Instituto Técnico Superior Córdoba
LAB 1 - Proyecto de Programación II - Sistema de Gestión de Tareas Pendientes
Profesor: BORDONE, Matías
Estudiantes: CONTRERAS, Yamila y FIGUEROA, Ezequiel
Fecha de entrega: 10 de Agosto

INFORME DEL PROYECTO

1. Introducción

Este proyecto de sistema de gestión de tareas permite a los usuarios organizar, planificar y seguir sus tareas y responsabilidades de manera efectiva. La propuesta tiene como objetivos fundamentales:

- Organizar las tareas agregadas en una lista y clasificarlas por prioridad.
- Gestionar las tareas y marcar cuando ya se completaron o eliminarlas.
- Monitorear las tareas a través de su visualización o búsqueda por descripción, id o categoría

De esta manera, se espera que la implementación de este sistema de gestión de tareas brinde a los usuarios una herramienta efectiva para organizar, gestionar y seguir el progreso de sus tareas.

2. Descripción del Sistema

Funcionalidades implementadas:

- **Agregar tarea:** Esta funcionalidad permite al usuario agregar tareas que se organizan de manera ordenada (por prioridad) a una lista. El usuario va a introducir una descripción de la tarea (nombre), va a poder seleccionar la prioridad de la misma ingresando 3 si la tarea es de prioridad alta, 2 si la tarea es de prioridad media y 1 si la tarea es de prioridad baja; además tendrá la posibilidad de organizarlas por categorías. Una funcionalidad interesante es buscar la tarea por descripción que el sistema permite chequear que existe una tarea antes de agregarla.
- **Completar una tarea:** Una vez que el usuario complete una tarea, con esta funcionalidad, podrá marcar la opción correspondiente y marcar la tarea como completada, para ello, el sistema le pedirá que ingrese el ID correspondiente de la tarea.
- **Eliminar tareas**: El usuario podrá eliminar tareas de la lista ingresando el ID correspondiente de la tarea que desea eliminar.
- **Mostrar todas las tareas**: Esta funcionalidad permite al usuario visualizar toda su lista de tareas. Si en la lista no hay tareas, el sistema le avisará que no hay tareas.
- **Mostrar las tareas pendientes:** Esta funcionalidad permite al usuario visualizar una lista de sus tareas pendientes.
- **Mostrar las tareas de una categoría:** El usuario podrá ingresar la categoría y esta funcionalidad le permitirá observar todas las tareas correspondientes a esa categoría.

- **Mostrar tareas por descripción:** Esta funcionalidad permite a los usuarios visualizar las tareas de la lista según su descripción.
- **Buscar tareas por ID:** Con esta funcionalidad, el usuario podrá buscar tareas por su ID y visualizarlas.
- **Contar tareas:** El sistema brinda esta funcionalidad a los usuarios para tener un registro de la cantidad de tareas existentes en su lista.
- **Contar tareas por categoría:** Esta funcionalidad permite a los usuarios tener un registro de cantidad de tareas existentes en su lista organizadas por categorías.
- **Priorizar tareas según su prioridad:** Gracias a esta funcionalidad el usuario podrá tener una lista de tareas organizadas por prioridad: alta, media y baja.

Estructura de datos utilizadas:

- **Implementación de lista enlazada y nodos en el sistema:** El sistema utiliza estas estructuras de datos para ayudar a los usuarios a mantener y gestionar sus datos de manera ordenada. Al implementar listas enlazadas y nodos, el sistema puede llevar a cabo diversas operaciones y ofrecer funcionalidades específicas. Los nodos son los componentes fundamentales de una lista enlazada, cada uno conectado al siguiente. En una lista enlazada, el acceso directo solo es posible al primer nodo, conocido como la cabeza. Para alcanzar otros nodos, es necesario empezar desde la cabeza y seguir los enlaces secuenciales de un nodo a otro. Esta estructura de datos facilita el acceso a los nodos que contienen la información de las tareas del usuario y permite implementar funciones para recorrer la lista, brindando así diversas funcionalidades a los usuarios.

