



Estruturas de Dados
Aula 12: Outras
Implementações de Listas

Fontes Bibliográficas

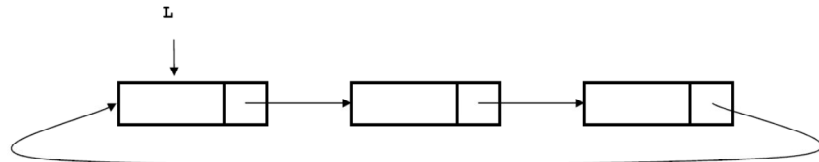


- Waldemar Celes, Renato Cerqueira, José Lucas Rangel, *Introdução a Estruturas de Dados*, Editora Campus (2004)
 - Capítulo 10 - Listas encadeadas

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



Função Imprime

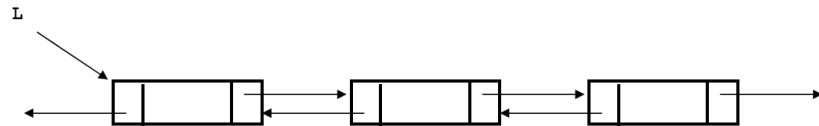


```
/* imprime valores dos elementos */
void lcirc_imprime (TipoLista* l)
{
    /* faz p apontar para a célula inicial */
    TipoLista* p = l;
    /* testa se lista não é vazia e então percorre com
    do-while */
    if (p) do {
        /* imprime informação da célula */
        printf("%d\n", p->Item.valor);
        /* avança para a próxima célula */
        p = p->prox;
    } while (p != l);
}
```

Listas Duplamente Encadeadas



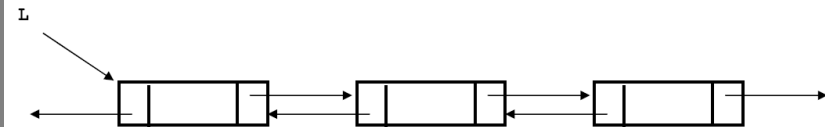
- Cada elemento tem um ponteiro para o próximo elemento e um ponteiro para o elemento anterior
- Dado um elemento, é possível acessar o próximo e o anterior
- Dado um ponteiro para o último elemento da lista, é possível percorrer a lista em ordem inversa



Listas Duplamente Encadeadas



```
typedef struct tipoitem TipoItem;  
typedef struct celula_str TipoListaDpl;
```

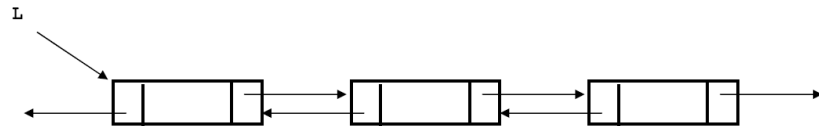


Listas Duplamente Encadeadas



```
struct tipoitem{
    int valor;
    /* outros componentes */
};

struct celula_str{
    TipoItem Item;
    struct celula_str* Prox, Ant;
};
```

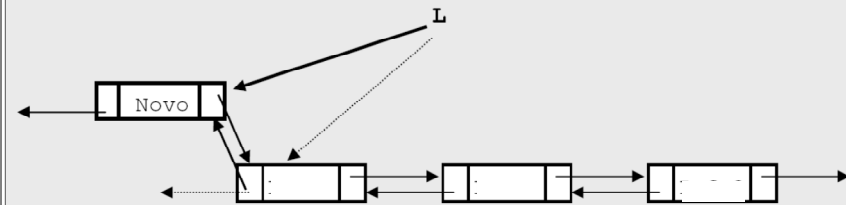


Função Inserir (duplamente encadeada)



```
/* inserção no início: retorna a lista atualizada */
TipoListaDpl* lstdpl_insere (TipoListaDpl* l, int v)
{
    TipoListaDpl* novo = (TipoListaDpl*)
                        malloc(sizeof(TipoListaDpl));

    novo->Item.valor = v;
    novo->Prox = l;
    novo->Ant = NULL;
    /* verifica se lista não estava vazia */
    if (l != NULL)
        l->Ant = novo;
    return novo;
}
```



Função de Busca



- Recebe a informação referente ao elemento a pesquisar
- Retorna o ponteiro da célula da lista que representa o elemento, ou NULL, caso o elemento não seja encontrado na lista
- implementação idêntica à lista encadeada (simples)

```
TipoListaDpl* busca (TipoListaDpl* l, valor v)
{TipoListaDpl* p;
  for (p=l; p!=NULL; p = p->Prox) {
    if (p->Item.valor == v)
      return p;
  }
  return NULL; /*não encontrou o elemento */
}
```

Função de Retirar (Exercício)



- Assinatura da função retira:

```
TipoListaDpl* lstdpl_retira (TipoListaDpl* l, int
v)
```

Função de Retirar



- Se p é um ponteiro para o elemento a ser retirado, devemos fazer:
 - 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 estiver em algum extremo da lista, devemos considerar as **condições de contorno**;
- 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 de Retirar



```
/* função retira: remove elemento da lista */
TipoListaDpl* lstdpl_retira (TipoListaDpl* l, int v)
{
    TipoListaDpl* p = busca(l,v);
    if (p == NULL)
        /* não achou o elemento: retorna lista inalterada */
        return l;
    /* 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



- A informação associada a cada célula (TipoItem) 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 TipoItem pode ser representado por um ponteiro para uma estrutura, em lugar da estrutura em si
- Independente da informação armazenada na lista, a estrutura da célula é sempre composta por:
 - um ponteiro para a informação e
 - um ponteiro para a próxima célula da lista

Exemplo: Lista de Retângulos



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

typedef struct retangulo Retangulo;
typedef struct celula_str{
    Retangulo* Item;
    struct celula_str* Prox;
} Celula;
typedef Celula TipoLista;
```

Função para alocar uma célula



```
static TipoLista* aloca (float b, float h)
{
    Retangulo* r = (Retangulo*) malloc(sizeof(Retangulo));
    TipoLista* p = (TipoLista*) malloc(sizeof(TipoLista));
    r->b = b;
    r->h = h;
    p->Item = 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->Item->b.

Listas Heterogêneas



- Como o campo Item da Célula é um ponteiro, podemos construir listas heterogêneas, ou seja, com células apontando para tipos diferentes;
- Por exemplo, imagine uma lista de retângulos, triângulos e círculos, cujas áreas são dadas por, respectivamente:

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

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;
```

Listas Heterogêneas



- A célula contém:
 - um ponteiro para a próxima célula 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 a célula armazena
 - consultando esse identificador, o ponteiro genérico pode ser convertido no ponteiro específico para o objeto e assim, os campos do objeto podem ser acessados

Listas Heterogêneas



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

typedef struct celula_str{
    int tipo;
    void* Item;
    struct celula_str* Prox;
} Celula;

typedef Celula TipoListaHet;
```

Listas Heterogêneas - Exercícios



- Defina as operações para alocar células:

```
TipoListaHet* cria_ret (float b, float h)
TipoListaHet* cria_tri (float b, float h)
TipoListaHet* cria_cir (float r)
```

Listas Heterogêneas - Exercícios



```
TipoListaHet* cria_ret (float b, float h)
{
    Retangulo* r = (Retangulo*) malloc(sizeof(Retangulo));
    r->b = b;
    r->h = h;
    TipoLista* p = (TipoLista*) malloc(sizeof(TipoLista));
    p->tipo = RET;
    p->Item = r;
    p->Prox = NULL;
    return p;
}
```

Listas Heterogêneas - Exercícios



- Fazer função que retorna a maior área entre os elementos da lista
 - 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

Listas Heterogêneas - Exercícios



```
/* 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;
}
```

Listas Heterogêneas - Exercícios



```
static float area (TipoListaHet* p){
    float a;
    switch (p->tipo) {
        case RET:
            a = ret_area (p->Item);
            break;
        case TRI:
            a = tri_area (p->Item);
            break;
        case CIR:
            a = cir_area (p->Item);
            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

Listas Heterogêneas - Exercícios



```
float max_area (TipoListaHet* l)
{
    float amax = 0.0;
    TipoListaHet* p;
    for (p=l; p!=NULL; p=p->Prox){
        float a = area(p);
        if (a>amax)
            amax = a;
    }
    return amax;
}
```