Lista Estática Encadeada

Listas Estáticas Encadeadas (Lee) são uma implementação de Lista Estática que usa uma estrutura de nós encadeados para armazenar os elementos. Em uma Lista Estática Encadeada, cada elemento é armazenado em um nó separado, que contém o elemento e um ponteiro para o próximo nó na lista. O acesso aos elementos na lista é feito seguindo os ponteiros de um nó ao próximo.

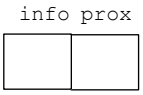
Uma vantagem da Lista Estática Encadeada sobre a Lista Estática Sequencial é que ela pode ser facilmente modificada sem a necessidade de realocar todos os elementos no vetor. Isso é possível porque cada nó contém um índice que funciona como um apontador, que vamos chamar de ponteiro, para o próximo nó, permitindo que os nós sejam rearranjados facilmente, sem a necessidade de copiar todos os elementos para uma nova lista.

Embora a Lista Estática Encadeada tenha algumas vantagens em relação à Lista Estática Sequencial, como a flexibilidade em relação ao tamanho e a possibilidade de inserção e remoção de elementos em qualquer posição da lista, ela também apresenta algumas desvantagens. Uma delas é que o acesso a um elemento em uma posição específica pode ser mais lento, uma vez que é necessário percorrer todos os nós anteriores até chegar ao nó desejado. Além disso, a alocação de memória para cada nó pode gerar um *overhead* adicional em termos de uso de memória. No entanto, esse *overhead* é geralmente muito pequeno em relação ao volume total de dados que a lista pode armazenar, e é compensado pela flexibilidade e facilidade de manipulação que a estrutura proporciona. Em geral, a escolha entre uma Lista Estática Sequencial e uma Lista Estática Encadeada depende das necessidades específicas de cada aplicação

Vamos agora ver como se dá a construção desta tal estrutura de nós. Primeiro criaremos uma classe simples, que chamaremos de `No`, que contém dois atributos: `info`, que armazenará o valor, e `prox`, que cuidará do endereço do próximo elemento na lista, conforme mostra o código a seguir.

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

Podemos ver este nó, graficamente, como algo assim:



Nesta representação, temos o nó como sendo este retângulo com duas caixas, uma que representa o atributo `info` e outra para o atributo `prox`. Em breve veremos esta caixa preenchida e tudo ficará mais claro.

Para termos o TAD por completo, precisaremos criar a nova classe para a lista, que chamaremos de `Lee`. Ela vai precisar de um pouquinho a mais de sua atenção.

```
class Lee:

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

        self.prox_pos_vazia = self.inicializa_estrutura()

        self.prim = -1
        self.ult = -1
```

Por enquanto você não viu a classe `No` em ação, mas ela já está aí e logo entenderemos como. Mas vamos antes entender o que temos aí.

Para a classe `Lee` estamos criando os atributos já conhecidos: `tam_maximo`, `vetor` e `quant`. Você verá que há mais dois atributos, muito importantes quando se fala de uma estrutura encadeada: `prim` e `ult` (este último, com perdão do trocadilho, por vezes não é utilizado em algumas implementações. Mais à frente mostraremos situações com e sem ele para vermos a diferença nas implementações das funções que se utilizam dele).

É importante destacar que o construtor da lista faz uma chamada à função `inicializa_estrutura()`, que retorna um valor atribuído a um novo atributo da nossa lista chamado `prox_pos_vazia`. Esse método, juntamente com esse atributo, são responsáveis por construir a estrutura da nossa lista e permitir que ela seja trabalhada de forma encadeada. Vamos agora conhecer melhor como esse método funciona:

```
    def inicializa_estrutura(self):
        for i in range(self.tam_maximo-1):
            self.vetor[i] = No(None,i+1)
        self.vetor[self.tam_maximo-1] = No(None,-1)
        return 0
```

Nesse método, o primeiro *loop* percorre toda a nossa lista física - vale ressaltar que ele percorre até o penúltimo espaço da lista - preenchendo cada índice com a estrutura da classe `No`. A chamada `No(None, i+1)` cria uma pequena estrutura que contém um local para armazenar a informação (que, neste caso, é `None`) e outro para armazenar o endereço do próximo nó, que é definido como `i+1`.

A última posição da lista, que não é alcançada pelo *loop*, é preenchida com um nó contendo `None` para `info` e `−1` para `prox`. Ao final da função, é retornado o endereço da primeira posição deste vetor, que no caso é o valor zero. Esse valor é recebido pelo atributo `prox_pos_vazia()` em nosso construtor (a função `init` implementado anteriormente).

Para ilustrar de forma mais clara como funciona a construção do vetor, vamos mostrar um exemplo gráfico do vetor antes e depois da chamada da função `inicializa_estrutura()`, bem como os dados dos novos atributos e o que eles representam.

Vamos supor que o valor de `tam_maximo` passado como argumento seja 10. Para facilitar o entendimento, vamos desenhar o vetor ao lado dos índices correspondentes. No entanto, vale lembrar que esses índices não fazem parte do TAD e são utilizados apenas para manipular os dados armazenados.

Então, executando o `init()` até antes da chamada da função `inicializa_estrutura()` temos assim nossos atributos:

```
tam_maximo: 10
quant: 0
vetor:
```

0	None
1	None
2	None
3	None
4	None
5	None
6	None
7	None
8	None
9	None

Após a execução da função `inicializa_estrutura()`:

```
tam_maximo: 10
quant: 0
prox_pos_vazia: 0
prim = -1
ult = -1
vetor:
```

	info	prox
0	None	1
1	None	2
2	None	3
3	None	4
4	None	5
5	None	6
6	None	7
7	None	8
8	None	9
9	None	-1

Podemos interpretar os valores do nosso vetor da seguinte forma: ao ser criada, nossa lista possui tamanho máximo igual a 10 e quantidade de elementos igual a zero. Além disso, ela conta com um vetor de posições vazias, em que cada posição guarda o endereço da próxima posição

vazia no atributo `prox` do nó ali criado. A posição 9, por ser a última, indica com o valor -1 que não há mais posições vazias a seguir.

