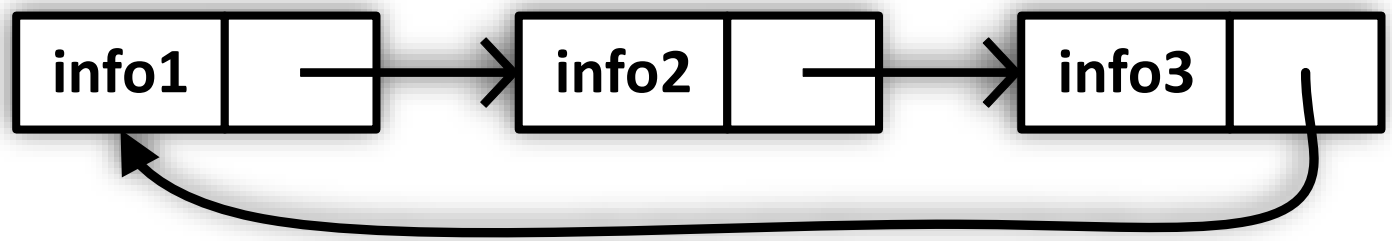
# Outras variações de listas encadeadas

# Outras variações de listas encadeadas

- Listas circulares
- Listas com acesso às duas extremidades
- Listas duplamente encadeadas

# Listas circulares

- O último nó da lista tem como "seu próximo" o primeiro nó da lista, formando um ciclo




- Benefício:
  - Maior legibilidade em algoritmos que percorrem infinitamente uma estrutura de dados

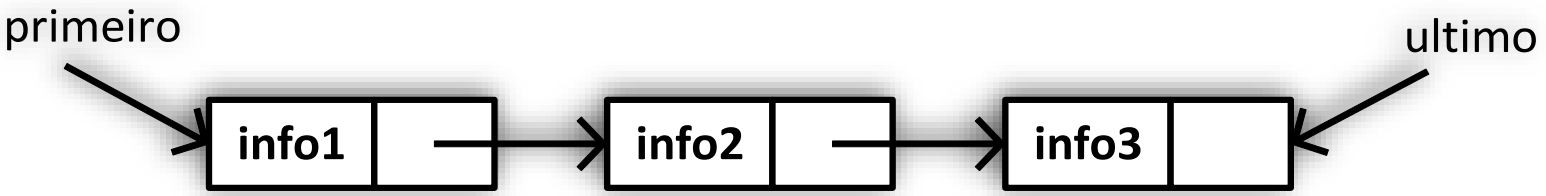
# Listas circulares
## Impacto nos algoritmos

- Necessário escrever novos algoritmo para impedir execução infinita
- Exemplo: Exibir o conteúdo da lista

```
Algoritmo: exibirCircular()

NoLista p ← primeiro;

se (p ≠ null) então
  repita
    escrever(p.info);
    p  ← p.proximo;
  enquanto (p ≠ primeiro);
fim-se;
```

- Percorre todos os nós a partir da referência do nó inicial até alcançar novamente este mesmo nó
- Se a lista é vazia, a referência para o nó inicial é nula

