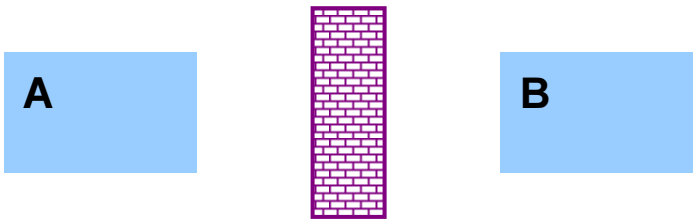
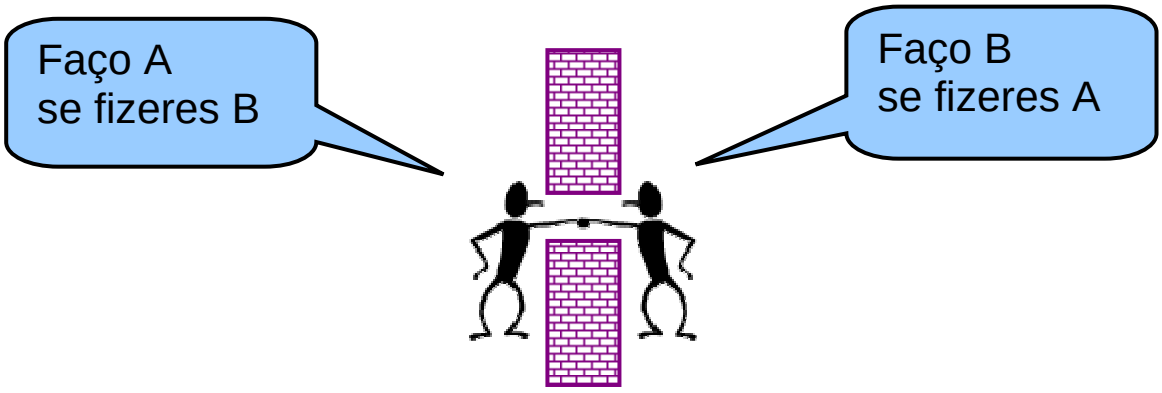
Capítulo 3 : Abstracção dos Tipos de Dados

{ Programação Estruturada }

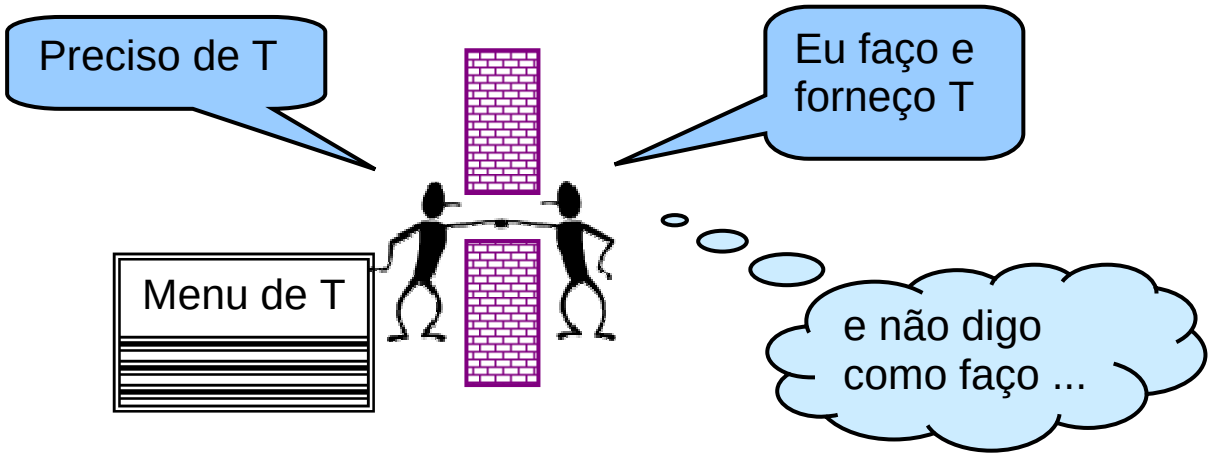
A Modularidade dos Algoritmos:



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 }

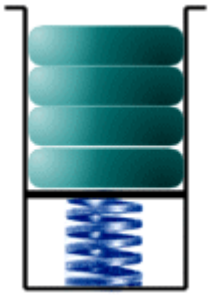
- A definição do TAD Lista Linear Ordenada é **independente** da Estrutura de Dados escolhida para a sua **representação** (ponteiros, cursores, circular, ...)

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:

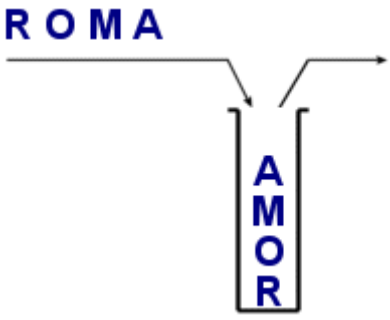
  Exercício: Implemente as cinco operações.

- **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 }

{ Pilha vazia Λ fim de linha ⇒ Parêntesis Equilibrados }

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

➡ Conversão *infix* → *postfix* :

- **Notação Polaca** (Lukasiewicz, 1878-1956)

  Na notação habitual (*infix*) das Expressões Aritméticas, cada operador é colocado **entre** os dois operandos correspondentes.

  a + b ∗ c

  Para saber a ordem por que são efectuados os cálculos, é necessário estabelecer um conjunto de **Regras de Prioridade** entre os operadores e (para alterar essa ordem) é também necessário o uso de **Parêntesis**.

- Na **Notação Polaca** (*postfix*), cada operador é colocado **depois** dos dois operandos correspondentes.

  A ordem dos cálculos a efectuar fica univocamente definida, sem Regras de Prioridade nem Parêntesis.

