

# Programação Estruturada

Listas Ligadas

---

Professores Emílio Francesquini e Carla Negri Lintzmayer

2018.Q3

Centro de Matemática, Computação e Cognição  
Universidade Federal do ABC



## Disclaimer

- Esses slides foram preparados para um curso de Programação Estruturada ministrado na UFABC.
- Este material pode ser usado livremente desde que sejam mantidos os créditos dos autores e da instituição.
- Muitos dos exemplos apresentados aqui foram retirados de materiais preparados pelos Profs. Tomasz Kowaltowski e Orlando Lee da UNICAMP assim como do Prof. Paulo Feofiloff do IME-USP.

## Listas ligadas

---

# Lista ligada

- Uma **lista ligada** (= lista encadeada = *linked list*) é uma estrutura de dados para armazenar uma sequência de elementos
  - Neste sentido é parecida com o bom e velho vetor
- Cada elemento é armazenado em uma **célula** (= **nó** = **nodo**)
- Cada célula também armazena o **endereço do próximo elemento da lista**



## Lista ligada - Estrutura

```
1 struct reg {  
2     int conteudo;  
3     struct reg *prox;  
4 };
```



É comum criamos um **typedef** para facilitar o uso

```
1 typedef struct reg celula;
```

E então podemos declarar uma célula e um ponteiro para uma célula assim:

```
1 celula c;  
2 celula *p;
```

## Listas ligadas - estrutura

- Se  $c$  é uma célula então  $c.conteudo$  é o conteúdo da célula e  $c.prox$  é o endereço da próxima célula
- Se  $p$  é o endereço de uma célula, então  $p->conteudo$  é o conteúdo da célula e  $p->prox$  é o endereço da próxima célula
- Se  $p$  é o endereço da última célula da lista então  $p->prox$  vale **NULL**



## Listas ligadas - Representando a lista

Repare que a definição das células que compõem uma lista ligada é uma **definição recursiva**

---

```
1 struct reg {  
2     int  conteudo;  
3     struct reg *prox;  
4 };
```

---



Não por acaso, algoritmos que lidam com listas ligadas podem ser expressos recursivamente de maneira natural.

## Listas ligadas - Representando a lista

Podemos representar a lista através da primeira célula. Assim, o endereço da lista ligada é, efetivamente, o endereço da sua primeira célula

```
1 celula *lista;
```



**Pergunta: Como verificar se uma lista está vazia?**

```
1 int lista_vazia(celula *lista) {  
2     ...  
3 }
```

**Pergunta: Como verificar se uma lista está vazia?**

```
1 int lista_vazia(celula *lista) {  
2     /* equivalente a if (!lista), por quê? */  
3     if (lista == NULL)
```

# Manipulando listas ligadas

---

- A lista ligada é uma das **estruturas de dados** mais comuns.
- Muitas linguagens de programação oferecem **bibliotecas** recheadas com estruturas de dados para os mais diversos usos.
- Programas que queira utilizar essas bibliotecas devem seguir um contrato de utilização especificado pela sua **API** (*Application Programming Interface*).
  - Em C a maneira mais comum é através do uso de arquivos `.h` como, por exemplo o `math.h`.

Agora que já definimos uma função (`lista_vazia`), quais outras funções seriam interessantes de incluir na API da nossa lista ligada?

# Listas ligadas - API

---

```
1  struct reg {
2      int  conteudo;
3      struct reg *prox;
4  };
5
6  typedef struct reg celula;
7
8  int  lista_vazia(celula* lista);
9  void imprime_elementos(celula* lista);
10 celula* busca_elemento(int elem, celula* lista);
11 void insere_elemento(int elem, celula* lista);
12 void remove_elemento(celula* lista);
13 int  busca_e_remove(int elem, celula* lista);
```

---

```
1  struct reg {
2      int conteudo;
3      struct reg *prox;
4  };
5
6  typedef struct reg celula;
7
8  ...
9
10 void imprime_elementos(celula* ll) {
11     celula* atual = ll;
12     while (atual) {
13         printf("%d\n", atual->conteudo);
14         atual = atual->prox;
15     }
16 }
```

