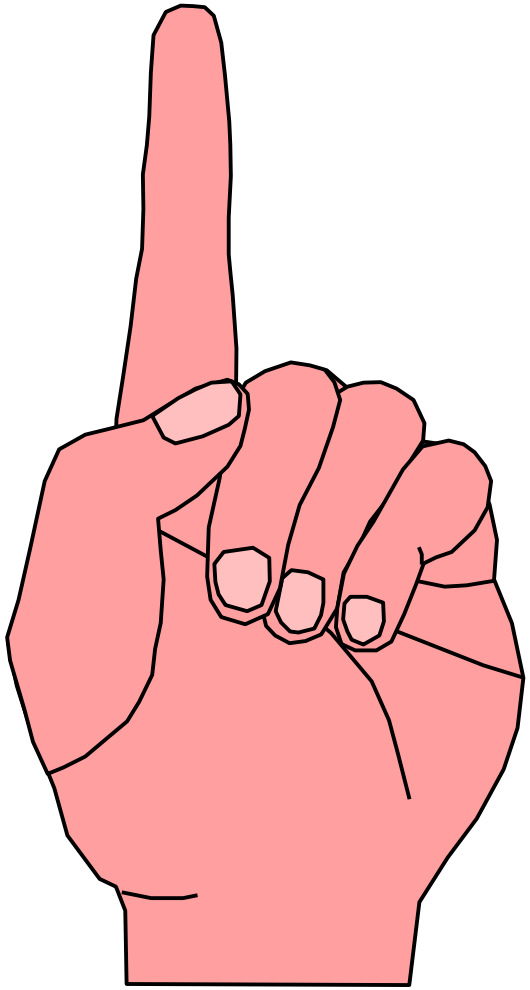
Estruturas de Dados

Lista utilizando Vetores

Lista - Definição

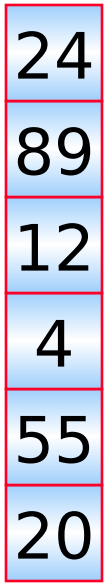


- Uma Lista é um conjunto de dados dispostos e/ou acessáveis em uma sequência determinada.
  - Este conjunto de dados pode possuir uma ordem intrínseca (Lista Ordenada) ou não;
  - Este conjunto de dados pode ocupar espaços de memória fisicamente consecutivos, espelhando a sua ordem, ou não;
  - Se os 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).

Vimos dois aspectos

- Em um projeto de software, dois aspectos devem ser considerados:
  - de que forma estão organizados os dados, qual a sua estrutura;
  - quais procedimentos atuam sobre estes dados, que operações podem ser realizadas sobre eles.
- Vamos ver agora estes aspectos para as **listas**.

Listas

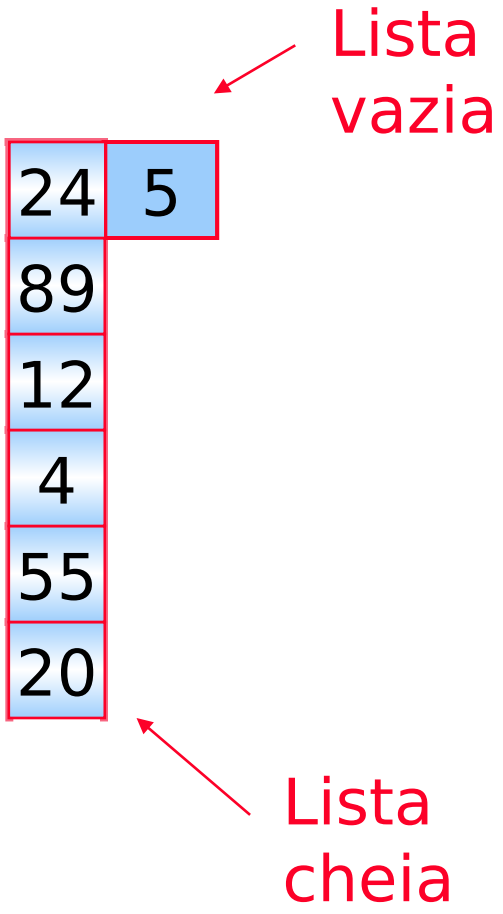


A Lista é uma estrutura de dados cujo funcionamento é inspirado no de uma lista “natural”.

