

---

# Estrutura de Dados: Lista Linear

---

Parte I –  
Introdução e  
Listas Sequenciais Estáticas

# Estrutura de dados: Lista Linear

- *Def.* Uma **Lista Linear** é uma coleção ordenada de componentes de um mesmo tipo.
- Ela é
  - ou vazia
  - ou pode ser escrita como
$$(a_1, a_2, \dots, a_n),$$
onde
    - $a_i$  são átomos de um mesmo conjunto  $S$ ;
    - $a_1$  é o primeiro elemento;
    - $a_i$  precede  $a_{i+1}$ ;
    - $a_n$  é o último elemento da lista
- Ex. listas de nomes, de peças, de valores, de pessoas, de compras, etc.

# Lista Linear

- Estrutura Homogênea: elementos de um mesmo tipo base
- Uma lista pode ser ordenada (campo “chave”) ou não-ordenada
  - Em geral, é ordenada segundo algum critério...
  - Operações básicas:
    - Verificar se lista vazia;
    - Inserção de elemento na lista;
    - Eliminação de elemento da lista;
    - Busca (Acesso) por um elemento, dada uma chave ou uma posição
- Outras operações:
  - ordenar, concatenar, inverter, etc.

# Lista Linear – Tipos de Representação

## (Mapeamento na Memória)

- **Seqüencial:** sucessor lógico de um elemento ocupa posição física consecutiva na memória (endereços consecutivos)

$L = (\text{ana}, \text{maria}, \text{paulo})$  MP:

ana	1
maria	:
paulo	$n$

- **Encadeada:** elementos logicamente consecutivos não implicam elementos (endereços) consecutivos na memória

$L = (\text{ana}, \text{maria}, \text{paulo})$  MP:

paulo	$n$
...	
ana	1
...	
maria	2

# Lista Linear – Tipos de Representação

- **Seqüencial:** sucessor lógico de um elemento ocupa posição física consecutiva na memória (endereços consecutivos)

L = (ana, maria, paulo) MP: 

ana	1
maria	:
paulo	n

**ARRAY**  
**(estática)**

- **Encadeada:** elementos logicamente consecutivos não implicam elementos (endereços) consecutivos na memória

L = (ana, maria, paulo) MP: 

paulo	<u>n</u>
...	-
ana	1
...	
maria	2

**PONTEIROS**  
**(dinâmica)**

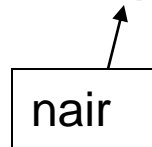
# Lista Linear – Ordenação

- Qualquer que seja o tipo de representação, a lista pode diferir quanto à ordenação:

- **Ordenada:** elementos ordenados segundo valores do campo chave e, eventualmente, de outros campos

- inserção é feita em local definido pela ordenação

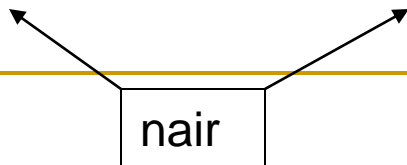
L=(ana, maria, paulo)



- **Não ordenada:**

- inserção ocorre nas extremidades (mais barato)

L=(paulo, ana, maria)



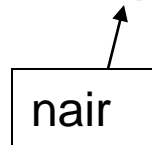
# Lista Linear – Ordenação

- Qualquer que seja o tipo de representação, a lista pode diferir quanto à ordenação:

- **Ordenada:** elementos ordenados segundo valores do campo chave e, eventualmente, de outros campos

- inserção é feita em local definido pela ordenação

L=(ana, maria, paulo)



*Se Sequencial → deslocamento de registros na MP → > tempo*

- **Não ordenada:**

- inserção ocorre nas extremidades (mais barato)

L=(paulo, ana, maria)



---

# Lista Linear

- Escolher entre uma ou outra representação (sequencial ou encadeada) vai depender do comportamento da lista na aplicação (tamanho, operações mais freqüentes, etc.).
- A eficiência das operações depende também da representação usada e de outros fatores: se a lista está ordenada, se é grande ou pequena, etc.



# Organização vs. alocação de memória

- **Alocação Estática**: reserva de memória em tempo de compilação
- **Alocação Dinâmica**: em tempo de execução

Organização da memória

	<b>Seqüencial</b>	<b>Encadeada</b>
Alocação da memória	<b>Estática</b>	<b>Dinâmica</b>

- **Seqüencial** e **estática** : Uso de arrays
- **Encadeada** e **dinâmica** : Uso de ponteiros
- **Encadeada** e **estática** ?
- **Seqüencial** e **dinâmica** ?

# Organização vs. alocação de memória

- **Alocação Estática**: reserva de memória em tempo de compilação
- **Alocação Dinâmica**: em tempo de execução

## Organização da memória

	<b>Seqüencial</b>	<b>Encadeada</b>
<b>Alocação da memória</b>	<b>Estática</b>	<b>Dinâmica</b>

- **Seqüencial** e **estática** : Uso de arrays
- **Encadeada** e **dinâmica** : Uso de ponteiros
- **Encadeada** e **estática** : Array simulando Mem. Princ.
- **Seqüencial** e **dinâmica** : Alocação dinâmica de Array

# Exemplo: TAD Lista (versão ordenada e não ordenada)

## Interface

**Valores**: *tipo\_elem*: pode ser composto por vários campos, sendo um deles o campo chave (*tipo\_chave*).

- Um campo é dito chave se possui valores distintos para elementos distintos (p.ex. RG, CPF, Nro. USP), e é usado sempre que se quer identificar unicamente um elemento.

# Exemplo: TAD Lista

## Operações sobre a Lista:

- Inicialização (criar lista vazia)
  - *Definir (lista)*
  
- Inserir Elemento (ordenada e não ordenada)
  - *Inserir\_ord (elemento, lista)*  
Insere de maneira a manter a lista ordenada
  
  - *Inserir\_posic (elemento, posic, lista)*  
Insere elemento na posição posic da lista (para lista não ordenada).

# Exemplo: TAD Lista

- Busca a posição (na lista) de um elemento cuja chave é dada
  - *Busca(chave, lista, posic);*  
retorna a posição do elemento na lista ou retorna um valor inválido caso ele não esteja na lista NÃO ORDENADA. A lista não é alterada.
  - *Busca\_ord(chave, lista, posic);*  
retorna a posição do elemento na lista ou retorna um valor inválido caso ele não esteja na lista ORDENADA. A lista não é alterada.

---

# Exemplo: TAD Lista

- Busca elemento dada a posição na lista.

Buscar\_posic(posic, lista, registro);

retorna o registro que ocupa a posição p dada, se for uma posição válida da lista. Retorna true, se posição válida, ou false, c.c.

---

# Exemplo: TAD Lista

- Eliminar elemento na posição dada

- *Remover(posic, lista);*

Antes de remover, ocorre a localização da posição, dada a chave.

- Eliminar elemento, dada sua chave

- *Remover\_ch(chave, lista, flag\_sucesso);*

Verifica se x está na lista antes de remover. Retorna true se remover; false, c.c.

# Exemplo: TAD Lista

- Contar número de elementos na Lista

- *Tamanho(lista);*

- Destruir a Lista

- *Apagar(lista);*

A lista é destruída (logicamente) e fica vazia: seus elementos não são mais acessíveis

- Verificar se lista está vazia

- *Vazia(lista);*

Retorna true se vazia, false, c.c.

- Verificar se lista está cheia

- *Cheia(lista);*

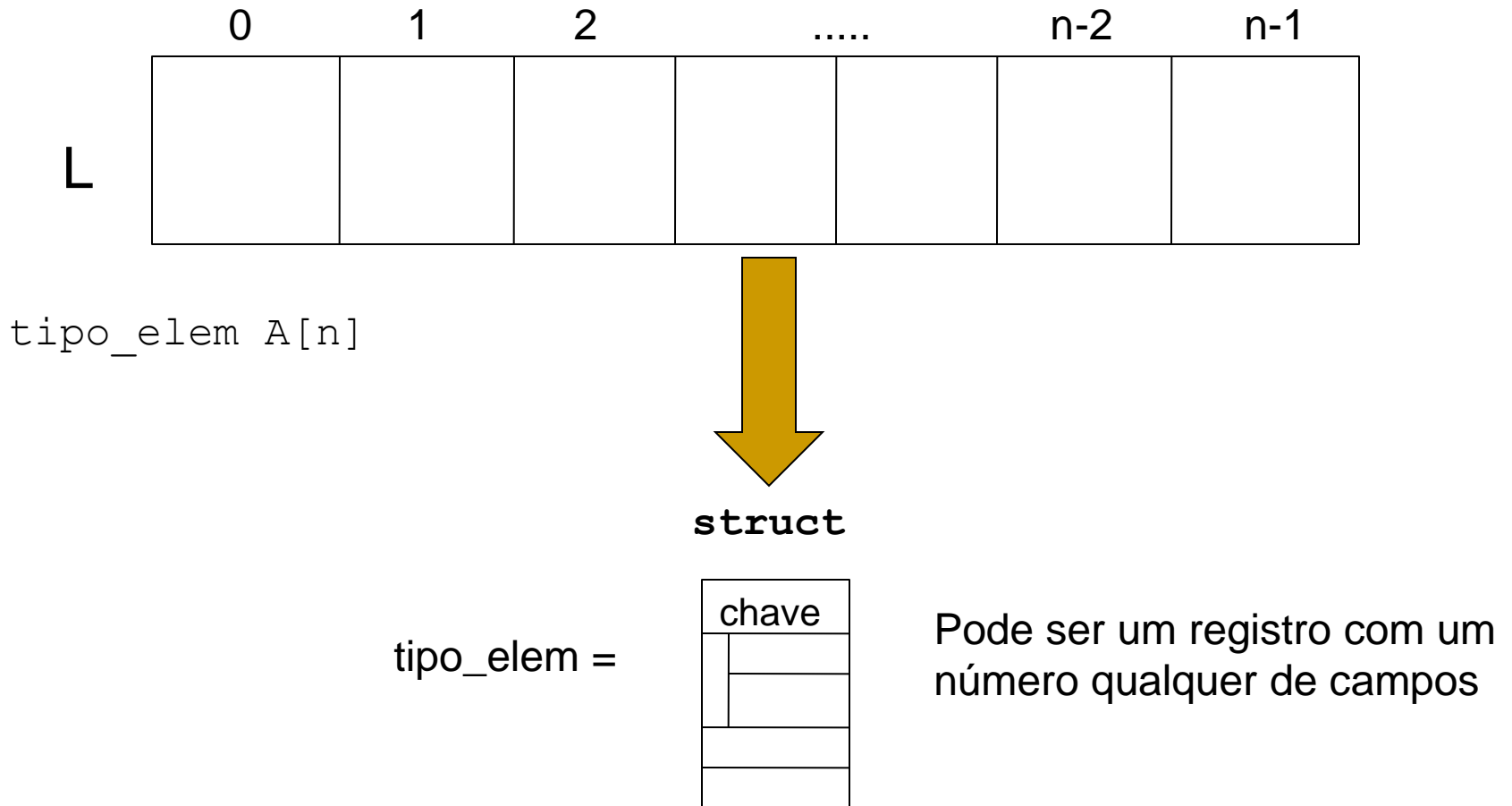
Verifica se todo espaço para a lista está ocupado (apenas se for estática). Retorna true se cheia, false, c.c.



# TAD Lista – Implementação Sequencial Estática

- As escolhas dos tipos de dados a seguir valem tanto para lista **ordenada** como **não ordenada**. O que vai diferir no caso de não ordenada são as funções de inserção, busca e remoção.
- No caso da combinação **Sequencial Estática**, usa-se o **ARRAY** para armazenar os elementos da lista.

# Tipo de Dado para Lista Sequencial Estática



# TAD Lista – implementação

```
#define MAX 100 /*estimativa do tamanho máximo da lista*/
#define TRUE 1 /*define tipo boleano - não existe na linguagem C*/
#define FALSE 0
#define boolean int

typedef int tipo_chave; /*tipo da chave - deve admitir comparações*/

typedef struct{ /*tipo registro*/
    char nome[30];
    // ...
}tipo_info;

typedef struct{ /*tipo elemento*/
    tipo_chave chave;
    tipo_info info;
}tipo_elem;

typedef struct{
    int nelem; /*número atual de elementos*/
    tipo_elem A[MAX];
}Lista;
```

```
Lista L; /*exemplo de declaração*/
```

```
boolean Vazia(Lista *L){
    /*Retorna true (1) se lista vazia, false (0) caso contrário*/
    return (L->nelem == 0);
}

boolean Cheia(Lista *L){
    /*Retorna true (1) se lista cheia, false (0) caso contrário*/
    return (L->nelem == MAX);
}

void Definir(Lista *L){
    /*Cria uma lista vazia. Este procedimento deve ser chamado para
    cada nova lista antes de qualquer outra operação.*/
    L->nelem = 0;
}

void Apagar(Lista *L){
    /*Apaga logicamente uma lista*/
    L->nelem = 0;
}
```

```

boolean Inserir_posic(tipo_elem x, int p, Lista *L){
    /* Insere novo elemento, x, na posição p da Lista.
    Se  $L = a_1, a_2, \dots, a_n$  então temos  $a_1, a_2, \dots, a_{p-1} \times a_{p+1} \dots a_n$ .
    Devolve true se sucesso, false c.c. (L não tem nenhuma posição p
    ou Lista cheia). Operação para LISTA NÃO ORDENADA! */

    int q;

    if (Cheia(L) || p > L->nelem+1 || p < 1){
        /* lista cheia ou posição não existe */
        return FALSE;

    } else {
        for (q=L->nelem; q>=p; q--)
            L->A[q] = L->A[q-1];

        L->A[p-1] = x; /* o p-ésimo elemento ocupa posição p-1 */
        L->nelem++;
        return TRUE; /* inserção com sucesso */
    }
}

```

**(nelem - p+1) Movimentos (cópia de registros)**

```
boolean Inserir_ord(tipo_elem x, Lista *L){
    /*Insere novo elemento de forma a manter a Lista ordenada
    (crescente). Devolve true se sucesso, false caso contrário
    Supõe a inserção de registros com chave repetida*/

    if (Cheia(L) return FALSE;

    if (Vazia(L)) /* insere 1º. elemento na posição 1 */
        return Inserir_posic(x, 1, L);

    else { /*acha posição de inserção*/
        int i = 0;
        while (i <= L->nelem-1)
            if (x.chave <= L->A[i].chave)
                return Inserir_posic(x, i+1, L);
            else i++;

        return Inserir_posic(x, i+1, L); /*insere na última posição*/
    }
}
```

```
boolean Inserir_ord(tipo_elem x, Lista *L){
    /*Insere novo elemento de forma a manter a Lista ordenada
    (crescente). Devolve true se sucesso, false caso contrário
    Supõe ERRO ao inserir registros com chave repetida*/

    if (Cheia(L) return FALSE;

    if (Vazia(L)) /* insere 1°. elemento na posição 1 */
        return Inserir_posic(x, 1, L);

    else { /*acha posição de inserção*/
        int i = 0;
        while (i <= L->nelem-1)
            if (x.chave < L->A[i].chave)
                return Inserir_posic(x,i+1,L);
            else if (x.chave == L->A[i].chave)
                return FALSE; /* já está na lista */
            else i++;

        return Inserir_posic(x,i+1,L); /*insere na última posição*/
    }
}
```

```

boolean Buscar(tipo_chave x, Lista *L, int *p){
    /*Retorna true se x ocorre na posição p. Se x ocorre mais de
    uma vez, retorna a posição da primeira ocorrência. Se x não
    ocorre, retorna false. Para Listas NÃO ORDENADAS*/
    /* Primeira implementação com busca linear simples*/
    if (!Vazia(L)){
        int i = 0;
        while (i <= L->nelem-1)
            if (L->A[i].chave == x){
                *p = i;
                return TRUE;
            } else
                i++;
    }
    return FALSE; /*não achou*/
}

```

**Nro Max de Comparações: nelem + 1**



```

boolean Buscar_ord(tipo_chave x, Lista *L, int *p){
    /*Retorna true se x ocorre na posição p. Se x ocorre mais de
    uma vez, retorna a posição da primeira ocorrência. Se x não
    ocorre, retorna false. Para Listas ORDENADAS*/    /*
    Implementação com busca linear simples*/
    if (!Vazia(L)){
        int i = 0;
        while (i <= L->nelem-1)
            if (L->A[i].chave >= x)
                if (L->A[i].chave == x){
                    *p = i;
                    return TRUE;
                } else
                    return FALSE;
                /*achou maior, pode parar*/
            else
                i++;
        }
    return FALSE; /*não achou*/
}

```

**Nro Max de Comparações: nelem + 1**

```

boolean Busca_bin(tipo_chave x, Lista *L, int *p){
    /*Retorna em p a posição de x na Lista ORDENADA, e true. Se x não
    ocorre retorna false*/
    /* Implementação de Busca Binária */
    int inf = 0;
    int sup = L->nelem-1;
    int meio;
    while (!(sup < inf)){
        meio = (inf + sup)/2;
        if (L->A[meio].chave == x) {
            *p = meio; /*sai da busca*/
            return TRUE;
        } else {
            if (L->A[meio].chave < x)
                inf = meio+1;
            else
                sup = meio-1;
        }
    }
    return FALSE;
}

```

<p><b>Nro Max de Comparações:</b>  <math>\log_2(\text{nelem})</math></p>
--

## Versão Recursiva da Busca Binária (booleana)

```
boolean Busca_bin_rec(tipo_chave x, Lista *L, int inf, int sup, int *p){
    /*Se encontrar x, retorna em p sua posição e true; false, c.c.*/
    if (inf > sup)
        return FALSE;
    else {
        int meio = (inf + sup)/2;
        if (L->A[meio].chave > x)
            return Busca_bin_rec(x, L, inf, meio-1, p);
        else if (L->A[meio].chave < x)
            return Busca_bin_rec(x, L, meio+1, sup, p);
        else { /*achou x na posição meio*/
            *p = meio;
            return TRUE;
        }
    }
}
```

**Nro Max de Comparações:**  
 **$\log_2(\text{nelem})$**

## Função externa para o usuário (booleana)

```
boolean Busca_bin(tipo_chave x, Lista *L, int *p){
    return Busca_bin_rec(x, L, 0, L->nelem-1, p);
}
```

```
void Remover_ch(tipo_chave x, Lista *L, boolean *removeu){
    /* Remover dada a chave. Retorna true, se removeu, ou
    false, c.c.*/
    int *p = malloc(sizeof(int));
    *removeu = FALSE;
    if (Busca_bin(x, L, p)){ /*procura por busca binária*/
        Remover_posic(p, L);
        *removeu = TRUE;
    }
}
```

**Nro de Mov = (nelem - p)**

```

void Remover_posic(int *p, Lista *L){
    /* Só é ativada após a busca ter retornado a posição p
    do elemento a ser removido - Nro de Mov = (nelem - p)*/
    int i;
    for (i = *p+1; i < L->nelem-1; i++)
        L->A[i-1] = L->A[i];

    L->nelem--;
}

```

No programa, se quero eliminar o registro com chave x:

```

boolean sucesso
...
Remover_ch(x, L, &sucesso);
if (!sucesso) ... /*x não está na lista*/
else ... /*removeu*/

```

```
void Imprimir(Lista *L){
    /*Imprime os elementos na sua ordem de precedência*/
    int i;
    if (!Vazia(L))
        for (i = 0; i < L->nelem-1; i++)
            Impr_elem(L->A[i]);
}

void Impr_elem(tipo_elem t){
    printf("chave: %d", t.chave);
    printf("info: %s", t.info.nome);
    //...
}

int Tamanho(Lista *L){
    /* Retorna o tamanho da Lista. Se L é vazia retorna 0 */
    return L->nelem;
}
```

---

# Lista Linear Sequencial Estática:

## Resumo

### ■ Vantagens:

- Acesso Direto a cada elemento:
  - `Lista.A[i]` e `(Lista->A[i])`
- Tempo Constante (decorrência do array)
  - independente do valor de  $i$
- Busca pode ser Binária, se Lista Ordenada (decorrência do array) –  $O(\log_2(\mathit{nelem}))$

### ■ Desvantagens:

- Movimento de dados na Inserção e Eliminação, se Lista Ordenada
  - Tamanho máximo da lista é delimitado a priori (decorrência do array)
    - risco de overflow
-

---

# Lista Linear Sequencial Estática:

## Resumo

- Quando optar por ela?
    - listas pequenas → custo insignificante
    - conhecimento prévio do comportamento da lista:
      - poucas inserções/eliminações (ou “comportadas”)
-



# Lista de Exercícios

Escreva o código das seguintes operações adicionais para o TAD Lista:

- 1) Verificar se L está ordenada (crescente ou decrescente)
- 2) Fazer uma cópia de Lista L1 em outra L2
- 3) Fazer uma cópia da Lista L1 em L2, eliminando repetidos
- 4) Inverter L1, colocando o resultado em L2
- 5) Inverter a própria L1
- 6) Intercalar (*merge*) L1 com L2, gerando L3 (considere L1 e L2 ordenadas)
- 7) Eliminar de L1 todas as ocorrências de um dado elemento (L1 está ordenada)