

Listas Circulares

As listas circulares são uma estrutura de dados em que os elementos são organizados em um circuito fechado, de modo que o último elemento está conectado ao primeiro, formando uma estrutura circular. Essa estrutura pode ser implementada em listas estáticas, com um tamanho fixo, ou em listas dinâmicas, com um tamanho variável.

As listas circulares são usadas em situações em que a ordem dos elementos é importante e precisa ser mantida, mas em que também é necessário que a lista seja acessada de forma circular, ou seja, sem que a primeira ou a última posição sejam uma barreira para a navegação pelos elementos. Isso é útil em diversos contextos, como em aplicações gráficas, sistemas de navegação, jogos, dentre outros.

As listas circulares permitem que a lista seja percorrida de forma circular, acessando cada elemento em ordem, independentemente de sua posição no vetor, porém deve-se ter muito cuidado pois elas podem apresentar problemas de circularidade infinita, caso não sejam implementadas corretamente.

Listas Estáticas Circulares

Uma lista estática sequencial circular é uma estrutura de dados que consiste em um vetor de tamanho fixo, onde os elementos são armazenados em sequência, permitindo a inserção e remoção de elementos em qualquer posição. A principal diferença entre uma lista estática sequencial circular e uma lista sequencial linear é que, na lista circular, a posição de inserção e remoção pode ser qualquer posição, mas a lista continua circular, ou seja, a última posição da lista é conectada à primeira posição.

A utilidade da lista estática sequencial circular está em permitir a manipulação dos elementos da lista de forma eficiente, com complexidade de tempo constante para inserção e remoção no início e no final da lista. Além disso, a lista circular evita o desperdício de espaço, já que os elementos podem ser inseridos e removidos em qualquer posição sem deixar espaços vazios.

A implementação da lista estática sequencial circular envolve o uso de um vetor de tamanho fixo e variáveis adicionais para controlar a posição do primeiro e do último elemento da lista. Quando a lista está vazia, as variáveis de posição apontam para a mesma posição no vetor, indicando que a lista é circular. À medida que novos elementos são inseridos ou removidos, as variáveis de posição são atualizadas de forma circular, mantendo a estrutura circular da lista.

Sua implementação é auxiliada pela utilização do operador módulo (%) na linguagem de programação. Esse operador permite que a posição da lista seja calculada de forma circular porque ele retorna o resto da divisão entre dois números. Em outras palavras, ao realizar a operação $(i + 1) \% \text{tam_maximo}$, o resultado é o resto da divisão entre $i + 1$ e tam_maximo , o que garante que o valor retornado será sempre um número entre 0 e $\text{tam_maximo} - 1$, de forma circular, garantindo que a lista sempre mantenha sua estrutura circular, mesmo após muitas inserções e remoções.

Exemplificando, suponha que temos uma lista com tamanho máximo igual a 5, ou seja, $\text{tam_maximo} = 5$. Vamos considerar que a posição atual é a posição 4 e queremos encontrar a próxima posição de forma circular. Para calcular essa posição, podemos utilizar o operador módulo %. Se somarmos 1 à posição atual ($4 + 1 = 5$), teríamos um valor maior que o tamanho máximo da lista. Mas ao utilizar o operador %, podemos calcular a posição de forma circular, ou seja, o valor retornado será o resto da divisão entre 5 e 5, que é igual a 0.

Mais um exemplo: digamos que a posição atual fosse a posição 3 e quiséssemos encontrar a próxima posição circular, poderíamos realizar a operação $(3 + 1) \% 5$, que retorna o resto da divisão entre 4 e 5, que é igual a 4. Dessa forma, conseguimos calcular a posição de forma circular, ainda que seja um simples “passo à frente”, a utilização do operador mantém sua correta execução.

Assim, quando um novo elemento é adicionado ao final da lista, ele é inserido na posição `fim` do vetor, que é a posição seguinte à última posição ocupada na lista, calculada como $(\text{fim} + 1) \% \text{tam_maximo}$, onde tam_maximo é o tamanho máximo da lista. Se $\text{fim} + 1$ for maior que tam_maximo , a operação % retorna o resto da divisão, permitindo que a lista continue circular. Dessa forma, a próxima posição após a última posição da lista é a primeira posição do vetor, permitindo que novos elementos sejam adicionados no início da lista após o último elemento existente.

Por exemplo, se a última posição da lista estiver na posição 4 e você quiser adicionar um novo elemento, ele será adicionado na posição seguinte, que é a posição 0, calculada como $(\text{fim} + 1) \% \text{tam_maximo}$. Isso é possível graças à estrutura circular da lista, que permite que a próxima posição seja a primeira posição do vetor.

Para remover um elemento do início da lista, a posição `inicio` é atualizada para a próxima posição circular da lista. Por exemplo, se o primeiro elemento da lista está na posição 4 e você o remove, a próxima posição do início da lista é a posição 0, pois a lista é circular. Isso permite que

a lista possa ser percorrida de forma circular, acessando cada elemento em ordem, independentemente de sua posição no vetor.

Para dar início à implementação de nossa lista estática sequencial circular, na nossa classe `Lescirc`, vamos primeira considerar algumas modificações em relação à lista estática sequencial (a classe `Les`) já vista. Para conhecê-la vamos considerar as alterações necessárias nos seus atributos, e consequentemente no seu construtor, e nas suas funções básicas: `inserir_inicio`, `inserir_fim`, `remover_inicio`, `remover_fim`, `ver_primeiro`, `ver_ultimo` e `show`. Com o entendimento delas você terá condições de implementar todas as demais.

Desta forma, a lista circular será implementada com a adição dos atributos `inicio` e `fim`, que indicam as posições do primeiro e, tenham atenção aqui, a posição posterior ao último elemento da lista, respectivamente. O atributo `fim` representa a posição onde será inserido o próximo elemento. Ou seja, se não houver elementos na lista, o `fim` será a mesma posição do `inicio`, que representa a posição onde seria inserido o primeiro elemento. Quando um elemento é inserido, ele ocupa a posição `fim` e, em seguida, o `fim` é atualizado para a próxima posição circular do vetor.

Esses atributos são atualizados a cada inserção ou remoção de elementos.

```
def __init__(self, tamanho):  
    self.tam_maximo = tamanho  
    self.vetor = [None] * tamanho  
    self.inicio = 0  
    self.fim = 0  
    self.quant = 0
```

As funções `inserir_fim` e `remover_fim` originais são, então, modificadas para atualizar a posição do último elemento da lista.

