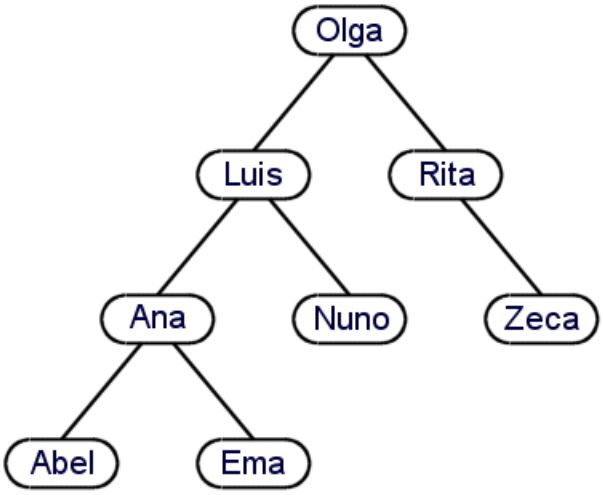
❒ Árvores Binárias de Pesquisa



- Numa **Árvore Binária de Pesquisa**, a informação está registada nos vértices **em-ordem**.
- Uma travessia em-ordem equivale a uma listagem ordenada. Para cada vértice, todos os elementos contidos na sub-árvore esquerda são inferiores e todos os elementos contidos na sub-árvore direita são superiores.
- Portanto, uma **Pesquisa** numa ABP é semelhante à Pesquisa Binária e, no caso de uma árvore (quase) perfeita, tem também complexidade $O(\log_2 n)$.

```
procedure Procurar (elemento: tipo;  t : ABP,  var achou : boolean);

    begin  if vazia(t)
           then  achou:= false;
           else  if  elemento = consultar(t)
                 then achou:= true
                 else  if elemento < consultar(t)
                       then  Procurar(estenome, esquerda(t), achou)
                       else  Procurar(estenome, direita(t), achou)
    end; { Procurar }
```

Consideremos:

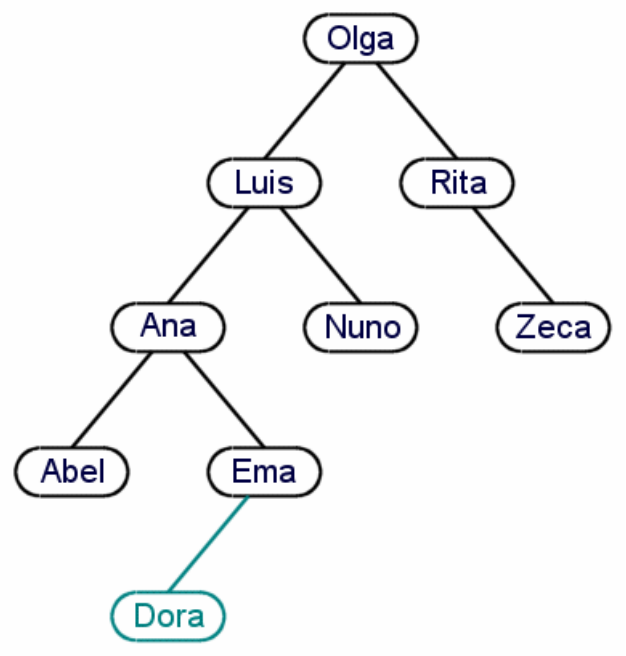
```
type  ABP  = ^no;
      no   = record  elem : tipo;
                     esq, dir : ABP
             end;
```

Pesquisa numa ABP

- A **pesquisa** de um elemento deverá fornecer o ponteiro para o nó que o contém (ou informação a ele associada).
- No caso do elemento não existir, o ponteiro ***nil*** encontrado indica a posição onde deveria estar na ABP.

Inserção de um novo elemento numa ABP

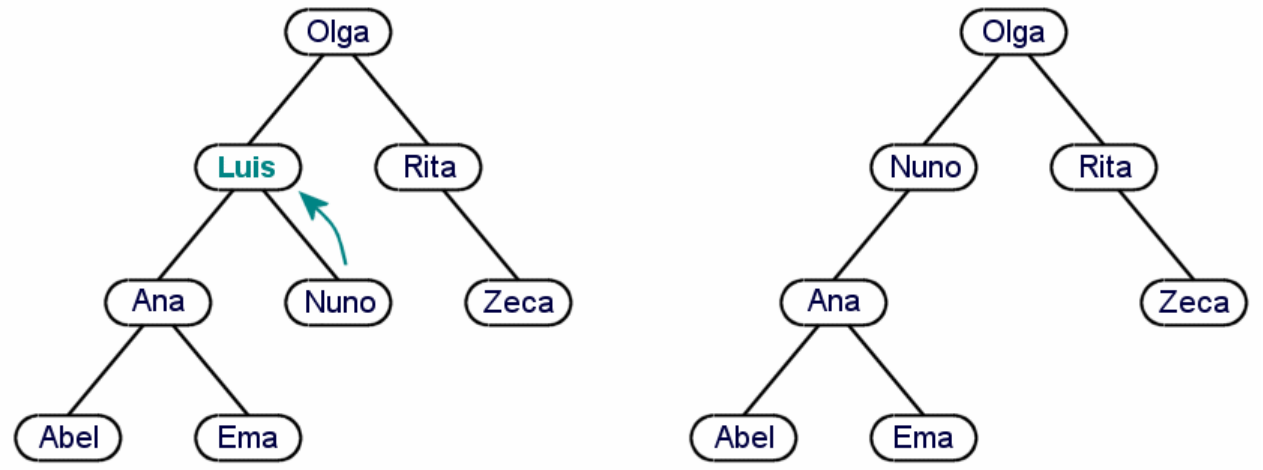
- Se o novo elemento não estiver na ABP, a sua **inserção** consiste na pesquisa do ponteiro ***nil*** que indica a sua posição, seguida da substituição desse ***nil*** por uma nova **folha**.



```
procedure Inserir ( novo: tipo; var t : ABP );

    begin if t = nil
          then begin new(t);
                     with t^ do
                          begin esq := nil;
                                dir := nil;
                                elem := novo
                          end
               end

          else with t^ do
                    if novo < elem
                    then Inserir(novo, esq)
                    else Inserir(novo, dir)

    end; { Inserir }
```

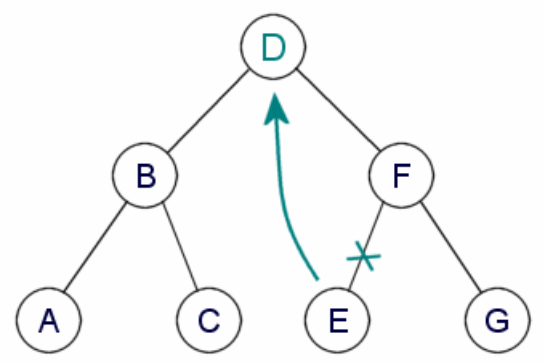
➡ Remoção de um elemento de uma ABP

- Se o elemento a remover for uma **folha**, a operação é simples.
- Quando o **vértice seguinte** é uma folha, basta copiar o seu elemento e eliminá-la.

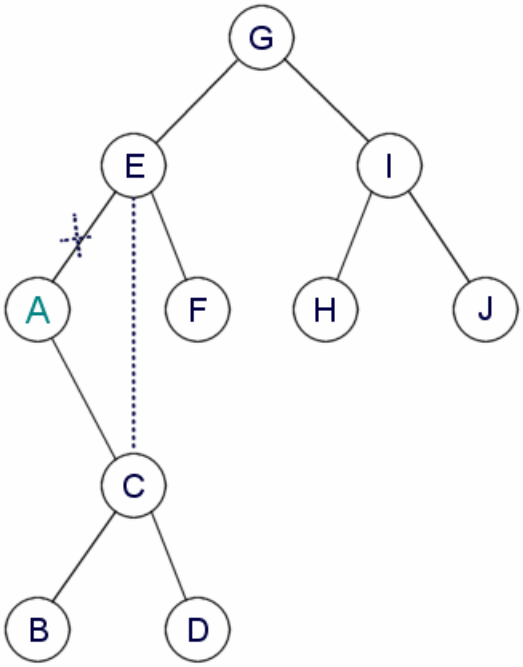


- Se o **vértice anterior** for uma folha, a solução é análoga.
- No caso geral,

  remover este vértice:
  procurar o seu sucessor em-ordem;
  copiar o elemento do sucessor para este;
  eliminar o sucessor.



- O **sucessor em-ordem** de um vértice é o ***descendente mais esquerdo*** do seu ***filho direito***.
- Comecemos por construir um módulo auxiliar, para **eliminar** o ***descendente mais esquerdo*** de uma (sub-)ABP, **guardando** o seu elemento.
  Como a árvore está em-ordem, este é **primeiro** vértice.



{ Eliminar o primeiro vértice de uma ABP, não vazia, copiando o seu elemento. }

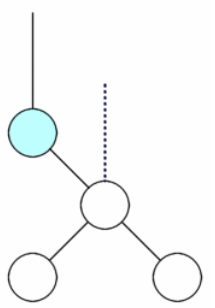
```
procedure EliminarPrimeiro ( var t : ABP; var primeiro : tipo );
    var aux : ABP;

    begin if t <> nil
          then begin if t^.esq = nil
                     then begin primeiro := t^.elem;
                                aux := t;
                                t := t^.dir;
                                dispose(aux)
                          end

                     else EliminarPrimeiro(t^.esq, primeiro)
               end

    end; { EliminarPrimeiro }
```

- O procedimento para **remover** um dado **vértice**, referido pelo seu ponteiro, deverá considerar os 4 casos possíveis:

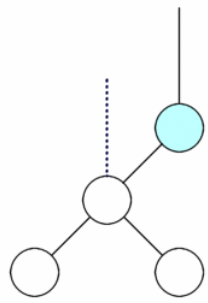
  1. é uma folha
  2. só tem filho direito
  3. só tem filho esquerdo
  4. tem 2 filhos – caso geral : utilizar o módulo anterior

{ Remover um vértice de uma ABP, referido pelo ponteiro ***remt*** }

```
procedure RemoverNo ( var remt : ABP );
var   t : ABP;
      este : tipo;

begin  if remt <> nil
       then  begin  if (remt^.esq = nil) and (remt^.dir = nil)
                    then  begin  { folha }
                                 dispose(remt);
                                 remt:= nil
                                 end
```



```
                    else  if  remt^.esq = nil
                          then  begin  { só filho direito }
                                       t := remt;
                                       remt := remt^.dir;
                                       dispose(t)
                                end
```

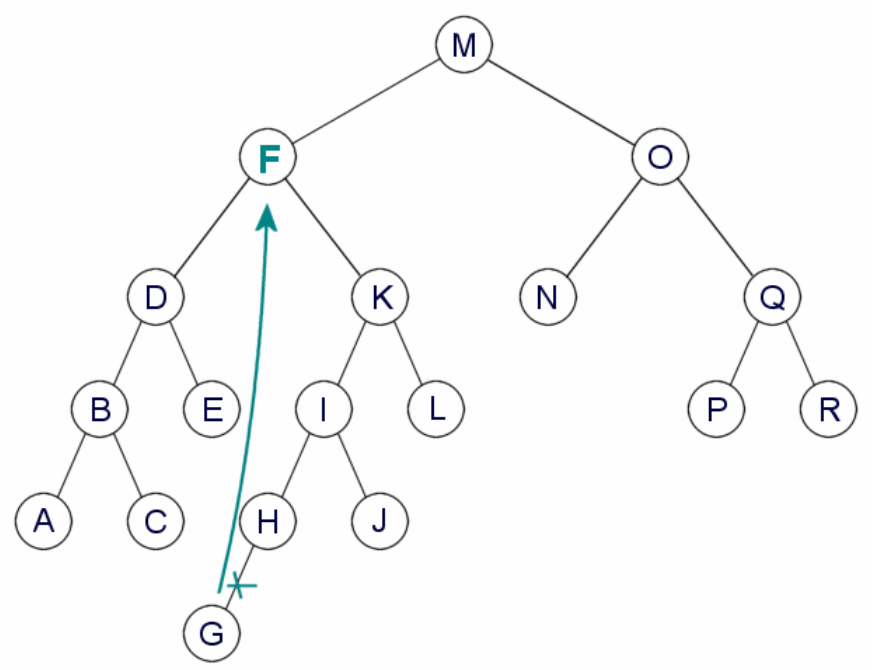


```
                          else  if  remt^.dir = nil
                                then  begin  { só filho esquerdo }
                                             t := remt;
                                             remt := remt^.esq;
                                             dispose(t)
                                      end

                                else  begin  { caso geral }
                                             { elinimar sucessor }
                                             EliminarPrimeiro(
                                                   remt^.dir, este);
                                             { copiar }
                                             remt^.elem := este
                                      end
                 end

end; { RemoverNo }
```

- E finalmente, a operação completa deverá incluir a pesquisa do elemento a remover.

{ Remover um elemento de uma ABP }

```
procedure Remover ( elemento : tipo;  var  t : ABP;
                                      var  sucesso : boolean );

  begin  if t = nil
         then sucesso := false

         else    if elemento = t^.elem
                 then  begin  RemoverNo(t);
                              sucesso := true
                       end

                 else if  elemento < t^.elem
                      then  Remover(elemento, t^.esq, sucesso)
                      else  Remover(elemento, t^.dir, sucesso)
  end; { Remover }
```

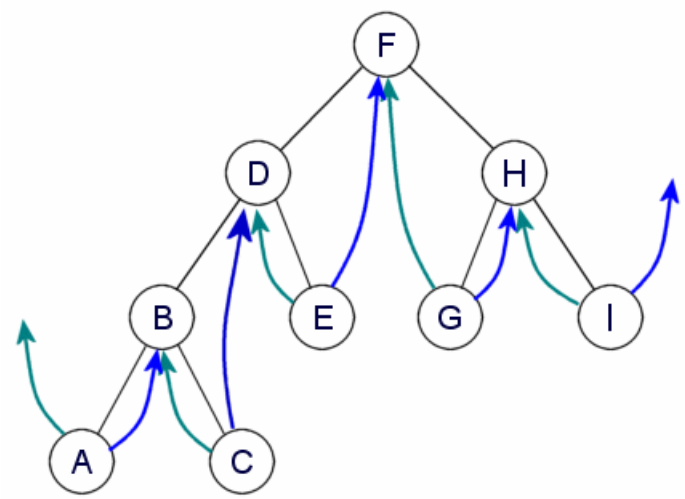


- Notar que as operações de **gestão** de uma ABP não preservam o seu **equilíbrio** e a desejada complexidade de $O(\log_2 n)$ pode, no pior dos casos, tornar-se de $O(n)$.

❒ Árvores Binárias Enlaçadas *(threaded)*

Observações:

- Toda a árvore binária com n > 0 vértices tem n–1 arestas.
- Para representar uma árvore binária com n vértices, são necessários 2 n ponteiros.
- Portanto, 2 n – (n–1) = n + 1 ponteiros são nulos.
- Como aproveitar o espaço ocupado pelos ponteiros nulos?
- Por exemplo, uma **árvore binária** pode ser **enlaçada** de forma a permitir uma **travessia em-ordem** sem recorrência e sem pilha, em ambos os sentidos.



- Cada ponteiro **direito** nulo passará a ser um **laço** para o vértice **sucessor** em-ordem, e cada ponteiro **esquerdo** nulo passará a ser um **laço** para o vértice **antecessor** em-ordem.

Mas:

- Como funciona a travessia?
- Como distinguir os ponteiros normais dos laços?
- Onde ligar o primeiro e o último?

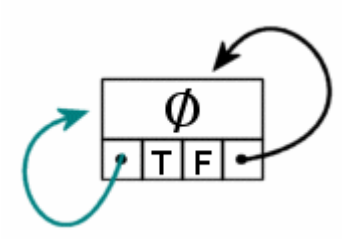
Representação de uma ABE

```
type  ABE  = ^no;
      no   = record  elem : tipo;
                     esq, dir : ABE;
                     laçoesq, laçodir : boolean
             end;
```

- laçoesq = true significa que esq é um **laço** para o predecessor em-ordem.
- laçoesq = false significa que esq é um **ponteiro** normal para a sub-árvore esquerda.

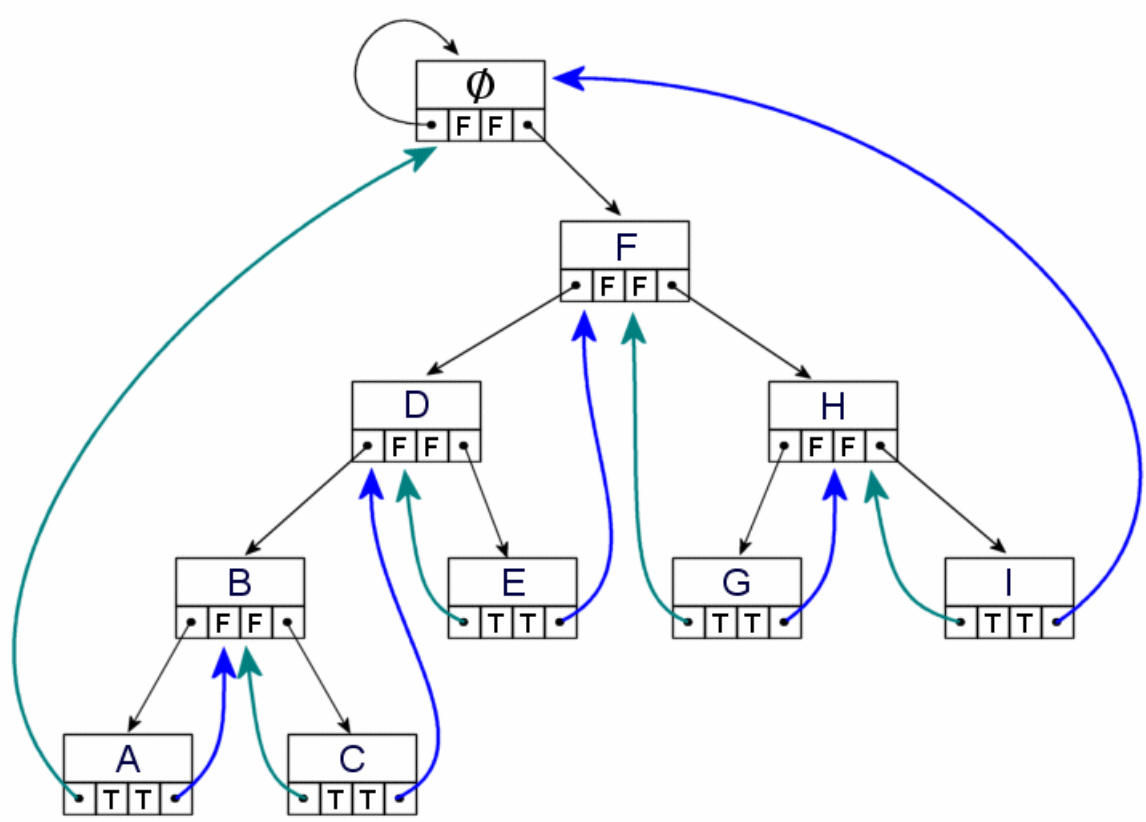
- À semelhança das listas ligadas circulares, convém aqui utilizar um **vértice base**.

- Uma árvore binária enlaçada **vazia** será então:




```
procedure  criar ( var base : ABE );

           begin  new(base);
                  with  base^  do
                        begin  laçoesq := true;
                               esq := base;
                               laçodir := false;
                               dir := base
                        end
           end; { criar }
```

- Para o exemplo anterior:



➡ Determinação do sucessor em-ordem numa ABE

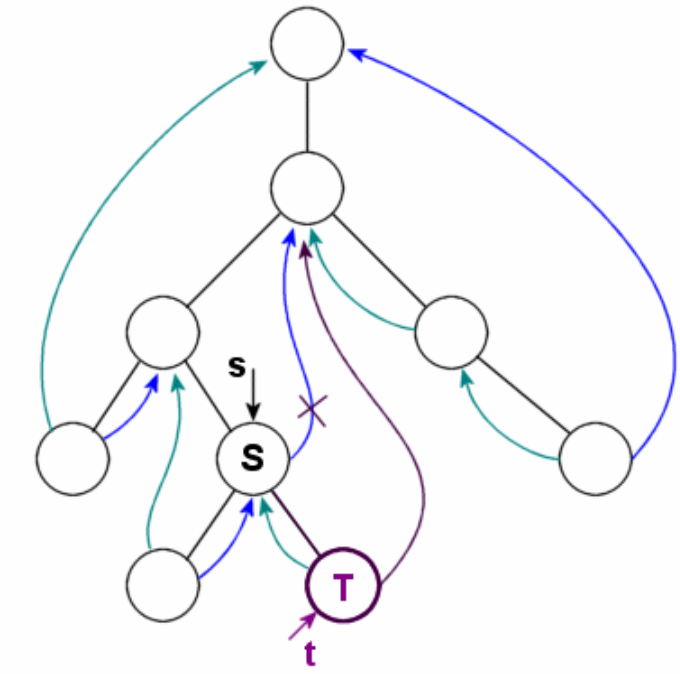
- Se laçodir = true então o sucessor é referido pelo laço dir.
- Se laçodir = false significa que dir é uma aresta da própria árvore. O sucessor encontra-se a partir de dir, seguindo sempre os ponteiros esq até que laçoesq = true.

```
function  sucessor_em_ordem ( t : ABE ) : ABE;
          var s : ABE;

          begin  s := t^.dir;
                 { se for laço, é este }
                 if  not  t^.laçodir
                 then  { descer para a esquerda deste }
                       while  not  s^.laçoesq do
                              s := s^.esq;

                 sucessor_em_ordem := s
          end; { sucessor_em_ordem }
```

