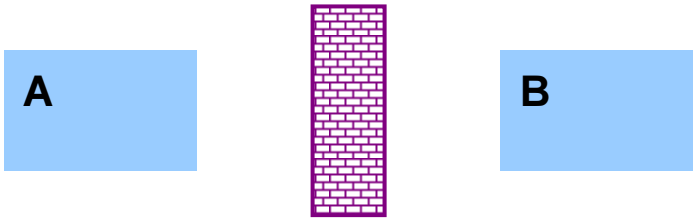
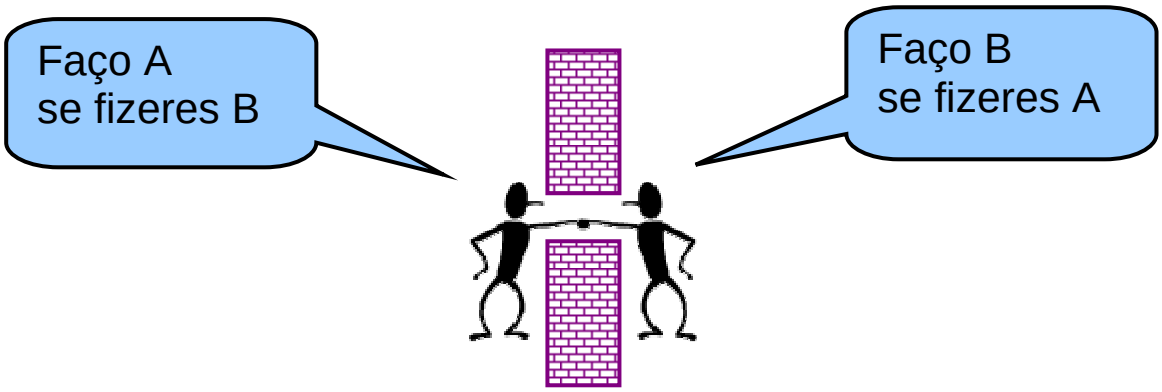
Capítulo 5 : Tipos Abstractos de Dados

{ Programação Estruturada }

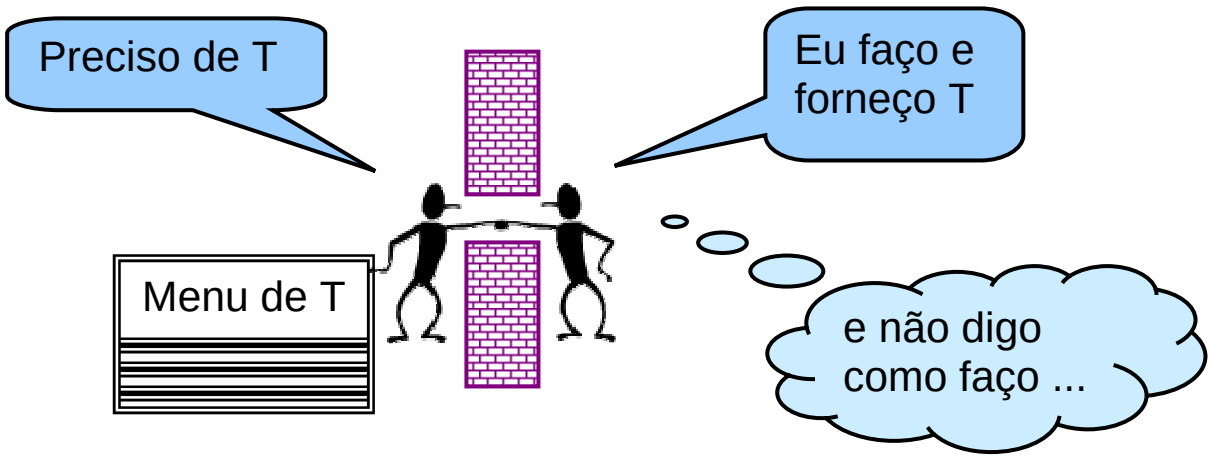
A Modularidade dos Algoritmos:

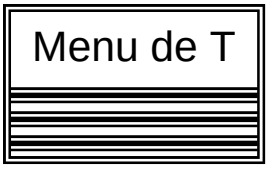


A Comunicação entre os Algoritmos:



A Modularidade das Estruturas de Dados:





- Chama-se **Tipo Abstracto de Dados** (TAD) a um conjunto de **Operações** sobre uma colecção de **Dados**.

Exemplo: **Lista Linear Ordenada TAD**

{ Definição de Lista Linear Ordenada, em termos de um conjunto de operações. }

criar(L)	{ criar uma lista vazia }
comprimento(L)	{ número de elementos }
vazia(L)	{ verificar se a lista é vazia }
consultar(k, L)	{ consultar o elemento de ordem k }
inserir(e, L)	{ inserir ordenadamente um elemento }
remover(e, L)	{ remover um elemento }
listar(L)	{ listagem sequencial }

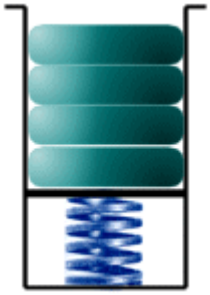
- A definição do TAD Lista Linear Ordenada é **independente** da Estrutura de Dados escolhida para a sua **representação** (array, ligada, circular, ...)

Exemplo: **Polinómio de uma variável real TAD**

criar(P)	{ criar o polinómio nulo }
nulo(P)	{ verificar se o polinómio é nulo }
grau(P)	{ o maior expoente }
coeficiente(k, P)	{ o coeficiente do termo de expoente k }
inserir(c, e, P)	{ inserir termo de coeficiente c e expoente e }
somar(A, B, C)	{ somar polinómios }
escrever(P)	{ escrever o polinómio }

❒ O TAD Pilha (*Stack*)

{ O último elemento a entrar
é o primeiro a sair.
(**LIFO**: Last In First Out) }



- **Operações**:

criar(S)	{ criar uma Pilha vazia }
vazia(S)	{ verificar se a Pilha está vazia }
por(e, S)	{ colocar um elemento no topo da Pilha }
tirar(S)	{ remover o elemento do topo da Pilha }
topo(S)	{ consultar o elemento do topo da Pilha }

{ Create(S); IsEmpty(S); Push(e, S); Pop(S); Top(s); }

Operadores e Axiomas:

$$\begin{aligned} \text{criar} &: \emptyset \rightarrow S \\ \text{vazia} &: S \rightarrow \{ \text{verdadeiro, falso} \} \\ \text{por} &: e \times S \rightarrow S \\ \text{tirar} &: S \rightarrow S \\ \text{topo} &: S \rightarrow e \end{aligned}$$

vazia(criar) = verdadeiro
vazia(por(e, S)) = falso
tirar(criar) = “erro”
topo(criar) = “erro”
tirar(por(e, S)) = S
topo(por(e, S)) = e

- **Representações**:

  O TAD Pilha pode ser representado por Estruturas de Dados muito simples, em particular por uma Lista Ligada Linear, pois todos os acessos são feitos ao primeiro elemento:

  Exercício: Implemente as cinco operações.

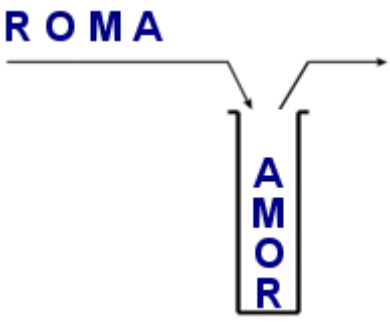
- **Aplicações**:

  As aplicações do TAD Pilha são variadas e importantes.
  A sua utilização mais comum consiste na "transformação" de algoritmos recorrentes em algoritmos iterativos.

Inverter uma palavra:

{ Ler uma palavra, terminada por um espaço,
e escrever as suas letras por ordem inversa }

```
var  c : char;
     S : PilhadeCaracteres;

     ...
     criar(S);
     repeat  read(c);
             por(c, S)
     until c = ' ';

     repeat  c:= topo(S);
             write(c);
             tirar(S)
     until vazia(S);
     ...
```



Porque é ineficiente o cálculo recorrente do factorial?

{ Utilizando uma Pilha, simular a execução do Algoritmo Recorrente para o cálculo do factorial de n }

Algoritmo Recorrente:

$\forall n \in \mathbb{N}_0$ se n = 0
então n! ← 1
senão n! ← n * (n-1)!

Simulação:

```
var  n : integer;
     S : PilhadeInteiros;

     ...
     criar(S);
     por(n, S);

     while  n > 0  do
            begin  n:= n - 1;
                   por(n, S)
            end;
     { n = 0 }

     tirar(S); { tirar o zero }
     factorial:= 1;

     while  not vazia(S)  do
            begin  factorial:= factorial * topo(S);
                   tirar(S)
            end;
     ...
```

Verificar concordância de parêntesis :

(2 + (x ∗ (1 + (x / 2)) – ((2 ∗ (x + 1))) ∗ x)

{ Dada uma expressão aritmética, contendo parêntesis, verificar se estão ou não equilibrados. }

- Percorrendo a expressão da esquerda para a direita, cada parêntesis '**(**' é colocado na Pilha e, por cada parêntesis '**)**', é removido um elemento da Pilha.
  Os parêntesis estão equilibrados se a Pilha estiver vazia no fim (e só no fim) da expressão.

