Estruturas de Dados

Lista Encadeada

Departamento de Computação
Prof. Martín Vigil
Adaptado de `https://github.com/jeanmartina/data_structures`

2016.2

UNIVERSIDADE FEDERAL
DE SANTA CATARINA

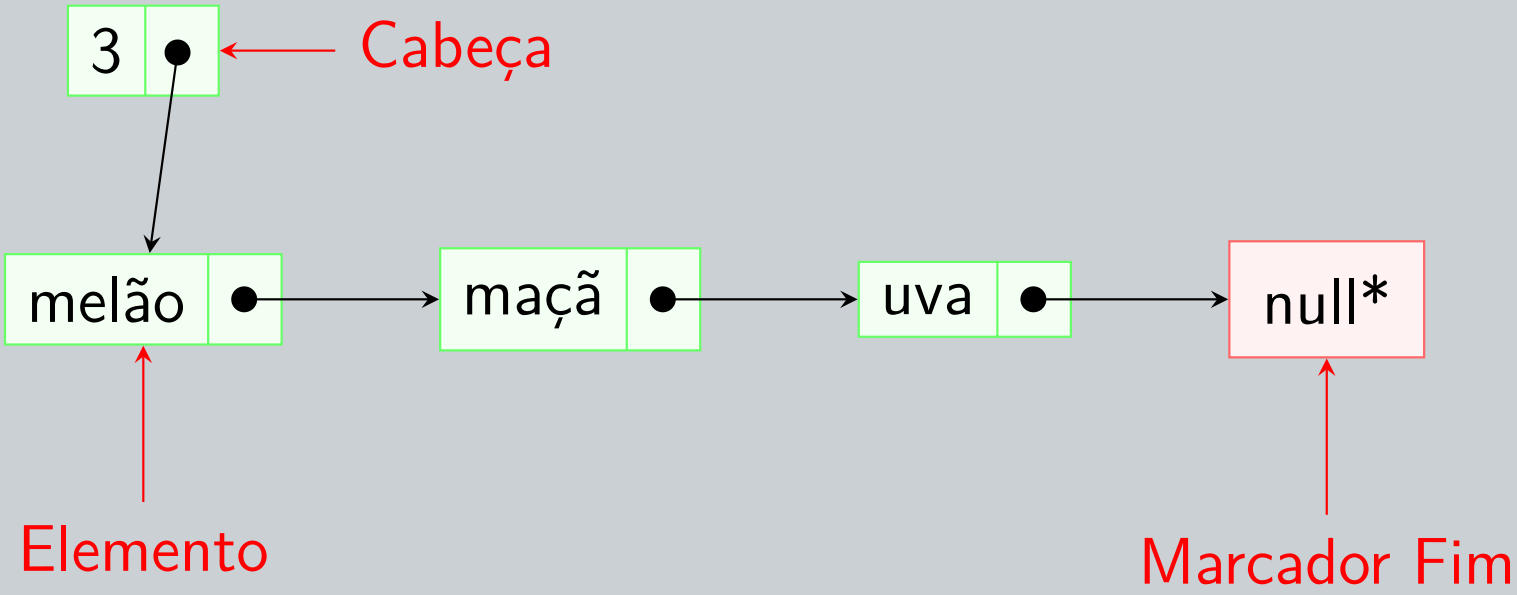
Definição de Lista

- 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
- Para referenciar o primeiro elemento utilizamos um objeto do tipo ***cabeça de lista***.

Lista Encadeada



Modelagem da Cabeça de Lista

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

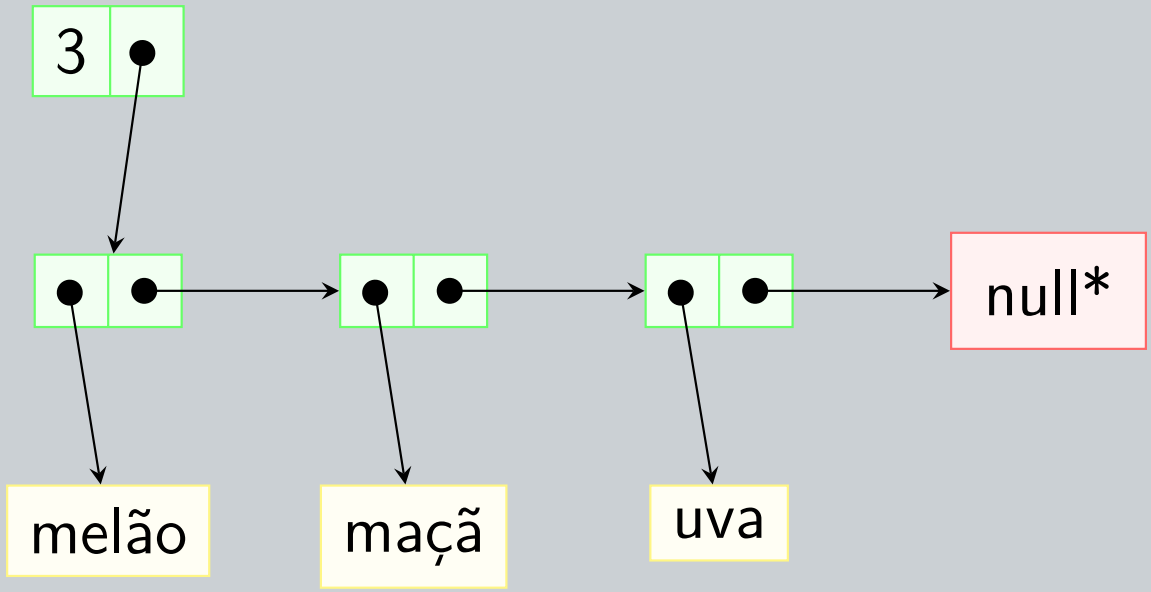
```
estrutura Lista {
 Elemento *_dados;
 inteiro _tamanho;
};
```

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.

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

Modelagem de Lista Encadeada



Modelagem da Elemento de Lista

- Aspecto Estrutural:
  - Necessitamos um ponteiro para o próximo elemento da lista;
  - Necessitamos um ponteiro do tipo da informação que vamos armazenar.
  - T necessita de um destrutor próprio, assim como a lista (neste caso a cabeça) vai precisar de um também;