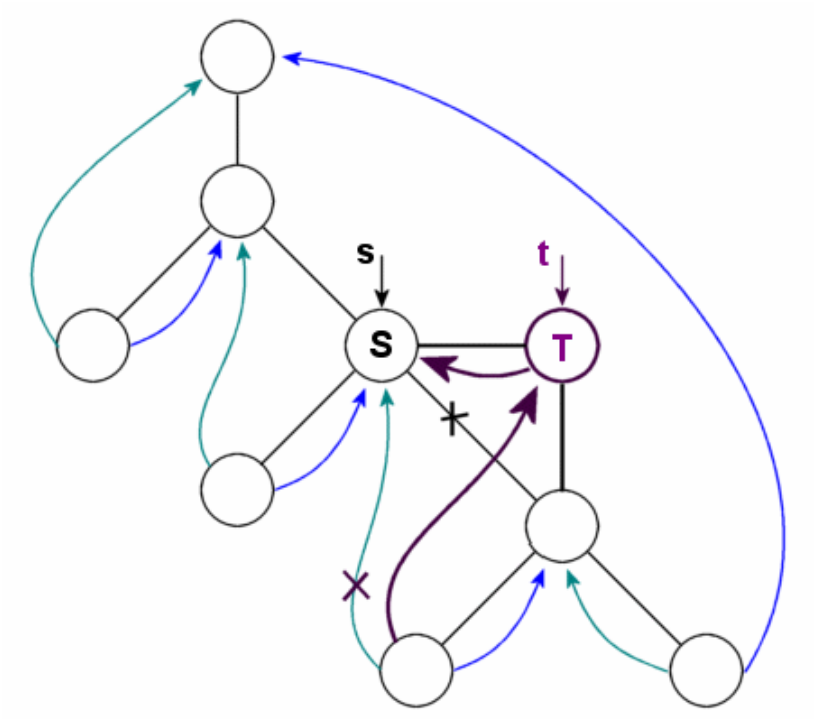
Exercício: Travessia em-ordem numa ABE.

➡ Inserção de um novo vértice numa ABE

- Inserir um novo vértice, referido por um ponteiro t, como **filho direito** do vértice referido por s.
- **1º Caso**: O vértice referido por s não tem filho direito.



- **2º Caso**: O vértice referido por s tem dois filhos.




```
procedure Inserir_direita ( s, t : ABE );
var w : ABE:

   begin  { estabelecer ligação à direita de t }
          t^.dir := s^.dir;
          t^.laçodir := s^.laçodir;

          { a esquerda de t é sempre um laço }
          t^.esq := s;
          t^.laçoesq := true;

          { t é o filho direito de s }
          s^.dir := t;
          s^.laçodir := false;
```

```
            { caso geral : se s já tinha 2 filhos }

            if not t^.laçodir
            then  begin  { procurar sucessor em-ordem do s original }
                         w := sucessor_em_ordem(t);

                         { e re-enlaçar }
                         w^.esq := t
                  end

end; { Inserir_direita }
```

Exercícios:

- Inserir à esquerda.
- Remover um vértice.
- Construir Árvore Binária Enlaçada em-ordem.
- Enlaçar em pre-ordem.

Observações:

- Dificuldade na abstracção das operações de gestão de uma ABE.
- As operações de remoção e inserção são limitadas a casos particulares.
- Uma programação em termos apenas de um conjunto de operações básicas não é satisfatória.

❒ Árvores Dinâmicas de Pesquisa

➡ (Quase) Equilíbrio

- Como vimos, a complexidade das operações de gestão de uma ABP pode variar entre $O(\log_2 n)$, para uma árvore perfeita, e $O(n)$ no pior dos casos.

Definição: **Equilíbrio em altura**

- A árvore binária **vazia** é **equilibrada em altura**.
- Uma árvore binária não vazia T, com T_E e T_D sub-árvores esquerda e direita, é **equilibrada em altura** se e só se:

  a) T_E e T_D são equilibradas em altura.
  b) $| h(T_E) - h(T_D) | \leq 1$

Definição: O **Factor de Equilíbrio** FE(T) de um **vértice** (sub-árvore) T é dado por $h(T_D) - h(T_E)$.

Portanto, numa árvore binária equilibrada em altura $FE(T) \in \{ -1, 0, 1 \}$ para qualquer vértice T.

- No **Melhor dos Casos**, FE(T) = 0 para todo o T, a árvore binária é perfeita e portanto $h = \lfloor \log_2 n \rfloor + 1$, estando garantida a complexidade logarítmica.

- No **Pior dos Casos** teremos | FE(T) = 1 | em todo vértice T. Se a altura em T for h, as sua sub-árvores terão alturas h – 1 e h – 2.

- Seja N_h o **número mínimo de vértices** de uma árvore binária equilibrada de altura h.
  Nesse caso,

$$N_h = N_{h-1} + N_{h-2} + 1$$

com $N_0 = 0$, $N_1 = 1$ e $N_2 = 2$.

Verificamos que N_h segue a **Sucessão de Fibonacci** com,

$$N_h = F_{h+2} - 1, \quad \forall\, h \geq 0$$

Sabendo que,

$$F_h \approx \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^h = \frac{\phi^h}{\sqrt{5}} \quad \text{com} \quad \phi = \frac{1+\sqrt{5}}{2} = 1.618...$$

então

$$N_h \approx \frac{\phi^{h+2}}{\sqrt{5}} - 1$$

- Assim, no **pior dos casos** de uma árvore binária equilibrada em altura com n vértices,

$$n \approx \frac{\phi^{h+2}}{\sqrt{5}} - 1$$

$$h \approx \log_\phi(\sqrt{5}\,(n+1)) - 2$$

...

$$h < log_2 n$$

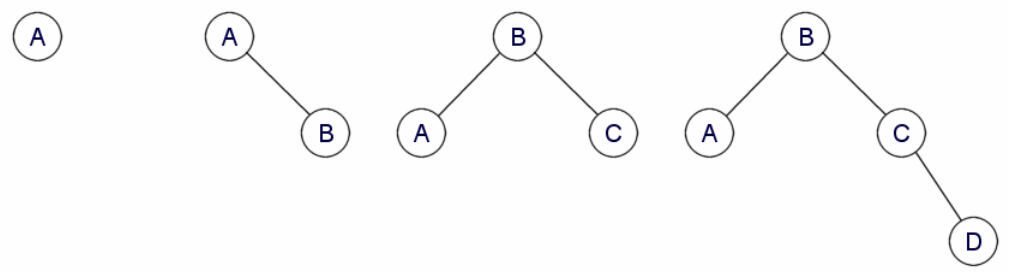
e a complexidade das operações de gestão é ainda $O(\log_2 n)$.

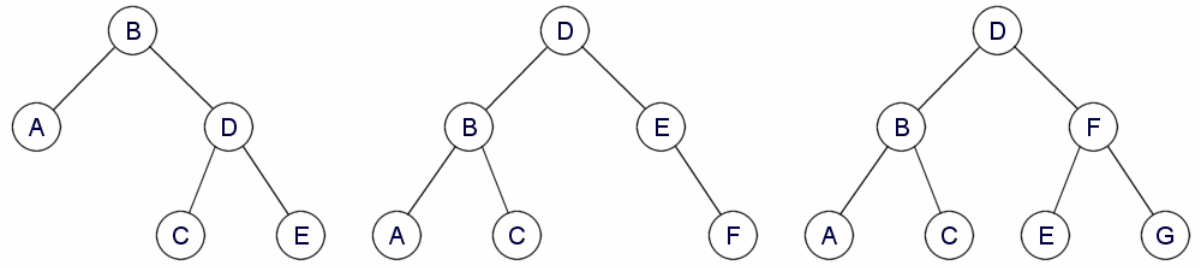
Árvores AVL

{ G. Adelson-Velskii e E. M. Landis, 1962 }

A ideia:

- Se pretendermos construir uma árvore binária de pesquisa, inserindo a sequência de elementos: A B C D E F G, o resultado seria uma árvore degenerada.
- Contudo, se a inserção de um novo vértice provocar um **desequilíbrio**, podemos efectuar uma **rotação** de forma a repor o equilíbrio.





- Consideremos,

```
type  AVL  = ^no;
      no   =  record  elem : tipo;
                      esq, dir : AVL;
                      contador : integer;
                      factor : -1 .. 1
              end;
```

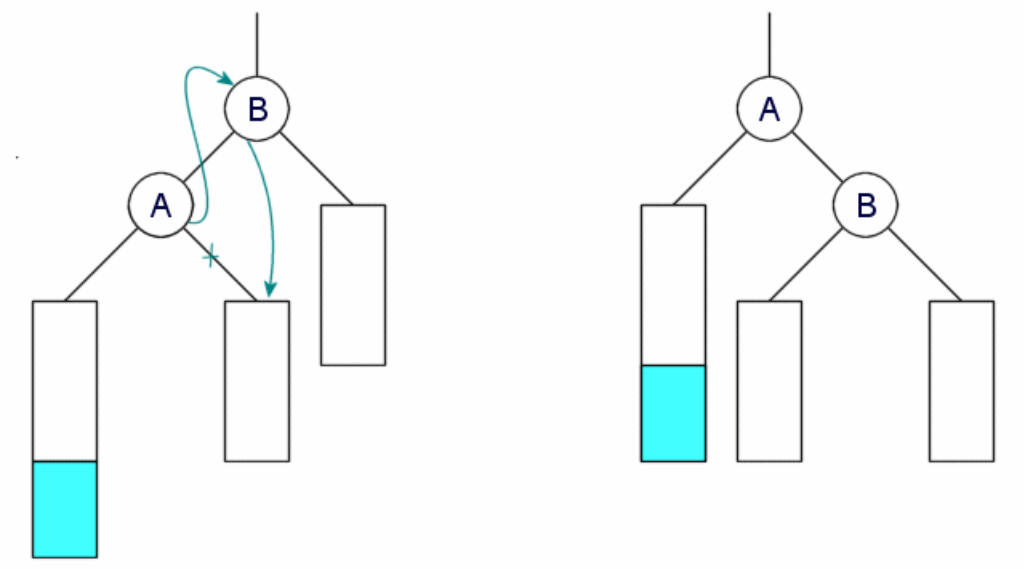
➡ Inserção de um novo elemento numa árvore AVL

