UFRJ

INSTITUTO DE COMPUTAÇÃO UFRJ

Programação de Computadores II

Lista Ligada

Profa. Giseli Rabello Lopes

Sumário

- Lista Ligada
- Funções de gerenciamento
- Implementação estática
- Implementação dinâmica

Adaptado de material preparado por Profs. Luciano Digiampietri e Norton T. Roman

Lista Ligada

- Para evitar o deslocamento de elementos durante a inserção e a exclusão utilizaremos uma lista ligada:
  - É uma estrutura linear (cada elemento possui no máximo um predecessor e um sucessor)
  - A ordem lógica dos elementos (a ordem “vista” pelo usuário) não é a mesma ordem física (em memória principal) dos elementos
  - Cada elemento precisa indicar quem é o seu sucessor

Funções de gerenciamento

- Inicializar a estrutura
- Retornar a quantidade de elementos válidos
- Exibir os elementos da estrutura
- Buscar por um elemento na estrutura
- Inserir elementos na estrutura
- Excluir elementos da estrutura
- Reinicializar a estrutura

Lista Ligada - Implementação estática

Lista Ligada - Implementação estática

- Chamaremos de lista ligada implementação estática, porque nossos registros serão armazenados em um arranjo criado inicialmente
- Adicionalmente, cada elemento da nossa lista terá um campo para indicar a posição (no arranjo) de seu sucessor

Modelagem (lista estática)

```
#define MAX 50
#define INVALIDO -1

typedef int TIPOCHAVE;

typedef struct{
  TIPOCHAVE chave;
  // outros campos...
} REGISTRO;
```

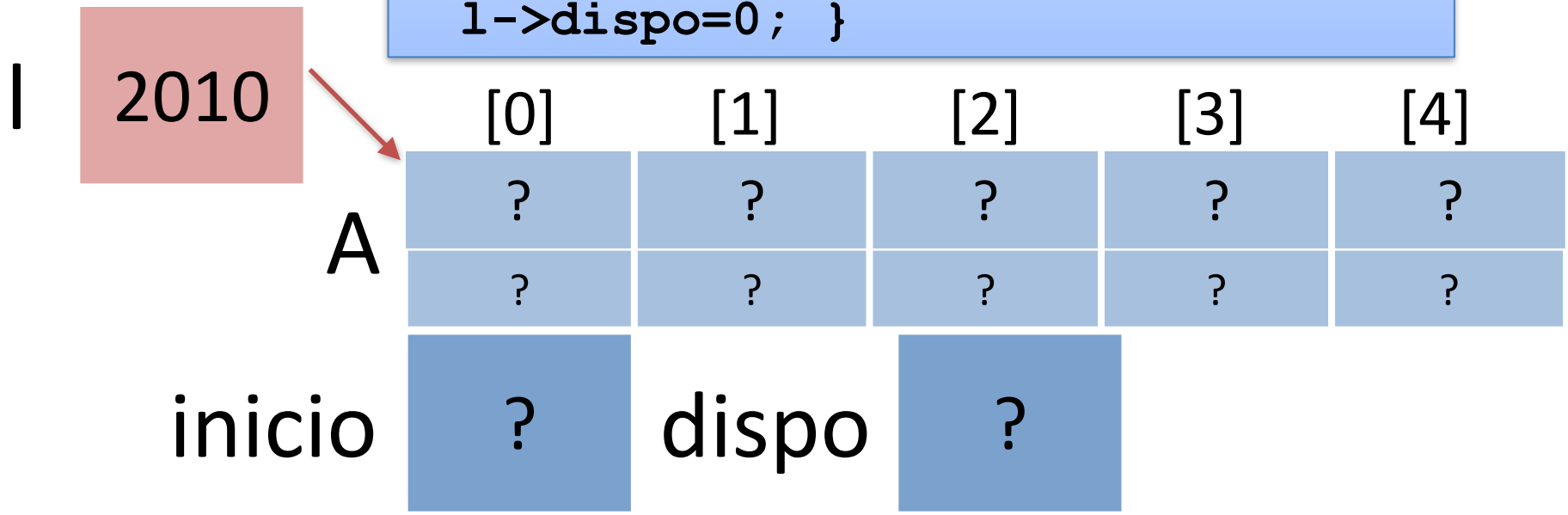
```
typedef struct{
  REGISTRO reg;
  int prox;
} ELEMENTO;

typedef struct {
  ELEMENTO A[MAX];
  int inicio;
  int dispo;
} LISTA;
```

