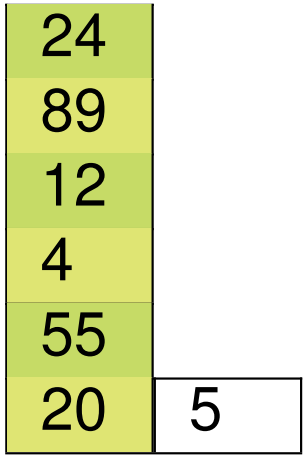
Estruturas de Dados

A. G. Silva, A. von Wangenheim, J. E. Martina

Revisado em 30 de agosto de 2018

Listas em vetor – desvantagens

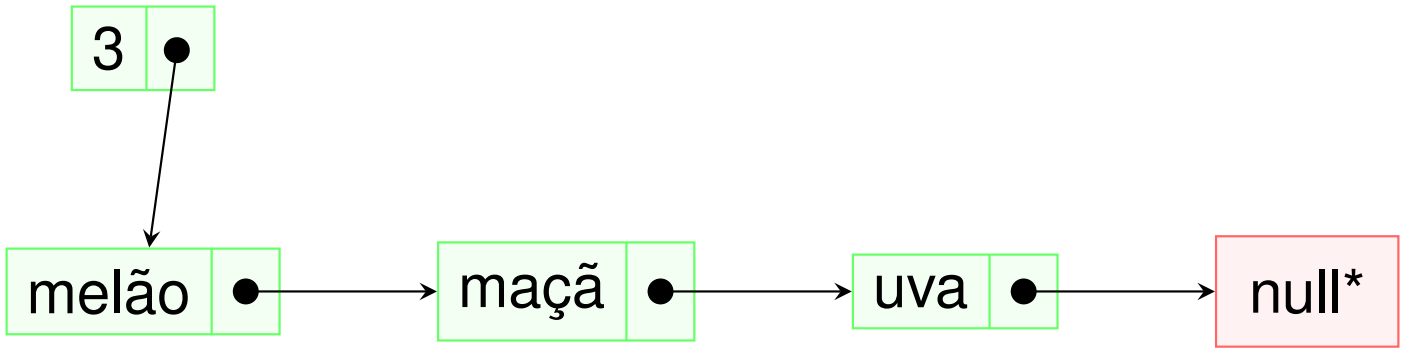
- Tamanho máximo fixo;
- Mesmo vazias ou quase vazias, ocupam grande espaço em memória;
- Operações podem envolver muitos deslocamentos de dados:
  - Inserção em uma posição ou no início;
  - Remoção de uma posição ou no início.



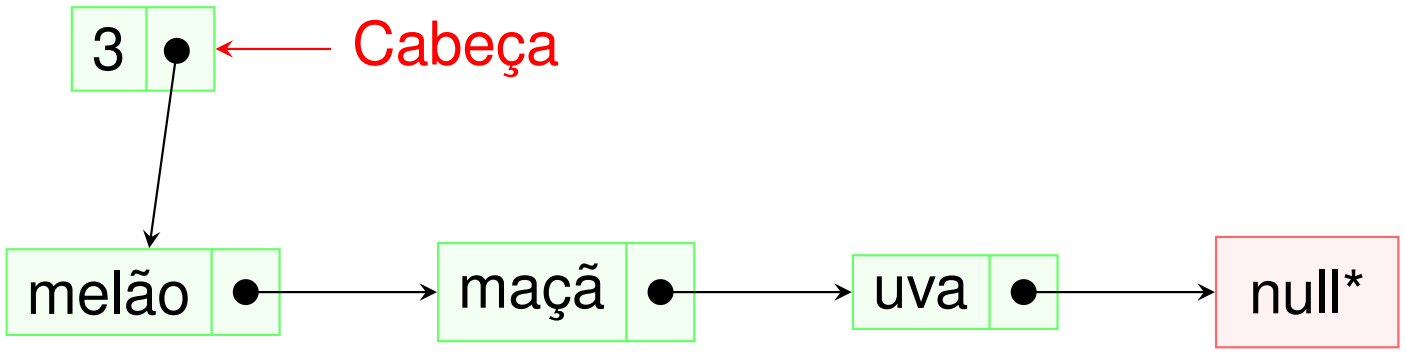
Definições de Lista Encadeada

- 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, utiliza-se um objeto cabeça de lista.

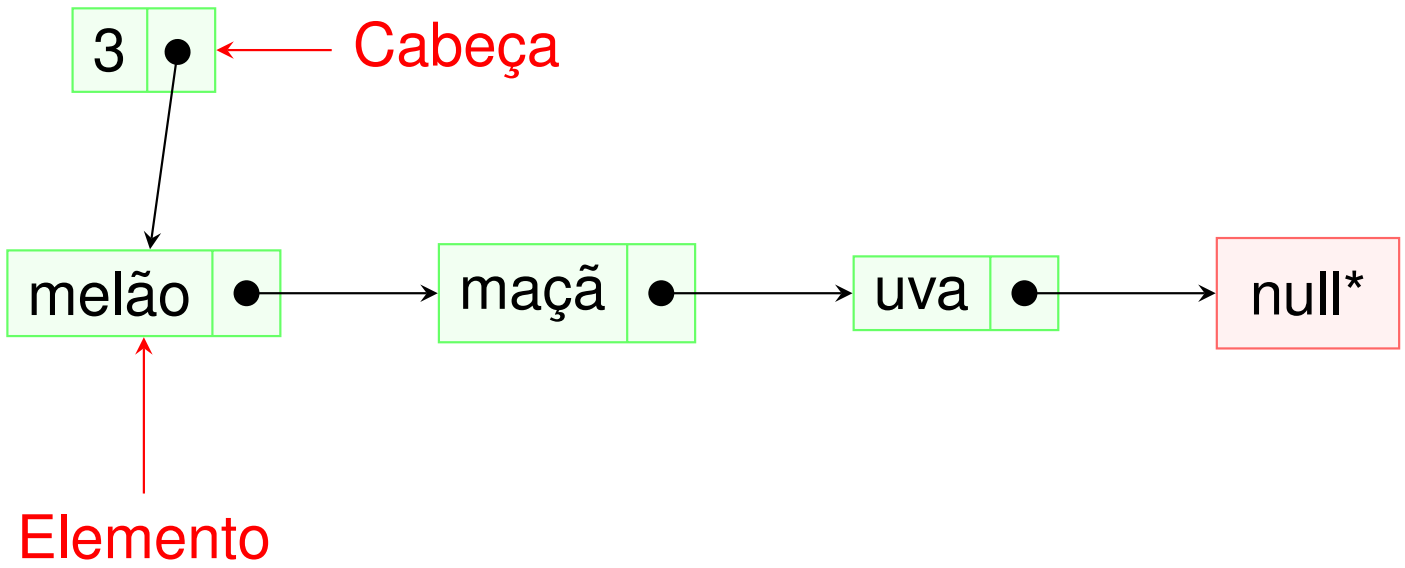
Lista Encadeada



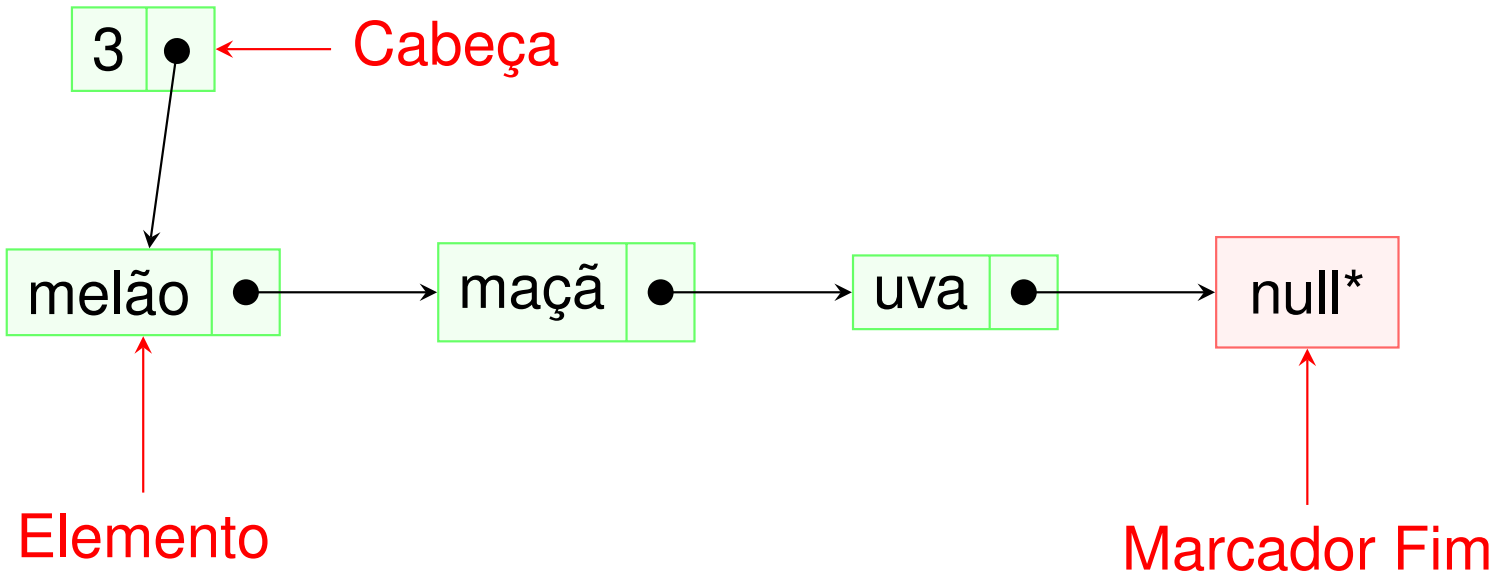
Lista Encadeada



Lista Encadeada



Lista Encadeada



Modelagem da cabeça de Lista

- Aspecto estrutural:
  - É necessário um ponteiro para o primeiro elemento da lista;
  - É necessário um inteiro para indicar a quantidade de elementos da lista.

