MCTA028-15: Programação Estruturada

Aula 12: Listas
(Primeira Parte)

Wagner Tanaka Botelho
wagner.tanaka@ufabc.edu.br / wagtanaka@gmail.com
Universidade Federal do ABC (UFABC)
Centro de Matemática, Computação e Cognição (CMCC)

Introdução

Introdução

- Na Ciência da Computação, uma lista:
  - É uma estrutura de dados linear utilizada para armazenar e organizar dados em um computador;
  - É uma sequência de elementos do **MESMO** tipo;
  - Pode possuir N (N>=0) elementos ou itens;
  - Se N=0, a lista está VAZIA.

- Vamos estudar:
  - Lista sequencial estática (primeira parte);
  - Lista dinâmica encadeada (segunda parte).

Alocação de Memória

Alocação de Memória

- **Alocação estática**:
  - Espaço de memória é alocado no momento da compilação do programa;
  - É necessário definir o **NÚMERO MÁXIMO** de elementos que a lista irá possuir.
- **Alocação dinâmica**:
  - Espaço de memória é alocado em TEMPO DE EXECUÇÃO;
  - A lista CRESCE à medida que novos elementos são armazenados;
  - A lista DIMINUI à medida que elementos são removidos.

Acesso aos Elementos

Acesso aos Elementos

- **Acesso sequencial**:
  - Os elementos são armazenados de forma consecutiva na memória (como um *array* ou vetor);
  - A posição de um elemento pode ser facilmente obtida a partir do **ÍNDICE** da lista.
- **Acesso encadeado**:
  - Cada elemento pode estar em uma área distinta da memória, não necessariamente consecutivas;
  - É necessário que cada elemento da lista armazene, além da sua informação, o endereço de memória onde se encontra o **PRÓXIMO** elemento;
  - Para acessar um elemento, é preciso **PERCORRER** todos os seus antecessores na lista.

Operações Básicas

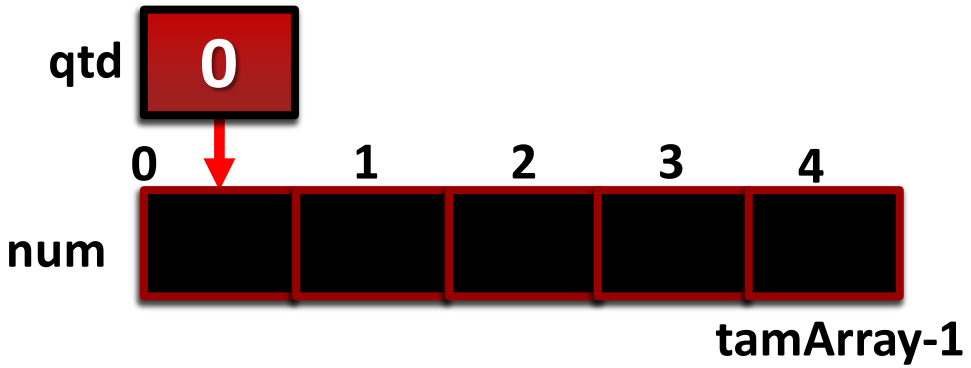
Operações Básicas

- Independente do tipo de alocação e acesso, as seguintes operações básicas podem ser implementadas:
  - Criação da lista;
  - Inserção de um elemento na lista;
  - Remoção de um elemento na lista;
  - Busca por um elemento da lista;
  - Destruição da lista;
  - Informações sobre tamanho, se a lista está cheia ou vazia, etc.

Lista Sequencial Estática

Lista Sequencial Estática

- Uma lista sequencial estática ou lista linear estática é uma lista definida utilizando **ALOCAÇÃO ESTÁTICA** e **ACESSO SEQUENCIAL** dos elementos;
- Considerado o tipo mais SIMPLES de lista possível;
- Definida utilizando um *array*, de modo que o sucessor de um elemento ocupe a posição física seguinte deste;
- Além do *array*, a lista utiliza um **CAMPO ADICIONAL** (qtd) que serve para indicar o quanto do *array* já está ocupado pelos elementos inseridos na lista.



Lista Sequencial Estática: Vantagens e Desvantagens

- **Vantagens**:
  - Acesso rápido e direto aos elementos (índice do *array*);
  - Facilidade para modificar as suas informações;
  - .....
- **Desvantagens**:
  - Definição prévia do tamanho do *array* e, consequentemente, da lista;
  - Dificuldade para inserir e remover um elemento entre outros dois:
    - É necessário deslocar os elementos para abrir espaço dentro do *array.*

Lista Sequencial Estática: Quando Utilizar?

- Em geral, utiliza-se nas seguintes situações:
  - Listas pequenas;
  - Inserção e remoção apenas no final da lista;
  - Tamanho máximo da lista bem definido;
  - ....

Definindo o Tipo

Lista Sequencial Estática: Definindo o Tipo

- Antes de implementar a lista, é necessário definir o TIPO DE DADO que será armazenado nela;
- Um **ponteiro** deve ser criado para a estrutura que define a lista.

Ex_02.c

```c
#include<stdio.h>
#include<stdlib.h>

#define tamArray 5

struct lista{
    int qtd;
    int num[tamArray];
};

typedef struct lista Lista;
```

Definindo uma constante (tamanho do *array*).

