

# Estruturas de Dados

## Lista Encadeada

Departamento de Computação  
Prof. Martín Vigil  
Adaptado de

2020.1



UNIVERSIDADE FEDERAL  
DE SANTA CATARINA

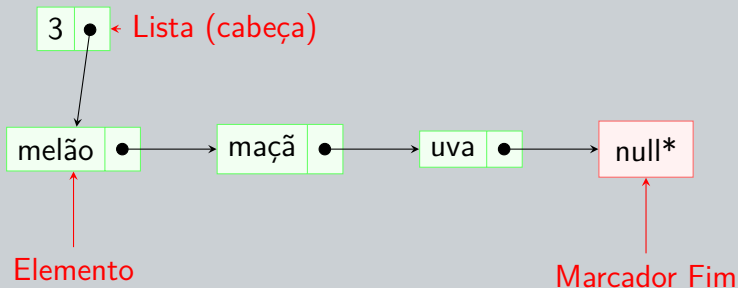
# Definição de Lista

- É uma ***sequência*** de dados
- Exemplos:
  - <Pedro, João, Maria, Ivan>
  - <5, 9, 70, 3>
  - <😊, 📞, ⚡, 🎵>

# Definições de Lista Encadeada

- É uma estrutura de dados que simula uma lista de dados
- É composta por objetos (instâncias) do tipo ***Elemento***
- Cada elemento
  - contém um dado da lista
  - referencia o próximo elemento
  - é alocado dinamicamente somente quando necessário
- Usa-se um objeto *cabeça* da ***Lista*** para:
  - Indicar a quantidade de elementos na lista
  - Referenciar o primeiro elemento da Lista

# Lista Encadeada



# Modelagem de Lista

- Aspecto Estrutural:
  - Necessitamos um ponteiro para o primeiro elemento da lista;
  - Necessitamos um inteiro para indicar quantos elementos a lista possui.

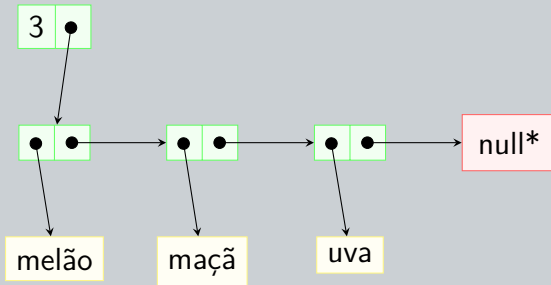
```
estrutura Lista {  
    Elemento* _primeiro;  
    int _quantidade;  
};
```

# Modelagem do Elemento de Lista

- Aspecto Estrutural:
  - Necessitamos um ponteiro para o próximo elemento da lista;
  - Necessitamos um ponteiro **genérico**  $T^*$  para o dado que vamos armazenar.
  - O dado necessita de um destrutor próprio, assim como a lista vai precisar de um também;

```
estrutura Elemento {  
    Elemento* _proximo;  
    T* _dado;  
};
```

# Modelagem de Lista Encadeada



# Modelagem da Lista Encadeada

- Aspecto Funcional:
  - Temos que colocar e retirar dados da lista;
  - Temos que testar se a lista está vazia (dentre outros testes);
  - Temos que inicializar a lista e garantir a ordem de seus elementos.



# Modelagem da Lista Encadeada

- Inicializar ou limpar:
  - `Lista* iniciaLista();`
  - `destroiLista(Lista* umaLista);`
- Testar se a lista está vazia ou cheia e outros testes:
  - `bool listaVazia(Lista* umaLista);`
  - `int posicao(Lista* umaLista, T* umDado);`
  - `bool contem(Lista* umaLista, T* umDado);`

# Modelagem da Lista Encadeada

- Colocar e retirar dados da lista:
  - adiciona(Lista\* umaLista, T\* umDado);
  - adicionaNoInicio(Lista\* umaLista, T\* umDado);
  - adicionaNaPosicao(Lista\* umaLista, T\* umDado, int umaPosicao);
  - adicionaNoFim(Lista\* umaLista, T\* umDado);
  - T\* retiraDoInicio(Lista\* umaLista);
  - T\* retiraDaPosicao(Lista\* umaLista, int umaPosicao);
  - T\* retiraDoFim(Lista\* umaLista);
  - T\* retiraEspecifico(Lista\* umaLista, T\* umDado);

