# Estrutura de Dados

Hamilton José Brumatto

Bacharelado em Ciências da Computação - UESC

2 de fevereiro de 2024

# Listas

# É uma estrutura linear de dados

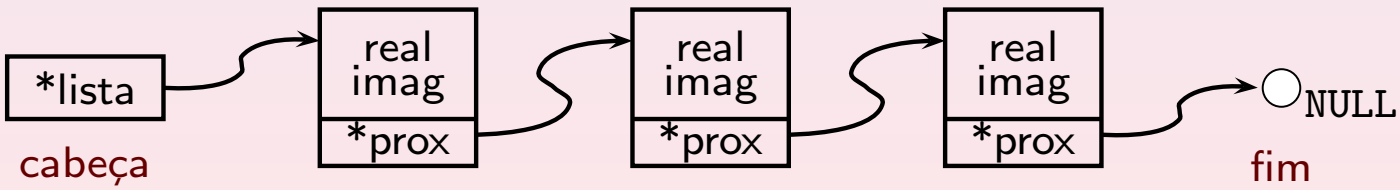
- Uma lista encadeada (ou ligada) permite criar um modelo dinâmico.
- Não há uma indexação dos elementos, um elemento está ligado ao próximo.
- A desvantagem em relação a um vetor é que o acesso não é aleatório, e sim sequencial.
  - Para chegar em uma posição tem de passar pelos anteriores.
- A navegação entre os elementos se dá através de ponteiros.
- A ligação entre os elementos se dá através de ponteiros.
- Cada elemento é criado dinamicamente através de alocação de memória. Não há limite como nos vetores.
- Um ponteiro especial: "cabeça", sempre aponta para o primeiro elemento da lista.

# Exemplo básico de uma lista encadeada

- Declarando a estrutura

```
typedef struct complexo_s {
    double real;
    double imag;
    struct complexo_s *prox;
} complexo;
```

- A lista é simplesmente um conjunto de células encadeadas.
- A primeira célula aponta para a segunda.
- A segunda célula aponta para a terceira...
- A última célula é aterrada (aponta para o NULL)




`NULL` *é uma constante declarada em alguns includes, por exemplo,* `stdlib.h` e `stddef.h`

# Exemplo de lista

Vamos construir, manualmente, a lista anterior:

```
int main(int argc, char **args) {
      complexo *lista, *p;
      lista = malloc(sizeof(complexo)); // Alocamos o primeiro elemento
      p = lista;                  // lista é o cabeça, aponta para o primeiro elemento.
      p->real = 2; p->imag = 3;
      p->prox = malloc(sizeof(complexo)); // Prox recebe novo elemento
      p = p->prox;                // p agora aponta para o segundo elemento
      p->real = 3; p->imag = 4;
      p->prox = malloc(sizeof(complexo));
      p = p->prox;
      p->real = 4; p->imag = 5;
      p->prox = NULL; // O último elemento aponta (prox) para NULL
      p = lista;
      while(p != NULL) { // Navegando na lista
            printf("(%f + i%f)..",p->real,p->imag); p=p->prox;
      }
      printf("\n");
      return 0;
}
```

```
(2.000000 + i3.000000)..(3.000000 + i4.000000)..(4.000000 + i5.000000)..
```

# Ponteiro para ponteiro para ...

- Quando temos um ponteiro para uma estrutura, podemos apontar para um elemento da estrutura:
  `p->real`, por exemplo.
- O elemento pode também ser outro ponteiro, que pode apontar para um elemento da estrutura que aponta...

```
p = lista;
printf("(%f + i%f)\n",p->prox->real, p->prox->prox->imag);
```

- O que resulta do código acima:

```
(3.000000 + i5.000000)
```

# Manipulando a estrutura da lista

- A busca se dá de forma sequencial, temos sempre de começar pelo cabeça ($O(n)$)
- Para inserir e remover um elemento, precisamos buscá-lo. Então sempre existe o custo da busca.
- Inserir um elemento implica em ter um ponteiro para o anterior (busca), fazer o elemento anterior apontar para o que será inserido e este para o próximo. $O(1)$
- Para remover um elemento precisamos de um ponteiro para o anterior (busca), e fazer o elemento anterior apontar para o próximo.
- A lista encadeada que mostramos carrega como informação dois valores que representam um complexo.

# Implementando uma Lista Encadeada:

- Para implementar e manipular uma lista precisamos de funções que realizam:
  - Criar um elemento para a lista;
  - Percurso (busca) na lista encadeada
  - Inserção de elementos
  - Remoção de elementos
  - Troca de posição entre elementos

```
#ifndef __LISTA_H__
#define __LISTA_H__

typedef struct complexo_s {
  double real;
  double imag;
  complexo *prox;
} complexo;

complexo *criar(double real, double imag);
complexo *busca(complexo *lista, int pos);
complexo *inserir(complexo *lista, complexo *elemento, int pos);
complexo *remover(complexo *lista, int pos);
complexo *trocar(complexo *lista, int posA, int posB);

#endif // __LISTA_H__
```

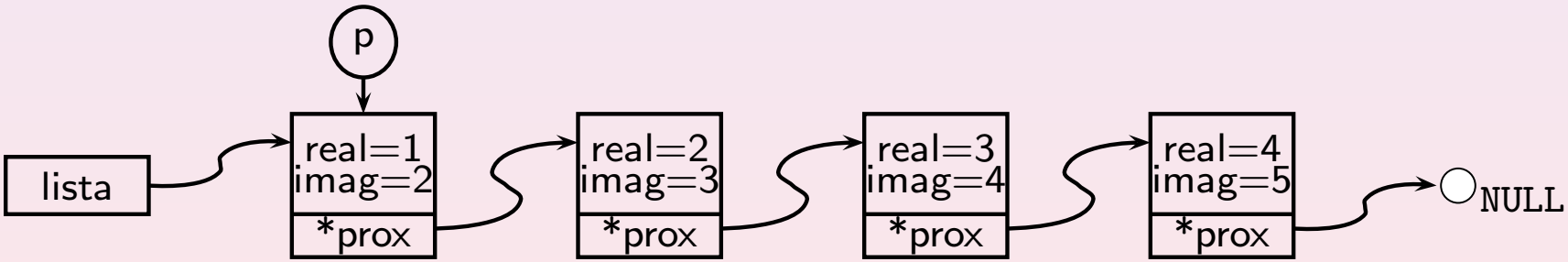
# Criar um elemento

- Precisamos alocar a memória para um elemento
- Precisamos iniciar os valores do elemento
- Precisamos aterrar (NULL) o ponteiro de próximo do elemento

```
complexo *criar(double real, double imag) {
  complexo *elemento = malloc(sizeof (complexo));
  elemento->real = real;
  elemento->imag = imag;
  elemento->prox = NULL;
  return elemento;
}
```

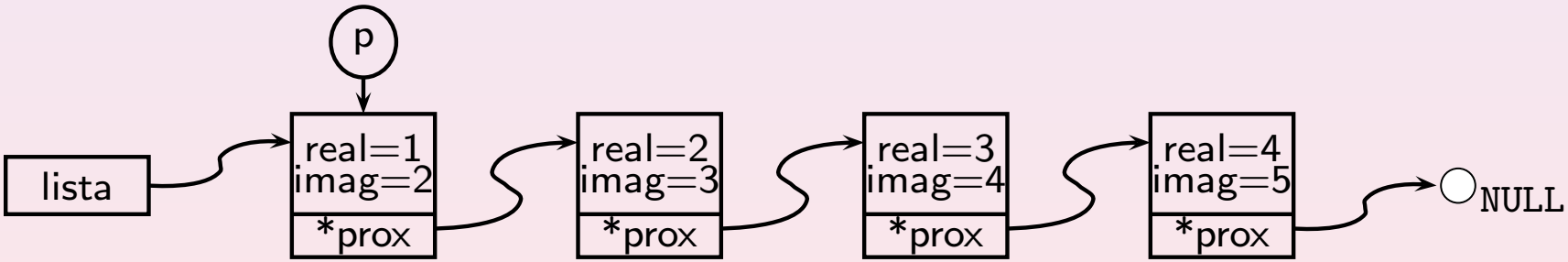
# Percurso na lista

```
p = lista; pos = 0;
while(p != NULL) {
    printf("pos = %d (%f + i%f)\n",pos,p->real,p->imag);
    p=p->prox; pos++;
}
```



# Percurso na lista

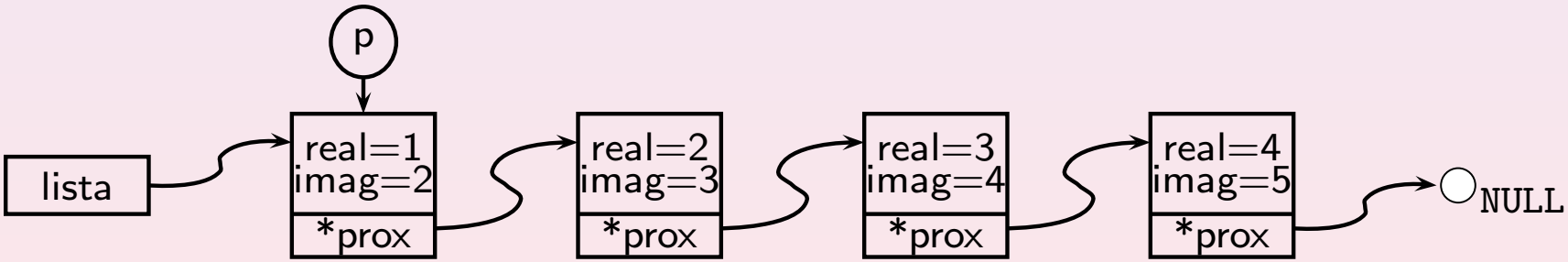
```
p = lista; pos = 0;
while(p != NULL) {
    printf("pos = %d (%f + i%f)\n",pos,p->real,p->imag);
    p=p->prox; pos++;
}
```



# Percurso na lista

```
p = lista; pos = 0;
while(p != NULL) {
    printf("pos = %d (%f + i%f)\n",pos,p->real,p->imag);
    p=p->prox; pos++;
}
```

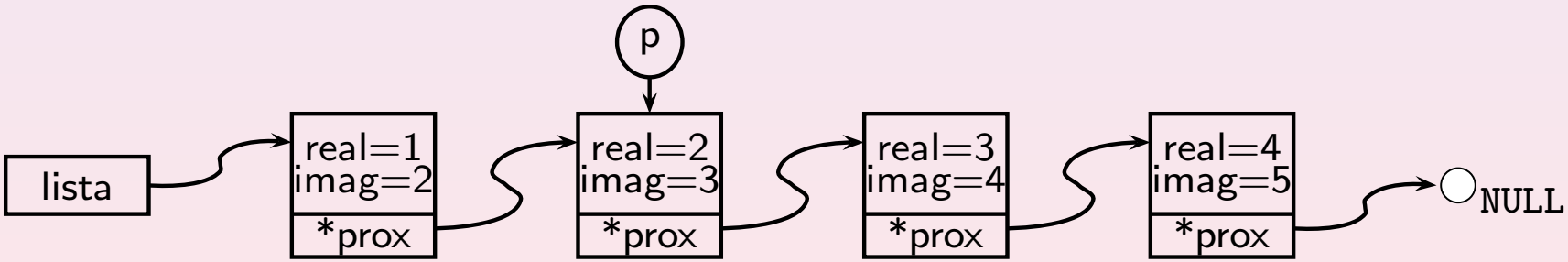
```
pos = 0 (1.000000 + i2.000000)
```



# Percurso na lista

```
p = lista; pos = 0;
while(p != NULL) {
    printf("pos = %d (%f + i%f)\n",pos,p->real,p->imag);
    p=p->prox; pos++;
}
```

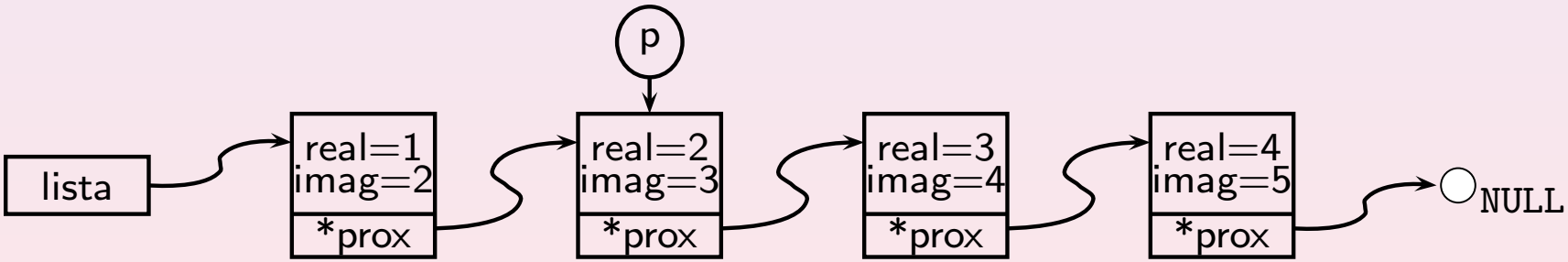
```
pos = 0 (1.000000 + i2.000000)
```



# Percurso na lista

```
p = lista; pos = 0;
while(p != NULL) {
    printf("pos = %d (%f + i%f)\n",pos,p->real,p->imag);
    p=p->prox; pos++;
}
```

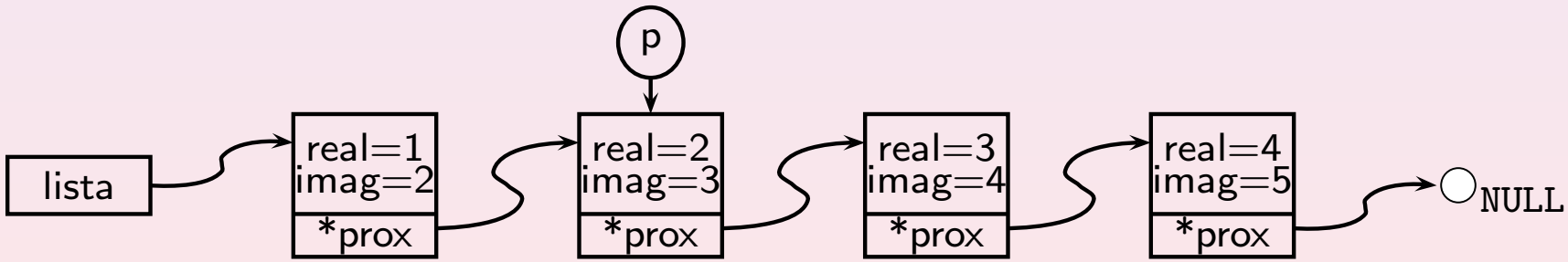
```
pos = 0 (1.000000 + i2.000000)
```



# Percurso na lista

```
p = lista; pos = 0;
while(p != NULL) {
    printf("pos = %d (%f + i%f)\n",pos,p->real,p->imag);
    p=p->prox; pos++;
}
```

