CENTRO DE CIÊNCIA E TECNOLOGIA
LABORATÓRIO DE CIÊNCIAS MATEMÁTICAS
UNIVERSIDADE ESTADUAL DO NORTE FLUMINENSE

Listas Genéricas e Implementação TAD

Disciplina: Estrutura de Dados I

Prof. Fermín Alfredo Tang Montané

Curso: Ciência da Computação

Listas (Lists)

Definição

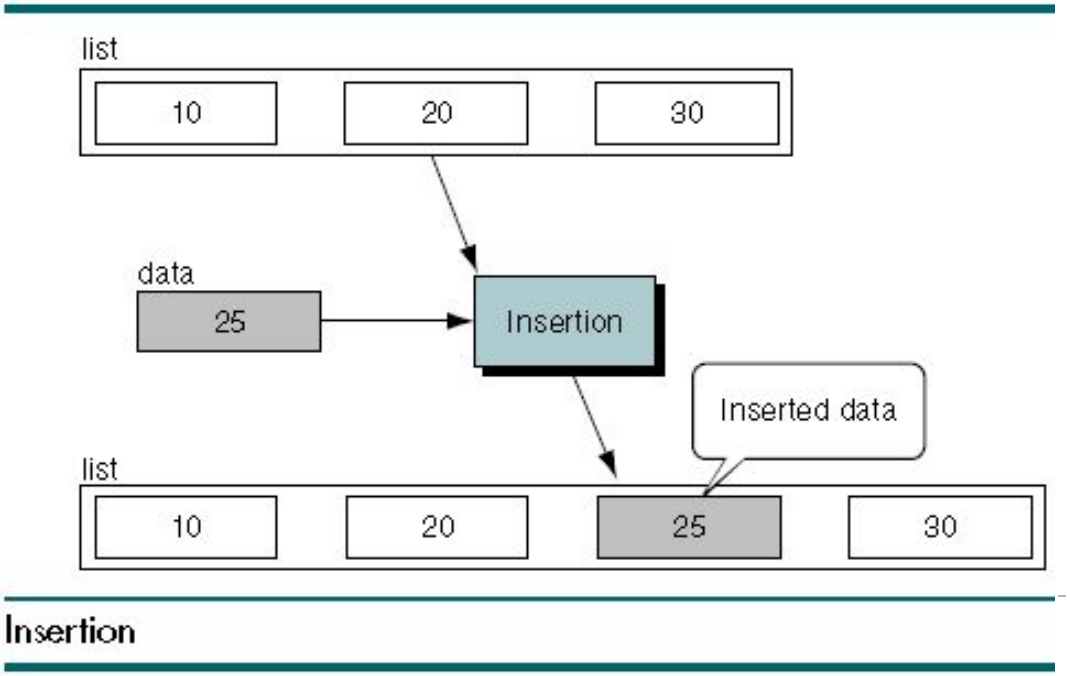
- Uma lista é uma estrutura linear na qual operações como **recuperações, inserções e remoções** podem ser realizadas em qualquer lugar da lista, seja **no começo, no meio ou no fim da lista**.
- Utilizamos diferentes tipos de listas no dia-a-dia.
- Usamos listas de empregados, listas de estudantes ou listas de músicas preferidas. Quando processamos nossa lista de músicas favoritas, devemos ser capazes de procurar por uma música, adicionar uma nova música ou remover uma que não gostamos mais.
- Parte-se do suposto que a lista é uma **lista ordenada**. Uma sequência ordenada com base no seus dados ou com base em uma **chave** que identifica os dados.
- Existe também o conceito de **lista aleatória** (**random list**), em que não existe uma relação sequencial entre os dados. Neste caso as inserções são sempre realizadas no final da lista, **lista cronológica**.

Listas (Lists)
Operações Básicas

- Existem quatro operações básicas de listas:
  - Inserir (addNode)
  - Remover (removeNode)
  - Recuperar (retrieveNode)
  - Posicionar (traverse)

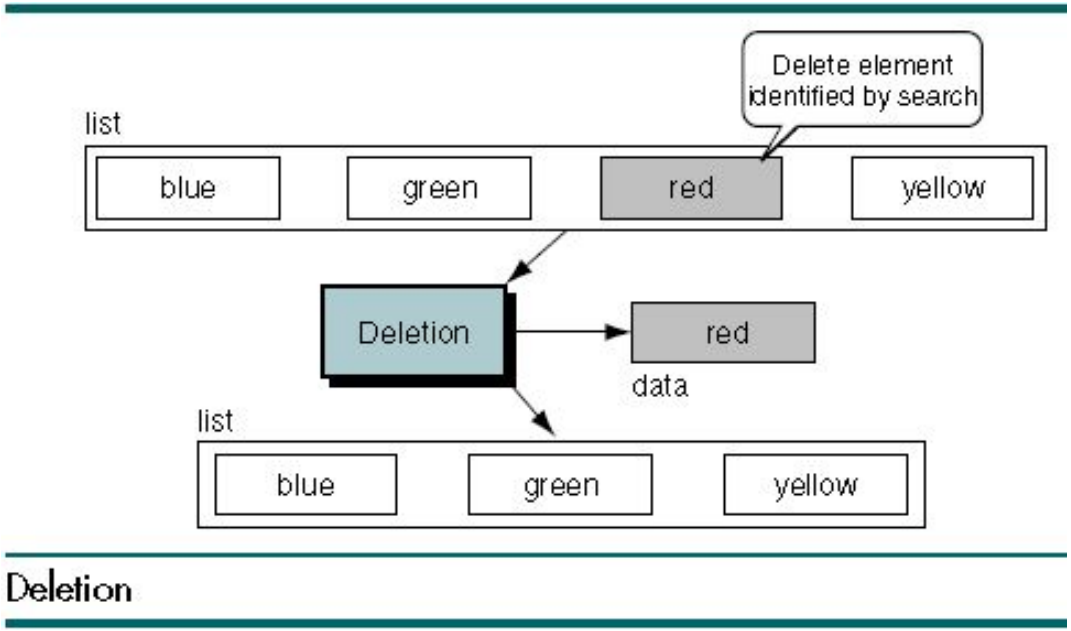
Listas (Lists)
Operação Inserir (addNode)

- A operação Inserir deve inserir elementos em uma lista ordenada de maneira que a ordenação seja preservada.
- Preservar a ordem pode exigir que a inserção seja realizada no começo ou no final da lista, embora na maioria dos casos a inserção deva ser realizada em algum lugar no meio da lista.
- Para determinar onde inserir um elemento é preciso executar um algoritmo de busca.



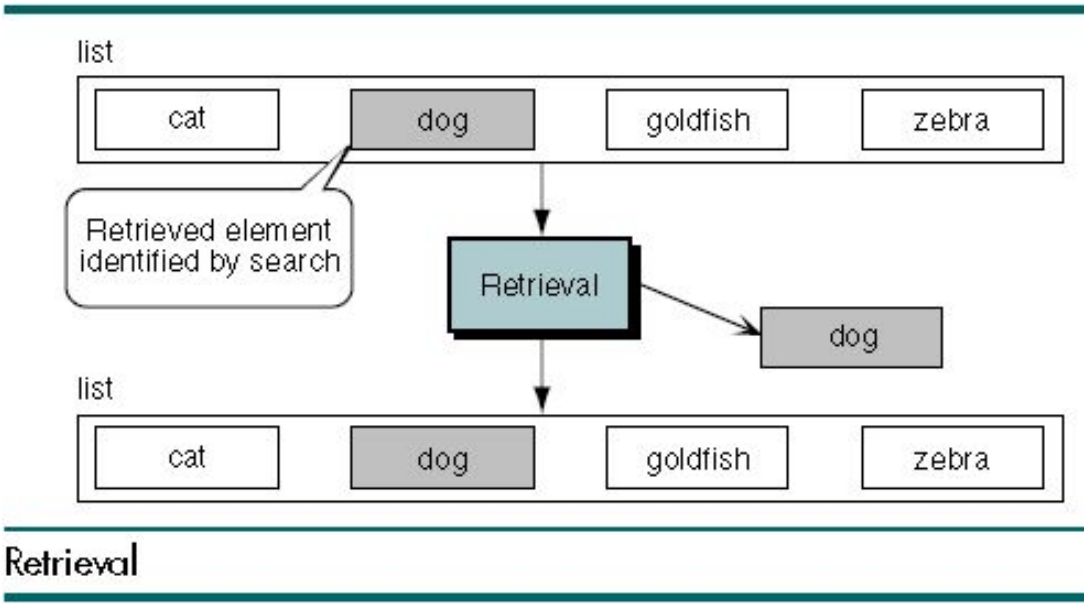
Listas (Lists)
Operação Remover (removeNode)

- A operação de remoção de uma lista requer que a lista seja pesquisada para localizar o dado que esta sendo removido.
- Uma vez localizado, o dado é removido da lista.



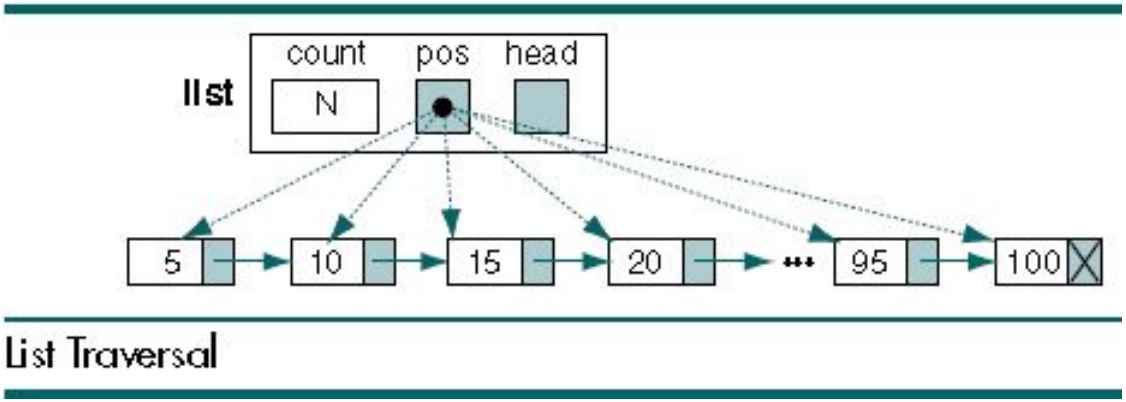
Listas (Lists)
Operação Recuperar (retrieveNode)

- A operação de recuperação de um nó da lista requer que a lista seja pesquisada para localizar o dado que esta sendo recuperado.
- Uma vez localizado, o dado é repassado para o modulo que fez a chamada **sem mudar o conteúdo da lista**.



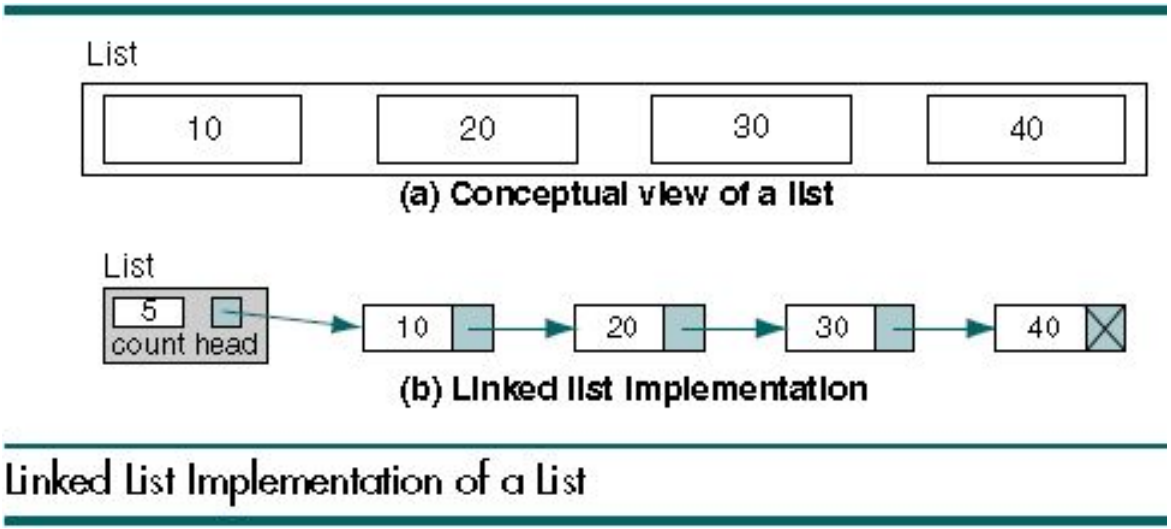
Listas (Lists)
Operação Posicionar (traverse)

- A operação posicionar utiliza um ponteiro para registrar a posição do último nó visitado na lista entre uma operação e outra.



Listas (Lists)
Implementação como Listas Encadeadas

- Estrutura conceitual e Estrutura Física como Lista Encadeada.



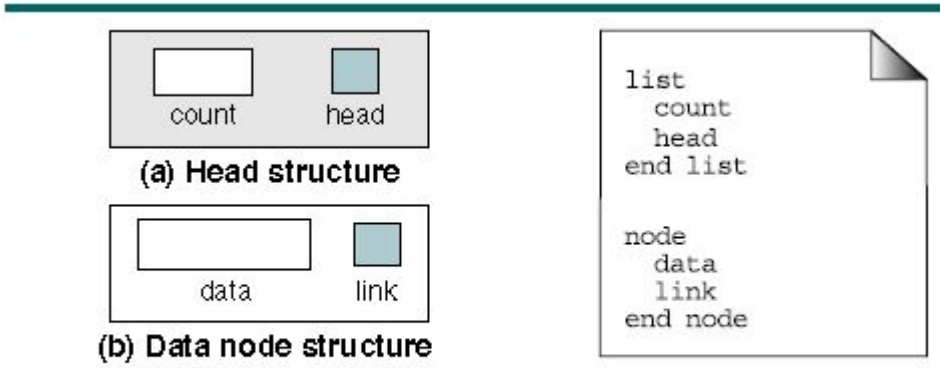
Linked List Implementation of a List

- Precisamos de duas estruturas diferentes para implementar uma lista:
  - Uma estrutura de cabeçalho da lista (Head);
  - Uma Estrutura de nó da lista (Node).