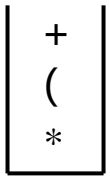
```
...
criar(S);
continua:= true;
while  continua and not eoln(input) do
        begin { Ler um caracter }
                read(c);
                { Ignorar todos os outros caracteres }
                if c in ['(', ')']
                then  { É um parêntesis }
                        if c = '(',
                        then  por(c, S)
                        else  if not vazia(S)
                                then tirar(S)
                                else continua:= false
        end;
{ Pilha vazia V fim de linha }

{ Pilha vazia Λ ¬ fim de linha ⇒ Parêntesis ')' a mais }
{ ¬ Pilha vazia Λ fim de linha ⇒ Parêntesis '(' a mais }

if vazia(S) and eoln(input)
then ...  { Parêntesis Equilibrados }
else ...  { Parêntesis Desequilibrados }
...
```

Versão iterativa da conversão *infix* → *postfix* :

{ Uma Pilha é utilizada
para guardar os operadores e os parêntesis '(' }

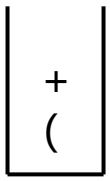
Exemplo 1: a ∗ (b + c) → a b c + ∗

car = ’a’	escrever (’a’)		
car = ’∗’	por (’∗’, S)		
car = ’(’	por (’(’, S)		
car = ’b’	escrever (’b’)		
car = ’+’	por (’+’, S)		
car = ’c’	escrever (’c’)		
car = ’)’	topo (S) = ’+’ ;	escrever (’+’) ;	tirar (S)
	topo (S) = ’(’ ;	tirar(S)	
	topo (S) = ’∗’ ;	escrever (’∗’) ;	tirar (S)



Exemplo 2: (a + b) ∗ c → a b + c ∗

car = ’(’	por (’(’, S)		
car = ’a’	escrever (’a’)		
car = ’+’	por (’+’, S)		
car = ’b’	escrever (’b’)		
car = ’)’	topo (S) = ’+’ ;	escrever (’+’) ;	tirar (S)
	topo (S) = ’(’ ;	tirar (S)	
car = ’∗’	por (’∗’, S)		
car = ’c’	escrever (’c’)		
	topo (S) = ’∗’ ;	escrever (’∗’) ;	tirar (S)



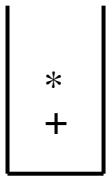
Percorrendo a expressão da esquerda para a direita:

- cada operando é escrito;
- cada parêntesis '(' é colocado na Pilha;
- cada parêntesis ')' faz escrever e remover operadores da Pilha;
- cada operador é colocado na Pilha, mas …

… mas como resolver o problema da Prioridade dos operadores?

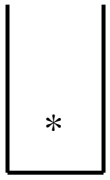
Exemplo 3: a + b ∗ c → a b c ∗ +

car = 'a'	escrever ('a')	
car = '+'	por ('+', S)	
car = 'b'	escrever ('b')	
car = '∗'	por ('∗', S)	
car = 'c'	escrever ('c')	
	topo (S) = '∗' ;	escrever ('∗') ; tirar (S)
	topo (S) = '+' ;	escrever ('+') ; tirar (S)



Exemplo 4: a ∗ b + c → a b ∗ c +

car = 'a'	escrever ('a')	
car = '∗'	por ('∗', S)	
car = 'b'	escrever ('b')	
car = '+'	topo (S) = '∗' ;	
	{ '∗' > '+' }	
		escrever ('∗') ; tirar (S)
	por ('+', S)	
car = 'c'	escrever ('c')	
	topo (S) = '+' ;	escrever ('+') ; tirar (S)



- cada operador é colocado na Pilha, mas apenas se tiver prioridade superior à do operador no topo da Pilha.
  Caso contrário, só é colocado na Pilha depois de removidos e escritos todos os operadores de prioridade superior, ou até encontrar um parêntesis '('.

- Para simplificar, as expressões serão representadas por vectores de caracteres.

```
type   expressao = array [1 .. max] of char;
       tipodecar = (operando, operador, abrir, fechar);
```

- Assumimos definido o TAD PilhadeCaracteres e implementadas as cinco operações básicas.

- Construímos ainda as duas funções auxiliares:

```
function  prioridade ( op : char) : integer;
          begin  if op in ['+', '-', '*', '/' ]
                 then  case  op of
                             '+', '-'  : prioridade := 1;
                             '*', '/'  : prioridade := 2
                       end
                 else : prioridade := 0
          end;
```

```
function  tipode ( car : char) : tipodecar;
          begin  if car in ['+', '-', '*', '/' ]
                 then tipode:= operador
                 else if car = '('
                      then tipode:= abrir
                      else if car = ')'
                           then tipode:= fechar
                           else tipode:= operando
          end;
```

```
procedure converter ( infix : expressao; cinf : integer;
                                  var postfix : expressao; var cpos : integer);
var   car, cartopo : char;
      i : integer;
      pronto : boolean;
      S : PilhadeCaracteres;
begin  criar(S); cpos:= 0;
       for  i:= 1 to cinf do
       begin  car:= infix[i];
              case tipode(car) of

operando :   begin  cpos:= cpos +1;
                    postfix[cpos]:= car
             end;

abrir : por(car, S);

fechar :  begin    cartopo := topo(S);
                   while  cartopo <> '(' do
                          begin  tirar(S);
                                 cpos:= cpos +1;
                                 postfix[cpos]:= cartopo;
                                 cartopo:= topo(S)
                          end;
                   tirar(S)
            end;

operador :  begin  pronto:= false;
                   while  not vazia(S) and not pronto do
                          begin  cartopo:= topo(S);
                                 if (cartopo <> '(') and
                                (prioridade(cartopo) >= prioridade(car))
                                 then begin   cpos:= cpos +1;
                                              postfix[cpos]:= cartopo;
                                              tirar(S)
                                      end
                                 else pronto:= true
                          end;
                   por(car, S)
            end;

            end {case};
      end {for};
```

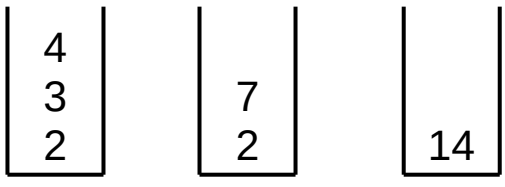
```
        { Só falta acabar de copiar os operadores que ficaram na Pilha }

        while  not vazia() do
               begin  cartopo:= topo(S);
                      tirar(S);
                      cpos:= cpos +1;
                      postfix[cpos]:= cartopo;
               end

end {converter};
```

- Os operandos são escritos na expressão *postfix* pela mesma ordem com que aparecem na expressão *infix* dada.
- Operações com a mesma prioridade são efectuadas pela mesma ordem,

$$a +_1 b +_2 c +_3 d \rightarrow a\ b +_1 c +_2 d +_3$$

- O procedimento anterior assume que a expressão *infix* dada está correcta. Tente introduzir alguma validação de dados e consequentes mensagens de erro …
- O valor de uma expressão aritmética escrita em *postfix* pode ser calculado por um algoritmo recorrente ou por um algoritmo iterativo bastante simples, utilizando uma Pilha de números reais.

Exemplo : $2 * (3 + 4) \rightarrow 2\ 3\ 4\ +\ *\ =\ 14$



Uma versão iterativa das Torres de Hanoi :

{ "Transformação directa" do Algoritmo Recorrente. }

```
MoverTorre (n, origem, auxiliar, destino):
        MoverTorre(n-1, origem, destino, auxiliar);
        MoverDisco(origem, destino);
        MoverTorre(n-1, auxiliar, origem, destino).
```

- Consideremos uma Pilha, cujos elementos são "movimentos",

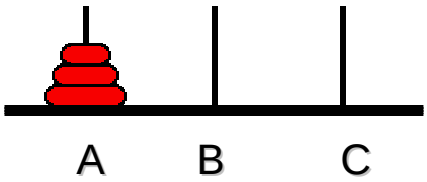
```
type movimento = record  n : natural;
                         origem, auxiliar, destino : torre
                 end;
var   S : PilhadeMovimentos;
```

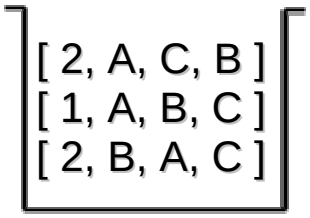
- e adaptemos as operações básicas, de modo a facilitar os acessos aos quatro campos do registo.

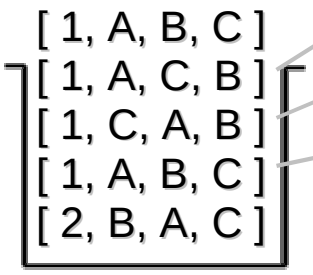
```
...
criar(S);
write('Quantos discos?'); readln(n);
por(n, 'A', 'B', 'C', S);
repeat  topo(k, orig, aux, dest, S);
        tirar(S);
        if k = 1
        then writeln(orig, ' --> ', dest)
        else begin  por(k-1, aux, orig, dest, S);
                    por(  1,  orig, aux, dest, S);
                    por(k-1, orig, dest, aux, S)
             end
until vazia(S);
...
```

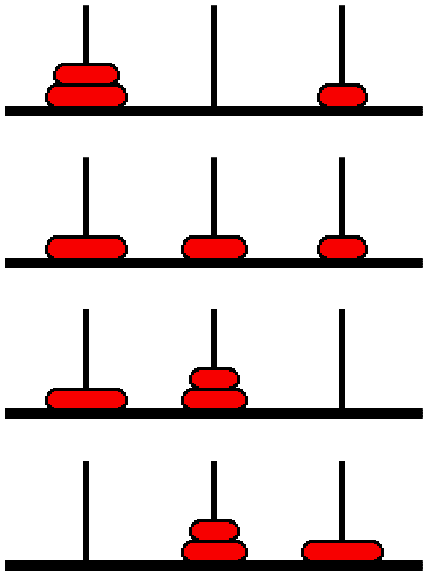
- Note que os "movimentos" são colocados na Pilha, por ordem inversa à das chamadas do Algoritmo Recorrente.

