Cátedra: Estructuras de Datos

Carrera: Licenciatura en Sistemas

Año: 2024 – 2do Cuatrimestre

Trabajo Práctico Nro 3:

Estructuras de Datos Lineales

Facultad de Ciencias de la Administración
Universidad Nacional de Entre Ríos

Objetivos

- El Trabajo Práctico N.º: 3 pretende que el alumno:
  - Comprenda el funcionamiento de las estructuras de datos lineales modificando las implementaciones vistas en clase.
  - Ponga en práctica los conceptos vistos en teoría respetando la semántica de las estructuras de datos lineales de acceso restringido.

Condiciones de Entrega

- El trabajo práctico deberá ser:
  - Realizado en forma individual o en grupos de no más de tres alumnos.
  - Cargado en la sección del Campus Virtual correspondiente, junto con los archivos de código fuente que requieran las consignas. Cada uno de ellos en sus respectivos formatos pero comprimidos en un único archivo con formato zip, rar, tar.gz u otro. Si el trabajo práctico fue realizado en grupo, deberá especificarse en el nombre del archivo los apellidos de los integrantes del mismo.
  - Entregado el **Lunes 30 de Septiembre de 2024.**
- El práctico será evaluado:
  - Con nota numérica y conceptual (Excelente, Muy Bueno, Bueno, Regular y Desaprobado), teniendo en cuenta la exactitud y claridad de las respuestas, los aportes adicionales fuera de los objetivos de los enunciados y la presentación estética.
- Las soluciones del grupo deben ser de autoría propia. Aquellas que se detecten como idénticas entre diferentes grupos o producto de una I.A. serán clasificadas como **MAL** para todos los involucrados en esta situación que será comunicada en la devolución.
- Los ejercicios que exijan codificación se valorarán de acuerdo a su exactitud, prolijidad (identación y otras buenas prácticas).

Ejercicios de Programación

1. Programe la clase **`LinkedStackExt`** de forma que extienda la clase **`LinkedStack`** (vista en clase) y a su vez implemente los métodos definidos en la clase abstracta **`LinkedStackExtAbstract`**:

```python
from abc import ABC, abstractmethod
from typing import Any, List

class LinkedStackExtAbstract(ABC):
    """Representa un conjunto de métodos para extender la implementación original
    de LinkedStack.

    Args:
        ABC (_type_): _description_
    """
    @abstractmethod
    def multi_pop(self, num: int) -> List[Any]:
    """Realiza la cantidad de operaciones pop() indicada por num.

    Args:
        num (int): número de veces que se va a ejecutar pop().

    Raises:
        Exception: Arroja excepción si se invoca a pop() por cuando la
        estructura está vacía.

    Returns:
        List[Any]: lista formada por todos los topes que se quitaron de la
        pila.
    """
    pass

    @abstractmethod
    def replace_all(self, param1: Any, param2: Any) -> None:
        """Reemplaza todas las ocurrencias de param1 en la pila por param2.

        Args:
            param1 (Any): Valor a buscar/reemplazar.
            param2 (Any): Nuevo valor.
        """
        pass

    @abstractmethod
    def __imul__(self, other: int) -> None:
        """Repite tantas veces como lo indique other los elmentos actuales de la
        pila y los inserta al final de la misma.

        Args:
            other (int): cantidad de veces que se deben repetir los elementos
            de la pila.
        """
        pass
```

2. Programe en un módulo diferente un cliente con nombre **`linked_stack_ext_client`** donde se ponga a prueba la clase **`LinkedStackExt`** programada en el ejercicio anterior. ***(No hacer ingreso de datos por consola)***.

3. Programe la clase **LinkedListExt** de forma que extienda la clase **LinkedList** (vista en clase) y a su vez implemente los métodos definidos en la clase abstracta **LinkedListExtAbstract**:

```
from abc import ABC, abstractmethod
from typing import Any, List

class LinkedListExtAbstract(ABC):

    @abstractmethod
    def __reversed__(self) -> None:
        """Invierte el orden de los elementos en la lista utilizando un LinkedStack."""
        pass

    @abstractmethod
    def pop(self) -> None:
        """Quita el último elemento de la estructura."""
        pass

    @abstractmethod
    def add_first(self, other: Any) -> None:
        """Agrega el elemento al principio de la estructura.

        Args:
            other (Any): elemento a agregar en la lista.
        """
        pass

    @abstractmethod
    def __iadd__(self, other: List[Any]) -> None:
        """Agrega todos los elementos de other en la estructura actual.

        Args:
            other (List[Any]): se trata de una lista nativa python cuyos elementos se
            agregarán al principio de la actual.
        """
        pass
```

4. Programe en un módulo diferente un cliente con nombre **linked_list_ext_client** donde se ponga a prueba la clase **LinkedListExt** programada en el ejercicio anterior. ***(No hacer ingreso de datos por consola)***.

5. Un **Deque (Double Ended Queue) o Cola Doble** es una estructura de datos lineal que soporta inserción y eliminación por ambos extremos de la colección. Utilizando una representación por enlaces (usando la clase **ListNode** del repositorio GitHub de la asignatura) implemente la interfaz que se detalla a continuación:

```
from typing import Any
from abc import ABC, abstractmethod

class DequeAbstract(ABC):
    """ Representa una estructura de datos lineal de acceso restringido
    que soporta inserción y eliminación por ambos extremos.

    Args:
        ABC (_type_):
```

```
    """

    @abstractmethod
    def __len__(self) -> int:
        """Devuelve la cantidad actual de elementos en la estructura.

        Returns:
            int: Devuelve la cantidad de elementos, cero la si estructura está vacía.
        """
        pass

    @abstractmethod
    def __str__(self) -> str:
        """ Concatena en un único string todos los elementos almacenados
        en los nodos de la estructura.

        Returns:
            str: convierte en str todos los elementos y los concatena en un string.
        """
        pass

    @abstractmethod
    def is_empty(self) -> bool:
        """Indica si la estructura está vacía.

        Returns:
            bool: True si la estructura está vacía, False en caso contrario.
        """
        pass

    @abstractmethod
    def first(self) -> Any:
        """Devuelve el elemento ubicado en el frente de la estructura.

        Raises:
            Exception: Arroja excepción si la estructura está vacía.

        Returns:
            Any: Devuelve el elemento dato correspondiente al frente de la
            estructura.
        """
        pass

    @abstractmethod
    def last(self) -> Any:
        """ Devuelve el elemento correspondiente al nodo ubicado al final de
        la estructura.

        Raises:
            Exception: Arroja excepción si la estructura está vacía.

        Returns:
            Any: Devuelve el elemento dato correspondiente al final de la estructura.
        """
        pass

    @abstractmethod
    def add_first(self, element : Any) -> None:
        """_summary_

        Args:
            element (Any): elemento que va a ser agregado al principio de la
            estructura.
        """
        pass
```

```python
@abstractmethod
def add_last(self, element : Any) -> None:
    """Agrega un elemento al final de la estructura.

    Args:
        element (Any): elemento que va a ser agregado al final de la estructura.
    """
    pass

@abstractmethod
def delete_first(self) -> None:
    """Quita el elemento ubicado en el frente de la estructura.

    Raises:
        Exception: Arroja excepción si la estructura está vacía.
    """
    pass

@abstractmethod
def delete_last(self) -> None:
    """Quita el elemento ubicado al final de la estructura.

    Raises:
        Exception: Arroja excepción si la estructura está vacía.
    """
        pass
```

6. Ponga a prueba la clase **`Deque`** y sus métodos programados en un archivo con nombre **`deque_client`**. ***(No hacer ingreso de datos por consola)***.
7. Se requiere programar una aplicación de interactiva de consola Python que permita al usuario listar, agregar, quitar y persistir en una base de datos **SQLite** instancias de clase **`Trámite`**. En memoria los objetos de tipo **`Trámite`** deberán mantenerse ordenados por orden de ingreso en una **`LinkedQueueExt`**. A tal fin deberá programar:
   a) La clase **`Trámite`** que permita registrar los siguientes datos: **`número: int, apellido: str, nombre: str, requerimiento: str, terminada: bool`** (recordar que en **SQLite** tiene que ser modelado como un **`INT`**).
      i. Incluir **`__eq_() y __str__().`**
   b) Una clase derivada de **`LinkedQueExt`** que cuente con el método **`__iter__()`** el cual permita obtener a través de un bucle **`for`** todos los objetos almacenados en la estructura.
   c) La clase **`TrámitesAdmin`** la cual gestione las operaciones de recuperar desde la base de datos, persistir, agregar, quitar y listar. **`TrámitesAdmin`** solo permitirá quitar las tareas que se encuentren en el frente o **`front`** de la cola. Al persistir los cambios los trámites terminados deberán marcarse como terminados en la tabla de base de datos.
   d) Un módulo cliente donde se realice el ingreso de datos por consola y se pongan a prueba clases y métodos de los incisos anteriores.