## Função *iniciaLista*

- Inicializamos o ponteiro para nulo;
- Inicializamos o tamanho para “0”;
- Complexidade de tempo:  $\Theta(1)$

```
iniciaLista()  
inicio  
    Lista* umaLista <- ALOQUE(Lista);  
    umaLista._primeiro <- null;  
    umaLista._quantidade <- 0;  
    RETORNE umaLista;  
fim;
```

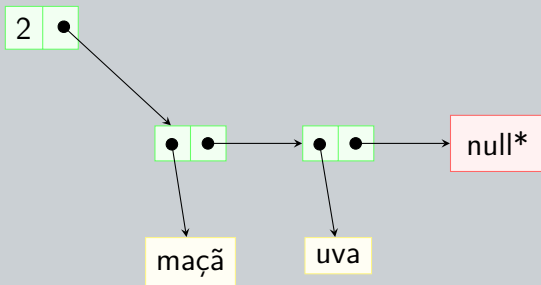
## Função *listaVazia*

```
bool listaVazia(Lista* umaLista)
inicio
SE (umaLista._quantidade = 0) ENTAO
    RETORNE(Verdadeiro)
SENAO
    RETORNE(Falso);
fim;
```

- Complexidade de tempo:  $\Theta(1)$

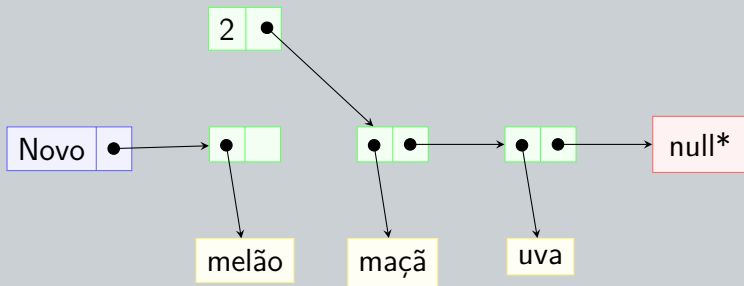
## Função *adicionaNoInicio*

- 1 Teste se é possível alocar um novo elemento;
- 2 Faça o próximo do novo elemento ser o primeiro da lista;
- 3 Faça a cabeça de lista apontar para o novo elemento.



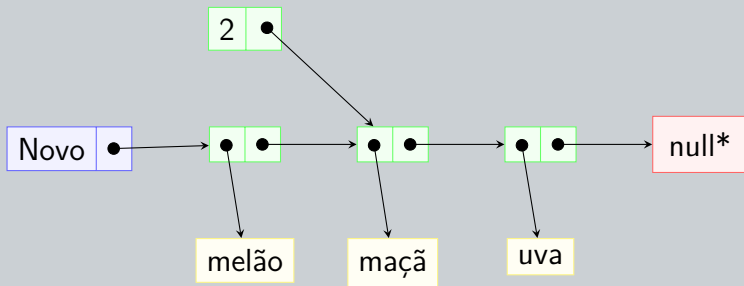
## Função *adicionaNoInicio*

- 1 Teste se é possível alocar um novo elemento;
- 2 Faça o próximo do novo elemento ser o primeiro da lista;
- 3 Faça a cabeça de lista apontar para o novo elemento.



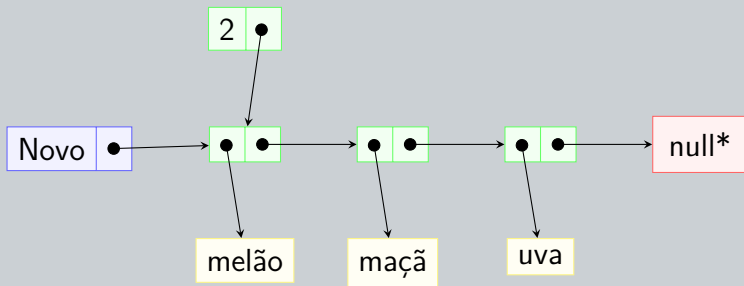
## Função *adicionaNoInicio*

- 1 Teste se é possível alocar um novo elemento;
- 2 Faça o próximo do novo elemento ser o primeiro da lista;
- 3 Faça a cabeça de lista apontar para o novo elemento.



