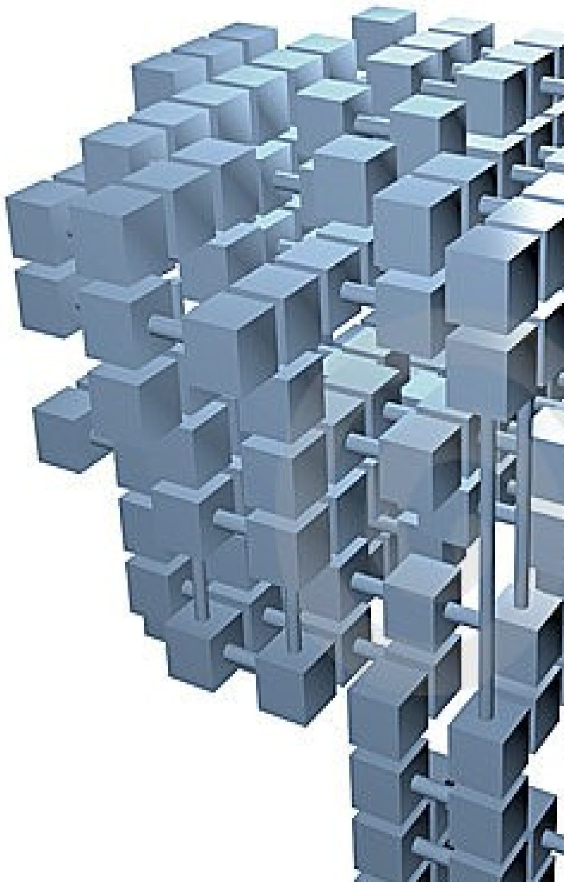
Campus de Frutal
UEMG UNIVERSIDADE DO ESTADO DE MINAS GERAIS

# SISTEMAS DE INFORMAÇÃO

## Estrutura de Dados 1

## Árvores

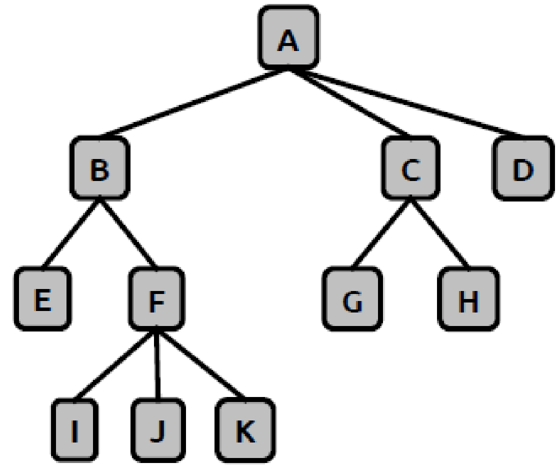


Prof. Ivan José dos Reis Filho
ivanfilhoreis@gmail.com

# Estrutura de Dados

- Por que "estruturar" os dados?
  - Principal operação: BUSCAS
- Preocupações constantes:
  - Tempo:
- Custo para buscar ou manter a estrutura (inserir/remover)
- Não queremos comparar (nem tocar) muitos dados
  - Espaço:
- Como representar esta estrutura
- Questões relevantes:
  - Dados dinâmicos
  - Critérios de ordenação
  - Tamanho/quantidade dos dados

# Árvores

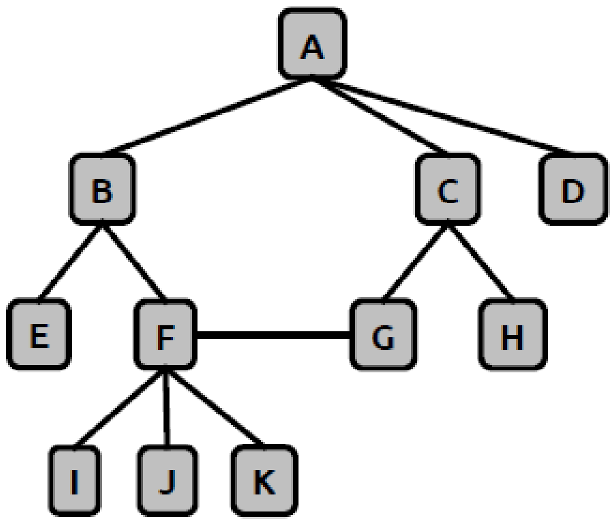


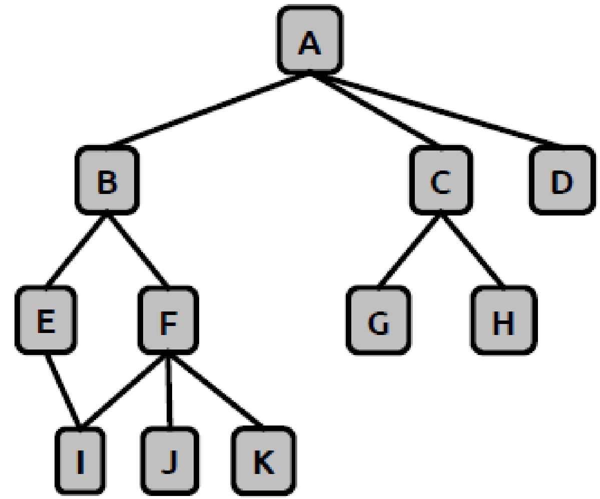

Nomenclatura utilizada:

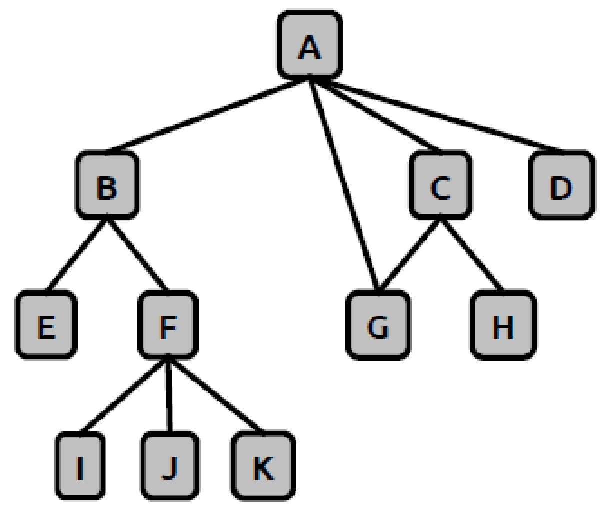
- **A é *pai de B, C é pai de G***
- **J é *filho de F, C é filho de A***
- **A é a *raiz da árvore (nó sem pai)***
- **C é *irmão de B***
- **A, B, C e F são nós *internos (com pelo menos um filho)***
- **E, I, J, K, G, H e D são nós *externos ou folhas (sem filhos)***
- **B é *ancestral de J, A é ancestral de H***
- **I é *descendente de A, H é descente de A***
- *Sub-árvore: nó e seus descendentes*
- **C é *raíz de uma sub-árvore***
- *Profundidade de um nó: número de ascendentes*
- *Altura de uma árvore (ou sub-árvore): maior profundidade*
- *Grau: maior número de filhos*

# Árvores

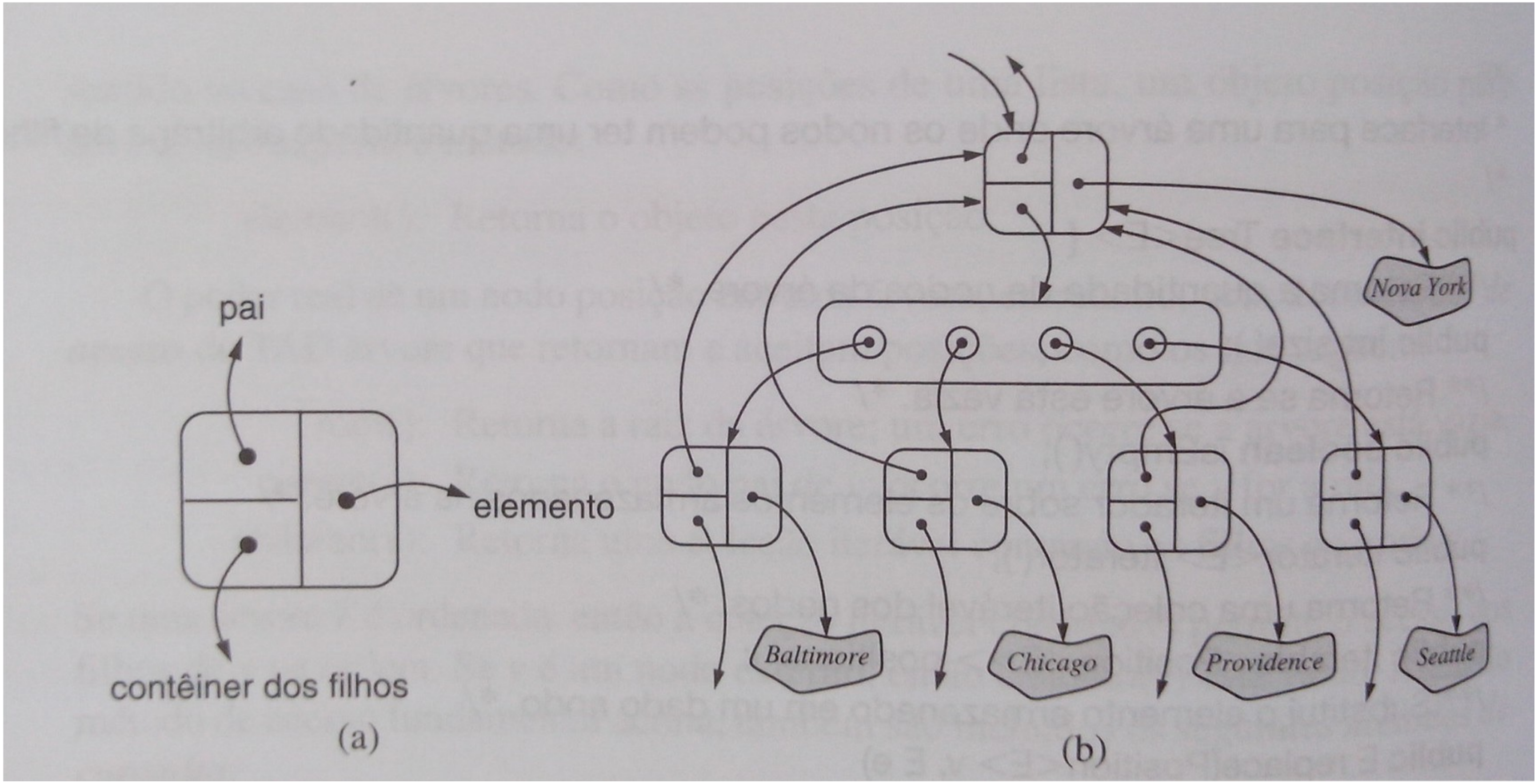
- Não são árvores!!!