O atributo `prox_pos_vazia`, como o próprio nome sugere, informa que a primeira posição vazia no vetor é a posição 0. Já os atributos `prim` e `ult` indicam, respectivamente, a posição do primeiro e do último elemento da lista no vetor. Como não há nenhum valor armazenado, eles estão com o valor -1.

No contexto da implementação de uma lista encadeada estática, o valor -1 é frequentemente usado para indicar que não há mais elementos na sequência e que, portanto, não há um próximo elemento a ser apontado pelo nó atual. O mesmo vale quando temos prim e ult com este valor, o que indica que não há valores indicados por eles.

Vale ressaltar que não seria necessário inicializar o vetor com valores que seriam posteriormente substituídos pela função `inicializa_estrutura()`. Em vez disso, poderíamos ter criado o vetor como uma lista do Python e, na função "`inicializa_estrutura()`", ir criando uma a uma as novas posições utilizando a função `append()` do Python. No entanto, optei por não fazer isso aqui para não utilizar recursos de lista do Python, como já havia sido mencionado no início de nossos estudos.

Agora vamos implementar um método de inserção para começarmos a visualizar esta lista sendo preenchida. Para inserir um valor no início da lista, temos duas situações a serem tratadas: a lista está vazia ou já possui ao menos um elemento.

No primeiro caso, os atributos "`prim`" e "`ult`" serão atualizados para cuidar desse único valor inserido, já que ele é, ao mesmo tempo, o primeiro e o último elemento da lista. Já no segundo caso, precisaremos atualizar os atributos "`prim`" ou "`ult`" separadamente, dependendo da situação.

É importante ressaltar que deixaremos o tratamento de uma lista cheia para o programador que utilizará a lista, assim como fizemos na implementação da Lista Estática Sequencial. Vale lembrar também que, quando dizemos que "`prim`" e "`ult`" cuidam da informação que está armazenada, queremos dizer que eles armazenam o índice em que está informação ficou guardada no vetor.

```
def inserir_inicio(self,valor):
   if self.quant == 0:
       self.prim = self.ult = self.ocupa_no(valor,-1)
       self.quant += 1
   else:
       self.prim = self.ocupa_no(valor,self.prim)
       self.quant += 1
```

Lembrando que a função `inserir_inicio` serve para inserir um novo elemento no início da lista encadeada, temos que o código começa com uma verificação se a lista está vazia (ou seja, se `self.quant` é igual a zero). Se isso for verdadeiro, significa que estamos inserindo o primeiro elemento na lista e, portanto, o atributo `prim` (que representa o índice da posição do primeiro elemento no vetor) e `ult` (que representa o índice da posição do último elemento no vetor) devem ser atualizados para apontar para a posição em que o novo elemento será inserido.

A função `ocupa_no` é chamada para alocar uma posição vazia no vetor e preenchê-la com o valor passado como argumento (valor) e com o valor -1 para o atributo `prox`, indicando que é o último elemento da lista pois, se a lista está vazia, este elemento, inserido no início, é ao mesmo tempo o primeiro e o último. Assim, o retorno da função `ocupa_no` é armazenado em `self.prim` e `self.ult`, e a quantidade de elementos (`self.quant`) é incrementada em 1.

Se a lista não estiver vazia (ou seja, se `self.quant` for diferente de zero), significa que já temos elementos na lista e, portanto, basta atualizar o atributo `prim` para apontar para a nova posição do novo elemento. A função `ocupa_no` é novamente chamada para alocar uma posição vazia no vetor e preenchê-la com o valor passado como argumento (valor) e com o valor do atributo `prim` atual (representando o próximo elemento na lista). O retorno da função `ocupa_no` é armazenado em `self.prim` e a quantidade de elementos (`self.quant`) é incrementada em 1.

Note que em ambos os casos, a função `ocupa_no` é chamada para alocar uma posição vazia no vetor. Essa função retorna o índice da posição vazia no vetor, preenchendo-a com os valores passados como argumento. Além disso, ela atualiza o atributo `prox_pos_vazia` para apontar para a próxima posição vazia no vetor, permitindo assim que a próxima chamada de `ocupa_no` aloque essa posição.

Então, como visto, a função `ocupa_no()` é responsável por alocar um novo nó na lista encadeada estática.

```
def ocupa_no(self,valor,proximo):
    posicao_livre = self.prox_pos_vazia
    self.prox_pos_vazia = self.vetor[self.prox_pos_vazia].prox
    self.vetor[posicao_livre] = No(valor,proximo)
    return posicao_livre
```

A lógica da função é relativamente simples: primeiro, ele verifica qual é a próxima posição livre no vetor, que é armazenada no atributo `prox_pos_vazia` da instância da classe Lee. Esse atributo indica a posição no vetor que armazena o endereço do próximo nó vazio, ou seja, uma posição que ainda não foi preenchida com um nó.

Em seguida, a função atualiza o valor de `prox_pos_vazia` para apontar para o próximo nó vazio, que será ocupado em uma próxima chamada da função.

Com a posição livre encontrada, a função cria um novo nó, com o valor e endereço do próximo nó fornecidos como parâmetros, e o insere na posição livre no vetor. Finalmente, a função retorna o índice da posição onde o novo nó foi armazenado no vetor.

Agora vamos ver como isso tudo funciona na prática. Primeiro, vamos organizar o código de nosso TAD no nosso arquivo que chamei aqui de Lee.py. Vamos antes acrescentar duas funções que ajudarão a visualizar como tudo está acontecendo internamente em nossa estrutura: o `show()` e, excepcionalmente, o `show_vetor()`. Digo excepcionalmente porque devemos nos lembrar que nosso usuário programador não deverá ter acesso a forma como nosso TAD é implementado. Ela está aqui mais para efeito didático.

