Estruturas de dados lineares

Alocação contígua

Estruturas de Dados
Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Uma **estrutura de dados** é uma forma de organizar e armazenar dados para facilitar a sua manipulação (operações).

Qual é a relação entre TADs e estruturas de dados?

Usamos estruturas de dados para implementar TADs.

Em uma **estrutura de dados linear**, os elementos são organizados de forma sequencial, um após o outro. Cada elemento possui no máximo um predecessor e um sucessor.

A estrutura de dados linear mais comum é o **arranjo**.

As duas principais características dos arranjos são:

- Os elementos são armazenados de forma contígua na memória, ou seja, em posições consecutivas.
- Cada elemento do arranjo pode ser acessado diretamente em tempo constante.

Os arranjos podem ser:

- Estáticos: a quantidade de elementos não muda.
- Dinâmicos: a quantidade de elementos pode mudar.

O tipo `list` do Python é de fato um arranjo dinâmico.

Diferente de outras linguagens, o Python não oferece um tipo pré-definido para arranjos estáticos.

Por hora não vamos mais utilizar o tipo `list`, e sim o tipo `array`, que "simula" um arranjo de tamanho fixo.

O tipo arranjo está definido na biblioteca `ed`, que está disponível para download na página da disciplina.

Arranjos estáticos em Python

```
>>> from ed import array
>>> # Cria um arranjo com 5 zeros
>>> x: array[int] = array(5, 0)
>>> x
array([0, 0, 0, 0, 0])
>>> x[0] = 10
>>> x[4] = 2
>>> x
array([10, 0, 0, 0, 2])
>>> len(x)
5
>>> x[5]
Traceback (most recent call last):
...
IndexError: list index out of range
```

```
>>> # cria um arranjo a partir de uma lista
>>> x = array([5, 1, 3, 8])
>>> soma = 0
>>> for v in x:
...     soma = soma + v
>>> soma
17
>>> x.append(10)
Traceback (most recent call last):
...
AttributeError: 'array' object has no attribute 'append'
>>> x.pop()
Traceback (most recent call last):
...
AttributeError: 'array' object has no attribute 'pop'
```

A seguir veremos três TADs e como eles podem ser implementados usando arranjos.

Exemplo parênteses

Projete uma função que verifique se os parênteses em uma expressão aritmética (representada por uma string) estão corretos, isso é:

- Cada `'('` tem um `')'` correspondente;
- Um `')'` não pode aparecer antes do `'('` correspondente.

```
def parenteses_corretos(expr: str) -> bool:
    '''
    Produz True se os parênteses de *expr*
    estão corretos, False caso contrário.
    '''
    '''
    Exemplos:
    >>> parenteses_corretos('()')
    True
    >>> parenteses_corretos('(')
    False
    >>> parenteses_corretos(')')
    False
    >>> parenteses_corretos('())')
    False
    >>> parenteses_corretos('((a)*(b-c)-10)*(4-2)')
    True
    '''
```

Como implementar essa função?

Podemos utilizar a seguinte ideia:

- Analisar a string um caractere por vez.
- Manter um contador de parênteses que foram abertos mas ainda não foram fechados.
- Incrementar o contador a cada abre parênteses e decrementar a cada fecha parênteses (o contador não pode ficar negativo).
- No final, se o contador for 0 e não ficou negativo, os parênteses estão corretos.

```python
def parenteses_corretos(expr: str) -> bool:
    '''
    Produz True se os parênteses de *expr*
    estão corretos, False caso contrário.
    '''
    '''
    Exemplos:
    >>> parenteses_corretos('()')
    True
    >>> parenteses_corretos('(')
    False
    >>> parenteses_corretos(')')
    False
    >>> parenteses_corretos('())')
    False
    >>> parenteses_corretos('((a)*(b-c)-10)*(4-2)')
    True
    '''
```

```python
def parenteses_corretos(expr: str) -> bool:
    abertos = 0
    corretos = True
    i = 0
    while i < len(expr) and corretos:
        if expr[i] == '(':
            abertos = abertos + 1
        elif expr[i] == ')':
            abertos = abertos - 1
            if abertos < 0:
                corretos = False
        i = i + 1
    return abertos == 0 and corretos
```