---

```
1  struct reg {
2      int conteudo;
3      struct reg *prox;
4  };
5  typedef struct reg celula;
6
7  ...
8
9  void imprime_rec(celula* ll) {
10     if (ll) {
11         printf("%d\n", ll->conteudo);
12         imprime_rec(ll->prox);
13     }
14 }
```

---

- **Exercício 1** – Faça uma versão **iterativa** de `imprime_elementos` que imprime os elementos da lista na ordem inversa.
- **Exercício 2** – Faça uma versão **recursiva** de `imprime_elementos` que imprime os elementos da lista na ordem inversa.

```
1  struct reg {
2      int conteudo;
3      struct reg *prox;
4  };
5
6  typedef struct reg celula;
7  ...
8
9  celula* busca_elemento(int elem, celula* lista) {
10     celula* atual = lista;
11     while (atual) {
12         if (atual->conteudo == elem)
13             return atual;
14         atual = atual->prox;
15     }
16     return NULL;
17 }
```

```
1  struct reg {
2      int conteudo;
3      struct reg *prox;
4  };
5
6  typedef struct reg celula;
7
8  ...
9
10 celula* busca_elemento_rec(int elem, celula* lista) {
11     if (!lista)
12         return NULL;
13     if (lista->conteudo == elem)
14         return lista;
15     else
16         return busca_elemento_rec(elem, lista->prox);
17 }
```

---

```
1 void insere_elemento(int elem, celula *lista) {  
2     celula nova;  
3     nova.conteudo = elem;  
4     nova.prox = lista->prox;  
5     lista->prox = &nova;  
6 }
```

---



# Inserção

```
1 void insere_elemento(int elem, celula *inicial) {  
2     celula *nova;  
3     nova = malloc(sizeof(celula));  
4     nova->conteudo = elem;  
5     nova->prox = inicial;  
6     inicial = nova;  
7 }
```



```
1 celula *primeiro;  
2 int k = 7;  
3 ...  
4 /* cria cópia de primeiro */  
5 insere_elemento(k, primeiro);
```

Ao voltar da chamada, o valor da variável primeiro **não foi alterado**.

---

```
1  /* Remove a célula seguinte à célula apontada por p */
2  void remove_elemento(celula *p) {
3      celula *removida;
4      removida = p->prox;
5      p->prox = removida->prox;
6      free(removida);
7  }
```

---

Note que a operação remove a célula seguinte àquela apontada por p.

**Qual o problema disso?**

- Nossas implementações de inserção e remoção **não funcionam!**
  - **Como inserir o primeiro** item na lista?
  - Supondo que já existam itens na lista, **como remover o primeiro deles?**

## Inserção - 1ª tentativa de conserto

---

```
1  celula* insere_elemento(int elem, celula* inicial) {
2      celula *nova;
3      nova = malloc(sizeof(celula));
4      nova->conteudo = elem;
5      nova->prox = inicial;
6      return nova; /* devolve o início da lista */
7  }
8
9  ...
10 primeiro = insere_elemento(k, primeiro);
```

---

Solução um tanto **artificial**: fica atualizando o ponteiro para o início sempre.

## Inserção - 2ª tentativa de conserto

---

```
1 void insere_elemento(int elem, celula** pini) {
2     celula *nova;
3     nova = malloc(sizeof(celula));
4     nova->conteudo = elem;
5     nova->prox = *pini;
6     *pini = nova;
7 }
8
9 ...
10
11 insere_elemento(k, &primeiro);
```

---

- Um pouco difícil de ler por causa da indireção.
- Nenhuma das soluções é satisfatória porque elas diferem do caso geral.

- Em vez passar como parâmetro o nó anterior, por que não passar o nó que desejamos remover?
- **Isso não é uma boa ideia. Por quê?**

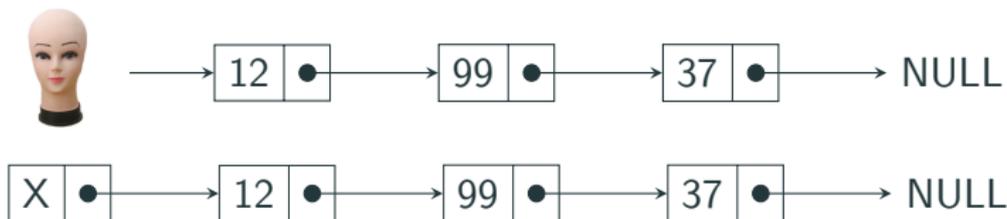
# Consertando a inserção e a remoção

- Temos dois problemas semelhantes: tanto a inserção quanto a remoção não estão muito “redondas” no caso do primeiro nó.
- **Como resolver?**

Solução: **Listas com cabeças!**



## Listas com cabeça



- A **cabeça** (= *head*) da lista ligada serve apenas como um marco de início. (Como tal, **ignoramos** o seu conteúdo.)
- Uma lista encadeada *lista com cabeça* está vazia se e somente se `lista->prox == NULL`.

