

Estruturas de Dados

Lista Encadeada

Departamento de Informática e de Estatística
Prof. Jean Everson Martina
Prof. Aldo von Wangenheim

2016.2



UNIVERSIDADE FEDERAL
DE SANTA CATARINA

Listas Usando Vetores:

Desvantagens

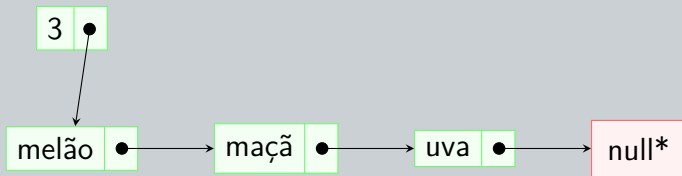
- Tamanho máximo fixo;
- Mesmo vazias ocupam um grande espaço de memória;
- Operações podem envolver muitos deslocamentos de dados:
 - Inclusão em uma posição ou no início;
 - Exclusão em uma posição ou no início.

24	
89	
12	
4	
55	
20	5

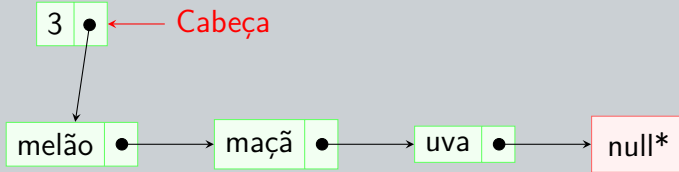
Definições de Lista Encadeada

- São listas onde cada elemento está armazenado em um objeto chamada elemento de lista;
- Cada elemento de lista referencia o próximo e só é alocado dinamicamente quando necessário;
- Para referenciar o primeiro elemento utilizamos um objeto cabeça de lista.

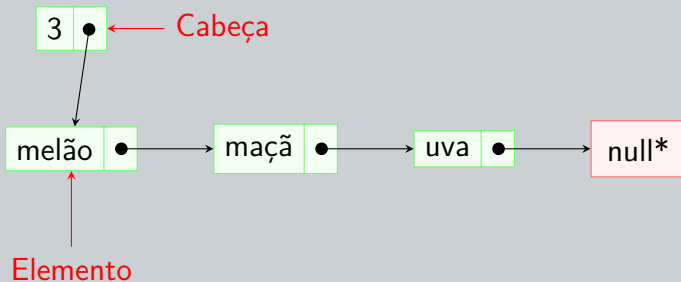
Lista Encadeada



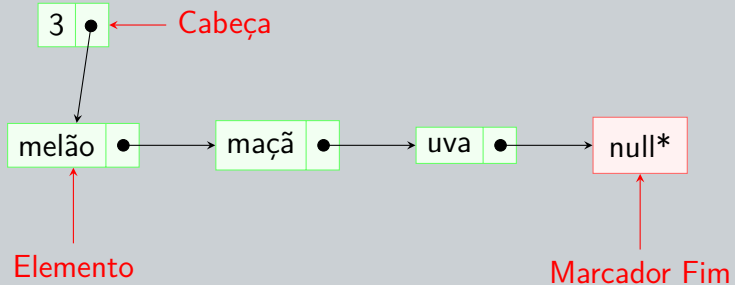
Lista Encadeada



Lista Encadeada



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.

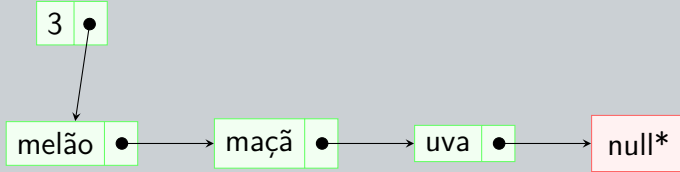
```
classe Lista {  
    Elemento *_dados;  
    inteiro _tamanho;  
};
```


Modelagem da Elemento de Lista

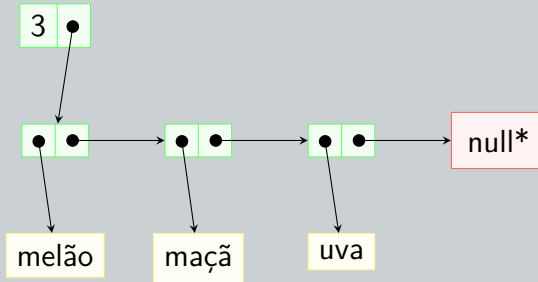
- Aspecto Estrutural:
 - Necessitamos um ponteiro para o próximo elemento da lista;
 - Necessitamos um campo do tipo da informação que vamos armazenar.

```
classe Elemento {  
    Elemento *_proximo;  
    T _info;  
};
```

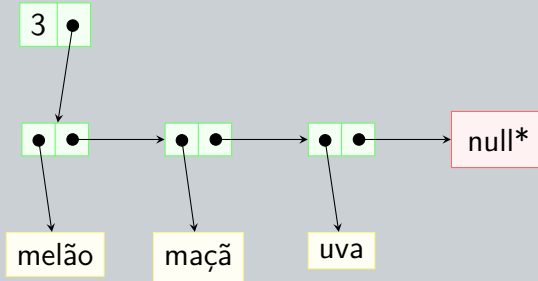
Modelagem de Lista Encadeada



Modelagem de Lista Encadeada



Modelagem de Lista Encadeada



Para tornar todos os algoritmos da lista mais genéricos, fazemos o campo info ser um ponteiro para um elemento de informação.

