Lógica de Programação

LISTAS

Uma lista é uma coleção de elementos do mesmo tipo dispostos linearmente que podem ou não seguir determinada organização, por exemplo: [E_1, E_2, E_3, E_4, E_5,.....E*n*], onde *n* deve ser >=0.

Como exemplos de listas podemos citar: lista de chamada de alunos, lista de compras de supermercado, lista telefônica, entre outros. O exemplo apresentado a seguir será é de uma lista de pagamentos a serem efetuados em um mês, exemplo que também será utilizado nos tópicos seguintes.

Lista de pagamentos
Prestacão do carro
Cartão de crédito
Conta de luz
Condomínio
TV a cabo
Crediário das Casas Bahia

lista simples

Quando criamos uma lista para utiliza-la como estrutura de dados, podemos utilizar como contêiner para armazenamento dos dados um vetor ou uma matriz, então dizemos que esta é uma lista implementada por meio de arranjo; ou então podemos utilizar a alocação dinâmica, isto é, não criamos um contêiner para armazenar os dados, mas precisamos referenciar os elementos seguinte e anterior de cada elemento, então teremos uma lista encadeada.

Veja um exemplo de lista simples:

Lista de pagamentos ←[prestação do carro, cartão de crédito, conta de luz, condomínio, Tv a cabo, crediário das Casas Bahia].

Essa é uma lista que possui seis elementos do tipo caracter, e os elementos estão armazenados em um valor.

LISTAS ENCADEADAS

Uma lista encadeada é um conjunto de elementos que estão dispostos em uma dada organização física não linear, isto é, estão espalhados pela memória. Para organizar a lista de maneira que possa ser utilizada como um conjunto linear, é necessário que cada elemento do conjunto possua informações sobre o seu elemento anterior e o seu elemento seguinte. Para exemplificar será utilizada uma lista de pagamentos que devem ser efetuados no mês. Os pagamentos estão dispostos em uma ordem aleatória, isto é, linear:

Lista de pagamentos
Prestacão do carro
Cartão de crédito
Conta de luz
Condomínio
TV a cabo
Crediário das Casas Bahia

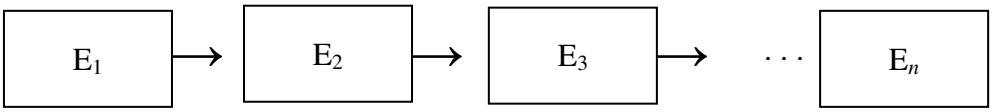
lista simples

Olhando para a lista pode-se perceber qual é o primeiro elemento, qual é o segundo e assim por diante, mas quando desejamos implementar essa lista em uma estrutura de dados precisamos dizer qual será o próximo elemento. Para isso, cada elemento da lista é representado por um nó, e cada nó deve conter os dados e um campo que indique qual é o próximo elemento da lista – esse campo é chamado de ponteiro. Observe a lista seguinte:

Lista de pagamentos	Ponteiro para o próximo elemento
Prestacão do carro	2
Cartão de crédito	3
Conta de luz	4
Condomínio	5
TV a cabo	6
Crediário das Casas Bahia	Obs.: este é o último elemento do conjunto, então não aponta para nenhum outro.

lista com um campo para encadeamento

O elemento 1 aponta para o elemento 2, o elemento 2 aponta para o elemento 3 e assim por diante:

