Grado en Ingeniería Informática.
Curso 2019/2020

Control de Aduanas en un Aeropuerto, **septiembre 2020**

Víctor Gamonal Sánchez
Laura Mambrilla Moreno

Universidad de Alcalá

Índice

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, lista y árbol. 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 Pila, Cola, Lista y Árbol, nos encontramos con las clases Pasajero, Box y Aeropuerto.

1.1. Pila

Hemos hecho uso de la pila para insertar en ella a los distintos pasajeros provenientes del árbol, ya ordenados por inorden, donde el pasajero que entrará el último en el aeropuerto (con menor ID y mayor hora de llegada) será apilado al fondo de ésta (estructura LIFO). Posteriormente, serán desapilados uno a uno cuando la hora de llegada de cada uno de ellos coincida con el tiempo transcurrido en el funcionamiento del programa.

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

- ***Operaciones***
    - void apilar(Pasajero p) - Función encargada de apilar (insertar dentro de la pila) el pasajero que recibe por parámetro.
    - void desapilar() - Función encargada de desapilar (sacar fuera de la pila) al pasajero actualmente en la cima.
    - void imprimir() - Procedimiento encargado de imprimir por pantalla la pila de pasajeros.
    - Pasajero getCimaPasajero() - Getter para obtener únicamente al pasajero de la cima.
    - bool isVacia() - Función que devuelve un booleano en función de si la pila está o no vacía, lo cual será de mucha utilidad para comprobar que no queden pasajeros por llegar al aeropuerto.
    - Pila (); // ~Pila(); - Constructor y deconstructor

Pila.hpp

```hpp
#ifndef PILA_HPP
#define PILA_HPP

#include "NodoPila.hpp"
#include "Pasajero.hpp"

class Pila
{
public:
    Pila();
    ~Pila();

    //metodos y funciones
    void apilar(Pasajero p);
    void desapilar();
    void imprimir();

    //getter y setter
    NodoPila* getCima();
    Pasajero getCimaPasajero();
    bool isVacia();

private:
    NodoPila* cima;
};

#endif // PILA_HPP

```

1.2. NodoPila

Este nodo contendrá información sobre un pasajero dentro de la pila y un puntero al siguiente de éste.

- ***Atributos***
    - Pasajero pasajero_np
    - NodoPila* siguiente_np

- ***Operaciones***
    - NodoPila(Pasajero p, NodoPila* sig); - Constructor con los parámetros necesarios, en este caso un pasajero y un puntero al siguiente, que serán inicializados a NULL.

NodoPila.hpp

```
#ifndef NODOPILA_HPP
#define NODOPILA_HPP

#include "Pasajero.hpp"

#include <iostream>
using namespace std;

class Pasajero;

class NodoPila
{
public:
    NodoPila(Pasajero p, NodoPila* sig);
    ~NodoPila();

private:
    Pasajero pasajero_np;
    NodoPila* siguiente_np;
    friend class Pila;
};

#endif // NODOPILA_HPP
```

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 *ultimo*, ambos de tipo puntero *NodoCola*, ya que son nodos y actúan como tales.

- ***Atributos***
  - NodoCola* primero
  - NodoCola* ultimo

- ***Operaciones***
  - void encolar (Pasajero p)
  - void desencolar() - Ambos para controlar el tránsito de los pasajeros dentro de la cola.
  - void imprimir() - Procedimiento que muestra por pantalla el estado actual de la cola

- Pasajero getPrimeroPasajero() - Getter que recoge únicamente la información del pasajero 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 ultimo de la cola.
- bool isVacia - Función que retorna un booleano en función de si la cola está o no vacía.
- Cola(); // ~Cola(); - Constructor y deconstructor

Cola.hpp

```hpp
#ifndef COLA_HPP
#define COLA_HPP

#include "NodoCola.hpp"
#include "Pasajero.hpp"

class NodoCola;

class Cola
{
public:
    Cola();
    ~Cola();
    void encolar(Pasajero p);
    void desencolar(); //solo para cola_inicial (para insertar en árbol)
    void imprimir();

    //getters y setters
    Pasajero getPrimeroPasajero();
    NodoCola* getPrimero();
    NodoCola* getUltimo();
    bool isVacia();

private:
    NodoCola* primero;
    NodoCola* ultimo;

};

#endif // COLA_HPP

```

1.4. NodoCola

Este nodo contendrá información sobre un pasajero dentro de la cola y un puntero al siguiente.

- ***Atributos***
  - Pasajero pasajero_nc

- 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.

- ***Operaciones***
  - NodoCola (Pasajero p, NodoCola* sig) - Constructor con los parámetros necesarios, en este caso un pasajero y un puntero al siguiente, que serán inicializados a NULL.
  - NodoCola(); // ~NodoCola(); - Constructor sin parámetros y deconstructor
  - NodoCola()* getSiguiente(); - Getter que recoge únicamente la información del siguiente nodo del nodo actual. Esta función es necesaria ya que en la clase main debemos calcular el *tiempo_espera_total* sumando el tiempo de espera de los pasajeros recorriendo la cola desde esa clase. Esta última explicación corresponde también a la siguiente función *getPasajeroNC()*.
  - Pasajero getPasajeroNC(); - Getter que recoge únicamente la información del pasajero del nodo actual.

NodoCola.hpp

```
#ifndef NODOCOLA_HPP
#define NODOCOLA_HPP

#include "Pasajero.hpp"

class NodoCola
{
public:
    NodoCola();
    NodoCola(Pasajero p, NodoCola* sig);
    ~NodoCola();
    NodoCola* getSiguiente();
    Pasajero getPasajeroNC();

private:
    Pasajero pasajero_nc;
    NodoCola* siguiente_nc;

    friend class Cola;
};

#endif // NODOCOLA_HPP
```

1.5. Lista

Este TAD lo hemos utilizado a modo de cola de espera en la que los pasajeros que van llegando al aeropuerto esperan (o no, si el Box estuviese libre) en ella, siendo insertados todos provenientes de la pila y por la derecha ya que vienen previamente ordenados en función de la hora de llegada de cada uno de ellos, por lo que no es necesaria una comprobación dentro de ésta de si la hora de llegada de un pasajero nuevo es superior o inferior a la de los que ya residen en ella. Permanecerán en esta lista hasta que puedan ser atendidos en el Box por orden de llegada (desenlistar), almacenando el tiempo de espera de cada uno de ellos hasta poder entrar en él.