# Árvores

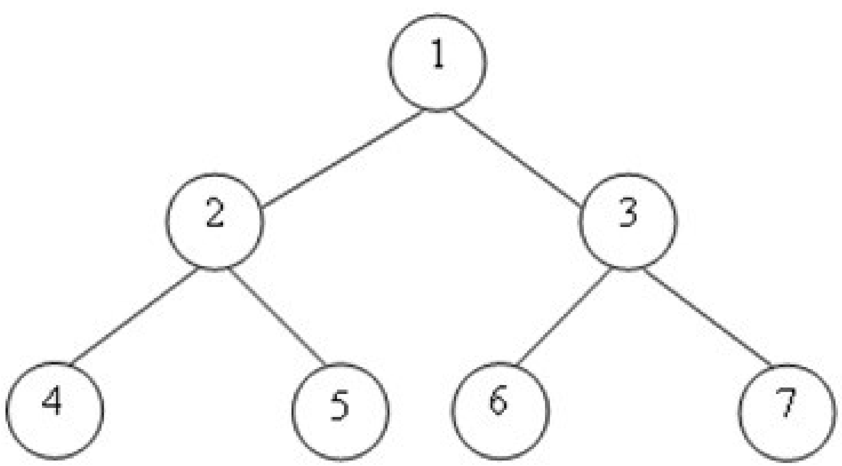


pai
elemento
contêiner dos filhos
(a)
Nova York
Baltimore
Chicago
Providence
Seattle
(b)

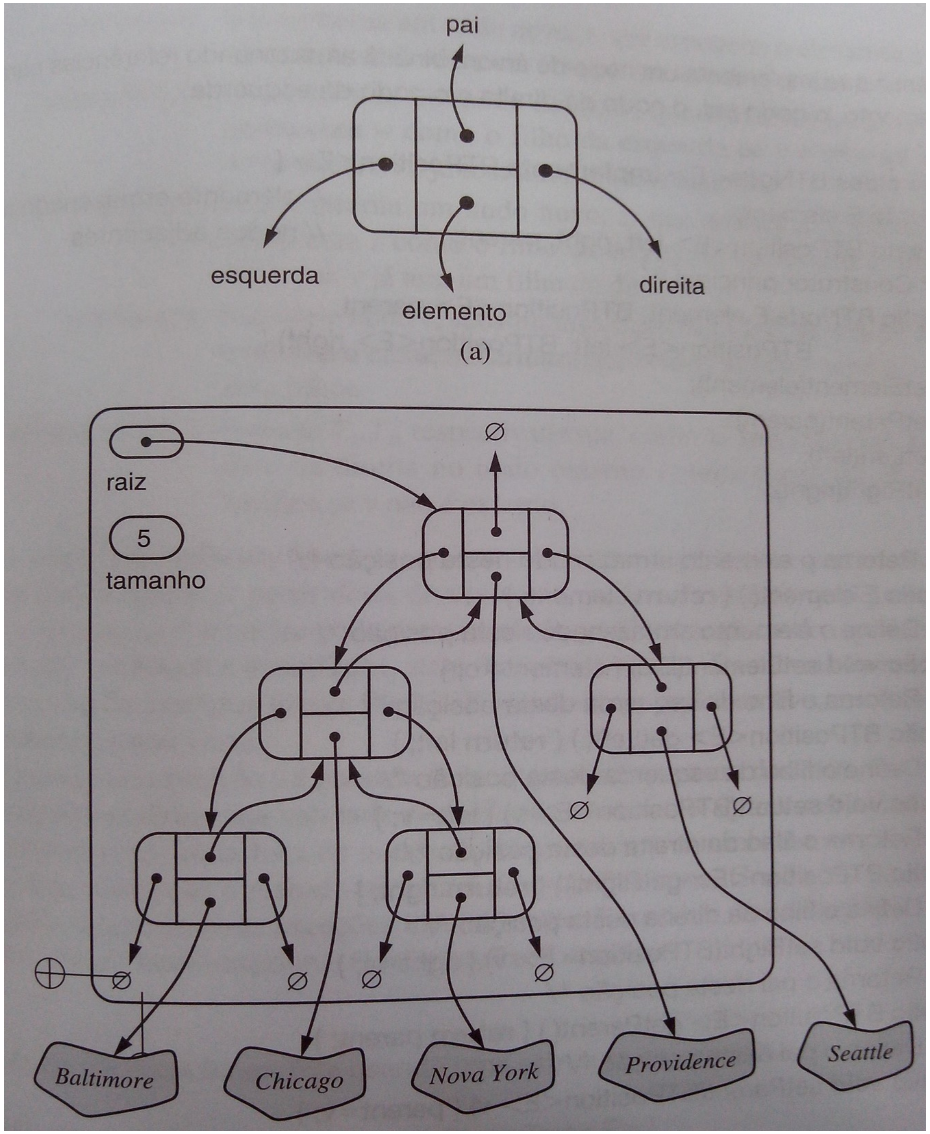
# Árvores Binárias

No máximo 2 filhos

- Árvore própria: apenas 0 ou 2 filhos
- Nomenclatura:
  - Filha da esquerda/direita
  - Sub-árvore da esquerda/direita



# Árvores Binárias



# Árvores Binárias

- Atravessamento (ou caminhamento) de árvore é a passagem de forma sistemática por cada um de seus nós.
- Diferentes formas de percorrer os nós de uma árvore.

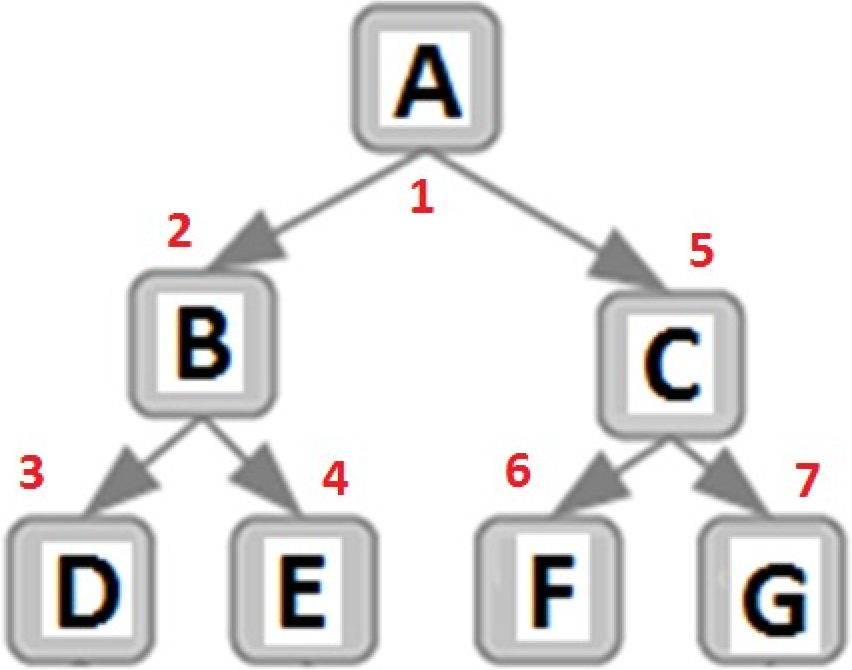
  - Pré-Ordem

  - Em Ordem

  - Pôs Ordem

  - Em nível

# Percurso: Pré-Ordem

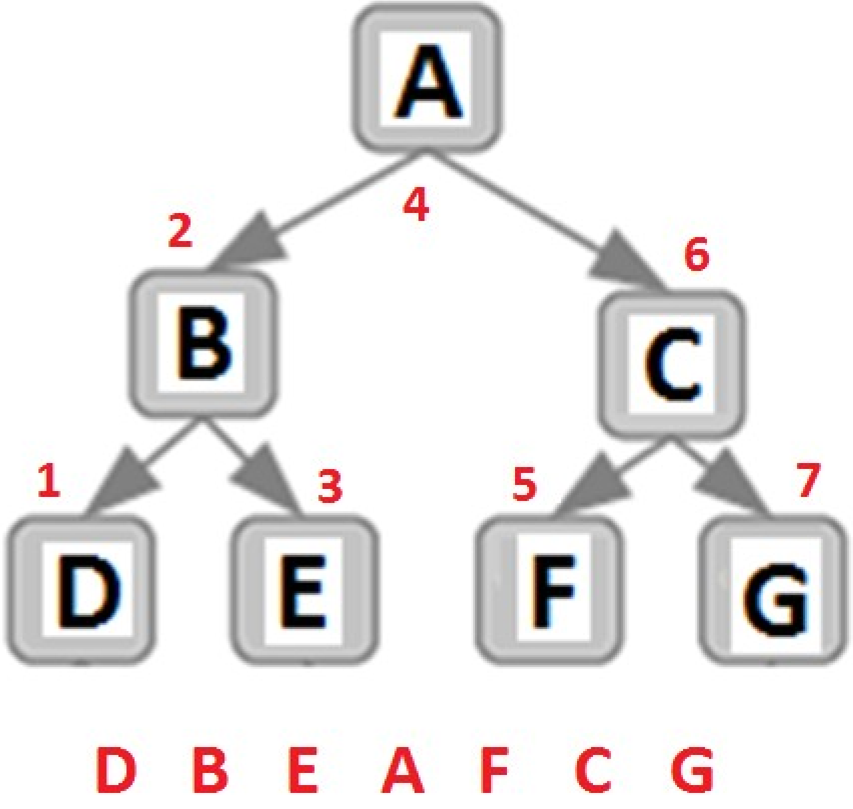



A B D E C F G

# Percurso: Pré-Ordem

```
void preorderPrint( const Tree& T, const Position& p) {
    cout << *p;

    if( p.left() )
        preorderPrint( T, p.left() );

    if( p.right() )
        preorderPrint( T, p.right() );
}
```

