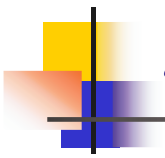
Árvores

Algoritmos e Estuturas de Dados

Cátia Vaz



Árvores

- As árvores são estruturas de dados usadas em diversas aplicações:
  - Bases de dados de grande dimensão.
  - Reconhecimento de frases geradas por linguagens (ex: programas, expressões aritméticas,...).
  - Modelação de sistemas organizacionais (ex: famílias, directórios de um computador, hierarquia de comando de uma empresa,...).
  - Determinação do caminho mais curto entre dois computadores de uma rede.

Cátia Vaz

Árvores

- Definição: Uma **árvore** é um par **(V,E)** de dois conjuntos não vazios em que **V** é um conjunto finito e **E** é uma relação binária em **V** (isto é, **E** $\subseteq$ **V** $\times$ **V**), que satisfazem duas condições:
  - entre dois nós existe um único caminho (um **caminho** - *path* - é uma sequência de arestas entre dois nós);
  - um único nó, denominado **raiz** (*root*), só existe como primeiro elemento nos pares de E; os restantes nós são um segundo elemento dos pares de E (podendo também ser primeiro elemento de alguns pares).
- **Nota:** o conjunto V é designado pelo conjuntos dos nós ou vértices (vertex) e E é o conjunto das arcos (edges).
- **Nota:** uma árvore é um caso especial de um grafo.

Árvores

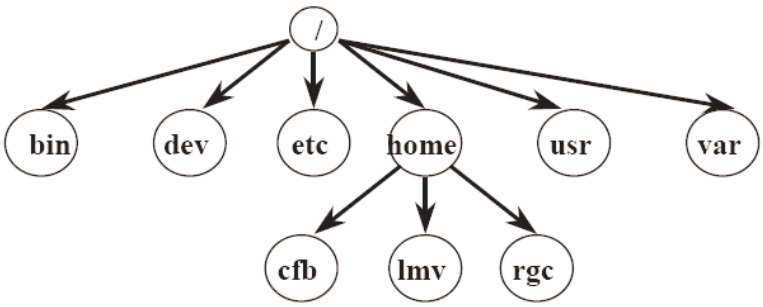
- Definição: se (v1,v2) $\in$ E, v1 é nó **ascendente** (*parent*) e v2 é nó **filho** (*child*).
  - A raiz é o único nó sem ascendente.
  - Nós sem filhos são designados por **folhas** (*leaves*).
  - Nós com filhos são designados **não-terminais**. Nós não-terminais, distintos da raiz, são designados **intermédios**.
- Definição: A árvore é de tipo K, sse todos os nós intermédios tiverem K filhos. Quando K=2, a árvore diz-se **binária**.
- Definição: O **nível** (*level*) de um nó é o número de arcos entre a raiz e o nó (**raiz tem nível 0**).

Árvores

- Definição: A **altura** (*heigh*) de uma árvore é o máximo dos níveis das folhas (uma árvore só com a raiz tem altura 0).
- **Representação gráfica das árvores:**
  - raiz no topo,
  - nós representados por círculos,
  - arestas representadas por linhas (ou setas no sentido da raiz para a folha).

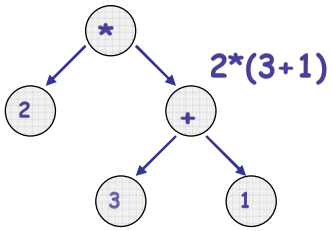
Árvores-Exemplos

- directórios do sistema operativo Linux



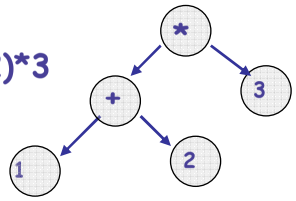
Árvores-Exemplos

expressões aritméticas



2*(3+1)

