#### *Listas Lineares – implementação com Arranjo*

**Introdução**

Até o momento, ao longo da disciplina de Estrutura de Dados, utilizou-se ***arranjo*** para armazenamento de dados em memória principal. Foi assim no estudo de *Listas*, *Pilhas* e *Filas*. Já é notório que ***arranjo*** é uma forma conveniente para armazenar e manipular coleções de dados homogêneos, como nas estruturas até aqui estudadas. Um *arranjo unidimensional* – *vetor* – é por essencia uma estrutura linear, adequado aos casos em estudo. Ainda possui uma indexação implícita, que permite acesso direto a qualquer elemento da coleção.

Porém ***arranjos*** possuem alguns inconvenientes. A alocação de memória é estática, ou seja, um *vetor* é teoricamente alocado em tempo de compilação de código. Deste modo, em aplicações onde não se conhece previamente a quantidade esperada de elementos da coleção de dados, corre-se o risco de alocar um vetor muito grande, causando subutilização de memória, ou muito pequeno, com risco de *overflow* da estrutura e adoção de medidas operacionalmente custosas de realocação de memória. *Arranjos* ainda ocupam posições contíguas de memória. Isso pode causar dificuldades de encontrar memória disponíveis para estruturas muito grande. Pode ainda obrigar deslocamentos de elementos nos casos de inserções ou retiradas de elementos em posições intermediárias da coleção.

Uma forma alternativa para armazenar coleções de dados em memória é por *alocação dinâmica* de memória com ***encadeamento***, que será detalhada nas seções seguintes.

**Alocação dinâmica de memória e ponteiros na linguagem C**

A maioria das linguagens de programação modernas possuem mecanismos para alocar dinamicamente memória, que são manipuladas por meio de *referências* (endereços) de memória, os chamados *apontadores*, ou ***ponteiros*** ou ainda *pointers*. Com esse mecanismos, o programador pode alocar a qualquer momento, área de memória para armazenar os dados que eventualmente esteja manipulando.

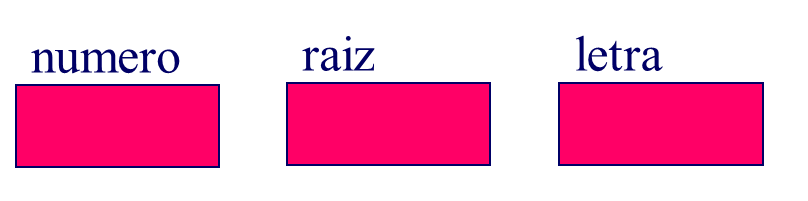
Nas linguagens de programação, as variáveis convencionais podem ser definidas como ***variáveis estáticas***. Estas possuem um formato e um tamanho de memória pre-determinados e são alocadas previamente em tempo de compilação e permanecem durante todo o tempo de execução. Por isso, são chamadas de *estáticas*. Exemplo de variáveis estáticas na linguagem C:

int numero;

float raiz;

char letra;

Estas instruções na prática alocam áreas de memórias que o programa acessa por meio do nome definido para a variável. A figura abaixo abstrai estas variáveis.



Variáveis estáticas armazenam valores dentro do escopo de seu respectivo tipo de dados. Exemplo:

Numero = 10;

raiz = 13.45;

letra = ꞌAꞌ;



Por outro lado, ***variáveis dinâmicas*** são aquelas que são alocadas dinamicamente em tempo de execução, podem ter tamanho indefinidos e são referenciadas por endereços de memória, também denominados *apontadores*, ***ponteiros*** ou *pointers*.

Define-se ***ponteiro*** com uma variável (dinâmica) que armazena um ***endereço de memória***, e não um valor de um tipo de dado. Porém o endereço que o ponteiro armazena esta associado a um tipo de dados, simples ou primitivo (em C, *int*, *float*, *char*, etc.) ou composto (*struct* ou *arranjo*). Ponteiros ou referencias são usados para:

* Alocar dinamicamente em tempo de execução memória para estruturas associadas;
* Liberar espaço de memória alocado dinamicamente;
* Armazenar referencias (endereços de memória) de estruturas de dados;
* Encadear elementos de dados formando estruturas encadeadas.

**Declarando ponteiros na linguagem C**

Em C, declara-se ***ponteiro*** da mesma forma que uma variável estática, porém com um ***asterisco*** precedendo o nome do ponteiro. Exemplos:

int \*pInt; **// ponteiro para área de memória que armazena um inteiro**

float \*pReal; **// ponteiro para área de memória que armazena um valor real**

char \*pChar; **// ponteiro para área de memória que armazena um caractere**

Aluno \*aluno; **// ponteiro para área de memória que armazena um tipo**

**// estruturado *Aluno* (admitindo já previamente declarado).**

O exemplo a seguir traz uma pequena demonstração de uso de um ponteiro para um valor inteiro na linguagem C:

01: int \*pI;

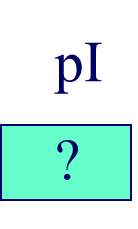
02: int i = 10;

03: pI = &(i);

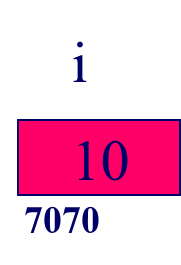
04: printf(“%d”,\*pI);

05: printf(“%d”,pI);

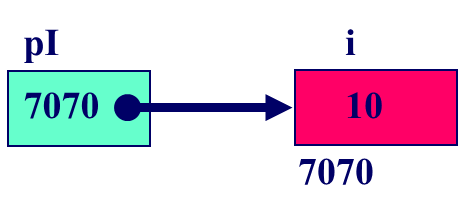
No código acima, a instrução da linha 01 aloca uma área de memória denominada ***pI***, que permite armazenar ***endereço*** de uma área de memória que armazena um valor inteiro, porém inicialmente sem nenhum endereço especificado.



A instrução da linha 02 cria uma *variável* *estática* denominada ***i*** (para efeito didático, admite-se que seu endereço seja **7070)**, e atribui a essa área, o valor inteiro 10.



A instrução da linha 03 atribui à variável ***pI*** o ***endereço*** da variável ***i***, ou seja, **7070** conforme definido acima. Observe na abstração abaixo, que ***pI*** “aponta” para a área de memória da variável ***i***, daí que *variáveis dinâmicas* são denominadas ***ponteiros***.



A instrução da linha 04 apresenta na tela o valor ***10***, ou seja, na linguagem C, ao referenciar uma variável dinâmica com o asterisco ( ***\*pI*** ), está-se referenciando o ***conteúdo*** ***da área apontada***.

A instrução da linha 05 apresenta na tela o endereço ***7070***, ou seja, ao referenciar uma variável dinâmica sem o asterisco ( ***pI*** ), está-se referenciando o seu próprio conteúdo, que é o endereço da área apontada.

**Ponteiros para estruturas (*structs*) na linguagem C**

Avalie agora o trecho de código C abaixo:

01:  **struct** ponto {**float** x, y};

02: **typedef** **struct** ponto **Ponto**;

03: **Ponto** p;

04: p.x = 10;

05: p.y = 12;

06: **Ponto** \* pp = &p;

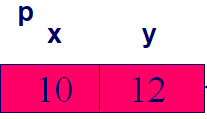
07: **. . .**

08: pp = (pp)**malloc**(**sizeof**(**struct** ponto));

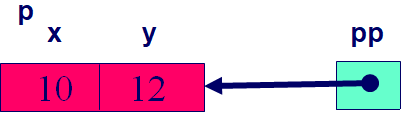
09: pp->x = 3;

10: pp->y = 4;

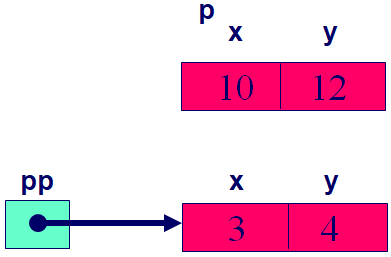
O código da linha 01 cria um tipo de dado *ponto* (estrutura com dois campos *float*) enquanto que a linha 02 dá o nome (ou álias) ***Ponto*** para esta estrutura. O código da linha 03 cria uma variável (*estática*) do tipo *Ponto* de nome ***p*** e as linhas 04 e 05 atribuem valores para os campos ***x*** e ***y*** de p, conforme ilustrado a seguir:



A instrução da linha 06 cria um ponteiro para a estrutura ***Ponto*** de nome ***pp*** e atribui a este o endereço de ***p***. O resultado está abstraído na figura abaixo (***pp*** aponta para ***p***):



Após uma sequencia de processamento que não importam ao contexto (generalizado pela linha de código 07), o código da linha 08 instancia (por meio do comando ***malloc*** – abreviatura de *memory allocation* – alocar memória) nova área de memória e atribui endereço dessa nova área para a variavel dinâmica ***pp***. As instruções 09 e 10 atribuem valores para os campos ***x*** e ***y*** da variável dinâmica apontada por ***pp***. Observe na figura abaixo que a variável estática ***p*** continua existindo, mas ***pp*** não referencia mais ***p*** e sim a nova área criada pelo comando ***malloc***.

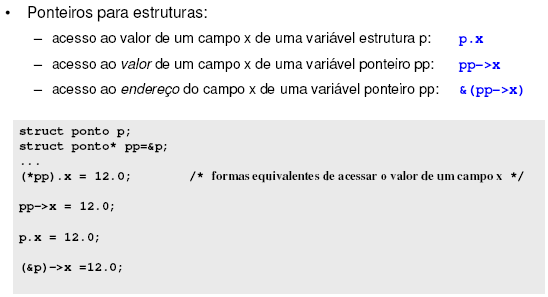


Melhor detalhando a instrução da linha08:

08: pp = (pp)**malloc**(**sizeof**(**struct** ponto));

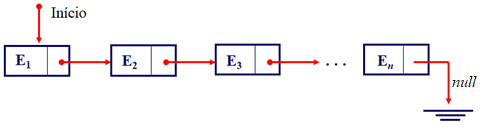
Na linguagem C, a função ***malloc*** é utilizada para alocar dinamicamente (em tempo de execução) memória. Esta função aloca o tamanho de memória recebido como parâmetro. Neste caso, o tamanho é definido pela valor retornado pela função ***sizeof***(***struct ponto***) – ***sizeof*** é uma função que retorna o tamanho, em bytes, da estrutura recebida como parâmetro, no caso o tamanho de ***struct ponto***. Esta área é formatada conforme a estrutura do ponteiro passado entre os parênteses que precedem o ***malloc***, ou seja, formatado conforme a estrutura de ***(pp)***. Por fim, ***malloc*** retorna o endereço da área que ele alocou e este endereço é atribuído à variável dinamica ***pp***.[[1]](#footnote-1)

A sintaxe em C para acessar campos de estruturas por meio de variáveis estáticas e de variáveis dinâmicas (ponteiros):



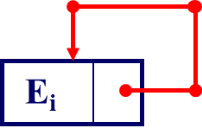
**Estruturas encadeadas**

Mas como alocar dinamicamente área de memória para armazenar uma coleção de dados de forma que estes se mantenham interligados, como ocorre com *arranjos*? Bem, a solução é estabelecer um encadeamento entre os elementos por meio de referências de memória, como ilustrado na figura a seguir:



Pela figura, cada elemento de dados fica armazenado em uma área de memória, que contém uma referência (ou ponteiro) para a área de memória que contém o próximo elemento. A referência do último elemento é ***null*** (representado na figura pelo símbolo de aterramento da engenharia elétrica), visto que não há um próximo. Assim basta ter uma referência para o primeiro elemento da coleção, e os demais estarão “encadeados” por referências de memória (ou ***ponteiros***).

Nessa abordagem de alocação dinâmica de memória com encadeamento, um conceito essencial é o de ***célula***, detalhado a seguir:



A ***célula***, conforme ilustrado acima, é uma estrutura de memória com duas áreas bem distintas:

* A primeira área é reservada para armazenar o elemento de dados que deseja-se manipular, representado na ilustração pelo elemento **Ei**. Este elemento varia de acordo com a aplicação.
* A segunda área é reservada para armazenar uma ***auto referência***, isto é, um ponteiro para a prória célula (representado na ilustração pela seta vermelha). Por meio desta referência, viabiliza-se o encadeamento das células, formando uma coleção interrelacionada.

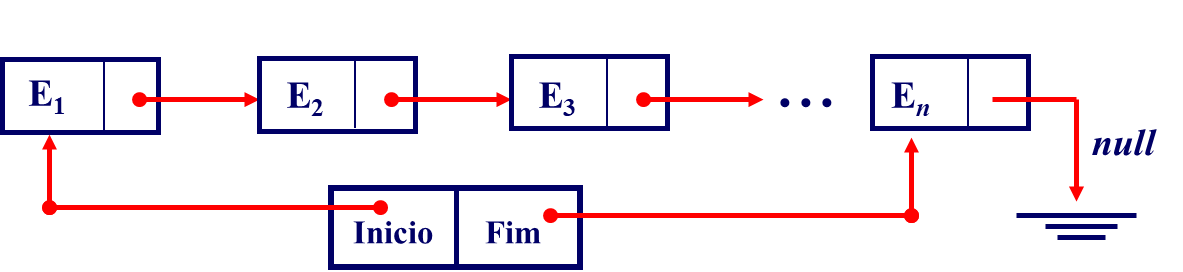


Algumas característisca da aborgem de alocação dinâmica com encadeamento:

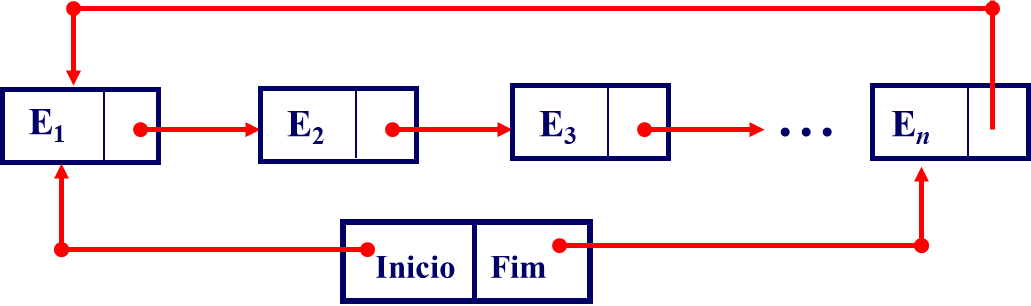
* Permite utilizar posições não contíguas de memória;
* Não há limite para o tamanho da lista e nem desperdício de memória alocada mas não utilizada;
* Permite inserir e retirar elementos sem precisar deslocar os itens seguintes da coleção;
* Paga-se o preço de utilizar espaço adicional para os ponteiros;
* Também não há possibilidade de acesso direto aos dados, o acesso só é possível sequencialmente, a partir do ponteiro da célula inicial.

**Principais formas de estruturas Encadeadas**

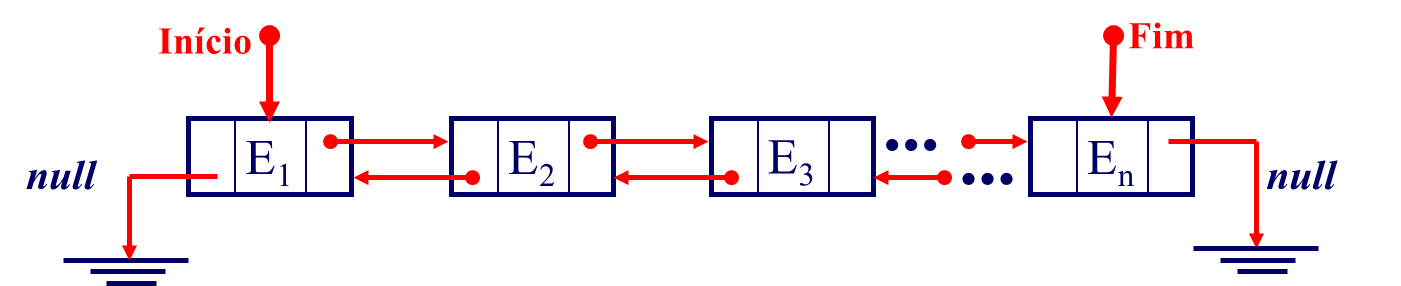
Estrutura ***simplemente encadeada*** (é sempre conveniente manter uma referênca para a o início e o fim da estrutura, como ilustrado):



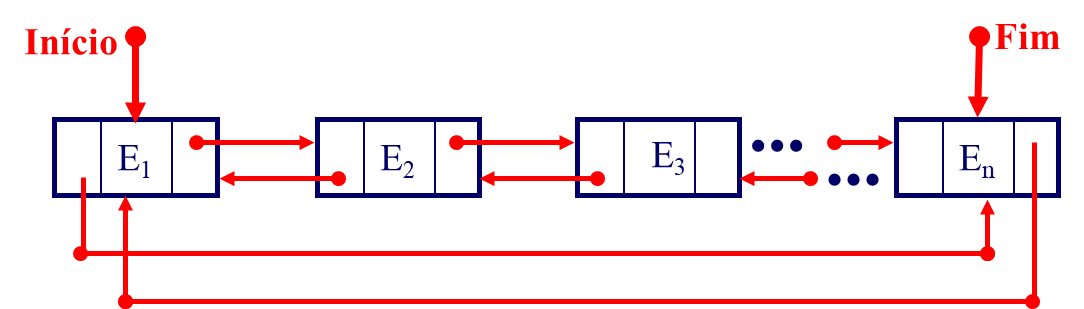
Estrutura ***simplesmente encadeada*** ***circular*** (último elemento “aponta” para o primeiro):



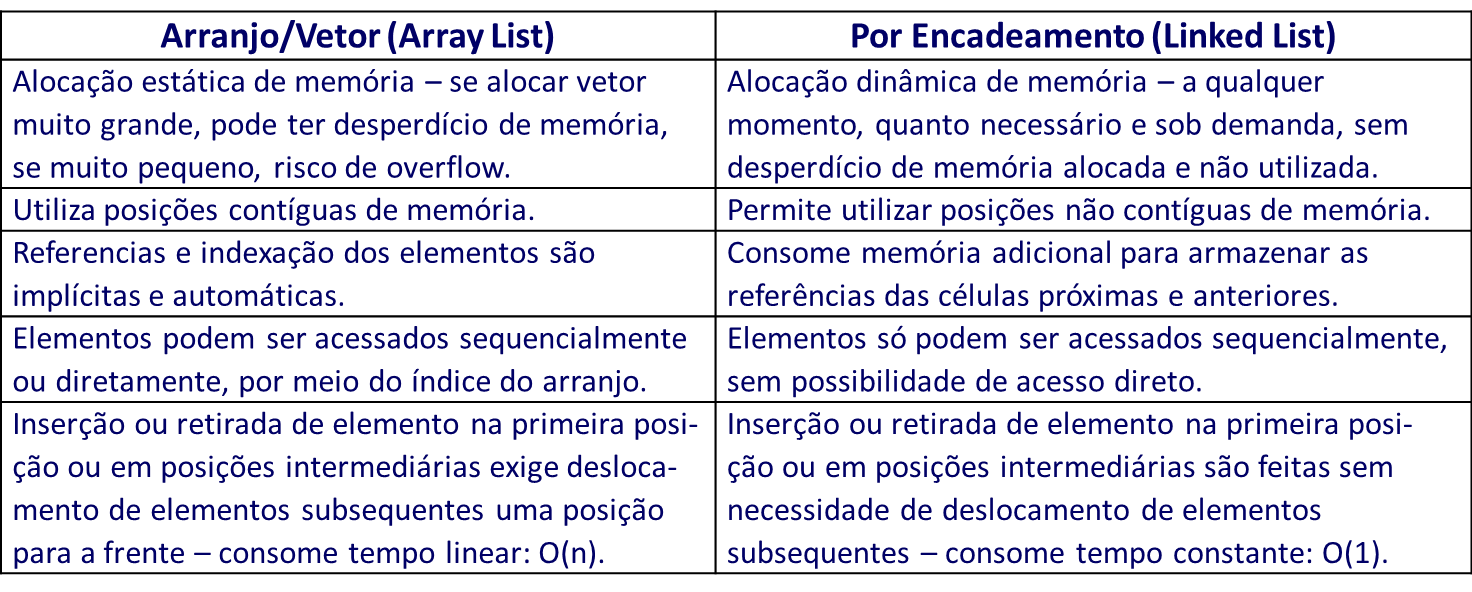
Estrutura ***duplamente encadeada*** (cada célula possui um ponteiro para o sucessor e outro para o antecessor):