Uma lista pode ser ordenada ou não:

- quando for ordenada, pode o ser por alguma característica intrínseca dos dados (ex: ordem alfabética);
- pode também refletir a ordem cronológica (ordem de inserção) dos dados;
- em Smalltalk, a **OrderedCollection** é uma lista cronológica (**add:**), a **SortedCollection** é uma lista ordenada por um critério interno.

Listas usando Vetores



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

Modelagem de Listas

- Modelaremos a Estrutura de Dados Lista utilizando a técnica da Programação Orientada a Objeto, armazenando os dados em um Vetor (Array).
  - Veremos somente algoritmos;
  - veremos algoritmos tanto para uma lista cronológica quanto para uma lista ordenada.
- Procedimento Didático:
  - modelagem da Lista e de seus algoritmos usando Orientação a Objetos;
  - exercícios.

Modelagem da Lista

- Aspecto Estrutural:
  - necessitamos de um vetor para armazenar as informações;
  - necessitamos de um indicador da posição atual do último elemento da lista;
  - necessitamos de uma constante que nos diga quando a lista está cheia e duas outras para codificar erros.

Pseudo-código:

```
constantes MAXLISTA = 100;


classe Lista {
T dados[MAXLISTA];
inteiro último;
};
```

Modelagem da Lista

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

Modelagem da Lista

- Operações: adicionar e retirar dados da lista
  - Adiciona(dado)
  - AdicionaNoInício(dado)
  - AdicionaNaPosição(dado, posição)
  - AdicionaEmOrdem(dado)
  - Retira()
  - RetiraDoInício()
  - RetiraDaPosição(posição)
  - RetiraEspecífico(dado)

Modelagem da Lista

- Operações: testar a lista
  - ListaCheia
  - ListaVazia
  - Posição(dado)
  - Contém(dado)
  - Igual(dado1, dado2)
  - Maior(dado1, dado2)
  - Menor(dado1, dado2)

Modelagem da Lista

- Operações: inicializar ou limpar a lista
  - InicializaLista
  - DestróiLista

Construtor Lista

```
Lista()
  início
     // T = new T[MAXLISTA];
     último <- -1;
  fim;
```

Algoritmo DestróiLista

```
MÉTODO destróiLista()
   início
      Último <- -1;
      // delete dados;
      // dados = new T[MAXLISTA];
   fim;
```

Observação: este algoritmo parece redundante. Colocar a sua semântica em separado da inicialização, porém, é importante. Mais tarde veremos que, utilizando alocação dinâmica de memória, possuir um destrutor para Lista é muito importante .

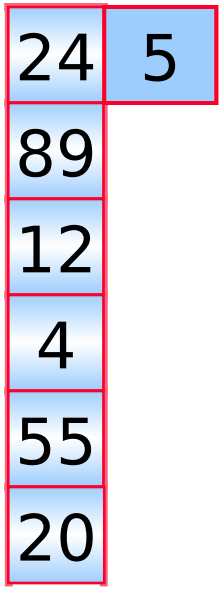
Algoritmo ListaCheia

```
Booleano MÉTODO listaCheia()
   início
      SE (último = MAXLISTA - 1) ENTÃO
         RETORNE(Verdadeiro)
      SENÃO
         RETORNE(Falso);
   fim;
```

Algoritmo ListaVazia

```
Booleano MÉTODO listaVazia()
  início
     SE (último = -1) ENTÃO
        RETORNE(Verdadeiro)
     SENÃO
        RETORNE(Falso);
  fim;
```

Algoritmo Adiciona



- Procedimento:
  - testamos se há espaço;
  - incrementamos o último;
  - adicionamos o novo dado.
- Parâmetros:
  - o dado a ser inserido;

Algoritmo Adiciona

```
Constantes
   ERROLISTACHEIA = -1;
   ERROLISTAVAZIA = -2;
   ERROPOSIÇÃO = -3;

Inteiro MÉTODO adiciona(T dado)
   início
      SE (listaCheia) ENTÃO
         THROW(ERROLISTACHEIA)
      SENÃO
         último <- último + 1;
         dados[último] <- dado;
         RETORNE(último);
      FIM SE
   fim;
```

Algoritmo Retira

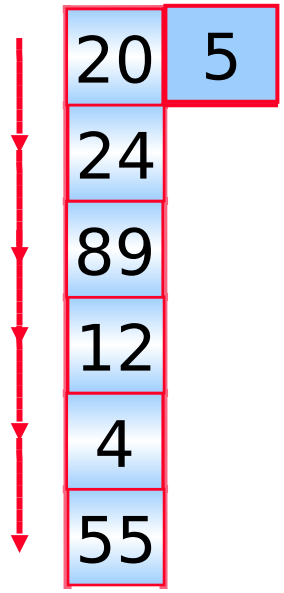
- Procedimento:
  - testamos se há elementos;
  - decrementamos o último;
  - devolvemos o último elemento.

