

**“Año de la recuperación y consolidación de la economía
peruana”**

ASIGNATURA:

Estructura de Datos
(Plan 2024)

NRC: 29901

NRO GRUPO: 4

PROYECTO GRUPAL:

Gestión de Procesos con Estructuras Dinámicas

DOCENTE:

Rosario Delia Osorio Contreras

INTEGRANTES:

1. Mijael Joseph Bejarano Miche
2. Luis Enrique Cueva Barrera
3. Damián Javier Lopez Naula
4. Nelinho Capcha Hidalgo

Huancayo, 2025

ÍNDICE

Capítulo 1: Análisis del Problema	3
1.1. Descripción del Sistema	3
1.2. Requerimientos del Sistema	3
1.2.1. Requerimientos funcionales	3
1.2.2. Requerimientos no funcionales	4
1.3. Estructuras de datos propuestas	5
1.4. Justificación de la elección	6
Capítulo 2: Diseño de la Solución	8
2.1. Descripción de estructuras de datos y operaciones	8
2.2. Algoritmos principales	10
2.3. Diagramas de Flujo	16
2.4. Justificación del diseño	22
Capítulo 3: Solución Final	22
3.1. Código documentado en C++	22
3.1.1. Definiciones preliminares	22
3.1.2. Funciones del sistema	25
3.1.3. Menú del sistema	32
3.2. Pruebas de validación	33
3.3. Manual de Usuario	39
Capítulo 4: Evidencias de Trabajo en Equipo	44
4.1. Control de Versiones	44
4.1.1. Registro de commits	44
4.1.2. Evidencias adicionales	47
4.2. Plan de Trabajo y Roles Asignados	48
4.2.1. Descripción del rol de cada miembro por semana	48
4.2.2. Cronograma con fechas límite	51
4.2.3. Actas de reunión	53

Capítulo 1: Análisis del Problema

1.1. Descripción del Sistema

El **Sistema de Gestión de Procesos** es una aplicación desarrollada en **Pseint y Dev-C++**, orientada a simular la administración de tareas en un entorno similar al de un sistema operativo. Su diseño se basa exclusivamente en el uso de estructuras de datos dinámicas lineales usadas en la unidad 3 del curso, tales como colas, pilas y listas enlazadas. El sistema se organiza en tres módulos principales:

- **Gestor de Procesos:** Utiliza una lista enlazada para registrar todos los procesos activos. Este módulo permite agregar, buscar y eliminar procesos, así como modificar la prioridad de un proceso determinado.
- **Planificador de CPU:** Utilizando una cola de prioridad, permite simular la ejecución de procesos según su nivel de prioridad. Se encarga de encolar procesos de acuerdo con su prioridad, desencolar un proceso, es decir ejecutar aquel con mayor prioridad, además de permitir la visualización del estado actual de la cola.
- **Gestor de Memoria:** Usa una pila para simular la asignación y liberación de memoria a los procesos, siguiendo el principio LIFO (primero en entrar último en salir). Permite asignar memoria (push), liberarla (pop), y verificar el estado actual de memoria.

1.2. Requerimientos del Sistema

1.2.1. Requerimientos funcionales

Los requerimientos funcionales deben asegurar el cumplimiento de las funciones básicas de un administrador de tareas. Para ello, cada módulo del sistema debe implementar sus funciones usando las estructuras de datos vistas en clase durante la Unidad 3. A continuación, se detallan las funciones que debe cumplir cada módulo.

A. Gestor de Procesos (Lista Enlazada):

El módulo gestor de procesos del sistema debe permitir las siguientes funciones:

- **RF1.1:** Insertar nuevos procesos (nombre, ID, prioridad).
- **RF1.2:** Crear una lista enlazada de los procesos insertados, con nodos que contienen su información.
- **RF1.3:** Eliminar procesos por ID.

- **RF1.4:** Buscar procesos por ID.
- **RF1.5:** Modificar la prioridad de un proceso.

B. Planificador de CPU (cola de prioridad):

Para el correcto funcionamiento, el módulo planificador de CPU debe implementar las siguientes funciones:

- **RF2.1:** Encolar procesos en base a prioridad, es decir ordenarlos en base a la prioridad,
- **RF2.2:** Ejecutar procesos (desencolar).
- **RF2.3:** Ver la cola actual de procesos.

C. Gestor de Memoria (pila):

El módulo gestor de procesos del sistema debe contar con las siguientes funcionalidades:

- **RF3.1:** Asignar memoria a procesos (push).
- **RF3.2:** Liberar memoria (pop) usando la lógica LIFO, (último en entrar, primero en salir).
- **RF3.3:** Mostrar el estado de la memoria.

1.2.2. Requerimientos no funcionales

Los requerimientos no funcionales del sistema deben asegurar una buena experiencia de usuario y detalles de calidad. A continuación, se detallan aspectos que el sistema debe cumplir:

- **RNF 1:** La interfaz debe ser simple de usar y comprensible para un usuario promedio.
- **RNF 2:** El sistema debe manejar validación de números enteros o de carácter o errores como pila o lista vacía.
- **RNF 3:** El sistema debe responder a las operaciones del usuario en menos de 10 segundos para un número razonable de procesos (por ejemplo, hasta 50 procesos).
- **RNF 4:** El código debe estar organizado y documentado para facilitar futuras modificaciones y ampliaciones.

1.3. Estructuras de datos propuestas

En este sistema, se utilizan tres estructuras básicas para manejar los procesos y la memoria; una lista enlazada simple para almacenar los procesos, una pila para simular la asignación y liberación de memoria, y una cola con prioridad para gestionar la ejecución de procesos en la CPU. A continuación, una explicación de cómo están estructuradas y para qué sirve cada una.

a. Lista enlazada simple para procesos

Para guardar todos los procesos creados o cargados, usamos una lista enlazada simple. Esto significa que cada proceso es un "nodo" que contiene la información del proceso (como su ID, nombre y prioridad) y un puntero que señala al siguiente proceso en la lista.

```
struct Proceso {  
    string id;  
    string nombre;  
    int prioridad;  
    Proceso* siguiente;  
};
```

Cada nodo contiene:

- **ID:** Identificador único que distingue al proceso.
- **Nombre:** Un nombre descriptivo.
- **Prioridad:** Un número que indica qué tan importante es el proceso.
- **Puntero siguiente:** Que conecta este proceso con el siguiente en la lista.

Esta estructura permite agregar, eliminar o buscar procesos fácilmente, sin necesidad de mover datos o usar memoria fija. Es muy útil cuando la cantidad de procesos cambia constantemente.

b. Cola con prioridad para la CPU

Para decidir qué proceso se ejecuta primero, usamos una cola ordenada por prioridad. Esto significa que los procesos se mantienen en fila, pero quienes tienen mayor prioridad están más adelante en la cola.

```
struct NodoCola {  
    string id;  
    string nombre;  
    int prioridad;  
    NodoCola* siguiente;  
};
```

Cada nodo contiene:

- **ID y nombre:** Identifican el proceso.
- **Prioridad:** Indica la urgencia o importancia.
- **Puntero siguiente:** Apunta al siguiente proceso en la cola.

Al insertar procesos, estos se colocan en la posición correcta según su prioridad, de modo que el proceso con mayor prioridad siempre sea el primero en la cola y, por tanto, el primero en ejecutarse en la CPU. Así, la cola sigue una lógica FIFO donde el primero en la cola es el que tiene mayor prioridad.

c. Pila para simular la memoria

Para manejar la memoria asignada a los procesos, usamos una pila, que funciona como una pila de platos; el último proceso al que se le asigna memoria es el primero en liberarla (LIFO).

```
struct NodoMemoria {  
    string procesoID;  
    NodoMemoria* siguiente;  
};
```

Cada nodo de la pila tiene:

- **ID del proceso:** Para saber a qué proceso se le asignó esa parte de memoria.
- **Puntero siguiente:** Que apunta al nodo anterior en la pila (el que se asignó justo antes).

Esta estructura es ideal para simular memoria que se asigna y libera de forma ordenada (último en entrar, primero en salir), y las operaciones de asignar o liberar memoria son muy rápidas.

1.4. Justificación de la elección

El objetivo principal de este código es simular el proceso y comportamiento de un administrador de tareas en un sistema operativo, donde se manejan procesos, su ejecución en la CPU y la asignación de memoria.

Para lograr esto, se han elegido nodos en vez de trabajar con arreglos, por los siguientes motivos:

- **Gestión eficiente de memoria:** Los nodos permiten una gestión dinámica de la memoria, ya que se asigna sólo el espacio necesario para cada elemento. Por otro lado, los arreglos requieren una cantidad fija de memoria, lo que puede llevar a la sobrecarga de espacio o a problemas de desbordamiento si no se conoce el número exacto de elementos de antemano.
- **Adaptabilidad a cambios en el tamaño de la estructura:** Los nodos son ideales cuando el tamaño de la estructura de datos cambia con frecuencia o no se conoce de antemano. Se pueden agregar o eliminar nodos sin necesidad de redimensionar la estructura.

- **Mejor rendimiento en aplicaciones con datos cambiantes:** Cuando los datos que se manejan no son fijos, como en el caso de tu código, los nodos permiten un manejo más ágil y menos costoso en comparación con los arreglos estáticos.

Entonces, a partir de estos nodos se crearon las estructuras dinámicas vistas en la Unidad 3 (pilas, listas y colas), ya que representan la forma más práctica de implementar el gestor de procesos, el planificador de CPU y el gestor de memoria.

- **Lista enlazada para los procesos:** Las listas enlazadas, que usan nodos, permiten insertar y eliminar elementos en cualquier posición de manera eficiente, actualizando únicamente los punteros de los nodos adyacentes. En cambio, con los arreglos, insertar o eliminar elementos en una posición intermedia requiere mover otros elementos, lo cual puede ser costoso en términos de tiempo.
- **Cola con prioridad para la CPU:** La CPU ejecuta los procesos organizándolos según su prioridad, de manera similar a un planificador de tareas. En este sistema, cada proceso se inserta en la cola según su nivel de prioridad, asegurando que el proceso con mayor prioridad se ubique siempre al frente y sea el primero en ejecutarse. Así, la cola mantiene un orden FIFO, donde el primero en la lista es el proceso con la prioridad más alta, lo que permite una gestión eficiente y enfocada en la importancia de cada tarea.

Para manejar esta estructura, se utilizan nodos y punteros que facilitan la administración dinámica de la memoria. Los nodos almacenan la información de cada proceso, mientras que los punteros enlazan estos nodos en la cola, permitiendo agregar nuevos procesos al final y eliminar los que ya se ejecutaron sin necesidad de reorganizar toda la estructura. Esto incrementa la eficiencia y agiliza el control sobre los recursos y los tiempos de ejecución dentro de la cola.

- **Pila para la Memoria:** La pila simula la asignación y liberación de memoria siguiendo el principio LIFO, que es común en el manejo real de memoria en los sistemas operativos. Esto permite representar cómo se asigna memoria a los procesos y cómo se libera en orden inverso, facilitando una simulación realista y eficiente.

El uso de nodos y punteros es crucial en este proceso LIFO, ya que permiten gestionar dinámicamente la memoria. Los nodos almacenan los datos de los procesos, y los punteros mantienen la conexión entre ellos, lo que facilita la inserción y eliminación de elementos en la pila sin necesidad de mover toda la memoria, optimizando así el rendimiento y la eficiencia en la gestión de recursos.

En conjunto, estas estructuras no solo son sencillas y fáciles de implementar, sino que también representan fielmente el funcionamiento básico de un administrador de tareas. Así, el código simula de manera clara y didáctica cómo se gestionan procesos, su ejecución y la memoria asociada, sirviendo como una herramienta educativa o base para sistemas más complejos.

Capítulo 2: Diseño de la Solución

2.1. Descripción de estructuras de datos y operaciones

En el proceso de diseño de este sistema gestor de procesos y administración de memoria, se optó por un enfoque inicial centrado en estructuras simples y estáticas, con el objetivo de facilitar la implementación, el análisis y la validación lógica. En esta etapa, se descartó el uso de estructuras dinámicas como listas enlazadas, pilas o colas implementadas con nodos y punteros, priorizando el uso de arreglos unidimensionales como base estructural del sistema. A continuación, se describen las principales estructuras de datos utilizadas:

A. Lista de procesos:

Se emplean tres arreglos principales para representar los procesos: **procesos_id[100]**, **procesos_nombre[100]** y **procesos_prioridad[100]**. El primero almacena el identificador único de cada proceso, actuando como clave de referencia para los demás, mientras que **procesos_nombre** contiene el nombre asociado y **procesos_prioridad** registra su nivel de prioridad. Estos tres arreglos funcionan de forma paralela como una lista secuencial, donde cada índice representa un proceso completo. Gracias a esta organización, es posible realizar operaciones como inserción, búsqueda, eliminación y ordenamiento sin necesidad de punteros ni estructuras dinámicas. Además, la variable **totalProcesos** cumple un rol clave al llevar el conteo de procesos activos y gestionar la posición de inserción, simulando así eficientemente el comportamiento de una lista simple.

Operaciones:

- **Insertión:** Se agrega un nuevo proceso en la siguiente posición disponible del arreglo usando **totalProcesos** como índice. Esto implica almacenar el ID, nombre y prioridad en los arreglos correspondientes y luego incrementar **totalProcesos**.
- **Búsqueda:** Se recorre el arreglo **procesos_id** para localizar un proceso que coincida con el ID buscado
- **Eliminación:** Para eliminar un proceso, se localiza su índice y luego se desplazan los elementos posteriores una posición hacia atrás, sobrescribiendo el proceso eliminado y manteniendo la secuencia continua.
- **Ordenamiento:** Se reordenan los elementos en base a un criterio, como la prioridad, intercambiando los valores correspondientes en los tres arreglos para mantener la coherencia de los datos.

B. Cola de ejecución

Aunque no se implementó una estructura de cola explícita, el sistema simula una cola de ejecución por prioridades. Para ello, se aplica un algoritmo de ordenamiento sobre los procesos, reorganizándolos de tal forma que los de mayor prioridad (menor valor) se encuentren en las primeras posiciones del arreglo. Luego, para poder realizar las operaciones.

Operaciones:

- **Ordenar procesos por prioridad:**
Reorganiza los procesos en los arreglos para que aquellos con mayor prioridad (menor número) aparezcan primero.
- **Ejecutar procesos con mayor prioridad:**
Simula una extracción (desencolar) de la cola, eliminando el proceso con la prioridad más alta.
- **Visualizar cola actual(ordenada):**
Muestra la lista de procesos ordenada por prioridad, representando el estado actual de la cola de ejecución.

C. Pila de memoria

El arreglo **memoria[100]** simula una pila LIFO para representar bloques de memoria asignados a procesos, controlada por la variable **topePila** que indica el último elemento insertado. Las operaciones básicas incluyen añadir un bloque (push) y eliminar el último bloque (pop), manteniendo así el orden de gestión de memoria.

Operaciones:

- **Asignar memoria (Push):**
Inserta un ID de proceso en la cima de la pila, simulando la asignación de un bloque de memoria.
- **Liberar memoria (Pop):**
Elimina el elemento que se encuentra en la cima de la pila, representando la liberación del bloque de memoria más recientemente asignado.
- **Ver estado actual de memoria:**
Muestra el contenido actual de la pila desde la cima hacia abajo, permitiendo visualizar los procesos que tienen memoria asignada.