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;

```
classe Elemento {  
    Elemento *_proximo;  
    T *_info;  
};
```

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();`
 - `limpaLista();`
 - `~Lista();`
- Testar se a lista está vazia ou cheia e outros testes:
 - `bool listaVazia();`
 - `int posicao(dado);`
 - `bool contem(dado);`

Modelagem da Lista Encadeada

- Colocar e retirar dados da lista:
 - adiciona(T dado);
 - adicionaNoInicio(T dado);
 - adicionaNaPosicao(T dado, int posicao);
 - adicionaEmOrdem(T dado);
 - T retira();
 - T retiraDoInicio();
 - T retiraDaPosicao(int posicao);
 - T retiraEspecifico(dado);

Método *Lista()*

- Inicializamos o ponteiro para nulo;
- Inicializamos o tamanho para “0”;

```
Lista()  
inicio  
    _dados = null;  
    _tamanho <- 0;  
fim;
```

Método *~Lista()*

- Chamamos DestroiLista();

```
~Lista()  
inicio  
    DestroiLista();  
fim;
```

Método *listaVazia()*

```
bool listaVazia()  
inicio  
SE (_tamanho = 0) ENTÃO  
    RETORNE(Verdadeiro)  
SENÃO  
    RETORNE(Falso);  
fim;
```

- Um algoritmo ListaCheia não existe na Lista Encadeada;

Método *listaVazia()*

```
bool listaVazia()  
inicio  
SE (_tamanho = 0) ENTÃO  
    RETORNE(Verdadeiro)  
SENÃO  
    RETORNE(Falso);  
fim;
```

- Um algoritmo ListaCheia não existe na Lista Encadeada;
- Verificar se houve espaço na memória para um novo elemento será responsabilidade de cada operação de adição.

Método *adicionaNoInicio(T dado)*

- Testamos se é possível alocar um elemento;

Método *adicionaNoInicio(T dado)*

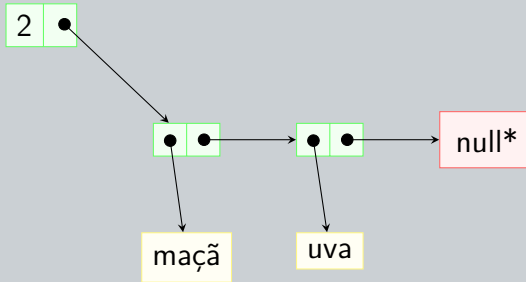
- Testamos se é possível alocar um elemento;
- Fazemos o próximo deste novo elemento ser o primeiro da lista;

Método *adicionaNoInicio(T dado)*

- Testamos se é possível alocar um elemento;
- Fazemos o próximo deste novo elemento ser o primeiro da lista;
- Fazemos a cabeça de lista apontar para o novo elemento.

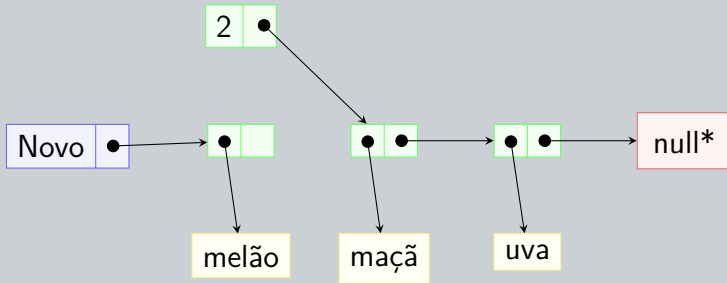
Método *adicionaNoInicio(T dado)*

- Testamos se é possível alocar um elemento;
- Fazemos o próximo deste novo elemento ser o primeiro da lista;
- Fazemos a cabeça de lista apontar para o novo elemento.



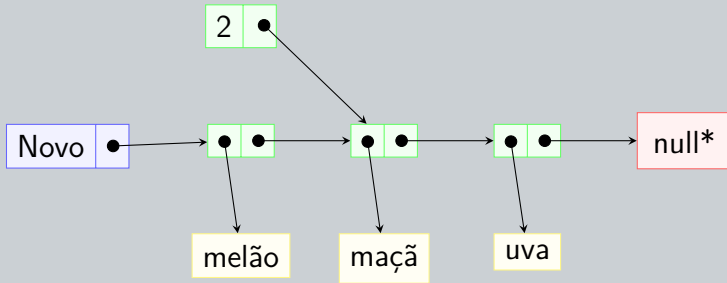
Método *adicionaNoInicio(T dado)*

- Testamos se é possível alocar um elemento;
- Fazemos o próximo deste novo elemento ser o primeiro da lista;
- Fazemos a cabeça de lista apontar para o novo elemento.



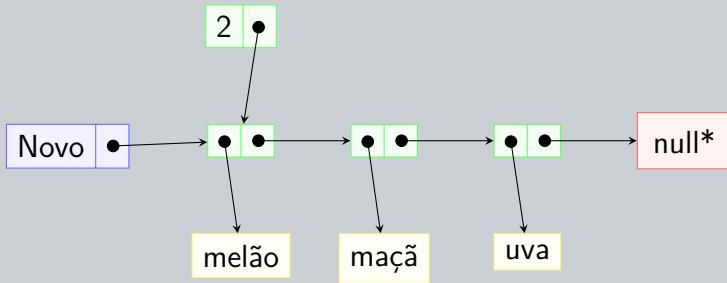
Método *adicionaNoInicio(T dado)*

- Testamos se é possível alocar um elemento;
- Fazemos o próximo deste novo elemento ser o primeiro da lista;
- Fazemos a cabeça de lista apontar para o novo elemento.



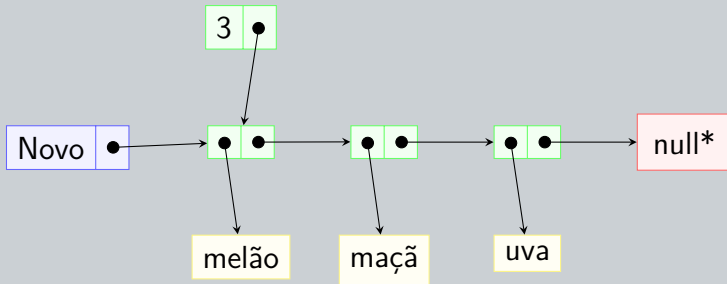
Método *adicionaNoInicio*(*T dado*)

- Testamos se é possível alocar um elemento;
- Fazemos o próximo deste novo elemento ser o primeiro da lista;
- Fazemos a cabeça de lista apontar para o novo elemento.



Método *adicionaNoInicio(T dado)*

- Testamos se é possível alocar um elemento;
- Fazemos o próximo deste novo elemento ser o primeiro da lista;
- Fazemos a cabeça de lista apontar para o novo elemento.



Método *adicionaNoInicio(T dado)*

```
adicionaNoInicio(T dado)
  Elemento *novo; //Variável auxiliar.
  início
    novo <- aloque(Elemento);
    SE (novo = NULO) ENTAO
      THROW(ERROLISTACHEIA);
    SENAO
      novo->_proximo <- _dados;
      novo->_info <- dado;
      _dados <- novo;
      _tamanho <- _tamanho + 1;
    FIM SE
  fim;
```

Método *adicionaNoInicio(T dado)*

```
adicionaNoInicio(T dado)
  Elemento *novo; //Variável auxiliar.
  início
    novo <- aloque(Elemento);
    SE (novo = NULO) ENTAO
      THROW(ERROLISTACHEIA);
    SENA0
      novo->_proximo <- _dados;
      novo->_info <- dado;
      _dados <- novo;
      _tamanho <- _tamanho + 1;
    FIM SE
  fim;
```

Método *adicionaNoInicio(T dado)*

```
adicionaNoInicio(T dado)
  Elemento *novo; //Variável auxiliar.
  início
    novo <- aloque(Elemento);
    SE (novo = NULO) ENTAO
      THROW(ERROLISTACHEIA);
    SENAO
      novo->_proximo <- _dados;
      novo->_info <- dado;
      _dados <- novo;
      _tamanho <- _tamanho + 1;
    FIM SE
  fim;
```

Método *T retiraDoInicio()*

- Testamos se há elementos;

Método *T retiraDoInicio()*

- Testamos se há elementos;
- Decrementamos o tamanho;

Método *T retiraDoInicio()*

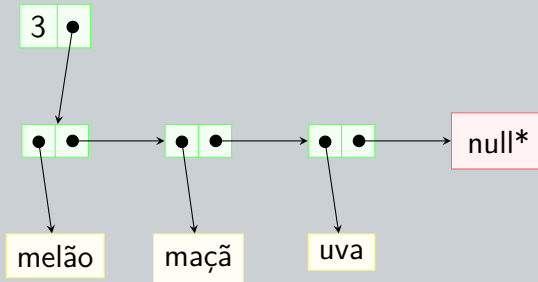
- Testamos se há elementos;
- Decrementamos o tamanho;
- Liberamos a memória do elemento;

Método *T retiraDoInicio()*

- Testamos se há elementos;
- Decrementamos o tamanho;
- Liberamos a memória do elemento;
- Devolvemos a informação.

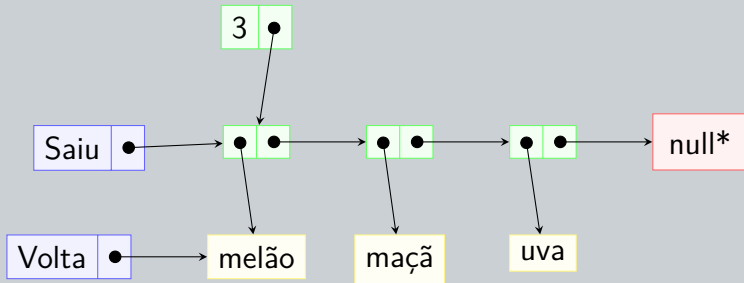
Método *T retiraDoInicio()*

- Testamos se há elementos;
- Decrementamos o tamanho;
- Liberamos a memória do elemento;
- Devolvemos a informação.



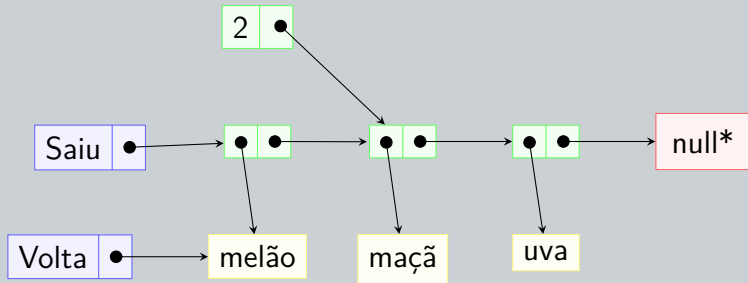
Método *T retiraDoInicio()*

- Testamos se há elementos;
- Decrementamos o tamanho;
- Liberamos a memória do elemento;
- Devolvemos a informação.



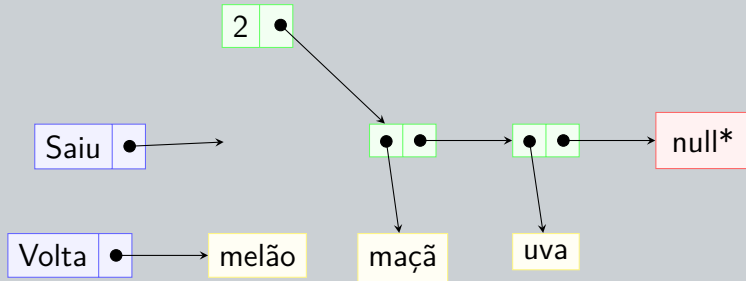
Método *T retiraDoInicio()*

- Testamos se há elementos;
- Decrementamos o tamanho;
- Liberamos a memória do elemento;
- Devolvemos a informação.



Método *T retiraDoInicio()*

- Testamos se há elementos;
- Decrementamos o tamanho;
- Liberamos a memória do elemento;
- Devolvemos a informação.



Método *T retiraDoInicio()*

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


Método *T retiraDoInicio()*

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

Método *T retiraDoInicio()*

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

Método *T retiraDoInicio()*

```
T retiraDoInicio()  
  Elemento *saiu; //Variável auxiliar elemento.  
  T *volta; //Variável auxiliar tipo T.  
  início  
  SE (listaVazia()) ENTAO  
    THROW(ERROLISTAVAZIA);  
  SENA0  
    saiu <- _dados;  
    volta <- saiu->info;  
    _dados <- saiu->próximo;  
    _tamanho <- _tamanho - 1;  
    LIBERE(saiu);  
    RETORNE(volta);  
  FIM SE  
  fim;
```

Método *eliminaDoInicio()*

```
eliminaDoInicio()  
  Elemento *saiu; //Variável auxiliar elemento.  
início  
SE (listaVazia()) ENTAO  
  THROW(ERROLISTAVAZIA);  
SENAO  
  saiu <- _dados;  
  volta <- saiu->info;  
  _dados <- saiu->próximo;  
  _tamanho <- _tamanho - 1;  
  LIBERE(saiu);  
  LIBERE(saiu->info);  
FIM SE  
fim;
```

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.

Método *adicionaNaPosicao*(*T dado*,
int posicao)

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

Método *adicionaNaPosicao*(*T dado*, *int posicao*)

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

Método *adicionaNaPosicao*(*T dado*, *int posicao*)

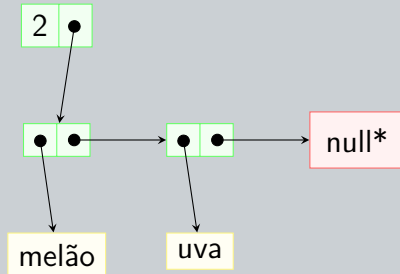
- Testamos se a posição existe e se é possível alocar;
- Caminhamos até a posição;
- Adicionamos o novo dado na posição;

Método *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.

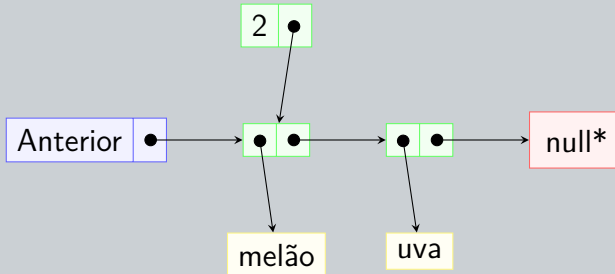
Método *adicionaNaPosicao(T dado, int posicao)*

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



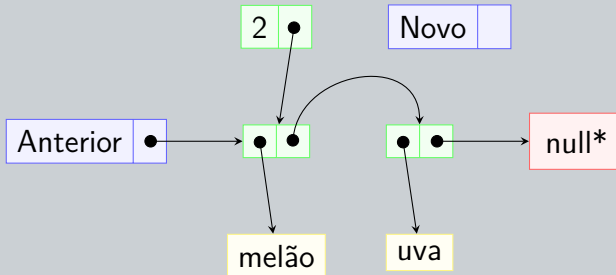
Método *adicionaNaPosicao*(*T* dado, *int posicao*)

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



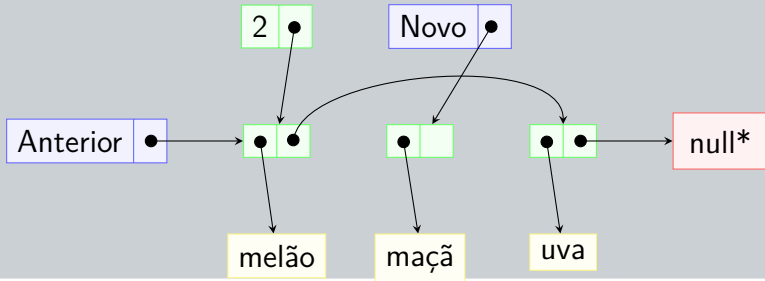
Método *adicionaNaPosicao*(*T dado*, *int posicao*)

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



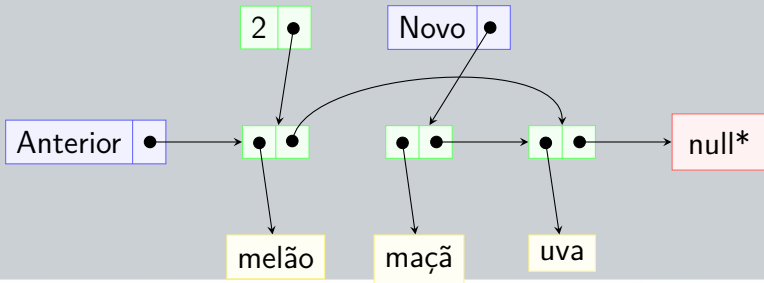
Método *adicionaNaPosicao*(*T* dado, *int posicao*)

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



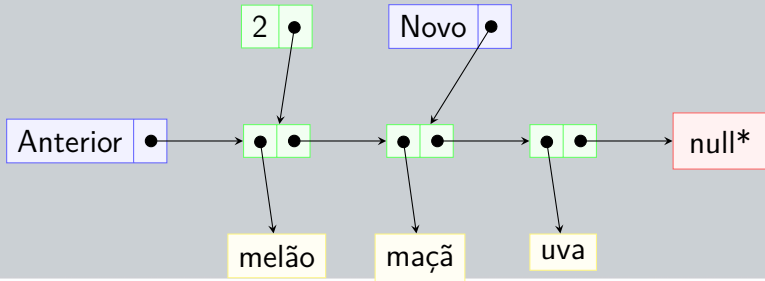
Método *adicionaNaPosicao*(*T* dado, *int posicao*)

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



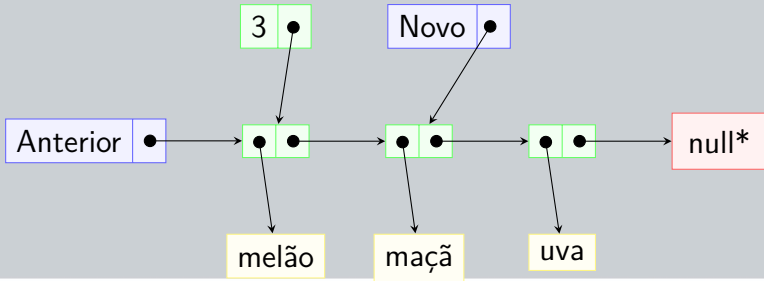
Método *adicionaNaPosicao*(*T dado*, *int posicao*)

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



Método *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.



Método *adicionaNaPosicao*(*T* dado, *int posicao*)

```
adicionaNaPosicao(T dado, int posicao)
  Elemento *novo, *anterior; // auxiliares.
  inicio
  SE (posicao > _tamanho + 1) ENTAO THROW(ERROPOSICAO);
  SENA0
  SE (posicao = 1) ENTAO RETORNE(adicionaNoInicio(info));
  SENA0
  novo <- aloque(Elemento);
  SE (novo = NULO) ENTÃO THROW(ERROLISTACHEIA);
  SENA0
  anterior <- _dados;
  REPITA (posicao - 2) VEZES
    anterior <- anterior->_proximo;
  novo->_proximo <- anterior->_proximo;
  novo->_info <- info;
  anterior->_proximo <- novo;
  _tamanho <- _tamanho + 1;
  FIM SE
  FIM SE
  FIM SE
fim;
```

Método *adicionaNaPosicao*(*T* dado, *int posicao*)

```
adicionaNaPosicao(T dado, int posicao)
  Elemento *novo, *anterior; // auxiliares.
  inicio
  SE (posicao > _tamanho + 1) ENTAO THROW(ERROPOSSICAO);
  SENA0
  SE (posicao = 1) ENTAO RETORNE(adicionaNoInicio(info));
  SENA0
  novo <- aloque(Elemento);
  SE (novo = NULO) ENTÃO THROW(ERROLISTACHEIA);
  SENA0
  anterior <- _dados;
  REPITA (posicao - 2) VEZES
    anterior <- anterior->_proximo;
  novo->_proximo <- anterior->_proximo;
  novo->_info <- info;
  anterior->_proximo <- novo;
  _tamanho <- _tamanho + 1;
  FIM SE
  FIM SE
  FIM SE
fim;
```

Método *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(adicionaNoInicio(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->_info <- info;
    anterior->_proximo <- novo;
    _tamanho <- _tamanho + 1;
  FIM SE
  FIM SE
  FIM SE
fim;
```

Método *adicionaNaPosicao*(*T* dado, *int* posicao)

```
adicionaNaPosicao(T dado, int posicao)
  Elemento *novo, *anterior; // auxiliares.
  inicio
  SE (posicao > _tamanho + 1) ENTAO THROW(ERROPOSICAO);
  SENA0
  SE (posicao = 1) ENTAO RETORNE(adicionaNoInicio(info));
  SENA0
  novo <- aloque(Elemento);
  SE (novo = NULO) ENTÃO THROW(ERROLISTACHEIA);
  SENA0
  anterior <- _dados;
  REPITA (posicao - 2) VEZES
    anterior <- anterior->_proximo;
  novo->_proximo <- anterior->_proximo;
  novo->_info <- info;
  anterior->_proximo <- novo;
  _tamanho <- _tamanho + 1;
  FIM SE
  FIM SE
  FIM SE
fim;
```

Método *adicionaNaPosicao*(*T* dado, *int posicao*)

```
adicionaNaPosicao(T dado, int posicao)
  Elemento *novo, *anterior; // auxiliares.
  inicio
  SE (posicao > _tamanho + 1) ENTAO THROW(ERROPOSICAO);
  SENA0
  SE (posicao = 1) ENTAO RETORNE(adicionaNoInicio(info));
  SENA0
  novo <- aloque(Elemento);
  SE (novo = NULO) ENTÃO THROW(ERROLISTACHEIA);
  SENA0
    anterior <- _dados;
    REPITA (posicao - 2) VEZES
      anterior <- anterior->_proximo;
    novo->_proximo <- anterior->_proximo;
    novo->_info <- info;
    anterior->_proximo <- novo;
    _tamanho <- _tamanho + 1;
  FIM SE
  FIM SE
  FIM SE
fim;
```

Método *adicionaNaPosicao*(*T* dado, *int posicao*)

