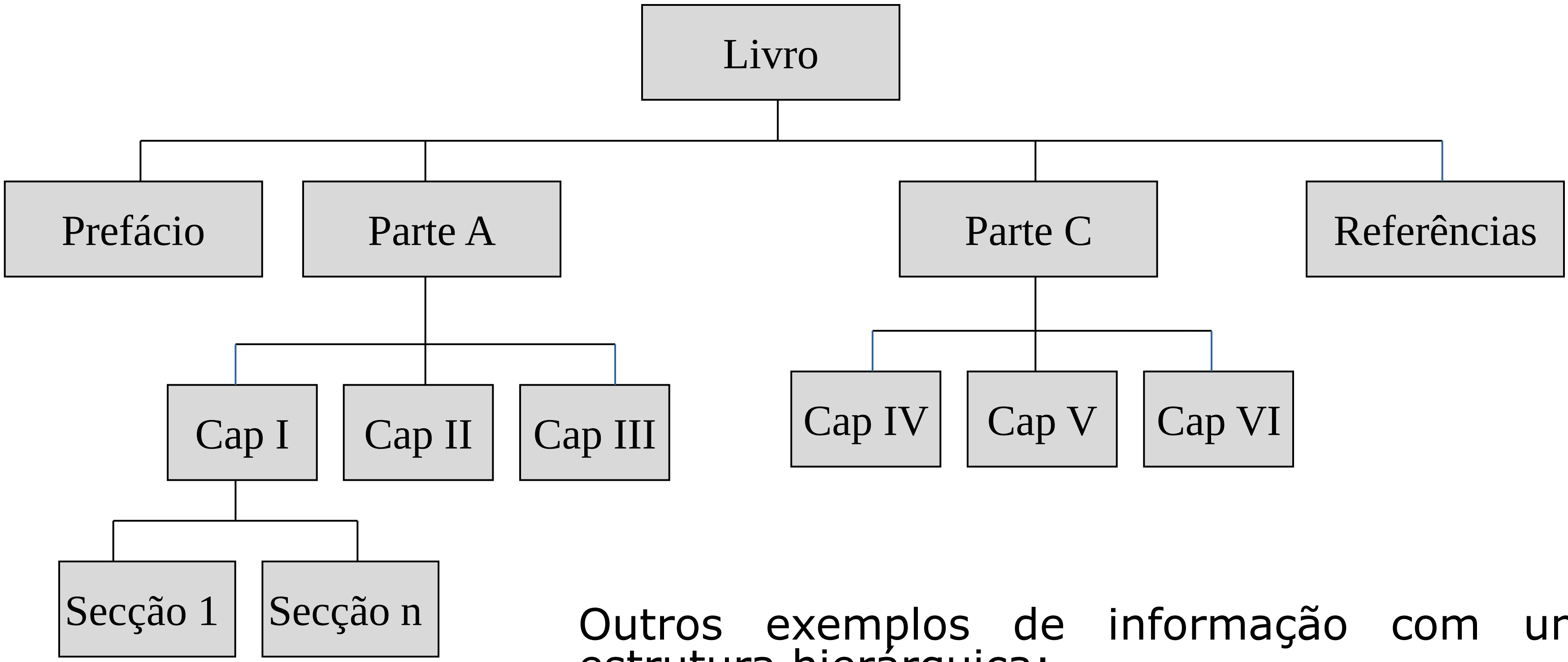
Estruturas de Informação

Árvores

Departamento de Engenharia Informática (DEI/ISEP)
Fátima Rodrigues
mfc@isep.ipp.pt

Árvores

São estruturas de dados cujos elementos apresentam uma relação hierárquica



Outros exemplos de informação com uma estrutura hierárquica:

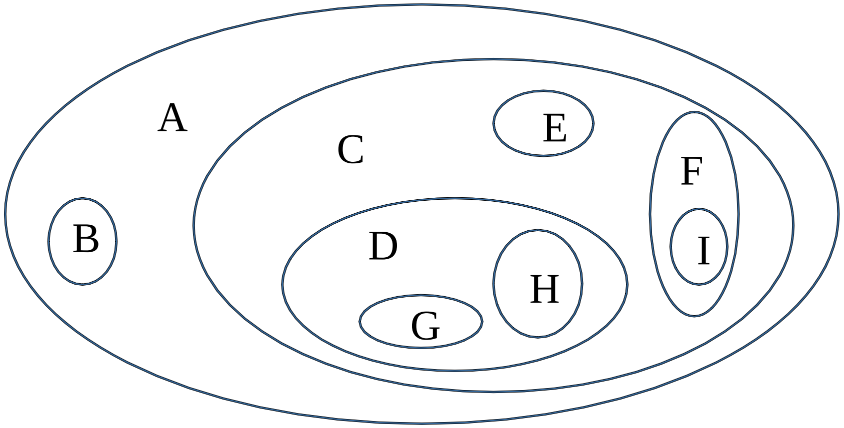
- organigrama de uma empresa
- árvore genealógica
- codificação de produtos

Árvores – Formas de Representação

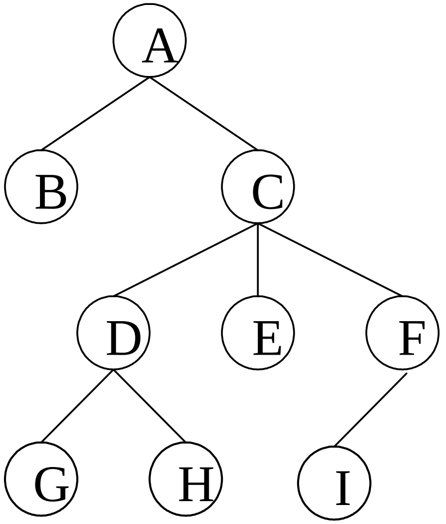
Representação por parêntesis aninhados

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

Diagrama de inclusão



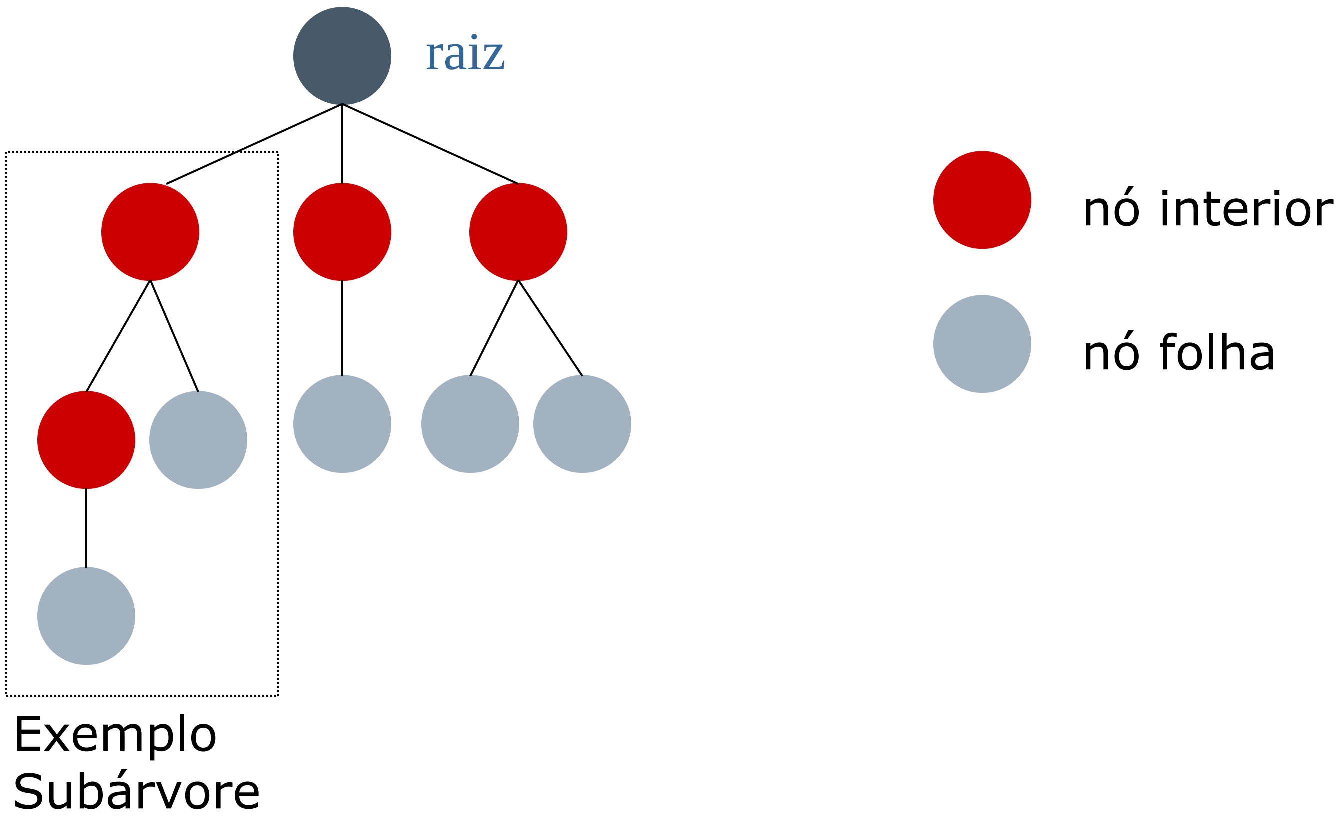
Representação hierárquica



Definição e Terminologia

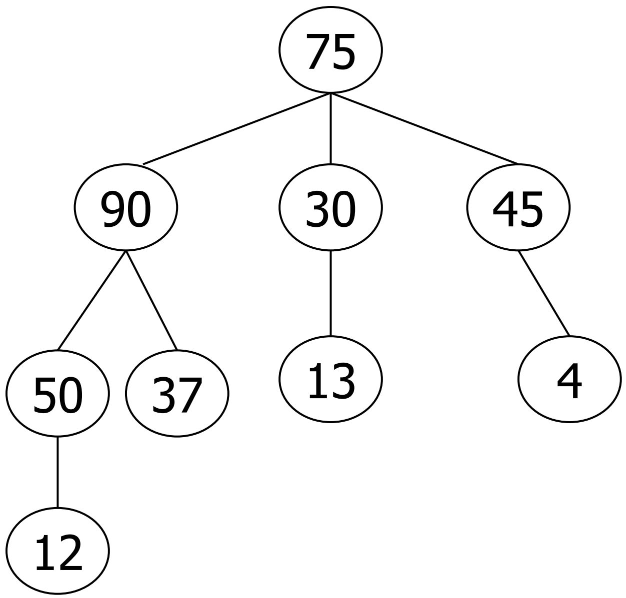
- Árvore é um tipo abstracto de dados que guarda os elementos -**nós** hierarquicamente
- Na representação gráfica de árvores os **nós** ligam-se por **ramos** – aresta orientada entre dois nós
- Com excepção do elemento de **topo** cada elemento tem um elemento pai e zero ou mais elementos filhos
- O elemento de topo é designado por **raiz** **-** não tem elemento pai nem ascendentes
- Os elementos que não possuem filhos são designados por **folhas**
- Os outros nós da árvore dizem-se **interiores** – nó com pelo menos um filho
- Por sua vez cada elemento numa árvore é a raiz da **subárvore** que é definida pelo nó e todos os descendentes do nó

Definição e Terminologia



Quantas subárvores existem na árvore representada ?