- ***Atributos***
  - NodoLista* primero - Un puntero de tipo NodoLista que apunta a la primera posición de la lista (al primer pasajero).
  - NodoLista* ultimo - Un puntero de tipo NodoLista que apunta a la última posición de la lista (al último pasajero).

- ***Operaciones***
  - void enlistar (Pasajero p) - Función encargada de insertar dentro de la lista al pasajero que se le pasa por parámetro.
  - void insertarDerecha (Pasajero p) - Función que recoge los procedimientos necesarios para insertar por la derecha de la lista al pasajero que se pasa por parámetro.
  - void insertarIzquierda (Pasajero p) - Función que recoge los procedimientos necesarios para insertar por la izquierda de la lista al pasajero que se pasa por parámetro (no utilizado en nuestro programa, pero lo hemos definido y comprobado que funciona).
  - void desenlistar() - Función encargada de sacar de la lista al pasajero que se encuentre el primero.
  - void imprimir() - Procedimiento encargado de imprimir por pantalla toda la información de la lista actual.
  - Pasajero getPrimeroPasajero() - Getter para obtener únicamente el pasajero en primera posición de la lista.
  - Pasajero getUltimoPasajero() - Getter para obtener únicamente el pasajero en última posición de la lista.
  - Lista(); // ~Lista(); - Constructor sin parámetros y deconstructor

Lista.hpp

```
#ifndef LISTA_HPP
#define LISTA_HPP

#include "NodoLista.hpp"

class Lista
{
public:
    Lista();
    ~Lista();
    void enlistar(Pasajero p);
    void insertarDerecha(Pasajero p);
    void insertarIzquierda(Pasajero p);
    void desenlistar();
    void imprimir();

    //getters y setters
    Pasajero getPrimeroPasajero();
    Pasajero getUltimoPasajero();
    bool isVacia();

private:
    NodoLista* primero;
    NodoLista* ultimo;

};

#endif // LISTA_HPP
```

1.6. NodoLista

Este nodo contendrá información sobre un pasajero dentro de la lista y un puntero al anterior y otro puntero al siguiente de éste.

- ***Atributos***
  - Pasajero pasajero_nl
  - NodoLista* anterior_nl
  - NodoLista* siguiente_nl

- ***Operaciones***
  - NodoLista(Pasajero p, NodoLista* ant, NodoLista* sig) - Constructor con los parámetros necesarios, en este caso un pasajero y un puntero al anterior y otro al siguiente, que serán inicializados a NULL.

NodoLista.hpp

```
#ifndef NODOLISTA_HPP
#define NODOLISTA_HPP

#include "Pasajero.hpp"

class NodoLista
{
public:
    NodoLista(Pasajero p, NodoLista* ant, NodoLista* sig);
    ~NodoLista();

private:
    Pasajero pasajero_nl;
    NodoLista* anterior_nl;
    NodoLista* siguiente_nl;

    friend class Lista;
};

#endif // NODOLISTA_HPP
```

1.7 Árbol

Los árboles binarios de búsqueda se caracterizan por el hecho de que su raíz tiene como máximo 2 nodos hijos: uno a la izquierda y otro a la derecha. El de la izquierda será el que tenga un valor (el valor por el que estamos colocando los datos) menor o igual al de su raíz, mientras que el de la derecha será uno con mayor valor. Así pues, nos encontramos con que la clase Arbol tiene tres atributos, todos ellos del tipo puntero *NodoArbol*: *raizz*, *izquierda* y *derecha*.

- ***Atributos***
    - NodoArbol* raizz
    - NodoArbol* izquierda
    - NodoArbol* derecha

- ***Operaciones***
    - void insertar (Pasajero, NodoArbol *&) - Función encargada de meter en el árbol al pasajero que se le mete por parámetro, y además pasando por puntero a referencia un nodo que irá variando en cada llamada a este método, ya que lo hacemos de forma recursiva buscando hacia la izq o hacia la dcha. del nodo raíz hasta que pueda insertar a ese pasajero.
    - Lista inorden (NodoArbol* nabb, Lista& lista_arbol) - Función encargada de almacenar en una lista a todos los pasajeros ordenados por inorden

(izquierda, centro, derecha) utilizando un *NodoArbol* y esa lista donde se almacenan pasada por referencia, ya que su contenido irá variando de forma constante.

- void imprimir() - Procedimiento que muestra por pantalla el estado actual del árbol.
- Arbol(); // ~Arbol(); - Constructor sin parámetros y deconstructor
- NodoArbol* getRaiz() - Getter que devuelve el puntero al nodo raíz del árbol
- int getRaizPasajero() - Getter que devuelve el *ID del pasajero* que está en la raíz.

Arbol.hpp

```
#define ARBOL_HPP

#include "NodoArbol.hpp"
#include "Lista.hpp"

#include <iostream>
#include <string.h>

using namespace std;

class Arbol
{
public:
    Arbol();
    ~Arbol();
    Lista inorden(NodoArbol* nabb, Lista& lista_arbol); //el primer parametro que tenemos que meter es la raiz del arbol
    void insertar(Pasajero, NodoArbol*&); //paso por referancia para que el arbol se modifique
    void imprimir(NodoArbol* nodo, int contador);

    //getters y setters
    NodoArbol* getRaiz();
    int getRaizPasajero();

private:
    NodoArbol* raizz;
    NodoArbol* izquierda;
    NodoArbol* derecha;

    friend class Lista;

};
#endif // ARBOL_HPP
```

1.8 NodoArbol

Cada nodo del árbol contiene un pasajero y dos punteros: un puntero que corresponde al nodo de su izquierda y otro al de su derecha, por lo que tiene como atributos esta clase:

- ***Atributos***
    - Pasajero pasajero_na
    - NodoArbol* izquierda_na
    - NodoArbol* derecha_na

- ***Operaciones***
    - NodoArbol (Pasajero p, NodoArbol* izqq, NodoArbol* dcha) - Constructor con los parámetros necesarios, en este caso un pasajero y un puntero al nodo de la izquierda y otro al de la derecha, que serán inicializados a NULL.