Definindo o tipo lista com dois campos:

+ qtd (int): indica quantidade de elementos inseridos na lista

+ *array* num do tipo int: tipo de dado a ser armazenado na lista.

Redefinindo a *struct* para encurtar o comando.

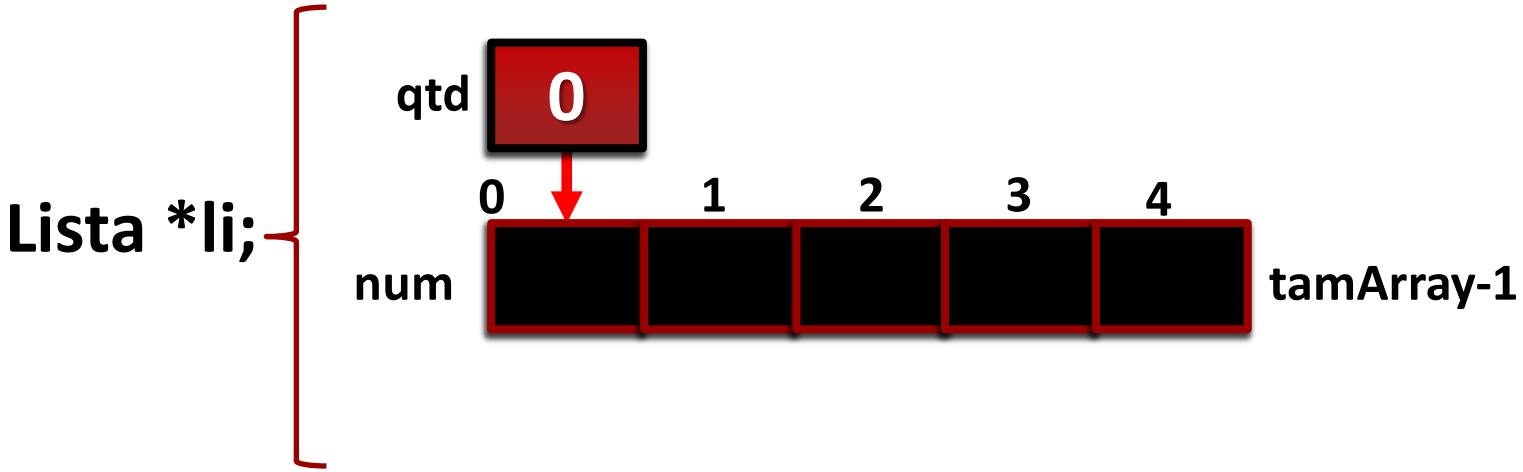
Criando a Lista

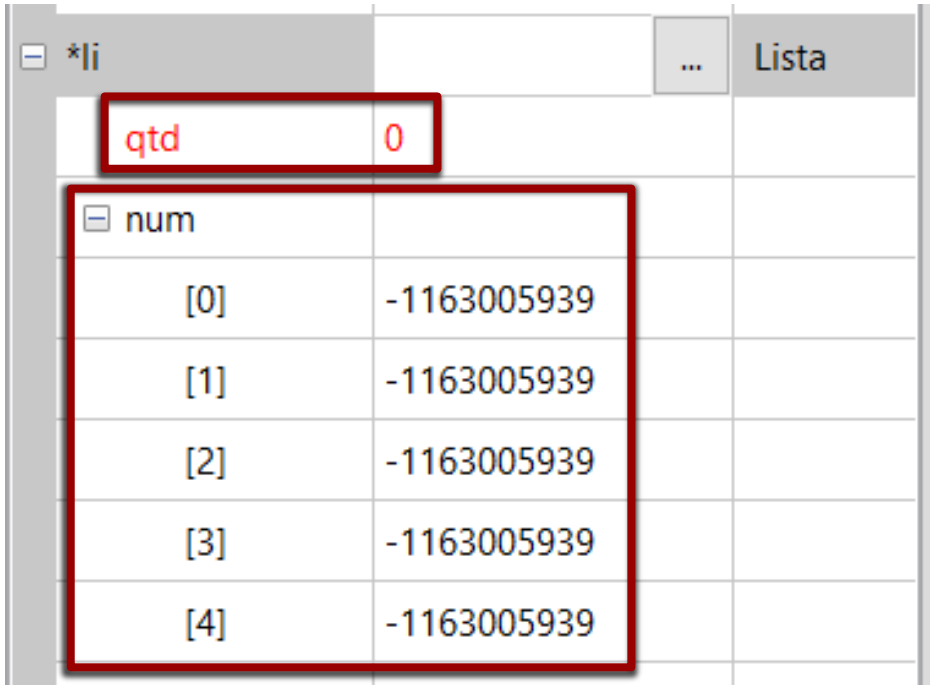
```
19  Lista *cria_Lista(){
20      Lista *li;
21      li = (Lista*) malloc(sizeof(Lista));
22
23      if(li!=NULL){
24          li->qtd=0;
25      }
26      return li;
27  }
```

Ponteiro para estrutura lista.

Alocando a área de memória para a lista.