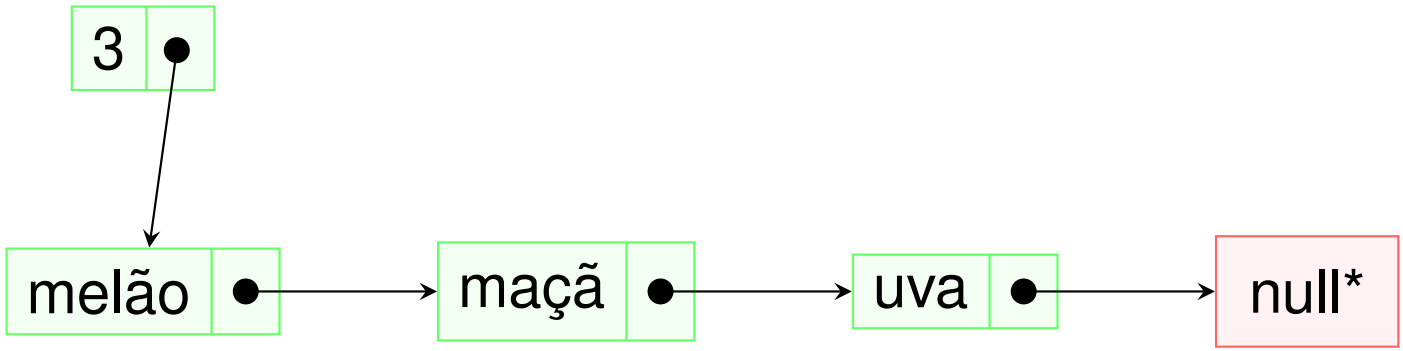
```
classe Lista {
    Elemento *_dados;
    inteiro _tamanho;
};
```

Modelagem do elemento de Lista

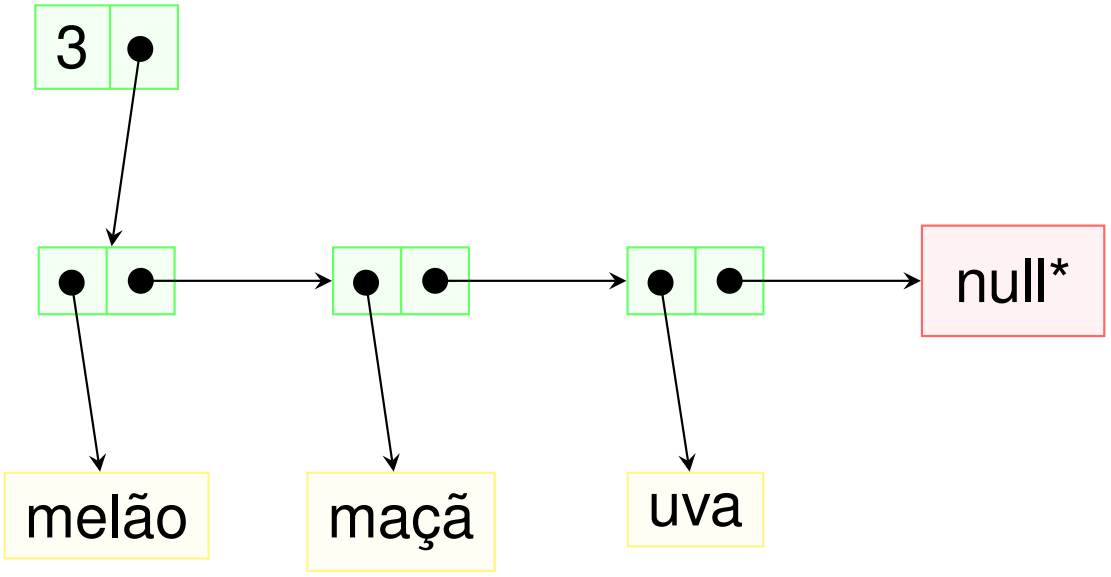
- Aspecto estrutural:
  - É necessário um ponteiro para o próximo elemento da lista;
  - É necessário um campo do tipo da informação a armazenar.

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

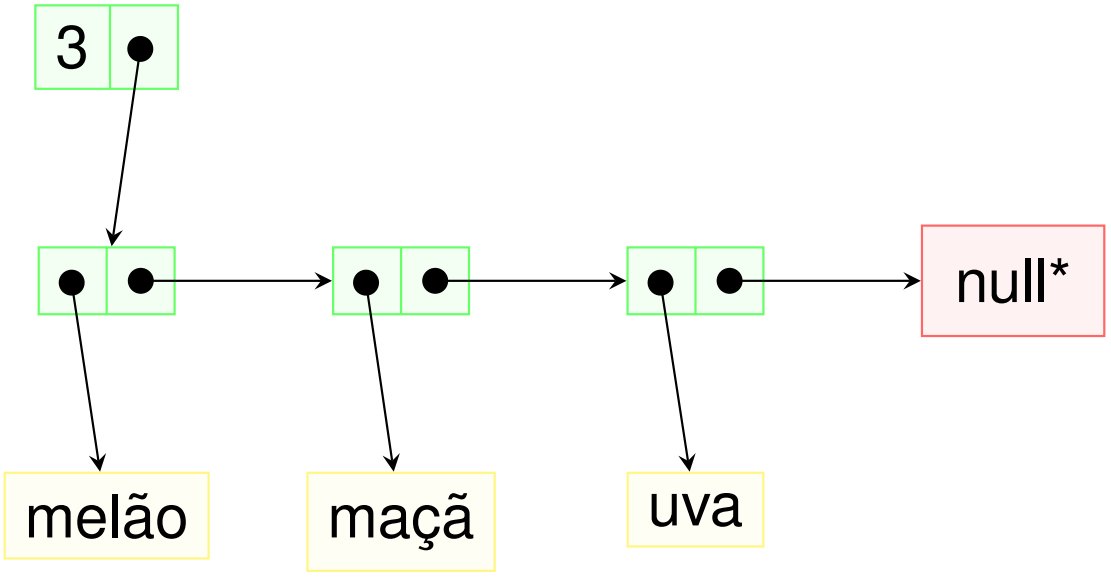
Modelagem de Lista Encadeada



Modelagem de Lista Encadeada



Modelagem de Lista Encadeada



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

Modelagem da Elemento de Lista

- Aspecto estrutural:
  - É necessário um ponteiro para o próximo elemento da lista;
  - É necessário um ponteiro do tipo da informação a 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:
  - É preciso inserir e remover dados da lista;
  - É preciso testar se a lista está vazia (dentre outros testes);
  - É preciso 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(T *dado);`
  - `bool contem(T *dado);`

Modelagem da Lista Encadeada

- Inserir e remover 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);`
  - `eliminaDoInicio();`

Método `Lista()`

- Inicializa o ponteiro para NULO;
- Inicializa o tamanho para “0”;

```
Lista()
    inicio
        _dados ← NULO;
        _tamanho ← 0;
    fim
```

Método ~Lista()

- Executa `limpaLista();`

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

Método `listaVazia()`

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

- Não é preciso efetuar o teste de lista cheia em Lista Encadeada;
- A verificação de disponibilidade de espaço em memória para cada novo elemento será responsabilidade de cada operação de inserção.

Método `adicionaNoInicio(T *dado)`

- Verifica a possibilidade de alocação de um elemento;

Método `adicionaNoInicio(T *dado)`

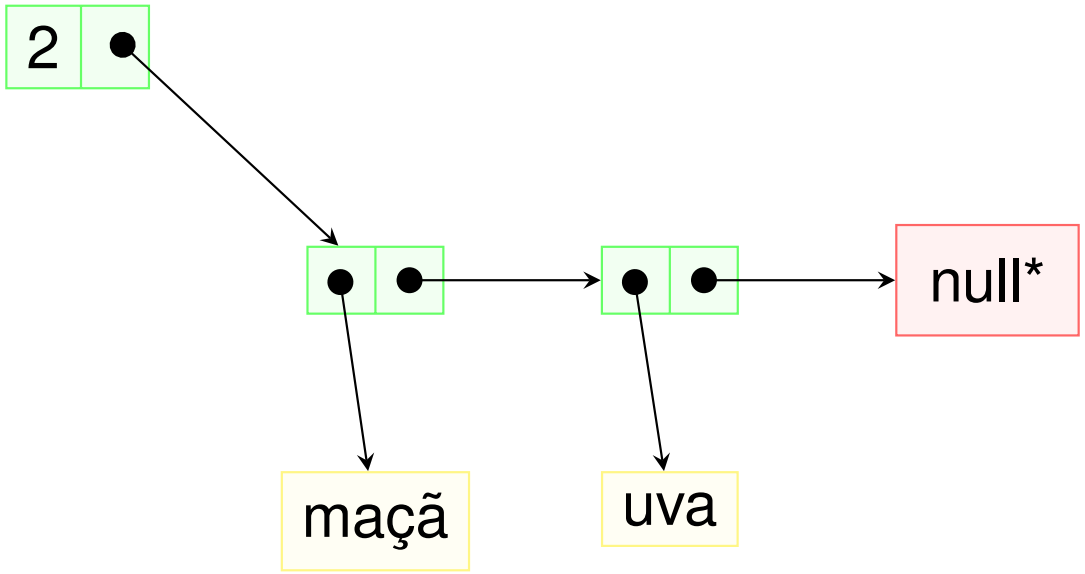
- Verifica a possibilidade de alocação de um elemento;
- O próximo deste novo elemento passa ser o primeiro da lista;

Método `adicionaNoInicio(T *dado)`

- Verifica a possibilidade de alocação de um elemento;
- O próximo deste novo elemento passa ser o primeiro da lista;
- A cabeça de lista passa a apontar para o novo elemento.

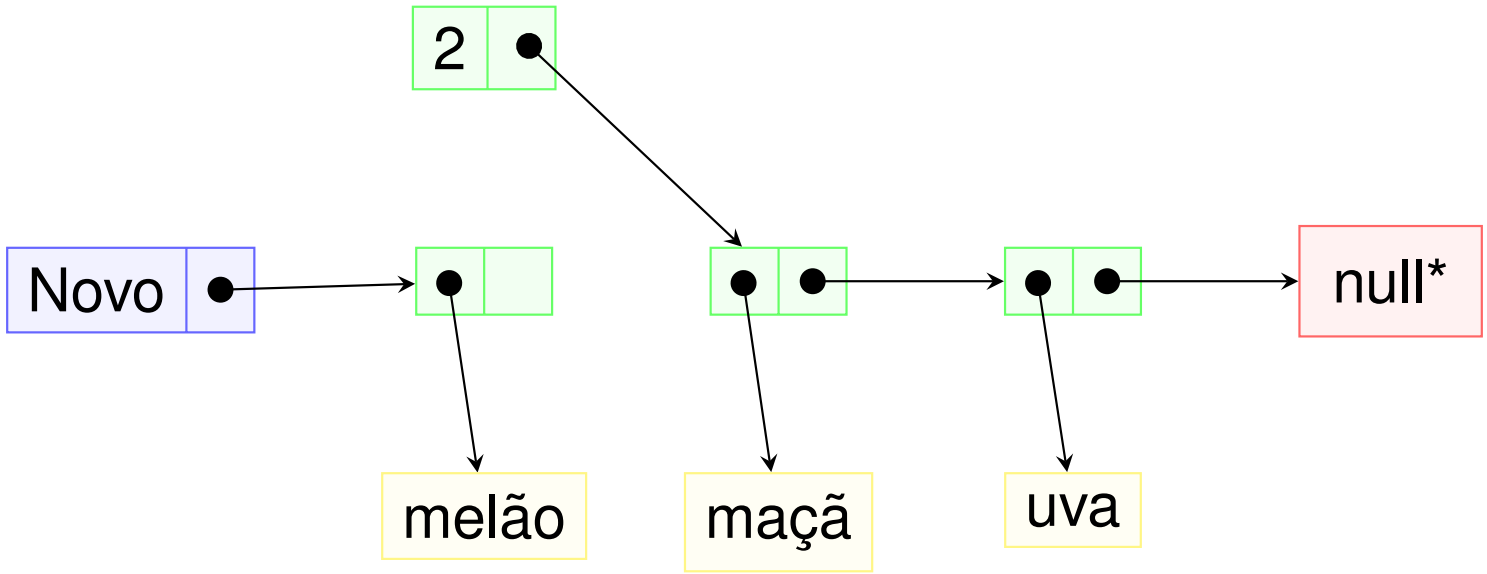
Método `adicionaNoInicio(T *dado)`

- Verifica a possibilidade de alocação de um elemento;
- O próximo deste novo elemento passa ser o primeiro da lista;
- A cabeça de lista passa a apontar para o novo elemento.