Primeiramente, vamos explicar o que cada função faz:

- `show()`: Essa função percorre todos os nós da lista, iniciando pelo nó `prim` e seguindo através do atributo `prox` de cada nó, e imprime o valor armazenado no atributo `info` de cada nó.
- `show_vetor()`: Essa função imprime o estado atual do vetor utilizado para armazenar os nós da lista, bem como os valores dos atributos `prim` e `ult`. É importante lembrar que o vetor é utilizado para armazenar os nós, mas o acesso aos elementos da lista é feito através dos atributos `prim` e `prox` dos nós, e não através dos índices do vetor.

Agora, vamos entender um pouco mais sobre o funcionamento dessas funções.

```
    def show(self):
        aux = self.prim
        while aux != -1:
            print(self.vetor[aux].info)
            aux = self.vetor[aux].prox
```

A função `show()` é relativamente simples. Ela inicia a variável `aux` com o valor do atributo `prim` e, em seguida, percorre todos os nós da lista, imprimindo o valor do atributo `info` de cada nó e atualizando o valor de `aux` para o valor do atributo `prox` de cada nó. Esse processo é repetido até que `aux` tenha o valor -1, indicando que chegamos ao final da lista.

```
  def show_vetor(self):
        print('Prim=',self.prim)
        print('Ult=',self.ult)
        print('Primeira pos vazia=',self.prox_pos_vazia)
        print('Vetor=')
        for i in range(self.tam_maximo):
            print(i, self.vetor[i].info, self.vetor[i].prox)
```

A função `show_vetor()` é um pouco mais complexa, pois ela imprime informações sobre o vetor utilizado para armazenar os nós da lista. Primeiro, ela imprime os valores dos atributos `prim` e `ult`, que indicam, respectivamente, o índice do primeiro nó da lista e o índice do último nó da lista. Também é impresso o valor que informa qual é a primeira posição vazia. Em seguida, ela percorre todo o vetor, imprimindo o índice de cada posição, bem como o valor armazenado nos atributos `info` e `prox` de cada nó.

Por fim, é importante lembrar que a função `show_vetor()` é uma função de *debug* e não deve ser utilizada no código final do nosso TAD, uma vez que ela expõe detalhes de implementação que não são relevantes para o usuário final.

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

class Lee:
  def __init__(self,tamanho):
      self.tam_maximo = tamanho
      self.vetor = [None] * self.tam_maximo
      self.quant = 0
      self.prox_pos_vazia = self.inicializa_estrutura()
      self.prim = -1
      self.ult = -1

  def inicializa_estrutura(self):
      for i in range(self.tam_maximo-1):
          self.vetor[i] = No(None,i+1)
      self.vetor[self.tam_maximo-1] = No(None,-1)
      return 0

  def ocupa_no(self,valor,proximo):
      posicao_livre = self.prox_pos_vazia
      self.prox_pos_vazia = self.vetor[self.prox_pos_vazia].prox
      self.vetor[posicao_livre] = No(valor,proximo)
      return posicao_livre

  def inserir_inicio(self,valor):
      if self.quant == 0:
          self.prim = self.ult = self.ocupa_no(valor,-1)
          self.quant += 1
      else:
          self.prim = self.ocupa_no(valor,self.prim)
          self.quant += 1

    def show(self):
        aux = self.prim
        while aux != -1:
            print(self.vetor[aux].info)
            aux = self.vetor[aux].prox

    def show_vetor(self):
        print('Prim=',self.prim)
        print('Ult=',self.ult)
        print('Primeira pos vazia=',self.prox_pos_vazia)
        print('Vetor=')
        for i in range(self.tam_maximo):
```

```
            print(i, self.vetor[i].info, self.vetor[i].prox)
```

Vamos fazer agora um programa simples, com uma lista de capacidade igual a 5, para testar a função `inserir_inicio()` e, consequentemente, testar também as demais funções explicadas até agora.

```
import Lee

lista = Lee.Lee(5)

print('Lista:')
lista.show()
lista.show_vetor()

lista.inserir_inicio('C')
print('Lista após inserir início C: (esperado - C)')
print('Lista')
lista.show()
lista.show_vetor()

lista.inserir_inicio('D')
print('Lista após inserir início D: (esperado - DC)')
lista.show()
lista.show_vetor()
```


Vamos acompanhar passo a passo a construção da lista com a saída do programa:

Iniciamos criando uma nova instância da classe Lee com tamanho máximo igual a 5:

```
lista = Lee.Lee(5)
```

Neste momento podemos visualizar a estrutura interna da lista da seguinte forma:

```
tam_maximo: 5
quant: 0
prox_pos_vazia: 0
prim = -1
ult = -1
vetor:
```

	info	prox
0	None	1
1	None	2
2	None	3
3	None	4
4	None	-1

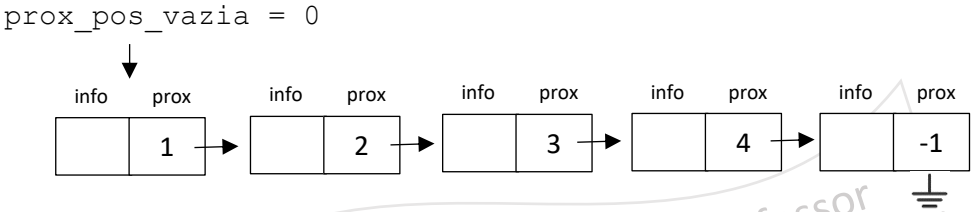
Vamos fazer duas representações gráficas para mostrar como está a lógica deste encadeamento neste momento:

Primeiro, como está a lista propriamente dita:

O símbolo de aterramento indica que os atributos prim e ult estão com valor -1, logo, sem endereço, ou seja, não possuem referência para nenhuma posição no vetor. Isso significa que a lista está vazia, sem nenhum elemento armazenado.

Além disso, temos também uma lista paralela de posições vazias, a qual é controlada pelo atributo `prox_pos_vazia`. Quando um elemento é inserido na lista real, uma posição para ele é buscada nesta lista de posições vazias. Quando um elemento é removido da lista, a posição por ele ocupada anteriormente é adicionada a essa lista, permitindo que ela possa ser reutilizada para inserção de novos elementos. Para facilitar a visualização, deixaremos em branco o quadro da esquerda, referente ao atributo `info` do nó, quando seu conteúdo for `None`, indicando que não há valor válido armazenado naquela posição.

Para facilitar seu entendimento, sugiro que compare cada nós desta lista lógica com os nós na lista física apresentada acima.



Seguindo a execução do programa, agora mostramos a lista vazia:

```
print('Lista')
lista.show()
```

Com a esperada saída, obviamente sem nada impresso pois a lista está vazia:

```
Lista:
```

```
lista.inserir_inicio('C')
```

O que implica em uma série de modificação na nossa estrutura interna da lista, dada a execução da função `inserir_lista()`. No caso, será executado este trecho desta função, visto que a lista está vazia:

```
if self.quant == 0:
   self.prim = self.ult = self.ocupa_no(valor,-1)
   self.quant += 1
```

Será criado um novo nó utilizando a função `ocupa_no()`, que retorna a posição no vetor onde o novo nó foi criado. Em seguida, os atributos `prim` e `ult` são atualizados com a posição do novo nó, pois, como a lista está vazia, ele é o primeiro e único elemento na lista. O atributo `quant` é incrementado em 1 para indicar que mais um elemento foi adicionado.

Voltando à função `ocupa_no()`, o primeiro comando cria uma variável `posicao_livre` e atribui a ela o valor da primeira posição vazia da lista, que é armazenada no atributo `prox_pos_vazia`. O segundo comando atribui a este atributo o valor que está armazenado no atributo `prox` do nó que acabou de ser alocado – que será a nova primeira posição vazia. Em seguida, o novo nó é criado na posição `posicao_livre` do vetor, sendo o valor 'C' atribuído ao atributo `info` do nó e o valor -1 é atribuído ao atributo `prox`, indicando que não há mais elementos na lista após ele. Por fim, o valor desta posição é retornado.

No programa teste fazemos as seguintes chamadas de funções para visualizar como estão nossa lista lógica e nossa lista física:

```
print('Lista após inserir início C: (esperado - C)')
print('Lista')
lista.show()
lista.show_vetor()
```

Com isso, temos como saída:

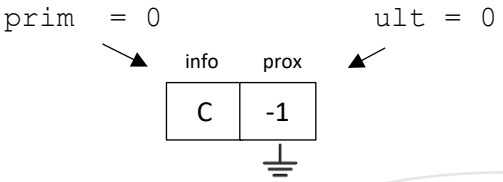
```
Lista após inserir início C: (esperado - C)

Lista:

C

Show vetor:

Prim= 0

Ult= 0

Primeira pos vazia= 1

Vetor=

0 C -1

1 None 2

2 None 3

3 None 4

4 None -1
```

Esta saída reflete a estrutura interna da lista que está desta forma:

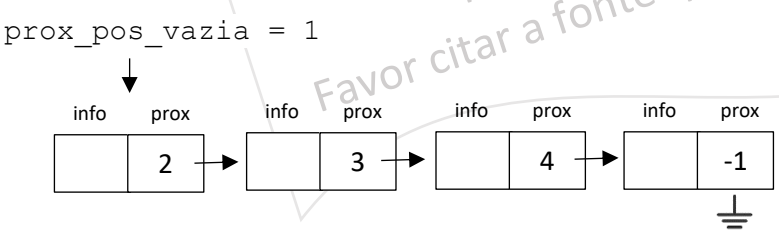
```
tam_maximo: 5
quant: 1
prox_pos_vazia: 1
prim = 0
ult = 0
vetor:
```

	info	prox
0	C	-1
1	None	2
2	None	3
3	None	4
4	None	-1

Numa visualização gráfica, teríamos agora a lista desta forma:



Uma visualização gráfica da nossa "lista paralela" de posições vazias ficaria, agora, desta forma:



Continuando a execução do nosso programa teste, temos agora:

```
lista.inserir_inicio('D')
print('Lista após inserir início D: (esperado - DC)')
lista.show()
lista.show_vetor()
```

Cuja saída é:

```
Lista após inserir início D: (esperado - DC)
Lista:
D
C
Show vetor:
Prim= 1
Ult= 0
Primeira pos vazia= 2
Vetor=
0 C -1
1 D 2
2 None 3
3 None 4
4 None -1
```


Vamos entender o que aconteceu na execução deste trecho:

Agora, na chamada da função `inserir_inicio()`, será executado este trecho, visto que a lista agora já tem ao menos um elemento.

```
self.prim = self.ocupa_no(valor,self.prim)
self.quant += 1
```

Novamente, um novo nó é criado utilizando a função `ocupa_no()`, que retorna a posição no vetor onde o novo nó foi criado. No entanto, observe que o valor de `self.prim` é passado como segundo argumento para o atributo `prox` do novo nó. Isso ocorre porque, ao inserir um elemento no início da lista, o próximo nó será aquele que era, até o momento, o primeiro da lista.

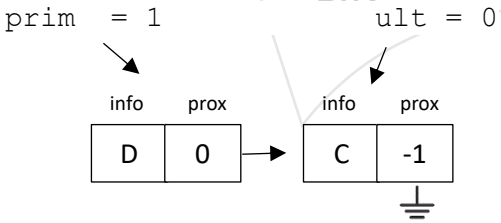
Em seguida, o atributo `prim` é atualizado com a posição deste novo nó, índice este retornado pela função `ocupa_no()`. O atributo `quant` é incrementado em 1 para indicar que mais um elemento foi adicionado à lista.

A saída acima reflete a estrutura interna da lista que, agora, pode ser vista desta forma:

```
tam_maximo: 5
quant: 2
prox_pos_vazia: 2
prim = 1
ult = 0
vetor:
```

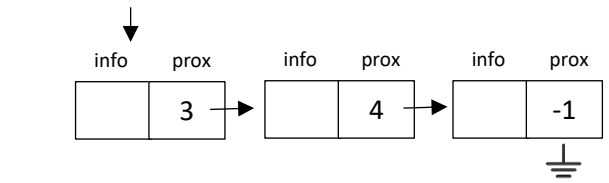
	info	prox
0	C	-1
1	D	0
2	None	3
3	None	4
4	None	-1

Numa visualização gráfica, teríamos agora a lista desta forma:



Este é o momento para observar que a ordem do vetor não está mais relacionada à ordem da lista. No vetor, tem-se o valor 'C' seguido de 'D', enquanto a lista, controlada pelo atributo `prim`, começa com o valor 'D', seguido do valor 'C'.

Agora, uma visualização gráfica da nossa "lista paralela" de posições vazias ficaria, agora, desta forma:



Vamos trabalhar agora a função `inserir_fim()`. Sabemos que temos um atributo, `ult`, que cuida do último elemento da lista. E é para lá que devemos olhar ao fazer a inserção. Se a lista estiver vazia, faremos o mesmo que foi feito com a função `inserir_inicio()`, ou seja, ocupa-se um nó e os dois atributos, `prim` e `ult`, passam a apontar para ele.

```
def inserir_fim(self,valor):
    if self.quant == 0:
        self.prim = self.ult = self.ocupa_no(valor,-1)
        self.quant += 1
    else:
        self.vetor[self.ult].prox = self.ult = self.ocupa_no(valor,-1)
        self.quant += 1
```

Caso a lista não esteja vazia, é criado um novo nó usando a função `ocupa_no()` passando o valor a ser inserido e o valor -1 para o atributo `prox` do nó, pois, sendo o novo último elemento da lista, não haverá nenhum elemento após ele. Em seguida, o atributo `prox` do nó que era o último da lista é atualizado para apontar para o novo nó criado, e o atributo `ult` da lista é atualizado para apontar para o novo nó criado, pois ele é o último elemento da lista agora. Por fim, o atributo `quant` da lista é incrementado em 1 para indicar que mais um elemento foi adicionado.

Vamos acrescentar o trecho abaixo no nosso programa teste para entendemos o funcionamento da função `inserir_fim()`.

```
lista.inserir_fim('B')
print('Lista após inserir fim B: (esperado - DCB)')
print('Show:')
lista.show()
print('Show vetor:')
lista.show_vetor()
```

O código está inserindo o valor 'B' no final da lista, utilizando a função `inserir_fim()`. A função `ocupa_no()` é chamado para alocar um novo nó na lista, com o valor 'B' e o atributo

prox igual a -1, indicando que ele será o último elemento da lista. O atributo prox do nó anterior ao novo nó é atualizado para apontar para a posição onde o novo nó foi alocado. Em seguida, o atributo ult da lista é atualizado para a posição do novo nó.

A saída do código mostra a lista após a inserção do novo elemento, utilizando a função show() da classe Lee, que mostra os valores armazenados na lista.

Também é mostrado o vetor utilizado para implementar a lista, utilizando a função show_vetor() da classe Lee, que mostra os valores armazenados em cada posição do vetor. A saída esperada para a lista após a inserção é 'DCB'.

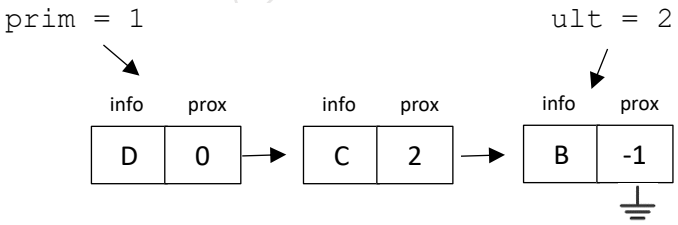
```
Lista após inserir fim B: (esperado - DCB)

Show:

D

C

B

Show vetor:

Prim= 1

Ult= 2

Primeira pos vazia= 3

Vetor=

0 C 2

1 D 0

2 B -1

3 None 4

4 None -1
```


A estrutura interna da lista pode ser vista desta forma:

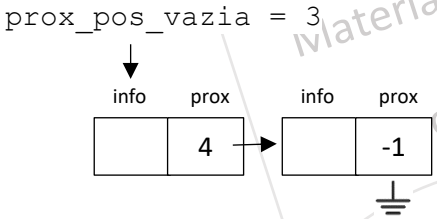
```
tam_maximo: 5
quant: 3
prox_pos_vazia: 3
prim = 1
ult = 2
vetor:
```

	info	prox
0	C	2
1	D	0
2	B	-1
3	None	4
4	None	-1

Acompanhando a visualização lógica, teríamos agora a lista desta forma:



Agora, uma visualização gráfica da nossa "lista paralela" de posições vazias ficaria desta forma, visto que utilizamos mais uma posição vazia para inserir o valor 'B':



Vamos conhecer outra função que se fará necessária no nosso TAD para apoiar as tarefas de remoção de elementos: a `devolve_no()`, cujo objetivo é informar que aquele espaço de memória anteriormente ocupado no vetor volta a estar liberado para uso.

```
    def devolve_no(self, no):
        self.vetor[no].prox = self.prox_pos_vazia
        self.prox_pos_vazia = no
```

Para liberar a posição do nó no vetor para que possa ser utilizada novamente futuramente, essa função recebe como argumento o índice do nó que se deseja devolver para a lista de posições vazias. Em seguida, o atributo `prox` do nó é atualizado para apontar para a primeira posição vazia da lista (que é armazenada no atributo `prox_pos_vazia`). E, por fim, o atributo `prox_pos_vazia` é atualizado para armazenar o índice do nó que acabou de ser devolvido.

Implementaremos agora a função `remove_inicio()` que, como o próprio nome diz, remove o elemento da primeira posição da lista. Ela começa verificando se a lista possui apenas um elemento pois, nesse caso, a função precisa simplesmente devolver o nó para a lista de posições vazias utilizando a função `devolve_no()`, que recebe como parâmetro o endereço do nó a ser devolvido, e atribui -1 aos atributos `prim` e `ult` da lista, indicando que a lista está vazia.