Algoritmo Retira

```
T MÉTODO retira()
  início
     SE (listaVazia) ENTÃO
        THROW(ERROLISTAVAZIA)
     SENÃO
        último <- último – 1;
        RETORNE(dados[último + 1]);
     FIM SE
  fim;
```

Observação: note que aqui se aplicam as mesmas restrições das diversas variantes do algoritmo **`desempilha`**.

Algoritmo AdicionaNoInício



- Procedimento:
  - testamos se há espaço;
  - incrementamos o último;
  - empurramos tudo para trás;
  - adicionamos o novo dado na primeira posição.
- Parâmetros:
  - o dado a ser inserido;

Algoritmo AdicionaNoInício

```
Inteiro MÉTODO adicionaNoInício(T dado)
   variáveis
       inteiro posição; //Variável auxiliar para "caminhar"
   início
       SE (listaCheia) ENTÃO
           THROW(ERROLISTACHEIA)
       SENÃO
           último <- último + 1;
           posição <- último;
           ENQUANTO (posição > 0) FAÇA
               //Empurrar tudo para trás
               dados[posição] <- dados[posição - 1];
               posição <- posição - 1;
           FIM ENQUANTO
           dados[0] <- dado;
           RETORNE(0);
       FIM SE
   fim;
```

Algoritmo AdicionaNoInício

```
Inteiro MÉTODO adicionaNoInício(T dado)
   variáveis
       inteiro posição; //Variável auxiliar para "caminhar"
   início
       SE (listaCheia) ENTÃO
           THROW(ERROLISTACHEIA)
       SENÃO
           último <- último + 1;
           posição <- último;
           ENQUANTO (posição > 0) FAÇA
               //Empurrar tudo para trás
               dados[posição] <- dados[posição - 1];
               posição <- posição - 1;
           FIM ENQUANTO
           dados[0] <- dado;
           RETORNE(0);
       FIM SE
   fim;
```

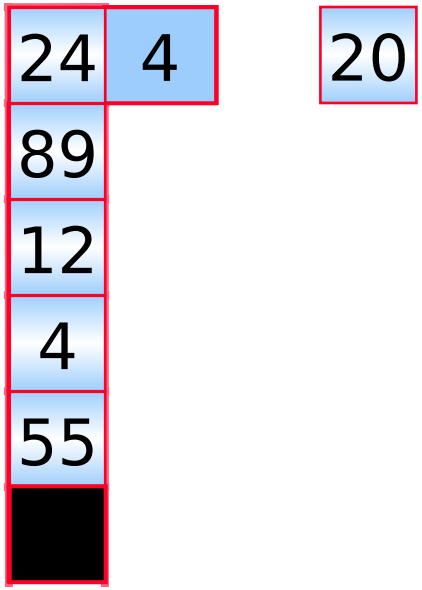
Algoritmo AdicionaNoInício

```
Inteiro MÉTODO adicionaNoInício(T dado)
   variáveis
       inteiro posição; //Variável auxiliar para "caminhar"
   início
       SE (listaCheia) ENTÃO
           THROW(ERROLISTACHEIA)
       SENÃO
           último <- último + 1;
           posição <- último;
           ENQUANTO (posição > 0) FAÇA
               //Empurrar tudo para trás
               dados[posição] <- dados[posição - 1];
               posição <- posição - 1;
           FIM ENQUANTO
           dados[0] <- dado;
           RETORNE(0);
       FIM SE
   fim;
```

Algoritmo AdicionaNoInício

```
Inteiro MÉTODO adicionaNoInício(T dado)
   variáveis
       inteiro posição; //Variável auxiliar para "caminhar"
   início
       SE (listaCheia) ENTÃO
           THROW(ERROLISTACHEIA)
       SENÃO
           último <- último + 1;
           posição <- último;
           ENQUANTO (posição > 0) FAÇA
               //Empurrar tudo para trás
               dados[posição] <- dados[posição - 1];
               posição <- posição - 1;
           FIM ENQUANTO
           dados[0] <- dado;
           RETORNE(0);
       FIM SE
   fim;
```

Algoritmo RetiraDoInício



- Procedimento:
  - testamos se há elementos;
  - decrementamos o último;
  - salvamos o primeiro elemento;
  - empurramos tudo para a frente.

Algoritmo RetiraDoInício

```
T MÉTODO retiraDoInício()
   variáveis
       inteiro posição, valor;
   início
       SE (listaVazia) ENTÃO
           THROW(ERROLISTAVAZIA)
       SENÃO
           último <- último - 1;
           valor <- dados[0];
           posição <- 0;
           ENQUANTO (posição <= último) FAÇA
               //Empurrar tudo para a frente
               dados[posição] <-  dados[posição + 1];
               posição <- posição + 1;
           FIM ENQUANTO
           RETORNE(valor);
       FIM SE
   fim;
```

Algoritmo RetiraDoInício

```
T MÉTODO retiraDoInício()
   variáveis
       inteiro posição, valor;
   início
       SE (listaVazia) ENTÃO
           THROW(ERROLISTAVAZIA)
       SENÃO
           último <- último - 1;
           valor <- dados[0];
           posição <- 0;
           ENQUANTO (posição <= último) FAÇA
               //Empurrar tudo para a frente
               dados[posição] <- dados[posição + 1];
               posição <- posição + 1;
           FIM ENQUANTO
           RETORNE(valor);
       FIM SE
   fim;
```

Algoritmo RetiraDoInício

```
T MÉTODO retiraDoInício()
   variáveis
       inteiro posição, valor;
   início
       SE (listaVazia) ENTÃO
           THROW(ERROLISTAVAZIA)
       SENÃO
           último <- último - 1;
           valor <- dados[0];
           posição <- 0;
           ENQUANTO (posição <= último) FAÇA
               //Empurrar tudo para a frente
               dados[posição] <- dados[posição + 1];
               posição <- posição + 1;
           FIM ENQUANTO
           RETORNE(valor);
       FIM SE
   fim;
```

Algoritmo RetiraDoInício

```
T MÉTODO retiraDoInício()
   variáveis
       inteiro posição, valor;
   início
       SE (listaVazia) ENTÃO
           THROW(ERROLISTAVAZIA)
       SENÃO
           último <- último - 1;
           valor <- dados[0];
           posição <- 0;
           ENQUANTO (posição <= último) FAÇA
               //Empurrar tudo para a frente
               dados[posição] <- dados[posição + 1];
               posição <- posição + 1;
           FIM ENQUANTO
           RETORNE(valor);
       FIM SE
   fim;
```

Algoritmo AdicionaNaPosição

- Procedimento:
  - 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.
- Parâmetros:
  - o dado a ser inserido;
  - a posição onde inserir;

Algoritmo AdicionaNaPosição

```
Inteiro MÉTODO adicionaNaPosição(T dado, inteiro destino)
   variáveis
       inteiro posição;
   início
       SE (listaCheia) ENTÃO
           THROW(ERROLISTACHEIA)
       SENÃO
           SE (destino > último + 1 OU destino < 0) ENTÃO
               THROW(ERROPOSIÇÃO);
           FIM SE
           último <- último + 1;
           posição <- último;
           ENQUANTO (posição > destino) FAÇA
               //Empurrar tudo para trás
               dados[posição] <- dados[posição - 1];
               posição <- posição - 1;
           FIM ENQUANTO
           dados[destino] <- dado;
           RETORNE(destino);
       FIM SE
   fim;
```

Algoritmo AdicionaNaPosição

```
Inteiro MÉTODO adicionaNaPosição(T dado, inteiro destino)
   variáveis
       inteiro posição;
   início
       SE (listaCheia) ENTÃO
           THROW(ERROLISTACHEIA)
       SENÃO
           SE (destino > último + 1 OU destino < 0) ENTÃO
               THROW(ERROPOSIÇÃO);
           FIM SE
           último <- último + 1;
           posição <- último;
           ENQUANTO (posição > destino) FAÇA
               //Empurrar tudo para trás
               dados[posição] <- dados[posição - 1];
               posição <- posição - 1;
           FIM ENQUANTO
           dados[destino] <- dado;
           RETORNE(destino);
       FIM SE
   fim;
```

