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

    $(a_1, a_2, ..., a_n)$,

  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.

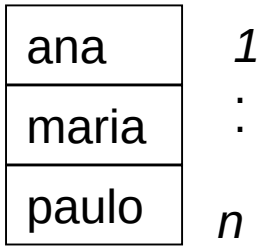
Lista Linear

- 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)

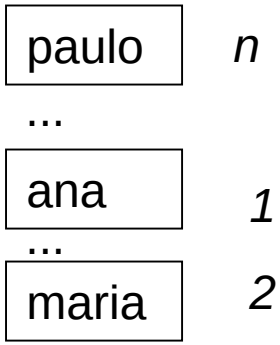
- **Seqüencial:** sucessor lógico de um elemento ocupa posição física consecutiva na memória (endereços consecutivos)

L = (ana, maria, paulo) MP:



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

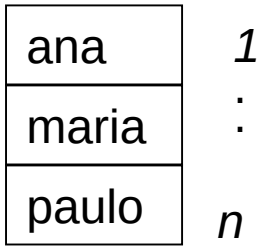
L = (ana, maria, paulo) MP:



Lista Linear – Tipos de Representação

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

L = (ana, maria, paulo) MP:

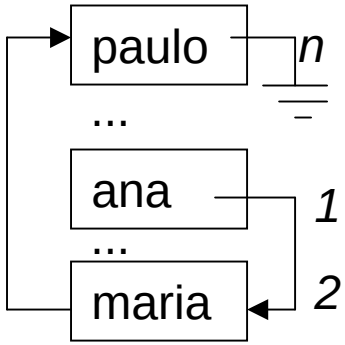


ARRAY

(estática)

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

L = (ana, maria, paulo) MP:



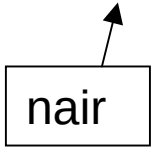
PONTEIROS

(dinâmica)

Lista Linear – Ordenação

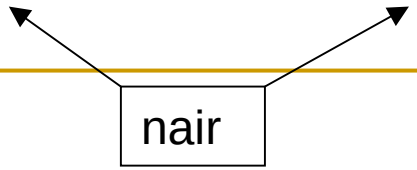
- Qualquer que seja o tipo de representação, a lista pode diferir quanto à ordenação:
  - Ordenada: elementos ordenados segundo valores do campo chave e, eventualmente, de outros campos
    - inserção é feita em local definido pela ordenação

    L=(ana, maria, paulo)



  - Não ordenada:
    - inserção ocorre nas extremidades (mais barato)

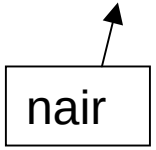
    L=(paulo, ana, maria)



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

Alocação da memória \ Organização da memória	Seqüencial	Encadeada
Estática		
Dinâmica		

- 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

Alocação da memória \ Organização da memória	Seqüencial	Encadeada
Estática		
Dinâmica		

- 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)
  - *void Definir (Lista *L)*
- Inserir Elemento (ordenada e não ordenada)
  - *void Inserir_ord (tipo_elem elemento, Lista *L)*

    Insere de maneira a manter a lista ordenada
  - *void Inserir_posic (tipo_elem elemento, int p, Lista *L)*

    Insere elemento na posição p da lista (para lista não ordenada).

Exemplo: TAD Lista

- Busca a posição (na lista) de um elemento cuja chave é dada

  - *int Busca(tipo_chave ch, Lista *L);*

    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.

  - *int Busca_ord(tipo_chave ch, Lista *L);*

    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.

boolean Buscar_posic(int p; Lista *L; tipo_elem *reg);

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
  - *void Remover(int p, Lista *L)*;

    Antes de remover, ocorre a localização da posição p, dada a chave.
- Eliminar elemento, dada sua chave
  - *boolean Remover_ch(tipo_chave x, Lista *L);*

    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
  - *int Tamanho(Lista *L)*;
- Destruir a Lista
  - *void Apagar(Lista *L);*

    A lista é destruída (logicamente) e fica vazia: seus elementos não são mais acessíveis
- Verificar se lista está vazia
  - *boolean Vazia(Lista *L);*

    Retorna true se vazia, false, c.c.
- Verificar se lista está cheia
  - *boolean Cheia(Lista *L);*

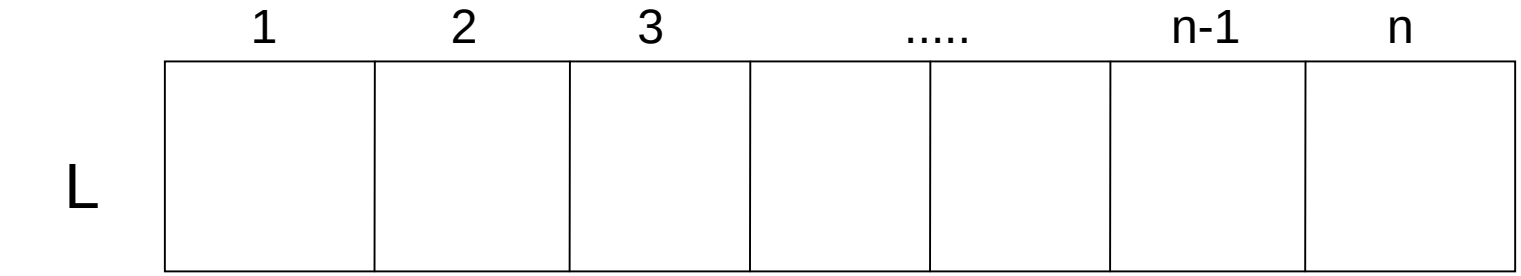
    Verifica se todo espaço para a lista está ocupado (apenas se for estática). Retorna true se cheia, false, c.c.

TAD Lista – Implementação Sequencial Estática

Implementation

- 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.

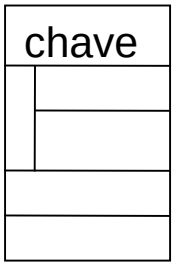
Tipo de Dado para Lista Sequencial Estática




```
tipo_elem A[n+1]
```

`struct`

tipo_elem =



Pode ser um registro com um número qualquer de campos

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+1];
}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;
   L->A[0].chave = 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 = a1, a2,... an então temos a1, a2,...ap-1 x ap+1 ... an.
   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+1] = L->A[q];

        L->A[p] = x;
        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*/

   int i = 1;

   if (Vazia(L))
       return Inserir_posic(x, i, L);

   else {    /*acha posição de inserção*/

       while (i <= L->nelem)
         if (x.chave < L->A[i].chave)
               return Inserir_posic(x,i,L);
         else  i++;

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

Só chamar Inserir_ord se lista não estiver cheia!!!!

```
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 = 1;
        while (i <= L->nelem)
                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 = 1;
        while (i <= L->nelem)
               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 = 1;
   int sup = L->nelem;
   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;
}
```

Nro Max de Comparações: $\log_2(nelem)$

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: $\log_2(nelem)$

Função externa para o usuário (booleana)

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

No caso geral, um usuário do TAD chamaria uma busca (ordenada ou não) por chave, que aí sim ativaria um dos métodos implementados. Por exemplo:

```
boolean Busca_não_ord(tipo_chave x, Lista *L, int *p){
/*chama a busca. Retorna true e posição p, se encontrou;
false, caso contrário*/
        *p = Buscar(x, L);
        return (*p != L->nelem+1);
}
```

Obs.: `Buscar(x, L)` seria a função do TAD

```
void Remover_ch(tipo_chave x, Lista *L, boolean *removeu){
   /* Remover dada a chave. Retorna true, se removeu, ou
   false, c.c.*/
   int *p;
   *removeu = FALSE;
   if (Busca_bin(x, L, p)){
       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; i++)
        L->A[i-1] = L->A[i];

   L->nelem--;
}
```

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

```
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 = 1; i< L->nelem; 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) – $O(\log_2(nelem))$
- 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)