NodoArbol.hpp

```
#ifndef NODOARBOL_HPP
#define NODOARBOL_HPP

#include "Pasajero.hpp"

class NodoArbol
{
public:
    NodoArbol(Pasajero p, NodoArbol* izqq, NodoArbol* dcha);
    NodoArbol();
    ~NodoArbol();

private:
    Pasajero pasajero_na;
    NodoArbol* izquierda_na;
    NodoArbol* derecha_na;

    friend class Arbol;

};

#endif // NODOARBOL_HPP
```

2. Solución adoptada: descripción de las dificultades encontradas

Para estar seguros de que el funcionamiento del programa es correcto, hemos comprobado los tiempos de llegada al aeropuerto y los tiempos de espera manualmente.

La primera tabla (**PASAJEROS EN ORDEN DE LLEGADA**) recoge a los pasajeros con sus ids, horas de llegada y duraciones de atención en el box.

La segunda tabla (**CRONOLOGÍA**) muestra de manera visual el tránsito de los pasajeros por el aeropuerto y, por ende, por el box. Cada fila de la tabla representa un minuto (el valor de cada minuto se puede ver en la primera columna de la tabla). La segunda columna representa al box, por lo que podemos observar qué pasajero está dentro del box en cada minuto. La tercera columna muestra cuándo llega cada pasajero al aeropuerto. A partir de la línea negra más gruesa que las demás nos encontramos con varias columnas, pero todas ellas representan la lista de espera para el acceso al box.

PASAJEROS EN ORDEN DE LLEGADA

PASAJERO	ID	HORA LLEGADA	DURACIÓN
Pasajero 7	9	0	6
Pasajero 9	8	4	7
Pasajero 3	7	9	7
Pasajero 4	6	14	3
Pasajero 1	5	15	5
Pasajero 5	4	18	4
Pasajero 2	3	22	6
Pasajero 8	2	26	3
Pasajero 6	1	27	17