Listas (Lists)

Implementação como Listas Encadeadas - Estruturas

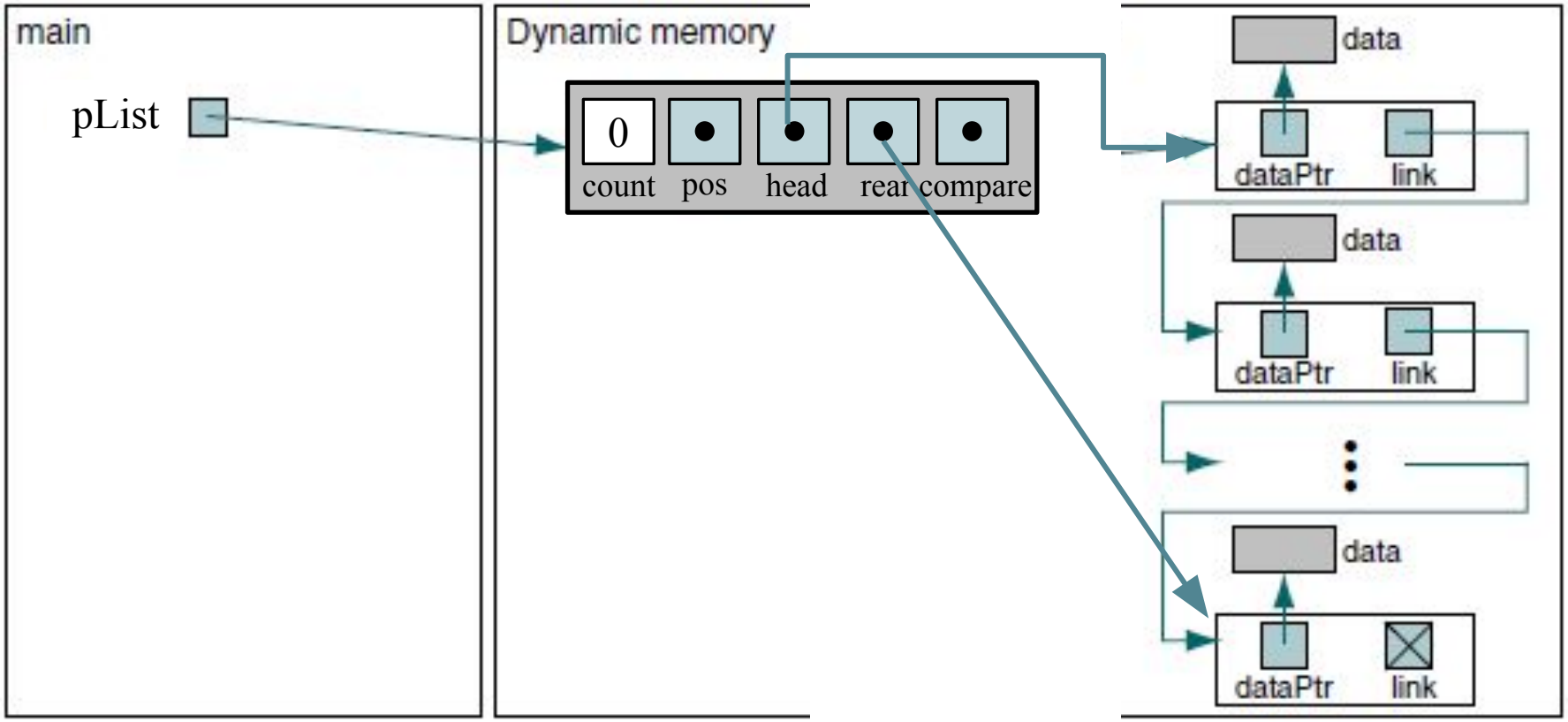
- **Cabeçalho da Lista (Head).-** Embora somente seja necessário um ponteiro para identificar a lista, é conveniente criar um nó cabeçalho para armazenar dados referentes a lista. Além do ponteiro ao começo da lista podemos ter um ponteiro ao final, outro para posição corrente, um contador, entre outros (p.e maior elemento, menor elemento).
- **Nó da Lista (Node).-** contém os dados do usuário (ou um ponteiro para eles) e um ponteiro para outro nó da lista. Os dados armazenados dependem completamente da aplicação.
- Tanto os nós como o cabeçalho são armazenados em memória dinâmica.



Head Node and Data Node

TAD Lista (List ADT)

Implementação como Listas Encadeadas



TAD Lista (Lista ADT)

Estrutura

- Em vez de armazenar um dado em cada nó, se armazena um ponteiro a esse dado.
- O programa de aplicação terá a responsabilidade de alocar memória para o dado e de passar o seu endereço ao TAD lista.

TAD Lista (List ADT)

Estrutura da Lista

- As definições de tipos para a lista é mostrada no código:

P5-01.h - Estruturas

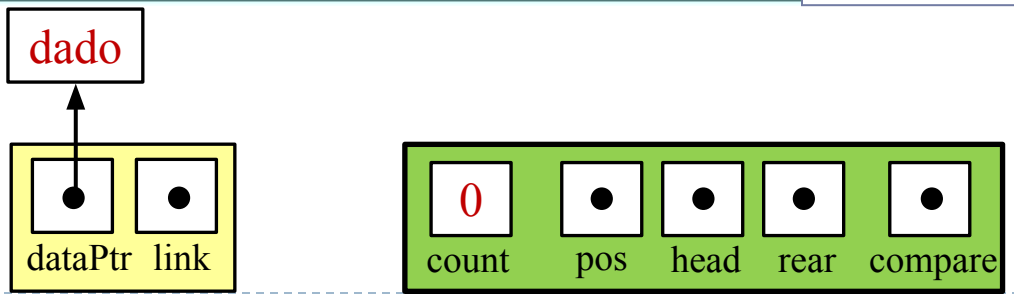
```
1  //List ADT Type Defintions
2    typedef struct node
3       {
4        void*        dataPtr;
5        struct node* link;
6       } NODE;
7
8    typedef struct
9       {
10       int    count;
11       NODE* pos;
12       NODE* head;
13       NODE* rear;
14       int    (*compare) (void* argu1, void* argu2)
15      } LIST;
```

```
// Definição do tipo nó da lista (NODE)

//  Ponteiro genérico
//  Ponteiro de ligação

//  Definição do tipo (cabeçalho) lista (LIST)

// Contador de elementos
// Ponteiro ao nó corrente da lista
// Ponteiro ao primeiro nó da lista
// Ponteiro ao último nó da lista
// Ponteiro a uma função de comparação
escrita pelo programa de aplicação
```



TAD Lista (List ADT)

Estrutura da Lista

- Junto as definições de tipos devemos incluir os protótipos das funções.
- Estas definições devem ser incluídas em um arquivo cabeçalho (header file) de maneira que qualquer aplicação que precise definir uma lista possa fazê-lo facilmente.

P5-02.h – Protótipos das Funções

```
1  //Prototype Declarations
2    LIST* createList   (int (*compare)
3                        (void* argu1, void* argu2));
4    LIST* destroyList  (LIST* list);
5
6    int   addNode   (LIST* pList, void* dataInPtr);
7
8    bool  removeNode   (LIST*  pList,
9                        void*  keyPtr,
10                       void** dataOutPtr);
11
12   bool  searchList   (LIST*  pList,
13                       void*  pArgu,
14                       void** pDataOut);
15
16   bool  retrieveNode (LIST*  pList,
17                       void*  pArgu,
18                       void** dataOutPtr);
19
```

TAD Lista (List ADT)

Estrutura da Lista

- Além das funções usadas para manipular a lista são definidas três funções internas que não são de acesso público aos programas de aplicação, mas apenas para uso interno do TAD lista.
- As funções são definidas como **static**.

P5-03.h – Protótipos das Funções, continuação...

```
20    bool  traverse     (LIST*  pList,
21                        int    fromWhere,
22                        void** dataOutPtr);
23
24    int   listCount    (LIST*  pList);
25    bool  emptyList    (LIST*  pList);
26    bool  fullList     (LIST*  pList);
27
28    static int _insert   (LIST* pList,
29                          NODE* pPre,
30                          void* dataInPtr);
31
32    static void _delete  (LIST*  pList,
33                          NODE*  pPre,
34                          NODE*  pLoc,
35                          void** dataOutPtr);
36    static bool _search  (LIST*  pList,
37                          NODE** pPre,
38                          NODE** pLoc,
39                          void*  pArgu);
40  //End of List ADT Definitions
```

TAD Lista (List ADT)

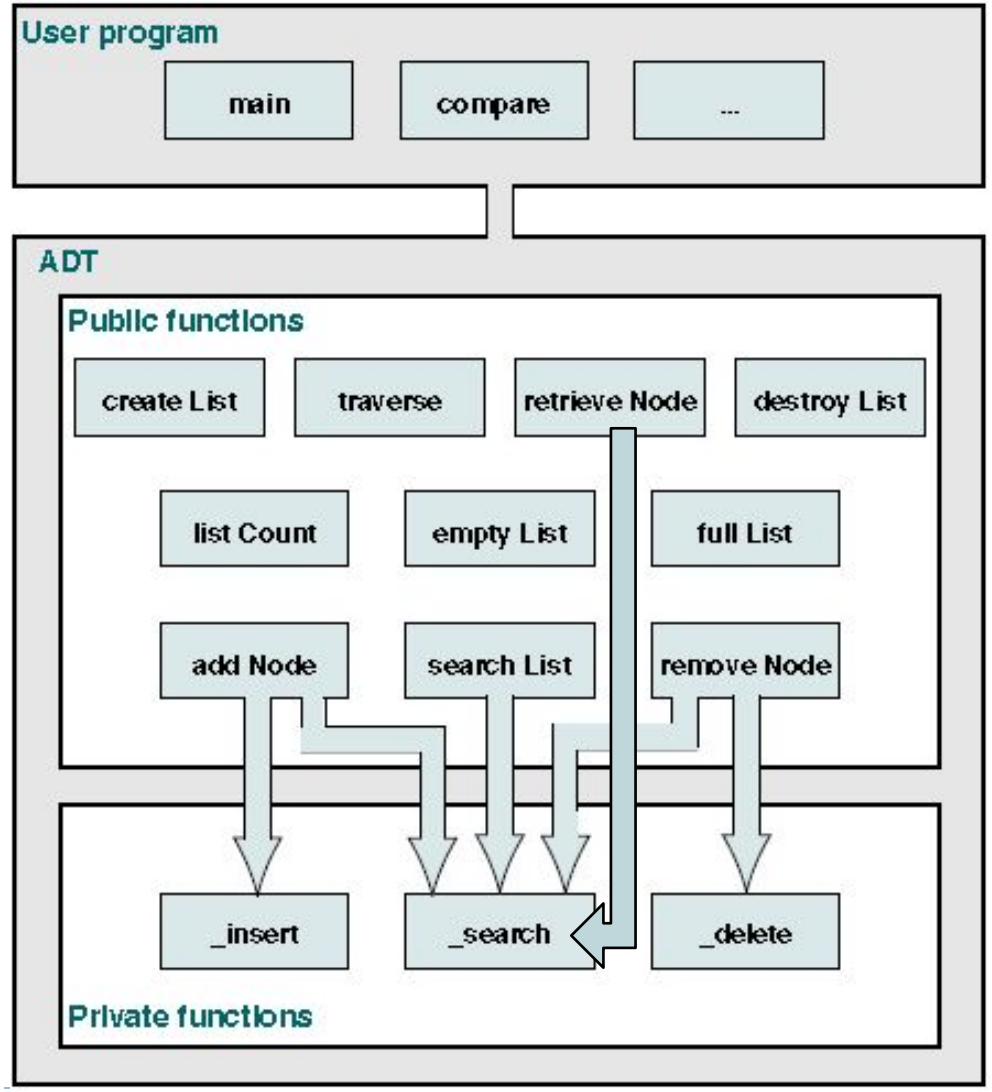
Estrutura da Lista

- Em C as funções são todas globais. A palavra reservada **static** antes do nome de uma função a torna estática.
- O acesso as funções estáticas fica restrito ao arquivo no qual elas foram declaradas. Assim, fazer as funções estáticas é uma maneira de restringir o acesso as funções.
- Outro motivo para fazer as funções estáticas é reutilizar o mesmo nome da função em outros arquivos.

TAD Lista (List ADT)

Estrutura da Lista

- A figura classifica as funções que fazem parte da TAD em duas categorias, as funções públicas que podem ser acessas pelo programa de aplicação e as funções internas ou privadas que somente podem ser acessadas pelo TAD.
- A dependência entre funções publicas e privadas é mostrada mediante as setas.



TAD Lista (List ADT)

Operações de suporte do TAD Lista

- Devemos ter operações de suporte ao funcionamento da Lista.
- Serão apresentadas as seguintes operações:
  - Criar Lista (create List);
  - Adicionar No (add Node);
  - Remover No (remove Node)
  - Procurar Lista (search List);
  - Recuperar No (retrieve Node);
  - Posicionar (traverse);
  - Lista Vazia (empty List);
  - Lista Cheia (full List);
  - Contador Lista (list Count);
  - Destruir Lista (destroy List).
- Essas operações serão implementadas em C.

TAD Lista (List ADT)

Criar Lista (Create List)

- O código para a função Criar Lista (createList()) é o seguinte:
- Observe que a função recebe um ponteiro a função de comparação definida pelo usuário.

P5-03.h – Função pública Criar Lista

```
1  /*=============== createList ==============
2    Allocates dynamic memory for a list head
3    node and returns its address to caller
4       Pre    compare is address of compare function
5              used to compare two nodes.
6       Post   head has allocated or error returned
7       Return head node pointer or null if overflow
8  */
9  LIST* createList
10       (int (*compare) (void* argu1, void* argu2))
11 {
12 //Local Definitions
13   LIST* list;
14
15 //Statements
16   list = (LIST*) malloc (sizeof (LIST));
17   if (list)
18       {
19        list->head    = NULL;
20        list->pos     = NULL;
21        list->rear    = NULL;
22        list->count   = 0;
23        list->compare = compare;
24       } // if
25
26   return list;
27 } // createList
```

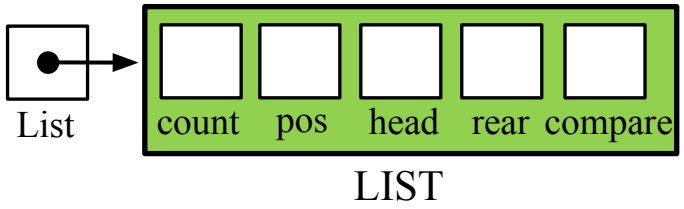
TAD Lista (List ADT)

Criar Lista (Create List)

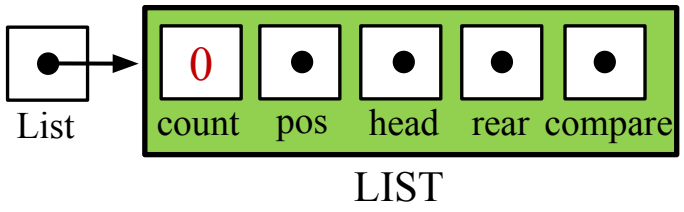
- Criar lista aloca memória para o nó cabeçalho da lista.
- Inicializa os ponteiros inicio, fim e pos com nulo e fixa o contador em zero.
- Em caso de overflow, na tentativa de alocação de memória, a função retorna um ponteiro nulo.
- As figuras ilustram este processo.

1 `LIST*  list;`

2 `list:=(LIST*)malloc(sizeof(LIST));`



3
```
list->head=NULL;
list->pos=NULL;
list->rear=NULL;
list->count=0;
list->compare=compare;
```



TAD Lista (List ADT)
Adicionar Nó (addNode)

- A função Adicionar Nó (addNode()) é uma função pública que recebe os dados a serem inseridos na lista e realiza uma busca na lista para encontrar o ponto de inserção. A busca é realizada por uma função privada (_search()).
- Caso a busca seja bem sucedida, outra função privada (_insert()) é chamada para realizar a inserção do novo dado na lista.
- Nesta implementação, evita-se a inserção de dados duplicados (ou dados com chaves duplicadas).
- A função Adicionar Nó (addNode()) pode retornar três valores possíveis:
  - -1: indica um overflow na memória dinâmica (não existe memória para inserir um novo elemento);
  - 0: indica sucesso na inserção;
  - +1: indica um elemento duplicado (ou com chave duplicada).

TAD Lista (List ADT)

Adicionar Nó (addNode)

- O código para a função AdicionarNó (addNode()) é o seguinte:
- A função realiza uma busca na lista para encontrar a posição de inserção que identificada pelo ponteiro ao predecessor pPre.
- A função pode retornar o ponteiro ao nó duplicado pLoc, que neste caso não interessa.

P5-04.h – Função pública Adicionar Nó

```
1  /*================== addNode ================
2    Inserts data into list.
3       Pre     pList is pointer to valid list
4               dataInPtr pointer to insertion data
5       Post    data inserted or error
6       Return  -1 if overflow
7               0 if successful
8               1 if dupe key
9  */
10 int addNode (LIST* pList, void* dataInPtr)
11 {
12 //Local Definitions
13   bool found;
14   bool success;
15
16   NODE* pPre;
17   NODE* pLoc;
18
19 //Statements
20   found = _search (pList, &pPre, &pLoc, dataInPtr);
21   if (found)
22      // Duplicate keys not allowed
23      return (+1);
24
25   success = _insert (pList, pPre, dataInPtr);
26   if (!success)
27      // Overflow
28      return (-1);
29   return (0);
30 } // addNode
```

TAD Lista (List ADT)

Inserir (_insert)

- O código para a função interna Inserir (_insert()) é o seguinte:

- A função Inserir (_insert()) contempla dois casos:
  - Inserir no inicio da lista (inclusive em uma lista vazia);
  - Inserir no meio ou no final da lista.