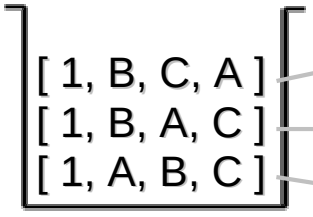
- Simulação para (3, A, B, C)

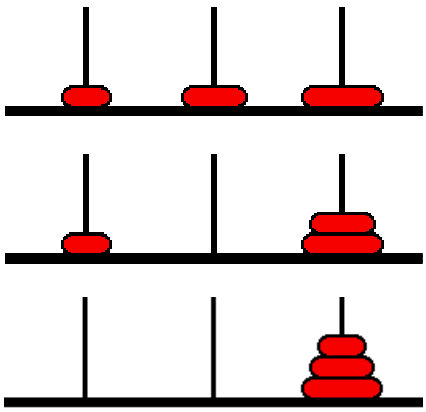




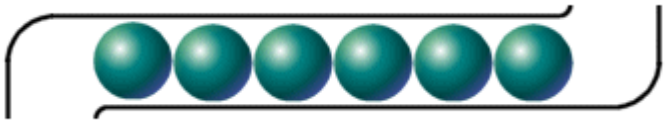








O TAD Fila (*Queue*)



{ O primeiro elemento a entrar é o primeiro a sair.
(**FIFO**: First In First Out) }

- **Operações**:

criar(Q)	{ criar uma Fila vazia }
vazia(Q)	{ verificar se a Fila está vazia }
juntar(e, Q)	{ juntar um elemento no fim da Fila }
remover(Q)	{ remover o primeiro elemento da Fila }
frente(Q)	{ consultar o primeiro elemento da Fila }

Operadores e Axiomas:

criar :	$\emptyset \rightarrow Q$
vazia :	$Q \rightarrow$ { verdadeiro, falso }
juntar :	$e \times Q \rightarrow Q$
remover :	$Q \rightarrow Q$
frente :	$Q \rightarrow e$

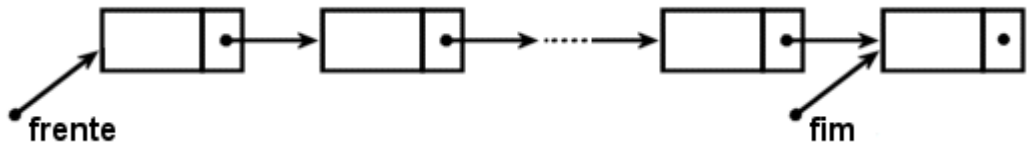
vazia(criar) = verdadeiro
vazia(juntar(e, Q)) = falso
remover(criar) = “erro”
frente(criar) = “erro”
remover(juntar(e, Q)) { se vazia(Q)
então = criar(Q)
senão = juntar(e, remover(Q)) }
frente(juntar(e, Q)) { se vazia(Q)
então = e
senão = frente(Q) }

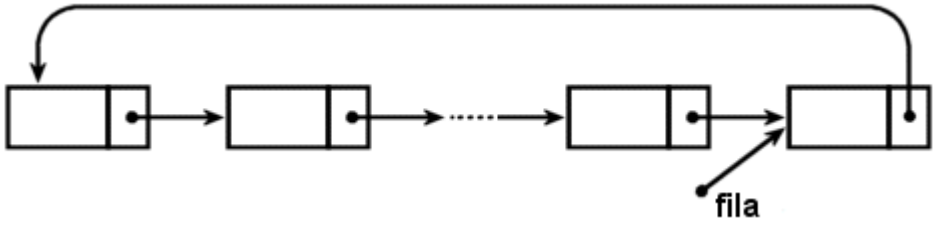
- Os dois conjuntos de Axiomas caracterizam univocamente o funcionamento dos TADs Pilha e Fila. Note-se que a "assinatura" dos dois conjuntos de Operadores é idêntica.

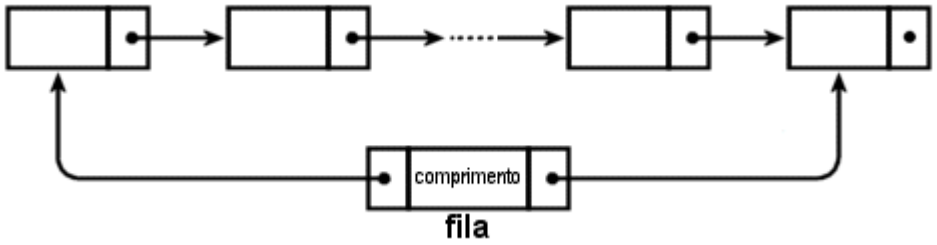
- **Representações**:

  No TAD Fila são necessários acessos a ambos os extremos. Contudo, deve ser escolhida uma Estrutura de Dados onde a complexidade da Operações Básicas seja independente do comprimento da Fila.

  Exemplos:







  Exercício: Implemente as cinco operações em cada uma das Estruturas propostas, tendo em conta que a complexidade de cada operação deve ser $\mathcal{O}(1)$.

- **Aplicações**:

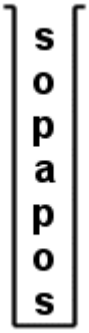
  O TAD Fila tem uma vasta gama de aplicações práticas: Filas de Espera e Simulação de Processos (redes de comunicação, tráfego, ...), Gestão de Recursos em Sistemas Operativos, ...

Verificar se uma dada palavra é uma capicua (palíndromo)

{ A palavra **capicua** provém do catalão
capicua (cap + i + cua), que significa: "cabeça e cauda" }
{ A palavra **palíndromo** provém do grego
palíndromos, que significa: "que corre para trás" }

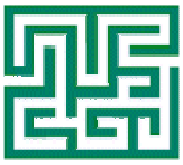
- Basta portanto verificar se a palavra tem "cabeça igual à cauda" ou, se "corre para trás" como "corre para a frente":

```
var  c : char;
     S : PilhadeCaracteres;
     Q : FiladeCaracteres;

     ...
     criar(S); criar(Q);
     while  not eoln(input)  do
            begin  read(c);
                   por(c, S);
                   juntar(c, Q)
            end;

     capicua:= true;
     while  not vazia(S)  and not vazia(Q) and capicua do
            if  topo(S) = frente(Q)
            then  begin  tirar(S);
                         remover(Q)
                  end
            else capicua:= false
     ...
```



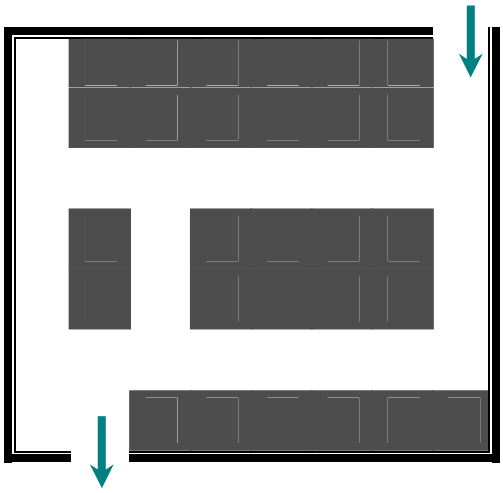
{ A utilização de Pilhas e Filas na construção de Algoritmos }

❑ Um Problema: Resolução de Labirintos



- **Representação do Labirinto**

Para simplificar, o Labirinto será representado por uma matriz de elementos booleanos.



var labirinto : **array** [0 .. m+1, 0 .. n+1] **of** boolean;

As casas já visitadas serão registadas em:

var visitadas : **array** [0 .. m+1, 0 .. n+1] **of** integer;

A partir de cada casa [i, j], os quatros movimentos possíveis (N, E, S, W) serão calculados, por soma com elementos da matriz:

	1	2
1	-1	0
2	0	1
3	1	0
4	0	-1

var move : **array** [1 .. 4, 1 .. 2] **of** integer;

As coordenadas das casas de Entrada e de Saída serão, respectivamente, [ie, je] e [is, js].

- **Solução Recorrente – Algoritmo de *Backtracking***

{ Ver Cap. 3, pág. 12 }

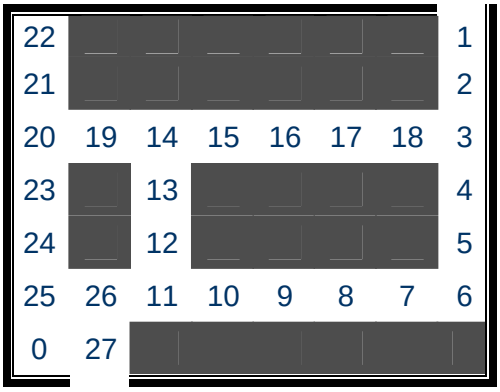
...

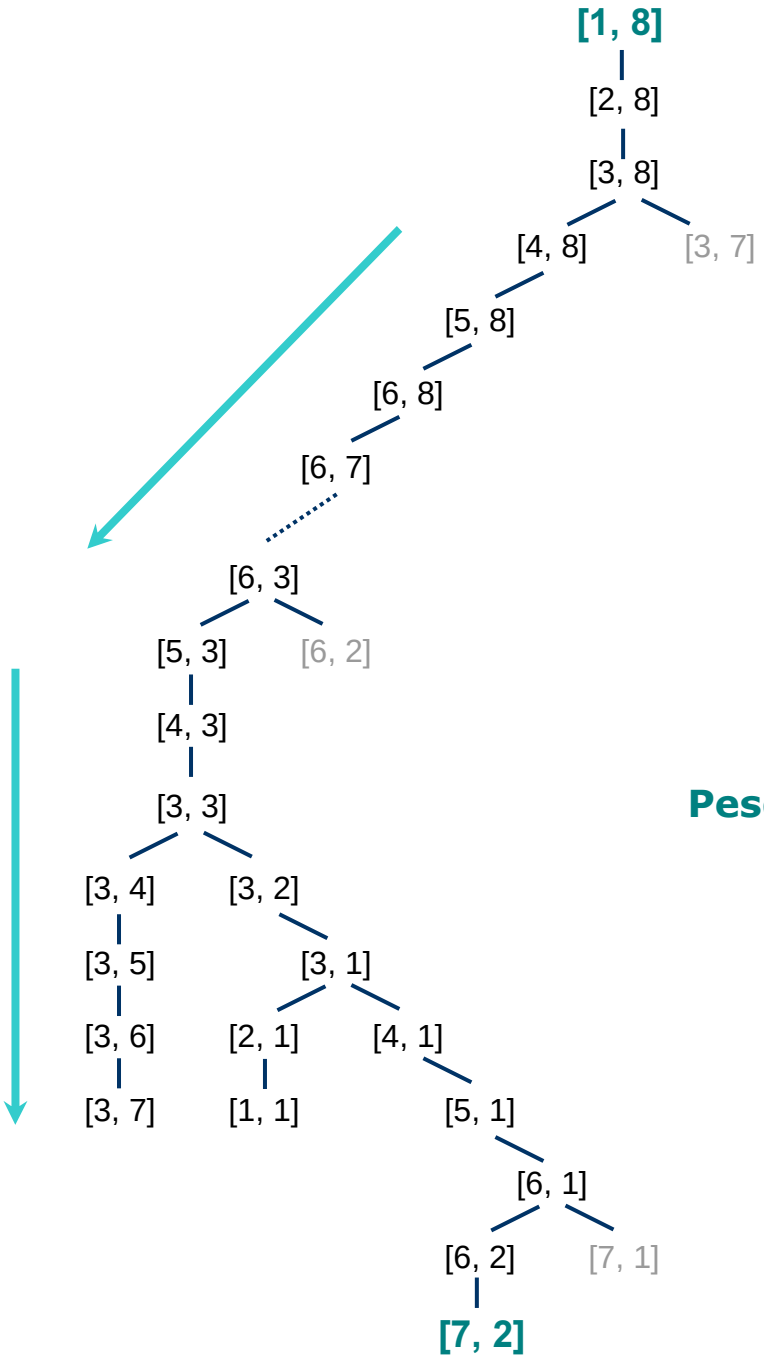
```
procedure   TentarAndar ( i, j : integer);
     var proxi, proxj, dir : integer;

     begin  for  dir:= 1 to 4 do
                 begin  proxi:= i + move[dir, 1];
                        proxj:= j + move[dir, 2];

                        if labirinto[proxi, proxj] and (visitadas[proxi, proxj] = 0)
                        then  begin  ordem:= ordem + 1;
                                     visitadas[proxi, proxj]:= ordem;
                                     if (proxi = is) and (proj = js)
                                     then  begin  EscreverSolucao(labirinto);
                                                  goto 13
                                           end
                                     else TentarAndar(proxi, proxj)
                                     { Recuar }
                              end
                 end
     end { TentarAndar };


begin  { Programa Principal }
       Ler(labirinto, m, n, ie, je, is, js);
       Inicilalizar(labirinto, m, n);
       { colocar false nas linhas 0 e m+1 e nas colunas 0 e n+1 }
       Construir(visitadas);
       { colocar 0 em [1..m, 1..n] e maxint nas linhas 0 e m+1
                                                  e nas colunas 0 e n+1 }
       TentarAndar(ie, je);
       writeln(' Labirinto sem solução ');
  13 : { instrução nula }

end.
```

