Estruturas de dados lineares

Alocação encadeada

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Como podemos implementar os TAD's Pilha, Fila, Fila Dupla e Lista sem usar arranjos?

Como podemos representar uma quantidade arbitrária de dados sem arranjos?

É isso que vamos ver agora!

Mas antes, vamos falar de valores opcionais.

Valores opcionais

Queremos representar uma pessoa com um nome e uma idade, sendo que a idade é opcional. Também queremos fazer uma função `faz_aniversario` que aumenta a idade de uma pessoa, se a idade está presente, em 1 ano.

```
@dataclass
class Pessoa:
    nome: str
    idade: int
```

Como representar a ausência da idade? Uma opção é usar um valor inválido para a idade.

```
>>> NENHUMA = -1
>>> p1 = Pessoa('Joao', 21)
>>> p2 = Pessoa('Maria', NENHUMA)
>>> p1
Pessoa(nome='Joao', idade=21)
>>> p2
Pessoa(nome='Maria', idade=-1)
```

Agora temos que projetar a função `faz_aniversario`.

Valores opcionais

```
def faz_aniversario(p: Pessoa):
    '''
    Se a idade está presente, aumenta
    a idade da pessoa *p* em 1 ano.

    Exemplos
    >>> p1 = Pessoa('Joao', 21)
    >>> faz_aniversario(p1)
    >>> p1
    Pessoa(nome='Joao', idade=22)
    >>> p2 = Pessoa('Maria', NENHUMA)
    >>> faz_aniversario(p2)
    >>> p2
    Pessoa(nome='Maria', idade=-1)
    '''

    if p.idade != NENHUMA:
        p.idade += 1
```

O que aconteceria se esquecêssemos de fazer a verificação se a idade está presente? O teste iria falhar...

E se não tivéssemos teste? O programa executaria mas produziria resultados incorretos.

Porque esse erro possível?

Esse tipo de erro só é possível porque estamos usando um valor do mesmo tipo para representar a ausência de valor, então, qualquer operação válida para os valores do tipo também é válida para o valor que representa a ausência de valor!

Existe mais algum problema com essa estratégia?

Sim, o leitor vê a definição `idade: int` e supõe que a idade é requerida, só entendendo que é opcional se isso estiver escrito como comentário.

Podemos fazer melhor? Sim!

Valores opcionais

Em Python existe um valor especial chamado `None` (do tipo `None` – sim, o tipo e o valor do tipo tem o mesmo nome!), que fica armazenado em uma célula de memória específica, que é usado para representar a ausência de um valor.

Para que uma variável possa referenciar o valor `None`, é preciso informar isso na declaração do tipo da variável, por exemplo `a: int | None`. Está declaração está dizendo que variável `a` pode referenciar uma célula com um inteiro ou com `None`.

Note que é possível declarar uma variável apenas do tipo `None`, por exemplo `a: None`, mas isso não faz muito sentido pois o único valor válido para `a` seria `None`!

```
>>> # a pode referenciar um inteiro ou None
>>> a: int | None = 20
>>> a
20
>>> a = None
>>> a
```

```
>>> a = 30
>>> a
30
>>> # b só pode referenciar o valor None!
>>> b: None = None
>>> b
```

Como o uso do `None` muda o código?

```
@dataclass
class Pessoa:
    nome: str
    idade: int | None
```

A intenção está clara, `idade` pode ser um inteiro ou `None`.

```
>>> p1 = Pessoa('Joao', 21)
>>> faz_aniversario(p1)
>>> p1
Pessoa(nome='Joao', idade=22)
>>> p2 = Pessoa('Maria', None)
>>> faz_aniversario(p2)
>>> p2
Pessoa(nome='Maria', idade=None)
```

Os exemplos também ficam mais claros.

```python
def faz_aniversario(p: Pessoa):
    if p.idade is not None:
        p.idade += 1
```

O que aconteceria se esquecêssemos de fazer a verificação se a idade está presente?

