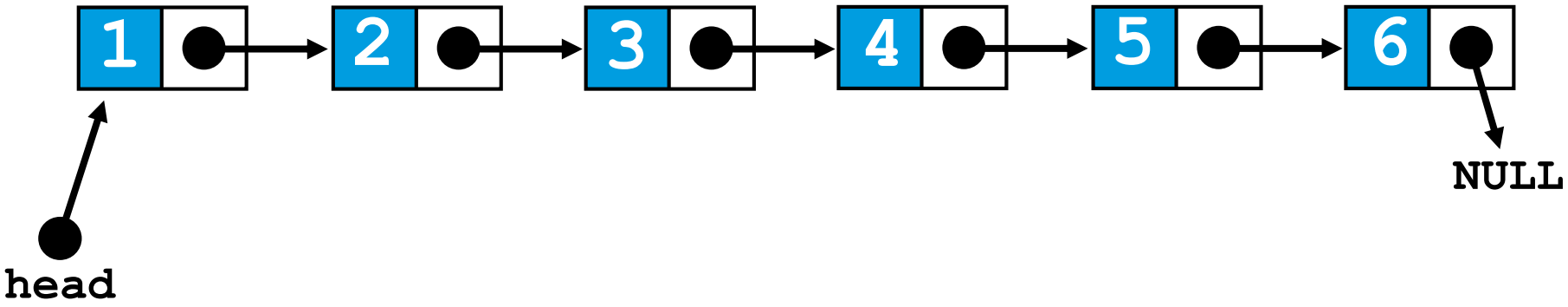
DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

Estruturas auto-referenciadas & Listas



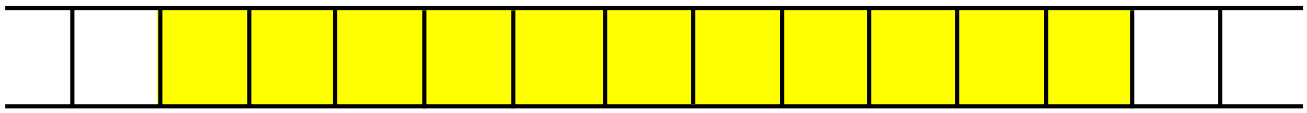
IAED

Tabelas/Vectores

- Colecção de items
  - Inteiros, reais, caracteres
  - Estruturas
  - Tabelas, Ponteiros

- Guardados em posições consecutivas de memória

```
int tab[N];
```



- Programador é responsável por respeitar limites

Tabelas/Vectores

- Em C tabelas podem ser
  - De dimensão fixa
  - Alocadas dinamicamente

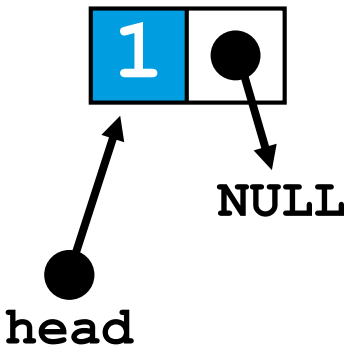
```
#define N 100
int tab1[N];
int *tab2 = (int *) malloc(N*sizeof(int));
```