Algoritmo AdicionaNaPosição

```
Inteiro MÉTODO adicionaNaPosição(T dado, inteiro destino)
   variáveis
       inteiro posição;
   início
       SE (listaCheia) ENTÃO
           THROW(ERROLISTACHEIA)
       SENÃO
           SE (destino > último + 1 OU destino < 0) ENTÃO
               THROW(ERROPOSIÇÃO);
           FIM SE
           último <- último + 1;
           posição <- último;
           ENQUANTO (posição > destino) FAÇA
               //Empurrar tudo para trás
               dados[posição] <- dados[posição - 1];
               posição <- posição - 1;
           FIM ENQUANTO
           dados[destino] <- dado;
           RETORNE(destino);
       FIM SE
   fim;
```

Algoritmo AdicionaNaPosição

```
Inteiro MÉTODO adicionaNaPosição(T dado, inteiro destino)
   variáveis
       inteiro posição;
   início
       SE (listaCheia) ENTÃO
           THROW(ERROLISTACHEIA)
       SENÃO
           SE (destino > último + 1 OU destino < 0) ENTÃO
               THROW(ERROPOSIÇÃO);
           FIM SE
           último <- último + 1;
           posição <- último;
           ENQUANTO (posição > destino) FAÇA
               //Empurrar tudo para trás
               dados[posição] <- dados[posição - 1];
               posição <- posição - 1;
           FIM ENQUANTO
           dados[destino] <- dado;
           RETORNE(destino);
       FIM SE
   fim;
```

Revisitando métodos

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

Algoritmo RetiraDaPosição

- Procedimento:
  - testamos se há elementos e se a posição existe;
  - decrementamos o último;
  - salvamos elemento na posição;
  - empurramos tudo para frente até posição.
- Parâmetros:
  - o dado a ser inserido;
  - a posição onde inserir;

Algoritmo RetiraDaPosição

```
T MÉTODO retiraDaPosição(inteiro fonte)
   variáveis
       inteiro posição, valor;
   início
       SE (fonte > último OU fonte < 0) ENTÃO
           THROW(ERROPOSIÇÃO)
       SENÃO
           SE (listaVazia) ENTÃO
               THROW(ERROLISTAVAZIA)
           SENÃO
               último <- último - 1;
               valor <- dados[fonte];
               posição <- fonte;
               ENQUANTO (posição <= último) FAÇA
                   //Empurrar tudo para frente
                   dados[posição] <- dados[posição + 1];
                   posição <- posição + 1;
               FIM ENQUANTO
               RETORNE(valor);
           FIM SE
       FIM SE
   fim;
```

Algoritmo RetiraDaPosição

```
T MÉTODO retiraDaPosição(inteiro fonte)
   variáveis
       inteiro posição, valor;
   início
       SE (fonte > último OU fonte < 0) ENTÃO
           THROW(ERROPOSICÃO)
       SENÃO
           SE (listaVazia) ENTÃO
               THROW(ERROLISTAVAZIA)
           SENÃO
               último <- último - 1;
               valor <- dados[fonte];
               posição <- fonte;
               ENQUANTO (posição <= último) FAÇA
                   //Empurrar tudo para frente
                   dados[posição] <- dados[posição + 1];
                   posição <- posição + 1;
               FIM ENQUANTO
               RETORNE(valor);
           FIM SE
       FIM SE
   fim;
```

Algoritmo RetiraDaPosição

```
T MÉTODO retiraDaPosição(inteiro fonte)
   variáveis
        inteiro posição, valor;
   início
        SE (fonte > último OU fonte < 0) ENTÃO
             THROW(ERROPOSIÇÃO)
        SENÃO
             SE (listaVazia) ENTÃO
                  THROW(ERROLISTAVAZIA)
             SENÃO
                  último <- último - 1;
                  valor <- dados[fonte];
                  posição <- fonte;
                  ENQUANTO (posição <= último) FAÇA
                       //Empurrar tudo para frente
                       dados[posição] <- dados[posição + 1];
                       posição <- posição + 1;
                  FIM ENQUANTO
                  RETORNE(valor);
             FIM SE
        FIM SE
   fim;
```