Na função `inserir_fim`, o novo elemento é adicionado na posição `fim` e a posição posterior ao último elemento é atualizada para a próxima posição circular do vetor, que é calculada como $(fim + 1) \% tam_maximo$.

```
def inserir_fim(self, valor):  
    self.vetor[self.fim] = valor  
    self.fim = (self.fim + 1) % self.tam_maximo
```

```
self.quant += 1
```

Suponha que a lista tenha tamanho máximo igual a 5, ou seja, `tam_maximo = 5`. Considere que a lista contém três elementos, ocupando as posições 2, 3 e 4 do vetor, e que `quant = 3`, `inicio = 2` e `fim = 0` (lembrando que as posições são circulares):

0	1	2	3	4
None	None	A	B	C

`tam_maximo = 5`

`quant = 3`

`inicio = 2`

`fim = 0`

Ao chamar a função `inserir_fim(valor)` o valor 'D' é inserido na posição `fim` do vetor, que é a posição 0, e `fim` é atualizado para a próxima posição circular do vetor, que é a posição 1 (calculada como $(fim + 1) \% tam_maximo$):

0	1	2	3	4
D	None	A	B	C

`tam_maximo = 5`

`quant = 4`

`inicio = 2`

`fim = 0`

Por fim, a quantidade de elementos da lista (`quant`) é incrementada para 4.

Já na função `remover_fim`, a posição do último elemento é atualizada para a posição circular anterior, que é calculada como $(fim - 1) \% tam_maximo$:

```
def remover_fim(self):  
    self.fim = (self.fim - 1) % self.tam_maximo  
    self.quant -= 1
```

Suponha que a lista tenha tamanho máximo igual a 5 e que a lista contenha três elementos, ocupando as posições 3, 4 e 0 do vetor, e que `quant = 3`, `inicio = 3` e `fim = 1` (lembrando mais uma vez que as posições são circulares):

0	1	2	3	4
C	None	None	A	B

`tam_maximo = 5`

```
quant = 3
inicio = 3
fim = 1
```

Ao chamar a função `remover_fim()`, a posição do último elemento é atualizada para a posição circular anterior, que é $(\text{fim} - 1) \% \text{tam_maximo}$, ou seja, a posição 0. Por fim, a quantidade de elementos da lista (`quant`) é decrementada para 2:

0	1	2	3	4
C	None	None	A	B

```
tam_maximo = 5
quant = 2
inicio = 3
fim = 0
```

As funções `inserir_inicio` e `remover_inicio` também são modificadas para atualizar a posição do primeiro elemento da lista.

Na função `inserir_inicio`, o novo elemento é adicionado na posição `inicio` e a posição do primeiro elemento é atualizada para a posição circular anterior, que é calculada como $(\text{inicio} - 1) \% \text{tam_maximo}$.

```
def inserir_inicio(self, valor):
    self.inicio = (self.inicio - 1) % self.tam_maximo
    self.vetor[self.inicio] = valor
    self.quant += 1
```

Vamos trabalhar com uma lista que tem tamanho máximo igual a 5 e que contém três elementos, ocupando as posições 3, 4 e 0 do vetor, e que `quant = 3`, `inicio = 3` e `fim = 1` (sempre lembrando que as posições são circulares):

0	1	2	3	4
C	None	None	A	B

```
tam_maximo = 5
quant = 3
inicio = 3
fim = 1
```

Ao chamar a função `inserir_inicio('D')`, a posição do primeiro elemento é atualizada para a posição circular anterior, que é $(\text{inicio} - 1) \% \text{tam_maximo}$, ou seja, a posição 2. Em seguida, o novo elemento é inserido na posição inicial da lista. Por fim, a quantidade de elementos da lista (`quant`) é incrementada em 1:

0	1	2	3	4
C	None	D	A	B

```

tam_maximo = 5
quant = 3
inicio = 2
fim = 1

```

Na função `remover_inicio`, a posição do primeiro elemento é atualizada para a próxima posição circular do vetor, que é calculada como $(\text{inicio} + 1) \% \text{tam_maximo}$:

```

def remover_inicio(self):
    self.inicio = (self.inicio + 1) % self.tam_maximo
    self.quant -= 1

```

Suponha que a lista tenha tamanho máximo igual a 5 e que a lista contenha três elementos, ocupando as posições 2, 3 e 4 do vetor, e que `quant = 3`, `inicio = 2` e `fim = 0` (lembrando que as posições são circulares):

0	1	2	3	4
None	None	D	A	B

```

tam_maximo = 5
quant = 3
inicio = 2
fim = 0

```

Ao chamar a função `remover_inicio()`, a posição do primeiro elemento é atualizada para a próxima posição circular do vetor, que é $(\text{inicio} + 1) \% \text{tam_maximo}$, ou seja, a posição 3, não havendo necessidade de deslocamento dos elementos, já que a lista é circular. Em seguida, a quantidade de elementos da lista é decrementada em 1.

0	1	2	3	4
None	None	D	A	B

```

tam_maximo = 5

```

```
quant = 2
inicio = 3
fim = 0
```

Da mesma forma, a função `show` é modificada para percorrer a lista circularmente, usando a mesma estratégia para andar pela lista de forma circular. Para isso, começa definindo a variável `aux` com o valor da posição do primeiro elemento da lista (`inicio`). Em seguida, ele entra em um laço `while` que é executado enquanto `aux` não for igual à posição posterior ao último elemento da lista (`fim`). Dentro do laço, o código imprime na tela o elemento que está na posição `aux` da lista, e depois atualiza o valor de `aux` para a próxima posição circular da lista, através do comando `i = (aux + 1) % self.tam_maximo`:

```
def show(self):
    aux = self.inicio
    while aux!=self.fim:
        print(self.vetor[aux], end=' ')
        aux = (aux + 1) % self.tam_maximo
    print()
```

As funções `ver_primeiro` e `ver_ultimo` retornam o primeiro e o último elemento da lista, respectivamente, utilizando as posições `inicio` e `fim`. A função `ver_primeiro` retorna o valor do primeiro elemento da lista, que está na posição `inicio`. A função `ver_ultimo` retorna o valor do último elemento da lista, que está na posição circular anterior a `fim`, que é calculada como `fim - 1) % tam_maximo`:

```
def ver_primeiro(self):
    return self.vetor[self.inicio]

def ver_ultimo(self):
    return self.vetor[(self.fim - 1) % self.tam_maximo]
```