Após procurada a localização do novo vértice, se necessário, será efectuada uma de entre quatro rotações:

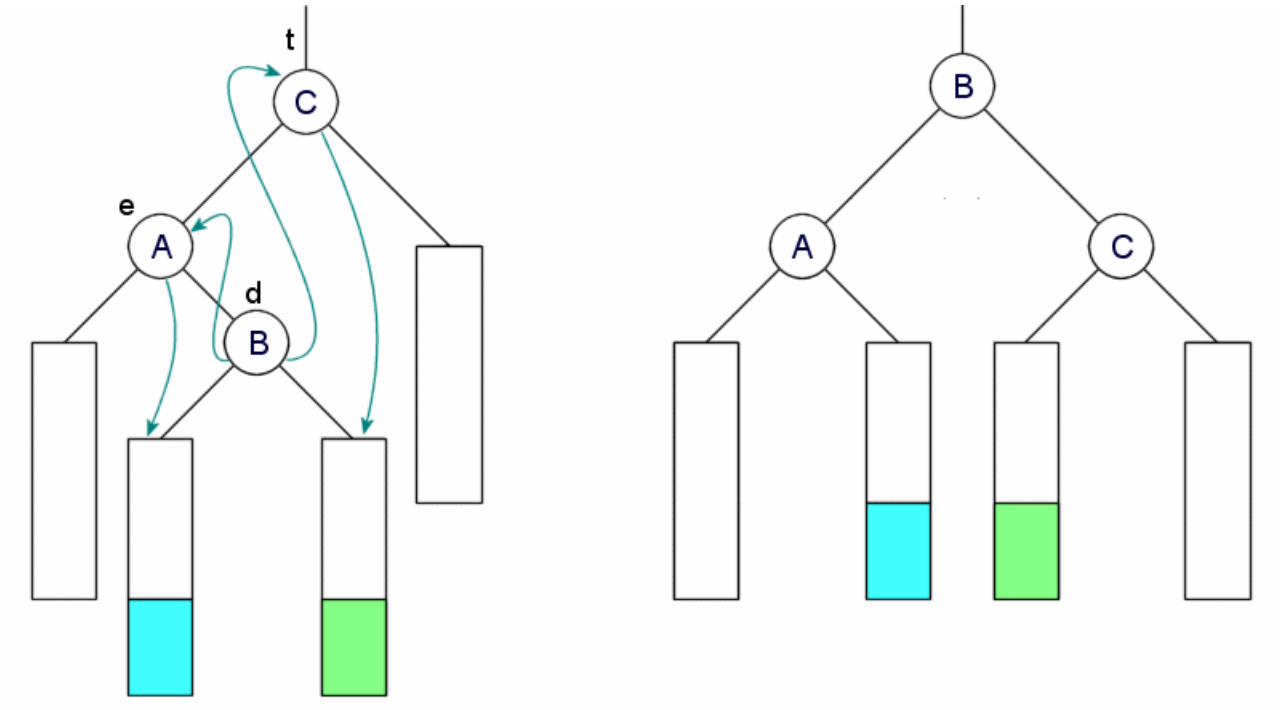
- rotação simples EE
- rotação dupla ED
- rotação simples DD
- rotação dupla DE

```
procedure Inserir ( novo: tipo; var t : AVL; var cresceu : boolean );
var e, d : AVL;

begin  if t = nil
       then  begin  new(t);
                    cresceu := true; { h : 0 → 1 }
                    with t^ do
                         begin  elem := novo;
                                contador := 1;
                                esq := nil; dir := nil;
                                factor := 0
                         end
             end

       else  { procurar lugar em-ordem }
             if  novo < t^.elem
             then  begin  inserir(novo, t^.esq, cresceu);

                          if cresceu { a sub-árvore esquerda }
                          then  case t^.factor of

                                  1 :  begin  { equilibrou }
                                              t^.factor := 0;
                                              cresceu := false
                                       end;
                                  0 :  t^.factor := -1;
                                       { ainda fica equilibrada }

                                 -1 :  { é necessário reequilibrar  ... }
```

```
-1 :  begin  { reequilíbrio por inserção à esquerda }
             e := t^.esq;

             if e^.factor = -1

             then  begin  { rotação simples EE }

                          t^.esq := e^.dir;
                          e^.dir := t;
                          t^.factor := 0;
                          t := e
                   end
```

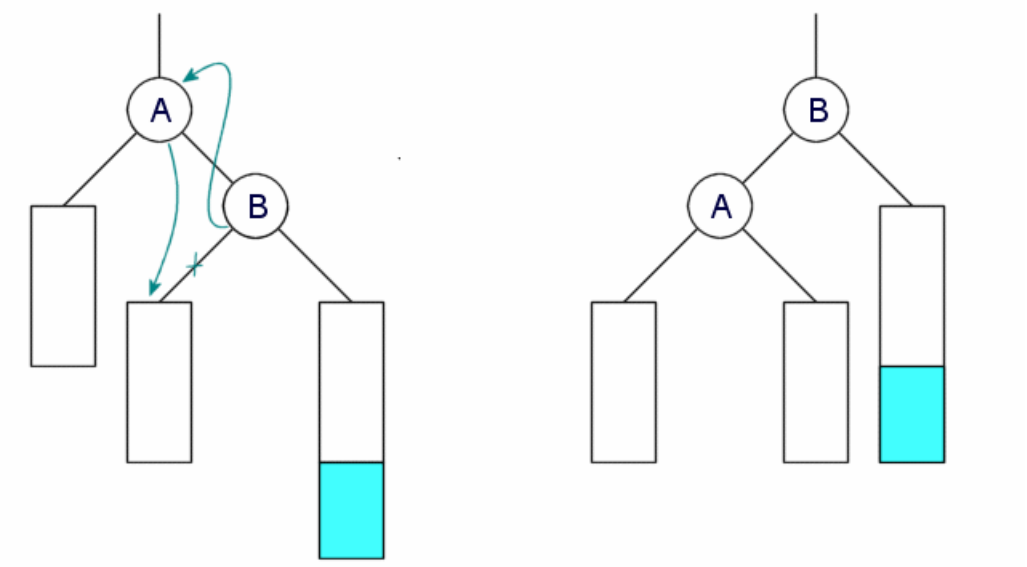


```
else   begin  { rotação dupla ED }

                d := e^.dir;
                e^.dir := d^.esq;
                d^.esq := e;
                t^.esq := d^.dir;
                d^.dir := t;
```



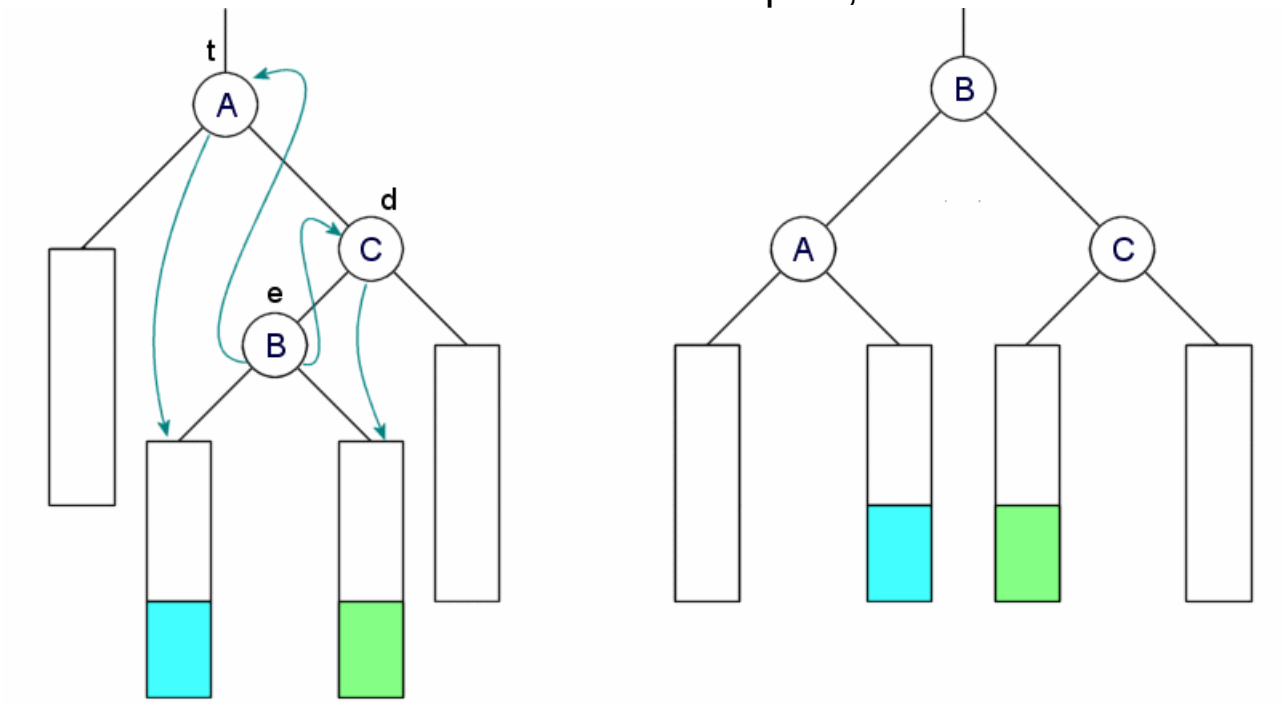

```
                if d^.factor = -1
                then t^.factor := 1
                else t^.factor := 0;

                if d^.factor = 1
                then e^.factor := -1
                else e^.factor := 0;

                t := d
            end;
        t^.factor := 0;
        cresceu := false

    end { reequilíbrio por inserção à esquerda }
  end { case}
end { inserção à esquerda }
```

```
else  if  novo > t^.elem
      then  begin  inserir(novo, t^.dir, cresceu);

                   if cresceu { a sub-árvore direita }
                   then  case t^.factor of

                              -1 :  begin  { equilibrou }
                                           t^.factor := 0;
                                           cresceu := false
                                    end;

                              0 :  t^.factor := 1;
                                   { ainda fica equilibrada }
```




```
                              1 :  begin  { reequilíbrio por inserção à direita }
                                          d := t^.dir;

                                          if d^.factor = 1
                                          then  begin  { rotação simples DD }

                                                       t^.dir := d^.esq;
                                                       d^.esq := t;
                                                       t^.factor := 0;
                                                       t := d
                                                end
```

```
else   begin  { rotação dupla DE }

                  e := d^.esq;
                  d^.esq := e^.dir;
                  e^.dir := d;
                  t^.dir := e^.esq;
                  e^.esq := t;
```




```
                  if e^.factor = 1
                  then t^.factor := -1
                  else t^.factor := 0;

                  if e^.factor = -1
                  then d^.factor := 1
                  else d^.factor := 0;

                  t := e
            end;
      t^.factor := 0;
      cresceu := false

   end { reequilíbrio por inserção à direita }
end { case}
end { inserção à direita }
```