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

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();
  - 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);
  - adicionaEmOrdem(umaLista, umDado);
  - 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._dados <- null;
  umaLista._tamanho <- 0;
  retorne umaLista;
fim;
```

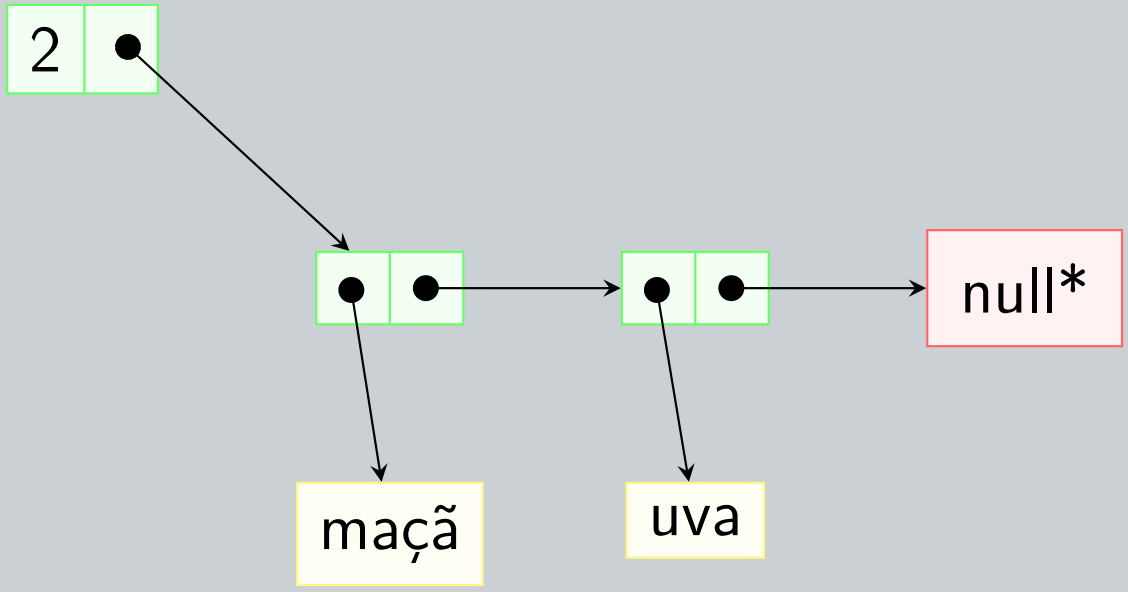
Função *listaVazia(umaLista)*

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

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

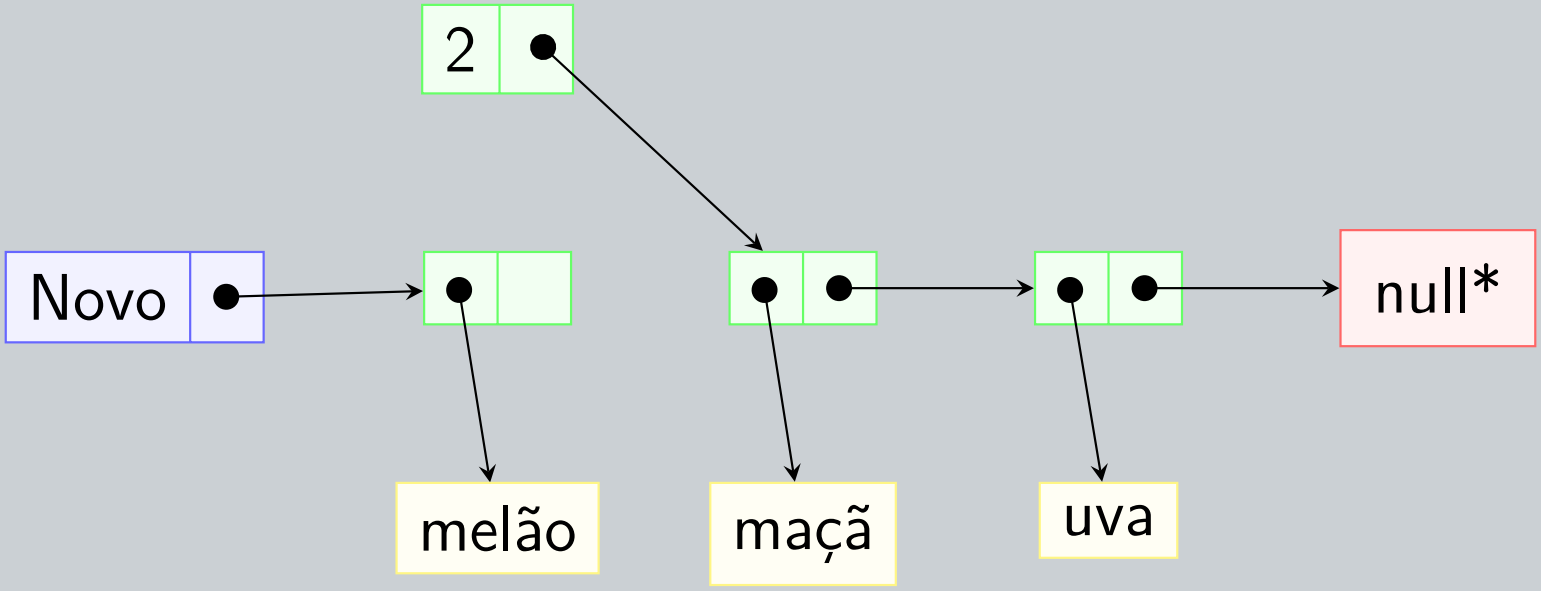
Função *adicionaNoInicio(umaLista, umDado)*

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.



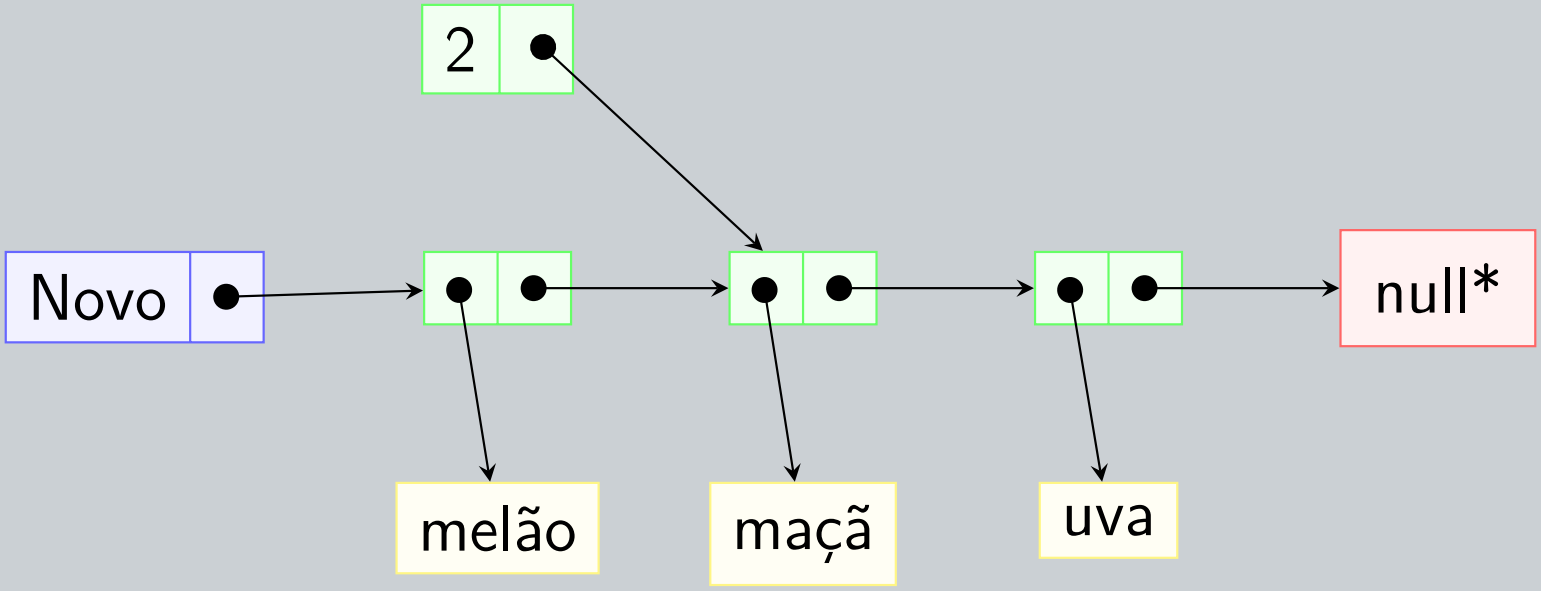
Função *adicionaNoInicio(umaLista, umDado)*

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.



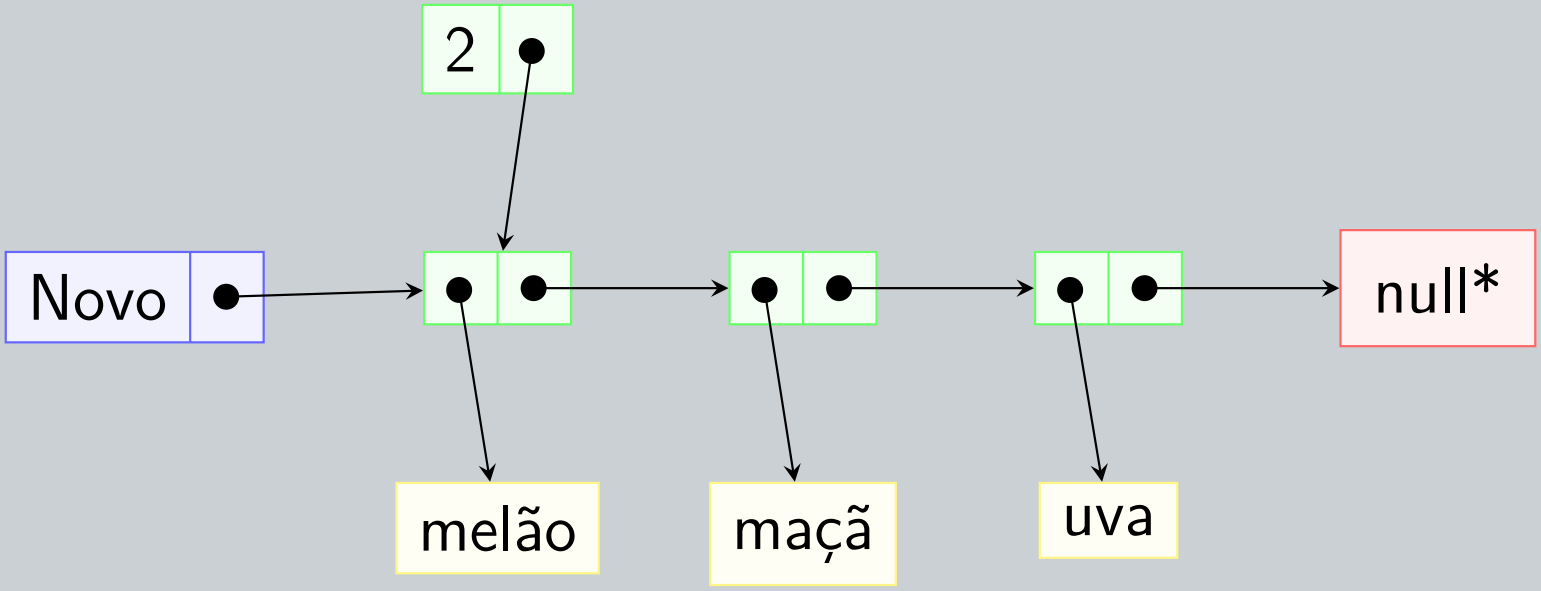
Função *adicionaNoInicio(umaLista, umDado)*

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.



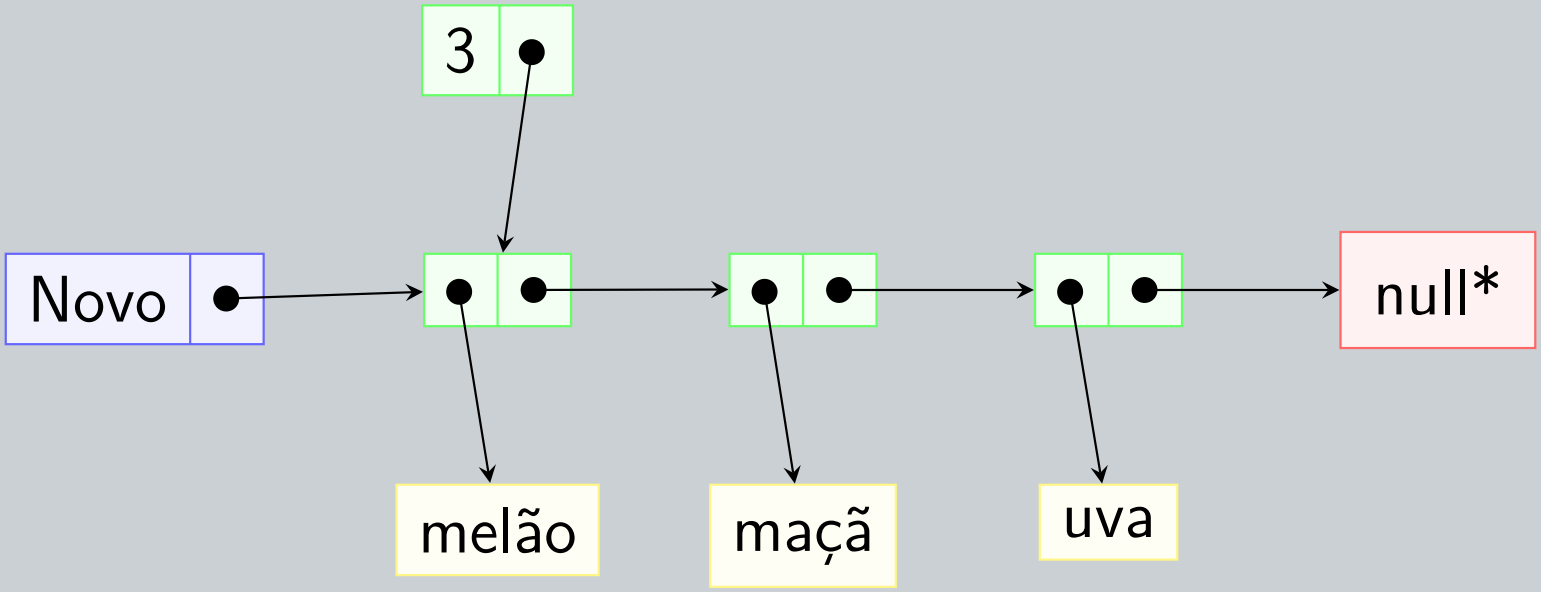
Função *adicionaNoInicio(umaLista, umDado)*

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(umaLista, umDado)*

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.



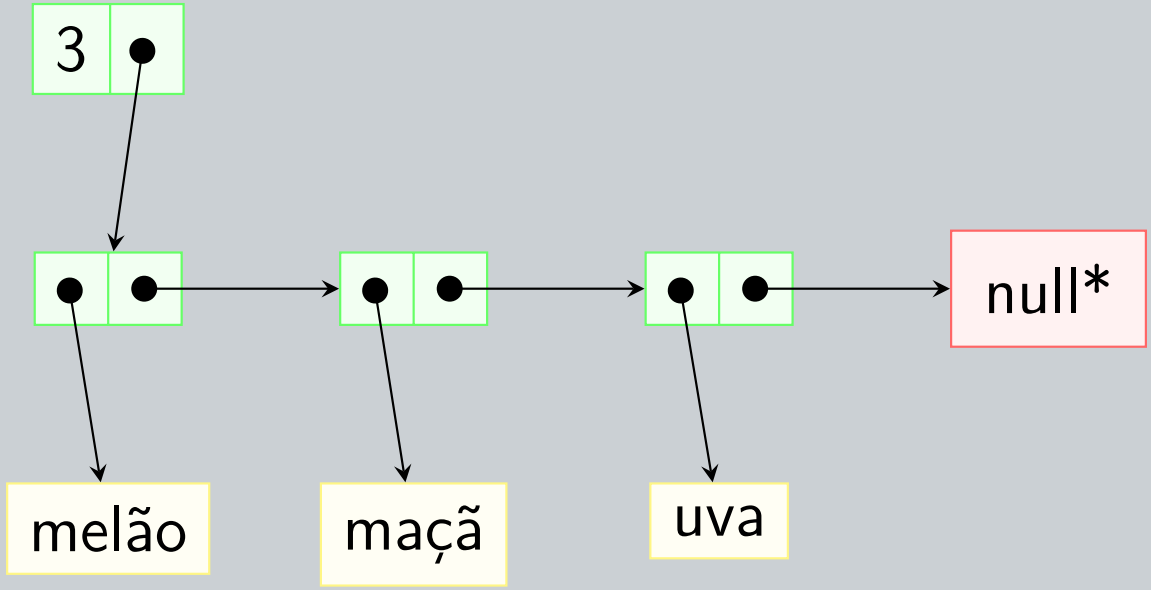
Função *adicionaNoInicio(umaLista, umDado)*