Exemplos:

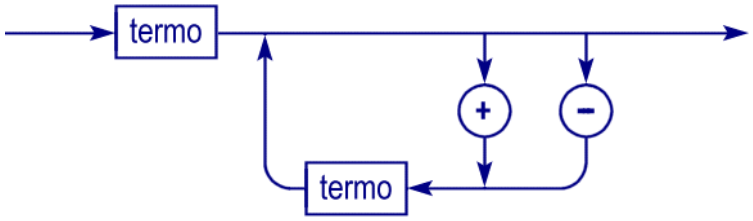
infix	*postfix*
a + b	a b +
(a + b) ∗ c	a b + c ∗
(a + b) ∗ (c - d)	a b + c d - ∗
a + b ∗ c - d	a b c ∗ + d -
a + b ∗ (c - d)	a b c d - ∗ +
(a + b) ∗ c - d	a b + c ∗ d -

Problema: Conversão *infix* → *postfix*

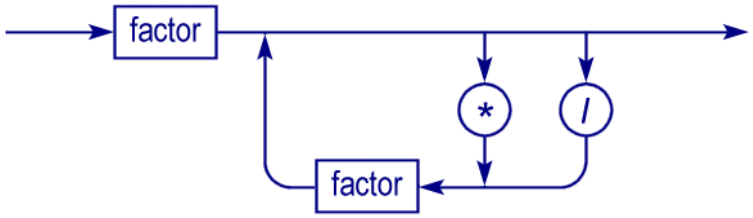
Vamos considerar "uma" Linguagem de Alto Nível, que consiste numa simplificação do Pascal.

Definições:

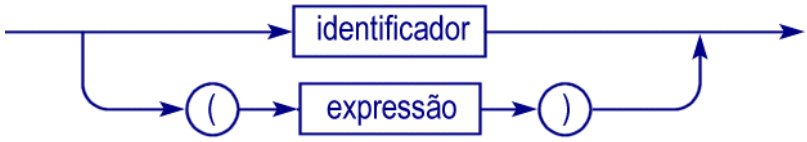
Expressão:



Termo:



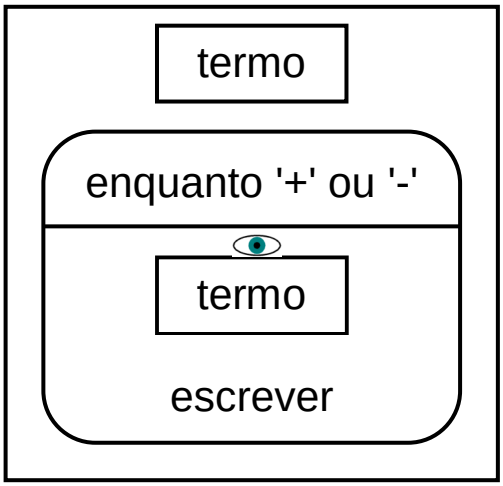
Factor:



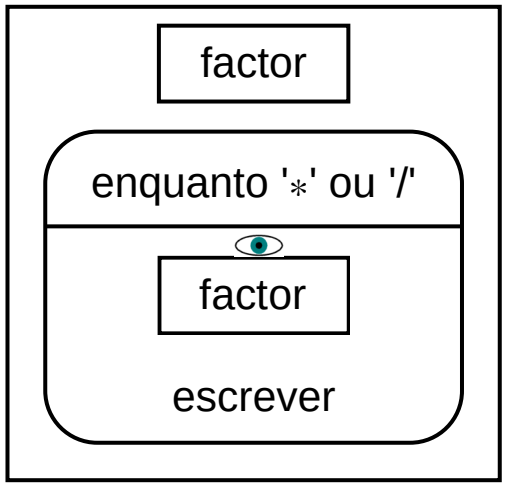
Identificador:

- Para resolver o problema, basta construir um Módulo (recorrente) para cada uma das definições (recorrentes):

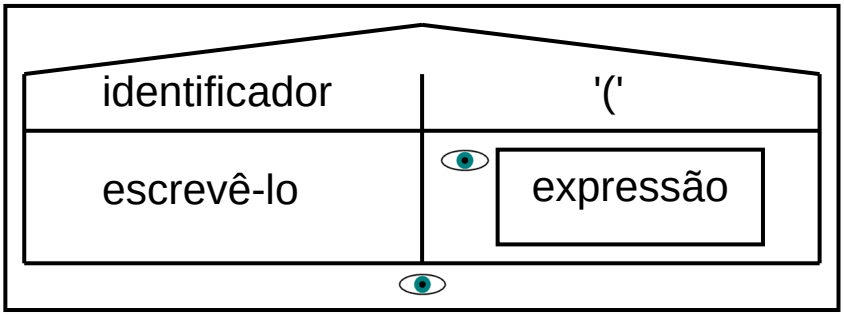
expressão



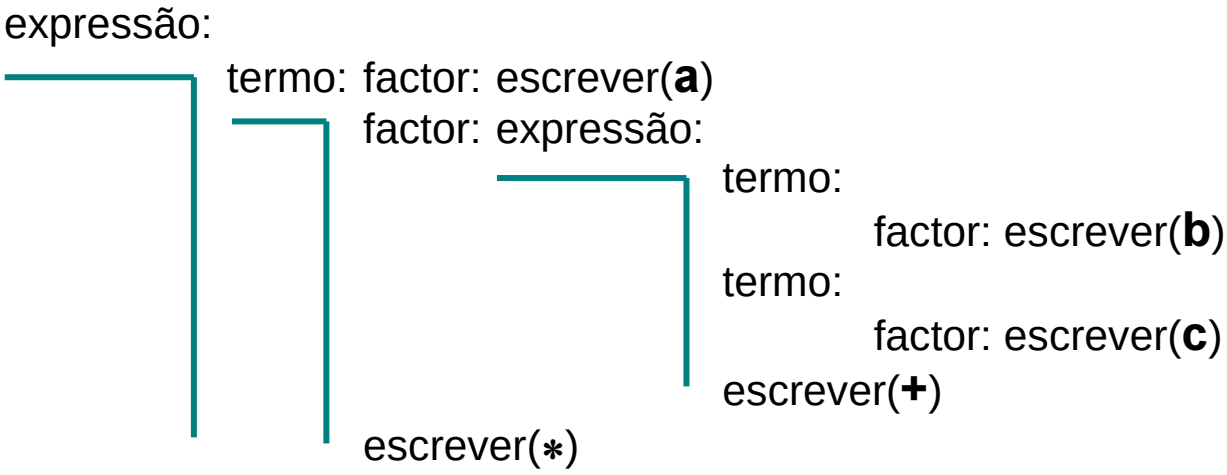
termo



factor



Exemplo: $a * (b + c) \rightarrow a\ b\ c\ +\ *$



Versão Recorrente em Pascal:

```
program InfixPostfix(input, output);
var car : char;

procedure   procurar;
            begin   repeat read(car);
                    until (car <> ' ') or eoln(input)
            end (* procurar *);
```

```
procedure  expressao;
           var operaditivo : char;

           procedure  termo;
                      var opermult : char;

                      procedure   factor;
                                  begin   (* factor *)
                                          if car = '('
                                          then   begin procurar;
                                                       expressao
                                                 end
                                          else write(car);
                                          procurar
                                  end (* factor *);

                      begin   (* termo *)
                              factor;
                              while  (car = '*') or (car = '/') do
                                     begin   opermult:= car;
                                             procurar;
                                             factor;
                                             write(opermult)
                                     end
                      end (* termo *);

           begin   (* expressao *)
                   termo;
                   while  (car = '+') or (car = '-') do
                          begin   operaditivo:= car;
                                  procurar;
                                  termo;
                                  write(operaditivo)
                          end
           end (* expressao *);

begin (* Programa Principal *)
      procurar;
      repeat   expressao;
               writeln
      until car = '.'

end (*InfixPostfix *).
```