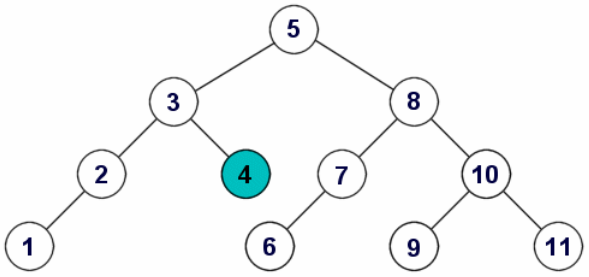
```
        else { afinal o novo elemento já lá estava }
                begin  t^.contador := t^.contador +1;
                       cresceu := false
                end

end; { Inserir }
```

Observações:

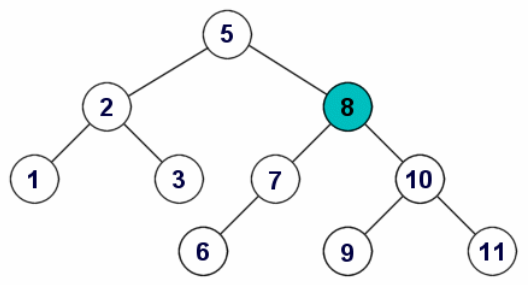
- Se uma inserção modifica a altura de um árvore AVL, não exige reequilíbrio. Se uma inserção exige reequilíbrio, não modifica a altura.
- As árvores AVL devem ser usadas quando o número estimado de operações de gestão é bastante inferior ao das pesquisas.

➡ Remoção de um elemento numa árvore AVL

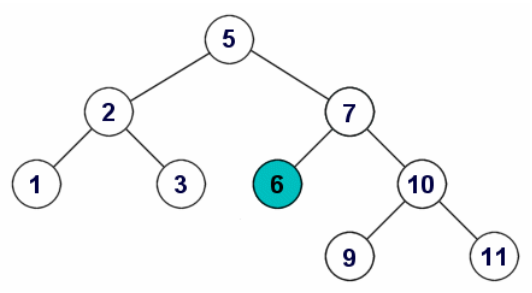
Exemplo: Remover a sequência de vértices: 4 8 6 5 2 1 7

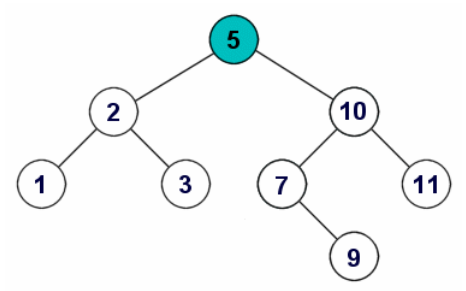


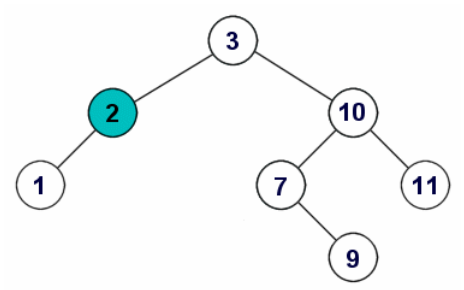
Rotação simples EE



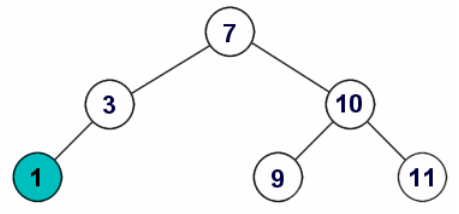
Rotação simples DD

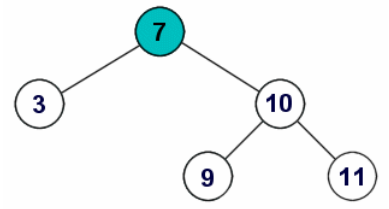




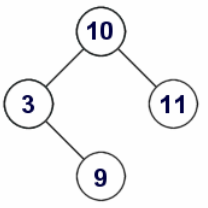


Rotação dupla DE





Rotação dupla ED



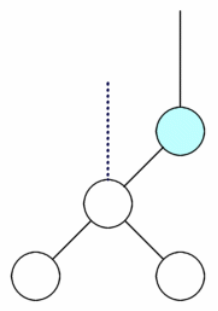
- O processo é análogo ao algoritmo para a remoção em árvores de pesquisa, seguido das necessárias rotações.

Serão utilizados três módulos auxiliares:

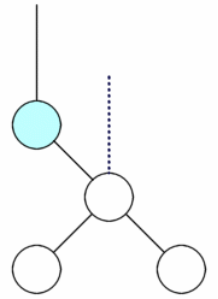
- **rem** : remover descendente mais direito
- **equilibrar1** : quando t^.esq diminuiu em altura
- **equilibrar2** : quando t^.dir diminuiu em altura

```
procedure  remover (  elemento: tipo;
                      var t : AVL; var diminuiu, sucesso : boolean );
var este : AVL;

begin  if t = nil
       then  begin  { não está lá }
                    sucesso := false;
                    diminuiu := false
             end

       else  { procurar em-ordem }
             if  elemento < t^.elem
             then  begin  remover(elemento, t^.esq, diminuiu);

                          if diminuiu { a sub-árvore esquerda }
                          then equilibrar1(t, diminuiu)
                   end

             else  if  elemento > t^.elem
                   then  begin remover(elemento, t^.dir, diminuiu);

                               if diminuiu { a sub-árvore direita }
                               then equilibrar2(t, diminuiu)
                         end

                   else  begin { remover este }
```

```
        else   begin  { remover este }
                      sucesso := true;
                      este := t;
```

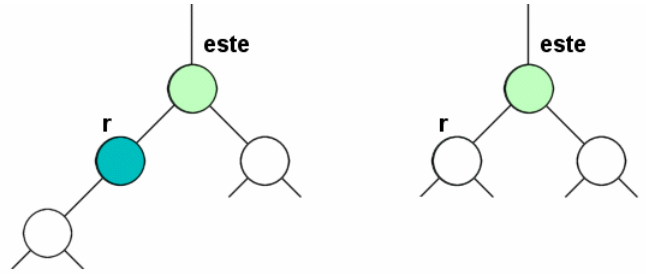


```
                      if este^.dir = nil
                      then  begin  t := este^.esq;
                                   diminuiu := true
                            end
```

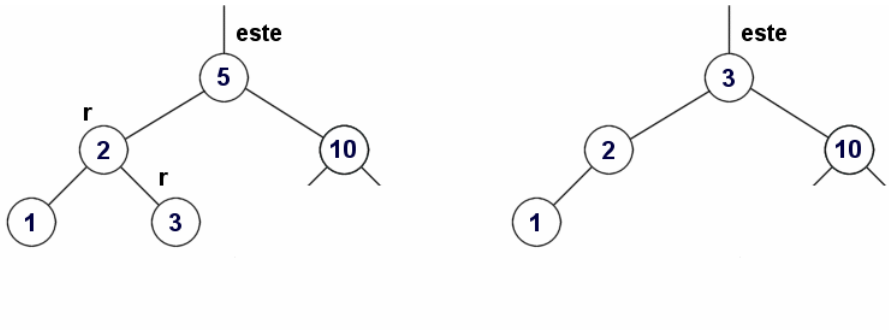


```
                      else  if este^.esq = nil
                            then  begin  t := este^.dir;
                                         diminuiu := true
                                  end

                            else  begin  { tem 2 filhos }
                                         rem(este^.esq, diminuiu);

                                         if diminuiu
                                         then equilibrar1(t, diminuiu)
                                  end

                end

end; { remover }
```

```
procedure rem ( var r : AVL; var diminuiu : boolean );

begin  { a partir de r = este^.esq , remover descendente mais direito }

        if r^.dir = nil
        then  begin  este^.elem := r^.elem;
                     este^.contador := r^.contador;
                     r := r^.esq;
                     diminuiu := true
              end
```

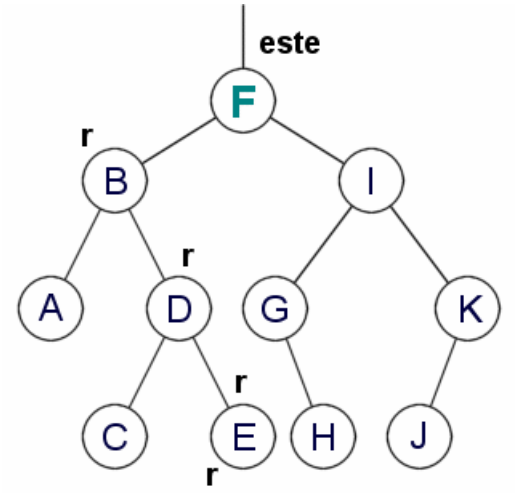


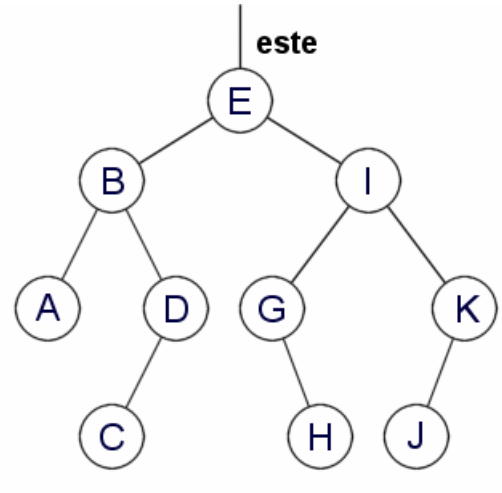

```
        else  begin  rem(r^.dir, diminuiu);
                     if diminuiu
                     then equilibrar2(r, diminuiu)
              end
```




```
end; { rem }
```

exemplo: Para remover (**F**): procurar o anterior (E)
eliminar (E)
substituindo **F** por E.



