PCC104 - Projeto e Análise de Algoritmos

Marco Antonio M. Carvalho

(baseado nas notas de aula do prof. Túlio A. M. Toffolo)
Departamento de Computação
Instituto de Ciências Exatas e Biológicas
Universidade Federal de Ouro Preto

Conteúdo

Fonte

Este material é baseado nos livros

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
- S. Halim. *Competitive Programming*. 3rd Edition, 2013.
- Ian Parberry and William Gasarch. *Problems on Algorithms*. Second Edition, 2002.
- Ian Parberry *Lecture Notes on Algorithm Analysis and Complexity Theory*. Fourth Edition, 2001.

Licença

Definição

Estruturas de dados são um modo particular de armazenamento e organização de coleções de dados.

Estas coleções de dados manipuladas por algoritmos podem aumentar, diminuir, serem alterados e também pesquisados – por isso são chamados de coleções dinâmicas.

Estas coleções podem ser representadas por estruturas de dados simples, que por sua vez podem compor estruturas avançadas.

A representação computacional escolhida deve ser capaz de suportar as operações em coleções dinâmicas.

Operações Frequentes

- **Pesquisar**(S, k) - Dada uma coleção S, pesquisar a existência do elemento k;
- **Inserir**(S, x) - Insere o elemento indicado por x na coleção S;
- **Remover**(S, x) - Remove o elemento indicado por x da coleção S;
- **Mínimo**(S) - Pesquisa a coleção S e retorna a menor chave encontrada;
- **Máximo**(S) - Pesquisa a coleção S e retorna a maior chave encontrada;
- **Sucessor**(S, x) - Pesquisa a coleção ordenada S e retorna o elemento após x, ou, caso x seja o último elemento, retorna um valor nulo;
- **Antecessor**(S, x) - Pesquisa a coleção ordenada S e retorna o elemento anterior a x, ou, caso x seja o primeiro elemento, retorna um valor nulo.

Descrição

Sequência linear de zero ou mais elementos $x_0, x_1, \ldots, x_{n-1}$, na qual x_i é de um determinado tipo e n representa o tamanho da lista.

Sua principal propriedade estrutural envolve as posições relativas dos elementos em uma dimensão:

- Assumindo $n > 0$, x_0 é o primeiro item da lista e x_{n-1} é o último item da lista;
- x_i precede x_{i+1} para $i = 0, 2, \ldots, n\text{-}2$;
- x_i sucede x_{i-1} para $i = 1, 3, \ldots, n\text{-}1$.

Operações

Um conjunto de operações necessário a uma maioria de aplicações é:

- Criar uma lista linear vazia;
- Inserir um novo item imediatamente após o i-ésimo item;
- Remover o i-ésimo item;
- Localizar o i-ésimo item para examiná-lo e/ou alterá-lo;
- Combinar duas ou mais listas lineares em uma lista única;
- Dividir uma lista linear em duas ou mais listas;
- Fazer uma cópia da lista linear;
- Ordenar os elementos da lista em ordem ascendente ou descendente;
- Pesquisar a ocorrência de um item com um valor particular em algum elemento.

Implementações

Várias estruturas de dados podem ser usadas para representar listas lineares, cada uma com vantagens e desvantagens particulares.

As duas representações mais utilizadas são:

- Implementação por arranjos;
- Implementação por ponteiros.

Implementação por Arranjos

Os elementos são armazenados em um arranjo de tamanho suficiente para armazenar a lista.

O campo *último* aponta para a posição seguinte à do último elemento da lista.

O i-ésimo elemento da lista está armazenado na (i-1)-ésima posição do arranjo, $0 \leq i <$ *último*.

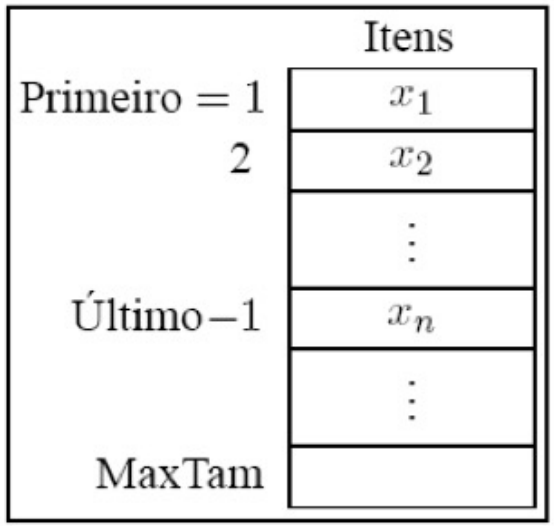
A constante *MaxTam* define o tamanho máximo permitido para a lista.

Implementação por Arranjos

Os elementos da lista são armazenados em posições contíguas de memória.

A lista pode ser percorrida em qualquer direção.

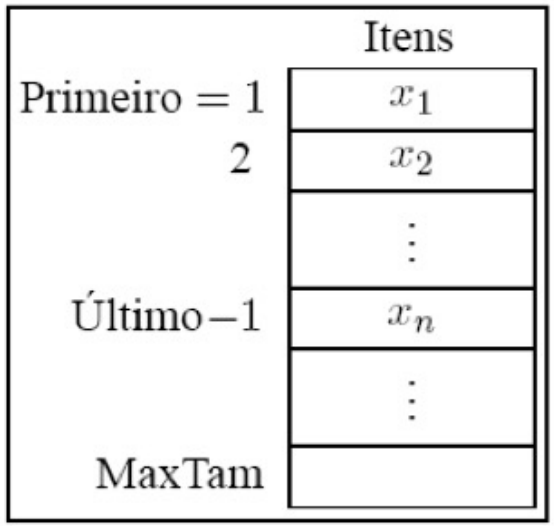
A inserção de um novo item no final tem custo $O(\mathbf{1})$.



Implementação por Arranjos

A inserção de um novo item no meio da lista requer um deslocamento de todos os elementos, com custo custo $O(n)$.

Retirar um item do início da lista requer deslocamentos para preencher o espaço vazio, com custo $O(n)$.



Operações em Listas Usando Arranjos com Cabeça

```c
#include <stdio.h>
#include <stdlib.h>

#define INICIO 0
#define MAXTAM 1000

typedef struct {
    int chave;
    // outros componentes
} TItem;

typedef struct {
    TItem item[MAXTAM];
    int primeiro, ultimo;
} TLista;
```

Operações em Listas Usando Arranjos com Cabeça

```
void TLista_FazVazia(TLista *pLista) {
    pLista->primeiro = INICIO;
    pLista->ultimo = pLista->primeiro;
}

int TLista_EhVazia(Tlista *pLista) {
    return (pLista->ultimo == pLista->primeiro);
}

int TLista_Insere(Tlista *pLista, TItem x) {
    if (pLista->ultimo == MaxTam)
        return 0; // lista cheia
    pLista->item[pLista->ultimo++] = x;
    return 1;
}
```

Operações em Listas Usando Arranjos com Cabeça

```
int TLista_Retira(Tlista *pLista, int p, TItem *pX) {
    if (LEhVazia(pLista) || p >= pLista->ultimo)
        return 0;

    int cont;
    *pX = pLista->Item[p];
    pLista->ultimo--;
    for (cont = p+1; cont <= pLista->ultimo; cont++)
        pLista->item[cont - 1] = pLista->item[cont];
    return 1;
}
```

Operações em Listas Usando Arranjos com Cabeça

```
void TLista_Imprime(Tlista *pLista) {
    int i;
    for (i = pLista->primeiro; i < pLista->ultimo; i++)
        printf("%d\n", pLista->item[i].chave);
}
```

Resumo da Complexidade – Implementação por Arranjos

- Inserção
  - No início: $O(n)$;
  - No fim: $\Theta(1)$.
- Remoção
  - No início: $O(n)$;
  - No meio: $O(n)$;
  - No fim: $\Theta(1)$.
- Pesquisa: $O(n)$.

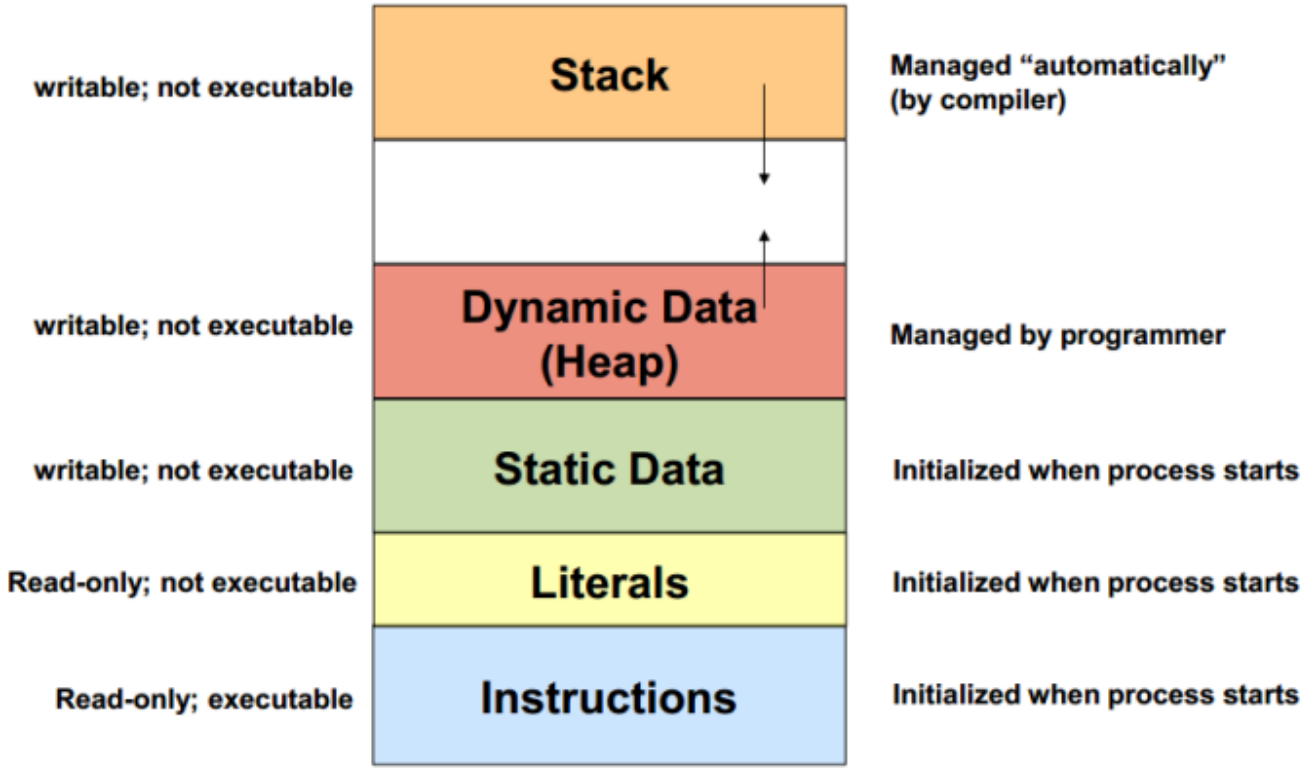
Vantagens de Listas Implementadas por Arranjos