# Listas com acesso as duas extremidades



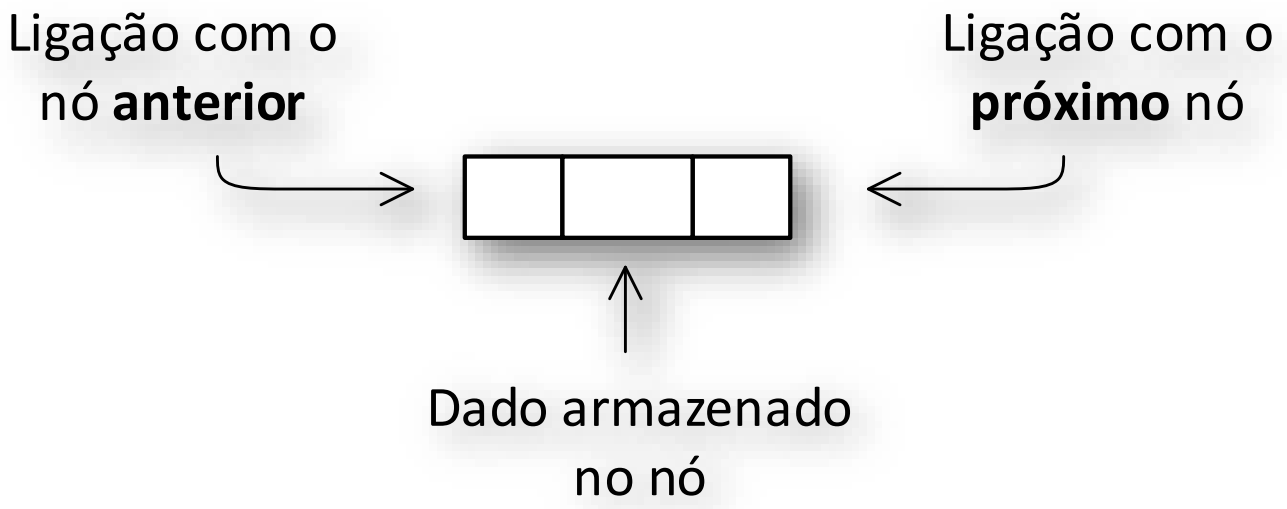

- Benefício:
  - Permite estabelecer outra ordem de inclusão de elementos

# Listas duplamente encadeadas

FURB
UNIVERSIDADE DE BLUMENAU

# Lista duplamente encadeada

- Cada nó possui uma referência para o próximo nó e para o nó anterior



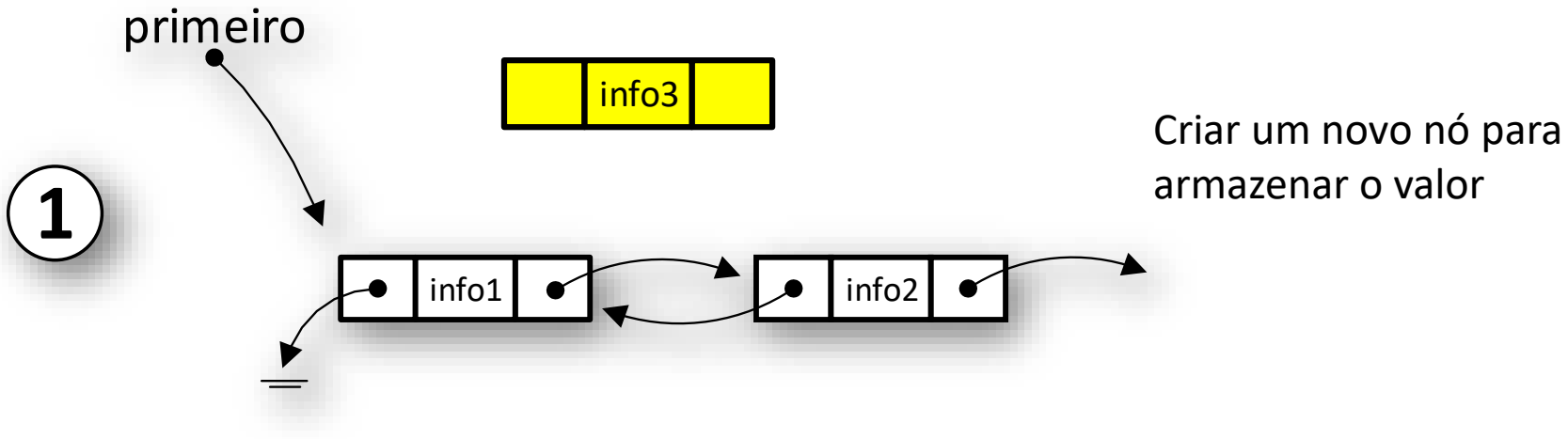
