

# Estruturas de Dados

## Fila Encadeada, Pilha Encadeada e Lista Duplamente Encadeada

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

2016.2



UNIVERSIDADE FEDERAL  
DE SANTA CATARINA

# Extensões do conceito de Lista Encadeada

- A idéia da Lista Encadeada vista até agora é o modelo mais geral e simples;
- Pode ser especializada e extendida das mais variadas formas;
- Especializada:
  - Pilhas encadeadas;
  - Filas;
- Extendida:
  - Listas Duplamente Encadeadas;
  - Listas Circulares Simples e Duplas.

# Conceito de Pilhas

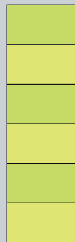
## Pilha

É uma estrutura de dados cujo funcionamento é inspirado no conceito “natural” de empilhar, onde o primeiro a entrar é o último a sair (LIFO).

# Conceito de Pilhas

## Pilha

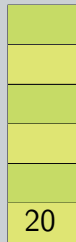
É uma estrutura de dados cujo funcionamento é inspirado no conceito “natural” de empilhar, onde o primeiro a entrar é o último a sair (LIFO).



# Conceito de Pilhas

## Pilha

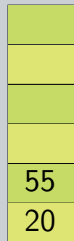
É uma estrutura de dados cujo funcionamento é inspirado no conceito “natural” de empilhar, onde o primeiro a entrar é o último a sair (LIFO).



# Conceito de Pilhas

## Pilha

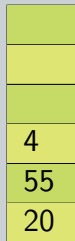
É uma estrutura de dados cujo funcionamento é inspirado no conceito “natural” de empilhar, onde o primeiro a entrar é o último a sair (LIFO).



# Conceito de Pilhas

## Pilha

É uma estrutura de dados cujo funcionamento é inspirado no conceito “natural” de empilhar, onde o primeiro a entrar é o último a sair (LIFO).



# Conceito de Pilhas

## Pilha

É uma estrutura de dados cujo funcionamento é inspirado no conceito “natural” de empilhar, onde o primeiro a entrar é o último a sair (LIFO).

12
4
55
20



# Conceito de Pilhas

## Pilha

É uma estrutura de dados cujo funcionamento é inspirado no conceito “natural” de empilhar, onde o primeiro a entrar é o último a sair (LIFO).

89
12
4
55
20

# Conceito de Pilhas

## Pilha

É uma estrutura de dados cujo funcionamento é inspirado no conceito “natural” de empilhar, onde o primeiro a entrar é o último a sair (LIFO).

24
89
12
4
55
20

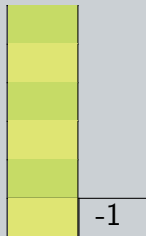
# Pilhas Usando Vetores

- Vetores possuem um espaço limitado para armazenar dados;
- Precisamos definir um espaço grande o suficiente para a nossa pilha;
- Precisamos de um indicador de qual elemento do vetor é o atual topo da pilha.



# Pilhas Usando Vetores

- Vetores possuem um espaço limitado para armazenar dados;
- Precisamos definir um espaço grande o suficiente para a nossa pilha;
- Precisamos de um indicador de qual elemento do vetor é o atual topo da pilha.
- **Pilha Vazia!**



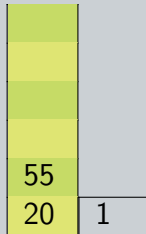
# Pilhas Usando Vetores

- Vetores possuem um espaço limitado para armazenar dados;
- Precisamos definir um espaço grande o suficiente para a nossa pilha;
- Precisamos de um indicador de qual elemento do vetor é o atual topo da pilha.



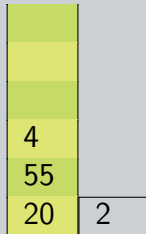
# Pilhas Usando Vetores

- Vetores possuem um espaço limitado para armazenar dados;
- Precisamos definir um espaço grande o suficiente para a nossa pilha;
- Precisamos de um indicador de qual elemento do vetor é o atual topo da pilha.



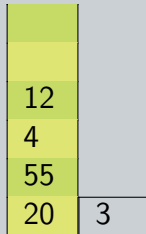
# Pilhas Usando Vetores

- Vetores possuem um espaço limitado para armazenar dados;
- Precisamos definir um espaço grande o suficiente para a nossa pilha;
- Precisamos de um indicador de qual elemento do vetor é o atual topo da pilha.



# Pilhas Usando Vetores

- Vetores possuem um espaço limitado para armazenar dados;
- Precisamos definir um espaço grande o suficiente para a nossa pilha;
- Precisamos de um indicador de qual elemento do vetor é o atual topo da pilha.





# Pilhas Usando Vetores

- Vetores possuem um espaço limitado para armazenar dados;
- Precisamos definir um espaço grande o suficiente para a nossa pilha;
- Precisamos de um indicador de qual elemento do vetor é o atual topo da pilha.

89	
12	
4	
55	
20	4

# Pilhas Usando Vetores

- Vetores possuem um espaço limitado para armazenar dados;
- Precisamos definir um espaço grande o suficiente para a nossa pilha;
- Precisamos de um indicador de qual elemento do vetor é o atual topo da pilha.

24	
89	
12	
4	
55	
20	5

# Pilhas Usando Vetores

- Vetores possuem um espaço limitado para armazenar dados;
- Necessitamos definir um espaço grande o suficiente para a nossa pilha;
- Necessitamos de um indicador de qual elemento do vetor é o atual topo da pilha.
- **Pilha Cheia!**

24	
89	
12	
4	
55	
20	5

# Pilhas Encadeadas

- A estrutura é limitada pela memória disponível;
- Não é necessário definir um valor fixo para o tamanho da Pilha;



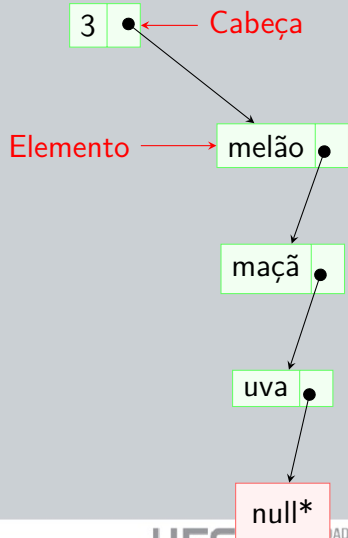
# Pilhas Encadeadas

- A estrutura é limitada pela memória disponível;
- Não é necessário definir um valor fixo para o tamanho da Pilha;



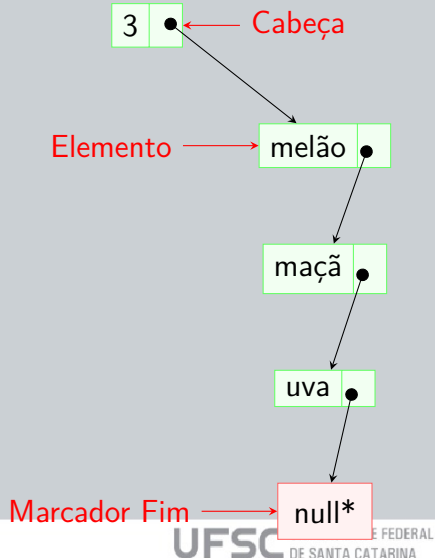
# Pilhas Encadeadas

- A estrutura é limitada pela memória disponível;
- Não é necessário definir um valor fixo para o tamanho da Pilha;



# Pilhas Encadeadas

- A estrutura é limitada pela memória disponível;
- Não é necessário definir um valor fixo para o tamanho da Pilha;



# Modelagem da Cabeça de Pilha

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

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



# Modelagem da Elemento de Pilha

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

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

# Modelagem da Elemento de Pilha

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

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

# Modelagem da Pilha Encadeada

- Aspecto Funcional:
  - Temos que colocar e retirar dados da pilha;
  - Temos que testar se a pilha está vazia;
  - Temos que inicializar a pilha.

