Sistema de Gestión de Procesos Grupo_01

Integrantes:

-Caso Caysahuana Henrik Anderson
-Cabrera Ortega Dhastin Ray
-Yupanqui Suarez Jesus Angel
-Quinte Perez Erick Zahid

CAPÍTULO 1: Análisis del Problema

1. Descripción del Problema

Imagina que tu computadora está ejecutando varios programas al mismo tiempo: el navegador, un editor de código, la música de fondo... Cada uno de estos es un *proceso* que consume recursos. El problema es: ¿cómo administrar eficientemente estos procesos para que el sistema no colapse?

Nuestro Sistema de Gestión de Procesos simula esta situación, permitiendo:

- Registrar procesos nuevos (como abrir un programa).
- Priorizarlos (que el antivirus tenga más CPU que un juego).
- Asignarles memoria (evitando que se queden sin RAM).
- Ejecutarlos en orden según su importancia.

2. Requerimientos del Sistema

Funcionales (¿Qué debe hacer?)

2.1. Gestión de Procesos:

- Añadir procesos (ej.: ID: 1, Nombre: Chrome, Prioridad: 5, Memoria: 500MB).
- Eliminar procesos (cuando se cierran).
- Buscar procesos por nombre o ID (para depurar errores).

2.2. Planificación de CPU:

- Ordenar procesos por prioridad (los de prioridad alta se ejecutan primero).
- Simular su "ejecución" (desencolarlos).

2.3. Gestión de Memoria:

- Asignar memoria a procesos (como cuando abres Photoshop y pide 2GB).
- Liberar memoria al cerrarlos.
- Mostrar cuánta memoria hay libre y en uso.

No Funcionales (¿Cómo debe hacerlo?)

☑ Rendimiento:

- Que no se tarde años en buscar/eliminar un proceso (usar listas enlazadas).
- Prioridades bien gestionadas (cola de prioridad).

☑ Interfaz:

- Menú sencillo en consola (sin pantallas complicadas).
- Mensajes claros: "¡Proceso 'Chrome' eliminado!".

☑ Tecnología:

- Solo C++ estándar (sin frameworks raros).
- Hecho en Dev-C++ (como lo pide el profesor).

3. Estructuras de Datos Elegidas

Parte del Sistema	Estructura	¿Por qué?
Lista de todos los procesos	Lista enlazada simple	Fácil añadir/eliminar procesos dinámicamente (como una lista de tareas).
Planificador de CPU	Cola de prioridad	Garantiza que lo urgente (prioridad alta) se atienda primero.
Gestor de Memoria	Pila	Simula cómo los sistemas reales liberan memoria (el último en entrar, primero en salir).

4. ¿Por qué estas estructuras?

☑ Lista Enlazada:

- Es flexible: si se cierra un programa, se elimina su nodo sin reorganizar todo.
- Perfecta para cantidades variables de procesos (como en un PC real).

☑ Cola de Prioridad:

- Ejemplo: Si tienes un antivirus (prioridad 10) y un juego (prioridad 2), el antivirus se ejecuta primero.
- Se implementó como una lista ordenada por prioridad (más sencillo que un *heap*).

☑ Pila para Memoria:

- Ejemplo práctico: Cuando cierras Chrome, libera la memoria que usabas.
- Es eficiente: `push()` y `pop()` son operaciones rápidas (O(1)).

☑ Ventajas:

- Simple pero potente: Cumple con lo requerido sin complicaciones.
- Se parece a un SO real: Prioridades, memoria limitada... ¡como Windows o Linux!

☑ Limitaciones:

- Si hay 10,000 procesos, la búsqueda sería lenta (pero para este proyecto, es suficiente).
- No usa archivos para guardar datos (pero se podría añadir después).

Capítulo 2: Diseño de la Solución

1. Descripción de estructuras de datos y operaciones:

Análisis del Sistema de Gestión de Procesos
Este programa implementa un sistema de gestión de procesos utilizando tres estructuras de datos principales: una lista enlazada, una cola de prioridad y una pila.

Estructuras de Datos

1. Estructura Proceso
Representa un proceso con los siguientes campos:

id: Identificador único del proceso
nombre: Nombre descriptivo del proceso
prioridad: Valor numérico que indica la prioridad (1-10)
memoria: Cantidad de memoria asignada en MB
siguiente: Puntero al siguiente proceso en una lista enlazada

2. Clase ListaProcesos
Implementa una lista enlazada simple para gestionar todos los procesos del sistema.

Operaciones principales:

insertarProceso(): Añade un nuevo proceso al inicio de la lista
eliminarProceso(): Elimina un proceso por su ID
buscarPorNombre(): Busca procesos cuyo nombre contenga una cadena dada
modificarPrioridad(): Cambia la prioridad de un proceso existente
mostrarProcesos(): Muestra todos los procesos en la lista
getCabeza(): Devuelve el puntero al primer proceso (para integración con otras estructuras)

3. Clase ColaCPU
Implementa una cola de prioridad para planificar la ejecución de procesos.

Operaciones principales:

encolar(): Añade un proceso a la cola ordenado por prioridad (mayor prioridad primero)

desencolar(): Extrae y devuelve el proceso con mayor prioridad para ejecución

mostrarCola(): Muestra todos los procesos en la cola de CPU

4. Clase PilaMemoria

Implementa una pila para gestionar la asignación y liberación de memoria.

Operaciones principales:

asignarMemoria(): Añade un proceso a la pila (asigna memoria)
liberarMemoria(): Elimina el último proceso añadido (libera memoria)
mostrarMemoria(): Muestra todos los procesos usando memoria y estadísticas de uso

Flujo del Programa

1. El programa principal (main) implementa un menú interactivo que permite:
2. Gestionar procesos (añadir, eliminar, buscar, modificar)
3. Planificar ejecución (encolar procesos en CPU, ejecutar procesos)
4. Gestionar memoria (asignar, liberar, ver estado)
5. Visualizar información (lista de procesos, cola de CPU, estado de memoria)

