Memoria Estructura de Datos PL1

Miguel de Sande Moreno

Marcos Bonilla de las Morenas

0.Presentando el planteamiento

Este ejercicio trataba de calcular el tiempo medio de estancia de unos procesos en un sistema operativo. Para abordarlo, además, debimos tener en cuenta que cada uno de estos procesos cuenta con una prioridad propia, y que estos deben ser insertados al principio del programa desde un menú. El funcionamiento es el siguiente: unos procesos salen de una pila acorde a un tiempo de salida y son ejecutados en un núcleo, donde también tienen un periodo de estancia. Mientras los núcleos estén ocupados, si uno de los procesos fuese a salir de la pila, este debe ser almacenado en una cola, la cual debe estar ordenada por su prioridad, y cuando los núcleos se desalojen se colocará el primero de la cola, que será el que más prioridad tenga.

1. ¿Cómo estructuramos la prioridad?

Para empezar, declaramos las especificaciones de la **Pila** y la **Cola**, haciendo también una clase **NodoPila** y **NodoCola** para cada una, ya que de esta manera nos será más sencillo controlar los punteros.

-Implementamos para **Pila** los siguientes atributos y métodos:

Atributos **privados**:

pnodo cima*; //Un puntero que señala a la cima de la pila.*

Métodos **públicos**:

Pila(); *//Siendo este el constructor.*

~Pila(); *//Siendo este el destructor.*

bool esVacia(); *//Que comprueba y devuelve si la pila es vacía o no.*

void apilar(Proceso proceso); *//Siendo esta una función que, pasado un proceso p como parámetro, lo coloca como último elemento de la pila.*

void desapilar(); *//Que elimina el último de la pila.*

void mostrar(); *//Que imprime la pila por la pantalla.*

Proceso cimaPila(); *//Que devuelve el proceso en la cima de la pila.*

void insertarTiempo(Proceso proceso); *//La función clave de la pila. Cuando se le pasa un proceso como parámetro, esta función lo inserta en la pila en función de su orden de salida (dejará por delante los procesos que tengan un menor tiempo de salida y por detrás los que tengan uno mayor.)*

-Implementamos para **Cola** los siguientes atributos y métodos:

Atributos **privados**:

NodoCola *primero; *//Apunta al primer elemento de la cola.*

NodoCola *ultimo; *//Apunta al último elemento de la cola.*

int longitud; *//Un integer que representa la longitud de la cola.*

Métodos **públicos**:

Cola(); *//Siendo este el constructor.*

~Cola(); *//Siendo este el destructor.*

void encolar(Proceso p); *//Siendo esta una función que, pasado un proceso p como parámetro, lo coloca como último elemento de la cola.*

Proceso inicio(); *//Que devuelve el primer elemento de la cola.*

Proceso fin(); *//Que devuelve el último elemento de la cola.*

int colaLongitud(); *//Que devuelve la longitud de la cola.*

Proceso desencolar(); *//Que elimina el primer elemento de la cola y devuelve el proceso desencolado.*

bool esVacia(); *//Que comprueba si la cola es vacía o no y devuelve un booleano en función del resultado.*

void mostrarCola(); *//Que imprime por pantalla la cola.*

void encolarPrioridad(Proceso p); *//La función clave de la cola. Cuando se le pasa un proceso como parámetro, esta función lo inserta en la cola en función de su prioridad (dejará por delante los procesos con más prioridad y por detrás los que menos prioridad tengan.)*

void actualizar() *//Suma 1 al tiempo que los elementos en la cola llevan en el sistema operativo.*

-Implementamos para **NodoPila** los siguientes atributos y métodos (cabe destacar que, como tiene *friend class Pila*, la pila puede acceder a las funciones y atributos privados de esta clase):

Atributos **privados**:

Proceso proceso; *//Los nodos deben contener procesos.*

NodoPila *siguiente; *//Apunta al siguiente elemento de la pila.*

Métodos **públicos**:

NodoPila(); *//Siendo este el constructor vacío.*

NodoPila(Proceso proceso, NodoPila *sig = NULL); *//Siendo este otro constructor pero al que pasas un parámetro de proceso y otro de NodoPila.*

~NodoPila(); *//Siendo este el destructor.*

-Implementamos para **NodoCola** los siguientes métodos y atributos (cabe destacar que, como tiene *friend class Cola*, la cola puede acceder a las funciones y atributos privados de esta clase):

Atributos **privados**:

Proceso proceso; *//Los nodos deben contener procesos.*

```
NodoCola *siguiente;      //Apunta al siguiente elemento de la
                          cola.
```

Métodos **públicos**:

```
NodoCola();          //Siendo este el constructor vacío.
NodoCola(Proceso p, NodoCola *sig = NULL);      //Siendo este otro
                                                constructor pero al que
                                                pasas un parámetro de
                                                proceso y otro de NodoCola.
~NodoCola();         //Siendo este el destructor.
```

Hasta ahora, estas implementaciones son muy similares a las vistas en clase, pero cabría profundizar en las funciones encolarPrioridad, insertarTiempo y actualizar.

1.1 Nuevos métodos

void Pila::insertarTiempo(Proceso proceso)

```
{
   if (esVacia())
   {
      apilar(proceso);
   }
   else
   {
      Pila *aux = new Pila();
      while (!esVacia() && cimaPila().tiempoInicio <= proceso.tiempoInicio)
      {
         aux->apilar(cimaPila());
         desapilar();
      }
```

```
        apilar(proceso);
        while (!aux->esVacia())
        {
            apilar(aux->cimaPila());
            aux->desapilar();
        }
        delete aux;
    }
}
```

Esta función, como su nombre indica, se encarga de insertar en la pila un proceso en función de su **tiempo de inicio.** Profundizaremos en este atributo más adelante, cuando analicemos los procesos.

La función comprueba en primer lugar si la pila es vacía, y, si es así, simplemente apila el proceso. En caso contrario, se crea **una nueva pila auxiliar**, y luego se va iterando la pila comprobando si el primer elemento tiene un tiempo de inicio menor o igual al del proceso que queremos introducir. Si este es el caso, se deposita el elemento de forma provisional en la pila auxiliar. Una vez se han colocado todos los elementos con un tiempo de inicio menor o igual de esta manera, **introducimos** nuestro elemento. Es importante remarcar el **igual** en esta operación, ya que si el tiempo que queremos insertar coincide con algún tiempo de los elementos introducidos en la pila, el que insertemos será el último de los elementos con ese tiempo que introduzcamos. Cuando se haya introducido nuestro elemento, iteraremos la pila auxiliar, **devolviendo** así los elementos a la **pila original**, y concluyendo la función **eliminando la pila auxiliar**.

De esta manera nuestro elemento habrá quedado insertado en la pila acorde a su tiempo de inicio respecto al resto de elementos.

---

void Cola::encolarPrioridad(Proceso p)

```
{
    NodoCola *nuevo = new NodoCola(p);
    if (esVacia())
```

```
    {
        primero = nuevo;
        ultimo = nuevo;
    }
    else
    {
        Cola *Cola_aux = new Cola();
        while (primero != nullptr && primero->proceso.prioridad >= p.prioridad)
        {
            Cola_aux->encolar(desencolar());
        }
        Cola_aux->encolar(p);
        while (!esVacia())
        {
            Cola_aux->encolar(desencolar());
        }
        while (!Cola_aux->esVacia())
        {
            encolar(Cola_aux->desencolar());
        }
        delete Cola_aux;
    }
    longitud++;
}
```

En este caso, para la función se trabaja con **punteros**. Al inicio de la función creamos un nuevo NodoCola. Si la cola es vacía, insertamos el elemento, ya que tanto primero como último = nuevo (siendo “nuevo” el NodoCola que creamos). Si no, al igual que en la función anterior, trabajaremos con una cola auxiliar. Iremos iterando esta cola

mientras el puntero del primer elemento no apunte a null (es decir, que esté vacía), y mientras que el primer elemento tenga una **prioridad** mayor o igual que la del proceso que vamos a insertar.

Esta función se puede considerar una modificación de la anterior ya que el funcionamiento es el mismo. Ambas se encargan de revisar si los elemento de las estructuras tienen uno de los atributos de los procesos en mayor o menor cantidad e insertarlo donde corresponda. Al igual que pasaba con insertarTiempo, los elementos que se vayan desencolando se irán almacenando en la cola auxiliar, con la diferencia de que una vez encolado nuestro elemento, **será necesario encolar el resto de la cola** en la cola auxiliar para que al restaurarla esta resulte en el orden correcto. Una vez hemos vaciado la cola en la cola auxiliar (con el proceso insertado), restauramos la cola y eliminamos la cola auxiliar

---

void Cola::actualizar()

```
{
   if (!esVacia())
   {
      NodoCola *aux = primero;
      while (aux != NULL)
      {
         aux->proceso.tiempoEnSO++;
         aux = aux->siguiente;
      }
   }
}
```

Es importante remarcar que la aplicación solo entrará a esta función si la cola no es vacía. La función crea un nodo auxiliar y recorre la cola sumando 1 a todos los **tiempos en el Sistema Operativo** de la cola.

2.Nuevas clases

El enunciado pedía el tiempo medio en el que unos procesos son ejecutados en tres núcleos. Por lo tanto, hemos desarrollado dos clases más: una para los **procesos**, y otra para los **núcleos**.

-Implementamos para **Proceso** los siguientes métodos y atributos.

Atributos **públicos**:

int prioridad; *//Siendo esta la prioridad del proceso para ser ejecutado.*

int PID; *//Siendo este el identificador del proceso.*

int PPID; *//Siendo este el identificador del proceso padre.*

int tiempoInicio; *//Siendo este el tiempo en el que el proceso sale de la pila.*

int tiempoEjecucion; *//Siendo este el tiempo que el proceso pasa dentro del núcleo (el tiempo que se está ejecutando).*

int tiempoEnSO; //Siendo este el tiempo que el proceso ha pasado en el sistema operativo.

Métodos **públicos**:

Proceso(); *//Siendo este el constructor vacío.*

Proceso(int PID, int prioridad, int tiempoInicio, int tiempoEjecucion);

//Siendo este el constructor parametrizado.

~Proceso(); *//Siendo este el destructor.*

Hay que remarcar que el tiempo en el **sistema operativo se inicializa en 0**, y, **el PPID**, al ser irrelevante para la práctica, **será inicializado en 1**.

-Implementamos para **Núcleo** los siguientes métodos y atributos:

Atributos **privados**:

Proceso proceso; *//Siendo este el proceso que será ejecutado en el núcleo.*

bool libre; *//Siendo este un parámetro que indica si el núcleo está libre o no (dicho de otra manera, si está ejecutando algún proceso).*

int tiempoProcesado; *//Siendo este el tiempo que lleva siendo ejecutado un proceso en el núcleo.*

Métodos **públicos**:

Nucleo(); *//Siendo este el constructor.*

~Nucleo(); *//Siendo este el destructor.*

bool esVacio(); *//Siendo esto un bool que comprueba si el núcleo está vacío o no y devuelve un booleano en función de ello.*

void procesar(Proceso proceso); *//Siendo esta la función en la que un proceso se introduce en el núcleo para ser ejecutado.*

void quitarProceso(); *//Siendo esta la función en la que un proceso sale del núcleo.*

void mostrarNucleo(); *//Siendo esta la función que muestra el proceso dentro de un núcleo.*

int actualizar(); *//Siendo esta la función que suma 1 al tiempo en el sistema operativo y al tiempo procesado a los procesos dentro del núcleo, y luego comprueba si el proceso ha terminado de ser ejecutado. Si es así, llama a la función "quitarProceso()".*

Es importante recalcar que la función "actualizar()" ha sido colocada tanto en la cola como en los núcleos, ya que de esta manera se irá sumando 1 a los tiempos en el sistema operativo de los procesos que estén tanto en la cola como en los núcleos, que componen el sistema operativo.

Anteriormente se realizó otra interpretación en la que se trataba el núcleo como una cola de un solo elemento, pero al no ser esta la función específica para la que se diseñaron las colas, se decidió implementar de esta manera que es igualmente funciona.

3. Métodos del núcleo

En este apartado profundizaremos en las funciones más importantes del núcleo, remarcando cómo se inicializan y señalando detalles importantes.

Nucleo::Nucleo()

```
{
   proceso = Proceso();
   libre = true;
   tiempoProcesado = 0;
}
```

Esta es la forma en la que se instancia un núcleo, con libre = True y el tiempoProcesado = 0.

---

bool Nucleo::esVacio()

```
{
   return libre;
}
```

Para esVacio() es un booleano que devuelve si el núcleo es libre o no (no olvidemos que "libre" se inicializa a True).

---

void Nucleo::quitarProceso()

```
{
```

```
    if(!esVacio())
    {
        proceso = Proceso();
        libre = true;
    }
}
```

Esta función se ejecuta si el núcleo no es vacío, ya que en caso de estar libre no se podría quitar un proceso del núcleo. Reemplaza el proceso por un “proceso vacío”, y declara el booleano libre a True, por lo que nuevos procesos pueden entrar en el núcleo.

---

void Nucleo::mostrarNucleo()

```
{
    if(!esVacio())
    {
        cout << "      PID: " << proceso.PID << endl;
        cout << "      PPID: " << proceso.PPID << endl;
        cout << "      Prioridad: " << proceso.prioridad << endl;
        cout << "      Minuto de entrada en el sistema: " << proceso.tiempoInicio << endl;
        cout << "      Tiempo que tarda en ejecutarse: " << proceso.tiempoEjecucion <<
endl;
        cout << "      Tiempo que lleva en el sistema operativo: " << proceso.tiempoEnSO
+1 << endl;
        cout << endl;
    }
    else
    {
        cout << "      Núcleo vacío" << endl;
```

```
        cout << endl;
    }
}
```

Esta función imprime por la pantalla las características del proceso que se está ejecutando en el núcleo en caso de haberlo. Si el núcleo está vacío, imprime “Núcleo Vacío” en su lugar.

---

int Nucleo::actualizar()

```
{
   if(!esVacio())
   {
      proceso.tiempoEnSO += 1;
      tiempoProcesado += 1;

      if (proceso.tiempoEjecucion == tiempoProcesado)
      {
         int tiempoTotalProceso = proceso.tiempoEnSO;
         cout << "El proceso con PID " << proceso.PID << " ha terminado! Ha estado " << tiempoTotalProceso << " minutos en el SO."<< endl;
         cout << endl;
         quitarProceso();
         tiempoProcesado = 0;
         return tiempoTotalProceso;
      }else
      {
```

```
            return 0;
        }
    }else{
        return 0;
    }
}
```

Esta función comprueba si el núcleo está ejecutando algún proceso, y, en caso de ser así, suma 1 a su tiempo procesado y al tiempo que lleva ese proceso en el sistema operativo. Después, si el tiempo de ejecución del proceso es el mismo tiempo que lleva procesado, significa que ha terminado de ejecutarse y por lo tanto el núcleo puede desalojarlo para dar paso a nuevos procesos. Al final de la función, el tiempo procesado se vuelve a inicializar a 0 y la función devuelve el tiempo total que ese proceso ha estado en el sistema operativo para hacer la media una vez todos los procesos hayan sido ejecutados. Hay varios “else” con return 0 para evitar warnings.

---

4. El main, esqueleto del programa

La interpretación que le dimos al programa y que se ve reflejada en el main, es la siguiente:

Lo primero que se muestra en la pantalla, es un menú que tiene las siguientes opciones:

```
void menuInicio()
{
    cout << endl;
    cout << "#-#-#-# Menú de opciones #-#-#-#" << endl;
    cout << "    1. Ingresar proceso" << endl;
    cout << "    2. Mostrar procesos" << endl;
```

```
    cout << "   3. Borrar pila de procesos" << endl;
    cout << "   4. Ejecutar manualmente" << endl;
    cout << "   5. Ejecutar automáticamente" << endl;
    cout << "   6. Salir" << endl;
    cout << "#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#" << endl;
    cout << endl;
    cout << "Ingrese una opción: ";
}
```

La opción 1 te permite añadir un proceso (o varios) de la pila de procesos disponibles, una estructura llena de procesos predefinidos en el código. Es importante remarcar que todos los procesos que se añadan lo harán con la función insertarTiempo, por lo que se ingresarán en la pila en función de su tiempo de inicio.

La opción 2 te permite mostrar los procesos de la pila que se vayan a ejecutar en el sistema operativo.

La opción 3 te permite borrar los procesos de la pila.

La opción 4 te permite ejecutar la aplicación dejando al usuario la libertad de añadir el tiempo que quiera que pase en el sistema operativo y, conforme avanza la ejecución de los procesos, también te permite mostrar la información de los procesos que están siendo ejecutados en los núcleos.

La opción 5 te permite ejecutar la aplicación de modo automático, es decir, acaba el proceso y te da la media del tiempo que los procesos han estado en el sistema operativo, proporcionando además toda la información, minuto a minuto.

La opción 6 te permite salir del programa.

---

void crearProcesos(int *n*)

```
{
    for (int i = 0; i < n; i++)
    {
        if (!procesosDisponibles.esVacia())
```

```
        {
            pila.insertarTiempo(procesosDisponibles.cimaPila());
            procesosDisponibles.desapilar();
            if(procesosDisponibles.esVacia())
            {
                cout << "Ya se ha añadido el máximo número de porcesos." << endl;
            }
        }else{
            cout << "No hay más procesos disponibles :(" << endl;
            break;
        }
    }

}
```

Esta función se encarga de añadir procesos de la pila de procesos disponibles a la pila de procesos que se quieran ejecutar.

---

void detallesProcesosEjecucion()

```
{
    cout << "- Proceso en ejecución en núcleo 1: " << endl;
    n1.mostrarNucleo();
    cout << "- Proceso en ejecución en núcleo 2: " << endl;
    n2.mostrarNucleo();
    cout << "- Proceso en ejecución en núcleo 3: " << endl;
    n3.mostrarNucleo();
```

```
    cout << endl;
}
```

Muestra información de los núcleos y los procesos que contienen.

---

void mostrarPilayCola()

```
{
    cout << "La pila ahora es así: ", pila.mostrar();
    cout << "La cola ahora es así: ", cola.mostrarCola();
    cout << endl;
}
```

Muestra el estado actual de la pila de procesos y la cola de espera.

---

void menuEjecucion()

```
{
    cout << endl;
    cout << "---------- Menú de ejecución ----------" << endl;
    cout << "   1. Mostrar detalles de procesos en ejecución" << endl;
    cout << "   2. Continuar" << endl;
    cout << "   3. Continuar N minutos" << endl;
    cout << "   4. Terminar Automáticamente" << endl;
    cout << "-------------------------------------------------" << endl;
    cout << endl;
```

```
    cout << "Ingrese una opción: ";
}
```

Muestra un menú con opciones una vez el sistema operativo ha terminado un ciclo de tiempo. Esta opción solo se puede ver si el usuario ha seleccionado “modo manual”.

La opción 1 muestra los detalles de los núcleos y los procesos que en ellos se están ejecutando.

La opción 2 recorre otro ciclo de tiempo de un minuto.

La opción 3 le permite al usuario seleccionar el número de minutos que desee que transcurran en la aplicación.

La opción 4 cambia el modo manual a modo automático y hace que la aplicación termine, mostrando la información minuto a minuto.

---

void ejecutar(bool *manual*)

```
{
    int contador = 0;
    int sumar = 0;

    while (!cola.esVacia() || !pila.esVacia() || !n1.esVacio() || !n2.esVacio()
|| !n3.esVacio())
    {
        if(manual)
        {
            menuEjecucion();

            int opcion = 0;
            cin >> opcion;

            if (cin.fail()) {
```

```
    cin.clear();
    cin.ignore();
    opcion = 0;
}

switch (opcion)
{
case 1:
    detallesProcesosEjecucion();
    sumar = 0;
    break;
case 2:
    sumar = 1;
    break;
case 3:
    cout << "Ingrese el tiempo a sumar: ";
    cin >> sumar;
    cout << endl;
    break;
case 4:
    manual = false;
    sumar = 0;
    break;

default:
    cout << "Opción no válida" << endl;
    cout << endl;
    break;
}
```

```
}else{
   sumar = 1;
}

for (int i = 0; i < sumar; i++)
{
   contador += 1;

   cout << "_____" << endl;
   cout << " * Minuto actual: " << contador << endl;
   cout << endl;

   //Saco a la cola los procesos que arrancan en este minuto
   while (pila.cimaPila().tiempoInicio == contador && !pila.esVacia())
   {
      cola.encolarPrioridad(pila.cimaPila());
      pila.desapilar();
   }

   // Asigno procesos a los núcleos si están libres
   if (n1.esVacio() && !cola.esVacia()){
      n1.procesar(cola.desencolar());
   }
   if (n2.esVacio() && !cola.esVacia()){
      n2.procesar(cola.desencolar());
   }
   if (n3.esVacio() && !cola.esVacia()){
      n3.procesar(cola.desencolar());
   }
```

```
if (!manual)
{
   mostrarPilayCola();
   detallesProcesosEjecucion();
   cout << endl;
}

int t1 = n1.actualizar();
int t2 = n2.actualizar();
int t3 = n3.actualizar();



sumaTiempos += t1 + t2 + t3;

if (t1 != 0)
{
   procesosEjecutados += 1;
}
if (t2 != 0)
{
   procesosEjecutados += 1;
}
if (t3 != 0)
{
   procesosEjecutados += 1;
}
```

```
            cola.actualizar();


            if (cola.esVacia() && pila.esVacia() && n1.esVacio() && n2.esVacio() &&
n3.esVacio())
            {
                break;
            }
        }
    }


    cout << "Se han necesitado " << contador << " minutos para procesar todos los
procesos." << endl;
    cout << "Se han ejecutado " << procesosEjecutados << " procesos." << endl;


    if (procesosEjecutados != 0)
    {
        cout << "El tiempo medio de estancia de los procesos en el sistema operativo ha
sido de " << sumaTiempos/procesosEjecutados << " minutos." << endl;
    }else{
        cout << "No se ha ejecutado ningún proceso." << endl;
    }


    // Reinicio de variables para una posible nueva iteración
    contador = 0;
    sumaTiempos = 0;
    procesosEjecutados = 0;
    definirProcesos();
}
```

La siguiente función es en la que reside la mayor parte del peso de la aplicación. El usuario entra a ella una vez selecciona “modo manual” o “modo automático” y solo sale de ella cuando ya se han ejecutado todos los procesos. Controla el paso del tiempo y la correcta suma de los tiempos que los procesos tardan en ejecutarse.

Lo primero que hace es inicializar las variables contador y suma a 0. Contador es una variable que guarda el tiempo que transcurre durante el uso de la aplicación. Lleva el “tiempo actual”. Sumar, por otra parte, es el número de minutos que introduce el usuario cuando introduce en un menú la cantidad de minutos que quiere que transcurran en el programa.

Tras esto, entra en el bucle while que comprueba que:

-La pila no está vacía

-La cola no está vacía

-Los núcleos no están vacíos

Es decir, el bucle finaliza cuando ya no queden procesos que ejecutar.

Una vez se ha entrado en el bucle, se pasa por un if. Si se ha accedido de manera manual, se muestra el menú de ejecución que vimos anteriormente (si selecciona la primera opción, permite ver los detalles de los núcleos, si selecciona la segunda, establece el valor de “sumar” a 1, si selecciona la tercera, solicita al usuario una cantidad de minutos y establece el valor de “sumar” a esa cantidad de minutos), y si ha entrado en modo automático, establece el valor de sumar a 1.

Tras esto, se entra en un bucle for que se ejecuta tantas veces como el valor de “sumar”. Incrementa en 1 el valor del contador, todos los procesos que salgan de la pila en ese minuto, se trasladan a la cola en orden de prioridad, introduce los procesos de la cola en los núcleos (siempre que se pueda, ya que primero tiene que comprobar que están vacíos), ejecuta la función actualizar() de los núcleos (explicada anteriormente) que devuelve 0 si el proceso no ha terminado de ser ejecutado o devuelve el tiempo que ha estado el proceso en el sistema operativo en caso contrario. Posteriormente, suma esos tiempos e incrementa la variable sumaTiempos, una variable en la que se recogerá el total de tiempo que han estado todos los procesos en el sistema operativo (y con la que ser hará la media una vez haya terminado la aplicación). En caso de haber devuelto la función actualizar algo distinto de 0 (lo que significa el fin de la ejecución de un proceso), suma 1 a los procesosEjecutados. Por último, ejecuta la función actualizar() de las colas (sumando así 1 a los minutos que llevan los procesos en la cola).

De esta manera habría salido del bucle for y del bucle while, y solo quedaría hacer la división de todo el tiempo conjunto que han estado los procesos en el SO (sumaTiempos) y el número de procesos que se han ejecutado (procesosEjecutados).

```
int main(){

    bool salir = false;
    definirProcesos();

    while(salir == false)
    {
        menuInicio();

        int opcion = 0;
        cin >> opcion;

        if (cin.fail()) {
            cin.clear();
            cin.ignore();
            opcion = 0;
        }

        int cuantos = 0;
        switch (opcion)
        {
        case 1:

            cout << "Ingrese la cantidad de procesos a crear: ";
            cin >> cuantos;

            if (cin.fail()) {
                cin.clear();
                cin.ignore();
```

```
        cuantos = 0;
    }else{
        crearProcesos(cuantos);
    }
    break;
case 2:
    cout << endl;
    cout << "La pila actual es esta: " << endl ,
    pila.mostrar();
    break;
case 3:
    while (!pila.esVacia())
    {
        pila.desapilar();;
    }
    cout << "La pila ha sido borrada." << endl;
    if (procesosDisponibles.esVacia())
    {
        definirProcesos();
        cout << "Se han redefinido los procesos" << endl;
    }

    break;
case 4:
    ejecutar(true);
    break;
case 5:
    ejecutar(false);
    break;
```

```
        case 6:
            salir = true;
            break;

        default:
            cout << "Opción no válida" << endl;
            break;
        }
    }
    return 0;
}
```

Por último, hay que abordar la función main() de la aplicación.

Toda esta función es un bucle While en la que cuando salir = False se vuelve a ejecutar. Al seleccionar la opción 6, salir = True.

Lo primero que hace es mostrar el menú de inicio, comentado anteriormente. Lo único remarcable sobre esto es que cuando se selecciona la opción 4, llama a ejecutar con manual = True y cuando se selecciona la opción 5, llama a ejecutar con manual = False.

A partir de ahí, es la función ejecutar la que se encarga de la ejecución de los procesos y del correcto funcionamiento del programa.

SEGUNDA PARTE: IMPLEMENTACIÓN DE LISTAS

0’ Presentando el planteamiento

Para esta segunda parte, había que llevar a cabo una serie de modificaciones. En primer lugar, ahora ya no hay tres núcleos fijos sino que hay que hacer una implementación de una lista de núcleos en los que se irán alojando los procesos. Cada uno de esos núcleos tendrá una cola de prioridades en las que se irán almacenando los procesos, y, cuando la cola de un núcleo tenga más de dos procesos, se creará un nuevo núcleo en la lista con su correspondiente cola. El objetivo de la aplicación sigue siendo el mismo: calcular el tiempo medio que pasan una serie de procesos dentro del sistema operativo.

1’ ¿Qué hemos modificado?

Respecto a la primera parte, se han modificado varias cosas. En primer lugar, como se estableció en el planteamiento, se ha añadido un atributo **Cola colaProcesos;** a los núcleos, que será la cola de prioridades en la que se vayan colocando los procesos.

También se ha llevado a cabo la implementación de la clase **Lista** con los siguientes atributos y métodos:

Atributos **privados:**

Nucleo primeroLista; #Guarda el primer elemento de la lista. (Una lista está por definición compuesta por un “primero” y un “resto” que a su vez es otra lista).

int longitud; #Un entero con la longitud de la lista.

bool vacia; #Guarda si la lista está vacía.

Lista *restoListaPtr; #Un puntero que apunta al resto de la lista.

Métodos **públicos**:

Lista(); #Un constructor sin parámetros.

Lista(Nucleo *nucleo*, Lista **resto* = new Lista()); #Un constructor parametrizado.

~Lista(); #El destructor.

bool esVacia(); #Devuelve si la lista es vacía.

Lista resto(Lista *lista*); #Que devuelve la lista desencolando el primer elemento.

Lista* restoPtr(Lista *lista*); #Devuelve un puntero que apunta al resto de la lista.

Nucleo primero(Lista *lista*); #Te devuelve el núcleo en primera posición de la lista.

Nucleo* obtenerNodo(int *posicion*); #Devuelve un puntero al núcleo en una determinada posición.

Lista* eliminarNodo(int *posicion*); #Devuelve un puntero a la lista sin el elemento de una determinada posición.

Lista* añadirIzquierda(Nucleo *nucleo*); #Devuelve un puntero a una lista a la que se ha añadido un elemento por la izquierda.

Lista* añadirDerecha(Nucleo *nucleo*); #Devuelve un puntero a una lista a la que se ha añadido un elemento por la derecha.

int longitudLista(); #Devuelve el valor de la longitud de la lista.

Nucleo* primeroPtr(); #Un puntero que apunta hacia el primer elemento de la lista.

void mostrarLista(Lista *lista*); #Que muestra la lista por la pantalla.

Como se puede observar, hay varios métodos que se repiten, ya que uno devuelve el objeto y otro devuelve un puntero hacia el objeto. Es importante que esté implementado de esta manera ya que se precisa trabajar con punteros para realizar una correcta implementación de las listas.

2’ Funciones importantes

Entre los cambios que se han implementado, hay algunos menos remarcables como una pequeña actualización en el apartado de “cola” para revisar la manera en la que se mide la longitud. Sin embargo, dedicaremos este apartado a abordar las nuevas funciones que sean especialmente relevantes para el funcionamiento del código.

```
Lista::Lista()
{
   vacia = true;
   restoListaPtr = NULL;
   longitud = 0;
}

Lista::Lista(Nucleo nucleo, Lista *resto)
{
   vacia = false;
   primeroLista = nucleo;
   this->restoListaPtr = resto;
   longitud = resto->longitud + 1;
}

bool Lista::esVacia()
{
   return vacia;
}

Lista* Lista::añadirIzquierda(Nucleo nucleo)
{
   return new Lista(nucleo, this);
}

Lista* Lista::añadirDerecha(Nucleo nucleo)
{
   if (esVacia())
```

```
    {
        primeroLista = nucleo;
        restoListaPtr = new Lista();
        vacia = false;
        longitud = 1;
    }
    else
    {
        Lista* p = this;
        while (p->restoListaPtr != NULL && !p->restoListaPtr->esVacia())
        {
            p = p->restoListaPtr;
        }
        p->restoListaPtr = new Lista(nucleo, new Lista());
        longitud++;
    }
    return this;
}
```

Nucleo Lista::primero(Lista lista)

```
{
    if (!esVacia())
    {
        return lista.primeroLista;
    }
    else
    {
        return Nucleo();
    }
```

```
}
```

Lista Lista::resto(Lista lista)

```
{
   if (!esVacia())
   {
      return *lista.restoListaPtr;
   }
   else
   {
      return Lista();
   }
}
```

Lista Lista::eliminarNodo(int posicion)

```
{
   if (!esVacia() && posicion <= longitud)
   {
      if (posicion == 1)
      {
         Lista* temp = this->restoListaPtr;
         *this = *temp;
         delete temp;
      }
      else if (posicion == 2)
      {
         Lista* temp = this->restoListaPtr->restoListaPtr;
         delete this->restoListaPtr;
         this->restoListaPtr = temp;
```

```
    this->longitud--;
}
else
{
    Lista* p = this;
    Lista* ultimo;
    for (int i = 2; i < posicion; i++)
    {
        p->longitud--;
        p = p->restoListaPtr;
        if (p->restoListaPtr->restoListaPtr == NULL)
        {
            ultimo = p;
        }
    }

    if (p->restoListaPtr == NULL)
    {
        delete p;
        ultimo->restoListaPtr = NULL;
    }
    else
    {
        Lista* q = p->restoListaPtr->restoListaPtr;
        delete p->restoListaPtr;
        p->restoListaPtr = q;
    }
}
```

```
    }
    if (longitud == 0)
    {
        vacia = true;


    }
    return *this;
}
```

Lista* Lista::restoPtr(Lista lista)

```
{
    if (!esVacia())
    {
        return lista.restoListaPtr;
    }
    else
    {
        return NULL;
    }
}
```

void Lista::mostrarLista(Lista lista)

```
{
    while (!lista.esVacia())
    {
        cout << lista.primero(lista).getID() << ", ";
        lista = lista.resto(lista);
    }
    cout << endl;
```

```
}
```

Nucleo* Lista::obtenerNodo(int posicion)

```
{
   if (!esVacia() && posicion <= longitud)
   {
      Lista* p = this;
      for (int i = 1; i < posicion; i++)
      {
         p = p->restoListaPtr;
      }
      return p->primeroPtr();
   }
   else
   {
      cout << "La posición no existe" << endl;
      return NULL;
   }
}
```

int Lista::longitudLista()

```
{
   return longitud;
}
```

Lista::~Lista()

```
{
```

```
}

Nucleo* Lista::primeroPtr()
{
   if (!esVacia())
   {
      return &this->primeroLista;
   }
   else
   {
      return NULL;
   }
}
```

Para empezar, cabe resaltar que el constructor vacío de lista se inicializa de esta manera:

Vacía = True

restoLista = null

longitud = 0

Esto se debe a que la lista es vacía, tiene longitud 0 porque no tiene ningún núcleo al empezar y el resto de la lista es nulo.

Cuando la lista se inicia con argumentos (con un Núcleo y un puntero al resto de la lista), la inicialización es la siguiente:

Vacía = False

primeroLista = núcleo

restoListaPtr = resto

longitud = longitud del resto + 1

No será vacía porque ya tiene un núcleo, el primer elemento de la lista es el propio núcleo, el puntero de la lista apunta al resto y la longitud será la que tenga el resto de la lista + 1.

Otra función remarcable es añadirIzquierda, que devuelve un puntero a la lista con el elemento añadido por la izquierda.

También tenemos la función añadirDerecha, que itera la lista hasta llegar al último elemento y hace que el puntero restoLista del último elemento, en lugar de apuntar a Null, apunte al nuevo elemento de la lista. El restoLista de este último sí apuntará a Null. Después, devuelve el puntero al primero de la lista.

Hay que hablar también de eliminarNodo, que, al pasar el usuario una posición (llamémosla p), itera la lista hasta la posición (p-1). El atributo restoLista de ese Nodo, hace que en lugar de apuntar a p, apunte a (p+1). Después, libera (p).

Asimismo, tenemos la función obtenerNodo, la cual sigue un procedimiento similar al de la función anteriormente explicada. El usuario pasa una posición (de nuevo, llamémosla p), y, esta vez, la función itera la lista hasta la posición (p) y devuelve el puntero que apunta a esa posición.

El resto de funciones han sido abordadas también, pero debido a su brevedad (y a que su funcionamiento se puede resumir observando el punto **1’ ¿Qué hemos modificado?**), no han sido incluidas en este apartado.

---

3. El nuevo main()

Para que la aplicación funcione de manera fructífera, se ha tenido que modificar también el main, pero tampoco de manera desmesurada. Lo que antes se hacía en tres núcleos distintos, ahora lo lleva un bucle for que se encarga del correcto funcionamiento de nuestras nuevas implementaciones.

Abordaremos brevemente los nuevos cambios realizados.

```
void ejecutar(bool manual)
{
   int contador = 0;
   int sumar = 0;


   while (!pila.esVacia() || (listaNucleos->longitudLista() != 1 && !listaNucleos->primeroPtr()->esVacio()))
   {
      if(manual)
      {
         menuEjecucion();


         int opcion = 0;
         cin >> opcion;


         if (cin.fail()) {
            cin.clear();
            cin.ignore();
            opcion = 0;
         }


         switch (opcion)
         {
         case 1:
            detallesProcesosEjecucion();
            sumar = 0;
            break;
         case 2:
            sumar = 1;
```

```
            break;
        case 3:
            cout << "Ingrese el tiempo a sumar: ";
            cin >> sumar;
            cout << endl;
            break;
        case 4:
            manual = false;
            sumar = 0;
            break;

        default:
            cout << "Opción no válida" << endl;
            cout << endl;
            break;
        }
    }else{
        sumar = 1;
    }

    for (int i = 0; i < sumar; i++)
    {


        cout << "_____" << endl;
        cout << " * Minuto actual: " << contador << endl;
        cout << endl;

        mostrarPila();
```

```
        //Saco a los núcleos los procesos que arrancan en este minuto
        while (pila.cimaPila().tiempoInicio == contador && !pila.esVacia())
        {
            cout << "-----COMPRUEBO SI LOS NÚCLEOS ESTÁN LLENOS-----" <<
endl;
            cout << endl;

            // Compruebo si están todos llenos
            bool llenos = true;
            for (int j = 1; j <= listaNucleos->longitudLista(); j++)
            {
                if (listaNucleos->obtenerNodo(j)->NdeProcesosEnCola() < 3)
                {
                    llenos = false;
                    cout << "Núcleo " << j << " no está lleno, tiene " << listaNucleos-
>obtenerNodo(j)->NdeProcesosEnCola() << " procesos en cola." << endl;
                    cout << "Cola de núcleo " << j << ": ";
                    listaNucleos->obtenerNodo(j)->mostrarColaNucleo();
                    cout << endl;
                }

            }
            // Si es así, añado un nuevo núcleo
            if (llenos)
            {
                listaNucleos = listaNucleos->añadirDerecha(Nucleo());
                cout << "Están todos llenos, hemos añadido un nuevo núcleo" << endl;
```

```
            cout << "Ahora hay " << listaNucleos->longitudLista() << " núcleos" <<
endl;
        }
        cout << endl;



        cout << "-----PARA ENCOLAR COMPRUEBO CUAL ES EL NÚCLEO
CON MENOS COLA-----" << endl;

                        for (int j = 1; j <= listaNucleos->longitudLista(); j++)
                        {
                           cout << "Cola de núcleo " << j << ": ";
                           listaNucleos->obtenerNodo(j)->mostrarColaNucleo();
                           cout << endl;
                        }

        // Compruebo cual es el núcleo con menos procesos en cola y encolo
        Nucleo* menor = listaNucleos->obtenerNodo(1);
        for (int j = 1; j <= listaNucleos->longitudLista(); j++)
        {

            if (listaNucleos->obtenerNodo(j)->NdeProcesosEnCola() < menor-
>NdeProcesosEnCola())
            {
               cout << "El núcleo " << j << " tiene menos procesos en cola" << endl;
               menor = listaNucleos->obtenerNodo(j);
            }
        }
        menor->encolarProceso(pila.cimaPila());
        cout << endl;
```

```
            cout << "Encolo en el menor" << endl;


                        for (int j = 1; j <= listaNucleos->longitudLista(); j++)
                        {
                            cout << "Cola de núcleo " << j << ": ";
                            listaNucleos->obtenerNodo(j)->mostrarColaNucleo();
                            cout << endl;
                        }


            pila.desapilar();
        }
                        if (listaNucleos->longitudLista() < 0){
                        for (int j = 1; j <= listaNucleos->longitudLista(); j++)
                        {
                            cout << "Cola de núcleo " << j << ": ";
                            listaNucleos->obtenerNodo(j)->mostrarColaNucleo();
                            cout << endl;
                        }}


        for (int j = 1; j <= listaNucleos->longitudLista(); j++)
        {
            if (listaNucleos->obtenerNodo(j)->esVacio() && !listaNucleos-
>obtenerNodo(j)->obtenerColaNucleo()->esVacia())
            {
                cout << "Pongo a porcesar el proceso en el núcleo " << j << endl;
                listaNucleos->obtenerNodo(j)->procesar(listaNucleos->obtenerNodo(j)-
>obtenerColaNucleo()->desencolar());
            }
        }
```

```
                    if (listaNucleos->longitudLista() < 0){
                    for (int j = 1; j <= listaNucleos->longitudLista(); j++)
                    {
                       cout << "Cola de núcleo " << j << ": ";
                       listaNucleos->obtenerNodo(j)->mostrarColaNucleo();
                       cout << endl;
                    }}


if (!manual)
{

                    for (int j = 1; j <= listaNucleos->longitudLista(); j++)
                    {
                       cout << "Cola de núcleo " << j << ": ";
                       listaNucleos->obtenerNodo(j)->mostrarColaNucleo();
                       cout << endl;
                    }
   mostrarPila();
   detallesProcesosEjecucion();
   cout << endl;
}

for (int j = 1; j <= listaNucleos->longitudLista(); j++)
{
   int tiempo = listaNucleos->obtenerNodo(j)->actualizar();
   if (tiempo != 0)
   {
      procesosEjecutados += 1;
```

```
            }
            sumaTiempos += tiempo;
        }

        for (int j = 1; j <= listaNucleos->longitudLista(); j++)
        {
            listaNucleos->obtenerNodo(j)->obtenerColaNucleo()->actualizar();
        }

        int nucleosVacios = 0;
        for (int j = 1; j <= listaNucleos->longitudLista(); j++)
        {
            if (listaNucleos->obtenerNodo(j)->esVacio() && listaNucleos-
>obtenerNodo(j)->obtenerColaNucleo()->esVacia())
            {
                nucleosVacios += 1;
            }
        }

        for (int j = 1; j <= listaNucleos->longitudLista(); j++)
        {
            if (listaNucleos->obtenerNodo(j)->esVacio() && listaNucleos-
>obtenerNodo(j)->obtenerColaNucleo()->esVacia() && nucleosVacios > 2)
            {
                listaNucleos = listaNucleos->eliminarNodo(j);
                nucleosVacios -= 1;
            }
        }
```

```
        if (pila.esVacia() && (listaNucleos->longitudLista() == 1 && listaNucleos->primeroPtr()->esVacio()))
        {
            break;
        }
        contador += 1;
    }
}

cout << "Se han necesitado " << contador + 1 << " minutos para procesar todos los procesos." << endl;
cout << "Se han ejecutado " << procesosEjecutados << " procesos." << endl;

if (procesosEjecutados != 0)
{
    cout << "El tiempo medio de estancia de los procesos en el sistema operativo ha sido de " << (float)sumaTiempos/(float)procesosEjecutados << " minutos." << endl;
}else{
    cout << "No se ha ejecutado ningún proceso." << endl;
}

// Reinicio de variables para una posible nueva iteración
contador = 0;
sumaTiempos = 0;
procesosEjecutados = 0;
definirProcesos();
}
```

Algunas de partes de este código tienen como utilidad comprobar que las funciones se están llevando a cabo de manera satisfactoria, pero grosso modo vamos a señalar cuales son exactamente las funciones que lleva a cabo este bucle for y qué diferencias presenta respecto al anterior main de la primera parte.

La primera diferencia remarcable es que antes los procesos salían de la pila de procesos y se metían en la cola de prioridades. En esta nueva parte, salen de la pila y los procesos tienen que elegir en qué cola introducirse (ya que los núcleos tienen una cola de prioridad), y está implementado de tal manera que entre en aquella cola de procesos que tenga menos elementos. Si todas las colas están llenas, en ese caso se creará un núcleo nuevo y se introducirán en la cola de ese núcleo.

Una vez en las colas, se itera la lista de núcleos para introducir los procesos en los núcleos en caso de ser posible. Posteriormente, se vuelve a iterar la lista de núcleos para ejecutar la función actualizar (que se explicó en la primera parte) y devuelve el tiempo si ha terminado un proceso y 0 si no ha terminado.

Si ha terminado el proceso, suma 1 al total de procesos ejecutados (procesosEjecutados) y suma el tiempo (sumaTiempos) que ha tardado en ejecutarse (ambos elementos con los que se hace la media que se expresa como solución, esto se mantiene como en la primera parte).

Además, el proceso cuenta con la implementación de que si hay más de dos núcleos vacíos, los elimina.

Después vuelve a iterar la lista de los núcleos para ejecutar la función actualizar en sus colas (como en la cola de prioridades de la parte 1).

Y este es el funcionamiento de la aplicación con ambas partes.

TERCERA PARTE: IMPLEMENTACIÓN DE ÁRBOLES

Para esta tercera parte era necesaria la implementación de árboles de búsqueda binarios en el código. Una vez acabada la ejecución del proceso en la **parte 2**, el proceso debía introducirse en el nodo del árbol que tuviera el valor de su prioridad.

1'' Últimas Modificaciones

Para empezar a plantear las nuevas modificaciones, hubo primero que implementar la clase **"Árboles Binarios de Búsqueda para Procesos"** (ABBProcesos) con los siguientes atributos y métodos:

Atributos **privados**:

ListaProcesos raiz; *//Una lista con diferentes procesos dentro de cada nodo del árbol*

ABBProcesos* izquierda; *//Un puntero que apunta al hijo izquierdo de un nodo.*

ABBProcesos* derecha; *//Un puntero que apunta al hijo derecho de un nodo.*

Métodos **públicos**:

ABBProcesos(); *//Un constructor sin parámetros.*

ABBProcesos(int prioridad); *//Un constructor al que le pasas la prioridad del nodo.*

~ABBProcesos(); *//Un destructor.*

bool esVacio(); *//Devuelve un booleano que indica si la lista de procesos de un nodo está vacía.*

bool esHoja(); *//Devuelve booleano que indica si el nodo del árbol no tiene hijos.*

ListaProcesos* obtenerRaiz(); *//Una función que te devuelve un puntero a la lista de procesos de un determinado nodo.*

ABBProcesos* obtenerIzquierda(); *//Una función que te devuelve un puntero al hijo izquierdo de un determinado nodo.*

```
ABBProcesos* obtenerDerecha();    //Una función que te devuelve un puntero al hijo derecho de un determinado nodo.

void setRaiz(ListaProcesos listaProcesos);    //Permite colocar una lista de procesos en un nodo.
void setIzquierda(ABBProcesos*);    //Coloca un nodo hijo a la izquierda de un nodo.
void setDerecha(ABBProcesos*);    //Coloca un nodo hijo a la derecha de un nodo.

void insertar(Proceso proceso);    //Permite insertar un proceso en el árbol en su nodo correspondiente.
void borrarNodo(int prioridad, ABBProcesos* arbol); //Borra un nodo del árbol.
ABBProcesos copiarArbol(ABBProcesos* arbol);  //Realiza una copia del árbol.

ABBProcesos* obtenerNodoMenor();    //Devuelve el puntero al nodo con menor valor del árbol (el valor de los nodos se establece en función de la prioridad de los procesos).
ABBProcesos* obtenerNodoMayor();    //Devuelve el puntero al nodo con mayor valor del árbol (el valor de los nodos se establece en función de la prioridad de los procesos).
ABBProcesos* obtenerNodoEspecifico(int prioridad);    //Devuelve el puntero a un nodo del árbol con un valor específico.
int obtenerPrioridadMenor();    //Obtiene el menor valor que tenga un nodo en un árbol.
int obtenerPrioridadMayor();    //Obtiene el mayor valor que tenga un nodo en un árbol.
ListaProcesos* obtenerPrioridadEspecifica(int prioridad);    //Devuelve un puntero a la lista de procesos de un nodo en el árbol que tenga una prioridad específica.
```

```
int numProcesosEjecutados();        //Cuenta el número de procesos ejecutados en el transcurso de la aplicación.
float mediaTiempos();               //Devuelve la media de los tiempos de los procesos del sistema para ser ejecutados.
int mayorNumProcesos();             //Devuelve el número de procesos que tenga el nodo con más procesos en el árbol.
int menorNumProcesos();             //Devuelve el número de procesos que tenga el nodo con menos procesos en el árbol.
void mostrarInOrden();              //Muestra los valores del árbol en inorden.
void mostrarPromedioPreOrden();     //Muestra el promedio de cada nodo en preorden.
```

2'' Funciones Importantes

Una vez hemos abordado los métodos y atributos, procedemos a indagar en funciones más complejas que se han llevado a cabo para el desarrollo de la tercera parte de la práctica.

ABBProcesos::ABBProcesos()

```
{
   raiz = ListaProcesos(-1);
   izquierda = nullptr;
   derecha = nullptr;
}
```

En este caso, lo primero que se hizo fue un nodo con una prioridad -1 que tendrá como hijo derecho un nodo con la prioridad del siguiente proceso que se ejecute (ya que por definición en el enunciado ningún proceso tendrá prioridad negativa).

ABBProcesos::ABBProcesos(int prioridad)

```
{
   raiz = ListaProcesos(prioridad);
   izquierda = nullptr;
```

```
    derecha = nullptr;
}
```

La función crea un nodo sin hijos con la prioridad del proceso que termine de ejecutarse.

bool ABBProcesos::esVacio()

```
{
    return raiz.esVacia();
}
```

La función devuelve si la lista de procesos de un nodo es vacía.

bool ABBProcesos::esHoja()

```
{
    return (izquierda == nullptr && derecha == nullptr);
}
```

La función devuelve si el nodo es una hoja, comprobando para ello si no tiene hijos izquierdos ni derechos.

ListaProcesos* ABBProcesos::obtenerRaiz()

```
{
    return &raiz;
}
```

Devuelve un puntero a la lista de procesos del nodo.

ABBProcesos* ABBProcesos::obtenerIzquierda()

```
{
   return izquierda;
}
```

Devuelve un puntero al hijo izquierdo del nodo.

ABBProcesos* ABBProcesos::obtenerDerecha()

```
{
   return derecha;
}
```

Devuelve un puntero al hijo derecho del nodo.

void ABBProcesos::setRaiz(ListaProcesos listaProcesos)

```
{
   this->raiz = listaProcesos;
}
```

Sobreescribe la lista de procesos de un nodo.

void ABBProcesos::setIzquierda(ABBProcesos* nuevoIzquierda)

```
{
   this->izquierda = nuevoIzquierda;
}
```

Coloca un árbol hijo por la izquierda del nodo.

void ABBProcesos::setDerecha(ABBProcesos* nuevoDerecha)

```
{
   this->derecha = nuevoDerecha;
}
```

Coloca un árbol hijo por la derecha del nodo.

ABBProcesos* buscarPosicion(ABBProcesos* arbol, int prioridad)

```
{
   if (arbol == nullptr) {
      return nullptr;
   } else {
      if (arbol->obtenerRaiz()->obtenerPrioridad() == prioridad) {
         return arbol;
      } else {
         if (prioridad <= arbol->obtenerRaiz()->obtenerPrioridad()) {
            return buscarPosicion(arbol->obtenerIzquierda(), prioridad);
         } else {
            return buscarPosicion(arbol->obtenerDerecha(), prioridad);
         }
      }
   }
}
```

Te devuelve un puntero a un nodo con una prioridad determinada. Recursivamente, va descendiendo desde el nodo padre a los nodos hijos en función de si la prioridad es mayor o menor. Si llega a un nodo cuya prioridad sea igual al parámetro que se le pasa a la función, se devuelve el puntero al nodo. En caso de no existir el nodo, te devuelve un nullptr.

ABBProcesos* buscarPadre(int prioridad, ABBProcesos* arbol)

```
{
   if (arbol == nullptr) {
      return nullptr;
   } else {
      if (prioridad <= arbol->obtenerRaiz()->obtenerPrioridad()) {
         if (arbol->obtenerIzquierda() == nullptr || prioridad == arbol-
>obtenerIzquierda()->obtenerRaiz()->obtenerPrioridad()) {
            return arbol;
         } else {
            return buscarPadre(prioridad, arbol->obtenerIzquierda());
         }
      } else {
         if (arbol->obtenerDerecha() == nullptr || prioridad == arbol->obtenerDerecha()-
>obtenerRaiz()->obtenerPrioridad()) {
            return arbol;
         } else {
            return buscarPadre(prioridad, arbol->obtenerDerecha());
         }
      }
   }
}
```

Te devuelve un puntero al nodo que sea padre de un nodo que pasas como parámetro. Recursivamente, va descendiendo desde el nodo padre a los hijos. En el caso de que la prioridad que se le pasa a la función como parámetro sea menor o igual a la del nodo en el que esté, comprueba si el nodo no tiene hijo izquierdo (ya que, de ser así, se entiende que entonces el nodo en el que se encuentra debe ser el padre) o si la prioridad de su hijo izquierdo es igual a la que pasamos como parámetro. En ese caso, devuelve el puntero al nodo. En caso contrario, la función se llama a sí misma con el hijo izquierdo del nodo. Todo este proceso se realiza de igual manera con el hijo derecho en el caso de que la prioridad sea mayor.

ABBProcesos* ABBProcesos::obtenerNodoMenor()

```
{
   ABBProcesos* min = this;
   if (min->obtenerRaiz()->obtenerPrioridad() == -1) {
      min = min->obtenerDerecha();
   }
   while(min->obtenerIzquierda() != nullptr)
   {
      min = min->obtenerIzquierda();
   }
   return min;
}
```

Devuelve un puntero al nodo con menor valor del árbol. En el caso de que la prioridad del nodo sea -1 (es decir, en el caso de que nos encontremos en el nodo que se implementó al inicio de la práctica), obtiene su hijo derecho, y a partir de ahí baja recursivamente hasta que encuentre el último hijo izquierdo que no tenga hijo izquierdo (que será el nodo con menor valor). Una vez lo haya encontrado, devuelve el puntero a ese nodo.

ABBProcesos* ABBProcesos::obtenerNodoMayor()

```
{
   ABBProcesos* max = this;
   while(max->obtenerDerecha() != nullptr)
   {
      max = max->obtenerDerecha();
   }
   return max;
}
```

Realiza los mismos pasos que la función anterior, salvo que esta vez no tiene que comprobar si la prioridad del nodo es -1 ya que accederá al hijo derecho del nodo constantemente hasta encontrar un hijo derecho que no tenga hijo derecho (que será el nodo con mayor valor del árbol) y devolverá un puntero a él.

ABBProcesos* ABBProcesos::obtenerNodoEspecifico(int prioridad)

```
{
   return buscarPosicion(this, prioridad);
}
```

Busca un nodo en el árbol tras pasarlo una prioridad como parámetro.

int ABBProcesos::obtenerPrioridadMenor()

```
{
   return this->obtenerNodoMenor()->obtenerRaiz()->obtenerPrioridad();
}
```

Obtiene el menor valor que tenga un nodo en un árbol.

int ABBProcesos::obtenerPrioridadMayor()

```
{
   return this->obtenerNodoMayor()->obtenerRaiz()->obtenerPrioridad();
}
```

Obtiene el mayor valor que tenga un nodo en un árbol.

ListaProcesos* ABBProcesos::obtenerPrioridadEspecifica(int prioridad)

```
{
   ABBProcesos* nodo = buscarPosicion(this, prioridad);
   if (nodo != nullptr) {
      return nodo->obtenerRaiz();
   } else {
      return nullptr;
   }
}
```

Te devuelve la lista de procesos de un determinado nodo en el árbol. En caso de que el nodo no exista, te devuelve un nullptr.

ABBProcesos* crearNodo(ABBProcesos* arbol, int prioridad)

```
{
   if (prioridad <= arbol->obtenerRaiz()->obtenerPrioridad())
   {
      arbol->setIzquierda(new ABBProcesos(prioridad));
      return arbol->obtenerIzquierda();
   }
   else
   {
      arbol->setDerecha(new ABBProcesos(prioridad));
      return arbol->obtenerDerecha();
   }
}
```

Crea un nuevo nodo hijo en el árbol. Si la prioridad del nodo que se quiere insertar es menor que la del padre, se insertará por la izquierda. En caso contrario, se insertará por la derecha.

ABBProcesos* insertarEn(ABBProcesos* arbol, int prioridad)

```
{
   ABBProcesos* raiz = buscarPosicion(arbol, prioridad);
   if (raiz == nullptr) {
      raiz = crearNodo(buscarPadre(prioridad, arbol), prioridad);
   }
   return raiz;
}
```

Busca la posición en la que insertar un nodo en el árbol.

void ABBProcesos::insertar(Proceso proceso)

```
{
   insertarEn(this, proceso.prioridad)->obtenerRaiz()->añadirDerecha(proceso);
}
```

Inserta un nodo en el árbol llamando a la función anterior para saber dónde colocarlo.

```
void ABBProcesos::borrarNodo(int prioridad, ABBProcesos* arbol)
{
   ABBProcesos* hijo = buscarPosicion(arbol, prioridad);
   if (hijo != nullptr) {
      ABBProcesos* padre = buscarPadre(prioridad, arbol);
      if (hijo->esHoja()) {
         if(prioridad <= padre->obtenerRaiz()->obtenerPrioridad()) {
```

```
            padre->setIzquierda(nullptr);
        } else {
            padre->setDerecha(nullptr);
        }
        delete hijo;
    } else if (hijo->obtenerIzquierda() == nullptr) {
        ABBProcesos* aux = hijo->obtenerDerecha();
        *hijo = *hijo->obtenerDerecha();
        delete aux;
    } else if (hijo->obtenerDerecha() == nullptr) {
        ABBProcesos* aux = hijo->obtenerIzquierda();
        *hijo = *hijo->obtenerIzquierda();
        delete aux;
    } else {
        ABBProcesos* max = hijo->obtenerIzquierda()->obtenerNodoMayor();
        ListaProcesos raizMax = *max->obtenerRaiz();
        borrarNodo(max->obtenerRaiz()->obtenerPrioridad(), hijo);
        hijo->setRaiz(raizMax);
    }
  }
}
```

Elimina un nodo del árbol. En primer lugar, busca el nodo que se desee eliminar acorde a la prioridad pasada como parámetro. Una vez lo tiene, examina los siguientes casos:

-Si el nodo no tiene hijos (es una hoja), el puntero de su padre apunta a nullptr y se elimina así el espacio.

-Si el nodo solo tiene un hijo, se sustituye la información del nodo que se quiere borrar por la de su hijo.

-Si tiene más de un hijo, se sustitye la información del nodo que se quiere borrar por la del nodo mayor por la izquierda, y se borra dicho nodo.

ABBProcesos ABBProcesos::copiarArbol(ABBProcesos* arbol)

```
{
   ABBProcesos copia = ABBProcesos();
   copia.setRaiz(*arbol->obtenerRaiz());
   if (arbol->obtenerDerecha() != nullptr) {
      copia.setDerecha(new ABBProcesos(copiarArbol(arbol->obtenerDerecha())));
   }
   if (arbol->obtenerIzquierda() != nullptr) {
      copia.setIzquierda(new ABBProcesos(copiarArbol(arbol->obtenerIzquierda())));
   }
   return copia;
}
```

Itera el árbol y realiza una copia de este colocando copias de los hijos en los nodos que corresponden.

---

int siguienteInOrden(ABBProcesos* arbol, int actual = -1)

```
{
   ABBProcesos ABBaux = arbol->copiarArbol(arbol);
   while (ABBaux.obtenerPrioridadMenor() != actual)
   {
      ABBaux.borrarNodo(ABBaux.obtenerPrioridadMenor(), &ABBaux);
   }
   ABBaux.borrarNodo(ABBaux.obtenerPrioridadMenor(), &ABBaux);
   return ABBaux.obtenerPrioridadMenor();
}
```

Te devuelve cual es el siguiente nodo en inorden dada una prioridad. Para ello, emplea una copia del árbol la cual va itera mientras borra los nodos necesarios.

void ABBProcesos::mostrarInOrden()

```
{
   cout << "InOrden: " << endl;
   int prioridadActual = this->obtenerPrioridadMenor();
   while (prioridadActual != this->obtenerPrioridadMayor())
   {
      this->obtenerPrioridadEspecifica(prioridadActual)->mostrarListaProcesos();
      prioridadActual = siguienteInOrden(this, prioridadActual);
   }
   this->obtenerPrioridadEspecifica(prioridadActual)->mostrarListaProcesos();
}
```

Itera el árbol recursivamente borrando nodos hasta encontrar el nodo con la menor prioridad. Una vez hecho esto, te devuelve todos los elementos del árbol colocados en inorden. Al ser el parámetro actual = -1, una vez más se tiene en cuenta la restricción de que la prioridad del primer nodo que creamos es -1