- Acesso alternativo a tabelas

```
x = tab2[i];
y = *(tab2+i);
```

DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

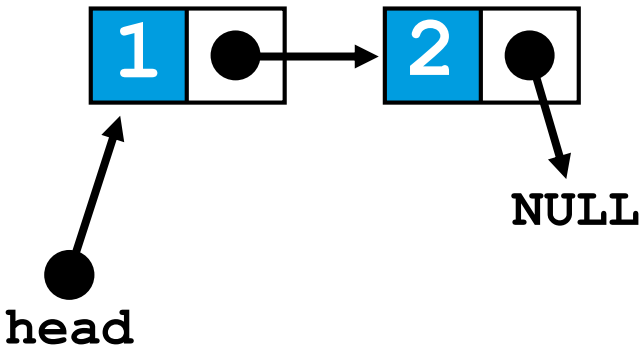
Estruturas auto-referenciadas & Listas



IAED

DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

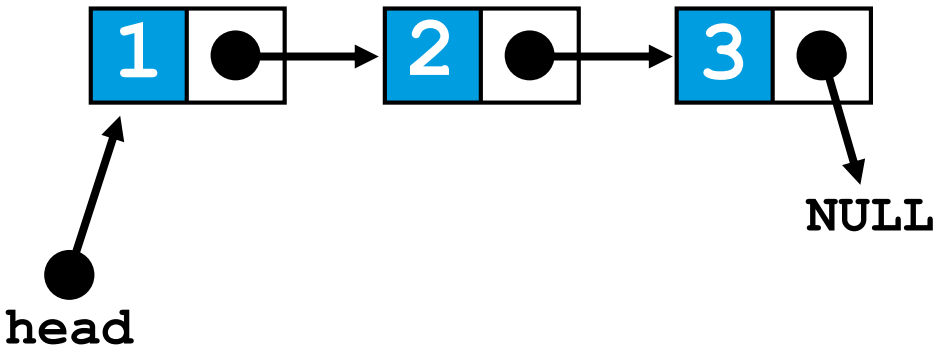
Estruturas auto-referenciadas & Listas



IAED

DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

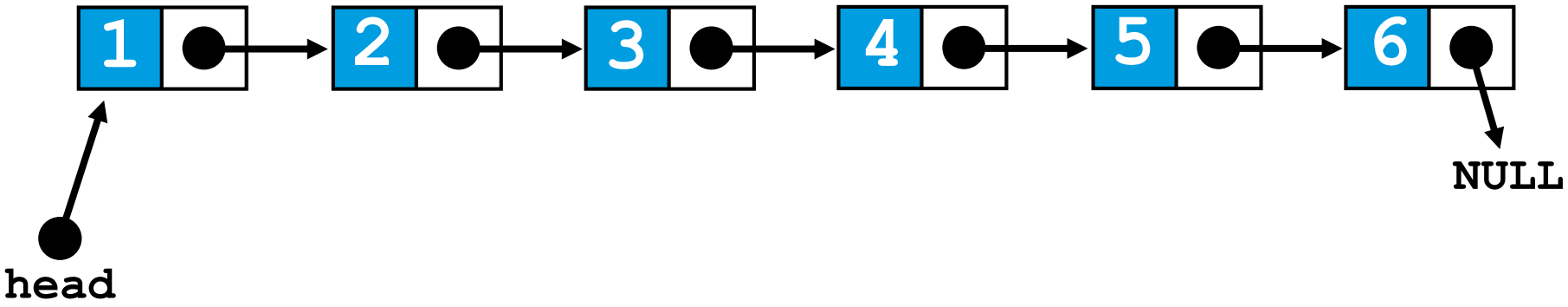
Estruturas auto-referenciadas & Listas



IAED

DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

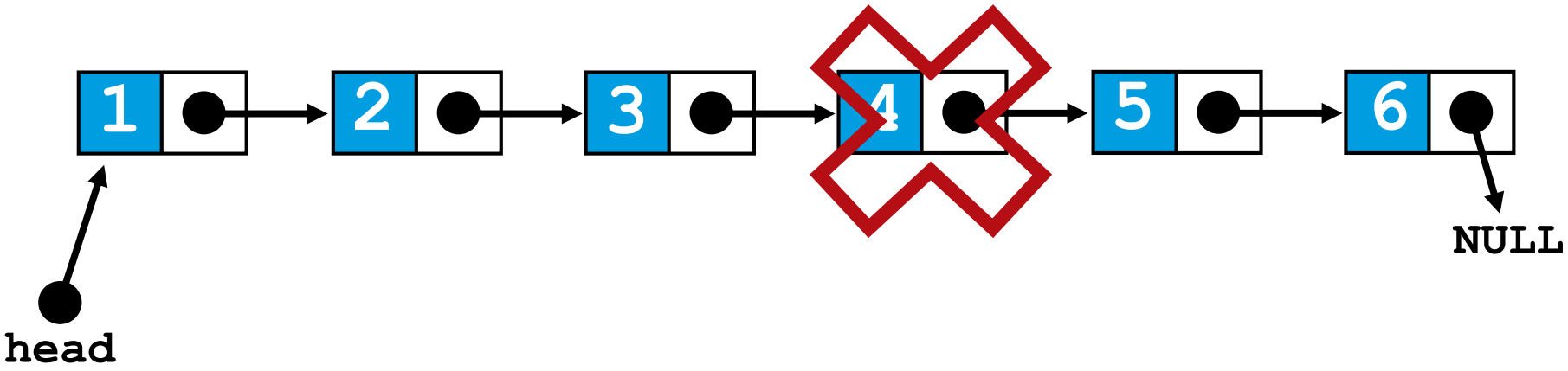
Estruturas auto-referenciadas & Listas



IAED

DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

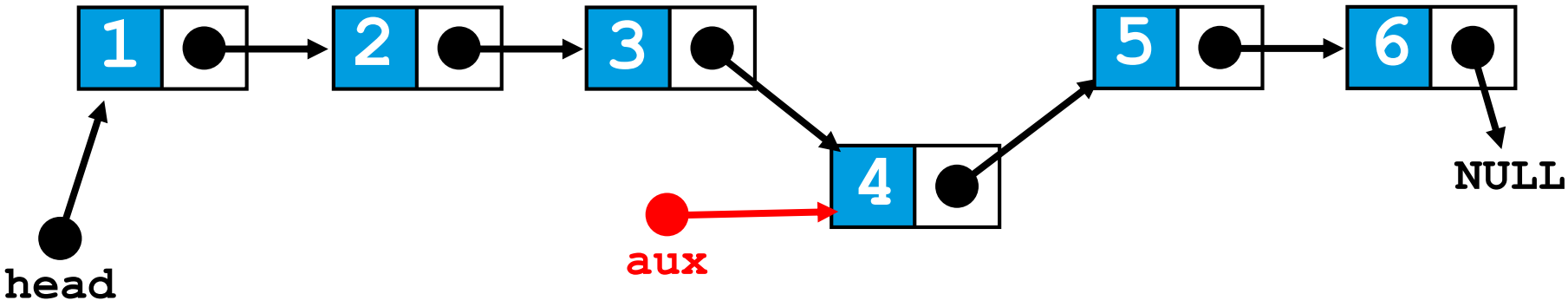
Estruturas auto-referenciadas & Listas



IAED

DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

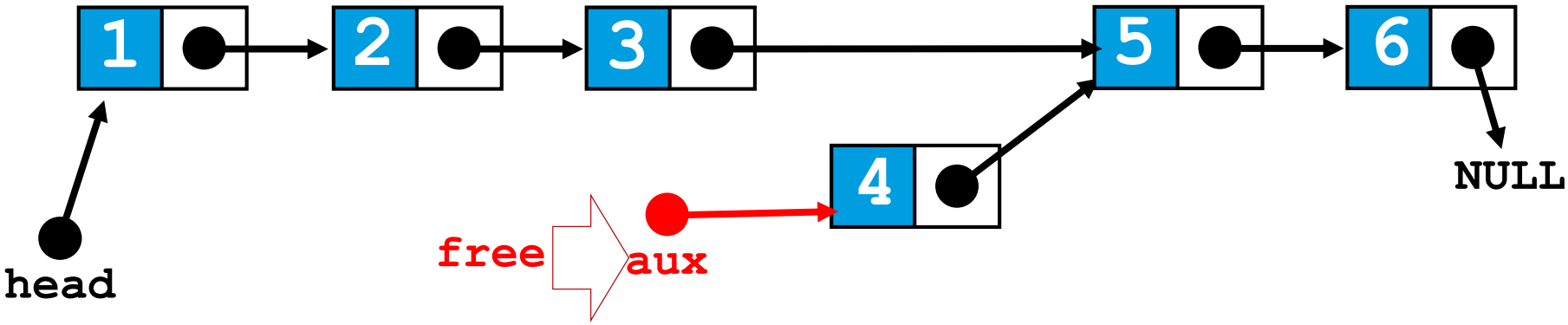
Estruturas auto-referenciadas & Listas



IAED

DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

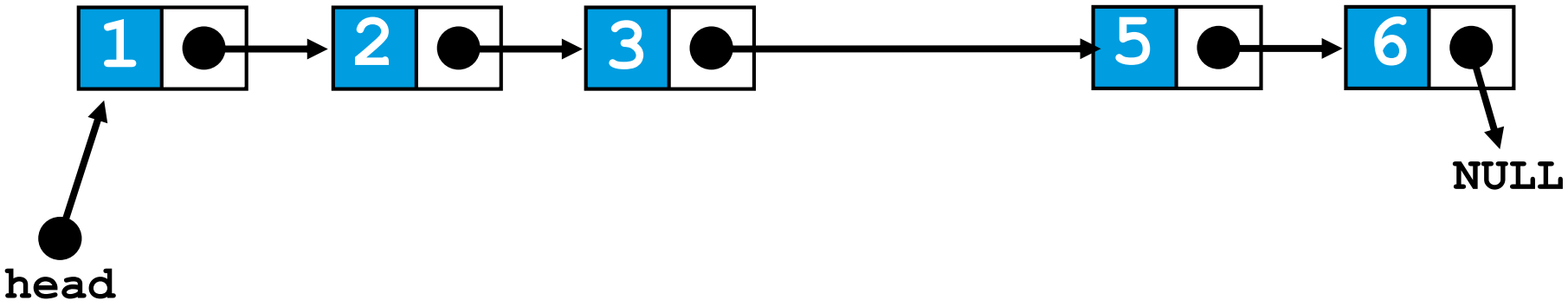
Estruturas auto-referenciadas & Listas



IAED

DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

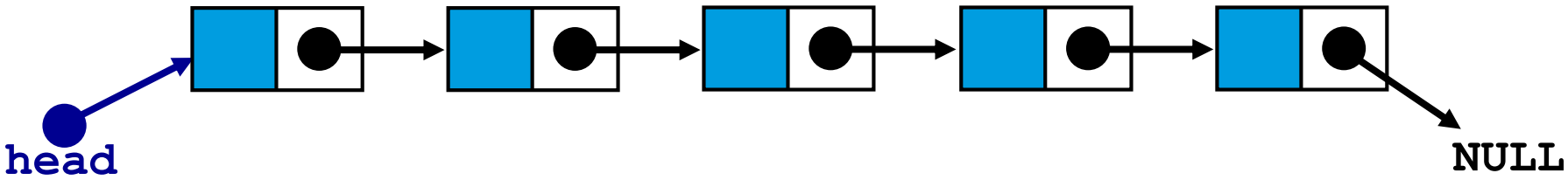
Estruturas auto-referenciadas & Listas



IAED

Tabelas/Vectores

- Vantagens
  - Manipulação simples
  - Facilidade de acesso ao n-ésimo elemento
- Desvantagens
  - Tamanho limitado
  - Necessidade de realocar e copiar todos os elementos se desejar aumentar dimensão da tabela
  - Desperdicio de memória
- Alternativa: ***Listas***



O que é uma estrutura auto-referenciada?

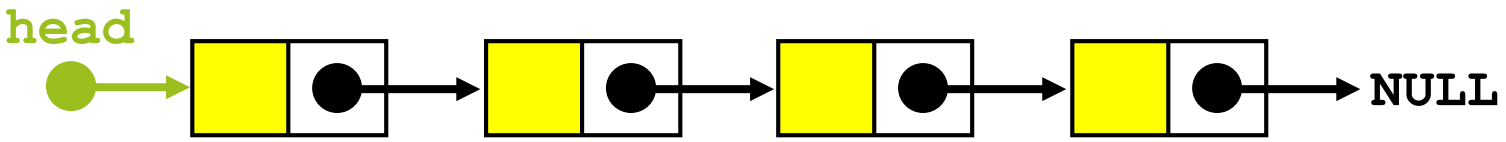
- Estrutura em que um campo da estrutura é um apontador para outra estrutura do mesmo tipo

```
typedef struct ponto {
  double x;
  double y;
  struct ponto *proximo;
} Ponto;
```

- As estruturas auto-referenciadas permitem criar estruturas de dados dinâmicas, utilizando ponteiros:
  - Listas (simplesmente e duplamente ligadas)
  - Árvores
  - etc.

