Listas

Uma Lista é um tipo abstrato de dados (TAD) que representa uma coleção de elementos, onde cada elemento tem uma posição única e conhecida. As listas permitem armazenar, acessar, inserir e remover elementos.

As operações comuns em uma lista incluem:

- Inserir um elemento em uma determinada posição
- Remover um elemento em uma determinada posição
- Acessar um elemento em uma determinada posição
- Verificar o tamanho da lista
- Verificar se a lista está vazia

As listas podem ser implementadas de diferentes maneiras, como *arrays* ou listas encadeadas, cada uma com suas próprias vantagens e desvantagens. A escolha da implementação depende das necessidades específicas da aplicação. Além disso, as listas podem ser ordenadas ou não ordenadas, dependendo das necessidades da aplicação.

Lista Estática

Uma Lista Estática é uma implementação de uma lista que usa um *array* como estrutura de dados subjacente. Em uma Lista Estática, o tamanho da lista é fixo e pré-definido no momento da sua criação, e não pode ser alterado durante a execução do programa. Isso significa que, uma vez que o tamanho da lista é definido, não é possível adicionar ou remover elementos sem antes realocar a memória usada pela lista.

Um *array* é uma estrutura de dados que armazena uma coleção de elementos do mesmo tipo, cada um identificado por um índice único. Em outras palavras, um *array* é uma sequência contígua de locais de memória que são usados para armazenar elementos.

Os elementos de um *array* são acessados através de seus índices, que começam a partir de zero. Isso significa que, para acessar um elemento em um *array*, é necessário conhecer seu índice e usá-lo para calcular o endereço de memória do elemento.

Os *arrays* são úteis quando é necessário armazenar e acessar uma coleção de elementos de forma eficiente, especialmente quando se precisa acessar elementos aleatoriamente. Além disso, os *arrays* são geralmente mais rápidos do que outras estruturas de dados para

operações como pesquisa e ordenação, pois os elementos estão armazenados contiguamente na memória.

Em muitas linguagens de programação, os *arrays* são um tipo de dados *built-in* e são amplamente utilizados em vários tipos de aplicações.

Um tipo de dados *built-in* é um tipo de dados que é fornecido diretamente pelo sistema ou pelo compilador de uma linguagem de programação e não precisa ser definido ou implementado pelo programador. Em outras palavras, esses tipos de dados são pré-definidos e estão disponíveis para uso imediato na linguagem de programação.

Exemplos de tipos de dados *built-in* incluem inteiros, reais, caracteres, *strings* e *arrays*. Esses tipos de dados são usados para representar diferentes tipos de dados em uma aplicação. Eles geralmente são otimizados para serem eficientes e se integrar perfeitamente com as operações e os recursos da linguagem de programação.

Os tipos de dados *built-in* são diferentes dos tipos de dados definidos pelo usuário, que são definidos e implementados pelo programador para atender às necessidades específicas da aplicação.

A vantagem de uma lista estática é que ela é muito fácil de implementar e usa pouca memória dinâmica. Além disso, o acesso aos elementos na lista é feito em tempo constante, o que é uma vantagem em relação a outras implementações de listas. No entanto, a desvantagem de uma lista estática é que ela não é flexível quanto ao seu tamanho, o que pode levar a desperdício de memória ou erros de falta de memória se o tamanho da lista não for definido corretamente.

As listas estáticas podem ser implementadas de forma sequencial ou encadeada, que serão vistas nas seções a seguir.

Lista Estática Sequencial

Uma Lista Estática Sequencial é uma implementação de uma lista que usa um *array* como estrutura de dados subjacente. Em uma lista estática sequencial, o tamanho da lista é fixo e pré-definido no momento da sua criação, e não pode ser alterado durante a execução do programa.

Os elementos da lista são armazenados contiguamente na memória, e o acesso aos elementos é feito através de seus índices. Isso significa que, para acessar um elemento em uma lista estática sequencial, é necessário conhecer seu índice e usá-lo para calcular o endereço de memória do elemento.

Uma Lista Estática Sequencial tem a vantagem de permitir o acesso rápido aos elementos, já que eles são acessados em tempo constante. No entanto, essa lista tem duas desvantagens. A primeira, relacionada ao fato de ser estática, é que ela tem um tamanho fixo que não pode ser alterado, o que pode resultar em desperdício de memória ou erros de falta de memória se o tamanho não for definido corretamente. A segunda desvantagem, relacionada ao fato dela ser sequencial, é que as operações de inserção e remoção de elementos podem ser demoradas, pois é necessário remanejar outros elementos na lista para alocar ou liberar espaço para o novo elemento ou o elemento removido.