Juntando tudo, nosso código fica assim:

```
class Lescircular:
    def __init__(self, tamanho):
        self.tam_maximo = tamanho
```

```
self.vetor = [None] * tamanho
self.inicio = 0
self.fim = 0
self.quant = 0

def inserir_fim(self, valor):
    self.vetor[self.fim] = valor
    self.fim = (self.fim + 1) % self.tam_maximo
    self.quant += 1

def remover_fim(self):
    self.fim = (self.fim - 1) % self.tam_maximo
    self.quant -= 1

def inserir_inicio(self, valor):
    self.inicio = (self.inicio - 1) % self.tam_maximo
    self.vetor[self.inicio] = valor
    self.quant += 1

def remover_inicio(self):
    self.inicio = (self.inicio + 1) % self.tam_maximo
    self.quant -= 1

def esta_vazia(self):
    return self.quant == 0

def esta_cheia(self):
    return self.quant == self.tam_maximo

def show(self):
    for i in range(self.quant):
        print(self.vetor[(self.inicio + i) % self.tam_maximo], end=' ')
    print()

def ver_primeiro(self):
    return self.vetor[self.inicio]

def ver_ultimo(self):
    return self.vetor[(self.fim - 1) % self.tam_maximo]
```


Lista Dinâmica Simplesmente Encadeada Circular

Na implementação de uma lista dinâmica simplesmente encadeada circular, ao contrário de uma lista simplesmente encadeada convencional, o último nó aponta para o primeiro nó, criando assim um ciclo que pode ser percorrido de forma circular. Essa implementação é feita sem a necessidade de alterações no construtor do nó ou da lista. Todos os atributos se mantêm os mesmos.

```
class No:

    def __init__(self, valor, proximo):
        self.info = valor
        self.prox = proximo

class Ldse:

    def __init__(self):
        self.prim = self.ult = None
        self.quant = 0
```

A construção da ideia de ser uma lista circular é obtida através da implementação cuidadosa das funções de inserção e remoção, que garantem que a lista permaneça circular mesmo após muitas operações. Dessa forma, seria possível percorrer a lista em um loop infinito, de forma que o último nó aponte para o primeiro, mantendo a estrutura circular.



Podemos ver isso na implementação da função `inserir_inicio`. Para isso, como já falamos aqui, precisamos usar o desenho para nos ajudar a construir o código. Assim, começamos com uma lista vazia:

`prim = None` `ult = None`


 

Para inserir um valor precisamos construir o nó:

`prim = None` `ult = None`

info	prox
A	None

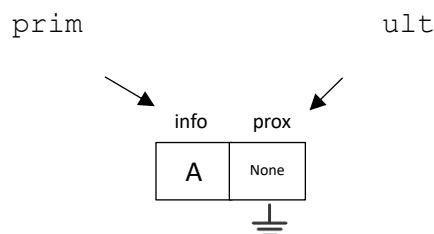


Na implementação de uma lista dinâmica simplesmente encadeada circular, é necessário garantir que o último nó aponte para o primeiro nó, criando um ciclo. Para isso, podemos pensar em inicializar o primeiro e o último nó com o mesmo valor, ou seja, `self.prim = self.ult = No(valor, self.prim)`.

No entanto, é importante lembrar que no momento da criação do primeiro nó, `self.prim` ainda não está definido, portanto, fica melhor de visualizarmos e evitarmos cair em erro realizarmos a inicialização do nó utilizando `None` para o valor do atributo `prox`, como representado na figura acima e que se reflete no código abaixo:

```
No(valor, None)
```

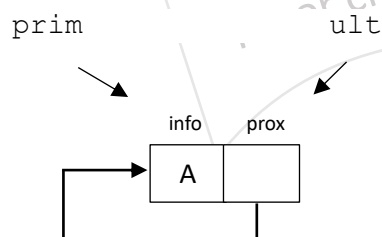
Agora, precisamos fazer `prim` e `ult` apontar para este nó recém-criado.



Que se reflete nas atribuições realizadas no código:

```
self.prim = self.ult = No(valor, None)
```

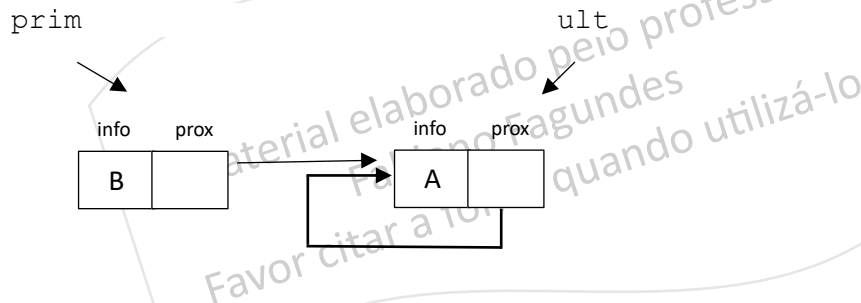
Por fim, precisamos atualizar o atributo `prox` do último nó para apontar para o primeiro nó, o que é feito com a instrução `self.ult.prox = self.prim`.



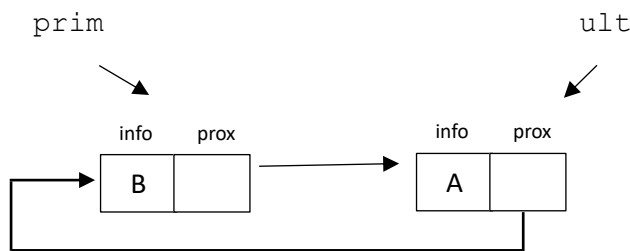
Assim, temos o código correto para inserir o primeiro elemento em nossa lista quando ela estiver vazia:

```
if self.quant == 0:
    self.prim = self.ult = No(valor, None)
    self.ult.prox = self.prim
```

Caso a lista já tenha elementos, a inserção no início é feita, da forma tradicional, criando-se um novo nó com o valor desejado e fazendo-o apontar para o nó que atualmente é o primeiro da lista e, em seguida, atualizando o atributo `self.prim` para apontar para o novo nó.



Agora, para manter a lista circular, o atributo `prox` do último nó da lista é atualizado para apontar para o novo nó.



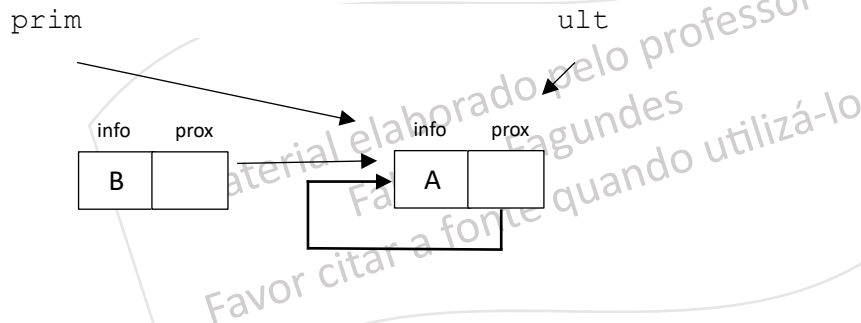
Assim, teremos o código da função `inserir_inicio` da seguinte forma:

```
def inserir_inicio(self, valor):
    if self.quant == 0:
        self.prim = self.ult = No(valor, None)
        self.ult.prox = self.prim
    else:
        self.prim = No(valor, self.prim)
        self.ult.prox = self.prim
    self.quant += 1
```

Na implementação da função `remover_inicio`, assim como na função `inserir_inicio`, devemos tomar cuidado com a circularidade da lista.