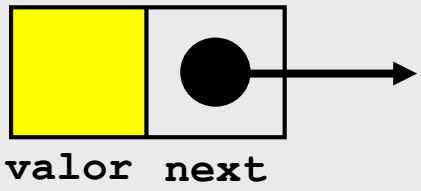
Lista Simplesmente Ligada

- Conjunto de nós



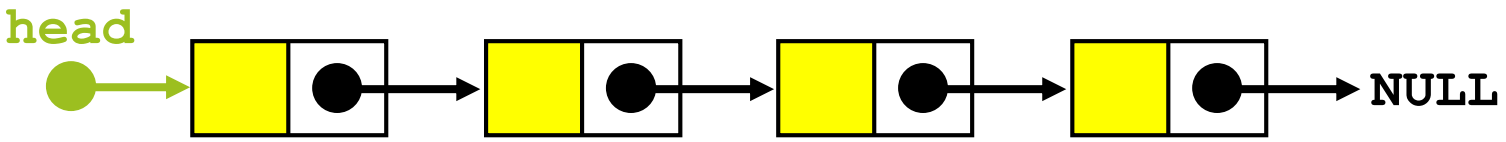
- Cada nó contém
  - Informação útil
  - Ponteiro para o próximo nó

```
struct node {
  int valor;
  struct node *next;
};
```



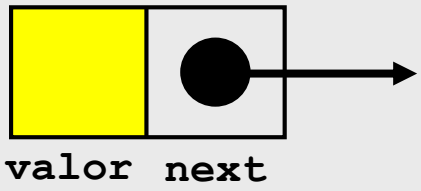
Lista Simplesmente Ligada

- Conjunto de nós



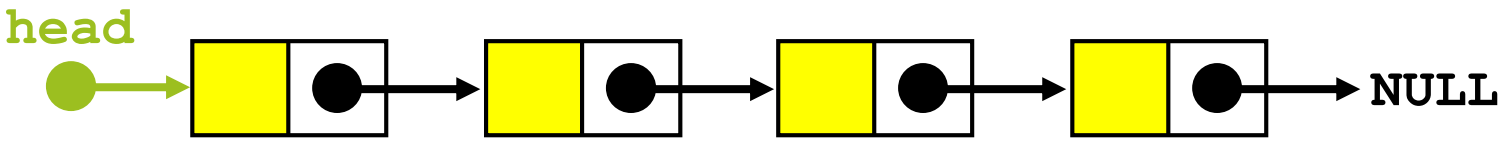
- Cada nó contém
  - Informação útil
  - Ponteiro para o próximo nó

```
struct node {
  int valor;
  struct node *next;
};
```



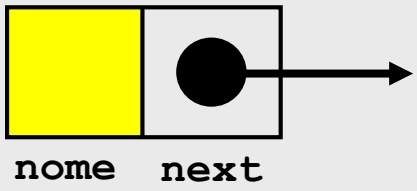
Lista Simplesmente Ligada

- Conjunto de nós



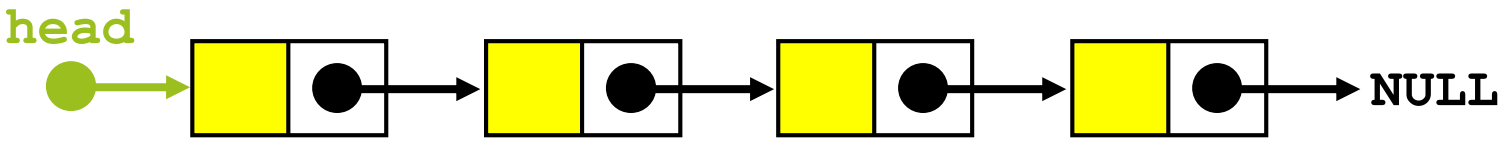
- Cada nó contém
  - Informação útil
  - Ponteiro para o próximo nó

```
struct node {
  char nome[256];
  struct node *next;
};
```

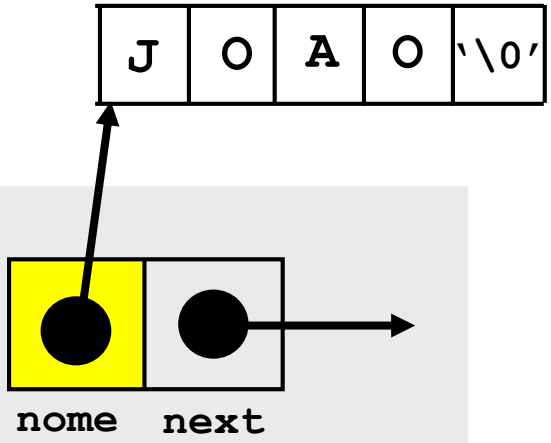


Lista Simplesmente Ligada

- Conjunto de nós



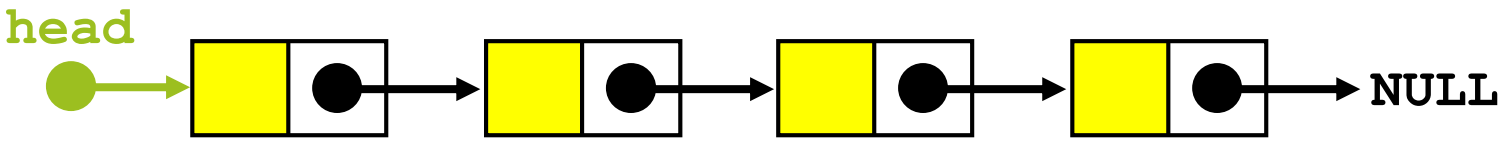
- Cada nó contém
  - Informação útil
  - Ponteiro para o próximo nó




```
struct node {
  char *nome;
  struct node *next;
};
```

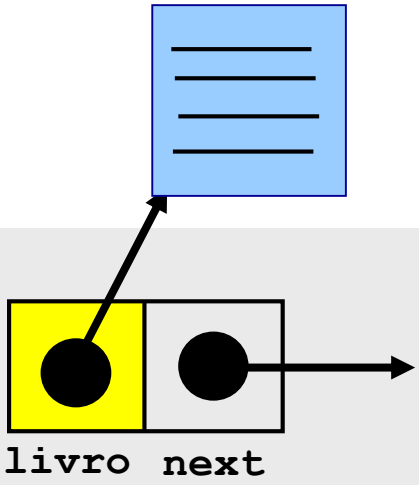
Lista Simplesmente Ligada

- Conjunto de nós



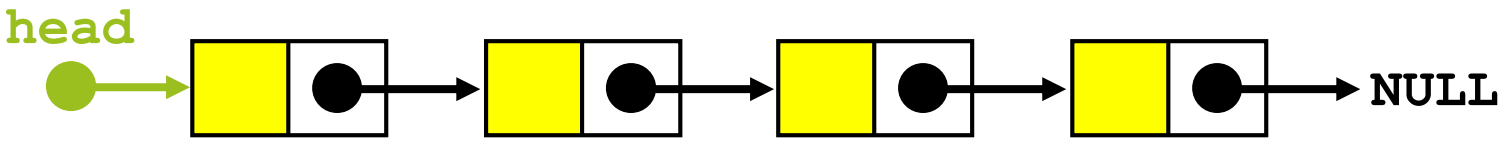
- Cada nó contém
  - Informação útil
  - Ponteiro para o próximo nó

```
struct node {
  Livro *livro;
  struct node *next;
};
```



Lista Simplesmente Ligada

- Conjunto de nós

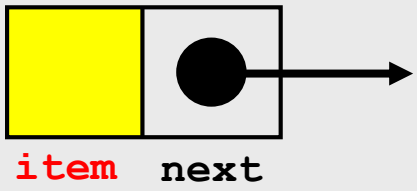


- Cada nó contém
  - Informação útil
  - Ponteiro para o próximo nó

A nossa abstração habitual

```
struct node {
  Item item;
  struct node *next;
};
```