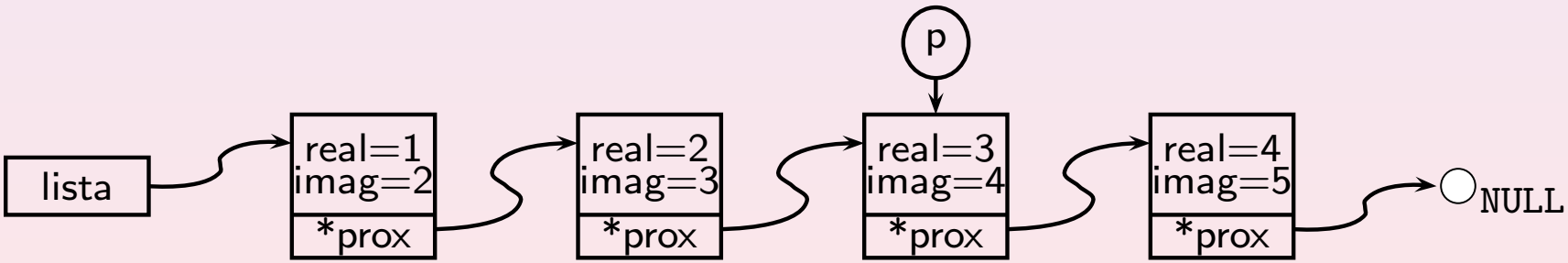
```
pos = 0 (1.000000 + i2.000000)
pos = 1 (2.000000 + i3.000000)
```



# Percurso na lista

```
p = lista; pos = 0;
while(p != NULL) {
    printf("pos = %d (%f + i%f)\n",pos,p->real,p->imag);
    p=p->prox; pos++;
}
```

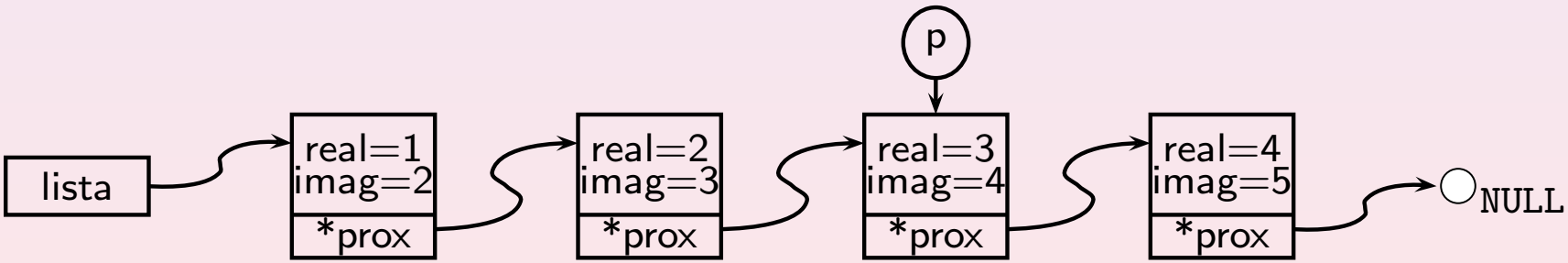
```
pos = 0 (1.000000 + i2.000000)
pos = 1 (2.000000 + i3.000000)
```



# Percurso na lista

```
p = lista; pos = 0;
while(p != NULL) {
    printf("pos = %d (%f + i%f)\n",pos,p->real,p->imag);
    p=p->prox; pos++;
}
```

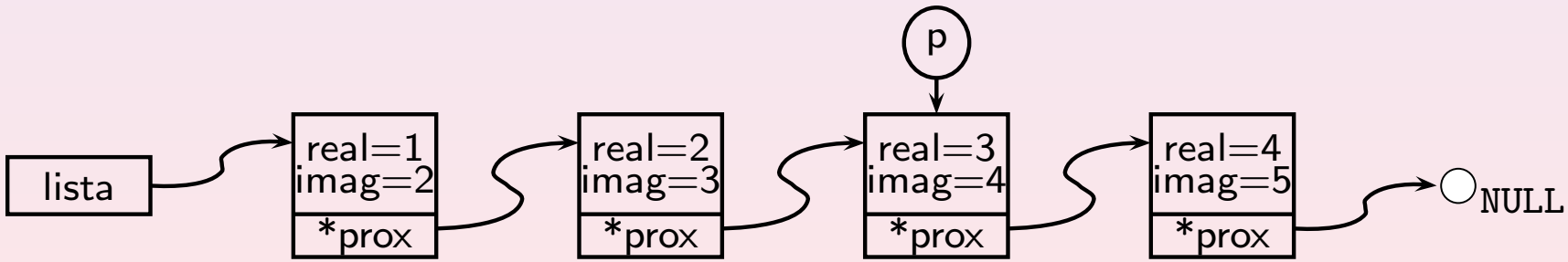
```
pos = 0 (1.000000 + i2.000000)
pos = 1 (2.000000 + i3.000000)
```



# Percurso na lista

```
p = lista; pos = 0;
while(p != NULL) {
    printf("pos = %d (%f + i%f)\n",pos,p->real,p->imag);
    p=p->prox; pos++;
}
```

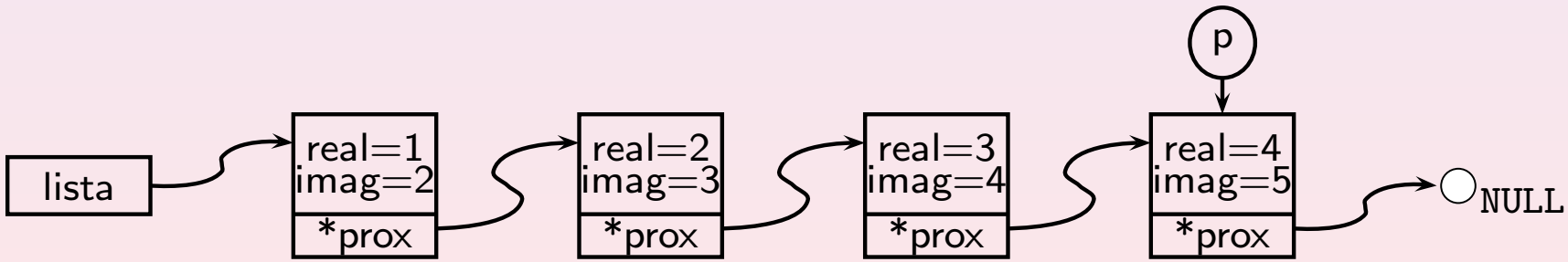
```
pos = 0 (1.000000 + i2.000000)
pos = 1 (2.000000 + i3.000000)
pos = 2 (3.000000 + i4.000000)
```



# Percurso na lista

```
p = lista; pos = 0;
while(p != NULL) {
    printf("pos = %d (%f + i%f)\n",pos,p->real,p->imag);
    p=p->prox; pos++;
}
```

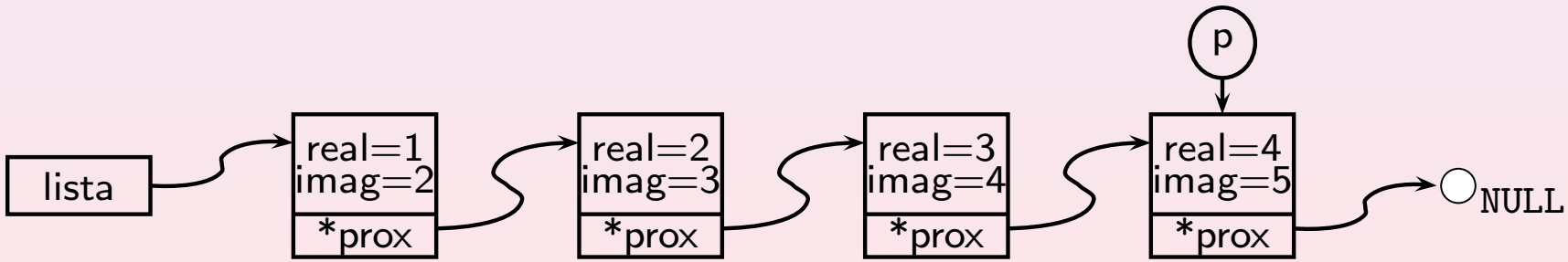
```
pos = 0 (1.000000 + i2.000000)
pos = 1 (2.000000 + i3.000000)
pos = 2 (3.000000 + i4.000000)
```



# Percurso na lista

```
p = lista; pos = 0;
while(p != NULL) {
    printf("pos = %d (%f + i%f)\n",pos,p->real,p->imag);
    p=p->prox; pos++;
}
```

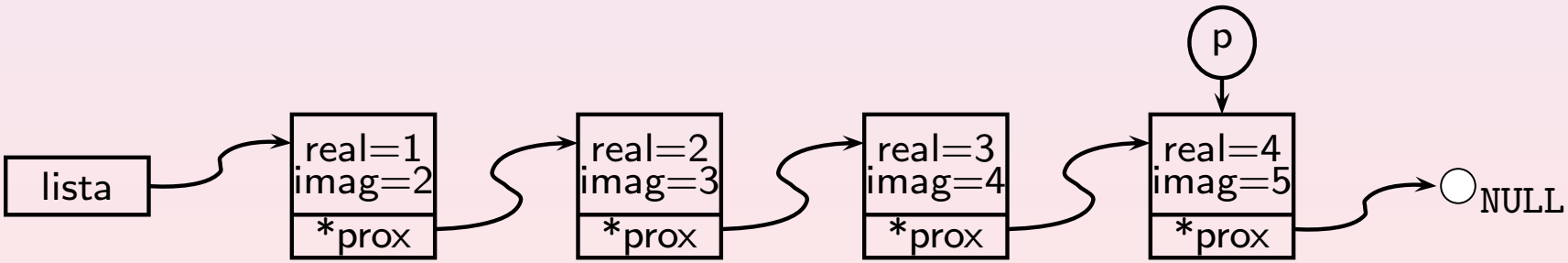
```
pos = 0 (1.000000 + i2.000000)
pos = 1 (2.000000 + i3.000000)
pos = 2 (3.000000 + i4.000000)
```



# Percurso na lista

```
p = lista; pos = 0;
while(p != NULL) {
    printf("pos = %d (%f + i%f)\n",pos,p->real,p->imag);
    p=p->prox; pos++;
}
```

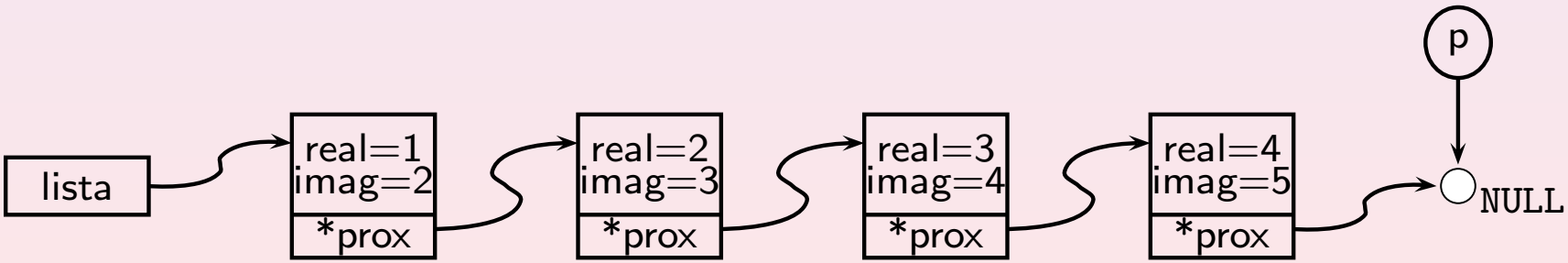
```
pos = 0 (1.000000 + i2.000000)
pos = 1 (2.000000 + i3.000000)
pos = 2 (3.000000 + i4.000000)
pos = 3 (4.000000 + i5.000000)
```



# Percurso na lista

```
p = lista; pos = 0;
while(p != NULL) {
    printf("pos = %d (%f + i%f)\n",pos,p->real,p->imag);
    p=p->prox; pos++;
}
```

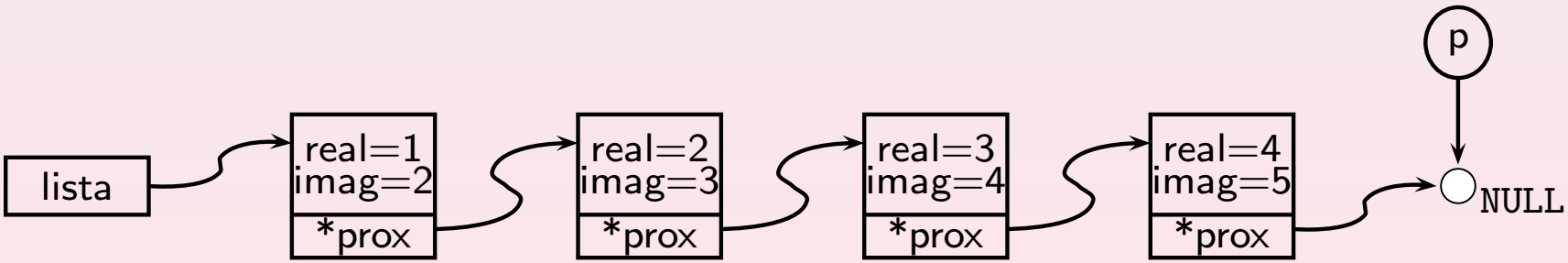
```
pos = 0 (1.000000 + i2.000000)
pos = 1 (2.000000 + i3.000000)
pos = 2 (3.000000 + i4.000000)
pos = 3 (4.000000 + i5.000000)
```



# Percurso na lista

```
p = lista; pos = 0;
while(p != NULL) {
    printf("pos = %d (%f + i%f)\n",pos,p->real,p->imag);
    p=p->prox; pos++;
}
```

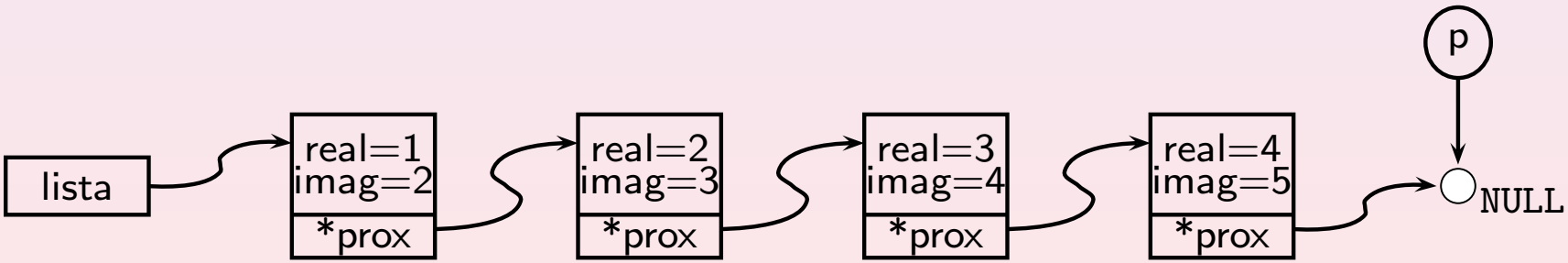
```
pos = 0 (1.000000 + i2.000000)
pos = 1 (2.000000 + i3.000000)
pos = 2 (3.000000 + i4.000000)
pos = 3 (4.000000 + i5.000000)
```