---

```
1  /* versão sem cabeça */           /* versão com cabeça */
2  int lista_vazia(celula *lst) {    int lista_vazia(celula *lst) {
3      return !lst;                  return !lst->prox;
4  }                                  }
```

---

Para **criar** uma lista encadeada vazia com cabeça

---

```
1 celula *lista;
2 lista = malloc(sizeof(celula));
3 lista->prox = NULL;
```

---

Para **varrer** (imprimindo) os elementos:

---

```
1 void imprima(celula *lista) {
2     celula *p;
3     for (p = lista->prox; p != NULL; p = p->prox)
4         printf ("%d\n", p->conteudo);
5 }
```

---

Como **adaptar** as demais funções da API?

- **Busca**

---

```
1  celula* busca_elemento(int elem, celula* lista) {
2      celula* atual = lista->prox; /* lista aponta para a
   ↪  cabeça */
3      while (atual) {
4          if (atual->conteudo == elem)
5              return atual;
6          atual = atual->prox;
7      }
8      return NULL;
9  }
```

---

- **Inserção e remoção**

- Nada muda.

## Busca e remoção integradas

---

```
1  /* Remove da lista (com cabeça) 'lista' a primeira célula que  
   ↪ contém 'elem' */  
2  void busca_e_remove(int elem, celula* lista) {  
3      celula *p, *q;  
4      p = lista;  
5      q = lista->prox;  
6      while (q != NULL && q->conteudo != elem) {  
7          p = q;  
8          q = q->prox;  
9      }  
10     if (q != NULL) {  
11         p->prox = q->prox;  
12         free(q);  
13     }  
14 }
```

---

**Exercício 3** – Implemente uma versão semelhante ao `busca_e_remove` que funda as operações de busca e inserção.

**Vetores ou listas ligadas?**

---

**Pergunta:** O que é melhor usar: **vetores** ou **listas ligadas**?

**Resposta:** Depende.

- Vetores permitem indexação e usar menos memória (não precisa de ponteiros)
- Listas ligadas são mais flexíveis
  - Não é preciso saber o número de elementos a priori
  - Algumas operações podem ser mais simples

**Exemplo** Suponha que queremos manter um conjunto  $S$  de inteiros. Devemos ser capazes de realizar as seguintes operações sobre esse conjunto:

- mínimo
- $k$ -ésimo menor
- busca
- inserção
- remoção

Considere que  $n = |S|$ .

Implementação como um **vetor ordenado**:

- mínimo: custo 1
- $k$ -ésimo menor: custo 1
- busca: custo  $\leq \log_2 n$  (busca binária)
- inserção: custo  $\leq n$  (busca + movimentação de dados)
- remoção: custo  $\leq n$  (busca + movimentação de dados)

Implementação como uma **lista ligada ordenada**:

- mínimo: custo 1
- $k$ -ésimo menor: custo  $k$
- busca: custo  $\leq n$
- inserção: custo  $\leq n$  (busca)
- remoção: custo  $\leq n$  (busca)

## **Informações extras - Ordenação**

---

Considere a lista ligada definida abaixo:

---

```
1 struct No {  
2     int info;  
3     struct No *prox;  
4 }  
5  
6 typedef struct No No;
```

---

- Vamos descrever três algoritmos para ordenar uma lista ligada com cabeça pelo campo info.
- Vocês já conhecem alguns desses algoritmos implementados com vetores!