Ascendentes e Descendentes de um nó



Ascendentes de um nó X são todos os nós que existem no caminho desde esse nó até à raiz. Ascendentes do nó 12: 50, 90 e 75

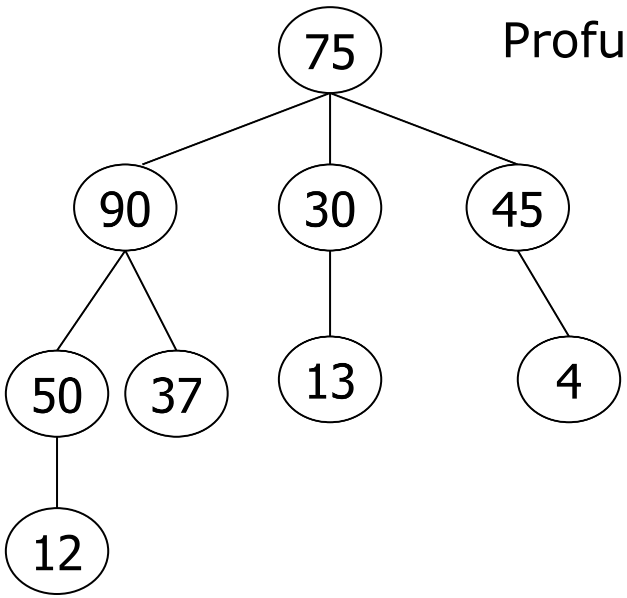
Descendentes de um nó X são todos os nós alcançáveis a partir desse nó Descendentes do nó 90: 50, 37 e 12

O movimento de um nó para os seus descendentes faz-se através de um único caminho

Profundidade de um nó

Profundidade de um nó é o número de ramos existentes no caminho entre o nó e a raiz

$$\text{Profundidade do nó (X)} = \begin{cases} 0 & \text{se X é } \textbf{raiz} \\ 1 + \text{Profundidade do nó (pai(X))} & \text{para os outros nós} \end{cases}$$

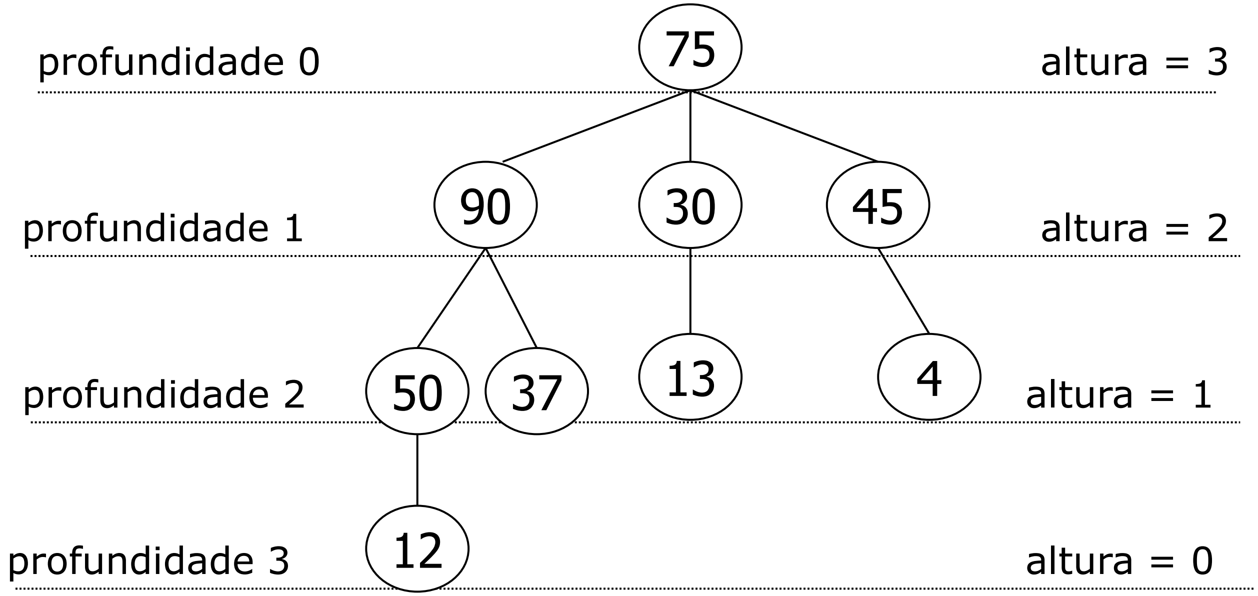


Profundidade da raiz → 0

Profundidade do nó 13 → 2

Altura de uma árvore

Altura de uma árvore é a máxima profundidade apresentada pelos seus nós



Altura de uma árvore é a altura da raiz

Altura da árvore → 3

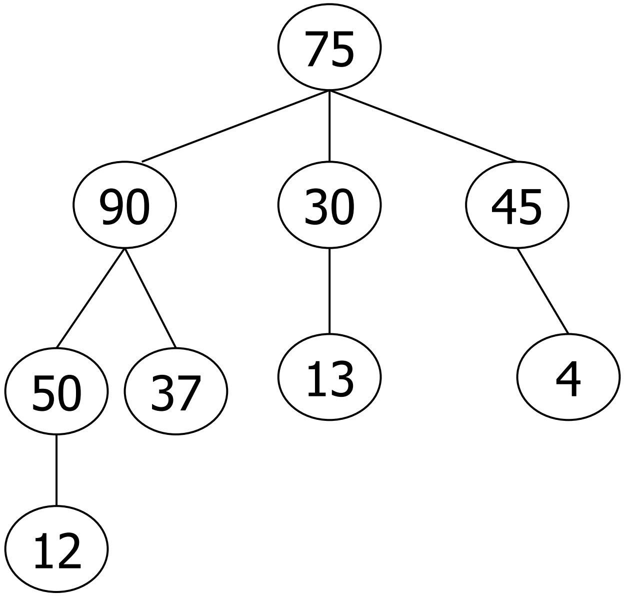
Grau de uma árvore

Grau de um nó é o número de filhos que esse nó tem

Grau do Nó 90 → 2

Grau de uma árvore é o máximo grau dos seus nós

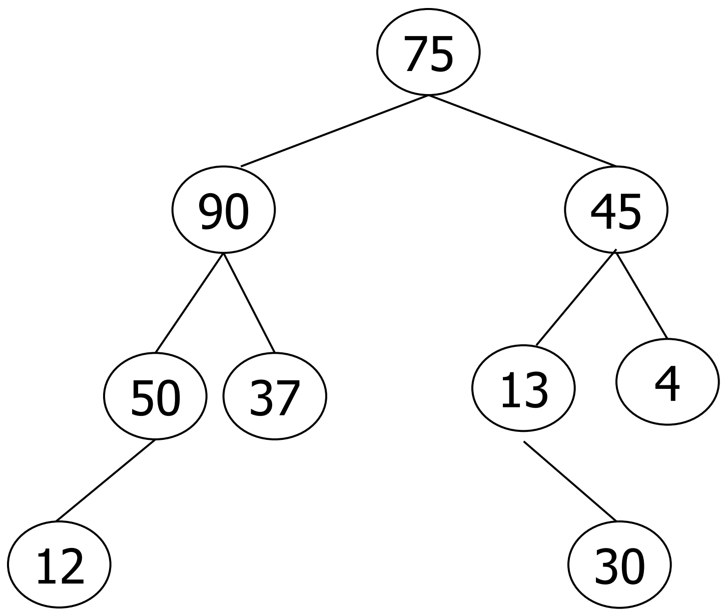
Um nó folha tem grau → 0



Grau da árvore → 3

Árvores Binárias

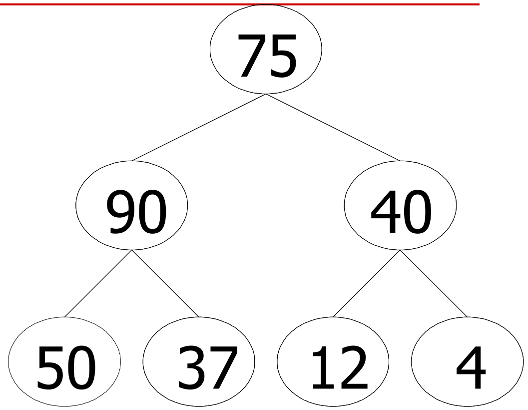
Árvores Binárias



São árvores que ou são nulas ou são constituídas por um nó raiz e duas subárvores binárias: subárvore esquerda e subárvore direita

Por sua vez cada uma destas subárvores ou são nulas ou são constituídas por um nó raiz e duas subárvores binárias: subárvore esquerda e subárvore direita, ...

Propriedades Árvores Binárias



- Uma árvore binária com **n** nós tem **n - 1** ramos
- Uma árvore binária de altura **h** tem:
  - no mínimo **h+1** elementos
  - no máximo $\mathbf{2^{h+1} - 1}$ elementos
- Uma árvore binária com $\mathbf{2^{h+1} - 1}$ elementos diz-se **completamente cheia**
- Numa árvore binária **completamente cheia** o número de **nós folha** é igual ao número de **nós internos + 1**
- A altura de uma árvore binária com **n** elementos (**n > 0**) é
  - no máximo n-1
  - no mínimo $\log_2(n+1) - 1$

$$n = 2^{h+1} - 1$$
$$2^{h+1} = n + 1$$
$$\log_2 (2^{h+1}) = \log_2 (n + 1)$$
$$h = \log_2 (n + 1) - 1$$

Métodos de Travessia

Existem várias maneiras de percorrer ou visitar todos os nós de uma árvore binária de forma sistemática

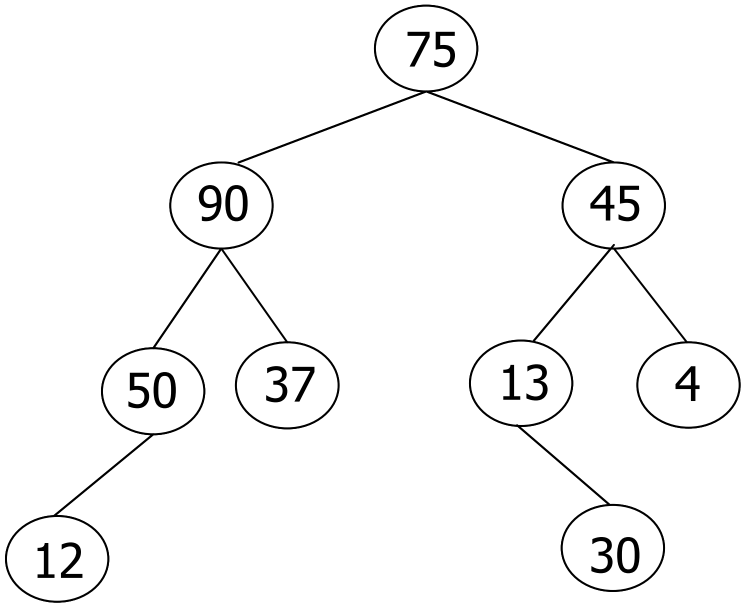
- Visita em ordem (ou simétrica) (Esq, Raiz, Dir)
- Visita em preordem (Raiz, Esq, Dir)
- Visita em posordem (Esq, Dir, Raiz)
- Visita por nível

os nomes das visitas são relativos à raiz

As versões recursivas destas visitas são extremamente simples devido à própria natureza recursiva da estrutura árvore binária

Os algoritmos iterativos necessitam de uma stack auxiliar

Métodos de Travessia



- Visita em ordem: 12, 50, 90, 37, 75, 13, 30, 45, 4
- Visita em pre-ordem: 75, 90, 50, 12, 37, 45, 13, 30, 4
- Visita em pos-ordem: 12, 50, 37, 90, 30, 13, 4, 45, 75
- Visita por nível: 75, 90, 45, 50, 37, 13, 4, 12, 30