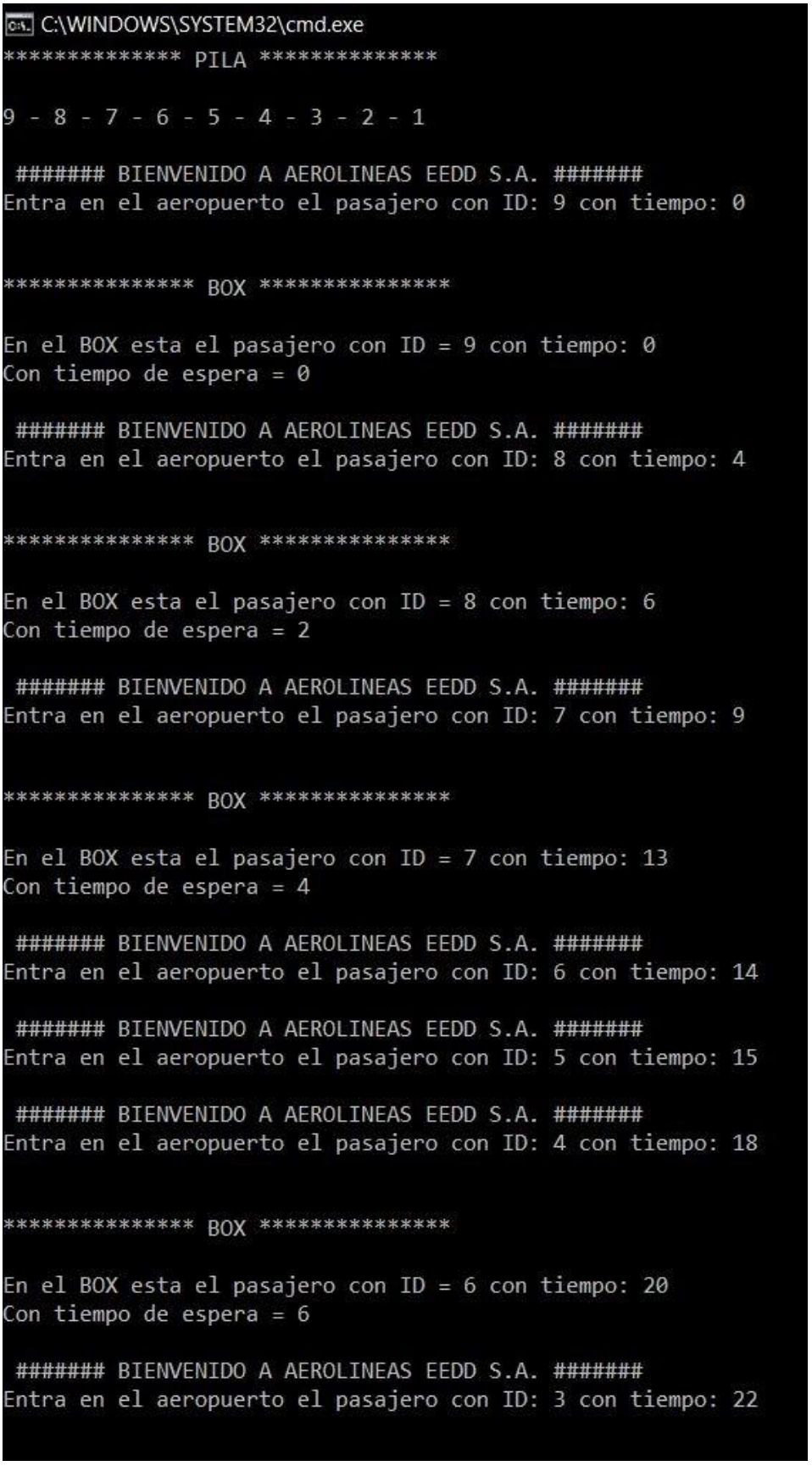
CRONOLOGÍA

MINUTOS	BOX	LLEGADA	ESPERANDO					
0	7(id-9)	7(id-9)	7(id-9) Espera: **0**					
1	7(id-9)							
2	7(id-9)							
3	7(id-9)							
4	7(id-9)	9(id-8)	9(id-8)					
5	7(id-9) - Total: **6**		9(id-8) - Espera: **2**					
6	9(id-8)							
7	9(id-8)							
8	9(id-8)							
9	9(id-8)	3(id-7)	3(id-7)					
10	9(id-8)		3(id-7)					
11	9(id-8)		3(id-7)					
12	9(id-8) - Total: **9**		3(id-7) - Espera: **4**					
13	3(id-7)							
14	3(id-7)	4(id-6)	4(id-6)					
15	3(id-7)	1(id-5)	4(id-6)	1(id-5)				
16	3(id-7)		4(id-6)	1(id-5)				
17	3(id-7)		4(id-6)	1(id-5)				
18	3(id-7)	5(id-4)	4(id-6)	1(id-5)	5(id-4)			
19	3(id-7) - Total: **11**		4(id-6) - Espera:**6**	1(id-5)	5(id-4)			

20	4(id-6)			1(id-5)	5(id-4)			
21	4(id-6)			1(id-5)	5(id-4)			
22	4(id-6) - Total: **9**	2(id-3)		1(id-5) - Espera: **8**	5(id-4)	2(id-3)		
23	1(id-5)				5(id-4)	2(id-3)		
24	1(id-5)				5(id-4)	2(id-3)		
25	1(id-5)				5(id-4)	2(id-3)		
26	1(id-5)	8(id-2)			5(id-4)	2(id-3)	8(id-2)	
27	1(id-5) - Total: **13**	6(id-1)			5(id-4) - Espera: **10**	2(id-3)	8(id-2)	6(id-1)
28	5(id-4)					2(id-3)	8(id-2)	6(id-1)
29	5(id-4)					2(id-3)	8(id-2)	6(id-1)
30	5(id-4)					2(id-3)	8(id-2)	6(id-1)
31	5(id-4) - Total: **14**					2(id-3) - Espera: **10**	8(id-2)	6(id-1)
32	2(id-3)						8(id-2)	6(id-1)
33	2(id-3)						8(id-2)	6(id-1)
34	2(id-3)						8(id-2)	6(id-1)
35	2(id-3)						8(id-2)	6(id-1)
36	2(id-3)						8(id-2)	6(id-1)
37	2(id-3) - Total: **16**						8(id-2) - Espera: **12**	6(id-1)
38	8(id-2)							6(id-1)
39	8(id-2)							6(id-1)
40	8(id-2) - Total: **15**							6(id-1) - Espera: **14**
41	6(id-1)							

42	6(id-1)							
43	6(id-1)							
44	6(id-1)							
45	6(id-1)							
46	6(id-1)							
47	6(id-1)							
48	6(id-1)							
49	6(id-1)							
50	6(id-1)							
51	6(id-1)							
52	6(id-1)							
53	6(id-1)							
54	6(id-1)							
55	6(id-1)							
56	6(id-1)							
57	6(id-1) - Total: **31**							

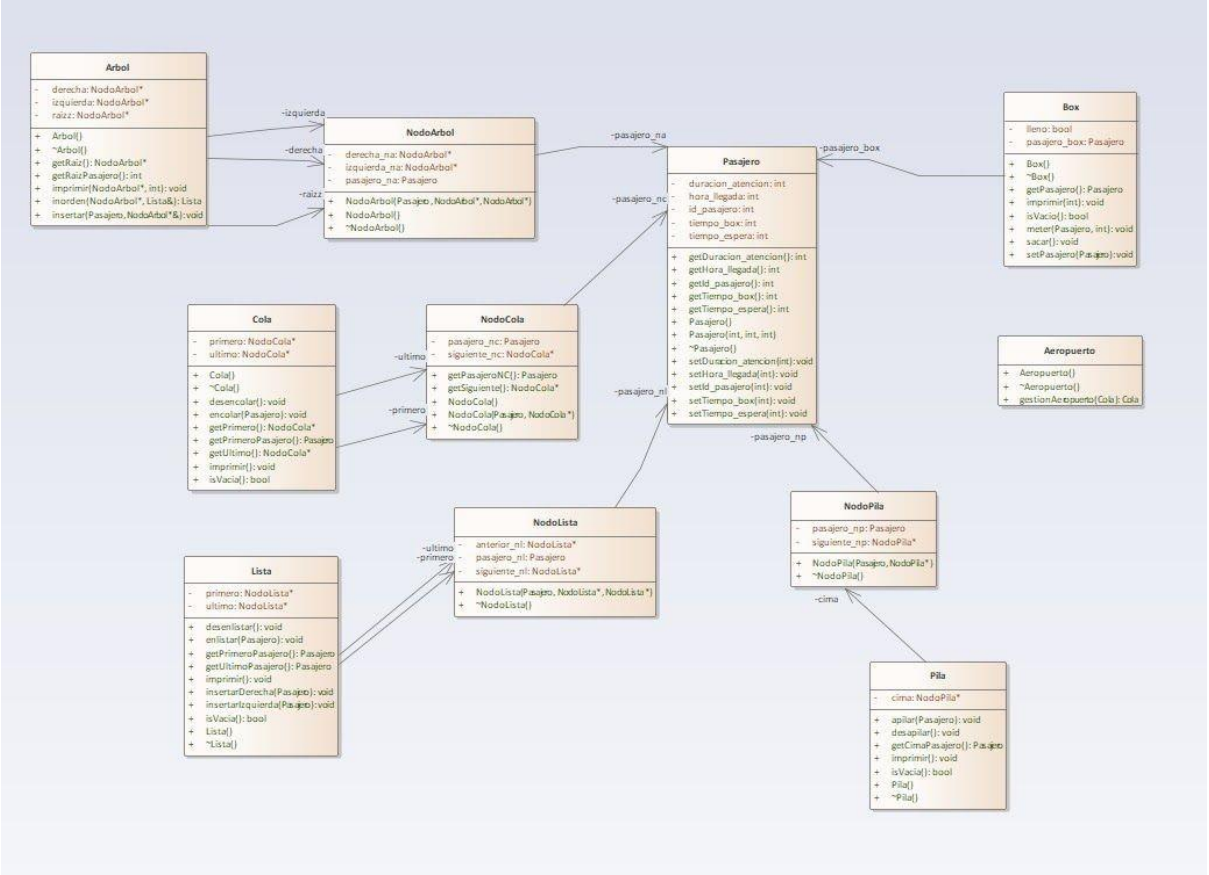
Se puede comprobar que los resultados obtenidos en la tabla son los mismos resultados que se generan tras la ejecución del programa:


C:\WINDOWS\SYSTEM32\cmd.exe
********** PILA **********
9 - 8 - 7 - 6 - 5 - 4 - 3 - 2 - 1
####### BIENVENIDO A AEROLINEAS EEDD S.A. #######
Entra en el aeropuerto el pasajero con ID: 9 con tiempo: 0
********** BOX **********
En el BOX esta el pasajero con ID = 9 con tiempo: 0
Con tiempo de espera = 0
####### BIENVENIDO A AEROLINEAS EEDD S.A. #######
Entra en el aeropuerto el pasajero con ID: 8 con tiempo: 4
********** BOX **********
En el BOX esta el pasajero con ID = 8 con tiempo: 6
Con tiempo de espera = 2
####### BIENVENIDO A AEROLINEAS EEDD S.A. #######
Entra en el aeropuerto el pasajero con ID: 7 con tiempo: 9
********** BOX **********
En el BOX esta el pasajero con ID = 7 con tiempo: 13
Con tiempo de espera = 4
####### BIENVENIDO A AEROLINEAS EEDD S.A. #######
Entra en el aeropuerto el pasajero con ID: 6 con tiempo: 14
####### BIENVENIDO A AEROLINEAS EEDD S.A. #######
Entra en el aeropuerto el pasajero con ID: 5 con tiempo: 15
####### BIENVENIDO A AEROLINEAS EEDD S.A. #######
Entra en el aeropuerto el pasajero con ID: 4 con tiempo: 18
********** BOX **********
En el BOX esta el pasajero con ID = 6 con tiempo: 20
Con tiempo de espera = 6
####### BIENVENIDO A AEROLINEAS EEDD S.A. #######
Entra en el aeropuerto el pasajero con ID: 3 con tiempo: 22

```
********** BOX **********

En el BOX esta el pasajero con ID = 5 con tiempo: 23
Con tiempo de espera = 8

 ####### BIENVENIDO A AEROLINEAS EEDD S.A. #######
Entra en el aeropuerto el pasajero con ID: 2 con tiempo: 26

 ####### BIENVENIDO A AEROLINEAS EEDD S.A. #######
Entra en el aeropuerto el pasajero con ID: 1 con tiempo: 27


********** BOX **********

En el BOX esta el pasajero con ID = 4 con tiempo: 28
Con tiempo de espera = 10


********** BOX **********

En el BOX esta el pasajero con ID = 3 con tiempo: 32
Con tiempo de espera = 10


********** BOX **********

En el BOX esta el pasajero con ID = 2 con tiempo: 38
Con tiempo de espera = 12


********** BOX **********

En el BOX esta el pasajero con ID = 1 con tiempo: 41
Con tiempo de espera = 14

--------------------------------------------------

        GRACIAS POR VIAJAR CON NOSOTROS

--------------------------------------------------

********** COLA FINAL **********

9 - 8 - 7 - 6 - 5 - 4 - 3 - 2 - 1

--------------------------------------------------

Tiempo media de espera: 7.33333

--------------------------------------------------
```

3. Diseño de la relación entre las clases de los TAD's

3.1. Diagrama UML



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

3.2. Explicación de los métodos más destacados

Pila - Método *APILAR*

```
14  void Pila::apilar(Pasajero p)
15  {
16      NodoPila* nodo_nuevo = new NodoPila (p, cima);
17      cima = nodo_nuevo;      // la nueva cima es el nuevo pasajero
18  }
```

Consiste en crear un nuevo NodoPila, guardando como pasajero el que se le pasa como parámetro y el puntero al siguiente será el pasajero almacenado en la cima, ya que éste quedará por debajo del nuevo y la nueva cima será el nuevo pasajero.

Pila - Método *DESAPILAR*

```
20  void Pila::desapilar()
21  {
22      NodoPila* nodo_aux; //guardamos el nodo siguiente de cima
23
24      if (cima)      // si pila no vacía
25      {
26          nodo_aux = cima;
27          cima = nodo_aux->siguiente_np; // el nuevo primero es el siguiente al que desapilo ahora
28          delete nodo_aux;               // borrarlo de memoria
29      }
30      else
31      {
32          cout << "Pila Vacía" << endl;
33      }
34  }
```

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, todo esto siempre y cuando la pila 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.

Cola - Método *ENCOLAR*

```
13  void Cola::encolar(Pasajero p)
14  {
15      NodoCola* nodo_nuevo = new NodoCola(p, NULL);
16
17      if (primero)
18      {
19          ultimo->siguiente_nc = nodo_nuevo;
20      }
21      else
22      {
23          primero = nodo_nuevo;
24      }
25
26      ultimo = nodo_nuevo;
27  }
```

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

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

Cola - Método *DESENCOLAR*

```
29  void Cola::desencolar()
30  {
31      NodoCola* nodo_aux;          //guardamos primero
32
33      if (primero != NULL)         //podemos desencolar porque la cola no esta vacia
34      {
35          nodo_aux = primero;
36          primero = nodo_aux->siguiente_nc; // El nuevo primero es el siguiente al que desencolo
37          delete nodo_aux;                  // borrarlo de memoria
38      }
39      else
40      {
41          cout << "Cola Vacia" << endl;
42      }
43  }
```

Siempre y cuando la cola no esté vacía, lo que haremos será almacenar en un nodo auxiliar al primer pasajero de la cola (el cual vamos a desencolar), y establecemos que el nuevo *primero* será el siguiente al actual.