Exemplo agrupamento

Projete uma função que verifique se os parênteses, colchetes e chaves em uma expressão aritmética (representada por uma string) estão corretos.

```
def grupos_corretos(expr: str) -> bool:
    '''
    Produz True se os parênteses,
    colchetes e chaves de *expr*
    estão corretos, False caso contrário.

    Exemplos:
    >>> parenteses_corretos('([{}])')
    True
    >>> parenteses_corretos('[](){}')
    True
    >>> parenteses_corretos('({)}')
    False
    >>> parenteses_corretos('(2*[3*{5+2]})')
    False
    >>> parenteses_corretos('([a]*{b-c}-[10])*({(4-2)/8})')
    True
    '''
```

Usar um contador (ou mais) não é suficiente. Precisamos saber não apenas quantos “grupos” foram abertos e ainda não foram fechados, mas também qual é a ordem e o tipo do grupo (parênteses, colchetes ou chaves).

```
def grupos_corretos(expr: str) -> bool:
    '''
    Produz True se os parênteses,
    colchetes e chaves de *expr*
    estão corretos, False caso contrário.

    Exemplos:
    >>> parenteses_corretos('([{}])')
    True
    >>> parenteses_corretos('[](){}')
    True
    >>> parenteses_corretos('({)}')
    False
    >>> parenteses_corretos('(2*[3*{5+2}])')
    False
    >>> parenteses_corretos('([a*{(b)-c}]-[10])')
    True
    '''
```

Ideia da implementação:

- Analisar a string um caractere por vez.
- Manter uma coleção com as ocorrências dos grupos que foram abertos mas ainda não foram fechados.
- Quando um caractere de início de grupo é encontrado ele é *adicionado* na coleção.
- Quando um caractere de fim de grupo é encontrado ele precisa fechar (*remover* da coleção) o grupo *mais recentemente aberto* que ainda não foi fechado.
- No final, se todos os grupos foram abertos e fechados corretamente, a expressão está correta.

A ideia de implementação que acabamos de ver requer o uso de uma coleção de valores que tem operações específicas.

Note que no momento não estamos interessados em *como* implementar essas operações. Nós queremos *utilizar* essas operações para resolver o problema em questão.

Então podemos definir um TAD com essas operações, resolver o problema que estamos interessados e implementar o TAD depois.

De fato, o TAD que precisamos já é conhecido e é chamado de pilha.

Pilha

Uma **pilha** (*stack* em inglês) é tipo abstrato de dados que representa uma coleção de itens que é mantida de acordo com a regra:

- O elemento mais *recentemente inserido* é o primeiro a ser removido.

Em inglês essa política é chamada de LIFO (*Last In*, *First Out*).

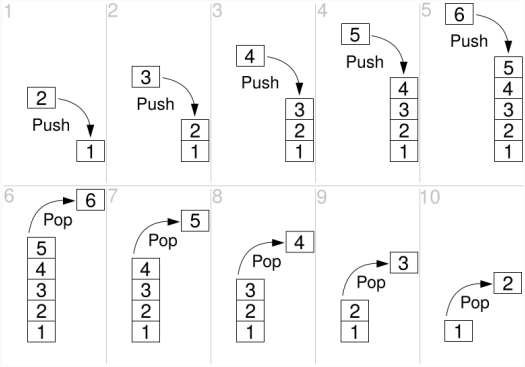
Pilha

```
class Pilha:
    '''Uma coleção de strings que segue a
    política LIFO: o elemento mais recente-
    mente inserido é o primeiro a ser
    removido.'''

    def empilha(self, item: str):
        '''Adiciona o *item* na pilha.'''

    def desempilha(self) -> str:
        '''Devolve o elemento mais
        recentemente adicionado
        da pilha.
        Requer que a pilha não esteja vazia.'''

    def vazia(self) -> bool:
        '''Devolve True se a pilha está vazia,
        False caso contrário.'''
```



O método empilha é chamado de *push* em inglês.

O método desempilha é chamado de *pop* em inglês.