## Ordenação por seleção (= *selection sort*)

- A ideia básica consiste, em cada iteração:
  - **encontrar e remover** o menor elemento  $x$  da lista original  $ini$
  - inserir  $x$  no final de uma lista ordenada formada pelos elementos previamente removidos
- Vamos trabalhar usando uma **lista com cabeça**

## Ordenação por seleção

---

```
1 void selectionsort(No *ini) {
2     faça t apontar para a lista ini;
3     faça ini ser a lista vazia;
4     enquanto t for não vazia {
5         remova o menor elemento de t;
6         insira o elemento no final de ini;
7     }
8 }
```

---

Note que é conveniente ter um ponteiro para o final da lista.

## Ordenação por seleção

---

```
1 void selectionsort(No *ini) {
2     No *t = malloc(sizeof(No)), *last, *min;
3
4     t->prox = ini->prox;
5     ini->prox = NULL;
6     last = ini;
7     while (t->prox != NULL) {
8         min = remove_minimo(t);
9         last->prox = min;
10        last = min;
11        last->prox = NULL;
12    }
13    free(t);
14 }
```

---

## Ordenação por seleção

```
1  No *remove_minimo(No *ini) {
2      No *p, *q, *ant;
3      if (!ini->prox)
4          return NULL;
5      p = ant = ini;
6      q = ini->prox;
7      while (q) {
8          if (q->info < ant->prox->info)
9              ant = p;
10             p = q;
11             q = q->prox;
12         }
13         q = ant->prox;
14         ant->prox = q->prox;
15         return q;
16     }
```

---

Na função, `ant` aponta para o **nó anterior** ao nó com menor `info` encontrado até o momento.

# Ordenação por seleção

---

```
1  No *remove_minimo(No *ini) { /* outro jeito */
2      No **p, **min, *q;
3      if (!ini->prox)
4          return NULL;
5      p = min = &(ini->prox);
6      while (*p) {
7          if ((*p)->info < (*min)->info)
8              min = p;
9          p = &((*p)->prox);
10     }
11     q = (*min);
12     *min = q->prox;
13     return q;
14 }
```

---

As variáveis `p` e `min` são ponteiros para campos `prox` de nós da lista.

- Podemos analisar a complexidade da função `selectionsort` em função de  $n$ : o tamanho da lista `ini`
- O tempo gasto por `remove_minimo` é proporcional a  $n$
- `selectionsort` chama `remove_minimo`  $n$  vezes
- Assim, no total o tempo gasto é **proporcional a  $n^2$**

**Exercício 4** – É possível simplificar o código apresentado para efetuar a ordenação por seleção de forma que ela coloque o vetor em ordem crescente buscando a célula que contém o máximo da lista e colocando no início da lista de resposta. Implemente esta versão supondo que a lista tem cabeça.

**Exercício 5** – Escreva uma versão de `No *selectionsort(No *ini)` que recebe uma lista ligada **sem cabeça** `ini` e devolve um ponteiro para uma **lista ligada ordenada sem cabeça** com os nós da lista original.

**Sugestão:** em vez de usar uma função `remove_minimo` é melhor escrever o código que faz isto dentro da função `selectionsort`, para evitar o problema de ter de remover um nó no início da lista. Não fica tão modularizado, obviamente. Outra ideia é usar um ponteiro para ponteiro como parâmetro.

## Ordenação por inserção (= *insertion sort*)

- A ideia básica consiste, em cada iteração:
  - remover o primeiro elemento  $x$  da lista original  $ini$
  - inserir  $x$  na posição correta de uma lista ordenada formada pelos elementos previamente removidos
- Na implementação em C, passamos o nó cabeça da lista que queremos ordenar e este deve ser o nó cabeça da lista ordenada.

## Ordenação por inserção

---

```
1 void insertionsort(No *ini) {
2     faça t apontar para a lista ini;
3     faça ini ser a lista vazia;
4     enquanto t for não vazia {
5         remova o primeiro elemento de t;
6         insira o elemento em ordem na lista ini;
7     }
8 }
```

---

# Ordenação por inserção

---

```
1 void insertionsort(No *ini) {
2     No *t, *x;
3     t = ini->prox;
4     ini->prox = NULL;
5     while (t) {
6         x = t;
7         t = t->prox;
8         insere_ordenado(ini, x);
9     }
10 }
```

---

A lista  $t$  não tem cabeça.

# Ordenação por inserção

---

```
1 void insere_ordenado(No *ini, No *x) {
2     No *p, *q,
3     p = ini;
4     q = ini->prox;
5     while (q && q->info < x->info) {
6         p = q;
7         q = q->prox;
8     }
9     x->prox = q;
10    p->prox = x;
11 }
```

---

- Podemos analisar a complexidade da função `insertionsort` em função de  $n$ , o tamanho da lista `ini`.
- O tempo gasto por `insere_ordenado` é proporcional a  $n$  no pior caso.
- `insertionsort` chama `insere_ordenado`  $n$  vezes.
- Assim, no total o tempo gasto é **proporcional a no máximo  $n^2$** .

**Exercício 6** – Escreva uma versão de `insere_ordenado` que usa ponteiros para ponteiros (como na segunda versão de `remove_minimo`).

## Intercalando duas listas

**Problema:** intercalar (= *merge*) duas listas ordenadas

Queremos implementar a seguinte função:

```
No *intercala(No *s, No *t);
```

A função `intercala` recebe duas **listas ligadas ordenadas sem cabeça** `s` e `t`, e devolve uma **lista ordenada sem cabeça** contendo os nós das listas `s` e `t`.

Também podemos usar listas com cabeça: o código é praticamente idêntico. Só é preciso descartar (`free`) uma das cabeças ao final.

---

```
1 No *intercala(No *s, No *t) {
2     No *p, *q, cabeca, *last;
3     last = &cabeca;
4     cabeca.prox = NULL;
5     p = s;
6     q = t;
7     while (p && q) {
8         if (p->info < q->info) {
9             last->prox = p;
10            last = p;
11            p = p->prox;
12        } else {
13            last->prox = q;
14            last = q;
15            q = q->prox;
16        }
17    }
18    if (p)
19        last->prox = p;
20    else
21        last->prox = q;
22    return cabeca.prox;
23 }
```

A ideia do algoritmo implementado pela função `mergesort` é conceitualmente muito simples.

- Se a lista `ini` for pequena, então ordene diretamente.
- Caso contrário, divida a lista `ini` em duas listas `s` e `t` de tamanhos aproximadamente iguais.
- Recursivamente, ordene as listas `s` e `t`.
- Intercale as listas `s` e `t` e devolva a lista resultante.

# Mergesort

---

```
1 No *mergesort(No *ini) {
2     No *met, *metade;
3     /* caso base */
4     if (ini == NULL || ini->prox == NULL)
5         return ini;
6     metade = acha_metade(ini);
7     met = metade->prox;
8     metade->prox = NULL; /* divide a lista */
9     ini = mergesort(ini);
10    met = mergesort(met);
11    return intercala(ini, met);
12 }
```

---

## Como achar a metade rapidamente?

---

```
1  No *acha_metade(No *ini) {
2      No *slow, *fast;
3      if (ini == NULL)
4          return ini;
5      slow = fast = ini;
6      while (fast->prox && fast->prox->prox) {
7          slow = slow->prox;
8          fast = fast->prox->prox;
9      }
10     return slow;
11 }
```

---

## Mergesort - Análise (informal)

Suponha que  $T(n)$  seja o tempo de pior caso que o mergesort leva para ordenar uma lista ligada com  $n$  nós.

- Se  $n \geq 2$  então acontecem os seguintes passos:
  - Divide a lista
  - Duas chamadas recursivas
  - Intercalação
- Logo o tempo gasto é:

$$T(n) = n + 2T(n/2) + n = 2T(n/2) + 2n$$

- Assim,

$$T(n) = \begin{cases} 2T(n/2) + 2n, & \text{se } n \geq 2 \\ 1, & \text{caso contrário} \end{cases}$$

Para ter uma intuição, suponha que  $n = 2^k$ . Note que há:

- 1 chamada para uma lista de tamanho  $n$  (**nível 1**)
- 2 chamadas para uma lista de tamanho  $n/2$  (**nível 2**)
- 4 chamadas para uma lista de tamanho  $n/4$  (**nível 3**)
- $2^{i-1}$  chamadas para uma lista de tamanho  $n/2^{i-1}$  (**nível  $i$** )
- $2^k - 1$  chamadas para uma lista de tamanho 2 (**nível  $k - 1$** )
- $2^k$  chamadas para uma lista de tamanho 1 (**nível  $k$** )

## Mergesort - Análise (informal)

Ignore o tempo de dividir a lista. Vamos contar o custo das intercalações de listas (IL) em cada nível.