# Lista duplamente encadeada

- Exemplo em Java para representar um nó da lista duplamente encadeada

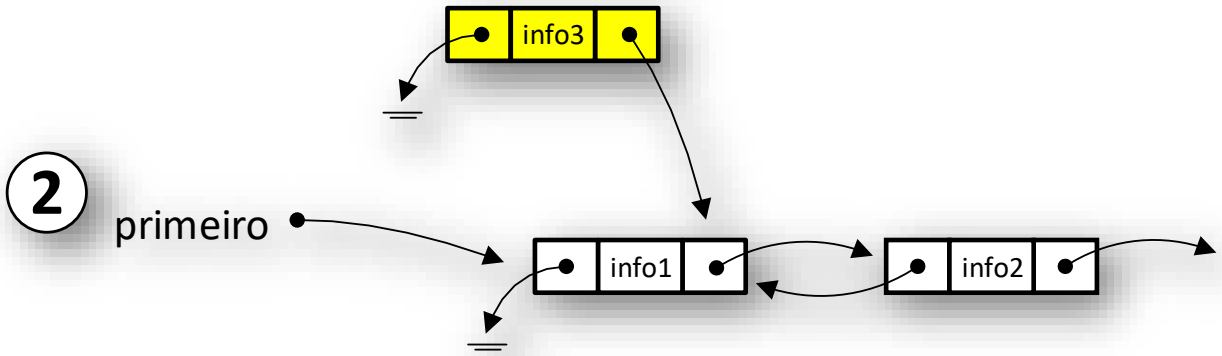
```
public class NoListaDupla {

    private int info;
    private NoListaDupla proximo;
    private NoListaDupla anterior;

}
```

# Incluir valor na lista duplamente encadeada

- Necessário:
  - Incluir novo objeto **`NoListaDupla`**
  - Encadear o novo nó no início da lista existente