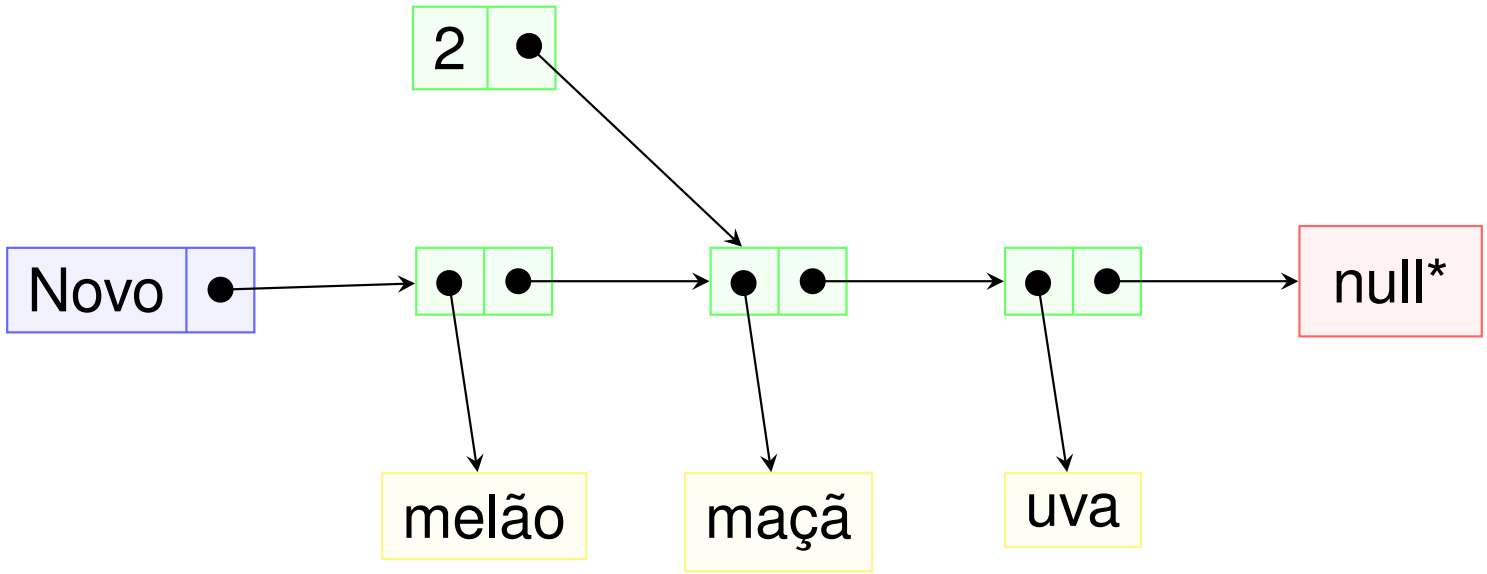
Método `adicionaNoInicio(T *dado)`

- Verifica a possibilidade de alocação de um elemento;
- O próximo deste novo elemento passa ser o primeiro da lista;
- A cabeça de lista passa a apontar para o novo elemento.



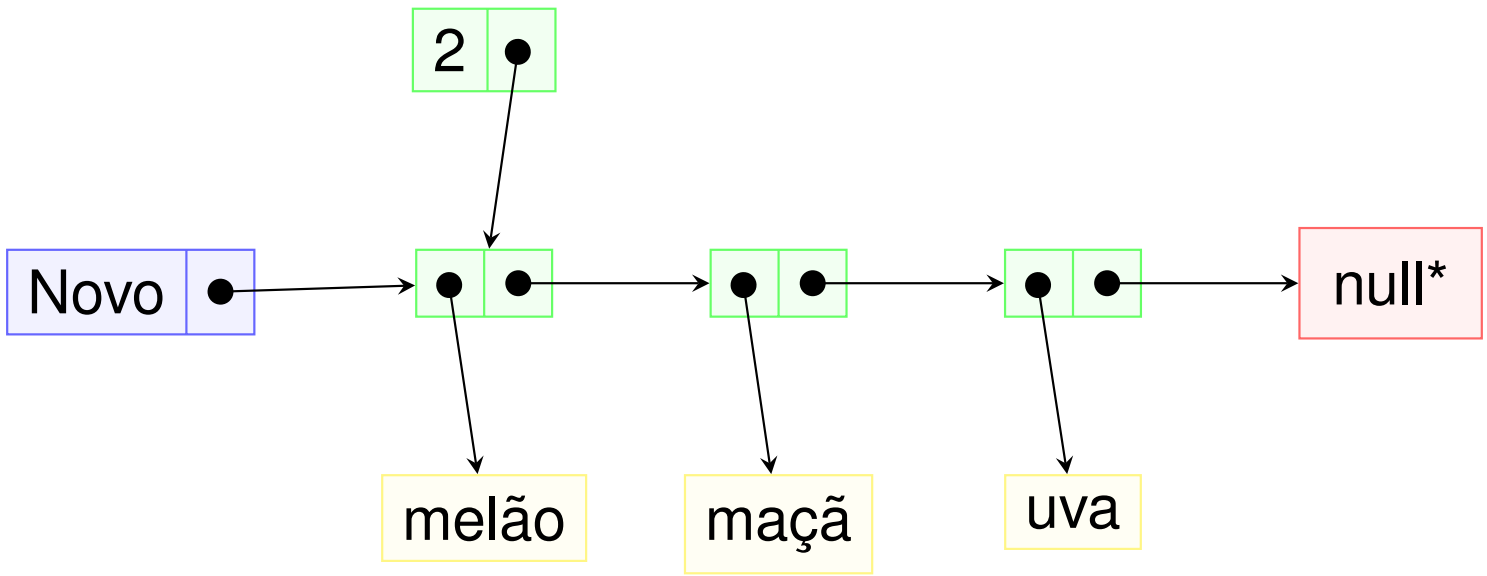
Método `adicionaNoInicio(T *dado)`

- Verifica a possibilidade de alocação de um elemento;
- O próximo deste novo elemento passa ser o primeiro da lista;
- A cabeça de lista passa a apontar para o novo elemento.



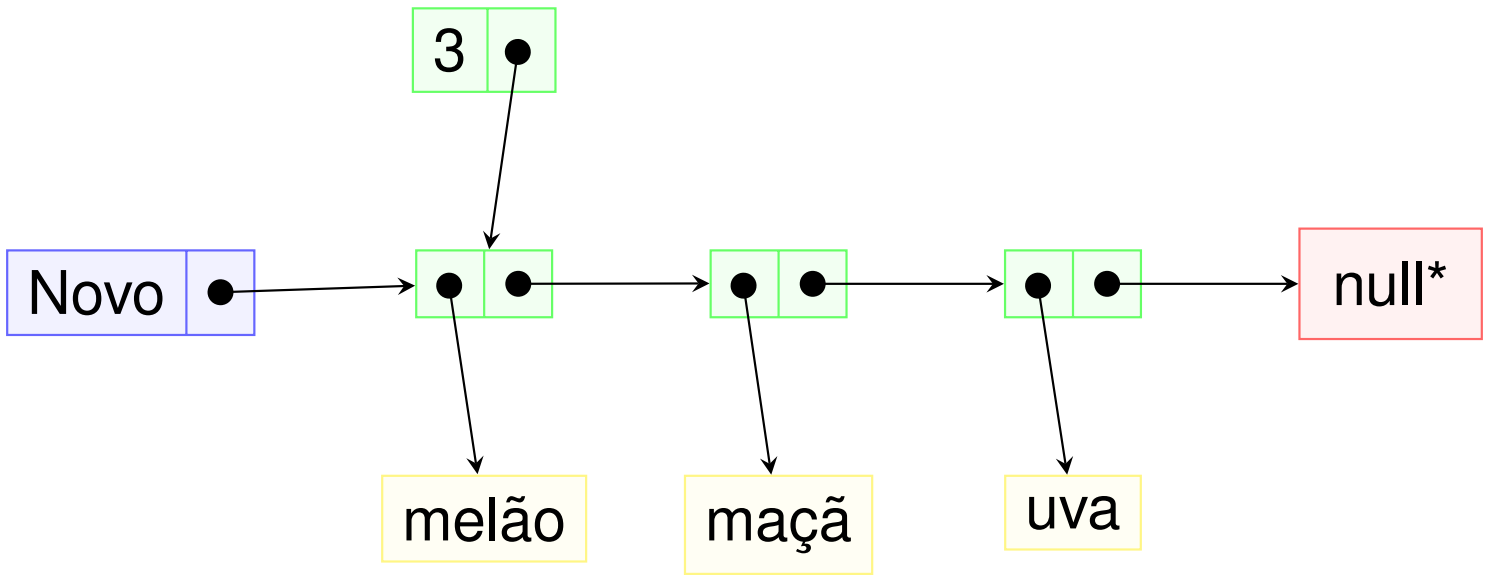
Método `adicionaNoInicio(T *dado)`

- Verifica a possibilidade de alocação de um elemento;
- O próximo deste novo elemento passa ser o primeiro da lista;
- A cabeça de lista passa a apontar para o novo elemento.



Método `adicionaNoInicio(T *dado)`

- Verifica a possibilidade de alocação de um elemento;
- O próximo deste novo elemento passa ser o primeiro da lista;
- A cabeça de lista passa a apontar para o novo elemento.