P5-05.h – Função privada de inserção

```
/*================== _insert ==================
  Inserts data pointer into a new node.
     Pre    pList pointer to a valid list
            pPre  pointer to data's predecessor
            dataInPtr data pointer to be inserted
     Post   data have been inserted in sequence
     Return boolean, true  if successful,
                     false if memory overflow
*/
static bool _insert (LIST* pList, NODE* pPre,
                     void* dataInPtr)
{
//Local Definitions
  NODE* pNew;

//Statements
  if (!(pNew = (NODE*) malloc(sizeof(NODE))))
     return false;

  pNew->dataPtr    = dataInPtr;
  pNew->link       = NULL;

  if (pPre == NULL)
     {
      // Adding before first node or to empty list.
      pNew->link       = pList->head;
      pList->head      = pNew;
      if (pList->count == 0)
       // Adding to empty list. Set rear
         pList->rear = pNew;
     } // if pPre
```

TAD Lista (List ADT)

Inserir (_insert)

- A função Inserir (_insert()) contempla dois casos:
  - Inserir no inicio da lista (inclusive em uma lista vazia);
  - Inserir no meio ou no final da lista.

P5-05.h – Função privada de inserção, continuação...

```
32    else
33       {
34        // Adding in middle or at end
35        pNew->link  = pPre->link;
36        pPre->link  = pNew;
37
38       // Now check for add at end of list
39        if (pNew->link     == NULL)
40             pList->rear   =  pNew;
41       } // if else
42
43    (pList->count)++;
44    return true;
45 } // _insert
```

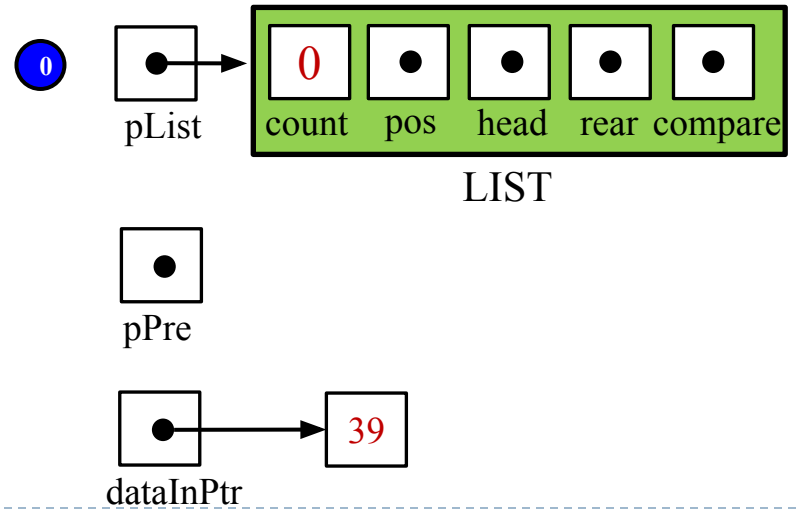
TAD Lista (List ADT)

Inserir (_insert)

- A função Inserir (_insert()) recebe como parâmetros um ponteiro a lista (pList), um ponteiro a posição de inserção (pPre) e um ponteiro ao dado a ser inserido (dataInPtr).

```
static bool _insert (LIST* pList, NODE* pPre, void* dataInPtr)
```

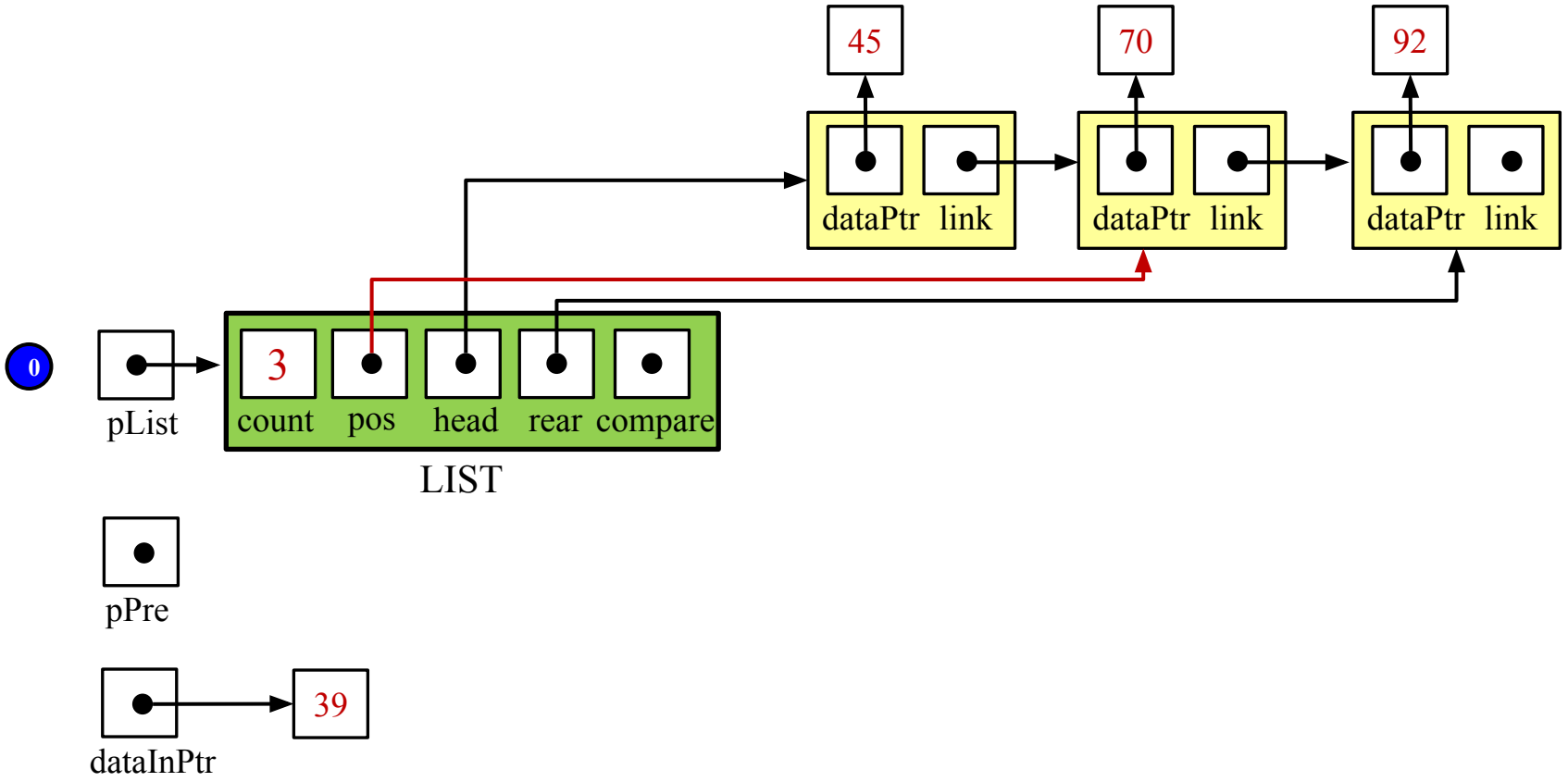
- O **primeiro caso** acontece ao inserir no inicio da lista. Aqui temos duas situações: A primeira situação é quando a lista está vazia. Com isso, a posição de inserção será nula.



TAD Lista (List ADT)

Inserir (_insert)

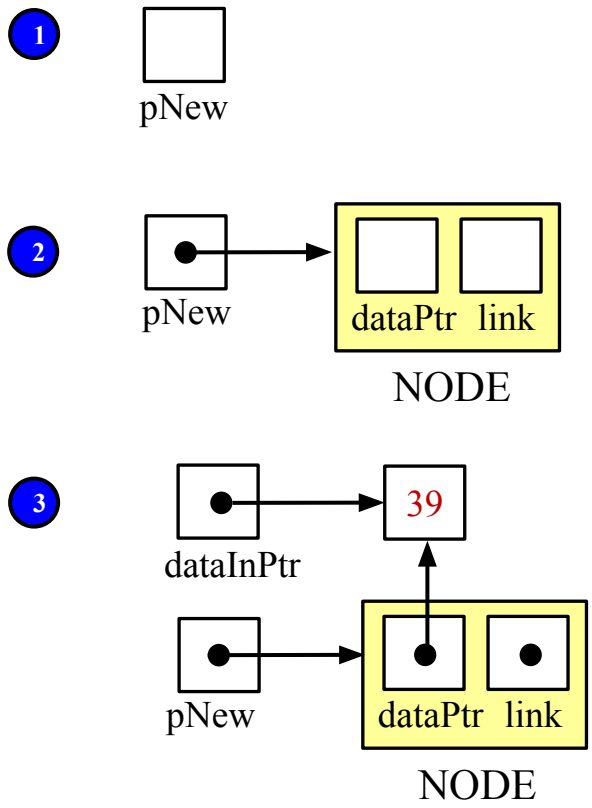
- A segunda situação acontece ao inserir um elemento na primeira posição em uma lista que não está vazia. Com isso, a posição de inserção também será nula.



TAD Lista (List ADT)

Inserir (_insert)

- Para inserir um novo dado, devemos: alocar memória para um novo nó e preencher esse novo nó com o dado correspondente. Ou seja, preparar o nó que será inserido.

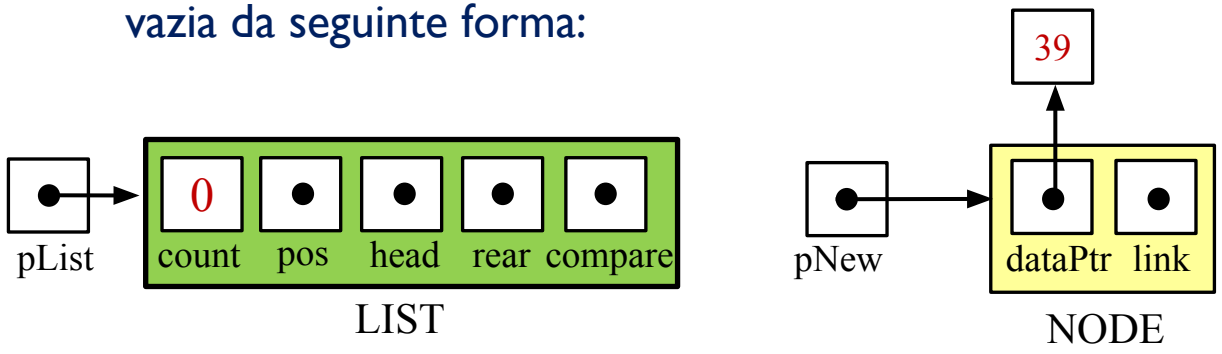


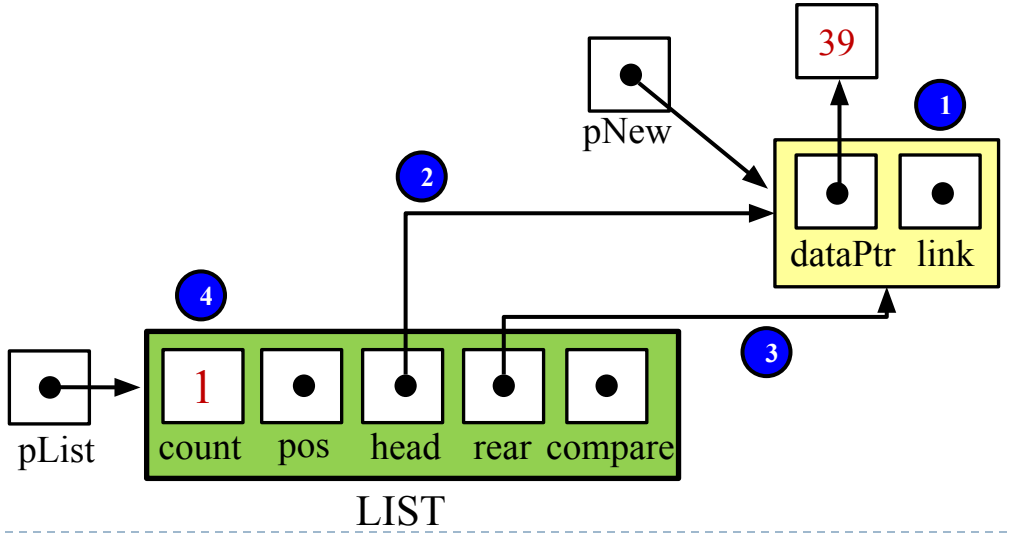

```
1 NODE*   pNew;
2 pNew = (NODE*) malloc(sizeof(NODE));
3 pNew->dataPtr = dataInPtr;
  pNew->link    = NULL;
```

TAD Lista (List ADT)

Inserir (_insert)

- No **primeiro caso**, na primeira situação, o novo nó é inserido em uma lista vazia da seguinte forma:



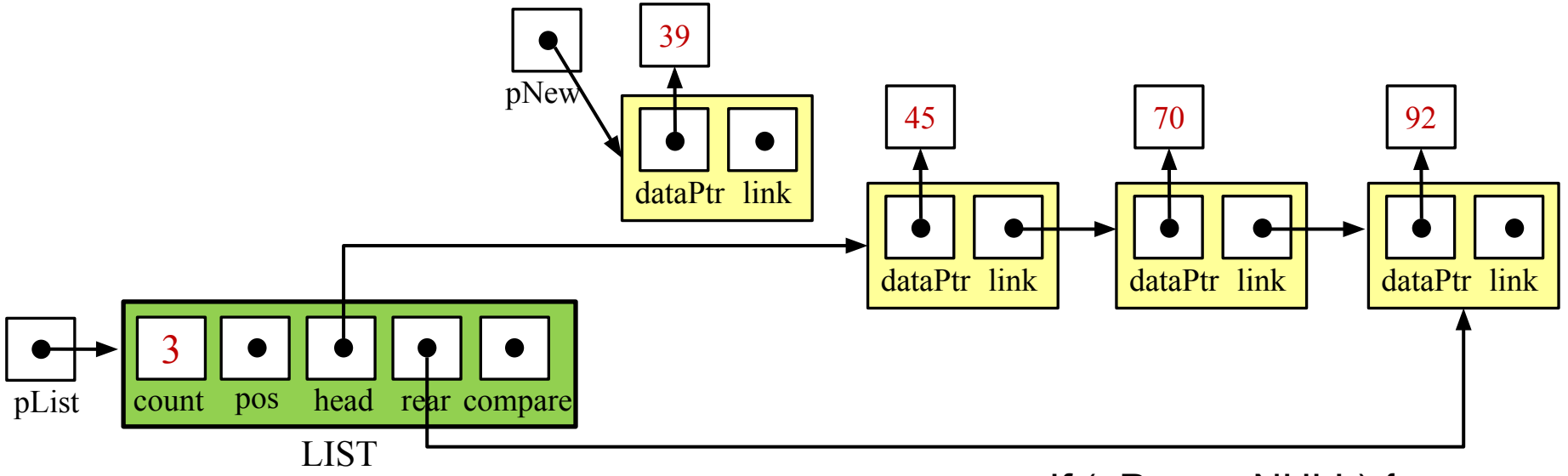



```
    If (pPre == NULL) {
1       pNew->link  = pList->head;
2       pList->head = pNew;
        if (pList->count == 0)
3           pList->rear = pNew;
    }
4   (pList->count)++;
```

TAD Lista (List ADT)

Inserir (_insert)

- No **primeiro caso**, na segunda situação, o novo nó é inserido em uma lista que já contem elementos, da seguinte forma:

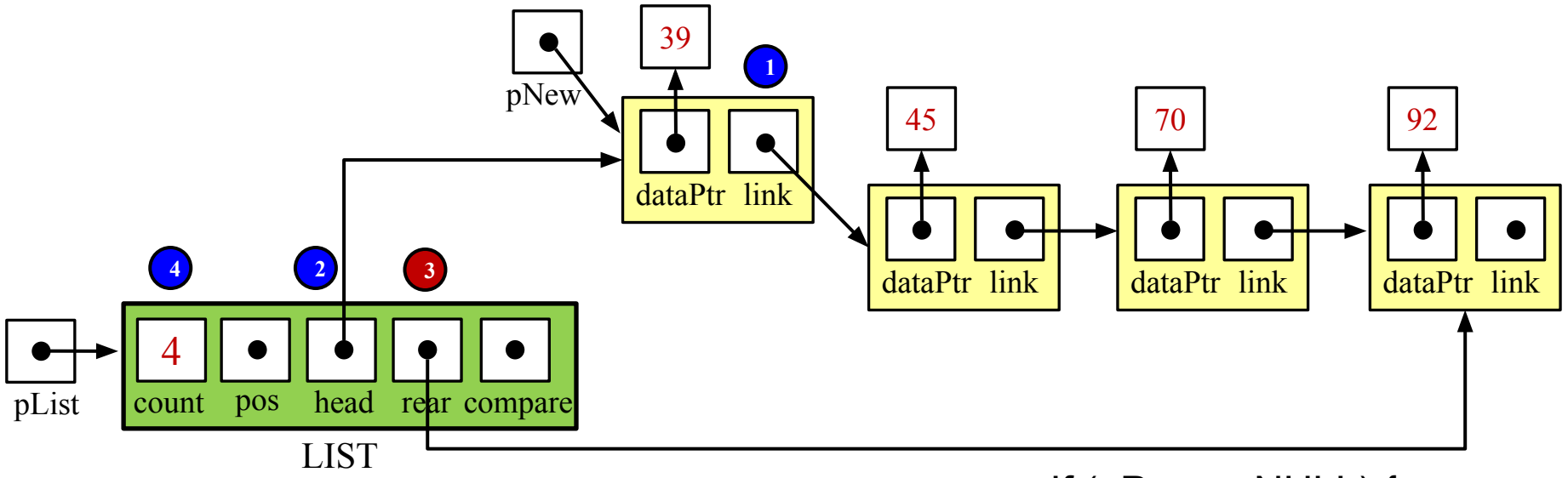



```
If (pPre == NULL) {
1     pNew->link  = pList->head;
2     pList->head = pNew;
      if (pList->count == 0)
3         pList->rear = pNew;
   }
4  (pList->count)++;
```

TAD Lista (List ADT)

Inserir (_insert)

- O resultado da inserção no **primeiro caso**, na segunda situação é o seguinte:



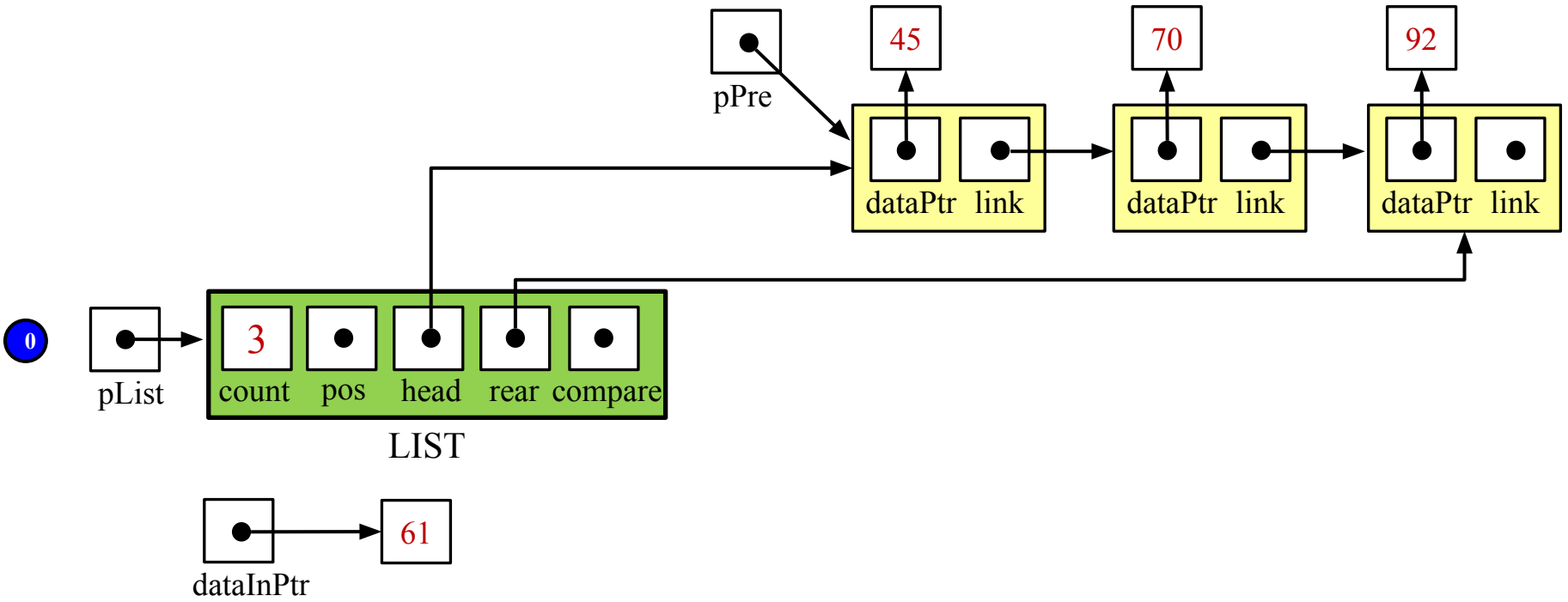

```
    If (pPre == NULL) {
1       pNew->link  = pList->head;
2       pList->head = pNew;
        if (pList->count == 0)
3           pList->rear = pNew;
    }
4   (pList->count)++;
```

Observe que o código é o mesmo em ambas situações do primeiro caso. No entanto na segunda situação a linha em 3 não é executada.

TAD Lista (List ADT)

Inserir (_insert)

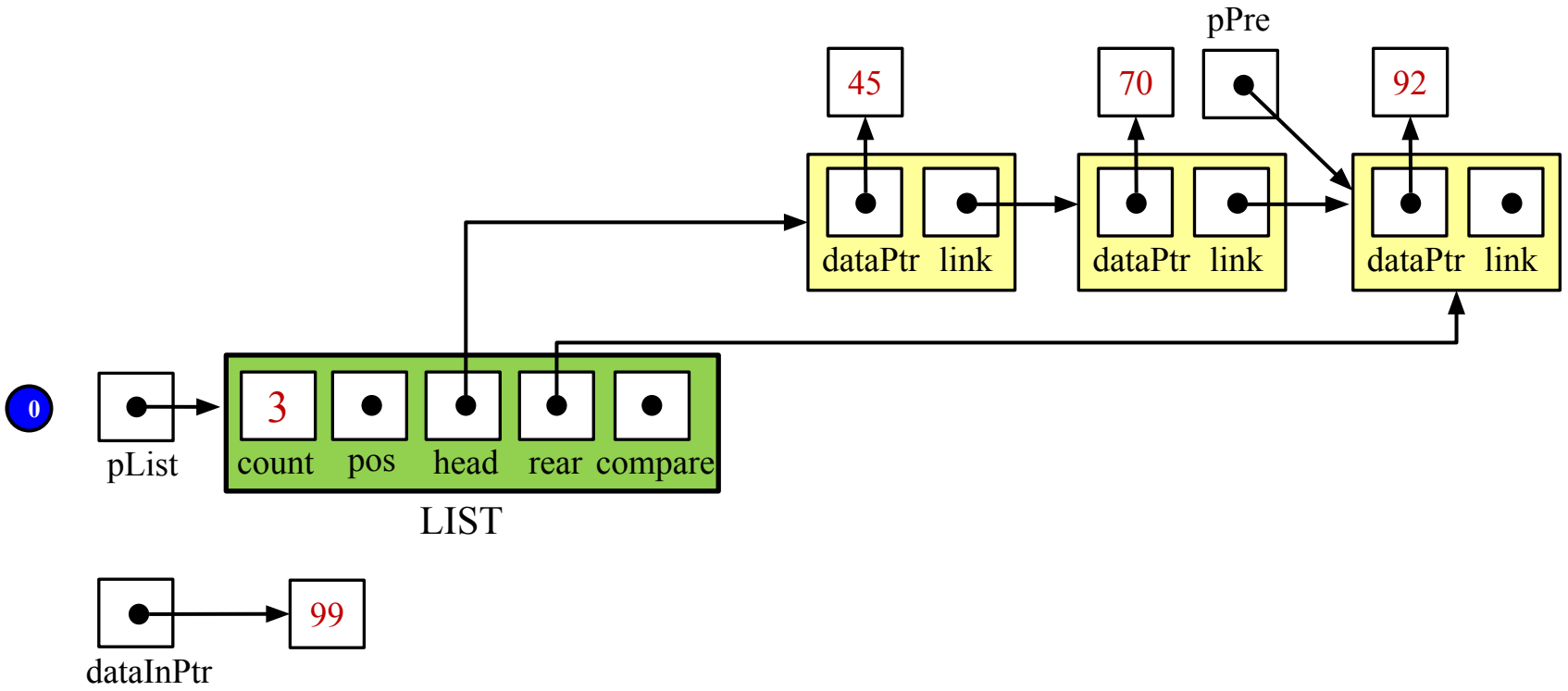
- Já, o **segundo caso** de inserção acontece quando a lista não está vazia e o elemento a ser inserido deve ocupar uma posição no meio da lista ou no final da lista. Com isso, a posição de inserção é dada pelo ponteiro pPre.
- A figura ilustra a primeira situação, a inserção **no meio da lista**.



TAD Lista (List ADT)

Inserir (_insert)

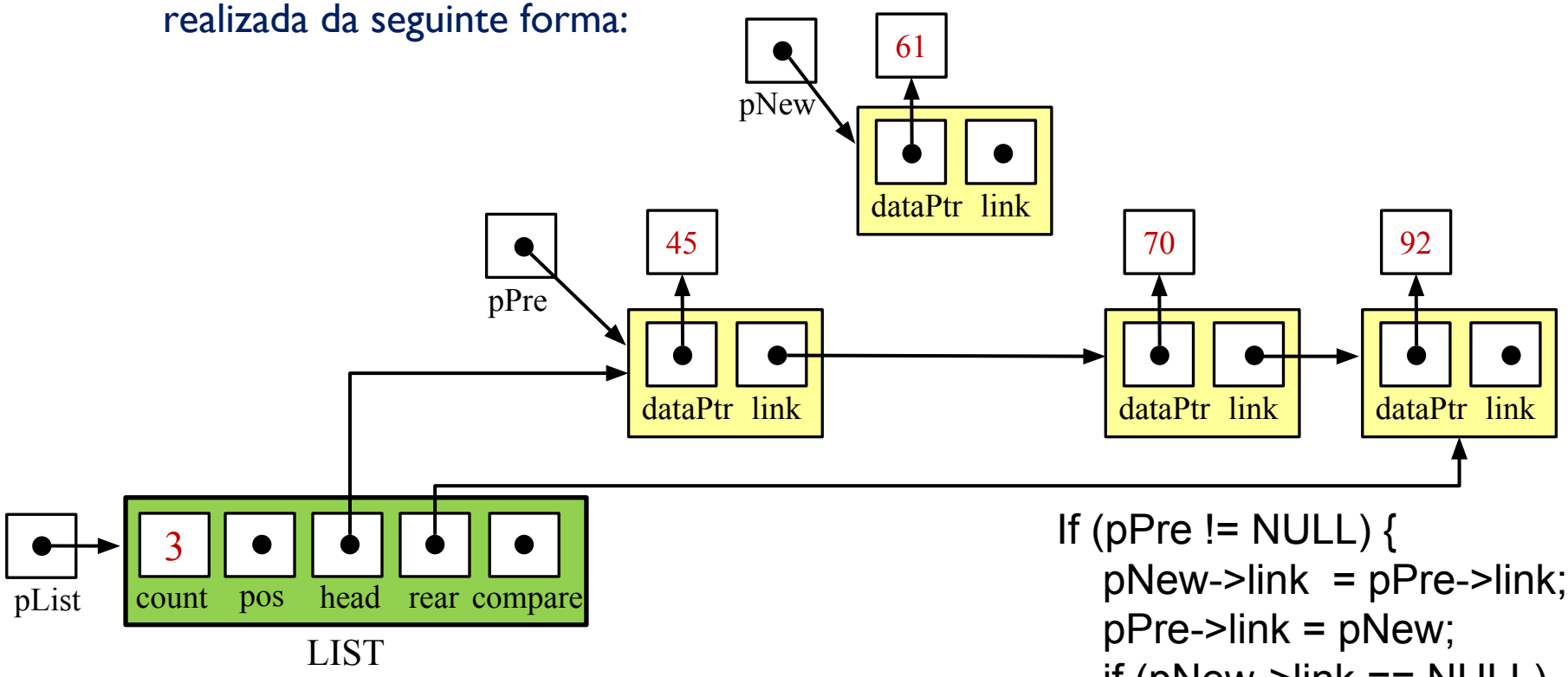
- A figura ilustra a segunda situação do **segundo caso**, a inserção **no final da lista**. Onde a posição de inserção é dada pelo ponteiro pPre.



TAD Lista (List ADT)

Inserir (_insert)

- A primeira situação do **segundo caso,** a inserção **no meio da lista** é realizada da seguinte forma:

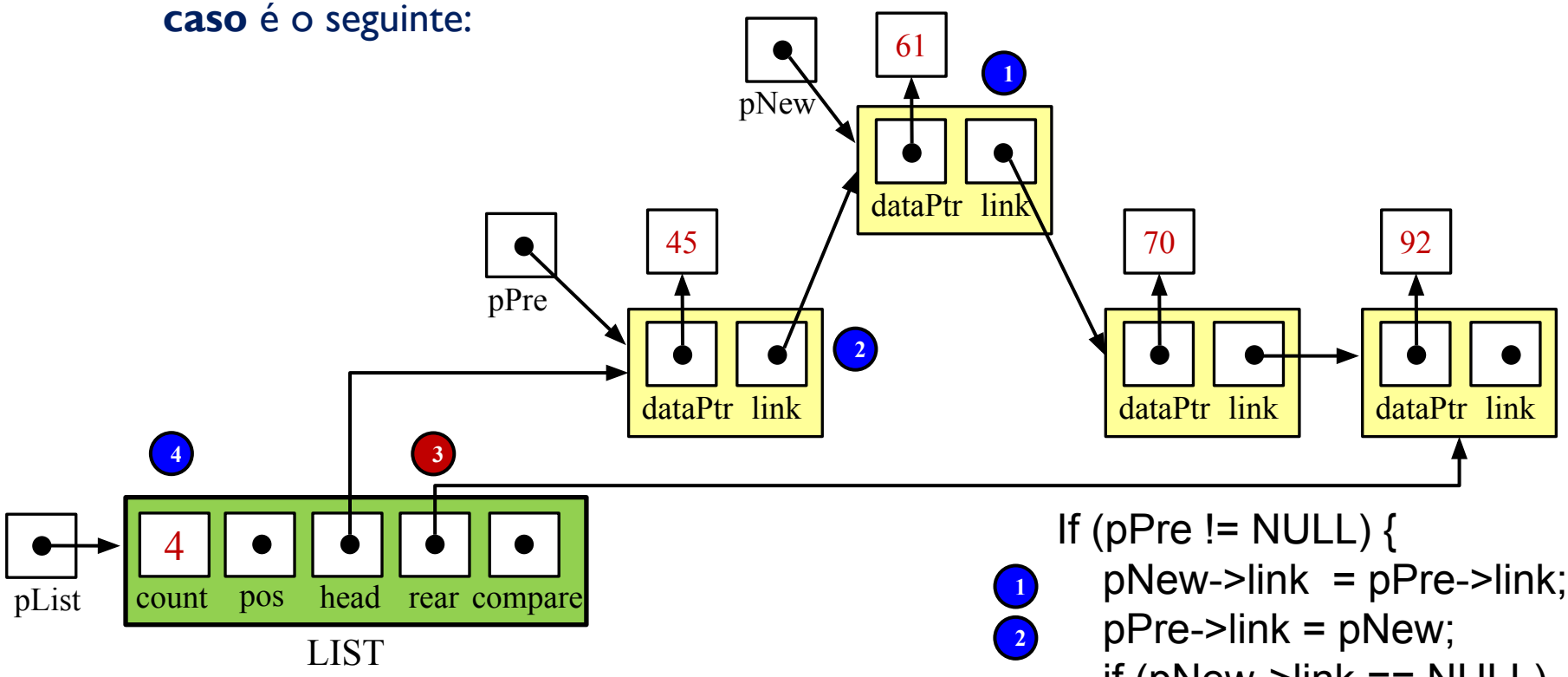



```
If (pPre != NULL) {
    pNew->link  = pPre->link;
    pPre->link = pNew;
    if (pNew->link == NULL)
        pList->rear = pNew;
}
(pList->count)++;
```