# Modelagem da Pilha Encadeada

- Inicializar ou limpar:
  - Pilha();
  - ~Pilha();
  - detroiPilha();
- Testar se a pilha está vazia:
  - bool pilhaVazia();
- Colocar e retirar dados da pilha:
  - empilha(T dado);
  - T desempilha();
  - T topo();

# Método *Pilha()*

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

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

# Método *~Pilha()*

- Chamamos DestroiPilha();

```
~Pilha()  
inicio  
    DestroiPilha();  
fim;
```

## Método *pilhaVazia()*

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

- Um algoritmo pilhaCheia não existe na Pilha Encadeada;

# Método *pilhaVazia()*

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

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



## Método *empilha*( $T$ dado)

- Testamos se é possível alocar um elemento;

## Método *empilha*( $T$ dado)

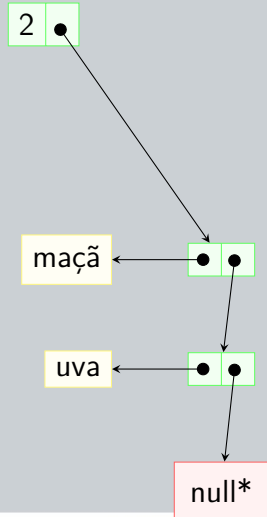
- Testamos se é possível alocar um elemento;
- Fazemos o próximo deste novo elemento ser o topo da pilha;

## Método *empilha*( $T$ dado)

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

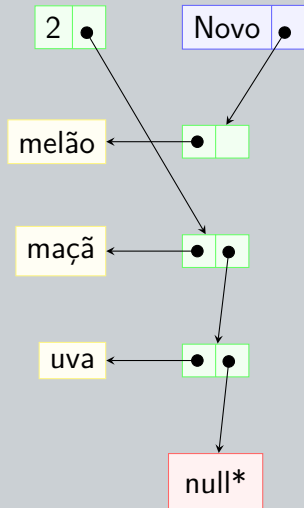
# Método *empilha(T dado)*

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



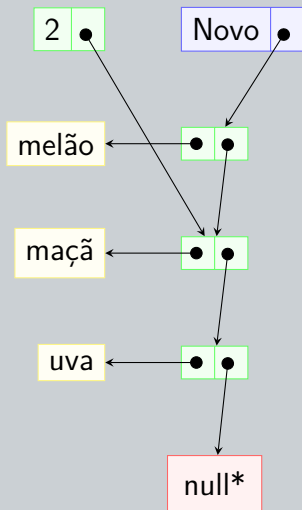
## Método *empilha(T dado)*

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



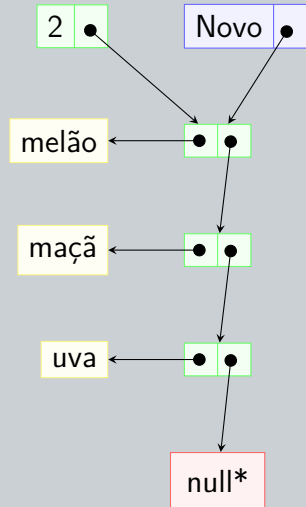
## Método *empilha(T dado)*

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



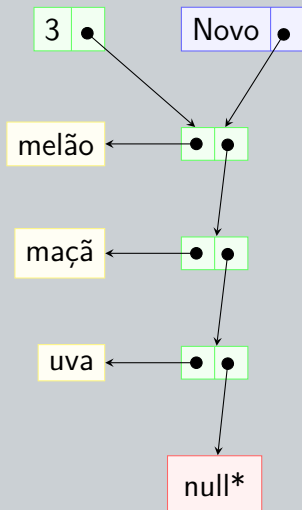
# Método *empilha(T dado)*

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



## Método *empilha(T dado)*

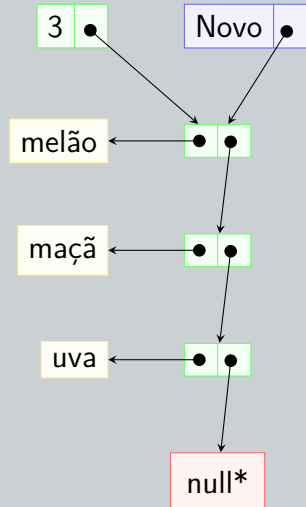
- Testamos se é possível alocar um elemento;
- Fazemos o próximo deste novo elemento ser o topo da pilha;
- Fazemos a cabeça de pilha apontar para o novo elemento.





# Método *empilha(T dado)*

- Testamos se é possível alocar um elemento;
- Fazemos o próximo deste novo elemento ser o topo da pilha;
- Fazemos a cabeça de pilha apontar para o novo elemento.
- Semelhanças??



# Método *T desempilha()*

- Testamos se há elementos;

# Método *T desempilha()*

- Testamos se há elementos;
- Decrementamos o tamanho;

## Método *T desempilha()*

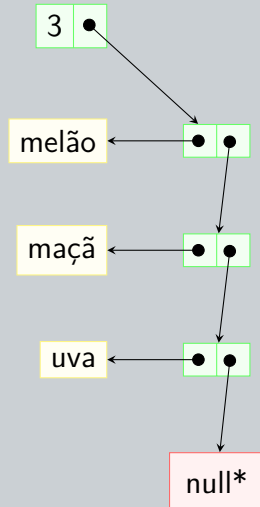
- Testamos se há elementos;
- Decrementamos o tamanho;
- Liberamos a memória do elemento;