O Item pode ser definido como um ponteiro



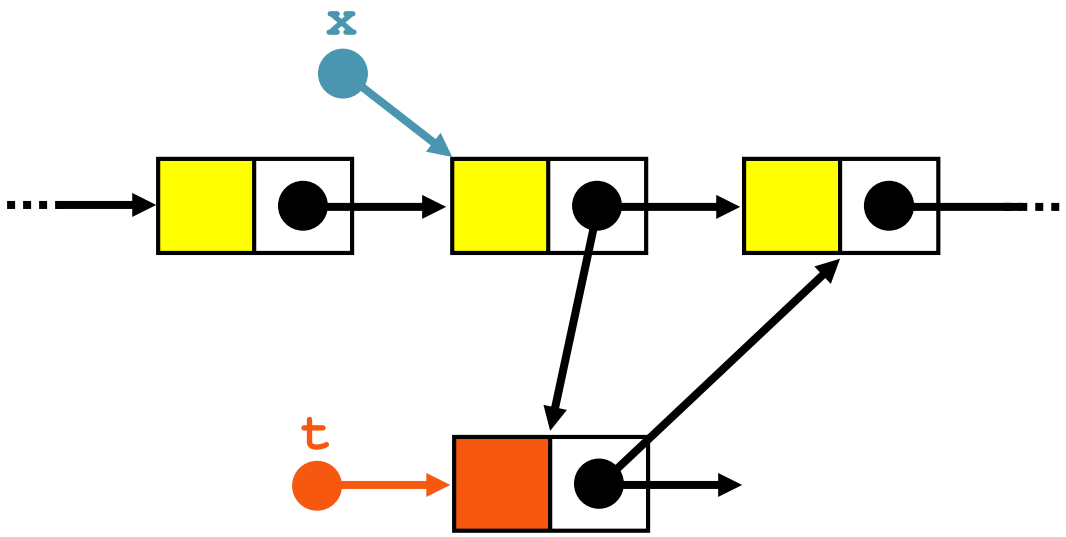
Listas

- Vantagens
  - Tamanho ilimitado (limite na memória disponível na máquina)
  - Alocação de memória apenas para os elementos que queremos representar
  - Inserção e remoção simples

- Desvantagens
  - Mais difícil o acesso ao n-ésimo elemento

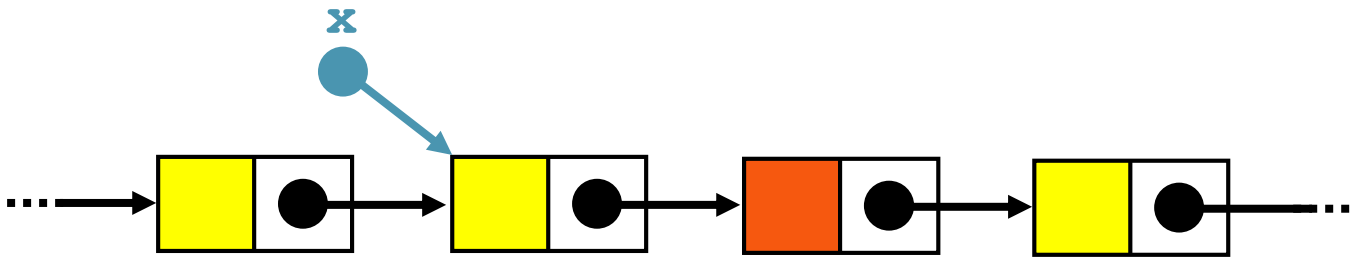
Inserção na Lista de um elemento *t depois de *x

```
t->next = x->next;
x->next = t;
```



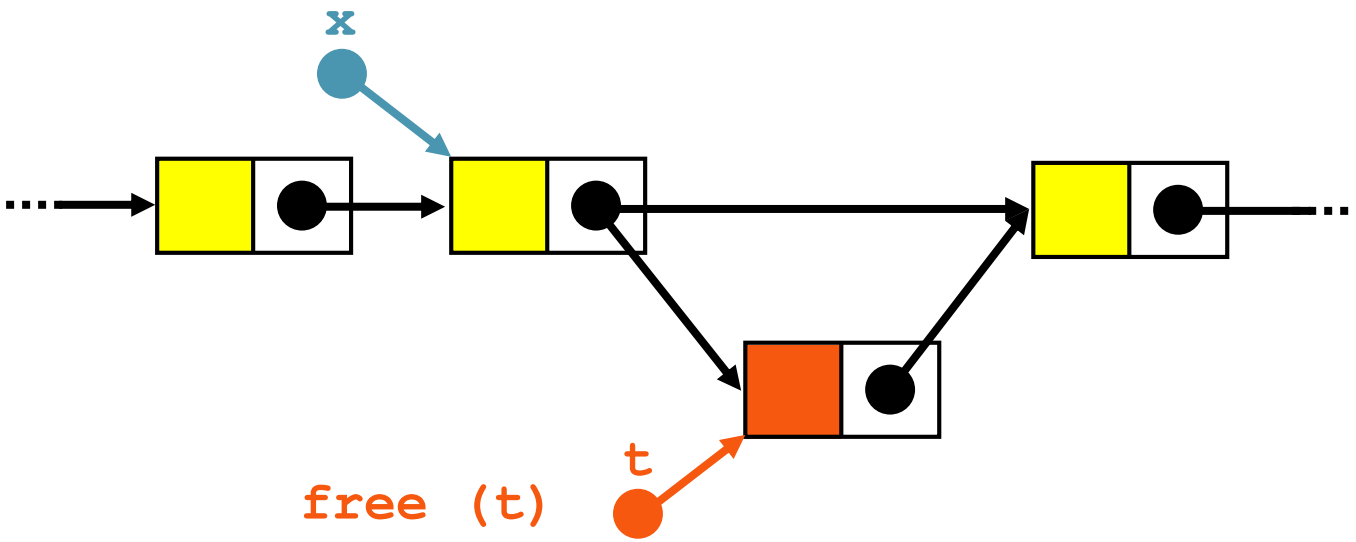
Inserção na Lista

```
t->next = x->next;
x->next = t;
```



Remoção do elemento depois de `x` da Lista

```
t = x->next;
x->next = t->next;
free(t);
```



Exemplo: Lista de Inteiros

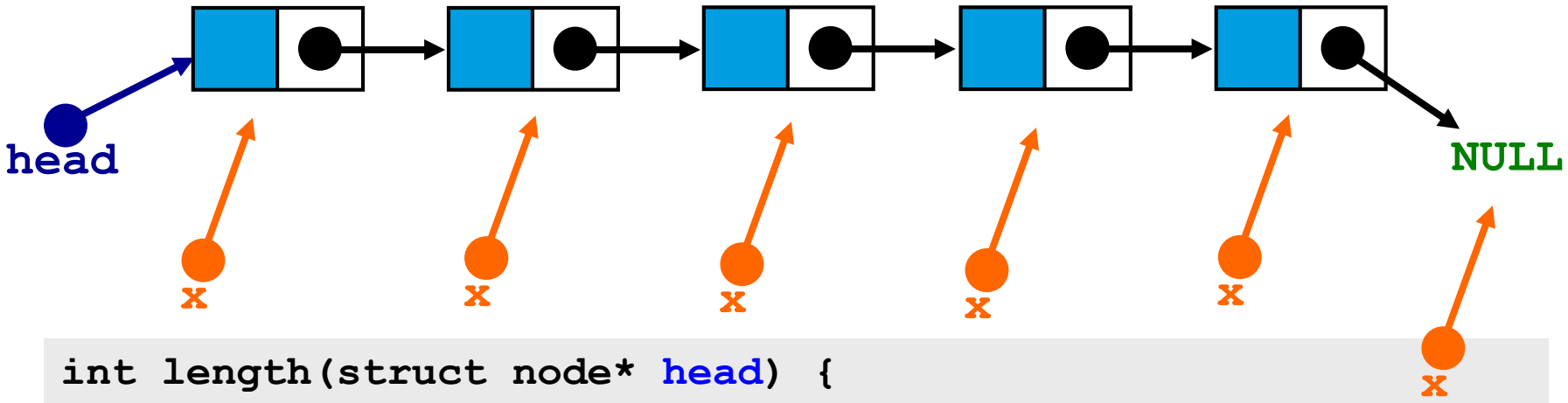
Problema:

- Como obter o número de elementos da Lista?

