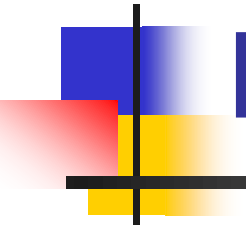
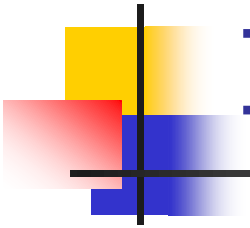


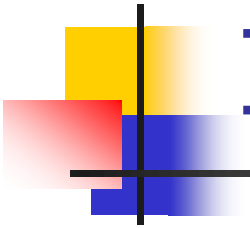
Listas Encadeadas

Prof. Rui Jorge Tramontin Jr.



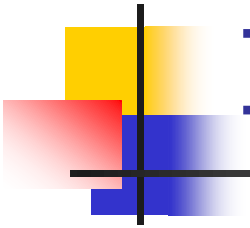
Índice

- Introdução
- Listas Encadeadas
  - Definição
  - Modelagem
  - Operações
  - Exemplos de uso do TAD
- Exercícios



Introdução

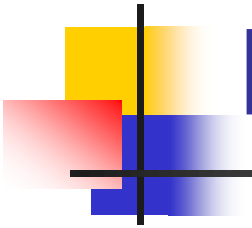
- Uma lista é uma estrutura na qual os seus dados são organizados e acessados numa seqüência determinada.
- Exemplos de particulares de listas são pilhas e filas, cuja manipulação é sempre feita nas extremidades.
- Uma lista permite outros tipos de operações, por exemplo a manipulação de dados no meio dela.



Introdução

- Implementação de listas:
  - Vetores
  - Alocação dinâmica

- Embora a implementação usando vetores seja mais simples e intuitiva, algumas desvantagens precisam ser consideradas:
  - Tamanho máximo fixo (é preciso testar se lista está cheia).
  - Mesmo vazias, ocupam todo o espaço alocado.
  - Operações no meio da lista envolvem muitos deslocamentos de dados (baixo desempenho).



Listas Encadeadas

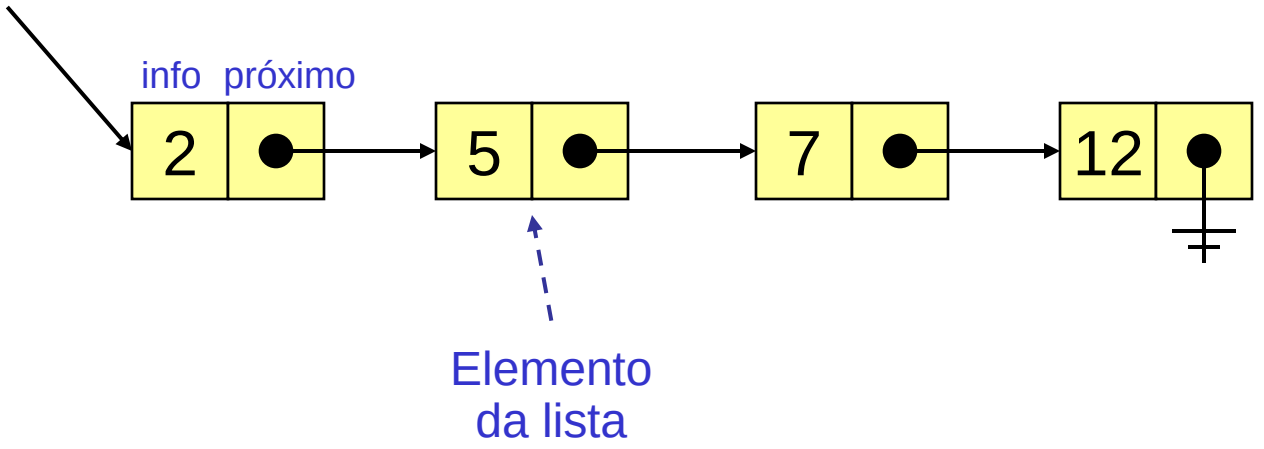
- São listas onde cada elemento está armazenado em uma estrutura (definida por um TAD elemento de lista).
- Cada elemento só é alocado quando necessário.
- Cada elemento contém:
  - Campo contendo a informação armazenada;
  - Ponteiro para o próximo elemento;
- Último elemento aponta para nulo (*NULL*).

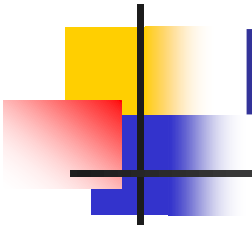
Listas Encadeadas

```
// define estrutura elemento de lista
typedef struct elemento {
     int info;
     struct elemento *proximo;
} Elemento;
```

Esta estrutura não é genérica, pois info é do tipo inteiro.

Ponteiro para o 1º. elemento, permitindo o acesso aos demais elementos da lista

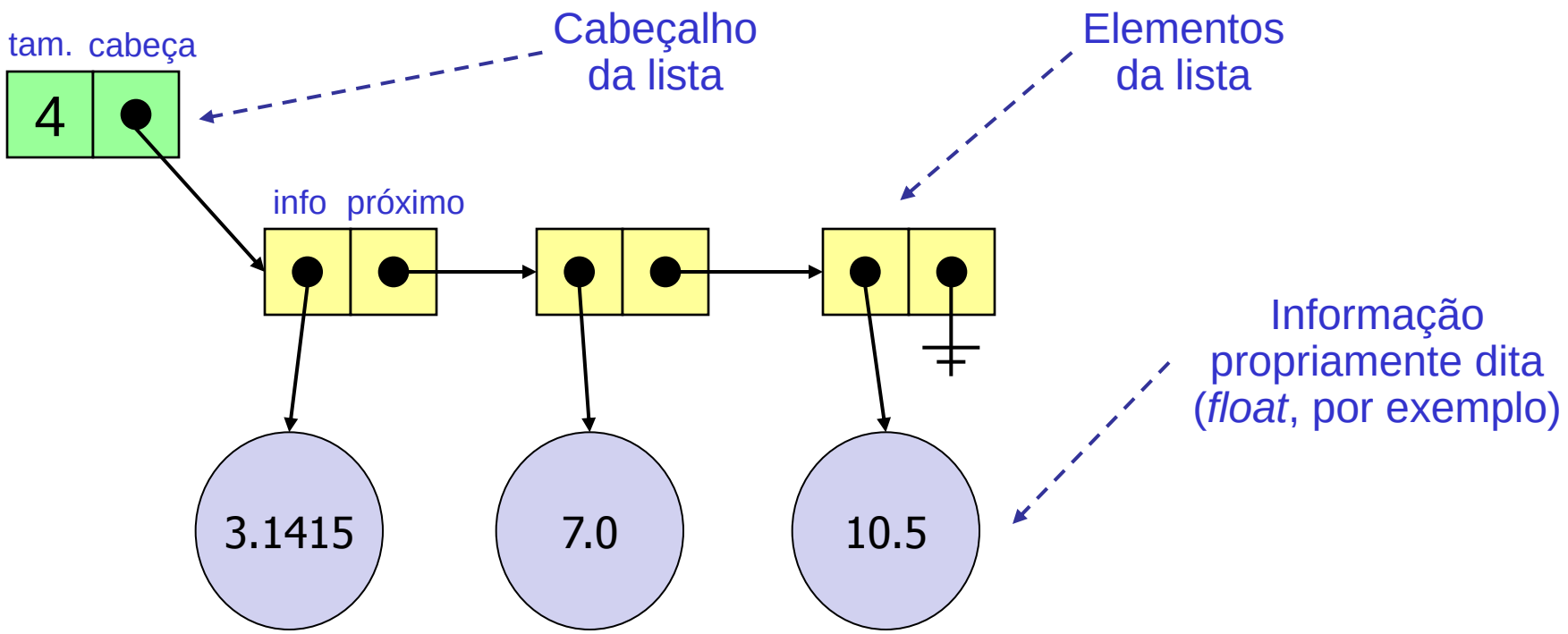




Modelagem da Lista

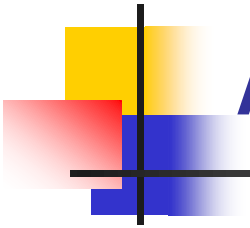
- Para tornar mais genérico o código das operações da lista, é modificar a estrutura de um elemento da lista.
- O campo info passa a ser um ponteiro do tipo void.
- É preciso definir um cabeçalho (ou descritor) para a lista, que contém:
  - Tamanho da informação a ser armazenada: indica quantos *bytes* serão copiados durante as operações de manipulação da lista;
  - Ponteiro para o primeiro elemento, o cabeça da lista.

Lista Encadeada Genérica



Modelagem da Lista Genérica

```
// definição de constantes
#define ERRO_LISTA_VAZIA -1


// define estrutura elemento de lista
typedef struct elemento {
     void *info;
     struct elemento *proximo;
} Elemento;

// define estrutura cabeçalho de lista
typedef struct {
     int tamanhoInfo;
     Elemento *cabeca;
} Lista;
```



Aspecto Funcional

- Inicialização: criar lista, limpar lista;
- Testes: lista vazia, posição do elemento, contém elemento;
- Manipulação (no início, no fim, na posição, em ordem)
  - Insere;
  - Remove;
  - Modifica.
- Mostrar os dados da lista na tela;
- Ordenação da lista;

Exemplo de uso do TAD Lista

```
#include "Lista.h"

int main(int argc, char *argv[]){
   Lista *lista;

   inicializaLista(&lista, sizeof(float)); // Tamanho da informação é passada
                                           // na inicialização...
   float x = 7.5;

   insereNoInicio(&lista, &x); // Informação é passada por referência, pois
                               // a lista é genérica (usa um ponteiro void).
   x = 10;

   adicionaNoFim(&lista, &x);

   removeDoInicio(&lista, &x);

   printf("valor removido: %f\n", x);
}
```

Inicialização da Lista

- Definir tamanhoInfo com base no parâmetro (normalmente valor obtido com a função *sizeof*);
- Definir cabeça = NULL;

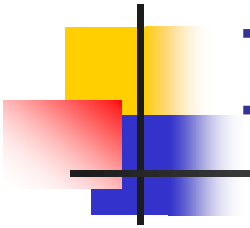
```
void inicializaLista(Lista *l, int tamanho){
    l->tamanhoInfo = tamanho;
    l->cabeca = NULL;
}
```

Testa se lista está vazia

- Basta testar se cabeça é nulo (NULL);
- **Obs**: não é preciso testar se lista está cheia, pois cada operação de inserção aloca espaço na memória dinamicamente.

```
int listaVazia(Lista l){
   return l.cabeca == NULL;
}
```

Inserir no início



- Alocar espaço para elemento;
- Alocar espaço para informação;
- Copiar dados do ponteiro passado como parâmetro para a área alocada (usando *memcpy*);
- Campo "proximo" do novo elemento aponta para onde o "cabeça" da lista aponta;
- "cabeça" da lista aponta para novo elemento;

Inserir no início

```
int insereNoInicio(Lista *l, void *info){
   Elemento *novo = (Elemento *) malloc(sizeof(Elemento));
   if(novo == NULL)
      return 0; // Erro, falta de memória!

   novo->info = malloc(l->tamanhoInfo);
   if(novo->info == NULL){
      free(novo);
      return 0; // Erro, falta de memória!
   }
   memcpy(novo->info, info, l->tamanhoInfo);
   novo->proximo = l->cabeca;
   l->cabeca = novo;
   return 1; // Sucesso.
}
```

Remover do início

- Testar se está vazia → erro!;
- Obter referência do primeiro elemento (cabeça);
- Copiar dados do elemento para a área apontada pelo ponteiro passado como parâmetro (usando *memcpy*);
- Desalocar área da informação;
- Cabeça da lista aponta para o "próximo" do primeiro elemento;
- Desalocar elemento;

Remover do início

```
int removeDoInicio(Lista *l, void *info){
   if( listaVazia(*l) )
      return ERRO_LISTA_VAZIA;

   Elemento *p = l->cabeca;
   memcpy(info, p->info, l->tamanhoInfo);
   free(p->info);
   l->cabeca = p->proximo;
   free(p);
   return 1; // Sucesso.
}
```

Inserir no fim

- Se lista vazia, inserir no início.

Caso geral:

- Obter a referência do primeiro elemento;
- Percorrer a lista, enquanto o próximo do elemento atual não for nulo;
- Quando chegar no último elemento, repetir o procedimento feito em *insereNoInicio*, mas:
  - "proximo" do novo elemento é nulo;
  - "proximo" do elemento atual aponta para o novo.

Inserir no fim

```
int insereNoFim(Lista *l, void *info){
   if( listaVazia(*l) )
      return insereNoInicio(l, info);

   Elemento *novo = (Elemento *) malloc(sizeof(Elemento));
   if(novo == NULL)
      return 0; // Erro, falta de memória!

   novo->info = malloc(l->tamanhoInfo);
   if(novo->info == NULL){
      free(novo);
      return 0; // Erro, falta de memória!
   }
   Elemento *p = l->cabeca;
   while(p->proximo != NULL){
      p = p->proximo;
   }

   memcpy(novo->info, info, l->tamanhoInfo);
   novo->proximo = NULL;
   p->proximo = novo;
   return 1; // Sucesso.
}
```

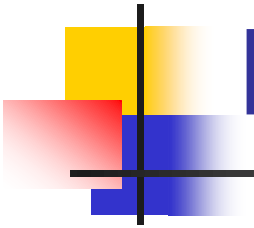
Remover do fim

- Testa se está vazia → erro!;
- Testa se há um elemento → remove do início;

- Caso geral: percorrer até o penúltimo elemento.
  - Obter a referência do primeiro elemento;
  - Percorrer a lista, enquanto o próximo do próximo do elemento atual não for nulo (até chegar no penúltimo);
  - Obter referência do último elemento;
  - Copiar informação;
  - Desalocar informação;
  - Obter referência da informação;
  - "próximo" do penúltimo é definido como nulo;
  - Desalocar último.

Remover do fim

```
int removeDoFim(Lista *l, void *info){
   if( listaVazia(*l) )
      return ERRO_LISTA_VAZIA;

   if(l->cabeca->proximo == NULL) // Lista contém apenas um elemento!
         return removeDoInicio(l, info);

   Elemento *p = lista->cabeca;
   while(p->proximo->proximo != NULL){
       p = p->proximo;
   }
   Elemento *ultimo = p->proximo;
   memcpy(info, ultimo->info, l->tamanhoInfo);
   free(ultimo->info);
   free(ultimo);
   p->proximo = NULL;
}
```



Mostrar dados da lista

- Esta função percorre a lista e mostra na tela os dados contidos nos seus elementos;
- Como a lista é genérica (void *), não há como prever qual a semântica da informação armazenada (*float*, *int*, tipos abstratos, etc.);
- Para resolver este problema, pode-se utilizar funções passadas como parâmetro (*callback*);
- Um parâmetro desse tipo é definido como um ponteiro para função.

Mostrar dados da lista

- Um ponteiro para função é definido da seguinte forma:

```
[tipo_retorno] (*nome_ponteiro) ( [lista_de_parâmetros] )
```

- Por exemplo: `void (*mostra_info) (void *)`

- Este ponteiro aceita qualquer função do tipo *void* e que receba um *ponteiro void* como parâmetro;

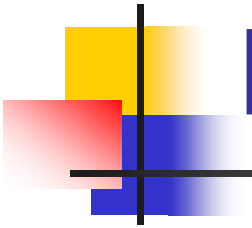
- Dessa forma, a função *mostraLista* pode invocar uma outra função que é responsável por mostrar individualmente cada informação;

Mostrar lista

```
void mostraLista(Lista l, void (*mostraInfo)(void *) ){
   if( listaVazia(l) ){
      printf("Lista vazia!\n");
   }
   else{
      printf("Dados da lista:\n");
      Elemento *p = l.cabeca;
      while(p != NULL){
         mostraInfo(p->info); // Invocação por callback
         p = p->proximo;
      }
   }
}
```

Invocando o *mostraLista*

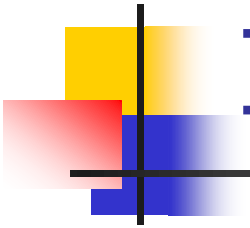
```
#include "Lista.h"

void mostra_float(void *info){   // Função que mostra um dado float...
   float *p = (float *) info;    // a partir de um ponteiro void.
   printf("%.2f\n", *p);
}


int main(int argc, char *argv[]){
   Lista *lista;
   inicializaLista(&lista, sizeof(float));

   // Insere dados na lista...

   mostraLista(lista, mostra_float); // passando a função como parâmetro
}
```



Exercícios

- Implementar as demais operações:
  - insereNaPosicao(lista, info, pos)
  - removeDaPosicao(lista, info, pos)
  - modificaNaPosicao(lista, info, pos)
  - retornaDaPosicao(lista, info, pos)
  - insereEmOrdem(lista, info, (comparador))
  - posicaoElemento(lista, info, (comparador))
  - contemElemento(lista, info, (comparador))

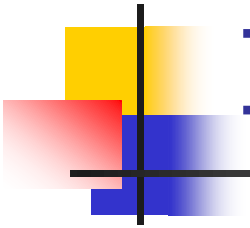
Listas Duplamente Encadeadas



Introdução a LDE

- Listas encadeadas possuem a desvantagem de permitir o percurso apenas em uma direção.
  - Para acessar um elemento já visitado, é preciso uma variável auxiliar que aponta para o elemento anterior.
  - O único meio para acessar outros elementos ainda anteriores consiste em começar de novo o percurso.

- Uma Lista Duplamente Encadeada é uma estrutura de lista que permite o percurso em ambos os sentidos.



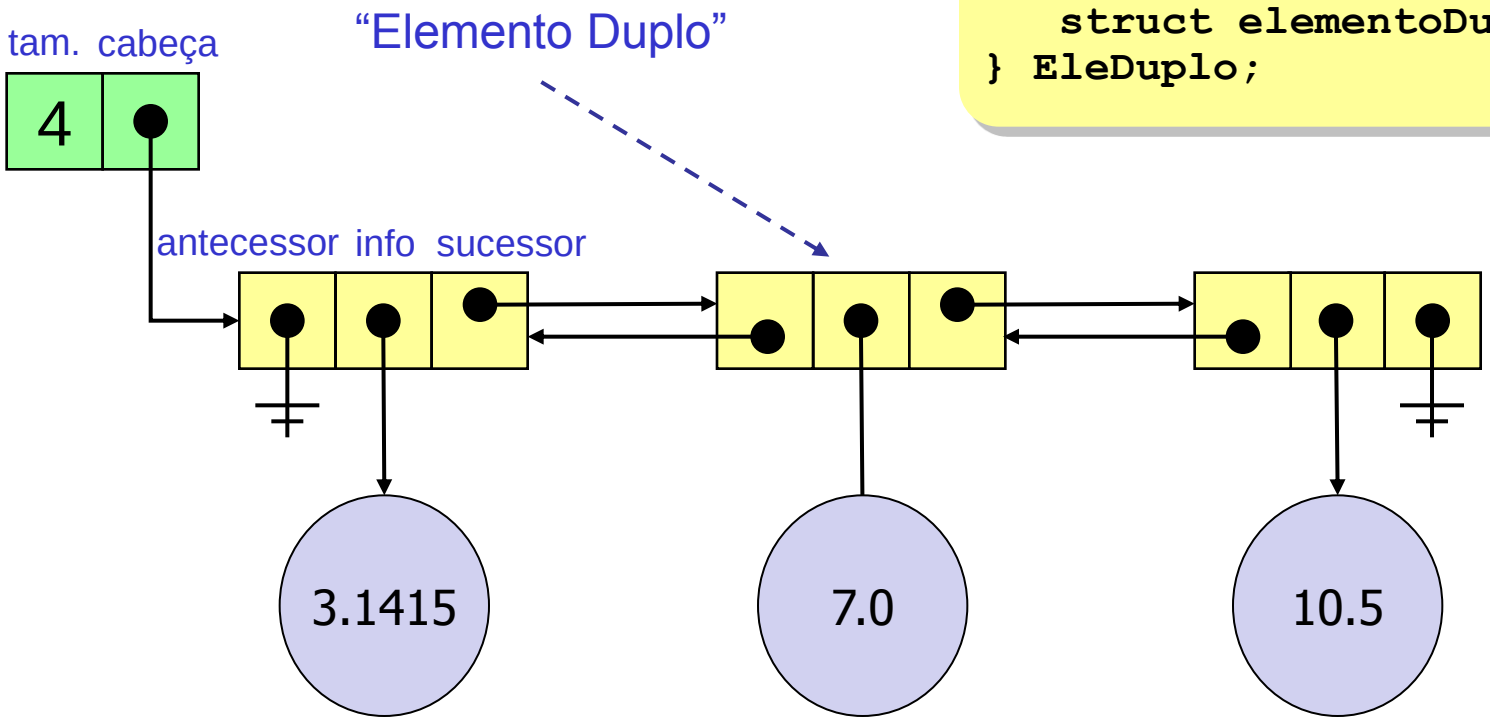
Introdução a LDE

- Listas duplamente encadeadas são úteis para representar conjuntos de eventos ou objetos a serem percorridos em dois sentidos.

- Exemplos:
  - Itinerários de ônibus ou trem;
  - Busca de dados, onde é preciso se mover nos dois sentidos, ajustando o percurso.

LDE: Modelagem

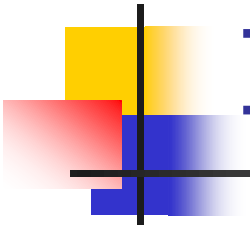
```
typedef struct eleDuplo {
   TipoInfo *info;
   struct elementoDuplo *ant;
   struct elementoDuplo *suc;
} EleDuplo;
```



LDE: Aspecto Funcional

- Operações de criação, teste para ver está vazia e para mostrar a lista são idênticas às da lista encadeada simples.

- A diferença está nas operações de manipulação *(no início, no fim, na posição, em ordem)*:
  - Insere;
  - Remove;
  - Modifica.



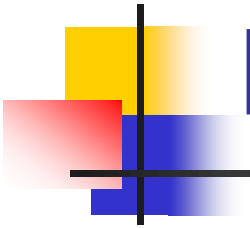
Inserir no início

- Alocar espaço para elemento;
- Alocar espaço para informação;
- Copiar dados do ponteiro passado como parâmetro para a área alocada (usando *memcpy*);
- Campo "sucessor" do novo elemento aponta para onde o "cabeça" da lista aponta;
- "antecessor" aponta para nulo;
- Se houver, "antecessor" do "sucessor" aponta para o novo elemento;
- "cabeça" da lista aponta para novo elemento.

Inserir no início

```
int insereNoInicioDuplo(ListaDupla *l, void *info){
   EleDuplo *novo = (EleDuplo *) malloc(sizeof(EleDuplo));
   if(novo == NULL)
      return 0; // Erro, falta de memória!

   novo->info = malloc(l->tamanhoInfo);
   if(novo->info == NULL){
      free(novo);
      return 0; // Erro, falta de memória!
   }
   memcpy(novo->info, info, l->tamanhoInfo);
   novo->suc = l->cabeca;
   novo->ant = NULL;
   if(novo->suc != NULL)
      novo->suc->ant = novo;
   l->cabeca = novo;
   return 1; // Sucesso.
}
```

Remover do início



- Testar se está vazia → erro!;
- Obter referência do primeiro elemento (cabeça);
- Copiar dados do elemento para a área apontada pelo ponteiro passado como parâmetro (usando *memcpy*);
- Desalocar área da informação;
- Cabeça da lista aponta para o "sucessor" do primeiro elemento;
- Se o elemento possuir sucessor, o antecessor do sucessor será nulo;
- Desalocar elemento;

Remover do início

```
int removeDoInicioDuplo(ListaDupla *l, void *info){
   if( listaDuplaVazia(*l) )
      return ERRO_LISTA_VAZIA;

   EleDuplo *p = l->cabeca;
   memcpy(info, p->info, l->tamanhoInfo);
   free(p->info);
   l->cabeca = p->suc;
   if(p->suc != NULL)
      p->suc->ant = NULL;
   free(p);
   return 1; // Sucesso.
}
```

Inserir no fim

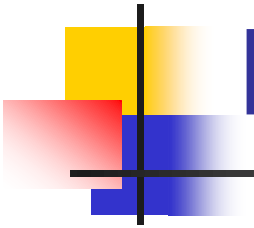
- Se lista vazia, inserir no início.

Caso geral:

- Obter a referência do primeiro elemento;
- Percorrer a lista, enquanto o sucessor do elemento atual não for nulo;
- Quando chegar no último elemento, repetir o procedimento feito em *insereNoInicio*, mas:
  - "sucessor" do novo elemento é nulo;
  - "sucessor" do elemento atual aponta para o novo;
  - "antecessor" do novo elemento aponta para o elemento atual.

Inserir no fim

```
int insereNoFimDuplo(ListaDupla *l, void *info){
   if( listaDuplaVazia(*l) )
      return insereNoInicioDuplo(l, info);

   EleDuplo *novo = (EleDuplo *) malloc(sizeof(EleDuplo));
   if(novo == NULL)
      return 0; // Erro, falta de memória!

   novo->info = malloc(l->tamanhoInfo);
   if(novo->info == NULL){
      free(novo);
      return 0; // Erro, falta de memória!
   }
   EleDuplo *p = l->cabeca;
   while(p->suc != NULL){
      p = p->suc;
   }

   memcpy(novo->info, info, l->tamanhoInfo);
   novo->suc = NULL;
   novo->ant = p;
   p->suc = novo;
   return 1; // Sucesso.
}
```



Exercícios

- Implementar o TAD ListaDupla.

- Além das operações apresentadas em aula, implementar as demais operações:
  - removeDoFim;
  - insereNaPosicao;
  - removeDaPosicao;
  - insereEmOrdem;
  - ...