# Método *T desempilha()*

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

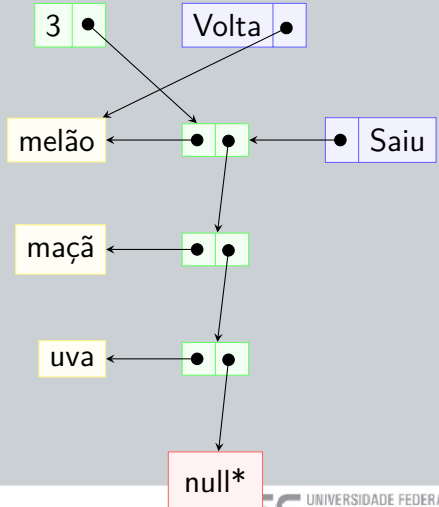
# Método *T desempilha()*

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



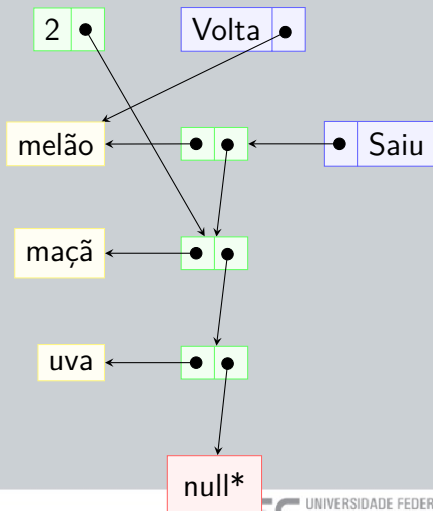
# Método *T desempilha()*

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



# Método *T desempilha()*

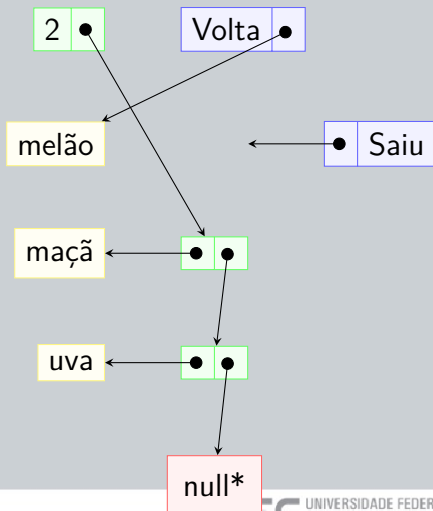
- Testamos se há elementos;
- Decrementamos o tamanho;
- Liberamos a memória do elemento;
- Devolvemos a informação.





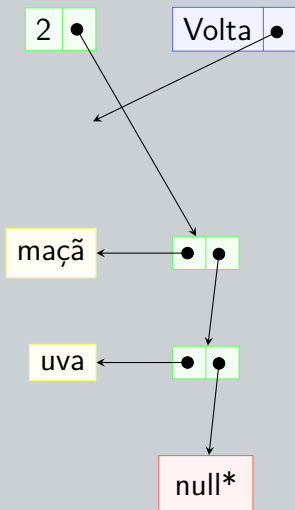
# Método *T desempilha()*

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



# Método *T desempilha()*

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



# Trabalho Pilha Encadeada

- Implemente uma classe Pilha todas as operações vistas;
- Implemente a pilha 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.

# Conceito de Filas

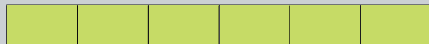
## Fila

É uma estrutura de dados cujo funcionamento é inspirado no conceito “natural” de enfileiramento, onde o primeiro a entrar é o primeiro a sair (FIFO).

# Conceito de Filas

## Fila

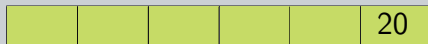
É uma estrutura de dados cujo funcionamento é inspirado no conceito “natural” de enfileiramento, onde o primeiro a entrar é o primeiro a sair (FIFO).



# Conceito de Filas

## Fila

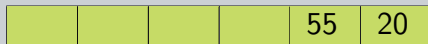
É uma estrutura de dados cujo funcionamento é inspirado no conceito “natural” de enfileiramento, onde o primeiro a entrar é o primeiro a sair (FIFO).



# Conceito de Filas

## Fila

É uma estrutura de dados cujo funcionamento é inspirado no conceito “natural” de enfileiramento, onde o primeiro a entrar é o primeiro a sair (FIFO).



# Conceito de Filas

## Fila

É uma estrutura de dados cujo funcionamento é inspirado no conceito “natural” de enfileiramento, onde o primeiro a entrar é o primeiro a sair (FIFO).

			4	55	20
--	--	--	---	----	----



# Conceito de Filas

## Fila

É uma estrutura de dados cujo funcionamento é inspirado no conceito “natural” de enfileiramento, onde o primeiro a entrar é o primeiro a sair (FIFO).

		12	4	55	20
--	--	----	---	----	----

# Conceito de Filas

## Fila

É uma estrutura de dados cujo funcionamento é inspirado no conceito “natural” de enfileiramento, onde o primeiro a entrar é o primeiro a sair (FIFO).

	89	12	4	55	20
--	----	----	---	----	----

# Conceito de Filas

## Fila

É uma estrutura de dados cujo funcionamento é inspirado no conceito “natural” de enfileiramento, onde o primeiro a entrar é o primeiro a sair (FIFO).

24	89	12	4	55	20
----	----	----	---	----	----

# Conceito de Filas

## Fila

É uma estrutura de dados cujo funcionamento é inspirado no conceito “natural” de enfileiramento, onde o primeiro a entrar é o primeiro a sair (FIFO).

	24	89	12	4	55
--	----	----	----	---	----

# Conceito de Filas

## Fila

É uma estrutura de dados cujo funcionamento é inspirado no conceito “natural” de enfileiramento, onde o primeiro a entrar é o primeiro a sair (FIFO).

		24	89	12	4
--	--	----	----	----	---

# Conceito de Filas

## Fila

É uma estrutura de dados cujo funcionamento é inspirado no conceito “natural” de enfileiramento, onde o primeiro a entrar é o primeiro a sair (FIFO).

		24	89	12	4
--	--	----	----	----	---

- Duas operações:
  - Incluir no Fim;
  - Retirar do Inicio;

# Conceito de Filas

- É importante para gerência de dados/processos por ordem cronológica:
  - Fila de impressão em uma impressora de rede;
  - Fila de pedidos de uma expedição ou tele-entrega;
- É importante para simulação de processos sequenciais:
  - chão de fábrica: fila de camisetas a serem estampadas;
  - comércio: simulação de fluxo de um caixa de supermercado;
  - tráfego: simulação de um cruzamento com um semáforo.

