

ESTRUCTURAS DE DATOS

Práctica de Laboratorio - 1



Patricia Cuesta Ruiz - 03211093V
Isabel Blanco Martínez - 04866897M

ÍNDICE

1. Especificación concreta de la interfaz de los TAD's implementados

1.1 TAD's Creados

1.2 Definición de las operaciones del TAD (Nombre, argumentos y retorno)

2. Solución adoptada: descripción de las dificultades adoptadas

3. Diseño de la relación entre las clases de los TAD implementados

3.1 Diagrama UML

3.2 Explicación de los métodos más destacados

3.3 Explicación del comportamiento del programa

4. Código fuente

5. Bibliografía

1. ESPECIFICACIÓN CONCRETA DE LA INTERFAZ DE LOS TAD's

Para la correcta resolución de esta práctica hemos utilizado las siguientes estructuras de datos: pila, cola y lista. Cada una de estas estructuras tiene su propia clase en el proyecto, al igual que tienen su propia clase los nodos que forman cada una de estas.

Además de los TADs principales como PilaPedidos, ColaPedidos y Lista Pedidos, nos encontramos con las clases Pedido y Correos.

1.1 TAD's creados

1.1.1 Pila

Una pila P es una estructura lineal tal que las inserciones, las consultas y las eliminaciones solo se permiten en un único punto. Las pilas son estructuras denominadas LIFO (Last In, First Out), nombre que hace referencia al modo en que se accede a los elementos.

Atributos

- **NodoPila* cima:** un puntero de tipo NodoPila que almacenará la información del pedido en la cima de la pila (es decir, el siguiente en ser desapilado) y un puntero al siguiente pedido en la pila.

Operaciones

- **PilaPedidos() // ~PilaPedidos()** - constructor y destructor
- **void apilar(Pedido p)** - función encargada de apilar (insertar dentro de la pila) el pedido que recibe por parámetro.
- **void desapilar()** - función encargada de desapilar (sacar fuera de la pila) al pedido actualmente en la cima.
- **void mostrarCima()** - función encargada de imprimir los valores de la cima de la pila.
- **void apilarPrioridad(Pedido p)** - función encargada de situar el pedido en una posición u otra dependiendo del tipo de cliente que ha hecho el pedido.

- `int comprobarCimaRegistrado()` - función que devuelve 0 si la pila está vacía, 1 si el primer elemento es un pedido de un cliente VIP o NVIP y 2 si es de un cliente NR.
- `void esVacia()` - función que devuelve un booleano en función de si la pila está o no vacía.
- `Pedido getCimaPedido()` - getter para obtener únicamente el pedido de la cima.

```

1  #ifndef PILAPEDIDOS_H
2  #define PILAPEDIDOS_H
3  #include "NodoPila.h"
4  #include "Pedido.h"
5
6  class PilaPedidos
7  {
8      private:
9          NodoPila* cima;
10
11     public:
12         // Constructores y destructores
13         PilaPedidos();
14         ~PilaPedidos();
15         // Métodos
16         void apilar(Pedido p);
17         void desapilar();
18         void mostrarCima();
19         void apilarPrioridad(Pedido p);
20         int comprobarCimaRegistrado();
21         // Getters y Setters
22         bool esVacia();
23         Pedido getCimaPedido();
24     };
25
26 #endif // PILAPEDIDOS_H

```

1.1.2 NodoPila

Este nodo contendrá información sobre un pedido de la pila y un puntero al siguiente de este.

Atributos

- **Pedido pedido_np**
- **NodoPila* siguiente_np**

Operaciones

- `NodoPila(Pedido p, NodoPila* sig)` - constructor con los parámetros necesarios, en este caso pedido y un puntero al siguiente, que serán inicializados a NULL.
- `~NodoPila()` - destructor

```
1  #ifndef NODOPILA_H
2  #define NODOPILA_H
3  #include "Pedido.h"
4
5  class NodoPila
6  {
7      friend class PilaPedidos;
8
9      private:
10         Pedido pedido_np;
11         NodoPila* siguiente_np;
12
13     public:
14         NodoPila(Pedido p, NodoPila* sig);
15         ~NodoPila();
16
17     };
18
19 #endif // NODOPILA_H
```

1.1.3 Cola

Esta estructura de datos formada por nodos, tiene un funcionamiento FIFO (First in, First Out), es decir, el primero que llega es el primero que sale. Esto significa que el que metamos siempre va a ser el último de la cola, y el que vamos a sacar siempre va a ser el primero. Por esta razón la clase Cola tiene dos atributos: *primero* y *último*, ambos de tipo puntero *NodoCola*, ya que son nodos y actúan como tales.

Atributos

- `NodoCola* primero`
- `NodoCola* ultimo`

Operaciones

- ColaPedidos() // ~ColaPedidos() - constructor y destructor
- void encolar(Pedido p) - función encargada de encolar (insertar dentro de la cola) el pedido que recibe por parámetro.
- void desencolar() - función encargada de desencolar el primer pedido de la cola.
- void mostrarCola() - procedimiento que muestra por pantalla el estado actual de la cola.
- Pedido getPrimerPedido() - getter que recoge únicamente la información del pedido en la primera posición de la cola.
- NodoCola* getPrimero() - getter que recoge únicamente la información del nodo primero de la cola.
- NodoCola* getUltimo() - getter que recoge únicamente la información del nodo último de la cola.
- bool esVacia() - función que retorna un booleano en función de si la cola está o no vacía.

```
1  #ifndef COLAPEDIDOS_H
2  #define COLAPEDIDOS_H
3  #include "NodoCola.h"
4  #include "Pedido.h"
5
6  class ColaPedidos
7  {
8      private:
9          NodoCola* primero;
10         NodoCola* ultimo;
11
12     public:
13         // Constructores y destructores
14         ColaPedidos();
15         ~ColaPedidos();
16         // Métodos
17         void encolar(Pedido p);
18         void desencolar();
19         void mostrarCola();
20         // Getters y Setters
21         Pedido getPrimerPedido();
22         NodoCola* getPrimero();
23         NodoCola* getUltimo();
24         bool esVacia();
25     };
26
27 #endif // COLAPEDIDOS_H
28
29
```

1.1.4 NodoCola

Este nodo contiene información sobre un pedido dentro de la cola y un puntero al siguiente.

Atributos

- **NodoCola* siguiente_nc**: ya que la cola funciona con FIFO, solo es necesario saber el nodo siguiente y no el anterior, pues partimos desde el primer nodo de la cola y vamos avanzando, nunca retrocedemos a la hora de reasignar un nuevo nodo primero.
- **Pedido pedido_nc**

Operaciones

- **NodoCola(Pedido p, NodoCola *sig)** - constructor con los parámetros necesarios, en este caso un pedido y un puntero al siguiente, que serán inicializados a NULL.
- **~NodoCola()** - destructor

```
1  #ifndef NODOCOLA_H
2  #define NODOCOLA_H
3  #include "Pedido.h"
4
5  class NodoCola
6  {
7      friend class ColaPedidos;
8
9      private:
10         NodoCola* siguiente_nc;
11         Pedido pedido_nc;
12
13     public:
14         NodoCola(Pedido p, NodoCola *sig);
15         ~NodoCola();
16
17     };
18
19 #endif // NODOCOLA_H
20
```

1.1.5 ListaPedido

Hemos usado este TAD como una lista en la cual se van almacenando los pedidos según llegan previamente ordenados por la función `apilarPrioridad`.

Permanecerán en esta lista hasta que el tiempo de preparación de dicho pedido sea 0 y se envíe (desenlistar).

Atributos

- **NodoLista* primero:** un puntero de tipo `NodoLista` que apunta a la primera posición de la lista (al primer pedido).
- **NodoLista* ultimo:** un puntero de tipo `NodoLista` que apunta a la última posición de la lista.

Operaciones

- `ListaPedidos()` - constructor
- `~ListaPedidos()` - destructor
- `void enlistar(Pedido p)` - función encargada de insertar dentro de la lista el pedido que se pasa por parámetro.
- `void insertarDerecha(Pedido p)` - función que inserta por la derecha de la lista el pedido que se le pasa por parámetro.
- `void insertarIzquierda(Pedido p)` - función que inserta por la izquierda de la lista el pedido que se le pasa por parámetro.
- `void desenlistar()` - función encargada de sacar de la lista el pedido que encuentra el primero.
- `void mostrarLista()` - función encargada de imprimir por pantalla toda la información de la lista actual.
- `bool esVacia()` - función que retorna un booleano en función de si la cola está o no vacía.


```

1  #ifndef LISTAPEDIDOS_H
2  #define LISTAPEDIDOS_H
3  #include "NodoLista.h"
4
5  class ListaPedidos
6  {
7      private:
8          NodoLista* primero;
9          NodoLista* ultimo;
10     public:
11         // Constructores y Destructores
12         ListaPedidos();
13         ~ListaPedidos();
14         // Métodos
15         void enlistar(Pedido p);
16         void insertarDerecha(Pedido p);
17         void insertarIzquierda(Pedido p);
18         void desenlistar();
19         void mostrarLista();
20         // Getters y Setters
21         bool esVacia();
22     };
23
24 #endif // LISTAPEDIDOS_H

```

1.1.6 NodoLista

Este nodo contiene la información sobre un pedido dentro de la lista, un puntero al anterior y otro puntero al siguiente de este.

Atributos

- **Pedido pedido_nl**
- **NodoLista* anterior_nl**
- **NodoLista* siguiente_nl**

Operaciones

- `NodoLista(Pedido p, NodoLista* ant, NodoLista* sig)` - constructor con los parámetros necesarios, en este caso un pedido, un puntero al anterior y otro al siguiente, que serán inicializados a NULL.
- `~NodoLista()` - destructor

```
1  #ifndef NODOLISTA_H
2  #define NODOLISTA_H
3  #include "Pedido.h"
4
5  class NodoLista
6  {
7      friend class ListaPedidos;
8
9      private:
10         Pedido pedido_nl;
11         NodoLista* anterior_nl;
12         NodoLista* siguiente_nl;
13     public:
14         NodoLista(Pedido p, NodoLista* ant, NodoLista* sig);
15         ~NodoLista();
16     };
17
18 #endif // NODOLISTA_H
```

2. SOLUCIÓN ADOPTADA: DESCRIPCIÓN DE LAS DIFICULTADES ENCONTRADAS

A grandes rasgos, los problemas más grandes que hemos encontrado han sido los siguientes.

A lo largo del desarrollo del proyecto hemos ido comprobando con las funciones

- mostrarCima()
- mostrarCola()
- mostrarLista()

para ir comprobando que según íbamos introduciendo o eliminando datos de dichas estructuras se hacía de forma correcta y mostraba lo que pensábamos que tenía que ir mostrando.

Además, encontramos cierta dificultad a la hora de introducir los pedidos en la lista, ya que había que introducir tres que fueran registrados (VIP y NVIP) y uno no registrado (NR) y tras muchos intentos y muchos bucles infinitos conseguimos que funcionase como debía hacerlo, comprobando a mano también cómo se tenían que ir moviendo los pedidos de un TAD a otro.

Otra dificultad que encontramos fue el crear un código que leyese bien los pedidos dados por un txt pero finalmente conseguimos resolverlo con rapidez.

La imagen está también en la carpeta con el resto de archivos a entregar para que se pueda ver mejor.

3.2 EXPLICACIÓN DE LOS MÉTODOS MÁS DESTACADOS

A lo largo del proyecto se han ido implementando diferentes métodos que consideraremos como los métodos más destacados. Entre ellos encontramos:

PilaPedidos - Método DESAPILAR

```
19 void PilaPedidos::apilar(Pedido p)
20 {
21     NodoPila* nuevo = new NodoPila(p, cima);
22     cima = nuevo; //la cima es el nuevo pedido
23 }
```

Este método consiste en establecer un nuevo nodo cima, que será el siguiente a la cima actual (es decir, el que vamos a desapilar) haciendo uso de un nuevo nodo auxiliar, siempre y cuando la lista no esté vacía.

Esto se consigue estableciendo que la nueva cima será un puntero al siguiente del nodo auxiliar en el que guardamos la cima que vamos a borrar.

PilaPedidos - Método APILARPRIORIDAD

```
46 void PilaPedidos::apilarPrioridad(Pedido p) // hay que comprobar si es erroneo y si lo es pasarlo por la funcion
47 {
48     PilaPedidos pilaAux;
49     if (esVacia()) {
50         apilar(p);
51     } else {
52         if (p.getTipoCliente() == "VIP" || p.getTipoCliente() == "NVIP") {
53             apilar(p);
54         } else { // para cuando son NR
55             while (!esVacia() && getCimaPedido().getTipoCliente() != "NR") {
56                 pilaAux.apilar(getCimaPedido());
57                 desapilar();
58             }
59             apilar(p); // apilamos en esa posicion el nuevo pedido que entra
60             while (!pilaAux.esVacia()) { // sacamos de nueva todos los pedidos de Registrados en la pila normal
61                 apilar(pilaAux.getCimaPedido());
62                 pilaAux.desapilar();
63             }
64         }
65     }
66 }
```

El método de apilarPrioridad tiene dos opciones:

- Si la pila es vacía simplemente apila
- Si no es vacía, entonces comprueba el tipo de cliente de pedido. Si es un cliente *Registrado* (Vip o NVIP) entonces simplemente apila. Si por el contrario es un cliente *No Registrado*, mientras no sea vacía apilaremos en una pila auxiliar y desapilamos de la pila. Después de apilar en esa posición el pedido, volvemos a introducir los pedidos de la pila auxiliar a la pila.

ColaPedidos - Método ENCOLAR

```
21 void ColaPedidos::encolar(Pedido p)
22 {
23     NodoCola* aux = new NodoCola(p, NULL);
24     if (primero) {
25         ultimo->siguiente_nc = aux;
26     } else {
27         primero = aux;
28     }
29     ultimo = aux;
30 }
```

Este método consiste en crear un nuevo `NodoCola`, cuyo pedido será el que le pasemos como parámetro a la función y el puntero al siguiente será nulo, puesto que el nuevo pedido que será encolado también será el último. Tiene dos opciones:

- Si ya existe un primero en la cola (no está vacía), entonces el siguiente al último pedido existente será el nuevo pedido, y además, también el último.
- Si no existe un primer pedido (la cola está vacía), entonces este nuevo pedido será el primero y además, el último.

ColaPedidos - Método DESENCOLAR

```
32 void ColaPedidos::desencolar()
33 {
34     NodoCola* aux; //guardamos primero
35     if (primero != NULL) { //podemos desencolar porque la cola no está vacía
36         aux = primero;
37         primero = aux->siguiente_nc; // el nuevo primero es el siguiente al que desencolo
38         delete aux;
39     } else {
40         cout << "La cola está vacía" << endl;
41     }
42 }
```

Lo que este método se encargará de hacer, siempre y cuando la cola no esté vacía, será almacenar en un nodo auxiliar el primer pedido de la cola (el cual vamos a desencolar) y establecemos que el nuevo primero será el siguiente al actual.

Lista - Método ENLISTAR

```
18 void ListaPedidos::enlistar(Pedido p)
19 {
20     insertarDerecha(p);
21     //insertarIzquierda(p);
22 }
```

Este método hace una llamada a la función InsertarDerecha que mencionaremos a continuación. En vez de InsertarDerecha podríamos invocar InsertarIzquierda, ya que las listas nos permiten insertar elementos en ellas por ambos extremos.

Lista - Método INSERTARDERECHA

```
24 void ListaPedidos::insertarDerecha(Pedido p)
25 {
26     NodoLista* nodo_nuevo;
27     if (!primero) { //lista vacia
28         nodo_nuevo = new NodoLista (p, NULL, NULL);
29         primero = nodo_nuevo;
30     } else {
31         nodo_nuevo = new NodoLista (p, ultimo, NULL);
32         ultimo->siguiente_nl = nodo_nuevo;
33     }
34     ultimo = nodo_nuevo; // el NUEVO ultimo es el nuevo que meto
35 }
```

En el método insertarDerecha tenemos dos opciones: En el caso de que la lista esté vacía, lo que haremos será crear un NodoLista nuevo, cuyo pedido será el que le pasemos como parámetro a la función y los punteros al anterior y al último serán NULL, puesto que no existen.

Este nuevo nodo, además, será el primero y el último de la lista. En el caso de que la lista no esté vacía, estableceremos también ese nuevo nodo pero indicando esta vez que el puntero al anterior será el actual último elemento de la lista, además de que este también será el nuevo último.

Lista - Método INSERTARIZQUIERDA

```
37 void ListaPedidos::insertarIzquierda(Pedido p) //no lo utilizamos pero es una funcion propia de listas
38 {
39     NodoLista* nodo_nuevo;
40     if (!primero) { //lista vacia
41         nodo_nuevo = new NodoLista (p, NULL, NULL);
42     } else {
43         nodo_nuevo = new NodoLista (p, NULL, primero);
44     }
45     primero = nodo_nuevo; // el NUEVO primero es el nuevo que esta
46 }
```

En el método `insertarIzquierda`, lo único que cambia respecto al método anterior es que el puntero al siguiente será el actual primer elemento de la lista y actualizaremos ese *primero* al nuevo que estamos insertando.

Lista - Método DESENLISTAR

```
48 void ListaPedidos::desenlistar() // Sacamos hacia el primer pedido de la lista
49 {
50     NodoLista* nodo_aux; //guardamos primero
51     if (primero != NULL) { //podemos desenlistar porque la lista no esta vacia
52         nodo_aux = primero;
53         primero = nodo_aux->siguiente_n1; // El nuevo primero es el siguiente al que desenlisto
54         delete nodo_aux; // borrarlo de memoria
55     } else {
56         cout << "La lista esta vacia" << endl;
57     }
58 }
```

Para una lista no vacía, lo que hará será almacenar el primer elemento de la lista en un nodo auxiliar de tipo *NodoLista*, y estableceremos un nuevo *primero*, que será el siguiente al actual que vamos a desenlistar.

3.3 EXPLICACIÓN DEL COMPORTAMIENTO DEL PROGRAMA

Nuestro programa funciona de tal forma que en el main solo inicializamos tres variables (además de las correspondientes para poder leer el txt):

- Correos correos
- ColaPedidos colaR
- ColaPedidos colaNR

y llamamos a la función gestionCorreos, creada en el objeto Correos, que se encargará de ir simulando el envío de los pedidos y el paso del tiempo.

El programa nada más empezar lee el txt y va guardando los pedidos en dos colas diferentes dependiendo de si el cliente es VIP, NVIP o NR.

```
11 int main()
12 {
13     // VARIABLES //
14
15     Correos correos;
16     ColaPedidos colaR;
17     ColaPedidos colaNR;
18
19     // CÓDIGO PARA LEER TXT //
20
21     string desc;
22     string nombre;
23     string dir;
24     string type;
25     long long creditCard;
26     int time;
27     ifstream fe;
28     string str;
29     fe.open("D:/Universidad/2º/1º Cuatrimestre/Estructura de Datos 2.0/Laboratorio/Práctica 1/datos.txt", ios::in);
30     if (fe.fail()) {
31         cout << "No se pudo abrir el archivo." << endl;
32         exit(1);
33     } else {
34         cout << "PEDIDOS" << endl;
35         do {
36             getline(fe, str);
37             str.empty();
38             long posicion;
39             posicion = str.find("/");
40             desc = str.substr(0, posicion);
41             str.erase(0, posicion + 2);
42             long posicion1;
43             posicion1 = str.find("/");
44             nombre = str.substr(0, posicion1);
45             str.erase(0, posicion1 + 2);
46             long posicion2;
47             posicion2 = str.find("/");
48             dir = str.substr(0, posicion2);
49             str.erase(0, posicion2 + 2);
```

```

50     long posicion3;
51     posicion3 = str.find("/");
52     type = str.substr(0,posicion3);
53     str.erase(0,posicion3 + 2);
54     long posicion4;
55     posicion4 = str.find("/");
56     creditCard = std::stoll(str.substr(0,posicion4));
57     str.erase(0,posicion4 + 2);
58     long posicion5;
59     posicion5 = str.find("/");
60     time = stoi(str.substr(0, posicion5));
61     str.erase(0,posicion5 + 2);
62     Pedido pedido(desc, nombre, dir, type, creditCard, time);
63     if (pedido.getTipoCliente() == "VIP" || pedido.getTipoCliente() == "NVIP"){
64         cout << pedido.imprimir() << endl;
65         colaR.encolar(pedido);
66     }
67     else if(pedido.getTipoCliente() == "NR"){
68         cout << pedido.imprimir() << endl;
69         colaNR.encolar(pedido);
70     }
71     } while (!fe.eof());
72     fe.close();
73
74     // FIN LECTURA TXT //
75
76     cout << "" << endl;
77     correos.gestionCorreos(colaR, colaNR);
78 }
79

```

Como se puede observar, toda la estructura del funcionamiento de nuestro programa está definida en una función en la clase Correos, gestionCorreos(ColaPedidos colaRegistrados, ColaPedidos colaNoRegistrados), por la que pasamos las dos colas en las que se almacenan los pedidos leídos del txt. Esta función lleva a cabo toda la ejecución del programa:

- En primer lugar instanciamos todos los TAD's (ListaPedidos listaPrepEnviar, PilaPedidos pilaPedidosErroneos) que vamos a usar, además de los objetos que usaremos (Pedido pedido, pedidoC).
- Después, insertamos los cuatro primeros elementos en la lista listaPrepEnviar, siendo tres de ellos VIP o NVIP y otro NR. Como podemos ver, si el pedido es correcto lo inserta en la lista y si es erróneo lo apila en la pila pilaPedidosErroneos.

```

26     if (!colaRegistrados.esVacia()) {
27         int contR = 0;
28         while (contR != 3) {
29             if (!colaRegistrados.esVacia()) {
30                 pedido = colaRegistrados.getPrimerPedido();
31                 colaRegistrados.desencolar();
32                 if (pedido.comprobarPedidoErroneo()) {
33                     pilaPedidosErroneos.apilarPrioridad(pedido);
34                     cout << "Se apila en pila de pedidos erroneos: " << pedido.imprimir() << endl;
35                 } else {
36                     listaPrepEnviar.enlistar(pedido);
37                     cout << "El pedido registrado introducido en la lista es: " << pedido.imprimir() << endl;
38                     contR++;
39                 }
40             } else {
41                 contR = 3;
42             }
43         }
44     }

```

```

45     if (!colaNoRegistrados.esVacia()) {
46         int contNR = 0;
47         while (contNR != 1) {
48             if (!colaNoRegistrados.esVacia()) {
49                 pedido = colaNoRegistrados.getPrimerPedido();
50                 colaNoRegistrados.desencolar();
51                 if (pedido.comprobarPedidoErroneo()) {
52                     pilaPedidosErroneos.apilarPrioridad(pedido);
53                     cout << "Se apila en pila de pedidos erroneos: " << pedido.imprimir() << endl;
54                 } else {
55                     listaPrepEnviar.enlistar(pedido);
56                     cout << "El pedido no registrado introducido en la lista es: " << pedido.imprimir() << endl;
57                     contNR++;
58                 }
59             } else {
60                 contNR = 1;
61             }
62         }
63     }

```

- A continuación, hacemos un bucle *while* en el que mientras ninguno de los TAD's (listaPrepEnviar, pilaPedidosErroneos, colaRegistrados, colaNoRegistrados) esté vacío se ejecute todo el rato. También inicializaremos una variable *int* que irá guardando el tiempo de preparación de cada envío para que al final de la ejecución muestre el tiempo total que han tardado todos los pedidos en ser enviados.
- En primer lugar, coge el primer elemento de la lista que si está preparado para enviarlo lo envía y lo desenlista, sino, va reduciendo el tiempo de preparación del envío simulando el paso de los minutos para luego enviarlo, es decir desenlistarlo.

```

64     int tiempoTotal = 0;
65     while (!listaPrepEnviar.esVacia() || !colaRegistrados.esVacia() || !colaNoRegistrados.esVacia() || !pilaPedidosErroneos.esVacia()) {
66         // Procesamos el primer pedido de la lista
67         if (!listaPrepEnviar.esVacia()) {
68             cout << "" << endl;
69             pedido = listaPrepEnviar.getPrimerPedido();
70             cout << "El pedido " << pedido.imprimir() << " entra en preparacion" << endl;
71             while (pedido.getTiempoPrepEnvio() != 0) {
72                 cout << "Al pedido " << pedido.imprimir() << " le faltan " << pedido.getTiempoPrepEnvio() << " minutos" << endl;
73                 pedido.reducirTiempoPrepEnvio();
74                 tiempoTotal++;
75             }
76             cout << "El pedido " << pedido.imprimir() << " esta listo para ser enviado" << endl;
77             cout << "" << endl;
78             listaPrepEnviar.desenlistar(); // Enviamos el primer pedido de la lista
79         }

```

- Lo siguiente que hace el programa es, al igual que antes, volver a introducir otros tres pedidos de clientes registrados.

```

80     // Introducimos tres pedidos nuevos de la cola de registrados
81     if (!colaRegistrados.esVacia()) {
82         int contCR = 0;
83         while (contCR != 3) {
84             if (!colaRegistrados.esVacia()) {
85                 pedidoCR = colaRegistrados.getPrimerPedido();
86                 colaRegistrados.desencolar();
87                 if (pedidoCR.comprobarPedidoErroneo()) {
88                     pilaPedidosErroneos.apilarPrioridad(pedidoCR);
89                     cout << "Se apila en pila de pedidos erroneos: " << pedidoCR.imprimir() << endl;
90                 } else {
91                     listaPrepEnviar.enlistar(pedidoCR);
92                     cout << "El pedido registrado introducido en la lista es: " << pedidoCR.imprimir() << endl;
93                     contCR++;
94                 }
95             } else {
96                 contCR = 3;
97             }
98         }
99     }

```

- Luego comprueba que la cola de no registrados y la pila de erróneos no estén vacías. Entonces, si hay un pedido de un cliente registrado en la pila, lo completa y lo mete en la lista, sino, coge un pedido de la cola de clientes no registrados y sino coge un pedido de un cliente no registrado de la pila de erróneos y lo completa.

```

100 // Introducimos un pedido de la cola de no registrados o de la pila de pedidos erroneos
101 if (!pilaPedidosErroneos.esVacia() || !colaNoRegistrados.esVacia()) {
102     if (pilaPedidosErroneos.comprobarCimaRegistrado() == 1) {
103         pedidoPE = pilaPedidosErroneos.getCimaPedido();
104         pilaPedidosErroneos.desapilar();
105         pedidoPE.completarPedido();
106         cout << "Se corrige el pedido " << pedidoPE.imprimir() << endl;
107         listaPrepEnviar.enlistar(pedidoPE);
108         cout << "El pedido registrado introducido en la lista es: " << pedidoPE.imprimir() << endl;
109     } else if (!colaNoRegistrados.esVacia()) {
110         int contCNR = 0;
111         while (contCNR != 1) {
112             if (!colaNoRegistrados.esVacia()) {
113                 pedidoCNR = colaNoRegistrados.getPrimerPedido();
114                 colaNoRegistrados.desencolar();
115                 if (pedidoCNR.comprobarPedidoErroneo()) {
116                     pilaPedidosErroneos.apilarPrioridad(pedidoCNR);
117                     cout << "Se apila en pila de pedidos erroneos: " << pedidoCNR.imprimir() << endl;
118                 } else {
119                     listaPrepEnviar.enlistar(pedidoCNR);
120                     cout << "El pedido no registrado introducido en la lista es: " << pedidoCNR.imprimir() << endl;
121                     contCNR++;
122                 }
123             } else {
124                 contCNR = 1;
125             }
126         }
127     } else {
128         pedidoPE = pilaPedidosErroneos.getCimaPedido();
129         pilaPedidosErroneos.desapilar();
130         pedidoPE.completarPedido();
131         cout << "Se corrige el pedido " << pedidoPE.imprimir() << endl;
132         listaPrepEnviar.enlistar(pedidoPE);
133         cout << "El pedido no registrado introducido en la lista es: " << pedidoPE.imprimir() << endl;
134     }
135 }
136
137 }

```

- Por último y no menos importante, el programa devuelve el tiempo total de envío de todos los pedidos.

```

142 cout << "El tiempo total es: " << tiempoTotal << " minutos" << endl;
143 }

```

4. CÓDIGO FUENTE

A continuación se incluirán las capturas de pantalla en las que se puede visualizar el código fuente de nuestro proyecto.

- PilaPedidos

a) PilaPedidos.h

```
1  #ifndef PILAPEDIDOS_H
2  #define PILAPEDIDOS_H
3  #include "NodoPila.h"
4  #include "Pedido.h"
5
6  class PilaPedidos
7  {
8      private:
9          NodoPila* cima;
10
11      public:
12          // Constructores y destructores
13          PilaPedidos();
14          ~PilaPedidos();
15          // Métodos
16          void apilar(Pedido p);
17          void desapilar();
18          void mostrarCima();
19          void apilarPrioridad(Pedido p);
20          int comprobarCimaRegistrado();
21          // Getters y Setters
22          bool esVacia();
23          Pedido getCimaPedido();
24  };
25
26  #endif // PILAPEDIDOS_H
```

b) PilaPedidos.cpp

```
1  #include "PilaPedidos.h"
2
3  using namespace std;
4
5  // CONSTRUCTORES Y DESTRUCTORES //
6
7  PilaPedidos::PilaPedidos()
8  {
9      cima = NULL;
10 }
11
12 PilaPedidos::~PilaPedidos()
13 {
14 }
15
```



```

17 // MÉTODOS //
18
19 void PilaPedidos::apilar(Pedido p)
20 {
21     NodoPila* nuevo = new NodoPila(p, cima);
22     cima = nuevo; //la cima es el nuevo pedido
23 }
24
25 void PilaPedidos::desapilar()
26 {
27     NodoPila* aux;
28     if (cima) {
29         aux = cima;
30         cima = aux->siguiente_np;
31         delete aux;
32     } else {
33         cout << "La pila esta vacia" << endl;
34     }
35 }
36
37 void PilaPedidos::mostrarCima()
38 {
39     if (esVacia()) {
40         cout << "La pila esta vacia" << endl;
41     } else {
42         cout << "Cima pila: " << cima->pedido_np.imprimir() << endl;
43     }
44 }

```

```

46 void PilaPedidos::apilarPrioridad(Pedido p) // hay que comprobar si es erronea y si lo es pasarla por la funcion
47 {
48     PilaPedidos pilaAux;
49     if (esVacia()) {
50         apilar(p);
51     } else {
52         if (p.getTipoCliente() == "VIP" || p.getTipoCliente() == "NVIP") {
53             apilar(p);
54         } else { // para cuando son NR
55             while (!esVacia() && getCimaPedido().getTipoCliente() != "NR") {
56                 pilaAux.apilar(getCimaPedido());
57                 desapilar();
58             }
59             apilar(p); // anilamos en esa posicion el nuevo pedido que entra
60             while (!pilaAux.esVacia()) { // recolocamos de nuevo todos los pedidos de Registrados en la pila normal
61                 apilar(pilaAux.getCimaPedido());
62                 pilaAux.desapilar();
63             }
64         }
65     }
66 }
67
68 int PilaPedidos::comprobarCimaRegistrado()
69 {
70     int i;
71     if (esVacia()) {
72         return i = 0;
73     } else {
74         if (getCimaPedido().getTipoCliente() == "VIP" || getCimaPedido().getTipoCliente() == "NVIP") {
75             return i = 1;
76         } else {
77             return i = 2;
78         }
79     }
80 }

```

```

82 // GETTERS Y SETTERS //
83
84 Pedido PilaPedidos::getCimaPedido()
85 {
86     return cima->pedido_np;
87 }
88
89 bool PilaPedidos::esVacia()
90 {
91     if (cima) {
92         return false;
93     } else {
94         return true;
95     }
96 }

```

- NodoPila

a) NodoPila.h

```

1  #ifndef NODOPILA_H
2  #define NODOPILA_H
3  #include "Pedido.h"
4
5  class NodoPila
6  {
7      friend class PilaPedidos;
8
9      private:
10         Pedido pedido_np;
11         NodoPila* siguiente_np;
12
13     public:
14         NodoPila(Pedido p, NodoPila* sig);
15         ~NodoPila();
16 };
17
18 #endif // NODOPILA_H

```

b) NodoPila.cpp

```

1  #include "NodoPila.h"
2
3  // CONSTRUCTORES Y DESTRUCTORES //
4
5  NodoPila::NodoPila(Pedido p, NodoPila* sig)
6  {
7      pedido_np = p;
8      siguiente_np = sig;
9  }
10
11  NodoPila::~~NodoPila()
12  {
13  }
14

```

- ColaPedidos

a) ColaPedidos.h

```
1  #ifndef COLAPEDIDOS_H
2  #define COLAPEDIDOS_H
3  #include "NodoCola.h"
4  #include "Pedido.h"
5
6  class ColaPedidos
7  {
8      private:
9          NodoCola* primero;
10         NodoCola* ultimo;
11
12     public:
13         // Constructores y Destructores
14         ColaPedidos();
15         ~ColaPedidos();
16         // Métodos
17         void encolar(Pedido p);
18         void desencolar();
19         void mostrarCola();
20         // Getters y Setters
21         Pedido getPrimerPedido();
22         NodoCola* getPrimero();
23         NodoCola* getUltimo();
24         bool esVacia();
25
26     };
27
28 #endif // COLAPEDIDOS_H
```

b) ColaPedidos.cpp

```
1  #include <iostream>
2  #include "ColaPedidos.h"
3
4  using namespace std;
5
6  // CONTRUCTORES Y DESTRUCTORES //
7
8  ColaPedidos::ColaPedidos()
9  {
10     primero = NULL;
11     ultimo = NULL;
12 }
13
14 ColaPedidos::~ColaPedidos()
15 {
16 }
17
```



```

19 // MÉTODOS //
20
21 void ColaPedidos::encolar(Pedido p)
22 {
23     NodoCola* aux = new NodoCola(p, NULL);
24     if (primero) {
25         ultimo->siguiente_nc = aux;
26     } else {
27         primero = aux;
28     }
29     ultimo = aux;
30 }
31
32 void ColaPedidos::desencolar()
33 {
34     NodoCola* aux; //guardamos primero
35     if (primero != NULL) { //podemos desencolar porque la cola no esta vacia
36         aux = primero;
37         primero = aux->siguiente_nc; // el nuevo primero es el siguiente al que desencolo
38         delete aux;
39     } else {
40         cout << "La cola esta vacia" << endl;
41     }
42 }
43
44 void ColaPedidos::mostrarCola()
45 {
46     NodoCola* aux;
47     aux = primero;
48     if (esVacia()) {
49         cout << "La cola esta vacia" << endl;
50     } else {
51         cout << "Pedidos dentro de la cola: " << endl;
52         while (aux->siguiente_nc != NULL) {
53             cout << aux->pedido_nc.imprimir() << endl;
54             aux = aux->siguiente_nc;
55         }
56     }
57 }

```

```

59 // GETTERS Y SETTERS //
60
61 Pedido ColaPedidos::getPrimerPedido()
62 {
63     return primero->pedido_nc;
64 }
65
66 NodoCola* ColaPedidos::getPrimero()
67 {
68     return primero;
69 }
70
71 NodoCola* ColaPedidos::getUltimo()
72 {
73     return ultimo;
74 }
75
76 bool ColaPedidos::esVacia()
77 {
78     if (primero) {
79         return false;
80     } else {
81         return true;
82     }
83 }
84

```

- **NodoCola**

a) **NodoCola.h**

```
1  #ifndef NODOCOLA_H
2  #define NODOCOLA_H
3  #include "Pedido.h"
4
5  class NodoCola
6  {
7      friend class ColaPedidos;
8
9      private:
10         NodoCola* siguiente_nc;
11         Pedido pedido_nc;
12
13     public:
14         NodoCola(Pedido p, NodoCola *sig);
15         ~NodoCola();
16     };
17
18 #endif // NODOCOLA_H
```

b) **NodoCola.cpp**

```
1  #include "NodoCola.h"
2
3  //CONSTRUCTORES Y DESTRUCTORES //
4
5  NodoCola::NodoCola(Pedido p, NodoCola* sig)
6  {
7      pedido_nc = p;
8      siguiente_nc = sig;
9  }
10
11  NodoCola::~~NodoCola()
12  {
13  }
14
```

- **ListaPedidos**

a) ListaPedidos.h

```
1  #ifndef LISTAPEDIDOS_H
2  #define LISTAPEDIDOS_H
3  #include "NodoLista.h"
4
5  class ListaPedidos
6  {
7      private:
8          NodoLista* primero;
9          NodoLista* ultimo;
10     public:
11         // Constructores y destructores
12         ListaPedidos();
13         ~ListaPedidos();
14         // Métodos
15         void enlistar(Pedido p);
16         void insertarDerecha(Pedido p);
17         void insertarIzquierda(Pedido p);
18         void desenlistar();
19         void mostrarLista();
20         // Getters y Setters
21         bool esVacia();
22         Pedido getPrimerPedido();
23         Pedido getUltimoPedido();
24     };
25
26 #endif // LISTAPEDIDOS_H
```

b) ListaPedidos.cpp

```
1  #include "ListaPedidos.h"
2
3  // CONSTRUCTORES Y DESTRUCTORES //
4
5  ListaPedidos::ListaPedidos()
6  {
7      primero = NULL;
8      ultimo = NULL;
9  }
10
11  ListaPedidos::~ListaPedidos()
12  {
13  }
14
```

```

16 // MÉTODOS //
17
18 void ListaPedidos::enlistar(Pedido p)
19 {
20     insertarDerecha(p);
21     //insertarIzquierda(p);
22 }
23
24 void ListaPedidos::insertarDerecha(Pedido p)
25 {
26     NodoLista* nodo_nuevo;
27     if (!primero) { //lista vacia
28         nodo_nuevo = new NodoLista (p, NULL, NULL);
29         primero = nodo_nuevo;
30     } else {
31         nodo_nuevo = new NodoLista (p, ultimo, NULL);
32         ultimo->siguiente_nl = nodo_nuevo;
33     }
34     ultimo = nodo_nuevo; // el NUEVO ultimo es el nuevo que meto
35 }
36
37 void ListaPedidos::insertarIzquierda(Pedido p) //no lo utilizamos pero es una funcion propia de listas
38 {
39     NodoLista* nodo_nuevo;
40     if (!primero) { //lista vacia
41         nodo_nuevo = new NodoLista (p, NULL, NULL);
42     } else {
43         nodo_nuevo = new NodoLista (p, NULL, primero);
44     }
45     primero = nodo_nuevo; // el NUEVO primero es el nuevo que meto
46 }
47
48 void ListaPedidos::desenlistar() // Sacamos hacia el primer pedido de la lista
49 {
50     NodoLista* nodo_aux; //guardamos primero
51     if (primero != NULL) { //podemos desenlistar porque la lista no esta vacia
52         nodo_aux = primero;
53         primero = nodo_aux->siguiente_nl; // El nuevo primero es el siguiente al que desenlisto
54         delete nodo_aux; // borrarlo de memoria
55     } else {
56         cout << "La lista esta vacia" << endl;
57     }
58 }
59
60 void ListaPedidos::mostrarLista()
61 {
62     if (primero) {
63         NodoLista* nodo;
64         nodo = primero;
65         while(nodo != ultimo->siguiente_nl) {
66             cout << nodo->pedido_nl.imprimir() << endl;
67             nodo = nodo->siguiente_nl;
68         }
69     } else {
70         cout << "La lista esta vacia" << endl;
71     }
72 }

```

```

74 // GETTERS Y SETTERS //
75
76 bool ListaPedidos::esVacia()
77 {
78     if (primero) {
79         return false;
80     } else {
81         return true;
82     }
83 }
84
85 Pedido ListaPedidos::getPrimerPedido()
86 {
87     return primero->pedido_nl;
88 }
89
90 Pedido ListaPedidos::getUltimoPedido()
91 {
92     return ultimo->pedido_nl;
93 }

```

- NodoLista

a) NodoLista.h

```

1  #ifndef NODOLISTA_H
2  #define NODOLISTA_H
3  #include "Pedido.h"
4
5  class NodoLista
6  {
7      friend class ListaPedidos;
8
9      private:
10         Pedido pedido_nl;
11         NodoLista* anterior_nl;
12         NodoLista* siguiente_nl;
13     public:
14         NodoLista(Pedido p, NodoLista* ant, NodoLista* sig);
15         ~NodoLista();
16     };
17
18 #endif // NODOLISTA_H

```


b) NodoLista.cpp

```
1  #include "NodoLista.h"
2
3  // CONSTRUCTORES Y DESTRUCTORES //
4
5  NodoLista::NodoLista(Pedido p, NodoLista* ant, NodoLista* sig)
6  {
7      pedido_nl = p;
8      anterior_nl = ant;
9      siguiente_nl = sig;
10 }
11
12 NodoLista::~NodoLista()
13 {
14     //dtor
15 }
```

- Pedido

a) Pedido.h

```
1  #ifndef PEDIDO_H
2  #define PEDIDO_H
3  #include <string>
4  #include <iostream>
5
6  using namespace std;
7
8
9  class Pedido
10 {
11 private:
12     string descripcionArt, nombreCliente, direccionEntrega, tipoCliente;
13     long long numTarjeta;
14     int tiempoPrepEnvio;
15 public:
16     // Constructores y destructores
17     Pedido();
18     Pedido(string descripcion, string nombre, string direccion, string tipo, long long tarjeta, int tiempo);
19     ~Pedido();
20     // Métodos
21     bool comprobarPedidoErroneo();
22     string imprimir();
23     void completarPedido();
24     void reducirTiempoPrepEnvio();
25     // Getters y Setters
26     string getDescripcionArt();
27     string getNombreCliente();
28     string getDireccionEntrega();
29     string getTipoCliente();
30     long long getNumTarjeta();
31     int getTiempoPrepEnvio();
32 };
33
34 #endif // PEDIDO_H
```

b) Pedido.cpp

```
1  #include "Pedido.h"
2  #include <stdlib.h>
3  #include <cstdlib>
4  #include <ctime>
5  #include <iostream>
6  #include <string>
7
8  using namespace std;
9
10 // CONSTRUCTORES Y DESTRUCTORES //
11
12 Pedido::Pedido()
13 {
14 }
15
16
17 Pedido::Pedido(string descripcion, string nombre, string direccion, string tipo, long long tarjeta, int tiempo)
18 {
19     descripcionArt = descripcion;
20     nombreCliente = nombre;
21     direccionEntrega = direccion;
22     tipoCliente = tipo;
23     numTarjeta = tarjeta;
24     tiempoPrepEnvio = tiempo;
25 }
26
27 Pedido::~Pedido()
28 {
29 }
30
31
32 // MÉTODOS //
33
34 bool Pedido::comprobarPedidoErroneo()
35 {
36     string i = to_string(numTarjeta);
37     int longTarjeta = i.length();
38     if (descripcionArt == "" || nombreCliente == "" || direccionEntrega == "" || longTarjeta != 12 || (10 < tiempoPrepEnvio || tiempoPrepEnvio < 1)) {
39         return true;
40     } else {
41         return false;
42     }
43 }
44
45 string Pedido::imprimir()
46 {
47     string i = to_string(numTarjeta);
48     string j = to_string(tiempoPrepEnvio);
49     string pedido = descripcionArt + "/" + nombreCliente + "/" + direccionEntrega + "/" + tipoCliente + "/" + i + "/" + j;
50     return pedido;
51 }
```

```

53 void Pedido::completarPedido()
54 {
55     string i = to_string(numTarjeta);
56     int longTarjeta = i.length();
57     while (comprobarPedidoErroneo()) {
58         srand (time(NULL));
59         int random = 1 + rand() % 6;
60         if (descripcionArt == "") {
61             if (random == 1) {
62                 descripcionArt = "Xiaomi Mi Smart Band 5";
63             } else if (random == 2) {
64                 descripcionArt = "PlayStation 5";
65             } else if (random == 3) {
66                 descripcionArt = "GeForce RTX 3090";
67             } else if (random == 4) {
68                 descripcionArt = "i7-9700K 3.6GHz";
69             } else {
70                 descripcionArt = "Iphone 12";
71             }
72         } else if (nombreCliente == "") {
73             if (random == 1) {
74                 nombreCliente = "Jorge Arroyo Megia";
75             } else if (random == 2) {
76                 nombreCliente = "Isabel Blanco Martinez";
77             } else if (random == 3) {
78                 nombreCliente = "Maria Sanz Espeja";
79             } else if (random == 4) {
80                 nombreCliente = "Patricia Cuesta Ruiz";
81             } else {
82                 nombreCliente = "Lucia Campos Diaz";
83             }
84         } else if (direccionEntrega == "") {
85             if (random == 1) {
86                 direccionEntrega = "Calle Pesogordo, 2";
87             } else if (random == 2) {
88                 direccionEntrega = "Calle Mayor, 1";
89             } else if (random == 3) {
90                 direccionEntrega = "Plaza Santo Domingo, 4";
91             } else if (random == 4) {
92                 direccionEntrega = "Avenida del Ejercito, 23";
93             } else {
94                 direccionEntrega = "Calle Bardales, 7";
95             }
96         } else if (longTarjeta != 12) {
97             if (random == 1) {
98                 numTarjeta = 654236874523;
99             } else if (random == 2) {
100                 numTarjeta = 514300453469;
101             } else if (random == 3) {
102                 numTarjeta = 937933494195;
103             } else if (random == 4) {
104                 numTarjeta = 548744023638;
105             } else {
106                 numTarjeta = 615271819368;
107             }
108         } else if (10 < tiempoPrepEnvio || tiempoPrepEnvio < 1) {
109             int tiempoRandom = 1 + rand() % 11;
110             tiempoPrepEnvio = tiempoRandom;
111         }
112     }
113 }

```



```
115     void Pedido::reducirTiempoPrepEnvio()
116     {
117         tiempoPrepEnvio--;
118     }
119
120     // GETTERS Y SETTERS
121     string Pedido::getDescripcionArt()
122     {
123         return descripcionArt;
124     }
125
126     string Pedido::getNombreCliente()
127     {
128         return nombreCliente;
129     }
130
131     string Pedido::getDireccionEntrega()
132     {
133         return direccionEntrega;
134     }
135
136     string Pedido::getTipoCliente()
137     {
138         return tipoCliente;
139     }
140
141     long long Pedido::getNumTarjeta()
142     {
143         return numTarjeta;
144     }
145
146     int Pedido::getTiempoPrepEnvio()
147     {
148         return tiempoPrepEnvio;
149     }
```

- Correos

a) Correos.h

```
1  #ifndef CORREOS_H
2  #define CORREOS_H
3  #include "Pedido.h"
4  #include "ColaPedidos.h"
5  #include "PilaPedidos.h"
6  #include "ListaPedidos.h"
7
8  class Correos
9  {
10 public:
11     Correos();
12     ~Correos();
13     void gestionCorreos(ColaPedidos colaRegistrados, ColaPedidos colaNoRegistrados);
14 };
15
16 #endif // CORREOS_H
```

b) Correos.cpp

```
1  #include "Correos.h"
2  #include <iostream>
3
4  using namespace std;
5
6  // CONSTRUCTORES Y DESTRUCTORES //
7
8  Correos::Correos()
9  {
10 }
11
12
13 Correos::~Correos()
14 {
15 }
16
```

```

20 void Correos::gestionCorreos(ColaPedidos colaRegistrados, ColaPedidos colaNoRegistrados)
21 {
22     ListaPedidos listaPrepEnviar;
23     PilaPedidos pilaPedidosErroneos;
24     Pedido pedido, pedidoCR, pedidoCNR, pedidoPE;
25     // Metemos los primeros elementos de la lista de pedidos para enviar
26     if (!colaRegistrados.esVacia()) {
27         int contR = 0;
28         while (contR != 3) {
29             if (!colaRegistrados.esVacia()) {
30                 pedido = colaRegistrados.getPrimerPedido();
31                 colaRegistrados.desencolar();
32                 if (pedido.comprobarPedidoErroneo()) {
33                     pilaPedidosErroneos.apilarPrioridad(pedido);
34                     cout << "Se apila en pila de pedidos erroneos: " << pedido.imprimir() << endl;
35                 } else {
36                     listaPrepEnviar.enlistar(pedido);
37                     cout << "El pedido registrado introducido en la lista es: " << pedido.imprimir() << endl;
38                     contR++;
39                 }
40             } else {
41                 contR = 3;
42             }
43         }
44     }
45
46     if (!colaNoRegistrados.esVacia()) {
47         int contNR = 0;
48         while (contNR != 1) {
49             if (!colaNoRegistrados.esVacia()) {
50                 pedido = colaNoRegistrados.getPrimerPedido();
51                 colaNoRegistrados.desencolar();
52                 if (pedido.comprobarPedidoErroneo()) {
53                     pilaPedidosErroneos.apilarPrioridad(pedido);
54                     cout << "Se apila en pila de pedidos erroneos: " << pedido.imprimir() << endl;
55                 } else {
56                     listaPrepEnviar.enlistar(pedido);
57                     cout << "El pedido no registrado introducido en la lista es: " << pedido.imprimir() << endl;
58                     contNR++;
59                 }
60             } else {
61                 contNR = 1;
62             }
63         }
64     }
65
66     int tiempoTotal = 0;
67     while (!listaPrepEnviar.esVacia() || !colaRegistrados.esVacia() || !colaNoRegistrados.esVacia() || !pilaPedidosErroneos.esVacia()) {
68         // Procesamos el primer pedido de la lista
69         if (!listaPrepEnviar.esVacia()) {
70             cout << " " << endl;
71             pedido = listaPrepEnviar.getPrimerPedido();
72             cout << "El pedido " << pedido.imprimir() << " entra en preparacion" << endl;
73             while (pedido.getTiempoPrepEnvio() != 0) {
74                 cout << "Al pedido " << pedido.imprimir() << " le faltan " << pedido.getTiempoPrepEnvio() << " minutos" << endl;
75                 pedido.reducirTiempoPrepEnvio();
76                 tiempoTotal++;
77             }
78             cout << "El pedido " << pedido.imprimir() << " esta listo para ser enviado" << endl;
79             cout << " " << endl;
80             listaPrepEnviar.desenlistar(); // Enviamos el primer pedido de la lista
81         }
82         // Introducimos tres pedidos nuevos de la cola de registrados
83         if (!colaRegistrados.esVacia()) {
84             int contCR = 0;
85             while (contCR != 3) {
86                 if (!colaRegistrados.esVacia()) {
87                     pedidoCR = colaRegistrados.getPrimerPedido();
88                     colaRegistrados.desencolar();
89                     if (pedidoCR.comprobarPedidoErroneo()) {
90                         pilaPedidosErroneos.apilarPrioridad(pedidoCR);
91                         cout << "Se apila en pila de pedidos erroneos: " << pedidoCR.imprimir() << endl;
92                     } else {
93                         listaPrepEnviar.enlistar(pedidoCR);
94                         cout << "El pedido registrado introducido en la lista es: " << pedidoCR.imprimir() << endl;
95                         contCR++;
96                     }
97                 } else {
98                     contCR = 3;
99                 }
100             }
101         }
102     }

```

```

100 // Introducimos un pedido de la cola de no registrados o de la pila de pedidos erroneos
101 if (!pilaPedidosErroneos.esVacia() || !colaNoRegistrados.esVacia()) {
102     if (pilaPedidosErroneos.comprobarCimaRegistrado() == 1) {
103         pedidoPE = pilaPedidosErroneos.getCimaPedido();
104         pilaPedidosErroneos.desapilar();
105         pedidoPE.completarPedido();
106         cout << "Se corrige el pedido " << pedidoPE.imprimir() << endl;
107         listaPrepEnviar.enlistar(pedidoPE);
108         cout << "El pedido registrado introducido en la lista es: " << pedidoPE.imprimir() << endl;
109     } else if (!colaNoRegistrados.esVacia()) {
110         int contCNR = 0;
111         while (contCNR != 1) {
112             if (!colaNoRegistrados.esVacia()) {
113                 pedidoCNR = colaNoRegistrados.getPrimerPedido();
114                 colaNoRegistrados.desencolar();
115                 if (pedidoCNR.comprobarPedidoErroneo()) {
116                     pilaPedidosErroneos.apilarPrioridad(pedidoCNR);
117                     cout << "Se apila en pila de pedidos erroneos: " << pedidoCNR.imprimir() << endl;
118                 } else {
119                     listaPrepEnviar.enlistar(pedidoCNR);
120                     cout << "El pedido no registrado introducido en la lista es: " << pedidoCNR.imprimir() << endl;
121                     contCNR++;
122                 }
123             } else {
124                 contCNR = 1;
125             }
126         }
127     } else {
128         pedidoPE = pilaPedidosErroneos.getCimaPedido();
129         pilaPedidosErroneos.desapilar();
130         pedidoPE.completarPedido();
131         cout << "Se corrige el pedido " << pedidoPE.imprimir() << endl;
132         listaPrepEnviar.enlistar(pedidoPE);
133         cout << "El pedido no registrado introducido en la lista es: " << pedidoPE.imprimir() << endl;
134     }
135 }
136
137 }
138 cout << "El tiempo total es: " << tiempoTotal << " minutos" << endl;
139 }

```

5. BIBLIOGRAFÍA

- ❑ Tema 2: Pilas - Asignatura Estructuras de Datos, Universidad de Alcalá - M^a José Domínguez Alda
- ❑ Tema 3: Colas - Asignatura Estructuras de Datos, Universidad de Alcalá - M^a José Domínguez Alda
- ❑ Tema 4: Listas - Asignatura Estructuras de Datos, Universidad de Alcalá - M^a José Domínguez Alda
- ❑ Tema 5: Árboles Binarios y de Búsqueda - Asignatura Estructuras de Datos, Universidad de Alcalá - M^a José Domínguez Alda
- ❑ Introducción C/C++ (Powerpoint) - Asignatura Estructuras de Datos, Universidad de Alcalá - Hamid Tayebi