TAD Lista (List ADT)

Inserir (_insert)

- O resultado da inserção **no meio da lista**, a primeira situação do **segundo caso** é o seguinte:



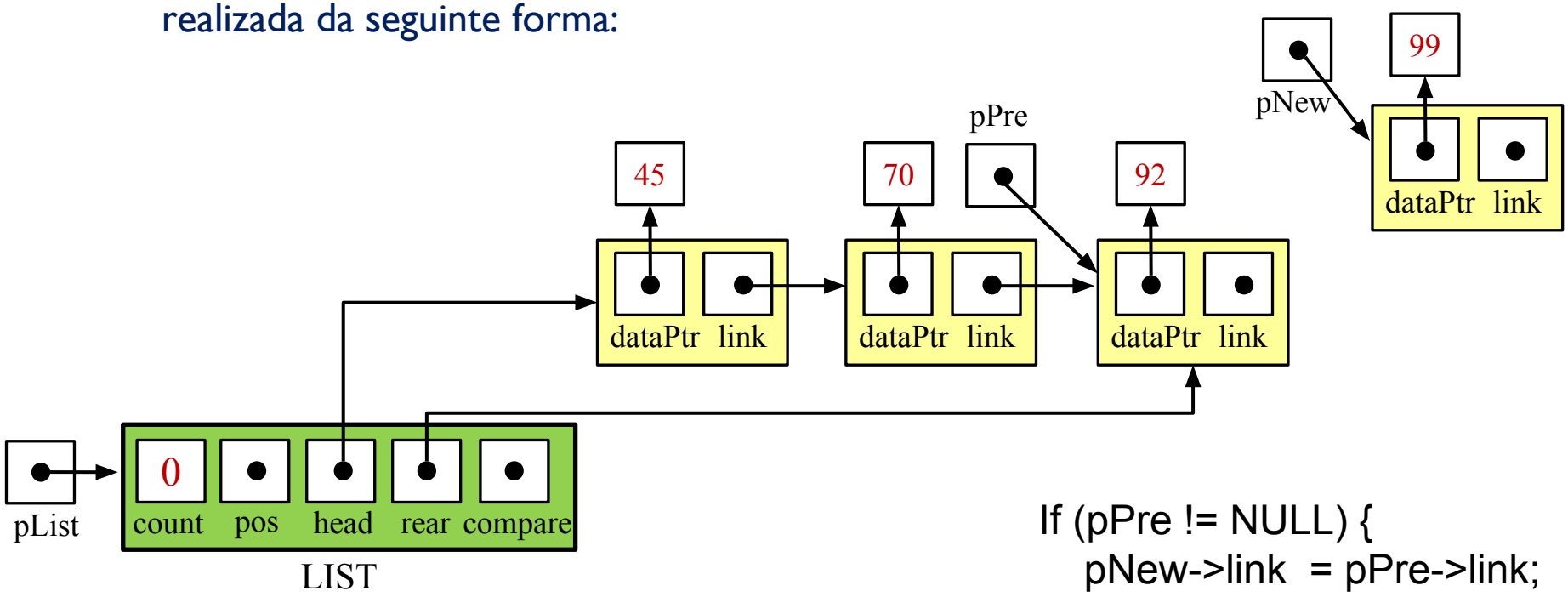

```
If (pPre != NULL) {
1     pNew->link  = pPre->link;
2     pPre->link = pNew;
      if (pNew->link == NULL)
3         pList->rear = pNew;
}
4  (pList->count)++;
```

Observa-se que a linha em 3 não é executada.

TAD Lista (List ADT)

Inserir (_insert)

- A segunda situação do **segundo caso,** a inserção **no final da lista** é realizada da seguinte forma:

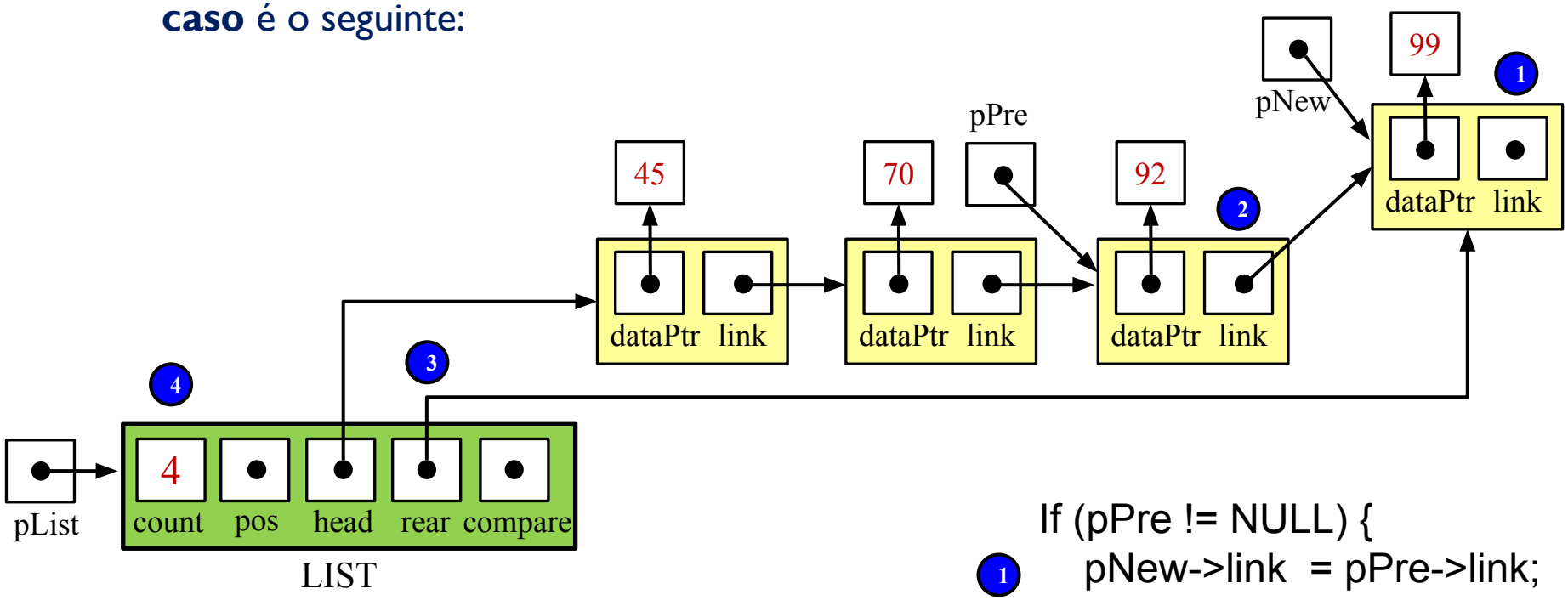



```
If (pPre != NULL) {
    pNew->link  = pPre->link;
    pPre->link = pNew;
    if (pNew->link == NULL)
        pList->rear = pNew;
}
(pList->count)++;
```

TAD Lista (List ADT)

Inserir (_insert)

- O resultado da inserção **no final da lista**, a segunda situação do **segundo caso** é o seguinte:




```
    If (pPre != NULL) {
1       pNew->link  = pPre->link;
2       pPre->link = pNew;
        if (pNew->link == NULL)
3           pList->rear = pNew;
    }
4   (pList->count)++;
```

Observa-se que todas as linhas são executadas.

TAD Lista (List ADT)
Remover Nó (removeNode)

- A função Remover Nó (removeNode()) é uma função pública que recebe o dado a ser removido da lista e realiza uma busca na lista para encontrar esse dado.
- A busca é realizada por uma função privada (_search).
- Existem dois resultados possíveis na remoção de um nó:
  - A busca foi bem sucedida e o nó foi removido (True);
  - A busca falhou porque o nó não foi encontrado (False).

TAD Lista (List ADT)

Remover Nó (removeNode)

- A função Remover Nó (removeNode()) realiza a busca pelo nó que contém o dado desejado.
- Para isso utiliza duas funções internas:
- _search() realiza a busca pelo nó que contém o dado desejado. A função retorna verdadeiro o falso.
- _delete() elimina o nó seguinte ao nó apontado por pPre.

P5-06.h – Função pública Remover Nó

```
1  /*================ removeNode ===============
2    Removes data from list.
3       Pre    pList pointer to a valid list
4              keyPtr pointer to key to be deleted
5              dataOutPtr pointer to data pointer
6       Post   Node deleted or error returned.
7       Return false not found; true deleted
8  */
9  bool removeNode   (LIST*  pList, void*  keyPtr,
10                    void** dataOutPtr)
11 {
12 //Local Definitions
13   bool found;
14
15   NODE* pPre;
16   NODE* pLoc;
17
18 //Statements
19   found = _search (pList, &pPre, &pLoc, keyPtr);
20   if (found)
21      _delete (pList, pPre, pLoc, dataOutPtr);
22
23   return found;
24 } // removeNode
```

TAD Lista (List ADT)
Eliminar(_delete)

- A função Eliminar (_delete()) redefine os ponteiros da lista de maneira a excluir o nó apontado por pLoc.
- Diminui o contador de elementos.
- Libera a memória alocada ao nó apontado por pLoc.

P5-07.h – Função privada de eliminação

```
11  void _delete (LIST* pList, NODE*  pPre,
12                NODE* pLoc,  void** dataOutPtr)
13  {
14  //Statements
15    *dataOutPtr = pLoc->dataPtr;
16    if (pPre == NULL)
17       // Deleting first node
18       pList->head = pLoc->link;
19    else
20       // Deleting any other node
21       pPre->link = pLoc->link;
22
23    // Test for deleting last node
24    if (pLoc->link == NULL)
25        pList->rear = pPre;
26
27    (pList->count)--;
28    free (pLoc);
29
30    return;
31  } // _delete
```

TAD Lista (List ADT)

Procurar Lista (search List)

- A função Procurar Lista (searchList()) localiza um nó na lista que contem o elemento procurado e repassa o endereço do dado na lista.
- Para isso utiliza uma função interna que localiza o nó buscado pLoc e seu predecessor (pPre).

P5-08.h – Função pública de Busca na lista

```
 9  bool searchList (LIST*  pList, void* pArgu,
10                   void** pDataOut)
11  {
12  //Local Definitions
13    bool  found;
14
15    NODE* pPre;
16    NODE* pLoc;
17
18  //Statements
19    found = _search (pList, &pPre, &pLoc, pArgu);
20    if (found)
21        *pDataOut = pLoc->dataPtr;
22    else
23        *pDataOut = NULL;
24    return found;
25  } // searchList
```

TAD Lista (List ADT)

Buscar (_search)

P5-09.h – Função privada de Busca

- A função _search() tenta localizar o nó apontado em referencia dupla por pLoc.
- Utiliza uma macro para a função de comparação.
- Se a lista estiver vazia retorna falso.
- Se o resultado da comparação com o último elemento for >0 (o elemento procurado é maior que o último) retorna falso.

```
15  bool _search (LIST*  pList, NODE** pPre,
16                NODE** pLoc,  void*  pArgu)
17  {
18  //Macro Definition
19  #define COMPARE \
20    ( ((* pList->compare) (pArgu, (*pLoc)->dataPtr)) )
21
22  #define COMPARE_LAST \
23    ((* pList->compare) (pArgu, pList->rear->dataPtr))
24
25  //Local Definitions
26    int result;
27
28  //Statements
29    *pPre  = NULL;
30    *pLoc  = pList->head;
31    if (pList->count == 0)
32        return false;
33
34    // Test for argument > last node in list
35    if ( COMPARE_LAST > 0)
36       {
37        *pPre = pList->rear;
38        *pLoc = NULL;
39        return false;
40       } // if
```

TAD Lista (List ADT)

Buscar (_search)

P5-09.h – Função privada de Busca, continuação...

- Enquanto o resultado da comparação for > 0 (o elemento procurado for maior que o atual) atualiza os ponteiros pPre e pLoc.
- Se o resultado da comparação for = 0 (encontrou o elemento procurado) retorna verdadeiro; caso contrário retorna falso.

```
41
42    while ( (result = COMPARE) > 0 )
43       {
44        // Have not found search argument location
45        *pPre = *pLoc;
46        *pLoc = (*pLoc)->link;
47       } // while
48
49    if (result == 0)
50       // argument found--success
51       return true;
52    else
53       return false;
54 } // _search
```

TAD Lista (List ADT)

Recupera No (retrieveNode)

- A função Recupera Nó (retrieveNode()) realiza uma busca na lista e recupera o ponteiro para o dado na lista que coincide com o argumento passado como parâmetro.

P5-10.h – Função pública Recuperar Nó

```
1  /*================== retrieveNode ================
2    This algorithm retrieves data in the list without
3    changing the list contents.
4       Pre    pList pointer to initialized list.
5              pArgu pointer to key to be retrieved
6       Post   Data (pointer) passed back to caller
7       Return boolean true success; false underflow
8  */
9  static bool retrieveNode (LIST*  pList,
10                            void*  pArgu,
11                            void** dataOutPtr)
12 {
13 //Local Definitions
14   bool  found;
15
16   NODE* pPre;
17   NODE* pLoc;
18
19 //Statements
20   found = _search (pList, &pPre, &pLoc, pArgu);
21   if (found)
22      {
23       *dataOutPtr = pLoc->dataPtr;
24       return true;
25      } // if
26
27   *dataOutPtr = NULL;
28   return false;
29 } // retrieveNode
```

TAD Lista (List ADT)
Lista Vazia (emptyList)

- A função Lista Vazia (emptyList()) verifica se a lista está vazia.
- Se o contador do número de elementos está zerado, retorna true.
- Caso contrário, retorna false.

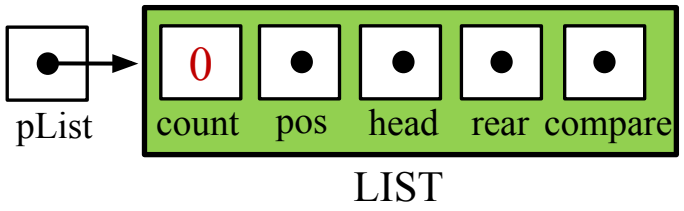
P5-11.h – Função pública Lista Vazia

```
1  /*================ emptyList ================
2    Returns boolean indicating whether or not the
3    list is empty
4       Pre    pList is a pointer to a valid list
5       Return boolean true empty; false list has data
6  */
7  bool emptyList (LIST* pList)
8  {
9  //Statements
10   return (pList->count == 0);
11 } // emptyList
```

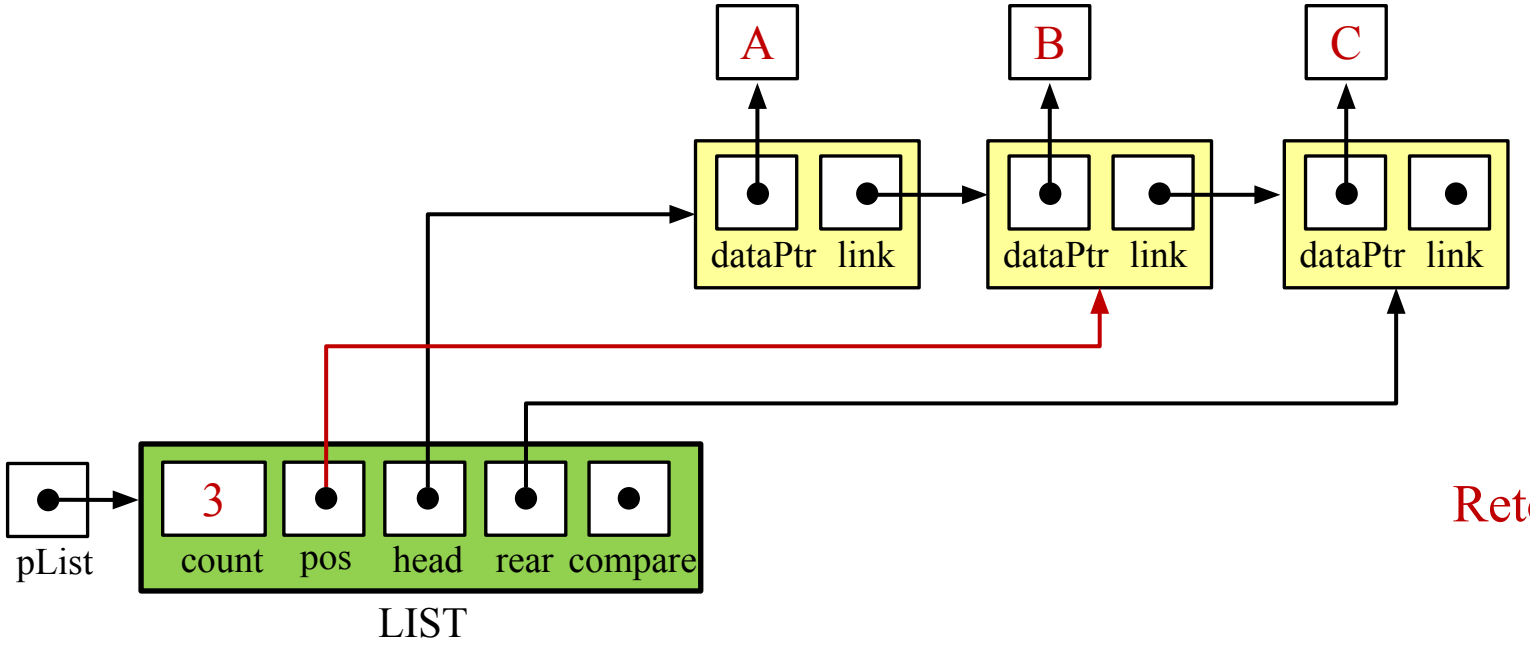
TAD Fila (List ADT)

Lista Vazia (emptyList)

- A figura ilustra os dois casos possíveis para a função Lista Vazia (Empty List).