Alguns resultados:

input	output
(a+b)∗(c -d)	ab+cd-∗
a+b∗c-d	abc∗+d-
(a +b)∗ c-d	ab+c∗d-
a+b∗(c-d)	abcd-∗+
a ∗a ∗a∗a	aa∗a∗a∗
b+c ∗(d+ca∗a) ∗b + a	bcdca∗a∗+∗b∗+a+
(a∗ b) / c	ab∗c/
a∗b/c	ab∗c/
a∗(b/c)	abc/∗
a∗a/a∗a/a∗a .	aa∗a/a∗a/a∗

Comentários:

- A Notação Polaca é utilizada na Compilação das Expressões Aritméticas de todas as Linguagens de Alto Nível.
- O código produzido pelos módulos recorrentes não é eficiente, não sendo portanto adequado à construção de Compiladores.
- Contudo, a partir da abordagem recorrente, é possível construir a correspondente versão iterativa.

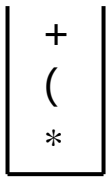
Exercícios:

- Calcular o valor de uma expressão *postfix*.
- Conversão *postfix* → *infix*.

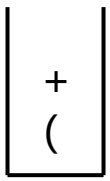
➡ Versão iterativa da conversão *infix* → *postfix* :

{ 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)

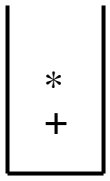
Percorrendo a expressão da esquerda para a direita:

- 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’)

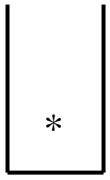


topo (S) = ’∗’ ; escrever (’∗’) ; tirar (S)

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};
```

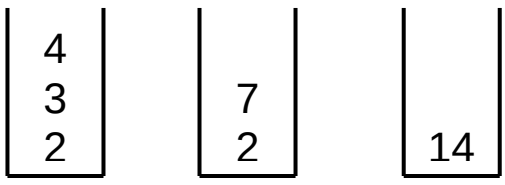
```
        { Só falta acabar de copiar os operadores que ficaram na Pilha }

        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",

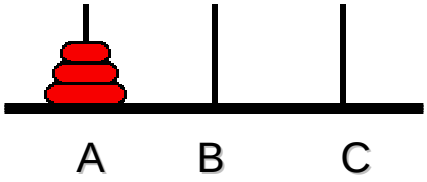
```
type movimento = record  n : natural;
                         origem, auxiliar, destino : torre
                 end;
var   S : PilhadeMovimentos;
```

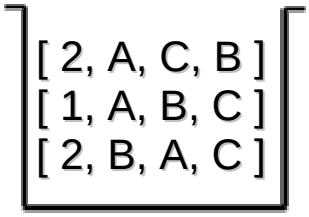
- e adaptemos as operações básicas, de modo a facilitar os acessos aos quatro campos do registo.

