



INF 1010

Estruturas de Dados Avançadas

Revisão de C - Listas

listas

revisão

Motivação

vetor

ocupa um espaço contíguo de memória

permite acesso randômico aos elementos

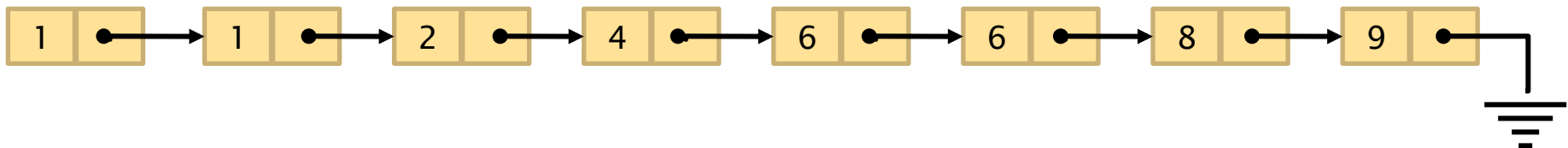
deve ser dimensionado com um número máximo de elementos



estrutura dinâmica

cresce (ou decresce) à medida que elementos são inseridos (ou removidos)

Ex.: listas encadeadas -
amplamente usadas para
implementar outras estruturas
de dados



Lista Encadeada

seqüência encadeada de elementos, chamados de nós da lista

nó da lista é representado por dois campos:

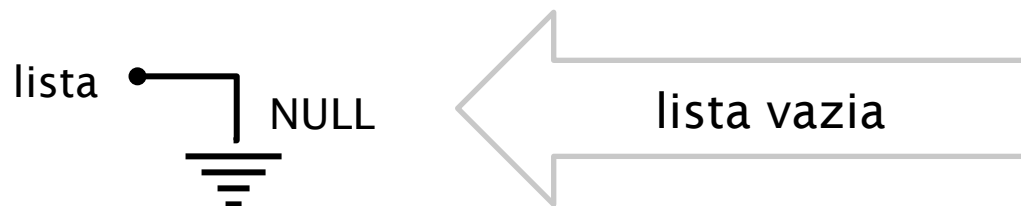
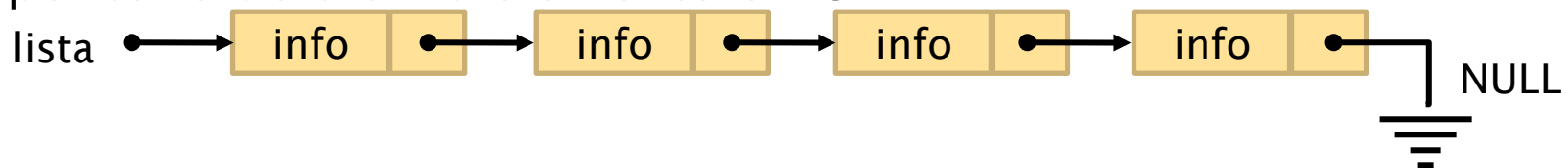
a **informação** armazenada e



o **ponteiro** para o próximo elemento da lista

a lista é representada por um ponteiro para o primeiro nó

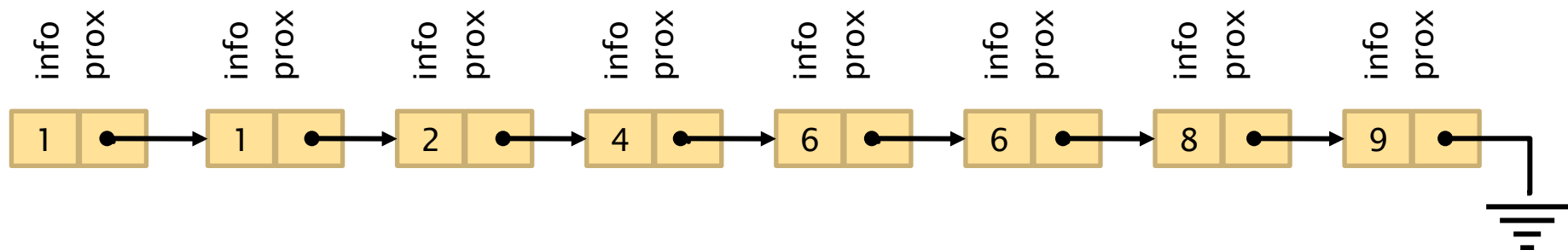
o ponteiro do último elemento é NULL



Estrutura com ponteiro para ela mesma

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

```
/* declaração e inicialização da lista */  
Lista *lst = NULL;
```



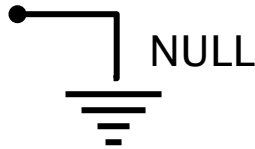
Exemplo: Lista de inteiros (outra forma)

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

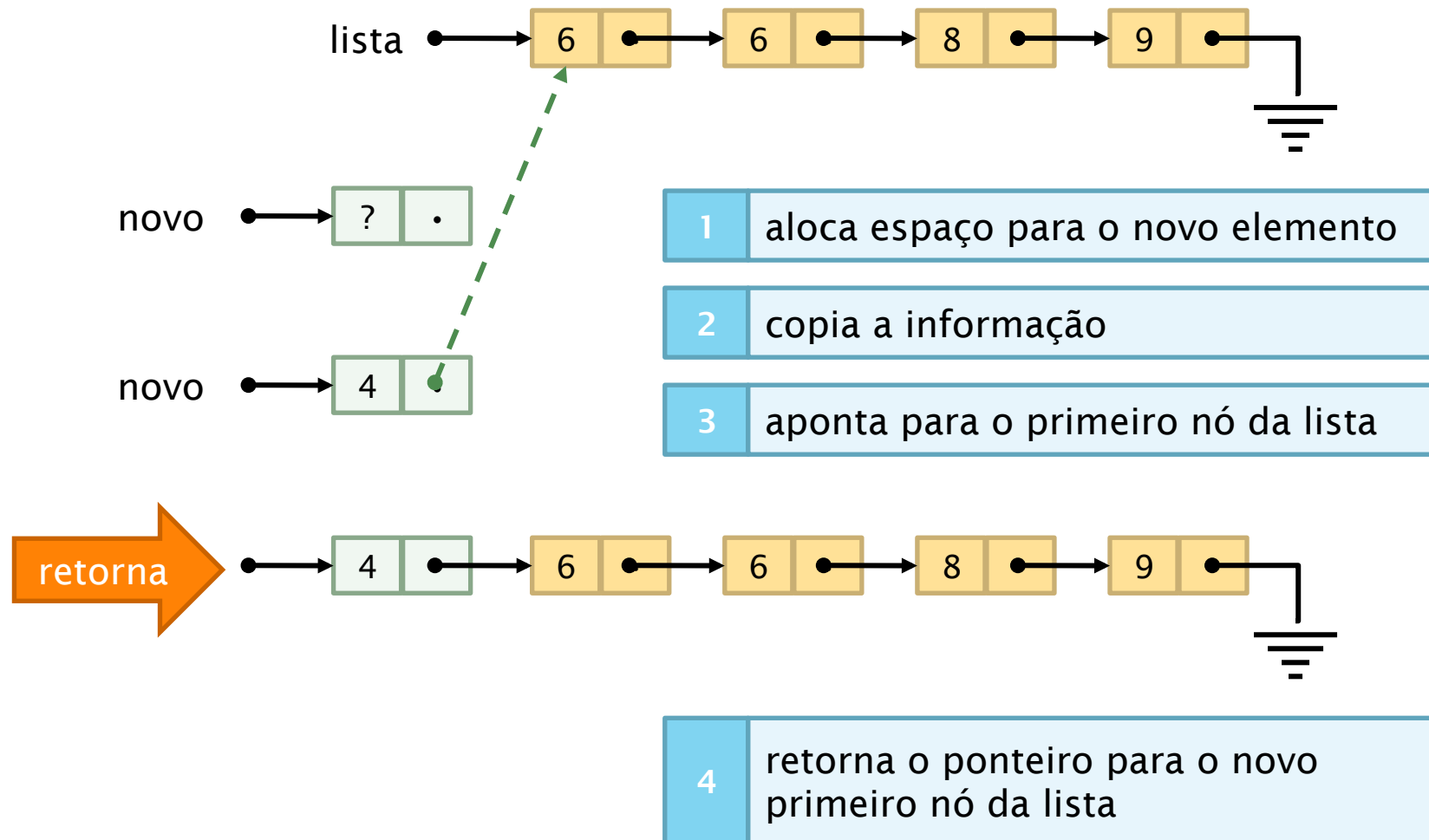
```
typedef struct lista Lista;  
  
struct lista {  
    int info;  
    Lista* prox;  
};
```