```
adicionaNoInicio(umaLista, umDado)
 inicio
  novo <- aloque(Elemento); //Variável auxiliar.
  SE (novo = NULO) ENTAO
   THROW(ERROSEMMEMORIA);
  SENAO
   novo._proximo <- umaLista._dados;
   novo._dado <- umDado;
   umaLista._dados <- novo;
   umaLista._tamanho <- umaLista._tamanho + 1;
  FIM SE
  fim;
```

- Complexidade de tempo: ?

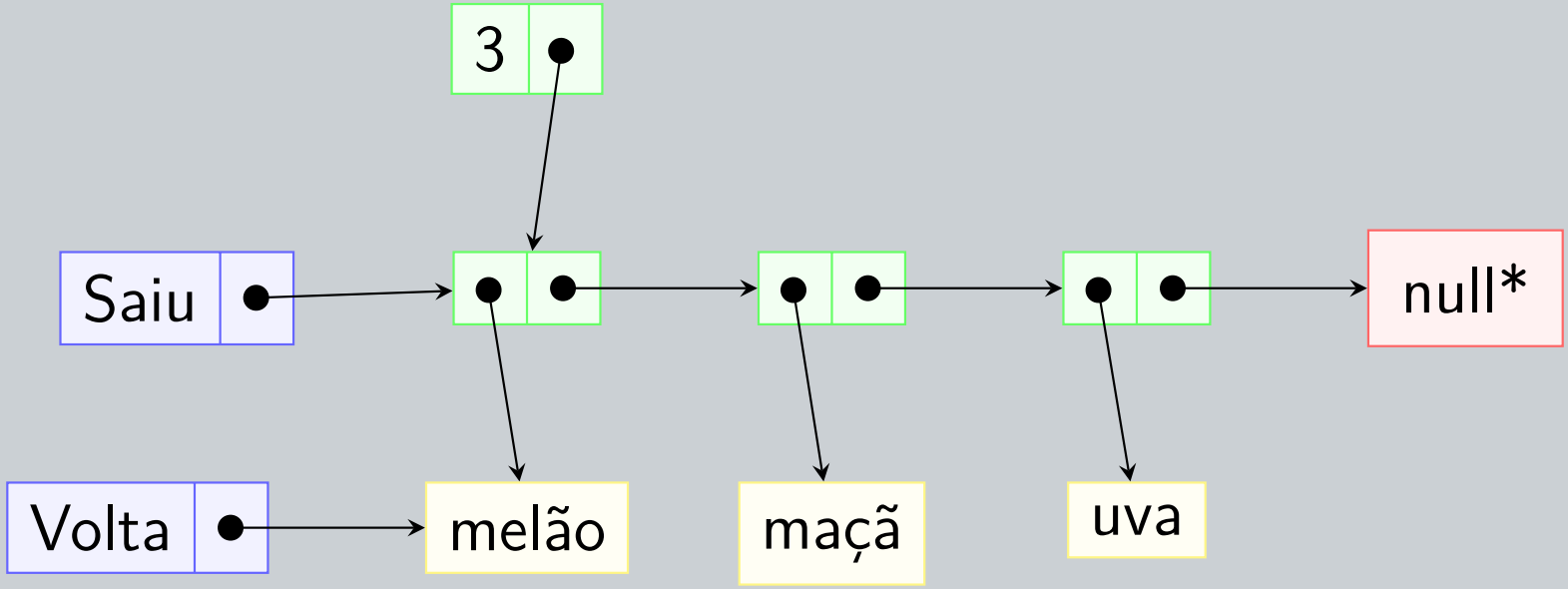
Função *T retiraDoInicio(umaLista)*

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



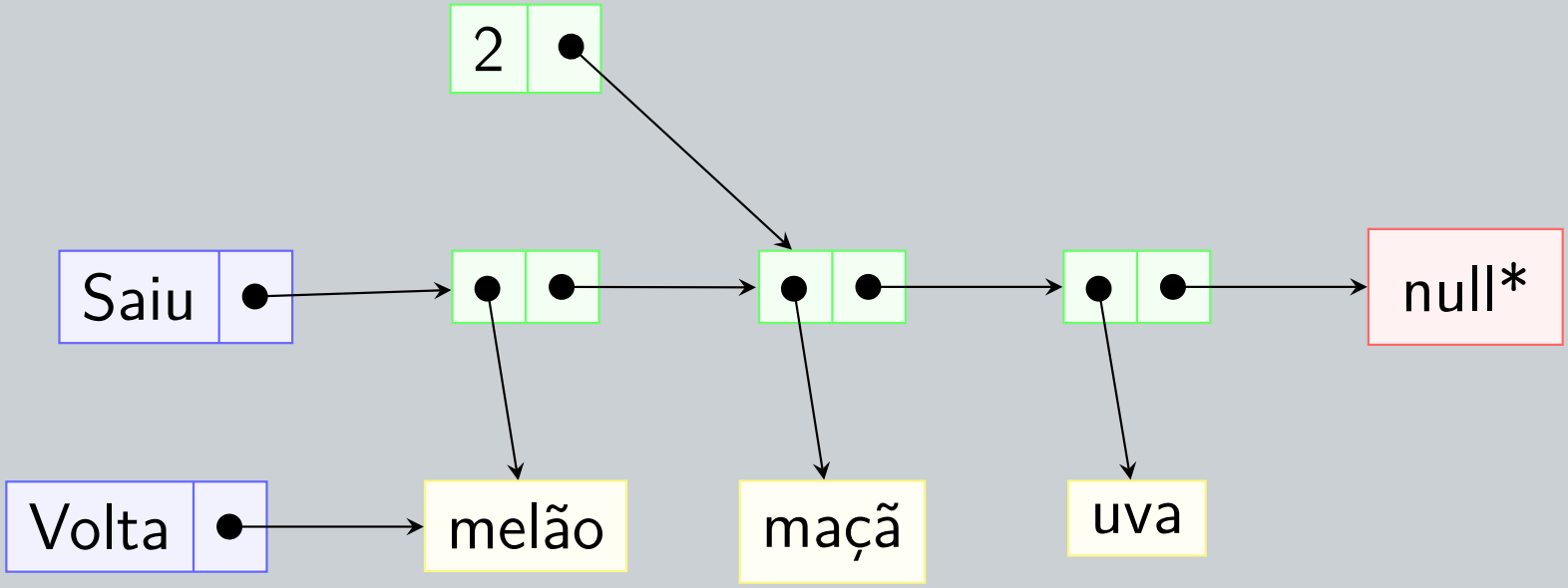
Função *T retiraDoInicio(umaLista)*

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



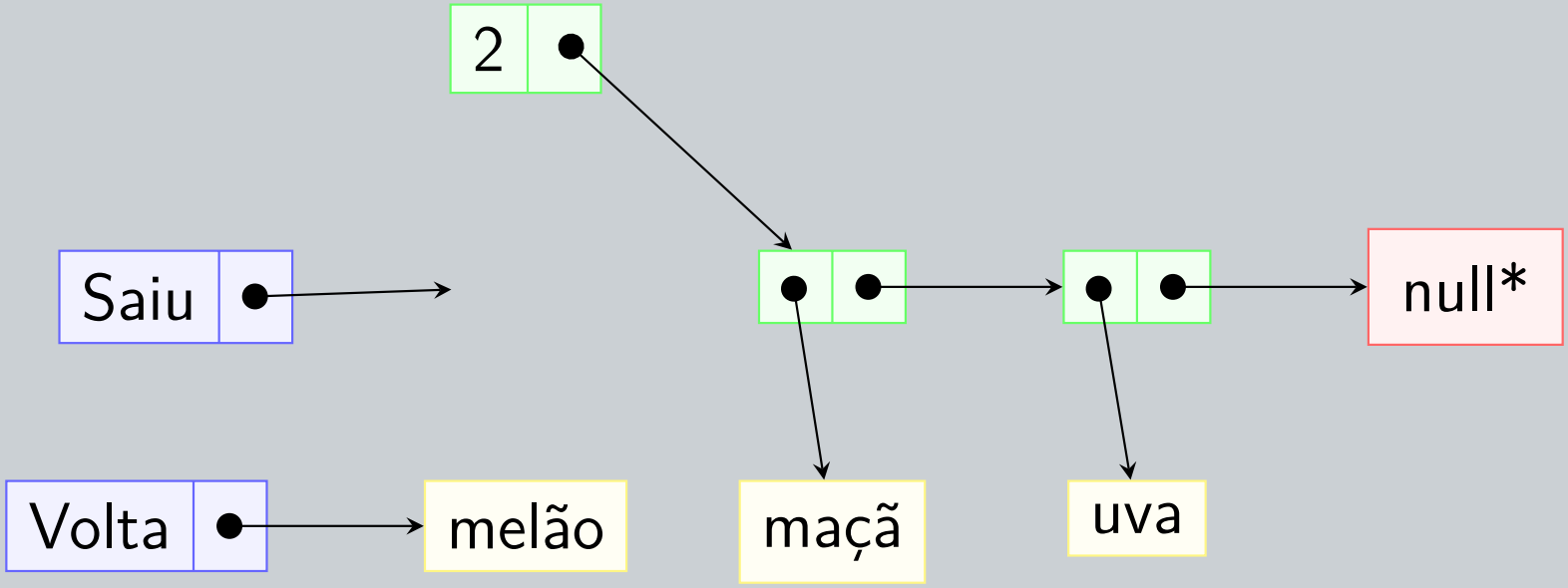
Função *T retiraDoInicio(umaLista)*

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



Função *T retiraDoInicio(umaLista)*

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



Função *T retiraDoInicio(umaLista)*