Método adicionaNoInicio(T *dado)

```
adicionaNoInicio(T *dado)
    Elemento *novo;  // variavel auxiliar
    inicio
        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;  // variavel auxiliar
    inicio
        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;  // variavel auxiliar
    inicio
        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()`

- Testa a existência de elementos;

Método `T *retiraDoInicio()`

- Testa a existência de elementos;
- Decrementa o tamanho;

Método `T *retiraDoInicio()`

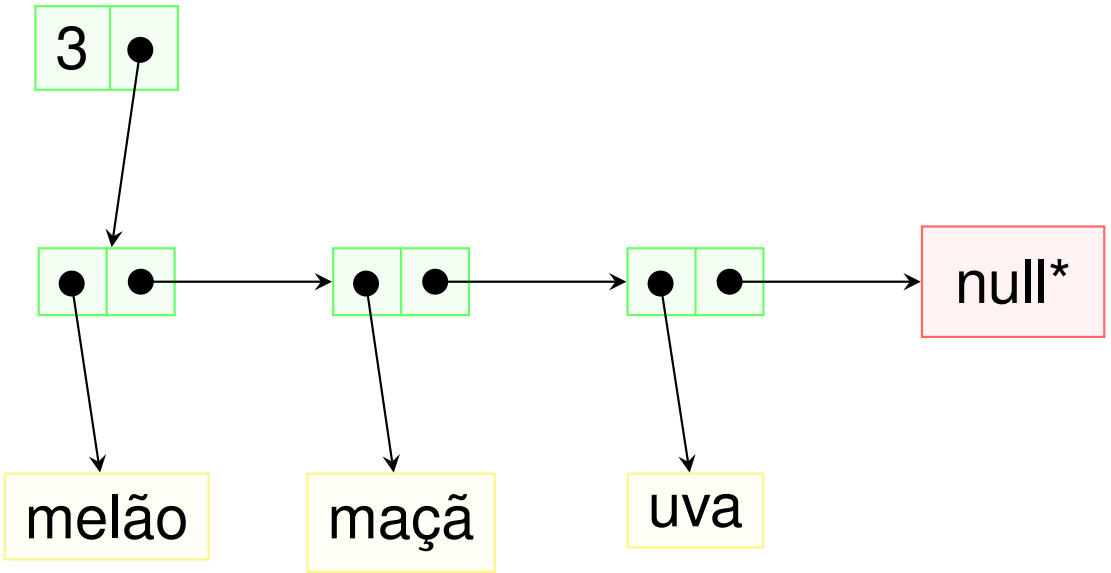
- Testa a existência de elementos;
- Decrementa o tamanho;
- Libera a memória do elemento;

Método `T *retiraDoInicio()`

- Testa a existência de elementos;
- Decrementa o tamanho;
- Libera a memória do elemento;
- Devolve a informação.

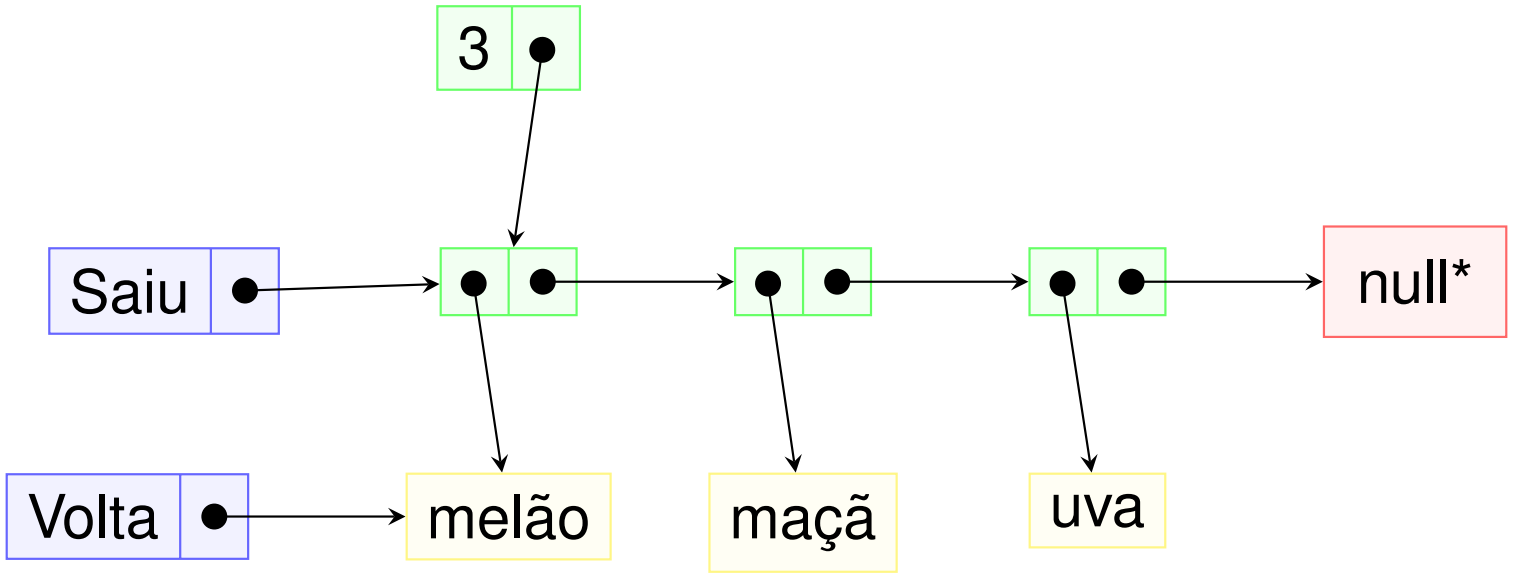
Método `T *retiraDoInicio()`

- Testa a existência de elementos;
- Decrementa o tamanho;
- Libera a memória do elemento;
- Devolve a informação.



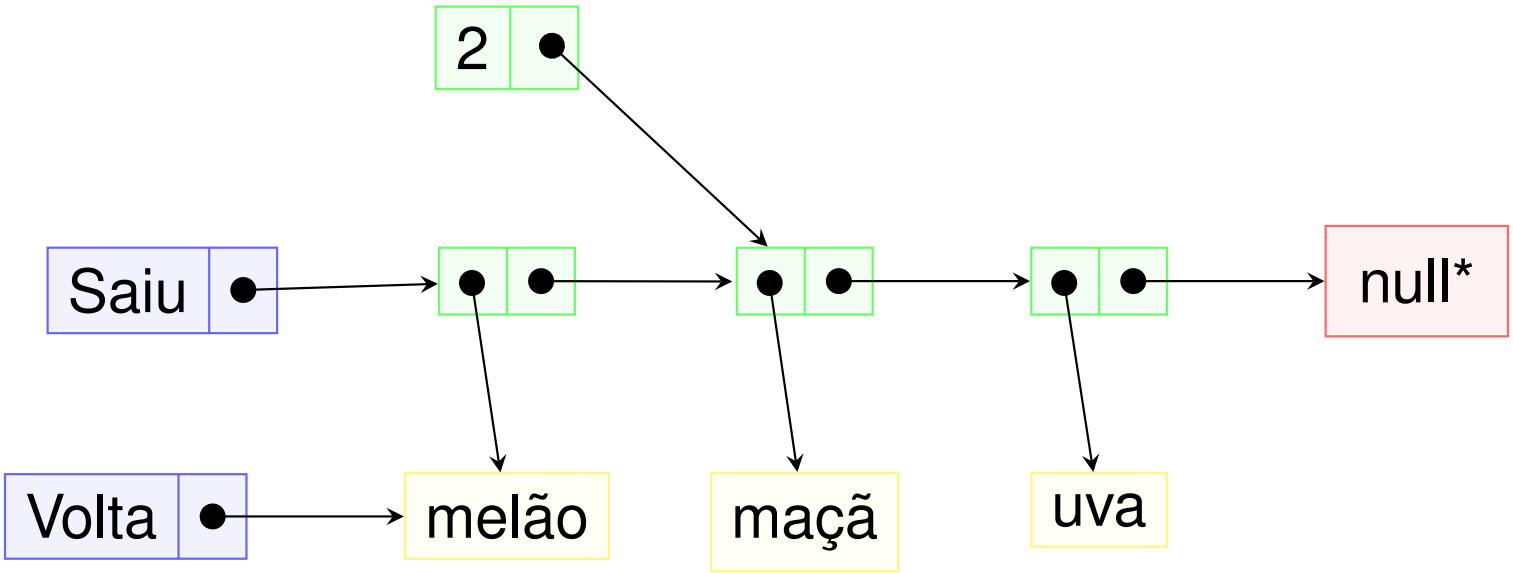
Método `T *retiraDoInicio()`

- Testa a existência de elementos;
- Decrementa o tamanho;
- Libera a memória do elemento;
- Devolve a informação.



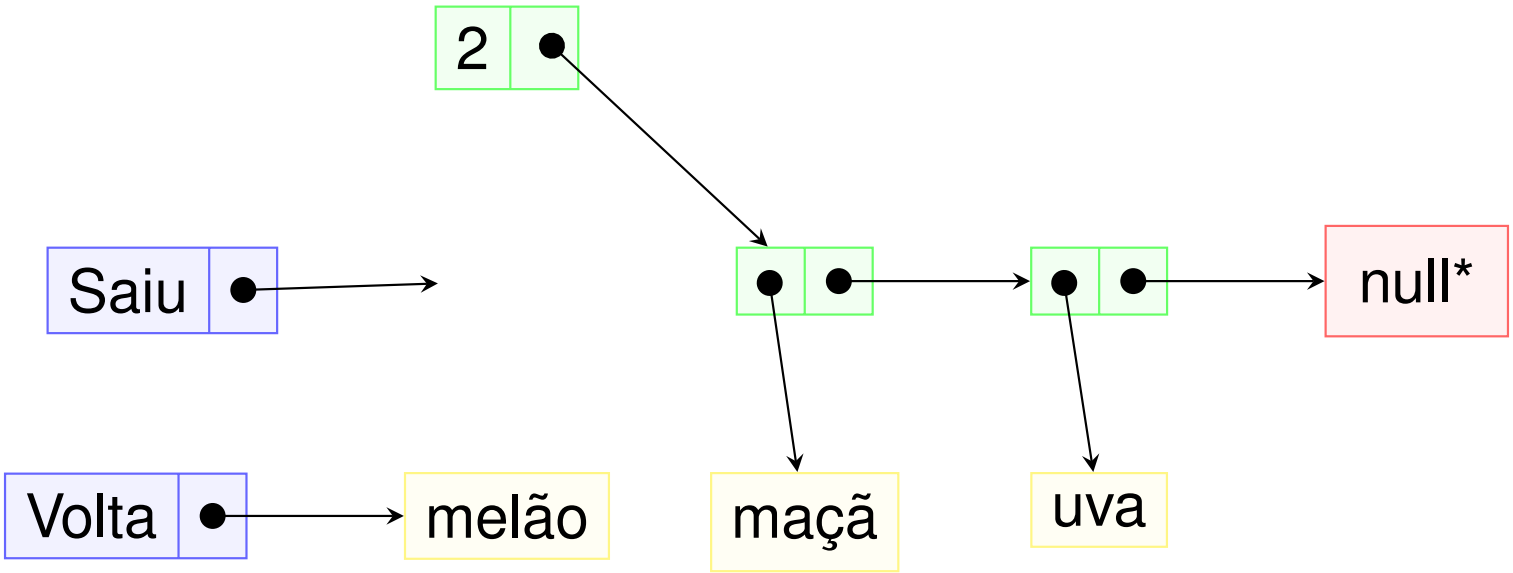
Método `T *retiraDoInicio()`

- Testa a existência de elementos;
- Decrementa o tamanho;
- Libera a memória do elemento;
- Devolve a informação.



Método `T *retiraDoInicio()`

- Testa a existência de elementos;
- Decrementa o tamanho;
- Libera a memória do elemento;
- Devolve a informação.



Método T *retiraDoInicio()

```
T *retiraDoInicio()
    Elemento *saiu;  // variavel auxiliar elemento
    T *volta;  // variavel auxiliar tipo T
    inicio
        SE (listaVazia()) ENTAO
            THROW(ERROLISTAVAZIA);
        SENAO
            saiu ← _dados;
            volta ← saiu->info;
            _dados ← saiu->proximo;
            _tamanho ← _tamanho - 1;
            LIBERE(saiu);
            RETORNE(volta);
        FIM SE
    fim
```

Método T *retiraDoInicio()

```
T *retiraDoInicio()
    Elemento *saiu;  // variavel auxiliar elemento
    T *volta;  // variavel auxiliar tipo T
    inicio
        SE (listaVazia()) ENTAO
            THROW(ERROLISTAVAZIA);
        SENAO
            saiu ← _dados;
            volta ← saiu->info;
            _dados ← saiu->proximo;
            _tamanho ← _tamanho - 1;
            LIBERE(saiu);
            RETORNE(volta);
        FIM SE
    fim
```

Método T *retiraDoInicio()

```
T *retiraDoInicio()
    Elemento *saiu;  // variavel auxiliar elemento
    T *volta;  // variavel auxiliar tipo T
    inicio
        SE (listaVazia()) ENTAO
            THROW(ERROLISTAVAZIA);
        SENAO
            saiu ← _dados;
            volta ← saiu->info;
            _dados ← saiu->proximo;
            _tamanho ← _tamanho - 1;
            LIBERE(saiu);
            RETORNE(volta);
        FIM SE
    fim
```

Método T *retiraDoInicio()

```
T *retiraDoInicio()
    Elemento *saiu;  // variavel auxiliar elemento
    T *volta;  // variavel auxiliar tipo T
    inicio
        SE (listaVazia()) ENTAO
            THROW(ERROLISTAVAZIA);
        SENAO
            saiu ← _dados;
            volta ← saiu->info;
            _dados ← saiu->proximo;
            _tamanho ← _tamanho - 1;
            LIBERE(saiu);
            RETORNE(volta);
        FIM SE
    fim
```

Método eliminaDoInicio()

