Estruturas de Dados

Aula 6 — Listas Encadeadas: Simples, Circular, Dupla e Dupla Circular

Prof. Ana Carolina Sokolonski

Bacharelado em Sistemas de Informação
Instituto Federal da Bahia — Campus Feira de Santana

2026

Agenda

Listas Encadeadas

Listas Simplesmente Encadeadas

Listas Simplesmente Encadeadas Circulares

Listas Duplamente Encadeadas

Listas Duplamente Encadeadas Circulares

Comparação e Exercícios

Listas Encadeadas

Estruturas dinâmicas com ponteiros

O que é uma Lista Encadeada?

Definição

Uma lista encadeada é uma estrutura de dados **dinâmica** que implementa uma coleção ordenada de valores, na qual o mesmo valor pode ocorrer mais de uma vez. Os elementos são interligados por **ponteiros**.

Propriedades fundamentais

- Nós criados dinamicamente com `malloc`
- Memória **não contígua**: nós em endereços arbitrários
- Sem indexação: acesso sempre percorrendo do início
- Inserção/remoção eficiente: apenas redireciona ponteiros

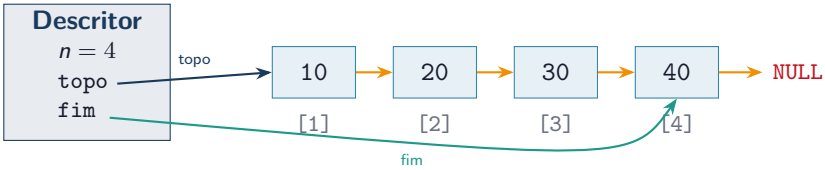
Vetor vs. Lista

	Vetor	**Lista**
Tamanho	fixo	dinâmico
Memória	contígua	dispersa
Acesso	$O(1)$	$O(N)$
Inserção	$O(N)$	$O(1)$*
Remoção	$O(N)$	$O(1)$*

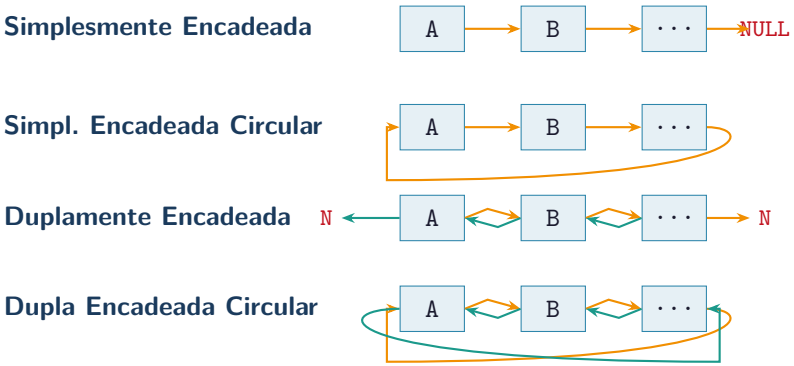
*com ponteiro para o nó

Estrutura de uma Lista com Descritor

- **Descritor aponta para o início e o fim; guarda o tamanho *n***



Tipos de Listas Encadeadas



Listas Simplesmente Encadeadas

Encadeamento unidirecional

Lista Simplesmente Encadeada — Conceito

Definição

Estrutura em que os nós são encadeados de forma **unidirecional**: pode-se percorrer do primeiro ao último, mas não na direção inversa. Cada nó contém um **dado** e um **ponteiro para o próximo nó**.