{ O **goto** permite interromper o processo recorrente, logo que encontrada a primeira solução. }

- **Análise da Solução Recorrente – Árvore de Pesquisa**

{ A solução obtém-se, a partir da casa de saída, recuando sempre para a casa adjacente de menor ordem, diferente de zero. }





Pesquisa em Profundidade

(depth-first)

- **Correspondente Versão Iterativa**

Utilizamos uma Pilha, cujos elementos são "passos",

```
type passo =  record   linha, coluna : integer;
                       sentido : 1..4
              end;

var   S : PilhadePassos;
```

e adaptamos as cinco operações básicas:

criar(S)
vazia(S)
por(linha, coluna, sentido, S)
tirar(S)
topo(linha, coluna, sentido, S)

Modificamos também,

```
var visitadas : array [0 .. m+1, 0 .. n+1] of boolean;
```

pois o resultado do percurso fica na Pilha.

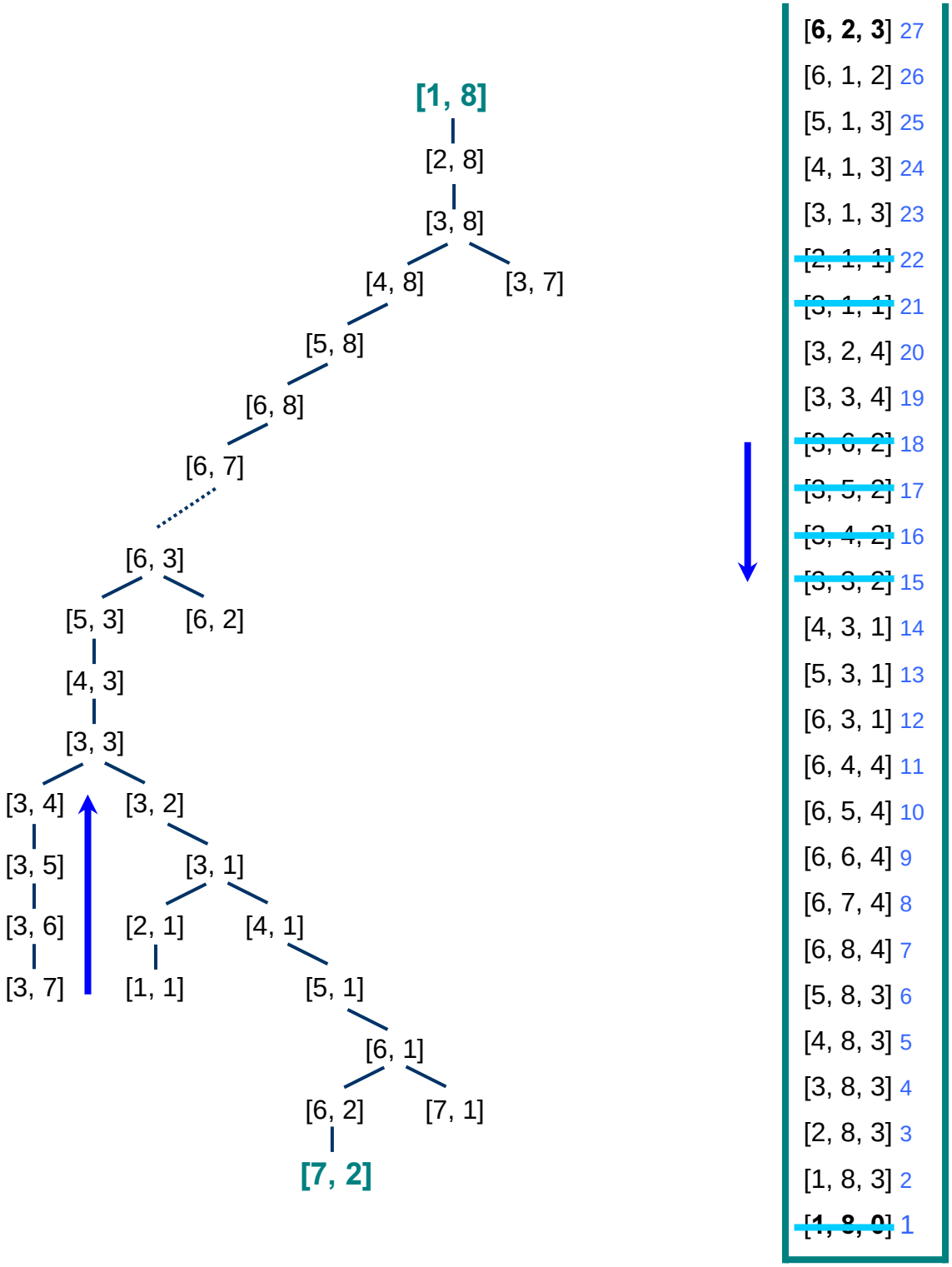
{ *Como alternativa, poderíamos utilizar uma Pilha com elementos do tipo* [linha, coluna]*. Nesse caso, os elementos da matriz* visitadas *teriam de ser inteiros.* }

```
...
criar(S);
por(ie, je, 0, S); { casa de entrada }

achou:= false;
while  not (achou or vazia(S)) do
       begin  topo(i, j, k, S);
              k:= k+1; { tentar sentido seguinte }
              tirar(S);

              while (k <= 4) and not achou do
                     begin  proxi:= i + move[dir, 1];
                            proxj:= j + move[dir, 2];

                            if labirinto[proxi, proxj] and not visitadas[proxi, proxj]
                            then  begin  { saída? }
                                         achou:= (proxi = is) and (proj = js);
                                         visitadas[proxi, proxj]:= true;

                                         por(i, j, k, S);

                                         { andar }
                                         i:= proxi; j:= proxj; k:= 0
                                  end;
                            k:= k+1
                     end

       end;
...

if achou
then EscreverSolucao { registada em S }
else writeln(' Labirinto sem solução '); { Pilha vazia }
```

{ Gestão da Pilha }

{ Equivalente às sucessivas chamadas de TentarAndar(i, j) }

- O funcionamento da Pilha descreve a estrutura da Árvore de Pesquisa. As colocações dos sucessivos "passos" na Pilha representam as "descidas" na Árvore e, nos casos de *Backtracking*, as remoções correspondem a "subidas" na Árvore.



- Um processo de *Backtracking* (ou a sua simulação com uma Pilha) gera sempre uma **Travessia em Pré-Ordem** na Árvore de Pesquisa das Soluções.

- No final do processo, o percurso realizado fica na Pilha. A Pilha só esvazia quando o Labirinto não tem solução.

- A solução é a primeira a ser encontrada e não necessariamente a **Solução Óptima** do problema.