```
eliminaDoInicio()
    Elemento *saiu;  // variavel auxiliar elemento
    inicio
        SE (listaVazia()) ENTAO
            THROW(ERROLISTAVAZIA);
        SENAO
            saiu ← _dados;
            volta ← saiu->info;
            _dados ← saiu->proximo;
            _tamanho ← _tamanho - 1;
            LIBERE(saiu->info);   // desaloca a info!
            LIBERE(saiu);
        FIM SE
    fim
```

Algoritmo `eliminaDoInicio()`

- Cuidados com o uso do `LIBERE(saiu->info)`:
  - Se o `T` for por sua vez um conjunto estruturado de dados com referências internas por meio de ponteiros (outra lista, por exemplo), a chamada à função `LIBERE(saiu->info)` só desalocará o primeiro nível da estrutura (aquele apontado diretamente);
  - Tudo o que for referenciado por meio de ponteiros em `info` permanecerá em algum lugar da memória, provavelmente inatingível (*garbage*);
  - Para evitar isto, pode-se criar uma função `destroi(info)` para o `T` que será chamada no lugar de `LIBERE`.

Importância do destrutor

- O destrutor é responsável pela desalocação do objeto ao sair de seu escopo;
- No mínimo, deve haver desalocação da 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)`

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

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

- Testa se a posição existe e se é possível alocar;
- Percorre até a posição;

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

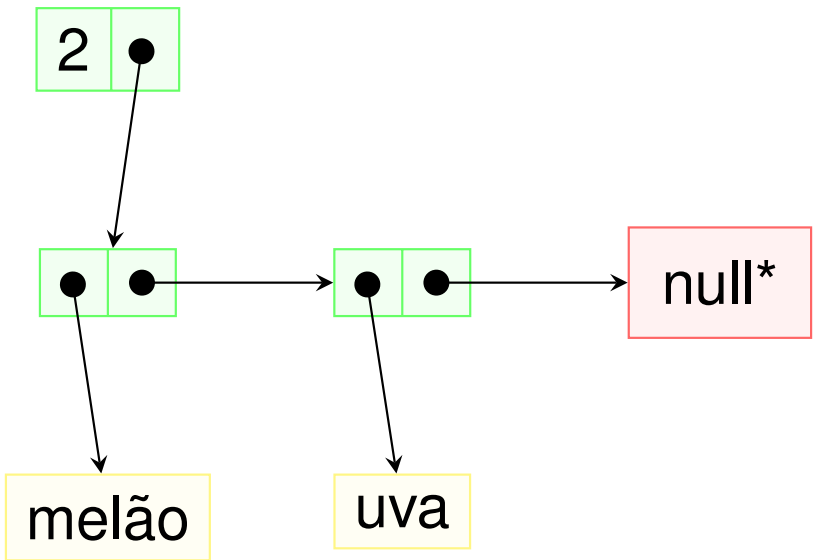
- Testa se a posição existe e se é possível alocar;
- Percorre até a posição;
- Adiciona o novo dado na posição;

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

- Testa se a posição existe e se é possível alocar;
- Percorre até a posição;
- Adiciona o novo dado na posição;
- Incrementa o tamanho.

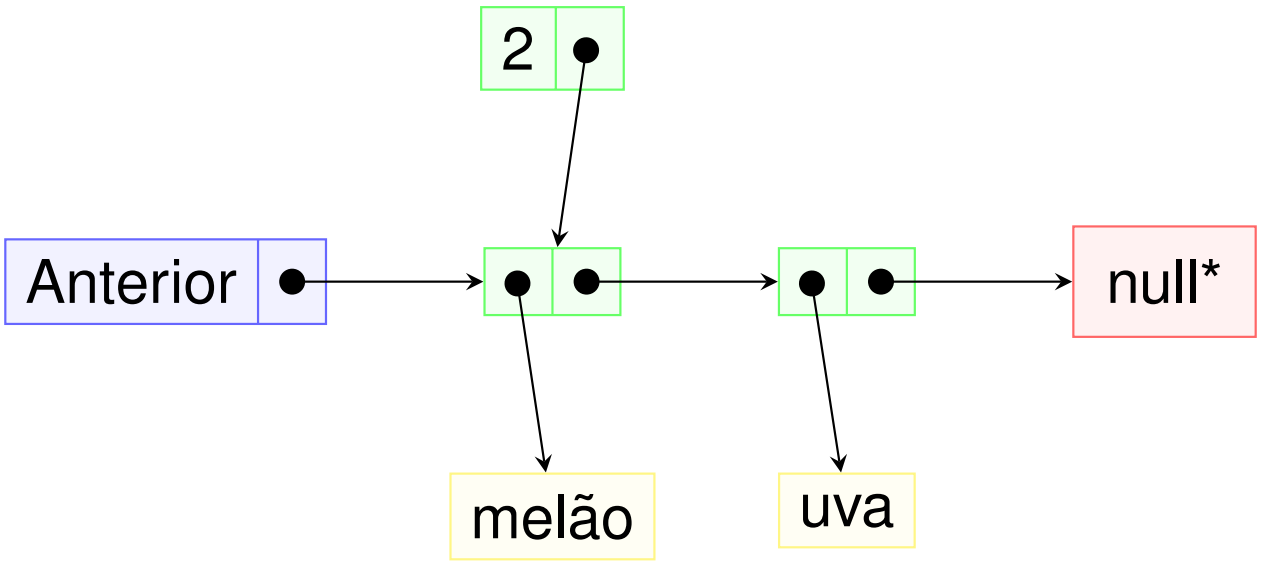
Método `adicionaNaPosicao(T *dado, int posicao)`

- Testa se a posição existe e se é possível alocar;
- Percorre até a posição;
- Adiciona o novo dado na posição;
- Incrementa o tamanho.



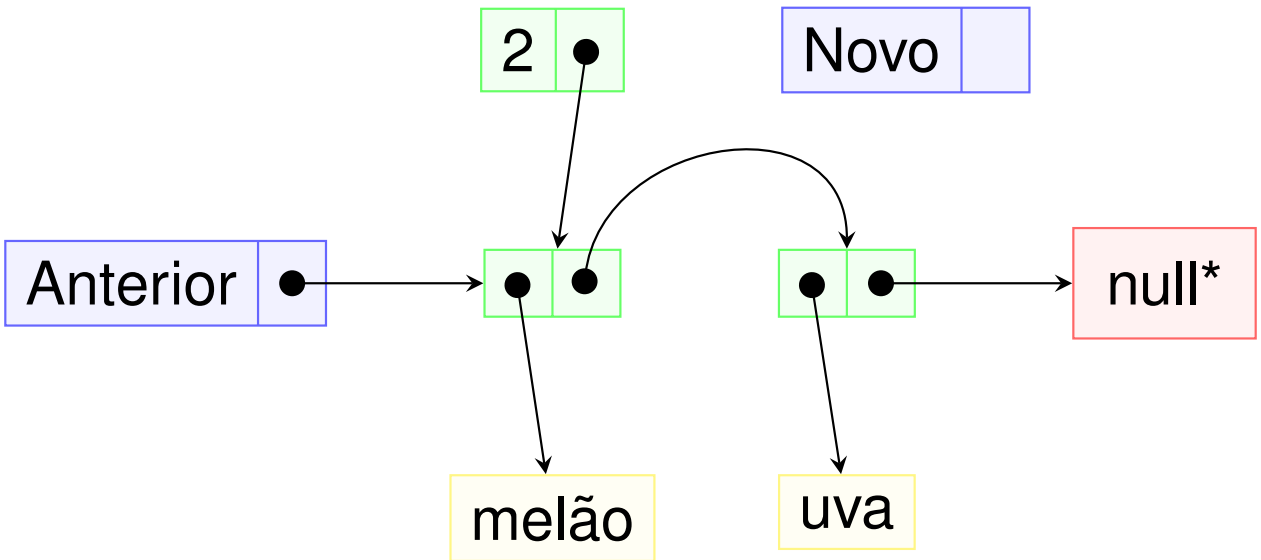
Método `adicionaNaPosicao(T *dado, int posicao)`

- Testa se a posição existe e se é possível alocar;
- Percorre até a posição;
- Adiciona o novo dado na posição;
- Incrementa o tamanho.



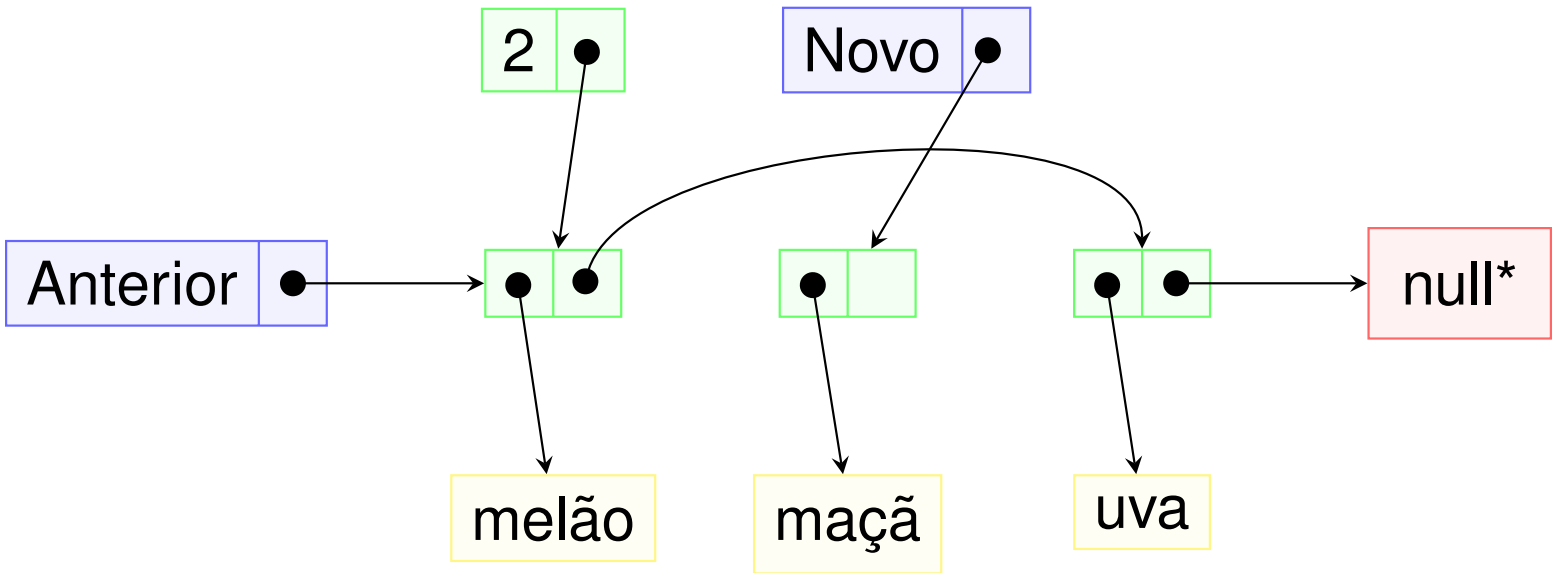
Método `adicionaNaPosicao(T *dado, int posicao)`

- Testa se a posição existe e se é possível alocar;
- Percorre até a posição;
- Adiciona o novo dado na posição;
- Incrementa o tamanho.



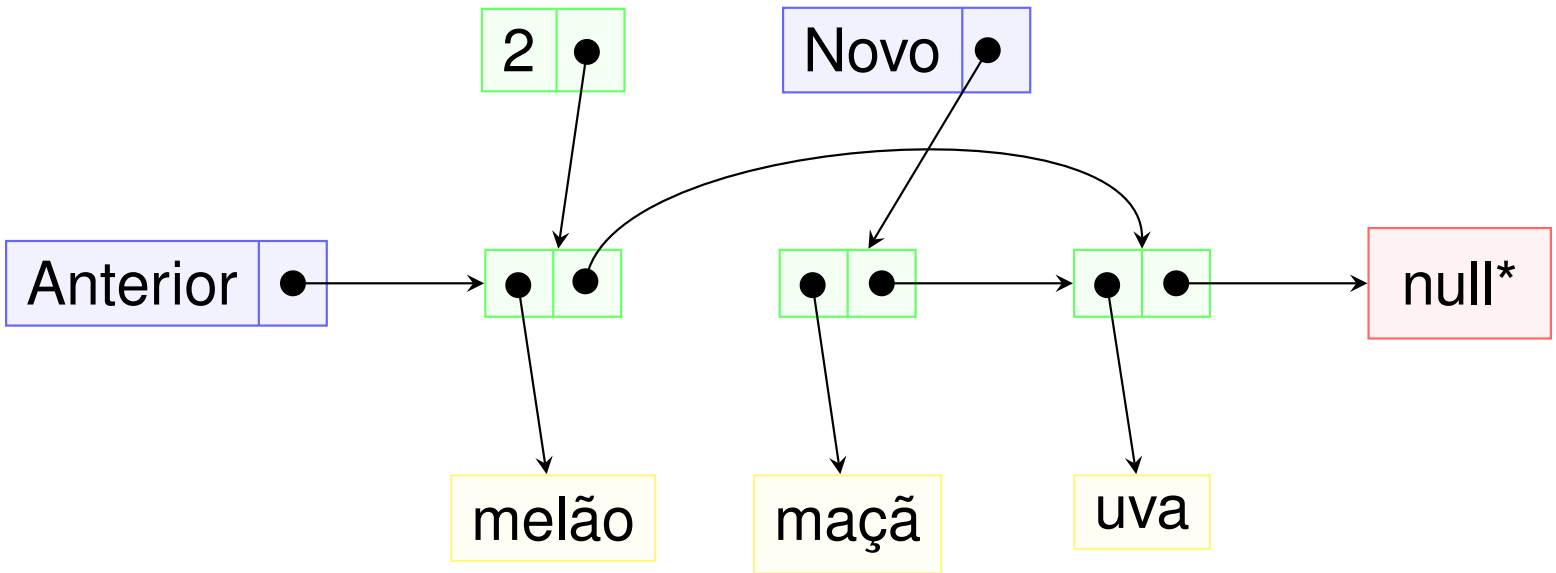
Método `adicionaNaPosicao(T *dado, int posicao)`

- Testa se a posição existe e se é possível alocar;
- Percorre até a posição;
- Adiciona o novo dado na posição;
- Incrementa o tamanho.



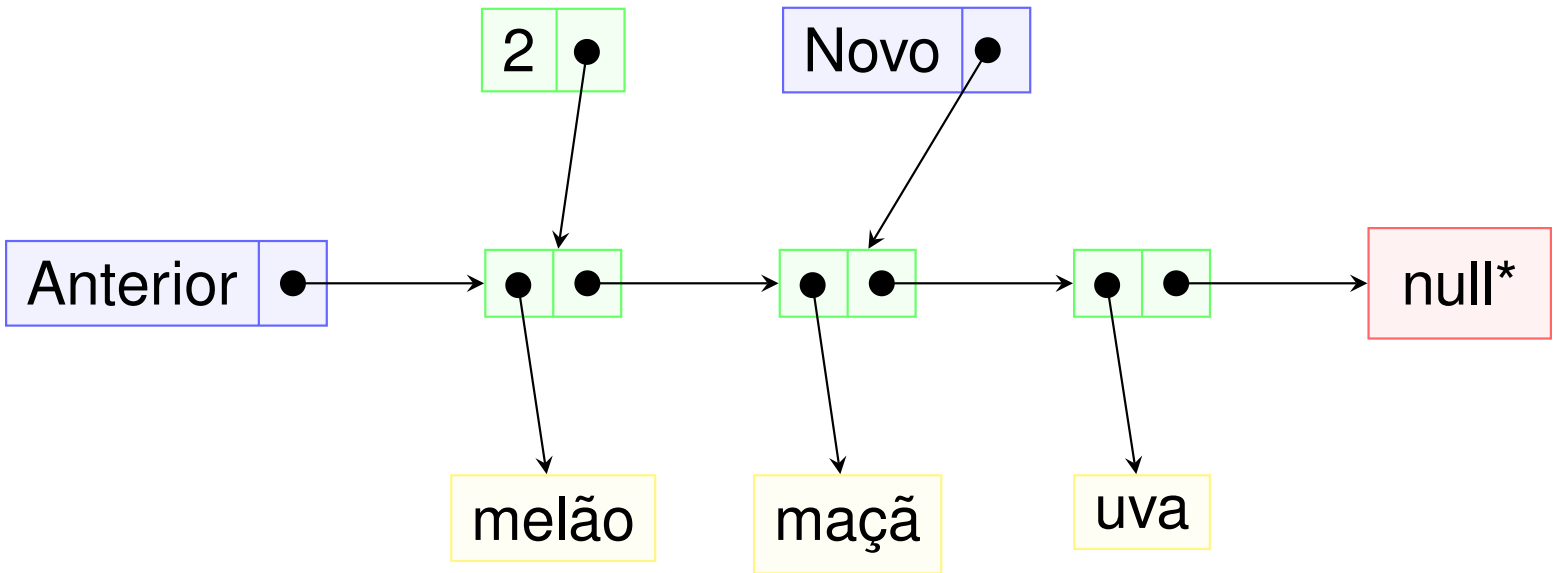
Método `adicionaNaPosicao(T *dado, int posicao)`

- Testa se a posição existe e se é possível alocar;
- Percorre até a posição;
- Adiciona o novo dado na posição;
- Incrementa o tamanho.



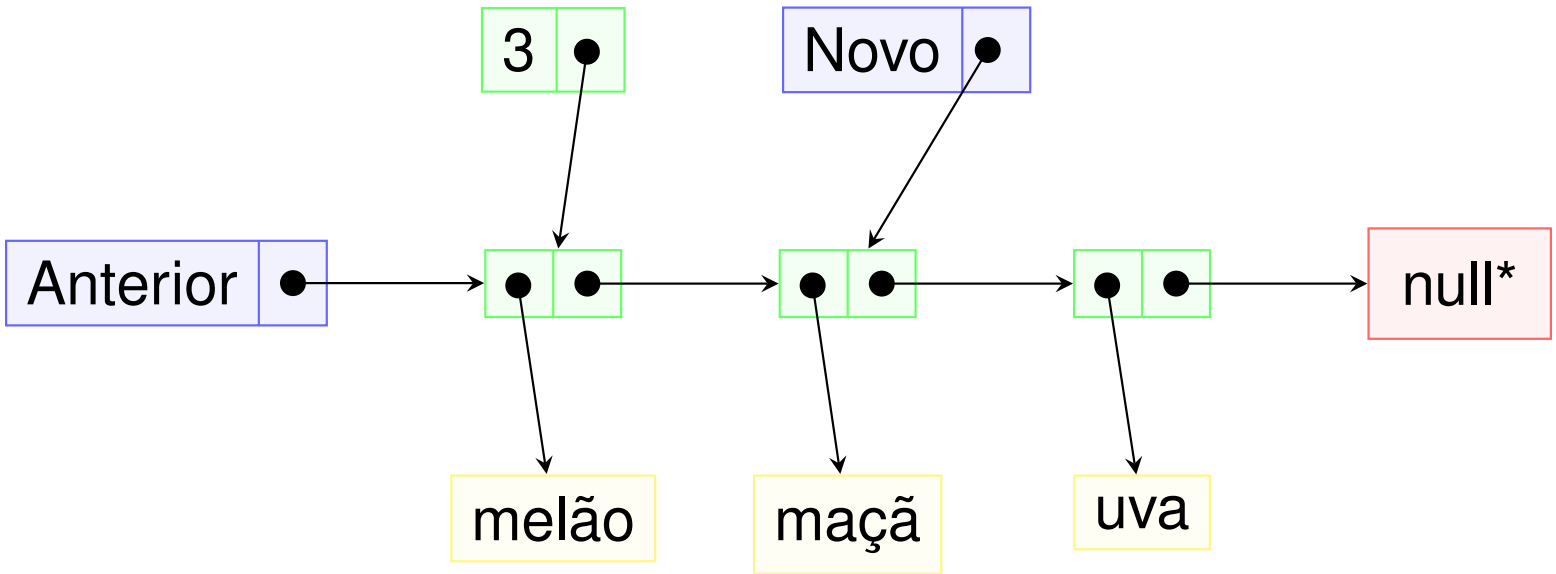
Método `adicionaNaPosicao(T *dado, int posicao)`

- Testa se a posição existe e se é possível alocar;
- Percorre até a posição;
- Adiciona o novo dado na posição;
- Incrementa o tamanho.



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

- Testa se a posição existe e se é possível alocar;
- Percorre até a posição;
- Adiciona o novo dado na posição;
- Incrementa o tamanho.



Método adicionaNaPosicao(T *dado, int posicao)