Inicialização

- Para inicializar uma lista ligada (implementação estática), precisamos:
  - Colocar todos os elementos na "lista" de disponíveis
  - Acertar a variável **`dispo`** (primeiro item disponível)
  - Acertar a variável **`inicio`** (para indicar que não há nenhum item válido)

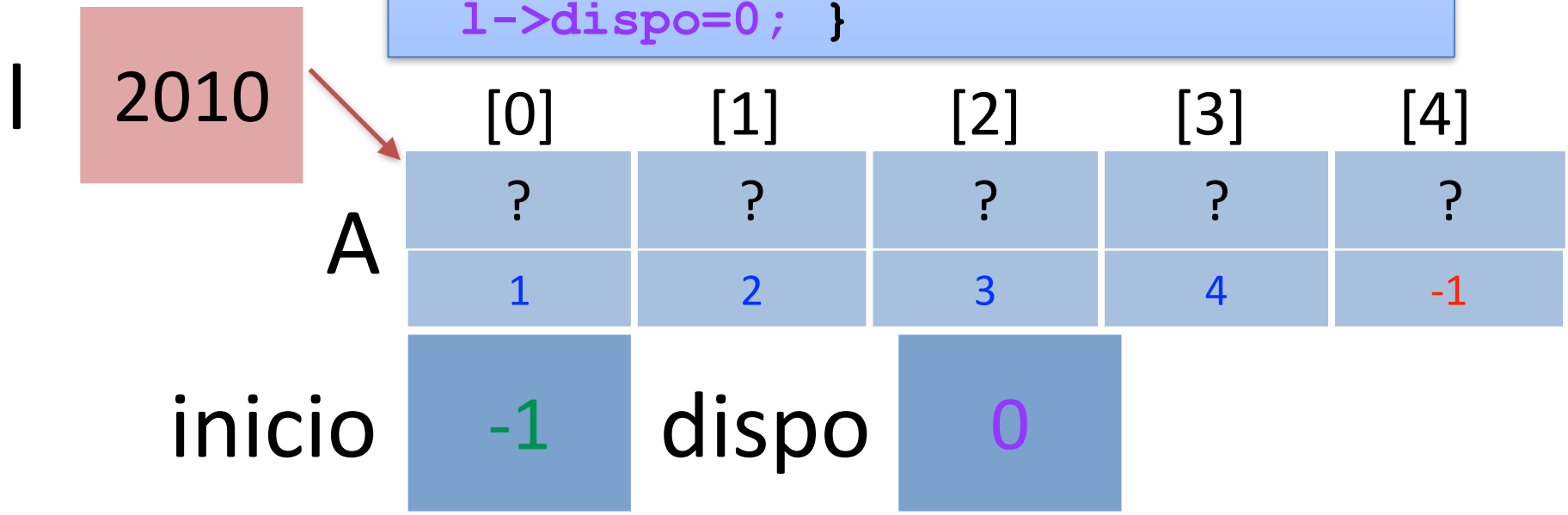
Inicialização

```
void inicializarLista(LISTA* l) {
  int i;
  for (i=0; i<MAX-1; i++)
    l->A[i].prox = i + 1;
  l->A[MAX-1].prox=INVALIDO;
  l->inicio=INVALIDO;
  l->dispo=0; }
```



Inicialização

```
void inicializarLista(LISTA* l) {
  int i;
  for (i=0; i<MAX-1; i++)
    l->A[i].prox = i + 1;
  l->A[MAX-1].prox=INVALIDO;
  l->inicio=INVALIDO;
  l->dispo=0; }
```

