Estrutura de Dados I

Estrutura de dados: Lista

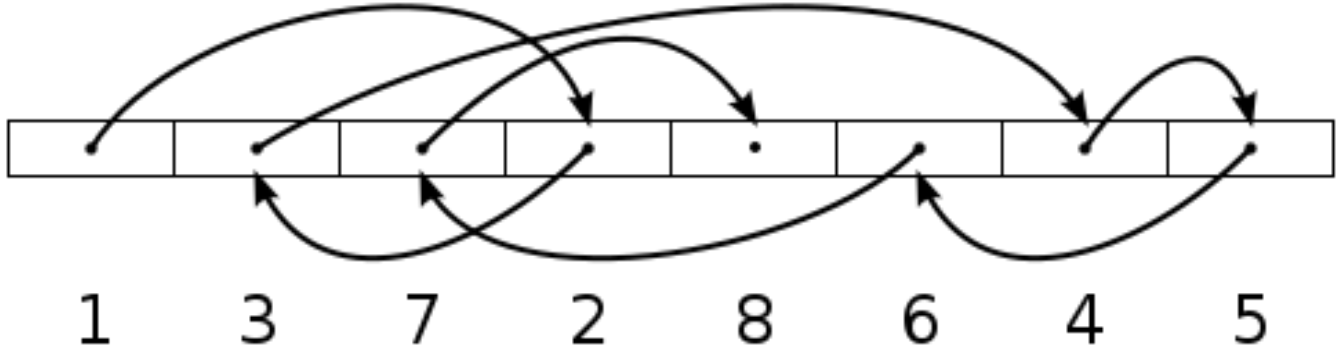
Prof. Rodrigo Minetto

Universidade Tecnológica Federal do Paraná

Sumário

Introdução

Um **vetor** ocupa um espaço contíguo na memória e nos permite acessar qualquer um de seus elementos a partir do ponteiro para o primeiro elemento. Dizemos que o vetor é uma estrutura que possibilita o **acesso aleatório** aos elementos.



Introdução

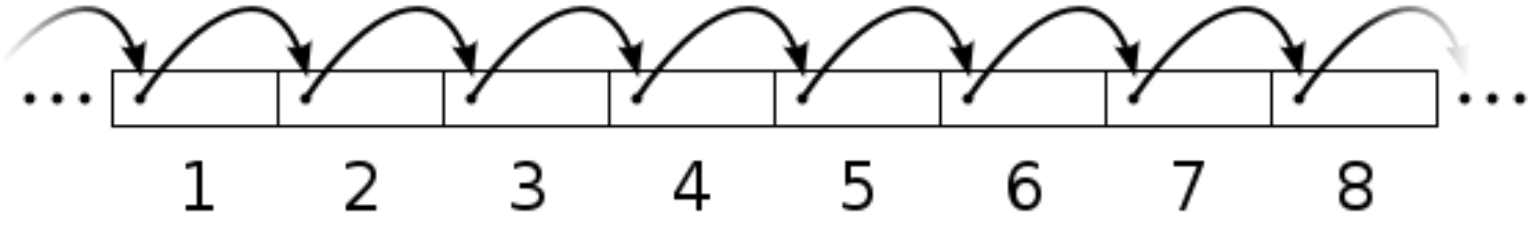
Problemas: o vetor é uma **estrutura não flexível**. **1)** O que acontece se o número de elementos que precisamos armazenar **exceder** a dimensão do vetor? **2)** O que acontece se o número de elementos for muito **inferior** a sua dimensão? Não existe maneira barata (computacionalmente) para alterar a dimensão do vetor em tempo de execução.

Introdução

Solução: utilizar estruturas de dados que **cresçam** conforme precisamos **armazenar** novos elementos (e **diminuam** conforme precisamos **retirar** elementos armazenados anteriormente). Estas estruturas são chamadas **dinâmicas** e armazenam cada um dos elementos por alocação dinâmica.

Introdução

No entanto, em uma **lista encadeada**, a cada inserção alocamos um espaço de memória para armazená-lo mas **não** há garantia que eles ocuparão um **espaço de memória contíguo**. **Desvantagem**: não temos acesso aleatório aos elementos da lista (somente sequêncial):

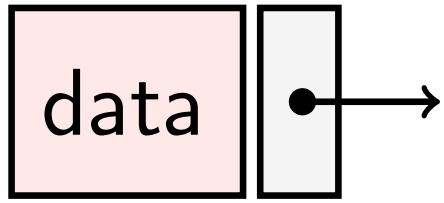


Sumário

Estrutura

Cada elemento da lista é em geral chamado de **nó** (do inglês **node**). Um nó é representado por uma estrutura que contém a informação armazenada e um ponteiro.

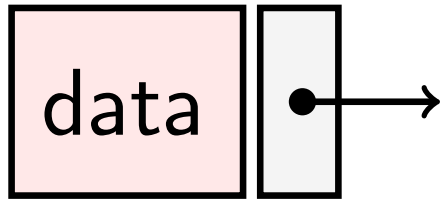
```
struct node {
    int data;
    struct node* next;
};
```



Estrutura

Embora não seja essencial, é uma boa prática de programação definir o tipo da estrutura. Assim **List** $*\ell$ passa a representar **struct node** $*\ell$

```
typedef struct node {
    int data;
    struct node* next;
} List;
```

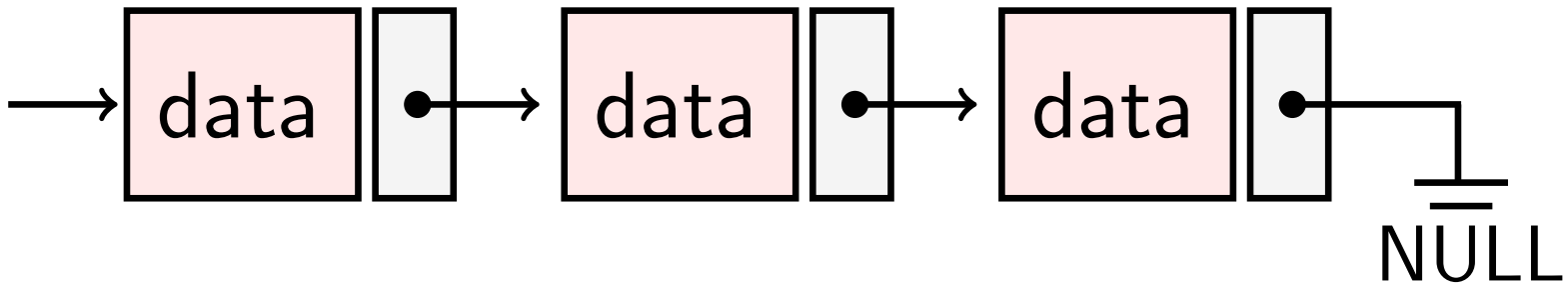


Estrutura

Para percorrer todos os elementos da lista devemos explicitamente guardar o seu **encadeamento**, que é realizado ao atribuir ao ponteiro de cada nó o endereço do próximo elemento.

Estrutura

A lista é representada por um ponteiro para o primeiro elemento (ou nó). O último elemento tem como próximo um ponteiro para **NULL**, e sinaliza, que não existe um próximo elemento.



Tipo abstrato de dados

Tipo abstrato de dados **lista** (**interface**)

create: inicializa uma estrutura de dados lista

insert: adiciona um nó na lista (qual posição?)

remove: remove um nó da lista

size: retorna o número de nós na lista

print: imprime todos os nós da lista

destroy: remove todos os nós da lista

Sumário

Listas encadeadas (criar)

O endereço de uma lista encadeada é definido como ponteiro para o primeiro elemento. Na variante **lista sem cabeça**, uma lista vazia é definida como um ponteiro para **NULL**

```
1: List* create (void)
---------------------------------
2:     return NULL;
3:
```

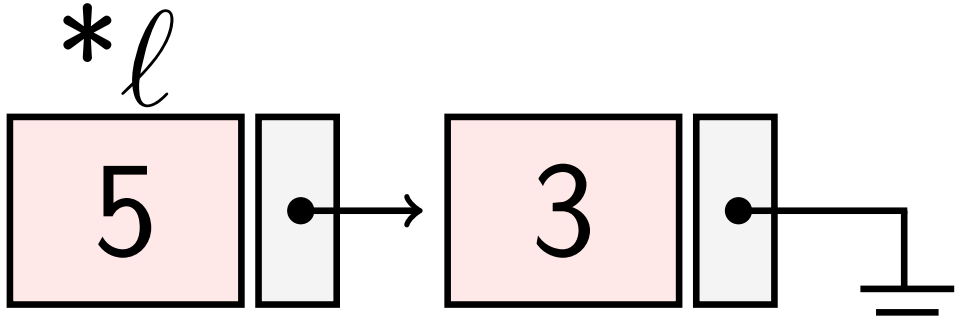
Sumário

Listas encadeadas (inserir)

A função de inserção mais simples insere um novo elemento no **início** da lista. No entanto, a nova cabeça da lista passa a apontar para o novo primeiro elemento.

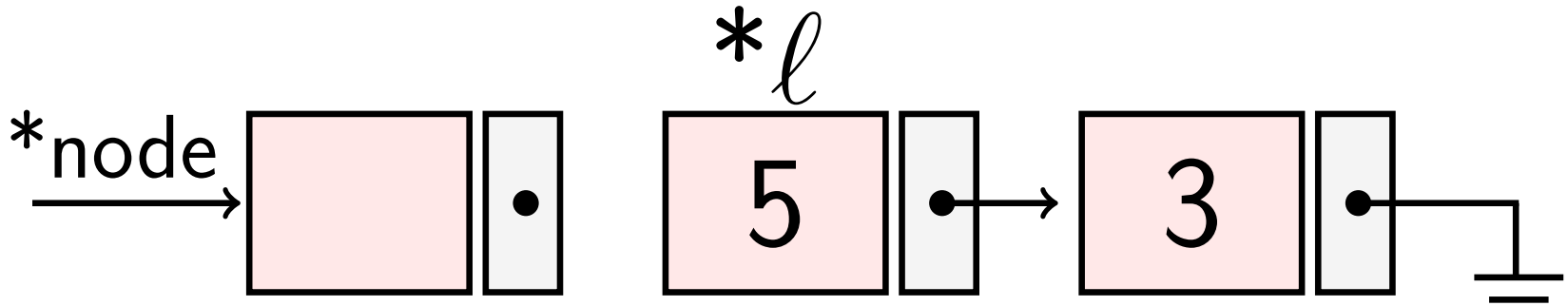
```
1: List* insert (List *ℓ, int elem)
---------------------------------------------
2:     List *node = (List*)malloc(sizeof(List));
3:     node→data = elem;
4:     node→next = ℓ;
5: return node;
```

Listas encadeadas (inserir)