O `mypy` iria gerar um erro:

```
pessoa.py:23: error: Unsupported operand types for + ("None" and "int")  [operator]
pessoa.py:23: note: Left operand is of type "int | None"
Found 1 error in 1 file (checked 1 source file)
```

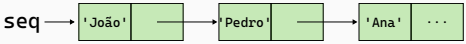
O que ganhamos com isso?

Antes era possível cometer um erro incrementando a idade quando ela não estivesse presente, agora isso não é mais possível! Além disso, a detecção do erro acontece de forma estática, sem precisar executar o programa (como é feito nos testes).

Como podemos representar uma quantidade arbitrária de dados sem arranjos?

Suponha que queremos representar uma coleção de nomes de pessoas. Podemos fazer isso usando estruturas. A ideia é criar uma estrutura com um nome de uma pessoa e uma referência para outra instância da mesma estrutura, que conterá o nome da próxima pessoa e uma referência para outra instância da mesma estrutura...

```
from __future__ import annotations
@dataclass
class Seq:
    nome: str
    proximo: Seq
```



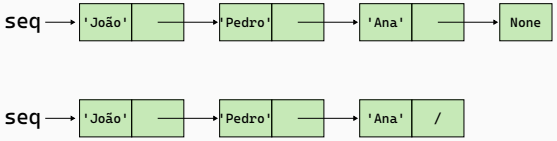

```
>>> # Queremos representar a coleção
>>> # com os nomes 'Joao', 'Pedro' e 'Ana'.
>>> seq = Seq('Joao', Seq('Pedro', Seq('Ana', ...)))
```

O que está faltando? Uma forma de encerrar a sequência!

```python
@dataclass
class Seq:
    nome: str
    proximo: Seq | None
```

```
>>> # Queremos representar a coleção
>>> # com os nomes 'Joao', 'Pedro' e 'Ana'.
>>> seq = Seq('Joao', Seq('Pedro', Seq('Ana', None)))
```

Na representação gráfica podemos utilizar / para indicar uma referência para None



O que têm de diferente na declaração de **Seq** em relação as classes que definimos anteriormente? Uma **autorreferência**, ou seja, a utilização da classe em sua própria definição.

Encadeamento

Os tipos com autorreferência (ou recursivos) permitem a representação de quantidade de dados arbitrárias pelo **encadeamento** de instâncias do tipo. Usamos `None` para representar o fim do encadeamento.

O tipo utilizado no encadeamento é comumente chamado de **No**, dessa forma, usamos um encadeamento de nós para criar uma coleção de valores.

```
@dataclass
class No:
    item: int
    prox: No | None
```

```
>>> p = No('Joao', No('Pedro', No('Ana', None)))
```

Antes de prosseguirmos, vamos revisar o uso de múltiplas referências para a mesma célula de memória.

Vimos que em Python toda variável referencia uma célula de memória. Em algumas situações, como quando atribuímos uma variável para outra ou passamos uma variável como parâmetro, temos mais de uma variável referenciando a mesma célula de memória.

Essa situação pode gerar alguns dificuldades para a escrita e entendimento do código, mas é necessária para manipulação de encadeamentos.

Vamos usar o Python Tutor para visualizar algumas situações de múltiplas referências.

O Python Tutor não aceita o uso de `@dataclass`, então vamos definir uma classe “normal” e definir um construtor manualmente.

Múltiplas referências

```python
@dataclass
class Ponto:
    x: int
    y: int

@dataclass
class Retangulo:
    canto: Ponto
    largura: int
    altura: int
```

```python
class Ponto:
    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y

class Retangulo:
    def __init__(self, canto: Ponto,
                       largura: int,
                       altura: int):
        self.canto = canto
        self.largura = largura
        self.altura = altura
```

Acesse esse exemplos no Python Tutor.

