Estrutura de Dados I

TAD Listas Auto-referenciadas

Professor: Vinícius Fernandes Soares Mota
www.inf.ufes.br/~vinicius.mota
vinicius.mota@inf.ufes.br

Licença de uso e créditos

Universidade Federal do Espírito Santo

- Sob licença Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.
- Apresentação baseada em:
  - Programação descomplicada – Prof. André Backes (UFU)
  - Projeto de Algoritmos. Nivio Ziviani.
  - Slides da prof. Patrícia D. Costa

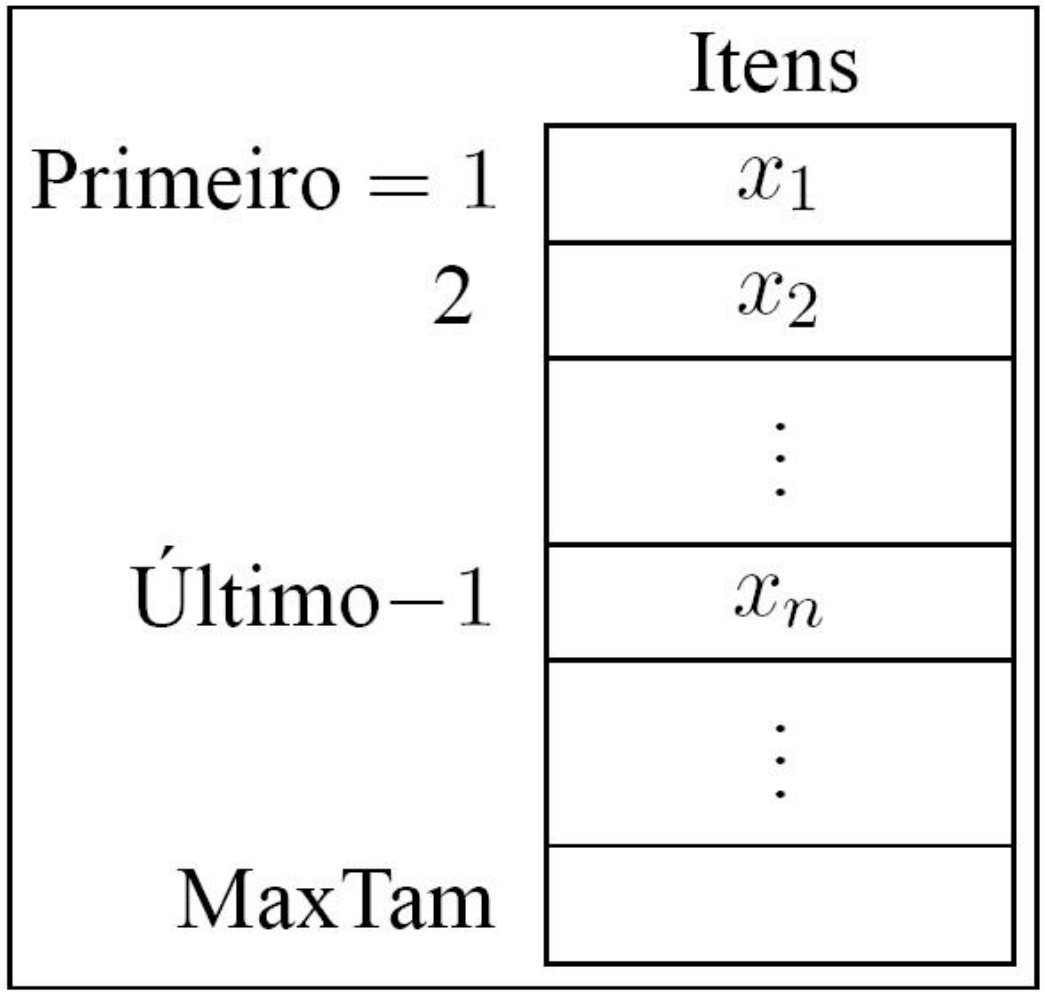
Leitura recomendada

- Livros:
  - Projeto de Algoritmos (Nivio Ziviani): Capítulo 3;
  - Introdução a Estruturas de Dados (Celes, Cerqueira e Rangel): Capítulo 10;
  - Estruturas de Dados e seus Algoritmos (Szwarefiter, et. al): Capítulo 2;
  - Algorithms in C (Sedgewick): Capítulo 3;

