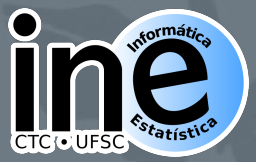
Estruturas de Dados

Listas Utilizando Vetores

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

2016.2



UNIVERSIDADE FEDERAL
DE SANTA CATARINA

Conceito de Listas

Lista

Uma Lista é um conjunto de dados dispostos e/ou acessáveis em uma sequência determinada.

Definições de Lista

- Lista é um conjunto de dados que pode possuir uma ordem intrínseca (Lista Ordenada) ou não;
- Este conjunto de dados pode ocupar espaços de memória fisicamente consecutivos, espelhando a sua ordem, ou não;
- Se os dados estiverem dispersos fisicamente, para que este conjunto seja uma lista, ele deve possuir operações e informações adicionais que permitam que seja tratado como tal (Lista Encadeada).

Definições de Lista

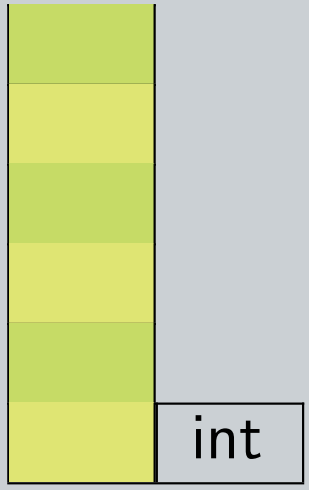
Lista

A Lista é uma estrutura de dados cujo funcionamento é inspirado no de uma lista "natural".

- Uma lista pode ser ordenada ou não:
- Quando for ordenada, pode o ser por alguma característica intrínseca dos dados (ex: ordem alfabética);
- Ela pode também refletir a ordem cronológica (ordem de inserção) dos dados.

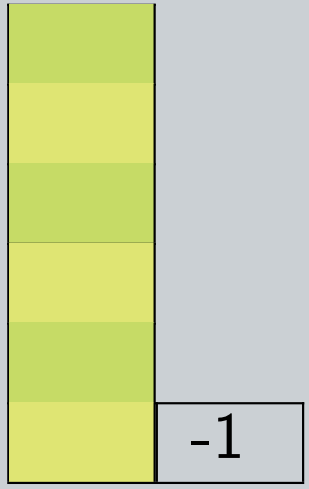
Listas Usando Vetores

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



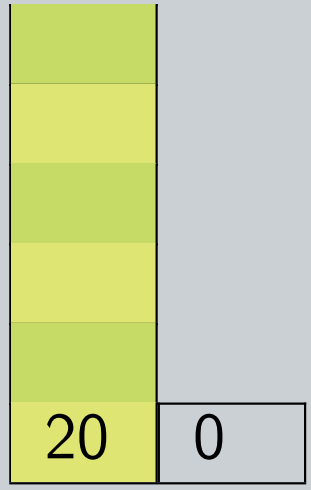
Listas Usando Vetores

- Vetores possuem um espaço limitado para armazenar dados;
- Necessitamos definir um espaço grande o suficiente para a nossa lista;
- Necessitamos de um indicador de qual elemento do vetor é o atual último da lista.
- Lista Vazia!



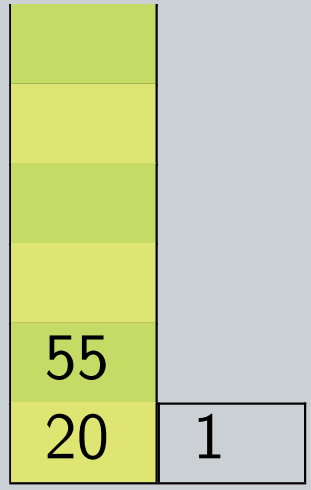
Listas Usando Vetores

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



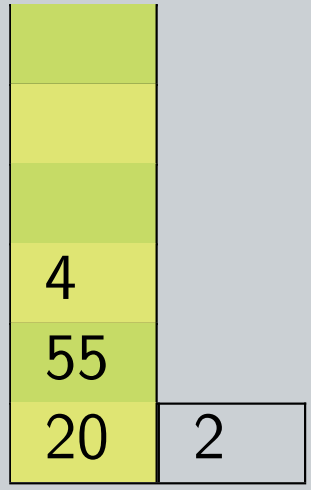
Listas Usando Vetores

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



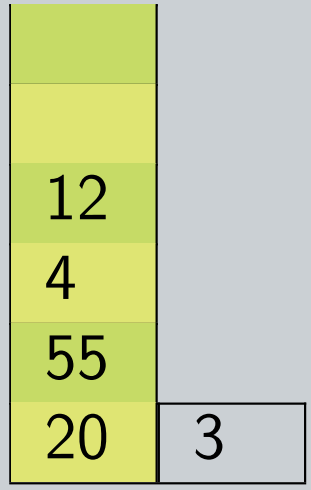
Listas Usando Vetores

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



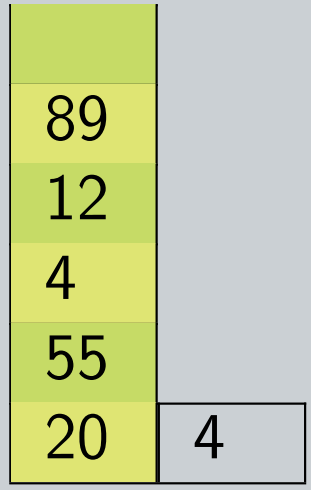
Listas Usando Vetores

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



Listas Usando Vetores

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



Listas Usando Vetores

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

Listas Usando Vetores

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

Modelagem da Lista de Vetor

- Aspecto Estrutural:
  - Necessitamos de um vetor para armazenar as informações;
  - Necessitamos de um indicador da posição atual do ultimo elemento da lista;

```
constantes MAXLISTA = 100;

classe Lista {
 T _dados[MAXLISTA]; // Vetor Estatico
 inteiro _ultimo;
};
```

Modelagem da Lista de Vetor

- Aspecto Estrutural:
  - Necessitamos de um vetor para armazenar as informações;
  - Necessitamos de um indicador da posição atual do ultimo elemento da lista;

```
constantes MAXLISTA = 100;

classe Lista {
 T* _dados; // _dados = new T[MAXLISTA];
 inteiro _ultimo;
 inteiro _tam = MAXLISTA;
};
```

Modelagem da Lista de Vetor

- Aspecto Funcional:
  - Temos que colocar e retirar dados da lista;
  - Temos que testar se a lista está vazia ou cheia (dentre outros testes);
  - Temos que inicializar a lista e garantir a ordem de seus elementos.

Modelagem da Lista de Vetor

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

Modelagem da Lista de Vetor

- 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()*

```
Lista()
inicio
   _dados = new T[_tam];
   _ultimo <- -1;
fim;
```

Método *Lista(int tam)*

```
Lista(int tam)
inicio
   _tam = tam;
   _dados = new T[tam];
   ultimo <- -1;
fim;
```

Método *limpaLista()*

```
void limpaLista()
inicio
  ultimo <- -1;
fim;
```

Método *Lista()*

```
void ~Lista()
inicio
  delete _dados;
fim;
```

Observação:

Este método é desnecessário. Ele só está aqui por completude da interface. Mais tarde veremos que, utilizando alocação dinâmica de memória, possuir um destrutor para Lista é muito importante.