Visita em Ordem

Corresponde a visitar simetricamente a subárvore esquerda seguida da visita ao nó raiz, seguida da visita simétrica à subárvore direita

Algoritmo Recursivo

```
Visita-emOrdem (raiz)
  Se (raiz /= Null)
    Visita-emOrdem (subárv esquerda)
    imprime Nó
    Visita-emOrdem (subárv direita)
  Fse
Fim Visita-emOrdem
```

Algoritmo iterativo

```
Visita-emOrdem (raiz)
  r = raiz
  Repetir
    Enqto (r /= Null)
      s.push(r)
      r = r→esq
    FEnqto
    Se (stack /= vazia)
      escreve s.top()
      s.pop()
      r = r→dir
    Fse
  Até (s.vazia() ∧ r==Null)
Fim Visita-emOrdem
```

Visita Pre-ordem

Corresponde a visitar o nó, seguido das visitas em pre-ordem da subárvore esquerda e da subárvore direita

Algoritmo Recursivo

```
Visita-preOrdem (raiz)
   Se (raiz /= Null)
     imprime Nó
     Visita-preOrdem (subárv esquerda)
     Visita-preOrdem (subárv direita)
   Fse
Fim Visita-preOrdem
```

Algoritmo iterativo

```
Visita-preOrdem (raiz)
 r = raiz
 Repetir
   Enqto (r /= Null)
        escreve r->key
        s.push(r)
        r=r→esq
   FEnqto
   se (stack /= vazia)
        s.pop()
        r=r→dir
   Fse
 Até (s.vazia() ∧ r==Null)
Fim Visita-PreOrdem
```

Visita Pos-ordem

Corresponde a visitar o nó depois de ter feito a visita em pos-ordem à subárvore esquerda e a visita em pos-ordem à subárvore direita

Algoritmo Recursivo

```
Visita-posOrdem (raiz)
        Se (raiz /= Null)
            Visita-posOrdem(subárv esquerda)
            Visita-posOrdem(subárv direita)
            imprime Nó
        Fse
Fim Visita-posOrdem
```

Algoritmo iterativo

- Só poderá ser feito o pop definitivo de um elemento da stack depois de ter sido feita a visita em posordem às subárvores esq^{da} e dir^{ta}
- Para além da stack de apontadores com os nós visitados é necessário controlar para cada nó se já foram visitadas as duas subárvores esq^{da} e dir^{ta}

Visita Pos-ordem Iterativo

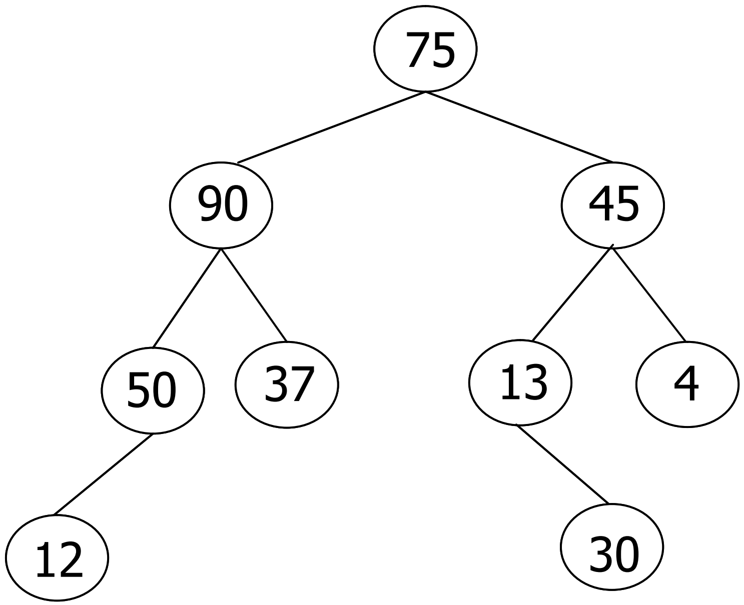
```
Visita-posOrdem (raiz)
   rz = raiz
   s1.push(rz) s2.push(0) ;
   Enqto (!s1.vazia())
       rz = s1.top()
       se (s2.top() == 0){
         s2.pop(); s2.push(1);
         se (rz→esq \= Null){
           rz = rz→esq ;
           s1.push(rz) ; s2.push(0); }}
       se (s2.top() == 1){
           s2.pop(); s2.push(2);
           se (rz→dir \= Null) {
             rz = rz→dir ;
             s1.push(rz) ; s2.push(0); }}
       se (s2.top() == 2){
         s1.pop(rz); s2.pop(freq)
         escreve rz→key ; }
   FEnqto
Fim Visita-posOrdem
```

Visita Por Níveis

Corresponde a visitar os nós da árvore nível a nível

```
Visita-porNiveis
  r = raiz
  se (r /= Null)
    junta-fila(r)
  Fse
  Enqto (fila /= vazia)
    retira-fila(r)
    imprime r→key
    se (r→esq /= Null)
      junta-fila(r→esq)
    Fse
    se (r→dir /= Null)
       junta-fila(r→dir)
    Fse
  FEnqto
Fim Visita-porNiveis
```

Pesquisar um Elemento



Qual a complexidade?

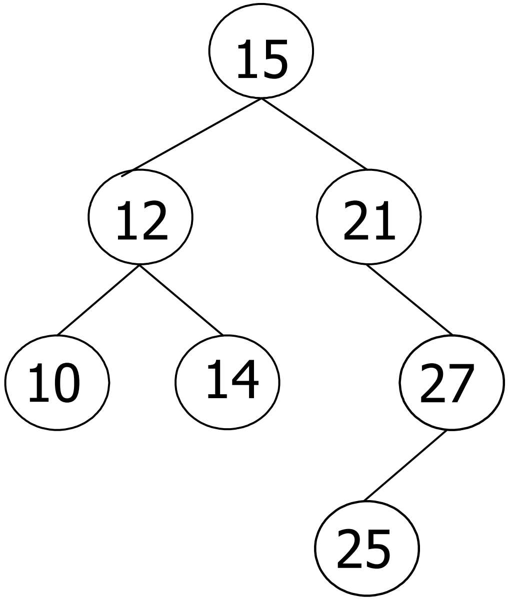
Árvores Binárias de Pesquisa

(ou de Busca)

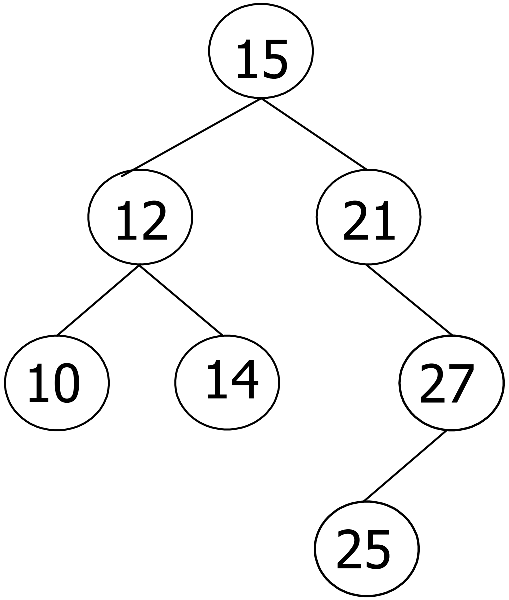
Árvore Binária de Pesquisa - Definição

É um tipo especial de árvore binária em que cada nó tem obrigatoriamente **um campo chave (único)** que verifica a seguinte propriedade. Para cada nó:

- todas as chaves da sub-árvore esquerda são menores que a chave desse nó
- todas as chaves da sub-árvore direita são maiores que a chave desse nó



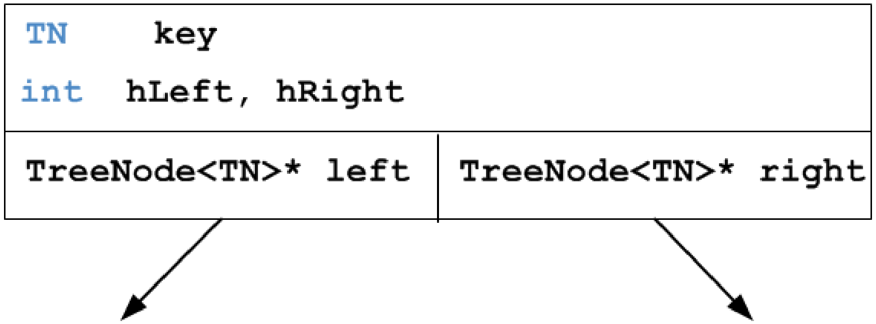
Árvore Binária de Pesquisa



- Visita em ordem: 10, 12, 14, 15, 21, 25, 27
- Visita em pre-ordem: 15, 12, 10, 14, 21, 27, 25
- Visita em pos-ordem: 10, 14, 12, 25, 27, 21, 15
- Visita por nível: 15, 12, 21, 10, 14, 27, 25

A **visita em ordem** dá-nos os valores das chaves da árvore binária de pesquisa **ordenados de forma crescente**

Classe Template treeNode

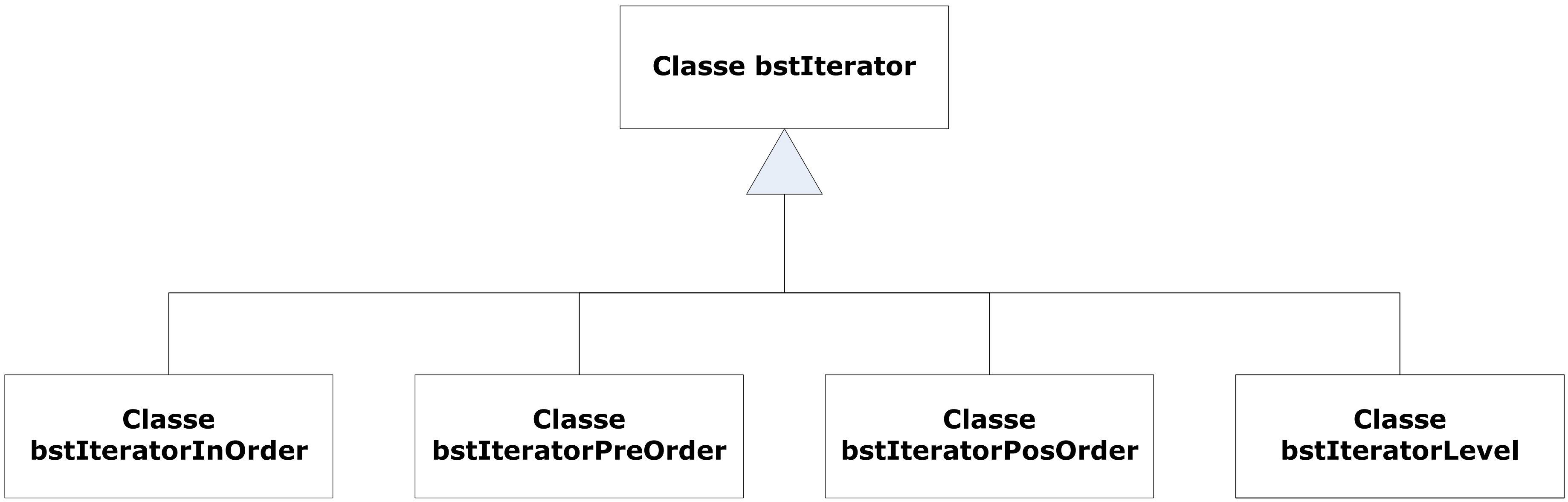



```
class treeNode
{
  private:
     TN key;
     treeNode <TN> *left, *right;
     int hLeft, hRight;

  public:
     treeNode();
     treeNode(const TN& key, int hLeft=0, int hRight=0,
              treeNode<TN>* left=NULL, treeNode<TN>* right=NULL );
     treeNode(const treeNode<TN>& n);
     ~treeNode();

     TN getKey() const;
     treeNode<TN>* getLeft() const;
     void setLeft(treeNode <TN>* left);

     treeNode<TN>* getRight() const;
     void setRight(treeNode <TN>* right);
```