- **nível 1:** 1 IL de tamanho  $n/2$ ; tempo  $2 \times (n/2) = n$
- **nível 2:** 2 ILs de tamanho  $n/4$ ; tempo  $2 \times 2 \times (n/4) = n$
- **nível 3:** 4 ILs de tamanho  $n/8$ ; tempo  $2 \times 4 \times (n/8) = n$
- **nível  $i$ :**  $2^{i-1}$  ILs de tamanho  $n/2^i$ ; tempo  $2 \times 2^{i-1} \times (n/2^i) = n$
- **nível  $k-1$ :**  $2^{k-2}$  ILs de tamanho 2; tempo  $2 \times 2^{k-2} \times (2) = n$
- **nível  $k$ :**  $2^{k-1}$  ILs de tamanho 1; tempo  $2 \times 2^{k-1} \times (1) = n$

Assim, o tempo total é  $(k-1)n \approx n \log_2 n$ .

- Para  $n$  grande, temos que  $n \log_2 n \ll n^2$ .
- Assim, o algoritmo `mergesort` é bem mais eficiente que os algoritmos quadráticos `bubblesort`, `selectionsort` e `insertionsort`.

**Exercício 7** – Escreva uma versão **recursiva** da função `No`  
`*intercala(No *s, No *t)` que não use laços.

## **Informações extras: listas duplamente ligadas**

---

# Listas duplamente ligadas

---

```
1 struct NoDuplo {
2     int info;
3     struct NoDuplo *ant;
4     struct NoDuplo *prox;
5 };
6 typedef struct NoDuplo NoD;
```

---

**Vantagens:** maior acessibilidade

**Desvantagens:** dobro de ponteiros e mais trabalho para manter a lista.

A função `busca` recebe uma lista duplamente ligada `ini` e um inteiro `k`, e devolve um ponteiro para o primeiro nó com chave `k` ou `NULL`, se não houver.

---

```
1 No *busca(NoD *ini, int k) {
2     NoD *p = ini;
3     while (p && p->info != k)
4         p = p->prox;
5     return p;
6 }
```

---

Virtualmente idêntica à função `busca` para lista ligadas simples.

# Inserção

A função `insere` recebe uma lista duplamente ligada `ini` e um elemento `k` e insere um novo nó com conteúdo `k` entre o nó apontado por `p` e o seguinte. Só faz sentido se `p != NULL`.

---

```
1 void insere(NoD *p, int k) {
2     NoD *novo;
3     novo = malloc(sizeof(NoD));
4     novo->info = k;
5     novo->prox = p->prox;
6     novo->ant = p;
7     if (p->prox)
8         p->prox->ant = novo;
9     p->prox = novo;
10 }
```

---

Não faz inserção no início de uma lista duplamente ligada, a não ser que tenha nó cabeça.

## Remoção

A função `remove` recebe um ponteiro `q` para um nó de uma lista duplamente ligada e o remove.

---

```
1 void remove(NoD *q) {
2     No *p = q->ant;
3     p->prox = q->prox;
4     if (q->prox)
5         q->prox->ant = p;
6     free(q);
7 }
```

---

Note a diferença com listas ligadas simples.

Não funciona se `q` aponta para o primeiro elemento da lista, a não ser que tenha nó cabeça.

## **Informações extras: listas ligadas circulares**

---

Tem a mesma declaração de uma lista ligada simples, mas o campo `prox` do último nó aponta para o primeiro nó.

- **Lista ligada circular sem cabeça:** um problema é a lista vazia.
- **Lista ligada circular com cabeça** `ini`
  - **Lista vazia:** `ini->prox == ini`

Pode-se implementar outras variantes:

- listas ligadas circulares com ou sem cabeça
- listas duplamente ligadas circulares
- ou com ambas as formas.

## Busca em lista circular

A função busca recebe uma lista ligada circular com cabeça `ini` e um inteiro `k`, e devolve um ponteiro para o primeiro nó com chave `k` ou `NULL`, se não houver.

---

```
1 No *busca(No *ini, int k) {
2     No *p = ini->prox;
3     ini->info = k; /* sentinela */
4     while (p->info != k)
5         p = p->prox;
6     ini->info = NULL;
7     if (p == ini)
8         return NULL;
9     else
10        return p;
11 }
```

---

**Exercício 8** – Escreva uma função `remove_llcc(No *ini, int k)` que recebe um ponteiro para uma lista ligada circular com cabeça `ini` e remove o primeiro nó com chave `k`.