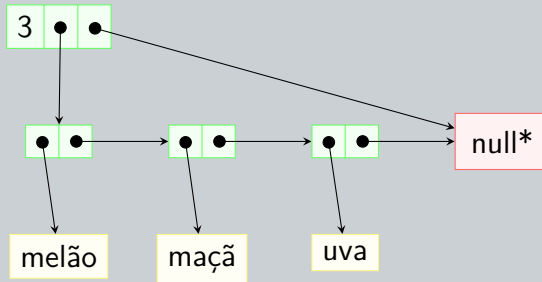
# Filas Usando Vetores

- Vetores possuem um espaço limitado para armazenar dados;
- Precisamos definir um espaço grande o suficiente para a nossa pilha;
- Precisamos de um indicador de qual elemento do vetor é o atual topo da pilha.
- Incluímos sempre no fim.
- **Fila Cheia!**

24	
89	
12	
4	
55	
20	5



# Modelagem de Fila Encadeada



# Modelagem da Cabeça de Fila Encadeada

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

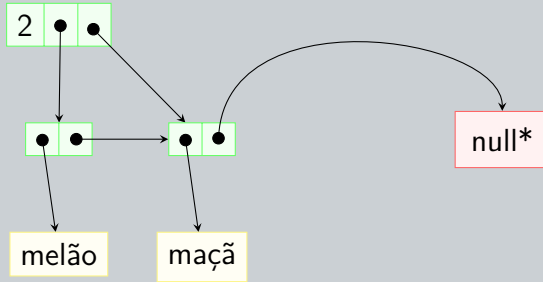
```
classe Lista {  
    Elemento *_dados;  
    Elemento *_fim;  
    inteiro _tamanho;  
};
```

## Método *Fila()*

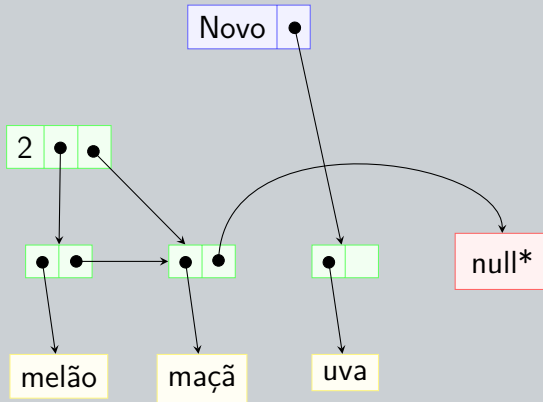
- Inicializamos o ponteiro para `_dados` nulo;
- Inicializamos o ponteiro para `_fim` nulo;
- Inicializamos o tamanho para "0";

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

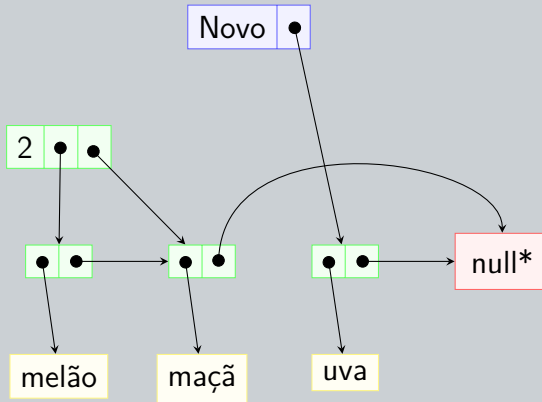
## Método *adiciona*(*T dado*)



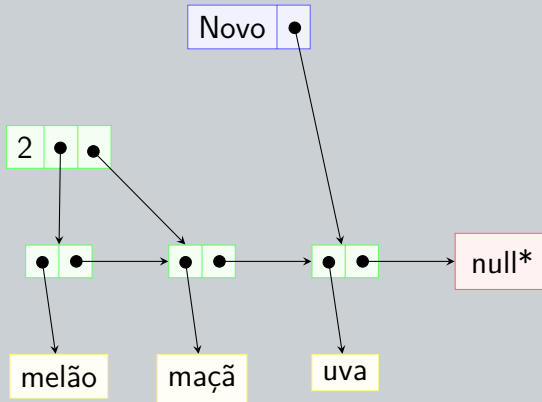
# Método *adiciona*(*T dado*)



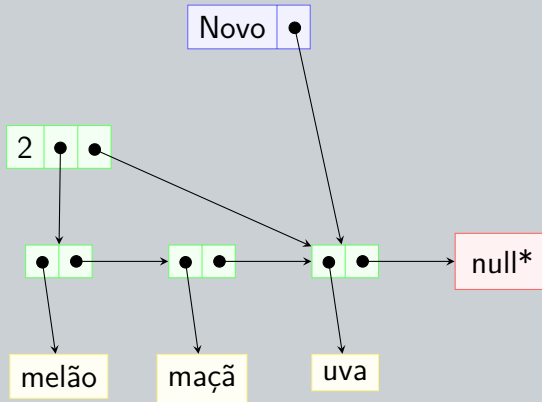
# Método *adiciona*(*T dado*)



# Método *adiciona*(*T dado*)

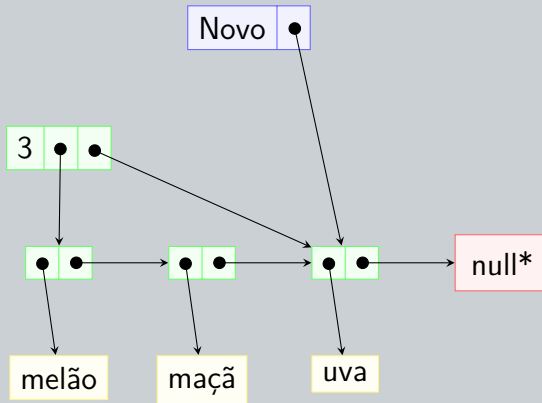


# Método *adiciona*(*T dado*)

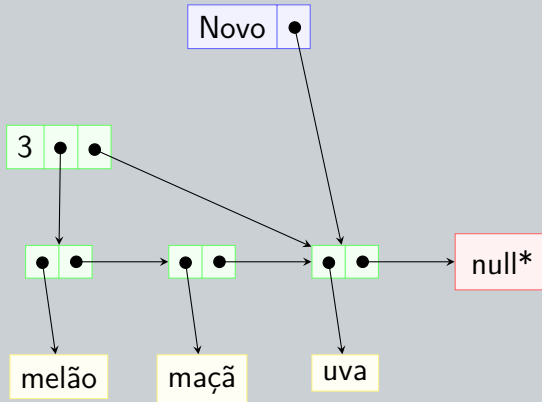




# Método *adiciona*(*T dado*)

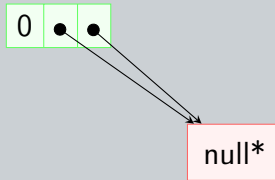


# Método *adiciona*(*T dado*)

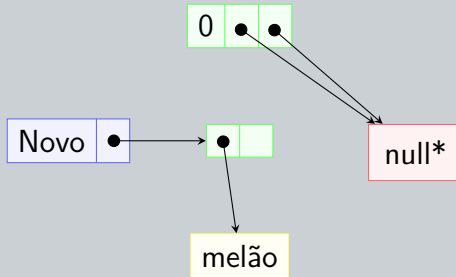


- Semelhanças??