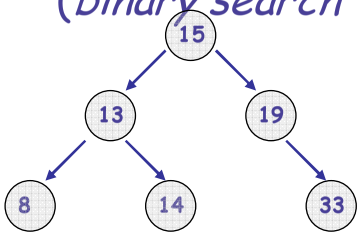
(1+2)*3

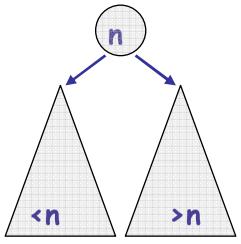


- árvore é avaliada de forma ascendente (*bottom-up*): 2*(3+1) = 8
- Parêntesis, usados nas expressões aritméticas para ultrapassar as prioridades dos operadores, são indicados na árvore através da posição dos nós

Árvores-Exemplos

árvore de pesquisa binária sem chaves repetidas (*binary search tree*) - BST





- sub-árvore da direita (esquerda) armazena números maiores (menores) que o nó.
  - Vantagens: pesquisa mais rápida-O(log N)
  - Inconvenientes: pode degenerar lista com pesquisa-O(N)

Árvore - representação

```
public class BST<E extends Comparable<E>> {
  private static class Node<E>{
      E key;
      Node<E> left, right, parent;
      Node(E v){ key= v; }
        }
  private Node<E> root;
  public static <E extends Comparable<E>> boolean less(E f, E g){
      return (f.compareTo(g)<0)? true:false;
  }

 public static <E extends Comparable<E>> boolean equals(E f, E g){
      return (f.compareTo(g)==0)? true:false;
 }
}
```


Árvores - propriedades

- **Teorema**: Uma árvore binária com N nós não-terminais possui N+1 folhas, desde que cada nó não-terminal tenha 2 descendentes.

  Estratégia de prova: indução, com base na construção da árvore a partir de duas sub-árvores.

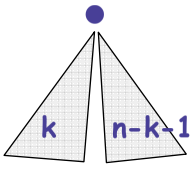
  Se N=0, a árvore possui um único nó (que é raiz e folha em simultâneo).

  Seja N>0 o número de nós não-terminais:

  - a subárvore esquerda tem **k** nós não-terminais;
  - a subárvore direita tem **N-k-1** nós não-terminais (**0<k<N+1**).

  Por hipótese de indução, a subárvore esquerda tem **k+1** folhas e a subárvore direita tem **N-k** folhas.

  Somando, a árvore tem **(k+1)+(N-k)=N+1** folhas.



Árvores - propriedades

- **Teorema**: Uma árvore binária com N nós não-terminais possui **2*N** arestas (**N-1 para os nós não-terminais e N+1 para as folhas**), desde que cada nó não-terminal tenha 2 descendentes.

  Estratégia de prova: contar o número de arestas para cada nó.

  Exceptuando a raiz, cada nó possui um único ascendente, pelo que só há uma aresta entre um nó e o seu ascendente.

  Pelo teorema anterior e observação anterior, **há N+1 arestas para as folhas**.

  Pelas duas observações anteriores, há N-1 arestas para os nós intermédios.

  Total:(N+1)+(N-1)=2*N arestas.

Árvores - propriedades

- **Teorema**: Numa árvore binária com N nós, o nível das folhas varia entre $\lfloor \log_2 N \rfloor$ e N-1

  Estratégia de prova: identificar níveis máximo e mínimo.

  - O nível máximo é o da árvore degenerada numa lista.

    Neste caso, o nível é N-1.

  - O nível mínimo é o da árvore balanceada,:

    - todos os níveis k têm 2^k nós, excepto o último eventualmente.Logo

$$\sum_{k=0}^{i-1} 2^k < N <= \sum_{k=0}^{i} 2^k. \quad \text{(i nível das folhas)}$$

  Então, $i < \log_2(N+1) <= i+1$ e portanto i é o maior inteiro igual ou inferior a $\log_2 N$. □

Árvores - varrimento

Existem diversas estratégias de percurso/varrimento (*transverse*) de árvores:

- **Prefixo**: antes do varrimento das sub-árvores.
- **Infixo**: varrer primeiro sub-árvore esquerda, imprimir operador e varrer depois a sub-árvore direita.
- **Pósfixo**: depois do varrimento das sub-árvores.