Estrutura ***duplamente encadeada circular***:

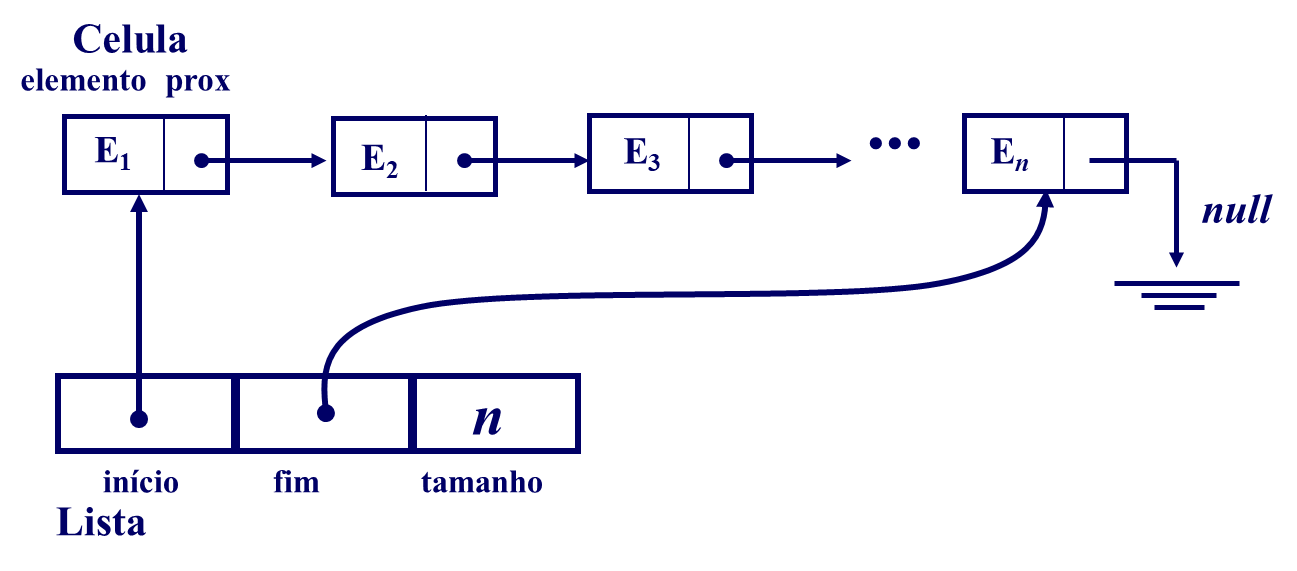


Comparativo entre as abordagens de gerência de memória: Arranjo X Encadeamento:



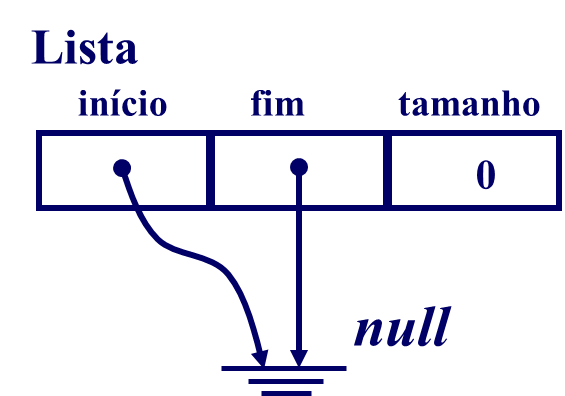
**Implementação do *TAD Lista* por *Encadeamento***

Assim como foi feito para suportar uma *Lista* implementada com ***arranjo***, será desenvolvido um ***TAD Lista*** utilizando ***alocação dinâmica de memória e encadeamento***. Os elementos da lista são encadeados via apontadores, conforme a figura a seguir.



Observe pela figura que há duas estruturas de dados envolvidas: a *primeira*, denominada ***Celula***, com os campos ***elemento*** e ***prox***, onde ***elemento*** é uma ***struct*** que descreve o elemento de dado a ser tratado na lista, enquanto que ***prox*** é um *auto relacionamento*, ou seja, um ponteiro para a própria estrutura ***Celula***. A *segunda* estrutura é a ***Lista***, propriamente dita. Esta é composta por dois ponteiros para ***Celula***, um para o ***inicio*** e outro para o ***fim*** da *lista*, e um campo para manter o ***tamanho*** dinâmico da lista.

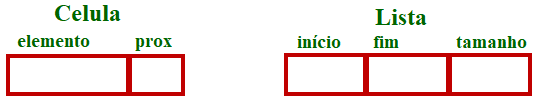
Sendo assim, uma *lista vazia* é abstraída quando os ponteiros para inicio e fim são *null* e o tamanho é igual a 0, conforme ilustrado a seguir. Numa *lista vazia*, não há (ainda) *células*:



Estas estruturas podem ser mapeadas para uma linguagem computacional como[[2]](#footnote-2):

|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| Elemento: **Registro**  ***// campos da estrutura de dados da Lista***  **FimRegistro**  Celula: **Registro**  elemento: Elemento  prox: ↑Celula  **FimRegistro**  Lista: **Registro**  inicio, fim: ↑Celula  tamanho: Inteiro  **FimRegistro** | **typedef struct {**  ***// campos da estrutura de dados da Lista***  **} Elemento;**  **typedef struct** Cel \* **Ponteiro;**  **typedef struct** Cel **{**  **Elemento** elemento;  **Ponteiro** prox;  **} Celula;**  **typedef struct {**  **Ponteiro** inicio, fim;  **int** tamanho;  **} Lista** |

Com a estrutura definida acima tem-se as abstrações de ***Celula*** e ***Lista*** na figura abaixo:



**Operações do *TAD Lista***

Nesta implementação do ***TAD Lista*** por ***encadeamento*** serão consideradas as mesmas operações desenvolvidas na implementação do *TAD Lista* por *arranjo*. A seguir o desenvolvimento de cada operação:

* criaLista (Lista \*lista);
* vazia (Lista \*lista);
* insereFinal (Elemento x, Lista \*lista);
* buscaLista(Elemento x, Lista \*lst);
* excluiLista(Elemento x, Lista \*lista).

