Estruturas de Dados

Módulo 10 – Listas Encadeadas

Referências

Waldemar Celes, Renato Cerqueira, José Lucas Rangel, *Introdução a Estruturas de Dados*, Editora Campus (2004)

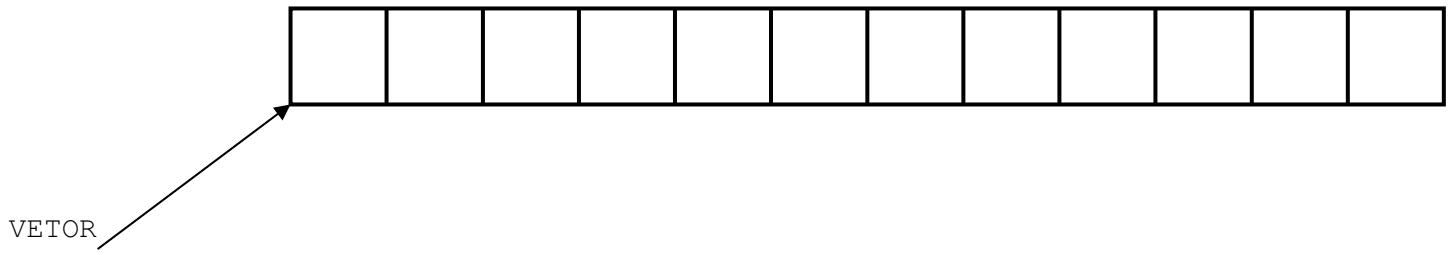
Capítulo 10 – Listas encadeadas

Tópicos

- Motivação
- Listas encadeadas
- Implementações recursivas
- Listas circulares
- Listas duplamente encadeadas
- Listas de tipos estruturados

Motivação

- Vetor
  - ocupa um espaço contíguo de memória
  - permite acesso randômico aos elementos
  - deve ser dimensionado com um número máximo de elementos

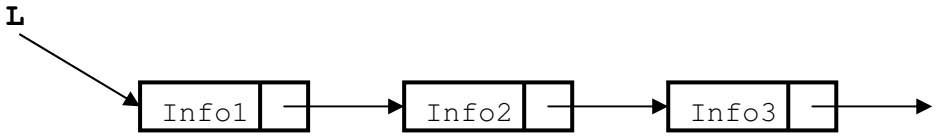


Motivação

- Estruturas de dados dinâmicas:
  - crescem (ou decrescem) à medida que elementos são inseridos (ou removidos)
  - Exemplo:
    - listas encadeadas:
      - amplamente usadas para implementar outras estruturas de dados

Listas Encadeadas

- 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



Listas Encadeadas

- Exemplo:
  - lista encadeada armazenando valores inteiros
  - estrutura lista
    - estrutura dos nós da lista
  - tipo Lista
    - tipo dos nós da lista

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

lista é uma estrutura auto-referenciada,
pois o campo *prox* é um ponteiro
para uma próxima estrutura do mesmo tipo

uma lista encadeada é representada pelo
ponteiro para seu primeiro elemento, do tipo Lista*

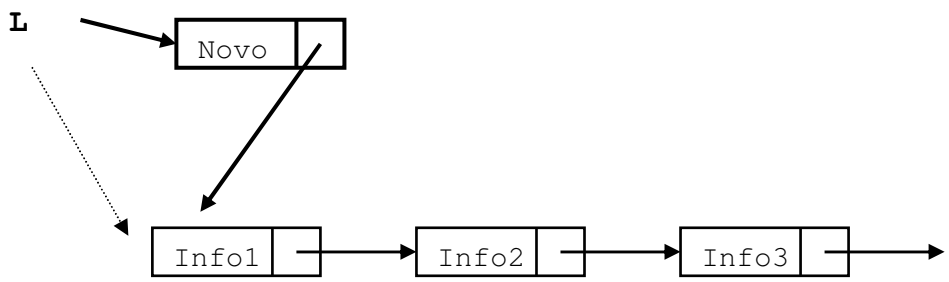
Listas Encadeadas

- Exemplo - Função de criação
  - cria uma lista vazia, representada pelo ponteiro NULL

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

Listas Encadeadas

- Exemplo - Função de inserção
  - aloca memória para armazenar o elemento
  - encadeia o elemento na lista existente




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

Listas Encadeadas

- Exemplo - Trecho de código
  - cria uma lista inicialmente vazia e insere novos elementos

```
int main (void)
{
   Lista* l;                  /* declara uma lista não inicializada */
   l = lst_cria();            /* cria e inicializa lista como vazia */
   l = lst_insere(l, 23);     /* insere na lista o elemento 23      */
   l = lst_insere(l, 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

- Exemplo - Função para imprimir uma lista
  - imprime os valores dos elementos armazenados

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

variável auxiliar p:

- ponteiro, usado para armazenar o endereço de cada elemento
- dentro do loop, aponta para cada um dos elementos da lista

Listas Encadeadas

- Exemplo - Função para verificar se uma lista está vazia
  - retorna 1 se a lista estiver vazia ou 0 se não estiver vazia

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

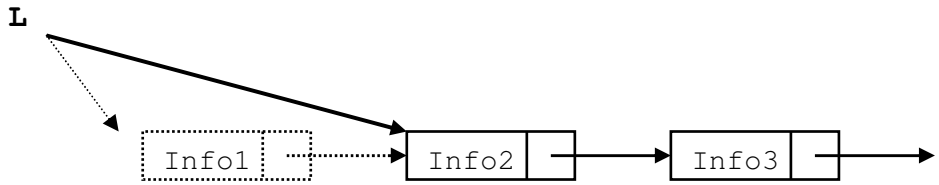
Listas Encadeadas

- Exemplo - Função de 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

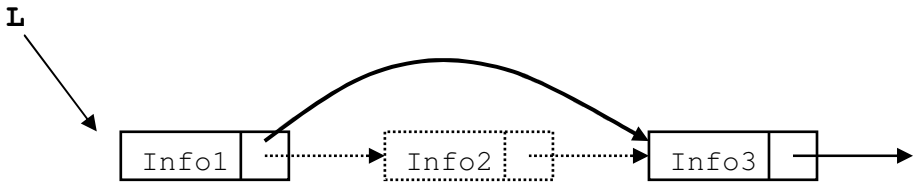
```
/* função busca: busca um elemento na lista */
Lista* busca (Lista* l, int v)
{ Lista* p;
   for (p=l; p!=NULL; p = p->prox) {
      if (p->info == v)
         return p;
   }
   return NULL;       /* não achou o elemento */
}
```

Listas Encadeadas

- Exemplo - Função para retirar um elemento da lista
  - recebe como entrada a lista e o valor do elemento a retirar
  - atualiza o valor da lista, se o elemento removido for o primeiro



  - caso contrário, apenas remove o elemento da lista



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

Listas Encadeadas

- Exemplo - Função para liberar a lista
  - destrói a lista, liberando todos os elementos alocados

```
void lst_libera (Lista* l)
{
   Lista* p = l;
   while (p != NULL) {
      Lista* 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

- TAD Lista de inteiros

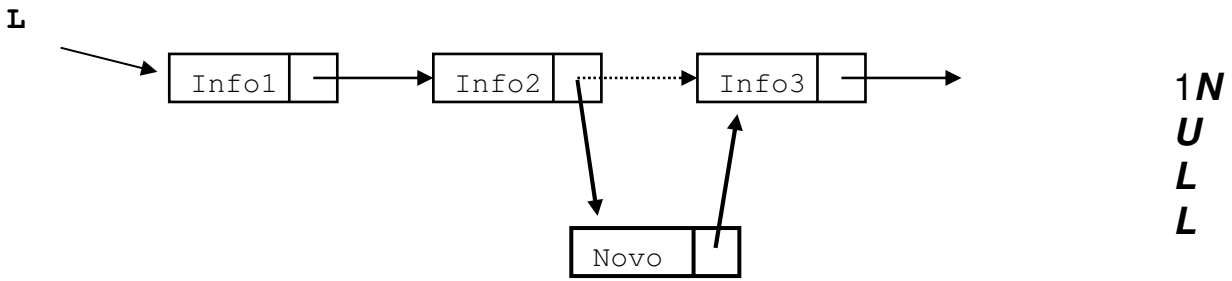
```
/* TAD: lista de inteiros */

typedef struct lista Lista;

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

Lista* lst_insere (Lista* l, int i);
Lista* lst_retira (Lista* l, int v);

int lst_vazia (Lista* l);
Lista* lst_busca (Lista* l, int v);
void lst_imprime (Lista* l);
```

programa que utiliza as funções de lista exportadas

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

Listas Encadeadas

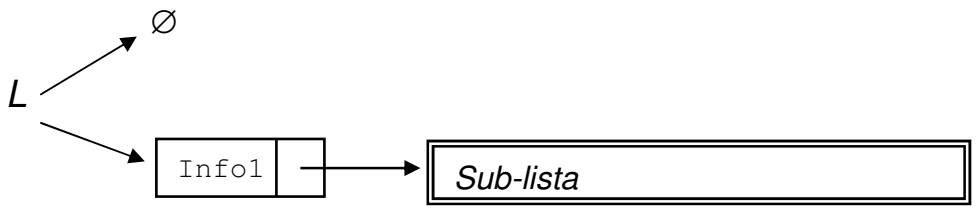
- Manutenção da lista ordenada
  - função de inserção percorre os elementos da lista até encontrar a posição correta para a inserção do novo



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

Implementações Recursivas

- Definição recursiva de lista:
  - uma lista é
    - uma lista vazia; ou
    - um elemento seguido de uma (sub-)lista



Implementações Recursivas

- Exemplo - Função para imprimir uma lista
  - se a lista for vazia, não imprima nada
  - caso contrário,
    - imprima a informação associada ao primeiro nó, dada por l->info
    - imprima a sub-lista, dada por l->prox,
      chamando recursivamente a função

Implementações Recursivas

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

Implementações Recursivas

- Exemplo - função para retirar um elemento da lista
  - retire o elemento, se ele for o primeiro da lista (ou da sub-lista)
  - caso contrário, chame a função recursivamente para retirar o elemento da sub-lista

Implementações Recursivas

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

é necessário re-atribuir o valor de l->prox na chamada recursiva, já que a função pode alterar o valor da sub-lista

Implementações Recursivas

- Exemplo - função para testar igualdade de duas listas

  int lst_igual (Lista* l1, Lista* l2);

  – implementação não recursiva
    - 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

Implementações Recursivas

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

Implementações Recursivas

- Exemplo - função para testar igualdade de duas listas

  int lst_igual (Lista* l1, Lista* l2);

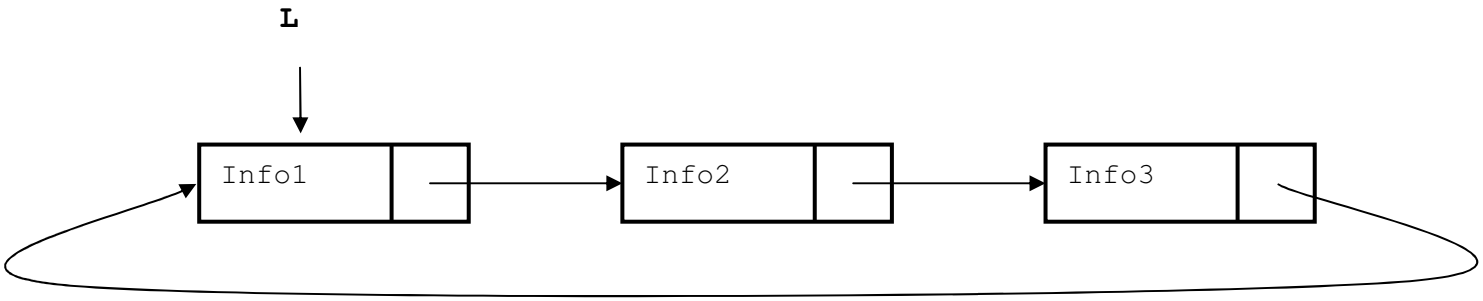
  - implementação recursiva
    - se as duas listas dadas são vazias, são iguais
    - se não forem ambas vazias, mas uma delas é vazia, são diferentes
    - se ambas não forem vazias, teste
      - se informações associadas aos primeiros nós são iguais e
      - se as sub-listas são iguais

Implementações Recursivas

```
int lst_igual (Lista* l1, Lista* l2)
{
   if (l1 == NULL && l2 == NULL)
      return 1;
   else if (l1 == NULL || l2 == NULL)
      return 0;
   else
      return l1->info == l2->info && lst_igual(l1->prox, l2->prox);
}
```

Listas Circulares

- 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



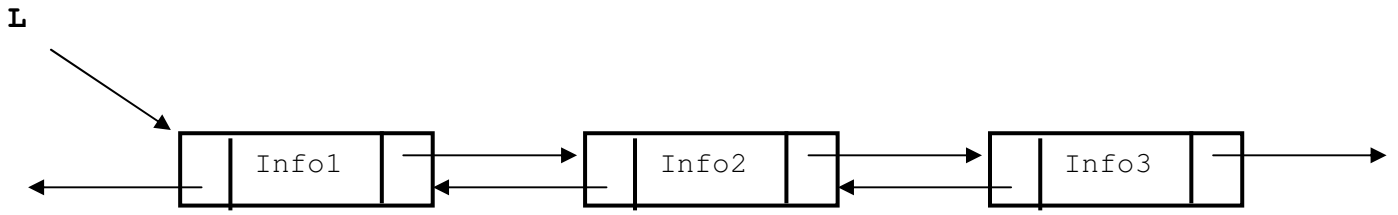
Listas Circulares

- Exemplo - Função para imprimir uma lista circular
  - 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* l)
{
  Lista* p = l;       /* 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 != l);
}
```

Listas Duplamente Encadeadas

- 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



Listas Duplamente Encadeadas

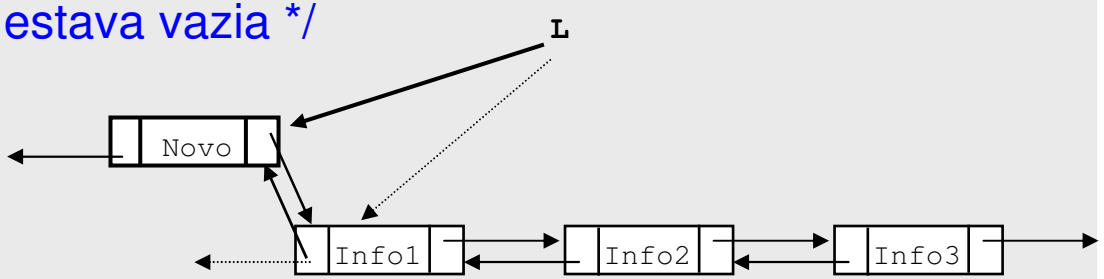
- Exemplo:
  - lista encadeada armazenando valores inteiros
  - estrutura lista2
    - estrutura dos nós da lista
  - tipo Lista2
    - tipo dos nós da lista

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

Listas Duplamente Encadeadas

- Exemplo - Função de inserção (no início da lista)

```
/* inserção no início: retorna a lista atualizada */
Lista2* lst2_insere (Lista2* l, int v)
{
  Lista2* novo = (Lista2*) malloc(sizeof(Lista2));
  novo->info = v;
  novo->prox = l;
  novo->ant = NULL;
  /* verifica se lista não estava vazia */
  if (l != NULL)
    l->ant = novo;
  return novo;
}
```



Listas Duplamente Encadeadas

- Exemplo - Função de 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
  - implementação idêntica à lista encadeada (simples)

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

Listas Duplamente Encadeadas

- Exemplo - Função para retirar um elemento da lista
  - 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->a;
  - 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 será removido

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

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

  /* retira elemento do encadeamento */
  if (l == p)         /* testa se é o primeiro elemento */
    l = 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 l;
}
```

Listas de Tipos Estruturados

- Lista de tipo estruturado:
  - 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

Listas de Tipos Estruturados

- Lista de tipo estruturado (cont.):
  - 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 de retângulos

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

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

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

Listas de Tipos Estruturados

- Exemplo – Função auxiliar para alocar um nó

```
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;
}
```

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

O valor da base associado a um nó p seria acessado por: p->info->b.

Listas de Tipos Estruturados

- 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

Listas de Tipos Estruturados

- Exemplo:
  - listas de retângulos, triângulos ou círculos
  - áreas desses objetos são dadas por:

$$r = b * h \qquad t = \frac{b * h}{2} \qquad c = \pi\ r^2$$

```
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;
```

Listas de Tipos Estruturados

- Exemplo:
  - 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
      - deve ser do tipo genérico (ou seja, do tipo void*)
        pois pode apontar para um retângulo, um triângulo ou um círculo
    - um identificador indicando qual objeto o nó armazena
      - consultando esse identificador, o ponteiro genérico
        pode ser convertido no ponteiro específico para o objeto e
        os campos do objeto podem ser acessados

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

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

Listas Duplamente Encadeadas

- Exemplo – Função para a criação de um nó da lista

```
/* 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

Listas Duplamente Encadeadas

- Exemplo – Função para calcular a maior área
  - retorna a maior área entre os elementos da lista
  - para cada nó, de acordo com o tipo de objeto que armazena, chama uma função específica para o cálculo da área

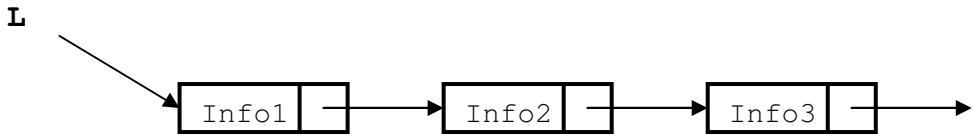
```
/* função para cálculo da área de um retângulo */
static float ret_area (Retangulo* r)
{
  return r->b * r->h;
}

/* 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;
}
```

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

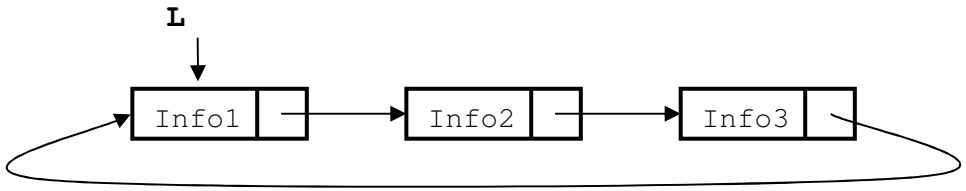
a conversão de ponteiro genérico para ponteiro específico ocorre quando uma das funções de cálculo da área é chamada:
passa-se um ponteiro genérico, que é atribuído a um ponteiro específico, através da conversão implícita de tipo

Resumo

Listas encadeadas



Listas circulares



Listas duplamente encadeadas