DEPARTAMENTO DE INFORMÁTICA
PUC·RIO

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

INF 1010 – Estrutura de Dados Avançadas

Aula 03 – Revisão de Listas Encadeadas

2020.1

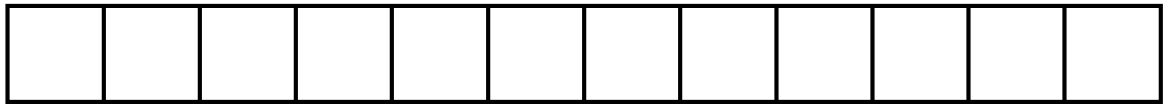
Prof. Augusto Baffa
<abaffa@inf.puc-rio.br>

LES ECOA PUCRIO

ICAD GAMES visionlab visualization laboratory

Introdução

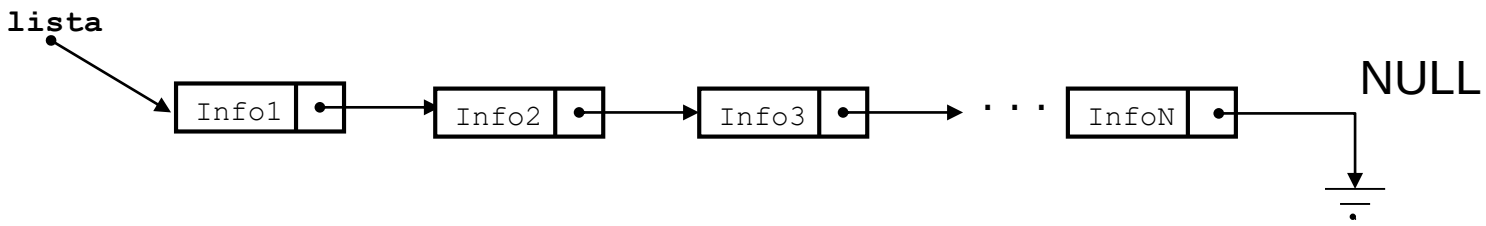
- **Vetores:**
  - Ocupa um espaço contíguo de memória;
  - Permite acesso randômico;
  - Requer pré-dimensionamento de espaço de memória;



- **Estruturas de Dados Dinâmicas:**
  - Crescem (ou decrescem) à medida que elementos são inseridos (ou removidos);
  - Exemplo: Listas Encadeadas;
  - Outras estruturas: pilhas, filas...

Listas Encadeadas

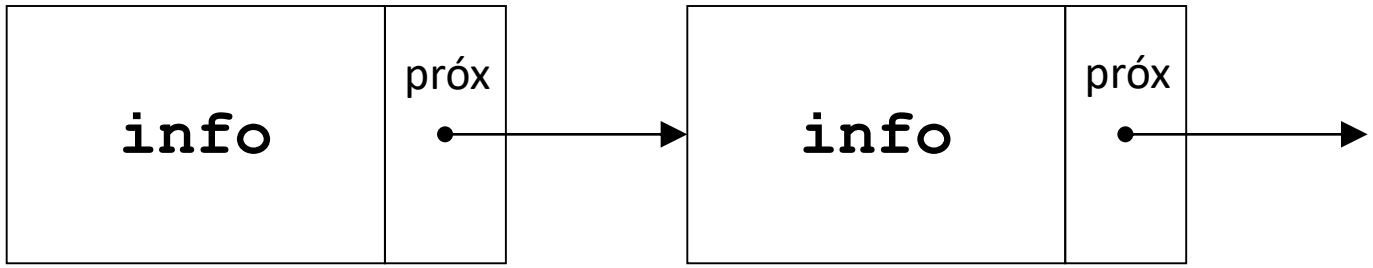
- Uma **Lista Encadeada** é uma sequência de elementos, onde cada elemento tem uma informação armazenada e um ponteiro para o próximo elemento da sequência:



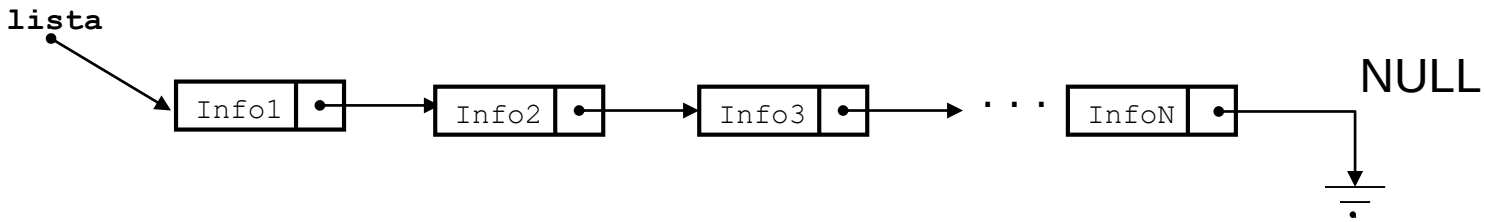
  - sequência encadeada de elementos, chamados de nós da lista;
  - nó da lista é representado por dois campos:
    - informação armazenada;
    - ponteiro para o próximo elemento da lista;
  - a lista é representada por um ponteiro para o primeiro nó;
  - o ponteiro do último elemento é NULL;

Listas Encadeadas

```
struct elemento
{
    int info;
    struct elemento *prox;
};
typedef struct elemento Elemento;
```