2.2. Algoritmos principales

a. (Opción 1): Inserta los procesos

Permite al usuario registrar un nuevo proceso contando con un máximo de 100 procesos registrar, solicita al usuario el ID, nombre y la prioridad que va a tener el nuevo proceso, estos datos son guardados en los arreglos **procesos_id**, **procesos_nombre** y **procesos_prioridad** y aumenta el contador llamado **totalProcesos**.

```
1: // Insertar proceso
Si totalProcesos < 100 Entonces
    Escribir "Ingrese ID del proceso:"
    Leer idNuevo
    Escribir "Ingrese nombre del proceso:"
    Leer nombre
    Escribir "Ingrese prioridad (entero):"
    Leer prioridad

    totalProcesos ← totalProcesos + 1
    procesos_id[totalProcesos] ← idNuevo
    procesos_nombre[totalProcesos] ← nombre
    procesos_prioridad[totalProcesos] ← prioridad

    Escribir "Proceso insertado correctamente."
Sino
    Escribir "¡Límite máximo de procesos alcanzado!"
FinSi
```

b. (Opción 2): Muestra la lista de los procesos

Recorre los arreglos y muestra los procesos añadidos de orden cronológicamente, no se aplica el orden por prioridad, además si no se tiene ningún proceso se muestra el mensaje "No hay procesos registrados".

```
2: // Listar procesos sin ordenar
Si totalProcesos = 0 Entonces
    Escribir "No hay procesos registrados."
Sino
    Escribir "Lista de procesos:"
    Para i ← 1 Hasta totalProcesos
        Escribir "ID: ", procesos_id[i], " | Nombre: ", procesos_nombre[i], " | Prioridad: ", procesos_prioridad[i]
    FinPara
FinSi
```

c. (Opción 3): Elimina procesos por el ID

Primero verifica si hay procesos registrados y no muestra un mensaje al usuario, si hay procesos pide el ID del proceso y lo busca dentro del arreglo **procesos_id**, si encuentra el ID lo elimina desplazando los datos en los arreglos para no dejar espacios vacíos, si no encuentra el ID se informa al usuario que no existe el proceso con ese ID.

```
3: // Eliminar proceso por ID
Si totalProcesos = 0 Entonces
    Escribir "No hay Procesos para eliminar."
SiNo
    Escribir "Ingrese el ID del proceso a eliminar:"
    Leer idBuscado
    encontrado ← -1
    Para i ← 1 Hasta totalProcesos
        Si procesos_id[i] = idBuscado Entonces
            encontrado ← i
        FinSi
    FinPara

    Si encontrado ≠ -1 Entonces
        Si encontrado < totalProcesos Entonces
            Para i ← encontrado Hasta totalProcesos - 1
                procesos_id[i] ← procesos_id[i+1]
                procesos_nombre[i] ← procesos_nombre[i+1]
                procesos_prioridad[i] ← procesos_prioridad[i+1]
            FinPara
        FinSi
        totalProcesos ← totalProcesos - 1
        Escribir "Proceso eliminado correctamente."
    Sino
        Escribir "No se encontró un proceso con ese ID."
    FinSi
FinSi
```

d. (Opción 4): Busca procesos por ID

Solicita el ID del proceso que se desea buscar y recorre el arreglo **procesos_id**, Si lo encuentra muestra toda la información del proceso, si en caso el ID no se encuentra o no existe se informa de que no se encontró.

```
4: // Buscar proceso por ID
Escribir "Ingrese el ID del proceso a buscar:"
Leer idBuscado
encontrado ← -1
Para i ← 1 Hasta totalProcesos
    Si procesos_id[i] = idBuscado Entonces
        encontrado ← i
    FinSi
FinPara

Si encontrado ≠ -1 Entonces
    Escribir "Proceso encontrado:"
    Escribir "ID: ", procesos_id[encontrado]
    Escribir "Nombre: ", procesos_nombre[encontrado]
    Escribir "Prioridad: ", procesos_prioridad[encontrado]
Sino
    Escribir "No se encontró un proceso con ese ID."
FinSi
```

e. (Opción 5): Modifica la prioridad de un proceso

Permite modificar la prioridad del proceso buscando por su ID y solicita al usuario ingresar la nueva prioridad luego actualiza el valor en el arreglo **procesos_prioridad**, además de que permite cambiarlo sin alterar los demás datos del proceso.

```
5: // Modificar prioridad
Escribir "Ingrese el ID del proceso para modificar prioridad:"
Leer idBuscado
encontrado ← -1
Para i ← 1 Hasta totalProcesos
    Si procesos_id[i] = idBuscado Entonces
        encontrado ← i
    FinSi
FinPara

Si encontrado ≠ -1 Entonces
    Escribir "Prioridad actual: ", procesos_prioridad[encontrado]
    Escribir "Ingrese nueva prioridad (entero):"
    Leer prioridadAux
    procesos_prioridad[encontrado] ← prioridadAux
    Escribir "Prioridad modificada correctamente."
Sino
    Escribir "No se encontró un proceso con ese ID."
FinSi
```

f. (Opción 6): Ordena procesos (Mayor -> Menor)

Permite ordenar cada proceso por prioridad mayor a prioridad menor, usa el tipo de ordenamiento burbuja no solo reorganiza el arreglo de **procesos_prioridad** también actualiza en paralelo los arreglos **procesos_id** y **procesos_nombre** para mantener la información de los procesos alineados, para evitar errores o pérdidas se usó variables auxiliares durante los intercambios.

```

6: // Ordenar procesos por prioridad (mayor a menor)
Si totalProcesos ≥ 2 Entonces
  Para i ← 1 Hasta totalProcesos - 1
    Para j ← i + 1 Hasta totalProcesos
      Si procesos_prioridad[i] > procesos_prioridad[j] Entonces
        prioridadAux ← procesos_prioridad[i]
        procesos_prioridad[i] ← procesos_prioridad[j]
        procesos_prioridad[j] ← prioridadAux

        idAux ← procesos_id[i]
        procesos_id[i] ← procesos_id[j]
        procesos_id[j] ← idAux

        nombreAux ← procesos_nombre[i]
        procesos_nombre[i] ← procesos_nombre[j]
        procesos_nombre[j] ← nombreAux
      FinSi
    FinPara
  FinPara
  Escribir "Procesos ordenados por prioridad (menor número = mayor prioridad)."
  Escribir "Lista ordenada:"
  Para i ← 1 Hasta totalProcesos
    Escribir "ID: ", procesos_id[i], " | Nombre: ", procesos_nombre[i], " | Prioridad: ", procesos_prioridad[i]
  FinPara
Sino
  Escribir "No hay suficientes procesos para ordenar."
FinSi

```

g. (Opción 7): Desencolamiento del proceso

Esta opción ejecuta el proceso con mayor prioridad (menor número en **procesos_prioridad**). Si **totalProcesos** es 0, muestra un mensaje. Si no, busca el índice **indiceMin** del proceso con menor prioridad, mostrando su información. Luego, elimina ese proceso desplazando los valores de los arreglos (**procesos_id**, **procesos_nombre**, **procesos_prioridad**) una posición hacia arriba y reduce **totalProcesos** en uno.

```

7: // Ejecutar proceso con mayor prioridad (menor número)
Si totalProcesos = 0 Entonces
  Escribir "No hay procesos para ejecutar."
Sino
  // Buscar índice con prioridad mínima
  Definir indiceMin Como Entero
  indiceMin ← 1
  Para i ← 2 Hasta totalProcesos
    Si procesos_prioridad[i] < procesos_prioridad[indiceMin] Entonces
      indiceMin ← i
    FinSi
  FinPara

  // Mostrar proceso a ejecutar
  Escribir "Ejecutando proceso con mayor prioridad:"
  Escribir "ID: ", procesos_id[indiceMin], " | Nombre: ", procesos_nombre[indiceMin], " | Prioridad: ", procesos_prioridad[indiceMin]

  // Eliminar proceso desplazando
  Para i ← indiceMin Hasta totalProcesos - 1
    procesos_id[i] ← procesos_id[i + 1]
    procesos_nombre[i] ← procesos_nombre[i + 1]
    procesos_prioridad[i] ← procesos_prioridad[i + 1]
  FinPara
  totalProcesos ← totalProcesos - 1
FinSi

```

h. (Opción 8): Visualiza la cola actual

Primero verifica si hay procesos registrados, si hay ordena temporalmente los arreglos **procesos_id**, **procesos_nombre** y **procesos_prioridad** utilizando el algoritmo de burbuja para el ordenamiento y muestra todos los procesos ordenados por prioridad de manera ascendente, similar a la opción 6, sin embargo, no se modifica permanentemente si están ordenados.

```
8: // Visualizar cola actual (ordenada)
  Si totalProcesos = 0 Entonces
    Escribir "La cola está vacía."
  Sino
    // Ordenar antes de mostrar para asegurar orden ascendente por prioridad
    Si totalProcesos ≥ 2 Entonces
      Para i ← 1 Hasta totalProcesos - 1
        Para j ← i + 1 Hasta totalProcesos
          Si procesos_prioridad[i] > procesos_prioridad[j] Entonces
            prioridadAux ← procesos_prioridad[i]
            procesos_prioridad[i] ← procesos_prioridad[j]
            procesos_prioridad[j] ← prioridadAux

            idAux ← procesos_id[i]
            procesos_id[i] ← procesos_id[j]
            procesos_id[j] ← idAux

            nombreAux ← procesos_nombre[i]
            procesos_nombre[i] ← procesos_nombre[j]
            procesos_nombre[j] ← nombreAux
          FinSi
        FinPara
      FinPara
    FinSi

    Escribir "Cola actual (ordenada por prioridad):"
    Para i ← 1 Hasta totalProcesos
      Escribir "ID: ", procesos_id[i], " | Nombre: ", procesos_nombre[i], " | Prioridad: ", procesos_prioridad[i]
    FinPara
  FinSi
```

i. (Opción 9): Asignar memoria(push)

Simula el comportamiento de una pila de memoria donde el último proceso asignado es el primero en liberarse, pero primero verifica el espacio de la memoria es decir el **topePila**, si hay espacio solicita el ID del proceso y lo almacena en el arreglo *memoria*. Luego incrementa el **topePila**.

```
9: // Asignar memoria (push)
  Si topePila < 100 Entonces
    Escribir "Ingrese ID del proceso para asignar memoria:"
    Leer bloqueID
    topePila ← topePila + 1
    memoria[topePila] ← bloqueID
    Escribir "Memoria asignada correctamente al proceso."
  Sino
    Escribir "¡La memoria está llena!"
  FinSi
```

j. (Opción 10): Libera memoria(pop)

Primero verifica el estado de la memoria o el arreglo **topePila** si el **topePila** es 0 no hay procesos almacenados y se informa al usuario luego libera el último bloque de memoria asignado para luego disminuir el **topePila** borrando el contenido de esa posición y actualiza el **topePila**, simula la liberación de la memoria de una pila.

```
10: // Liberar memoria (pop)
Si topePila > 0 Entonces
    Escribir "Liberando memoria del proceso ID: ", memoria[topePila]
    memoria[topePila] ← "" // Aquí limpiamos el valor visiblemente
    topePila ← topePila - 1
Sino
    Escribir "La memoria ya está vacía."
FinSi
```

k. (Opción 11): Muestra el estado de la memoria

Muestra el estado de la memoria desde el más reciente al anterior en tiempo real. Si no hay procesos en la memoria es decir **topePila** es 0 indica que la memoria está vacía y se informa al usuario, si hay procesos recorre el arreglo *memoria* desde el tope hacia abajo mostrando los ID de los procesos.

```
11: // Ver estado actual de la memoria
Si topePila = 0 Entonces
    Escribir "La memoria está vacía."
Sino
    Escribir "Estado actual de la memoria:"
    Para i ← topePila Hasta 1 Con Paso -1
        Escribir "Posición del proceso ", i, ": ", memoria[i]
    FinPara
FinSi
```

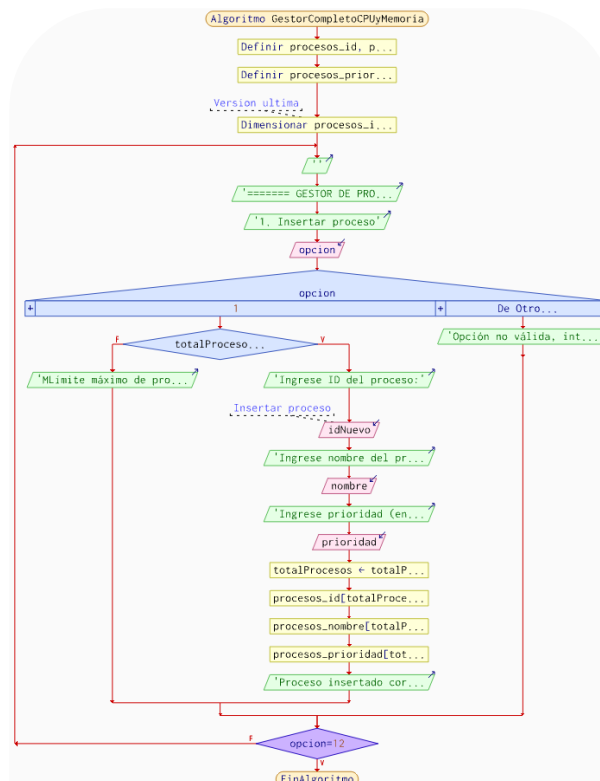
l. (Opción 12): Salir

Finalización del programa controlando el flujo y saliendo del bucle principal que contiene el menú, al final se muestra un mensaje.