```
adicionaNaPosicao(T *dado, int posicao)
Elemento *novo, *anterior;  // auxiliares
inicio
 SE (posicao > _tamanho) ENTAO THROW(ERROPOSICAO);
 SENAO
   SE (posicao = 0) ENTAO
       RETORNE(adicionaNoInicio(info));
   SENAO
       novo ← aloque(Elemento);
       SE (novo = NULO) ENTAO THROW(ERROLISTACHEIA);
       SENAO
           anterior ← _dados;
           REPITA (posicao - 1) 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) ENTAO THROW(ERROPOSICAO);
 SENAO
   SE (posicao = 0) ENTAO
       RETORNE(adicionaNoInicio(info));
   SENAO
       novo ← aloque(Elemento);
       SE (novo = NULO) ENTAO THROW(ERROLISTACHEIA);
       SENAO
           anterior ← _dados;
           REPITA (posicao - 1) 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) ENTAO THROW(ERROPOSICAO);
 SENAO
   SE (posicao = 0) ENTAO
       RETORNE(adicionaNoInicio(info));
   SENAO
       novo ← aloque(Elemento);
       SE (novo = NULO) ENTAO THROW(ERROLISTACHEIA);
       SENAO
           anterior ← _dados;
           REPITA (posicao - 1) 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) ENTAO THROW(ERROPOSICAO);
 SENAO
   SE (posicao = 0) ENTAO
       RETORNE(adicionaNoInicio(info));
   SENAO
       novo ← aloque(Elemento);
       SE (novo = NULO) ENTAO THROW(ERROLISTACHEIA);
       SENAO
           anterior ← _dados;
           REPITA (posicao - 1) 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) ENTAO THROW(ERROPOSICAO);
 SENAO
   SE (posicao = 0) ENTAO
       RETORNE(adicionaNoInicio(info));
   SENAO
       novo ← aloque(Elemento);
       SE (novo = NULO) ENTAO THROW(ERROLISTACHEIA);
       SENAO
           anterior ← _dados;
           REPITA (posicao - 1) 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) ENTAO THROW(ERROPOSICAO);
 SENAO
   SE (posicao = 0) ENTAO
       RETORNE(adicionaNoInicio(info));
   SENAO
       novo ← aloque(Elemento);
       SE (novo = NULO) ENTAO THROW(ERROLISTACHEIA);
       SENAO
           anterior ← _dados;
           REPITA (posicao - 1) 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)`

- Testa se a posição existe;

Método `T *retiraDaPosicao(int posicao)`

- Testa se a posição existe;
- Percorre até a posição;