```
T retiraDoInicio()
 Elemento saiu; //Variável auxiliar elemento.
 T volta; //Variável auxiliar tipo T.
    início
        SE (listaVazia(umaLista)) ENTAO
            THROW(ERROLISTAVAZIA);
        SENAO
            saiu <- umaLista._dados;
            volta <- saiu->dado;
            umaLista._dados <- saiu->próximo;
            umaLista._tamanho <- umaLista._tamanho
                - 1;
            LIBERE(saiu);
            RETORNE(volta);
        FIM SE
    fim;
```

- Complexidade de tempo: ?

Função *T retiraDoInicio()*

```
T retiraDoInicio()
 Elemento *saiu; //Variável auxiliar elemento.
 T *volta; //Variável auxiliar tipo T.
^^Iinício
^^I^^I SE (listaVazia()) ENTAO
^^I^^I   THROW(ERROLISTAVAZIA);
^^I^^I SENAO
^^I^^I  saiu <- _dados;
^^I^^I  volta <- saiu.info;
^^I^^I  _dados <- saiu.próximo;
^^I^^I  _tamanho <- _tamanho - 1;
^^I^^I  LIBERE(saiu);
^^I^^I  RETORNE(volta);
^^I^^IFIM SE
^^Ifim;
```

Função *T retiraDoInicio()*

```
T retiraDoInicio()
 Elemento *saiu; //Variável auxiliar elemento.
 T *volta; //Variável auxiliar tipo T.
^^Iinício
^^I^^ISE (listaVazia()) ENTAO
^^I^^I  THROW(ERROLISTAVAZIA);
^^I^^ISENAO
^^I^^I   saiu <- _dados;
^^I^^I   volta <- saiu.info;
^^I^^I   _dados <- saiu.próximo;
^^I^^I   _tamanho <- _tamanho - 1;
^^I^^I  LIBERE(saiu);
^^I^^I  RETORNE(volta);
^^I^^IFIM SE
^^Ifim;
```

Função *T retiraDoInicio()*

```
T retiraDoInicio()
 Elemento *saiu; //Variável auxiliar elemento.
 T *volta; //Variável auxiliar tipo T.
^^Iinício
^^I^^ISE (listaVazia()) ENTAO
^^I^^I  THROW(ERROLISTAVAZIA);
^^I^^ISENAO
^^I^^I  saiu <- _dados;
^^I^^I  volta <- saiu.info;
^^I^^I  _dados <- saiu.próximo;
^^I^^I  _tamanho <- _tamanho - 1;
^^I^^I   LIBERE(saiu);
^^I^^I   RETORNE(volta);
^^I^^IFIM SE
^^Ifim;
```

Função *eliminaDoInicio()*