Método *listaCheia()*

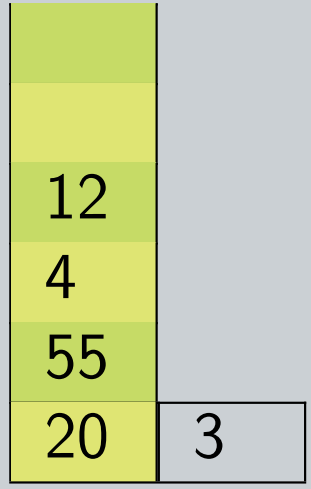
```
bool listaCheia()
inicio
 SE (_ultimo = _tam - 1) ENTAO
  RETORNE(Verdadeiro);
 SENAO
  RETORNE(Falso);
fim
```

Método *listaVazia()*

```
bool listaVazia()
inicio
SE (_ultimo = -1) ENTAO
 RETORNE(Verdadeiro)
SENAO
 RETORNE(Falso);
fim;
```

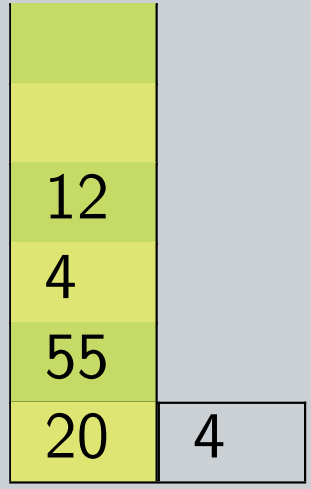
Método *adiciona(T dado)*

- Testamos se há espaço;
- Incrementamos o último;
- Adicionamos o novo dado na posição último.



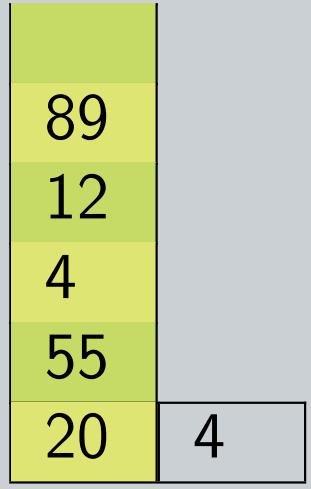
Método *adiciona(T dado)*

- Testamos se há espaço;
- Incrementamos o último;
- Adicionamos o novo dado na posição último.



Método *adiciona(T dado)*

- Testamos se há espaço;
- Incrementamos o último;
- Adicionamos o novo dado na posição último.

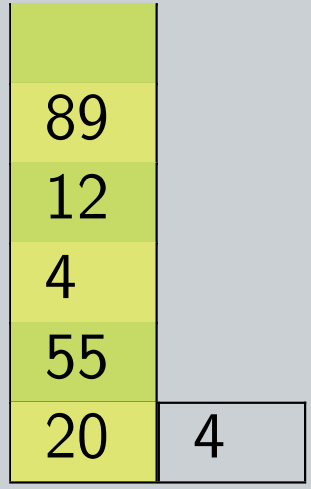


Método *adiciona(T dado)*

```
adiciona(T dado)
inicio
  SE (listaCheia) ENTAO
   THROW(ERROLISTACHEIA)
  SENAO
   _ultimo <- _ultimo + 1;
   _dados[_ultimo] <- dado;
  FIM SE
fim;
```

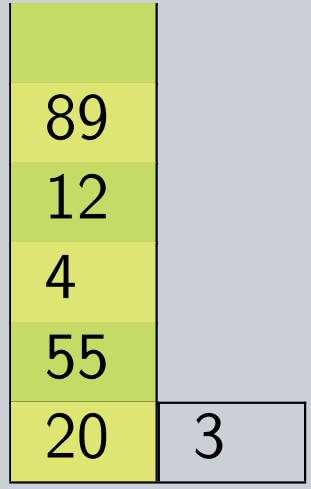
Método *T retira()*

- Testamos se há elementos;
- Decrementamos o último;
- Devolvemos o último elemento.



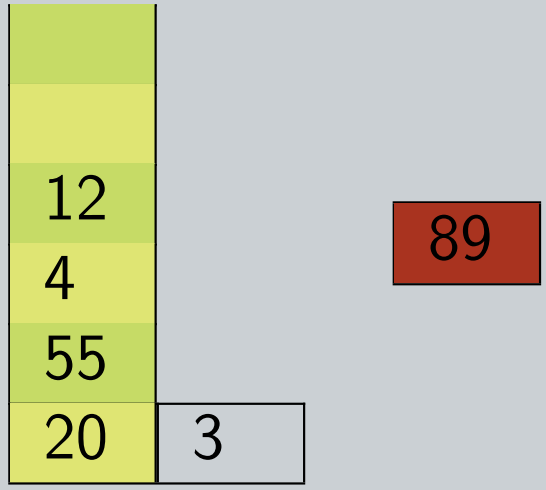
Método *T retira()*

- Testamos se há elementos;
- Decrementamos o último;
- Devolvemos o último elemento.



Método *T retira()*

- Testamos se há elementos;
- Decrementamos o último;
- Devolvemos o último elemento.

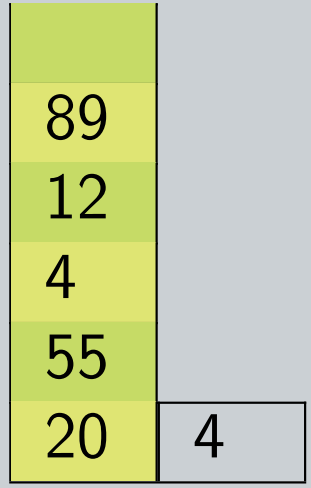


Método *T retira()*

```
T retira()
inicio
 SE (listaVazia) ENTAO
  THROW(ERROLISTAVAZIA)
 SENAO
  _ultimo <- _ultimo - 1;
  RETORNE(_dados[_ultimo + 1]);
 FIM SE
fim;
```

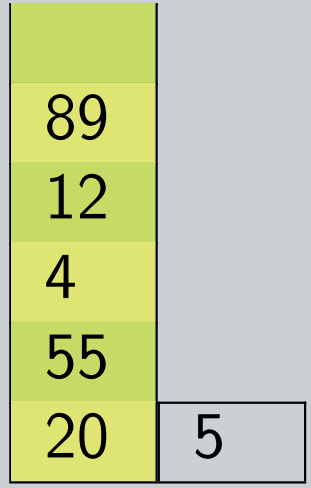
Método *adicionaNoInicio(T dado)*

- Testamos se há espaço;
- Incrementamos o último;
- Empurramos tudo para trás;
- Adicionamos o novo dado na primeira posição.



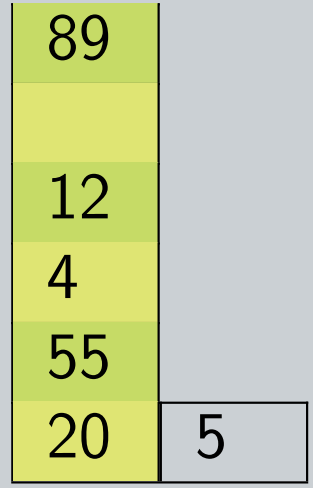
Método *adicionaNoInicio(T dado)*

- Testamos se há espaço;
- Incrementamos o último;
- Empurramos tudo para trás;
- Adicionamos o novo dado na primeira posição.