Pilha

```
class Pilha:
    '''Uma coleção de strings que segue a
    política LIFO: o elemento mais recente-
    mente inserido é o primeiro a ser
    removido.'''

    def empilha(self, item: str):
        '''Adiciona o *item* na pilha.'''

    def desempilha(self) -> str:
        '''Devolve o elemento mais
        recentemente adicionado
        da pilha.
        Requer que a pilha não esteja vazia.'''

    def vazia(self) -> bool:
        '''Devolve True se a pilha está vazia,
        False caso contrário.'''
```

```
>>> p = Pilha()
>>> p.vazia()
True
>>> p.empilha('O')
>>> p.empilha('que')
>>> p.empilha('escrever?')
>>> p.vazia()
False
>>> p.desempilha()
'escrever?'
>>> p.empilha('fazer')
>>> p.empilha('agora?')
>>> while not p.vazia():
...     p.desempilha()
'agora?'
'fazer'
'que'
'O'
```

O arquivo `pilha.py`, disponível na página da disciplina, contém uma implementação para `Pilha`.

Faça o download do arquivo e use uma pilha para fazer a implementação da função que verifica se os parênteses, colchetes e chaves em uma expressão aritmética estão corretos.

Exemplo agrupamento

```python
def grupos_corretos(expr: str) -> bool:
    '''
    Produz True se os parênteses,
    colchetes e chaves de *expr*
    estão corretos, False caso contrário.

    Exemplos:
    >>> parenteses_corretos('([{}])')
    True
    >>> parenteses_corretos('[](){}')
    True
    >>> parenteses_corretos('({)}')
    False
    >>> parenteses_corretos('(2*[3*{5+2]})')
    False
    >>> parenteses_corretos('([a*{(b)-c}]-[10])')
    True
    '''
```

```python
def grupos_corretos(expr: str) -> bool:
    p = Pilha()
    corretos = True
    i = 0
    while i < len(expr) and corretos:
        if expr[i] in '([{':
            p.empilha(expr[i])
        elif expr[i] in ')]}':
            if p.vazia() or \
                    not par(p.desempilha(), expr[i]):
                corretos = False
        i = i + 1
    return p.vazia() and corretos

def par(a: str, b: str) -> bool:
    return a == '(' and b == ')' or \
            a == '[' and b == ']' or \
            a == '{' and b == '}'
```

Comparação parênteses e agrupamento

```python
def parenteses_corretos(expr: str) -> bool:
    abertos = 0
    corretos = True
    i = 0
    while i < len(expr) and corretos:
        if expr[i] == '(':
            abertos = abertos + 1
        elif expr[i] == ')':
            abertos = abertos - 1
            if abertos < 0:
                corretos = False
        i = i + 1
    return abertos == 0 and corretos
```

```python
def grupos_corretos(expr: str) -> bool:
    p = Pilha()
    corretos = True
    i = 0
    while i < len(expr) and corretos:
        if expr[i] in '([{':
            p.empilha(expr[i])
        elif expr[i] in ')]}':
            if p.vazia() or \
                    not par(p.desempilha(), expr[i]):
                corretos = False
        i = i + 1
    return p.vazia() and corretos
```

Como implementar uma pilha usando um arranjo estático?

Definimos um tamanho máximo para o arranjo e usamos um inteiro para armazenar o `topo` da pilha, isso é, o índice no arranjo do último elemento que foi inserido na pilha e que ainda não foi removido:

- Construtor: inicializa o arranjo e o `topo` com `-1`.
- Vazia: verifica se `topo == -1`.
- Empilha: incrementa `topo` e armazena o item na posição `topo`.
- Desempilha: devolve o item na posição `topo` e decrementa `topo`.

Implementação de Pilha usando arranjo estático

```
CAPACIDADE = 100
class Pilha:
    valores: array[str]
    # O índice do elemento que está no topo
    # da pilha, -1 para pilha vazia.
    topo: int

    def __init__(self):
        '''Cria uma nova pilha com capacidade
        para armazenar CAPACIDADE elementos.'''
        self.valores = array(CAPACIDADE, '')
        self.topo = -1
```

Qual é a complexidade de tempo da função `Pilha.__init__`?

$O(\text{CAPACIDADE})$, cada um dos `CAPACIDADE` elementos deve ser inicializado com `''` (o que é feito pela função `array`).

```
def empilha(self, item: str):
    assert self.topo < CAPACIDADE - 1
    self.topo = self.topo + 1
    self.valores[self.topo] = item

def desempilha(self) -> str:
    assert not self.vazia()
    item = self.valores[self.topo]
    self.topo = self.topo - 1
    return item

def vazia(self) -> bool:
    return self.topo == -1
```

Qual é a complexidade de tempo das funções `empilha`, `desempilha` e `vazia`? $O(1)$, todas as operações dessas funções têm tempo constante.

