

Estruturas de Dados e Algoritmos - T2

Estruturas lineares – Listas.

Rui Camacho
(slides por Luís Teixeira)
LEEC 2021/2022

LISTA

É uma **sequência de elementos**, geralmente do mesmo tipo: L_1, L_2, \dots, L_N

Uma **lista vazia** é uma lista com zero elementos

Operações comuns:

- ▢ criar uma lista vazia
- ▢ adicionar/remover um elemento a uma lista
- ▢ determinar a posição de um elemento na lista
- ▢ determinar o comprimento (nº de elementos) de uma lista
- ▢ concatenar duas listas

TÉCNICAS DE IMPLEMENTAÇÃO DE LISTAS

Baseada em vetores (*arrays*) dinâmicos

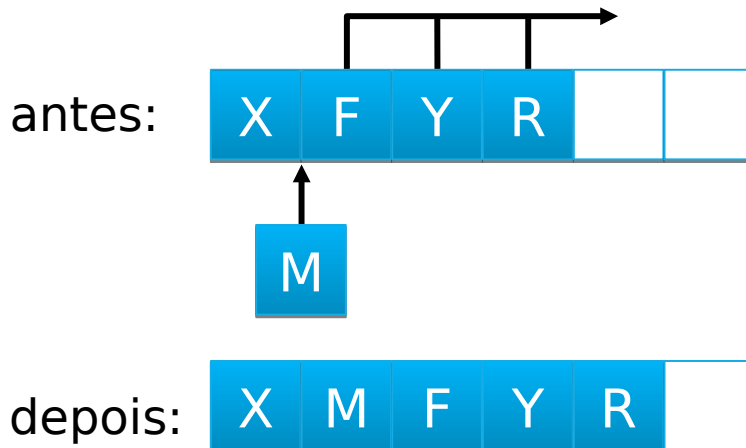
Baseada em apontadores:

- ▢ Listas ligadas
- ▢ Listas circulares
- ▢ Listas duplamente ligadas

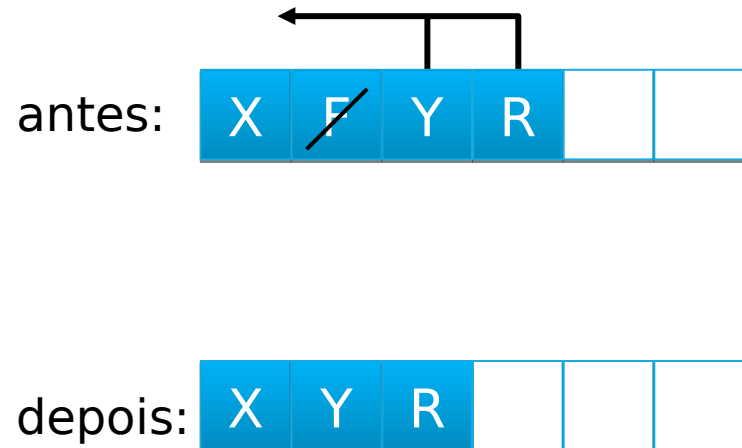
IMPLEMENTAÇÃO BASEADA EM VETORES

Os elementos da lista são guardados num **vetor dinâmico**. O vetor é uma estrutura de dados com alocação dinâmica de memória para armazenar N elementos de um dado tipo. O tamanho do vetor exige monitorização constante.

Inserção de M:

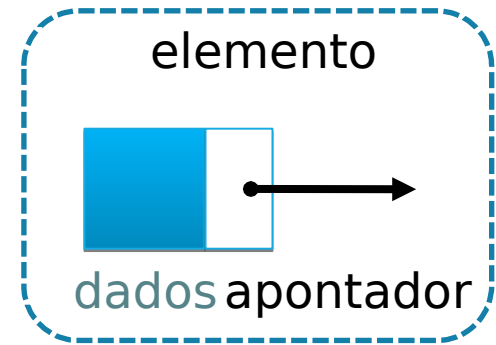


Remoção de F:



LISTA LIGADA

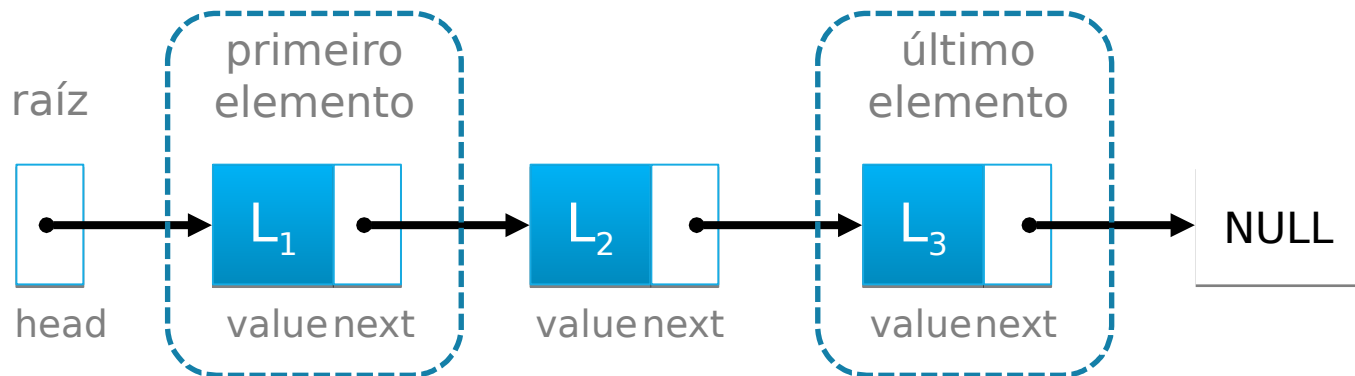
É uma estrutura de dados linear que consiste numa sequência de elementos, em que cada **elemento** inclui um (ou mais) campo(s) com **dados** e um **apontador**. O tamanho da lista é facilmente alterado por alocação dinâmica.



- Os **dados** podem ser de **qualquer tipo** e, em geral, os elementos da lista ligada são todos do mesmo tipo de dados.
- Cada elemento inclui um **apontador** para o endereço de memória do **próximo elemento** da lista.
- O apontador do **último elemento** da lista aponta para **NULL**.
- A lista é acessível através de um apontador “externo”, ou **raiz**, que contém o endereço do primeiro elemento da lista.

LISTA LIGADA

```
struct listItem {  
    data_type value;           /* valor do elemento da lista */  
    struct listItem *next;    /* apontador para o próximo elemento */  
};  
  
struct listItem* head = NULL; /* apontador para o início da lista (raíz) */
```



VECTOR DINÂMICO VS. LISTA LIGADA

Implementação baseada em Vetores

Para inserir ou eliminar um elemento no vector poderá ser necessário (desvantagens):

- **mover outros elementos** para posições posteriores ou anteriores (no pior caso, todos os elementos!)
- **re-alocar memória**, quando a capacidade actual do vector é excedida; ou diminuir o tamanho do vector, quando se eliminam muitos elementos, evitando ocupação desnecessária de memória

Pesquisa, inserção e remoção de elementos:

- operações de complexidade temporal **$O(\text{tamanho})$ ***
- inserção pode ter complexidade espacial **$O(\text{tamanho})$ ***

* Indicam tendência de crescimento
Mais informações nas próximas aulas

VECTOR DINÂMICO VS. LISTA LIGADA

Implementação baseada em Listas Ligadas

Não é possível aceder a posições aleatórias da lista através do uso de índices.

Os elementos requerem **mais espaço de memória** do que na implementação baseada em vectores porque, para além dos dados, é necessário armazenar a referência do próximo elemento.

Inserção e remoção de elementos:

- operações de complexidade temporal constante **$O(1)$**
(nota: se a posição já estiver especificada)

Pesquisa de elementos:

- operação de complexidade temporal **$O(\text{tamanho})$**

LISTA LIGADA (EXEMPLO DE IMPLEMENTAÇÃO)

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

struct listItem {
    int value;
    struct listItem * next;
};

typedef struct listItem element;

/* cria um apontador para um novo elemento do
tipo listItem com o valor inteiro val */
element * newItem(int val);

/* percorre e imprime os dados armazenados na
lista, começando no elemento apontado por item
*/

void printList(element * item);
```

```
int main() {

    element *curr, *head = NULL;
    int i;

    head = newItem(0);
    curr = head;
    for(i=1 ; i<10; i++) {
        curr->next = newItem(i);
        curr = curr->next;
    }
    curr->next = NULL;

    printList(head);

    /* libertar memória! */
}
```

LISTA LIGADA (EXEMPLO DE IMPLEMENTAÇÃO)

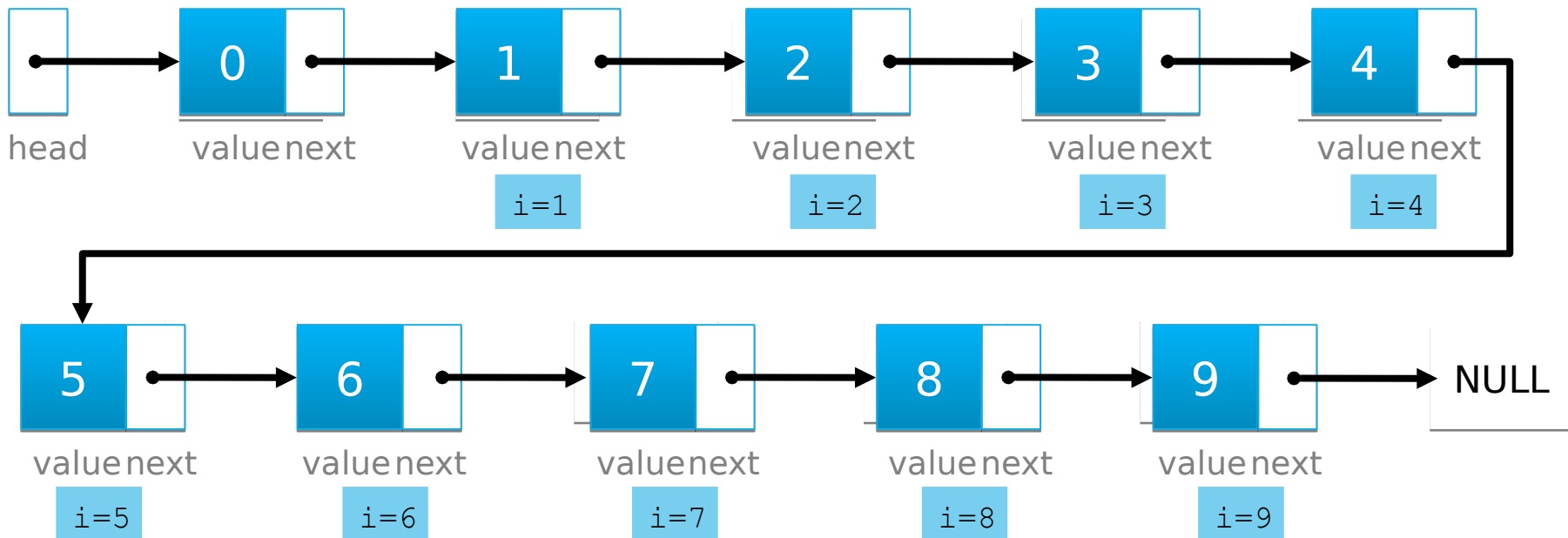
```
head = newItem(0);
```

```
curr = head;
```

} Cria e insere o primeiro elemento da lista

```
for (i=1 ; i<10; i++) {  
    curr->next = newItem(i);  
    curr = curr->next;  
}
```

} Cria e insere novos elementos no fim da lista



LISTA LIGADA (EXEMPLO DE IMPLEMENTAÇÃO)

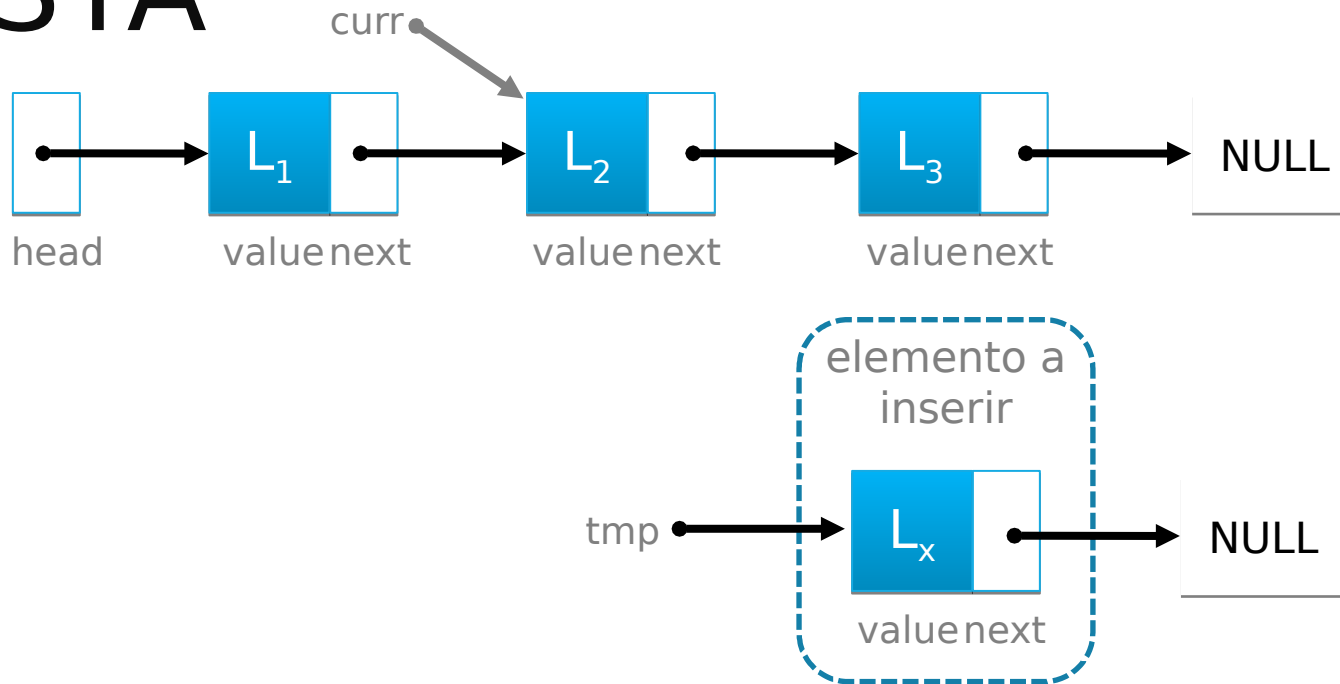
```
element * newItem(int val) {
    element *item = (element *) malloc(sizeof(element));
    item->value = val;
    item->next = NULL;
    return item;
};

void printList(element * item) {

    if (item == NULL) {
        printf("Lista vazia!\n"); return;
    }

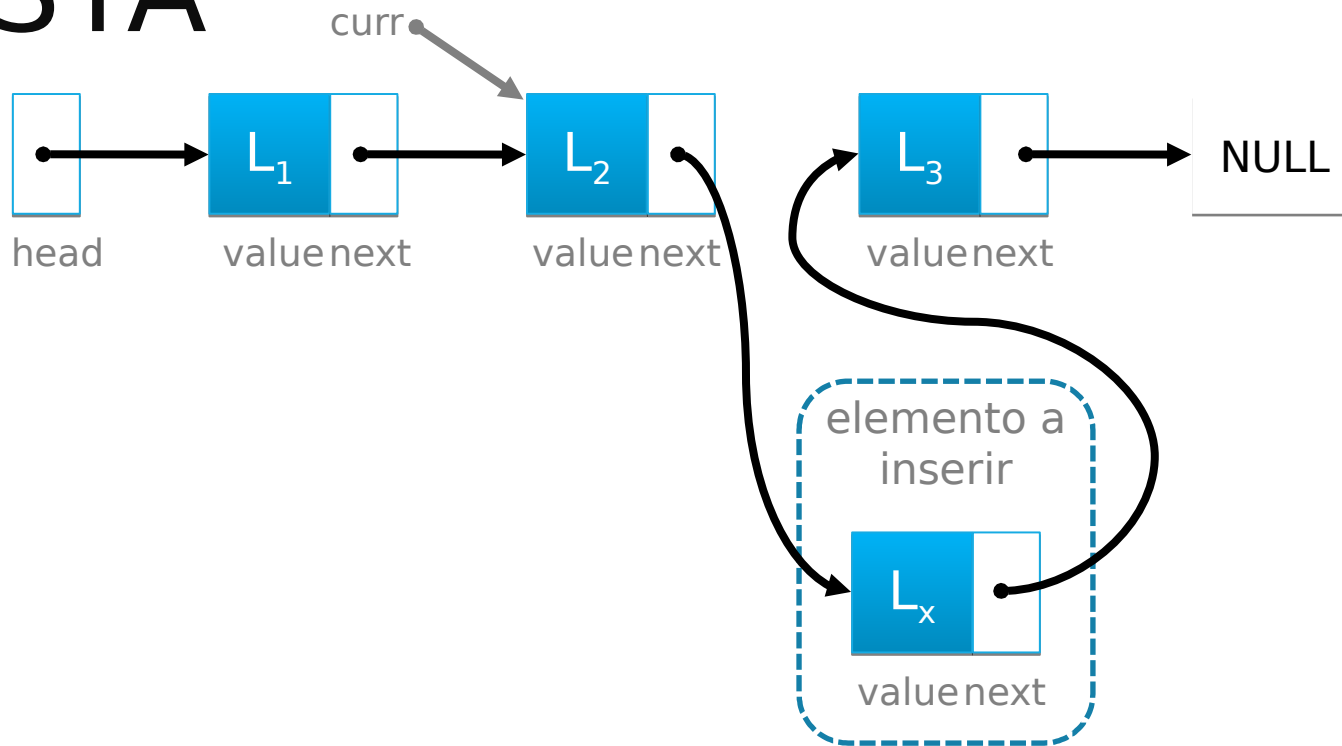
    printf("Lista:");
    while(item != NULL) { /* percorre todos os elementos da lista */
        printf(" %d", item->value);
        item = item->next ;
    }
    printf("\n");
}
```

INSERIR ELEMENTOS NA LISTA



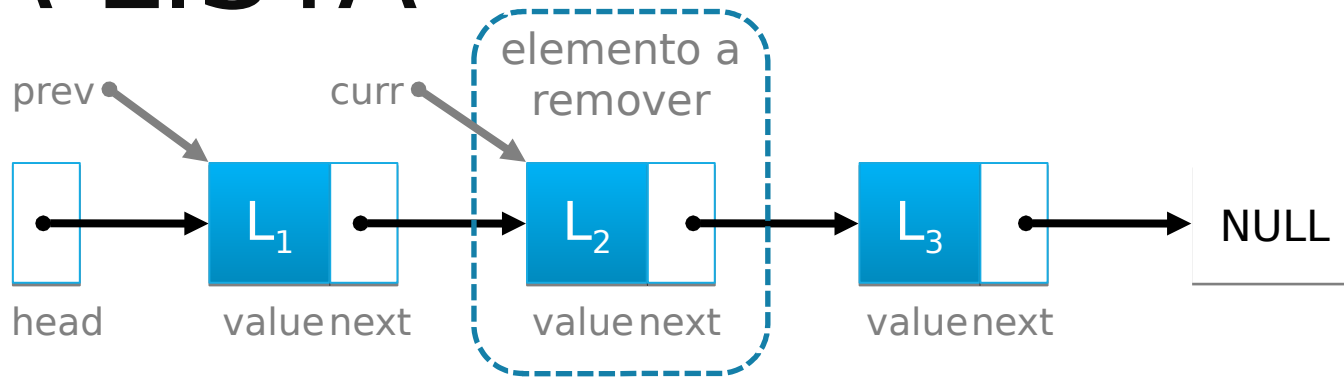
Para inserir um novo elemento na lista, referenciado por um apontador temporário `tmp`, é necessário **identificar a posição onde esse elemento irá ficar** e obter um apontador para o elemento **seguinte**, i.e. `curr->next`.

INSERIR ELEMENTOS NA LISTA



A inserção efetua-se atualizando os apontadores $tmp \rightarrow next = curr \rightarrow next$ e $curr \rightarrow next = tmp$.

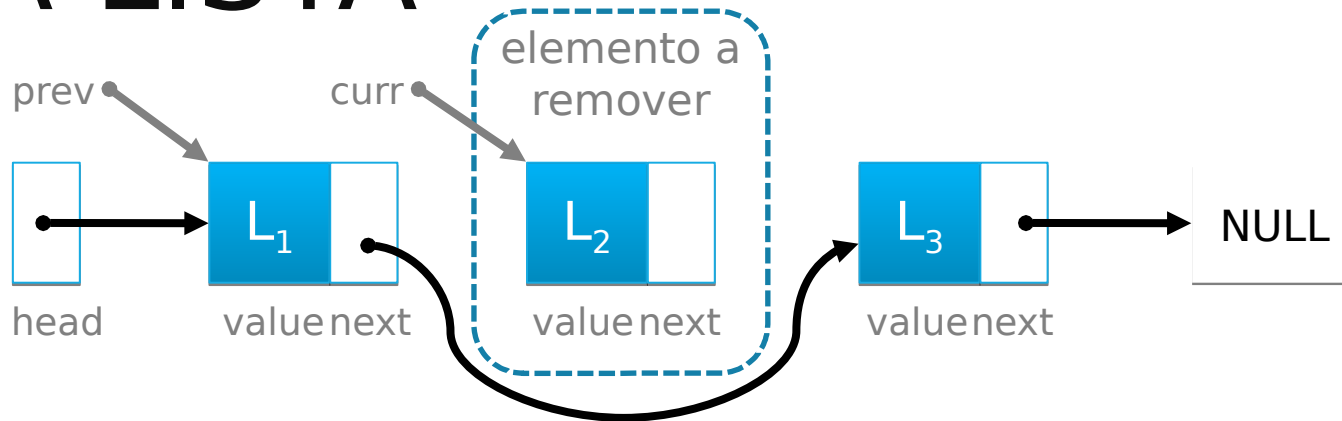
REMOVER ELEMENTOS DA LISTA



Para remover um elemento da lista é necessário **identificar a posição do elemento que se deseja remover** e ter acesso ao elemento **anterior** da lista.

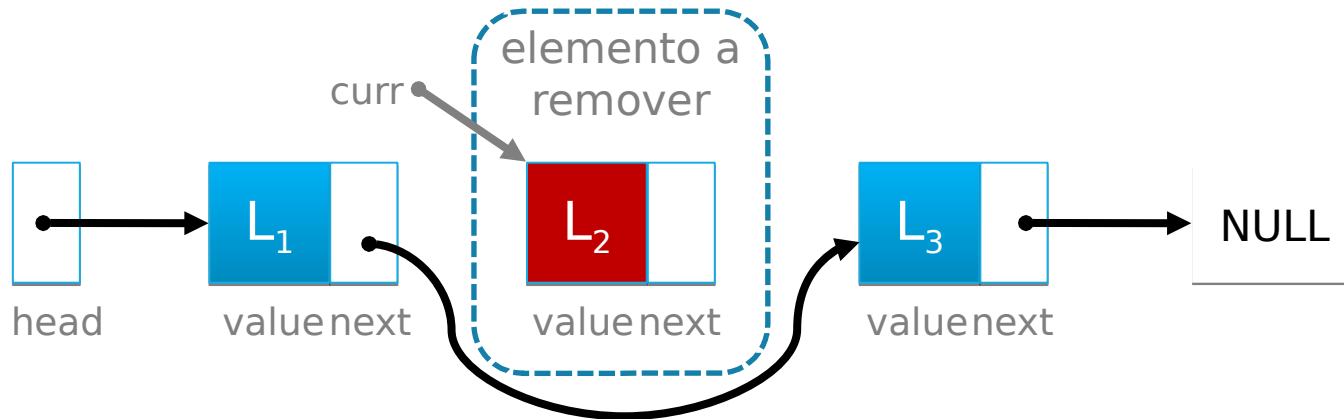
Sendo `prev` um apontador para o elemento anterior e `curr` um apontador para o elemento atual, compara-se o valor `curr->value` ao valor a remover e, enquanto for diferente, atualiza-se `prev = curr` e `curr = curr->next`.

REMOVER ELEMENTOS DA LISTA



Depois de identificar o elemento a remover, é necessário atualizar o apontador $prev \rightarrow next = curr \rightarrow next$ e eliminar o apontador $curr$ da memória.

FUGAS DE MEMÓRIA



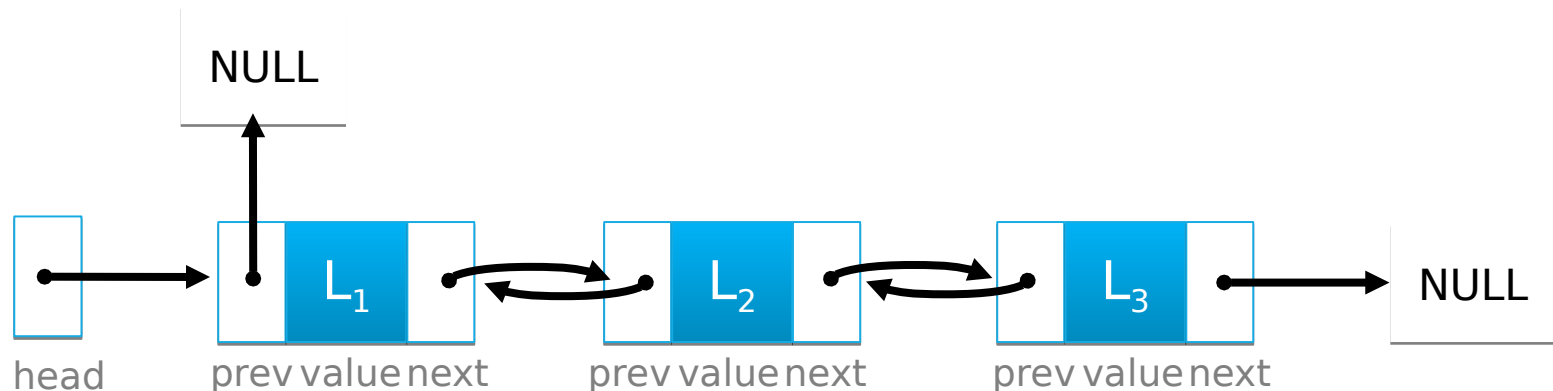
O que acontecerá ao elemento removido, quando não se liberta ou se atribui um novo endereço ao apontador do elemento (curr)?

O elemento fica permanentemente inacessível e não será possível libertar o espaço de memória que tinha sido alocado para o armazenar. Esta “perda” de elementos origina **fugas de memória** (***memory leaks***). Quando as fugas são significativas pode ocorrer um *crash* do sistema!

LISTA DUPLAMENTE LIGADA

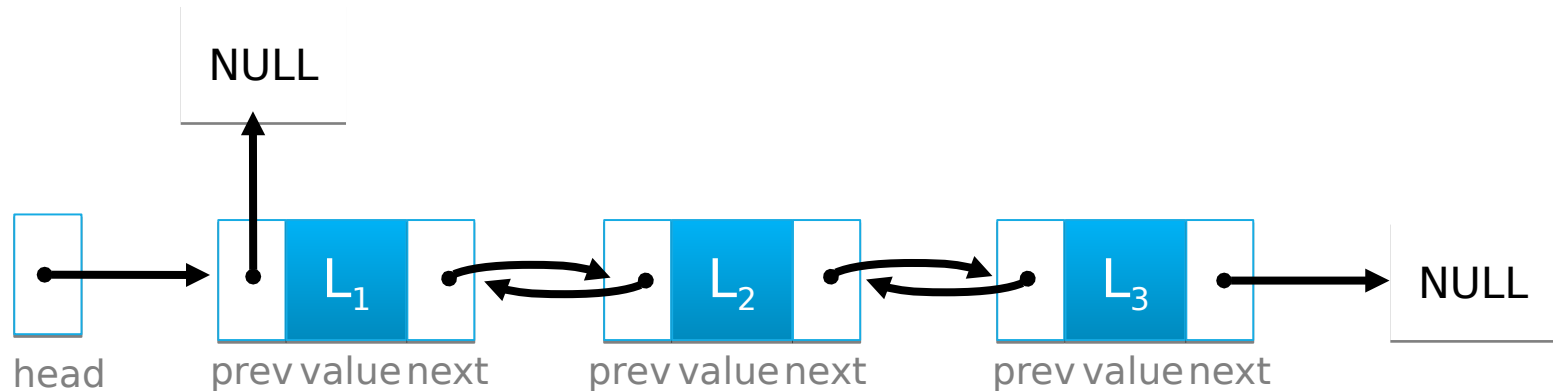
Uma **lista duplamente ligada** é um tipo especial de lista, em que cada elemento inclui dois apontadores, um para o elemento anterior e outro para o elemento seguinte da lista.

Estas listas são úteis em aplicações em que há necessidade de percorrer a lista em ambos os sentidos.



LISTA DUPLAMENTE LIGADA

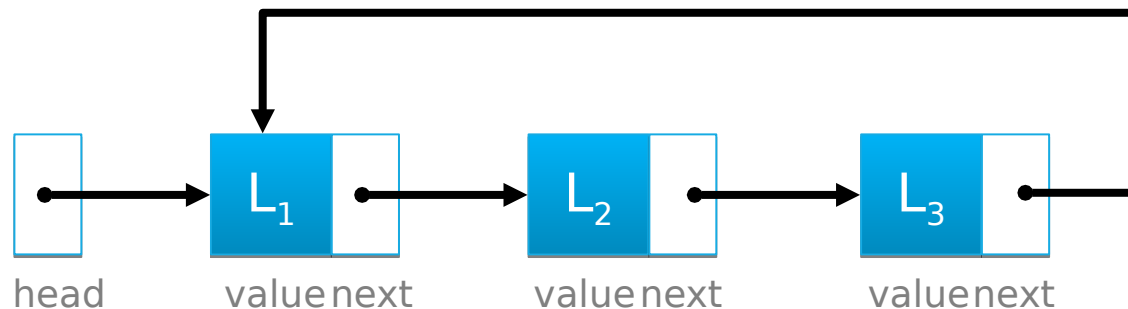
```
struct listItem {  
    data_type value; /* valor do elemento da lista */  
    struct listItem *prev; /* apontador para o elemento anterior */  
    struct listItem *next; /* apontador para o próximo elemento */  
};  
  
struct listItem* head = NULL; /* apontador para o início da lista (raiz) */  
/* pode também definir-se um apontador para  
o fim da lista */
```



LISTAS CIRCULARES

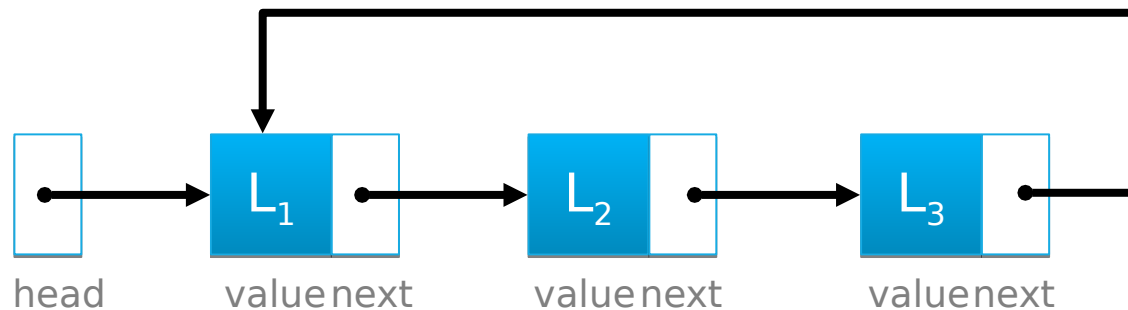
Uma **lista circular** é um tipo especial de lista ligada, em que o último elemento referencia o primeiro elemento da lista.

Estas listas são úteis em aplicações em que há necessidade de percorrer a lista em modo “loop”, uma vez que ligam o fim ao início da lista.



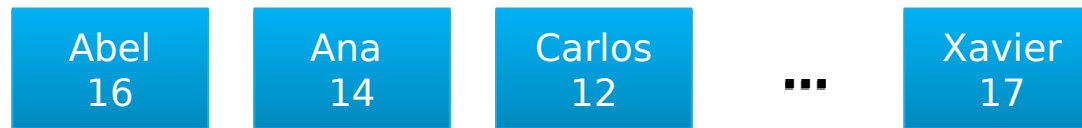
LISTAS CIRCULARES

```
struct listItem {  
    data_type value;      /* valor do elemento da lista */  
    struct listItem *next; /* apontador para o próximo elemento */  
};  
  
struct listItem* head = NULL; /* apontador para o início da lista (raiz) */  
/* pode também definir-se um apontador para o fim da lista */
```



EXEMPLO DE UMA LISTA

Lista dos alunos inscritos a Estrutura de Dados e Algoritmos



A lista guarda informação sobre o nome e nota dos alunos

Operações:

- ▢ adicionar um aluno à lista
- ▢ remover alunos da lista
- ▢ listar todos os alunos e respectivas notas
- ▢ pesquisar um aluno pelo nome
- ▢ ordenar a lista por ordem alfabética (ou nota)
- ▢ etc.

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define NCAR 100

typedef struct _aluno {
    char nome[NCAR];
    int nota;
    struct _aluno *proximo;
} aluno;

aluno *primeiro = NULL; /* = 0 */

void ler_str(char *s, int n) { /* ler um string */
    fgets(s, n, stdin);
    s[strlen(s)-1] = '\\0';
}

void ler_aluno(aluno *a) { /* ler um registo de aluno */
    printf("Nome: "); ler_str(a->nome, NCAR);
    printf("Nota: "); scanf("%d", &a->nota);
}

void escrever_aluno(aluno *a) { /* escrever um registo de aluno */
    printf("Nome: %s\\n", a->nome);
    printf("Nota: %2d\\n", a->nota);
}

```

```

aluno* novo_aluno(void) {
    aluno *p;
    p = (aluno*)malloc(sizeof(aluno));
    p->proximo = NULL;
    return p;
}

void insere() { /* insere um novo registo no inicio da lista */
    aluno *p;

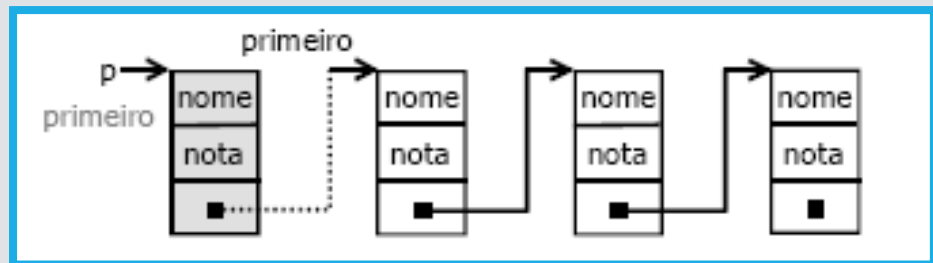
    p = novo_aluno();

    ler_aluno(p);

    if(primeiro == NULL)
        primeiro = p;
    else
    {
        p->proximo = primeiro;
        primeiro = p;
    }
}

```

Inserção no início da lista



```

aluno* busca() { /* efectua a busca de um nome na lista */
    aluno *p;
    char s[NCAR];
    printf("Nome a procurar? ");
    ler_str(s, NCAR);
    p = primeiro;
    while(p != NULL && strcmp(p->nome, s) != 0)
        p = p->proximo;
    if(p == NULL) printf("Nao foi encontrado\n");
    else escrever_aluno(p);
    return p;
}

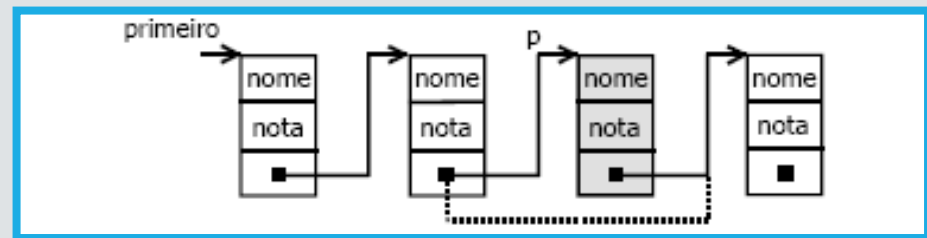
```

```

void elimina() { /* elimina um registo da lista ligada */
    aluno *p, *pa; char resp;
    p = busca();
    if(p == NULL) return;
    printf("Deseja remover (s,n)? ");
    scanf(" %c", &resp);
    if(resp == 's') {
        if(p == primeiro) {
            primeiro = p->proximo;
            free(p);
        } else {
            pa = primeiro;
            while(pa->proximo != p) pa=pa->proximo;
            pa->proximo = p->proximo;
            free(p);
        }
    }
}

```

Remoção de um elemento (a cinzento) de uma lista




```

void lista() {
    aluno *p;
    p = primeiro;
    while(p != NULL) {
        escrever_aluno(p);
        p=p->proximo;
    }
}

void menu() {
    char op;
    printf("Operacoes: i(nserte), e(limina), b(usca), l(ista), s(ai)\n");
    printf(">> Operacao desejada? ");
    scanf(" %c", &op);
    switch(op) {
        case 'i': insere(); break;
        case 'e': elimina(); break;
        case 'b': busca(); break;
        case 'l': lista(); break;
        case 's': exit(0);
        default: printf("Operacao nao definida\n");
    }
}

int main() {
    while(1) menu();
}

```