Retorna True!!!



Retorna False!!!

TAD Lista (List ADT)

Lista Cheia (fullList)

- A função Lista Cheia (fullList()) verifica se a lista está cheia.
- A linguagem C não fornece uma maneira de testar se existe espaço disponível na memória dinâmica.
- Uma forma de fazer isso é realizando uma tentativa de alocação de um nó e verificar se a tentativa foi bem sucedida ou não.
- Se o nó for alocado, esse nó deve ser liberado e retorna-se false. A lista não está cheia.
- Caso contrário, retorna-se true. A lista está cheia.

TAD Lista (List ADT)

Lista Cheia (fullList)

- O código da função Lista Cheia (fullList()) é mostrado a seguir:

P5-12.h – Função pública Lista Cheia

```
1  /*================== fullList ================
2    Returns boolean indicating no room for more data.
3    This list is full if memory cannot be allocated for
4    another node.
5       Pre    pList pointer to valid list
6       Return boolean true if full
7                      false if room for node
8  */
9  bool fullList (LIST* pList)
10 {
11 //Local Definitions
12 NODE* temp;
13
14 //Statements
15   if ((temp = (NODE*)malloc(sizeof(*(pList->head)))))
16      {
17       free (temp);
18       return false;
19      } // if
20
21   // Dynamic memory full
22   return true;
23
24 } // fullList
```

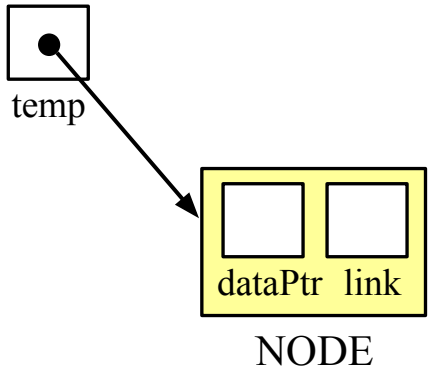
TAD Lista (List ADT)
Lista Cheia (fullList)

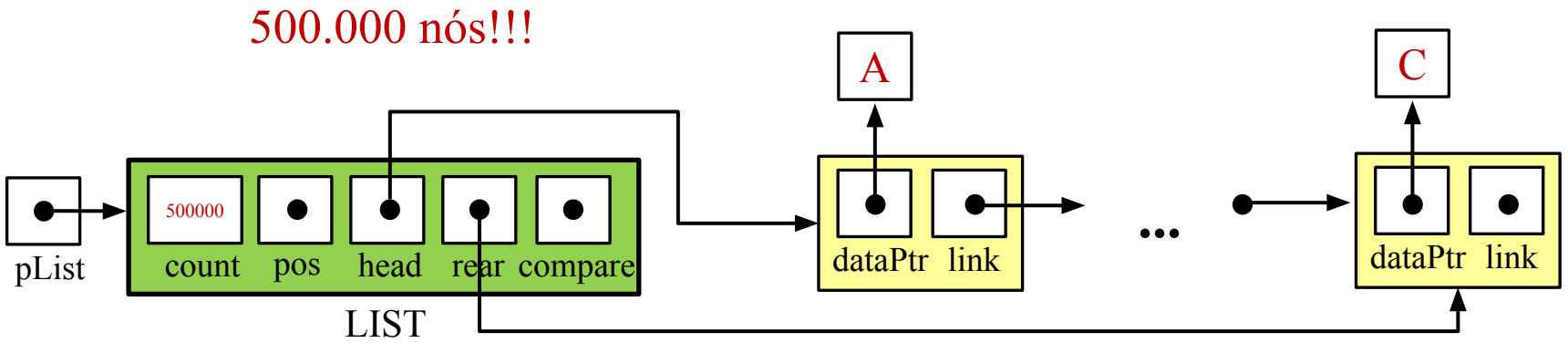
- A figura ilustra a função Fila Cheia (Full Queue):

temp:=(NODE*)malloc(sizeof(*(pList->head)));

ou

temp:=(NODE*)malloc(sizeof(NODE));





temp==NULL
Lista Cheia!!!

TAD Lista (List ADT)
Contador Lista (listCount)

- A função Contador Lista (listCount()) retorna o valor do contador que registra o número de nós da lista. O contador se encontra no nó cabeça-lho da lista.
- O código da função é o seguinte:

P5-13.h – Função pública Contador da Lista

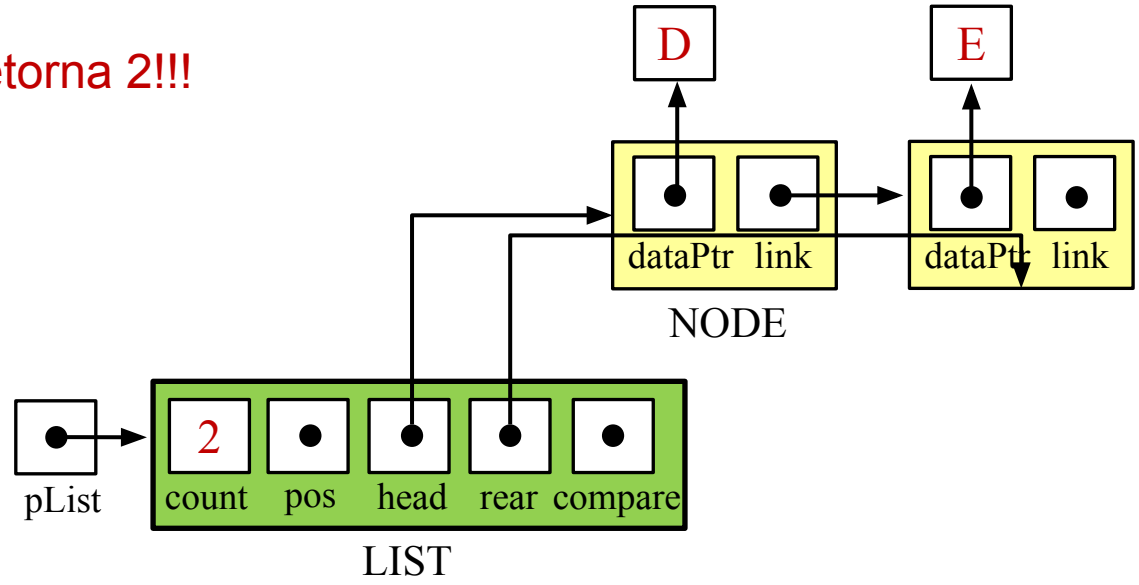
```
1  /*================== listCount ==================
2    Returns number of nodes in list.
3       Pre    pList is a pointer to a valid list
4       Return count for number of nodes in list
5  */
6  int listCount(LIST* pList)
7  {
8  //Statements
9
10   return pList->count;
11
12 } // listCount
```

TAD Lista (List ADT)
Contador Lista (listCount)

- A figura ilustra a função Contador Lista (listCount).

```
return   pList->count;
```

Retorna 2!!!



TAD Lista (List ADT)

Posicionar (traverse)

- A função Posicionar (traverse()) atualiza o ponteiro de acesso a um dado da lista. Pode ser o ponteiro ao primeiro dado da lista ou aquele que ocupa a posição corrente da lista.
- A função usa um parâmetro (FromWhere) para decidir qual ponteiro recuperar (apontando ao primeiro elemento ou ao elemento corrente).
- A função retorna verdadeiro caso o ponteiro for localizado. Caso contrário, atinge o fim da lista e retorna falso. O código da função é o seguinte:

TAD Lista (List ADT)

Posicionar (traverse)

- O código da função é o seguinte:

Flag, usado para indicar se o percurso começa no primeiro elemento ou na posição registrada.

P5-14.h – Função pública Posicionar na Lista

```
1  /*================== traverse ================
2    Traverses a list. Each call either starts at the
3    beginning of list or returns the location of the
4    next element in the list.
5       Pre    pList       pointer to a valid list
6              fromWhere   0 to start at first element
7              dataPtrOut  address of pointer to data
8       Post   if more data, address of next node
9       Return true node located; false if end of list
10 */
11 bool traverse (LIST*  pList,
12                 int    fromWhere,
13                 void** dataPtrOut)
14 {
15 //Statements
16   if (pList->count == 0)
17       return false;
18
19   if (fromWhere == 0)
20      {
21       // Start from first node
22       pList->pos  = pList->head;
23       *dataPtrOut = pList->pos->dataPtr;
24       return true;
25      } // if fromwhere
```

TAD Lista (List ADT)

Posicionar (traverse)

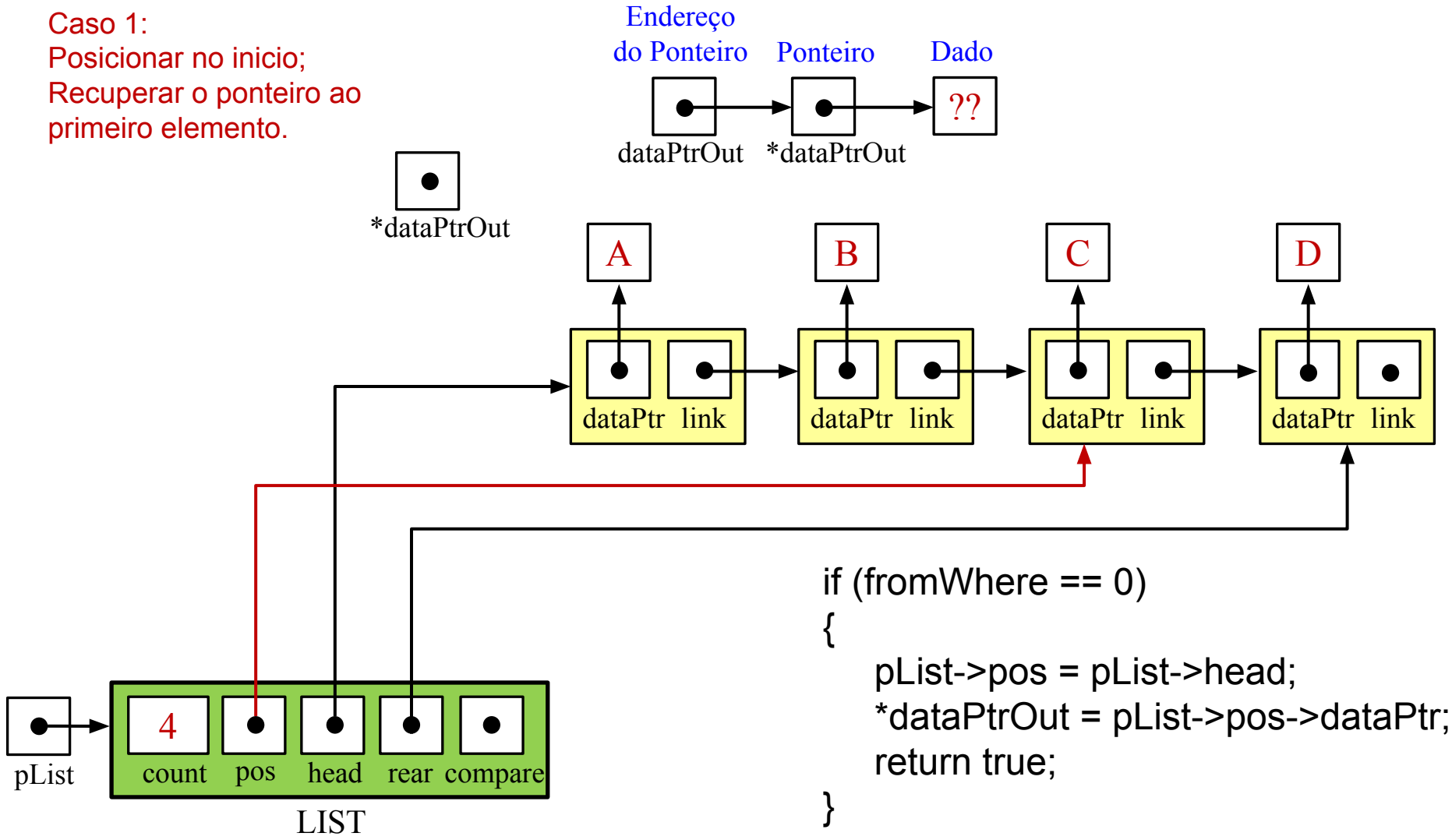
P5-14.h – Função pública Posicionar na Lista, continuação...

```
26    else
27       {
28        // Start from current position
29        if (pList->pos->link == NULL)
30            return false;
31        else
32           {
33            pList->pos  = pList->pos->link;
34            *dataPtrOut = pList->pos->dataPtr;
35            return true;
36           } // if else
37       } // if fromwhere else
38  } // traverse
```

TAD Lista (List ADT)

Posicionar (traverse)

Caso 1:
Posicionar no inicio;
Recuperar o ponteiro ao primeiro elemento.

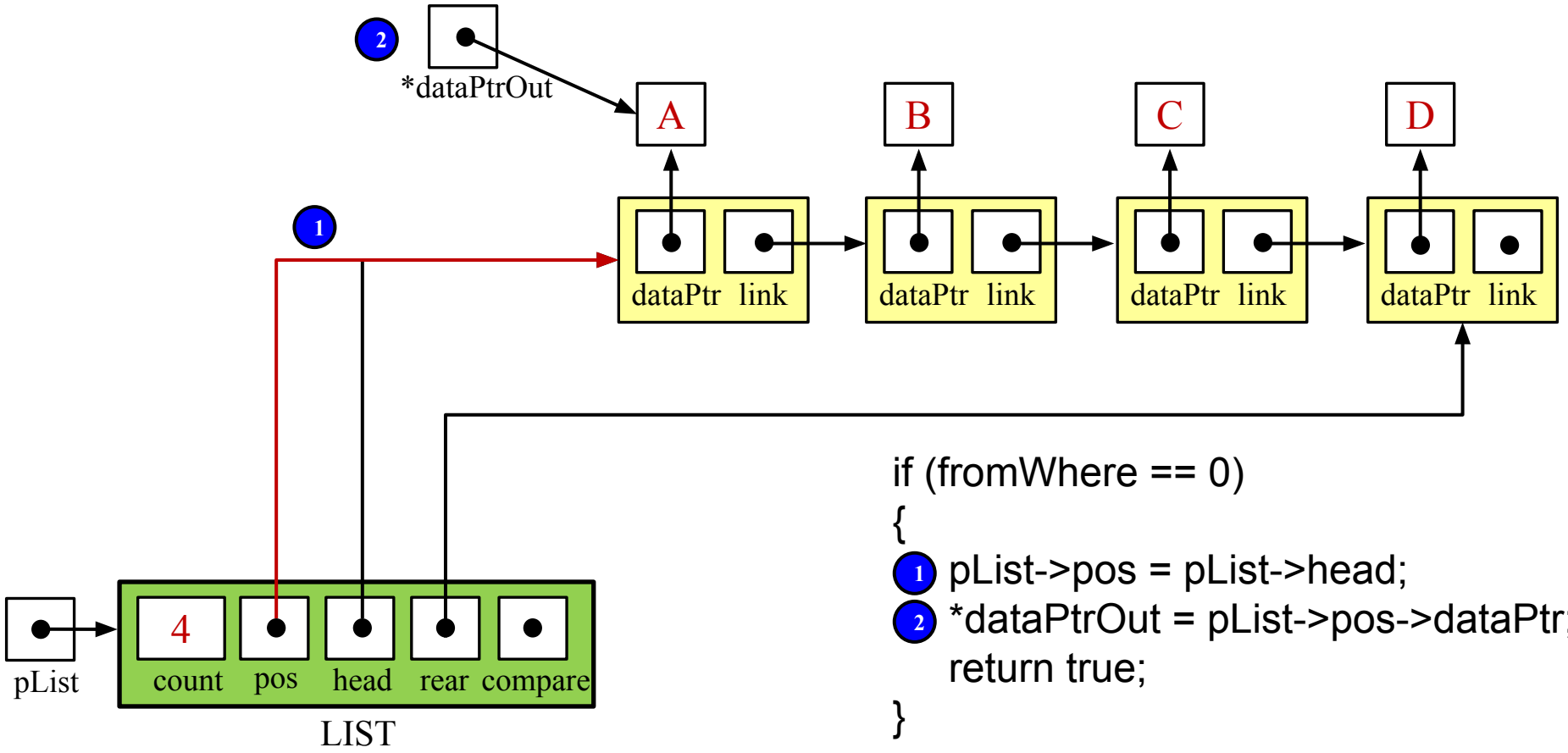



```
if (fromWhere == 0)
{
    pList->pos = pList->head;
    *dataPtrOut = pList->pos->dataPtr;
    return true;
}
```

TAD Lista (List ADT)

Posicionar (traverse)

Caso 1:
Posicionar no inicio;
Recuperar o ponteiro ao primeiro elemento.

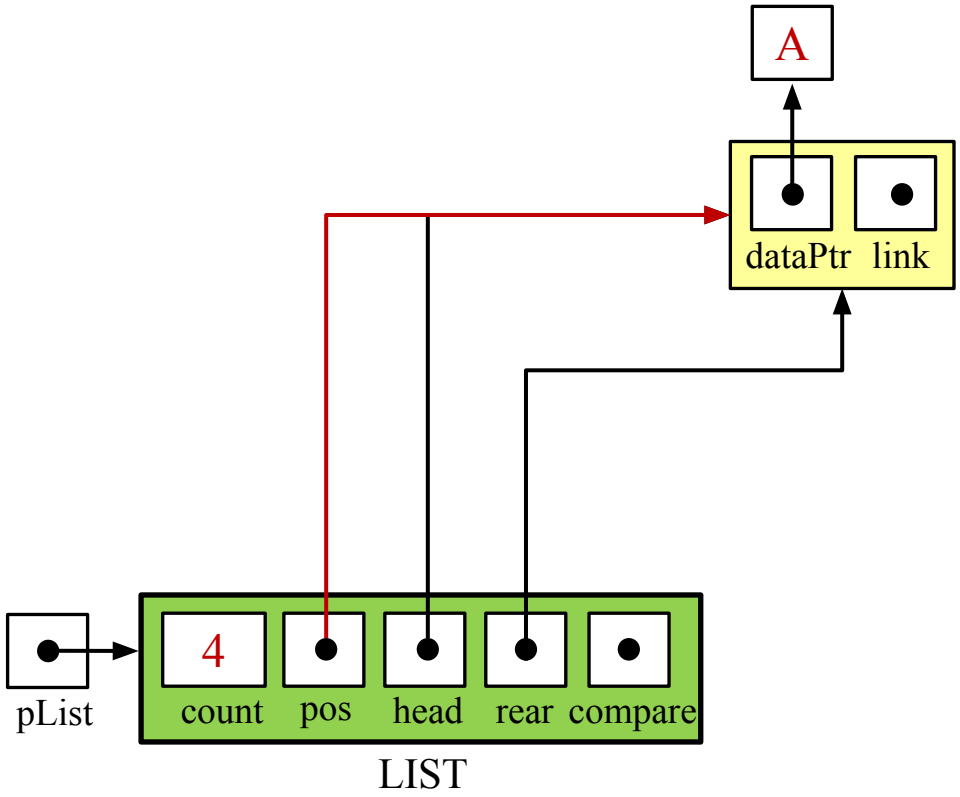



```
if (fromWhere == 0)
{
1 pList->pos = pList->head;
2 *dataPtrOut = pList->pos->dataPtr;
    return true;
}
```

TAD Lista (List ADT)

Posicionar (traverse)

Caso 2:
Não existe seguinte a posição corrente;
Posicionar após a posição corrente;
Recuperar ponteiro ao seguinte elemento da posição corrente.

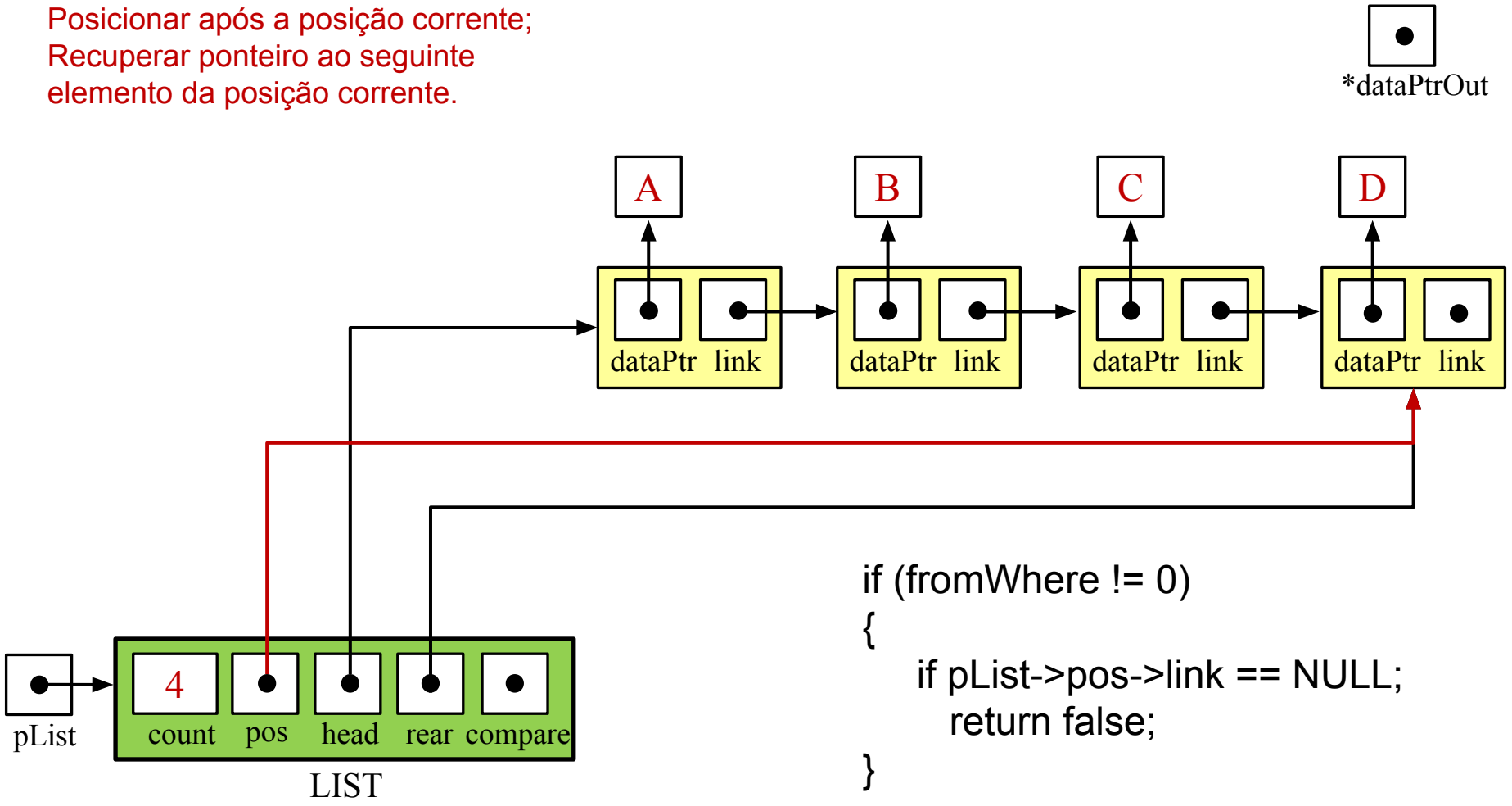



```
if (fromWhere != 0)
{
    if pList->pos->link == NULL;
       return false;
}
```

TAD Lista (List ADT)

Posicionar (traverse)

Caso 2:
Não existe seguinte a posição corrente;
Posicionar após a posição corrente;
Recuperar ponteiro ao seguinte
elemento da posição corrente.

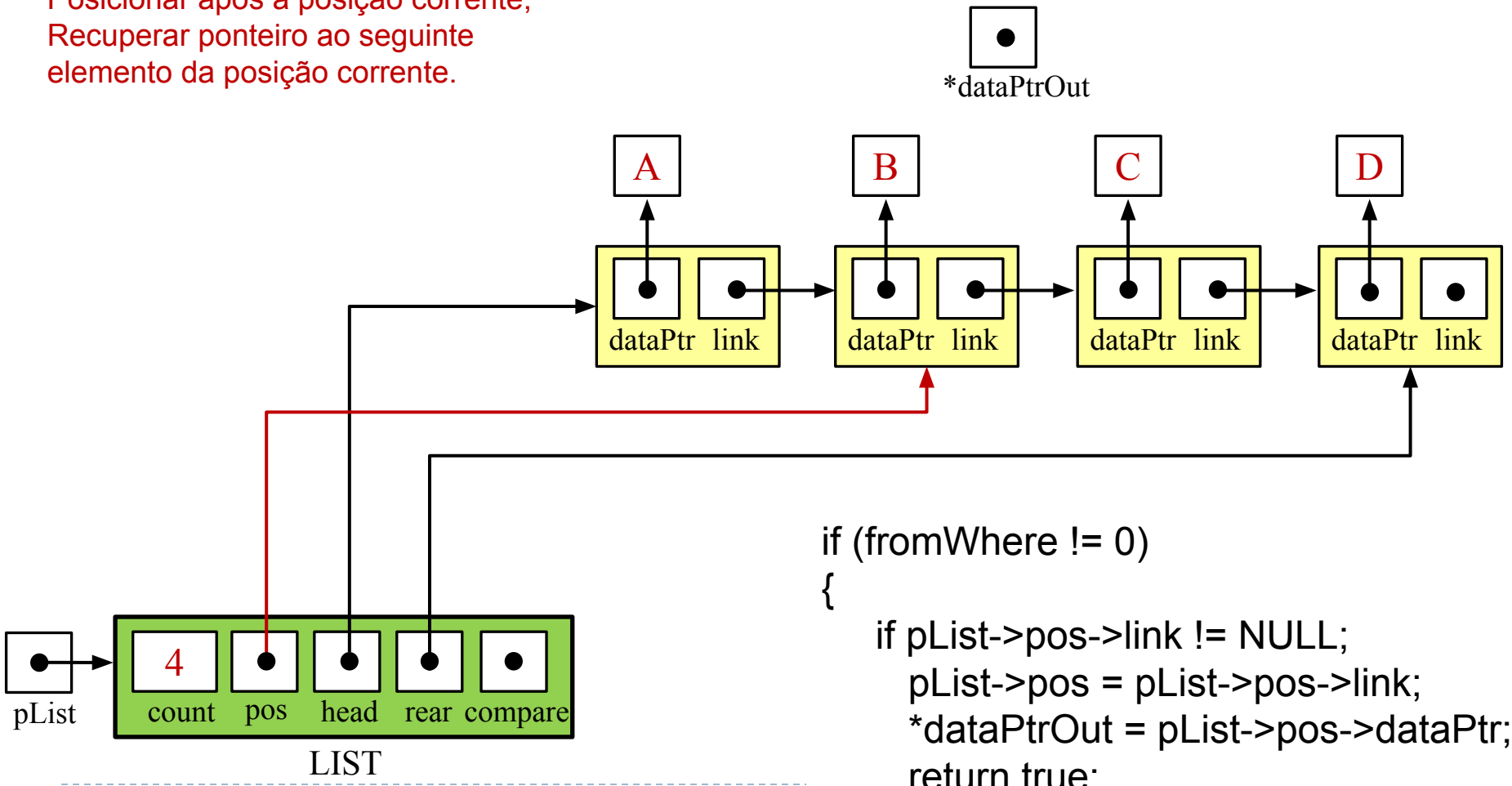



```
if (fromWhere != 0)
{
    if pList->pos->link == NULL;
       return false;
}
```

TAD Lista (List ADT)

Posicionar (traverse)

Caso 3:
Existe seguinte a posição corrente;
Posicionar após a posição corrente;
Recuperar ponteiro ao seguinte elemento da posição corrente.

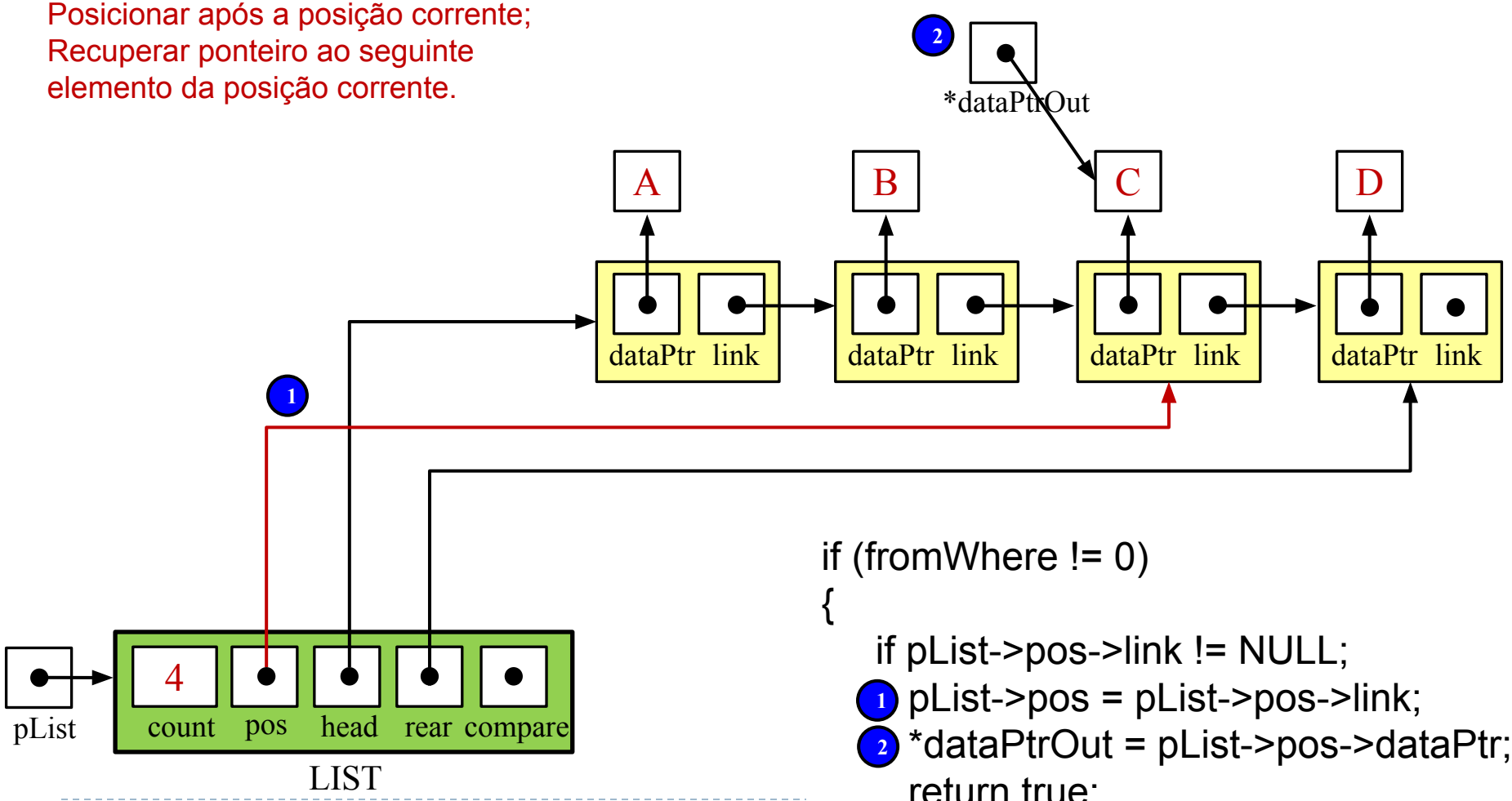



```
if (fromWhere != 0)
{
    if pList->pos->link != NULL;
       pList->pos = pList->pos->link;
       *dataPtrOut = pList->pos->dataPtr;
       return true;
}
```

TAD Lista (List ADT)

Posicionar (traverse)

Caso 3:
Existe seguinte a posição corrente;
Posicionar após a posição corrente;
Recuperar ponteiro ao seguinte elemento da posição corrente.




```
if (fromWhere != 0)
{
    if pList->pos->link != NULL;
  1 pList->pos = pList->pos->link;
  2 *dataPtrOut = pList->pos->dataPtr;
    return true;
}
```

TAD Lista (Lista ADT)

Destruir Lista (destroyList)

- A função Destruir Lista (destroyList) considera dois casos.
- O caso mais simples acontece se a lista está vazia. Neste caso, libera-se a memória alocada ao cabeçalho da lista e retorna-se um ponteiro nulo.
- O outro caso acontece se a lista contém elementos. Neste caso, precisamos liberar tanto a memória alocada aos dados de cada nó quanto a memória alocada a estrutura de cada nó.
- Começando pelo primeiro nó, liberamos primeiro o dado e depois a estrutura.
- Após processar todos os nós, libera-se a memória alocada ao cabeçalho da lista e retorna-se um ponteiro nulo.