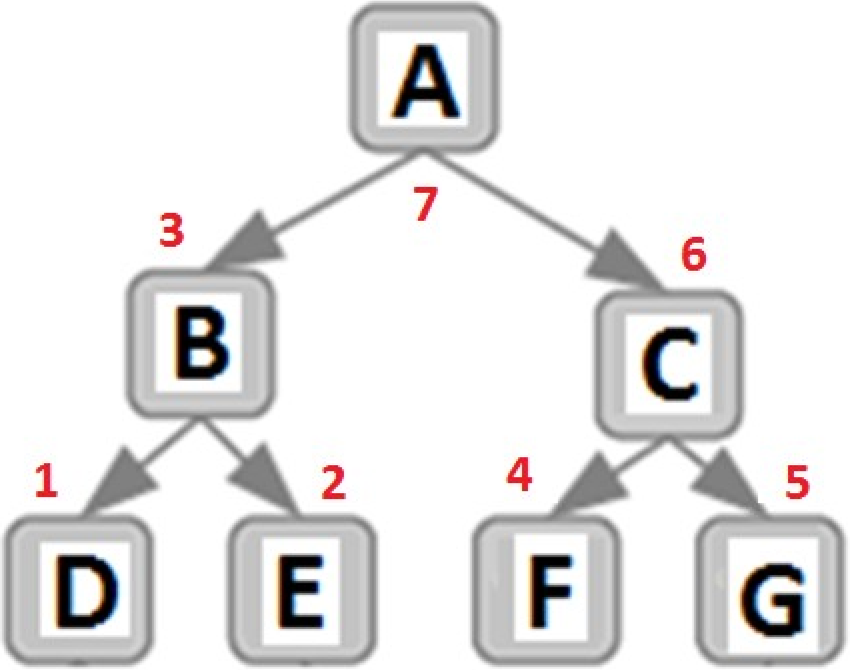
# Percurso: Em Ordem



# Percurso: Em Ordem

**Algorithm** inorder$(T, p)$:
  **if** $p$ is an internal node **then**
    inorder$(T, p.\text{left}())$    {recursively traverse left subtree}
  perform the "visit" action for node $p$
  **if** $p$ is an internal node **then**
    inorder$(T, p.\text{right}())$    {recursively traverse right subtree}

# Percurso: Pós-Ordem

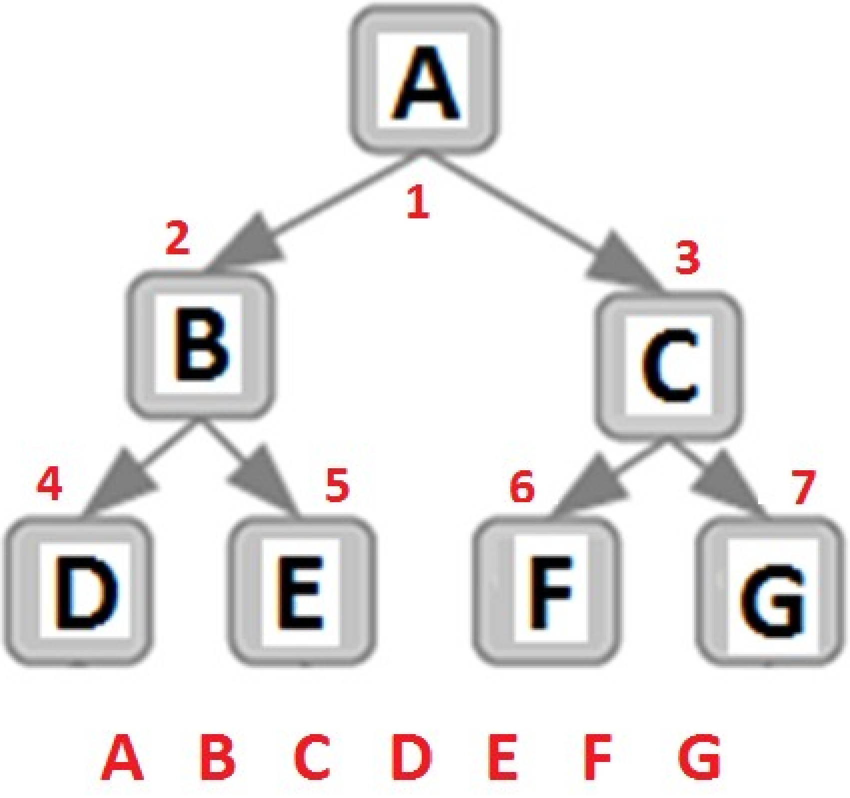



D E B F G C A

# Percurso: Pós-Ordem

**Algorithm** binaryPostorder($T$, $p$):
  **if** $p$ is an internal node **then**
    binaryPostorder($T$, $p$.left())   {recursively traverse left subtree}
    binaryPostorder($T$, $p$.right())   {recursively traverse right subtree}
  perform the "visit" action for the node $p$

# Percurso: Em Nível



# Percurso: Em Nível

```
void levelsPrint( const Tree& T, const Position& p) {
    cout << *p;
    PositionList ch;
    ch.insertBack( p.left() );
    ch.insertBack( p.right() );

    while( !ch.empty() ) {

        Position e = ch.front();
        ch.eraseFront();

        cout << *e;

        ch.insertBack( e.left() );
        ch.insertBack( e.right() );
    }
}
```

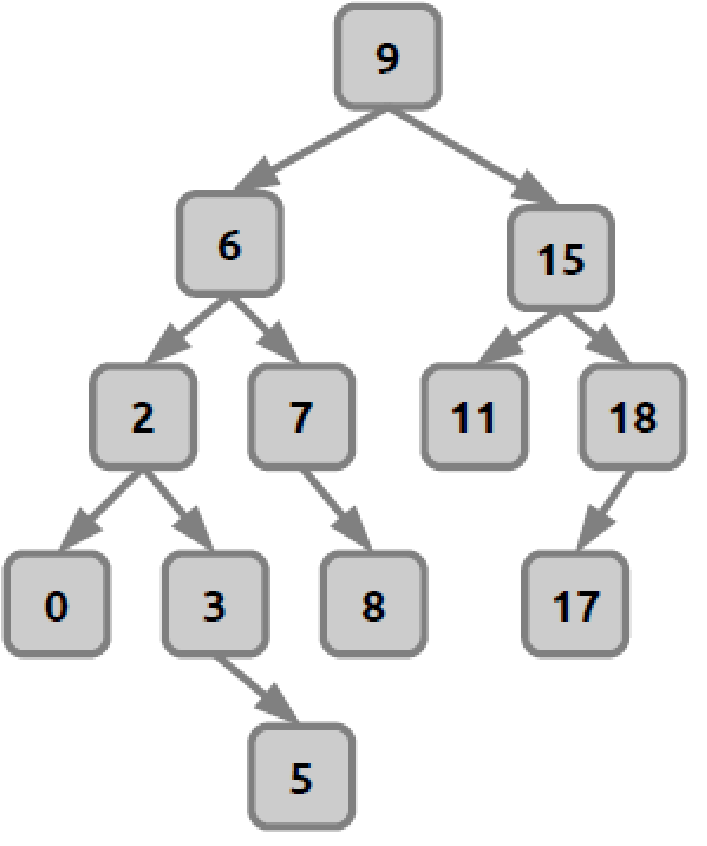
# Árvore Binária de Busca (ABB – BST)

Critério de ordenação:

– A chave de cada nó é:

<u>Maior</u> que todas as chaves da sub-árvore à esquerda;

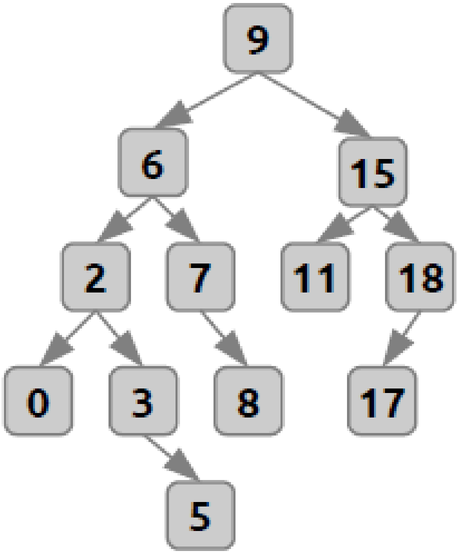
<u>Menor</u> que todas as chaves da sub-árvore à direita.