# Percurso na lista

```
p = lista; pos = 0;
while(p != NULL) {
    printf("pos = %d (%f + i%f)\n",pos,p->real,p->imag);
    p=p->prox; pos++;
}
```

```
pos = 0 (1.000000 + i2.000000)
pos = 1 (2.000000 + i3.000000)
pos = 2 (3.000000 + i4.000000)
pos = 3 (4.000000 + i5.000000)
```



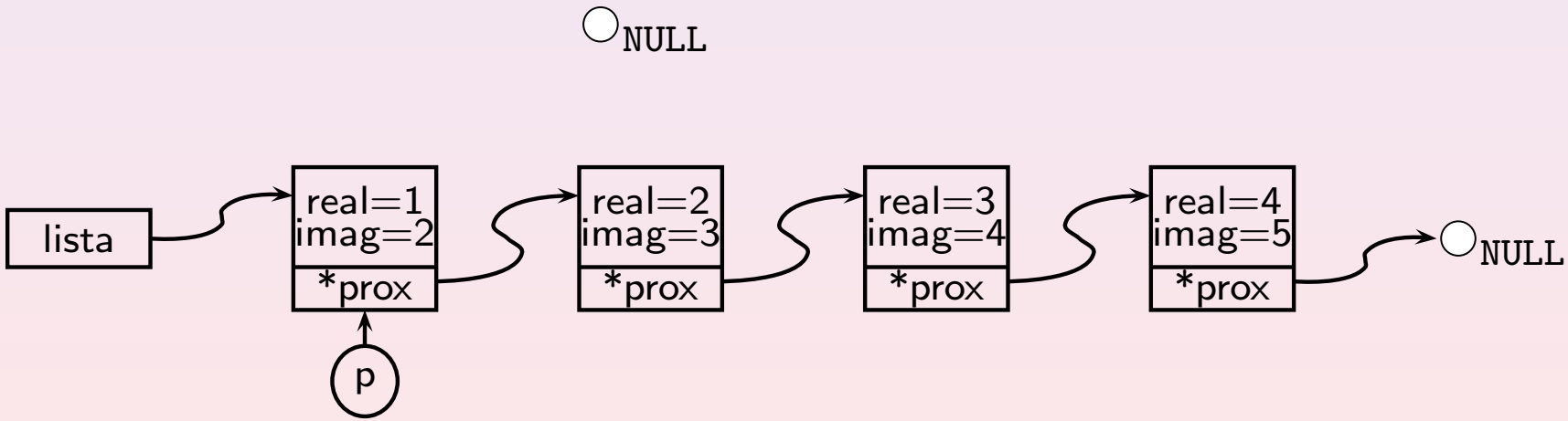
# Busca: realiza um percurso na lista

- A busca deve retornar o elemento da posição buscada.
- Se a posição for maior que o tamanho da lista, retorna NULL
- Não há um controle para o tamanho da lista.

```
complexo *busca(complexo *lista, int pos) {
 complexo *p = lista;
 int i = 0;
 while(i < pos && p != NULL) {
  p = p->prox; i++;
 }
 return p;
}
```

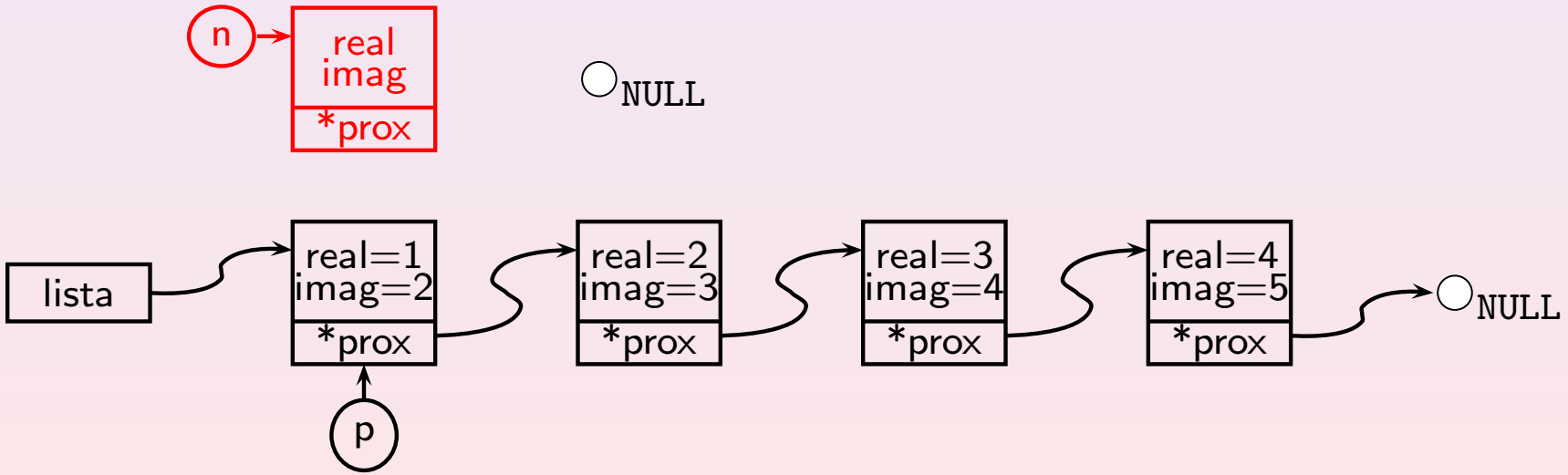
# Inserindo um elemento no início da lista

```
n = malloc(sizeof(complexo));
n->real = 5; n->imag = 6;
n->prox = NULL;
n->prox = p;
if(p == lista) {
    lista = n;
}
```



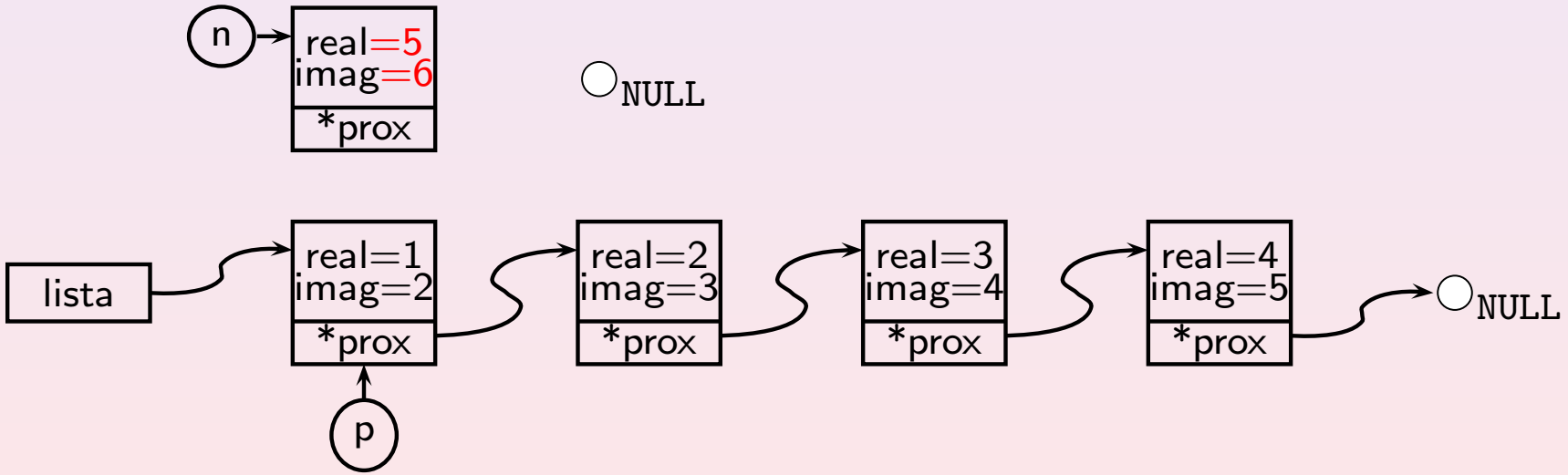
# Inserindo um elemento no início da lista

```
n = malloc(sizeof(complexo));
n->real = 5; n->imag = 6;
n->prox = NULL;
n->prox = p;
if(p == lista) {
    lista = n;
}
```



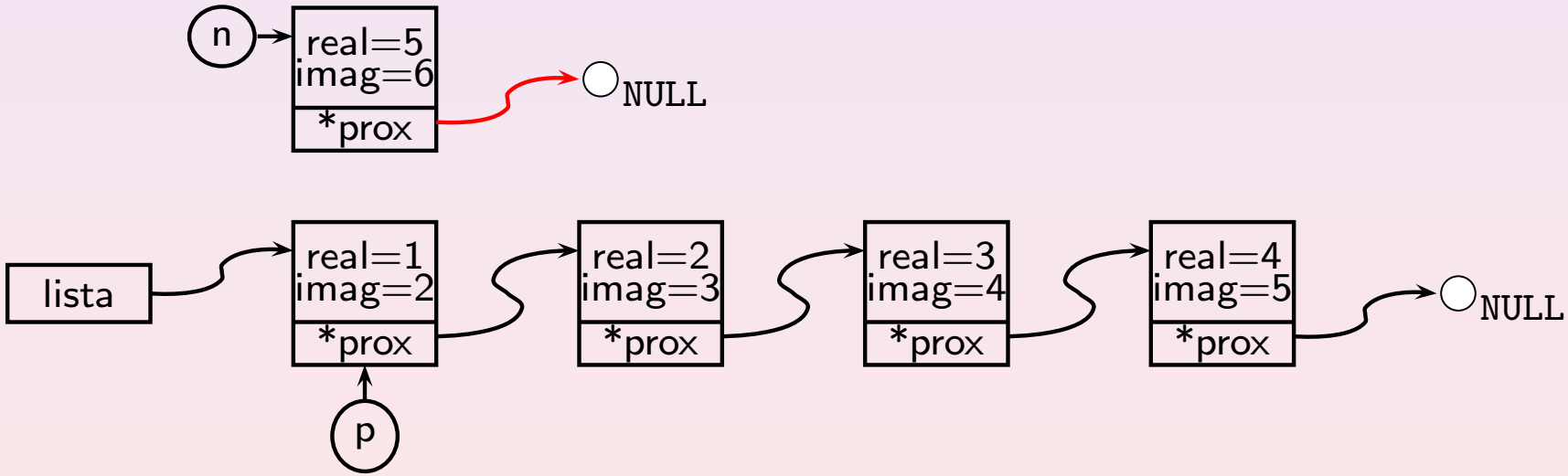
# Inserindo um elemento no início da lista

```
n = malloc(sizeof(complexo));
n->real = 5; n->imag = 6;
n->prox = NULL;
n->prox = p;
if(p == lista) {
    lista = n;
}
```



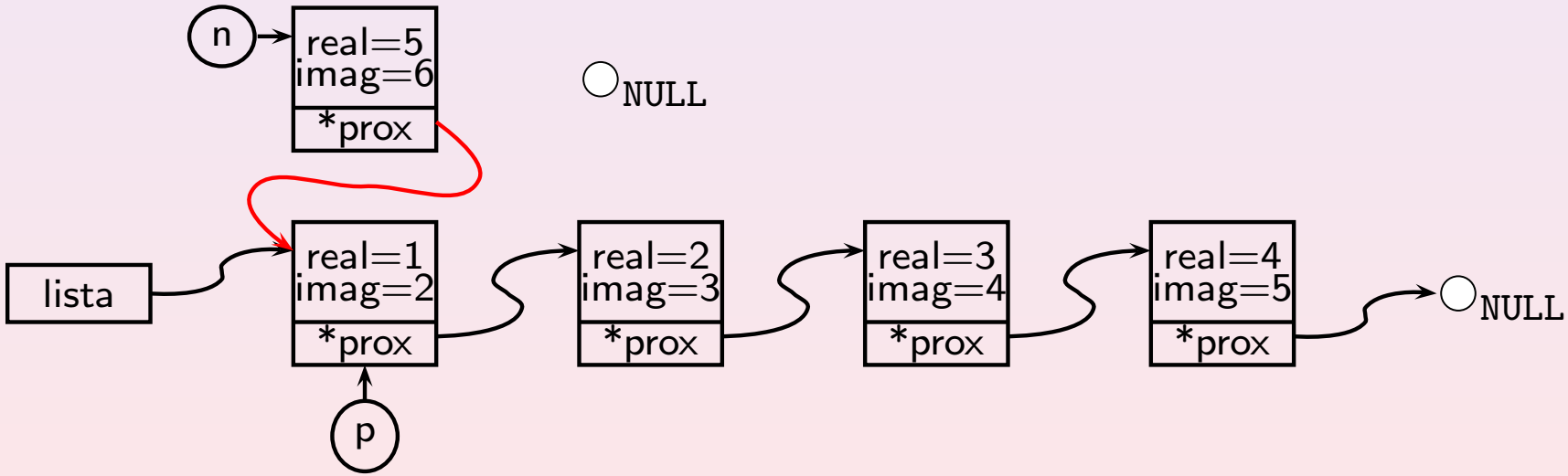
# Inserindo um elemento no início da lista

```
n = malloc(sizeof(complexo));
n->real = 5; n->imag = 6;
n->prox = NULL;
n->prox = p;
if(p == lista) {
    lista = n;
}
```