Solução:

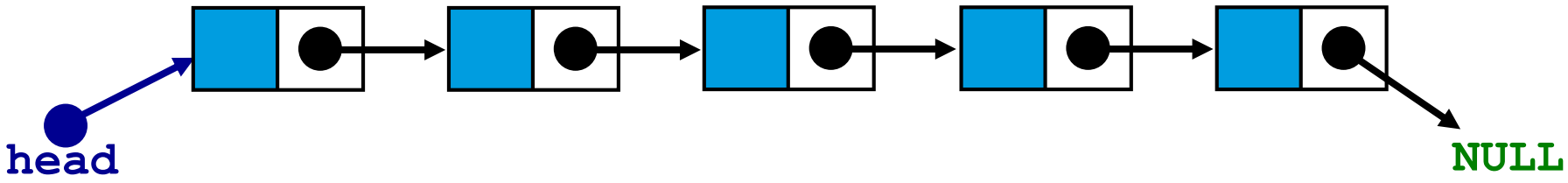
- Necessário percorrer toda a Lista!!

Exemplo: Lista de Inteiros



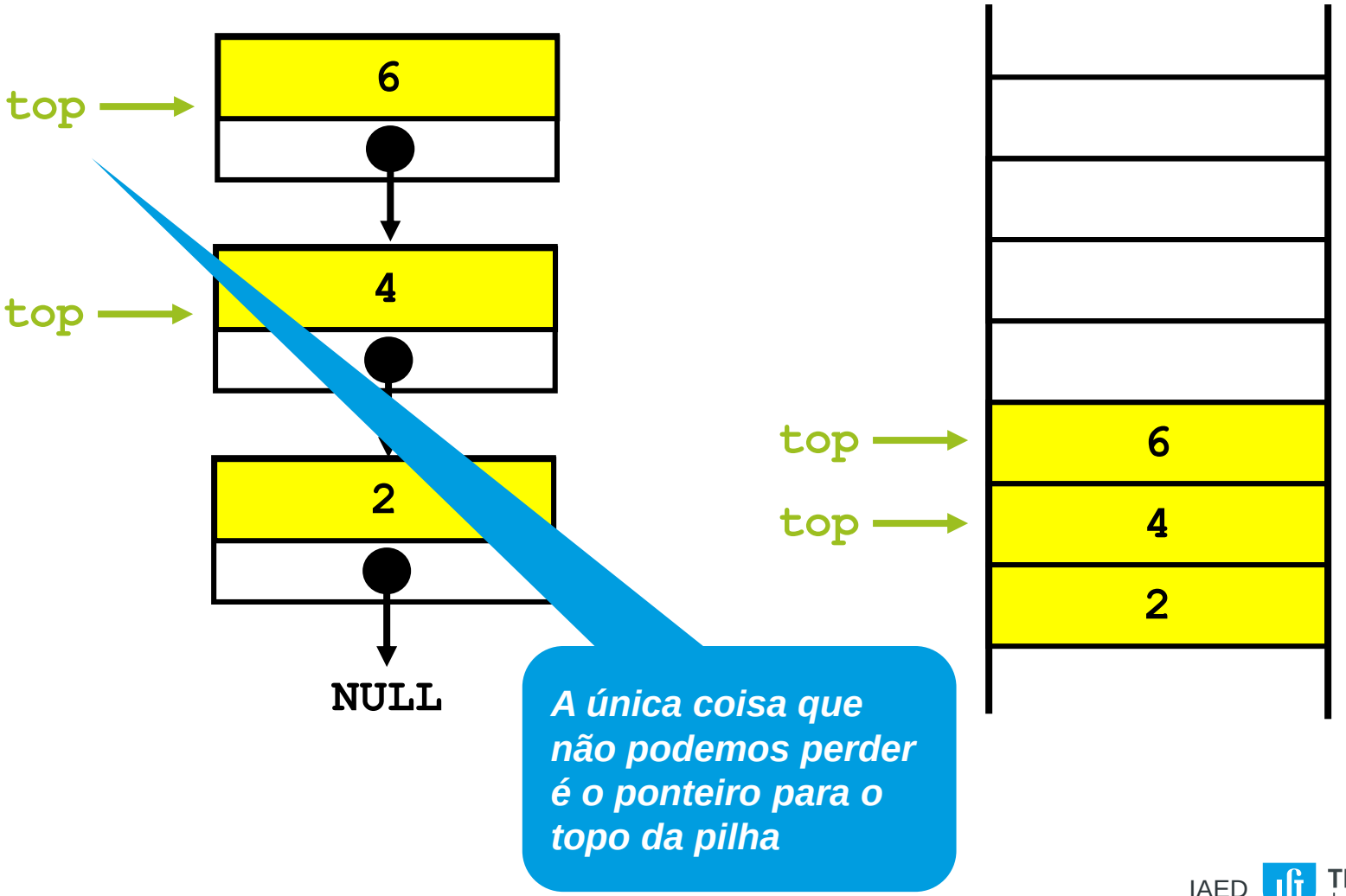

```
int length(struct node* head) {
    int count = 0;
    struct node *x = head;
    while(x != NULL) {
        count++;
        x = x->next;
    }
    return count;
}
```

Exemplo: Lista de Inteiros




```
int length(struct node* head) {
    int count = 0;
    struct node *x;

    for(x = head; x != NULL; x = x->next)
        count++;
    return count;
}
```

Exemplo: Pilha Dinâmica de Inteiros



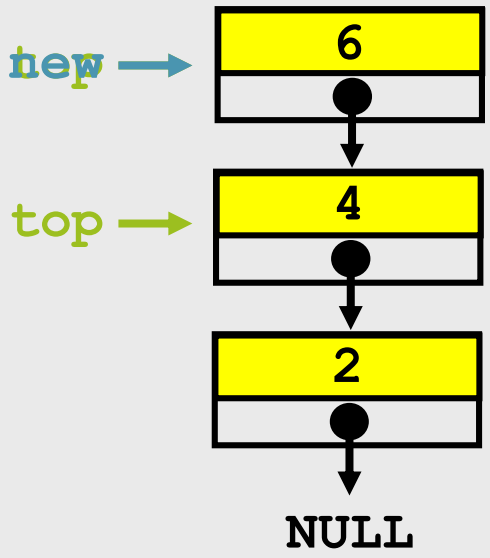
Exemplo: Pilha Dinâmica de Inteiros

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

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

static struct node *top;

void init() /* inicializa a pilha */
{
   top = NULL;
}
```

Variável global

Exemplo: Pilha Dinâmica de Inteiros

```
void push(int value) /* introduz novo elemento no topo */
{
  struct node *new;

  new = (struct node *) malloc(sizeof(struct node));
  new->value = value;
  new->next = top;
  top = new;
}
```



Exemplo: Pilha Dinâmica de Inteiros

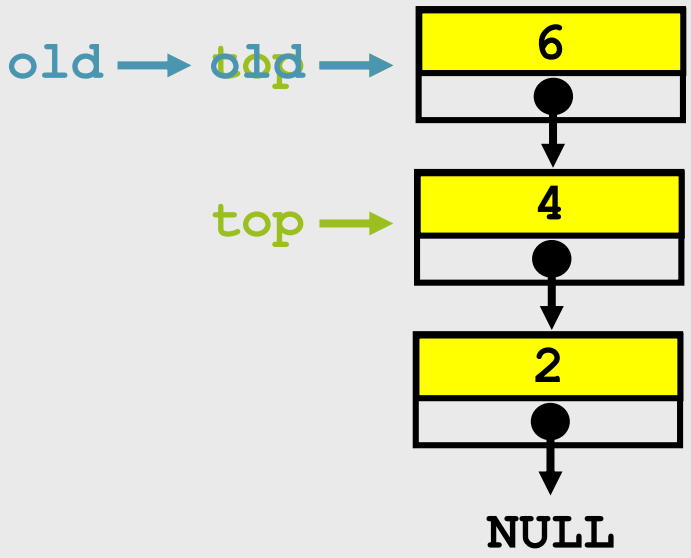
```
int is_empty() /* pergunta se está vazia */
{
  if(top == NULL) return 1;
  return 0;
}
```

Exemplo: Pilha Dinâmica de Inteiros