# ABB - Busca

- Para buscar uma chave ***key,*** percorremos um caminho partindo da raiz para as folhas;
- O próximo nó a ser visitado depende do resultado da comparação
- Se alcançamos uma folha e esta não é chave, significa que não encontramos

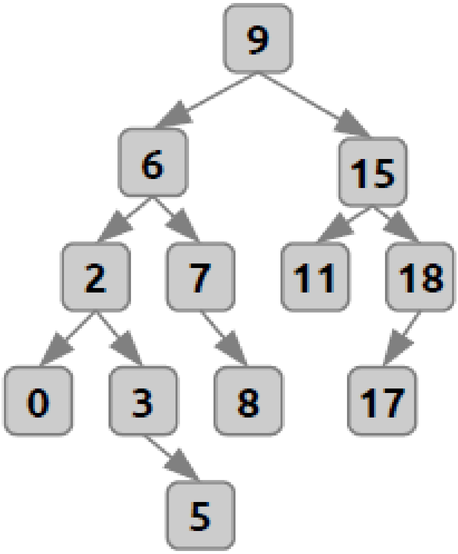
```
node<T>& Tree::search( node<T> &n, T key ) {
    if( !n )
        return 0;
    if( key < n.key )
        return search( n.left, key );
    else if( key > n.key )
        return search ( n.right, key );
    else // key == n.key
        return n;
}
```

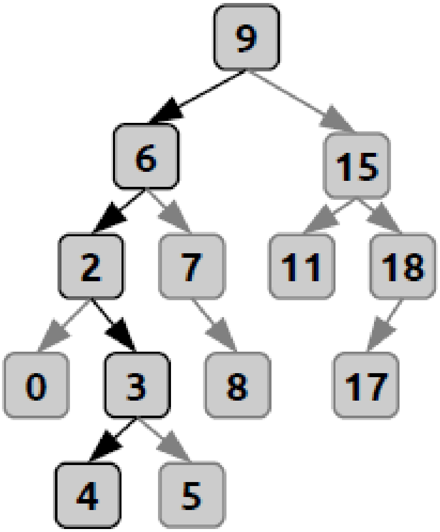


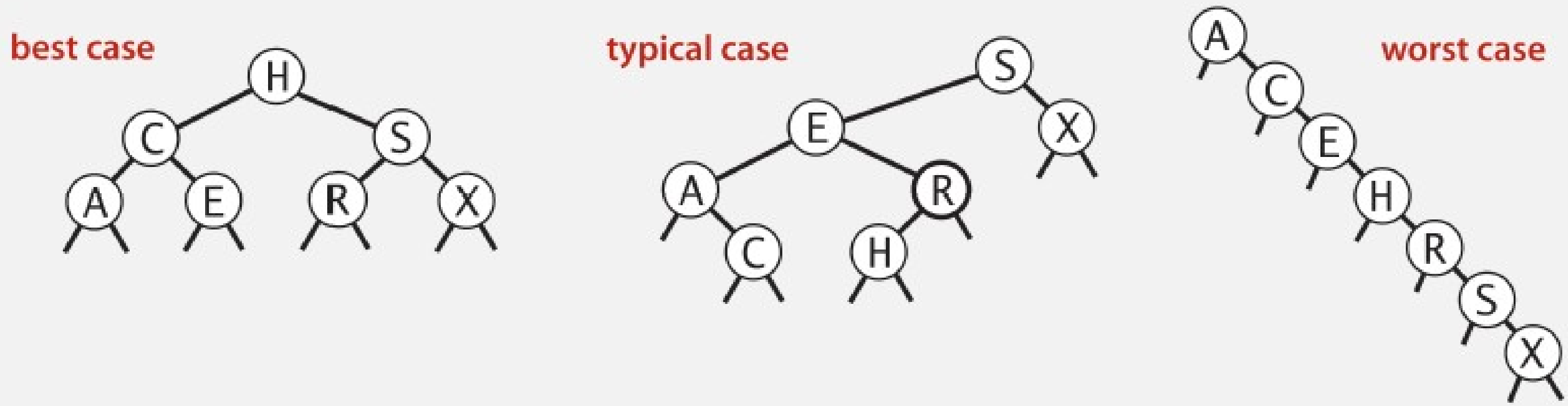
# ABB - Inserção

- Procuramos pelo elemento e inserimos assim que encontramos a posição correta
- Elementos repetidos?
- Folhas sem chaves?

```
void Tree::insert( T key ) {
    root = insert( root, key);
}

node<T>& Tree::insert( node<T> &n, T key ) {
    if( !n )
        return new node<T>(key);
    if( key < n.key )
        n.left = insert( n.left, key );
    else if( key > n.key )
        n.right = insert ( n.right, key );
    // else // key == n.key
    // atualizar campos?!
    return n;
}
```



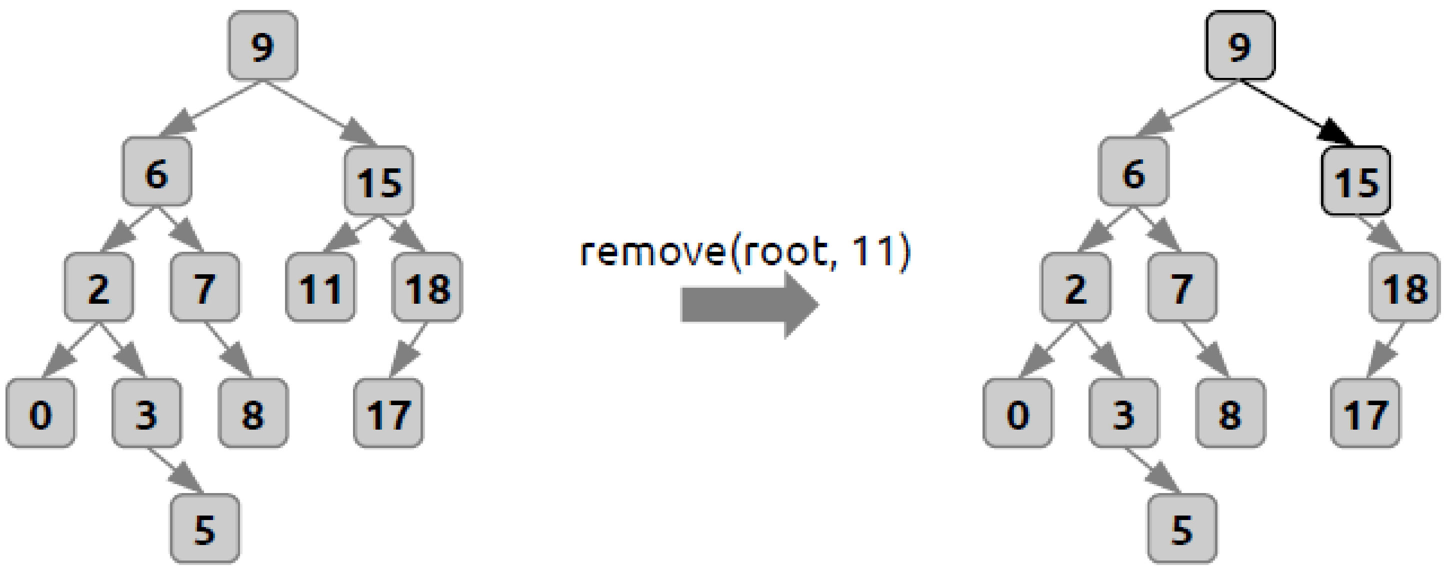






Remark. Tree shape depends on order of insertion.

# ABB - Remoção

- Caso 1: nó sem filhos



# ABB - Remoção

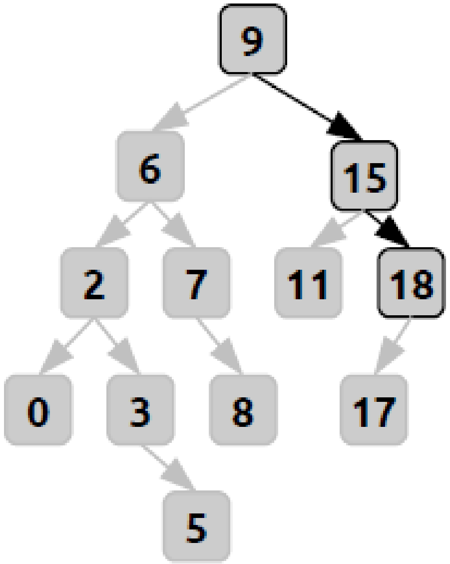
- Caso 2: nó com apenas um filho

# ABB - Remoção

- Caso 3: nó com dois filhos

# ABB - Máximo

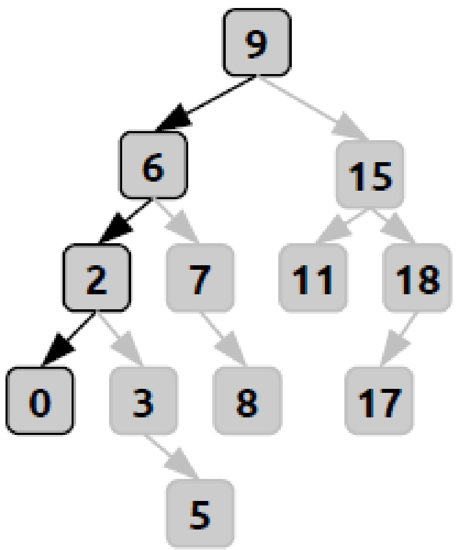
- Encontrando o maior elemento de uma ABB:




```
T Tree::max() {
    return max(root).key;
}

node<T>& Tree::max( node<T> &n ) {
    if( ! n.right )
        return n;
    else
        return max( n.right );
}
```

# ABB - Mínimo

- Encontrando o menor elemento de uma ABB:




```
T Tree::min() {
    return min(root).key;
}

node<T>& Tree::min( node<T> &n ) {
    if( ! n.left )
        return n;
    else
        return min( n.left );
}
```

# ABB - Remoção

```
void Tree::remove( T key ) {
    root = remove(root, key);
}

node<T>& Tree::remove( node<T> &n, T key ) {

    if(!n)
        return 0;
    if( key < n.key)
        n.left = remove(n.left,key);
    else if( key > n.key)
        n.right = remove(n.right,key);
    else {
        if( !n.right ) {
            node<T> &left = n.left;
            delete n;
            return left;
        }
        if( !n.left ) {
            node<T> &right = n.right;
            delete n;
            return right;
        }
        n.key = max(n.left).key; // e outros dados
        n.left = remove( n.left, n.key );
        return n;
    }
}
```