Da mesma forma que na lista dinâmica simplesmente encadeada, se a lista tiver apenas um elemento, devemos setar `self.prim` e `self.ult` como `None`, indicando que a lista está vazia.

Agora, se a lista tiver mais de um elemento, basta atualizar `self.prim` para o segundo nó da lista, ou seja, `self.prim = self.prim.prox`. Além disso, é preciso atualizar o último nó para apontar para o novo primeiro nó da lista, garantindo a circularidade, o que pode ser feito com `self.ult.prox = self.prim`.

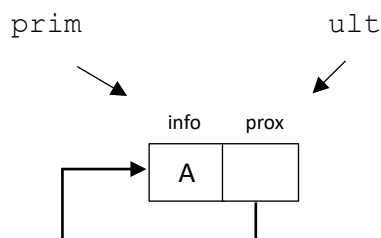


Segue o código da função `remover_inicio`:

```
def remover_inicio(self):  
    if self.quant == 1:  
        self.prim = self.ult = None  
    else:  
        self.prim = self.prim.prox  
        self.ult.prox = self.prim  
    self.quant -= 1
```

De forma geral, atualizamos `self.prim` para o segundo nó da lista, indicado por `self.prim.prox`. Em seguida, atualizamos o último nó, `self.ult.prox`, para apontar para o novo primeiro nó da lista, `self.prim`. Dessa forma, a lista continua circular. Por fim, decrementamos a quantidade de elementos `quant` em 1. Lembremos da função do *garbage collector* aqui, de devolver o nó não referenciado para a memória.

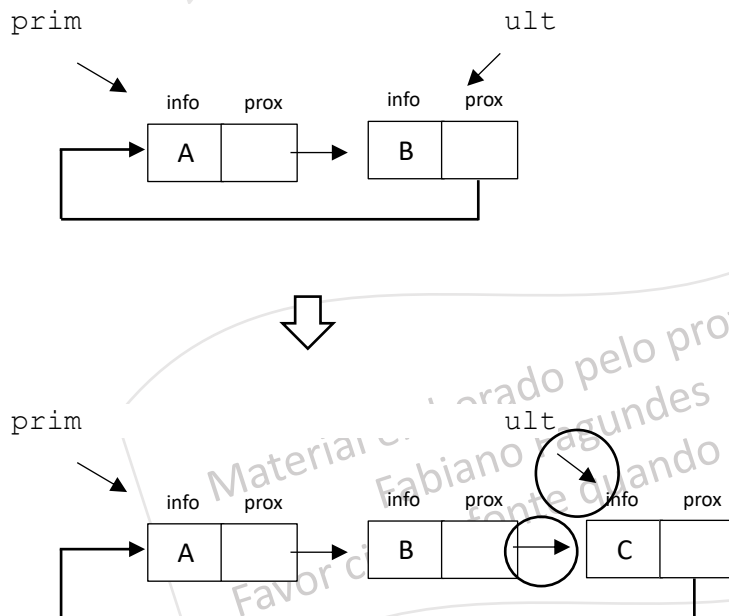
A implementação da função `inserir_fim` inicia-se da mesma forma que a função `inserir_inicio`. Consideremos a figura a seguir:



Como a figura acima apresenta, no caso de lista vazia o novo elemento será tanto o primeiro quanto o último, e seu atributo `prox` apontará para ele mesmo, criando uma estrutura circular.

```
if self.quant == 0:
    self.prim = self.ult = No(valor, None)
    self.ult.prox = self.prim
```

Caso contrário, o novo elemento é criado, com seu atributo `prox` apontando para o primeiro elemento da lista, para manter a circularidade. Depois, ele é inserido na última posição da lista, ou seja, o atributo `prox` do antigo último elemento aponta para o novo elemento e, por fim, atualizamos o ponteiro do último elemento para este novo elemento inserido.



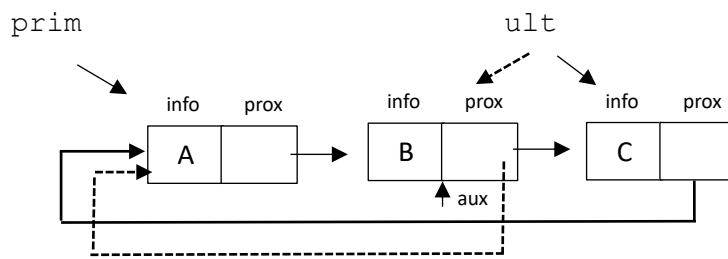
Com isso, temos o código da função `inserir_fim` da seguinte forma:

```
def inserir_fim(self, valor):
    if self.quant == 0:
        self.prim = self.ult = No(valor, None)
        self.ult.prox = self.prim
    else:
        self.ult.prox = No(valor, None)
        self.ult = self.ult.prox
    self.quant += 1
```

Aproveitando todo este entendimento sobre o que acontece o fim da lista, vamos agora implementar a função `remove_fim`.

Da mesma forma que a função `remove_fim` da lista dinâmica simplesmente encadeada, primeiro verificamos se só há um elemento. Neste caso, devemos setar `self.prim` e `self.ult` como `None`, indicando que a lista estará vazia.

Caso contrário, havendo mais de um elemento, é necessário percorrer a lista até o penúltimo nó, atualizando o atributo `prox` deste nó para quem `ult` está apontando, ou seja, para `prim`, pois devemos manter a circularidade da lista. Em seguida, o último nó é atualizado para o penúltimo nó encontrado e a quantidade de elementos é decrementada. Na figura temos a variável auxiliar indicando este penúltimo nó e, em linhas tracejadas, as alterações que devem ser realizadas.



Assim, temos o código da função `remove_fim` a seguir:

```
def remove_fim(self):
    if self.quant == 1:
        self.prim = self.ult = None
    else:
        aux = self.prim
        while aux.prox != self.ult:
            aux = aux.prox
        aux.prox = self.ult.prox # poderia ser self.prim
        self.ult = aux
        self.quant -= 1
```

Primeiro, a função verifica se a lista possui apenas um elemento, ou seja, se o valor da variável `quant` é igual a 1. Se for esse o caso, então a lista fica vazia, e a função simplesmente atribui `None` aos atributos `prim` e `ult`.