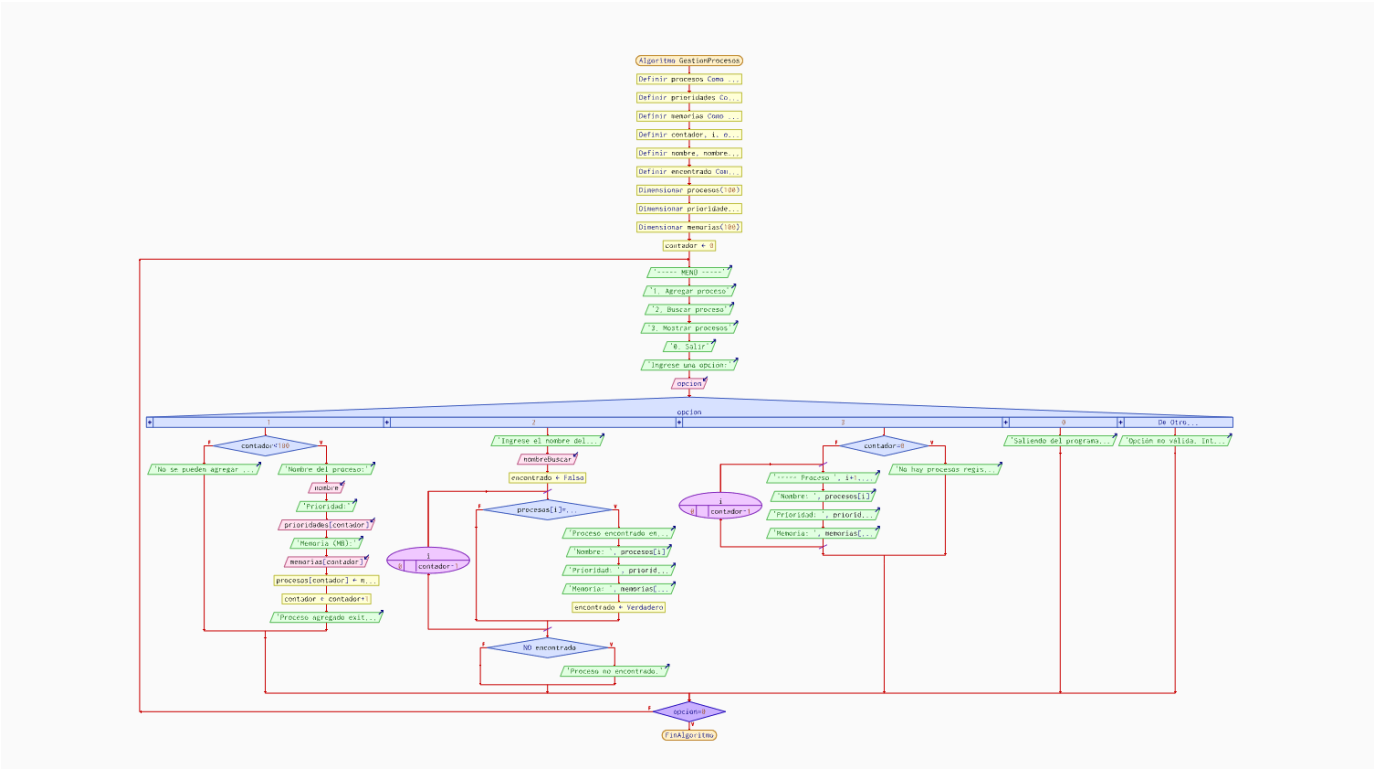
Características destacables

1. Gestión integrada: Las tres estructuras trabajan juntas usando punteros a los mismos objetos Proceso.
2. Asignación dinámica de memoria: Uso de new y delete para gestión manual de memoria.
3. Interfaz de usuario: Menú claro con validación básica de entrada.
4. Estadísticas de memoria: Muestra memoria total (4GB simulados), usada, disponible y porcentaje.

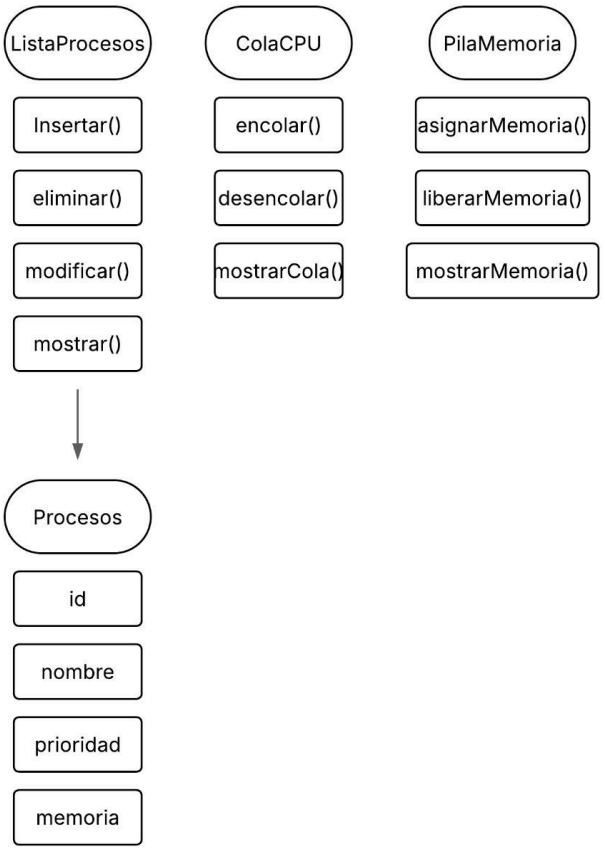
2. Algoritmos principales:

```
Definir memorias Como Entero
Definir contador, i, opcion Como Entero
Definir nombre, nombreBuscar Como Cadena
Definir encontrado Como Lógico
Dimensionar procesos(100)
Dimensionar prioridades(100)
Dimensionar memorias(100)
contador ← 0
Repetir
    Escribir '----- MENÚ -----'
    Escribir '1. Agregar proceso'
    Escribir '2. Buscar proceso'
    Escribir '3. Mostrar procesos'
    Escribir '0. Salir'
    Escribir 'Ingrese una opción:'
    Leer opcion
    Según opcion Hacer
        1:
            Si contador<100 Entonces
                Escribir 'Nombre del proceso:'
                Leer nombre
                Escribir 'Prioridad:'
                Leer prioridades[contador]
                Escribir 'Memoria (MB):'
                Leer memorias[contador]
                procesos[contador] ← nombre
                contador ← contador+1
                Escribir 'Proceso agregado exitosamente.'
            SiNo
                Escribir 'No se pueden agregar más procesos (límite alcanzado).'
            FinSi
        2:
            Escribir 'Ingrese el nombre del proceso a buscar:'
            Leer nombreBuscar
            encontrado ← Falso
            Para i←0 Hasta contador-1 Hacer
                Si procesos[i]=nombreBuscar Entonces
                    Escribir 'Proceso encontrado en la posición ', i+1
                    Escribir 'Nombre: ', procesos[i]
                    Escribir 'Prioridad: ', prioridades[i]
                    Escribir 'Memoria: ', memorias[i], ' MB'
                    encontrado ← Verdadero
                FinSi
            FinPara
```

```
            Si  NO encontrado Entonces
                Escribir 'Proceso no encontrado.'
            FinSi
        3:
            Si contador=0 Entonces
                Escribir 'No hay procesos registrados.'
            SiNo
                Para i<-0 Hasta contador-1 Hacer
                    Escribir '----- Proceso ', i+1, ' -----'
                    Escribir 'Nombre: ', procesos[i]
                    Escribir 'Prioridad: ', prioridades[i]
                    Escribir 'Memoria: ', memorias[i], ' MB'
                FinPara
            FinSi
        0:
            Escribir 'Saliendo del programa...'
        De Otro Modo:
            Escribir 'Opción no válida. Intente de nuevo.'
    FinSegún
Hasta Que opcion=0
FinAlgoritmo

```



3. Diagramas de Flujo:



4. Justificación del diseño:

El diseño del sistema responde a la necesidad de simular la gestión de procesos como ocurre en un sistema operativo, utilizando estructuras de datos fundamentales:

Lista Enlazada – Gestión Global de Procesos

La lista enlazada permite insertar, eliminar y modificar procesos de manera flexible, sin requerir una cantidad fija de espacio ni alinear elementos, lo que favorece la eficiencia en memoria y operaciones dinámicas.

Cola de Prioridad – Planificación de CPU

El uso de una cola ordenada por prioridad refleja el comportamiento de un planificador de CPU real, en donde los procesos más prioritarios deben ser ejecutados antes. Esta estructura permite una inserción controlada y extracción eficiente para simulación de ejecución.

Pila – Gestión de Memoria

La pila simula la asignación de memoria con comportamiento tipo LIFO (Last In, First Out), útil para reflejar cómo ciertos sistemas liberan recursos en orden inverso al de

su asignación. Además, permite implementar fácilmente estadísticas de uso de memoria.

Integración entre estructuras

El uso de punteros comunes a objetos de tipo `Proceso` permite una gestión centralizada, evitando duplicación de datos y permitiendo sincronización entre estructuras. Por ejemplo, un proceso añadido a la pila o cola mantiene su vínculo con la lista principal.

Interfaz y validación

El sistema incorpora un menú interactivo sencillo con validación de entrada, lo que facilita la experiencia del usuario final sin sacrificar el control del flujo del programa.

Asignación dinámica y simulación realista

El diseño aprovecha el control manual de memoria con `new` y `delete`, lo que permite simular con mayor realismo cómo se gestiona la memoria en sistemas reales, al mismo tiempo que proporciona al estudiante una experiencia completa de gestión de recursos.

Capítulo 3: Solución Final

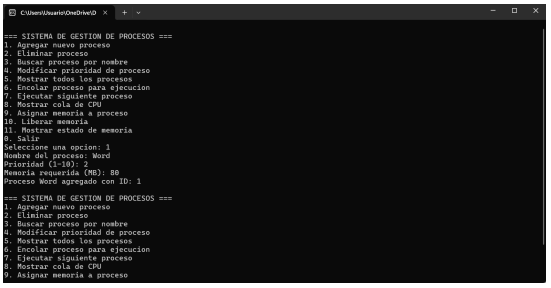
1. Código limpio, bien comentado y estructurado.