**Implementação da operação Cria Lista Vazia**: Esta operação consiste apenas em receber como parâmetro uma variável do tipo *Lista* e iniciar os ponteiros ***inicio*** e ***fim*** com *null* e atribuir 0 ao campo ***tamanho***.

|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Procedimento** criaLista (**var** lista: Lista)  **Inicio**  lista↑inicio 🡨 null  lista↑fim 🡨 null  lista↑tamanho 🡨 0  **FimProcedimento** | **void** criaLista (**Lista** \*lista)  **{**  lst->inicio = NULL;  lst->fim = lst->inicio;  lst->tam = 0;  **}** |

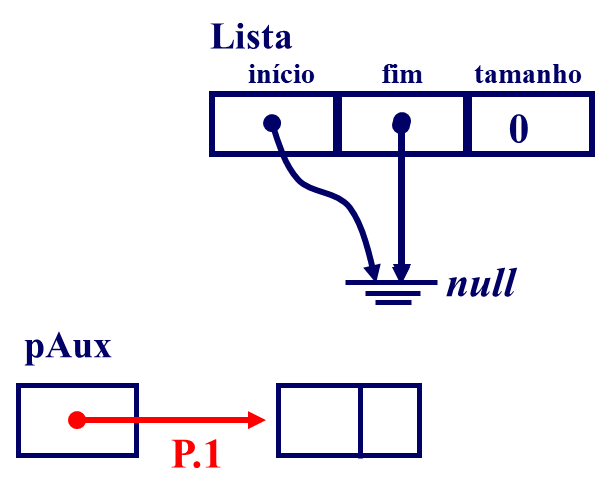
**Implementação da operação Testa Lista Vazia**: Esta operação , *booleana*, consiste apenas em testar se o *ponteiro* ***inicio*** é *null*.

|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Funcao** vazia (**var** lista: Lista): boolean  **Inicio**  **Retorne** (lista**↑**inicio = null)  **FimFuncao** | **int** vazia (**Lista** \*lista)  **{**  **return** (lista->inicio == NULL);  **}** |

**Implementação da operação de Inserção – Insere no Final**: Esta operação consiste em inserir elemento, recebido como parâmetro, no final da lista, como ilustrado a seguir:

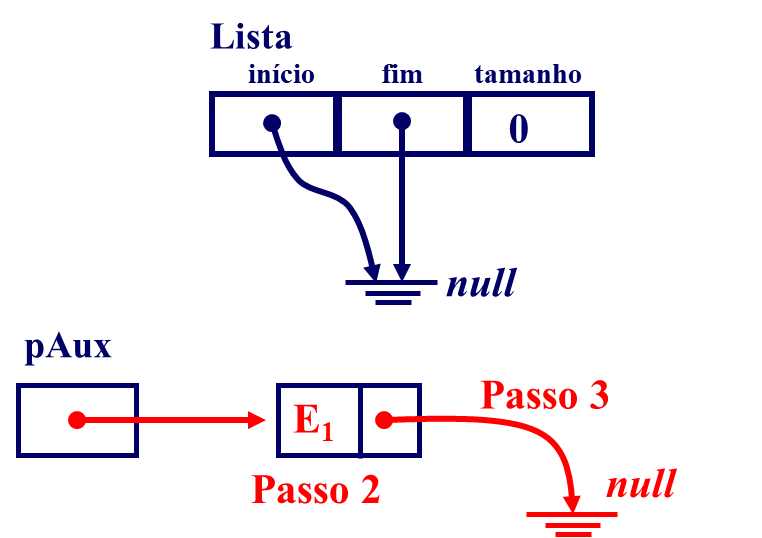
Inserção na *lista* inicialmente vazia:

Passo 1: aloca memória para uma célula e guarda endereço em ***pAux***;



Passo 2: coloca o elemento de dados na célula criada;

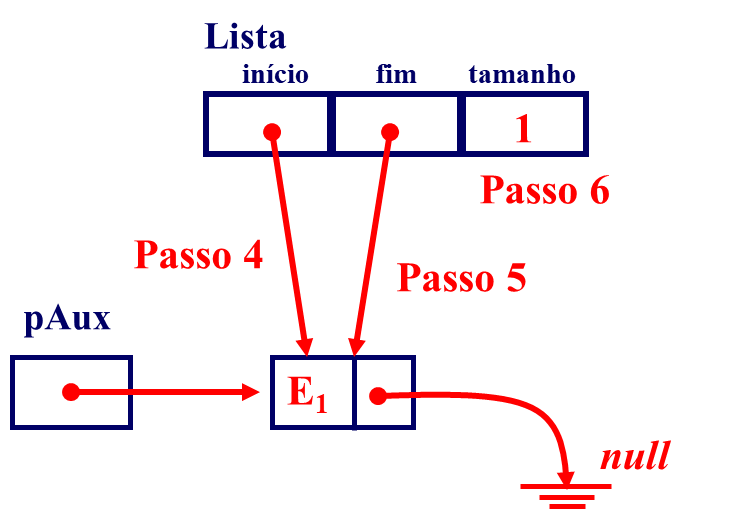
Passo 3: atribui *NULL* ao campo ***prox*** da célula criada;



Passo 4: ajusta o ponteiro ***inicio***;

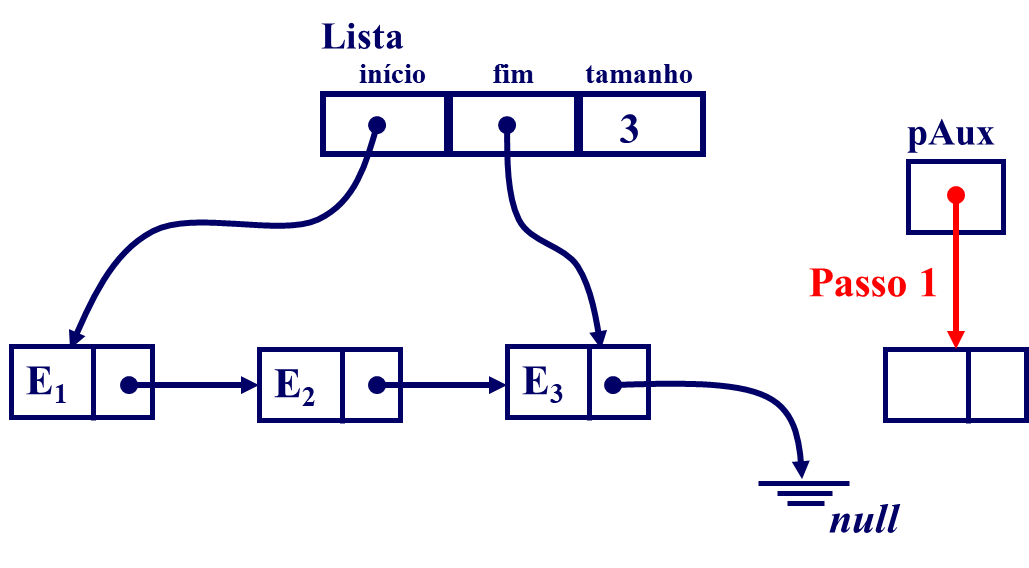
Passo 5: ajusta o ponteiro ***fim***;

Passo 6: ajusta o campo ***tamanho***.



