

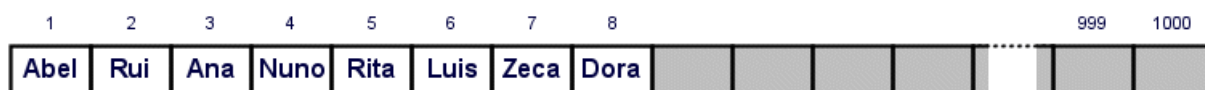
Capítulo 4 : Estruturas de Dados

❑ A utilização de Estruturas Dinâmicas de Dados

Consideremos uma “lista de nomes”, sobre a qual iremos precisar de efectuar diversas operações: **Como vamos representá-la?**

1ª Versão: com um Vector,

```
var lista : array [1 .. 1000] of nome;
```



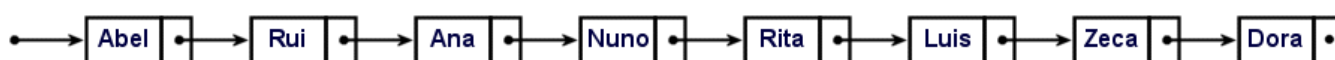
Vantagens:

- Acesso aleatório (directo);
- Se o Vector estiver ordenado, permite pesquisas em $O(\log_2 n)$;

Desvantagens:

- Comprimento fixo (necessidade de declarar a dimensão máxima + desperdício do espaço não utilizado + perigo de transbordo);
- Ineficiência na inserção/remoção de elementos (deslocamento de todos os elementos seguintes); os ordenamentos também requerem deslocamentos;

2ª Versão: com uma Lista Ligada,

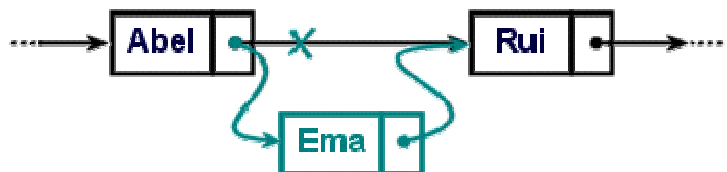
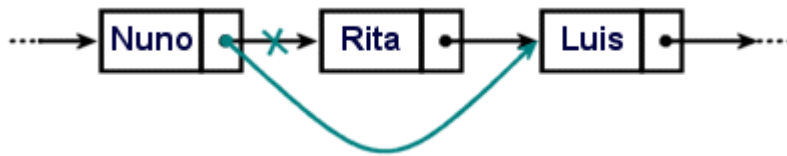


Desvantagens:

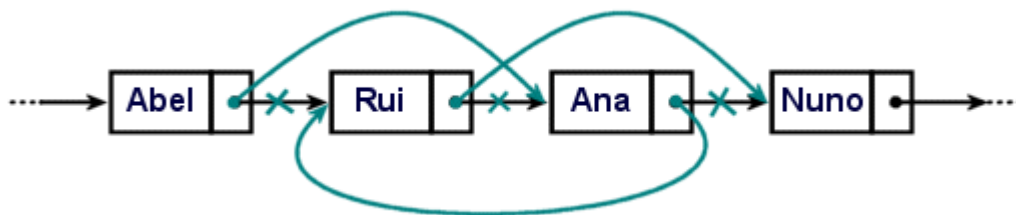
- Acesso sequencial;
- Pesquisas, só sequenciais, em $O(n)$;

Vantagens:

- Comprimento variável;
- As Remoções e Inserções não afectam os restantes elementos;



- As Trocas não alteram a posição (na Memória) de nenhum dos elementos;

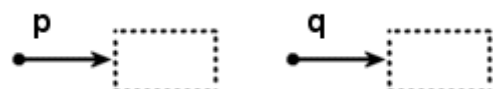


□ Ponteiros

- Um Ponteiro contém uma Referência para uma zona da Memória, onde vai ser registado um Elemento (simples ou estruturado).

```
type nome = array[1..100] of char;
seta = ^nome;
```

```
var p, q : seta;
```

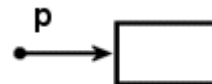


- Contrariamente à declaração de Variáveis Estáticas, a Declaração de uma Variável do Tipo Ponteiro **não** reserva o espaço de Memória onde irão ser registados os Elementos;

{ Assim, a declaração **var p, q : seta;**
cria os Ponteiros p e q, mas não reserva espaço para os nomes }

- Para isso, é necessário utilizar o Procedimento pré-definido:

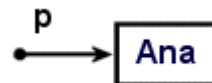
new(p);



Agora, já foi reservado espaço de Memória para registar um nome, que podemos **aceder** através de p.

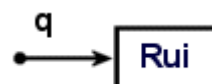
{ p[^] é a zona de Memória apontada por p }

p[^] := 'Ana';



- { Se repetirmos new(p); será criado **outro** espaço, referido pelo ponteiro p. Assim, o espaço de Memória que contém 'Ana', permanece ocupado, mas fica completamente inacessível. }

new(q);
q[^] := 'Rui';

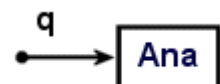
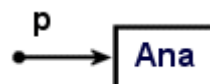


writeln(p[^], ' e ', q[^]);

[Ana e Rui]

- Atribuição entre Elementos** referidos por Ponteiros:

q[^] := p[^];

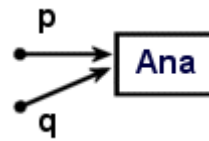


writeln(p[^], ' e ', q[^]);

[Ana e Ana]

- **Atribuição entre Ponteiros:**

$q := p;$



$\text{writeln}(p^{\wedge}, 'e', q^{\wedge});$

[Ana e Ana]

{ O espaço anteriormente referido por q permanece ocupado, mas torna-se inacessível }

- Para **libertar** o espaço de Memória referido por um Ponteiro:

$\text{dispose}(p);$

{ O valor do Ponteiro torna-se indefinido }

- O Ponteiro **Nulo** é aquele que não aponta para nada:

nil



{ Podemos atribuir $p := \text{nil};$

Contudo, uma sequência de instruções do tipo:

```

new(p);
p^:= umnome;
p := nil;
  
```

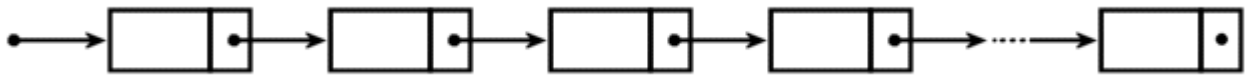
não faz sentido. O espaço $p^{\wedge}$, referido por p , não é libertado, ficando a ocupar um espaço inacessível de Memória }

O Ponteiro **nil** é muito útil, em particular para assinalar o fim de uma Lista Ligada.

while $p \neq \text{nil}$ **do**

...