## Função *adicionaNoInicio*

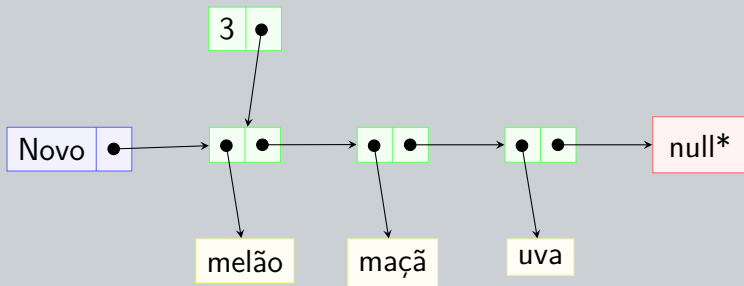
- 1 Teste se é possível alocar um novo elemento;
- 2 Faça o próximo do novo elemento ser o primeiro da lista;
- 3 Faça a cabeça de lista apontar para o novo elemento.





## Função *adicionaNoInicio*

- 1 Teste se é possível alocar um novo elemento;
- 2 Faça o próximo do novo elemento ser o primeiro da lista;
- 3 Faça a cabeça de lista apontar para o novo elemento.



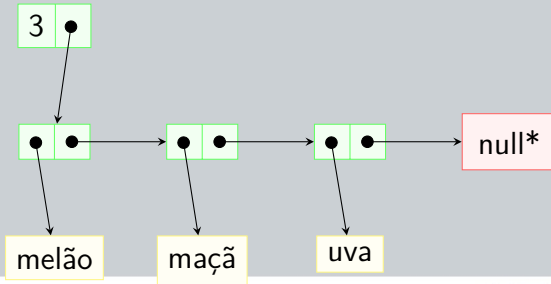
## Função *adicionaNoInicio*

```
adicionaNoInicio(Lista* umaLista, T* umDado)
inicio
  Elemento* novo <- ALOQUE(Elemento);
  SE (novo = NULO) ENTAO
    THROW(ERROSEMMEMORIA);
  SENA0
    novo._proximo <- umaLista._primeiro;
    novo._dado <- umDado;
    umaLista._primeiro <- novo;
    umaLista._quantidade <- umaLista._quantidade +
      1;
  FIM SE
fim;
```

- Complexidade de tempo: ?

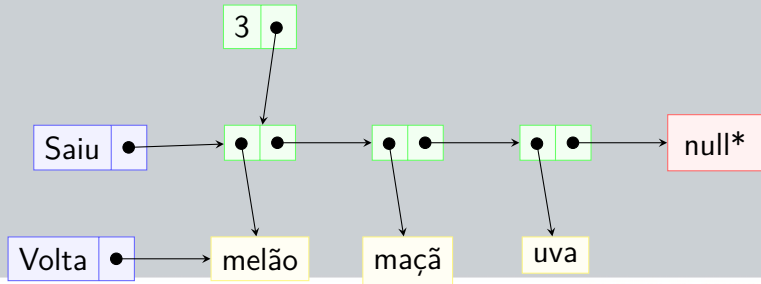
# Função *retiraDoInicio*

- Verifique que a lista não está vazia;
- Decremente o tamanho;
- Faça a lista apontar para o segundo elemento
- Desaloque a memória do primeiro elemento;
- Devolva o dado retirado.



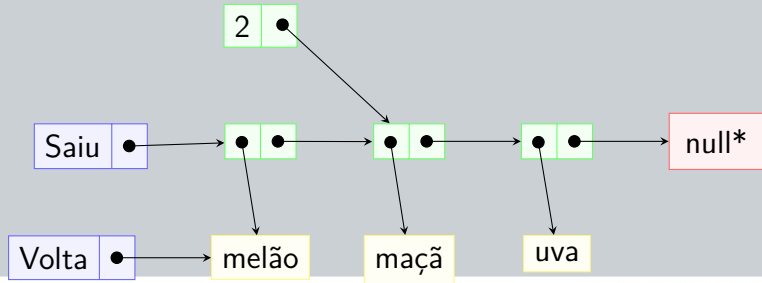
# Função *retiraDoInicio*

- Verifique que a lista não está vazia;
- Decremente o tamanho;
- Faça a lista apontar para o segundo elemento
- Desaloque a memória do primeiro elemento;
- Devolva o dado retirado.



