Métodos Computacionais

Variações de Listas Encadeadas

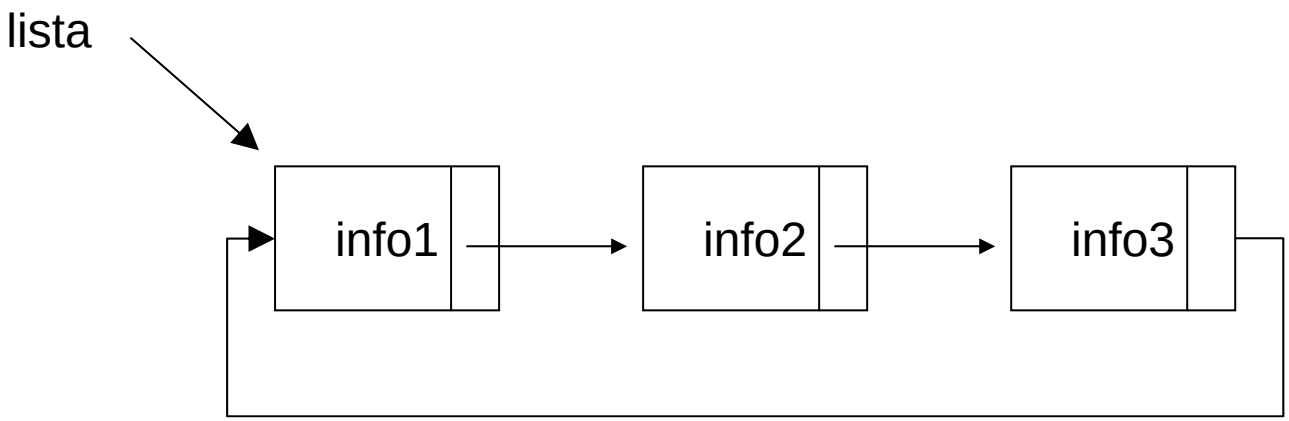
Variações de Listas Encadeadas

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

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



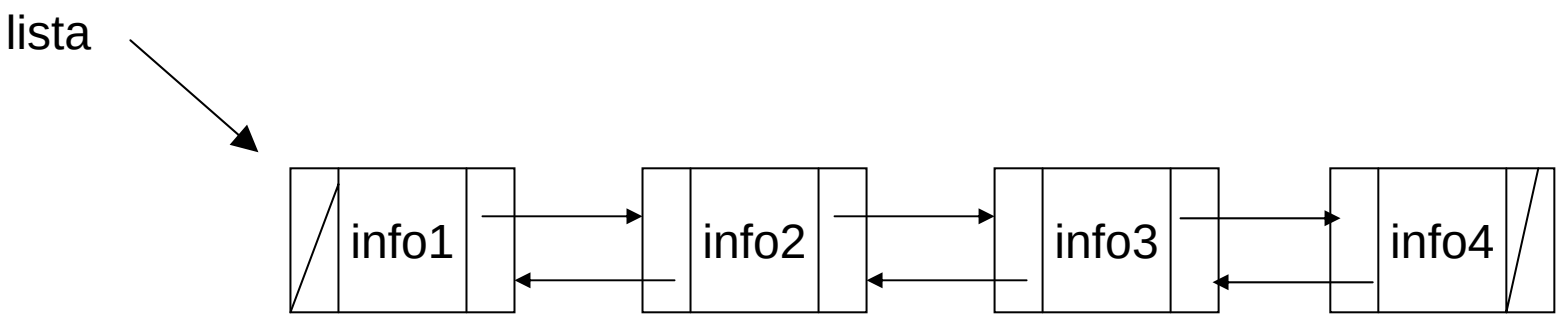
Imprimindo uma Lista Circular

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

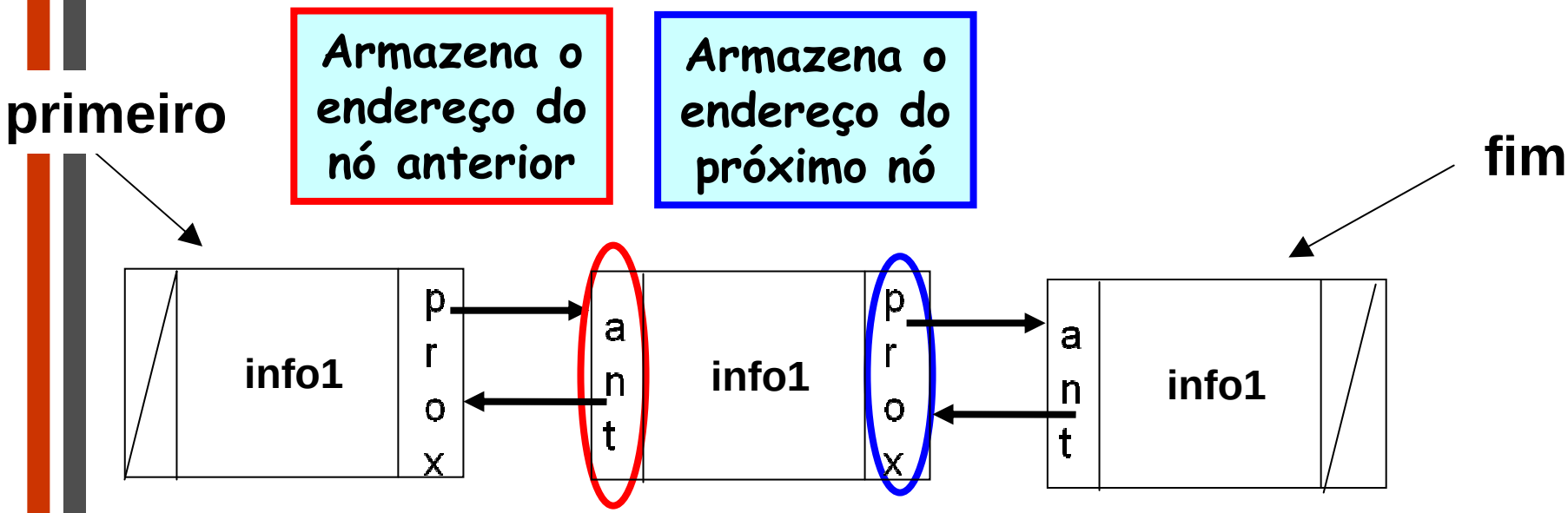