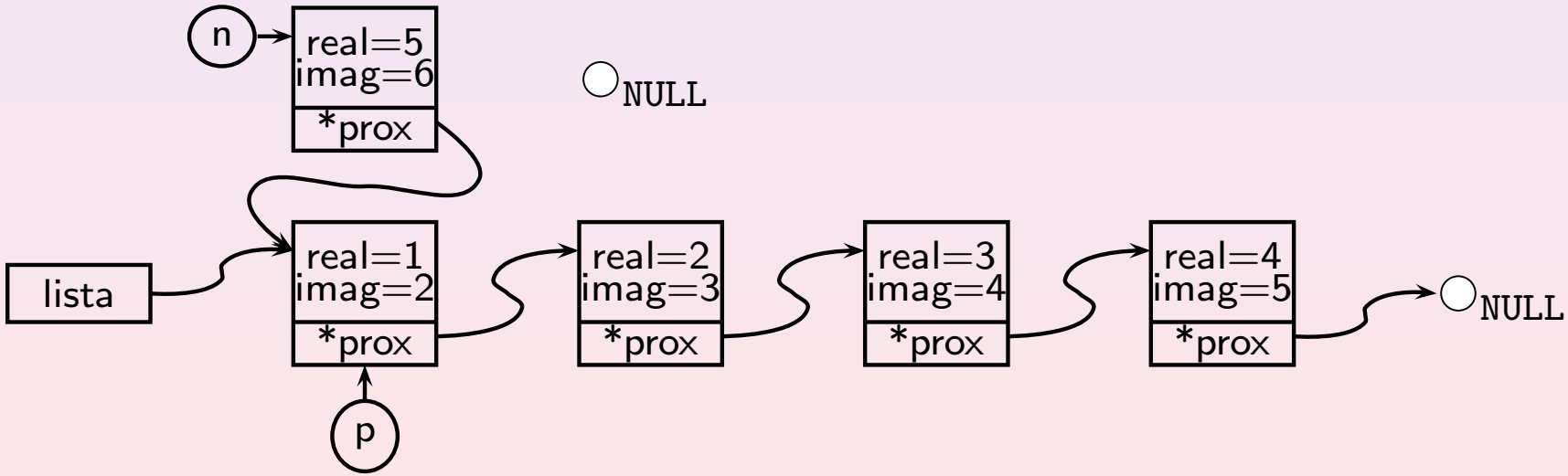
# Inserindo um elemento no início da lista

```
n = malloc(sizeof(complexo));
n->real = 5; n->imag = 6;
n->prox = NULL;
n->prox = p;
if(p == lista) {
    lista = n;
}
```



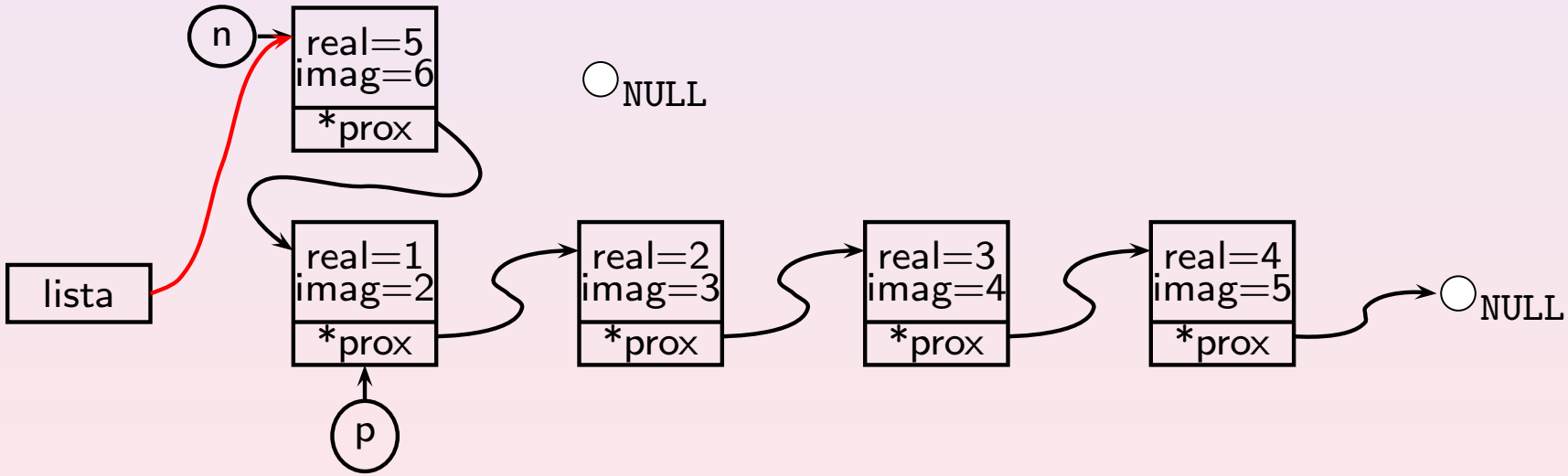
# Inserindo um elemento no início da lista

```
n = malloc(sizeof(complexo));
n->real = 5; n->imag = 6;
n->prox = NULL;
n->prox = p;
if(p == lista) {
    lista = n;
}
```



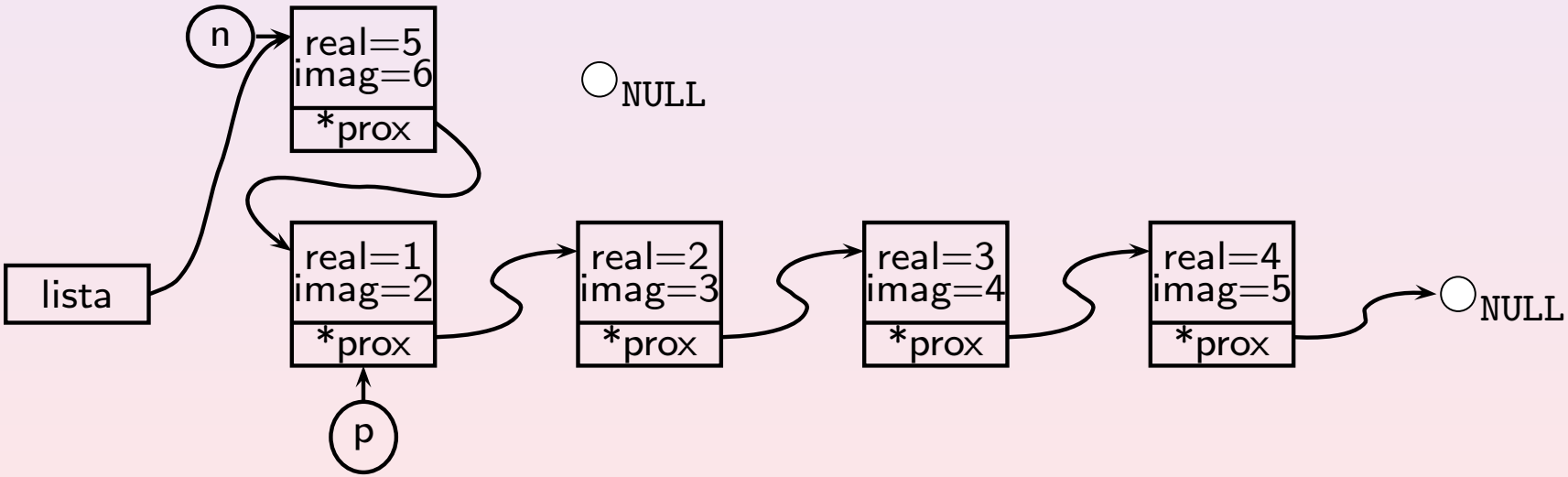
# Inserindo um elemento no início da lista

```
n = malloc(sizeof(complexo));
n->real = 5; n->imag = 6;
n->prox = NULL;
n->prox = p;
if(p == lista) {
    lista = n;
}
```



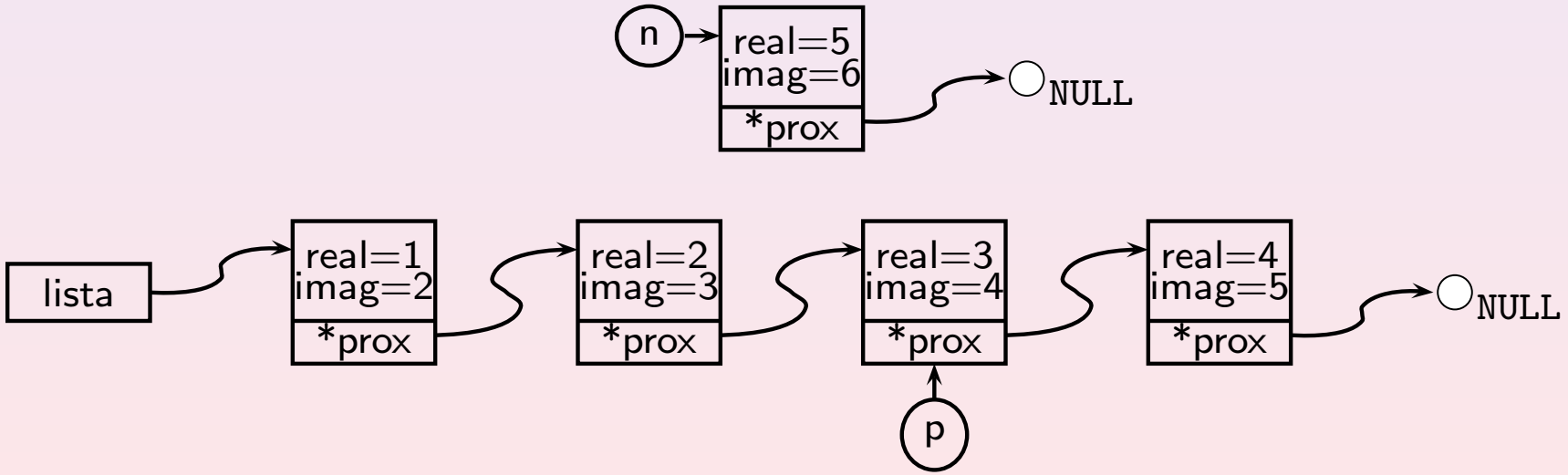
# Inserindo um elemento no início da lista

```
n = malloc(sizeof(complexo));
n->real = 5; n->imag = 6;
n->prox = NULL;
n->prox = p;
if(p == lista) {
    lista = n;
}
```



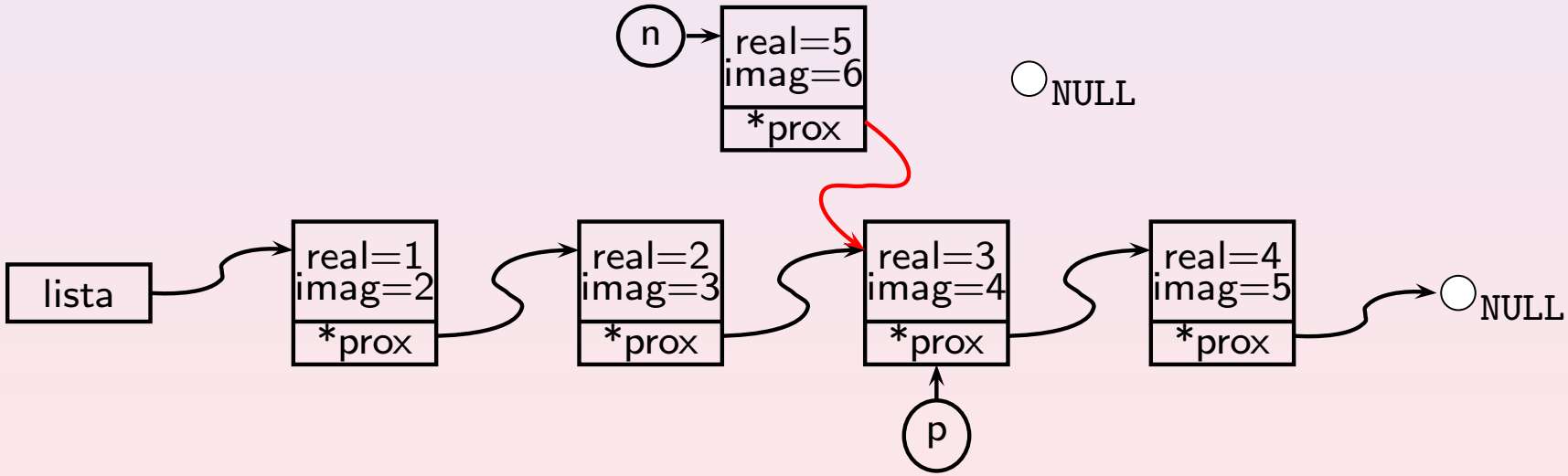
# Inserindo elemento do meio pro fim da lista

```
n->prox = p;
if(p == lista) {
    lista = n;
} else {
    a = lista;
    while(a->prox != p) a = a->prox;
    a->prox = n;
}
```



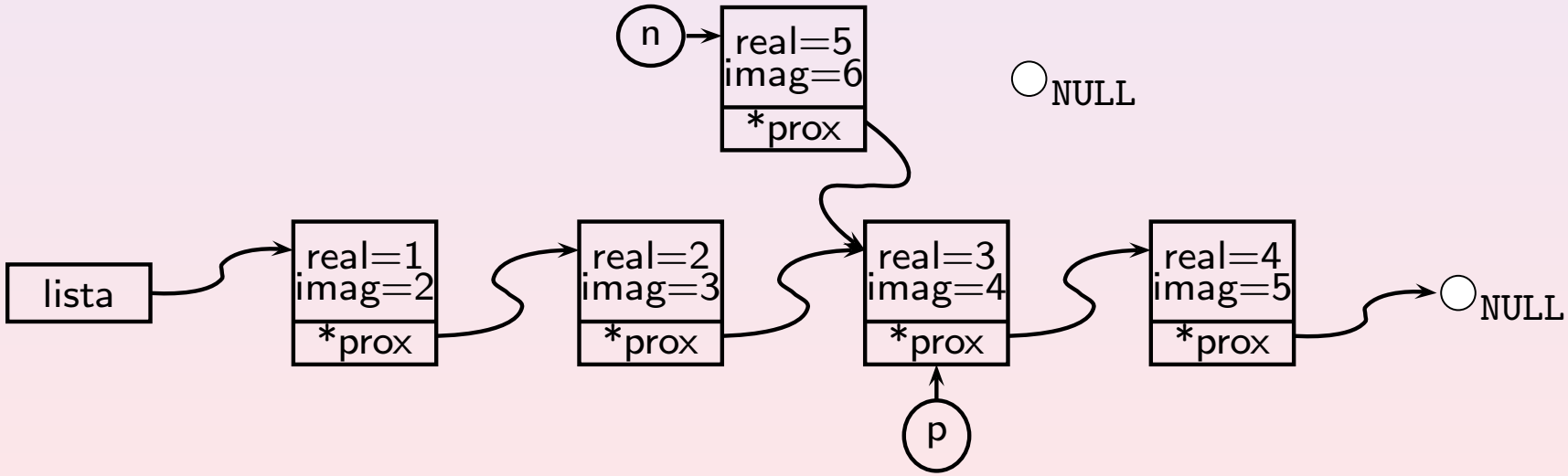
# Inserindo elemento do meio pro fim da lista

```
n->prox = p;
if(p == lista) {
    lista = n;
} else {
    a = lista;
    while(a->prox != p) a = a->prox;
    a->prox = n;
}
```