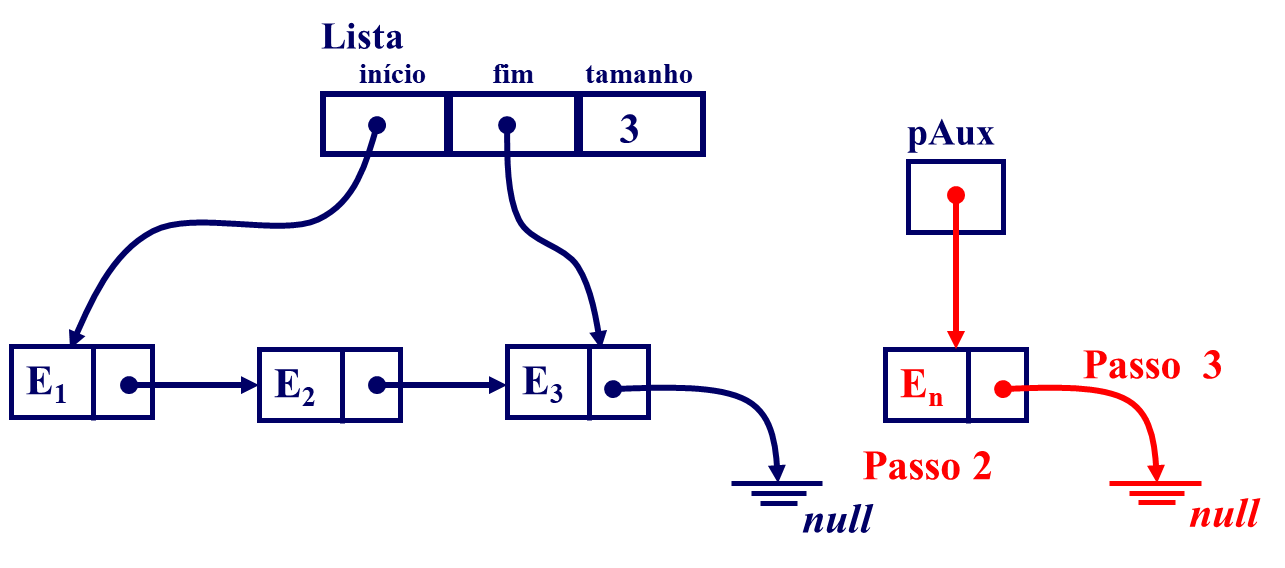
Inserção na *lista* não vazia (inserção do n-ésimo elemento, n > 1):

Passo 1: aloca memória para uma célula e guarda endereço em ***pAux***;



Passo 2: coloca o elemento de dados na célula criada;

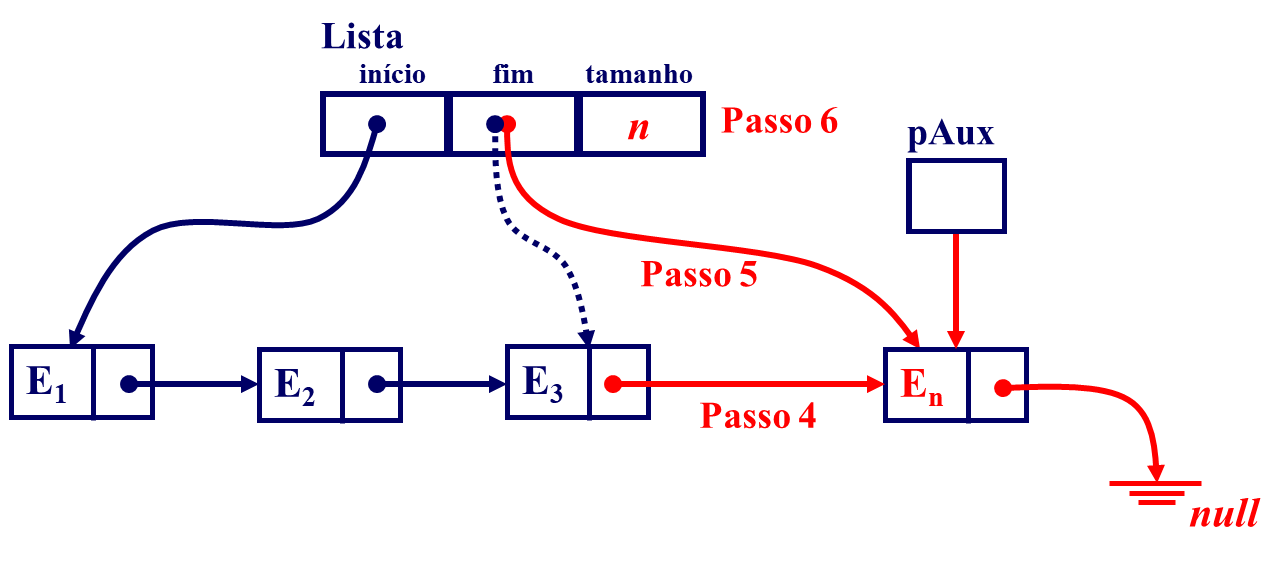
Passo 3: atribui *NULL* ao campo ***prox*** da célula criada;



Passo 4: ajusta o *ponteiro* da última célula (***fim->prox***);

Passo 5: ajusta o *ponteiro* ***fim***;

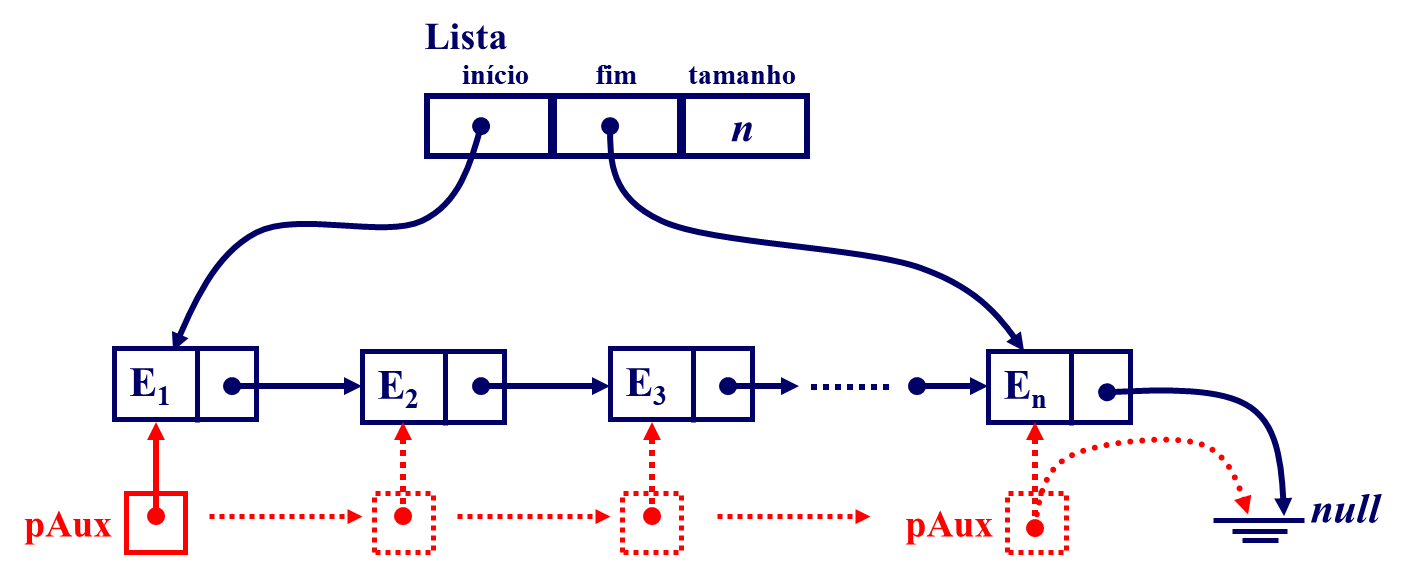
Passo 6: ajusta o *campo* ***tamanho***.