Visão Geral

- Estrutura de Dados básicas
- TAD LISTA
- Operações em uma lista
- Implementações
  - Por vetores (arranjos)
  - Auto-referenciada (ponteiros)
  - Listas duplamente encadeadas

Listas Lineares em Alocação Sequencial




```
#define InicioVetor 1
#define MaxTam  1000

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

struct tipolista{
  TipoItem Item[MaxTam];
  Posicao Primeiro, Ultimo;
};
```

Lista por arranjo vantagens e desvantagens

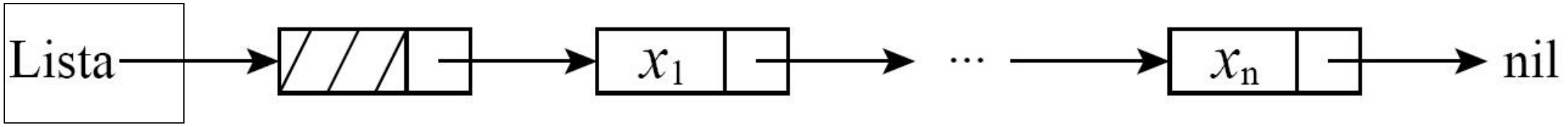
- Vantagem: Simplicidade de implementação e acesso direto ao elemento na i-ésima posição.
- Desvantagens:
  - custo para inserir ou retirar itens da lista, que pode causar um deslocamento de todos os itens, no pior caso;
  - em aplicações em que não existe previsão sobre o crescimento da lista, a utilização de arranjos em linguagens como o Pascal e o C pode ser problemática pois, neste caso, o tamanho máximo da lista tem de ser definido em tempo de compilação.

Listas com alocação não sequencial e dinâmica

- Cada item é encadeado com o seguinte mediante uma variável do tipo Ponteiro.
- Permite utilizar posições não contíguas de memória.
- É possível inserir e retirar elementos sem necessidade de deslocar os itens seguintes da lista.
- Há uma célula cabeça para simplificar as operações sobre a lista
- Estrutura Encadeada

Listas com alocação não sequencial e dinâmica

- Cada item é encadeado com o seguinte mediante uma variável do tipo Ponteiro.
- Permite utilizar posições não contíguas de memória.
- É possível inserir e retirar elementos sem necessidade de deslocar os itens seguintes da lista.
- Há uma célula cabeça para simplificar as operações sobre a lista



Lista com alocação não sequencial e dinâmica: vantagens e desvantagens

- Vantagens:
  - Permite inserir ou retirar itens do meio da lista a um custo constante (importante quando a lista tem de ser mantida em ordem).
  - Bom para aplicações em que não existe previsão sobre o crescimento da lista (o tamanho máximo da lista não precisa ser definido *a priori*).
- Desvantagem: utilização de memória extra para armazenar os ponteiros.

Listas encadeadas

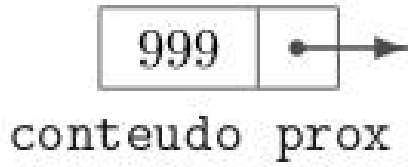
- Simples - Sem sentinela
  - Apenas o endereço do próximo elemento da lista

- Com sentinela
  - Mantém-se um apontador para o último elemento da lista

Lista encadeada simples

- A lista é constituída de células.
- Cada célula contém um item da lista e um ponteiro para a célula seguinte.
- O registro (struct) TipoLista contém um ponteiro para a célula cabeça e um ponteiro para a última célula da lista.

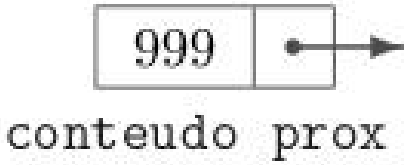
```
typedef struct {
    int        conteudo;
    celula *prox;
 }celula;
```

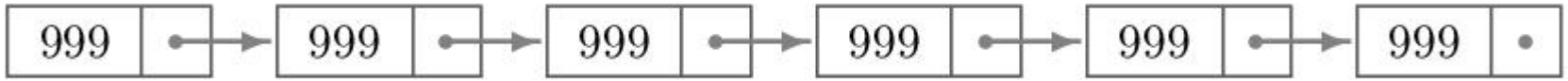


Lista encadeada simples

- A lista é constituída de células.
- Cada célula contém um item da lista e um ponteiro para a célula seguinte.
- O registro (struct) TipoLista contém um ponteiro para a célula cabeça e um ponteiro para a última célula da lista.

```
typedef struct {
    int        conteudo;
    celula *prox;
 }celula;
```