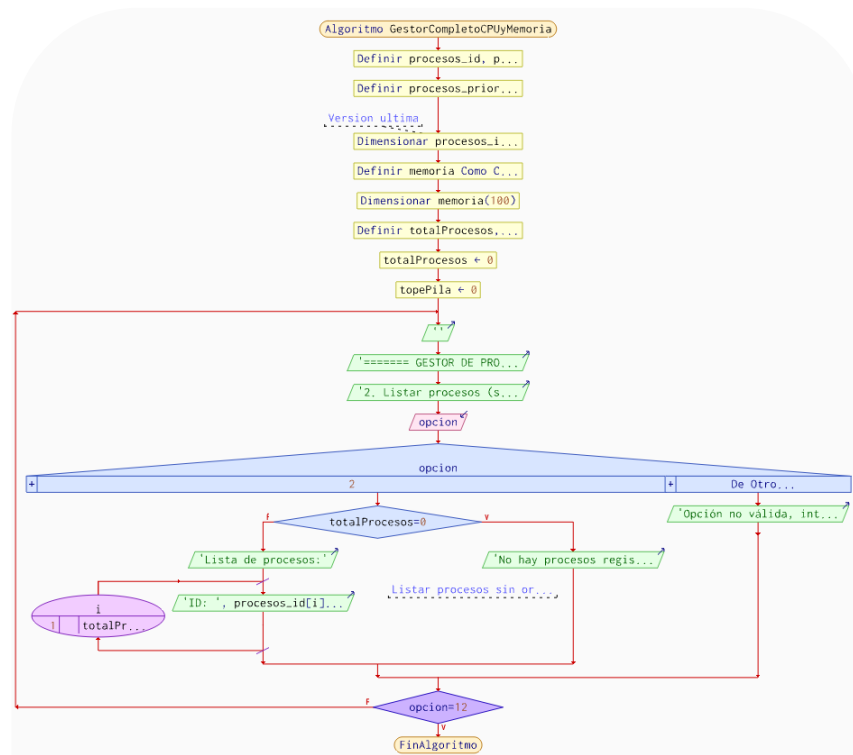
```
12:
    Escribir "Saliendo del programa."
```

2.3. Diagramas de Flujo

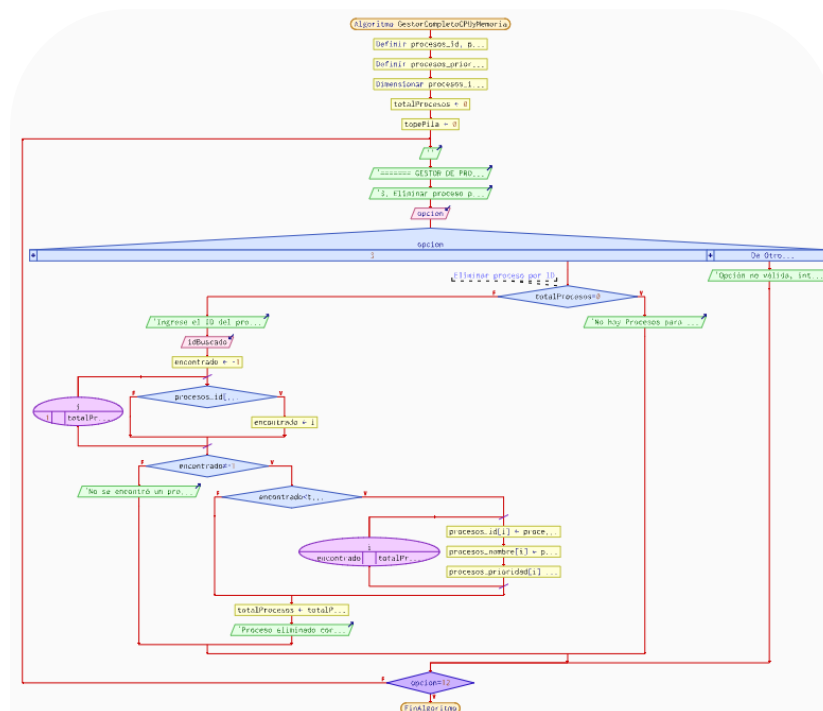
A. Opción 1. Insertar proceso:



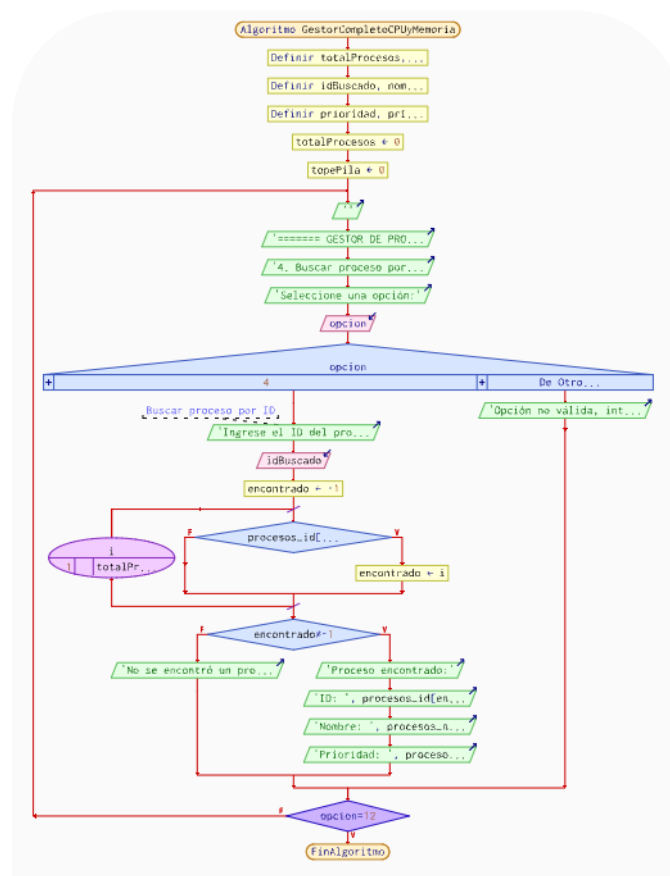
B. Opción 2. Listar Procesos (en desorden):



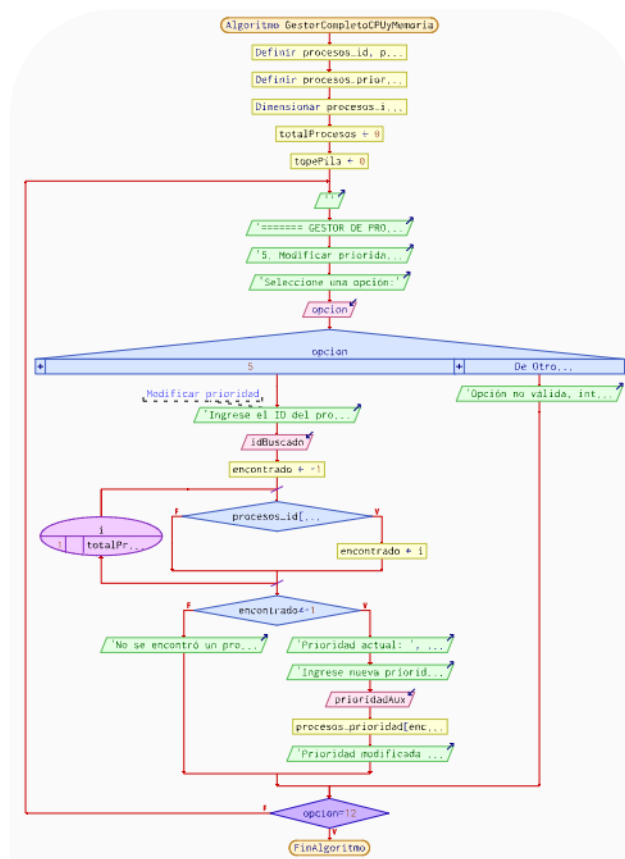
C. Opción 3. Eliminar proceso por ID:



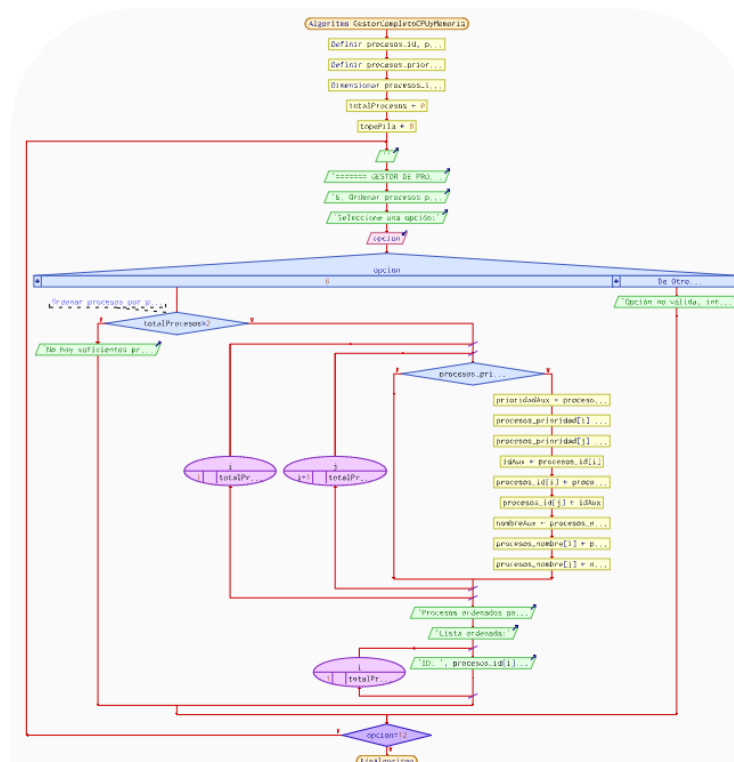
D. Opción 4. Buscar proceso por ID:



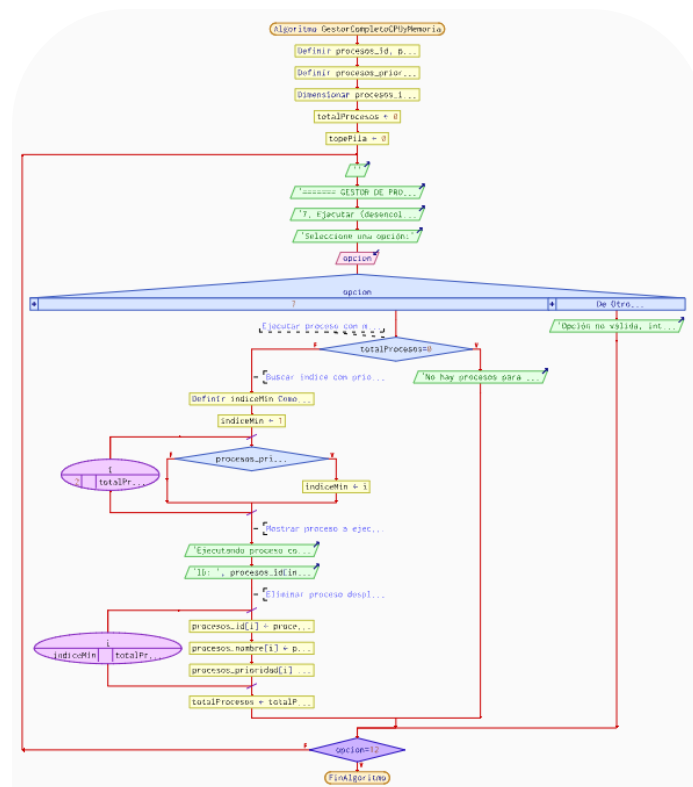
E. Opción 5. Modificar prioridad de un proceso:



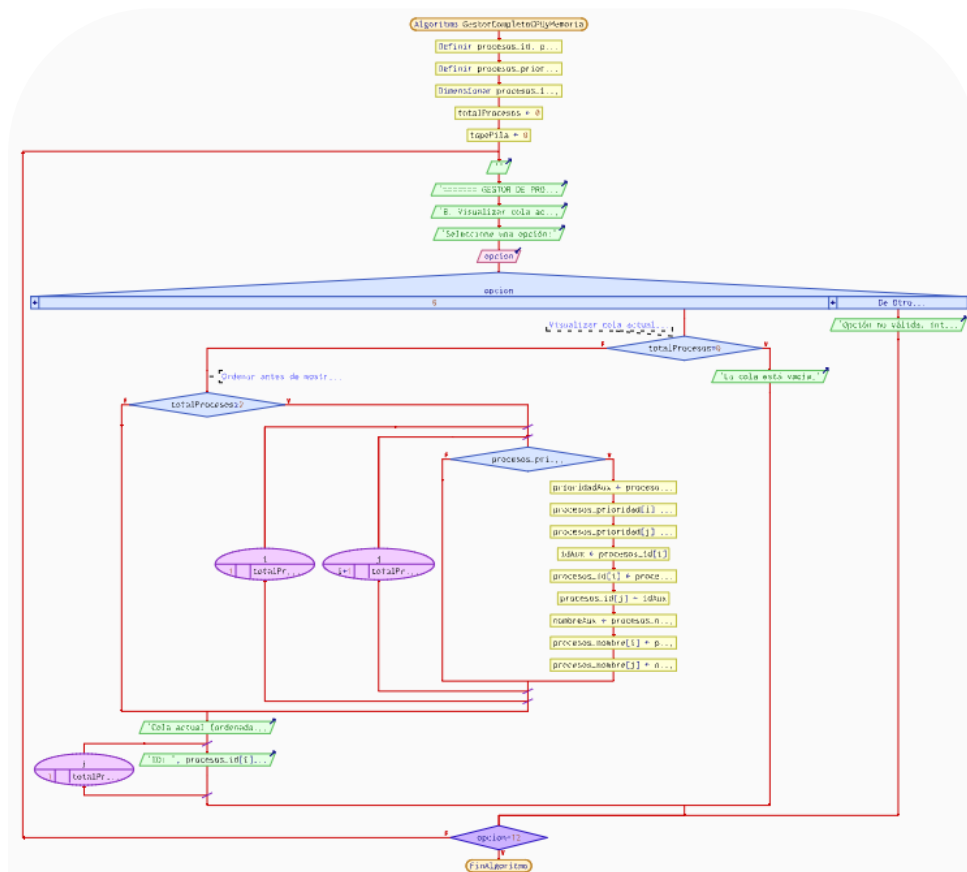
F. Opción 6. Ordenar procesos por prioridad:



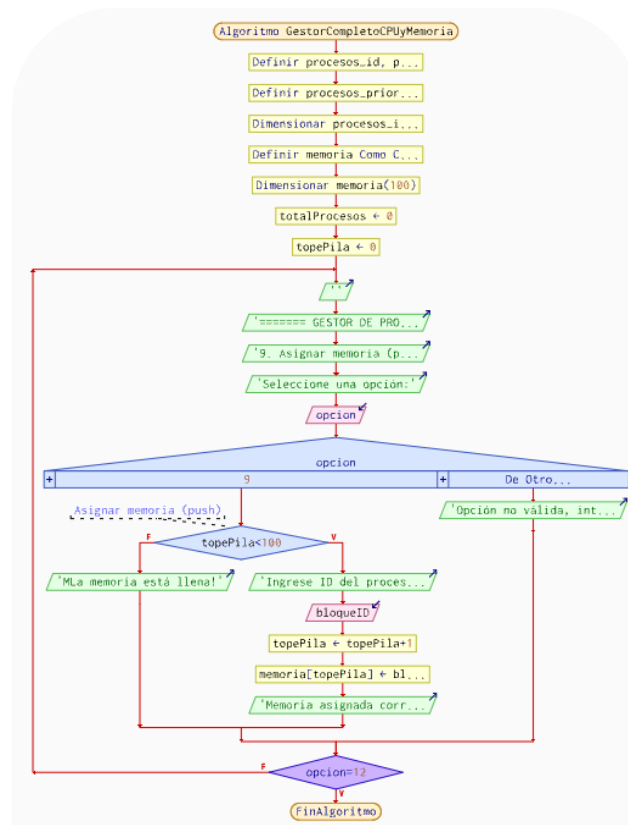
G. Opción 7. Ejecutar proceso con mayor prioridad:



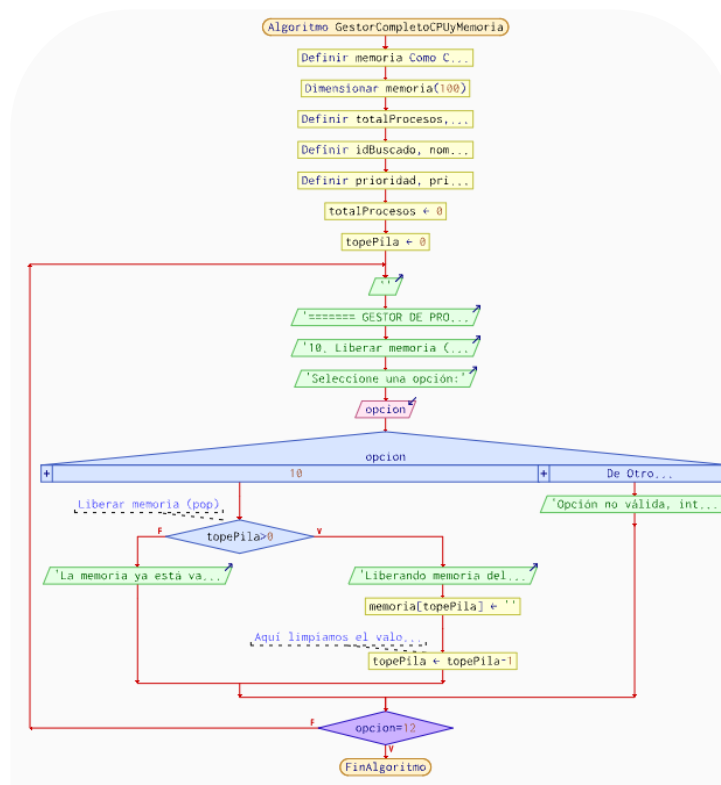
H. Opción 8. Visualizar cola actual:



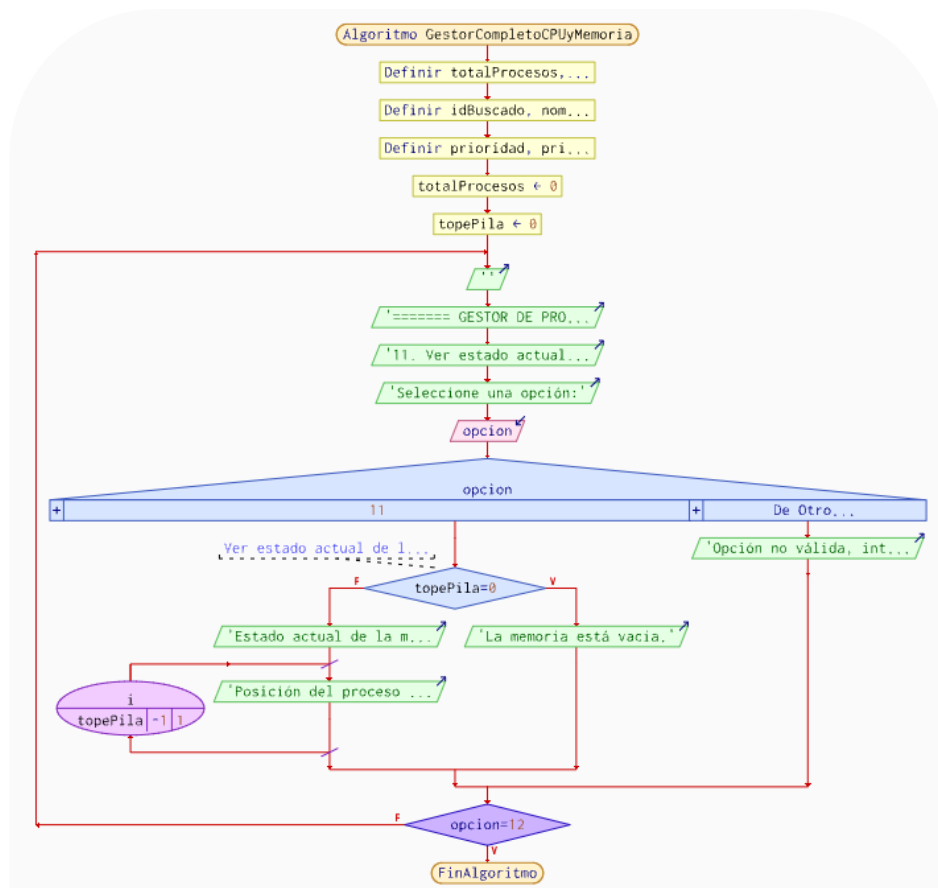
I. Opción 9. Asignar memoria (push):



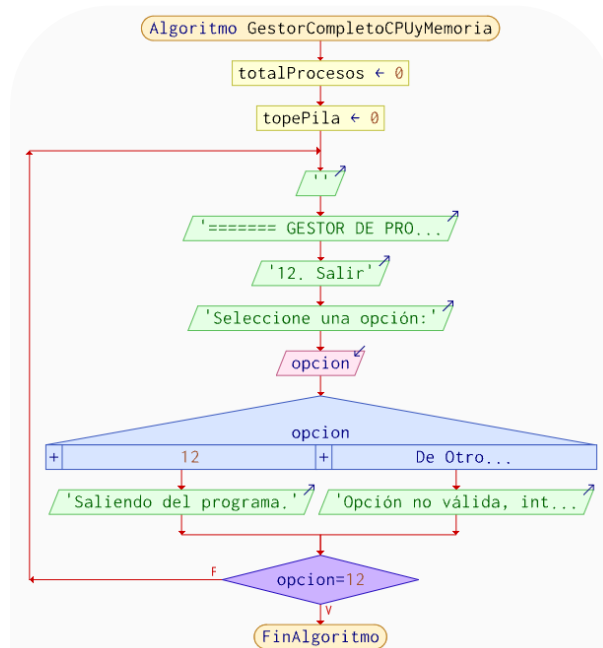
J. Opción 10. Liberar memoria (pop):



K. Opción 11. Ver estado de memoria actual:



L. Opción 12. Salir:



2.4. Justificación del diseño

El diseño del sistema se basa en el uso de arreglos unidimensionales para simular estructuras de datos como listas, pilas y colas, ya que PSeInt no cuenta con soporte nativo para estas estructuras. Esta elección permitió implementar de manera funcional las operaciones necesarias para la gestión de procesos, asignación de memoria y control de solicitudes, todo ello dentro de las limitaciones del entorno. Además, trabajar con arreglos simplifica el proceso de codificación y facilita la visualización del comportamiento del algoritmo paso a paso, lo que resulta beneficioso tanto para su validación como para la comprensión lógica del funcionamiento del sistema.

Capítulo 3: Solución Final

3.1. Código documentado en C++

3.1.1. Definiciones preliminares

A. Inclusión de Bibliotecas

El inicio de nuestro código en C++ incluye tres bibliotecas esenciales: `<iostream>` para manejar la entrada y salida estándar mediante objetos como `cin` y `cout`, `<fstream>` para crear y manipular archivos .txt que almacenan las operaciones realizadas en el gestor de procesos, utilizando clases como `ifstream` y `ofstream`, y `<string>` para emplear la clase `string`, que facilita el manejo de cadenas de texto. Además, la línea `using namespace std;` nos permite escribir el código de forma más corta, ya que no es necesario poner `std::` delante de palabras como `cout` o `string`.

```
2 #include <iostream>    // Librería para entrada y salida estándar
3 #include <fstream>     // Librería para manejo de archivos
4 #include <string>      // Librería para uso de cadenas
5 using namespace std;
```

B. Estructuras y Variables Globales

- **Estructura Proceso (Lista enlazada)** : Esta estructura funciona como un nodo que representa un proceso en una **lista enlazada**. Contiene un identificador único (`id`), el nombre del proceso (`nombre`), su prioridad (`prioridad`), y un puntero (`siguiente`) al siguiente proceso en la lista, permitiendo enlazar varios procesos. Además, incluye un constructor que inicializa estos valores y asigna **NULL** al puntero siguiente, indicando que al crearse no está conectado a otro nodo.

```

===== ESTRUCTURAS =====
// Estructura para representar un proceso en la lista enlazada
struct Proceso {
    string id;           // Identificador único del proceso
    string nombre;       // Nombre del proceso
    int prioridad;       // Nivel de prioridad del proceso (a mayor número, mayor prioridad)
    Proceso* siguiente;  // Puntero al siguiente proceso en la lista

    // Constructor que inicializa los datos del proceso
    Proceso(string _id, string _nombre, int _prioridad) {
        id = _id;
        nombre = _nombre;
        prioridad = _prioridad;
        siguiente = NULL; // Apunta a NULL al crearse
    }
}

```

- **Estructura NodoMemoria (Pila):** Esta estructura define un nodo que representa un proceso en una **pila de memoria**. Contiene un identificador (**procesoID**) y un puntero (**siguiente**) al nodo anterior, siguiendo la lógica **LIFO** (último en entrar, primero en salir).

```

// Nodo para representar un proceso en la pila de memoria
struct NodoMemoria {
    string procesoID; // ID del proceso asignado en memoria
    NodoMemoria* siguiente; // Apunta al nodo anterior (comportamiento de pila)
};

```

- **Estructura NodoCola (Cola):** Esta estructura simula un nodo que representa un proceso en una cola de CPU. **NodoCola** contiene un identificador (**id**), nombre (**nombre**), prioridad (**prioridad**) y un puntero (**siguiente**) al siguiente nodo, que enlaza los procesos en orden de llegada **FIFO** (primero en entrar, primero en salir).

```

// Nodo para representar un proceso en la cola de CPU
struct NodoCola {
    string id;           // ID del proceso
    string nombre;       // Nombre del proceso
    int prioridad;       // Prioridad del proceso
    NodoCola* siguiente; // Apunta al siguiente nodo en la cola
};

```

- **Variables globales:** Estas variables globales representan las estructuras principales para gestionar los procesos: **listaProcesos** es la lista enlazada con todos los procesos, **pilaMemoria** simula la memoria usando una pila, y **colaCPU** organiza los procesos en una cola según su prioridad para ser ejecutados por la CPU.

```

// ===== VARIABLES GLOBALES =====

Proceso* listaProcesos = NULL; // Lista enlazada que contiene todos los procesos
NodoMemoria* pilaMemoria = NULL; // Pila (stack) que simula la memoria (push/pop)
NodoCola* colaCPU = NULL; // Cola de procesos ordenada por prioridad

```

C. Funciones auxiliares

- **Validación para la opción 7:** Esta función se usa para evitar duplicados al encolar procesos (opción 7). Recorre la cola de CPU comparando el ID de cada proceso con el ID buscado. Si encuentra un proceso con ese ID, devuelve **true**; si no, devuelve **false**.

```
// Verifica si un proceso con determinado ID ya está en la cola de CPU
bool procesoYaEnCola(string id) {
    NodoCola* temp = colaCPU;
    while (temp) {
        if (temp->id == id)
            return true;    // Ya está en la cola
        temp = temp->siguiente;
    }
    return false;    // No está en la cola
}
```

- **Función de búsqueda para la opción 4:** Esta función se usa como base para la opción 4 del sistema. Recorre la lista enlazada desde el primer nodo (**listaProcesos**), comparando el ID de cada proceso con el ID buscado. Si encuentra ese ID, devuelve un puntero a ese nodo; si no, devuelve **NULL**, indicando que el proceso no se encontró.

```
// Busca un proceso por su ID dentro de la lista enlazada de procesos
Proceso* buscarProcesoPorID(string id) {
    Proceso* actual = listaProcesos;
    while (actual) {
        if (actual->id == id)
            return actual;    // Proceso encontrado
        actual = actual->siguiente;
    }
    return NULL;    // No se encontró el proceso
}
```

- **Funciones para almacenar texto :** La función **guardarProcesosEnArchivo** se encarga de almacenar en un archivo de texto llamado "**procesos.txt**" todos los procesos de la lista enlazada **listaProcesos**. Recorre cada proceso y guarda su ID, nombre y prioridad en el archivo para conservar esta información de manera persistente.

Por otro lado, la función **cargarProcesosDesdeArchivo** lee los datos almacenados en "**procesos.txt**" para cargar los procesos previamente guardados y agregarlos nuevamente a la lista enlazada. Antes de insertar cada proceso, verifica que no haya duplicados buscando el ID en la lista.

```
// Guarda todos los procesos de la lista en un archivo de texto
void guardarProcesosEnArchivo() {
    ofstream archivo("procesos.txt");    // Abre el archivo para escritura
    Proceso* actual = listaProcesos;
    while (actual) {
        archivo << "ID: " << actual->id << ", "
                << "Nombre: " << actual->nombre << ", "
                << "Prioridad: " << actual->prioridad << "\n";
        actual = actual->siguiente;
    }
    archivo.close();    // Cierra el archivo
}
```



```

// Carga los procesos almacenados previamente en el archivo "procesos.txt"
void cargarProcesosDesdeArchivo() {
    ifstream archivo("procesos.txt"); // Abre el archivo para lectura
    string id, nombre;
    int prioridad;

    // Lee los datos línea por línea
    while (archivo >> id >> nombre >> prioridad) {
        // Verifica que no se dupliquen
        if (buscarProcesoPorID(id) == NULL) {
            Proceso* nuevo = new Proceso(id, nombre, prioridad);
            if (!listaProcesos) {
                listaProcesos = nuevo;
            } else {
                Proceso* temp = listaProcesos;
                while (temp->siguiente) temp = temp->siguiente;
                temp->siguiente = nuevo;
            }
        }
    }

    archivo.close(); // Cierra el archivo
}

```

3.1.2. Funciones del sistema

A. Insertar proceso:

Esta función corresponde a la opción 1 del sistema y permite al usuario insertar un nuevo proceso. Primero solicita el ID del proceso y verifica que no esté repetido mediante la función **buscarProcesoPorID**. Luego pide el nombre del proceso y valida que la prioridad esté entre 1 y 10. Si los datos son válidos, se crea un nuevo nodo nuevo con los valores ingresados. Si **listaProcesos** está vacía, se asigna directamente; de lo contrario, se recorre la lista con actual hasta el último nodo y se enlaza nuevo al final usando **actual->siguiente = nuevo**. Finalmente, los procesos actualizados se guardan en un archivo de texto mediante las funciones antes creadas.

```

// 1. Función para insertar un nuevo proceso ingresado por el usuario
void insertarProceso() {
    string id, nombre;
    int prioridad;

    cout << "Ingrese ID del proceso (entero): ";
    cin >> id;

    // Evita que se repita un proceso con el mismo ID
    if (buscarProcesoPorID(id)) {
        cout << "Ya existe un proceso con ese ID.\n";
        system("pause");
        system("CLS");
        return;
    }

    // Solicita datos del nuevo proceso
    cout << "Ingrese nombre del proceso: ";
    cin >> nombre;

    // Validamos Prioridad
    do {
        cout << "Ingrese prioridad (entero del 1 al 10): ";
        cin >> prioridad;

        if (prioridad < 1 || prioridad > 10) {
            cout << "\tError (entero del 1 al 10) .... \n";
        }
    } while (prioridad < 1 || prioridad > 10);
}

```

```
// Crea e inserta el proceso al final de la lista
Proceso* nuevo = new Proceso(id, nombre, prioridad);
if (!listaProcesos) {
    listaProcesos = nuevo;
} else {
    Proceso* actual = listaProcesos;
    while (actual->siguiente) {
        actual = actual->siguiente;
    }
    actual->siguiente = nuevo;
}

guardarProcesosEnArchivo(); // Guarda los datos actualizados en archivo
system("pause");
system("CLS");
```

B. Listar procesos:

Función que define al puntero actual apuntando al inicio de la lista (**listaProcesos**). El bucle recorre la lista mientras actual no sea NULL, procesando cada nodo. Luego, con **actual = actual->siguiente**, se avanza al siguiente proceso hasta llegar al final de la lista.

```
// 2. Lista todos los procesos existentes en la lista
void listarProcesos() {
    if (!listaProcesos) {
        cout << "No hay procesos registrados.\n";
    } else {
        Proceso* actual = listaProcesos;
        cout << "Lista de procesos:\n";
        while (actual) {
            cout << "ID: " << actual->id
                << " | Nombre: " << actual->nombre
                << " | Prioridad: " << actual->prioridad << "\n";
            actual = actual->siguiente;
        }
    }
    system("pause");
    system("CLS");
}
```

