

Listas

Algoritmos e Estruturas de Dados 2

2017-1

Flavio Figueiredo (<http://flaviovdf.github.io>)

Listas Lineares

- Sequência de zero ou mais itens
 - x_1, x_2, \dots, x_n , na qual x_i é de um determinado tipo e n representa o tamanho da lista linear
- Sua principal propriedade estrutural envolve as posições relativas dos itens em uma dimensão:
 - Assumindo $n \geq 1$, x_1 é o primeiro item da lista e x_n é o último item da lista.
 - x_i precede x_{i+1} para $i = 1, 2, \dots, n - 1$
 - x_i sucede x_{i-1} para $i = 2, 3, \dots, n$
 - o elemento x_i é dito estar na i -ésima posição da lista

Voltando para o TP1

- Alocando uma quantidade muito grande de contas correntes
- Alocando uma quantidade grande de transações
 - Como fazer?

Solução Zero

- Malloc de Tamanho Fixo!!
- Quais os problemas?

Solução Zero

- Criamos um TAD no .h
- Lista com tamanho máximo
- E se passar de 30?
 - Pode ser um número gigante 30000
 - Mesmo assim vai extrapolar

```
#ifndef ARRAY_LIST_H
#define ARRAY_LIST_H

#define MAX_SIZE 30

typedef struct {
    int *data; //data de dados em inglês
    int nElements;
} array_list_t;

array_list_t *createList();
void addElement(int element, array_list_t *list);
void destroyList(array_list_t *list);
void printList(array_list_t *list);
#endif
```

Como Implementar?

Como Implementar?

```
array_list_t *createList() {
    int *data = (int *) malloc(MAX_SIZE * sizeof(int));
    if (data == NULL) {
        printf("Error, sem memória!!");
        exit(1);
    }
    array_list_t *list = (array_list_t *) malloc(sizeof(array_list_t));
    if (list == NULL) {
        printf("Error, sem memória!!");
        exit(1);
    }
    list->data = data;
    list->nElements = 0;
    return list;
}
```

Como Implementar?

- Imprime é trivial

```
void imprimeLista(array_list_t *list) {  
    int i;  
    for(i = 0; i < list->nElements; i++)  
        printf("%d ", list->data[i]);  
    printf("\n");  
}
```


Como Implementar?

- Destruindo a lista (lembre-se free the mallocs!)
- Chamado quando você não precisa mais da lista
- Qual o motivo dos 2 frees?

```
void destroyList(array_list_t *list) {  
    free(list->data);  
    free(list);  
}
```

Como Implementar?

- Destruindo a lista (lembre-se free the mallocs!)
- Chamado quando você não precisa mais da lista
- Qual o motivo dos 2 frees?
 - 1 para o “malloc interno”
 - 1 para o struct

```
void destroyList(array_list_t *list) {  
    free(list->data);  
    free(list);  
}
```

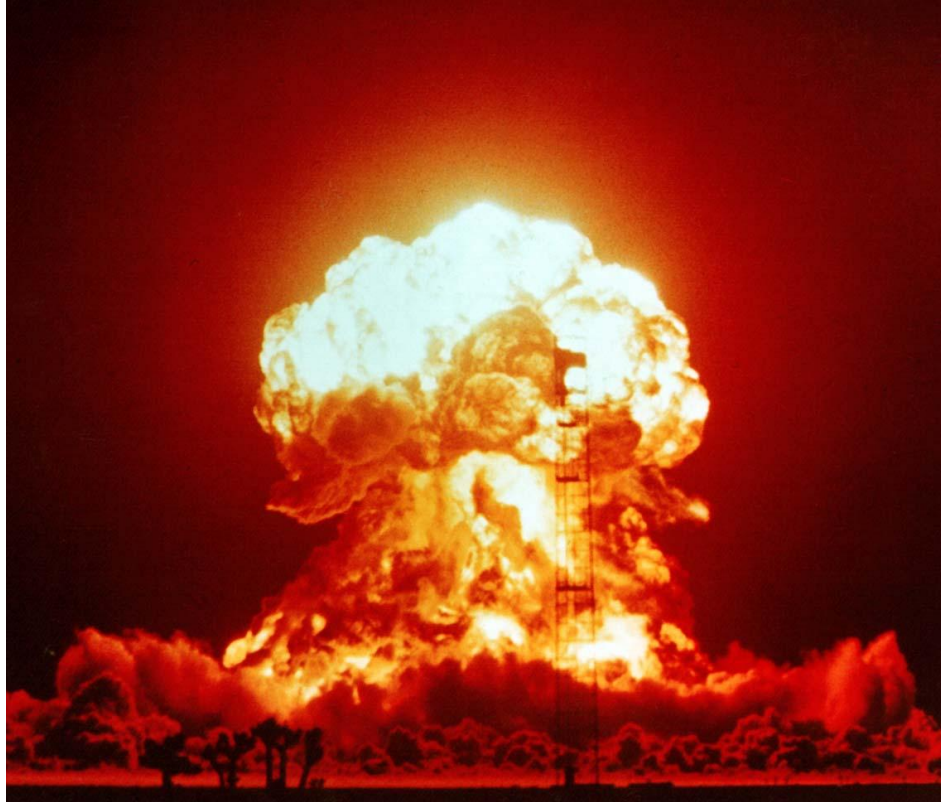
Como Implementar?

- Adicionando elementos

```
void addElement(int element, array_list_t *list) {  
    if (list->nElements == MAX_SIZE) {  
        free(list->data);  
        printf("Error, sem memória!!");  
        exit(1);  
    }  
    list->data[list->nElements] = element;  
    list->nElements++;  
}
```

Mais de 30 elementos?

Mais de 30 elementos?



Solução Melhor

- TAD similar ao anterior
- Cresce dinamicamente
- Ainda com desperdício
- Você consegue explicar o .h ao lado?

```
#ifndef ARRAY_LIST_H
#define ARRAY_LIST_H

#define INIT_SIZE 30

typedef struct {
    int nElements;
    int capacity;
    int *data;
} array_list_t;

array_list_t *createList();
void addElement(int element, array_list_t *list);
void destroyList(array_list_t *list);
void printList(array_list_t *list);
#endif
```

Como Implementar? Criando a Lista

```
array_list_t *createList() {
    int *data = (int *) malloc(INIT_SIZE * sizeof(int));
    if (data == NULL) {
        printf("Error, sem memória!!");
        exit(1);
    }
    array_list_t *list = (array_list_t *) malloc(sizeof(array_list_t));
    if (list == NULL) {
        printf("Error, sem memória!!");
        exit(1);
    }
    list->data = data;
    list->nElements = 0;
    list->capacity = INIT_SIZE;
    return list;
}
```

Alguns Métodos são Similar a Abordagem Inicial

- Imprime
- Destrói

Complicação

- Adicionar elemento

```
void addElement(int element, array_list_t *list) {
    if (list->nElements == list->capacity) {
        //Duplica tamanho da lista. Removi o IF se alocou por espaço no slide
        int *newData = (int *) malloc(list->nElements * 2 * sizeof(int));

        /*
         * Outra forma de fazer, melhor! Fiz com for para o exemplo.
         * memcpy(newData, list->data, list->nElements * sizeof(int));
         */
        for (int i = 0; i < list->nElements; i++)
            newData[i] = list->data[i];

        free(list->data);
        list->data = newData;
        list->capacity = list->nElements * 2;
    }
    list->data[list->nElements] = element;
    list->nElements++;
}
```

```

void addElement(int element, array_list_t *list) {
    if (list->nElements == list->capacity) {
        //Duplica tamanho da lista. Removi o IF se alocou por espaço no slide
        int *newData = (int *) malloc(list->nElements * 2 * sizeof(int));

        /*
         * Outra forma de fazer, melhor! Fiz com for para o exemplo.
         * memcpy(newData, list->data, list->nElements * sizeof(int));
         */
        for (int i = 0; i < list->nElements; i++)
            newData[i] = list->data[i];

        free(list->data);
        list->data = newData;
        list->capacity = list->nElements * 2;
    }
    list->data[list->nElements] = element;
    list->nElements++;
}

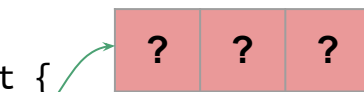
```

Free the malloc! Copiamos os elementos para uma lista nova. Libere a antiga

Passo a Passo

Malloc do tamanho inicial. Vamos supor que seja igual a 3

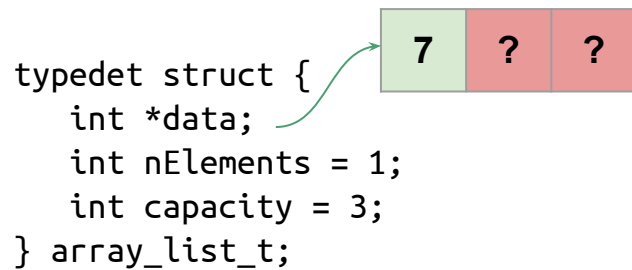
```
typedef struct {  
    int *data;  
    int nElements = 0;  
    int capacity = 3;  
} array_list_t;
```



The diagram illustrates the initial state of a linked list structure. It shows a C struct definition for `array_list_t` with three fields: `int *data`, `int nElements = 0`, and `int capacity = 3`. To the right of the struct definition, there is a horizontal array of three red boxes, each containing a question mark (`?`). A green arrow originates from the `data` field in the struct definition and points to the first box in the array, indicating that the array represents the initial memory allocation for the `data` pointer.

Passo a Passo


Inserindo 1 elemento



Passo a Passo

Outro

```
typedef struct {  
    int *data;  
    int nElements = 2;  
    int capacity = 3;  
} array_list_t;
```




The diagram illustrates the state of an array. It consists of three adjacent boxes. The first box contains the number '7', the second contains '14', and the third contains a question mark '?'. The first two boxes are light green, while the third is light red. A green arrow originates from the pointer field in the struct definition and points to the first element of the array.

Passo a Passo

+1

```
typedef struct {  
    int *data;  
    int nElements = 3;  
    int capacity = 3;  
} array_list_t;
```




The diagram illustrates the memory layout for the `array_list_t` struct. A green arrow points from the `int *data;` field in the struct definition to a horizontal array of three green boxes. The first box contains the number 7, the second contains 14, and the third contains 0, representing the elements stored in the array.

Passo a Passo

+ outro ?

```
typedef struct {  
    int *data;  
    int nElements = 3;  
    int capacity = 3;  
} array_list_t;
```



7	14	0
---	----	---

```
if (list->nElements == list->capacity) {  
    //Duplica tamanho da lista  
    int *newData = (int *) malloc(list->nElements * 2 * sizeof(int));  
    for (int i = 0; i < list->nElements; i++)  
        newData[i] = list->data[i];  
}
```


Passo a Passo

+ outro ?

```
typedef struct {  
    int *data;  
    int nElements = 3;  
    int capacity = 6;  
} array_list_t;
```

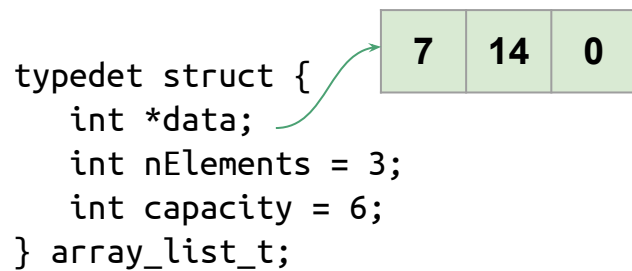
A green arrow points from the `int *data;` line in the struct definition to a green rectangular box representing an array of three integers: 7, 14, and 0.

```
int *newdata; →
```

A green arrow points from the `int *newdata;` declaration to a red rectangular box representing an array of six unknown integers, each indicated by a question mark.

Passo a Passo


+ outro ?



Passo a Passo

+ outro ?

```
typedef struct {  
    int *data;  
    int nElements = 3;  
    int capacity = 6;  
} array_list_t;
```



A diagram showing a horizontal array of three green boxes containing the numbers 7, 14, and 0. A green arrow points from the first parameter of the struct definition, `int *data;`, to the first box containing the number 7.

```
int *newdata;
```

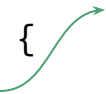


A diagram showing a horizontal array of six boxes. The first three boxes are green and contain the numbers 7, 14, and 0. The last three boxes are red and contain question marks. A green arrow points from the variable `newdata` to the first box containing the number 7.

```
free(list->data);  
list->data = newdata;  
list->capacity = list->nElements * 2;
```

Passo a Passo

+ outro ?

```
typedef struct {  
    int *data;  NULL!  
    int nElements = 3;  
    int capacity = 6;  
} array_list_t;
```



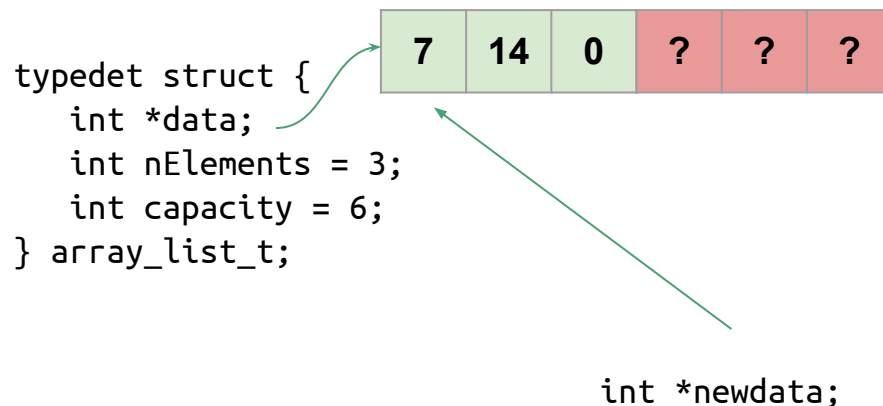
```
free(list->data);
```

```
list->data = newData;
```

```
list->capacity = list->nElements * 2;
```

Passo a Passo

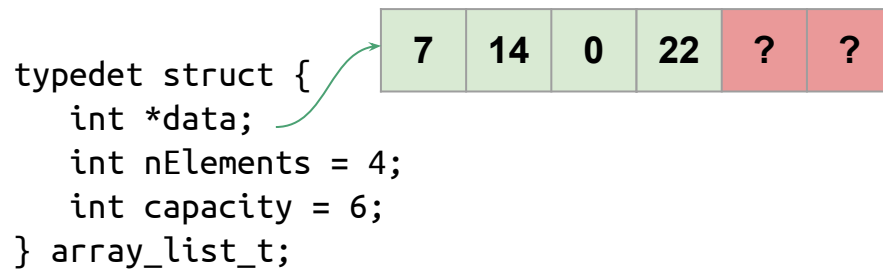
+ outro ?



New data é apenas um apontador, não precisa de free. Podemos ignorar.

Passo a Passo

+ outro ?



E para remover o último elemento?

- Quero uma nova operação no meu TAD
 - Atualizar .h
 - Implementar no .c
- Remover o último elemento

```
void removeLast(array_list_t *list);
```

E para remover o último elemento?

- Quero uma nova operação no meu TAD
- Remover o último elemento

```
//no .h
```

```
void removeLast(array_list_t *list);
```

```
//no .c
```

```
void removeLast(array_list_t *list) {  
    list->nElements--;  
}
```


E para remover o último elemento?

- Quero uma nova operação no meu TAD
- Remover o último elemento

```
//no .h  
void removeLast(array_list_t *list);
```

```
//no .c  
void removeLast(array_list_t *list) {  
    list->nElements--;  
}
```

- Pode ficar um lixo no fim da fila. Mas ok, se usarmos nElements para iterar não acessamos o mesmo.

Removendo o Primeiro Elemento

Removendo o Primeiro Elemento

- Uma var nova start no struct
- Inicia com 0
- Para remover o elemento inicial atualizamos ela
- Lixo no inicio da fila

```
//no .h
//...
typedef struct {
    int nElements;
    int capacity;
    int *data;
    int start;
} array_list_t;
//...
void removeFirst(array_list_t *list);
//...

//no .c
void removeFirst(array_list_t *list) {
    list->start++;
}
```

Complexidade!

- Qual a complexidade de:
- Adicionar um novo elemento
- Remover o último elemento
- Imprimir a Lista
- Acessar um elemento i

Complexidade!

- Qual a complexidade de:
- Adicionar um novo elemento
 - $O(n)$
- Remover o último/primeiro elemento
 - $O(1)$
- Imprimir a Lista
 - $O(n)$
- Acessar um elemento i
 - $O(1)$

Problemas

- Lixo de memória no início e no fim ao longo do tempo
- Como tratar?
- Limpar o lixo com realocações

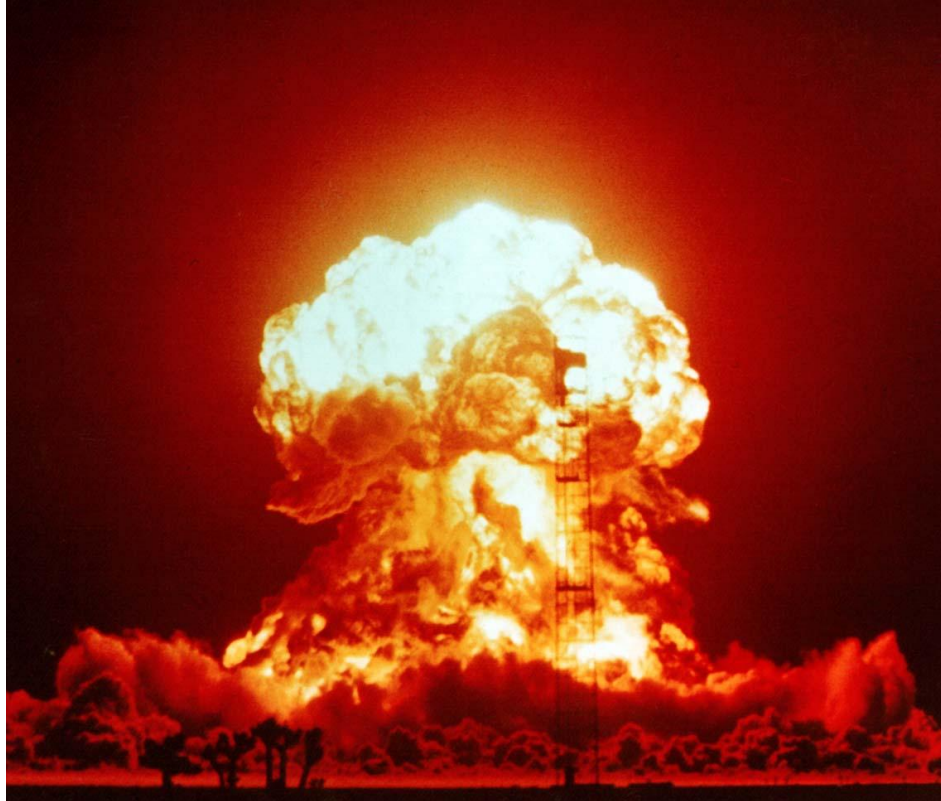
Problemas

- Lixo de memória no início e no fim ao longo do tempo
- Como tratar?
- Limpar o lixo com realocações
 - $O(n)$

+ Operações: Como fazer?

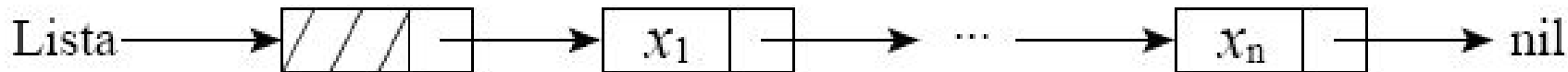
- Remover um elemento no meio da lista
 - `removeElement(arraylist_t array_list, int i);`
- Adicionar um elemento no meio da lista
 - `addElement(arraylist_t array_list, int elem, int i);`
- Alterar um elemento no meio da lista
 - `resetElement(arraylist_t array_list, int i, int value);`

+ Operações



Lista com Ponteiros

- Algumas operações são complicadas com arrays/arranjos
- Podemos representar uma lista de outras formas



Cabeçalho

```
#ifndef POINTER_LIST_H
#define POINTER_LIST_H

typedef struct node {
    int value;
    struct node *next;
} node_t;

typedef struct {
    node_t *first;
    node_t *last;
} pointer_list_t;

pointer_list_t *createList();
void addElement(int element, pointer_list_t *list);
void destroyList(pointer_list_t *list);
void printList(pointer_list_t *list);
void removeElement(pointer_list_t *list, int i);
#endif
```

Cabeçalho

- Segredo está nestes dois structs
- Um pouco chato por causa de C
- Vamos entender

```
typedef struct node {  
    int info;  
    struct node *next;  
} node_t;
```

```
typedef struct {  
    node_t *first;  
    node_t *last;  
} pointer_list_t;
```

Iniciamos Assim

```
typedef struct node {  
    int value;  
    struct node *next;  
} node_t;
```

The diagram illustrates the initial state of a linked list node. A red box containing a question mark represents the memory location for the 'value' field. The text 'NULL!' represents the initial value of the 'next' pointer field.

Ficamos Assim

```
typedef struct node {  
    int info;  
    struct node *next;  
} node_t;
```

7

```
typedef struct node {  
    int info;  
    struct node *next;  
} node_t;
```

23

```
typedef struct node {  
    int info;  
    struct node *next;  
} node_t;
```

Ficamos Assim

```
typedef struct node {  
    int info;  
    struct node *next;  
} node_t;
```

7

```
typedef struct node {  
    int info;  
    struct node *next;  
} node_t;
```

23

- Um struct aponta para outro
- Cada Struct mantém um valor

```
typedef struct node {  
    int info;  
    struct node *next;  
} node_t;
```

```
typedef struct {  
    node_t *first;  
    node_t *last;  
} pointer_list_t;
```

```
typedef struct node {  
    int info;  
    struct node *next;  
} node_t;
```

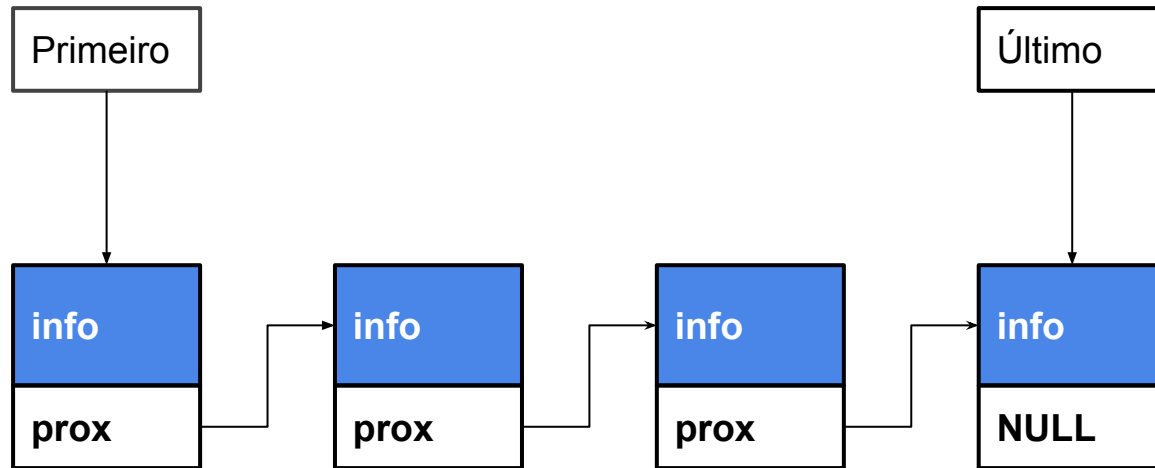
7

```
typedef struct node {  
    int info;  
    struct node *next;  
} node_t;
```

23

```
typedef struct node {  
    int info;  
    struct node *next;  
} node_t;
```


Mais Abstrato

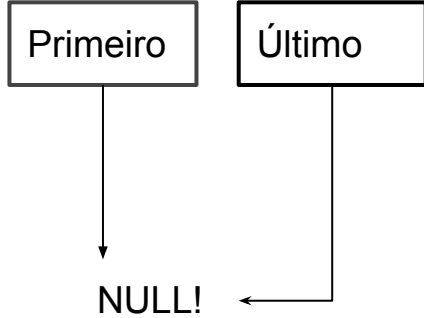


Vamos Implementar?

Criando uma Lista

```
pointer_list_t *createList() {
    pointer_list_t *list = (pointer_list_t *) malloc(sizeof(pointer_list_t));
    if (list == NULL) {
        printf("Error, sem memória!!");
        exit(1);
    }
    list->first = NULL;
    list->last = NULL;
    return list;
}
```

Criando uma Lista



```
typedef struct {  
    node_t *first; → NULL!  
    node_t *last; → NULL!  
} pointer_list_t;
```

Adicionando Elementos

```
void addElement(int element, pointer_list_t *list) {
    node_t *new = (node_t *) malloc(sizeof(node_t));
    if (new == NULL) {
        printf("Error, sem memória!!");
        exit(1);
    }
    new->info = element;
    new->next = NULL;

    if (list->last != NULL) //Cria ponteiro para novo elemento
        list->last->next = new;

    //Atualize first e last
    list->last = new;
    if (list->first == NULL)
        list->first = new;
}
```

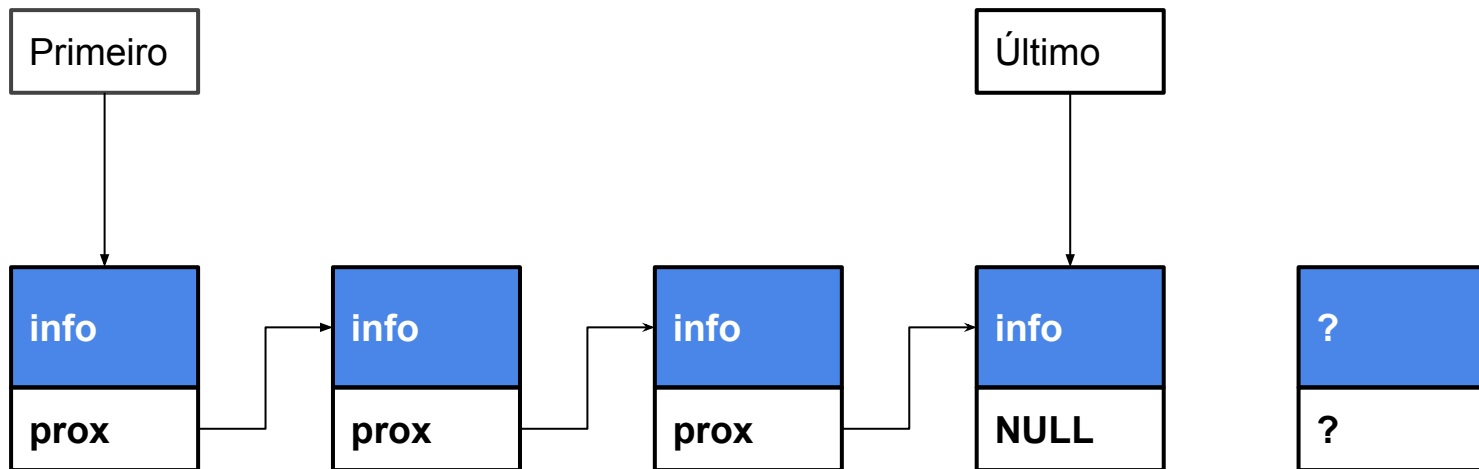
Simplificando (assumir sem erros)

```
void addElement(int element, pointer_list_t *list) {
    node_t *new = (node_t *) malloc(sizeof(node_t));
    if (list->last != NULL) //Cria ponteiro para novo elemento
        list->last->next = new;

    //Atualize first e last
    list->last = new;
    if (list->first == NULL)
        list->first = new;
    new->info = element;
    new->next = NULL;
}
```

Aloco um novo elemento

```
node_t *new = (node_t *) malloc(sizeof(node_t));
```



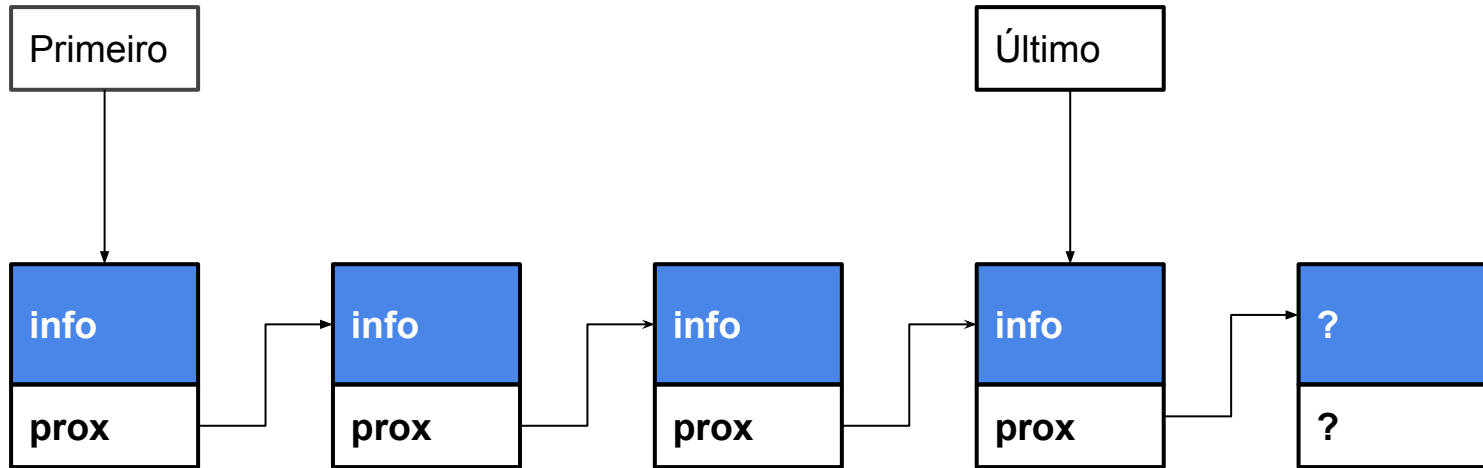
```
if (list->last != NULL) //Cria ponteiro para novo elemento
    list->last->next = new;
```

```
//Atualize first e last
```

```
list->last = new;
```

```
if (list->first == NULL)
```

```
    list->first = new;
```



Atualizo Ponteiros

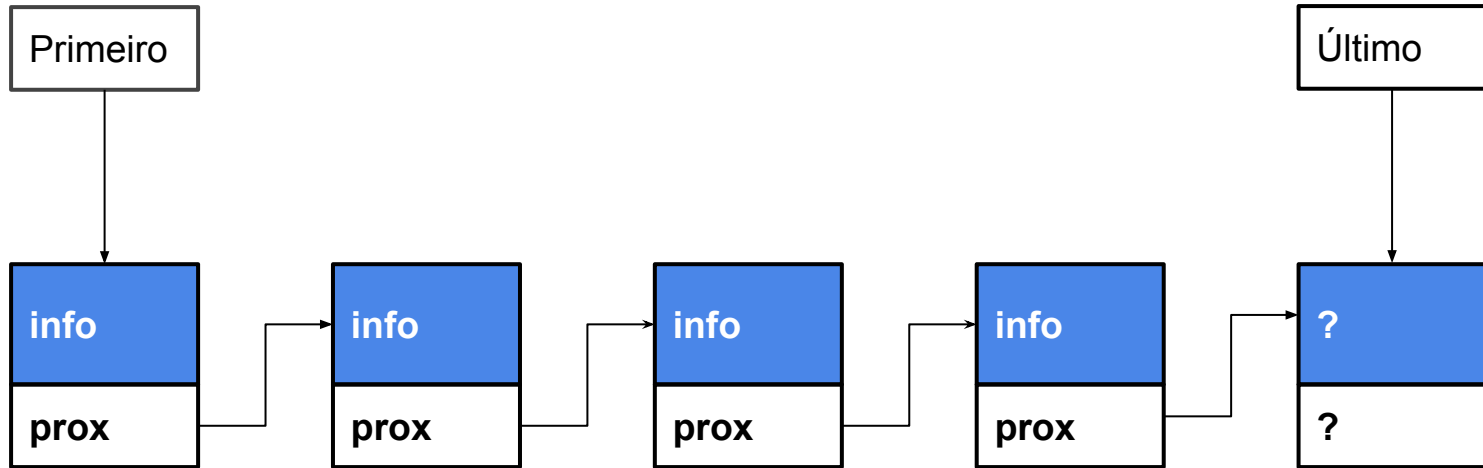

```
if (list->last != NULL) //Cria ponteiro para novo elemento
    list->last->next = new;
```

```
//Atualize first e last
```

```
list->last = new;
```

```
if (list->first == NULL)
```

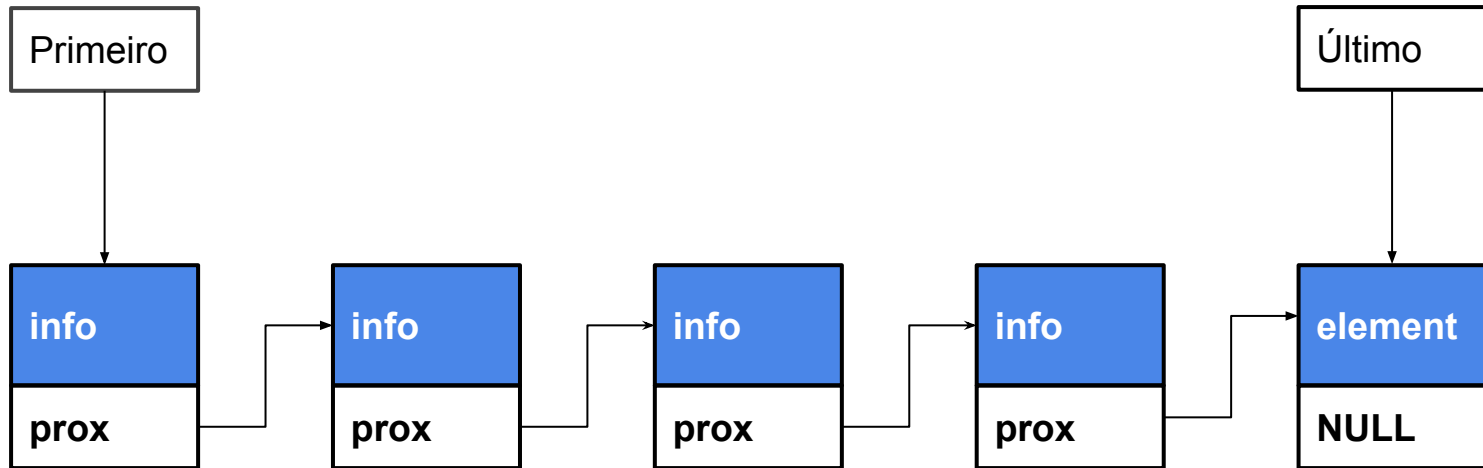
```
    list->first = new;
```



Atualizo Ponteiros

Atualiza Valores

```
new->info = element;  
new->next = NULL;
```



Limpar a Lista

```
void destroyList(pointer_list_t *list) {  
    node_t *toVisit = list->first;  
    node_t *toFree;  
    while (toVisit != NULL) {  
        toFree = toVisit;  
        toVisit = toVisit->next;  
        free(toFree);  
    }  
    free(list);  
}
```

```
typedef struct {  
    node_t *first;  
    node_t *last;  
} pointer_list_t;
```

```
typedef struct node {  
    int info;  
    struct node *next;  
} node_t;
```

7

```
typedef struct node {  
    int info;  
    struct node *next;  
} node_t;
```

23

```
typedef struct node {  
    int info;  
    struct node *next;  
} node_t;
```

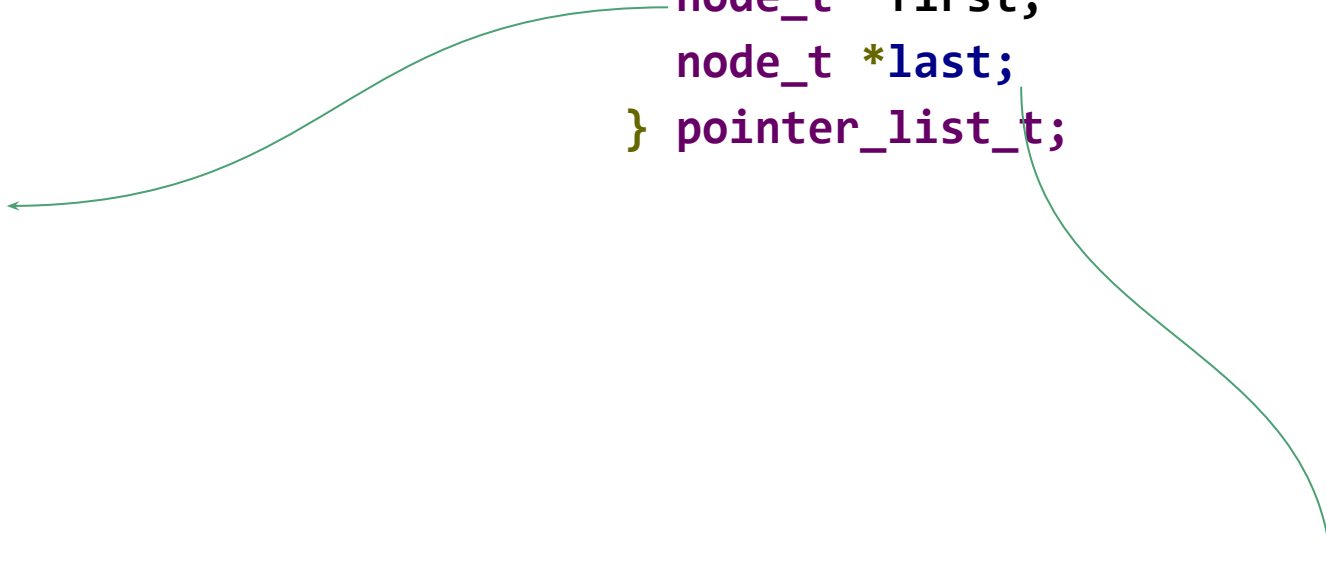
```
typedef struct {  
    node_t *first;  
    node_t *last;  
} pointer_list_t;
```

```
typedef struct node {  
    int info;  
    struct node *next;  
} node_t;
```

23

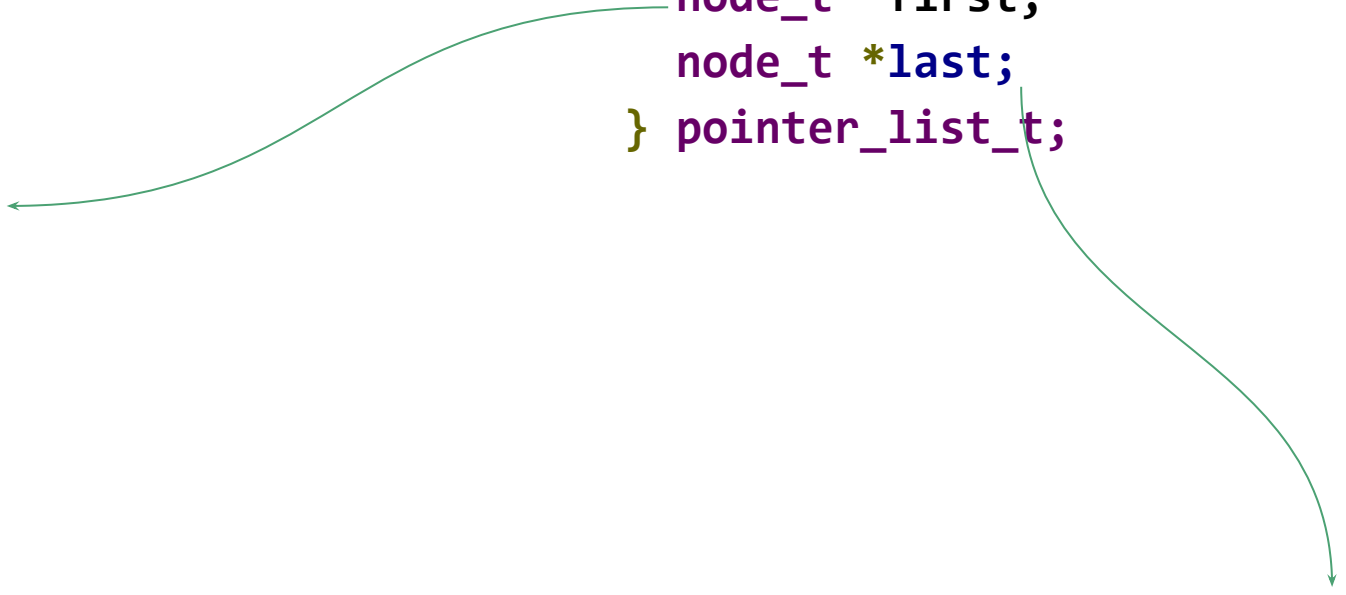
```
typedef struct node {  
    int info;  
    struct node *next;  
} node_t;
```

```
typedef struct {  
    node_t *first;  
    node_t *last;  
} pointer_list_t;
```

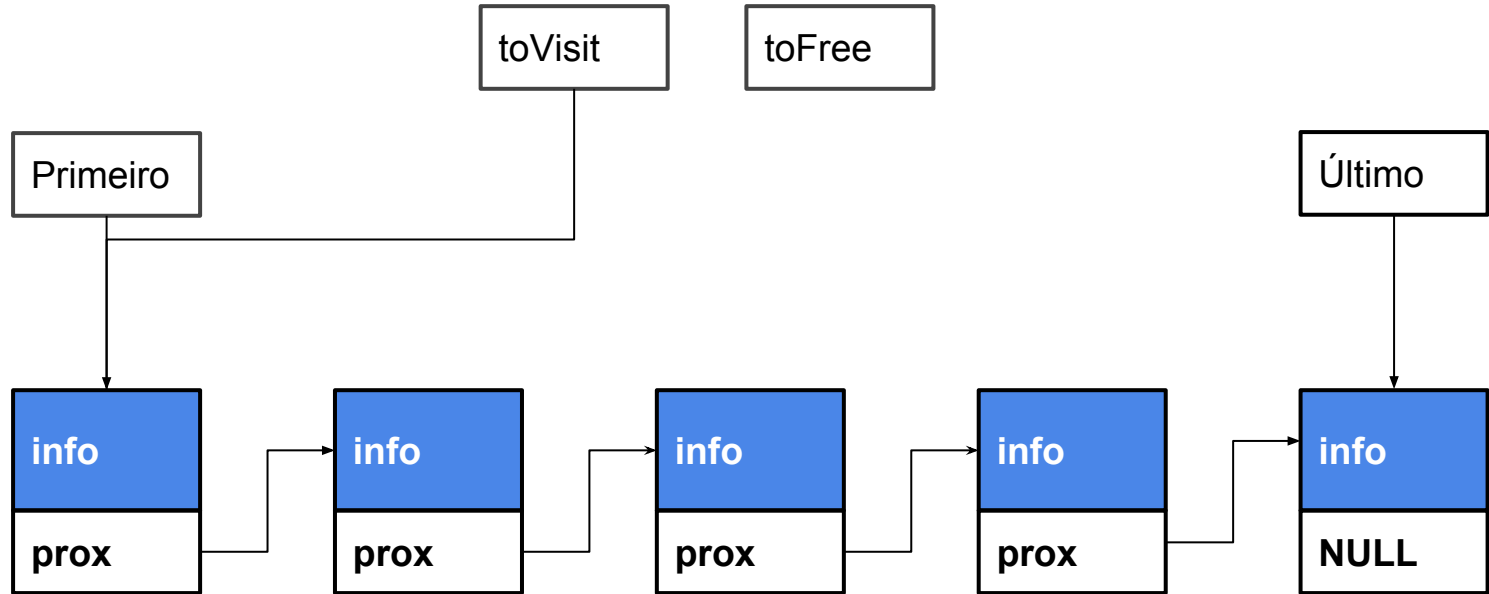


```
typedef struct node {  
    int info;  
    struct node *next;  
} node_t;
```

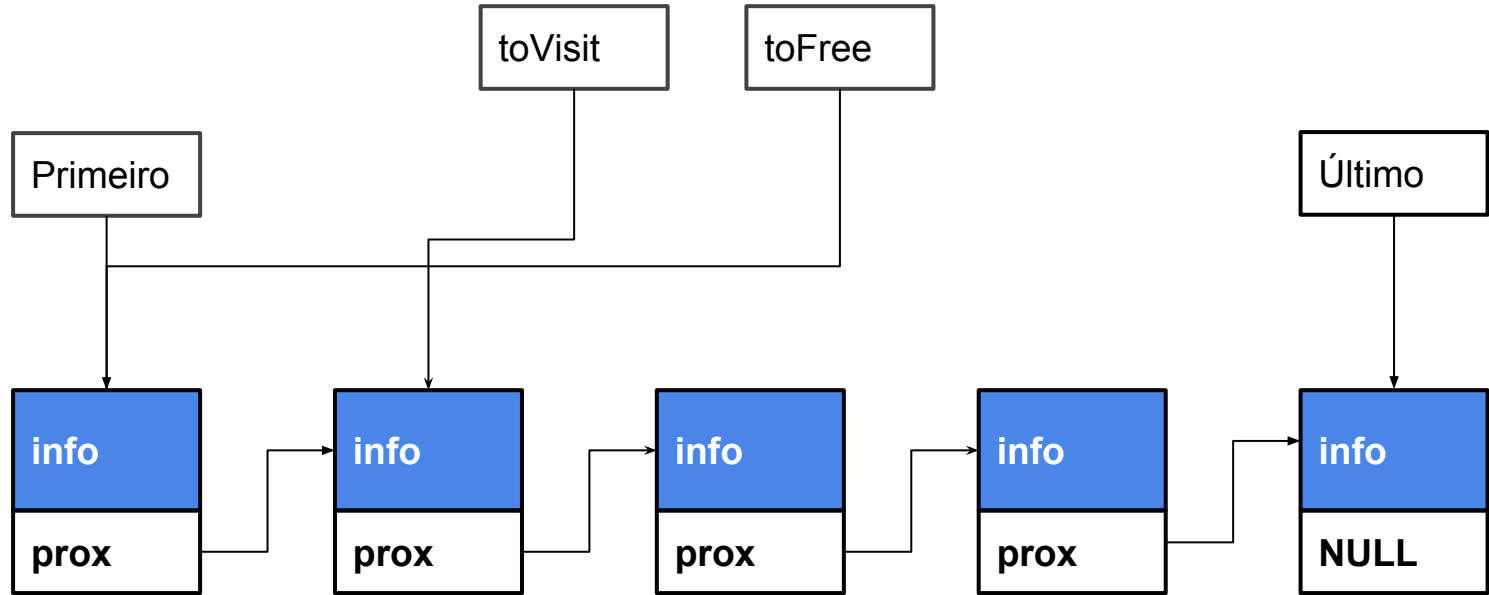
```
typedef struct {  
    node_t *first;  
    node_t *last;  
} pointer_list_t;
```



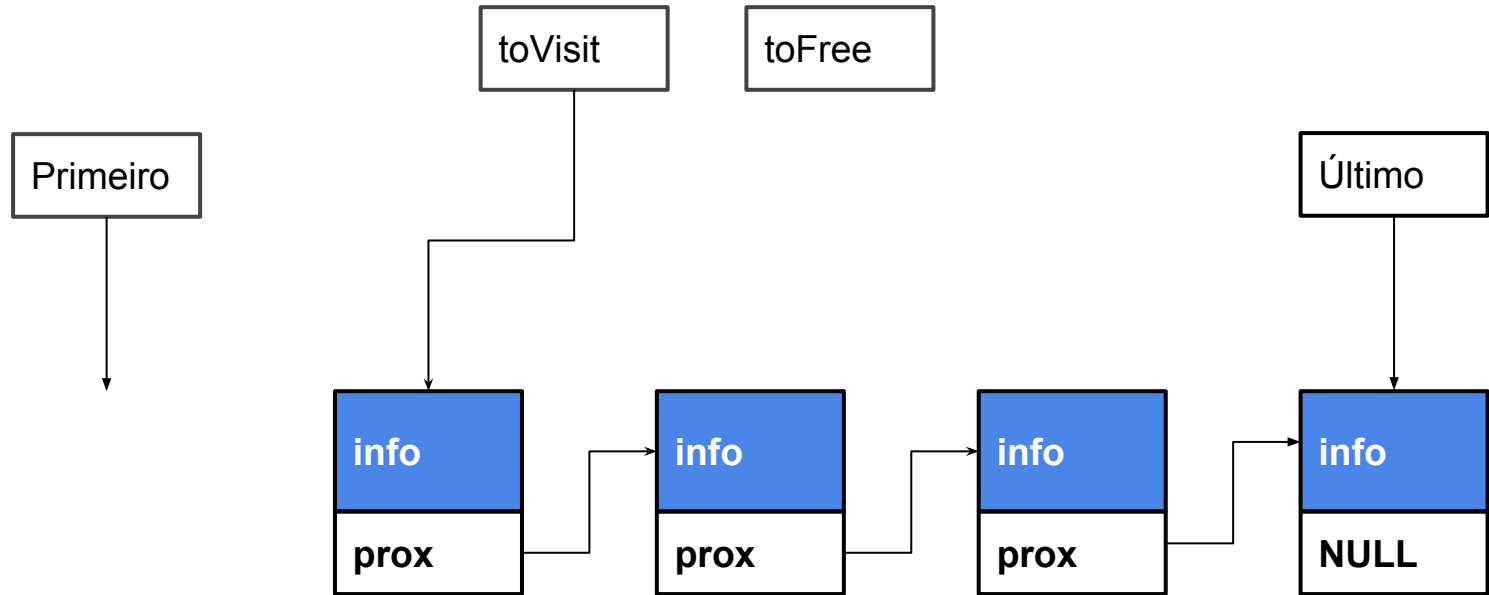
Limpendo a Lista



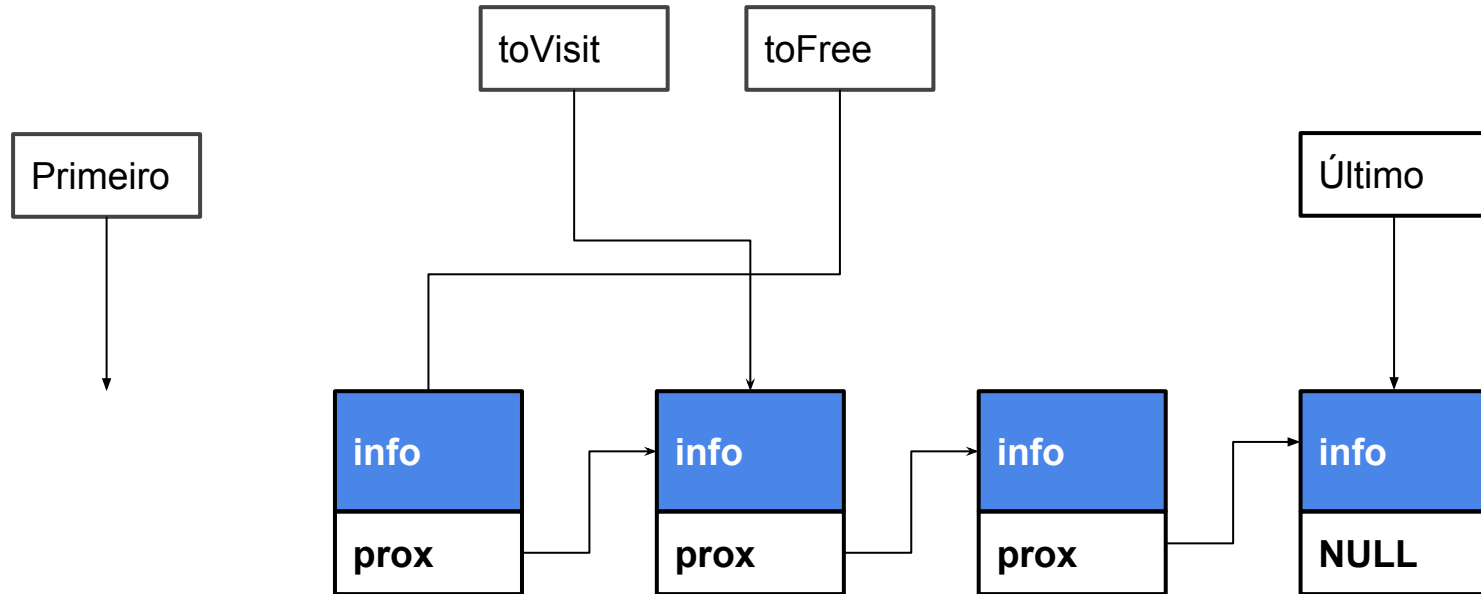
Limpendo a Lista



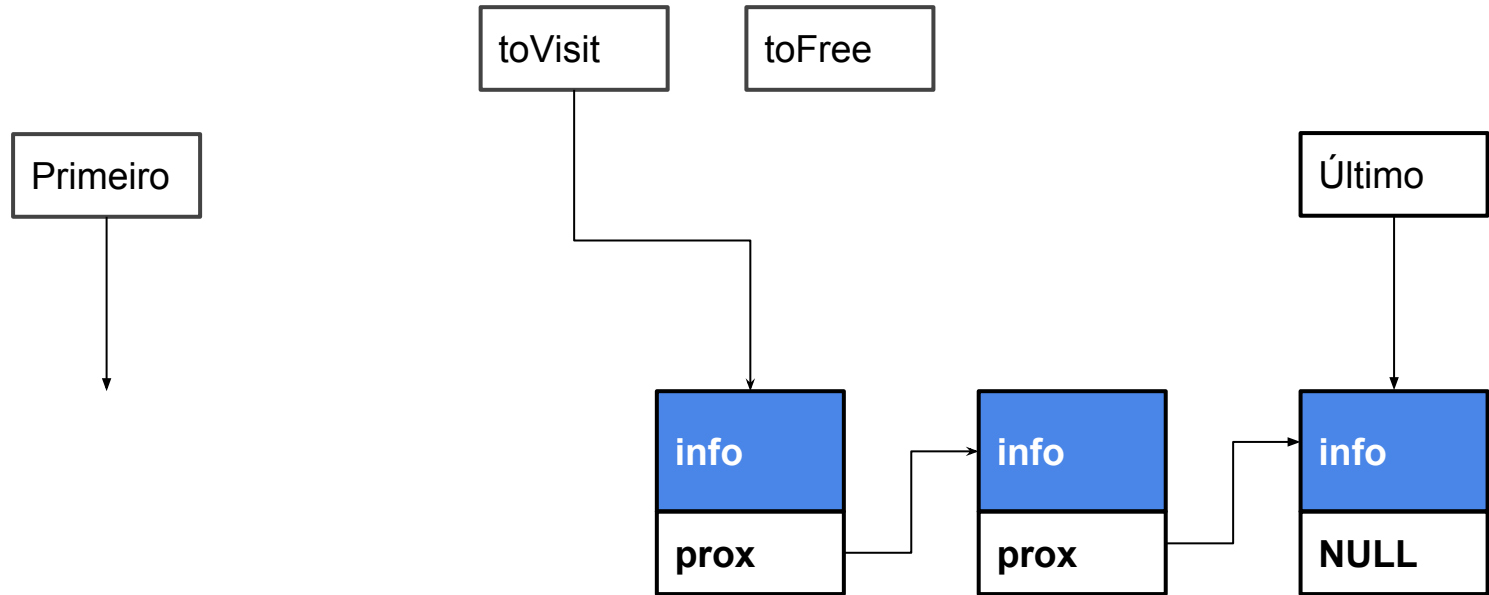
Limpendo a Lista



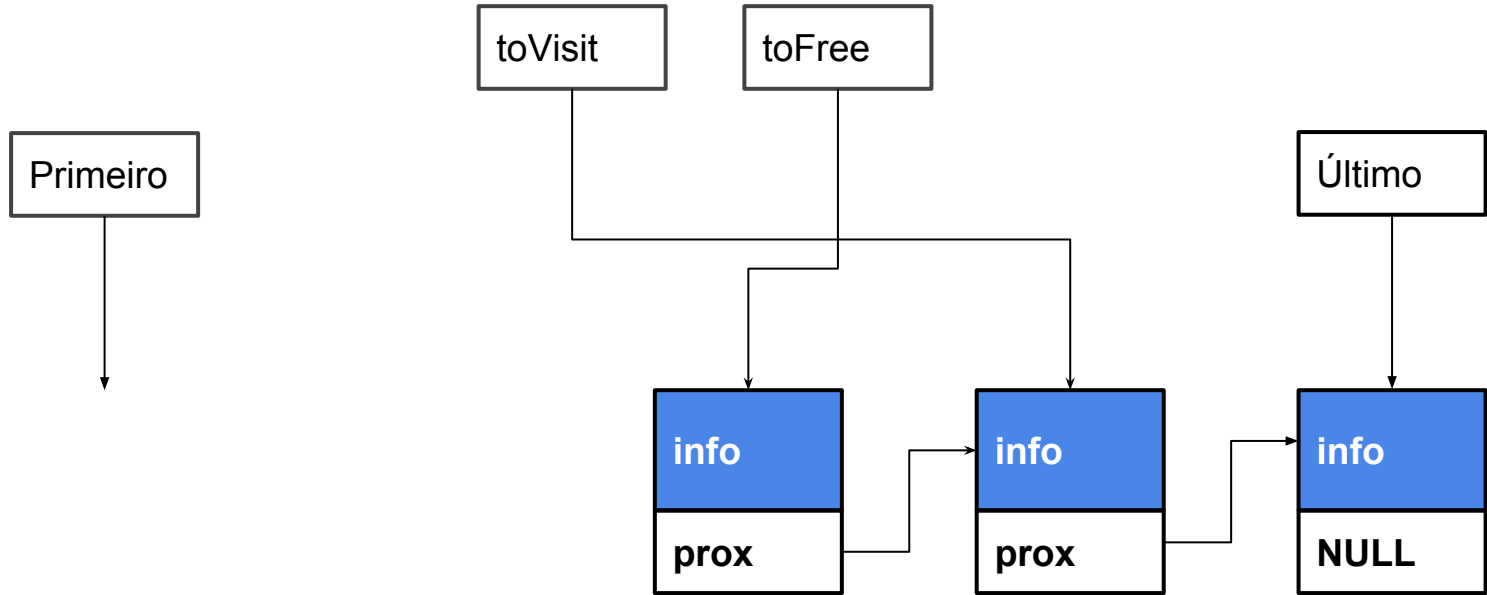
Limpendo a Lista



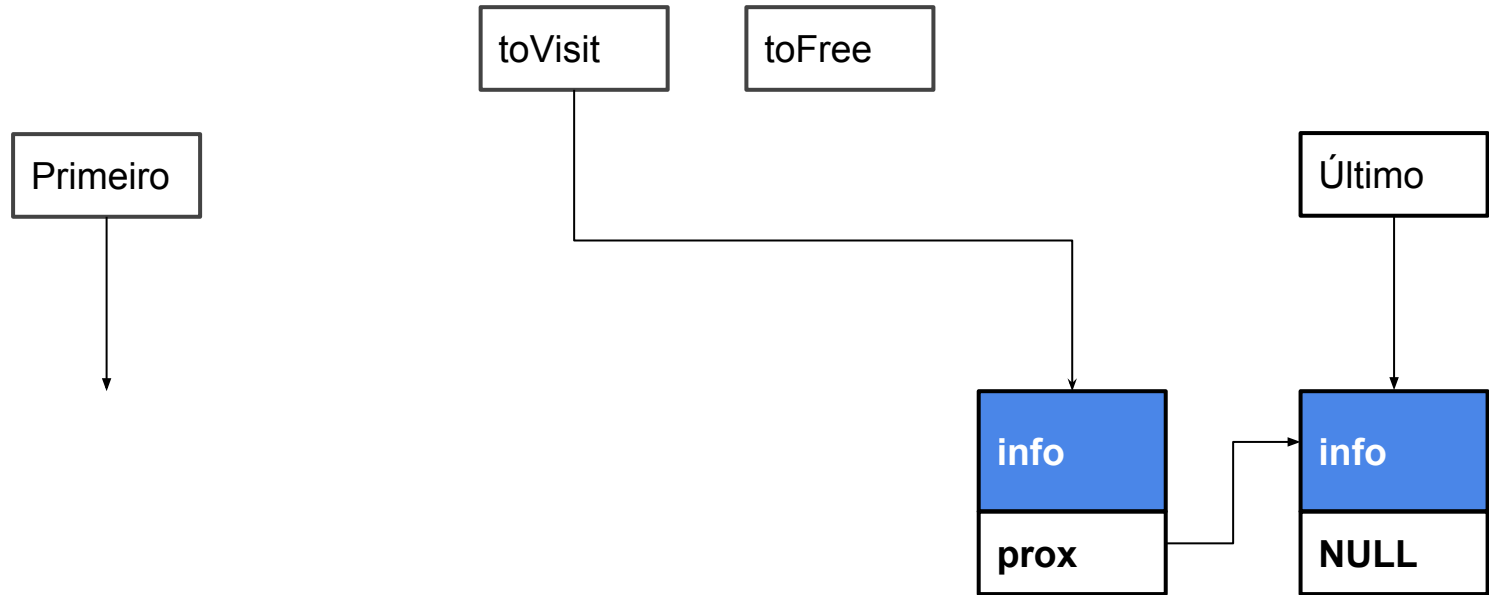
Limpendo a Lista



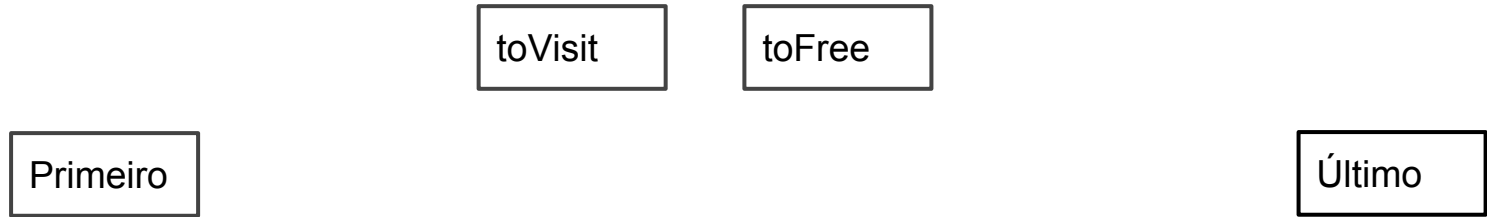
Limpendo a Lista



Limpendo a Lista



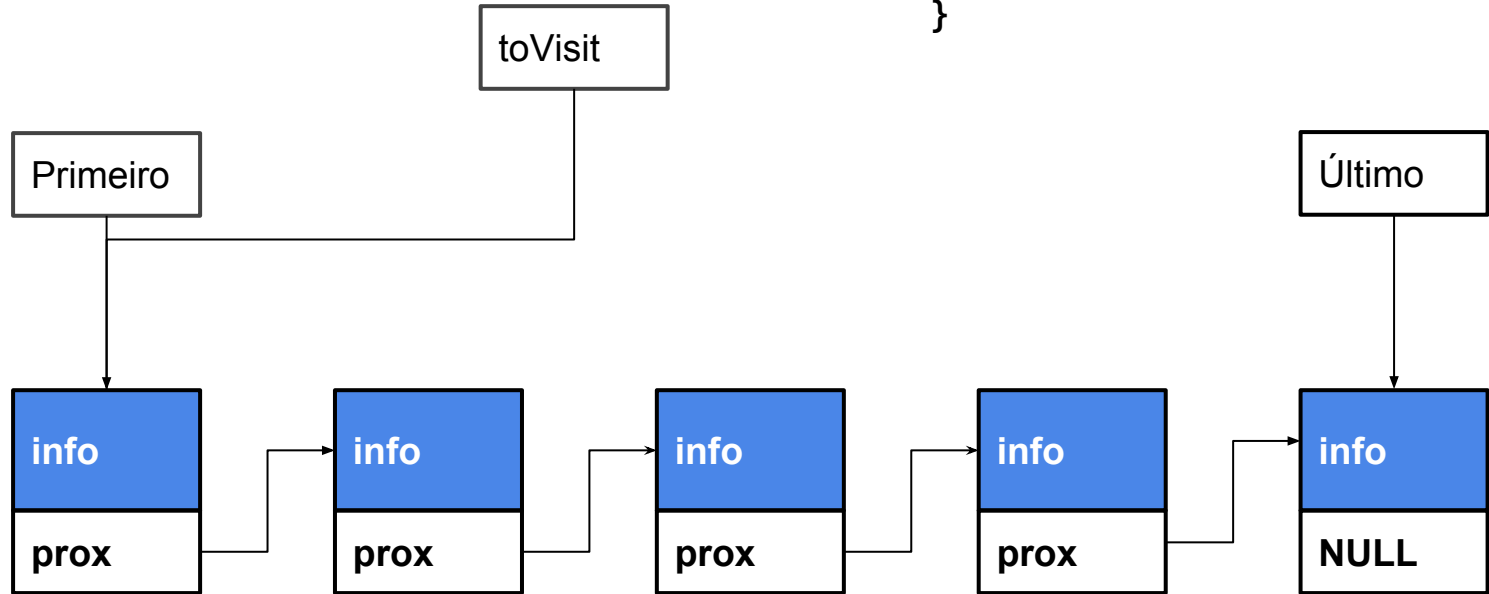
Mais um Pouco Disto



Imprimindo a Lista

```
void printList(pointer_list_t *list) {  
    node_t *toVisit = list->first;  
    while (toVisit != NULL) {  
        printf("%d ", toVisit->info);  
        toVisit = toVisit->next;  
    }  
    printf("\n");  
}
```

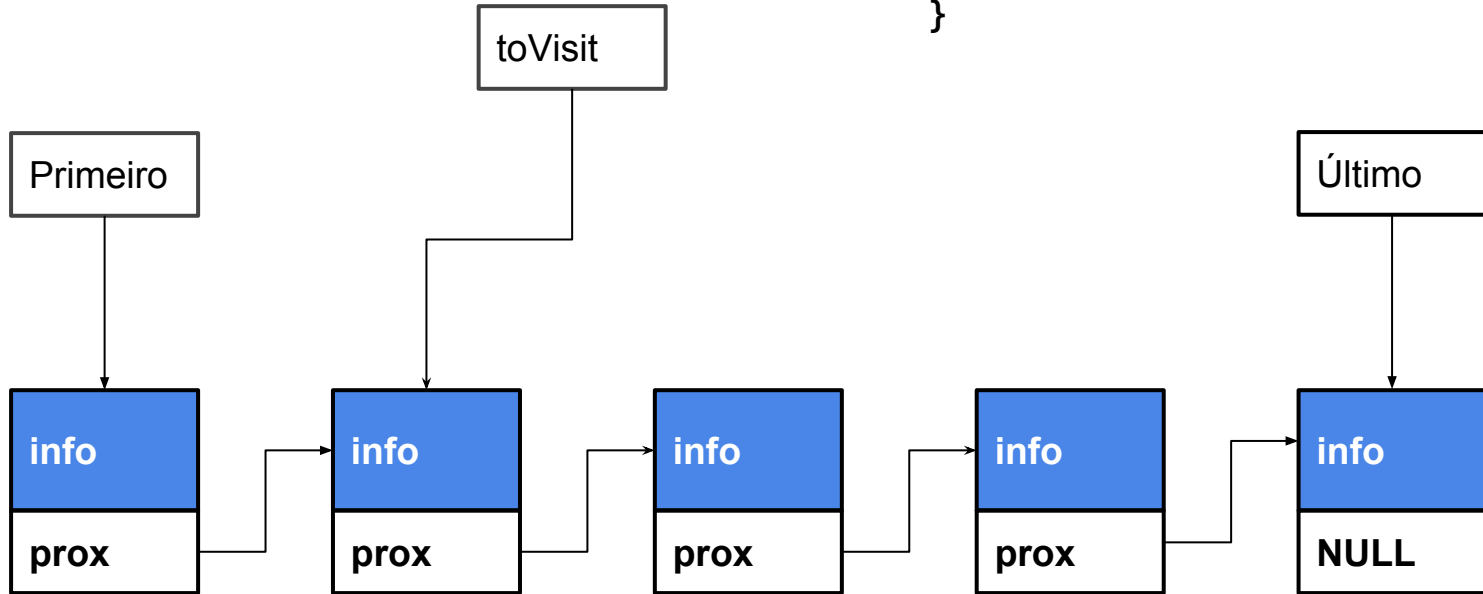
Imprimindo a Lista



```
while (toVisit != NULL) {  
    printf("%d ", toVisit->info);  
    toVisit = toVisit->next;  
}
```

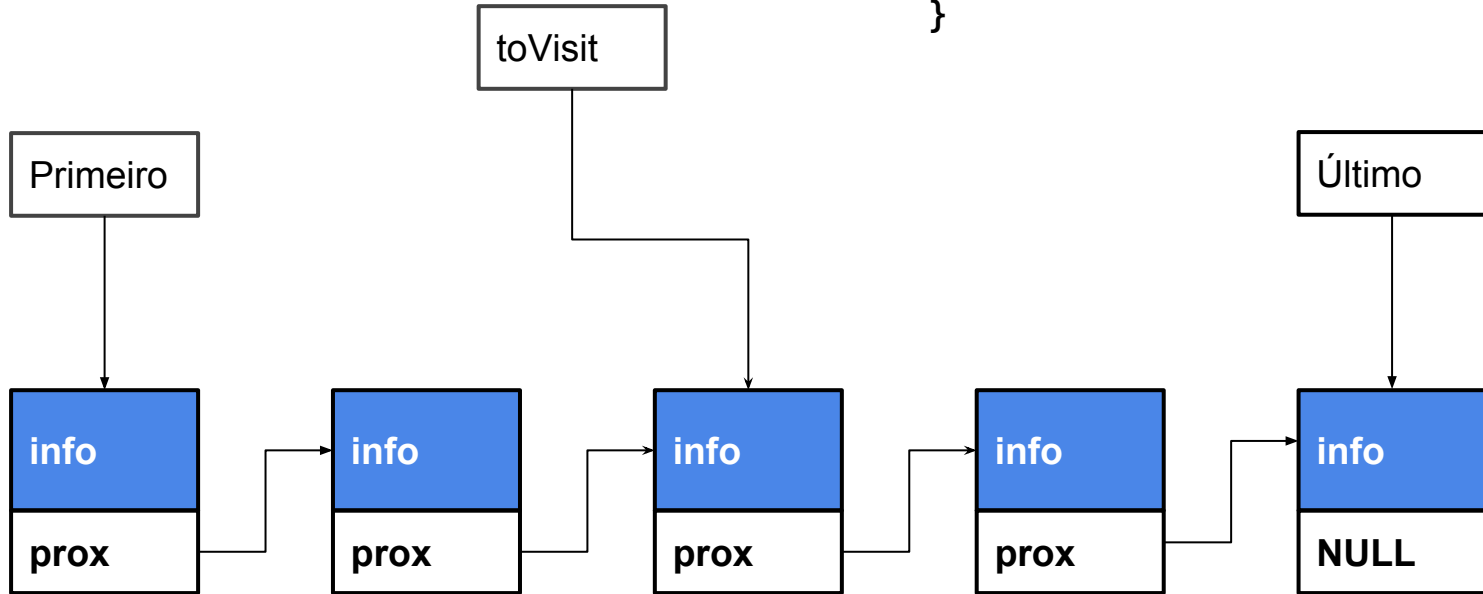
Imprimindo a Lista

```
while (toVisit != NULL) {  
    printf("%d ", toVisit->info);  
    toVisit = toVisit->next;  
}
```



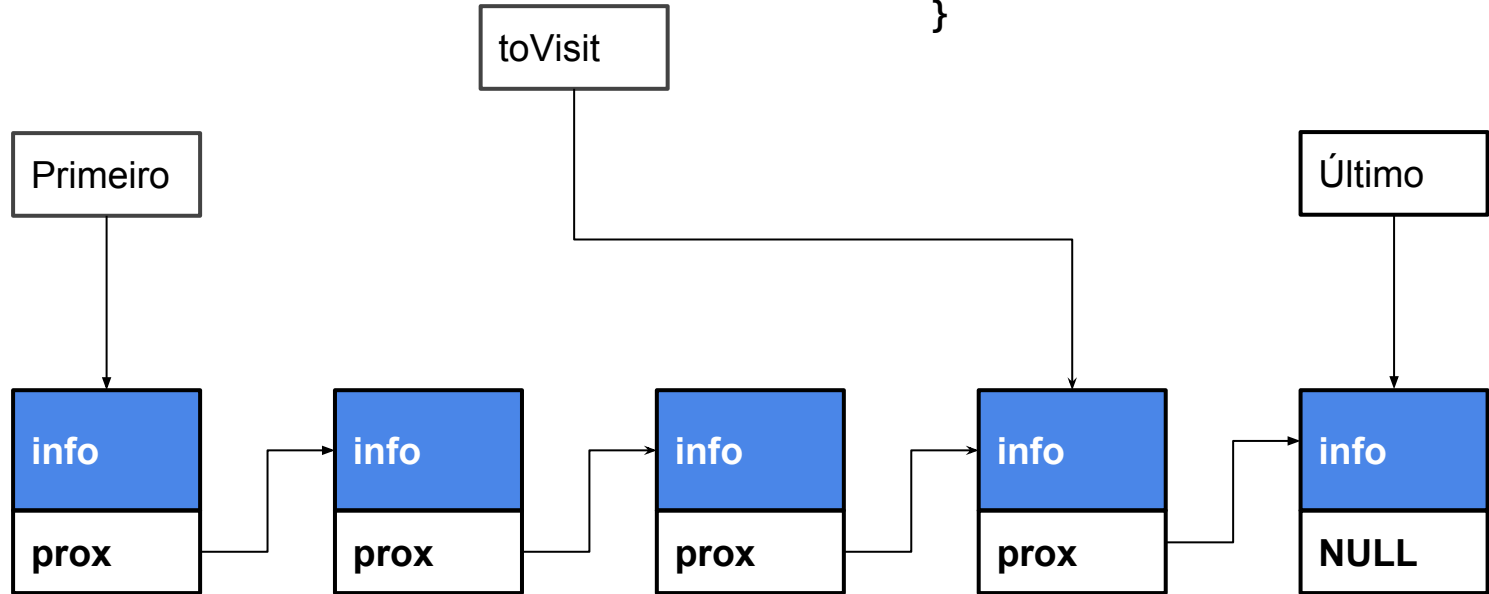
Imprimindo a Lista

```
while (toVisit != NULL) {  
    printf("%d ", toVisit->info);  
    toVisit = toVisit->next;  
}
```



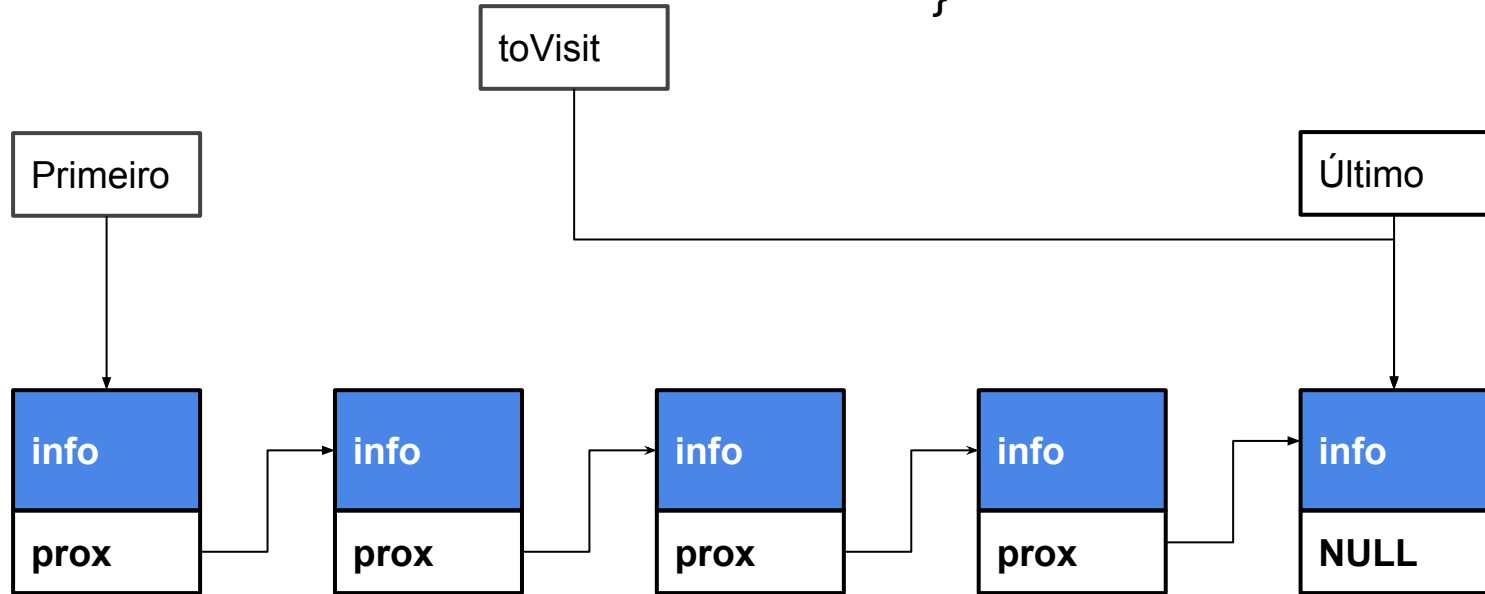
Imprimindo a Lista

```
while (toVisit != NULL) {  
    printf("%d ", toVisit->info);  
    toVisit = toVisit->next;  
}
```



Imprimindo a Lista

```
while (toVisit != NULL) {  
    printf("%d ", toVisit->info);  
    toVisit = toVisit->next;  
}
```



Removendo Elemento do Meio da Lista

- Inverter a ordem um pouco
- Vou assumir que já tenho um ponteiro toFree
 - Aponta para o elemento i que quero remover
- Achar ele é fácil, laço
- Também tenho um ponteiro para o elemento anterior

Removendo Elemento do Meio da Lista

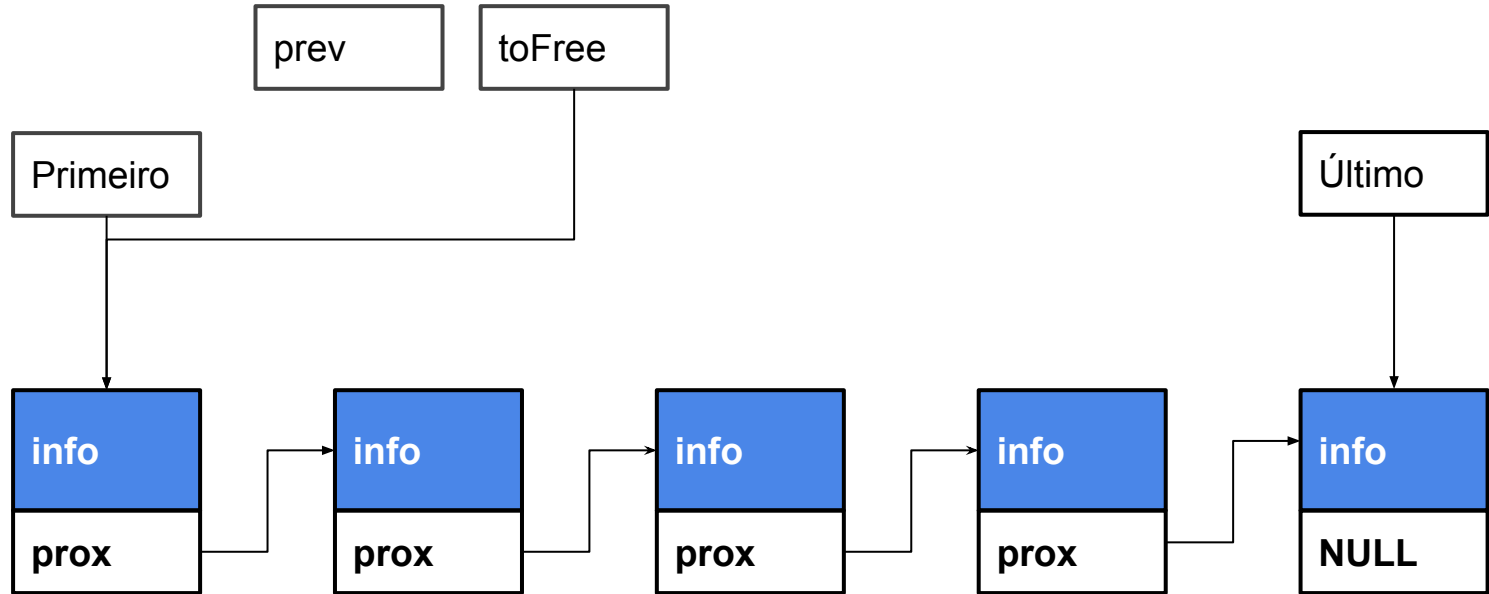
- Inverter a ordem um pouco
- Vou assumir que já tenho um ponteiro toFree
 - Aponta para o elemento i que quero remover
- Achar ele é fácil, laço
- Também tenho um ponteiro para o elemento anterior

```
void removeElement(pointer_list_t *list, int i);
```

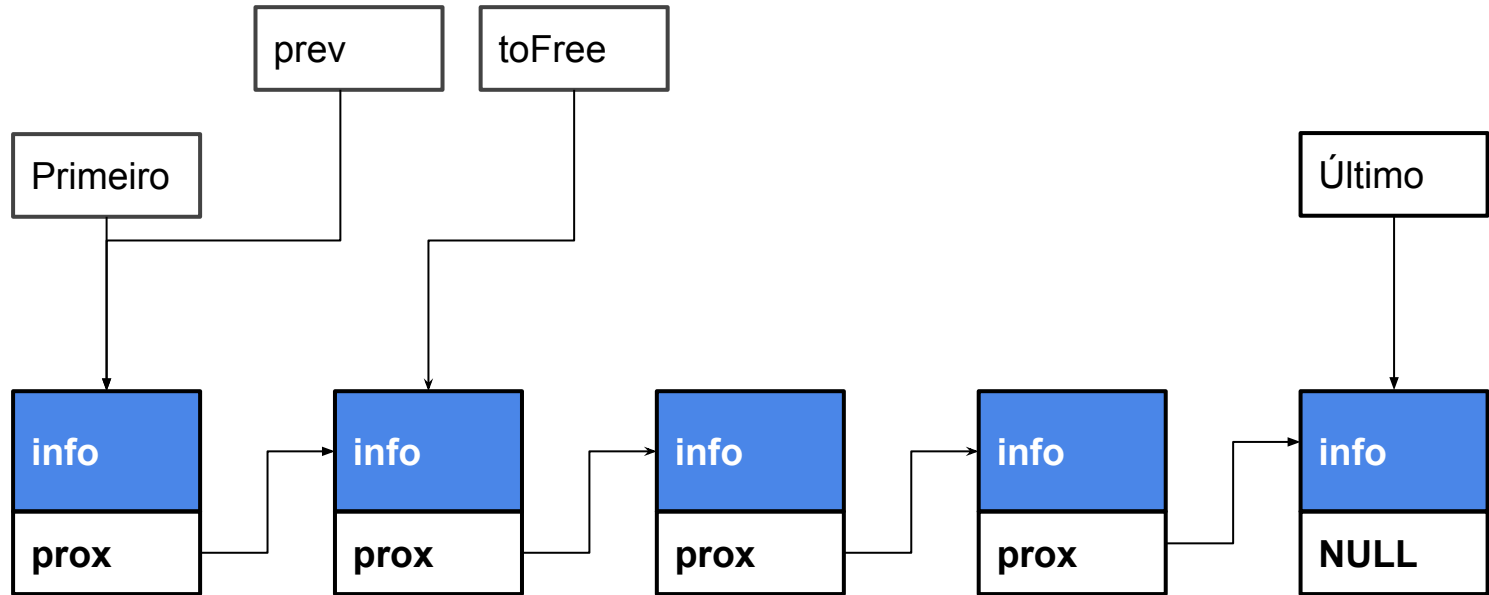
Achando prev e toFree

```
node_t *toFree = list->first;
node_t *prev = NULL;
int curr = 0;
while (toFree != NULL) { //Caminha até achar o elemento
    if (curr == i)
        break;
    prev = toFree;
    toFree = toFree->next;
    curr++;
}
```

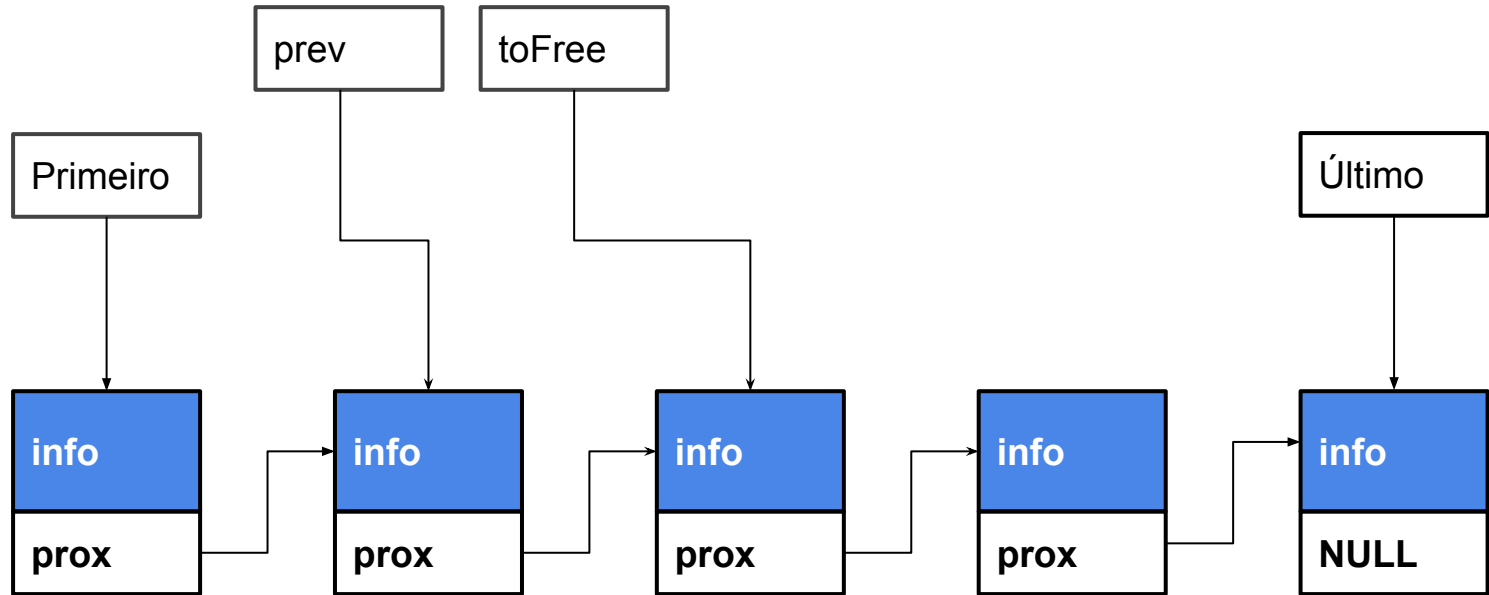
Removendo Elemento do Meio da Lista (i=2)



Removendo Elemento do Meio da Lista (i=2)



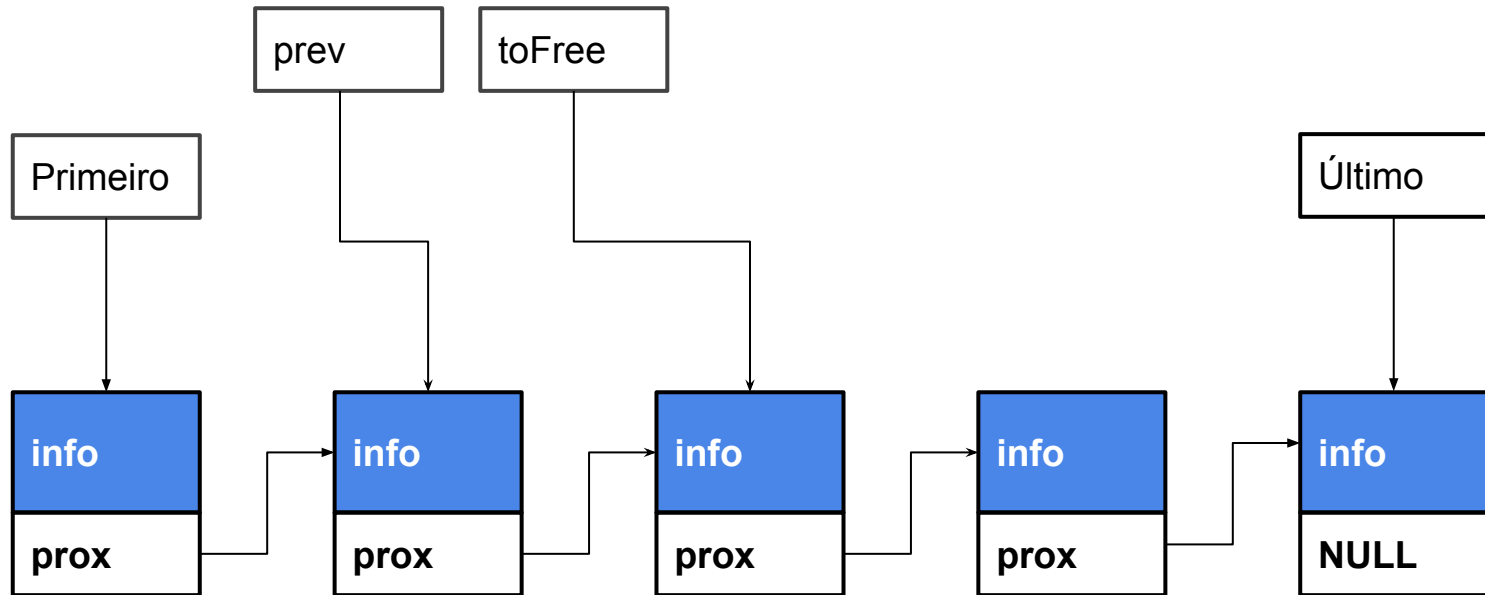
Removendo Elemento do Meio da Lista (i=2)



Removendo

//Free!

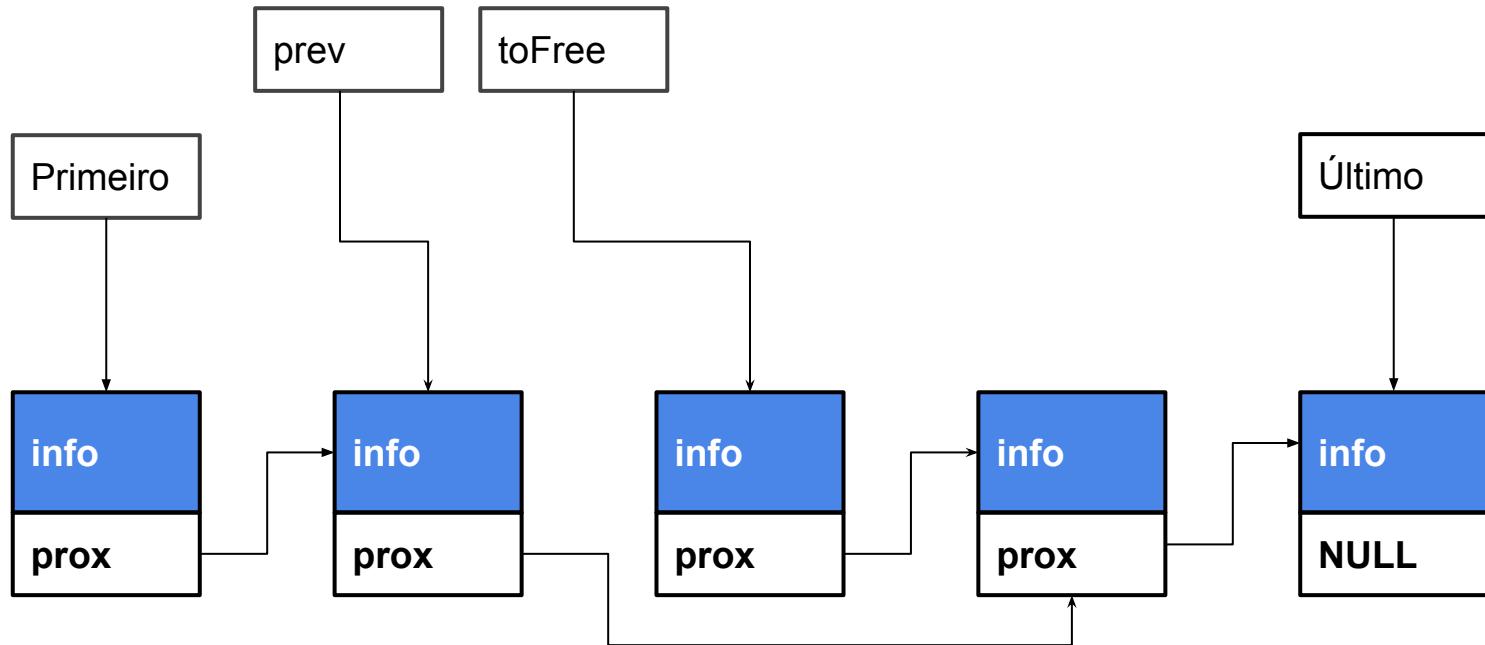
```
prev->next = toFree->next;  
free(toFree);
```



Removendo

//Free!

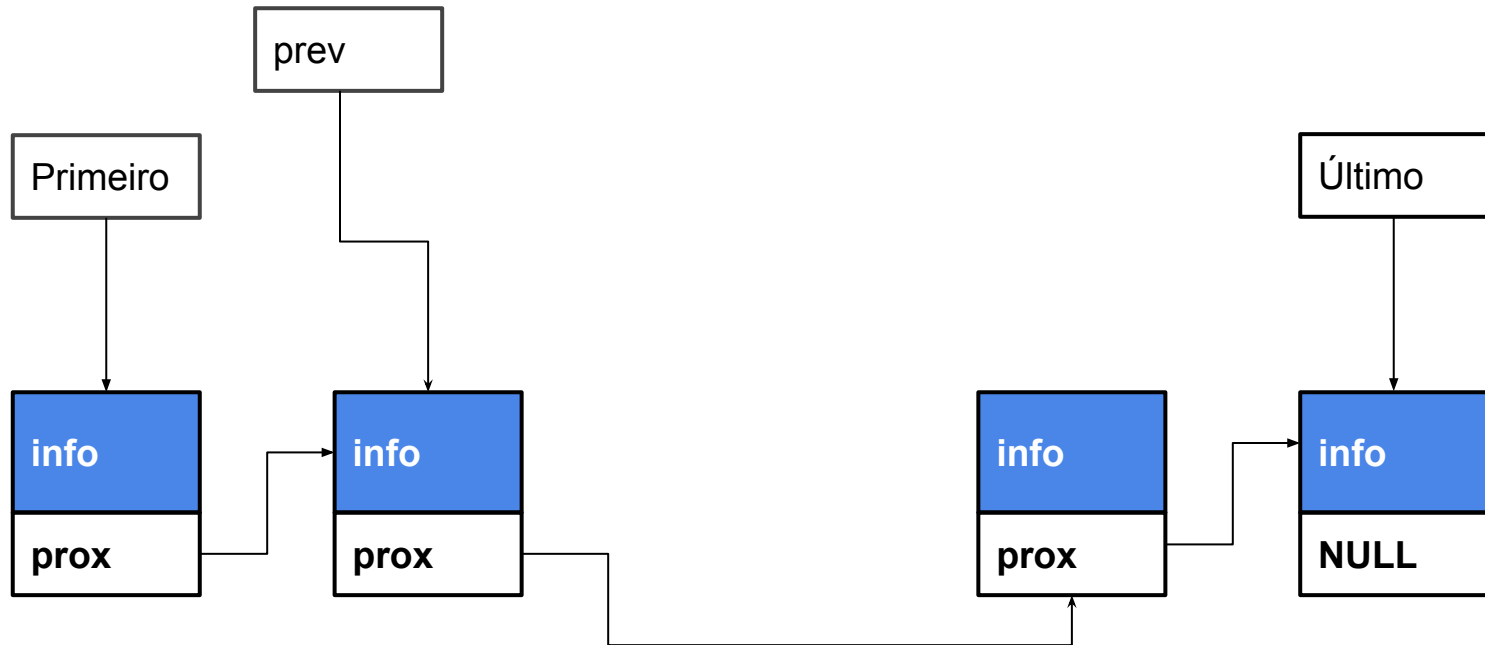
```
prev->next = toFree->next;  
free(toFree);
```



Removendo

//Free!

```
prev->next = toFree->next;  
free(toFree);
```

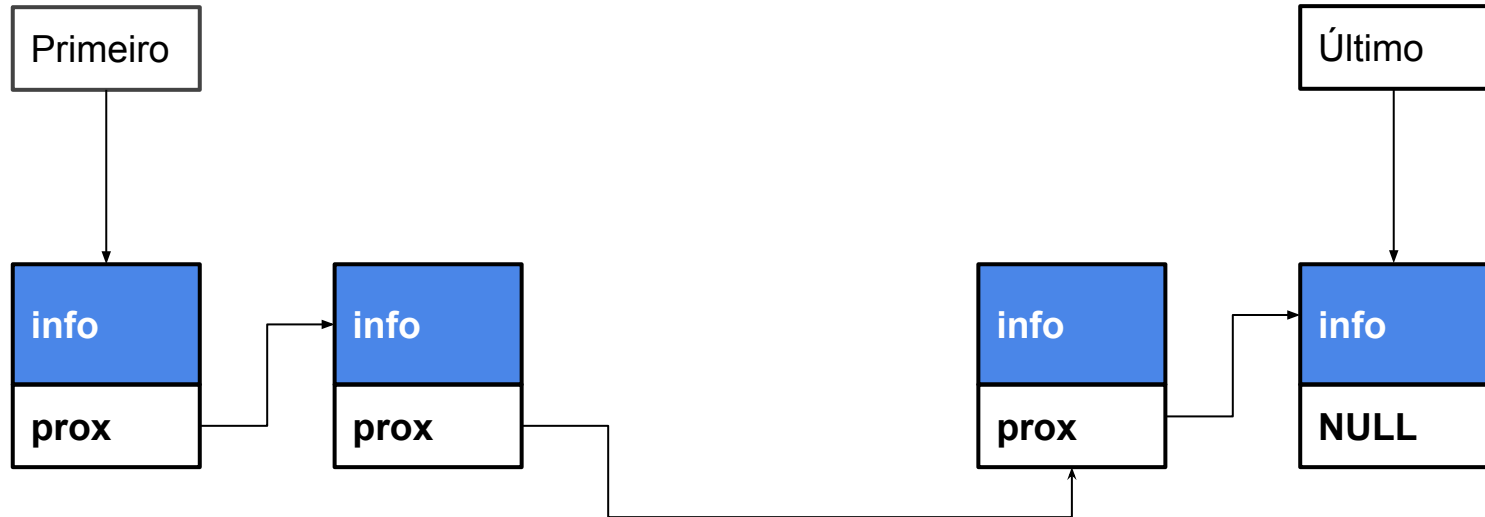


Removendo

```
//Free!
```

```
prev->next = toFree->next;
```

```
free(toFree);
```



Toda Função

- Achar Elemento

```
void removeElement(pointer_list_t *list, int i) {
    node_t *toFree = list->first;
    node_t *prev = NULL;
    int curr = 0;
    while (toFree != NULL) { //Itera até achar o elemento
        if (curr == i)
            break;
        prev = toFree;
        toFree = toFree->next;
        curr++;
    }
    if (toFree == NULL) return; //Lista Vazia ou i > lista
    if (toFree == list->first) //Atualiza first
        list->first = toFree->next;
    if (toFree == list->last) //Atualiza last
        list->last = prev;
    if (prev != NULL) //Atualiza o ponteiro anterior
        prev->next = toFree->next;
    free(toFree);
}
```

Toda Função

- Achar Elemento
-

```
void removeElement(pointer_list_t *list, int i) {
    node_t *toFree = list->first;
    node_t *prev = NULL;
    int curr = 0;
    while (toFree != NULL) { //Itera até achar o elemento
        if (curr == i)
            break;
        prev = toFree;
        toFree = toFree->next;
        curr++;
    }
    if (toFree == NULL) return; //Lista Vazia ou i > lista
    if (toFree == list->first) //Atualiza first
        list->first = toFree->next;
    if (toFree == list->last) //Atualiza last
        list->last = prev;
    if (prev != NULL) //Atualiza o ponteiro anterior
        prev->next = toFree->next;
    free(toFree);
}
```

Toda Função

- Achar Elemento
- Atualizar first e last caso necessário
- Atualizar o ponteiro anterior caso necessário

```
void removeElement(pointer_list_t *list, int i) {
    node_t *toFree = list->first;
    node_t *prev = NULL;
    int curr = 0;
    while (toFree != NULL) { //Itera até achar o elemento
        if (curr == i)
            break;
        prev = toFree;
        toFree = toFree->next;
        curr++;
    }
    if (toFree == NULL) return; //Lista Vazia ou i > lista
    if (toFree == list->first) //Atualiza first
        list->first = toFree->next;
    if (toFree == list->last) //Atualiza last
        list->last = prev;
    if (prev != NULL) //Atualiza o ponteiro anterior
        prev->next = toFree->next;
    free(toFree);
}
```

Toda Função

- Achar Elemento
- Atualizar first e last caso necessário
- Atualizar o ponteiro anterior caso necessário
- Free the malloc!

```
void removeElement(pointer_list_t *list, int i) {
    node_t *toFree = list->first;
    node_t *prev = NULL;
    int curr = 0;
    while (toFree != NULL) { //Itera até achar o elemento
        if (curr == i)
            break;
        prev = toFree;
        toFree = toFree->next;
        curr++;
    }
    if (toFree == NULL) return; //Lista Vazia ou i > lista
    if (toFree == list->first) //Atualiza first
        list->first = toFree->next;
    if (toFree == list->last) //Atualiza last
        list->last = prev;
    if (prev != NULL) //Atualiza o ponteiro anterior
        prev->next = toFree->next;
    free(toFree);
}
```

Exercício

- Inserir um elemento no meio de uma lista

```
void addElement(pointer_list_t *list, int element, int i);
```

Complexidade!

	arraylist_t	pointerlist_t
Inserir Fim	O(n)	O(1)
Remover Fim	O(1) //desperdício mem O(n) //sem desperdício	O(1)
Inserir Início	O(1) //desperdício mem O(n) //sem desperdício	O(1)
Remover Início	O(n)	O(1)
Inserir Meio	O(n)	O(n)
Remover Meio	O(n)	O(n)
Liberar Mem	O(n)	O(n)
Imprimir	O(n)	O(n)

Complexidade!

Mais Comum Inserir/Remover do Início/Fim

	arraylist_t	pointerlist_t
Inserir Fim	O(n)	O(1)
Remover Fim	O(1) //desperdício mem O(n) //sem desperdício	O(1)
Inserir Início	O(1) //desperdício mem O(n) //sem desperdício	O(1)
Remover Início	O(n)	O(1)
Inserir Meio	O(n)	O(n)
Remover Meio	O(n)	O(n)
Liberar Mem	O(n)	O(n)
Imprimir	O(n)	O(n)

Struct com Ponteiro para Void

```
#ifndef POINTER_LIST_H
#define POINTER_LIST_H

typedef struct node {
    void *value;
    struct node *next;
} node_t;

typedef struct {
    node_t *first;
    node_t *last;
} pointer_list_t;

pointer_list_t *createList();
void addElement(pointer_list_t *list, void *element);
void destroyList(pointer_list_t *list);
void removeElement(pointer_list_t *list, int i);
#endif
```

Struct com Ponteiro para Void

- Cuidados especiais com casts
- Cuidados especiais com tamanhos

```
#ifndef POINTER_LIST_H  
#define POINTER_LIST_H
```

```
typedef struct node {  
    void *value;  
    struct node *next;  
} node_t;
```

```
typedef struct {  
    node_t *first;  
    node_t *last;  
} pointer_list_t;
```

```
pointer_list_t *createList();  
void addElement(pointer_list_t *list, void *element);  
void destroyList(pointer_list_t *list);  
void removeElement(pointer_list_t *list, int i);  
#endif
```

Exemplos

<https://github.com/flaviovdf/AEDS2-2017-1/tree/master/exemplos/listas>