3. Metodología y Diseño del Código:

En este sistema de gestión de tareas, el código está desarrollado en un lenguaje de programación llamado Python. Si bien este lenguaje permite la utilización de estructuras de datos ya predefinidas como las listas, en este sistema creamos nosotros mismos la estructura de datos.
Nuestro sistema también se basa en la Programación Orientada a Objetos, en donde se crean objetos para que interactúen entre sí y de esa manera poder brindar las funcionalidades correspondientes del sistema.
Las clases principales utilizadas fueron Tarea, Nodo y ListaEnlazada. A continuación se puede observar el diseño de cada clase creada y una breve descripción de ellas:

```
class Tarea:
    def __init__(self, id, descripcion, prioridad, categoria="General"):
        self.id = id
        self.descripcion = descripcion
        self.prioridad = prioridad
        self.completada = False
        self.categoria = categoria
```

La **clase Tarea,** crea objetos tareas que representan las tareas que el usuario va a agregar a su lista a través de este sistema de gestión de tareas.

En el constructor __init__ se puede observar que se toman como parámetros el objeto en sí mismo (self), el id de la tarea agregada (que luego a través de un método se va actualizando a medida que el usuario ingrese más tareas), la descripción de la tarea (nombre), la prioridad (1: prioridad baja, 2: prioridad media y 3: prioridad alta) y la categoría de la tara que en el caso que el usuario no ingrese por defecto se asigna a una categoría "General". Otro atributo que define las características de la tarea es self.completada, es decir, que inicialmente cuando se ingresa una tarea es False porque no está completada.

```
class Nodo:
    def __init__(self, tarea):
        self.tarea = tarea
        self.siguiente = None
```

La **clase Nodo**, crea objetos nodos que funcionarán en esta lista enlazada cómo un componente que almacena una tarea y un enlace al siguiente nodo en la lista.
En el constructor __init__ se puede observar que se toman como parámetros el objeto en sí mismo (self) y el objeto tarea que se asigna al atributo self.tarea del nodo, en este atributo se almacena la información que el nodo contiene. El atributo self.siguiente se utiliza para enlazar el nodo con el siguiente nodo en la lista. Al principio, no hay ningún nodo siguiente, por lo que se establece en None.
Además de la definición del constructor __init__ que se visualiza anteriormente, en esta clase está definido el método __str__ que proporciona una forma legible de representar el nodo como una cadena, mostrando todos los detalles de la tarea almacenada en el nodo, cómo su id, descripción, prioridad, categoría y estado.

```
class ListaEnlazada:
    def __init__(self):
        self.cabeza = None
        self.id_actual = 1
        self.cantidad_tareas = 0
        self.categorias_pendientes = {}
```

La **clase ListaEnlazada,** se define cómo estructura para manejar una lista enlazada.
En el constructor __init__ se puede observar que se toman como parámetros el objeto en sí mismo. Los atributos de una lista enlazada son:

- self.cabeza: Este atributo representa la cabeza de la lista enlazada. En una lista enlazada, la cabeza es el primer nodo de la lista. Inicialmente, se establece en None, lo que significa que la lista está vacía y no tiene nodos aún.
- self.id_actual: este atributo mantiene un identificador único para las tareas o nodos en la lista. Se inicializa en 1 y puede ser utilizado para asignar ID únicos a nuevas tareas a medida que se agregan a la lista.
- self.cantidad_tareas: este atributo lleva el conteo del número de tareas o nodos en la lista. Se inicializa en 0 porque al principio la lista está vacía.
- self.categorias_pendientes: este atributo corresponde a un diccionario que utilizamos para almacenar categorías de tareas que están pendientes.