- Economia de memória (os apontadores são implícitos nesta estrutura);
- Acesso a qualquer elemento da lista é feito em tempo $O(1)$.

Desvantagens de Listas Implementadas por Arranjos

- Custo para inserir elementos da lista pode ser $O(n)$;
- Custo para remover elementos da lista pode ser $O(n)$;
- Quando não existe previsão sobre o crescimento da lista, o arranjo que define a lista deve ser alocado de forma dinâmica;
- O arranjo pode precisar ser realocado completamente.

Esquema de Memória



Alocação Estática vs. Alocação Dinâmica

Na alocação estática, o espaço para as variáveis é reservado automaticamente, e liberado posteriormente pelo compilador:

- int a; int b[20];

Na alocação dinâmica, o espaço para as variáveis é alocado dinamicamente durante a execução do programa pelo programador:

- int *a = (int*) malloc(sizeof(int));

Ponteiros

As variáveis alocadas dinamicamente são chamadas de **ponteiros** ou **apontadores**, pois armazenam o endereço de memória de outra variável

- Número inteiro (32 ou 64 bits) indicando um endereço da memória.

A memória alocada dinamicamente faz parte de uma área da memória chamada *heap*.

É possível alocar e desalocar porções de memória do *heap* durante a execução.

Liberação de Memória

A memória deve ser liberada após o término de seu uso.

A liberação deve ser feita por quem fez a alocação:

- Alocação Estática: compilador.
- Alocação Dinâmica: programador.

Alocação Dinâmica

Dinâmica (C)

- Declaração da variável
  - Tipo *p;
- Alocação de memória
  - p = (Tipo*) malloc(sizeof(Tipo));
- Liberação de memória
  - free(p);
- Acesso ao conteúdo
  - *p;
- Endereço de memória apontado
  - p;
- Valor nulo
  - NULL;

Estática

- Declaração da variável
  - Tipo p;
- Acesso ao conteúdo
  - p;
- Endereço de memória
  - &p;

Alocação Dinâmica

Dinâmica (C++)

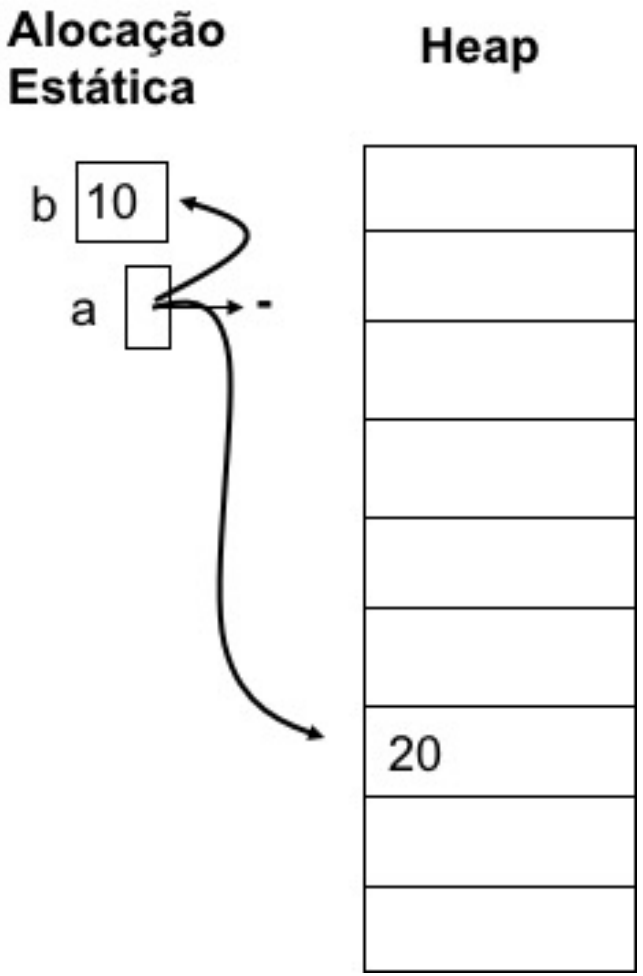
- Declaração da variável
  - Tipo *p;
- Alocação de memória
  - p = new (Tipo);
- Liberação de memória
  - delete(p);
- Acesso ao conteúdo
  - *p;
- Endereço de memória apontado
  - p;
- Valor nulo
  - nullptr;

Estática

- Declaração da variável
  - Tipo p;
- Acesso ao conteúdo
  - p;
- Endereço de memória
  - &p;

Alocação Dinâmica

```
int *a, b;
...
b = 10;
a = (int *) malloc(sizeof(int));
*a = 20;
a = &b;
```



Alocação Dinâmica

Erros Comuns

- Esquecer de alocar memória e tentar acessar o conteúdo da variável;
- Copiar o valor do apontador ao invés do valor da variável apontada;
- Esquecer de desalocar memória;
- Alocar/desalocar memória dentro de loops ou recursivamente;
- Tentar acessar o conteúdo da variável depois de desalocá-la.

Uso Comum

Apontadores são normalmente utilizados com tipos estruturados.

```
typedef struct {
  int idade;
  double salario;
} TRegistro;

int main() {
  TRegistro *a;
  ...
  a = (TRegistro *) malloc(sizeof(TRegistro))
  a->idade = 30; // (*a).idade = 30
  a->salario = 80;
  ...
}
```

Listas Encadeadas

Qual o principal problema de utilizar arranjo para implementar listas?

- A lista aumentar e depois diminuir drasticamente de tamanho.

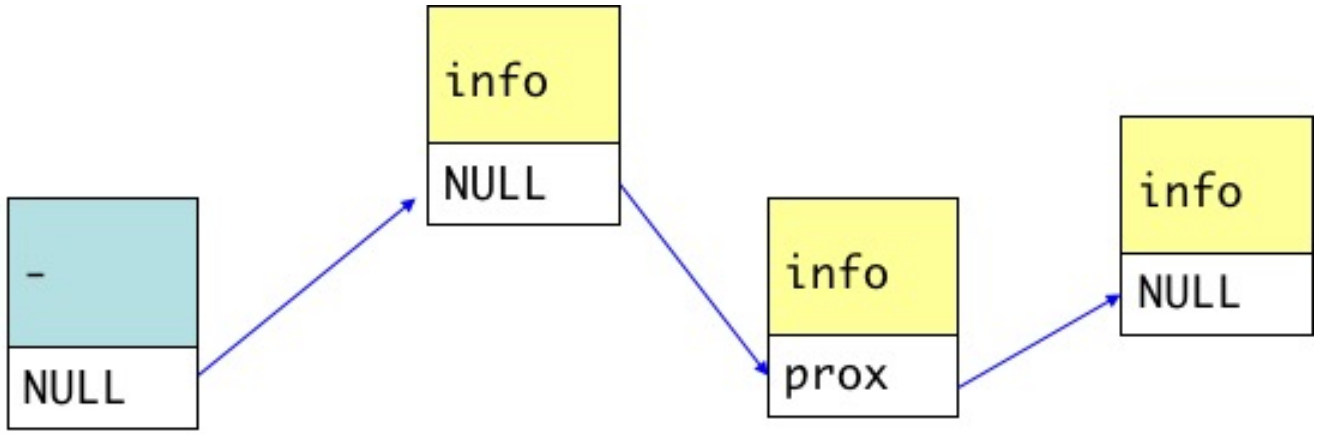
Solução: listas encadeadas!

- Implementação de uma lista utilizando apenas ponteiros.

Listas Encadeadas

Características:

- Tamanho da lista não é pré-definido;
- Cada elemento guarda quem é o próximo;
- Elementos não estão contíguos na memória.

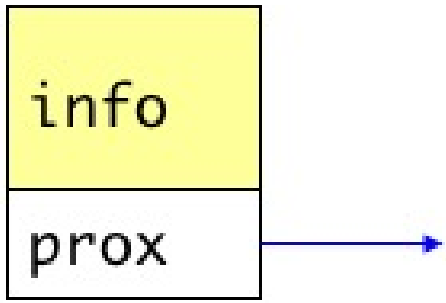


Estrutura

A unidade básica de informação de uma lista encadeada é um **elemento**, que armazena as informações sobre cada elemento e também sobre o próximo.

Para isso define-se cada elemento como uma estrutura que possui:

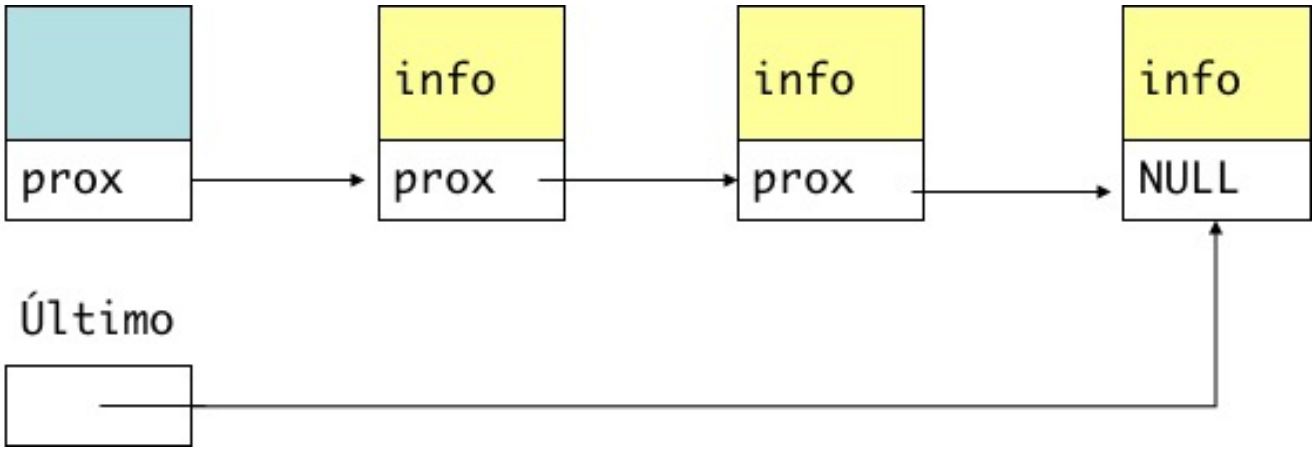
- Campos de informações;
- Ponteiro para o próximo elemento.



Estrutura

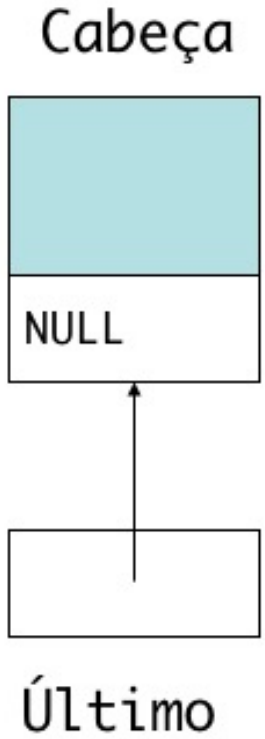
Opcionalmente,

- Uma lista pode ter uma célula cabeça;
- Uma lista pode ter um apontador para o último elemento.



Criação de uma lista vazia

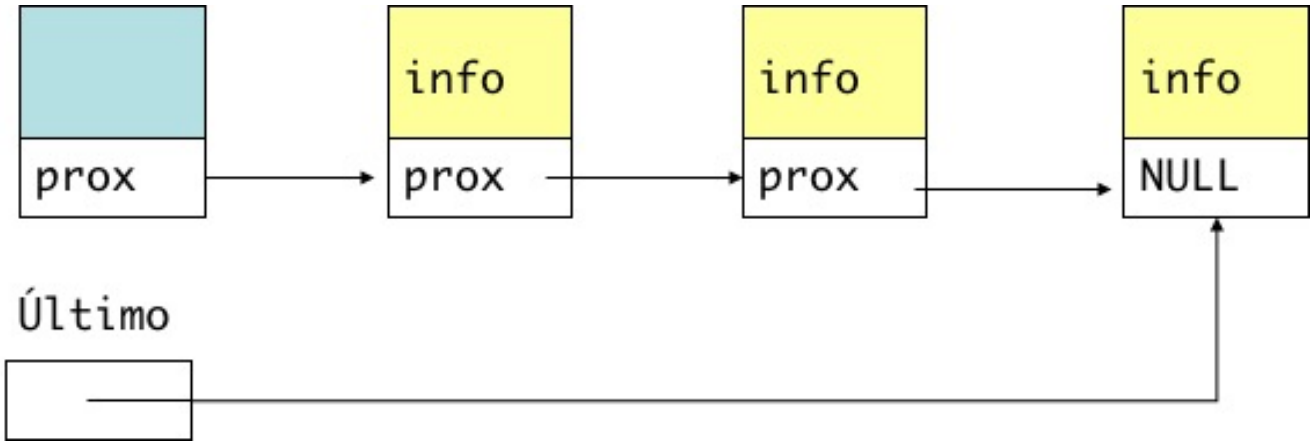
Para criarmos uma lista vazia, podemos fazer com que a cabeça seja o último elemento.



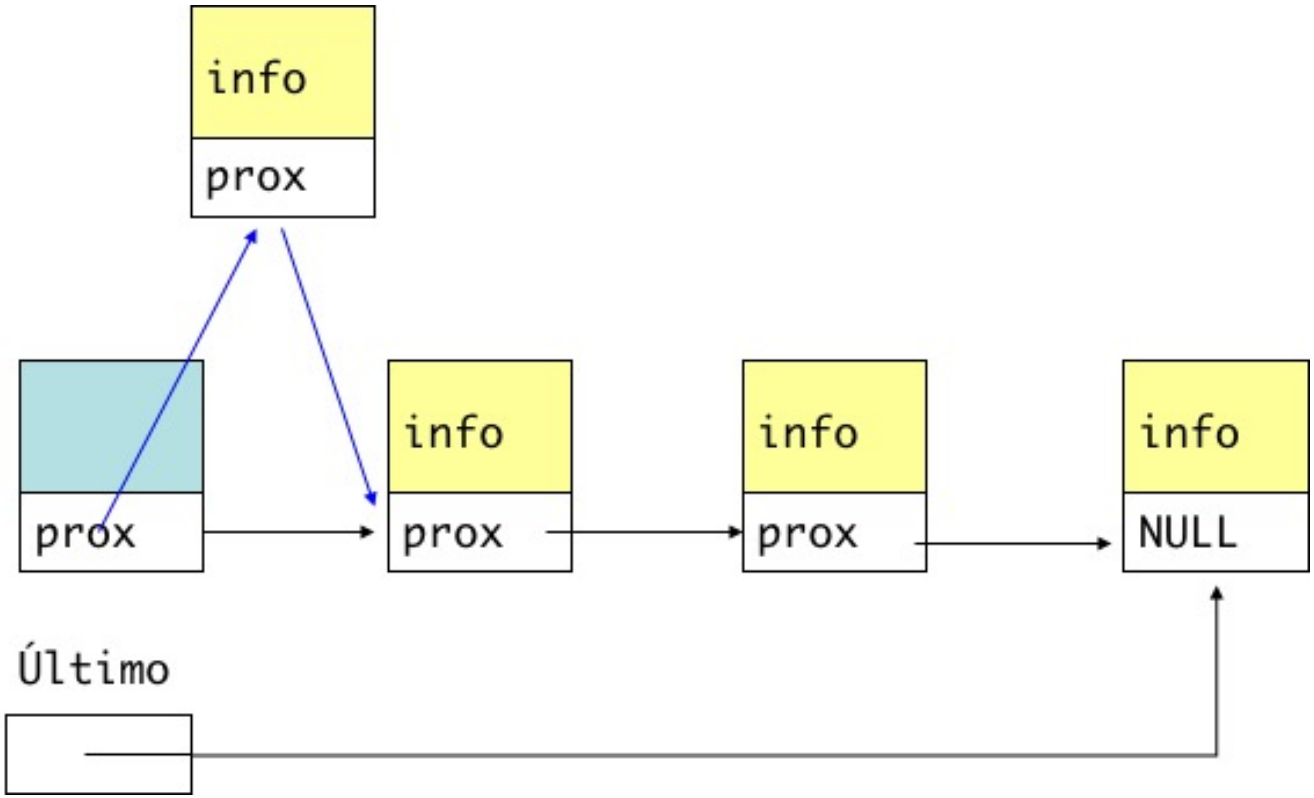
Inserção

Temos três opções de posição onde pode inserir um novo elemento:

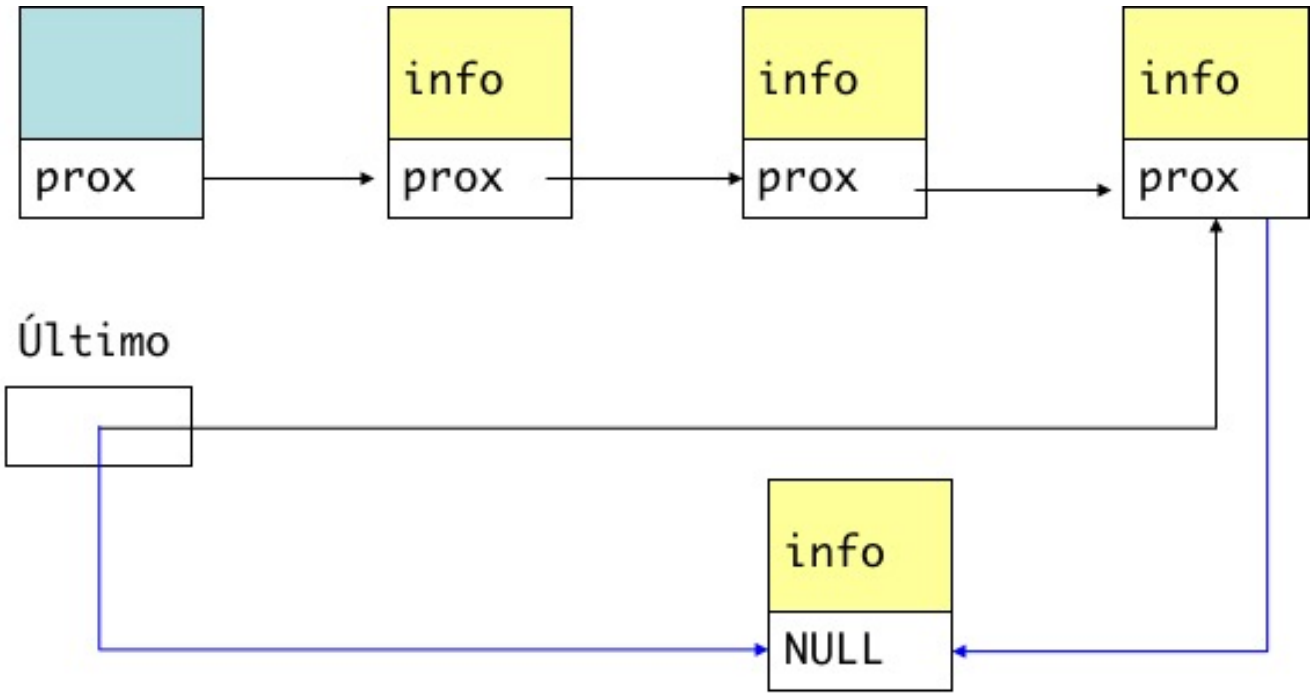
- Primeira posição;
- Última posição;
- Após um elemento qualquer E.



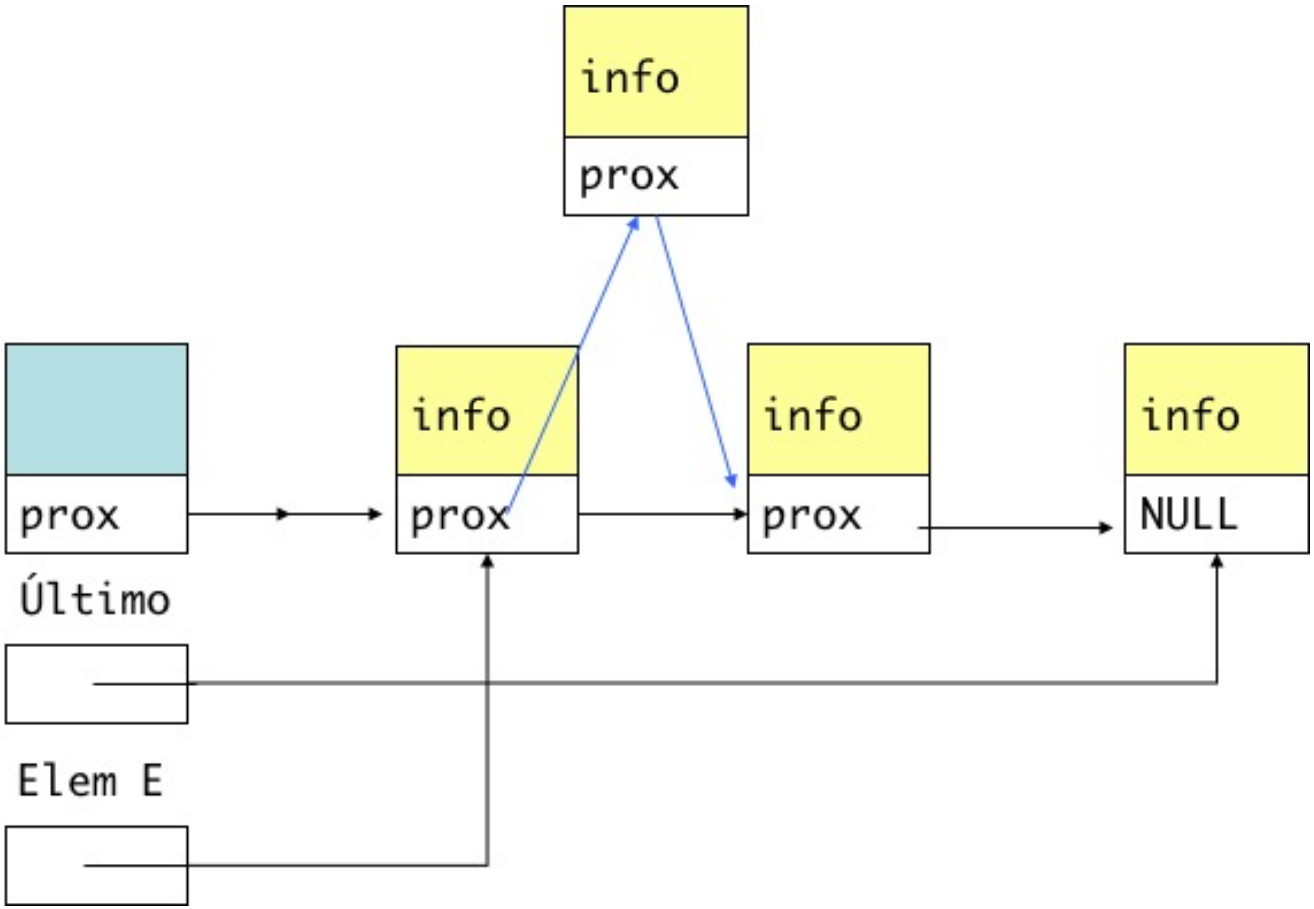
Inserção na Primeira Posição



Inserção na Última Posição



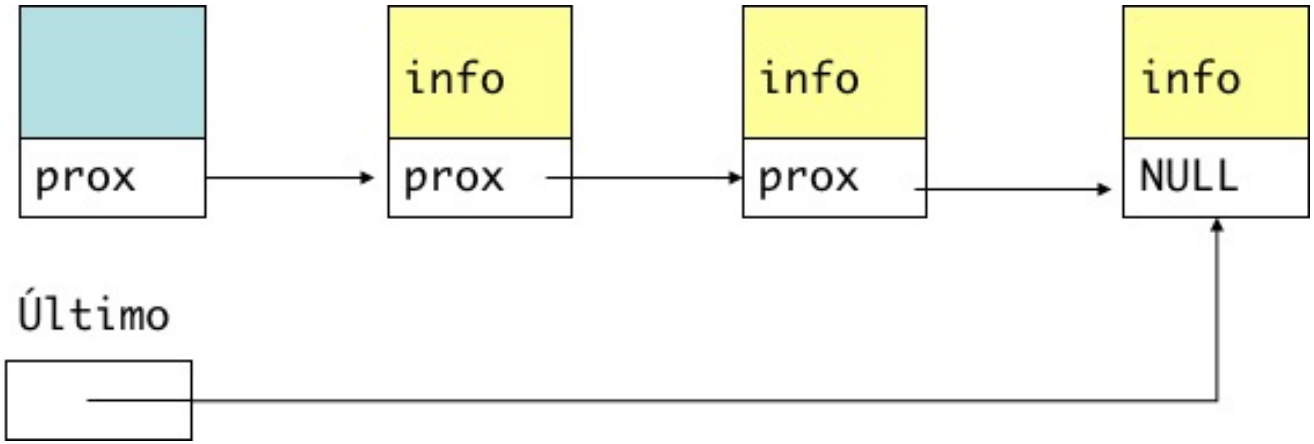
Inserção Após o Elemento E



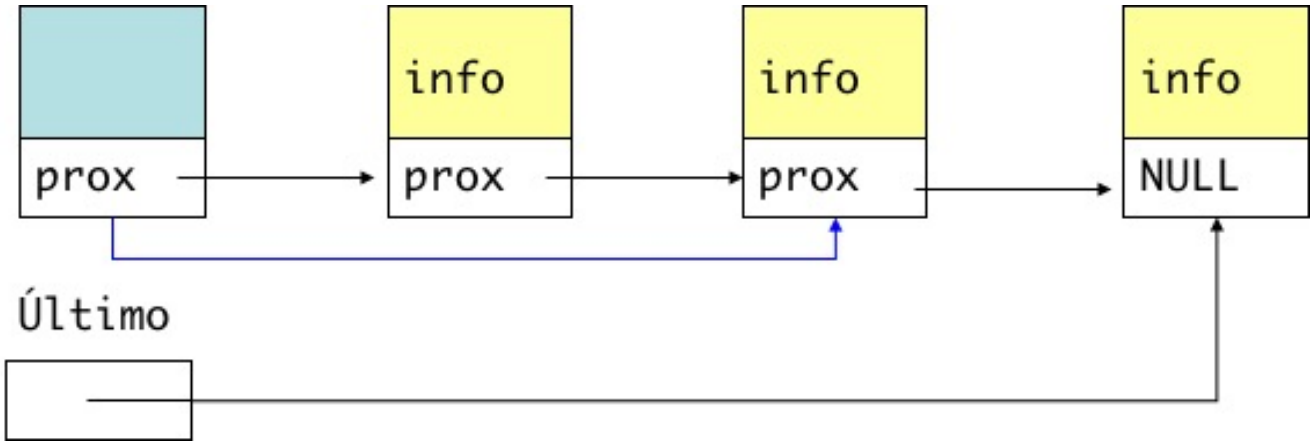
Remoção

Temos três opções de posição de onde remover um elemento da lista:

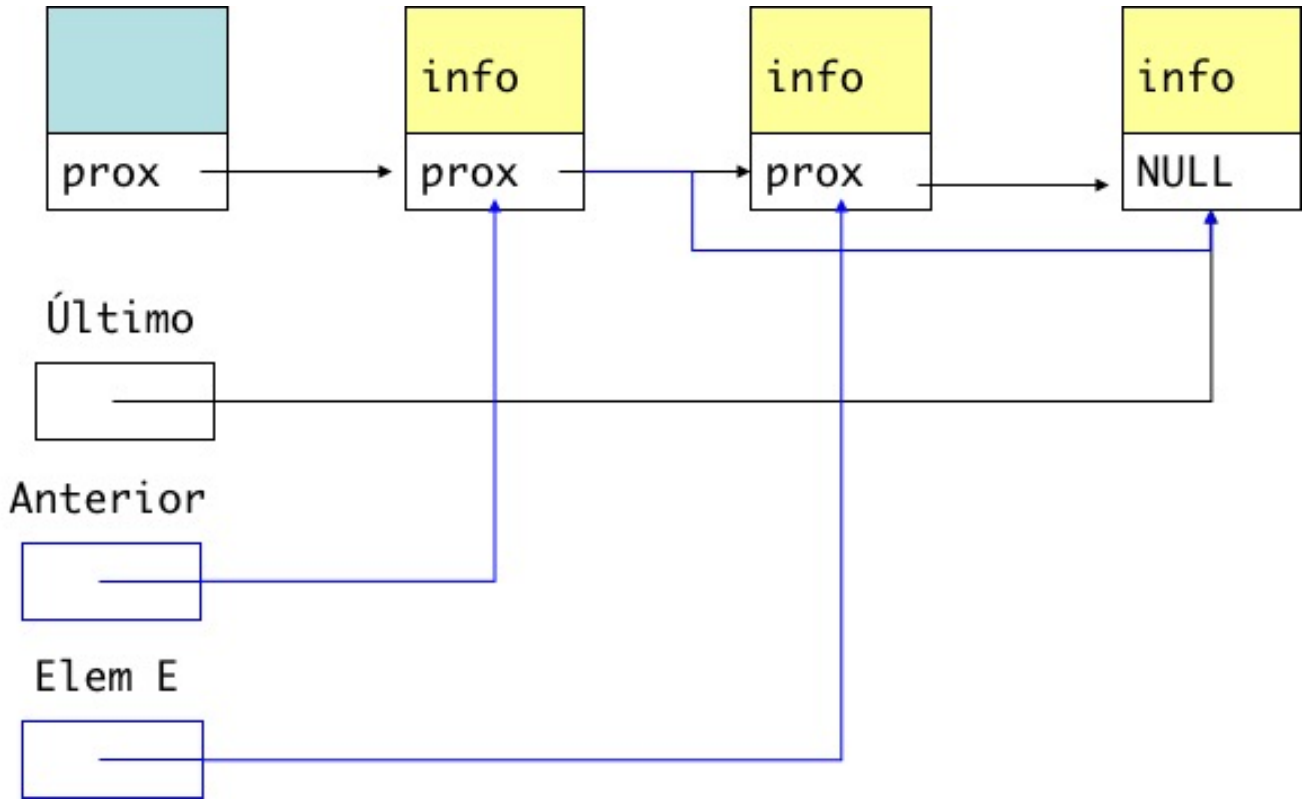
- Primeira posição.
- Última posição.
- Após um elemento qualquer E.



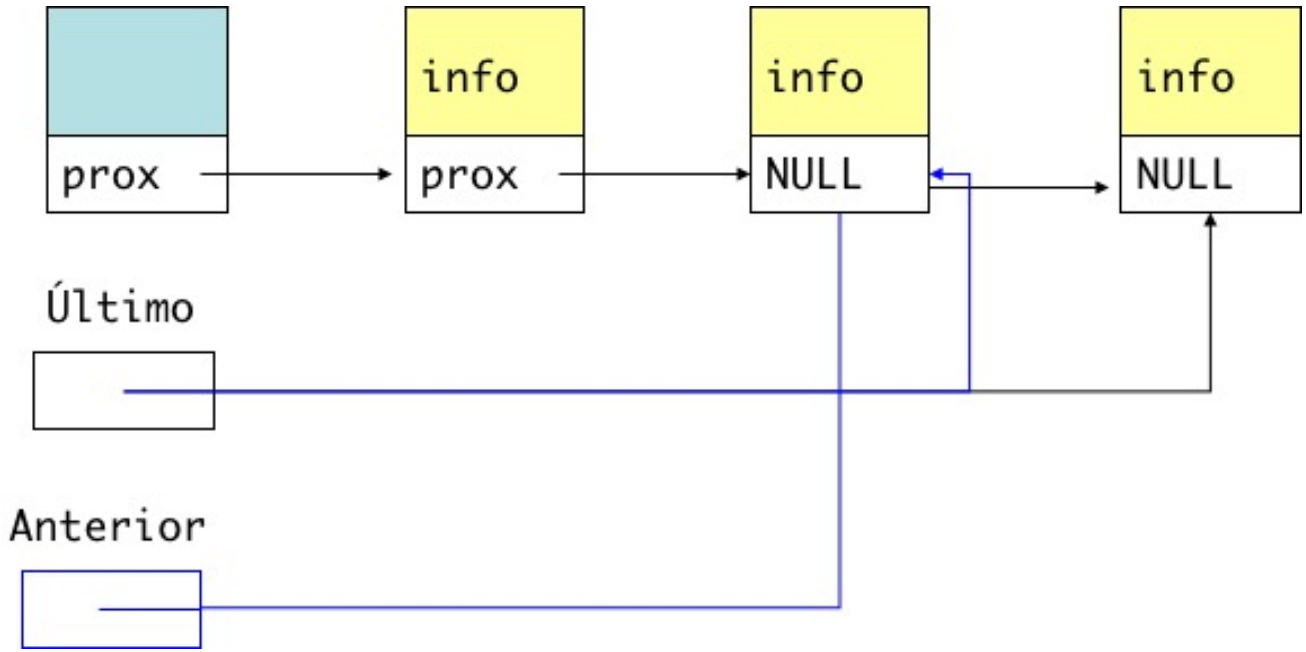
Remoção do Elemento da Primeira Posição



Remoção do Elemento E da Lista



Remoção do Último Elemento da Lista



Operações em Listas Encadeadas com Cabeça

```
typedef struct {
    int chave;
    // outros componentes
} TItem;

typedef struct celula {
    struct celula *pProx;
    TItem item;
    // outros componentes
} TCelula;

typedef struct {
    TCelula *pPrimeiro, *pUltimo;
} TLista;
```

Operações em Listas Encadeadas com Cabeça

```
// Inicia as variaveis da lista
void TLista_Inicia(TLista *pLista) {
    pLista->pPrimeiro = (TCelula*) malloc(sizeof(TCelula));
    pLista->pUltimo = pLista->pPrimeiro;
    pLista->pPrimeiro->pProx = NULL;
}

//Retorna se a lista esta vazia
int TLista_EhVazia(TLista *pLista) {
    return (pLista->pPrimeiro == pLista->pUltimo);
}
```

Operações em Listas Encadeadas com Cabeça

```
// Insere um item no final da lista
int TLista_InsereFinal(TLista *pLista, TItem x) {
    pLista->pUltimo->pProx = (TCelula*) malloc(sizeof
                                        (TCelula));
    pLista->pUltimo = pLista->pUltimo->pProx;
    pLista->pUltimo->Item = x;
    pLista->pUltimo->pProx = NULL;
}
```

Operações em Listas Encadeadas com Cabeça

```
// Insere um item no inicio da lista
int TLista_InsereInicio(TLista *pLista, TItem x) {
    TCelula* aux;
    aux = (TCelula*) malloc(sizeof (TCelula));
    aux->prox  =  pLista->pPrimeiro->pProx;
    aux->item = x;
    pLista->pPrimeiro->pProx = aux;
    }
```

Operações em Listas Encadeadas com Cabeça

```
// Retira o primeiro item da lista
int TLista_RetiraPrimeiro(TLista *pLista, TItem *pX) {
    if (TLista_EhVazia(pLista))
        return 0;

    TCelula *pAux;
    pAux = pLista->pPrimeiro->pProx;
    *pX = pAux->item;
    pLista->pPrimeiro->pProx = pAux->pProx;
    free(pAux);
    return 1;
}
```

Operações em Listas Encadeadas com Cabeça

```
// Imprime os elementos da lista
void TLista_Imprime(TLista *pLista) {
    TCelula *pAux;
    pAux = pLista->pPrimeiro->pProx;
    while (pAux != NULL) {
        printf("%d\n", pAux->item.chave);
        pAux = pAux->pProx; // proxima celula
    }
}
```

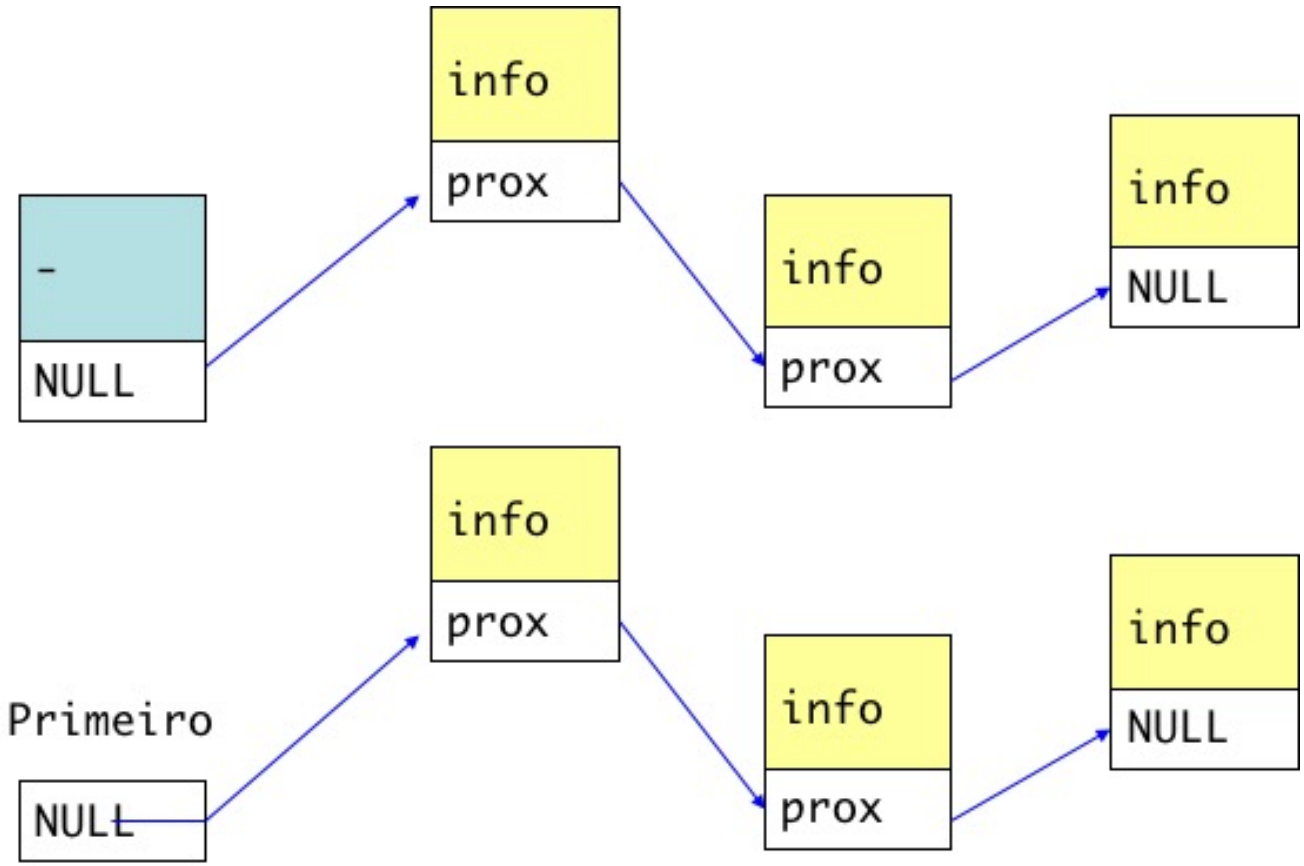
Vantagens de Listas Implementadas por Ponteiros

- Permite inserir ou retirar itens do meio da lista a um custo constante (importante quando a lista tem de ser mantida em ordem);
- Bom para aplicações em que não existe previsão sobre o crescimento da lista (o tamanho máximo da lista não precisa ser definido *a priori*).

Desvantagens de Listas Implementadas por Ponteiros

- Utilização de memória extra para armazenar os apontadores;
- Percorrer a lista, procurando o i-ésimo elemento, tem custo $O(n)$.

Listas Encadeadas Com e Sem Cabeça

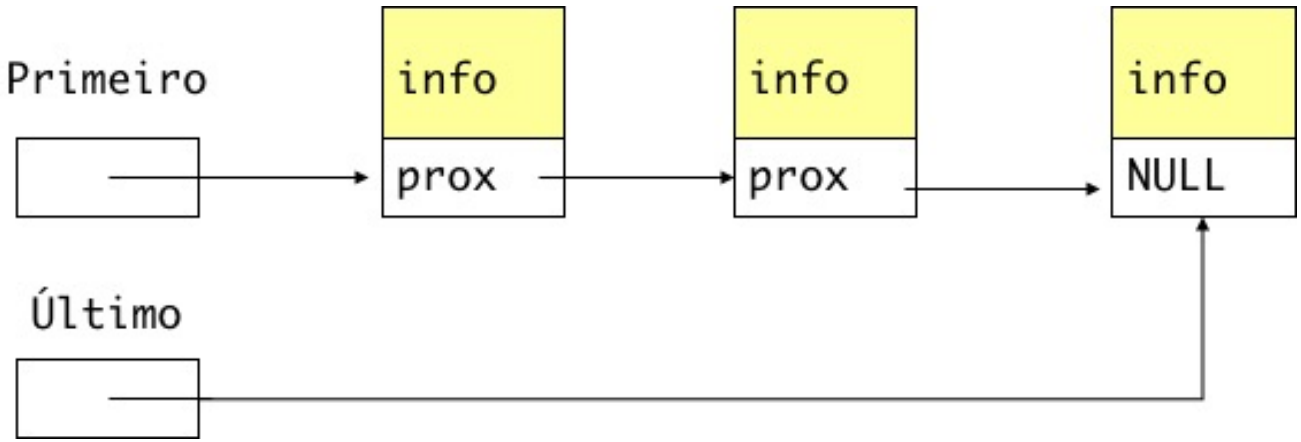


Lista Encadeada Sem Cabeça

Estrutura

Opcionalmente,

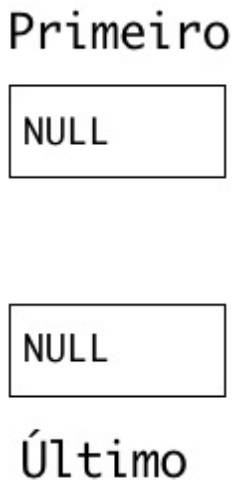
- Uma lista pode ter um apontador para o primeiro elemento;
- Uma lista pode ter um apontador para o último elemento.



Lista Encadeada Sem Cabeça

Criação de uma lista vazia

Para criarmos uma lista vazia, podemos fazer com que o ponteiros para o primeiro e último elementos sejam nulos.

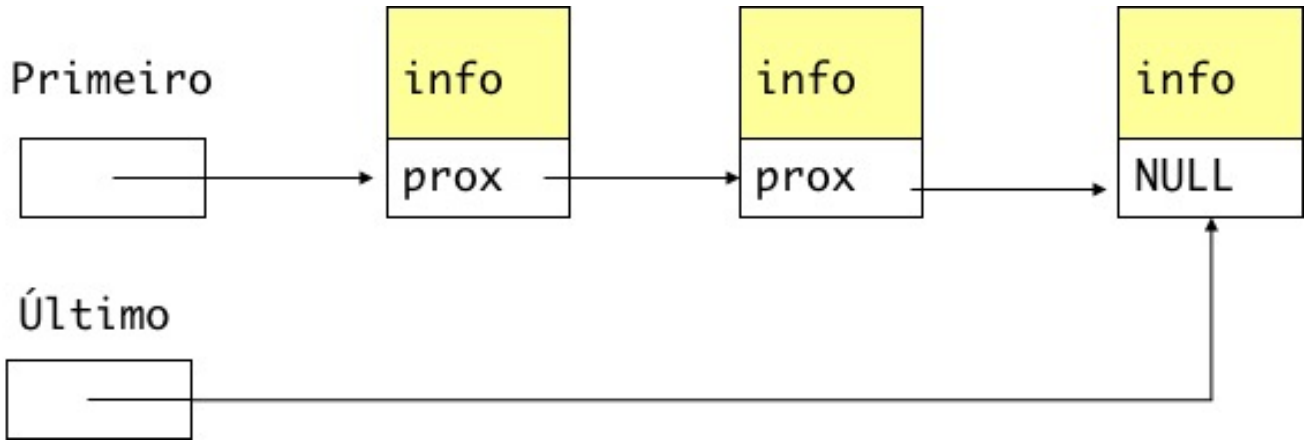


Lista Encadeada Sem Cabeça

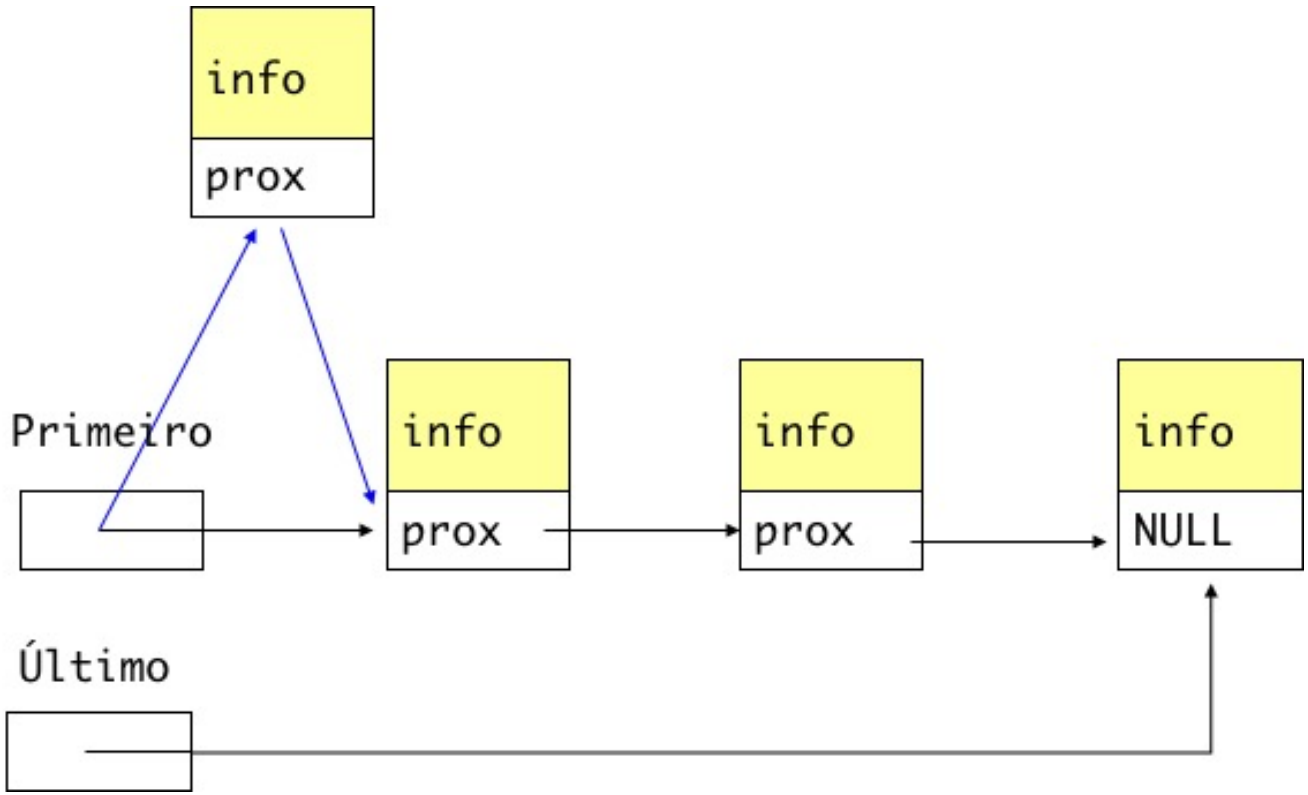
Inserção e Remoção

Novamente, temos três opções para inserção e remoção em listas encadeadas sem cabeça:

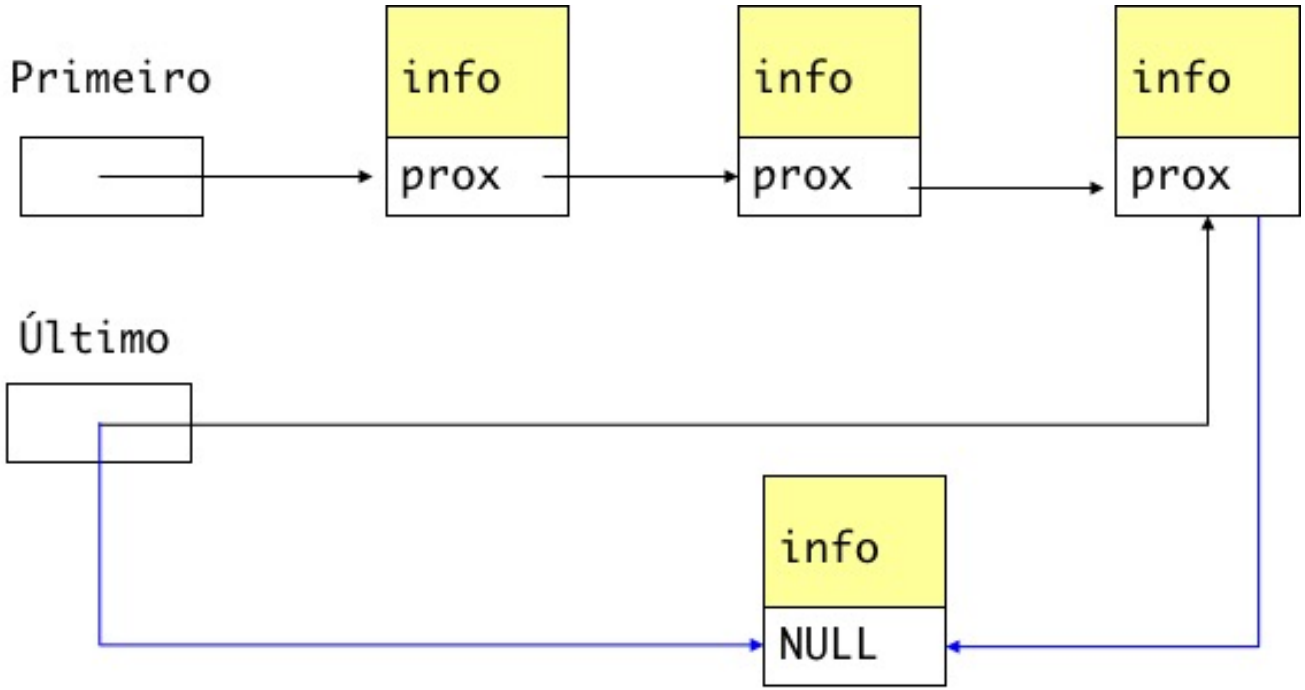
- Primeira posição;
- Última posição;
- Em relação a um elemento qualquer E.



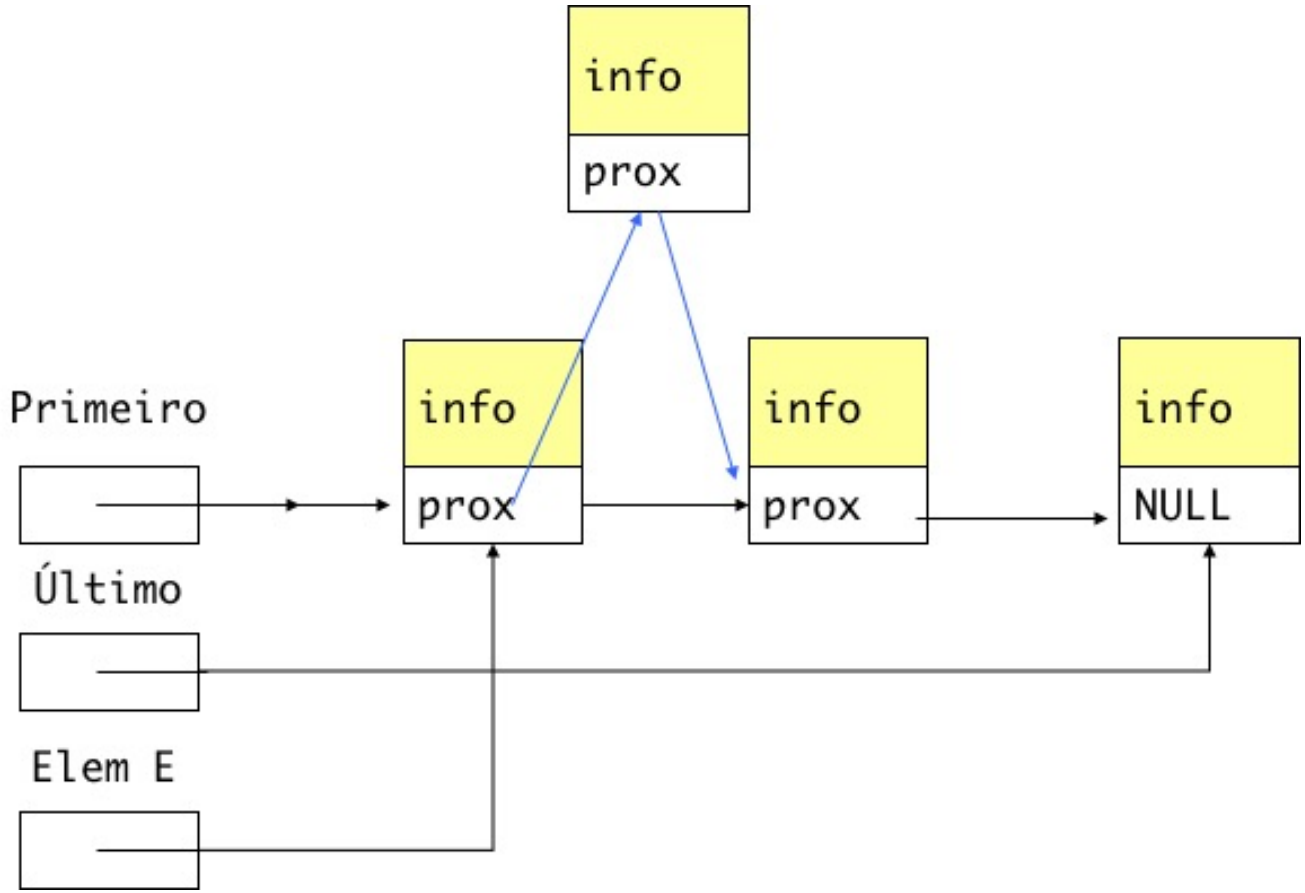
Inserção na Primeira Posição



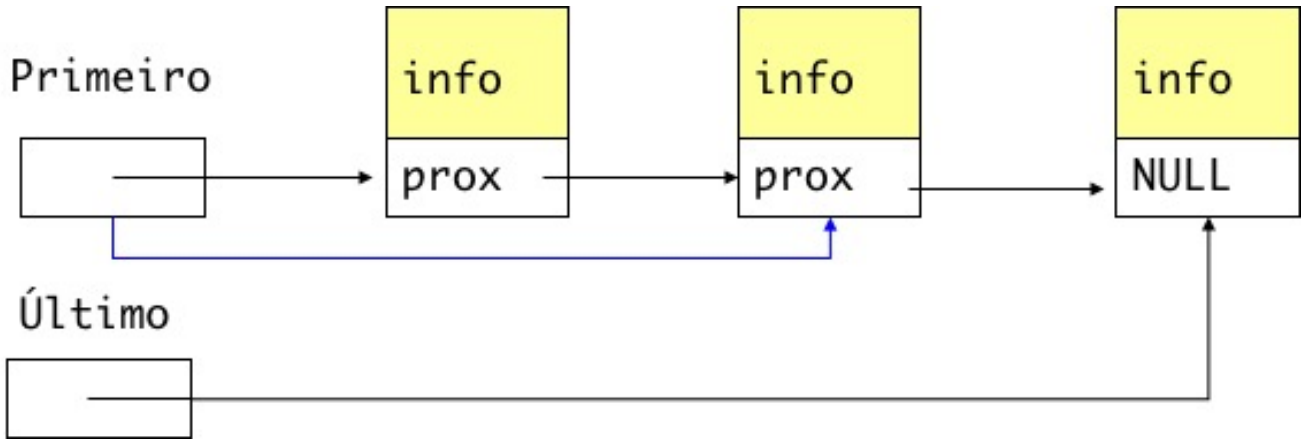
Inserção na Última Posição



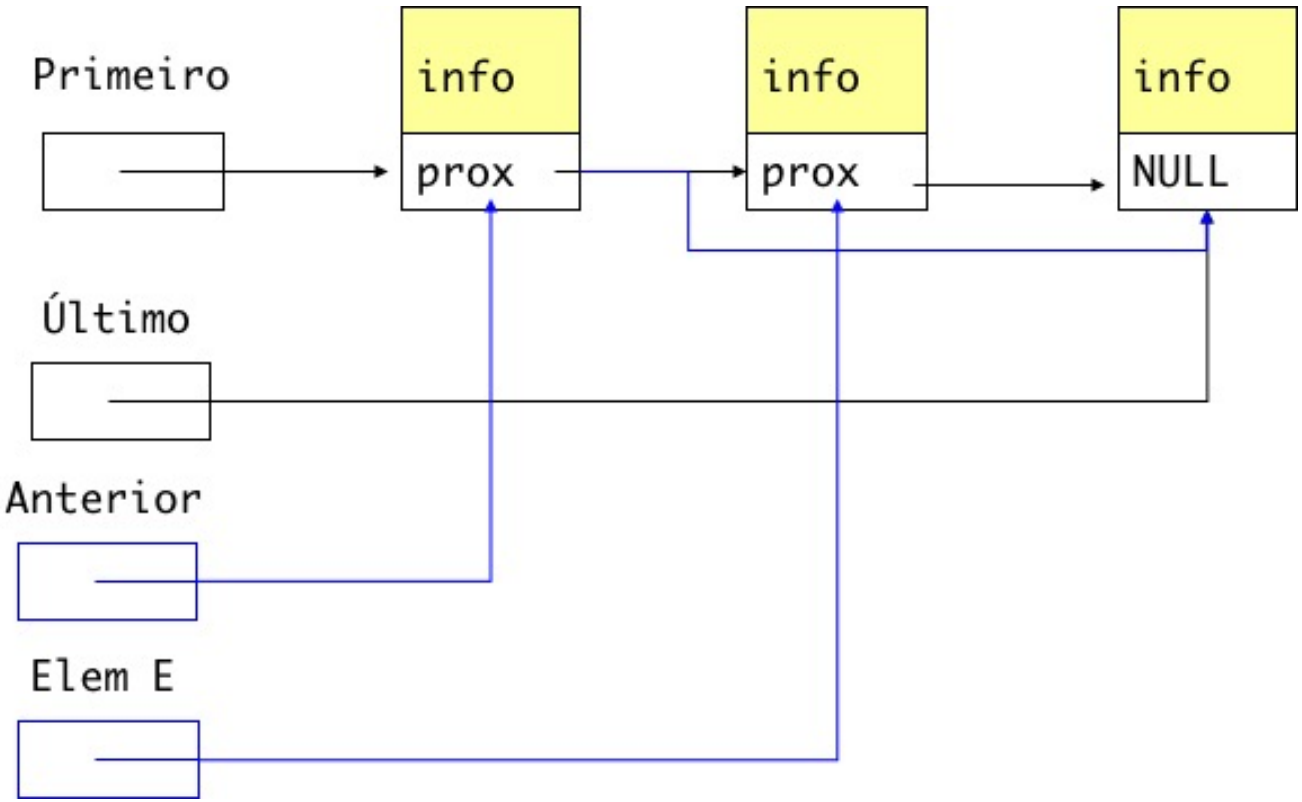
Inserção Após o Elemento E



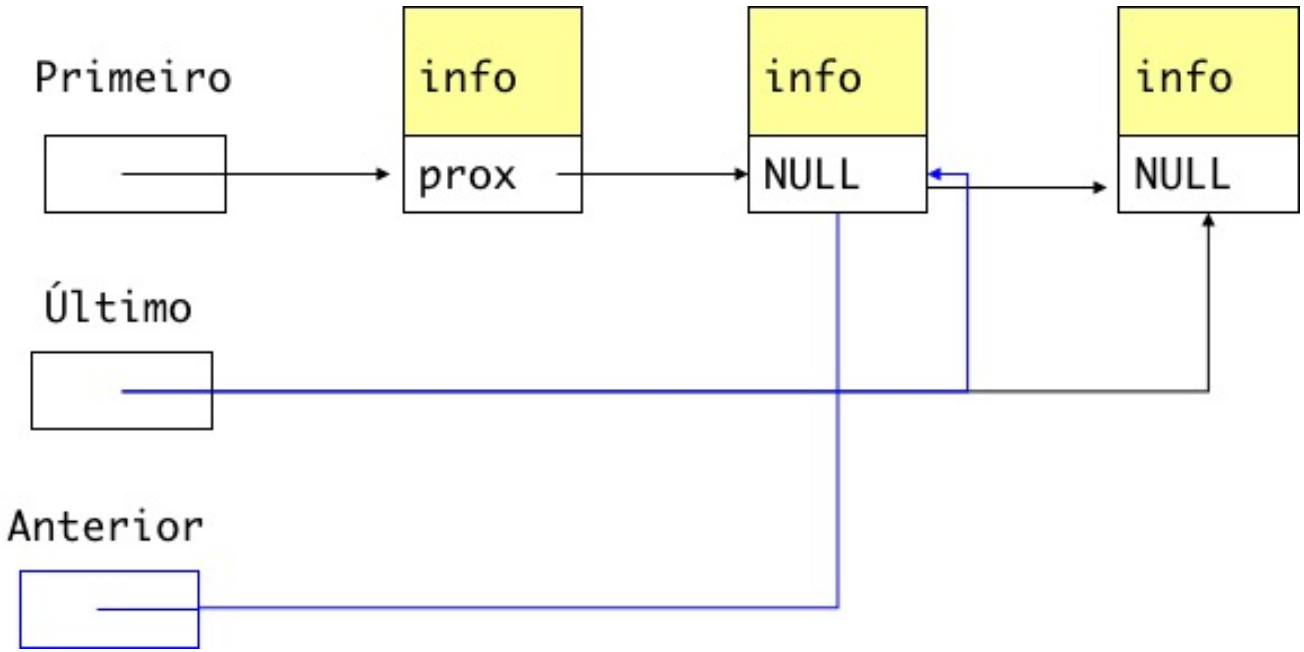
Remoção na Primeira Posição



Remoção do Elemento E



Remoção do Último Elemento



Operações em Listas Encadeadas Sem Cabeça

```
// Inicia as variaveis da lista
void TLista_Inicia(TLista *pLista) {
    pLista->pPrimeiro = NULL;
    pLista->pUltimo = NULL;
}

// Retorna se a lista esta vazia
int TLista_EhVazia(TLista *pLista) {
    return (pLista->pPrimeiro == NULL);
}
```

Operações em Listas Encadeadas Sem Cabeça

```
// Insere um item no final da lista
int TLista_Insere(TLista *pLista, TItem x) {
    if (pLista->pPrimeiro == NULL) {
        pLista->pPrimeiro = (TCelula*) malloc(sizeof(TCelula));
        pLista->pUltimo = pLista->pPrimeiro;
    }
    else {
        pLista->pUltimo->pProx = (TCelula*) malloc(sizeof(TCelu
        pLista->pUltimo = pLista->pUltimo->pProx;
    }
    pLista->pUltimo->Item = *pItem;
    pLista->pUltimo->pProx = NULL;
}
```

Operações em Listas Encadeadas Sem Cabeça

```
// Retira o primeiro item da lista
int TLista_RetiraPrimeiro(TLista *pLista, TItem *pX) {
    if (TLista_EhVazia(pLista))
        return 0;

    TCelula *pAux;
    pAux = pLista->pPrimeiro;
    *pX = pAux->Item;
    pLista->pPrimeiro = pLista->pPrimeiro->pProx;
    free(pAux);
    if (pLista->pPrimeiro == NULL)
        pLista->pUltimo = NULL; // lista vazia
    return 1;
}
```

Operações em Listas Encadeadas Sem Cabeça

```
// Imprime os elementos da lista
void TLista_Imprime(TLista *pLista) {
    TCelula *pAux;
    pAux = pLista->pPrimeiro;
    while (pAux != NULL) {
        printf("%d\n", pAux->item.chave);
        pAux = pAux->pProx; // proxima celula
    }
}
```

Resumo da Complexidade – Implementação por Ponteiros

- Inserção
  - No início: $\Theta(1)$;
  - No fim: $\Theta(1)$;
- Remoção
  - No início: $\Theta(1)$;
  - No meio: $O(n)$;
  - No fim: $\Theta(n)$;
- Pesquisa: $O(n)$.

Dúvidas?