```
 eliminaDoInicio()
 Elemento *saiu; //Variável auxiliar elemento.
^^Iinício
^^I^^ISE (listaVazia()) ENTAO
^^I^^I  THROW(ERROLISTAVAZIA);
^^I^^ISENAO
^^I^^I  saiu <- _dados;
^^I^^I  volta <- saiu.info;
^^I^^I  _dados <- saiu.próximo;
^^I^^I  _tamanho <- _tamanho - 1;
^^I^^I  LIBERE(saiu);
^^I^^I  LIBERE(saiu.info);
^^I^^IFIM SE
^^Ifim;
```

Algoritmo eliminaDoInicio()

- Observe que a linha LIBERE(saiu.info) possui um perigo:
  - Se o T for por sua vez um conjunto estruturado de dados com referências internas através de ponteiros (outra lista, por exemplo), a chamada à função LIBERE(saiu.info) só liberará o primeiro nível da estrutura (aquele apontado diretamente);
  - Tudo o que for referenciado através de ponteiros em info permanecerá em algum lugar da memória, provavelmente inatingível (garbage);
  - Para evitar isto pode-se criar uma função destrói(info) para o T que será chamada no lugar de LIBERE.

Importância do Destrutor

- O destrutor diz como o objeto será destruído quando sair de escopo;
- No mínimo deve liberar a memória que foi alocada por chamadas “new”no construtor;
- Se nenhum destrutor for declarado será gerado um default, que aplicará o destrutor correspondente a cada dado da classe;
- A recursão tem que ser garantida pelo objeto.

Função *adicionaNaPosicao(T dado, int posicao)*

- Testamos se a posição existe e se é possível alocar;

Função *adicionaNaPosicao(T dado, int posicao)*