Método `T *retiraDaPosicao(int posicao)`

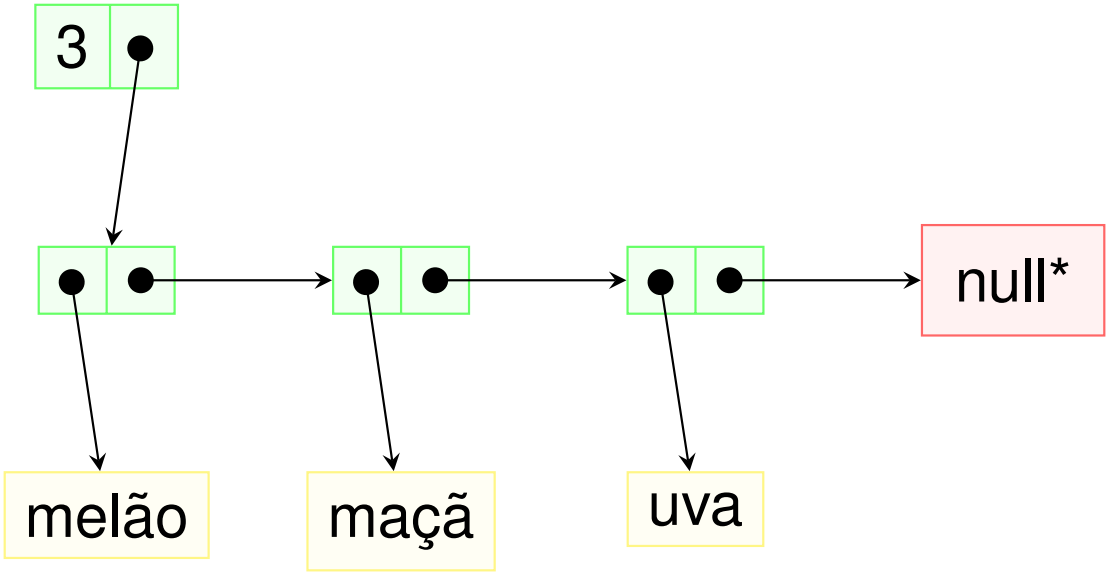
- Testa se a posição existe;
- Percorre até a posição;
- Retira o dado da posição;

Método `T *retiraDaPosicao(int posicao)`

- Testa se a posição existe;
- Percorre até a posição;
- Retira o dado da posição;
- Decrementa o tamanho.

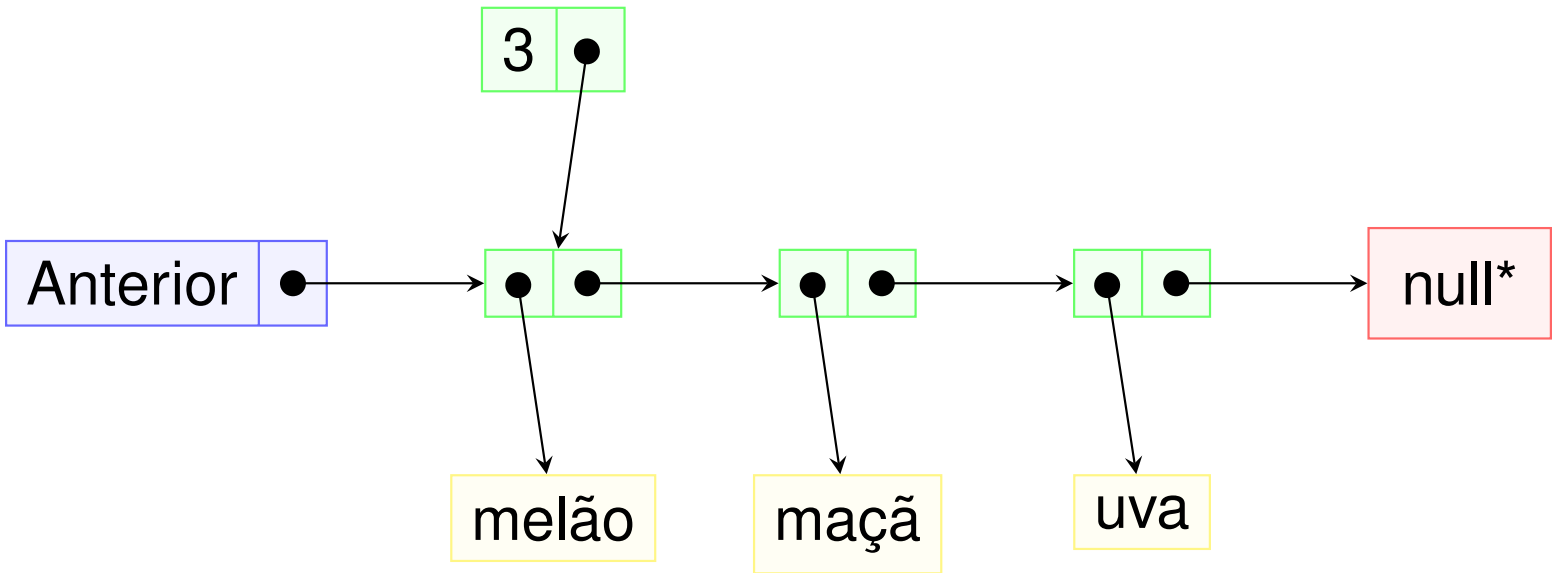
Método `T *retiraDaPosicao(int posicao)`

- Testa se a posição existe;
- Percorre até a posição;
- Retira o dado da posição;
- Decrementa o tamanho.



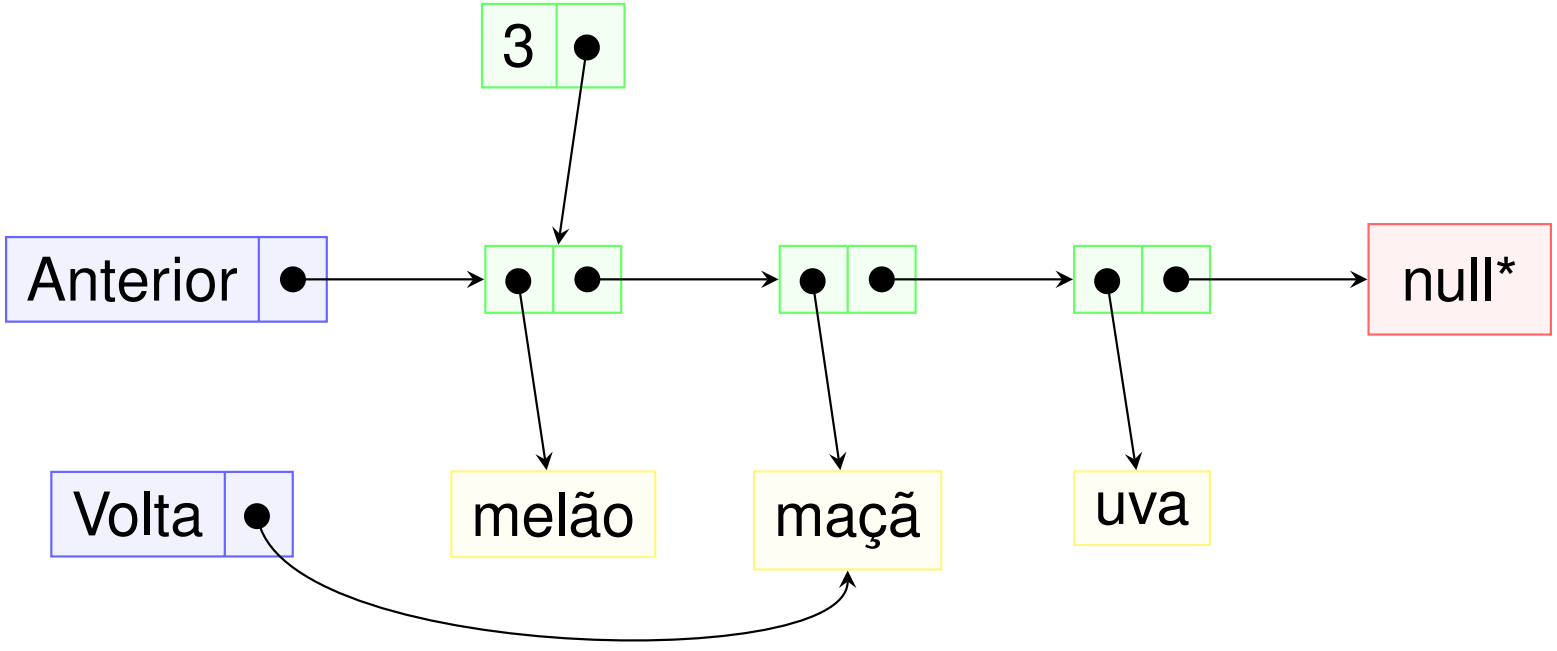
Método `T *retiraDaPosicao(int posicao)`

- Testa se a posição existe;
- Percorre até a posição;
- Retira o dado da posição;
- Decrementa o tamanho.



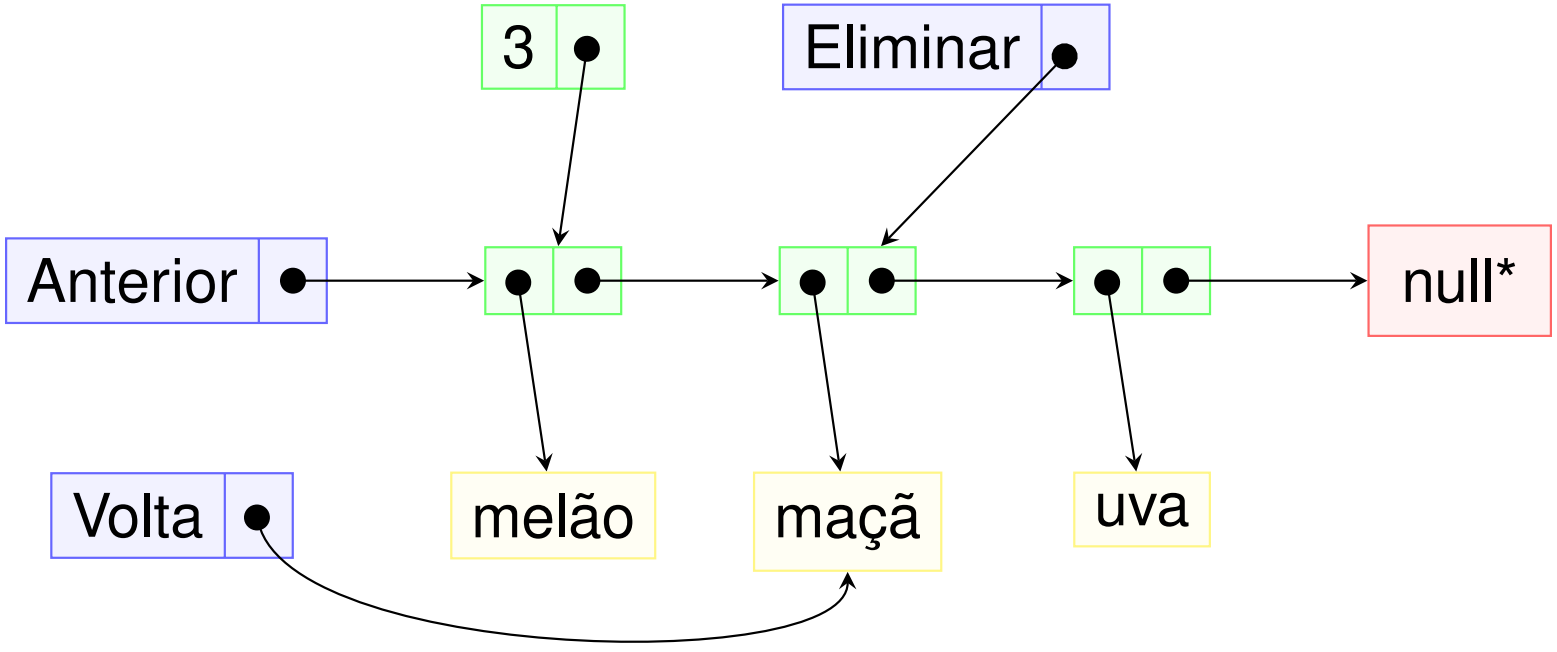
Método `T *retiraDaPosicao(int posicao)`

- Testa se a posição existe;
- Percorre até a posição;
- Retira o dado da posição;
- Decrementa o tamanho.