□ A Lista Ligada (Linear)

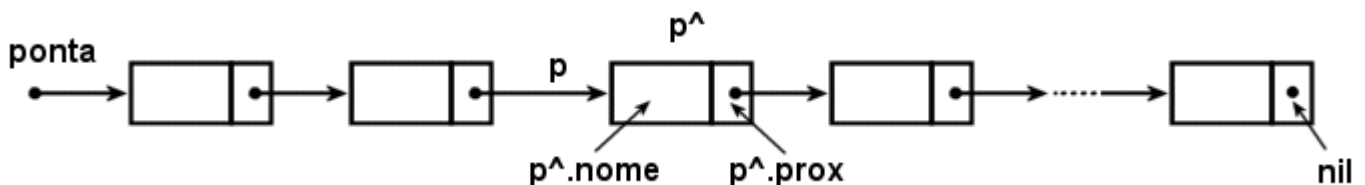


• Declaração dos elementos :

```

type ponteiro = ^caixa;
      caixa = record nome : tipo;
                prox : ponteiro
      end;
{ Notar a natureza recorrente desta declaração }
var p, este, ponta : ponteiro;

```



Operações Elementares sobre uma Lista Ligada:

• Travessias:

{ Escrever os nomes contidos nos Nós da Lista, a partir da Ponta }

```

procedure listar (ponta : ponteiro);
  var p : ponteiro;

  begin p:= ponta;
    while p <> nil do
      begin { Visitar este Nó }
        writeln(p^.nome);
        { Avançar para o Nó seguinte }
        p:= p^.prox
      end
  end; { listar }

```

```
{ Determinar o Comprimento de uma Lista }  
function comprimento (ponta : ponteiro) : integer;  
    var p : ponteiro;  
        c : integer;  
    begin p:= ponta;  
        c:= 0;  
        while p <> nil do  
            begin { Contar mais este Nó }  
                c:= c +1;  
                { Avançar para o Nó seguinte }  
                p:= p^.prox  
            end;  
        comprimento:= c  
    end; { comprimento }
```

{ Pela sua natureza recorrente, as Listas Ligadas são particularmente adequadas à utilização de Algoritmos Recorrentes }

```
procedure listar (ponta : ponteiro);  
    begin if ponta <> nil  
        then begin { Visitar o primeiro Nó }  
                writeln(ponta^.nome);  
                { Listar os Nós seguintes }  
                listar(ponta^.prox)  
            end  
    end; { listar }
```

E qual será o resultado da seguinte Travessia?

```
procedure listarR (ponta : ponteiro);  
    begin if ponta <> nil  
        then begin listarR(ponta^.prox);  
                writeln(ponta^.nome)  
            end  
    end; { listarR }
```

{ Determinar o Comprimento de uma Lista: versão Recorrente }

```
function comprimento (ponta : ponteiro) : integer;
  begin if ponta = nil
    then comprimento := 0
    else comprimento := comprimento(ponta^.prox) + 1
  end; { comprimento }
```

{ Verificar se duas Listas são iguais: versão Recorrente }

{ Abordagem recorrente:

$$\text{iguais}(p, q) \Leftrightarrow (p.^{\text{nome}} = q.^{\text{nome}}) \wedge \text{iguais}(p.^{\text{prox}}, q.^{\text{prox}})$$

contudo, é necessário verificar os casos de p ou q serem nulos, (onde os campos nome e prox não existiriam) e, para maior eficiência, convém utilizar uma variável lógica auxiliar }

```
function iguais (p, q : ponteiro) : boolean;
  var ig : boolean;

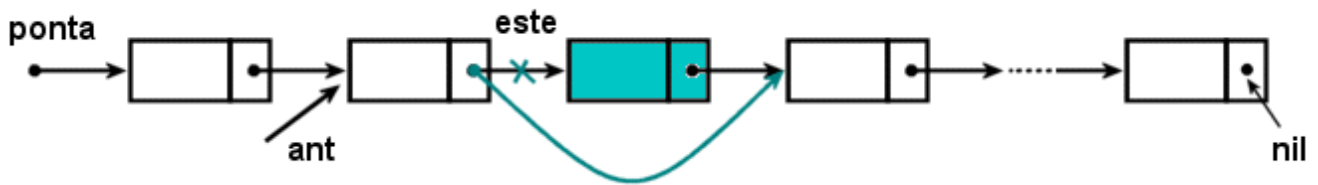
  begin ig := (p = nil) and (q = nil);      ←
    {  $\neg \text{ig} = (p \neq \text{nil}) \vee (q \neq \text{nil})$  }

    if (p <> nil) and (q <> nil)
    then begin ig := p.^nome = q.^nome;
      if ig
      then ig := iguais(p.^prox, q.^prox)
    end;
    iguais := ig
  end; { iguais }
```

Exercício: Construa a correspondente versão iterativa.

- **Remoção de um Elemento:**

{ Retirar da Lista, o Nó referido por este Ponteiro }



- Conhecer o valor do Ponteiro este não basta. É necessário conhecer o Ponteiro ant, do elemento anterior da Lista;

- A Remoção pretendida:

```
ant^.prox:= este^.prox;
dispose(este);
```

- Não é efectivamente necessário conhecer o Ponteiro este,

```
aux:= ant^.prox;
ant^.prox:= aux^.prox;
dispose(aux);
```

- A instrução,

```
ant^.prox:= ant^.prox^.prox;
```

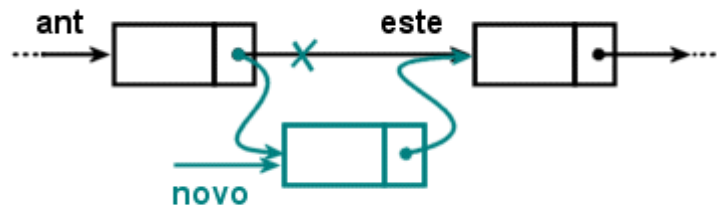
também eliminaria o elemento pretendido, mas não libertaria o espaço de Memória ocupado por este^.

- A Remoção do primeiro elemento da Lista tem de ser tratada como um caso particular:

```
aux:= ponta;
ponta:= ponta^.prox;
dispose(aux);
```

- **Inserção de um Elemento:**

{ Inserir um Elemento, referido pelo Ponteiro novo,
antes do Nó da Lista referido por este. }

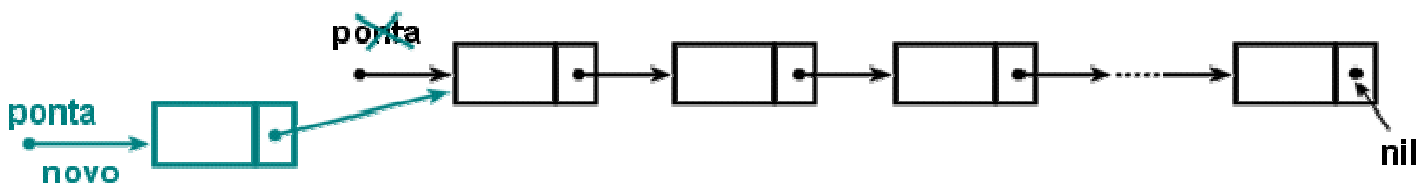


- Igualmente, é necessário conhecer o Ponteiro ant, do elemento anterior da Lista;
- A Inserção pretendida obtém-se por:

```
novo^.prox:= este;      { novo^.prox:= ant^.prox; }
ant^.prox:= novo;
```