Observa-se que a distinção entre os algoritmos de inserção com a *lista* vazia e não vazia está apenas no passo 4, onde no primeiro caso ajusta-se o ponteiro inicio da lista e no segundo, ajusta-se o ponteiro prox da última célula. Deste modo, pode-se juntar os dois algoritmos, deixando o passo 4 condicional, conforme descrito a seguir.

|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Procedimento** insereFinal(x:Elemento, **var** lst:Lista)  **Var** pAux: ↑Celula  **Inicio**  pAux 🡨 Aloc (↑Celula)  pAux↑elemento 🡨 x  pAux↑prox 🡨 null  **Se** (vazia(lst)) **então**  lst↑inicio 🡨 pAux  **Senao**  lst↑fim↑prox 🡨 pAux  **FimSe**  lst↑fim 🡨 pAux  lst↑tamanho 🡨 lst↑tamanho  **FimProcedimento** | void insereFinal (Elemento x, Lista \*lista)  {  Ponteiro pAux;  pAux = (Ponteiro)malloc(sizeof (Celula)); ***// Passo1***  pAux->elemento = x; ***// Passo2***  pAux->prox = NULL; ***// Passo3***  if (vazia(lst)) ***// Passo4 - condicional***  lst->inicio = pAux;  else  lst->fim->prox = pAux;  lst->fim = pAux; ***// Passo5***  lst->tamanho++; ***// Passo6***  } |

**Implementação da operação de Busca**: Esta operação consiste em caminhar em toda Lista pesquisando por um elemento informado pelo parâmetro x. A pesquisa é implementada comparando a chave de pesquisa com a chave do registro apontado por ***pAux***. Resultado da pesquisa pode ser *Sucesso* (encontrou) ou *Fracasso* (não encontrou).



Observe que a operação ***buscaLista***( ) na abordagem implementada com *arranjo*, retornava o *índice* do elemento buscado no vetor ou **–1** no caso de não encontrar o elemento. Como nessa implementação por *encadeamento*, as *células* não possuem *índices*, será retornado a referência (endereço) da célula com o elemento buscado, ou ***null***, no caso de não encontrar o elemento. No caso de lista vazia, a primeira atribuição faz ***pAux = null*** e neste caso não ocorrerá a iteração e ***null*** é retornado pela função.

|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Funcao** buscaLista(x:Elemento, **var** l:Lista): ↑Celula  **Var** pAux: ↑Celula  **Inicio**  pAux 🡨 l↑inicio  **Enquanto** (pAux <> null) **faça**  **Se** ( pAux↑elemento↑chave = x.chave) **então**  **Retorne** (pAux) ***// encontrou!***  **Senao**  pAux 🡨 pAux↑prox  **FimSe**  **FimEnquanto**  **Retorne** (pAux) ***// não encontrou!***  **FimFuncao** | **Ponteiro** buscaLista(**Elemento** x, **Lista** \*lst)  **{**  Ponteiro pAux = lst->inicio;  while (pAux != NULL) {  if (pAux->elemento->*chave* == x.*chave*)  return (pAux); ***// Encontrou elemento***  else  pAux = pAux->prox;  }  return(pAux); ***// Não ncontrou elemento***  **}** |

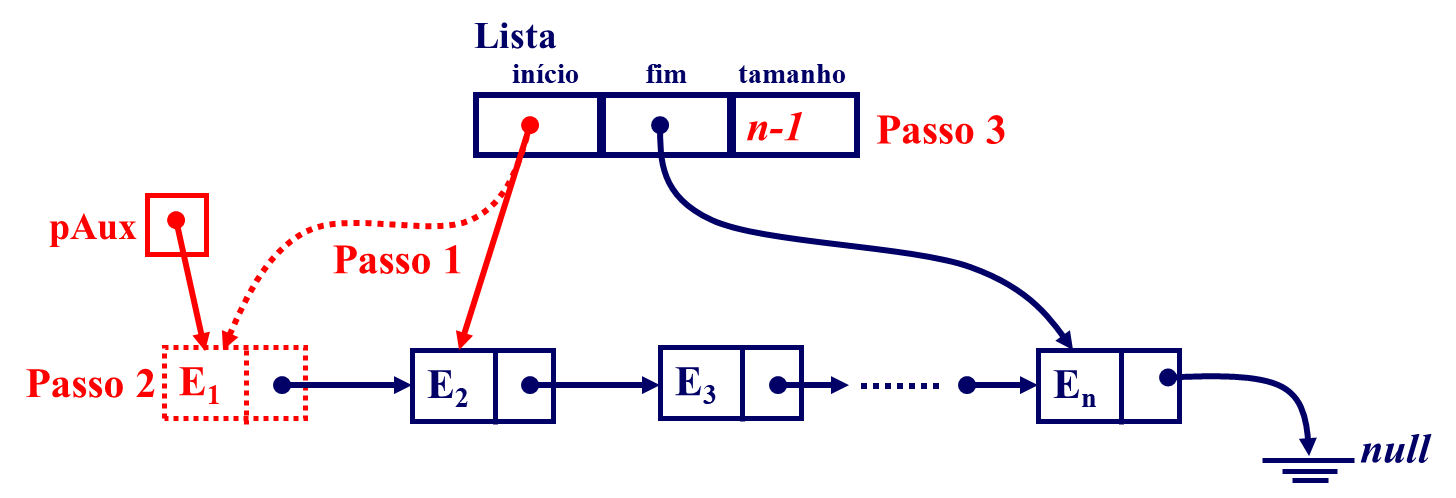
**Implementação da operação de Exclusão**: Consiste em localizar o item a ser excluído, retirá-lo da lista, readequando os ponteiros de forma a manter íntegro o encadeamento da estrutura. A localização do elemento a ser excluído é feita pela operação ***buscaLista***( ) definida acima. A seguir é ilustrado o processo de exclusão já considerando o elemento localizado e apontado pelo ponteiro auxiliar ***pAux***.

Exclusão do **Primeiro Elemento** da Lista, indicado pelo ponteiro ***pAux***:

Passo 1: Ponteiro ***inicio*** aponta para o próximo (***inicio = inicio->prox***);

Passo 2: Libera memória da célula excluída (***free(pAux)***);

Passo 3: Ajusta tamanho da lista (***lst->tamanho--***);



Exclusão do ***outros elementos exceto o primeiro*** da Lista, indicado pelo ponteiro ***pAux*** retornado por ***buscaLista***( ), Neste caso, é necessário ter referencia da célula anterior a que será excluída, para correto ajuste dos ponteiros, conforme ilustração a seguir onde é simulada a exclusão do elemento ***E3***):

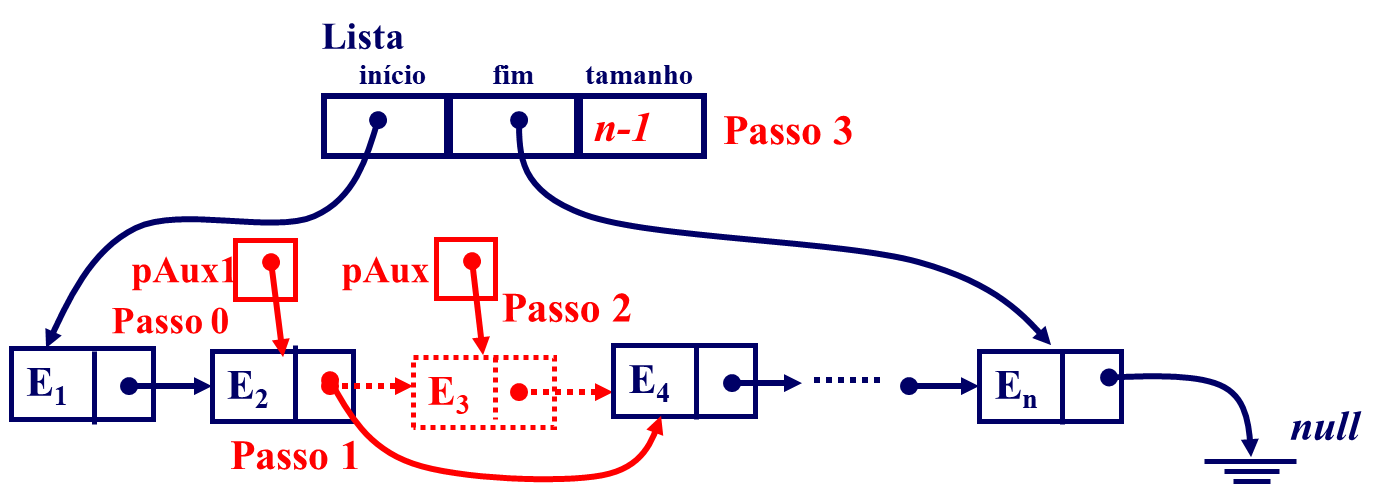
Passo 0: Encontra a célula anterior a pAux (***pAux1->prox = pAux***);

Passo 1: Ponteiro da célula anterior aponta para próxima célula após a célula a ser excluída

(***pAux1->prox = pAux->prox***);

Passo 2: Libera memória da célula excluída (***free(pAaux)***);

Passo 3: Ajusta tamanho da lista (***lst->tamanho--***);



|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Funcao** excluiLista (x: Dados, **var** l: Lista): inteiro  **Var** pAux, pAux1: ↑Celula  **Inicio**  pAux 🡨 buscaLista(x, l);  **Se** ( pAux = null) **então**  **Retorne** (999) *//Elemento não encontrado*  **FimSe**  **Se** ( pAux = l↑inicio ) **então**  l↑inicio 🡨 l↑inicio↑prox *//Passo 1*  **Se** (l↑inicio = null) **então**  l↑fim 🡨 null  **FimSe**  **Senão**  pAux1 🡨 l↑inicio *//Passo 0*  **Enquanto** (pAux↑prox <> pAux) **Faça**  pAux1 🡨 pAux1↑prox  **FimEnquanto**  pAux1↑prox 🡨 pAux↑prox *//Passo 1*  **Se** (l↑fim = pAuxl) **então**  l↑fim 🡨 pAux1  **FimSe**  **FimSe**  LiberaMemoria (pAux) *//Passo 2*  l↑tamanho 🡨 l↑tamanho + 1 *//Passo 3*  **Retorne** (0) *// Operação realizada com sucesso!*  **FimFuncao** | **int exclui**Lista(**Elemento** x, **Lista** \*lista)  **{**  **Ponteiro** pAux1, Paux;  pAux = buscaLista(x, lista);  if(pAux == NULL)  return 999; ***// \* Elemento não encontrado***  if(pAux == lst->inicio){***//Exclusão 1º Elemento***  lst->inicio = lst->inicio->prox; ***// Passo1***  **if**(lst->inicio==NULL) lst->fim=NULL;  ***//Ajusta lst->fim se lista se tornar vazia***  }  else { ***// Exclusão de elemento que não 1º***  pAux1 = lista.inicio; ***// Passo 0***  while(pAux1->prox != pAux)  pAux1 = pAux1->prox;  pAux1->prox = pAux->prox; ***// Passo 1***  **if**(lst->fim == pAux) lst->fim = pAux1;  ***//Ajusta lst->fim******for exclusão do último***  }  free (pAux); ***// Passo 2***  lista->tamanho--; ***// Passo3***  return 0; ***// Exclusão com sucesso***  **}** |

1. Pesquise outras instruções da linguagem C que tratam alocaçã dinâmica de memória: ***free***( ); ***calloc***( ); ***realloc***( ); [↑](#footnote-ref-1)
2. O ***Visualg*** não suporta ponteiros, entretanto para efeitos didáticos, será adotada a seguinte sintaxe para representar ponteiros em Português Estruturado:

   **var: ↑Tipovar** : declara uma *variável dinâmica* ponteiro – exemplo: **pInt: ↑Inteiro** – neste caso, **pInt** é um *ponteiro* para *inteiro*. outro exemplo: **Lista: ↑lista** cria um *ponteiro* para uma estrutura ***Lista***.

   **Aloc(↑pntr)**: aloca memória no tamanho e formato do ponteiro ***pntr*** e retorna endereço da memória alocada – exemplo **lista 🡨 Aloc(↑Lista) –** aloca memória no formato e tamanho de *Lista* e atribui endereço a ***lista***.

   **var↑campo** : define o acesso a um *campo* de uma *variável dinâmica* estruturada – exemplo: **lista↑inicio**  é o acesso ao *campo* ***inicio*** da *variável dinâmica* ***lista***. [↑](#footnote-ref-2)