

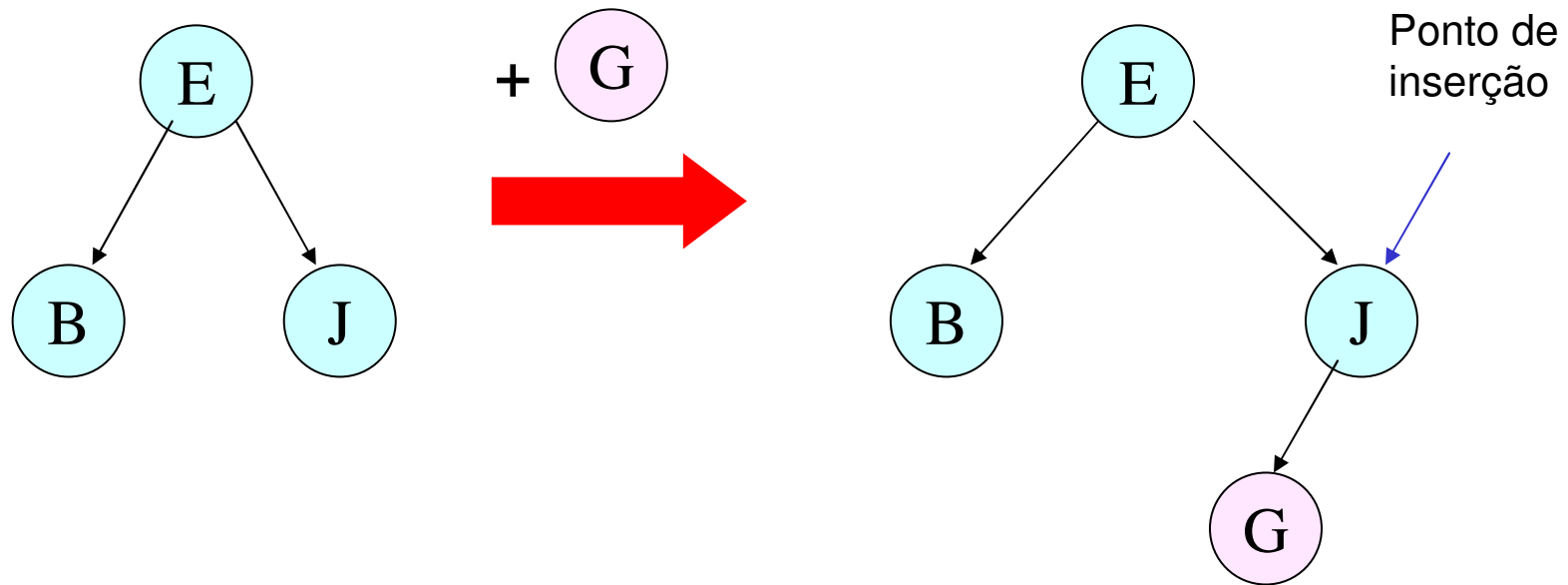
Árvores Binárias de Busca

Fernando Vanini

IC – Unicamp

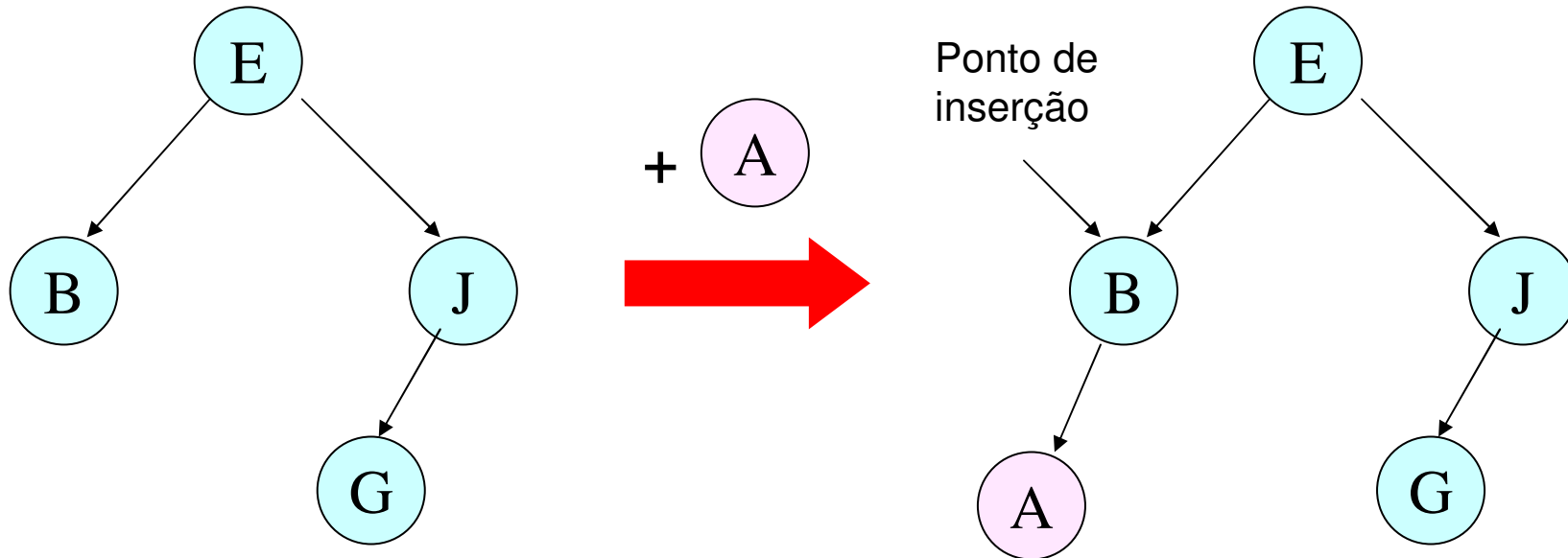
Klais Soluções

Inserção de um novo nó

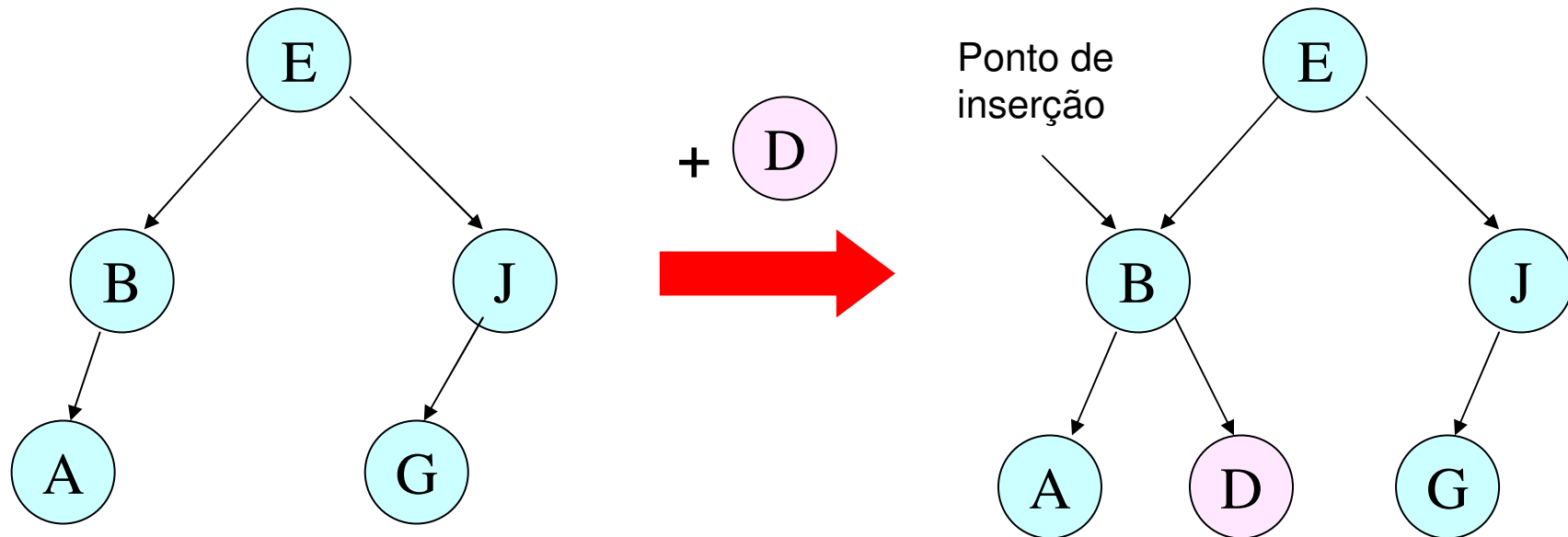


- A localização do 'ponto de inserção' é semelhante à busca por um valor na árvore.
- Após a inserção do novo elemento, a árvore deve manter as propriedades de 'árvore binária de busca'.
- O nó inserido é sempre uma folha.