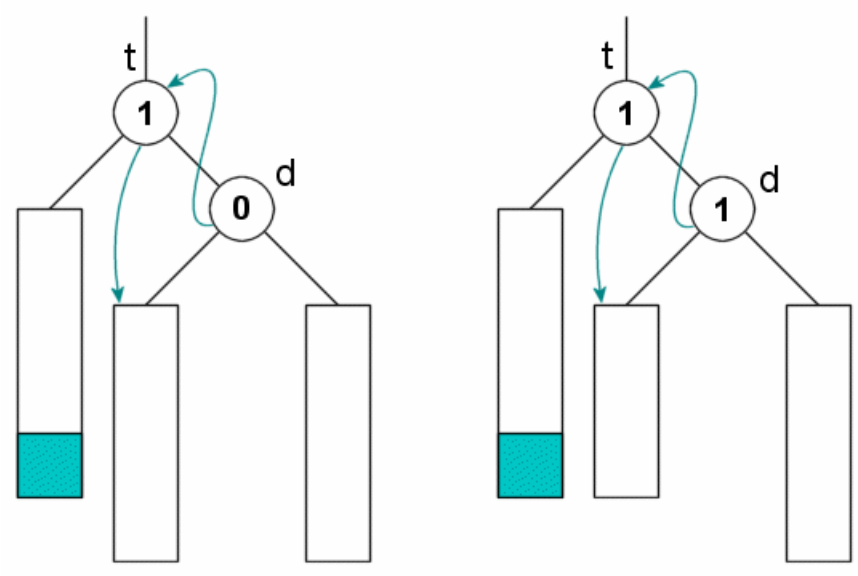


- Módulos para as rotações.

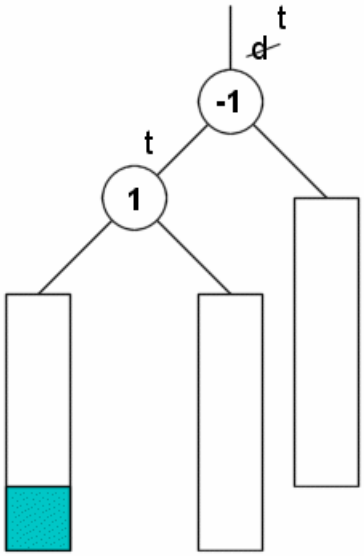
{ Quando t^.esq diminuiu em altura }

```
procedure equilibrar1( var t : AVL; var diminuiu : boolean );
var   e, d : AVL;
      ef, df : -1..1;

begin  case  t^.factor of

        -1 :  { equilibrou }
              t^.factor := 0;

         0 :  begin  { ainda fica equilibrada }
                     t^.factor := 1;
                     diminuiu := false
              end;

         1 :  begin  { reequilíbrio por remoção à esquerda }
```

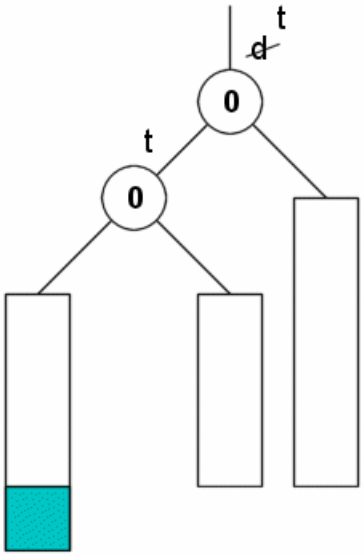
```
1 :   begin  { reequilíbrio por remoção à esquerda }
             d := t^.dir;
             df := d^.factor;
             if df >= 0
             then  begin  { rotação simples DD }
                          t^.dir := d^.esq;
                          d^.esq := t;
```




```
                          if df = 0
                          then  begin  t^.factor := 1;
                                       d^.factor := -1;
                                       diminuiu := false
                                end

                          else  begin  t^.factor := 0;
                                       d^.factor := 0
                                end;
                          t := d
             end
```

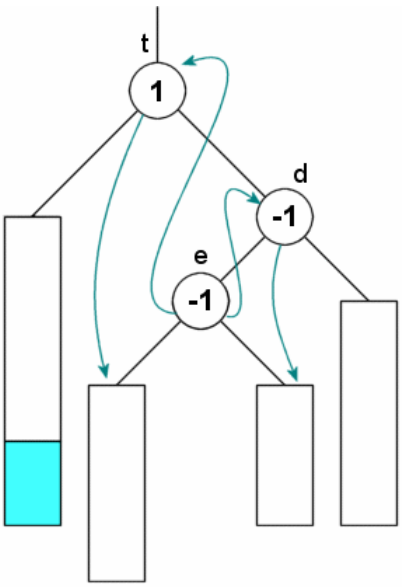


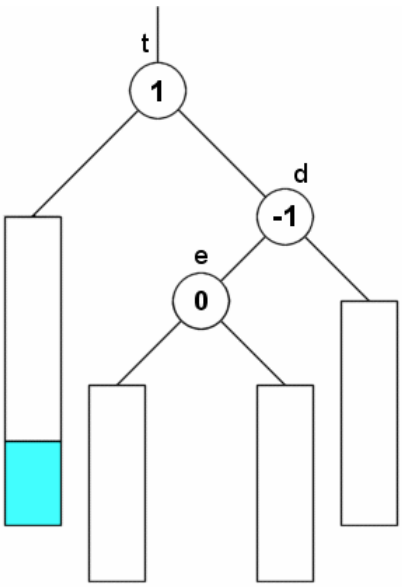
{ não diminuiu }

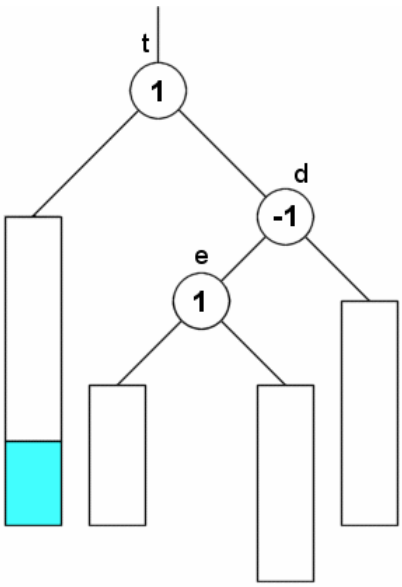


{ diminuiu }

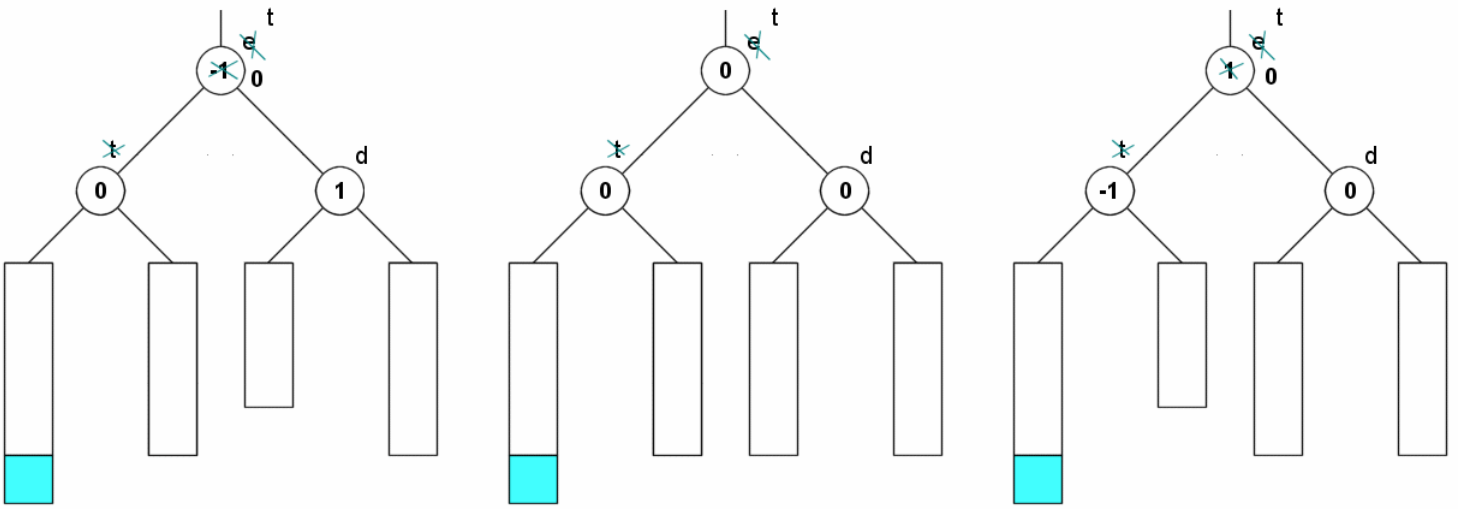
```
else  begin  { df = -1 } { rotação dupla DE }
             e := d^.esq;
             ef := e^.factor;

             d^.esq := e^.dir;
             e^.dir := d;
             t^.dir := e^.esq:
             e^.esq := t;
```







```
                                    { diminuiu }
                                    if ef = 1
                                    then   t^.factor := -1
                                    else   t^.factor := 0;

                                    if ef = -1
                                    then   d^.factor := 1
                                    else   d^.factor := 0;

                                    t := e;
                                    e^.factor := 0
```



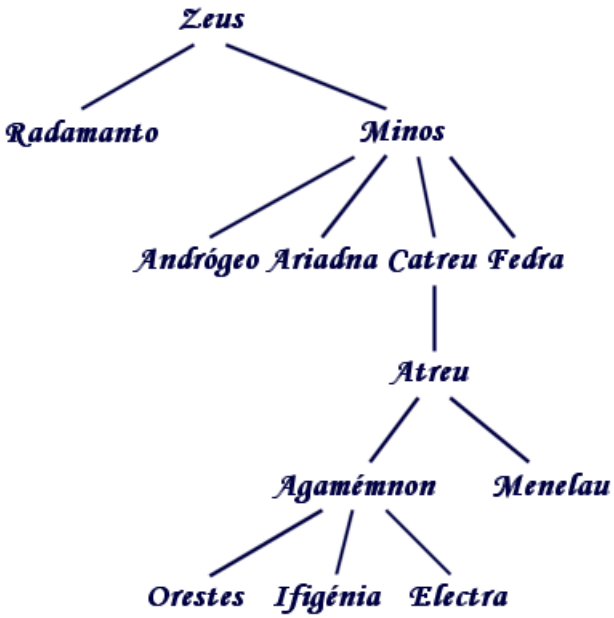

```
                                end { rotação dupla DE }
                            end { t^.factor = 1 }
                        end { case }
end; { equilibrar1 }
```

exercício:

{ Quando t^.dir diminuiu em altura }

```
procedure  equilibrar2(  var t : AVL; var diminuiu : boolean );
```

❒ Árvores e Florestas



{ No contexto das Estruturas de Dados }

Definição: Uma **Árvore** é um conjunto finito de um ou mais **Nós**, tais que:

- Existe um nó particular chamado **raiz**.
- Os restantes nós estão divididos em **n ≥ 0** conjuntos disjuntos $\mathbf{T_1, T_2, ..., T_n}$.
- Cada conjunto $\mathbf{T_i}$, i = 1, ..., n é uma **Árvore**,
- cujas raízes $\mathbf{r_i}$, i = 1, ..., n são filhas da **raiz**.