```
    def remover_inicio(self):
        if self.quant == 1:
            self.devolve_no(self.prim)
            self.prim = self.ult = -1
        else:
            novo_prim = self.vetor[self.prim].prox
            self.devolve_no(self.prim)
            self.prim = novo_prim
        self.quant-=1
```

No caso da lista possuir mais de um elemento, a função atribui à variável `novo_prim` a posição do segundo elemento da lista, pois este passará a ser o novo primeiro elemento. Em seguida, a função devolve a posição ocupada primeiro elemento para a lista de posições vazias e atualiza o valor do atributo `prim` para o endereço do novo primeiro elemento – este que está armazenado na variável `novo_prim`. Por fim, em ambas as situações, o atributo `quant` é decrementado em 1 para indicar que a lista possui um elemento a menos.

Para testar e compreender melhor estas funções, vamos acrescentar ao nosso programa de teste o trecho a seguir:

```
lista.remover_inicio()
print('Lista após remover início: (esperado - CB)')
print('Show:')
lista.show()
print('Show vetor:')
lista.show_vetor()
```

Esse trecho de código chama a função `remover_inicio()` para remover o primeiro elemento da lista representada pelo objeto lista. Em seguida, ele exibe a lista e o vetor atualizados após a remoção do elemento.

Dentro da função `remover_inicio()`, primeiro verifica-se se a lista tem apenas um elemento. Se sim, o nó correspondente é devolvido à lista de posições vazias através da função `devolve_no()`, e os atributos `prim` e `ult` da lista são atualizados para -1, indicando que a lista está vazia. Se a lista tem mais de um elemento, o próximo nó após o primeiro é definido como o novo primeiro nó, o antigo primeiro nó é devolvido à lista de posições vazias através da função `devolve_no()`, e o atributo `prim` da lista é atualizado para a posição do novo primeiro nó.

Dentro da função `devolve_no()`, o nó que está sendo devolvido é inserido na lista de posições vazias, tornando-se a primeira posição vazia disponível, através do atributo `prox` do nó.

Por fim, os métodos `show()` e `show_vetor()` são chamados para exibir a lista e o vetor atualizados após a remoção do elemento, como visto a seguir:

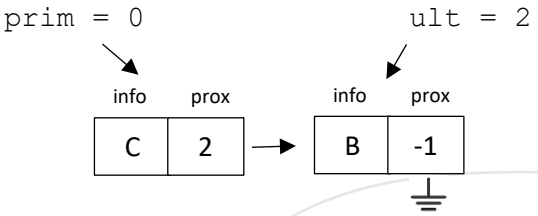
```
Lista após remover início: (esperado - CB)
Show:
C
B
Show vetor:
Prim= 0
Ult= 2
Primeira pos vazia= 1
Vetor=
0 C 2
1 D 3
2 B -1
3 None 4
4 None -1
```

Assim, temos a estrutura interna da lista desta forma:

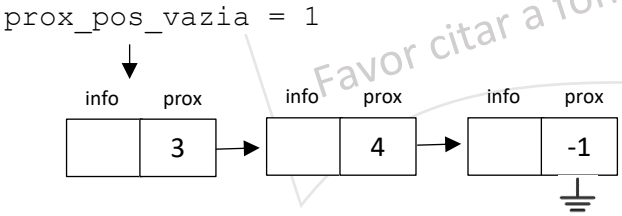
```
tam_maximo: 5
quant: 2
prox_pos_vazia: 1
prim = 0
ult = 2
vetor:
```

	info	prox
0	C	2
1	D	3
2	B	-1
3	None	4
4	None	-1

Acompanhando a visualização lógica, teríamos agora a lista desta forma:



A visualização da nossa "lista paralela" de posições vazias ficaria desta forma agora que devolvemos a posição ocupada pelo valor 'D', que era o primeiro elemento da lista.



Observe que no vetor ainda aparece o valor 'D', porém ele não faz mais parte da lista, pois seu endereço, o índice 1, não é guardado pelo atributo `prox` de nenhum nó que está ligado à lista. Na verdade, o índice 1 é o valor da `prox_pos_vazia`, logo, quando houver outra inserção, será justamente esta posição que será ocupada.

A função `remover_fim()` será interessante pois ela chama a atenção para uma característica da Lista Estática Encadeada que é justamente o encadeamento que impossibilita o acesso indexado a um elemento. Para remover o último elemento da lista, é necessário encontrar o nó

que está imediatamente antes dele, pois este será o novo último elemento e deverá ser cuidado pelo atributo `ult`. Porém, não há como, conhecendo o último elemento, saber quem é o anterior a eles. Isso pode ser feito percorrendo a lista a partir do primeiro elemento, enquanto guarda o endereço do nó anterior. Aí sim, ao chegar no último elemento, basta atualizar o atributo `prox` do nó anterior para -1 e devolver o nó removido à lista de posições vazias.

Por fim, o atributo `ult` da lista deve ser atualizado para apontar para o novo último elemento da lista. É importante lembrar que, se a lista possuir apenas um elemento, a remoção do último elemento implica na remoção do único elemento da lista, ou seja, os atributos `prim` e `ult` deverão receber -1.

Sua implementação, então, fica desta forma:

```
def remover_fim(self):
    if self.quant == 1:
        self.devolve_no(self.prim)
        self.prim = self.ult = -1
    else:
        aux = self.prim
        while self.vetor[aux].prox != self.ult:
            aux = self.vetor[aux].prox
        self.devolve_no(self.ult)
        self.vetor[aux].prox = -1
        self.ult = aux
    self.quant-=1
```

Inicialmente, é feita uma verificação para saber se a lista contém apenas um elemento. Se for o caso, este único elemento é devolvido usando a função `devolve_no()` e os atributos `prim` e `ult` são resetados para -1, indicando que a lista está vazia.

Se a lista contém mais de um elemento, então um *loop* é iniciado para percorrer todos os nós até encontrar o penúltimo elemento, ou seja, o nó que aponta para o último elemento. Isso é feito verificando o atributo `prox` de cada nó. Quando este atributo possui o mesmo valor que `ult` guarda, é porque ele é o penúltimo elemento.

Assim, quando o penúltimo nó é encontrado, o último elemento é devolvido usando a função `devolve_no()` e o atributo `prox` do penúltimo nó é atualizado para -1, indicando que agora

este nó é o último elemento da lista. O atributo `ult` é atualizado para a posição deste penúltimo nó.

Por fim, em ambas as situações o atributo `quant` é decrementado em 1 para indicar que um elemento foi removido da lista.

Para testar este código e explicá-lo, precisamos primeiro inserir mais alguns valores na lista para mostrar melhor a complexidade de sua execução. Então, acrescentaremos o seguinte trecho ao nosso programa teste, para inserir dois novos valores e, na sequência, remover o último elemento da lista:

```
lista.inserir_fim('F')
print('Lista após inserir fim F: (esperado - CBF)')
print('Show:')
lista.show()
print('Show vetor:')
lista.show_vetor()

lista.inserir_inicio('E')
print('Lista após inserir início E: (esperado - ECBF)')
print('Show:')
lista.show()
print('Show vetor:')
lista.show_vetor()

lista.remover_fim()
print('Lista após remover fim: (esperado - ECB)')
print('Show:')
lista.show()
print('Show vetor:')
lista.show_vetor()
```

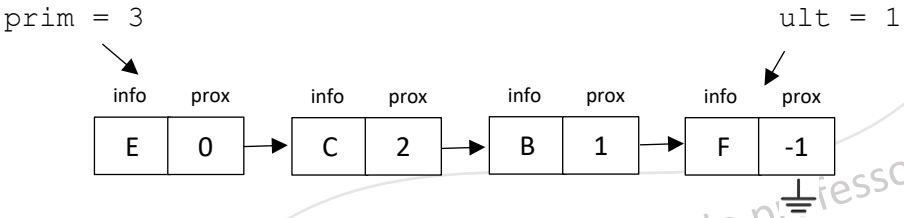

Vamos acompanhar o estado da lista após as duas inserções, em suas várias formas de visualização:

Assim, temos a estrutura interna da lista desta forma:

```
tam_maximo: 5
quant: 4
prox_pos_vazia: 4
prim = 3
ult = 1
vetor:
```

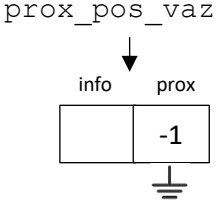
	info	prox
0	C	2
1	F	-1
2	B	1
3	E	0
4	None	-1

Acompanhando a visualização lógica, teríamos agora a lista desta forma:



Aqui dá para observar bem a diferença da sequência existente no vetor para a sequência da lista lógica: o primeiro elemento da nossa lista, que contém o valor 'E', está armazenado na última posição ocupada do vetor.

A visualização da nossa "lista paralela" de posições vazias está agora desta forma.



Focaremos agora na execução da remoção, cujo trecho do resultado é apresentado a seguir:

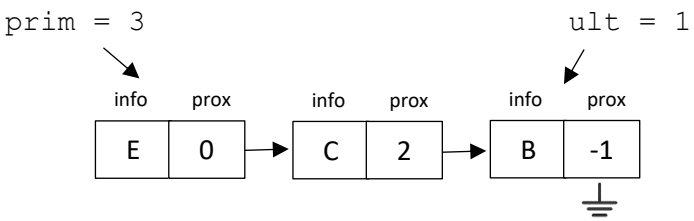
```
Lista após remover fim: (esperado - ECB)

Show:

E

C

B

Show vetor:

Prim= 3

Ult= 2

Primeira pos vazia= 1

Vetor=

0 C 2

1 F 4

2 B -1

3 E 0

4 None -1
```


Então, neste momento, temos as seguintes configurações para nossa lista:

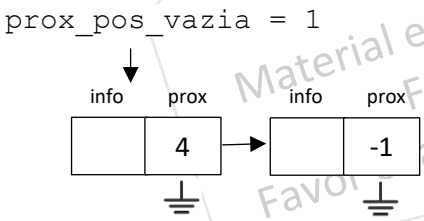
```
tam_maximo: 5
quant: 3
prox_pos_vazia: 1
prim = 3
ult = 2
vetor:
```

	info	prox
0	C	2
1	F	4
2	B	-1
3	E	0
4	None	-1

Acompanhando a visualização lógica, teríamos agora a lista desta forma:



A "lista paralela" de posições vazias está agora desta forma.



Atendendo a pedidos, deixarei aqui implementações alternativas das funções remover_fim() e show() utilizando for no lugar do while.

```
    def show(self):
        aux = self.prim
        for i in range(self.quant):
            print(self.vetor[aux].info)
            aux = self.vetor[aux].prox

    def remover_fim(self):
        if self.quant == 1:
            self.devolve_no(self.prim)
            self.prim = self.ult = -1
        else:
            aux = self.prim
            for i in range(self.quant-2):
                aux = self.vetor[aux].prox
            self.devolve_no(self.ult)
            self.vetor[aux].prox = -1
            self.ult = aux
        self.quant-=1
```

É um bom exercício implementar estas funções e analisar sua execução por meio de um teste de mesa comparando com o resultado da execução pela máquina.

Vamos ver agora outras quatro funções simples que retornam informações sobre a Lista Estática Encadeada.

```
    def tamanho_atual(self):
        return self.quant

    def capacidade(self):
        return self.tam_maximo

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

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


Estas funções não diferem da implementação realizada para a Lista Estática Sequencial pois realizam operações simples com os atributos da classe e não precisam de uma explicação muito detalhada.

A função `tamanho_atual()` retorna o número de elementos atualmente armazenados na lista, que está armazenado no atributo quant.

A função `capacidade()` retorna o tamanho máximo da lista, que está armazenado no atributo `tam_maximo`.

A função `esta_vazia()` verifica se a lista está vazia, ou seja, se `quant` é igual a zero, enquanto a função `esta_cheia()` verifica se a lista está cheia, ou seja, se `quant` é igual a `tam_maximo`.

As funções `ver_primeiro()` e `ver_ultimo()` utilizam a lógica de percorrer a lista encadeada para retornar o primeiro e o último elemento, respectivamente. A função `ver_primeiro()` retorna o valor armazenado no atributo info do nó apontado pelo atributo `prim`, que é o primeiro elemento da lista encadeada. Já a função `ver_ultimo()` vale-se da existência do atributo `ult`, que indica qual é a posição do último elemento da lista, retornando o valor armazenado no atributo `info` deste nó.

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

def ver_ultimo(self):
    return self.vetor[self.ult].info
```

Um bom teste de aprendizado da implementação deste TAD está na função remover_irmãos(self,valor) que remove os elementos imediatamente anterior e posterior a 'valor' na lista. Consideremos que não há valores repetidos na lista.

De início precisamos considerar as várias situações especiais ao implementar esta função:

1. A lista está vazia: se a lista estiver vazia, a função não deve realizar nenhuma operação, pois não há elementos para serem removidos.
2. A lista possui somente um elemento: se a lista possui apenas um elemento, a função não deve realizar nenhuma operação, pois não existem irmãos para serem removidos.
3. O valor em questão é o primeiro elemento da lista: se o valor em questão é o primeiro elemento da lista, não há irmão anterior a ser removido, portanto a função deve remover somente o irmão posterior. É importante ressaltar que nessa situação não existe a possibilidade de não haver irmão posterior, já que a situação em que não há irmão anterior e posterior foi tratada na situação 2.
4. O valor em questão é o último elemento da lista: se o valor em questão é o último elemento da lista, não há irmão posterior a ser removido, portanto a função deve remover somente o irmão anterior. De maneira semelhante à situação anterior, não há a possibilidade de não existir o irmão anterior, já que essa situação foi tratada na situação 2.
5. O valor em análise é o segundo elemento da lista: nessa situação, ao remover o irmão anterior (que até então era o primeiro elemento da lista), o elemento em questão se torna o novo primeiro elemento da lista.
6. O valor em análise é o penúltimo elemento da lista: nessa situação, ao remover o irmão posterior (que até então era o último elemento da lista), o elemento em questão se torna o novo último elemento da lista.

Considerando estes casos, temos a implementação da função remover_irmãos a seguir.

```
    def remover_irmaos(self,valor):
        if self.quant !=1 and self.quant !=0:
            anterior_do_anterior = -1
            anterior = -1
```

```
        atual = self.prim
        while self.vetor[atual].info != valor and atual != -1:
            # andando pela lista até achar valor
            # ou chegar ao fim da lista
            anterior_do_anterior = anterior
            anterior = atual
            atual = self.vetor[atual].prox
        if self.vetor[atual].info == valor:
        # ou seja, achou valor na lista e ele está no índice atual
            if self.vetor[self.prim].prox == atual:
            # age se o irmão anterior for o primeiro da lista
                self.remover_inicio()
            else:
                if atual != self.prim:
                # caso o irmão anterior não seja o primeiro da lista
                    self.vetor[anterior_do_anterior].prox = atual
                    self.devolve_no(anterior)
                    self.quant -= 1
            if self.vetor[atual].prox == self.ult:
            # age se o irmão posterior for o próximo da lista
                self.remover_fim()
            else:
                if atual != self.ult:
                # caso o irmão posterior não seja o primeiro da lista
                    proximo = self.vetor[atual].prox
                    self.vetor[atual].prox = self.vetor[proximo].prox
                    self.devolve_no(proximo)
                    self.quant -= 1
```

O código possui os seguintes passos:

A função começa com uma verificação para garantir que a lista não esteja vazia ou tenha apenas um elemento. Em seguida, a função percorre a lista até encontrar o valor cujos irmãos devem ser removidos ou até chegar ao final da lista.

Se o valor é encontrado na lista, então a função verifica se o irmão anterior é o primeiro elemento da lista. Se for, a função chama a função `remover_inicio()` para removê-lo.

Se o irmão anterior não é o primeiro elemento da lista, então a função atualiza os ponteiros para "pular" o elemento anterior, ou seja, removendo-o. Isso é feito ajustando o ponteiro do elemento anterior ao anterior (variável `anterior_do_anterior`) para apontar para o elemento atual.

Em seguida, a função verifica se o irmão posterior é o último elemento da lista. Se for, a função chama a função `remover_fim()` para removê-lo.

Se o irmão posterior não for o último elemento da lista, a função atualiza os ponteiros para ignorá-lo, ou seja, "pulando" ele. Isso é feito ajustando o ponteiro do elemento atual para o elemento seguinte ao elemento posterior, com isso removendo o elemento posterior.

Por fim, a função atualiza a contagem de elementos na lista (`quant`).

Agora, retorno aqui as sugestões de funções que indiquei lá na Lista Estática Sequencial para você implementar e testar seu aprendizado até o momento:

- `remover_elemento(self, valor)`: remove o elemento com valor especificado da lista, caso ele exista. Caso o elemento não esteja na lista, a função não faz nenhuma ação.
- `buscar(self, valor)`: busca o valor especificado na lista e retorna a posição do elemento na lista (posição 0, 1...). Se o valor não estiver na lista, a função retorna `None`.
- `inserir_apos(self, valor1, valor2)`: insere o `valor1` após o `valor2` na lista, caso o `valor2` exista. Considera-se que não há valores repetidos na lista. Se o `valor2` não estiver presente na lista, a função não faz nenhuma ação.
- `inserir_antes(self, valor1, valor2)`: insere o `valor1` antes do `valor2` na lista, caso o `valor2` exista. Considera-se que não há valores repetidos na lista. Se o `valor2` não estiver presente na lista, a função não faz nenhuma ação.

Ambas as implementações de Lista Estática – Sequencial ou Encadeada - têm suas próprias vantagens e desvantagens, e a escolha da implementação adequada depende das necessidades específicas da aplicação. Por exemplo, as listas estáticas sequenciais são geralmente mais rápidas para acessar elementos aleatoriamente, enquanto as listas estáticas encadeadas são mais flexíveis quanto à inserção e remoção de elementos.

Entretanto, há uma desvantagem inerente às duas listas, que decorre do fato delas serem estáticas: a impossibilidade de crescimento da lista em tempo de execução. Será sobre isso que tralharemos no próximo capítulo.