Estruturas de dados lineares - Alocação contígua - Prática

Marco A L Barbosa malbarbo.pro.br

1) Modifique a implementação de `Pilha` vista em sala adicionando um método para verificar se a pilha está cheia.

2) Modifique a implementação de `Pilha` (do exercício anterior) para que o construtor receba como parâmetro a quantidade máxima de elementos que a pilha pode armazenar. Adicione um método que devolve a capacidade da pilha e ajuste os exemplos para funcionarem com essas modificações.

3) Projete uma função que receba como parâmetro uma pilha e modifique a pilha invertendo a ordem dos seus elementos. Faça a implementação usando uma pilha auxiliar. Qual a complexidade de tempo da função?

```
>>> p = Pilha()
>>> p.empilha('um')
>>> p.empilha('carro')
>>> p.empilha('mouse')
>>> inverte_pilha(p)
>>> p.desempilha()
'um'
>>> p.desempilha()
'carro'
>>> p.desempilha()
'mouse'
```

4) Projete uma função que receba como parâmetro uma pilha e remova todos os elementos da pilha que sejam vazios. Note que ordem relativa dos elementos que permanecem na pilha não de ser altera.

```
>>> p = Pilha()
>>> p.empilha('um')
>>> p.empilha('')
>>> p.empilha('carro')
>>> p.empilha('')
>>> p.empilha('')
>>> remove_vazios(p)
>>> p.desempilha()
'carro'
>>> p.desempilha()
'um'
```

5) Projete uma função que exiba os elementos de uma pilha na ordem em que eles foram adicionados. Use uma pilha auxiliar para fazer a implementação. Note que a pilha deve permanecer como ela foi passada para a função.

```
>>> p = Pilha()
>>> p.empilha('um')
>>> p.empilha('carro')
>>> p.empilha('mouse')
>>> exibe_pilha(p)
um
carro
mouse
```

```
>>> p.desempilha()
'mouse'
>>> p.desempilha()
'carro'
>>> p.desempilha()
'um'
```

6) Projete uma função que receba como parâmetro duas pilhas e troque os elementos de uma pilha com os elementos da outra pilha.

7) (Desafio) Faça uma implementação alternativa do TAD Pilha que use uma única string para armazenar todos os elementos da pilha.

8) Modifique a implementação de `Fila` do aquivo `fila_arranjo_fim.py` adicionando um método para verificar se a fila está cheia. Use o novo método para simplificar a implementação de `enfileira`.

9) Modifique a implementação de `Fila` do aquivo `fila_arranjo_inicio_fim.py` adicionando um método para verificar se a fila está cheia. Use o novo método para simplificar a implementação de `enfileira`.

10) Modifique a implementação de `Fila` dos arquivos `fila_arranjo_fim.py`, `fila_arranjo_inicio_fim.py` e `fila_arranjo_circular.py` para que o construtor receba como parâmetro a quantidade máxima de elementos que a fila pode armazenar. Adicione um método que devolve a capacidade da fila e ajuste os exemplos para funcionarem com essas modificações.

11) A implementação de `Fila` do arquivo `fila_arranjo_circular.py` usa a ideia de "índice circular", quando o índice chega no final no arranjo, ele volta para o início. Essa ideia é usada nos métodos `enfileira`, `desenfileira` e `cheia`. Crie um método auxiliar para calcular o próximo índice a partir de um índice qualquer e use esse método para simplificar a implementação dos métodos `enfileira`, `desenfileira` e `cheia`.

12) Implemente a função chamada `tempo_fila` do arquivo abaixo:

```python
from fila_arranjo_inicio_fim import Fila
# from fila_arranjo_fim import Fila

def tempo_fila(n: int):
    # Criar uma fila com capacidade para n elementos
    # Inserir n elementos na fila
    # Esvaziar a fila
    return

if __name__ == '__main__':
    from timeit import timeit
    for n in [1000, 2000, 4000]:
        tempo = timeit(f'tempo_fila({n})',
                       setup='from __main__ import tempo_fila',
                       number=10)
        print(n, tempo)
```

Execute o arquivo com o comando `python arquivo.py` e veja na saída os tempos de execução da função `tempo_fila` para $n = 1000, 2000, 4000$.

Troque a implementação de fila usada no arquivo comentando a linha `from fila_arranjo_inicio_fim import Fila` e descomentando a linha `from fila_arranjo_fim import Fila`.

Execute novamente o arquivo e observe os tempos de execução.

Os tempos de execução foram diferentes? Explique.

13) Implemente uma fila usando duas pilhas.

14) Implemente uma pilha usando duas filas.

15) (Desafio) Faça uma implementação alternativa do TAD Fila que use uma única string para armazenar todos os elementos da fila.

16) Defina um TAD para fila dupla com métodos para inserir e remover da esquerda e direita. Implemente o TAD usando a estratégia de arranjo circular.

17) Crie um programa (semelhante ao exercício do `tempo_fila`) que mostre a diferença do tempo de execução do método `popleft` da classe `collections.deque` e do método `pop(0)` da classe `list` (pré-definidos em Python).

18) Altere a implementação do método `lista.remove` para que `valores` nunca fique com menos que 25% da sua capacidade utilizada (exceto quanto a capacidade for menor ou igual a 10). Como isso afeta a complexidade de tempo? Dica: veja o método `__cresce` e seu uso em `insere`. Escreve um método auxiliar `__diminui`, que reduz a capacidade de `valores` pela metade.

19) Altere a seguinte função para usar uma `lista` (implementada no exercício anterior) ao invés do `list` do Python.

```python
def primos(lim: int) -> list[int]:
    '''
    Encontra todos os números primos menores que *lim*.

    Exemplos:
    >>> primos(2)
    []
    >>> primos(20)
    [2, 3, 5, 7, 11, 13, 17, 19]
    '''
    primos: list[int] = []
    n = 2
    while n < lim:
        eh_primo = True
        i = 0
        while eh_primo and i < len(primos):
            if n % primos[i] == 0:
                eh_primo = False
            i = i + 1

        if eh_primo:
            primos.append(n)

        n = n + 1
    return primos
```