- Testamos se a posição existe e se é possível alocar;
- Caminhamos até a posição;

Função *adicionaNaPosicao(T dado, int posicao)*

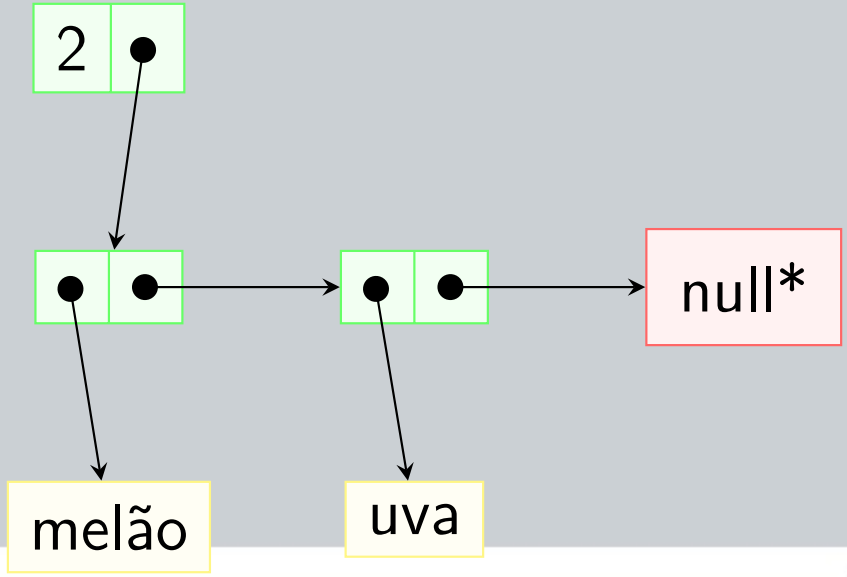
- Testamos se a posição existe e se é possível alocar;
- Caminhamos até a posição;
- Adicionamos o novo dado na posição;

Função *adicionaNaPosicao(T dado, int posicao)*

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

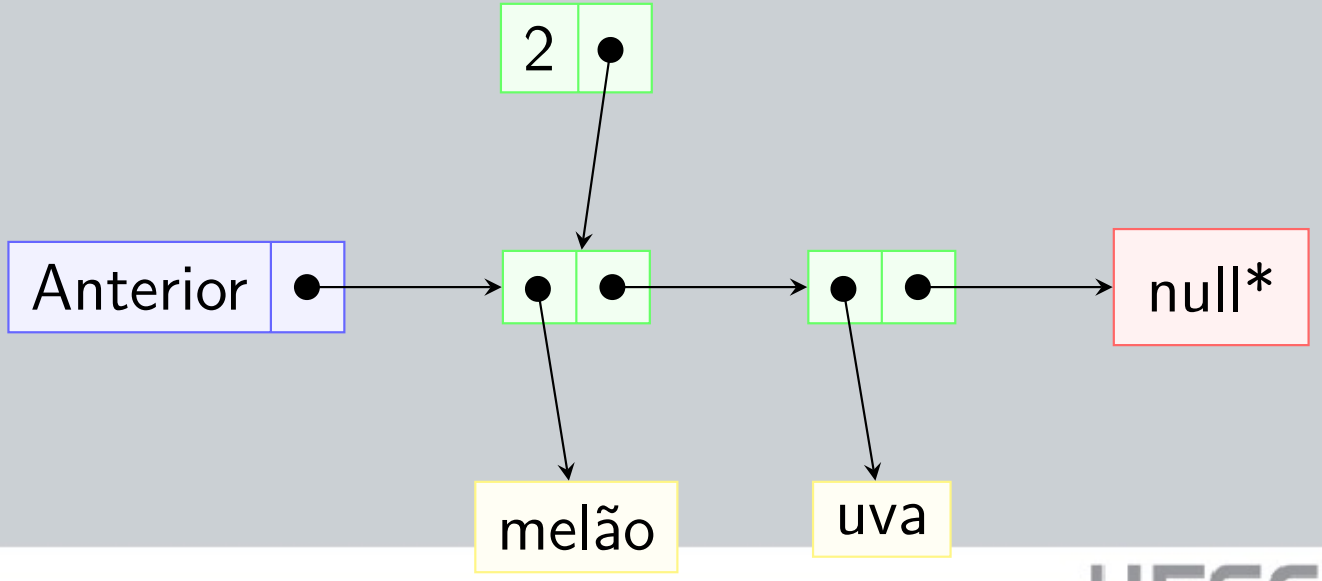
Função *adicionaNaPosicao(T dado, int posicao)*

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



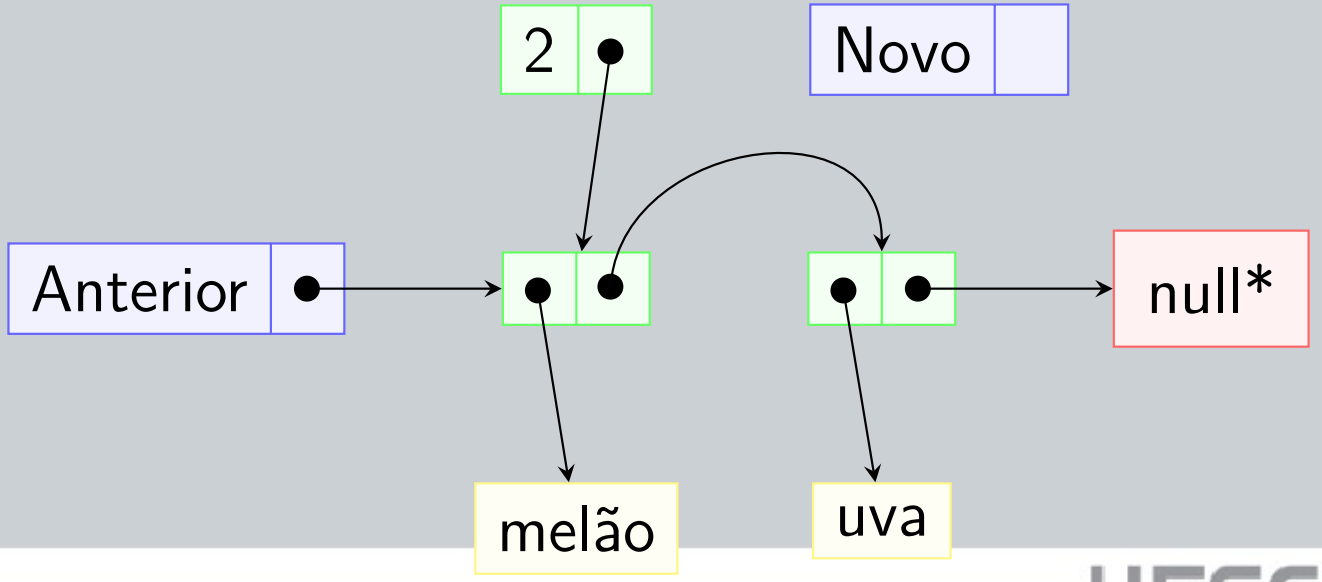
Função *adicionaNaPosicao(T dado, int posicao)*

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



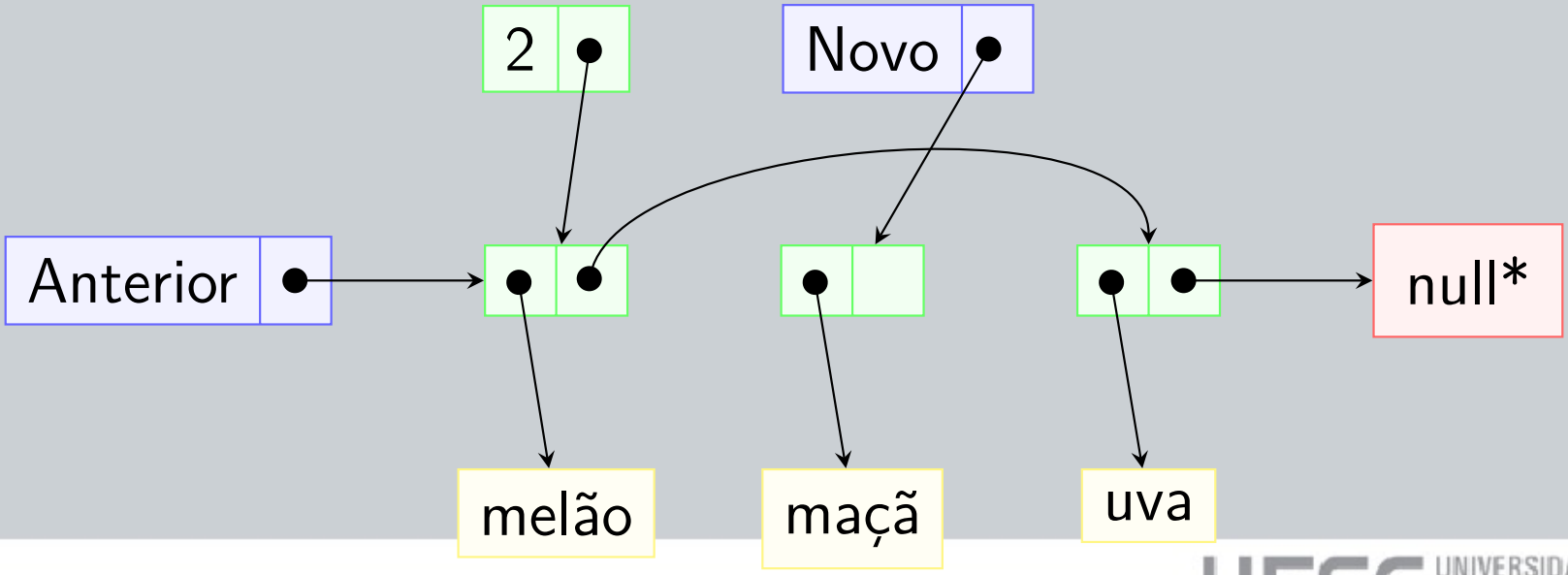
Função *adicionaNaPosicao(T dado, int posicao)*

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



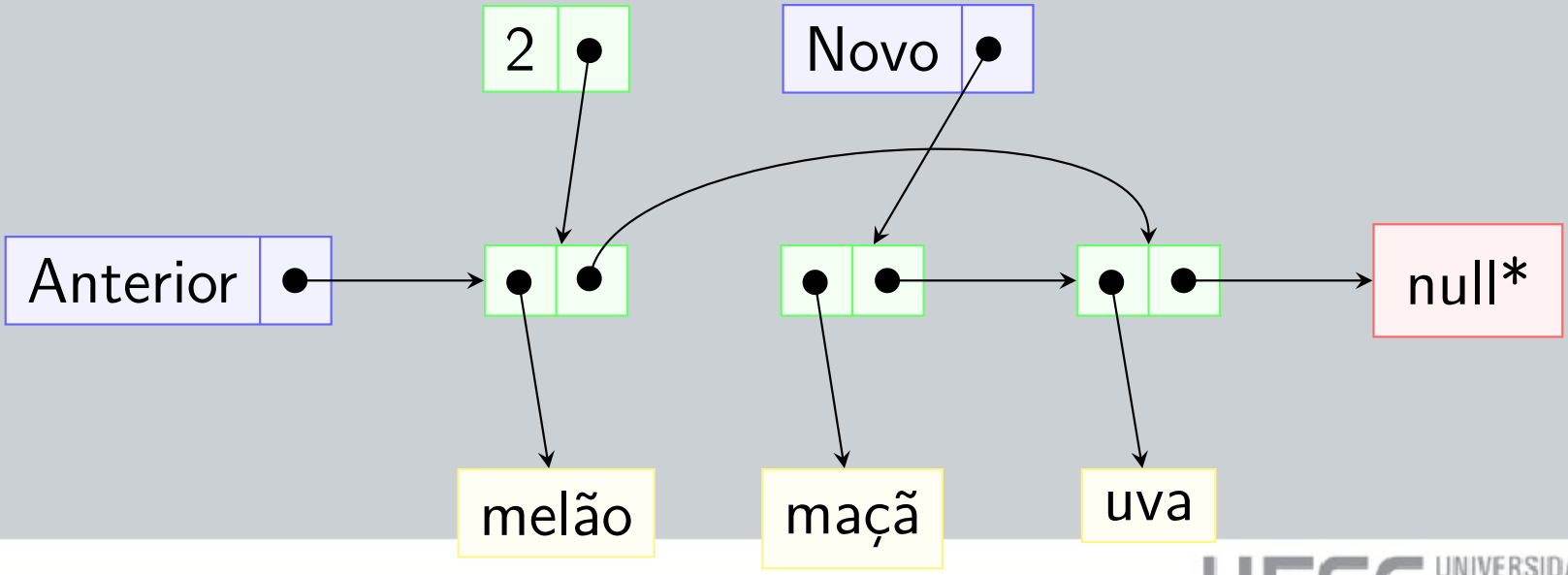
Função *adicionaNaPosicao(T dado, int posicao)*

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



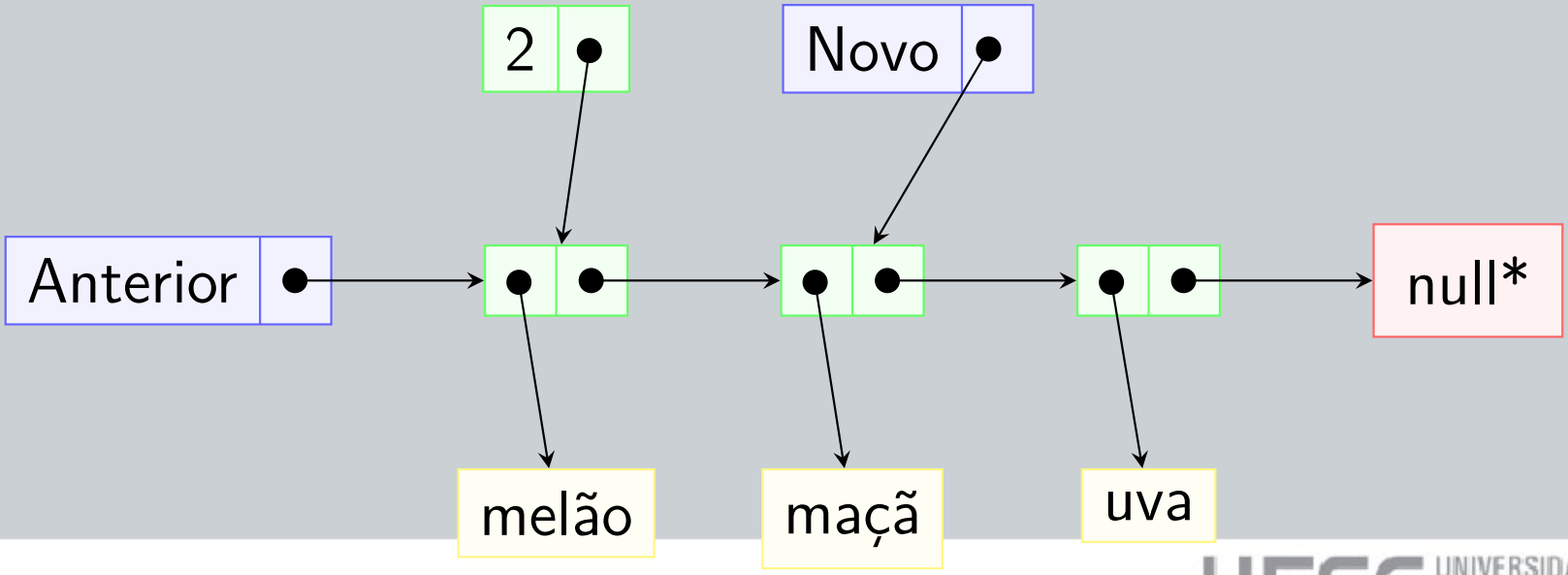
Função *adicionaNaPosicao(T dado, int posicao)*

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



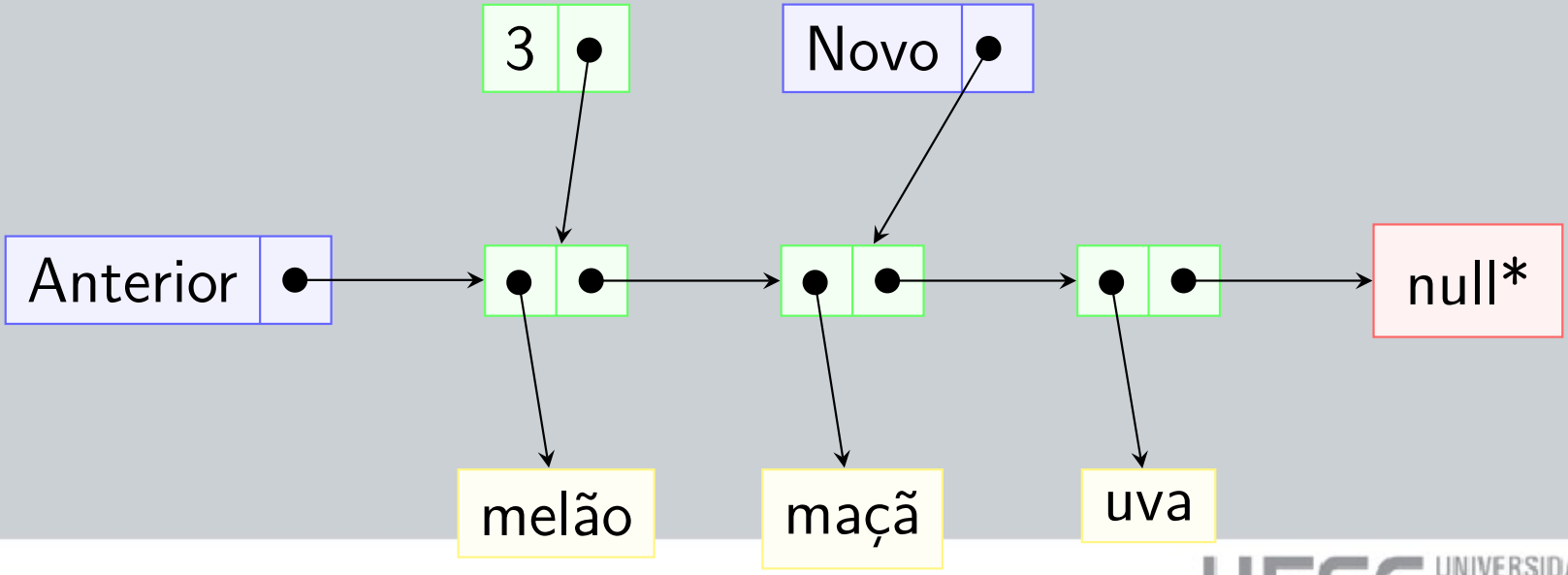
Função *adicionaNaPosicao(T dado, int posicao)*

- 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(T dado, int posicao)*

- 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(T dado, int posicao)*