```
adicionaNaPosicao(T dado, int posicao)
  Elemento *novo, *anterior; // auxiliares.
  inicio
  SE (posicao > _tamanho + 1) ENTAO THROW(ERROPOSICAO);
  SENA0
  SE (posicao = 1) ENTAO RETORNE(adicionaNoInicio(info));
  SENA0
  novo <- aloque(Elemento);
  SE (novo = NULO) ENTÃO THROW(ERROLISTACHEIA);
  SENA0
  anterior <- _dados;
  REPITA (posicao - 2) VEZES
    anterior <- anterior->_proximo;
    novo->_proximo <- anterior->_proximo;
    novo->_info <- info;
    anterior->_proximo <- novo;
    _tamanho <- _tamanho + 1;
  FIM SE
  FIM SE
  FIM SE
fim;
```

Método *T retiraDaPosicao(int posicao)*

- Testamos se a posição existe;

Método *T retiraDaPosicao(int posicao)*

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

Método *T retiraDaPosicao(int posicao)*

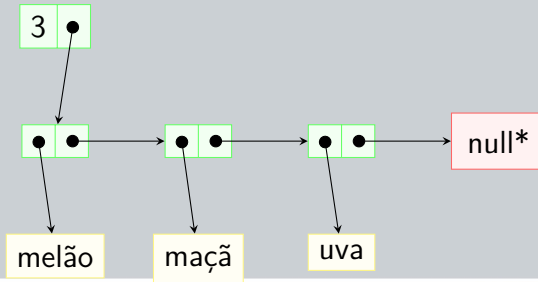
- Testamos se a posição existe;
- Caminhamos até a posição;
- Retiramos o dado da posição;

Método *T retiraDaPosicao(int posicao)*

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

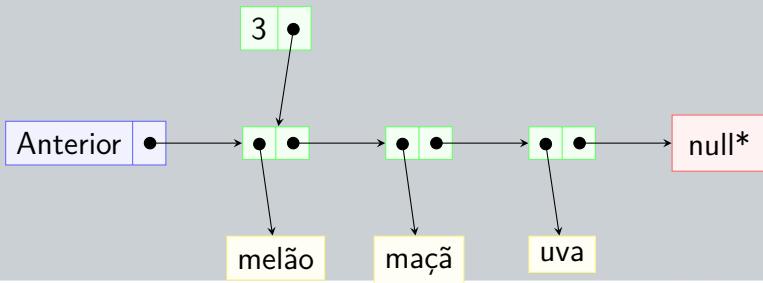
Método *T retiraDaPosicao(int posicao)*

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



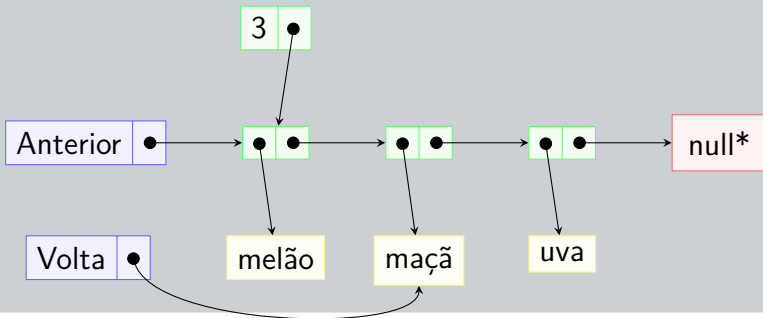
Método *T retiraDaPosicao(int posicao)*

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



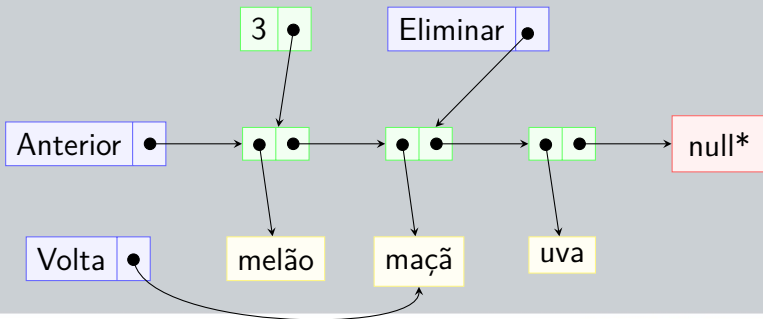
Método *T* *retiraDaPosicao*(int *posicao*)

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



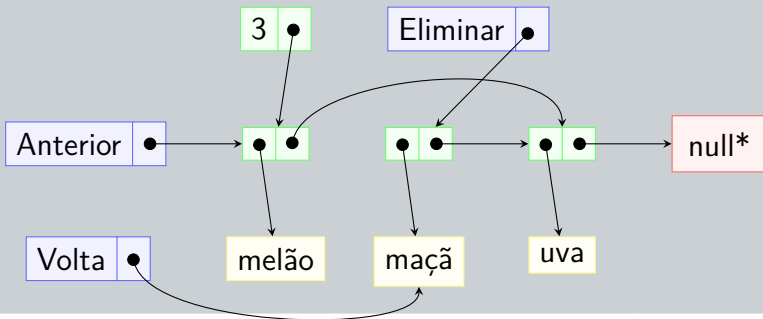
Método *T retiraDaPosicao(int posicao)*

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



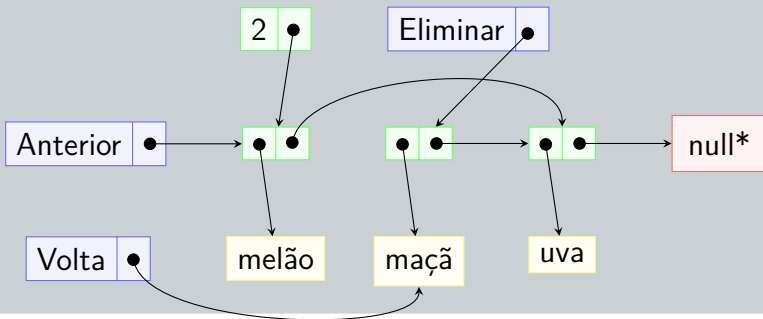
Método *T retiraDaPosicao(int posicao)*

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



Método *T retiraDaPosicao(int posicao)*

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



Método *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);
  SENA0
  SE (posicao = 1) ENTAO RETORNE(retiraDoInicio());
  SENA0
  anterior <- _dados;
  REPITA (posicao - 2) VEZES
    anterior <- anterior->_proximo;
  eliminar <- anterior->_proximo;
  volta <- eliminar->_info;
  anterior->_proximo <- eliminar->_proximo;
  _tamanho <- _tamanho - 1;
  LIBERE(eliminar);
  RETORNE(volta);
  FIM SE
  FIM SE
fim;
```

Método *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);
  SENA0
    SE (posicao = 1) ENTAO RETORNE(retiraDoInicio());
  SENA0
    anterior <- _dados;
    REPITA (posicao - 2) VEZES
      anterior <- anterior->_proximo;
    eliminar <- anterior->_proximo;
    volta <- eliminar->_info;
    anterior->_proximo <- eliminar->_proximo;
    _tamanho <- _tamanho - 1;
    LIBERE(eliminar);
    RETORNE(volta);
  FIM SE
FIM SE
fim;
```

Método *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->_info;
    anterior->_proximo <- eliminar->_proximo;
    _tamanho <- _tamanho - 1;
    LIBERE(eliminar);
    RETORNE(volta);
  FIM SE
FIM SE
fim;
```

