

Estruturas de Dados

Lista Encadeada

Departamento de Computação
Prof. Martín Vigil

Adaptado de https://github.com/jeanmartina/data_structures

2020.1



UNIVERSIDADE FEDERAL
DE SANTA CATARINA

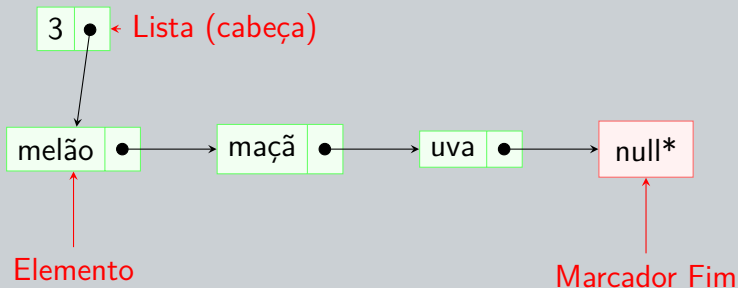
Definição de Lista

- 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.

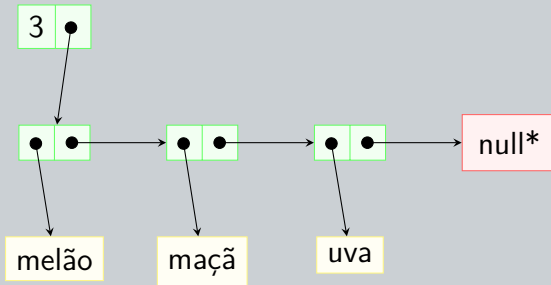
```
estrutura Lista {  
    Elemento* _primeiro;  
    inteiro _quantidade;  
};
```

Modelagem da 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:
 - `iniciaLista();`
 - `limpaLista(umaLista);`
 - `destroiLista(umaLista);`
- Testar se a lista está vazia ou cheia e outros testes:
 - `bool listaVazia(umaLista);`
 - `int posicao(umaLista, umDado);`
 - `bool contem(umaLista, umDado);`

Modelagem da Lista Encadeada

- Colocar e retirar dados da lista:
 - adiciona(umaLista, umDado);
 - adicionaNoInicio(umaLista, umDado);
 - adicionaNaPosicao(umaLista, umDado, umaPosicao);
 - T retira(umaLista);
 - T retiraDoInicio(umaLista);
 - T retiraDaPosicao(umaLista, umaPosicao);
 - T retiraEspecifico(umaLista, umDado);

Função *iniciaLista*

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

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

Função *destroiLista()*

- Percorrer a lista desalocando na seguinte ordem:
 - 1 dado
 - 2 elemento
- Manter referência para próximo elemento
- Repetir enquanto não chegar ao fim (elemento nulo)

Função *destroiLista()*

```
destroiLista(umaLista)
  Elemento *atual, *anterior; //Variável auxiliar para caminhar.
  inicio
  SE (listaVazia(umaLista)) ENTAO THROW(ERROLISTAVAZIA);
  SENA0
    atual <- umaLista._primeiro;
    ENQUANTO (atual != NULO) FACA
      //Eliminar até o fim.
      anterior <- atual;
      //Vou para o próximo mesmo que seja nulo.
      atual <- atual._proximo;
      //Liberar primeiro a Info.
      DESALOQUE(anterior._dado);
      //Liberar o elemento que acabei de visitar.
      DESALOQUE(anterior);
    FIM ENQUANTO
  FIM SE
fim;
```