= (*li).qtd=0; **armazena a quantidade de elementos inseridos na lista.**

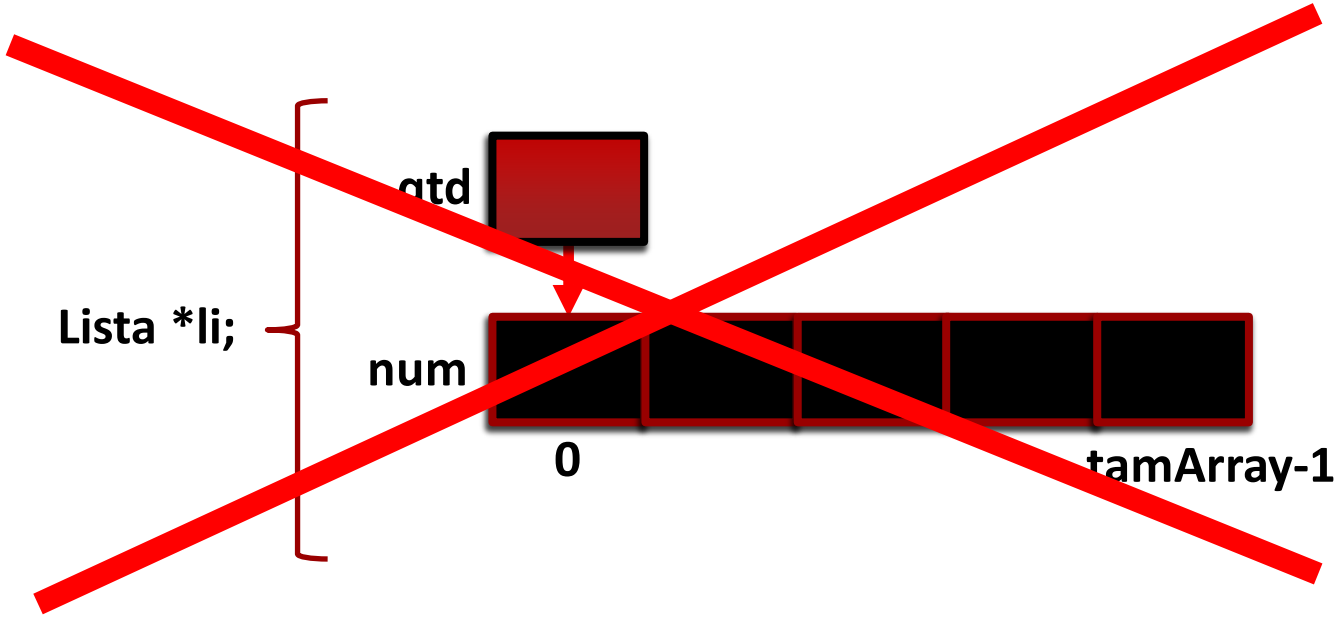




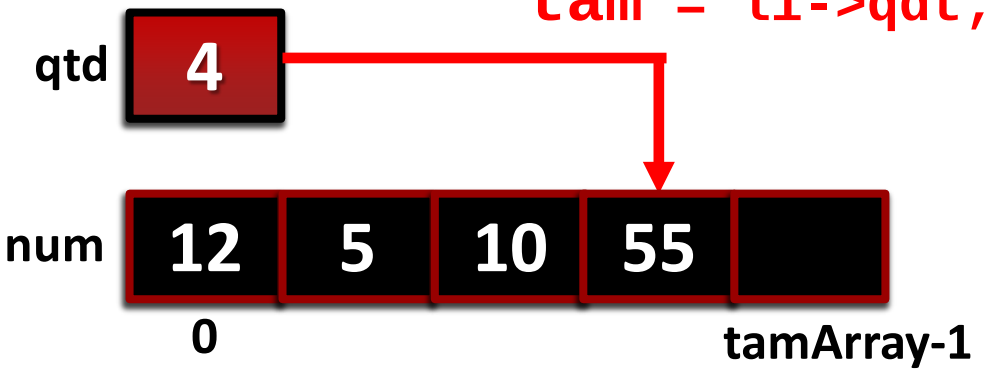
“Destruindo” a Lista

```
33  void libera_lista(Lista *li){
34      free(li);
35  }
```

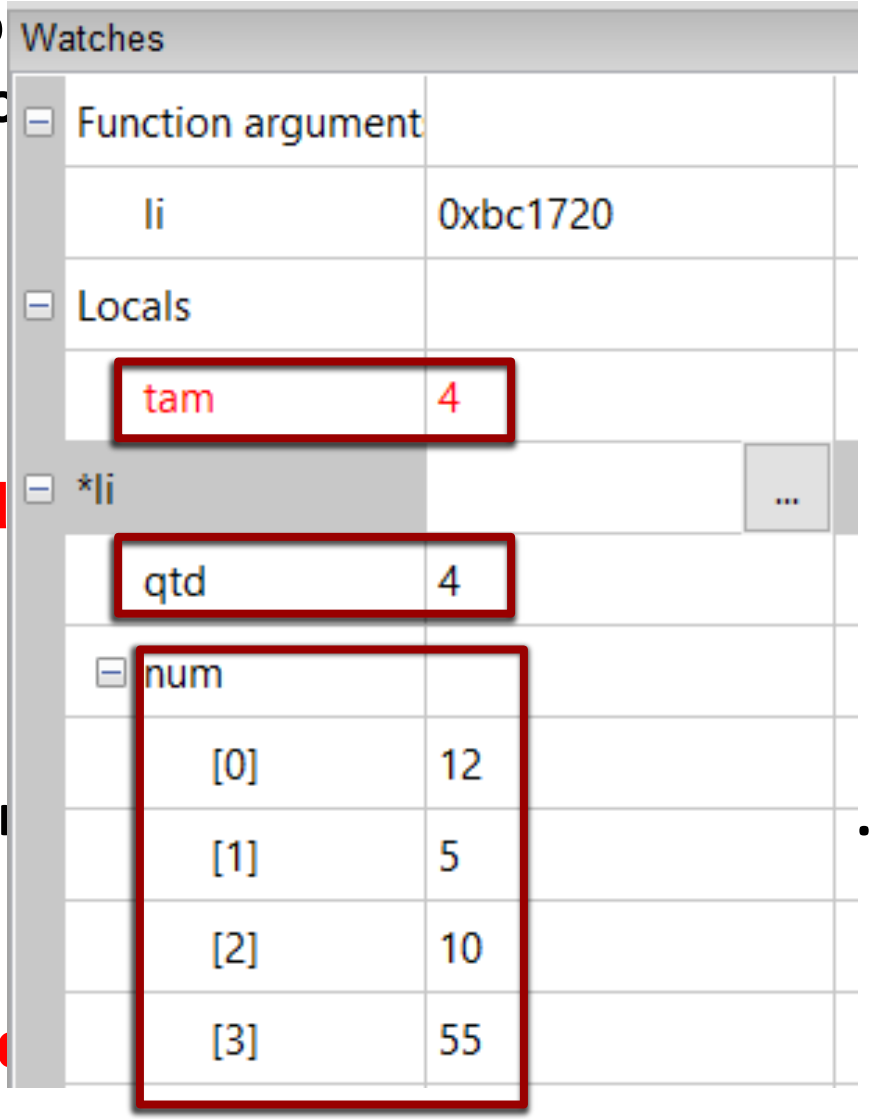
Liberando a memória alocada para a estrutura que representa a lista.



Tamanho da Lista



`tam = li->qdt;` → Indica o quanto do
elementos inserido



Para saber o tamanho da lista, basta retornar o valor de `qtd`

```
37  int tamanho_Lista(Lista* li){
38      int tam=0;
39
40      if(li == NULL){
41          return -1;
42      }
43      else{
44          tam = li->qtd;
45          return tam;
46      }
47  }
```

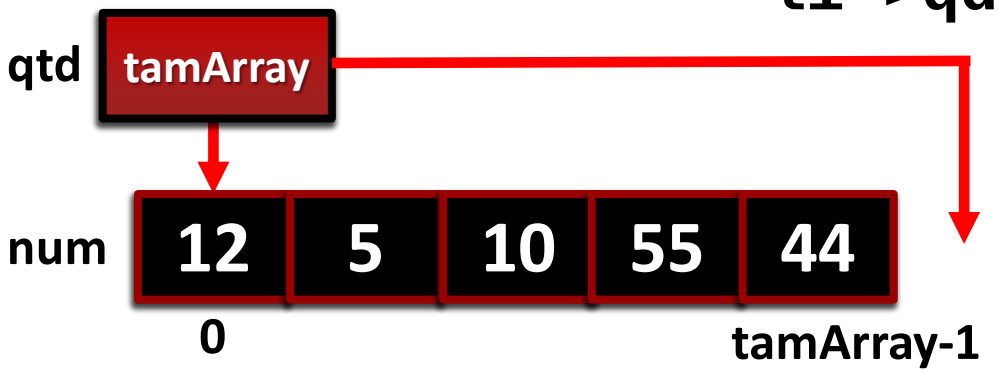
→ Se for verdade, algum proble

→ `tam` recebe o valor de `qt`

Lista Cheia

Lista Cheia:

`li->qdt == tamArray`




```
49  int lista_Cheia(Lista *li){
50      if(li == NULL){
51          return -1;
52      }
53      else{
54          if(li->qtd == tamArray){
55              return 1;
56          }
57          else{
58              return 0;
59          }
60      }
61  }
```

Se ocorreu algum problema na criação da lista, o valor retornado será -1.

A variável **qtd** também é utilizada para saber se a lista está cheia. Basta verificar se **qtd =** ao tamanho do *array* (**tamArray**).

Se a lista não estiver cheia, o valor retornado será 0.

Lista Vazia

Lista Vazia:

li->qdt == 0

```
63  int lista_Vazia(Lista *li){
64      if(li == NULL){
65          return -1;
66      }
67      else{
68          if(li->qtd == 0){
69              return 1;
70          }
71          else{
72              return 0;
73          }
74      }
75  }
```

Se ocorreu algum problema na criação da lista, o valor retornado será -1.

A variável qtd também é utilizada para saber se a lista está vazia. Basta verificar se `qtd = 0`.

Se a lista não estiver vazia, o valor retornado será 0.

Inserindo na Lista

```
68  int insere_lista(Lista *li, int n){
69      if(li == NULL){
70          return 0;
71      }
72      else{
73          if(li->qtd == tamArray){
74              return 0;
75          }
76          else{
77              li->num[li->qtd] = n;
78              li->qtd++;
79              return 1;
80          }
81      }
82  }
```

Se ocorreu algum problema na criação da lista, o valor retornado será 0.

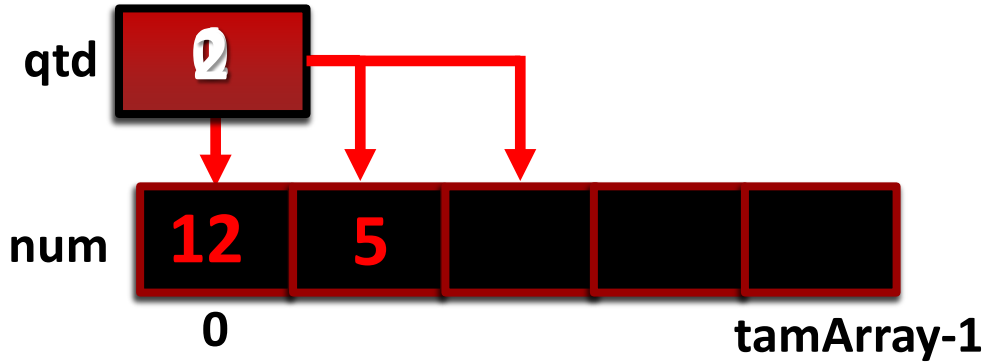
Se a lista estiver cheia, o valor retornado será 0.

Inserindo na próxima posição livre do *array*.

Incrementando a quantidade de elementos.

insere_lista(pList, 12);
- li->num[0]=12
- li->qtd++

insere_lista(pList, 5);
- li->num[1]=5
- li->qtd++



Inserindo no Final da Lista

Inserindo no Final da Lista

- Diferente da inserção no início, a inserção no **FINAL** de uma lista sequencial estática NÃO necessita que se mude o lugar dos demais elementos da lista;
- Deve-se inserir o elemento após a última posição ocupada do *array* que representa a lista.

```
84  int insere_lista_final(Lista *li, int n){
85      if(li == NULL){
86          return 0;
87      }
88      else{
89          if(li->qtd == tamArray){
90              return 0;
91          }
92          else{
93              li->num[li->qtd] = n;
94              li->qtd++;
95              return 1;
96          }
97      }
98  }
```

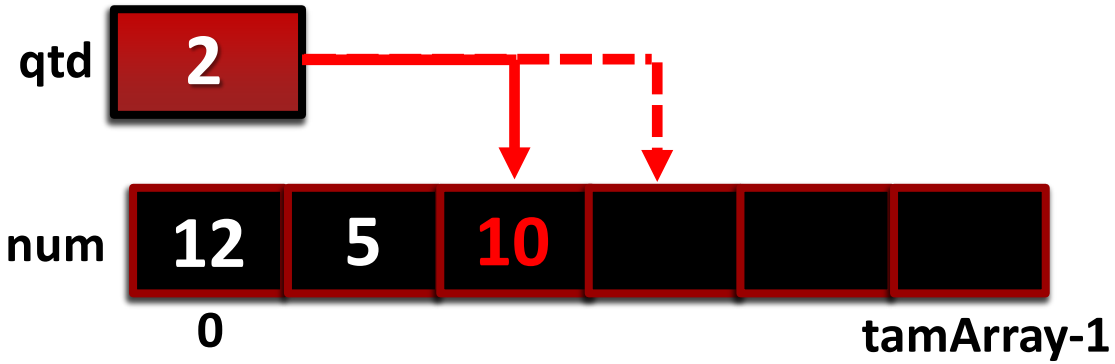
Se ocorreu algum problema na criação da lista, o valor retornado será 0.

Se a lista estiver cheia, o valor retornado será 0.

Inserindo na próxima posição livre do *array*.

Incrementando a quantidade de elementos.

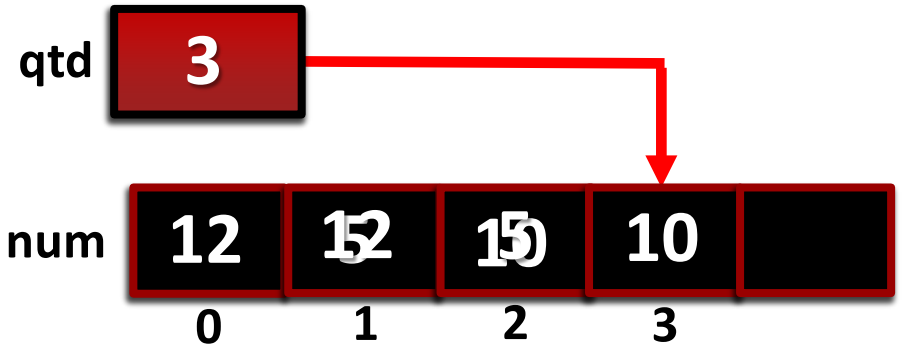
insere_lista_final(pList, 10);
- li->num[2]=10
- li->qtd++



Inserindo no Início da Lista

Inserindo no Início da Lista

- A inserção no início de uma lista sequencial estática necessita que se mude o lugar dos demais elementos da lista:
  - Isso deixa o início da lista (a posição ZERO do *array*) livre para inserir um novo elemento.



- Todos elementos da lista devem ser PERCORRIDOS e COPIADOS para uma posição para frente:
  - Isso deve ser feito do ÚLTIMO ELEMENTO até o PRIMEIRO, evitando, assim, que a cópia de um elemento sobrescreva o outro.