Qual é a limitação dessa implementação?

- O TAD Pilha não tem capacidade máxima;
- A implementação tem capacidade fixa, o que gera um estouro da pilha (*stack overflow*) quando o usuário tenta empilhar um elemento e a pilha está cheia.

Qual é a limitação da definição do TAD pilha?

- A possibilidade de estouro (negativo) da pilha (*stack underflow*), isso é, a tentativa de remover um elemento quando a pilha está vazia.

Veremos posteriormente como superar essas limitações.

```
class Pilha:
    def empilha(self, item: str):
        assert self.topo < CAPACIDADE - 1
        self.topo = self.topo + 1
        self.valores[self.topo] = item

    def desempilha(self) -> str:
        assert not self.vazia()
        item = self.valores[self.topo]
        self.topo = self.topo - 1
        return item

    def vazia(self) -> bool:
        return self.topo == -1
```

Podemos melhor o código?

Em geral, usamos o **assert** para verificar condições que devem ser verdadeiras durante a execução do programa e cuja a correção depende apenas do projetista do código. Adicionamos o **assert** como uma rede de segurança, mas esperamos que ele não falhe.

Para condições que devem ser verdadeiras mas a correção depende do usuário do código, usamos uma condicional para fazer a verificação e uma exceção para indicar erro.

O resultado final das duas abordagens é semelhante: a falha do programa. No entanto, o uso de exceções torna claro que o erro é esperado e permite a recuperação e a continuação de execução do programa (não veremos como fazer isso).

```
class Pilha:
    def empilha(self, item: str):
        if self.topo >= CAPACIDADE - 1:
            raise ValueError('pilha cheia')
        self.topo = self.topo + 1
        self.valores[self.topo] = item

    def desempilha(self) -> str:
        if self.vazia():
            raise ValueError('pilha vazia')
        item = self.valores[self.topo]
        self.topo = self.topo - 1
        return item

    def vazia(self) -> bool:
        return self.topo == -1
```

```
>>> p = Pilha()
>>> p.desempilha()
Traceback (most recent call last):
...
ValueError: a pilha está vazia
```

Podemos fazer mais melhorias?

Podemos escrever

```
self.topo += 1
```

Ao invés de

```
self.topo = self.topo + 1
```

Note que essa forma funciona para qualquer operador binário.

Fila

Uma **fila** (*queue* em inglês) é uma estrutura de dados que representa uma coleção de itens que é mantida de acordo com a política FIFO (*First in*, *First out*):

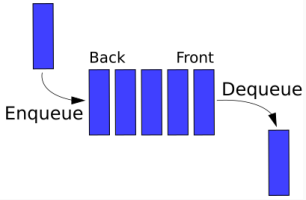
- O primeiro elemento inserido é o primeiro a ser removido.

```
class Fila:
    '''Uma coleção de strings que segue a
    política FIFO: o primeiro a ser
    inserido é o primeiro a ser removido'''

    def enfileira(self, item: str):
        '''Adiciona o *item* no final da
        fila.'''

    def desenfileira(self) -> str:
        '''Remove e devolve o primeiro
        elemento da fila.
        Requer que a fila não esteja
        vazia.'''

    def vazia(self) -> bool:
        '''Devolve True se a fila está
        vazia, False caso contrário.'''
```



O método enfileira é chamado de *enqueue* em inglês.

O método desenfileira é chamado de *dequeue* em inglês.

Fila

```python
class Fila:
    '''Uma coleção de strings que segue a
    política FIFO: o primeiro a ser
    inserido é o primeiro a ser removido.'''

    def enfileira(self, item: str):
        '''Adiciona o *item* no final da
        fila.'''

    def desenfileira(self) -> str:
        '''Remove e devolve o primeiro
        elemento da fila.
        Requer que a fila não esteja
        vazia.'''

    def vazia(self) -> bool:
        '''Devolve True se a fila está
        vazia, False caso contrário.'''
```

```
>>> f = Fila()
>>> f.vazia()
True
>>> f.enfileira('Amanda')
>>> f.enfileira('Fernando')
>>> f.enfileira('Márcia')
>>> f.vazia()
False
>>> f.desenfileira()
'Amanda'
>>> f.enfileira('Pedro')
>>> f.enfileira('Alberto')
>>> while not f.vazia():
...     f.desenfileira()
'Fernando'
'Márcia'
'Pedro'
'Alberto'
```

Implementação de fila usando arranjo estático

Como implementar uma fila usando um arranjo estático?