Algoritmo RetiraDaPosição

```
T MÉTODO retiraDaPosição(inteiro fonte)
   variáveis
        inteiro posição, valor;
   início
        SE (fonte > último OU fonte < 0) ENTÃO
             THROW(ERROPOSIÇÃO)
        SENÃO
             SE (listaVazia) ENTÃO
                  THROW(ERROLISTAVAZIA)
             SENÃO
                  último <- último - 1;
                  valor <- dados[fonte];
                  posição <- fonte;
                  ENQUANTO (posição <= último) FAÇA
                       //Empurrar tudo para frente
                       dados[posição] <- dados[posição + 1];
                       posição <- posição + 1;
                  FIM ENQUANTO
                  RETORNE(valor);
             FIM SE
        FIM SE
   fim;
```

Algoritmo RetiraDaPosição

```
T MÉTODO retiraDaPosição(inteiro fonte)
   variáveis
       inteiro posição, valor;
   início
       SE (listaVazia) ENTÃO
           THROW(ERROLISTAVAZIA)
       SENÃO
                 SE (fonte > último OU fonte < 0) ENTÃO
                       THROW(ERROPOSIÇÃO)
           SENÃO
               último <- último - 1;
               valor <- dados[fonte];
               posição <- fonte;
               ENQUANTO (posição <= último) FAÇA
                   //Empurrar tudo para frente
                   dados[posição] <- dados[posição + 1];
                   posição <- posição + 1;
               FIM ENQUANTO
               RETORNE(valor);
           FIM SE
       FIM SE
   fim;
```

Algoritmo AdicionaEmOrdem

- Procedimento:
  - necessitamos de uma função para comparar os dados (maior);
  - testamos se há espaço;
  - procuramos pela posição onde inserir comparando dados;
  - chamamos **`adicionaNaPosição`**.
- Parâmetros:
  - o dado a ser inserido;

Algoritmo Maior

```
Booleano MÉTODO maior(T dado1, T dado2)
  início
      SE (dado1 > dado2) ENTÃO
          RETORNE(Verdadeiro)
      SENÃO
          RETORNE(Falso);
      FIM SE
  fim;
```

Observação: quando o dado a ser armazenado em uma lista for algo mais complexo do que um inteiro, a comparação de precedência não será mais tão simples (ex.: Empregado1 > Empregado2) e será resultado de um conjunto mais complexo de operações.

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)

Algoritmo AdicionaEmOrdem

```
Inteiro MÉTODO adicionaEmOrdem(T dado)
   variáveis
       inteiro posição; //Variável auxiliar para "caminhar"
   início
       SE (listaCheia) ENTÃO
           THROW(ERROLISTACHEIA)
       SENÃO
           posição <- 0;
           ENQUANTO (posição <= último E
               (dado > dados[posição])) FAÇA
               //Encontrar posição para inserir
               posição <- posição + 1;
           FIM ENQUANTO
           RETORNE(adicionaNaPosição(dado, posição));
       FIM SE
   fim;
```

Algoritmo RetiraEspecífico

- Retira um dado específico da lista.
- Procedimento:
  - testamos se há elementos;
  - testamos se o dado existe e qual sua posição;
  - necessitamos de uma função **`Posição(dado);`**
  - chamamos **`RetiraDaPosição`**.
- Parâmetros:
  - o dado a ser retirado;

Algoritmo Posição

```
Inteiro MÉTODO posição(T dado)
   variáveis
      inteiro posição;
   início
      posição <- 0;
      ENQUANTO (posição <= último E
         NÃO(IGUAL(dado, dados[posição]))) FAÇA
         posição <- posição + 1;
      FIM ENQUANTO
      SE (posição > último) ENTÃO
        THROW(ERROPOSIÇÃO)
      SENÃO
        RETORNE(posição);
      FIM SE
   fim;
```

Algoritmo RetiraEspecífico

```
T MÉTODO retiraEspecífico(T dado)
   variáveis
       inteiro posição;
  início
       SE (listaVazia) ENTÃO
           THROW(ERROLISTAVAZIA)
       SENÃO
           posição <- posição(dado);
           SE (posição < 0) ENTÃO
               THROW(ERROPOSIÇÃO)
           SENÃO
               RETORNE(retiraDaPosição(posição));
           FIM SE
       FIM SE
   fim;
```

Algoritmos Restantes

- As seguintes funções ficam por conta do aluno :
  - booleano Contém(dado)
    - Baseado no Posicao(dado)
  - booleano Igual(dado1, dado2)
  - booleano Menor(dado1, dado2)
- Não esqueça da sobrecarga de operadores:
  - T::operator=(T t)
  - T::operator<(T t)

Exercício: Trabalho de 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.