- Existe também varrimento de acordo com o nível:
  - **Profundidade** (*depth-first*)
  - **Largura** (*breadth-first*)

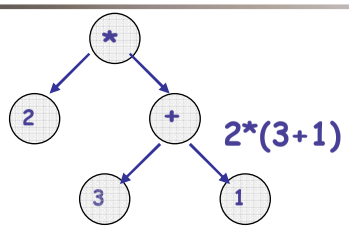
Árvore binária- varrimento

```
//...
private void transverseR(Node<E> h){
  if(h==null) return;
  h.visit();//visitar o nó
  transverseR(h.left);
  transverseR(h.right);
}
```

```
//...
public void transverse(){
    transverseR(root);
}
```

- Estes métodos implementam um varrimento **prefixo**.
- Se a invocação do método **`visit`** ocorrer entre as duas invocações recursivas do método **`tranverseR`**, obtemos um varrimento **infixo**.
- Se a invocação do método visit ocorrer após as invocações recursivas do método **`tranverseR`**, obtemos um varrimento **pósfixo**.

Varrimento - exemplo




```
public static void printR(Node h){
  if(h==null)return;
  if(!h.oper)
    System.out.print(h.item+" ");
  else{
      printR(h.left);printR(h.right);
     if (h.item==0)//multiplicação
        System.out.print("* ");
     else
        System.out.print("+ ");
        }
   }
```

Ex:expressões aritméticas

- Ao invocar o método **print** passando como parâmetro o nó com o operador *, este retorna:

  2 3 1 + *

```
public static void print(Node root){
   printR(root);
   System.out.println();}
```

Nota1:varrimento posfixo!
Nota2: cada nó tem que guardar informação se é um operando ou um operador! Não está a ser utilizada a classe BST anterior.

Árvore binária- varrimento

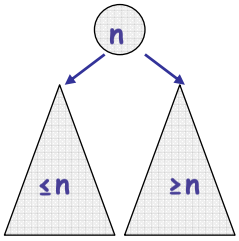
- O varrimento em **profundidade** é idêntico um varrimento **prefixo**.
- O varrimento em **largura** é feito por nível. As sub-árvores são colocadas **no fim de uma lista**.

```
void breadhtF(DLinkedList3<Node<E>> list){ //uma lista ligada que irá conter os
   if(list.size()==0) return;               //nós da árvore
   else {
     Node<E> aux = list.getFirst(); aux.visit(); //o getFirst retorna o obtido
     if(aux.left!=null) list.addLast(aux.left); //adiciona ao final da lista
     if(aux.right!=null)list.addLast(aux.right);
     list.remove(aux);
     breadhtF(list); }
    }

public void breadthF(){ //inicialmente a lista tem que conter a raíz
   DLinkedList3<Node<E>> list=new DLinkedList3<Node<E>>(); list.add(root);
   breadhtF(list);}
//(...)
```

Árvores Binárias de Pesquisa (BST)

Definição: Uma **árvore binária de pesquisa** *(binary search tree)* é uma árvore binária, que tem uma chave associada a cada nó, satisfazendo a seguinte propriedade:

- uma chave associada a um nó é **maior ou igual** que as chaves associadas a todos os nós da **sub-árvore da esquerda**;
- uma chave associada a um nó é **menor ou igual** que as chaves associadas a todos os nós da **sub-árvore da direita**.



Árvores Binárias de Pesquisa (BST): Procura

```
//(...)
public Node<E> search(E e){
  return recursiveSearch(root,e);
}
```

TREE-SEARCH(x, k)
if $x =$ NIL or $k = key[x]$
  then return x
if $k < key[x]$
  then return TREE-SEARCH$(left[x], k)$
  else return TREE-SEARCH$(right[x], k)$

Initial call is TREE-SEARCH$(root[T], k)$.

```
//(...)
private static <E extends Comparable<E>>
                      Node<E> recursiveSearch(Node<E> h,E x){
   if(h==null || equals(x,h.key)) return h;
   if(less(x, h.key)) return searchR(h.left,x);
   else return searchR(h.right,x);
}
```