Struct em C

`conteudo`: dado armazenado

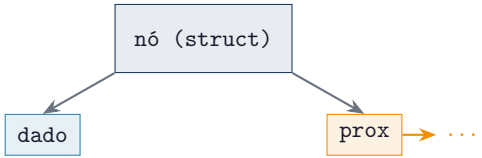
`prox`: ponteiro para o próximo nó

Último nó: `prox = NULL`

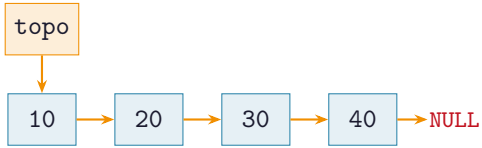
Acesso

Sempre a partir do `topo` (cabeça). Para acessar o k-ésimo elemento: percorrer $k-1$ ponteiros $O(N)$.

Estrutura de um nó



Lista com 4 elementos



Lista Simplesmente Encadeada — Struct e Inicialização

lista_simples.c

```c
// Definição do nó da lista
typedef struct No {
    int        conteudo; // dado
    struct No *prox;     // próximo
} No;
typedef struct {
    No *topo; // primeiro nó
    No *fim;  // último nó
    int n;    // qtd elementos
} Lista;
// Inicializa lista vazia
void inicializa(Lista *L) {
    L->topo = NULL;
    L->fim  = NULL;
    L->n    = 0;
}
```

Por que `typedef`?

`typedef` cria um alias para o tipo, permitindo escrever `No *p` em vez de `struct No *p` em todo o código.

Inicialização

Sempre inicializar `topo` e `fim` como `NULL` e `n = 0` antes de usar a lista. Lista vazia: `topo == NULL`.

Atenção com malloc

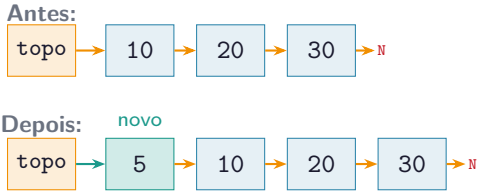
Todo nó criado com `malloc` deve ser liberado com `free` quando for removido, evitando vazamento de memória.

Lista Simplesmente Encadeada — Inserir no Início

inserir.c

```c
void inserir(Lista *L, int val){
  No *novo = (No*) malloc(sizeof(
      No));
  novo->conteudo = val;
  novo->prox     = NULL;
  if(L->topo == NULL){
//lista vazia: novo é único
    L->topo = novo;
    L->fim  = novo;
  }else{ // insere no início
    novo->prox = L->topo;
    L->topo    = novo;
  }
  L->n++;
}
```

Inserindo 5 em {10, 20, 30}

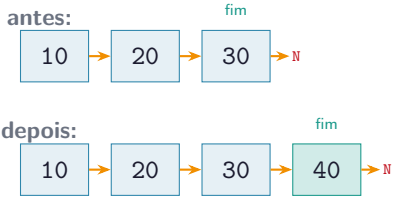


Lista Simplesmente Encadeada — Inserir no Final

inserirFinal.c

```c
void inserirFinal(Lista *L, int val){
  No *novo =(No*)malloc(sizeof(No));
  novo->conteudo = val;
  novo->prox     = NULL;
  if (L->topo == NULL) {
    L->topo = novo;
    L->fim  = novo;
  }else{
/* fim aponta para o último nó */
    L->fim->prox = novo;
    L->fim = novo;
  }
  L->n++;
}
```

Inserindo 40 no final de {10, 20, 30}



Complexidade

$O(1)$ com ponteiro `fim`.

Sem `fim`: $O(N)$ (percorre tudo).

Lista Simplesmente Encadeada — Imprimir

Versão iterativa

```c
void imprimirIter(Lista *L) {
  No *atual = L->topo;
  printf("[");
  while (atual != NULL) {
    printf("%d", atual->
        conteudo);
    if (atual->prox != NULL)
        printf(" -> ");
    atual = atual->prox;
   }
   printf("]\n");
}
```

Versão recursiva

```c
void imprimirRec(No *no) {
    if (no == NULL) {
        printf("NULL\n");
        return;
    }
    printf("%d -> ", no->
        conteudo);
    imprimirRec(no->prox);
}
/* Chamada: */
/* imprimirRec(L.topo); */
```