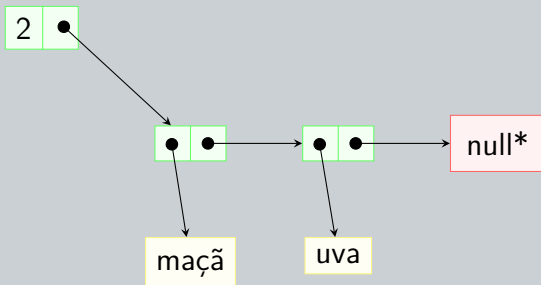
Função *listaVazia*

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

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

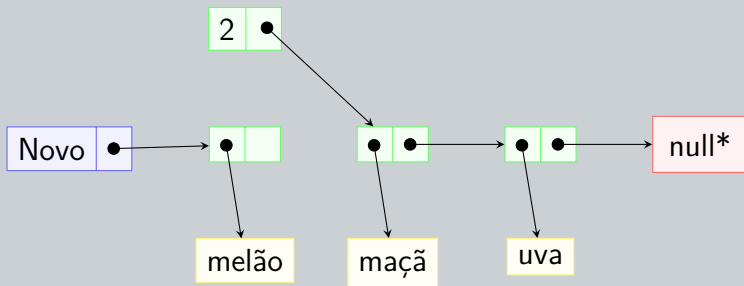
Função *adicionaNoInicio*

- 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.



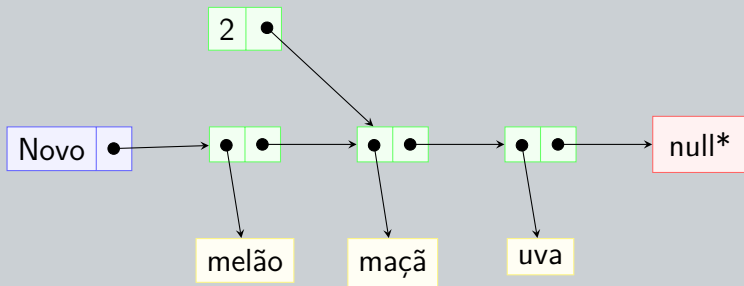
Função *adicionaNoInicio*

- 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.



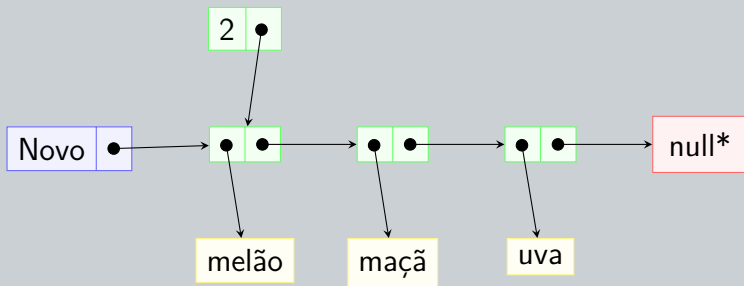
Função *adicionaNoInicio*

- 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.



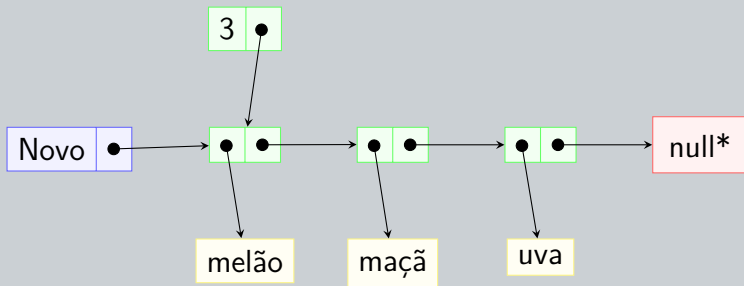
Função *adicionaNoInicio*

- 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.