Classe Template treeNode

```
    int getLeftHeight() const;
    void setLeftHeight(int lh);
    void setLeftSide(treeNode<TN>* left, int lh);

    int getRightHeight() const;
    void setRightHeight(int rh);
    void setRightSide(treeNode<TN>* right, int rh);

    int getBalanceFactor() const ;
};
```

Classes para Iterar Árvore



Classe Template bstIterator

```
class bstIterator
{
  protected:
     treeNode<TN>* now;

  public:
     bstIterator();
     bstIterator(treeNode<TN>* now);
     bstIterator(const bstIterator& i);

     TN operator*() const ;
     TN operator->() const ;
     bool operator == (const bstIterator<TN>& i) const ;
     bool operator != (const bstIterator<TN>& i) const ;

     virtual bstIterator<TN>& operator = (const bstIterator<TN>& i);
     virtual bstIterator<TN>& operator++ (int) { return *this;}
};
```

Classe Template bstIteratorInOrder

```
class bstIteratorInOrder : public bstIterator<TN> {
  private:
     stack <treeNode<TN>* > unVisited;

     void findFirstElement();

  public:
     bstIteratorInOrder ();
     bstIteratorInOrder (const bstIterator<TN>& i);
     bstIteratorInOrder (const bstIteratorInOrder<TN>& i);

     bstIterator<TN>& operator++ (int) ;
     bstIterator<TN>& operator = (const bstIterator<TN>& i);
};
```

Classe Template bstIteratorPreOrder

```
class bstIteratorPreOrder : public bstIterator<TN> {
  private:
     stack <treeNode<TN>* > unVisited;

  public:
     bstIteratorPreOrder ();
     bstIteratorPreOrder (const bstIterator<TN>& i);
     bstIteratorPreOrder (const bstIteratorPreOrder<TN>& i);

     bstIterator<TN>& operator++ (int) ;
     bstIterator<TN>& operator = (const bstIterator<TN>& i);
};
```

Classe Template bstIteratorPosOrder

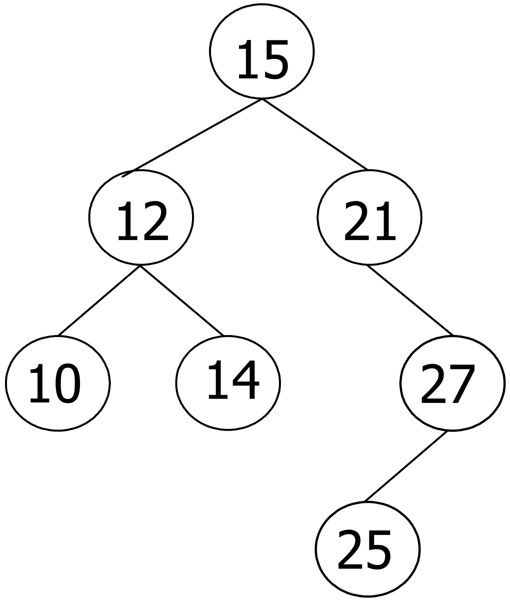
```
 class bstIteratorPosOrder : public bstIterator<TN> {
  private:
     stack <treeNode<TN>*> unVisited;
     stack <bool>  unVisitedRight;

     void findDeepestPosOrder();

   public:
     bstIteratorPosOrder ();
     bstIteratorPosOrder (const bstIterator<TN>& i);
     bstIteratorPosOrder (const bstIteratorPosOrder<T>& i);

     bstIterator<TN>& operator++ (int) ;
     bstIterator<TN>& operator = (const bstIterator<TN> &i);
};
```

Classe Template bstIteratorLevel

```
class bstIteratorLevel : public bstIterator<TN> {
  private:
     queue <treeNode<TN>* > unVisited;

  public:
     bstIteratorLevel ();
     bstIteratorLevel (const bstIterator<TN>& i);
     bstIteratorLevel (const bstIteratorLevel<TN>& i);

     bstIterator<TN> & operator++ (int) ;
     bstIterator<TN> & operator = (const bstIterator<TN>& i);
};
```

Pesquisar um elemento na árvore

Qual a complexidade ?



Análise de Complexidade

O número máximo de comparações que se faz até se concluir se existe ou não a chave, é no máximo a altura da árvore h → h + 1

Se a árvore for (**mais ou menos**) **equilibrada**, **os nós folha todos com a mesma profundidade**, podemos relacionar a altura da árvore com o total de elementos n

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

$$2^{(h+1)} = n + 1$$

$$h+1 = \log_2 (n+1)$$

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

assim podemos dizer que para todos os valores de $n \geq 1$, é possível encontrar uma constante C, tal que:

$$\log_2 (n+1) - 1 \leq C \times \log_2 n$$

Complexidade temporal do algoritmo de Pesquisa → **T(n) = O(log n)**

Pesquisar

Algoritmo Recursivo

```
Pesquisa (chave,rz)
   se (!rz)
      devolve falso
   senão
      se (rz→key == chave)
         devolve verd
      senão
         se (rz→key > chave)
            Pesquisa(chave,rz→esq)
         senão
            Pesquisa(chave,rz→dir)
         Fse
      Fse
   Fse
Fim Pesquisa
```

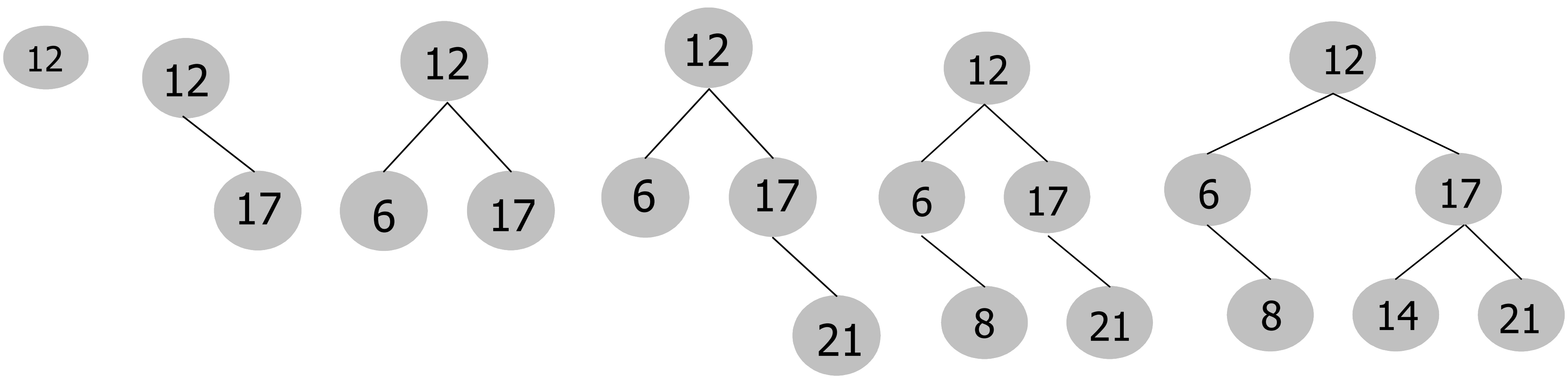
Algoritmo iterativo

```
Pesquisa (chave)
   temp = raiz
   enc = false
   Enqto(temp != null ∧ !enc)
      se (temp→key > chave)
         temp = temp→esq
      senão
         se (temp→key < chave)
            temp = temp→dir
         senão
            enc = true
         Fse
      Fse
   FEnqto
   devolve enc
Fim Pesquisa
```

Inserir

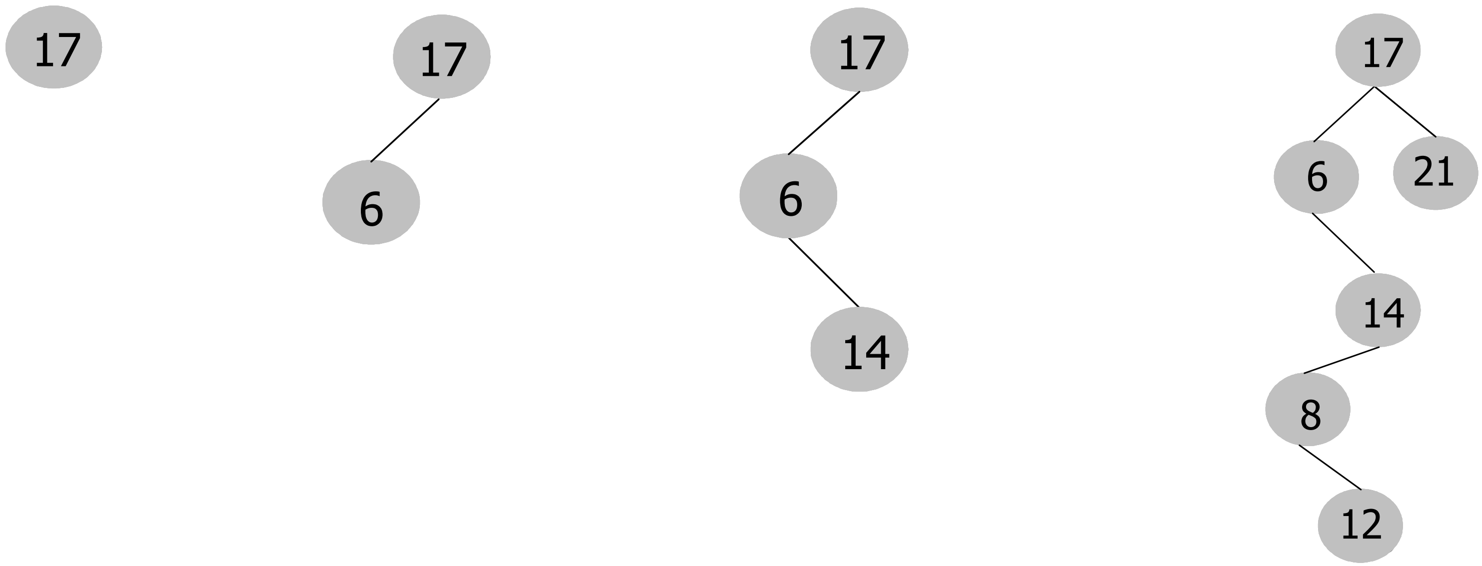
- Descer a árvore a partir da raiz, escolhendo sucessivamente a sub-árvore apropriadada. Ao chegar a uma folha, inserir do lado apropriado

- A forma da árvore depende da ordem de inserção dos elementos 12, 17, 6, 21, 8, 14



Inserir

- A forma da árvore depende da ordem de inserção dos elementos 17, 6, 14, 21, 8, 12



- O que acontece se os elementos forem inseridos na árvore por ordem crescente ou decrescente ?