Además de la definición del constructor __init__ que se visualiza anteriormente, en esta clase están definidos los métodos que van a interactuar con los objetos de la clase Tarea y de la clase Nodo. Métodos cómo:

- esta_vacía: devuelve False si la lista está vacía o True si no lo está.
- agregar_tarea: toma una instancia de la clase tarea cómo objeto para agregar a la lista y que a su vez se vincula con una instancia de la clase nodo para guardar esos datos en él.
- buscar_tarea_descripción: toma los objetos nodo para verificar si la tarea que el usuario está por agregar existe o no en ellos.
- completar_tarea: toma los objetos nodo para recorrerlos y encontrar el id de la tarea que corresponde con el id que el usuario ingresó para marcarla como completada.
- eliminar_tarea: recorre los nodos para encontrar el id de la tarea que corresponde con el id que el usuario ingresó para eliminarla.
- mostrar_tareas: recorre los nodos para mostrar la lista de tareas agregadas.
- mostrar_tareas_pendientes: recorre los nodos para mostrar la lista de tareas pendientes.
- mostrar_tareas_descripcion: recorre los nodos para mostrar la lista de tareas según la descripción ingresada por el usuario.
- mostrar_tareas_categorias: recorre los nodos para mostrar la lista de tareas según la categoría ingresada por el usuario.
- contar_tareas_pendientes_cte: devuelve la cantidad de tareas pendientes.
- contar_tareas_completadas: devuelve la cantidad de tareas completadas.
- mostrar_estadisticas: muestra la cantidad de tareas pendientes por categoría, el total de tareas pendientes y el total de tareas completadas.
- agregar_tarea_existente: añade una nueva tarea a una lista enlazada que mantiene las tareas ordenadas por prioridad. También actualiza el conteo de tareas en cada categoría pendiente y gestiona el ID para asignar nuevos IDs únicos. La tarea se inserta en la lista de manera que siempre esté en el lugar correcto según su prioridad.

4. Implementación y Ejemplos

Implementación de funcionalidades principales, casos de uso y ejemplos:
Al utilizar este sistema de gestión de tareas, el usuario visualizará el siguiente menú:

```
Menú:
1. Agregar tarea
2. Completar tarea
3. Eliminar tarea
4. Mostrar todas las tareas
5. Mostrar tareas pendientes
6. mostrar tareas por caracteristicas
7. Guardar tareas en archivo CSV
8. Cargar tareas desde archivo CSV
9. buscar tarea
10. Mostrar estadisticas
11. Salir
Seleccione una opción:
```

A continuación comentaremos acerca de la implementación de algunas funcionalidades principales del sistema:

1. **Agregar tarea:** el usuario ingresará esta opción 1 si desea agregar una tarea. Al marcar la opción, deberá ingresar la descripción de la tarea, la categoría y la prioridad.

```
Ingrese la descripción de la tarea: dormir
Ingrese la categoria de la tarea: descanso
Ingrese la prioridad de la tarea (1 = baja, 2 = media, 3 = alta): 3
Tarea agregada con éxito.
```

Código implementado en esta funcionalidad:

Al ingresar la opción 1, se guarda en descripción la descripción que el usuario ingresa y en categoría la categoría que el usuario ingresa, cómo así también se guarda en prioridad la prioridad (1,2,3) que el usuario ingrese. Con try y except, manejamos errores por si el usuario ingresa una opción inválida.

```
if opcion == "1":
    descripcion = input("Ingrese la descripción de la tarea: ")
    categoria = input("Ingrese la categoria de la tarea: ")
    try:
        prioridad =int(input("Ingrese la prioridad de la tarea (1 = baja, 2 = media, 3 = alta): "))
        if prioridad in {1,2,3}:
            lista_tareas.agregar_tarea(descripcion, prioridad, categoria)
        else:
            print ("Prioridad invalida. Debe ser un numero del 1 al 3")
    except ValueError:
        print ("Entrada invalida. Debe ser un numero del 1 al 3")
```

