***Cátedra: Estructuras de Datos***

***Carrera: Licenciatura en Sistemas***

***Año: 2024 – 2do Cuatrimestre***

***Trabajo Práctico Nro 4:***

***Árboles Binarios y Heaps***

Facultad de Ciencias de la Administración
Universidad Nacional de Entre Ríos

## *Objetivos*

- El Trabajo Práctico N.º: 4 pretende que el alumno:
  - Entienda las diferencias entre estructuras jerárquicas y sus aplicaciones prácticas.
  - Implementar Estructuras de Datos Jerárquicas: Desarrollar implementaciones de árboles binarios, heaps y colas de prioridad, aplicando conceptos teóricos a la práctica.

## *Condiciones de Entrega*

- El trabajo práctico deberá ser:
  - Realizado en forma individual o en grupos de no más de tres alumnos.
  - Publicado en un repositorio en GitHub creado por el equipo de trabajo cuya URL deberá especificarse en la Actividad del Campus Virtual o 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.
    - En caso de realizar el Trabajo Práctico en grupo, deberá indicarse el apellido y nombre de los integrantes del mismo.
  - Ser entregado el **Lunes 21 de Octubre 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 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).

- ***Consideraciones***

  El equipo de cátedra considera que la corrección de los Trabajos Prácticos presentados por los alumnos es más ágil si no se programan interfaces de consola de comandos que exigen al usuario interactuar con la aplicación. Por tanto, se solicita no programar clases cliente que hagan uso de la función **`input()`** o similares.

## *Ejercicios de Programación*

1. Programe la clase **`LinkedBinaryTreeExt`** como clase derivada de **`LinkedBinaryTree`** y de **`LinkedBinaryTreeExtAbstract`** detallada a continuación:

```
from abc import ABC, abstractmethod
from typing import Any, List, Union
from data_structures import BinaryTreeNode


class LinkedBinaryTreeExtAbstract(ABC):
    """Conjunto de métodos adicionales a LinkedBinaryTree"""

    @abstractmethod
    def ancestro_mas_inmediato(self, nodo1: BinaryTreeNode, nodo2: BinaryTreeNode) -> BinaryTreeNode:
        """Devuelve el ancetro común entre nodo1 y nodo2 de mayor profundidad.

        Args:
            nodo1 (BinaryTreeNode): debe no ser None y pertenecer al árbol.
            nodo2 (BinaryTreeNode): debe no ser None y pertencer al árbol.

        Returns:
            BinaryTreeNode: devuelve el ancestro común de nodo1 y nodo2.
        """

    @abstractmethod
    def hojas(self) -> List[Any]:
        """Devuelve los elementos de los nodos que no tienen ningún hijo.

        Returns:
            List[Any]: lista formada por los elementos de los nodos hoja.
        """
        pass

    @abstractmethod
    def nivel(self, nodo: BinaryTreeNode) -> int:
        """Busca el nodo pasado por parámetro en el árbol y si lo encuentra
         Retorna su nivel. El nivel del nodo raíz es 0 el nivel de los hijos
         de la raíz es 1 y así sucesivamente.

        Args:
            nodo (BinaryTreeNode): nodo del que se quiere conocer su nivel.

        Returns:
            List[Any]: lista formada por los elementos de los nodos hoja.
        """
        pass

    @abstractmethod
    def diametro(self) -> int:
        """Indica el diámetro o ancho máximo de un árbol.
        El ancho máximo es la cantidad máxima de nodos que hay en un mismo nivel del árbol.

        Returns:
            int: devuelve la máxima cantidad de nodos entre todos los niveles que conforman el árbol.
        """
        pass

    @abstractmethod
    def es_balanceado(self) -> bool:
        """Comprueba si el árbol está balanceado
```

```
    Un árbol está balanceado si para cada uno de sus nodos se cumple que la diferencia de altura
    entre el subárbol izquierdo y el derecho no difiere en más de una unidad.

    Returns:
        bool: True en caso que el árbol esté balanceado. False en caso contrario.
    """
    pass
```

2. Programe en un archivo con nombre **`linked_binary_tree_ext_client.py`** un cliente para **`LinkedBinaryTreeExt`** donde se pongan a prueba **TODOS** los métodos definidos en esa clase con el siguiente árbol binario ***(No hacer ingreso de datos por consola)***:

```
from data_structures import BinaryTreeNode
from linked_binary_tree_ext import LinkedBinaryTreeExt

nodo_a = BinaryTreeNode('A')
nodo_b = BinaryTreeNode('B')
nodo_c = BinaryTreeNode('C')
nodo_d = BinaryTreeNode('D')
nodo_f = BinaryTreeNode('F')
nodo_g = BinaryTreeNode('G')
nodo_h = BinaryTreeNode('H')
nodo_i = BinaryTreeNode('I')
nodo_k = BinaryTreeNode('K')
nodo_m = BinaryTreeNode('M')
nodo_n = BinaryTreeNode('N')

arbol = LinkedBinaryTreeExt()
arbol.add_root(nodo_a)

arbol.add_left_child(nodo_a, nodo_b)
arbol.add_right_child(nodo_a, nodo_f)

arbol.add_left_child(nodo_b, nodo_c)
arbol.add_right_child(nodo_b, nodo_d)

arbol.add_left_child(nodo_f, nodo_g)
arbol.add_right_child(nodo_f, nodo_k)

arbol.add_left_child(nodo_g, nodo_h)
arbol.add_right_child(nodo_g, nodo_i)

arbol.add_left_child(nodo_k, nodo_m)
arbol.add_right_child(nodo_k, nodo_n)

print(arbol)
```

3. Una **`Cola de Prioridad`** es una estructura de datos similar a una **`Cola`** donde los elementos tienen una prioridad asignada. Los elementos con menor prioridad serán procesados primero. Dentro del archivo **`sorted_priority_queue.py`** construya la clase **`SortedPriorityQueue`** que utilice cono estructura de datos subyacente una lista nativa Python. En cada operación de mutación (inserciones y eliminaciones) deberá asegurar que la lista se mantenga ordenada. Para tal fin inserte elementos de clase **`_Item`** en la lista nativa Python. **`SortedPriorityQueue`** deberá extender la siguiente clase:

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

class SortedPriorityQueueAbstract(ABC):
    """Cola de prioridad mínima ordenada utilizando representación posicional."""

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

        Returns:
            int: Cantidad de elementos en la estructura. 0 en caso que esté vacía.
        """
        pass
```

```python
    @abstractmethod
    def is_empty(self) -> bool:
        """ Indica si la estructura está vacía o no.

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

    @abstractmethod
    def add(self, k: Any, v: Any) -> None:
        """ Inserta un nuevo ítem al final de la estructura.

        Args:
            k (Any): Clave que determina la prioridad del ítem.
            v (Any): Valor del ítem.
        """
        pass

    @abstractmethod
    def min(self) -> Tuple[Any]:
        """ Devuelve una tupla conformada por la clave y valor del ítem con menor valor de
                        clave.
          Raises:

              Exception: Arroja excepción si se invoca cuando la estructura está vacía.


        Returns:
            Tuple[Any]: Tupla de dos elementos: Clave y Valor del ítem.
        """
        pass

    @abstractmethod
    def remove_min(self) -> Tuple[Any]:
        """ Quita de la estructura el ítem con menor valor de clave.
          Raises:

              Exception: Arroja excepción si se invoca cuando la estructura está vacía.

        Returns:
            Tuple[Any]: Tupla de dos elementos: Clave y Valor del ítem.
        """
        pass

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

        Returns:
            str: formado por todos los elementos de la estructura convertidos en str.
        """
        pass
```

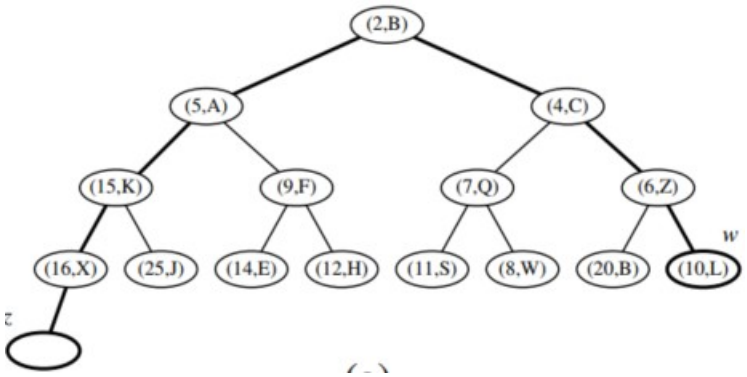
4. Programe en un archivo con nombre **`sorted_priority_queue_client.py`** un cliente para **`SortedPriorityQueue`** donde se pongan a prueba **TODOS** los métodos definidos en esa clase ***(No hacer ingreso de datos por consola)***.

5. Programe la clase **`ArrayHeapExt`** que extienda las clases **`ArrayHeap`** (vista en clases) y **`ArrayHeapExtAbstract`** que se detalla a continuación:

```python
from abc import ABC, abstractmethod
from typing import Any
from data_structures import ArrayHeap


class ArrayHeapExtAbstract(ABC):

  @abstractmethod
  def fusionar(self, otro: ArrayHeap) -> None:
    """Combina los elementos de otro heap en este dejándolos perfectamente ordenados.

    Args:
        otro (ArrayHeap): ArrayHeap pasado por parámetro.
    """
```

```
    pass

  @abstractmethod
  def vaciar(self) -> None:
    """ Una vez invocado el Heap queda sin elementos. """
    pass

  @abstractmethod
  def eliminar_por_prioridad(self, k: Any) -> None:
    """ Elimina el elemento con prioridad igual al parámetro k.
    Luego de ser invocado la estructura debe preservar la condición de orden

    Args:
        k (Any): prioridad del elemento a eliminar.
    """

  @abstractmethod
  def cambiar_prioridad(self, k_actual: Any, k_nueva: Any) -> None:
    """ Cambia la prioridad de los nodos con prioridad igual k_actual y establece como nueva prioridad k_nueva.

    Args:
      k_actual (Any): prioridad actual del elemento a modificar
      k_nueva (Any): nueva prioridad que debe ser asignada.
    """
    pass
```

6. Programe en un archivo con nombre **`array_heap_ext_client.py`** un cliente para **`ArrayHeapExt`** donde:
   a) Logre la siguiente disposición de elementos en la estructura:




   b) Pongan a prueba **TODOS** los métodos definidos en **`ArrayHeapExt.`**

## *Proyectos de Programación (opcional)*

7. Implemente un Heap mínimo utilizando representación por enlaces.