Para implementar nossa TAD Lista Estática Sequencial (vou chamá-la carinhosamente de Les) vamos pensar inicialmente qual seria sua estrutura e as operações que podem ser realizadas sobre ela.

Em Python temos já implementada, como uma classe, a lista, que é uma coleção de elementos que não precisam ser do mesmo tipo. No entanto, as listas em Python não são implementadas como posições contíguas de memória, mas sim como uma coleção de elementos interconectados por ponteiros. As listas em Python são um tipo de dados mutável que pode armazenar elementos de diferentes tipos, como inteiros, *floats*, *strings* e até outras listas. Elas são dinâmicas, ou seja, podem ser alteradas em tempo de execução, permitindo a adição, remoção e modificação de elementos. As listas podem ser acessadas por meio de índices numéricos, semelhante a *arrays* em outras linguagens de programação, embora não sejam implementadas em posições contíguas de memória.

Existe, sim, o conceito de *array* em Python, que é uma estrutura de dados que armazena uma coleção de elementos do mesmo tipo em uma única sequência contígua de memória. No entanto, a implementação de *arrays* em Python é um pouco diferente de outras linguagens de programação, como C e Java, onde os *arrays* são uma estrutura de dados primitiva.

Em Python, o equivalente mais próximo é o objeto `array` do módulo `array`. O objeto `array` permite criar *arrays* homogêneos, ou seja, todos os elementos devem ser do mesmo tipo, como inteiros, *floats*, caracteres etc. Além disso, os *arrays* em Python são tipados, ou seja, você precisa definir o tipo de dados que será armazenado no *array*.

Para facilitar o aprendizado, vamos criar uma estrutura de lista que se comporte de maneira semelhante a uma estrutura estática, sem utilizar o módulo `array` do Python, para que não haja confusão com as estruturas já existentes. Apesar de ser possível alterar o tamanho de uma lista no Python, para fins didáticos, definiremos o tamanho da lista e não o alteraremos posteriormente. Também não utilizaremos os métodos de lista já existentes, como `append()`,

`extend`, `insert`, `remove`, `pop`, `index`, `count`, `sort`, `reverse`, `clear`, `copy`, `len`. Em vez disso, implementaremos nossas próprias funções (ou métodos) para realizar algumas dessas tarefas em nossa implementação personalizada de lista.

Vamos dar início à implementação criando o método de inicialização de nosso TAD `Les` e, a partir dele, entender a sua proposta.

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


Seguindo nossa proposta, temos a possibilidade do usuário (sim, os programadores que farão utilizarão nosso TAD é que serão nossos usuários) definir, ao criar a lista, o seu tamanho máximo, que será fixo pois, lembremo-nos, é uma lista estática, ou seja, seu tamanho não cresce em tempo de execução.

No código apresentado, uma lista de `Nones` é criada para inicializar a lista com valores padrão. O tamanho da lista é especificado no momento da criação do objeto da classe, e a lista é inicializada com o mesmo tamanho, preenchida com o valor `None`.

A criação da lista de `Nones` é necessária porque a lista tem tamanho fixo, ou seja, uma vez que é criada com um tamanho específico, esse tamanho não pode ser alterado posteriormente. Porém, ao criar uma lista de tamanho fixo, é possível que os elementos dessa lista ainda não tenham sido inseridos. Portanto, ao criar uma lista de `Nones`, cada posição do vetor é inicializada com um valor padrão que representa a ausência de um elemento válido.

A inicialização da lista com `Nones` permite que cada posição no vetor tenha um valor inicial e evita a necessidade de verificar se uma posição específica na lista está vazia ou não. Além disso, essa técnica também ajuda a evitar possíveis erros ou comportamentos inesperados que podem ocorrer se a lista for inicializada com valores arbitrários que possam confundir o programador.

Ao instanciar um objeto da classe `Les`, que representa uma lista estática sequencial, as informações da lista ficariam como mostrado a seguir.

