

# Sistema de Gestión de Procesos y Memoria

Aplicación práctica de estructuras de datos dinámicas en C++

Universidad Continental de Huancayo

Curso: Estructura de Datos | Noviembre 2025



# Introducción al Proyecto

Este proyecto simula el comportamiento esencial de un sistema operativo en la administración de sus dos recursos fundamentales: **procesos y memoria**.

## Propósito Principal

Visualizar de manera clara y didáctica la gestión dinámica de recursos en un entorno de simulación.

## Conceptos Clave Aplicados

Aplicación rigurosa de estructuras como **listas enlazadas, colas de prioridad y pilas (LIFO)** para la organización interna.



# Objetivos del Sistema de Simulación

Nuestro proyecto busca demostrar la capacidad de gestionar y manipular elementos dinámicos, reflejando las tareas de un kernel básico.



## Objetivo General

Simular la gestión integral de procesos y la asignación/liberación de memoria utilizando estructuras dinámicas en C++.



## Planificación de CPU

Simular la cola de prioridad de la CPU, ejecutando siempre el proceso con el nivel de importancia más alto (prioridad).



## Manipulación de Procesos

Implementar funciones robustas para la creación, búsqueda, modificación y eliminación efectiva de procesos.



## Administración de Memoria

Gestionar la asignación de bloques de memoria mediante una pila, garantizando el principio LIFO (Último en entrar, primero en salir).

# Planteamiento del Problema y Requerimientos

La principal motivación fue la dificultad de visualizar la gestión dinámica y en tiempo real de los recursos. La solución propuesta debe ser funcional, intuitiva y modular.

## Problema Identificado

Existe una necesidad académica de comprender y **visualizar la naturaleza dinámica** de la gestión de procesos y la asignación secuencial de recursos por parte del sistema operativo.



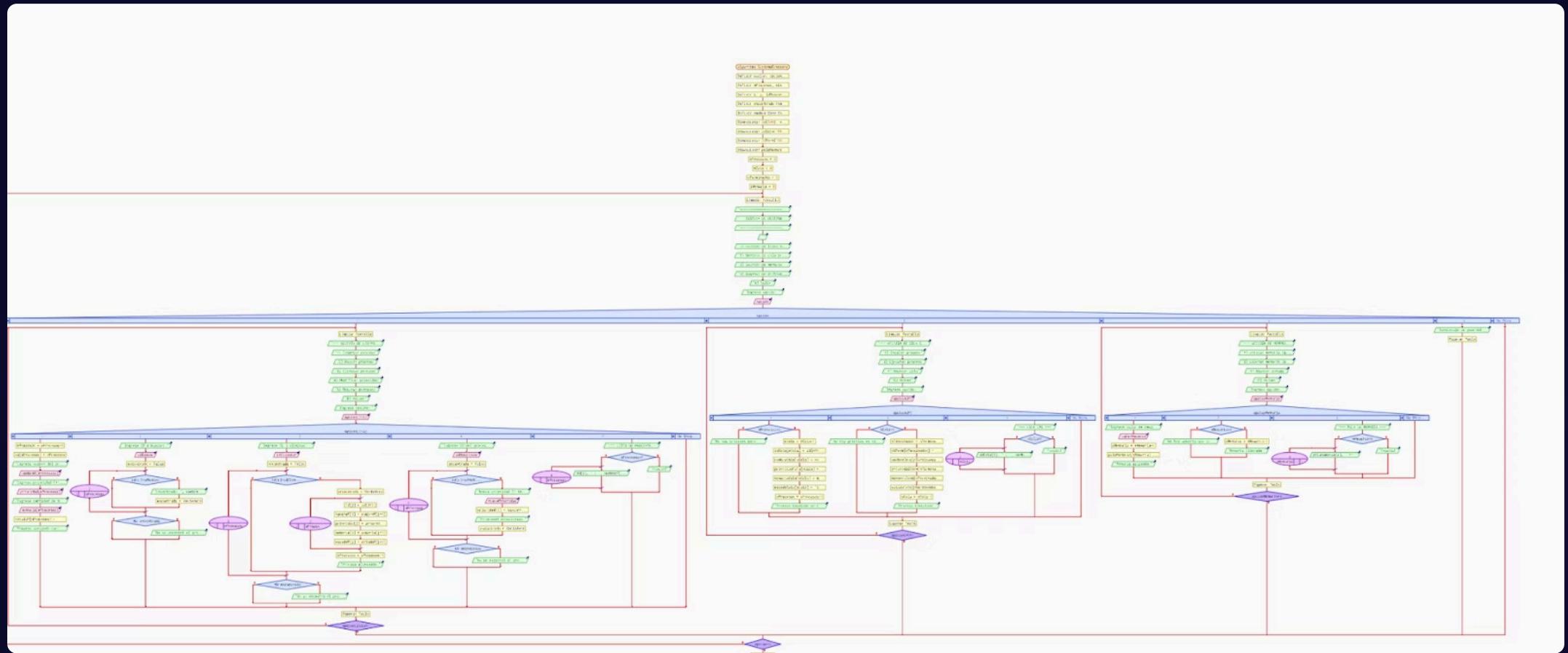
## Requerimientos Clave

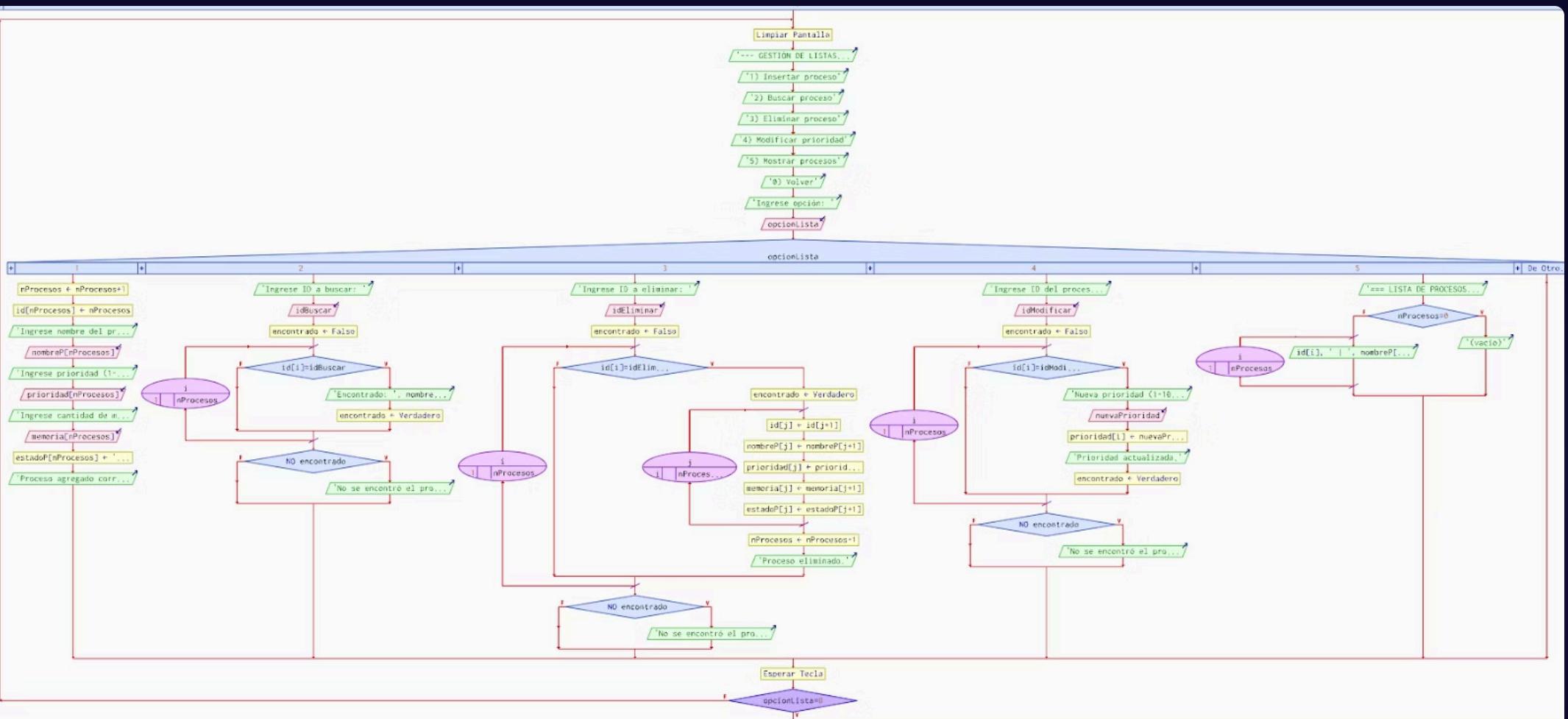
### Funcionales

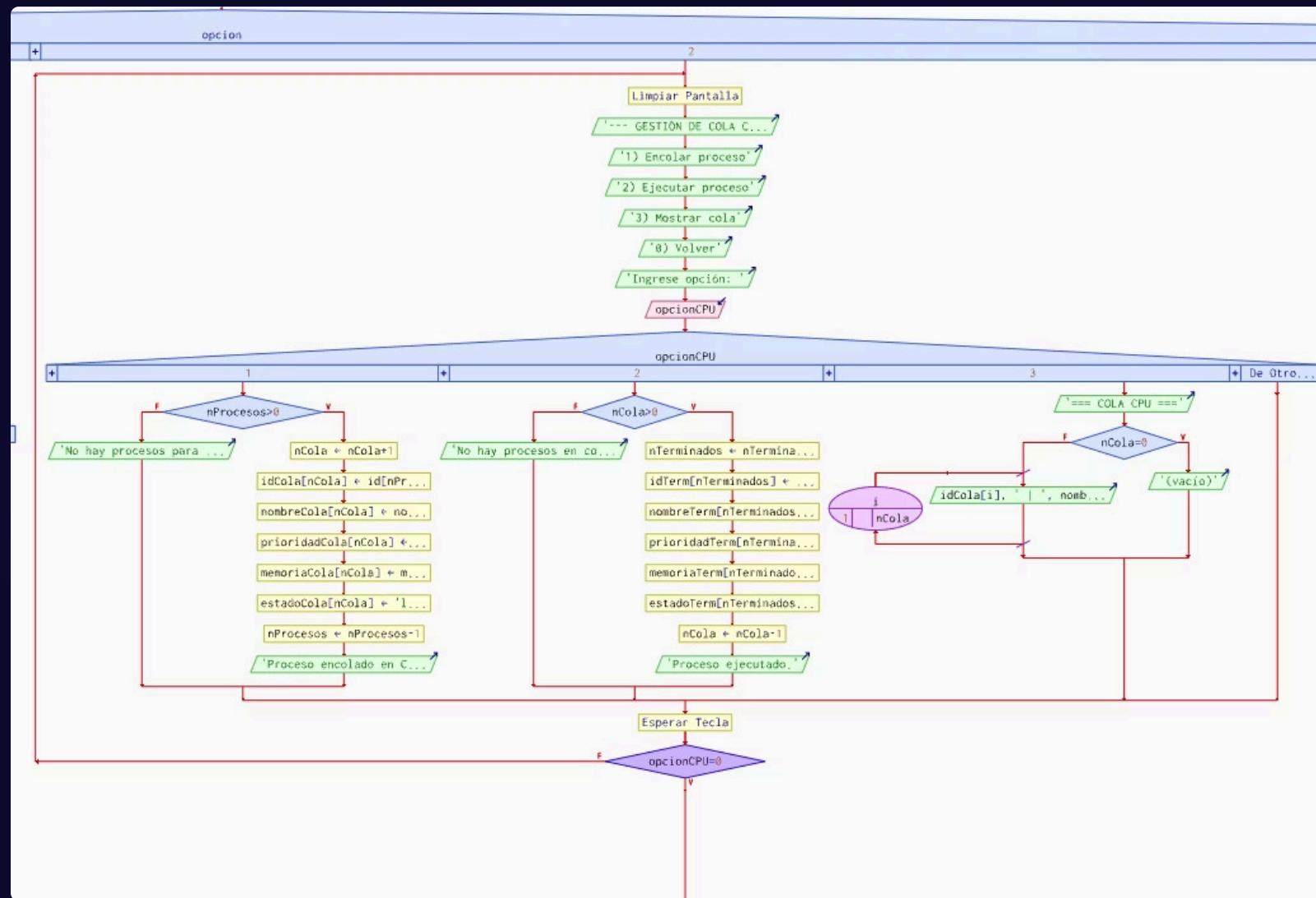
Creación, modificación de prioridad, ejecución y persistencia de procesos (guardado).

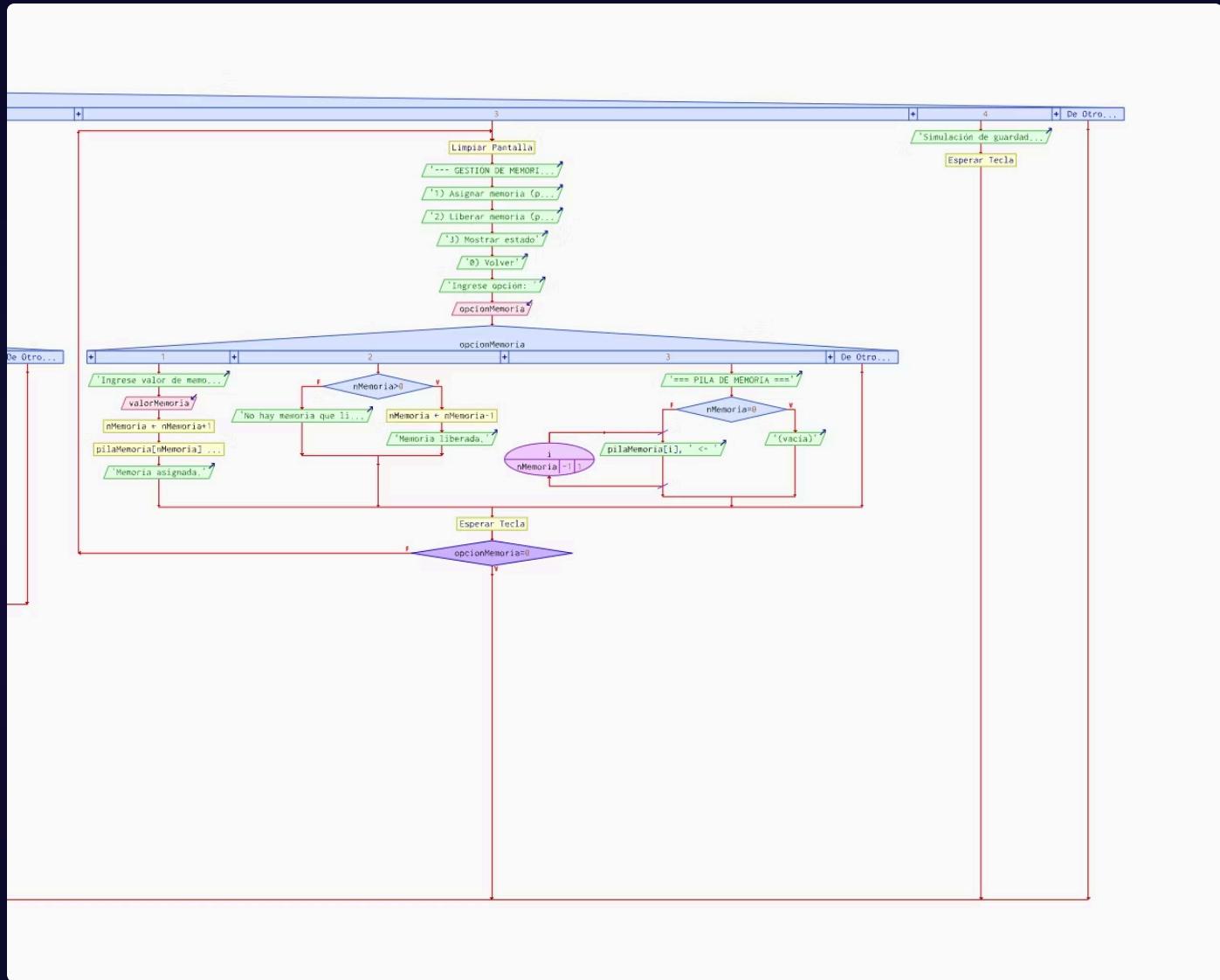
### No Funcionales

Validación de datos de entrada, interfaz de consola amigable, compatibilidad con Windows y código modular.



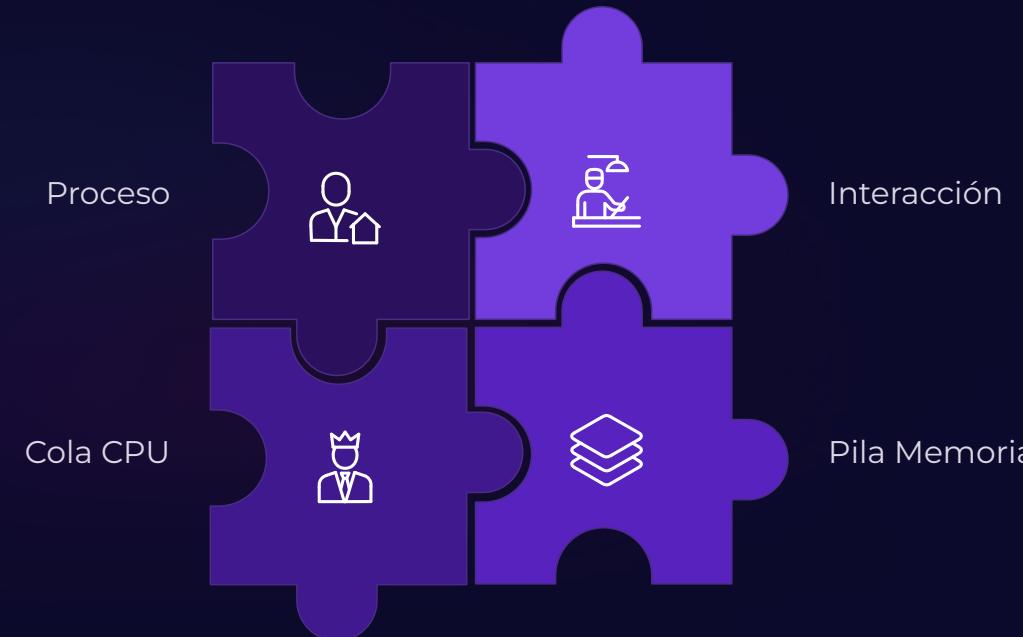






# Diseño del Sistema: Estructuras Dinámicas

El corazón del sistema se basa en la implementación de tres estructuras de datos fundamentales para modelar la realidad del sistema operativo.



Proceso	Lista Enlazada	Mantiene un registro de todos los procesos creados en el sistema.
Cola CPU	Cola Prioritaria	Organiza la ejecución de procesos según su nivel de prioridad (Scheduler).
Pila Memoria	Pila (LIFO)	Asigna y libera bloques de memoria de manera secuencial y estricta.

# Funciones Esenciales (1/3): Utilidades y Persistencia

Antes de la gestión de procesos, el código incluye módulos de soporte para interacción con el usuario y persistencia de datos.



## pausar() y limpiarPantalla()

Funciones de utilidad para mejorar la experiencia del usuario y controlar el flujo de la consola, incluyendo el uso de `system("cls")`.

```
35 void limpiarPantalla() {
36     system("cls");
37 }

28 void pausar() {
29     cout << "\nPresiona ENTER para continuar...";
30     cin.ignore();
31     cin.get();
32 }
```



## guardarEnArchivo()

Permite la persistencia de la información. Recorre la lista de procesos, la cola y la pila de memoria, escribiendo todos los datos en `procesos.txt`.

```
41 // ===== GUARDAR EN ARCHIVO =====
42
43 void guardarEnArchivo() {
44     ofstream archivo("procesos.txt");
45     archivo << "LISTA DE PROCESOS ==\n";
46     Proceso* aux = listaProcesos;
47     while (aux) {
48         archivo << aux->id << " | " << aux->nombre << " | P:"
49         << aux->prioridad << " | Mem: " << aux->memoria
50         << "MB | Estado: " << aux->estado << "\n";
51         aux = aux->sig;
52     }
53
54     archivo << "***** COLA DE CPU *****\n";
55     aux = colaCPU;
56     while (aux) {
57         archivo << aux->id << " | " << aux->nombre << " | P:"
58         << aux->prioridad << " | Mem: " << aux->memoria
59         << "MB | Estado: " << aux->estado << "\n";
60         aux = aux->sig;
61     }
62
63     archivo << "***** PROCESOS TERMINADOS *****\n";
64     aux = listaTerminados;
65     while (aux) {
66         archivo << aux->id << " | " << aux->nombre << " | P:"
67         << aux->prioridad << " | Mem: " << aux->memoria
68         << "MB | Estado: " << aux->estado << "\n";
69         aux = aux->sig;
70     }
71
72     archivo.close();
73     cout << "Los datos han sido guardados en procesos.txt";
74 }
```



## titulo()

Muestra el encabezado estandarizado del sistema en cada sección, facilitando la navegación visual del usuario.

```
39 void titulo() {
40     cout << "===== SISTEMA DE GESTIÓN DE PROCESOS Y MEMORIA =====\n";
41     cout << " ";
42     cout << "\n";
43 }
```

# Funciones Esenciales (2/3): Gestión de Procesos (Lista Enlazada)

La lista enlazada es crucial para mantener y manipular el registro maestro de todos los procesos activos en el sistema.

## insertarProceso()

Crea un nuevo proceso, asignándole un ID único, nombre, prioridad (1-10), memoria requerida y estado inicial.

```
90 // ----- LISTA PROCESOS -----
91
92 void insertarProceso() {
93     Procesc* nuevo = new Procesc();
94
95     nuevo->id = nextID++; // ID automática
96
97     cout << "\nIngrese nombre del proceso: ";
98     cin >> nuevo->nombre;
99
100    cout << "Prioridad (1 = más alta, 10 = más baja): ";
101    cin >> nuevo->prioridad;
102
103    if (nuevo->prioridad < 1 || nuevo->prioridad > 10) {
104        cout << "\nPrioridad inválida (debe ser entre 1 y 10).\n";
105        delete nuevo;
106        return;
107    }
108
109    cout << "Ingrese cantidad de memoria (1 - 4096 MB): ";
110    cin >> nuevo->memoria;
111    if (nuevo->memoria < 1 || nuevo->memoria > 4096) {
112        cout << "\nMemoria inválida (debe estar entre 1 y 4096 MB).\n";
113        delete nuevo;
114        return;
115    }
116
117    nuevo->estado = "nuevo";
118    nuevo->sig = listaProcesos;
119    listaProcesos = nuevo;
120
121    cout << "\nProceso agregado correctamente. (ID asignado: " << nuevo->id << ")";
122 }
```

## buscarProceso()

Localiza un proceso específico utilizando su identificador único para operaciones posteriores de edición o eliminación.

```
123 void buscarProcesoMenu() {
124     int id;
125     cout << "\nIngrese ID a buscar: ";
126     cin >> id;
127     Procesc* p = buscarProceso(id);
128     if (p)
129         cout << "\n Proceso encontrado: " << p->nombre << " | P: "
130             << p->prioridad << " | Mem: " << p->memoria << "MB | " << p->estado;
131     else
132         cout << "\n No existe un proceso con ese ID.";
```

## eliminarProceso()

Remueve un proceso de la lista enlazada según el ID proporcionado por el usuario.

```
135 void eliminarProceso() {
136     int id;
137     cout << "\nIngrese ID a eliminar: ";
138     cin >> id;
139
140     Procesc* aux = listaProcesos, *ant = NULL;
141     while (aux != NULL) {
142         if (aux->id == id) {
143             if (ant) ant->sig = aux->sig;
144             else listaProcesos = aux->sig;
145             delete aux;
146             cout << "\n Proceso eliminado.";
147             return;
148         }
149         ant = aux;
150         aux = aux->sig;
151     }
152     cout << "\n Proceso no encontrado.";
153 }
```

## modificarPrioridad()

Permite cambiar el valor de prioridad de un proceso existente, con validaciones estrictas de rango.

```
155 void modificarPrioridad() {
156     int id;
157     cout << "\nIngrese ID del proceso: ";
158     cin >> id;
159
160     Procesc* p = buscarProceso(id);
161     if (!p) {
162         cout << "\n Proceso no encontrado.";
163         return;
164     }
165
166     int nueva;
167     cout << "Nueva prioridad (1-10): ";
168     cin >> nueva;
169
170     if (nueva < 1 || nueva > 10) {
171         cout << "\n Valor inválido.";
172         return;
173     }
174     p->prioridad = nueva;
175     cout << "\n Prioridad actualizada.";
176 }
```



# Funciones Esenciales (3/3): CPU y Memoria

Estos módulos simulan el **Scheduler** (planificador) y el **Memory Manager** (administrador de memoria) del sistema operativo.

## Gestión de CPU (Cola Prioritaria)

- **encolarCPU()**: Mueve el proceso de mayor prioridad de la lista a la cola de ejecución.

```
192 void encolarCPU() {
193     if (!listaProcesos) {
194         cout << "\nNo hay procesos disponibles." << endl;
195         return;
196     }
197
198     Proceso* aux = listaProcesos, *mejor = aux, *ant = NULL, *antMejor = NULL;
199     while (aux) {
200         if (aux->prioridad < mejor->prioridad) {
201             mejor = aux;
202             antMejor = ant;
203             aux = aux->sig;
204         } else {
205             aux = aux->sig;
206         }
207
208         if (antMejor == NULL) {
209             aux = listaProcesos->sig;
210         } else {
211             antMejor->sig = aux;
212             aux = colapCPU;
213             colapCPU = aux;
214         }
215     }
216     cout << "\nProceso encolado en CPU." << endl;
217 }
```

- **ejecutarCPU()**: Simula la ejecución. Saca el primer proceso de la cola y lo marca como *terminado* en la lista maestra.

```
234 void mostrarColaCPU() {
235     cout << "\n==== COLA CPU ===\n";
236     Proceso* aux = colapCPU;
237     if (!aux) cout << "(vacio)\n";
238     while (aux) {
239         cout << aux->id << " | " << aux->nombre << " | P: "
240             << aux->prioridad << " | Mem: " << aux->memoria
241             << "#B" | " << aux->estado << "\n";
242         aux = aux->sig;
243     }
244 }
```

- **mostrarColaCPU()**: Visualiza los procesos en espera, ordenados por prioridad.

```
234 void mostrarColaCPU() {
235     cout << "\n==== COLA CPU ===\n";
236     Proceso* aux = colapCPU;
237     if (!aux) cout << "(vacio)\n";
238     while (aux) {
239         cout << aux->id << " | " << aux->nombre << " | P: "
240             << aux->prioridad << " | Mem: " << aux->memoria
241             << "#B" | " << aux->estado << "\n";
242         aux = aux->sig;
243     }
244 }
```

## Gestión de Memoria (Pila LIFO)

- **pushMemoria()**: Asigna un nuevo bloque de memoria (PUSH).

```
void pushMemoria() {
    int v;
    cout << "\nIngrese el valor de memoria: ";
    cin >> v;
    PilaMemoria* nuevo = new PilaMemoria();
    nuevo->valor = v;
    nuevo->sig = pilaMemoria;
    pilaMemoria = nuevo;
    cout << "\nMemoria asignada.";
```

- **popMemoria()**: Libera el último bloque de memoria asignado (POP), manteniendo la disciplina LIFO.

```
void popMemoria() {
    if (!pilaMemoria) {
        cout << "\nNo hay memoria que liberar." << endl;
        return;
    }
    PilaMemoria* temp = pilaMemoria;
    pilaMemoria = pilaMemoria->sig;
    delete temp;
    cout << "\nMemoria liberada.";
```

- **mostrarMemoria()**: Muestra la ocupación actual de la pila de memoria.

```
270 void mostrarMemoria() {
271     cout << "\n==== ESTADO DE LA PILA DE MEMORIA ===\n";
272     PilaMemoria* aux = pilaMemoria;
273     if (!aux) cout << "(vacio)\n";
274     while (aux) {
275         cout << aux->valor << " - ";
276         aux = aux->sig;
277     }
278     cout << "\n";
```

# Estructura de Menús y Ejecución

El sistema está organizado en un menú principal y tres submenús, garantizando una navegación clara y específica para cada función de gestión.



## Menú Principal (main)

Punto de acceso a las tres áreas de gestión (Lista, CPU, Memoria) y a la función de guardado/salida del sistema.

```
359 int main() {
360     setlocale(LC_CTYPE, "Spanish");
361     int op;
362     do {
363         limpiarPantalla();
364         titulo();
365         cout << "\n===== MENÚ PRINCIPAL =====\n";
366         cout << "1) Gestión de listas de procesos\n";
367         cout << "2) Gestión de cola prioridad (CPU)\n";
368         cout << "3) Gestión de memoria (pila)\n";
369         cout << "4) Guardar todo en archivo\n";
370         cout << "0) Salir\n> ";
371         cout << "Ingrese opción (0-4):";
372         cin >> op;
```



## menuLista()

Dedicado a la administración CRUD (Crear, Buscar, Modificar, Eliminar) y visualización de todos los procesos en el sistema.

```
283 void menuLista() {
284     int op;
285     do {
286         limpiarPantalla();
287         titulo();
288         cout << "\n--- GESTIÓN DE LISTAS DE PROCESOS ---\n";
289         cout << "1) Insertar proceso\n";
290         cout << "2) Buscar proceso\n";
291         cout << "3) Eliminar proceso\n";
292         cout << "4) Modificar prioridad\n";
293         cout << "5) Mostrar procesos\n";
294         cout << "0) Volver\n> ";
295         cout << "Ingrese opción (0-5):";
296         cin >> op;
297 }
```



## menuCPU()

Permite el control de la simulación del planificador: encolar procesos, ejecutar el proceso prioritario y verificar el estado de la cola.

```
void menuCPU() {
    int op;
    do {
        limpiarPantalla();
        titulo();
        cout << "\n--- GESTIÓN COLA PRIORIDAD (CPU) ---\n";
        cout << "1) Encolar proceso\n";
        cout << "2) Ejecutar proceso (desencolar)\n";
        cout << "3) Mostrar cola\n";
        cout << "0) Volver\n> ";
        cout << "Ingrese opción (0-3):";
        cin >> op;
```



## menuMemoria()

Gestión directa de la pila: incluye las operaciones de asignación (push) y liberación (pop) de los bloques de memoria.

```
void menuMemoria() {
    int op;
    do {
        limpiarPantalla();
        titulo();
        cout << "\n--- GESTIÓN DE MEMORIA (PILA) ---\n";
        cout << "1) Asignar memoria (push)\n";
        cout << "2) Liberar memoria (pop)\n";
        cout << "3) Mostrar estado de la memoria\n";
        cout << "0) Volver\n> ";
        cout << "Ingrese opción (0-3):";
        cin >> op;
```

# Resultados, Conclusiones y Futuras Mejoras

El proyecto no solo cumple con la simulación, sino que también demostró la efectividad del trabajo en equipo y la correcta aplicación de algoritmos.

## Pruebas y Validación

- Verificación de la lógica de prioridad en la cola de CPU.
- Validación de límites de entrada (e.g., Prioridad entre 1 y 10).
- Comprobación del guardado correcto de la información en procesos.txt.

## Recomendaciones Futuras

### Automatización

Implementar la carga inicial de procesos desde el archivo de persistencia.

### Evolución

Migrar el código de C a C++ moderno, utilizando Programación Orientada a Objetos (POO).

## Conclusiones Finales

El sistema desarrollado cumple los objetivos: simular la gestión de recursos y validar la aplicación de estructuras dinámicas de datos.

El trabajo colaborativo, usando herramientas como GitHub, fue fundamental para el éxito del proyecto.

