Algoritmos e estrutura de dados

Listas

Marco A L Barbosa

Conteúdo

Listas

Listas

- Uma **Lista** é um tipo abstrato de dados usado para armazenar uma sequência de elementos $x_0, x_1, ..., x_{n-1}$
  - n é o tamanho da lista
  - x_i está na posição i (i-ésima posição)
  - x_i precede x_j para todo $i < j < n$
  - x_i sucede x_j para todo $n > i > j$
- Uma lista pode aumentar ou diminuir de tamanho
- Operações comuns em listas
  - Buscar um elemento
  - Consultar um elemento em uma posição
  - Inserir um elemento no início ou no final da lista
  - Inserir um elemento em uma posição
  - Remover um elemento
  - Remover um elemento no início ou no final da lista
  - Remover um elemento de uma posição
  - Etc

Listas encadeadas

Listas encadeadas

- Vamos supor que não existisse arranjos em Python
- Neste caso, como representar uma lista de elementos?

Listas encadeadas

- Vamos supor que não existisse arranjos em Python
- Neste caso, como representar uma lista de elementos?

- Criando um encadeamento de instâncias de estruturas

Listas encadeadas

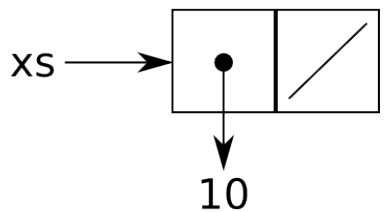
- Considere a seguinte estrutura

```
class No(Struct):
    '''
    Representa um Nó em uma lista encadeada,
        valor : Qualquer - é o valor armazenado no nó
        prox : No - é uma referência para o próximo nó
    '''
    def __init__(self, valor, prox):
        self.init(valor, prox)
```

- Vamos criar um encadeamento de instâncias da classe `No`
- Chamamos esta estrutura de **lista encadeada**

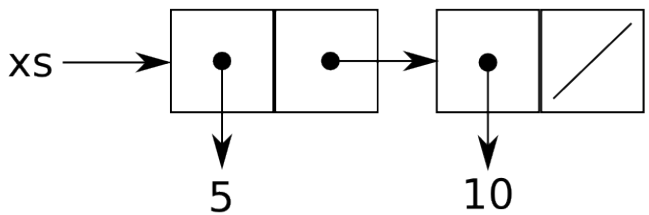
Listas encadeadas

```
>>> xs = No(10, None)
>>> xs
No(10, None)
```



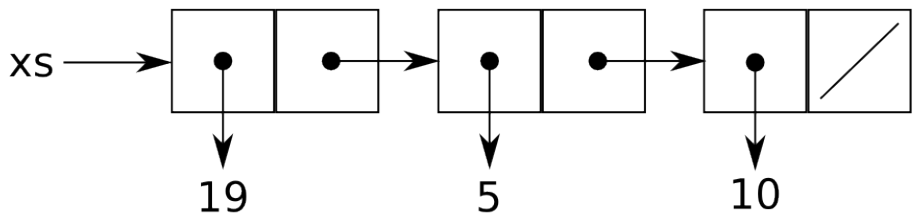
Listas encadeadas

```
>>> xs = No(5, xs)
>>> xs
No(5, No(10, None))
```



Listas encadeadas

```
>>> xs = No(19, xs)
>>> xs
No(19, No(5, No(10, None)))
```



Listas encadeadas

- Vamos implementar alguns operações de listas usando listas encadeadas
- Para isto, vamos definir um classe `ListaEncadeada`

```
class ListaEncadeada(Struct):
    '''
    Representa uma lista encadeada.
        _primeiro : No - é uma referência para o
                         primeiro nó da lista
        _ultimo : No - é uma referência para o
                       último nó da lista
    '''
    def __init__(self):
        self.init()
        self._primeiro = None
        self._ultimo = None
```