- **Saída para {10, 20, 30}**

`[10 -> 20 -> 30]` (iterativa) `10 -> 20 -> 30 -> NULL` (recursiva)

Lista Simplesmente Encadeada — Buscar

Versão iterativa

```
No *buscarIter(Lista *L, int val) {
  No *atual = L->topo;
  while(atual != NULL){
    if (atual->conteudo == val)
        return atual; /* achou */
      atual = atual->prox;
  }
  return NULL;//não encontrado
}
```

Versão recursiva

```
No *buscarRec(No *no,int val){
  if(no == NULL) return NULL;
  if(no->conteudo == val)
    return no;
  return buscarRec(no->prox,
      val);
}
/* Chamada: */
/* buscarRec(L.topo, 20); */
```

Retorno

Ponteiro para o nó encontrado, ou `NULL` se o valor não existir na lista.

Complexidade

Melhor caso: $O(1)$ (topo).

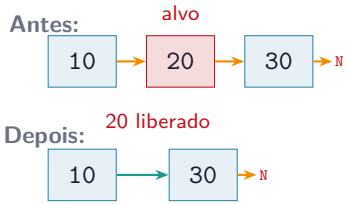
Pior caso: $O(N)$ (último ou ausente).

Lista Simplesmente Encadeada — Remover

remover.c

```c
void remover(Lista *L, int val) {
  No *ant  = NULL;
  No *curr = L->topo;
  while(curr!=NULL && curr->
      conteudo!=val){
    ant  = curr;
    curr = curr->prox;
  }
  if(curr==NULL) return;//não achou
  if(ant==NULL)
    L->topo = curr->prox;//era topo
  else ant->prox = curr->prox;
  if (curr == L->fim)
    L->fim = ant;//era fim
  free(curr);
  L->n--;
}
```

Removendo 20 de {10, 20, 30}



- **Dois passos**

1. Redirecionar `ant->prox`
2. `free(alvo)` liberar memória

Listas Simplesmente Encadeadas Circulares

O último aponta para o primeiro

Lista Circular — Conceito

Definição

Lista em que o último nó aponta de volta para o primeiro, formando um **ciclo**. Não existe início, meio ou fim absolutos. O encadeamento é ainda unidirecional.

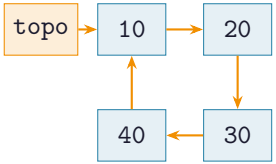
Diferença da lista simples

- Último nó: `prox = topo` (não `NULL`)
- Condição de fim de percurso: `atual == topo`
- Útil para fila circular, round-robin, agendas de processos

Cuidado com loops

Um laço `while (atual != NULL)` nunca termina em lista circular. Usar o nó inicial como referência sentinela.

Lista circular com 4 nós



Lista Circular — Inserir e Imprimir

Inserir no início

```
void inserirCirc(Lista *L,int val){
  No *novo =(No*)malloc(sizeof(No));
  novo->conteudo = val;
  if (L->topo == NULL){
    novo->prox = novo;//aponta p/ si
    L->topo = novo;
    L->fim  = novo;
  } else {
    novo->prox = L->topo;
    L->fim->prox = novo;
    L->topo = novo;
  }
  L->n++;
}
```

Imprimir (sentinela = topo)

```
void imprimirCirc(Lista *L) {
  if (L->topo == NULL) {
    printf("[vazia]\n"); return;
  }
  No *atual = L->topo;
  printf("[");
  do {
    printf("%d",atual->conteudo);
    atual = atual->prox;
    if (atual != L->topo)
      printf(" -> ");
  }while (atual != L->topo);
  printf("]\n");
}
```

Lista Circular — Buscar e Remover

Buscar

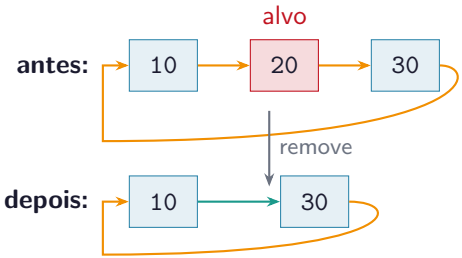
```
No *buscarCirc(Lista *L,int val){
  if(L->topo==NULL) return NULL;
  No *atual = L->topo;
  do {
    if(atual->conteudo==val)
      return atual;
    atual = atual->prox;
  }while (atual != L->topo);
  return NULL;
}
```

Remover por conteúdo

```
void removerCirc(Lista *L,int val){
  if (L->topo==NULL) return;
  No *ant = L->fim;//começa do fim
  No *curr = L->topo;
  do {
    if (curr->conteudo==val){
      ant->prox = curr->prox;
      if (curr==L->topo)
        L->topo = curr->prox;
      if (curr==L->fim) L->fim=ant;
      free(curr);
      L->n--; return;
    }
    ant  = curr;
    curr = curr->prox;
  }while (curr != L->topo);
}
```

Lista Circular — Remoção (visualização)

Removendo 20 de $\{10 \rightarrow 20 \rightarrow 30 \rightarrow (10)\}$



Caso especial

Ao remover o único nó:

```
L->topo = NULL;
L->fim = NULL;
```

Listas Duplamente Encadeadas

Encadeamento bidirecional

Lista Duplamente Encadeada — Conceito

Definição

Cada nó contém um ponteiro para o **próximo** e um para o **anterior**, permitindo percorrer a lista em ambos os sentidos.

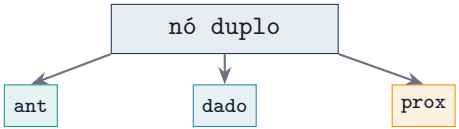
Vantagens

- Remoção em $O(1)$ com ponteiro direto ao nó
- Percurso em ordem crescente e decrescente
- Inserção antes ou depois de um nó dado

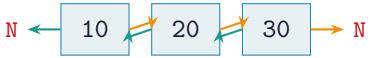
Custo extra

Cada nó usa mais memória (dois ponteiros). Inserção/-remoção manipulam dois ponteiros em vez de um atenção à **ordem das atribuições**.

Struct e representação



Lista com 3 nós

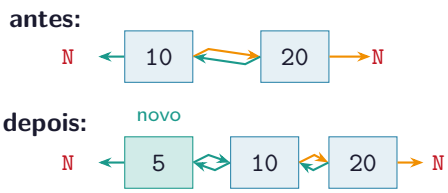


Lista Duplamente Encadeada — Struct e Inserir

lista_dupla.c

```
typedef struct NoD {
  int conteudo;
  struct NoD *prox;
  struct NoD *ant;
} NoD;
void inserirDupla(Lista *L,int val){
  NoD *novo=(NoD*)malloc(sizeof(NoD));
  novo->conteudo = val;
  novo->ant  = NULL;
  novo->prox = NULL;
  if (L->topo == NULL){
    L->topo = novo; L->fim = novo;
  }else {// insere no início
    novo->prox = L->topo;
    L->topo->ant = novo;
    L->topo = novo;
  } L->n++; }
```

Inserindo 5 em {10, 20}



Lista Duplamente Encadeada — Imprimir e Buscar

Imprimir (crescente e decrescente)

```c
void imprimirCres(Lista *L){
  NoD *p = L->topo;
  while (p != NULL) {
    printf("%d ", p->conteudo);
    p = p->prox;
  }
  printf("\n");
}
void imprimirDesc(Lista *L){
  NoD *p = L->fim;
  while (p != NULL){
    printf("%d ", p->conteudo);
    p = p->ant;
  }
  printf("\n");
}
```

