

Notas de Aula de Algoritmos e Programação de Computadores

FLÁVIO KEIDI MIYAZAWA

com a colaboração de

TOMASZ KOWALTOWSKI

Instituto de Computação - UNICAMP

Versão 2000.1

Estas notas de aula não devem ser usadas como única fonte de estudo. O aluno deve ler outros livros disponíveis na literatura.

Nenhuma parte destas notas pode ser reproduzida, qualquer que seja a forma ou o meio, sem a permissão dos autores.

Os autores concedem a permissão explícita para a utilização e reprodução deste material no contexto do ensino de disciplinas regulares dos cursos de graduação sob a responsabilidade do Instituto de Computação da UNICAMP.

© Copyright 2000

Instituto de Computação
UNICAMP
Caixa Postal 6176
13083-970 Campinas-SP
{fkm,tomasz}@ic.unicamp.br

15 Ponteiros

Ponteiros (*pointers*) ou apontadores são variáveis que armazenam um endereço de memória (e.g., endereço de outras variáveis).

Na linguagem Pascal cada ponteiro tem um tipo, podendo ser um ponteiro para um tipo pré-definido da linguagem Pascal, ou um tipo definido pelo programador. Assim, podemos ter ponteiros para *integer*, ponteiro para *real*, ponteiro para *TipoAluno*, etc.

Quando um ponteiro contém o endereço de uma variável, dizemos que o ponteiro está “apontando” para essa variável.

Como um ponteiro é apenas um endereço de memória, não é preciso muitos bytes para sua definição, em geral são usados 4 bytes para isso.

Vamos representar um ponteiro P da seguinte maneira:



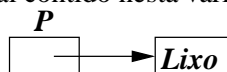
Neste caso, a variável–Ponteiro P , lado esquerdo, está apontando para a variável/memória que está do lado direito. Internamente, a variável P contém o endereço da variável apontada.

• Declaração:

Para declarar um ponteiro de um certo tipo, usamos a seguinte sintaxe:

```
var Lista_Ident_Pont: ^TipoPonteiro;
```

Ao declarar um ponteiro P , o endereço inicial contido nesta variável deve ser considerado como *lixo*, i.e.:



• Atribuindo um endereço para um ponteiro:

Vamos supor que temos uma variável, V , e queremos que um ponteiro, P , (ponteiro para **integer**) aponte para esta variável. Para isso, usamos o operador $@$ para obter o endereço de V , da seguinte maneira:

```
P := @V;
```

Um ponteiro também pode receber o conteúdo de outro ponteiro. Além disso, existe um endereço especial, chamado **nil** (nulo), que serve para dizer que é um endereço nulo e não teremos nenhuma variável neste endereço de memória. Este endereço é útil para dizer que o ponteiro ainda não tem nenhum endereço válido.

```
P := nil;
```

• Trabalhando com a memória apontada pelo ponteiro:

Para acessar a memória que o ponteiro P está apontando, usamos o operador $^$, com a sintaxe $[P^]$. Naturalmente P deverá ter um endereço válido, i.e., P deve estar apontando para uma variável ou para uma memória de tipo compatível.

Na figura seguinte apresentamos exemplos das operações com ponteiros e a representação de cada comando.

{Programa}	{Configuração da Memória}
var X:integer; p,q:^integer; begin	p lixo X q lixo lixo
X := 10;	p lixo X q lixo 10
p := @X;	p X q lixo 10
q^ := 30;	{ERRO: q aponta p/ lixo}
q := p;	p X q 10
q^ := p^ + 10;	p X q 20
writeln(p^); end.	{ Imprime o valor 20 }

O comando $q^:=30;$ causa um erro no programa; assim, vamos supor que este comando é apenas ilustrativo e não

afeta os demais comandos.

Exemplo 15.1 Considere dois vetores de “alunos” (registros) usando o tipo *TipoAluno* (veja página 90), que temos que listar ordenados por nome, digamos pelo algoritmo *QuickSort* (veja página 110). Os dois vetores não devem ser alterados, nem mesmo na ordem dos seus elementos.