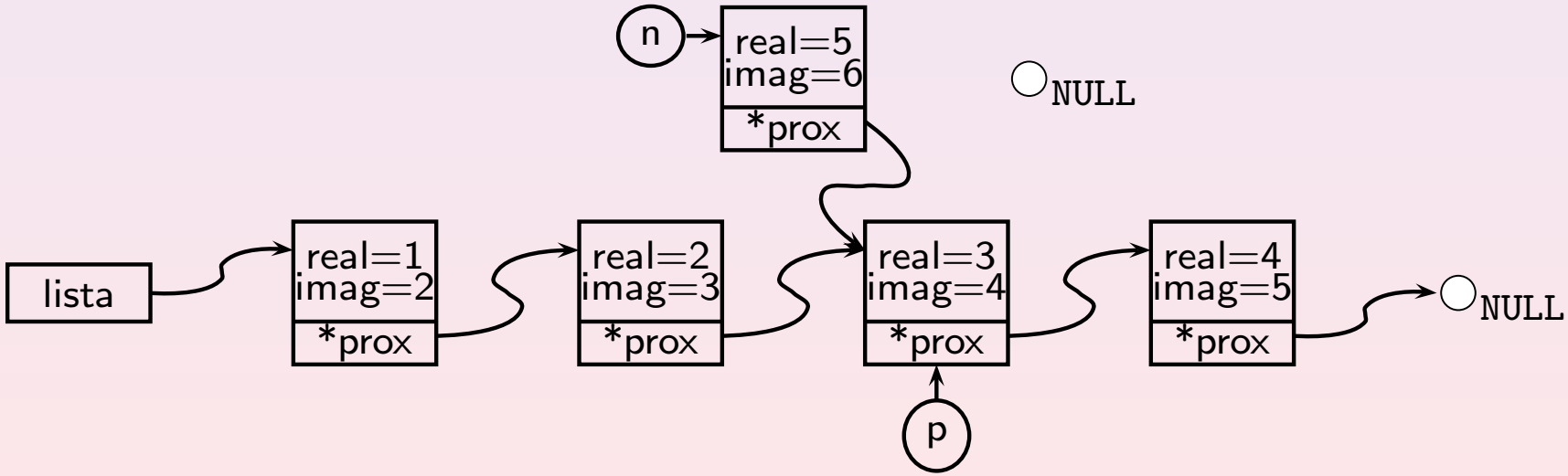
# Inserindo elemento do meio pro fim da lista

```
n->prox = p;
if(p == lista) {
    lista = n;
} else {
    a = lista;
    while(a->prox != p) a = a->prox;
    a->prox = n;
}
```



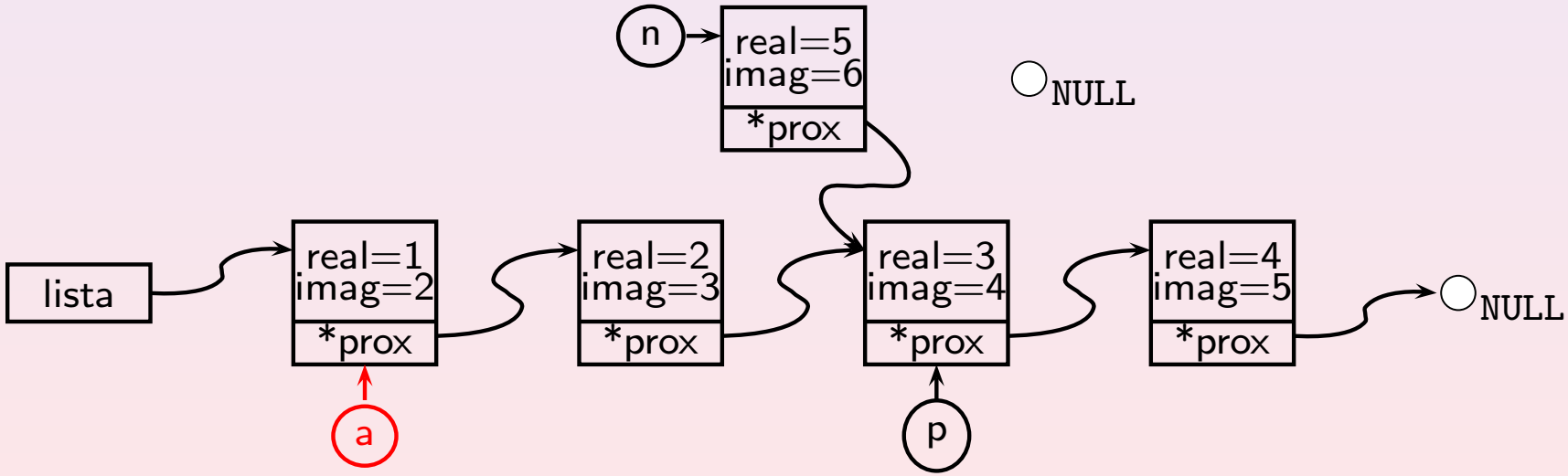
# Inserindo elemento do meio pro fim da lista

```
n->prox = p;
if(p == lista) {
    lista = n;
} else {
    a = lista;
    while(a->prox != p) a = a->prox;
    a->prox = n;
}
```



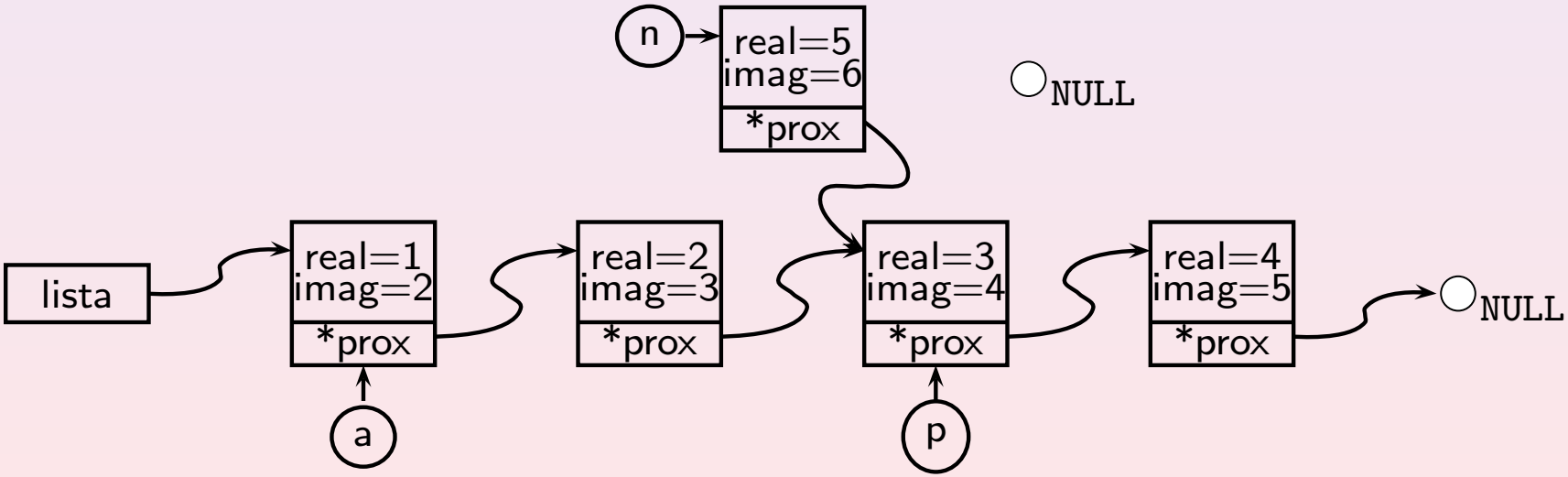
# Inserindo elemento do meio pro fim da lista

```
n->prox = p;
if(p == lista) {
    lista = n;
} else {
    a = lista;
    while(a->prox != p) a = a->prox;
    a->prox = n;
}
```



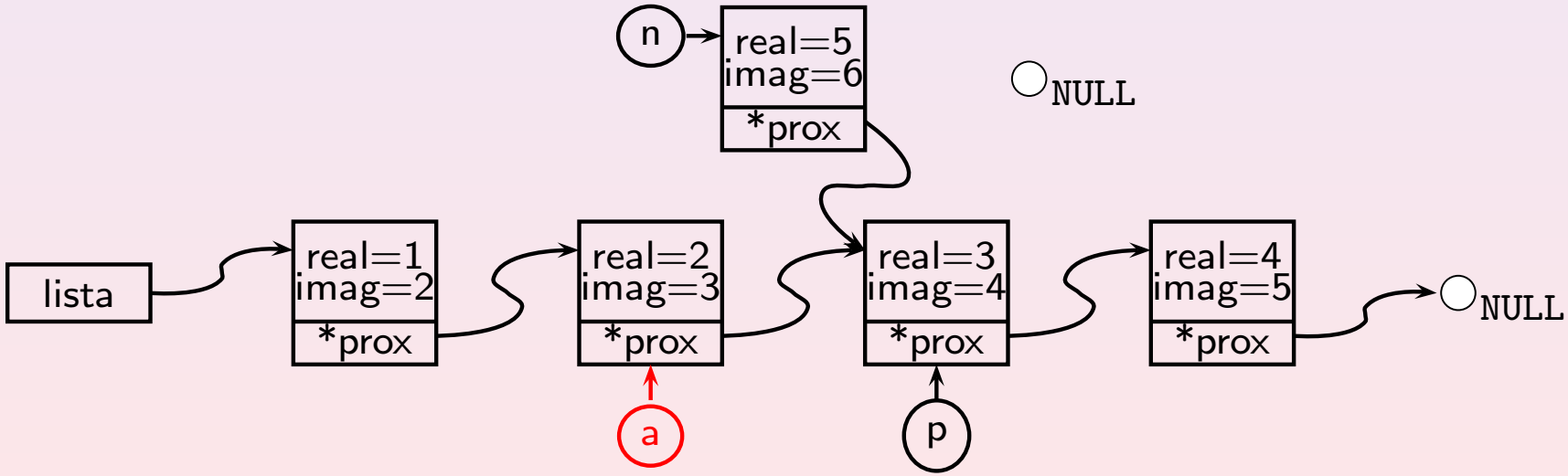
# Inserindo elemento do meio pro fim da lista

```
n->prox = p;
if(p == lista) {
    lista = n;
} else {
    a = lista;
    while(a->prox != p) a = a->prox;
    a->prox = n;
}
```



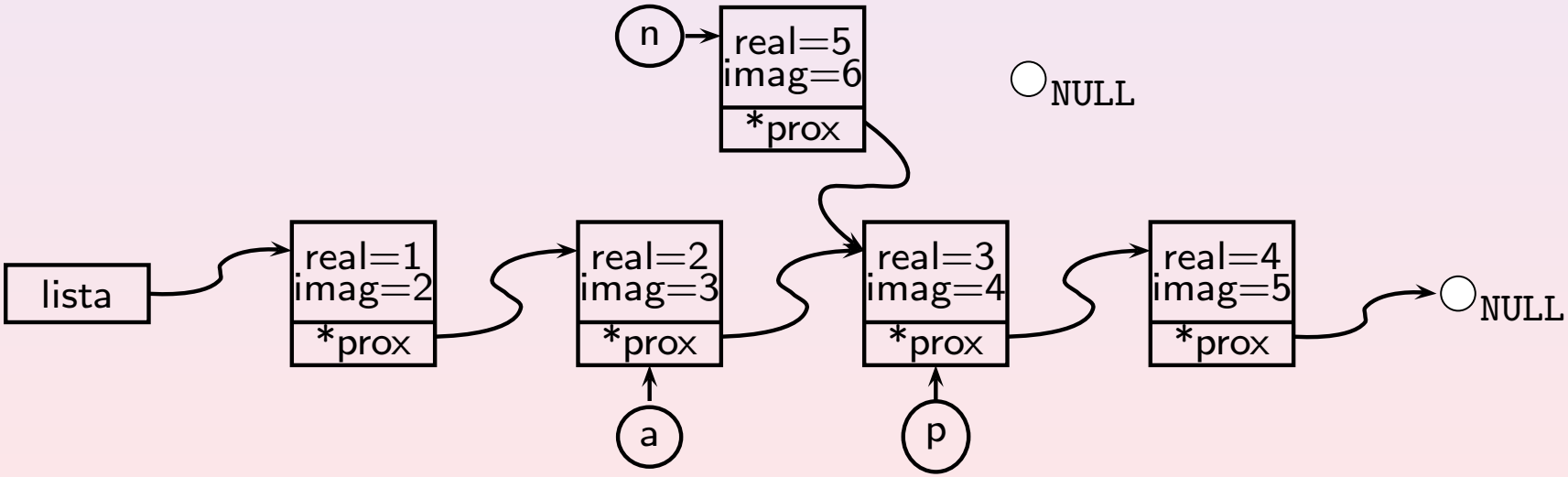
# Inserindo elemento do meio pro fim da lista

```
n->prox = p;
if(p == lista) {
    lista = n;
} else {
    a = lista;
    while(a->prox != p) a = a->prox;
    a->prox = n;
}
```



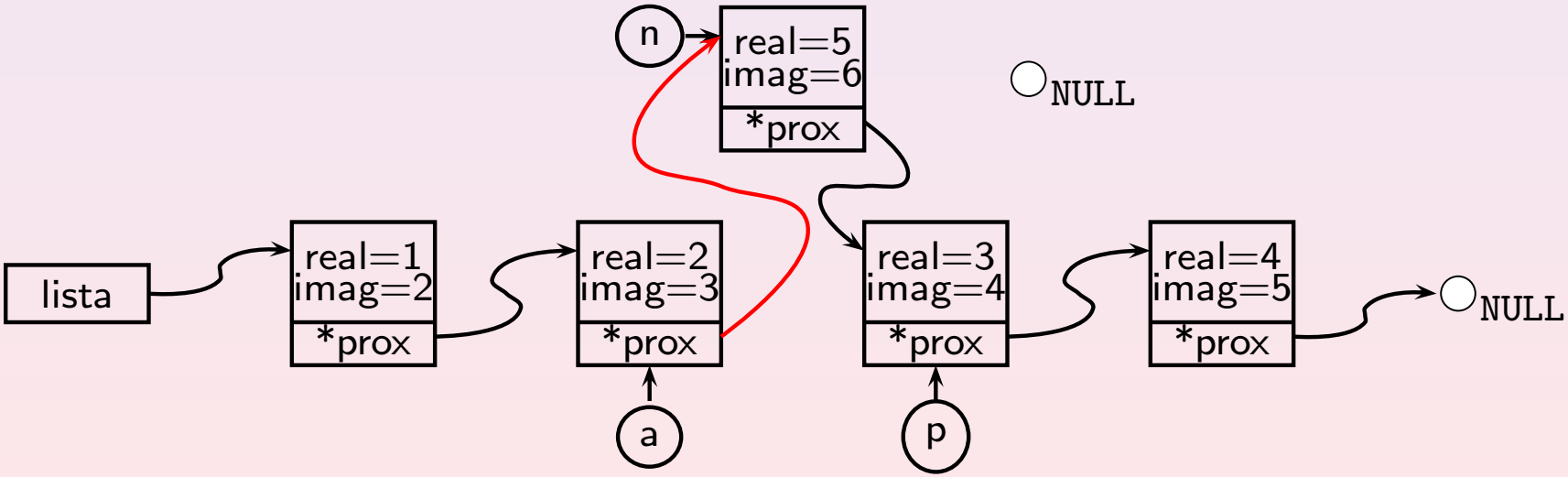
# Inserindo elemento do meio pro fim da lista

```
n->prox = p;
if(p == lista) {
    lista = n;
} else {
    a = lista;
    while(a->prox != p) a = a->prox;
    a->prox = n;
}
```



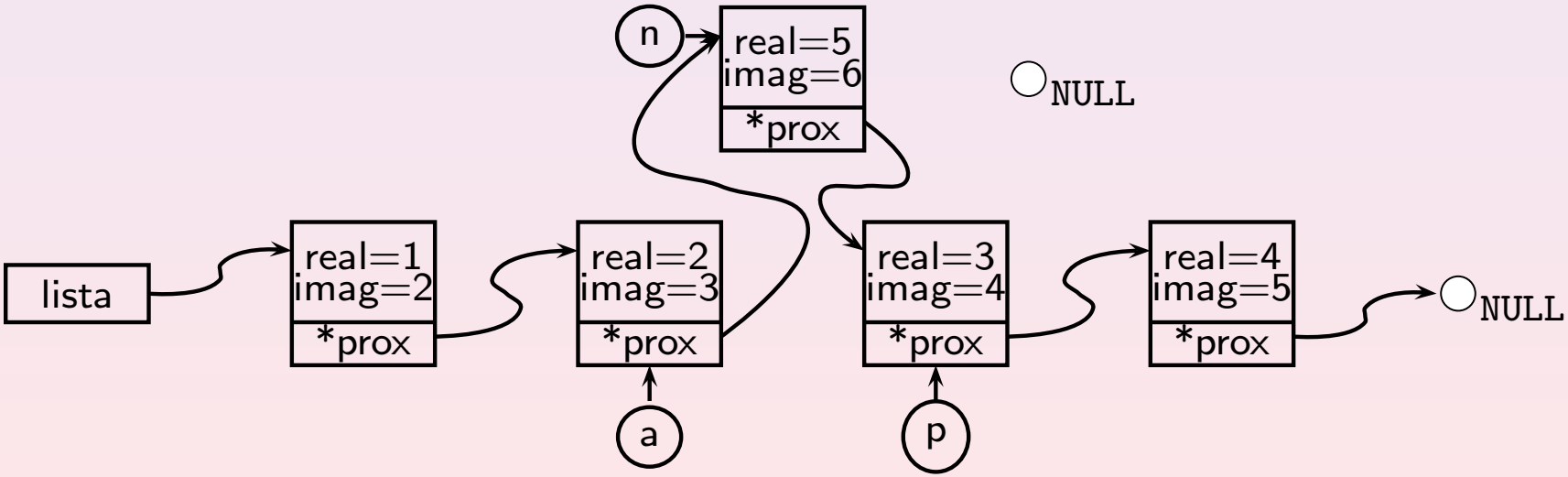
# Inserindo elemento do meio pro fim da lista

```
n->prox = p;
if(p == lista) {
    lista = n;
} else {
    a = lista;
    while(a->prox != p) a = a->prox;
    a->prox = n;
}
```



# Inserindo elemento do meio pro fim da lista

```
n->prox = p;
if(p == lista) {
    lista = n;
} else {
    a = lista;
    while(a->prox != p) a = a->prox;
    a->prox = n;
}
```



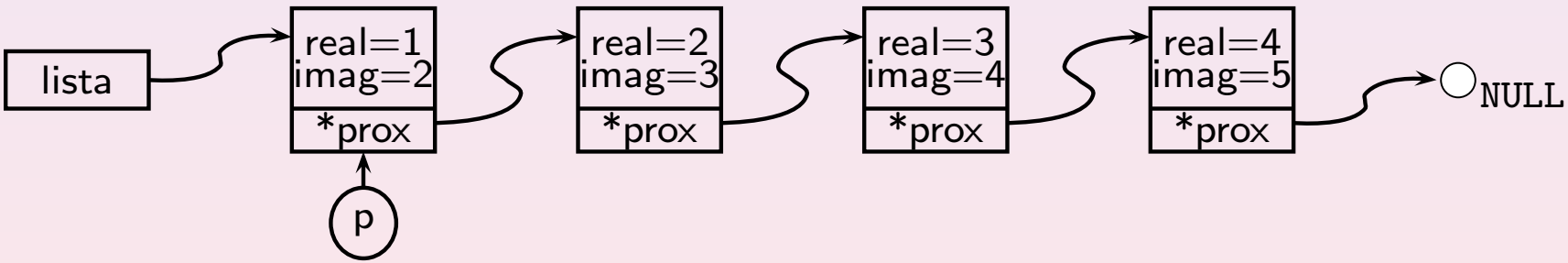
# Inserir: insere um elemento em uma posição da lista

- Se a lista não existe o elemento é a lista.
- Se a posição é após o final da lista, o elemento entra na última posição
- A função retorna a lista, para poder atualizar o cabeça.

```
complexo *inserir(complexo *lista, complexo *elemento, int pos) {
  complexo *a, *p;
  if(lista) {
    p = busca(lista,pos);
    elemento->prox = p;
    if(p == lista) {
      lista = elemento;
    } else {
      a = lista;
      while(a->prox != p) a = a->prox;
      a->prox = elemento;
    }
  } else {
    lista = elemento;
  }
  return lista;
}
```

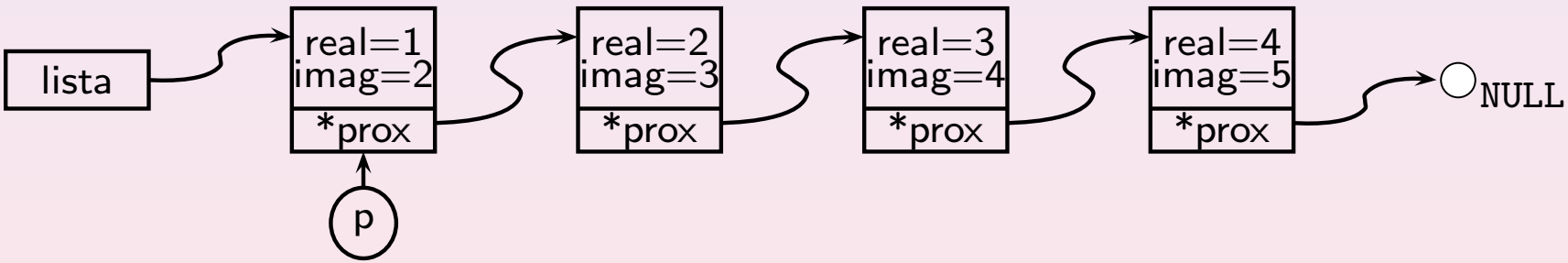
# Removendo o primeiro elemento da lista

```
if(p != NULL) {
    if(p == lista) {
        lista = p->prox;
    }
} free(p);
```



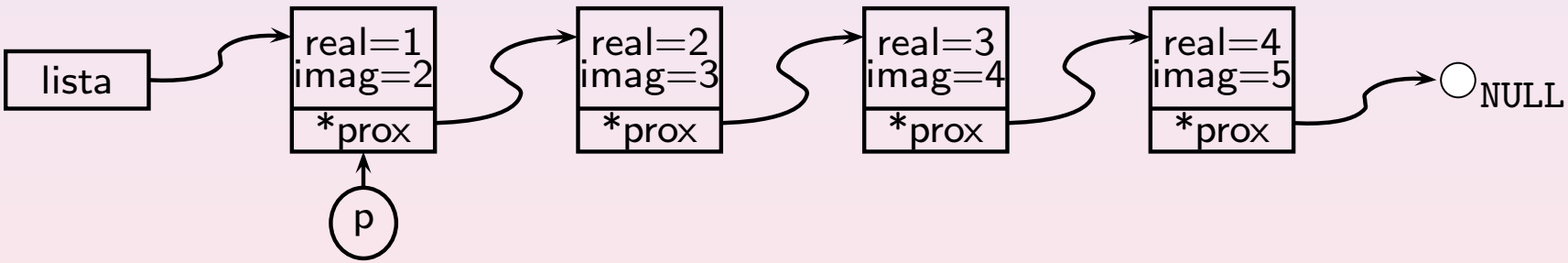
# Removendo o primeiro elemento da lista

```
if(p != NULL) {
    if(p == lista) {
        lista = p->prox;
    }
} free(p);
```



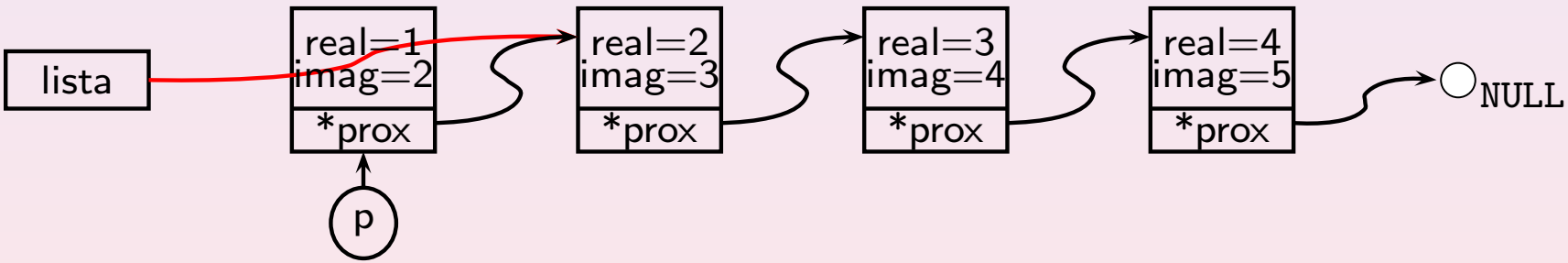
# Removendo o primeiro elemento da lista

```
if(p != NULL) {
    if(p == lista) {
        lista = p->prox;
    }
} free(p);
```



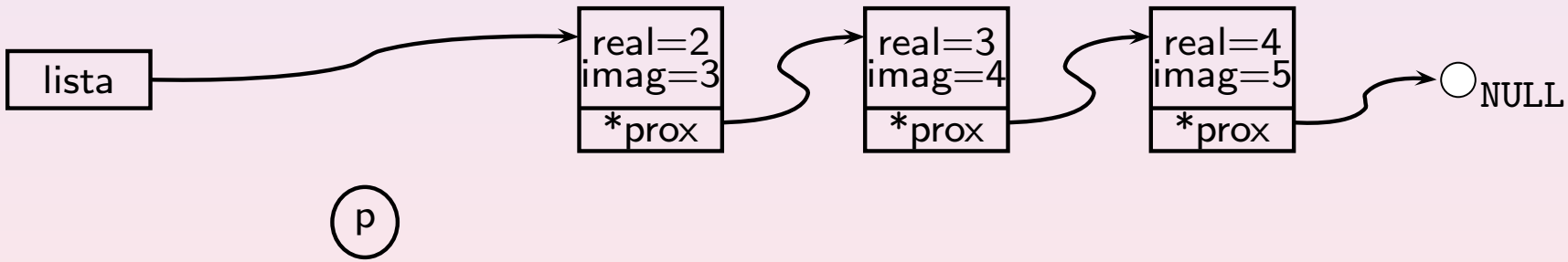
# Removendo o primeiro elemento da lista

```
if(p != NULL) {
    if(p == lista) {
        lista = p->prox;
    }
} free(p);
```



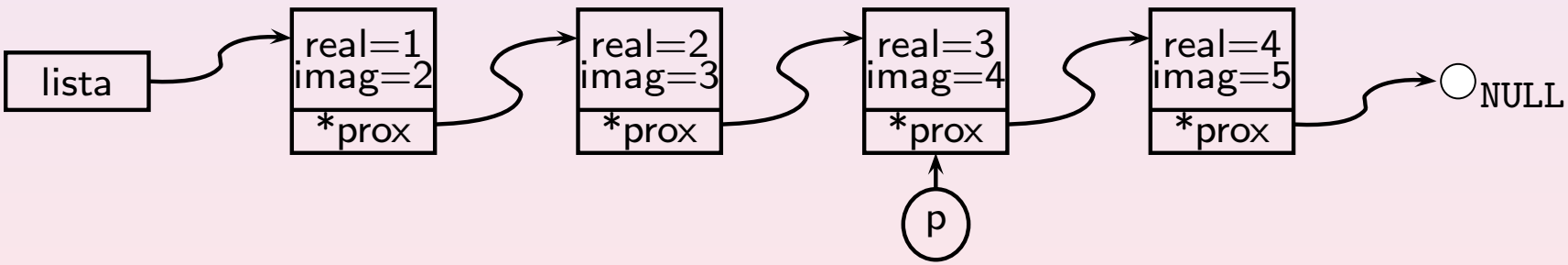
# Removendo o primeiro elemento da lista

```
if(p != NULL) {
    if(p == lista) {
        lista = p->prox;
    }
} free(p);
```



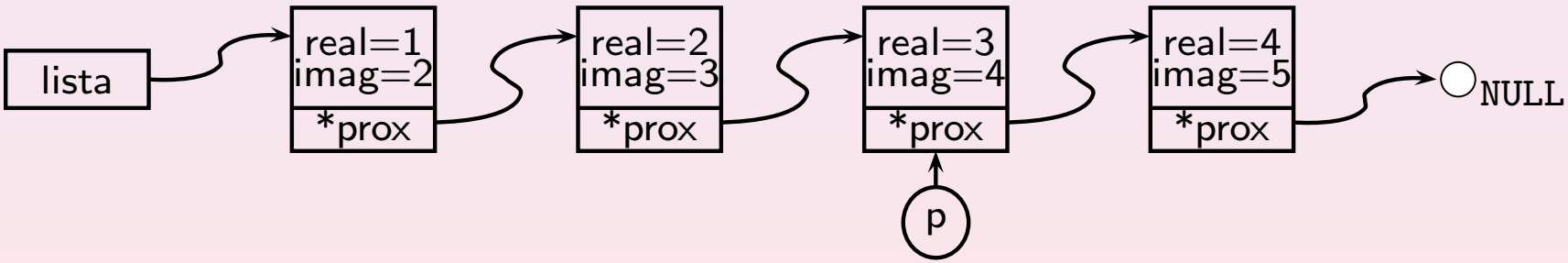
# Removendo um elemento do meio pro fim

```
if(p != NULL) {
    if(p == lista) {
        lista = p->prox;
    } else {
        a = lista;
        while(a->prox != p)  a = a->prox;
        a->prox = p->prox;
    }
} free(p);
```



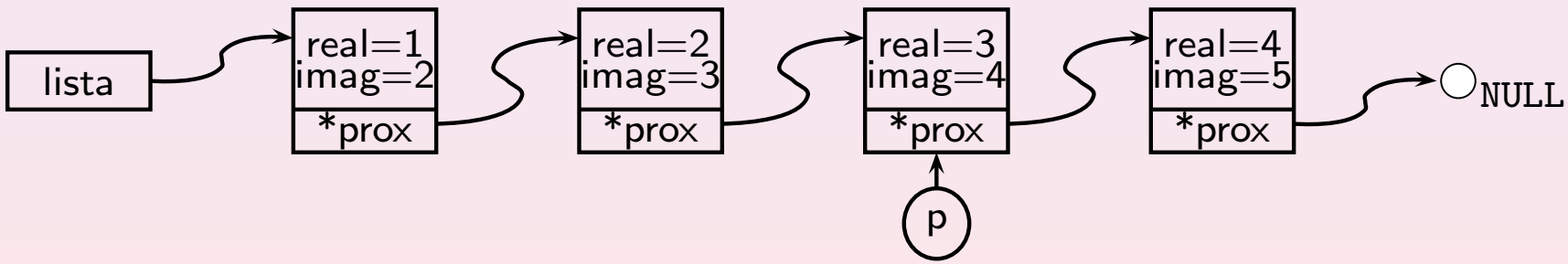
# Removendo um elemento do meio pro fim

```
if(p != NULL) {
    if(p == lista) {
        lista = p->prox;
    } else {
        a = lista;
        while(a->prox != p)  a = a->prox;
        a->prox = p->prox;
    }
} free(p);
```



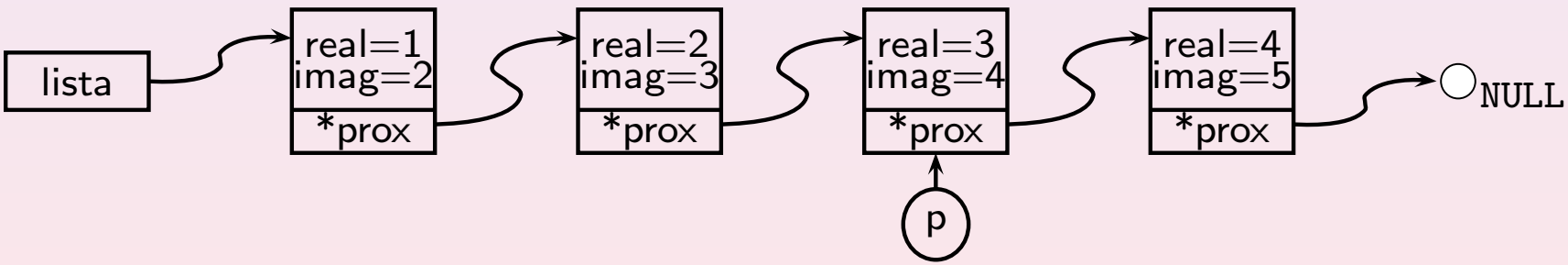
# Removendo um elemento do meio pro fim

```
if(p != NULL) {
    if(p == lista) {
        lista = p->prox;
    } else {
        a = lista;
        while(a->prox != p)  a = a->prox;
        a->prox = p->prox;
    }
} free(p);
```



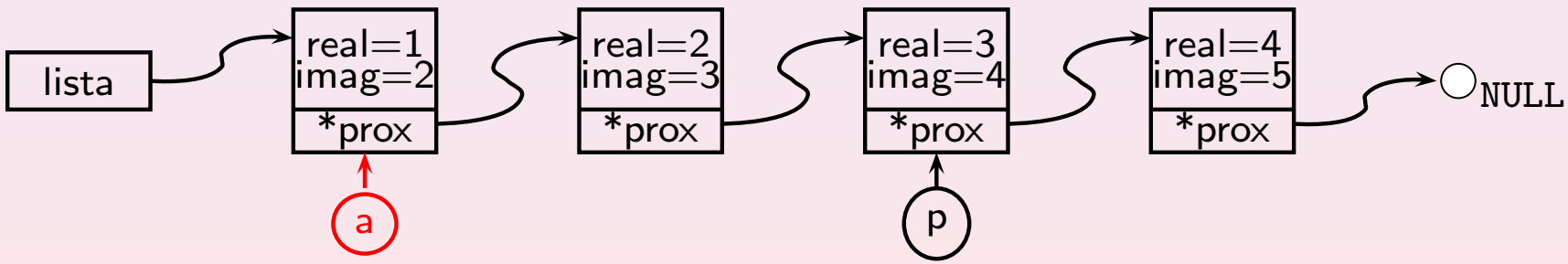
# Removendo um elemento do meio pro fim

```
if(p != NULL) {
    if(p == lista) {
        lista = p->prox;
    } else {
        a = lista;
        while(a->prox != p)  a = a->prox;
        a->prox = p->prox;
    }
} free(p);
```



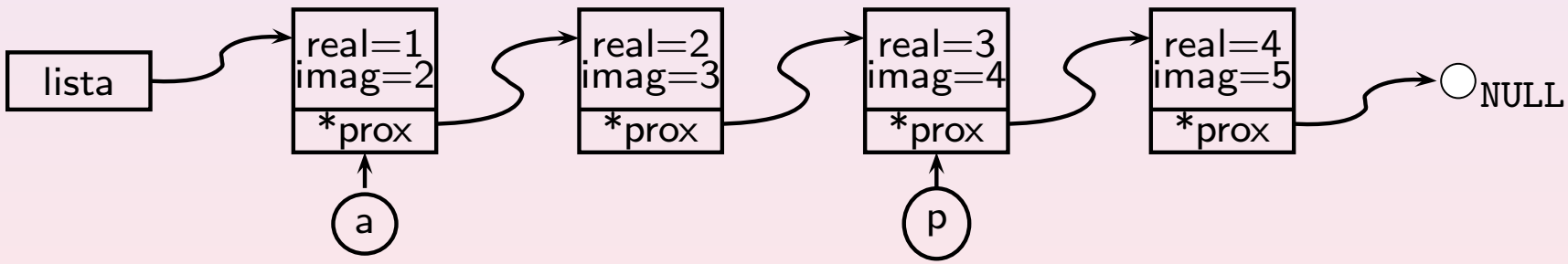
# Removendo um elemento do meio pro fim

```
if(p != NULL) {
    if(p == lista) {
        lista = p->prox;
    } else {
        a = lista;
        while(a->prox != p)  a = a->prox;
        a->prox = p->prox;
    }
} free(p);
```



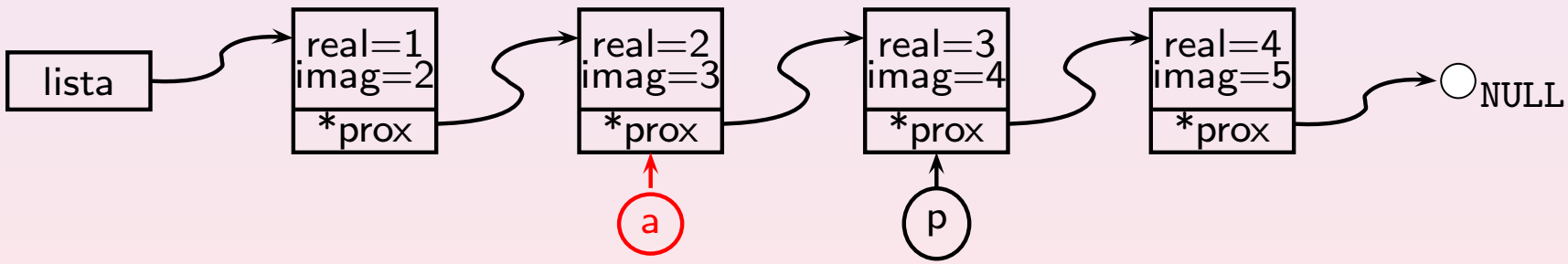
# Removendo um elemento do meio pro fim

```
if(p != NULL) {
    if(p == lista) {
        lista = p->prox;
    } else {
        a = lista;
        while(a->prox != p)  a = a->prox;
        a->prox = p->prox;
    }
} free(p);
```



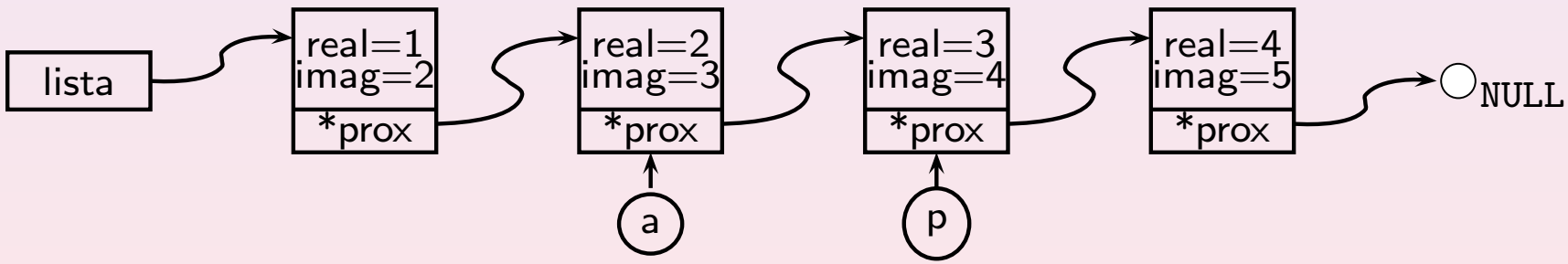
# Removendo um elemento do meio pro fim

```
if(p != NULL) {
    if(p == lista) {
        lista = p->prox;
    } else {
        a = lista;
        while(a->prox != p)  a = a->prox;
        a->prox = p->prox;
    }
} free(p);
```



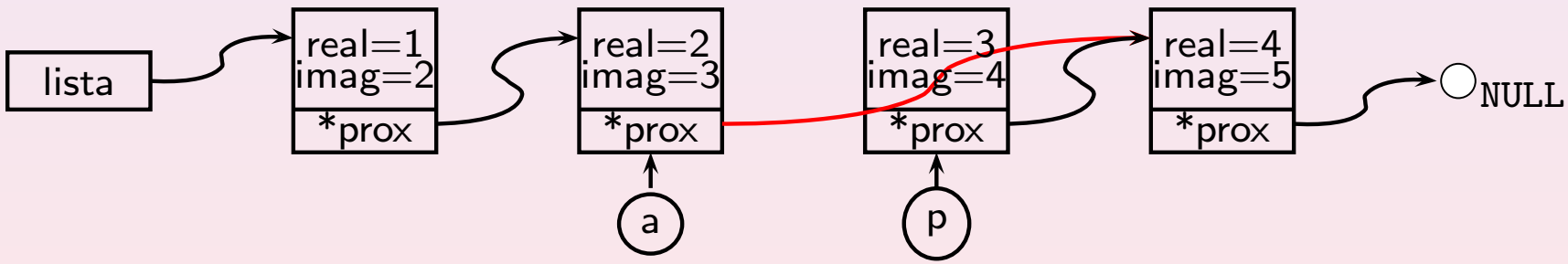
# Removendo um elemento do meio pro fim

```
if(p != NULL) {
    if(p == lista) {
        lista = p->prox;
    } else {
        a = lista;
        while(a->prox != p)  a = a->prox;
        a->prox = p->prox;
    }
} free(p);
```



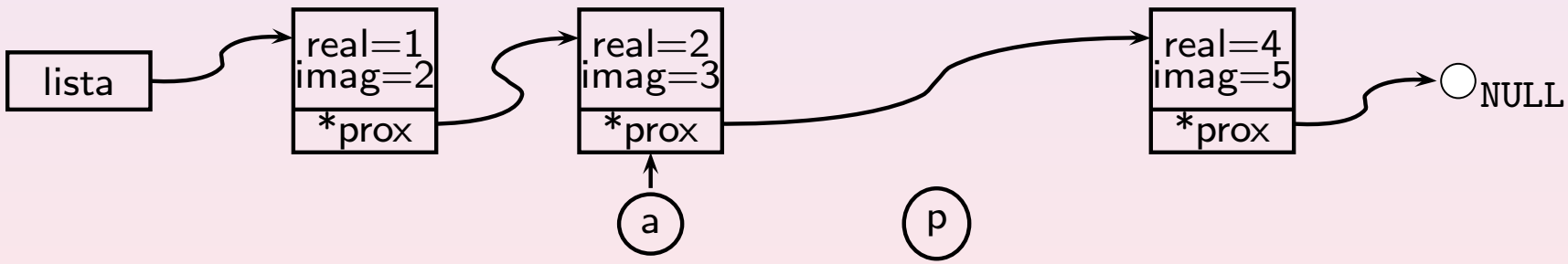
# Removendo um elemento do meio pro fim

```
if(p != NULL) {
    if(p == lista) {
        lista = p->prox;
    } else {
        a = lista;
        while(a->prox != p)  a = a->prox;
        a->prox = p->prox;
    }
} free(p);
```



# Removendo um elemento do meio pro fim

```
if(p != NULL) {
    if(p == lista) {
        lista = p->prox;
    } else {
        a = lista;
        while(a->prox != p)  a = a->prox;
        a->prox = p->prox;
    }
} free(p);
```



# Remover: remove um elemento da lista

- A posição precisa ter um elemento para ser removido
- A função retorna 1 (true) se removeu ou 0 (false) caso contrário
- A lista é atualizada pois foi passada como parâmetro.

```
complexo *remover(complexo *lista, int pos) {
  complexo *p, *a;
  p = busca(lista,pos);
  if(p) {
    if(p == lista) {
      lista = p->prox;
    } else {
      a = lista;
      while(a->prox!=p) a = a->prox;
      a->prox = p->prox;
    }
    free(p);
  }
  return lista;
}
```

# Trocando dois elementos

- Vamos considerar duas posições: p1 e p2, sendo p2 maior que p1.
- A troca se dá pelos seguintes passos:
  1. Remover p2, guardando o elemento.
  2. Inserir o elemento p2 na posição p1.
  3. Remover p1, guardando o elemento.
  4. Inserir o elemento p1 na posição p2.
- Poderíamos fazer a troca só redirecionando ponteiros sem remover e inserir, o que economizaria ações.
- Apesar de envolver um conjunto maior de ações, as operações envolvidas ficam mais simples, sem precisar verificar inúmeras situações: p1 é o primeiro? p2 é o último? p1 é antecessor direto de p2?

# Trocando dois elementos

```
complexo *trocar(complexo *lista, int posA, int posB) {
  complexo *pb, *pa, *a;
  if(posA > posB) {
    posA ^= posB;
    posB ^= posA;
    posA ^= posB;
  }
  pb = busca(lista,posB);
  if(pb && posB > posA) {
    a = lista;
    while(a->prox!=pb) a = a->prox;
    a->prox = pb->prox;
    lista = inserir(lista,pb,posA);
    pa = pb->prox;
    pb->prox = pa->prox;
    lista = inserir(lista,pa,posB);
  }
  return lista;
}
```

# Colocando tudo junto

```
int main() {
complexo *lista = NULL, *elemento = NULL;
elemento = criar(1,2);
lista = inserir(lista,elemento,0);
elemento = criar(2,3);
lista = inserir(lista,elemento,3); imprimir(lista);
elemento = criar(3,4);
lista = inserir(lista,elemento,1);
elemento = criar(4,5);
lista = inserir(lista,elemento,0); imprimir(lista);
lista = remover(lista,3); imprimir(lista);
lista = trocar(lista,0,2); imprimir(lista);
return 0;
}
```

```
1+i2..2+i3..
4+i5..1+i2..3+i4..2+i3..
4+i5..1+i2..3+i4..
3+i4..1+i2..4+i5..
```

# Qual o custo das operações?

- Busca: $O(n)$
- Inserir: $O(n)$
- Remover: $O(n)$
- Trocar: $O(n)$

# Estrutura de dados: Listas

- Quando falamos de uma estrutura de dados, a estrutura representa um conjunto de dados e operações para manipulação destes dados.
- No caso de Listas (como estrutura) as operações são: `Buscar`, `Inserir` e `Remover`.
- Podíamos ter outras operações como (`ehVazia`, `ehCheia`, `Trocar`, ...)
- Como organizar o conjunto de dados?
  - Poderia ser um vetor...
  - Poderia ser uma lista encadeada...
- Precisamos então distinguir a Estrutura, em si, de seu conjunto de dados.

# Estrutura de dados: Listas

- Revendo nossa implementação em C, observamos que:
  - A lista se confunde com o conteúdo.
  - Se a lista está vazia a estrutura some, cabe ao programador controlar a existência da lista.
- Revendo nossa implementação em C++, observamos que:
  - O conteúdo é distinto da Lista, a lista contém conteúdos.
  - Porém a lista se confunde com um item da lista.
  - A lista deveria ter a informação de "Cabeca", porém cada item contém esta informação.
  - Cabeca em si é também um item (possui conteúdo, mesmo que não inciado), e não um apontador especial

# Estrutura de dados: Listas

- Precisamos então:
- Uma estrutura que define a lista contendo:
  - Um conjunto de itens para armazenar os dados;
  - Controle dos limites da estrutura (se for vetor: FIM; se for lista encadeada: CABECA)
- O item representa um elemento da lista, para cada item há:
  - Um tipo abstrato que representa o dado em si (distinto do item que o armazena);
  - Uma referência à sua posição (índice, no caso do vetor; PROX no caso de lista encadeada).
- Do valor do dado (na forma do tipo abstrato) que será armazenado na lista.

# Estrutura de dados: Listas

- Além dos dados, a estrutura necessita das operações primitivas.
- Vamos considear nas listas as seguintes operações primitivas: buscar, inserir e remover elementos.
- Também vamos trabalhar com outras operações relacionadas: como informar se a lista está vazia ou cheia e quantos itens possui.
- As listas podem ser implementadas na forma de vetor ou de listas encadeadas:
  - Somente para listas na forma de vetor corremos o risco de atingir seu tamanho, aí a informação de listaCheia é válida.
  - Opcionalmente poderíamos construir de forma dinâmica onde, se a lista encher, alocamos um vetor de tamanho maior para acomodá-la.

# Estrutura de dados: Linguagem

- A linguagem interfere na estrutura? Sim e Não:
  - Na estrutura todas as operações são sempre as mesmas independente do tipo.
  - Em uma linguagem fortemente "tipada" para cada tipo você precisa reescrever a estrutura.
  - Por exemplo: em C - uma lista depende dos itens que podem ser: int, double, struct, ...
  - Pior ainda, em C, em qualquer função, o return representa ou um tipo (primitivo) ou um ponteiro para tipo.
  - Se formos pensar em C++ (com os tipos *templates*) e o Python que não pré-determina um tipo, uma única programação da estrutura representa uma estrutura de qualquer tipo.
  - Também nestas linguagens os retornos de funções podem representar um objeto de qualquer classe.

# Implementação com Vetores

- Para simplificar vamos implementar, em C, lista de inteiros, os códigos em C++ são baseados em *templates*.
- Nada muda significativamente se o tipo for algum tipo abstrato (vejam as aulas sobre tipos de dados)
- Vamos usar um vetor de tamanho $N$ para armazenar a lista ($N$ é um parâmetro na declaração da lista)
- A lista é uma estrutura que contém o vetor e as informações sobre o início e fim da lista.
- Vamos tomar por enquanto o primeiro elemento de índice 0.
- Como regra comum (veja a documentação STL da linguagem C++) o início é o valor do índice do primeiro elemento.
- Fim é o índice da primeira posição após o último elemento: $n$.

# Implementação de uma lista com Vetores

```
#ifndef __LISTA_H__
#define __LISTA_H__

#define N 5 // O tamanho máximo de nossa lista
typedef int lista_t; // Vale para qualquer tipo primitivo
typedef int bool; // Padronizando com C++
enum {false,true};

typedef struct lista {
  lista_t itens[N]; // vetor de itens
  int fim; // controle da lista
} lista;

bool iniciar(lista *l); bool inserir(lista *l, lista_t val, int pos);
bool remover(lista *l, int pos);
bool ler(lista *l, int pos);
int ehVazia(lista *l);
int ehCheia(lista *l);

// Aqui vai a implementação das operações
#endif // __LISTA_H__
```

# Implementação da lista: no arquivo lista.h

```
int inserir(lista *l, lista_t val, int pos) {
  bool res = false;
  if(l->fim < N) {
  if(pos > l->fim) pos = l->fim;    for(int i = l->fim; i>pos; i--) l->itens[i]=l->itens[i-1];
    l->fim++;
    l->itens[pos] = val;
    res = true;
  }
  return res;
}
int remover(lista *l, int pos) {
  bool res = false;
  if(pos < l->fim) {
    l->fim--;
    for(int i = pos; i < l->fim; i++) l->itens[i] = l->itens[i+1];
    res = true;
  }
  return res;
}
int buscar(lista *l, int pos) {
  lista_t res=0;
  if(pos < l->fim) res = l->itens[pos];}
  return res
}
int ehVazia(lista *l) { return l->fim == 0; }
int ehCheia(lista *l) { return l->fim == N; }
```

# Uso da lista

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

void imprimir(lista l) {
  for(int i = 0; i<l.fim; i++)
    printf("%d..",buscar(&l,i));
  printf("\n");
  return;
}
int main() {
  lista l; iniciar(&l);
  if(ehVazia(&l)) printf("Vazia\n");
  else printf("Não vazia\n");
  inserir(&l,1,0); imprimir(l);
  inserir(&l,2,0); inserir(&l,3,1); imprimir(l);
  remover(&l,0); inserir(&l,4,1); imprimir(l);
  if(ehVazia(&l)) printf("Vazia\n");
  else printf("Não vazia\n");
  if(ehCheia(&l)) printf("Cheia\n");
  else printf("Não cheia\n");
  remover(&l,3); inserir(&l,5,2);
  inserir(&l,6,4); imprimir(l);
  if(ehCheia(&l)) printf("Cheia\n");
  else printf("Não cheia\n");
  inserir(&l,8,0); remover(&l,5); imprimir(l);
  remover(&l,4); imprimir(l);
  return 0;
}
```

# Uso da lista

Vazia
1..
2..3..1..
3..4..1..
Não vazia
Não cheia
3..4..5..1..6..
Cheia
3..4..5..1..6..
3..4..5..1..

# Qual o custo das operações

- Busca: $O(1)$
- Inserir: $O(n)$
- Remover: $O(n)$

# Implementação com Lista Encadeada

- Novamente, vamos implementar em C, lista de inteiros. Mas vale a pena pensar em trabalhar com uma linguagem mais maleável.
- O parâmetro de controle da lista, era FIM na lista com vetores, aqui temos a `cabeca`
- A lista é uma estrutura que contém um ponteiro para o começo de uma lista encadeada.
- Vamos tomar por enquanto o primeiro elemento a posição 0.
- Precisamos também de contar quantos elementos tem na lista para saber seu tamanho

# Implementação da Estrutura com Lista Encadeada:

```
#ifndef __LISTA_H__
#define __LISTA_H__
#include <malloc.h>
typedef int lista_t;// Vale para qualquer tipo primitivo
typedef int bool; // Padronizando com C++
enumfalse,true;
typedef struct item_lista { // Lista encadeada
  lista_t valor; // valor guardado
  struct item_lista *prox; // encadeamento
} item_lista;
typedef struct lista { // A lista (ED) contém apenas um ponteiro para
  item_lista *cabeca; // um elemento da estrutura (o primeiro).
} lista;

bool iniciar(lista *l);
bool inserir(lista *l, lista_t val, int pos);
bool remover(lista *l, int pos);
lista_t buscar(lista *l, int pos);
bool ehVazia(lista *l);
int tamanho(lista *l);
#endif // __LISTA_H__
```

# Implementação das funções:

```
bool iniciar(lista *l) {
 l->cabeca = NULL;
}

bool inserir(lista *l, lista_t val, int pos) {
 int i;
 item_lista *a, *p;
 item_lista *elemento= malloc(sizeof(item_lista));
 elemento->valor = val; elemento->prox = NULL;
 if(l->cabeca) {
  for(i=pos, p=l->cabeca; i>0 && p; i--, p=p->prox);
  elemento->prox = p;
  if(p==l->cabeca) l->cabeca=elemento;
  else {
   for(a=l->cabeca;a->prox!=p;a=a->prox);
   a->prox = elemento;
  }
 } else l->cabeca = elemento;
 return true;
}
```

# Implementação das funções:

```
bool remover(lista *l, int pos) {
 int i;
 bool res = false;
 item_lista *p, *a;
 for(i=pos, p=l->cabeca; i>0 && p; i--, p=p->prox);
 if(p) {
  if(p==l->cabeca) l->cabeca = l->cabeca->prox;
  else {
   for(a=l->cabeca;a->prox!=p;a=a->prox);
   a->prox = p->prox;
  }
  free(p);
  res = true;
 }
 return res;
}
```

# Implementação das funções:

```
lista_t buscar(lista *l, int pos) {
 int i;
 item_lista *p;
 lista_t res;
 for(i=pos, p=l->cabeca; i>0 && p; i--, p=p->prox);
 if(p) res = p->valor;
 return res;
}

bool ehVazia(lista *l) {
 return (l->cabeca == NULL);
}

int tamanho(lista *l) {
 int i = 0;
 for(item_lista *p=l->cabeca; p; i++, p=p->prox);
 return i;
}
```

# Usando a lista:

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

void imprimir(lista l) {
  for(int i = 0; i < tamanho(&l); i++)
    printf("%d..",buscar(&l,i));
  printf("\n");
  return;
}

int main() {
  lista l; iniciar(&l);
  if(ehVazia(&l)) printf("Vazia\n");
  else printf("Não vazia\n");
  inserir(&l,1,0); imprimir(l);
  inserir(&l,2,0); inserir(&l,3,1); imprimir(l);
  remover(&l,0); inserir(&l,4,1); imprimir(l);
  if(ehVazia(&l)) printf("Vazia\n");
  else printf("Não vazia\n");
  printf("Tamanho da lista:  %d\n",tamanho(&l));
  remover(&l,3); inserir(&l,5,2);
  inserir(&l,6,4); imprimir(l);
  printf("Tamanho da lista:  %d\n",tamanho(&l));
  inserir(&l,8,0); remover(&l,5); imprimir(l);
  remover(&l,4); imprimir(l);
  return 0;
}
```

# Usando a lista:

Saída para o código:

```
Vazia
1..
2..3..1..
3..4..1..
Não vazia
Tamanho da lista:  3
3..4..5..1..6..
Tamanho da lista:  5
8..3..4..5..1..
8..3..4..5..
```

# Vantagens do C++

- Através de "templates", é possível definir um tipo genérico que é substituído em tempo de execução por qualquer outro, assim uma única classe que define uma lista pode ser usada para representar lista de complexos, racionais, inteiros, ...
- Assim evitamos construir uma estrutura para cada tipo de dado. Uma única estrutura serve para todos.
- A classe já inclui todas as operações sobre a própria lista, é o encapsulamento da orientação a objetos.
- Como com o uso dos templates o tipo somente é definido no uso da biblioteca. A biblioteca não pode ser compilada até a definição do tipo.
- Todo o código genérico fica em arquivos ".hpp" e são compilados quando usados. Diferente de uma biblioteca C padrão que guarda as funções na forma de objetos.

## Variações sobre a lista encadeada

- Lista Circular
  - O último elemento, ao invés de ser aterrado ao NULL pode apontar para o primeiro elemento.
  - Quando a lista não é circular, precisamos de dois ponteiros, a "cabeça", que aponta para o primeiro elemento e um ponteiro de navegação que sempre vai do primeiro ao último.
  - Na lista circular, basta o ponteiro de navegação, pois não há cabeça na lista. Inserir e Remover é sempre na posição do elemento apontado.
  - Buscar atualiza o elemento apontado
  - Inserir e Remover precisa que um ponteiros auxiliar dê a volta na lista para saber quem aponta para a posição corrente.

## Aplicação de lista circular

- Um exemplo de aplicação da lista circular é o jogo da roleta russa.
- $n$ amigos formam um círculo, numerados de 1 a $n$.
- Um número $k$ é sorteado.
- Começando com o número 1, conta-se, de forma circular, até o número $k$ e ele é morto.
- O próximo da sequência reinicia a contagem.
- O jogo acaba quando só um sobrevive.

# Implementação da roleta russa usando lista circular

```
int main() {
 int n, k;
 lista<int> l;
 cin >> n >> k;
 for(int i=n; i>0; i--) l.inserir(i);
 while(--n) {
  cout << "Contando:  ";
  l.busca(k-1);
  cout << " morreu" << endl;
  l.remover();
 }
 cout << "Ganhador:  " << l.buscar(0) << endl;
 return 0;
}
```

```
5 3
Contando: 3 morreu
Contando: 1 morreu
Contando: 5 morreu
Contando: 2 morreu
Ganhador: 4
```

# Listas Duplamente Encadeadas

- O conceito é o semelhante de listas encadeadas.
- Além da ligação de um elemento com seu próximo, há a ligação de um elemento com seu anterior.
- O primeiro elemento da lista é ligado, na ligação com o anterior, ao `NULL`
- Além do apontador especial "`cabeça`", há o apontador especial "`cauda`".
- A navegação da lista pode começar no início ou final e pode andar em ambos sentidos.
- Apesar da facilidade na navegação, há uma complicação na manipulação, são mais ligações a serem refeitas.
- Porém as operações Inserir, Remover e Buscar, têm suas versões Reversa (começando pela cauda)