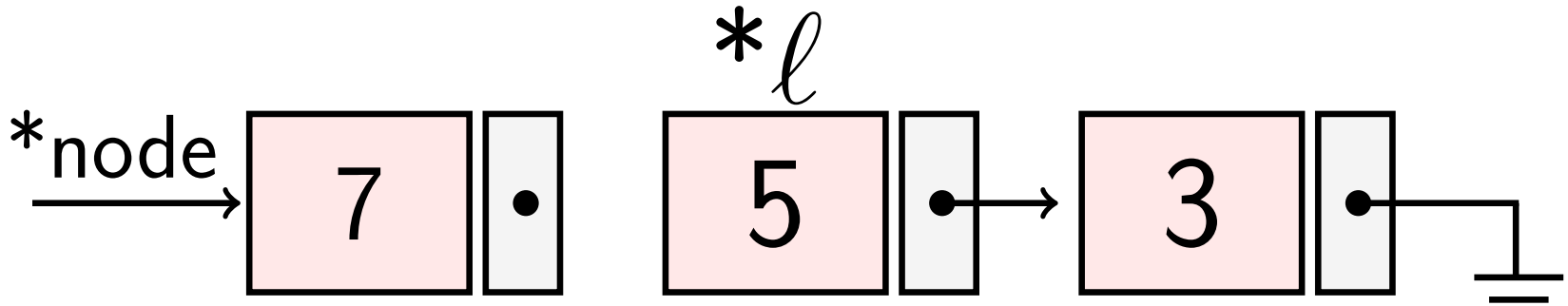

```
1: List* insert (List *ℓ, int elem)
---------------------------------------------------
2:     List *node = (List*)malloc(sizeof(List));
3:     node→data = elem;
4:     node→next = ℓ;
5: return node;
```

Listas encadeadas (inserir)



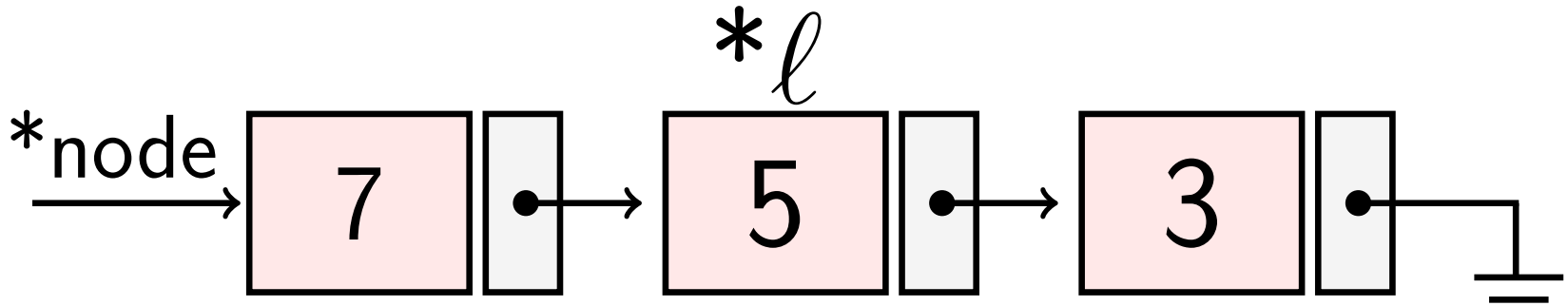

```
1: List* insert (List *ℓ, int elem)
---------------------------------------------------
2:     List *node = (List*)malloc(sizeof(List));
3:     node→data = elem;
4:     node→next = ℓ;
5: return node;
```

Listas encadeadas (inserir)



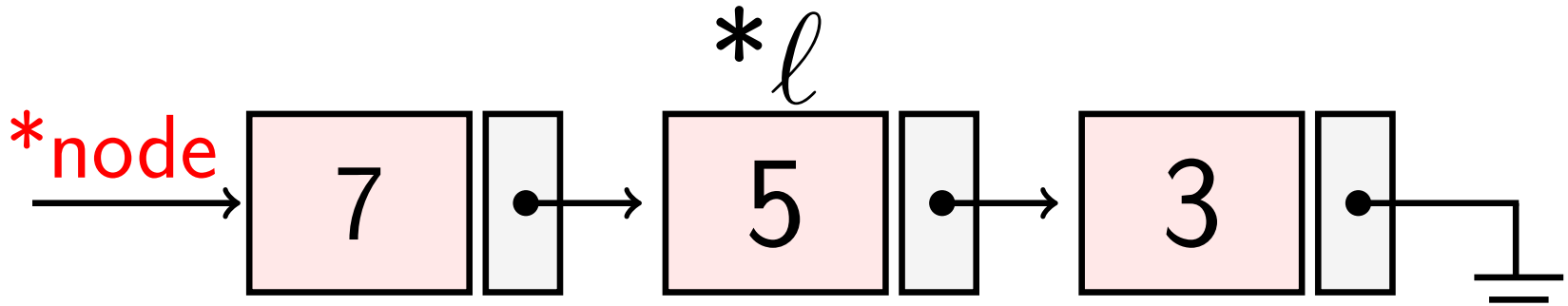

```
1: List* insert (List *ℓ, int elem)
---------------------------------------------
2:     List *node = (List*)malloc(sizeof(List));
3:     node→data = elem;
4:     node→next = ℓ;
5: return node;
```

Listas encadeadas (inserir)




```
1: List* insert (List *ℓ, int elem)
---------------------------------------------
2:     List *node = (List*)malloc(sizeof(List));
3:     node→data = elem;
4:     node→next = ℓ;
5: return node;
```

Listas encadeadas (inserir)



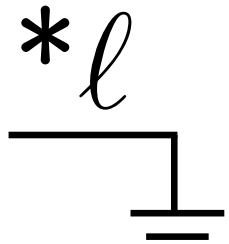

```
1: List* insert (List *ℓ, int elem)
---------------------------------------------
2:     List *node = (List*)malloc(sizeof(List));
3:     node→data = elem;
4:     node→next = ℓ;
5: return node;
```

Sumário

Listas encadeadas (inserir)

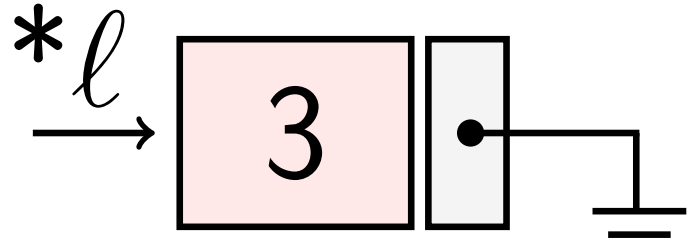
```
1: int main (void)
---------------------------------------
2:     List *ℓ = create ();
3:     ℓ = insert (ℓ, 3);
4:     ℓ = insert (ℓ, 5);
5:     ℓ = insert (ℓ, 7);
6: return 0
```

Listas encadeadas (inserir)



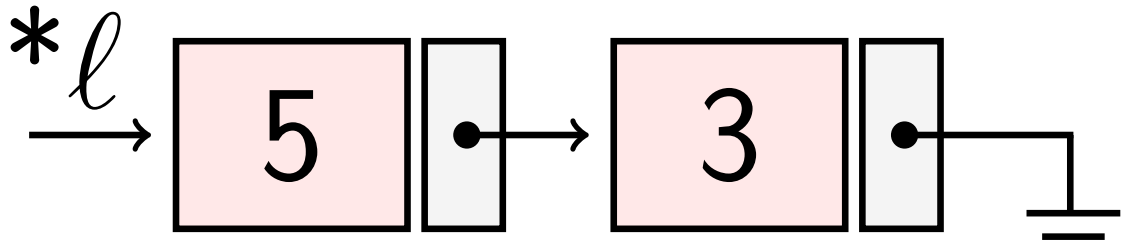

```
1: int main (void)
---------------------------------------
2:     List *ℓ = create ();
3:     ℓ = insert (ℓ, 3);
4:     ℓ = insert (ℓ, 5);
5:     ℓ = insert (ℓ, 7);
6: return 0
```

Listas encadeadas (inserir)



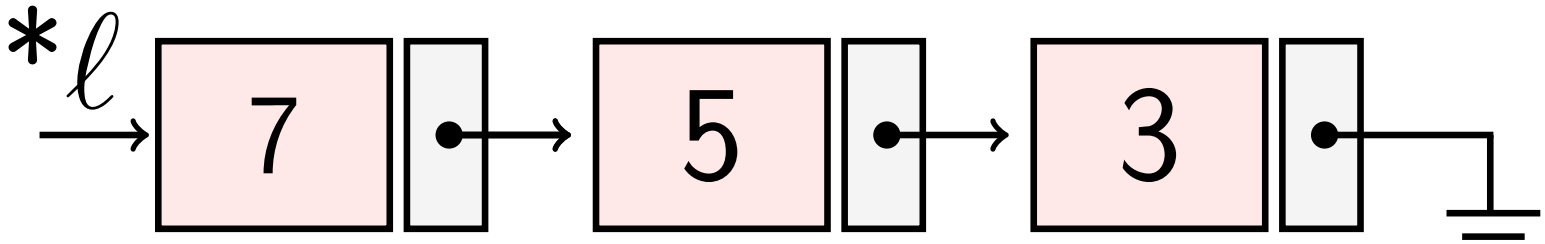

```
1: int main (void)
---------------------------------------
2:     List *ℓ = create ();
3:     ℓ = insert (ℓ, 3);
4:     ℓ = insert (ℓ, 5);
5:     ℓ = insert (ℓ, 7);
6: return 0
```

Listas encadeadas (inserir)




```
1: int main (void)
---------------------------------------
2:     List *ℓ = create ();
3:     ℓ = insert (ℓ, 3);
4:     ℓ = insert (ℓ, 5);
5:     ℓ = insert (ℓ, 7);
6: return 0
```

Listas encadeadas (inserir)




```
1: int main (void)
---------------------------------------
2:     List *ℓ = create ();
3:     ℓ = insert (ℓ, 3);
4:     ℓ = insert (ℓ, 5);
5:     ℓ = insert (ℓ, 7);
6: return 0
```

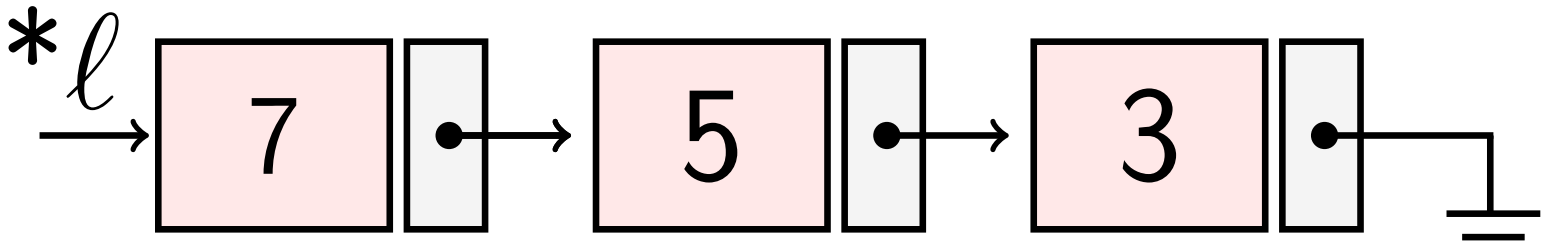
Sumário

Listas encadeadas (imprimir)

A função para **imprimir** usa o operador diferente ($\neq$) para determinar o final da lista. Observe a importância que o último ponteiro seja **NULL** (fonte comum de erros).

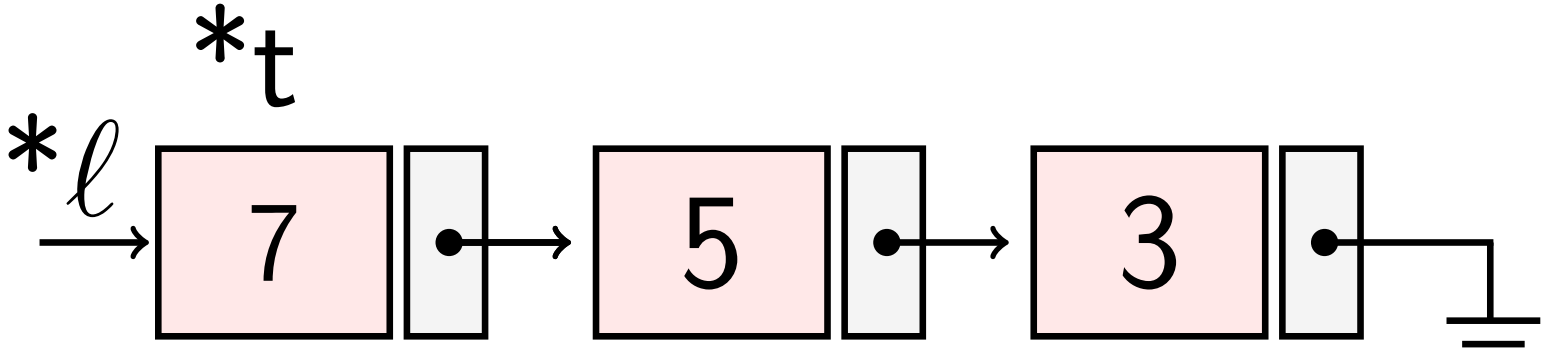
```
1: void print (List *ℓ)
2:     List *t;
3:     for (t = ℓ; t ≠ NULL; t = t→next) {
4:         printf("Data: %d", t→data);
5:     }
6: }
```

Listas encadeadas (imprimir)



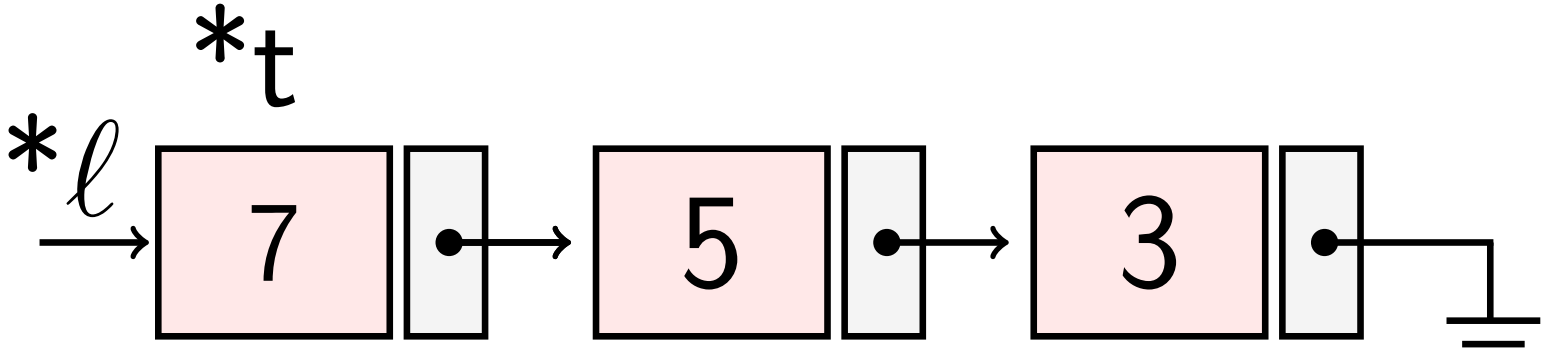

```
1: void print (List *ℓ)
2:     List *t;
3:     for (t = ℓ; t ≠ NULL; t = t→next) {
4:         printf("Data: %d", t→data);
5:     }
6: }
```

Listas encadeadas (imprimir)



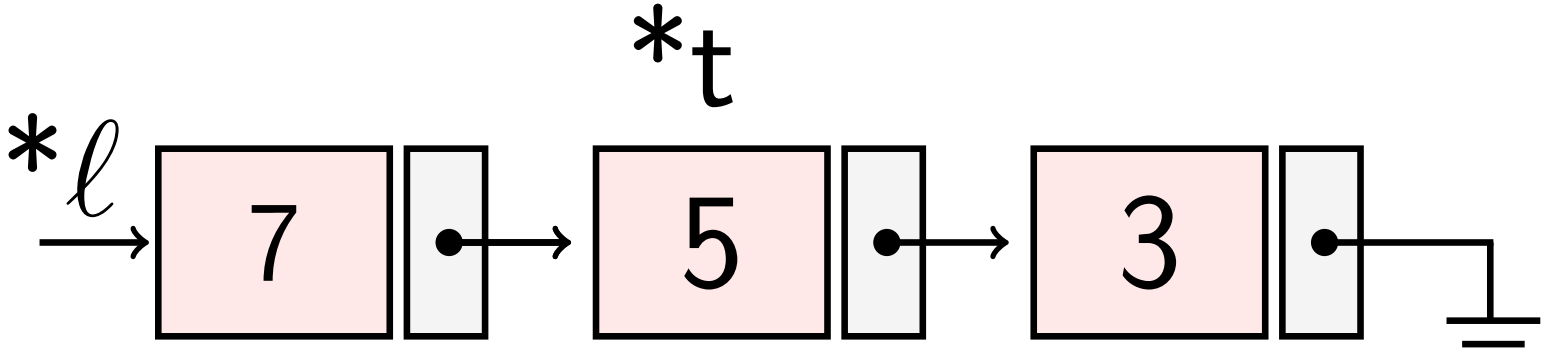

```
1: void print (List *ℓ)
2:     List *t;
3:     for (t = ℓ; t ≠ NULL; t = t→next) {
4:         printf("Data: %d", t→data);
5:     }
6: }
```

Listas encadeadas (imprimir)



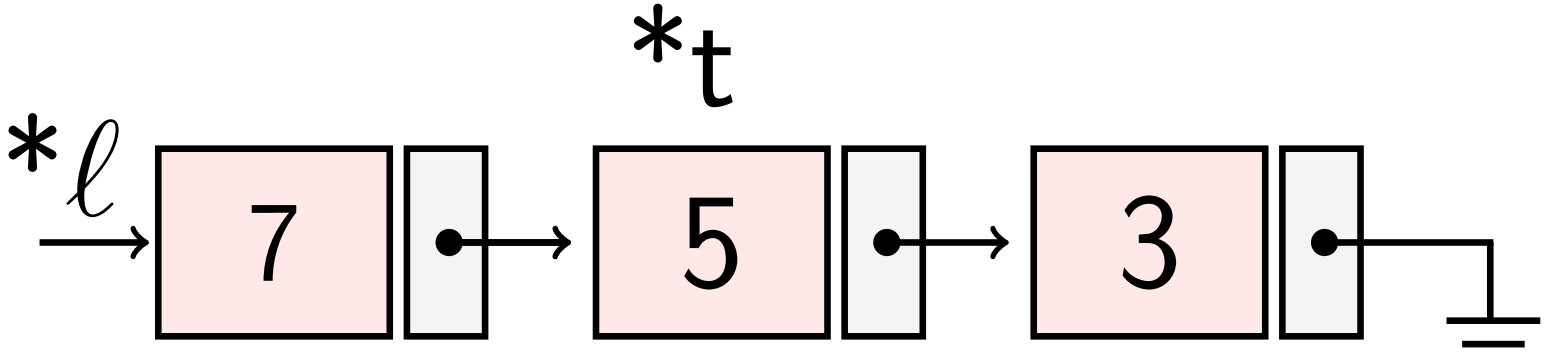

```
1: void print (List *ℓ)
2:     List *t;
3:     for (t = ℓ; t ≠ NULL; t = t→next) {
4:         printf("Data: %d", t→data);
5:     }
6: }
```

Listas encadeadas (imprimir)



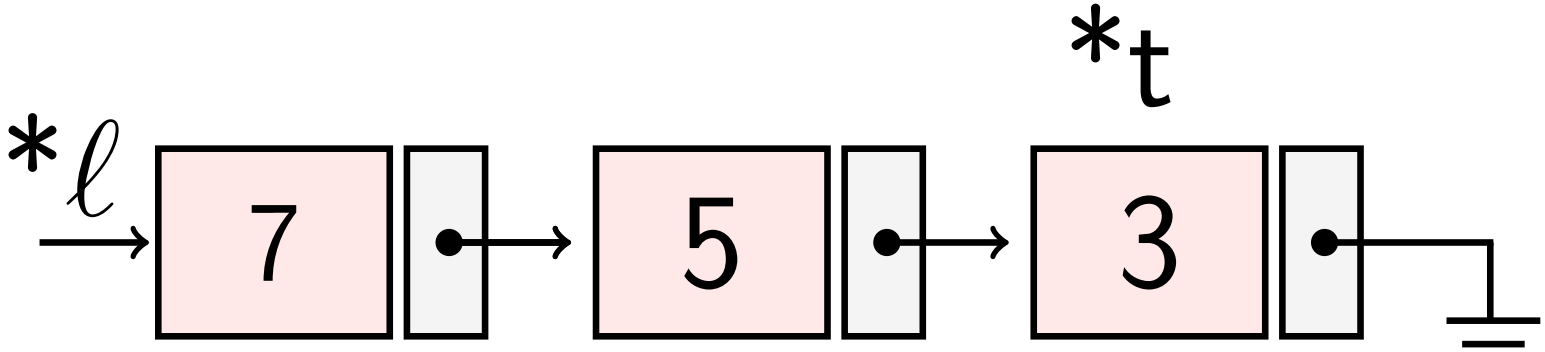

```
1: void print (List *ℓ)
2:     List *t;
3:     for (t = ℓ; t ≠ NULL; t = t→next) {
4:         printf("Data: %d", t→data);
5:     }
6: }
```

Listas encadeadas (imprimir)



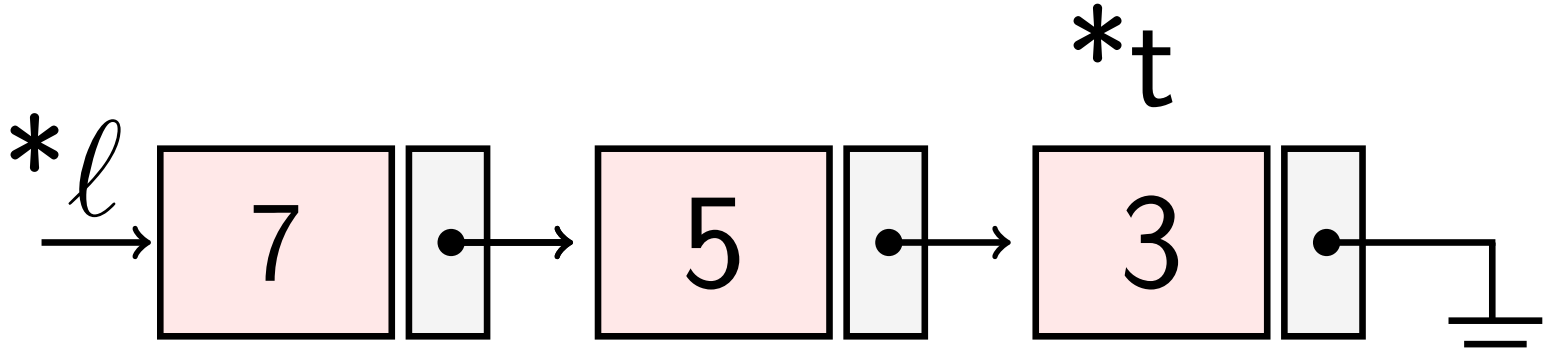

```
1: void print (List *ℓ)
2:     List *t;
3:     for (t = ℓ; t ≠ NULL; t = t→next) {
4:         printf("Data: %d", t→data);
5:     }
6: }
```

Listas encadeadas (imprimir)



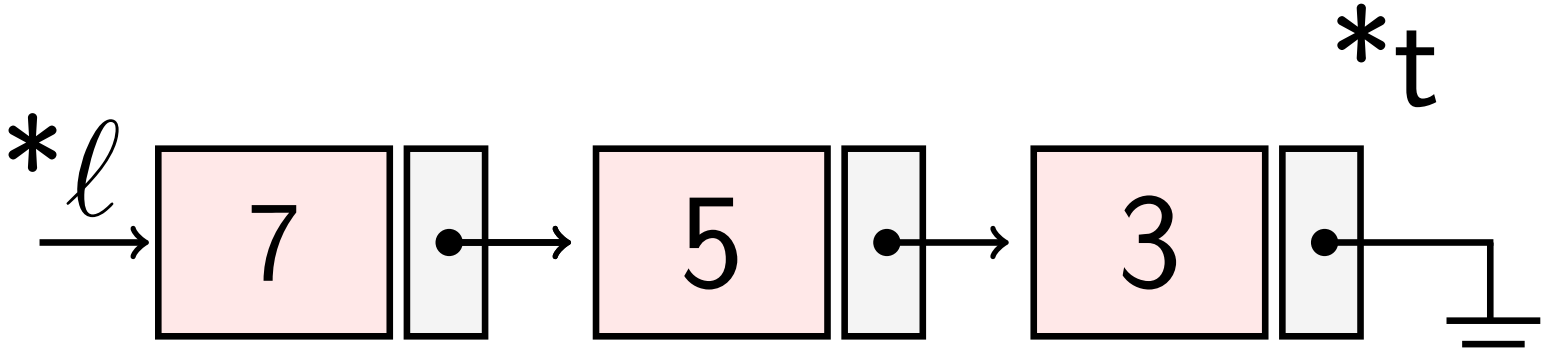

```
1: void print (List *ℓ)
2:     List *t;
3:     for (t = ℓ; t ≠ NULL; t = t→next) {
4:         printf("Data: %d", t→data);
5:     }
6: }
```

Listas encadeadas (imprimir)




```
1: void print (List *ℓ)
2:     List *t;
3:     for (t = ℓ; t ≠ NULL; t = t→next) {
4:         printf("Data: %d", t→data);
5:     }
6: }
```

Listas encadeadas (imprimir)




```
1: void print (List *ℓ)
2:     List *t;
3:     for (t = ℓ; t ≠ NULL; t = t→next) {
4:         printf("Data: %d", t→data);
5:     }
6: }
```

Sumário

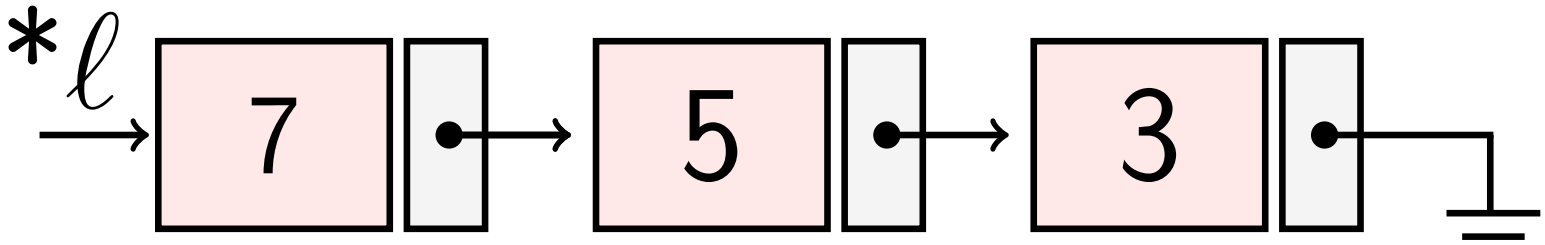
Listas encadeadas (remover)

A função para **remover** um nó da lista precisa buscar o elemento e então descobrir a posição na lista (que influencia na reconexão da lista)

```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL) and (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

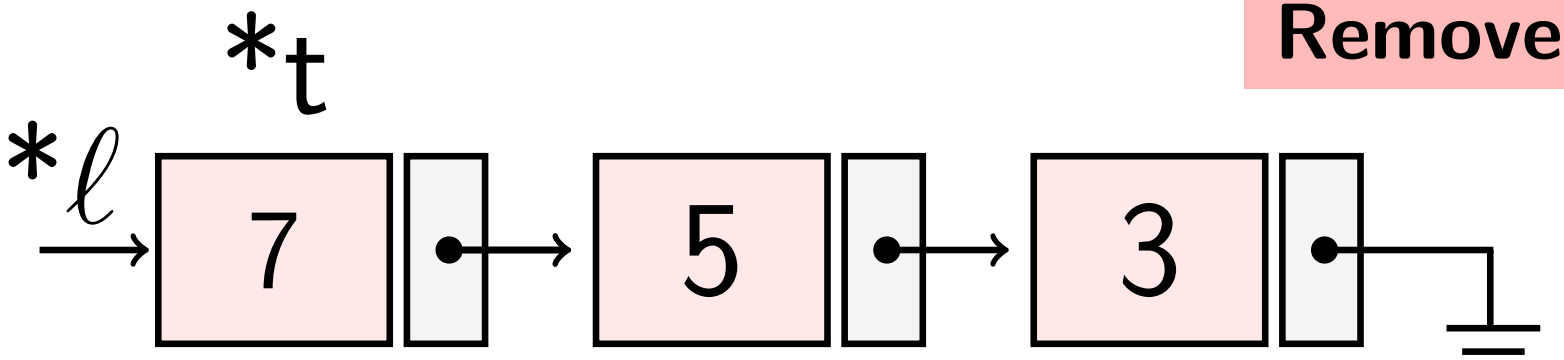
Remover: 5




```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

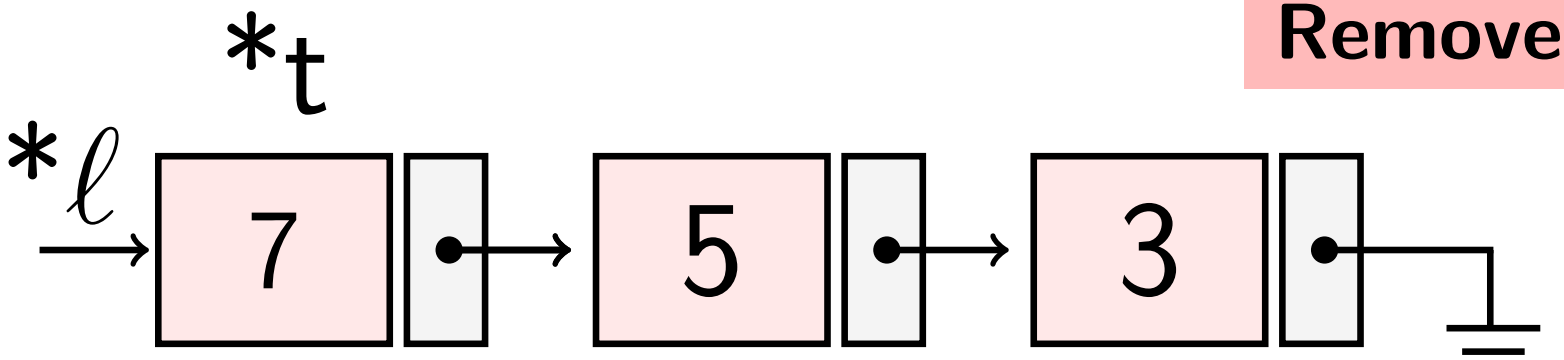
Remover: 5




```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

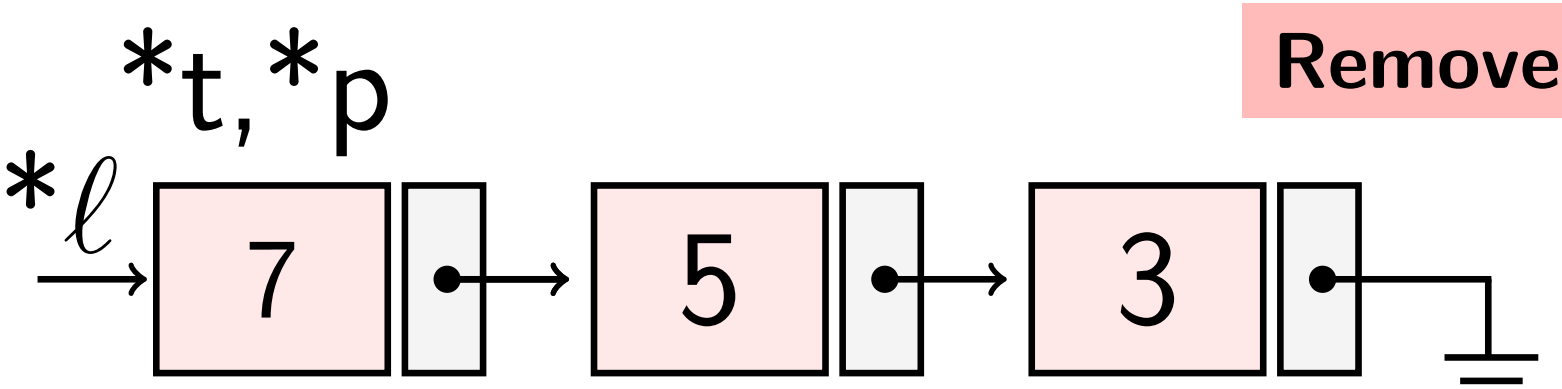
Remover: 5




```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

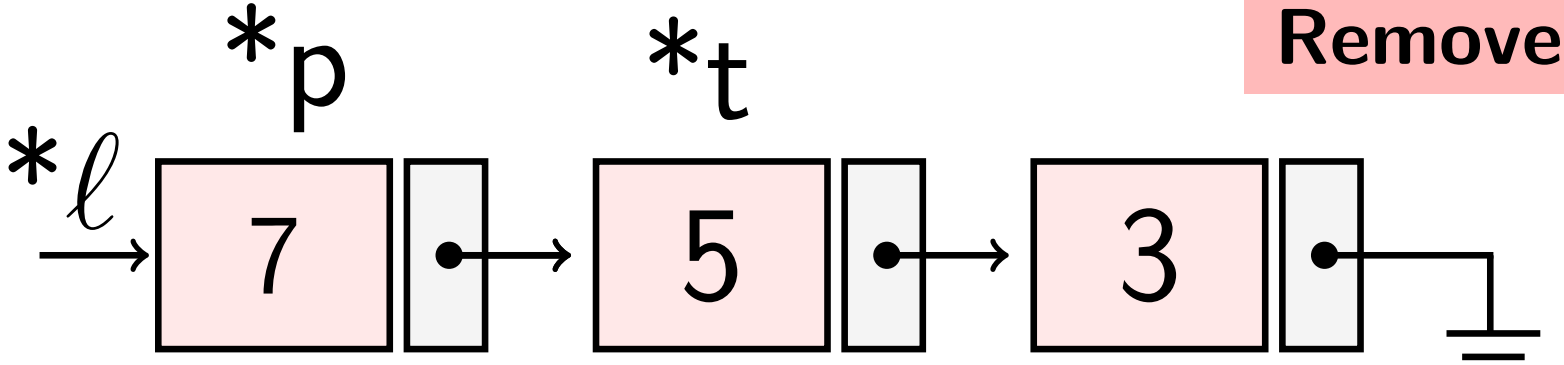
Remover: 5




```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

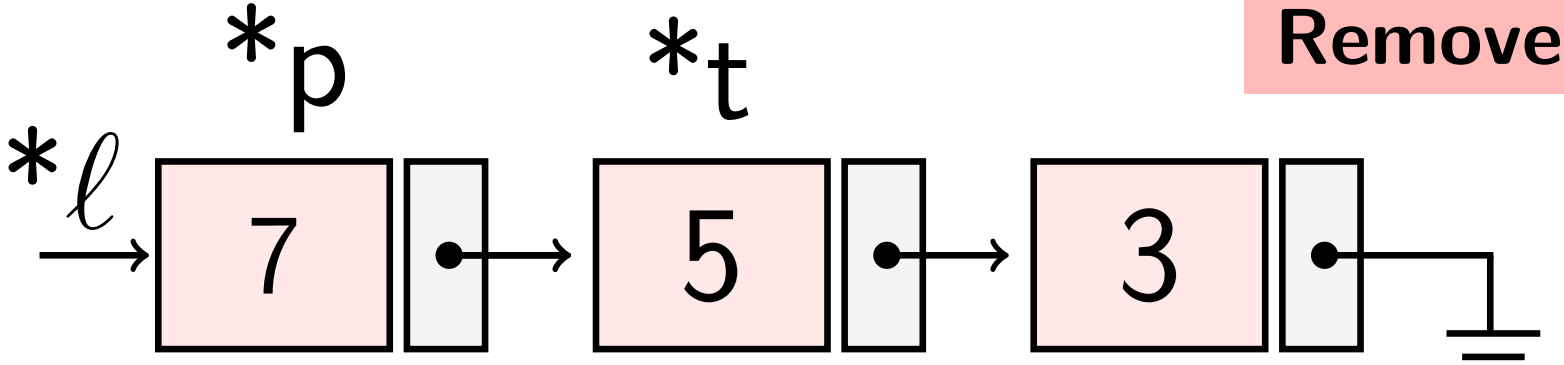
Remover: 5




```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

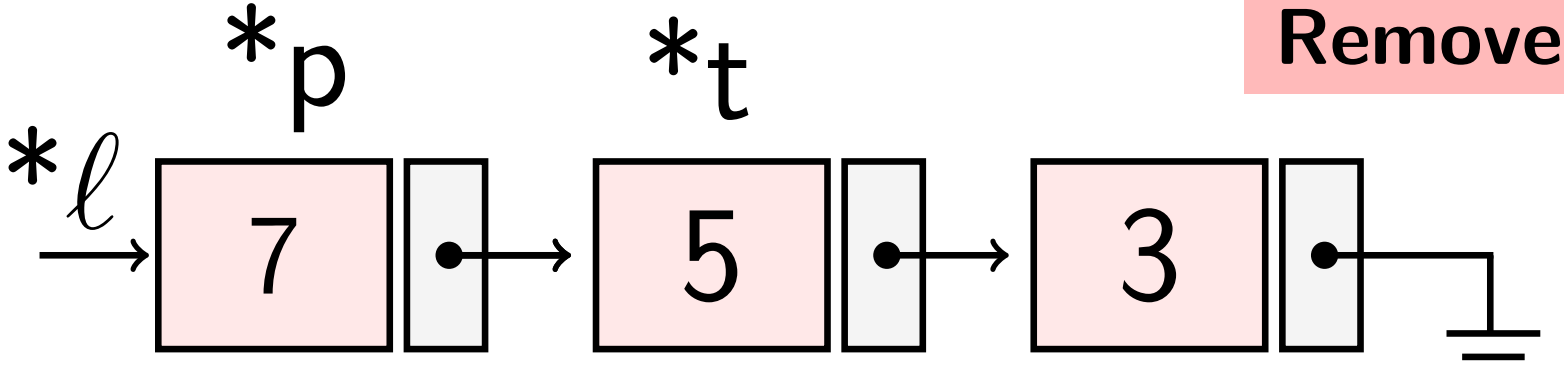
Remover: 5




```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

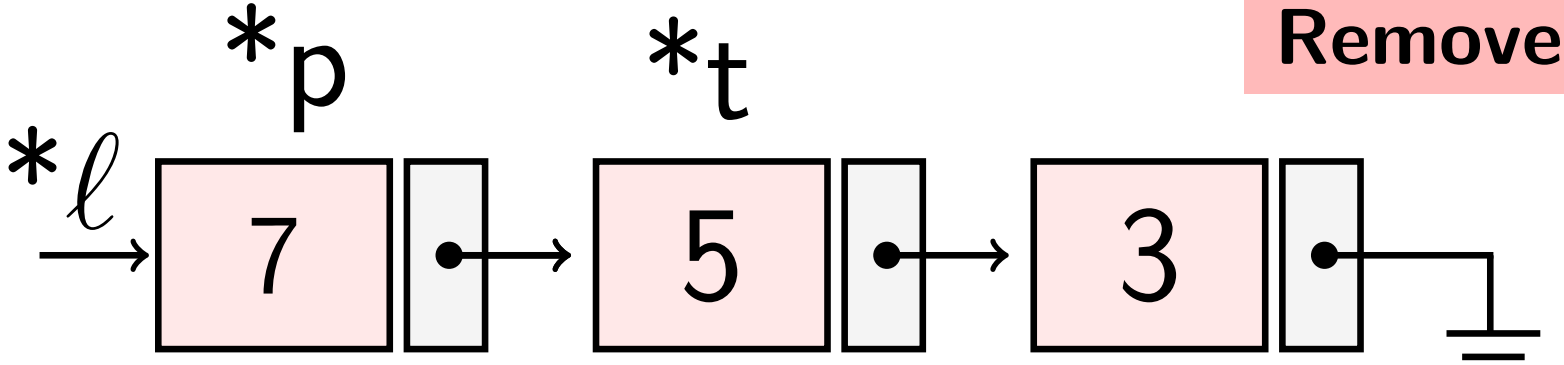
Remover: 5




```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

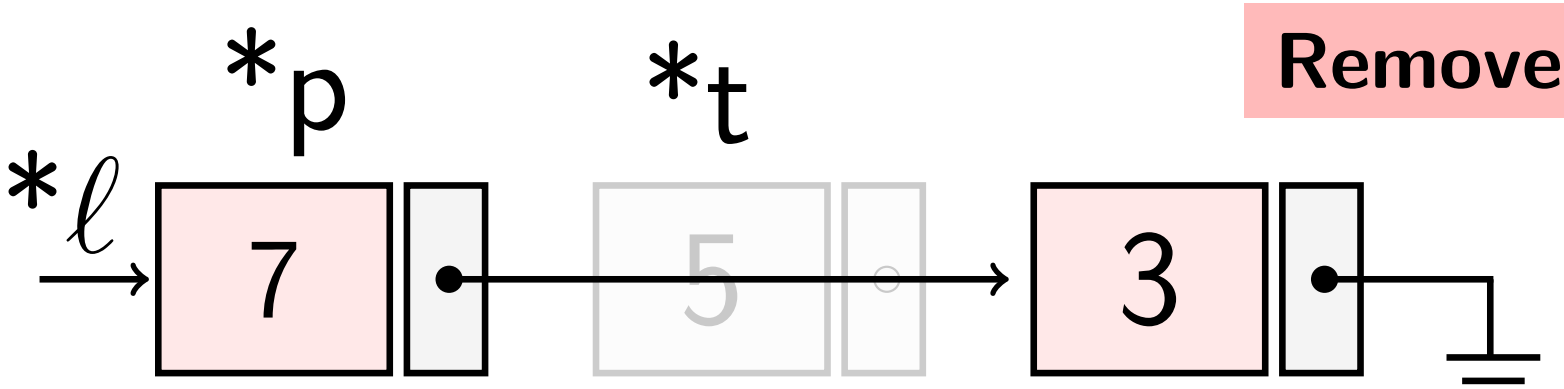
Listas encadeadas (remover)

Remover: 5




```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

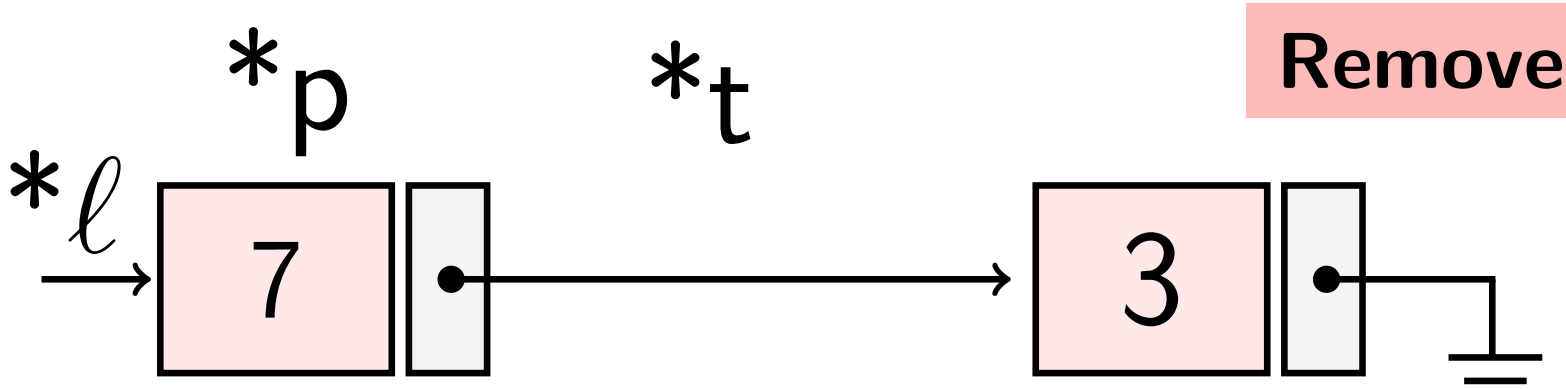
Listas encadeadas (remover)



Remover: 5

```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

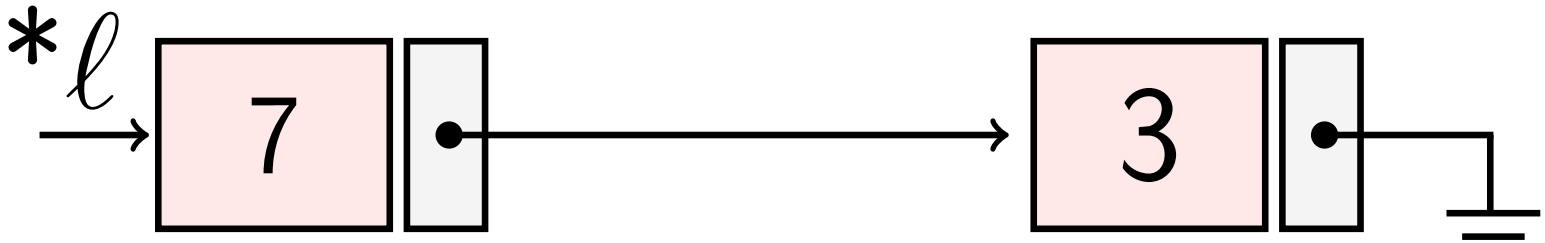


Remover: 5

```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

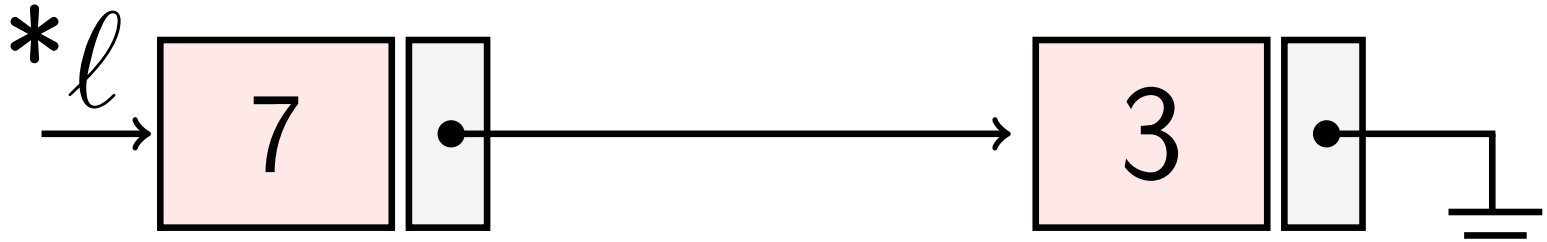
Remover: 5




```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

Remover: 3




```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

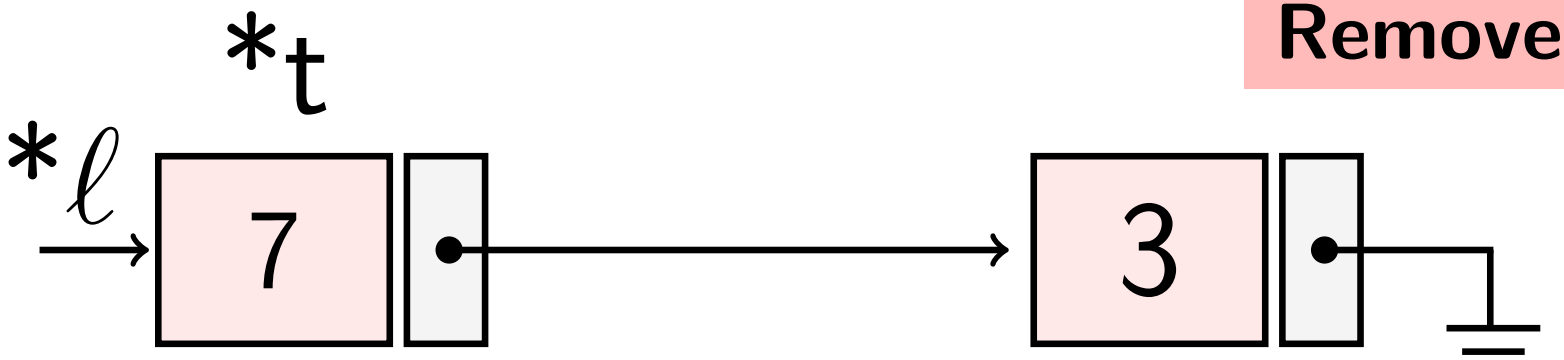
Listas encadeadas (remover)

Remover: 3


```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

Remover: 3




```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

Remover: 3

```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

Remover: 3

```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

Remover: 3

```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

Remover: 3

```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

Remover: 3

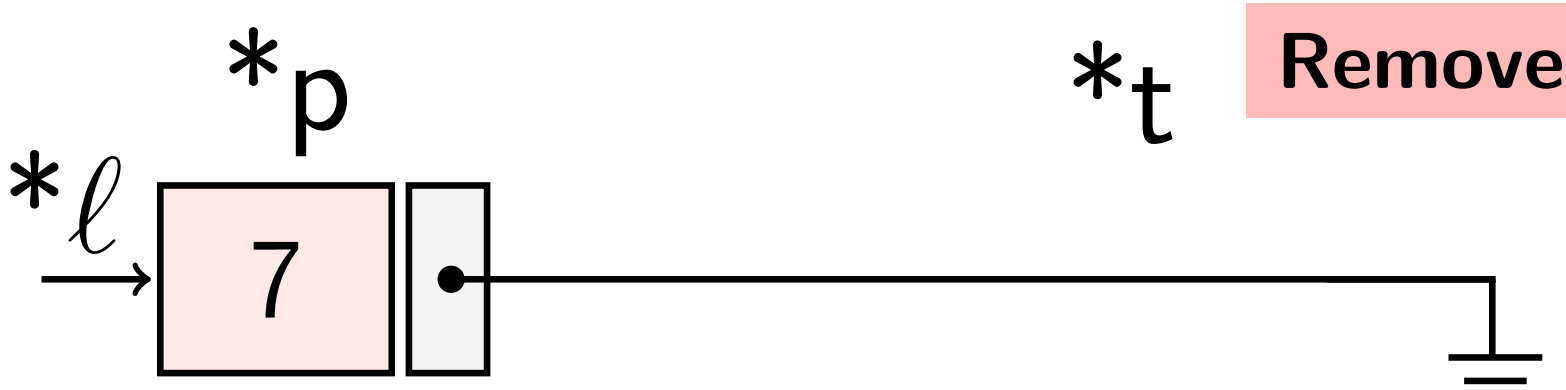
```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

Remover: 3

```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

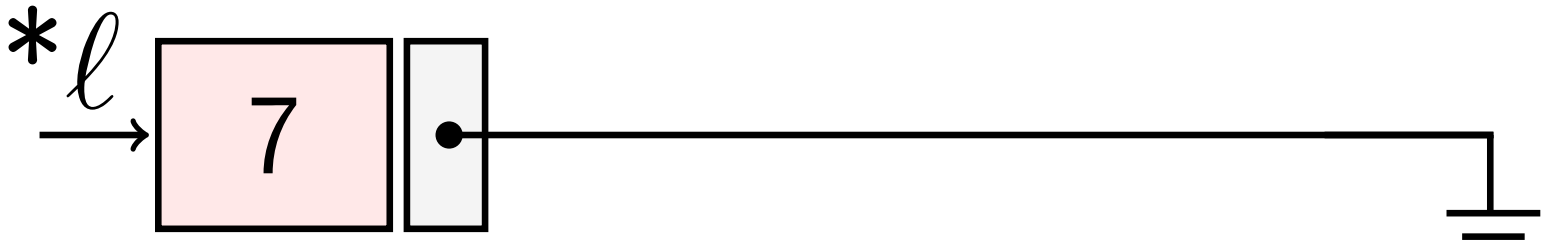


Remover: 3

```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

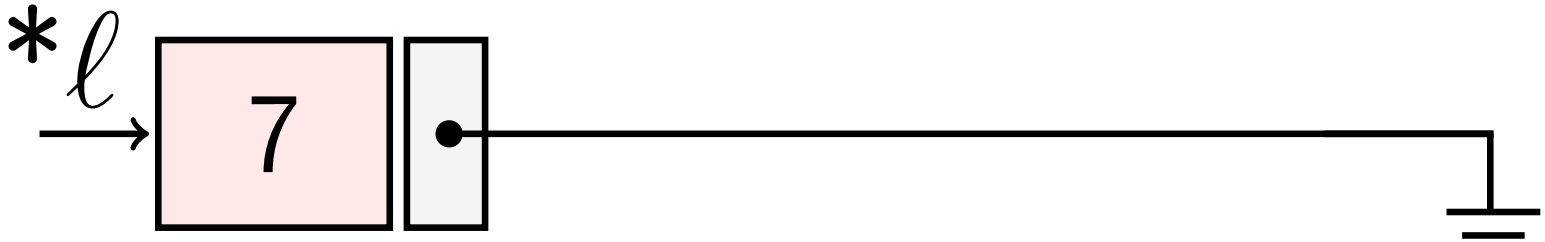
Remover: 3




```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

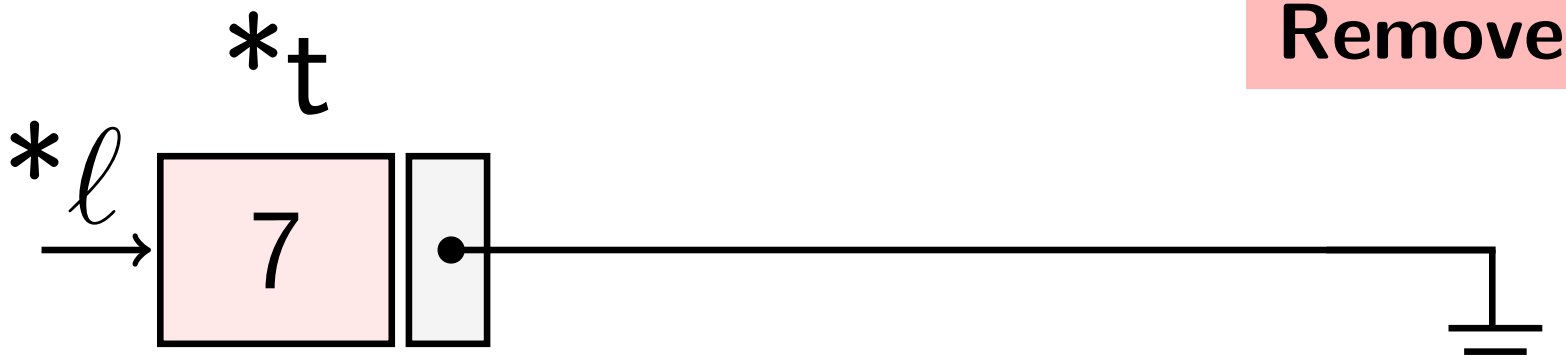
Remover: 7




```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

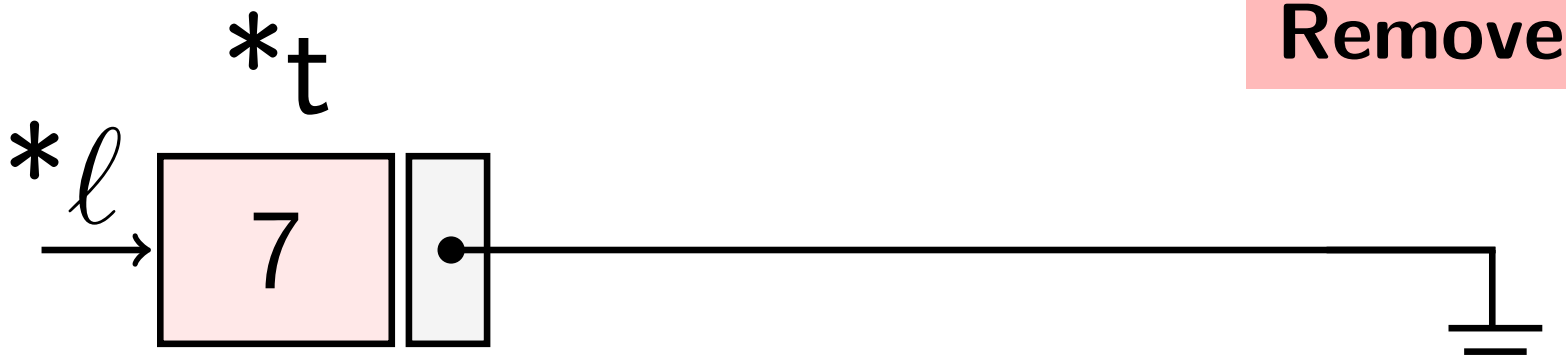
Remover: 7




```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

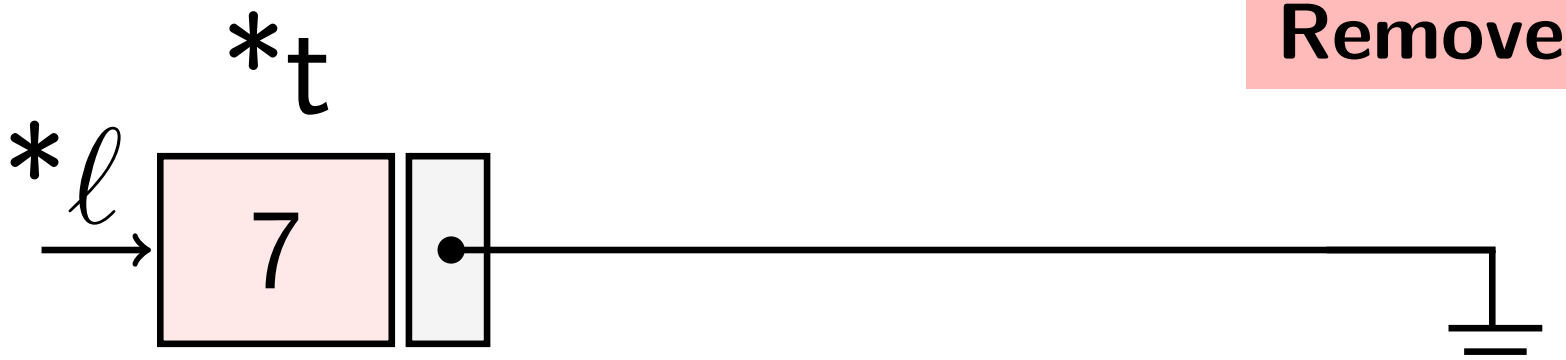
Remover: 7




```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

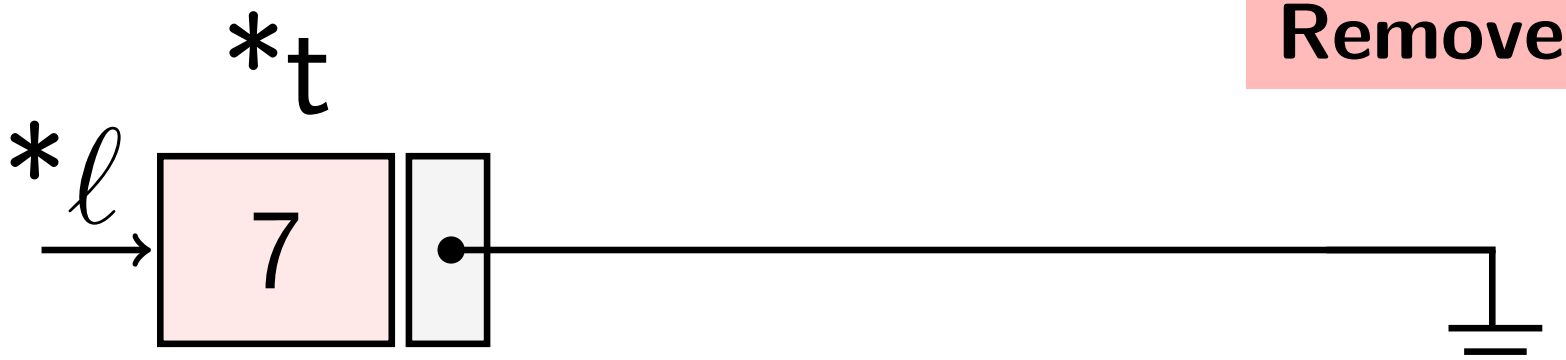
Remover: 7




```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

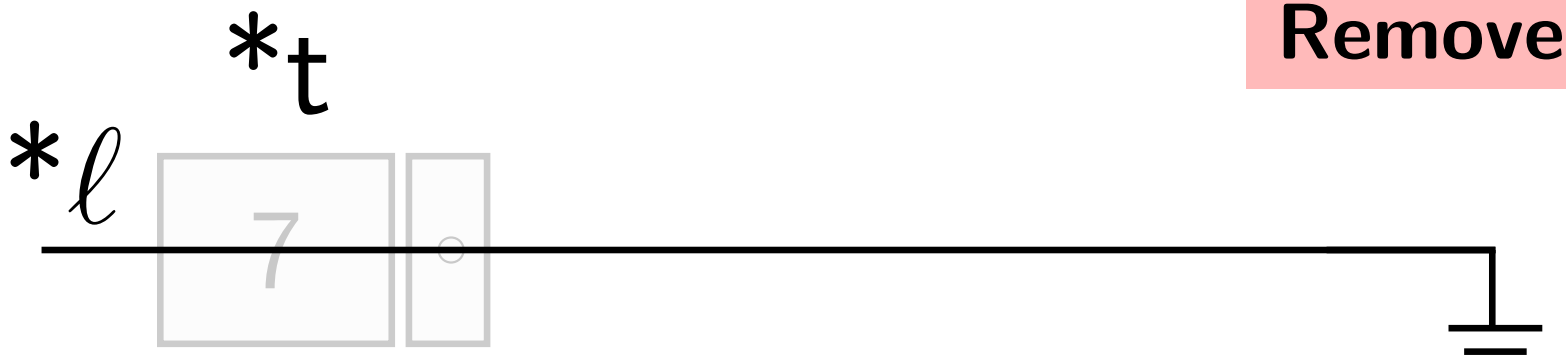
Remover: 7




```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

Remover: 7




```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

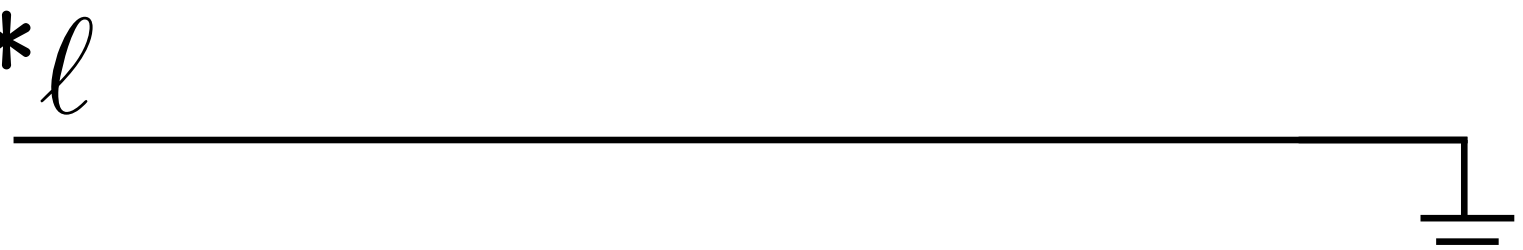
Listas encadeadas (remover)

Remover: 7


```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

Listas encadeadas (remover)

Remover: 7




```
1:  List* remove (List *ℓ, int elem)
2:      List *t = ℓ, *p = NULL;
3:      while ((t ≠ NULL)  and  (t→data ≠ elem))
4:          p = t;
5:          t = t→next;
6:      if (t = NULL) { return ℓ; }
7:      if (p = NULL) { ℓ = t→next; }
8:      else { p→next = t→next; }
9:      free (t);
10: return ℓ;
```

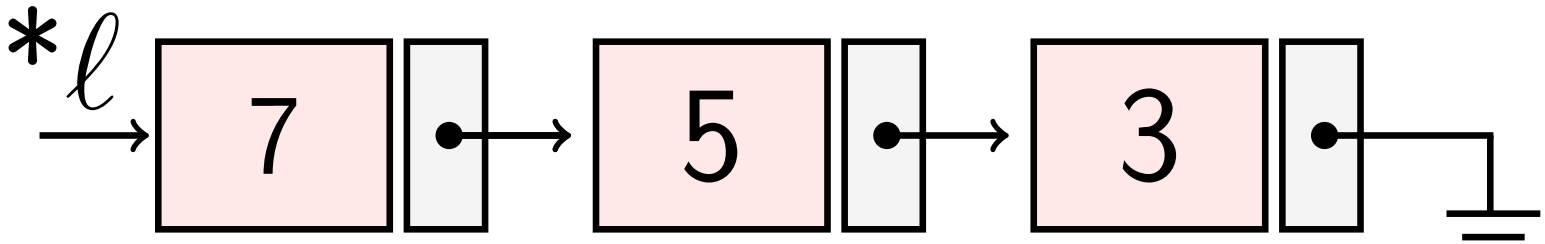
Sumário

Listas encadeadas (destruir)

A função para **destruir (ou desalocar)** uma lista precisa ter o cuidado de não liberar um nó antes de guardar o ponteiro do próximo elemento.

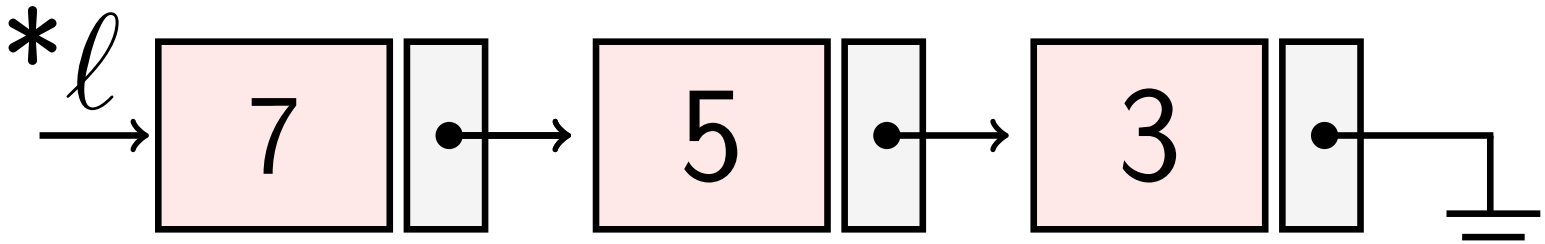
```
1: void destroy (List *ℓ)
2:     while (ℓ ≠ NULL)
3:         List *t = ℓ →next;
4:         free (ℓ);
5:         ℓ = t;
6: }
```

Listas encadeadas (destruir)



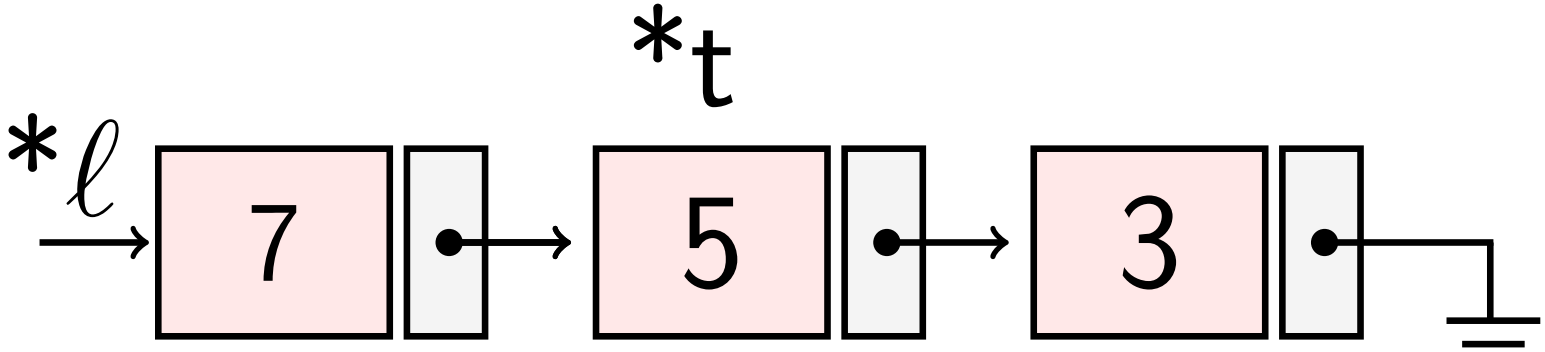

```
1: void destroy (List *ℓ)
2:     while (ℓ ≠ NULL)
3:         List *t = ℓ →next;
4:         free (ℓ);
5:         ℓ = t;
6: }
```

Listas encadeadas (destruir)



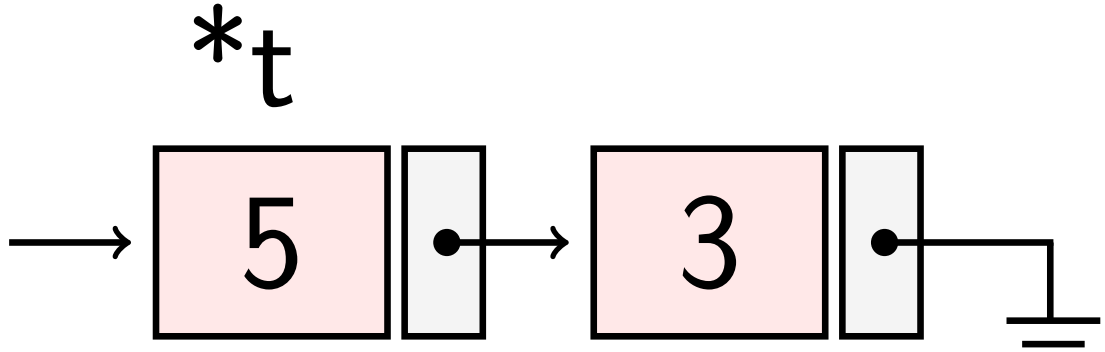

```
1: void destroy (List *ℓ)
2:     while (ℓ ≠ NULL)
3:         List *t = ℓ →next;
4:         free (ℓ);
5:         ℓ = t;
6: }
```

Listas encadeadas (destruir)



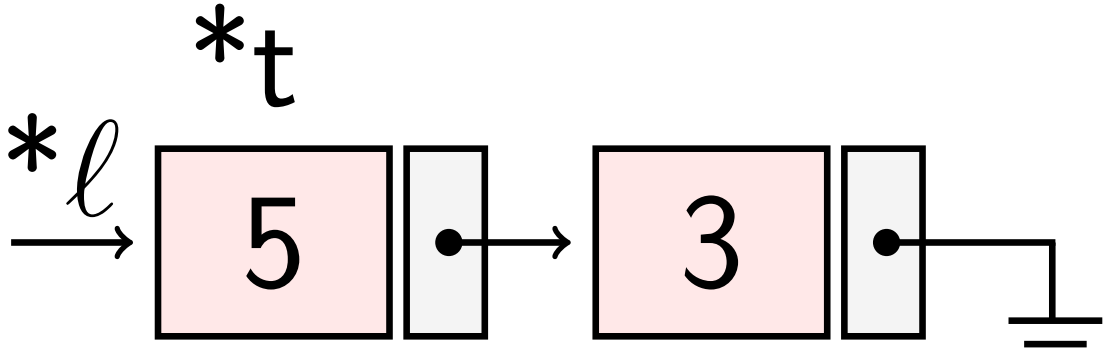

```
1: void destroy (List *ℓ)
2:     while (ℓ ≠ NULL)
3:         List *t = ℓ →next;
4:         free (ℓ);
5:         ℓ = t;
6: }
```

Listas encadeadas (destruir)



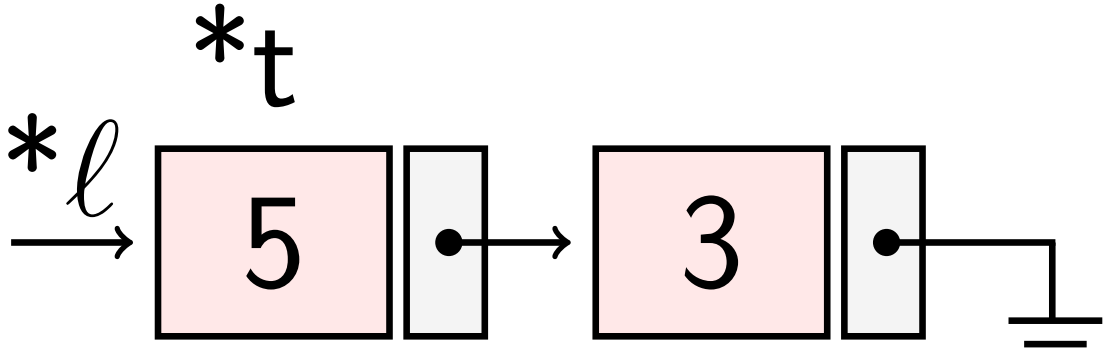

```
1: void destroy (List *ℓ)
2:     while (ℓ ≠ NULL)
3:         List *t = ℓ →next;
4:         free (ℓ);
5:         ℓ = t;
6: }
```

Listas encadeadas (destruir)



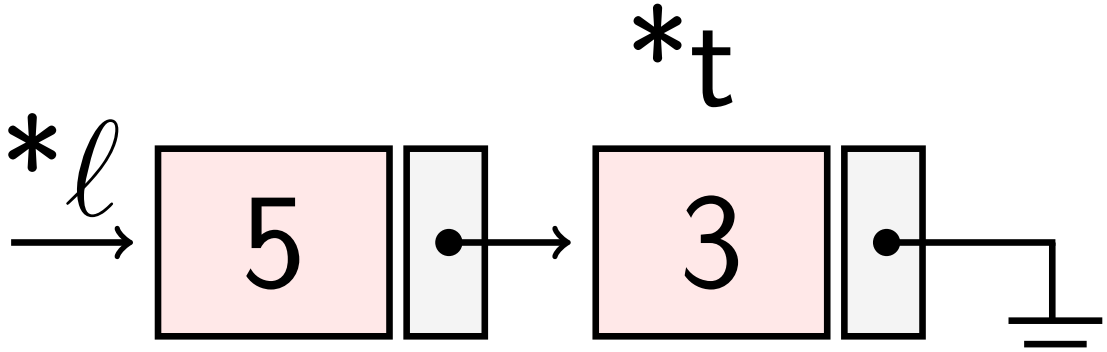

```
1: void destroy (List *ℓ)
2:     while (ℓ ≠ NULL)
3:         List *t = ℓ →next;
4:         free (ℓ);
5:         ℓ = t;
6: }
```

Listas encadeadas (destruir)



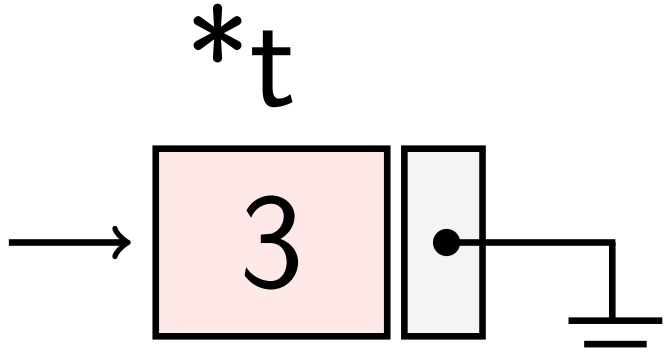

```
1: void destroy (List *ℓ)
2:     while (ℓ ≠ NULL)
3:         List *t = ℓ →next;
4:         free (ℓ);
5:         ℓ = t;
6: }
```

Listas encadeadas (destruir)



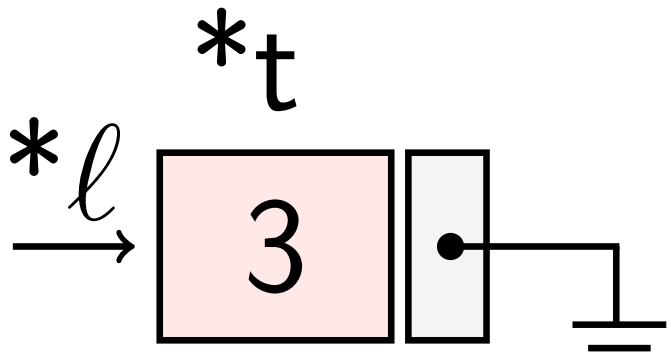

```
1: void destroy (List *ℓ)
2:     while (ℓ ≠ NULL)
3:         List *t = ℓ →next;
4:         free (ℓ);
5:         ℓ = t;
6: }
```

Listas encadeadas (destruir)



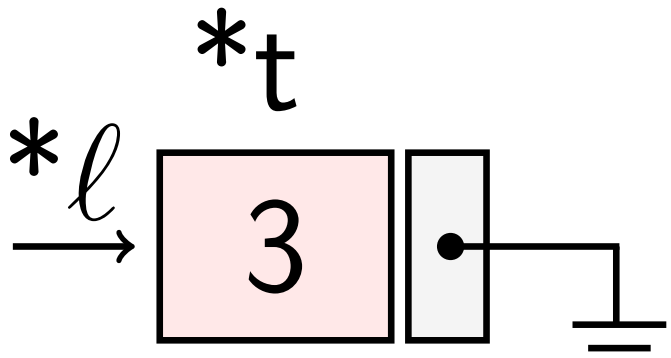

```
1: void destroy (List *ℓ)
2:     while (ℓ ≠ NULL)
3:         List *t = ℓ →next;
4:         free (ℓ);
5:         ℓ = t;
6: }
```

Listas encadeadas (destruir)



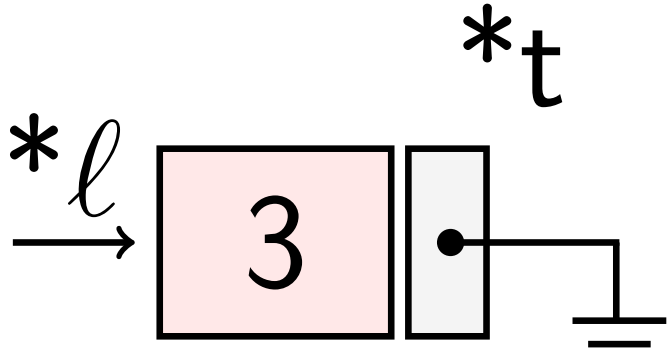

```
1: void destroy (List *ℓ)
2:     while (ℓ ≠ NULL)
3:         List *t = ℓ →next;
4:         free (ℓ);
5:         ℓ = t;
6: }
```

Listas encadeadas (destruir)



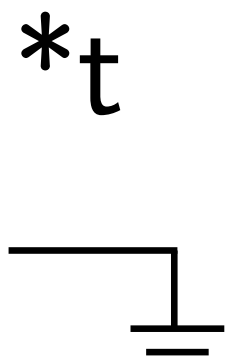

```
1: void destroy (List *ℓ)
2:     while (ℓ ≠ NULL)
3:         List *t = ℓ →next;
4:         free (ℓ);
5:         ℓ = t;
6: }
```

Listas encadeadas (destruir)



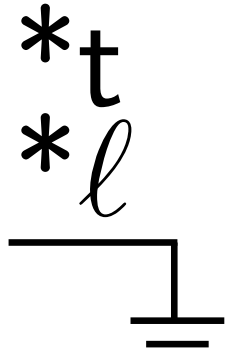

```
1: void destroy (List *ℓ)
2:     while (ℓ ≠ NULL)
3:         List *t = ℓ →next;
4:         free (ℓ);
5:         ℓ = t;
6: }
```

Listas encadeadas (destruir)



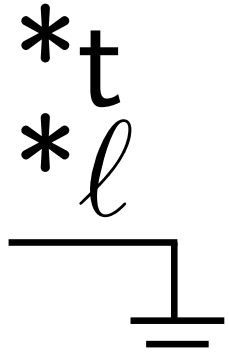

```
1: void destroy (List *ℓ)
2:     while (ℓ ≠ NULL)
3:         List *t = ℓ →next;
4:         free (ℓ);
5:         ℓ = t;
6: }
```

Listas encadeadas (destruir)




```
1: void destroy (List *ℓ)
2:     while (ℓ ≠ NULL)
3:         List *t = ℓ →next;
4:         free (ℓ);
5:         ℓ = t;
6: }
```

Listas encadeadas (destruir)




```
1: void destroy (List *ℓ)
2:     while (ℓ ≠ NULL)
3:         List *t = ℓ →next;
4:         free (ℓ);
5:         ℓ = t;
6: }
```