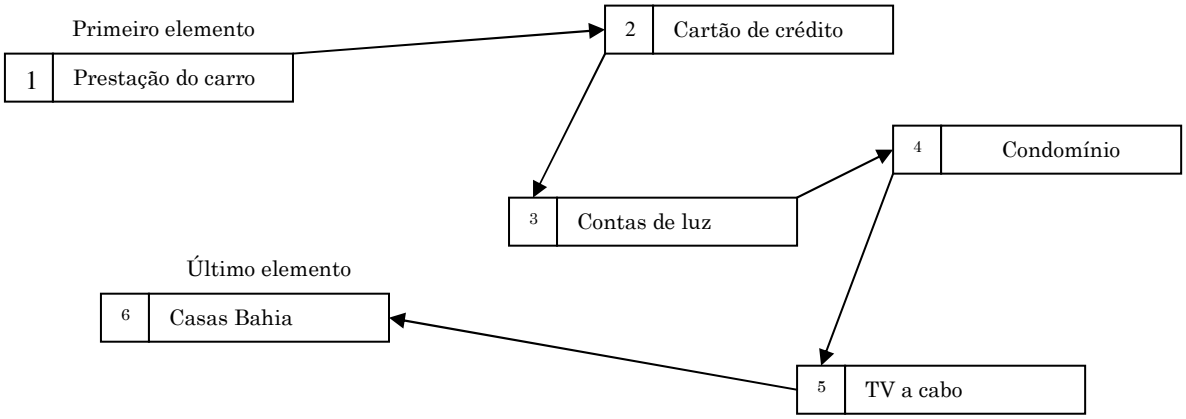


| Figura 1 | *Listas encadeadas*

onde:

- o primeiro elemento da lista é E_1;
- o último elemento da lista é E_n;
- o predecessor de E_2 é E_1;
- o sucessor de E_2 é E_3;
- e assim por diante até o último elemento.

De acordo com o exemplo apresentado teremos:



| Figura 2 | *Listas encadeadas*

Trata-se de uma lista de encadeamento simples, onde:

- O primeiro elemento da lista, ou seja, o seu começo, é prestação do carro;
- O seu sucessor é cartão de crédito, que tem como predecessor prestação do carro e assim por diante;
- O último elemento da lista, ou seja, o seu final, é crediário das Casas Bahia.

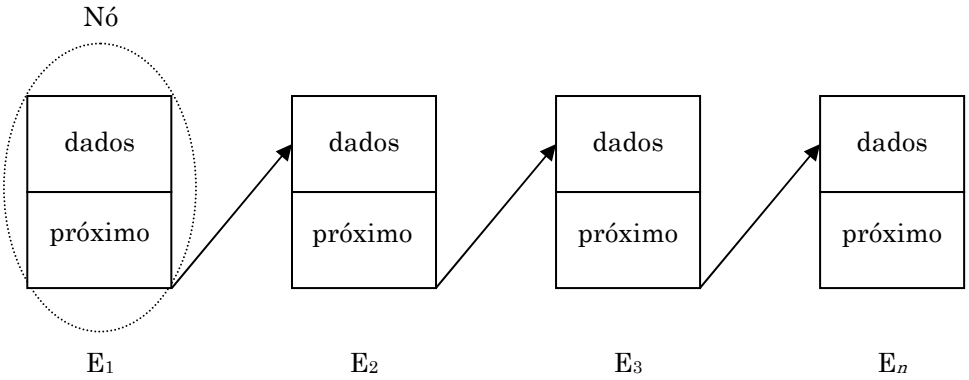
Nota: O ponteiro guarda o endereço de memória do elemento; o exemplo acima é hipotético.

TIPOS DE LISTAS ENCADEADAS

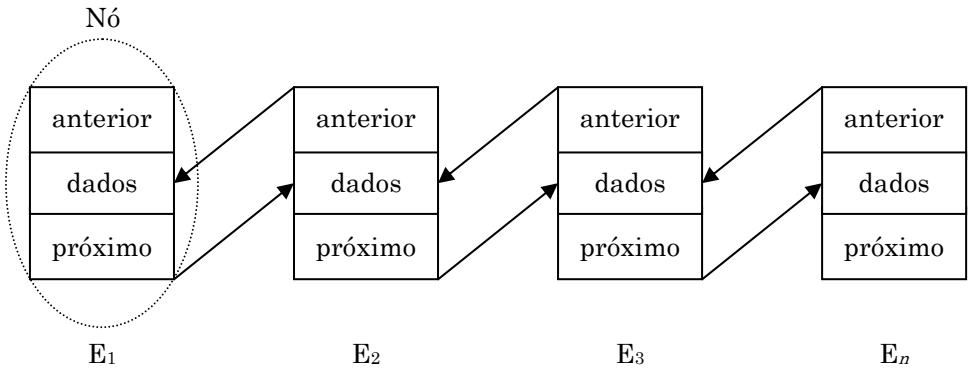
As Listas Encadeadas podem ser do tipo:

