Listas Encadeadas

Listas Encadeadas:

- Definição: uma Lista Encadeada também é um tipo de Lista Linear, ou seja, um TAD que possui elementos que mantém uma relação de ordem entre eles

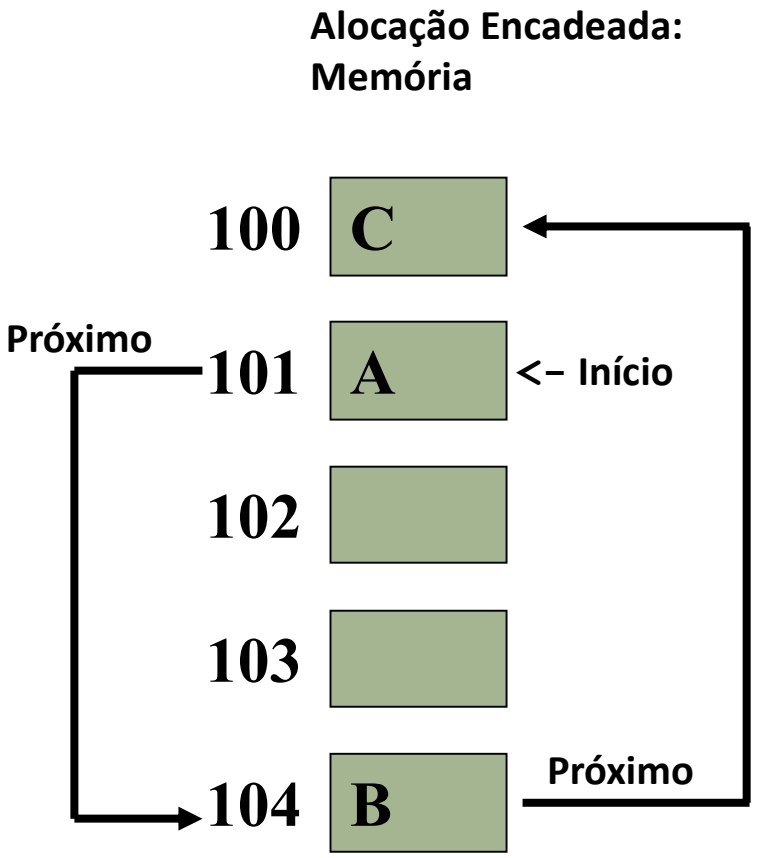
- Entretanto, em uma lista encadeada a alocação da memória, necessária ao armazenamento dos elementos, é feita dinamicamente e sob demanda

- Isso significa que o próximo elemento da lista pode ser posicionado em qualquer lugar livre da memória

- Portanto, uma lista encadeada precisa de um mecanismo para garantir o sequenciamento de seus elementos

Listas Encadeadas:

- Estrutura baseada na alocação dinâmica de memória:
  - Elementos **não** estão necessariamente em posições de memória adjacentes
  - Seqüência "**lógica**"
  - Para manter essa sequencia, uma lista encadeada faz uso de **ponteiros**



Listas Encadeadas:

- Estrutura dinâmica, criada vazia
- Os elementos são chamados de “nós”
- Estrutura homogênea: os nós são todos do mesmo tipo
- Tamanho da lista é dado pelo número de nós da lista
- Condiciona o crescimento da lista à disponibilidade de memória
- Os nós não estão em seqüência na memória
- Os conceitos de ponteiros para registros e registros contendo ponteiros são muito úteis
  - Cada nó guarda: informações (info) e o endereço do próximo nó (prox)
  - Mantemos um ponteiro para o início da lista
  - O prox do último nó deve apontar para NULL

Nós de Encadeamento

- Seja o nó definido abaixo:

```
typedef struct _node {
   /* info */
   struct _node *prox;
} node;
```

- Tal registro contém dois atributos:
  - “info” (principal): qualquer tipo conhecido (int, float, char, struct aluno, struct pessoa, etc.)
  - “prox”: ponteiro para um registro do tipo “node”. A idéia é poder apontar para o próximo registro da lista

Tipo de Encadeamento:

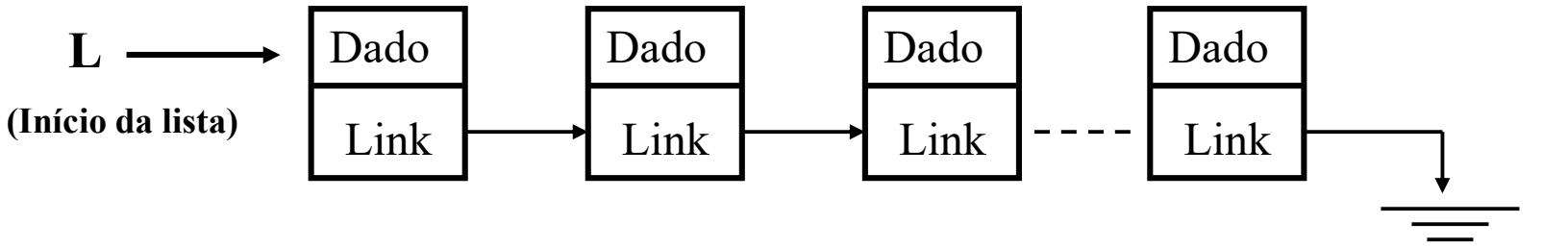
- Geralmente, listas encadeadas são apresentadas em dois tipos:

- Simplesmente Encadeada: contém um elo com o próximo item de dado (usa apenas um ponteiro);

- Duplamente Encadeada: contém elos tanto com o elemento anterior quanto com o próximo elemento da lista (uso de duas variáveis do tipo ponteiro).

Encadeamento Simples:

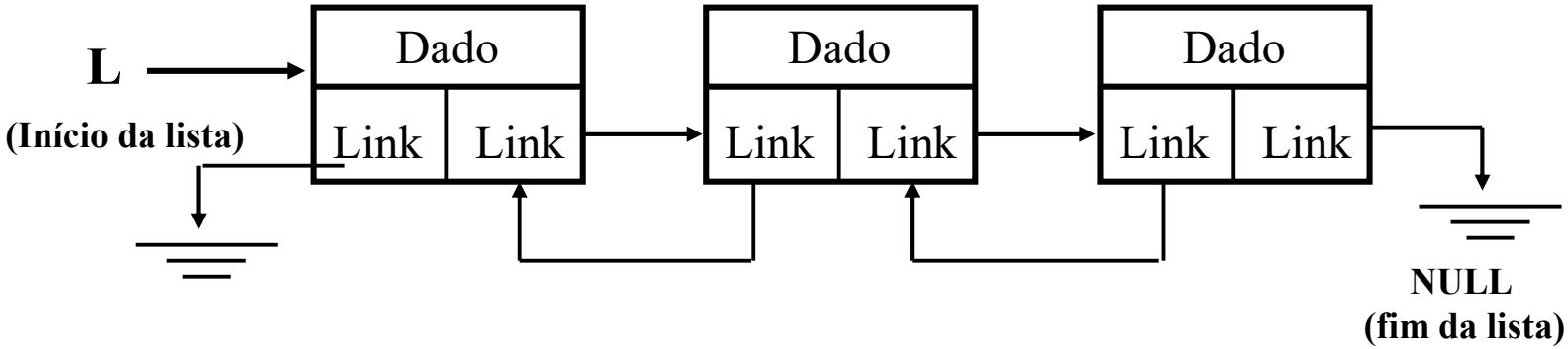
- Representação:



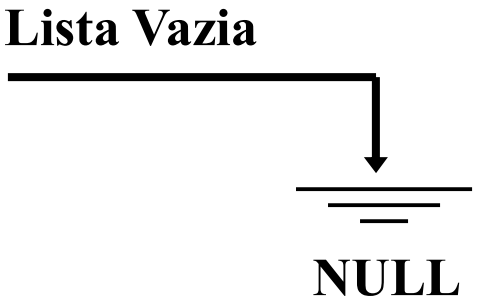
Lista Singularmente Encadeada

Encadeamento Duplo:

- Representação:



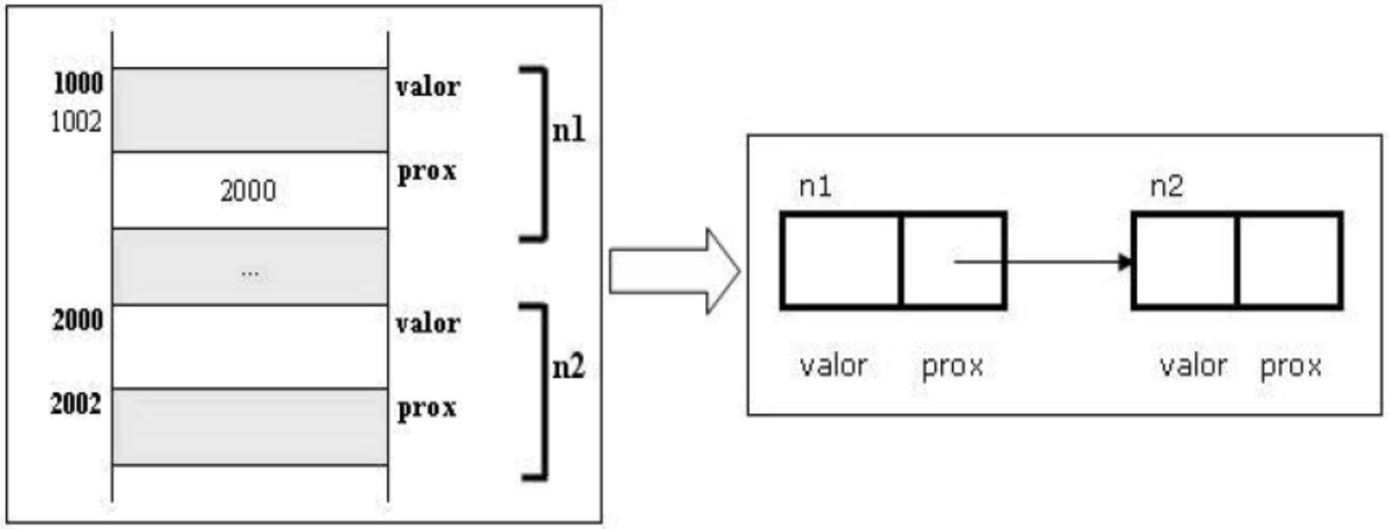
Lista Duplamente Encadeada



Encadeamento

- Discutiremos o funcionamento do encadeamento simples

- A instrução `n1.prox = &n2` cria o encadeamento entre n1 e n2:

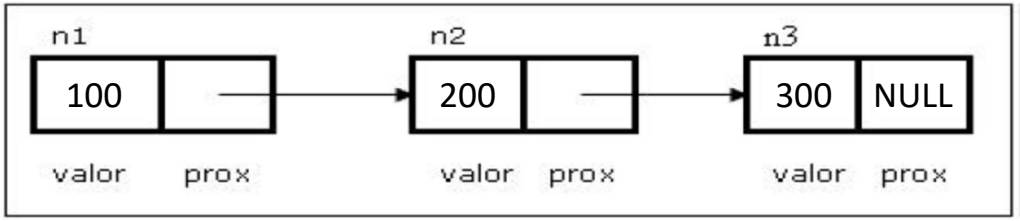


- O efeito de encadeamento pode, então, ser utilizado para a criação de uma lista, através da adição de novos nós

Encadeamento

- Considere o código abaixo:

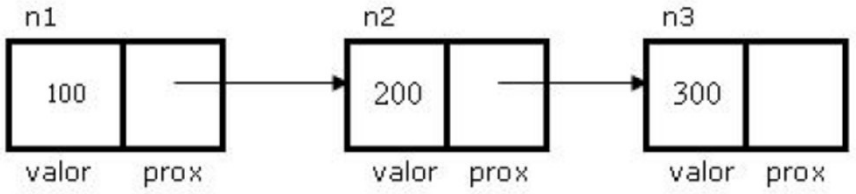
```
node n1, n2, n3;
int i;
n1.valor = 100;
n2.valor = 200;
n3.valor = 300;
n1.prox = &n2;
n2.prox = &n3;
n3.prox = NULL;
```



- Qual seria o resultado das instruções abaixo?

```
a.  i = (n1.prox) -> valor;
b.  i = n1.prox.valor;
c.  n1.prox = n2.prox;
```

Respostas



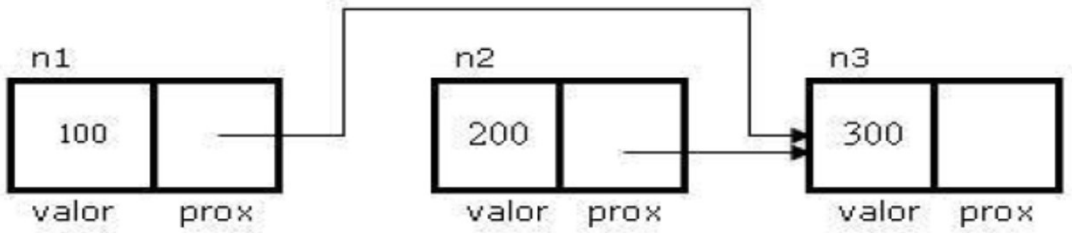
a. `i = (n1.prox) -> valor;`

b. `i = n1.prox.valor;`

INCORRETO!

a. `n1.prox = n2.prox;`

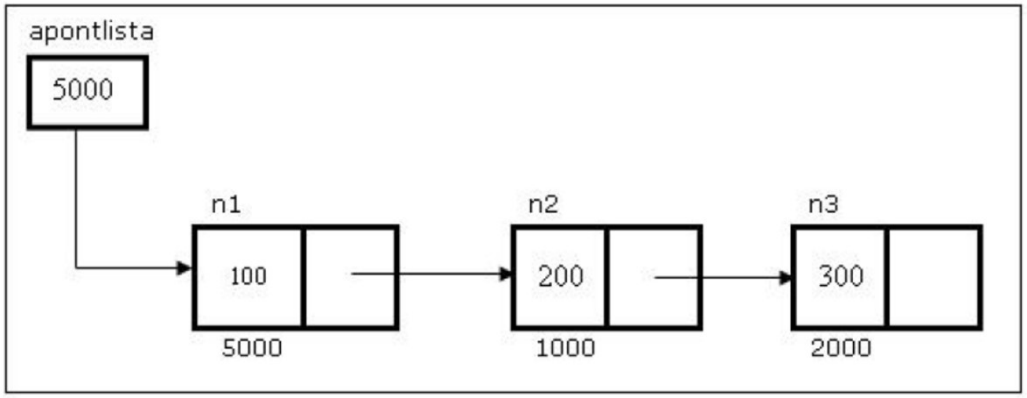
Remove o elemento n2 da lista



Considerações

- Em geral, associamos a uma lista encadeada pelo menos um ponteiro para o primeiro elemento.
  - Ponteiro pra o início da lista: ponteiro cabeçalho (header pointer)

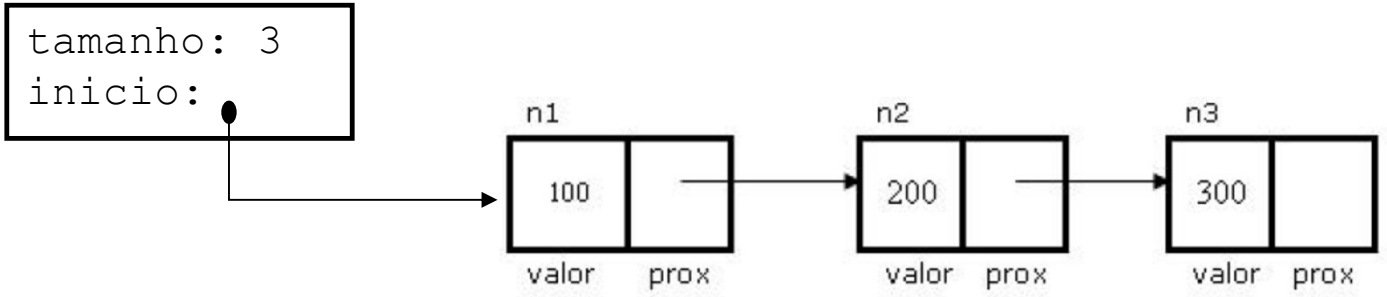
```
node *apontlista;
.
.
.
apontlista = &n1;
```



Considerações

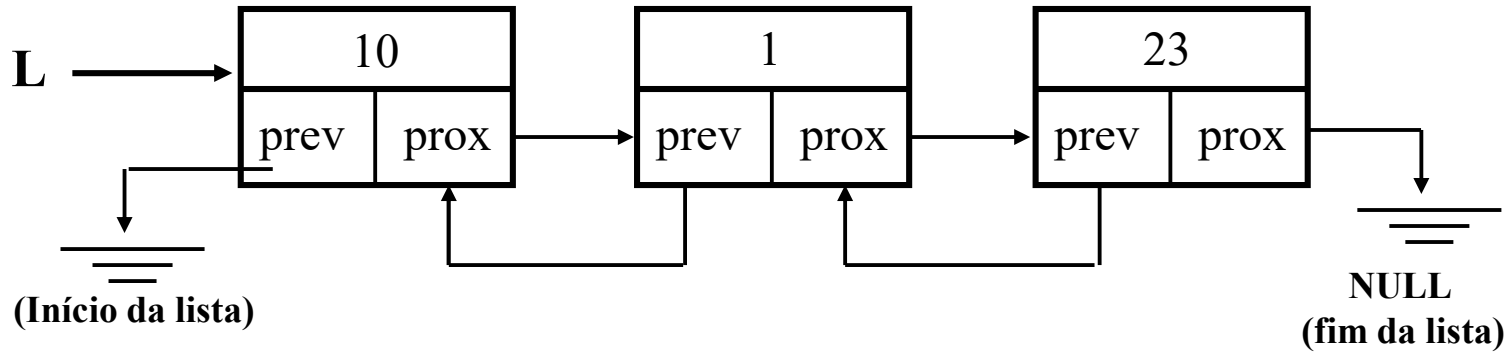
- Muitas vezes é interessante armazenar outras informações sobre a lista. Ex: tamanho da lista
- Para isso, podemos utilizar um “nó” (estrutura) para indicar o início da lista, mas que contenha as informações desejadas

```
typedef struct _headerNode {
   int tamanho;
   node *inicio;
} headerNode;
```



Listas Encadeadas: operações

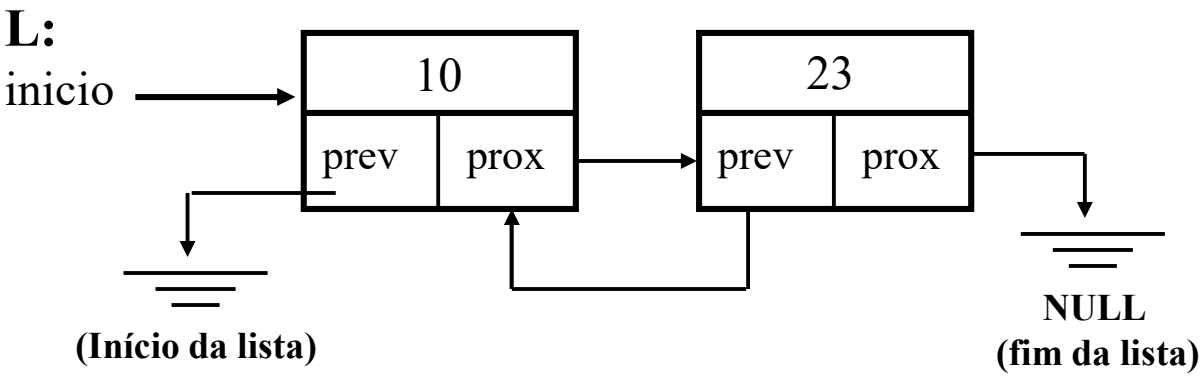
- Como exemplo, consideraremos uma lista não-ordenada, não-circular e duplamente encadeada



Busca em Listas Encadeadas:

- Este método encontra um elemento de chave *k* na lista L, através de busca linear simples, retornando um ponteiro para o elemento encontrado, ou NULL, caso *k* não seja encontrado em L

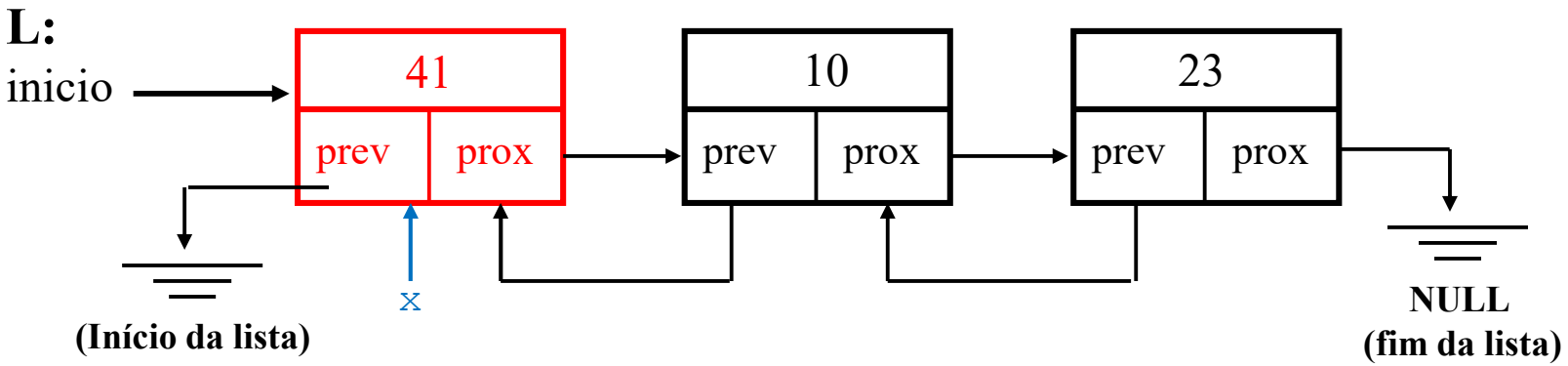
```
Busca(L, k)
  x = L->inicio
  while x <> NULL && x->dado <> k
    x = x->prox
  return x
```



Inserção em Listas Encadeadas:

- Este método recebe um elemento *x* (do tipo node), para o qual a chave *k* já foi acertada, e o coloca na primeira posição da lista (não ordenada)

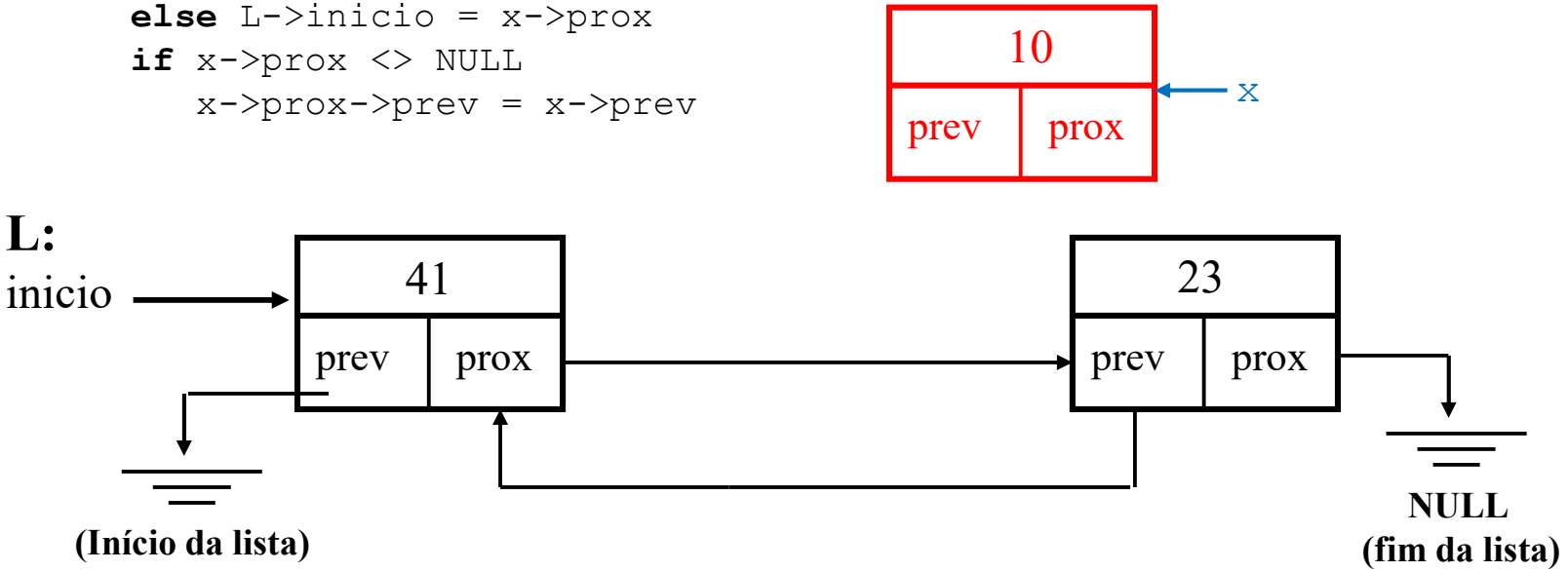
```
Insert(L, x)
  x->prox = L->inicio
  if L->inicio <> NULL
     L->inicio->prev = x
  L->inicio = x
  x->prev = NULL
```



Remoção de Listas Encadeadas:

- Usa ponteiro *x* para um elemento da lista e, então, o remove pela movimentação dos ponteiros de seus vizinhos. Note que, o elemento a ser removido deve ser encontrado antes (Busca)

```
Delete(L, x)
  if x->prev <> NULL
     x->prev->prox = x->prox
  else L->inicio = x->prox
  if x->prox <> NULL
     x->prox->prev = x->prev
```

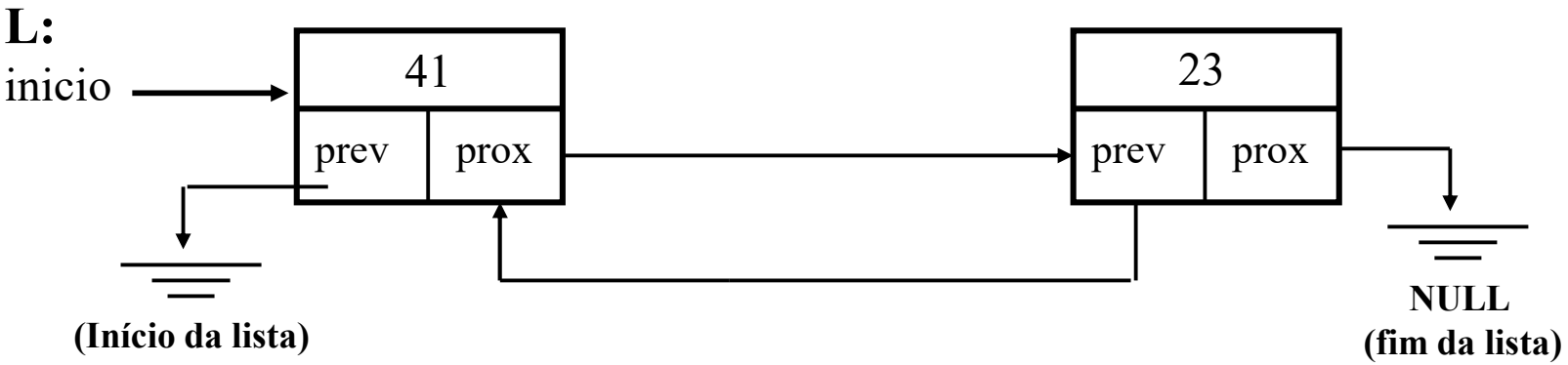


Remoção de Listas Encadeadas:

- Usa ponteiro *x* para um elemento da lista e, então, o remove pela movimentação dos ponteiros de seus vizinhos. Note que, o elemento a ser removido deve ser encontrado antes (Busca)

```
Delete(L, x)
  if x->prev <> NULL
     x->prev->prox = x->prox
  else L->inicio = x->prox
  if x->prox <> NULL
     x->prox->prev = x->prev
```

Observação: para simplificar esta função, e remover as verificações de guarda, considere o uso de um *sentinela*



Listas Encadeadas: operações

- Observações:
  - As operações sobre listas encadeadas são executadas em tempo $O(1)$
  - Entretanto, considere que, a remoção geralmente demanda uma busca ($\Theta(n)$, no pior caso)
  - O uso de sentinelas, em geral, aumenta a clareza, não a eficiência (considerar o uso extra de memória)

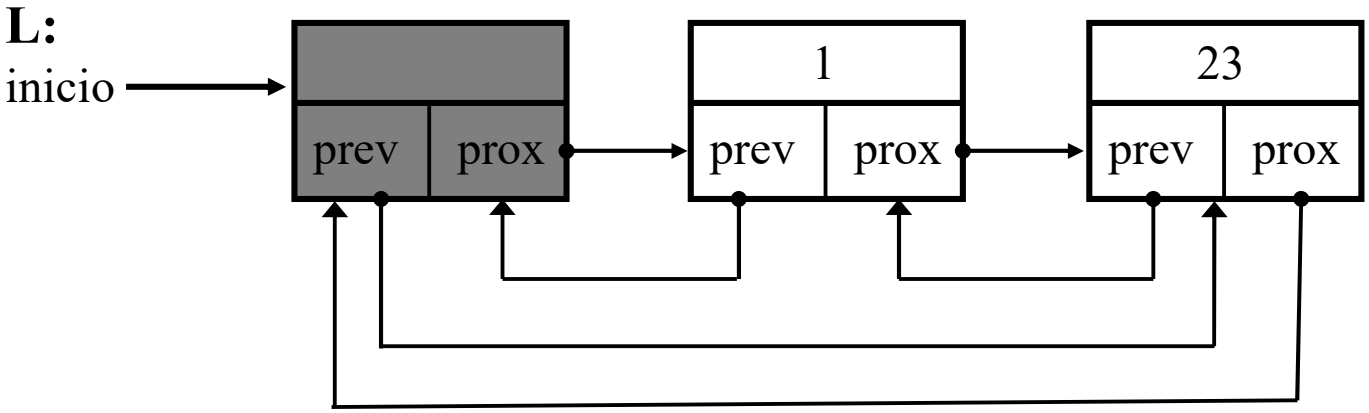
```
Busca(L, k)
  x = L->inicio
  while x <> NULL && x->dado <> k
    x = x->prox
  return x
```

```
Insert(L, x)
  x->prox = L->inicio
  if L->inicio <> NULL
     L->inicio->prev = x
  L->inicio = x
  x->prev = NULL
```

```
Delete(L, x)
  if x->prev <> NULL
     x->prev->prox = x->prox
  else L->inicio = x->prox
  if x->prox <> NULL
     x->prox->prev = x->prev
```

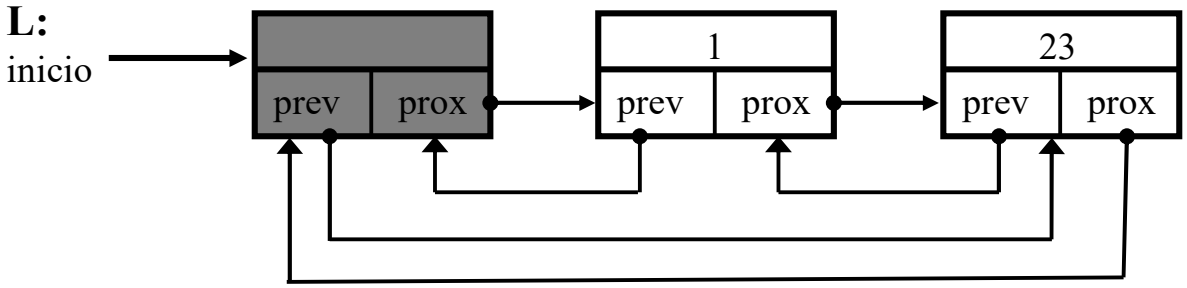
Listas Encadeadas

- Exercício: reescreva as funções anteriores, para o uso com sentinela



Listas Encadeadas

- Exercício: reescreva as funções anteriores, para o uso com sentinela




```
Busca(L, k)
  x = L->inicio->prox
  while x <> L->inicio && x->dado <> k
    x = x->prox
  return x
```

```
Insert(L, x)
  x->prox = L->inicio->next
  L->inicio->prox->prev = x
  L->inicio->prox = x
  x->prev = L->inicio
```

```
Delete'(L, x)
  x->prev->prox = x->prox
  x->prox->prev = x->prev
```

TADs sobre Listas Ligadas:

- Desenvolvimento Assistido 01 (laboratório):

  - Vamos aplicar os conceitos de listas ligadas na implementação de uma TAD **Lista** convencional, em que elementos são inseridos em ordem crescente e removidos de acordo com um seu valor de chave

  - Depois faremos a mesma coisa para um TAD **Fila**

  - O conceito funcional de Listas e Filas pode ser encontrado nos slides anteriores