Cómo se observa en el fragmento de código anterior, una vez que el usuario ingrese la prioridad de la tarea, se ejecuta el método agregar tarea que se visualiza a continuación:

```
def agregar_tarea(self, descripcion, prioridad, categoria):

    tarea = Tarea(self.id_actual, descripcion, prioridad, categoria)

    if not self.buscar_tarea_descripcion(descripcion):
        nuevo_nodo = Nodo(tarea)
        self.id_actual += 1
        self.cantidad_tareas +=1

        if self.esta_vacia() or tarea.prioridad > self.cabeza.tarea.prioridad:
            nuevo_nodo.siguiente = self.cabeza
            self.cabeza = nuevo_nodo

        else:
            actual = self.cabeza
            while actual.siguiente is not None and actual.siguiente.tarea.prioridad >= tarea.prioridad:
                actual = actual.siguiente

            nuevo_nodo.siguiente = actual.siguiente
            actual.siguiente = nuevo_nodo

        if not tarea.completada:
            if tarea.categoria in self.categorias_pendientes:
                self.categorias_pendientes[tarea.categoria] += 1
            else:
                self.categorias_pendientes[tarea.categoria] = 1

        print("Tarea agregada con éxito.")

    else:
        print("No se pudo cargar la tarea. La tarea ya existe")
```

Este método de agregar_tarea, crea una nueva tarea con los parámetros proporcionados por el usuario y la añade a la lista enlazada si no existe ya una tarea con la misma descripción. Primero, se verifica si la tarea ya está en la lista utilizando buscar_tarea_descripcion. Si no está, se crea un nuevo nodo con la tarea, se actualizan los contadores de tareas y se inserta el nodo en la posición correcta según la prioridad. Además, se actualiza el conteo de tareas pendientes por categoría. Si la tarea ya existe, muestra un mensaje de error.

2. **Completar tarea:** el usuario ingresará esta opción 2 si desea marcar una tarea como completada. Al marcar la opción, deberá ingresar el id de la tarea a marcar como completada.

```
Seleccione una opción: 2
Ingrese el ID de la tarea a completar: 1
Se completo la tarea
```

Código implementado en esta funcionalidad:

Al ingresar la opción 2, el usuario deberá ingresar el ID de una tarea para marcarla como completada. Luego se llama al método completar_tarea en el objeto lista_tareas para marcar la tarea correspondiente como completada. Con try y except, manejamos errores por si el usuario ingresa una opción inválida.

```
elif opcion == "2":
    try:
        id_tarea = int(input("Ingrese el ID de la tarea a completar: "))
        lista_tareas.completar_tarea(id_tarea)

    except ValueError:
        print ("Entrada invalida. El ID es un numero")
```

Cómo se observa en el fragmento de código anterior, una vez que el usuario ingrese el ID de la tarea a completar, se ejecuta el método completar tarea que se visualiza a continuación:

```
def completar_tarea(self, id):
    actual = self.cabeza

    while actual is not None:
        if actual.tarea.id == id:
            if not actual.tarea.completada:
                actual.tarea.completada = True
                print("Se completo la tarea")
                self.cantidad_tareas -=1
                if actual.tarea.categoria in self.categorias_pendientes:
                    self.categorias_pendientes[actual.tarea.categoria] -= 1
                    if self.categorias_pendientes[actual.tarea.categoria] == 0:
                        del self.categorias_pendientes[actual.tarea.categoria]
                return
            else:
                print("Error: La tarea ya fue completada")
                return
        actual = actual.siguiente
    print ("No hay tareas con ese numero de ID")
```

Este método de completar_tarea, busca una tarea en la lista enlazada por su ID. Si encuentra la tarea y no está completada, marca la tarea como completada, reduce el contador total de tareas y ajusta el conteo de tareas pendientes por categoría. Si la categoría no tiene más tareas pendientes, se elimina del diccionario. Si la tarea ya estaba completada o no se encuentra en la lista, se muestra un mensaje de error.

3. **Eliminar tarea:** el usuario ingresará esta opción 3 si desea eliminar una tarea. Al marcar la opción, deberá ingresar el id de la tarea a eliminar.

```
Seleccione una opción: 3
Ingrese el ID de la tarea a eliminar: 1
Tarea eliminada: ID: 1, Descripción: dormir, Prioridad: 3, Categoría: descanso, Estado: Completad
```

Código implementado en esta funcionalidad:

Al ingresar la opción 3, el usuario deberá ingresar el ID de una tarea para eliminarla. Luego se llama al método eliminar_tarea en el objeto lista_tareas para eliminar la tarea. Con try y except, manejamos errores por si el usuario ingresa una opción inválida.

```
elif opcion == "3":
    try:
        id_tarea = int(input("Ingrese el ID de la tarea a eliminar: "))
        lista_tareas.eliminar_tarea(id_tarea)

    except ValueError:
        print("Entrada invalida. El ID es un numero")
```

Cómo se observa en el fragmento de código anterior, una vez que el usuario ingrese el ID de la tarea a eliminar, se ejecuta el método eliminar tarea que se visualiza a continuación:

```
def eliminar_tarea(self, id):
    actual = self.cabeza
    previo = None
    while actual is not None:
        if actual.tarea.id == id:
            if previo is None:
                self.cabeza = actual.siguiente
            else:
                previo.siguiente = actual.siguiente
            print(f"Tarea eliminada: {actual}")
            if not actual.tarea.completada:
                if actual.tarea.categoria in self.categorias_pendientes:
                    self.categorias_pendientes[actual.tarea.categoria] -= 1
                    if self.categorias_pendientes[actual.tarea.categoria] == 0:
                        del self.categorias_pendientes[actual.tarea.categoria]
            self.cantidad_tareas = self.cantidad_tareas -1
            return
        previo = actual
        actual = actual.siguiente
    print(f"Tarea con ID {id} no encontrada.")
```

Este método de eliminar_tarea busca una tarea en la lista enlazada por su ID. Si encuentra la tarea, la elimina de la lista ajustando los enlaces entre nodos, y actualiza el contador de tareas y el conteo de tareas pendientes por categoría. Si la tarea estaba pendiente y su

categoría ya no tiene tareas pendientes, elimina la categoría del diccionario. Si no encuentra la tarea o el ID no coincide con ningún nodo, muestra un mensaje indicando que la tarea no fue encontrada.

4. **Mostrar todas las tareas:** el usuario ingresará esta opción 4 si desea visualizar todas las tareas de su lista.

```
Seleccione una opción: 4
ID: 3, Descripción: comer, Prioridad: 3, Categoría: alimentacion, Estado: Pendiente
ID: 4, Descripción: bañarse, Prioridad: 3, Categoría: higiene, Estado: Pendiente
ID: 2, Descripción: dormir, Prioridad: 2, Categoría: descanso, Estado: Pendiente
```

Código implementado en esta funcionalidad:

Al ingresar la opción 4, el usuario podrá observar su lista de tareas.

```
elif opcion == "4":
    lista_tareas.mostrar_tareas()
```

Cómo se observa en el fragmento anterior, al ingresar esta opción llama al método mostrar_tareas del objeto lista_tareas para mostrar todas las tareas en la lista. Esto permite al usuario ver una representación de las tareas almacenadas en el sistema.

```
def mostrar_tareas(self):
    actual = self.cabeza
    if actual is None:
        print("No hay tareas agregadas")
        return
    while actual is not None:
        estado = "Completada" if actual.tarea.completada else "Pendiente"
        print (actual)
        actual = actual.siguiente
```

Este método mostrar_tareas recorre la lista enlazada desde la cabeza y muestra cada tarea. Si la lista está vacía, imprime un mensaje indicando que no hay tareas agregadas. Para cada tarea, imprime su representación utilizando el método __str__ de la clase Nodo, que incluye el estado de la tarea como "Completada" o "Pendiente", y continúa hasta que todas las tareas en la lista hayan sido mostradas.

5. **Preguntas Conceptuales**

Preguntas para Análisis:

- ¿Qué sucede si intentamos agregar una tarea que ya existe en la lista?

Si intentamos agregar una tarea que ya existe en la lista, el método agregar_tarea verifica primero si la tarea ya está presente mediante el método buscar_ tarea_descripcion(descripcion). Si este método devuelve True, indicando que la tarea con la misma descripción ya existe, el método agregar_tarea imprime el mensaje "No se pudo

cargar la tarea. La tarea ya existe" y no realiza ninguna otra acción, por lo que la tarea no se agrega a la lista.