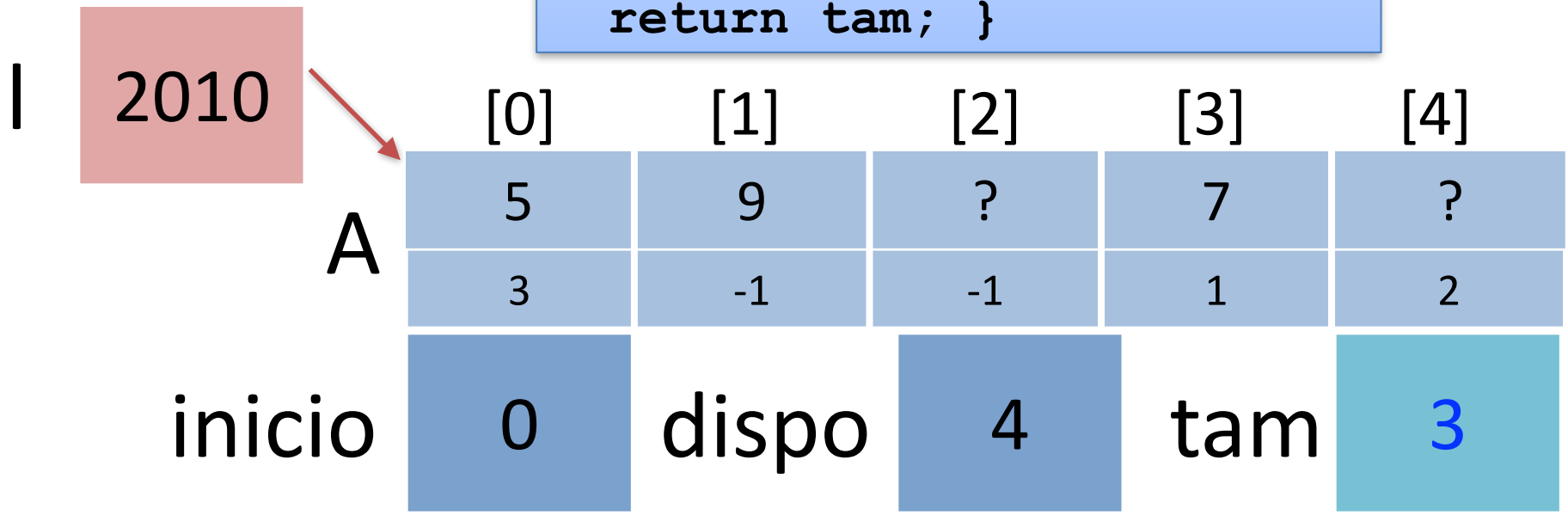


Retornar número de elementos

- Já que optamos por não criar um campo com o número de elementos na lista, precisaremos percorrer todos os elementos válidos para contar quantos são

Retornar número de elementos

```
int tamanho(LISTA* l) {
   int i = l->inicio;
   int tam = 0;
   while (i != INVALIDO){
      tam++;
      i = l->A[i].prox; }
   return tam; }
```



Exibição/Impressão

- Para exibir os elementos da estrutura precisaremos iterar pelos elementos válidos e, por exemplo, imprimir suas chaves

Exibição/Impressão

```
void exibirLista(LISTA* l) {
  int i = l->inicio;
  printf("Lista: \" ");
  while (i != INVALIDO) {
    printf("%i ", l->A[i].reg.chave);
    i = l->A[i].prox; }
  printf("\"\n"); }
```

l 2010

A

[0]	[1]	[2]	[3]	[4]
5	9	?	7	?
3	-1	-1	1	2

inicio 0 dispo 4

Saída:
`$ Lista: " 5 7 9 "`

Busca de elemento

(veremos mais adiante métodos de busca e ordenação)

- A função de busca deverá:
  - Receber uma chave do usuário
  - Retornar a posição em que este elemento se encontra no arranjo (caso seja encontrado)
  - Retornar INVALIDO caso não haja um registro com essa chave na lista

Inserção de um elemento

(veremos mais adiante métodos de busca e ordenação)

- O usuário passa como parâmetro um registro a ser inserido na lista
  - Realizaremos a inserção ordenada pelo valor da chave do registro passado e não permitiremos a inserção de elementos repetidos
  - Na inserção precisamos identificar entre quais elementos o novo elemento será inserido
  - O novo elemento será inserido no lugar do primeiro que estiver na lista de disponíveis

