GESTIÓN DE PROCESOS

ESTRUCTURAS DINÁMICAS

GRUPO

- Mijael Joseph Bejarano Miche
- Nelinho Capcha Hidalgo
- Luis Enrique Cueva Barrera
- Damian Javier Lopez Naula

A

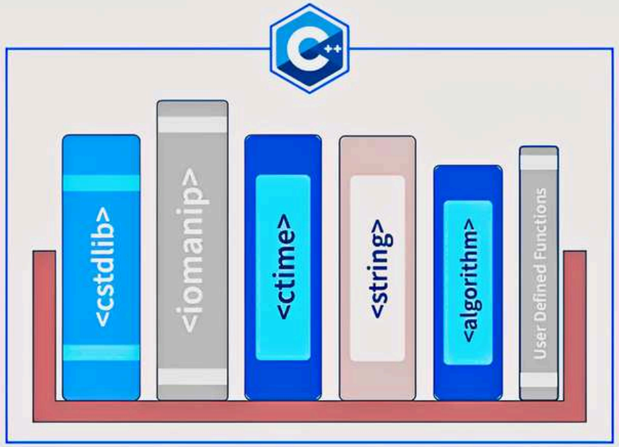
DEFINICIONES PRELIMINARES

1. BIBLIOTECAS

`<iostream>` → Sirve para realizar **operaciones de entrada y salida (Input/Output)**, como leer datos del teclado o mostrar información en la pantalla.

`<fstream>` → Se utiliza para trabajar **con archivos: leer, escribir o ambos,** permitiendo **guardar la información de los procesos** vinculada a la lista enlazada de forma persistente.

`<string>` → Sirve para **trabajar con cadenas de texto de forma más sencilla** y segura que con los arreglos de caracteres (char[]).



```
// 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
    Proceso* siguiente;     // Puntero al siguiente proceso en la lis

    // 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
    }
};
```

2. ESTRUCTURAS

struct Proceso

id	nombre	prioridad	siguiente
_id	_nombre	_prioridad	-> NULL

struct NodoMemoria

procesoid	siguiente
_	->

```
// Nodo para representar un proceso en la pila de memoria
struct NodoMemoria {
    string procesoID;
    NodoMemoria* siguiente;
};
```

struct NodoCola

```
// Nodo para representar una cola de prioridad
struct NodoCola {
    string id;
    string nombre;
    int prioridad;
    NodoCola* siguiente;
};
```

id	nombre	prioridad	siguiente
_id	_nombre	_prioridad	-›

C. VARIABLES GLOBALES

`Proceso* listaProcesos = NULL;`

Puntero al inicio de una lista enlazada de procesos. Guarda todos los procesos registrados, estén o no en ejecución.

`NodoMemoria* pilaMemoria= NULL;`

Puntero al tope de una pila (stack) que simula la memoria. Simula la asignación y liberación de memoria con los procesos.

`NodoCola* colaCPU= NULL;`

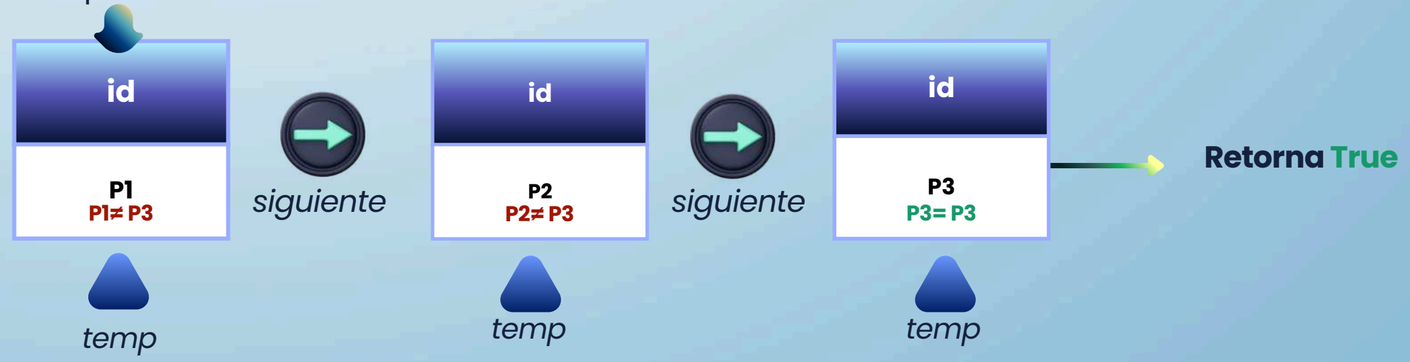
Puntero al frente de una cola de prioridad. Representa la cola de ejecución de la CPU.

3. FUNCIONES AUXILIARES

```
// Verifica si un proceso 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
}
```

bool procesoYaEnCola

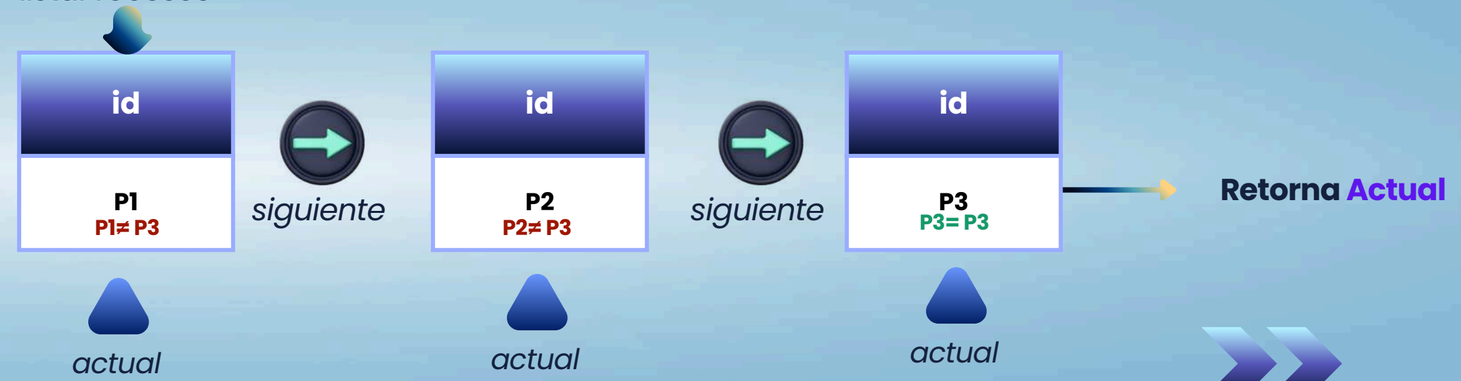
Búsqueda del Proceso P3
colaCpu



Proceso* buscarProcesoPorID

```
// Busca un proceso por su ID
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
}
```

Búsqueda del Proceso P3
listaProcesos



void guardarProcesosEnArchivo

Creación del archivo “procesos.txt”
Lista enlazada de procesos

id	nombre	prioridad
P2	Navegador	4

siguiente

id	nombre	prioridad
P3	Editor	10

actual

procesos.txt

ID: P2, **Nombre:** Navegador, **Prioridad:** 4
ID: P3, **Nombre:** Editor, **Prioridad:** 10

actual

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

void cargarProcesosDesdeArchivo

procesos.txt

ID: P2, **Nombre:** Navegador, **Prioridad:** 4
ID: P3, **Nombre:** Editor, **Prioridad:** 10

id	nombre	prioridad
P2	Navegador	4

```cpp
// 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 linea por linea
    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
}
```

B

LISTAS ENLAZADAS

1.INSERTAR PROCESOS

id
P7

if buscarProcesoPorID

P7≠ ∀ ID la lista → sigue

P7=P7 → *"Ya existe"*

do

id	nombre	prioridad
P7	Firefox	4

if **prioridad<1 o prioridad >10** → *"Error (entero 1 al 10)"*

```
// 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);
```

`Proceso* nuevo`

new

id	nombre	prioridad
P7	Firefox	4

siguiente
-> NULL

```
// 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();
system("pause");
system("CLS");
```

if

listaProcesos= NULL

id	nombre	prioridad
P7	Firefox	4

siguiente
->

listaProcesos

else

listaProcesos≠NULL

listaProcesos

id	nombre	prioridad
P2	Vcode	7

siguiente

id	nombre	prioridad
P7	Firefox	4

siguiente

actual

actual

`guardarProcesosEnArchivo` → `system("pause")` → `system("CLS")`

2.INSERTAR PROCESOS

`if` **listaProcesos= NULL** → *"No hay procesos registrados"*

`else`

listaProcesos≠NULL
listaProcesos

```
// 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");
}
```

`cout<<`

id	nombre	prioridad	siguiente
P2 →	VCode →	7	→
actual	*actual*	*actual*	*actual*

Recorrer todos los procesos

Opción 1

ID: P2 | **Nombre:** VCode | **Prioridad:** 7
ID: P3 | **Nombre:** Firexfox | **Prioridad:** 10

...

3.ELIMINAR PROCESOS

`Proceso* actual` → **listaProcesos**

`Proceso* anterior` → **NULL**

Mientras actual exista (no sea NULL) y **el id del proceso actual no sea igual al que estoy buscando.**

Búsqueda ID=P7 // listaProcesos


```
// 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;
    }
```

if → *actual* **= NULL** → *"Proceso no encontrado"*

```
    // 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");
}
```

if → `Proceso* anterior` **=NULL**

Búsqueda ID=P2
listaProcesos

id	nombre	prioridad
P2	VCode	7

siguiente

id	nombre	prioridad
P3	Firefox	4

anterior *actual*

id	nombre	prioridad
P2	VCode	4

siguiente

else → `Proceso* anterior` **≠NULL**

Búsqueda ID=P3 // listaProcesos

id	nombre	prioridad
P2	VCode	7

siguiente

anterior

id	nombre	prioridad
P3	Firefox	4

actual

actual
anterior

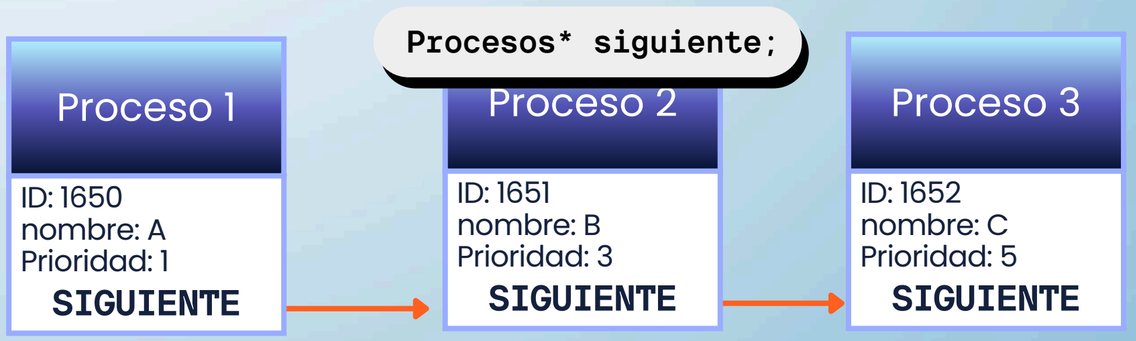
id	nombre	prioridad
P2	VCode	7

siguiente

anterior

4. Buscar Procesos por ID

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

    Proceso(string _id, string _nombre, int _prioridad) {
        id = _id;
        nombre = _nombre;
        prioridad = _prioridad;
        siguiente = NULL;
    }
};
```




```
void buscarProceso() {
    string id;
    cout << "Ingrese el ID del proceso a buscar: ";
    cin >> id;

    Proceso* resultado = buscarProcesoPorID(id);
    if (resultado) {
        cout << "Proceso encontrado:\n";
        cout << "ID: "
        << resultado->id << " | Nombre: " << resultado->nombre
        << " | Prioridad: " << resultado->prioridad << "\n";
    } else {
        cout << "Proceso no encontrado.\n";
    }
    system("pause");
    system("CLS");
}
```

Se ingresa el ID del proceso a Buscar

En caso de encontrar el proceso, mostrará el ID, nombre y prioridad

En caso de ingresar un ID que no existe o incorrecto lanzará un mensaje del proceso no encontrado y se cerrará el programa

```
ListarProcesos();
```

Proceso 1
ID: 1652
nombre: A
Prioridad: 1

Proceso 2
ID: 1551
nombre: B
Prioridad: 5

Proceso 3
ID: 1652
nombre: C
Prioridad: 8

NULL

ID INGRESADO: 1551

```
Proceso* resultado = buscarProcesoPorID(id);
```

Resultado = ID Ingresado

Se compara con el primer Nodo

If

Proceso 1
ID: **1652**
nombre: A
Prioridad: 5

Else

Proceso no encontrado

If

Se compara con el primer Nodo

Proceso 2
ID: **1551**
nombre: B
Prioridad: 5

Else

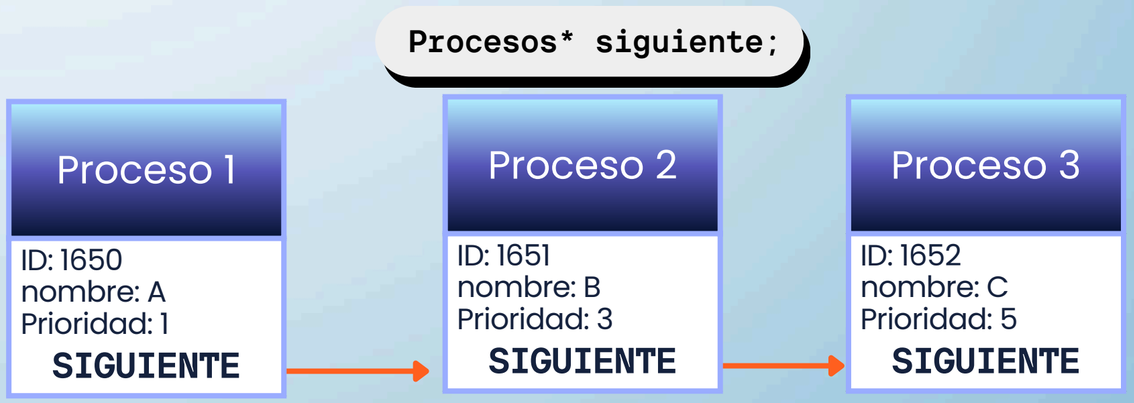
Proceso encontrado

Proceso 2
ID: **1551**
nombre: B
Prioridad: 5

5. Modificar Prioridad de un Proceso

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

    Proceso(string _id, string _nombre, int _prioridad) {
        id = _id;
        nombre = _nombre;
        prioridad = _prioridad;
        siguiente = NULL;
    }
};
```

Procesos* siguiente;




```
void modificarPrioridad() {
    string id;
    cout << "Ingrese el ID del proceso: ";
    cin >> id;

    Proceso* p = buscarProcesoPorID(id);
    if (!p) {
        cout << "Proceso no encontrado.\n";
        system("pause");
        system("CLS");
        return;
    }
```

Busca el proceso con el ingreso de ID para modificar la prioridad

En caso de no encontrarlo, saldra un mensaje de error y mandara al menu principal

```
    int nuevaPrioridad;
    cout << "Prioridad actual: " << p->prioridad << "\n";
    do {
        cout << "Ingrese nueva prioridad: ";
        cin >> nuevaPrioridad;
        if (nuevaPrioridad<1 || nuevaPrioridad>10) {
                cout<<"\tError (entero del 1 al 10) .... \n";
        }
    } while (nuevaPrioridad<1 || nuevaPrioridad>10);

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

Se encontrará el proceso y mostrará la prioridad actual para luego ingresar una prioridad nueva de valor entero (1-10)

Si ingresó correctamente la prioridad del 1 al 10 y entero, saldrá mensaje de prioridad modificada y se pausara para luego ir al menu

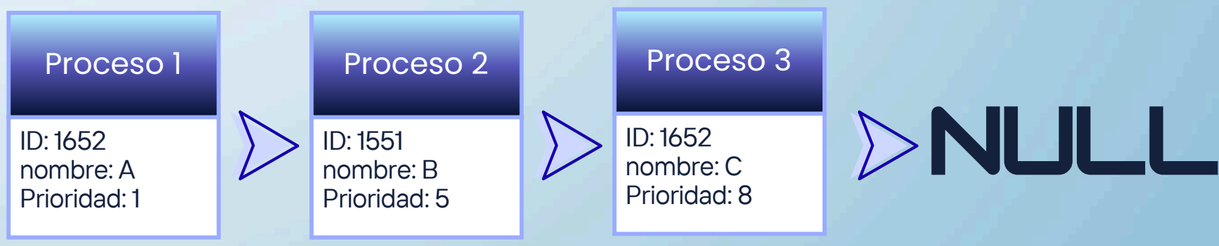
guardarProcesosEnArchivo();

El proceso modificado se guardará en la memoria

Ingresar el ID a modificar: 1550

`buscarProcesoPorID(id);`

PROCESOS ANTES DE LA MODIFICACIÓN



`Proceso* resultado = buscarProcesoPorID(id);`

If

Se compara con el primer Nodo

Proceso 1
ID: **1652**
nombre: A
Prioridad: 5

Else

Proceso no encontrado

If

Se compara con el primer Nodo

Proceso 2
ID: **1551**
nombre: B
Prioridad: 5

Else

Proceso encontrado

Proceso 2 | ID: **1551** nombre: B Prioridad: 5

`int nuevaPrioridad;` **Valor entero 1-10**

If

Proceso 2 | ID: **1551** nombre: B Prioridad: **14**

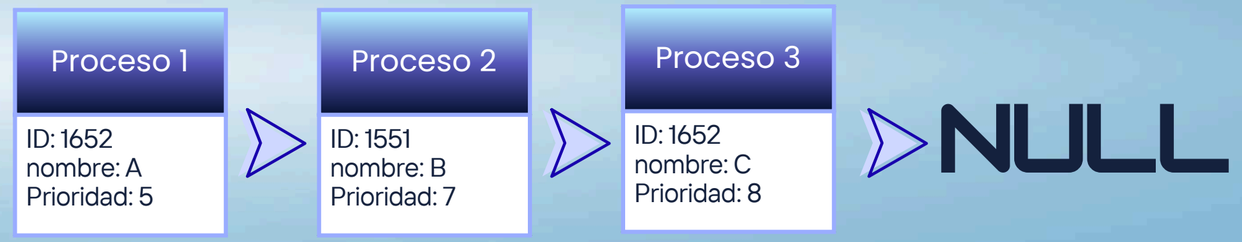
error (entero 1 -10)

If

Proceso 2 | ID: **1551** nombre: B Prioridad: **7**

Proceso modificado

PROCESOS DESPUÉS DE LA MODIFICACIÓN

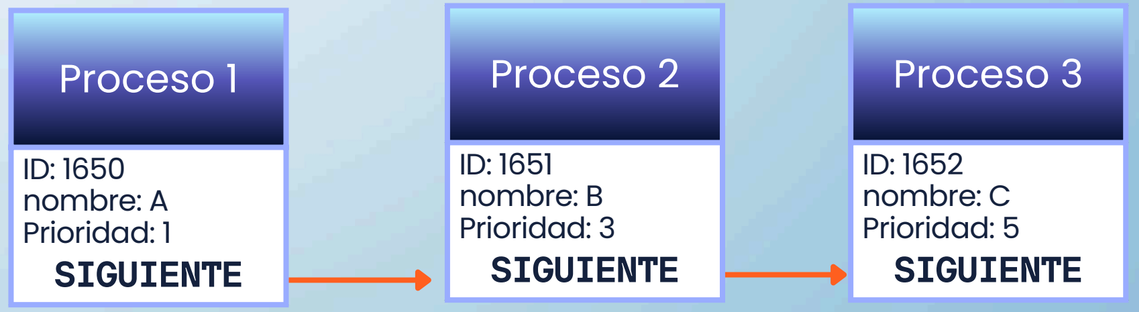


6. Ordenar los Procesos por Prioridad

[MAYOR - MENOR]

Procesos* siguiente;

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

    Proceso(string _id, string _nombre, int _prioridad) {
        id = _id;
        nombre = _nombre;
        prioridad = _prioridad;
        siguiente = NULL;
    }
};
```




```
void ordenarProcesos() {
    if (!listaProcesos || !listaProcesos->siguiente) {
        cout << "No hay suficientes procesos para ordenar.\n";
        system("pause");
        system("CLS");
        return;
    }
```

Busca si existen procesos en el void listaProcesos, en caso de que no exista saldrá error y cerrara la opción

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

Se encontrarón procesos, ahora recorre los nodos de todas las listas para comparar las prioridades verificando si actual->prioridad es menor que siguiente->prioridad, si es menor se intercambian los nodos

La variable anterior ayuda a mentener la conexión entre los nodos tras el intercambio realizado

La variable intercambio se coloca como verdadero para avanzar al siguiente ciclo

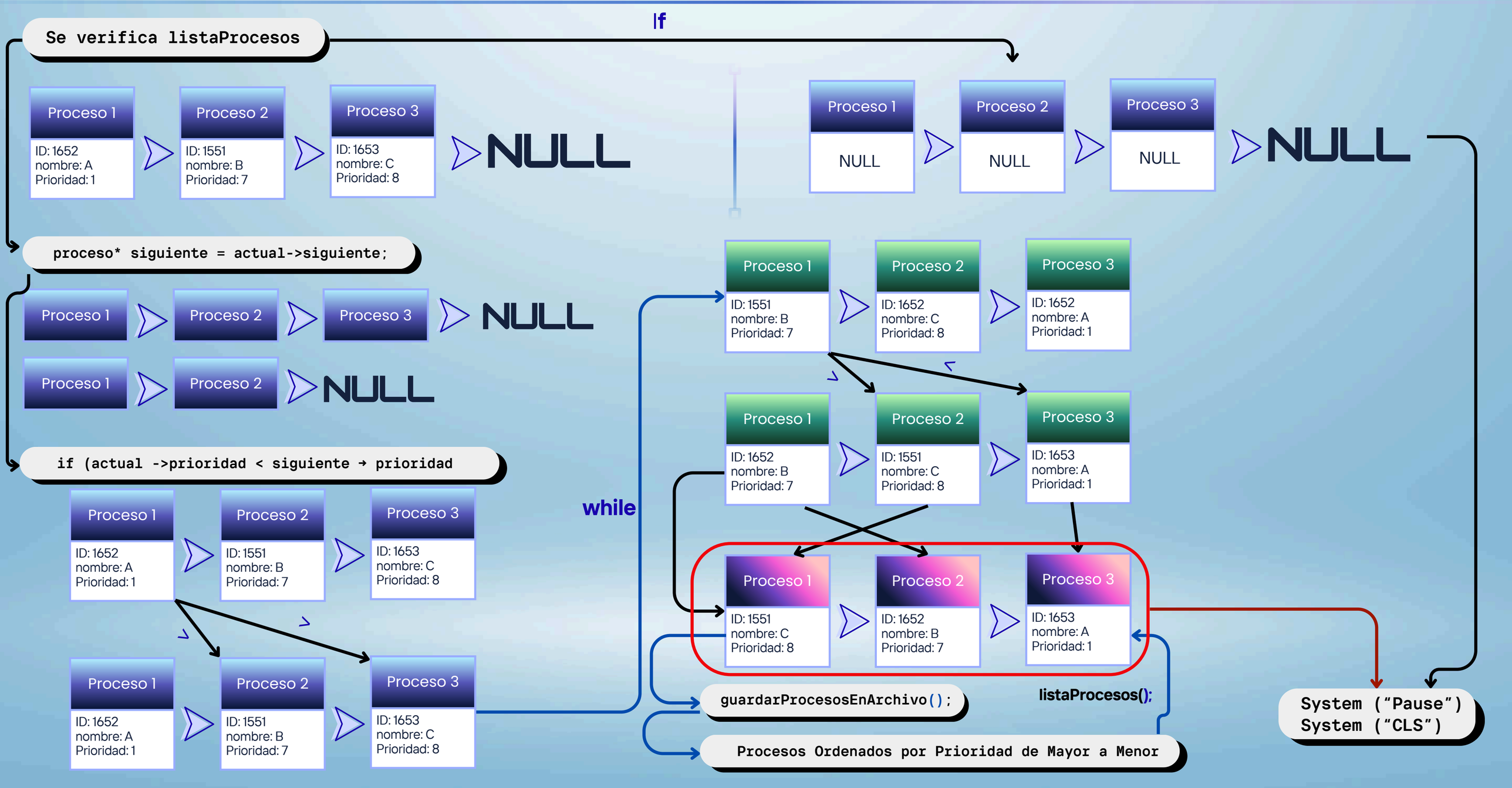
```
guardarProcesosEnArchivo(); // Guarda el nuevo orden
cout << "Procesos ordenados por prioridad (Mayor -> Menor).\n";
listarProcesos(); //mostramos la lista ordenada
system("pause");
system("CLS");
```

Luego del ordenamiento se almacena en:

`guardarProcesosEnArchivo();`

Finalmente muestra la lista con:

`listaProcesos();`


Se verifica listaProcesos
If
Proceso 1
ID: 1652
nombre: A
Prioridad: 1
Proceso 2
ID: 1551
nombre: B
Prioridad: 7
Proceso 3
ID: 1653
nombre: C
Prioridad: 8
NULL
Proceso 1
NULL
Proceso 2
NULL
Proceso 3
NULL
NULL
proceso* siguiente = actual->siguiente;
Proceso 1
Proceso 2
Proceso 3
NULL
Proceso 1
Proceso 2
NULL
if (actual ->prioridad < siguiente → prioridad
while
Proceso 1
ID: 1551
nombre: B
Prioridad: 7
Proceso 2
ID: 1652
nombre: C
Prioridad: 8
Proceso 3
ID: 1652
nombre: A
Prioridad: 1
Proceso 1
ID: 1652
nombre: B
Prioridad: 7
Proceso 2
ID: 1551
nombre: C
Prioridad: 8
Proceso 3
ID: 1653
nombre: A
Prioridad: 1
Proceso 1
ID: 1551
nombre: C
Prioridad: 8
Proceso 2
ID: 1652
nombre: B
Prioridad: 7
Proceso 3
ID: 1653
nombre: A
Prioridad: 1
Proceso 1
ID: 1652
nombre: A
Prioridad: 1
Proceso 2
ID: 1551
nombre: B
Prioridad: 7
Proceso 3
ID: 1653
nombre: C
Prioridad: 8
guardarProcesosEnArchivo();
listaProcesos();
Procesos Ordenados por Prioridad de Mayor a Menor
System ("Pause")
System ("CLS")

COLA CPU

7. Encolar

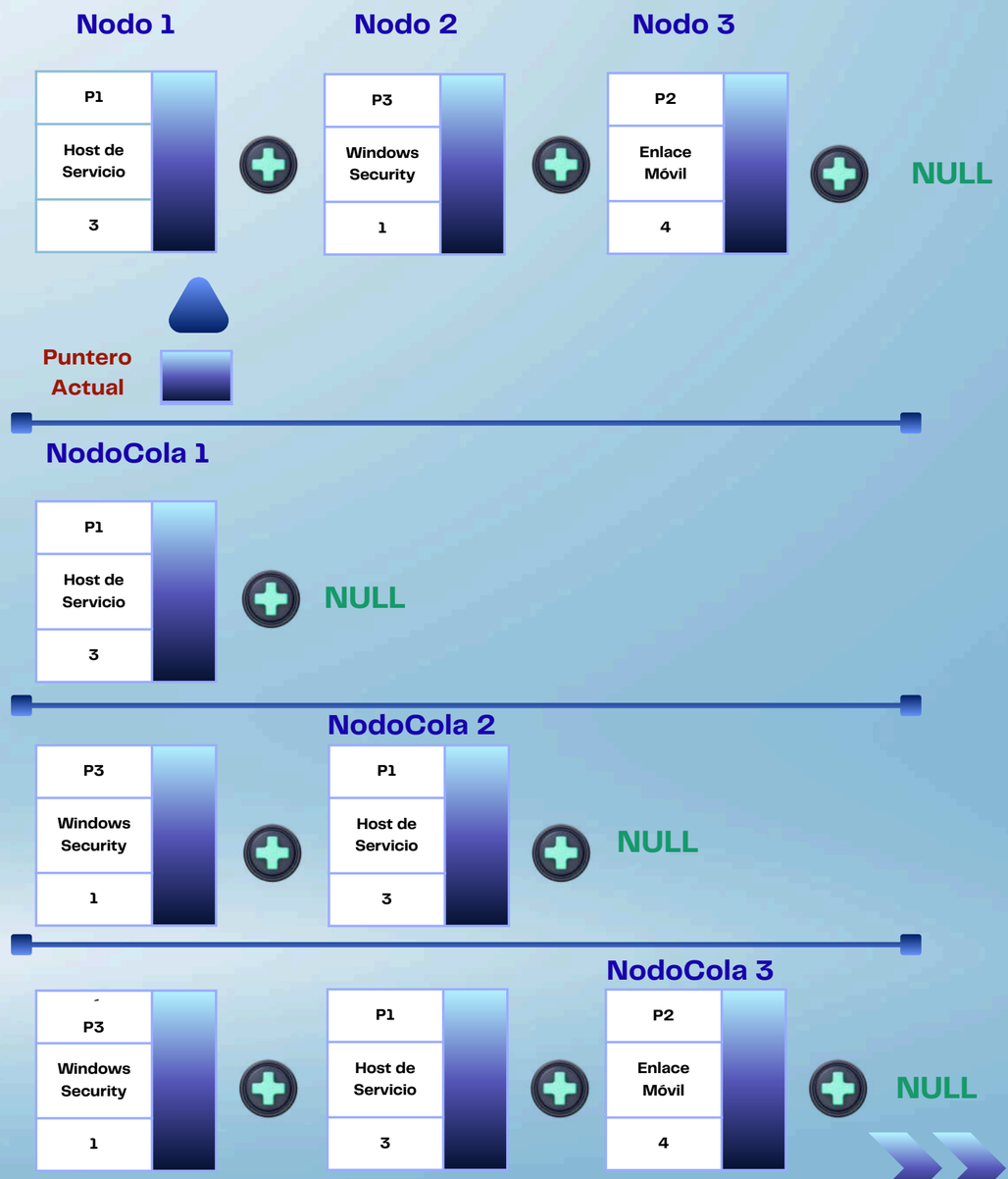
`struct ListaProcesos`

```
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;
    }
```

Asignamos el puntero “actual” a nuestra lista de procesos

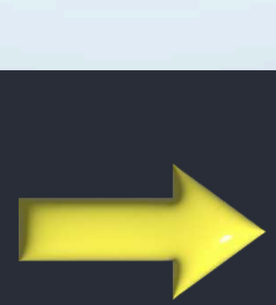
Se crea un Nodo (nuevo), para la cola

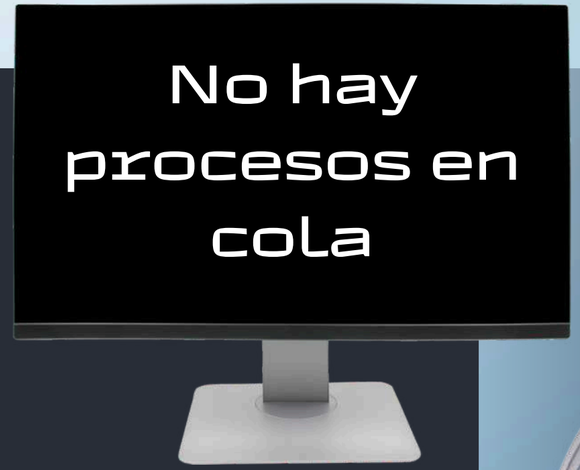
`struct NodoCola`



8. Desencolar

```
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";
    cout << "ID: " << temp->id << " | Nombre: " << temp->nombre << " | Prioridad: " << temp->prioridad << "\n";
    delete temp;    // Libera memoria del proceso ejecutado
    system("pause");
    system("CLS");
}
```





`struct NodoCola`

NodoCola 1

P3
Windows Security
1

NodoCola 2

P1
Host de Servicio
3

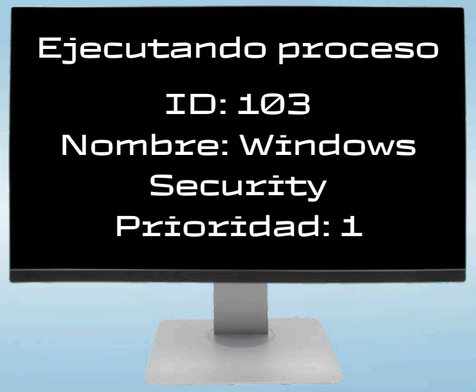
NodoCola 3

P2
Enlace Móvil
4

NULL

Puntero TEMP

Puntero colaCPU



NodoCola 1

P1
Host de Servicio
3

NodoCola 2

P2
Enlace Móvil
4

NULL

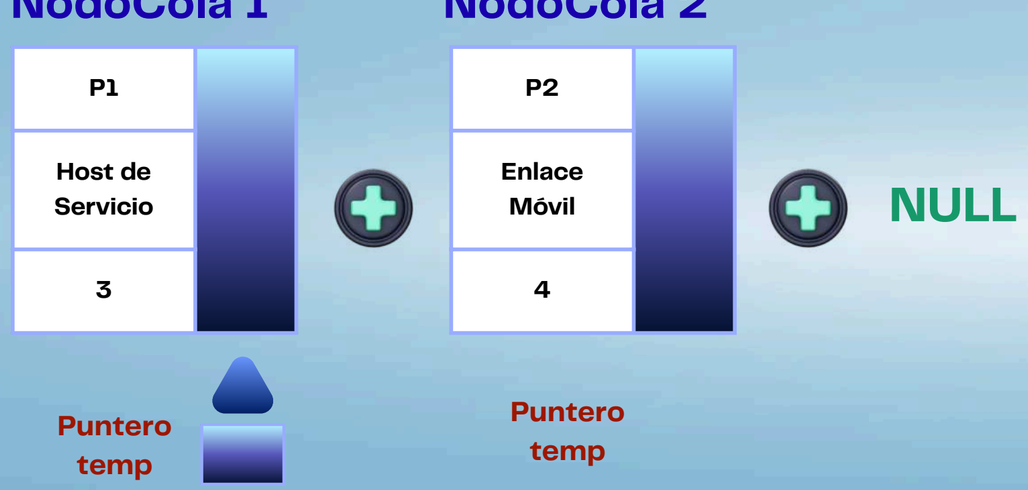
Puntero colaCPU

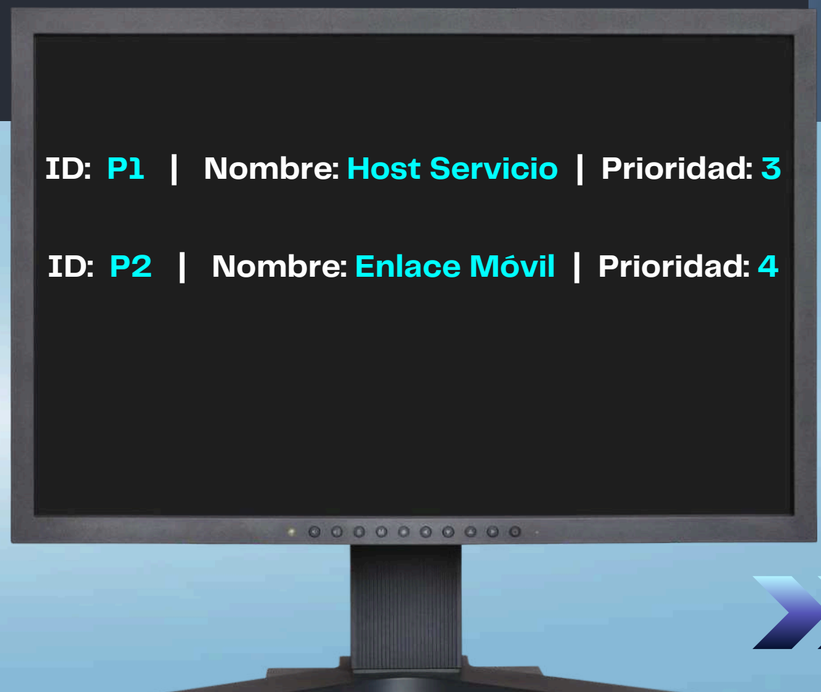
9. Mostrar cola

```
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");
}
```

Verifica si la cola está vacía. Imprime si es el caso

El puntero se mueve al siguiente nodo

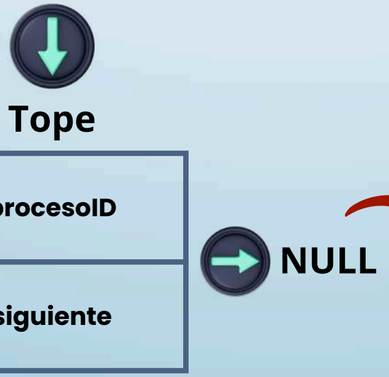




PILA

10. Asignacion de memoria (push)

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




```
NodoMemoria* pilaMemoria = NULL;
```

Creación del NodoMemoria con un dato “procesoID” y su puntero “siguiente”

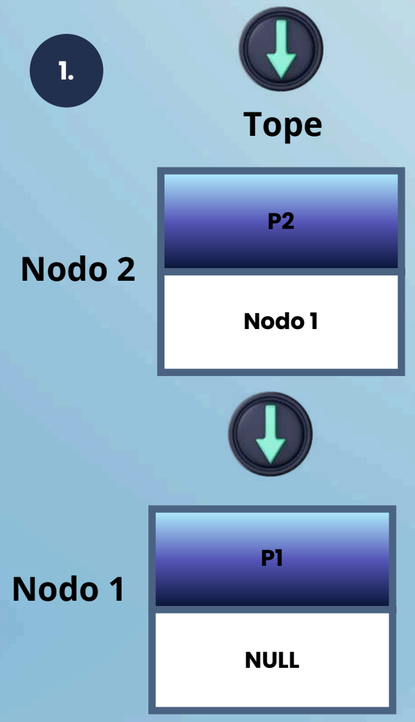
```
void pushMemoria() {
    string id;
    cout << "Ingrese el ID del proceso para asignar memoria: ";
    cin >> id;

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

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

Solicita el ID para asignar una parte de la memoria y lo guarda en la variable “id”

Este bloque de código crea un nuevo nodo con el ID del proceso y lo coloca en la cima de la pila, actualizando el puntero pilaMemoria. Así, el nodo más reciente siempre queda en el tope, cumpliendo el comportamiento de una pila (LIFO).

10. Asignacion de memoria (push)

1.

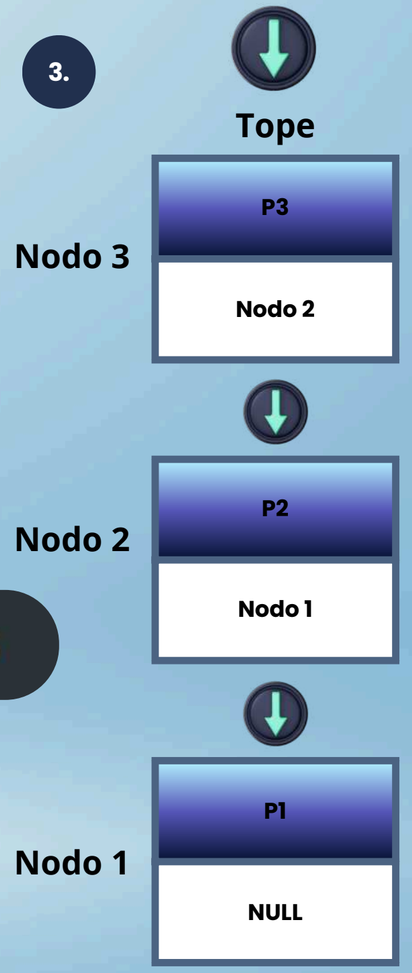


2.


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

```
pilaMemoria = nuevo;
```

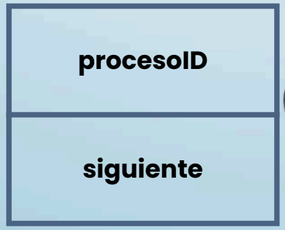
3.



11. Liberación de memoria (pop)

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

Tope



NULL

```
NodoMemoria* pilaMemoria = NULL;
```

Creación del NodoMemoria con un dato “procesoID” y su puntero “siguiente”

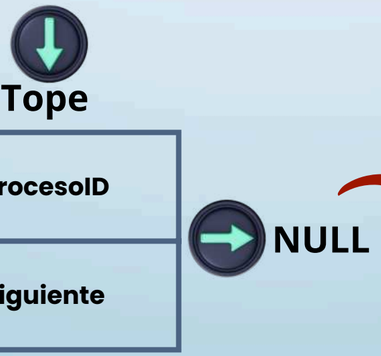

```
void popMemoria() {
    if (!pilaMemoria) {
        cout << "La memoria ya esta vacia.\n";
    } else {
        cout << "Liberando memoria del proceso ID: " << pilaMemoria->procesoID << "\n";
        NodoMemoria* temp = pilaMemoria;
        pilaMemoria = pilaMemoria->siguiente;
        delete temp;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

Verifica si la pila de memoria está vacía y muestra un mensaje si es así.

Este bloque libera la memoria del nodo que está en la cima de la pila, mostrando el ID del proceso que se está eliminando. Luego actualiza el puntero pilaMemoria para que apunte al siguiente nodo, manteniendo el funcionamiento tipo pila (LIFO).

12. Estado actual de la memoria

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




```
NodoMemoria* pilaMemoria = NULL;
```

Creación del NodoMemoria con un dato “procesoID” y su puntero “siguiente”

```
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");
}
```

Verifica si la pila de memoria está vacía y muestra un mensaje si es así.

Este bloque de código recorre la pila desde la cima (pilaMemoria) y muestra en pantalla el ID de cada proceso junto con su posición, permitiendo visualizar el contenido de la pila de forma ordenada.

12. Estado actual de la memoria

Avanza nodo por nodo desde la cima hasta el final, permitiendo acceder a cada elemento para mostrar su posición y su ID.

```
NodoMemoria* temp = pilaMemoria;
```

GRACIAS!!!

Stay curious and explore!