Árvores Binárias de Pesquisa (BST): Procura

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ key[x]
2      do if k < key[x]
3            then x ← left[x]
4            else x ← right[x]
5  return x
```

TREE-SEARCH(x, k)
if $x =$ NIL or $k = key[x]$
  then return x
if $k < key[x]$
  then return TREE-SEARCH$(left[x], k)$
  else return TREE-SEARCH$(right[x], k)$

Initial call is TREE-SEARCH$(root[T], k)$.

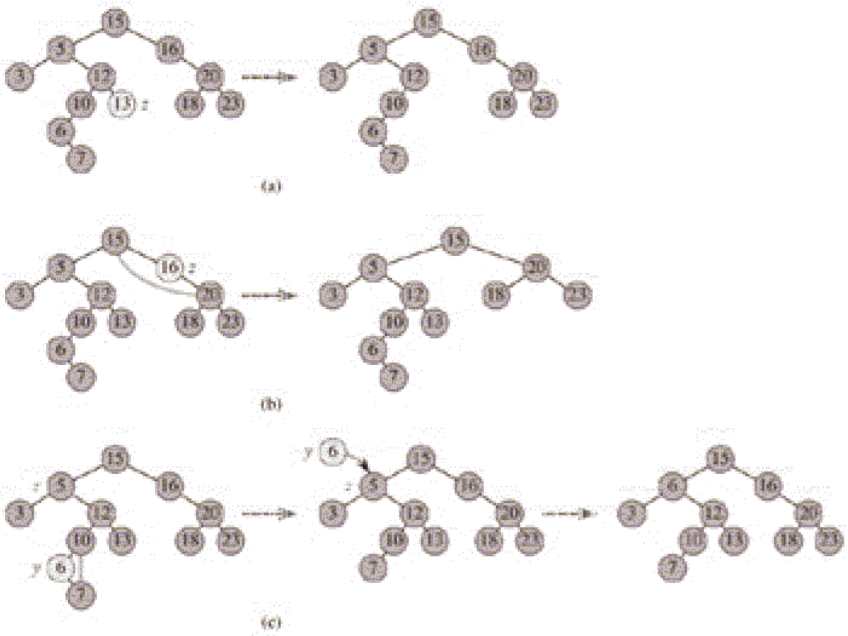
Árvores Binárias de Pesquisa (BST): Sucessor

```
TREE-SUCCESSOR(x)
1  if right[x] ≠ NIL
2      then return TREE-MINIMUM (right[x])
3  y ← p[x]
4  while y ≠ NIL and x = right[y]
5      do x ← y
6         y ← p[y]
7  return y
```

Árvores Binárias de Pesquisa (BST): Inserção

```
TREE-INSERT(T, z)
y ← NIL
x ← root[T]
while x ≠ NIL
    do y ← x
       if key[z] < key[x]
         then x ← left[x]
         else x ← right[x]
p[z] ← y
if y = NIL
  then root[T] ← z                ▷ Tree T was empty
  else if key[z] < key[y]
         then left[y] ← z
         else right[y] ← z
```


Árvores - Remocão

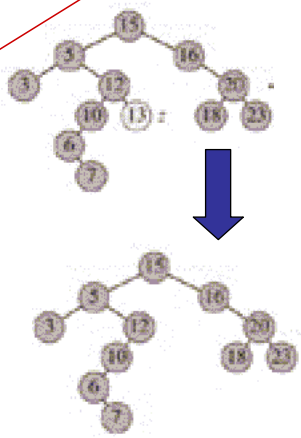


Árvores Binárias de Pesquisa (BST): Remoção