- ¿Cómo se implementa la priorización de tareas en la lista enlazada?

Se implementa la priorización de tareas en la lista enlazada al agregar una tarea a la lista enlazada de acuerdo con su nivel de prioridad. Primero, se solicita al usuario que ingrese una prioridad que debe ser un número entre 1 y 3. Luego, al llamar al método agregar_tarea, se inserta la tarea en la lista enlazada en la posición correcta basada en su prioridad. Las tareas con mayor prioridad se colocan antes en la lista, ya que el código compara la prioridad de la nueva tarea con la de los nodos existentes y la inserta antes si tiene una prioridad mayor. Esto asegura que las tareas más prioritarias aparecen antes en la lista.

- ¿Qué sucede si dos tareas tienen la misma prioridad?

Si dos tareas tienen la misma prioridad se mantienen en el orden en el que fueron añadidas. La tarea nueva se inserta después de la tarea existente con la misma prioridad. Esto se debe a cómo se maneja la inserción en la lista enlazada. Al agregar una nueva tarea con prioridad igual a la de las tareas existentes, el código sigue el enlace entre nodos para encontrar la posición correcta para la nueva tarea. Durante el recorrido, el código compara las prioridades y avanza hasta encontrar el primer nodo con una prioridad menor. En el bucle de inserción, si encuentra nodos con la misma prioridad que la nueva tarea, avanza al siguiente nodo sin realizar ninguna acción. Finalmente, el nuevo nodo se inserta después del último nodo con la misma prioridad existente en la lista.

- ¿Cuál es la complejidad temporal del método para eliminar una tarea de la lista enlazada?

La complejidad temporal del método eliminar_tarea para eliminar una tarea de una lista enlazada es O(n), donde n es el número de tareas en la lista.
El método recorre la lista enlazada desde la cabeza hasta encontrar el nodo que contiene la tarea con el ID especificado. En el peor de los casos, el nodo que se desea eliminar podría estar al final de la lista, lo que implica que se necesita recorrer toda la lista hasta encontrarlo. Durante el recorrido, se realiza una comparación del ID de cada tarea con el ID buscado. Esta comparación ocurre n veces en el peor caso, donde n es el número total de tareas en la lista. Una vez encontrado el nodo a eliminar, el proceso de ajustar los enlaces (actualizar el nodo anterior para saltar el nodo eliminado y posiblemente actualizar el encabezado de la lista) y de ajustar los contadores y el diccionario de categorías pendientes son operaciones que se realizan en tiempo constante, O(1).
Por lo tanto, el tiempo total requerido para encontrar y eliminar una tarea en la lista enlazada es proporcional al número total de elementos en la lista, resultando en una complejidad temporal de O(n).

- ¿Cómo podríamos modificar el sistema para soportar múltiples categorías por tarea?

Luego de investigar, consideramos que para soportar múltiples categorías por tarea, hay que modificar la clase Tarea para usar un set en lugar de una lista.
Un set en Python es una colección que almacena elementos únicos, sin duplicados. Utilizar un set para gestionar categorías permite almacenar múltiples categorías para cada tarea.. Las operaciones como agregar (add) y eliminar (discard) categorías son rápidas y automáticas en cuanto a evitar duplicados. La clase Tarea se inicializa con un set de categorías y proporciona métodos para agregar o quitar categorías usando las operaciones de add y discard del set.

6. Resultados y Conclusiones

Los resultados obtenidos fueron los esperados, logramos implementar las funcionalidades, descubrimos y aprendimos mucho más sobre estructuras de datos. A la hora de desarrollar el código tuvimos en cuenta sobre análisis de algoritmos buscando la mayor eficiencia en el código.

7. Referencias

Para llevar a cabo este proyecto utilizamos como referencia el material teórico de la asignatura; capítulo 3 sobre Estructuras de Datos.