Caso contrário, a função precisa percorrer a lista até encontrar o penúltimo elemento. Para fazer isso, é criada uma variável `aux` que começa apontando para o primeiro elemento da lista (`prim`). Em seguida, ela percorre a lista enquanto o elemento apontado por `aux.prox` não for o último elemento (`ult`). Isso significa que `aux` terminará apontando para o penúltimo elemento da lista.

Então, a função atualiza o atributo `prox` do penúltimo elemento para apontar para o elemento seguinte ao último elemento (`ult.prox`) que, lembremos, é o `prim`, dada a circularidade da lista. Em seguida, ela atualiza o valor da variável `ult` para que aponte para o penúltimo elemento. Por fim, ela decrementa a variável `quant` para refletir a remoção do último elemento da lista.

O resultado final é que o antigo último elemento é desligado da lista, e a lista continua sendo circular, pois o novo último elemento agora aponta para o primeiro elemento da lista (`prim`).

Para vermos o resultado destas implementações precisamos da função `show`. Porém, agora, não temos o `None` como critério de parada ao percorrer toda a lista, e também não podemos considerar a hipótese de usar `aux.prox != self.prim` para isso, pois correríamos o risco de não imprimir o primeiro elemento caso ele seja o único da lista.

Então, podemos nos aproveitar da quantidade de elementos da lista, armazenada no atributo `quant`, para fazer este percurso. Assim, temos o seguinte código:

```
def show(self):
    aux = self.prim
    for i in range(self.quant):
        print(aux.info, end=' ')
        aux = aux.prox
    print('\n')
```

Nesta implementação a variável `aux` é inicializada como sendo o primeiro nó da lista (`self.prim`) e é utilizado um laço de repetição `for`, que percorre a lista de acordo com a quantidade de elementos (`self.quant`). Para cada iteração do laço, o valor do atributo `info` do nó atual é impresso na tela, seguido de um espaço em branco, e a variável `aux` é atualizada para o próximo nó (`aux.prox`).

Por fim, temos as classes da `Ldsecirc` e suas funções básicas implementadas a seguir:

```

class No:
    def __init__(self, valor, proximo):
        self.info = valor
        self.prox = proximo

class Ldsecirc:
    def __init__(self):
        self.prim = self.ult = None
        self.quant = 0

    def inserir_inicio(self, valor):
        if self.quant == 0:
            self.prim = self.ult = No(valor, None)
            self.ult.prox = self.prim
        else:
            self.prim = No(valor, self.prim)
            self.ult.prox = self.prim
        self.quant += 1

    def remover_inicio(self):
        if self.quant == 1:
            self.prim = self.ult = None
        else:
            self.prim = self.prim.prox
            self.ult.prox = self.prim
        self.quant -= 1

    def inserir_fim(self, valor):
        if self.quant == 0:
            self.prim = self.ult = No(valor, None)
            self.ult.prox = self.prim
        else:
            self.ult.prox = No(valor, None)
            self.ult = self.ult.prox

```



```
        self.ult.prox = self.prim
    self.quant += 1

def remover_fim(self):
    if self.quant == 1:
        self.prim = self.ult = None
    else:
        aux = self.prim
        while aux.prox != self.ult:
            aux = aux.prox
        aux.prox = self.ult.prox
        self.ult = aux
    self.quant -= 1

def show(self):
    aux = self.prim
    for i in range(self.quant):
        print(aux.info, end=" ")
        aux = aux.prox
    print("\n")

def estaVazia(self):
    return self.quant == 0

def tamanho_atual(self):
    return self.quant

def ver_primeiro(self):
    return self.prim.info

def ver_ultimo(self):
    return self.ult.info
```

Lista dinâmica duplamente encadeada circular

Uma lista dinâmica duplamente encadeada circular é uma estrutura de dados na qual cada nó tem um ponteiro para o nó anterior e outro para o próximo nó. Além disso, o último nó da lista aponta para o primeiro nó, e o nó anterior ao primeiro aponta para o último, formando assim um ciclo. Isso permite que a lista seja percorrida indefinidamente, sempre retornando ao início.

Para implementar essa estrutura de dados, não é necessário alterar o construtor do nó ou da lista em relação à lista dinâmica duplamente encadeada original. Todos os atributos permanecem os mesmos. A circularidade da lista é garantida apenas por meio da atualização dos ponteiros do último nó e do nó anterior ao primeiro. O ponteiro "anterior" do primeiro nó deve apontar para o último nó da lista, enquanto o ponteiro "próximo" do último nó deve apontar para o primeiro nó da lista. Com isso, a lista pode ser percorrida indefinidamente, em qualquer direção, mantendo sua estrutura circular.

```
class No:

    def __init__(self, valor, proximo):
        self.info = valor
        self.prox = proximo
```

```
class Ldse:

    def __init__(self):
        self.prim = self.ult = None
        self.quant = 0
```

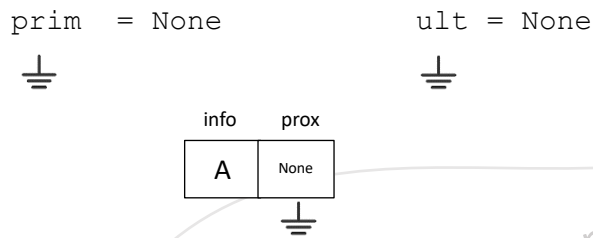
Da mesma forma que na lista simplesmente encadeada, aqui a construção de uma lista circular é obtida através da implementação das funções de inserção e remoção, que garantem que a lista permaneça circular.

Vemos isso na implementação da função `inserir_inicio`. Novamente, vamos usar o desenho para nos ajudar a construir o código. Assim, começamos com uma lista vazia:

```
prim = None          ult = None
```

Para inserir um valor precisamos construir o nó:

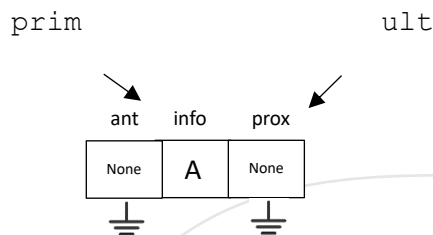


Precisamos garantir que o último nó aponte para o primeiro nó e, aqui a diferença nesta lista, que o primeiro nó aponte para o último, criando um ciclo que existirá nos dois sentidos.

Na construção do primeiro nó, podemos utilizar o valor `None` para os atributos `prox` e `ant`, visto que ainda não temos nenhum elemento na lista.

`No(None, valor, None)`

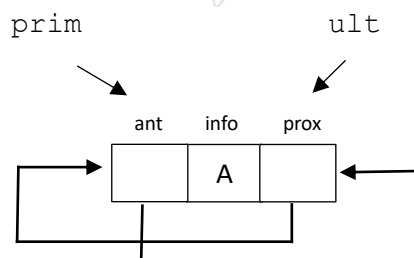
Seguimos fazendo `prim` e `ult` apontar para este nó recém-criado.



Assim, criamos o nó com `No(valor, None, None)` e atribuímos tanto para `self.prim` quanto para `self.ult`, para garantir que eles apontem para o mesmo nó.

`self.prim = self.ult = No(None, valor, None)`

Então, precisamos atualizar os atributos `prox` e `ant` deste nó para apontar si mesmo, o que é feito com a instrução `self.prim.ant = self.ult.prox = self.prim`.



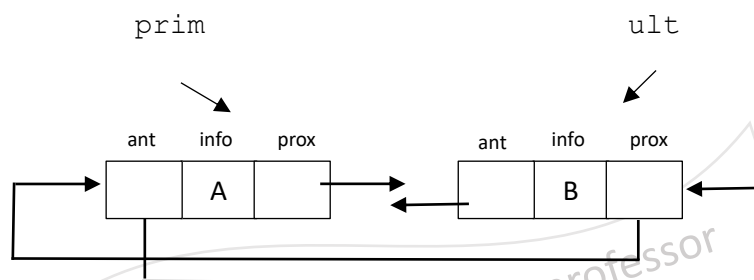
É importante ressaltar que `prim` e `ult` fazem referência ao mesmo nó, o que nos dá a possibilidade de utilizar, na atribuição da criação do nó, apenas um dos atributos. No entanto, utilizamos ambos os atributos para facilitar a visualização da circularidade da lista, visto que o

atributo `ant` do primeiro elemento deve apontar para o último, e o atributo `prox` do último elemento deve apontar para o primeiro elemento. Assim, ao utilizar os dois atributos na atribuição, já deixamos claro a circularidade da lista e evitamos confusões ou erros em futuras modificações no código.

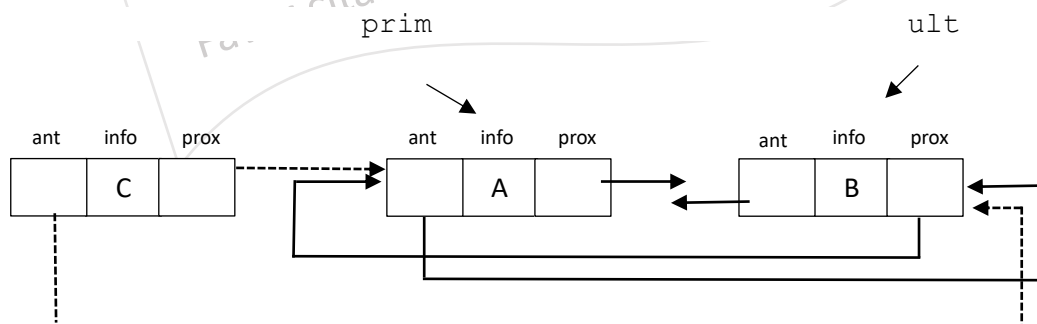
Assim, temos o código para inserir o primeiro elemento em nossa lista quando ela estiver vazia:

```
if self.quant == 0:
    self.prim = self.ult = No(valor, None)
    self.ult.prox = self.prim
```

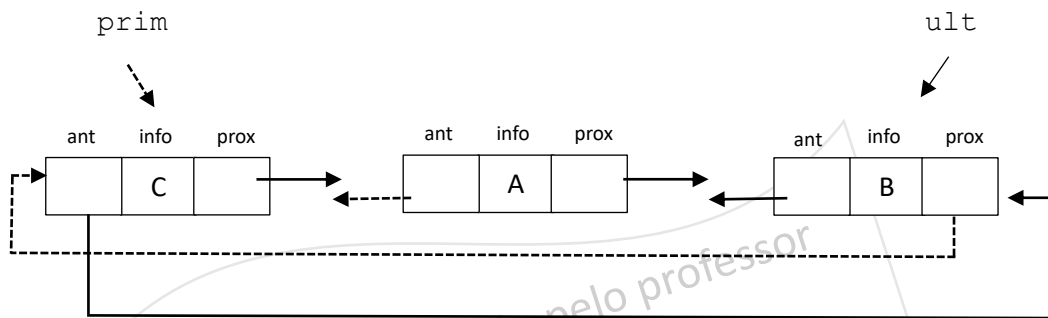
Caso a lista já tenha elementos, a inserção no início é feita, da forma tradicional, criando-se um novo nó com o valor desejado e fazendo-o apontar para o nó que atualmente é o primeiro da lista e, em seguida, atualizando o atributo `self.prim` para apontar para o novo nó.



Se a lista não está vazia, um novo nó é criado com valor recebido, e seus atributos são atualizados para apontar para o último nó (`self.ult`) como seu antecessor e para o primeiro nó atual como seu sucessor (`self.prim`).



Em seguida, o atributo `ant` do primeiro nó (`self.prim.ant`) é atualizado para referenciar o novo nó criado, e `self.prim` é atualizado para apontar para o novo nó. Por fim, o atributo `prox` do último nó (`self.ult.prox`) é atualizado para referenciar o primeiro nó (`self.prim`).



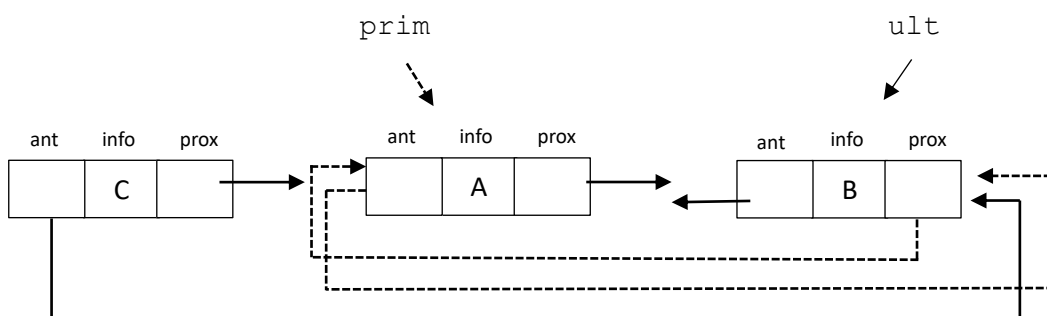
Assim, teremos o código da função `inserir_inicio` da seguinte forma:

```
def inserir_inicio(self, valor):
    if self.quant == 0:
        self.prim = self.ult = No(None, valor, None)
        self.prim.ant = self.ult.prox = self.prim
    else:
        self.prim.ant = self.prim = No(self.ult, valor, self.prim)
        self.ult.prox = self.prim
    self.quant += 1
```

Na implementação da função `remover_inicio` em uma lista dinâmica duplamente encadeada circular, assim como na função `inserir_inicio`, é importante garantir que a circularidade da lista seja mantida.

Se a lista tiver apenas um elemento, devemos setar `self.prim` e `self.ult` como `None`, indicando que a lista está vazia.

No caso de a lista ter mais de um elemento, precisamos atualizar `self.prim` para o segundo nó da lista, ou seja, `self.prim = self.prim.prox`. Além disso, é preciso atualizar o atributo `ant` do novo primeiro nó para que ele aponte para o último nó da lista, com o código `self.prim.ant = self.ult`, e o último nó para apontar para o novo primeiro nó da lista, o que pode ser feito com `self.ult.prox = self.prim`, garantindo a circularidade.

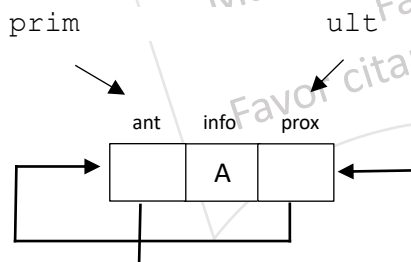


Segue o código da função `remover_inicio`:

```
def remover_inicio(self):  
    if self.quant == 1:  
        self.prim = self.ult = None  
    else:  
        self.prim = self.prim.prox  
        self.prim.ant = self.ult  
        self.ult.prox = self.prim  
    self.quant -= 1
```

Resumindo, se a lista tiver apenas um elemento, setamos `self.prim` e `self.ult` como `None`. Se tiver mais de um elemento, atualizamos `self.prim` para o segundo nó da lista e atualizamos o atributo `ant` do novo primeiro nó para apontar para o último nó da lista. Também atualizamos o atributo `prox` do último nó para apontar para o novo primeiro nó, garantindo a circularidade. Por fim, decrementamos a quantidade de elementos e lembramos da função do *garbage collector* que será o responsável por devolver a área ocupada pelo nó que contém o valor 'C' para a memória.

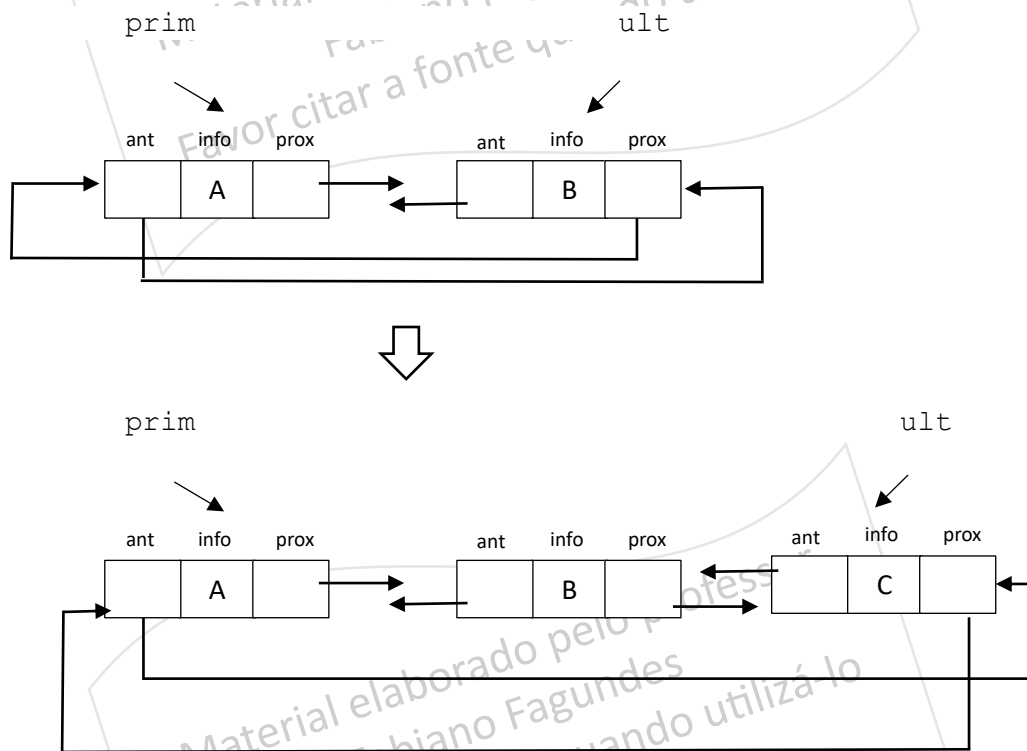
A implementação da função `inserir_fim` inicia da mesma forma que a função `inserir_inicio`. Consideremos a figura a seguir:



Como a figura acima apresenta, no caso de lista vazia o novo elemento será tanto o primeiro quanto o último, e seus atributos `ant` e `prox` apontarão para ele mesmo, criando uma estrutura circular.

```
if self.quant == 0:  
    self.prim = self.ult = No(valor, None)  
    self.ult.prox = self.prim
```

Caso a lista já tenha elementos, é necessário criar um novo nó com o valor desejado e atualizar os ponteiros dos nós para manter a circularidade. Para isso, o novo elemento deve ser criado com seu atributo `ant` apontando para o atual último elemento e seu atributo `prox` apontando para o primeiro elemento da lista. Por fim, ele é inserido na última posição da lista, ou seja, o atributo `prox` do antigo último elemento aponta para o novo elemento e é necessário atualizar tanto o ponteiro do último elemento quando o atributo `ant` do primeiro elemento para apontar para este novo elemento inserido.



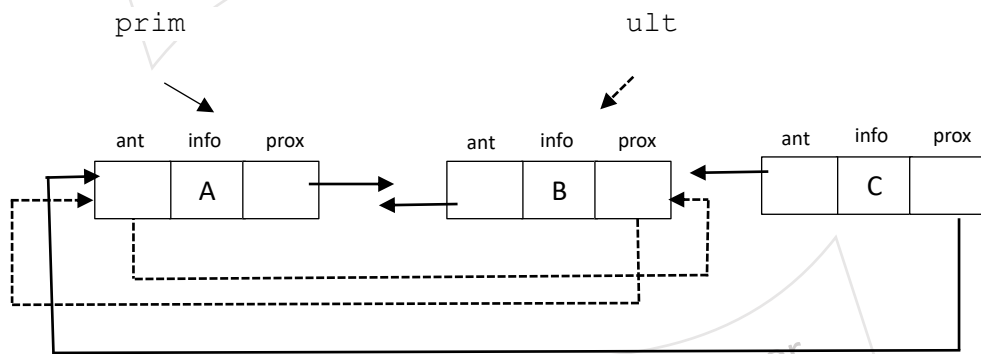
Com isso, temos o código da função `inserir_fim` da seguinte forma:

```
def inserir_fim(self, valor):
    if self.quant == 0:
        self.prim = self.ult = No(valor)
        self.prim.ant = self.ult.prox = self.prim
    else:
        self.ult.prox = self.ult = No(self.ult, valor, self.prim)
        self.prim.ant = self.ult
    self.quant += 1
```

Aproveitando o entendimento sobre o que acontece o fim da lista, vamos agora implementar a função `remover_fim`.

Da mesma forma que a função `remover_fim` da lista dinâmica simplesmente encadeada, primeiro verificamos se só há um elemento. Neste caso, devemos setar `self.prim` e `self.ult` como `None`, indicando que a lista estará vazia.

Caso contrário, atualizamos o atributo `ult` para o penúltimo nó da lista, que pode ser acessado diretamente pelo seu atributo `ant`. Em seguida, atualizamos o atributo `prox` do novo último nó para apontar para o primeiro nó, garantindo a circularidade. Por fim, atualizamos o atributo `ant` do primeiro nó para apontar para o novo último nó. A quantidade de elementos é decrementada em 1.



Assim, temos o código da função `remover_fim` a seguir:

```
def remover_fim(self):
    if self.quant == 1:
        self.prim = self.ult = None
    else:
        self.ult = self.ult.ant
        self.ult.prox = self.prim
        self.prim.ant = self.ult
    self.quant -= 1
```

Primeiro, a função verifica se a lista possui apenas um elemento, ou seja, se o valor da variável `quant` é igual a 1. Se for esse o caso, então a lista fica vazia, e a função simplesmente atribui `None` aos atributos `prim` e `ult`.

Caso contrário, a função atualiza o valor do atributo `ult` para que aponte para o penúltimo elemento e o atributo `prox` do novo último elemento para apontar para o primeiro elemento

da lista. Por fim, ela faz o atributo `ant` do primeiro elemento para apontar para o novo último elemento e decrementa a variável `quant` para refletir a remoção realizada.

Para visualizar o conteúdo da lista precisamos da função `show`. De novo, por ser circular, não podemos usar o critério `None` para indicar o final da lista. Além disso, não podemos usar a condição `aux.prox != self.prim` como critério de parada, pois isso pode levar a um problema de perda do primeiro elemento se ele for o único da lista. Assim, vamos utilizar a quantidade de elementos da lista, armazenada no atributo `quant`, para percorrê-la. O código para percorrer e mostrar a lista seria o seguinte, que em nada difere do código da função `show` da lista dinâmica simplesmente encadeada circular:

```
def show(self):
    aux = self.prim
    for i in range(self.quant):
        print(aux.info, end=' ')
        aux = aux.prox
    print('\n')
```

Agora, por ser duplamente encadeada, podemos implementar a função `show_inverso`, que mostrará os elementos da lista do último para o primeiro elemento, de traz para a frente. E, como já vimos anteriormente, ela é similar à função `show`, sendo implementada de forma simétrica, começando agora inicializando a variável auxiliar com o ponteiro do último elemento e atualizando-a a cada iteração com o valor do ponteiro `ant` de cada nó. Isso pode ser visto no código abaixo:

```
def show_inverso(self):
    aux = self.ult
    for i in range(self.quant):
        print(aux.info, end=' ')
        aux = aux.ant
    print('\n')
```

Por fim, temos as classes da `Lddecirc` e suas funções básicas implementadas a seguir:

```
class No:
    def __init__(self, valor, proximo, anterior):
        self.info = valor
        self.prox = proximo
        self.ant = anterior

class Lddecirc:
    def __init__(self):
        self.prim = self.ult = None
        self.quant = 0

    def inserir_inicio(self, valor):
        if self.quant == 0:
            self.prim = self.ult = No(None, valor, None)
            self.prim.ant = self.ult.prox = self.prim
        else:
            self.prim.ant = self.prim = No(self.ult, valor, self.prim)
            self.ult.prox = self.prim
        self.quant += 1

    def remover_inicio(self):
        if self.quant == 1:
            self.prim = self.ult = None
        else:
            self.prim = self.prim.prox
            self.prim.ant = self.ult
            self.ult.prox = self.prim
        self.quant -= 1

    def inserir_fim(self, valor):
        if self.quant == 0:
            self.prim = self.ult = No(valor)
            self.prim.ant = self.ult.prox = self.prim
        else:
            self.ult.prox = self.ult = No(self.ult, valor, self.prim)
            self.prim.ant = self.ult
        self.quant += 1

    def remover_fim(self):
```

```

    if self.quant == 1:
        self.prim = self.ult = None
    else:
        self.ult = self.ult.ant
        self.ult.prox = self.prim
        self.prim.ant = self.ult
    self.quant -= 1

def show(self):
    aux = self.prim
    for i in range(self.quant):
        print(aux.info, end=" ")
        aux = aux.prox
    print("\n")

def show_inverso(self):
    aux = self.ult
    for i in range(self.quant):
        print(aux.info, end=' ')
        aux = aux.ant
    print('\n')

def estaVazia(self):
    return self.quant == 0

def tamanho_atual(self):
    return self.quant

def ver_primeiro(self):
    return self.prim.info

def ver_ultimo(self):
    return self.ult.info

```

As listas circulares são muito úteis em diversas aplicações, como por exemplo em algoritmos de ordenação, onde a circularidade da lista permite que as operações sejam mais eficientes, ou em programas de reprodução de áudio ou vídeo, onde é necessário percorrer uma lista de arquivos de forma cíclica. Além disso, as listas circulares podem ser usadas em estruturas de dados como

filas de impressão ou em simulações de processos industriais, onde as entidades são processadas em uma sequência circular de operações.

No entanto, é importante lembrar que a implementação de listas circulares requer um pouco mais de atenção e cuidado, especialmente em relação à circularidade, para evitar problemas como *loops* infinitos.

Material elaborado pelo professor
Fabiano Fagundes
Favor citar a fonte quando utilizá-lo

Material elaborado pelo professor
Fabiano Fagundes
Favor citar a fonte quando utilizá-lo