C. Eliminar procesos:

Esta función corresponde a la opción 3 del sistema. Permite eliminar un proceso ingresando su ID. Primero recorre la lista enlazada con dos punteros (**actual** y **anterior**) hasta encontrar el nodo con el ID indicado. Si no lo encuentra, muestra un mensaje y sale. Si lo encuentra, ajusta los punteros para eliminar el nodo: si es el primero, actualiza **listaProcesos**; si está en otra posición, enlaza anterior con el nodo **siguiente**. Luego libera la memoria, actualiza el archivo de texto y confirma la eliminación.

```
// 3. Elimina un proceso por su ID
void eliminarProceso() {
    string id;
    cout << "Ingrese el ID del proceso a eliminar: ";
    cin >> id;

    Proceso* actual = listaProcesos;
    Proceso* anterior = NULL;

    // Busca el proceso a eliminar
    while (actual && actual->id != id) {
        anterior = actual;
        actual = actual->siguiente;
    }

    // Si no lo encuentra
    if (!actual) {
        cout << "Proceso no encontrado.\n";
        system("pause");
        system("CLS");
        return;
    }

    // Elimina el nodo correspondiente
    if (!anterior) {
        listaProcesos = actual->siguiente;
    } else {
        anterior->siguiente = actual->siguiente;
    }

    delete actual; // Libera la memoria
    guardarProcesosEnArchivo(); // Actualiza el archivo
    cout << "Proceso eliminado correctamente.\n";
    system("pause");
    system("CLS");
}
}
```

D. Buscar procesos por ID:

Esta función corresponde a la opción 4 del sistema y permite buscar un proceso por su ID. Solicita al usuario ingresar el ID del proceso y utiliza la función **buscarProcesoPorID** para recorrer la lista enlazada y encontrar el nodo correspondiente. Si el proceso existe, muestra sus datos (ID, nombre y prioridad). Si no se encuentra, muestra un mensaje indicando que el proceso no fue encontrado. Finalmente, pausa la consola y limpia la pantalla para continuar con el menú.

```
4. Busca y muestra un proceso por ID
void buscarProceso() {
    string id;
    cout << "Ingrese el ID del proceso a buscar: ";
    cin >> id;

    Proceso* resultado = buscarProcesoPorID(id); //Buscar el proceso en la lista
    if (resultado) {
        cout << "Proceso encontrado:\n"; // muestra los datos en caso de encontrarlo
        cout << "ID: " << resultado->id
        << " | Nombre: " << resultado->nombre
        << " | Prioridad: " << resultado->prioridad << "\n"; //datos
    } else {
        cout << "Proceso no encontrado.\n"; // muestra el siguiente mensaje si no encuentra el proceso
    }
    system("pause");
    system("CLS");
}
}
```

E. Modificar prioridad de un proceso:

La función **modificarPrioridad** utiliza un puntero llamado **p** para almacenar la dirección del nodo del proceso buscado mediante **buscarProcesoPorID(id)**, evitando recorrer nuevamente la lista. La condición **if (!p)** verifica si el proceso no fue encontrado (puntero nulo), mostrando un mensaje y terminando la función. Si el proceso existe, se muestra su prioridad actual y se solicita una nueva prioridad al usuario, validando que esté entre 1 y 10. Finalmente, se actualiza el campo prioridad del proceso con el nuevo valor y se guardan los cambios en el archivo de texto.

```
// 5. Permite cambiar la prioridad de un proceso ya registrado
void modificarPrioridad() {
    string id;
    cout << "Ingrese el ID del proceso: "; //busca el proceso mediante ID
    cin >> id;

    Proceso* p = buscarProcesoPorID(id);
    if (!p) {
        cout << "Proceso no encontrado.\n"; //si no se ubica sale un mensaje
        system("pause");
        system("CLS");
        return;
    }

    int nuevaPrioridad;
    cout << "Prioridad actual: " << p->prioridad << "\n";

    //Validamos Prioridad
    do {
        cout << "Ingrese nueva prioridad: ";
        cin >> nuevaPrioridad; // solicita nueva prioridad

        if (nuevaPrioridad<1 || nuevaPrioridad>10) {
            cout<<"\tError (entero del 1 al 10) .... \n"; // la prioridad tendra que ser entero
        }

    } while (nuevaPrioridad<1 || nuevaPrioridad>10);

    p->prioridad = nuevaPrioridad; // Asigna prioridad
    guardarProcesosEnArchivo(); // Actualiza archivo
    cout << "Prioridad modificada correctamente.\n";
    system("pause");
    system("CLS");
}
```

F. Ordenamiento de los procesos por prioridad:

La función **ordenarProcesos** ordena la lista enlazada **listaProcesos** para que los procesos con mayor prioridad estén primero. Primero verifica si la lista tiene al menos dos procesos para ordenar. Usa un algoritmo de burbuja adaptado a listas enlazadas, recorriendo nodos con un puntero **actual** y otro **anterior**. Compara la prioridad de **actual** y su siguiente; si **actual** tiene menor prioridad, intercambia ambos nodos actualizando los punteros correspondientes. Este ciclo se repite hasta que no haya más intercambios. Al finalizar, guarda la lista ordenada en el archivo y muestra la lista actualizada.

```

// 6. Ordena los procesos por prioridad de forma descendente (mayor prioridad primero)
void ordenarProcesos() {
    if (!listaProcesos || !listaProcesos->siguiente) {
        cout << "No hay suficientes procesos para ordenar.\n";
        system("pause");
        system("CLS");
        return;
    }

    // Algoritmo de burbuja adaptado a Lista enlazada
    bool intercambiado;
    do {
        intercambiado = false;
        Proceso* actual = listaProcesos;
        Proceso* anterior = NULL;

        while (actual && actual->siguiente) {
            Proceso* siguiente = actual->siguiente;
            // Compara prioridades para intercambiar
            if (actual->prioridad < siguiente->prioridad) {
                if (anterior) {
                    anterior->siguiente = siguiente;
                } else {
                    listaProcesos = siguiente;
                }
                actual->siguiente = siguiente->siguiente;
                siguiente->siguiente = actual;

                intercambiado = true;
            } else {
                anterior = actual;
                actual = actual->siguiente;
            }
        }
    } while (intercambiado);

    guardarProcesosEnArchivo(); // Guarda el nuevo orden
    cout << "Procesos ordenados por prioridad (Mayor -> Menor).\n";
    listarProcesos(); // mostramos la lista ordenada
    system("pause");
    system("CLS");
}

```

G. Encolar procesos (Cola CPU):

La función **encolarCPU** recorre la lista de procesos **listaProcesos** y añade a la cola de CPU (**colaCPU**) aquellos procesos que aún no están encolados, verificando con **procesoYaEnCola**. Para cada proceso, crea un nuevo nodo **NodoCola** con sus datos. La condición **if (!colaCPU || nuevo->prioridad < colaCPU->prioridad)** inserta el nuevo nodo al inicio si la cola está vacía o si su prioridad es mayor (es decir, un valor numérico menor), manteniendo el orden por prioridad. Al final, muestra un mensaje confirmando que los procesos fueron encolados.

```

// 7. Encola todos los procesos a la cola de CPU, ordenados por prioridad
void encolarCPU() {
    Proceso* actual = listaProcesos;
    while (actual) {
        if (!procesoYaEnCola(actual->id)) {
            NodoCola* nuevo = new NodoCola;
            nuevo->id = actual->id;
            nuevo->nombre = actual->nombre;
            nuevo->prioridad = actual->prioridad;
            nuevo->siguiente = NULL;

            // Inserta en la posición correcta según la prioridad
            if (!colaCPU || nuevo->prioridad < colaCPU->prioridad) {
                nuevo->siguiente = colaCPU;
                colaCPU = nuevo;
            } else {
                NodoCola* temp = colaCPU;
                while (temp->siguiente && temp->siguiente->prioridad <= nuevo->prioridad) {
                    temp = temp->siguiente;
                }
                nuevo->siguiente = temp->siguiente;
                temp->siguiente = nuevo;
            }
        }
        actual = actual->siguiente;
    }

    cout << "Procesos encolados a la CPU segun prioridad.\n";
    system("pause");
    system("CLS");
}

```

H. Desencolar procesos (ejecutar):

La función **ejecutarCPU** se encarga de desencolar el proceso con mayor prioridad (el de menor número). Primero verifica si la cola de procesos (**colaCPU**) está vacía; si es así, muestra el mensaje "No hay procesos en la cola". Si hay procesos, usa un puntero temporal (**temp**) para apuntar al primer nodo de la cola, que es el proceso con mayor prioridad. Luego actualiza el inicio de la cola (**colaCPU**) para que apunte al siguiente proceso. Después muestra en pantalla el ID, nombre y prioridad del proceso almacenado en **temp**, y finalmente libera la memoria con `delete temp` para eliminar ese proceso de la cola.

```
8. Desencola y ejecuta el proceso con mayor prioridad
void ejecutarCPU() {
    if (!colaCPU) {
        cout << "No hay procesos en la cola.\n";
        system("pause");
        system("CLS");
        return;
    }

    NodoCola* temp = colaCPU;
    colaCPU = colaCPU->siguiente;
    cout << "Ejecutando proceso:\n"; // muestra el proceso
    cout << "ID: " << temp->id << " | Nombre: " << temp->nombre << " | Prioridad: " << temp->prioridad << "\n";
    delete temp; // Libera memoria del proceso ejecutado
    system("pause");
    system("CLS");
}
```

I. Mostrar Cola CPU:

La función **mostrarColaCPU** muestra los procesos en la cola de CPU (**colaCPU**) ordenados por prioridad. Primero verifica si la cola está vacía; si es así, muestra el mensaje "La cola está vacía". Si no, recorre la cola con un puntero temporal (**temp**) que apunta al primer nodo. Mientras **temp** no sea nulo, imprime el ID, nombre y prioridad del proceso actual, y avanza al siguiente nodo hasta mostrar todos los procesos en la cola.

```
9. Muestra todos los procesos en la cola de CPU
void mostrarColaCPU() {
    if (!colaCPU) {
        cout << "La cola esta vacia.\n";
    } else {
        cout << "Cola actual (ordenada por prioridad):\n";
        NodoCola* temp = colaCPU;
        while (temp) {
            cout << "ID: " << temp->id << " | Nombre: " << temp->nombre << " | Prioridad: " << temp->prioridad << "\n";
            temp = temp->siguiente;
        }
    }
    system("pause");
    system("CLS");
}
```


J. Asignar memoria (Push):

La función **pushMemoria** simula la asignación de memoria a un proceso utilizando una pila. Primero, solicita al usuario ingresar el ID del proceso al que se le asignará memoria. Luego crea un nuevo nodo de tipo **NodoMemoria**, asigna el ID ingresado a ese nodo y lo inserta al inicio de la pila (pilaMemoria), actualizando el puntero para que apunte al nuevo nodo. Finalmente, muestra un mensaje confirmando que la memoria fue asignada correctamente.

```
// 10. Simula la asignación de memoria (Push en la pila)
void pushMemoria() {
    string id;
    cout << "Ingrese el ID del proceso para asignar memoria: ";
    cin >> id;

    NodoMemoria* nuevo = new NodoMemoria;
    nuevo->procesoID = id;
    nuevo->siguiente = pilaMemoria;
    pilaMemoria = nuevo;

    cout << "Memoria asignada correctamente al proceso.\n";
    system("pause");
    system("CLS");
}
```

K. Liberación de memoria (Pop):

La función **popMemoria** libera la memoria del proceso que fue el último en entrar (LIFO) utilizando una pila. Primero verifica si la pila de memoria (pilaMemoria) está vacía; si es así, muestra un mensaje indicando que no hay memoria para liberar. Si hay nodos, muestra el ID del proceso cuya memoria será liberada, luego usa un puntero temporal para almacenar el nodo superior de la pila, actualiza el puntero de la pila para que apunte al siguiente nodo y, finalmente, elimina el nodo superior, liberando así la memoria asignada a ese proceso.

```
// 11. Simula la liberación de memoria (Pop en la pila)
void popMemoria() {
    if (!pilaMemoria) {
        cout << "La memoria ya esta vacia.\n";
    } else {
        cout << "Liberando memoria del proceso ID: " << pilaMemoria->procesoID << "\n";
        NodoMemoria* temp = pilaMemoria;
        pilaMemoria = pilaMemoria->siguiente;
        delete temp;
    }
    system("pause");
    system("CLS");
}
```

K. Mostrar estado de memoria:

La función **mostrarMemoria** muestra el estado actual de la pila de memoria. Primero verifica si la pila (pilaMemoria) está vacía; si es así, muestra el mensaje "La memoria está vacía". Si no, recorre la pila utilizando un puntero temporal que comienza en el nodo superior. Mientras haya nodos, imprime la posición en la pila y el ID del proceso asociado a cada nodo, avanzando hasta mostrar todos los procesos almacenados en la pila.

```
12. Muestra el estado actual de la pila de memoria
void mostrarMemoria() {
    if (!pilaMemoria) {
        cout << "La memoria esta vacia.\n";
    } else {
        NodoMemoria* temp = pilaMemoria;
        int pos = 1;
        while (temp) {
            cout << "Posicion " << pos++ << ": " << temp->procesoID << "\n";
            temp = temp->siguiente;
        }
    }
    system("pause");
    system("CLS");
}
```

3.1.3. Menú del sistema

La función main es el menú con trece opciones para gestionar procesos y memoria. Al inicio, se llama la función **cargarProcesosDesdeArchivo** que contiene al archivo de texto "procesos.txt" y se limpia ese archivo para empezar con un estado nuevo.

Luego, dentro de un bucle **do-while**, el programa presenta el menú al usuario y solicita que ingrese una opción. Para asegurar que la opción sea válida, se usa otro bucle **do-while** que verifica que el número ingresado esté entre 1 y 13. Si el usuario introduce un valor fuera de ese rango, se muestra un mensaje de error.