```
adicionaNaPosicao(T dado, int posicao)
 Elemento *novo, *anterior; // auxiliares.
 inicio
  SE (posicao > _tamanho + 1) ENTAO THROW(ERROPOSICAO);
  SENAO
   SE (posicao = 1) ENTAO RETORNE(adicionaNoInício(info));
   SENAO
    novo <- aloque(Elemento);
    SE (novo = NULO) ENTÃO THROW(ERROLISTACHEIA);
    SENAO
     anterior <- _dados;
     REPITA (posicao - 2) VEZES
      anterior <- anterior._proximo;
     novo._proximo <- anterior._proximo;
     novo._dado <- info;
     anterior._proximo <- novo;
     _tamanho <- _tamanho + 1;
    FIM SE
   FIM SE
  FIM SE
fim;
```

Função *T retiraDaPosicao(int posicao)*

- Testamos se a posição existe;

Função *T retiraDaPosicao(int posicao)*

- Testamos se a posição existe;
- Caminhamos até a posição;

Função *T retiraDaPosicao(int posicao)*

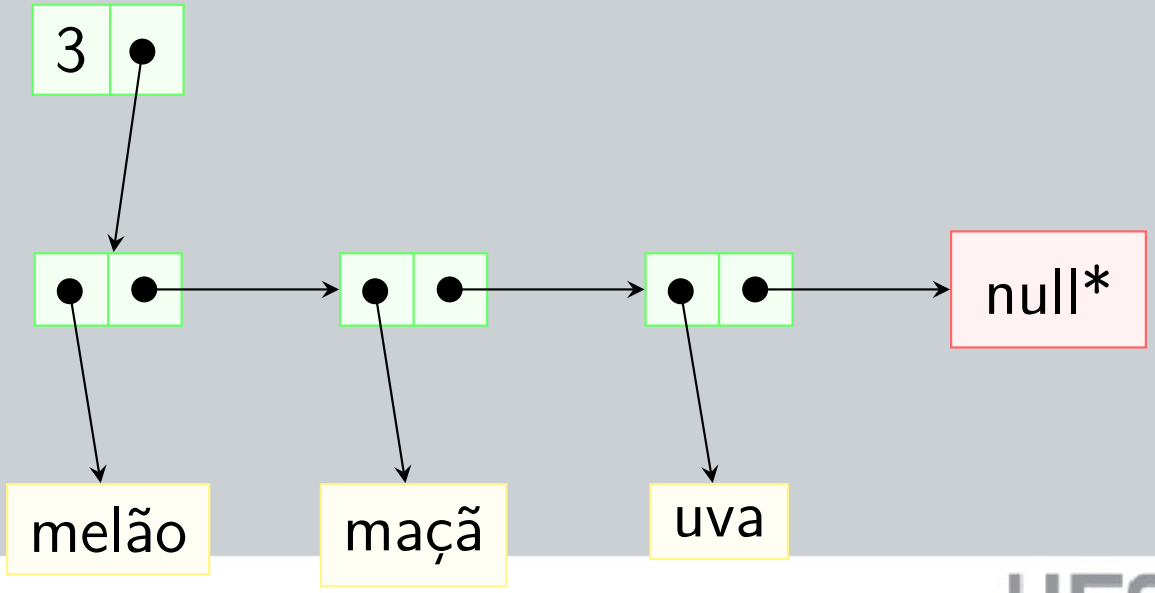
- Testamos se a posição existe;
- Caminhamos até a posição;
- Retiramos o dado da posição;

Função *T retiraDaPosicao(int posicao)*

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

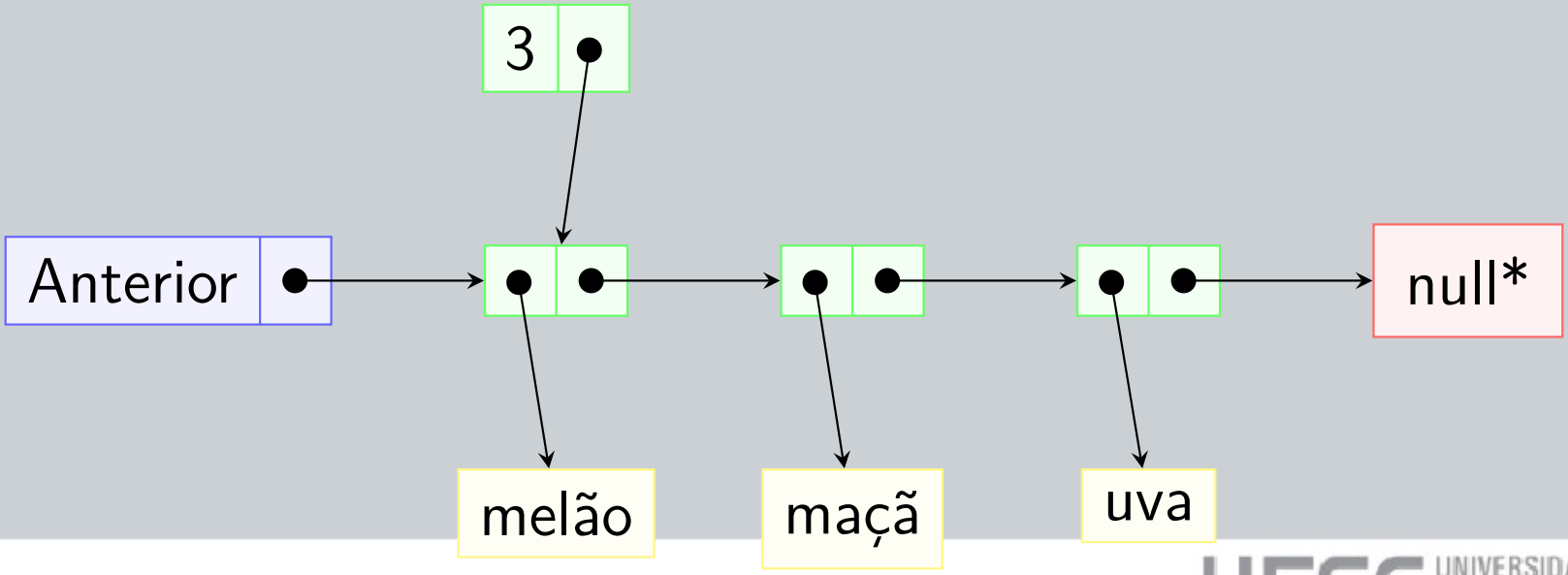
Função *T retiraDaPosicao(int posicao)*

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



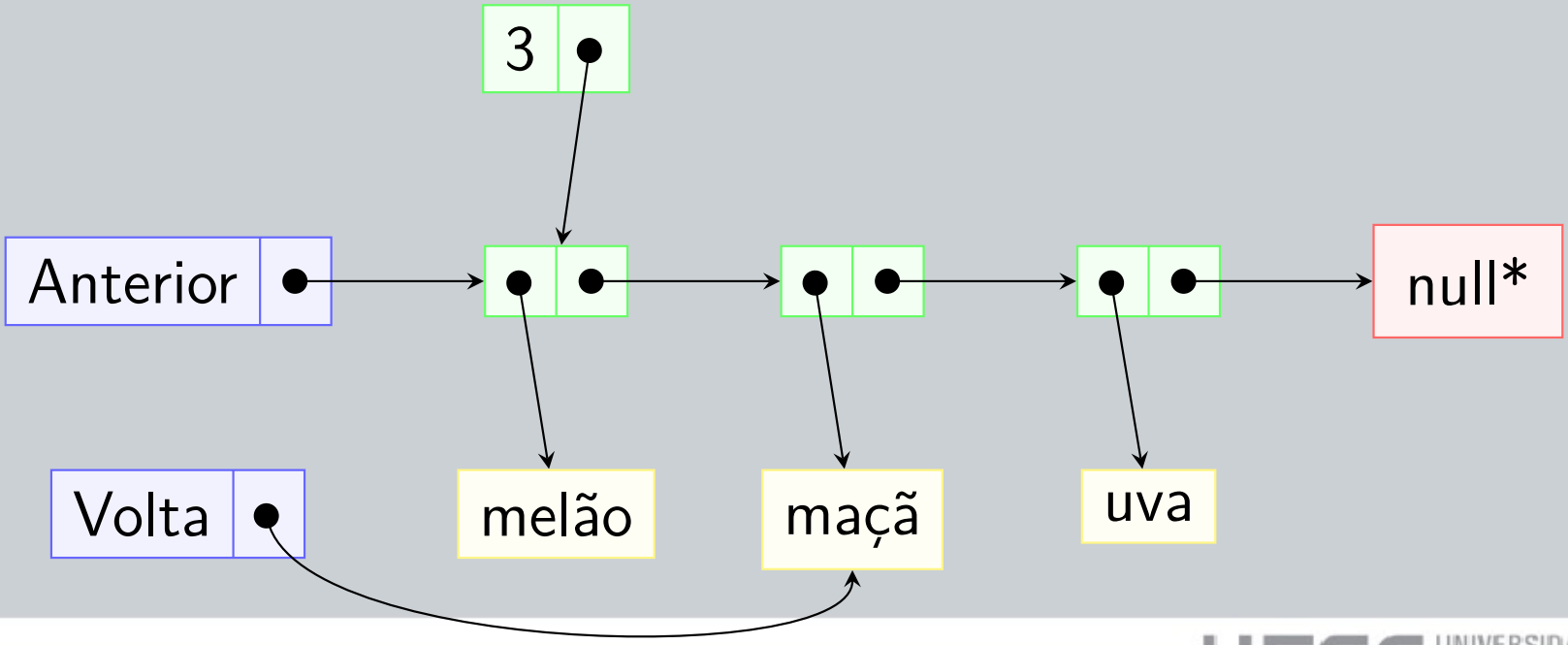
Função *T retiraDaPosicao(int posicao)*

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



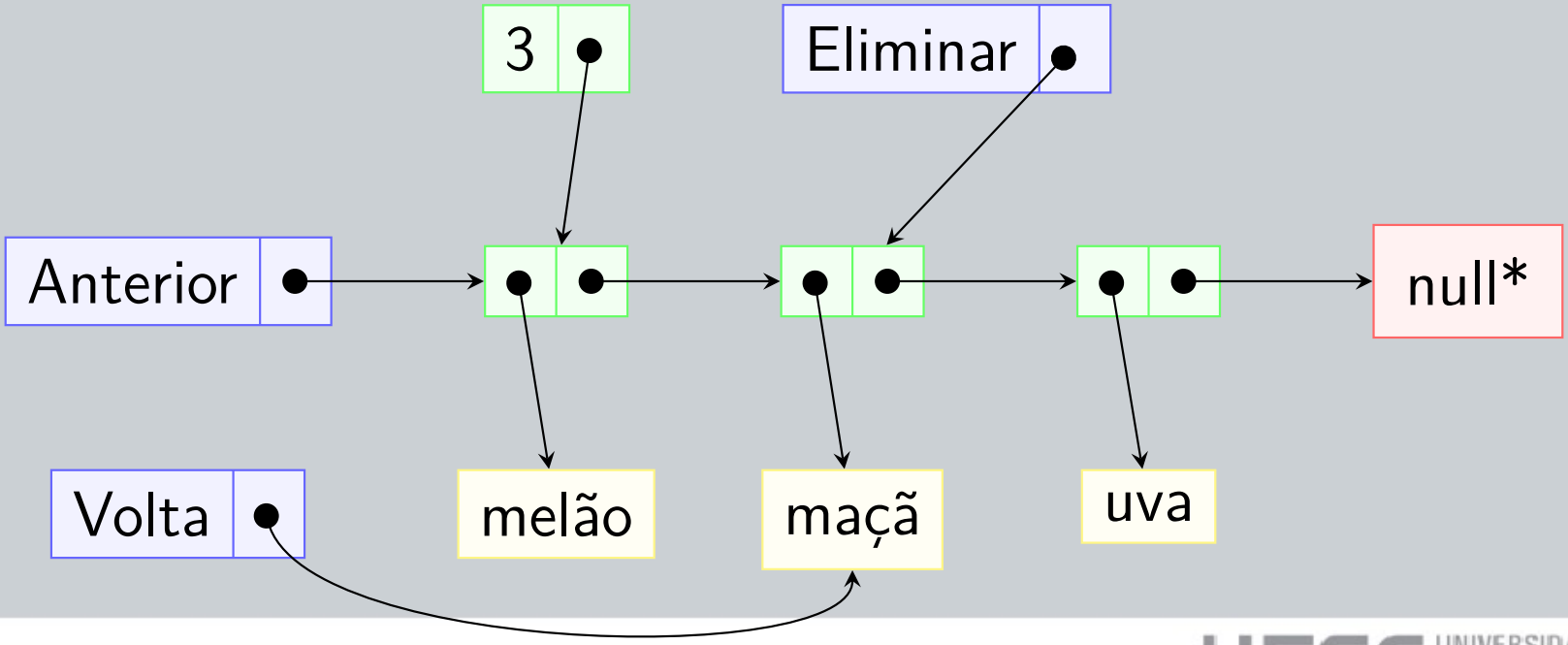
Função *T retiraDaPosicao(int posicao)*

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



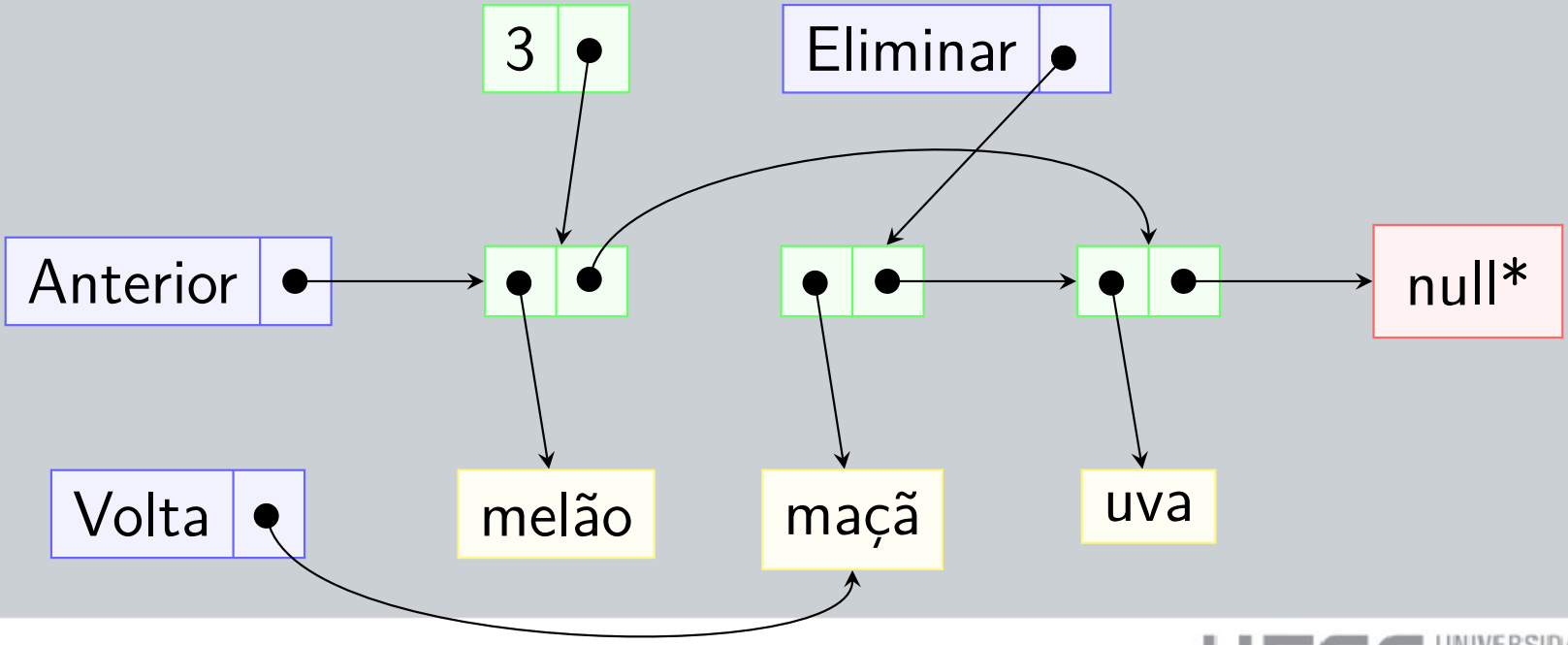
Função *T retiraDaPosicao(int posicao)*

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



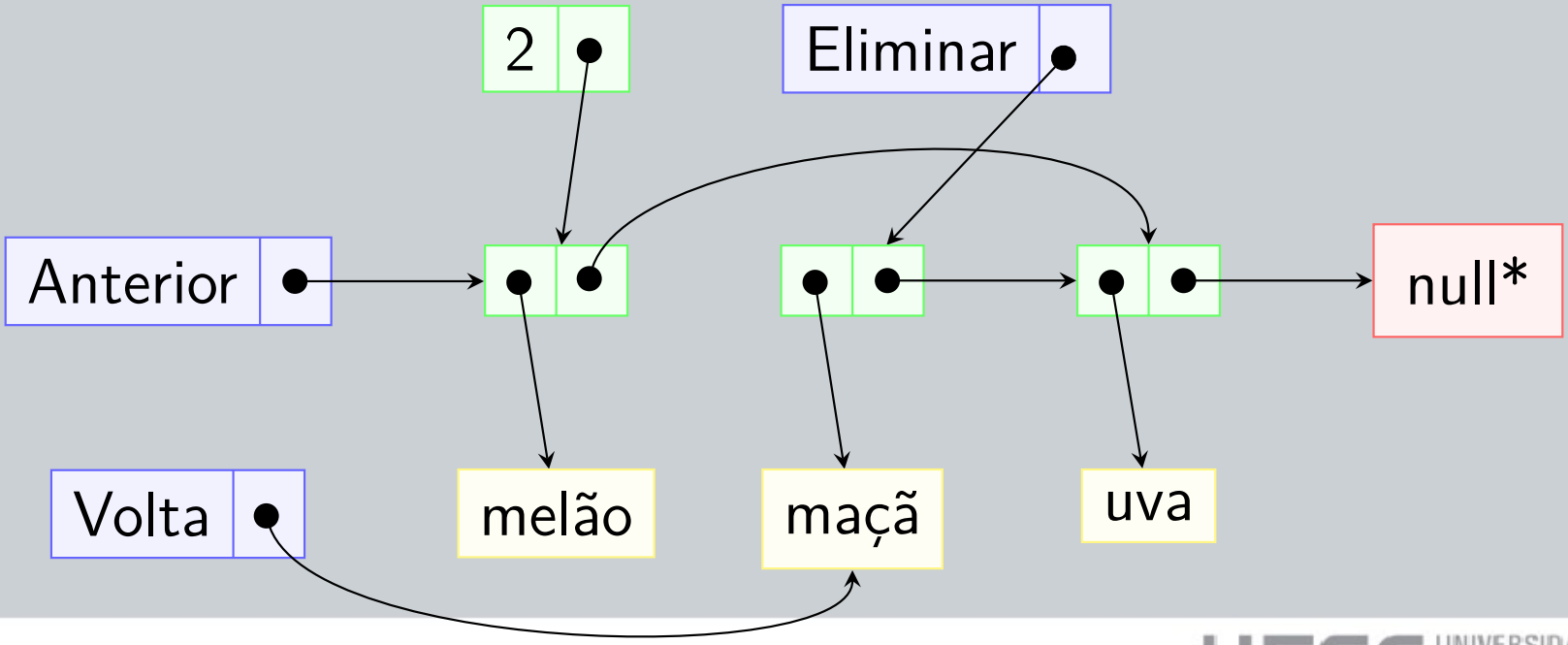
Função *T retiraDaPosicao(int posicao)*

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



Função *T retiraDaPosicao(int posicao)*

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



Função *T retiraDaPosicao(int posicao)*

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

Função *adicionaEmOrdem(T dado)*

- Necessitamos de uma função para comparar os dados (operator::>);
- Procuramos pela posição onde inserir comparando dados;
- Chamamos adicionaNaPosicao(T dado, int posicao).

Dica!

Podemos implementar um versão polimórfica de adicionaNaPosicao(T dado, int posicao) que é adicionaNaPosicao(T dado, Elemento* posicao)

Função *adicionaEmOrdem(T dado)*