Exclusão de um elemento

(veremos mais adiante métodos de busca e ordenação)

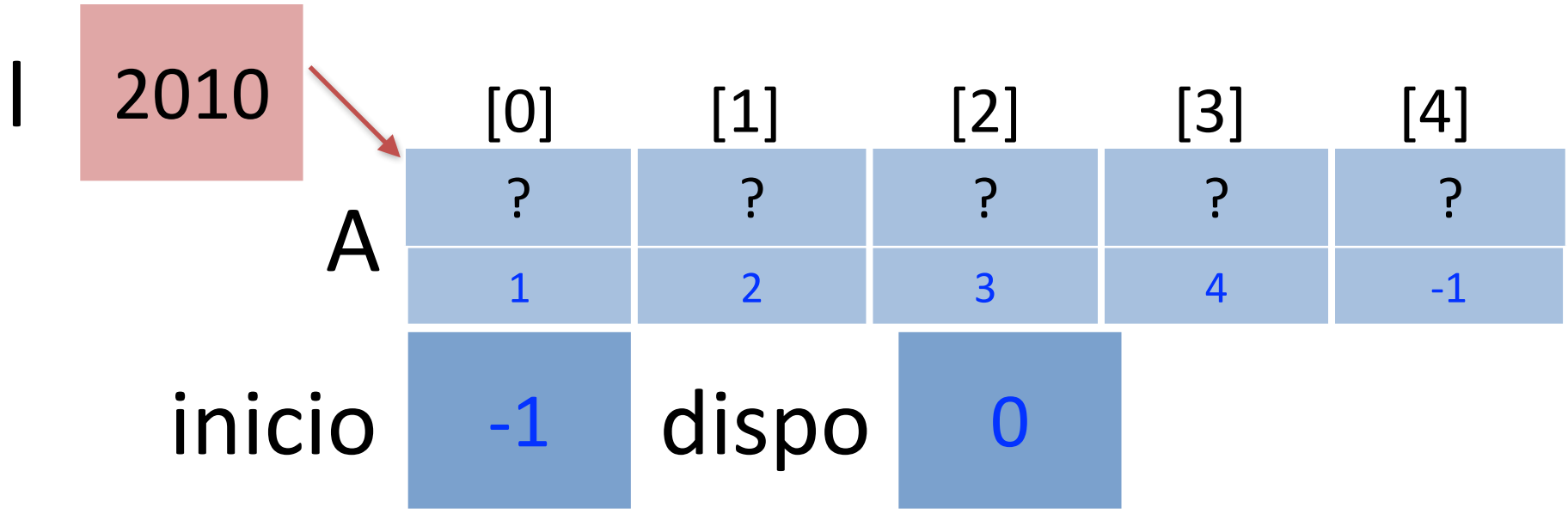
- O usuário passa a chave do elemento que ele quer excluir
  - Se houver um elemento com esta chave na lista, "exclui este elemento" da lista de elementos válidos e o insere na lista de disponíveis
  - Adicionalmente, acerta os "ponteiros" envolvidos e retorna *true*
  - Caso contrário, retorna *false*

Reinicialização

- Para reinicializar esta estrutura basta chamarmos a função de inicialização ou executarmos os mesmos comandos lá executados

Reinicialização

```
void reinicializarLista(LISTA* l) {
   inicializarLista(l);
}
```



Lista Ligada - Implementação dinâmica

Lista Ligada - Implementação dinâmica

- Alocaremos e desalocaremos a memória para os elementos sob demanda
- **Vantagem:** não precisamos gastar memória que não estamos usando
- Adicionalmente, cada elemento da nossa lista terá um ponteiro para indicar seu sucessor

Modelagem (lista dinâmica)

```
#include <stdio.h>
#include <stdlib.h>
#define true 1
#define false 0

typedef int bool;

typedef int TIPOCHAVE;

typedef struct {
  TIPOCHAVE chave;
  // outros campos...
} REGISTRO;
```

