Árvores de Busca Balanceadas

Fernando Vanini
IC - UNICAMP

Árvores Binárias de Busca

- Operações de busca, inserção e remoção de elementos são simples.
- O tempo dessas operações, no pior caso é linear.
- O tempo mínimo, logarítmico, só ocorre se a árvore estiver perfeitamente balanceada.

Árvores de Busca Balanceadas

- As operações de inserção e remoção de elementos numa árvore de busca balanceada garantem que a árvore se mantém balanceada após a sua realização.
- As técnicas utilizadas nesse sentido em geral envolvem
  - campos adicionais em cada nó, para controle do balanceamento ou
  - árvores de ordem maior que 2

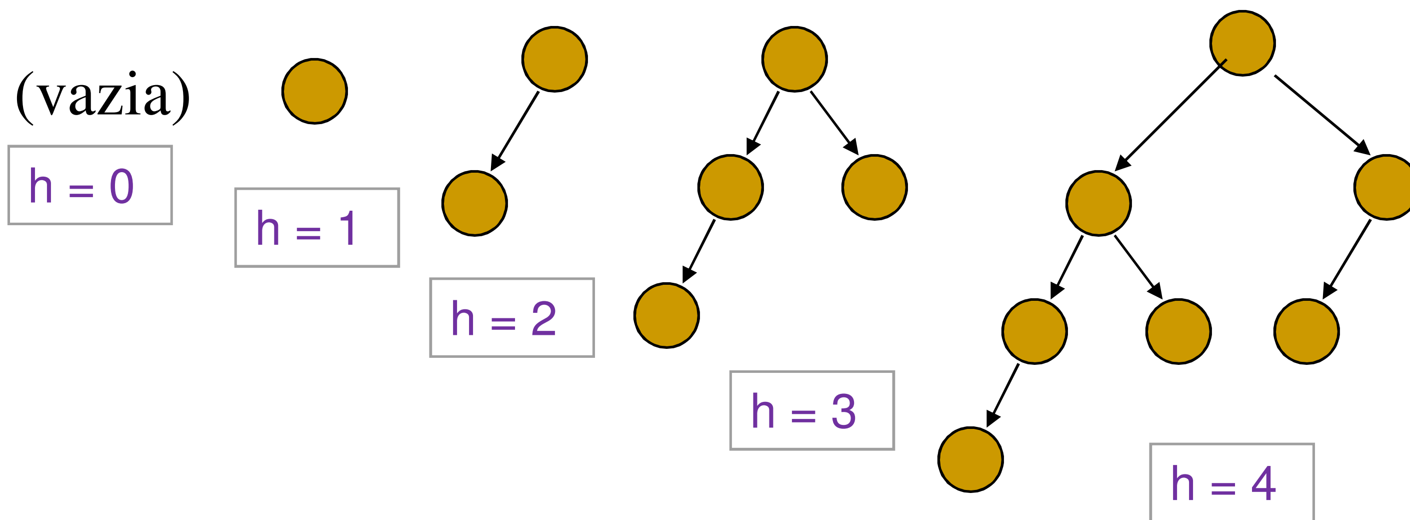
Árvores AVL

- Árvore balanceada tipo AVL:
  - para todos os seus nós, a diferença de altura entre as sub-árvores esquerda e direita é no máximo 1.

  (Adel'son-Vel'Skii e E. M. Landis)

Árvore AVL de altura máxima

- Para uma altura h, a árvore AVL com o número máximo de elementos pode construída de forma sistemática



$N(h) = N(h-1)+N(h-2)+1$

Altura aproximada: $1.44 \log_2(n+2)$

Implementação de árvores AVL

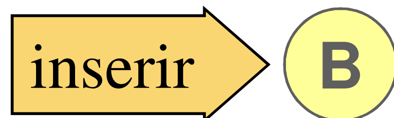
- Cada nó tem associado um *fator de balanceamento* que representa a diferença de altura entre as sub-árvores esquerda e direita.
- O fator de balanceamento pode ser
  - implícito - calculado a cada acesso, o que é muito caro)
  - explícito - cada nó tem um campo a mais indicando o fator de balanceamento (-1, 0 ou +1)

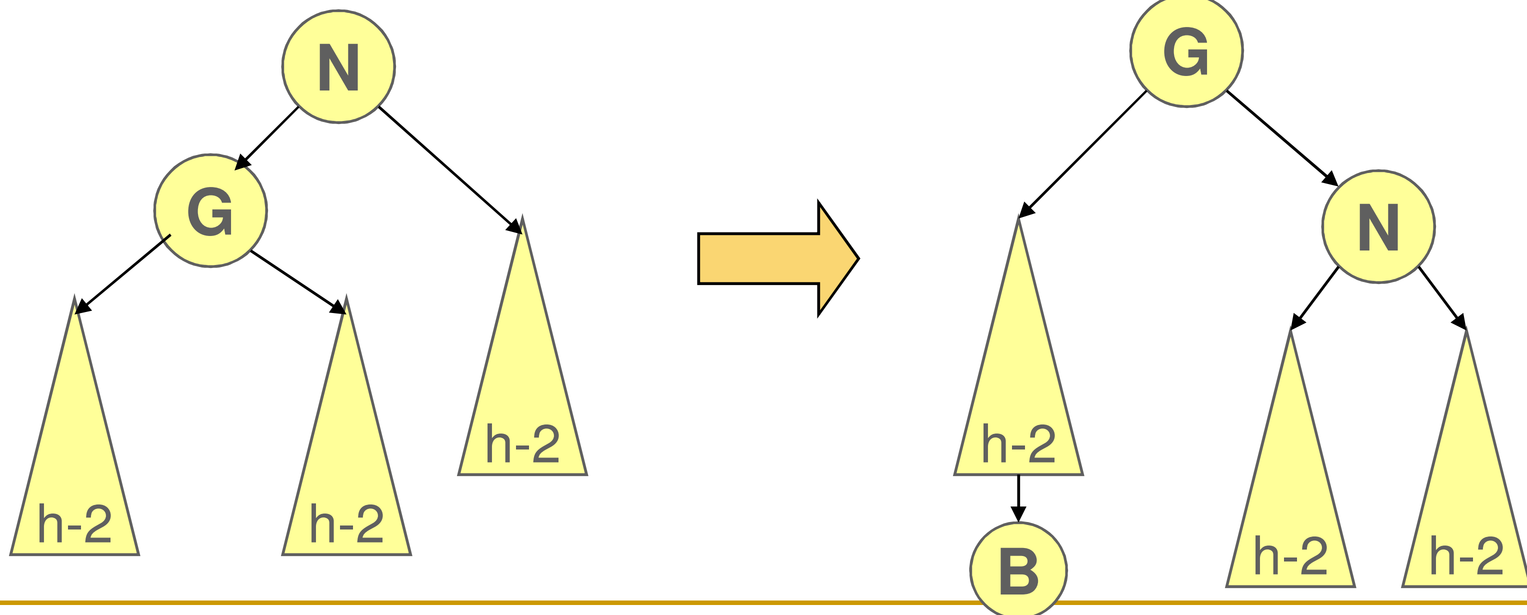
Inserção em árvores AVL

- Árvore vazia: a nova árvore 'nasce' balanceada.
- Inserção do lado mais baixo: a altura final se mantém (dois casos: esquerdo e direito).
- Inserção quando as alturas das sub-árvores são iguais: a altura final aumenta mas a árvore continua satisfazento ao 'critério AVL' (dois casos: esquerdo e direito).

Inserção em árvores AVL

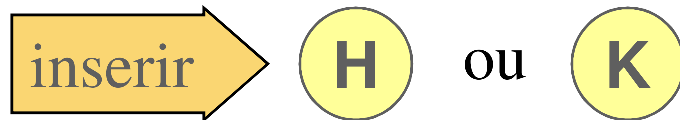
- Inserção do lado mais alto: é necessário um rearranjo da árvore.
- rotação simples:





Inserção em árvores AVL

- rotação dupla:



N
F
I
h-2
h-3
h-3
h-2

I
F
N
h-2
h-3
h-3
h-2
H
K
ou

Implementação da Árvore AVL

```
typedef struct tnode* tnodePtr;
typedef struct tnode{
            tnodePtr esq, dir;
            int fb;
            char * info;
          } tnode;
```

Rotação (simples) à Esquerda

```
tnodePtr LL(tnodePtr b){
  tnodePtr a = b->esq;
  tnodePtr t2 = a->dir;
  a->dir = b;
  b->esq = t2;
  return a;
}
```

Rotação (simples) à Direita

```
tnodePtr RR(tnodePtr b){
  tnodePtr a = b->dir;
  tnodePtr t2 = a->esq;
  a->esq = b;
  b->dir = t2;
  return a;
}
```

Rotação Dupla Esquerda-Direita

```
tnodePtr LR(tnodePtr c){
  tnodePtr b = c->esq;
  tnodePtr a = b->dir;
  tnodePtr t2 = a->dir;
  tnodePtr t3 = a->esq;
  c->esq = t2;
  b->dir = t3;
  a->dir = c;
  a->esq = b;
  return a;
}
```

Rotação Dupla Direita-Esquerda

```
tnodePtr RL(tnodePtr c){
  tnodePtr b = c->dir;
  tnodePtr a = b->esq;
  tnodePtr t2 = a->esq;
  tnodePtr t3 = a->dir;
  c->dir = t2;
  b->esq = t3;
  a->esq = c;
  a->dir = b;
  return a;
}
```

Inserção na Árvore AVL

```
/* devolve true se a altura da árvore aumentou */
int insereAVL(tnodePtr *p, char* k){
  int cmp;
  if(*p == NULL) {
     *p = newnode(k,NULL,NULL);
     return true;
  }
  if(cmp = strcmp(k,(*p)->info)==0) return false;
  if(cmp < 0) {
               <<< INSERIR À ESQUERDA >>>
  } else {
               <<< INSERIR À DIREITA >>>
  }
}
```

Inserção à Esquerda

```
if(insereAVL(&((*p)->esq),k)){
   switch ((*p)->fb){
     case  1: (*p)->fb =  0;  return false;
     case  0: (*p)->fb = -1;  return true;
     case -1:
             /*** rebalancear ***/
             if(((*p)->esq)->fb == -1) *p = LL(*p);
             else {
                   LR(*p); (*p)->fb = 0;
                   << ajustar fb >>
                  }
             return false;
     }
  }
```

Inserção à Direita

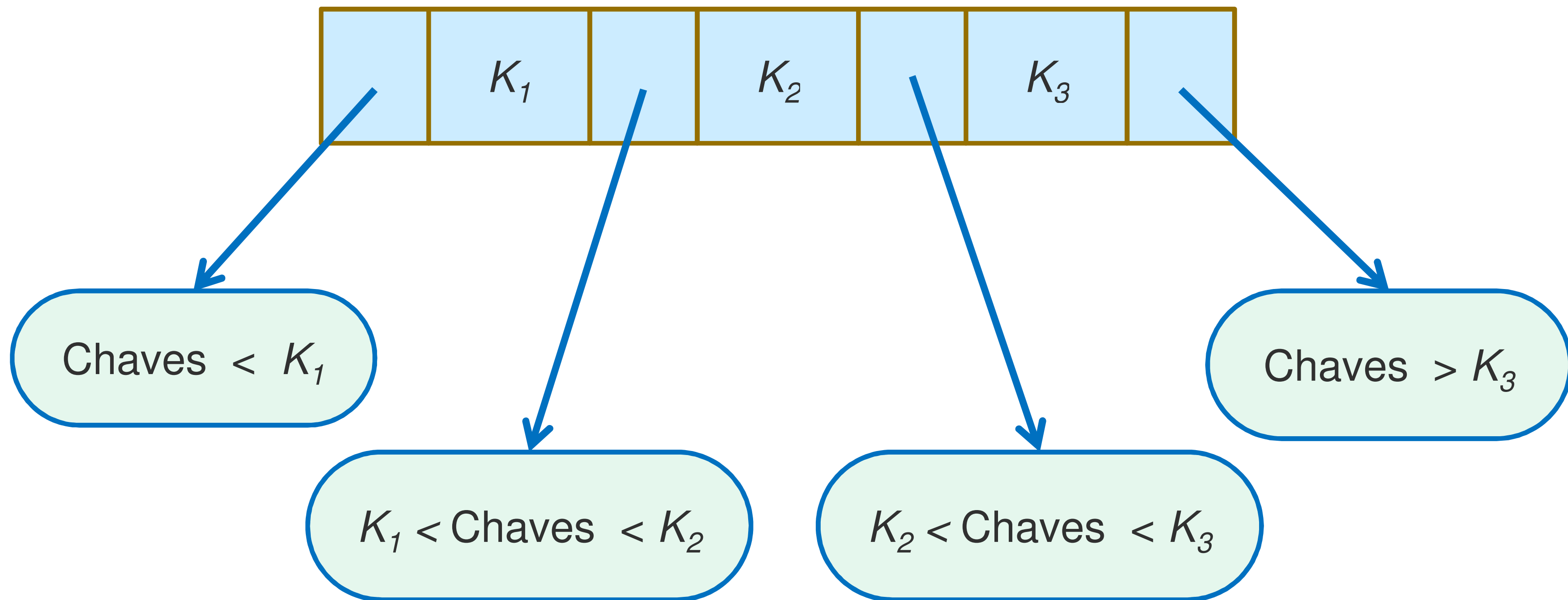
```
if(insereAVL(&((*p)->dir),k)){
   switch ((*p)->fb){
     case -1: (*p)->fb =  0;  return false;
     case  0: (*p)->fb =  1;  return true;
     case  1:
              /*** rebalancear ***/
              if(((*p)->dir)->fb == 1) *p = RR(*p);
              else {
                     (*p)->fb = 0;
                     << ajustar fb >>
                   }
              return false;
      }
   }
```

Referências na WEB

- AVL Tree algorithm
- Wikipedia
- AlgorithmDesign.net
- Animação de árvores AVL
- Animação de Árvores AVL (2)

Árvores de Busca de Ordem > 2

- Cada nó pode ter mais que um campo ‘chave’



Árvores 2-3-4

- Árvore de busca de ordem 4
  - Cada nó *interno* pode ter 2, 3 ou 4 filhos (1, 2 ou 3 campos *chave*).
  - Todas as *folhas* têm a mesma altura.
  - animação

Inserção em Árvores 2-3-4

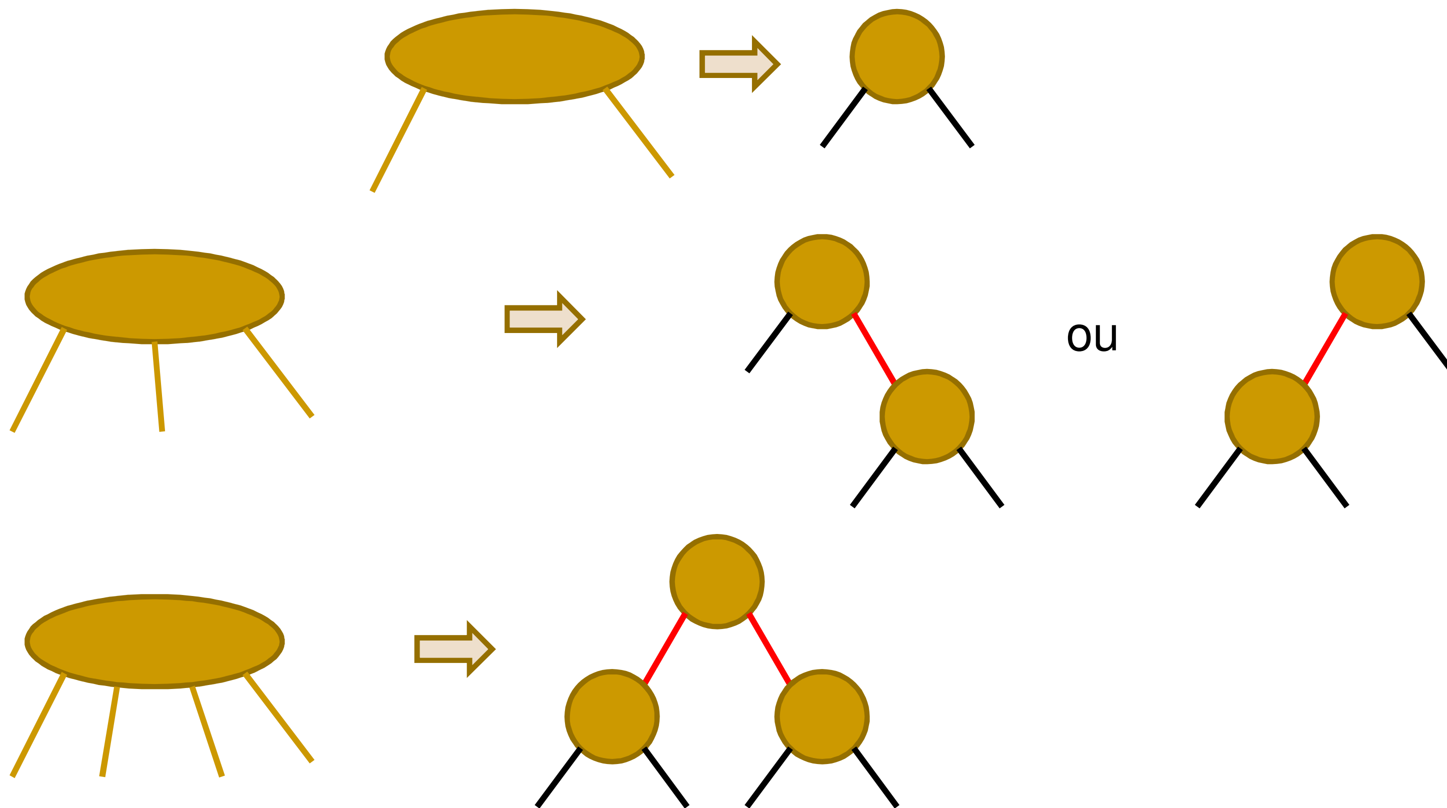
- As operações de inserção e remoção em árvores 2-3-4 devem respeitar as restrições quanto à altura das folhas e número de filhos por nó.
  - Se o nó no 'ponto de inserção' tiver menos que 4 filhos, inserir o novo nó nesse ponto, fazendo uma eventual redistribuição dos filhos.
  - Se o nó no 'ponto de inserção' tiver 4 filhos, 'quebrar' esse no em 2 abrindo espaço para o novo no. A 'quebra' implica numa inserção um nível acima. Essa 'quebra' nível acima se propaga enquanto necessário até a raiz.

Árvores Rubro-Negras

- As árvores 2-3-4 têm algumas desvantagens em conseqüência do número de filhos por nó:
  - Nas inserções e retiradas, a complexidade dos rearranjos do tipo ‘quebra’ e ‘junção’.
  - Mais comparações ‘na horizontal’
- Arvores rubro-negras:
  - Implementam a mesma idéia das árvores 2-3-4 usando a representação de árvores binárias.

Árvores Rubro-Negras

- A idéia original é baseada nas árvores 2-3-4



Árvores rubro-negras

- Propriedades:

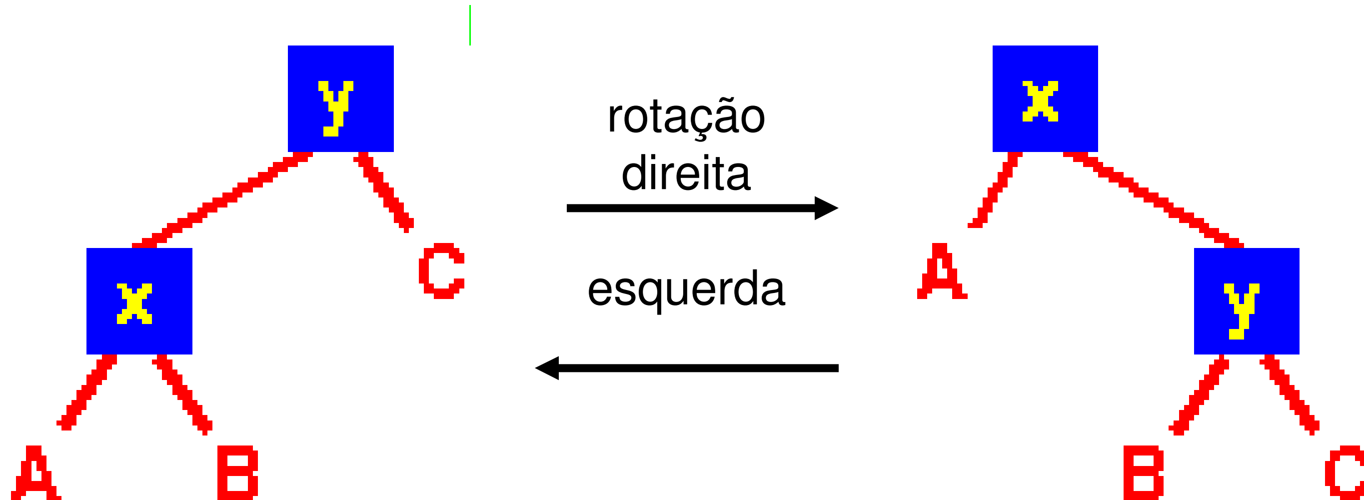
1. Todo nó é vermelho ou preto e a raiz é preta.
2. Toda folha é preta.
3. Se um nó é vermelho então seus filhos são pretos.
4. Todo caminho da raiz até qualquer folha tem sempre o mesmo número de nós pretos (mesma *profundidade preta*).

Características

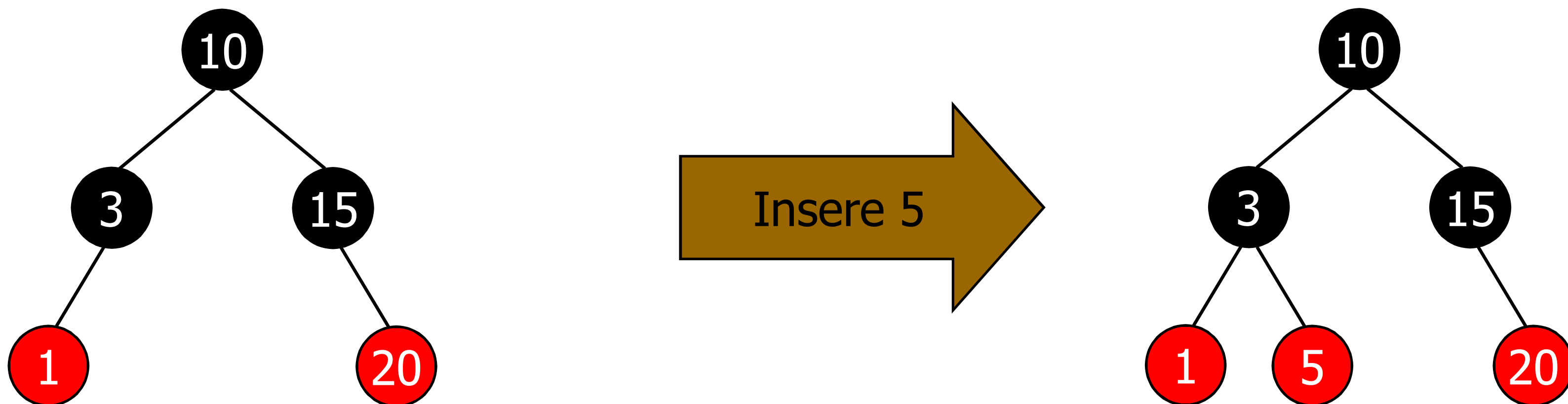
- Uma árvore rubro-negra com n nós tem altura menor ou igual a 2**log(*n*+1)**.
- Uma busca numa árvore leva um tempo **O(log n)**.
- Inserções e retiradas podem, se feitas como nas árvores binárias de busca ‘normais’ podem destruir as propriedades ‘rubro-negras’.
- Para restabelecer as propriedades,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

Rotação

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

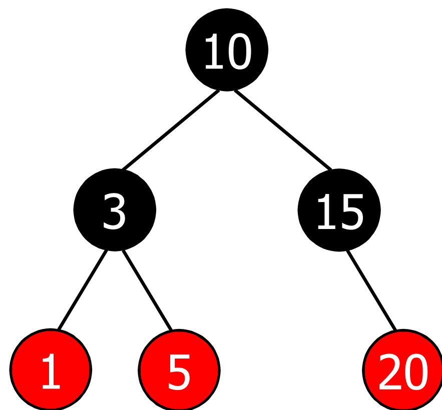


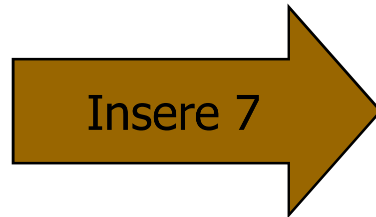
Inserção - exemplo

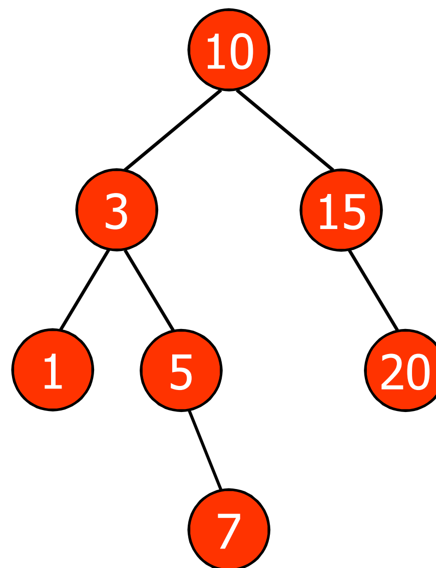


A cor é definida pelas propridades.

Inserção - exemplo

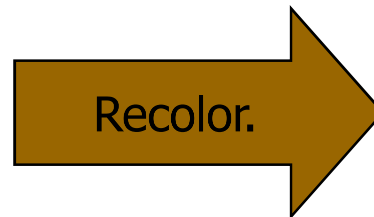


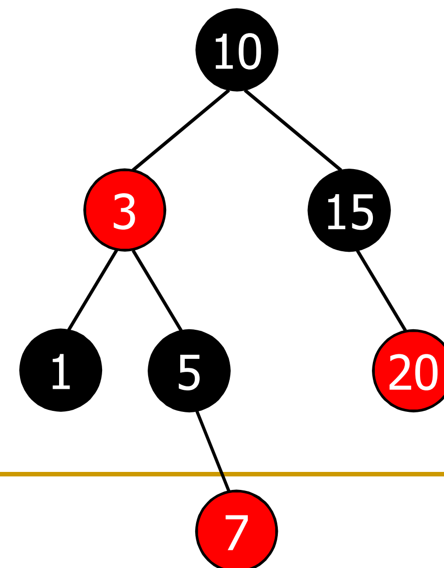




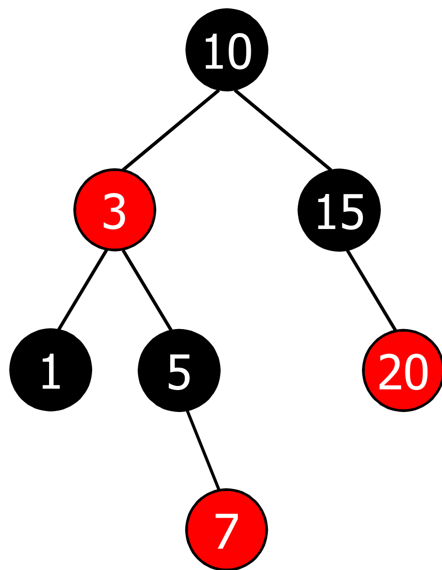
Nó vermelho com filho vermelho.

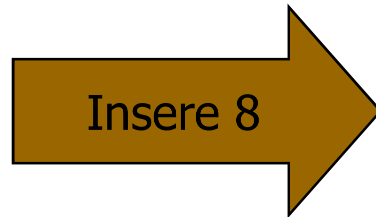
Recoloração: o novo nó começa com vermelho e é alterado em caso de conflito.

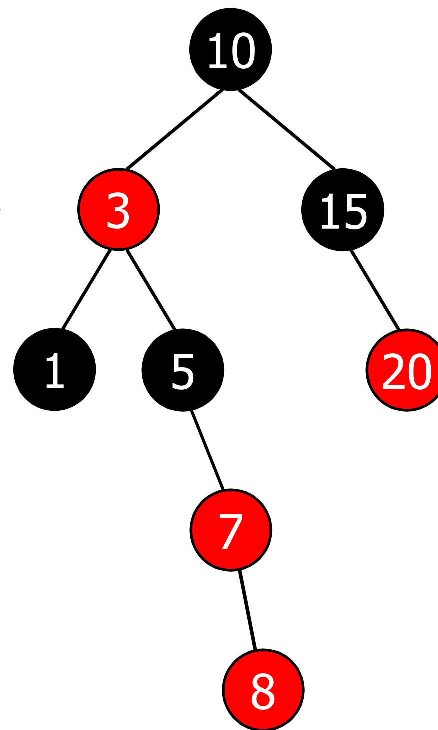




Inserção - Exemplo

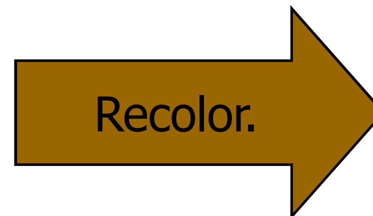


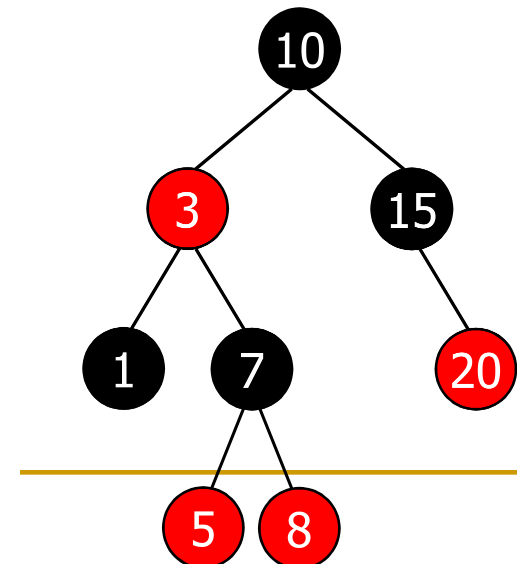




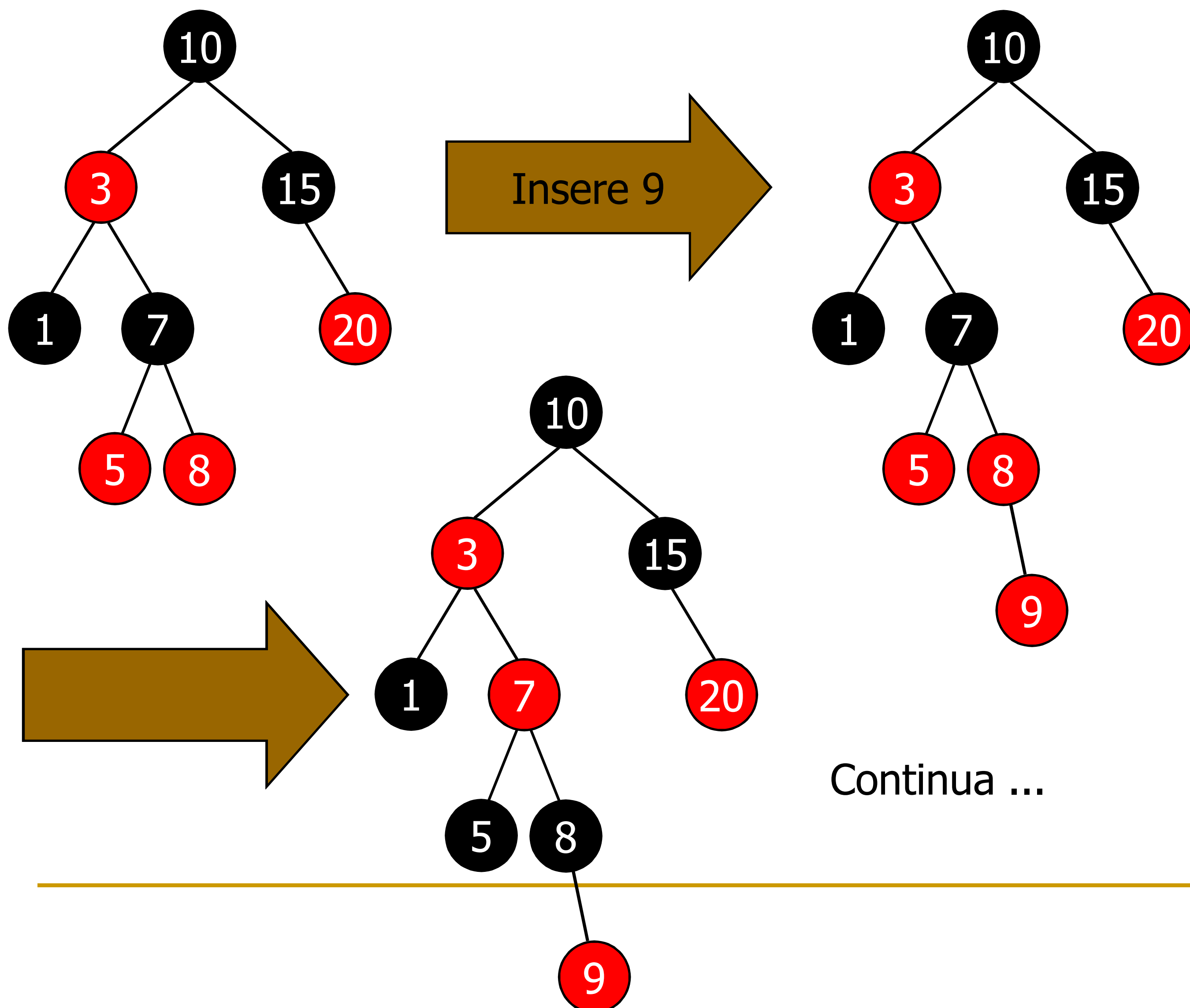
A recoloração anterior não funciona

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



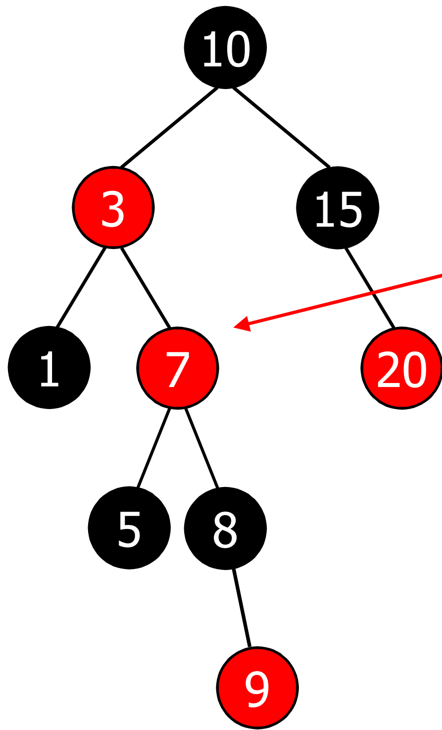


Inserção - exemplo

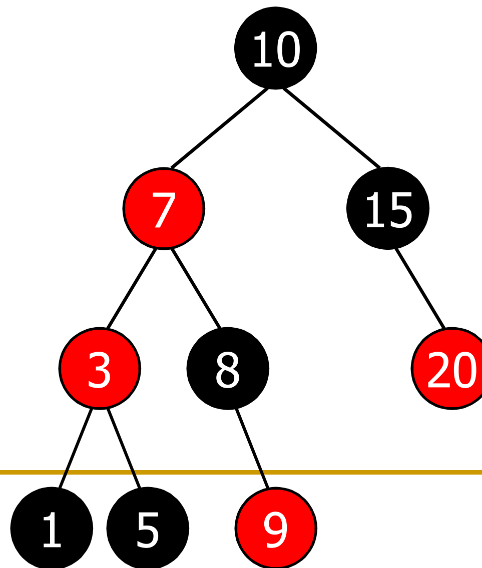


Continua ...

Inserção (cont.)

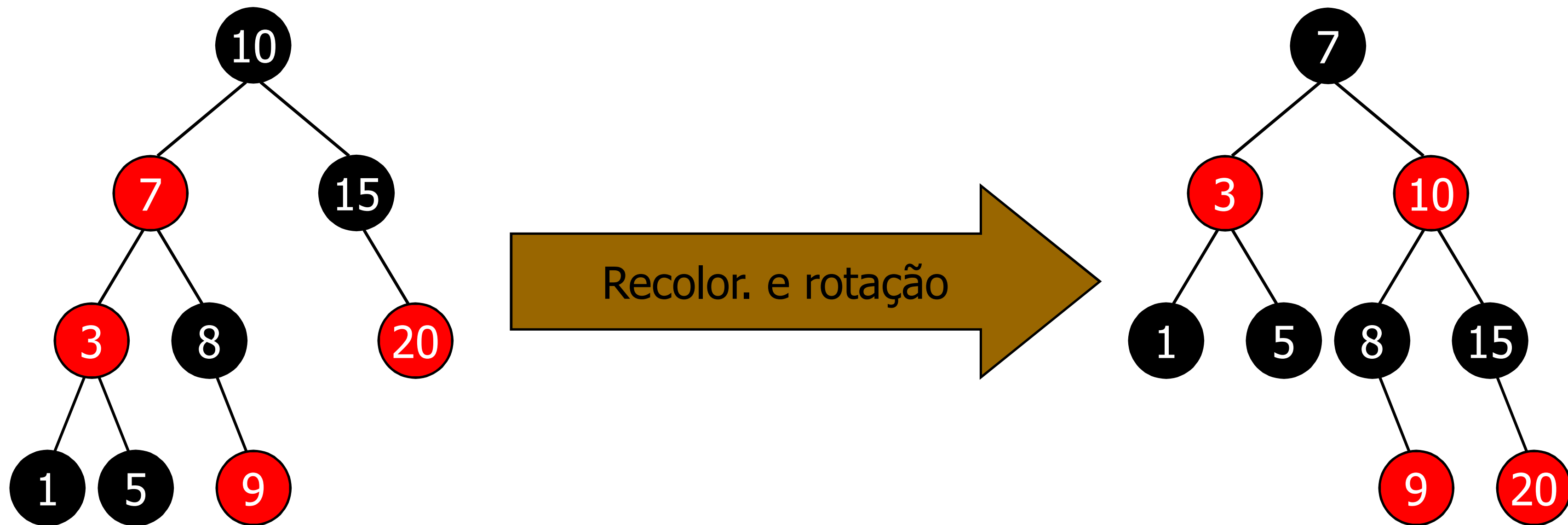


É necessário recolorir o pai e o avô.



O problema ainda não foi eliminado ...

Inserção (cont)



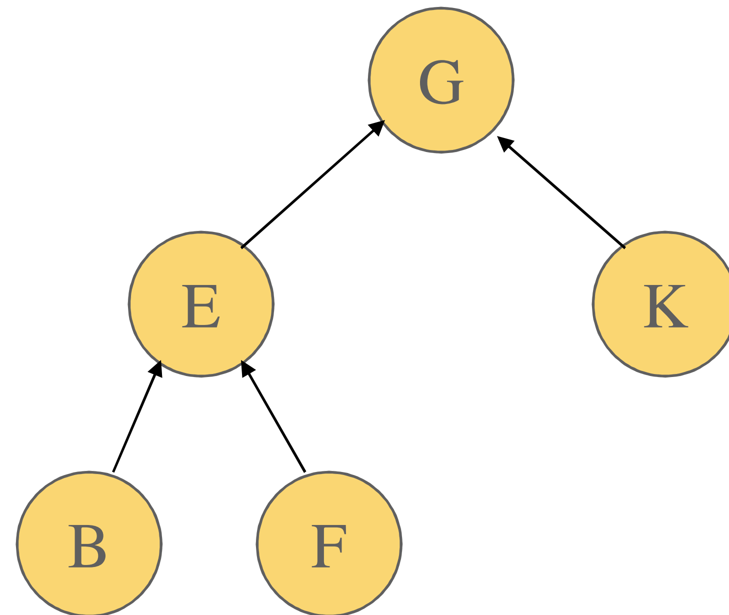
Referências na web

- http://users.cs.cf.ac.uk/Paul.Rosin/CM0212/DEMOS/RBTree/redblack.html
- http://ww3.algorithmdesign.net/handouts/RedBlackTrees.pdf
- http://www.cs.buap.mx/~titab/files/RedBlackTrees.pdf
- http://www.cs.buap.mx/~titab/files/RedBlackTrees.pdf
- http://www.cs.dal.ca/~nzeh/Teaching/Fall%202003/3110/RedBlackTrees.pdf
- animação 1

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 que são equivalentes ?

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

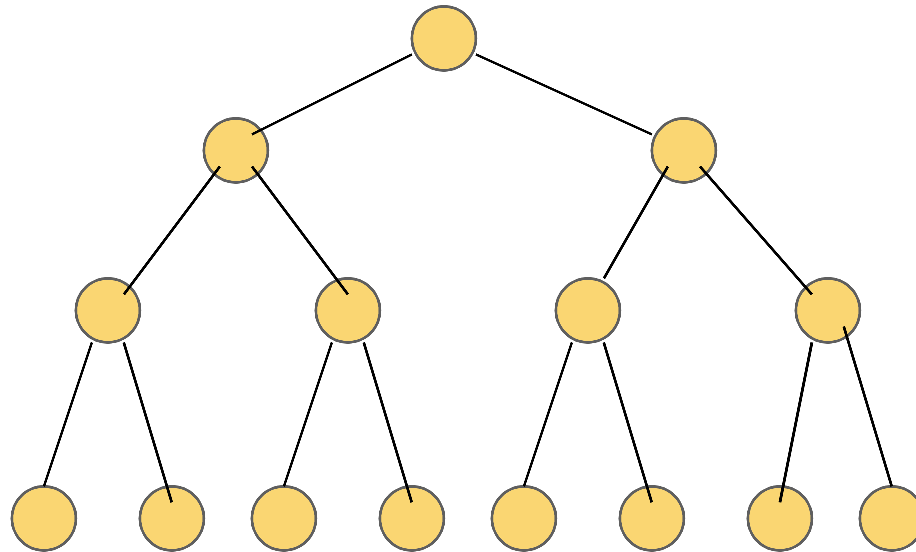
Um exemplo: classes de equivalência (I)

- fazer p e q equivalentes:

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

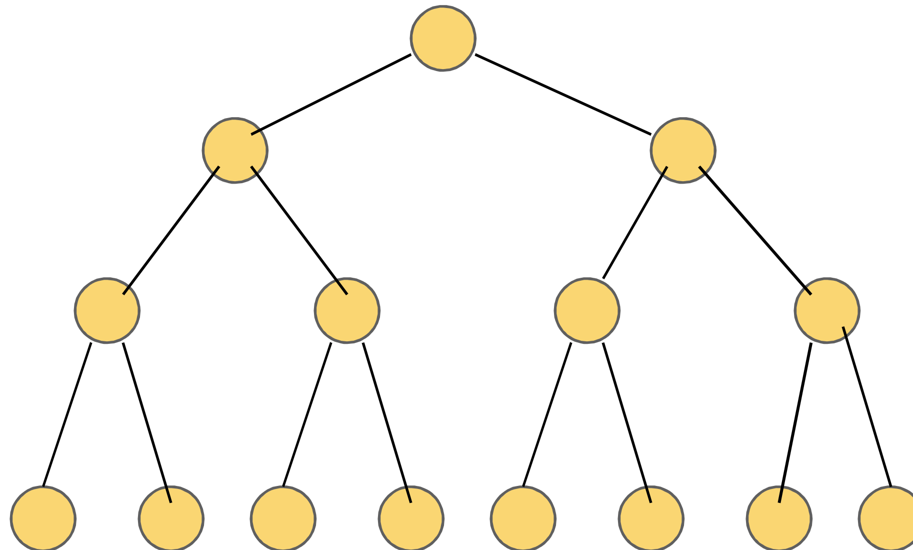
Árvore binária completa

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



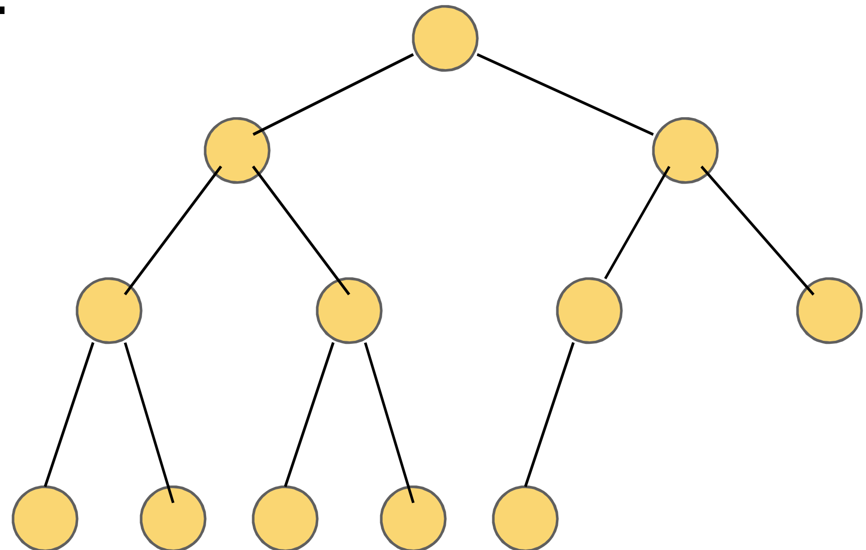
Árvore binária completa

- 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+1]



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.

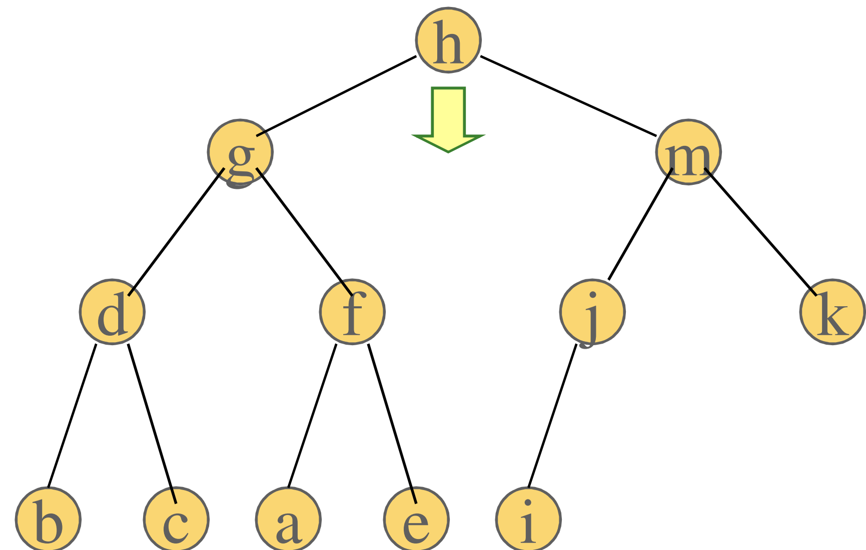


Filas de prioridade

- 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);
    }
}
```

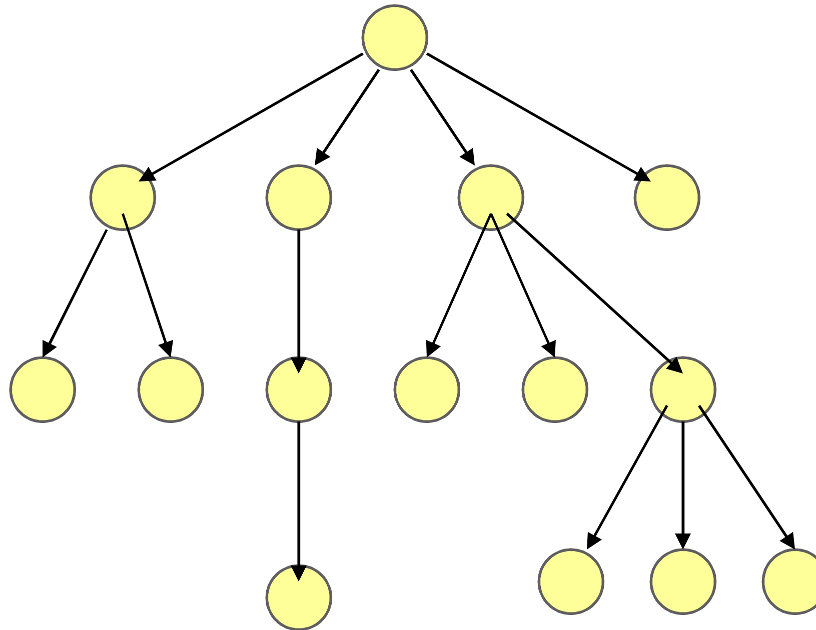
Referência na web

- Uma boa referência na web, em portugues, 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, ...T_m, árvores disjuntas.



Árvores gerais

- Representação binária