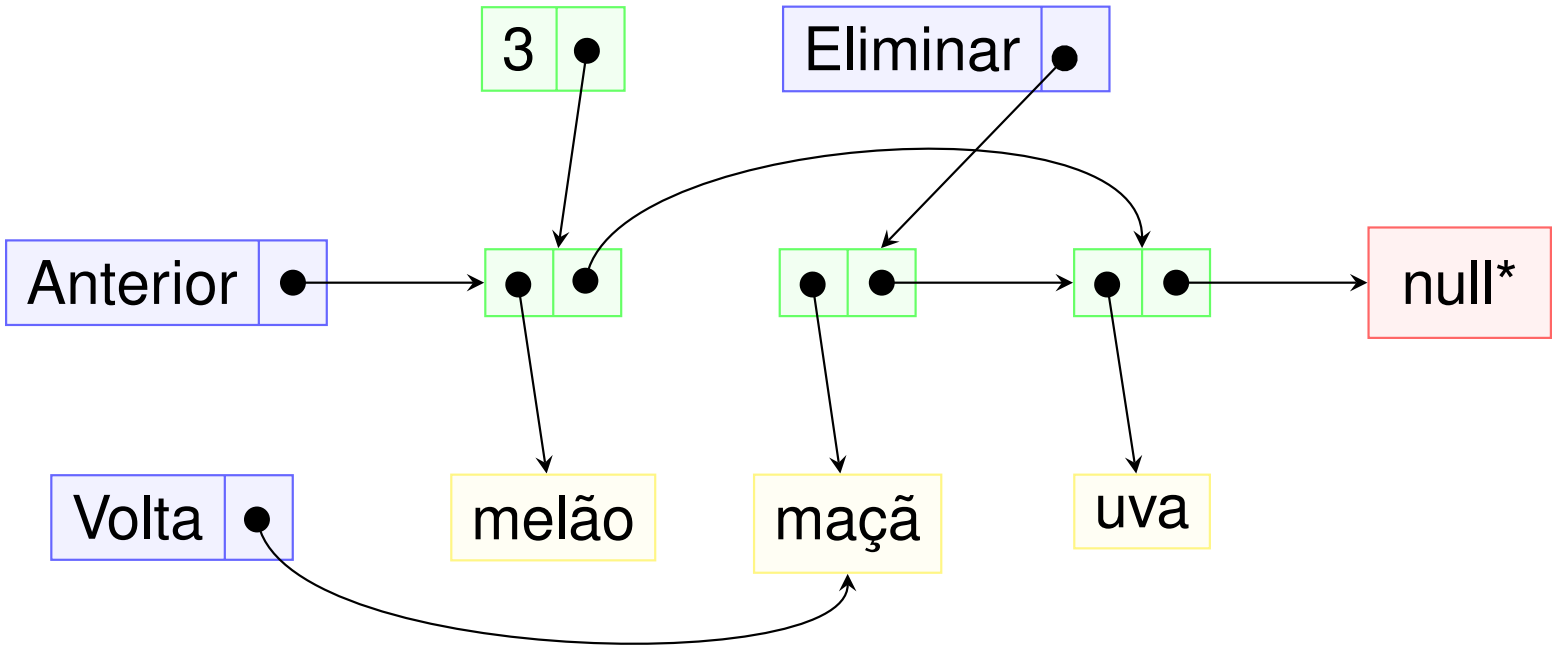
Método `T *retiraDaPosicao(int posicao)`

- Testa se a posição existe;
- Percorre até a posição;
- Retira o dado da posição;
- Decrementa o tamanho.



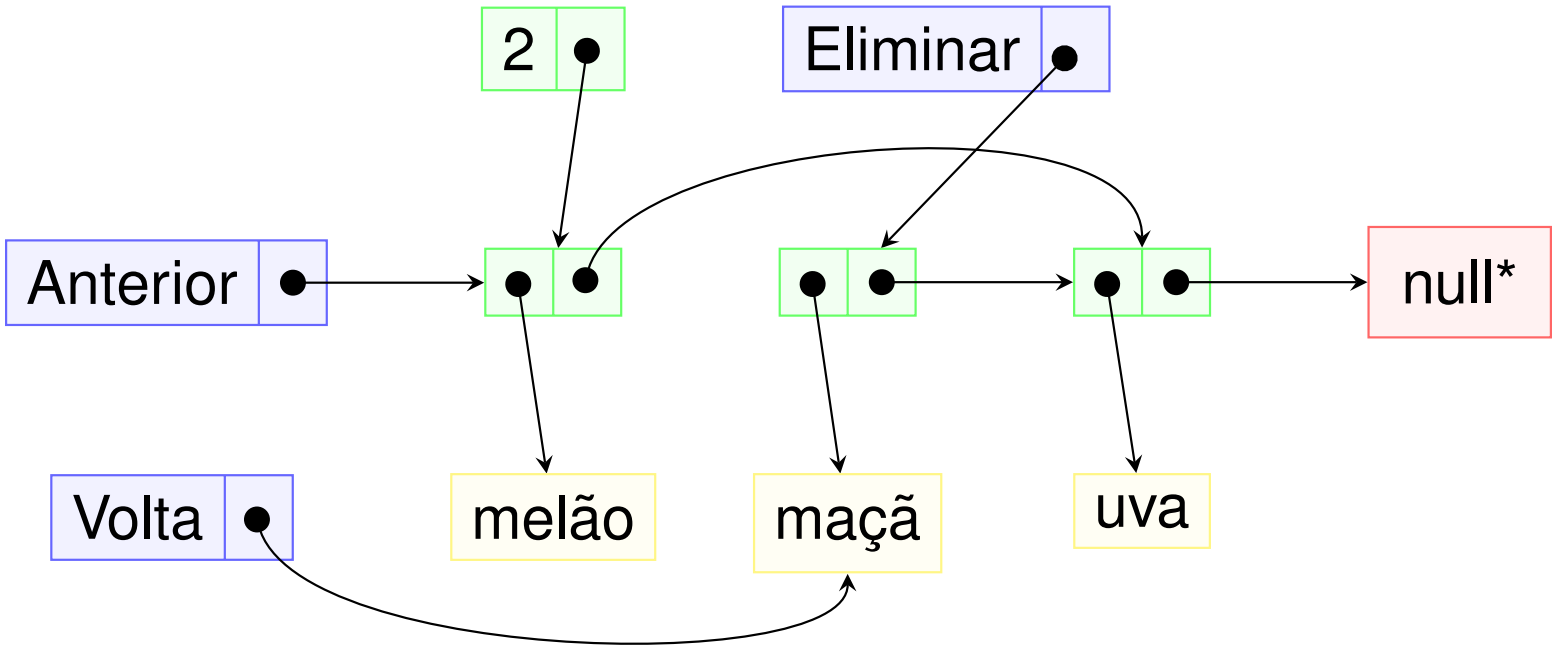
Método `T *retiraDaPosicao(int posicao)`

- Testa se a posição existe;
- Percorre até a posição;
- Retira o dado da posição;
- Decrementa o tamanho.



Método `T *retiraDaPosicao(int posicao)`

- Testa se a posição existe;
- Percorre até a posição;
- Retira o dado da posição;
- Decrementa o tamanho.



Método T *retiraDaPosicao(int posicao)

```
T *retiraDaPosicao(int posicao)
 Elemento *anterior, *eliminar;  // auxiliares
 T *volta;  // variavel tipo T
 inicio
   SE (posicao ≥ _tamanho) ENTAO THROW(ERROPOSICAO);
   SENAO
     SE (posicao = 0) ENTAO RETORNE(retiraDoInicio());
     SENAO
       anterior ← _dados;
       REPITA (posicao - 1) 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;  // auxiliares
 T *volta;  // variavel tipo T
 inicio
   SE (posicao ≥ _tamanho) ENTAO THROW(ERROPOSICAO);
   SENAO
     SE (posicao = 0) ENTAO RETORNE(retiraDoInicio());
     SENAO
       anterior ← _dados;
       REPITA (posicao - 1) 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;  // auxiliares
 T *volta;  // variavel tipo T
 inicio
   SE (posicao ≥ _tamanho) ENTAO THROW(ERROPOSICAO);
     SENAO
       SE (posicao = 0) ENTAO RETORNE(retiraDoInicio());
     SENAO
       anterior ← _dados;
       REPITA (posicao - 1) 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;  // auxiliares
 T *volta;  // variavel tipo T
 inicio
   SE (posicao ≥ _tamanho) ENTAO THROW(ERROPOSICAO);
   SENAO
     SE (posicao = 0) ENTAO RETORNE(retiraDoInicio());
     SENAO
       anterior ← _dados;
       REPITA (posicao - 1) 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 *retira   DaPosicao(int posicao)
 Elemento *anterior, *eliminar;  // auxiliares
 T *volta;  // variavel tipo T
 inicio
   SE (posicao ≥ _tamanho) ENTAO THROW(ERROPOSICAO);
   SENAO
     SE (posicao = 0) ENTAO RETORNE(retiraDoInicio());
     SENAO
       anterior ← _dados;
       REPITA (posicao - 1) 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 `adicionaNaPosicao(...)`

Dica

Pode-se implementar um versão polimórfica de `adicionaNaPosicao(T *dado, int posicao)`, de modo que o elemento seja passado como parâmetro: `adicionaNaPosicao(Elemento* elemento, int posicao)`

Método `adicionaEmOrdem(T *dado)`

- É necessário um método para comparar os dados (`operator::>`);
- Procura a posição de inserção comparando os dados;
- Executa `adicionaNaPosicao(T *dado, int posicao)`.

Método adicionaEmOrdem(T *dado)

```
adicionaEmOrdem(T *dado)
    Elemento *atual;  // variavel para caminhar
    int posicao;  // posicao de insercao
    inicio
        atual ← _dados;
        posicao ← 0;
        ENQUANTO (atual ≠ NULO E
                  atual->_info < dado) FACA
            // encontrar posicao de insercao
            atual ← atual->_proximo;
            posicao ← posicao + 1;
        FIM ENQUANTO
        RETORNE(adicionaNaPosicao(dado, posicao));
    fim
```

Método `limpaLista()`

```
limpaLista()
 Elemento *atual, *anterior;  // auxiliares
 inicio
  SE (listaVazia()) ENTAO THROW(ERROLISTAVAZIA);
  SENAO
   atual ← _dados;
   ENQUANTO (atual ≠ NULO) FACA
    // eliminar ate o fim
    anterior ← atual;
    // deslocar para o proximo mesmo que seja nulo
    atual ← atual->_proximo;
    // desalocar primeiro a 'info'
    LIBERE(anterior->_info);
    // desalocar o elemento visitado
    LIBERE(anterior);
   FIM ENQUANTO
  FIM SE
  _dados ← NULO;
  _tamanho ← 0;
 fim
```

Método `limpaLista()`

Dica

Pode-se implementar o `limpaLista()` como um sequência de `eliminaDoInicio()` (também desaloca o `info`), até que lista se torne vazia.

Algoritmos restantes como exercício

- `int posicao(dado);`
- `Elemento* posicao(dado);`
- `bool contem(dado);`

Trabalho sobre 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, 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?


ine
Informática
Estatística
CTC • UFSC


ARS ET SCIENTIA
UNIVERSIDADE FEDERAL
DE SANTA CATARINA
UFSC