Buscar por conteúdo

```c
NoD *buscarDupla(Lista *L,int
    val){
  NoD *p = L->topo;
  while (p != NULL){
    if (p->conteudo == val)
      return p;
    p = p->prox;
  }
  return NULL;
}
```

Percurso inverso

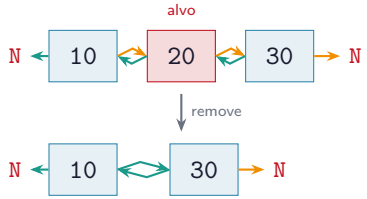
Graças ao ponteiro `ant`, pode-se percorrer a partir do `fim` sem custo extra de busca, o que é útil para histórico, deque, etc.

Lista Duplamente Encadeada — Remover

removerDupla.c

```
1  void removerDupla(Lista *L,int val){
2    NoD *p = L->topo;
3    while (p!=NULL && p->conteudo!=val)
4      p = p->prox;
5    if (p == NULL) return;//não achou
6  // reconecta vizinhos
7    if (p->ant != NULL)
8      p->ant->prox = p->prox;
9    else
10     L->topo = p->prox;//era o topo
11   if (p->prox != NULL)
12     p->prox->ant = p->ant;
13   else
14     L->fim = p->ant;//era o fim
15   free(p); L->n--;
16 }
```

Removendo 20 de {10, 20, 30}



- **4 ponteiros**

Reconectar: `ant.prox`, `prox.ant`, e verificar se é `topo` ou `fim`.

Listas Duplamente Encadeadas Circulares

Bidirecional sem início nem fim

Lista Dupla Circular — Conceito

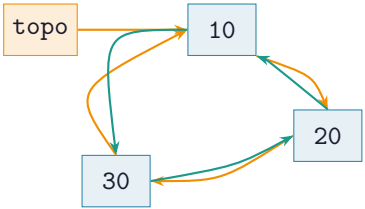
Definição

Lista duplamente encadeada cujo último nó aponta para o primeiro (`prox`) e o primeiro aponta para o último (`ant`). Não existe ponteiro `NULL`: a lista forma um **anel bidirecional**.

Propriedades

- Nó sozinho: `prox = ant = ele_mesmo`
- Sem `NULL` em nenhum ponteiro
- Percurso em qualquer sentido, partindo de qualquer nó
- Base para filas de prioridade e estruturas de SO (escalonador de processos)

Lista dupla circular com 3 nós



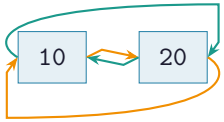
Lista Dupla Circular — Inserir

inserirDC.c

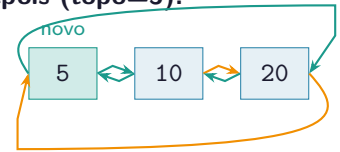
```
1  void inserirDC(Lista *L, int val){
2      NoD *novo =(NoD*)malloc(sizeof(NoD));
3      novo->conteudo = val;
4      if (L->topo == NULL) { //único nó
5          novo->prox = novo;
6          novo->ant  = novo;
7          L->topo    = novo;
8          L->fim     = novo;
9      } else { // insere antes do topo
10         NoD *fim     = L->fim;
11         novo->prox   = L->topo;
12         novo->ant    = fim;
13         fim->prox    = novo;
14         L->topo->ant = novo;
15         L->topo      = novo;
16     }
17     L->n++;}
```

Inserindo 5 em {10, 20}

antes (fim=20, topo=10):



depois (topo=5):



Lista Dupla Circular — Imprimir e Buscar

Imprimir (sentido crescente)

```
void imprimirDC(Lista *L){
  if (L->topo == NULL) {
    printf("[vazia]\n"); return;
  }
  NoD *p = L->topo;
  printf("[");
  do {
    printf("%d", p->conteudo);
    p = p->prox;
    if (p != L->topo)
      printf(" <-> ");
  }while(p!=L->topo);
  printf("]\n");
}
```

Buscar

```
NoD *buscarDC(Lista *L,int val){
  if (L->topo==NULL) return NULL;
  NoD *p = L->topo;
  do{
    if(p->conteudo==val) return p
        ;
    p = p->prox;
  }while(p != L->topo);
  return NULL;
}
```

- **`do-while` obrigatório**

Sem `do-while`, a condição `p != L->topo` seria falsa imediatamente ao começar pelo próprio topo.