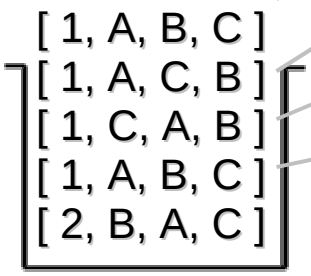
```
...
criar(S);
write('Quantos discos?'); readln(n);
por(n, 'A', 'B', 'C', S);
repeat  topo(k, orig, aux, dest, S);
        tirar(S);
        if k = 1
        then writeln(orig, ' --> ', dest)
        else begin  por(k-1, aux, orig, dest, S);
                    por(  1,  orig, aux, dest, S);
    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

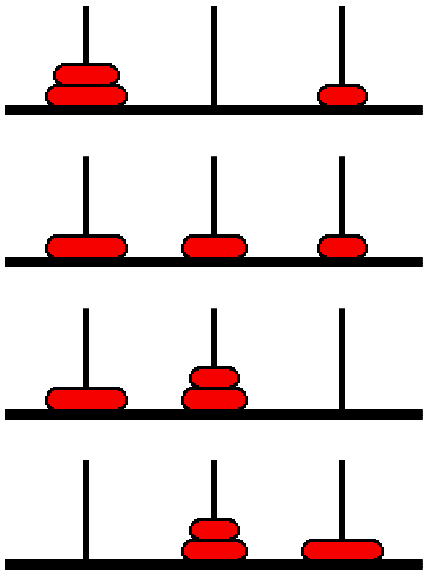
- Note que os "movimentos" são colocados na Pilha, por ordem inversa à das chamadas do Algoritmo Recorrente.

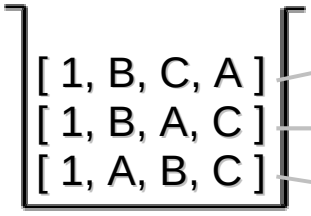
- Simulação para (3, A, B, C)

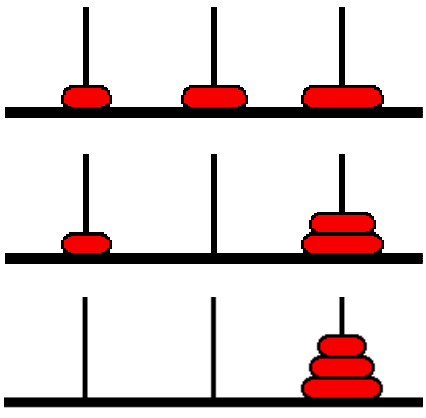




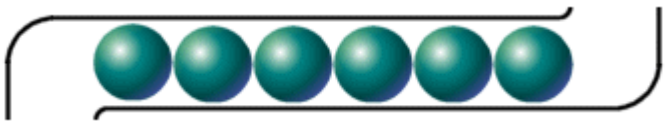








O TAD Fila (*Queue*)



{ 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 :	$Q \rightarrow e$

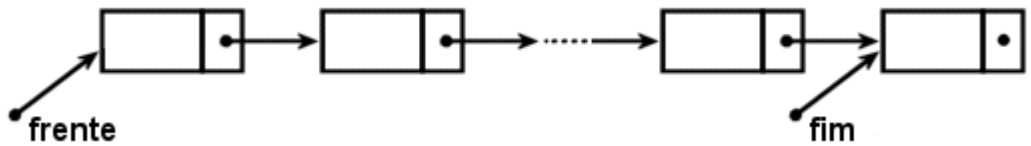
vazia(criar) = verdadeiro
vazia(juntar(e, Q)) = falso
remover(criar) = “erro”
frente(criar) = “erro”
remover(juntar(e, Q)) { se vazia(Q)
então = criar(Q)
senão = juntar(e, remover(Q)) }
frente(juntar(e, Q)) { se vazia(Q)
então = e
senão = frente(Q) }

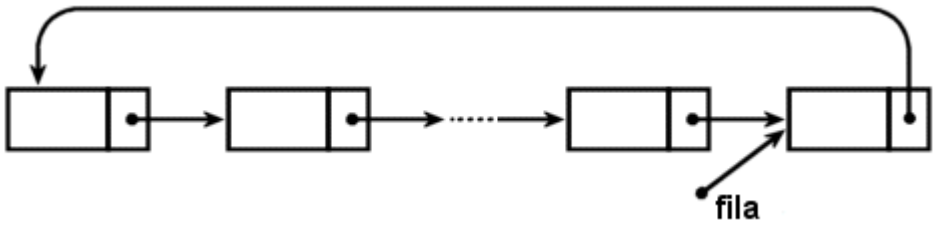
- Os dois conjuntos de Axiomas caracterizam univocamente o funcionamento dos TADs Pilha e Fila. Note-se que a "assinatura" dos dois conjuntos de Operadores é idêntica.

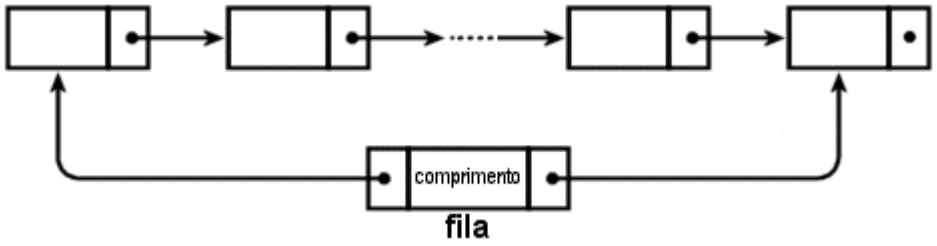
- **Representações**:

  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**:

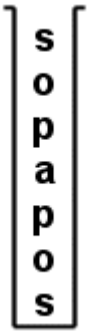
  O TAD Fila tem uma vasta gama de aplicações práticas: Filas de Espera e Simulação de Processos (redes de comunicação, tráfego, ...), Gestão de Recursos em Sistemas Operativos, ...

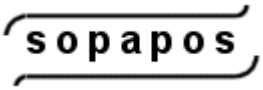
➡ Verificar se uma dada palavra é uma capicua (palíndromo)

{ 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":

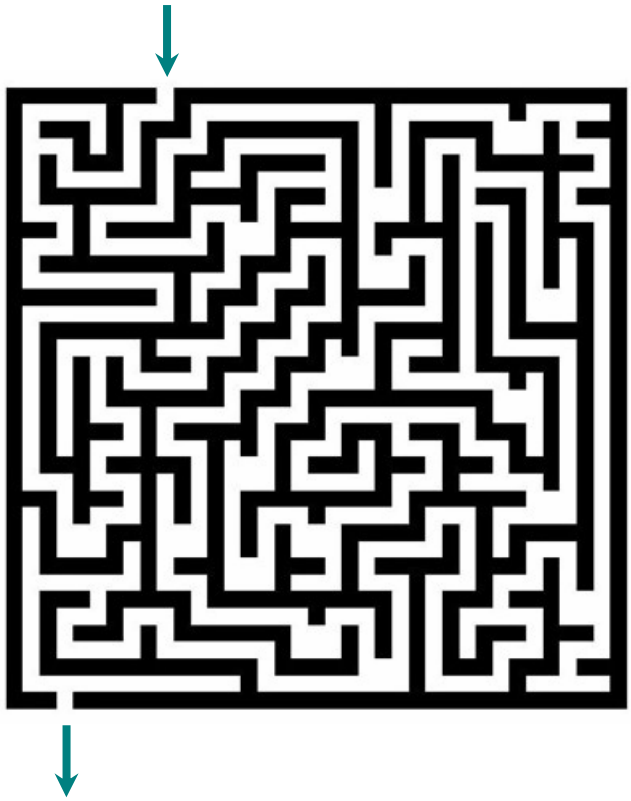
```
var  c : char;
     S : PilhadeCaracteres;
     Q : FiladeCaracteres;

     ...
     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

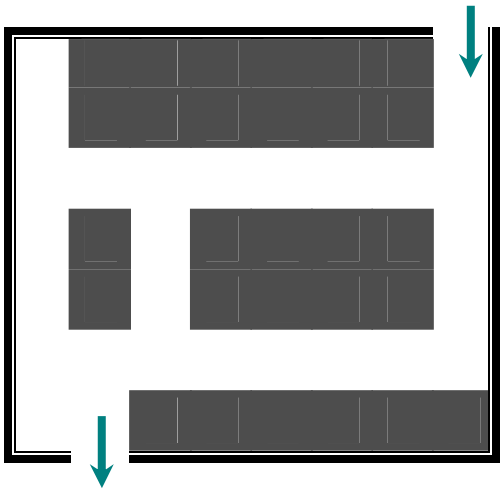
❑ Problema: Resolução de Labirintos



- 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, …

Uma Representação simplificada de um Labirinto

O Labirinto será representado por uma matriz de booleanos.



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

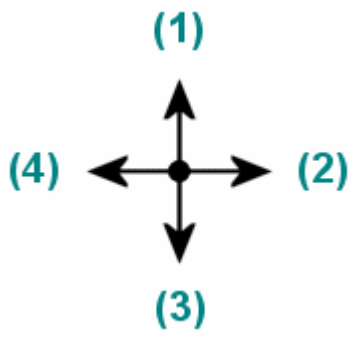
As casas já visitadas serão registadas em:

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

A partir de cada casa [i, j], os quatro movimentos possíveis serão calculados, por soma com os elementos da matriz:

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

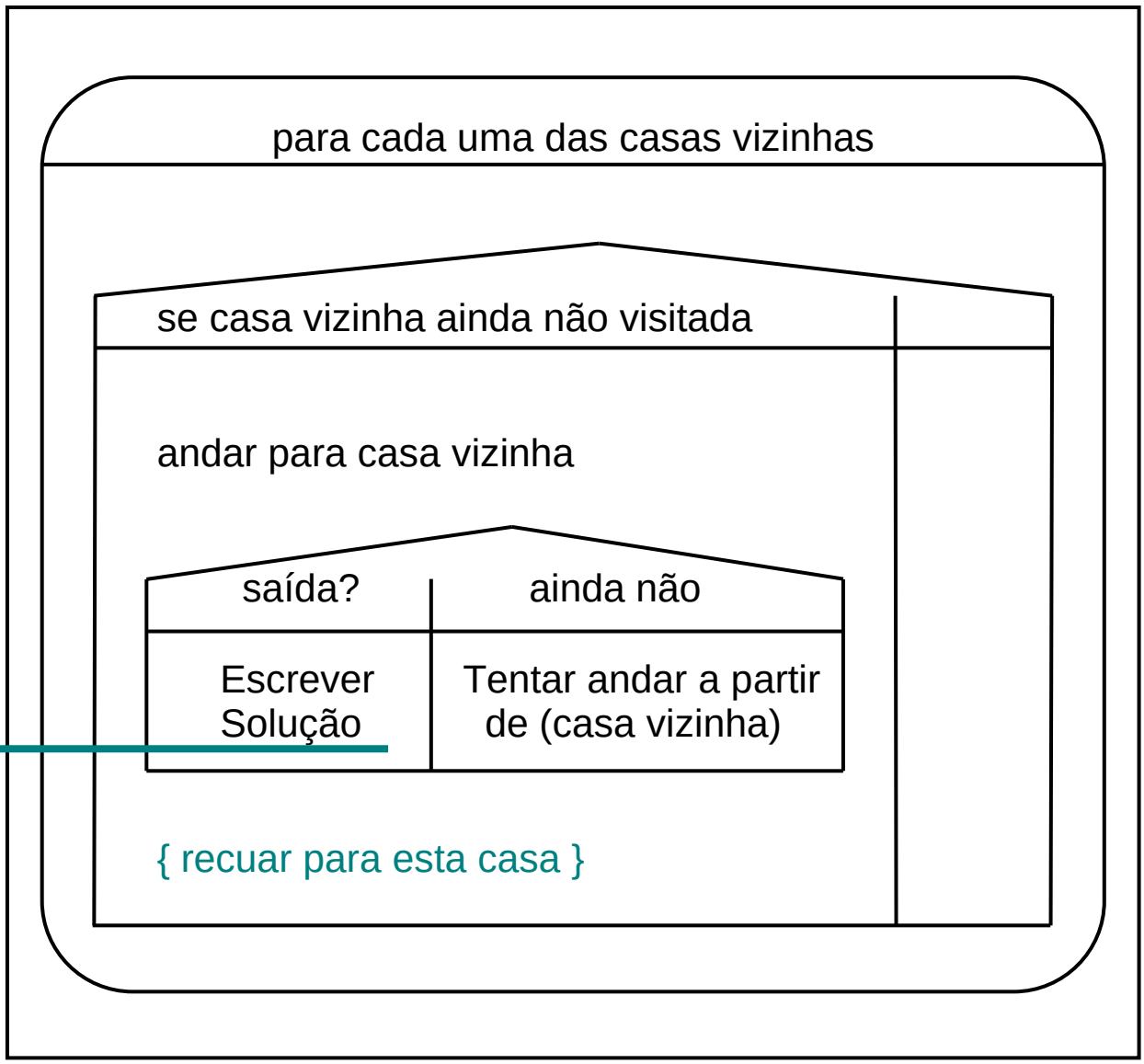
	1	2
1	-1	0
2	0	1
3	1	0
4	0	-1



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

➡ Solução Recorrente – Algoritmo de *Backtracking*

Tentar andar a partir de (esta casa):



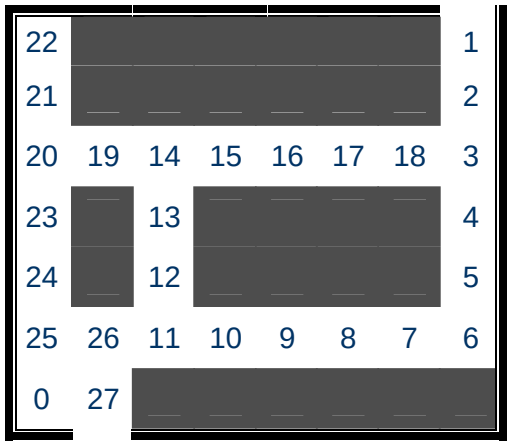
{ **goto** para interromper o processo recorrente,
logo que encontrada a **primeira solução**.

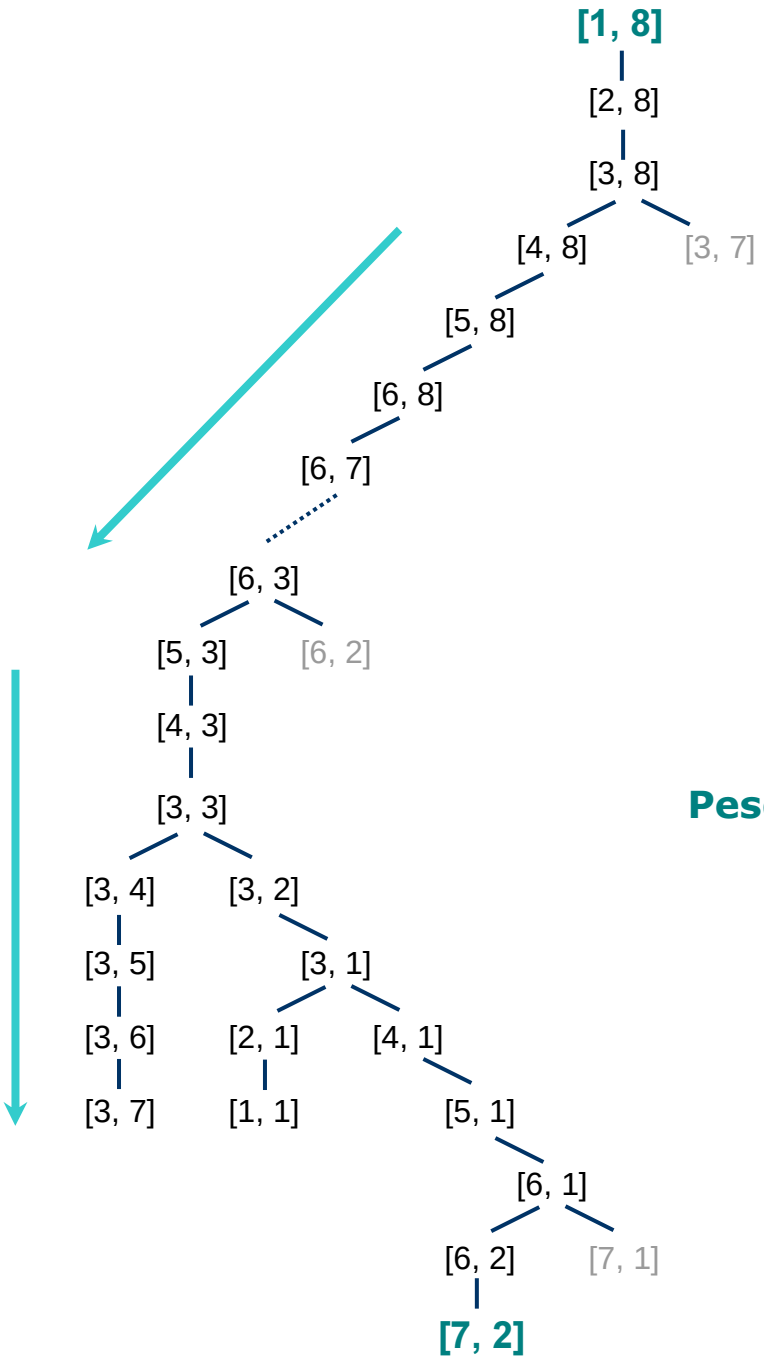
Caso contrário seriam geradas todas as soluções. }

```
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.
```

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.

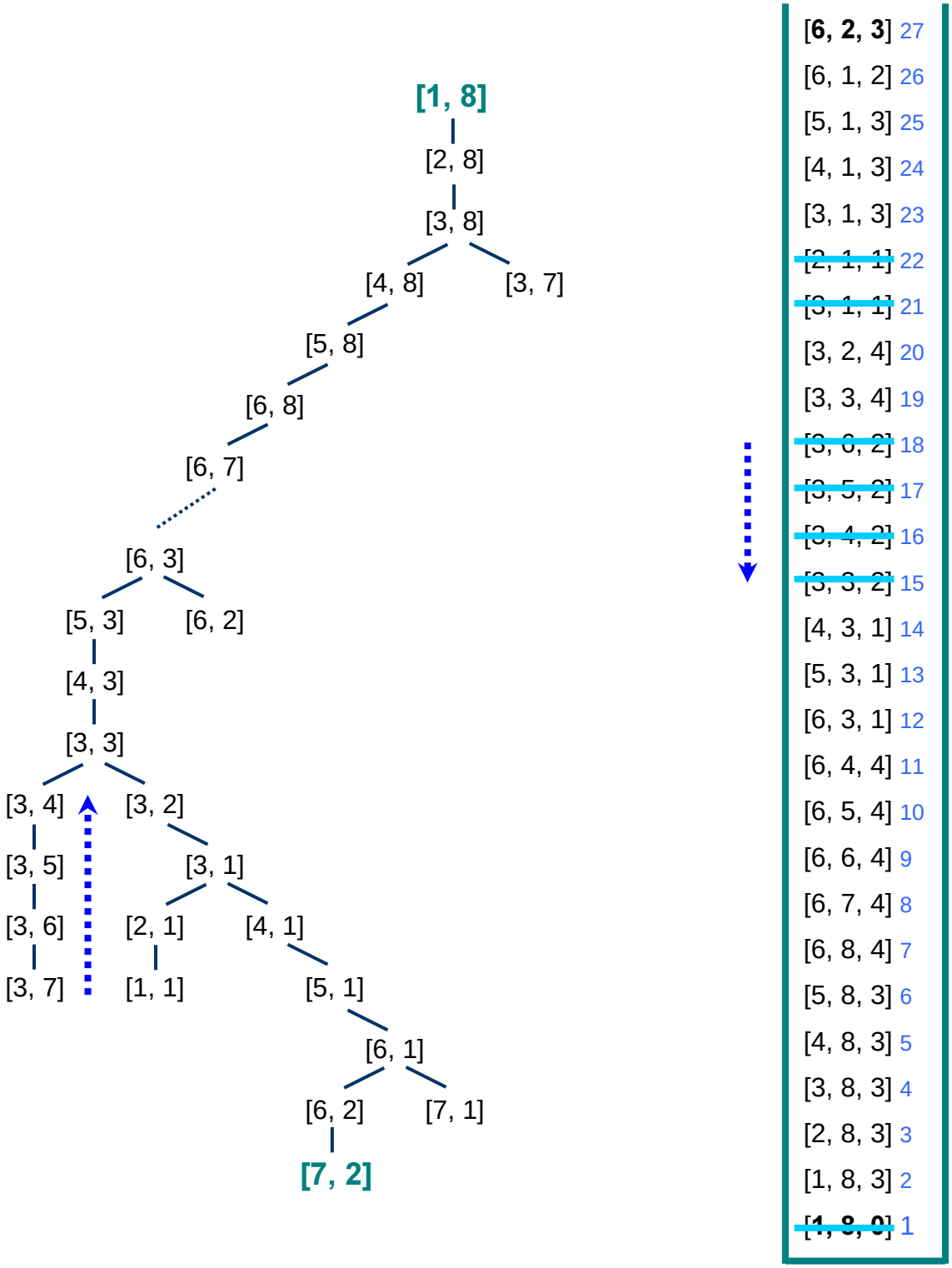
{ Como alternativa, poderíamos utilizar uma Pilha com elementos do tipo [linha, coluna]*. Nesse caso, os elementos da matriz* visitadas *teriam de ser inteiros. }*

```
…

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;
```

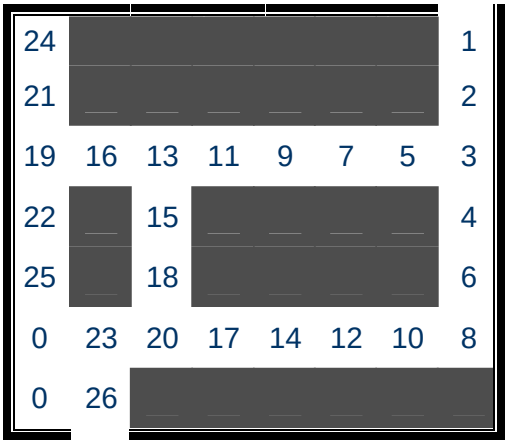
onde o resultado do percurso fica registado, pelos sucessivos valores da variável, **var** ordem : integer;

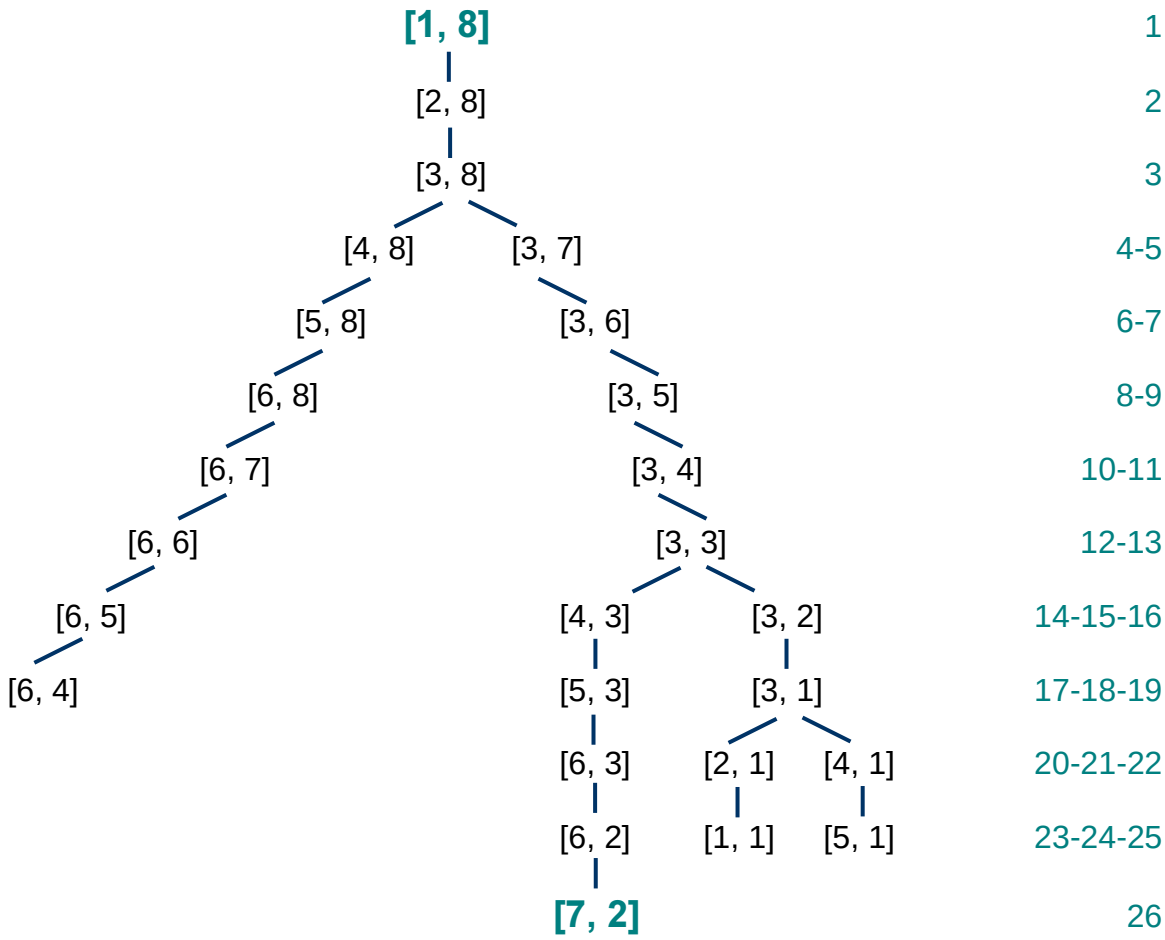
```
...

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 ');
```

{ Gestão da Fila }

Análise da Solução – Árvore de Pesquisa

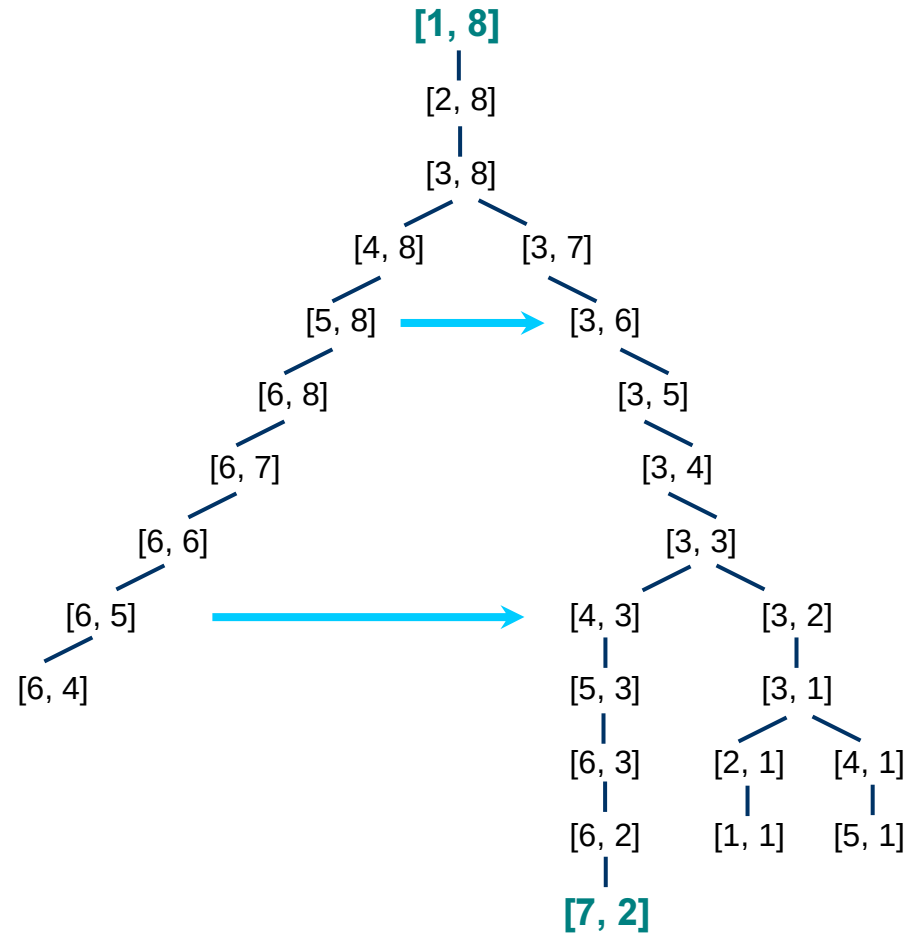
{ A solução obtém-se de modo análogo ao anterior. }





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.

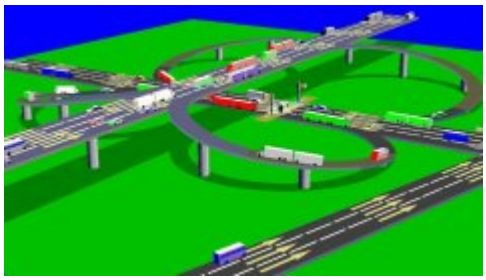


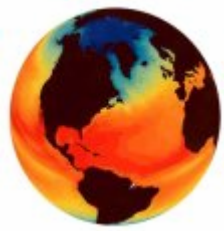
[1, 8] [2, 8] [3, 8] [4, 8] [3, 7] [5, 8] [3, 6] [6, 8] [3, 5] [6, 7] [3, 4] [6, 6] [3, 3] [6, 5]

[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.

❒ **Simulação:** Processo de elaboração de um **modelo** abstracto a partir de uma situação real, a fim de verificar o resultado de modificações introduzidas e o efeito de diferentes estratégias.





Fases do processo de simulação:

- Formulação do problema
- Construção do modelo
- Elaboração de um programa para simulação do modelo
- Análise dos resultados
- Validação do modelo
- Estudo do comportamento perante modificações

Tipos de simulação:

- Estática e Dinâmica
- Determinista e Estocástica
- Contínua e Discreta

Simulação Discreta:

- Controlada pelo **tempo** ***(time driven)***
- Controlada pelos **eventos** ***(event driven)***

➡ Um exemplo de Simulação Estocástica

Cálculo do valor de π

- O processo consiste no lançamento aleatório de dardos num quadrado unitário, no qual foi inscrito um quarto de círculo.
- Assumindo uma distribuição uniforme dos lançamentos no quadrado, a proporção de dados que atingem o quarto de círculo deverá ser igual a $\pi/4$.
- Podemos facilmente simular este processo utilizando um gerador de números aleatórios para os valores das coordenadas (x, y) do resultado de cada lançamento.

```
function  MonteCarloPi ( n : integer ) : real;
          { para n lançamentos }
          var   i, in : integer;
                x, y : real;
          begin  randomize;
                 { inicializa a geração de números aleatórios }
                 in := 0;
                 for  i := 1 to n do
                      begin  x := random(1);
                             y := random(1);
                             { números aleatórios reais ∈ [0, 1] }

                             if  sqr(x) + sqr(y) < 1
                             then in := in + 1
                      end;

                 MonteCarloPi := 4 * in / n
          end;
```

exercício: Utilize este método para calcular o valor de um integral.

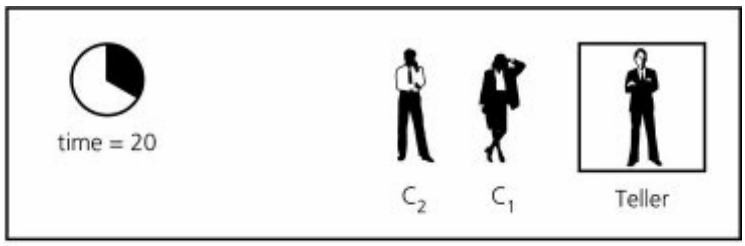
A utilização de Filas em Simulação

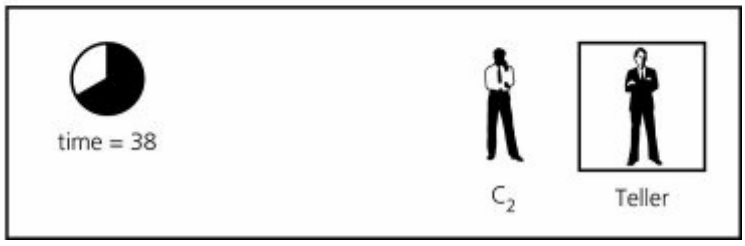
F.M. Carrano, P. Helman and R. Veroff,
Data Structures and Problem Solving with Turbo Pascal: WALLS AND MIRRORS 1993

{ ftp://ftp.aw.com/cseng/authors/carrano/tpas/ }

- Consideremos o caso de um banco, com uma só caixa. Pretende-se averiguar o tempo de espera dos clientes.
- Num ficheiro de texto, existe informação de estatísticas efectuadas sobre tempos de chegada e de atendimento de clientes:

chegada	atendimento
12	26
20	5
22	4
23	2
30	3
...	





Eventos, Filas e Listas

- **Evento Chegada**: Indica a chegada de um cliente, que é atendido ou espera na fila.
  Evento **externo**, especificado pelo ficheiro de dados.

- **Evento Partida**: Indica o fim do atendimento de um cliente (e o início do atendimento do próximo).
  Evento **interno**, determinado pelo processo de simulação.

- **Fila de Espera**: dos clientes no banco.

- **Lista de Eventos**: com informação sobre todos os futuros eventos, ordenados pelo tempo.

Simulação controlada pelo tempo (*time driven*)

- Em cada passo da simulação, o **tempo** avança de uma unidade e são processados os eventos chegada e partida.

```
tempo ← 0
criar fila de espera vazia

enquanto  tempo <= fim

          se ocorrer uma chegada
          então processar evento chegada;

          se ocorrer uma partida
          então processar evento partida;

          tempo ← tempo + 1
```

Simulação controlada pelos eventos (*event driven*)

- Em cada passo da simulação, o **tempo** avança directamente para o próximo evento.

criar **fila de espera** vazia

enquanto existirem **eventos na lista**

{ tempo ← tempo do próximo evento }

se o evento for uma chegada
então **processar evento chegada**
setão **processar evento partida**

- Onde:

processar evento chegada:

remover **evento chegada** da lista;

se a **fila de espera** estiver vazia
então calcular e inserir **evento partida** na lista;

se não for o fim do ficheiro de dados
então **ler** outro **evento chegada** e inserir na lista;

processar evento partida:

remover cliente na frente da **fila de espera**;

remover **evento partida** da lista;

se a **fila de espera** não estiver vazia
então próximo cliente: inserir **evento partida** na lista