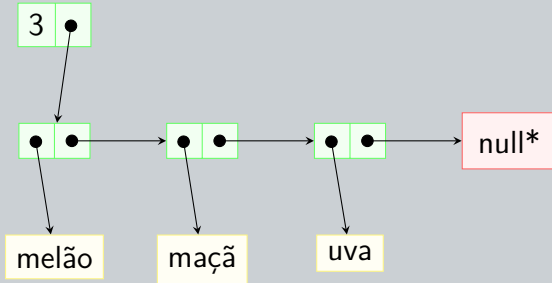
Função *adicionaNoInicio*

```
adicionaNoInicio(umaLista, umDado)
inicio
  novo <- ALOQUE(Elemento); //Variável auxiliar.
  SE (novo = NULO) ENTAO
    THROW(ERROSEMEMORIA);
  SENAO
    novo._proximo <- umaLista._primeiro;
    novo._dado <- umDado;
    umaLista._primeiro <- novo;
    umaLista._quantidade <- umaLista._quantidade +
      1;
  FIM SE
fim;
```

- Complexidade de tempo: ?

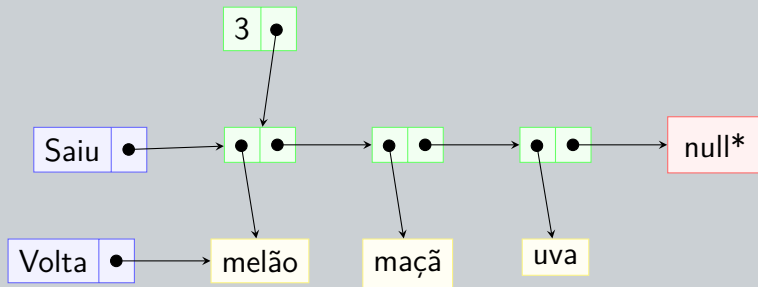
Função *retiraDoInicio*

- Teste se há elementos;
- Decremente o tamanho;
- Desaloque a memória do elemento;
- Devolva o dado retirado.



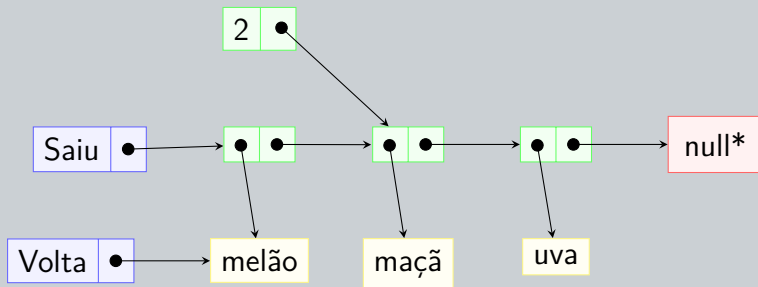
Função *retiraDoInicio*

- Teste se há elementos;
- Decremente o tamanho;
- Desaloque a memória do elemento;
- Devolva o dado retirado.



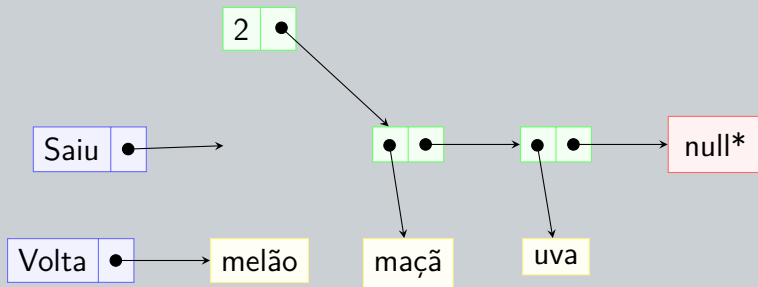
Função *retiraDoInicio*

- Teste se há elementos;
- Decremente o tamanho;
- Desaloque a memória do elemento;
- Devolva o dado retirado.



Função *retiraDoInicio*

- Teste se há elementos;
- Decremente o tamanho;
- Desaloque a memória do elemento;
- Devolva o dado retirado.