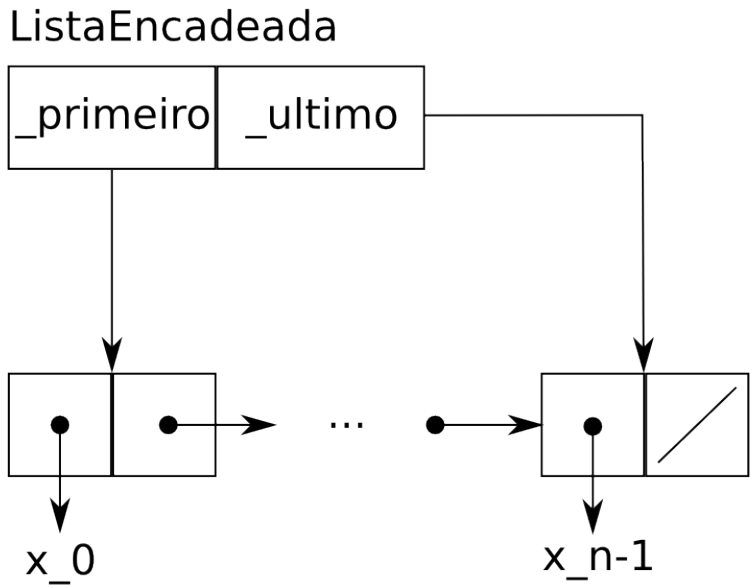
- O campo `_ultimo` será útil para implementar inserção no final

Listas encadeadas

- Lembre-se que campos com o nome iniciados com _ são campos internos. Os campos internos não são especificados na criação da instância e não podem ser acessados pelos clientes da classe. No caso da classe `ListaEncadeada` isto significa que o cliente da classe não pode alterar os campos `_primeiro` e `_ultimo` diretamente, esses campos são alterados através dos métodos (a seguir) definidos na classe

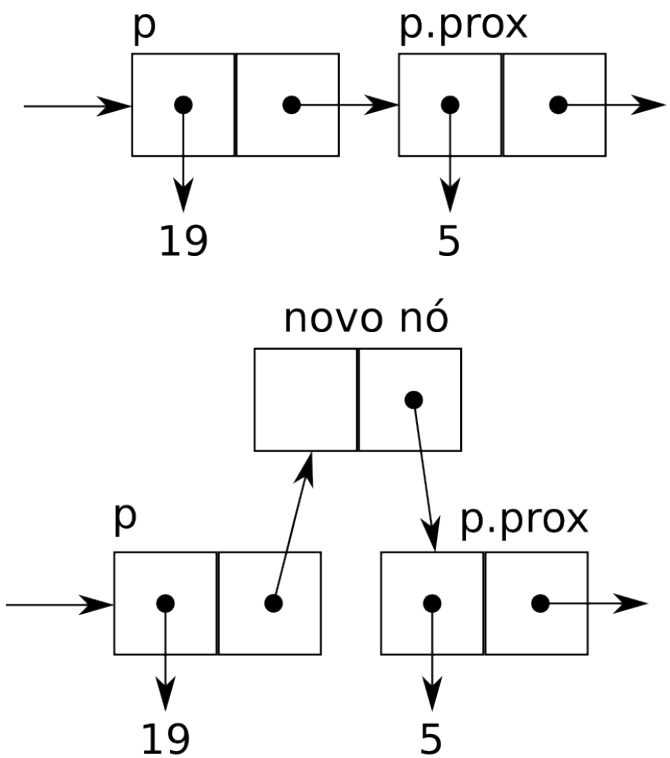
Listas encadeadas

- Representação



Listas encadeadas / Inserção

- Ideia geral da inserção



- Tratamento especial para inserção no início e fim

Listas encadeadas / Inserção no início

- Inserção no início
  - Criar um nó com o novo elemento na frete do primeiro
  - Atualizar o primeiro para o novo nó
  - Se o último for igual a `None`, o último passa referenciar o primeiro
  - Análise do tempo de execução
    - A quantidade de operações não depende da quantidade de nós na lista
    - Portanto, o tempo de execução é $O(1)$