Lista encadeada de inteiros: Criação

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



Lista encadeada de inteiros: Inserção



Lista encadeada de inteiros: Inserção

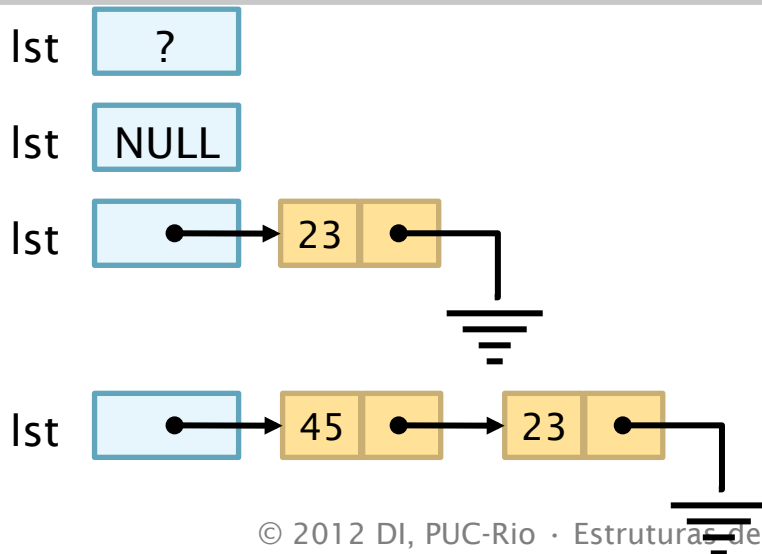
```
/* inserção no início: retorna a lista atualizada
*/
Lista* lst_insere (Lista* lst, int val)
{
    Lista* novo = (Lista*) malloc(sizeof(Lista));
    novo->info = val;
    novo->prox = lst;
    return novo;
}
```

1	aloca espaço para o novo elemento
2	copia a informação
3	aponta para o primeiro nó da lista
4	retorna o ponteiro para o novo primeiro nó da lista

Lista encadeada de inteiros: Exemplo

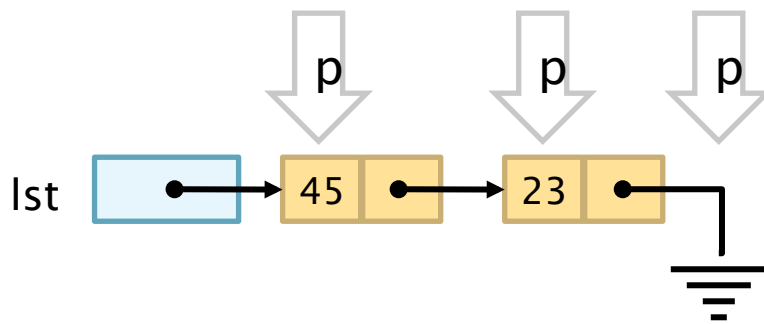
```
int main (void)
{
    Lista* lst;      /* declara lista não inicializada */
    lst = lst_cria(); /* cria e inicializa lista vazia */

    lst = lst_insere(lst, 23); /* insere o elemento 23 */
    lst = lst_insere(lst, 45); /* insere o elemento 45 */
    ...
    return 0;
}
```



Lista encadeada de inteiros: Impressão

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



```
info = 45  
info = 23  
fim
```

Lista encadeada de inteiros: Teste de vazia

Retorna 1, se a lista estiver vazia ou
0, caso contrário

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

Lista encadeada de inteiros: Busca

recebe a informação referente ao elemento a pesquisar

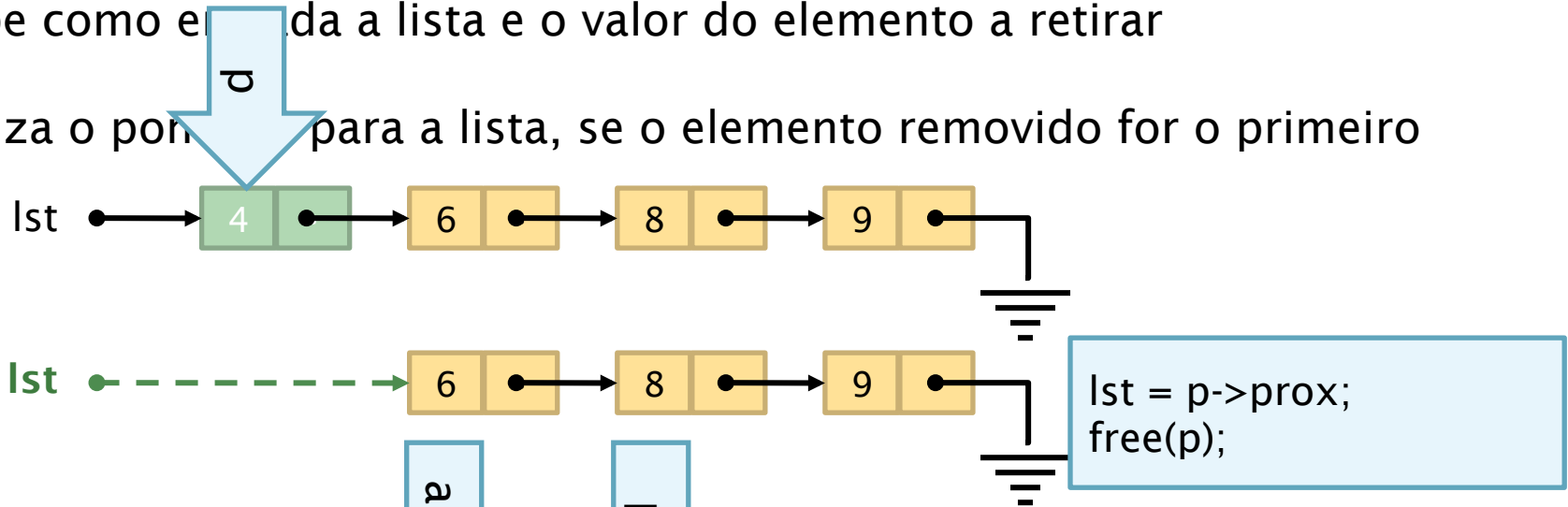
retorna o ponteiro do nó da lista que representa o elemento, ou NULL, caso o elemento não seja encontrado na lista