Usando um inteiro para indicar o `fim` da fila:

- Construtor: inicializa o arranjo e o `fim` com `-1`.
- Vazia: verifica se `fim == -1`
- Enfileira: incrementa `fim` e armazena o item na posição `fim`.
- Desenfileira: devolve o item na posição 0, move os itens ($1 \longrightarrow 0$, $2 \longrightarrow 1$, etc) e decrementa o `fim`.

```
class Fila:
    valores: array[str]
    fim: int

    def enfileira(self, item: str):
        if self.fim >= CAPACIDADE - 1:
            raise ValueError('fila cheia')
        self.fim += 1
        self.valores[self.fim] = item

    def desenfileira(self) -> str:
        if self.vazia():
            raise ValueError('fila vazia')
        item = self.valores[0]
        for i in range(1, self.fim + 1):
            self.valores[i - 1] = self.valores[i]
        self.fim -= 1
        return item
```

Implementação de fila usando arranjo estático

```
class Fila:
    valores: array[str]
    fim: int

    def enfileira(self, item: str):
        if self.fim >= CAPACIDADE - 1:
            raise ValueError('fila cheia')
        self.fim += 1
        self.valores[self.fim] = item

    def desenfileira(self) -> str:
        if self.vazia():
            raise ValueError('fila vazia')
        item = self.valores[0]
        for i in range(1, self.fim + 1):
            self.valores[i - 1] = self.valores[i]
        self.fim -= 1
        return item
```

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

Qual é a complexidade de tempo do método `desenfileira`? $O(n)$, onde n é a quantidade de elementos da fila. Os elementos das posições $1, 2, 3, \ldots, n - 1$ são movidos para as posições $0, 1, 2, \ldots, n - 2$.

Podemos fazer melhor? Sim!

Implementação de fila usando arranjo estático

Podemos usar inteiros para indicar o `inicio` e o `fim` da fila da seguinte maneira:

- Construtor: inicializa o arranjo, `inicio` com 0 e `fim` com -1.
- Vazia: verifica se `fim < inicio`
- Enfileira: incrementa `fim` e armazena o item na posição `fim`.
- Desenfileira: devolve o item na posição `inicio` e incrementa `inicio`.

```
class Fila:
    valores: array[str]
    # Indíce do último elemento da fila
    fim: int
    # Indíce do primeiro elemento da fila
    inicio: int

    def enfileira(self, item: str):
        if self.fim >= CAPACIDADE - 1:
            raise ValueError('fila cheia')
        self.fim += 1
        self.valores[self.fim] = item

    def desenfileira(self) -> str:
        if self.vazia():
            raise ValueError('fila vazia')
        item = self.valores[self.inicio]
        self.inicio += 1
        return item
```

Implementação de fila usando arranjo estático

```
class Fila:
    valores: array[str]
    # Indíce do último elemento da fila
    fim: int
    # Indíce do primeiro elemento da fila
    inicio: int

    def enfileira(self, item: str):
        if self.fim >= CAPACIDADE - 1:
            raise ValueError('fila cheia')
        self.fim += 1
        self.valores[self.fim] = item

    def desenfileira(self) -> str:
        if self.vazia():
            raise ValueError('fila vazia')
        item = self.valores[self.inicio]
        self.inicio += 1
        return item
```

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

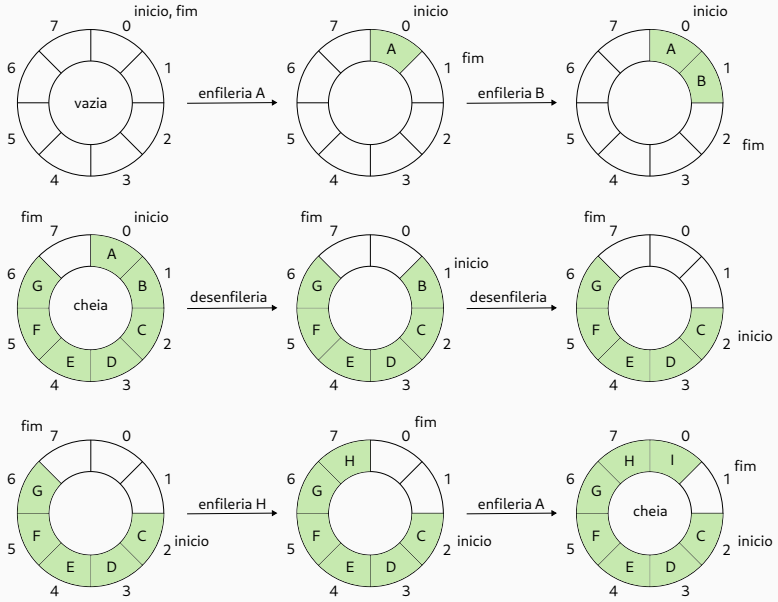
Qual é a complexidade de tempo do método `desenfileira`? $O(1)$.

Existe alguma limitação nessa implementação? Sim, a fila pode estar vazia e não ser possível enfileirar um novo item!

Podemos fazer melhor? Sim!

Usamos índices `inicio` e `fim` que avançam de forma “circular”, isto é, são incrementados até chegarem no final do arranjo e depois voltam para 0. O `fim` representa o índice onde o próximo elemento será inserido.

Para uma fila com capacidade C alocamos um arranjo de tamanho $C + 1$. Isto permite diferenciar entre fila vazia (`inicio == fim`) e fila cheia (o próximo valor de `fim` é igual ao `inicio`).


inicio, fim
vazia
enfileria A
inicio
A
fim
enfileria B
inicio
A
B
fim
fim
inicio
cheia
desenfileria
fim
inicio
desenfileria
fim
inicio
fim
inicio
enfileria H
fim
H
inicio
enfileria A
fim
H
I
cheia
inicio

Implementação de fila circular

Descrição da implementação dos métodos

- Construtor: inicializa o arranjo, `inicio` e `fim` com `0`.
- Vazia: verifica se `inicio == fim`
- Enfileira: armazena o item na posição `fim` e avança `fim`.
- Desenfileira: devolve o item na posição `inicio` e avança `inicio`.

```
class Fila:
    def enfileira(self, item: str):
        if self.cheia():
            raise ValueError('fila cheia')
        self.valores[self.fim] = item
        if self.fim == CAPACIDADE:
            self.fim = 0
        else:
            self.fim += 1

    def desenfileira(self) -> str:
        if self.vazia():
            raise ValueError('fila vazia')
        item = self.valores[self.inicio]
        if self.inicio == CAPACIDADE:
            self.inicio = 0
        else:
            self.inicio += 1
        return item
```

Implementação de fila circular

```
class Fila:
    def enfileira(self, item: str):
        if self.cheia():
            raise ValueError('fila cheia')
        self.valores[self.fim] = item
        if self.fim == CAPACIDADE:
            self.fim = 0
        else:
            self.fim += 1

    def desenfileira(self) -> str:
        if self.vazia():
            raise ValueError('fila vazia')
        item = self.valores[self.inicio]
        if self.inicio == CAPACIDADE:
            self.inicio = 0
        else:
            self.inicio += 1
        return item
```

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

A implementação “circular” de fila não tem as limitações das duas implementações anteriores, e por isso é bastante utilizada na prática, mas na forma de uma fila dupla.

Fila dupla

Uma **fila dupla** (*double ended queue* - *deque* em inglês) é uma sequência linear onde os itens podem ser inseridos e removidos dos dois extremos, o que de certa forma é uma generalização de pilha e fila.

Muitas linguagens não oferecem na biblioteca padrão uma implementação separada para pilha, fila e fila dupla, mas apenas uma implementação de fila dupla. Este é o caso do Python (`collections.deque`), do Rust (`std::collections::VecDeque`), entre outras.

Como os termos início e fim podem parecer confusos para uma fila dupla, alguns autores usam os termos esquerda e direita.

Uma **lista** é um tipo abstrato de dados que representa uma sequência de itens.

Na pilha e fila a inserção e remoção dos elementos segue uma política específica, já em uma lista, os elementos podem ser inseridos e removidos sem restrições. Além disso, os elementos de uma lista podem ser consultados sem serem removidos.