- **Versão Iterativa utilizando uma Fila**

  Neste caso, vamos utilizar uma Fila cujos elementos são,

```
type casa =  record   linha, coluna : integer
             end;

var   Q : FiladeCasas;
```

  com as cinco operações básicas:

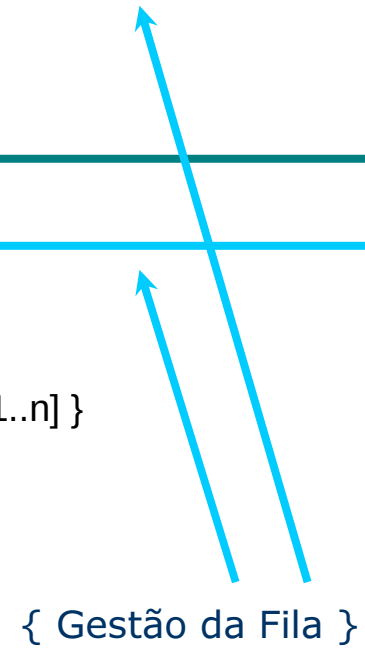
```
criar(Q)
vazia(Q)
juntar(linha, coluna, Q)
remover(Q)
frente(linha, coluna, Q)
```

  Vai ser necessária a matriz,

```
var visitadas : array [0 .. m+1, 0 .. n+1] of integer;
```

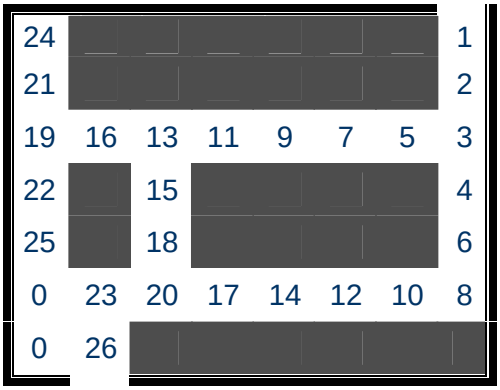
  onde o resultado do percurso fica registado, pelos sucessivos valores da variável, **var** ordem : integer;

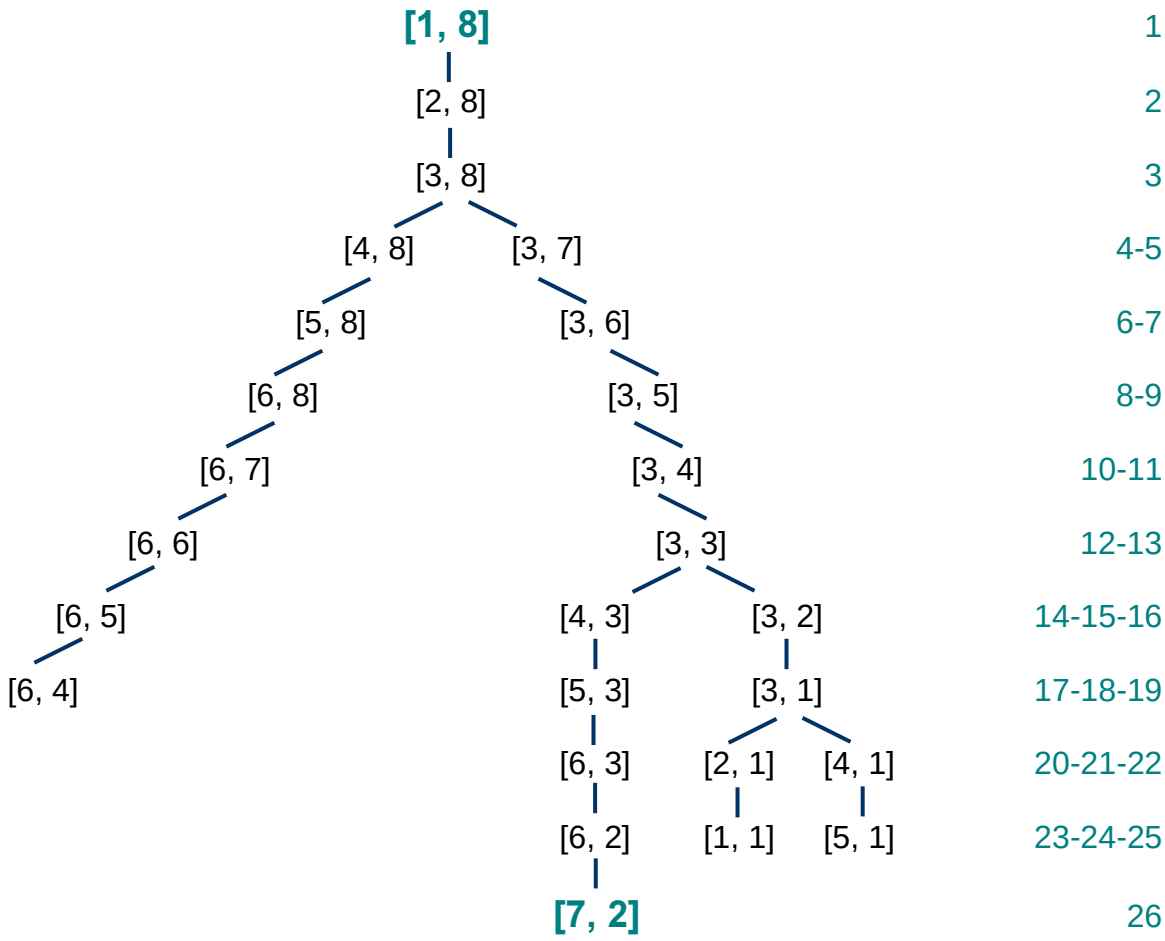
```
...
criar(Q);
juntar(ie, je, Q); { casa de entrada }
visitadas[ie, je]:= 1;
ordem:= 1;

achou:= false;

while  not (achou or vazia(Q)) do
       begin  frente(i, j, Q); { caminho alternativo }
              remover(Q);

              k:= 1; { tentar casas vizinhas }

              while (k <= 4) and not achou do
                     begin  proxi:= i + move[dir, 1];
                            proxj:= j + move[dir, 2];

                            if labirinto[proxi, proxj] and (visitadas[proxi, proxj] = 0)
                            then  begin  { saída? }
                                         achou:= (proxi = is) and (proj = js);
                                         ordem:= ordem+1;
                                         visitadas[proxi, proxj]:= ordem;
                                         { novo caminho alternativo }

                                         juntar(proxi, proxj, Q);

                                  end;
                            k:= k+1
                     end

       end;

...

if achou
then EscreverSolucao { registada em visitadas[1..m, 1..n] }
else writeln(' Labirinto sem solução ');
```



- **Análise da Solução – Árvore de Pesquisa**

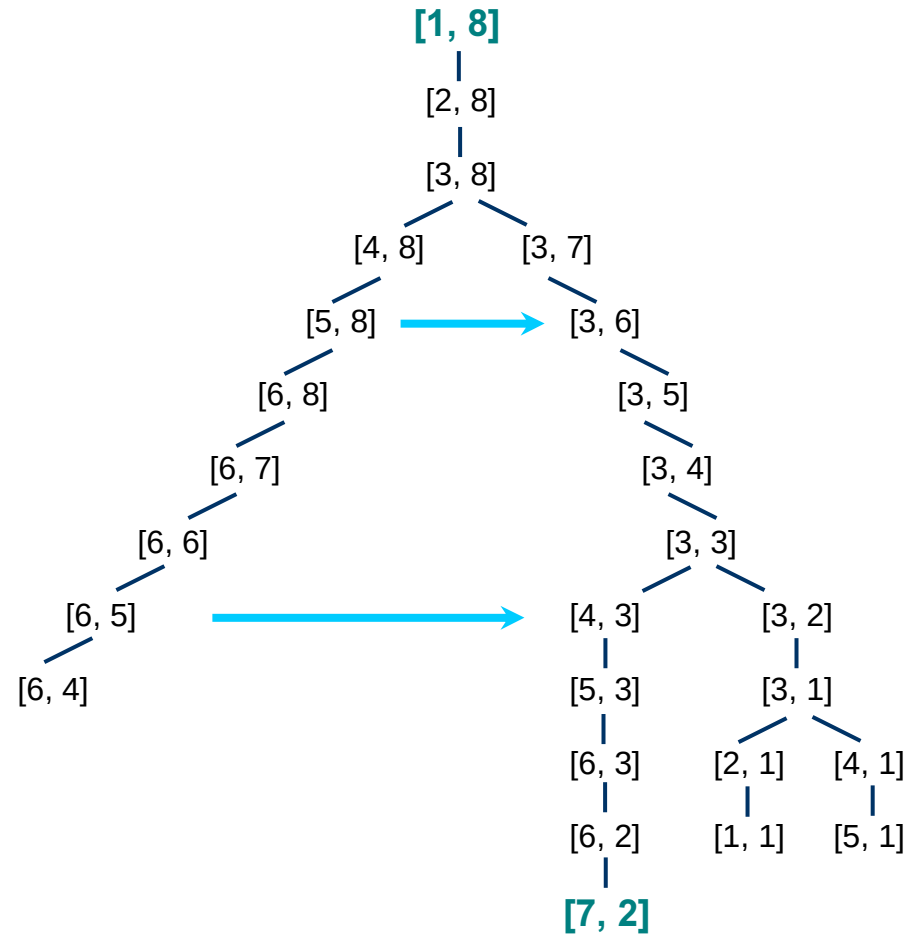
{ A solução obtém-se de modo análogo ao anterior. }





Pesquisa em Largura
(breadth-first)

- A ordem pela qual as sucessivas "casas" são colocadas na Fila corresponde a uma **Travessia por Níveis** na Árvore de Pesquisa.
- No final do processo a Fila fica vazia.



[1, 8] [2, 8] [3, 8] [4, 8] [3, 7] [5, 8] [3, 6] [6, 8] [3, 5] [6, 7] [3, 4] [6, 6] [3, 3] [6, 5]

[4, 3] [3, 2] [6, 4] [5, 3] [3, 1] [6, 3] [2, 1] [4, 1] [6, 2] [1, 1] [5, 1] **[7, 2]**

- É encontrada a **Solução Óptima** do problema, pois todas as soluções são efectivamente testadas.

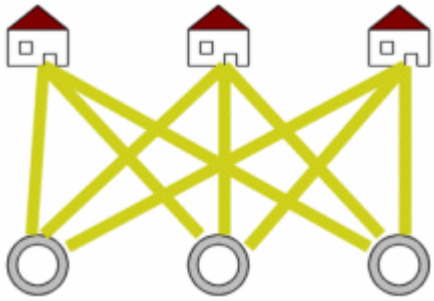
- Os labirintos são apenas um modelo para o estudo de um conjunto muito importante de problemas matemáticos, com uma vasta gama de aplicações práticas: redes (telefónicas, WWW, ATMs, …) projecto de circuitos VLSI, tráfego, …

- Claro que, para problemas reais, a Estrutura de Dados que representa o "labirinto" não pode ser uma matriz.

❒ O TAD Grafo

G = (V , E)

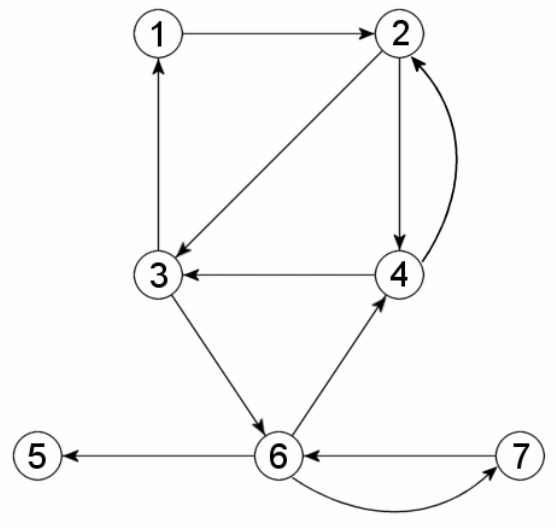
V = { vértices }
E = { arestas }



- **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 }

⇨ Grafos Orientados *(digrafos)*

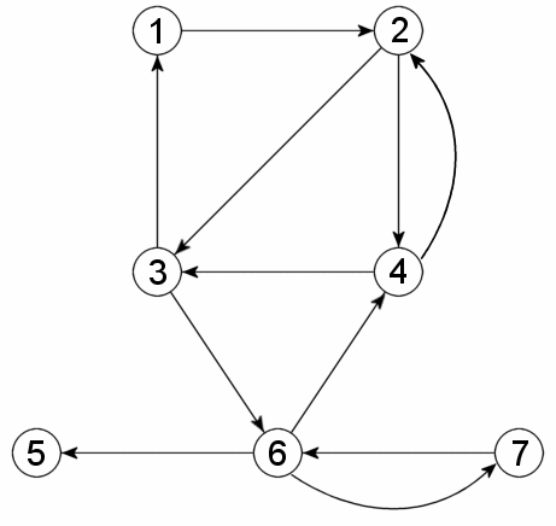


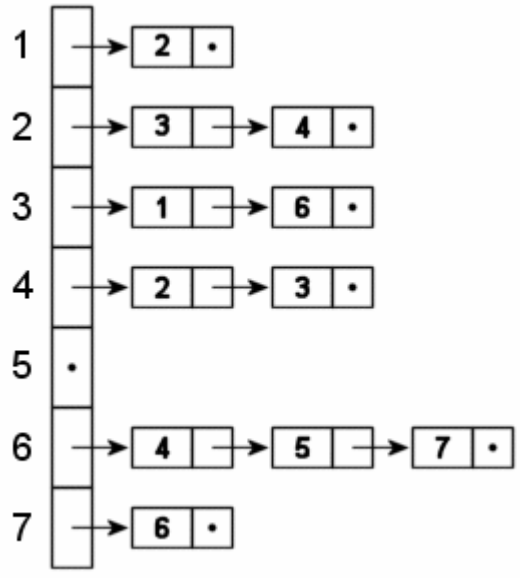
	1	2	3	4	5	6	7
1	0	1	0	0	0	0	0
2	0	0	1	1	0	0	0
3	1	0	0	0	0	1	0
4	0	1	1	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	1	1	0	1
7	0	0	0	0	0	1	0

Matriz de Adjacências

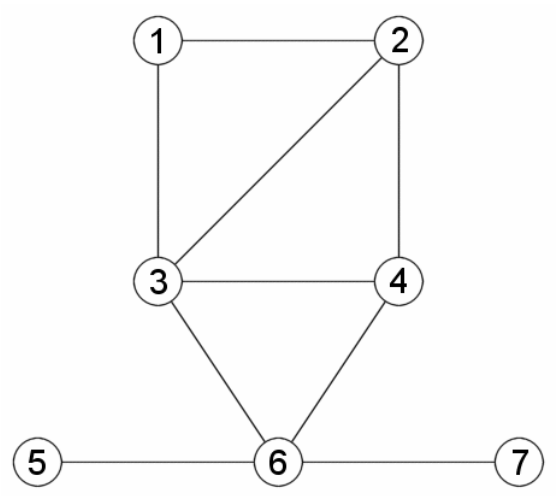
- Na maior parte dos casos, a Matriz de Adjacências é demasiado grande e muito rarefeita.

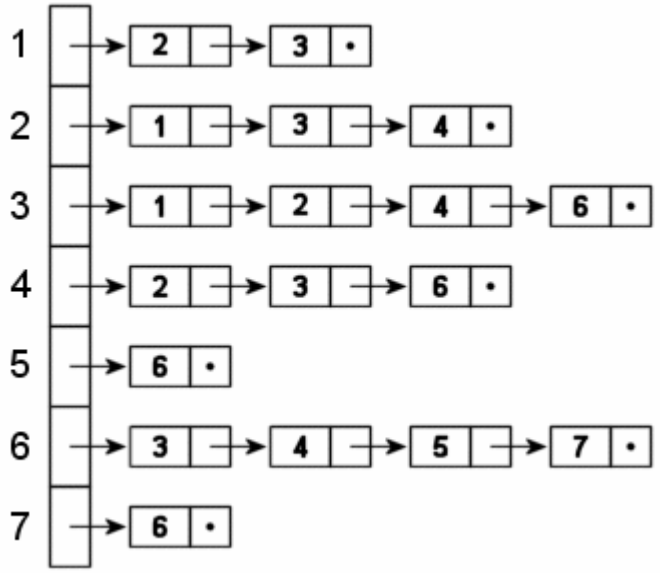
- **Representação** por um vector de **Listas de Adjacências**:



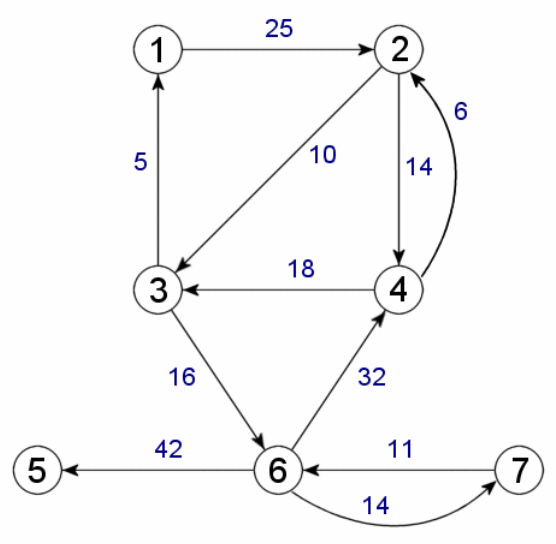


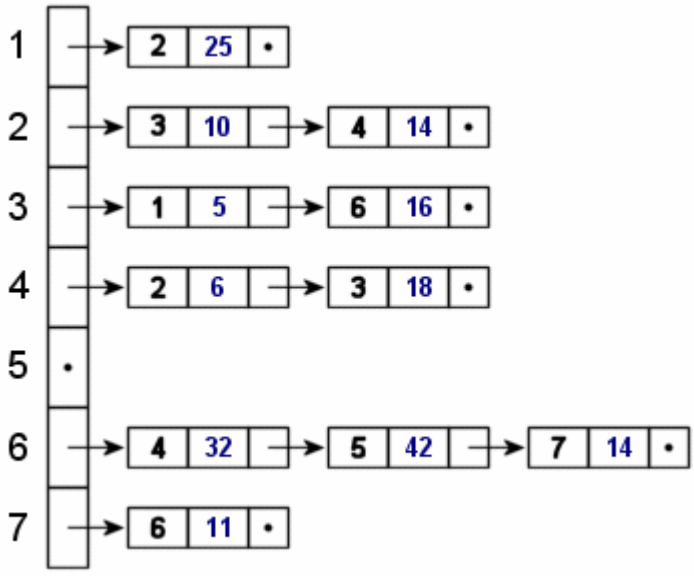
⇨ Grafos *(não-orientados)*





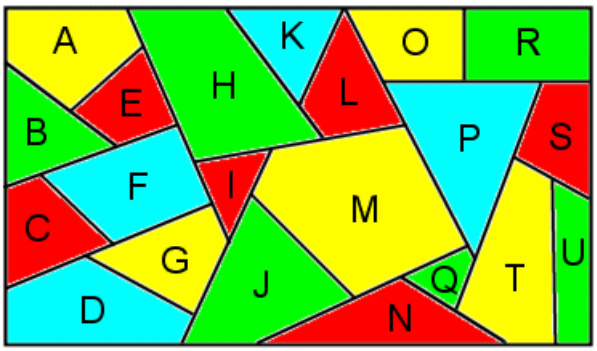
⇨ Redes

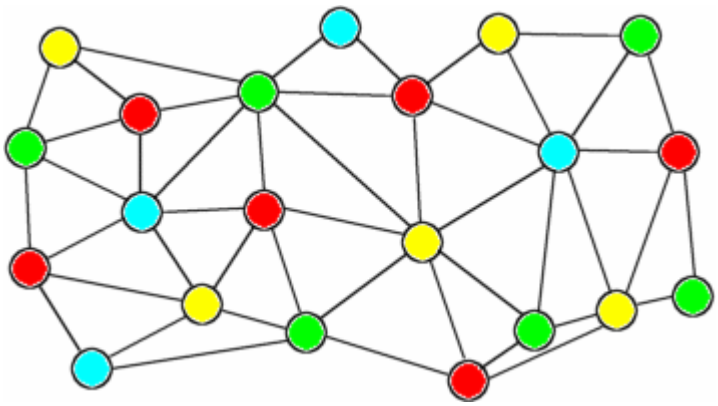




- **Exercício**: Defina o TAD Labirinto por um conjunto adequado de operações. Estabeleça uma representação na forma de Vector de Listas de Adjacências. Adapte os algoritmos anteriores à nova representação.

- **Exercício**: Recorde o Problema da Coloração de Mapas (Cap. 3, pág. 15). As relações de vizinhança entre países são naturalmente representáveis por relações de adjacência entre os vértices de um Grafo planar. Assim, esse problema é um caso particular do Problema da Coloração dos Vértices de um Grafo Planar com Quatro Cores.

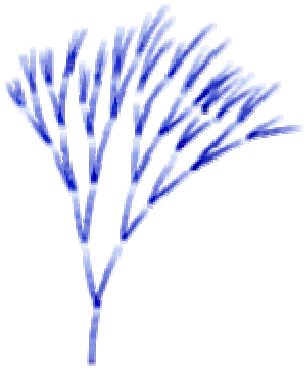




Construa dois procedimentos para colorir um dado mapa, tais que a **ordem** de coloração seja:

no primeiro - em profundidade
no segundo - em largura.