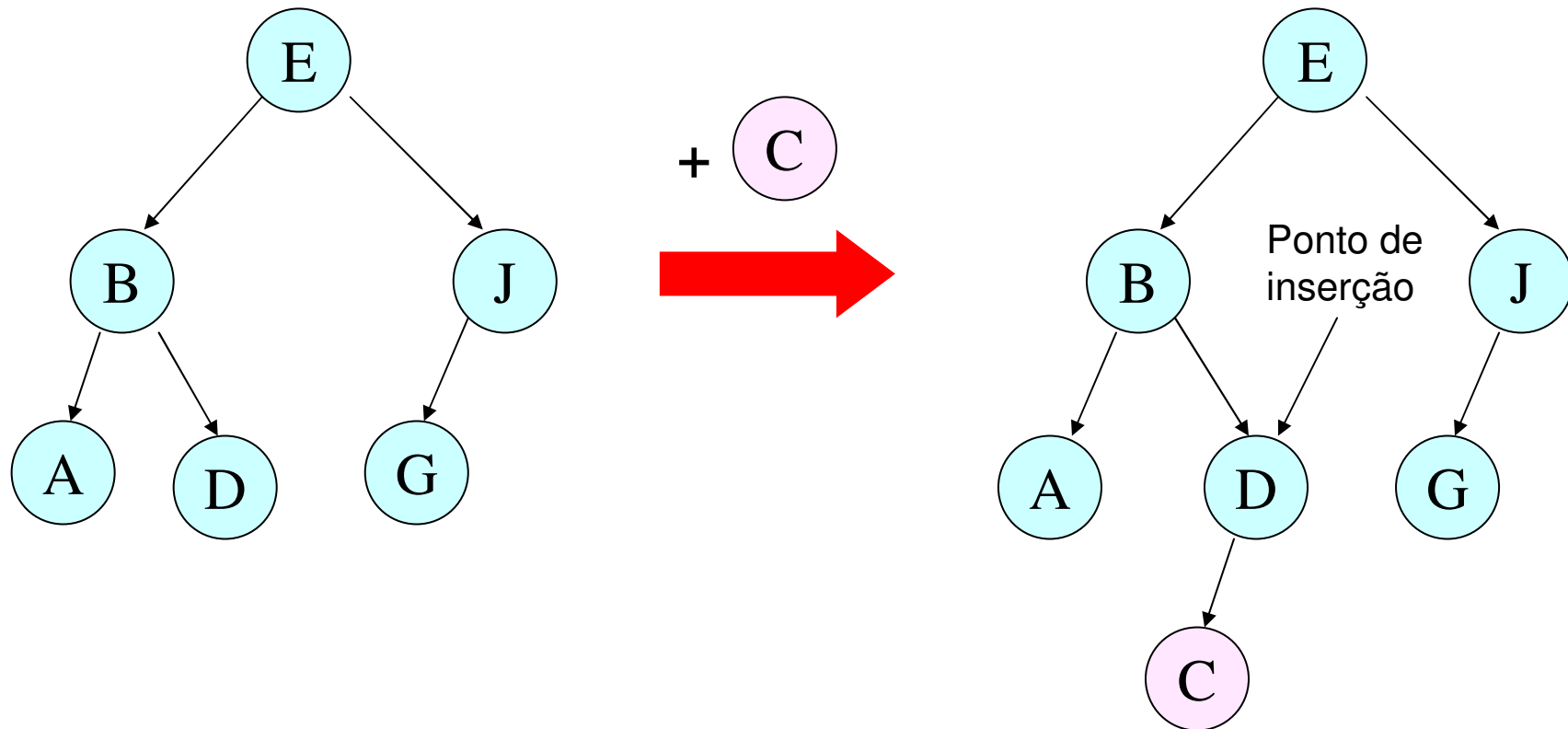
Insertão de um novo nó



Inserção de um novo nó

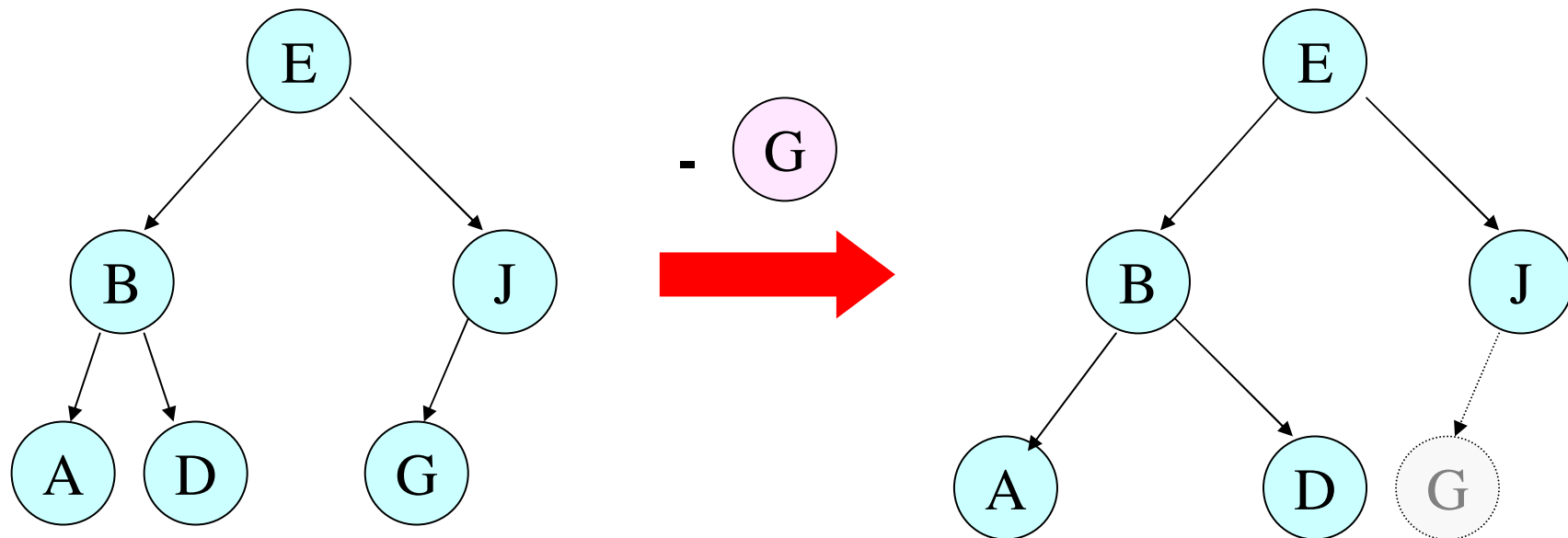


Insertão de um novo nó



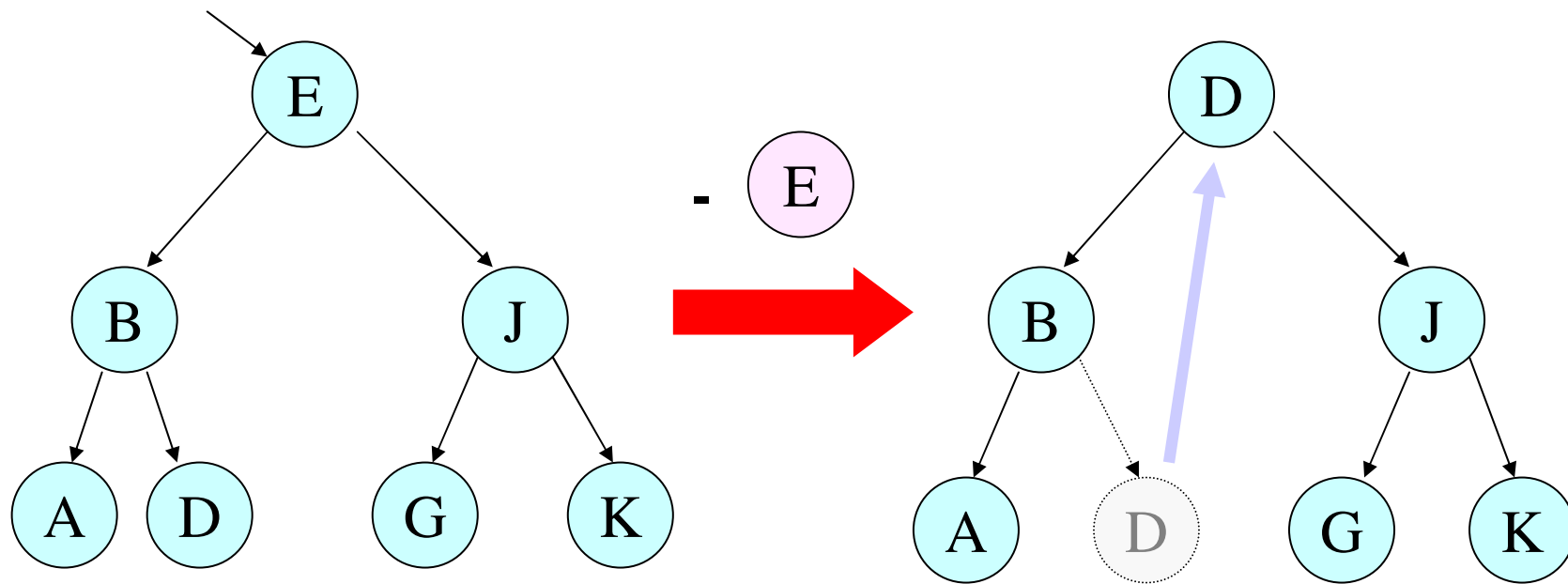
Remoção de um nó

- Caso 1 – O nó a ser removido é uma folha



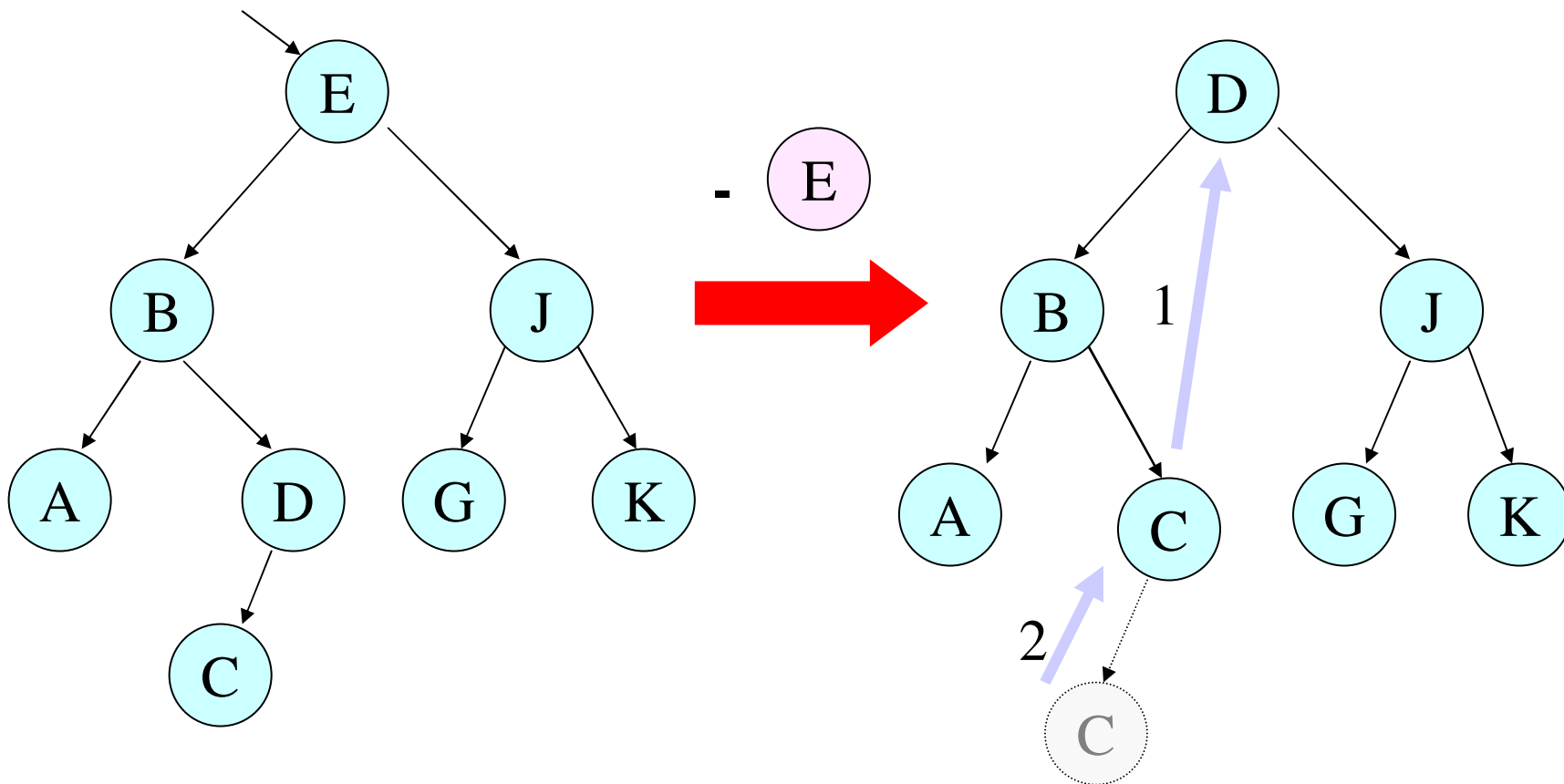
Remoção de um nó

- Caso 2 – O nó X a ser removido não é uma folha
 - 2a. esqDir = nó mais à direita da sub-arvore esquerda;
conteúdo do nó X = conteúdo de esqDir;
remover (recursivamente) o nó esqDir;



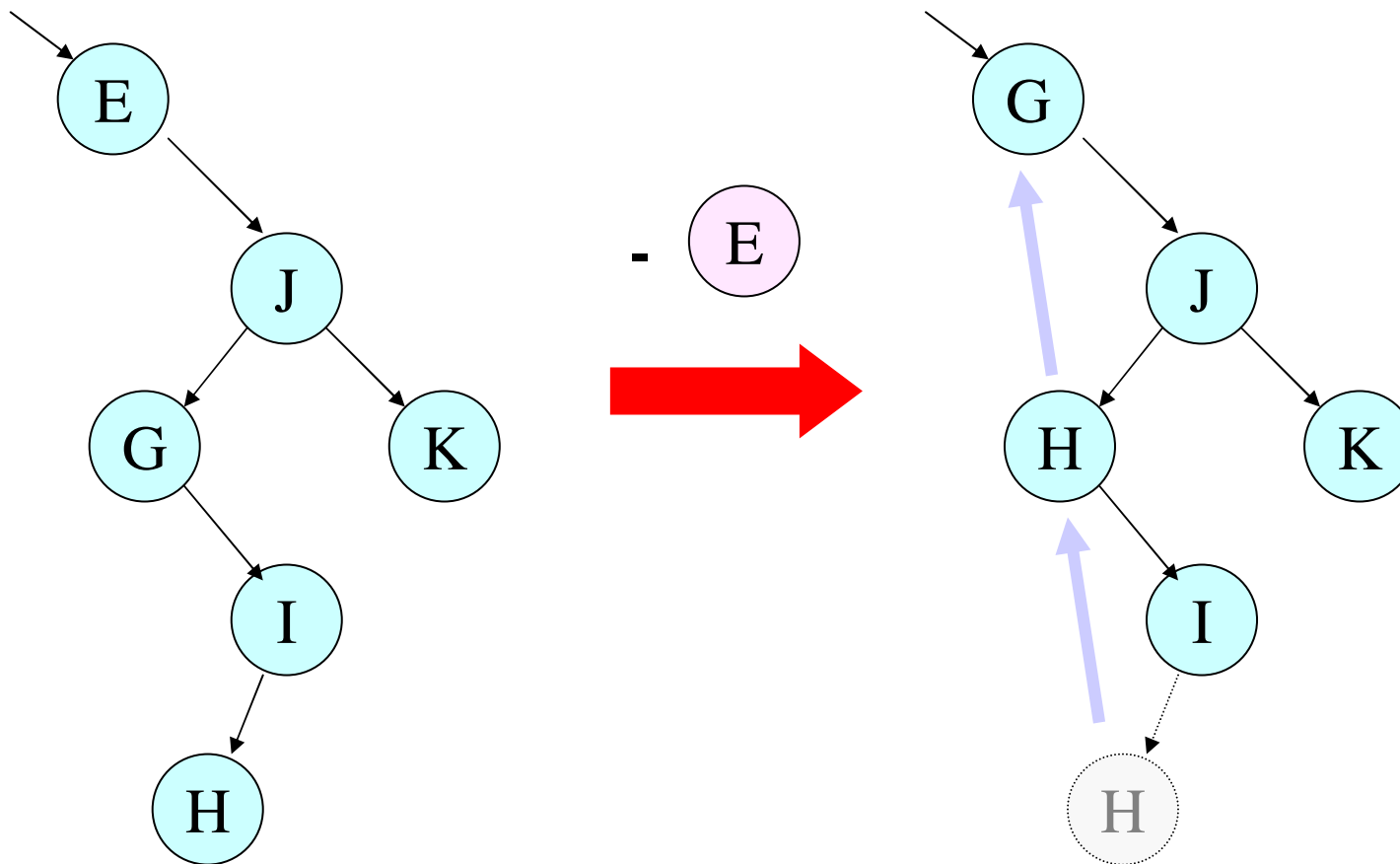
Remoção de um nó

- 2a. esqDir = nó mais à direita da sub-arvore esquerda;
conteúdo do nó X = conteúdo de esqDir;
remover (recursivamente) o nó esqDir;



Remoção de um nó

- 2b. dirEsq = nó mais à esquerda da sub-arvore direita;
conteúdo do nó X = conteúdo de dirEsq;
remover (recursivamente) o nó dirEsq;



Árvores de busca balanceadas

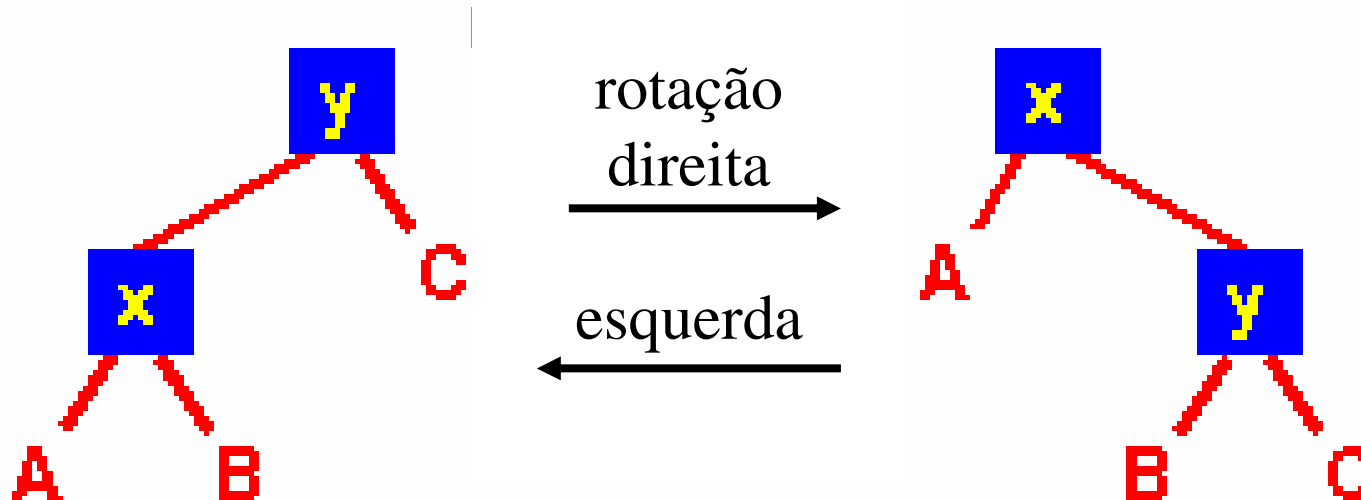
- Árvore binária balanceada: uma árvore binária é dita balanceada se:
 - o número de nós das sub-árvores esquerda difere no máximo de 1 do número de nós da sub-árvore direita.
 - os sub-árvores esquerda e direita são balanceadas.
- Propriedade: a altura de uma árvore binária balanceada com n nós é igual a $\lceil \log_2 n + 1 \rceil$

Árvores AVL

- Árvore binária de busca auto balanceável
- Autores: Adelson-Vesky e Landis (1962)
- Idéia básica: cada nó mantém uma informação adicional, chamada fator de balanceamento, que indica a diferença de altura entre as sub-árvores esquerda e direita.
- As operações de inserção e remoção mantêm o fator de balanceamento entre -1 e +1.
- Embora não sejam perfeitamente balanceadas, é possível demonstrar que a altura de uma árvore AVL ainda é proporcional a $\log_2 n$ ($\sim 1.5 * \log_2 n$)

Árvores AVL

- Nas operações de inserção e remoção de elementos, o balanceamento da árvore resultante é ajustado através da operação de rotação, que preserva a ordenação da árvore.



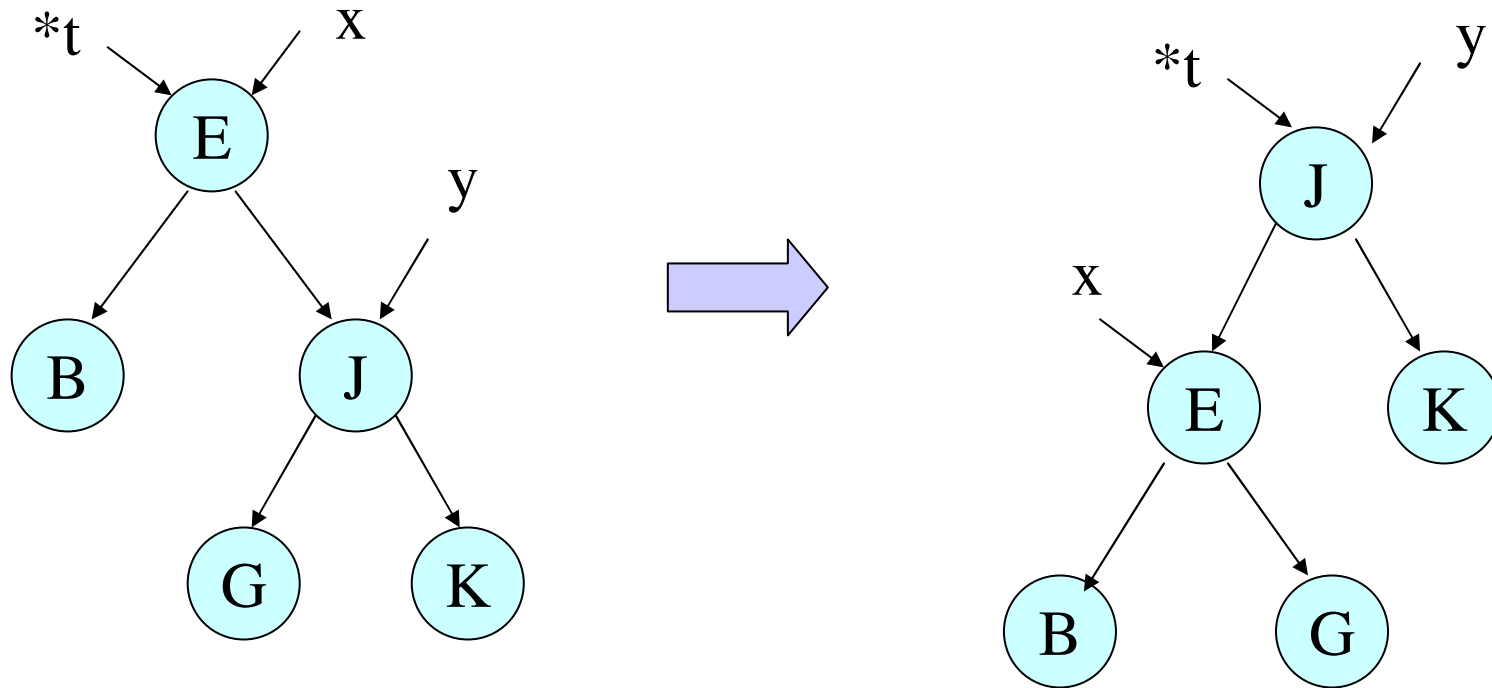
Árvores AVL

- As operações de inserção e remoção de nós numa árvore AVL, incluindo as eventuais operações de rotação, são realizadas em tempo proporcional a $\log_2 n$.

Rotação à esquerda

```
void rotacaoEsquerda(tnodeptr * t){
    tnodeptr x = (*t);          /* != NULL */
    tnodeptr y = x->dir;        /* != NULL */
    x->dir = y->esq;
    if(y->esq != NULL) (y->esq)->pai = x;
    y->pai = x->pai;
    if(x->pai == NULL) *t = y;
    else if(x == (x->pai)->esq) (x->pai)->esq = y;
        else (x->pai)->dir = y;
    y->esq = x;
    x->pai = y;
}
```

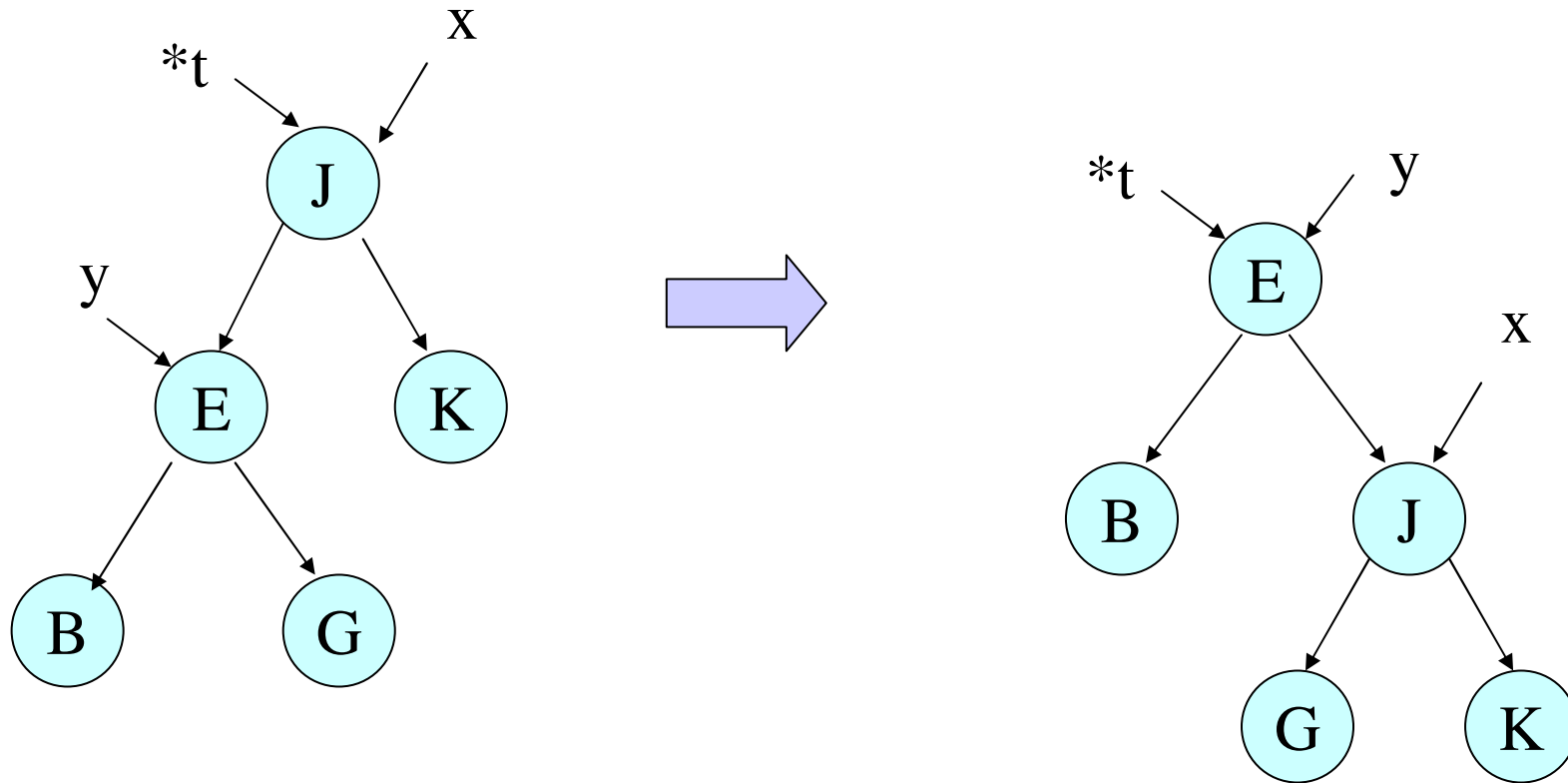
Rotação à esquerda



Rotação à direita

```
void rotacaoDireita(tnodeptr * t){
    tnodeptr x = (*t);          /* != NULL */
    tnodeptr y = x->esq;        /* != NULL */
    x->esq = y->dir;
    if(y->dir != NULL) (y->dir)->pai = x;
    y->pai = x->pai;
    if(x->pai == NULL) *t = y;
    else if(x == (x->pai)->dir) (x->pai)->dir = y;
        else (x->pai)->esq = y;
    y->dir = x;
    x->pai = y;
}
```

