FaculdadeBarretos
www.faculdadebarretos.com.br

Algoritmos e Estruturas de Dados

(Aula 9 - Listas Encadeadas I Estrutura e Implementação)

Prof. Me. Diogo Tavares da Silva
contato: *diogotavares@unibarretos.com.br*

FaculdadeBarretos
www.faculdadebarretos.com.br

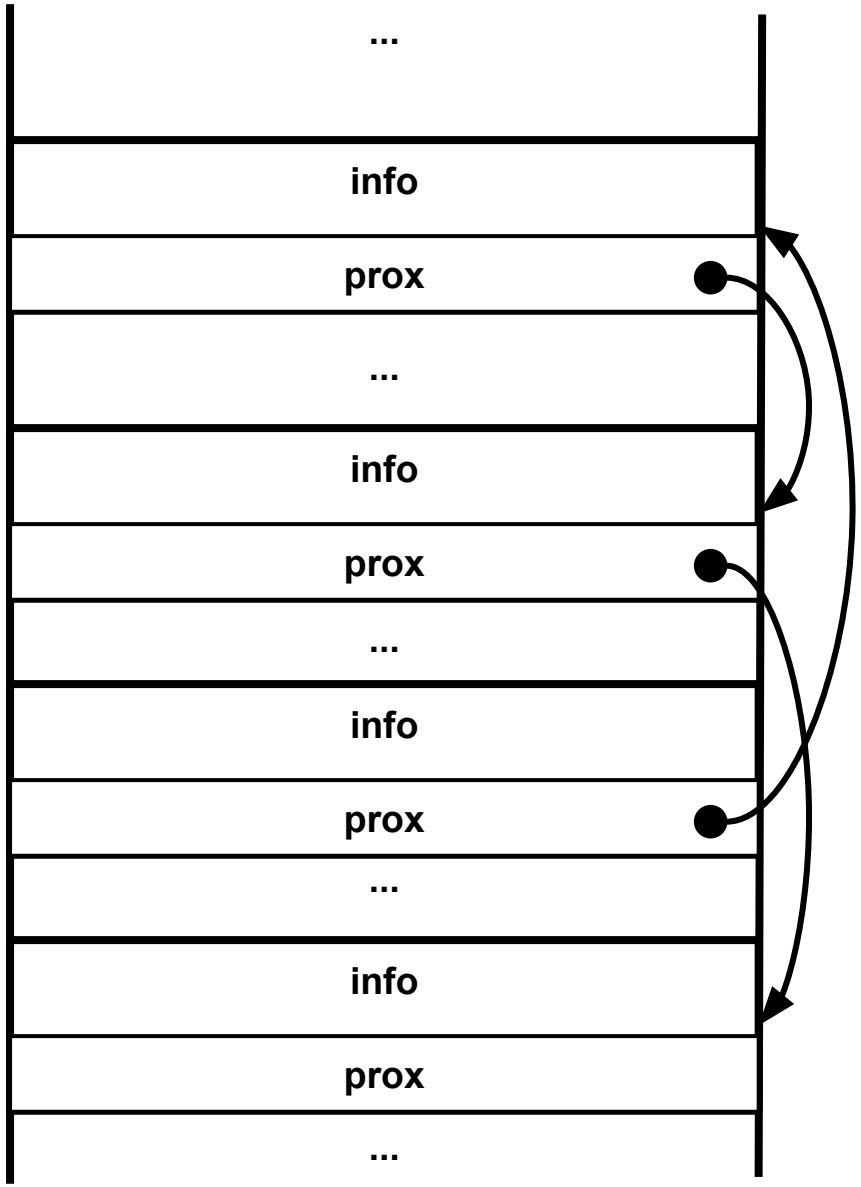
Nas últimas aulas aprendemos...

- Uma visão geral de como criar uma estrutura encadeada que cresce nó a nó e que pode ser espalhada na memória ao invés de contígua.
  - Lista encadeada
  - suas variações e aplicações

Lista encadeada

- Hoje aprenderemos a projetar uma lista simplesmente encadeada.
- Projeto de biblioteca de lista encadeada
  - projetar sua estrutura
  - projetar as operações sobre ela
    - criação
    - percurso
    - inserção
    - busca
    - remoção

A idéia de listas encadeadas



Memória principal

```
struct nodo{
   int info;
   struct nodo *prox;
};

typedef struct nodo Nodo;
```

Criação do tipo lista

- A partir da criação do elemento unitário (**nodo**) criaremos a lista
  - Referenciamos a lista por meio de um ponteiro para o primeiro elemento (assim como o vetor)
    - Mantemos sempre esse ponteiro apontado para o primeiro elemento (Para não perder o início da lista)
      - usaremos ponteiros auxiliares quando precisarmos percorrê-la

```
Nodo *lista;
```

Criação da biblioteca de lista

- Deste modo criaremos:

Arquivo em que estarão implementadas a estrutura de nós e todas as funções de criação, acesso e manipulação da lista.

Arquivo de *header* contendo as interfaces (protótipos) das funções e com as interfaces de dados (typedef) contidas em **lista.cpp**

Arquivo principal que irá importar nossa implementação de lista (**#include "lista.h"**) sempre que necessário

Projeto do tipo Lista

- Feito a criação dos arquivos para implementar nossa biblioteca, vamos criar primeiro nosso tipo de dado **Nodo** como discutido anteriormente:

#no arquivo lista.cpp

```
#include <iostream>
using namespace std;

struct nodo{
    int info;
    struct nodo *prox;
};

typedef struct nodo Nodo;
```

Projeto do tipo Lista

```
#no arquivo lista.h
    ...
    typedef struct nodo Nodo;
    …
```

Projeto do tipo Lista

- A partir deste ponto **TUDO** é decisão de projeto
  - Vai depender de como quer implementar sua estrutura
  - Que funcionalidades quer inserir
  - Como quer que as funcionalidades se comportem
    - Importante sempre documentar para que seu usuário final da biblioteca saiba como utilizá-la

Projeto do tipo Lista

- Primeiro vamos pensar nas funcionalidades básicas para uma estrutura de dados:
  - **Criação**
  - **Inserção de elementos**
  - **Busca por elementos**
  - **Remoção de elementos**
  - **Percurso**

Criação do tipo lista

- Como vamos criar nossa lista inicialmente?
  - Vazia?
  - Com algum elemento?
- Mais plausível iniciá-la vazia e ir inserindo elementos conforme necessário
- deste modo:

Criação do tipo lista

*//no arquivo **lista.cpp***

```
/**
FUNÇÃO: criaLista
PARAM: nenhum
RETORNO: Nodo* (ponteiro para onde lista começa apontando)
DESC: Lista começa vazia apontando pra NULL
**/
Nodo* criaLista(){
    cout << "Lista criada com sucesso!" << endl;
    return NULL;
}
```

Criação do tipo lista

```
//no arquivo lista.h
...
Nodo* criaLista();
...
```

```
//no arquivo main.c
...
main(){
   Nodo* lista;
   lista = criaLista();
}
```

Função de teste lista*Vazia()*

- Como vimos para vetores, a forma como procedemos para inserir, buscar e remover depende se a estrutura contém elementos ou está vazia.
- Por essa razão, é conveniente criar uma função que testa se a lista está vazia
  - **int listaVazia(Node* lista)**
    - **retorna 1, se lista vazia**
    - **retorna 0, caso contrário**

Criação da função *vazia()*

*//no arquivo **lista.cpp***

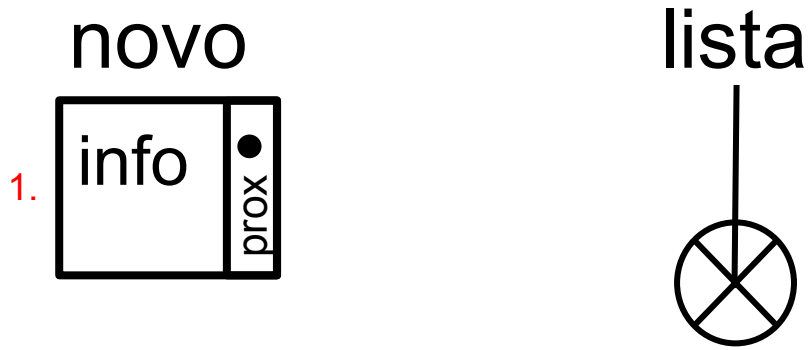
```
/**
FUNÇÃO: listaVazia
PARAM: Nodo* lista (lista a ser testada)
RETORNO: int (retorno do teste)
DESC: função retorna '1' se lista vazia, e '0' caso contrário
**/
int listaVazia(Nodo *lista){
    if (!lista)
        return 1;
    else return 0;
}
```

Inserção na lista

- Três situações de inserção:
  - No início
    - mais simples
  - No final
    - precisa percorrer a lista toda nó por nó
  - No meio
    - percorrer até onde deseja inserir segundo algum critério

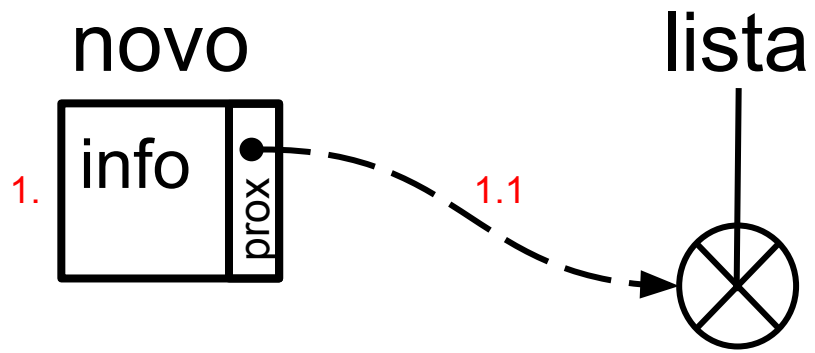
Inserção no início:

- Se a lista estiver vazia:
  - 1. aloca o novo nó



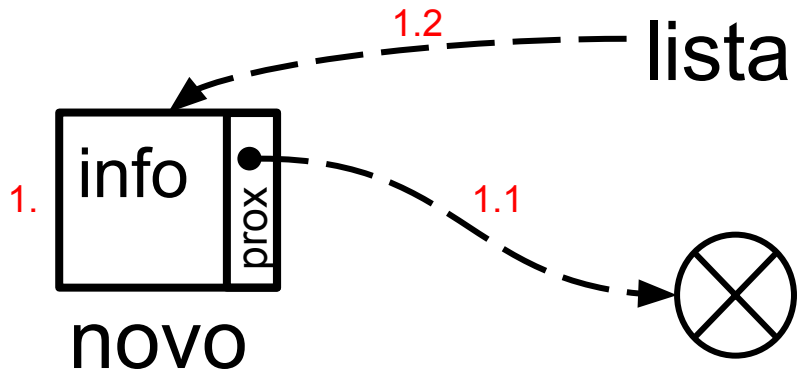
Inserção no início:

- Se a lista estiver vazia:
  - 1. aloca o novo nó
    - novo->prox = lista;



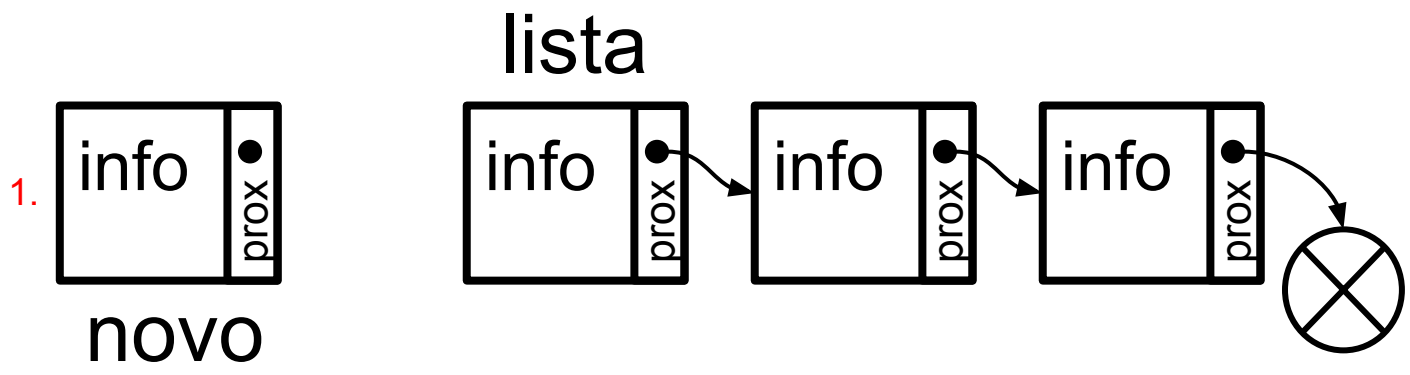
Inserção no início:

- Se a lista estiver vazia:
  - 1. aloca o novo nó
    - novo->prox = lista;
    - lista = novo;



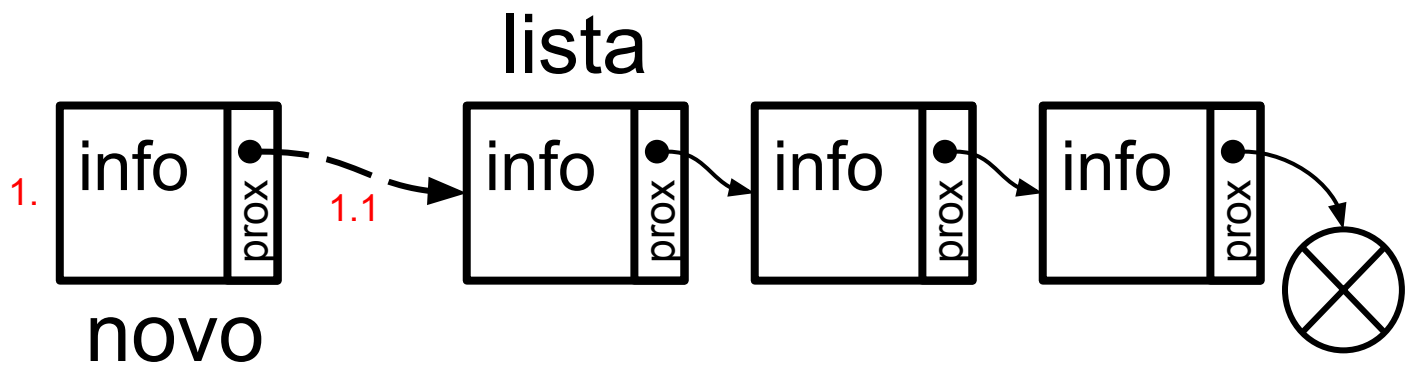
Inserção no início:

- Se a lista tiver elementos:
  - 1. aloca o novo nó



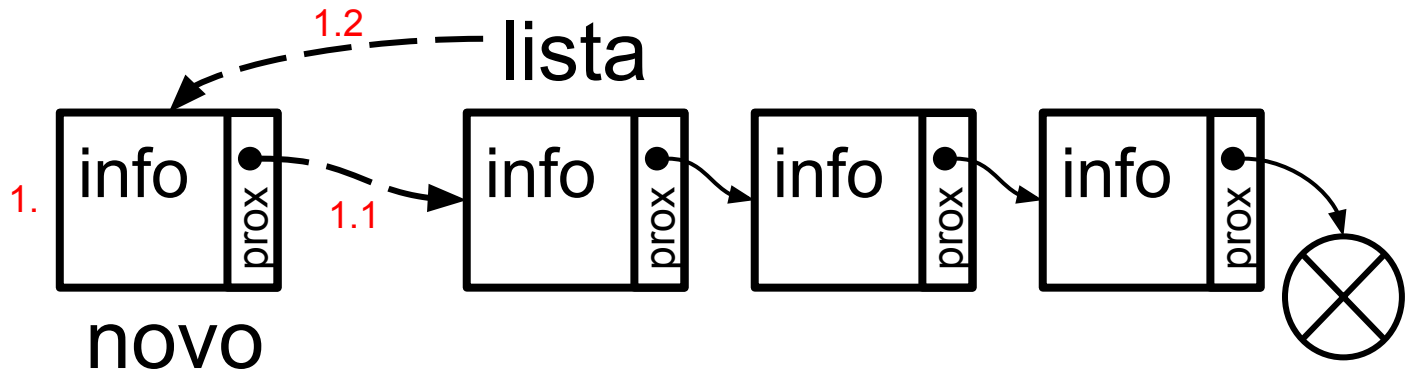
Inserção no início:

- Se a lista tiver elementos:
  - 1. aloca o novo nó
    - 1.1 novo->prox = lista



Inserção no início:

- Se a lista tiver elementos:
  - 1. aloca o novo nó
    - 1.1 novo->prox = lista
    - 1.2 lista = novo



Inserção no início da lista

//no arquivo ***lista.c***

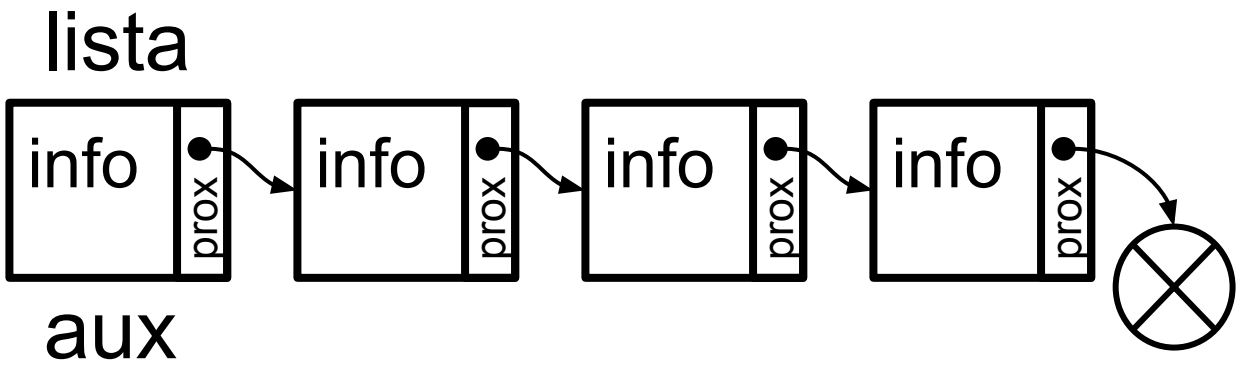
```
/**
FUNÇÃO: insereIni
PARAM: Nodo* lista, int elem (lista e elemento a ser inserido)
RETORNO: Nodo* (ponteiro para o início da lista)
DESC: insere um elemento no início da lista
**/
Nodo* insereIni(Nodo* lista, int elem){
    //criação do nó
    Nodo *novo;
    novo = new Nodo;
    novo->info = elem; //campo info recebe elem
    //se a lista é vazia ou com elementos tanto faz
    novo->prox = lista;
    return novo; //equivalente a lista = novo
}
```

Percurso na lista

- Para percorrer uma lista até o fim sempre devemos utilizar um ponteiro auxiliar
  - preservar o ponteiro que aponta para o início
- Para isso vamos criar uma função que **imprime** os elementos da lista
  - Se vazia
    - Não faz nada
  - Senão
    - percorre imprimindo os elementos

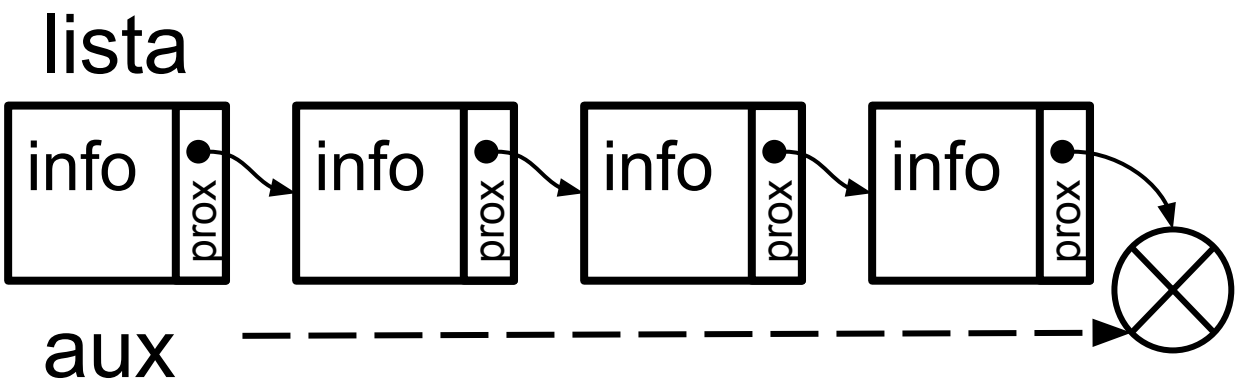
Percorrendo a lista

- Para percorrer a lista:
  - 1. Cria o ponteiro auxiliar e aponta para **lista**



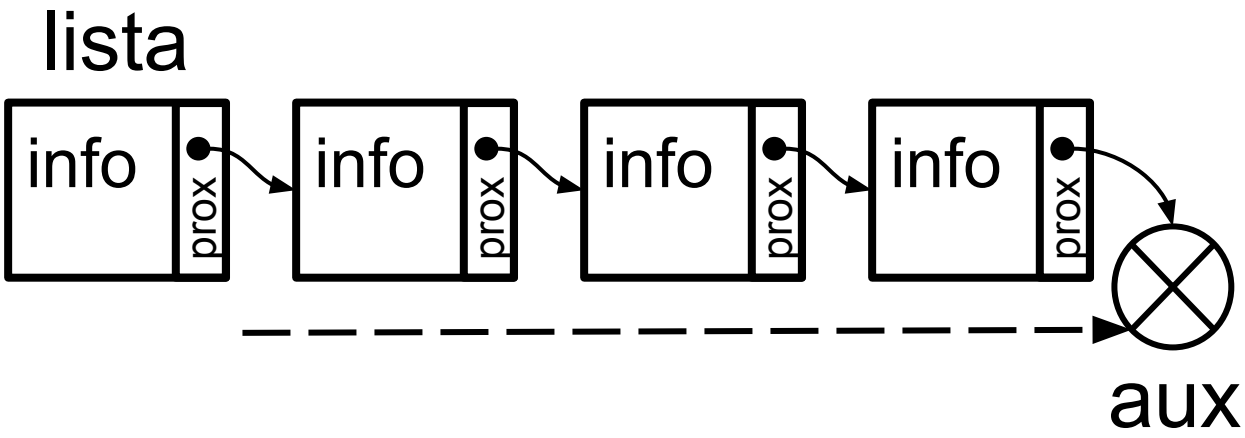
Percorrendo a lista

- Para percorrer a lista:
  - 1. Cria o ponteiro auxiliar e aponta para **lista**
    - Enquanto aux não for NULL
      - imprime aux->info;
      - aux = aux->prox;



Percorrendo a lista

- Para percorrer a lista:
  - 1. Cria o ponteiro auxiliar e aponta para **lista**
    - Enquanto aux não for NULL
      - imprime aux->info;
      - aux = aux->prox;



Percorrer e imprimir a lista

//no arquivo ***lista.cpp***

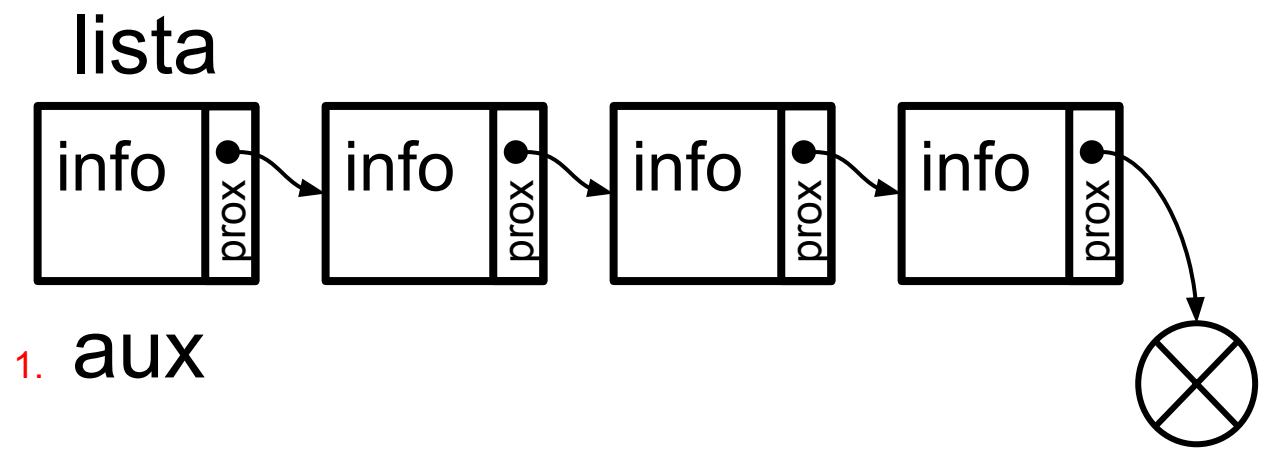
```
/**
FUNÇÃO: printLista
PARAM: Nodo* lista (lista  a imprimir)
RETORNO: void
DESC: imprime o conteúdo da lista
**/
void printLista(Nodo* lista){
    if(!listaVazia(lista)){
        Nodo* aux = lista; //cria e aponta pra lista
        while(aux){ //enquanto lista != NULL
            cout << "( " << aux->info << " ) -> ";  //imprime
            aux=aux->prox; //percorre até o fim
        }
        cout << "|X|" << endl; //imprimir o fim da lista bunitim
    }else{
        cout << "lista vazia!" << endl;
    }
}
```

Remoção da lista

- Três situações de remoção:
  - No início
    - mais simples
  - No final
    - percorrer até o penúltimo elemento
  - No meio
    - percorrer até o elemento que deseja remover segundo algum critério

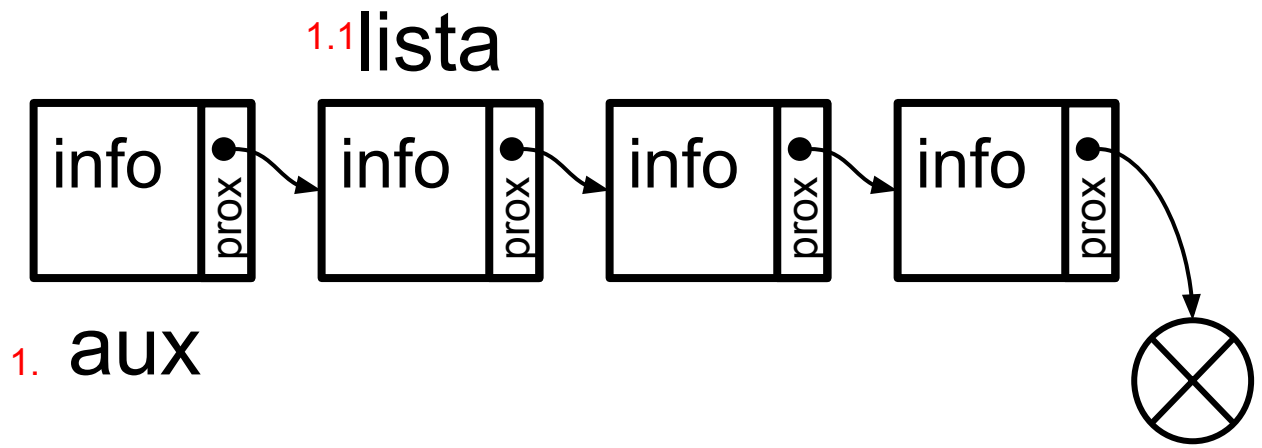
Remoção no início

- Se a lista está vazia
  - Não há o que remover!
- Se a lista possui elementos
  - 1. aux = lista



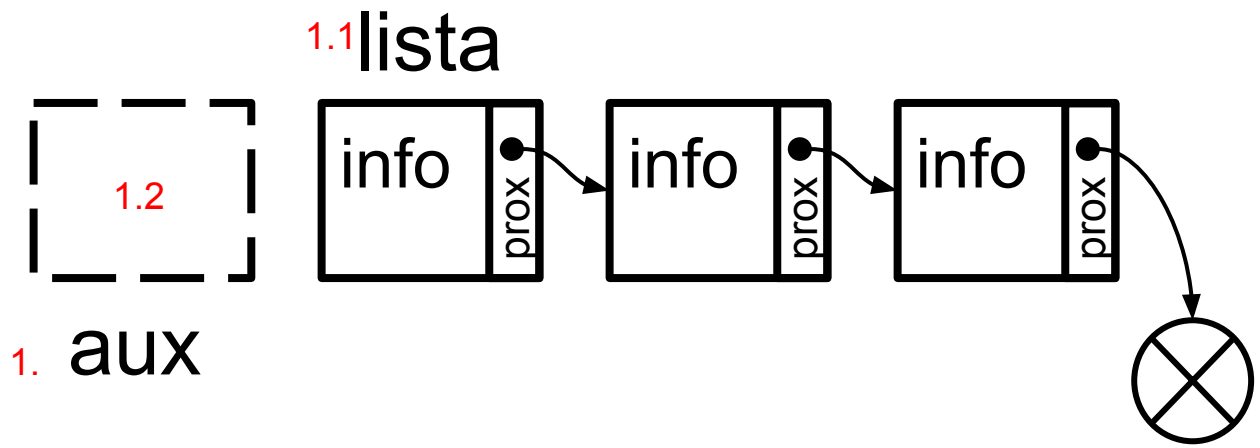
Remoção no início

- Se a lista está vazia
  - Não há o que remover!
- Se a lista possui elementos
  - 1. aux = lista
  - 1.1 lista = lista->prox



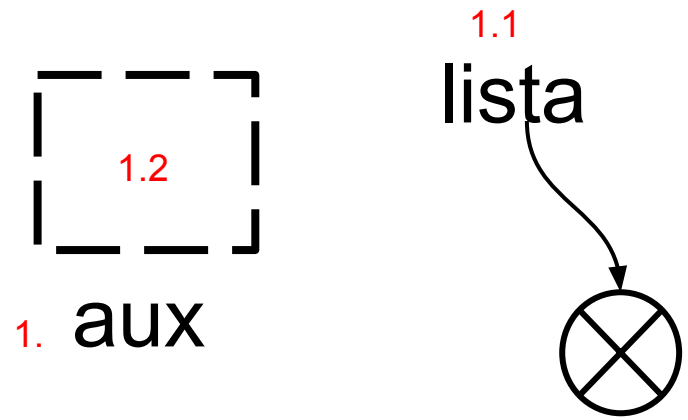
Remoção no início

- Se a lista está vazia
  - Não há o que remover!
- Se a lista possui elementos
  - 1. aux = lista
  - 1.1 lista = lista->prox
  - 1.2 delete aux



Remoção no início

- Se a lista está vazia
  - Não há o que remover!
- Se a lista possui elementos
  - 1. aux = lista
  - 1.1 lista = lista->prox
  - 1.2 delete aux



** Se a lista tiver um elemento só, algoritmo é o mesmo, vai esvaziar e lista ficara nula. **

Remoção no início

//no arquivo ***lista.cpp***

```
/**
FUNÇÃO: removeIni
PARAM: Nodo* lista (lista a se remover a cabeça)
RETORNO: Nodo* (lista atualizada)
DESC: remove o primeiro elemento da lista
**/
Nodo* removeIni(Nodo* lista){
    if(!listaVazia(lista)){
        Nodo* aux = lista;
        lista = lista-> prox; //aponta a cabeça pro próximo
        delete aux; //libera o primeiro nó
    }else{
        cout << "A lista já se encontra vazia!" << endl;
    }
    return lista;
}
```

Remoção no início

*//no arquivo **lista.h** até o momento..*

```
#ifndef LISTA_H
#define LISTA_H

typedef struct nodo Nodo;
Nodo* criaLista();
int listaVazia(Nodo *lista);
Nodo* insereIni(Nodo* lista, int elem);
void printLista(Nodo* lista);
Nodo* removeIni(Nodo* lista);



#endif
```

Remoção no início

//exemplo de função main importando a biblioteca

```
#include <iostream>
#include "lista.h"
using namespace std;

int main()
{    Nodo *lista;
     //criação
     lista = criaLista();
     //teste vazio
     if (listaVazia(lista))
          cout << "A lista esta vazia!" << endl;
     //inserções no início
     lista = insereIni(lista,7);
     lista = insereIni(lista,9);
     lista = insereIni(lista,12);
     //percurso com impressão
     printLista(lista);
     //remoção no início
     lista = removeIni(lista);
     printLista(lista);
}
```

Continuação

Nesta aula analisaremos:

- Inserção no final
- Remoção do final
- Inserção no meio (inserir ordenado)
- Buscar elemento
- Remover no meio (remover elemento pós busca)

Inserção no final:

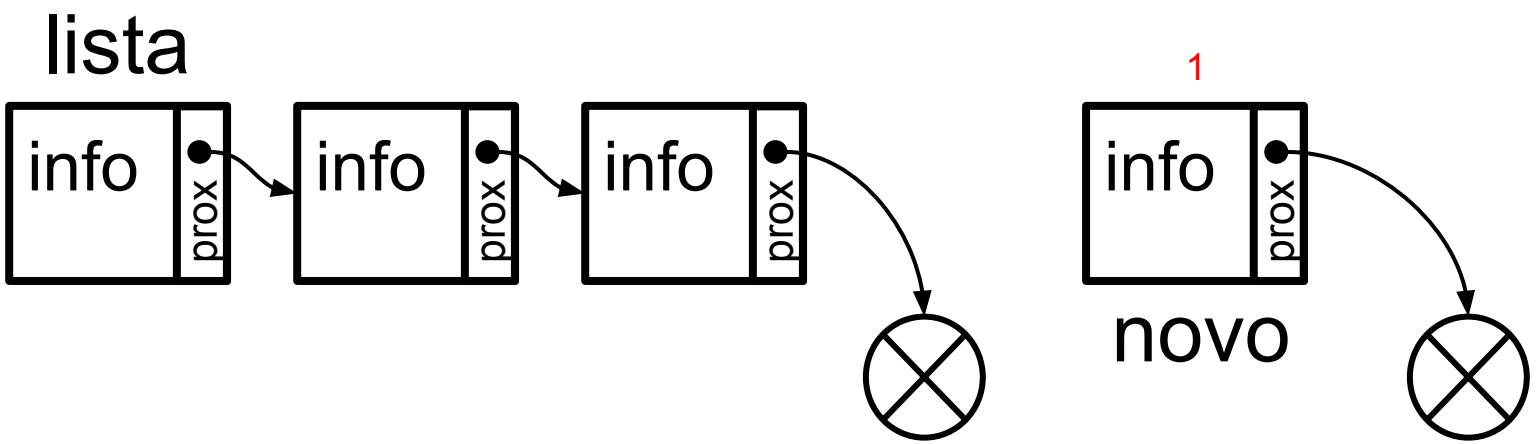
- Se a lista estiver vazia:
  - Mesmo modo que no início
    - aloca o novo nó e atualiza o ponteiro da cabeça para ele
    - return novo

Inserção no final:

- Se a lista tiver elementos:
  - Precisamos chegar até o último elemento!
  - Se usarmos while(aux)?
    - Saímos da lista
  - Temos que usar
    - **while(aux->prox)**
      - Enquanto o ponteiro auxiliar tiver um próximo percorremos a lista!
      - Pararemos no último elemento

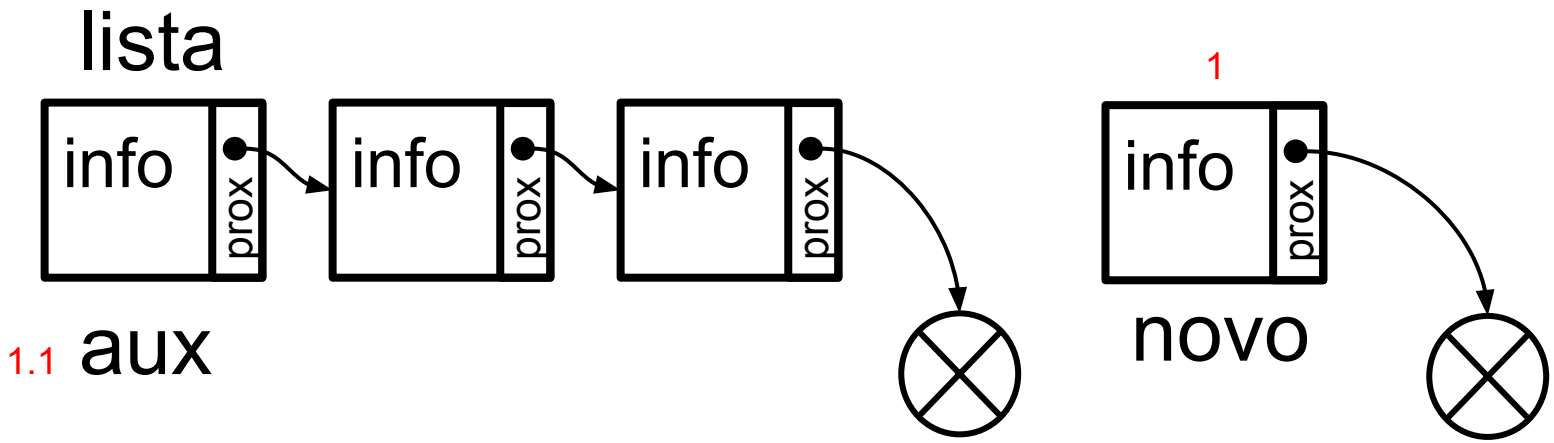
Inserção no final:

- deste modo:
  - 1. alocamos o novo nó



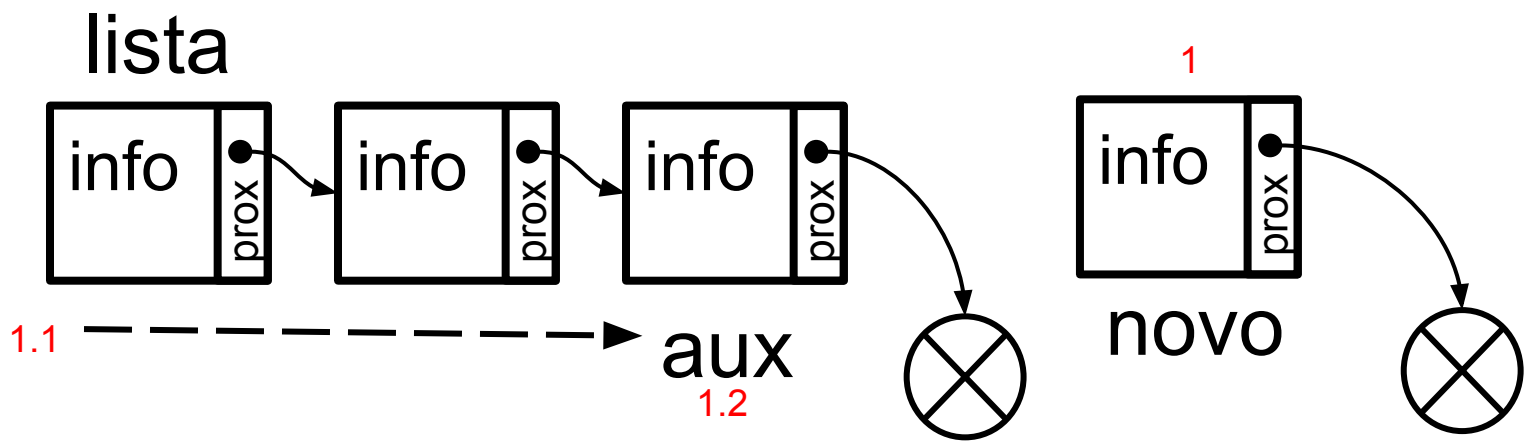
Inserção no final:

- deste modo:
  - 1. alocamos o novo nó
  - 1.1. criamos ponteiro **aux** e apontamos pra **lista**



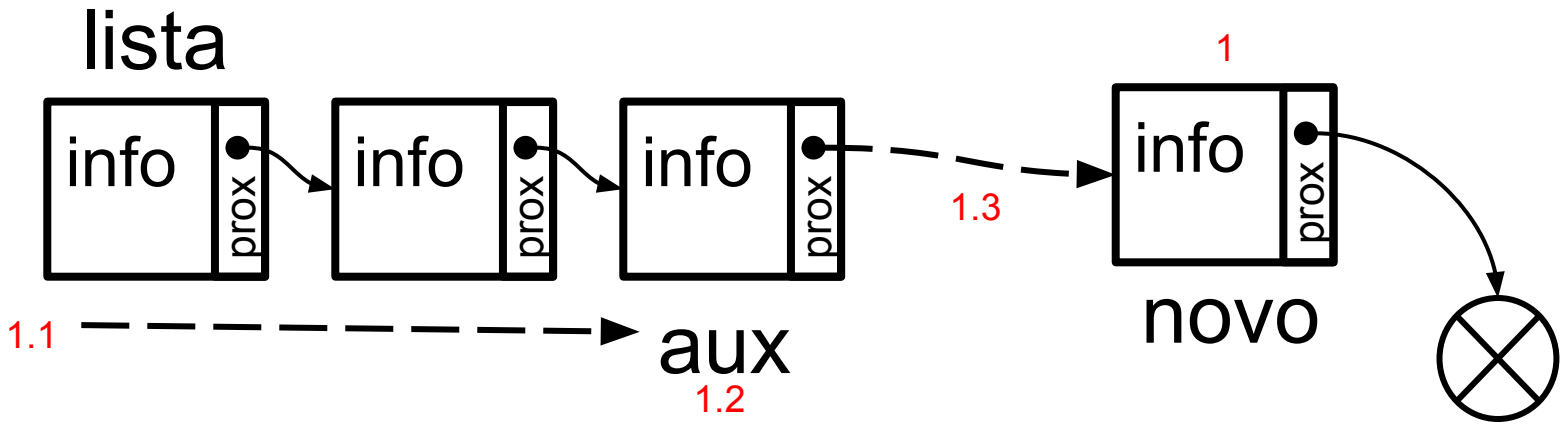
Inserção no final:

- deste modo:
  - 1. alocamos o novo nó
  - 1.1. criamos ponteiro **aux** e apontamos pra **lista**
  - 1.2. Enquanto existir **aux->prox**
    - **aux = aux->prox** (vai parar no ultimo elemento)



Inserção no final:

- deste modo:
  - 1. alocamos o novo nó
  - 1.1. criamos ponteiro **aux** e apontamos pra **lista**
  - 1.2. Enquanto existir **aux->prox**
    - **aux = aux->prox** (vai parar no ultimo elemento)
  - 1.3. aux->prox = novo



Inserção no final:

//no arquivo lista.cpp

```
Nodo* insereFim(Nodo* lista, int elem){
    //cria novo nodo
    Nodo* novo = new Nodo;
    novo ->info = elem;
    novo->prox = NULL;
    if(!listaVazia(lista)){
        Nodo *aux = lista;
        //percorrer até o último elemento
        while(aux->prox){
            aux = aux->prox;
        }
        //inserir o novo ultimo
        aux -> prox = novo;
    }else{
        //análogo a inserir no começo
        lista = novo;
    }
    return lista;
}
```

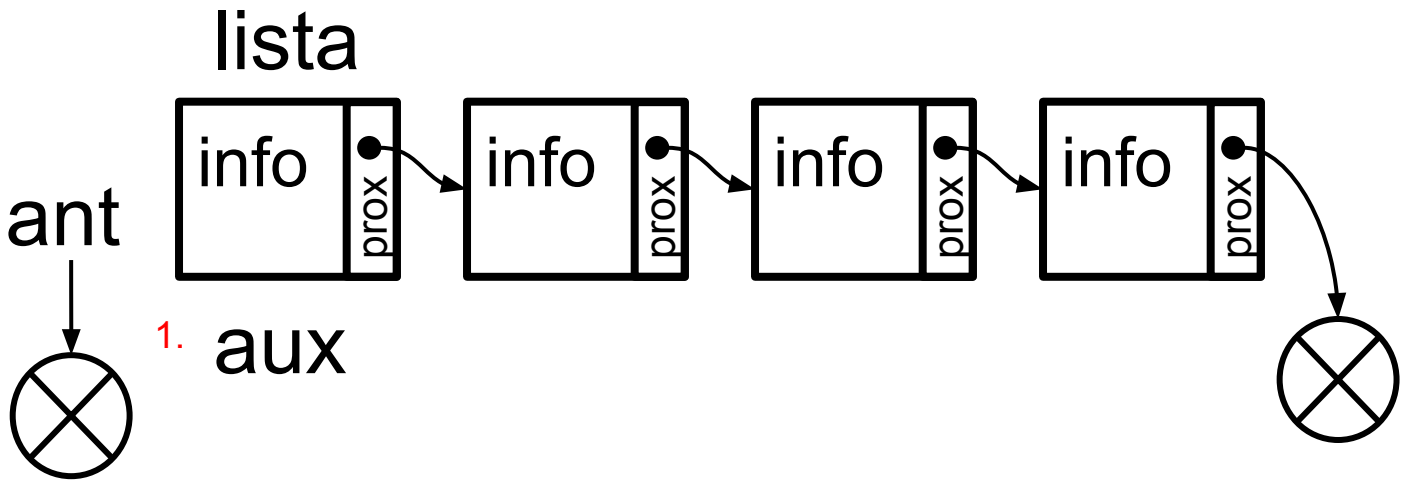
Remoção no final

- Precisamos chegar ao último elemento
  - percorremos a lista como na inserção no final
  - PORÉM…
    - precisamos atualizar a referência do valor de “prox” do penúltimo pra NULL
      - pois ele se tornará o último
      - usar um ponteiro “anterior”
        - aponta para um antes do auxiliar
    - Se a lista só tem um elemento precisamos atualizar a referência da cabeça da lista para NULL

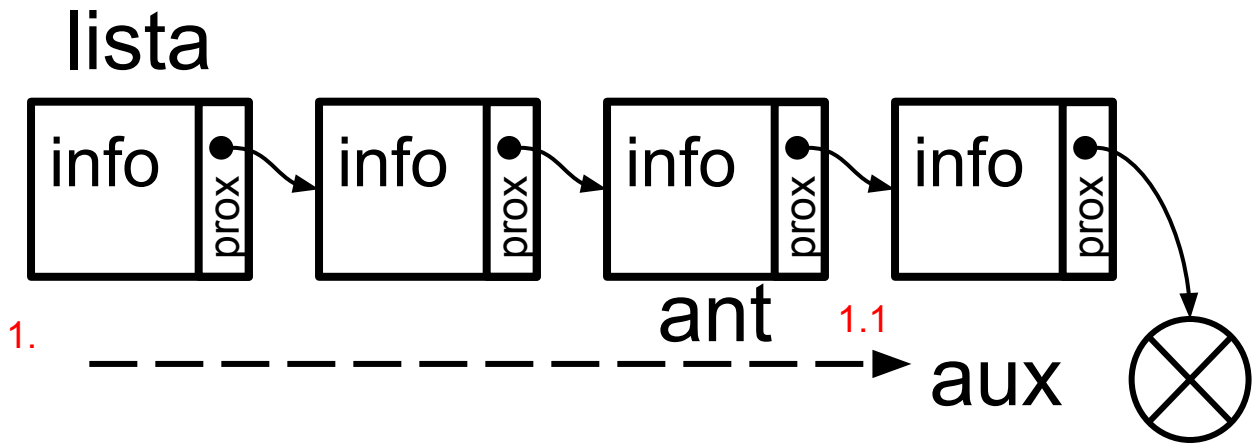
Remoção no final

- Se lista não está vazia
  - 1.aux = lista e ant = NULL
  - 1.1 Percorrer até o final
    - Enquanto (aux->prox)
      - ant = aux
      - aux = aux->prox
  - 1.2 Testa se a lista só tem um elemento
    - 1.2.1 se aux != lista (tem mais elementos)
      - ant->prox = NULL
    - 1.2.2 Senão (se só tem um elemento)
      - lista = NULL
  - 1.3 libera aux
    - delete aux

Remoção no final

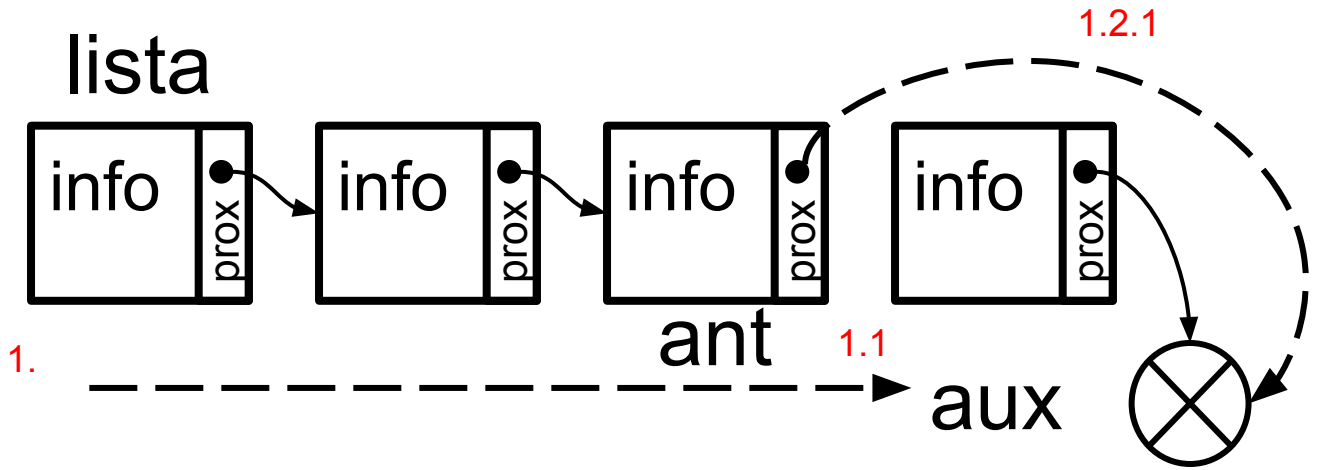


Remoção no final



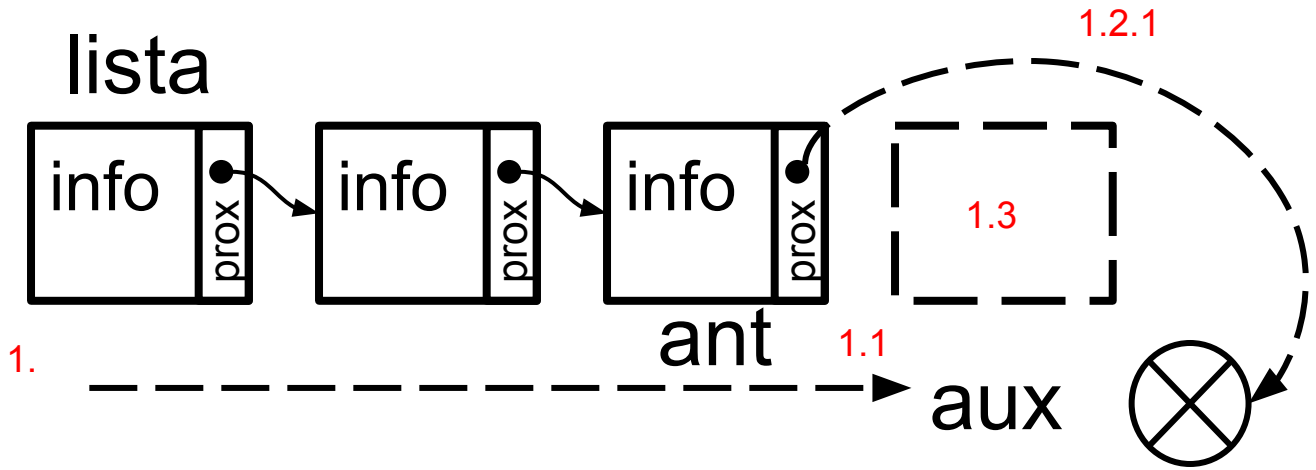
Remoção no final

Situação 1: Lista tem mais de um elemento



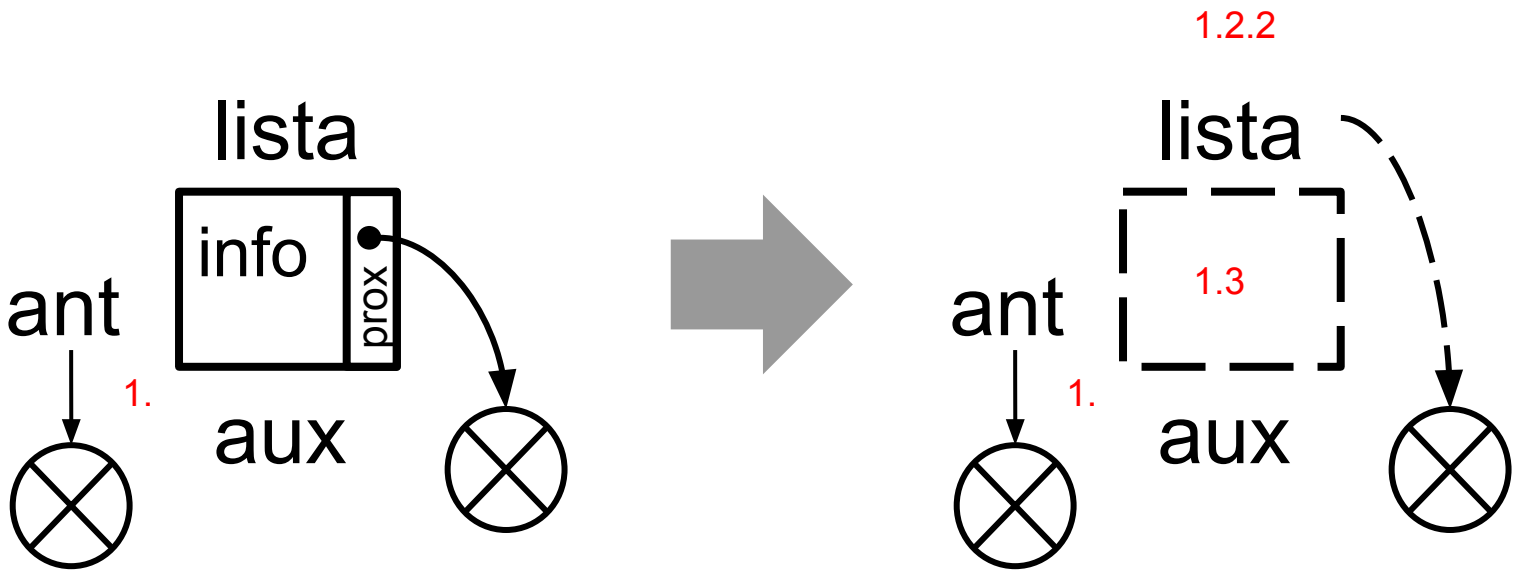
Remoção no final

Situação 2: Lista tem mais de um elemento



Remoção no final

Situação 2: Lista só tem um elemento



Remoção no final

// no arquivo lista.cpp

```
Nodo* removeFim(Nodo* lista){
    if(!listaVazia(lista)){
        Nodo *ant = NULL; //aponta pra um antes de aux
        Nodo *aux = lista; //percorre a lista
        //percorre até chegar no último e penúltimo
        while(aux->prox){
            ant = aux;
            aux = aux->prox;
        }
        //se lista tem mais de um elemento
        if (aux != lista){
            ant->prox = NULL; //novo final
        }else{
            lista = NULL;
        }
        delete aux;

    }else{
        cout << "A lista se encontra vazia!";
    }
    return lista;
}
```

Inserção no meio da lista:

- Como discutimos, a inserção no meio da lista (entre elementos) sempre dependerá de um critério de inserção
  - “Inserir no meio”
    - onde exatamente
- Vamos usar como exemplo a **inserção ordenada**
  - Inserir os elementos de modo que eles fiquem em ordem crescente de valor segundo algum campo
    - No nosso caso o campo **.info**

Inserção no meio da lista:

- Três situações
  - Novo elemento não é o menor da lista
    - percorrer a lista e inserir no meio ou final
  - Novo elemento é o menor da lista
    - Inserir no começo
  - Lista vazia
    - insere e pronto

Inserção no meio da lista:

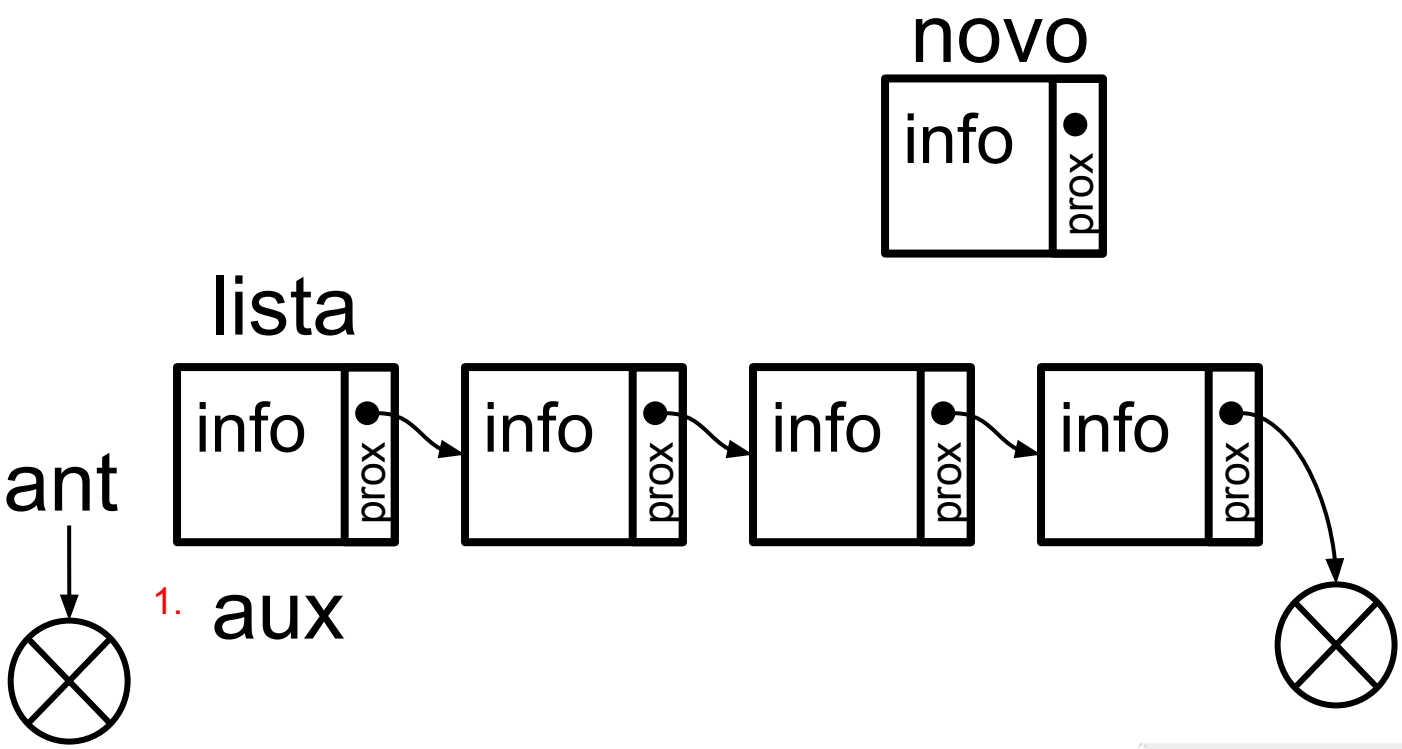
- Se novo elemento não é o menor da lista
  - Então teremos que percorrer com ponteiro auxiliar até o novo elemento ser menor que aux.
    - Temos também que ter um ponteiro para o elemento anterior a **aux**
      - para ligar
        - ant->prox = novo
        - novo->prox = aux

Inserção no meio da lista:

- Se novo elemento não é o menor da lista
  - Deste modo:
    - 1. (Se novo->info > lista->info)
      - aux = lista e ant = NULL
    - 1.1 Enquanto aux e elem > aux->info
      - ant = aux
      - aux = aux->prox
    - 1.2 Liga novo a lista
      - ant->prox = novo
      - novo->prox = aux

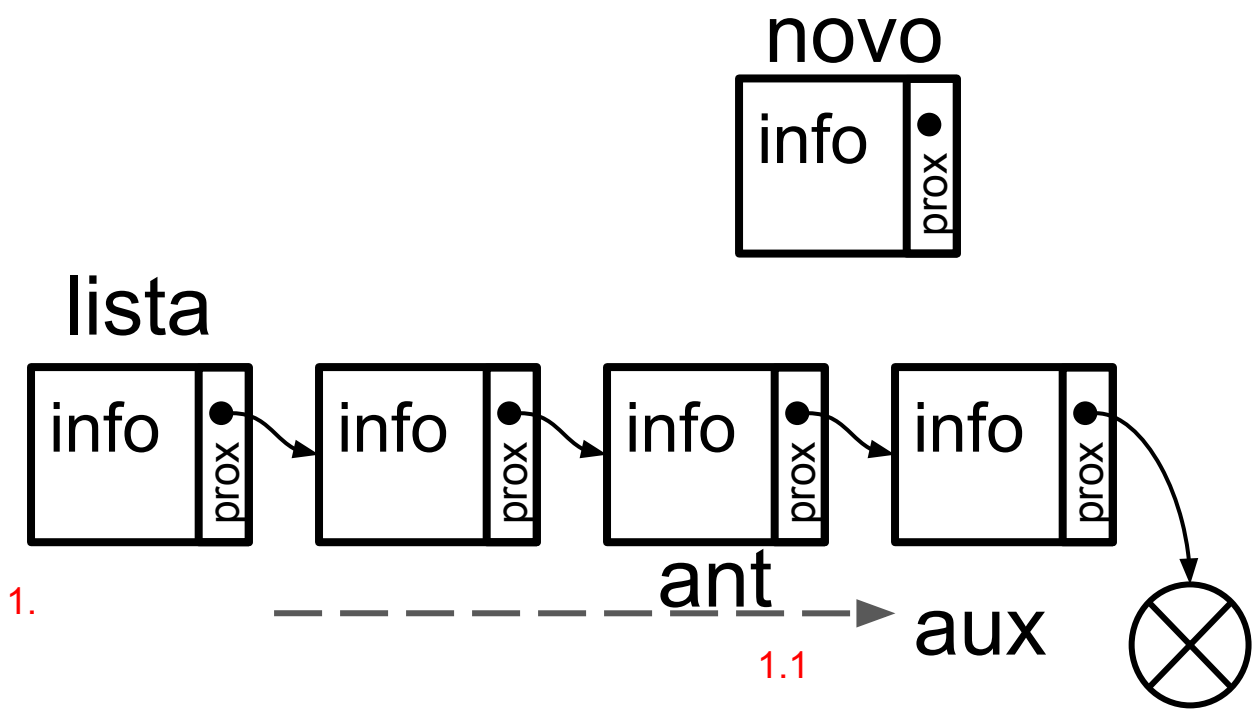
Inserção no meio da lista:

- 1.



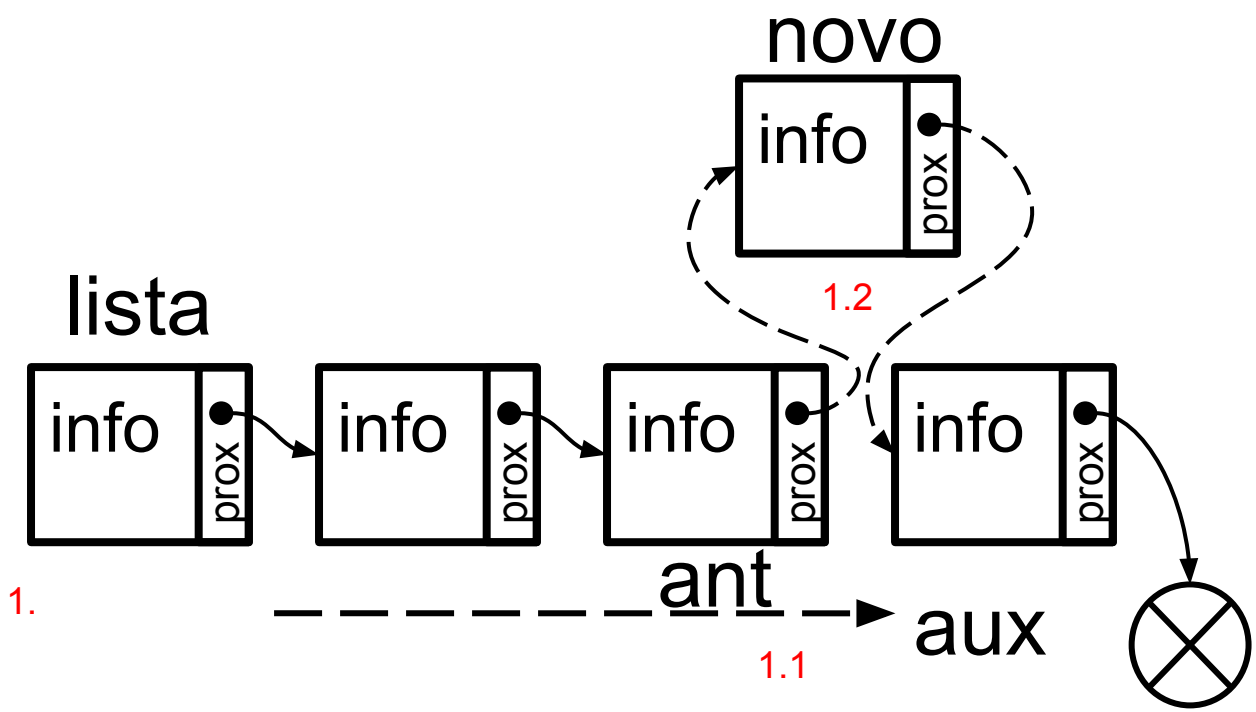
Inserção no meio da lista:

- 1.1



Inserção no meio da lista:

- 1.2
  - *obs: se aux cair em NULL, novo ficará por último porém o código não muda*



Inserção no meio da lista:

- Se novo elemento é o menor da lista
  - análogo a inserir no início
  - Se novo->info < lista->info
    - novo->prox = lista
    - lista = novo

Inserção ordenada

```
Nodo* insereOrd(Nodo *lista, int elem){
    //cria novo elemento
    Nodo* novo = new Nodo;
    novo->info = elem;
    novo->prox = NULL;
    if(!listaVazia(lista)){
        //se novo não é o menor da lista
        if(novo->info > lista->info){
            Nodo* aux = lista;
            Nodo* ant = NULL;
            while(aux && elem > aux->info){
                ant = aux;
                aux = aux->prox;
            }
            ant->prox = novo;
            novo->prox = aux;

        }
        //se novo é o menor da lista...
        else{//equivale a inserção no início
            novo->prox = lista;
            lista = novo;
        }
    }else{//lista vazia insere e pronto
        lista = novo;
    }
    return lista;
}
```

Remoção no meio da estrutura

- Assim como na inserção, remover no meio da estrutura sempre vai envolver um critério que oriente exatamente qual elemento da estrutura o algoritmo deverá remover.
- Desse modo, vamos projetar a função para buscar um elemento e em seguida removê-lo.

Remoção no meio da estrutura

- Como discutimos esta é uma disciplina de projeto
  - Envolve tomar decisões sobre o comportamento das funções projetadas
  - “Professor, e se acontecer X?”
    - Projete a função que trate essa situação!
    - Acostume-se, esse ja é, ou será seu trabalho no futuro…
    - logo..

Buscar e remover elemento

- Vamos projetar a função que:
  - dado um elemento
    - busque pela primeira ocorrência deste elemento
      - Remova a ocorrência da lista
      - Retorne a lista atualizada
    - Se não houver ocorrências
      - Não faça nada e retorne a lista como está

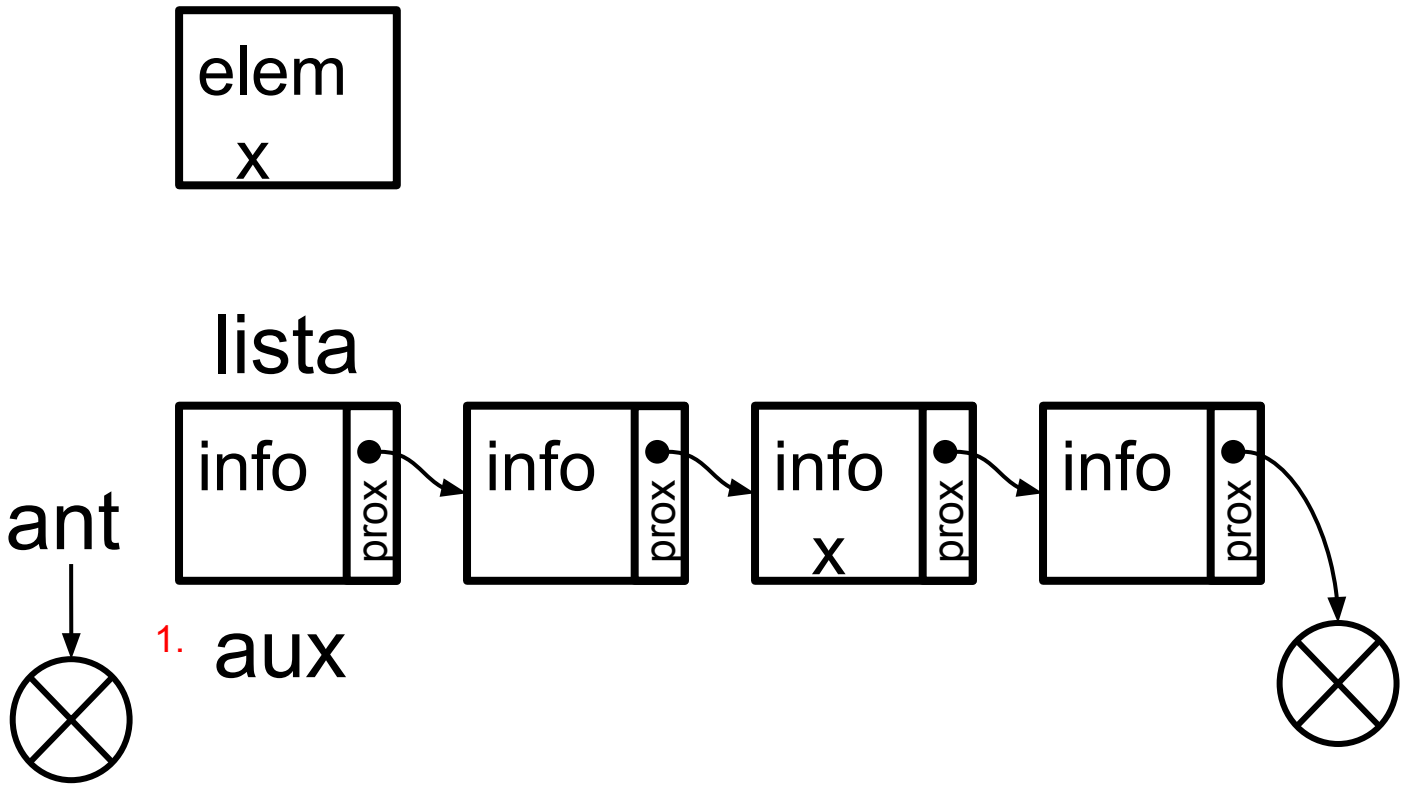
Buscar e remover elemento *elem*

- Se a lista não estiver vazia
  - 1. aux = lista e ant = NULL
  - 1.1 Percorrer até o elemento
    - Enquanto aux && aux->info != elem
      - ant = aux
      - aux = aux->prox
  - 1.2 Se achou o elemento
    - ...

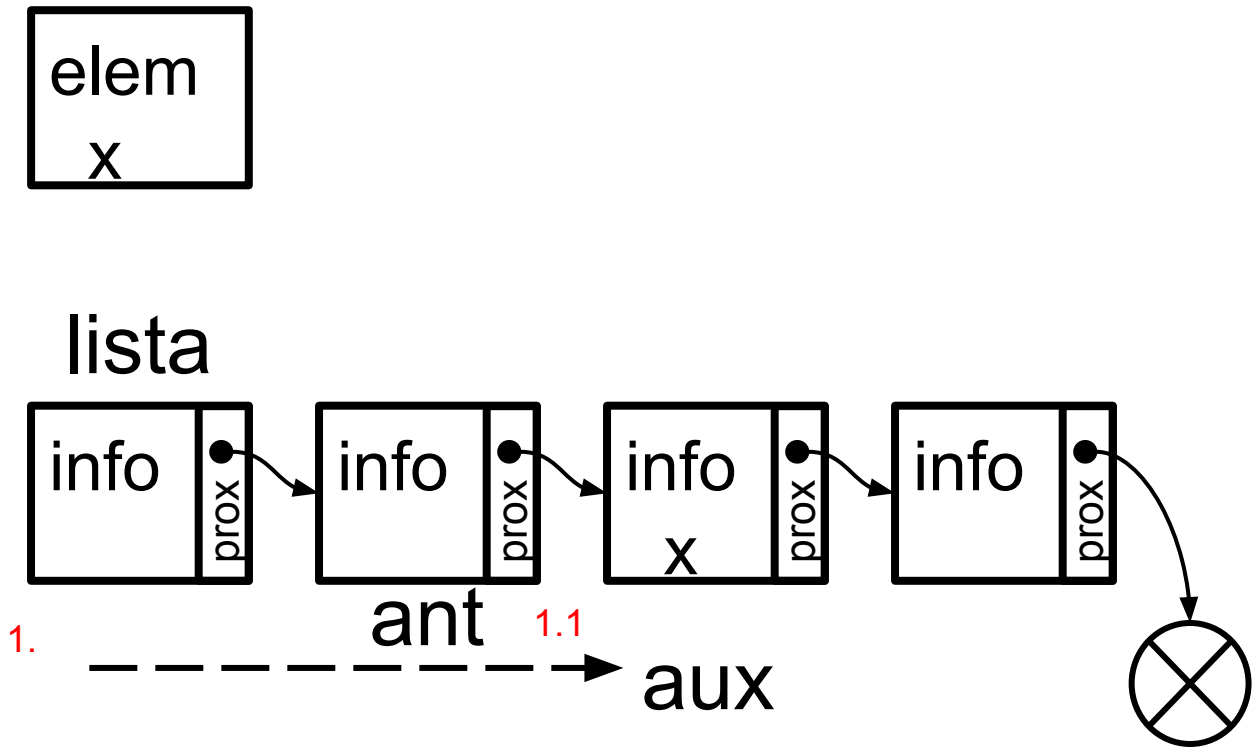
Buscar e remover elemento *elem*

- ...
- 1.2 Se achou o elemento
  - se (aux)
  - 1.2.1 Se o elem está no meio ou no final
    - Se aux != lista
      - ant->prox = aux->prox;
  - 1.2.2 Se o elem é a cabeça da lista
    - se aux==lista
      - lista = lista->prox
- 1.3 Libera aux
  - delete aux

Buscar e remover elemento *elem*

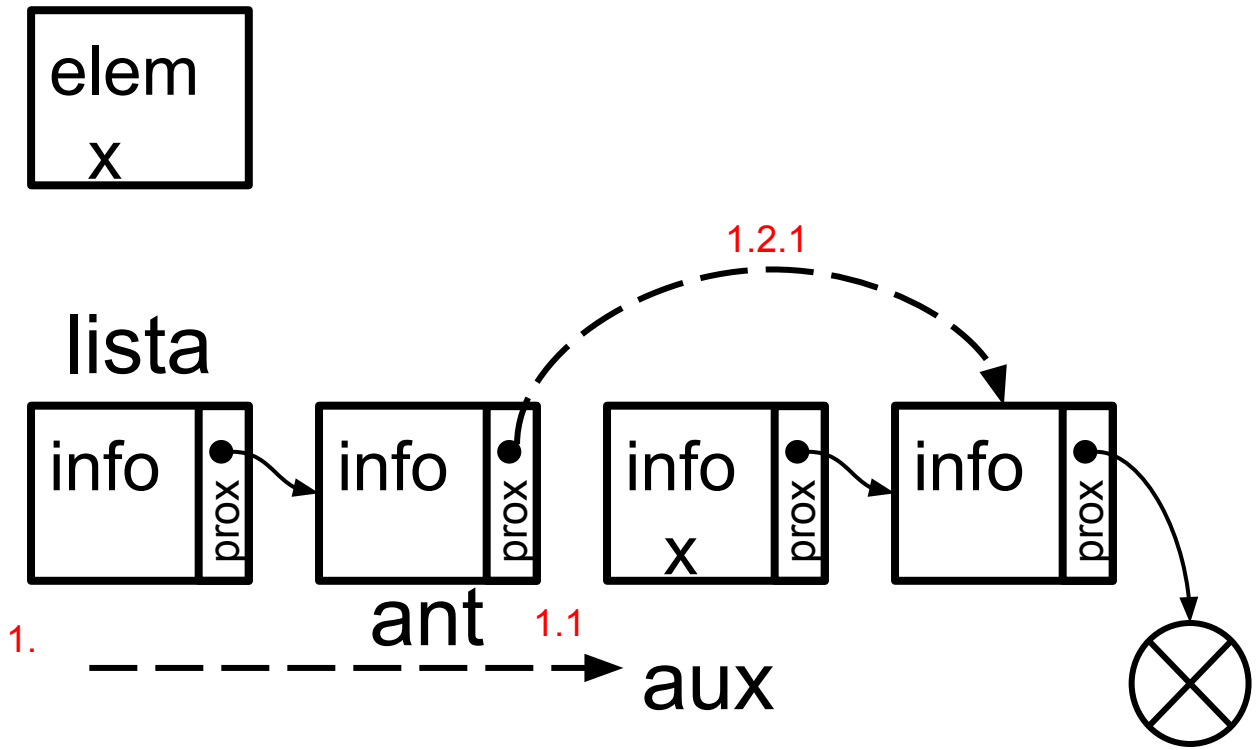


Buscar e remover elemento *elem*



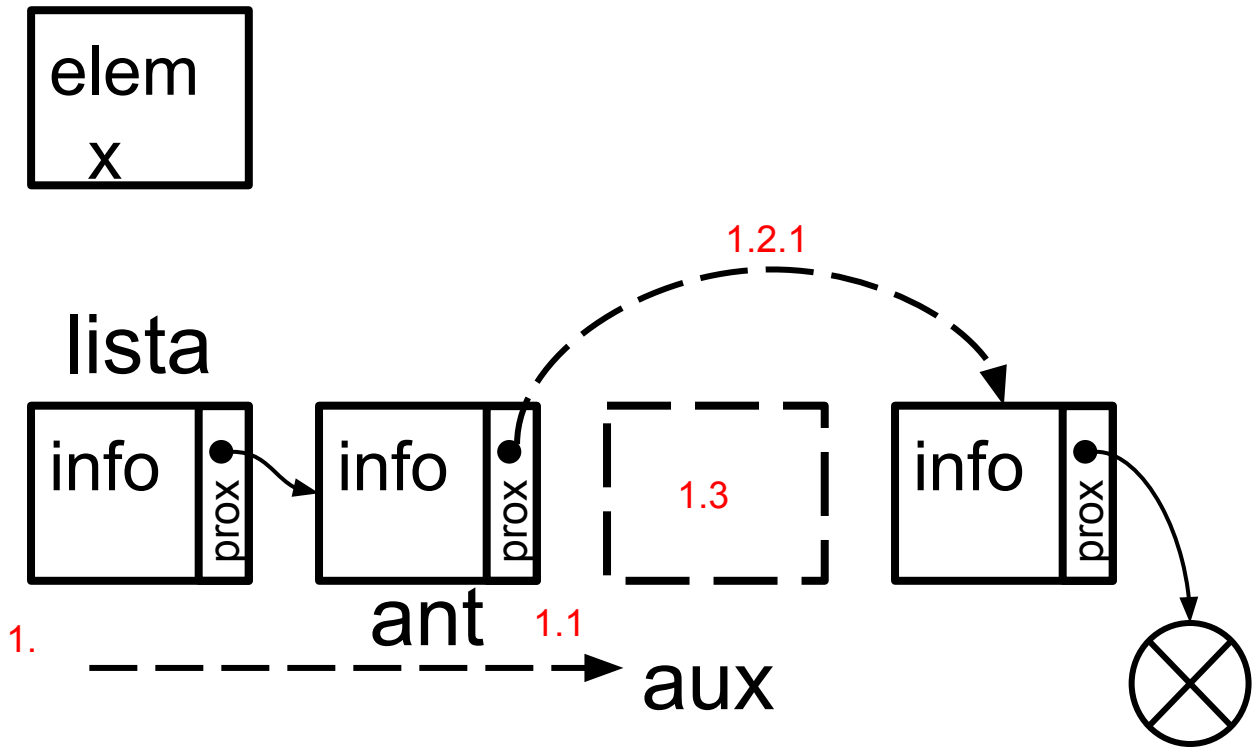
Buscar e remover elemento *elem*

1.2 Situação 1: Lista tem mais elementos



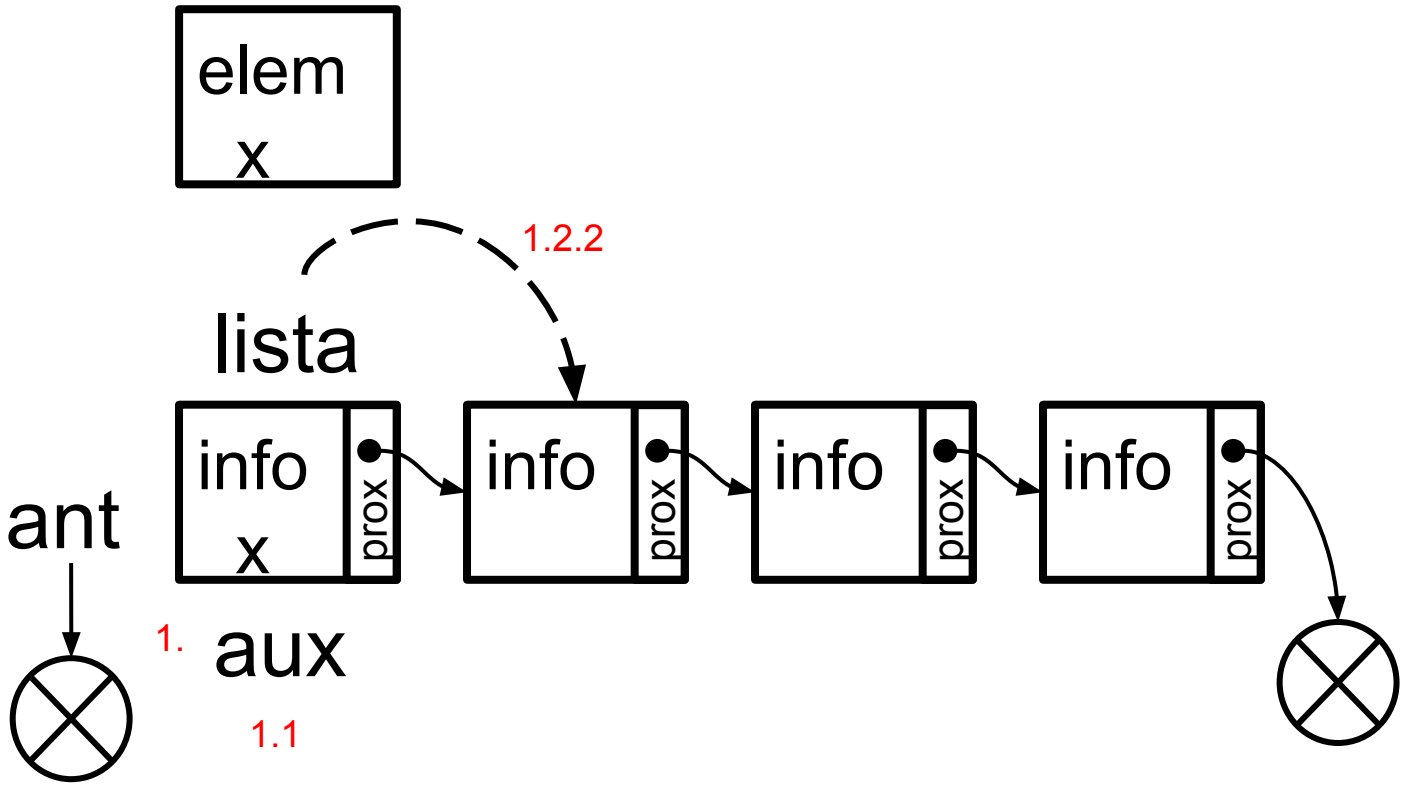
Buscar e remover elemento *elem*

1.2 Situação 1: Lista tem mais elementos



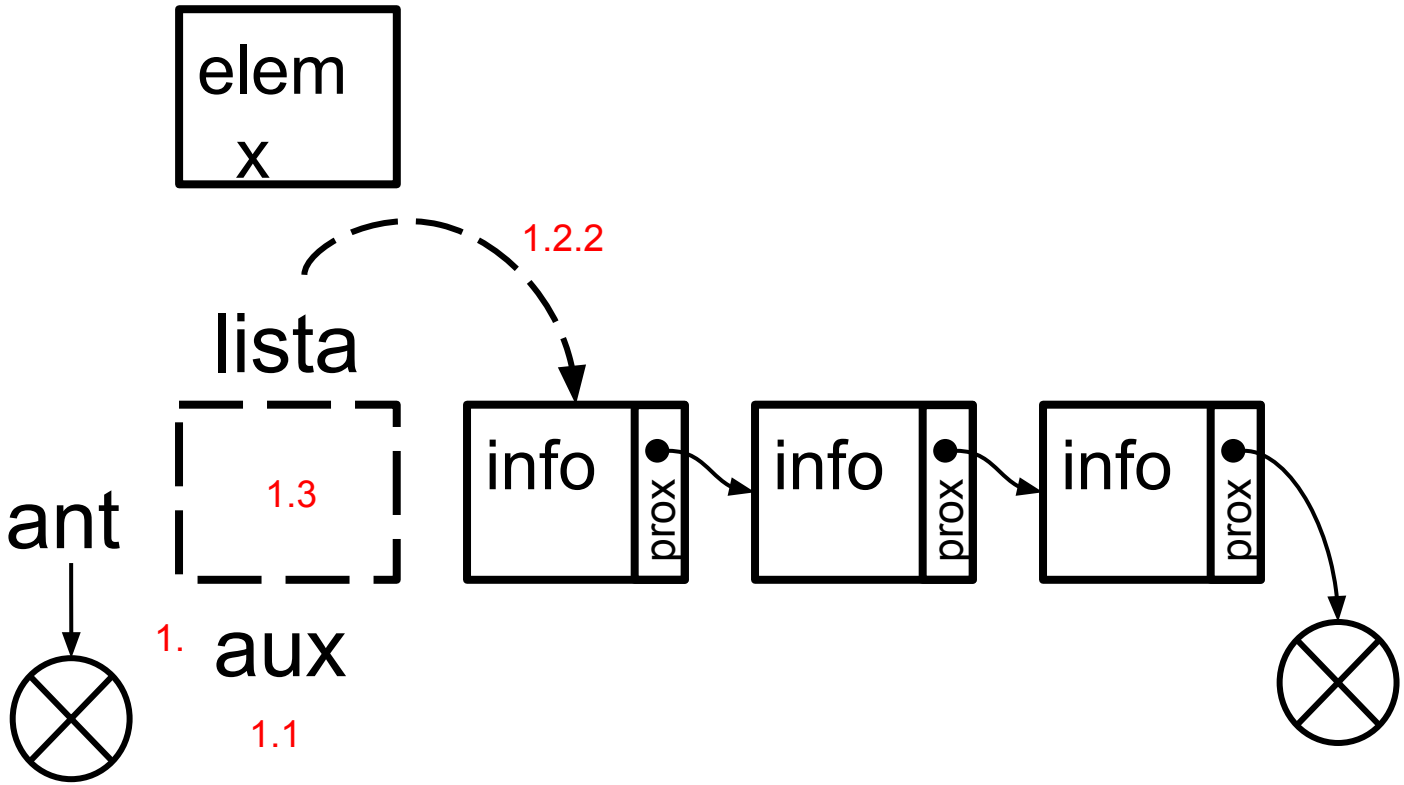
Buscar e remover elemento *elem*

1.2 Situação 2: elem é cabeça da lista



Buscar e remover elemento *elem*

1.2 Situação 2: elem é cabeça da lista



Buscar e remover elemento

// no arquivo lista.c

```
Nodo* RemoveElem(Nodo *lista, int elem){
    if(!listaVazia(lista)){
        Nodo* aux = lista;
        Nodo* ant = NULL;
        //percorre até encontrar elemento ou encerrar a lista
        while(aux && aux->info!=elem){
            ant = aux;
            aux = aux->prox;
        }
        if(aux){
            if (aux != lista){ //caso a lista tenha mais de um elemento
                ant->prox = aux->prox;
            }else{//caso tenha um só
                lista = lista->prox;
            }
            delete aux;
        }
        else{
            cout << "O elemento buscado não encontra-se na lista";
        }
    }else{
        cout << "A lista se encontra vazia!";
    }
    return lista;
}
```

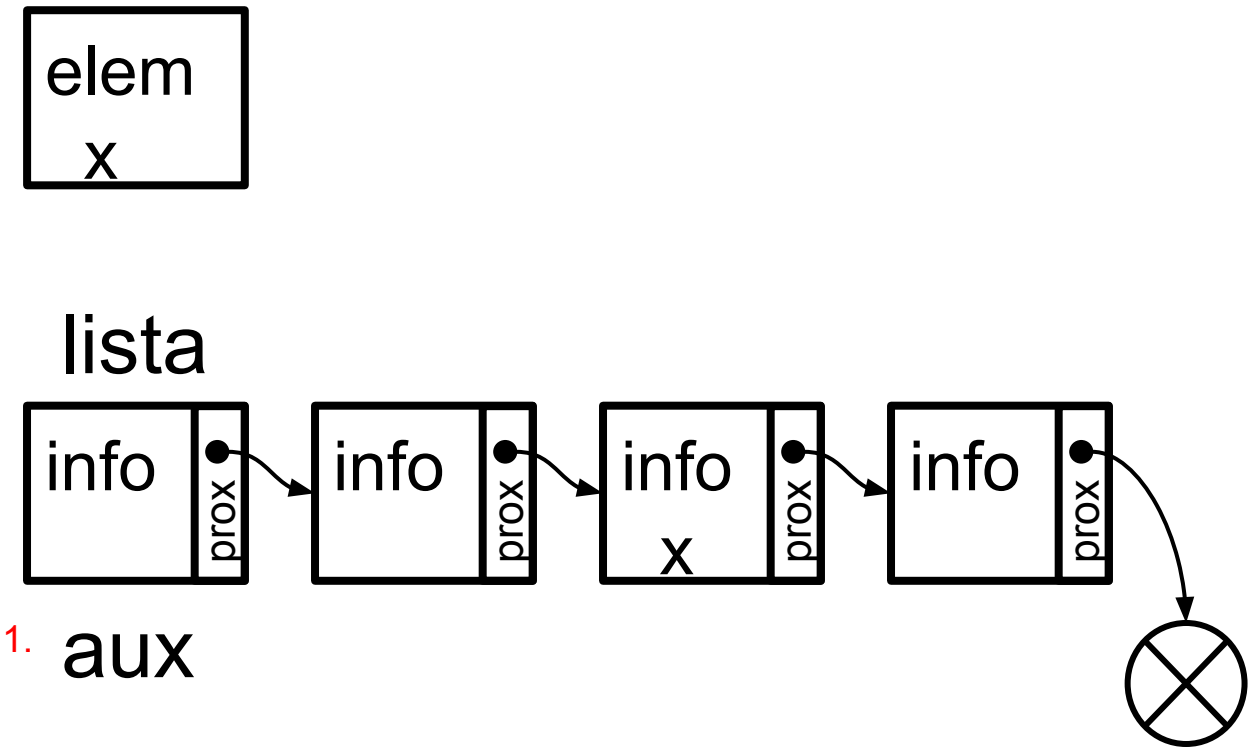
Buscar elemento

- E se eu quiser apenas buscar um elemento, sem necessariamente removê-lo?
- Podemos buscar um elemento apenas para acessar sua informação ou editá-la.
  - Vamos projetar uma função que **busca um elemento** segundo alguma chave de busca
    - algum campo da estrutura
  - **Retorna** um ponteiro para o nó em que o elemento está armazenado.
    - Ou **NULL** se o elemento buscado não existe

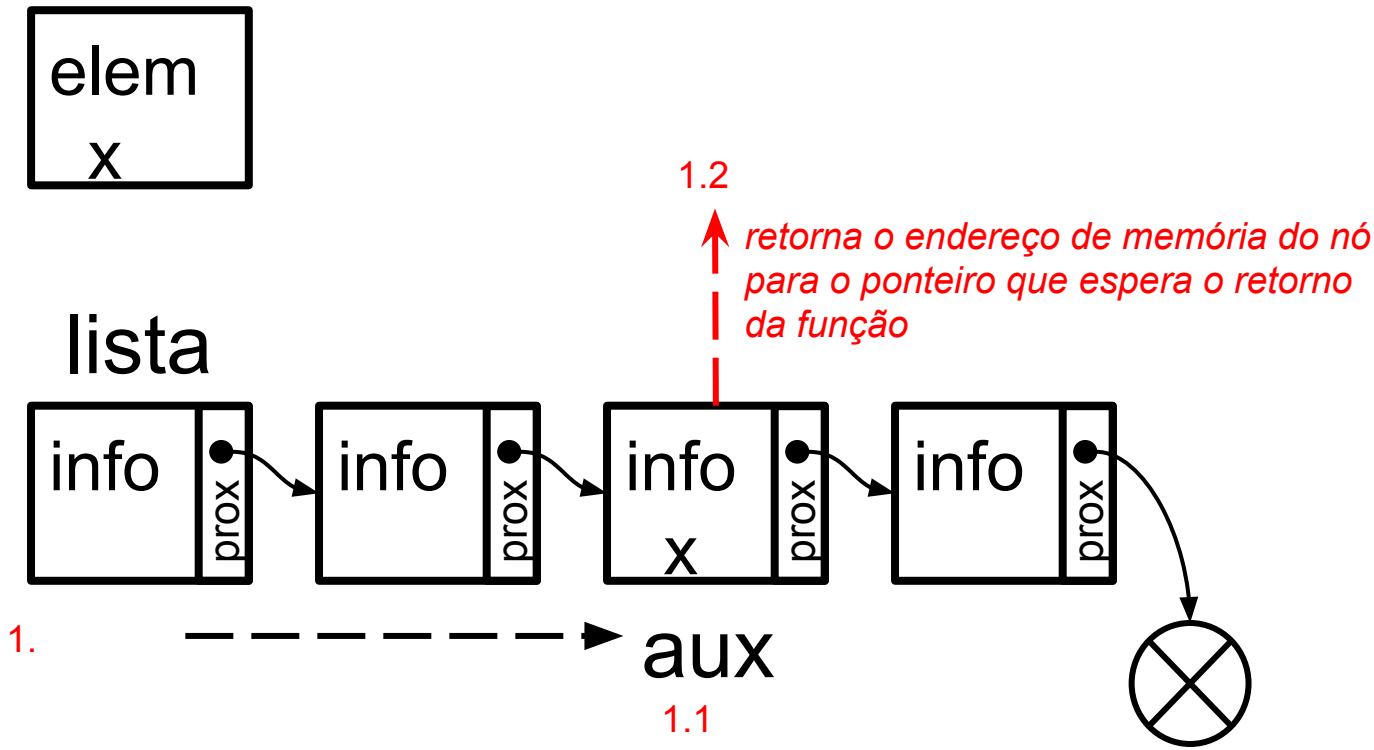
Buscar elemento

- Se lista não vazia, dado um elemento ***elem***
  - 1. aux = lista
  - 1.1 Percorre até achar ocorrencia do elem
    - Enquanto (aux && aux->info!=elem)
      - aux = aux->prox
  - 1.2 Retorna aux
    - obs: retorna de qualquer jeito. Se estiver apontando pra um nó é nosso elemento buscado, senão percorreu tudo e aux está em NULL.

Buscar elemento *elem*



Buscar elemento *elem*



Editar ou acessar elemento

- Depois de construir a função que retorna o ponteiro para o endereço buscado, construir funções para acessar seu conteúdo ou para editá-lo é simples.
- Desde que o ponteiro retornado não esteja vazio (NULL) basta acessar seus campos.

Editar elemento (depois de buscar)

```
// no arquivo lista.cpp

Nodo* buscaElem(Nodo *lista, int elem){
    Nodo* aux = lista;
    while(aux && aux->info!=elem){
        aux = aux->prox;
    }
    return aux;
}

Nodo* editaElem(Nodo *pontElem, int edit){
    if (pontElem)
        pontElem->info = edit;
}

int acessaElem(Nodo *pontElem){
    return pontElem->info;
}
```