```
TREE-DELETE(T, z)
▷ Determine which node y to splice out: either z or z's successor.
if left[z] = NIL or right[z] = NIL
   then y ← z
   else y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL if y has no children.
if left[y] ≠ NIL
   then x ← left[y]
   else x ← right[y]
▷ y is removed from the tree by manipulating pointers of p[y] and x.
if x ≠ NIL
   then p[x] ← p[y]
if p[y] = NIL
   then root[T] ← x
   else if y = left[p[y]]
           then left[p[y]] ← x
           else right[p[y]] ← x
▷ If it was z's successor that was spliced out, copy its data into z.
if y ≠ z
   then key[z] ← key[y]
        copy y's satellite data into z
return y
```

Caso 1:
z não tem nós filhos

Exemplo

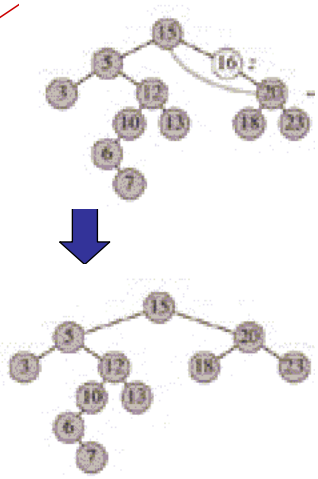


Árvores Binárias de Pesquisa (BST): Remoção

```
TREE-DELETE(T, z)
▷ Determine which node y to splice out: either z or z's successor.
if left[z] = NIL or right[z] = NIL
   then y ← z
   else y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL if y has no children.
if left[y] ≠ NIL
   then x ← left[y]
   else x ← right[y]
▷ y is removed from the tree by manipulating pointers of p[y] and x.
if x ≠ NIL
   then p[x] ← p[y]
if p[y] = NIL
   then root[T] ← x
   else if y = left[p[y]]
           then left[p[y]] ← x
           else right[p[y]] ← x
▷ If it was z's successor that was spliced out, copy its data into z.
if y ≠ z
   then key[z] ← key[y]
        copy y's satellite data into z
return y
```

Caso 2:
z tem um nó filho

Exemplo

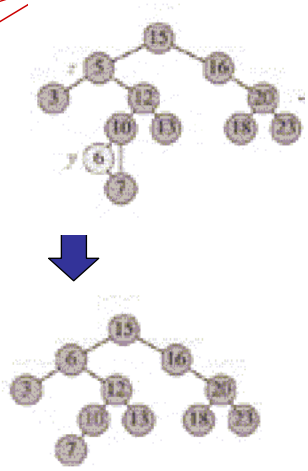


Árvores Binárias de Pesquisa (BST): Remoção