{ Qual seria o resultado da troca das duas atribuições? }

- A Inserção do primeiro elemento da Lista também tem de ser tratada como um caso particular:



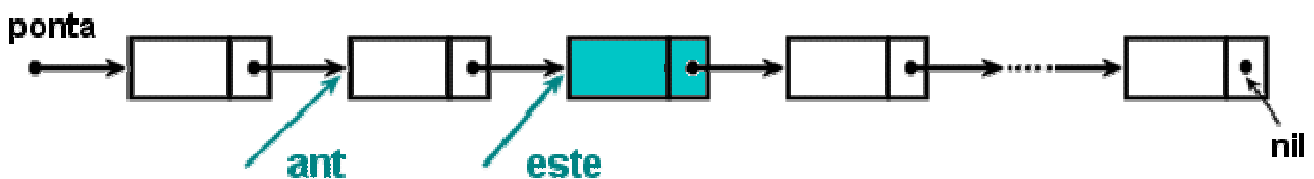
```
novo^.prox:= ponta;
ponta:= novo;
```

- As operações de **Pesquisa** numa Lista Ligada devem portanto fornecer, não apenas o Ponteiro que refere o Elemento procurado mas também o do Elemento anterior a esse.

- **Pesquisas numa Lista Ligada:**

{ Procurar estenome numa Lista, a partir da sua ponta. }

- Uma Pesquisa consiste numa Travessia na Lista, destinada a identificar o Ponteiro este (onde estenome = este^.nome) bem como o Ponteiro ant (tal que ant^.prox = este):



- Uma primeira tentativa de solução seria talvez:

```
ant:= nil; este:= ponta;
```

```
while (estenome <> este^.nome) do
  begin ant:= este;
        este:= este^.prox;
  end;
```

contudo, se estenome não pertencer à Lista, ocorreria um **erro** no fim da Lista (este = **nil**) quando a condição do Ciclo tentasse aceder ao campo este^.nome.

- Uma tentativa de correcção como,

```
...
while (este <> nil) and (estenome <> este^.nome) do
...
```

provocaria o **mesmo erro**, pois a segunda parte da Expressão Lógica é calculada, mesmo que a primeira tenha o valor Falso.
{ A linguagem Pascal não possui o operador **cand** }

- Como sempre, a solução mais simples (e correcta) consiste na utilização de uma variável lógica:

{ Módulo de Pesquisa numa Lista Ligada Linear }

```
procedure pesquisar( estenome : tipo; ponta : ponteiro;
                    var ant, este : ponteiro; var achou : boolean);

    begin ant:= nil;
        este:= ponta;

        achou:= false;
        while not achou and (este <> nil) do
            begin if estenome = este^.nome
                then achou:= true  ←
                else begin ant:= este;
                        este:= este^.prox
                end
            end;
        { achou or (este = nil) }

    end; { pesquisar }
```

- Este Módulo obedece efectivamente às especificações inicialmente estipuladas pois, à saída do Ciclo:

{ achou **or** (este = nil) } $\wedge$ { achou = (estenome = este^.nome) }

assim, se achou (= Verdadeiro)
 então { estenome = este^.nome } $\wedge$
 { ant^.prox = este }
 senão { **not** achou } $\wedge$ { este = nil }

- Este Módulo de Pesquisa tem uma vasta gama de aplicações, como por exemplo:

{ **Remover** o Nó que contém estenome, caso exista }

- O Módulo de Remoção deve também informar se o Elemento da Lista foi efectivamente removido;
- Note-se que o valor do Parâmetro **ponta** pode ser alterado, pois o elemento removido pode ser o primeiro da Lista;



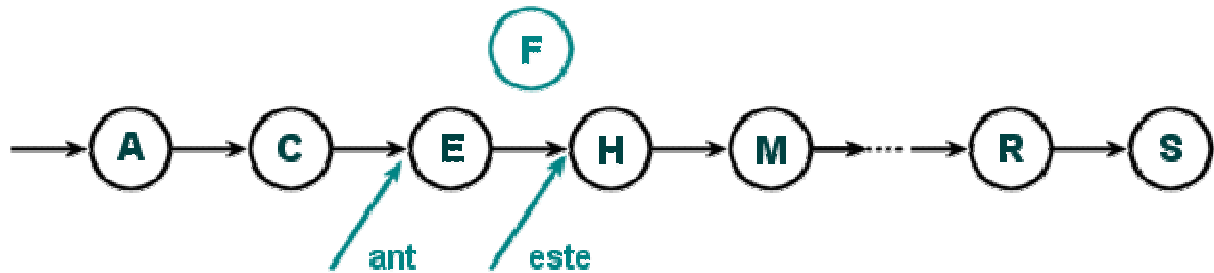
```
procedure remover ( estenome : tipo; var ponta : ponteiro;  
                    var sucesso : boolean);  
    var ant, este : ponteiro;  
        achou : boolean;  
  
    begin pesquisar(estenome, ponta, ant, este, achou);  
        sucesso:= achou;  
  
        if achou  
        then begin if ant = nil  
            then { O primeiro elemento }  
                ponta:= este^.prox  
            else { Caso geral, com (ant <> nil) }  
                ant^.prox:= este^.prox;  
                dispose(este)  
            end  
        end; { remover }
```



- **Listas Ordenadas:**

{ Pesquisar estenome numa Lista Ligada Ordenada }

- Um Módulo de Pesquisa Ordenada deve devolver a referência do Elemento que contém estenome, caso exista, bem como a do Elemento anterior, ou a localização que teria se existisse;



```

procedure PesqOrd( estenome : tipo; ponta : ponteiro;
                    var ant, este : ponteiro; var achou : boolean);
var maior : boolean;

begin ant:= nil; este:= ponta;
      achou:= false; maior:= true;

      while maior and (este <> nil) do
        begin if estenome <= este^.nome
              then maior:= false
              else begin ant:= este;
                    este:= este^.prox
                  end
        end;

      { not maior or (este = nil) }
      { (estenome <= este^.nome) or (este = nil) }

      if este <> nil
      then achou := estenome = este^.nome
           {(este = nil) ⇒ (estenome > “último nome”) }
      end; { PesqOrd }
  
```

{ Inserir um Elemento com estenome numa Lista Ligada Ordenada }

- O Módulo de Inserção deve também informar se o Elemento foi efectivamente inserido na Lista;

```
procedure InserOrd ( estenome : tipo; var ponta : ponteiro;  
                    var sucesso : boolean);  
    var ant, este, novo : ponteiro;  
        achou : boolean;  
  
    begin PesqOrd(estenome, ponta, ant, este, achou);  
  
        sucesso:= not achou;  
  
        if not achou  
        then begin new(novo);  
                    novo^.nome:= estenome;  
                    if ant = nil  
                    then begin { Primeiro elemento }  
                                novo^. prox:= ponta;  
                                ponta:= novo  
                                end  
                    else begin { Caso geral }  
                                novo^. prox:= ant^.prox;  
                                ant^.prox:= novo  
                                end  
                    end  
  
        end  
  
    end; { InserOrd }
```

- Verifique que a situação do Elemento inserido ser o último da Lista (este = **nil**), não precisa de tratado como caso particular.