TAD Lista (Lista ADT)

Destruir Lista (destroyList)

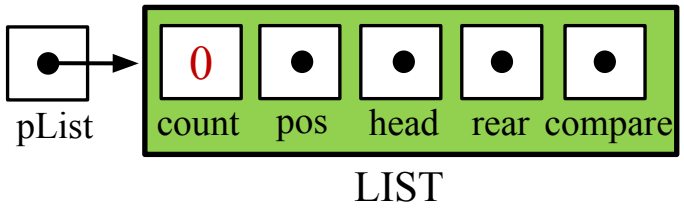
P5-15.h – Função pública Destruir Lista

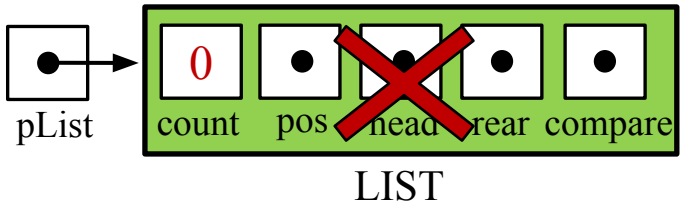
- O código da função é o seguinte:

```
1  /*================== destroyList ================
2    Deletes all data in list and recycles memory
3       Pre    List is a pointer to a valid list.
4       Post   All data and head structure deleted
5       Return null head pointer
6  */
7  LIST* destroyList (LIST* pList)
8  {
9  //Local Definitions
10   NODE* deletePtr;
11
12 //Statements
13   if (pList)
14      {
15       while (pList->count > 0)
16          {
17           // First delete data
18           free (pList->head->dataPtr);
19
20           // Now delete node
21           deletePtr    = pList->head;
22           pList->head  = pList->head->link;
23           pList->count--;
24           free (deletePtr);
25          } // while
26       free (pList);
27      } // if
28   return NULL;
29 } // destroyList
```

TAD Lista (Lista ADT)
Destruir Lista (destroyList)

- A figura ilustra o primeiro caso para a função Destruir lista (destroyList).

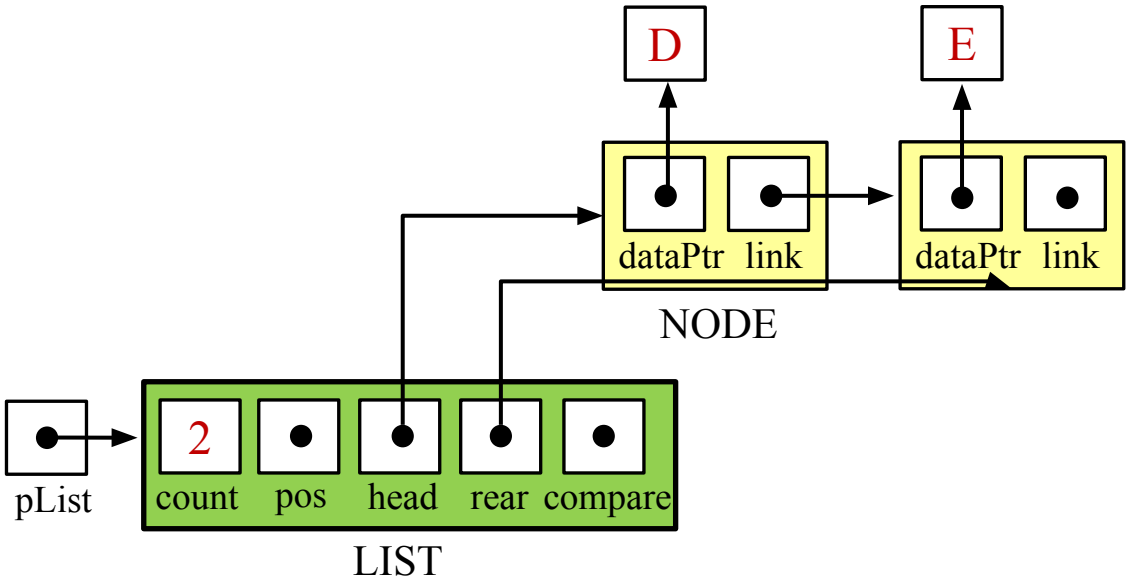





```
free (pList);
Return NULL;
```

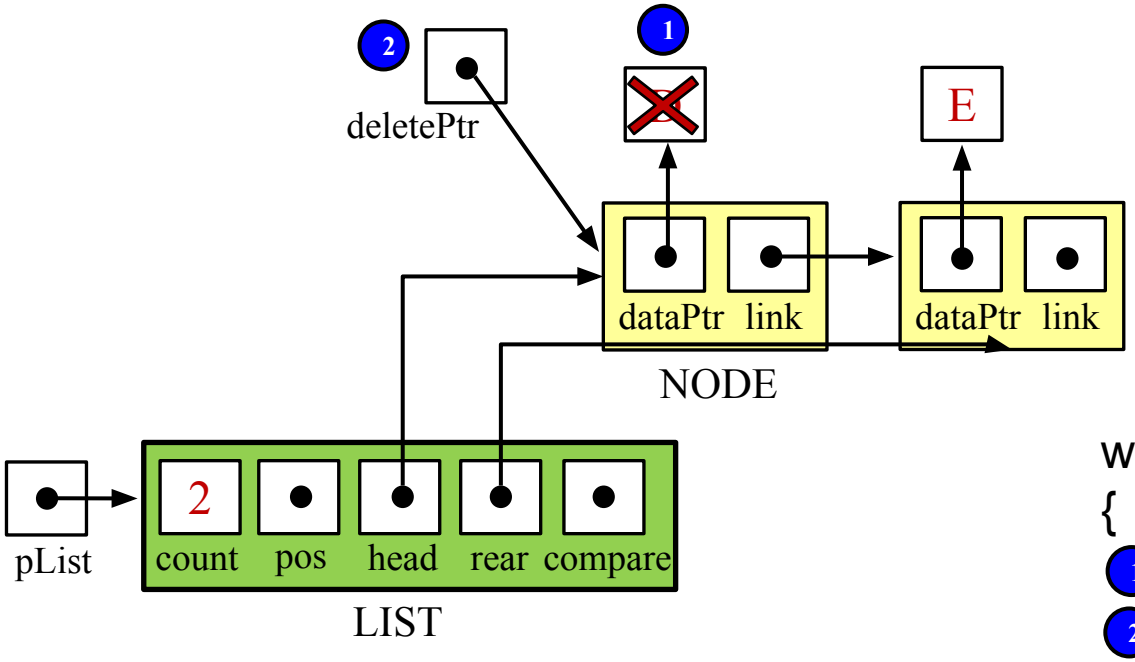
TAD Lista (Lista ADT)
Destruir Lista (destroyList)

- A figura ilustra o segundo caso para a função Destruir Lista (destroyList).



TAD Lista (Lista ADT)
Destruir Lista (destroyList)

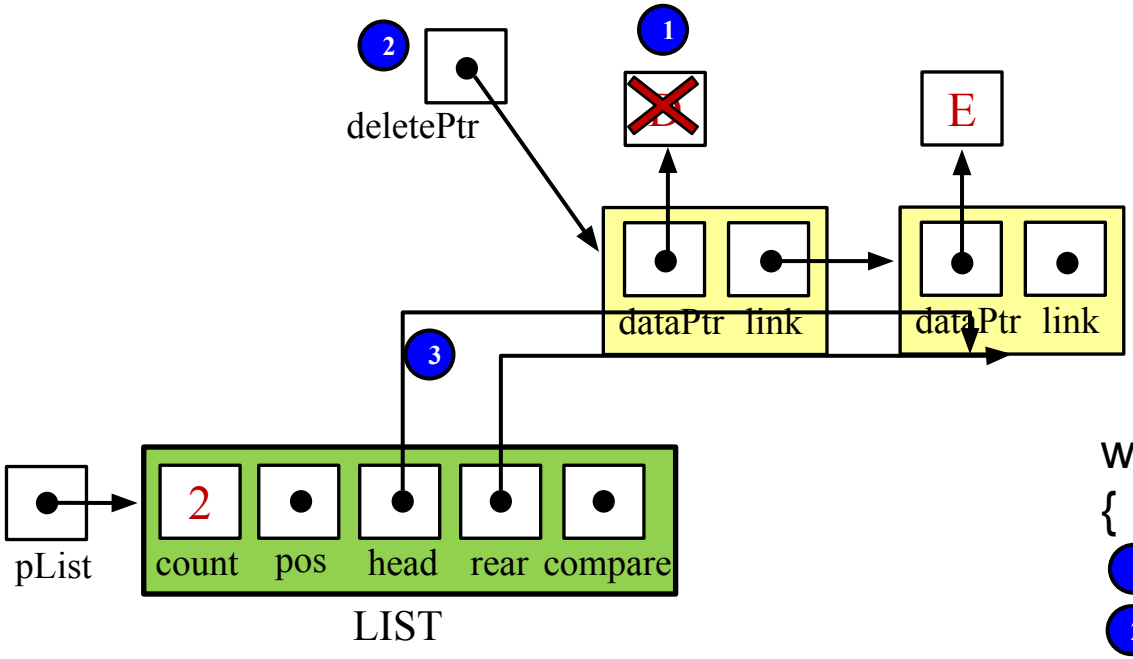
- A figura ilustra o segundo caso para a função Destruir Fila (Destroy Count).




```
while (queue->front != NULL)
{
1   free (pList->head->dataPtr);
2   deletePtr=pList->head);
    pList->head=pList->head->link;
    pList->count--;
    free (deletePtr);
}
free (pList);
return NULL;
```

TAD Lista (Lista ADT)
Destruir Lista (destroyList)

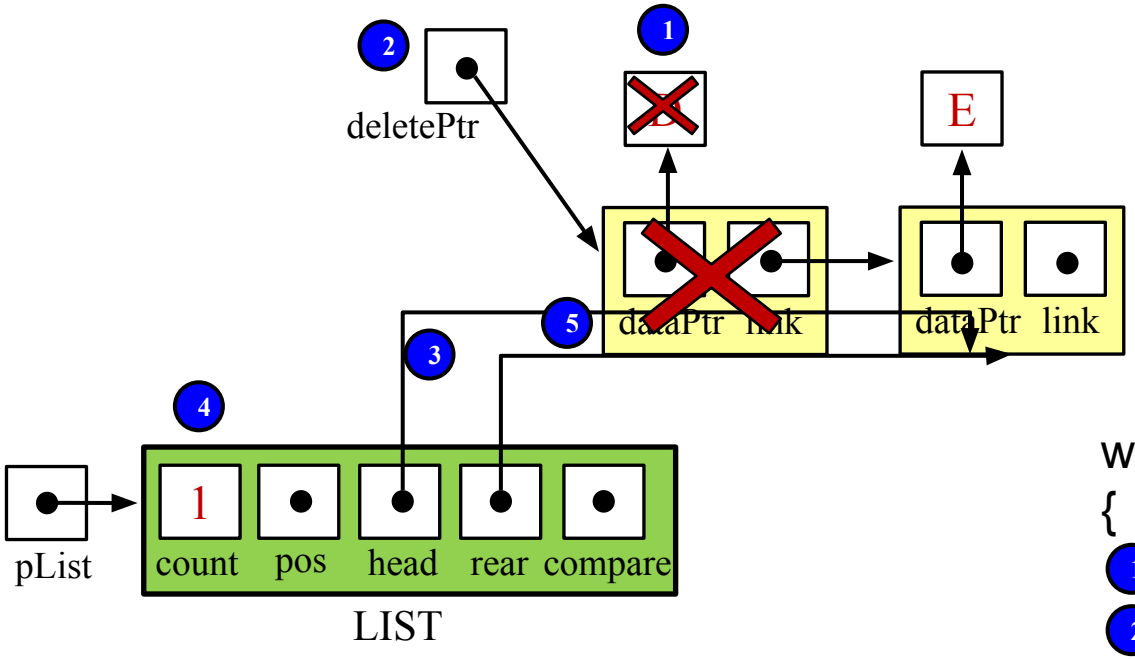
- A figura ilustra o segundo caso para a função Destruir Fila (Destroy Count).




```
while (queue->front != NULL)
{
 1 free (pList->head->dataPtr);
 2 deletePtr=pList->head);
 3 pList->head=pList->head->link;
   pList->count--;
   free (deletePtr);
}
free (pList);
return NULL;
```

TAD Lista (Lista ADT)
Destruir Lista (destroyList)

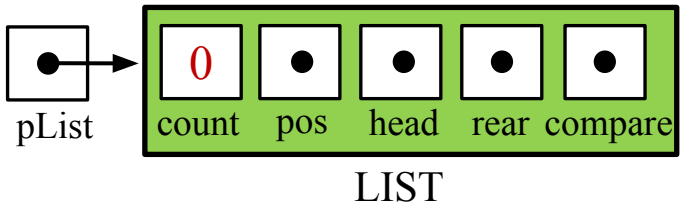
- A figura ilustra o segundo caso para a função Destruir Fila (Destroy Count).



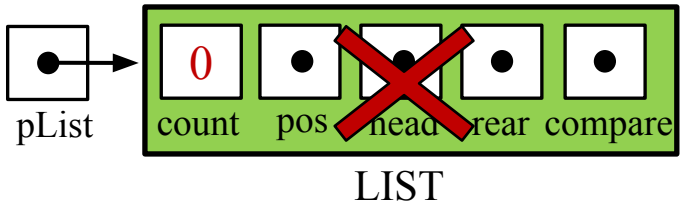

```
while (queue->front != NULL)
{
1 free (pList->head->dataPtr);
2 deletePtr=pList->head);
3 pList->head=pList->head->link;
4 pList->count--;
5 free (deletePtr);
}
free (pList);
return NULL;
```

TAD Lista (Lista ADT)
Destruir Lista (destroyList)

- A figura ilustra o segundo caso para a função Destruir lista (destroyList).



free (pList);
Return NULL;



Referências

- Gilberg, R.F. e Forouzan, B. A. Data Structures_A Pseudocode Approach with C. Lists 5. General Linear Lists. Segunda Edição. Editora Cengage, Thomson Learning, 2005.