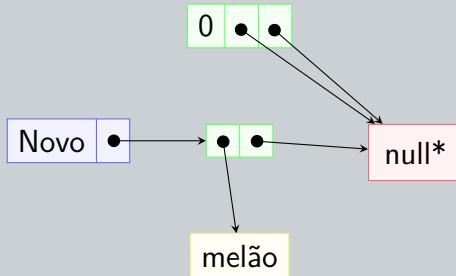
# Método *adiciona*(*T dado*) - Caso Especial Fila Vazia



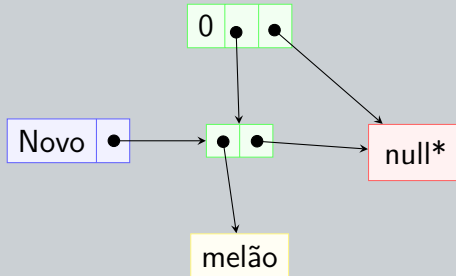
# Método *adiciona*(*T dado*) - Caso Especial Fila Vazia



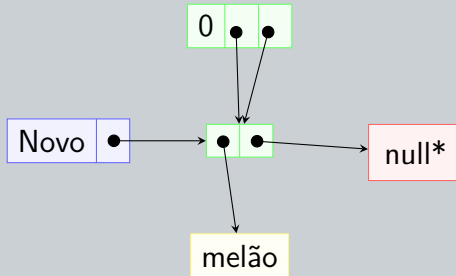
# Método *adiciona*(*T dado*) - Caso Especial Fila Vazia



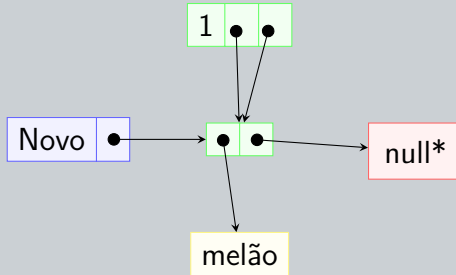
# Método *adiciona*(*T dado*) - Caso Especial Fila Vazia



# Método *adiciona*(*T dado*) - Caso Especial Fila Vazia



# Método *adiciona*(*T dado*) - Caso Especial Fila Vazia

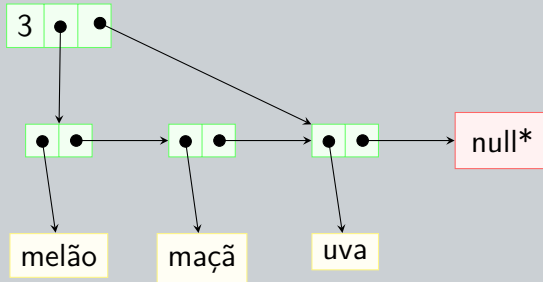




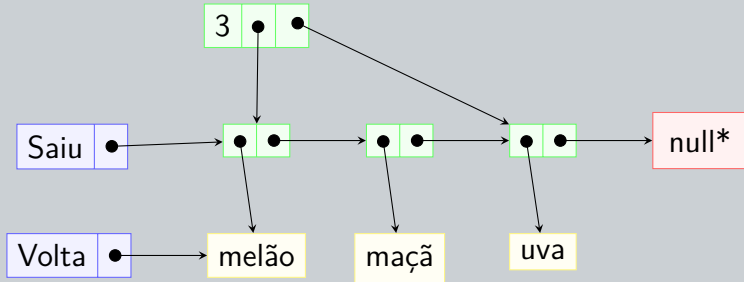
# Método *adiciona*(*T dado*)

```
adicionaNaPosicao(T dado, int posicao)
  Elemento *novo; // auxiliar.
  inicio
    novo <- aloque(Elemento);
    SE ( novo == NULO) THROW(ERROFILACHEIA);
    SE filaVazia() ENTAO
      _dados <- novo
    SENA0
      _fim->_proximo <- novo;
    FIM SE
    novo->_proximo <- NULO;
    novo->_info <- dado;
    _fim <- novo;
    _tamanho <- _tamanho + 1;
  FIM SE
fim;
```

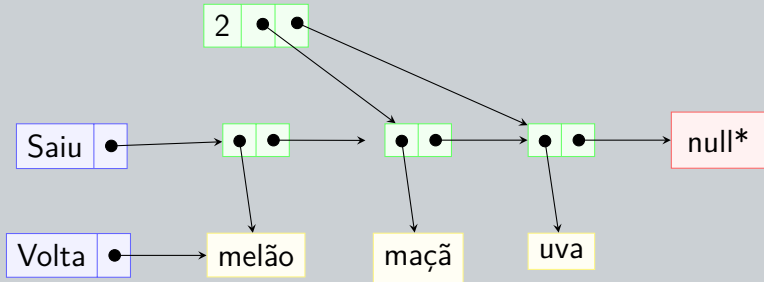
# Método $T\ retira()$



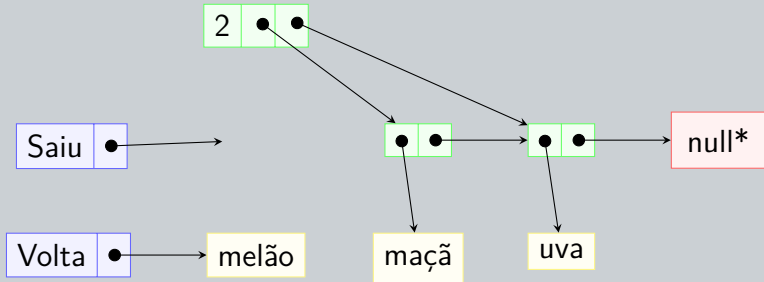
# Método $T\ retira()$



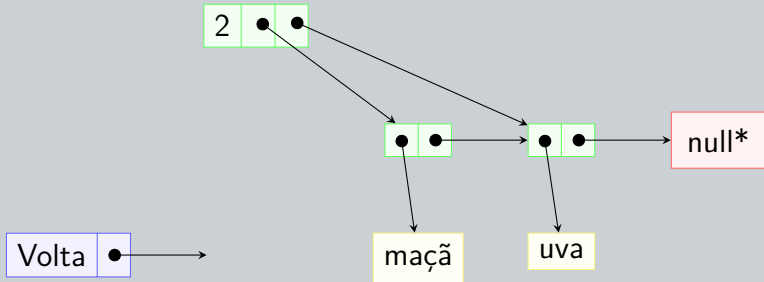
# Método $T\ retira()$



# Método $T\ retira()$



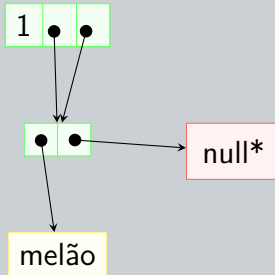
# Método $T$ *retira()*



- Semelhanças??