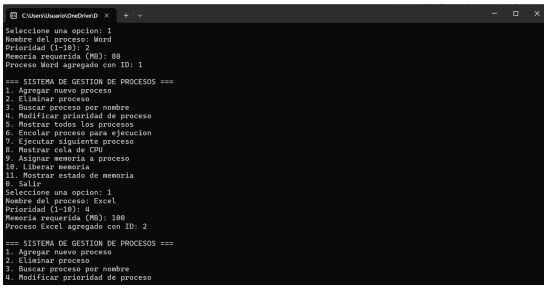
```cpp
#include <iostream>    // Incluye la librería para entrada y salida estándar
#include <string>      // Incluye la librería para manejo de cadenas de texto (strings)
using namespace std;   // Usa el espacio de nombres std para evitar escribir std:: antes de cout, string, etc.

// Estructura para representar un proceso
struct Proceso {
    int id;             // Identificador único del proceso
    string nombre;      // Nombre del proceso
    int prioridad;      // Prioridad del proceso
    int memoria;        // Memoria asignada al proceso (en MB)
    Proceso* siguiente; // Puntero al siguiente proceso en la lista enlazada
};

// Clase para la Lista Enlazada (Gestor de Procesos)
class ListaProcesos {
private:
    Proceso* cabeza;    // Puntero al primer proceso de la lista
    int contadorId;     // Contador para asignar IDs únicos a los procesos
public:
    ListaProcesos() : cabeza(NULL), contadorId(1) {}  // Constructor que inicializa cabeza en NULL y contadorId en 1

    // Insertar nuevo proceso
    void insertarProceso(string nombre, int prioridad, int memoria) {
        Proceso* nuevo = new Proceso;            // Crea un nuevo proceso dinámicamente
        nuevo->id = contadorId++;                // Asigna un ID único y luego incrementa el contador
        nuevo->nombre = nombre;                  // Asigna el nombre al proceso
        nuevo->prioridad = prioridad;            // Asigna la prioridad al proceso
        nuevo->memoria = memoria;                // Asigna la memoria al proceso
        nuevo->siguiente = cabeza;               // El nuevo proceso apunta al proceso que era cabeza (inserción al inicio)
        cabeza = nuevo;                          // Actualiza cabeza para que apunte al nuevo proceso
        cout << "Proceso " << nombre << " agregado con ID: " << nuevo->id << endl; // Mensaje confirmando la inserción
    }

    // Eliminar proceso por ID
    bool eliminarProceso(int id) {
        Proceso* actual = cabeza;    // Apunta al primer proceso de la lista
        Proceso* anterior = NULL;    // Apunta al proceso anterior al actual (inicialmente NULL)

        while (actual != NULL) {     // Recorre la lista hasta que actual sea NULL
            if (actual->id == id) {  // Si encuentra el proceso con el ID buscado
                if (anterior == NULL) {          // Si es el primer proceso (cabeza)
                    cabeza = actual->siguiente;  // Actualiza cabeza al siguiente proceso
                } else {
                    anterior->siguiente = actual->siguiente; // Salta el proceso actual en la lista
                }
                delete actual;  // Libera la memoria del proceso eliminado
                cout << "Proceso con ID " << id << " eliminado." << endl; // Mensaje confirmando eliminación
                return true;   // Retorna true indicando éxito
            }
            anterior = actual;          // Actualiza anterior para la próxima iteración
            actual = actual->siguiente; // Avanza al siguiente proceso
        }
        cout << "Proceso con ID " << id << " no encontrado." << endl; // Mensaje si no encuentra el proceso
        return false;  // Retorna false indicando que no se encontró el proceso
    }

    // Buscar proceso por nombre
    void buscarPorNombre(string nombre) {
        Proceso* actual = cabeza;  // Apunta al primer proceso de la lista
        bool encontrado = false;   // Bandera para saber si se encontró algún proceso

        cout << "\nResultados de busqueda para '" << nombre << "':" << endl; // Encabezado de resultados
        while (actual != NULL) {   // Recorre la lista hasta el final
            if (actual->nombre.find(nombre) != string::npos) {  // Busca si el nombre contiene la cadena dada
                cout << "ID: " << actual->id << ", Nombre: " << actual->nombre ;
                cout << ", Prioridad: " << actual->prioridad ;
                cout << ", Memoria: " << actual->memoria << "MB" << endl;  // Muestra info del proceso encontrado
                encontrado = true;   // Marca que encontró al menos un proceso
            }
            actual = actual->siguiente;  // Avanza al siguiente proceso
        }

        if (!encontrado) {  // Si no encontró ningún proceso
            cout << "No se encontraron procesos con ese nombre." << endl;  // Mensaje de no resultados
        }
    }

    // Modificar prioridad de un proceso
    bool modificarPrioridad(int id, int nuevaPrioridad) {
        Proceso* actual = cabeza;  // Apunta al primer proceso de la lista

        while (actual != NULL) {   // Recorre la lista
            if (actual->id == id) {  // Busca el proceso con el ID dado
                actual->prioridad = nuevaPrioridad;  // Cambia la prioridad
                cout << "Prioridad del proceso ID " << id << " actualizada a " << nuevaPrioridad << endl; // Mensaje de confirmación
                return true;  // Retorna true indicando éxito
            }
            actual = actual->siguiente;  // Avanza al siguiente proceso
        }
        cout << "Proceso con ID " << id << " no encontrado." << endl; // Mensaje si no encuentra el proceso
        return false;  // Retorna false indicando que no encontró el proceso
    }

    // Mostrar todos los procesos
    void mostrarProcesos() {
        Proceso* actual = cabeza;  // Apunta al primer proceso de la lista

        cout << "\n=== LISTA DE PROCESOS ===" << endl;  // Encabezado para mostrar la lista
        if (actual == NULL) {    // Si la lista está vacía
            cout << "No hay procesos registrados." << endl;  // Mensaje de lista vacía
            return;  // Sale del método
        }

        while (actual != NULL) {  // Recorre toda la lista
            cout << "ID: " << actual->id << ", Nombre: " << actual->nombre;
            cout << ", Prioridad: " << actual->prioridad;
            cout << ", Memoria: " << actual->memoria << "MB" << endl;  // Muestra los datos de cada proceso
            actual = actual->siguiente;  // Avanza al siguiente proceso
        }
    }

    // Método para obtener el puntero a la cabeza de la lista (solo lectura)
    Proceso* getCabeza() const {
        return cabeza;  // Retorna el puntero al primer proceso
    }
};

// Nodo para la cola de prioridad
struct NodoCola {
    Proceso* proceso;    // Puntero al proceso almacenado en el nodo
    NodoCola* siguiente; // Puntero al siguiente nodo en la cola
};

// Clase para la Cola de Prioridad (PLanificador de CPU)
class ColaCPU {
private:
    NodoCola* frente;   // Puntero al frente de la cola
    NodoCola* final;    // Puntero al final de la cola
public:
    ColaCPU() : frente(NULL), final(NULL) {}  // Constructor que inicializa punteros en NULL

    // Encolar proceso según prioridad (mayor prioridad primero)
    void encolar(Proceso* proceso) {
        NodoCola* nuevo = new NodoCola;  // Crea un nuevo nodo para la cola
        nuevo->proceso = proceso;        // Asigna el proceso al nodo
        nuevo->siguiente = NULL;         // El siguiente nodo es NULL (al final)
```

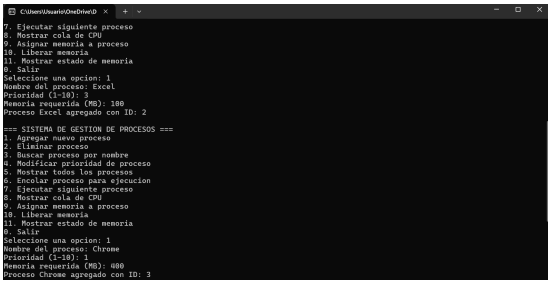
```
        if (frente == NULL) {             // Si la cola está vacía
            frente = final = nuevo;       // El nuevo nodo es frente y final
        } else if (proceso->prioridad > frente->proceso->prioridad) {  // Si el nuevo proceso tiene mayor prioridad que el frente
            nuevo->siguiente = frente;    // Inserta el nuevo nodo antes del frente
            frente = nuevo;               // Actualiza frente al nuevo nodo
        } else {
            NodoCola* actual = frente;    // Nodo auxiliar para recorrer la cola
            // Busca la posición donde insertar el nuevo nodo manteniendo orden por prioridad descendente
            while (actual->siguiente != NULL &&
                   actual->siguiente->proceso->prioridad >= proceso->prioridad) {
                actual = actual->siguiente;  // Avanza al siguiente nodo
            }
            nuevo->siguiente = actual->siguiente;  // Inserta el nuevo nodo en la posición encontrada
            actual->siguiente = nuevo;
            if (nuevo->siguiente == NULL) {  // Si el nuevo nodo quedó al final
                final = nuevo;               // Actualiza el final
            }
        }
    }

    // Desencolar proceso (ejecutar)
    Proceso* desencolar() {
        if (frente == NULL) {   // Si la cola está vacía
            return NULL;        // Retorna NULL
        }

        NodoCola* temp = frente;      // Guarda temporalmente el nodo del frente
        Proceso* proceso = frente->proceso;  // Obtiene el proceso del frente
        frente = frente->siguiente;   // Avanza el frente al siguiente nodo

        if (frente == NULL) {         // Si la cola queda vacía
            final = NULL;             // Actualiza final a NULL
        }

        delete temp;                  // Libera la memoria del nodo eliminado
        return proceso;               // Retorna el proceso desencolado
    }

    // Mostrar cola de CPU
    void mostrarCola() {
        NodoCola* actual = frente;  // Apunta al primer nodo de la cola

        cout << "\n=== COLA DE CPU (Ordenados por prioridad) ===" << endl;  // Encabezado de la cola
        if (actual == NULL) {       // Si la cola está vacía
            cout << "No hay procesos en la cola de CPU." << endl;  // Mensaje de cola vacía
            return;                 // Sale del método
        }

        while (actual != NULL) {    // Recorre la cola mostrando los procesos
            cout << "ID: " << actual->proceso->id << ", Nombre: " << actual->proceso->nombre;
            cout << ", Prioridad: " << actual->proceso->prioridad;
            cout << ", Memoria: " << actual->proceso->memoria << "MB" << endl;  // Muestra info del proceso
            actual = actual->siguiente;  // Avanza al siguiente nodo
        }
    }
};
// Clase para la Pila (Gestor de Memoria)
class PilaMemoria {
private:
    Proceso* tope; // Puntero al tope de la pila (último proceso asignado en memoria)
public:
    PilaMemoria() : tope(NULL) {} // Constructor que inicializa el tope como NULL (memoria vacía)

    // Push (asignar memoria)
    void asignarMemoria(Proceso* proceso) {
        proceso->siguiente = tope; // El nuevo proceso apunta al tope actual
        tope = proceso; // Se actualiza el tope al nuevo proceso

        cout << "Memoria asignada al proceso " << proceso->nombre << endl; // Mensaje de confirmación
    }

    // Pop (liberar memoria)
    Proceso* liberarMemoria() {
        if (tope == NULL) { // Si la pila está vacía
            return NULL; // Retornar NULL (no hay procesos para liberar)
        }

        Proceso* proceso = tope; // Guardar el proceso actual del tope
        tope = tope->siguiente; // Actualizar el tope al siguiente proceso
        proceso->siguiente = NULL; // Eliminar el vínculo al siguiente
        cout << "Memoria liberada del proceso " << proceso->nombre << endl; // Mensaje de liberación
        return proceso; // Retornar el proceso que fue liberado
    }

    // Mostrar estado de la memoria
    void mostrarMemoria() {
        Proceso* actual = tope; // Puntero auxiliar para recorrer la pila
        int memoriaUsada = 0; // Variable para sumar la memoria total usada

        cout << "\n=== ESTADO DE LA MEMORIA ===" << endl; // Encabezado del estado de memoria
        if (actual == NULL) { // Si no hay procesos
            cout << "No hay procesos usando memoria." << endl; // Mensaje indicativo
        }

        while (actual != NULL) { // Mientras haya procesos en la pila
            cout << "ID: " << actual->id << ", Nombre: " << actual->nombre
                 << ", Memoria usada: " << actual->memoria << "MB" << endl; // Mostrar info del proceso
            memoriaUsada += actual->memoria; // Sumar su memoria al total
            actual = actual->siguiente; // Avanzar al siguiente proceso
        }

        const int MEMORIA_TOTAL = 4096; // 4GB de RAM simulada
        cout << "\nResumen de memoria:" << endl;
        cout << "Memoria total: " << MEMORIA_TOTAL << "MB" << endl; // Memoria disponible
        cout << "Memoria usada: " << memoriaUsada << "MB" << endl; // Memoria actualmente usada
        cout << "Memoria disponible: " << MEMORIA_TOTAL - memoriaUsada << "MB" << endl; // Memoria libre

        float porcentaje = (static_cast<float>(memoriaUsada) / MEMORIA_TOTAL) * 100; // Calcular porcentaje usado
        cout << "Porcentaje usado: " << porcentaje << "%" << endl; // Mostrar porcentaje
    }
};

// Función para mostrar el menú principal
void mostrarMenu() {
    cout << "\n=== SISTEMA DE GESTION DE PROCESOS ===" << endl; // Título
    cout << "1. Agregar nuevo proceso" << endl; // Opción 1
    cout << "2. Eliminar proceso" << endl; // Opción 2
    cout << "3. Buscar proceso por nombre" << endl; // Opción 3
    cout << "4. Modificar prioridad de proceso" << endl; // Opción 4
    cout << "5. Mostrar todos los procesos" << endl; // Opción 5
    cout << "6. Encolar proceso para ejecucion" << endl; // Opción 6
    cout << "7. Ejecutar siguiente proceso" << endl; // Opción 7
    cout << "8. Mostrar cola de CPU" << endl; // Opción 8
    cout << "9. Asignar memoria a proceso" << endl; // Opción 9
    cout << "10. Liberar memoria" << endl; // Opción 10
    cout << "11. Mostrar estado de memoria" << endl; // Opción 11
    cout << "0. Salir" << endl; // Opción 0
    cout << "Seleccione una opcion: "; // Solicitar entrada del usuario
}

int main() {
    ListaProcesos lista; // Crear objeto para manejar lista de procesos
    ColaCPU colaCPU; // Crear objeto para manejar cola de CPU
    PilaMemoria pilaMemoria; // Crear objeto para manejar memoria como pila
    int opcion; // Variable para almacenar la opción del menú
```

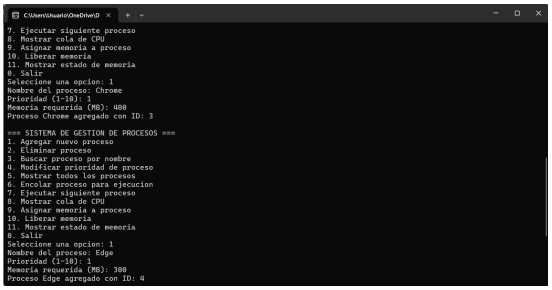
```
    do {
        mostrarMenu(); // Mostrar el menú en cada iteración
        cin >> opcion; // Leer opción del usuario
        cin.ignore(); // Limpiar el buffer del teclado

        switch (opcion) { // Evaluar la opción seleccionada
            case 1: {
                string nombre;
                int prioridad, memoria;

                cout << "Nombre del proceso: "; // Solicitar nombre
                getline(cin, nombre); // Leer nombre con espacios
                cout << "Prioridad (1-10): "; // Solicitar prioridad
                cin >> prioridad;
                cout << "Memoria requerida (MB): "; // Solicitar memoria
                cin >> memoria;

                lista.insertarProceso(nombre, prioridad, memoria); // Insertar nuevo proceso en la lista
                break;
            }
            case 2: {
                int id;
                cout << "ID del proceso a eliminar: "; // Solicitar ID
                cin >> id;
                lista.eliminarProceso(id); // Eliminar proceso por ID
                break;
            }
            case 3: {
                string nombre;
                cout << "Nombre a buscar: "; // Solicitar nombre
                getline(cin, nombre); // Leer nombre
                lista.buscarPorNombre(nombre); // Buscar proceso por nombre
                break;
            }
            case 4: {
                int id, nuevaPrioridad;
                cout << "ID del proceso: "; // Solicitar ID
                cin >> id;
                cout << "Nueva prioridad (1-10): "; // Solicitar nueva prioridad
                cin >> nuevaPrioridad;
                lista.modificarPrioridad(id, nuevaPrioridad); // Modificar prioridad del proceso
                break;
            }
            case 5:
                lista.mostrarProcesos(); // Mostrar todos los procesos
                break;
            case 6: {
                int id;
                cout << "ID del proceso a encolar: "; // Solicitar ID
                cin >> id;

                // Buscar el proceso en la lista
                Proceso* actual = lista.getCabeza(); // Obtener el inicio de la lista
                Proceso* procesoEncolar = NULL; // Inicializar puntero

                while (actual != NULL) { // Recorrer la lista
                    if (actual->id == id) { // Si se encuentra el proceso
                        procesoEncolar = actual;
                        break; // Salir del ciclo
                    }
                    actual = actual->siguiente; // Avanzar en la lista
                }

                if (procesoEncolar != NULL) { // Si se encontró el proceso
                    colaCPU.encolar(procesoEncolar); // Encolarlo para ejecución
                    cout << "Proceso " << procesoEncolar->nombre << " encolado para ejecucion." << endl;
                } else {
                    cout << "Proceso con ID " << id << " no encontrado." << endl; // No encontrado
                }
                break;
            }
            case 7: {
                Proceso* procesoEjecutado = colaCPU.desencolar(); // Desencolar el siguiente proceso
                if (procesoEjecutado != NULL) { // Si hay proceso
                    cout << "Ejecutando proceso: " << procesoEjecutado->nombre << endl;
                } else {
                    cout << "No hay procesos en la cola de CPU." << endl; // Cola vacía
                }
                break;
            }
            case 8:
                colaCPU.mostrarCola(); // Mostrar cola de procesos
                break;
            case 9: {
                int id;
                cout << "ID del proceso para asignar memoria: "; // Solicitar ID
                cin >> id;

                // Buscar el proceso en la lista
                Proceso* actual = lista.getCabeza(); // Obtener el inicio de la lista
                Proceso* procesoMemoria = NULL;

                while (actual != NULL) { // Recorrer la lista
                    if (actual->id == id) {
                        procesoMemoria = actual;
                        break;
                    }
                    actual = actual->siguiente;
                }

                if (procesoMemoria != NULL) {
                    pilaMemoria.asignarMemoria(procesoMemoria); // Asignar memoria
                } else {
                    cout << "Proceso con ID " << id << " no encontrado." << endl; // No encontrado
                }
                break;
            }
            case 10: {
                Proceso* procesoLiberado = pilaMemoria.liberarMemoria(); // Liberar el último proceso de memoria
                if (procesoLiberado == NULL) { // Si no había procesos
                    cout << "No hay procesos usando memoria." << endl;
                }
                break;
            }
            case 11:
                pilaMemoria.mostrarMemoria(); // Mostrar estado de la memoria
                break;
            case 0:
                cout << "Saliendo del sistema..." << endl; // Finalizar programa
                break;
            default:
                cout << "Opcion no valida. Intente de nuevo." << endl; // Opción incorrecta
        }
    } while (opcion != 0); // Repetir hasta que el usuario elija salir

    return 0; // Fin del programa
}
```

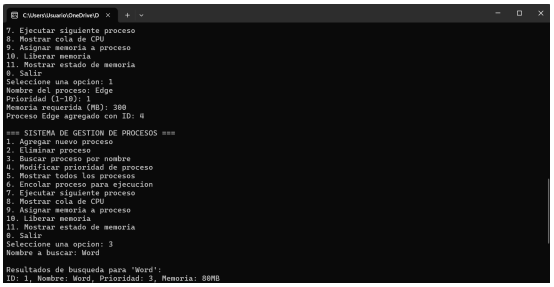
2. Capturas de pantalla de las ventanas de ejecución con las diversas pruebas de validación de datos

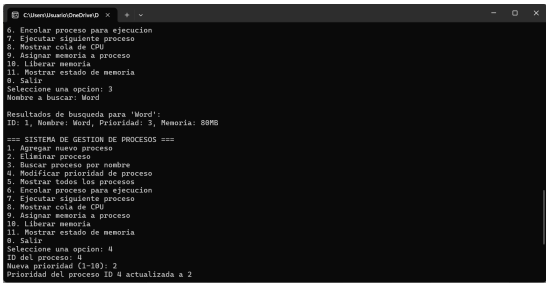


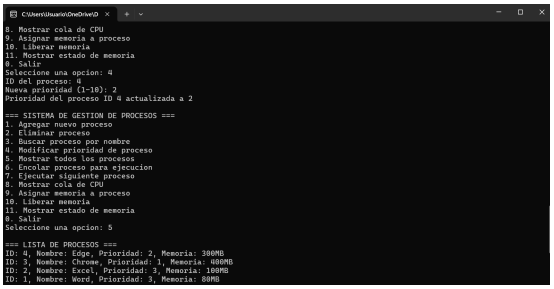


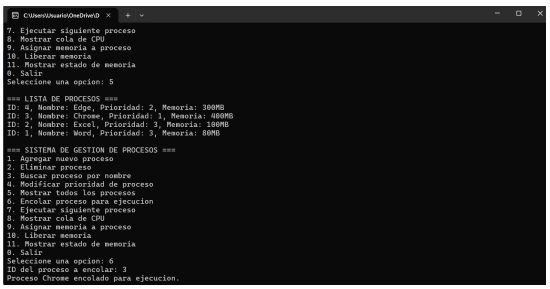


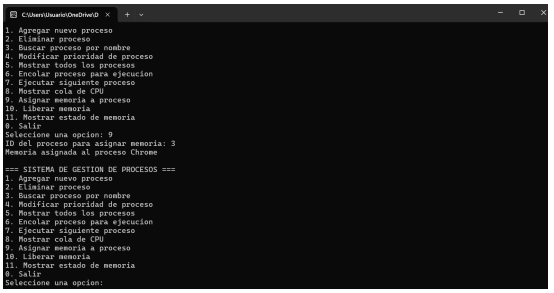


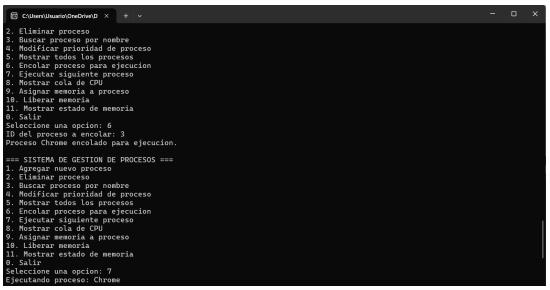












3. Manual de Usuario

Sistema de Gestión de Procesos

Introducción
¡Hola! Estimado usuario supongo que tendrás algunas dudas de como manejar este gestor de procesos a través de C++, pues no te preocupes aqui esta sistema te ayuda a gestionar procesos como si fueras el administrador de un pequeño sistema operativo. Puedes:

-Crear, eliminar y buscar procesos
-Planificar su ejecución en la CPU (¡como un jefe de proyectos!)
-Administrar la memoria (asignar y liberar recursos)

Todo se controla desde un menú sencillo, así que no te preocupes, no necesitas ser un experto en programación para usarlo.

Funcionalidades principales

Módulo	¿Qué puedes hacer?
Gestión de procesos	Agregar, eliminar, buscar y modificar procesos.
Planificación de CPU	Ordenar procesos por prioridad y ejecutarlos.
Gestión de memoria	Asignar y liberar memoria (como una pila de libros: el último que entra es el primero que sale).

Instrucciones paso a paso
1. Agregar un nuevo proceso (Opción 1)

Paso 1: Elige la opción 1 en el menú.
Paso 2: Escribe el nombre del proceso (ej: "Chrome").
Paso 3: Asigna una prioridad (1-10) (10 = más importante).
Paso 4: Indica cuánta memoria (MB) necesita.

- ☑ El sistema te dará un ID automático.

Ejemplo:
Nombre: Spotify, Prioridad: 7, Memoria: 150 MB → Se crea con ID: 3

2. Eliminar un proceso (Opción 2)
Paso 1: Elige la opción 2.
Paso 2: Ingresa el ID del proceso a eliminar.

- ☑ Si existe, se borrará.
- **X** Si no, te dirá que no lo encontró.

3. Buscar un proceso por nombre (Opción 3)
Paso 1: Elige la opción 3.
Paso 2: Escribe parte del nombre (no hace falta el nombre completo).

- ☑ Te mostrará todos los que coincidan.

Ejemplo: Si buscas "chr", aparecerá "Chrome".

4. Cambiar la prioridad de un proceso (Opción 4)
Paso 1: Elige la opción 4.
Paso 2: Ingresa el ID del proceso.
Paso 3: Pon la nueva prioridad (1-10).

- ☑ Se actualizará si existe.

5. Ver todos los procesos (Opción 5)
Paso 1: Elige la opción 5.

- ☑ Te mostrará una lista con todos los procesos.

6. Poner un proceso en la cola de CPU (Opción 6)
Paso 1: Elige la opción 6.
Paso 2: Ingresa el ID del proceso.

- ☑ Se encolará según su prioridad (los más urgentes primero).

7. Ejecutar el siguiente proceso (Opción 7)
Paso 1: Elige la opción 7.

- ☑ **Ejecutará el proceso con mayor prioridad.**
- **X Si no hay nada en cola, te avisará.**

8. Ver la cola de CPU (Opción 8)
Paso 1: Elige la opción 8.

- ☑ **Te mostrará qué procesos están esperando para ejecutarse, ordenados por prioridad.**

9. Asignar memoria a un proceso (Opción 9)
Paso 1: Elige la opción 9.

Paso 2: Ingresa el ID del proceso.

- ☑ Se asignará memoria (como guardar un libro en una pila).

10. Liberar memoria (Opción 10)
Paso 1: Elige la opción 10.

- ☑ Liberará la memoria del último proceso que la usó (como sacar el libro de arriba de la pila).

11. Ver estado de la memoria (Opción 11)
Paso 1: Elige la opción 11.

- ☑ Te dirá qué procesos están ocupando memoria.

Consejos útiles
✔Prioridad: Usa números del 1 al 10 (10 = más importante).
✔Memoria: Se gestiona como una pila (LIFO) → El último en asignar memoria es el primero en liberarla.
✔Búsqueda:No necesitas el nombre exacto, el sistema buscará coincidencias.

¿Listo para empezar?
¡Eso es todo! Ahora puedes gestionar tus procesos como un profesional. Si tienes dudas, revisa el menú o prueba las opciones.

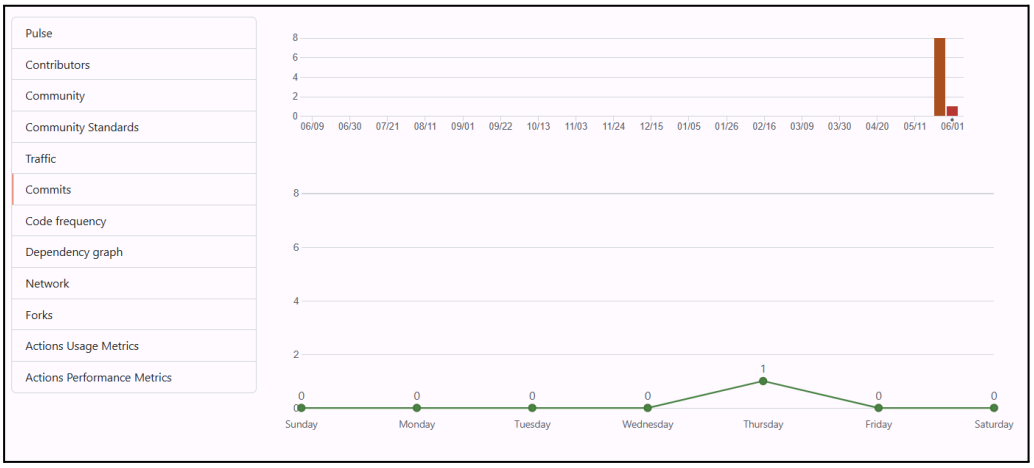
¿Algo no funciona?
Si el sistema no hace lo que esperabas, verifica que el ID sea correcto o que el proceso exista.

Capítulo 4: Evidencias de Trabajo en Equipo

1. Repositorio con Control de Versiones (Capturas de Pantalla)

· Registro de commits claros y significativos que evidencian aportes individuales (proactividad).

· Historial de ramas y fusiones si es aplicable.

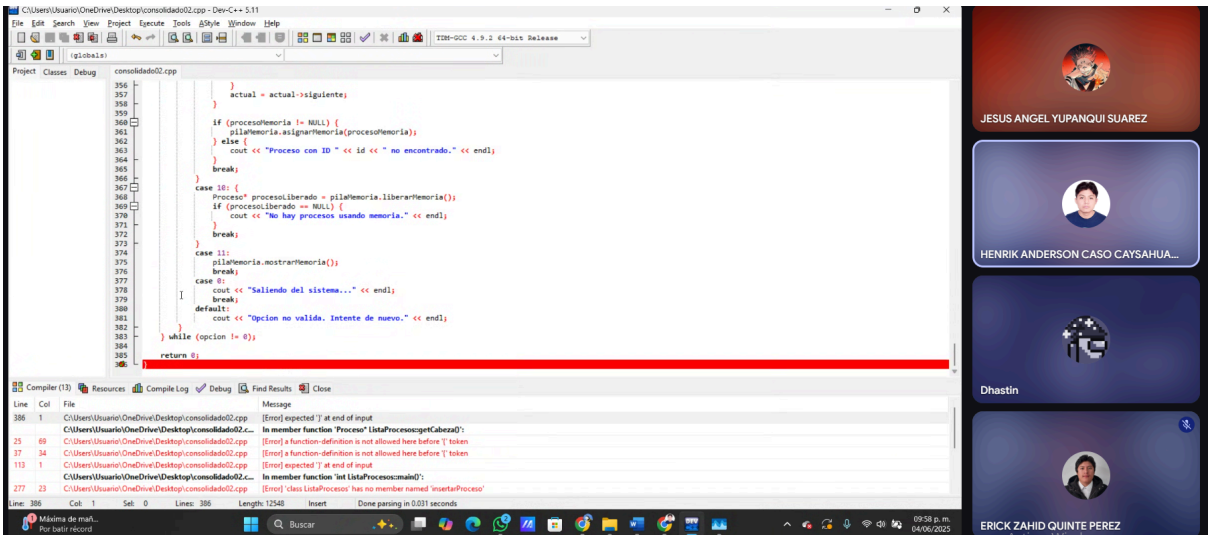


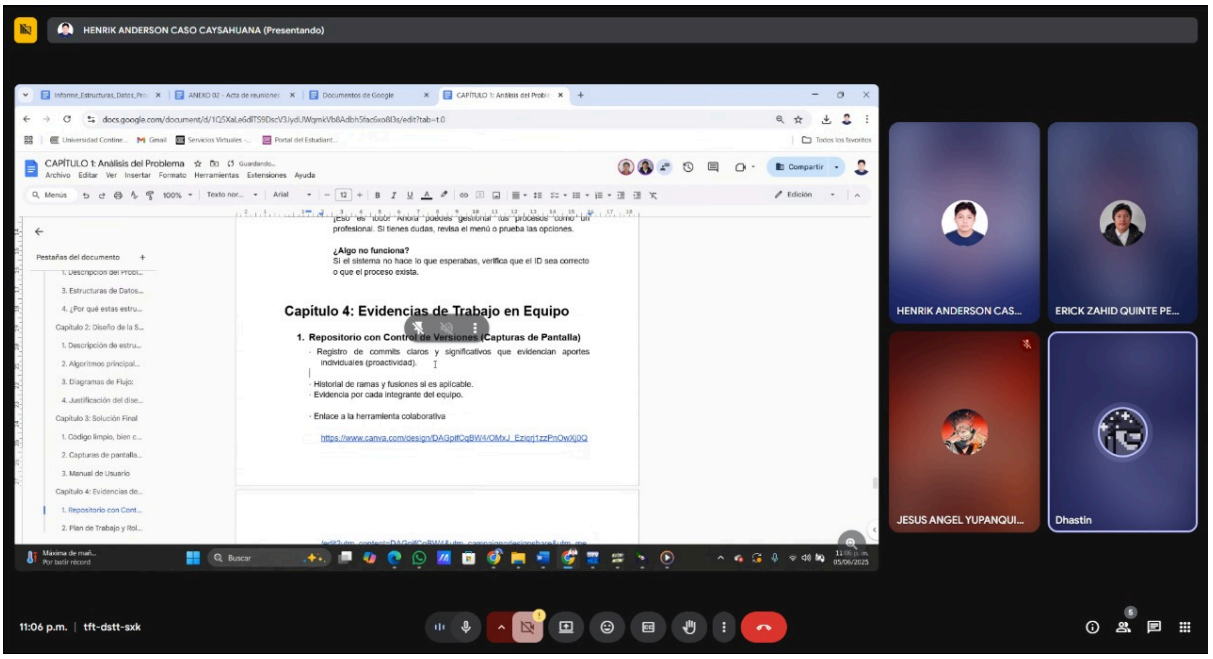
· Enlace a la herramienta colaborativa

https://www.canva.com/design/DAGpifCqBW4/OMxJ_Eziqrj1zzPnOwXj0Q/edit?utm_content=DAGpifCqBW4&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

https://github.com/zahid09190/SistemaGesti-nProcesos.git

· Evidencias de trabajo





2. Plan de Trabajo y Roles Asignados

· Documento inicial donde se asignan tareas y responsabilidades.

Asignatura	**ESTRUCTURA DE DATOS**	Fecha:	29/05/2025
Responsable de grupo	Yupanqui Suarez Jesus Angel	Hora de inicio:	15:00 PM
Modalidad de Reunión	**Virtual**	Hora de fin	11:00 PM

Integrantes:

Apellidos y nombres	Asistió (Si/No)	% Participación	Firma
1. Cabrera Ortega Dhastin Ray	Si	100%	

2. Caso Caysahuana Henrik Anderson	Si	100%	
3. Quinte Perez Erick Zahid	Si	100%	
4. Yupanqui Suarez Jesus Angel	Si	100%	

Temas tratados	Acuerdos	Responsables	Fecha de entrega
Revisión de consigna del proyecto	Todos tenian que revisar la consigna para así poder darle a cada uno su rol.	Dhastin Ray, Henrik Anderson, Erick Zahid y Jesus Angel	
Distribución de roles y tareas sobre el código	Distribución de roles para cada integrante del grupo para realizar la actividad sobre el código.	Yupanqui Suarez Jesus Angel	
Diseño de estructuras de del código	Dar todos nuestras opiniones para que quede bien diseñado el código como el Menú de opciones.	Henrik Anderson y Erick Zahid	
Redacción del documento de planificación del código.	Trabajar todos los integrantes para realizar el trabajo y ayudarnos con lo que está mal.	Dhastin Ray, Henrik Anderson, Erick Zahid y Jesus Angel	
Inicio de la programación del código	Completar el código así como los errores.	Henrik Anderson, Erick Zahid	
Definición de herramientas colaborativas	Realización y descripción de los códigos.	Yupanqui Suarez Jesus Angel	