Listas encadeadas / Inserção no início

```
    def insere_inicio(lista, v):
        '''
        ListaEncadeada, Qualquer -> None
        Insere o valor v no início da lista.
        Exemplos:
        >>> xs = ListaEncadeada()
        >>> insere_inicio(xs, 3)
        >>> xs._primeiro
        No(3, None)
        >>> xs._ultimo
        No(3, None)
        >>> insere_inicio(xs, 1)
        >>> xs._primeiro
        No(1, No(3, None))
        >>> xs._ultimo
        No(3, None)
        >>> insere_inicio(xs, 27)
        >>> xs._primeiro
        No(27, No(1, No(3, None)))
        >>> xs._ultimo
        No(3, None)
        '''
```

Listas encadeadas / Inserção no início

```
def insere_inicio(lista, v):
    '''
    ...
    '''
    lista._primeiro = No(v, lista._primeiro)
    if lista._ultimo == None:
        lista._ultimo = lista._primeiro
```

Listas encadeadas / Definição de métodos

- Vamos transformar está função em um método
- Um método é uma função definida dentro de uma classe

Listas encadeadas / Definição de métodos

- Vamos transformar está função em um método
- Um método é uma função definida dentro de uma classe

- Vantagens
  - Os dados da estrutura e as funções que operam nestes dados ficam na mesma unidade sintática (`class`)
  - É possível ter métodos com o mesmo nome em classes diferentes
  - Polimorfismo

Listas encadeadas / Definição de métodos

- Como fazer a mudança?
  - Colocar a função dentro da classe
  - Modificar o nome do primeiro parâmetro para `self` (mudar todas as ocorrências do nome dentro da função)
  - Alterar as chamadas de funções do tipo `funcao(x, ...)` para chamadas de métodos `x.funcao(...)`

Listas encadeadas / Definição de métodos

- Como fazer a mudança?
  - Colocar a função dentro da classe
  - Modificar o nome do primeiro parâmetro para `self` (mudar todas as ocorrências do nome dentro da função)
  - Alterar as chamadas de funções do tipo `funcao(x, ...)` para chamadas de métodos `x.funcao(...)`

- Como escolher entre função ou método?
  - Quando o primeiro argumento for do tipo da classe, definir como método, caso contrário, como uma função
  - Requer experiência

Observe com atenção as diferenças nos 2 próximos slides

Listas encadeadas / Definição de métodos

Definição de `insere_inicio` como função

```
class ListaEncadeada(Struct):
    # descrição omitida
    def __init__(self):
        # corpo omitido

def insere_inicio(lista, v):
    '''
    ...
    >>> xs = ListaEncadeada()
    >>> insere_inicio(xs, 3)
    >>> xs._primeiro
    No(3, None)
    >>> xs._ultimo
    No(3, None)
    # restante dos testes omitidos
    '''
    lista._primeiro = No(v, lista._primeiro)
    if lista._ultimo == None:
        lista._ultimo = lista._primeiro
```

Listas encadeadas / Definição de métodos

Definição de `insere_inicio` como método

```
class ListaEncadeada(Struct):
    # descrição omitida
    def __init__(self):
        # corpo omitido

    def insere_inicio(self, v):
        '''
        ...
        >>> xs = ListaEncadeada()
        >>> xs.insere_inicio(3)
        >>> xs._primeiro
        No(3, None)
        >>> xs._ultimo
        No(3, None)
        # restante dos testes omitidos
        '''
        self._primeiro = No(v, self._primeiro)
        if self._ultimo == None:
            self._ultimo = self._primeiro
```