Lista - Métodos *ENLISTAR, INSERTARDERECHA, INSERTARIZQUIERDA*

```
13  void Lista::enlistar(Pasajero p)
14  {
15      insertarDerecha(p);
16      //insertarIzquierda(p);
17  }
18
19  void Lista::insertarDerecha(Pasajero p)
20  {
21      NodoLista* nodo_nuevo;
22      if (!primero) //lista vacia
23      {
24          nodo_nuevo = new NodoLista (p, NULL, NULL);
25          primero = nodo_nuevo;
26      }
27      else
28      {
29          nodo_nuevo = new NodoLista (p, ultimo, NULL);
30          ultimo->siguiente_nl = nodo_nuevo;
31      }
32      ultimo = nodo_nuevo;  // el NUEVO ultimo es el nuevo que meto
33  }
34
35  void Lista::insertarIzquierda(Pasajero p)   //no lo utilizamos pero es una funcion propia de listas
36  {
37      NodoLista* nodo_nuevo;
38      if (!primero) //lista vacia
39      {
40          nodo_nuevo = new NodoLista (p, NULL, NULL);
41      }
42      else
43      {
44          nodo_nuevo = new NodoLista (p, NULL, primero);
45      }
46      primero = nodo_nuevo;  // el NUEVO primero es el nuevo que meto
47  }
```

Las listas tienen una peculiaridad, y es que podemos insertar elementos en ellas bien sea por la izquierda o por la derecha de la lista.

- En el caso de *InsertarDerecha*, tenemos dos opciones. En el caso de que la lista esté vacía, lo que haremos será crear un *NodoLista* nuevo, cuyo pasajero 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 *ultimo* de la lista. En el caso de que la lista no esté vacía, estableceremos también ese nuevo nodo pero indicándole esta vez que el puntero al anterior será el actual último elemento de la lista, además de que éste también será el nuevo último.

- En el caso de *InsertarIzquierda*, lo único que varía es que jugaremos con el factor de que el puntero al siguiente será el actual primer elemento de la lista y actualizaremos ese *primero* al nuevo que estamos insertando.

- El método *enlistar* simplemente hace una llamada a la función *InsertarDerecha*.

Lista - Método *DESENLISTAR*

```
49  void Lista::desenlistar() // Sacamos hacia el box el primer pasajero de la lista (ya estan en orden en teoria)
50  {
51      NodoLista* nodo_aux; //guardamos primero
52
53      if (primero != NULL) //podemos desenlistar porque la lista no esta vacia
54      {
55          nodo_aux = primero;
56          primero = nodo_aux->siguiente_nl; // El nuevo primero es el siguiente al que desenlisto
57          delete nodo_aux; // borrarlo de memoria
58      }
59      else
60      {
61          cout << "Lista Vacia" << endl;
62      }
63  }
```

Para una lista no vacía, lo que haremos 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.

Arbol - Método *INORDEN*

```
15  Lista Arbol::inorden(NodoArbol* nabb, Lista& lista_arbol) //el primer parametro que tenemos que meter es la raiz del arbol
16  {
17      if (nabb != NULL)  //la raiz distinta de nulo
18      {
19          inorden(nabb->izquierda_na, lista_arbol);
20          lista_arbol.enlistar(nabb->pasajero_na);
21          inorden(nabb->derecha_na, lista_arbol);
22      }
23      return lista_arbol;
24  }
```

Este método almacena en una lista el resultado de recorrer los pasajeros del árbol en inorden (izquierda, raíz, derecha) de tal forma que primero establecemos una llamada recursiva a la función pasándole como parámetro el nodo izquierdo al nodo en el que nos encontramos actualmente de modo que lleguemos hasta el extremo izquierdo, almacenándolo así en la lista; después, enlistaremos el pasajero de la raíz y posteriormente haremos el mismo procedimiento con los pasajeros en la derecha, de tal forma que establecemos una llamada recursiva a la función pasándole por parámetro el nodo derecho del cual nos encontramos, hasta llegar al extremo.

Arbol - Método *INSERTAR*

```
27  void Arbol::insertar(Pasajero p, NodoArbol*& nabb)    //nabb va a ser la raiz cuando lo llamemos por primera vez
28  {
29      NodoArbol* nodo_nuevo = new NodoArbol (p, NULL, NULL);
30
31      if (!nabb) //el arbol no tiene raiz, es decir, esta vacio
32      {
33          nabb = nodo_nuevo;
34          raizz = nabb;
35      }
36      else //sí hay raiz, arbol no vacio
37      {
38          if (p.getId_pasajero() <= nabb->pasajero_na.getId_pasajero()) //si el id es menor o igual, es decir, por la izquierda
39          {
40              if (nabb->izquierda_na == NULL)  //si el nodo izq esta vacio
41              {
42                  nabb->izquierda_na = nodo_nuevo;
43              }
44              else
45              {
46                  insertar(p, nabb->izquierda_na);
47              }
48          }
49          else //mayor, es decir, por la derecha
50          {
51              if (nabb->derecha_na == NULL)  //si el nodo dcho esta vacio
52              {
53                  nabb->derecha_na = nodo_nuevo;
54              }
55              else
56              {
57                  insertar(p, nabb->derecha_na);
58              }
59          }
60      }
61  }
```

En este método establecemos un nuevo *NodoArbol*, que almacenará en la *raíz* al pasajero que estamos pasándole por parámetro a la función y cuyos punteros a la *izquierda* y a la *derecha* serán NULL, puesto que tratamos a cada nodo como un “nuevo árbol”. Entonces, si

el árbol no tiene *raíz*, diremos que su *raíz* ahora será el nuevo nodo que hemos creado (que recordemos, tiene al *pasajero* y sus punteros a *izquierda* y *derecha* como NULL).

En cambio, si el árbol posee de *raíz* (no está vacío), estudiaremos dónde insertar el nuevo pasajero: si a la izquierda o a la derecha de ésta.

- Si el *ID del pasajero* que pretendemos insertar es menor o igual al del pasajero de la raíz:
  - Si su *izquierda* es NULL, establecemos que ahora ese nuevo pasajero (nodo) será la izquierda de la raíz.
  - Si su *izquierda* NO es NULL, entonces llamaremos a la función pasándole como parámetro raíz, esta vez, el nodo izquierdo a la antigua raíz, y así se hará de forma recursiva hasta que el pasajero pueda ser insertado (comprobando nuevamente toda la información anterior y posterior a este punto).
- Si el *ID del pasajero* que pretendemos insertar es mayor al del pasajero de la raíz:
  - Procederemos a comprobar la información previamente definida, salvo que esta vez se consultará el nodo derecho al de la raíz, y no el izquierdo, y así de forma recursiva.

Box - Método *METER* (ATENCIÓN! No es un TAD)

```
12  void Box::meter(Pasajero p, int tiempo)
13  {
14      pasajero_box = p;
15      lleno = true;
16      pasajero_box.setTiempo_espera(tiempo - pasajero_box.getHora_llegada() + 1); // una vez el pasajero entra al Box, defino el tiempo que ha estado esperando (en la lista)
17      pasajero_box.setTiempo_box(tiempo);
18  }
```

Este método sirve para meter a un pasajero en el Box, y hemos pensado en poner un booleano *lleno* a true para indicar que no puede entrar ningún pasajero más, y además establecer el tiempo de espera del pasajero (es decir, el tiempo que ha transcurrido desde que llegó al aeropuerto hasta que ha sido atendido) así como el tiempo en el que ha entrado al box, para poder comprobar en la clase Aeropuerto cuándo el pasajero debe salir (que será cuando lleve “duración” tiempo).

4. Explicación del comportamiento del programa

Nuestro programa funciona de tal forma que hemos creado dos funciones en la clase principal, como requería el enunciado de la práctica: una **función de entrada**, en la cual definimos a los pasajeros mediante su *ID*, su *hora de llegada* y la *duración de su atención* dentro del box y los insertamos dentro de una *cola inicial* para un correcto curso de ejecución del programa (que está definida en una función de la clase Aeropuerto la cual explicamos a continuación) y una **función de salida**, que será la encargada de calcular y mostrar por pantalla el *tiempo medio de espera* de los pasajeros dentro del aeropuerto a partir de una *cola final* en la que hemos ido insertando los pasajeros según salían del Box.

```
16  int main(int argc, char* argv[])
17  {
18      // VARIABLES //
19      Aeropuerto aeropuerto;
20      Cola cola_final;
21      //float tiempo_espera_total;
22
23      //FUNCIONES
24      cola_final = llegadaPasajeros(aeropuerto);
25      salida(cola_final);
26  }
```

```
28  Cola llegadaPasajeros(Aeropuerto aeropuerto)
29  {
30      Cola cola_inicio;
31
32      Pasajero p1 = Pasajero(5, 15, 5);
33      Pasajero p2 = Pasajero(3, 22, 6);
34      Pasajero p3 = Pasajero(7, 9, 7);
35      Pasajero p4 = Pasajero(6, 14, 3);
36      Pasajero p5 = Pasajero(4, 18, 4);
37      Pasajero p6 = Pasajero(1, 27, 17);
38      Pasajero p7 = Pasajero(9, 0, 6);
39      Pasajero p8 = Pasajero(2, 26, 3);
40      Pasajero p9 = Pasajero(8, 4, 7);
41
42      cola_inicio.encolar(p1);
43      cola_inicio.encolar(p2);
44      cola_inicio.encolar(p3);
45      cola_inicio.encolar(p4);
46      cola_inicio.encolar(p5);
47      cola_inicio.encolar(p6);
48      cola_inicio.encolar(p7);
49      cola_inicio.encolar(p8);
50      cola_inicio.encolar(p9);
51
52      return aeropuerto.gestionAeropuerto(cola_inicio);
53  }
```

```
55  void salida(Cola cola_final)
56  {
57      float tiempo_espera_total = 0;
58
59      if(cola_final.getPrimero())
60      {
61          NodoCola* nodo;
62          nodo = cola_final.getPrimero();
63          while(nodo != cola_final.getUltimo()->getSiguiente())
64          {
65              tiempo_espera_total = tiempo_espera_total + nodo->getPasajeroNC().getTiempo_espera();
66              nodo = nodo->getSiguiente();
67          }
68      }
69
70      cout << "-------------------------------------------------" << endl;
71      cout << "" << endl;
72      cout << "        GRACIAS POR VIAJAR CON NOSOTROS" << endl;
73      cout << "" << endl;
74      cout << "-------------------------------------------------" << endl;
75      cout << " " << endl;
76      cout << "*********** COLA FINAL ***********" << endl;
77      cout << " " << endl;
78      cola_final.imprimir();
79      cout << "-------------------------------------------------" << endl;
80      cout << "" << endl;
81      cout << "Tiempo media de espera: " << tiempo_espera_total / 9 << endl;
82      cout << "" << endl;
83      cout << "-------------------------------------------------" << endl;
84      cout << "" << endl;
85  }
```

Como acabamos de adelantar, toda la estructura del funcionamiento de nuestro programa está definida en una función de nuestra clase Aeropuerto: **gestionAeropuerto()**, llamada desde la función de entrada de la clase principal y que recibe como parámetro la cola en la que metimos a todos los pasajeros desde un primer momento y devuelve la *cola_final*. Esta función es la encargada de llevar a cabo toda la ejecución del código:

- En primer lugar, instanciamos todos los TADs (Pila, Lista, NodoArbol y Árbol), además de las clases Pasajero y Box.
- Después, imprimimos esa *cola_inicial* en la que metimos a los pasajeros y de la cual saldrán todos y cada uno de ellos a un árbol binario de búsqueda, haciendo uso de un bucle *while* en el que mientras que la *cola_inicial* no esté vacía, llamaremos a la función *arbol.insertar(p, nabb)* y desencolaremos.

```
23      cout << "********** COLA INICIO **********" << endl;
24      cout << " " << endl;
25      cola_inicio.imprimir();
26      while(!cola_inicio.isVacia())                    // COLA_INICIO --> ARBOL
27      {
28          Pasajero p = cola_inicio.getPrimeroPasajero(); // cogemos el primer pasajero de la cola de inicio
29          arbol.insertar(p, nabb);                     // metemos el pasajero en el arbol
30          cola_inicio.desencolar();                    // desencolamos quitando el primer pasajero que ya hemos metido en el arbol
31      }
```

- A continuación, insertaremos a todos los pasajeros del árbol en una *lista_arbol* en inorden; con esto conseguiremos ordenarlos en función de su *ID* (ergo, en función de su *hora de llegada*) e imprimiremos esta *lista_arbol* para ver que realmente se ordenan.

```
33      cout << "************ ARBOL ************" << endl;
34      cout << " " << endl;
35      arbol.imprimir(nabb, 0);
36      cout << "  " << endl;
37
38      lista_arbol = arbol.inorden(nabb, lista_arbol);     // ARBOL --> LISTA_ARBOL
39
40      cout << "********** LISTA ARBOL **********" << endl;
41      cout << " " << endl;
42      lista_arbol.imprimir();
```

- Posteriormente, de esta lista serán apilados uno a uno en una *pila* respetando el orden de sus *ID*, apilando en primer lugar a aquellos pasajeros cuyo *ID* es menor (es decir, cuya *hora de llegada* es más tarde).

```
43
44      while(!lista_arbol.isVacia())                    // LISTA_ARBOL --> PILA
45      {
46          Pasajero pa = lista_arbol.getPrimeroPasajero(); // cogemos el primer pasajero de la lista_arbol
47          pila.apilar(pa);                             // metemos el pasajero en la pila (último abajo con menor ID, cima con mayor ID)
48          lista_arbol.desenlistar();                   // desenlistamos quitando el pasajero que hemos metido en la pila
49      }
50
51      cout << "************* PILA *************" << endl;
52      cout << " " << endl;
53      pila.imprimir();
```

Una vez que tenemos insertados a todos los pasajeros en la pila, establecemos un nuevo bucle *while* que será el que llevará el total control de la afluencia de los pasajeros en el aeropuerto; y es que mientras que la pila no esté vacía (es decir, queden pasajeros por llegar al aeropuerto), o que la lista no esté vacía (es decir, queden pasajeros por ser atendidos) o que el box no esté vacío (es decir, quede un pasajero por acabar de ser atendido), haremos lo siguiente:

- Una comprobación mientras que la pila no esté vacía de que la *hora de llegada* del pasajero que está en la *cima* (el siguiente que debe llegar) sea igual a una variable *tiempo* que irá incrementando a lo largo de la ejecución del programa, que

comenzará en 0 e irá sumando a 1, 2, 3… para poder insertarlo así en la lista.

```
//pila vacia --> todos en el aeropuerto
//lista vacia --> todos han pasado al box
//box vacio --> todos atendidos y fuera del aeropuerto

while((!pila.isVacia()) || (!lista.isVacia()) || (!box.isVacio()))  //PILA --> LISTA --> BOX
{
    if (!pila.isVacia())
    {
        Pasajero pasajeroPila = pila.getCimaPasajero();
        if(pasajeroPila.getHora_llegada() == tiempo) // PILA --> LISTA
        {
            lista.enlistar(pasajeroPila);
            pila.desapilar();
            cout << " ####### BIENVENIDO A AEROLINEAS EEDD S.A. ####### " << endl;
            cout << "Entra en el aeropuerto el pasajero con ID: " << pasajeroPila.getId_pasajero() << " con tiempo: " << tiempo << endl;
            cout << "" << endl;
        }
    }
```

- Una comprobación mientras que la lista no esté vacía, mirar si el primer pasajero esperando en esa lista puede entrar al Box (siempre que éste esté vacío). En caso afirmativo, llamaremos a la función *box.meter(p, tiempo)* para meterle en el Box y definir su *tiempo de espera* como la diferencia entre el valor de la variable tiempo en la que sale de la lista y el valor de cuando entró; además, actualizaremos el valor global de la variable *tiempo_espera_total*, incrementándolo tantas unidades como tiempo haya esperado ese pasajero.

- En caso negativo, es decir, que el Box esté lleno, también se comprobará constantemente que el pasajero que esté dentro pueda salir, que será cuando su tiempo de *duración* se haya completado y además será añadido a una *cola final* como indica el enunciado de la práctica.

```
75          // DE LISTA NORMAL A BOX
76          if (!lista.isVacia())                       //la lista esta llena
77          {
78              Pasajero pasajeroLista = lista.getPrimeroPasajero();
79              if(box.isVacio())
80              {
81                  box.meter(pasajeroLista, tiempo-1); // LISTA --> BOX
82                  //tiempo_espera_total = tiempo_espera_total + box.getPasajero().getTiempo_espera();
83                  lista.desenlistar();
84                  box.imprimir(tiempo);
85              }
86              else   //cuando en el box hay un pasajero
87              {
88                  if (box.getPasajero().getTiempo_box() == tiempo)
89                  {
90                      cola_final.encolar(box.getPasajero());
91                      box.sacar();
92                  }
93              }
94          }
95          else        //la lista esta vacia
96          {
97              if (!box.isVacio())
98              {
99                 if (box.getPasajero().getTiempo_box() == tiempo)
100                 {
101                     cola_final.encolar(box.getPasajero());
102                     box.sacar();
103                 }
104             }
105         }
106         tiempo++;
107     }
108     return cola_final;
109 }
```

Cuando tanto la *pila* como la *lista* como el *box* estén vacíos, retornaremos finalmente la *cola_final* en la que hemos ido insertando a todos los pasajeros que salían del Box; será entonces cuando, en la función de *salida* de nuestro programa principal, recorramos todos y cada uno de ellos para ir incrementando el valor de la variable *tiempo_espera_total*, para calcular así el tiempo medio de espera de todos los pasajeros. En nuestro caso será un total de 66 unidades de tiempo, que entre 9 pasajeros, da un total de 7,33 minutos de espera media.

```
--------------------------------------------------

Tiempo media de espera: 7.33333

--------------------------------------------------
```

Un buen resumen de todo el procedimiento del programa, para visualizar y comprender el paso por cada TAD, es el siguiente:

Cola_Inicio → Árbol → Lista_Arbol → Pila → Lista → Box → Cola_Final

5. Bibliografía

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