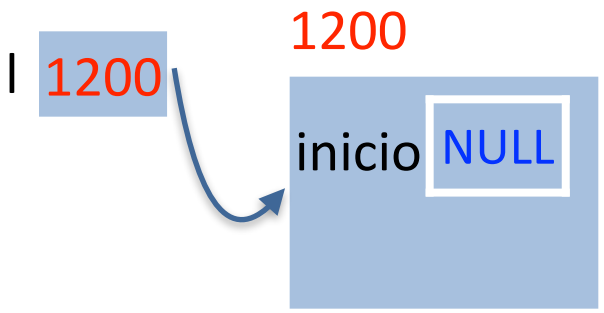
```
typedef struct aux {
  REGISTRO reg;
  struct aux* prox;
} ELEMENTO;

typedef ELEMENTO* PONT;

typedef struct {
  PONT inicio;
} LISTA;
```

Inicialização

- Para inicializar uma lista ligada (implementação dinâmica), precisamos:
  - Colocar o valor **`NULL`** na variável **`inicio`**

Inicialização

```
void inicializarLista(LISTA* l){
  l->inicio = NULL;
}
```

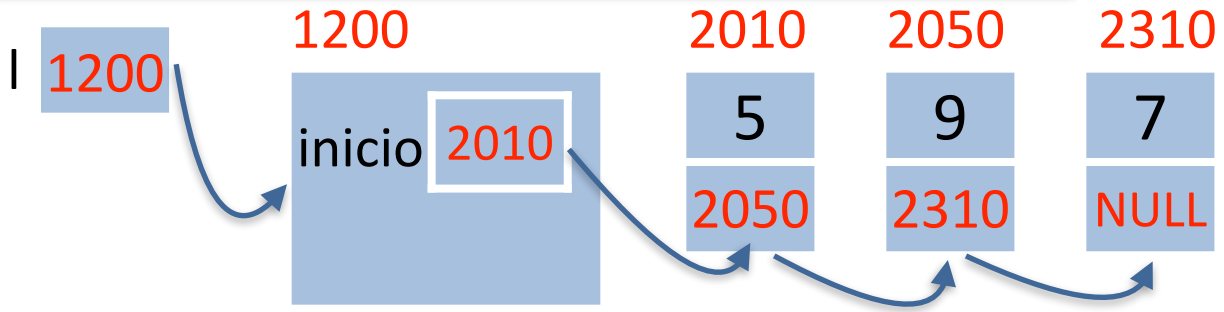


Retornar número de elementos

- Já que optamos por não criar um campo com o número de elementos na lista, precisaremos percorrer todos os elementos para contar quantos são

Retornar número de elementos

```
int tamanho(LISTA* l) {
   PONT end = l->inicio;
   int tam = 0;
   while (end != NULL) {
      tam++;
      end = end->prox;
   }
   return tam; }
```

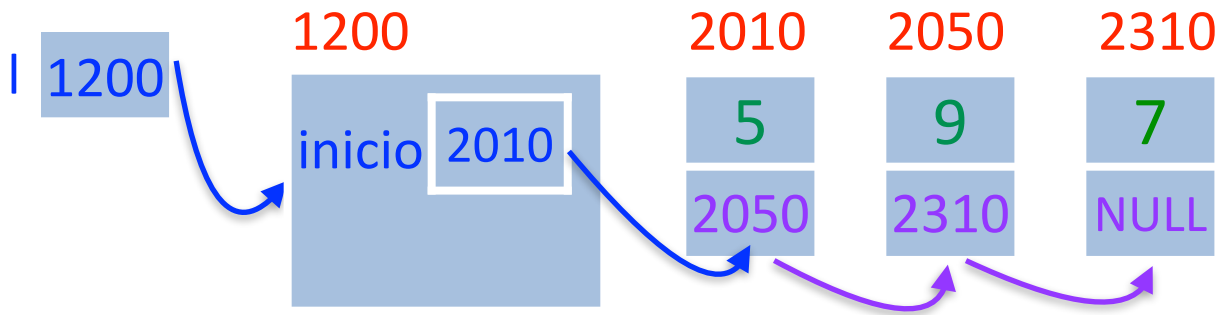


Exibição/Impressão

- Para exibir os elementos da estrutura precisaremos iterar pelos elementos e, por exemplo, imprimir suas chaves