Este problema poderia ser resolvido facilmente usando um terceiro vetor para conter a união dos dois primeiros e em seguida ordenamos este terceiro vetor por nome. Desta maneira manteríamos os dois vetores originais intactos. Entretanto, se cada registro do tipo *TipoAluno* for muito grande e a quantidade de alunos nos dois vetores também for grande, teríamos dois problemas:

1. A declaração do terceiro vetor deve usar uma memória grande o suficiente para acomodar a união dos dois primeiros vetores.
2. Os algoritmos de ordenação, baseados em troca de elementos, iriam fazer muitas transferências de bytes (lembrando que para trocar dois elementos fazemos 3 atribuições de variáveis, veja rotina *troca* na página 68).

Uma maneira mais econômica e rápida para resolver este problema é usar o terceiro vetor como um vetor de ponteiros para alunos. I.e., cada elemento do terceiro vetor aponta para um elemento dos dois vetores. Note que cada elemento do terceiro vetor deve gastar em torno de 4 bytes, já que é apenas um endereço, gastando menos memória que do modo anterior. Na figura 40(a) apresentamos a representação dos três vetores com os elementos do terceiro vetor apontando para os elementos dos dois vetores originais. Na figura 40(b) apresentamos o terceiro vetor já ordenado. No programa da página 145 apresentamos a função de comparação e a rotina de troca de elementos usadas para ordenar o terceiro vetor.

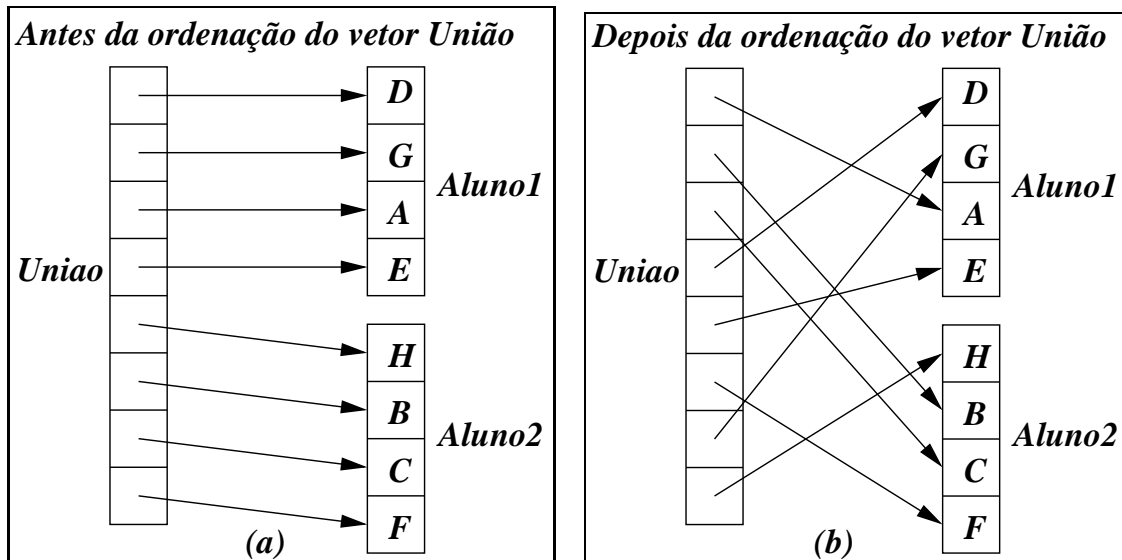


Figura 40: Ordenação de dois vetores, usando um terceiro vetor de ponteiros.

```

program OrdenaDoisVetores;
const MAXALUNOS      = 100;
type
  { Inserir declarações de TipoAluno }
  TipoApontadorAluno = ^TipoAluno; { Apontador de TipoAluno }
  TipoVetorAluno = array[1..MAXALUNOS] of TipoAluno;
  TipoVetorApontadorAluno = array[1..2*MAXALUNOS] of TipoApontadorAluno; { Vetor de Apontadores }
procedure LeVetorAluno(var V : TipoVetorAluno;var n:integer);
var i : integer;
begin
  write('Entre com a quantidade de alunos a ler: '); readln(n);
  for i:=1 to n do begin
    write('Entre com o nome do aluno: '); readln(V[i].nome);
    { ... leia outros atributos do aluno ... }
  end;
end; { LeVetorAluno }
procedure TrocaApontadorAluno(var ApontAluno1,ApontAluno2 : TipoApontadorAluno);
var ApontAux : TipoApontadorAluno;
begin
  ApontAux:=ApontAluno1;
  ApontAluno1:=ApontAluno2;
  ApontAluno2:=ApontAux;
end; { TrocaApontadorAluno }
function ComparaApontadorAluno(var ApontAluno1,ApontAluno2:TipoApontadorAluno):char;
begin
  if (ApontAluno1^.nome > ApontAluno2^.nome) then ComparaApontadorAluno:= '>'
  else if (ApontAluno1^.nome < ApontAluno2^.nome) then ComparaApontadorAluno:= '<'
  else ComparaApontadorAluno:= '=';
end;
var Aluno1,Aluno2      : TipoVetorAluno;
  Uniao                : TipoVetorApontadorAluno;
  i,n1,n2,nuniao       : integer;
begin
  LeVetorAluno(Aluno1,n1);
  LeVetorAluno(Aluno2,n2);
  nuniao := 0;
  for i:=1 to n1 do begin
    nuniao := nuniao + 1;
    Uniao[nuniao] := @Aluno1[i];
  end;
  for i:=1 to n1 do begin
    nuniao := nuniao + 1;
    Uniao[nuniao] := @Aluno2[i];
  end;
  { Chamada da rotina QuickSort usando rotina de comparação: ComparaApontadorAluno
  e rotina de troca de elementos: TrocaApontadorAluno }
  QuickSortApontAluno(Uniao,nuniao,ComparaApontadorAluno);
  for i:=1 to nuniao do begin
    writeln(Uniao[i]^ .nome,Uniao[i]^ .rg,{ ... } Uniao[i]^ .cr);
  end;
end.

```

15.1 Alocação Dinâmica de Memória

A linguagem Pascal permite que possamos “pedir” mais memória que a usada para o sistema operacional. Este nos retornará o endereço da memória que nos foi alocada ou um aviso que não foi possível alocar a memória requisitada. A memória a ser requisitada deve ter um tipo associado a ela e para receber o endereço desta nova memória, devemos usar um ponteiro do mesmo tipo da memória sendo requisitada. Por exemplo, usamos um ponteiro para *TipoAluno* para obter memória do tipo *TipoAluno*.

O comando para obter a memória é através do comando **new** da seguinte maneira:

new(*p*);

Este comando aloca uma memória que será apontada pelo ponteiro *p*. Caso o sistema operacional não consiga alocar esta memória, o ponteiro *p* retornará com valor **nil**. Esta nova memória não é memória de nenhuma variável, mas sim uma memória nova que pode ser usada com o mesmo tipo daquele declarado para o ponteiro *p*. Além disso, qualquer memória que tenha sido obtida através do comando **new** pode ser liberada novamente para o sistema operacional, que poderá considerá-la para futuras alocações de memória. O comando para liberar uma memória obtida por alocação dinâmica, apontada por um apontador *p*, é através do comando **dispose** da seguinte maneira:

dispose(*p*);

A figura seguinte apresenta um programa contendo alguns comandos usando alocação dinâmica e a situação da memória após cada comando. Alguns dos comandos causam erro de execução e foram colocados apenas para fins didáticos, vamos supor que estes não afetam os demais comandos.

<i>program exemplo;</i> <i>type TipoRegistro = record</i> <i> nome: string[50]</i> <i> idade: integer;</i> <i>end;</i> <i>var p,q: ^TipoRegistro;</i> <i>R: TipoRegistro;</i> <i>begin</i>	CONFIGURAÇÃO DA MEMÓRIA APÓS COMANDOS
<i>new(p);</i> <i>R.nome := 'Carlos';</i> <i>R.idade := 50;</i> <i>writeln(p^.nome);</i>	<i>p</i> [] → lixo <i>R</i> = [lixo lixo] <i>q</i> [] → lixo
<i>p^.nome := 'José';</i> <i>p^.idade := 10;</i>	<i>p</i> [] → [lixo lixo] <i>R</i> = [Carlos 50] <i>q</i> [] → lixo PROBLEMA: <i>p^.nome</i> ainda não foi inicializado
<i>p := nil;</i>	<i>p</i> [] → [José 10] <i>R</i> = [Carlos 50] <i>q</i> [] → lixo PROBLEMA: Memória alocada antes foi perdida
<i>new(q);</i> <i>q^ := R;</i>	<i>p</i> [] → nil [José 10] <i>q</i> [] → [Carlos 50] <i>R</i> = [Carlos 50]
<i>writeln(p^.nome);</i> <i>p := @R;</i> <i>q^.idade := 60;</i>	ERRO: <i>p</i> não aponta para memória <i>p</i> [] → [José 10] <i>q</i> [] → [Carlos 60] <i>R</i> = [Carlos 50]
<i>writeln(q^.nome, ' ', q^.idade);</i>	{ <i>Imprime 'Carlos 60'</i> }
<i>dispose(p);</i>	ERRO: <i>p</i> não aponta para memória obtida por alocação dinâmica.
<i>dispose(q);</i> <i>end.</i>	<i>p</i> [] → [José 10] <i>q</i> [] → lixo <i>R</i> = [Carlos 50]

15.2 Listas Ligadas

Uma das grandes restrições de se representar um conjunto de elementos por vetores é a limitação imposta pelo número de elementos na declaração do vetor. O número de elementos declarado no tamanho de um vetor deve ser um valor fixo e nos traz dois inconvenientes:

1. Devemos assumir uma declaração do vetor usando muitos elementos para que não tenhamos problemas quando precisarmos manipular muitos elementos.
2. Uma declaração de um vetor com muitos elementos pode causar desperdício de memória. Note que se todos os programas usassem vetores com tamanhos muito grandes, provavelmente teríamos poucos programas residentes na memória do computador.

Um vetor pode representar vários elementos de maneira simples e direta. Isto é possível pois seus elementos estão em posições contíguas de memória, embora não seja possível estender esta memória quando precisarmos inserir mais elementos que o definido no tamanho do vetor.

Usando alocação dinâmica podemos representar um conjunto de dados usando uma quantidade de memória proporcional ao número de elementos sendo representados no conjunto. Isto pode ser feito fazendo uma interligação entre os elementos do conjunto usando tipos recursivos de dados. Um elemento de um *tipo recursivo* é um registro de um tipo onde alguns dos campos são apontadores para o mesmo tipo. A primeira vista parece estranho, uma vez que estamos definindo um campo que referencia um tipo que ainda estamos definindo. Vamos chamar estes campos de *campos de ligação*. Estes campos de ligação nos permitirão fazer um encadeamento entre os elementos de maneira a poder percorrer todos os elementos de um conjunto através de apenas um ponteiro inicial. Além disso, poderemos inserir novos elementos neste encadeamento, usando alocação dinâmica.

Na linguagem Pascal podemos definir tipos recursivos das seguintes maneiras:

type	type
<i>TipoRegistro</i> = record	<i>TipoApontadorRegistro</i> = ^ <i>TipoRegistro</i> ;
{ Campos com dados do elemento }	<i>TipoRegistro</i> = record
ListaDeCamposDeLigação: ^ <i>TipoRegistro</i> ;	{ Campos com dados do elemento }
end ;	ListaDeCamposDeLigação: <i>TipoApontadorRegistro</i> ;
	end ;

Um importante exemplo de um tipo recursivo é a estrutura de dados chamada *lista ligada*, que assim como o vetor, representará uma seqüência de n elementos.

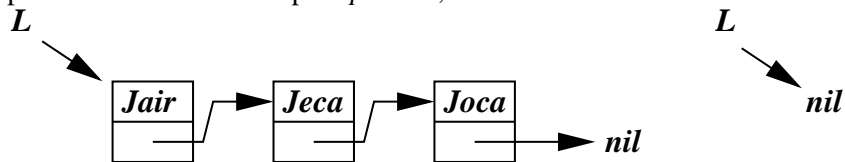
Dado um conjunto de elementos C , uma lista de elementos de C pode ser definida recursivamente como:

1. Uma seqüência vazia.
2. Um elemento de C concatenado a uma lista de elementos C .

Para representar listas ligadas em Pascal podemos usar de registros com apenas um campo de ligação, que indicará o próximo elemento da seqüência. Usaremos um ponteiro para representar toda esta seqüência. Caso este ponteiro esteja apontando para **nil**, estaremos representando a seqüência vazia. Vamos ver um exemplo de declaração de lista ligada:

```
type
  TipoElementoListaAluno = record
    NomeAluno: TipoString;
    proximo: ^TipoElementoListaAluno
  end;
  TipoLista = ^TipoElementoListaAluno
var Lista: TipoLista;
```

A seguinte figura apresenta duas listas do tipo *TipoLista*, uma com 3 elementos e uma lista vazia:



Vamos chamar o elemento (ou ponteiro) que representa o início da lista como *cabeçalho* ou *cabeça* da lista e a lista começando do segundo elemento em diante, também uma lista, chamada de *cauda* da lista.

Primeiramente vamos descrever uma rotina para listar os elementos contidos em uma lista ligada. Para isso, usaremos um outro ponteiro auxiliar para percorrer os elementos da lista. Inicialmente este ponteiro auxiliar começará no cabeçalho. Uma vez que tenhamos processado o elemento apontado por este ponteiro auxiliar, iremos para o próximo elemento, usando o campo *proximo* do elemento corrente. Isto é feito até que o ponteiro auxiliar chegue no fim da lista, i.e., quando o ponteiro auxiliar tiver valor **nil**. A seguir apresentamos uma rotina para imprimir os nomes de alunos de uma lista ligada, bem como os tipos usados.

```

type
  TipoApontElementoLista = ^TipoElementoLista;
  TipoElementoLista = record
    Aluno: TipoAluno;
    proximo: TipoApontElementoLista;
  end;
  TipoLista = TipoApontElementoLista;

procedure ImprimeAlunosLista(Lista: TipoLista);
var Paux:TipoApontElementoLista;
begin
  Paux := Lista;
  while (Paux<>nil) do begin
    writeln(Paux^.nome, ' ', Paux^.RA, ' ', Paux^.CR);
    Paux := Paux^.proximo;
  end;
end;

```

Vamos ver como podemos inserir um novo elemento no início da lista *L*. Primeiramente vamos alocar memória para o novo elemento, ler os dados nesta memória e por fim inserir o elemento no início da lista. Para esta última parte devemos tomar especial cuidado na ordem das operações para que nenhum elemento da lista seja perdido. Podemos dividir esta operação nos seguintes passos:

1. Alocar memória para o novo elemento, digamos *Paux*.
2. Ler os dados na memória apontada por *Paux*.
3. Como *Paux* será o primeiro elemento da lista *L*, os atuais elementos da lista devem seguir *Paux*. Assim, podemos fazer *Paux^.proximo* apontar para o primeiro elemento da lista atual.
4. Atualizar o cabeçalho *L* para que fique apontando para a posição *Paux*.

A figura 41 apresenta uma rotina para inserção de um elemento no início da lista e a configuração da memória após cada comando.

Note que mesmo que a lista *L* represente uma lista vazia, a lista resultante conterá uma lista com exatamente um elemento.

A idéia para desenvolver uma rotina para remover o primeiro elemento da lista segue a mesma idéia. A rotina *RemovePrimeiroAlunoLista*, descrita na página 149, apresenta tal operação.

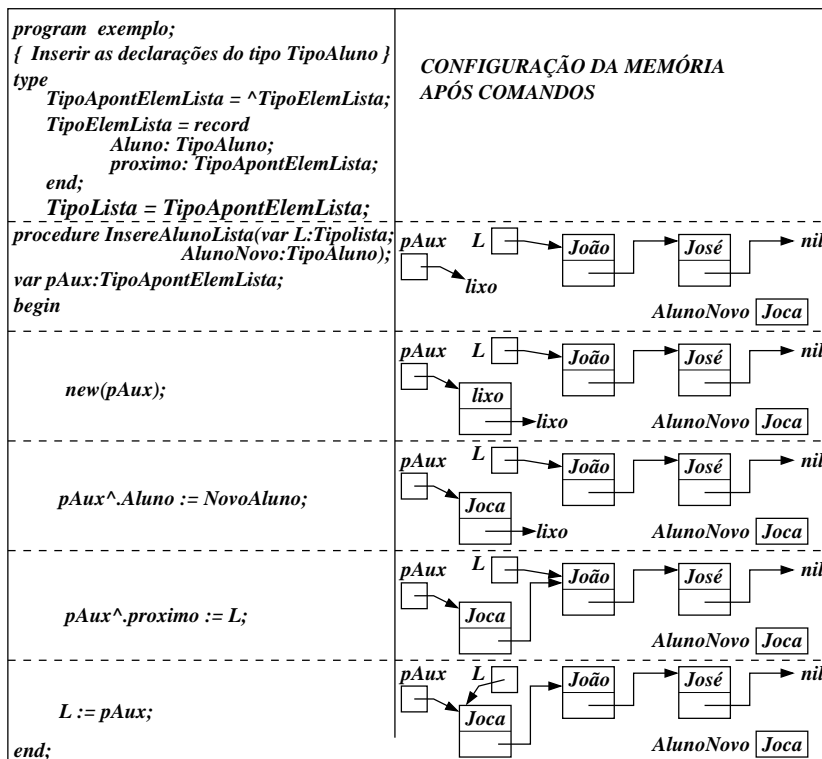


Figura 41: Inserção de elemento em uma lista ligada.

```

function RemovePrimeiroAlunoLista(var L:TipoLista; var AlunoRemovido:TipoAluno):boolean;
var Paux:TipoApontElemLista;
begin
  if L=nil then RemoveAlunoLista := false
  else begin
    Paux:=L; {Guarda o primeiro elemento antes de atualizar L}
    L:= L^.proximo; {Atualiza L para começar no segundo elemento}
    AlunoRemovido := Paux^.Aluno; {Copia os dados do elemento para o }
    dispose(Paux); {parâmetro de retorno, e libera a memória alocada para o elemento}
  end;
end;

```

Exemplo 15.2 O procedimento seguinte inverte a ordem dos elementos da lista.

```

procedure InverteListaAluno(var p: TipoLista);
var q,r,s : TipoApontElemLista;
begin
  q := nil;
  r := p;
  while r<>nil do begin
    s := r^.prox;
    r^.prox := q;
    q := r;
    r := s
  end;
  p := q
end;

```

Como tanto listas como vetores são usados para representar uma seqüência ordenada de elementos, vamos ver algumas diferenças entre estas duas estruturas de dados.

	Vetores	Listas simplesmente ligadas
Número máximo de elementos	restrito ao tamanho declarado no vetor.	Restrito a quantidade de memória disponibilizada pelo sistema operacional.
Acesso ao i -ésimo elemento	Direta.	A partir do primeiro elemento até chegar ao i -ésimo.
Inserção no início	É necessário deslocar todos os elementos de uma posição.	Em tempo constante.
Processamento de trás para frente	Processamento direto na ordem.	Necessita mais processamento ou memória.

15.3 Recursividade e Tipos Recursivos

Muitas vezes o uso de rotinas recursivas para manipular dados com tipos recursivos nos possibilita gerar rotinas simples e de fácil entendimento. Esta facilidade se deve pelo tipo ser recursivo, pois ao acessar um ponteiro para um tipo recursivo, digamos P , podemos visualizá-lo como uma variável que representa todo o conjunto de elementos, digamos C , acessíveis por ele. Do mesmo jeito, se um elemento deste conjunto contém um campo de ligação, digamos P' , este representa um subconjunto $C' \subset C$ com as mesmas características de P . Isto permite que possamos chamar a mesma rotina que foi definida para o ponteiro P recursivamente para P' (projeto indutivo). A figura 42 ilustra o tipo recursivo para o caso de lista ligada. O primeiro elemento apontado pelo ponteiro L representa a lista inteira. O ponteiro $L^{\wedge}.proximo$ representa uma sublista da lista original.

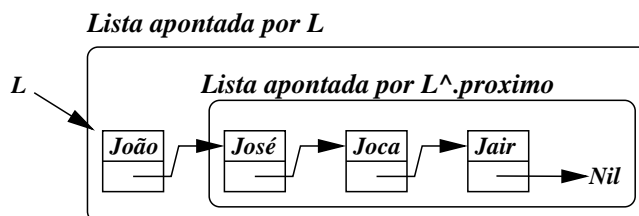


Figura 42: Inserção no fim de uma lista ligada usando recursividade.

Exemplo 15.3 *Este importante exemplo mostra como podemos fazer uso de recursividade e passagem de parâmetros por referência para inserir um elemento no fim de uma lista.*

```

procedure InsereAlunoFimLista(var L:TipoLista; var AlunoNovo:TipoAluno);
begin
  if L=nil then begin {Chegou no fim da lista}
    new(L);      {Aloca a memória para ser apontada por L}
    L^.Aluno := AlunoNovo; {Insera o último elemento}
    L^.proximo := nil;   {Ultimo elemento deve ter campo proximo igual a nil}
  end
  else InsereAlunoFimLista(L^.proximo,AlunoNovo); {Insera na lista definida por L^.proximo}
end;
  
```

Primeiramente verifique que a chamada desta rotina para uma lista vazia (L aponta para nil) insere de fato um novo elemento, fazendo com que a lista fique com um elemento. É importante observar que o ponteiro L retorna atualizado, uma vez que o parâmetro L foi declarado com passagem de parâmetro por referência (usando **var**).

Agora observe o caso que a lista contém pelo menos um elemento. Neste caso, a primeira chamada recursiva deve fazer uma nova chamada recursiva para que o elemento seja inserido no fim da lista $L^{\wedge}.proximo$. Note que $L^{\wedge}.proximo$ é um apontador e também representa uma lista. Assim, esta segunda chamada recursiva deve inserir o elemento no fim desta sublista como desejamos. Note que mesmo que $L^{\wedge}.proximo$ seja uma lista vazia ($L^{\wedge}.proximo=nil$), este voltará atualizado com o novo elemento.

A observação dos seguintes itens pode ajudar no desenvolvimento de rotinas para estruturas dinâmicas, como listas ligadas e árvores.

1. Desenvolva as rotinas simulando e representando a estrutura de dados de maneira gráfica, e.g., desenhadas em um papel. Para cada operação atualize seu desenho.
2. Tome cuidado na ordem das operações efetuadas pela rotina. Verifique se após a execução de um passo, não há memória dinâmica perdida.
3. Desenvolva inicialmente a rotina considerando uma estrutura contendo vários elementos. Verifique se a rotina funciona para a estrutura com poucos elementos, sem elementos ou com 1 elementos. Caso necessário, adapte sua rotina.
4. Aproveite a volta da recursão, para possível processamento posterior do elemento.
5. Se for o caso, faça a atualização dos campos de ligação por passagem de parâmetros por referência.

15.4 Exercícios

1. Faça um procedimento recursivo para imprimir uma lista de alunos, na ordem da lista.
2. Faça um procedimento recursivo para imprimir uma lista de alunos, na ordem inversa da lista.
3. Faça um procedimento não recursivo para inserir um aluno no fim da lista.
4. Faça um procedimento recursivo para remover um aluno com certo nome da lista.
5. Faça um procedimento recursivo para duplicar uma lista em ordem inversa.
6. Uma lista ligada é declarada da seguinte maneira:

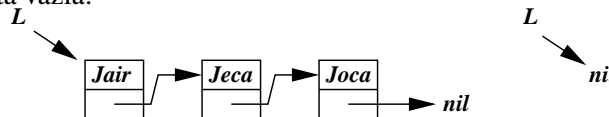
```

type  TipoString = string[255];
      TipoElementoLista = record
          nome: TipoString;
          proximo: ^TipoElementoLista;
      end;
      TipoLista = ^TipoElementoLista;
  
```

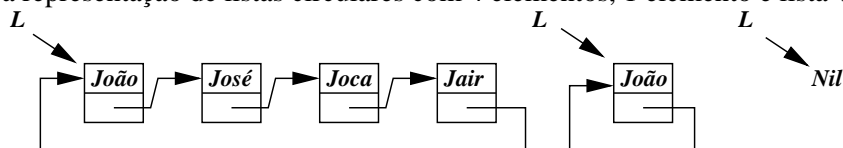
Faça uma rotina recursiva com o seguinte cabeçalho

procedure InserirOrdenadoLista(**var** L:TipoLista; Nome:TipoString);

A rotina insere um novo elemento na lista, com nome dado no parâmetro *Nome*. A função insere de maneira ordenada, i.e., os elementos devem estar ordenados. Além disso, a lista já contém os elementos ordenados antes da inserção. A seguinte figura apresenta exemplo de duas listas ordenadas do tipo *TipoLista*, uma com 3 elementos e uma lista vazia:



7. Faça um procedimento para ordenar uma lista ligada, usando o algoritmo do quicksort tomando sempre o primeiro elemento como pivô.
8. Uma lista é dita circular quando temos uma lista muito parecida com a lista ligada, com exceção que o último elemento da lista (campo próximo do último elemento) aponta para o primeiro elemento da lista. Na figura seguinte apresentamos a representação de listas circulares com 4 elementos, 1 elemento e lista vazia.



Faça uma rotina que insere um elemento em uma lista circular (não estamos preocupados na posição da lista circular onde o elemento vai ser inserido). Faça uma rotina para remover um elemento da lista, caso este exista.

9. Em um certo jogo temos vários homens que devem ser executados, sendo que eles estão dispostos de maneira a formar um círculo. O carrasco pode escolher um dos homens para ser libertado e usa a seguinte regra: Ele escolhe um primeiro homem a ser executado, executa-o e escolhe o segundo homem após o homem que acabou de ser executado como próximo homem a ser executado. Isto continua até que tenha sobrado apenas um homem, sendo que este homem é libertado. Faça um programa que leia uma lista de nomes de homens e coloque em uma lista circular. Em seguida le o nome do primeiro homem a ser executado. A partir disto, lista na ordem o nome dos homens a serem executados, e por fim o nome do homem libertado.
10. Quando temos uma lista ligada só podemos caminhar por esta lista apenas em um dos sentidos. Uma maneira de contornar este tipo de restrição é usar uma lista duplamente ligada. Neste tipo de lista temos em cada elemento dois campos de ligação. Um chamado *proximo*, que tem a mesma função que o campo proximo em listas ligadas e outro campo chamado *anterior*, que aponta para um elemento anterior na seqüência definida pelo campo proximo. O primeiro elemento da lista tem seu campo anterior com valor **nil**. A figura seguinte apresenta uma lista duplamente ligada com 4 elementos. Faça um programa com rotinas para inserção no início da lista, remoção do primeiro elemento e remoção de um elemento com um determinado nome.