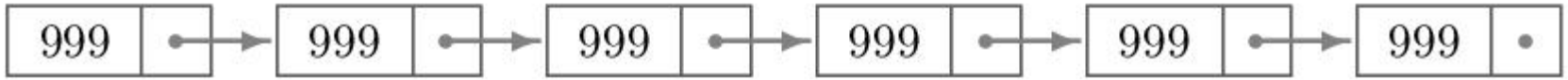




A figura pode dar a falsa impressão de que as células da lista ocupam posições consecutivas na memória. Na realidade, as células estão tipicamente espalhadas pela memória de maneira imprevisível.

Exercício

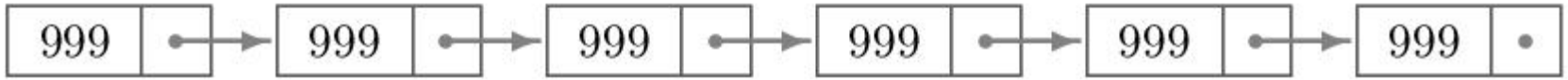
- Faça uma função para imprimir o conteúdo da lista anterior:




```
typedef struct {
    int        conteudo;
    celula *prox;
 }celula;
```

Exercício

- Faça uma função para imprimir o conteúdo da lista anterior:




```
typedef struct {
    int        conteudo;
    celula *prox;
  }celula;
```

```
void imprime (celula *le) {
   celula *p;
   for (p = le; p != NULL; p = p->prox)
      printf ("%d\n", p->conteudo);
}
```

Versão Recursiva

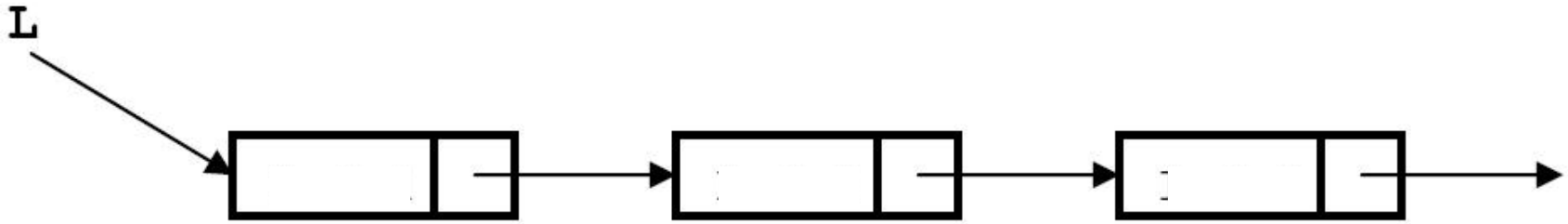
```
void imprime (celula *le) {
   if (le != NULL) {
      printf ("%d\n", le->conteudo);
      imprime (le->prox);
   }
}
```

Operações

- Mesmo conjunto de operações
  - Criar
  - Inserir
  - Remover
  - Destruir
  - Imprimir
  - etc etc

Lista Encadeada Simples

- Considere uma lista encadeada simples, sem célula cabeçalho e sem “sentinela”:




```
typedef struct tipoitem TipoItem;
typedef struct celula_str TipoLista;
```