Suponha que criamos uma lista estática sequencial com tamanho máximo igual a 5, fazendo:

```
lista = Les(5)
```

A partir dessa instanciação, temos:

`tam_maximo: 5`

representa o tamanho máximo da lista.

`vetor: [None, None, None, None, None]`

é uma lista que armazena os elementos da lista, onde cada elemento é inicializado como None para representar a ausência de um elemento válido.

`quant: 0`

é a quantidade atual de elementos na lista, inicialmente definida como zero.

Assim, podemos entender a representação internada da lista criada da seguinte forma:

0	1	2	3	4
None	None	None	None	None

`tam_maximo = 5`

`quant = 0`

Essa representação mostra a lista estática sequencial criada com 5 posições, inicializadas com o valor `None`, e o tamanho máximo da lista é definido como 5 e a quantidade atual de elementos é definida como 0.

Vamos agora implementar a função `inserir_fim(self, valor)`, para que possamos inserir um valor no final da lista.

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

A função `inserir_fim()` é responsável por adicionar um novo elemento ao final da lista estática sequencial. Ela recebe como argumento o valor do elemento que será adicionado.

A primeira linha da função `self.vetor[self.quant] = valor` insere o novo elemento no final da lista. O índice utilizado para inserção é o valor da variável `self.quant`, que é atualizada em seguida. O motivo de utilizar a variável `self.quant` é para que os elementos sejam inseridos em sequência, uma vez que a lista é estática, e o número de elementos é limitado.

A segunda linha `self.quant += 1` é responsável por incrementar a variável `self.quant`, que armazena a quantidade de elementos atualmente na lista estática. Esse incremento permite que a próxima inserção de elemento seja feita na posição seguinte na lista.

Dessa forma, a função `inserir_fim` permite adicionar um novo elemento no final da lista estática sequencial de forma eficiente, mantendo a ordem dos elementos e evitando a necessidade de deslocamento de elementos, o que poderia prejudicar o desempenho do código.

E quando a lista está vazia? Neste caso o conceito de "fim da lista" é indeterminado, uma vez que não há elementos na lista para definir uma posição final.

No entanto, na implementação de uma lista estática sequencial, é comum definir que o "fim da lista" é a posição logo após o último elemento válido na lista. Nesse caso, quando a lista está vazia, o "fim da lista" é a posição inicial, ou seja, a posição 0.

Dessa forma, ao inserir o primeiro elemento na lista estática sequencial vazia usando a função `inserir_fim`, o elemento é inserido na posição 0 e o "fim da lista" passa a ser a posição 1, que será a posição para inserção do próximo elemento.

Portanto, a função `inserir_fim` é capaz de lidar com o caso em que a lista estática sequencial está vazia, adicionando o primeiro elemento na posição 0 e atualizando o "fim da lista" para a posição 1.

Neste momento sentimos necessidade de depurar o código e pode prever como se dará a exibição de informações para o usuário.

Para isso o código apresentado a seguir define uma função `show(self)` para exibir os elementos armazenados na lista estática sequencial.

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

A função utiliza um *loop* for para percorrer todos os elementos da lista até a quantidade atual de elementos, representada pela variável `self.quant`. Essa variável armazena o número de elementos atualmente presentes na lista e garante que apenas os elementos válidos sejam exibidos.

Dentro do loop, a função utiliza a função `print` para exibir o elemento atual na lista, representado pela expressão `self.vetor[i]`. O parâmetro `end='  '` é utilizado para separar os elementos exibidos por um espaço em branco, em vez de quebrar a linha. Por fim, a função `print` é utilizada novamente para imprimir uma nova linha vazia, garantindo que a próxima chamada à função `print` seja exibida em uma nova linha.

Dessa forma, a função `show` permite exibir os elementos armazenados na lista estática sequencial em ordem, separados por um espaço em branco.