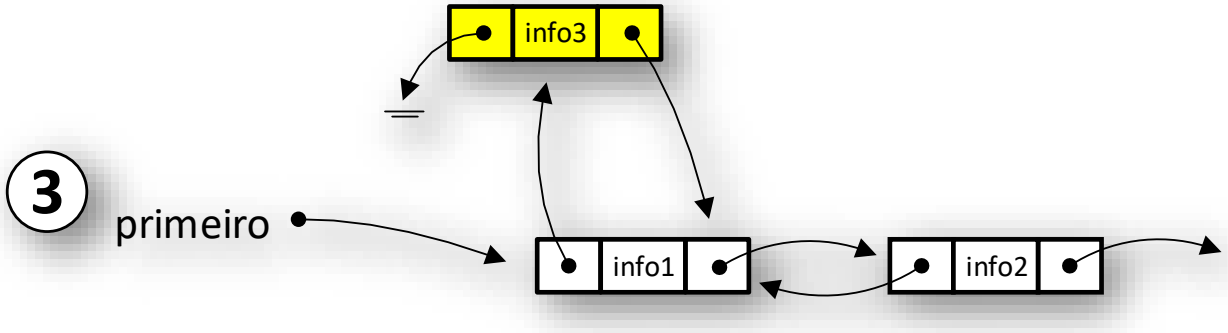


# Incluir valor na lista duplamente encadeada



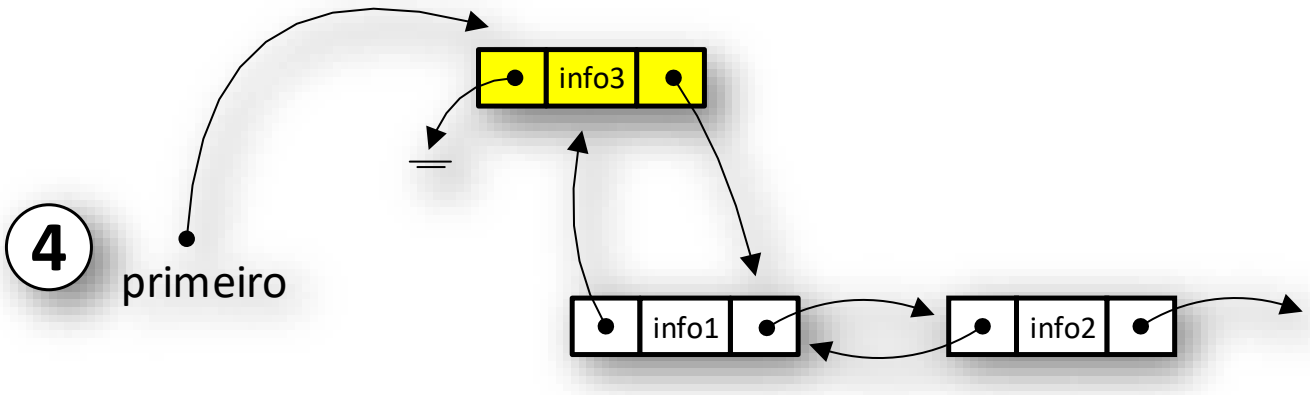

- Próximo do *novo nó* aponta para o primeiro
- Anterior do *novo nó* aponta para null

---




- *Primeiro nó* aponta para *novo nó,* como seu anterior

# Incluir valor na lista duplamente encadeada

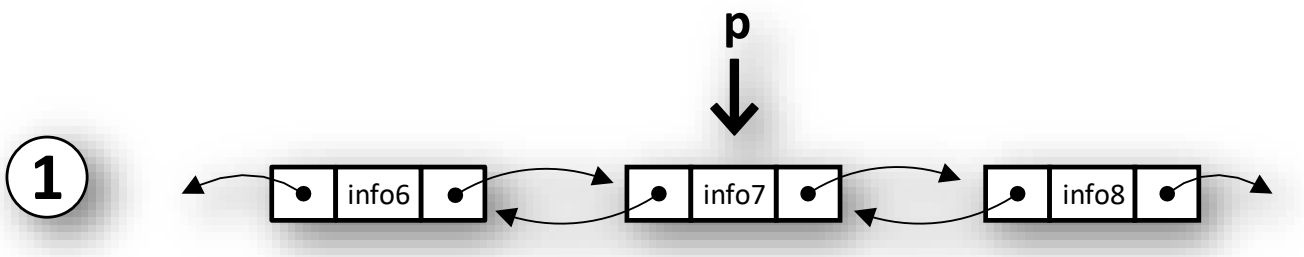



- *Primeiro* aponta para novo nó

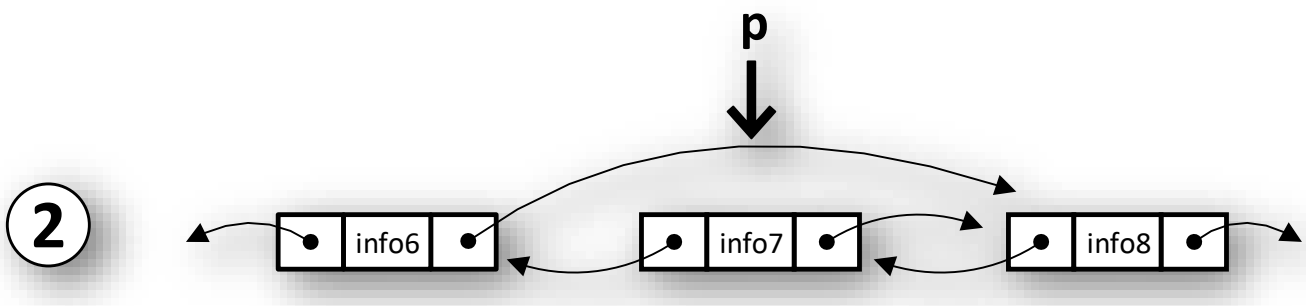
**Algoritmo: inserir(int valor)**

```
NoListaDupla novo ← new NoListaDupla();
novo.info ← valor;
novo.proximo ← primeiro;
novo.anterior ← null;
se (primeiro ≠ null) então
   primeiro.anterior  ← novo;
fim-se
primeiro ← novo;
```

# Retirar valor da lista duplamente encadeada



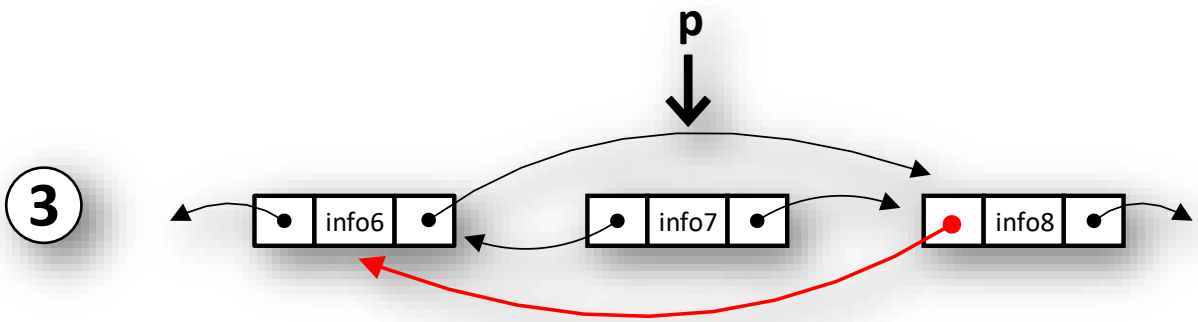

Localizar o nó a ser removido, apontado por “p”




O nó antecessor à *p* deve apontar (como próximo) para o próximo de *p*

```
p.anterior.proximo ← p.proximo;
```

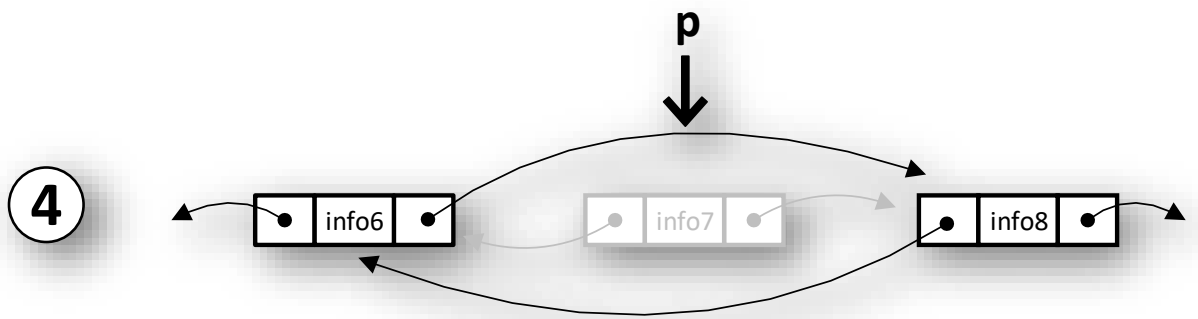
# Retirar valor da lista duplamente encadeada




O nó posterior à *p* deve apontar (como anterior) para o anterior de *p*

```
p.proximo.anterior ← p.anterior
```

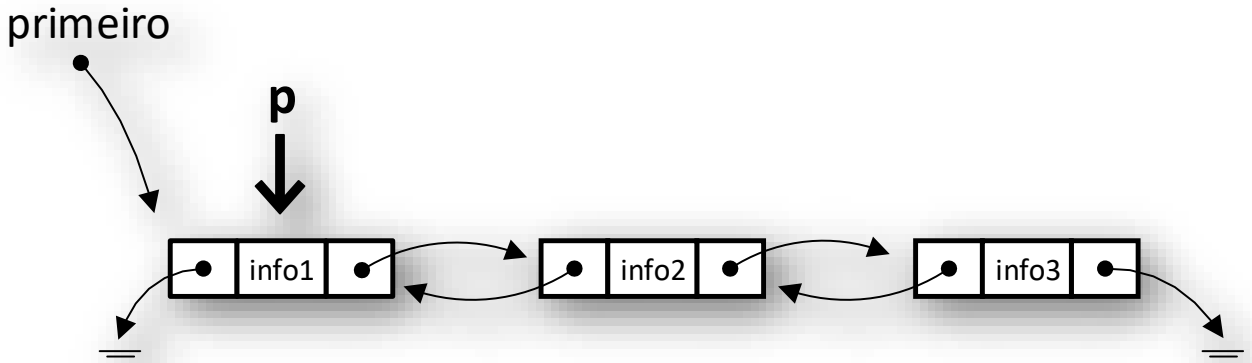
---




Nenhum outro nó da lista aponta para *p*

# Caso especial 1

## Exclusão do primeiro nó da lista




- Não é possível executar

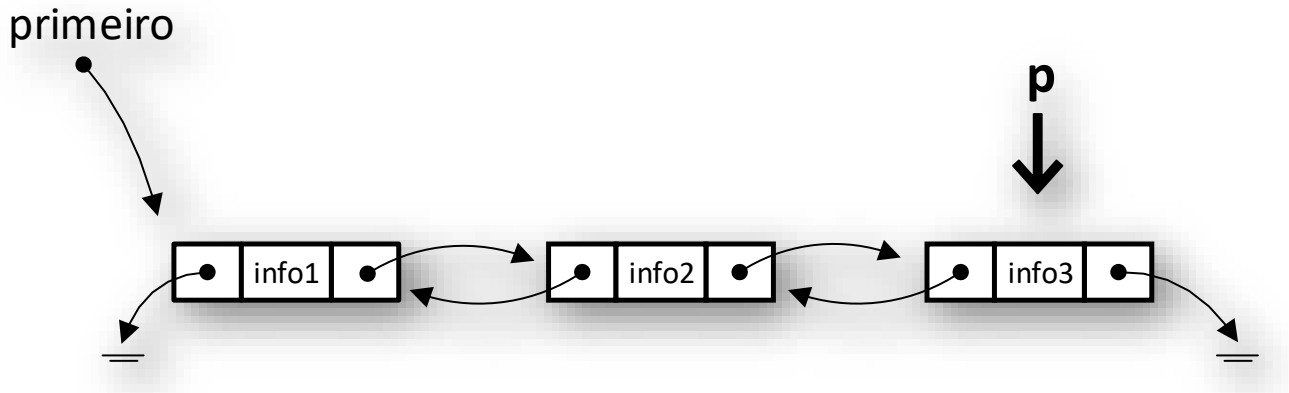
```
p.anterior.proximo ← p.proximo;
```

  Pois `p.anterior` é igual à `null`

- Neste caso, o “novo primeiro” é o próximo nó daquele que foi removido

# Caso especial 2

## Exclusão do último nó da lista




- Não é possível executar

  `p.proximo.anterior` ← `p.anterior`

  Pois `p.proximo` é `null`

# Retirar valor da lista duplamente encadeada

**Algoritmo: retirar(int valor)**

```
NoListaDupla p ← buscar(valor);

se (p ≠ null) então // achou
   se (primeiro = p) então // primeiro elemento ?
      primeiro ← p.proximo;
   senão
      p.anterior.proximo ← p.proximo;
   fim-se

   se (p.proximo ≠ null) então // não é o último?
      p.proximo.anterior ← p.anterior;
   fim-se
fim-se
```