Múltiplas referências

```python
class Ponto:
    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y

class Retangulo:
    def __init__(self, canto: Ponto,
                       largura: int,
                       altura: int):
        self.canto = canto
        self.largura = largura
        self.altura = altura

p = Ponto(10, 50)
l = 200
a = 450

r1 = Retangulo(p, l, a)
r2 = Retangulo(p, l, a)
```

```python
# Quais valores serão exibidos?
l = 300
print(r1.largura)
print(r2.largura)

# Quais valores serão exibidos?
p.x = 20
print(r1.canto.x, r1.canto.y)
print(r2.canto.x, r2.canto.y)

# Quais valores serão exibidos?
r1.canto.y = 70
print(p.x, p.y)
print(r2.canto.x, r2.canto.y)

# Quais valores serão exibidos?
r2.canto = Ponto(3, 17)
print(p.x, p.y)
print(r1.canto.x, r1.canto.y)
```

Manipulação de encadeamento

```
@dataclass
class No:
    item: int
    prox: No | None
```

Defina uma variável p com um encadeamento de nós com os valores 10, 4, 1.

```
>>> p = No(10, No(4, No(1, None)))
```

Escreva expressões para acessar o primeiro, o segundo e o terceiro item do encadeamento.

```
>>> p.item
10
>>> p.prox.item
4
>>> p.prox.prox.item
1
```

Modifique o segundo item para 7.

```
>>> p.prox.item = 7
```

Adicione um No com o item 2 no início.

```
>>> p = No(2, p)
```

Adicione um No com o item 20 no final.

```
>>> p.prox.prox.prox.prox = No(20, None)
```

Usando repetição!

```
>>> q = p
>>> while q.prox is not None:
...     q = q.prox
>>> q.prox = No(20, None)
```

Implementação de pilha

Como podemos implementar uma pilha usando um encadeamento de nós?

Usamos uma variável `topo` para armazenar o primeiro nó do encadeamento ou `None` se a pilha estiver vazia:

- Construtor: inicializa `topo` com `None`
- Vazia: verifica se `topo` é `None`
- Empilha: insere um novo nó com o item no início do encadeamento e muda `topo` para o novo início
- Desempilha: remove o primeiro nó do encadeamento e muda `topo` para o novo início

```
class Pilha:
    topo: No | None

    def empilha(self, item: str):
        self.topo = No(item, self.topo)

    def desempilha(self) -> str:
        if self.topo is None:
            raise ValueError('pilha vazia')
        item = self.topo.item
        self.topo = self.topo.prox
        return item
```

Qual a complexidade de tempo de `empilha` e `desempilha`? $O(1)$.

Implementação de Fila

Como podemos implementar uma fila usando um encadeamento de nós?

Usamos uma variável `inicio` para armazenar o primeiro nó do encadeamento ou `None` se a fila estiver vazia:

- Construtor: inicializa `inicio` com `None`
- Vazia: verifica se `inicio` é `None`
- Enfileira: insere um novo nó com o item no final do encadeamento
- Desenfileira: remove o primeiro nó do encadeamento e muda `inicio` para o novo início

```python
class Fila:
    inicio: No | None

    def enfileira(self, item: str):
        if self.inicio is None:
            self.inicio = No(item, None)
        else:
            # Encontra o último nó
            p = self.inicio
            while p.prox is not None:
                p = p.prox
            p.prox = No(item, None)

    def desenfileira(self) -> str:
        if self.inicio is None:
            raise ValueError('fila vazia')
        item = self.inicio.item
        self.inicio = self.inicio.prox
        return item
```

Implementação de Fila

```
class Fila:
    inicio: No | None

    def enfileira(self, item: str):
        if self.inicio is None:
            self.inicio = No(item, None)
        else:
            # Encontra o último nó
            p = self.inicio
            while p.prox is not None:
                p = p.prox
            p.prox = No(item, None)

    def desenfileira(self) -> str:
        if self.inicio is None:
            raise ValueError('fila vazia')
        item = self.inicio.item
        self.inicio = self.inicio.prox
        return item
```

Qual a complexidade de tempo de `desenfileira`? $O(1)$.

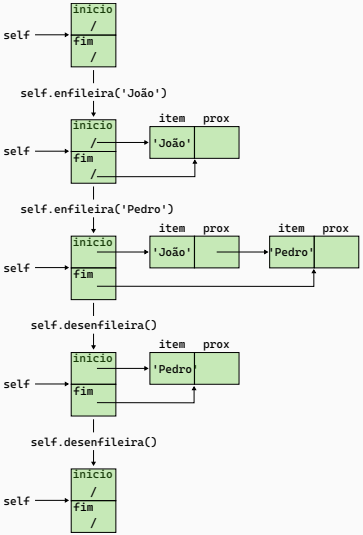
Qual a complexidade de tempo de `enfileira`? $O(n)$...