Lista Dupla Circular — Remover

removerDC.c

```c
void removerDC(Lista *L, int val) {
  if (L->topo == NULL) return;
  NoD *p = L->topo;
  do {
    if (p->conteudo == val) break;
    p = p->prox;
  } while (p != L->topo);
  if (p->conteudo != val) return;
  if (L->n == 1) { /* único elemento */
    L->topo = NULL; L->fim = NULL;
  } else {
    p->ant->prox = p->prox;
    p->prox->ant = p->ant;
    if (p == L->topo) L->topo = p->prox;
    if (p == L->fim)  L->fim  = p->ant;
  }
  free(p); L->n--;}
```

Casos especiais

1. Único nó: zerar `topo` e `fim`
2. Era o topo: avançar `topo`
3. Era o fim: recuar `fim`
4. Meio: apenas reconectar vizinhos

Complexidade

Busca: $O(N)$

Remoção (com ponteiro ao nó): $O(1)$

Inserção: $O(1)$

Comparação e Exercícios

Escolhendo a estrutura certa

Comparação entre os Tipos de Lista

Propriedade	Simples	Simpl. Circ.	Dupla	Dupla Circ.
Ponteiros por nó	1	1	2	2
Percurso inverso	Não	Não	Sim	Sim
Remoção $O(1)$*	Não	Não	Sim	Sim
Ponteiro `NULL`	Sim	Não	Sim	Não
Implementação	simples	média	média	complexa
Uso típico	pilha/fila	round-robin	editor/deque	SO/agenda

*com ponteiro direto ao nó a remover

Regra de ouro

Use a estrutura mais **simples** que atenda aos requisitos. Ponteiros extras custam memória e aumentam a complexidade de inserção/remoção.

Exercícios Propostos

Exercício 1

Dada uma lista simplesmente encadeada de inteiros, escreva uma função que a **divide em duas**: uma com os elementos pares e outra com os ímpares. Manipule apenas ponteiros, sem copiar conteúdos.

Exercício 2

Escreva funções que retornem:

- a **quantidade de elementos**
- o número de **ocorrências** do valor j

Implemente para lista simples e para lista circular.

Exercício 3

Implemente uma função `inverter(Lista *L)` para uma lista **duplamente encadeada** que inverte a ordem dos elementos apenas redirecionando ponteiros (sem criar novos nós).

Exercício 4 (desafio)

Usando uma **lista dupla circular**, implemente um simulador de round-robin: N processos com fatias de tempo iguais. A cada iteração, imprime o processo atual e avança para o próximo.

Fim da Aula 6

Dúvidas e próximos passos

Próxima Aula e Referências

Próxima aula

- **Pilhas**: definição, operações, implementação
- **Filas**: definição e implementação com lista
- Aplicações: avaliação de expressões, BFS

Referências

- CORMEN et al. *Introduction to Algorithms*, 4.ł ed. MIT Press, 2022. Cap. 10.
- TENENBAUM et al. *Estruturas de Dados usando C*. Makron Books, 1995. Cap. 4.
- SEDGEWICK; WAYNE. *Algorithms*, 4.ł ed. Cap. 1.3.

Resumo

- **Simples**: unidirecional, $O(N)$ busca
- **Circular**: sem NULL, sentinela `topo`
- **Dupla**: bidirecional, remoção $O(1)$
- **Dupla circular**: anel, sem NULL, base de SOs

Dúvidas?

carolsoko@ifba.edu.br

IFBA – Campus Feira de Santana