{**Aplicação:** Construir uma Lista Ordenada a partir de um conjunto desordenado de nomes, fornecidos pelo teclado. }

```
procedure CriarLista (var ponta : ponteiro);  
    begin ponta:= nil  
    end; { CriarLista }
```

```
procedure ConstruirListaOrd (var ponta : ponteiro);  
    var estenome : tipo;  
        sim : boolean;  
  
    begin CriarLista(ponta);  
        while not eof(input) do  
            begin readln(estenome);  
                InserOrd(estenome, ponta, sim)  
            end  
    end; { ConstruirListaOrd }
```

- Nesta versão, são ignorados os nomes que aparecerem repetidos. Noutros casos, o parâmetro sim poderia, por exemplo, ser utilizado para contar as repetições;

Exercício: Utilize o Módulo InserOrd para, a partir de uma Lista qualquer, construir uma Lista Ordenada.

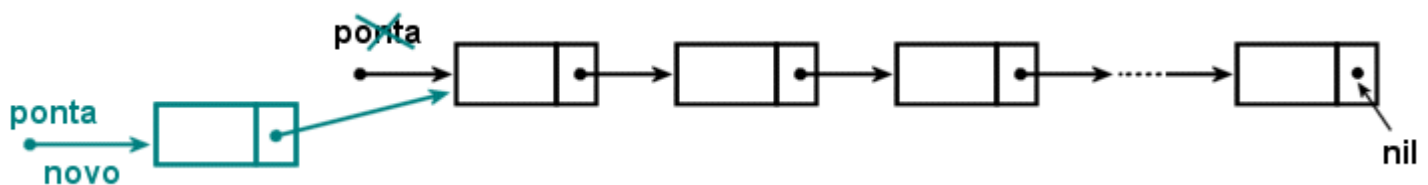
Com base nesta ideia, construa um Módulo para Ordenar uma dada Lista, sem gerar “novos” espaços de Memória. Vai precisar de alterar o Módulo InserOrd para:

```
procedure InserOrd2 (este : ponteiro;  
    var ponta : ponteiro;  
    var sucesso : boolean);
```

- **Construção Sequencial de Listas:**

{ Construir uma Lista, a partir de um conjunto de nomes fornecidos pelo teclado, pela mesma ordem que aparecem. }

- Antes disso, vejamos como seria mais fácil construir a Lista com os Elementos dispostos por ordem inversa da que são lidos;



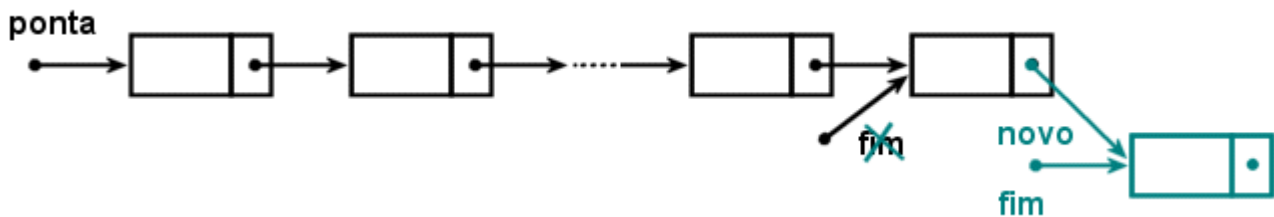
```

procedure ConstruirListaR (var ponta : ponteiro);
    var estenome : tipo;
        novo : ponteiro;

    begin ponta:= nil;
        while not eof(input) do
            begin readln(estenome);
                new(novo);
                novo^.nome:= estenome;
                novo^.prox:= ponta;
                ponta:= novo;
            end
    end; { ConstruirListaR }
  
```

- Note que, a inicialização `ponta:= nil;` tem dois efeitos: coloca o necessário **nil** no fim da Lista e, caso não existam nomes, gera a Lista Vazia.

- Vejamos agora a versão pretendida: a “dificuldade” consiste no facto de que os novos elementos são juntos no fim da Lista.



```

procedure ConstruirLista (var ponta : ponteiro);
    var estenome : tipo;
        novo, fim : ponteiro;

    begin if eof(input)
        then ponta:= nil
        else begin { Primeiro Elemento }
            readln(estenome);
            new(ponta);
            ponta^.nome:= estenome;
            ponta^.prox:= nil;
            fim:= ponta;
            { Restantes Elementos }
            while not eof(input) do
                begin readln(estenome);
                    new(novo);
                    novo^.nome:= estenome;
                    novo^.prox:= nil;
                    fim^.prox:= novo;
                    fim:= novo;
                end
            end
    end; { ConstruirLista }

```

- O **if** inicial bem como o tratamento especial do primeiro elemento podem ser evitados, começando por criar um nó “provisório” (ao qual são ligados todos os outros) que é posteriormente eliminado.

- **Observação:**

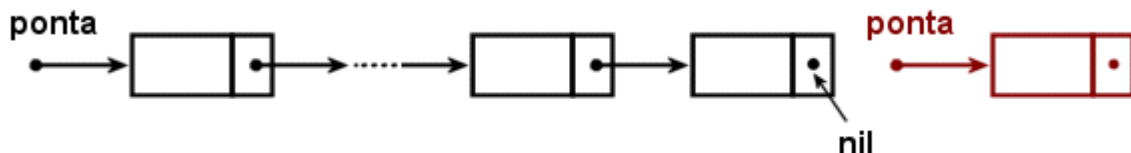
{ Uma **Estratégia Recorrente** para:
Juntar um Nó com estenome no fim de uma Lista. }



```

procedure Juntar ( estenome : tipo; var ponta : ponteiro);

    begin if ponta = nil
        then begin new(ponta);
            ponta^.nome:= estenome;
            ponta^.prox:= nil
        end
        else Juntar (estenome, ponta^.prox)
    end;
  
```



- A “ligação” é feita pela Ligação por Referência do próprio parâmetro **ponta** que (na última chamada recorrente) entra com o valor **nil** e sai com o valor gerado por **new(ponta)**;
- Apesar da sua aparente simplicidade, esta versão realiza efectivamente uma Travessia ao longo de toda a Lista, procurando o último elemento;
- Claro que esta não é esta a forma mais eficiente de juntar um elemento no fim de uma Lista.