```
int is_empty() /* pergunta se está vazia */
{
  return top == NULL;
}
```

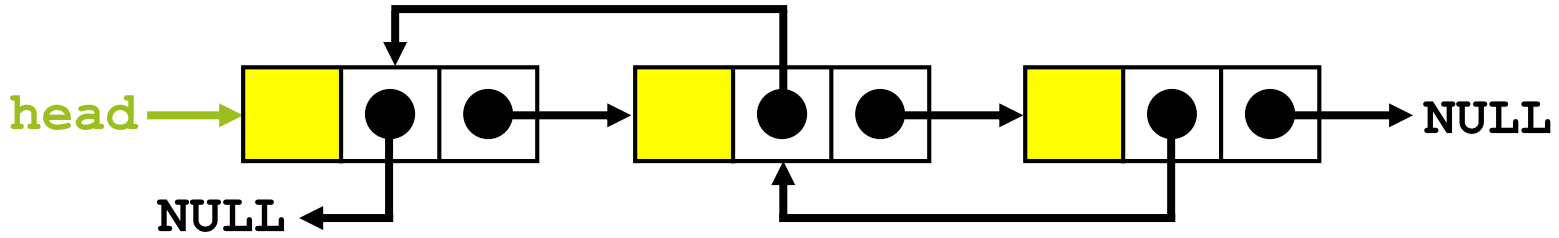
Exemplo: Pilha Dinâmica de Inteiros

```
int pop() /* apaga o topo e retorna o valor apagado */
{
  int value;
  struct node *old;

  if (!is_empty()) {
    value = top->value;
    old = top;
    top = top->next;
    free(old);
    return value;
  }
  else
    return -1;
}
```



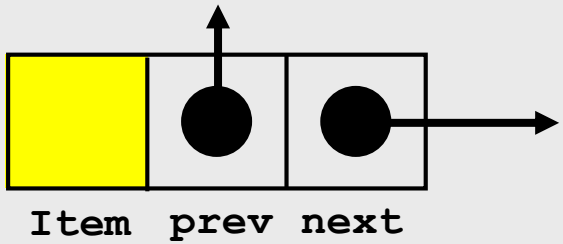
Lista Duplamente Ligada

- Conjunto de nós



- Cada nó contém
  - Informação útil
  - Ponteiro para o próximo nó e para o nó anterior

```
struct node {
  Item item;
  struct node *prev, *next;
};
```



Utilização de `typedef`

- É usual utilizar `typedef` na manipulação de estruturas auto-referenciadas

```
struct node{
  int value;
  struct node *next;
};

typedef struct node  Node;
typedef struct node* Node_ptr;
```

Utilização de `typedef`

- É usual utilizar `typedef` na manipulação de estruturas auto-referenciadas

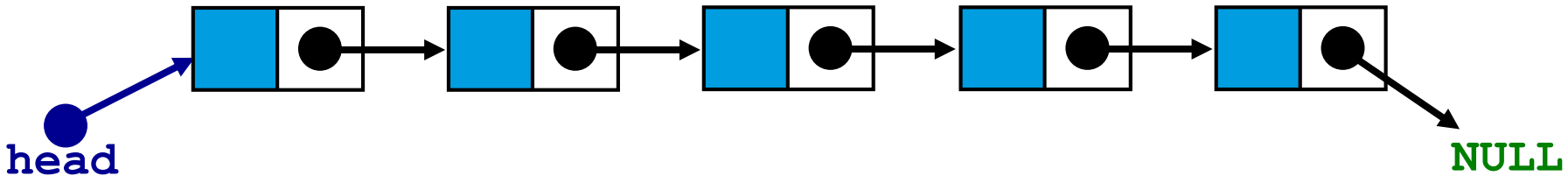
```
struct node{
  int value;
  struct node *next;
};

typedef struct node  Node;
typedef struct node* link;
```

Utilização de `typedef`

- É usual utilizar `typedef` na manipulação de estruturas auto-referenciadas

```
typedef struct node{
  int value;
  struct node *next;
} * link;
```

Utilização de `typedef`



Exemplo:

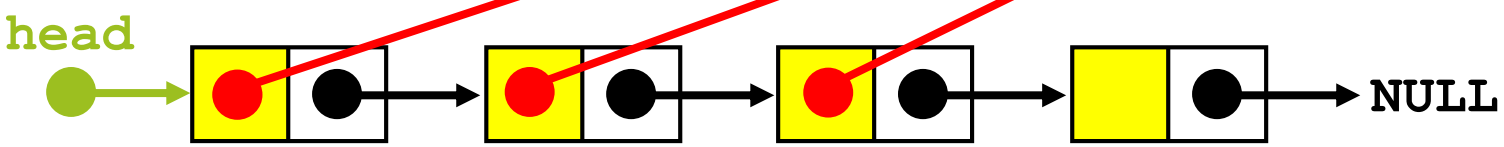
```
int length(link head) {
    int count=0;
    link x;

    for(x = head; x != NULL; x = x->next)
        count++;
    return count;
}
```

Criar lista com Argumentos da linha de comandos

```
$ ./myprogram bolo1 bolo2 bolo3 bolo4 bolo5
```

```
int main(int argc, char* argv[]){
...
}
```

argc = 6
argv[0] = “myprogram”
argv[1] = “bolo1”
argv[2] = “bolo2”
argv[3] = “bolo3”
...



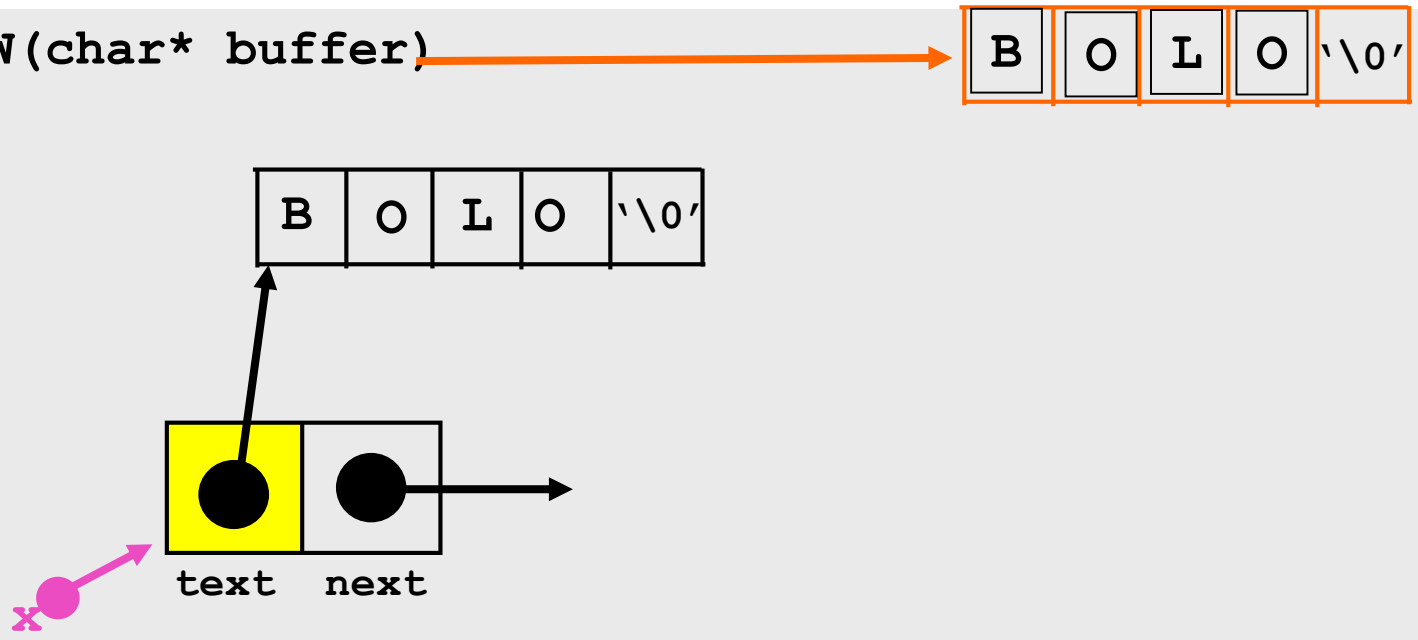
Criar lista com Argumentos da linha de comandos

```
int main(int argc, char* argv[])
{
    int i;
    link head = NULL;
    /* inserir todos os elementos na lista*/
    for(i = 1; i < argc; i++)
        head = insertEnd(head, argv[i]);
    print(head);              /* imprime toda a lista*/
    /* remover o elemento na posição i (lido do stdin) */
    scanf("%d", &i);
    head = delete(head, argv[i]);
    print(head);    /* voltamos a imprimir toda a lista */
    return 0;
}
```

Novo Elemento

Função auxiliar NEW responsável pela alocação de memória de tudo o que é necessário para um novo nó, i.e., a estrutura e a string.