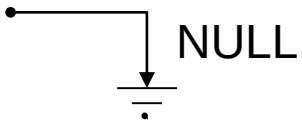
Listas Encadeadas



- Operações em listas encadeadas:
  - Criação;
  - Inserção;
  - Impressão;
  - Teste de vazia;
  - Busca;
  - Remover um elemento;
  - Libera a lista;
  - Manter lista ordenada;
  - Igualdade;

Listas Encadeadas - Criação

```
/* função de criação: retorna uma lista vazia */
Elemento* lista_cria (void)
{
   return NULL;
}
```



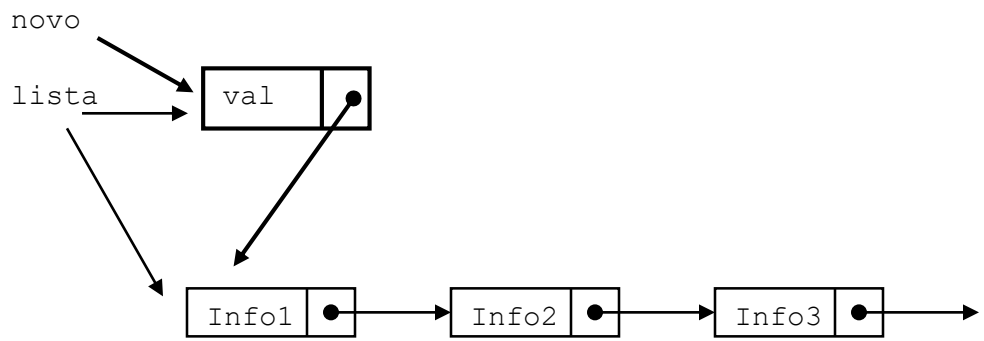
Cria uma lista vazia, representada pelo ponteiro NULL

Listas Encadeadas - Inserção

```
/* inserção no início: retorna a lista atualizada */
Elemento* lista_insere (Elemento* lista, int val)
{
    Elemento* novo = (Elemento*) malloc(sizeof(Elemento));
   novo->info = val;
   novo->prox = lista;
   return novo;
}
```

1. Aloca memória para armazenar o elemento;

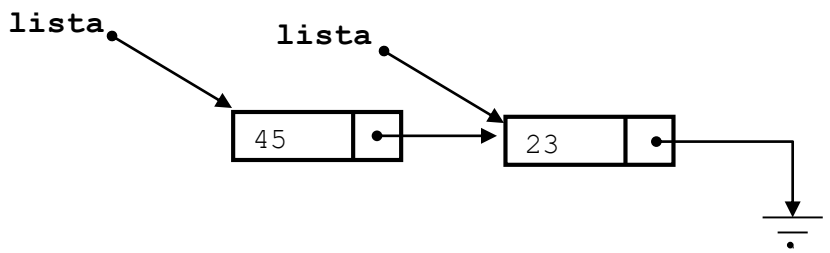
2. Encadeia o elemento na lista existente;



Listas Encadeadas - Exemplo

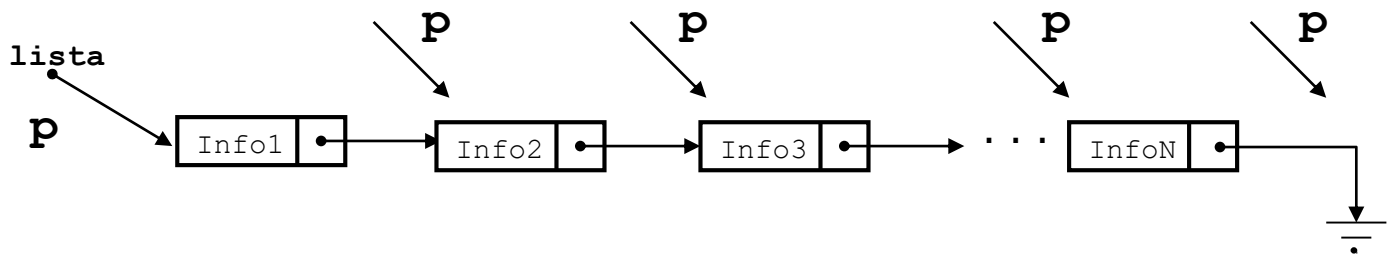
```
int main (void)
{
   Elemento* lista;              /* declara uma lista não inicializada */
⇨  lista = lista_cria();         /* cria e inicializa lista como vazia */

⇨  lista = lista_insere(lista, 23);/* insere na lista o elemento 23 */
⇨  lista = lista_insere(lista, 45);/* insere na lista o elemento 45 */
   ...
   return 0;
}
```

deve-se atualizar a variável que representa a lista a cada inserção de um novo elemento.



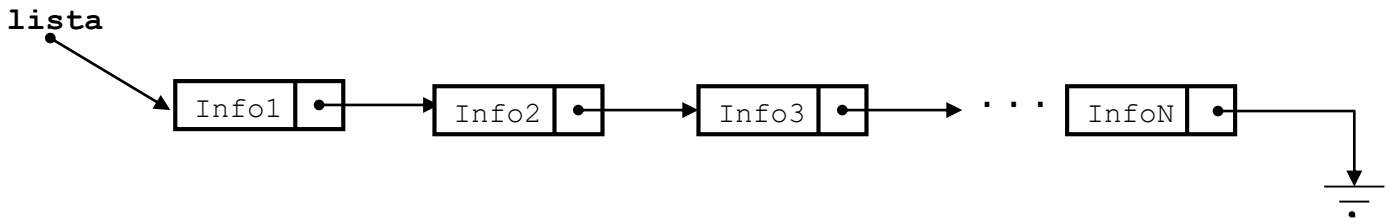
Listas Encadeadas - Impressão

```
/* função imprime: imprime valores dos elementos */
void lista_imprime (Elemento* lista)
{
   Elemento* p;
   for (p = lista; p != NULL; p = p->prox)
      printf("info = %d\n", p->info);
}
```



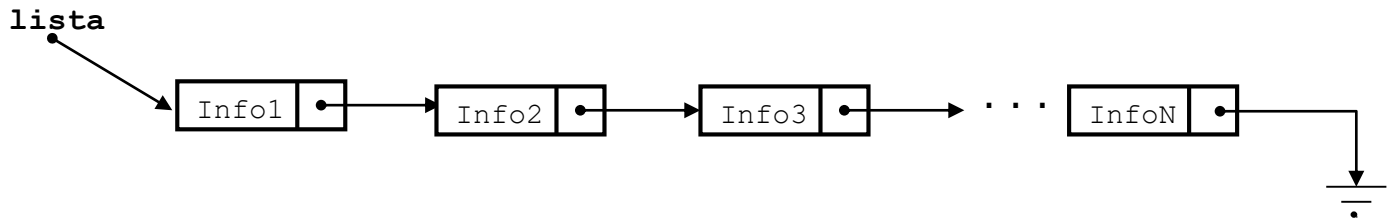
Listas Encadeadas - Teste Vazia

```c
/* função vazia: retorna 1 se vazia ou 0 se não vazia */
int lista_vazia (Elemento* lista)
{
  if (lista == NULL)
    return 1;
  else
    return 0;
}
```



Listas Encadeadas - Busca

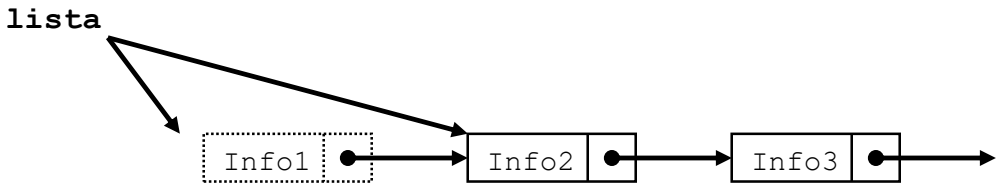
```
/* função busca: busca um elemento na lista */
Elemento* busca (Elemento* lista, int v)
{
   Elemento* p;
   for (p = lista; p != NULL; p = p-> prox)
   {
      if (p->info == v)
         return p;     /* achou o elemento */
   }
   return NULL;        /* não achou o elemento */
}
```



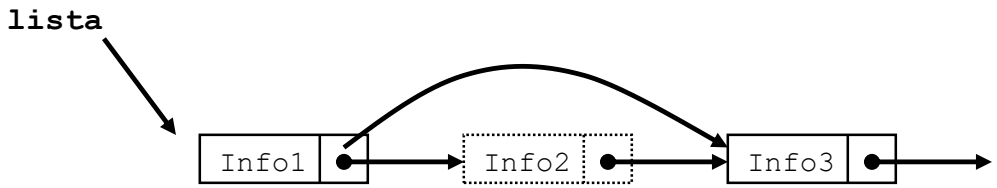
Listas Encadeadas - Remover

- Recebe como entrada a lista e o valor do elemento a retirar

- Se o elemento a ser removido for o primeiro, atualiza o ponteiro da lista:



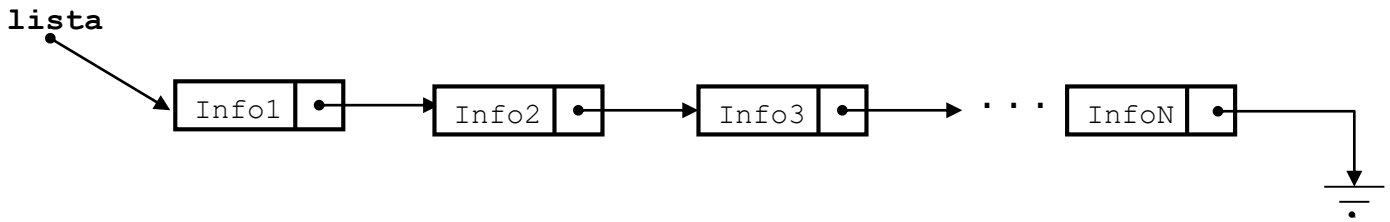
- Caso contrário, apenas remove o elemento da lista:



```c
/* função retira: retira elemento da lista */
Elemento* lista_retira (Elemento* lista, int val)
{
   Elemento* a = NULL;  /* ponteiro para elemento anterior */
   Elemento* p = lista; /* ponteiro para percorrer a lista */

   /* procura elemento na lista, guardando anterior */
   while (p != NULL && p->info != val) {
      a = p;
      p = p->prox;
    }

   /* verifica se achou elemento */
   if (p == NULL)
      return lista;     /* não achou: retorna lista original */

   /* retira elemento */
   if (a == NULL)
      lista = p->prox;  /* retira elemento do inicio */
   else
      a->prox = p->prox;/* retira elemento do meio da lista */

   free(p);
   return lista;
}
```

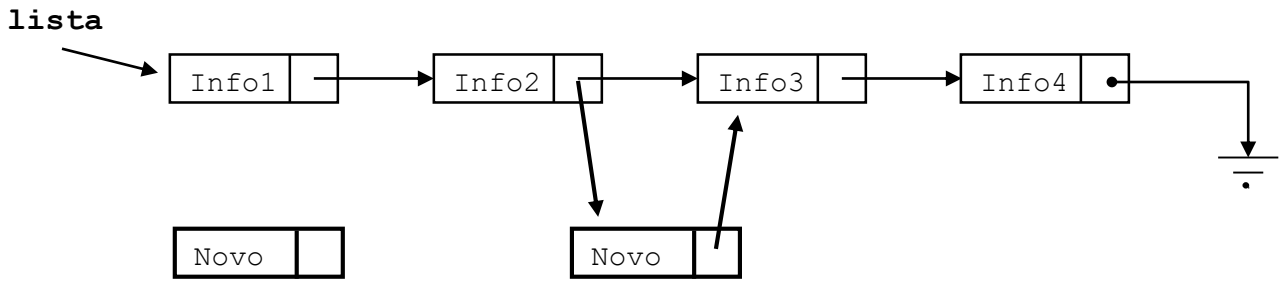
Listas Encadeadas - Libera Lista

```
/* funcao libera: libera todos os elementos alocados da lista */
void lista_libera (Elemento* lista)
{
   Elemento* p = lista;
   Elemento* t;
   while (p != NULL)
   {
      t = p->prox;          /* guarda referência p/ próx. elemento */
      free(p);              /* libera a memória apontada por p     */
      p = t;                /* faz p apontar para o próximo        */
   }
}
```



Listas Encadeadas – Manter Ordenação

- A função de inserção percorre os elementos da lista até encontrar a posição correta para a inserção do novo elemento:



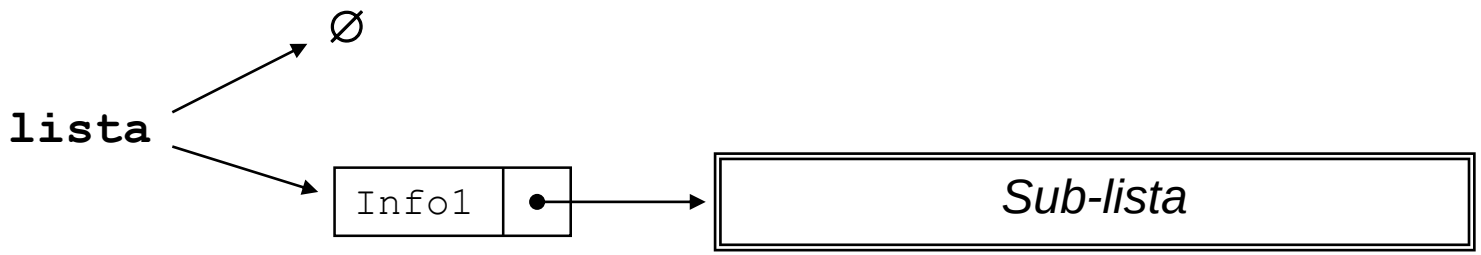
```c
/* função insere_ordenado: insere elemento em ordem */
Elemento* lista_insere_ordenado (Elemento* lista, int val)
{
   Elemento* novo;
   Elemento* a = NULL;   /* ponteiro para elemento anterior  */
   Elemento* p = lista;  /* ponteiro para percorrer a lista  */

   while (p != NULL && p->info < val){
      a = p;                /* procura posição de inserção */
      p = p->prox;
   }
   /* cria novo elemento */
   novo = (Elemento*) malloc(sizeof(Elemento));
   novo->info = val;
   if (a == NULL){
      novo->prox = lista;    /* insere elemento no início */
      lista = novo;
   } else {
      novo->prox = a->prox;  /* insere elemento no meio da lista */
      a->prox = novo;
   }
   return lista;
}
```

Listas Encadeadas - Igualdade

```
/* funcao igualdade: compara se duas listas são iguais */
int lista_igual(Elemento* lista1, Elemento* lista2)
{
  Elemento* p1; /* ponteiro para percorrer lista1 */
  Elemento* p2; /* ponteiro para percorrer lista2 */
  for (p1=lista1, p2=lista2; p1 != NULL && p2 != NULL;
                             p1 = p1->prox, p2 = p2->prox) {
    if (p1->info != p2->info)
      return 0;
  }
  if (p1 == p2) /* se ambos forem NULL as listas são iguais */
    return 1;
  else
    return 0;
}
```

Listas Encadeadas: Definição Recursiva

- Uma lista encadeada é
  - uma lista vazia; ou
  - um elemento seguido de uma (sub-)lista.



Listas Encadeadas- Impressão Recursiva

- Se a lista for vazia
  - Não imprima nada
- Caso contrário,
  - imprima a informação associada ao primeiro nó, dada por **`lista->info`**
  - imprima a sub-lista, dada por **`lista->prox`**, chamando recursivamente a função

```
/* Função imprime recursiva */
void lista_imprime_rec (Elemento* lista)
{
  if(!lista_vazia(lista)) {
    /*imprime primeiro elemento*/
    printf("info: %d\n",lista->info);
    lista_imprime_rec(lista->prox); /*imprime sub-lista*/
  }
}
```

Listas Encadeadas – Impressão Invertida Recursiva

```
/* Função imprime recursiva invertida */
void lista_imprime_rec_inv (Elemento* lista)
{
   if ( !lista_vazia(lista) ) {
      /* imprime sub-lista                */
      lista_imprime_rec_inv(lista->prox);
      /* imprime ultimo elemento  */
      printf("info: %d\n",lista->info);

   }
}
```

Listas Encadeadas – Retira Recursiva

- Retire o elemento, se ele for o primeiro da lista (ou da sub-lista);
- Caso contrário, chame a função recursivamente para retirar da sub-lista.

```
/* Função retira recursiva */
Elemento* lista_retira_rec (Elemento* lista, int val)
{
   if (!lista_vazia(lista)) {
      /* verifica se elemento a ser retirado é o primeiro */
      if (lista->info == val) {
         Elemento* t = lista;/* temporário para poder liberar */
         lista = lista->prox;
         free(t);
      }
      else {
         /* retira de sub-lista */
         lista->prox = lista_retira_rec(lista->prox,val);
      }
   }
   return lista;
}
```

Listas Encadeadas – Igualdade Recursiva

- Se as duas listas dadas são vazias, são iguais;
- Se não forem ambas vazias, mas uma delas é vazia, são diferentes;
- Se ambas não forem vazias, teste:
  - Se informações associadas aos primeiros nós são iguais; e
  - Se as sub-listas são iguais.

```
int lista_igual_rec(Elemento* lista1, Elemento* lista2)
{
   if (lista_vazia(lista1) == 1 && lista_vazia(lista2) == 1)
      return 1;
   else if (lista_vazia(lista1) == 1 || lista_vazia(lista2) == 1)
      return 0;
   else
      return (lista1->info == lista2->info) &&
             lista_igual_rec(lista1->prox, lista2->prox);
}
```

Listas de Tipos Estruturados

- A informação associada a cada nó de uma lista encadeada pode ser mais complexa, sem alterar o encadeamento dos elementos;
- As funções apresentadas para manipular listas de inteiros podem ser adaptadas para tratar listas de outros tipos;
- O campo da informação pode ser representado por um ponteiro para uma estrutura, em lugar da estrutura em si independente da informação armazenada na lista, a estrutura do nó é sempre composta por:
  - um ponteiro para a informação; e
  - um ponteiro para o próximo nó da lista.

Listas de Tipos Estruturados

- Exemplo – Lista Encadeada de Pontos

```
struct ponto {
   float x;
   float y;
};
typedef struct ponto Ponto;

struct elemento {
   Ponto* info;
   struct elemento *prox;
};
typedef struct elemento Elemento;
```

campo da informação representado por um ponteiro para uma estrutura, em lugar da estrutura em si

Listas de Tipos Estruturados

- Exemplo: Inserção em uma lista de pontos

```
/* inserção no início: retorna a lista atualizada */
Elemento* lista_insere(Elemento* lista, float px, float py)
{
   Elemento* p = (Elemento*) malloc(sizeof(Elemento));
   p->info = (Ponto*) malloc(sizeof(Ponto));
   p->info->x = px;
   p->info->y = py;
   p->prox = lista;
   return p;
}
```

Para alocar um nó, são necessárias duas alocações dinâmicas:
uma para criar a estrutura do ponto e outra para criar a estrutura do nó.

O valor de x associado a um nó **p** seria acessado por: **p->info->px.**

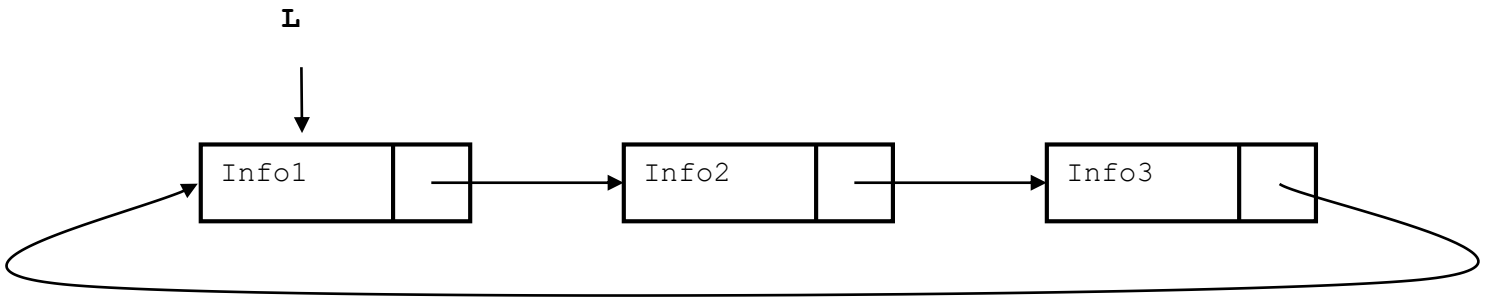
Listas Encadeadas

TÓPICOS COMPLEMENTARES

Listas Circulares

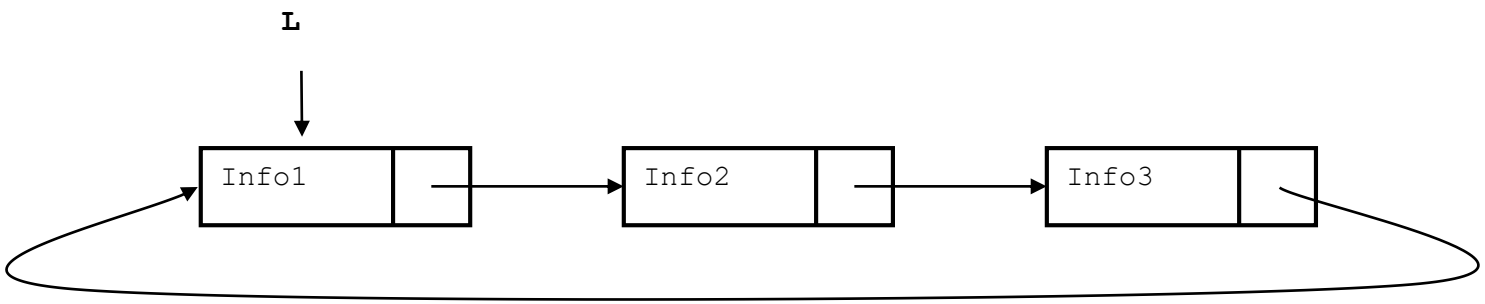
- O último elemento tem como próximo o primeiro elemento da lista, formando um ciclo

- A lista pode ser representada por um ponteiro para um elemento inicial qualquer da lista



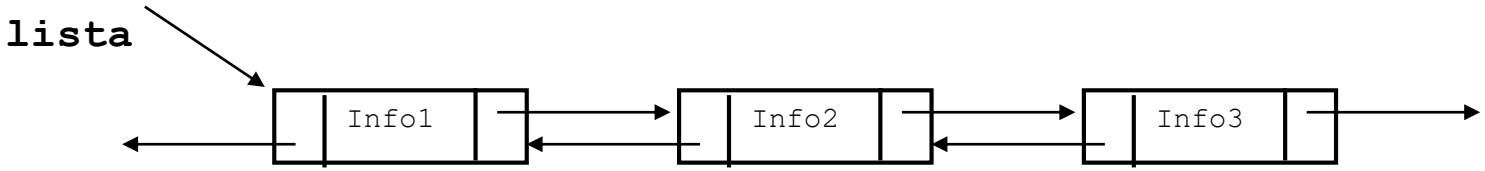
Listas Circulares - Impressão

```
/* função imprime: imprime valores dos elementos */
void listacircular_imprime (Elemento* lista)
{
   Elemento* p = lista;     /* faz p apontar para o nó inicial */
   /* testa se lista não é vazia e então percorre com do-while */
   if (p!=NULL) {
     do {
      printf("%d\n", p->info);   /* imprime informação do nó */
      p = p->prox;               /* avança para o próximo nó */
     } while (p != lista);
   }
}
```



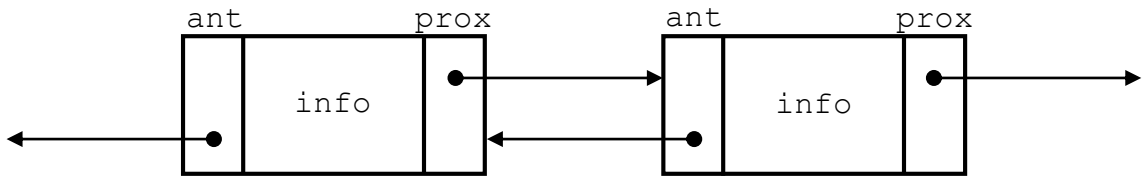
Listas Duplamente Encadeadas

- Cada elemento tem um ponteiro para o próximo elemento e um ponteiro para o elemento anterior;
- Dado um elemento, é possível acessar o próximo e o anterior;
- Dado um ponteiro para o último elemento da lista, é possível percorrer a lista em ordem inversa ;



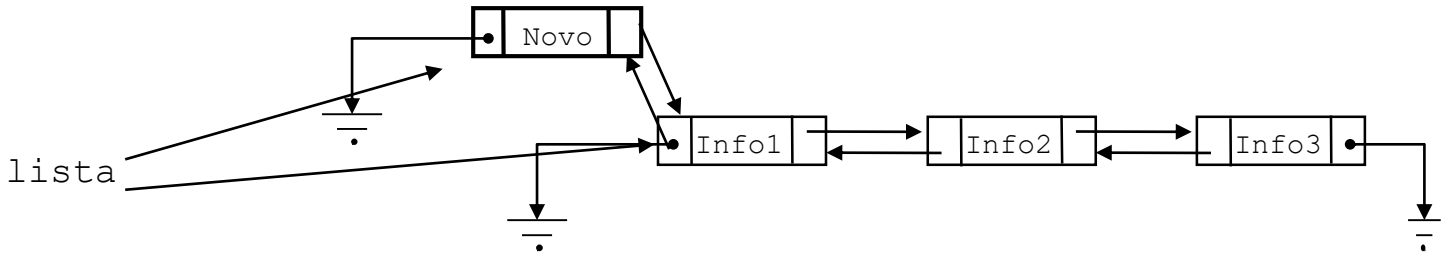
Listas Duplamente Encadeadas

```
struct lista2 {
   int info;
   struct lista2* ant;
   struct lista2* prox;
};
typedef struct lista2 Lista2;
```



Listas Duplamente Encadeadas - Inserção

```
/* inserção no início: retorna a lista atualizada */
Lista2* lista2_insere (Lista2* lista, int val)
{
   Lista2* novo = (Lista2*) malloc(sizeof(Lista2));
   novo->info = val;
   novo->prox = lista;
   novo->ant = NULL;
   /* verifica se lista não estava vazia */
   if (lista != NULL)
      lista->ant = novo;
   return novo;
}
```



Listas Duplamente Encadeadas - Busca

```
/* função busca: busca um elemento na lista */
Lista2* lista2_busca (Lista2* lista, int val)
{
   Lista2* p;
   for (p=lista; p!=NULL; p=p->prox)
      if (p->info == val)
         return p;
   return NULL;        /* não achou o elemento */
}
```

Listas Duplamente Encadeadas - Remover

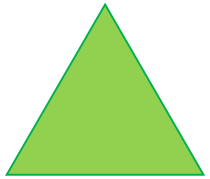
```
/* função retira: remove elemento da lista */
Lista2* lista2_retira (Lista2* lista, int val)
{
  Lista2* p = busca(lista,val);

  if (p == NULL)
   return lista; /*não achou o elemento: retorna lista inalterada*/

  /* retira elemento do encadeamento */
  if (lista == p)              /* testa se é o primeiro elemento */
    lista = p->prox;
  else
    p->ant->prox = p->prox;

  if (p->prox != NULL)         /* testa se é o último elemento */
    p->prox->ant = p->ant;

  free(p);

  return lista;
}
```

Listas Heterogêneas

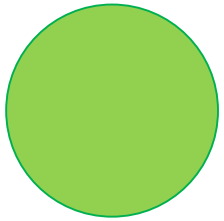
- A representação da informação por um ponteiro permite construir listas heterogêneas, isto é, listas em que as informações armazenadas diferem de nó para nó;

- Exemplo: Listas de retângulos, triângulos ou círculos. Áreas desses objetos são dadas por:

$$r = b * h$$



$$t = \frac{b * h}{2}$$



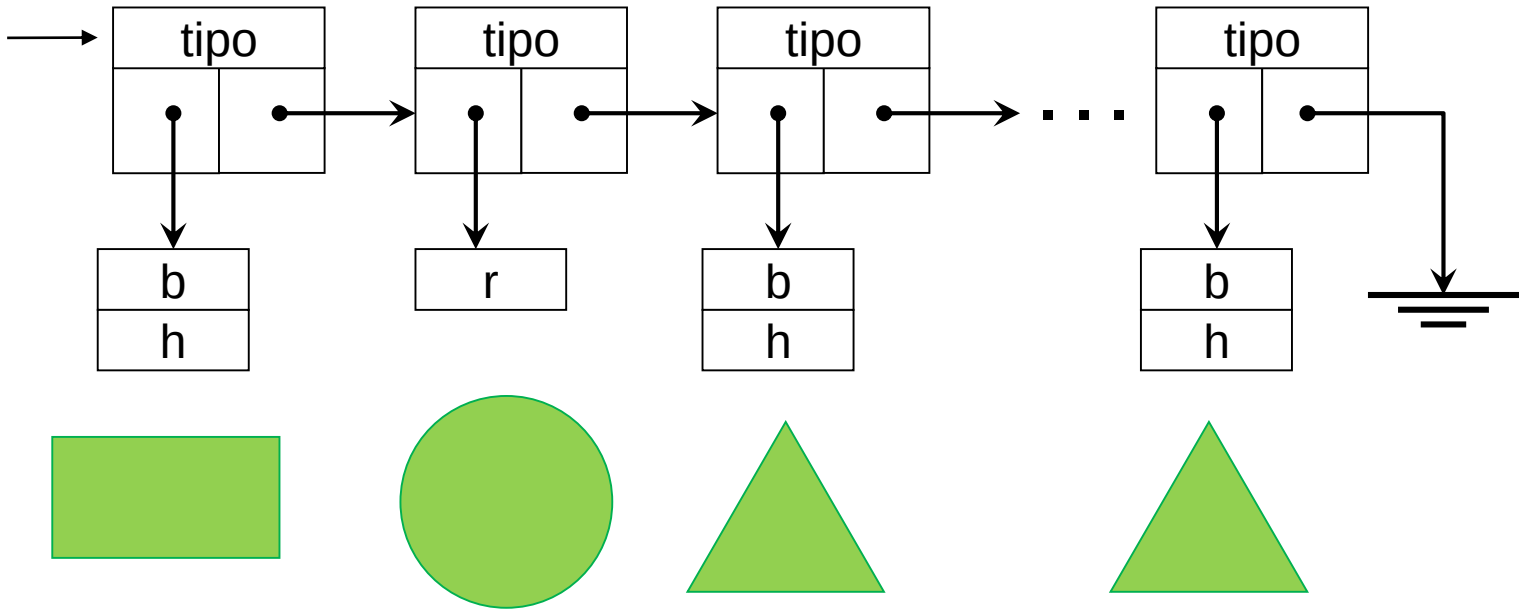
$$c = \pi\ r^2$$

Listas Heterogêneas

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

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

struct circulo {
   float r;
};
typedef struct circulo Circulo;
```

Lista homogênea de objetos heterogêneos

- A lista é homogênea - todos os nós contêm os mesmos campos:
  - um ponteiro para o próximo nó da lista
  - um ponteiro para a estrutura que contém a informação
  - um identificador indicando qual objeto o nó armazena



Lista homogênea de objetos heterogêneos

```
/* Definição dos tipos de objetos */
#define RET 0
#define TRI 1
#define CIR 2

/* Definição do nó da estrutura */
struct lista_het {
   int tipo;
   void *info;
   struct lista_het *prox;
};
typedef struct listahet ListaHet;
```

Lista homogênea de objetos heterogêneos - Inserção

```
/* Cria um nó com um retângulo */
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;
}
```

A função para a criação de um nó possui três variações, uma para cada tipo de objeto

Lista homogênea de objetos heterogêneos – Cálculo da Área

```
/* função para cálculo da área de um retângulo */
static float ret_area (Retangulo* r)
{
   return r->b * r->h;
}

/* função para cálculo da área de um triângulo */
static float tri_area (Triangulo* t)
{
   return (t->b * t->h) / 2;
}

/* função para cálculo da área de um círculo */
static float cir_area (Circulo* c)
{
   return PI * c->r * c->r;
}
```

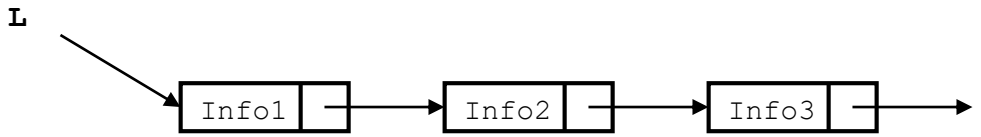
Lista homogênea de objetos heterogêneos – Cálculo da Área

```
/* função para cálculo da área do nó (versão 2) */
static float area (ListaHet* p)
{
   float a;
   switch (p->tipo) {
      case RET:
         a = ret_area((Retangulo*)p->info);
      break;
      case TRI:
         a = tri_area((Triangulo*)p->info);
      break;
      case CIR:
         a = cir_area((Circulo*)p->info);
      break;
   }
   return a;
}
```

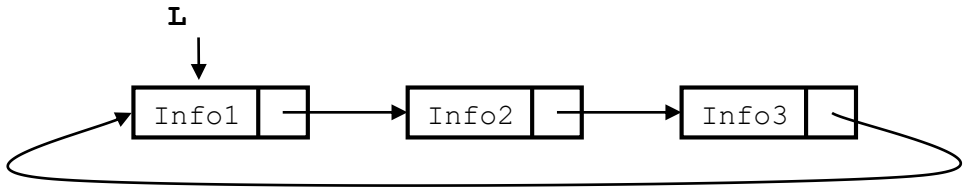
a conversão de ponteiro genérico para ponteiro específico ocorre quando uma das funções de cálculo da área é chamada:
passa-se um ponteiro genérico, que é atribuído a um ponteiro específico, através da conversão implícita de tipo

Resumo

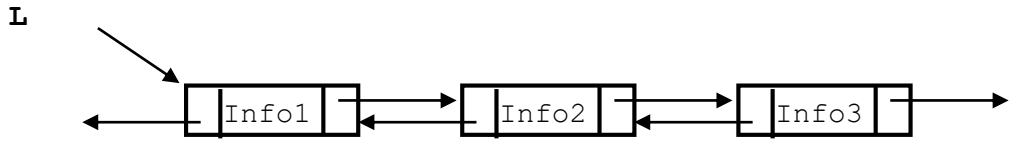
- Listas encadeadas



- Listas circulares



- Listas duplamente encadeadas



Revisão Linguagem C

VAMOS EXERCITAR?

Exercício Aula 2
Tipo de Dados Abstrato c/ Busca

- **Exemplo:** Criar um Tipo de Dados Abstrato para a estrutura de dados de alunos.
  - Deverá ser organizada em um vetor de ponteiros.
  - Utilizando a estrutura de dados abaixo.
  - Criar as funções para inserção, deleção, atualização e busca p/ nome
    - A busca pode ser linear ou binária (indique qual foi a escolhida)

  - Estrutura de dados de cada aluno:
    - matrícula: número inteiro
    - nome: cadeia com até 80 caracteres
    - endereço: cadeia com até 120 caracteres
    - telefone: cadeia com até 20 caracteres

- Obs: Não é necessário criar uma interface para coletar os dados que serão inseridos para testar a tabela com os dados. Podem ser inseridos programaticamente (hard-coded).

Exercício Aula 3
Lista Encadeada

- **Exemplo:** Criar um Tipo de Dados Abstrato para a estrutura de dados de alunos.
  - Deverá ser organizada em uma **lista encadeada**.
  - Utilizando a estrutura de dados abaixo.
  - Criar todas as funções necessárias palista e uma para busca p/ nome
    - Criação;
    - Inserção (Manter lista ordenada);
    - Atualização (Manter lista ordenada);
    - Remover um elemento (Manter lista ordenada);
    - Impressão;
    - Busca Binária;
      - Função Comparação
    - Igualdade (retorna verdadeiro ou falso);
    - Teste de vazia;
    - Libera a lista;

  - Estrutura de dados de cada aluno:
    - matrícula: número inteiro
    - nome: cadeia com até 80 caracteres
    - endereço: cadeia com até 120 caracteres
    - telefone: cadeia com até 20 caracteres

- Obs: Não é necessário criar uma interface para coletar os dados que serão inseridos para testar a tabela com os dados. Podem ser inseridos programaticamente (hard-coded).

Leitura Complementar

- Celes, W., Cerqueira, R., Rangel, J.L., **Introdução a Estruturas de Dados – Uma introdução com técnicas de programação em C**, Ed. Campus, 2004

  - **Capítulo 10 – Listas Encadeadas**