Verificar Exibição/Impressão

```
void exibirLista(LISTA* l) {
  PONT end = l->inicio;
  printf("Lista: \" ");
  while (end != NULL) {
    printf("%i ", end->reg.chave);
    end = end->prox; }
  printf("\"\n"); }
```



Saída:
$ Lista: " 5 9 7 "

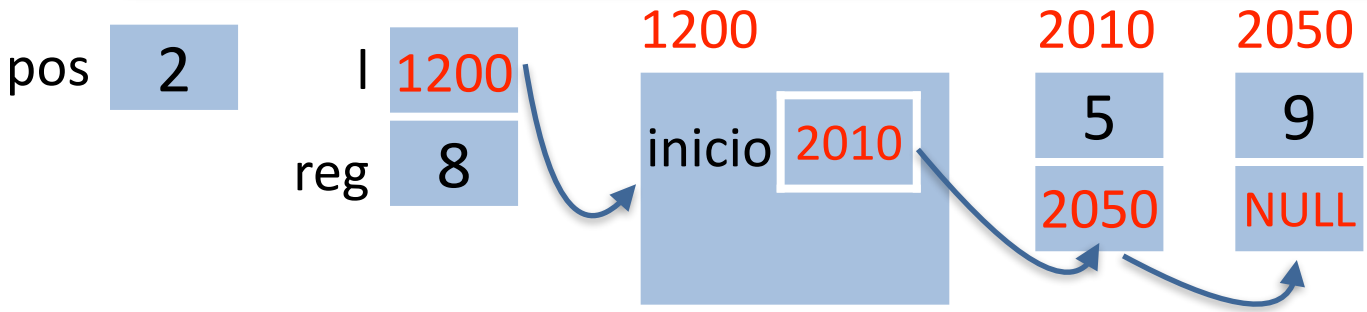
Busca de elemento

(veremos mais adiante métodos de busca e ordenação)

- A função de busca deverá:
  - Receber uma chave do usuário
  - Retornar o endereço em que este elemento se encontra no arranjo (caso seja encontrado)
  - Retornar **`NULL`** caso não haja um registro com essa chave na lista

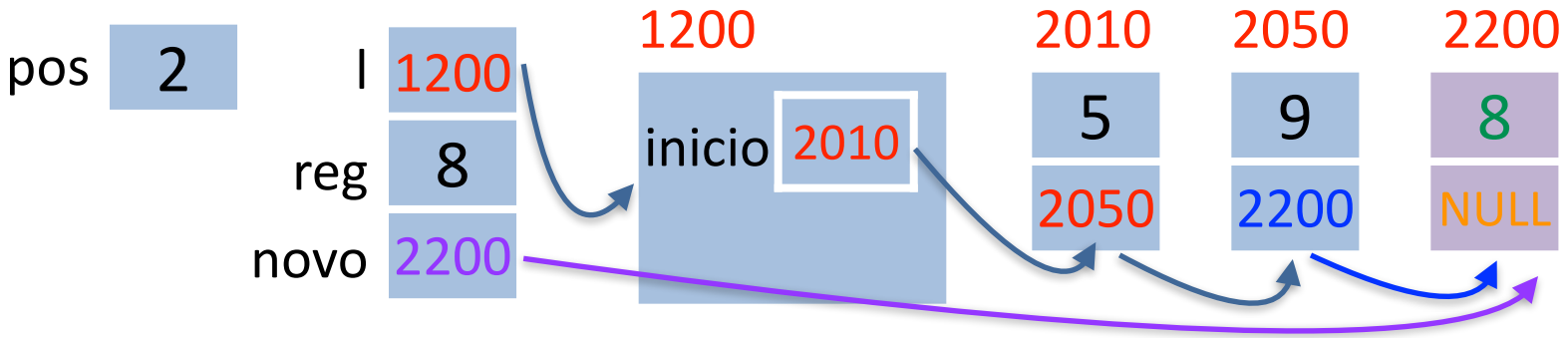
Inserção de um elemento em uma posição

```
bool insere(LISTA* l, REGISTRO reg, int pos) {
if (pos<0 || pos>tamanho(l)) return false;
ELEMENTO* novo = (ELEMENTO*) malloc(sizeof(ELEMENTO));
novo->reg = reg;
int i;
ELEMENTO* p;
if (pos == 0){
  novo->prox = l->inicio;
  l->inicio = novo;
}else{
  p = l->inicio;
  for (i=0;i<pos-1;i++) p = p->prox;
  novo->prox = p->prox;
  p->prox = novo;}
return true; }
```