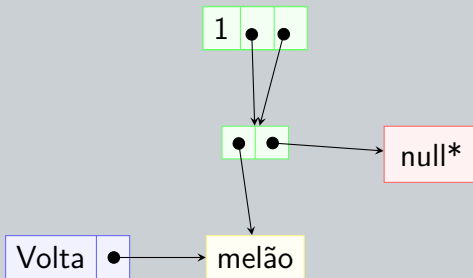
# Método $T\ retira()$ - Caso Especial

## Fila Unitária



# Método $T\ retira()$ - Caso Especial

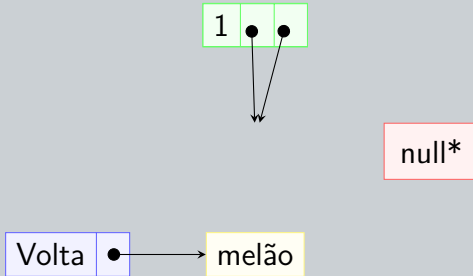
## Fila Unitária





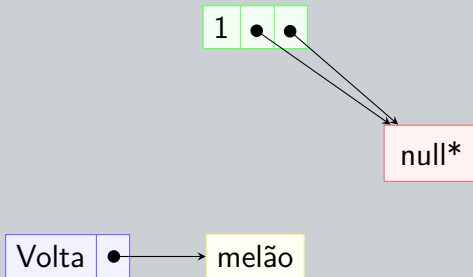
# Método $T\ retira()$ - Caso Especial

## Fila Unitária



# Método $T\ retira()$ - Caso Especial

## Fila Unitária



# Método *T retira()*

```
T retira()
  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;
  //Se SAIU for o único, próximo é NULO e está certo.
  SE (_tamanho = 1) ENTAO
    //Fila unitária: devo anular o _fim também.
    _fim <- NULO;
  FIM SE
  _tamanho <- _tamanho - 1;
  LIBERE(saiu);
  RETORNE(volta);
FIM SE
fim;
```

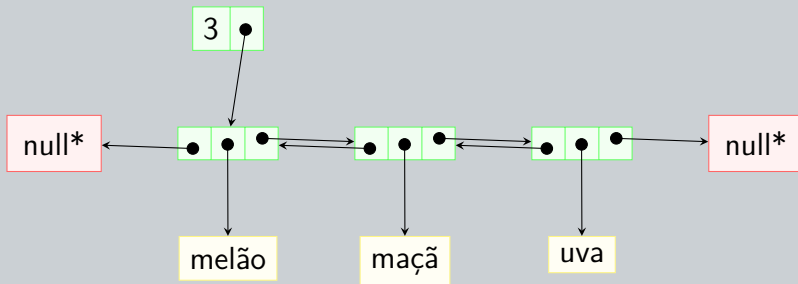
# Trabalho Fila Encadeada

- Implemente uma classe Fila todas as operações vistas;
- Implemente a fila 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.

# Listas Duplamente Encadeadas

- A Lista Encadeada e a Fila Encadeada possuem a desvantagem de somente podermos caminhar em uma direção:
  - Vimos que para olhar um elemento pelo qual “acabamos de passar” precisamos de uma variável auxiliar “anterior”;
  - Para olhar outros elementos ainda anteriores não temos nenhum meio, a não ser começar de novo.
- A Lista Duplamente Encadeada é uma estrutura de lista que permite deslocamento em ambos os sentidos:
  - Útil para representar conjuntos de eventos ou objetos a serem percorridos em dois sentidos;
  - Útil também quando realizamos uma busca aproximada e nos movemos para a frente e para trás.

# Lista Duplamente Encadeada



# Modelagem da Cabeça de Lista Dupla

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

```
classe ListaDupla {  
    ElementoDuplo *_dados;  
    inteiro _tamanho;  
};
```

# Modelagem da Elemento de Lista Dupla

- Aspecto Estrutural:
  - Necessitamos um ponteiro para o próximo elemento;
  - Necessitamos um ponteiro para o elemento anterior;
  - 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 ElementoDuplo {  
    Elemento *_proximo;  
    Elemento *_anterior;  
    T *_info;  
};
```



# Modelagem da Lista Duplamente 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 Duplamente Encadeada

- Inicializar ou limpar:
  - `ListaDupla();`
  - `limpaListaDupla();`
  - `~ListaDupla();`
- Testar se a lista está vazia ou cheia e outros testes:
  - `bool listaVaziaDupla();`
  - `int posicaoDupla(dado);`
  - `bool contemDupla(dado);`

# Modelagem da Lista Duplamente Encadeada

- Colocar e retirar dados da lista:
  - adicionaDupla(T dado);
  - adicionaNoInicioDupla(T dado);
  - adicionaNaPosicaoDupla(T dado, int posicao);
  - adicionaEmOrdemDupla(T dado);
  - T retiraDupla();
  - T retiraDoInicioDupla();
  - T retiraDaPosicaoDupla(int posicao);
  - T retiraEspecificoDupla(dado);

# Método *ListaDupla()*

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

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

# Método *~ListaDupla()*

- Chamamos DestroiLista();

```
~ListaDupla()  
inicio  
    DestroiListaDupla();  
fim;
```

## Método *listaVaziaDupla()*

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

- Um algoritmo ListaCheia não existe na Lista Duplamente Encadeada;

## Método *listaVaziaDupla()*

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

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

# Método *adicionaNoInicioDupla(T dato)*

- Testamos se é possível alocar um elemento;



## Método *adicionaNoInicioDupla(T dado)*

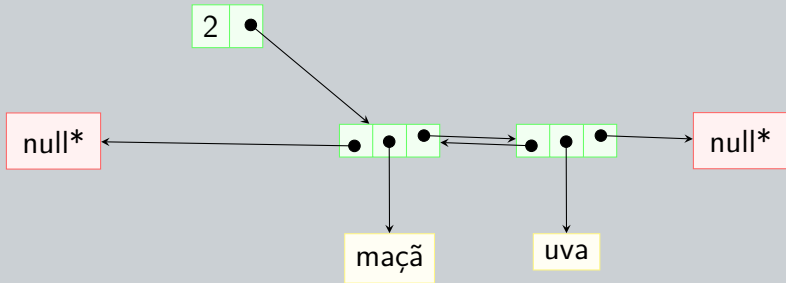
- Testamos se é possível alocar um elemento;
- Fazemos o próximo deste novo elemento ser o primeiro;

## Método *adicionaNoInicioDupla(T dato)*

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

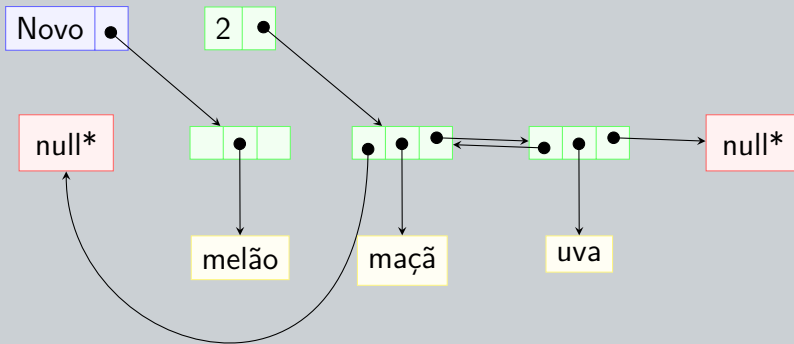
# Método *adicionaNoInicioDupla(T dado)*

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



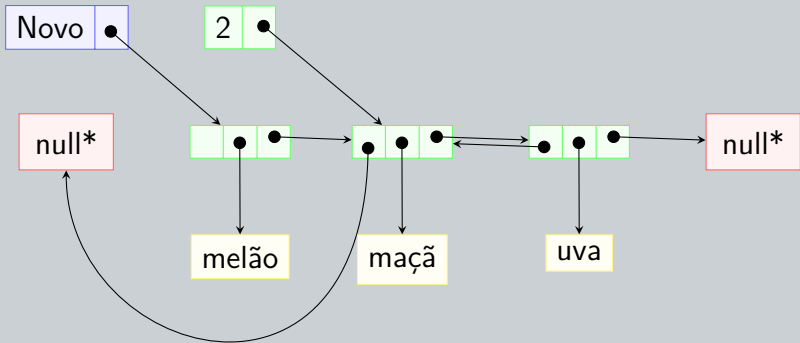
# Método *adicionaNoInicioDupla(T dado)*

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



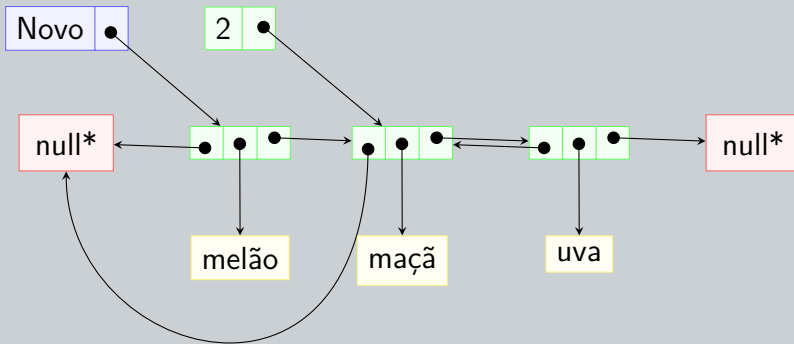
# Método *adicionaNoInicioDupla(T dado)*

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



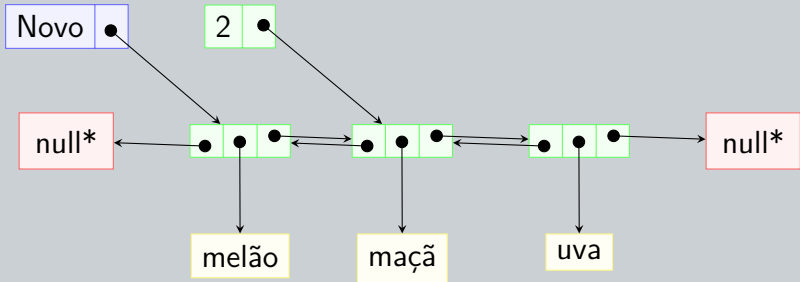
# Método *adicionaNoInicioDupla(T dado)*

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



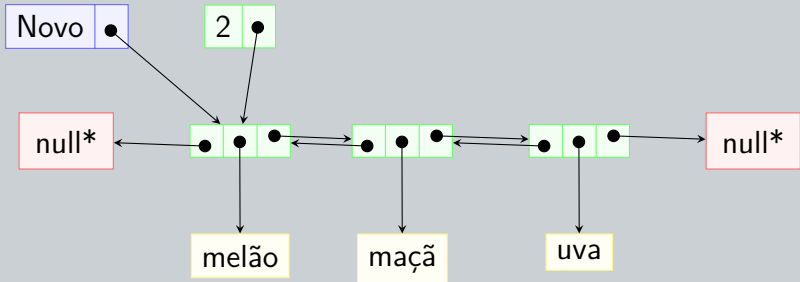
# Método *adicionaNoInicioDupla(T dado)*

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



# Método *adicionaNoInicioDupla(T dado)*

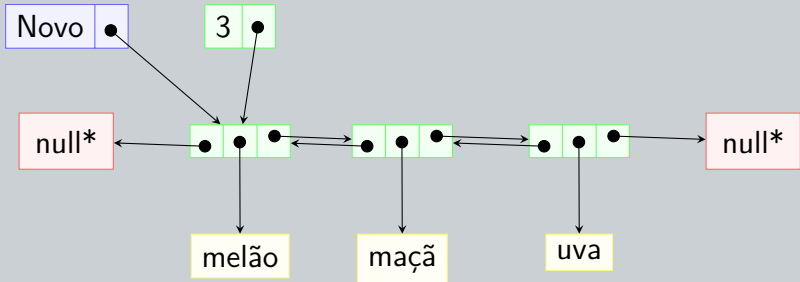
- Testamos se é possível alocar um elemento;
- Fazemos o próximo deste novo elemento ser o primeiro;
- Fazemos a cabeça de lista apontar para o novo elemento.





# Método *adicionaNoInicioDupla(T dado)*

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



## Método *adicionaNoInicioDupla*(T *dado*)

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

## Método *T* *retiraDoInicioDupla()*

- Testamos se há elementos;

## Método *T* *retiraDoInicioDupla()*

- Testamos se há elementos;
- Decrementamos o tamanho;

## Método *T retiraDoInicioDupla()*

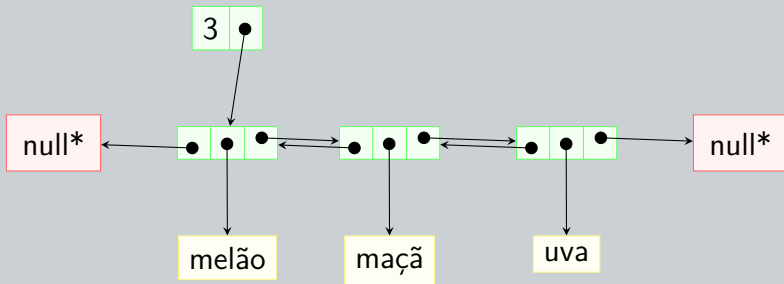
- Testamos se há elementos;
- Decrementamos o tamanho;
- Liberamos a memória do elemento;

## Método *T retiraDoInicioDupla()*

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

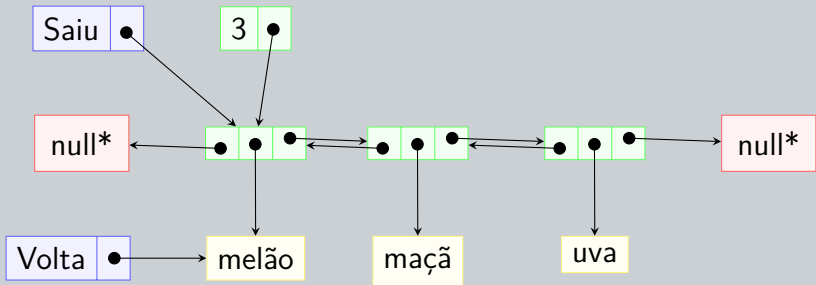
## Método *T* *retiraDoInicioDupla()*

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



# Método *T* *retiraDoInicioDupla()*

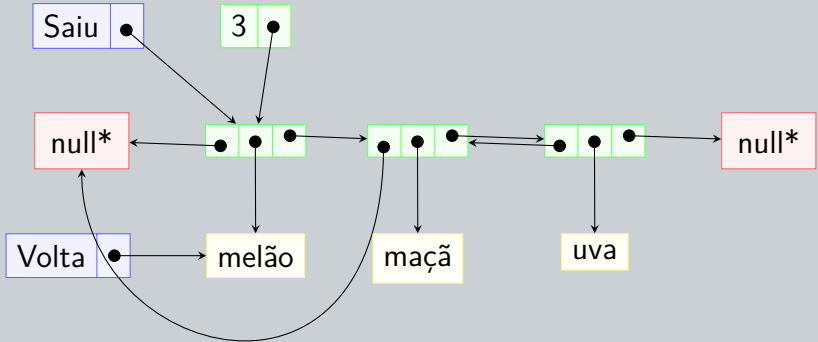
- Testamos se há elementos;
- Decrementamos o tamanho;
- Liberamos a memória do elemento;
- Devolvemos a informação.





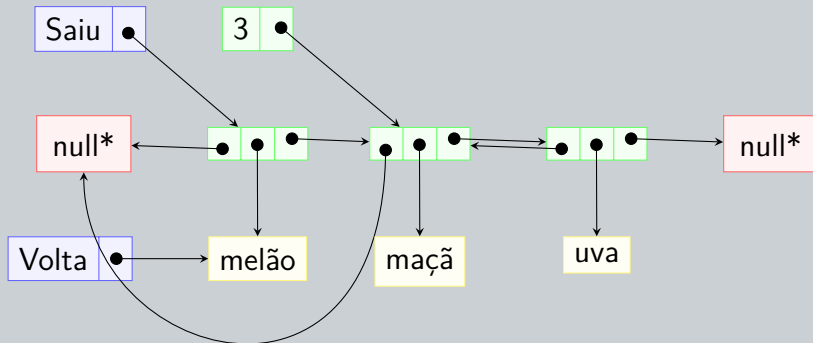
## Método *T* *retiraDoInicioDupla()*

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



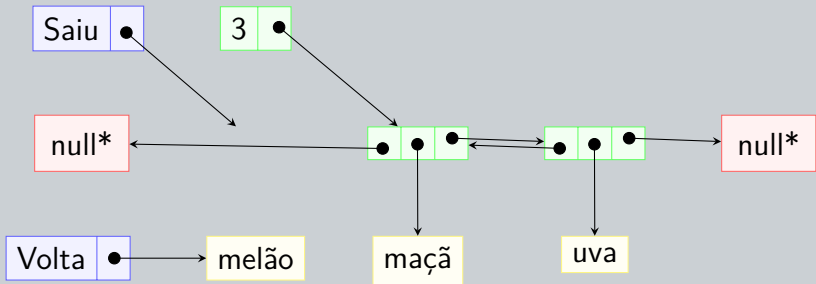
## Método *T* *retiraDoInicioDupla()*

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



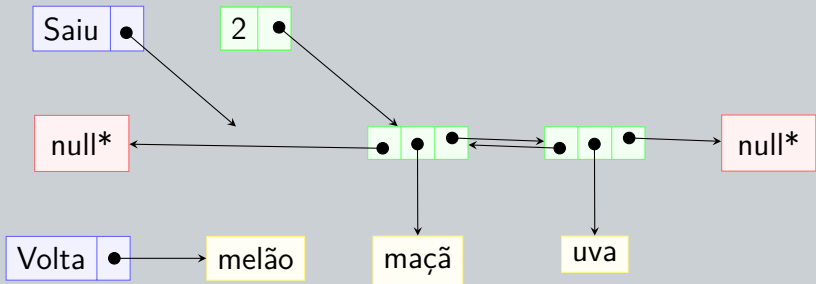
# Método *T* *retiraDoInicioDupla()*

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



# Método *T* *retiraDoInicioDupla()*

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



# Método *T* *retiraDoInicioDupla()*

```
T retiraDoInicioDupla()
  ElementoDuplo *saiu; //Variável auxiliar elemento.
  T *volta; //Variável auxiliar tipo T.
início
SE (listaVaziaDupla()) ENTAO
  THROW(ERROLISTAVAZIA);
SENAO
  saiu <- _dados;
  volta <- saiu->_info;
  _dados <- saiu->_proximo;
  SE (_dados ~= NULO) ENTAO
dados->_anterior <- NULO;
  FIM SE
  _tamanho <- _tamanho - 1;
  LIBERE(saiu);
  RETORNE(volta);
FIM SE
fim;
```

## Método *adicionaNaPosicaoDupla(T dado, int posicao)*

- Praticamente idêntico à lista encadeada;
- Procedimento:
  - Caminhamos até a posição;
  - Adicionamos o novo dado na posição;
  - Tratamos o caso especial;
  - Incrementamos o tamanho.
- Parâmetros:
  - O dado a ser inserido;
  - A posição onde inserir;

# Método *adicionaNaPosicaoDupla*(*T* *dado*, *int posicao*)

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

## Método *T retiraDaPosicaoDupla(int posicao)*

- Praticamente idêntico à lista encadeada;
- Procedimento:
  - Caminhamos até a posição;
  - Retiramos o novo dado na posição;
  - Tratamos o caso especial;
  - Decrementamos o tamanho.
- Parâmetros:
  - A posição onde retirar;



# Método *T* *retiraDaPosicaoDupla*(*int* *posicao*)

```
T retiraDaPosicaoDupla(int posicao)
  ElementoDuplo *anterior, *eliminar; //Variáveis elemento.
  T *volta; //Variável tipo T.
  inicio
  SE (posicao > _tamanho ) ENTAO THROW(ERROPOSSICAO);
  SENA0
  SE (posicao = 1) ENTAO RETORNE(retiraDoInicioDupla());
  SENA0
  anterior <- _dados;
  REPITA (posicao - 2) VEZES
    anterior <- anterior->_proximo;
  eliminar <- anterior->_proximo;
  volta <- eliminar->_info;
  anterior->_proximo <- eliminar->_proximo;
  SE eliminar->_proximo ~= NULO ENTAO
    eliminar->_proximo->_anterior <- anterior;
  _tamanho <- _tamanho - 1;
  LIBERE(eliminar);  RETORNE(volta);
  FIM SE
  FIM SE
fim;
```

# Método *adicionaEmOrdemDupla*(*T* *dado*)

- Idêntico à lista encadeada;
- Procedimento:
  - Necessitamos de uma função para comparar os dados “>”;
  - Procuramos pela posição onde inserir comparando dados;
  - Chamamos *adicionaNaPosiçãoDupla*().
- Parâmetros:
  - O dado a ser inserido;

## Por conta do aluno:

- Operações de inclusão e exclusão:
  - AdicionaDupla(dado);
  - RetiraDupla();
  - RetiraEspecíficoDupla(dado);
- Operações - inicializar ou limpar:
  - DestróiListaDupla();

# Trabalho Lista Duplamente Encadeada

- Implemente uma classe ListaDupla 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????



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