Vamos agora fazer um pequeno teste para visualizar o que acontece ao executar estas duas funções.


```
import Les

# Inicializa a lista estática sequencial com tamanho máximo
# igual a 5
lista = Les.Les(5)

# Insere o elemento 1 na lista
lista.inserir_fim(1)
lista.show()

# Insere o elemento 3 na lista
lista.inserir_fim(3)
lista.show()

# Insere o elemento 5 na lista
lista.inserir_fim(5)
lista.show()

# Insere o elemento 7 na lista
lista.inserir_fim(7)
lista.show()

# Insere o elemento 9 na lista
lista.inserir_fim(9)
lista.show()
```

A saída da execução deste código será:

```
1

1 3

1 3 5

1 3 5 7

1 3 5 7 9
```

Acompanhando a representação interna, a lista seria assim modificada a cada execução da função `inserir_fim()`:

0	1	2	3	4
1	None	None	None	None

```
tam_maximo = 5
quant = 1
```

0	1	2	3	4
1	3	None	None	None

```
tam_maximo = 5
quant = 2
```

0	1	2	3	4
1	3	5	None	None

```
tam_maximo = 5
quant = 3
```

0	1	2	3	4
1	3	5	7	None

```
tam_maximo = 5
quant = 4
```

0	1	2	3	4
1	3	5	7	9

```
tam_maximo = 5
quant = 5
```

Passaremos agora para a implementação de uma função igualmente simples, a `remover_fim(self)`, que objetiva remover o elemento do fim da lista. Esta implementação servirá também para ilustrarmos a diferença entre o que nós, criadores do TAD, "vemos" da estrutura interna da lista, e o que nosso usuário, o programador, espera da sua execução.

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

A função `remover_fim` é responsável por remover o último elemento da lista estática sequencial. Essa função não recebe nenhum parâmetro, pois ela sempre remove o último elemento válido da lista.

Na primeira linha da função `remover_fim`, é feita a operação `self.quant -= 1`. Essa operação é responsável por diminuir em 1 a quantidade de elementos atualmente presentes na lista.

A variável `self.quant` representa a quantidade atual de elementos na lista, portanto, quando é subtraído 1 dessa variável, é removido o último elemento inserido na lista. Isso acontece porque a lista estática sequencial é uma estrutura de dados que mantém os elementos em sequência, e a ordem dos elementos é importante.

Ao remover o último elemento, a quantidade de elementos válidos na lista diminui em 1, e o "fim da lista" é atualizado para a posição anterior, que passa a ser o último elemento válido na lista. Dessa forma, o elemento que foi removido é marcado como inválido, mas ainda permanece presente na lista, porém na posição anterior ao "fim da lista".

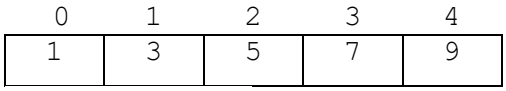
É importante notar que essa operação não remove o elemento em si, apenas o torna inválido. É possível que outros elementos possam ser adicionados na lista posteriormente, preenchendo a posição do elemento removido.

Para exemplificar o que foi dito acima, vamos utilizar a função show para exibir o estado da lista antes e depois da chamada da função `remover_fim`.

Suponha que temos a seguinte lista estática sequencial de inteiros:

1	3	5	7	9

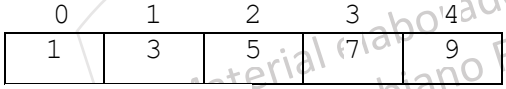
Na estrutura interna da lista tem-se as seguintes informações:




```
tam_maximo = 5
quant = 5
```

A quantidade de elementos válidos na lista é 5, que é o tamanho máximo da lista.

Se chamarmos a função `remover_fim`, a primeira linha da função irá subtrair 1 da variável `self.quant`, resultando em um novo valor para a variável `self.quant` igual a 4, que representa a nova quantidade de elementos válidos na lista. A sua estrutura interna ficará assim:




```
tam_maximo = 5
quant = 4
```


A lista, do ponto de vista do usuário, ficará da seguinte forma:

Note que o último elemento, 9, do ponto de vista do usuário, não está mais presente na lista, pois foi removido pela função `remover_fim`. Entretanto ele permanece na estrutura interna. Porém, como a variável `self.quant` agora tem valor 4, isso indica que a quantidade de elementos válidos na lista foi atualizada e, ao chamarmos a função `show()`, teremos a seguinte saída (para o usuário):

```
1  3  5  7
```


Note que o último elemento, 9, não foi exibido, pois ele foi removido da lista. A ordem dos elementos na lista foi mantida, uma vez que a função `remover_fim` remove apenas o último elemento inserido na lista. Dessa forma, a operação `self.quant -= 1` é responsável por remover o último elemento inserido na lista, enquanto mantém a ordem dos elementos na lista estática sequencial.

Precisamos agora dar uma paradinha e abrir um grande parêntese.

Você deve ter notado que as funções `inserir_fim` e `remover_fim` não tratam da possibilidade da lista estar cheia (antes de tentar inserir mais um valor) ou mesmo de estar vazia (antes de tentar remover).

Sim, estas funções poderiam ser assim implementadas:

```
    def inserir_fim(self, valor):
        if self.quant == self.tam_maximo:
            print("A lista está cheia")
        else:
            self.vetor[self.quant] = valor
            self.quant += 1

    def remover_fim(self):
        if self.quant == self.tam_maximo:
            print("A lista está vazia")
        else:
            self.quant -= 1
```


Os códigos apresentados interferem na forma de implementação do programador que usará o TAD da lista estática sequencial ao imprimir mensagens de erro diretamente no console da aplicação. Isso pode ser prejudicial para a interface do usuário e também para a manutenção do código, uma vez que pode dificultar o tratamento de erros e o desenvolvimento de testes automatizados.

Ao invés de imprimir mensagens de erro diretamente no console, é mais apropriado lançar exceções em caso de erro, permitindo que a aplicação capture essas exceções e realize um tratamento mais adequado.

Por exemplo, ao invés de imprimir uma mensagem de erro na função `inserir_fim` quando a lista está cheia, poderíamos lançar uma exceção `ListaCheiaException`. Dessa forma, o usuário da biblioteca poderia capturar essa exceção e realizar o tratamento adequado, sem comprometer a interface do usuário ou a manutenção do código.

Abaixo está uma sugestão de como ficaria nosso TAD com a implementação das funções `inserir_fim` e `remover_fim` utilizando exceções:

```
class ListaCheiaException(Exception):
    pass

class ListaVaziaException(Exception):
    pass
```

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

    def inserir_fim(self, valor):
        if self.quant == self.tam_maximo:
            raise ListaCheiaException("A lista está cheia")
        else:
            self.vetor[self.quant] = valor
            self.quant += 1

    def remover_fim(self):
        if self.quant == 0:
            raise ListaVaziaException("A lista está vazia")
        else:
            self.quant -= 1
```


Com a utilização de exceções, a interface do usuário e a manutenção do código são melhoradas, tornando o código mais seguro e fácil de ser utilizado. Segue um exemplo de programa que usa o TAD Les e trata as exceções `ListaCheiaException` e `ListaVaziaException`.


```
Import Les
lista = Les.Les(5)

try:
    lista.inserir_fim(1)
    lista.inserir_fim(2)
    lista.inserir_fim(3)
    lista.inserir_fim(4)
    lista.inserir_fim(5)
    lista.inserir_fim(6)
except ListaCheiaException as e:
```

```
    print(f"Erro: {e}")

try:
    lista.remover_fim()
    lista.remover_fim()
    lista.remover_fim()
    lista.remover_fim()
    lista.remover_fim()
    lista.remover_fim()
except ListaVaziaException as e:
    print(f"Erro: {e}")
```


No exemplo acima, é criada uma lista estática sequencial de tamanho 5. São feitas diversas chamadas às funções `inserir_fim` e `remover_fim`, simulando diferentes casos de exceção.

Ao chamar as funções `inserir_fim` e `remover_fim`, o programa envolve as chamadas em um bloco `try-except`, capturando as exceções `ListaCheiaException` e `ListaVaziaException` lançadas pelo TAD `Les`. No bloco `except`, o programa trata as exceções e exibe uma mensagem de erro apropriada.

Não é nosso propósito aqui entrar nos detalhes do tratamento de exceções, mas fica aí um exemplo que pode ser melhor estudado por você caso seja de seu interesse.

Também podemos fazer nosso próprio tratamento entendendo as funções de inserção e remoção como funções booleanas, que retornarão o valor `True` caso atinjam seu objetivo e `False` caso, por algo motivo, não seja possível fazer a devida inserção ou remoção.

Vamos ver como poderia ficar:


```
    def inserir_fim(self, valor):
        if self.quant != tam_maximo:
            self.vetor[self.quant] = valor
            self.quant += 1
            return True
        return False

    def remover_fim(self):
```

```
        f self.quant != 0:
            self.quant -= 1
            return True
        return False
```

A função `inserir_fim` recebe um valor como parâmetro e verifica se a quantidade de elementos na lista (`self.quant`) é menor do que o tamanho máximo da lista (`self.tam_maximo`). Se a quantidade de elementos for menor do que o tamanho máximo, o valor é inserido na última posição da lista (índice `self.quant`) e a quantidade de elementos é incrementada em 1 (linha `self.quant += 1`). Nesta implementação, a função retorna `True` para indicar que a operação foi realizada com sucesso. Caso contrário, a função não insere o valor e retorna `False` indicando que a operação falhou.

A função `remover_fim` verifica se a quantidade de elementos na lista (`self.quant`) é maior do que zero. Se a quantidade de elementos for maior do que zero, o último elemento da lista (índice `self.quant-1`) é removido e a quantidade de elementos é decrementada em 1 (linha `self.quant -= 1`). Agora, a função retorna `True` para indicar que a operação foi realizada com sucesso. Caso contrário, a função não remove nenhum elemento e retorna `False` indicando que a operação falhou.

Essas funções são úteis para permitir a manipulação de elementos em uma lista estática sequencial de forma segura e consistente, garantindo que a lista não estoure e que nenhum elemento seja acessado de forma indevida. Além disso, a função `inserir_fim` retorna um valor booleano indicando se a operação foi bem-sucedida ou não, o que pode ser útil para verificar se a inserção de um elemento foi feita com sucesso. A função `remover_fim` também retorna um valor booleano indicando se a operação foi bem-sucedida ou não, o que pode ser útil para verificar se a remoção de um elemento foi feita com sucesso.

Vamos fazer um programa para testar e mostrar como estas funções seriam implementadas seguindo este modelo.

```
lista = Les.Les(5)

# Inserção de elementos
print(f"Inserção do valor 1: {lista.inserir_fim(1)}")
print(f"Inserção do valor 2: {lista.inserir_fim(2)}")
print(f"Inserção do valor 3: {lista.inserir_fim(3)}")
```

```
print(f"Inserção do valor 4: {lista.inserir_fim(4)}")
print(f"Inserção do valor 5: {lista.inserir_fim(5)}")
print(f"Inserção do valor 6: {lista.inserir_fim(6)}")

# Remoção de elementos
print(f"Remoção do último elemento: {lista.remover_fim()}")
print(f"Remoção do último elemento: {lista.remover_fim()}")
print(f"Remoção do último elemento: {lista.remover_fim()}")
print(f"Remoção do último elemento: {lista.remover_fim()}")
print(f"Remoção do último elemento: {lista.remover_fim()}")
print(f"Remoção do último elemento: {lista.remover_fim()}")
```


Nesse exemplo, é criada uma lista estática sequencial de tamanho 5. São feitas chamadas às funções `inserir_fim` e `remover_fim`, simulando diferentes situações limítrofes, como a inserção de um valor além do tamanho máximo da lista e a remoção de um elemento de uma lista vazia.

O resultado da execução do programa será:

```
Inserção do valor 1: True
Inserção do valor 2: True
Inserção do valor 3: True
Inserção do valor 4: True
Inserção do valor 5: True
Inserção do valor 6: False
Remoção do último elemento: True
Remoção do último elemento: True
Remoção do último elemento: True
Remoção do último elemento: True
Remoção do último elemento: True
Remoção do último elemento: False
```

Note que, sem o tratamento de exceções, o programa simplesmente retorna `True` ou `False` ao chamar as funções `inserir_fim` e `remover_fim` indicando se a operação foi realizada com sucesso ou não. Portanto, é responsabilidade do programador tratar essas situações adequadamente em seu código, caso contrário o programa pode apresentar comportamentos inesperados ou erros.

Sobre a linha `print(f"Inserção do valor 1: {lista.inserir_fim(1)}")`

Essa linha de código é um exemplo de uso de uma `f-string`, uma funcionalidade do Python que permite inserir valores de variáveis dentro de *strings* de forma mais simples e legível.

No exemplo dado, a `f-string` está sendo usada para formatar uma mensagem de saída que indica se a operação de inserção na lista foi bem-sucedida ou não. A mensagem é composta por duas partes:

- "Inserção do valor 1: ": uma *string* fixa que indica qual operação foi realizada (inserção do valor 1);
- `{lista.inserir_fim(1)}`: uma expressão que indica o resultado da operação. A função `inserir_fim` da lista estática sequencial lista é chamado com o argumento 1 para tentar inserir o valor 1 na lista. O resultado da operação é um valor booleano (`True` se a inserção foi bem-sucedida, `False` caso contrário). Esse valor booleano é inserido na *string* formatada usando chaves `{}`.

Dessa forma, ao executar a linha de código, o resultado será uma mensagem de saída do tipo "`Inserção do valor 1: True`" ou "`Inserção do valor 1: False`", indicando se a operação foi bem-sucedida ou não.

Fechando nosso parêntese, voltamos a forma como optamos por implementar aqui:

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

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

E se o usuário quiser remover com a lista vazia? Ou inserir e a lista estiver cheia?

A ideia por trás dessa abordagem é simplificar as funções da lista estática sequencial para que sejam mais fáceis de entender e manter, deixando a responsabilidade pelo tratamento de exceções e outras validações com o usuário da biblioteca.

Com isso, espera-se que o programador que use o TAD possa entender melhor como a estrutura funciona e possa adaptá-la de acordo com as necessidades do seu projeto, sem a necessidade de conhecer os detalhes internos da implementação da lista estática sequencial.

Assim, será necessário adicionar duas funções na lista estática sequencial: `esta_vazia(self)` e `esta_cheia(self)`. O objetivo dessas funções é verificar se a lista está vazia ou cheia antes de inserir ou remover um elemento da lista.

A função `esta_vazia` retorna `True` se a lista está vazia, ou `False` caso contrário. Essa função é útil para evitar que o usuário tente remover um elemento da lista quando ela já estiver vazia, o que poderia causar um erro. De forma análoga, a função `esta_cheia` retorna `True` se a lista está cheia, ou `False` caso contrário sendo útil para evitar que o usuário tente inserir um elemento na lista quando ela já estiver cheia.

Essa abordagem ajuda a manter a coesão das funções, uma vez que cada uma delas é responsável por uma única tarefa específica, o que torna o código mais fácil de entender, testar e manter. Com essas funções implementadas, o usuário da lista estática sequencial pode usar essas funções para verificar se é seguro inserir ou remover um elemento da lista, sem a necessidade de conhecer os detalhes internos de sua implementação. Isso permite que o usuário use a lista de forma mais segura e com menos chance de erros.

Vamos dar prosseguimento com a implementação das funções `inserir_inicio(self, valor)` e `remover_inicio(self)`. Essas funções são simples, mas envolvem a movimentação dos elementos na lista, o que pode ser um pouco trabalhoso.

É importante lembrar que não precisamos nos preocupar em verificar se a lista está vazia ou cheia, pois isso fica por conta do nosso usuário programador. Também devemos ressaltar que, por fins didáticos, não estamos usando as funções de lista do próprio Python, o que significa que a manipulação dos elementos na lista estática sequencial está por nossa conta.

Para implementar a função `inserir_inicio(self, valor)`, precisamos mover todos os elementos da lista uma posição para a direita, para abrir espaço para o novo elemento. Podemos fazer isso utilizando um loop que percorre a lista da direita para a esquerda e copia

cada elemento para a posição seguinte. Depois disso, o novo elemento é inserido na primeira posição da lista.

```
    def inserir_inicio(self, valor):
        for i in range(self.quant, 0, -1):
            self.vetor[i] = self.vetor[i - 1]
        self.vetor[0] = valor
        self.quant += 1
```

Para exemplificar a execução da função `inserir_inicio` vamos considerar a seguinte lista:

3	5	7	9

Na estrutura interna da lista tem-se as seguintes informações:

0	1	2	3	4
3	5	7	9	None

`tam_maximo = 5`

`quant = 4`

Suponha que queremos inserir o valor 1 no início da lista. Ao chamar a função `inserir_inicio(1)`, o elemento 1 será inserido no início da lista. Para isso, será necessário deslocar todos os outros elementos para a direita. A lista e a estrutura interna da lista ficarão da seguinte forma:

Lista

1	3	5	7	9

Estrutura interna

0	1	2	3	4
1	3	5	7	9

`tam_maximo = 5`

`quant = 5`

Note que os elementos já existentes na lista foram deslocados para a direita, para o fim da lista, e o valor 1 foi inserido na posição 0. Além disso, a variável `quant` foi atualizada para refletir a nova quantidade de elementos válidos na lista.

Ao chamar a função `show` do TAD, teremos a seguinte saída:

```
1  3  5  7  9
```

A função `remover_inicio(self)` é similar à função `inserir_inicio(self, valor)`. Precisamos mover todos os elementos da lista uma posição para a esquerda, para preencher o espaço deixado pelo elemento removido. Podemos fazer isso utilizando um *loop* que percorre a lista da esquerda para a direita e copia cada elemento para a posição anterior. Depois disso, a quantidade de elementos válidos na lista é atualizada.

```
  def remover_inicio(self):
        for i in range(self.quant-1):
            self.vetor[i] = self.vetor[i+1]
        self.quant -= 1
```

Vamos supor que temos a seguinte lista estática sequencial de inteiros:

3	5	7	9

Na estrutura interna da lista tem-se as seguintes informações:

0	1	2	3	4
3	5	7	9	None

```
tam_maximo = 5
quant = 4
```

Com a chamada da função `remover_inicio`, ela irá percorrer a lista a partir do índice 1 (posição 2), copiando o valor do índice anterior (posição 1) para o índice atual (posição 0), e assim sucessivamente, até que a lista fique da seguinte forma:

5	7	9

Na estrutura interna da lista tem-se as seguintes informações:

0	1	2	3	4
5	7	9	9	None

```
tam_maximo = 5
quant = 3
```

Note que o elemento 3 foi removido do início da lista e a ordem dos elementos foi mantida; Ao final da função, teremos que `self.quant` é atualizado para 3, que representa a nova quantidade de elementos na lista. Se chamarmos a função `show`, teremos a seguinte saída:

```
5  7  9
```

Você deve ter observado que o valor 9 se mantém na estrutura interna da lista. Isso ocorre porque a função `remover_inicio` apenas movimenta os valores dos elementos para a posição anterior na lista, sem modificar o valor que está sendo movido. Dessa forma, o valor 9, que estava na posição 3, é movido para a posição 2 mas também permanece na lista na antiga posição.

Vamos recordar (já vimos algo parecido ao implementar o `remover_fim`) que o último elemento duplicado não precisa ser substituído porque, no contexto de uma lista estática sequencial, apenas os elementos dentro do intervalo `[0, self.quant - 1]` são considerados válidos. Dessa forma, o elemento duplicado na última posição da lista é simplesmente ignorado pelo usuário da lista, já que a quantidade de elementos válidos é decrementada na linha `self.quant -= 1`.

Assim, o valor 9 permanecer na lista na posição antiga não afeta o comportamento da lista ou de outras operações que possam ser realizadas na lista. Além disso, a estrutura interna da lista não é exposta ao usuário, que tem acesso apenas aos valores presentes na lista e não à sua implementação interna.

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

```
    def ver_primeiro(self):
        return self.vetor[0]

    def ver_ultimo(self):
        return self.vetor[self.quant - 1]
```

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

    def capacidade(self):
        return self.tam_maximo
```

A função `ver_primeiro` retorna o primeiro elemento da lista, que está armazenado na posição 0 do vetor interno da lista.

A função `ver_ultimo` retorna o último elemento válido da lista, que está armazenado na posição `self.quant - 1` do vetor interno da lista. Como a lista estática sequencial é implementada de forma que os elementos são inseridos sequencialmente a partir do índice 0, o último elemento é sempre o que está na posição `self.quant - 1`, que é a última posição do vetor que contém um elemento válido.

A função `tamanho_atual` retorna o número de elementos atualmente armazenados na lista, ou seja, o valor de `self.quant`.

A função `capacidade` retorna o tamanho máximo da lista, ou seja, o valor de `self.tam_maximo`. Esta função pode ser útil, por exemplo, para que o usuário possa verificar se a lista tem capacidade para inserir novos elementos antes de chamar funções de inserção.

Mais algumas funções podem enfeitar nosso TAD (e também organizar nosso conhecimento sobre a movimentação dos valores na simulação de *array* conforme proposto). O desafio aqui é você parar e implementá-las:

- `remover_elemento(self, valor)`: remove o elemento com valor especificado da lista, caso ele exista. Os elementos subsequentes serão movidos para ocupar o espaço deixado pelo elemento removido. Caso o elemento não esteja na lista, a função não faz nenhuma ação.
- `buscar(self, valor)`: busca o valor especificado na lista e retorna o índice da posição em que ele está armazenado. 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.

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