# Introdução a Programação


*Listas Encadeadas*

# Tópicos da Aula

- Hoje aprenderemos que existem, além de vetores, estruturas de dados **dinâmicas** que podem armazenar coleções de dados
  - Estruturas Dinâmicas e  Vetores
  - Conceito de listas encadeadas
  - Listas Encadeadas x Vetores
  - Funções de manipulação de listas encadeadas
  - Variações de listas encadeadas

# Vetores

- Declaração de vetor implica na especificação de seu tamanho
  - Não se pode aumentar ou diminuir tamanho
- Outra alternativa no uso de vetores é alocar dinamicamente o espaço necessário e guardar endereço inicial do vetor
- Porém, depois de determinado o tamanho do vetor, não podemos liberar posição de memória arbitrária

**Possível desperdício ou falta de memória!**

# Estruturas Dinâmicas

- Uma estrutura de dado dinâmica consiste em uma estrutura que pode aumentar ou diminuir de tamanho de acordo com a necessidade da aplicação (do programador)
  - Evita desperdício e/ou falta de memória
- Programador é encarregado de requisitar e liberar espaços de memória explicitamente
  - Uso de alocação dinâmica

**Requer mais esforço por parte do programador!**

# Listas Encadeadas

- Uma **lista encadeada** é uma estrutura dinâmica que pode ser utilizada para armazenar um conjunto de dados
- Junto a cada elemento deve-se armazenar o endereço para o próximo elemento (**elementos encadeados**)
  - **Elemento + ponteiro = nó da lista**
  - Diferentemente de vetores, elementos geralmente não são armazenados em espaços contíguos de memória
  - Caso não exista próximo elemento, o ponteiro para o próximo elemento é NULL

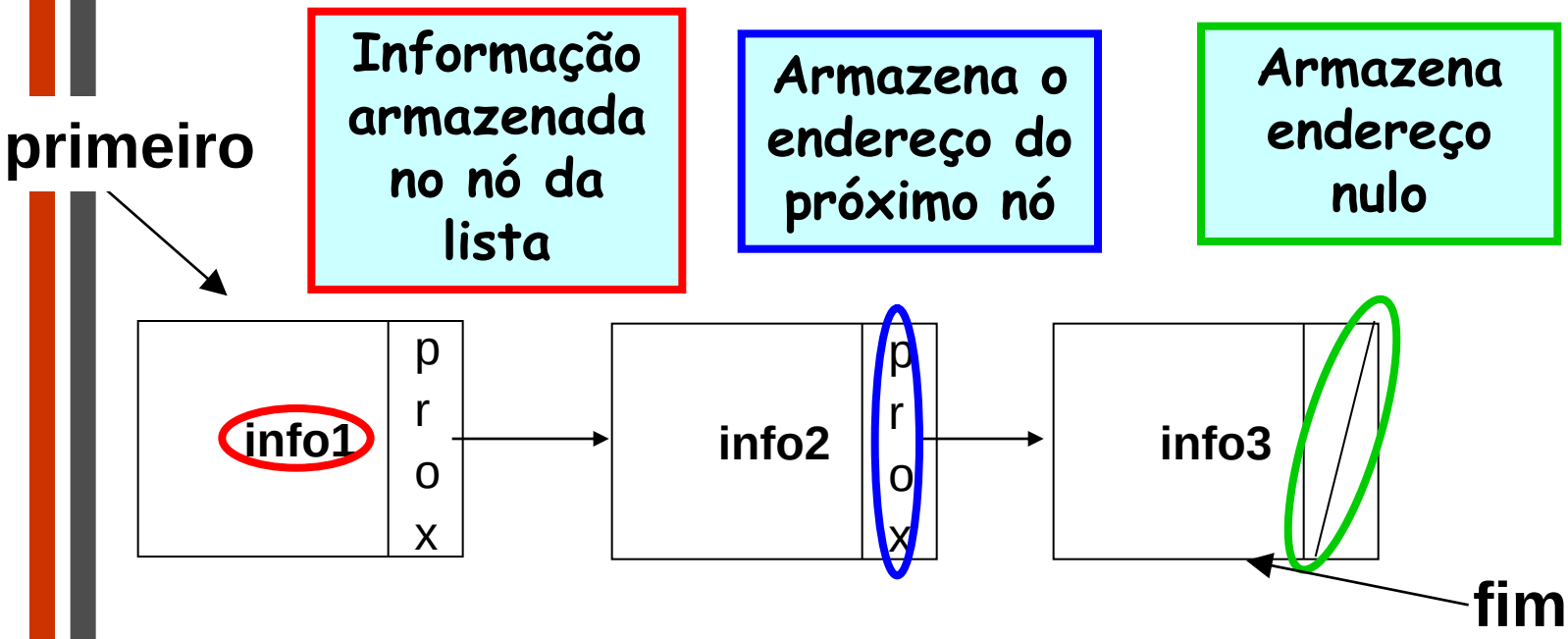
# Listas Encadeadas

- Para cada novo elemento inserido na lista, aloca-se um espaço de memória para armazená-lo
- Para remover elemento deve-se liberar o endereço de memória reservado para armazenar o elemento
- O acesso aos elementos é sempre seqüencial
  - Não se pode garantir que os elementos ocuparão um espaço de memória contíguo (não se pode localizar elementos com base em um deslocamento constante)