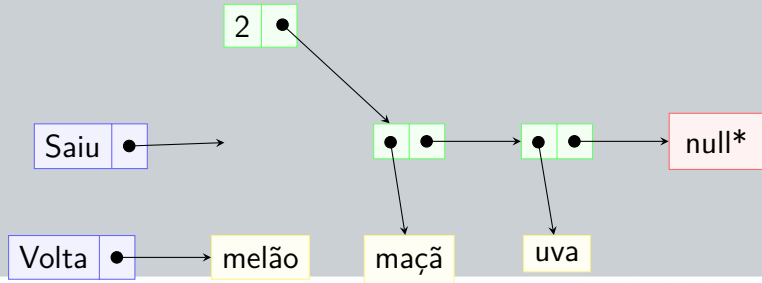
# Função *retiraDoInicio*

- Verifique que a lista não está vazia;
- Decremente o tamanho;
- Faça a lista apontar para o segundo elemento
- Desaloque a memória do primeiro elemento;
- Devolva o dado retirado.



# Função *retiraDoInicio*

- Verifique que a lista não está vazia;
- Decremente o tamanho;
- Faça a lista apontar para o segundo elemento
- Desaloque a memória do primeiro elemento;
- Devolva o dado retirado.



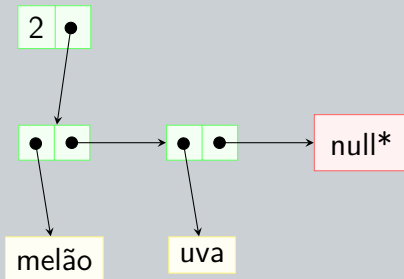
## Função *retiraDoInicio*

```
T* retiraDoInicio(Lista* umaLista)
Elemento* saiu; //Variável auxiliar elemento.
T* volta; //Variável auxiliar tipo T.
início
SE (listaVazia(umaLista)) ENTAO
    THROW(ERROLISTAVAZIA);
SENAO
    saiu <- umaLista._primeiro;
    volta <- saiu._dado;
    umaLista._primeiro <- saiu._proximo;
    umaLista._quantidade <- umaLista._quantidade -
        1;
    DESALOQUE(saiu);
    RETORNE(volta);
FIM SE
fim;
```

- Complexidade de tempo: ?

## Função *adicionaNaPosicao*

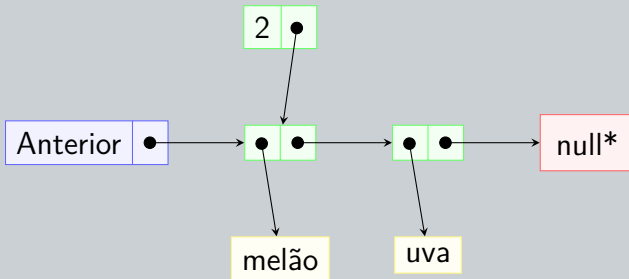
- Testamos se a posição existe e se é possível alocar;
- Caminhamos até a posição;
- Adicionamos o novo dado na posição;
- Incrementamos o tamanho.





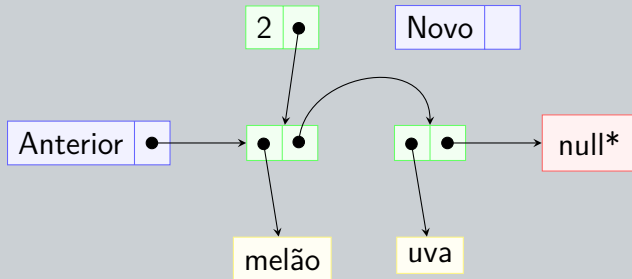
## Função *adicionaNaPosicao*

- Testamos se a posição existe e se é possível alocar;
- Caminhamos até a posição;
- Adicionamos o novo dado na posição;
- Incrementamos o tamanho.



