

Estruturas de Dados

Listas Lineares

Professores: Luiz Chaimowicz e Raquel Prates

Listas Lineares

- Maneira de representar um conjunto de elementos
- Itens podem ser acessados, inseridos ou retirados em qualquer posição de uma lista
- Com isso, as listas podem crescer ou diminuir de tamanho durante a execução
- Implementada na maioria das linguagens
 - STL (C++): list
 - java.util: List, ArrayList, LinkedList
 - python: lista = $[x_1, x_2, \dots, x_n]$

Definição formal de 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.

TAD: Lista

■ Duas Implementações:

- ❑ Sequencial (uso de arranjos, alocação estática)
- ❑ Encadeada (uso de apontadores, alocação dinâmica)

■ Operações:

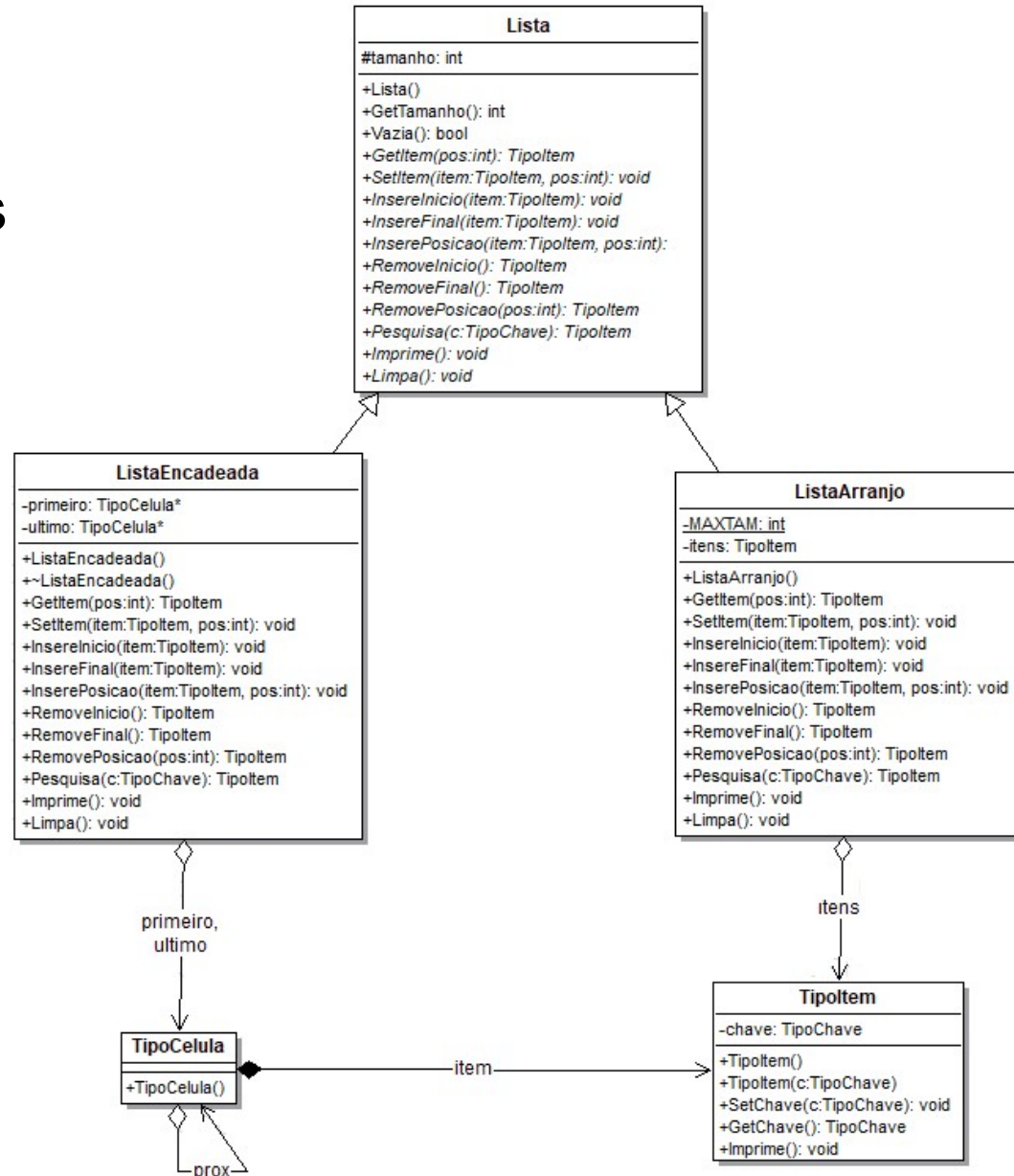
- ❑ Criar uma nova lista (construtor)
- ❑ Métodos de Acesso (Get, Set)
- ❑ Testar se é uma lista *vazia*
- ❑ Inserção: no início, no final, em uma posição p
- ❑ Remoção: do início, do final, de uma posição p
- ❑ Pesquisar por uma chave
- ❑ Imprimir a Lista
- ❑ Limpar a Lista

Disclaimer:

os códigos que serão apresentados devem ser considerados como exemplos. Eles não são, necessariamente, os mais modulares ou eficientes...

TAD Lista

Diagrama de Classes



TAD Lista

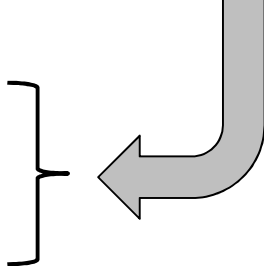
■ Class Lista

- ❑ Classe Abstrata: métodos implementados nas classes herdeiras
- ❑ Trata apenas o atributo *tamanho* (inicialização, acesso, teste Vazia)

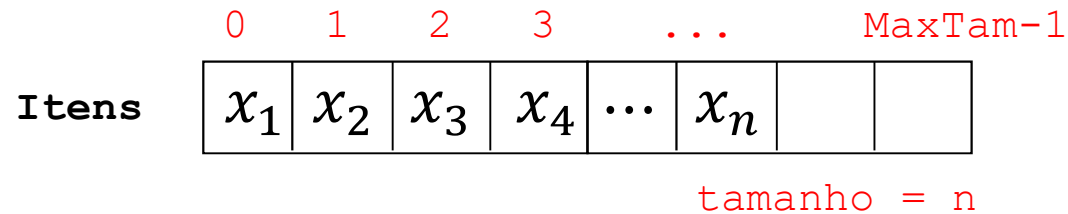
```
class Lista
{
    public:
        Lista() {tamanho = 0;};
        int GetTamanho() {return tamanho;};
        bool Vazia() {return tamanho == 0;};

        virtual TipoItem GetItem(int pos) = 0;
        virtual void SetItem(TipoItem item, int pos) = 0;
        virtual void InsereInicio(TipoItem item) = 0;
        virtual void InsereFinal(TipoItem item) = 0;
        virtual void InserePosicao(TipoItem item, int pos) = 0;
        virtual TipoItem RemoveInicio() = 0;
        virtual TipoItem RemoveFinal() = 0;
        virtual TipoItem RemovePosicao(int pos) = 0;
        virtual TipoItem Pesquisa(TipoChave c) = 0;
        virtual void Imprime() = 0;
        virtual void Limpa() = 0;

    protected:
        int tamanho;
};
```

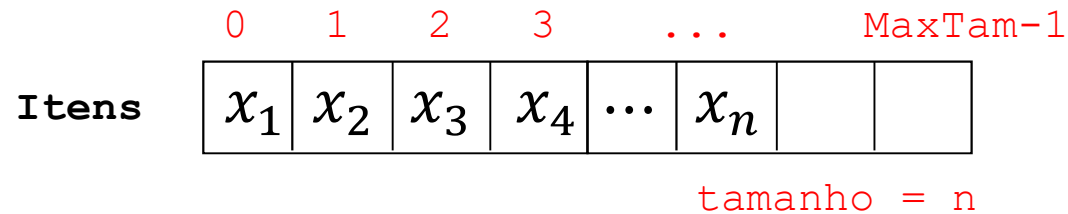


Alocação Sequencial



- Itens da lista são armazenados em um vetor
 - ❑ Alocação Estática, com um tamanho máximo
 - ❑ Permite acesso aleatório a qualquer posição em tempo $O(1)$
 - ❑ Permite percorrer a lista em ambas direções se necessário
 - ❑ Vetor começa em 0: i-ésimo item fica na posição $i-1$.
- Inserção e Remoção
 - ❑ No final tem custo constante: $O(1)$
 - ❑ Em qualquer outra posição causa o deslocamento dos itens à frente: $O(n)$ no pior caso

Class Lista Arranjo



```
class ListaArranjo : public Lista
{
    public:
        ListaArranjo() : Lista() {};
        TipoItem GetItem(int pos);
        void SetItem(TipoItem item, int pos);
        void InsereInicio(TipoItem item);
        void InsereFinal(TipoItem item);
        void InserePosicao(TipoItem item, int pos);
        TipoItem RemoveInicio();
        TipoItem RemoveFinal();
        TipoItem RemovePosicao(int pos);
        TipoItem Pesquisa(TipoChave c);
        void Imprime();
        void Limpa();

    private:
        static const int MAXTAM = 100;
        TipoItem itens[MAXTAM];
};
```


Class TipoItem

- Classe para representar os elementos da lista
 - ❑ Campo **int chave**: identificador único
 - ❑ Poderia ter outros campos, ou possuir um apontador para qualquer outro tipo de objeto
 - ❑ Possui métodos para inicialização, acesso e impressão

```
typedef int TipoChave; // TipoChave é um inteiro
```

```
class TipoItem
{
    public:
        TipoItem();
        TipoItem(TipoChave c);
        void SetChave(TipoChave c);
        TipoChave GetChave();
        void Imprime();

    private:
        TipoChave chave;
        // outros membros
};
```

Class TipoItem

Métodos para Inicialização
acesso e impressão

```
TipoItem::TipoItem()
{
    chave = -1; // indica um item vazio
}

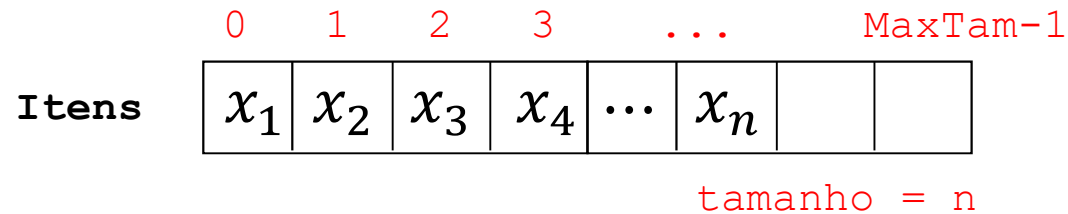
TipoItem::TipoItem(TipoChave c)
{
    chave = c;
}

void TipoItem::SetChave(TipoChave c)
{
    chave = c;
}

TipoChave TipoItem::GetChave()
{
    return chave;
}

void TipoItem::Imprime()
{
    printf("%d ", chave);
}
```

Class Lista Arranjo



```
class ListaArranjo : public Lista
{
    public:
        ListaArranjo() : Lista() {};
        TipoItem GetItem(int pos);
        void SetItem(TipoItem item, int pos);
        void InsereInicio(TipoItem item);
        void InsereFinal(TipoItem item);
        void InserePosicao(TipoItem item, int pos);
        TipoItem RemoveInicio();
        TipoItem RemoveFinal();
        TipoItem RemovePosicao(int pos);
        TipoItem Pesquisa(TipoChave c);
        void Imprime();
        void Limpa();

    private:
        static const int MAXTAM = 100;
        TipoItem itens[MAXTAM];
};
```

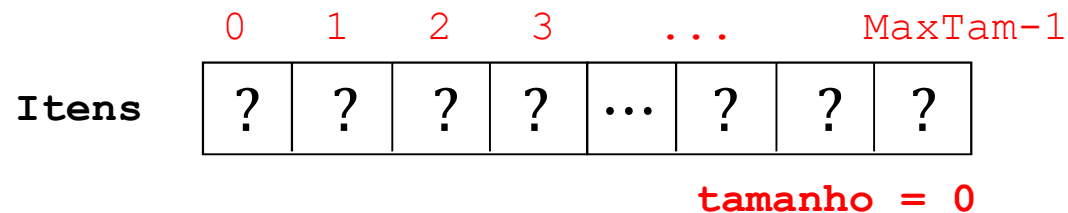
Class Lista Arranjo - Construtor

■ Construtor

- Apenas chama o construtor da classe pai, que inicializa o atributo **tamanho** com o valor 0.
- O conteúdo dos elementos do vetor **itens** não importa...

```
class ListaArranjo : public Lista
{
    public:
        ListaArranjo() : Lista() {};

    ...
};
```



ListaArranjo L;

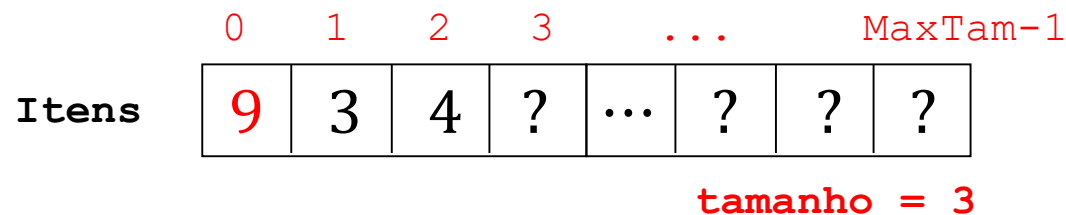
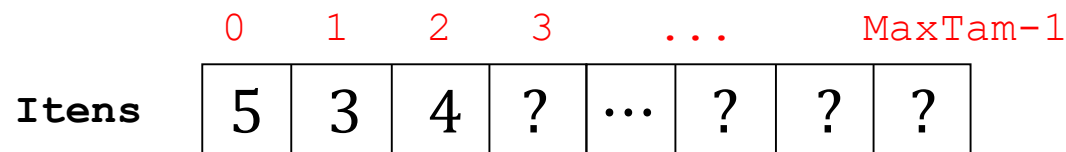
Class Lista Arranjo – Get & Set

```
TipoItem ListaArranjo::GetItem(int pos){  
    if ( (pos > tamanho) || (pos <= 0) )  
        throw "ERRO: Posicao Invalida!";  
  
    return itens[pos-1];  
}
```

$O(1)$

```
void ListaArranjo::SetItem(TipoItem item, int pos){  
    if ( (pos > tamanho) || (pos <= 0) )  
        throw "ERRO: Posicao Invalida!";  
  
    itens[pos-1] = item;  
}
```

$O(1)$



```
ListaArranjo L;  
TipoItem x;  
...  
x.SetChave(9)  
L.SetItem(x,1);  
x = L.GetItem(3)  
x.Imprime();
```

Posição Lógica x Posição Física
1º elemento está na posição 0

4

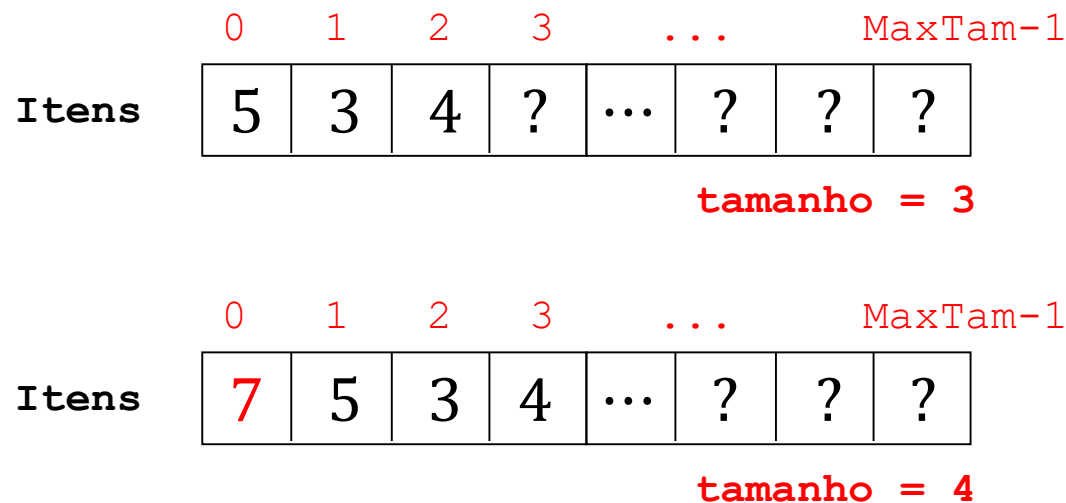
Class Lista Arranjo - Inserção

- Inserção pode ser feita no início, no final, ou em uma posição p qualquer
- A inserção que não seja feita no final causa o deslocamento de todos os itens do vetor
- Deve-se testar se há espaço para a inserção do novo item (alocação estática)
 - Gera uma exceção que pode ser tratada por quem chamou o método.

Class Lista Arranjo - Inserção

```
void ListaArranjo::InsereInicio(TipoItem item) {  
    int i;  
    if (tamanho == MAXTAM)  
        throw "ERRO: Lista cheia!";  
  
    tamanho++;  
    for(i=tamanho;i>0;i--)  
        itens[i] = itens[i-1];  
  
    itens[0] = item;  
};
```

$O(n)$

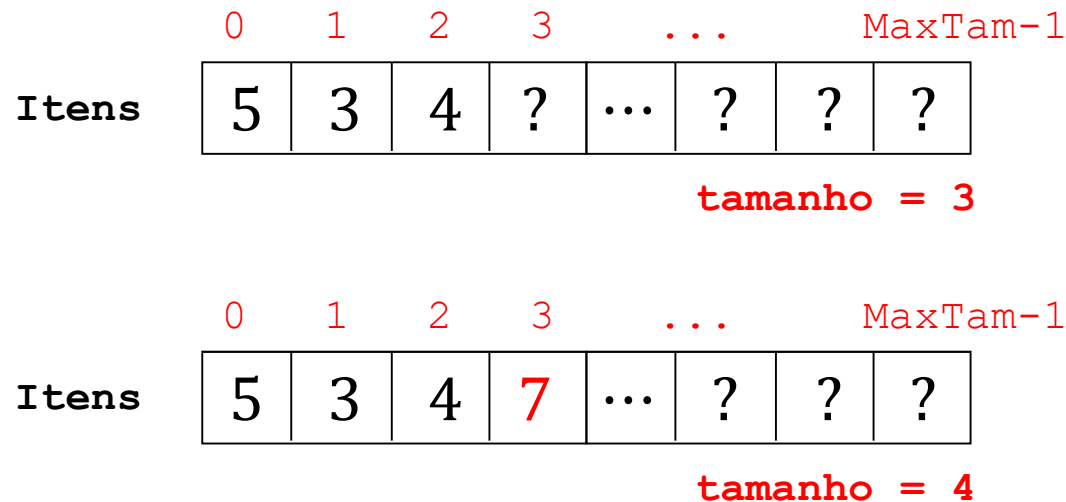


```
ListaArranjo L;  
TipoItem x;  
...  
x.SetChave(7)  
L.InsereInicio(x)
```

Class Lista Arranjo - Inserção

```
void ListaArranjo::InsereFinal(TipoItem item) {  
    if (tamanho == MAXTAM)  
        throw "ERRO: Lista cheia!";  
  
    itens[tamanho] = item;  
    tamanho++;  
};
```

$O(1)$



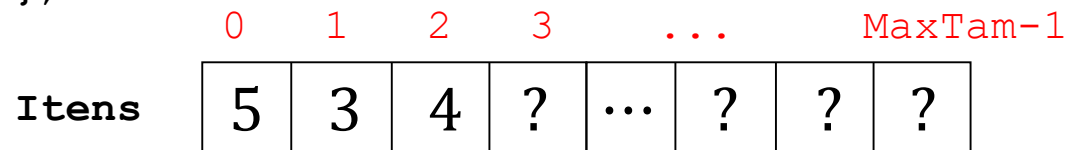
```
ListaArranjo L;  
TipoItem x;  
...  
x.SetChave(7)  
L.InsereFinal(x)
```


Class Lista Arranjo - Inserção

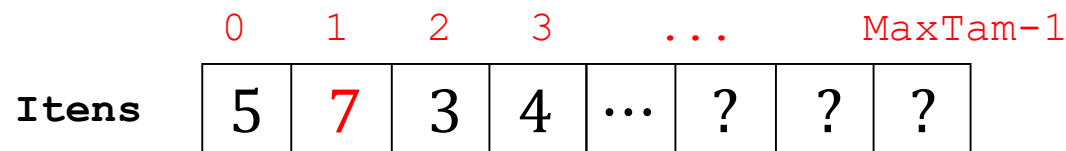
```
void ListaArranjo::InserePosicao(TipoItem item, int pos) {  
    int i;  
    if (tamanho == MAXTAM)  
        throw "ERRO: Lista cheia!";  
    if ( (pos > tamanho) || (pos <= 0) )  
        throw "ERRO: Posicao Invalida!";  
  
    pos--; // posição no vetor = pos-1 (vetor começa em 0)  
    tamanho++;  
    for(i=tamanho;i>pos;i--)  
        itens[i] = itens[i-1];  
  
    itens[pos] = item;  
};
```

Melhor Caso $O(1)$

Pior Caso $O(n)$



tamanho = 3



tamanho = 4

```
ListaArranjo L;  
TipoItem x;  
...  
x.SetChave(7)  
L.InserePosicao(x, 2)
```

Posição Lógica x Posição Física
2º elemento está na posição 1

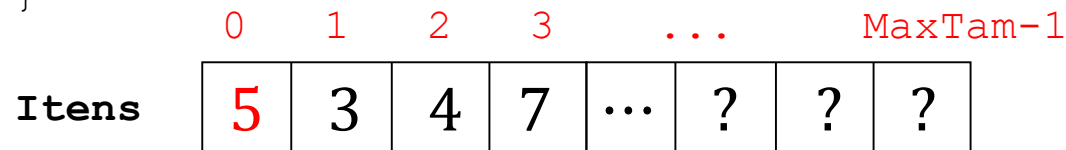
Class Lista Arranjo - Remoção

- Da mesma forma, a remoção pode ser feita no início, no final, ou em uma posição p qualquer
- A remoção que não seja feita no final causa o deslocamento de todos os itens do vetor
- Deve-se verificar se há elementos e se a posição de remoção é válida
 - Gera uma exceção que pode ser tratada por quem chamou o método.
- O elemento removido é retornado pelo método

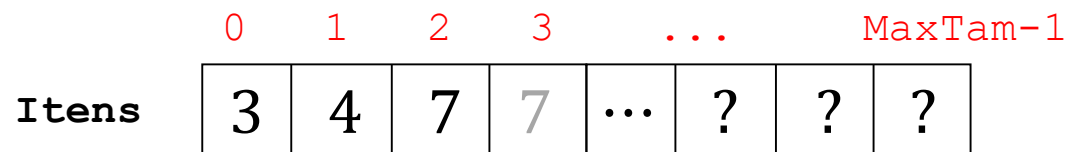
Class Lista Arranjo - Remoção

```
TipoItem ListaArranjo::RemoveInicio() {  
    int i;  
    TipoItem aux;  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    aux = itens[0];  
    for(i=0;i<tamanho;i++)  
        itens[i] = itens[i+1];  
  
    tamanho--;  
    return aux;  
}
```

$O(n)$



tamanho = 4



tamanho = 3

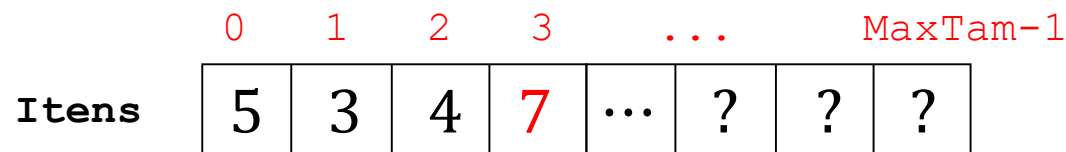
```
ListaArranjo L;  
TipoItem x;  
...  
x = L.RemoveInicio();  
x.Imprime();
```

5

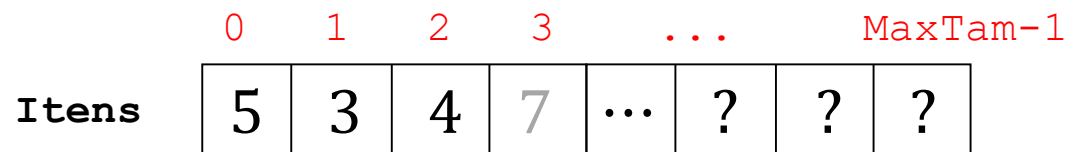
Class Lista Arranjo - Remoção

```
TipoItem ListaArranjo::RemoveFinal() {  
    TipoItem aux;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    tamanho--;  
    aux = itens[tamanho];  
    return aux;  
}
```

$O(1)$



tamanho = 4



tamanho = 3

```
ListaArranjo L;  
TipoItem x;  
...  
x = L.RemoveFinal();  
x.Imprime();
```

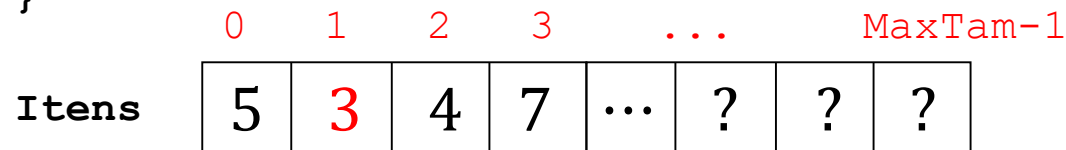
7

Class Lista Arranjo - Remoção

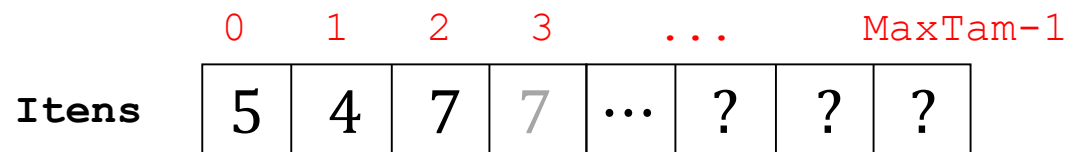
```
TipoItem ListaArranjo::RemovePosicao(int pos) {  
    int i; TipoItem aux;  
  
    if ( (pos > tamanho) || (pos <= 0) )  
        throw "ERRO: Posicao Invalida!";  
  
    pos--; // posição no vetor = pos-1 (vetor começa em 0)  
    aux = itens[pos];  
    for(i=pos; i<tamanho; i++)  
        itens[i] = itens[i+1];  
    tamanho--;  
    return aux;  
}
```

Melhor Caso $O(1)$

Pior Caso $O(n)$



tamanho = 4



tamanho = 3

```
ListaArranjo L;  
TipoItem x;  
...  
x = L.RemovePosicao(2);  
x.Imprime();
```

Posição Lógica x Posição Física
2º elemento está na posição 1

3

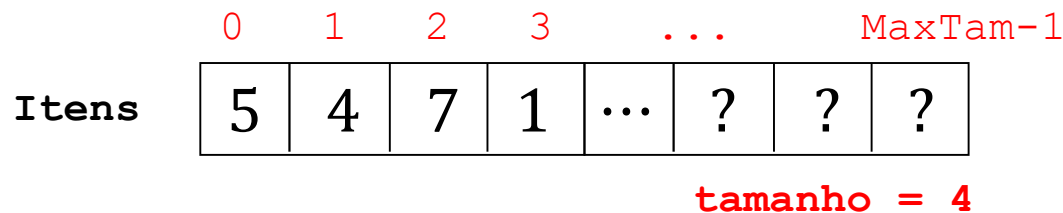
Class Lista Arranjo - Pesquisa

- Pesquisa por um item com uma determinada chave
 - Retorna o item encontrado ou um *flag* (-1)

```
TipoItem ListaArranjo::Pesquisa(TipoChave c) {  
    int i; TipoItem aux;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    aux.SetChave(-1); // indica pesquisa sem sucesso  
    for(i=0; i<tamanho; i++)  
        if(itens[i].GetChave() == c) {  
            aux = itens[i];  
            break;  
        }  
  
    return aux;  
};
```

Melhor
Caso $O(1)$

Pior
Caso $O(n)$



```
ListaArranjo L;  
TipoItem x;  
...  
x = L.Pesquisa(7);  
x.Imprime();
```

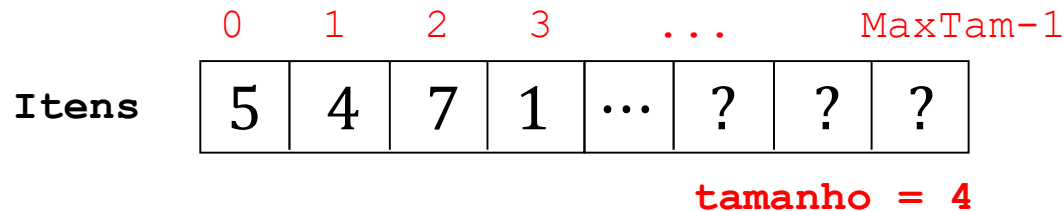
7

Class Lista Arranjo - Imprime

■ Imprime todos os elementos

```
void ListaArranjo::Imprime() {  
    int i;  
  
    for(i=0; i<tamanho; i++)  
        itens[i].Imprime();  
  
    printf("\n");  
};
```

$O(n)$



```
ListaArranjo L;  
TipoItem x;  
...  
L.Imprime();
```

5 4 7 1

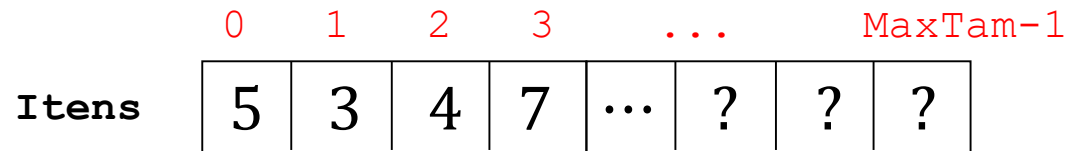
Class Lista Arranjo - Limpa

■ “Limpa” a Lista

- Basta fazer o tamanho = 0

```
void ListaArranjo::Limpa() {  
    tamanho = 0;  
};
```

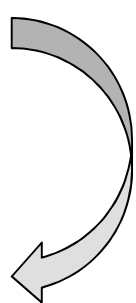
$O(1)$



tamanho = 4



tamanho = 0



```
ListaArranjo L;  
TipoItem x;  
...  
L.Limpa();
```


Alocação Sequencial

- Vantagens:

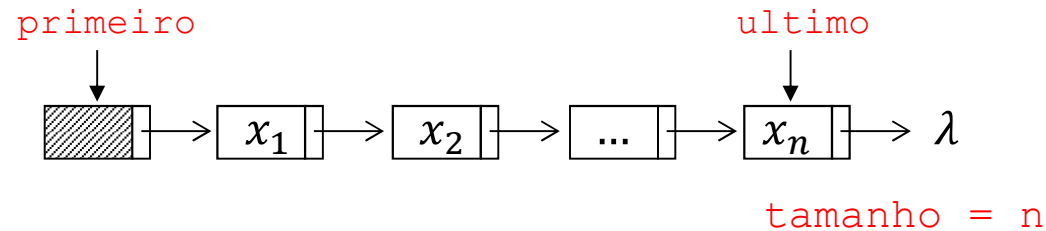
- ❑ **Não necessita de apontadores explícitos** para organizar os itens na lista
 - Economia de memória
 - Implementação mais simples
- ❑ Permite **acesso direto aos itens** em uma determinada posição
 - Métodos *Get* e *Set* são $O(1)$

Alocação Sequencial

- Desvantagens:

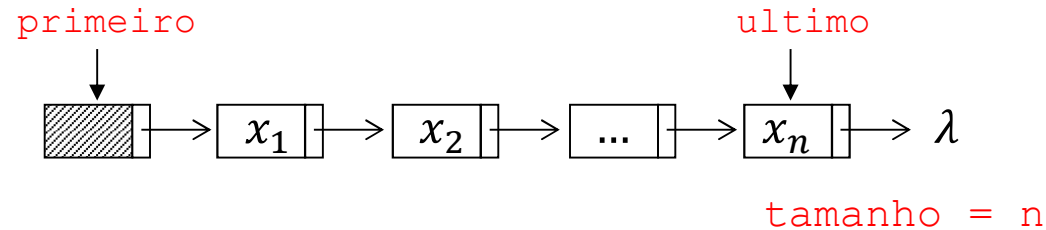
- Custo para inserir ou retirar itens da lista, que pode causar um deslocamento de todos os itens
 - $O(n)$ no pior caso
- O tamanho máximo da lista é **fixo** e definido em tempo de compilação. Pouco prático em aplicações onde o tamanho não pode ser previsto...
 - Pode causar *overflow* se número de itens for maior que o tamanho previsto
 - Desperdício de memória se o número de itens for muito menor que o tamanho previsto

Alocação Encadeada



- Itens da lista são armazenados em posições não contíguas da memória
 - ❑ Utilização de **células** que são encadeadas usando **apontadores**
 - ❑ Alocação dinâmica, permitindo crescimento e redução de tamanho
 - ❑ **Não** permite acesso direto a qualquer item
- Inserção e Remoção
 - ❑ Não requer deslocamento de itens

Class Lista Encadeada



```
class ListaEncadeada : public Lista {
public:
    ListaEncadeada();
    ~ListaEncadeada();

    TipoItem GetItem(int pos);
    void SetItem(TipoItem item, int pos);
    void InsereInicio(TipoItem item);
    void InsereFinal(TipoItem item);
    void InserePosicao(TipoItem item, int pos);
    TipoItem RemoveInicio();
    TipoItem RemoveFinal();
    TipoItem RemovePosicao(int pos);
    TipoItem Pesquisa(TipoChave c);
    void Imprime();
    void Limpa();

private:
    TipoCelula* primeiro;
    TipoCelula* ultimo;
    TipoCelula* Posiciona(int pos, bool antes);
};
```

Class TipoCélula

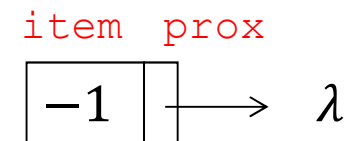
- Classe para representar as células da lista
 - ❑ Campo **TipoItem item**: armazena o item
 - ❑ Campo **TipoCelula *prox**: apontador para a próxima célula
 - ❑ Possui método para inicialização (constructor)
 - ❑ Permite o acesso de atributos privados pela classe ListaEncadeada
 - *Friend class*

```
class TipoCelula
{
    public:
        TipoCelula();

    private:
        TipoItem item;
        TipoCelula *prox;

    friend class ListaEncadeada;
};
```

```
TipoCelula::TipoCelula()
{
    item.SetChave(-1);
    prox = NULL;
}
```



Class Lista Encadeada - Construtor

■ Construtor

- ❑ Chama o construtor da classe pai, que inicializa o atributo *tamanho* com o valor 0, e inicializa os apontadores *primeiro* e *ultimo*.
- ❑ Uso de uma **célula cabeça**
 - Facilita as operações de inserção e remoção no início da lista
 - Primeiro elemento da lista vai estar na posição **primeiro**->**prox**

```
ListaEncadeada::ListaEncadeada() : Lista() {  
    primeiro = new TipoCelula();  
    ultimo = primeiro;  
}
```

primeiro



λ



ultimo

tamanho = 0

ListaEncadeada L;

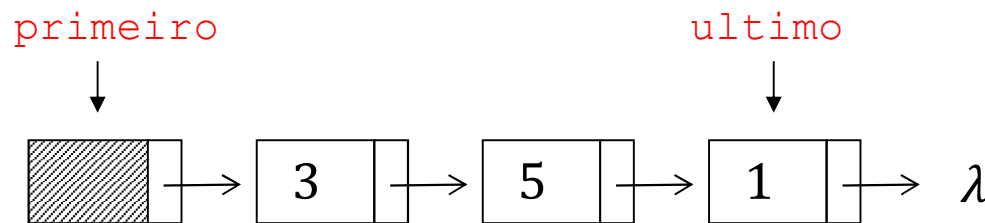
...

Class Lista Encadeada - Destrutor

■ Destrutor

- ❑ Como utilizamos a alocação dinâmica, é importante implementar um destrutor para desalocar a memória adequadamente
- ❑ Chama o método *Limpa*, que remove todas as células da lista e depois remove a célula cabeça

```
ListaEncadeada::~~ListaEncadeada()  
{  
    Limpa();  
    delete primeiro;  
}
```



```
ListaEncadeada *L;  
...  
delete L;
```

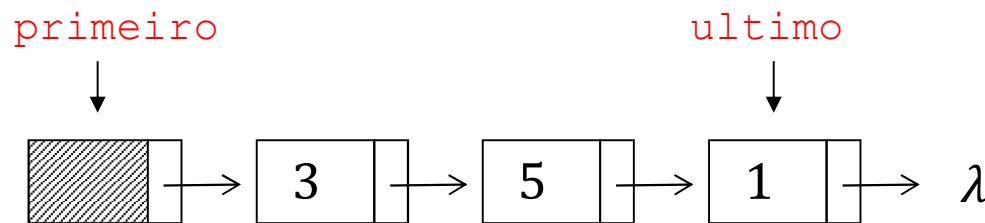
tamanho = 3

Class Lista Encadeada - Destrutor

■ Destrutor

- ❑ Como utilizamos a alocação dinâmica, é importante implementar um destrutor para desalocar a memória adequadamente
- ❑ Chama o método *Limpa*, que remove todas as células da lista e depois remove a célula cabeça

```
ListaEncadeada::~~ListaEncadeada()  
{  
    Limpa();  
    delete primeiro;  
}
```



```
ListaEncadeada *L;  
...  
delete L;
```

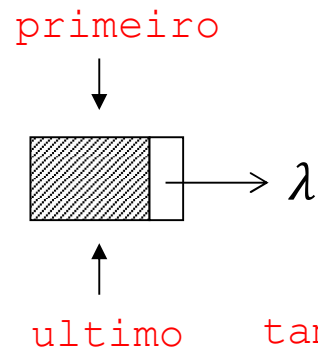
tamanho = 3

Class Lista Encadeada - Destrutor

■ Destrutor

- ❑ Como utilizamos a alocação dinâmica, é importante implementar um destrutor para desalocar a memória adequadamente
- ❑ Chama o método *Limpa*, que remove todas as células da lista e depois remove a célula cabeça

```
ListaEncadeada::~~ListaEncadeada()  
{  
    Limpa();  
    delete primeiro;  
}
```



```
ListaEncadeada *L;  
...  
delete L;
```

Class Lista Encadeada - Destrutor

■ Destrutor

- ❑ Como utilizamos a alocação dinâmica, é importante implementar um destrutor para desalocar a memória adequadamente
- ❑ Chama o método *Limpa*, que remove todas as células da lista e depois remove a célula cabeça

```
ListaEncadeada::~~ListaEncadeada()  
{  
    Limpa();  
    delete primeiro;  
}
```

$O(n)$

primeiro
↓
 λ
↑
ultimo

```
ListaEncadeada *L;  
...  
delete L;
```

Class Lista Encadeada - Posiciona

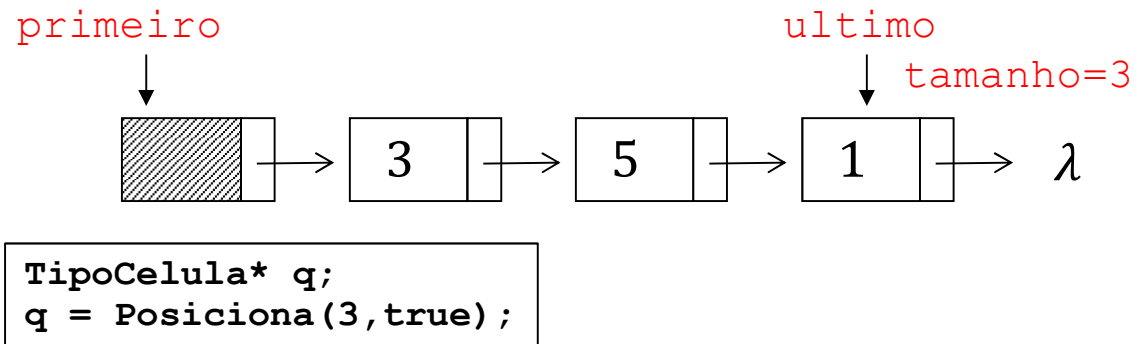
- **Função auxiliar** para posicionar um apontador em em uma determinada posição (célula) da lista
 - ▣ Opção de parar na célula anterior (útil para inserção e remoção)

```
TipoCelula* ListaEncadeada::Posiciona(int pos, bool antes=false){
    TipoCelula *p; int i;

    if ( (pos > tamanho) || (pos <= 0) )
        throw "ERRO: Posicao Invalida!";

    // Posiciona na célula anterior a desejada
    p = primeiro;
    for(i=1; i<pos; i++){
        p = p->prox;
    }
    // vai para a próxima
    // se antes for false
    if(!antes)
        p = p->prox;

    return p;
}
```



Class Lista Encadeada - Posiciona

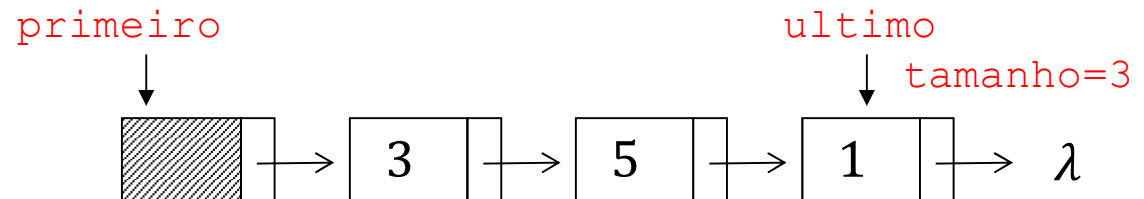
- **Função auxiliar** para posicionar um apontador em em uma determinada posição (célula) da lista
 - ▣ Opção de parar na célula anterior (útil para inserção e remoção)

```
TipoCelula* ListaEncadeada::Posiciona(int pos, bool antes=false){
    TipoCelula *p; int i;

    if ( (pos > tamanho) || (pos <= 0) )
        throw "ERRO: Posicao Invalida!";

    // Posiciona na célula anterior a desejada
    p = primeiro;
    for(i=1; i<pos; i++){
        p = p->prox;
    }
    // vai para a próxima
    // se antes for false
    if(!antes)
        p = p->prox;

    return p;
}
```



```
TipoCelula* q;
q = Posiciona(3,true);
```

Class Lista Encadeada - Posiciona

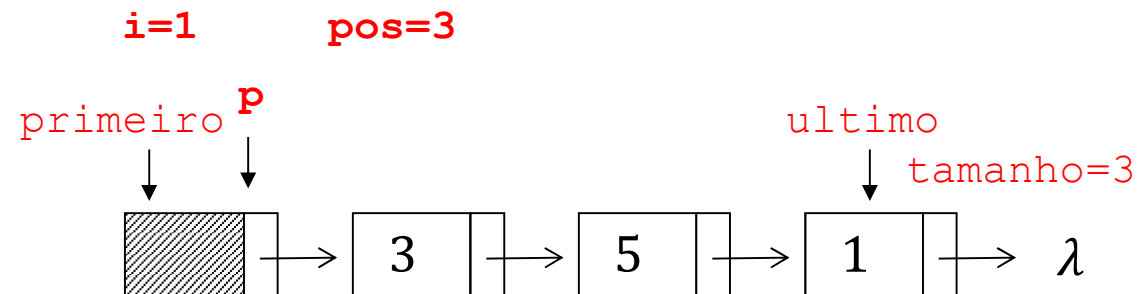
- **Função auxiliar** para posicionar um apontador em em uma determinada posição (célula) da lista
 - ▣ Opção de parar na célula anterior (útil para inserção e remoção)

```
TipoCelula* ListaEncadeada::Posiciona(int pos, bool antes=false){
    TipoCelula *p; int i;

    if ( (pos > tamanho) || (pos <= 0) )
        throw "ERRO: Posicao Invalida!";

    // Posiciona na célula anterior a desejada
    p = primeiro;
    for(i=1; i<pos; i++){
        p = p->prox;
    }
    // vai para a próxima
    // se antes for false
    if(!antes)
        p = p->prox;

    return p;
}
```



```
TipoCelula* q;
q = Posiciona(3,true);
```

Class Lista Encadeada - Posiciona

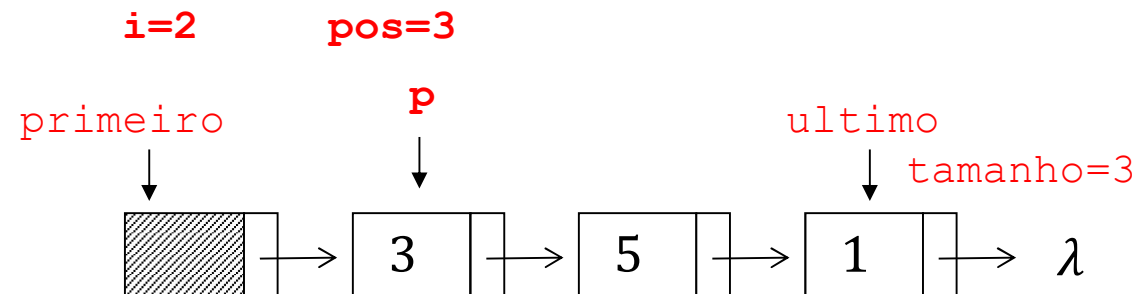
- **Função auxiliar** para posicionar um apontador em em uma determinada posição (célula) da lista
 - ▣ Opção de parar na célula anterior (útil para inserção e remoção)

```
TipoCelula* ListaEncadeada::Posiciona(int pos, bool antes=false){
    TipoCelula *p; int i;

    if ( (pos > tamanho) || (pos <= 0) )
        throw "ERRO: Posicao Invalida!";

    // Posiciona na célula anterior a desejada
    p = primeiro;
    for(i=1; i<pos; i++){
        p = p->prox;
    }
    // vai para a próxima
    // se antes for false
    if(!antes)
        p = p->prox;

    return p;
}
```



```
TipoCelula* q;
q = Posiciona(3,true);
```

Class Lista Encadeada - Posiciona

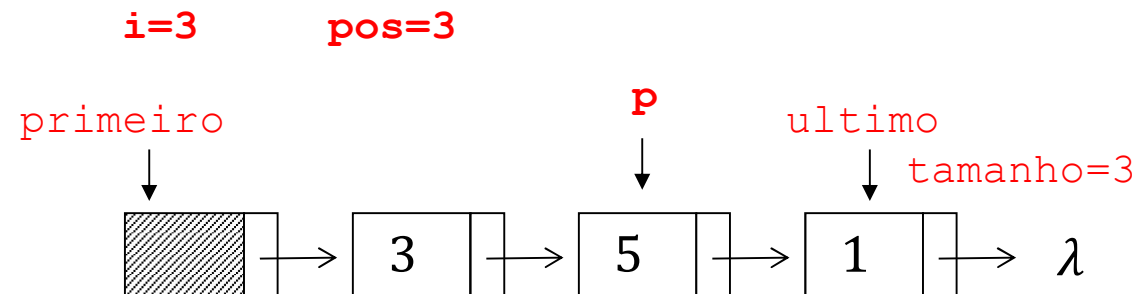
- **Função auxiliar** para posicionar um apontador em em uma determinada posição (célula) da lista
 - ▣ Opção de parar na célula anterior (útil para inserção e remoção)

```
TipoCelula* ListaEncadeada::Posiciona(int pos, bool antes=false){
    TipoCelula *p; int i;

    if ( (pos > tamanho) || (pos <= 0) )
        throw "ERRO: Posicao Invalida!";

    // Posiciona na célula anterior a desejada
    p = primeiro;
    for(i=1; i<pos; i++){
        p = p->prox;
    }
    // vai para a próxima
    // se antes for false
    if(!antes)
        p = p->prox;

    return p;
}
```



```
TipoCelula* q;
q = Posiciona(3,true);
```

Class Lista Encadeada - Posiciona

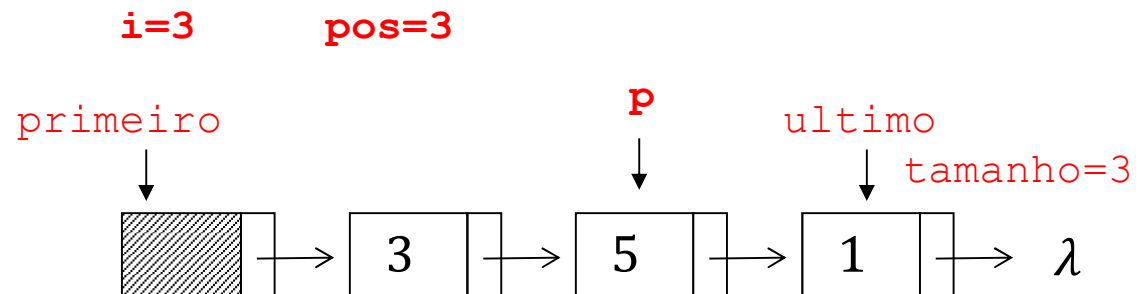
- **Função auxiliar** para posicionar um apontador em em uma determinada posição (célula) da lista
 - ▣ Opção de parar na célula anterior (útil para inserção e remoção)

```
TipoCelula* ListaEncadeada::Posiciona(int pos, bool antes=false){
    TipoCelula *p; int i;

    if ( (pos > tamanho) || (pos <= 0) )
        throw "ERRO: Posicao Invalida!";

    // Posiciona na célula anterior a desejada
    p = primeiro;
    for(i=1; i<pos; i++){
        p = p->prox;
    }
    // vai para a próxima
    // se antes for false
    if(!antes)
        p = p->prox;

    return p;
}
```



```
TipoCelula* q;
q = Posiciona(3,true);
```


Class Lista Encadeada - Posiciona

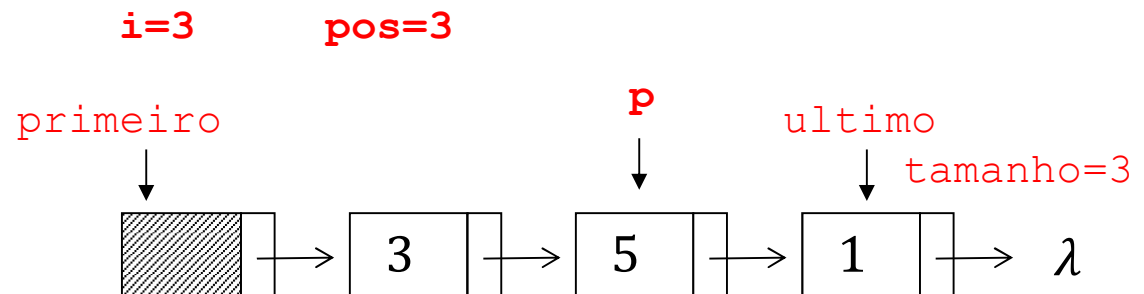
- **Função auxiliar** para posicionar um apontador em em uma determinada posição (célula) da lista
 - ▣ Opção de parar na célula anterior (útil para inserção e remoção)

```
TipoCelula* ListaEncadeada::Posiciona(int pos, bool antes=false){
    TipoCelula *p; int i;

    if ( (pos > tamanho) || (pos <= 0) )
        throw "ERRO: Posicao Invalida!";

    // Posiciona na célula anterior a desejada
    p = primeiro;
    for(i=1; i<pos; i++){
        p = p->prox;
    }
    // vai para a próxima
    // se antes for false
    if(!antes)
        p = p->prox;

    return p;
}
```



```
TipoCelula* q;
q = Posiciona(3,true);
```

Class Lista Encadeada - Posiciona

- **Função auxiliar** para posicionar um apontador em em uma determinada posição (célula) da lista
 - ▣ Opção de parar na célula anterior (útil para inserção e remoção)

```
TipoCelula* ListaEncadeada::Posiciona(int pos, bool antes=false){
    TipoCelula *p; int i;

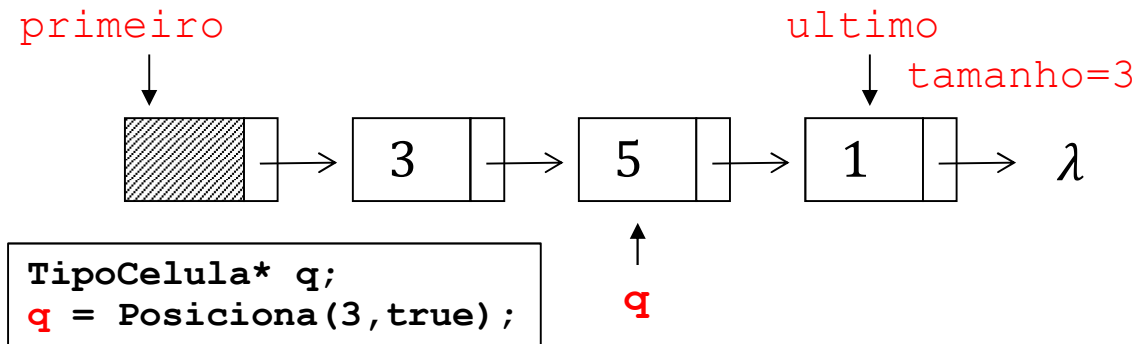
    if ( (pos > tamanho) || (pos <= 0) )
        throw "ERRO: Posicao Invalida!";

    // Posiciona na célula anterior a desejada
    p = primeiro;
    for(i=1; i<pos; i++){
        p = p->prox;
    }
    // vai para a próxima
    // se antes for false
    if(!antes)
        p = p->prox;

    return p;
}
```

Melhor Caso $O(1)$

Pior Caso $O(n)$



Class Lista Encadeada – Get & Set

```
TipoItem ListaEncadeada::GetItem(int pos){
    TipoCelula *p;

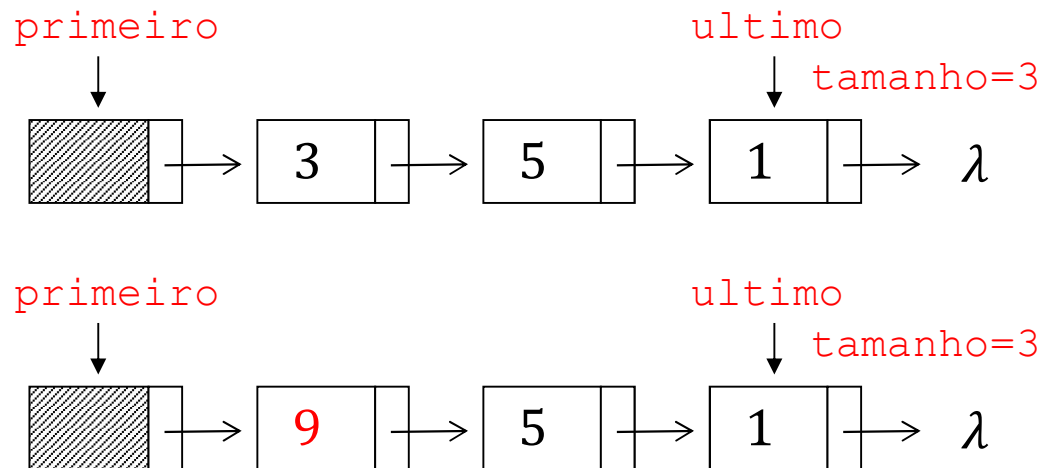
    p = Posiciona(pos);
    return p->item;
}
```

Melhor Caso $O(1)$

```
void ListaEncadeada::SetItem(TipoItem item, int pos){
    TipoCelula *p;

    p = Posiciona(pos);
    p->item = item;
}
```

Pior Caso $O(n)$



```
ListaEncadeada L;
TipoItem x;
...
x.SetChave(9)
L.SetItem(x, 1);
x = L.GetItem(3)
x.Imprime();
```

1

Class Lista Encadeada - Inserção

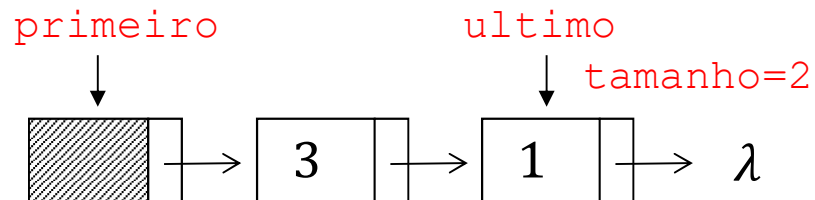
- Inserção pode ser feita no início, no final, ou em uma posição p qualquer
- Deve-se posicionar um apontador auxiliar **antes** da posição a ser inserida
- Uma nova célula é alocada dinamicamente e ligada a lista através da manipulação de apontadores.
- Se estiver inserindo na última posição, deve-se atualizar o apontador *ultimo*

Class Lista Encadeada - Inserção

```
void ListaEncadeada::InsereInicio(TipoItem item)
{
    TipoCelula *nova;

    nova = new TipoCelula();
    nova->item = item;
    nova->prox = primeiro->prox;
    primeiro->prox = nova;
    tamanho++;

    if(nova->prox == NULL)
        ultimo = nova;
};
```



ListaEncadeada L;
TipoItem x;

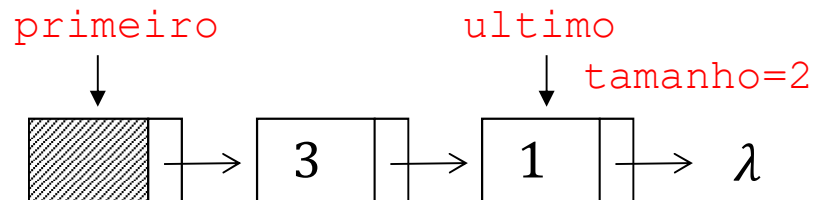
x.SetChave(7)
L.InsereInicio(x)

Class Lista Encadeada - Inserção

```
void ListaEncadeada::InsereInicio(TipoItem item)
{
    TipoCelula *nova;

    nova = new TipoCelula();
    nova->item = item;
    nova->prox = primeiro->prox;
    primeiro->prox = nova;
    tamanho++;

    if(nova->prox == NULL)
        ultimo = nova;
};
```



ListaEncadeada L;
TipoItem x;

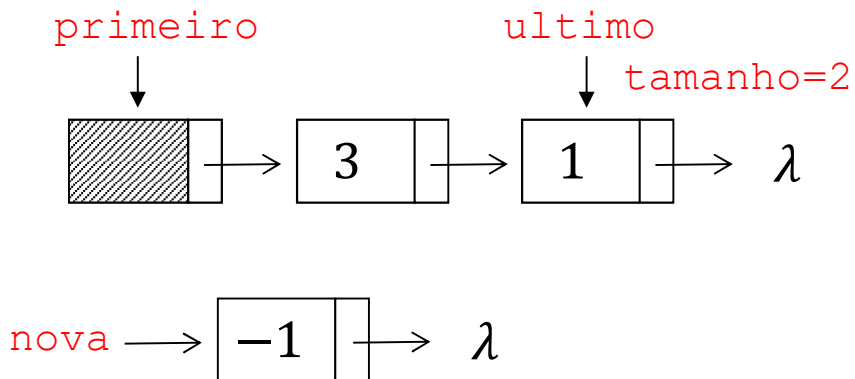
x.SetChave(7)
L.InsereInicio(x)

Class Lista Encadeada - Inserção

```
void ListaEncadeada::InsereInicio(TipoItem item)
{
    TipoCelula *nova;

    nova = new TipoCelula();
    nova->item = item;
    nova->prox = primeiro->prox;
    primeiro->prox = nova;
    tamanho++;

    if(nova->prox == NULL)
        ultimo = nova;
};
```



ListaEncadeada L;
TipoItem x;

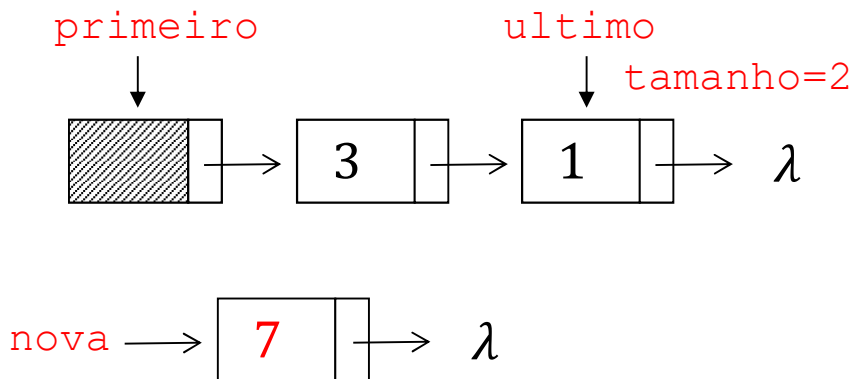
x.SetChave(7)
L.InsereInicio(x)

Class Lista Encadeada - Inserção

```
void ListaEncadeada::InsereInicio(TipoItem item)
{
    TipoCelula *nova;

    nova = new TipoCelula();
    nova->item = item;
    nova->prox = primeiro->prox;
    primeiro->prox = nova;
    tamanho++;

    if(nova->prox == NULL)
        ultimo = nova;
};
```



ListaEncadeada L;
TipoItem x;

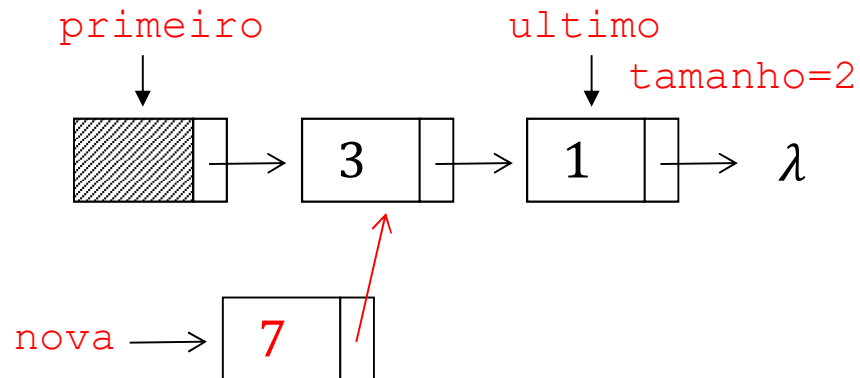
x.SetChave(7)
L.InsereInicio(x)

Class Lista Encadeada - Inserção

```
void ListaEncadeada::InsereInicio(TipoItem item)
{
    TipoCelula *nova;

    nova = new TipoCelula();
    nova->item = item;
    nova->prox = primeiro->prox;
    primeiro->prox = nova;
    tamanho++;

    if(nova->prox == NULL)
        ultimo = nova;
};
```



ListaEncadeada L;
TipoItem x;

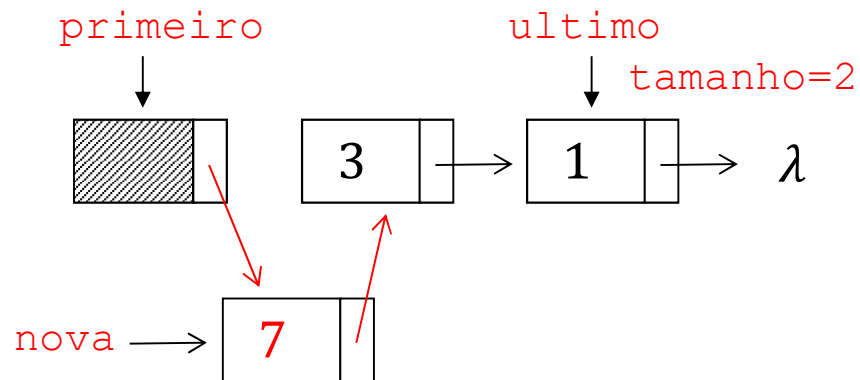
x.SetChave(7)
L.InsereInicio(x)

Class Lista Encadeada - Inserção

```
void ListaEncadeada::InsereInicio(TipoItem item)
{
    TipoCelula *nova;

    nova = new TipoCelula();
    nova->item = item;
    nova->prox = primeiro->prox;
    primeiro->prox = nova;
    tamanho++;

    if(nova->prox == NULL)
        ultimo = nova;
};
```



```
ListaEncadeada L;
TipoItem x;
```

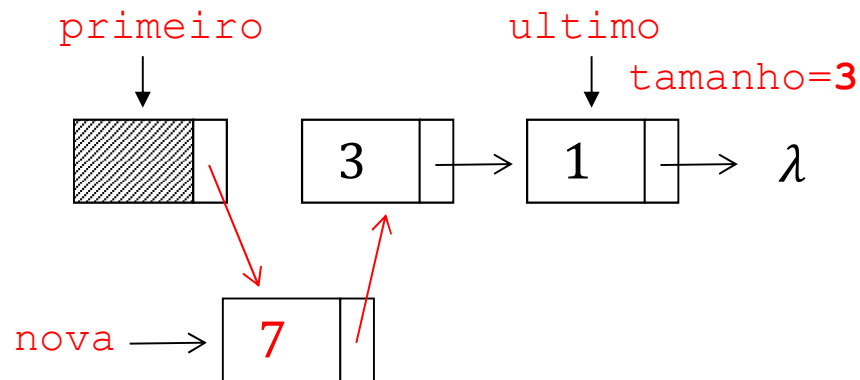
```
x.SetChave(7)
L.InsereInicio(x)
```

Class Lista Encadeada - Inserção

```
void ListaEncadeada::InsereInicio(TipoItem item)
{
    TipoCelula *nova;

    nova = new TipoCelula();
    nova->item = item;
    nova->prox = primeiro->prox;
    primeiro->prox = nova;
    tamanho++;

    if (nova->prox == NULL)
        ultimo = nova;
};
```



```
ListaEncadeada L;  
TipoItem x;
```

```
x.SetChave(7)  
L.InsereInicio(x)
```

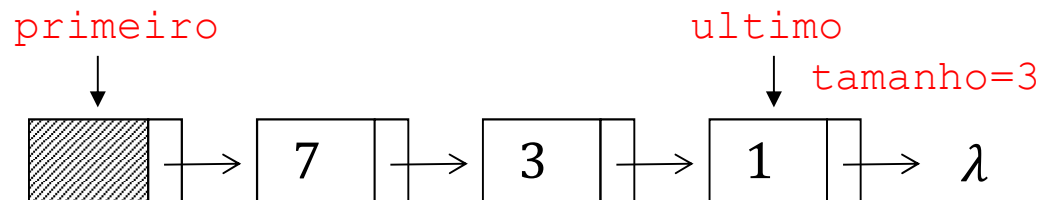
Class Lista Encadeada - Inserção

```
void ListaEncadeada::InsereInicio(TipoItem item)
{
    TipoCelula *nova;

    nova = new TipoCelula();
    nova->item = item;
    nova->prox = primeiro->prox;
    primeiro->prox = nova;
    tamanho++;

    if(nova->prox == NULL)
        ultimo = nova;
};
```

$O(1)$



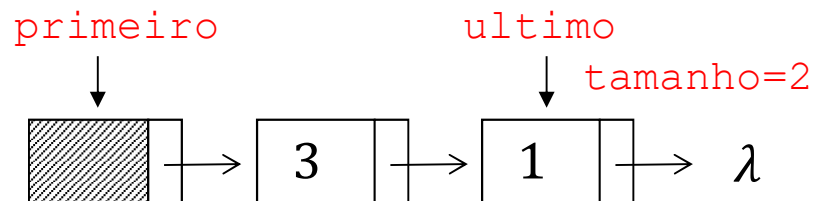
ListaEncadeada L;
TipoItem x;

x.SetChave(7)
L.InsereInicio(x)

Class Lista Encadeada - Inserção

```
void ListaEncadeada::InsereFinal(TipoItem item)
{
    TipoCelula *nova;

    nova = new TipoCelula();
    nova->item = item;
    ultimo->prox = nova;
    ultimo = nova;
    tamanho++;
};
```



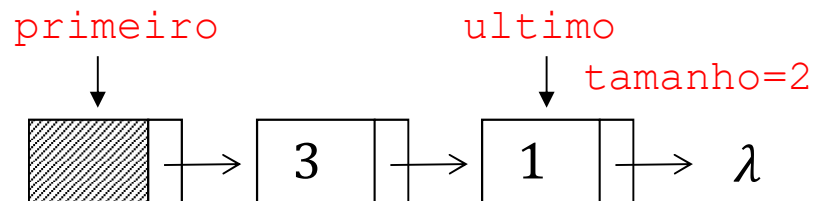
```
ListaEncadeada L;  
TipoItem x;
```

```
x.SetChave(7)  
L.InsereFinal(x)
```

Class Lista Encadeada - Inserção

```
void ListaEncadeada::InsereFinal(TipoItem item)
{
    TipoCelula *nova;

    nova = new TipoCelula();
    nova->item = item;
    ultimo->prox = nova;
    ultimo = nova;
    tamanho++;
};
```



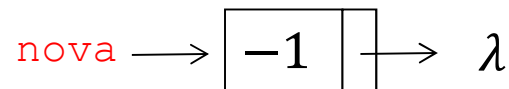
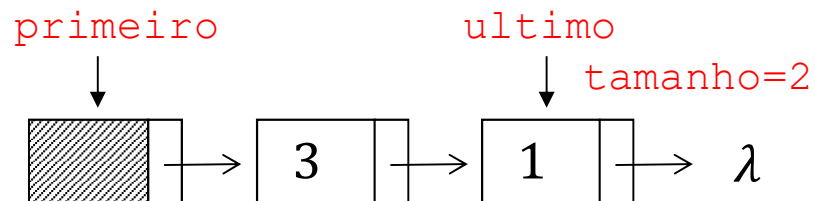
```
ListaEncadeada L;
TipoItem x;
```

```
x.SetChave(7)
L.InsereFinal(x)
```

Class Lista Encadeada - Inserção

```
void ListaEncadeada::InsereFinal(TipoItem item)
{
    TipoCelula *nova;

    nova = new TipoCelula();
    nova->item = item;
    ultimo->prox = nova;
    ultimo = nova;
    tamanho++;
};
```



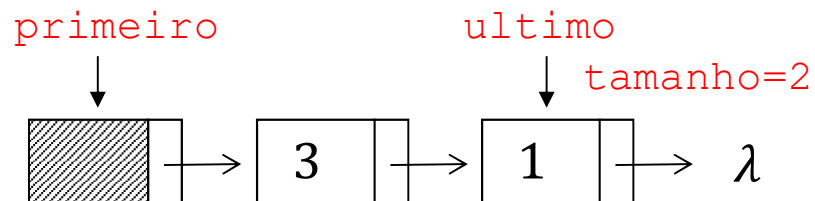
```
ListaEncadeada L;
TipoItem x;
```

```
x.SetChave(7)
L.InsereFinal(x)
```

Class Lista Encadeada - Inserção

```
void ListaEncadeada::InsereFinal(TipoItem item)
{
    TipoCelula *nova;

    nova = new TipoCelula();
    nova->item = item;
    ultimo->prox = nova;
    ultimo = nova;
    tamanho++;
};
```



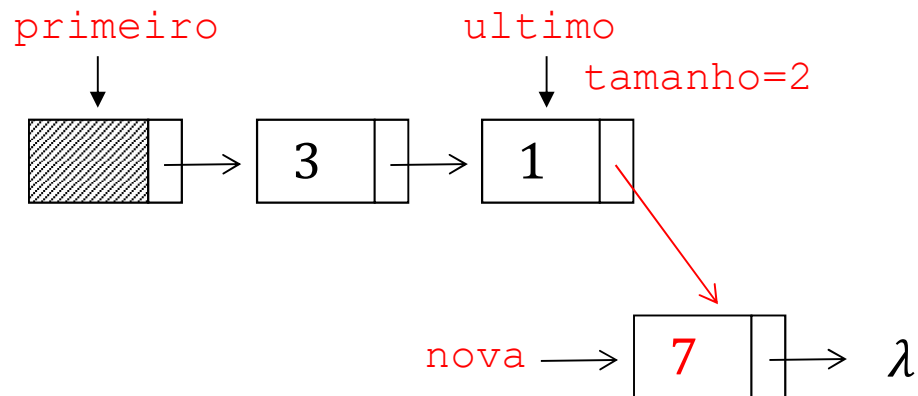
ListaEncadeada L;
TipoItem x;

x.SetChave(7)
L.InsereFinal(x)

Class Lista Encadeada - Inserção

```
void ListaEncadeada::InsereFinal(TipoItem item)
{
    TipoCelula *nova;

    nova = new TipoCelula();
    nova->item = item;
    ultimo->prox = nova;
    ultimo = nova;
    tamanho++;
};
```



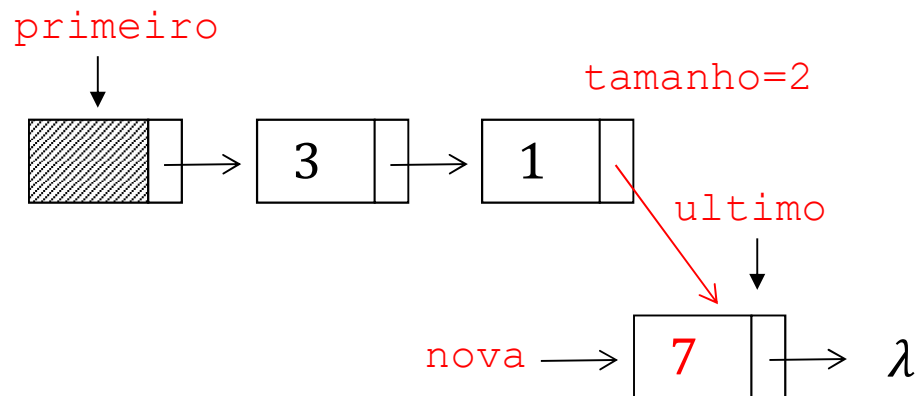
```
ListaEncadeada L;
TipoItem x;
```

```
x.SetChave(7)
L.InsereFinal(x)
```

Class Lista Encadeada - Inserção

```
void ListaEncadeada::InsereFinal(TipoItem item)
{
    TipoCelula *nova;

    nova = new TipoCelula();
    nova->item = item;
    ultimo->prox = nova;
    ultimo = nova;
    tamanho++;
};
```



```
ListaEncadeada L;  
TipoItem x;
```

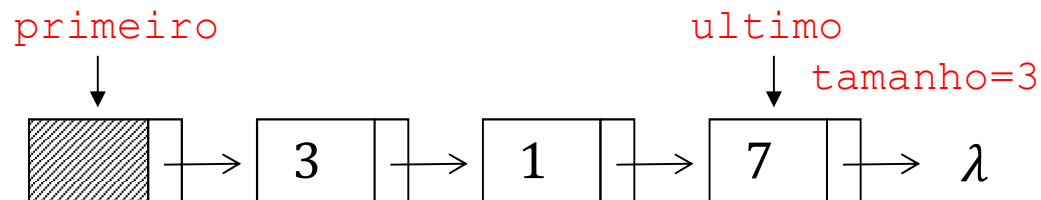
```
x.SetChave(7)  
L.InsereFinal(x)
```

Class Lista Encadeada - Inserção

```
void ListaEncadeada::InsereFinal(TipoItem item)
{
    TipoCelula *nova;

    nova = new TipoCelula();
    nova->item = item;
    ultimo->prox = nova;
    ultimo = nova;
    tamanho++;
};
```

$O(1)$



```
ListaEncadeada L;
TipoItem x;
```

```
x.SetChave(7)
L.InsereFinal(x)
```

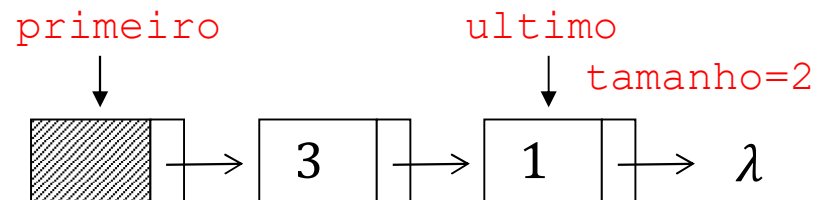
Class Lista Encadeada - Inserção

```
void ListaEncadeada::InserePosicao(TipoItem item, int pos) {
    TipoCelula *p, *nova;

    p = Posiciona(pos,true); // posiciona na célula anterior

    nova = new TipoCelula();
    nova->item = item;
    nova->prox = p->prox;
    p->prox = nova;
    tamanho++;

    if(nova->prox == NULL)
        ultimo = nova;
};
```

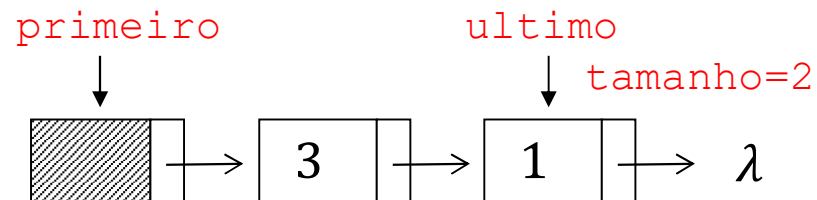


```
ListaEncadeada L;
TipoItem x;

x.SetChave(7)
L.InserePosicao(x,2)
```

Class Lista Encadeada - Inserção

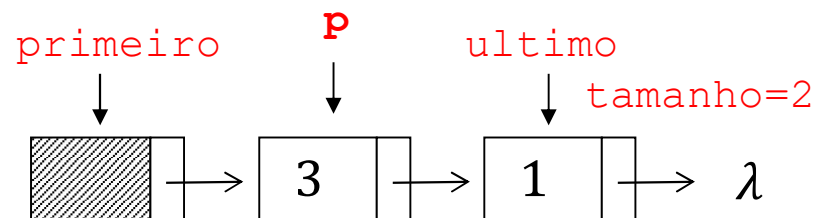
```
void ListaEncadeada::InserePosicao(TipoItem item, int pos) {  
    TipoCelula *p, *nova;  
  
    p = Posiciona(pos,true); // posiciona na célula anterior  
  
    nova = new TipoCelula();  
    nova->item = item;  
    nova->prox = p->prox;  
    p->prox = nova;  
    tamanho++;  
  
    if(nova->prox == NULL)  
        ultimo = nova;  
};
```



```
ListaEncadeada L;  
TipoItem x;  
  
x.SetChave(7)  
L.InserePosicao(x,2)
```

Class Lista Encadeada - Inserção

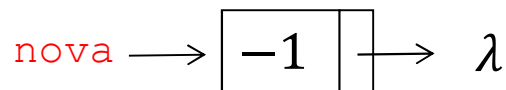
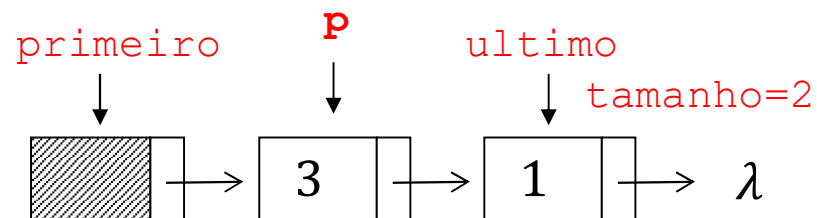
```
void ListaEncadeada::InserePosicao(TipoItem item, int pos) {  
    TipoCelula *p, *nova;  
  
    p = Posiciona(pos,true); // posiciona na célula anterior  
  
    nova = new TipoCelula();  
    nova->item = item;  
    nova->prox = p->prox;  
    p->prox = nova;  
    tamanho++;  
  
    if(nova->prox == NULL)  
        ultimo = nova;  
};
```



```
ListaEncadeada L;  
TipoItem x;  
  
x.SetChave(7)  
L.InserePosicao(x,2)
```

Class Lista Encadeada - Inserção

```
void ListaEncadeada::InserePosicao(TipoItem item, int pos) {  
    TipoCelula *p, *nova;  
  
    p = Posiciona(pos,true); // posiciona na célula anterior  
  
    nova = new TipoCelula();  
    nova->item = item;  
    nova->prox = p->prox;  
    p->prox = nova;  
    tamanho++;  
  
    if(nova->prox == NULL)  
        ultimo = nova;  
};
```



```
ListaEncadeada L;  
TipoItem x;  
  
x.SetChave(7)  
L.InserePosicao(x,2)
```

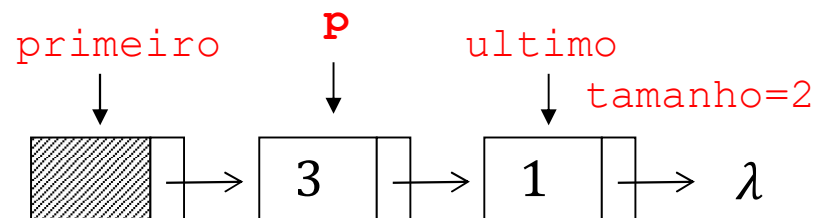
Class Lista Encadeada - Inserção

```
void ListaEncadeada::InserePosicao(TipoItem item, int pos) {
    TipoCelula *p, *nova;

    p = Posiciona(pos,true); // posiciona na célula anterior

    nova = new TipoCelula();
    nova->item = item;
    nova->prox = p->prox;
    p->prox = nova;
    tamanho++;

    if(nova->prox == NULL)
        ultimo = nova;
};
```



```
ListaEncadeada L;
TipoItem x;

x.SetChave(7)
L.InserePosicao(x,2)
```

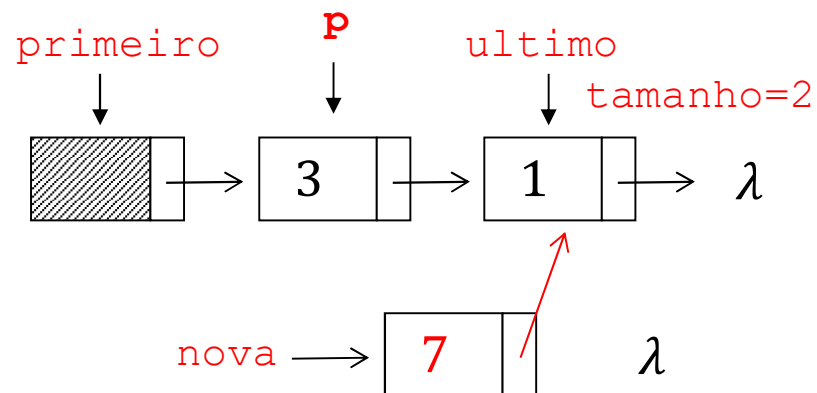

Class Lista Encadeada - Inserção

```
void ListaEncadeada::InserePosicao(TipoItem item, int pos) {
    TipoCelula *p, *nova;

    p = Posiciona(pos,true); // posiciona na célula anterior

    nova = new TipoCelula();
    nova->item = item;
    nova->prox = p->prox;
    p->prox = nova;
    tamanho++;

    if(nova->prox == NULL)
        ultimo = nova;
};
```



```
ListaEncadeada L;
TipoItem x;

x.SetChave(7)
L.InserePosicao(x,2)
```

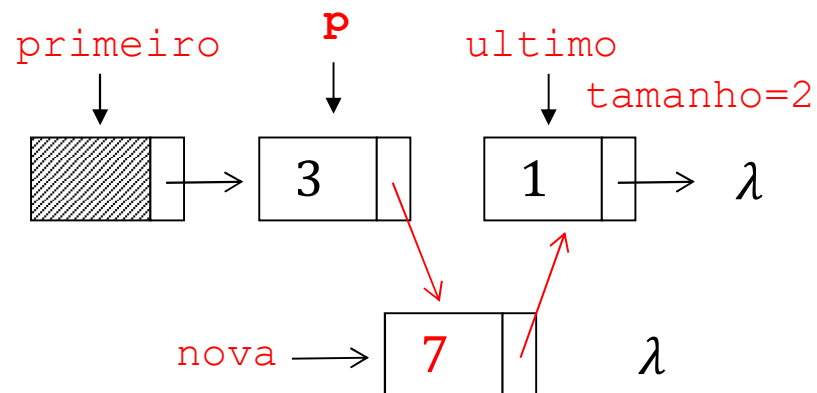
Class Lista Encadeada - Inserção

```
void ListaEncadeada::InserePosicao(TipoItem item, int pos) {
    TipoCelula *p, *nova;

    p = Posiciona(pos,true); // posiciona na célula anterior

    nova = new TipoCelula();
    nova->item = item;
    nova->prox = p->prox;
    p->prox = nova;
    tamanho++;

    if(nova->prox == NULL)
        ultimo = nova;
};
```

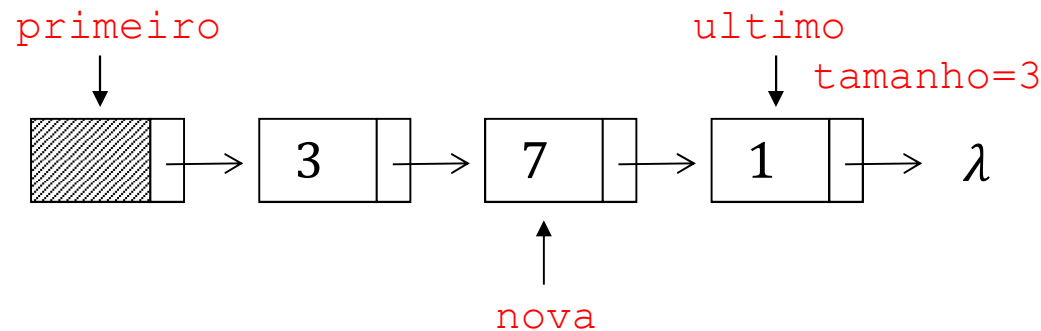


```
ListaEncadeada L;
TipoItem x;

x.SetChave(7)
L.InserePosicao(x,2)
```

Class Lista Encadeada - Inserção

```
void ListaEncadeada::InserePosicao(TipoItem item, int pos) {  
    TipoCelula *p, *nova;  
  
    p = Posiciona(pos,true); // posiciona na célula anterior  
  
    nova = new TipoCelula();  
    nova->item = item;  
    nova->prox = p->prox;  
    p->prox = nova;  
    tamanho++;  
  
    if(nova->prox == NULL)  
        ultimo = nova;  
};
```



```
ListaEncadeada L;  
TipoItem x;  
  
x.SetChave(7)  
L.InserePosicao(x,2)
```

Class Lista Encadeada - Inserção

```
void ListaEncadeada::InserePosicao(TipoItem item, int pos) {
    TipoCelula *p, *nova;

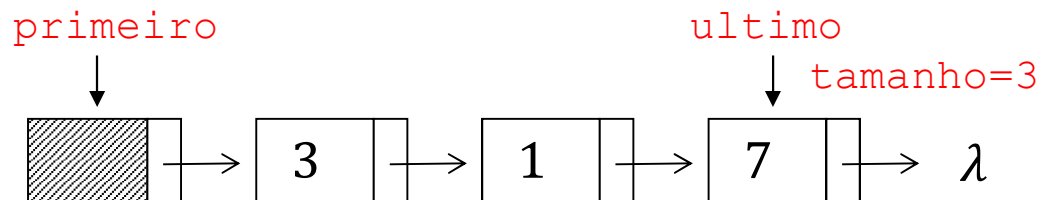
    p = Posiciona(pos,true); // posiciona na célula anterior

    nova = new TipoCelula();
    nova->item = item;
    nova->prox = p->prox;
    p->prox = nova;
    tamanho++;

    if(nova->prox == NULL)
        ultimo = nova;
};
```

Melhor Caso $O(1)$

Pior Caso $O(n)$



```
ListaEncadeada L;
TipoItem x;

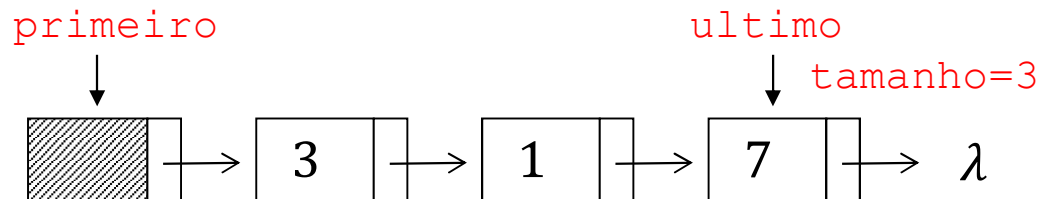
x.SetChave(7)
L.InserePosicao(x,2)
```

Class Lista Encadeada - Remoção

- Da mesma forma, a remoção pode ser feita no início, no final, ou em uma posição p qualquer
- Deve-se posicionar um apontador auxiliar **antes** da posição a ser removida
- Se estiver removendo na última posição, deve-se atualizar o apontador *ultimo*
- Deve-se verificar se há elementos e se a posição de remoção é válida
 - Gera uma exceção que pode ser tratada por quem chamou o método.
- O elemento removido é retornado pelo método e a célula desalocada da memória

Class Lista Encadeada - Remoção

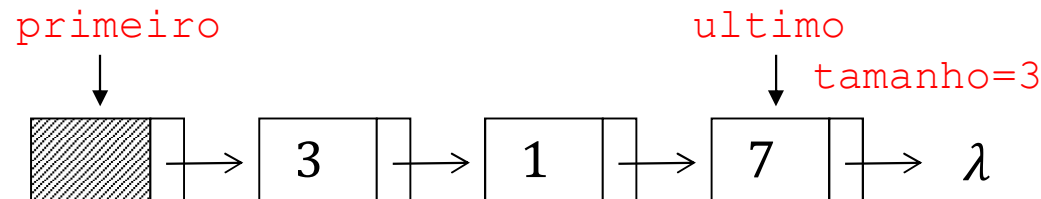
```
TipoItem ListaEncadeada::RemoveInicio() {;  
    TipoItem aux;  
    TipoCelula *p;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    p = primeiro->prox;  
    primeiro->prox = p->prox;  
    tamanho--;  
    if(primeiro->prox == NULL)  
        ultimo = primeiro;  
    aux = p->item;  
    delete p;  
  
    return aux;  
}
```



```
ListaEncadeada L;  
TipoItem x;  
  
x=L.RemoveInicio();  
x.Imprime();
```

Class Lista Encadeada - Remoção

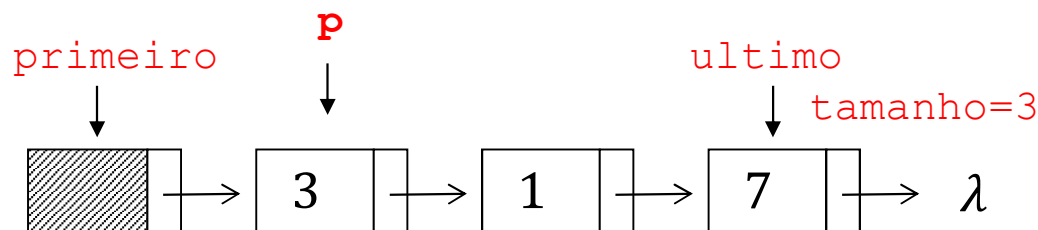
```
TipoItem ListaEncadeada::RemoveInicio() {  
    TipoItem aux;  
    TipoCelula *p;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    p = primeiro->prox;  
    primeiro->prox = p->prox;  
    tamanho--;  
    if(primeiro->prox == NULL)  
        ultimo = primeiro;  
    aux = p->item;  
    delete p;  
  
    return aux;  
}
```



```
ListaEncadeada L;  
TipoItem x;  
  
x=L.RemoveInicio();  
x.Imprime();
```

Class Lista Encadeada - Remoção

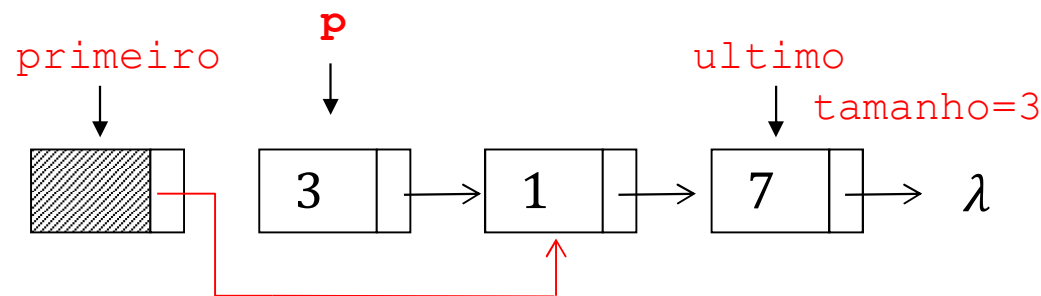
```
TipoItem ListaEncadeada::RemoveInicio() {  
    TipoItem aux;  
    TipoCelula *p;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    p = primeiro->prox;  
    primeiro->prox = p->prox;  
    tamanho--;  
    if(primeiro->prox == NULL)  
        ultimo = primeiro;  
    aux = p->item;  
    delete p;  
  
    return aux;  
}
```



```
ListaEncadeada L;  
TipoItem x;  
  
x=L.RemoveInicio();  
x.Imprime();
```


Class Lista Encadeada - Remoção

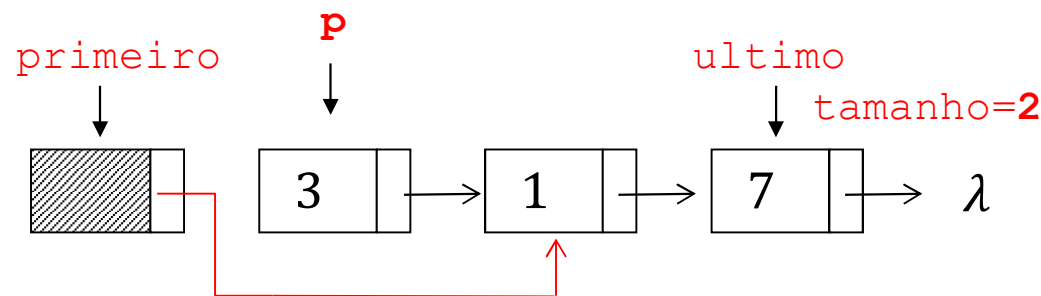
```
TipoItem ListaEncadeada::RemoveInicio() {  
    TipoItem aux;  
    TipoCelula *p;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    p = primeiro->prox;  
    primeiro->prox = p->prox;  
    tamanho--;  
    if (primeiro->prox == NULL)  
        ultimo = primeiro;  
    aux = p->item;  
    delete p;  
  
    return aux;  
}
```



```
ListaEncadeada L;  
TipoItem x;  
  
x=L.RemoveInicio();  
x.Imprime();
```

Class Lista Encadeada - Remoção

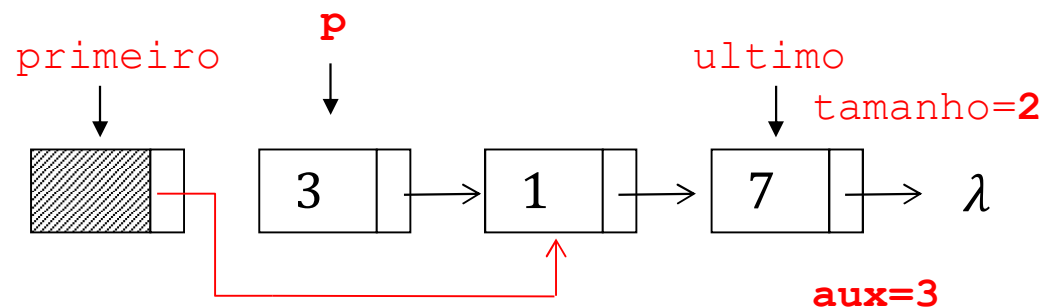
```
TipoItem ListaEncadeada::RemoveInicio() {  
    TipoItem aux;  
    TipoCelula *p;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    p = primeiro->prox;  
    primeiro->prox = p->prox;  
    tamanho--;  
    if(primeiro->prox == NULL)  
        ultimo = primeiro;  
    aux = p->item;  
    delete p;  
  
    return aux;  
}
```



```
ListaEncadeada L;  
TipoItem x;  
  
x=L.RemoveInicio();  
x.Imprime();
```

Class Lista Encadeada - Remoção

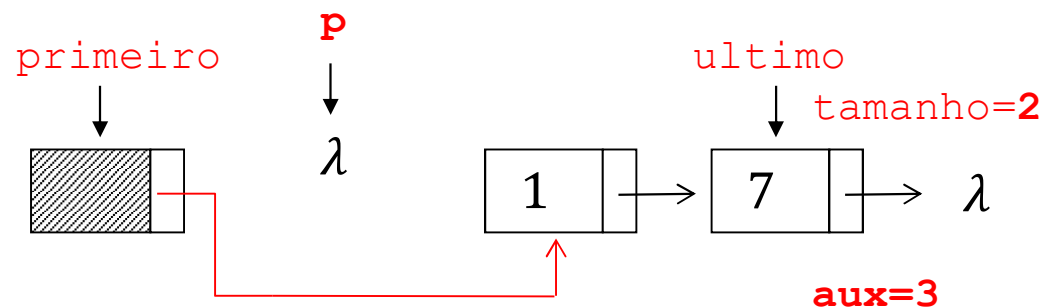
```
TipoItem ListaEncadeada::RemoveInicio() {  
    TipoItem aux;  
    TipoCelula *p;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    p = primeiro->prox;  
    primeiro->prox = p->prox;  
    tamanho--;  
    if(primeiro->prox == NULL)  
        ultimo = primeiro;  
    aux = p->item;  
    delete p;  
  
    return aux;  
}
```



```
ListaEncadeada L;  
TipoItem x;  
  
x=L.RemoveInicio();  
x.Imprime();
```

Class Lista Encadeada - Remoção

```
TipoItem ListaEncadeada::RemoveInicio() {;  
    TipoItem aux;  
    TipoCelula *p;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    p = primeiro->prox;  
    primeiro->prox = p->prox;  
    tamanho--;  
    if(primeiro->prox == NULL)  
        ultimo = primeiro;  
    aux = p->item;  
    delete p;  
  
    return aux;  
}
```

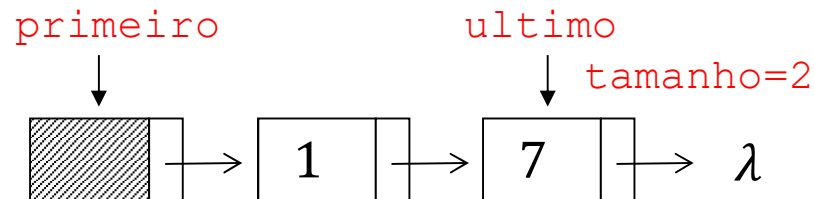


```
ListaEncadeada L;  
TipoItem x;  
  
x=L.RemoveInicio();  
x.Imprime();
```

Class Lista Encadeada - Remoção

```
TipoItem ListaEncadeada::RemoveInicio() {;  
    TipoItem aux;  
    TipoCelula *p;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    p = primeiro->prox;  
    primeiro->prox = p->prox;  
    tamanho--;  
    if(primeiro->prox == NULL)  
        ultimo = primeiro;  
    aux = p->item;  
    delete p;  
  
    return aux;  
}
```

$O(1)$

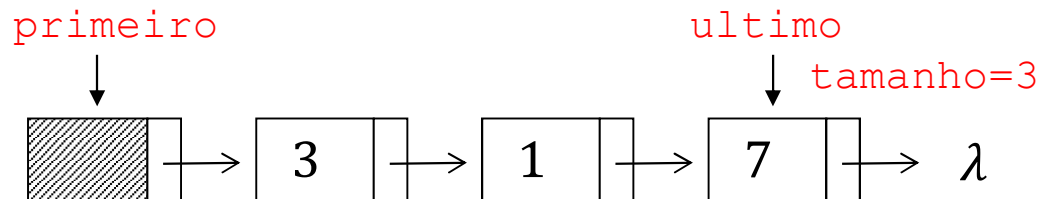


```
ListaEncadeada L;  
TipoItem x;  
  
x=L.RemoveInicio();  
x.Imprime();
```

3

Class Lista Encadeada - Remoção

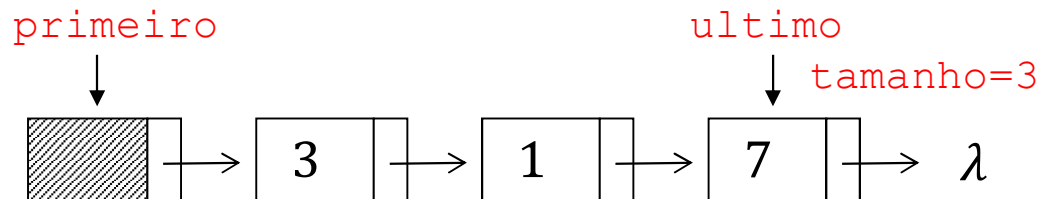
```
TipoItem ListaEncadeada::RemoveFinal() {  
    TipoItem aux;  
    TipoCelula *p;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    // posiciona p na celula anterior à última  
    p = Posiciona(tamanho, true);  
  
    p->prox = NULL;  
    tamanho--;  
    aux = ultimo->item;  
    delete ultimo;  
    ultimo = p;  
  
    return aux;  
}
```



```
ListaEncadeada L;  
TipoItem x;  
  
x=L.RemoveFinal();  
x.Imprime();
```

Class Lista Encadeada - Remoção

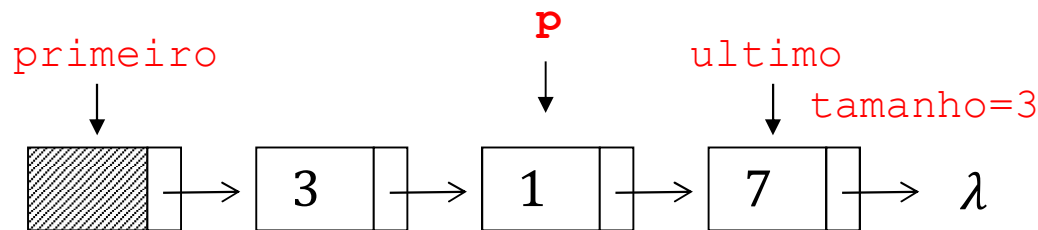
```
TipoItem ListaEncadeada::RemoveFinal() {  
    TipoItem aux;  
    TipoCelula *p;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    // posiciona p na celula anterior à última  
    p = Posiciona(tamanho, true);  
  
    p->prox = NULL;  
    tamanho--;  
    aux = ultimo->item;  
    delete ultimo;  
    ultimo = p;  
  
    return aux;  
}
```



```
ListaEncadeada L;  
TipoItem x;  
  
x=L.RemoveFinal();  
x.Imprime();
```

Class Lista Encadeada - Remoção

```
TipoItem ListaEncadeada::RemoveFinal() {  
    TipoItem aux;  
    TipoCelula *p;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    // posiciona p na celula anterior à última  
    p = Posiciona(tamanho, true);  
  
    p->prox = NULL;  
    tamanho--;  
    aux = ultimo->item;  
    delete ultimo;  
    ultimo = p;  
  
    return aux;  
}
```

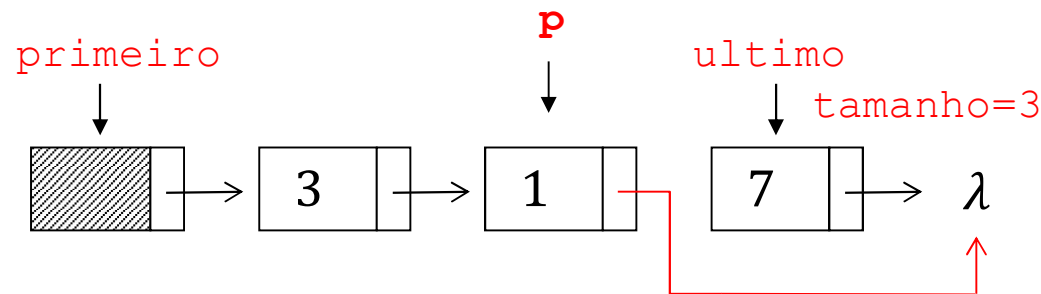


```
ListaEncadeada L;  
TipoItem x;
```

```
x=L.RemoveFinal();  
x.Imprime();
```


Class Lista Encadeada - Remoção

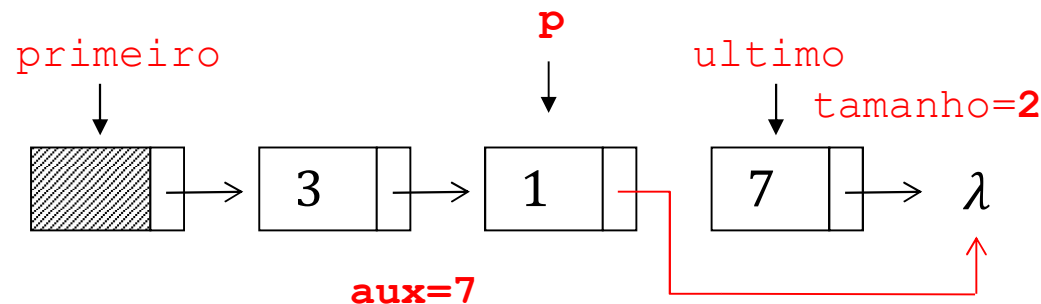
```
TipoItem ListaEncadeada::RemoveFinal() {  
    TipoItem aux;  
    TipoCelula *p;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    // posiciona p na celula anterior à última  
    p = Posiciona(tamanho, true);  
  
    p->prox = NULL;  
    tamanho--;  
    aux = ultimo->item;  
    delete ultimo;  
    ultimo = p;  
  
    return aux;  
}
```



```
ListaEncadeada L;  
TipoItem x;  
  
x=L.RemoveFinal();  
x.Imprime();
```

Class Lista Encadeada - Remoção

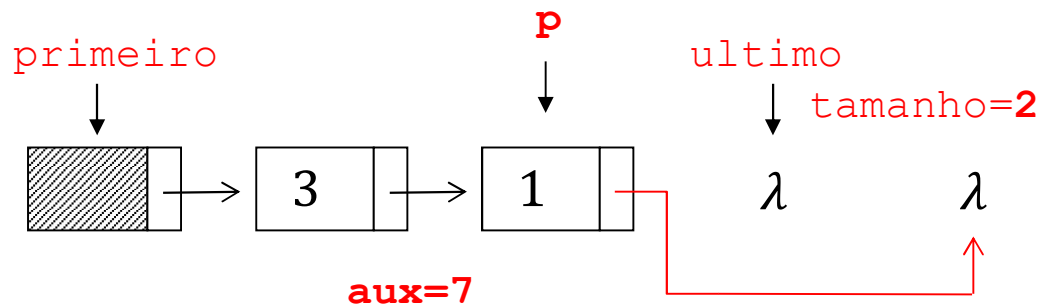
```
TipoItem ListaEncadeada::RemoveFinal() {  
    TipoItem aux;  
    TipoCelula *p;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    // posiciona p na celula anterior à última  
    p = Posiciona(tamanho, true);  
  
    p->prox = NULL;  
    tamanho--;  
    aux = ultimo->item;  
    delete ultimo;  
    ultimo = p;  
  
    return aux;  
}
```



```
ListaEncadeada L;  
TipoItem x;  
  
x=L.RemoveFinal();  
x.Imprime();
```

Class Lista Encadeada - Remoção

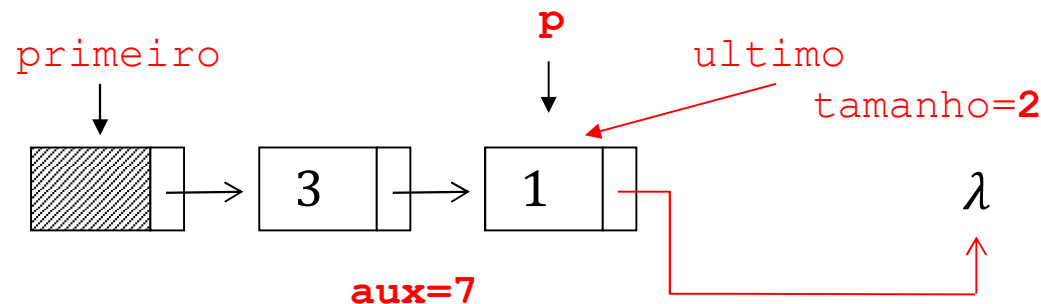
```
TipoItem ListaEncadeada::RemoveFinal() {  
    TipoItem aux;  
    TipoCelula *p;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    // posiciona p na celula anterior à última  
    p = Posiciona(tamanho, true);  
  
    p->prox = NULL;  
    tamanho--;  
    aux = ultimo->item;  
    delete ultimo;  
    ultimo = p;  
  
    return aux;  
}
```



```
ListaEncadeada L;  
TipoItem x;  
  
x=L.RemoveFinal();  
x.Imprime();
```

Class Lista Encadeada - Remoção

```
TipoItem ListaEncadeada::RemoveFinal() {  
    TipoItem aux;  
    TipoCelula *p;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    // posiciona p na celula anterior à última  
    p = Posiciona(tamanho, true);  
  
    p->prox = NULL;  
    tamanho--;  
    aux = ultimo->item;  
    delete ultimo;  
    ultimo = p;  
  
    return aux;  
}
```

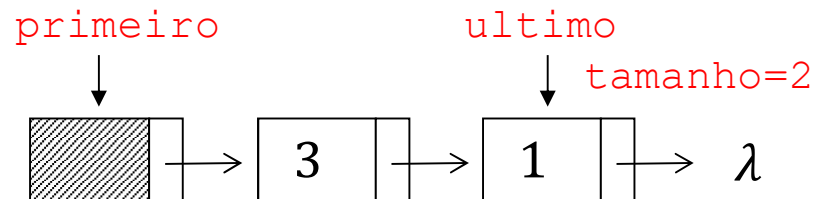


```
ListaEncadeada L;  
TipoItem x;  
  
x=L.RemoveFinal();  
x.Imprime();
```

Class Lista Encadeada - Remoção

```
TipoItem ListaEncadeada::RemoveFinal() {  
    TipoItem aux;  
    TipoCelula *p;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    // posiciona p na celula anterior à última  
    p = Posiciona(tamanho, true);  
  
    p->prox = NULL;  
    tamanho--;  
    aux = ultimo->item;  
    delete ultimo;  
    ultimo = p;  
  
    return aux;  
}
```

$O(n)$

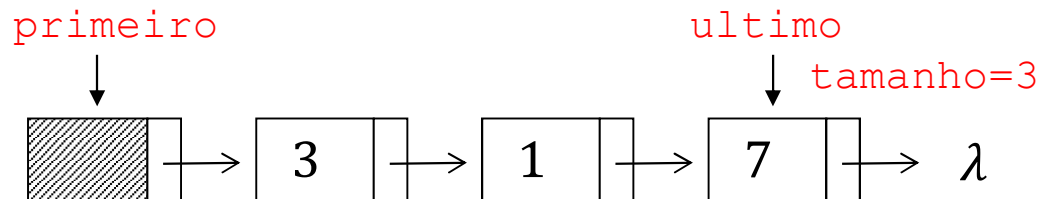


```
ListaEncadeada L;  
TipoItem x;  
  
x=L.RemoveFinal();  
x.Imprime();
```

7

Class Lista Encadeada - Remoção

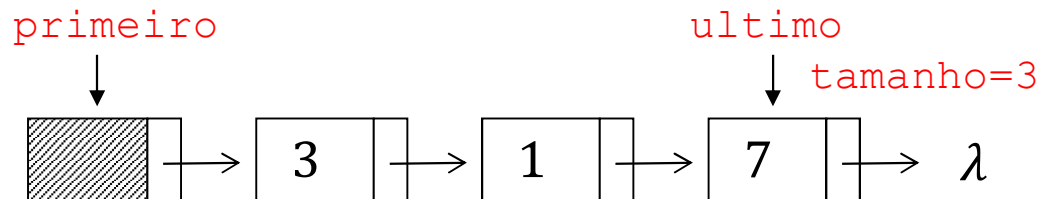
```
TipoItem ListaEncadeada::RemovePosicao(int pos) {  
    TipoItem aux;  
    TipoCelula *p, *q;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    // posiciona p na celula anterior à pos  
    p = Posiciona(pos, true);  
    q = p->prox;  
    p->prox = q->prox;  
    tamanho--;  
    aux = q->item;  
    delete q;  
    if(p->prox == NULL)  
        ultimo = p;  
    return aux;  
}
```



```
ListaEncadeada L;  
TipoItem x;  
  
x=L.RemovePosicao(2);  
x.Imprime();
```

Class Lista Encadeada - Remoção

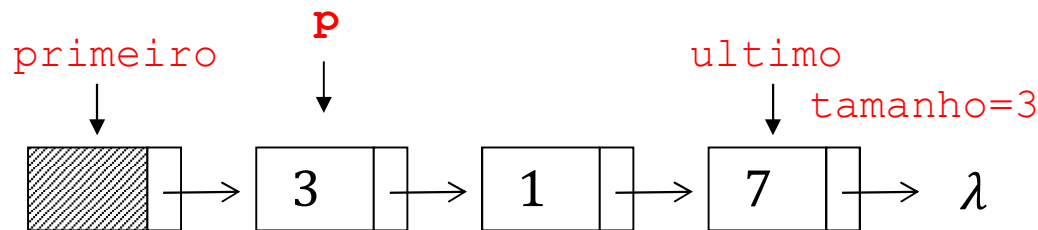
```
TipoItem ListaEncadeada::RemovePosicao(int pos) {  
    TipoItem aux;  
    TipoCelula *p, *q;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    // posiciona p na celula anterior à pos  
    p = Posiciona(pos, true);  
    q = p->prox;  
    p->prox = q->prox;  
    tamanho--;  
    aux = q->item;  
    delete q;  
    if(p->prox == NULL)  
        ultimo = p;  
    return aux;  
}
```



```
ListaEncadeada L;  
TipoItem x;  
  
x=L.RemovePosicao(2);  
x.Imprime();
```

Class Lista Encadeada - Remoção

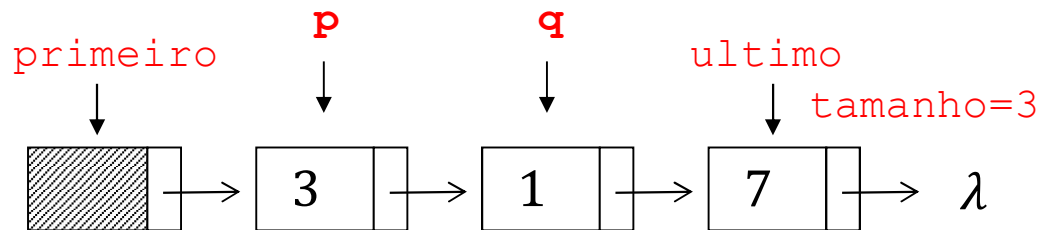
```
TipoItem ListaEncadeada::RemovePosicao(int pos) {  
    TipoItem aux;  
    TipoCelula *p, *q;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    // posiciona p na celula anterior à pos  
    p = Posiciona(pos, true);  
    q = p->prox;  
    p->prox = q->prox;  
    tamanho--;  
    aux = q->item;  
    delete q;  
    if(p->prox == NULL)  
        ultimo = p;  
    return aux;  
}
```



```
ListaEncadeada L;  
TipoItem x;  
  
x=L.RemovePosicao(2);  
x.Imprime();
```


Class Lista Encadeada - Remoção

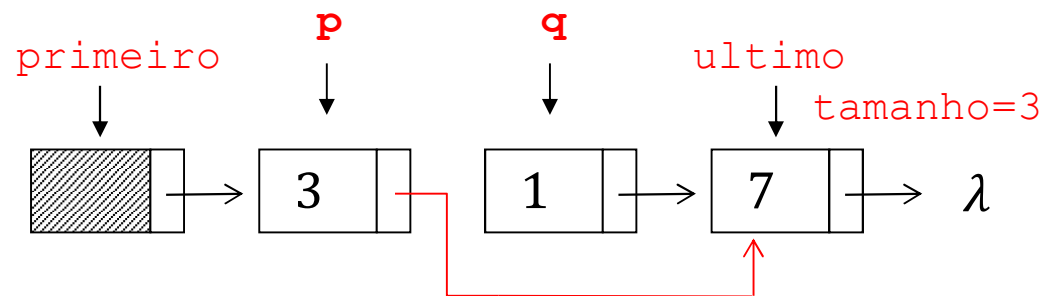
```
TipoItem ListaEncadeada::RemovePosicao(int pos) {  
    TipoItem aux;  
    TipoCelula *p, *q;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    // posiciona p na celula anterior à pos  
    p = Posiciona(pos, true);  
    q = p->prox;  
    p->prox = q->prox;  
    tamanho--;  
    aux = q->item;  
    delete q;  
    if(p->prox == NULL)  
        ultimo = p;  
    return aux;  
}
```



```
ListaEncadeada L;  
TipoItem x;  
  
x=L.RemovePosicao(2);  
x.Imprime();
```

Class Lista Encadeada - Remoção

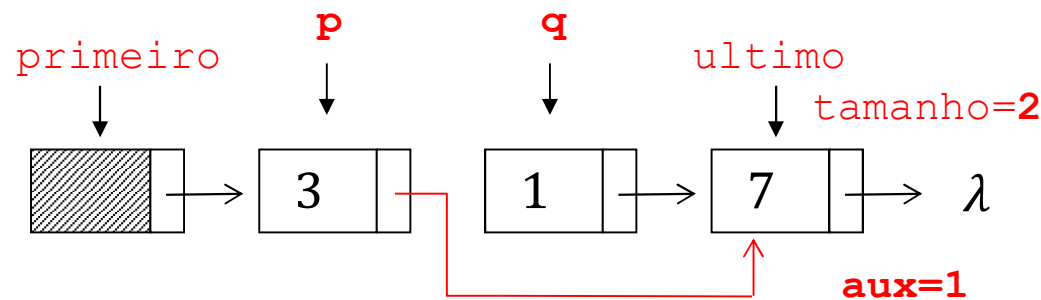
```
TipoItem ListaEncadeada::RemovePosicao(int pos) {  
    TipoItem aux;  
    TipoCelula *p, *q;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    // posiciona p na celula anterior à pos  
    p = Posiciona(pos, true);  
    q = p->prox;  
    p->prox = q->prox;  
    tamanho--;  
    aux = q->item;  
    delete q;  
    if(p->prox == NULL)  
        ultimo = p;  
    return aux;  
}
```



```
ListaEncadeada L;  
TipoItem x;  
  
x=L.RemovePosicao(2);  
x.Imprime();
```

Class Lista Encadeada - Remoção

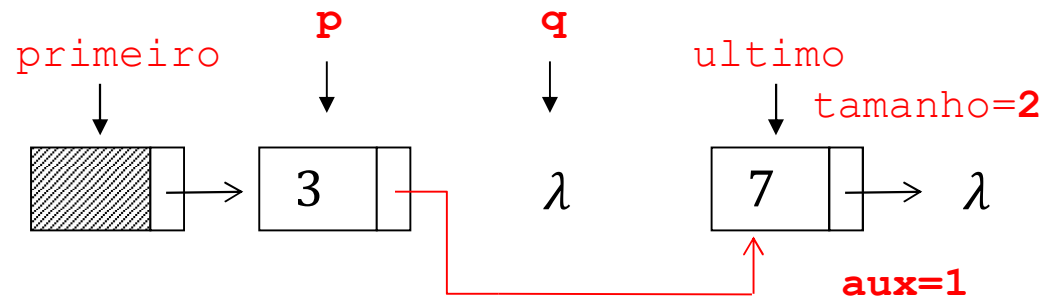
```
TipoItem ListaEncadeada::RemovePosicao(int pos) {  
    TipoItem aux;  
    TipoCelula *p, *q;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    // posiciona p na celula anterior à pos  
    p = Posiciona(pos, true);  
    q = p->prox;  
    p->prox = q->prox;  
    tamanho--;  
    aux = q->item;  
    delete q;  
    if(p->prox == NULL)  
        ultimo = p;  
    return aux;  
}
```



```
ListaEncadeada L;  
TipoItem x;  
  
x=L.RemovePosicao(2);  
x.Imprime();
```

Class Lista Encadeada - Remoção

```
TipoItem ListaEncadeada::RemovePosicao(int pos) {  
    TipoItem aux;  
    TipoCelula *p, *q;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    // posiciona p na celula anterior à pos  
    p = Posiciona(pos, true);  
    q = p->prox;  
    p->prox = q->prox;  
    tamanho--;  
    aux = q->item;  
    delete q;  
    if (p->prox == NULL)  
        ultimo = p;  
    return aux;  
}
```



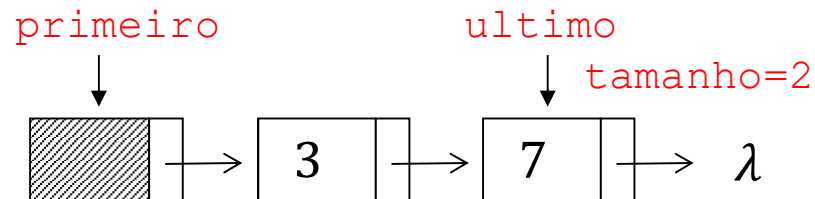
```
ListaEncadeada L;  
TipoItem x;  
  
x=L.RemovePosicao(2);  
x.Imprime();
```

Class Lista Encadeada - Remoção

```
TipoItem ListaEncadeada::RemovePosicao(int pos) {  
    TipoItem aux;  
    TipoCelula *p, *q;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    // posiciona p na celula anterior à pos  
    p = Posiciona(pos, true);  
    q = p->prox;  
    p->prox = q->prox;  
    tamanho--;  
    aux = q->item;  
    delete q;  
    if(p->prox == NULL)  
        ultimo = p;  
    return aux;  
}
```

Melhor Caso $O(1)$

Pior Caso $O(n)$



```
ListaEncadeada L;  
TipoItem x;  
  
x=L.RemovePosicao(2);  
x.Imprime();
```

1

Class Lista Encadeada - Pesquisa

- Pesquisa por um item com uma determinada chave
 - Retorna o item encontrado ou um *flag* (-1)

```
TipoItem  ListaEncadeada::Pesquisa(TipoChave c) {
    TipoItem aux;
    TipoCelula *p;

    if (tamanho == 0)
        throw "ERRO: Lista vazia!";

    p = primeiro->prox;
    aux.SetChave(-1);
    while (p!=NULL) {
        if (p->item.GetChave() == c) {
            aux = p->item;
            break;
        }
        p = p->prox;
    }

    return aux;
};
```

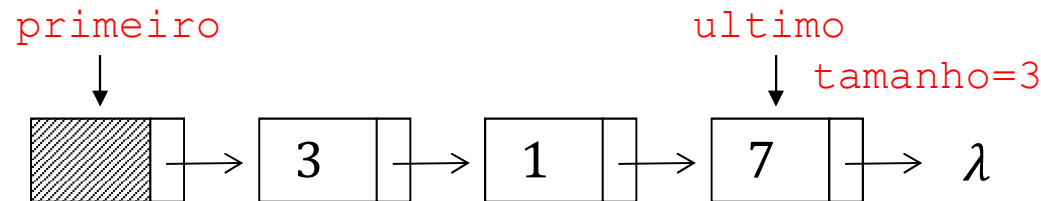
Class Lista Encadeada - Pesquisa

```
TipoItem ListaEncadeada::Pesquisa(TipoChave c) {
    TipoItem aux;
    TipoCelula *p;

    if (tamanho == 0)
        throw "ERRO: Lista vazia!";

    p = primeiro->prox;
    aux.SetChave(-1);
    while (p!=NULL) {
        if (p->item.GetChave() == c) {
            aux = p->item;
            break;
        }
        p = p->prox;
    }

    return aux;
};
```



```
ListaEncadeada L;
TipoItem x;

x = L.Pesquisa(1);
x.Imprime();
```

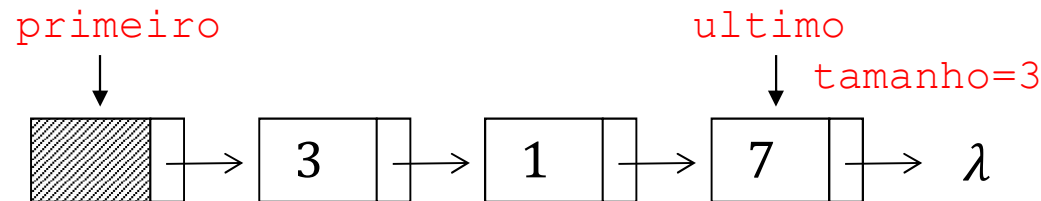
Class Lista Encadeada - Pesquisa

```
TipoItem ListaEncadeada::Pesquisa(TipoChave c) {
    TipoItem aux;
    TipoCelula *p;

    if (tamanho == 0)
        throw "ERRO: Lista vazia!";

    p = primeiro->prox;
    aux.SetChave(-1);
    while (p!=NULL) {
        if (p->item.GetChave() == c) {
            aux = p->item;
            break;
        }
        p = p->prox;
    }

    return aux;
};
```

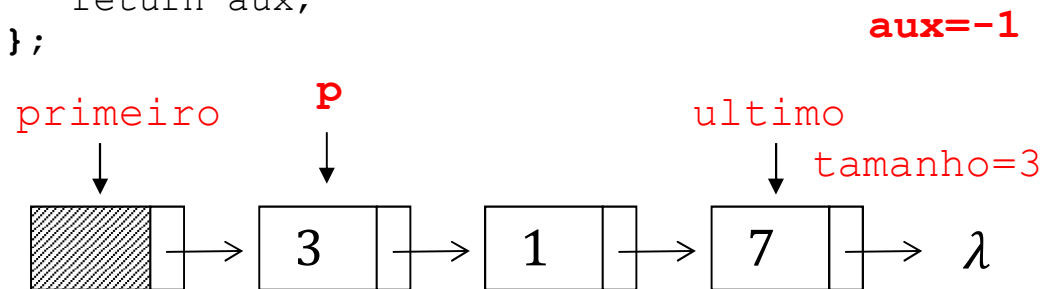


```
ListaEncadeada L;
TipoItem x;

x = L.Pesquisa(1);
x.Imprime();
```


Class Lista Encadeada - Pesquisa

```
TipoItem ListaEncadeada::Pesquisa(TipoChave c) {  
    TipoItem aux;  
    TipoCelula *p;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    p = primeiro->prox;  
    aux.SetChave(-1);  
    while (p!=NULL) {  
        if (p->item.GetChave() == c) {  
            aux = p->item;  
            break;  
        }  
        p = p->prox;  
    }  
  
    return aux;  
};
```



```
ListaEncadeada L;  
TipoItem x;  
  
x = L.Pesquisa(1);  
x.Imprime();
```

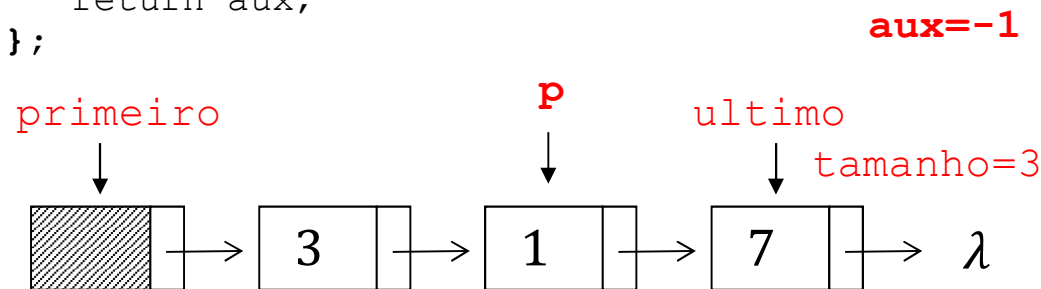
Class Lista Encadeada - Pesquisa

```
TipoItem ListaEncadeada::Pesquisa(TipoChave c) {
    TipoItem aux;
    TipoCelula *p;

    if (tamanho == 0)
        throw "ERRO: Lista vazia!";

    p = primeiro->prox;
    aux.SetChave(-1);
    while (p!=NULL) {
        if (p->item.GetChave() == c) {
            aux = p->item;
            break;
        }
        p = p->prox;
    }

    return aux;
};
```

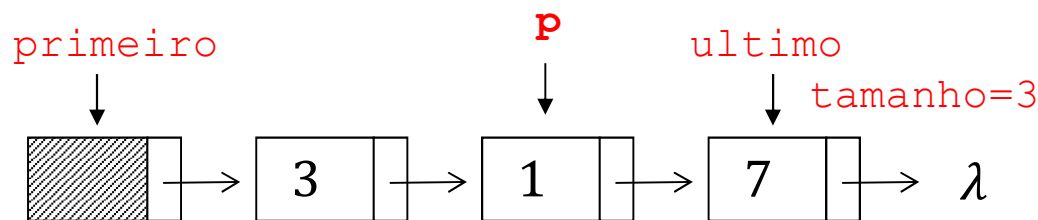


```
ListaEncadeada L;
TipoItem x;

x = L.Pesquisa(1);
x.Imprime();
```

Class Lista Encadeada - Pesquisa

```
TipoItem ListaEncadeada::Pesquisa(TipoChave c) {  
    TipoItem aux;  
    TipoCelula *p;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    p = primeiro->prox;  
    aux.SetChave(-1);  
    while (p!=NULL) {  
        if (p->item.GetChave() == c) {  
            aux = p->item;  
            break;  
        }  
        p = p->prox;  
    }  
  
    return aux;  
};
```



```
ListaEncadeada L;  
TipoItem x;  
  
x = L.Pesquisa(1);  
x.Imprime();
```

Class Lista Encadeada - Pesquisa

```
TipoItem ListaEncadeada::Pesquisa(TipoChave c) {
    TipoItem aux;
    TipoCelula *p;

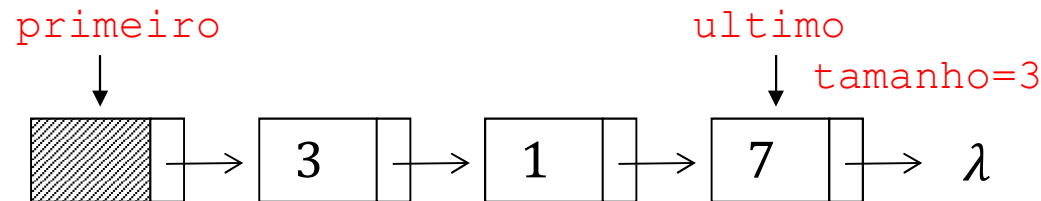
    if (tamanho == 0)
        throw "ERRO: Lista vazia!";

    p = primeiro->prox;
    aux.SetChave(-1);
    while (p!=NULL) {
        if (p->item.GetChave() == c) {
            aux = p->item;
            break;
        }
        p = p->prox;
    }

    return aux;
};
```

Melhor Caso $O(1)$

Pior Caso $O(n)$



```
ListaEncadeada L;
TipoItem x;

x = L.Pesquisa(1);
x.Imprime();
```

1

Class Lista Encadeada - Imprime

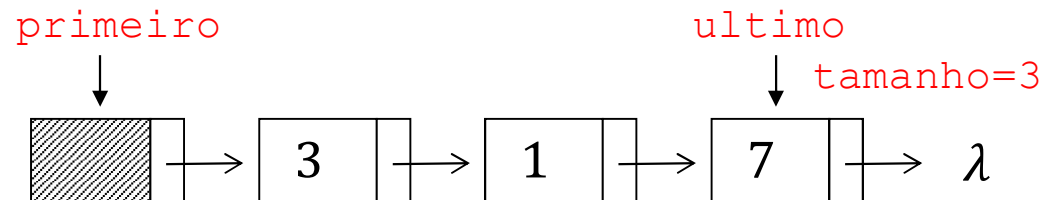
■ Imprime todos os elementos

```
void ListaEncadeada::Imprime() {  
    TipoCelula *p;  
  
    p = primeiro->prox;  
    while (p!=NULL) {  
        p->item.Imprime();  
        p = p->prox;  
    }  
  
    printf("\n");  
};
```

$O(n)$

ListaEncadeada L;

L.Imprime();

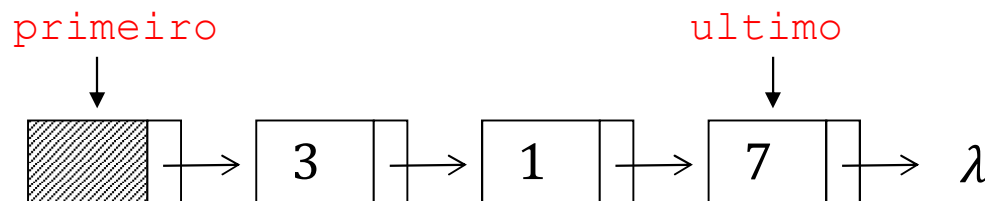


3 1 7

Class Lista Encadeada - Limpa

- “Limpa” a Lista
 - Percorre a lista desalocando memória

```
void ListaEncadeada::Limpa() {  
    TipoCelula *p;  
  
    p = primeiro->prox;  
    while (p!=NULL) {  
        primeiro->prox = p->prox;  
        delete p;  
        p = primeiro->prox;  
    }  
    ultimo = primeiro;  
    tamanho = 0;  
};
```

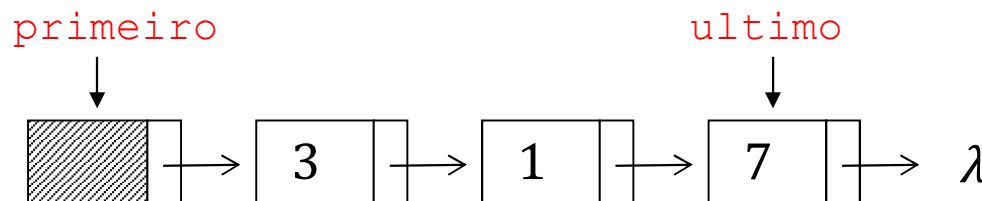


```
ListaEncadeada L;  
...  
L.Limpa();
```

Class Lista Encadeada - Limpa

- “Limpa” a Lista
 - Percorre a lista desalocando memória

```
void ListaEncadeada::Limpa() {  
    TipoCelula *p;  
  
    p = primeiro->prox;  
    while (p!=NULL) {  
        primeiro->prox = p->prox;  
        delete p;  
        p = primeiro->prox;  
    }  
    ultimo = primeiro;  
    tamanho = 0;  
};
```



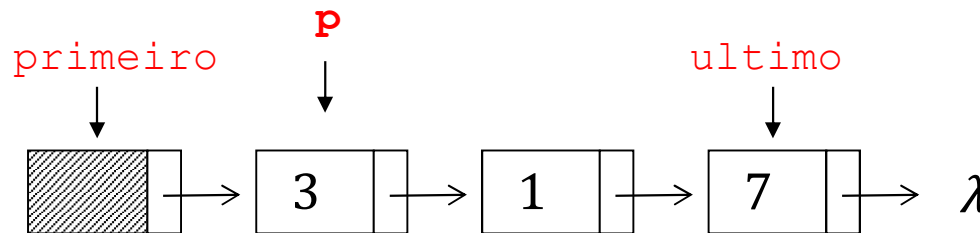
tamanho=3

```
ListaEncadeada L;  
...  
L.Limpa();
```

Class Lista Encadeada - Limpa

- “Limpa” a Lista
 - Percorre a lista desalocando memória

```
void ListaEncadeada::Limpa() {  
    TipoCelula *p;  
  
    p = primeiro->prox;  
    while (p!=NULL) {  
        primeiro->prox = p->prox;  
        delete p;  
        p = primeiro->prox;  
    }  
    ultimo = primeiro;  
    tamanho = 0;  
};
```



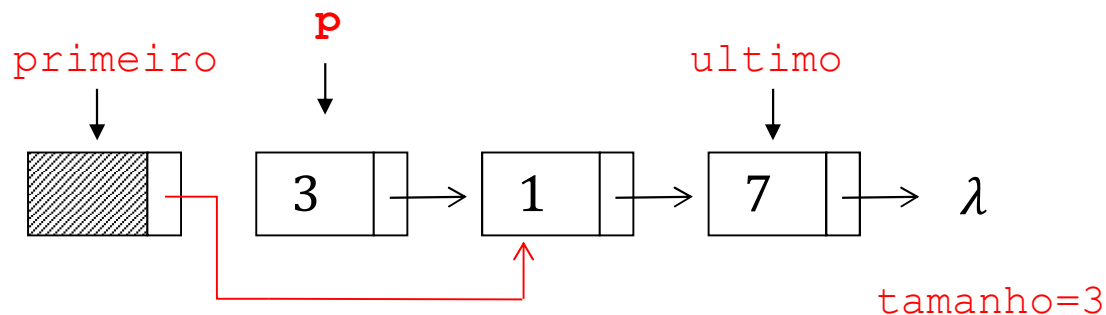
tamanho=3

```
ListaEncadeada L;  
...  
L.Limpa();
```


Class Lista Encadeada - Limpa

- “Limpa” a Lista
 - Percorre a lista desalocando memória

```
void ListaEncadeada::Limpa() {  
    TipoCelula *p;  
  
    p = primeiro->prox;  
    while (p!=NULL) {  
        primeiro->prox = p->prox;  
        delete p;  
        p = primeiro->prox;  
    }  
    ultimo = primeiro;  
    tamanho = 0;  
};
```

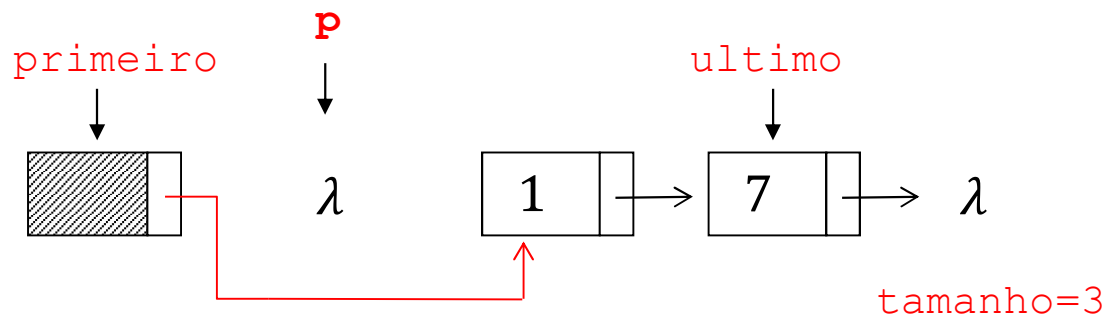


```
ListaEncadeada L;  
...  
L.Limpa();
```

Class Lista Encadeada - Limpa

- “Limpa” a Lista
 - Percorre a lista desalocando memória

```
void ListaEncadeada::Limpa() {  
    TipoCelula *p;  
  
    p = primeiro->prox;  
    while (p!=NULL) {  
        primeiro->prox = p->prox;  
        delete p;  
        p = primeiro->prox;  
    }  
    ultimo = primeiro;  
    tamanho = 0;  
};
```

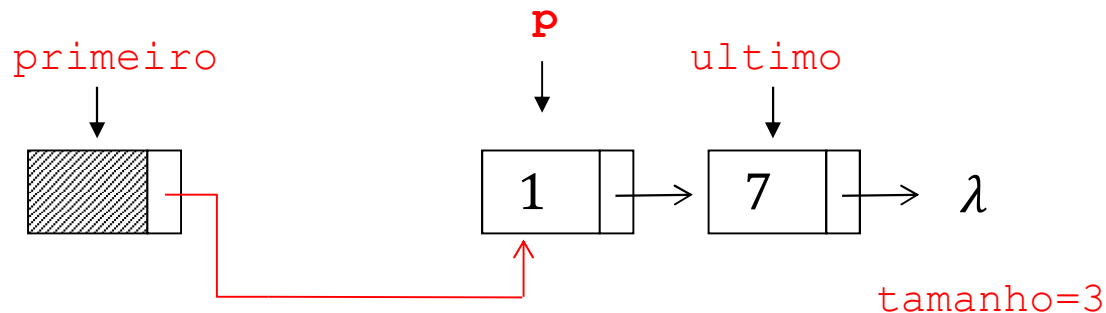


```
ListaEncadeada L;  
...  
L.Limpa();
```

Class Lista Encadeada - Limpa

- “Limpa” a Lista
 - Percorre a lista desalocando memória

```
void ListaEncadeada::Limpa() {  
    TipoCelula *p;  
  
    p = primeiro->prox;  
    while (p!=NULL) {  
        primeiro->prox = p->prox;  
        delete p;  
        p = primeiro->prox;  
    }  
    ultimo = primeiro;  
    tamanho = 0;  
};
```

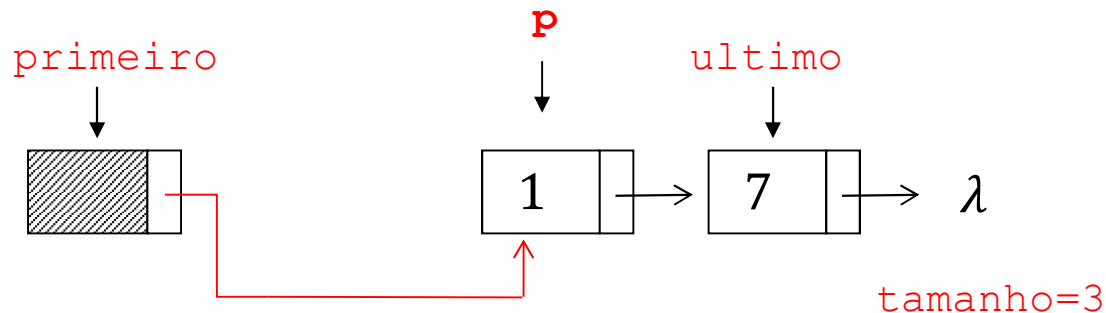


```
ListaEncadeada L;  
...  
L.Limpa();
```

Class Lista Encadeada - Limpa

- “Limpa” a Lista
 - Percorre a lista desalocando memória

```
void ListaEncadeada::Limpa() {  
    TipoCelula *p;  
  
    p = primeiro->prox;  
    while (p!=NULL) {  
        primeiro->prox = p->prox;  
        delete p;  
        p = primeiro->prox;  
    }  
    ultimo = primeiro;  
    tamanho = 0;  
};
```

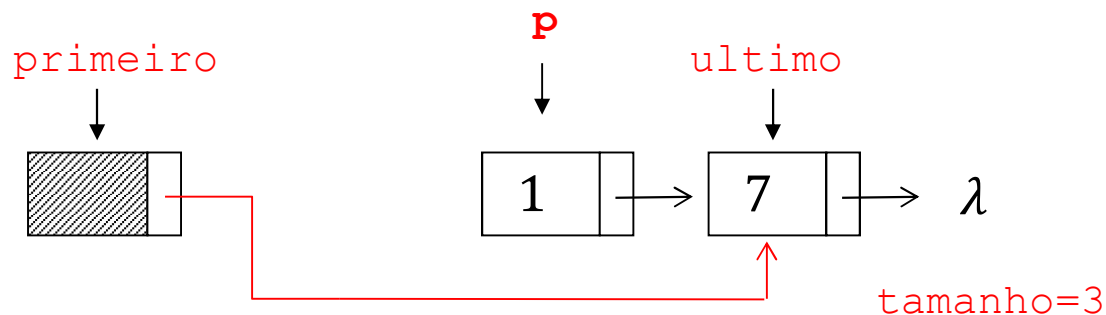


```
ListaEncadeada L;  
...  
L.Limpa();
```

Class Lista Encadeada - Limpa

- “Limpa” a Lista
 - Percorre a lista desalocando memória

```
void ListaEncadeada::Limpa() {  
    TipoCelula *p;  
  
    p = primeiro->prox;  
    while (p!=NULL) {  
        primeiro->prox = p->prox;  
        delete p;  
        p = primeiro->prox;  
    }  
    ultimo = primeiro;  
    tamanho = 0;  
};
```

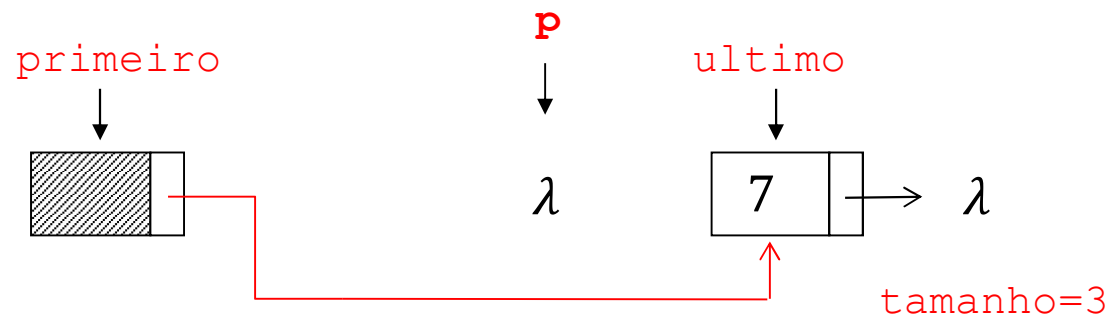


```
ListaEncadeada L;  
...  
L.Limpa();
```

Class Lista Encadeada - Limpa

- “Limpa” a Lista
 - Percorre a lista desalocando memória

```
void ListaEncadeada::Limpa() {  
    TipoCelula *p;  
  
    p = primeiro->prox;  
    while (p!=NULL) {  
        primeiro->prox = p->prox;  
        delete p;  
        p = primeiro->prox;  
    }  
    ultimo = primeiro;  
    tamanho = 0;  
};
```

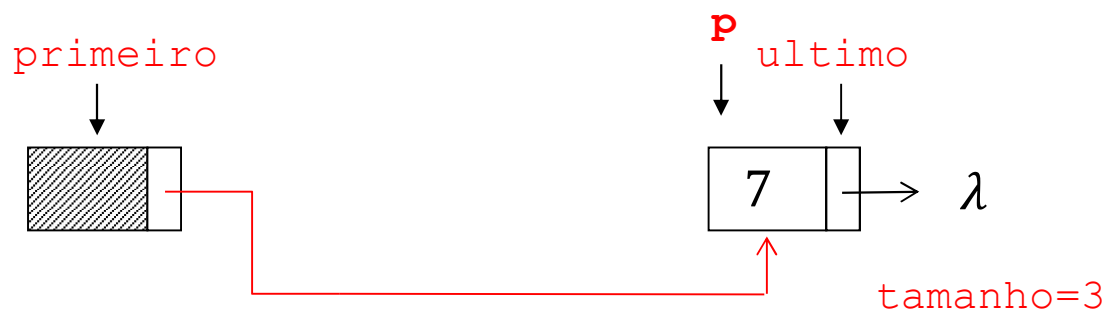


```
ListaEncadeada L;  
...  
L.Limpa();
```

Class Lista Encadeada - Limpa

- “Limpa” a Lista
 - Percorre a lista desalocando memória

```
void ListaEncadeada::Limpa() {  
    TipoCelula *p;  
  
    p = primeiro->prox;  
    while (p!=NULL) {  
        primeiro->prox = p->prox;  
        delete p;  
        p = primeiro->prox;  
    }  
    ultimo = primeiro;  
    tamanho = 0;  
};
```

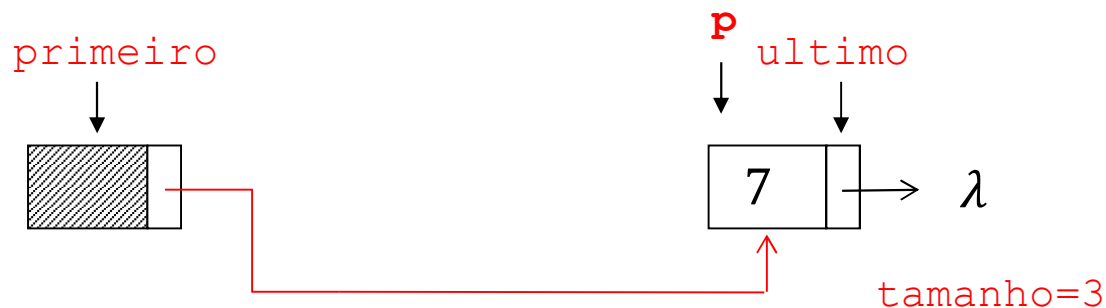


```
ListaEncadeada L;  
...  
L.Limpa();
```

Class Lista Encadeada - Limpa

- “Limpa” a Lista
 - Percorre a lista desalocando memória

```
void ListaEncadeada::Limpa() {  
    TipoCelula *p;  
  
    p = primeiro->prox;  
    while (p!=NULL) {  
        primeiro->prox = p->prox;  
        delete p;  
        p = primeiro->prox;  
    }  
    ultimo = primeiro;  
    tamanho = 0;  
};
```

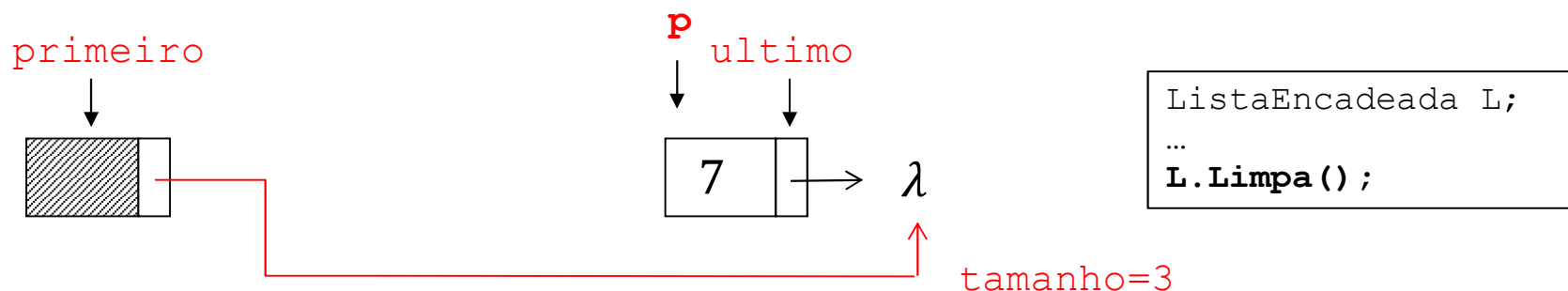


```
ListaEncadeada L;  
...  
L.Limpa();
```


Class Lista Encadeada - Limpa

- “Limpa” a Lista
 - Percorre a lista desalocando memória

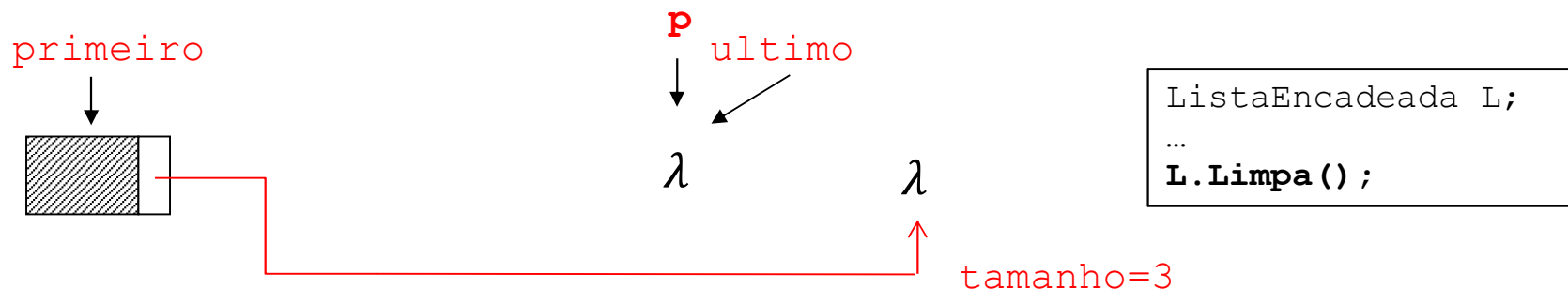
```
void ListaEncadeada::Limpa() {  
    TipoCelula *p;  
  
    p = primeiro->prox;  
    while (p!=NULL) {  
        primeiro->prox = p->prox;  
        delete p;  
        p = primeiro->prox;  
    }  
    ultimo = primeiro;  
    tamanho = 0;  
};
```



Class Lista Encadeada - Limpa

- “Limpa” a Lista
 - Percorre a lista desalocando memória

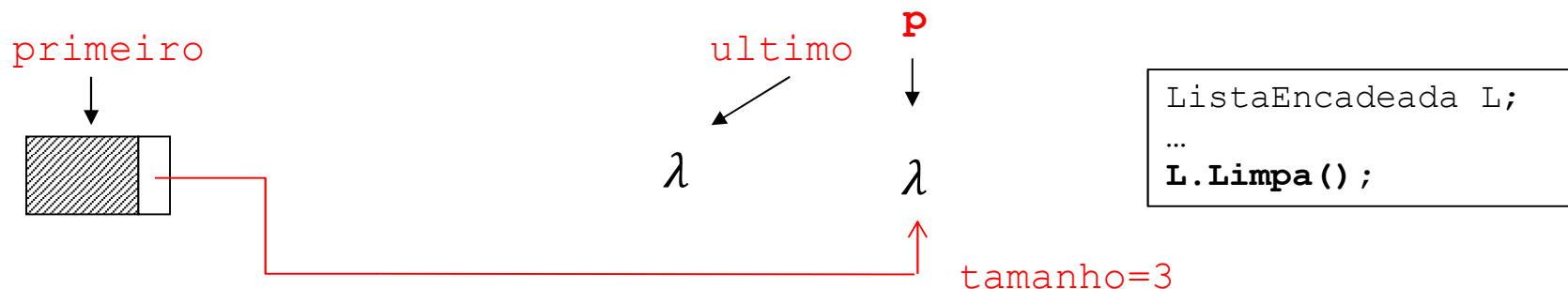
```
void ListaEncadeada::Limpa() {  
    TipoCelula *p;  
  
    p = primeiro->prox;  
    while (p!=NULL) {  
        primeiro->prox = p->prox;  
        delete p;  
        p = primeiro->prox;  
    }  
    ultimo = primeiro;  
    tamanho = 0;  
};
```



Class Lista Encadeada - Limpa

- “Limpa” a Lista
 - Percorre a lista desalocando memória

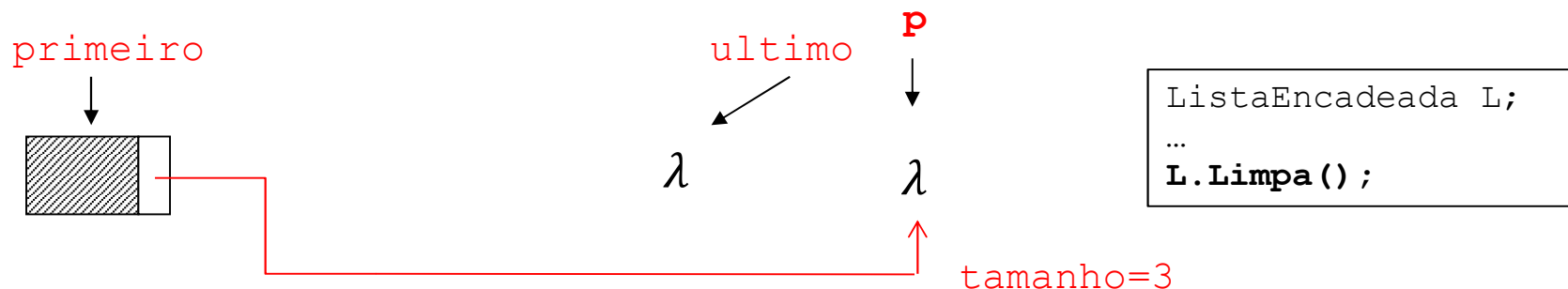
```
void ListaEncadeada::Limpa() {  
    TipoCelula *p;  
  
    p = primeiro->prox;  
    while (p!=NULL) {  
        primeiro->prox = p->prox;  
        delete p;  
        p = primeiro->prox;  
    }  
    ultimo = primeiro;  
    tamanho = 0;  
};
```



Class Lista Encadeada - Limpa

- “Limpa” a Lista
 - Percorre a lista desalocando memória

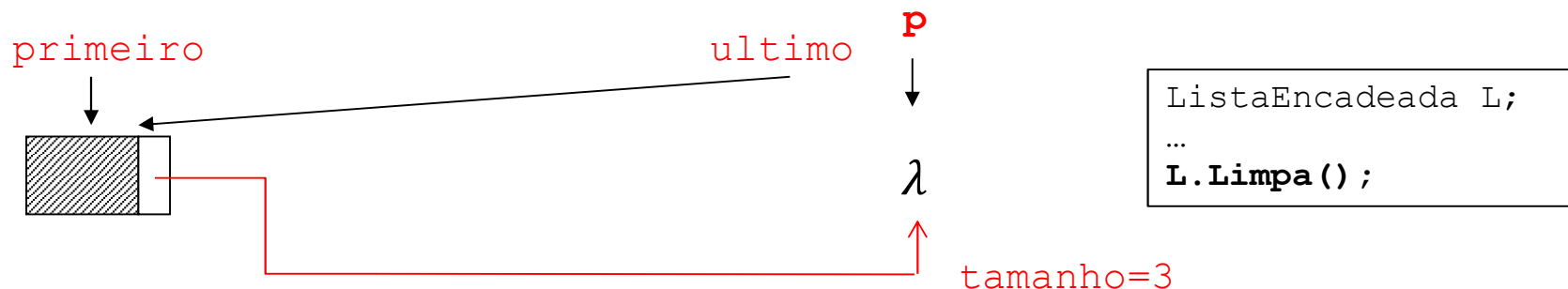
```
void ListaEncadeada::Limpa() {  
    TipoCelula *p;  
  
    p = primeiro->prox;  
    while (p!=NULL) {  
        primeiro->prox = p->prox;  
        delete p;  
        p = primeiro->prox;  
    }  
    ultimo = primeiro;  
    tamanho = 0;  
};
```



Class Lista Encadeada - Limpa

- “Limpa” a Lista
 - Percorre a lista desalocando memória

```
void ListaEncadeada::Limpa() {  
    TipoCelula *p;  
  
    p = primeiro->prox;  
    while (p!=NULL) {  
        primeiro->prox = p->prox;  
        delete p;  
        p = primeiro->prox;  
    }  
    ultimo = primeiro;  
    tamanho = 0;  
};
```

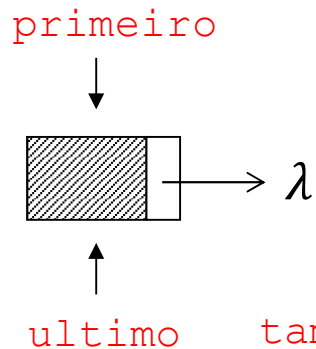


Class Lista Encadeada - Limpa

- “Limpa” a Lista
 - Percorre a lista desalocando memória

```
void ListaEncadeada::Limpa() {  
    TipoCelula *p;  
  
    p = primeiro->prox;  
    while (p!=NULL) {  
        primeiro->prox = p->prox;  
        delete p;  
        p = primeiro->prox;  
    }  
    ultimo = primeiro;  
    tamanho = 0;  
};
```

$O(n)$



```
ListaEncadeada L;  
...  
L.Limpa();
```

Alocação Encadeada

■ Vantagens:

- ❑ Tamanho da lista em memória é dinâmico
 - Bom para aplicações onde a previsão do tamanho não pode ser feita a priori
- ❑ Inserção e Remoção não requer o deslocamento de itens

■ Desvantagens

- ❑ Acesso a itens requer caminhar na lista
- ❑ Memória extra para armazenar os apontadores
- ❑ Código mais complexo

Alocação Sequencial x Encadeada

	Sequencial	Encadeada
Acesso a um Item	O(1)	O(n)
Inserção / Remoção	O(n)	O(1)*
Tamanho	Fixo	Dinâmico
Memória Extra	Não	Sim
Implementação Simples	Sim	Não

* Pode ser necessário posicionar um apontador auxiliar antes