Problema: Somar Polinômios

{ Somar dois Polinômios de coeficientes reais e (uma) variável real. }

- Como **Representar** um Polinômio?

$$A(x) = 3x^{14} + 2x^8 - 2x^7 + 1$$

Sendo um Polinômio (também) uma estrutura recorrente,

$$\text{Polinômio} = \begin{cases} \text{Polinômio Nulo} \\ \text{Termo} + \text{Polinômio} \end{cases}$$

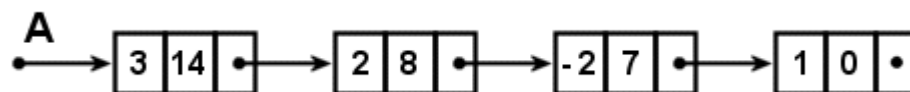
uma Representação em Lista Ligada é a mais conveniente:

```

type polinomio = ^termo;
      termo      = record coef : real;
                    expo : integer;
                    prox : polinomio
      end;
  
```

```

var A, B : polinomio;
  
```

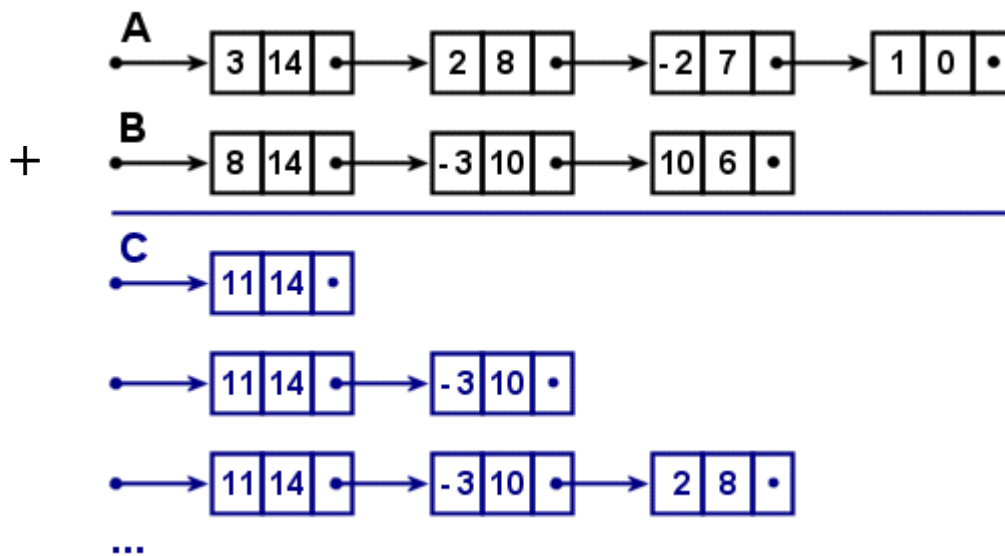


- Assim, torna-se possível reduzir o Problema da Soma de Polinômios a um Problema de **Fusão** Ordenada de Listas, com:

se dois termos tiverem o mesmo expoente

então juntar termo com esse expoente e a soma dos coeficientes

senão juntar o termo de maior expoente;



- A Construção Sequencial do Polinómio Soma (C) pode ser simplificada, com a utilização do Módulo:

```

procedure JuntarTermo ( var fim : polinomio;
                        coeficiente : real; expoente : integer);
var p : polinomio;

begin new(p);
      p^.coef:= coeficiente;
      p^.expo:= expoente;
      p^.prox:= nil;
      fim^.prox:= p;
      fim:= p
end; { JuntarTermo }

```

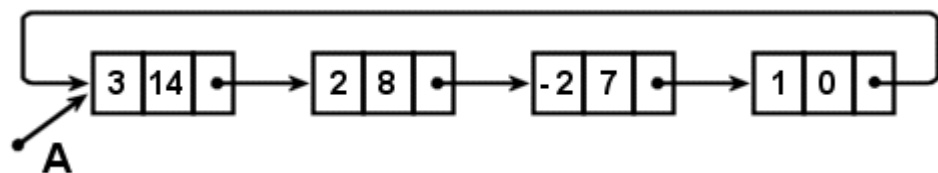
- Será também utilizado um "termo" provisório, que é criado antes do processo de Fusão e eliminado depois. Note-se ainda que a soma de dois Polinómios Nulos deve produzir um Polinómio **Nulo** (assim como a soma de dois Polinómios Simétricos).

```
procedure SomarPolinomios (A, B : polinomio; var C : polinomio);  
  var a, b, c, aux : polinomio;  
      soma : real;  
  begin { a e b são os ponteiros móveis em A e B }  
    a:= A, b:= B;  
    { Nó provisório para a construção de C }  
    new(C); C^.prox:= nil;  
    { c é o ponteiro móvel em C }  
    c:= C;  
    while (a <> nil) and (b <> nil) do  
      if a^.expo = b^.expo  
      then begin soma:= a^.coef + b^.coef  
        if soma <> 0  
        then JuntarTermo(c, soma, a^.expo);  
        a := a^.prox; b := b^.prox  
      end  
      else if a^.expo > b^.expo  
      then begin JuntarTermo(c, a^.coef, a^.expo);  
        a := a^.prox  
      end  
      else begin JuntarTermo(c, b^.coef, b^.expo);  
        b := b^.prox  
      end;  
    { Acabar de copiar o resto de A ou de B }  
    while (a <> nil) do  
      begin JuntarTermo(c, a^.coef, a^.expo);  
        a := a^.prox  
      end;  
    while (b <> nil) do  
      begin JuntarTermo(c, b^.coef, b^.expo);  
        b := b^.prox  
      end;  
    { Eliminar o Nó Prvisório }  
    aux:= C;  
    C:= C^.prox;  
    dispose(aux)  
  end; { SomarPolinomios }
```

- **Listas Circulares:**

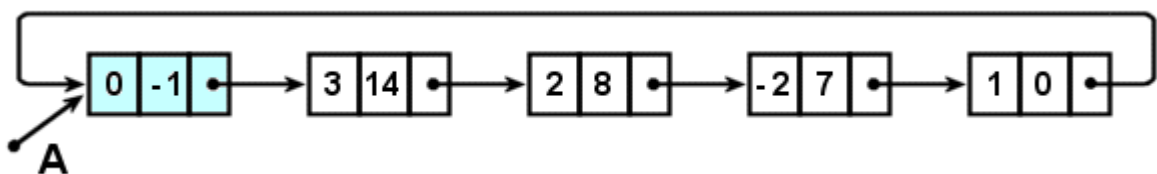
- Muitas vezes, torna-se conveniente introduzir uma pequena modificação na Representação das Listas Ligadas: o último elemento (em vez do Ponteiro Nulo) aponta para o primeiro.

Como por exemplo, na representação de Polinómios:



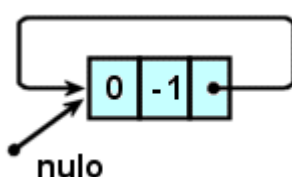
Nesse caso, em vez de **while** (p <> nil) do,
as Travessias são controladas por **while** (p <> ponta) do.
Então, com que valor de p começaria o ciclo?

- Por isso, as Listas Circulares costumam ter um elemento **Base**:



Na representação de Polinómios, a Base corresponde a um "Termo fictício". Convém identificá-lo com um expoente inferior a qualquer valor possível como, por exemplo, expo = -1.