❒ O TAD Árvore Binária



{ No contexto das Estruturas de Dados }

Uma **Árvore Binária** é um conjunto finito de Nós, tal que:

- ou é o conjunto **vazio**,
- ou é constituída por uma **raiz** e duas Árvores Binárias distintas (chamadas **sub-árvore esquerda** e **sub-árvore direita**).

- **Operações**:

criar(T)	{ criar uma Árvore Binária vazia }
vazia(T)	{ verificar se a Árvore Binária é vazia }
construir(R, e, S)	{ construir uma AB, a partir de um elemento e duas outras ABs }
esquerda(T)	{ a sub-árvore esquerda }
direita(T)	{ a sub-árvore direita }
consultar(T)	{ consultar o elemento da raiz }

Operadores e Axiomas:

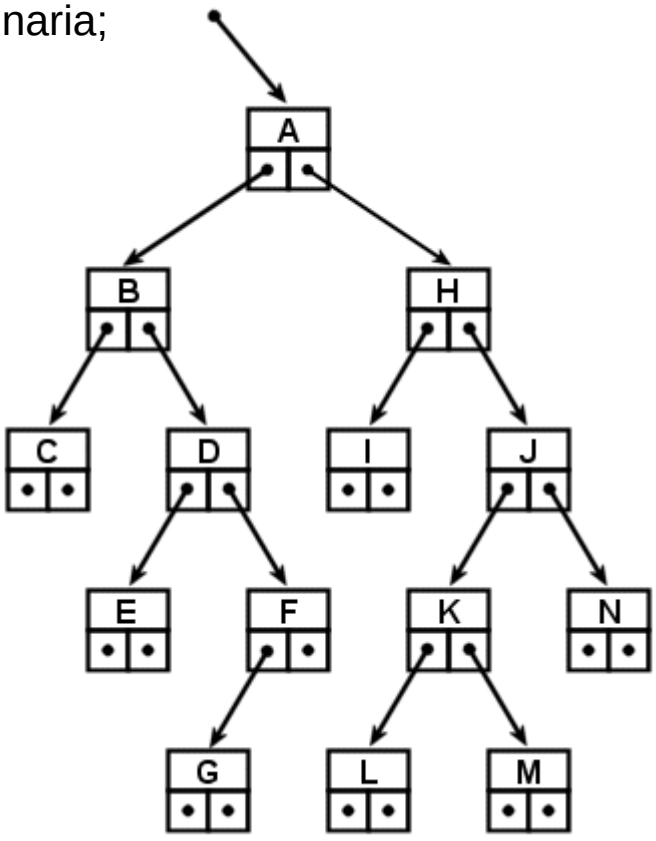
criar :	$\emptyset \rightarrow T$
vazia :	$T \rightarrow$ { verdadeiro, falso }
construir :	$T \times e \times T \rightarrow T$
direita :	$T \rightarrow T$
esquerda :	$T \rightarrow T$
consultar :	$T \rightarrow e$

vazia(criar) = verdadeiro
vazia(construir(R, e, S)) = falso
esquerda(criar) = “erro”
direita(criar) = “erro”
consultar(criar) = “erro”
esquerda(construir(R, e, S)) = R
direita(construir(R, e, S)) = S
consultar(construir(R, e, S)) = e

- **Representação**:

  O TAD Árvore Binária é facilmente representável por uma Estrutura duplamente ligada:

```
type  ArvoreBinaria  = ^no;
                no   = record  elemento : tipo;
                               esq, dir : ArvoreBinaria
                       end;

var raiz : ArvoreBinaria;
```



Exercício: Implemente as seis operações básicas.

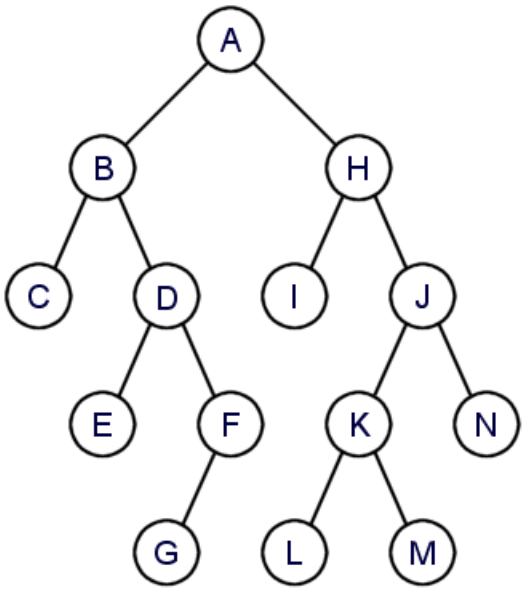
- A definição duplamente recorrente de Árvore Binária conduz, naturalmente, à construção de algoritmos duplamente recorrentes para: cópia, igualdade, travessias, pesquisas, ...
- A combinação da Recorrência Dupla com a Abstracção das Operações permite a elaboração de algoritmos simples e "compactos".

Construir uma Cópia de uma Árvore Binária:

```
R ← Cópia( T ) :
    se vazia(T)
    então   criar(R)
    senão   E ← Cópia( esquerda(T) )
            D ← Cópia( direita(T) )
            elemento ← consultar(T)
            R ← construir(E, elemento, D)
```

```
function copia (t : ArvoreBinaria) : ArvoreBinaria;
    var c, e, d : ArvoreBinaria;

    begin if vazia(t)
          then criar(c)
          else  begin e:= copia(esquerda(t));
                      d:= copia(direita(t));
                      c:= construir(e, consultar(t), d)
                end;
          copia:= c
    end; { copia }
```

- **Travessias de uma Árvore Binária**:



Pré-Ordem: 1º raiz;
2º sub-árvore esquerda;
3º sub-árvore direita.

{ A B C D E F G H I J K L M N}

Em-Ordem: 1º sub-árvore esquerda;
2º raiz;
3º sub-árvore direita.

{ C B E D G F A I H L K M J N}

Pos-Ordem: 1º sub-árvore esquerda;
2º sub-árvore direita;
3º raiz.

{ C E G F D B I L M K N J H A}

- Nas três Travessias Básicas a ordem das **"folhas"** permanece constante. Porque será?

Travessia por Níveis: **{ A B H C D I J E F K N G L M }**

- Na sua forma duplamente recorrente, a implementação das três Travessias Básicas é muito simples. Por exemplo,

```
procedure TravessiaPreOrdem (t : ArvoreBinaria);
          begin if not vazia(t)
                then begin writeln(consultar(t));
                           TravessiaPreOrdem(esquerda(t));
                           TravessiaPreOrdem(direita(t))
                     end
          end; { TravessiaPreOrdem }
```

- Contudo, as correspondentes versões iterativas são bem mais complicadas. A versão iterativa da travessia em Pré-Ordem é obtida pela utilização (quase) directa de uma Pilha, assim como a versão iterativa da travessia Por Níveis é obtida pela utilização de uma Fila auxiliar.

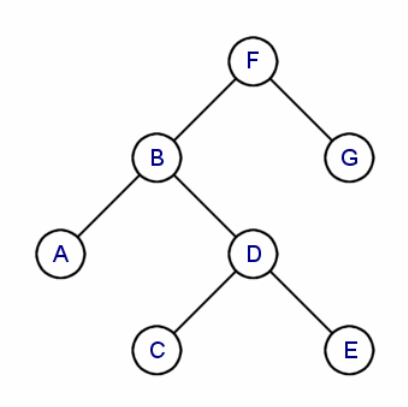
- Analisemos o caso da travessia Em-Ordem:

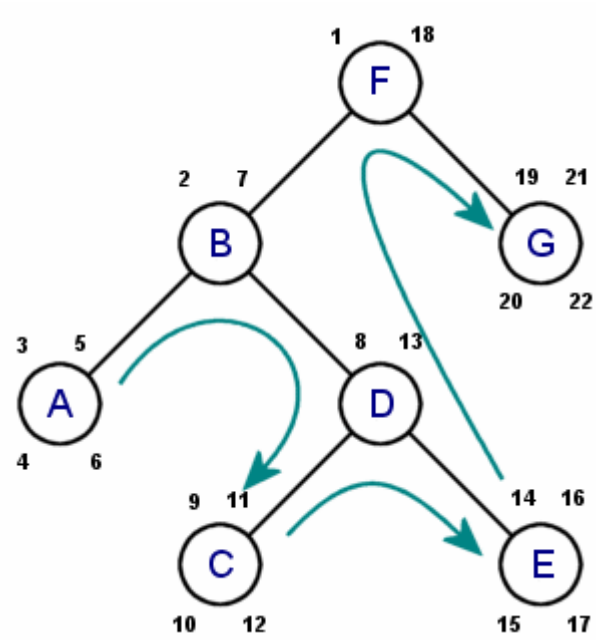
```
procedure TravessiaEmOrdem (t : ArvoreBinaria);
          begin if not vazia(t)
                then begin TravessiaEmOrdem(esquerda(t));
                           writeln(consultar(t));
                           TravessiaEmOrdem(direita(t))
                     end
          end; { TravessiaEmOrdem }
```