```
link NEW(char* buffer)
{
}
```

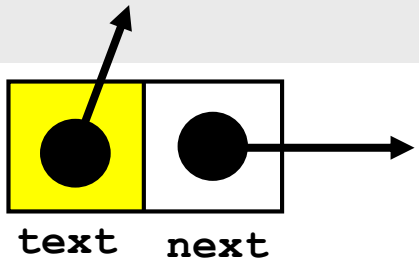


Novo Elemento

Função auxiliar NEW responsável pela alocação de memória de tudo o que é necessário para um novo nó, i.e., a estrutura e a string.

Reservar memória para o novo nó

```
link NEW(char* buffer)
{
    link x = ? ;
    x->text =
        ? ;
    strcpy(x->text, buffer);
    x->next = NULL;
    return x;
}
```

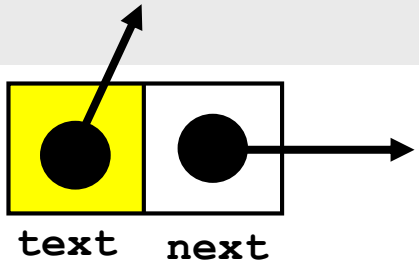


Novo Elemento

Função auxiliar NEW responsável pela alocação de memória de tudo o que é necessário para um novo nó, i.e., a estrutura e a string.

Reservar memória para o novo nó

```
link NEW(char* buffer)
{
    link x = (link) malloc(sizeof(struct node));
    x->text =
        ?                                          ;
    strcpy(x->text, buffer);
    x->next = NULL;
    return x;
}
```

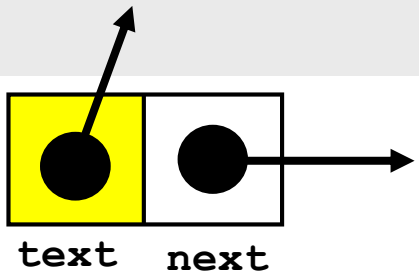


Novo Elemento

Função auxiliar NEW responsável pela alocação de memória de tudo o que é necessário para um novo nó, i.e., a estrutura e a string.

```
link NEW(char* buffer)
{
    link x = (link) malloc(sizeof(struct node));
    x->text =
        ?                                          ;
    strcpy(x->text, buffer);
    x->next = NULL;
    return x;
}
```

Reservar memória para a string

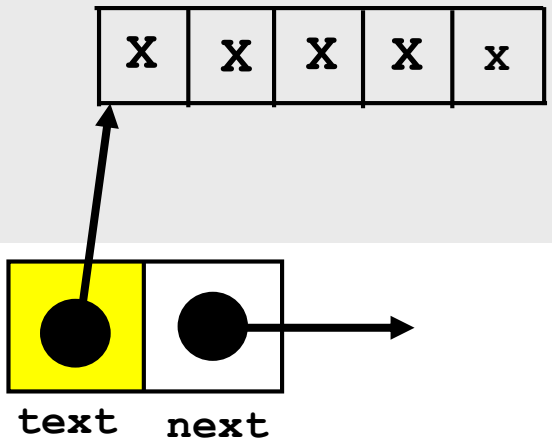


Novo Elemento

Função auxiliar NEW responsável pela alocação de memória de tudo o que é necessário para um novo nó, i.e., a estrutura e a string.

```
link NEW(char* buffer)
{
    link x = (link) malloc(sizeof(struct node));
    x->text =
       (char*) malloc(sizeof(char)*(strlen(buffer)+1));
    strcpy(x->text, buffer);
    x->next = NULL;
    return x;
}
```

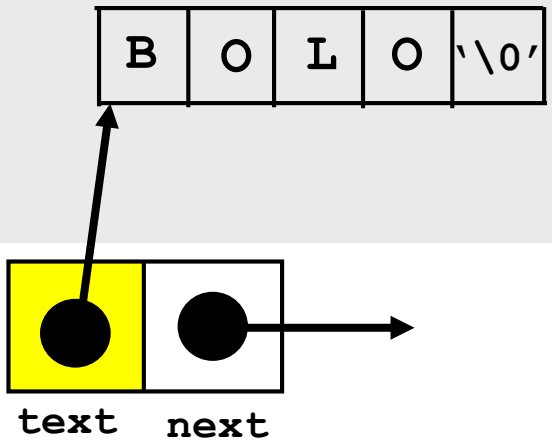
Reservar memória para a string



Novo Elemento

Função auxiliar NEW responsável pela alocação de memória de tudo o que é necessário para um novo nó, i.e., a estrutura e a string.

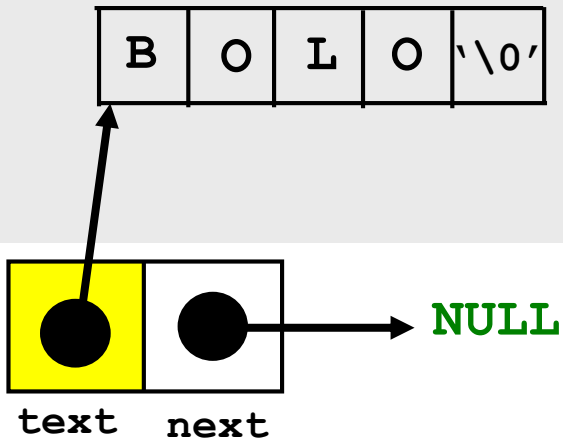
```
link NEW(char* buffer)
{
    link x = (link) malloc(sizeof(struct node));
    x->text =
       (char*) malloc(sizeof(char)*(strlen(buffer)+1));
    strcpy(x->text, buffer);
    x->next = NULL;
    return x;
}
```



Novo Elemento

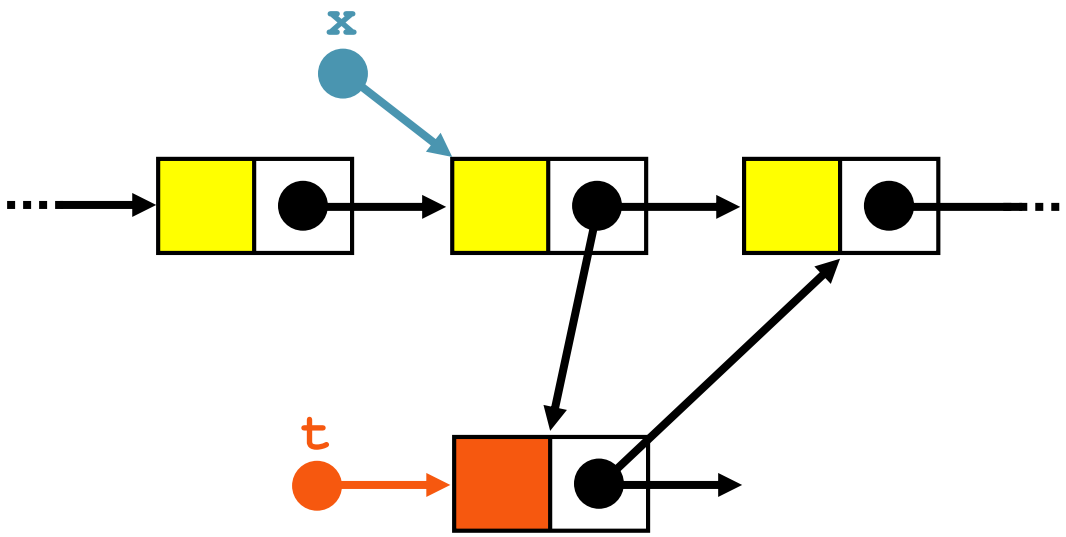
Função auxiliar NEW responsável pela alocação de memória de tudo o que é necessário para um novo nó, i.e., a estrutura e a string.