Para uma lista que representa a sequência $x_0, x_1, \ldots, x_{n-1}$:

- O tamanho da lista é n.
- O elemento x_i está na posição (índice) i.
- Para $n > 0$, x_0 é o primeiro elemento e x_{n-1} é o último elemento.
- x_i precede (é o predecessor) de x_{i+1} para $i = 0, 1, \ldots, n-2$.
- x_i sucede (é sucessor) x_{i-1} para $i = 1, 2, \ldots, n-1$.

Operações com lista

As definições de operações para o TAD lista dependem da aplicação, mas as seguintes operações são comuns:

- Consulta da quantidade de itens
- Acesso e modificação de um item de uma posição (indexação)
- Inserção de um item em uma posição
- Remoção de um item em uma posição
- Remoção de um item pelo valor do item
- Localização de um item
- Geração de uma representação em string da lista

```
class Lista:
    '''Uma sequência de números.'''

    def num_itens(self) -> int:
        '''Devolve a quantidade de itens da lista.'''

    def get(self, i: int) -> int:
        '''Devolve o item que está na posição *i* da lista.
        Requer que 0 <= i < self.num_itens().'''

    def set(self, i: int, item: int):
        '''Armazena *item* na posição *i* da lista.
        Requer que 0 <= i < self.num_itens().'''

    def insere(self, i: int, item: int):
        '''Insere *item* na posição *i* da lista. Os itens que estavam inicialmente
        nas posiçõe i, i+1, ..., passam a ficar nas posições i+1, i+2, ...
        Requer que 0 <= i <= self.num_itens().'''
```

```
class Lista:
    def remove(self, i: int) -> int:
        '''Remove e devolve o item na posição *i* da lista. Os itens que estavam
        inicialmente nas posições i, i+1, ..., passam a ficar nas posições
        i-1, i, ...
        Requer que 0 <= i < self.num_itens().'''

    def remove_item(self, item: int):
        '''Remove a primeira ocorrência de *item* da lista. Se i é a posição do
        *item*, então os itens que estavam inicialmente nas posições i, i+1,
        ..., passam a ficar nas posições i-1, i, ...
        Requer que o item esteja na lista.'''

    def indice(self, item: int) -> int:
        '''Devolve a posição da primeira ocorrência de *item* na lista.
        Requer que *item* esteja presente na lista.'''

    def str(self) -> str:
        '''Gera uma representação em string da lista.'''
```

Lista - Exemplos

```
>>> lst = Lista()
>>> lst.str()
'[]'
>>> lst.insere(0, 7)
>>> lst.insere(1, 20)
>>> lst.insere(2, 5)
>>> lst.get(0)
7
>>> lst.get(2)
5
>>> lst.num_itens()
3
>>> lst.str()
'[7, 20, 5]'
>>> lst.set(0, 10)
>>> lst.str()
'[10, 20, 5]'
```

```
>>> lst.insere(1, 8)
>>> lst.str()
'[10, 8, 20, 5]'
>>> lst.remove(2)
>>> lst.str()
'[10, 8, 5]'
>>> lst.insere(lst.num_itens(), 8)
>>> lst.str()
'[10, 8, 5, 8]'
>>> lst.indice(8)
1
>>> lst.remove_item(5)
>>> lst.str()
'[10, 8, 8]'
```

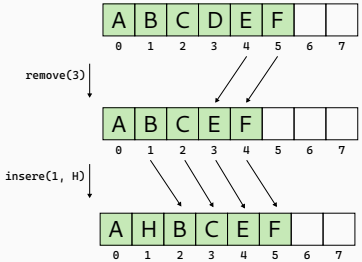
Lista - Implementação

Como implementar o TAD lista usando um arranjo estático?

Além do arranjo armazenamos a quantidade de elementos na lista.

As operações de tamanho, acesso e modificação em um posição, são diretas.

Para inserção e remoção (em uma posição) é preciso deslocar os elementos de maneira semelhante ao que fizemos para a primeira implementação de fila.



A busca por um elemento e a representação por string é feita com uma repetição pelos elementos da lista.

Por fim, a remoção de um item pode ser feita com uma busca seguido da remoção por posição.