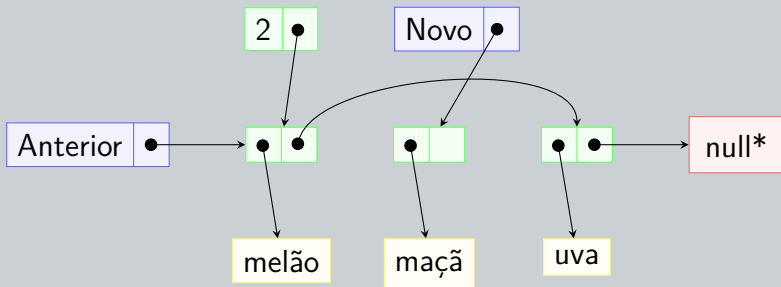
## Função *adicionaNaPosicao*

- Testamos se a posição existe e se é possível alocar;
- Caminhamos até a posição;
- Adicionamos o novo dado na posição;
- Incrementamos o tamanho.



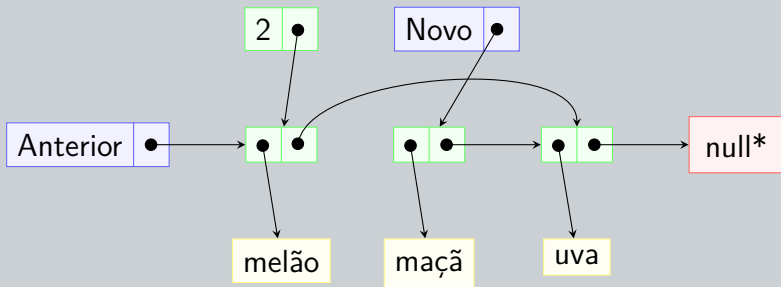
## Função *adicionaNaPosicao*

- Testamos se a posição existe e se é possível alocar;
- Caminhamos até a posição;
- Adicionamos o novo dado na posição;
- Incrementamos o tamanho.



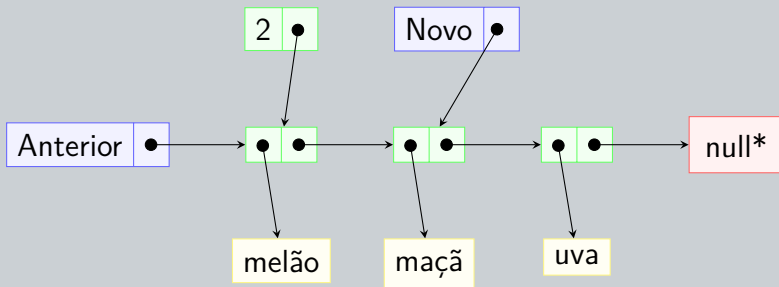
## Função *adicionaNaPosicao*

- Testamos se a posição existe e se é possível alocar;
- Caminhamos até a posição;
- Adicionamos o novo dado na posição;
- Incrementamos o tamanho.



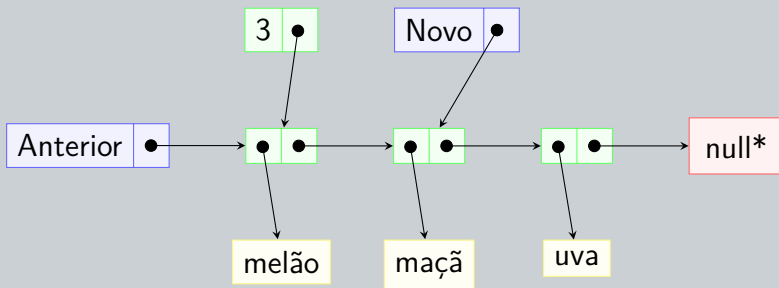
## Função *adicionaNaPosicao*

- Testamos se a posição existe e se é possível alocar;
- Caminhamos até a posição;
- Adicionamos o novo dado na posição;
- Incrementamos o tamanho.



## Função *adicionaNaPosicao*

- Testamos se a posição existe e se é possível alocar;
- Caminhamos até a posição;
- Adicionamos o novo dado na posição;
- Incrementamos o tamanho.



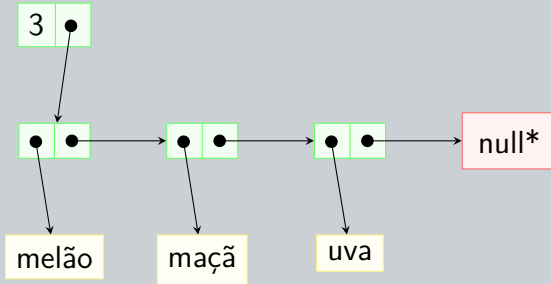
# Função *adicionaNaPosicao*

```
adicionaNaPosicao(Lista* umaLista, T* umDado, int umaPosicao)
    Elemento *novo, *anterior; // auxiliares.
inicio
    SE (posicao > umaLista._quantidade + 1) ENTAO THROW(
        ERROPOSICAO);
    SENA0
        SE (posicao = 1) ENTAO RETORNE(adicionaNoInicio(umaLista,
            umDado));
    SENA0
        novo <- ALOQUE(Elemento);
        SE (novo = NULO) ENTÃO THROW(ERROLISTACHEIA);
        SENA0
            anterior <- umaLista_primeiro;
            REPITA (posicao - 2) VEZES
                anterior <- anterior._proximo;
            novo._proximo <- anterior._proximo;
            novo._dado <- umDado;
            anterior._proximo <- novo;
            umaLista._quantidade <- umaLista._quantidade + 1;
        FIM SE
    FIM SE
FIM SE

fim;
```

# Função *retiraDaPosicao*

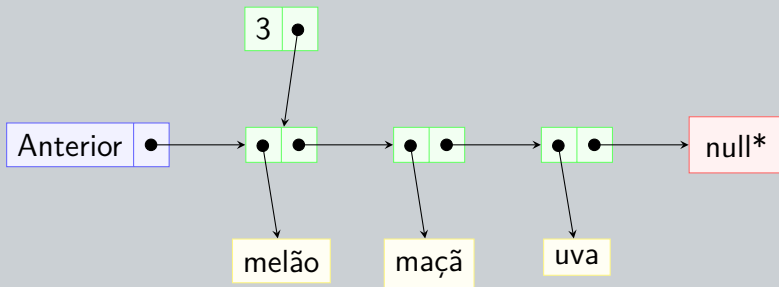
- Testamos se a posição existe;
- Caminhamos até a posição;
- Retiramos o dado da posição;
- Decrementamos o tamanho.





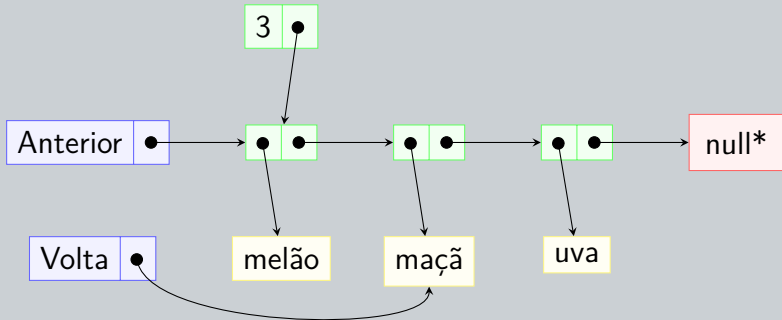
## Função *retiraDaPosicao*

- Testamos se a posição existe;
- Caminhamos até a posição;
- Retiramos o dado da posição;
- Decrementamos o tamanho.



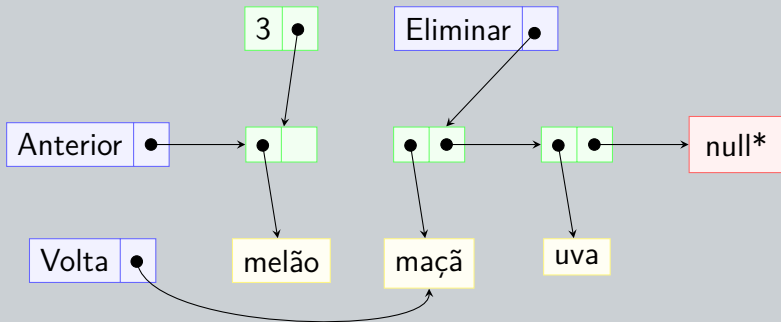
# Função *retiraDaPosicao*

- Testamos se a posição existe;
- Caminhamos até a posição;
- Retiramos o dado da posição;
- Decrementamos o tamanho.