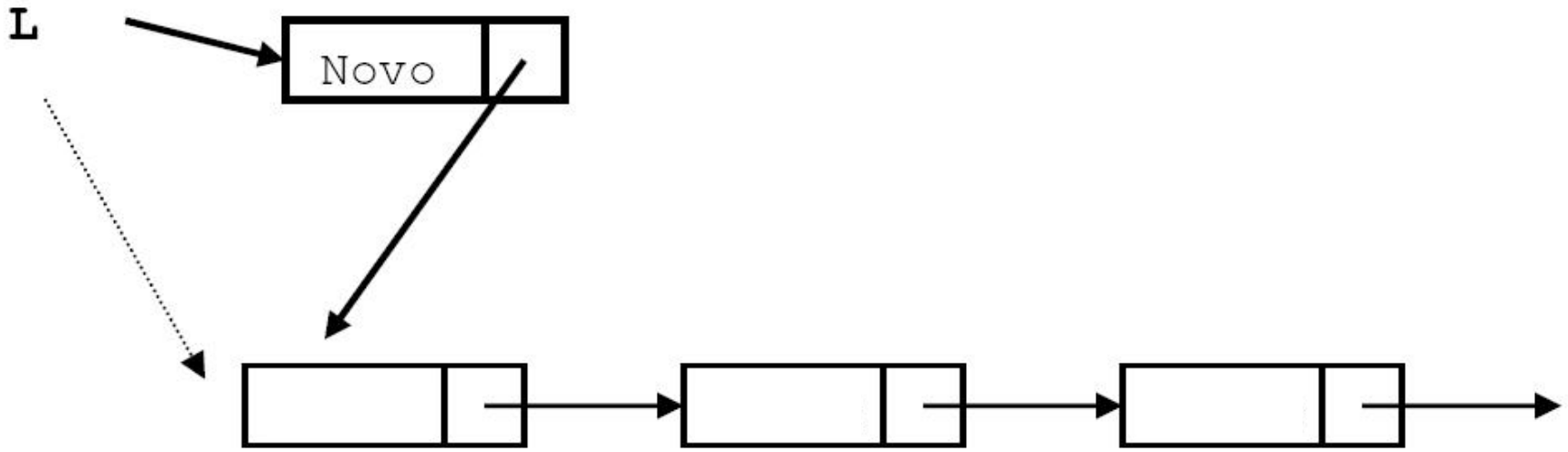
Lista Encadeada Simples

Universidade Federal do Espírito Santo

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

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

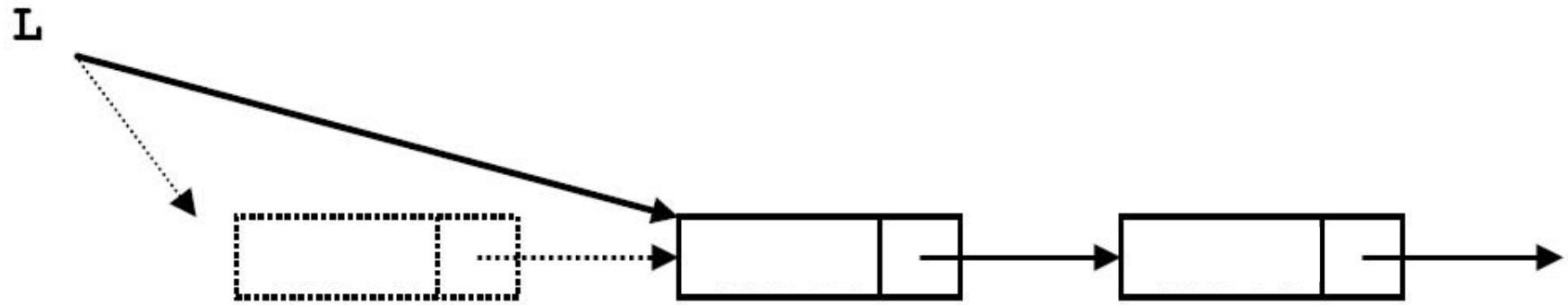
Função Inserir

```
TipoLista* lst_insere (TipoLista* l, TipoItem* item)
{
    /* cria uma nova celula */
    TipoLista* novo = (TipoLista*) malloc(sizeof(TipoLista));
    novo->Item = *item;
    novo->Prox = l;
    return novo;
}
```

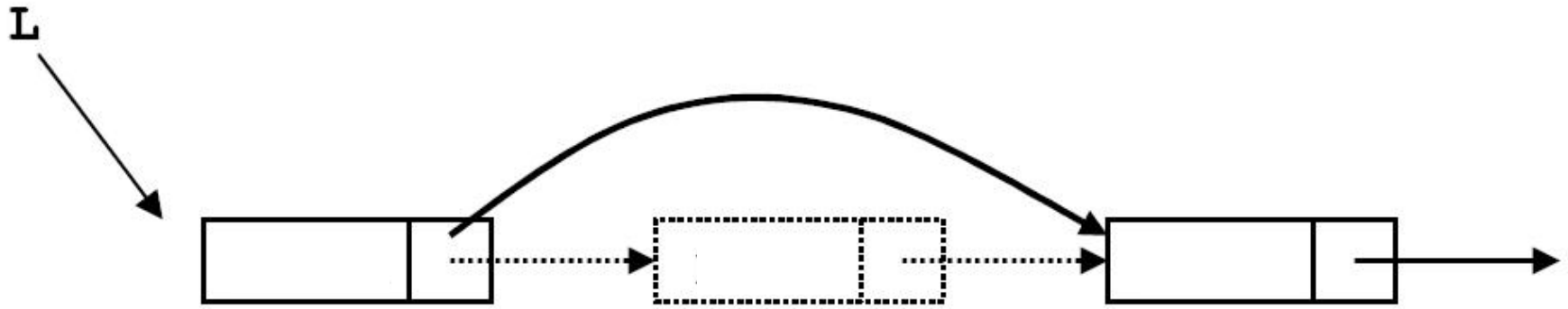


Função Retirar

- 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

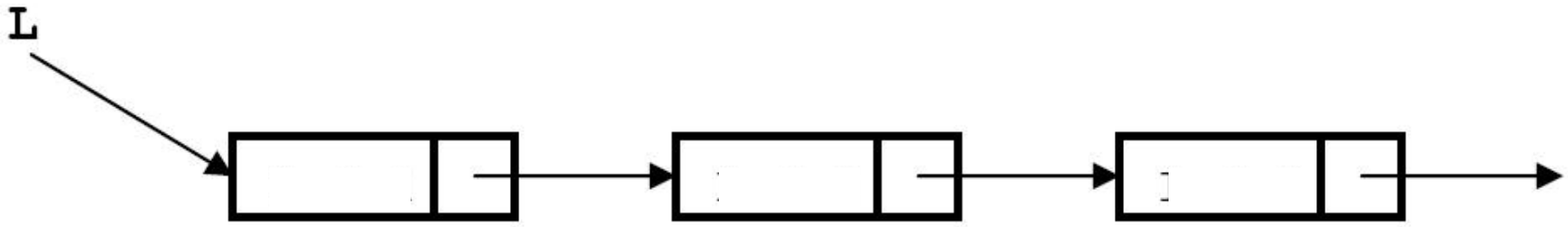


Função Retirar (código)

```
TipoLista* lst_retira (TipoLista* l, int v)
{
  TipoLista* ant = NULL;
  TipoLista* p = l;
  while (p != NULL && p->Item.valor != v)
  { ant = p;
    p = p->Prox;}
  if (p == NULL)
        return l;
  if (ant == NULL){
    l = p->Prox;}
  else{
    ant->Prox = p->Prox;}
  free(p);
  return l;
}
```

Exercício

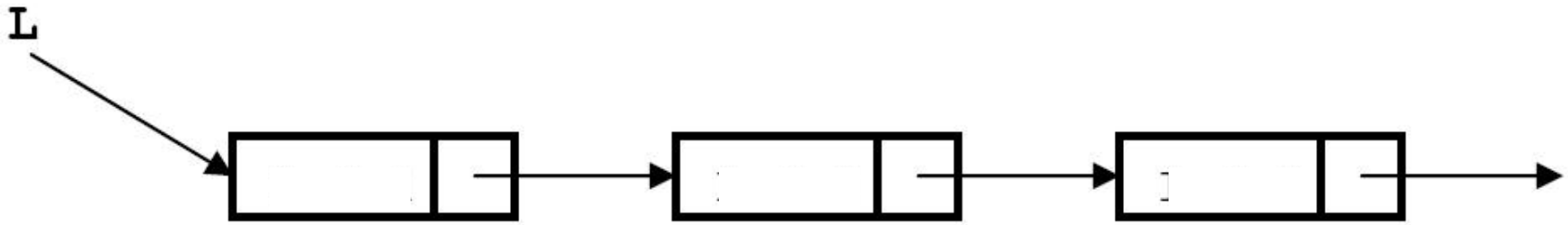
- Implemente a função Busca:




```
TipoLista* busca (TipoLista* l, int v)
{
        ....
}
```

Exercício

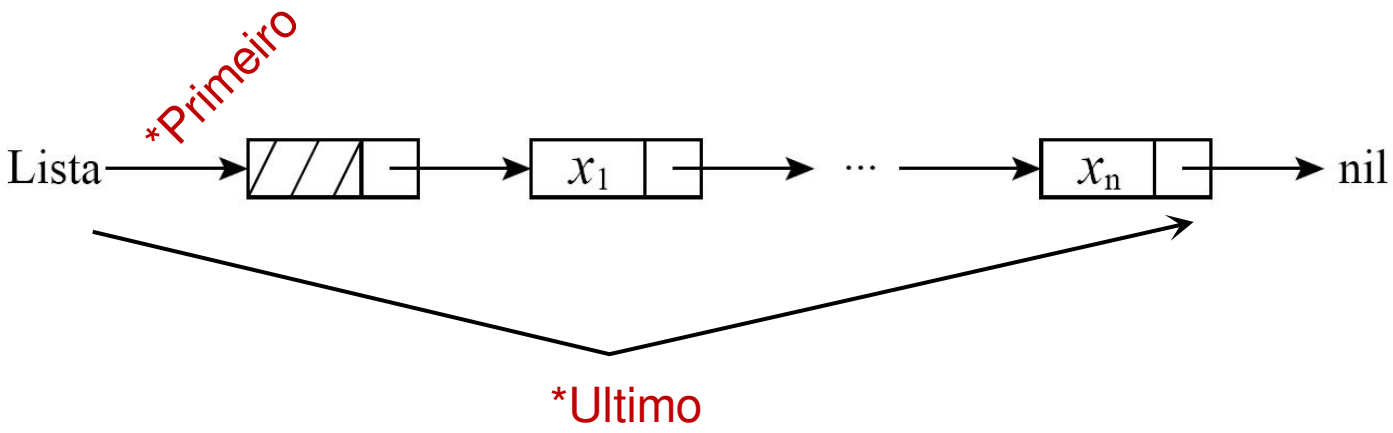
- Implemente a função Busca:




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

Estrutura da Lista com Sentinela

```
#include <stdio.h>
#include <stdlib.h>
#include "lista.h"

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

typedef struct celula_str Celula;

struct celula_str {
    TipoItem Item;
    Celula* Prox;
};

struct tipolista{
    Celula* Primeiro, Ultimo;
};
```



Estrutura da Lista encadeada com sentinela

Universidade Federal do Espírito Santo

- Operações devem atualizar os ponteiros:
  - proximo
  - primeiro
  - ultimo

Estrutura da Lista encadeada com sentinela
lista.h

Universidade Federal do Espírito Santo

```
typedef int Posicao;
typedef struct tipoitem TipoItem;
typedef struct tipolista TipoLista;

TipoLista* InicializaLista();
int Vazia (TipoLista* Lista);
void Insere (TipoItem* x, TipoLista* Lista);
void Retira (TipoLista* Lista, int v);
void Imprime (TipoLista* Lista);
TipoItem* InicializaTipoItem();
void ModificaValorItem (TipoItem* x, int valor);
void ImprimeTipoItem(TipoItem* x);
```

Implementação TAD Lista com Ponteiros

Universidade Federal do Espírito Santo

```
TipoLista* InicializaLista()
{
  TipoLista* lista = (TipoLista*)malloc(sizeof(TipoLista));
  Lista->Ultimo = NULL;
  Lista->Primeiro = NULL;
 return lista;
}
```

Implementação TAD Lista com Ponteiros

Universidade Federal do Espírito Santo

```
int Vazia (TipoLista* Lista)
{
 return (Lista->Primeiro == NULL);
}
```

```
void Insere (TipoItem* x, TipoLista *Lista){
  Celula* novo = (Celula*) malloc(sizeof(Celula));
  if (lista->Ultimo == NULL)
        lista->Primeiro = lista->Ultimo = novo;
  else
  {     Lista->Ultimo->Prox = novo;
        Lista->Ultimo = Lista->Ultimo->Prox;}
  Lista->Ultimo->Item =*x;
  Lista->Ultimo->Prox = NULL;
}
```

```
void Retira (TipoLista *Lista, int v)
{
   Celula* ant = NULL;
   Celula* p = Lista->Primeiro;
   while (p != NULL && p->Item.valor != v)
   { ant = p;
     p = p->Prox;}
   if (p == NULL)
            return;
   if (p == Lista->Primeiro && p == Lista->Ultimo){
        Lista->Primeiro = Lista->Ultimo = NULL;
             free (p);
             return; }
   if (p == Lista->Ultimo){
            Lista->Ultimo = ant; ant->Prox = NULL; free (p); return;}
   if (p == Lista->Primeiro)
            Lista->Primeiro = p->Prox;
   else
      ant->Prox = p->Prox;
  free(p);
}
```

TipoItem

- Como o TipoItem é opaco, precisamos de operações no TAD que manipulam este tipo:
  - InicializaTipoItem: cria um TipoItem
  - ModificaValorTipoItem: modifica o campo valor de um TipoItem
  - ImprimeTipoItem: Imprime o campo valor de um TipoItem

TipoItem (cont.)

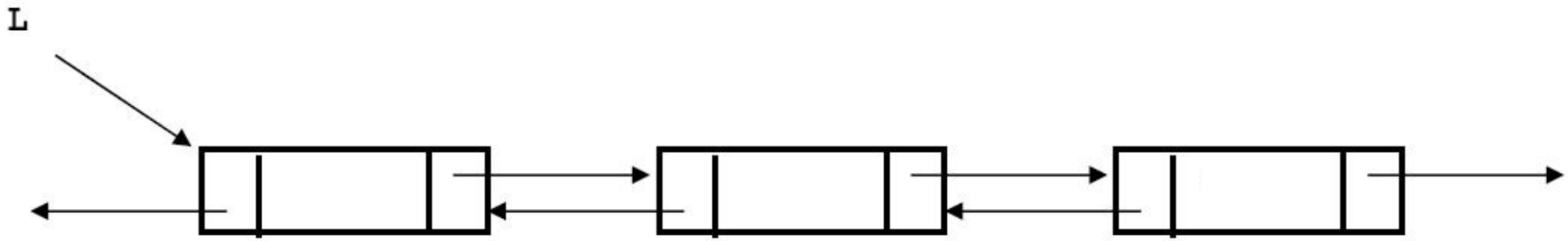
```
TipoItem* InicializaTipoItem() {
   TipoItem* item = (TipoItem*)malloc(sizeof(TipoItem));
   return item;
}

void ModificaValorItem (TipoItem* item, int valor) {
   item->valor = valor;
}

void ImprimeTipoItem (TipoItem* item){
   printf ("Campo valor: %d ", item->valor);
}
```

Outros tipos de lista

- Lista circular
- Lista duplamente-encadeada
- Listas hetereogeneas

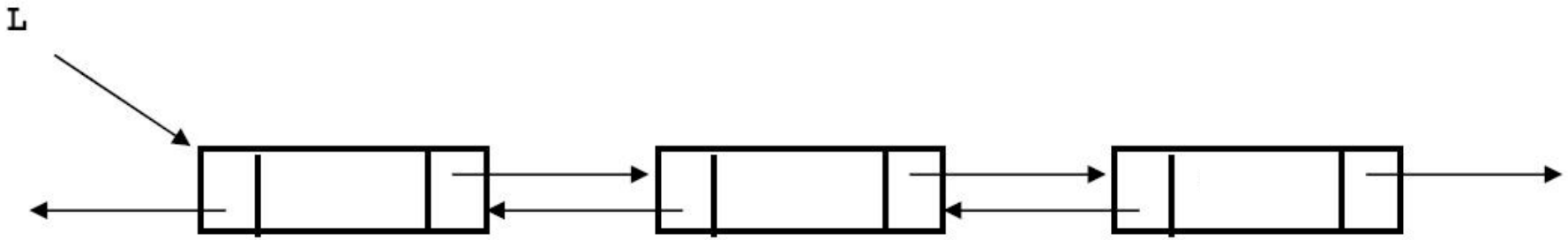
Listas Duplamente Encadeadas

- 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

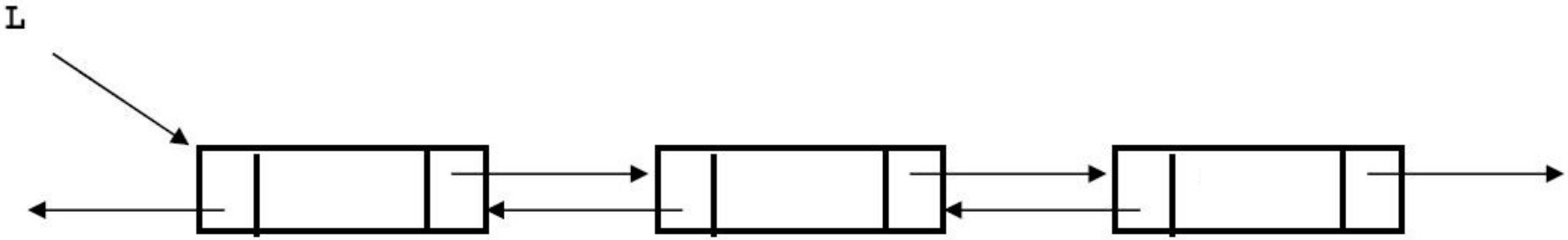

```
typedef struct tipoitem TipoItem;
typedef struct celula_str TipoListaDpl;
```



Listas Duplamente Encadeadas

Universidade Federal do Espírito Santo

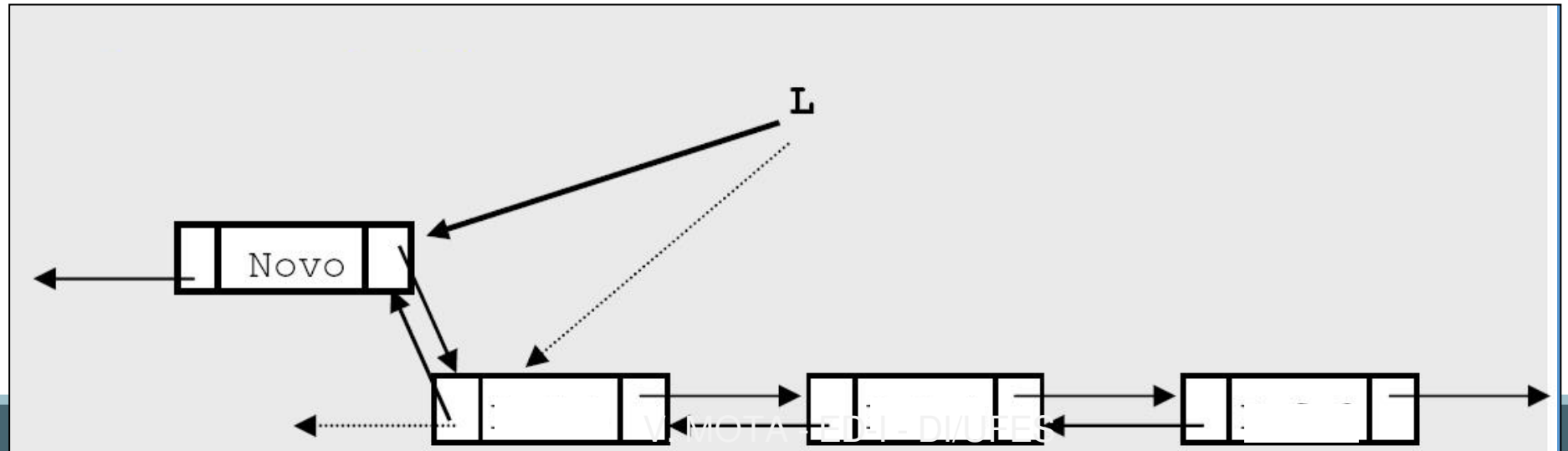
```
struct tipoitem{
    int valor;
    /* outros componentes */
};

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



Função Inserir (duplamente encadeada)

Universidade Federal do Espírito Santo

```
/* 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)

Universidade Federal do Espírito Santo

- 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

Universidade Federal do Espírito Santo

```
/* 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

Universidade Federal do Espírito Santo

- 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

Universidade Federal do Espírito Santo

```
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 heterogênias, 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

Universidade Federal do Espírito Santo

```
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

Universidade Federal do Espírito Santo

```
/* 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

Universidade Federal do Espírito Santo

```
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

Universidade Federal do Espírito Santo

```
/* 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){
```

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

```
    }
    return a;
}
```

Listas Heterogêneas - Exercícios

Universidade Federal do Espírito Santo

```
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;
}
```

Dúvidas?

Lista encadeada

Vinícius **F**ernandes **S**oares **M**ota