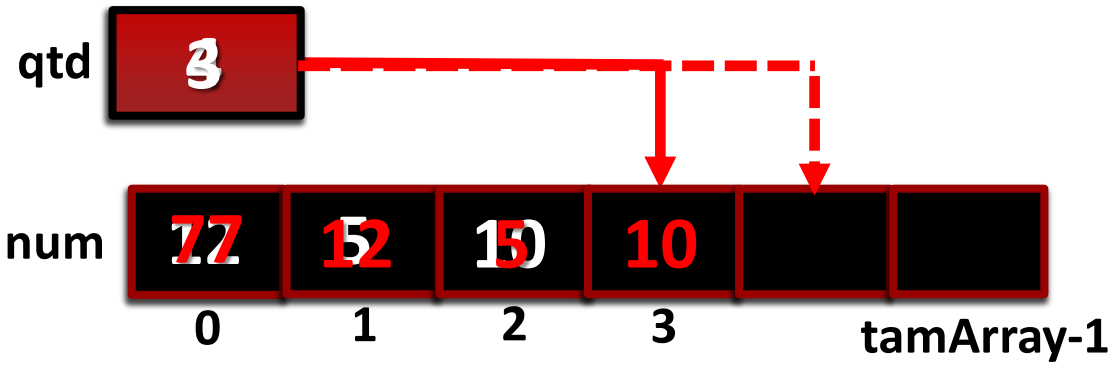
```
100 int insere_lista_inicio(Lista *li, int n){
101     if(li == NULL){
102         return 0;
103     }
104     else{
105         if(li->qtd == tamArray){
106             return 0;
107         }
108         else{
109             for(int i=li->qtd-1; i >= 0; i--){
110                 li->num[i+1] = li->num[i];
111             }
112             li->num[0] = n;
113             li->qtd++;
114         }
115     }
116 }
```

Desloca os elementos uma posição para frente.

Insere o elemento no início da lista.

insere_lista_inicio(pList, 77);

Linha	i	n	li->num[]
109	2		
110			[3]=10
109	1		
110			[2]=5
109	0		
110			[1]=12
109	-1		



Removendo do Início da Lista

Removendo do Início da Lista

- Como na inserção, a remoção de um elemento do início de uma lista sequencial estática necessita que se mude o lugar dos demais elementos da lista;
- Basicamente, deve-se movimentar todos os elementos da lista uma POSIÇÃO PARA TRÁS dentro do *array*:
  - Esta ação sobrescreve o início da lista (a posição ZERO do array);
  - Ao mesmo tempo, a quantidade de elementos diminuiu.

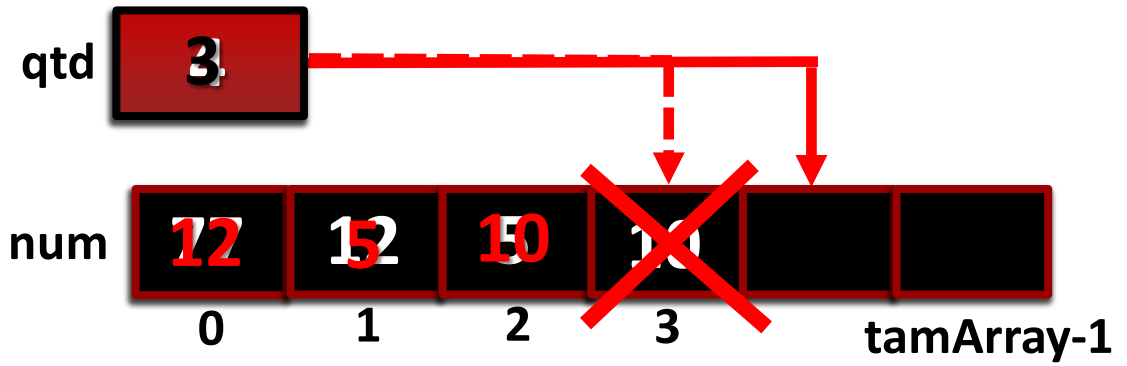
```
118 int remove_lista_inicio(Lista *li){
119     if(li == NULL){
120         return 0;
121     }
122     else{
123         if(li->qtd == 0){
124             return 0;
125         }
126         else{
127             for(int k=0; k < li->qtd-1; k++){
128                 li->num[k] = li->num[k+1];
129             }
130             li->qtd--;
131             return 1;
132         }
133     }
134 }
```

Desloca os elementos uma posição para trás.

Deve-se diminuir em uma unidade a quantidade de elementos armazenados na lista.

remove_lista_inicio(pList);

Linha	k	n	li->num[]
127	0		
128			[0]=12
127	1		
128			[1]=5
127	2		
128			[2]=10
127	3		



O último elemento da lista fica duplicado. Entretanto, está posição é considerada NÃO ocupada por elementos na lista.

Removendo do Final da Lista

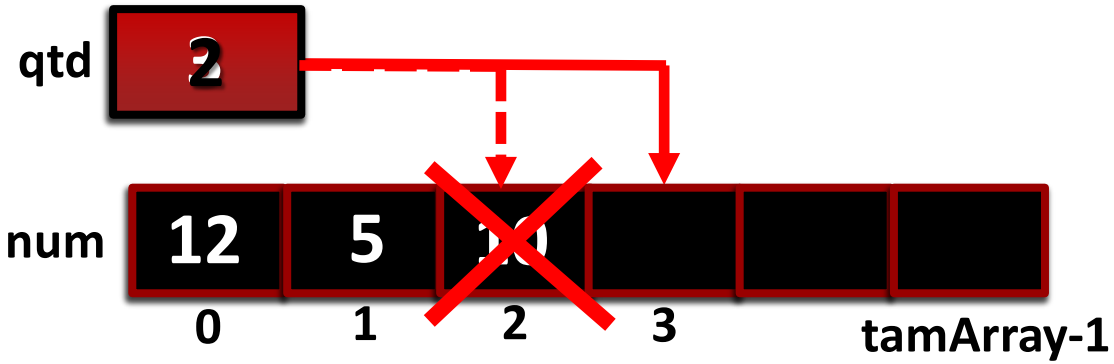
Removendo do Final da Lista

- Diferente da remoção do início, a remoção do final de uma lista sequencial estática não necessita que se mude o lugar dos demais elementos da lista:
  - Entretanto, deve-se alterar a quantidade de elementos na lista.
- O elemento removido continua no final da lista:
  - Isso não é um problema, pois a posição é considerada não ocupada por elementos da lista.

```
136 int remove_lista_final(Lista *li){
137     if(li == NULL){
138         return 0;
139     }
140     else{
141         if(li->qtd == 0){
142             return 0;
143         }
144         else{
145             li->qtd--;
146             return 1;
147         }
148     }
149 }
```

Alterar a quantidade de elementos na lista.

remove_lista_final(pList);



O elemento removido continua no final da lista. Entretanto, está posição é considerada NÃO ocupada por elementos na lista.

Removendo um Elemento Específico da Lista

Removendo um Elemento Específico da Lista

- Deve-se procurar o elemento a ser removido na lista, o qual pode estar no início, no meio ou no final da lista:
  - No início ou meio, e preciso mudar, após a remoção, o lugar dos demais elementos da lista.
- Basicamente, deve-se procurar o elemento da lista e movimentar **TODOS** os elementos que estão à frente na lista uma posição para trás dentro do *array*:
  - Isso sobrescreve o elemento a ser removido, ao mesmo tempo que diminui o número de elementos.

```
151 int remove_lista_elemento(Lista *li, int elem){
152     int k = 0, i = 0;
153     if(li == NULL){
154         return 0;
155     }
156     else{
157         if(li->qtd == 0){
158             return 0;
159         }
160         else{
161             while(i < li->qtd && li->num[i] != elem){
162                 i++;
163             }
164             if(i == li->qtd){
165                 return 0;
166             }
167             for(k=i; k < li->qtd-1; k++){
168                 li->num[k] = li->num[k+1];
169             }
170             li->qtd--;
171             return 1;
172         }
173     }
174 }
```

Procurando o elemento (elem) no *array*.

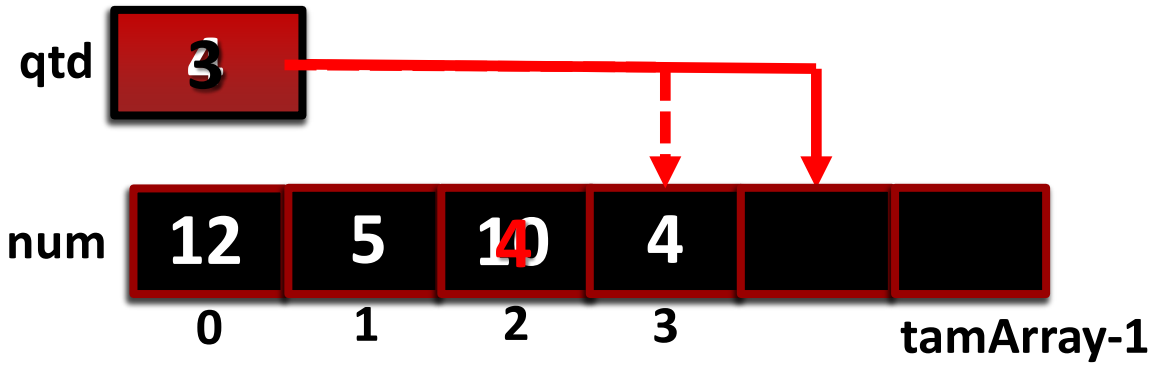
Se verdade, o elemento não foi encontrado no *array*.

Desloca os elementos uma posição para trás.

```
151 int remove_lista_elemento(Lista *li, int elem){
152     int k = 0, i = 0;
153     if(li == NULL){
154         return 0;
155     }
156     else{
157         if(li->qtd == 0){
158             return 0;
159         }
160         else{
161             while(i < li->qtd && li->num[i] != elem){
162                 i++;
163             }
164             if(i == li->qtd){
165                 return 0;
166             }
167             for(k=i; k < li->qtd-1; k++){
168                 li->num[k] = li->num[k+1];
169             }
170             li->qtd--;
171             return 1;
172         }
173     }
174 }
```

remove_lista_elemento(pList, 10);

Linha	i	k	elem	li->num[]
152	0	0		
162	1			
162	2			
167		2		
168				[2]=4
167		3		



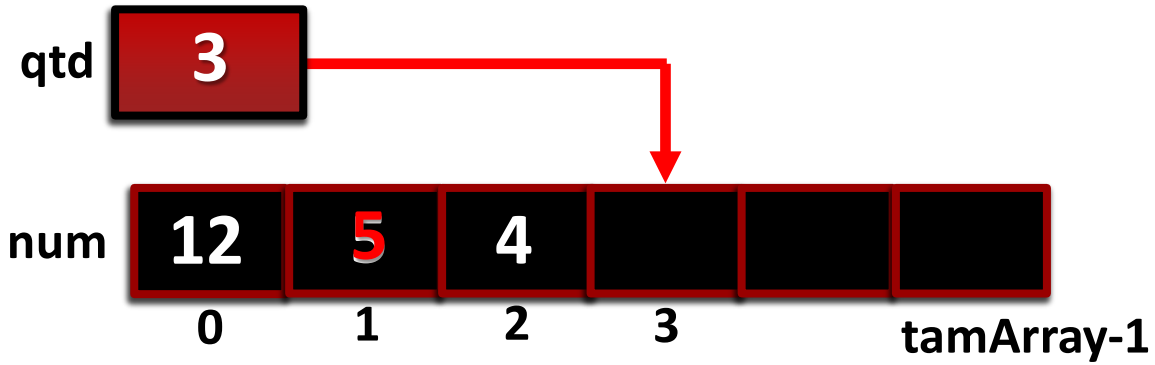
Busca por Posição na Lista

```
176 int busca_lista_pos(Lista *li, int pos){
177     int resp=0;
178     if(li == NULL || pos <= 0 || pos > li->qtd){
179         exit(0);
180     }
181     else{
182         resp = li->num[pos-1];
183         return resp;
184     }
185 }
```

Se o ponteiro li==NULL ou posição buscada (pos) é negativa ou posição (pos) é maior que o tamanho da lista.

resp recebe o elemento armazenado na posição (pos) recebida como parâmetro.

busca_lista_pos(pList, 2); → resp = li->num[1]



Busca por Conteúdo na Lista

```
187 int busca_lista_conteudo(Lista *li, int conteu){
188     int i = 0, resp = 0;
189     if(li == NULL){
190         return 0;
191     }
192     else{
193         while(i < li->qtd && li->num[i] != conteu){
194             i++;
195         }
196 
197         if(i == li->qtd){
198             return 0;
199         }
200         else{
201             return i;
202         }
203     }
204 }
```

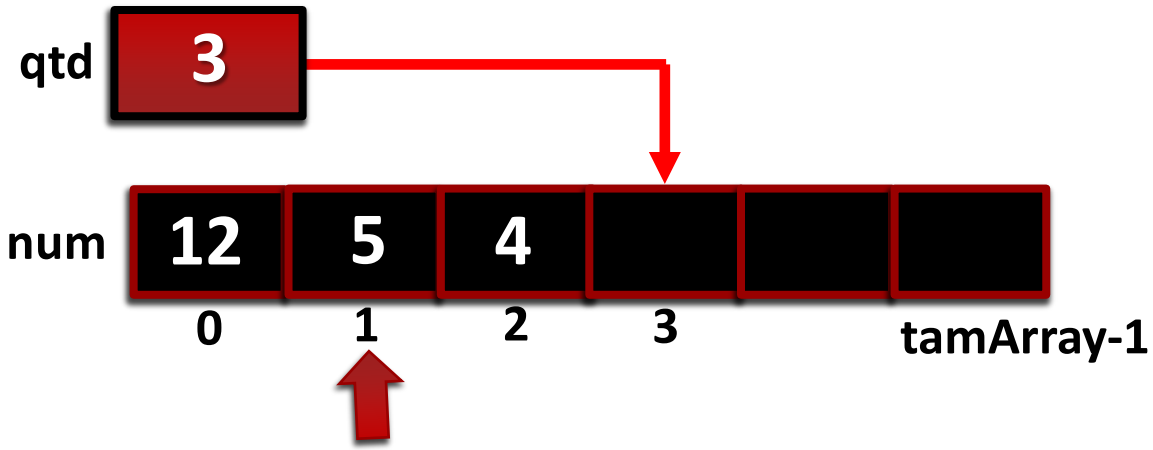
Procurando o elemento (elem) no *array*.

Se verdade, o elemento não foi encontrado no *array*.

Elemento encontrado, a sua posição é retornada.

busca_lista_pos(pList, 5);

Linha	i	resp	li->qtd	ii->num[]
188	0	0		
193			3	12
194	1			
201	1			



Referências

- Slides do Prof. Luiz Rozante;
- SALES, André Barros de; AMVAME-NZE, Georges. Linguagem C: roteiro de experimentos para aulas práticas. 2016;
- BACKES, André. Linguagem C Completa e Descomplicada. Editora Campus. 2013;
- SCHILDT, Herbert. C Completo e Total. Makron Books. 1996;
- DAMAS, Luís. Linguagem C. LTC Editora. 1999;
- DEITEL, Paul e DEITEL, Harvey. C Como Programar. Pearson. 2011;
- BACKES, André. Estrutura de Dados Descomplicada em Linguagem C. GEN LTC. 2016.