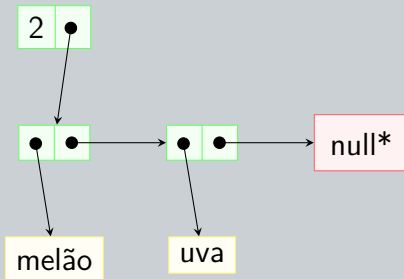
Função *T retiraDoInicio*

```
T* retiraDoInicio(umaLista)
  Elemento* saiu; //Variável auxiliar elemento.
  T* volta; //Variável auxiliar tipo T.
  início
    SE (listaVazia(umaLista)) ENTAO
      THROW(ERROLISTAVAZIA);
    SENA0
      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: ?

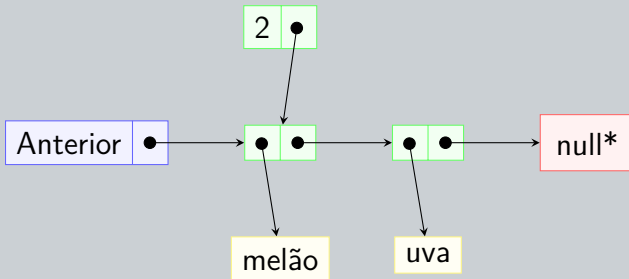
Função *adicionaNaPosicao*

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



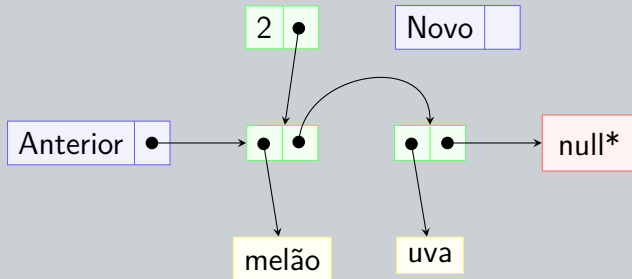
Função *adicionaNaPosicao*

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



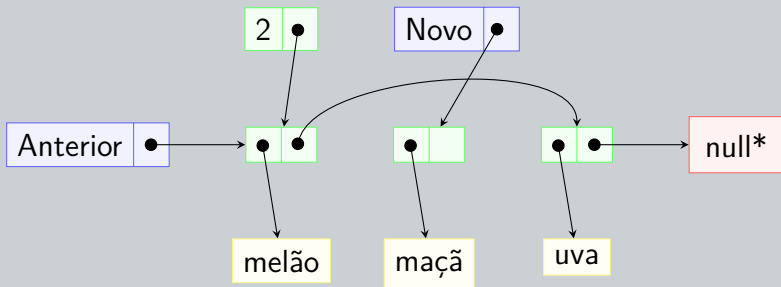
Função *adicionaNaPosicao*

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



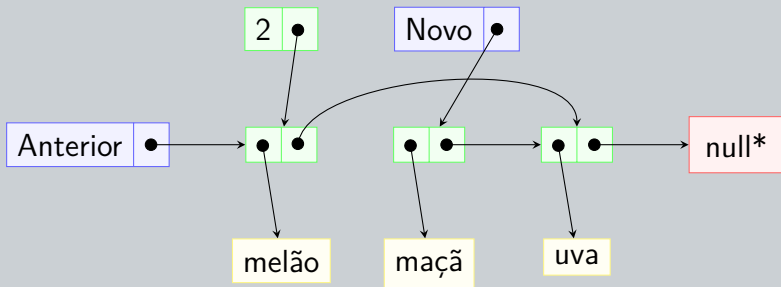
Função *adicionaNaPosicao*

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



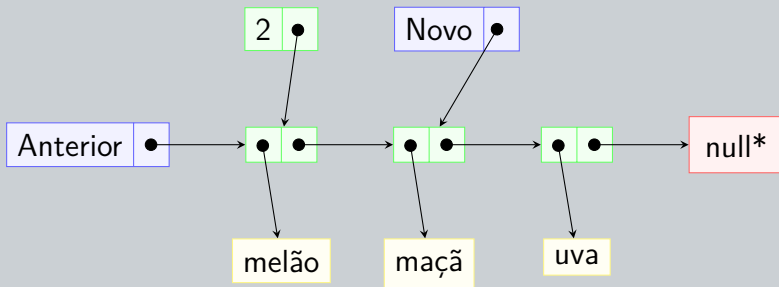
Função *adicionaNaPosicao*

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



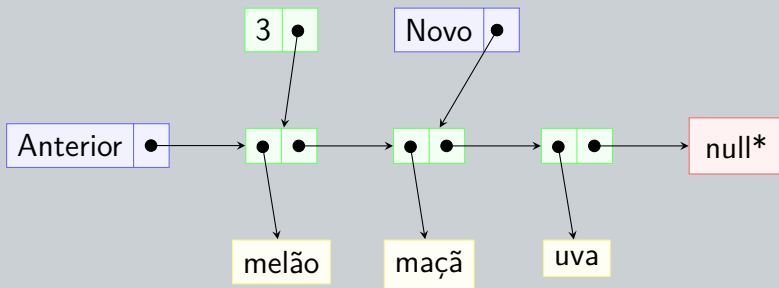
Função *adicionaNaPosicao*

- 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.



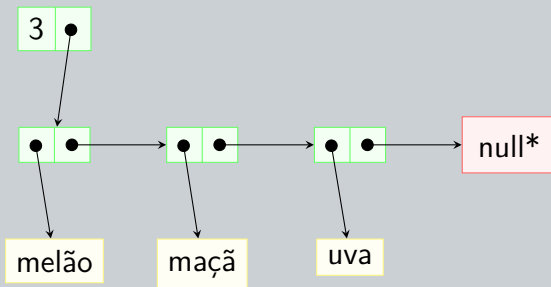
Função *adicionaNaPosicao*