- *Encadeamento simples*: os elementos da lista possuem apenas um ponteiro que aponta para o elemento sucessor ou próximo (como no exemplo apresentado anteriormente), como mostra a figura 3.
- *Duplamente encadeadas*: cada elemento possui um campo que aponta para o seu predecessor (anterior) e outro para o seu sucessor (próximo). Veja a Figura 4.
- *Ordenadas*: a ordem linear da lista corresponde à ordem linear dos elementos, isto é, quando um novo elemento é inserido na lista ele deve ser colocado em tal posição que garanta que a ordem da lista será mantida; essa ordem pode ser definida por um campo da área de dados, como por exemplo, se tivermos uma lista ordenada com seguintes valores [1, 5, 7, 9] e desejarmos incluir um novo elemento com o valor 6, este deverá ser incluído entre 5 e 7 (figura 5).

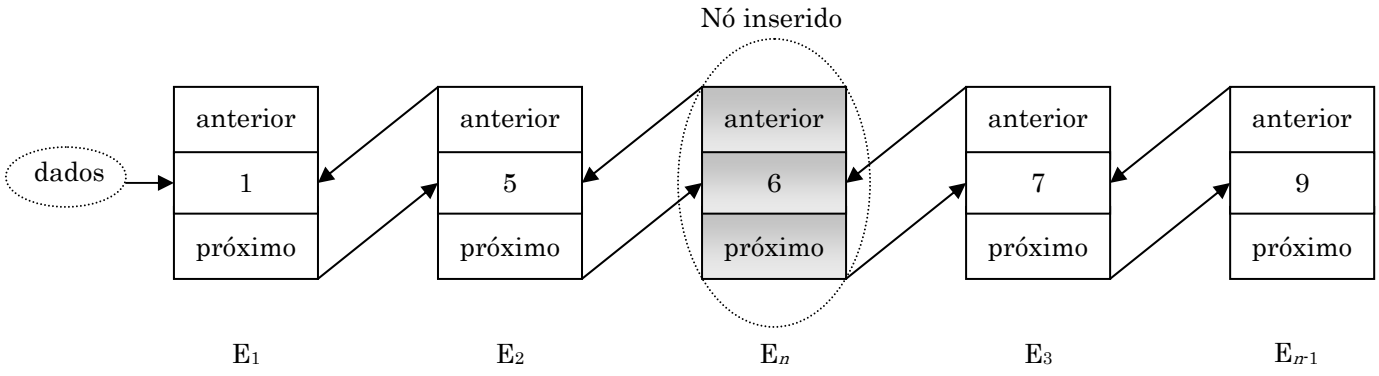
  Uma lista ordenada pode ser encadeamento simples ou duplo, mas o princípio para a ordenação é o mesmo.
- *Circulares*: o ponteiro próximo do último elemento aponta para o primeiro; e o ponteiro anterior do primeiro elemento aponta para o último. Na Figura 6, E_1 é o primeiro elemento e E*n*, o último elemento.



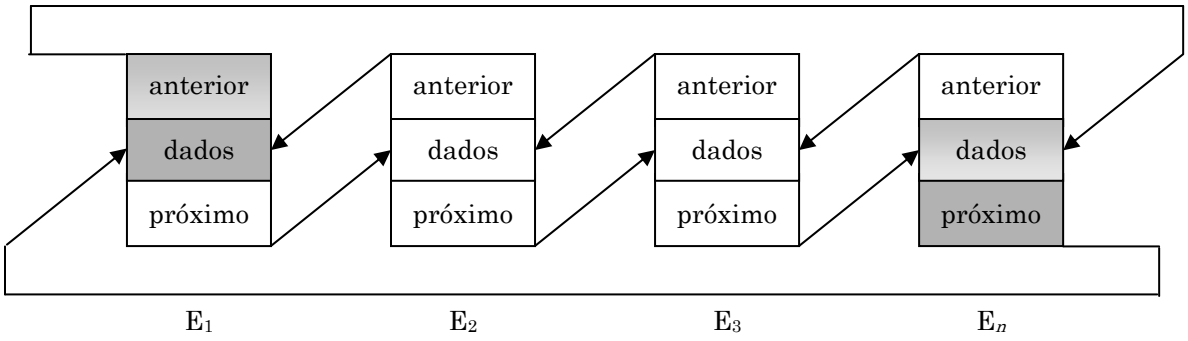
| Figura 3 | *Listas com encadeamento simples*



| Figura 4 | *Listas duplamente encadeadas*



| Figura 5 | *Listas ordenadas*



| Figura 6 | *Listas circulares*

Lembre-se: As listas podem ser implementadas por meio de arranjos ou apontadores. Na implementação por meio de arranjos os elementos da lista são armazenados em posição de memória seguidas. Os dados são armazenados em uma matriz com tamanho pré-definido.
As listas implementadas por meio de apontadores permitem que os seus elementos sejam armazenados em posições descontínuas na memória, tornando mais fáceis as operações de inserção e remoção de elementos.
Essas formas de implementação são válidas também para filas, pilhas e árvores devido à similaridade na construção; como vocês poderão notar, a diferença entre essas estruturas refere-se à manipulação.

LISTAS DE ENCADEAMENTO SIMPLES

Agora você já sabe o que é uma lista simples agora vamos implementá-la. A seguir será apresentado o algoritmo que representa a criação e a manipulação de uma lista de encadeamento simples.

EXEMPLO 1: CRIAÇÃO E MANIPULAÇÃO DE UMA LISTA SIMPLES.

```
1.  Algoritmo ExemploListaSimples
2.  Tipo apontador: ^NoSimples
3.        NoSimples = registro
4.                          valor: inteiro
5.                          prox: apontador
6.                    fim
7.        ListaSimples = registro
8.                           primeiro: apontador
9.                           ultimo: apontador
10.                   fim
11. Inicio
12.    ListaSimples.primeiro ← nulo
13.    ListaSimples.ultimo ← nulo
14.    Procedimento InsereNo_fim (var novoNo: NoSimples)
15.    inicio
16.       novoNo^.prox ← nulo
17.       Se (ListaSimples.primeiro = nulo) Então
18.          ListaSimples.primeiro ← novoNo
19.       Fim-Se
20.       Se (ListaSimples.ultimo <> nulo) Então
21.          Lista.Simples.ultimo^.prox ← novoNo
22.       Fim-se
23.       ListaSimples.ultimo ← novoNo
24.    Fim
25.    Procedimento excluiNo (var elemento: inteiro)
26.    var
27.       temp_no: NoSimples
28.       temp: NoSimples
29.    inicio
30.       temp ← ListaSimples.primeiro
31.       temp_no ← ListaSimples.primeiro^.prox
32.       Enquanto (temp_no^.prox <> nulo ou temp_no.valor <> elemento) faça
33.          temp ← temp_no
34.          temp_no ← temp_no^.prox
35.       Fim-Enquanto
36.       temp^.prox ← temp_no^.prox
37.       desposicione(temp_no)
38. Fim.
```

Na linha 2 a variável apontador é um tipo construído que terá a função de referenciar o próximo elemento de cada um dos elementos da lista, por isso ele deve ser declarado como sendo do mesmo tipo da variável que irá representar o nó. Em nosso exemplo, o nó está sendo representado pelo *NoSimples* . O ^ que precede o tipo de dado é utilizado para representar a função de 'apontar para' (*^NoSimples*), isto é, a variável apontador sempre apontará para algum outro elemento do tipo *NoSimples*.

Note, na linha 3, a declaração do *NoSimples*, também um tipo construído. Ele possui as variáveis *valor*, que é do tipo inteiro, e *prox*, que é do tipo apontador. Você se recorda de que o apontador irá fazer referência a um outro elemento; nesse caso ele irá fazer referência ao próximo nó do elemento em questão.

Na linha 7 está sendo declarada a *ListaSimples*, que também é um tipo construído; é um registro que contém as variáveis *primeiro* e *último*, que serão utilizadas na construção da lista.

Lembre-se: Isso será possível pois o primeiro^.prox *irá apontar para o próximo elemento, e o próximo para o próximo e assim por diante, pois a variável* apontador *é do tipo referência para* NoSimples.

O procedimento *InsereNo*, linha 14, recebe o parâmetro *novoNo*, que é uma variável do tipo *NoSimples*, então **lembre-se** que novoNo é um registro que possui as variáveis *valor* e *prox*. Esse procedimento é utilizado para inserir um novo nó no final da lista. Note que o *novoNo^.prox* recebe nulo – isso deve ser feito para que esse nó seja considerado o último.

Nas linhas seguintes é verificado se esse deverá ser o primeiro elemento da lista no teste *ListaSimples.primeiro = nulo*; se isso for verdadeiro, então o novoNoserá o primeiro da lista.

Na linha 20 é verificado se o último nó é diferente de nulo, quando já existem elementos na lista. Isso é feito para que o *novoNo* seja inserido na última posição da lista.

Na linha 23 a variável *ListaSimples.ultimo* recebe o valor do *novoNo*.

Dica: Se o prox *for nulo, significa que esse elemento é o último.*
Se o primeiro elemento e o último tiverem valor nulo, significa que a lista está vazia.
Se o prox *do primeiro elemento for nulo, significa que a lista só tem um elemento.*

O procedimento *excluiNo*, linha 25, recebe como parâmetro um valor para a variável *elemento*, que é do tipo inteiro, e irá guardar o dado do nó que deverá ser excluído, nesse caso para a variável *valor*. Esse procedimento também utiliza as variáveis *temp* e *temp_no*, que serão variáveis auxiliares do tipo *NoSimples*. Nas linhas 30 e 31 essas variáveis recebem,

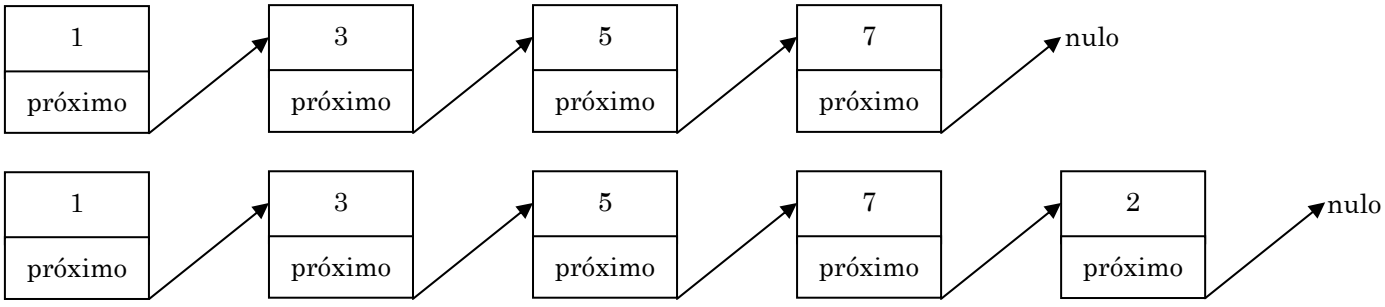
respectivamente, o valor do primeiro nó e o valor do próximo nó (do primeiro).

É aconselhável criar uma rotina para verificar se a lista está vazia ou com apenas um elemento antes de implementar esse método, senão, caso esteja, irá ocasionar um erro. Isso pode ser feito com o seguinte teste, que deve preceder a atribuição de valores das variáveis *temp* e *temp_no*:
Se ListaSimples.primeiro <> nulo e
ListaSimples.primeiro.prox <> nulo então ...

Na linha 32, é feito um teste de repetição, que utiliza a estrutura *Enquanto* para repetir as instruções enquanto o *prox*, do *temp_no*, for diferente de nulo ou *tempo_no. Valor* for diferente do elemento. Se o *temp_no.prox* for igual a nulo significa que o elemento lido é o último da lista e, se o valor do elemento for igual ao valo de *temp_no.valor*, significa que o elemento a ser excluído foi encontrado, então o laço de repetição é encerrado e são executadas as instruções *temp.prox* ← *temp_no^.prox*, por meio da qual o valor do próximo elemento do elemento a ser excluído é armazenado no próximo elemento do nó anterior ao que será excluído. **Lembre-se** de que o *temp* inicia com o valor do *primeiro* e o *temp_no*, com o valor do *primeiro.*prox. Com isso o elemento a ser excluído deixa de ser referenciado. Por último a instrução *desposicione (temp_no)* indica que o nó a ser excluído deixa definitivamente de ser referenciado – em algumas linguagens isso não precisa ser feito.

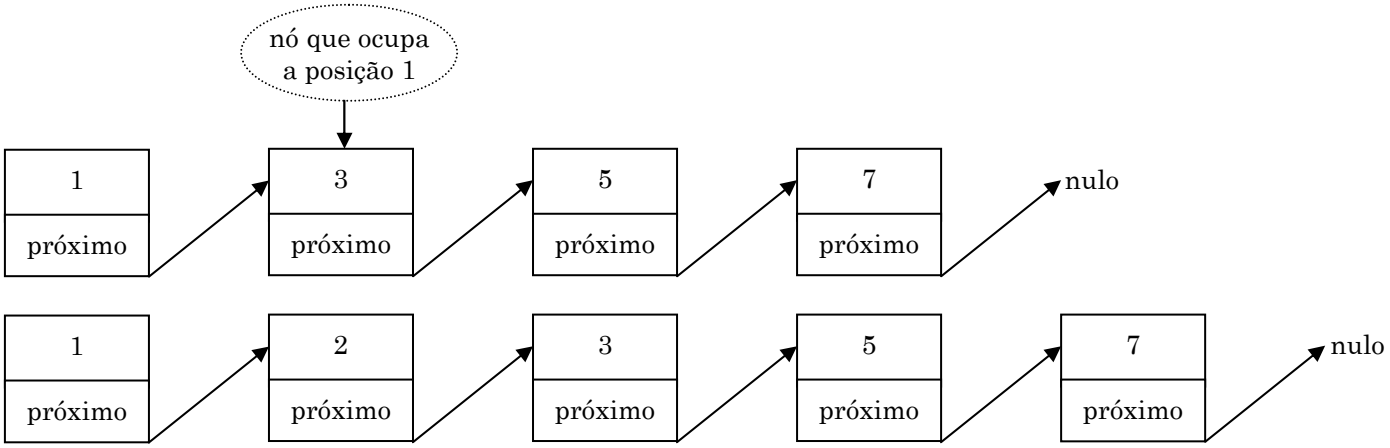
Vamos ver a seguir uma situação para utilização do algoritmo anterior. Suponha que a lista contenha os números inteiros 1, 3, 5, 7. Vamos inserir o número 2 na lista utilizando o processo *insereNo*.

O número 2 deverá ser passado como parâmetro para o procedimento e ficará armazenado na variável *novoNo*. Isso será feito no algoritmo principal da seguinte forma: *InsereNo (2)*
então o procedimento é acionado e irá verificar se esse será o primeiro elemento da lista. Como no nosso exemplo isso é falso, pois a lista já contém os elementos 1, 3,5 e 7, então será executada a instrução *ListaSimples.ultimo^.prox* ←*novoNo*, isto é a referência do último elemento irá apontar para o *novoNo*, no caso o número 2, e a variável que representa o último elemento irá receber o *novoNo*.



| Figura 7 | *Inserção de um nó no final da lista (Simples)*

Nós também podemos inserir um novo nó em uma determinada posição da lista. Para isso, é preciso que sejam passados como parâmetros o valor a ser inserido e a posição que ele deverá ocupar, por exemplo: *incluiNo (2,1)*.



|Figura 8| *Inserção de um nó em uma posição específica da lista (simples)*

A seguir serão apresentados os algoritmos para incluir nós em posição pré-determinadas:

A função *ContarNos* será utilizada para verificar a quantidade de nós existentes na lista. Ela não recebe parâmetros e retorna a quantidade de nós.

EXEMPLO 2: PSEUDOCÓDIGO PARA REPRESENTAR UMA FUNÇÃO PARA CONTAGEM DE NÓS DE UMA LISTA.

```
1.  Função ContarNos (  )
2.      var numero_nos: inteiro
3.      temp_no : NoSimples
4.  Inicio
5.      numero_nos ← 0
6.      temp_no ← ListaSimples.primeiro
7.      Enquanto (temp_no)^.prox <> nulo) faça
8.          numero_nos ← numero _nos + 1
9.          temp_no ← temp_no^.prox
10.    fim_enquanto
11. return (numero_nos)
```

O procedimento *insereNo_posicao* recebe como parâmetros as variáveis novoNo, que é do tipo *NoSimples*, e *posição*, que é do tipo inteiro.

EXEMPLO 3: PSEUDOCÓDIGO PARA REPRESENTAR A INSERÇÃO DE UM NÓ EM UMA POSIÇÃO ESPECÍFICA EM UMA LISTA DE ENCADEAMENTO SIMPLES.

```
1.  Procedimento InsereNo_posição (var novoNo: NoSimples, posição: inteiro)
2.  var
3.      temp_no: NoSimples
4.      pos_aux: inteiro
5.      numero_nos: inteiro
6.  inicio
7.      pos_aux ← 0
```

```
8.      numero_nos ←ContarNos ( )
9.      Se (posicao = 0) então
10.         novoNo^.prox ← ListaSimples.primeiro
11.         ListaSimples.primeiro ← novoNo
12.         Se (ListaSimples.ultimo = ListaSimples.primeiro) então
13.             ListaSimples.ultimo ← novoNo
14.         fim-se
15.         Senão
16.         Se (posicao <= numero_nos) então
17.             Enquanto (temp_no^.prox <> nulo .ou. posição <> pos_aux)
    então
18.                 temp_no ← temp_no^.prox
19.                 pos_aux ← pos_aux + 1
20.             Fim-Enquanto
21.             novoNo^.prox ← temp_no^.prox
22.             temp_no^.prox ← novoNo
23.             Senão
24.             Se (posicao > numero_nos( )) então
25.                 ListaSimples.ultimo.prox ← novoNo
26.                 ultimo ← novoNo
27.             fim-se
28.         fim-se
29.     fim-se
30. Fim
```

É verificado se a posição para inserção é 0 (linha 9), isto é, se o nó a ser inserido deverá ser o primeiro. Também é verificado se ele será o último elemento da lista (linha 12); se isso for verdadeiro, o valor do seu *prox* será nulo, o que significa que a lista só tem um elemento.

Se o novo nó tiver de ser inserido em outra posição, então é verificado se a posição é menor do que a quantidade de nós existentes na lista, o que é feito com o auxilio da função *ContarNos* construída anteriormente. Se isso for verdadeiro, então será utilizada uma estrutura de repetição(*Enquanto*) para encontrar o nó atual que ocupa a posição, depois é feito um deslocamento do ponteiro, e para isso é utilizada uma variável que armazena temporariamente os valores do nó. Esse deslocamento é feito nas linhas 21 e 22.

E por último é verificado se o nó deverá ser o último da lista, isto é, se a posição desejada é igual ao número de nós nas linhas 24 e 25. (Obs.: foi convencionado que qualquer valor para posição superior à quantidade de nós será o último da lista).

LISTAS DUPLAMENTE ENCADEADAS

Quando se percorre uma lista de encadeamento simples é bastante difícil fazer o caminho inverso, já nas listas de encadeamento duplo esse problema não existe, pois cada nó possui uma referência para o próximo elemento da lista e outra para o anterior.

A concepção de uma lista duplamente encadeada é bastante similar à dos simples, basta acrescentar ao nó uma variável que fará referência ao

elemento anterior, da mesma maneira que é feito com o próximo. Veja na linha 6 do *ExemploListaDupla*:

EXEMPLO 4: PSEUDOCÓDIGO PARA REPRESENTAR UMA LISTA DUPLAMENTE ENCADEADA.

```
1.  Algoritmo ExemploListaDupla
2.  Tipo apontador: ^NoDuplo
3.      NoDuplo = registro
4.                    valor: inteiro
5.                    prox: apontador
6.                    ant: apontador
7.                fim
8.      ListaDupla = registro
9.                     primeiro: apontador
10.                    ultimo: apontador
11.                    numero_nos: inteiro
12.                fim
13. Inicio
14.     ListaDupla.numero_nos ←0
15.     ListaDupla.primeiro ← nulo
16.     ListaDupla.ultimo ← nulo
```

Para escrevermos algoritmos de manipulação de lista duplamente encadeada podemos seguir o mesmo raciocínio adotado para o entendimento da lista de encadeamento simples, mas devemos lembrar que o nó anterior também precisa ser referenciado a cada manipulação.

No algoritmo a seguir, que é continuação do *ExemploListaDupla*, demonstraremos os procedimentos para inserir um nó no final da lista e para excluir um nó de acordo com a sua posição.

Nesta solução para exclusão de nó é necessário descobrir qual é o nó que ocupa a posição desejada para a exclusão; para isso, utilizaremos a função *pegarNo* ( ). Essa função recebe como parâmetro o índice do nó a ser excluído e retorna o próprio nó.

```
17. Procedimento InsereNo_fim (var novoNo: NoDuplo)
18.     Inicio
19.         NovoNo^.prox ← nulo
20.         NovoNo^.ant ← ultimo
21.         if (ListaDupla.primeiro = nulo)
22.             ListaDupla.primeiro ← novoNo
23.         fim-se
24.         if(ListaDupla.ultimo < > nulo)
25.             ListaDupla.ultimo^.prox ← novoNo
26.         fim-se
27.         ListaDupla.ultimo ← novoNo
28.     ListaDupla.numero_nos ← ListaDupla.numero_nos + 1
29.     fim
30.     Procedimento pegarNo (var indice: inteiro): NoDuplo
31.         var
32.         temp_no: NoDuplo
33.         i: inteiro
```

```
34.     inicio
35.         temp_no ← ListaDupla.primeiro
36.         Enquanto (temp_no < > nulo && i < = indice)
37.               temp_no ← temp_no^.prox
38.               i ← i + 1
39.         fim_enquanto
40.         Return temp_no
41.     fim
42.     Procedimento InsereNo_posicao (novoNo: NoDuplo, indice: inteiro)
43.     var
44.         temp_no: NoDuplo
45.     inicio
46.        temp_no ← pegarNo (indice)
47.        novoNo^.prox ← temp_no
48.        Se (temp_no^.prox < > nulo) então
49.           novoNo^.ant ← temp_no^.ant
50.           novoNo^.prox^.ant ← novoNo
51.        Senão
52.             novoNo^.ant ← novoNo
53.             ListaDupla.ultimo ← novoNo
54.        Fim-se
55.        Se (índice = 0) então
56.            ListaDupla.primeiro ← novoNo
57.            Senão
58.                novoNo^ant^.prox ← novoNo
59.            fim-se
60.            ListaDupla.numero_nos ← ListaDupla.numero_nos + 1
61.        fim
62.        Procedimento excluiNo (índice: inteiro)
63.        var
64.            temp_no: NoDuplo
65.        inicio
66.           Se (índice = 0) então
67.                ListaDupla.primeiro ← ListaDupla.primeiro^.prox
68.                Se (ListaDupla.primeiro < > nulo) então
69.                     ListaDupla.primeiro^.ant ← nulo
70.                fim-se
71.           Senão
72.                Se (temp_no < > ultimo) então
73.                   temp_no ← pegarNo (indice)
74.                   temp_no^.ant^.prox ← temp_no^.ant
75.                Senão
76.                     ListaDupla.ultimo ← temp_no
77.                fim-se
78.           fim-se
79.           ListaDupla.numero_nos ← ListaDupla.numero_nos – 1
80.        fim
81. fim.
```