Terminologia :

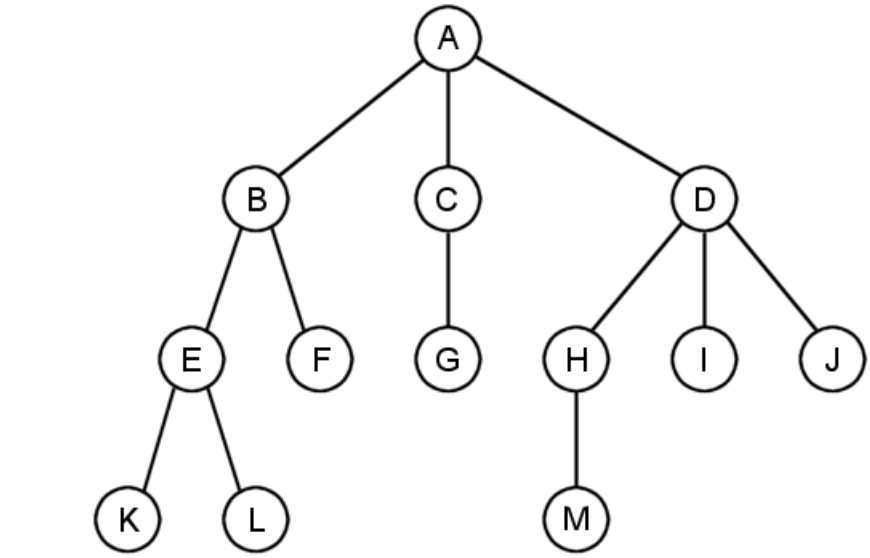
{ Pai, Filho, Irmão, Antepassado, Descendente, Folha, ... }

- Chama-se **Grau** ao número de sub-árvores de um **Nó**.
  O **Grau de uma Árvore** é o máximo dos Graus dos seus Nós.

Representações

- Uma representação com um **número fixo** (k = grau da árvore) de ponteiros por vértice não é satisfatória.
  Do total de n k ponteiros, n (k-1) + 1 seriam nulos.

- Contudo, podemos descrever a estrutura da Árvore por uma sequência de **parêntesis** emparelhados:

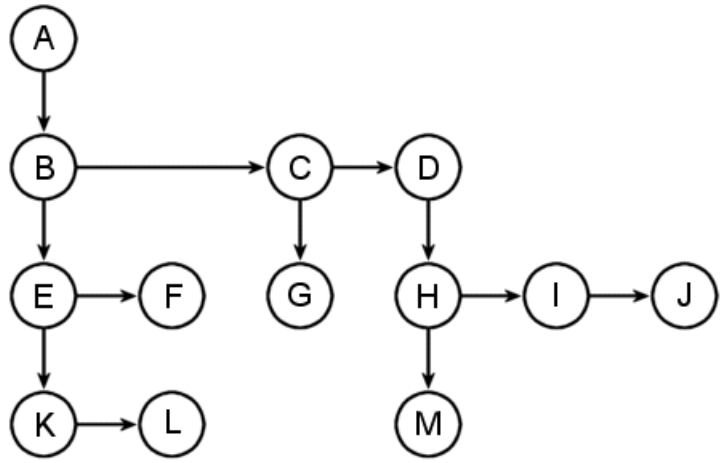


- Este tipo de abordagem permite introduzir a **informação** contida em cada vértice, com vértices irmãos separados por vírgulas.

  Então, a descrição da **Árvore** consiste numa **Lista** (Generalizada):

(A (B (E (K, L), F), C (G), D (H (M), I, J)))

- Assim, é possível uma Representação **Duplamente Ligada** para qualquer Árvore:

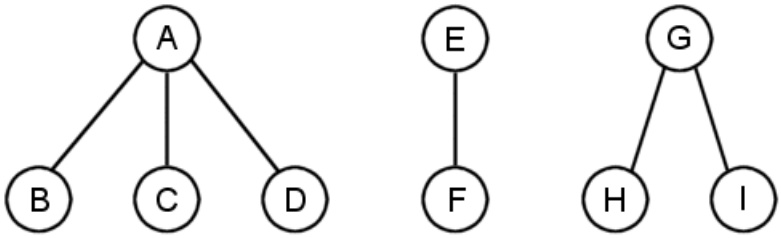


- Ou seja, **toda a Árvore** (ou Floresta) pode ser representada por uma **Árvore Binária**.
  É a chamada **Representação Filho-Irmão**.

Definição: Uma **Floresta** é um conjunto finito ordenado de (zero ou mais) Árvores disjuntas.

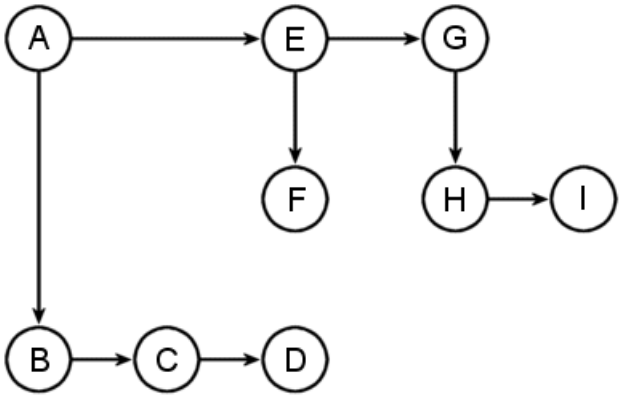
➡ Representação de Florestas

- Segundo a abordagem anterior, podemos descrever uma **Floresta** por uma **Lista**:



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

- E representar essa Lista por uma **Árvore Binária**:



- Mais formalmente:

  Seja $T_1, T_2, ..., T_n$, com $n \geq 0$, uma **Floresta**.

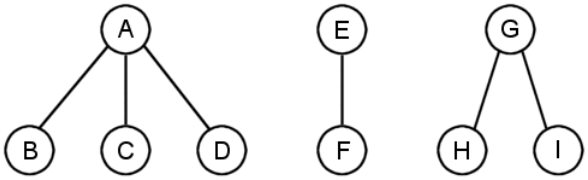
  A **Árvore Binária** $B(T_1, T_2, ..., T_n)$ que a **representa** é definida por:

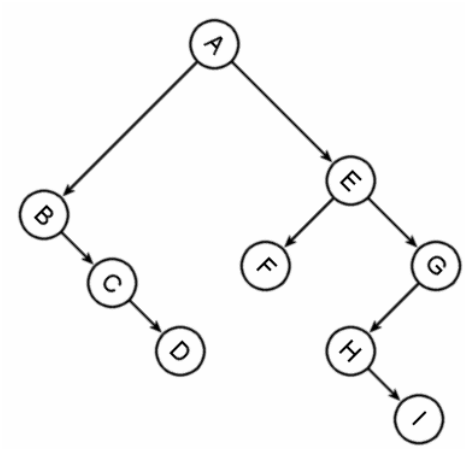
  a) B é vazia se e só se $n = 0$;

  b) B não vazia tem:
  raiz igual à raiz de T_1,
  sub-árvore esquerda igual a $B(T_{11}, T_{12}, ..., T_{1m})$
  { sub-árvores de T_1 }
  sub-árvore direita igual a $B(T_2, T_3, ..., T_n)$

- Deste modo, qualquer floresta pode ser univocamente representada por uma árvore binária.

Travessias de uma Floresta





- **Travessar Floresta** $T_1, T_2, ..., T_n$ em **pré-ordem** (também chamada **ordem dinástica**):

  Se $n \neq 0$ então:
  - visitar raiz de T_1 ;
  - **Travessar Floresta** $T_{11}, T_{12}, ..., T_{1m}$ em **pré-ordem** ;
  - **Travessar Floresta** $T_2, T_3, ..., T_n$ em **pré-ordem** .

  { A B C D E F G H I }

- **Travessar Floresta** $T_1, T_2, ..., T_n$ **em-ordem**:

  Se $n \neq 0$ então:
  - **Travessar Floresta** $T_{11}, T_{12}, ..., T_{1m}$ **em-ordem** ;
  - visitar raiz de T_1 ;
  - **Travessar Floresta** $T_2, T_3, ..., T_n$ **em-ordem** .

  { B C D A F E H I G }

- **Travessar Floresta** $T_1, T_2, ..., T_n$ em **pos-ordem**:

  Se $n \neq 0$ então:
  - **Travessar Floresta** $T_{11}, T_{12}, ..., T_{1m}$ em **pos-ordem** ;
  - **Travessar Floresta** $T_2, T_3, ..., T_n$ em **pos-ordem** ;
  - visitar raiz de T_1 .

  { D C B F I H G E A }

- De acordo com a **definição** anterior, o resultado da travessia de uma floresta é **igual** ao da travessia, na mesma ordem, da árvore binária que a representa.

- Contudo, muitos autores adoptam uma **definição** baseada no conceito de travessar sequencialmente cada uma das árvores. Nesse caso:

- **Travessar Floresta** $T_1, T_2, ..., T_n$ em **pré-ordem** :

  Se $n \neq 0$ então:
  - visitar raiz de T_1 ;
  - **Travessar as Árvores** $T_{11}, T_{12}, ..., T_{1m}$ em **pré-ordem** ;
  - **Travessar as Árvores** $T_2, T_3, ..., T_n$ em **pré-ordem** .

  { A B C D E F G H I }

- O resultado continua a ser equivalente.
  No entanto:

- **Travessar Floresta** $T_1, T_2, ..., T_n$ em **pos-ordem** :

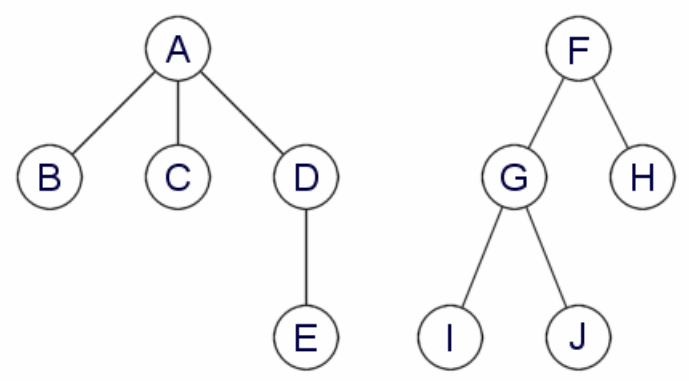
  Se $n \neq 0$ então:
  - **Travessar as Árvores** $T_{11}, T_{12}, ..., T_{1m}$ em **pos-ordem** ;
  - visitar raiz de T_1 ;
  - **Travessar as Árvores** $T_2, T_3, ..., T_n$ em **pos-ordem** .

  { B C D A F E H I G }

- Neste caso, a travessia em **pos-ordem** da floresta é equivalente à travessia **em-ordem** da árvore binária que a representa.

Representação de Florestas por ordem-nível

- A informação associada a cada vértice é listada em **pré-ordem** num ficheiro, precedida do **nível** a que se encontra.