Una vez que la opción es validada, se utiliza una estructura **switch** para ejecutar la función correspondiente a la elección del usuario. Esto incluye las 13 opciones del programa. El ciclo se repite hasta que el usuario selecciona la opción 13, que es salir del programa. En ese momento, se muestra un mensaje de despedida y el programa termina su ejecución.


```
// ===== FUNCION PRINCIPAL (MAIN) =====
int main() {
    cargarProcesosDesdeArchivo(); // Carga procesos guardados previamente
    ofstream limpiar("procesos.txt"); // Limpia el archivo (opcional)
    limpiar.close();

    int opcion;

    do {
        // Menú de opciones
        cout << "\n===== GESTOR DE PROCESOS Y MEMORIA =====\n";
        cout << "1. Insertar proceso\n";
        cout << "2. Listar procesos (sin ordenar)\n";
        cout << "3. Eliminar proceso por ID\n";
        cout << "4. Buscar proceso por ID\n";
        cout << "5. Modificar prioridad de un proceso\n";
        cout << "6. Ordenar procesos por prioridad (Mayor -> Menor)\n";
        cout << "7. Encolar proceso a la cola CPU\n";
        cout << "8. Ejecutar (desencolar) proceso con mayor prioridad\n";
        cout << "9. Visualizar cola actual (ordenada)\n";
        cout << "10. Asignar memoria (push)\n";
        cout << "11. Liberar memoria (pop)\n";
        cout << "12. Ver estado actual de memoria\n";
        cout << "13. Salir\n";
    }
```

```
// Validación de opción ingresada
do {
    cout << "Seleccione una opcion: ";
    cin >> opcion;
    if (opcion < 1 || opcion > 13) {
        cout << "Opcion no valida.\n";
        system("pause");
        system("CLS");
    }
} while (opcion < 1 || opcion > 13);

// Ejecución según opción
switch (opcion) {
    case 1: insertarProceso(); break;
    case 2: listarProcesos(); break;
    case 3: eliminarProceso(); break;
    case 4: buscarProceso(); break;
    case 5: modificarPrioridad(); break;
    case 6: ordenarProcesos(); break;
    case 7: encolarCPU(); break;
    case 8: ejecutarCPU(); break;
    case 9: mostrarColaCPU(); break;
    case 10: pushMemoria(); break;
    case 11: popMemoria(); break;
    case 12: mostrarMemoria(); break;
    case 13: cout << "Saliendo...\n"; break;
}
} while (opcion != 13);
```

3.2. Pruebas de validación

A. Insertar proceso:

La validación fue parcialmente exitosa. Se esperaba que el ID fuera un número entero, el nombre una cadena de texto (*string*), y la prioridad un entero entre 1 y 10. El sistema **valida correctamente la prioridad y permite insertar datos del proceso** cuando se respetan los tipos adecuados.

```
Seleccione una opcion: 1
Ingrese ID del proceso (entero): 3
Ingrese nombre del proceso: Firefox
Ingrese prioridad (entero del 1 al 10): 4
Presione una tecla para continuar . . . |
```

Sin embargo, se detectó que al ingresar valores no numéricos en el campo ID (que debe ser *int*), el programa muestra un mensaje de error y puede comportarse de forma inesperada.

```
Ingrese prioridad (entero del 1 al 10):      Error (entero del 1 al 10) ....
Ingrese prioridad (entero del 1 al 10):      Error (entero del 1 al 10) ....
Ingrese prioridad (entero del 1 al 10):      Error (entero del 1 al 10) ....
Ingrese prioridad (entero del 1 al 10):      Error (entero del 1 al 10) ....
Ingrese prioridad (entero del 1 al 10):      Error (entero del 1 al 10) ....
```

B. Listar procesos:

La validación fue exitosa. Se esperaba que se **creará una lista enlazada** con los procesos insertados mediante la opción 1, y que en caso de no haberse insertado ningún proceso, se **mostrará un mensaje indicando que no hay procesos**. El sistema cumple con ambos comportamientos de forma adecuada y muestra la lista correctamente.

```
Seleccione una opcion: 2
No hay procesos registrados.
Presione una tecla para continuar . . . |
```

```
Seleccione una opcion: 2
Lista de procesos:
ID: 3 | Nombre: Firefox | Prioridad: 4
ID: 2 | Nombre: Google | Prioridad: 10
ID: 5 | Nombre: Word | Prioridad: 1
Presione una tecla para continuar . . . |
```

C. Eliminar procesos:

La validación fue exitosa. Se esperaba que se **pudiera eliminar un proceso ingresado previamente mediante su ID**, y que en caso de no existir un proceso

con ese ID, se mostrará un mensaje informando que no fue encontrado. El sistema respondió correctamente en ambos casos, eliminando el proceso cuando correspondía y mostrando el mensaje adecuado cuando no existía.

```
Seleccione una opcion: 3
Ingrese el ID del proceso a eliminar: 4
Proceso no encontrado.
Presione una tecla para continuar . . . |
```

```
Seleccione una opcion: 3
Ingrese el ID del proceso a eliminar: 5
Proceso eliminado correctamente.
Presione una tecla para continuar . . . |
```

D. Buscar procesos por ID:

La validación fue exitosa. Se esperaba que se **pudiera buscar un proceso previamente insertado mediante su ID**, y que en caso de no existir dicho proceso, que se muestre un mensaje indicando que no fue encontrado. En ambas situaciones, el sistema respondió de forma adecuada, mostrando los datos del proceso cuando existía y el mensaje correcto cuando no.

```
Seleccione una opcion: 4
Ingrese el ID del proceso a buscar: 4
Proceso no encontrado.
Presione una tecla para continuar . . . |
```

```
Seleccione una opcion: 4
Ingrese el ID del proceso a buscar: 3
Proceso encontrado:
ID: 3 | Nombre: Firefox | Prioridad: 4
Presione una tecla para continuar . . . |
```

E. Modificar prioridad de un proceso:

La validación fue exitosa. El objetivo era que el sistema permitiera **modificar la prioridad de un proceso, ingresando su ID**. También se esperaba que, si el ID no coincidía con ningún proceso, se mostrará un mensaje indicando que no fue encontrado. En ambos casos, el sistema funcionó correctamente: permite

editar la prioridad cuando el proceso existía y mostró el mensaje correspondiente cuando no se encontró el ID ingresado.

```
Seleccione una opcion: 5
Ingrese el ID del proceso: 4
Proceso no encontrado.
Presione una tecla para continuar . . . |
```

```
Seleccione una opcion: 5
Ingrese el ID del proceso: 2
Prioridad actual: 10
Ingrese nueva prioridad: 7
Prioridad modificada correctamente.
Presione una tecla para continuar . . . |
```

F. Ordenamiento de los procesos por prioridad:

La validación fue exitosa. El objetivo era verificar que el sistema ordenará correctamente los procesos según su prioridad de forma descendente. Se esperaba que los procesos con mayor prioridad (es decir, con menor número) aparecieran al final de la lista. El sistema cumplió con lo esperado, reordenando los procesos.

```
Seleccione una opcion: 6
Procesos ordenados por prioridad (Mayor -> Menor).
Lista de procesos:
ID: 2 | Nombre: Google | Prioridad: 7
ID: 3 | Nombre: Firefox | Prioridad: 4
ID: 7 | Nombre: VCode | Prioridad: 2
Presione una tecla para continuar . . . |
```

G. Encolar procesos (Cola CPU):

La validación fue exitosa. El objetivo era encolar correctamente todos los procesos previamente insertados mediante la opción 1. El sistema cumplió con lo esperado, agregando los procesos a la cola de CPU según su prioridad. Este comportamiento se pudo verificar posteriormente usando la opción 9 para visualizar la cola actual.

```
Seleccione una opcion: 7
Procesos encolados a la CPU segun prioridad.
Presione una tecla para continuar . . . |
```

H. Desencolar procesos (ejecutar):

La validación fue exitosa. Se esperaba que el sistema desencolara y ejecutara el proceso con mayor prioridad (es decir, el de menor número) correctamente. Además, debía mostrar un mensaje indicando que no hay procesos cuando la cola estuviera vacía. En ambos casos, el sistema respondió de forma adecuada

```
Seleccione una opcion: 8
No hay procesos en la cola.
Presione una tecla para continuar . . . |
```

```
Seleccione una opcion: 8
Ejecutando proceso:
ID: 7 | Nombre: VCode | Prioridad: 2
Presione una tecla para continuar . . . |
```

I. Mostrar Cola CPU:

La validación fue exitosa. Se esperaba que el sistema mostrará correctamente el estado actual de la cola de procesos, ordenada por prioridad, con el proceso de mayor prioridad (el de menor número) en la primera posición. Además, en caso de no haber procesos encolados, el sistema debería mostrar un mensaje indicando esta situación. El sistema cumplió con este comportamiento.

```
Seleccione una opcion: 9
La cola esta vacia.
Presione una tecla para continuar . . . |
```

```
Seleccione una opcion: 9
Cola actual (ordenada por prioridad):
ID: 3 | Nombre: Firefox | Prioridad: 4
ID: 2 | Nombre: Google | Prioridad: 7
Presione una tecla para continuar . . . |
```

J. Asignar memoria (Push):

La validación fue parcialmente exitosa. Se esperaba que el sistema permitiera asignar memoria a un proceso específico mediante su ID y que mostrará un mensaje de error si el proceso no existía. En cuanto a la asignación de memoria, el sistema funcionó correctamente, permitiendo asignar memoria a los procesos indicados.

```
Seleccione una opcion: 10
Ingrese el ID del proceso para asignar memoria: 7
Memoria asignada correctamente al proceso.
Presione una tecla para continuar . . . |
```

Sin embargo, no se implementó una validación para evitar asignar memoria a procesos cuyo ID no existe.

```
Seleccione una opcion: 10
Ingrese el ID del proceso para asignar memoria: 1
Memoria asignada correctamente al proceso.
Presione una tecla para continuar . . . |
```

K. Liberación de memoria (Pop):

La validación fue exitosa. El objetivo era que el sistema liberará la memoria asignada al último proceso ingresado, siguiendo la lógica LIFO propia de las pilas. El sistema cumplió correctamente con este comportamiento.

```
Seleccione una opcion: 11
Liberando memoria del proceso ID: 7
Presione una tecla para continuar . . . |
```

L. Mostrar estado de memoria:

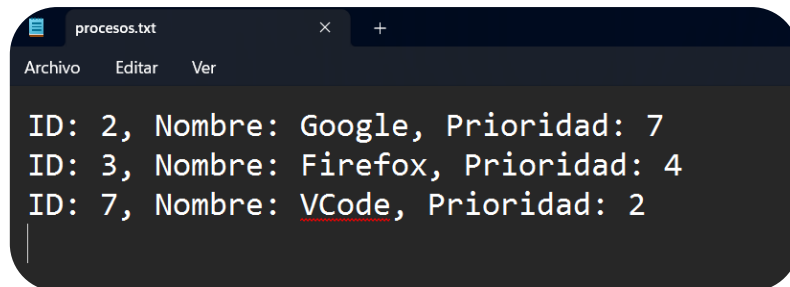
La validación fue exitosa. El objetivo era que el sistema mostrara el estado actual de la pila de memoria, indicando la posición de cada proceso según el orden de asignación. Se esperaba que el proceso más reciente apareciera en la parte superior, siguiendo la lógica LIFO. Además, en caso de no haber procesos con memoria asignada, se debía mostrar un mensaje que indicará esta situación. El sistema cumplió con estas expectativas correctamente.

```
Seleccione una opcion: 12
La memoria esta vacia.
Presione una tecla para continuar . . . |
```

```
Seleccione una opcion: 12
Posicion 1: 3
Posicion 2: 2
Presione una tecla para continuar . . . |
```

M. Persistencia de datos:

La validación fue exitosa. El objetivo era que el archivo de texto refleja correctamente el estado de los procesos insertados en el sistema. El sistema cumplió con lo esperado, generando el archivo con la información correspondiente.



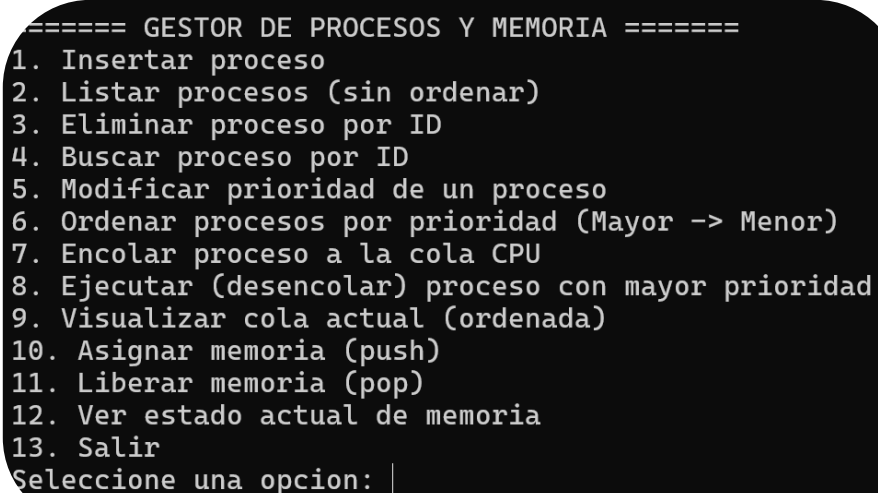
```
procesos.txt
Archivo  Editar  Ver

ID: 2, Nombre: Google, Prioridad: 7
ID: 3, Nombre: Firefox, Prioridad: 4
ID: 7, Nombre: VCode, Prioridad: 2
```

3.3. Manual de Usuario

Este sistema está diseñado para simular un administrador de procesos y memoria. A través de él, podrás ingresar nuevos procesos, eliminarlos, modificarlos y ordenarlos utilizando listas o colas. Además, tendrás la opción de gestionar la asignación y liberación de memoria mediante una estructura tipo pila.

Al iniciar, accederás a un menú con 13 opciones, cada una acompañada de una descripción clara e intuitiva para facilitar su uso.



```
===== GESTOR DE PROCESOS Y MEMORIA =====
1. Insertar proceso
2. Listar procesos (sin ordenar)
3. Eliminar proceso por ID
4. Buscar proceso por ID
5. Modificar prioridad de un proceso
6. Ordenar procesos por prioridad (Mayor -> Menor)
7. Encolar proceso a la cola CPU
8. Ejecutar (desencolar) proceso con mayor prioridad
9. Visualizar cola actual (ordenada)
10. Asignar memoria (push)
11. Liberar memoria (pop)
12. Ver estado actual de memoria
13. Salir
Seleccione una opcion: |
```

A. Insertar proceso:

La opción 1 es el punto de partida donde se ingresan nuevos procesos. Se solicitará ingresar un ID numérico entero para el proceso, un nombre

descriptivo y un nivel de prioridad, donde 1 representa la máxima prioridad y 10 la mínima.

```
Seleccione una opcion: 1
Ingrese ID del proceso: 1301
Ingrese nombre del proceso: Editar
Ingrese prioridad (entero): 3
Presione una tecla para continuar . . .
```

B. Listar procesos (sin ordenar):

Una vez ingresado varios procesos, se puede corroborar si los datos se registraron correctamente, la opción 2 te brindará la lista pero de manera desordenada, esto quiere decir que estará según el orden que se ingresaron los procesos.

```
Seleccione una opcion: 2
Lista de procesos:
ID: 1301 | Nombre: Editar | Prioridad: 3
ID: 1302 | Nombre: Dividir | Prioridad: 2
ID: 1303 | Nombre: Activar | Prioridad: 1
ID: 1304 | Nombre: Antivirus | Prioridad: 2
Presione una tecla para continuar . . .
```

C. Eliminar proceso por ID:

Se solicitará ingresar el número de ID del proceso que se desea eliminar. Por ejemplo, si se ingresa la ID 1301 correspondiente al proceso llamado "Editar", el sistema procederá a eliminarlo. Además, se mostrará un mensaje que confirmará si la eliminación fue exitosa o si el proceso no se encontró.

```
Seleccione una opcion: 3
Ingrese el ID del proceso a eliminar: 1301
Proceso eliminado correctamente.
Presione una tecla para continuar . . .
```