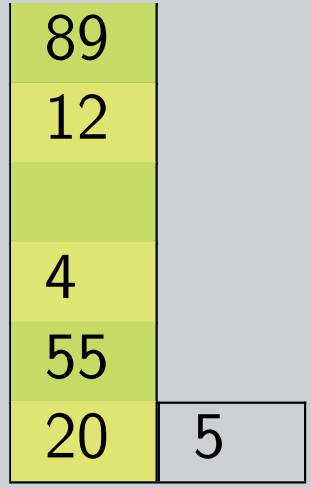
Método *adicionaNoInicio(T dado)*

- Testamos se há espaço;
- Incrementamos o último;
- Empurramos tudo para trás;
- Adicionamos o novo dado na primeira posição.



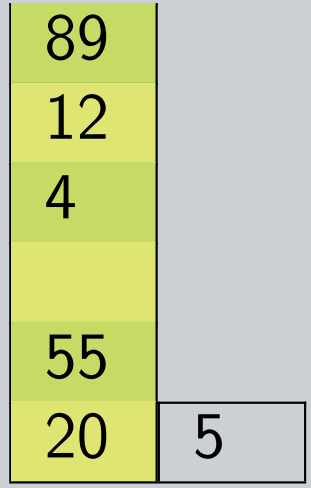
Método *adicionaNoInicio(T dado)*

- Testamos se há espaço;
- Incrementamos o último;
- Empurramos tudo para trás;
- Adicionamos o novo dado na primeira posição.



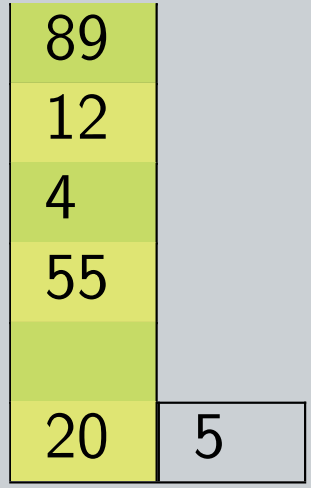
Método *adicionaNoInicio(T dado)*

- Testamos se há espaço;
- Incrementamos o último;
- Empurramos tudo para trás;
- Adicionamos o novo dado na primeira posição.



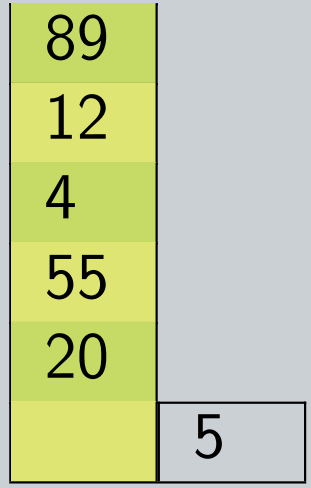
Método *adicionaNoInicio(T dado)*

- Testamos se há espaço;
- Incrementamos o último;
- Empurramos tudo para trás;
- Adicionamos o novo dado na primeira posição.



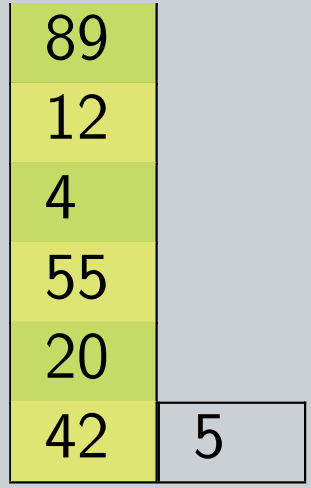
Método *adicionaNoInicio(T dado)*

- Testamos se há espaço;
- Incrementamos o último;
- Empurramos tudo para trás;
- Adicionamos o novo dado na primeira posição.



Método *adicionaNoInicio(T dado)*

- Testamos se há espaço;
- Incrementamos o último;
- Empurramos tudo para trás;
- Adicionamos o novo dado na primeira posição.



Método *adicionaNoInicio(T dado)*

```
adicionaNoInicio(T dado)
 int posicao; //Var auxiliar para "caminhar"
 inicio
  SE (listaCheia) ENTAO
   THROW(ERROLISTACHEIA)
  SENAO
   _ultimo <- _ultimo + 1;
   posicao <- _ultimo;
   ENQUANTO (posicao > 0) FACA
    //Empurrar tudo para tras
    _dados[posicao] <- _dados[posicao - 1];
    posicao <- posicao - 1;
   FIM ENQUANTO
   _dados[0] <- dado;
  FIM SE
fim;
```

Método *adicionaNoInicio(T dado)*

```
adicionaNoInicio(T dado)
 int posicao; //Var auxiliar para "caminhar"
 inicio
    SE (listaCheia) ENTAO
     THROW(ERROLISTACHEIA)
    SENAO
   _ultimo <- _ultimo + 1;
   posicao <- _ultimo;
   ENQUANTO (posicao > 0) FACA
    //Empurrar tudo para tras
    _dados[posicao] <- _dados[posicao - 1];
    posicao <- posicao - 1;
   FIM ENQUANTO
   _dados[0] <- dado;
  FIM SE
fim;
```

Método *adicionaNoInicio(T dado)*

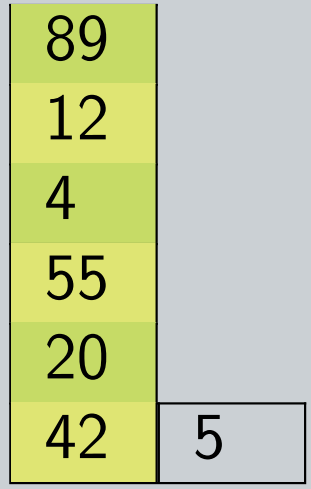
```
adicionaNoInicio(T dado)
 int posicao; //Var auxiliar para "caminhar"
 inicio
   SE (listaCheia) ENTAO)
    THROW(ERROLISTACHEIA)
   SENAO
    _ultimo <- _ultimo + 1;
    posicao <- _ultimo;
    ENQUANTO (posicao > 0) FACA
     //Empurrar tudo para tras
     _dados[posicao] <- _dados[posicao - 1];
     posicao <- posicao - 1;
    FIM ENQUANTO
   _dados[0] <- dado;
  FIM SE
fim;
```

Método *adicionaNoInicio(T dado)*

```
adicionaNoInicio(T dado)
 int posicao; //Var auxiliar para "caminhar"
 inicio
   SE (listaCheia) ENTAO
    THROW(ERROLISTACHEIA)
   SENAO
   _ultimo <- _ultimo + 1;
   posicao <- _ultimo;
   ENQUANTO (posicao > 0) FACA
    //Empurrar tudo para tras
    _dados[posicao] <- _dados[posicao - 1];
    posicao <- posicao - 1;
   FIM ENQUANTO
    _dados[0] <- dado;
  FIM SE
fim;
```

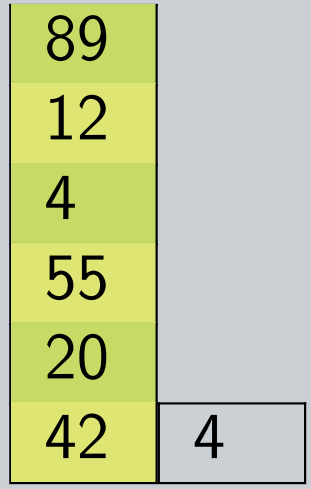
Método *T retiraDoInicio()*

- Testamos se há elementos;
- Decrementamos o último;
- Salvamos o primeiro elemento;
- Empurramos tudo para a frente.



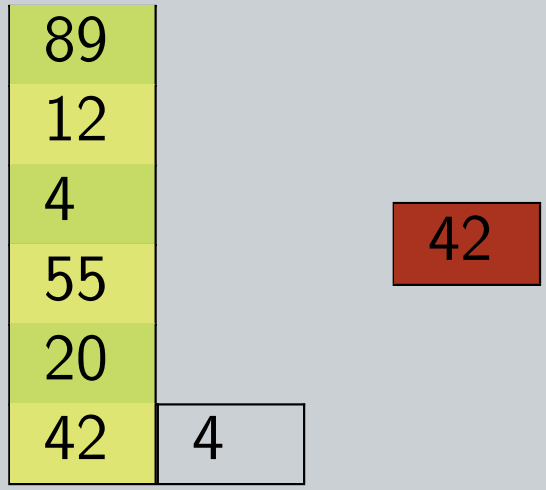
Método *T retiraDoInicio()*

- Testamos se há elementos;
- Decrementamos o último;
- Salvamos o primeiro elemento;
- Empurramos tudo para a frente.



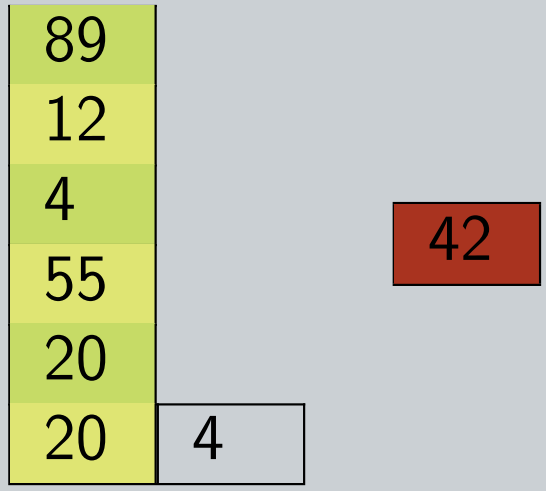
Método *T retiraDoInicio()*

- Testamos se há elementos;
- Decrementamos o último;
- Salvamos o primeiro elemento;
- Empurramos tudo para a frente.



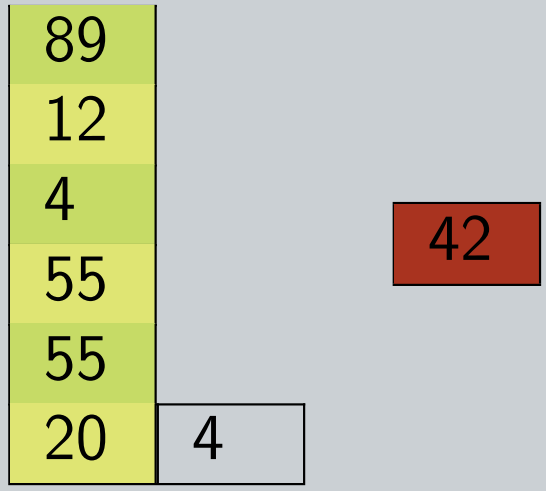
Método *T retiraDoInicio()*

- Testamos se há elementos;
- Decrementamos o último;
- Salvamos o primeiro elemento;
- Empurramos tudo para a frente.



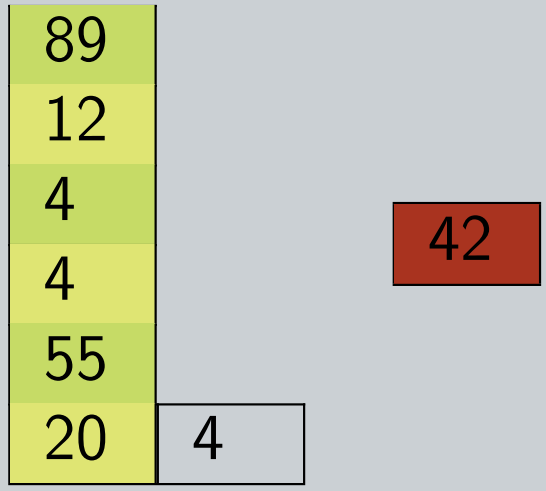
Método *T retiraDoInicio()*

- Testamos se há elementos;
- Decrementamos o último;
- Salvamos o primeiro elemento;
- Empurramos tudo para a frente.



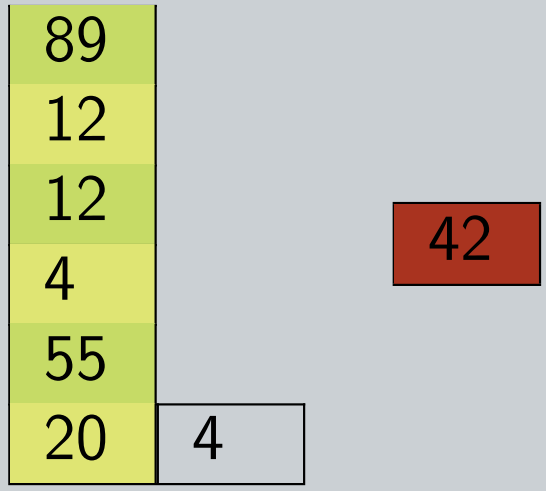
Método *T retiraDoInicio()*

- Testamos se há elementos;
- Decrementamos o último;
- Salvamos o primeiro elemento;
- Empurramos tudo para a frente.



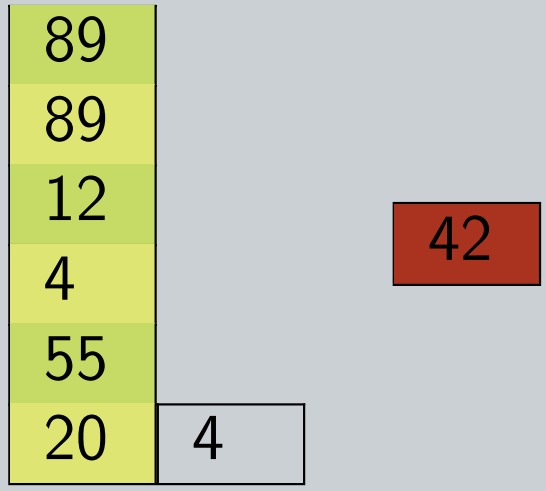
Método *T retiraDoInicio()*

- Testamos se há elementos;
- Decrementamos o último;
- Salvamos o primeiro elemento;
- Empurramos tudo para a frente.



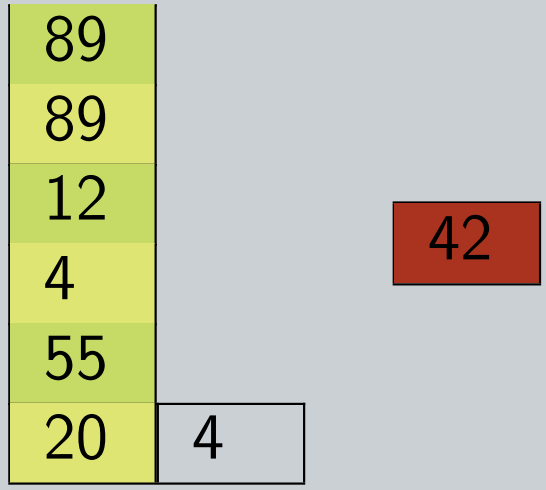
Método *T retiraDoInicio()*

- Testamos se há elementos;
- Decrementamos o último;
- Salvamos o primeiro elemento;
- Empurramos tudo para a frente.



Método *T retiraDoInicio()*

- Testamos se há elementos;
- Decrementamos o último;
- Salvamos o primeiro elemento;
- Empurramos tudo para a frente.
- Não perdemos tempo apagando o segundo "89", pois o _ultimo determina que a lista acaba antes dele!



Método *T retiraDoInicio()*

```
T retiraDoInicio()
 int posicao; T valor;
inicio
 SE (listaVazia) ENTAO
  THROW(ERROLISTAVAZIA)
 SENAO
  _ultimo <- _ultimo - 1;
  valor <- _dados[0];
  posicao <- 0;
  ENQUANTO (posicao <= _ultimo) FACA
   //Empurrar tudo para a frente
   _dados[posicao] <- _dados[posicao + 1];
   posicao <- posicao + 1;
  FIM ENQUANTO
  RETORNE(valor);
 FIM SE
fim;
```

Método *T retiraDoInicio()*

```
T retiraDoInicio()
 int posicao; T valor;
inicio
  SE (listaVazia) ENTAO
   THROW(ERROLISTAVAZIA)
  SENAO
   _ultimo <- _ultimo - 1;
   valor <- _dados[0];
   posicao <- 0;
   ENQUANTO (posicao <= _ultimo) FACA
    //Empurrar tudo para a frente
    _dados[posicao] <- _dados[posicao + 1];
    posicao <- posicao + 1;
   FIM ENQUANTO
   RETORNE(valor);
 FIM SE
fim;
```

Método *T retiraDoInicio()*

```
T retiraDoInicio()
 int posicao; T valor;
inicio
 SE (listaVazia) ENTAO
  THROW(ERROLISTAVAZIA)
 SENAO
  _ultimo <- _ultimo - 1;
   valor <- _dados[0];
  posicao <- 0;
  ENQUANTO (posicao <= _ultimo) FACA
   //Empurrar tudo para a frente
   _dados[posicao] <- _dados[_posicao + 1];
   posicao <- posicao + 1;
  FIM ENQUANTO
  RETORNE(valor);
 FIM SE
fim;
```

Método *T retiraDoInicio()*

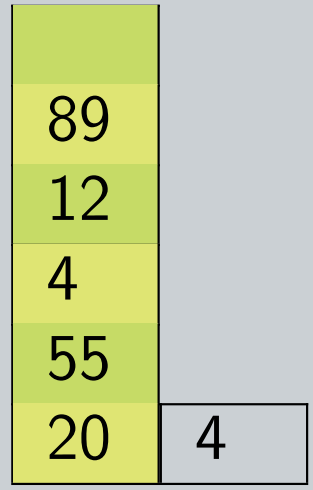
```
T retiraDoInicio()
 int posicao; T valor;
inicio
 SE (listaVazia) ENTAO
  THROW(ERROLISTAVAZIA)
 SENAO
  _ultimo <- _ultimo - 1;
  valor <- _dados[0];
   posicao <- 0;
   ENQUANTO (posicao <= _ultimo) FACA
    //Empurrar tudo para a frente
    _dados[posicao] <- _dados[posicao + 1];
    posicao <- posicao + 1;
   FIM ENQUANTO
  RETORNE(valor);
 FIM SE
fim;
```

Método *T retiraDoInicio()*

```
T retiraDoInicio()
 int posicao; T valor;
inicio
 SE (listaVazia) ENTAO
  THROW(ERROLISTAVAZIA)
 SENAO
  _ultimo <- _ultimo - 1;
  valor <- _dados[0];
  posicao <- 0;
  ENQUANTO (posicao <= _ultimo) FACA
   //Empurrar tudo para a frente
   _dados[_posicao] <- _dados[_posicao + 1];
   _posicao <- posicao + 1;
  FIM ENQUANTO
   RETORNE(valor);
 FIM SE
fim;
```

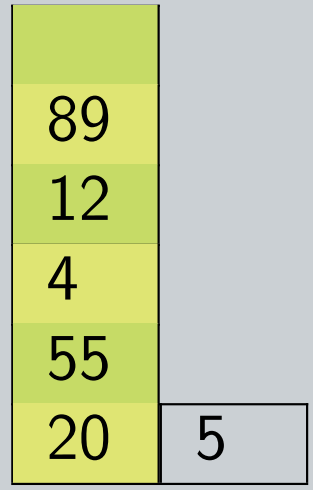
Método *adicionaNaPosicao(T dado, int posicao)*

- Testamos se há espaço e se a posição existe;
- Incrementamos o último;
- Empurramos tudo para trás a partir da posição;
- Adicionamos o novo dado na posição informada.



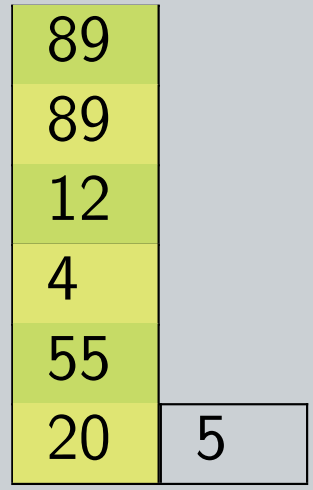
Método *adicionaNaPosicao(T dado, int posicao)*

- Testamos se há espaço e se a posição existe;
- Incrementamos o último;
- Empurramos tudo para trás a partir da posição;
- Adicionamos o novo dado na posição informada.



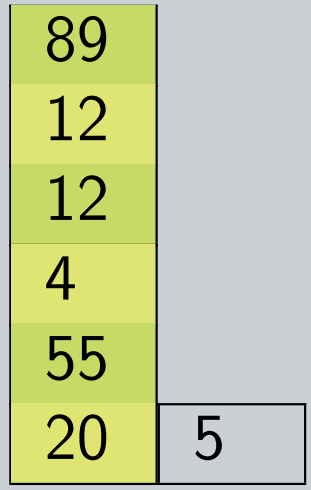
Método *adicionaNaPosicao(T dado, int posicao)*

- Testamos se há espaço e se a posição existe;
- Incrementamos o último;
- Empurramos tudo para trás a partir da posição;
- Adicionamos o novo dado na posição informada.



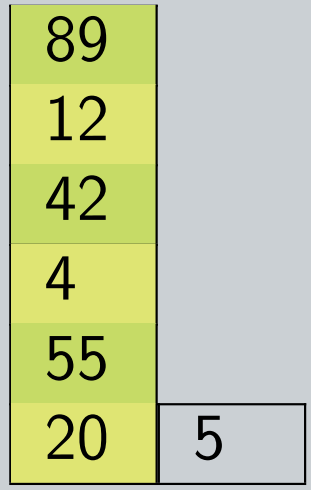
Método *adicionaNaPosicao(T dado, int posicao)*

- Testamos se há espaço e se a posição existe;
- Incrementamos o último;
- Empurramos tudo para trás a partir da posição;
- Adicionamos o novo dado na posição informada.



Método *adicionaNaPosicao(T dado, int posicao)*

- Testamos se há espaço e se a posição existe;
- Incrementamos o último;
- Empurramos tudo para trás a partir da posição;
- Adicionamos o novo dado na posição informada.



Método *adicionaNaPosicao(T dado, int posicao)*

```
adicionaNaPosicao(T dado, int posicao)
 int atual;
inicio
 SE (listaCheia) ENTAO
  THROW(ERROLISTACHEIA)
 SENAO
  SE (posicao > _ultimo+1 OU posicao < 0) ENTAO
   THROW(ERROPOSICAO);
  FIM SE
  _ultimo <- _ultimo+1; atual <- _ultimo;
  ENQUANTO (atual > posicao) FACA
   _dados[atual] <- _dados[atual - 1];
   atual <- atual - 1;
  FIM ENQUANTO
  _dados[posicao] <- dado;
 FIM SE
fim;
```

Método *adicionaNaPosicao(T dado, int posicao)*

```
adicionaNaPosicao(T dado, int posicao)
 int atual;
inicio
 SE (listaCheia) ENTAO
  THROW(ERROLISTACHEIA)
 SENAO
   SE (posicao > _ultimo+1 OU posicao < 0) ENTAO
    THROW(ERROPOSICAO);
   FIM SE
  _ultimo <- _ultimo+1; atual <- _ultimo;
  ENQUANTO (atual > posicao) FACA
   _dados[atual] <- _dados[atual - 1];
   atual <- atual - 1;
  FIM ENQUANTO
  _dados[posicao] <- dado;
 FIM SE
fim;
```

Método *adicionaNaPosicao(T dado, int posicao)*

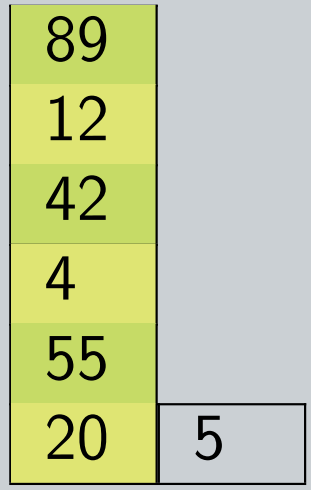
```
adicionaNaPosicao(T dado, int posicao)
 int atual;
inicio
 SE (listaCheia) ENTAO
  THROW(ERROLISTACHEIA)
 SENAO
  SE (posicao > _ultimo+1 OU posicao < 0) ENTAO
   THROW(ERROPOSICAO);
  FIM SE
  _ultimo <- _ultimo+1; atual <- _ultimo;
  ENQUANTO (atual > posicao) FACA
   _dados[atual] <- _dados[atual - 1];
   atual <- atual - 1;
  FIM ENQUANTO
   _dados[posicao] <- dado;
 FIM SE
fim;
```

Revisitando Métodos

- adicionaNoInicio(dado) == adicionaNaPosicao(dado,0);
- adiciona(dado) == adicionaNaPosicao(dado,ultimo+1);

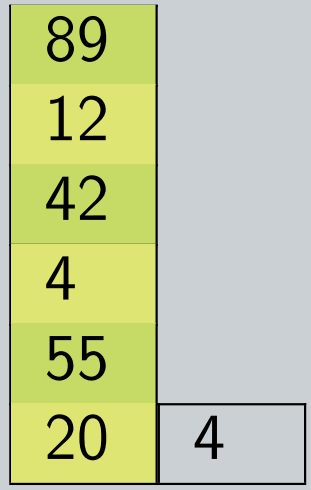
Método *T retiraDaPosicao(int posicao)*

- Testamos se há elementos e se a posição existe;
- Decrementamos o último;
- Salvamos elemento na posição;
- Empurramos tudo para frente até posição.



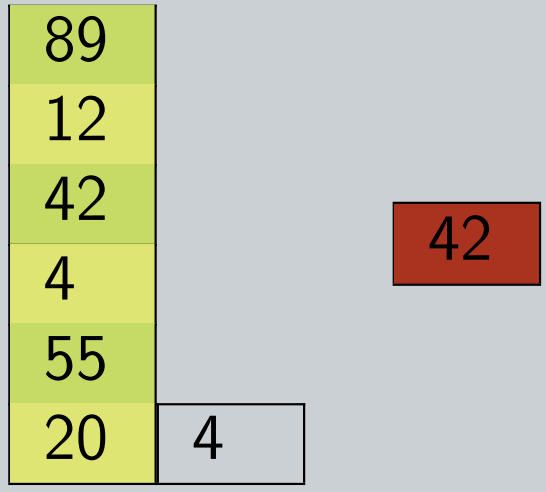
Método *T retiraDaPosicao(int posicao)*

- Testamos se há elementos e se a posição existe;
- Decrementamos o último;
- Salvamos elemento na posição;
- Empurramos tudo para frente até posição.



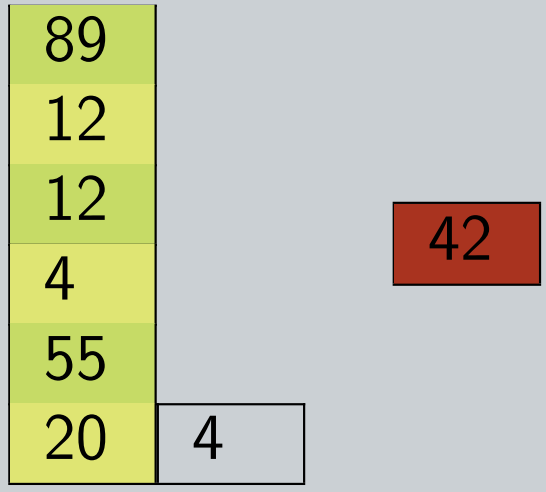
Método *T retiraDaPosicao(int posicao)*

- Testamos se há elementos e se a posição existe;
- Decrementamos o último;
- Salvamos elemento na posição;
- Empurramos tudo para frente até posição.



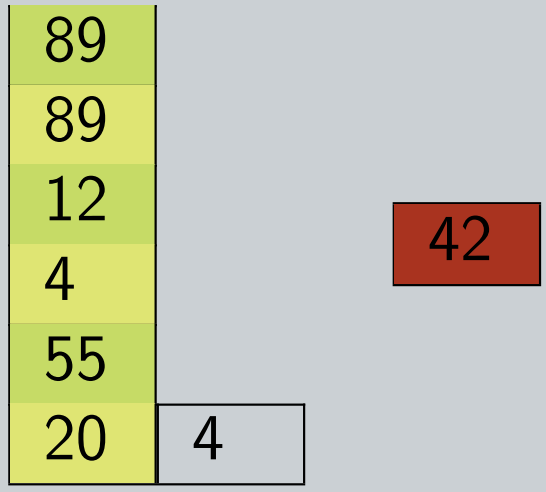
Método *T retiraDaPosicao(int posicao)*

- Testamos se há elementos e se a posição existe;
- Decrementamos o último;
- Salvamos elemento na posição;
- Empurramos tudo para frente até posição.



Método *T retiraDaPosicao(int posicao)*

- Testamos se há elementos e se a posição existe;
- Decrementamos o último;
- Salvamos elemento na posição;
- Empurramos tudo para frente até posição;
- Não perdemos tempo apagando o segundo ”89”, pois o _ultimo determina que a lista acaba antes!



Método *T retiraDaPosicao(int posicao)*

```
T retiraDaPosicao(int posicao)
 int atual; T valor;
inicio
 SE (posicao > _ultimo OU posicao < 0) ENTAO
  THROW(ERROPOSICAO)
 SENAO
  SE (listaVazia) ENTAO
   THROW(ERROLISTAVAZIA)
  SENAO
   _ultimo<-_ultimo - 1;
   valor<-_dados[posicao];
   atual <- posicao;
   ENQUANTO (atual <= _ultimo) FACA
    _dados[atual] <- _dados[atual + 1];
    atual <- atual + 1;
   FIM ENQUANTO
   RETORNE(valor);
  FIM SE
 FIM SE
fim;
```

Método *T retiraDaPosicao(int posicao)*

```
T retiraDaPosicao(int posicao)
 int atual; T valor;
inicio
  SE (posicao > _ultimo OU posicao < 0) ENTAO
   THROW(ERROPOSICAOO)
 SENAO
  SE (listaVazia) ENTAO
   THROW(ERROLISTAVAZIA)
  SENAO
   _ultimo<-_ultimo - 1;
   valor<-_dados[posicao];
   atual <- posicao;
   ENQUANTO (atual <= _ultimo) FACA
    _dados[atual] <- _dados[atual + 1];
    atual <- atual + 1;
   FIM ENQUANTO
   RETORNE(valor);
  FIM SE
 FIM SE
fim;
```

Método *T retiraDaPosicao(int posicao)*

```
T retiraDaPosicao(int posicao)
 int atual; T valor;
inicio
 SE (posicao > _ultimo OU posicao < 0) ENTAO
  THROW(ERROPOSICAO)
 SENAO
  SE (listaVazia) ENTAO
   THROW(ERROLISTAVAZIA)
  SENAO
   _ultimo<-_ultimo - 1;
    valor<-_dados[posicao];
   atual <- posicao;
   ENQUANTO (atual <= _ultimo) FACA
    _dados[atual] <- _dados[atual + 1];
    atual <- atual + 1;
   FIM ENQUANTO
   RETORNE(valor);
  FIM SE
 FIM SE
fim;
```

Método *T retiraDaPosicao(int posicao)*

```
T retiraDaPosicao(int posicao)
 int atual; T valor;
inicio
 SE (posicao > _ultimo OU posicao < 0) ENTAO
  THROW(ERROPOSICAO)
 SENAO
  SE (listaVazia) ENTAO
   THROW(ERROLISTAVAZIA)
  SENAO
   _ultimo<-_ultimo - 1;
   valor<-_dados[posicao];
   atual <- posicao;
    ENQUANTO (atual <= _ultimo) FACA
     _dados[atual] <- _dados[atual + 1];
     atual <- atual + 1;
    FIM ENQUANTO
   RETORNE(valor);
  FIM SE
 FIM SE
fim;
```

Método *T retiraDaPosicao(int posicao)*

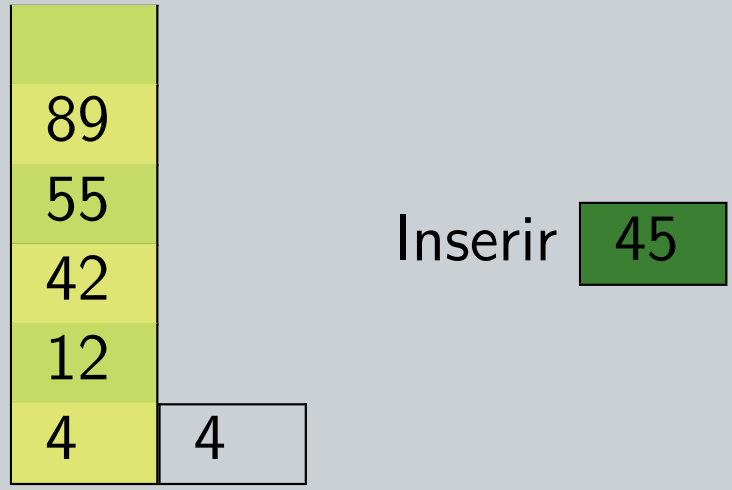
```
T retiraDaPosicao(int posicao)
 int atual; T valor;
inicio
 SE (posicao > _ultimo OU posicao < 0) ENTAO
  THROW(ERROPOSICAO)
 SENAO
  SE (listaVazia) ENTAO
   THROW(ERROLISTAVAZIA)
  SENAO
   _ultimo<-_ultimo - 1;
   valor<-_dados[posicao];
   atual <- posicao;
   ENQUANTO (atual <= _ultimo) FACA
    _dados[atual] <- _dados[atual + 1];
    atual <- atual + 1;
   FIM ENQUANTO
    RETORNE(valor);
  FIM SE
 FIM SE
fim;
```

Revisitando Métodos

- T retira() == retiraDaPosicao(_ultimo);
- T retiraDoInicio() == retiraDaPosicao(0);

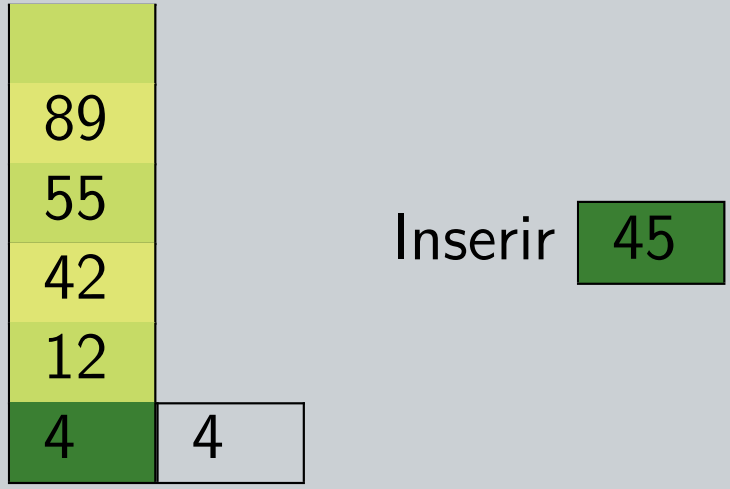
Método *adicionaEmOrdem(T dado)*

- Necessitamos de uma função para comparar os dados (próximo slide);
- Testamos se há espaço;
- Procuramos pela posição onde inserir comparando dados;
- Chamamos adicionaNaPosicao.



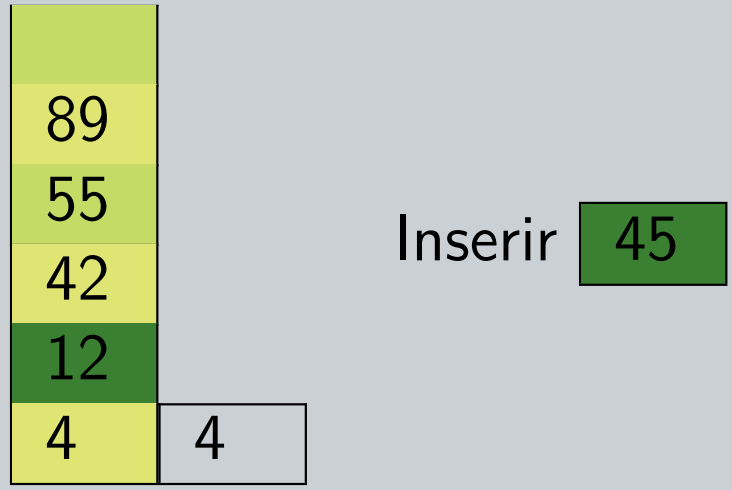
Método *adicionaEmOrdem(T dado)*

- Necessitamos de uma função para comparar os dados (próximo slide);
- Testamos se há espaço;
- Procuramos pela posição onde inserir comparando dados;
- Chamamos adicionaNaPosicao.



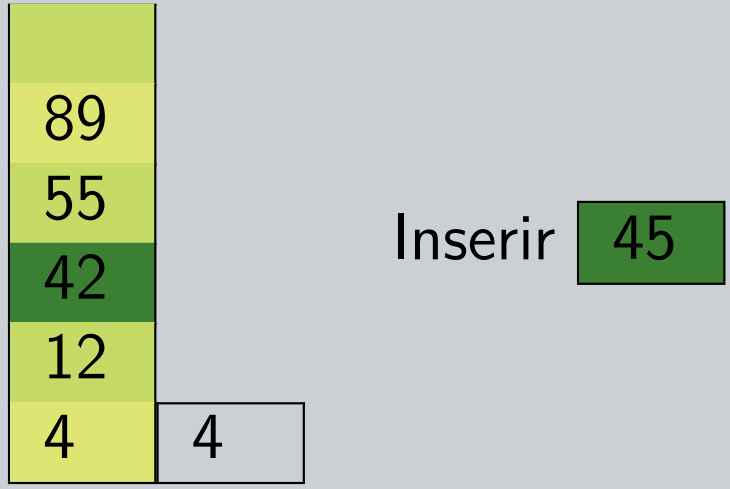
Método *adicionaEmOrdem(T dado)*

- Necessitamos de uma função para comparar os dados (próximo slide);
- Testamos se há espaço;
- Procuramos pela posição onde inserir comparando dados;
- Chamamos adicionaNaPosicao.



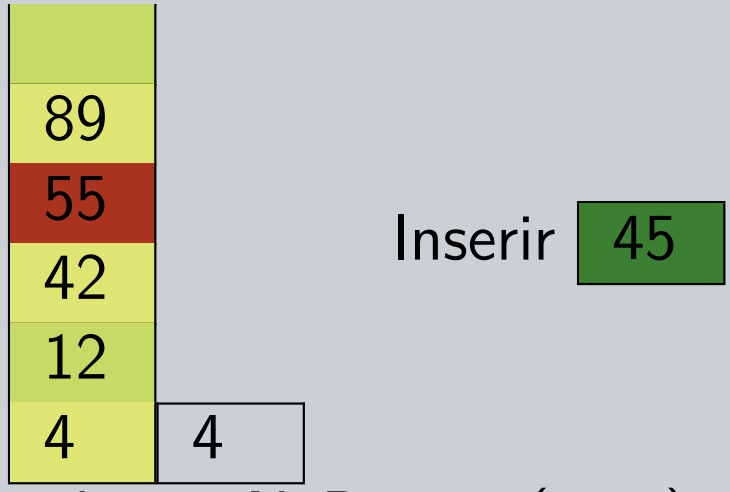
Método *adicionaEmOrdem(T dado)*

- Necessitamos de uma função para comparar os dados (próximo slide);
- Testamos se há espaço;
- Procuramos pela posição onde inserir comparando dados;
- Chamamos adicionaNaPosicao.



Método *adicionaEmOrdem(T dado)*

- Necessitamos de uma função para comparar os dados (próximo slide);
- Testamos se há espaço;
- Procuramos pela posição onde inserir comparando dados;
- Chamamos adicionaNaPosicao.



Método *T::operator>(T t)*

- Quando o dado a ser armazenado em uma lista não for um tipo primitivo, este tem que implementar a sobrecarga do operador “>”;
- Para deixar os algoritmos de operações sobre lista independentes do tipo de dado específico armazenado na lista, usamos uma função do tipo T::operator>(T t);
- Isso não deve ser implementado pelo programador da lista, mas pelo programador dos objetos que serão colocados na lista;

Método *adicionaEmOrdem(T dado)*

```
adicionaEmOrdem(T dado)
 int atual;
inicio
 SE (listaCheia) ENTAO
  THROW(ERROLISTACHEIA)
 SENAO
  atual <- 0;
  ENQUANTO (atual <= _ultimo E
   (dado > _dados[atual])) FACA
   //Encontrar posicao para inserir
   atual <- atual + 1;
  FIM ENQUANTO
  RETORNE(adicionaNaPosicao(dado, atual));
 FIM SE
fim;
```

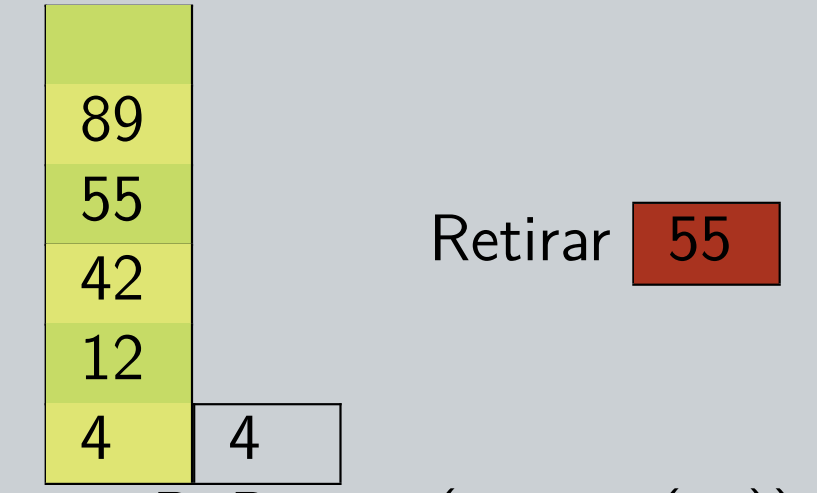
Método *adicionaEmOrdem(T dado)*

```
adicionaEmOrdem(T dado)
 int atual;
inicio
 SE (listaCheia) ENTAO
  THROW(ERROLISTACHEIA)
 SENAO
  atual <- 0;
  ENQUANTO (atual <= _ultimo E
   (dado > _dados[atual])) FACA
   //Encontrar posicao para inserir
   atual <- atual + 1;
  FIM ENQUANTO
  RETORNE(adicionaNaPosicao(dado, atual));
 FIM SE
fim;
```

- Por conta do Aluno: Essa implementação pode ser melhorada usando a técnica “dividir-para-conquistar”;

Método *T retiraEspecifico(dado)*

- Testamos se há elementos;
- Testamos se o dado existe e qual sua posição;
- Necessitamos de uma função posicao(dado);
- Chamamos retiraDaPosicao(posicao(dado));



Método *int posicao(T dado)*

```
posicao(T dado)
 int atual;
inicio
 atual <- 0;
 ENQUANTO (atual <= _ultimo E
   NAO(IGUAL(dado, _dados[atual]))) FACA
atual <- atual + 1;
FIM ENQUANTO
SE (atual > _ultimo) ENTAO
  THROW(ERROPOSICAO)
SENAO
  RETORNE(atual);
FIM SE
fim;
```

Algoritmos Restantes

- As seguintes funções ficam por conta do aluno:
  - T retiraEspecifico(T dado);
  - bool contem(dado);
- Não esqueça da sobrecarga de operadores nos tipos não primitivos:
  - T::operator=(T t);
  - T::operator<(T t).

Trabalho Lista de Vetor

- 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