

TÓPICO 09: LISTAS ENCADEADAS DUPLAS

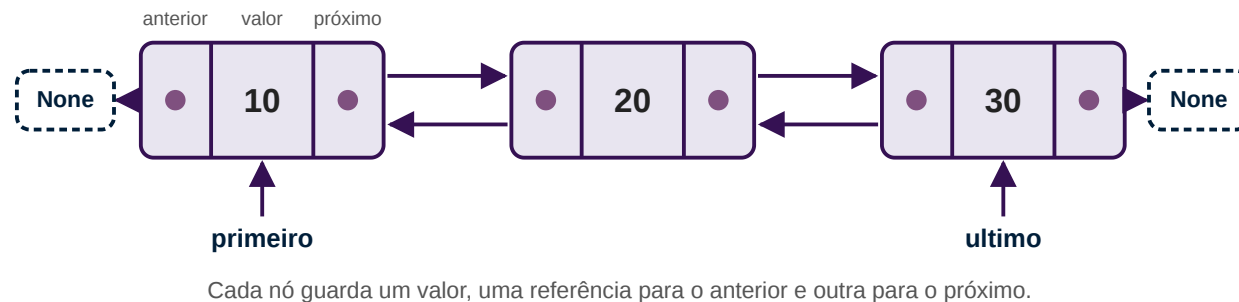
Estrutura de Dados - Professor Ramon Venson - SATC 2026.2

Lista Duplamente Encadeada

Estrutura de dados **linear e dinâmica** em que cada **nó** guarda um valor e **duas referências**: para o **próximo** e para o **anterior**.

Dessa forma, a lista pode ser percorrida nos **dois sentidos**.

Lista Duplamente Encadeada



Por que usar uma lista dupla?

A lista encadeada simples tem algumas limitações:

- Não é possível **voltar** de um nó para o seu anterior;
- **Remover o último elemento** exige percorrer toda a lista (**$O(n)$**);
- **Remover um nó conhecido** exige descobrir quem é o seu anterior (**$O(n)$**).

Com o ponteiro **anterior**, a lista dupla resolve esses casos. O custo é **memória extra**: cada nó guarda um ponteiro a mais.

O Nó Duplo

O **nó** agora possui dois ponteiros:

- **valor**: o dado armazenado;
- **proximo**: referência para o próximo nó (ou `None`);
- **anterior**: referência para o nó anterior (ou `None`).

```
classe No:  
  valor  
  proximo: No  
  anterior: No  
  
  construtor(valor):  
    este.valor <- valor  
    este.proximo <- None  
    este.anterior <- None
```

A Lista

Além do **primeiro**, a lista dupla também guarda a referência para o **último nó** — na literatura, esses ponteiros são chamados de *head* e *tail* — e a quantidade de elementos.

```
classe ListaDuplamenteEncadeada:  
    primeiro: No  
    ultimo: No  
    tamanho: inteiro  
  
    construtor():  
        este.primeiro <- None  
        este.ultimo <- None  
        este.tamanho <- 0
```

Quando `primeiro` é `None`, a lista está **vazia** (e `ultimo` também é `None`).

Verificando se a Lista está Vazia

```
esta_vazia():  
    retorne primeiro == None
```

A lista está vazia quando `primeiro` não aponta para nenhum nó.

Assim como na lista simples, **não existe** `esta_cheia()`: a lista cresce enquanto houver memória disponível.

Inserção no Início

Para inserir no início, o novo nó passa a apontar para o antigo `primeiro`, e o antigo `primeiro` passa a apontar de volta para o novo nó.

```
inserir_inicio(valor):  
    novo <- novo No(valor)  
    novo.proximo <- primeiro  
    se esta_vazia() então  
        ultimo <- novo  
    senao:  
        primeiro.anterior <- novo  
    primeiro <- novo  
    tamanho <- tamanho + 1
```

Inserir no início é **$O(1)$** , pois não depende do tamanho da lista.

Inserção no Final

Como a lista guarda o **ultimo**, inserir no final também é **O(1)**!

```
inserir_fim(valor):  
    novo <- novo No(valor)  
    se esta_vazia() então  
        primeiro <- novo  
    senao:  
        ultimo.proximo <- novo  
        novo.anterior <- ultimo  
    ultimo <- novo  
    tamanho <- tamanho + 1
```

Na lista simples, inserir no final era **O(n)**. O ponteiro **ultimo** elimina esse percurso.

Imprimindo os Elementos

Podemos percorrer a lista **do início para o fim**, usando `proximo` :

```
imprimir_frente():  
    atual <- primeiro  
    enquanto atual != None faça:  
        escreva(atual.valor)  
        atual <- atual.proximo
```

Imprimindo ao Contrário

E também **do fim para o início**, usando `anterior` :

```
imprimir_tras():  
    atual <- ultimo  
    enquanto atual != None faça:  
        escreva(atual.valor)  
        atual <- atual.anterior
```

O percurso continua sendo **$O(n)$** , mas agora existe nos dois sentidos.

Pesquisa de Elementos

A pesquisa é igual à da lista simples: percorremos os nós comparando os valores.

```
pesquisar(valor):  
    atual <- primeiro  
    enquanto atual != None faça:  
        se atual.valor == valor então  
            retorne atual  
        atual <- atual.proximo  
    retorne None
```

- **Melhor caso:** o elemento está no primeiro nó (**$O(1)$**).
- **Pior caso:** o elemento está no último nó ou não existe (**$O(n)$**).

Guardar os dois ponteiros **não** melhora a pesquisa: ainda pode ser necessário percorrer a lista.

Remoção de Elementos

Para remover, ligamos o **anterior** ao **próximo** do nó removido, cuidando dos casos em que o nó é o primeiro ou o último.

1. Se a lista estiver vazia, retornamos uma mensagem de erro.
2. Percorremos a lista até encontrar o nó a ser removido.
3. Se o nó não for encontrado, retornamos uma mensagem de erro.
4. Se o nó a ser removido for o `primeiro`, ajustamos o ponteiro `primeiro` para o próximo nó da lista.
5. Se o nó a ser removido for o `ultimo`, ajustamos o ponteiro `ultimo` para o nó anterior.
6. Caso contrário, ajustamos os ponteiros `proximo` e `anterior` dos nós vizinhos.

```
remover(valor):  
    se esta_vazia() então  
        retorne "Lista vazia"  
  
    atual <- primeiro  
    enquanto atual != None e atual.valor != valor faça:  
        atual <- atual.proximo  
  
    se atual == None então  
        retorne "Não encontrado"  
  
    se atual.anterior == None então # é o primeiro  
        primeiro <- atual.proximo  
    senao:  
        atual.anterior.proximo <- atual.proximo  
  
    se atual.proximo == None então # é o último  
        ultimo <- atual.anterior  
    senao:  
        atual.proximo.anterior <- atual.anterior  
  
    tamanho <- tamanho - 1  
    retorne "Removido"
```

Lista Simples vs Lista Dupla

Característica	Lista Simples	Lista Dupla
Ponteiros por nó	1 (proximo)	2 (proximo e anterior)
Memória por nó	Menor	Maior
Percurso	Só para frente	Nos dois sentidos
Inserir no início	O(1)	O(1)
Inserir no final	O(n)	O(1) com o ultimo
Remover no início	O(1)	O(1)
Remover no final	O(n)	O(1) com o ultimo
Remover um nó conhecido	O(n)	O(1)

Variações

- **Lista dupla circular:** o `proximo` do `ultimo` aponta para o `primeiro`, e o `anterior` do `primeiro` aponta para o `ultimo`;
- **Lista com sentinela:** um nó "cabeça" fictício que não guarda valor, usado para simplificar os casos de borda;
- Essas variações são comuns em bibliotecas e em linguagens como C++ (`std::list`) e Java (`LinkedList`).

Resumo

- A lista dupla é **dinâmica** e cada nó tem **valor**, **proximo** e **anterior**.
- Os ponteiros **primeiro** e **ultimo** apontam para as extremidades da lista; `None` indica o fim.
- O percurso pode ser feito nos **dois sentidos**.
- Com o **ultimo**, inserir e remover nas duas extremidades é **$O(1)$** .
- Pesquisar e percorrer continuam **$O(n)$** .
- O custo é **memória extra** para o ponteiro `anterior` .