#### Listas Lineares

**Preâmbulo: um problema a ser resolvido**

Você deve implementar uma solução para a seguinte situação: manter uma ***lista*** de **alunos** de uma turma de ED, cada aluno descrito pelo **nome**, **nº da matrícula** e a **nota**. Novos alunos serão inseridos sempre no final da ***lista***. Alunos serão excluídos da ***lista*** pelo nome. A aplicação da solução deve manter um **menu**, que permita ao usuário interagir continuamente com a lista, por meio das funcionalidades:

* Opção 1 – Inserir aluno: esta funcionalidade lê, pelo teclado, os dados do aluno (nome, matricula e nota) e insere este no final da lista.
* Opção 2 – Excluir aluno: esta funcionalidade obtém do teclado o nome do aluno a excluir, pesquisa e encontrando o aluno, o exclui da lista. Não encontrando, apresenta mensagem: “Aluno não cadastrado!”.
* Opção 3 – Mostrar listagem completa de alunos: esta funcionalidade mostra todos os dados dos alunos inseridos na lista.
* Alem das funcionalidades acima, o menu deve oferecer uma opção para encerrar a execução do aplicativo.

Você está livre para tomar decisões e adotar a solução de implementação que julgar mais adequada e conveniente. Portanto não é necessário pedir permissões ao professor sobre eventuais estratégias de implementação. Fique à vontade!

Mas em qualquer caso, não hesite em consultar o professor.

**Introdução**

O conceito de *Lista* é bem conhecido e sua abstração é intuitiva e imediata para qualquer um. Uma lista representa um conjunto de “coisas” que temos por fazer ou para guardar e consultar/manipular depois. São exemplos clássicos de listas: um catálogo telefônico, uma lista de compras no supermercado, uma lista de convidados para um churrasco, entre tantos outros. Nota-se portanto que uma lista constitui uma forma de armazenar informações a serem manipuladas para atender a algum fim ou objetivo.

Formalmente, **uma lista é uma sequência de zero ou mais elementos de um determinado tipo, interligados de uma forma pré-definida, formando uma estrutura linear ou unidimensional**. Essa definição é derivada da Teoria de Conjuntos e pode ser resumida como:

**Lista = { X 1 , X 2 , . . . , X i , . . . , X n }**

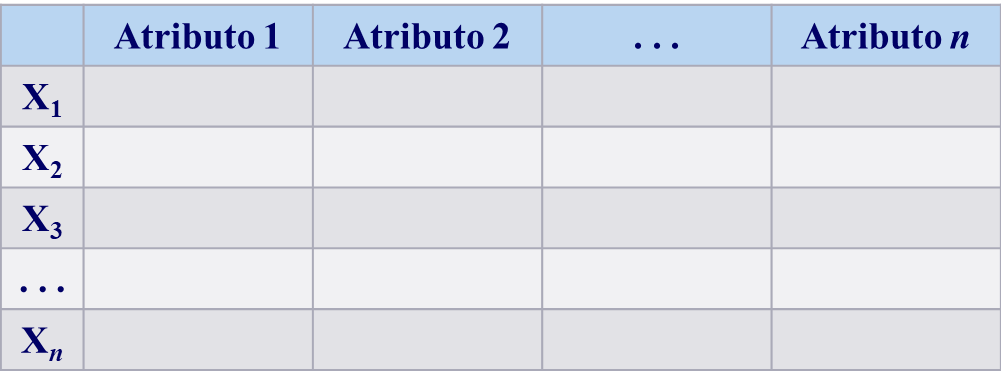
Por essa definição, não há qualquer restrição se os elementos da lista estejam ou não ordenados.

Listas constituem estruturas bastantes flexíveis porque:

* Podem crescer ou diminuir de tamanho de acordo com a demanda;
* Itens podem ser acessados, inseridos ou retirados de qualquer posição da lista;
* Duas ou mais listas podem ser concatenadas para formar uma única lista;
* Uma lista pode ser particionada em duas ou mais listas.

**O Tipo Abstrato de Dado *Lista* (TAD Lista)**

Pode-se abstrair a ***estrutura física*** de uma lista como uma TABELA, conforme a figura abaixo:

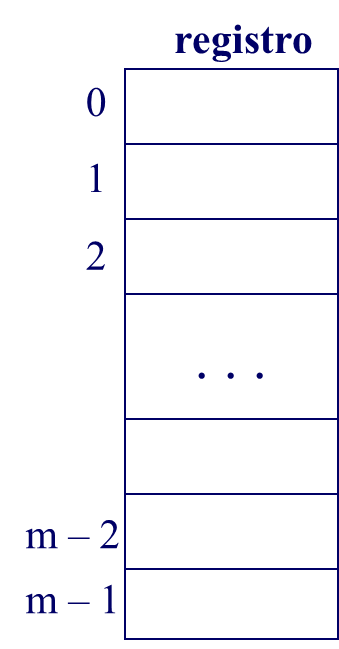


Observa-se pela figura que as colunas da tabela representam uma entidade qualquer do universo de discurso, descrita por ***n*** atributos. Ou seja, o conjunto de colunas pode ser perfeitamente representado por um ***registro*** onde cada campo descreve um atributo (ou coluna) Por sua vez, as linhas da tabela representam cada ocorrência, ***Xi***, da entidade considerada. Deste modo, um ***vetor*** é uma estrutura adequada para armazenar o conjunto de linhas.

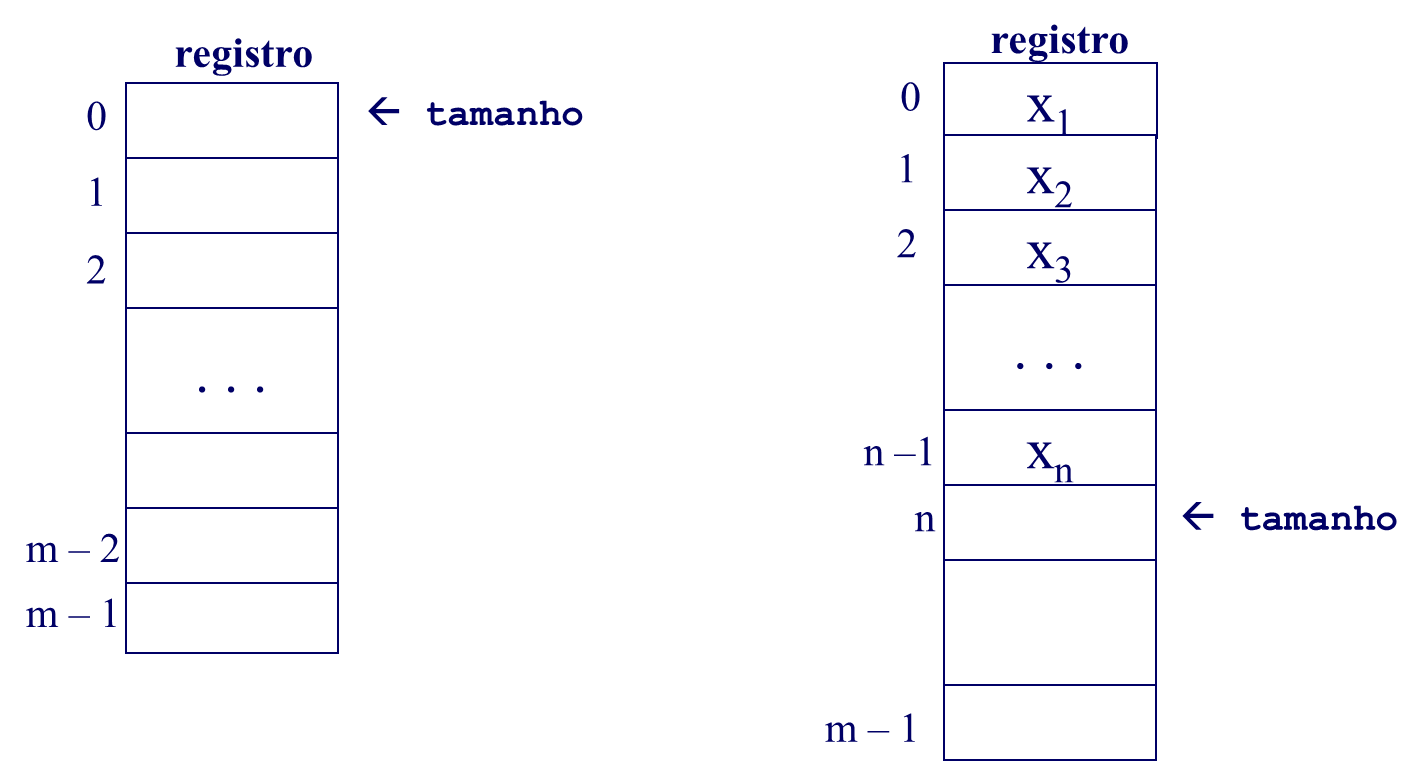
Por outro lado, é necessário criar um conjunto de ***operações*** sobre a estrutura da lista. Estas operações dependem de cada aplicação, não existindo um conjunto que seja adequado a todas as aplicações. Operações típicas são: criar uma lista; destruir uma lista; inserir um elemento na lista, excluir um elemento da lista, pesquisar um elemento da lista, classificar a lista (ordenar de forma ascendente ou descendente de algum de seus atributos), etc..

**Implementação do *TAD Lista* com *Arranjo***

A estrutura da lista abstraída como uma tabela pode ser implementada de imediato como um ***arranjo de registro*** ou ***vetor de registro***. De fato, o conjunto de colunas (ou atributos) pode ser implementado como um ***registro*** que descreve o tipo de entidade a ser manipulado na lista. E as linhas podem ser armazenadas em um ***arranjo*** com ***m*** elementos, indexados de ***0..m–1***, conforme ilustrado a seguir:



Porém, uma premissa essencial para a boa organização da lista é que **os elementos devem estar sempre compactados no início do vetor**, isto é, não pode haver elementos dispersos no vetor, com a ocorrência de posições vazias entre elementos. Deste modo, é necessário e fundamental a presença de uma marca que defina o tamanho da lista, ou seja , a porção do vetor ocupada por elementos da lista num dado instante. A figura a seguir ilustra esta situação.



A imagem à esquerda ilustra a situação de ***lista vazia***, isto é, sem nenhum elemento. Neste caso, a marca do ***tamanho*** está pertinentemente em **0**. Na imagem à direita está ilustrado o caso em que ***n*** elementos estão inseridos na lista. Estes elementos estão compactados nas posições ***0 .. n–1*** e a marca do ***tamanho*** está pertinentemente na posição ***n*** do arranjo. Conclui-se que nesta abordagem, ***tamanho*** marca sempre a primeira posição vazia (ou disponível) do arranjo.

Traduzindo a estrutura da imagem acima para uma linguagem computacional, tem-se:

|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| const MAX = 1000  Dados: **Registro**  ***// campos da estrutura de dados da Lista***  **FimRegistro**  Lista: **Registro**  lst: vetor [MAX] de Dados  tamanho: inteiro  **FimRegistro** | **#define** MAX 1000  **typedef struct {**  ***// campos da estrutura de dados da Lista***  **} Dados;**  **typedef struct {**  **Dados** lst[MAX];  **int** tamanho;  **} Lista;** |

Complementando a implementação do TAD Lista, faz-se necessário descrever as principais operações sobre a estrutura:

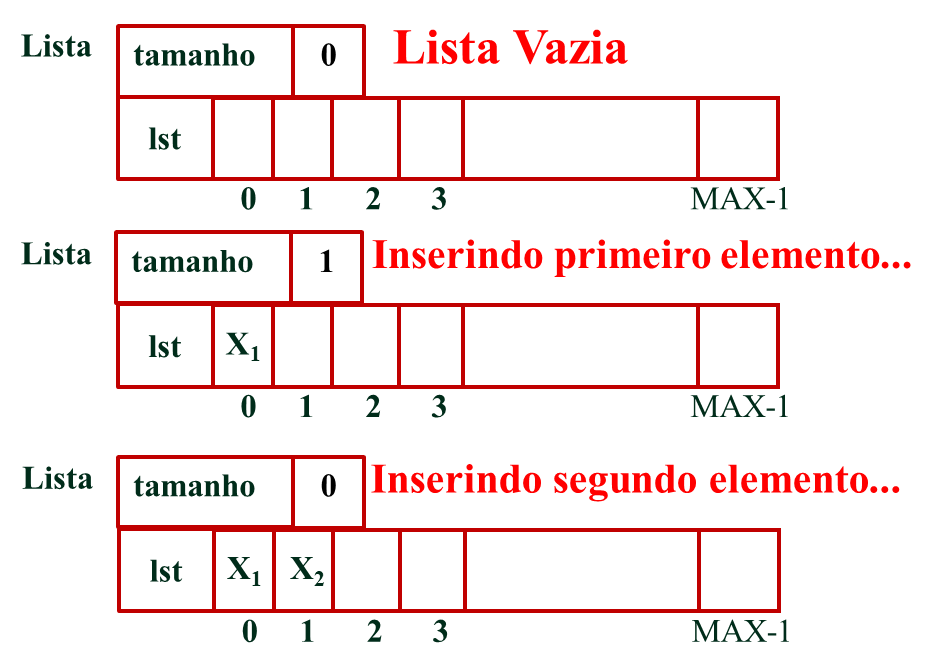
**Implementação da operação Cria Lista Vazia**: Esta operação consiste apenas em iniciar uma variável do tipo Lista, recebida como parâmetro da funçao, colocando a marca de **Fim** da Lista igual a zero.

|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Funcao** criaLista (**var** lista: Lista):  **Inicio**  lista.tamanho 🡨 0  **FimFuncao** | **void** criaLista (**Lista** \*lista)  **{**  lista->tamanho = 0;  **}** |

**Implementação da operação Testa Lista Vazia**: Esta operação , *booleana*, consiste apenas em avaliar se o tamanho da lista é igual a zero.

|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Funcao** vazia (**var** lista: Lista): boolean  **Inicio**  **Retorne** (lista.tamanho = 0)  **FimFuncao** | **int** vaziaLista (**Lista** \*lista)  **{**  **return** (lista->tamanho == 0);  **}** |

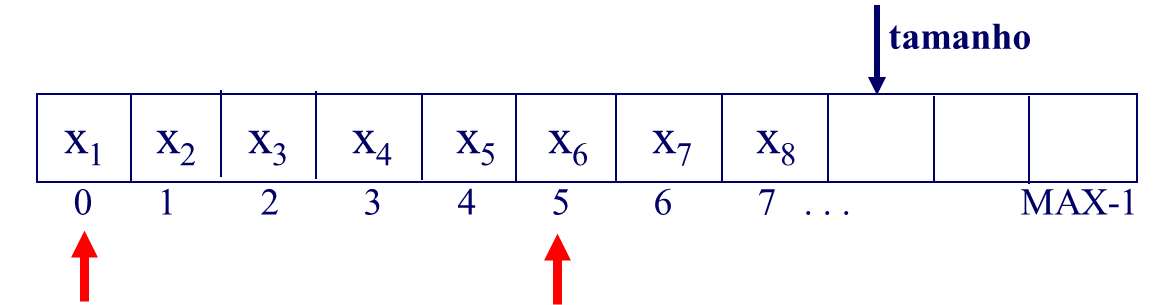
**Implementação da operação de Inserção – Insere no Final**: Esta operação consiste em inserir elementos no **final da lista**, como ilustrado a seguir:



Porém é necessário verificar *overflow* do arranjo, isto é, quando ***tamanho*** se torna igual a ***MAX***.

|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Funcao** insereFinal (x: Dados, **var** l: Lista): inteiro  **Inicio**  **Se** (l.tamanho >= MAX) **então**  **Retorne** (99) ***// código de overflow***  **FimSe**  l.lst[l.tamanho] 🡨 x  l.tamanho 🡨 l.tamanho + 1  **Retorne** (0) ***// operação bem sucedida***  **FimFuncao** | int insereFinal (Dados x, Lista \*lista)  {  if (lst->tamanho >= MAX) {  return 99; ***// código de overflow***  }  lista->lst[lista->tamanho] = x;  lista->tamanho++;  return 0; ***// operação bem sucedida***  } |

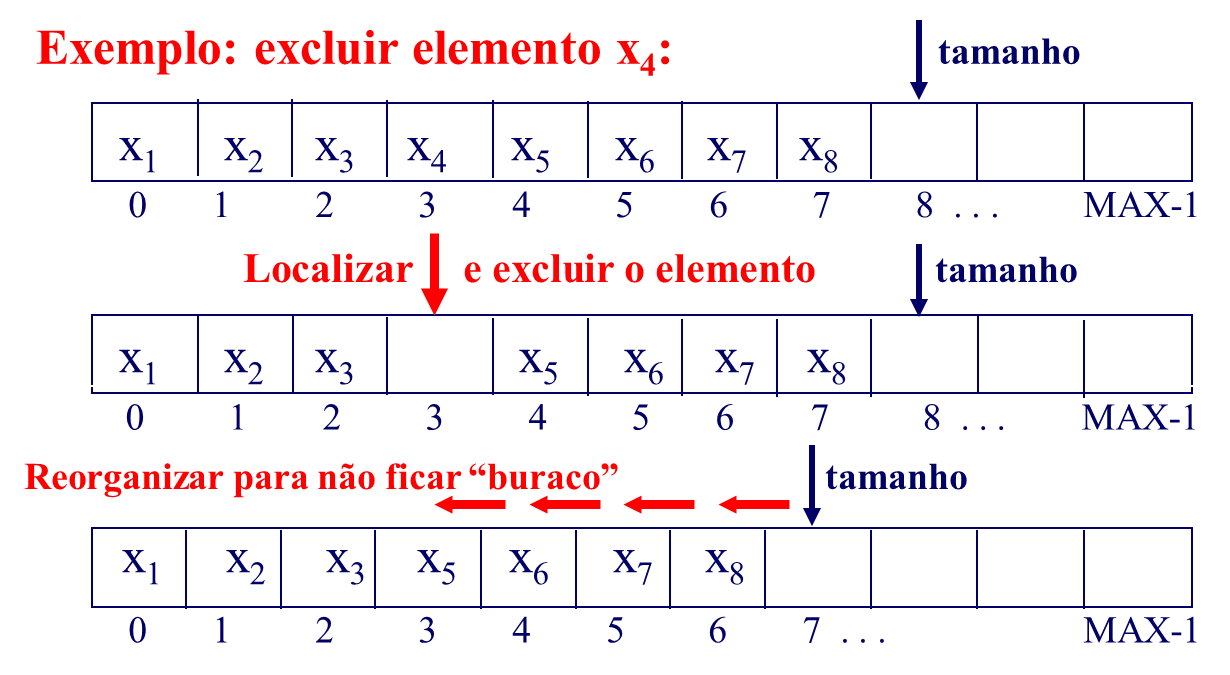
**Implementação da operação de Busca**: Esta operação consiste em caminhar em toda Lista pesquisando por um elemento informado pelo parâmetro x. Resultado da pesquisa pode ser *Sucesso* (encontrou) ou *Fracasso* (não encontrou).



Na figura, ao pesquisar pelo elemento **x6**, a operação de Busca retorna posição **5**, ao pesquisar pelo elemento **x1**, a operação retorna **0**, e assim, sucessivamente. Caso a pesquisa seja por um elemento não existente na Lista, isto é, fracasso na busca, a Operação retorna **–1**, por definição.

|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Funcao** buscaLista (x: Dados, **var** l: Lista): inteiro  **Var** pos: inteiro  **Inicio**  pos 🡨 0  **Enquanto** (pos < l.tamanho) **faça**  **Se** ( l.lst[pos] = x) **então**  **Retorne** (pos) ***// encontrou!***  **Senao**  pos 🡨 pos + 1  **FimSe**  **FimEnquanto**  **Retorne** (–1) ***// não encontrou!***  **FimFuncao** | **int** buscaLista(**Dados** x, **Lista** \*lista)  **{**  **int**  pos = 0;  **while** (pos < lista->tamanho) **{**  **if** (lista->lst[pos].chave == x.chave)  **return**(pos); ***// encontrou!***  **else** pos++;  **}**  **return**(-1) ; ***// não encontrou!***  **}** |

**Implementação da operação de Exclusão**: Consiste em localizar o item a ser excluído, retirá-lo do Arranjo e deslocar elementos para cobrir espaço vazio no Arranjo:



|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Funcao** excluiLista (x: Dados, **var** l: Lista): inteiro  **Var** i, j: inteiro  **Inicio**  **Se** ( vazia( l ) **então**  **Retorne** (999) *//Lista vazia, não pode excluir!*  **FimSe**  i 🡨 buscaLista (x, l)  **Se** ( i = -1 ) **então**  **Retorne** (88) *//Elemento x não encontrado!*  **FimSe**  **Para** j **de** i+1 **até** lista.tamanho - 1 **faça**  *// “Tapa buraco” !!!*  l.lst[ j-1 ] 🡨 l.lst[ j ]  **FimPara**  l.tamaho 🡨 l.tamanho – 1  **Retorne** (0) *// Operação realizada com sucesso!*  **FimFuncao** | **int exclui**Lista(**Dados** x, **Lista** \*lista)  **{**  **int**  i, j;  **if** (vaziaLista (\*lst)) {  return *999*; *// Lista vazia, não pode excluir!*  }  *// Pesquisa Elemento x na Lista !!!*  i = buscaLista(x, lista);  if (i == –1) {  return *88*; *// Elemento x não encontrado!*  }  for (j = i+1; aux < lista->tamanho; j++)  *// “Tapa buraco” !!!*  lista->lst[j – 1] = lista->lst[ j ];  lst->tamanho--;  return 0; *// Operação realizada com sucesso!*  **}** |