Inserir Nó - Algoritmo Recursivo

```
Inserir (chave,rz)
  se (rz == null)
      rz = cria_no(chave)
      devolve rz
  Fse
  se (rz→key == chave)
      Elemento repetido
      devolve rz
  Fse
  se (rz→key > chave)
       rz→esq = Inserir (chave,rz→esq)
  senão
       rz→dir = Inserir (chave,rz→dir)

  devolve rz
Fim Inserir
```

Inserir Nó - Algoritmo Iterativo

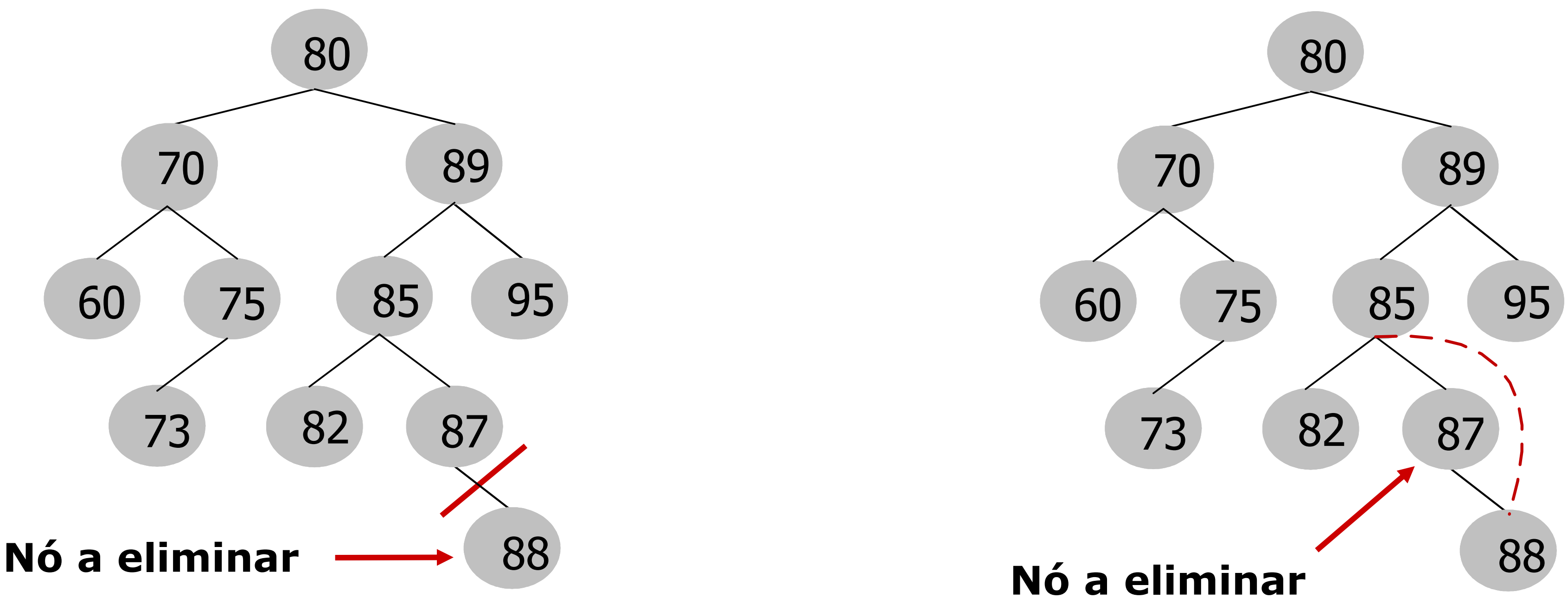
```
Inserir (chave)
  ant = temp = raiz
  enc = false
  novoNo = cria_no(chave)
  se (!raiz)
    raiz = novoNo ;
  senão
     Enqto (temp != null  ∧ !enc)
        se (temp→key > chave)
             ant = temp
             temp = temp → esq
        senão
            se (temp→key < chave)
               ant = temp
               temp = temp→dir
            senão
               enc = true
            Fse
        Fse
     FEnqto
     →
     se (!enc)
        se (ant→inf > chave)
           ant→esq = novoNo
        senão
           ant→dir = novoNo
        Fse
     Fse
  Fse
Fim Inserir
```

Remover

Na remoção de um nó de uma árvore binária de pesquisa vários casos podem acontecer. O nó a eliminar:

1. é folha (não tem as duas subárvores)
2. falta-lhe uma das subárvores
3. contem as duas subárvores

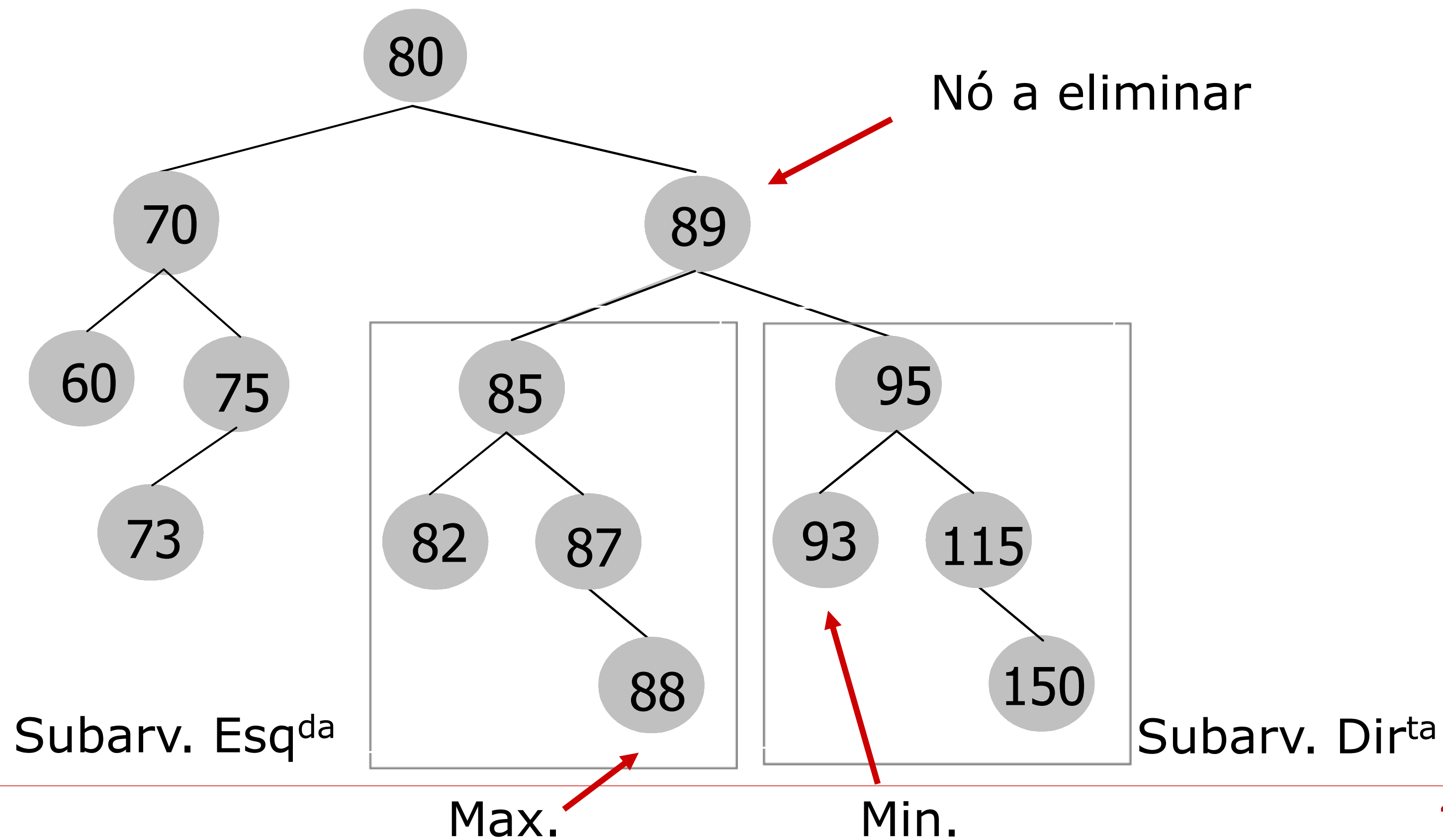
Os casos 1 e 2 são resolvidos ajustando o apontador do nó anterior (nó pai) que aponta para o nó que pretendemos eliminar



Remover

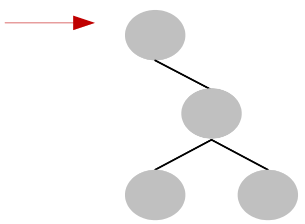
Caso 3:

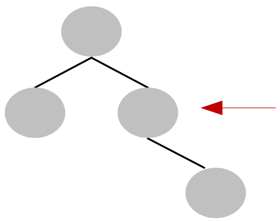
- substituir o nó a eliminar pelo **maior nó da subárvore esquerda** do nó a eliminar (nó imediatamente anterior na visita em ordem)
  ou
- substituir pelo **menor nó da subárvore direita** do nó a eliminar (nó seguinte à visita em ordem)



Remover – Algoritmo (1)

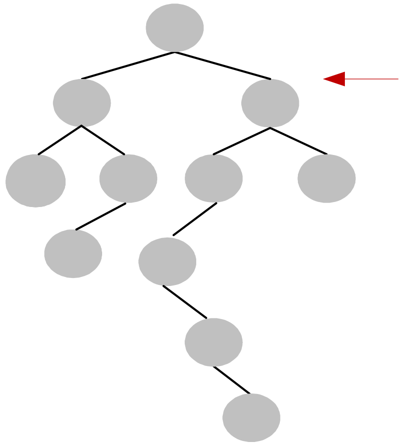
```
Remover (chave,rz,ant)
   Se (rz)
      Se (rz→key < chave)
         Remover (chave,rz→dir,rz)
      Senão
        Se (rz→key > chave)
           Remover (chave,rz→esq,rz) ;
        Senão
          Se (rz→esq == NULL)
             Se (ant == NULL)      //remover nó raiz da árvore
                raiz=rz→dir;
                apagar rz;
             Senão
                Se (ant→key < chave)
                   ant→dir=rz→dir;
                Senão
                   ant→esq=rz→dir;
                   apagar rz;
                Fse
         Senão  // Encontrar o nó filho esq que está mais à dir
```





Remover - Algoritmo (2)

```
  q=rz→esq;
  p=q→dir;
  Se (p == NULL)
     rz→key=q→inf;
     rz->esq=q→esq;
     apagar q;
  Senão
    Enqto (p→dir != NULL)
        q=p;
        p=p→dir;
    FEnqto
    rz→key=p→key;
    q→dir=p→esq;
    apagar p;
Senão
   "Elem nao existe na arvore"
Fim Remover
```



Classe Binary Search Tree (bst)

```
class bst
{
 protected:
   treeNode <TN> *root;
   int nodeCount;

   treeNode<TN>* copyRecursive (const treeNode<TN>* node);
   int getTreeHeight (treeNode<TN>* node) const;

   treeNode<TN>* deleteAndUpdate (treeNode<TN>* toDelete,
                treeNode<TN>* node, treeNode<TN>* &substitute);
   treeNode<TN>* popOneNode(treeNode<TN>* node);
   treeNode<TN>* popRecursive(treeNode<TN>* node, const TN &key,
                             bool &isDeleted);

   virtual treeNode<TN>* balanceNode(treeNode<TN>* node) { return
                                                          node; }
```

Classe Binary Search Tree (bst) (2)

```
    void clearRecursive(treeNode <TN> *node);
    treeNode<TN>* pushRecursive(treeNode<TN>* node, const TN &key,
                                            bool &isInserted);
public:
     bst();
     bst(const bst<TN> &bst);
     ~bst();
     int getTreeHeight() const;
     int size() const;
     bool push(const TN &key);
     bool pop(const TN &key);
     void clear();
     bstIterator<TN> begin();
     bstIterator<TN> end();
};
```

