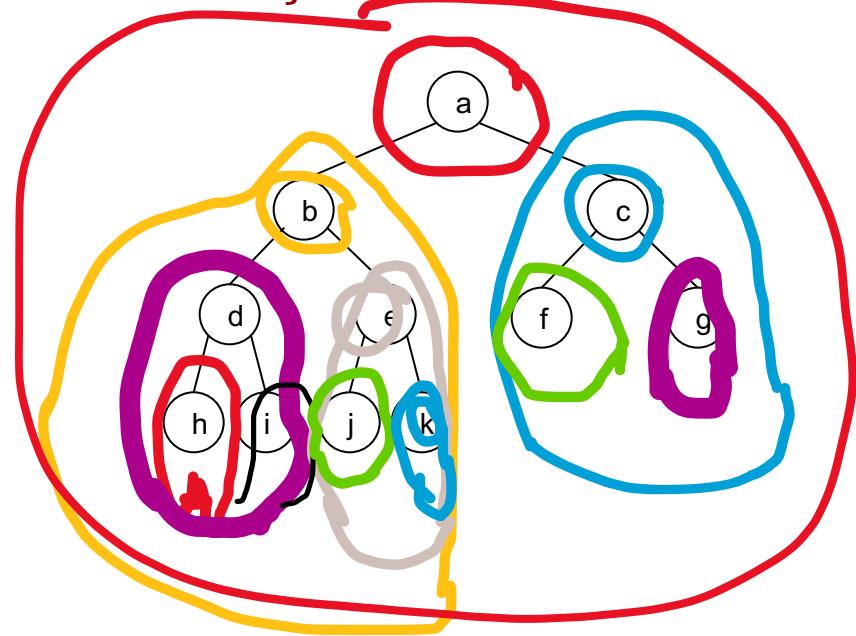
- ▶ Assim como listas, árvores binárias podem ser implementadas utilizando-se o armazenamento contíguo proporcionado por arrays
- ▶ A idéia é armazenar níveis sucessivos da árvore seqüencialmente no array



Implementando Árvores Binárias com Arrays

- ▶ Dado um nó armazenado no índice i , é possível computar o índice
 - ▶ do nó filho esquerdo de $i : 2i$
 - ▶ do nó filho direito de $i : 2i + 1$
 - ▶ do nó pai de $i : i \text{ div } 2$
- ▶ Para armazenar uma árvore de altura h precisamos de um array de $2^h - 1$ (número de nós de uma árvore cheia de altura h)
- ▶ Nós correspondentes a subárvores vazias precisam ser marcados com um valor especial diferente de qualquer valor armazenado na árvore .
- ▶ A cada índice computado é preciso se certificar que está dentro do intervalo permitido
 - ▶ Ex.: O nó raiz é armazenado no índice 1 e o índice computado para o seu pai é 0 .

Visitação ou caminhamento



Pre-ordem

visita a raiz

visita a subarvore esquerda

visita a subarvore direita

Simetrica

visita a subarvore esquerda

visita a raiz

visita a subarvore direita

Pos-ordem

visita a subarvore esquerda

visita a subarvore direita

visita a raiz

A B d h i e j k f g Pre-ordem

Visitação ou caminhamento



Pre-ordem
visita a raiz
visita a subárvore esquerda
visita a subárvore direita

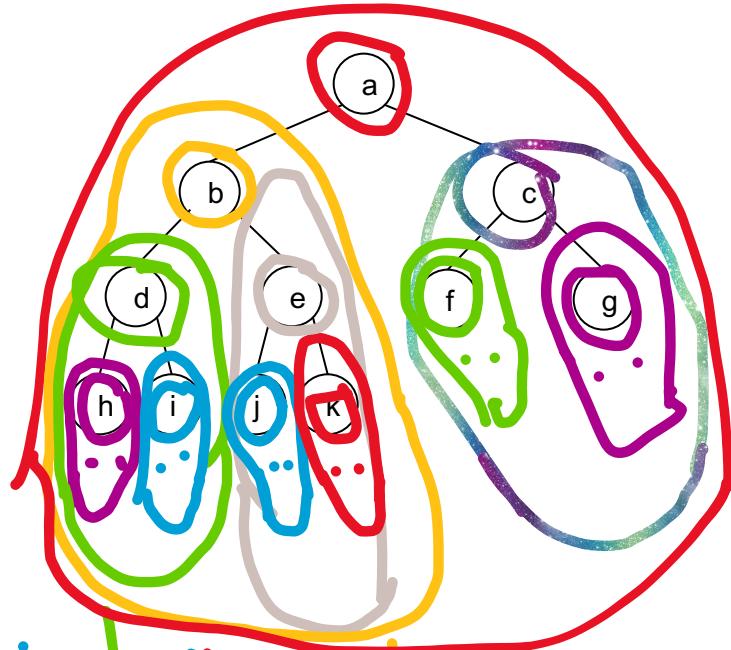
Simétrica

visita a subárvore esquerda
visita a raiz
visita a subárvore direita

Pos-ordem

visita a subárvore esquerda
visita a subárvore direita
visita a raiz

Visitação ou caminhamento



hidjKEbfgca p DS ORRgm

Pre-ordem

visita a raiz

visita a subarvore esquerda

visita a subarvore direita

Simetrica

visita a subarvore esquerda

visita a raiz

visita a subarvore direita

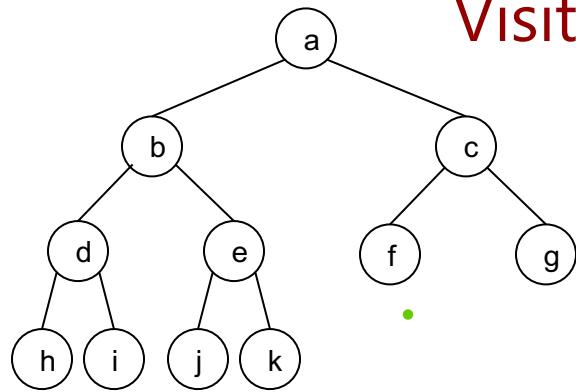
Pos-ordem

visita a subarvore esquerda

visita a subarvore direita

visita a raiz

Visitação ou caminhamento



Pre-ordem

visita a raiz

visita a subárvore esquerda

visita a subárvore direita

Simétrica

visita a subárvore esquerda

visita a raiz

visita a subárvore direita

Pos-ordem

visita a subárvore esquerda

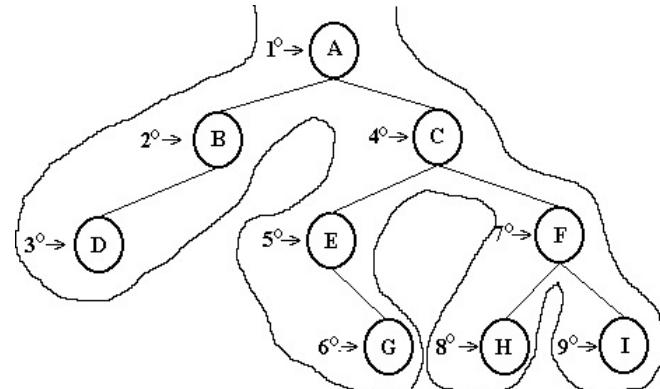
visita a subárvore direita

visita a raiz

A B d h i e j k e f g Pre-ordem
h d i b j e k a f c g simétrica
~~i d j k e b f g c a~~ pos ORDEM

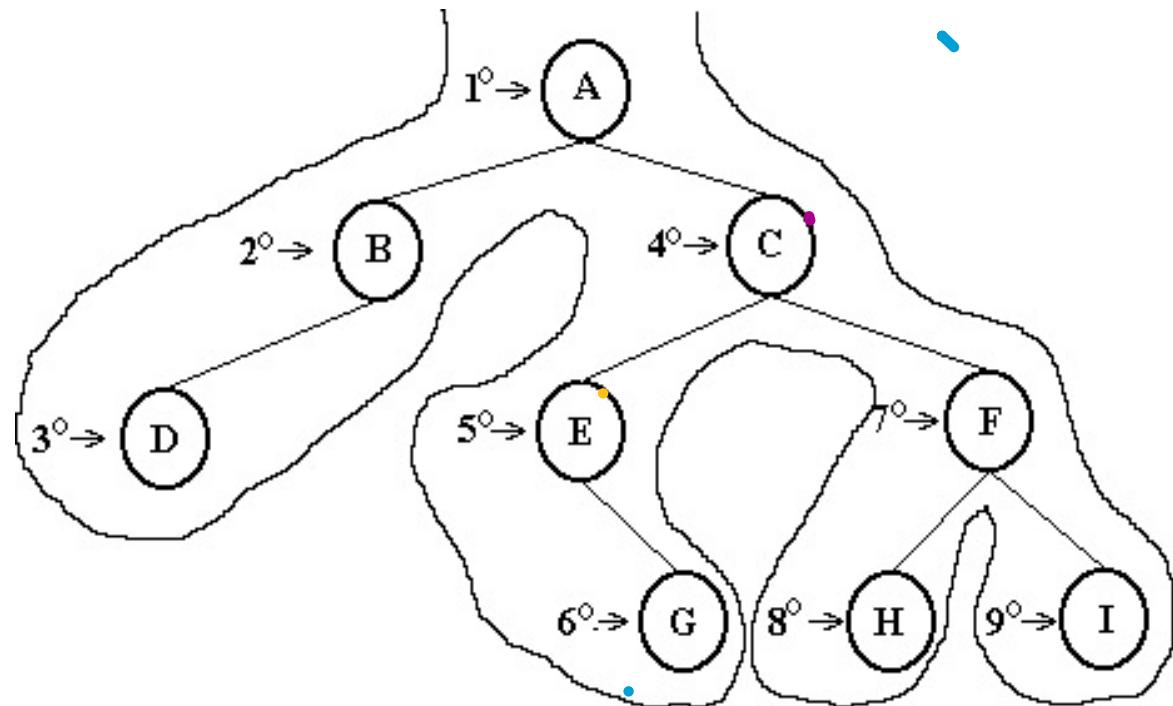
Árvores Binárias – Caminhamento

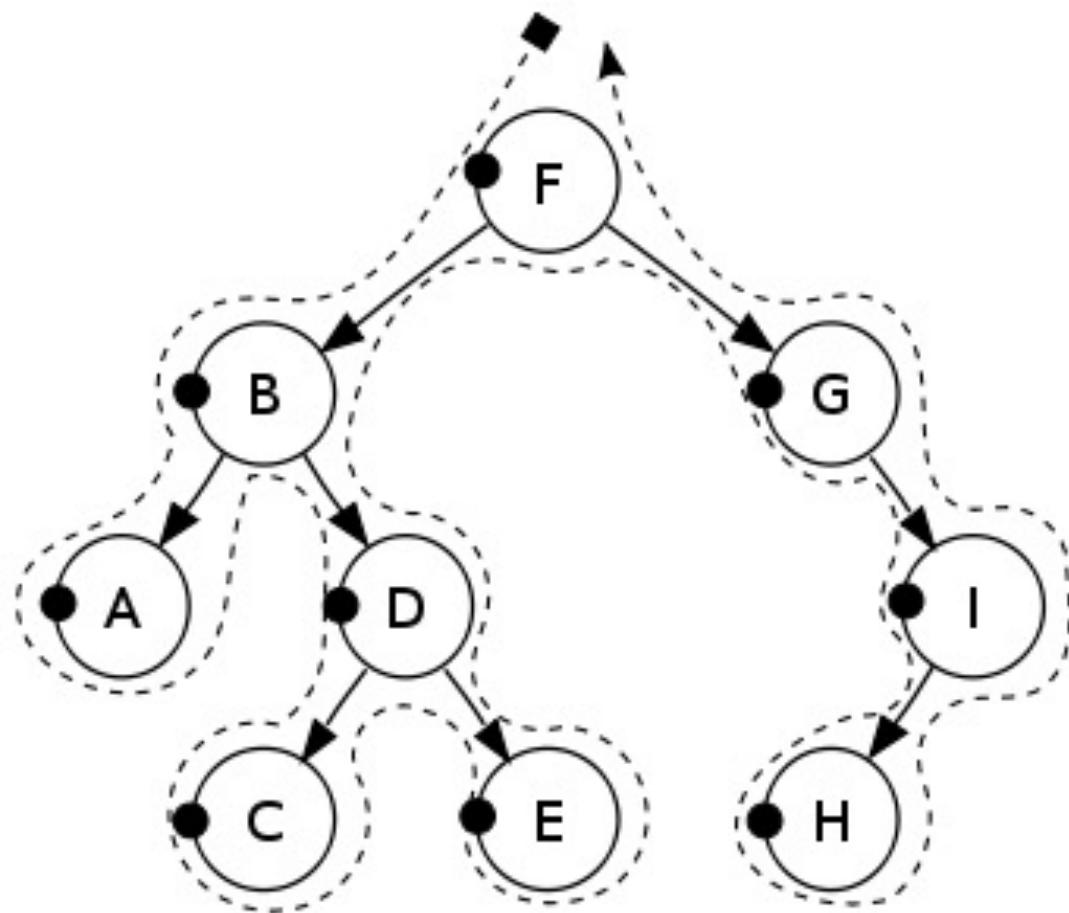
- ▶ Ordem em que se percorre todos os nós de uma árvore
- ▶ Três maneiras principais:
 - ▶ Pré-ordem – raiz, depois caminhamento em préordem das subárvores esquerda e direita
 - ▶ Pós-ordem – caminhamento pós-ordem das subárvores(esq. e dir.) seguida pela raiz
 - ▶ Simétrica – caminhamento simétrico da sub-árvore esquerda, seguida da raiz, e seguida do caminhamento simétrico da subárvore direita



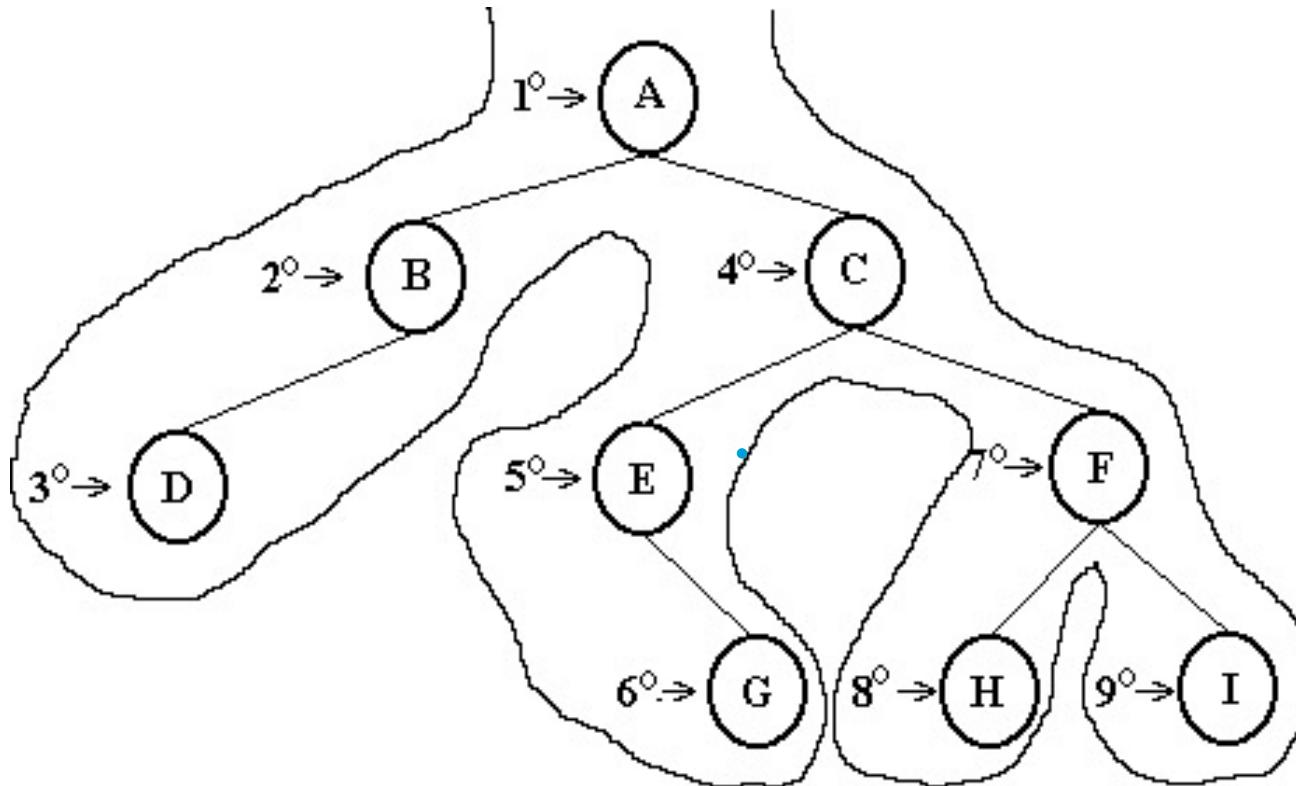
Árvores Binárias – Caminhamento

- ▶ Simétrica – caminhamento simétrico da sub-árvore esquerda, seguida da raiz, e seguida do caminhamento simétrico da subárvore direita