```
adicionaNaPosicao(umaLista, umDado, umaPosicao)
  Elemento *novo, *anterior; // auxiliares.
  inicio
    SE (posicao > umaLista._quantidade + 1) ENTAO THROW(
      ERROPOSIÇÃO);
  SENAO
    SE (posicao = 1) ENTAO RETORNE(adicionaNoInício(umaLista,
      umDado));
  SENAO
    novo <- ALOQUE(Elemento);
    SE (novo = NULO) ENTÃO THROW(ERROLISTACHEIA);
    SENAO
      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;
```

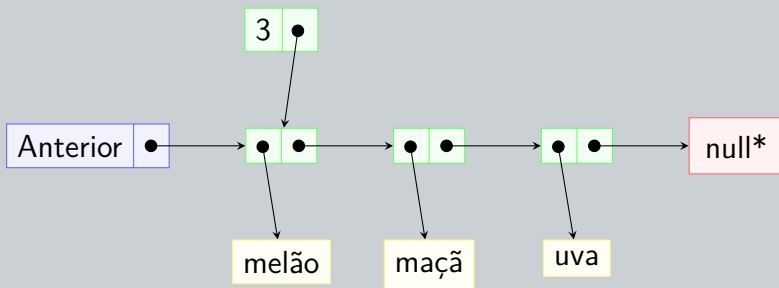
Função *retiraDaPosicao*

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



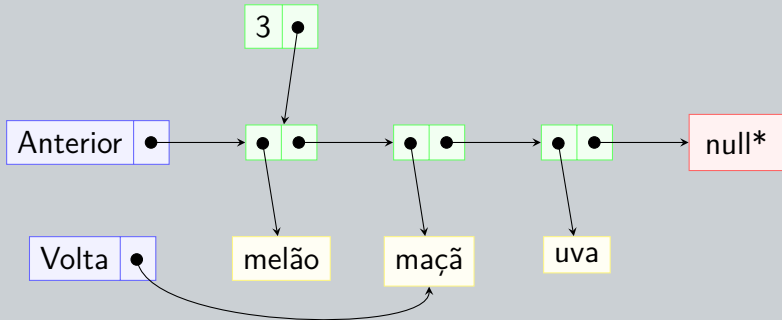
Função *retiraDaPosicao*

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



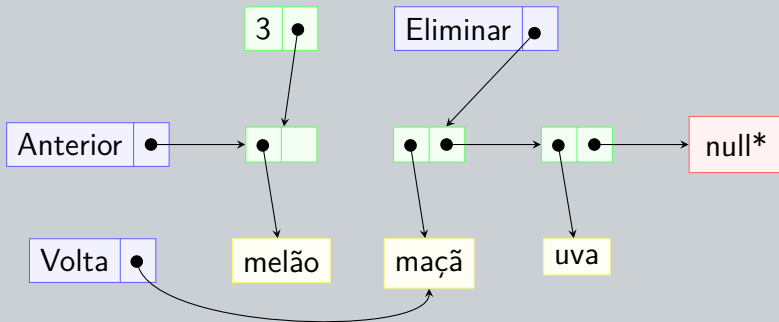
Função *retiraDaPosicao*

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



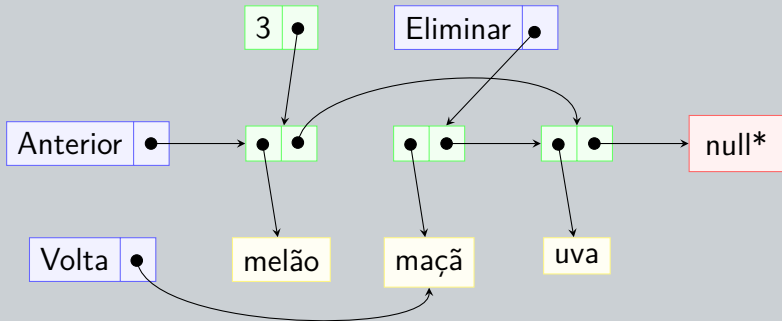
Função *retiraDaPosicao*

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



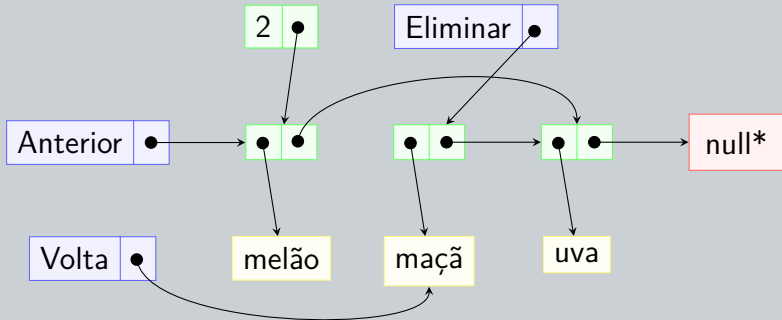
Função *retiraDaPosicao*

- 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(umaLista, 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;
```


Algoritmos Restantes - Por conta do aluno

- Adiciona(dado):
 - AdicionaNaPosicao(tamanho);
- Retira():
 - T RetiraDaPosicao(tamanho);
- T RetiraEspecifico(dado).
- int posicao(dado);
- Elemento* posicao(dado);
- bool contem(dado);

Função *destroiLista()*

```
destroiLista(umaLista)
  Elemento *atual, *anterior; //Variável auxiliar para caminhar.
  inicio
  SE (listaVazia(umaLista)) ENTAO THROW(ERROLISTAVAZIA);
  SENAQ
    atual <- umaLista._primeiro;
    ENQUANTO (atual != NULO) FACA
      //Eliminar até o fim.
      anterior <- atual;
      //Vou para o próximo mesmo que seja nulo.
      atual <- atual._proximo;
      //Liberar primeiro a Info.
      DESALOQUE(anterior._dado);
      //Liberar o elemento que acabei de visitar.
      DESALOQUE(anterior);
    FIM ENQUANTO
  FIM SE
fim;
```

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