Classe Binary Search Tree (bst) (3)

```
treeNode<TN>* bst<TN>::pushRecursive(treeNode<TN>* node,
                                const TN &key, bool &isInserted)
{
   if (!node) {
     isInserted=true;
     node = new treeNode<TN>(key);
     return node; }
   if (equalToKeys(node->getKey(), key)) {
      isInserted=false;
      return node; }
   if (lessThanKeys(node->getKey(), key)) {
     treeNode<TN> *rn=pushRecursive(node->getRight(),key,isInserted);
     if (isInserted) {
       node->setRightSide(rn,getTreeHeight(rn));
       node=balanceNode(node); }
   else
```

Classe Binary Search Tree (bst) (3)

```
else {
  treeNode<TN> *rn = pushRecursive(node->getLeft(),key,isInserted);
  if (isInserted) {
    node->setLeftSide(rn,getTreeHeight(rn));
    node=balanceNode(node);  }
 }
 return node;
}

bool bst<TN>::push(const TN &key){
   bool isInserted;
   root=pushRecursive(root,key,isInserted);
   if (isInserted) nodeCount++;
   return isInserted;
}
```

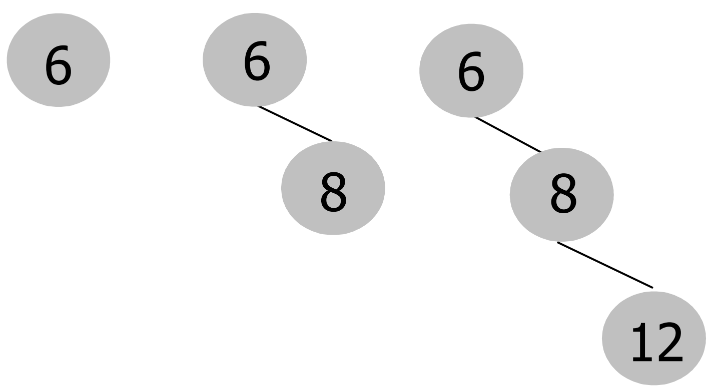
Aplicações

As árvores binárias de pesquisa são muito usadas em pesquisas:

- Tabelas de símbolos de compiladores, "assemblers"
- Aplicações de Bases de Dados

Pode ocasionalmente acontecer, devido à forma aleatória de inserção e remoção de elementos que conduza a tempos de resposta mais demorados - basta que a árvore se apresente desequilibrada

As árvores binárias de pesquisa apresentam uma desvantagem - a sua estrutura depende da ordem de inserção dos elementos: 6, 8, 12, 14, 17..



..... **a árvore degenera em lista ligada**

Esta limitação é ultrapassada com **Árvores Equilibradas (ou AVL)**

Árvores Equilibradas

ou Árvores AVL

Árvores Equilibradas

- A deficiência das árvores binárias de pesquisa é ultrapassada através de um controlo sobre a construção da estrutura da árvore

- Idealmente pretende-se que a árvore esteja balanceada, ou seja, para um qualquer nodo p a altura da subárvore esquerda seja **aproximadamente igual** à altura da subárvore direita

- Obviamente há um custo de **processamento extra para manter a árvore balanceada**, mas este é compensado quando os dados são **recuperados muitas vezes**

- A ideia de manter uma árvore binária balanceada dinamicamente, ou seja, enquanto os nodos são inseridos foi proposta em 1962 por dois soviéticos chamados **A**delson-**V**elskii e **L**andis $\rightarrow$ árvore **AVL**

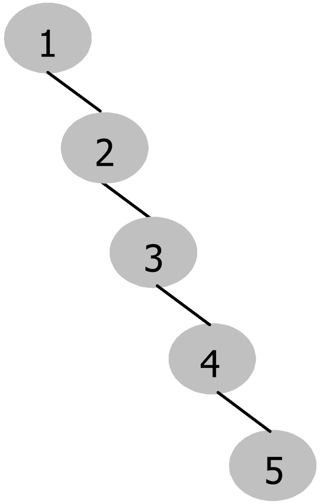
Definição – Árvores AVL

É uma árvore binária de pesquisa onde o módulo da diferença em altura entre as subárvores esquerda e direita é no máximo 1

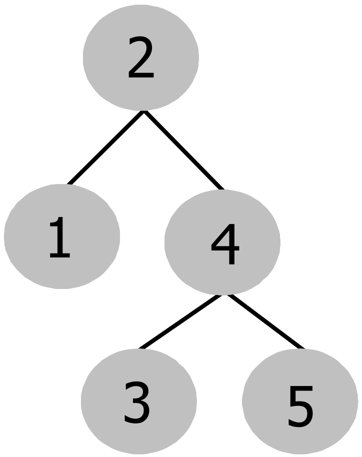
Assim, para cada nó define-se um **Factor de Equilíbrio (FE)**

FE(nodo p) = altura(subarv direita p) – altura(subarv esquerda p)

Árvore Binária de Pesquisa



Árvore AVL



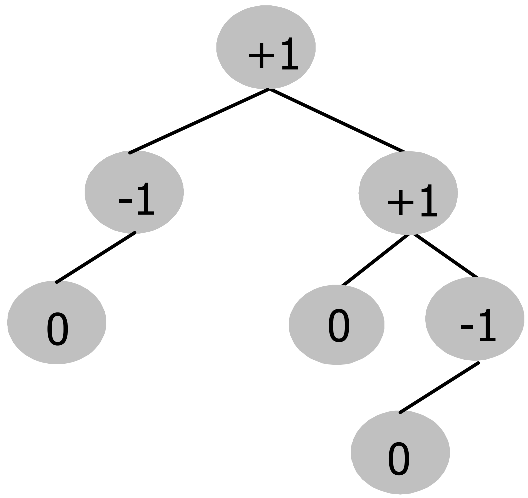
Isto garante que a árvore tenha profundidade logarítmica permitindo pesquisas em O(log n)

Factores de Equilíbrio (FE)

- **Factor de Equilíbrio negativo** de um nó significa que a altura da sua subárvore esquerda é maior (em pelo menos 1 nó) do que a altura da sua subárvore direita → **nó pesado à esquerda**

- **Factor de Equilíbrio positivo** de um nó significa que a altura da sua subárvore direita é maior (em pelo menos 1 nó) do que a altura da sua subárvore esquerda → **nó pesado à direita**

- **Factor de Equilíbrio nulo** de um nó significa que a altura da subárvore esquerda é igual à altura da subárvore direita
  → **nó equilibrado**



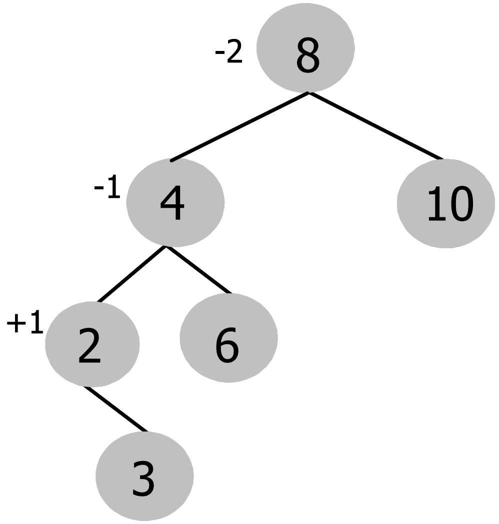
Balanceamento de Árvores AVL

É necessário sempre que a inserção ou remoção de um novo nó viole a **propriedade de balanceamento**

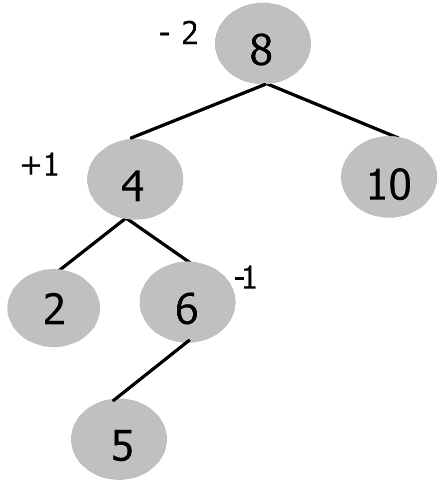
↪ árvore apresente nós cujos FE $\notin$ [-1,..,1]

O balanceamento da árvore é conseguido através de **Rotações**:

- **Simples** – quando o nó desequilibrado apresenta um FE com o mesmo sinal do FE do seu nó filho da subárvore desequilibrada
- **Duplas** – quando o nó desequilibrado apresenta um FE com sinal contrário ao FE do seu nó filho da subárvore desequilibrada



Rotação Simples

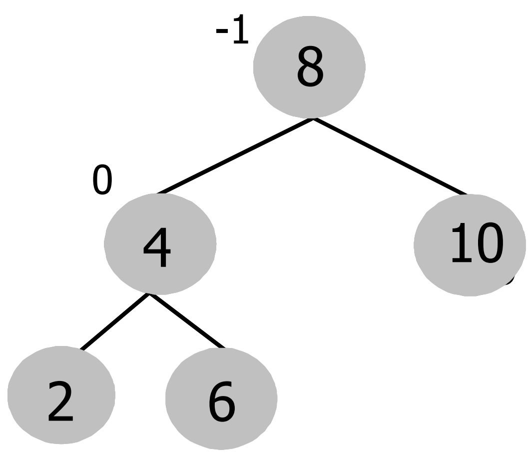


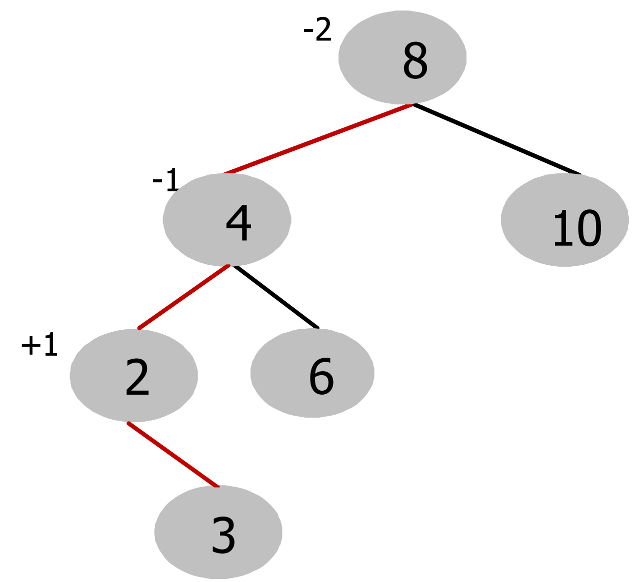
Rotação Dupla

Balanceamento de Árvores AVL

Existe uma propriedade muito importante nas árvores AVL:

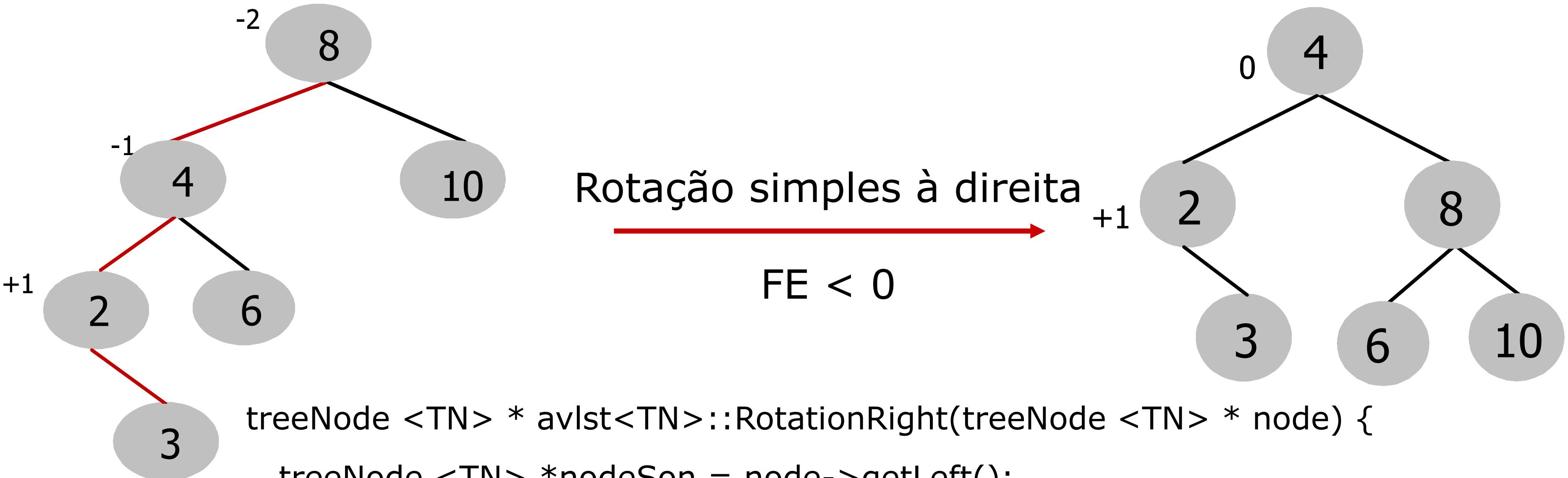
- após a inserção de um elemento existe a garantia de que apenas os nós que se encontram no caminho entre esse elemento e a raiz da árvore podem ficar desequilibrados
  - ↪ as operações de reequilíbrio só são necessárias sobre os nós que se encontram nesse caminho





Rotação Simples à Direita

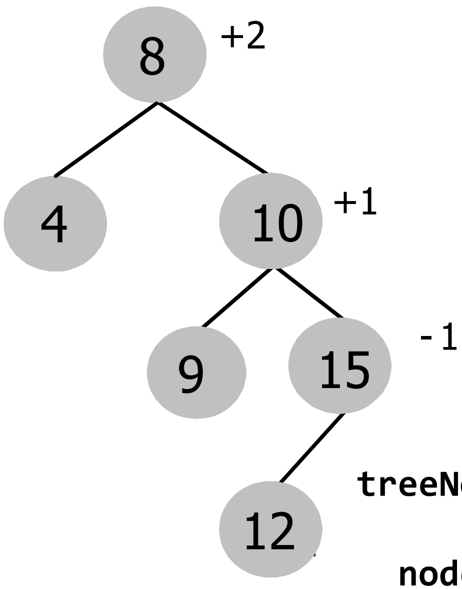
A **rotação** ocorre sempre no sentido contrário do desequilíbrio




```
treeNode <TN> * avlst<TN>::RotationRight(treeNode <TN> * node) {
   treeNode <TN> *nodeSon = node->getLeft();

   node->setLeft(nodeSon->getRight());
   nodeSon->setRight(node);

   node->setLeftHeight(getTreeHeight(node->getLeft()));
   node->setRightHeight(getTreeHeight(node->getRight()));

   nodeSon->setLeftHeight(getTreeHeight(nodeSon->getLeft()));
   nodeSon->setRightHeight(getTreeHeight(nodeSon->getRight()));

   node = nodeSon ;

   return node ;  }
```

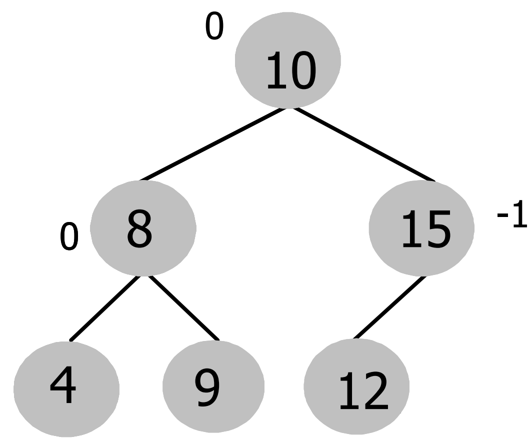
Rotação Simples à Esquerda

A **rotação** ocorre sempre no sentido contrário do desequilíbrio



Rotação simples à esquerda

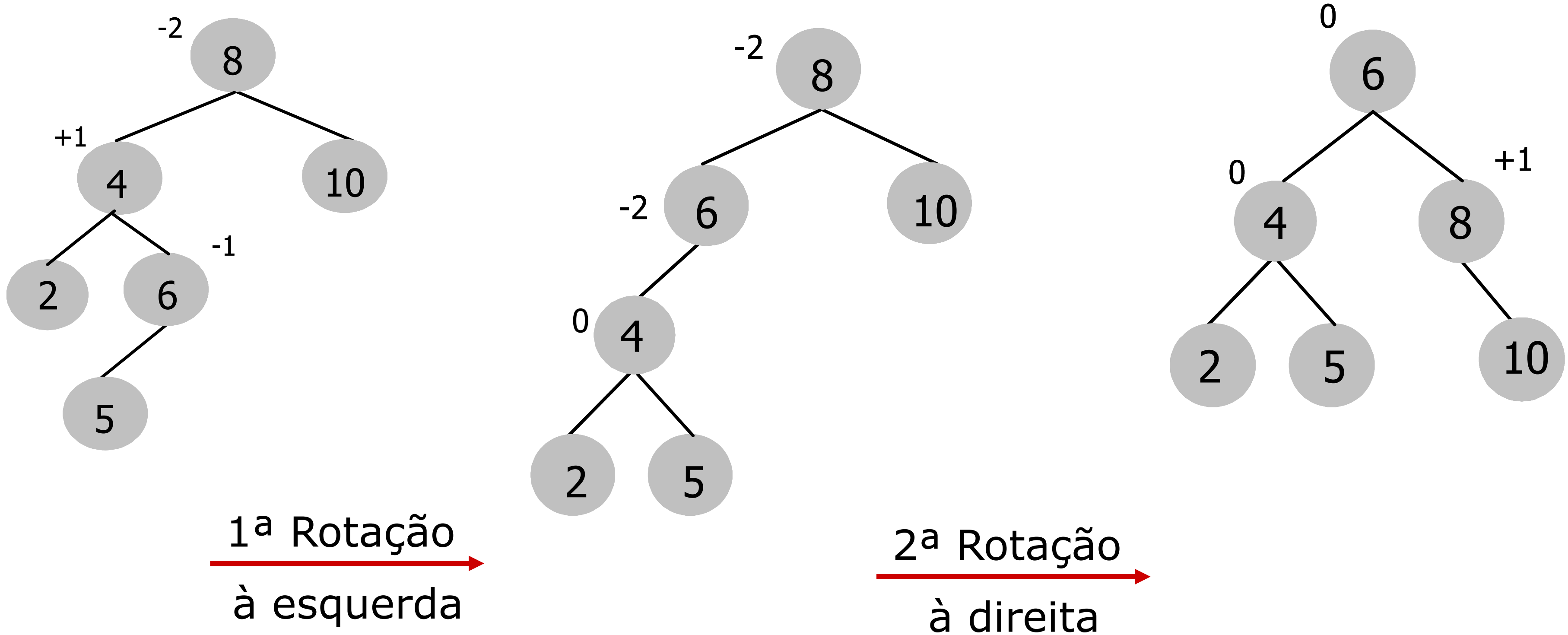
FE > 0



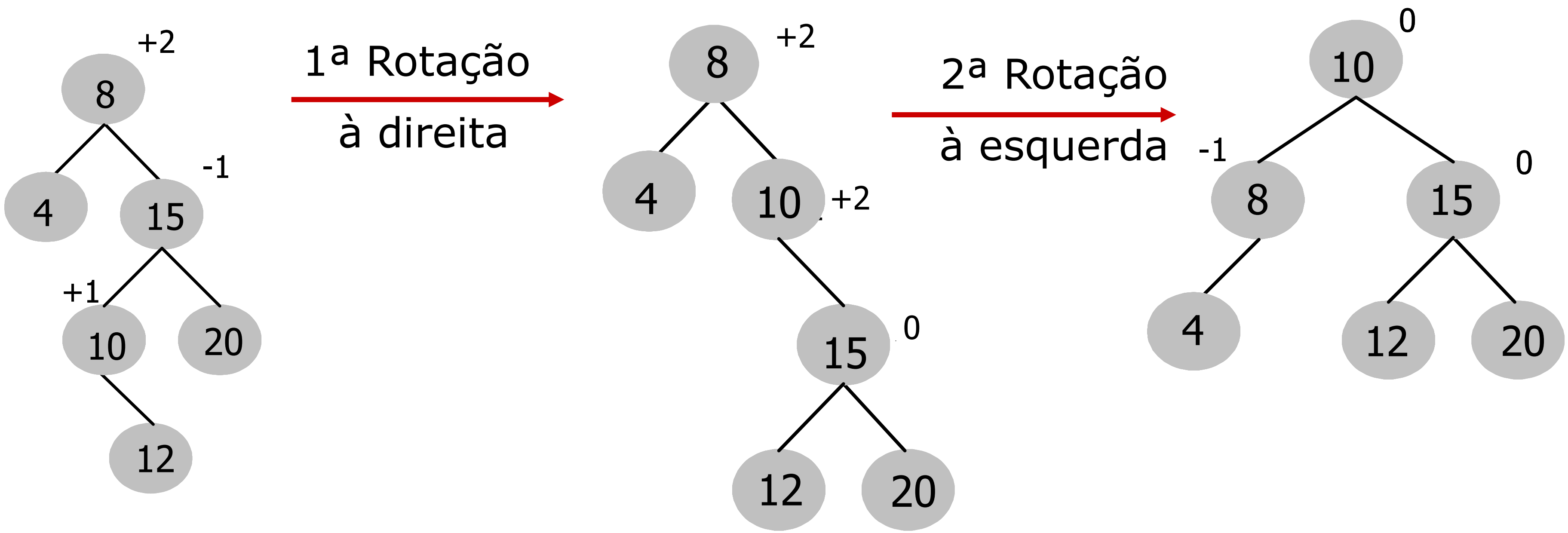

```
treeNode <TN>* avlst<TN>::RotationLeft(treeNode <TN>* node) {

  node->setRight(nodeSon->getLeft());
  nodeSon->setLeft(node);

  node->setLeftHeight(getTreeHeight(node->getLeft()));
  node->setRightHeight(getTreeHeight(node->getRight()));

  nodeSon->setLeftHeight(getTreeHeight(nodeSon->getLeft()));
  nodeSon->setRightHeight(getTreeHeight(nodeSon->getRight()));

  node = nodeSon ;

  return node ;   }
```

Rotação Dupla (Esq-Dir)

- A **primeira rotação** ocorre na raiz da subárvore do nó desequilibrado e sempre no sentido do desequilíbrio do nó
- A **segunda rotação** ocorre sempre no sentido contrário à primeira rotação