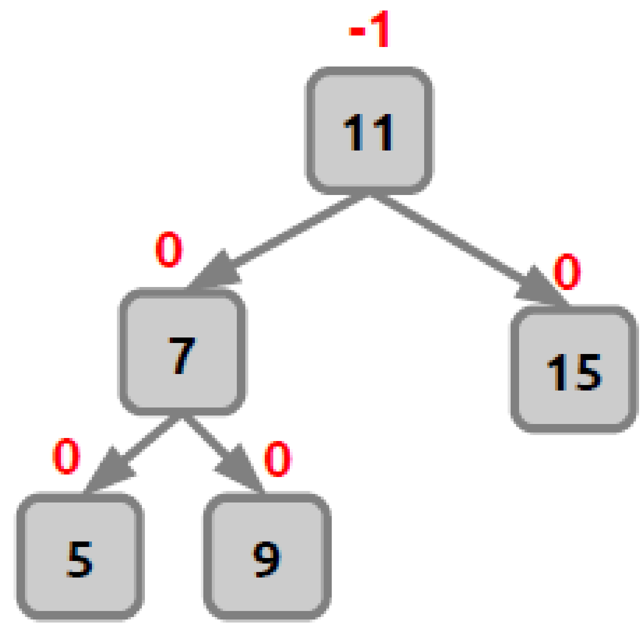
# Árvores AVL

Autores: Adel'son-Vel'skii e Landis (1962)

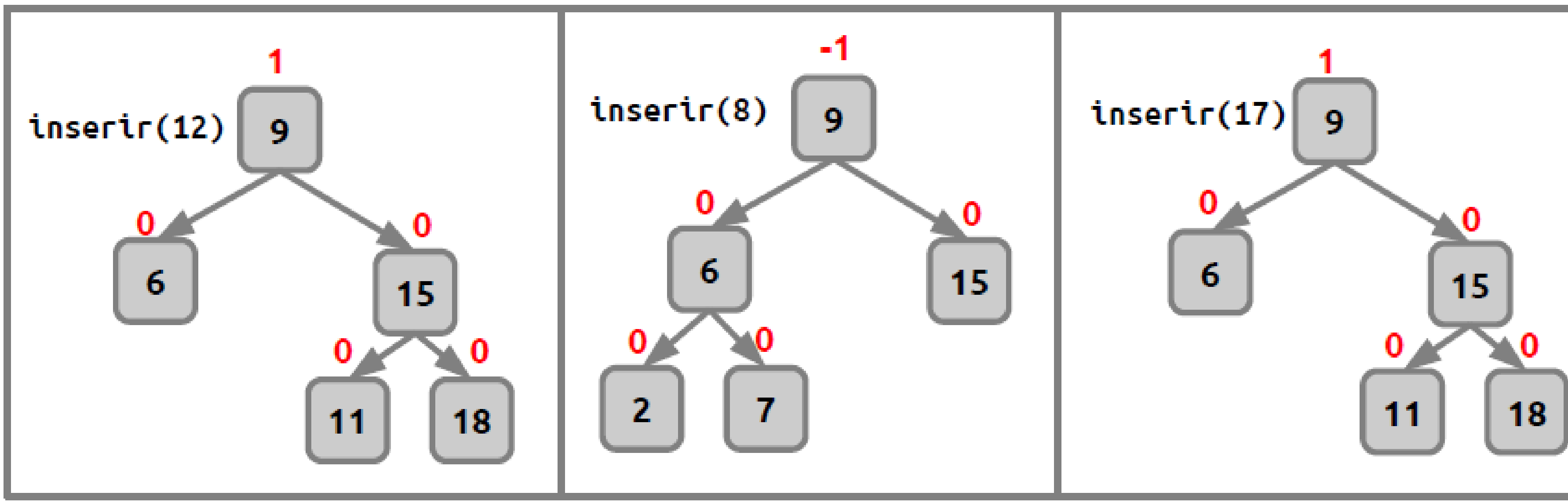
- Árvore BB
- Algoritmos de balanceamento na inserção e remoção
- Definição de árvore balanceada:
  - A diferença das alturas das sub-árvores (direita esquerda) de um nó é igual a -1, 0 ou 1.
- Cada nó armazena a diferença das alturas das sub-árvores

# Desbalanceamento

```
inserir(4)

inserir(6)

inserir(8)

inserir(10)

inserir(12)

inserir(16)
```

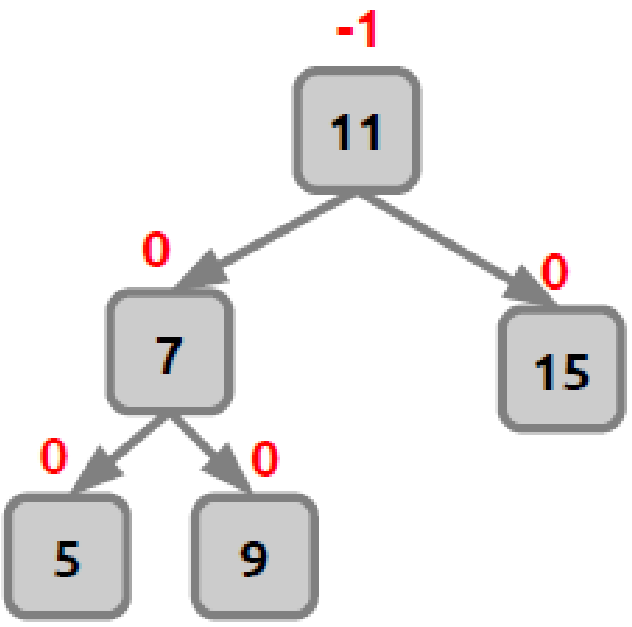


# Desbalanceamento



# Desbalanceamento

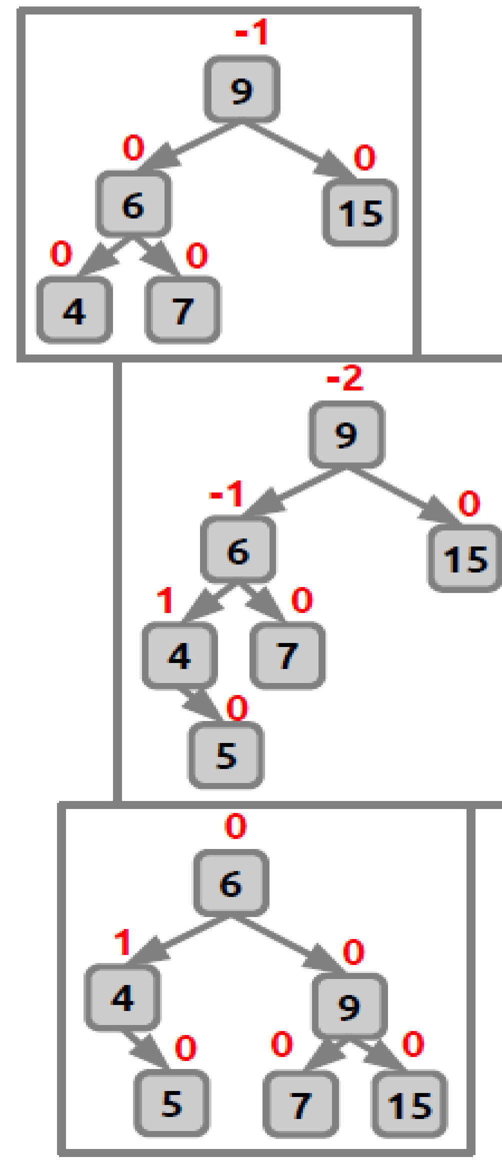
```
inserir(4)

inserir(6)

inserir(8)

inserir(10)

inserir(12)

inserir(16)
```



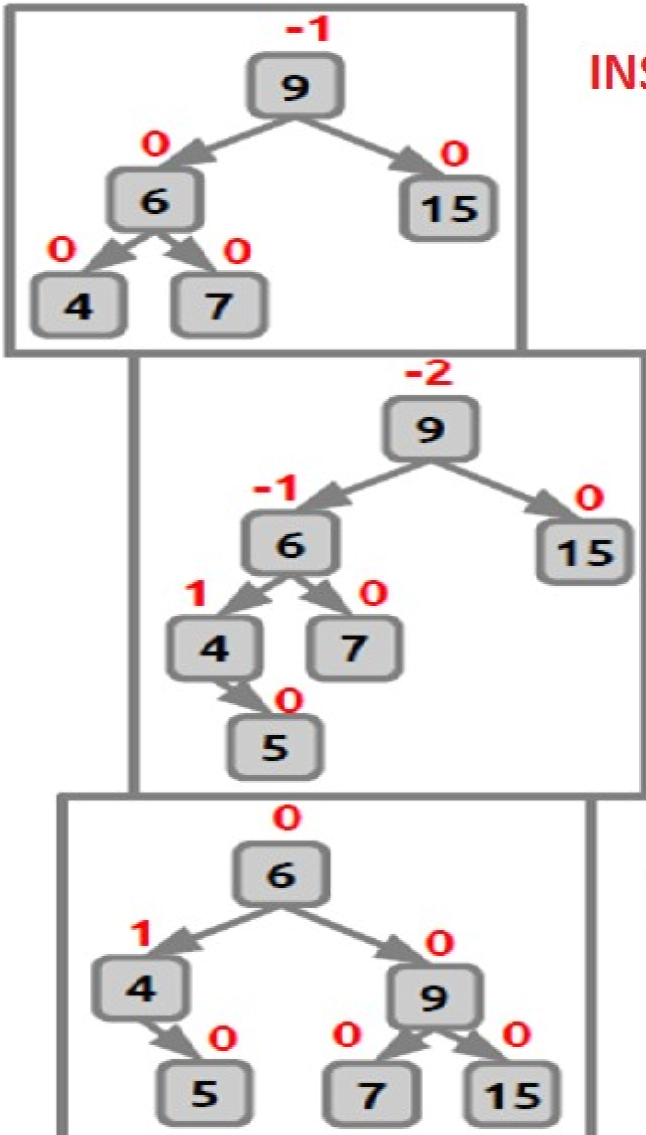
# Desbalanceamento

Inserir como uma árvore BB;

- Na “volta” da recursão verificamos nós que violam o balanceamento
- AVL, pois apenas sub-árvores que contém o elemento inserido mudaram de altura;
- Correção dos nós desbalanceados (só podem ser os ancestrais)
  - Identificação dos 3 nós “primários”
  - Casos: EE, DD, ED e DE



# Desbalanceamento




INSERIR O NÚMERO 5

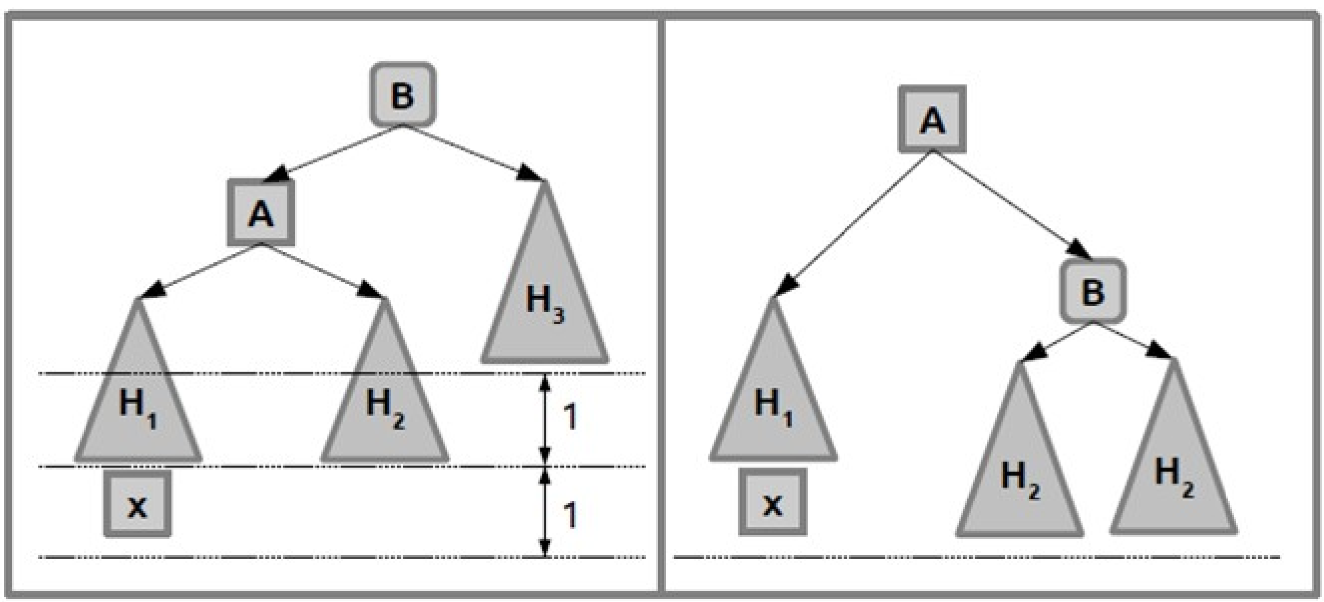
FICOU DESBALANCEADO

BALANCEADO

# AVL - Inserção

- Caso EE:

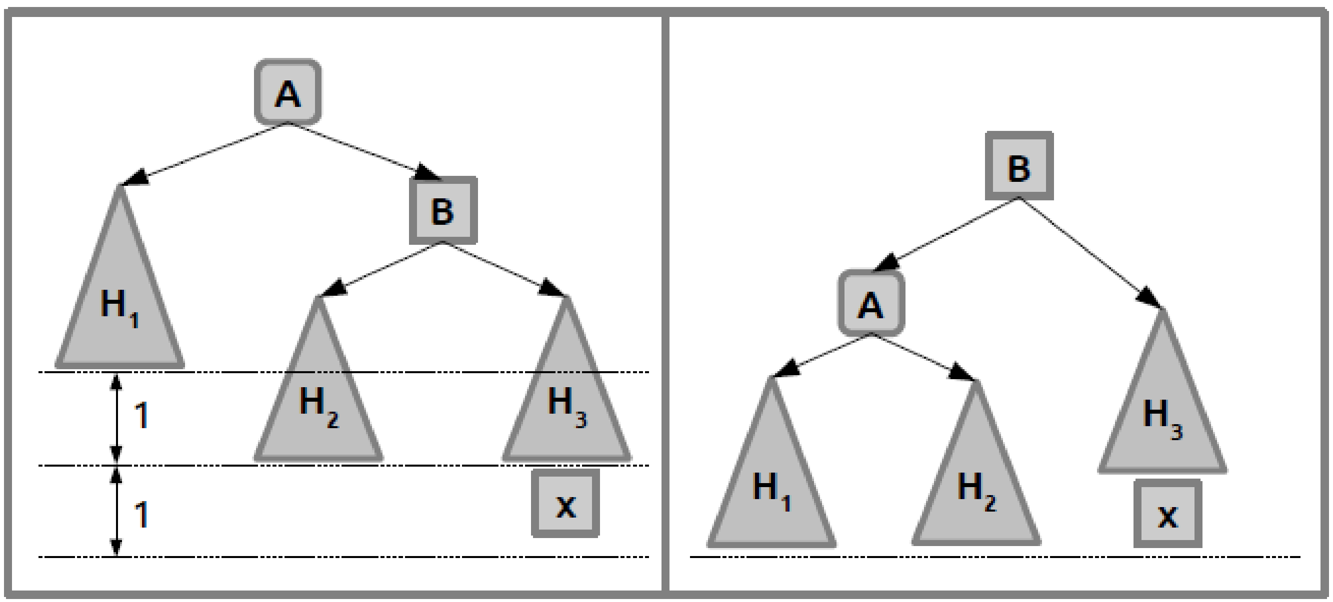
```
node<T>& Tree::rotEE( node<T> &n ) {
    node<T> &A = n.left;
    n.left = A.right;
    A.right = n;
    return A;
}
```



# AVL - Inserção

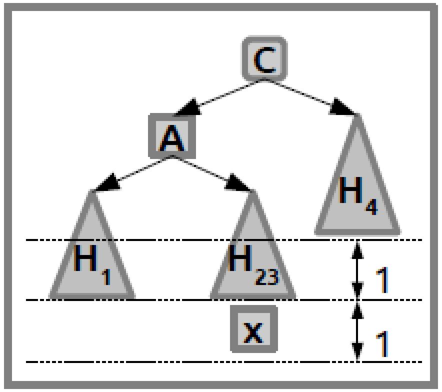
- Caso DD:

```
node<T>& Tree::rotDD( node<T> &n ) {
    node<T> &B = n.right;
    n.right = B.left;
    B.left = n;
    return B;
}
```

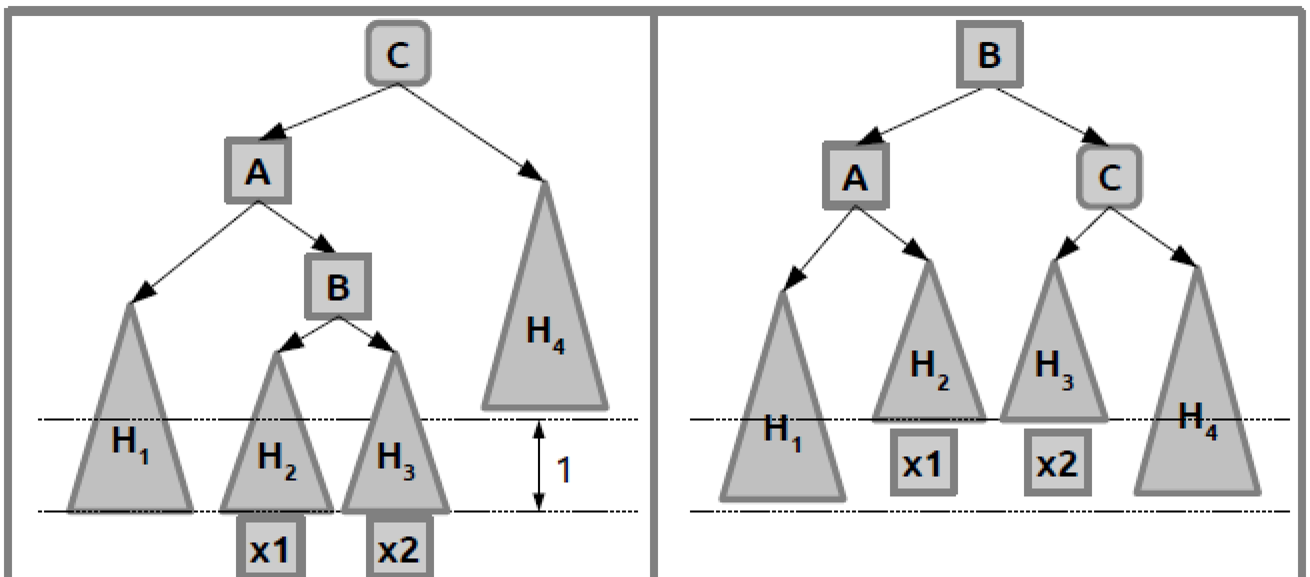


# AVL – Inserção

- Caso ED:

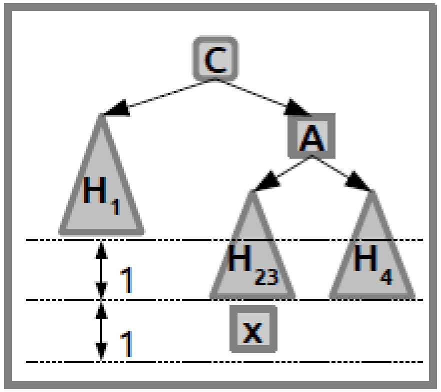



```
node<T>& Tree::rotED( node<T> &n ) {
    node<T> &A = n.left;
    node<T> &B = A.right;
    A.right = B.left;
    B.left = A;
    n.left = B.right;
    B.right = n;
    return B;
}
```

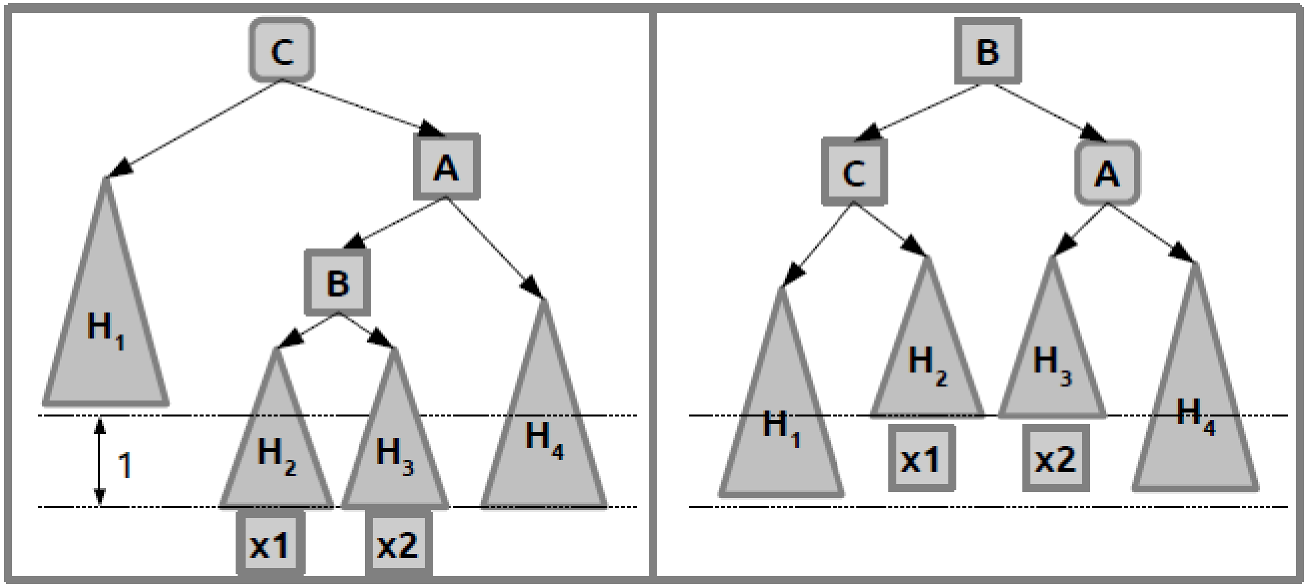


# AVL – Inserção

- Caso DE:




```
node<T>& Tree::rotDE( node<T> &n ) {
    node<T> &A = n.right;
    node<T> &B = A.left;
    A.left = B.right;
    B.right = A;
    n.right = B.left;
    B.left = n;
    return B;
}
```



# AVL - Inserção

```
node<T>& Tree::insert( node<T>& n, T key, bool &mudouAltura)  {
        if( !n)  {
                node<T>& novo = new node(key);
                novo.bal = 0;  mudouAltura = true;
                return novo;
        }
        if( key < n.key) {
                        n.left = insert(  n.left,  key, mudouAltura);
                        if( mudouAltura ) {
                                if( n.bal == 1 ) {
                                        n.bal = 0;    mudouAltura = false;
                                } else if( n.bal == 0 ) {
                                        n.bal = -1; // mudouAltura continua true
                                } else if( n.bal == -1 ) {
                                        node<T> &filho = n.left;
                                        if( filho.bal == -1 ) return rotEE(n,mudouAltura);
                                        else return rotED(n,mudouAltura);
                                }
                        }
        } else if ( key > n.key) {
                n.right = insert( n.right, key, mudouAltura);
                        if( mudouAltura ) {
                                if( n.bal == -1 ) {
                                        n.bal = 0;    mudouAltura = false;
                                } else if( n.bal == 0 ) {
                                        n.bal = 1; // mudouAltura continua true
                                } else if( n.bal == 1 ) {
                                        node<T> &filho = n.right;
                                        if( filho.bal == 1 ) return rotDD(n,mudouAltura);
                                        else  return rotDE(n,mudouAltura);
                                }
                        }
        } else { // key já estava na árvore
                mudouAltura = false;
                return n;
        }
}
```

# AVL - Inserção

```
node<T>& Tree::rotEE( node<T> &n, bool &mudouAltura ) {
    node<T> &A = n.left;
    n.left = A.right;
    A.right = n;
    A.bal = 0;
    n.bal = 0;
    mudouAltura = false;
    return A;
}
```

```
node<T>& Tree::rotED( node<T> &n, bool &mudouAltura  ) {
    node<T> &A = n.left;
    node<T> &B = A.right;
    A.right = B.left;
    B.left = A;
    n.left = B.right;
    B.right = n;
    if( B.bal == -1 ) {
        n.bal = 1; A.bal = 0;
    } else if( B.bal == 1 ) {
        n.bal = 0; A.bal = -1;
    } else { // B.bal == 0
        n.bal = 0; A.bal = 0;
    }
    B.bal = 0;
    mudouAltura = false;
    return B;
}
```

# AVL - Inserção

```
node<T>& Tree::rotDD( node<T> &n, bool &mudouAltura ) {
    node<T> &B = n.right;
    n.right = B.left;
    B.left = n;
    B.bal = 0;
    n.bal = 0;
    mudouAltura = false;
    return B;
}
```

```
node<T>& Tree::rotED( node<T> &n, bool &mudouAltura  ) {
    node<T> &A = n.left;
    node<T> &B = A.right;
    A.right = B.left;
    B.left = A;
    n.left = B.right;
    B.right = n;
    if(  B.bal == -1 ) {
        n.bal = 1; A.bal = 0;
    } else if( B.bal == 1 ) {
        n.bal = 0; A.bal = -1;
    } else { // B.bal == 0
        n.bal = 0; A.bal = 0;
    }
    B.bal = 0;
    mudouAltura = false;
    return B;
}
```

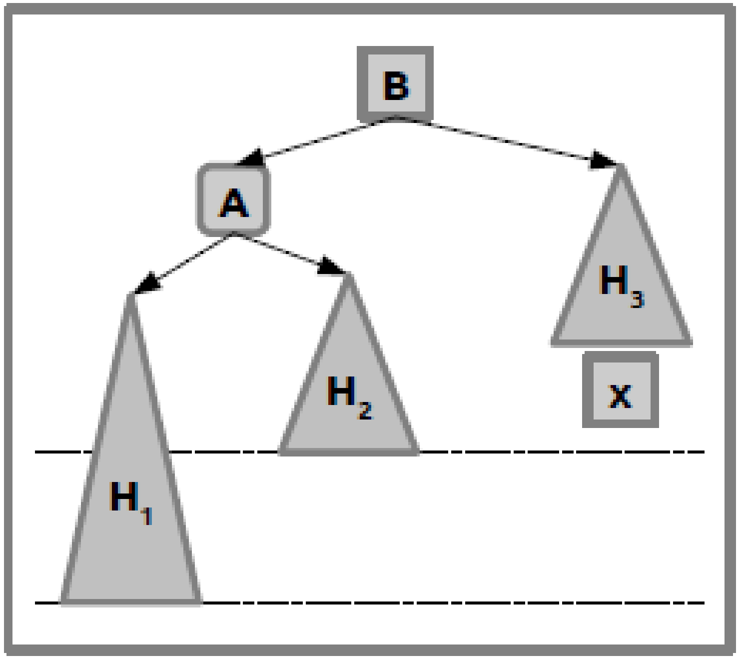
# AVL - Remoção

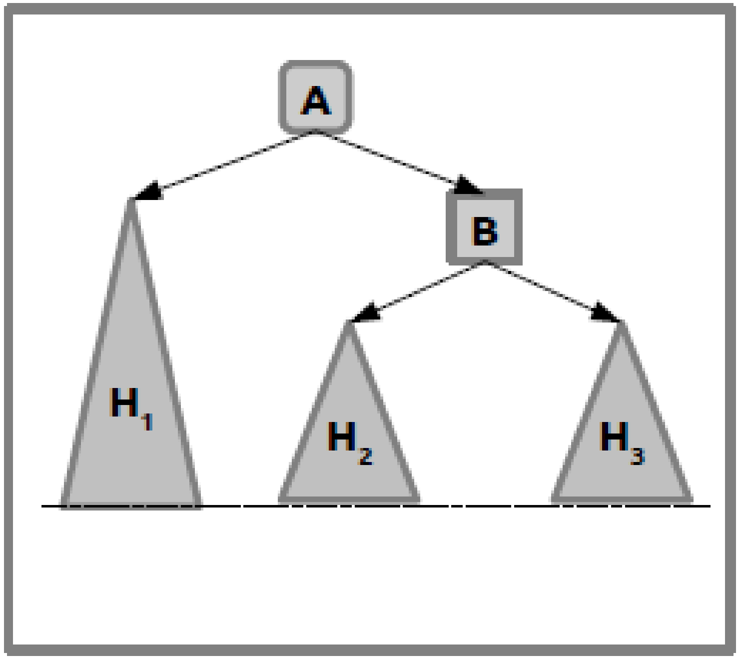
Remover como em uma arvore BB;

- Na “volta” da recursão verificamos nos que violam o balanceamento AVL;
- Correção dos nos desbalanceados (só podem ser os ancestrais)
- Diferenças com a inserção:
  - Uma situação a mais no EE e no DD
  - DE e ED alteram a altura da sub-árvore

# AVL - Remoção

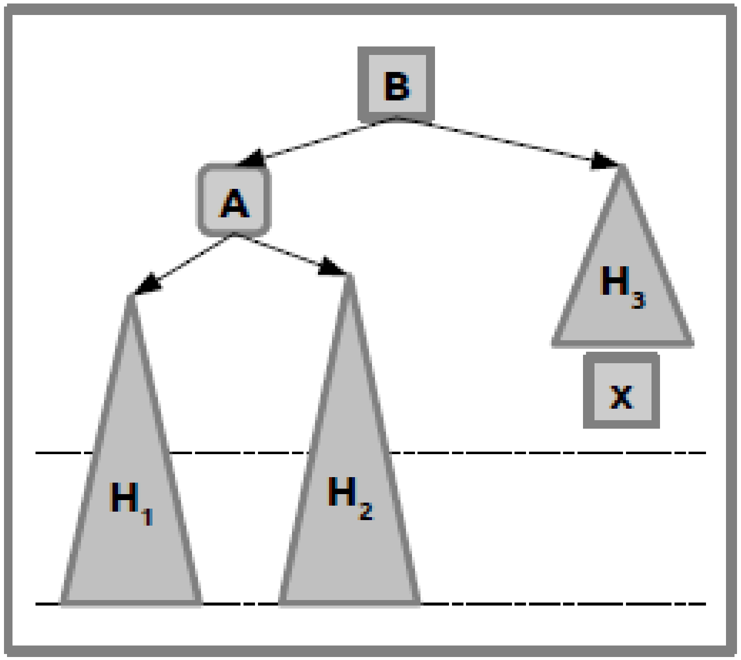
- Caso EE:

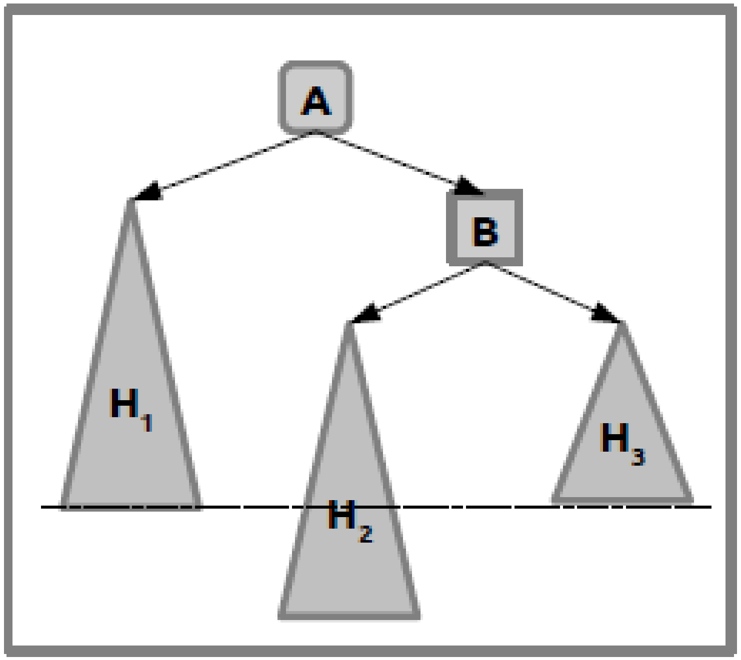


# AVL - Remoção

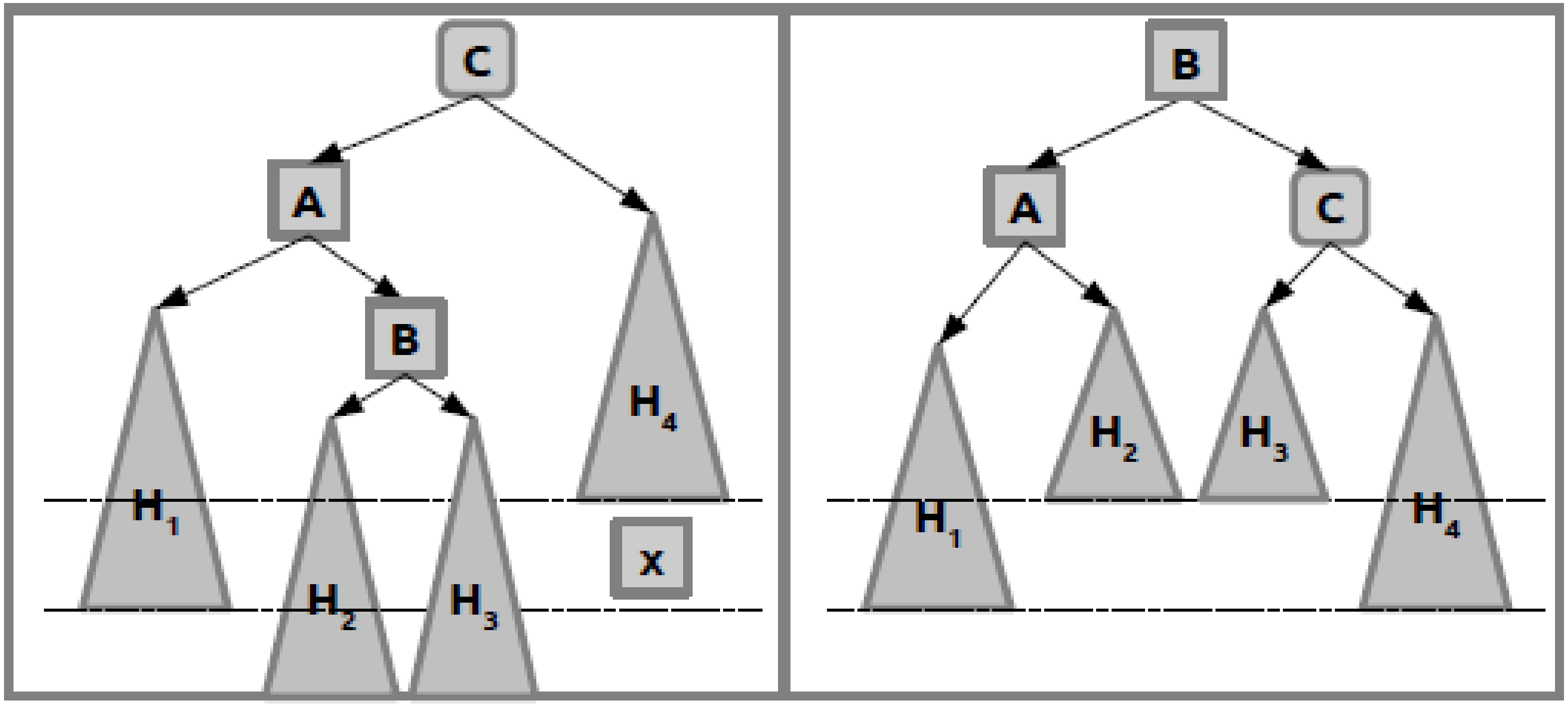
- Caso EE:



# AVL - Remoção

- Caso ED:



# AVL
# Remoção

```
node<T>* Tree::remove( node<T>* n, T key, bool &mudouAltura)  {
   if( !n)
      mudouAltura = false; //  não achou
   else if( key < n->key) {
      n->left = remove(  n->left,  key, mudouAltura);
      if( mudouAltura ) {
         if( n->bal == -1 ) {
            n->bal = 0;     // mudouAltura continua true
         } else if( n->bal == 0 ) {
            n->bal = 1;     mudouAltura = false;
         } else if( n->bal == 1 ) {
            node<T> *filho = n->right;
            if( filho->bal == -1 ) return rotDEremove(n,mudouAltura);
            else return rotDDremove(n,mudouAltura);
         }
      }
   } else if ( key > n->key) {
      n->right = remove( n->right, key, mudouAltura);
      if( mudouAltura ) {
         if( n->bal == 1 ) {
            n->bal = 0;     // mudouAltura continua true
         } else if( n->bal == 0 ) {
            n->bal = -1;    mudouAltura = false;
         } else if( n->bal == -1 ) {
            node<T> *filho = n->left;
            if( filho->bal == 1 ) return rotEDremove(n,mudouAltura);
            else       return rotEEremove(n,mudouAltura);
         }
      }
   } else {   // achou remover
```

# AVL
# Remoção

```
    if( !n->right && !n->left ) { // sem filhos
      mudouAltura = true;
      delete n;
      return 0;
    } else if( n->right && n->left ) { // dois filhos
      n->key = max( n->left )->key;
      n->left = remove(n->left, n->key, mudouAltura );
      if( mudouAltura ) {
        if( n->bal == -1 ) {
          n->bal = 0;  // mudouAltura continua true
        } else if( n->bal == 0 ) {
          n->bal = 1;          mudouAltura = false;
        } else if( n->bal == 1 ) {
          node<T> *filho = n->right;
          if( filho->bal == -1 ) return rotDEremove(n,mudouAltura);
          else return rotDDremove(n,mudouAltura);
        }
      }
    } else { // um filho
      mudouAltura = true;
      node<T> *filho;
      if( n->left )
        filho = n->left;
      else
        filho = n->right;
      delete n;
      return filho;
    }
  }
  return n;
}
```

- M. T. GOODRICH et al
  - Estruturas de Dados e Algoritmos em Java
  - Data Structures and Algorithms in Java
  - Data Structures and Algorithms in C++
  - Lectures Slides
- H. T. CORMEN et al
  - Introduction to Algorithms
  - Algoritmos - Teoria e Prática
- R. Sedgewick et al
  - An Introduction of the Analysis of Algorithms
  - Algorithms in C++
  - Lectures Slides
- Roberto Ferrari
  - Livro Virtual: (http://www2.dc.ufscar.br/~ferrari/ed/ed.html)