Listas encadeadas / Definição de métodos

De agora em diante vamos definir funções como métodos quando isto for adequado

Listas encadeadas / Inserção no fim

- Inserção no fim
  - Criar um nó com o novo elemento
  - Se a lista está vazia, atualizar o primeiro e último para o novo nó
  - Caso contrário colocar o novo nó após o último e atualizar o último
  - Análise do tempo de execução
    - A quantidade de operações não depende da quantidade de nós na lista
    - Portanto, o tempo de execução é $O(1)$

Listas encadeadas / Inserção no fim

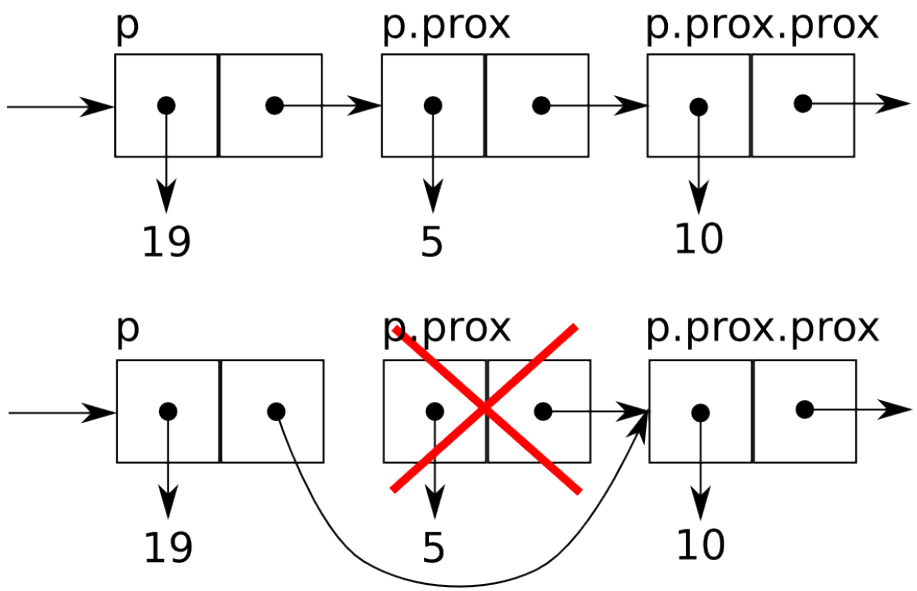
```
    def insere_fim(self, v):
        '''
        ListaEncadeada, Qualquer -> None
        Inser v no final da lista
        Exemplos:
        >>> xs = ListaEncadeada()
        >>> xs.insere_fim(3)
        >>> xs._primeiro
        No(3, None)
        >>> xs._ultimo
        No(3, None)
        >>> xs.insere_fim(10)
        >>> xs._primeiro
        No(3, No(10, None))
        >>> xs._ultimo
        No(10, None)
        >>> xs.insere_fim(20)
        >>> xs._primeiro
        No(3, No(10, No(20, None)))
        >>> xs._ultimo
        No(20, None)
        '''
```

Listas encadeadas / Inserção no fim

```
def insere_fim(self, v):
    '''
    ...
    '''
    p = No(v, None)
    if self._primeiro == None:
        self._primeiro = p
        self._ultimo = p
    else:
        self._ultimo.prox = p
        self._ultimo = p
```

Listas encadeadas / Remoção

- Ideia geral da remoção



- Tratamento especial para remoção no início e fim

Listas encadeadas / Remoção no início

- Remoção no início
  - Alterar o primeiro para referenciar o próximo do primeiro
  - Se o novo primeiro é `None`, isto é, a lista ficou vazia, atualizar o último para `None`
  - Análise do tempo de execução
    - A quantidade de operações não depende da quantidade de nós na lista
    - Portanto, o tempo de execução é $O(1)$

Listas encadeadas / Remoção no início

```
    # Veja os testes completos no arquivo listas.py
    def remove_inicio(self):
        '''
        ListaEncadeada -> Qualquer
        Remove o primeiro elemento da lista e devolve o valor removido.
        Lança uma exceção se a lista estiver vazia.
        >>> xs = ListaEncadeada()
        >>> xs.insere_inicio(3)
        >>> xs.insere_inicio(5)
        >>> xs.remove_inicio()
        5
        >>> xs._primeiro
        No(3, None)
        >>> xs._ultimo
        No(3, None)
        >>> xs.remove_inicio()
        3
        >>> xs._primeiro == None
        True
        >>> xs._ultimo == None
        True
        '''
```

Listas encadeadas / Remoção no início

```
# Veja os testes completos no arquivo listas.py
def remove_inicio(self):
    '''
    ...
    '''
    if self._primeiro == None:
        raise Exception('lista vazia')
    v = self._primeiro.valor
    self._primeiro = self._primeiro.prox
    if self._primeiro == None:
        self._ultimo = None
    return v
```

Listas encadeadas / Remoção no fim

- Remoção no fim
  - Se a lista tem apenas um nó, remover do início
  - Senão procurar o penúltimo nó e remover o próximo (que é o último)
  - Atualizar o último
  - Análise do tempo de execução
    - Para encontrar o penúltimo é necessário visitar $n - 1$ nós
    - Portanto, o tempo de execução é $O(n)$

Listas encadeadas / Remoção no fim

```
# Veja os testes no arquivo listas.py
def remove_fim(self):
    '''
    ListaEncadeada -> None
    Remove o último elemento da lista e devolve o valor removido.
    Lança uma exceção se a lista estiver vazia.
    '''
    if self._primeiro == None:
        raise Exception('lista vazia')
    if self._primeiro == self._ultimo:
        return self.remove_inicio()
    # procura o penúltimo nó
    p = self._primeiro
    while p.prox != self._ultimo:
        p = p.prox
    return self._remove_prox(p)
```

Listas encadeadas / Remoção no fim

```
def _remove_prox(self, p):
    '''
    ListaEncadeada, No -> Qualquer
    Remove o próximo a partir de p e devolve o valor armazenado no
    '''
    v = p.prox.valor
    p.prox = p.prox.prox
    if p.prox == None:
        self._ultimo = p
    return v
```

Listas encadeadas / Consulta e alteração por posição

- Consulta e alteração por posição
  - Iniciar uma referência `p` para o primeiro elemento
  - Avançar `p` para o próximo até que chegue na posição especificada ou a lista acabe
  - Se chegou na posição especificada, devolver ou alterar o valor do nó, caso contrário gerar um erro pois a posição está fora da lista
  - Análise do tempo de execução
    - No pior caso `p` avançara até acabar a lista, ou seja, vai passar por todos os nós
    - Portanto, o tempo de execução é $O(n)$

Listas encadeadas / Consulta e alteração por posição

- Exemplo inicial

```
>>> xs = ListaEncadeada()
>>> xs.insere_fim(4); xs.insere_fim(2); xs.insere_fim(7)
>>> xs.consulta(1)
2
>>> xs.consulta(2)
7
>>> xs.altera(1, 10)
>>> xs._primeiro
No(4, No(10, No(7, None)))
```

- Compare com o funcionamento da lista (pré-definida) do Python

```
>>> xs = [4, 2, 7]
>>> xs[1]
2
>>> xs[2]
7
>>> xs[1] = 10
>>> xs
[4, 10, 7]
```

Listas encadeadas / Consulta e alteração por posição

- O que é mais conveniente
  - `xs[1]` ou `xs.consulta(1)`?
  - `xs[1] = 10` ou `xs.altera(1, 10)`?
- A forma usada nas listas pré-definidas é mais conveniente
- Como fazer para tornar a consulta e alteração por posição em listas encadeada tão conveniente quanto das listas pré-definidas?
  - Alterar o nome do método `consulta` para `__getitem__`
  - Alterar o nome do método `altera` para `__setitem__`
  - Estas alterações permite escrever `xs[1]` ao invés de `xs.consulta(1)` e `xs[1] = 10` ao invés de `xs.altera(1, 10)`

Listas encadeadas / Consulta e alteração por posição

- Antes

```
>>> xs = ListaEncadeada()
>>> xs.insere_fim(4); xs.insere_fim(2); xs.insere_fim(7)
>>> xs.consulta(1)
2
>>> xs.consulta(2)
7
>>> xs.altera(1, 10)
>>> xs._primeiro
No(4, No(10, No(7, None)))
```

- Depois

```
>>> xs = ListaEncadeada()
>>> xs.insere_fim(4); xs.insere_fim(2); xs.insere_fim(7)
>>> xs[1]       # equivalente a x.__getitem__(1)
2
>>> xs[2]
7
>>> xs[1] = 10 # equivalente a x.__setitem__(1, 10)
>>> xs._primeiro
No(4, No(10, No(7, None)))
```

Listas encadeadas / Consulta e alteração posição

```
# Veja o código completo no arquivo listas.py

def _consulta_pos(self, pos):
    '''
    ListaEncadeada, Natural -> Qualquer
    Devolve o No na posição pos.
    Lança uma exceção se pos >= len(lista) ou pos < 0.
    '''
    p = self._primeiro
    i = pos
    while p != None and i > 0:
        p = p.prox
        i = i - 1
    if p != None and i == 0:
        return p
    else:
        raise Exception('índice fora do intervalo')
```

Listas encadeadas / Consulta e alteração posição

```
# Veja o código completo no arquivo listas.py

def __getitem__(self, pos):
    p = self._consulta_pos(pos)
    return p.valor

def __setitem__(self, pos, v):
    p = self._consulta_pos(pos)
    p.valor = v
```

Listas encadeadas / Inserção em uma posição

- Insere em uma posição
  - Se a posição for 0, inserir no início
  - Senão, procurar o nó na posição anterior e insirir o novo valor após este nó
  - Análise do tempo de execução
    - Inserir no início é constante
    - No no pior caso, todos os nós serão visitados se o índice estiver fora da faixa
    - Portanto, o tempo de execução é $O(n)$

Listas encadeadas / Inserção em uma posição

```
# Veja os exemplos no arquivo listas.py
def insere(self, pos, v):
    '''
    ListaEncadeada, Natural, Qualquer -> None
    Insere v na posição pos.
    '''
    if pos == 0:
        self.insere_inicio(v)
    else:
        # procura o nó na posição anterior a inserção
        p = self._consulta_pos(pos - 1)
        # insere após o nó encontrado
        self._insere_prox(p, v)
```

Listas encadeadas / Inserção em uma posição

```
# Veja os exemplos no arquivo listas.py
def _insere_prox(self, p, v):
    '''
    ListaEncadeada, No, Qualquer -> None
    Insere o v após o nó p.
    Exemplos: veja o método insere
    '''
    p.prox = No(v, p.prox)
    if p.prox.prox == None:
        self._ultimo = p.prox
```

Listas encadeadas / Remoção em uma posição

- Remoção em uma posição
  - Se a posição for 0, remover do início
  - Senão, procurar o nó na posição anterior e remover o próximo nó
  - Análise do tempo de execução
    - Remover no início é constante
    - No pior caso, todos os nós serão visitados se o índice estiver fora da faixa
    - Portanto, o tempo de execução é $O(n)$

Listas encadeadas / Remoção em uma posição

```
def remove_pos(self, pos):
    '''
    ListaEncadeada, Natural -> Qualquer
    >>> xs = ListaEncadeada()
    >>> xs.insere_fim(5)
    >>> xs.insere_fim(2)
    >>> xs.insere_fim(4)
    >>> xs.insere_fim(8)
    >>> xs.remove_pos(2)
    4
    >>> xs.remove_pos(0)
    5
    >>> xs.remove_pos(1)
    8
    '''
    if pos == 0:
        return self.remove_inicio()
    p = self._consulta_pos(pos - 1)
    return self._remove_prox(p)
```

Listas encadeadas

- Veja no arquivo `listas.py` a definição de outras operações
  - Verificar se um elemento está na lista
  - Remover um elemento
  - Contar a quantidade de nós
  - Criar uma representação textual

Listas encadeadas

- Melhorias
  - Manter também uma referência para o nó anterior
    - Permite remover qualquer nó em $O(1)$ (mais o tempo para encontrar o nó)
    - Este tipo de lista é chamada de **lista duplamente encadeada**
  - Criar um nó especial chamada sentinela que não faz parte da lista mas é o primeiro e último no encadeamento
    - Permite de as operações de inserção e remoção sejam escritas de forma genérica, sem distinguir se a operação é no início ou fim da lista

Arranjos dinâmicos

Arranjos dinâmicos

- Os arranjos nas linguagens C e Pascal são estáticos. Eles não podem mudar de tamanho
- Os arranjos em Python são dinâmicos, eles podem mudar de tamanho
- Um arranjo dinâmico pode suportar as mesmas operações de uma lista, em outras palavras, um arranjo dinâmico pode implementar o tipo abstrato Lista
- Como implementar um arranjo dinâmico em termos de um arranjo estático?

Arranjos dinâmicos / Inserção

- Quando um elemento é inserido em uma posição i, todos os elementos x_{i+1}, onde $i + 1 < n$, devem ser "empurrados" para a direita
  - Se não houver espaço, um novo arranjo com tamanho maior deve ser criado
  - Os valores do arranjo antigo são copiados para o novo arranjo
  - O arranjo antigo é descartado

Arranjos dinâmicos / Remoção

- Quando um elemento é removido de uma posição i, todos os elementos x_{i+1}, onde $i + 1 < n$, devem ser "empurrados" para a esquerda

Comparações

Comparações

	Lista Encadeada	Arranjo dinâmico
Consulta posição	$O(n)$	$O(1)$
Inserir no início	$O(1)$	$O(n)$
Inserir no fim	$O(1)$	$O(1)$
Remover do início	$O(1)$	$O(n)$
Remover do fim	$O(n)$	$O(1)$
Inserir próximo	Tempo de busca $+$ O(1)	$O(n)$

Filas

Filas

- Uma fila é como uma lista, mas as inserções e remoções deve seguir uma política
  - FIFO - First in, First out
  - O primeiro a entrar é o primeiro a sair
  - A operação de inserção também é conhecida como enfileira
  - A operação de remoção também é conhecida como desenfileira

Filas

- Se existisse uma classe `Fila`, como gostaríamos que ela funcionasse?

```
>>> f = Fila()
>>> f.insere(3)
>>> f.insere(5)
>>> f.insere(8)
>>> f.remove()
3
>>> f.insere(10)
>>> f.insere(20)
>>> f.remove()
5
>>> f.remove()
8
>>> f.remove()
10
>>> f.remove()
20
```

- Veja o arquivo `listas.py`

Pilhas

Pilhas

- Uma pilha é como uma lista, mas as inserções e remoções deve seguir uma política
  - LIFO - Last in, First out
  - O último a entrar é o primeiro a sair
  - A operação de inserção também é conhecida como empilha
  - A operação de remoção também é conhecida como desempilha

Pilhas

- Se existisse uma classe `Pilha`, como gostaríamos que ela funcionasse?

```
>>> p = Pilha()
>>> p.insere(3)
>>> p.insere(5)
>>> p.insere(8)
>>> p.remove()
8
>>> p.insere(10)
>>> p.insere(20)
>>> p.remove()
20
>>> p.remove()
10
>>> p.remove()
5
>>> p.remove()
3
```

- Veja o arquivo `listas.py`

Referências

Referências

- Projeto de algoritmos / Listas encadeadas. Paulo Feofiloff.
- Algoritmos: Teoria e prática. 3ª edição. Cormen, Thomas H at all. Capítulo 10.
- Algoritmos: Teoria e prática. 2ª edição. Cormen, Thomas H at all. Capítulo 10.