Método *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->_info;
      anterior->_proximo <- eliminar->_proximo;
      _tamanho <- _tamanho - 1;
      LIBERE(eliminar);
      RETORNE(volta);
  FIM SE
FIM SE
fim;
```

Método *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);
    SENA0
      SE (posicao = 1) ENTAO RETORNE(retiraDoInicio());
      SENA0
        anterior <- _dados;
        REPITA (posicao - 2) VEZES
          anterior <- anterior->_proximo;
          eliminar <- anterior->_proximo;
          volta <- eliminar->_info;
          anterior->_proximo <- eliminar->_proximo;
          _tamanho <- _tamanho - 1;
          LIBERE(eliminar);
          RETORNE(volta);
        FIM SE
      FIM SE
  fim;
```

Método *adicionaEmOrdem*(*T dado*)

- Necessitamos de um método 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*)

Método *adicionaEmOrdem(T dado)*

```
adicionaEmOrdem(T dado)
  Elemento *atual; //Variável para caminhar.
  int posicao; // Posicao de Insercao.
  inicio
    SE (listaVazia()) ENTAO RETORNE(adicionaNoInicio(dado));
    SENA0
      atual <- _dados;
      posicao <- 1;
      ENQUANTO (atual->_proximo ~= NULO E
                dado > atual->_info)) FACA
        //Encontrar posição para inserir.
        atual <- atual->_proximo;
        posicao <- posicao + 1;
      FIM ENQUANTO
      SE (dado > atual->_info) ENTAO //Parou porque acabou a lista.
        RETORNE(adicionaNaPosicao(dado, posicao + 1));
      SENA0
        RETORNE(adicionaNaPosicao(dado, posicao));
      FIM SE
    FIM SE
  fim;
```


Método *adicionaEmOrdem(T dado)*

```
adicionaEmOrdem(T dado)
  Elemento *atual; //Variável para caminhar.
  int posicao; // Posicao de Insercao.
  inicio
    SE (listaVazia()) ENTAO RETORNE(adicionaNoInicio(dado));
  SENA0
    atual <- _dados;
    posicao <- 1;
    ENQUANTO (atual->_proximo ~= NULO E
              dado > atual->_info)) FACA
      //Encontrar posição para inserir.
      atual <- atual->_proximo;
      posicao <- posicao + 1;
    FIM ENQUANTO
    SE (dado > atual->_info) ENTAO //Parou porque acabou a lista.
      RETORNE(adicionaNaPosicao(dado, posicao + 1));
    SENA0
      RETORNE(adicionaNaPosicao(dado, posicao));
    FIM SE
  FIM SE
fim;
```

Método *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->_info)) FACA
        //Encontrar posição para inserir.
        atual <- atual->_proximo;
        posicao <- posicao + 1;
      FIM ENQUANTO
      SE (dado > atual->_info) ENTAO //Parou porque acabou a lista.
        RETORNE(adicionaNaPosicao(dado, posicao + 1));
      SENAO
        RETORNE(adicionaNaPosicao(dado, posicao));
      FIM SE
    FIM SE
  fim;
```

Método *adicionaEmOrdem*(*T dado*)

```
adicionaEmOrdem(T dado)
  Elemento *atual; //Variável para caminhar.
  int posicao; // Posicao de Insercao.
  inicio
    SE (listaVazia()) ENTAO RETORNE(adicionaNoInicio(dado));
    SENA0
      atual <- _dados;
      posicao <- 1;
      ENQUANTO (atual->_proximo ~= NULO E
                dado > atual->_info)) FACA
        //Encontrar posição para inserir.
        atual <- atual->_proximo;
        posicao <- posicao + 1;
      FIM ENQUANTO
      SE (dado > atual->_info) ENTAO //Parou porque acabou a lista.
        RETORNE(adicionaNaPosicao(dado, posicao + 1));
      SENA0
        RETORNE(adicionaNaPosicao(dado, posicao));
      FIM SE
    FIM SE
  fim;
```

Método *adicionaEmOrdem(T dado)*

```
adicionaEmOrdem(T dado)
  Elemento *atual; //Variável para caminhar.
  int posicao; // Posicao de Insercao.
  inicio
    SE (listaVazia()) ENTAO RETORNE(adicionaNoInicio(dado));
    SENA0
      atual <- _dados;
      posicao <- 1;
      ENQUANTO (atual->_proximo ~= NULO E
                dado > atual->_info)) FACA
        //Encontrar posição para inserir.
        atual <- atual->_proximo;
        posicao <- posicao + 1;
      FIM ENQUANTO
      SE (dado > atual->_info) ENTAO //Parou porque acabou a lista.
        RETORNE(adicionaNaPosicao(dado, posicao + 1));
      SENA0
        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);

Método *destroiLista()*

```
destroiLista()  
  Elemento *atual, *anterior; //Variável auxiliar para caminhar.  
  inicio  
  SE (listaVazia()) ENTAO THROW(ERROLISTAVAZIA);  
  SENA  
    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->_info);  
      //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;
- Implemente a lista com um numero de elementos variável definido na instanciação;
- Use as melhores práticas de orientação a objetos;
- Documente todas as classes, métodos 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



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