Inserção de um elemento em uma posição

```
bool insere(LISTA* l, REGISTRO reg, int pos) {
if (pos<0 || pos>tamanho(l)) return false;
ELEMENTO* novo = (ELEMENTO*) malloc(sizeof(ELEMENTO));
novo->reg = reg;
int i;
ELEMENTO* p;
if (pos == 0){
  novo->prox = l->inicio;
  l->inicio = novo;
}else{
  p = l->inicio;
  for (i=0;i<pos-1;i++) p = p->prox;
  novo->prox = p->prox;
  p->prox = novo;}
return true; }
```



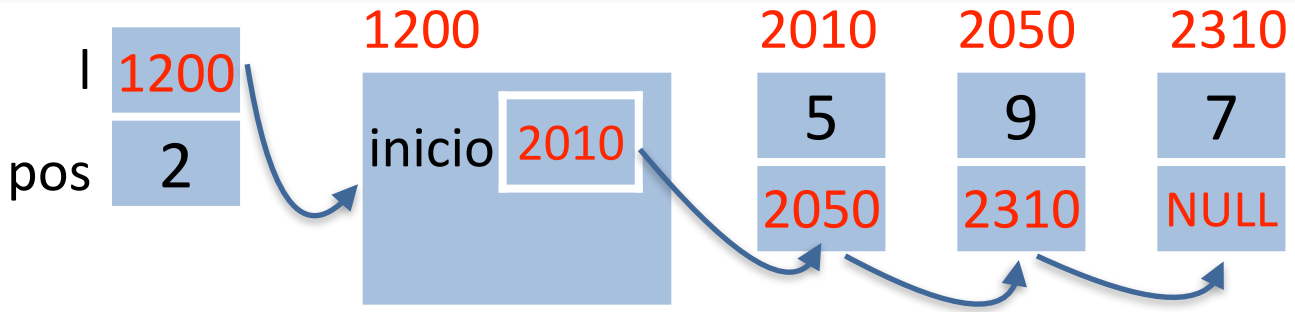
Inserção de um elemento

(veremos mais adiante métodos de busca e ordenação)

- O usuário passa como parâmetro um registro a ser inserido na lista
  - Realizaremos a inserção ordenada pelo valor da chave do registro passado e não permitiremos a inserção de elementos repetidos
  - Na inserção precisamos identificar entre quais elementos o novo elemento será inserido
  - Alocaremos memória para o novo elemento
  - Precisamos saber quem será o predecessor do elemento

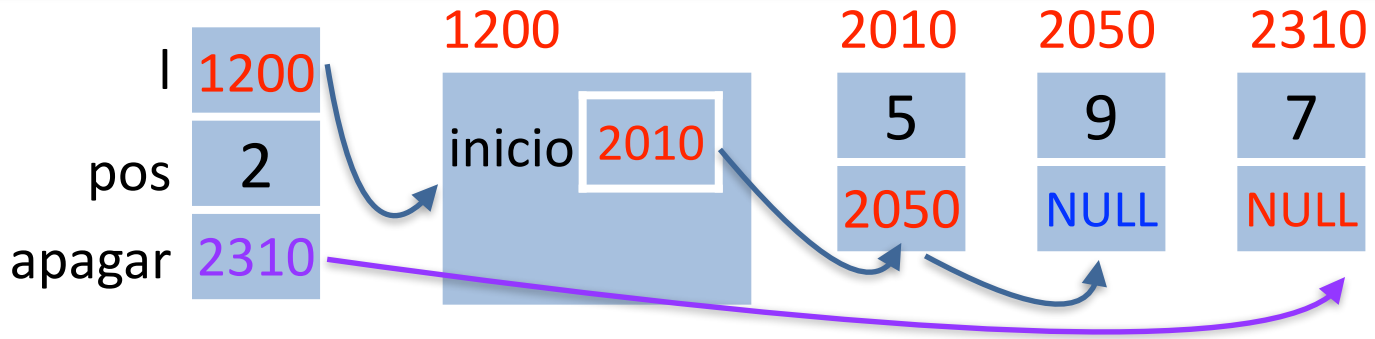
Exclusão de um elemento em uma posição

```
bool exclui(LISTA* l, int pos) {
  if (pos<0 || pos>tamanho(l)-1) return false;
  int i;
  ELEMENTO* p;
  ELEMENTO* apagar;
  if (pos == 0) {
    apagar = l->inicio;
    l->inicio = apagar->prox;
  }else {
    p = l->inicio;
    for (i=0;i<pos-1;i++) p = p->prox;
    apagar = p->prox;
    p->prox = apagar->prox; }
  free(apagar);
  return true; }
```