O Polinómio **Nulo** será então representado por:



```

procedure SomarPolinomiosCirculares (baseA, baseB : polinomio;
                                     var baseC : polinomio);
var a, b, c : polinomio;
    soma : real;
    acabou : boolean;

begin a:= baseA, b:= baseB;
    { Construção da baseC }
    new(baseC); baseC^.coef:= 0;
                    baseC^.expo:= -1;
    c:= baseC;
    acabou := false;
    while not acabou do
        if a^.expo = b^.expo
        then if a^.expo = -1
            then begin { Fim de ambos os Polinómios }
                acabou:= true;
                { Fechar a Lista Circular } ←
                c^.prox:= baseC
            end
        else { Expoentes iguais }
            begin soma:= a^.coef + b^.coef
                if soma <> 0
                then JuntarTermo(c, soma, a^.expo);
                    a := a^.prox; b := b^.prox
            end
        else { Expoentes diferentes }
            if a^.expo > b^.expo
            then begin JuntarTermo(c, a^.coef, a^.expo);
                a := a^.prox
            end
            else begin JuntarTermo(c, b^.coef, b^.expo);
                b := b^.prox
            end
        { Não existem restos de Listas para copiar }
        { Não existem Nós provisórios para eliminar }
    end; { SomarPolinomiosCirculares }

```

- **Listas Duplamente Ligadas:**

- Outra alternativa para a representação de uma Lista Ligada, consiste na utilização de **dois** ponteiros.

```

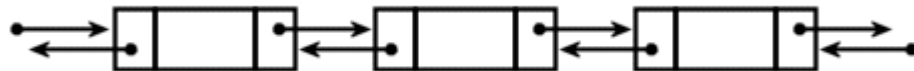
type ponteiro = ^registro;
      registro = record elemento : tipo;
                  ant, prox : ponteiro
      end;

```

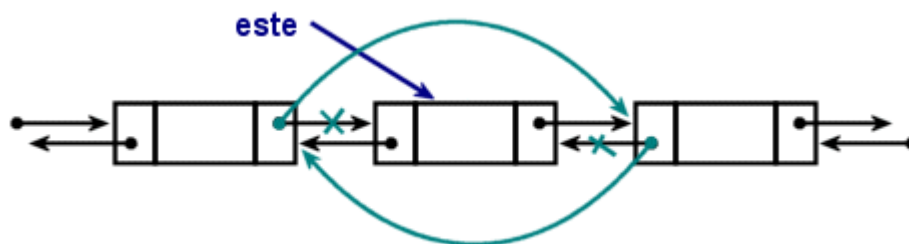
```

var p, este, aquele : ponteiro;

```



- A primeira vantagem é óbvia: são possíveis Travessias nos **dois sentidos** (uma seguindo os ponteiros ant, outra pelos prox).
- Outra vantagem é que, para a Remoção de um elemento, deixa de ser necessário conhecer a referência do elemento anterior.

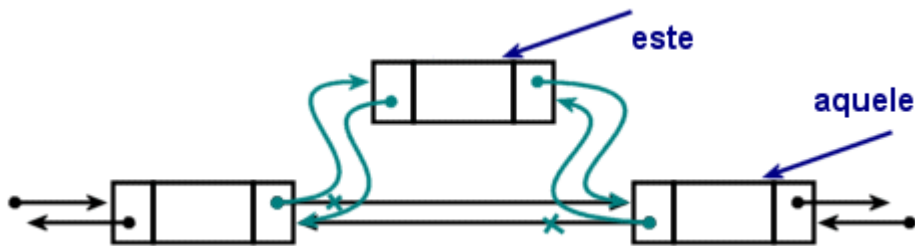


```

{ Eliminar este }
este^.ant^.prox:= este^.prox;
este^.prox^.ant:= este^.ant;
{ onde pode ser invertida a ordem das atribuições }
dispose(este);

```

- O mesmo se pode dizer acerca da Inserção de um elemento.



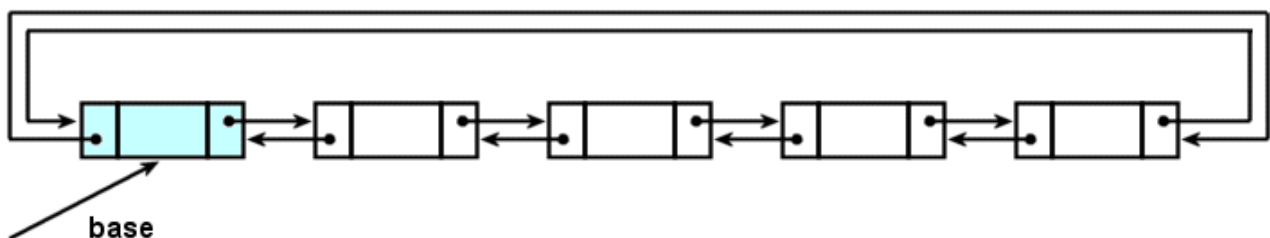
```

{ Inserir este antes de aquele }
new(este);
este^.elemento:= item;
este^.ant:= aquele^.ant;
este^.prox:= aquele;

aquele^.ant^.prox:= este;
aquele^.ant:= este;
{ onde a ordem das atribuições é vital }

```

- A maior desvantagem é também evidente: duplicação do número total de ponteiros utilizados.
- Também neste caso, pode ser conveniente a utilização de uma versão Circular, com elemento base:



- De um modo geral, podemos dizer que as Listas Duplamente Ligadas são mais "robustas", exigindo assim menor "esforço" de programação.

□ Representação de Polinómios de várias variáveis

$$P(x, y, z) = x^{10} y^3 z^2 + 2 x^8 y^3 z^2 + 3 x^8 y^2 z^2 + x^4 y^4 z + 6 x^3 y^4 z + 2 y z$$

- Como **Representar** este Polinómio?

{ 1ª solução:

coef	expo x	• →
expo y	expo z	

Desvantagem:

Dificuldade na adaptação a outros tipos de polinómios. }

- Procuremos uma solução com Nós de tamanho fixo:

$$\begin{aligned} P(x, y, z) &= x^{10} y^3 z^2 + 2 x^8 y^3 z^2 + 3 x^8 y^2 z^2 + x^4 y^4 z + 6 x^3 y^4 z + 2 y z \\ &= (x^{10} y^3 + 2 x^8 y^3 + 3 x^8 y^2) z^2 + (x^4 y^4 + 6 x^3 y^4 + 2 y) z \end{aligned}$$

{ Polinómio em z }

$$= ((x^{10} + 2 x^8) y^3 + 3 x^8 y^2) z^2 + ((x^4 + 6 x^3) y^4 + 2 y) z$$

{ Polinómio em z,

cujos coeficientes são Polinómios em y,

cujos coeficientes são Polinómios em x,

cujos coeficientes são reais. }

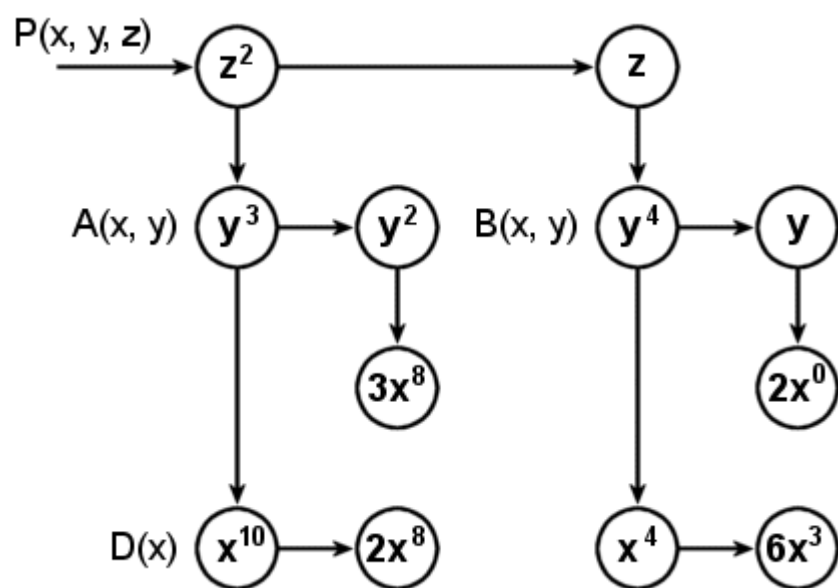
$$P(x, y, z) = \dots + A(x, y) z^2 + B(x, y) z + C(x, y)$$

$$\dots + D(x) y^3 + E(x) y^2 + F(x) y + G(x)$$

$$\dots + r x^{10} + s x^9 + t x^8 + \dots$$

{ Válido para qualquer Polinómio $P(x, y, z)$
e generalizável para qualquer número de variáveis }

Esquema de Representação:



Onde:

- Cada "lista horizontal" representa um Polinómio;
- Cada Ponteiro Horizontal representa uma Soma (de termos) e cada Ponteiro Vertical representa um Produto (por um coeficiente);
- O coeficiente de cada termo é um Polinómio (com menos uma variável);
- Os coeficientes de um Polinómio em x também são Polinómios (de grau zero : escalares).

Representação de um Polinómio de Variáveis Múltiplas:

```

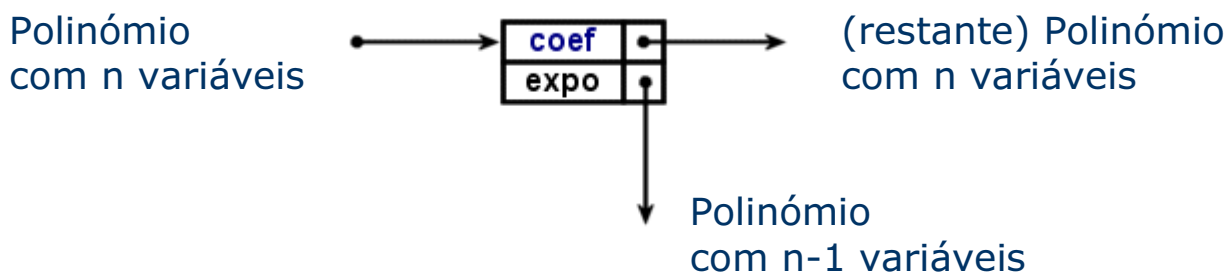
type polinomio = ^termo;
      termo      = record coef : real;
                    expo : integer;
                    baixo, direita : polinomio
      end;

```

```

var P: polinomio;

```



- Assim como um Polinómio de uma variável é uma estrutura **recorrente**,

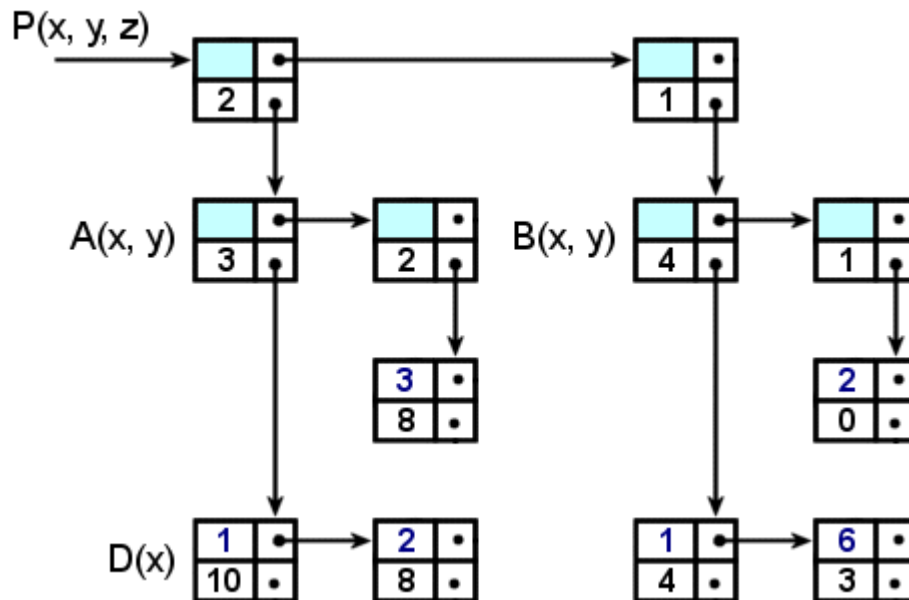
$$\text{Polinómio} = \begin{cases} \text{Polinómio Nulo} \\ [\text{Escalar} * \text{variável}^{\text{expoente}}] + \text{Polinómio} \end{cases}$$

um Polinómio de variáveis múltiplas é uma estrutura **duplamente recorrente**:

$$\text{Polinómio} = \begin{cases} \text{Polinómio Nulo} \\ [\text{Polinómio} * \text{variável}^{\text{expoente}}] + \text{Polinómio} \end{cases}$$

Portanto,

$$P(x, y, z) = ((x^{10} + 2 x^8) y^3 + 3 x^8 y^2) z^2 + ((x^4 + 6 x^3) y^4 + 2 y) z$$



- O campo coef : real; só é ocupado quando o coeficiente é um escalar (polinómio em x) e, nesse caso, baixo = **nil**.
- Uma Representação mais elaborada (correcta?) consistiria na utilização de um **campo variante** do tipo [real V ponteiro].
- A natureza duplamente recorrente desta estrutura conduz, naturalmente, à utilização de algoritmos duplamente recorrentes:

- Construir uma **Cópia** de um Polinómio:

```

C ← Cópia( P ) :
  se P = Nulo
  então C ← Nulo
  senão  Q ← Cópia( P^.baixo )
         R ← Cópia( P^.direita )
         C ← [ P^.coef, P^.expo, Q, R ]

```

{ Recorde-se que uma função calcula e fornece um valor (não estruturado), em particular um ponteiro. }



```

function copia (p : polinomio) : polinomio;
  var c, q, r : polinomio;
  begin c:= nil;
        if p <> nil
        then begin { Copiar para Baixo }
                   q:= copia(p^.baixo); ←
                   { Copiar para a Direita }
                   r:= copia(p^.direita); ←
                   { Construir Cópia deste Nó }
                   new(c);
                   c^.coef:= p^.coef;
                   c^.expo:= p^.expo;
                   c^.baixo:= q;
                   c^.direita:= r
        end;
        copia:= c
  end; { copia }

```

- E como verificar se dois Polinómios são **iguais**?

Abordagem duplamente recorrente:

$$\text{iguais}(p, q) \Leftrightarrow \begin{cases} p = q = \text{nil} \\ (p^{\wedge}.\text{coef} = q^{\wedge}.\text{coef}) \wedge (p^{\wedge}.\text{expo} = q^{\wedge}.\text{expo}) \\ \wedge \text{iguais}(p^{\wedge}.\text{baixo}, q^{\wedge}.\text{baixo}) \\ \wedge \text{iguais}(p^{\wedge}.\text{direita}, q^{\wedge}.\text{direita}) \end{cases}$$

```

function iguais (p, q : polinomio) : boolean;
var ig : boolean;

begin ig := (p = nil) and (q = nil);

    if (p <> nil) and (q <> nil)
    then begin ig := (p^{\wedge}.coef = q^{\wedge}.coef) and (p^{\wedge}.expo = q^{\wedge}.expo);
        if ig
        then begin ig := iguais(p^{\wedge}.baixo, q^{\wedge}.baixo);
            
            if ig
            then ig := iguais(p^{\wedge}.direita, q^{\wedge}.direita)
        end
    end;
    iguais := ig

end; { iguais }

```

- Por vezes, em estruturas duplamente recorrentes, é conveniente seguir uma **abordagem mista**: Iterativa + Recorrente.

- **Quantas variáveis** tem um dado Polinómio?

{ Basta determinar a **altura** (número de níveis) da estrutura }

Abordagem duplamente recorrente:

$$\text{altura}(p) = \begin{cases} 0 & \text{se } p = \text{nil} \\ 1 + \max_{p^{\text{.direita}}} \{ \text{altura}(p^{\text{.baixo}}) \} & \text{se } p \neq \text{nil} \end{cases}$$

{ Vamos utilizar uma abordagem **mista**: iterativa ao longo dos ponteiros direita e recorrente segundo os ponteiros baixo. }

```

function altura (p : polinomio) : integer;
var alt, max : integer;
    este : polinomio;

begin if p = nil
    then altura := 0
    else begin max := 0;
              este := p;
              { Travessia Iterativa para a "direita" }
              while este <> nil do
                  begin { Chamada recorrente para "baixo" }
                      alt := altura(este ^.baixo);
                      if alt > max
                      then max := alt;
                      este := este ^.direita
                  end;
              altura := 1 + max
    end
end; { altura }

```

□ Listas Generalizadas

- **Definição:** Lista

Uma **Lista** (Generalizada) é uma sequência finita de elementos,

$$A = (\alpha_1, \alpha_2, \dots, \alpha_n)$$

onde cada α_i ($i \in [1..n]$ com $n \geq 0$) é um **átomo** ou uma **Lista**.

- **Convenção:** Escrevemos os nomes de Listas em Maiúsculas e os nomes de átomos em minúsculas.
- **Terminologia:**

A é o **nome** da Lista;

n é o **comprimento** da Lista;

se $n = 0$ a Lista é **nula**;

se $n \geq 1$ então α_1 é a **cabeça** da Lista e

$(\alpha_2, \alpha_3, \dots, \alpha_n)$ é a **cauda** da Lista;

Exemplos:

() a Lista nula;

((a)) Lista com um elemento,
que é uma Lista com um elemento que é um átomo;

A = (a, (b, c)) Lista de nome A e comprimento 2;

cabeça(A) = a {um átomo}

cauda(A) = ((b, c)) {uma Lista}

cabeça(cauda(A)) = (b, c)

cauda(cauda(A)) = ()

$B = (A, A, ())$

Lista de nome B e comprimento 3;

 $\text{cabeça}(B) = A$ {uma Lista}

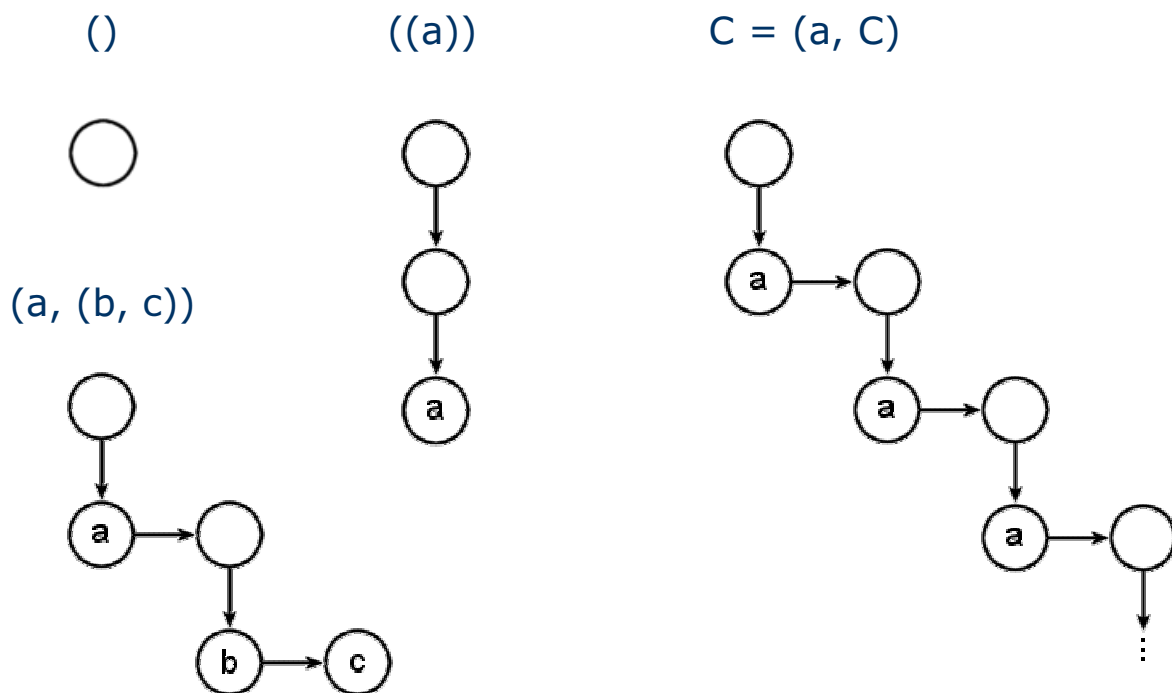
 $\text{cauda}(B) = (A, ())$
 $\text{cabeça}(\text{cauda}(B)) = A$
 $\text{cauda}(\text{cauda}(B)) = (())$
 $C = (a, C)$

Lista de nome C e comprimento 2;

Representa a sequência infinita:

 $(a, (a, (a, (a, \dots))))$

Representação Gráfica:



- Definição: **Lista Linear**

Uma Lista diz-se **Linear** se todos os seus elementos forem átomos.