**Exercício 9** – Escreva uma função `remove_ldlcc(NoD *p)` que recebe um ponteiro para um nó `p` de uma lista duplamente ligada circular com nó cabeça e o remove da lista. Naturalmente, suponha que `p` não é o nó cabeça da lista.

**Exercício 10** – Escreva uma função `insere_lcco(No *ini, int k)` que recebe uma lista ligada circular com cabeça ordenada `ini` e insere um novo nó com chave `k` na posição correta.

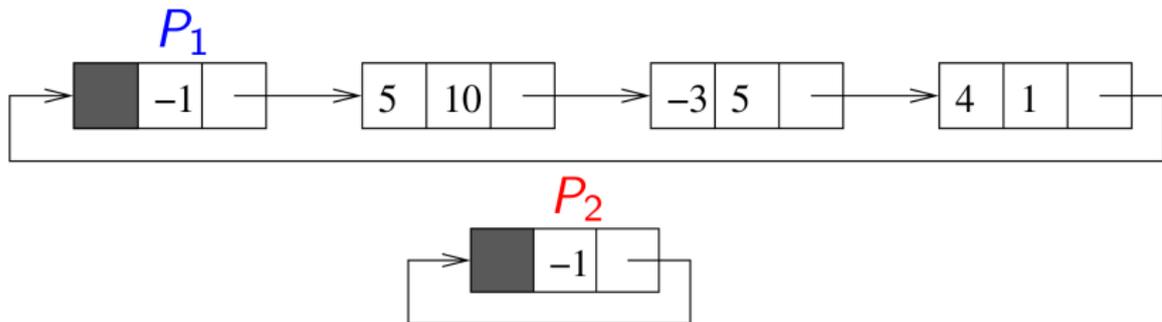
## Exercício: representando polinômios

Considere o polinômio:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_2 x^2 + a_1 x + a_0$$

onde  $a_n \neq 0$

Representação de  $P_1(x) = 5x^{10} - 3x^5 + 4$  e  $P_2(x) = 0$



## Exercício: representando polinômios

---

```
1 typedef struct AuxPol {
2     float coef;
3     int expo;
4     struct AuxPol *prox;
5 } Termo, *Polin;
```

---

A lista tem cabeça. O campo `expo` da cabeça é igual a  $-1$  para ser usado como sentinela.

Note que:

---

```
1 Termo *p;
2 Polin p;
```

---

são declarações **equivalentes**.

## Exercício: representando polinômios

A função `imprime(Polin p)` imprime um polinômio `p` exibindo os pares `(coef, expo)` de cada termo.

---

```
1 void imprime(Polin pol) {
2     Termo *p = pol->prox;
3     if (p == pol) {
4         printf("Polinomio nulo.\n");
5         return;
6     }
7     while (p->expo != -1) {
8         printf("%5.1f, %2d ", p->coef, p->expo);
9         p = p->prox;
10    }
11    printf("\n");
12 }
```

---

**Exercício 11** – Implemente em C funções para cada uma das operações abaixo:

- Calcule o valor de um polinômio  $P(x)$  em um ponto  $x_0$
- Calcule a soma de dois polinômios (usando o método da intercalação)
- Calcule o produto de dois polinômios
- Calcule a  $k$ -ésima derivada de um polinômio

## Exercício: Problema de Josephus

- Um grupo de  $N$  pessoas precisa eleger um líder.
- Decidiu-se usar a seguinte ideia para eleger um líder: forma-se um círculo com as  $N$  pessoas e escolhe-se um inteiro  $k$ . Começamos com uma pessoa qualquer e percorremos o círculo em sentido horário, eliminando cada  $k$ -ésima pessoa. A **última pessoa** que restar será o líder. Veja o verbete sobre Josephus na Wikipedia.

**Problema de Josephus:** coloque os números  $1, 2, \dots, N$  em um círculo nesta ordem e começando em 1 aplique o algoritmo acima com um valor  $k$ . Determine o último número, denotado  $J(N, k)$ .

**Exercício 12** – Escreva uma função `int josephus(int N, int k)` que calcula  $J(N, k)$ .