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

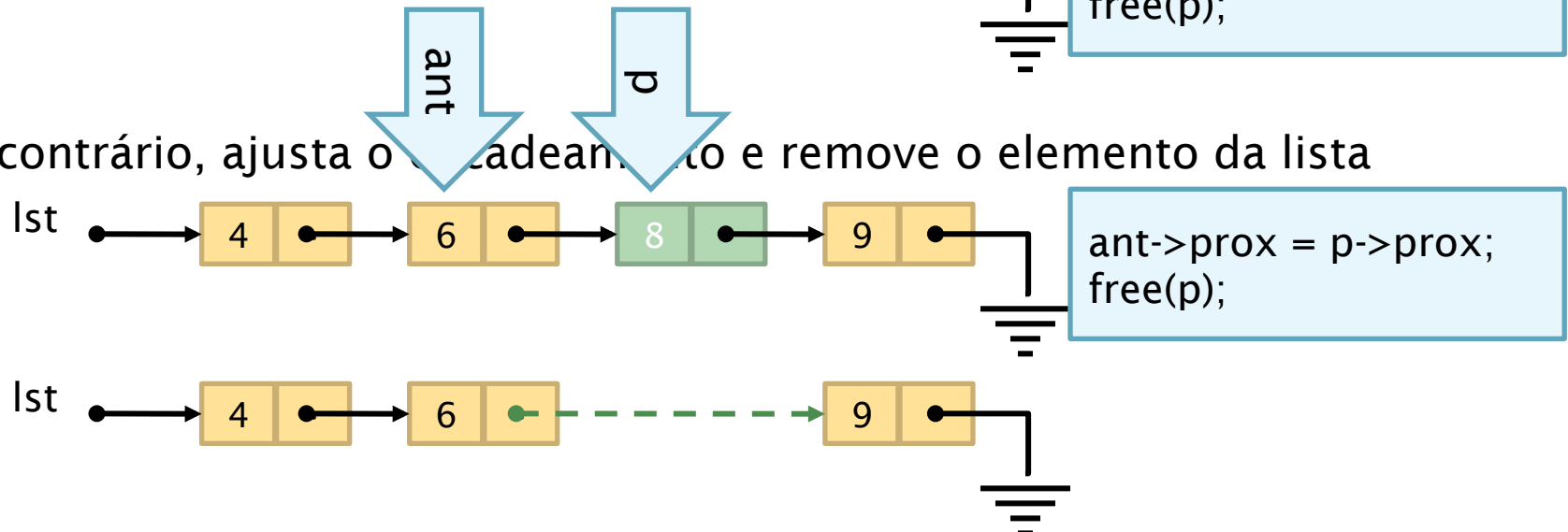
Lista encadeada de inteiros: Remoção

recebe como entrada a lista e o valor do elemento a retirar

atualiza o ponteiro para a lista, se o elemento removido for o primeiro



caso contrário, ajusta o encadeamento e remove o elemento da lista

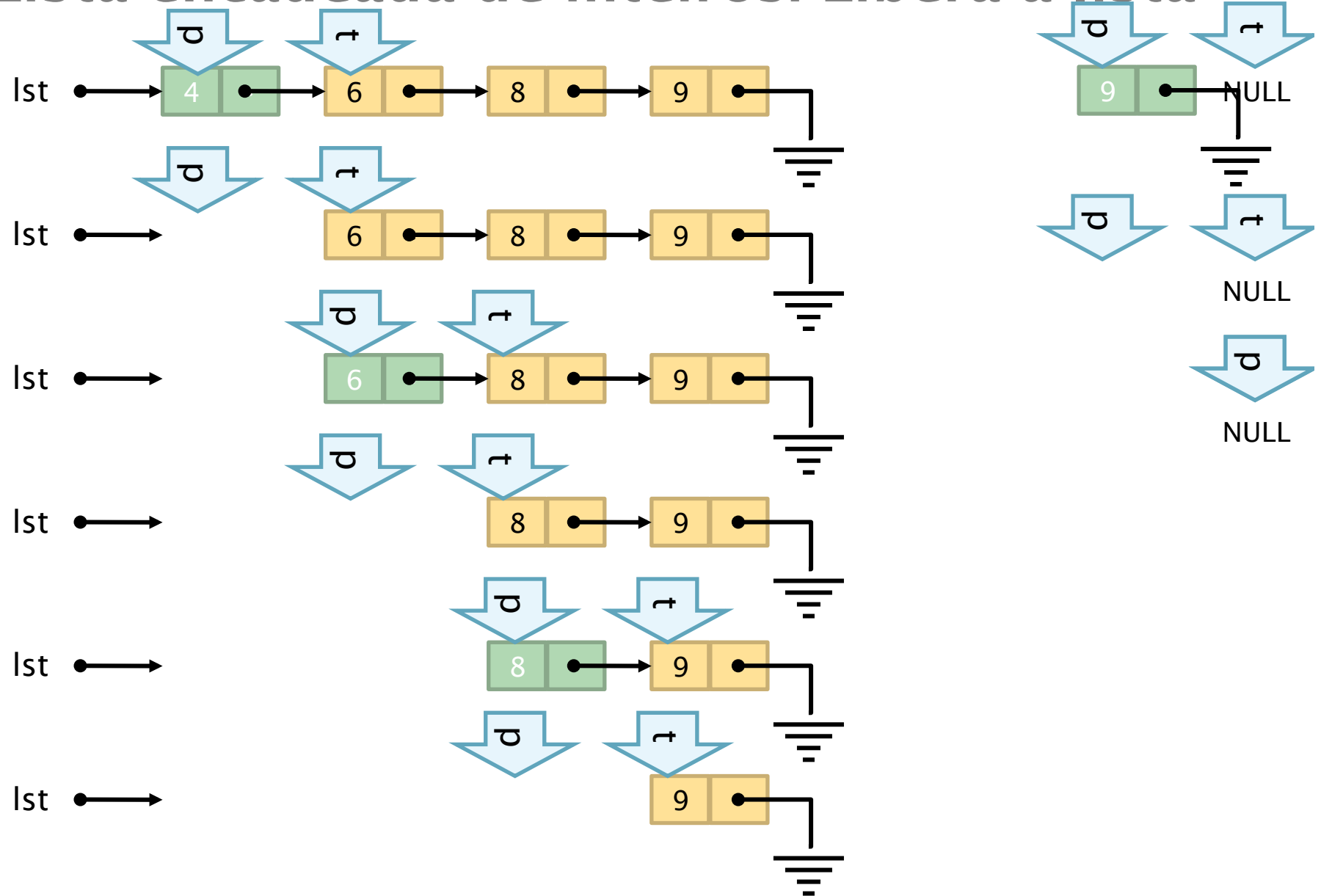


```

/* retira: retira elemento da lista */
Lista* lst_retira (Lista* lst, int val)
{
    Lista* ant = NULL;    /* ponteiro para elemento anterior */
    Lista* p = lst;       /* ponteiro para percorrer a lista */
    /* procura elemento na lista, guardando anterior */
    while (p != NULL && p->info != val) {
        ant = p;
        p = p->prox;
    }
    /* verifica se achou elemento */
    if (p == NULL)
        return lst;      /* não achou: retorna lista original */
    /* achou: retira */
    if (ant == NULL)      /* retira elemento do inicio */
        lst = p->prox;
    else                  /* retira elemento do meio da lista */
        ant->prox = p->prox;
    free(p);              /* libera espaço ocupado pelo elemento */
    return lst;
}

```

Lista encadeada de inteiros: Libera a lista



Lista encadeada de inteiros: Libera a lista

destrói a lista, liberando todos os elementos alocados

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

TAD Lista encadeada de inteiros

```
/* TAD: lista de inteiros */

typedef struct lista Lista;

Lista* lst_cria (void);
void lst_libera (Lista* lst);

Lista* lst_insere (Lista* lst, int val);
Lista* lst_retira (Lista* lst, int val);

int lst_vazia (Lista* lst);

Lista* lst_busca (Lista* lst, int val);

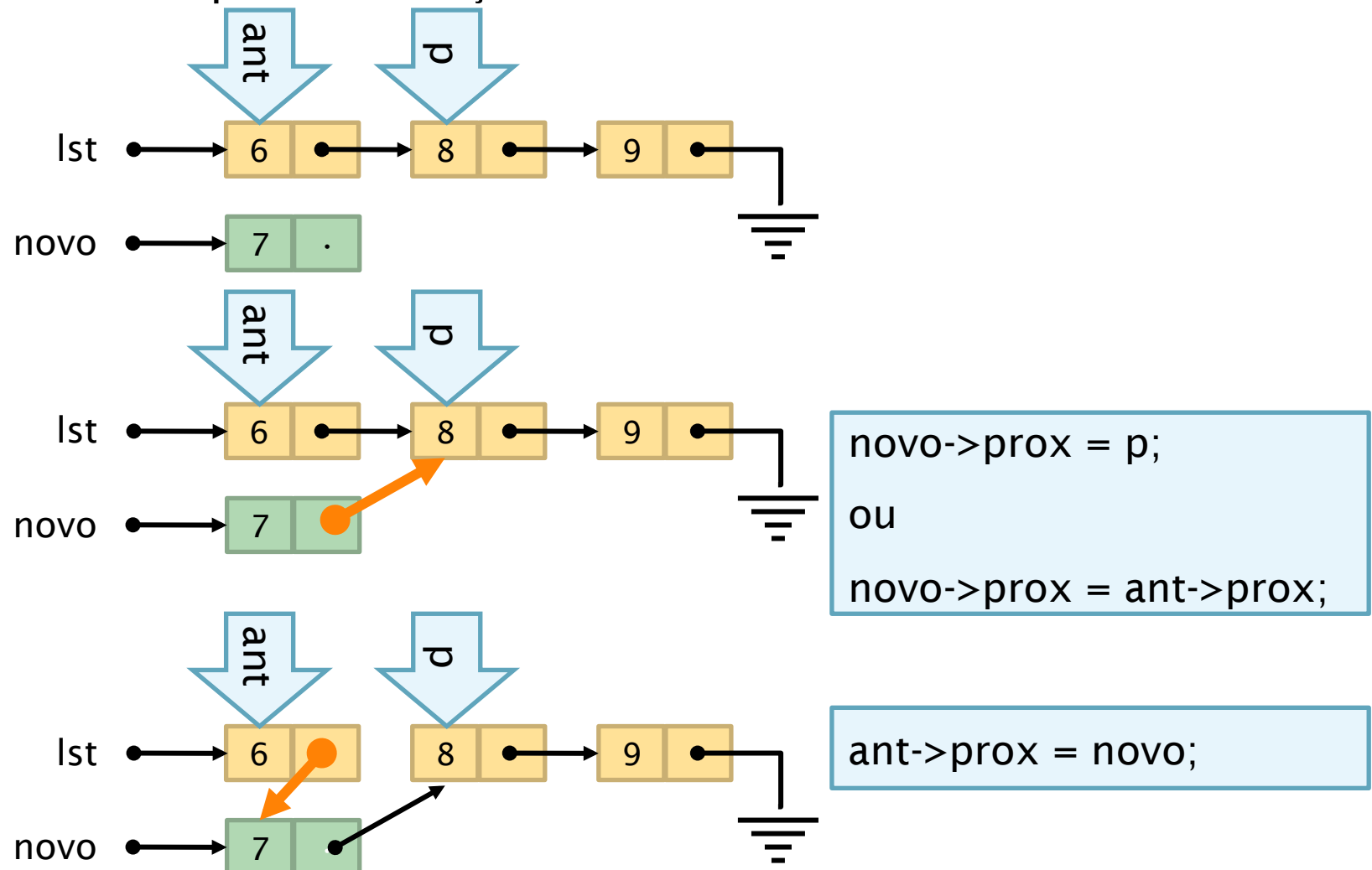
void lst_imprime (Lista* lst);
```

TAD Lista: Exemplo

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

Listas Encadeadas Ordenadas

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



```

/* insere_ordenado: insere elemento em ordem */
Lista* lst_insere_ordenado (Lista* lst, int val)
{
    Lista* novo;
    Lista* ant = NULL;    /* ponteiro para elemento anterior */
    Lista* p = lst;       /* ponteiro para percorrer a lista */
    /* procura posição para inserção */
    while (p != NULL && p->info < val) {
        ant = p;
        p = p->prox;
    }
    /* cria novo elemento */
    novo = (Lista*) malloc(sizeof(Lista));
    novo->info = val;
    /* encadeia elemento */
    if (ant == NULL) {          /* insere elemento no início */
        novo->prox = lst;
        lst = novo;
    }
    else {                     /* insere elemento no meio da lista */
        novo->prox = ant->prox;
        ant->prox = novo;
    }
    return lst; /* retorna ponteiro para o primeiro elemento */
}

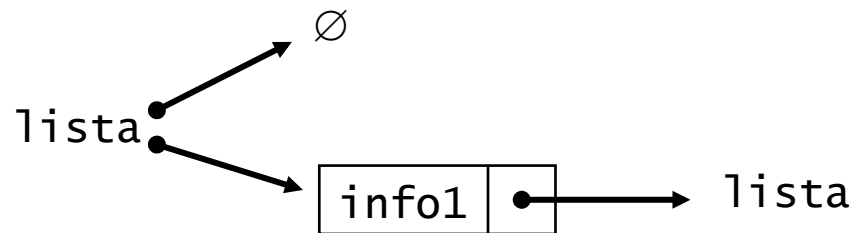
```

Definição recursiva de lista

uma lista é

uma lista vazia; ou

um elemento seguido de uma sublista



Exemplo 1 – Função recursiva para imprimir uma lista

se a lista for vazia, não imprima nada

caso contrário,

1. imprima a informação associada ao primeiro nó, dada por **lst->info**
2. imprima a sublista, dada por **lst->prox**, chamando recursivamente a função

```
/* função imprime recursiva */  
void lst_imprime_rec (Lista* lst)  
{  
    if (!lst_vazia(lst)) {  
        /* imprime primeiro elemento */  
        printf("info: %d\n", lst->info);  
        /* imprime sub-lista */  
        lst_imprime_rec(lst->prox);  
    }  
}
```

Exemplo 2 – Função recursiva para imprimir invertido

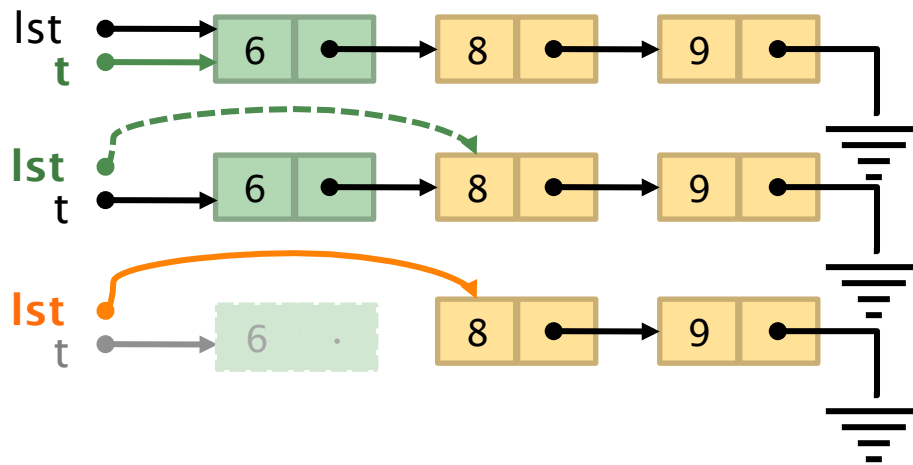
```
/* função imprime recursiva original */
void lst_imprime_rec (Lista* lst)
{
    if (!lst_vazia(lst)) {
        /* imprime primeiro elemento */
        printf("info: %d\n", lst->info);
        /* imprime sub-lista */
        lst_imprime_rec(lst->prox);
    }
}
```

```
/* função imprime recursiva invertida */
void lst_imprime_rec_inv (Lista* lst)
{
    if (!lst_vazia(lst) ) {
        /* imprime sub-lista */
        lst_imprime_rec_inv(lst->prox);
        /* imprime ultimo elemento */
        printf("info: %d\n", lst->info);
    }
}
```


Exemplo 3 – Lista* lst_retira_rec (Lista* lst, int val)

Função para retirar um elemento da lista

se o elemento for o primeiro da lista (ou da sublista), retire-o



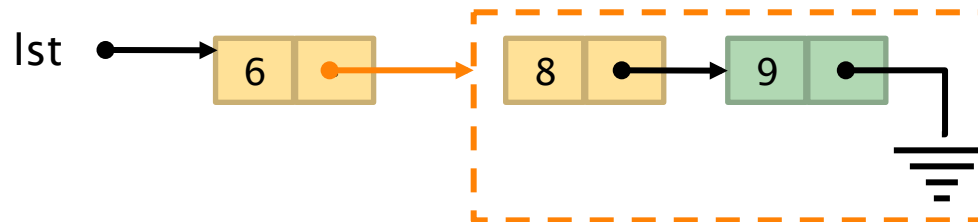
```
t = lst;
```

```
lst = lst->prox;
```

```
free(t);
```

```
return lst;
```

caso contrário, chame a função recursivamente para retirar o elemento da sublista



```
lst->prox = lst_retira_rec(  
    lst->prox, 9);
```

Lista: Retira recursiva

```
/* Função retira recursiva */
Lista* lst_retira_rec (Lista* lst, int val)
{
    if (!lst_vazia(lst)) {
        /* verifica se elemento a ser retirado é o primeiro */
        if (lst->info == val) {
            Lista* t = lst;
            lst = lst->prox;
            free(t);
        }
        else {
            /* retira de sub-lista */
            lst->prox = lst_retira_rec(lst->prox, val);
        }
    }
    return lst;
}
```

Por que t é necessário?

Lista: Igualdade de listas

```
int lst_igual (Lista* lst1, Lista* lst2);
```

retorna 1 se as listas forem iguais; 0 caso contrário

iterativa

percorre as duas listas, usando dois ponteiros auxiliares:

se duas informações forem diferentes, as listas são diferentes

ao terminar uma das listas (ou as duas):

se os dois ponteiros auxiliares são NULL, as duas listas têm o mesmo número de elementos e são iguais

caso contrário, são diferentes

recursiva

se as duas listas dadas são vazias, são iguais

se apenas uma delas é vazia, as listas são diferentes

se ambas não forem vazias, teste

se informações associadas aos primeiros nós são iguais e **se as sub-listas são iguais**

Listas iguais

```
int lst_igual (Lista* lst1, Lista* lst2)
{
    Lista* p1;           /* ponteiro para percorrer l1 */
    Lista* p2;           /* ponteiro para percorrer l2 */
    for (p1=lst1, p2=lst2;
         p1 != NULL && p2 != NULL;
         p1 = p1->prox, p2 = p2->prox)
    {
        if (p1->info != p2->info) return 0;
    }
    return p1==p2;
}
```

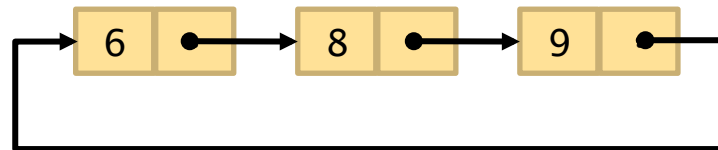
```
int lst_igual_rec(Lista* lst1, Lista* lst2)
{
    if (lst1 == NULL && lst2 == NULL)
        return 1;
    else if (lst1 == NULL || lst2 == NULL)
        return 0;
    else
        return (lst1->info == lst2->info) &&
               lst_igual_rec (lst1->prox, lst2->prox);
}
```

Lista circular

Lista circular

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

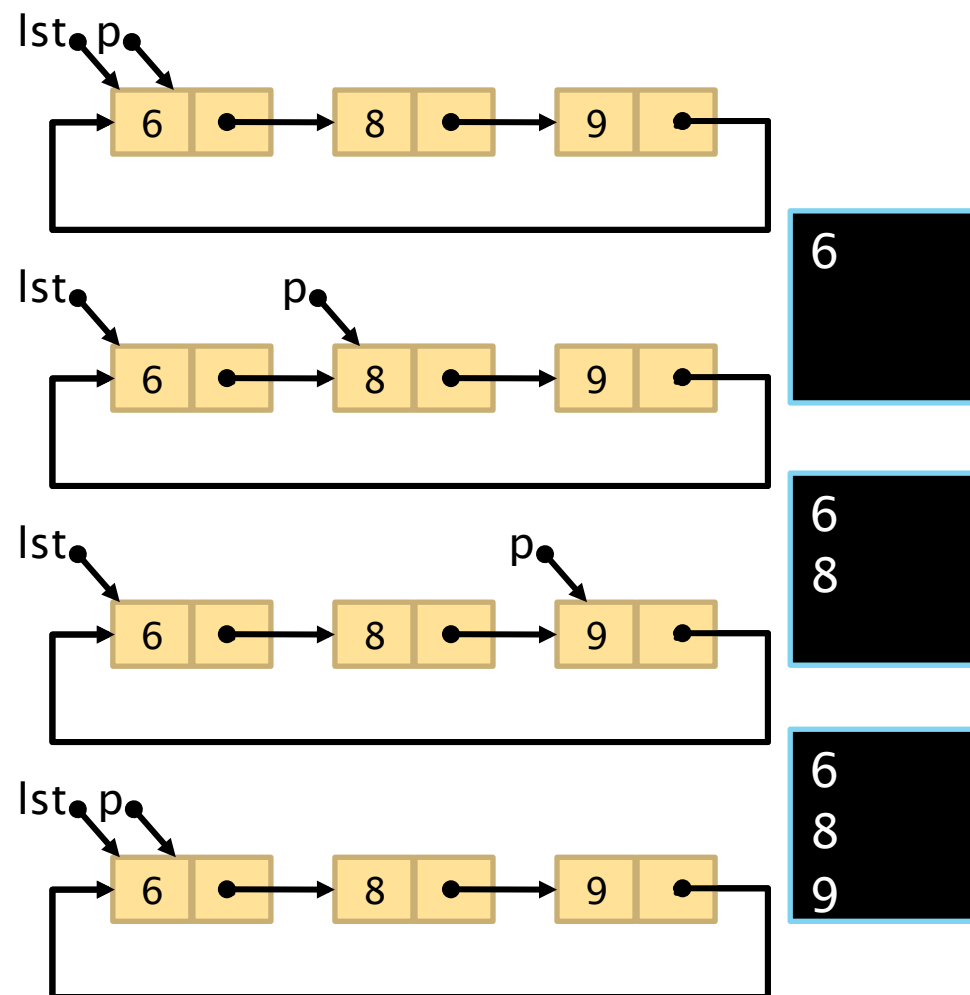
a lista pode ser representada por um ponteiro para um elemento inicial qualquer da lista



Lista circular - Imprime

visita todos os elementos
a partir do ponteiro do
elemento inicial até
alcançar novamente esse
mesmo elemento

se a lista é vazia, o
ponteiro para um
elemento inicial é NULL



Lista circular - Imprime

visita todos os elementos a partir do ponteiro do elemento inicial até alcançar novamente esse mesmo elemento

se a lista é vazia, o ponteiro para um elemento inicial é NULL

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

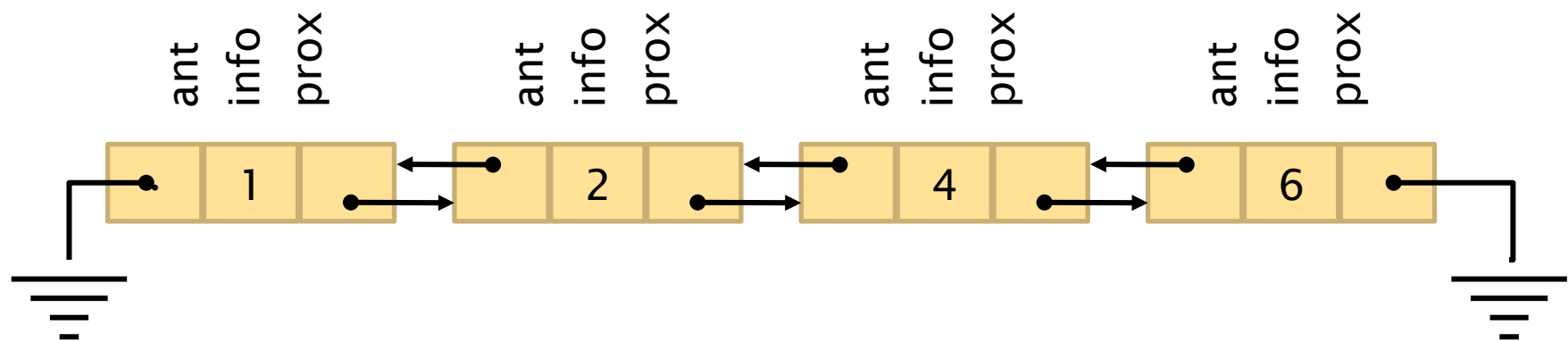

Lista duplamente encadeada

Lista duplamente encadeada

cada elemento tem um ponteiro para o próximo elemento e um ponteiro para o elemento anterior

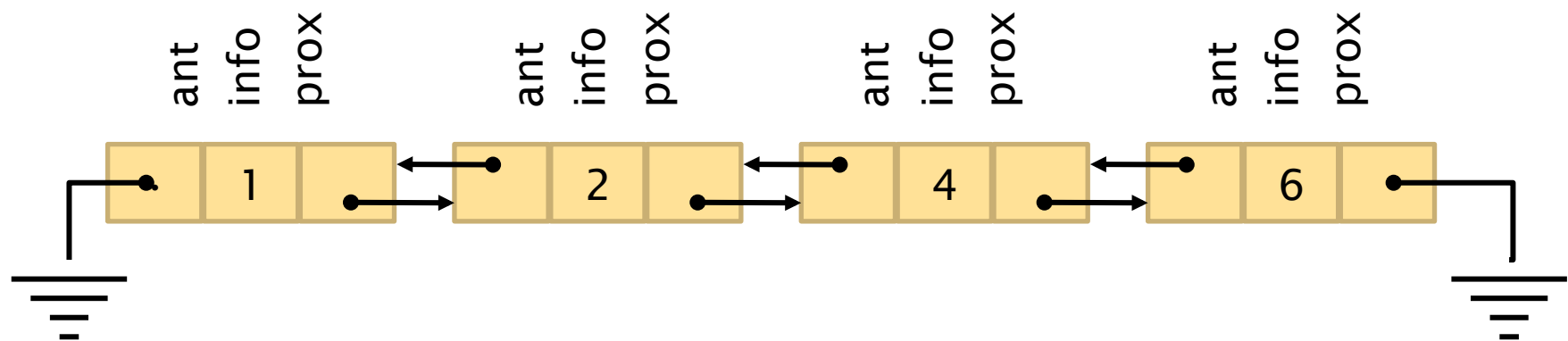
dado um elemento, é possível acessar o próximo e o anterior

dado um ponteiro para o último elemento da lista, é possível percorrer a lista em ordem inversa



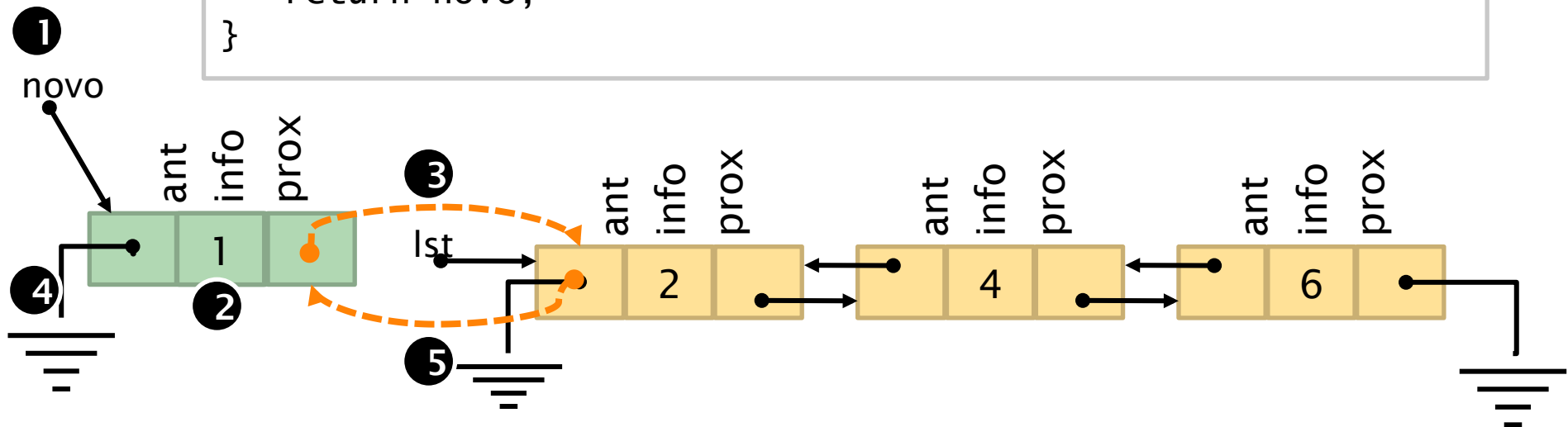
Lista duplamente encadeada – Exemplo

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



Lista duplamente encadeada – Inserção no início da lista

```
/* inserção no início: retorna a lista atualizada */  
Lista2* lst2_inserere (Lista2* lst, int val)  
{  
    Lista2* novo = (Lista2*) malloc(sizeof(Lista2));  
    novo->info = val;  
    novo->prox = lst;  
    novo->ant = NULL;  
    /* verifica se lista não estava vazia */  
    if (lst != NULL)  
        lst->ant = novo;  
    return novo;  
}
```



Lista duplamente encadeada – Busca elemento

recebe a informação referente ao elemento a pesquisar

retorna o ponteiro do nó da lista que representa o elemento, ou NULL, caso o elemento não seja encontrado na lista

implementação idêntica à lista encadeada (simples)

```
/* função busca: busca um elemento na lista */  
Lista2* lst2_busca (Lista2* lst, int val)  
{  
    Lista2* p;  
    for (p=lst; p!=NULL; p=p->prox)  
        if (p->info == val)  
            return p;  
    return NULL;          /* não achou o elemento */  
}
```

Lista duplamente encadeada – Retira elemento

p aponta para o elemento a retirar

se **p** aponta para um elemento no meio da lista:

o anterior passa a apontar para o próximo: **p->ant->prox = p->prox;**

o próximo passa a apontar para o anterior: **p->prox->ant = p->ant;**

se **p** aponta para o último elemento

não é possível escrever **p->prox->ant**, pois **p->prox** é **NULL**

se **p** aponta para o primeiro elemento

não é possível escrever **p->ant->prox**, pois **p->ant** é **NULL**

é necessário atualizar o valor da lista, pois o primeiro elemento pode ser removido

```

/* função retira: remove elemento da lista */
Lista2* lst2_retira (Lista2* lst, int val)
{
    Lista2* p = busca(lst, val);          /* busca elemento */

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

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

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

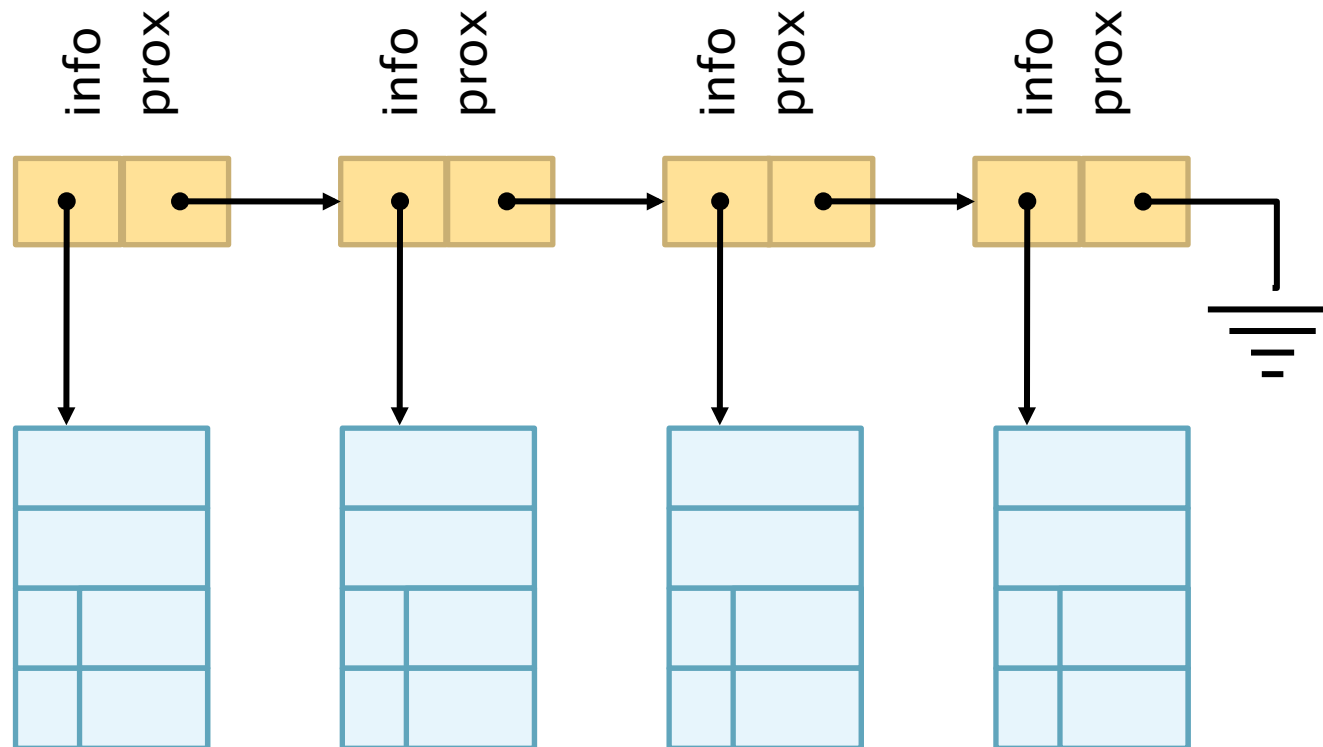
    free(p);

    return lst;
}

```

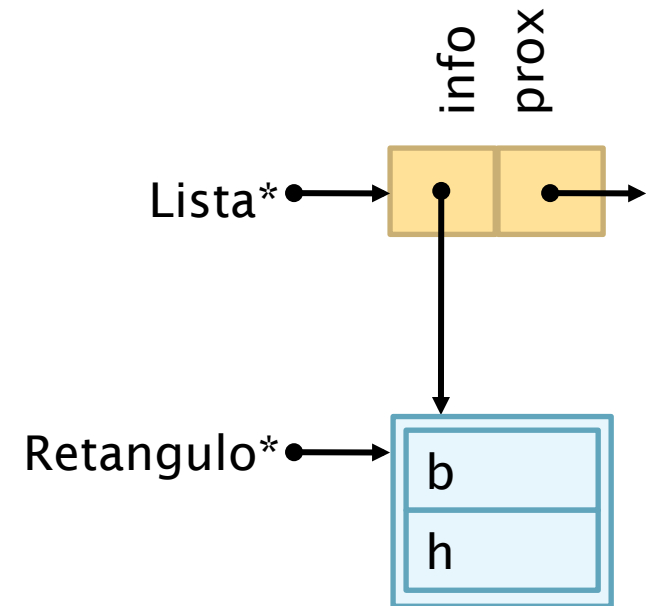
Lista de ponteiros para estruturas

Lista de ponteiros para tipos estruturados



Exemplo – Lista de retângulos

```
struct retangulo {  
    float b;  
    float h;  
};  
typedef struct retangulo Retangulo;  
  
struct lista {  
    Retangulo* info;  
    struct lista *prox;  
};  
typedef struct lista Lista;
```

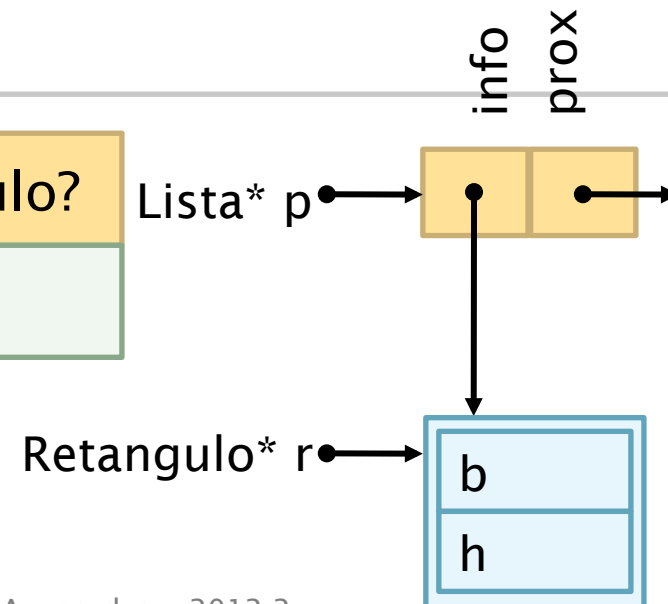


Exemplo – Lista de retângulos - Alocação

```
static Lista* aloca (float b, float h)
{
    Retangulo* 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;
}
```

Dado p, como acessar a altura do retângulo?

p->info->h



Lista heterogênea

Lista heterogênea

Cada nó pode possuir informações de um tipo diferente.

Exemplo: lista de formas geométricas

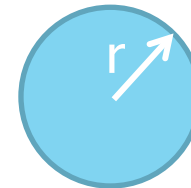
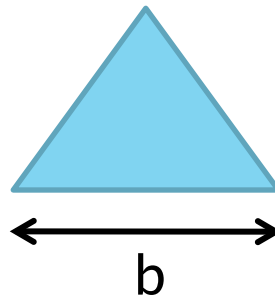
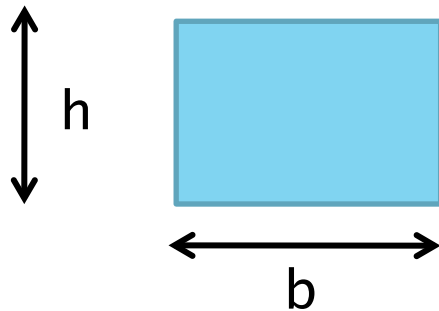
um elemento pode ser retângulo, triângulo ou círculo

as áreas desses objetos são:

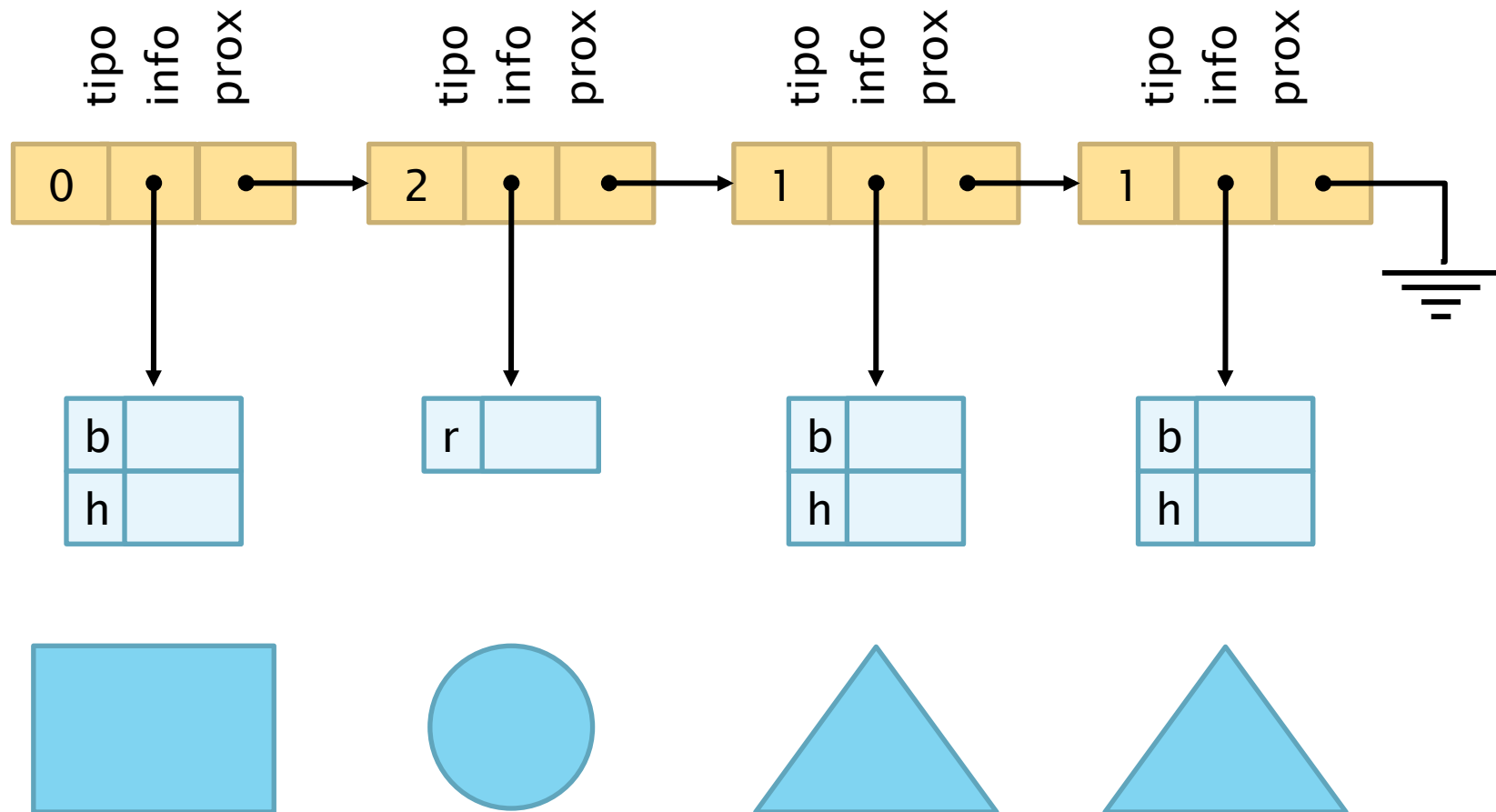
$$a_r = b * h$$

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

$$a_c = \pi r^2$$



Lista homogênea de objetos heterogêneos



```

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

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;

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

```

Exemplo: Lista de formas geométricas - Função para a criação de um nó de retângulo

```
/* 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:

- `cria_ret(float b, float h);`
- `cria_tri(float b, float h);`
- `cria_circ(float r);`

Exemplo – Função para calcular a maior área (funções auxiliares)

```
/* 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;
}
```

Exemplo – Função para calcular a maior área (funções auxiliares)

```
/* 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;
}
```

conversão de tipo

conversão de tipo

conversão de tipo

Dúvidas?

Motivação

Listas encadeadas

Implementações recursivas

Listas circulares

Listas duplamente encadeadas

Listas de (ponteiros para) tipos estruturados