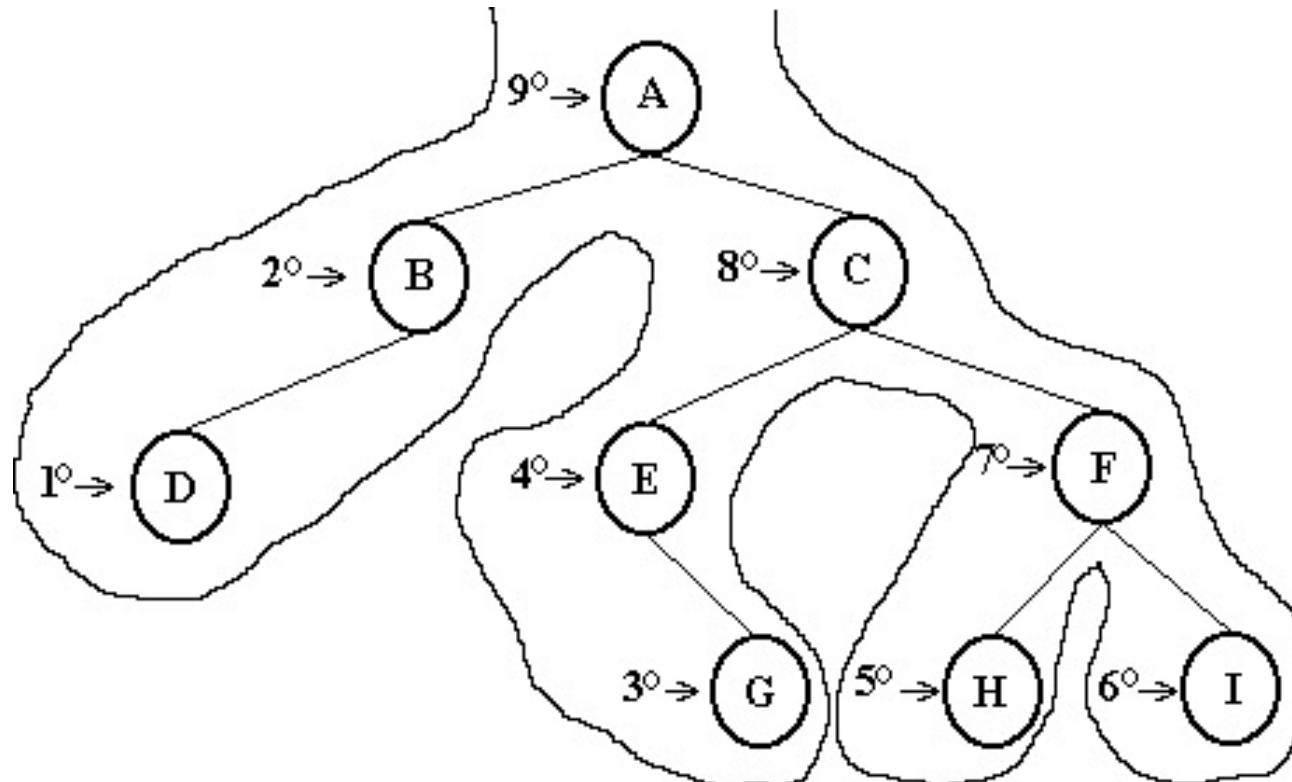




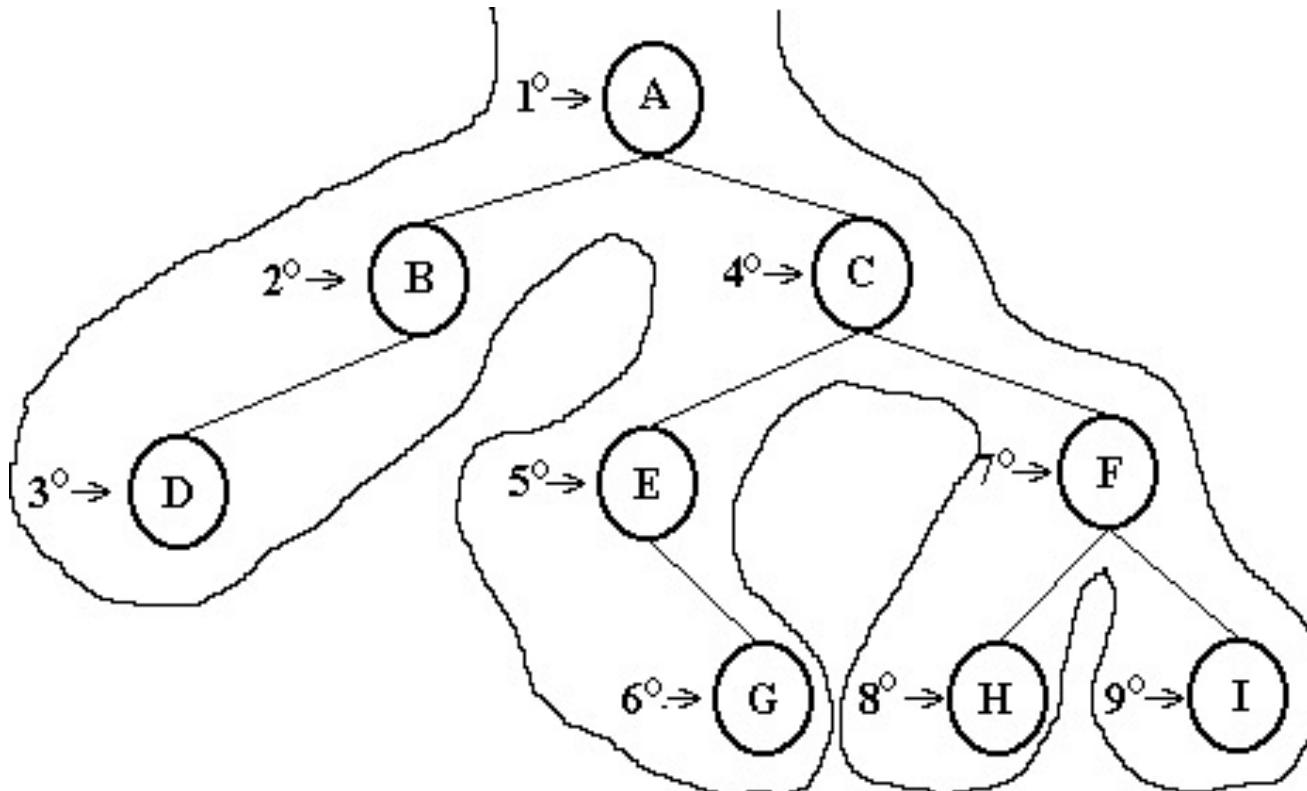
Árvores Binárias – Cam. PréOrdem



Árvores Binárias – Cam.PósOrdem



Árvores Binárias – Cam. Simétrico



Implementação - Caminhamento

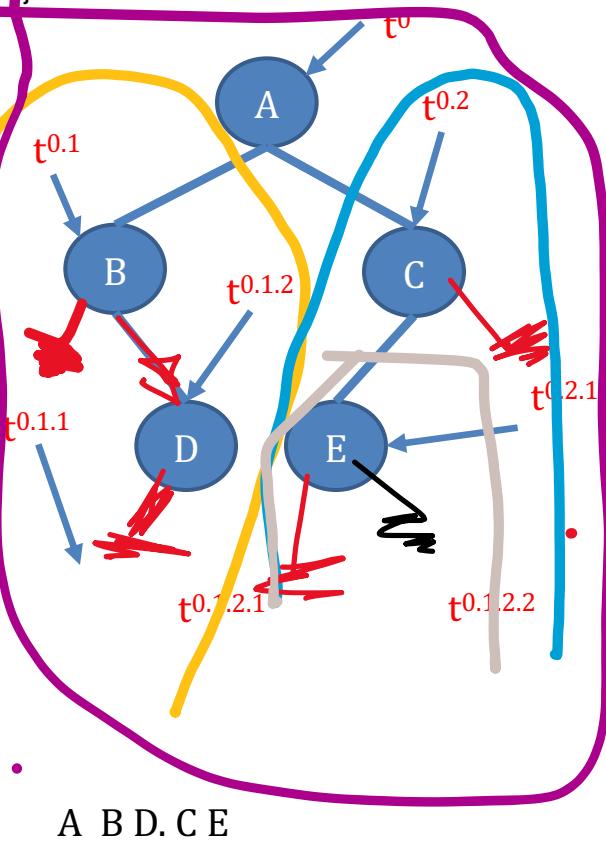
```
typedef struct _tnode_ {
    void *item;
    struct t_node *left;
    struct t_node *right;
}TNode;
```

```
void preOrdem ( TNode *t, void (*visit) (void *) )
{
    if ( t != NULL ) {
        visit(t->item);
        preOrdem ( t->left, visit);
        preOrdem ( t->right, visit);
    }
}
```

```

void preOrdem (t, visit){
    if ( t != NULL ) {
        visit(t->item);
        preOrdem ( t->left, visit);
        preOrdem ( t->right, visit);
    }
}

```



```

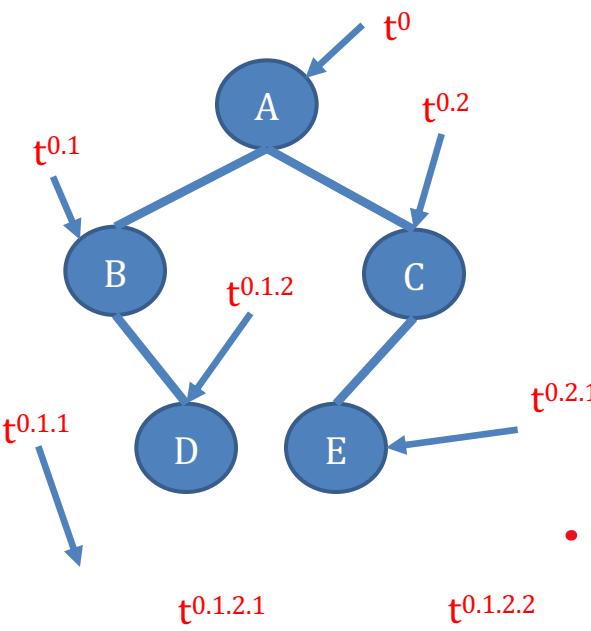
void preOrdem (t (&A), visit){
    if ( t^0 != NULL ) {
        visit (t^0->item);
        preOrdem ( t^0->left (t^0.1(&B), visit);
        if (t^0.1 != NULL ) {
            visit (t^0.1->item);
            preOrdem ( t^0.1->left (t^0.1.1(NULL), visit);
            if(t^0.1.1 != NULL ) { }
        }
        preOrdem ( t^0->right (t^0.2(&C), visit);
        if(t^0.2 != NULL ) {
            visit (t^0.2->item);
            preOrdem ( t^0.2->left (t^0.2.1(&E), visit);
            if (t^0.2.1!= NULL ) {
                visit (t^0.2.1->item);
                preOrdem (t^0.2.1->left, visit);
                preOrdem (t^0.2.1->right, visit)
            }
        }
    }
}

```

```

void preOrdem (t, visit){
    if ( t != NULL ) {
        visit(t->item);
        preOrdem ( t->left, visit);
        preOrdem ( t->right, visit);
    }
}

```



A B D C E

```

void preOrdem (t^0(&A), visit){
    if ( t^0 != NULL ) {
        visit(t^0->item);
        preOrdem ( t^0->left, visit) (t^0.1(&B), visit){
            if ( t^0.1 != NULL ) {
                visit(t^0.1->item);
                preOrdem ( t^0.1->left, visit) (t^0.1.1(NULL), visit) {
                    if ( t^0.1.1 != NULL ) {
                        visit(t->item);
                        preOrdem ( t->left, visit);
                        preOrdem ( t->right, visit);
                    }
                }
                preOrdem ( t^0.1->right, visit) (t^0.1.2(&D), visit) {
                    if ( t^0.1.2 != NULL ) {
                        visit(t^0.1.2->item);
                        preOrdem ( t^0.1.2->left, visit); (t^0.1.2.1(&NULL), visit)
                        preOrdem ( t^0.1.2->right, visit); (t^0.1.2.2(&NULL), visit)
                    }
                }
            }
        }
        preOrdem ( t^0->right, visit); (t^0.2(&C), visit){
            if ( t^0.2 != NULL ) {
                visit(t^0.2->item);
                preOrdem ( t^0.2->left, visit) (t^0.2.1(&E), visit) {
                    if ( t^0.2.1 != NULL ) {
                        visit(t^0.2.1->item);
                        preOrdem ( t^0.2.1->left, visit) (t^0.2.1.1(NULL), visit)
                        preOrdem ( t^0.2.1->right, visit) (t^0.2.1.2(NULL), visit)
                    }
                }
                preOrdem ( t^0.2->right, visit) (t^0.2.2(NULL), visit)
            }
        }
    }
}

```

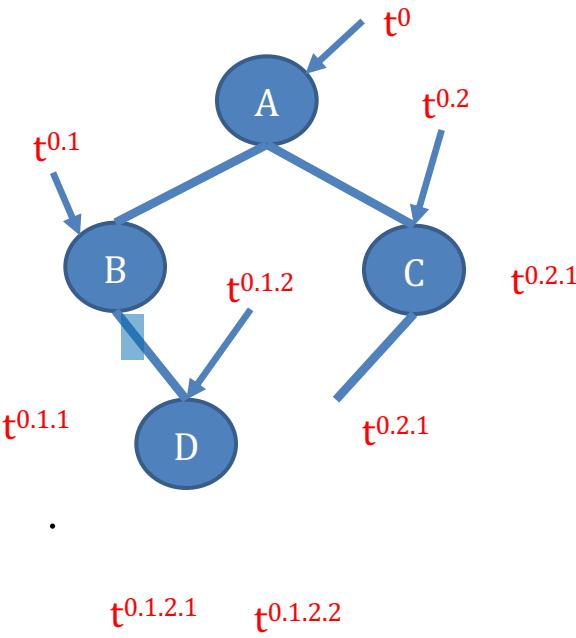
Implementação - Caminhamento

```
void posOrdem ( TNode *t, void (*visit) (void *) )
{
    if ( t != NULL ) {
        posOrdem ( t->left, visit);
        posOrdem ( t->right, visit);
        visit(t->item);
    }
}
```

```

void Simetrico (t,visit) {
    if ( t != NULL ) {
        Simetrico( t->left, visit);
        visit(t->item);
        Simetrico ( t->right, visit);
    }
}

```



```

void Simetrico (t0,visit) t0(&A){
    if ( t0 != NULL ) {
        Simetrico( t0->left, visit) (t0.1(&B) {
            if ( t0.1 != NULL ) {
                Simetrico( t0.1->left, visit); (t0.1.1(NULL) {
                    if ( t0.1.1 != NULL ) {
                        Simetrico( t0.1.1->left, visit);
                        visit(t0.1.1->item);
                        Simetrico ( t0.1.1->right, visit);
                    }
                }
                visit(t0.1->item);
                Simetrico ( t0.1->right, visit); t0.1.2(&D)
                if ( t0.1.2 != NULL ) {
                    Simetrico( t0.1.2->left, visit);
                    visit(t0.1.2->item);
                    Simetrico ( t0.1.2->right, visit);
                }
            }
        }
        visit(t0->item);
        Simetrico ( t0->right, visit); t0.2 (&C) {
            if ( t0.2 != NULL ) {
                Simetrico( t0.2->left, visit);
                visit(t0.2->item);
                Simetrico ( t0.2->right, visit);
            }
        }
    }
}

```

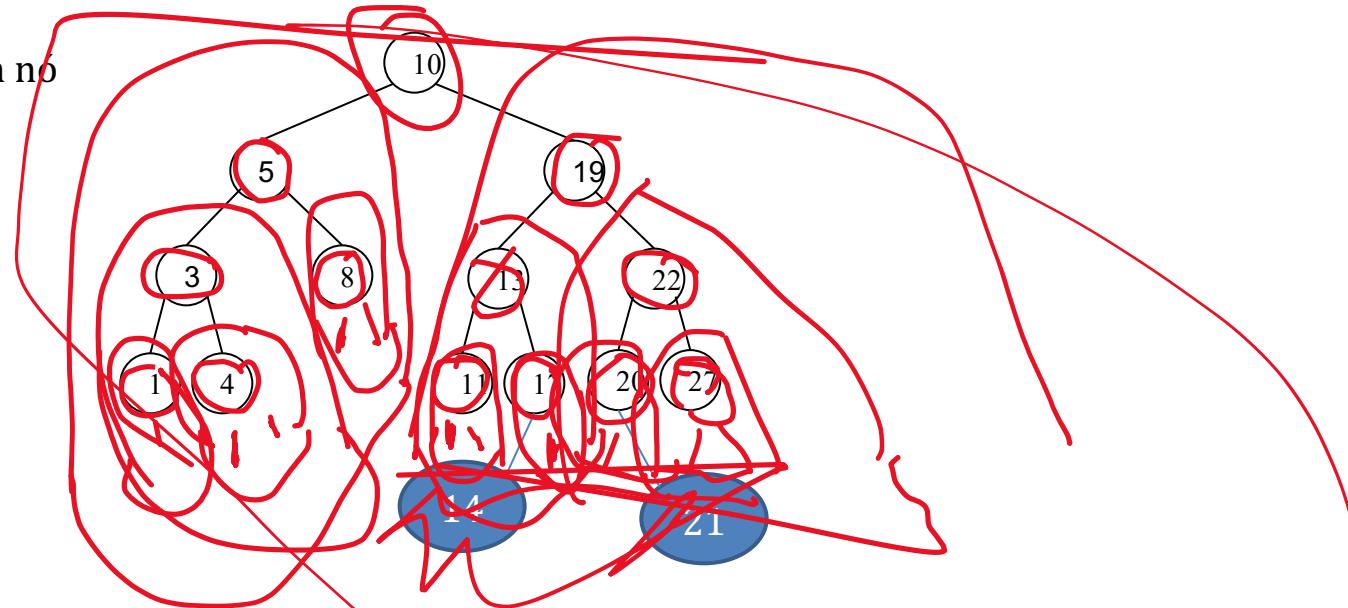
B D A C

Implementação - Caminhamento

```
void Simetrico ( TNode *t, void (*visit) (void *) )
{
    if ( t != NULL ) {
        Simetrico( t->left, visit);
        visit(t->item);
        Simetrico ( t->right, visit);
    }
}
```

Arvore Binária de Pesquisa (Busca)

- Muito rápido para consultar um nó
- $O(\log n)$



13 4 5 8 10 11 13 17 19 20 22 27

~~Lista de Exercício~~

fazer para a arvore que esta neste slide a visitacao com execução expandida do algoritmo PosOrdem

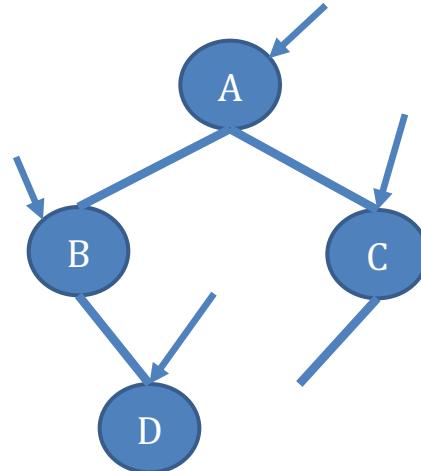
Implementar um TAD Arvore Binaria de Pesquisa com quatro funções:

- Criar uma arvore vazia
- Inserir num dado numa arvore
- Remover um dado de uma arvore
- Listar em visitação simétrica.

Refazer o programa que vc já fez duas vezes com as seguintes funções:

- Inserir dados
- Remover dado
- Consultar dado

Listar em visitação simétrica

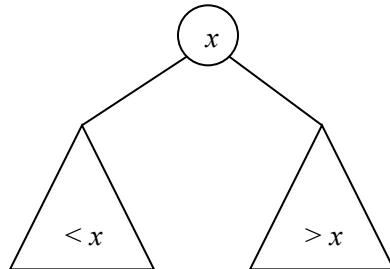


Dicionários

- ▶ A operação de busca é fundamental em diversos contextos da computação
- ▶ Por exemplo, um dicionário é uma estrutura de dados que reúne uma coleção de chaves sobre a qual são definidas as seguintes operações :
 - ▶ Inserir (x, T) : inserir chave x no dicionário T
 - ▶ Remover (x, T) : remover chave x do dicionário T
 - ▶ Buscar (x, T) : verdadeiro apenas se x pertence a T
- ▶ Outras operações são comuns em alguns casos:
 - ▶ Encontrar chave pertencente a T que sucede ou precede x
 - ▶ Listar todas as chaves entre x_1 e x_2

Árvores Binárias de Busca

- ▶ Uma maneira simples e popular de implementar dicionários é uma estrutura de dados conhecida como árvore binária de busca
- ▶ Numa árvore binária de busca, todos os nós na subárvore à esquerda de um nó contendo uma chave x são menores que x e todos os nós da subárvore à direita são maiores que x

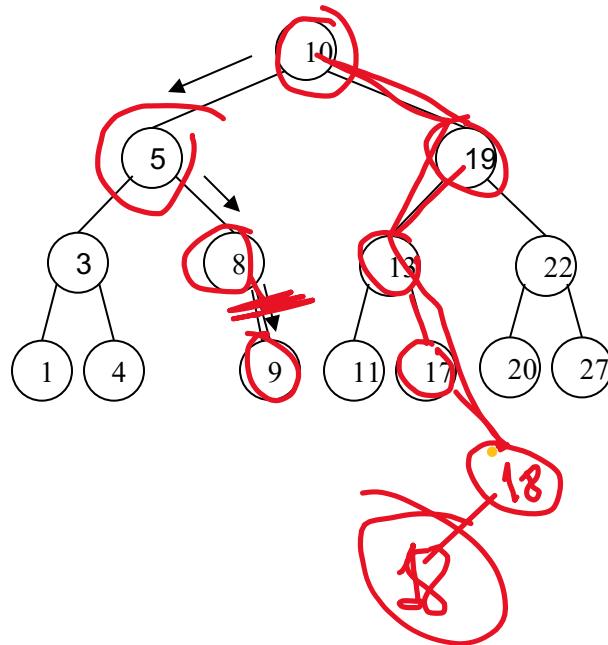


Busca e Inserção em Árvores Binárias de Busca

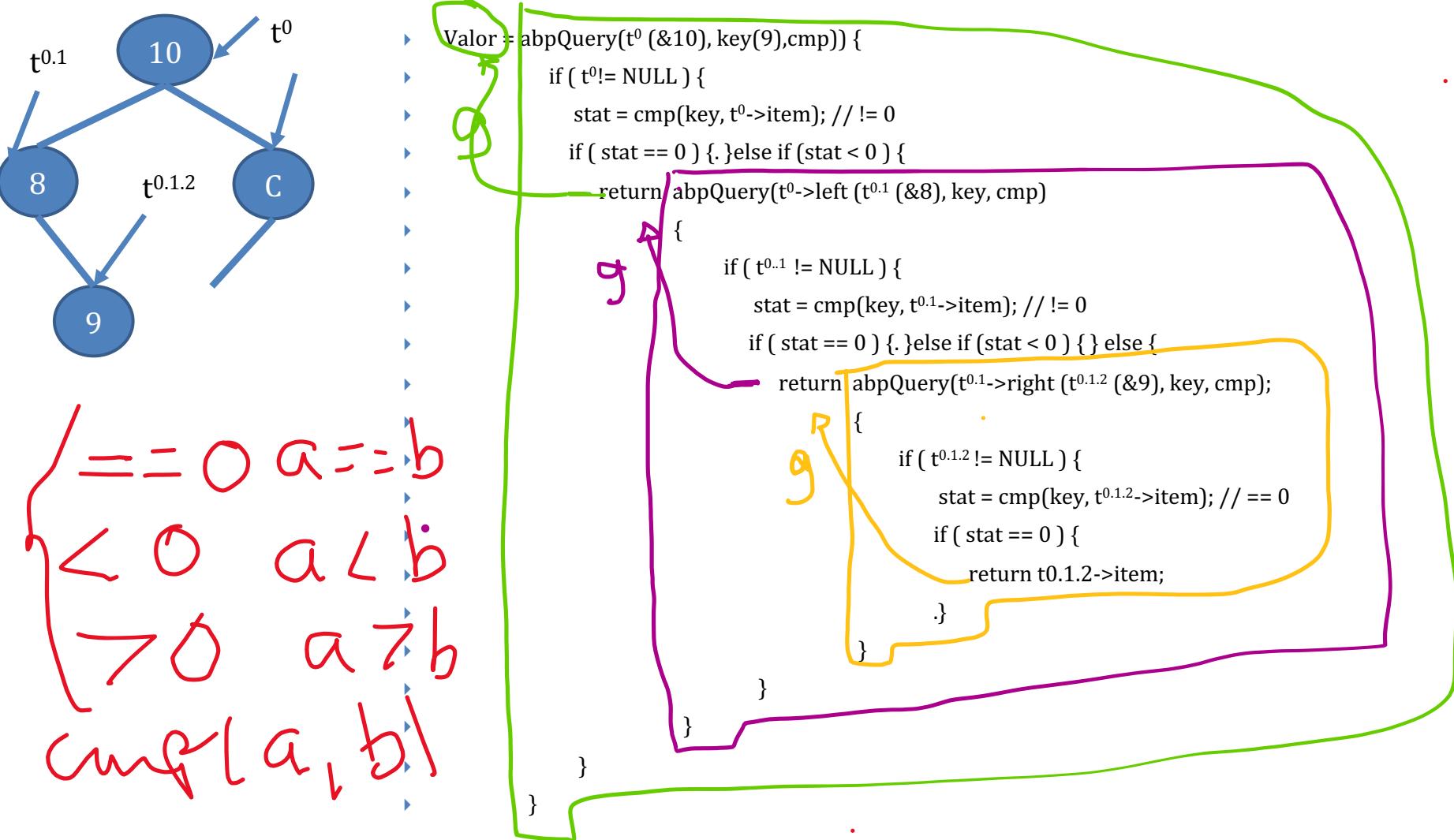
```
proc Buscar (Chave x, Árvore T) {  
    se T = Nulo então retornar falso  
    se x = T->Val então retornar verdadeiro  
    se x < T->Val então retornar Buscar (x, T->Esq)  
    retornar Buscar (x, T->Dir)  
}  
proc Inserir (Chave x, var Árvore T) {  
    se T = Nulo então {  
        T = Alocar (NoArvore)  
        T->Val, T->Esq, T->Dir = x, Nulo, Nulo  
    }  
    senão {  
        se x < T->Val então Inserir (x, T->Esq)  
        se x > T->Val então Inserir (x, T->Dir)  
    }  
}
```

Inserção em Árvores Binárias de Busca

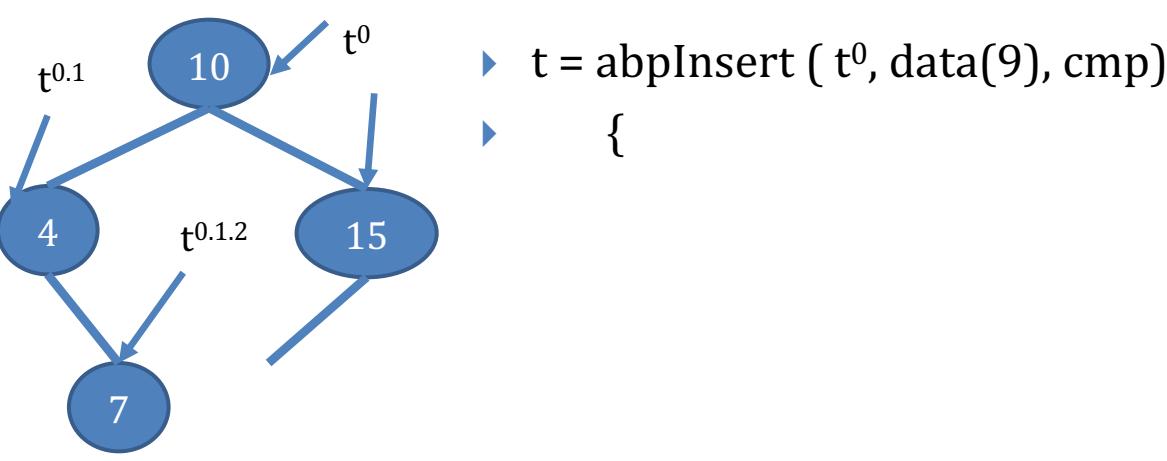
Inserir (9, T)



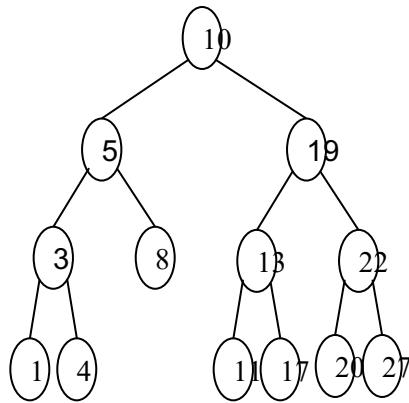
```
▶ void *abpQuery( TNode*t, void *key, int (*cmp)(void *, void *))  
▶     int stat;  
▶         if ( t != NULL ) {  
▶             stat = cmp(key, t->item);  
▶             if ( stat == 0 ) {  
▶                 return t->data;  
▶             }else if (stat < 0 ) {  
▶                 return abpQuery(t->left, key, cmp);  
▶             } else {  
▶                 return abpQuery(t->right, key, cmp);  
▶             }  
▶         }  
▶         return NULL;  
▶ }
```



```
▶ TNode *abpInsert( Tnode *t, void *data, int (*cmp)(void *, void *))  
▶ {.    Tnode *newnode;  int stat;  
▶     if ( t != NULL ) {  
▶         stat = cmp(data, t->data);  
▶         if ( stat <= 0 ) {  
▶             t->left = abpInsert(t, data, cmp);  
▶         }else {  
▶             t->right = abpInsert(t, data, cmp);  
▶         }  
▶         return t;  
▶     } else {  
▶         newnode = (Tnode *) malloc (sizeof(TNode));  
▶         if (newnode != NULL ) {  
▶             newnode->data = data;  
▶             newnode->left = newnode->right = NULL;  
▶             return newnode;  
▶         } else {  
▶             return NULL;  
▶         }  
▶     }  
▶ }
```



- ▶ $t = \text{abpInsert} (t^0, \text{data}(9), \text{cmp})$
- ▶ {

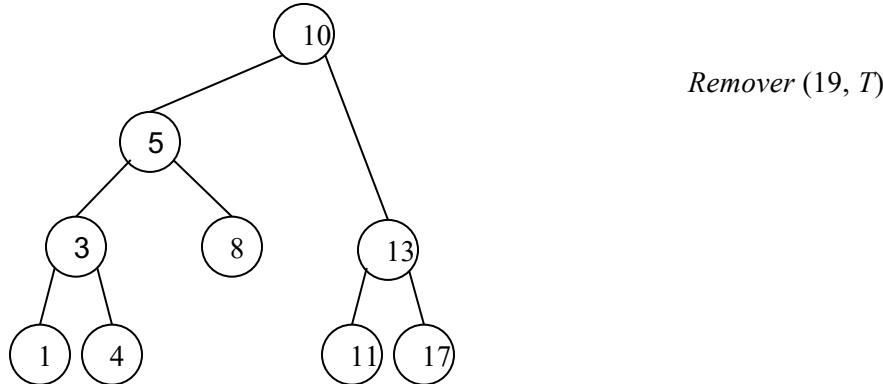


```

    t = abpInsert(t, data, cmp) { •
    if ( t != NULL ) {
        stat = cmp(data, t->data);
        if ( stat <= 0 ) {
            t->left = abpInsert(t->left , data, cmp) {
                if ( t != NULL ) {
                    stat = cmp(data, t->data);
                    if ( stat <= 0 ) {
                        }else {
                            t->right = abpInsert(t->right, data, cmp); {
                                if ( t != NULL ) {
                                    stat = cmp(data, t->data);
                                    if ( stat <= 0 ) {
                                        t->left = abpInsert(t->left , data, cmp) {
                                            if ( t != NULL ) {
                                                stat = cmp(data, t->data);
                                                if ( stat <= 0 ) {
                                                    t->right = abpInsert(t->right, data, cmp); {
                                                        if ( t != NULL ) {
                                                            newnode = (Tnode *) malloc (sizeof(TNode));
                                                            if (newnode != NULL ) {
                                                                newnode->data = data;
                                                                newnode->left = newnode->right = NULL;
                                                                return newnode;
                                                            } else {
                                                                return NULL;
                                                            }
                                                        }
                                                    }
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
  
```

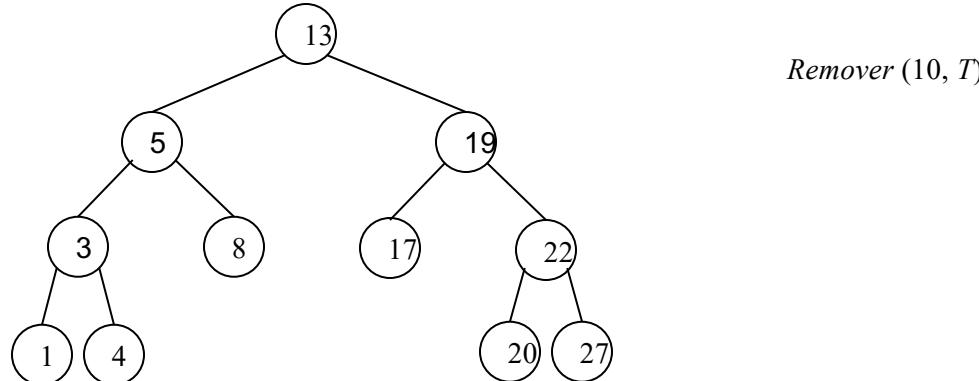
Remoção em Árvores Binárias de Busca

- ▶ Para remover uma chave x de uma árvore T temos que distinguir os seguintes casos
 - ▶ x está numa folha de T : neste caso, a folha pode ser simplesmente removida
 - ▶ x está num nó que tem sua subárvore esquerda ou direita vazia: neste caso o nó é removido substituído pela subárvore não nula



Remoção em Árvores Binárias de Busca

- Se x está num nó em que ambas subárvores são não nulas, é preciso encontrar uma chave y que a possa substituir. Há duas chaves candidatas naturais:
 - A menor das chaves maiores que x ou
 - A maior das chaves menores que x



```

TNode *abpRemove( Tnode *t, void *key, int (*cmp)(void *, void *), void **data)
{
    void *data2;
    if ( t != NULL ) {
        stat = cmp(data, t->data);
        if ( stat < 0 ) {
            t->left = abpRemove(t->left, key, cmp);
            return t;
        } else if ( stat > 0 ) {
            t->right = abpRemove(t->right, key, cmp);
        } else {
            if (t->left == NULL && t->right == NULL) {
                *data = t->data;
                free(t);
                return NULL;
            } else if (t->left == NULL) {
                aux = t->right; *data = t->data;
                free(t);
                return aux;
            } else if ( t->right == NULL ) {
                aux = t->left; *data = t->data;
                free(t);
                return aux;
            } else {
                *data = t->data;
                t->right = abpRemoveMenor(t->right, key, cmp, &data2);
                t->data = data2;
                return t;
            }
        }
    }
}

```

```

TNode *abpRemoveMenor( Tnode *t, void *key, int (*cmp)(void *, void *),
void **data)
{
    void *data2;
    if ( t!= NULL ) {
        if ( t-> left != NULL ) {
            t->left = abpRemoveMenor(t->left, key, cmp, &data2);
        } else {
            if (t->right != NULL ) {
                aux = t->right; *data = t->data;
                free(t);
                return aux;
            } else {
                *data = t->data;
                free(t);
                return NULL;
            }
        }
    }
    *data = NULL;
    return NULL;
}

```

Retorna o menor nó de uma arvore

```
void * abpGetMin ( Tnode *t)
{
    if ( t != NULL ){
        if ( t->left != NULL) {
            return abpGetMin (t->left);
        } else {
            returnm t->data;
        }
    }
    return NULL;
}
```

Retorna o menor nó de uma arvore

```
void * abpGetMax ( Tnode *t)
{
    if ( t != NULL ){
        if ( t->rioght != NULL) {
            return abpGetMin (t->right);
        } else {
            returnm t->data;
        }
    }
    return NULL;
}
```

Remoção em Árvores Binárias de Busca

```
proc RemoverMenor (var Árvore T) {  
    se T->Esq = Nulo então {  
        tmp = T  
        y = T->Val  
        T = T->Dir  
        Liberar (tmp)  
        retornar y  
    }  
    senão  
        retornar RemoverMenor (T->Esq)  
}
```

```
    ▶    TNode *abpRemoveMenor( Tnode *t, void *key, int (*cmp)(void *, void *),  
                                void **data)  
    ▶    {    void *data2;  
    ▶        if ( t->left == NULL ) {  
    ▶            *data2 = t->data;  
    ▶            free (t);  
    ▶            return NULL;  
    ▶        } else {  
    ▶            return abpMenor(t->right, key, cmp, &data2);  
    ▶        }  
    ▶    }
```

Remoção em Árvores Binárias de Busca

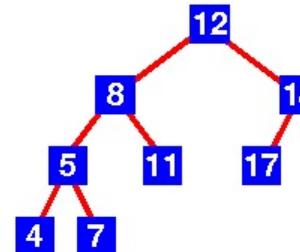
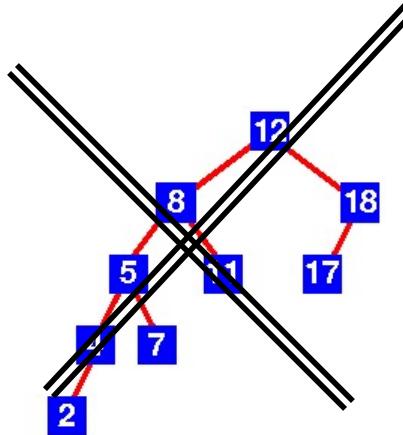
```
proc Remover (Chave x, Árvore T) {  
    se T 1 Nulo então  
        se x < T->val então Remover (x, T->Esq)  
        senão se x > T->val então Remover (x, T->Dir)  
        senão  
            se T->Esq = Nulo então {  
                tmp = T  
                T = T->Dir  
                Liberar (tmp)  
            }  
            senão se T->Dir = Nulo então {  
                tmp = T  
                T = T->Esq  
                Liberar (tmp)  
            }  
            senão T->Val = RemoverMenor (T->Dir)  
    }  
}
```

Árvores Binárias de Busca - Complexidade

- ▶ A busca em uma árvore binária tem complexidade $O(h)$
- ▶ A altura de uma árvore é,
 - ▶ no pior caso, n
 - ▶ no melhor caso, $\lfloor \log_2 n \rfloor + 1$ (árvore completa)
- ▶ Inserção e remoção também têm complexidade de pior caso $O(h)$, e portanto, a inserção ou a remoção de n chaves toma tempo
 - ▶ $O(n^2)$ no pior caso ou
 - ▶ $O(n \log n)$ se pudermos garantir que árvore tem altura logarítmica

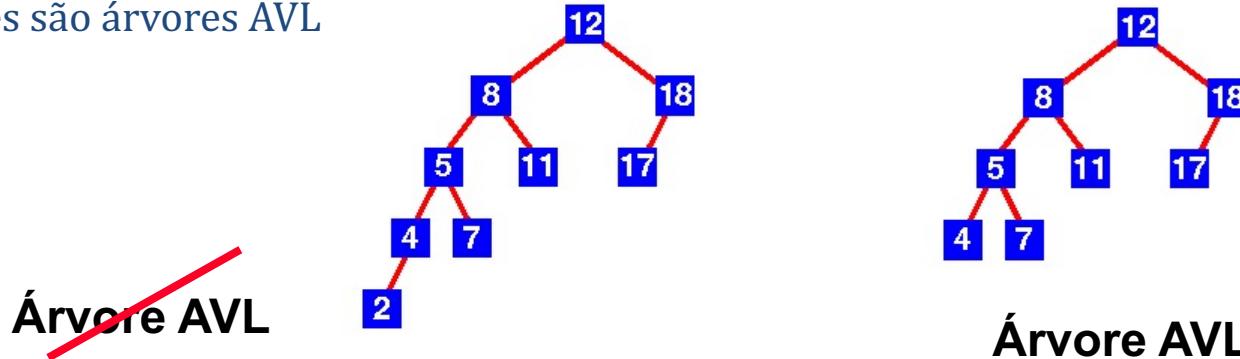
Árvore Binária Balanceada

- ▶ Árvore Binária Balanceada
 - ▶ para cada nó, as alturas de suas subárvoreas diferem de, no máximo 1.



Árvores AVL

- ▶ 1962: Matemáticos Russos G.M. Adelson-Velskki e E.M.Landis sugeriram uma definição para "near balance" e descreveram procedimentos para inserção e eliminação de nós nessas árvores.
- ▶ Propriedades
 - ▶ árvore binária
 - ▶ altura da subárvore esquerda e altura da subárvore direita diferem de no máximo uma unidade
 - ▶ subárvore são árvores AVL



Árvores AVL

- ▶ Em geral, rebalancear uma árvore quando ela deixa de ser completa (devido a uma inserção ou remoção por exemplo) pode ser muito custoso (até n operações)
- ▶ Uma idéia é estabelecer um critério mais fraco que, não obstante, garanta altura logarítmica
- ▶ O critério sugerido por Adelson-Velskii e Landis é o de garantir a seguinte invariante:
 - ▶ Para cada nó da árvore, a altura de sua subárvore esquerda e de sua subárvore direita diferem de no máximo 1
 - ▶ Para manter essa invariante depois de alguma inserção ou remoção que desbalanceie a árvore, utiliza-se operações de custo $O(1)$ chamadas rotações

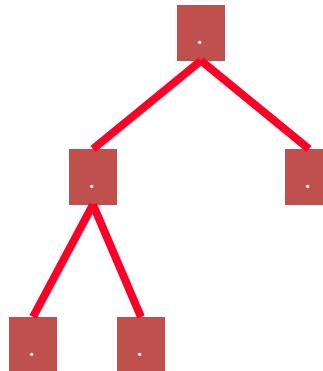
Árvores AVL

- ▶ Uma árvore AVL tem altura logarítmica?
 - ▶ Seja $N(h)$ o número mínimo de nós de uma árvore AVL de altura h
 - ▶ Claramente, $N(1) = 1$ e $N(2) = 2$
 - ▶ Em geral, $N(h) = N(h - 1) + N(h - 2) + 1$
 - ▶ Essa recorrência é semelhante à recorrência obtida para a série de Fibonacci
 - ▶ Sua solução resulta aproximadamente em

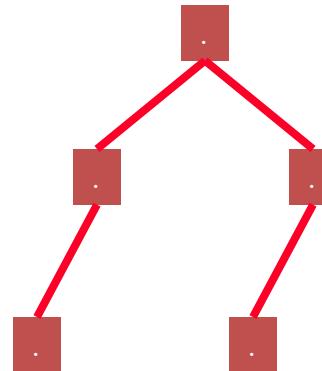
$$N(h) \approx \left(\frac{1+\sqrt{5}}{2} \right)^h \approx 1.618^h$$

Árvore Perfeitamente Balanceada

- Árvore Binária Perfeitamente Balanceada
 - para cada nó, o **número de nós** de suas subárvore diferem de no máximo, 1.
 - Árvore com menor altura para o seu número de nós.



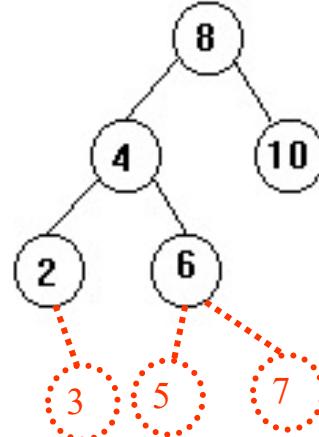
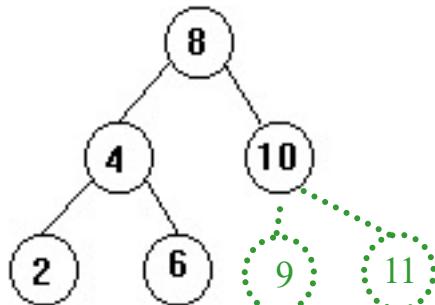
Balanceada



Perfeitamente Balanceada

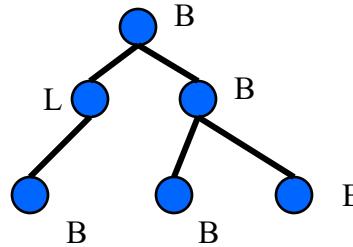
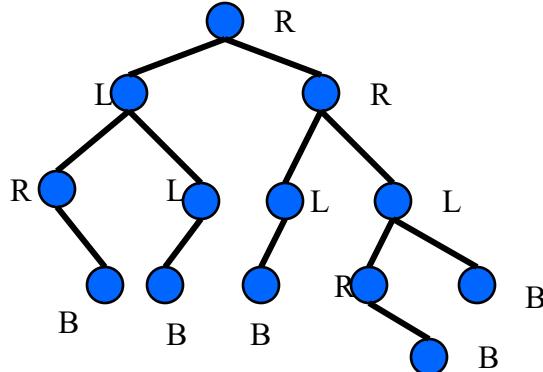
Árvores AVL - Exemplo Desbalanceamento

- ▶ Nós 9 ou 11 pode ser inseridos sem balanceamento .
- ▶ Inserção dos nós 3, 5 ou 7 requerem que a árvore seja rebalanceada!



Indicador de balanceamento

- ▶ Para prevenir desbalanceamento, cada nó possui um indicador de balanceamento:
 - ▶ LeftHeavy - subárvore esquerda do nó possui altura uma unidade maior que a da direita.
 - ▶ Balance - as duas subárvore possuem mesma altura
 - ▶ RightHeavy - subárvore direita do nó possui altura uma unidade maior que a da esquerda.



Árvores AVL - Estrutura de Dados

- ▶ Acrescenta-se o flag para indicar o estado de balanceamento

```
struct AVL_node {  
    char bf;  
    void *item;  
    struct AVL_node *left, *right;  
}AVLNode;
```

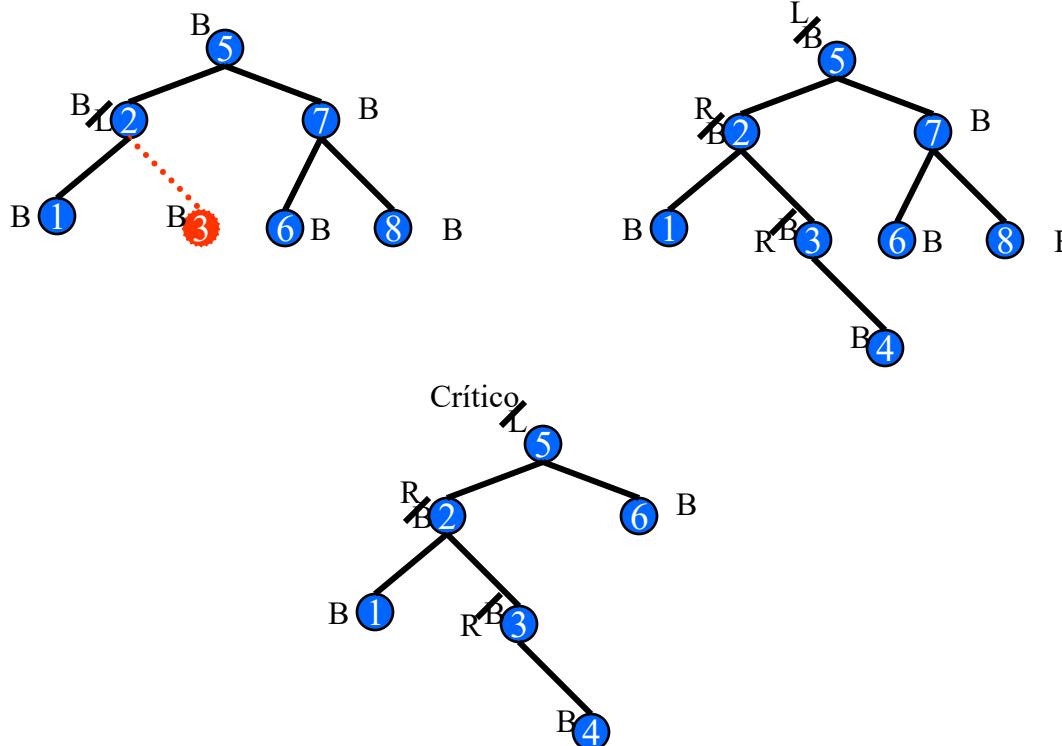
- ▶ Inserção

- ▶ Insira um novo nó, como em qualquer árvore binária
- ▶ Verifique se causou desbalanceamento e rebalanceie se necessário

Inserção - Atualização do Indicador de Balanceamento

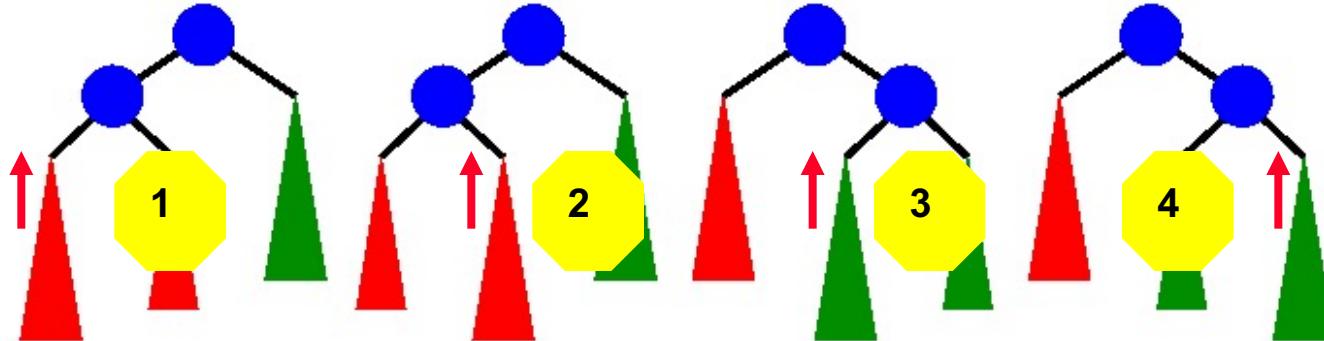
- ▶ Somente nós no caminho entre o nó inserido e a raiz (Path)
- ▶ Mudanças possíveis:
 - ▶ nó estava LeftHeavy ou RightHeavy e se tornou balanceado
 - ▶ o indicador de todos os ancestrais se mantém inalterados;
 - ▶ nó estava balanceado e se tornou LeftHeavy ou RightHeavy
 - ▶ muda os indicador dos ancestrais
 - ▶ nó estava LeftHeavy ou RightHeavy e se tornou desbalanceado
 - ▶ nó denominado nó crítico, sujeito a rebalanceamento

Inserção - Atualização do Indicador de Balanceamento



Árvores AVL - Rebalanceamento

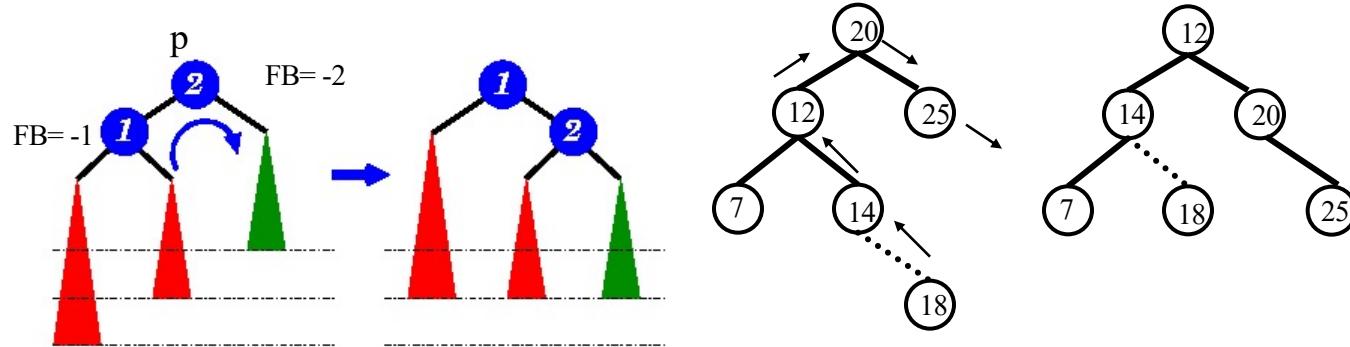
- ▶ Inserção pode gerar uma árvore não balanceada, quebrando a condição de árvore AVL
 - ▶ 4 casos



- ▶ 1 e 4 representam o mesmo caso espelhado
- ▶ 2 e 3 representam o mesmo caso espelhado

Árvores AVL - Rebalancemaneto

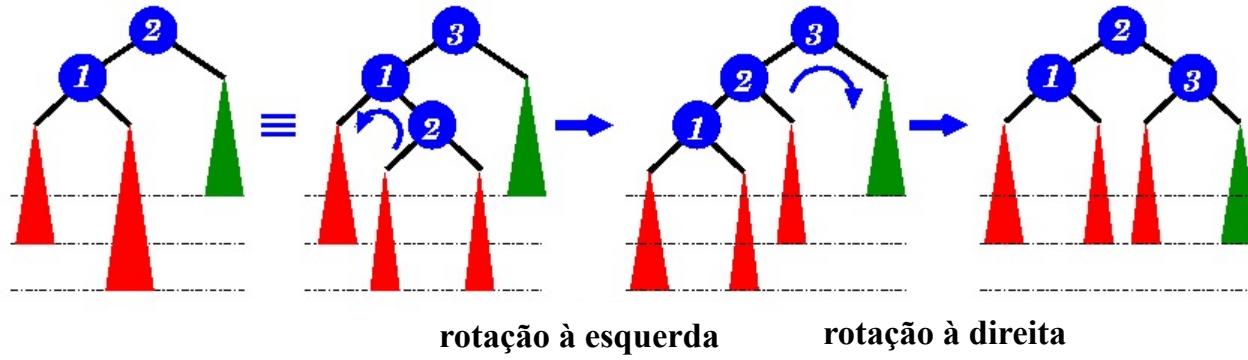
- ▶ Caso 1: Uma rotação



- ▶ Caso 4 resolvido com rotação semelhante

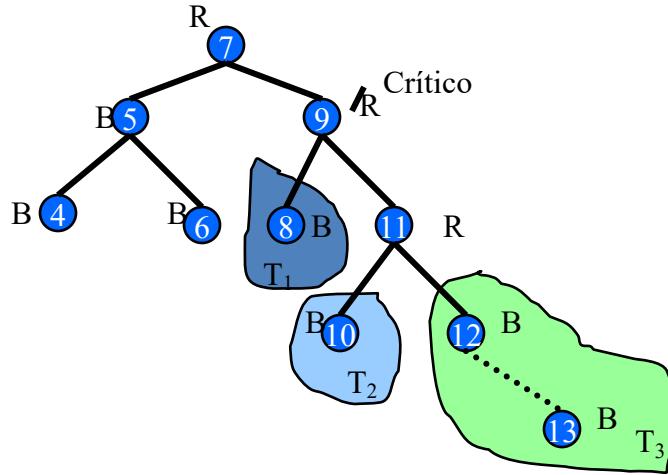
Árvores AVL - Rebalancimento

- ▶ Caso 2 : resolvido com duas rotações

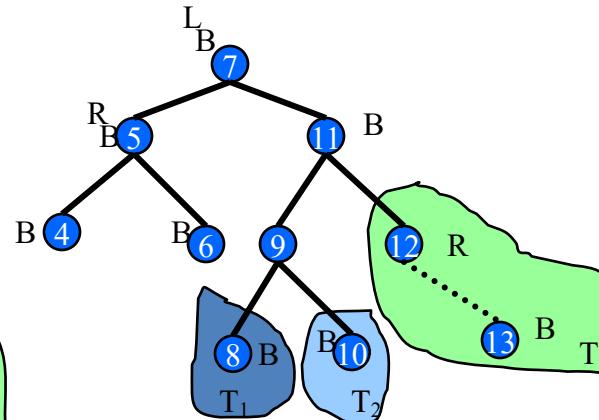


- ▶ Caso 3 é resolvido com as rotações inversas

Árvores AVL - Algoritmo de Inserção - Exemplos

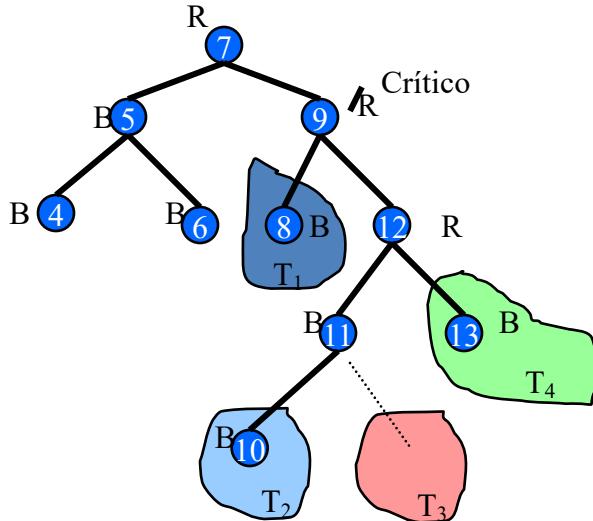


antes do balanceamento

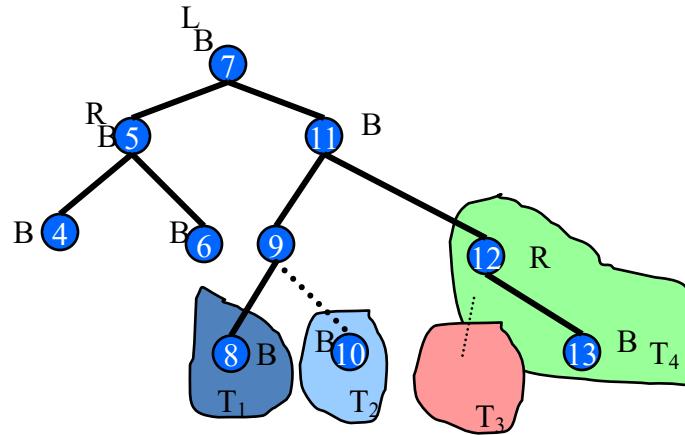


após o balanceamento

Árvores AVL - Algoritmo de Inserção - Exemplos



antes do balanceamento



após o balanceamento

Árvores AVL - Algoritmo de Inserção

```

void InsertAVL ( AVLNode *root, void *elm,
                 void (*Compare)(void *a, void *b))
{
    int level;  char direction[MAX];
    AVLNode path[MAX], *t;
    if ( root == NULL ) {
        root = NewNode( elm );
        return;
    }
    level = 0;  direction[level] = 'L';
    path[level] = root; t = root;
    while (inserted == FALSE) {
        if ( Compare(t, elm) < 0 ) {
            level++; path[level] = t;
            direction[level] = 'L';
            if( t->left != NULL ) {
                t = t->left;
            } else {
                new = NewNode(elm);
                t->left = new;
                inserted = TRUE;
            }
        } else if(Compare(t,elm) >=0 ) {
            level++; path[level] = t;
            direction[level] = 'R';
            if( t->right != NULL ) {
                t = t->right;
            } else {
                new = NewNode(elm);
                t->right = new;
                inserted = TRUE;
            }
        } /* fi */
    } /* elihw */
    UpdateBalancing(root, direction, path, level,
                    Compare);
}

```

Árvores AVL - Algoritmo de Inserção

```
void UpdateBalancing ( char *direction , AVLNode*path,
                      int level, void (*comp)(void *, void *)) {
    int mark = 0, i = level, found=FALSE;
    AVLNode x, y; char d;
    while(i>0 && found == FALSE){
        if(path[i]->bf != Balanced) {
            mark = i; found = TRUE;
        } /* fi */
        i++;
    } /* elihw */
    for (i=mark+1; i<level; i++) {
        if(comp(path[i]->data, elm) <0) {
            path[i]->bf = LeftHeavy;
        } else {
            path[i]->bf = RightHeavy;
        }
    }
    if ( mark = 0 ) {
        return;
    } else {
        } else {
            d = direction[mark];
            x = path[mark];
            y = path[mark+1];
            if ( x->bf != d ) {
                x->bf = Balanced;
                return;
            }
            if ( y->bf == d ) {
                BalanceCase1( x, y, path[mark-1] );
                return;
            }
            BalanceCase2(x, y, path[mark-1]);
        } /* fi mark=0 */
    }
}
```

Árvores AVL - Algoritmo de Inserção

```
void BalanceCase1 ( AVLNode *x, AVLNode *y, AVLNode *father, char direction)
{
    if (direction == 'L') {
        x->left = y->right;
        y->right = x;
    } else {
        x->right = y;
        y->left = x;
    }
    x->bf = y->bf = 'B';
    if ( x = father->left ) {
        father->left = y;
    } else {
        father->right = y;
    }
}
```

Árvores AVL - Algoritmo de Inserção

```
void BalanceCase1 ( AVLNode *x, AVLNode *y,
                    AVLNode *father, char direction)
{
    AVLNode *z;
    if (direction == 'L') {
        z = y->right;
        y->right = z->left;
        z->left = y;
        x->left = z->right;
        z->right = x;
    } else {
        z = y->right;
        y->left = z;
        z->right = y;
        x->right = z->left;
        z->left = x;
    }
    if ( x = father->left ) {
        father->left = z;
    } else {
        father->right = z;
    }

    if ( z->bf == direction ) {
        y->bf = z->bf = 'B';
        if ( direction = 'L' ) {
            x->bf = 'R';
        } else {
            x->bf = 'L';
        }
    } else {
        if ( z->bf = 'B' ) {
            x->bf = y->bf = z->bf = 'B';
        } else {
            x->bf = z->bf = 'B';
            y->bf = direction;
        }
    }
}
```

Árvores Rubro Negras

- ▶ Árvore binária com um atributo extra: a cor do nó (preto ou vermelho) e com a ligação com o pai.

```
struct _redblack_node {  
    enum { red, black } colour;  
    void *item;  
    struct redblack_node *left, *right, *parent;  
}RBNODE;
```

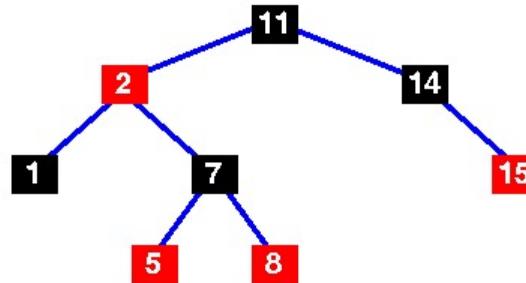
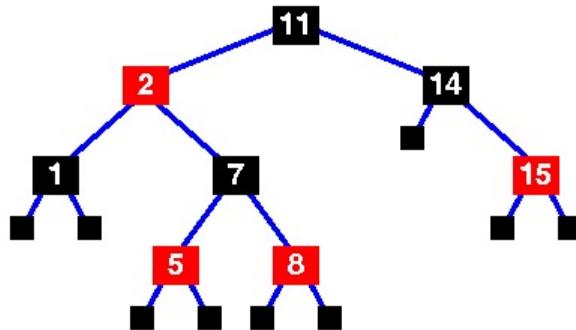
- ▶ Definição: árvore binária de busca com as seguintes propriedades:
 1. cada nó é vermelho ou preto.
 2. Cada nó folha é preto.
 3. Se um nó é vermelho, seus dois filhos são pretos.
 4. Cada caminho entre um nó e seus nós folhas descendentes contém o mesmo número de nós pretos.

Árvores Rubro-Negras

- ▶ São árvores balanceadas segundo um critério ligeiramente diferente do usado em árvores AVL
- ▶ A todos os nós é associada uma cor que pode ser vermelha ou negra de tal forma que:
 - ▶ Nós externos (folhas ou nulos) são negros
 - ▶ Todos os caminhos entre um nó e qualquer de seus nós externos descendentes percorre um número idêntico de nós negros
 - ▶ Se um nó é vermelho (e não é a raiz), seu pai é negro
- ▶ Observe que as propriedades acima asseguram que o maior caminho desde a raiz uma folha é no máximo duas vezes maior que o de o qualquer outro caminho até outra folha e portanto a árvore é aproximadamente balanceada

Árvores Rubro Negras

- raiz é sempre preto
- raiz subárvore inexistentes também são preto

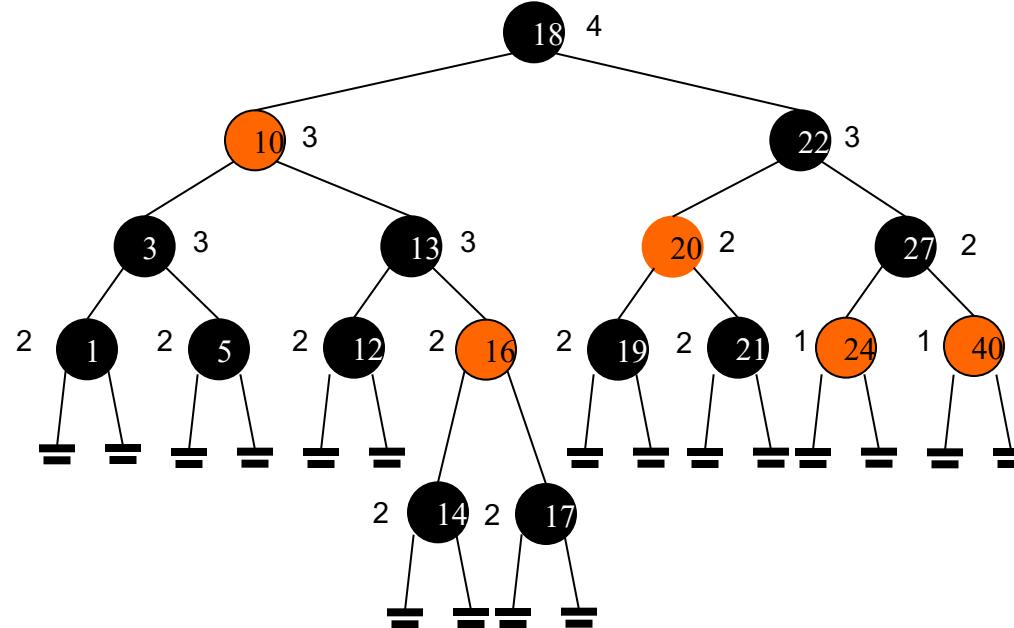


Vantagem:

Rebalanceamento é feito em um passo único

Árvores Rubro-Negras

- Altura negra de um nó = número de nós negros encontrados até qualquer nó folha descendente



Árvores Rubro-Negras

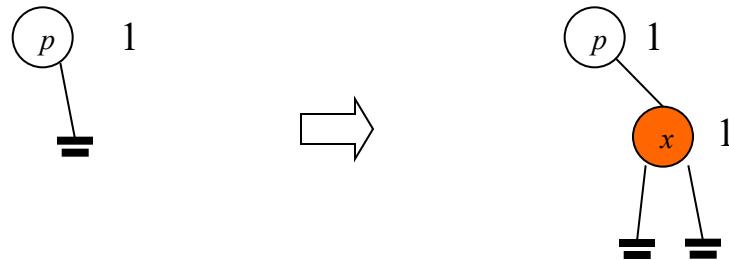
- ▶ Lema 1: Um nó x de uma árvore rubronegra tem no mínimo $2an(x) - 1$ nós internos, onde $an(x)$ é a altura negra de x
- ▶ Prova por indução
 - ▶ Caso base: Um nó de altura 0 (i.e., nó-folha) tem
 $0 = 2 \cdot 0 - 1$ nós internos
 - ▶ Caso genérico: Um nó x de altura $h > 0$ tem 2 filhos com altura negra $an(x)$ ou $an(x) - 1$, conforme x seja vermelho ou negro. No pior caso, x é negro e as subárvores enraizadas em seus 2 filhos têm $2an(x) - 1 - 1$ nós internos cada e x tem $2(2an(x) - 1 - 1) + 1 = 2an(x) - 1$ nós internos

Árvores Rubro-Negras

- ▶ Lema 2: Uma árvore rubro-negra com n nós tem no máximo altura $2 \log_2(n+1)$
 - ▶ Prova: Se uma árvore tem altura h , a altura negra de sua raiz será no mínimo $h/2$ (pelo critério 3 de construção) e a árvore terá $n \geq 2h/2 - 1$ nós internos (Lema 1)
- ▶ Como consequência, a árvore tem altura $O(\log n)$ e as operações de busca, inserção e remoção podem ser feitas em $O(\log n)$

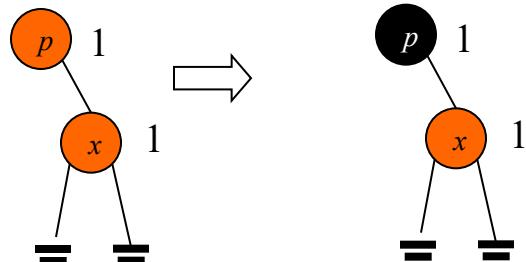
Inserção em Árvore Rubro-Negra

- ▶ Ao contrário da árvore AVL, agora temos agora vários critérios para ajustar simultaneamente
- ▶ Ao inserir um nó x numa posição vazia da árvore (isto é, no lugar de um nó nulo) este é pintado de vermelho. Isto garante a manutenção do critério (2), já que um nó vermelho não contribui para a altura negra



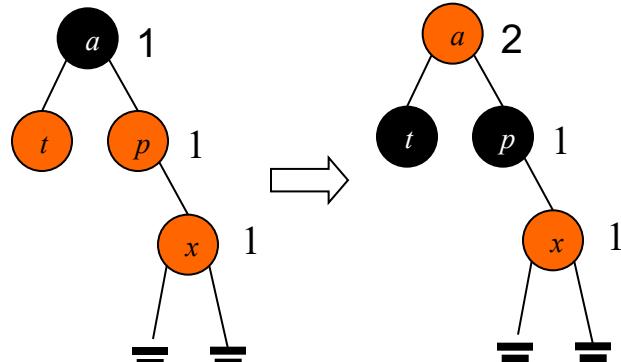
Inserção em Árvore Rubro-Negra

- ▶ [Caso 0] Se x não tem pai ou se p , o pai de x , é preto, nada mais precisa ser feito já que o critério (3) também foi mantido
- ▶ [Caso 1] Suponha agora que p é vermelho. Então, se p não tem pai, então p é a raiz da árvore e basta trocar a cor de p para preta



Inserção em Árvore Rubro-Negra

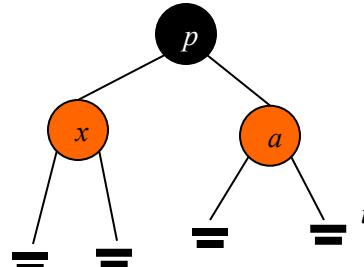
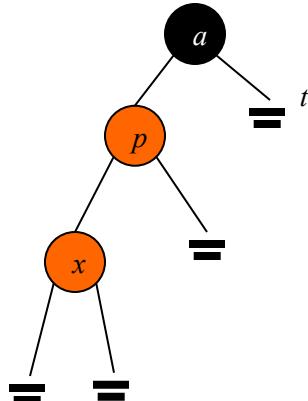
- ▶ [Caso 2] Suponha agora que p é vermelho e a, o pai de p (e avô de x) é preto. Se t, o irmão de p (tio de x) é vermelho, ainda é possível manter o critério (3) apenas fazendo a recoloração de a, t e p



Obs.: Se o pai de a é vermelho, o rebalanceamento tem que ser feito novamente

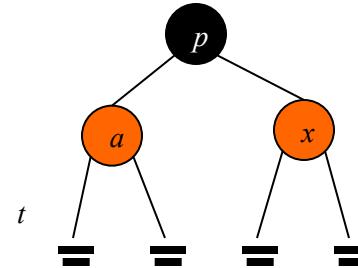
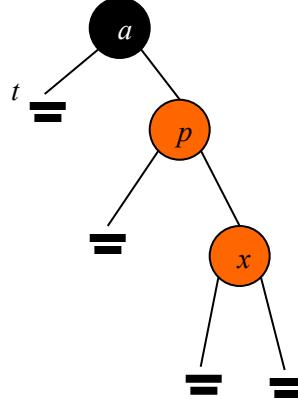
Inserção em Árvore Rubro-Negra

- ▶ [Caso 3] Finalmente, suponha que p é vermelho, seu pai a é preto e seu irmão t é preto. Neste caso, para manter o critério (3) é preciso fazer rotações envolvendo a , t , p e x . Há 4 subcasos que correspondem às 4 rotações possíveis:
 - ▶ [Caso 3a] Rotação Direita



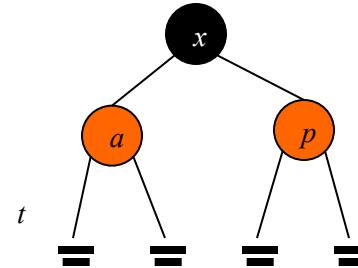
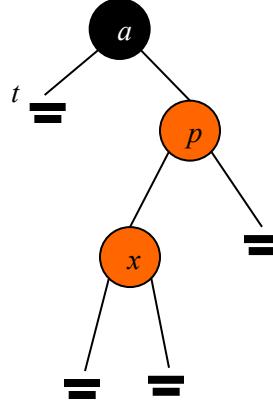
Inserção em Árvore Rubro-Negra

- ▶ [Caso 3b] Rotação Esquerda



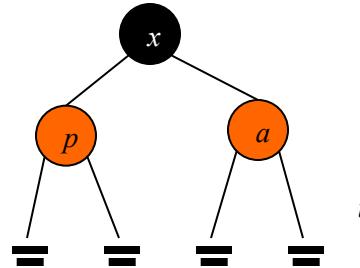
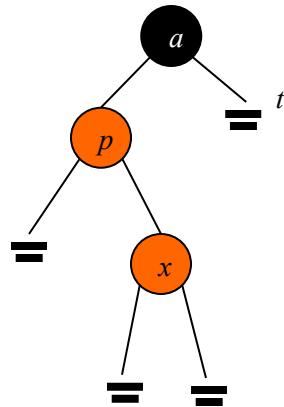
Inserção em Árvore Rubro-Negra

- ▶ [Caso 3c] Rotação Dupla Esquerda



Inserção em Árvore Rubro-Negra

- ▶ [Caso 3d] Rotação Dupla Direita



Inserção em Árvore Rubro-Negra

```
proc InsereRN (Chave v, var ArvoreRN a, p, x) {  
    se x = Nulo então {  
        x = Aloca (NoArvoreRN)  
        x->Esq, x->Dir, x->Val, x->Cor = Nulo, Nulo, v, Vermelho  
    } senão {  
        se v < x->val então  
            InsereRN (v, p, x, x->Esq)  
        senão se v > x->val então  
            InsereRN (v, p, x, x->Dir)  
    }  
    Rebalanceia (a, p, x)  
}
```

Inserção em Árvore Rubro-Negra

```
proc Rebalanceia (var ArvoreRN a, p, x) {  
    se x->Cor = Vermelho e p1 Nulo então  
        se p->Cor = Vermelho então  
            se a = Nulo então          % Caso 1  
                p->cor := Negro  
            senão se a->Cor = Negro então  
                se a->Esq1 Nulo e a->Dir1 Nulo e a->Esq->Cor = Vermelho  
                    e a->Dir->Cor = Vermelho então      % Caso 2  
                        a->Cor, a->Esq->Cor, a->Dir->Cor = Vermelho, Negro, Negro  
                    senão  
                        se p = a->Esq  
                            se x = p->Esq então RotacaoDireita (a)  % Caso 3a  
                            senão RotacaoDuplaDireita (a)      % Caso 3d  
                        senão  
                            se x = p->Dir então RotacaoEsquerda (a) % Caso 3b  
                            senão RotacaoDuplaEsquerda (a)        % Caso 3c
```

Inserção em Árvore Rubro-Negra

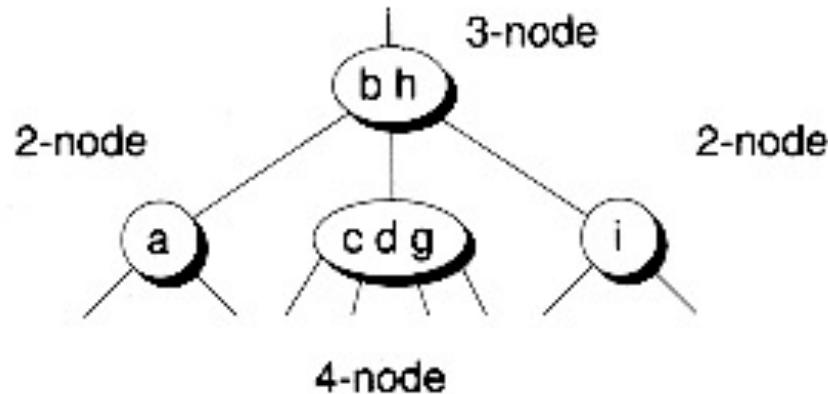
```
proc RotacaoDireita (var ArvoreRN T) {  
    T, T->Esq, T->Esq->Dir = T->Esq, T->Esq->Dir, T  
    T->Cor, T->Dir->Cor = T->Dir->Cor, T->Cor  
}  
proc RotacaoEsquerda (var ArvoreRN T) {  
    T, T->Dir, T->Dir->Esq = T->Dir, T->Dir->Esq, T  
    T->Cor, T->Esq->Cor = T->Esq->Cor, T->Cor  
}  
proc RotacaoDuplaDireita (var ArvoreRN T) {  
    RotacaoEsquerda (T->Esq)  
    RotacaoDireita (T)  
}  
proc RotacaoDuplaEsquerda (var ArvoreRN T) {  
    RotacaoDireita (T->Dir)  
    RotacaoEsquerda (T)  
}
```

Complexidade da Inserção em Árvore Rubro-Negra

- ▶ Rebalanceia tem custo $O(1)$
- ▶ RotacaoXXX têm custo $O(1)$
- ▶ InsereRN tem custo $O(\log n)$

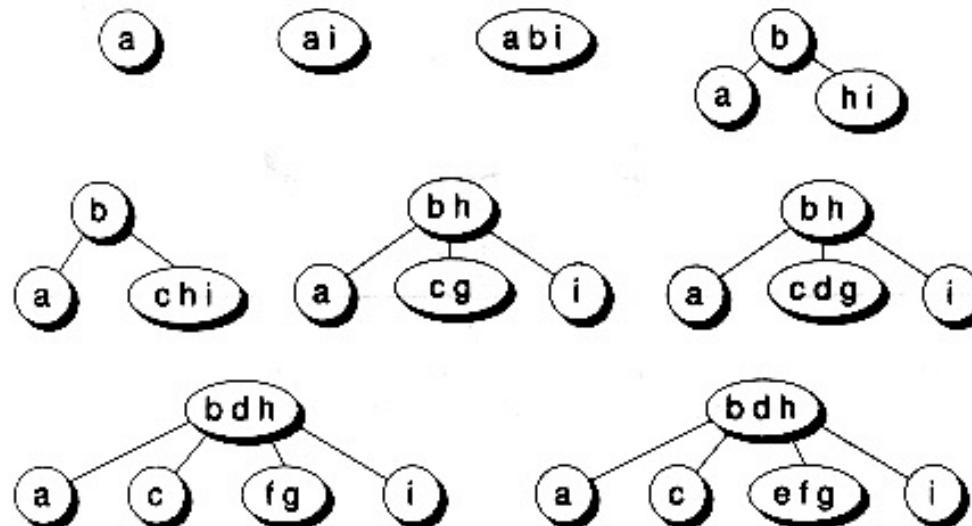
Árvores 2-3-4

- ▶ É uma árvore onde cada nó pode ter mais do que uma chave
- ▶ Na verdade, uma árvore 2-3-4 é uma árvore B onde a capacidade de cada nó é de até 3 chaves (4 ponteiros)



Árvores 2-3-4

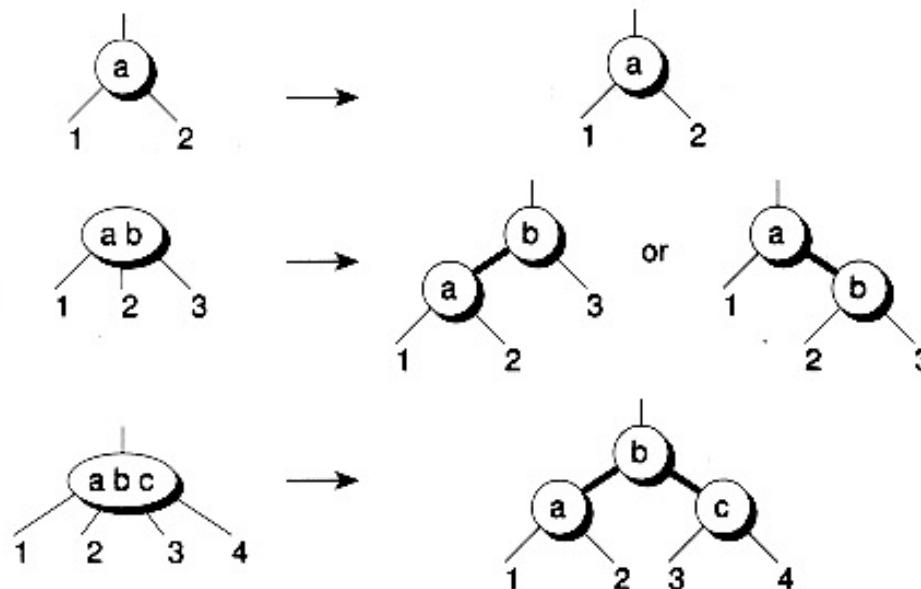
- Exemplo de inserção em árvores 2-3-4



Insertion sequence: a-i-b-h-c-g-d-f-e

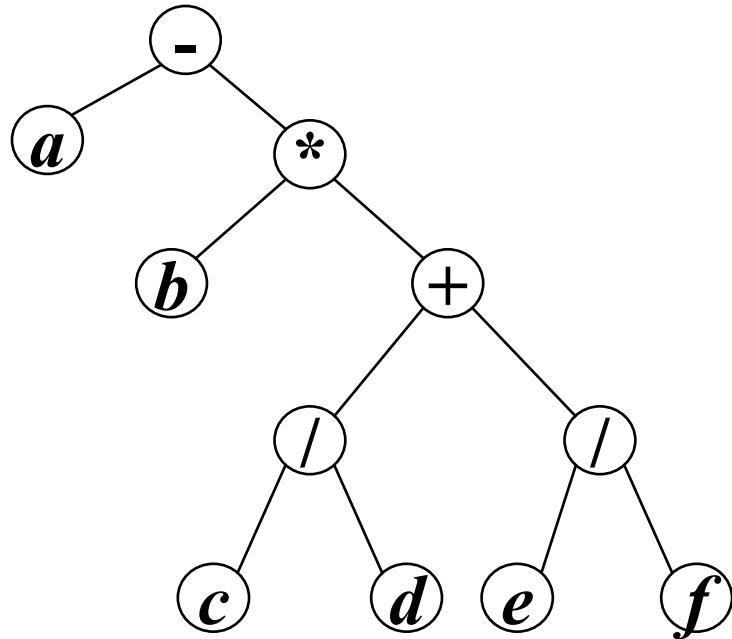
Árvores 2-3-4

- ▶ Na verdade, uma árvore 2-3-4 pode ser implementada como uma árvore binária Rubro-Negra



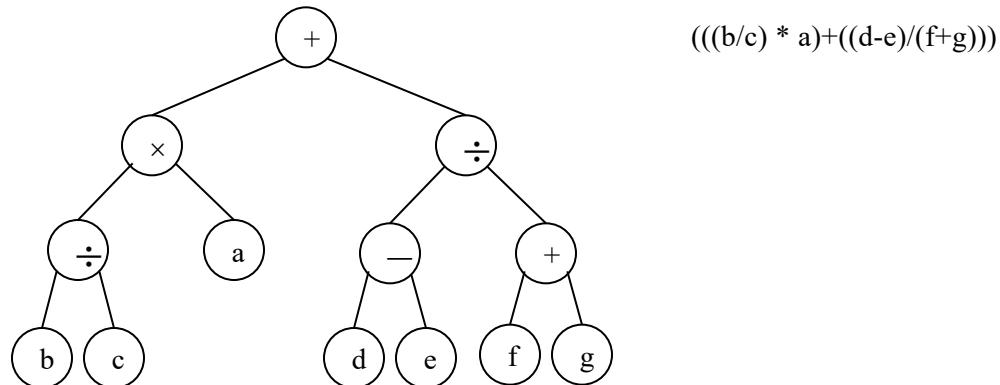
Aplicações

- ▶ Representação de expressões
- ▶ Exemplo:
 - ▶ $a - b(c / d + e / f)$ corresponde a arvore
- ▶ Caminhamento em:
- ▶ Preordem – prefixada
- ▶ Posordem - posfixada
- ▶ Simetrico - infixa



Aplicação: Expressões

- ▶ Uma aplicação bastante comum de árvores binárias é na representação e processamento de expressões algébricas, booleanas, etc



Avaliando uma Expressão

- ▶ Se uma expressão é codificada sob a forma de uma árvore, sua avaliação pode ser feita percorrendo os nós da árvore

```
proc Avalia (Arvore T) {  
    se T->Val é uma constante ou uma variável então  
        retornar o valor de T->Val  
    senão {  
        operando1 = Avalia (T->Esq)  
        operando2 = Avalia (T->Dir)  
        se T->Val = "+" então  
            retornar operando1 + operando2  
        senão se T->Val = "-" então  
            retornar operando1 - operando2  
        senão se T->Val = "*" então  
            retornar operando1 * operando2  
        senão se T->Val = "/" então  
            retornar operando1 / operando2  
    }  
}
```

Resultados em Arvores Binárias

- ▶ Numero de folhas em uma arvore binária é igual ao número de nós de grau dois mais 1
 - ▶ Prova por indução no numero de nós
- ▶ Arvore binária completa de altura h possui $2h+1-1$ nós
- ▶ Arvore binária completa de altura h possui 2^h folhas

Aplicação – Código de Huffman

- ▶ Problema: Codificar uma mensagem (seqüência de caracteres) como uma seqüência de bits
- ▶ Dadas as probabilidades de aparecimento dos caracteres. Por exemplo:
 - ▶ caracteres a, b, c, d, tem probabilidades de aparecimento .12, .4, .15, .08, .25
- ▶ Para decodificar a mensagem o código deve possuir a propriedade de prefixo, isto é não existe caractere cujo código seja prefixo do código de outro

Aplicação – Exemplo de Códigos

- ▶ Duas codificações possíveis:
- ▶ A codificação da mensagem bcd é 001010011 com o código 1, e 1101001 com código 2.

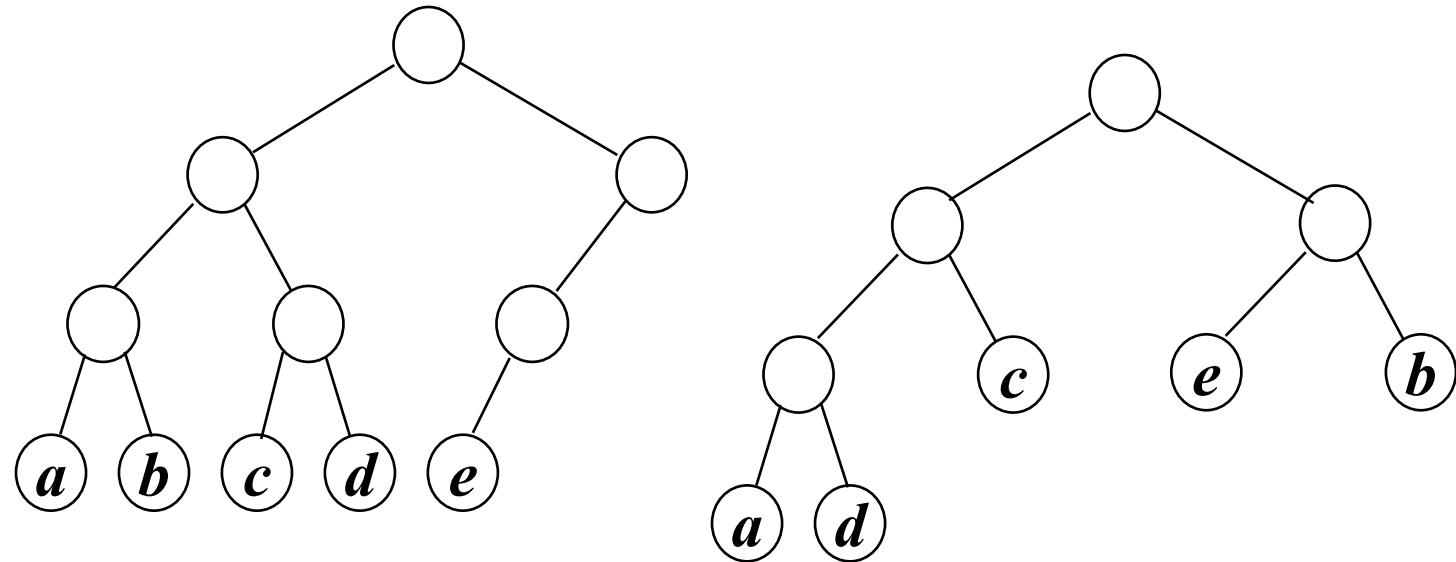
Aplicacao – Comprimento do Código

- ▶ Comprimento do código é a média ponderada dos comprimentos dos códigos para cada caractere
 - ▶ Por exemplo: código 1 tem comprimento 3, e código 2 possui 2.2
- ▶ Código de comprimento menor codificam mensagens com menos bits
- ▶ Objetivo: encontrar código com o menor comprimento possível

Prop. de Prefixo e Arv. Binarias

- ▶ Códigos binários correspondem a caminhos em uma arvore
 - ▶ Comece na raiz
 - ▶ Descendo pra direita adicione 1 ao código e descendo para a esquerda adicione zero
- ▶ Códigos para caminhos que vão ate as folhas possuem a propriedade de prefixo

Exemplos: - Códigos e Arvores



Construção da Árvore

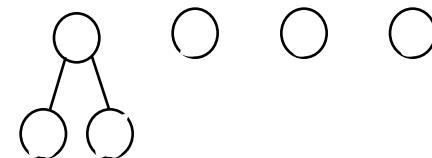
- ▶ Considere uma floresta de árvores, uma para cada caractere do código
- ▶ A probabilidade de cada árvore é igual a soma das probabilidades de suas folhas
- ▶ Cada passo do algoritmo combina as duas árvores com probabilidade mínima em uma só
- ▶ Quando há somente uma árvore esta é a arvore que gera códigos de comprimento mínimo

Exemplo de Código de Huffman

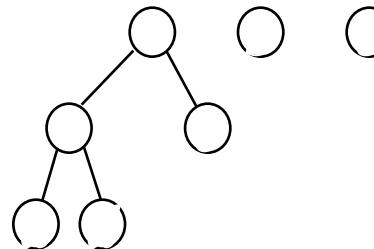
.12 .40 .15 .08 .25



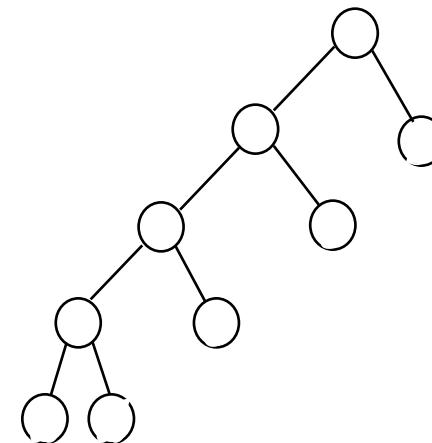
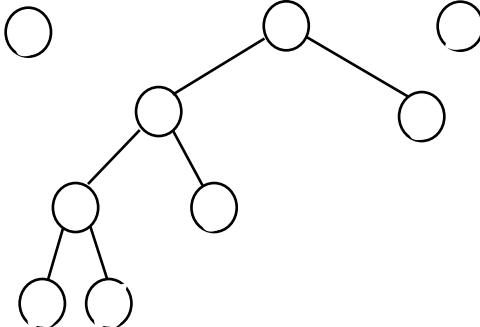
.20 .40 .15 .25



.35 .40 .25

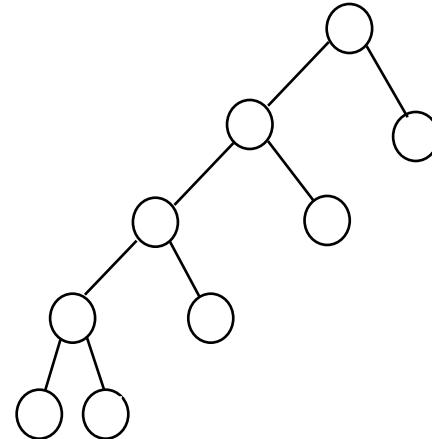


.60 .40



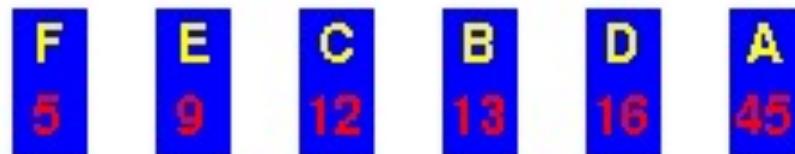
Exemplo de Código de Huffman

- ▶ O código ée:
 - ▶ a 0000
 - ▶ b 1
 - ▶ c 001
 - ▶ d 0001
 - ▶ e 01
- ▶ Com comprimento 2.15



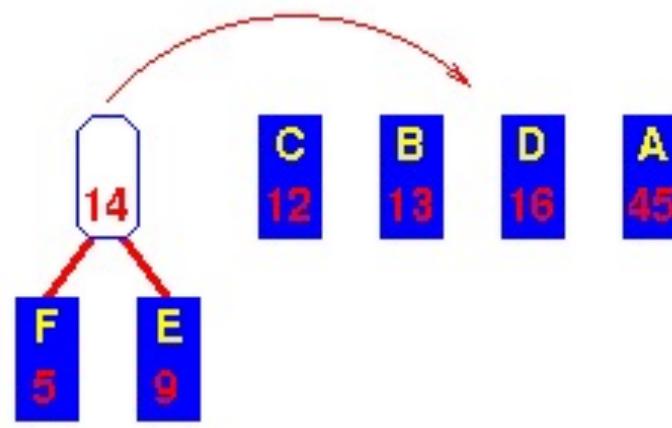
Huffman –Exemplo Codificação

1 - Cria árvores com um caractere em cada uma, ordenados pela probabilidade



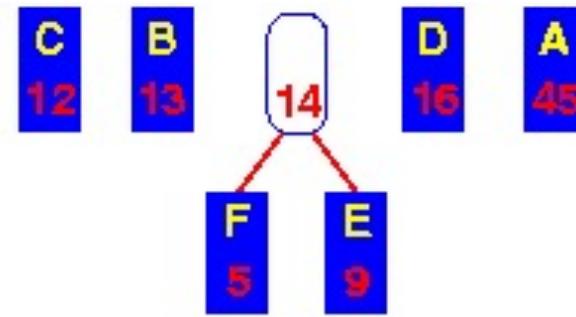
2 - Combina árvores de menor probabilidade

3 - Reordena as árvores



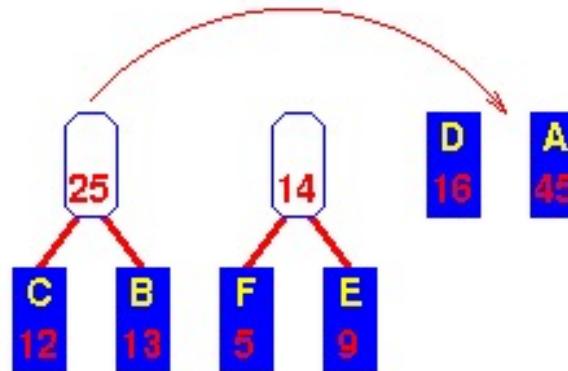
Huffman –Exemplo Codificação

Após reordenamento
das árvores ...



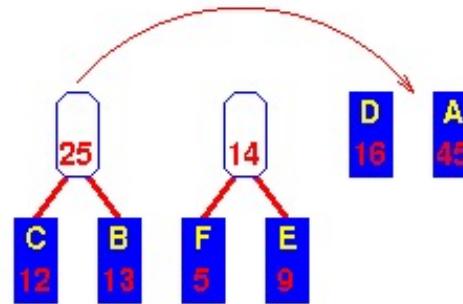
2 - Combina as de menor prob.

3 - Reordena novamente



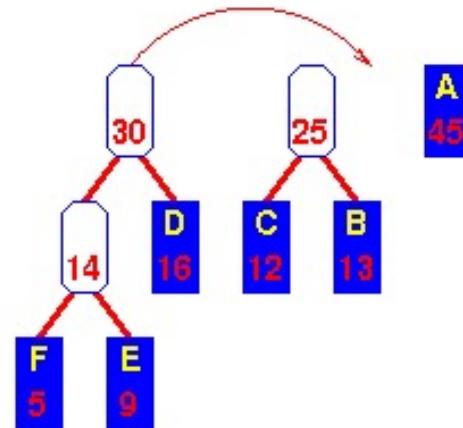
Huffman –Exemplo Codificação

Após a reordenação ...

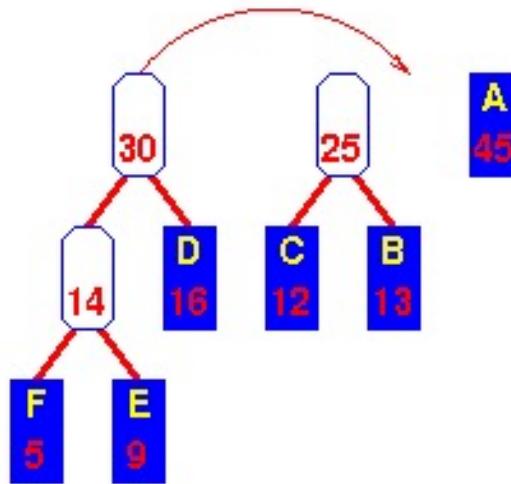


Agora as duas menores são
A de 14 e a D

Combina e reordena



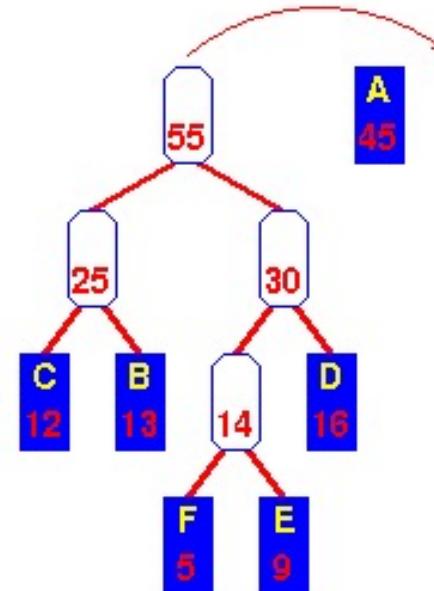
Huffman –Exemplo Codificação



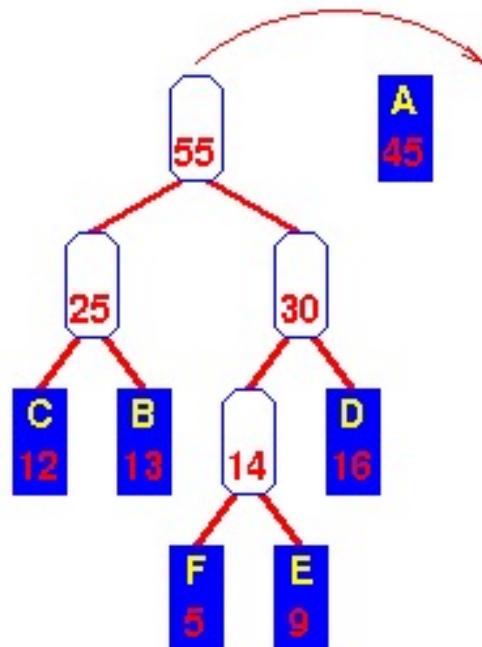
Apos a reordenação ...

Agora as menores são a de
25 e a de 30

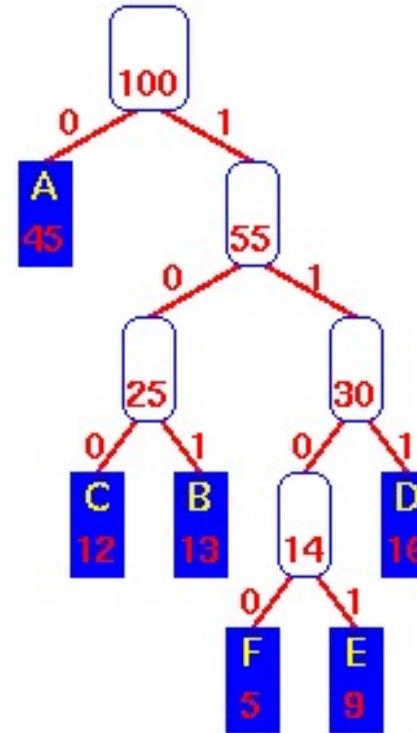
Combina e reordena



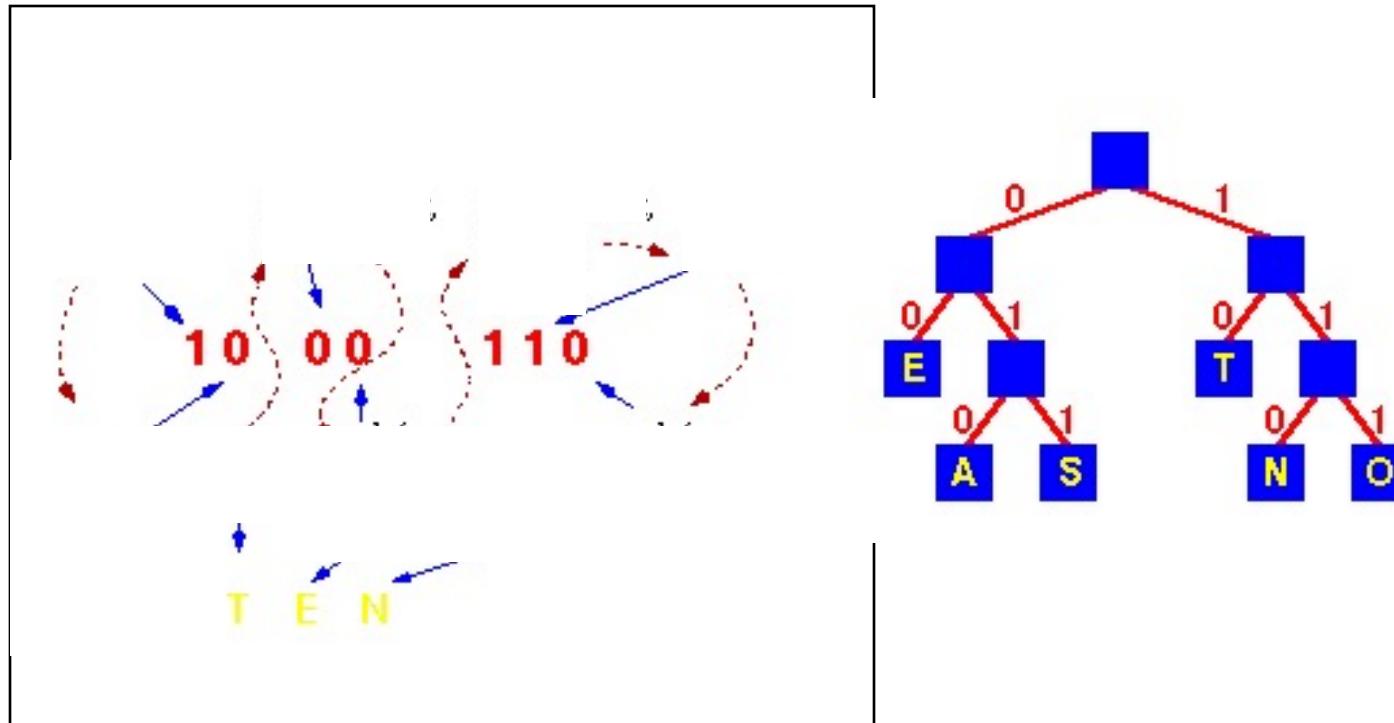
Huffman –Exemplo Codificação



Combina as
duas últimas



Huffman –Exemplo Decodificação



Árvores de Busca de Altura Ótima

- ▶ É fácil ver que podemos garantir uma árvore de altura ótima para uma coleção de chaves se toda vez que temos que escolher uma chave para inserir, optamos pela mediana:

```
proc InserirTodos (i, n, A [i .. i+n-1], var Árvore T) {  
    se n = 1 então Inserir (A [i], T)  
    senão {  
        j = Mediana (i, n, A)  
        trocar A[i] com A [j]  
        m = Particao (i, n, A)  
        Inserir (A [i+m], T)  
        InserirTodos (i, m, A, T->Esq)  
        InserirTodos (i+m+1, n-m-1, A, T->Dir)  
    }  
}
```

Árvores de Busca de Altura Ótima

- ▶ Sabendo que tanto Mediana quanto Particao podem ser feitos em $O(n)$ o algoritmo InsereTodos executa em tempo $O(n \log n)$
- ▶ O algoritmo pode ser reescrito de forma mais sucinta se admitirmos que o array A se encontra ordenado
- ▶ Nem sempre, entretanto, podemos garantir que conhecemos todas as chaves de antemão
- ▶ O que esperar em geral?
 - ▶ Percebemos que a altura da árvore final depende da ordem de inserção
 - ▶ Temos $n!$ possíveis ordens de inserção
 - ▶ Se todas as ordens de inserção são igualmente prováveis, no caso médio teremos uma árvore de altura $\approx 1 + 2.4 \log n$

Altura de Árvores Binárias de Busca – Caso Médio

- ▶ Eis uma explicação intuitiva para o fato de que no caso médio, a altura de uma árvore binária de busca é $O(\log n)$
- ▶ Considere uma coleção ordenada de n chaves
 - ▶ Vemos que se escolhemos a $k^{\text{ésima}}$ chave para inserir primeiro, teremos à esquerda do nó raiz uma subárvore com $k - 1$ nós e à direita uma subárvore com $n - k$ nós
 - ▶ No melhor caso $k = n / 2$
 - ▶ No pior caso, $k = 1$ ou $k = n$
 - ▶ Admitamos que o caso médio corresponde a $k = n/4$ ou $k=3n/4$ (estamos ignorando tetos, pisos, etc)

Altura de Árvores Binárias de Busca – Caso Médio

- ▶ Em qualquer caso, a subárvore com $3n/4$ nós vai dominar a altura da árvore como um todo
- ▶ Resolvendo a recursão (novamente ignorando o fato que $3n/4$ nem sempre é um inteiro), temos

▶

$$H(1) = 1$$

$$\begin{aligned} H(n) &= 1 + H(3n/4) \\ &= 2 + H(9n/16) \\ &= 3 + H(27n/64) \\ &= \dots \\ &= m + H((3/4)^m n) \end{aligned}$$

$$\therefore m = \frac{\log_2 n}{\log_2 4/3} \approx 2.4 \log_2 n$$

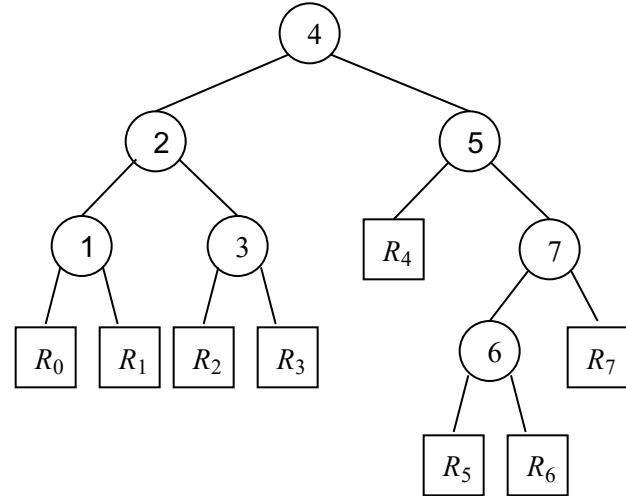
$$H(n) = 1 + 2.4 \log_2 n$$

Árvores Binárias de Busca Ótimas

- ▶ Dada uma árvore binária de busca, um dado importante é o número total de comparações que precisamos fazer durante a busca de uma chave
 - ▶ Se a chave buscada é uma chave s_k pertencente à árvore, o número de comparações é o nível da chave na árvore, isto é, l_k
 - ▶ Se a chave x sendo buscada não pertence à árvore, o número de comparações corresponde à subárvore vazia (também chamada de nó externo) que encontramos durante o processo de busca
 - ▶ Cada subárvore nula R_i corresponde a um intervalo entre duas chaves da árvore, digamos s_i e s_{i+1} , isto é, $s_i < x < s_{i+1}$ para algum i entre 1 e n
 - ▶ Os casos extremos R_0 e R_n correspondem a $x < s_1$ e $s_n < x$
 - ▶ O número de comparações para encontrar x , portanto, é o nível dessa subárvore nula (a que chamaremos de $l'k$) menos 1

Árvore de Busca Ótima

- Comprimento de Caminho Interno: $I(T) = \sum_{1 \leq i \leq n} l_i$
- Comprimento de Caminho Externo $E(T) = \sum_{0 \leq i \leq n} (l'_i - 1)$
- No exemplo,
 $I(T)=1+2*2+3*3+4=18$
 $E(T)=2+5*3+2*4=25$
- Em geral, $E(T)=I(T)+n$
- Árvores completas minimizam tanto $E(T)$ quanto $I(T)$



Árvore de Busca Ótima para Freqüências de Acesso Dadas

- ▶ Admitindo probabilidade uniforme no acesso de quaisquer chaves, as árvores completas são ótimas
- ▶ Entretanto, se a distribuição não é uniforme, precisamos empregar um algoritmo mais elaborado
- ▶ Sejam
 - ▶ f_k a freqüência de acesso à $k^{\text{ésima}}$ menor chave, armazenada em T no nível l_k ,
 - ▶ $f'k$ a freqüência de acesso a chaves que serão buscadas nos nós externos R_k , armazenados em T no nível $l'k$,
- ▶ Então, o custo médio de acesso é dado por

Árvore de Busca Ótima para Freqüências de Acesso Dadas

- ▶ O algoritmo para construção de árvores ótimas baseia-se no fato de que subárvores de árvores ótimas são também ótimas
 - ▶ se assim não fosse, poderíamos substituir uma subárvore não ótima por uma ótima e diminuir o custo da árvore ótima, o que é um absurdo
- ▶ O algoritmo consiste de testar todas as n chaves como raízes da árvore e escolher aquela que leva ao custo mínimo
 - ▶ As subárvores são construídas de forma recursivamente idêntica

Árvore de Busca Ótima para Freqüências de Acesso Dadas

- ▶ Seja $T(i, j)$ a árvore ótima para as chaves $\{s_{i+1}, s_{i+2}, \dots, s_j\}$
- ▶ Seja $F(i, j)$ a soma de todas as freqüências relacionadas com $T(i, j)$, isto é,

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

- ▶ Assumindo que $T(i, j)$ foi construída escolhendo s_k como raiz, então prova-se que

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

- ▶ Portanto, para encontrar o valor de k apropriado, basta escolher aquele que minimiza a expressão acima

Árvore de Busca Ótima para Freqüências de Acesso Dadas

- ▶ Seria possível em computar recursivamente $c(T(0,n))$ usando como caso base $c(T(i,i))=0$
- ▶ No entanto, esse algoritmo iria computar cada $c(T(i,j))$ e $F(i,j)$ múltiplas vezes
- ▶ Para evitar isso, valores já computados são armazenados em duas matrizes: $c[0 .. n, 0 .. n]$ e $F[0 .. n, 0 .. n]$
- ▶ Podemos também dispensar a construção recursiva e computar iterativamente o custo de todas as $n(n+1)/2$ árvores envolvidas no processo
 - ▶ As árvores com d nós depende apenas do custo de árvores com $0, 1, 2 \dots$ e até $d - 1$ nós
 - ▶ Computar $F(i,j)$ também não oferece dificuldade

Árvore de Busca Ótima para Freqüências de Acesso Dadas

```
proc CustoArvoreOtima (n, f [1 .. n], f '[0 .. n]) {
    array c [0 .. n, 0 .. n], F [0 .. n, 0 .. n]
    para j desde 0 até n fazer {
        c [j, j] = 0
        F [j, j] = f ' [j]
    }
    para d desde 1 até n fazer
        para i desde 0 até n - d fazer {
            j = i + d
            F [i, j] = F [i, j - 1] + f [j] + f ' [j]
            tmp = inf
            para k desde i + 1 até j fazer
                tmp = min(tmp, c [i, k - 1] + c [k, j])
            c [i, j] = tmp + F [i, j]
        }
    }
```

Árvore de Busca Ótima para Freqüências de Acesso Dadas

- ▶ O algoritmo para computar o custo da árvore ótima tem complexidade $O(n^3)$
- ▶ O algoritmo para criar a árvore ótima é trivial, bastando para isso usar como raízes os nós de índice k que minimizam o custo de cada subárvore (pode-se armazenar esses índices numa terceira matriz)
- ▶ É possível obter um algoritmo de complexidade $O(n^2)$ utilizando a propriedade de monotonicidade das árvores binárias de busca
 - ▶ Se s_k é a raiz da árvore ótima para o conjunto $\{s_i \dots s_j\}$ então a raiz da árvore ótima para o conjunto $\{s_i \dots s_j, s_{j+1}\}$ é s_q para algum $q \geq k$
 - ▶ (Analogamente, $q \leq k$ para $\{s_{i-1}, s_i \dots s_j\}$)