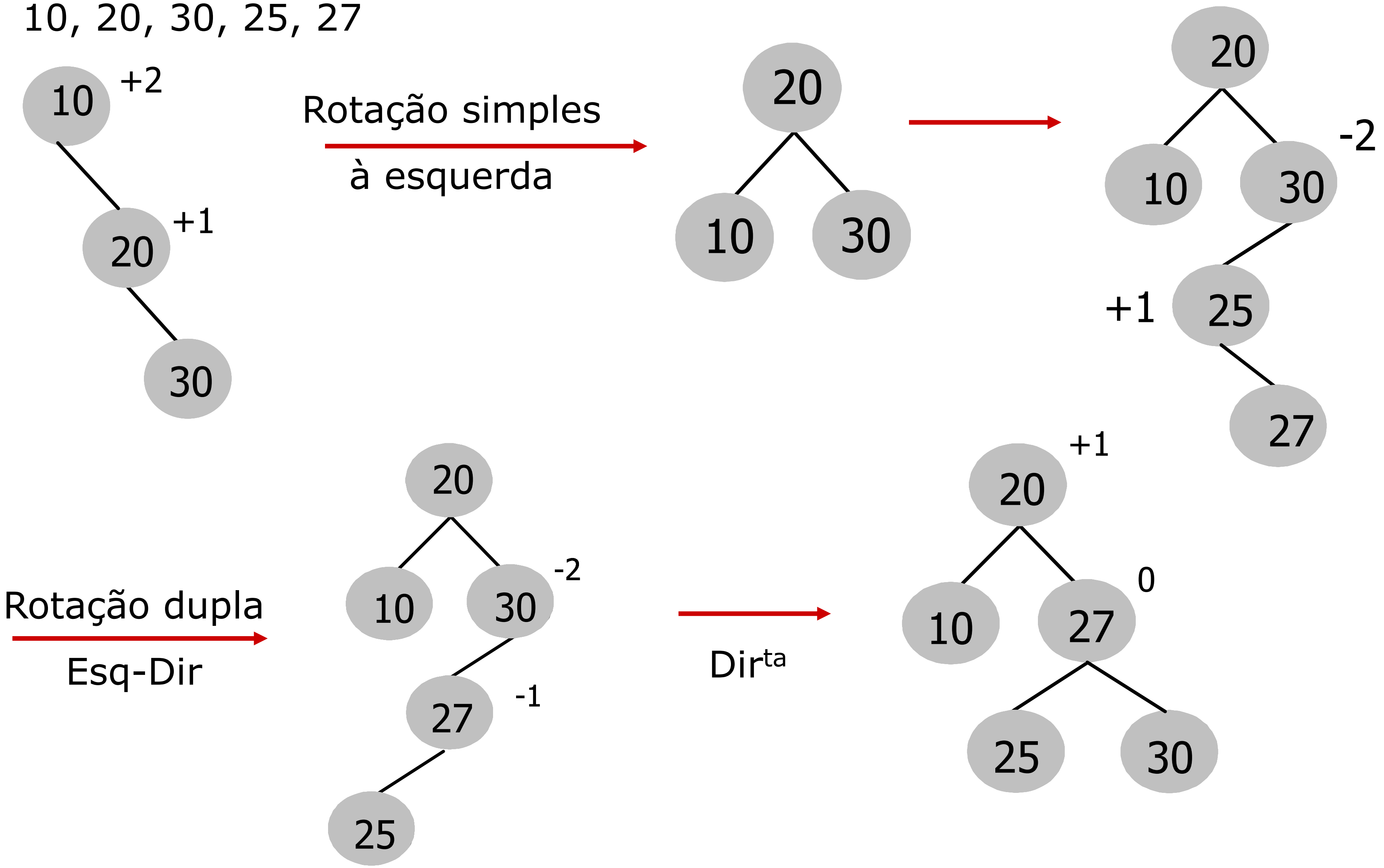
Rotação Dupla (Dir-Esq)




```
treeNode <TN> * avlst<TN>::twoRotations(treeNode <TN> * node) {

  if (node->getBalanceFactor() < 0) {
    node->setLeft(RotationLeft(node->getLeft()));
    node = RotationRight(node) ;   }
  else {
    node->setRight(RotationRight(node->getRight()));
    node = RotationLeft(node) ;   }

  return node ;   }
```

Inserção em Árvore AVL

10, 20, 30, 25, 27



Inserção em Árvore AVL

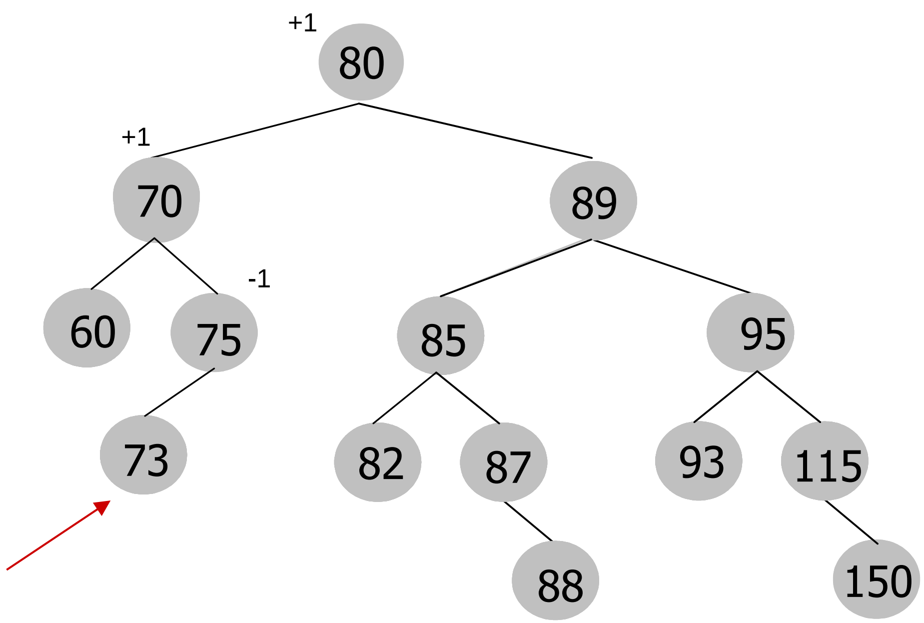
Inserir (chave, rz)
<u>se</u> (rz == null)
Insere o novo nó
<u>senão</u>
<u>se</u> (rz→info == chave)
Elemento repetido
<u>senão</u>
<u>se</u> (rz→info > chave)
procura recursivamente à esq[da] o ponto de inserção
actualiza o FE do nó
<u>se</u> FE(nó) $\notin$ [-1,..,1]
faz a rotação necessária
<u>senão</u>
procura recursivamente á dir[ta] o ponto de inserção
actualiza o FE do nó
<u>se</u> FE(nó) $\notin$ [-1, .., 1]
faz a rotação necessária
<u>Fse</u>
<u>Fse</u>
<u>Fse</u>
devolve rz
Fim Inserir

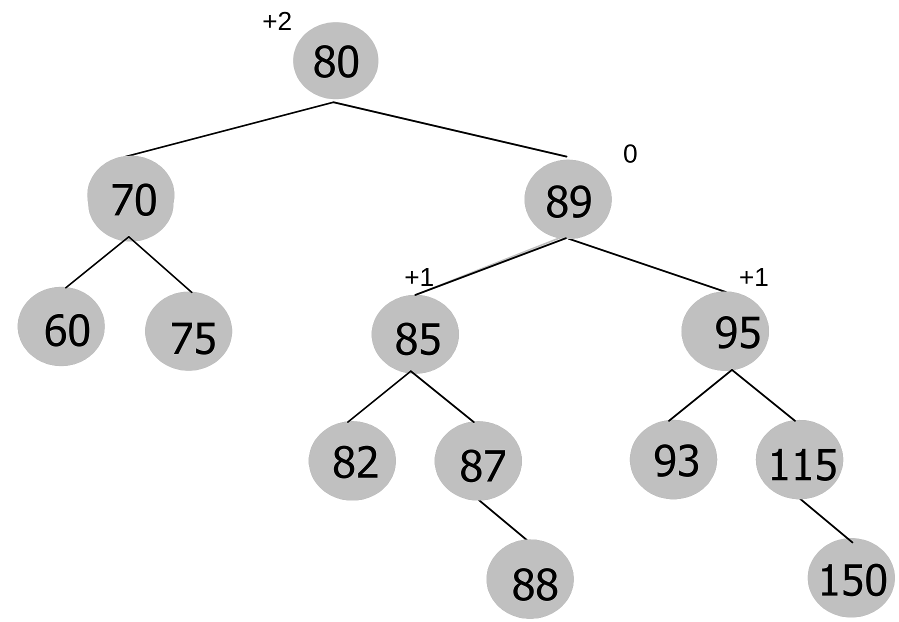
Nota: O re-equilíbrio só se faz "ao voltar para trás" na recursividade, após a inserção

Remoção em Árvore AVL

O princípio de reequilíbrio é o mesmo que foi usado nas inserções:

- após a remoção de um nó (usando o mesmo método das árvores binárias de pesquisa
- percorre-se o caminho de volta para a raiz reequilibrando os nós que necessitarem
- actualiza-se os respectivos factores de equilíbrio





Remoção 73:

Rotação simples, pivot = 80

Remoção em Árvore AVL

```
template <class T>
void ArvAVL<T>::eliminar(const T& chave, Nodo<T>* rz, Nodo<T>* ant)
{
   if (chave < rz->inf)
   {                    procura à esq o elemento a remover
      actualiza a altura dos nós e faz a rotação necessária
   }
   else if (chave > rz->inf)
   {                    procura à dir o elemento a remover
      actualiza a altura dos nós e faz a rotação necessária
   }
   else
      if (rz->esq == NULL && rz->dir == NULL) //remove nó sem descendentes
      {   }
      else if (rz->esq == NULL)               //remove nó com subarv dir
      {   }
      }
      else if (rz->dir == NULL)               //remove nó com subarv esq
      {     }
      else                                    //remove no com as duas subarvores
      {                         // Encontra maior nó da subarv. esquerda

       atualiza a altura do nó e faz a rotação necessária
      }
}
```

Classe AVL Search Tree (avlst)

```
template <class TN>
class avlst : public bst <TN>
{
  private:
          treeNode <TN> * RotationLeft(treeNode <TN> * node);
          treeNode <TN> * RotationRight(treeNode <TN> * node);
          treeNode <TN> * twoRotations(treeNode <TN> * node);
          treeNode <TN> * balanceNode(treeNode <TN> *node);
  public:
          avlst();
          avlst(const avlst<TN> &a);
         ~avlst() ;
};
```

Classe AVL Search Tree (avlst)

```
treeNode <TN> * avlst<TN>::balanceNode(treeNode <TN> * node) {
    if (node->getBalanceFactor() < -1)
       if (node->getLeft()->getBalanceFactor() < 0)
          node = RotationRight(node);
       else
          node = twoRotations(node) ;

    if (node->getBalanceFactor() > 1)
      if (node->getRight()->getBalanceFactor() > 0)
         node = RotationLeft(node) ;
      else
         node = twoRotations(node) ;

   return node ;
}
```

Árvores AVL vs. Binárias de Pesquisa

Em média 50% das inserções e eliminações obrigam a rotações

Estas conduzem a uma perda de eficiência nos algoritmos de inserção e eliminação

Assim, a utilização desta estrutura depende das aplicações:

- aplicações onde a **pesquisa seja a operação dominante** devem ser usadas **árvores AVL**, pois estas garantem uma ordem de complexidade **log n**
- aplicações onde as **inserções ou eliminações** são as operações mais frequentes devem ser usadas as **árvores binárias de pesquisa**