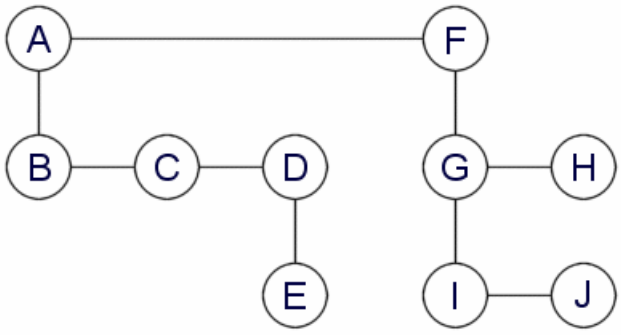



```
1 A
      2 B
      2 C
      2 D
            3 E
1 F
      2 G
            3 I
            3 J
      2 H
```

Conversão de representações: ordem-nível → filho-irmão

- Dada a lista da informação contida nos vértices em pré-ordem, com a indicação dos níveis, construir a respectiva árvore binária.

```
1 A
2 B
2 C
2 D
3 E
1 F
2 G
3 I
3 J
2 H
```



- Para elaborar uma versão iterativa do processo é necessária uma **Pilha**, que permitirá decidir se as ligações a efectuar deverão ser horizontais ou verticais.

```
type  ArvoreFI  = ^no;
              no  =  record  elem : tipo;
                             filho, irmao : ArvoreFI
                     end;
          registo  =  record  nivel : natural;
                             ponteiro : ArvoreFI
                     end;
...

var S : PilhadeRegistos;
```

- O TAD PilhadeRegistos é definido por:

criar(S)
vazia(S)
por(nivel, ponteiro, S)
tirar(S)
topo(nivel, ponteiro, S)

```
procedure converter_representacoes (  var F : text;
                                      var raiz : ArvoreFI;
                                      var sucesso : boolean );
var   n, n_ant : natural;
      t, t_ant : ArvoreFI;
      info : tipo;
      S : PilhadeRegistos;
      erro : boolean;

begin  criar(S);
       erro := false;

       { criar raiz com o primeiro elemento e guardar na Pilha }
       reset(F);
       readln(F, n, info);
       new(raiz);
       with raiz^ do
            begin  filho := nil;  irmao := nil;
                   elem := info
            end;
       por(n, raiz, S);

       while not (eof(F) or erro) do
            begin  { nó seguinte }
                   readln(F, n, info);
                   new(t);
                   with t^ do
                        begin  filho := nil;  irmao := nil;
                               elem := info
                        end;

                   { Onde ligá-lo? Consultar a Pilha }
                   topo(n_ant, t_ant, S);

                   if n > n_ant
                   then  { filho }
                         t_ant^.filho := t
```

```
                else  begin  { n ≤ n_ant }
                             { irmão : procurar na Pilha }

                             while n < n_ant do
                                   begin  tirar(S);
                                          topo(n_ant, t_ant, S)
                                   end;
                             { aqui n ≥ n_ant }

                             if n > n_ant
                             then  { impossível ... porquê? }
                                   erro := true

                             else begin  { n = n_ant }
                                         t_ant^.irmao := t;
                                         { t_ant  desnecessário }
                                         tirar(S)
                                  end
                        end; { irmão }

                { guardar este na Pilha }
                por(n, t, S);

           end; { while }
      { árvore construída ou erro nos dados }
      sucesso := not erro

end; { converter_representacoes }
```

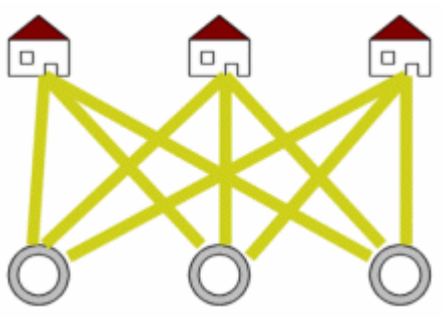
exercícios:

- Estando os elementos em pré-ordem, este processo admite uma versão recorrente, naturalmente sem Pilha.
- Estude também o problema inverso.

❒ Grafos

$G = (V , E)$

$V = \{ \text{vértices} \}$
$E = \{ \text{arestas} \}$



Os Grafos são:

- Uma importante área da Matemática discreta.
- Uma fértil área das Ciências da Computação.
- Com centenas de algoritmos conhecidos.
- E com milhares de aplicações práticas vitais.

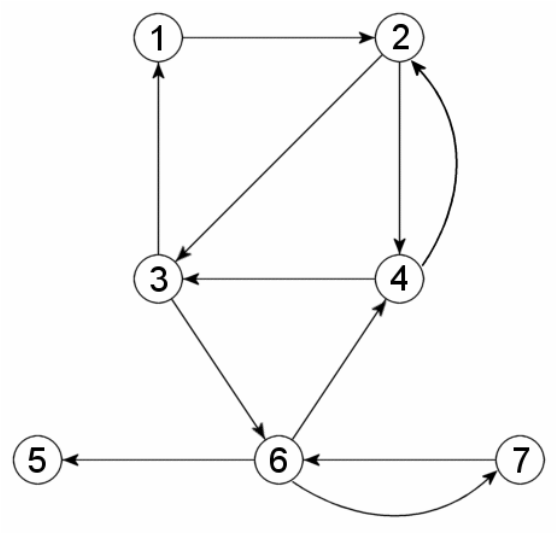
➡ O TAD Grafo

- **Operações**:

criar(G)	{ criar um Grafo vazio }
vazio(G)	{ verificar se o Grafo é vazio }
porVertice(v, G)	{ colocar um novo vértice }
porAresta(v, u, G)	{ colocar uma aresta entre dois vértices }
tirarVertice(v, G)	{ remover um vértice }
tirarAresta(v, u, G)	{ remover uma aresta }
consultar(v, G)	{ o elemento contido num vértice }
adjacentes(v, G)	{ o conjunto dos vértices adjacentes }

- Paralelamente ao desenvolvimento de algoritmos em Grafos, é importante a elaboração de estruturas de dados adequadas.

⇨ Grafos Orientados *(digrafos)*

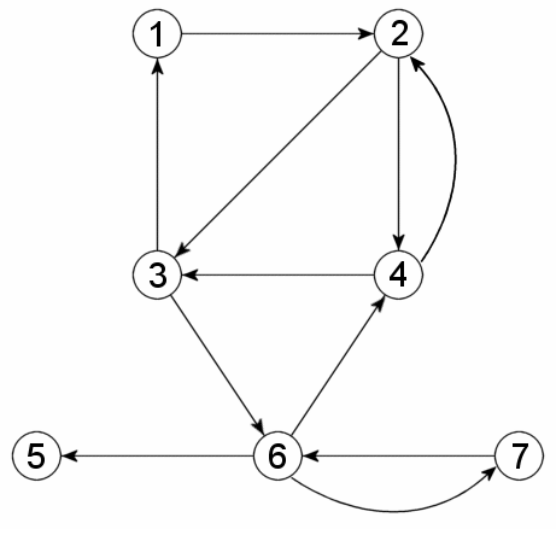


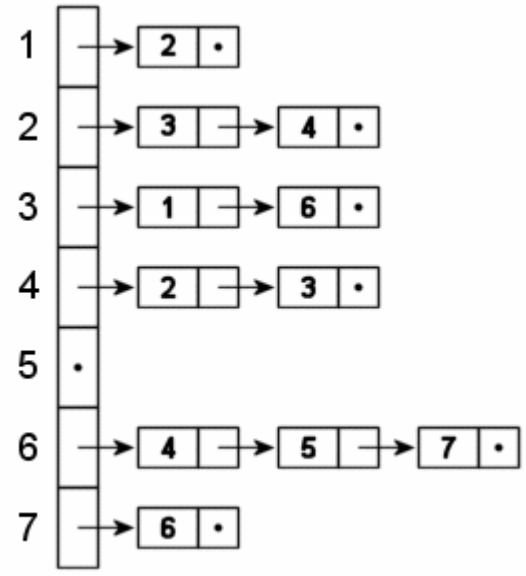
	1	2	3	4	5	6	7
1	0	1	0	0	0	0	0
2	0	0	1	1	0	0	0
3	1	0	0	0	0	1	0
4	0	1	1	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	1	1	0	1
7	0	0	0	0	0	1	0

Matriz de Adjacências

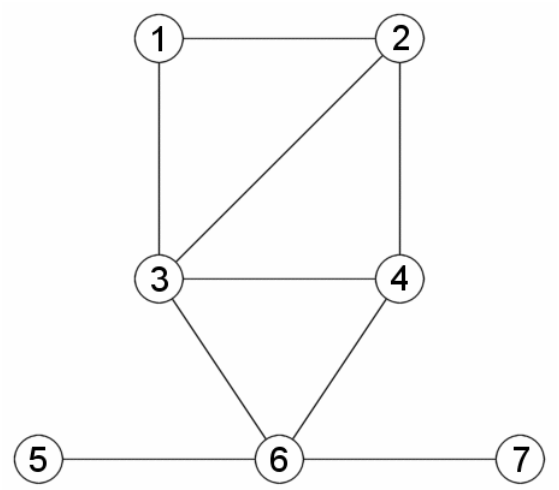
- Na maior parte dos casos, a representação por uma Matriz de Adjacências não é satisfatória, por ser demasiado grande e extremamente **rarefeita**.

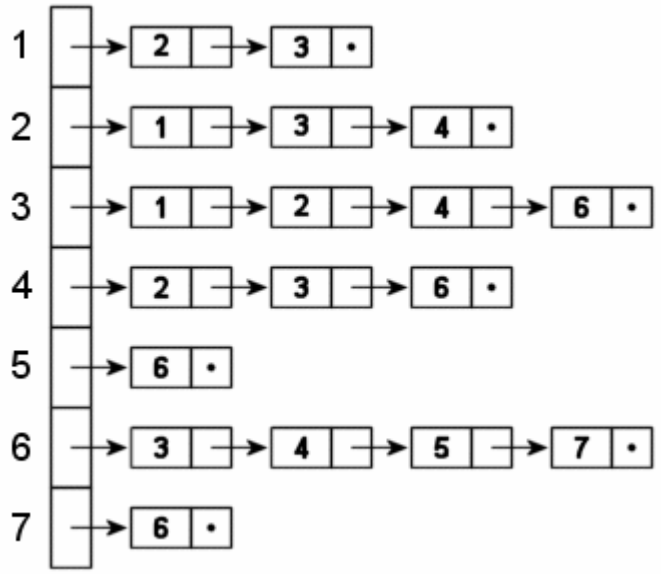
- **Representação** por um vector de **Listas de Adjacências**:





⇨ Grafos *(não-orientados)*





⇨ Redes