Practica 2

Tipo abstracto de datos

Ejercicio 1: Definir la interfaz de un tipo abstracto de datos para representar una celda. Una celda es un TAD capaz de contener un único valor a la vez, es decir, siempre que ponemos un nuevo valor, debemos quitar el valor anterior. Usar lenguaje natural.

Ejercicio 2: Basándose en la interfaz hecha en el ejercicio anterior, imp lementar en Python una clase Celda que que respete el comportamiento descrito. Escribir código cliente de dicha interfaz.

Ejercicio 3: Definir la interfaz de un Conjunto Inmutable. Los conjuntos inmutables son típicos en matemática. Se definen con elementos preestablecidos, nunca cambian, y la unica operacion posible es preguntarnos si un elemento pertenece o no al conjunto.

Ejercicio 4: Basándose en la interfaz hecha en el ejercicio anterior, implementar en Python una clase Conjunto que respete el comportamiento descrito. *Ayuda: para construir un conjunto pueden pasarle una lista o usar la sintaxis* args y que reciba una cantidad arbitraria de argumentos.*

Ejercicio 5: Dar una implementacion en Python del TAD Matriz, que representa una matriz de números reales de algebra. La interfaz queda definida por las siguientes operaciones:

`__init__`: Recibe dos dimensiones y crea una matriz de elementos nulos de dichas dimensiones (filas x columnas).

`get`: Recibe dos posiciones y devuelve el elemento guardado en la matriz en dicha posicion. Retorna `None` si las posiciones son invalidas. `put`: Recibe dos posiciones y un valor `float` y actualiza dicha posicion en la matriz con el valor de `float`. Devuelve el valor nuevo si las posiciones son validas, o `None` si son invalidas

`__add__`: Recibe como parametro otra matriz. Si las dimensiones de ambas matrices no son compatibles, devuelve `None`. Si las dimensiones son compatibles, devuelve una nueva matriz con el resultado de la suma matricial entre ambas, o sea sumar componente a componente.

Ejercicio 6: Implemente el TAD Conjunto Mutable. Un Conjunto Mutable representa un conjunto de la matematica, como en los ejercicios 3 y 4, pero ademas, soporta el agregado de elementos, y algunas operaciones adiciones. La interfaz esta dada por las siguiente operaciones:

`__init__`: Inicializa un conjunto *vacio*

`pertenece`: Toma un elemento y determina si pertenece al conjunto

`agrega`: Toma un elemento y lo agrega al conjunto. No se permiten elementos repetidos.

`anula`: Vacia el conjunto.

`union`: Recibe otro conjunto como argumento y devuelve un nuevo Conjunto con los elementos que pertenecen a cualquiera de los dos conjuntos.

`__len__`: Determina la cantidad de elementos de un conjunto

Lista

Ejercicio 7: Implemente el TAD Lista utilizando nodos enlazados.

Ejercicio 8:

a. Modifique el ejercicio anterior para que insert e index sean métodos recursivos en Nodo y la clase ListaEnlazada simplemente llame a estos.

b. ¿Podemos hacer lo mismo con el metodo `remove`? ¿Y para `[]`? Justifique.

c. Escriba codigo cliente que demuestre que las tres implementaciones de listas (list de python, la del ejercicio 7 y la de este ejercicio) se comportan igual.

Ejercicio 9:

a. Escriba una función `purga_a` que reciba una lista de python y devuelva la misma lista, pero sin elementos duplicados

b. Escriba una funcion `purga_b` que reciba una lista enlazada y devuelva la misma lista, pero sin elementos duplicados

Pila

Ejercicio 10: Implementar el TAD Pila (Stack) utilizando listas de python

Ejercicio 11: Dado un Stack, volcar su contenido en otro Stack, verificar que al vaciar el contenido de este, los elementos salen en el mismo orden en que ingresaron al primero. Utilizar solamente los métodos del TAD pila para resolver.

Ejercicio 12: Dado un Stack de números, reordenarlos para que estén abajo los impares y arriba los pares, pero que entre números del mismo tipo preserven el orden. *Ayuda: utilizar dos Stacks auxiliares de números pares e impares respectivamente*.

Ejemplo:

```
4       4
3  =>   2
2       3
1       1
```

Ejercicio 13: Construir un Stack con números y 4 operaciones representadas por los strings de sus símbolos

- suma +

- resta -
- multiplicación *
- división /

Se les dará como entrada una secuencia de números y operaciones en notación *posfija* y usarán el Stack para evaluarla.

La notación posfija tiene las operaciones a la derecha de los operandos, cuando nosotros normalmente las tenemos al medio, llamada notación infija. La ventaja de esta notación es que los paréntesis son innecesarios! Hay múltiples formas de evaluar una expresión posfija usando Stack, iterativas y recursivas. Pensar una solución cualquiera.

```
(1 + 2) * (3 - 4) # infija
1 2 + 3 4 - * # posfija
```

Ejercicio 14: Verificar si una expresión tiene paréntesis validos, es decir, no hay paréntesis sin cerrar en la expresión.

Adicional: Extienda la implementación para verificar tambien corchetes y llaves.

Ejemplos:

```
verificar_parentesis("((a+b))") => True \
verificar_parentesis("(((3*x)") => False
```

Cola

Ejercicio 15: Basándose en el TAD de cola, implementar una clase Queue con listas de Python

Ejercicio 16: Implementar FastQueue, una cola que internamente utiliza dos stacks de la siguiente manera:

- Inserta por uno de los stacks
- Remueve por el otro stack
- Cuando queremos remover de stack vacío, primero volcamos el stack de inserción en este y seguimos normalmente

Escribir codigo cliente para verificar que tanto `FastQueue` como `Queue` se comportan efectivamente como Colas.

Ejercicio 17: Implemente una PriorityQueue utilizando como politica de prioridad que los numeros mas pequeños salen primero.

Ejemplo: Metemos 5, 2, 6, 1. Sacamos 1, 2, 5, 6

Ejercicio 18: Se tiene una cola del supermercado. Cada cliente tiene un carrito con los productos apilados, por lo que al ponerlos en la cinta salen el orden inverso en el que fueron puestos en el mismo.

Dada una cola de carritos con productos, modele el problema con las estructuras vistas hasta ahora y escriba una clase Cinta que contenga una cola de carritos y tenga un método *historial* que muestre todos los productos que la lectora vería en el orden descrito.