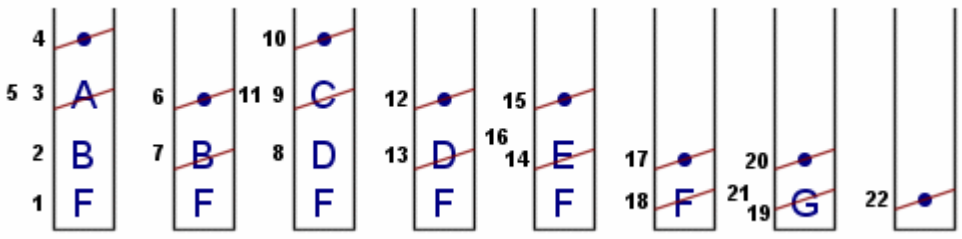
- Uma estratégia para a construção de uma versão iterativa da travessia Em-Ordem consiste na utilização de uma Pilha para guardar (os ponteiros para) as sub-árvores ainda não visitadas.

```
procedure TravessiaEmOrdemIterativa (t : ArvoreBinaria);
        var   p : ArvoreBinaria;
              S : PilhadeArvoresBinarias;
              acabou : boolean;

        begin  criar(S);
               p:= t;
               { começar por guardar a raiz }
               por(p, S);
               acabou:= false;

               while not acabou do

                     if not vazia(p)
                     then  begin  { descer para a esquerda e guardar }
                                  p:= esquerda(p);
                                  por(p, s)
                           end

                     else  begin { subir a partir da sub-árvore vazia }
                                  tirar(S); { remover a sub-árvore vazia }

                                  { consultar a Pilha }
                                  if vazia(S)
                                  then { toda a árvore foi visitada }
                                       acabou:= true

                                  else  begin  { visitar o topo da Pilha }
                                               p:= topo(S);
                                               writeln(consultar(t));
                                               { e remover }
                                               tirar(S);

                                               { descer para a direita }
                                               p:= direita(p);
                                               { e guardar }
                                               por(p, S)
                                        end
                           end

        end; { TravessiaEmOrdemIterativa }
```

Por exemplo,





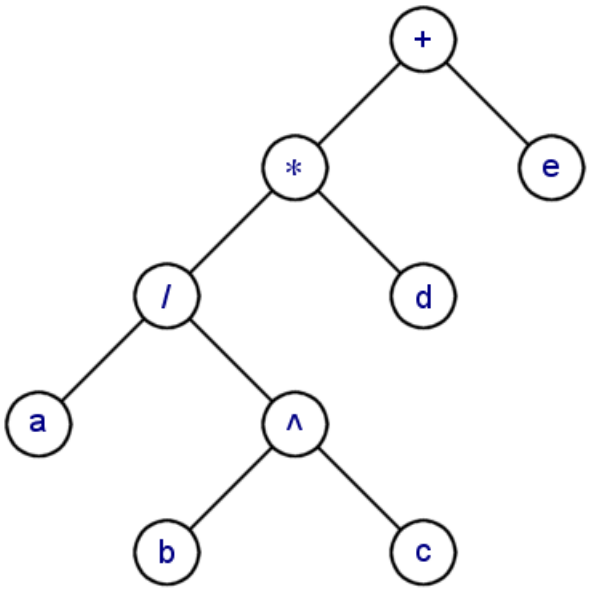


visita: A B C D E F G vazia(S)

- **Exercício**: A estratégia para a construção de uma versão iterativa para a travessia Pós-Ordem também utiliza uma Pilha auxiliar, mas é um pouco mais complicada …

- Uma Aplicação das três Travessias Básicas:

Representação de Expressões Aritméticas.



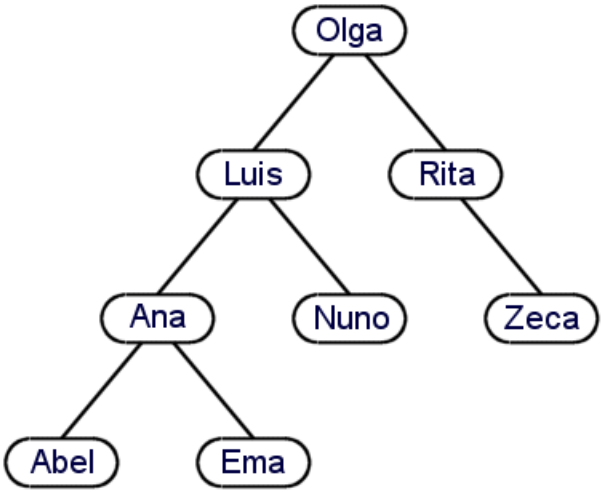
Travessia Pré-Ordem: + * / a ^ b c d e { Notação *prefix* }

Travessia Em-Ordem: a / b ^ c * d + e { Notação *infix* }

Travessia Pos-Ordem: a b c ^ / d * e + { Notação *postfix* }

Uma Aplicação da travessia Em-Ordem:

❒ As Árvores Binárias de Pesquisa

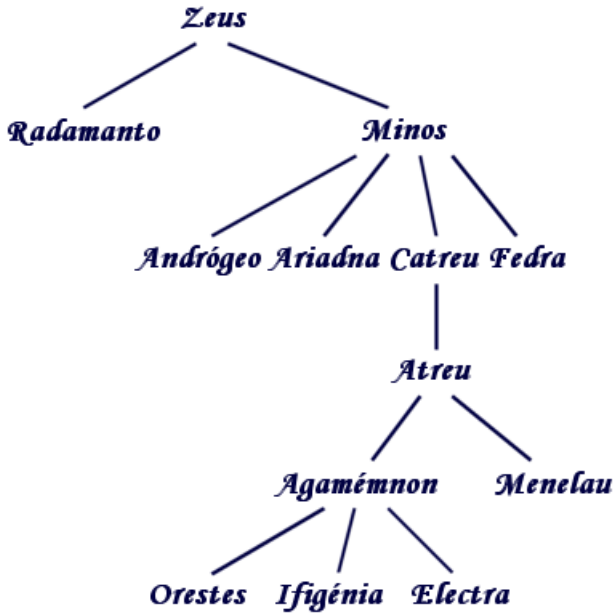


- Como os nomes estão colocados alfabeticamente Em-Ordem, uma Travessia Em-Ordem produz uma listagem ordenada.

- A Pesquisa numa Árvore Binária Em-Ordem é semelhante à Pesquisa Binária e, no caso de uma "árvore equilibrada", tem também complexidade $O(\log_2 n)$.

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

❒ Árvores e Florestas



{ No contexto das Estruturas de Dados }

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 $n \geq 0$ conjuntos distintos $T_1, T_2, \ldots, T_n$.
- Cada conjunto T_i, $i = 1, \ldots, n$ é uma **Árvore**, chamada sub-árvore da raiz.

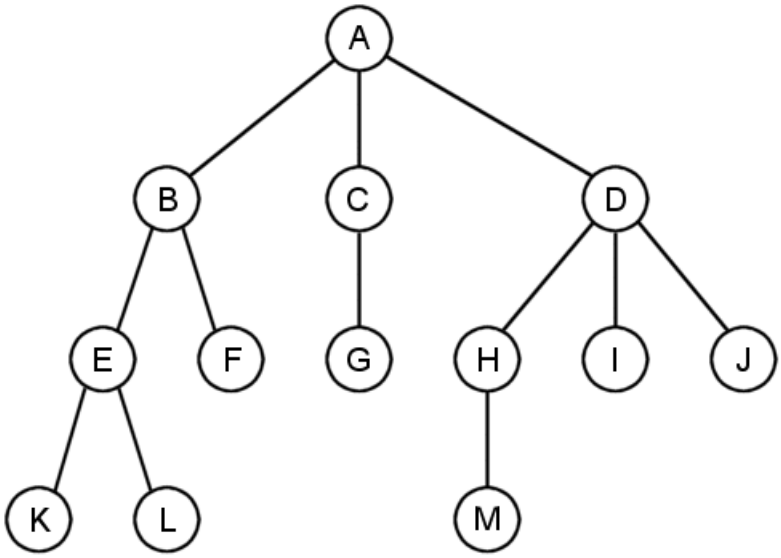
Terminologia :

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

Conceitos :

- Chama-se **Grau** ao número de sub-árvores de um Nó.
- O Grau de uma Árvore é o máximo dos Graus dos seus Nós.
- **Nível**(Nó) = $\begin{cases} 0 & \text{se Nó = Raiz} \\ 1 + \text{Nível(Pai(Nó))} & \text{senão} \end{cases}$
- A **Altura** (ou Profundidade) de uma Árvore é o valor máximo dos Níveis dos seus Nós.

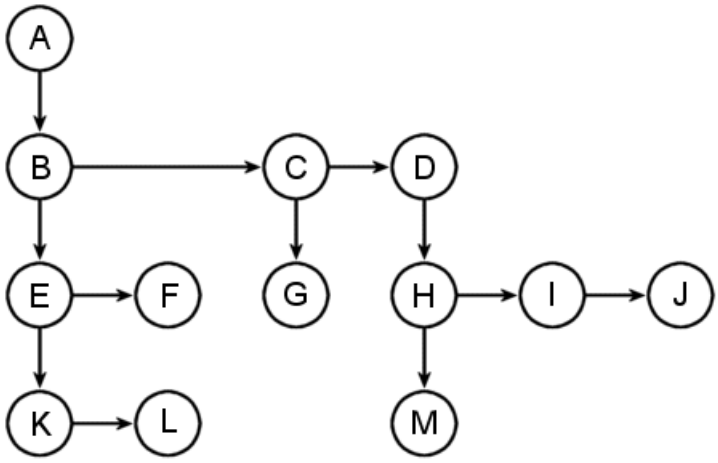
Representação :



- Podemos descrever uma Árvore como uma Lista Generalizada:

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

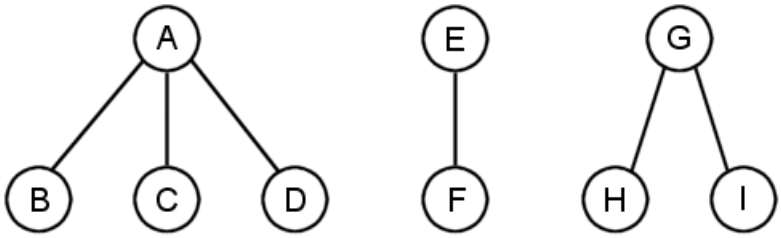
- Por isso, é possível utilizar a Representação Duplamente Ligada:



- Deste modo, toda a Árvore pode ser representada por uma Árvore Binária. É a chamada **Representação Filho-Irmão**.
- Esta Representação é válida também para Florestas.

Uma **Floresta** é um conjunto finito de (zero ou mais) Árvores.

- Prova-se que toda a Floresta é isomorfa a uma Lista Generalizada e, consequentemente, representável por uma Árvore Binária. Note que uma Floresta pode ser vazia (mas uma Árvore não) e, nesse caso, é representada pela Árvore Binária vazia.



Representação :

- Lista Generalizada:

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

- Árvore Binária: