

REVISÃO DE C

Vanessa Braganholo
Estruturas de Dados e Seus
Algoritmos

REVISÃO DE C

Ponteiros

Alocação dinâmica de memória

Tipos Estruturados de Dados

Listas Encadeadas

Recursão

PONTEIROS |

PONTEIROS

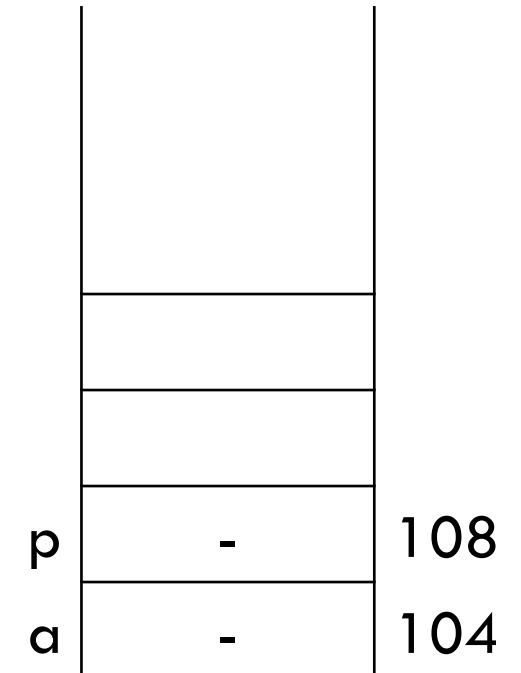
C permite o armazenamento e manipulação de valores de endereço de memória

Para cada tipo de dados existente, há um tipo de ponteiro que pode armazenar endereços de memória de variáveis daquele tipo

EXEMPLO DE DECLARAÇÃO DE VARIÁVEIS INTEIRO E PONTEIRO PARA INTEIRO

```
/* variável inteiro */  
int a;  
/* variável ponteiro p/ inteiro */  
int *p;
```

Memória Principal



OPERADORES

Operador Unário &

- **Endereço de**
- Aplicado a uma variável x , resulta no **endereço da posição de memória** reservada para a variável x

Operador Unário *

- **Conteúdo do endereço apontado por**
- Aplicado a variáveis do tipo ponteiro, acessa o **conteúdo do endereço de memória** armazenado pela variável ponteiro

```
int a, *p, c;  
/* a recebe o valor 5 */  
a = 5;
```

Memória Principal

c	-	112
p	-	108
a	5	104

```
int a, *p, c;

/* a recebe o valor 5 */

a = 5;

/* p recebe o endereço de a (p aponta
para a) */

p = &a;
```

Memória Principal

c	-	112
p	104	108
a	5	104


```
int a, *p, c;

/* a recebe o valor 5 */

a = 5;

/* p recebe o endereço de a (p aponta
para a) */

p = &a;

/* posição de memória apontada por p
recebe 6 */



*p = 6;


```

Memória Principal

c	-	112
p	104	108
a	6	104

```
int a, *p, c;

/* a recebe o valor 5 */

a = 5;

/* p recebe o endereço de a (p aponta
para a) */

p = &a;

/* posição de memória apontada por p
recebe 6 */

*p = 6;

/* c recebe o valor armazenado na posição
de memória apontada por p */

c = *p;
```

Memória Principal

c	6	112
p	104	108
a	6	104

EXEMPLO COM ERRO

```
int main ( void ) {  
    int a, b, *p;  
    a = 2;  
    *p = 3;  
    b = a + (*p);  
    printf(" %d ", b);  
    return 0;  
}
```

Erro na atribuição *p = 3

- Utiliza a memória apontada por p para armazenar o valor 3, sem que p tivesse sido inicializada
- Armazena 3 num espaço de memória **desconhecido**

Memória Principal

	3	632
	...	
		116
		112
p	(LIXO)	108
a	2	104

EXEMPLO CORRIGIDO

```
int main ( void ) {  
    int a, b, c, *p;  
    a = 2;  
    p = &c;  
    *p = 3;  
    b = a + (*p);  
    printf(" %d ", b);  
    return 0;  
}
```

- Atribuição *p = 3**
- p aponta para c
 - Atribuição armazena 3 no espaço de memória da variável c

Memória Principal

		114
c	3	112
p	112	108
a	2	104

PASSAGEM DE PONTEIROS PARA FUNÇÕES

C usa passagem de parâmetros **por valor**

- Alterações nos valores dos parâmetros não afetam as variáveis que foram usadas na chamada da função
- Caso seja necessário que as alterações sejam refletidas nas variáveis usadas na chamada, pode-se **passar endereços das variáveis como parâmetro** ao invés de variáveis comuns
- Nesse caso os parâmetros da função precisam ser ponteiros

```

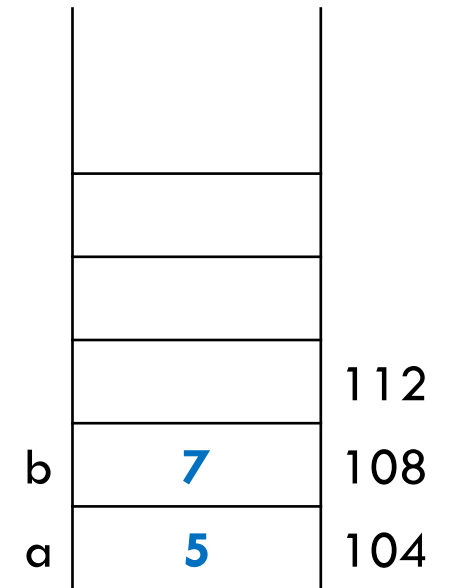
#include <stdio.h>

void troca (int *px, int *py ) {
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}

int main ( void ) {
    int a = 5, b = 7;
    troca(&a, &b); /* passamos os endereços das variáveis */
    printf("%d %d \n", a, b);
    return 0;
}

```

Memória Principal



```

#include <stdio.h>

void troca (int *px, int *py ) {
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}

int main ( void ) {
    int a = 5, b = 7;
    troca(&a, &b); /* passamos os endereços das variáveis */
    printf("%d %d \n", a, b);
    return 0;
}

```

Memória Principal

		120
py	108	116
px	104	112
b	7	108
a	5	104

```

#include <stdio.h>

void troca (int *px, int *py ) {
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}

int main ( void ) {
    int a = 5, b = 7;
    troca(&a, &b); /* passamos os endereços das variáveis */
    printf("%d %d \n", a, b);
    return 0;
}

```

Memória Principal

temp	-	120
py	108	116
px	104	112
b	7	108
a	5	104


```

#include <stdio.h>

void troca (int *px, int *py ) {
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}

int main ( void ) {
    int a = 5, b = 7;
    troca(&a, &b); /* passamos os endereços das variáveis */
    printf("%d %d \n", a, b);
    return 0;
}

```

Memória Principal

temp	5	120
py	108	116
px	104	112
b	7	108
a	5	104

```

#include <stdio.h>

void troca (int *px, int *py ) {
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}

int main ( void ) {
    int a = 5, b = 7;
    troca(&a, &b); /* passamos os endereços das variáveis */
    printf("%d %d \n", a, b);
    return 0;
}

```

Memória Principal

temp	5	120
py	108	116
px	104	112
b	7	108
a	7	104

```

#include <stdio.h>

void troca (int *px, int *py ) {
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}

int main ( void ) {
    int a = 5, b = 7;
    troca(&a, &b); /* passamos os endereços das variáveis */
    printf("%d %d \n", a, b);
    return 0;
}

```

Memória Principal

temp	5	120
py	108	116
px	104	112
b	5	108
a	7	104

```

#include <stdio.h>

void troca (int *px, int *py ) {
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}

int main ( void ) {
    int a = 5, b = 7;
    troca(&a, &b); /* passamos os endereços das variáveis */
    printf("%d %d \n", a, b);
    return 0;
}

```

Imprime 7 5

Memória Principal

		120
		116
		112
b	5	108
a	7	104

VETORES SÃO PONTEIROS

Um vetor é alocado em posições contíguas de memória

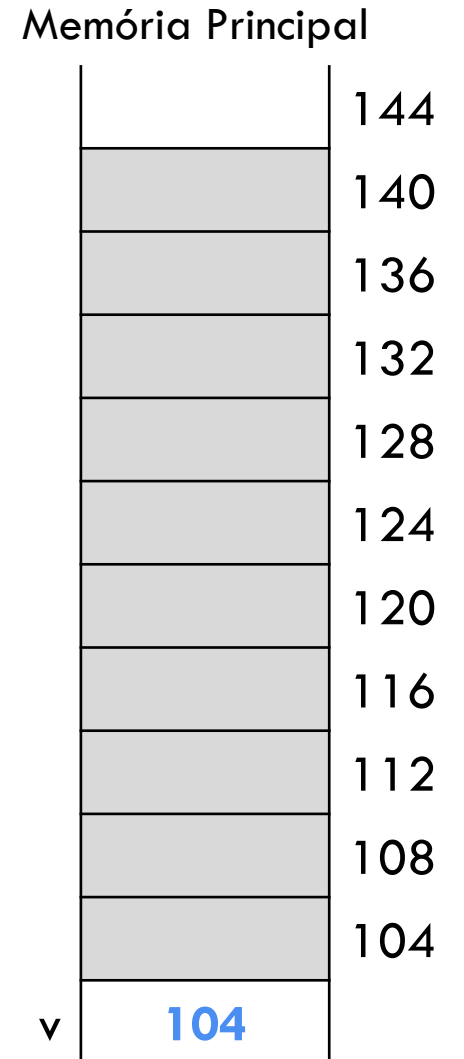
Exemplo:

- Vetor v de 10 elementos inteiros
- Espaço de memória de $v = 10 \times \text{valores inteiros de 4 bytes} = 40 \text{ bytes}$

Alocação estática! (espaço de memória é reservado no momento da declaração do vetor)

Nome do vetor é um ponteiro que aponta para o endereço inicial do vetor

```
int v[10];
```



VETORES E PONTEIROS

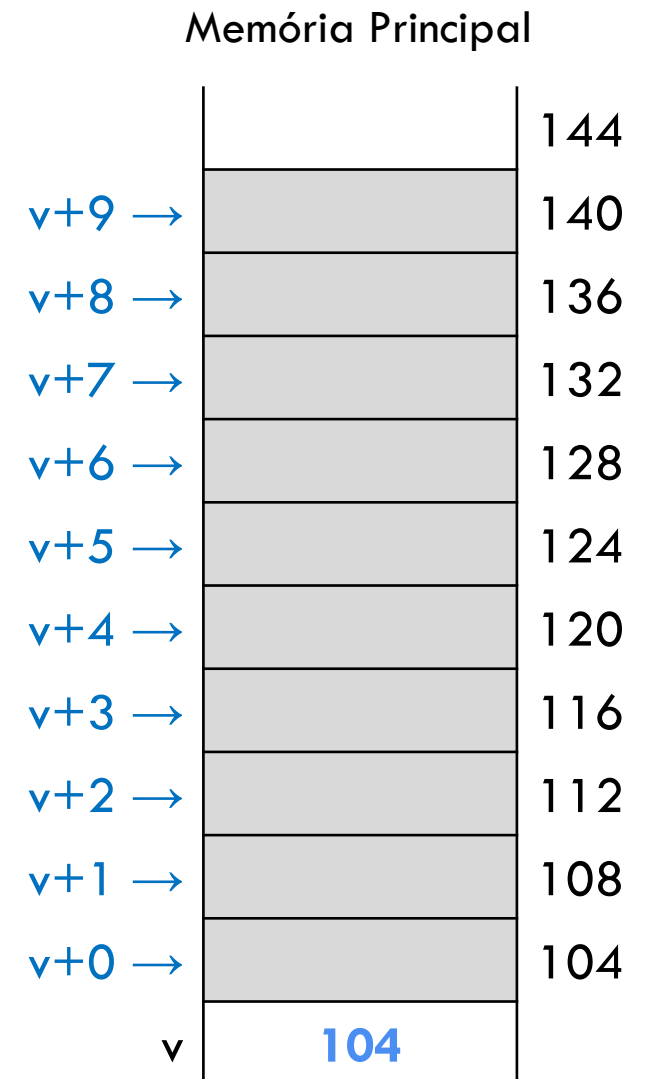
C permite aritmética de ponteiros

Exemplo:

- $v + 0$: primeiro elemento de v
- $v + 1$: segundo elemento de v
- ...
- $v + 9$: último elemento de v

Elementos do vetor podem ser acessados usando colchetes:

- $v[0]$ é o primeiro elemento do vetor



```
/* Cálculo da média de 10 números reais */
```

```
#include <stdio.h>
```

```
int main ( void ) {
```

```
float v[10];
```

```
float med = 0.0;
```

```
int i;
```

```
/* leitura dos valores */
```

```
for ( i = 0; i < 10; i++)
```

```
scanf("%f", &v[i]);
```

```
/* cálculo da média */
```

```
for ( i = 0; i < 10; i++)
```

```
med = med + v[i];
```

```
med = med / 10;
```

```
/* exibição do resultado */
```

```
printf ( "Media = %f \n", med );
```

```
return 0;
```

```
}
```

Acesso ao endereço da
i-ésima posição de v

Acesso ao conteúdo de
v[i]

Memória Principal

	i	-	148
	med	0.0	144
v+9 →			140
v+8 →			136
v+7 →			132
v+6 →			128
v+5 →			124
v+4 →			120
v+3 →			116
v+2 →			112
v+1 →			108
v+0 →			104
v		104	

PASSAGEM DE PARÂMETRO VETOR PARA FUNÇÃO

Como um vetor na verdade é um ponteiro, passar um vetor como parâmetro para uma função **exige** que o **parâmetro da função seja um ponteiro**


```
/* Cálculo da média de 10 números reais */
```

```
#include <stdio.h>
```

```
float media (int n, float *v) {
```

```
    int i;
```

```
    float s = 0.0;
```

```
    for (i = 0; i < n; i++)
```

```
        s += v[i];
```

```
    return s/n;
```

```
}
```

```
int main ( void ) {
```

```
    float v[10];
```

```
    float med;
```

```
    int i;
```

```
    for ( i = 0; i < 10; i++ )
```

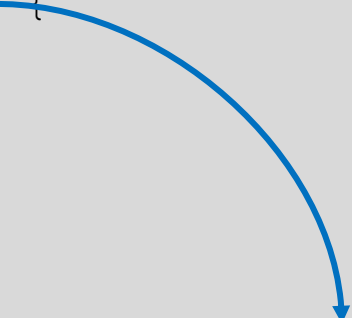
```
        scanf("%f", &v[i]);
```

```
    med = media(10, v);
```

```
    printf ( "Media = %.2f \n", med);
```

```
    return 0;
```

```
}
```



Parâmetro do tipo
ponteiro para float

FUNÇÃO PODE ALTERAR VALORES DO VETOR

Como o que é passado é o endereço do vetor, e não uma cópia dos valores,

- a **função pode alterar os valores do vetor**, e
- as **alterações serão refletidas no programa principal**

Vocês se lembram que isso também acontece em Python?

```
/* Incrementa elementos de um vetor */
#include <stdio.h>

void incr_vetor ( int n, int *v ) {
    int i;
    for (i = 0; i < n; i++)
        v[i]++;
}

int main ( void ) {
    int a[ ] = {1, 3, 5};
    incr_vetor(3, a);
    printf("%d %d %d \n", a[0], a[1], a[2]);
    return 0;
}
```

Imprime 2 4 6

ALOCAÇÃO DINÂMICA DE MEMÓRIA

ALOCAÇÃO DINÂMICA DE MEMÓRIA

Alocação dinâmica é usada quando não se sabe de antemão quanto espaço de memória será necessário

- Exemplo, quando não é possível determinar o tamanho de um vetor de antemão

Alocação dinâmica

- Espaço de memória é requisitado em tempo de execução
- Espaço permanece alocado até que seja explicitamente liberado
- Depois de liberado, espaço pode ser disponibilizado para outros usos e não pode mais ser acessado
- Espaço alocado e não liberado explicitamente é liberado ao final da execução do programa

ALOCAÇÃO ESTÁTICA X ALOCAÇÃO DINÂMICA

Alocação Estática

Memória Principal:

Código do Programa
Variáveis Globais e Locais estáticas

Alocação Dinâmica

Memória Principal:

Código do Programa
Variáveis Globais e Locais estáticas
Variáveis Alocadas Dinamicamente
Memória Livre que pode ser alocada pelo programa

BIBLIOTECA STDLIB.H

A biblioteca `stdlib.h` contém funções e constantes para alocação dinâmica de memória

FUNÇÕES IMPORTANTES DA STDBLI.H

sizeof(tipo)

- Retorna o número de bytes ocupado por um **tipo**

malloc(tamanho)

- Recebe como parâmetro o **tamanho** do espaço de memória (em bytes) se deseja alocar
- Retorna um ponteiro genérico para o endereço inicial da área de memória alocada, se houver espaço livre
 - Ponteiro genérico é representado por void*
 - Ponteiro é convertido automaticamente para o tipo apropriado
 - Ponteiro pode ser convertido explicitamente
- Retorna um endereço nulo (NULL) se não houver espaço livre

EXEMPLO

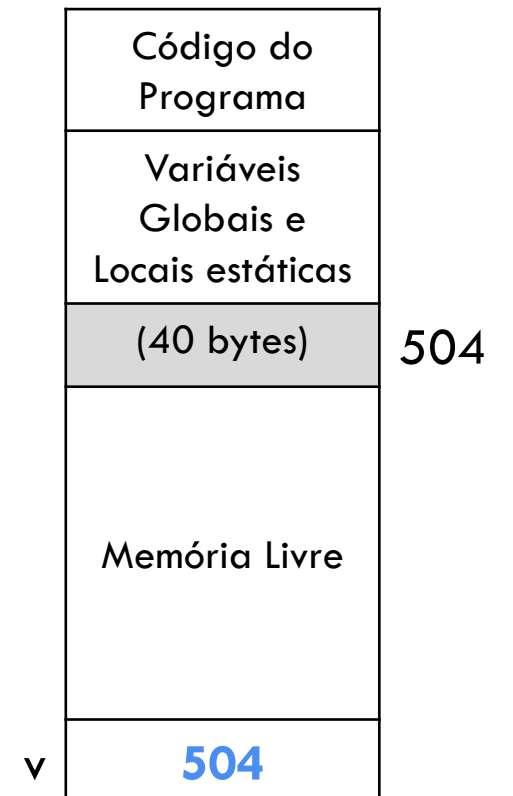
Alocação dinâmica de um vetor de inteiros com 10 elementos

- malloc retorna o endereço do espaço de memória alocado
- ponteiro de inteiro (v) recebe endereço inicial do espaço de memória alocado

```
int *v;  
v = (int*) malloc(10 * sizeof(int));
```

- no programa, v pode ser tratado como um vetor alocado estaticamente

Memória Principal



TRATAMENTO DE ERRO

```
...  
v = (int*) malloc(10*sizeof(int));  
if (v == NULL) {  
    printf("Memoria insuficiente.\n");  
    exit(1); /* aborta o programa e retorna 1 para o sist. operacional */  
}  
...
```

LIBERAÇÃO DE MEMÓRIA

free(ponteiro)

- Libera a memória apontada por **ponteiro**
- A posição de memória apontada por ponteiro **deve ter sido alocada dinamicamente**

```
/* Cálculo da média de um número n de números reais */
#include <stdio.h>
#include <stdlib.h>
int main ( void ) {
    int i,n;
    float *v;
    float med;
    scanf("%d", &n); /* leitura do número n de valores */
    v = (float*) malloc(n*sizeof(float)); /* alocação dinâmica */
    if (v==NULL) {
        printf("Memoria insuficiente.\n");
        return 1;
    }
    for (i = 0; i < n; i++)
        scanf("%f", &v[i]);
    med = media(n,v); /* chama a função de cálculo de média */
    printf("Media = %.2f \n", med);
    free(v); //libera memória */
    return 0;
}
```

PONTEIROS E TIPOS ESTRUTURADOS

TIPO ESTRUTURA (STRUCT)

Tipo de dado composto por outros elementos (campos)

Campos são acessados através do operador de acesso “ponto” (.)

```
struct ponto2d { /* declara ponto2d do tipo struct */
    float x;
    float y;
};
...
struct ponto2d p; /* declara p como variável do tipo struct ponto2d */
...
p.x = 10.0; /* acessa os elementos de ponto2d */
p.y = 5.0;
```

EXEMPLO COMPLETO

```
/* Captura e imprime as coordenadas de um ponto qualquer */
#include <stdio.h>
struct ponto2d {
    float x;
    float y;
};
int main (void) {
    struct ponto2d p;
    printf("Digite as coordenadas do ponto(x y): ");
    scanf("%f %f", &p.x, &p.y);
    printf("O ponto fornecido foi: (%.2f,%.2f)\n", p.x, p.y);
    return 0;
}
```

PONTEIROS PARA ESTRUTURAS

Acesso ao **valor** de um campo **x** de uma variável estrutura **p**: **p.x**

Acesso ao **valor** de um campo **x** de uma **variável ponteiro pp**: **pp->x**

Acesso ao **endereço** do campo **x** de uma **variável ponteiro pp**: **&pp->x**

```
struct ponto2d *pp;  
...  
pp->x = 12.0;
```


ALOCAÇÃO DINÂMICA DE ESTRUTURAS

Tamanho do espaço de memória alocado dinamicamente é dado pelo operador **sizeof** aplicado sobre o tipo estrutura

A função **malloc** retorna o endereço do espaço alocado, que é então convertido para o tipo ponteiro da estrutura

```
struct ponto2d* p;  
p = (struct ponto2d*) malloc (sizeof(struct ponto2d));  
...  
p->x = 12.0;
```

É MAIS SIMPLES DEFINIR UM NOME PARA O TIPO

typedef

- permite criar nomes de tipos
- útil para abreviar nomes de tipos e para tratar tipos complexos

```
struct ponto2d { /* ponto2d representa uma estrutura com 2 campos float */  
    float x;  
    float y;  
};  
typedef struct ponto2d TPonto2d; /*TPonto2d representa a estrutura ponto2d */
```

COMANDOS EQUIVALENTES

```
struct ponto2d { /* ponto2d representa uma estrutura com 2 campos float */
    float x;
    float y;
};
typedef struct ponto2d TPonto2d; // TPonto2d representa a estrutura ponto2d
```

```
typedef struct ponto2d { /* ponto2d representa uma estrutura com 2 campos
float */
    float x;
    float y;
} TPonto2d;
```

EXEMPLO PRÁTICO: LISTA SIMPLEMENTE ENCADEADA

DEFINIÇÃO DA LISTA

Vamos usar um exemplo onde a lista tem um campo inteiro **info**, e um ponteiro **prox** para o próximo nó da lista

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

FUNÇÕES DE MANIPULAÇÃO DA LISTA

`cria_lista`

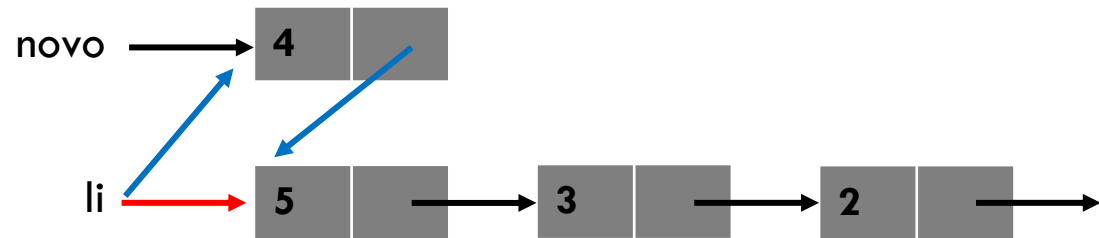
- Cria uma lista vazia, representada pelo ponteiro NULL

```
TLista* cria_lista (void) {  
    return NULL;  
}
```

FUNÇÕES DE MANIPULAÇÃO DA LISTA

insere_inicio

- Insere um elemento no início da lista
- Retorna a lista atualizada



```
TLista* insere_inicio (TLista* li, int i) {  
    TLista* novo = (TLista*) malloc(sizeof(TLista));  
    novo->info = i;  
    novo->prox = li;  
    return novo;  
}
```

FUNÇÕES DE MANIPULAÇÃO DA LISTA

imprime_lista

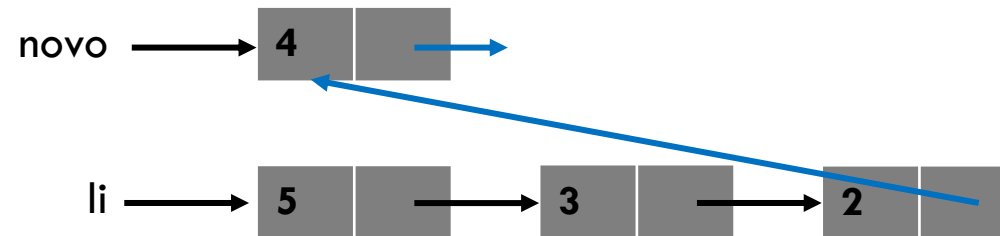
- Imprime os elementos da lista

```
void imprime_lista (TLista* li) {  
    TLista* p;  
    for (p = li; p != NULL; p = p->prox)  
        printf("info = %d\n", p->info);  
}
```


FUNÇÕES DE MANIPULAÇÃO DA LISTA

insere_fim

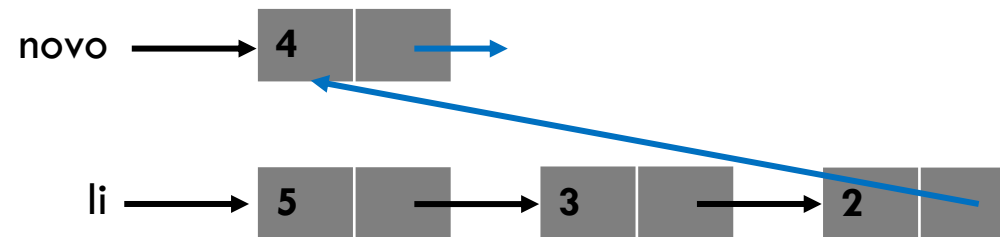
- Insere um elemento no fim da lista
- Retorna ponteiro para a lista atualizada



FUNÇÕES DE MANIPULAÇÃO DA LISTA

insere_fim

- Insere um elemento no fim da lista
- Retorna ponteiro para a lista atualizada

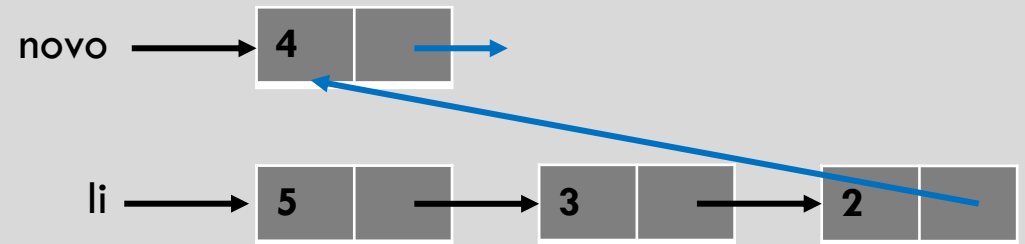


Implementem a função `insere_fim`:
`TLista* insere_fim (TLista* li, int i)`

```

TLista* insere_fim (TLista* li, int i) {
    TLista* novo = (TLista*) malloc(sizeof(TLista));
    novo->info = i;
    novo->prox = NULL;
    TLista* p = li;
    if (p == NULL) { //se a lista estiver vazia
        li = novo;
    }
    else {
        while (p->prox != NULL) { //encontra o ultimo elemento
            p = p->prox;
        }
        p->prox = novo;
    }
    return li;
}

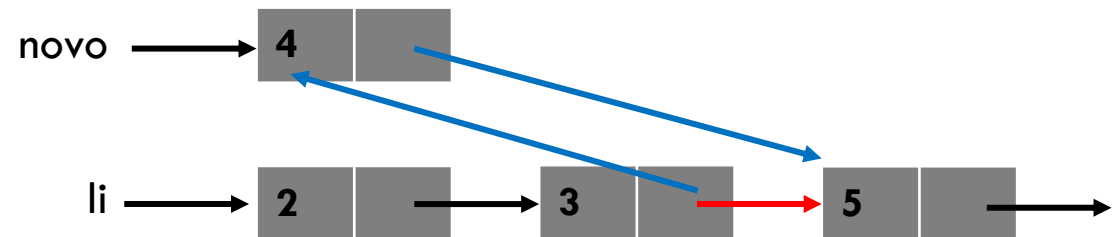
```



FUNÇÕES DE MANIPULAÇÃO DA LISTA

insere_ordenado

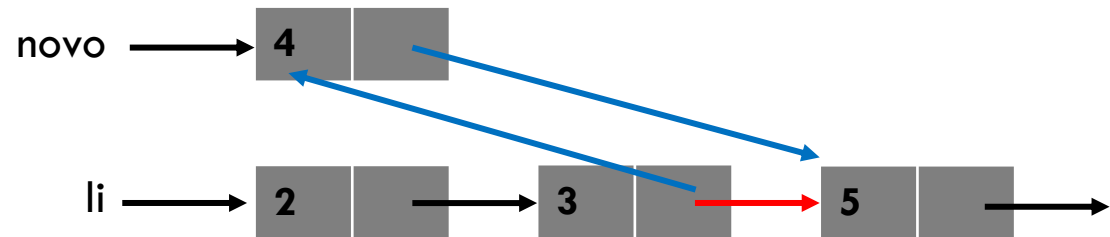
- Insere um elemento na lista de forma que ela esteja sempre ordenada
- Retorna ponteiro para a lista atualizada



FUNÇÕES DE MANIPULAÇÃO DA LISTA

insere_ordenado

- Insere um elemento na lista de forma que ela esteja sempre ordenada
- Retorna ponteiro para a lista atualizada

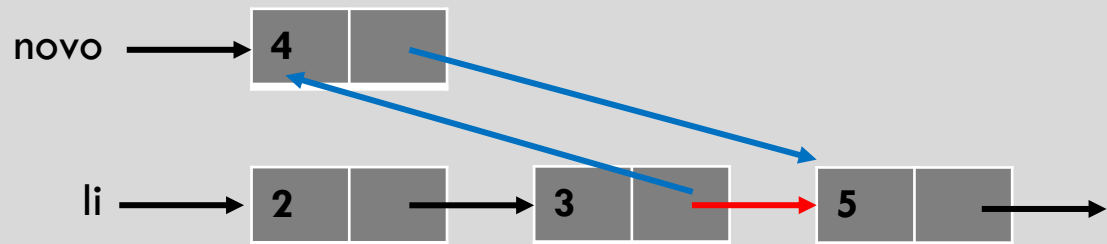


Implementem a função `insere_ordenado`:
`TLista* insere_ordenado (TLista* li, int i)`

```

TLista* insere_ordenado (TLista* li, int i) {
    TLista* novo;
    /* cria novo elemento */
    novo = (TLista*) malloc(sizeof(TLista));
    novo->info = i;
    TLista* p = li; /* ponteiro para percorrer a lista */
    if (p == NULL) { /* lista vazia -- insere no início */
        novo->prox = NULL;
        li = novo;
    }
    else if (p->info > i) { /* insere no início da lista */
        novo->prox = li;
        li = novo;
    }
    else { /* procura posição de inserção */
        while (p->prox != NULL && p->prox->info < i) {
            p = p->prox;
        }
        novo->prox = p->prox;
        p->prox = novo;
    }
    return li;
}

```



FUNÇÕES RECURSIVAS

FUNÇÕES RECURSIVAS

Quando uma **função chama a si mesma**, dizemos que a função é recursiva

Pontos importantes:

- Forma fácil de dividir para conquistar – trata-se um item e chama-se recursivamente para os demais
- É preciso ter um critério de parada para evitar loop infinito de chamadas

Exercícios:

- Função recursiva para inserção no final da lista
 - `TLista* insere_fim_recursivo (TLista* li, int i)`
- Função recursiva para inserção ordenada
 - `TLista* insere_ordenado_recursivo(TLista *li, int i)`


```
TLista* insere_fim_recursivo (TLista* li, int i) {  
    if (li == NULL || li->prox == NULL) {  
        TLista *novo = (TLista *) malloc(sizeof(TLista));  
        novo->info = i;  
        novo->prox = NULL;  
        if (li == NULL) {  
            li = novo;  
        } else li->prox = novo;  
    }  
    else insere_fim_recursivo(li->prox, i);  
    return li;  
}
```

```

TLista *insere_ordenado_recurativo(TLista *li, int i) {
    if (li == NULL || li->prox == NULL) { //lista vazia
        TLista *novo = (TLista *) malloc(sizeof(TLista));
        novo->info = i;
        novo->prox = NULL;
        if (li == NULL)
            li = novo;
        else { /* inserir no meio da lista */
            if (li->info > i) {
                novo->prox = li;
                li = novo;
            }
            else li->prox = novo;
        }
    } else {
        if (li->info > i) {
            TLista *novo = (TLista *) malloc(sizeof(TLista));
            novo->info = i;
            novo->prox = li;
            li = novo;
        }
        else li->prox = insere_ordenado_recurativo(li->prox, i);
    }
    return li;
}

```

EXERCÍCIOS

Função para excluir um elemento da lista:

- **TLista* exclui(TLista* li, int i)**

Função para alterar o valor de um elemento da lista:

- **TLista* altera(TLista* li, int vantigo, int vnov)**

AGRADECIMENTOS

Material baseado nos slides de Waldemar Celes e Marcelo Gattass

Departamento de Informática, PUC-Rio

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