Rotação à direita



Referências na web

- http://en.wikipedia.org/wiki/AVL_tree
- <http://www.site.uottawa.ca/~stan/csi2514/applets/avl/BT.html>
- <http://www.eli.sdsu.edu/courses/fall96/cs660/notes/avl/avl.html>
- <http://www.brpreiss.com/books/opus4/html/page320.html>

Árvores rubro-negras

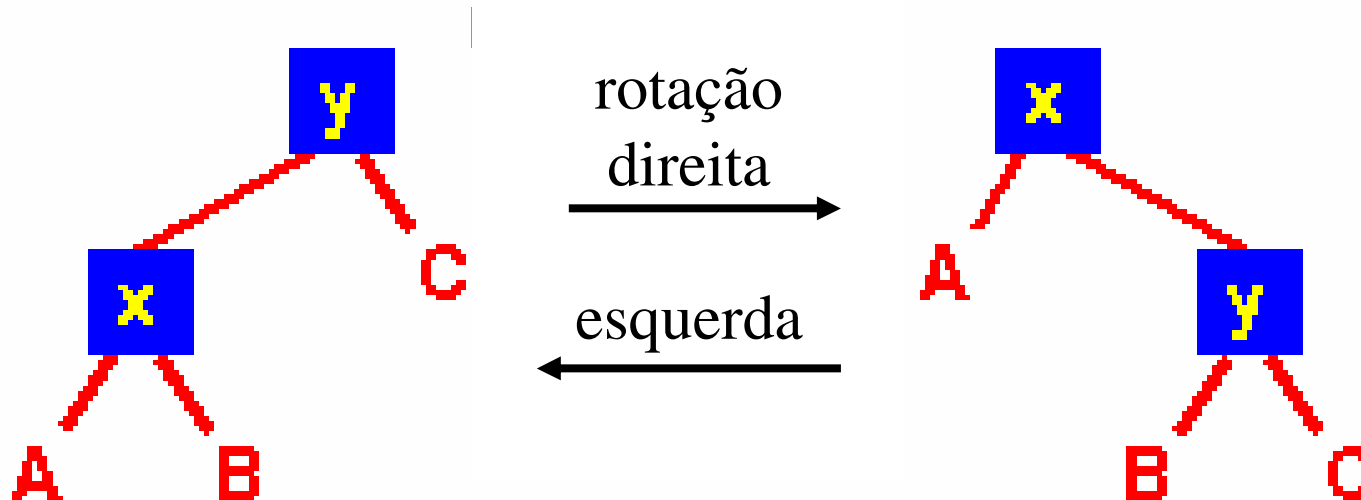
- Propriedades:
 1. Todo nó é vermelho ou preto.
 2. Toda folha é preta.
 3. Se um nó é vermelho então seus filhos são pretos.
 4. Todo caminho da raiz até qualquer folha tem sempre o mesmo número de nós pretos.

Características

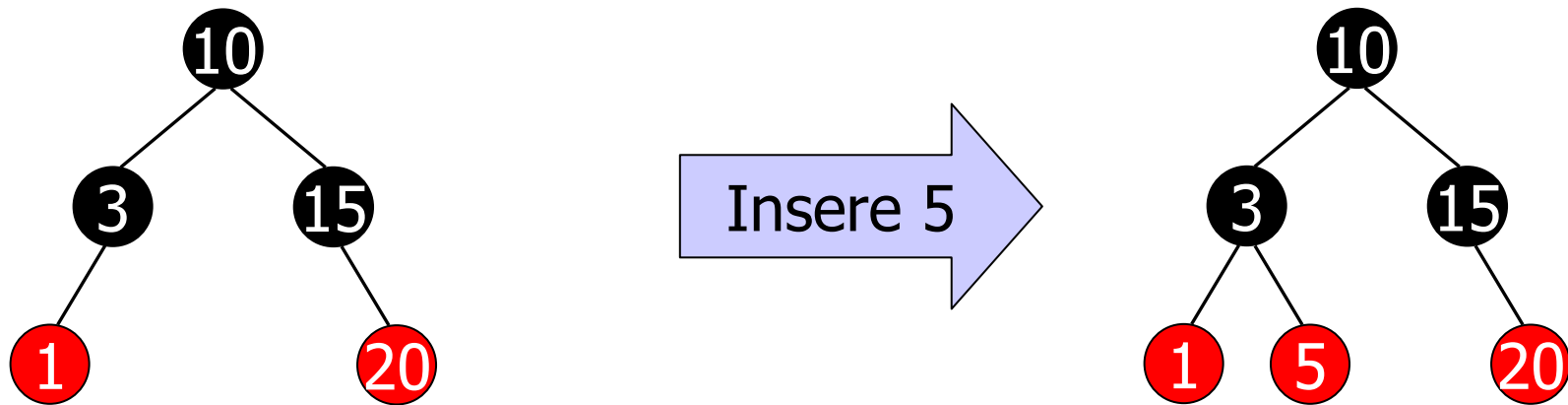
- Uma árvore rubro-negra com n nós tem altura menor ou igual a $2\log(n+1)$.
(*Cormen*, p 264)
- Uma busca numa árvore leva um tempo $O(\log n)$.
- Inserções e retiradas podem, se feitas como nas árvores binárias de busca ‘normais’ podem destruir as propriedades ‘rubro-negras’.
- Para restabelecer as propriedades, recorre-se a rotação e recoloração dos nós

Rotação

- rotação à esquerda ou à direita – essa operação preserva a ordenação da árvore.

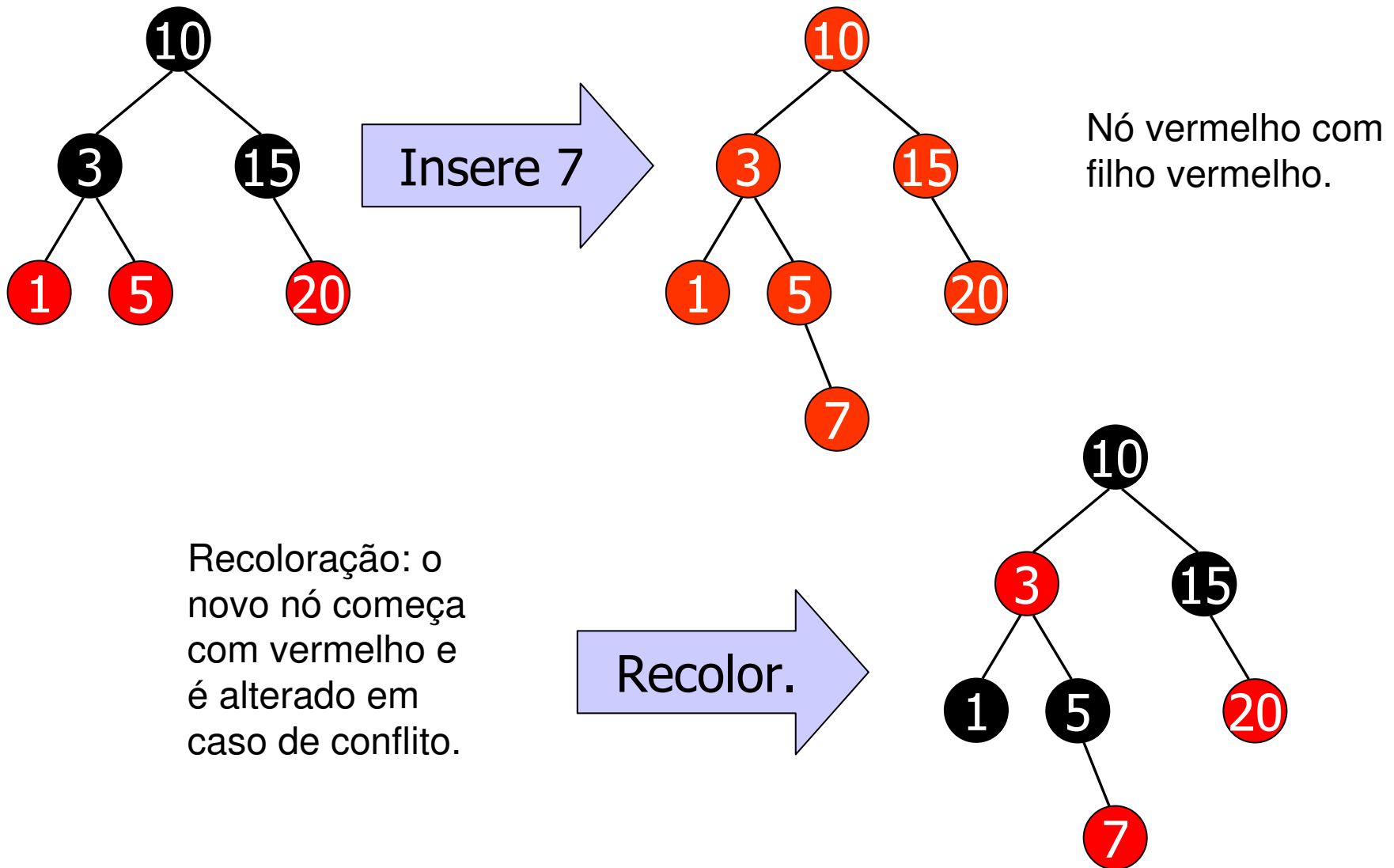


Inserção - exemplo

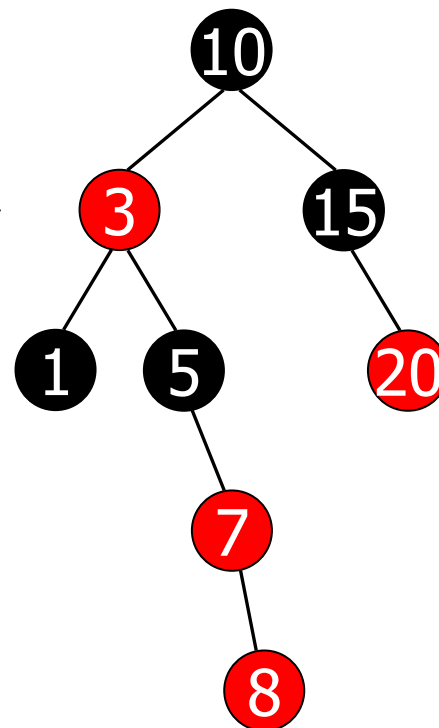
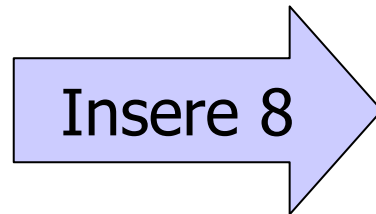
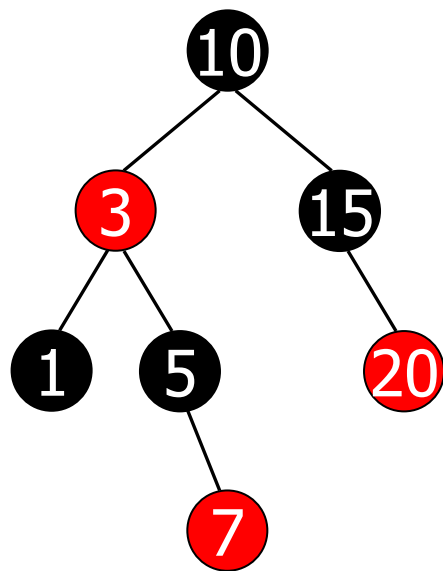


A cor é definida pelas propriedades.

Inserção - exemplo

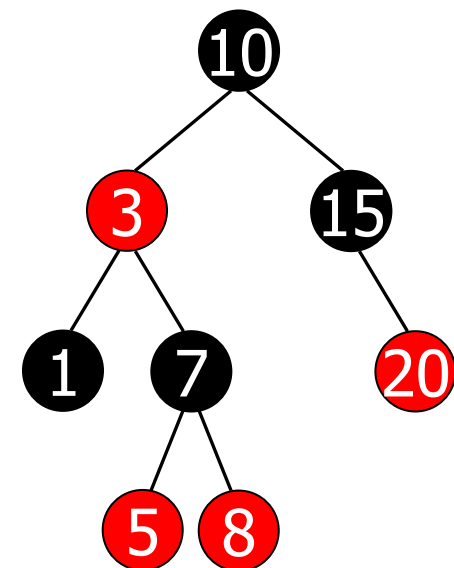
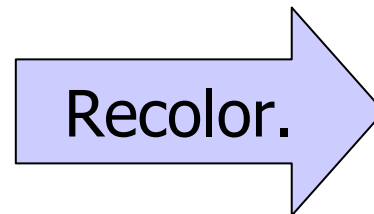


Inserção - Exemplo

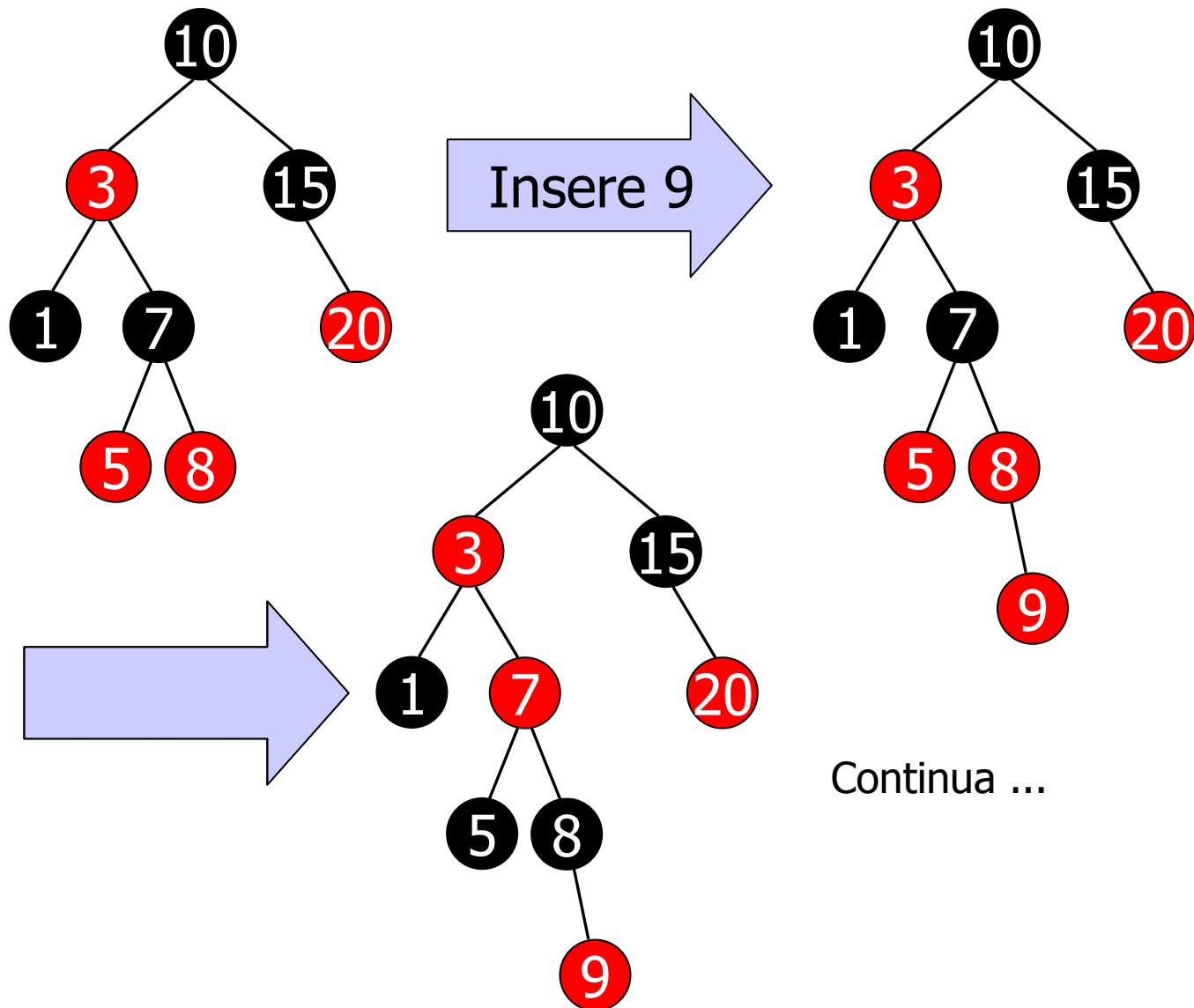


A recoloração anterior não funciona

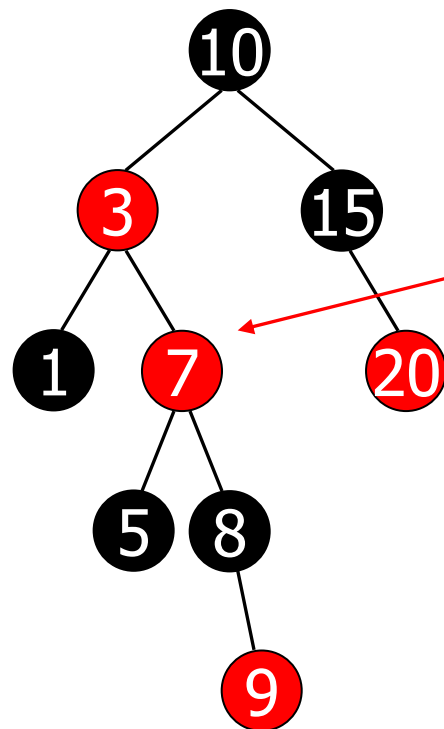
Pai -> preto.
Avô -> vermelho.
Rotação no avô.



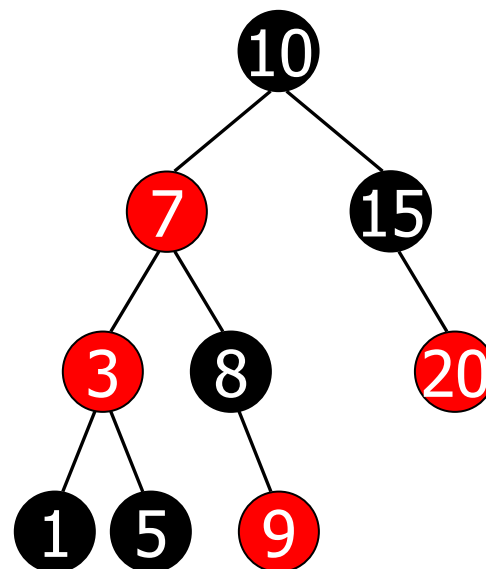
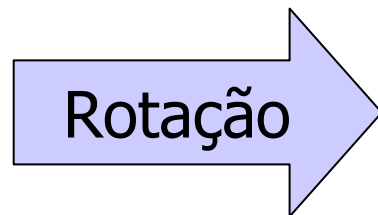
Inserção - exemplo



Inserção (cont.)

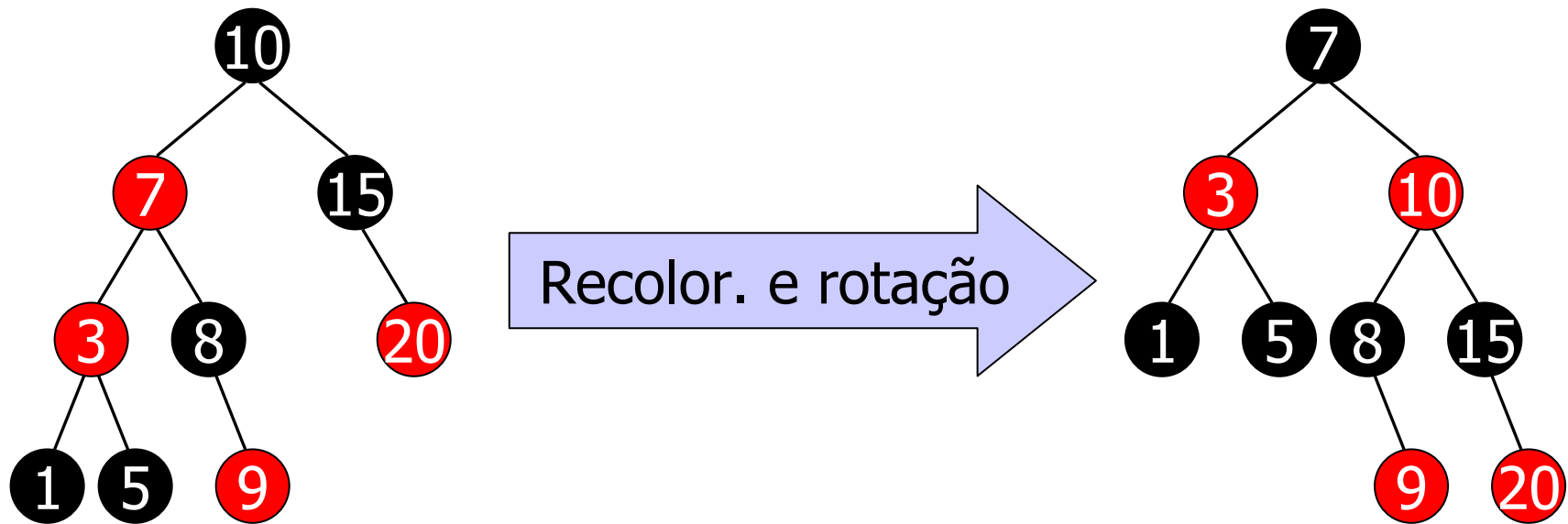


É necessário recolorir
o pai e o avô.



O problema ainda não
foi eliminado ...

Insertão (cont)



Referências na web

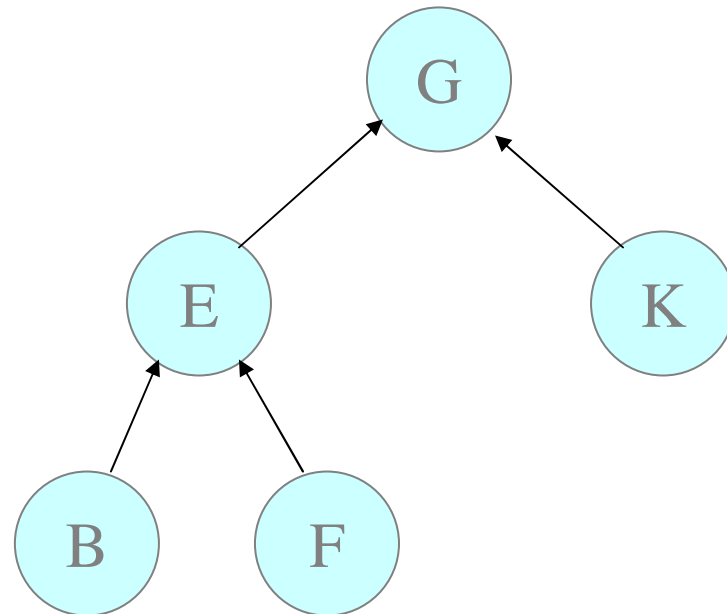
- <http://gauss.eecs.uc.edu/RedBlack/redblack.html>
- <http://ww3.algorithmdesign.net/handouts/RedBlackTrees.pdf>
- <http://www.cs.buap.mx/~titab/files/RedBlackTrees.pdf>
- <http://www.cs.dal.ca/~nzeh/Teaching/Fall%202003/3110/RedBlackTrees.pdf>
- http://en.wikipedia.org/wiki/Red-black_tree

Árvores:

Outras representações

- Dependendo da aplicação, pode ser necessário usar outra representação para árvores.
- Exemplo:

```
typedef struct no* apno;  
typedef struct no {  
    char* info;  
    apno pai  
}
```



Um exemplo:

classes de equivalência (I)

- inicialmente: `p->pai = p`; para todo elemento `p`.
- raiz:

```
apno raiz(apno p) {  
    apno r = p;  
    if(r->pai == r) return r;  
    return raiz(r->pai);  
}
```

- `p` e `q` são equivalentes ?

```
bool equiv(apno p, apno q) {  
    return(raiz(p) == raiz(q));  
}
```

Um exemplo:

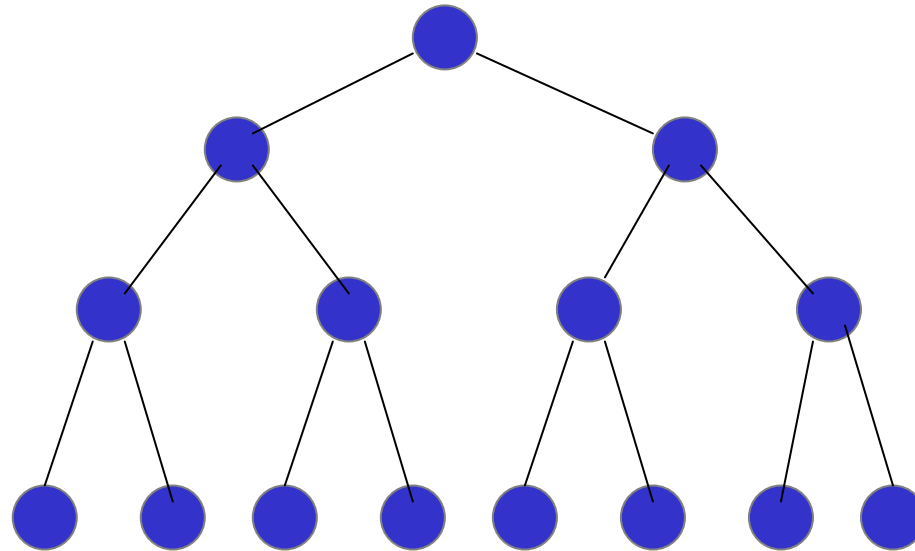
classes de equivalência (I)

- fazer p e q equivalentes:

```
void mkEquiv(apno p, apno q) {  
    apno r = raiz(p);  
    r->pai = raiz(q);  
}
```

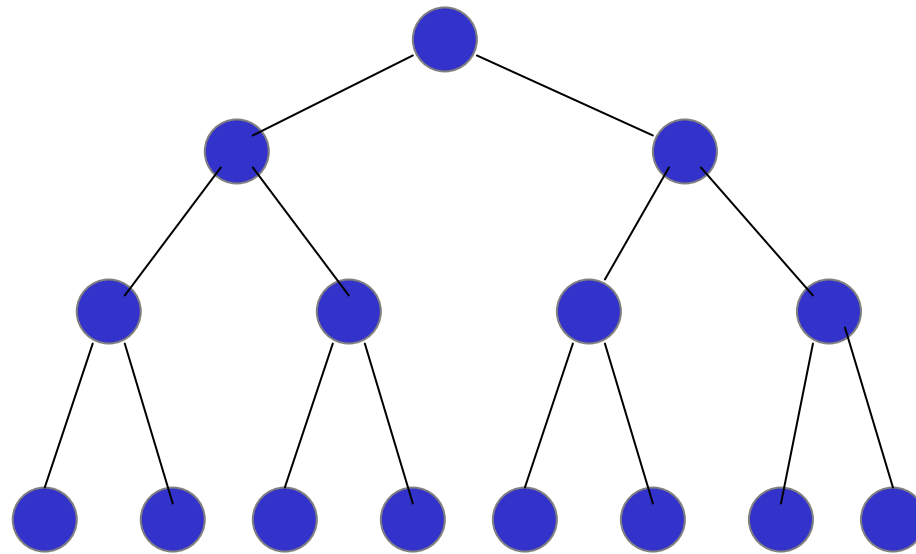
Árvore binária completa

- Uma árvore binária de altura h é completa se ela tiver $2^h - 1$ nós.



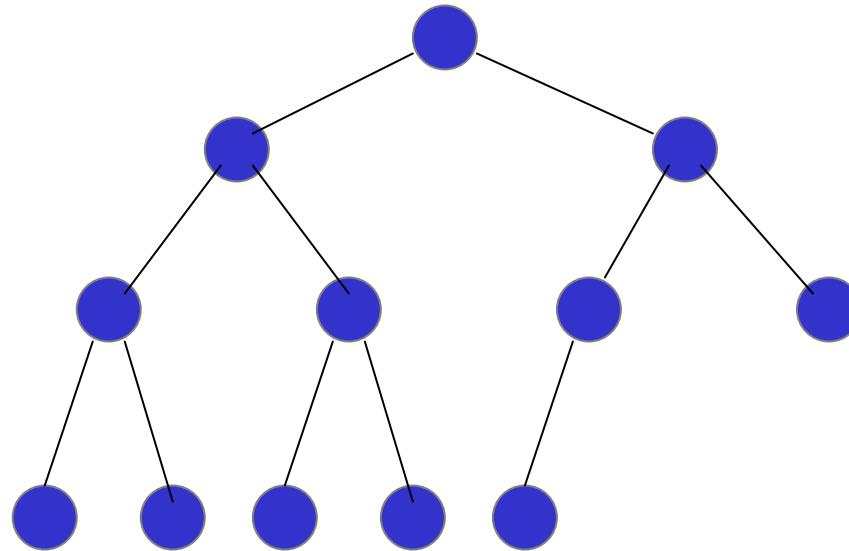
Árvore binária completa

- Uma árvore binária completa pode ser representada num vetor v :
 - raiz em $v[0]$
 - se um nó está em $v[i]$, seus filhos estão em $v[2*i+1]$ e $v[2*i+2]$



Árvore binária quase completa

- Uma árvore binária quase completa tem todos os seus níveis completos exceto o último, o qual tem apenas os elementos mais à esquerda.
- Uma árvore binária quase completa também pode ser representada num vetor.

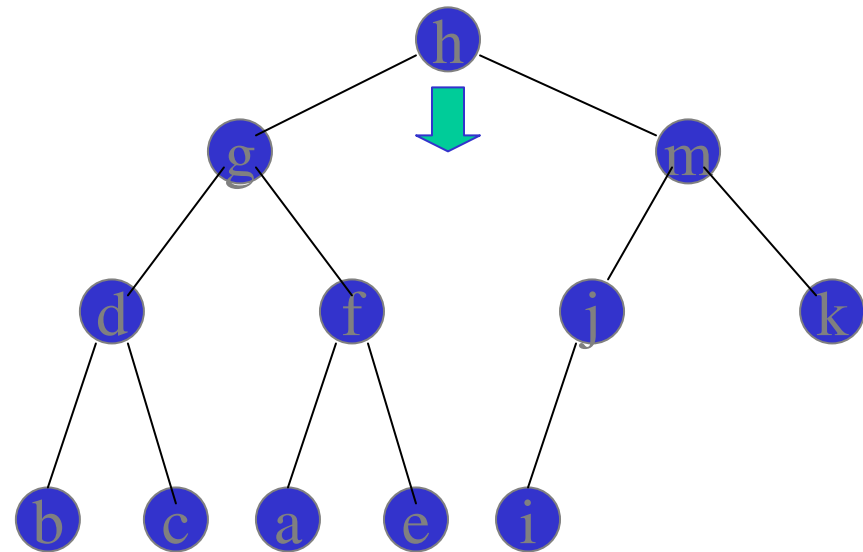


Filas de prioridade

- Uma fila de prioridade é uma árvore binária tal que
 - o valor associado a cada nó é maior (ou menor) que o valor associado a cada um dos seus filhos.
- Implementação:
 - se a fila de prioridade for uma árvore completa ou quase completa, ela pode ser implementada num vetor.

Filas de prioridade

- Rearranjo: o elemento fora de ordem é trocado com o seu maior filho, sucessivamente até restabelecer a 'condição de ordem' na fila de prioridade.
- No exemplo: 'h' seria trocado com 'm' e depois trocado com 'k'.



Filas de prioridade

- Rearranjo 'morro abaixo' : supondo que a fila de prioridade é implementada num vetor v e o valor da raiz está fora de ordem

```
void sift(int r, int m, int v[]) {  
    int x = v[r];  
    while (2*r < m) {  
        int f = 2*r+1;  
        if ((f < m) && (v[f] < v[f+1])) ++f;  
        if (x >= v[f]) break;  
        v[r] = v[f];  
        r = f;  
    }  
    v[r] = x;  
}
```

O trecho do vetor que contém a sub-árvore a ser rearranjada é delimitado por r e m .

Filas de prioridade

- Rearranjo 'morro acima' : a folha $v[m]$ está fora de ordem

```
void upHeap(int r, int m, int v[])
{
    int x = v[m];
    int j = m/2;
    while((j >= r) && (v[j] < x)) {
        v[m] = v[j];
        m = j; j = j/2;
    }
    v[m] = x;
}
```

O trecho do vetor que contém a sub-árvore a ser rearranjada é delimitado por r e m .

Filas de prioridade

- Construção da fila a partir de um vetor em que os elementos não mantém nenhuma relação de ordem:
 - A construção parte 'de baixo para cima' a partir do último elemento da primeira metade do vetor (penúltimo nível) porque o último nível já está organizado (cada sub-árvore só tem um nó).

```
void makePQueue (int n, int v[])
{
    int r;
    for (r = (n-1)/2; r >= 0; r--)
        sift(r, n-1, v);
}
```

- A construção da fila é feita em tempo linear (!)

Filas de prioridade

- Inserir um valor x na fila (supondo que existe espaço no vetor):

```
v[++m] = x;  
upHeap(0, m, v);
```

- Retirar o valor mais prioritário da fila:

```
x = v[0];  
v[0] = v[m--];  
sift(0, m, v);
```

Heapsort

- O algoritmo heapsort usa a fila de prioridade para ordenar um vetor:

```
void heapsort (int n, int v[])
{
    int p, m, x;
    for (p = (n-1)/2; p >= 0; p--)
        sift(p, n-1, v);
    for (m = n-1; m >= 1; m--) {
        x = v[0], v[0] = v[m], v[m] = x;
        sift(0, m-1, v);
    }
}
```

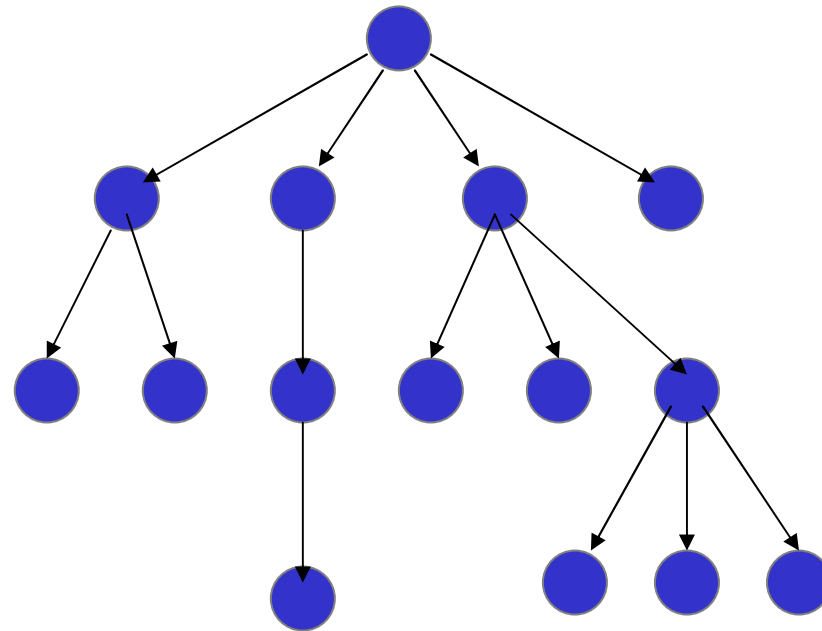
Referência na web

- Uma boa referência na web, em português, sobre estruturas de dados e algoritmos, além de uma introdução à linguagem C, estilo de programação, etc:

<http://www.ime.usp.br/~pf/algoritmos/>

Árvores gerais

- Conjunto T não vazio de objetos
 - um nó raiz
 - demais nós em conjuntos $T_1, \dots, T_m$, árvores disjuntas.



Árvores gerais

- Representação binária