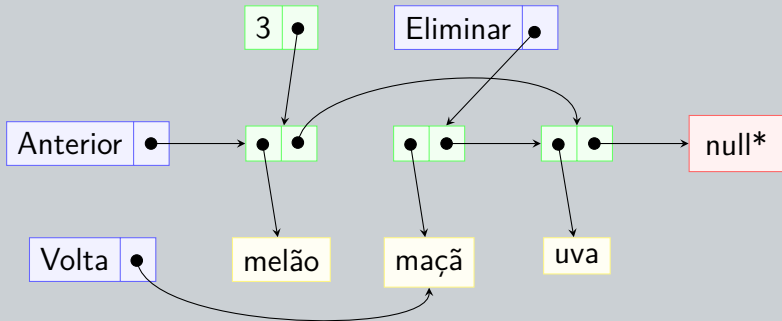
## Função *retiraDaPosicao*

- Testamos se a posição existe;
- Caminhamos até a posição;
- Retiramos o dado da posição;
- Decrementamos o tamanho.



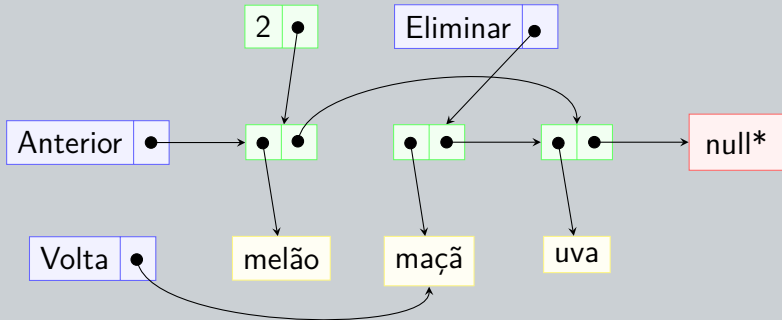
# Função *retiraDaPosicao*

- Testamos se a posição existe;
- Caminhamos até a posição;
- Retiramos o dado da posição;
- Decrementamos o tamanho.



# Função *retiraDaPosicao*

- Testamos se a posição existe;
- Caminhamos até a posição;
- Retiramos o dado da posição;
- Decrementamos o tamanho.



# Função *retiraDaPosicao(posicao)*

```
T* retiraDaPosicao(Lista *umaLista, int umaPosicao)
  Elemento *anterior, *eliminar; //Variáveis elemento.
T* volta; //Variável tipo T.
inicio
  SE (posicao > umaLista._quantidade) ENTAO THROW(ERROPOSICAO);
  SENAO
    SE (posicao = 1) ENTAO RETORNE(retiraDoInicio(umaLista));
    SENAO
      anterior <- umaLista._primeiro;
      REPITA (umaPosicao - 2) VEZES
        anterior <- anterior._proximo;
      eliminar <- anterior._proximo;
      volta <- eliminar._dado;
      anterior._proximo <- eliminar._proximo;
      umaLista._quantidade <- umaLista._quantidade - 1;
      DESALOQUE(eliminar);
      RETORNE(volta);
  FIM SE
FIM SE
fim;
```

## Função *destroiLista()*

- A lista precisa estar vazia
- Dúvida: por que não escrever uma única função em C que desaloque todos os elementos, dados e a lista?

# Algoritmos Restantes - Por conta do aluno

- `destroiLista(Lista* lista);`
- `posicao(Lista* lista, T* dado);`
- `contem(Lista* lista, T* dado);`
- `adicionaNoFim(Lista* lista, T* dado)`
- `retiraDoFim(Lista* lista, T* dado)`
- `retiraEspecifico(Lista * lista, T* dado);`



# Perguntas????



UNIVERSIDADE FEDERAL  
DE SANTA CATARINA



Este trabalho está licenciado sob uma Licença Creative Commons Atribuição 4.0 Internacional. Para ver uma cópia desta licença, visite

<http://creativecommons.org/licenses/by/4.0/>.



UNIVERSIDADE FEDERAL  
DE SANTA CATARINA