# Listas Encadeadas x Vetores

- Listas Encadeadas
  - + Uso eficiente da memória
  - - Complexidade de manipulação da estrutura
  - - Informação deve ser guardada em uma estrutura que guarda também um endereço (requer mais memoria por informação armazenada)
- Vetores
  - +Simplicidade no acesso (manipulação) aos elementos da estrutura
  - - Desperdício ou falta de memória
  - - Menor Flexibilidade

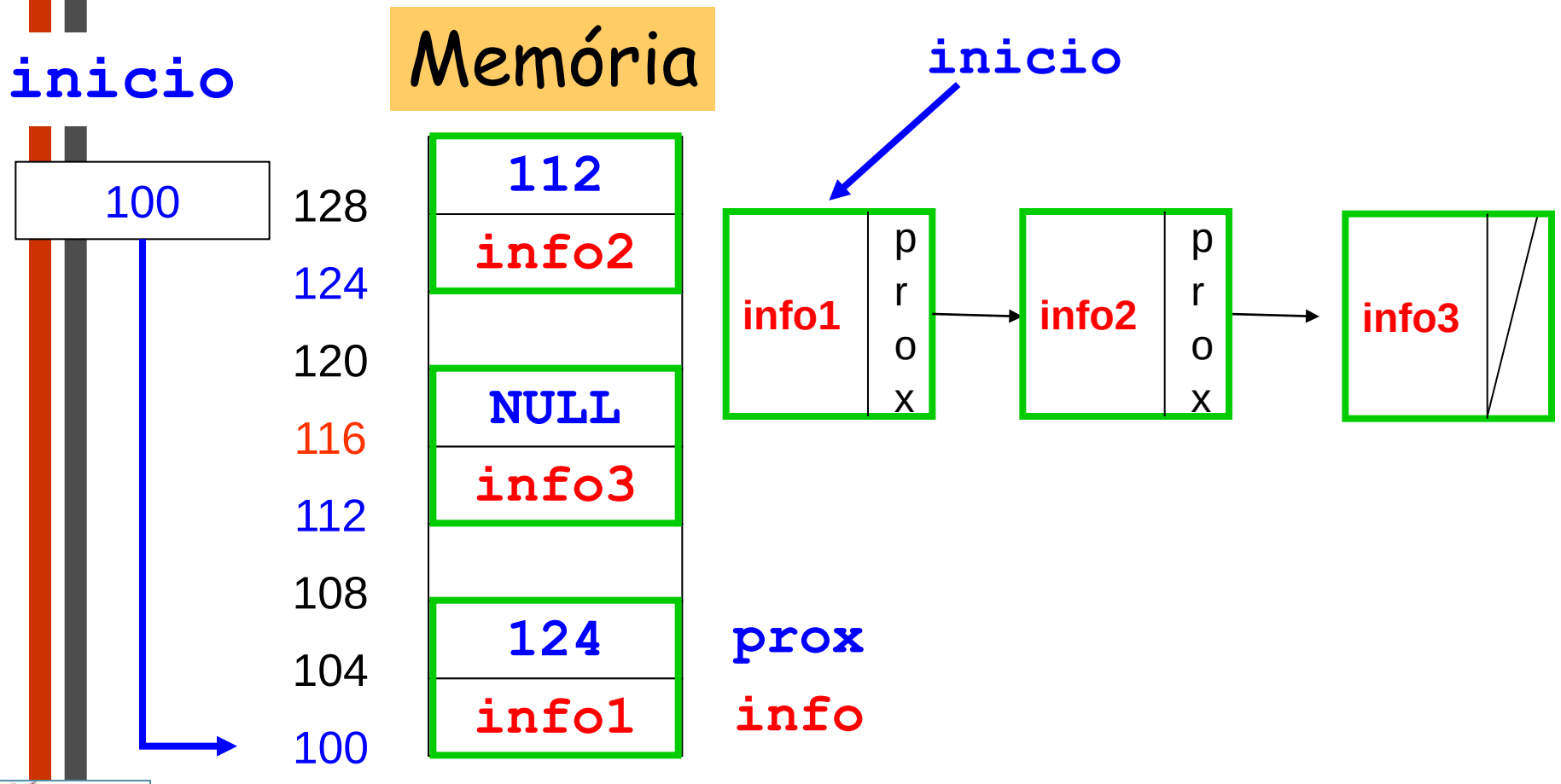
# Representação de Listas Encadeadas




```
struct   lista  {
  int info ; /* info pode ser de qualquer tipo */
  struct lista* prox;
} ;
typedef   struct lista   Lista ;
```

# Representando Listas Encadeadas na Memória

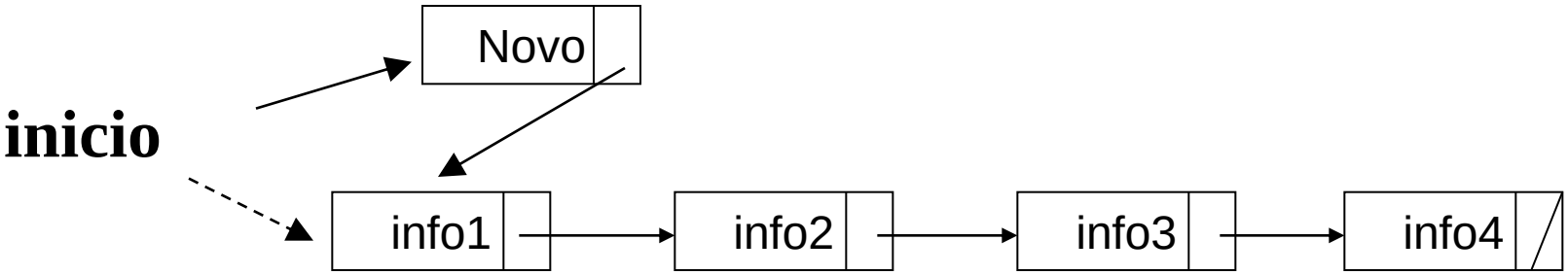
```
Lista* inicio;
```



# Inserção de Elementos em Listas Encadeadas

- Diferentes formas de inserir um elemento na lista:
  - No começo
    - Mais simples
  - No fim
    - Neste caso deve-se percorrer a lista toda

# Inserindo Elementos no Início de Listas Encadeadas




```
/* inserção início da lista */
Lista* lst_insere(Lista* inicio,int i){
   Lista* novo = (Lista*) malloc(sizeof(Lista));
   novo->info  =  i ;
   novo->prox  = inicio ;
   return   novo  ;
}
```

Endereço do começo da lista

Retorna endereço inicial atualizado

# Inserindo Elementos no Fim de Listas Encadeadas

```
/* inserção fim da lista */
Lista* lst_insere(Lista* inicio,int i){
  Lista* novo = (Lista*) malloc(sizeof(Lista));
  novo->info  =  i ;
  novo->prox  = NULL ;
  if (inicio  == NULL){
      inicio = novo;
  } else {
      Lista* p = inicio;
      while (p->prox != NULL){
            p = p->prox;
      }
      p->prox = novo;
  }
  return   inicio  ;
}
```

Se lista estiver vazia, inicio vai guardar endereço de novo

Senão temos que percorrer lista até último nó, para fazê-lo apontar para novo

# Usando Função de Inserção de Listas Encadeadas

```
int   main() {
   Lista* inicio;
   inicio = NULL;
   inicio = lst_insere(inicio,23); /* insere 23 */
   inicio = lst_insere(inicio,45); /* insere 45 */
   ...
   return   0  ;
}
```

Variavel que armazena endereço inicial da lista deve ser inicializada com **NULL**!

Não esquecer de atualizar a variável que guarda endereço inicial da lista a cada inserção!

# Imprimindo Listas Encadeadas

```
void  lst_imprime(Lista* inicio){
  Lista* p; /*variável auxiliar para percorrer a
              lista */
  for(p = inicio; p != NULL; p = p->prox){

      printf ( "info = %d\n ",  p->info ) ;
  }
}
```

Laço para percorrer todos os nós da lista

# Verificando se a Lista Está Vazia

```
/* função vazia: retorna 1 se vazia ou 0 se não
vazia */
int  lst_vazia(Lista* inicio){
  if (inicio == NULL)
      return 1;
  else
      return 0;
}
```

```
/* versão compacta da função lst_vazia */
int   lst_vazia  ( Lista* inicio ){
  return(inicio == NULL) ;
}
```

# Buscando um Elemento em Listas Encadeadas

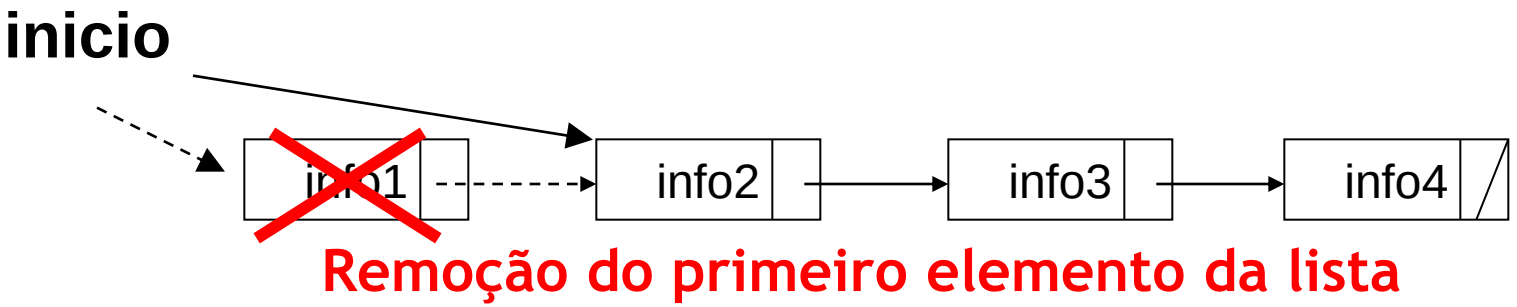
```
Lista*  lst_busca(Lista* inicio,int v){
   int achou = 0;
   Lista* p = inicio ;
   while (p != NULL && !achou) {
       if (p->info == v)
          achou = 1 ;
       else
          p = p->prox;
   }
   return p;
}
```

Se não achou elemento, retorna o endereço nulo!

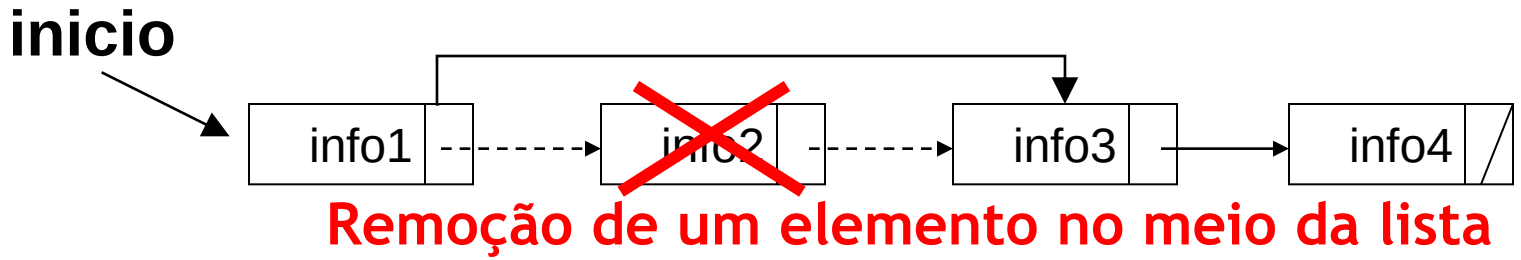
Retorna o endereço do primeiro nó que contém o elemento buscado

# Removendo Elementos de Listas Encadeadas

- Dois casos devem ser tratados para a remoção de um elemento da lista




Remoção do primeiro elemento da lista




Remoção de um elemento no meio da lista

# Removendo Elementos de Listas Encadeadas

```
Lista* lst_retira(Lista* inicio,int v){
  Lista* ant = NULL; /* guarda elemento anterior */
  Lista* p = inicio;
  while(p != NULL && p->info != v){
      ant  =  p ;
      p  =  p->prox ;
  }
/* verifica se achou o elemento */
  if (p != NULL) {
      if (ant == NULL)
            inicio = p->prox;
      else
            ant->prox = p->prox;
      free(p) ;
  }
  return inicio ;
}
```

Procura elemento na lista,guardando endereço do anterior

Elemento no inicio, retira elemento do início

Retira elemento do meio

Retorna endereço inicial da lista

# Liberando Listas Encadeadas

```
void    lst_libera(Lista* inicio){
  Lista* p  =  inicio ;
  while(p != NULL )  {
     Lista* t = p->prox;
     free(p); /* libera a memória apontada por "p"*/
     p = t;   /* faz  "p" apontar para o próximo */
  }
}
```

Guarda o endereço do próximo nó para poder libera-lo na próxima iteração

# Comparando Listas Encadeadas

```
int   lst_igual(Lista* lst1,Lista* lst2){
  Lista* p1 ;
  Lista* p2 ;
  int igual = 1
  for(p1 = lst1,p2 = lst2; p1!=NULL && p2!=NULL && igual;
      p1=p1->prox,p2=p2->prox){

        if(p1->info!= p2->info )
            igual = 0;
  }
  return(p1 == p2 && igual) ;
}
```

Se os elementos testados forem iguais, deve-se ainda ver se p1 e p2 apontam para NULL (tamanhos das listas iguais)

# Utilizando Funções de Manipulação de Lista de Inteiros

```
#include <stdio.h>
#include    "lista.h"
int main() {
   Lista  * ls ; /* declara uma lista não iniciada */
   ls = NULL; /*inicia lista vazia */
   ls = lst_insere(ls,23); /* insere o elemento 23 */
   ls = lst_insere(ls,45); /* insere o elemento 45 */
   ls = lst_insere(ls,56); /* insere o elemento 56 */
   ls = lst_insere(ls,78); /* insere o elemento 78 */
   lst_imprime(ls);        /* imprime: 78 56 45 23 */
   ls = lst_retira(ls,78);
   lst_imprime(ls);        /* imprime: 56 45 23 */
   ls = lst_retira(ls,45);
   lst_imprime(ls);        /* imprime: 56 23 */
   lst_libera(ls);
    return   0 ;
}
```

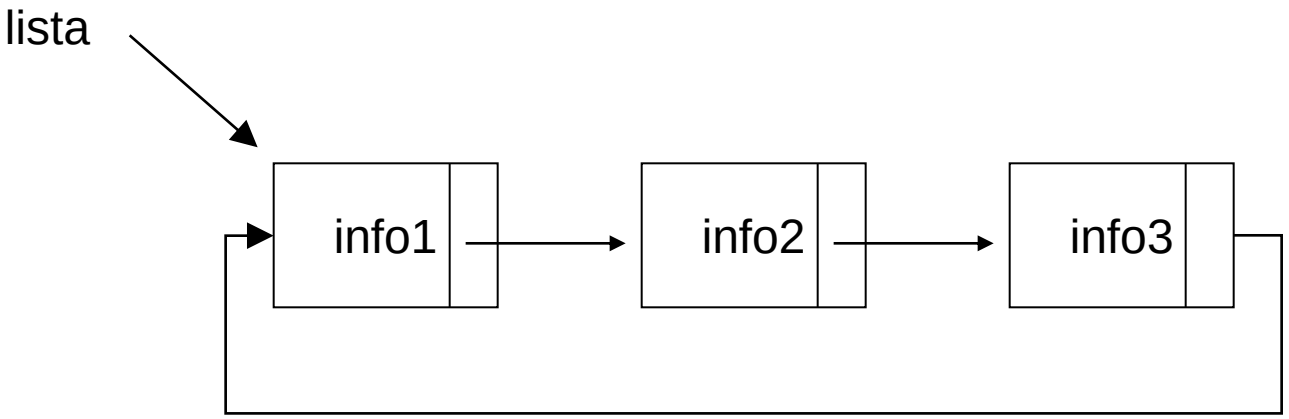
# Variações de Listas Encadeadas

- Listas podem variar quanto ao:
  - Tipo de encadeamento
    - Simples
    - Circulares
    - Duplas
    - Circulares e Duplas
  - Conteúdo
    - Tipos Primitivos
    - Tipos Estruturados

**Já vimos anteriormente**

# Listas Circulares

- Estrutura do nó da lista é idêntica a da lista simplesmente endadeada
- Não há noção de primeiro e último nó da lista
- Para saber se toda lista foi percorrida deve-se guardar o endereço do primeiro nó a ser lido e comparar com o endereço do nó que está sendo lido



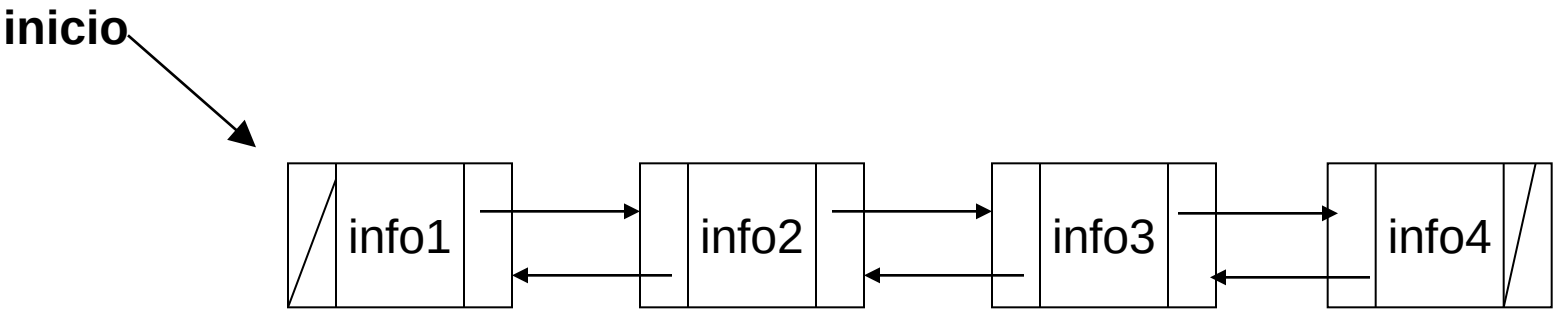
# Imprimindo uma Lista Circular

```
void lcirc_imprime (Lista* inicio) {
  Lista* p = inicio; /* faz p apontar para o
  nó inicial */
  /* testa se lista não é vazia*/
  if (p != NULL){
     do {
          printf("%d\n",p->info);
          p = p->prox;
     } while (p != inicio);
}
```

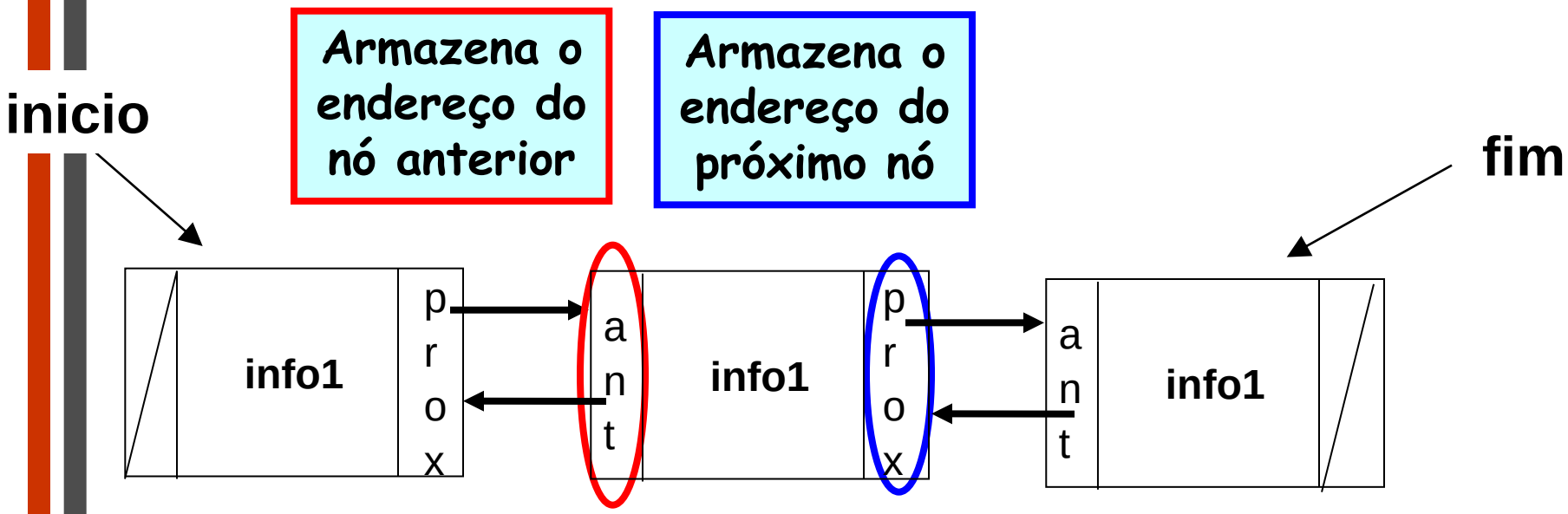
A condição de parada do laço é quando o nó a ser percorrido for igual ao nó inicial

# Lista duplamente encadeada

- Permite percorrer a lista em dois sentidos
- Cada elemento deve guardar os endereços do próximo elemento e do elemento anterior
- Facilita percorrer a lista em ordem inversa
- Retirada de elemento cujo endereço é conhecido se torna mais simples
  - Acesso ao nó anterior para ajustar encadeamento não implica em percorrer a lista toda



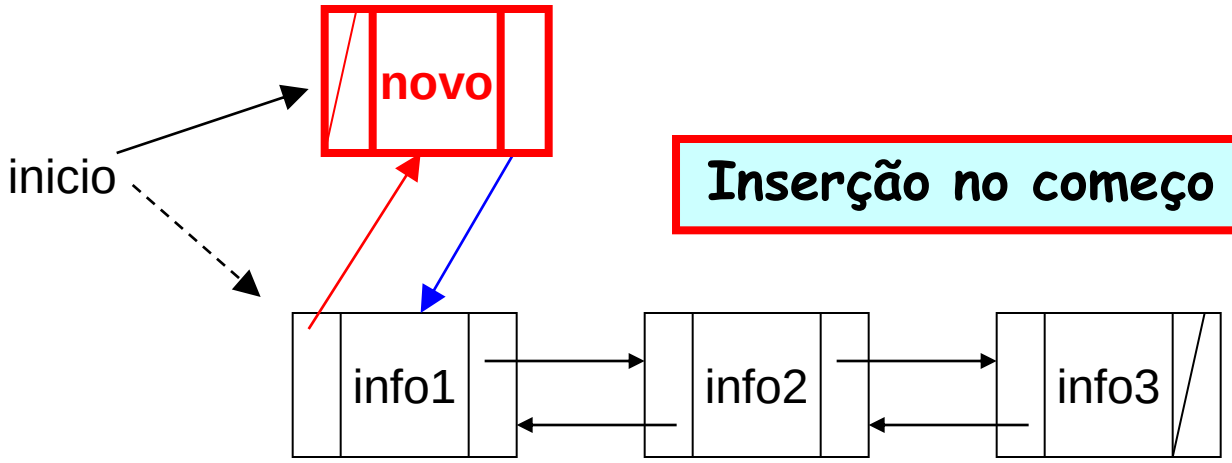
# Estrutura de Listas Duplamente Encadeadas




```
struct   lista2   {
   int info ;
   struct lista2* prox;

   struct lista2* ant;
} ;
typedef    struct lista2    Lista2 ;
```

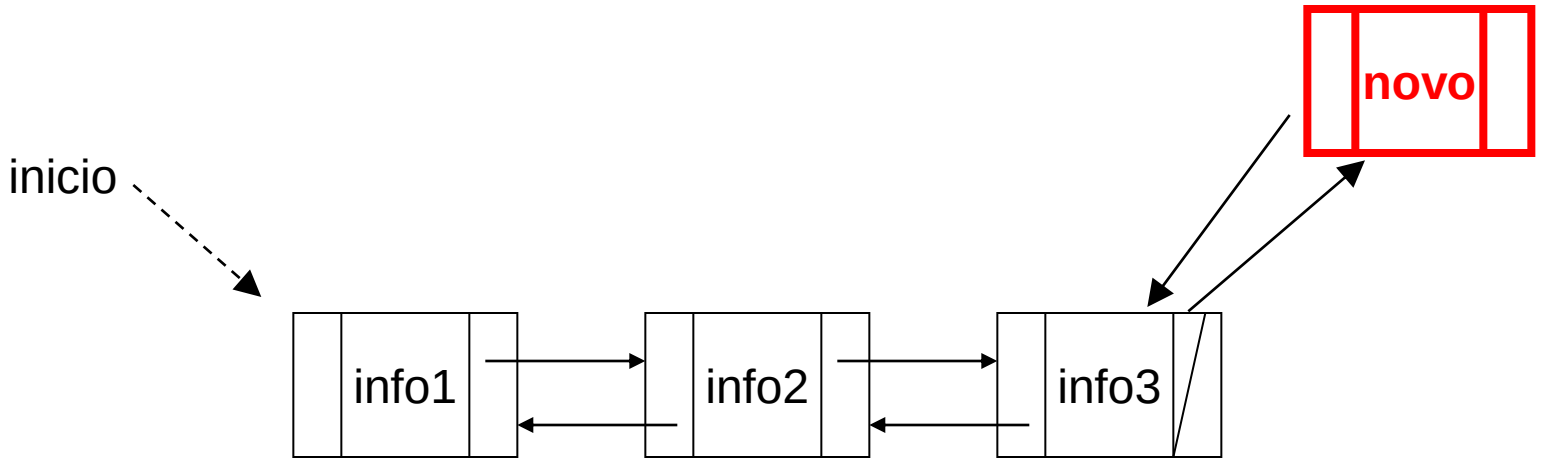
# Inserindo Elementos no Início de Listas Duplamente Encadeadas




**Inserção no começo da lista**

```
Lista2* lst2_insere (Lista2* inicio, int v) {
  Lista2* novo = (Lista2*) malloc(sizeof(Lista2));
  novo->info = v;  novo->prox = inicio;
  novo->ant = NULL;
  /* verifica se lista não está vazia */
  if (inicio != NULL)
     inicio->ant = novo;
  return novo;
}
```

# Inserindo Elementos no Fim de Listas Duplamente Encadeadas




Inserção no fim da lista

# Inserindo Elementos no Fim de Listas Duplamente Encadeadas

```
Lista2* lst2_insere (Lista2* inicio, int v) {
  Lista2* novo = (Lista2*) malloc(sizeof(Lista2));
  novo->info = v;  novo->prox = NULL;
  if (inicio == NULL) {
     novo->ant = NULL;
     inicio = novo
  } else {
     Lista2* p;
     while (p->prox != NULL) [
        p = p->prox;
     }
     p->prox = novo;
     novo->ant = p;
  }
  return inicio;
}
```

Se lista estiver vazia, endereço do nó anterior a novo é NULL

Senão temos que fazer com que o último nó aponte para novo e nó anterior de novo seja o antigo último nó

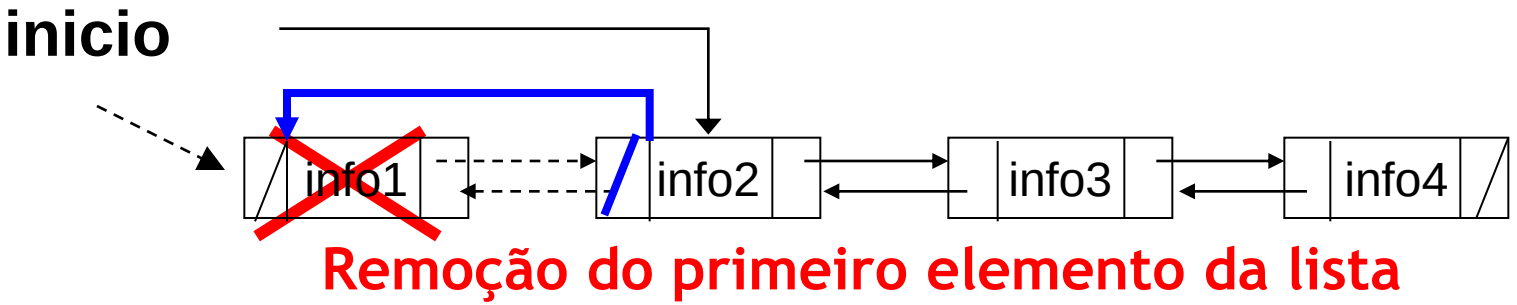
# Buscando um Elemento em Listas Duplamente Encadeadas

- Mesma lógica aplicada a listas simplesmente encadeadas

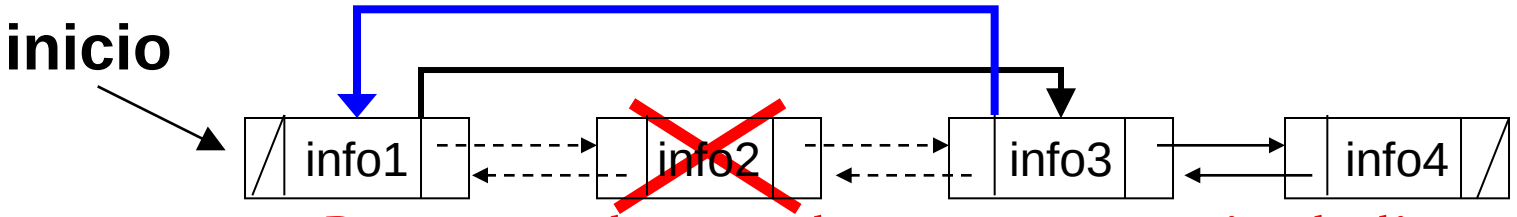
```
Lista2* lst_busca (Lista2* lista, int v)
{
  Lista2* p;
  int achou = 0
  for (p = lista; p != NULL && !achou; p = p->prox)
       if (p->info == v)
          achou = 1;
  return p;
}
```

# Removendo Elementos de Listas Duplamente Encadeadas

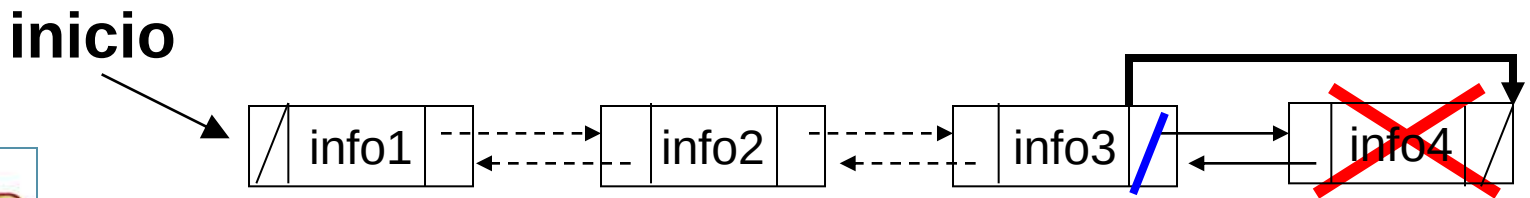
- Três casos devem ser tratados para a remoção de um elemento da lista




Remoção do primeiro elemento da lista




Remoção de um elemento no meio da lista




Remoção de um elemento do fim da lista

# Removendo Elementos de Listas Duplamente Encadeadas

```
Lista2* lst2_retira (Lista2* inicio, int v) {
  Lista2* p = busca(inicio, v);
  if (p != NULL){
      if (inicio == p)
            inicio = p->prox;
      else
          p->ant->prox = p->prox;
      if (p->prox != NULL)
            p->prox->ant = p->ant;
      free(p);
 }
 return inicio;
}
```

Usando a função de busca

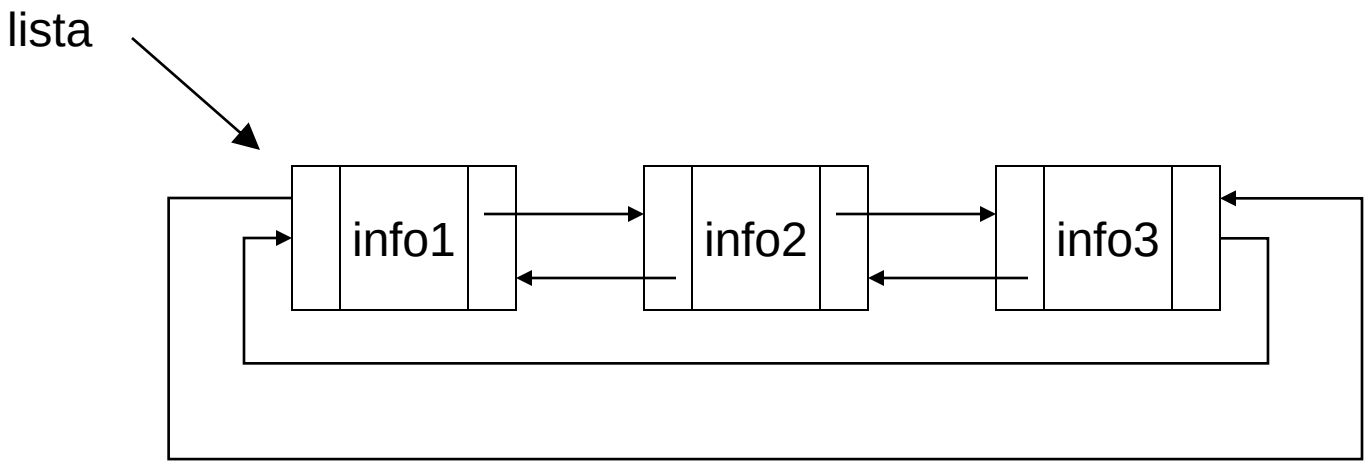
Retira elemento do começo da lista

Retira do meio

Só acerta o encadeamento do ponteiro para o anterior se NÃO for o último elemento

# Lista Circular Duplamente Encadeada

- Permite percorrer a lista nos dois sentidos, a partir de um ponteiro para um elemento qualquer



# Imprimindo no Sentido Inverso com um Lista Circular Duplamente Encadeada

```
void l2circ_imprime_inverso (Lista2* lista){
  Lista2* p = lista;
  if (p != NULL)
      do {
           p = p->ant;
           printf("%d\n", p->info);
      } while (p != lista);
}
```

Avança para o nó anterior

# Lista de Tipos Estruturados

- Uma lista pode ser composta de elementos estruturados
- Considere o tipo Retangulo

```
typedef struct retangulo {
    float b;  /* base */
    float h;  /* altura */
} Retangulo;
```

- Podemos definir uma lista cujos nós contenham ou um retângulo ou um ponteiro para um retângulo

```
typedef struct lista {
    Retangulo info;
    struct lista *prox;
} Lista;
```

**OU**

```
typedef struct lista {
    Retangulo* info;
    struct lista *prox;
} Lista;
```

# Inserindo um Nó em uma Lista de Tipos Estruturados

- Nó da lista contém um retângulo

```
Lista* insere (Lista* inicio,float b, float h) {
  Lista *novo;
  novo = (Lista*)malloc(sizeof(Lista));
  novo->info.b = b;
  novo->info.h = h;
  novo->prox = inicio
  return novo;
}
```

# Inserindo um Nó de uma Lista de Tipos Estruturados

- Nó da lista contém um ponteiro para um retângulo

```
Lista* insere (Lista* inicio,float b, float h){
  Retangulo *r;
  r = (Retangulo*) malloc(sizeof(Retangulo));
  Lista* novo = (Lista*)malloc(sizeof(Lista));
  r->b = b;
  r->h = h;
  novo->info = r;
  novo->prox = inicio;
  return novo;
}
```

Espaço para a estrutura Retangulo deve também ser alocado