Lista

```
class Lista:
    def insere(self, i: int, item: int):
        for j in range(self.tamanho, i, -1):
            self.valores[j] = self.valores[j - 1]
        self.valores[i] = item
        self.tamanho += 1

    def remove(self, i: int):
        for j in range(i + 1, self.tamanho):
            self.valores[j - 1] = self.valores[j]
        self.tamanho -= 1

    def str(self) -> str:
        s = '['
        if self.num_itens() != 0:
            s += str(self.valores[0])
            for i in range(1, self.num_itens()):
                s += ', ' + str(self.valores[i])
        return s + ']'
```

Veja o código completo (incluindo a verificação dos erros!) no arquivo `lista.py`.

A complexidade de tempo de `str`, `insere`, `remove`, `remove_item` e `indice` é $O(n)$. No caso especial de inserção e remoção por posição do final da lista a complexidade é $O(1)$. As demais operações são constantes.

Limitações das implementações com arranjos estáticos

Vimos como implementar os TADs Pilha, Fila, Fila Dupla e Lista usando arranjos estáticos.

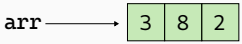
Também discutimos a principal limitação das implementações: a capacidade máxima de itens que podem ser armazenados.

Agora vamos ver como superar essa limitação!

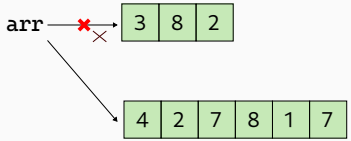
As variáveis em Python são referências para células de memórias. Quando atribuímos uma instância de um arranjo estático para uma variável, a variável passa a referenciar o bloco de células de memória do arranjo.

Apesar do bloco de memória reservado para o arranjo não poder mudar de tamanho, a variável que referencia o bloco de memória pode referenciar outro bloco de memória, com mais ou menos células.

```
>>> from ed import array
>>> arr = array([3, 8, 2])
>>> len(array)
3
```




```
>>> arr = array([4, 2, 7, 8, 1, 7])
>>> len(arr)
6
```



Como podemos utilizar esse fato para superar a limitação de capacidade máxima de itens?

Substituindo o arranjo por um com maior capacidade toda vez que a coleção ficar cheia!

- Alocamos um arranjo maior
- Copiamos os itens do arranjo cheio para o novo arranjo
- Atribuímos o novo arranjo para a variável

Veja a implementação dos métodos `Lista.insere` e `Lista.__cresce` do arquivo `lista.py`.

Essa é a forma mais comum utilizada para implementar arranjos dinâmicos. Essa é a forma que `list` do Python é implementado!

Algumas perguntas:

- Quanto maior? Muitas implementações dobram o tamanho.
- Como isso afeta a complexidade de tempo das operações? Inserir no fim, que era constante, fica com tempo *amortizado* de $O(1)$ (a maior parte das inserções no final é constante, mas algumas – quanto o arranjo está cheio – são $O(n)$).
- E se a coleção ficar com poucos itens? Podemos substituir o arranjo por um novo com menor capacidade! (veja a lista de exercícios)

Vimos 4 tipos abstratos de dados e como implementá-los usando arranjos:

- Pilha (inserção e remoção no mesmo extremo)
- Fila (inserção em um extremo e remoção do outro)
- Fila Dupla (inserção e remoção nos dois extremos)
- Lista (inserção e remoção em qualquer posição)

Se usarmos arranjos estáticos, então é preciso definir uma capacidade máxima, o que pode não ser adequado para algumas aplicações.

Se usarmos arranjos dinâmicos, então a capacidade não é limitada mas o tempo de execução de algumas operações é alterada.

Resumo

Estrutura / Operação	inserção	remoção
Pilha	$O(1)$	$O(1)$
Fila	$O(1)$	$O(1)$
Fila Dupla	$O(1)$	$O(1)$
Lista	$O(n)$ – $O(1)$ no fim	$O(n)$ – $O(1)$ no fim

Os tempos $O(1)$ são amortizados para as implementações com arranjos dinâmicos.

Veremos a seguir como fazer as implementações desses TADs de maneira que o tempo de execução dessas operações sejam $O(1)$ no pior caso.

Referências

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

Seção 10.1 - Pilhas e filas - Algoritmos: Teoria e Prática, 3a. edição, Cormen, T. at all.

Capítulo 2 - Array-Based Lists - Open Data Structures.

Dynamic Arrays na Wikipédia.