```
TREE-DELETE(T, z)
▷ Determine which node y to splice out: either z or z's successor.
if left[z] = NIL or right[z] = NIL
   then y ← z
   else y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL if y has no children.
if left[y] ≠ NIL
   then x ← left[y]
   else x ← right[y]
▷ y is removed from the tree by manipulating pointers of p[y] and x.
if x ≠ NIL
   then p[x] ← p[y]
if p[y] = NIL
   then root[T] ← x
   else if y = left[p[y]]
           then left[p[y]] ← x
           else right[p[y]] ← x
▷ If it was z's successor that was spliced out, copy its data into z.
if y ≠ z
   then key[z] ← key[y]
        copy y's satellite data into z
return y
```

Caso 3:
z tem nós filhos

Exemplo



Árvores Binárias de Pesquisa

- A complexidade da inserção depende do tipo de árvore:
  - **Árvore degenerada** (lista): O(N)
  - **Árvore perfeitamente balanceada**: a complexidade determinada pela pesquisa do local de inserção é |ln N|, ou seja, O(log N)
  - **Árvore de inserção aleatória**: considerando que a altura de cada subárvore possui uma distribuição uniforme 1..N, tem-se que a pesquisa do local de inserção custa ln N, ou seja, O(log N)
    - O custo é O(1) (visita à raiz) somado ao custo da visita à sub-árvore: esta pode ter entre 0 e N-1 nós, com distribuição uniforme

      $C(N) = O(1) + 1/N * \sum C(k-1) \quad 1 \le k \le N$, para $N \ge 2$

      $C(1) = O(1)$

Cátia Vaz

Árvores Binárias de Pesquisa

- **Árvore de inserção aleatória**: considerando que a altura de cada subárvore possui uma distribuição uniforme 1..N, tem-se que a pesquisa do local de inserção custa ln N, ou seja, O(log N)
  - O custo é O(1) (visita à raiz) somado ao custo da visita à sub-árvore: esta pode ter entre 0 e N-1 nós, com distribuição uniforme

    $C(N) = O(1) + 1/N * \sum C(k-1) \quad 1 \le k \le N$, para $N \ge 2$

    $C(1) = O(1)$

    Para eliminar ∑, multiplica-se ambos os lados por N, e subtrai-se a fórmula para N-1

    $NC(N) - (N-1)C(N-1) = N\,O(1) + \sum C(k-1) - ((N-1)O(1) + \sum C(k-2)) = O(1) + C(N-1)$

Árvores Binárias de Pesquisa

$NC(N) = NC(N-1) + O(1)$

$C(N) = C(N-1) + O(1)/N$

por substituição telescópica

$C(N) = C(1) + O(1/N + 1/(N-1) + \ldots + \frac{1}{2})$

Trata-se da série harmónica, pelo que

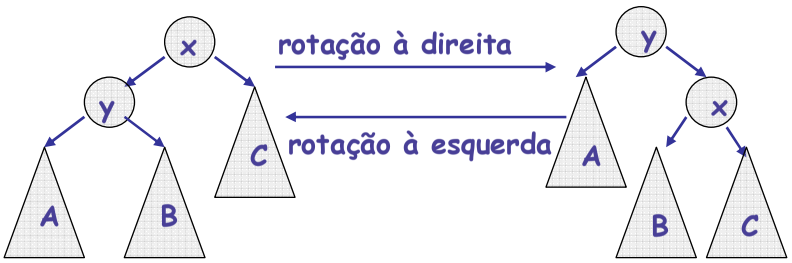
$C(N) = O(\ln N)$

□

- Nota: o custo da inserção é mínimo se a árvore for balanceada

Árvores Balanceadas

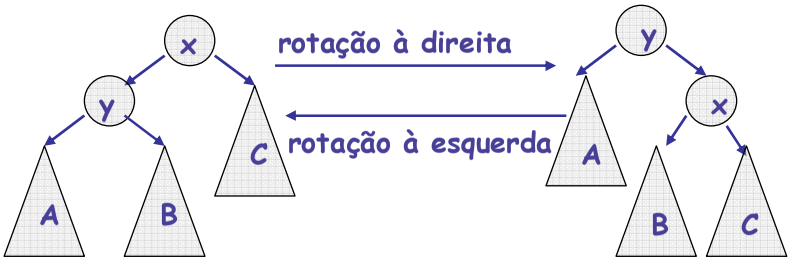
- Definição: Uma árvore diz-se **balanceada**, sse em todos os nós a diferença entre as alturas das sub-árvores for igual ou inferior a 1.
- Para manter a árvore balanceada, respeitando a ordem, depois da inserção pode ser necessário rodar a configuração de nós (rotação simples e rotação dupla)
- Operações de rotação são classificadas de acordo com o sentido (à direita e à esquerda).



Árvores Balanceadas

- Operações de rotação:

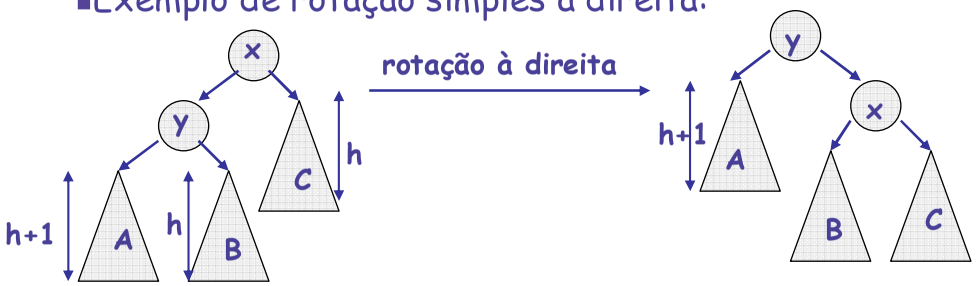
```
private Node rotRight(Node x){
   Node aux=x.left; x.left=aux.right; aux.right=x; return aux;}

