

---

# Listas:

a última das 3 estruturas lineares (Pilhas, Filas e Listas)

... árvores e grafos são não lineares!

14/9/2010

---

Conceitos,

todas as variações para listas: **ordenadas**, **com**  
**nó cabeça**, **duplamente encadeadas**, **circulares**  
e exemplos de aplicações

---

# O que aprendemos até aqui?

1. Conceito de tipos abstratos de dados – TADs
  2. Técnicas para armazenamento de um conjunto de elementos na memória: **representação** seqüencial ou encadeada, e **implementação** estática ou dinâmica
  3. O que é, para que serve, e como funciona uma estrutura **Pilha** e uma estrutura **Fila**
  4. Como implementar pilhas e filas como TADs, segundo as técnicas de rep/imp acima citadas, com ênfase em SEQ/EST e ENC/DIN
  5. Como utilizar TADs em programas de clientes sem conhecer os detalhes de implementação
-

# Lista Linear

Uma lista linear é um conjunto de elementos ou uma seqüência de zero ou mais elementos  $x_1, x_2, \dots, x_n$  na qual  $x_i$  é de um determinado tipo e  $n$  é o tamanho da lista.

- Sua principal propriedade estrutural envolve as posições relativas dos itens em uma dimensão:
  - Cada elemento tem um **único** predecessor e um **único** sucessor, exceto o primeiro (não tem predecessor) e o último (não tem sucessor).

**Ex.:** Listas de **nomes** de alunos aprovados no vestibular, de **peças**, de valores de **salários**, de **notas** de alunos de uma turma, de **compras**, etc.

# TAD lista

■ Não há concordância sobre suas operações básicas, mas operações típicas seriam:

- ❑ **Definir(L)**: criar uma lista vazia.
- ❑ **Esvaziar(L)**: tornar uma lista L já existente vazia.
- ❑ **Destruir(L)**: destruir uma lista L.
  
- ❑ **Acessar(L, i, x)**: consulta o i-ésimo elemento de L.
- ❑ **Inserir(L, i, x)**: Insere o elemento x depois do i-ésimo elemento da lista; altera os índices dos próximos elementos.
- ❑ **Remover(L, i, x)**: Elimina o i-ésimo elemento da lista; altera os índices dos próximos elementos.
  
- ❑ **Tamanho(L)**: retorna o tamanho da lista L.
- ❑ **Lista\_vazia(L)** e **Lista\_cheia(L)**: checa se L está vazia e checa se L está cheia.
- ❑ **Primeiro(L)** e **Fim(L)**: retornam o primeiro e a posição após o último elemento da lista L.
- ❑ **Localizar(L, x)**: pesquisa a ocorrência de um item com um dado valor; retorna a posição de x na lista

# TAD lista

- As três situações de acessos (inserção, eliminação e acessar) são **direcionadas a um elemento específico**, e não mais ao “primeiro” (ou último) a entrar na lista.
- Situações:
  - ❑ Cadastro de funcionários por ordem alfabética
  - ❑ Lista de passageiros em um voo
  - ❑ Lista de deputados presentes no congresso
  - ❑ Telefones da cidade
- Exemplos:
  - ❑ “Carla Peres” foi despedida (**retire** seu nome do cadastro)
  - ❑ “Edmundo Animal” é funcionário? **Verifique** se seu nome consta do cadastro.
  - ❑ Qual o salário do funcionário “Pedro Malan”? (**consulta**)
  - ❑ “Sandra Bulloc” foi contratada; **inclua-a** no cadastro
- Nos exemplos acima, os registros de cada funcionário são (ordenados e) procurados pelo nome. O nome, neste caso, é a chave de busca, ou simplesmente **chave**.

---

# Listas e aplicações

- Diversas necessidades podem ser requeridas por aplicações
    - Determinam: (a) como será a lista e (2) quais as operações
  
  - Requisitos para a decisão:
    - Capacidade de armazenamento
    - Velocidade de acesso
    - Facilidade de manipulação
-

# Capacidade de armazenamento

- Escolha entre Implementação **Estática e Dinâmica**
- **Sequencial Estática:** a lista pode ser percorrida em qualquer direção.
  - A inserção de um novo item pode ser realizada **após o último item** com custo **constante**.
  - A inserção de um novo item no **meio da lista** requer um **deslocamento** de todos os itens localizados após o ponto de inserção.
  - Da mesma forma, retirar um item do **início da lista (ou da posição i)** requer um **deslocamento** de itens para preencher o espaço deixado vazio.
- Vantagem: a economia de memória, pois os ponteiros são implícitos nesta estrutura.
- Como desvantagens :
  - o custo para inserir ou retirar itens da lista, pode causar um deslocamento de todos os itens, no pior caso;
  - em aplicações em que não existe previsão sobre o crescimento da lista, a utilização de vetores em linguagens como o Pascal pode ser problemática porque neste caso o tamanho máximo da lista tem que ser definido em tempo de compilação.
  - C já não sofre deste problema pois podemos realocar mais espaço para um vetor dinâmico, via realloc.

# Capacidade de armazenamento

## ■ Escolha entre **Estática e Dinâmica**

■ **Encadeada Dinâmica (lista ligada – linked list):** Este tipo de implementação permite utilizar posições não contíguas de memória, sendo possível **inserir e retirar** elementos **sem haver necessidade de deslocar** os itens seguintes da lista.

■ O nome *linked list* vem do fato de ponteiros serem chamados de links, sendo que cada ponteiro aponta seu sucessor nesta estrutura.

### ■ Vantagens:

- A implementação através de ponteiros permite **inserir ou retirar itens do meio** da lista a um **custo constante**, aspecto importante quando a lista tem que ser mantida em ordem.
- Em aplicações em que não existe previsão sobre o crescimento da lista é conveniente usar listas encadeadas dinâmicas, porque neste caso o tamanho máximo da lista não precisa ser definido a priori.

### ■ Desvantagem:

- A maior desvantagem deste tipo de implementação é a utilização de memória extra para armazenar os ponteiros.



# Capacidade de armazenamento

- Informação a ser armazenada
  - Exemplo: **inteiros vs. strings**
    - Melhor solução: uma tabela de inteiros e strings, usando-se os inteiros como informação
  - Por exemplo, em uma universidade, pode-se mapear

<b>Inteiro</b>	<b>String</b>
1	Professor
2	Estudante
3	Técnico
...	...

# Velocidade de acesso à informação

- Perguntas a serem respondidas
  - Que informação se quer acessar?
  - Com que frequência se quer acessar a informação?
- Pode determinar
  - Estrutura de dados: listas restritas (pilha, fila) ou genéricas
  - Organização da informação na lista
    - Insere-se elementos no início, no fim ou no meio da lista?  
**Informação ordenada ou não?**
    - Esta última decisão vai afetar o desempenho de algumas operações, pois é necessário inserir ordenadamente, por exemplo, em vez de inserir sempre no início (lista encadeada) ou fim (lista seqüencial).
  - Número de ponteiros a ser utilizado
    - É interessante manter pelo menos 2: primeiro e último para facilitar operações nas pontas.

# Facilidade de manipulação da lista

- Operações a serem efetuadas sobre os dados
  - Organização dos dados
  - Ponteiros
- Perguntas
  - Quais as operações mais freqüentes?
  - Qual o custo computacional de cada operação?
- Dá origem a **variações da lista** que veremos daqui a pouco

# Listas Ordenadas

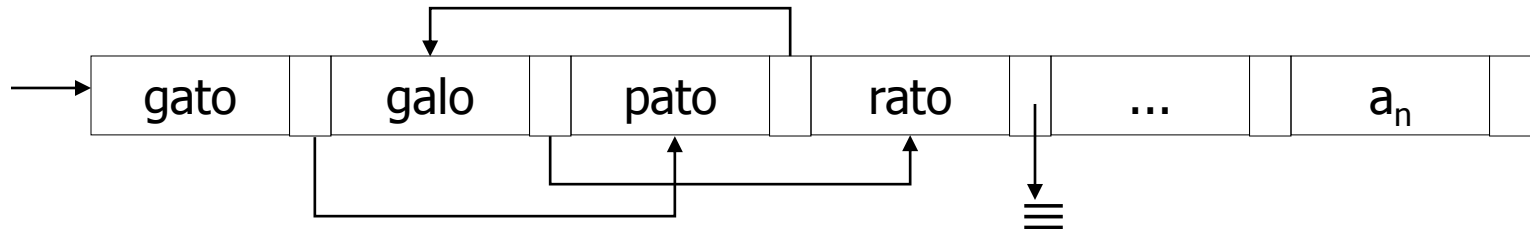
- **Uma lista pode ser ordenada ou não**
- Este fato vai alterar toda a composição das operações do TAD!
- Vai afetar o desempenho de algumas operações,
  - pois é necessário inserir ordenadamente, por exemplo, em vez de inserir sempre no início (**lista encadeada**) ou no fim (**lista seqüencial**).
- Não fizemos isto com Pilha/Fila
  - vejam que o TAD Fila de Prioridade é um outro conceito

# Lista estática

- Implementação usando array como seqüência de registros;
  - Consecutiva – lista seqüencial estática.



- Não consecutiva – lista encadeada estática (não focaremos nesta representação)



# Lista estática: 1) encadeada e 2) sequencial

## ■ Declaração da estrutura

```
#define TAM ...
```

```
typedef char elem;  
typedef struct lista Lista;
```

```
typedef struct bloco {  
    elem item;  
    int prox;  
} no;
```

```
struct lista {  
    int ini, primeiro_vazio;  
    no itens[TAM];  
    int tam;  
};
```

## ■ Declaração da estrutura

```
#define TAM ...
```

```
typedef char elem;  
typedef struct lista Lista;
```

```
struct lista{  
    int ini, primeiro_vazio;  
    elem itens[TAM];  
    int tam;  
};
```

---

# TAD LISTA

- Operações em vermelho: adequadas para LISTAS ORDENADAS
  - Operações em verde: opcionais; melhoram o entendimento do código
  - Outras operações (preto): servem para Listas ordenadas ou não ordenadas.
-

# Um conjunto básico de operações

## 1. Definir (L)

{ Cria uma lista vazia. Este procedimento deve ser chamado para cada nova lista antes da execução de qualquer operação }

## 2. Destruir (L)

{ Destroi uma lista }

## 3. $Y = \text{Lista\_vazia}(L)$

{ Retorna true se lista vazia, false caso contrário }

## 4. $Y = \text{Lista\_cheia}(L)$

{ Retorna true se lista cheia, false caso contrário }

## $Y = \text{Fim}(L)$

{ Retorna a posição **após** o último elemento de uma lista }

## 5. Tamanho (L)

{ Retorna o tamanho da Lista. Se L vazia retorna 0 }



6.  $Y = \text{Inserir}(L; x: \text{tipo\_elem}; p: \text{posicao})$

{ Insere novo elemento na posição  $p$  da Lista. Se  $L = a_1, a_2, \dots, a_n$  então

temos  $a_1, a_2, \dots, a_{p-1} \ x \ a_{p+1} \ \dots \ a_n$ . Se  $p = \text{Fim}(L)$  temos  $a_1, a_2, \dots, a_n, \ x$

Devolve true se sucesso, false caso contrário (L não tem nenhuma posição  $p$  ou Lista cheia)}

$Y = \text{Inserir\_ord}(L; x: \text{tipo\_elem})$

{ Insere novo elemento de forma a manter a Lista ordenada.

Devolve true se sucesso, false caso contrário }

7.  $Y = \text{Localizar}(L; x: \text{tipo\_elem})$

{Retorna a posição de  $x$  na Lista. Se  $x$  ocorre mais de uma vez, a posição da primeira ocorrência é retornada. Se  $x$  não aparece retorna  $\text{Fim}(L)$ . Localizar vai implementar a **busca linear**, pois funciona sempre}

$Y = \text{Localizar\_ord}(L; x: \text{tipo\_elem})$

{Retorna a posição de  $x$  na Lista. Se  $x$  não aparece retorna  $\text{Fim}(L)$ .

Localizar\_ord vai implementar a **busca binária**}

---

8.  $Y = \text{Acessar}(L; p: \text{posicao}; \text{var } x: \text{tipo\_elem})$

{ Recupera o elemento da posição  $p$  da Lista  $L$ . Se  $p = \text{Fim}(L)$  ou não existe posição  $p$  válida retorna false, caso contrário retorna true }

$Y = \text{Acessar\_Ord}(L; p: \text{posicao}; \text{var } x: \text{tipo\_elem})$

{ Recupera o elemento da posição  $p$  da Lista  $L$ , usando  $\text{Localizar\_Ord}$ . Se  $p = \text{Fim}(L)$  ou não existe posição  $p$  válida retorna false, caso contrário retorna true }

9.  $Y = \text{Remover}(L; p: \text{posicao})$

{ Remove o elemento na posição  $p$  da Lista. Se  $L = a_1, a_2, \dots, a_n$  então temos  $a_1, a_2, \dots, a_{p-1}, a_{p+1}, \dots, a_n$ . Devolve true se sucesso, false caso contrário ( $L$  não tem nenhuma posição  $p$  ou se  $p = \text{Fim}(L)$ ) }

$Y = \text{Remover\_Ord}(L; x: \text{tipo\_elem})$

{ Remove elemento  $x$  de forma a manter a Lista ordenada. Devolve true se sucesso, false caso contrário }

---

---

$Y = \text{Prox}(L; p: \text{posicao})$

{Retorna  $p + 1$ . Se  $p$  é a última posição de  $L$  então  $\text{Prox}(p,L) = \text{Fim}(L)$ .  
 $\text{Prox}(p,L)$  retorna 0 se  $p = \text{Fim}(L)$ }

$Y = \text{Ant}(L; p: \text{posicao})$

{ Retorna  $p-1$ .  $\text{Ant}(p,L)$  retorna 0 se  $p = 1$ }

$Y = \text{Primeiro}(L)$

{ Retorna 1. Se  $L$  é vazia retorna 0 }

10. Imprimir ( $L$ )

{ Imprime os elementos na sua ordem de aparecimento }

---

# Lista Seqüencial Estática

## ■ Vantagens:

- ❑ Acesso direto indexado a qualquer elemento da lista
- ❑ Busca pode ser Binária, se **Lista Ordenada** (decorrência do array) –  $O(\log_2(N))$
- ❑ Tempo constante ( $O(1)$ ) para acessar o elemento  $i$  - dependerá somente do índice **diferente** da implementação encadeada que requer  $O(n)$

## ■ Desvantagem:

- ❑ Como o tamanho máximo é pré-estimado: risco de overflow
- ❑ **Movimento** de dados na Inserção, **se Lista Ordenada**, e Eliminação do  $i$ -ésimo

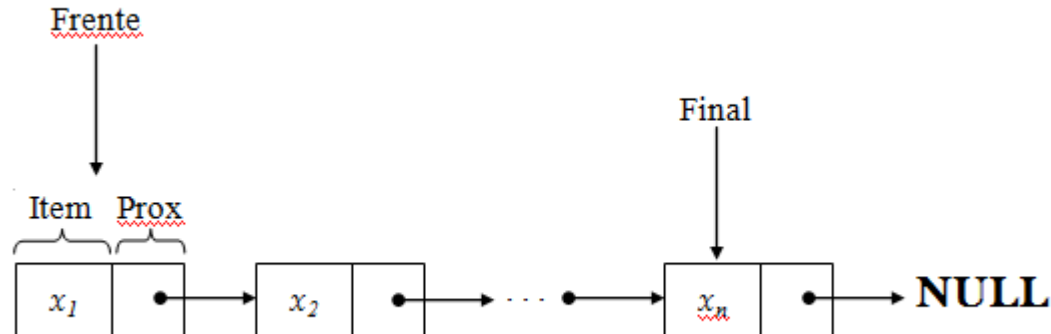
## ■ Quando usar:

- ❑ Listas pequenas
- ❑ Inserção/remoção no fim da lista
- ❑ Tamanho máximo bem definido pela aplicação

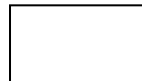
# Lista Encadeada Dinâmica

```
typedef char elem;  
typedef struct lista Lista;
```

Lista.h



tam



```
typedef struct bloco {  
    elem item;  
    struct bloco *prox;  
} no;
```

```
struct lista {  
    no *Frente, *Final;  
    int tam;  
};
```

Lista.c

# Simplificando a Estrutura

A estrutura de lista encadeada é representada pelo ponteiro para seu primeiro elemento (tipo Lista\*).

```
struct lista {  
    int info;  
    struct lista* prox;  
};
```

```
typedef struct lista Lista;
```

```
/* função de inicialização: retorna uma lista vazia */  
Lista* inicializa (void)  
{  
    return NULL; /* inserção no início: retorna a lista atualizada */  
}  
Lista* insere (Lista* l, int i)  
{  
    Lista* novo = (Lista*) malloc(sizeof(Lista));  
    novo->info = i;  
    novo->prox = l;  
    return novo;  
}
```

```
int main (void)
{
    Lista* l;          /* declara uma lista não inicializada */
    l = inicializa(); /* inicializa lista como vazia */
    l = insere(l, 23); /* insere na lista o elemento 23 */
    l = insere(l, 45); /* insere na lista o elemento 45 */
    ...
    return 0;
}
```

```
/* função vazia: retorna 1 se vazia ou 0 se não vazia */
```

```
int vazia (Lista* l)
```

```
{
    if (l == NULL)
        return 1;
    else
        return 0;
}
```

```
/* função busca: busca um elemento na lista */
```

```
Lista* busca (Lista* l, int v)
```

```
{
    Lista* p;
    for (p=l; p!=NULL; p=p->prox)
        if (p->info == v)
            return p;
    return NULL;          /* não achou o elemento */
}
```

```
/* função imprime: imprime valores dos elementos */
```

```
void imprime (Lista* l)
```

```
{
```

```
    Lista* p;    /* variável auxiliar para percorrer a lista */
```

```
    for (p = l; p != NULL; p = p->prox)
```

```
        printf("info = %d\n", p->info);
```

```
}
```

```
/* Função imprime recursiva */
```

```
void imprime_rec (Lista* l)
```

```
{
```

```
    if (vazia(l))
```

```
        return;
```

```
    /* imprime primeiro elemento */
```

```
    printf("info: %d\n", l->info);
```

```
    /* imprime sub-lista */
```

```
    imprime_rec(l->prox);
```

```
}
```

```
void libera (Lista* l)
```

```
{
```

```
    Lista* p = l;
```

```
    while (p != NULL) {
```

```
        Lista* t = p->prox; /* guarda referência para o próximo elemento
```

```
*/
```

```
        free(p);
```

```
        /* libera a memória apontada por p */
```

```
        p = t;
```

```
        /* faz p apontar para o próximo */
```

```
    }
```

```
}
```

```
void imprime (Lista * l) {
```

```
    Lista * p;
```

```
    while (p != NULL) {
```

```
        printf("info: %d\n", l->info);
```

```
        p = p->prox;
```

```
    }
```

```
}
```

```
void libera_rec (Lista* l)
```

```
{
```

```
    if (!vazia(l))
```

```
    {
```

```
        libera_rec(l->prox);
```

```
        free(l);
```

```
    }
```

```
}
```



---

Façam a função tamanho da lista  
iterativa e recursiva

# Tamanho da Lista

```
/* função tamanho retorna o tamanho da lista */
int tamanho (Lista* l)
{
    Lista* p; /* variável auxiliar para percorrer a lista */
    int n;

    n = 0;
    for (p = l; p != NULL; p = p->prox)
        n++;
    return n
}
```

```
/* função tamanho retorna o tamanho da lista */
int tamanho_rec (Lista* l)
{
    if (l == NULL) {
        return 0;
    }
    else return tamanho_rec(l->prox) + 1;
}
```

# Lista Encadeada Dinâmica

## ■ **Vantagens:**

- toda memória disponível para o programa durante a execução (*heap*)
- Movimentação de itens facilitada (inserção e remoção)

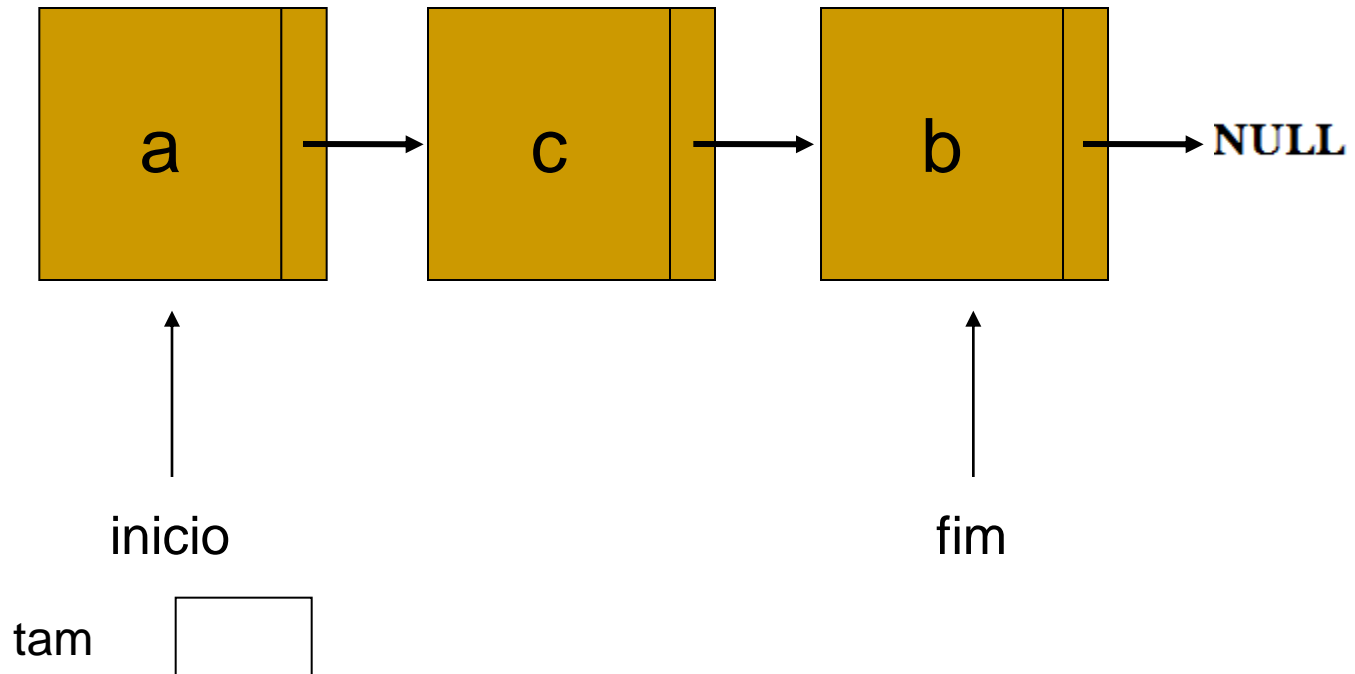
- Na lista, cada elemento deve referenciar a posição do seu sucessor: campo de ligação contém endereços reais da memória principal

## ■ **Desvantagem:**

- Busca em lista ordenada não é mais rápida do que na sequencial (nada de busca binária aqui)

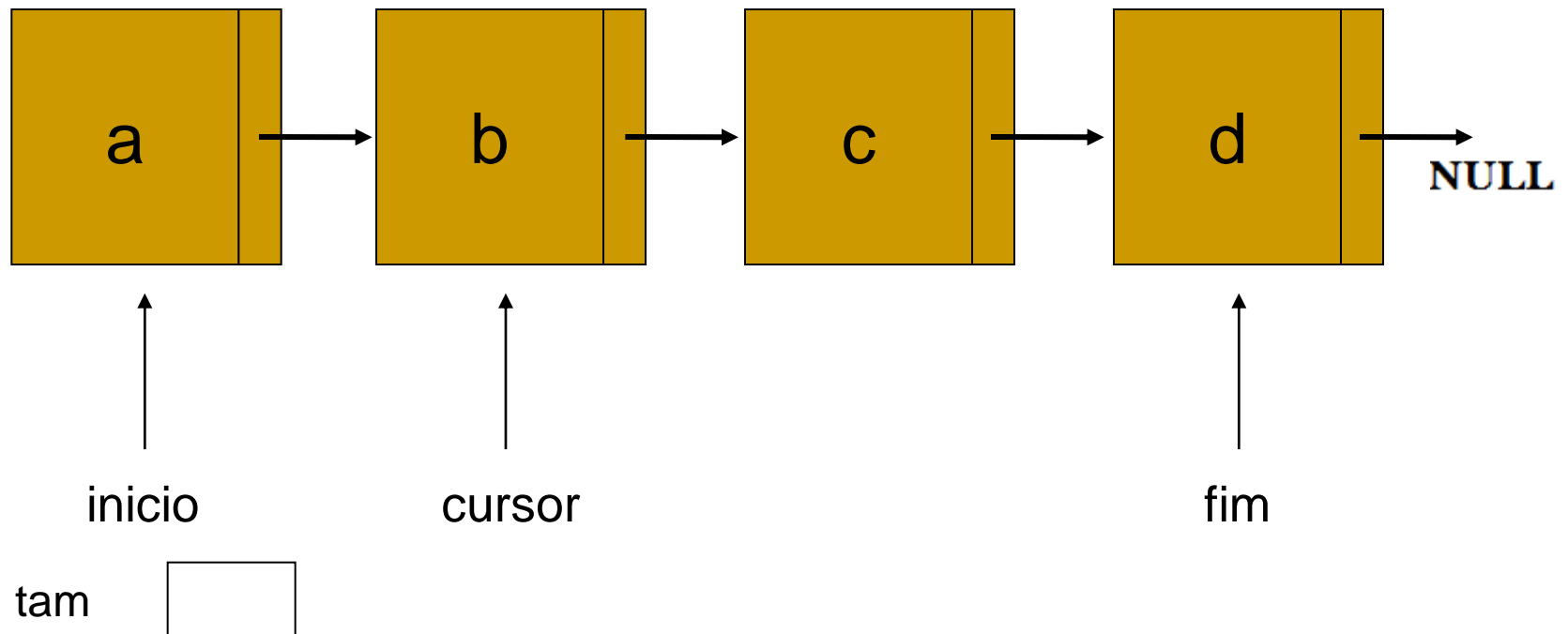
# Variações da lista

- Lista encadeada simples e desordenada



# Variações da lista

- Lista encadeada simples, ordenada e com ponteiro extra (implementação de um editor)



# Variações da lista

- Lista encadeada simples, ordenada e com ponteiro extra

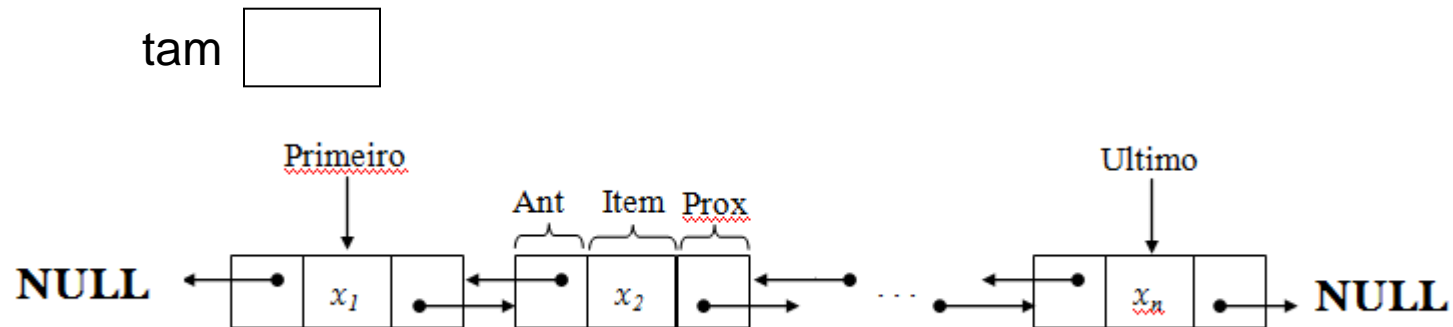
```
typedef struct bloco {  
    elem info;  
    struct bloco *prox;  
} no;
```

```
struct lista {  
    no *inicio, *fim, *cursor;  
    int tam;  
};
```

Lista.c

# Variações da lista

- Lista duplamente encadeada
  - Facilita navegação



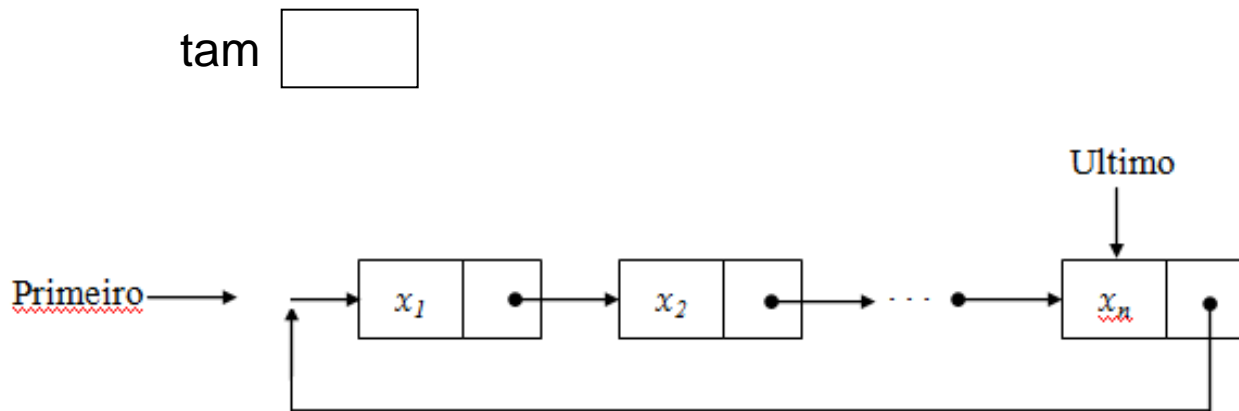
```
typedef struct bloco {  
    elem item;  
    struct bloco *prox, *ant;  
} no;
```

Lista.c

```
struct lista {  
    no *Primeiro, *Ultimo;  
    int tam;  
};
```

# Listas Circulares Encadeadas Dinâmicas

- Se o nó prox do último nó apontar para o primeiro, teremos uma lista circular.
- Esta representação elimina o perigo de acessar posição inválida de memória.
- Em listas circulares não temos primeiro nem último naturais. O esquema abaixo é o mais usual.
- Lista vazia: NULL





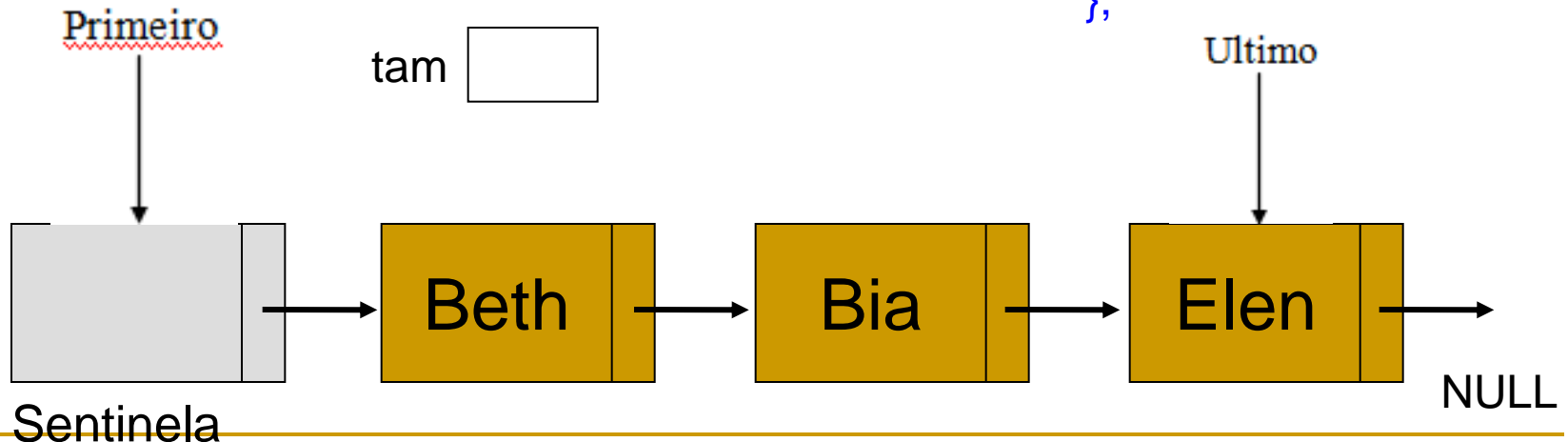
# Lista com nó de cabeçalho

```
typedef char elem;  
typedef struct lista Lista;
```

```
typedef struct bloco {  
    elem info;  
    struct bloco *prox;  
} no;
```

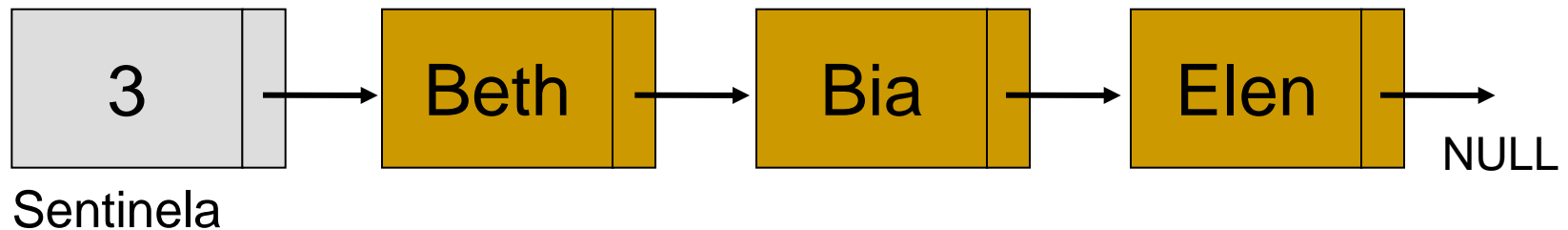
```
struct lista{  
    no *Primeiro, *Ultimo;  
    int tam;  
};
```

- Nó de cabeçalho
  - *Header*, sentinela, nó cabeça, etc.
- Para que? Quais usos????



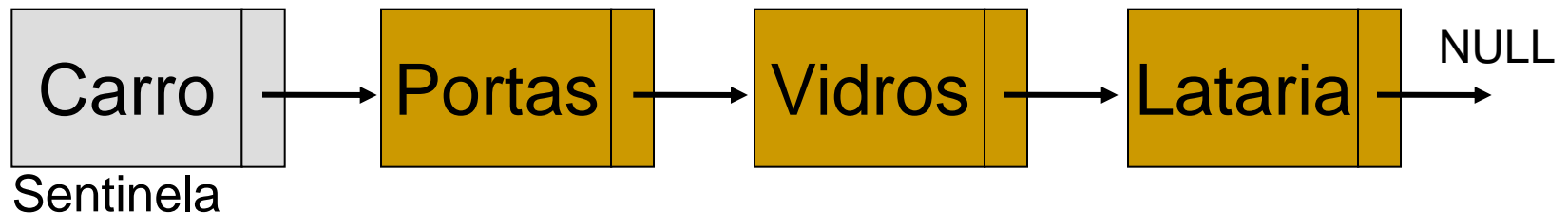
# Lista com nó de cabeçalho

- Possibilidades de uso
  - **Informação global** sobre a lista que possa ser necessária na aplicação
    - Armazenar número de elementos da lista, para que não seja necessário atravessá-la contando seus elementos



# Lista com nó de cabeçalho

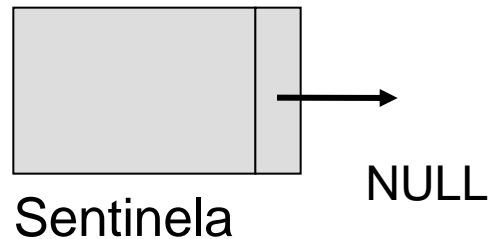
- Possibilidades de uso
  - **Informação global** sobre a lista que possa ser necessária na aplicação
    - Em uma fábrica, guarda-se as peças que compõem cada equipamento produzido, sendo este indicado pelo nó sentinela
    - Informações do voo correspondente a uma fila de passageiros



---

# Lista com nó de cabeçalho

- Lista vazia contém somente o nó sentinela



---

# Lista com nó de cabeçalho

- Possibilidades de uso
  - **Informação global** sobre a lista que possa ser necessária na aplicação
    - Operação de busca de informação pode ser simplificada se usarmos também a representação circular.

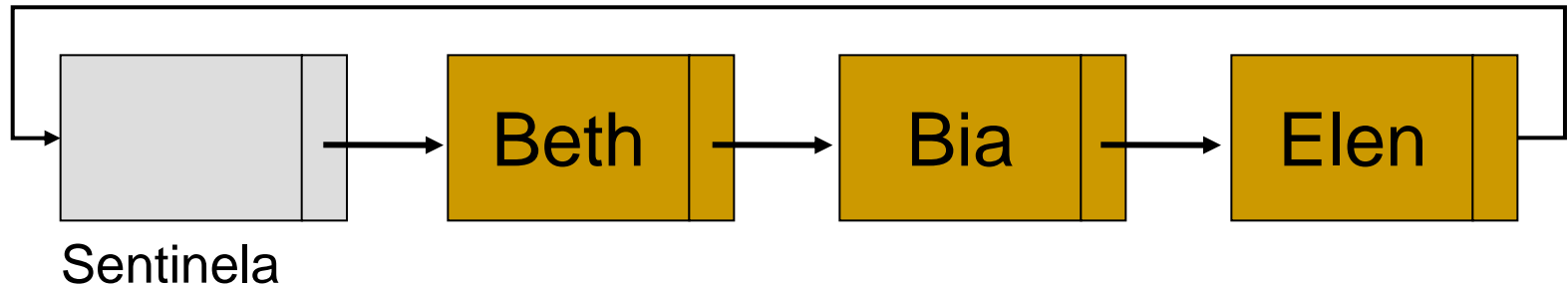


# Lista com nó de cabeçalho

- Possibilidades de uso

- Lista circular

- Não existe mais NULL no fim da lista, eliminando-se o risco de acessar uma posição inválida de memória

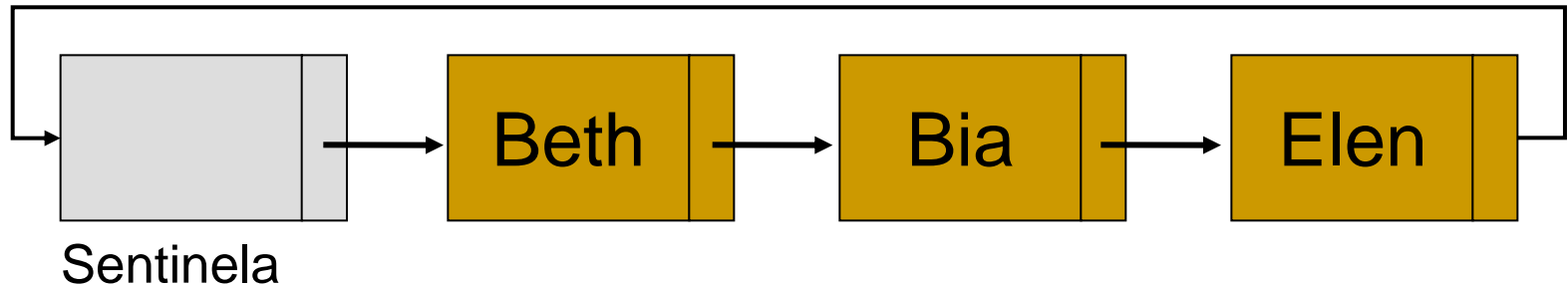


# Lista com nó de cabeçalho

- Possibilidades de uso

- Lista circular

- Como saber qual é o último elemento da lista?
    - Coloca-se o elemento de busca na sentinela!



---

# Lista com nó de cabeçalho

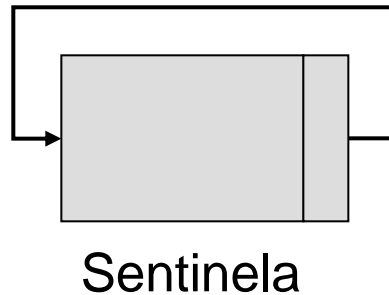
- Possibilidades de uso
  - Lista circular
    - Como representar a lista vazia?





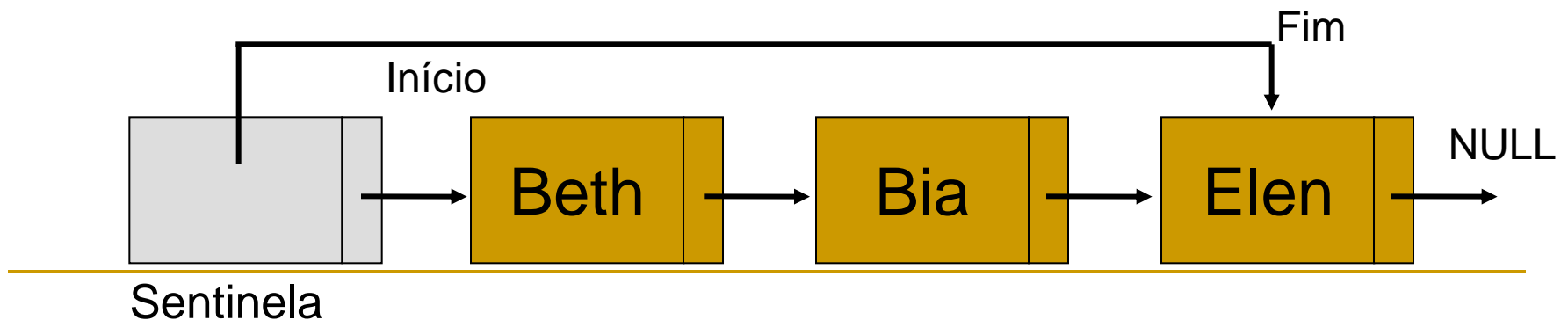
# Lista com nó de cabeçalho

- Possibilidades de uso
  - Lista circular
    - Como representar a lista vazia?



# Lista com nó de cabeçalho

- Possibilidades de uso
  - Informações para uso da **lista como pilha, fila, etc.**
    - Exemplo: em vez de um ponteiro de fim da fila, o nó sentinela pode apontar o fim
      - O campo info do nó sentinela passa a ser um ponteiro
      - Acaba por indicar o início da fila também



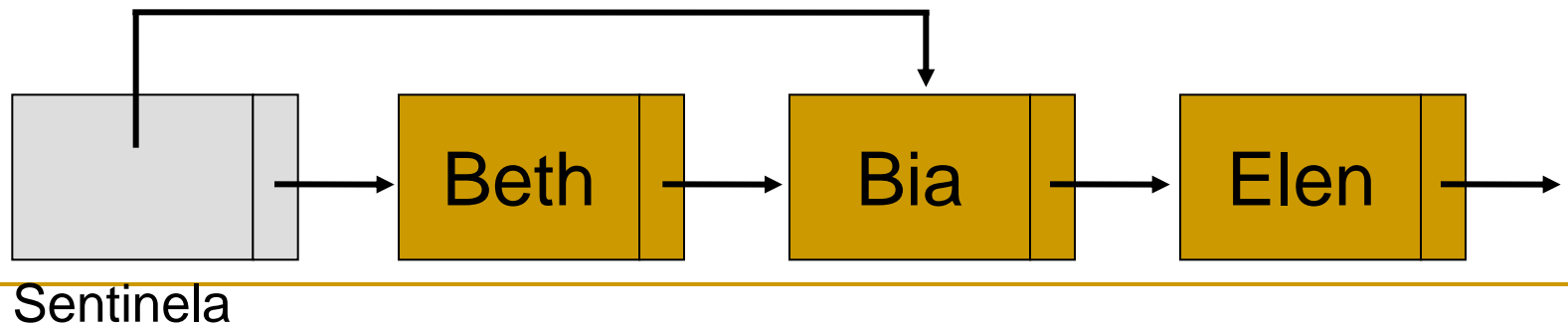
# Lista com nó de cabeçalho

- Possibilidades de uso

- Indica um **nó específico da lista**

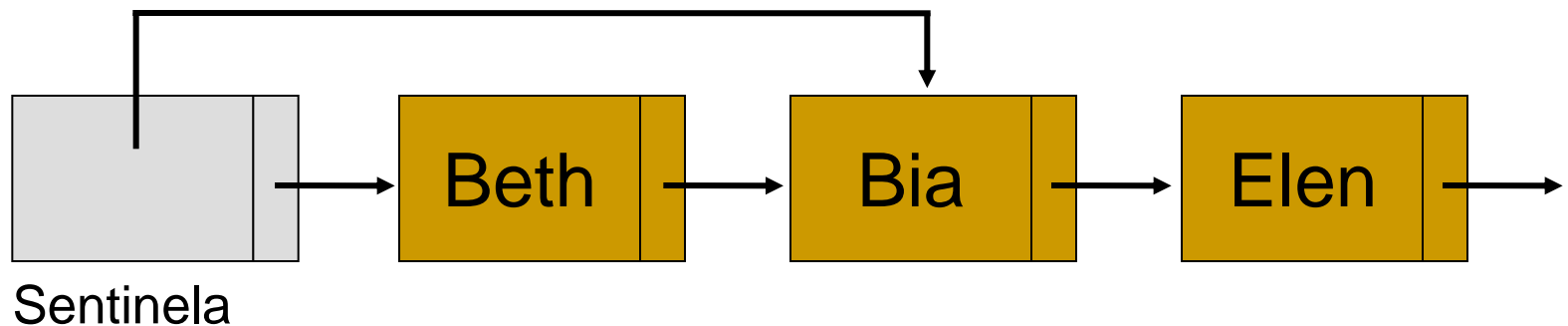
- Por exemplo, em buscas que são constantemente interrompidas

- Verificação de pessoas em ordem alfabética: poupa o esforço de se recomençar ou a necessidade de ter uma variável auxiliar



# Lista com nó de cabeçalho

- Possibilidades de uso
  - Nó sentinela com **ponteiro em seu campo info**
    - Vantagem: acesso possivelmente mais direto e imediato



# Aplicações

- Indique qual a melhor estrutura de dados para modelar cada caso a seguir (pilha, fila, fila circular, fila de prioridade, lista, etc.):
  - atendimento de caixa de banco \_\_\_\_\_
  - retirada e colocação de caixas (uma sobre a outra) em um estoque em um depósito \_\_\_\_\_
  - lanchonete \_\_\_\_\_
  - atendimento em banco com uma fila preferencial \_\_\_\_\_
  - a retirada de pratos empilhados em um armário \_\_\_\_\_
  - Editor para facilitar o acesso a primeira e últimas linhas, próxima linha e linha anterior

---

# Exercício: união de listas ordenadas

## ■ Situação

- ❑ Você recebeu duas listas dinâmicas e encadeadas ordenadas, de tamanhos possivelmente diferentes
  - ❑ Você não pode se dar ao luxo de contar com mais memória do que a já usada pelas listas, pois a aplicação rodando é crítica
- 
- Implemente uma função em C que faça a união das duas listas, mantendo a ordenação

---

# Exercícios

1) Implemente uma função que verifique se duas listas encadeadas são iguais. Duas listas são consideradas iguais se têm a mesma seqüência de elementos. O protótipo da função deve ser dado por:

**int igual (Lista\* l1, Lista\* l2);**

2) Implemente uma função que crie uma cópia de uma lista encadeada. O protótipo da função deve ser dado por:

**Lista\* copia (Lista\* l);**

---

# Exercícios

- Dada uma lista ordenada L1 encadeada alocada dinamicamente, escreva as operações:
  - Verifica se L1 está ordenada ou não (a ordem pode ser crescente ou decrescente)
  - Faça uma cópia da lista L1 em uma outra lista L2
  - Faça uma cópia da Lista L1 em L2, eliminando elementos repetidos
  - inverta L1 colocando o resultado em L2
  - inverta L1 colocando o resultado na própria L1
  - intercale L1 com a lista L2, gerando a lista L3 (L1, L2 e L3 ordenadas)



# Exercícios

- Escreva um programa que gera uma lista L2, a partir de uma lista L1 dada, em que cada registro de L2 contém dois campos de informação
  - *elem* contém um elemento de L1, e *count* contém o número de ocorrências deste elemento em L1
- Escreva um programa que elimine de uma lista L dada todas as ocorrências de um determinado elemento (L ordenada)
- Assumindo que os elementos de uma lista L são inteiros positivos,
  - escreva um programa que informe os elementos que ocorrem mais e menos em L (forneça os elementos e o número de ocorrências correspondente)

---

# Exercício: Cadastro de Almas do Paraíso

- Para facilitar a vida de Deus, você foi selecionado para construir um sistema que automatize o cadastro de almas que chegam ao paraíso. O sistema deve armazenar:
-

- 
- Um conjunto em ordem alfabética de todos os nomes das almas (não há limite para o número de almas de pessoas), o tempo (dia, mês e ano) que cada uma esteve na Terra e o número de amigos que teve;
-

- 
- os familiares de cada alma
    - (se a alma foi casada, então se devem armazenar nomes e idades da esposa/marido e filhos, vivos ou mortos;
    - se não for casada, nomes e idades dos pais, vivos ou mortos);
  
  - para cada familiar (vivo ou morto) de cada alma, um conjunto com nomes e espécies de animais de estimação que possuiu que entrarão no paraíso para ficarem com ele (enquanto houver espaço no paraíso para almas de animais, os primeiros do conjunto poderão entrar)
-

- 
- Como muita gente morre por dia, a organização e eficiência são a chave do sucesso do sistema! Portanto, conceitos de TAD e de complexidade de algoritmos são essenciais.



- 
- (a) Desenhe a estrutura de dados que desenvolveu (blocos de memória, ponteiros, etc.), justificando todas as suas escolhas.
  - 
  - (b) Declare todas as estruturas de dados necessárias para a implementação do sistema anterior.
  - 
  - (c) Implemente uma sub-rotina recursiva que verifique se um familiar de uma alma qualquer está vivo ou morto.
-