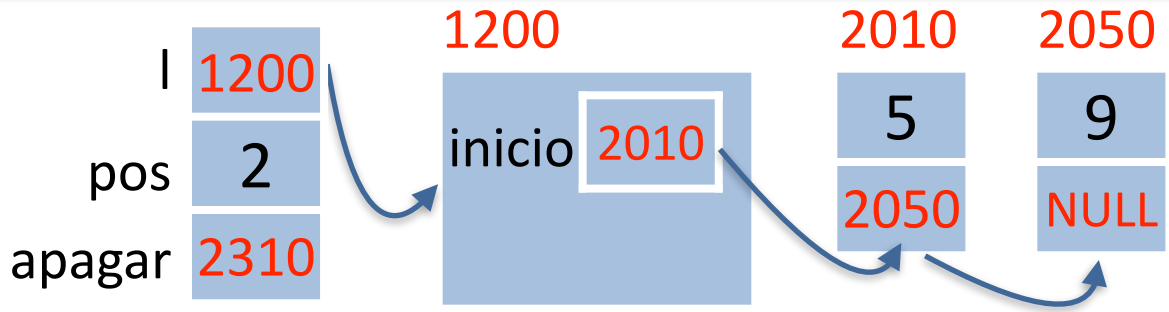
Exclusão de um elemento em uma posição

```
bool exclui(LISTA* l, int pos) {
  if (pos<0 || pos>tamanho(l)-1) return false;
  int i;
  ELEMENTO* p;
  ELEMENTO* apagar;
  if (pos == 0) {
    apagar = l->inicio;
    l->inicio = apagar->prox;
  }else {
    p = l->inicio;
    for (i=0;i<pos-1;i++) p = p->prox;
    apagar = p->prox;
    p->prox = apagar->prox; }
  free(apagar);
  return true; }
```



Exclusão de um elemento em uma posição

```
bool exclui(LISTA* l, int pos) {
  if (pos<0 || pos>tamanho(l)-1) return false;
  int i;
  ELEMENTO* p;
  ELEMENTO* apagar;
  if (pos == 0) {
    apagar = l->inicio;
    l->inicio = apagar->prox;
  }else {
    p = l->inicio;
    for (i=0;i<pos-1;i++) p = p->prox;
    apagar = p->prox;
    p->prox = apagar->prox; }
  free(apagar);
  return true; }
```



Exclusão de um elemento

(veremos mais adiante métodos de busca e ordenação)

- O usuário passa a chave do elemento que ele quer excluir
  - Se houver um elemento com esta chave na lista, exclui este elemento da lista de elementos, acerta os ponteiros envolvidos e retorna *true*
  - Caso contrário, retorna *false*
  - Para esta função precisamos saber quem é o predecessor do elemento a ser excluído

Reinicialização

- Para reinicializar a estrutura, precisamos excluir todos os seus elementos e atualizar o campo **`inicio`** para **`NULL`**

Reinicialização

```
void reinicializarFila(LISTA* l) {
  PONT end = l->inicio;
  while (end != NULL) {
    PONT apagar = end;
    end = end->prox;
    free(apagar); }
  l->inicio = NULL;
}
```