private Node rotLeft(Node y){
   Node aux=y.right; y.right=aux.left; aux.left=y; return aux;}
```



Árvores Balanceadas

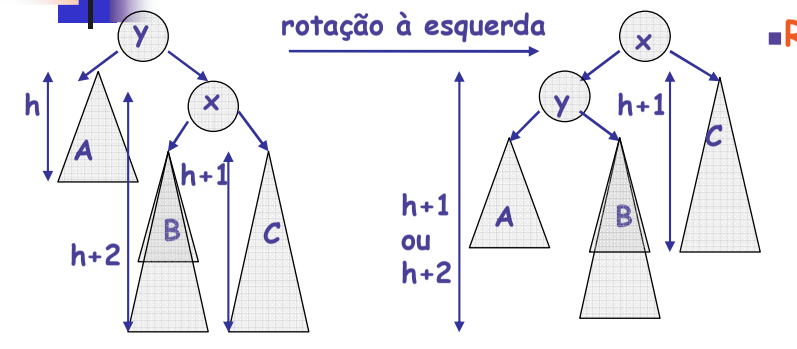
- Há dois pares de casos em que se torna necessário rodar sub-árvores:
  - Rotação Simples (à direita ou à esquerda)
  - Rotação Dupla (à direita ou à esquerda)

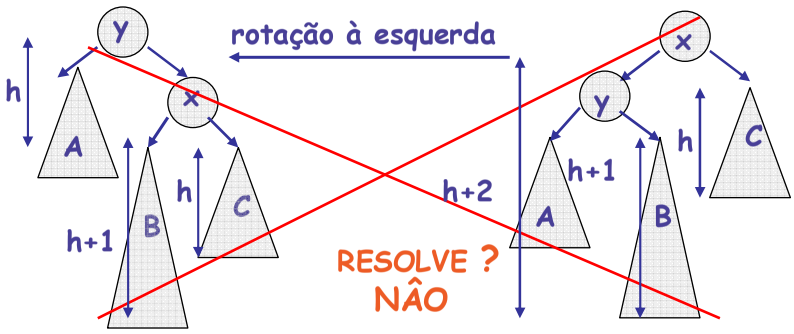
- Exemplo de rotação simples à direita:



Cátia Vaz

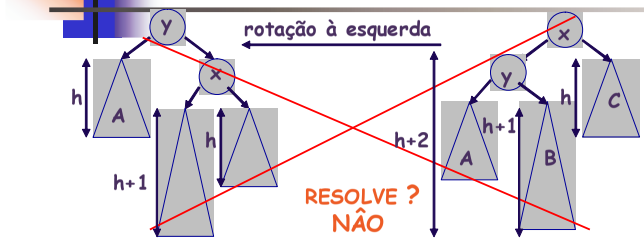
Árvores Balanceadas



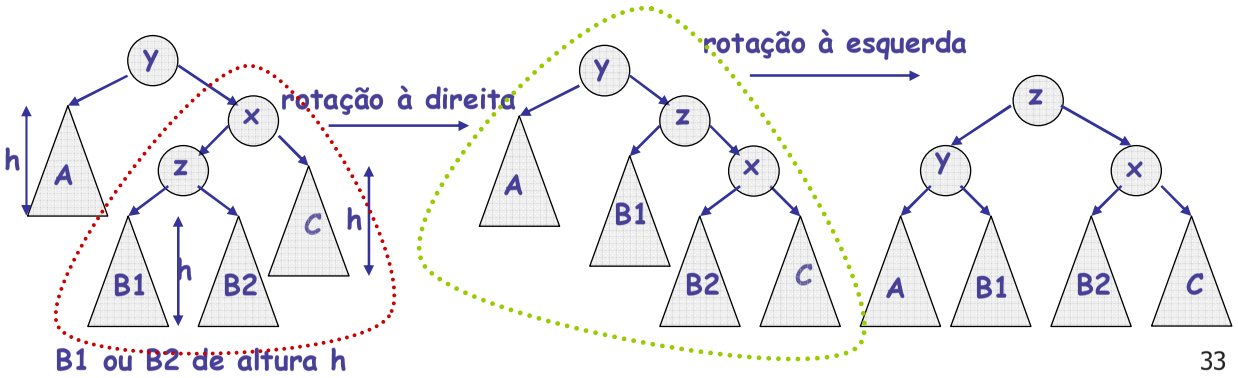


- **Rotação à Esquerda**:
  - se a altura da **sub-árvore direita (x)** for superior a 1 à altura da **sub-árvore esquerda (A)** .
  - a árvore fica balanceada? **Se e só se**, a altura da **sub-árvore direita de x (C)** for maior ou igual à altura da **sub-árvore esquerda de x (B)** .

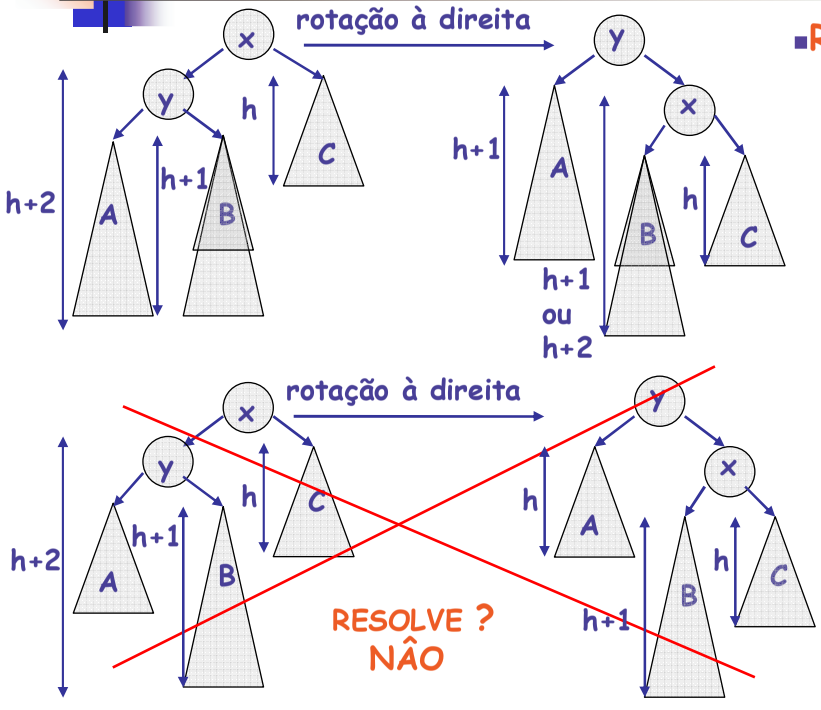
Árvores Balanceadas



- Se a altura da **sub-árvore direita de x (C)** for menor à altura da **sub-àrvore esquerda de x (B)**, para balancear a árvore é necessário uma **rotação dupla**.

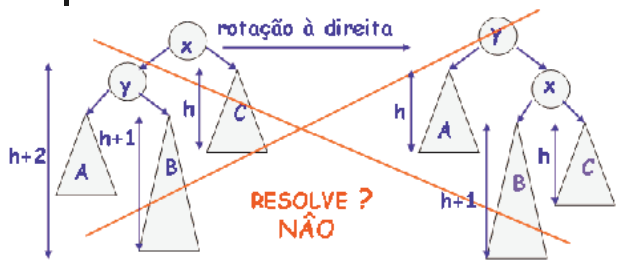


Árvores Balanceadas



- **Rotação à direita:**
  - se a altura da **sub-árvore esquerda (y)** for superior a 1 à altura da **sub-árvore direita (C)**.
  - a árvore fica balanceada? **Se e só se**, a altura da **sub-árvore esquerda de y (A)** for maior ou igual à altura da **sub-árvore direita de y (B)**.

Árvores Balanceadas



- Se a altura da **sub-árvore esquerda de y (A)** for menor à altura da **sub-àrvore direita de y (B)**, é necessário uma **rotação dupla**.