Podemo fazer melhor? Sim!

Vamos manter uma variável `fim` que referencia o último nó do encadeamento ou é `None` se a fila estiver vazia. Isso permite acessar o fim em tempo constante.

Ambos `inicio` e `fim` são considerados em `enfileira` e `desenfileira`.

Implementação de Fila




```python
class Fila:
    inicio: No | None
    fim: No | None

    def enfileira(self, item: str):
        if self.fim is None:
            self.inicio = No(item, None)
            self.fim = self.inicio
        else:
            self.fim.prox = No(item, None)
            self.fim = self.fim.prox

    def desenfileira(self) -> str:
        if self.inicio is None:
            raise ValueError('fila vazia')
        item = self.inicio.item
        self.inicio = self.inicio.prox
        if self.inicio is None:
            self.fim = None
        return item
```

Implementação de Fila

```
class Fila:
    inicio: No | None
    fim: No | None

    def enfileira(self, item: str):
        if self.fim is None:
            self.inicio = No(item, None)
            self.fim = self.inicio
        else:
            self.fim.prox = No(item, None)
            self.fim = self.fim.prox

    def desenfileira(self) -> str:
        if self.inicio is None:
            raise ValueError('fila vazia')
        item = self.inicio.item
        self.inicio = self.inicio.prox
        if self.inicio is None:
            self.fim = None
        return item
```

Qual a complexidade de tempo de `desenfileira`? $O(1)$.

Qual a complexidade de tempo de `enfileira`? $O(1)$.

Implementação de Fila Dupla

Como podemos implementar uma fila dupla usando um encadeamento de nós?

Precisamos implementar inserção e remoção nos dois extremos.

Mantendo `inicio` e `fim`, quais são as complexidades de tempos das operações?

Inserir no início: $O(1)$ (como `Pilha.empiha` – atualiza `fim` se necessário)

Remover do início: $O(1)$ (como `Fila.desenfileira`)

Inserir no fim: $O(1)$ (como `Fila.enfileira`)

Remover do fim: $O(n)$! É preciso localizar, a partir do início, o predecessor do fim no encadeamento.

```
def remove_fim(self) -> str:
    if self.fim is None:
        raise ValueError('fila vazia')

    # Salva o último elemento
    item = self.fim.item

    p = self.inicio
    assert p is not None
    if p.prox is None: # Único elemento?
        self.inicio = None
        self.fim = None
    else:
        # Encontra o penúltimo
        while p.prox is not self.fim:
            p = p.prox
        p.prox = None
        self.fim = p
    # Devolve o item
    return item
```

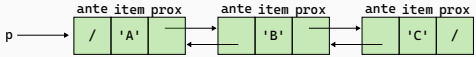
Podemos fazer melhor? Ou seja, podemos fazer uma implementação em que a remoção do fim seja constante? Sim!

Precisamos de um encadeamento duplo. Cada nó mantém, além de uma referência opcional para o próximo, também uma referência opcional para o nó anterior no encadeamento. Dessa forma é possível encontrar o antecessor de um nó em tempo constante.

```
@dataclass
class No:
    ante: No | None
    item: str
    prox: No | None
```

Trabalhar com encadeamento duplo requer ainda mais cuidado do que com encadeamento simples! Por isso é importante **fazer desenhos**!

Escreva o código para criar o seguinte encadeamento




```
>>> a = No(None, 'A', None)
>>> b = No(None, 'B', None)
>>> c = No(None, 'C', None)
>>> a.prox = b
>>> b.ante = a
>>> b.prox = c
>>> c.prev = b
>>> p = a
```

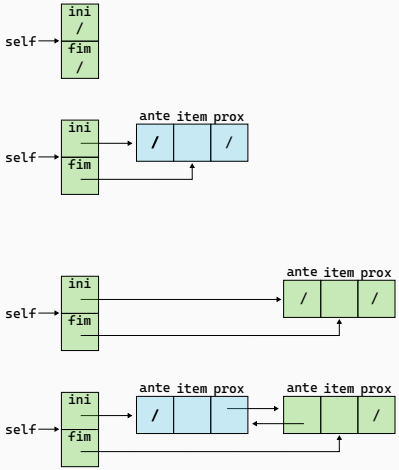
Note que como o encadeamento tem ciclos, ele não pode ser criado todo de uma vez. A estratégia que usamos foi criar os nós separados e depois ligá-los.

Na hora de exibir um encadeamento com ciclos, o Python usa `...` para evitar exibir o mesmo nó mais que uma vez.

```
>>> p
No(ante=None, item='A', prox=No(ante=..., item='B', prox=No(ante=..., item='C', prox=None)))
>>> b
No(ante=No(ante=None, item='A', prox=...), item='B', prox=No(ante=..., item='C', prox=None))
>>> c
No(ante=No(ante=No(ante=None, item='A', prox=...), item='B', prox=...), item='C', prox=None)
```

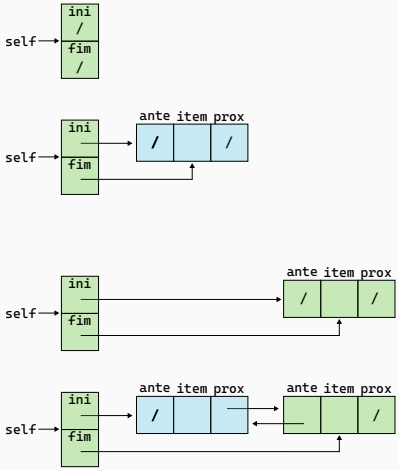
Agora vamos implementar uma fila dupla usando encadeamento duplo mantendo referências para o início e fim do encadeamento.

Fila dupla - Inserção e remoção no início (versão didática)



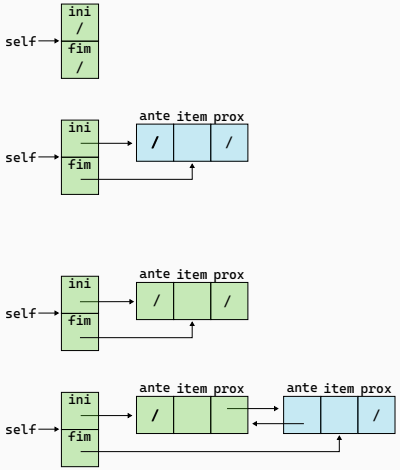

```python
def insere_inicio(self, item: str):
    if self.inicio is None:
        self.inicio = No(None, item, None)
        self.fim = self.inicio
    else:
        self.inicio.ante = No(None, item, self.inicio)
        self.inicio = self.inicio.ante

def remove_inicio(self) -> str:
    if self.inicio is None:
        raise ValueError('fila vazia')
    item = self.inicio.item
    # Só tem um nó?
    if self.inicio.prox is None:
        self.inicio = None
        self.fim = None
    else:
        self.inicio = self.inicio.prox
        self.inicio.ante = None
    return item
```

Fila dupla - Inserção e remoção no início (versão direta)



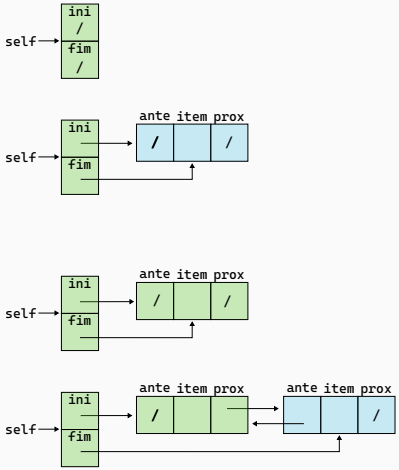

```python
def insere_inicio(self, item: str):
    self.inicio = No(None, item, self.inicio)
    if self.inicio.prox is None:
        self.fim = self.inicio
    else:
        self.inicio.prox.ante = self.inicio

def remove_inicio(self) -> str:
    if self.inicio is None:
        raise ValueError('fila vazia')

    item = self.inicio.item

    self.inicio = self.inicio.prox
    if self.inicio is None:
        self.fim = None
    else:
        self.inicio.ante = None

    return item
```

Fila dupla - Inserção e remoção no fim (versão didática)




```python
def insere_fim(self, item: str):
    if self.fim is None:
        self.inicio = No(None, item, None)
        self.fim = self.inicio
    else:
        self.fim.prox = No(self.fim, item, None)
        self.fim = self.fim.prox

def remove_fim(self) -> str:
    if self.fim is None:
        raise ValueError('fila vazia')
    item = self.fim.item
    # Só tem um nó?
    if self.fim.ante is None:
        self.inicio = None
        self.fim = None
    else:
        self.fim = self.fim.ante
        self.fim.prox = None
    return item
```

Fila dupla - Inserção e remoção no fim (versão direta)




```
def insere_fim(self, item: str):
    self.fim = No(self.fim, item, None)
    if self.fim.ante is None:
        self.inicio = self.fim
    else:
        self.fim.ante.prox = self.fim

def remove_fim(self) -> str:
    if self.fim is None:
        raise ValueError('fila vazia')

    item = self.fim.item

    self.fim = self.fim.ante
    if self.fim is None:
        self.inicio = None
    else:
        self.fim.prox = None

    return item
```

Com encadeamento duplo e referência para início e fim, os métodos do TAD de fila dupla têm complexidade de tempo de $O(1)$.

No entanto, a implementação parece complicada, cada um dos quadro métodos tem dois casos distintos.

Podemos simplificar o código? O que faz com que seja necessário dois casos?

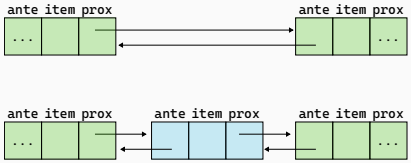
Vamos supor por um momento que todos os nós tenham antecessor e sucessor.

Como remover um nó `p` sabendo que existe um antecessor e um sucessor de `p`?

Como inserir um nó `novo` após um nó `p` sabendo que `p` tem um sucessor?

Como inserir um nó `novo` antes de um nó `p` sabendo que `p` tem um antecessor?

Fila dupla




```
def remove(p: No):
    p.prox.ante = p.ante
    p.ante.prox = p.prox

def insere_depois(p: No, novo: No):
    novo.ante = p
    novo.prox = p.prox
    p.prox.ante = novo
    p.prox = novo

def insere_antes(p: No, novo: No):
    novo.ante = p.ante
    novo.prox = p
    p.ante.prox = novo
    p.ante = novo
```

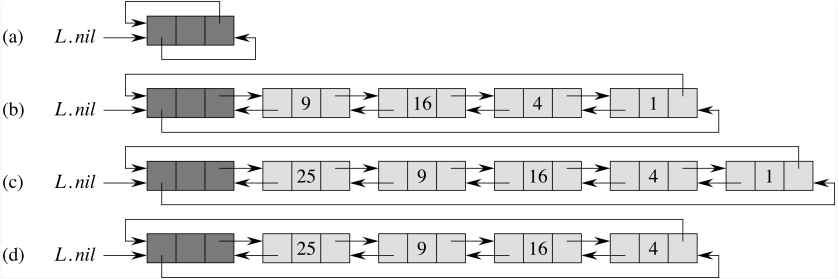
Não precisamos de dois métodos para inserir!

```
def insere_antes(p: No, novo: No):
    insere_depois(p.ante, novo)
```

Como podemos fazer para que cada nó tenha um antecessor e um sucessor?

Usamos uma **sentinela**, um nó especial, que é usando onde o valor `None` seria usado normalmente. Ou seja, a sentinela fica entre o primeiro e o último nó do encadeamento.

O resultado é comumente chamado de **lista circular duplamente encadeada com sentinela**! Na figura abaixo a `L` e o `self` e a sentinela é o `nil`.



Fila Dupla com sentinela

Como implementar o TAD de fila dupla com esse esquema?

Nesse esquema o `ante` e o `prox` não podem ser `None`, então precisamos mudar a definição de `No`:

```
@dataclass
class No:
    ante: No
    item: str
    prox: No

    def __init__(self, item: str) -> None:
        # Após a criação de um nó temos a responsabilidade
        # de alterar ante e prox para valores válidos!
        self.ante = None # type: ignore
        self.item = item
        self.prox = None # type: ignore
```

Mas isso cria um problema, que é a impossibilidade de instanciar um `No`! Conforme discutimos em sala, vamos usar uma inicialização em duas etapas, na primeira um `No` é criado com valores temporários `None` para `ante` e `prox` (usamos `# type: ignore` para que o `mypy` não indique o erro) e depois mudamos para os valores corretos.

Fila Dupla com sentinela

Tendo as funções auxiliares de inserção e remoção de um nó, como podemos implementar inserção e remoção do início e fim de uma fila com sentinela?

```
def remove(p: No) -> str:
    '''Remove *p* do seu encademaneto
       e devolve o item em *p*.'''
    item = p.item
    p.prox.ante = p.ante
    p.ante.prox = p.prox
    return item

def insere_depois(p: No, novo: No):
    '''Insere *novo* após *p* no
       encademaneto.'''
    novo.ante = p
    novo.prox = p.prox
    p.prox.ante = novo
    p.prox = novo
```

```
class FilaDupla:
    sentinela: No
    def __init__(self) -> None:
        self.sentinela = No('')
        self.sentinela.ante = self.sentinela
        self.sentinela.prox = self.sentinela
    def vazia(self) -> bool:
        return self.sentinela.prox is self.sentinela

    def insere_inicio(self, item: str):
        insere_depois(self.sentinela, No(item))

    def insere_fim(self, item: str):
        insere_depois(self.sentinela.ante, No(item))

    def remove_inicio(self, item: str) -> str:
        assert not self.vazia()
        return remove(self.sentinela.prox)

    def remove_fim(self, item: str) -> str:
        assert not self.vazia()
        return remove(self.sentinela.ante)
```

Devemos usar encadeamento simples ou duplo para implementar o TAD Lista?

Se o TAD de Lista não define função específica para remoção do fim, então o encadeamento simples é suficiente.

Qual o tempo de execução para operações de inserção e remoção em posição? $O(n)$, pois é preciso seguir o encadeamento até a posição especificada, que pode ser a última.

A implementação do TAD Lista fica como exercício!

Vimos quatro TAD's e como implementá-los usando arranjos (alocação contígua) e encadeamento de nós (alocação encadeada)

- Pilha
- Fila
- Fila Dupla
- Lista

Estrutura / Operação	get/set	ins/rem início	ins/rem fim	ins/rem	busca
Encadeamento Simples	$O(n)$	$O(1)$ / $O(1)$	$O(1)$ / $O(n)$	$O(n)$	$O(n)$
Encadeamento Duplo	$O(n)$	$O(1)$ / $O(1)$	$O(1)$ / $O(1)$	$O(n) - O(1)$ [1]	$O(n)$
Arranjo Dinâmico	$O(1)$	$O(n)$ / $O(n)$	$O(1)$[2] / $O(1)$	$O(n)$	$O(n)$

[1] Com a referência para o nó

[2] Amortizado

Alocação contígua versus encadeada

Característica	Contígua	Encadeada
Implementação	Simples	Elaborada
Aumento	Realocação	Criação de nó
Diminuição	Deslocamento / realocação	Remoção de nó
Acesso aleatório	Sim	Não

Referências

Capítulo 7, 8, 9 - Pilhas, filas e listas - Fundamentos de Python: Estruturas de dados. Kenneth A. Lambert. (Disponível na Minha Biblioteca na UEM).

Seção 10.2 - Listas ligadas - Algoritmos: Teoria e Prática, 3a. edição, Cormen, T. at all.

Capítulo 3 - Linked Lists - Open Data Structures.