```
link NEW(char* buffer)
{
    link x = (link) malloc(sizeof(struct node));
    x->text =
       (char*) malloc(sizeof(char)*(strlen(buffer)+1));
    strcpy(x->text, buffer);
    x->next = NULL;
    return x;
}
```



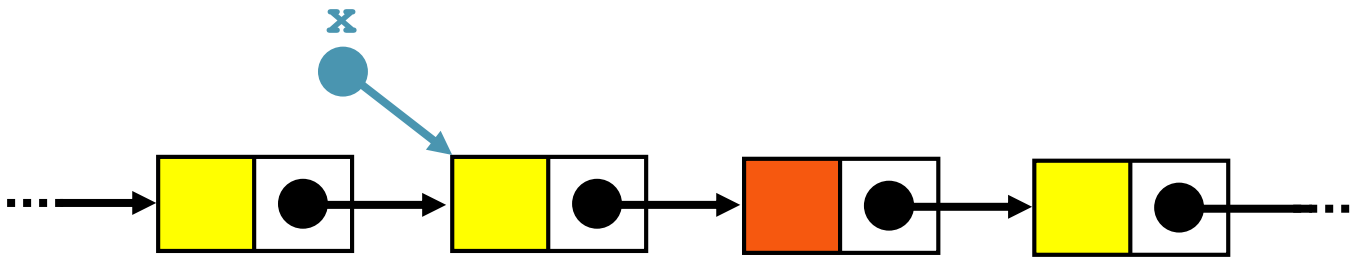
Inserção na Lista

```
t->next = x->next;
x->next = t;
```



Inserção na Lista

```
t->next = x->next;
x->next = t;
```



Inserção no Início (mais fácil)

```
link insertBegin(link head, char* text)
{
    link x = NEW(text);
    x->next = ? ;
    return x;
}
```

Inserção no Início

```
link insertBegin(link head, char* text)
{
    link x = NEW(text);
    x->next = head;
    return x;
}
```

Nesta versão, o insertBegin retorna sempre a (nova) "head"

```
int main(){
(...)
      head = insertBegin(head, "Bolo");
(...)
}
```

Inserção no Fim

```
link insertEnd(link head, char* text)
{
    link x;
    if(head == NULL)
        return NEW(text);
    for(                    ?                    )
        ;
    x->next = NEW(text);
    return head;
}
```

Como chego ao fim da lista?

Inserção no Fim

```
link insertEnd(link head, char* text)
{
    link x;
    if(head == NULL)
        return NEW(text);
    for(x = head; x->next != NULL; x = x->next)
            ;
    x->next = NEW(text);
    return head;
}
```

Como chego ao fim da lista?

Recorro à minha função NEW para alocar memória para um novo nó e copiar a string para lá.
A função NEW já coloca o next do novo nó a NULL

Procura na Lista

```
link lookup(link head, char* text)
{




}
```

Procura na Lista

```
link lookup(link head, char* text)
{
    link t;
    for(t = head; t != NULL; t = t->next)
        if(                ?                )
            return t;
    return NULL;
}
```

Procura na Lista

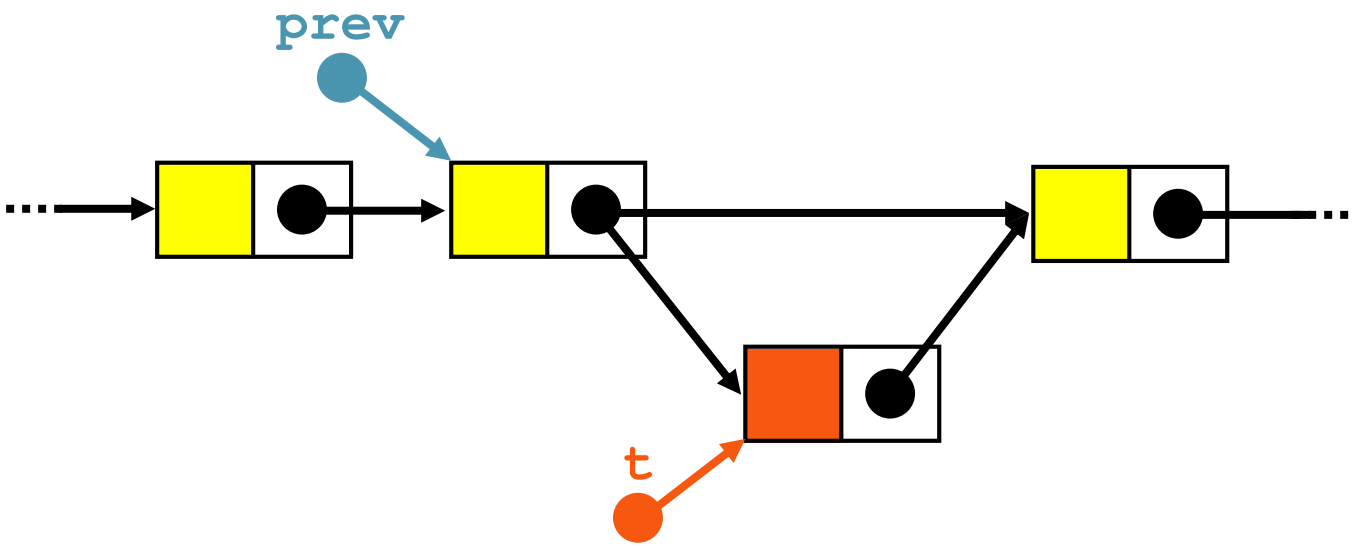
```
link lookup(link head, char* text)
{
    link t;
    for(t = head; t != NULL; t = t->next)
        if(strcmp(t->text, text) == 0)
            return t;
    return NULL;
}
```

Imprime o conteúdo da Lista

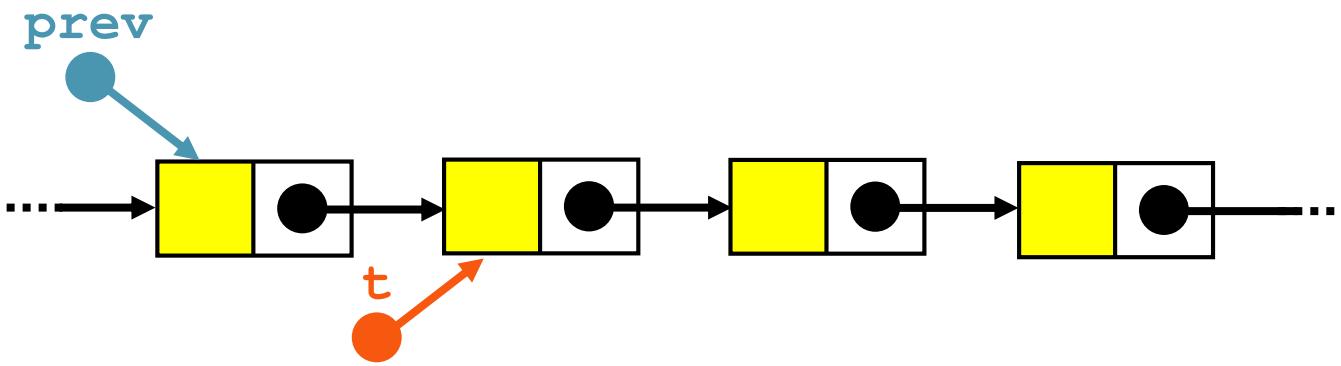
```
void print(link head)
{
    link t;
    for(t = head; t != NULL; t = t->next)
        printf("%s\n",t->text);
}
```

Remoção da Lista

```
t = prev->next;
prev->next = t->next;
```



Remoção da Lista com Procura



Remoção da Lista com Procura

```
link delete(link head, char* text)
{
    link t, prev;
    for(t = head, prev = NULL; t != NULL;
        prev = t, t = t->next) {
        if(strcmp(t->text, text) == 0) {
            if(t == head)
                ?
            else
                prev->next = t->next;
            free(t->text);
            free(t);
            break;
        }
    }
    return head;
}
```

Remoção da Lista com Procura

```
link delete(link head, char* text)
{
    link t, prev;
    for(t = head, prev = NULL; t != NULL;
        prev = t, t = t->next) {
        if(strcmp(t->text, text) == 0) {
            if(t == head)
                head = t->next;
            else
                prev->next = t->next;
            free(t->text);
            free(t);
            break;
        }
    }
    return head;
}
```

Remoção da Lista com Procura

```
link delete(link head, char* text)
{
    link t, prev;
    for(t = head, prev = NULL; t != NULL;
        prev = t, t = t->next) {
        if(strcmp(t->text, text) == 0) {
            if(t == head)
                head = t->next;
            else
                prev->next = t->next;
            FREEnode(t);

            break;
        }
    }
    return head;
}
```

```
void FREEnode(link t)
{
   free(t->text);
   free(t);
}
```