A condição de parada do laço é quando o nó a ser percorrido for igual ao nó inicial

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

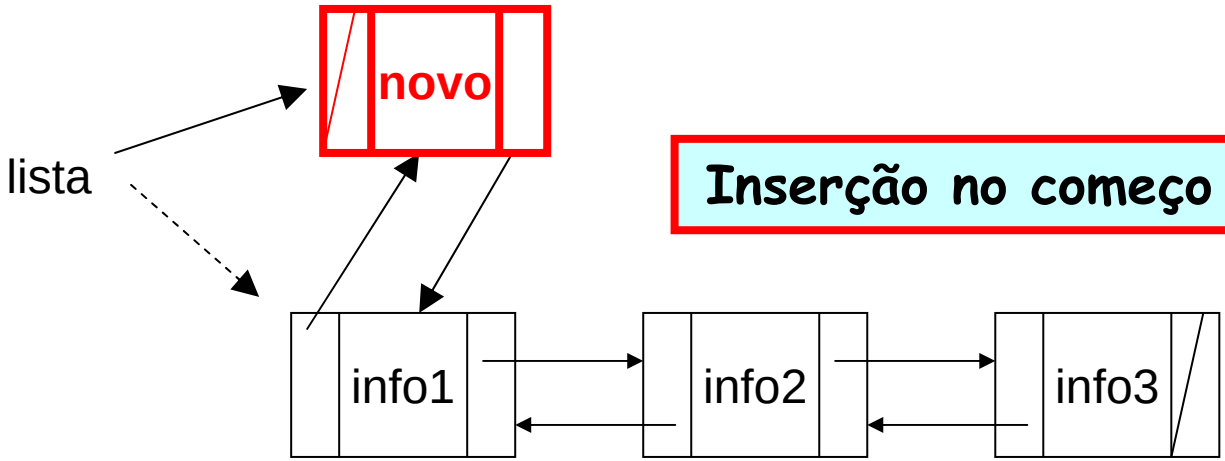


Estrutura de Listas Duplamente Encadeadas




```
struct   lista2  {
  int info ;
  struct lista2 *prox; /* armazena endereço do
                          próximo elemento */
  struct lista2 *ant; /* armazena endereço do
                         elemento anterior */
} ;
typedef   struct lista2   Lista2 ;
```

Inserindo Elementos em Listas Duplamente Encadeadas



Inserção no começo da lista

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

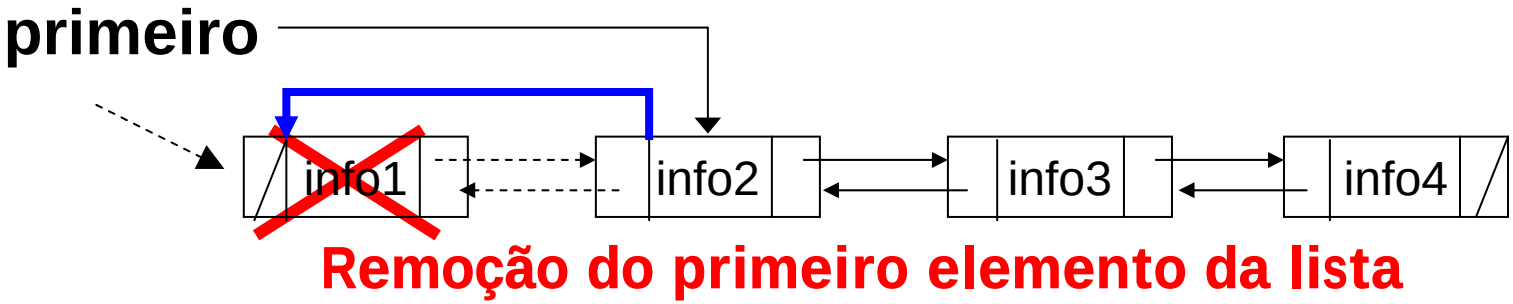
Buscando um Elemento em Listas Duplamente Encadeadas

- Mesma lógica aplicada a listas simplesmente encadeadas

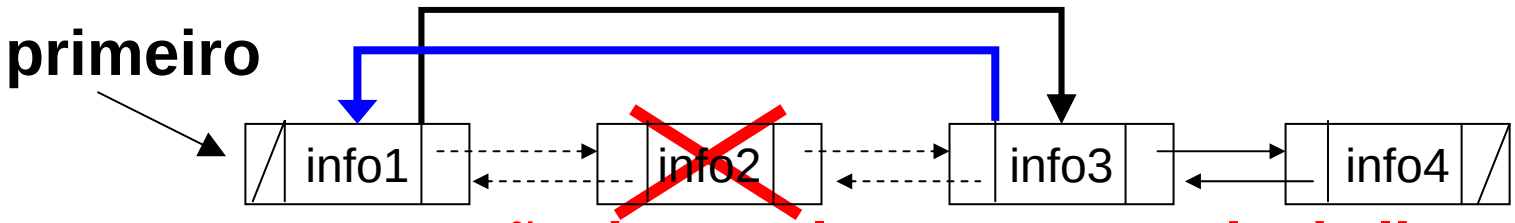
```
Lista2* lst_busca (Lista2* lista, int v)
{
  Lista2* p;
  for (p = lista; p != NULL; p = p->prox)
       if (p->info == v)
          return p;
  return NULL;/* não achou o elemento */
}
```

Removendo Elementos de Listas Duplamente Encadeadas

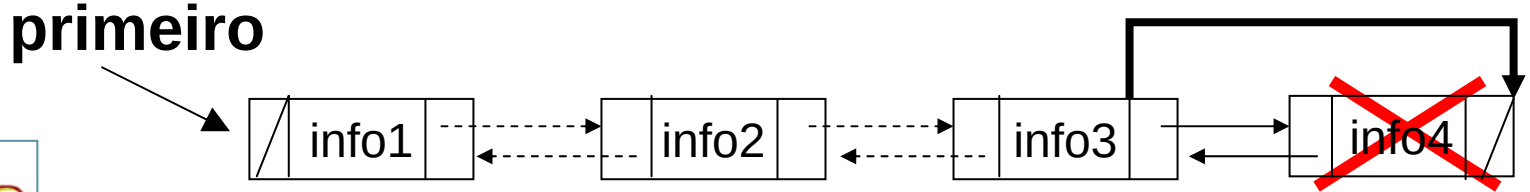
- Três casos devem ser tratados para a remoção de um elemento da lista



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* lista, int v) {
  Lista2* p = busca(lista, v);
  if (p == NULL)
     return lista;

  if (lista == p)
        lista = p->prox;
  else
        p->ant->prox = p->prox;
  if (p->prox != NULL)
        p->prox->ant = p->ant;
  free(p);
  return lista;
}
```

Usando a função de busca

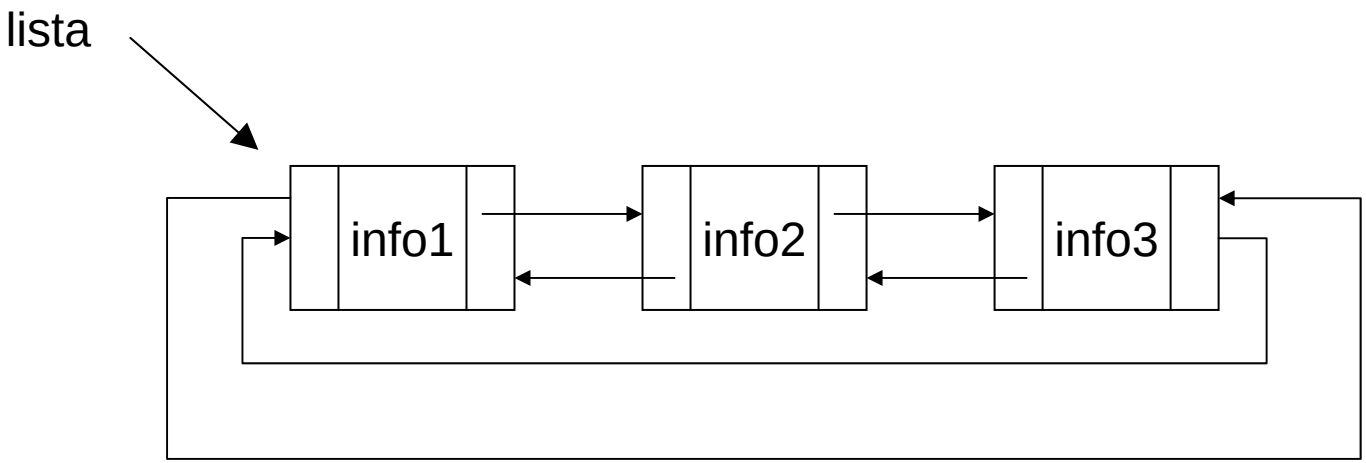
Retira elemento do começo da lista

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;
```

Alocando um Nó de uma Lista de Tipos Estruturados

- Nó da lista contém um retângulo

```
Lista* aloca (float b, float h) {
  Lista *p;
  p = (Lista*)malloc(sizeof(Lista));
  p->info.b = b;
  p->info.h = h;
  p->prox = NULL
  return p;
}
```

Alocando um Nó de uma Lista de Tipos Estruturados

- Nó da lista contém um ponteiro para um retângulo

```
Lista* aloca (float b, float h){
  Retangulo *r;
  r = (Retangulo*) malloc(sizeof(Retangulo));
  Lista* p = (Lista*)malloc(sizeof(Lista));
  r->b = b;
  r->h = h;
  p->info = r;
  p->prox = NULL;
  return p;
}
```

Espaço para a estrutura Retangulo deve também ser alocado

Lista Heterogênea

- São listas cujos nós possuem conteúdos de tipos distintos
  - Ex: uma lista de figuras geométricas (retângulo, círculo, triângulo, etc.).

Representação de Listas Heterogêneas

- Cada nó da lista será então formado por três campos:
  - identificador do tipo de objeto armazenado no nó;
  - ponteiro para a informação;
  - ponteiro para o próximo elemento.

Representação de Listas Heterogêneas

```
 /* definição dos identificadores dos tipos dos
nós  (Retângulo, Triângulo e Circulo) */
#define   RET   0
#define   TRI    1
#define   CIR    2
#define PI 3.1416
/* definição do nó da estrutura */
typedef struct   listahet  {
     int        tipo ;

     void    *info ; /* Ponteiro Genérico */

     struct   listahet   *prox  ;
} ListaHet;
```

Armazena endereço para qualquer tipo de dado

Exemplo de Lista Heterogênea

- Tipos de estruturas que serão armazenadas na lista heterogênea

```
typedef struct   retangulo  {
    float   b ;
    float   h ;
} Retangulo ;

typedef struct  circulo  {
    float   r ;
} Circulo;

typedef struct    triangulo   {
       float    b ;
       float    h ;
} Triangulo ;
```

Criando Nós com Retângulos

```
ListaHet* cria_ret (float b,float h ) {
      Retangulo *r;
      ListaHet  *p;
      /* aloca retângulo */
      r = (Retangulo*) malloc(sizeof(Retangulo)) ;
      r->b  =  b ;
      r->h  =  h ;
     /* aloca nó */
      p = (ListaHet*) malloc(sizeof(ListaHet)) ;
      p->tipo  =  RET ;
      p->info  =  r ;
      p->prox  =  NULL ;
      return    p ;
}
```

Atribuição do tipo de estrutura cujo endereço será armazenado no nó da lista

Criando Nós com Triângulos

```
ListaHet*   cria_tri (float b, float h)  {
      Triangulo *t ;
      ListaHet  *p ;
      /* aloca   triângulo  */
      t = (Triangulo*) malloc(sizeof(Triangulo)) ;
      t->b  =  b ;
      t->h  =  h ;
      /*  aloca   nó  */
      p = (ListaHet*) malloc(sizeof(ListaHet));
      p->tipo  =  TRI ;
      p->info  =   t ;
      p->prox  =  NULL ;
      return   p ;
}
```

Se o tipo for colocado errado, funções que manipulam a lista terão comportamentos imprevisíveis!

Criando Nós com Círculos

```
ListaHet*  cria_cir (float r){
      Circulo *c ;
      ListaHet *p ;
      /* aloca círculo */
       c = (Circulo*) malloc(sizeof(Circulo)) ;
       c->r  =  r ;
      /* aloca nó */
       p = (ListaHet*) malloc(sizeof(ListaHet)) ;
       p->tipo  =  CIR ;
       p->info  =  c ;
       p->prox  =  NULL ;
       return    p ;
}
```

Exemplo de Lista Heterogênea

- Uma vez criados os nós, podemos inseri-los na lista da mesma forma que nas listas definidas anteriormente.
- As constantes simbólicas que representam os tipos de objetos nos nós, podem ser agrupadas em uma enumeração:

```
enum  { RET  ,  TRI  ,  CIR }  ;
```

Calculando a Área

```
float area (ListaHet *p)  {
   float    a ;  /* área do elemento */
   switch (p->tipo)  {
     case   RET :
            Retangulo *r = (Retangulo*) p-> info ;
            a = r-> b * r-> h ;
            break  ;
     case   TRI  :
            Triangulo *t = (Triangulo*) p->info ;
            a = (t->b * t->h)/2 ;
            break ;
     case   CIR:
            Circulo *c = (Circulo *) p-> info ;
            a  =  PI * c->r * c->r ;
            break  ;
   }
   return   a ;
}
```

Converte p/ o tipo e calcula a área

Calculando o Elemento que Possui Maior Área

```
float    max_area   (  ListaHet   *lista  ){
      float   ar, amax = 0.0 ;
      ListaHet    *p ;
      for( p = lista; p != NULL; p =  p->prox )  {
            ar = area(p) ;  /* área do nó */
            if (ar > amax)
              amax  =  ar ;
      }
      return    amax  ;
}
```