```
adicionaEmOrdem(T dado)
 Elemento *atual; //Variável para caminhar.
 int posicao; // Posicao de Insercao.
 inicio
  SE (listaVazia()) ENTAO RETORNE(adicionaNoInicio(dado));
  SENAO
   atual <- _dados;
   posicao <- 1;
   ENQUANTO (atual._proximo ~= NULO E
             dado > atual._dado)) FACA
    //Encontrar posição para inserir.
    atual <- atual._proximo;
    posicao <- posicao + 1;
   FIM ENQUANTO
   SE (dado > atual._dado) ENTAO //Parou porque acabou a lista.
    RETORNE(adicionaNaPosicao(dado, posicao + 1));
   SENAO
    RETORNE(adicionaNaPosicao(dado, posicao));
   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()
 Elemento *atual, *anterior; //Variável auxiliar para caminhar.
 inicio
  SE (listaVazia()) ENTAO THROW(ERROLISTAVAZIA);
  SENAO
   atual <- _dados;
   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.
    LIBERE(anterior._dado);
    //Liberar o elemento que acabei de visitar.
    LIBERE(anterior);
   FIM ENQUANTO
  FIM SE
 fim;
```

Trabalho Lista Encadeada

- Implemente uma classe Lista todas as operações vistas;
- Implemente a lista usando Templates;
- Use as melhores práticas de orientação a objetos;
- Documente todas as classes, funções e atributos;
- Aplique os testes unitários disponíveis no moodle da disciplina para validar sua estrutura de dados;
- Entregue até a data definida no moodle.

Perguntas????

UNIVERSIDADE FEDERAL
DE SANTA CATARINA