D. Buscar proceso por ID:

Con esta opción se podrá encontrar cualquier proceso ingresado mediante su código ID (número entero), donde una vez encontrada te mostrará todas sus características como el nombre y la prioridad.


```
Seleccione una opcion: 4
Ingrese el ID del proceso a buscar: 1303
Proceso encontrado:
ID: 1303 | Nombre: Activar | Prioridad: 1
Presione una tecla para continuar . . . |
```

E. Modificar prioridad de un proceso:

Será posible modificar el número de prioridad asignado a un proceso. Para ello, se solicitará ingresar la ID del proceso que se desea modificar. A continuación, se mostrará la prioridad actual registrada, y se pedirá ingresar el nuevo número de prioridad (un valor entero) que se desea asignar.

```
Seleccione una opcion: 5
Ingrese el ID del proceso: 1302
Prioridad actual: 2
Ingrese nueva prioridad: 3
Prioridad modificada correctamente.
Presione una tecla para continuar . . . |
```

F. Ordenar procesos por prioridad (De mayor a menor):

Esta opción ordenará nuestros procesos de mayor a menor según la prioridad que tienen. Esto sirve mucho en algunos casos para ver nuestros procesos con mayor requerimiento o atención.

```
Seleccione una opcion: 6
Procesos ordenados por prioridad (Mayor -> Menor).
Presione una tecla para continuar . . . |
```

G. Encolar proceso a la cola CPU:

En esta opción se encolarán todos los procesos registrados en una nueva cola CPU, organizados según su prioridad (primero los de prioridad 1, luego 2, 3, y así sucesivamente). La cola seguirá el principio FIFO (First-In, First-Out), lo que significa que los procesos se atenderán en el orden en que fueron agregados.

```
Seleccione una opcion: 7
Procesos encolados a la CPU segun prioridad.
Presione una tecla para continuar . . . |
```

H. Ejecutar (desencolar) proceso con mayor prioridad:

Esta opción ejecuta los procesos de la cola comenzando por aquellos con prioridad 1, que son los de mayor prioridad y están al frente de la cola. Al ejecutarlos, estos procesos se eliminan de la cola.

```
Seleccione una opcion: 8
Ejecutando proceso:
ID: 1303 | Nombre: Activar | Prioridad: 1
Presione una tecla para continuar . . .
```

I. Visualizar cola actual (ordenada):

Esta opción es para corroborar nuestra cola de CPU actual, y ver cómo va quedando todo. El ordenamiento es según la prioridad.

```
Seleccione una opcion: 9
Cola actual (ordenada por prioridad):
ID: 1304 | Nombre: Antivirus | Prioridad: 2
ID: 1302 | Nombre: Dividir | Prioridad: 3
Presione una tecla para continuar . . .
```

J. Asignar memoria (push):

En esta opción se solicitará el ID de los procesos previamente registrados para asignarlos a una nueva estructura tipo pila denominada Memoria. Esta operación tiene como finalidad organizar los procesos bajo el principio LIFO (Last In, First Out), comúnmente utilizado para una gestión más ordenada y eficiente de procesos.

```
Seleccione una opcion: 10
Ingrese el ID del proceso para asignar memoria: 1304
Memoria asignada correctamente al proceso.
Presione una tecla para continuar . . . |
```

```
Seleccione una opcion: 10
Ingrese el ID del proceso para asignar memoria: 1302
Memoria asignada correctamente al proceso.
Presione una tecla para continuar . . . |
```

K. Liberar memoria (pop):

En esta opción se cumplirá con el principio de LIFO de una pila (Last-In, First-Out), es decir que eliminará o ejecutará el proceso que se ingresó

último, en nuestro ejemplo, el último ID en registrar fue el ID 1302, por ende es el primero en salir.

```
Seleccione una opcion: 11
Liberando memoria del proceso ID: 1302
Presione una tecla para continuar . . . |
```

L. Ver estado de memoria:

Se mostrará el estado actual de la memoria, reflejando los procesos asignados mediante sus respectivos ID. Estos están organizados siguiendo el orden de una pila, es decir, el proceso más reciente ingresado aparecerá en la parte superior.

```
Seleccione una opcion: 12
Posicion 1: 1310
Posicion 2: 1306
Posicion 3: 1305
Posicion 4: 1304
Presione una tecla para continuar . . . |
```

M. Salir:

Se finalizará el programa

```
Seleccione una opcion: 13
Saliendo...
-----
```

RECOMENDACIONES DE USO:

- Al insertar los IDs asegurarse que no se repitan y sea de forma consecutiva (1-2-3; 1200-1300-1400; 1201-1202-1203).
- Ingresar números válidos a la prioridad es decir valores enteros.

Capítulo 4: Evidencias de Trabajo en Equipo

4.1. Control de Versiones

4.1.1. Registro de commits

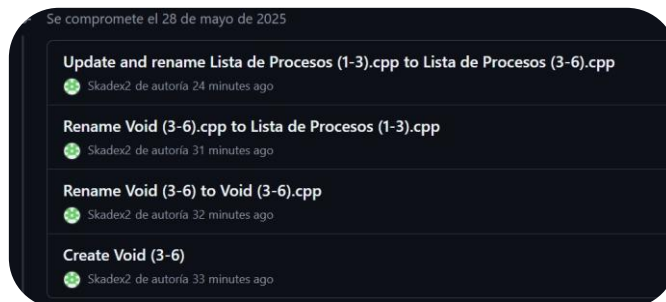
A. Primeras Actualizaciones:

El integrante declaró las estructuras necesarias para representar procesos, memoria y cola de CPU, además implementó funciones para insertar y listar procesos en una lista enlazada, permitiendo simular un manejo básico de procesos.



Integrante: Mijael Joseph Bejarano Miche

El integrante implementó funciones clave para la gestión de procesos, incluyendo la eliminación, búsqueda, modificación de prioridad y ordenamiento de procesos según su prioridad, permitiendo un control más completo y dinámico de la lista de procesos.



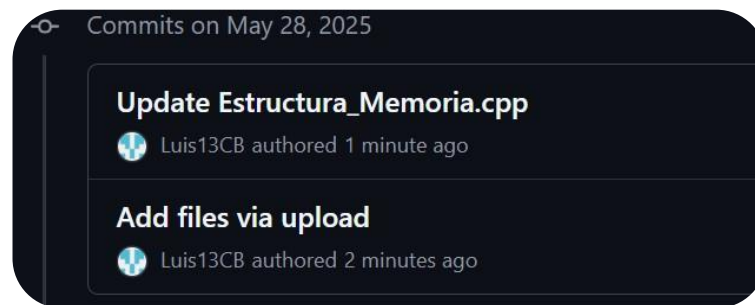
Integrante: Damián Javier Lopez Naula

El integrante desarrolló funciones para encolar procesos en una cola de prioridad y ejecutarlos según orden, además de mostrar el estado de la cola. Aprovechó las primeras funciones para simular de forma dinámica la gestión y ejecución de procesos en la CPU.



Integrante: Nelinho Capcha Hidalgo

El integrante implementó la gestión de memoria mediante una pila, desarrollando funciones para asignar(push), liberar(pop) y mostrar el estado actual de la memoria simulando así el manejo de memoria dinámica para procesos.



Integrante: Luis Enrique Cueva Barrera

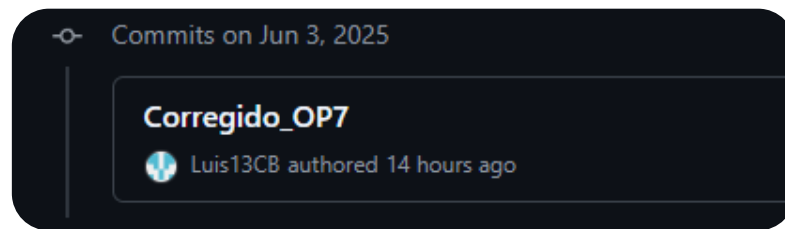
B. Segundas Actualizaciones

El integrante corrigió la opción 6 para que la lista se muestre ordenada de mayor a menor según la prioridad, y agregó una validación en la opción 1 para que las prioridades estén dentro de un rango de 1 a 10.



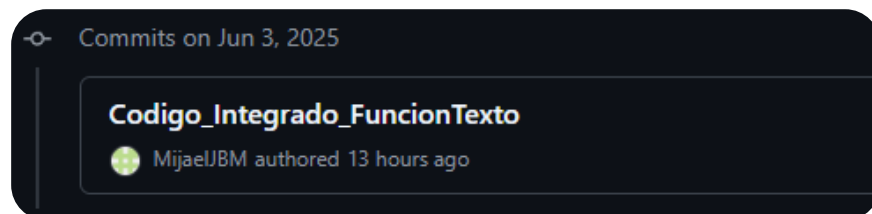
Integrante: Nelinho Capcha Hidalgo

El integrante corrigió la opción número 7 del menú en el código para evitar la duplicación al ejecutar los procesos más de una vez.



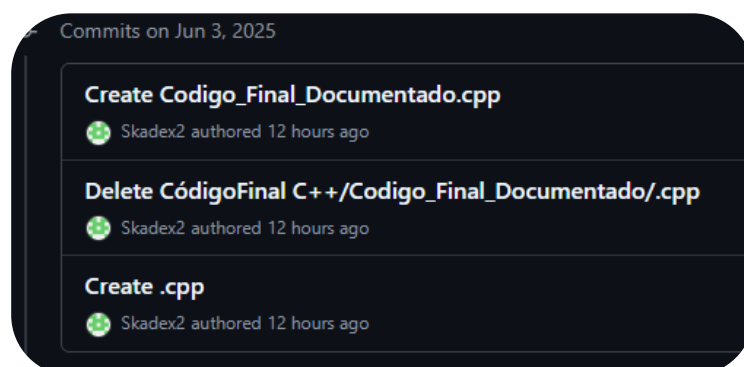
Integrante: Luis Enrique Cueva Barrera

El integrante desarrolló la función 'texto', que guarda los procesos en formato de texto y los carga utilizando listas enlazadas. Además, integró las correcciones de sus compañeros para obtener el código final.



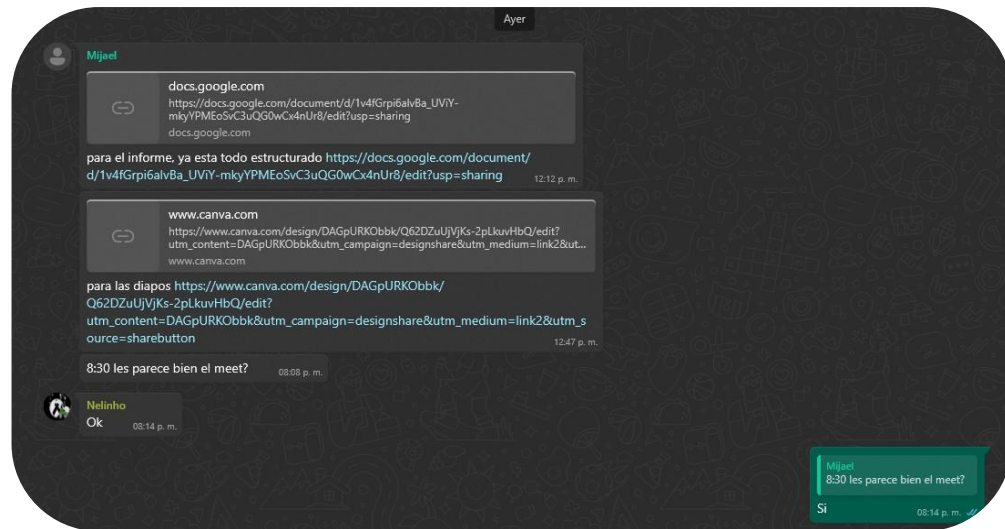
Integrante: Mijael Joseph Bejarano Miche

El integrante documentó el código añadiendo comentarios explicativos en las distintas secciones del programa, en las estructuras, funciones y lógicas con el fin de mejorar la comprensión del código.

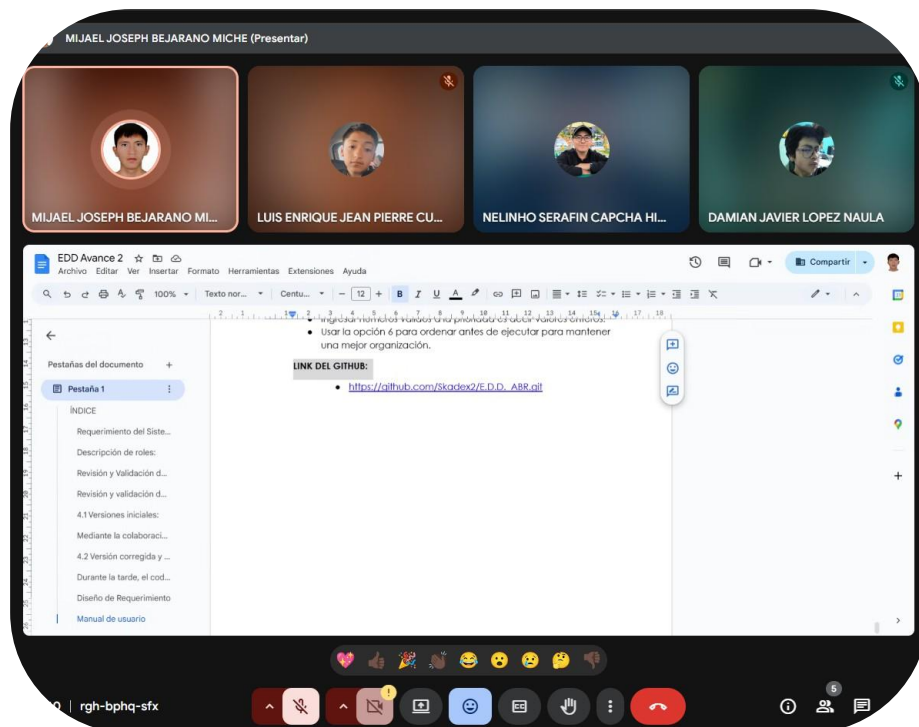


Integrante: Damián Javier Lopez Naula

4.1.2. Evidencias adicionales



Se usó el medio de “WhatsApp” para tener una mejor comunicación y coordinación en caso de que exista alguna duda con el desarrollo del trabajo.



Además, se usó la herramienta de Google Docs y Google Meet para realizar el trabajo correctamente sin ningún problema.

4.2. Plan de Trabajo y Roles Asignados

4.2.1. Descripción del rol de cada miembro por semana

A. Primera Semana:

a) Supervisor / Analista de Requerimientos (Bejarano Mijael):

Se encargó de definir la descripción y propósito del sistema, así como de establecer los requerimientos funcionales, no funcionales y los detalles técnicos del proyecto. Además, asumió el rol de supervisor, verificando y gestionando que los demás integrantes cumplieran con sus funciones asignadas, corrigiendo detalles del informe y código.

b) Codificador 1 (Cueva Luis):

- Implementó las funciones para registrar, buscar, eliminar y modificar procesos.
- Simuló listas enlazadas mediante arreglos en Pseint para representar los procesos.
- Ordenó los procesos según su prioridad.
- Entregó sus avances al supervisor/documentador para revisión y corrección de errores.

c) Codificador 2 (Capcha Nelinho):

- Desarrolló la cola de prioridad para simular la ejecución de procesos en la CPU.
- Implementó una pila para la asignación y liberación de memoria.
- Compartió sus versiones de código con el supervisor/documentador para validación y mejoras.

d) Documentador (López Damián):

- Describe los roles asignados a cada integrante del equipo y ejecutar entendiendo el código para luego realizar alguna observación a los codificadores.
- Documenta explicando los errores encontrados en el código y genera manuales de usuario y diagramas.
- Revisa cada módulo entregado por los codificadores.

B. Segunda Semana:

Para esta semana, el grupo se enfocó en la creación del sistema en C++. Para ello, se utilizó el pseudocódigo previamente desarrollado en PSeInt como base para estructurar el código. De este modo, todo el grupo asumió el rol de codificador, dividiéndolos por secciones del programa. Además, cada integrante asumió distintas labores para la realización y revisión del proyecto, a continuación, explicadas:

a) Coordinador/ Analista de código (Bejarano Mijael):

- Se encargó de coordinar las actividades y distribuir los roles y tareas entre los demás miembros del grupo.
- Como analista de código, en conjunto con el equipo, decidió crear las estructuras iniciales en C++, estableciendo también el uso de funciones *Void* para facilitar su integración mediante un menú final.
- En el rol de codificador, además de las estructuras mencionadas anteriormente, creó las dos primeras funciones del gestor de procesos: insertar proceso y listar procesos.

b) Codificador Principal (Cueva Luis):

- Se encargó de registrar y analizar las partes enviadas del código por los integrantes del equipo.
- Como codificador principal, fue responsable del gestor de memoria, es decir, de todo lo relacionado con la pila, además de integrar el código completo uniendo las diferentes secciones desarrolladas por los demás miembros.

c) Documentador 1 (López Damián):

- Como documentador, se encargó de revisar la versión final, añadiendo explicaciones a cada parte del código mediante comentarios.
- En su rol de codificador, fue responsable del gestor de procesos, abarcando desde la función 3 (eliminar proceso) hasta la función 6 (ordenar procesos por prioridad).

d) Documentador 2 (Capcha Nelinho):

- Se encargó de estructurar el informe de esta semana, elaborando el manual de usuario en DEV C++ y apoyando a otros miembros del equipo en la realización del informe.

- Como codificador, realizó el planificador de CPU, gestionando todo lo relacionado con la cola de prioridad, desde la función 7 (encolar CPU) hasta la función 9 (mostrar cola de CPU).

C. Tercera Semana:

Para esta semana, el grupo se enfocó en la creación de la versión final del código. Para ello, se utilizó el pseudocódigo y el código C++ previamente desarrollado en PSeInt y en Dev ++ como base para estructurar el código. De este modo, todo el grupo asumió el rol de codificador, dividiéndolos por secciones del programa. Además, cada integrante asumió distintas labores para la realización y revisión del proyecto, a continuación, explicadas:

a. Coordinador/ Analista de código (Capcha Nelinho):

- Describió el sistema de gestión de procesos además explicó los tres módulos principales del sistema.
- Mencionó los requerimientos funcionales y no funcionales que tendrá el sistema.
- Explicó las estructuras básicas que forma el sistema para luego justificar la elección.

b. Diseño de Solución (Cueva Luis):

- Explicó sobre el diseño y el proceso de la posible solución en el sistema de gestor de procesos y memorias.
- Detallo la lista de procesos, la cola de ejecución y la pila de memoria que tendrá el sistema con sus respectivas operaciones y funciones.
- Mencionó las definiciones de los algoritmos principales (Void).

c. Solución final del Código (Bejarano Mijael):

- Después de varios códigos corregidos y hechos con algunas observaciones mínimas se obtuvo la solución final del sistema de gestor de procesos y liberación de memorias.
- Mencionó la inclusión de las bibliotecas principales que formarán parte del código en C++, luego realizó las pruebas de validaciones para adquirir alguna observación extra en el menú.
- Realizó el manual de usuario explicando detalladamente cómo se interactúa con el sistema paso a paso, evitando cualquier colapso con el sistema por exceso de datos ingresados.

d. Evidencias de Trabajo en Equipo (López Naula):

- Redactó las evidencias de trabajo que hubo en el equipo mediante GitHub, códigos y documentos compartidos en Google Docs.
- Explico el rol que cumple cada integrante del equipo, desde la primera semana hasta la tercera semana.
- Realizó los cronogramas con fechas límites para evitar algún desacuerdo con la entrega del trabajo, además adjunto las actas de reuniones de las semanas reunidas mediante sesión meet.

4.2.2. Cronograma con fechas límite

Para hacer que el producto sea más organizado, se generó un cronograma de entrega de trabajos a cada integrante del equipo.

A. Primera Semana de Desarrollo del Sistema:

Primera Semana			
Roles	Día 1	Día 2	Día 3
Supervisor / Analista	-----	Recepción del código y análisis	Otorgar las observaciones del código a los codificadores
Codificador 1	Desarrollo del primer código	Envío del código mediante Git Hub	Realización del código con las observaciones
Codificador 2	Desarrollo del segundo código	Envío de la segunda parte del código en Git Hub	Realización del código con las respectivas observaciones
Documentador	-----	Redactar roles principales y primeras versiones del código	Redactar el código final explicando cada menú

B. Segunda Semana de Desarrollo del Sistema:

Segunda Semana				
Roles	Día 1	Día 2	Día 3	Día 4
Supervisor / Analista	Coordinación de las actividades	Recepción del código y análisis	Otorgar las observaciones del código a los codificadores	Añadir funciones al código
Codificador Principal	Desarrollo del código	Envío del código mediante Git Hub	Realización del código con las observaciones	-----
Documentador 1	-----	Evidencias de las primeras versiones	Redacción de las ventanas de ejecución del código	Capturas de pantallas de las ventanas de ejecución del código
Documentador 2	-----	Redactar los commits de los códigos al git hub	Tomar los commits actualizados en el git hub	Actualizar el historial de versiones

C. Tercera Semana de Desarrollo del Sistema:

Primera Semana			
Roles	Día 1	Día 2	Día 3
Supervisor / Analista	Coordinación	Recepción del código y análisis	Descripción del sistema
Codificador 1	Revisión completa del código	Envío del código mediante Git Hub	Código documentado correctamente
Documentador 1	-----	Redacción de la primera solución del código	Redacción de la solución final del Código
Documentador 2	-----	Redacción de roles de cada integrante por semana	Evidencias de Trabajo en Equipo en el desarrollo del sistema

4.2.3. Actas de reunión

ACTA DE REUNIÓN N° 1

Asignatura	ESTRUCTURA DE DATOS	Fecha:	19/05/2025
Responsable de grupo	Bejarano Miche Mijael	Hora de inicio:	20:00
Modalidad de Reunión	Virtual (Google Meet)	Hora de fin	20:36

Integrantes:

Apellidos y nombres	Asistió (Si/No)	% Participación	Firma
1. Bejarano Miche Mijael Joseph	Si	100%	M.J.
2. Captcha Hidalgo Nelinho	Si	100%	N.S.
3. Cueva Barrera Luis	Si	100%	L.E.
4. Lopez Naula Damián	Si	100%	D.J.

Temas tratados	Acuerdos	Responsables	Fecha de entrega
Distribución de Roles	Se distribuyeron correctamente los roles a cada integrante de equipo	Bejarano Miche	19/05/2025
Discusión del Pseudocódigo	Se analizó partes del código por realizar y mejoras que se realizarán	Todo el equipo	19/05/2025
Creación del GitHub	Se creó el repositorio donde se enviarán los pseudocódigos	Lopez Naula	19/05/2025

Observaciones:

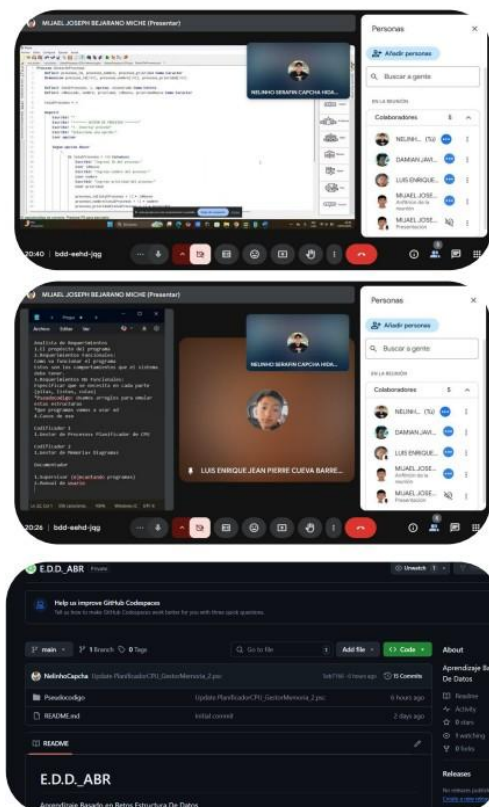
Hubo retrasos en la asignación durante 15 minutos, además se hubo una pequeña dificultad con la asignación de roles a cada integrante del equipo pero al final se asignó correctamente de una manera muy eficaz.

Evidencias de trabajo Grupal

Enlace de Herramienta Colaborativa:

- meet.google.com/bdd-eehd-iga
- https://github.com/Skadox2/E.D.D._ABR.git

Fotografías



En la primera reunión se propuso crear un repositorio en GitHub con el objetivo de subir el pseudocódigo de forma organizada. Inicialmente, el código se dividió en cuatro partes para facilitar su desarrollo individual y posterior integración en un código final. Además, se asignaron roles específicos a cada integrante del equipo para optimizar el avance del trabajo y asegurar una colaboración efectiva.

ACTA DE REUNIÓN N° 2

Asignatura	ESTRUCTURA DE DATOS	Fecha:	27/05/2025
Responsable de grupo	Bejarano Miche Mijael	Hora de inicio:	20:32
Modalidad de Reunión	Virtual (Google Meet)	Hora de fin	21:45

Integrantes:

Apellidos y nombres	Asistió (Si/No)	% Participación	Firma
1. Bejarano Miche, Mijael	SI	100	M.B
2. Capcha Hidalgo, Nelinho	SI	100	N.C
3. Cueva Barrera, Luis	SI	100	L.C
4. Lopez Naula, Damián	SI	100	D.L

Temas tratados	Acuerdos	Responsables	Fecha de entrega
Discusión del informe	Se planificó una estructura adecuada para presentar el informe	Todo el equipo	27/05/2025
Distribución del Informe	Se realizó la estructura del informe en partes equitativas, mediante un sorteo se estableció las responsabilidades de cada integrante.	Bejarano	27/05/2025
Distribución del Código	Se distribuyó la estructura del Código C++ mediante sorteo	Bejarano	27/05/2025
Creación de una carpeta en GH	Se creó otra carpeta en el repositorio de GH para colaborar en el código según lo planeado	Bejarano	28/05/2025

Observaciones:

Se esperó la conexión de uno de los integrantes, ya que llegó tarde por motivos académicos. Otro detalle para recalcar en la demora que tuvimos para estructurar el código, por lo que tardamos más de 40 min.

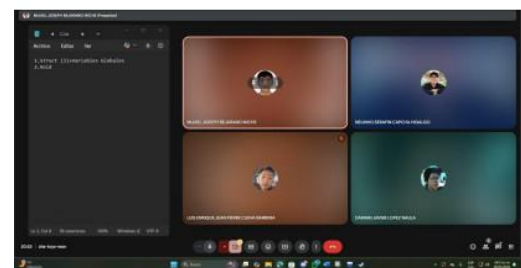
Evidencias de trabajo Grupal

Enlace de Herramienta Colaborativa:

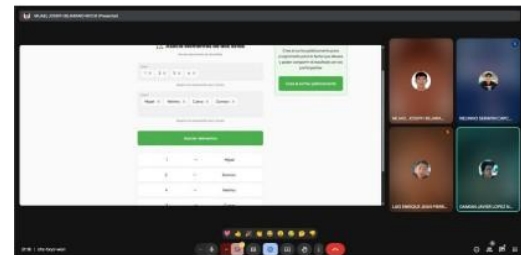
Google Doc: https://docs.google.com/document/d/1EP5805-15Ht-Y4A_153h6t1anp1xMoRwH0xaiE1Ered1?usp=sharing
 Meet: <http://meet.google.com/zhz-xyx-won>

GitHub: <https://github.com/Skadex2/E.D.D.-ABR.git>

Reunión Meet:



Distribución del trabajo



Sorteo de la estructura del Código y del informe

En la segunda semana se distribuyó el desarrollo del código, esta vez en lenguaje C++, dividiéndolo nuevamente en cuatro partes para luego integrarlas en un solo programa. Además, se asignaron roles a cada integrante con el fin de documentar el informe final, promoviendo así la participación equitativa dentro del equipo.

ACTA DE REUNIÓN N° 3

Asignatura	ESTRUCTURA DE DATOS	Fecha:	02/06/2025
Responsable de grupo	Bejarano Miche Mijael	Hora de inicio:	20:00
Modalidad de Reunión	Virtual (Google Meet)	Hora de fin	20:45

Integrantes:

Apellidos y nombres	Asistió (Si/No)	% Participación	Firma
1. Bejarano Miche Mijael	SI	100%	M.J.
2. Captcha Hidalgo Nelinho	SI	100%	N.S.
3. Cueva Barrera Luis	SI	100%	L.E.
4. Lopez Naula Damián	SI	100%	D.J.

Temas tratados	Acuerdos	Responsables	Fecha de entrega
Distribución de Roles	Se distribuyeron correctamente los roles a cada integrante de equipo	Bejarano Miche	02/06/2025
Discusión del código C++	Se analizó partes del código por realizar y mejoras que se realizarán	Todo el equipo	02/06/2025
Generación del archivo PPT	Se creó el archivo ppt para presentación del código	Bejarano Miche	03/06/2025

Observaciones:

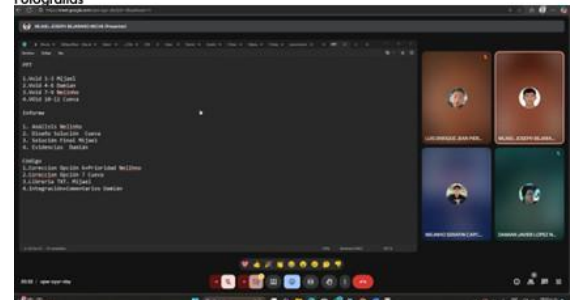
Hubo retrasos en la reunión de solo 3 a 5 minutos por falta de un integrante que por motivos académicos demora, pero al final logro llegar a la reunión, sin embargo, se distribuyó las partes correctamente a cada integrante del equipo.

Evidencias de trabajo Grupal

Enlace de Herramienta Colaborativa:

<http://meet.google.com/rqh-bpgh-sfx> // <https://www.canva.com/design/DAGpURKObbk/Q62DzuUvJKs-2pUkvHbQ/edil>

Fotografías



CANVA:



En la tercera semana se revisó el código, resaltando pequeños errores que debían corregirse. También se asignaron las secciones correspondientes para el desarrollo del informe. Además, se elaboró un diseño en Canva con el propósito de explicar la estructura del programa y preparar la exposición programada para esa semana.

LINK DEL GITHUB:

- https://github.com/Skadex2/E.D.D._ABR.git

LINK DE LA PRESENTACIÓN:

- https://www.canva.com/design/DAGpURKObbk/Q62DZuUjVjKs-2pLkuvHbQ/edit?utm_content=DAGpURKObbk&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton