PEC2

Estructuras de datos.

INDICE

DATOS PERSONALES

Nombre:

- Juan

Apellidos:

- Casado Ballesteros

DNI:

- 09108762A

Correo:

- jcb7777777@gmail.com

Asignatura:

- Estructuras de datos GII martes 10:00-12:00
- Laboratorio con Hamid Tayebi de 12:00 a 14:00

Se utiliza el símbolo **$** para señalar en cada apartado el contenido añadido o corregido con respecto a la PEC1.

COLA

Detalles y justificación de la implementación.

Utilizo el TAD cola para implementar los servidores y como estructura para almacenar a los Clientes antes de haberlos introducido en un servidor. Cada cliente está almacenado en un NodoCola que son los elementos de los que la Cola está compuesta.
Cola se comporta como una estructura FIFO en la que el primer elemento que haya entrado será el primero en salir, para hacer esto me apoyo en dos punteros, uno que marca el inicio de la cola llamado primeros que es por donde saco los Clientes y otro llamado último que es por donde introduzco Clientes nuevos.

Constructor:

En el constructor iniciaré ambos punteros que dan soporte a la estructura de la Cola, primero y último a NULL para indicar que la cola está vacía, también inicio cantidad_de_clientes a 0 por el mismo motivo.

Métodos públicos:

void insertar (Cliente c);

- Introduce un Cliente nuevo en la Cola, para ello hará apuntar el nuevo cliente al que está apuntado por el puntero que marca el último y actualiza el valor de ese puntero para que apunte al nuevo introducido que ahora ocupará la última posición.
- Esta función al ser llamada incrementa en uno clientes_en_cola de modo que se lleva un recuento activo de ellos.

Cliente eliminar ();

- Elimina y retorna el Cliente que entró primero de todos en la Cola, para ello accede al puntero primero lo hace apuntar al siguiente elemento y elimina el que antes era el primero.
- Esta función al ser llamada decrementa en uno clientes_en_cola de modo que se lleva un recuento activo de ellos.

void mostrar ();

- Lee la Cola de forma iterativa accediendo en cada paso al NodoCola apuntado por el NodoCola anterior por el puntero siguiente ejecutando la función show_cliente() de cada cliente en cada NodoCola.

bool es_vacia ();

- Retorna primero como un booleano, primero valdrá NULL==0==false cuando la cola esté vacía, en cualquier otro caso retornará true.
- NOTA: es_vacia() devuelve FALSE cuando es vacía y TRUE cuando está llena, aunque por el nombre de la función parece que debería de devolver los resultados al revés al hacerlo de este modo me evito negar dos veces, una en el return de la función y otra al utilizarla.

int cantidad_de_clientes();

- Retorna la variable privada clientes_en_cola que ya hemos explicado como actualiza su valor que se corresponderá en todo momento con los clientes que haya en la cola.

Cliente ver_primero (void);

- Me devuelve sin borrarlo de la cola al primer cliente que entró en ella y que todavía no ha salido.

$ void vaciar();

- Elimina todos los clientes de la cola mediante llamadas a eliminar() mientras queden clientes en ella.

Atributos privados:

pnodo primero,ultimo;

- Mediante estos punteros a los NodoCola en los que se almacena el primer y el último Cliente gestionamos la cola permitiendo introducir datos en ella o sacándolos.

Int clientes_en_cola;

- En esta variable almacenan dinámicamente según se añaden o se eliminan la cantidad de clientes que hay en la cola en todo momento.

Soluciones a las dificultades encontradas.

Para poder saber en todo momento y de forma eficiente la cantidad de clientes en cada cola actualizo contantemente la variable clientes_en_cola al añadir o eliminar NodoClola a la Cola. Así evito tener que recorrer la lista contándolos para saber cuántos hay.

NODOCOLA

Detalles y justificación de la implementación.

Utilizo este TAD para almacenar los Clientes en una cola. El TAD está compuesto por un Cliente que es el dato que queremos almacenar en la Cola y un puntero que da sustento a la estructura pues apuntará al siguiente NodoCola de la Cola.
A los punteros que apuntan a un NodoCola le damos el nombre de pnodoCola de forma que hacemos el código más legible.

Constructor:

Iniciamos cada NodoCola con un cliente y con un puntero de tipo NodoCola apuntando a NULL. Esto lo hacemos para poder hacer apuntar ese puntero al NodoCola que corresponda cuando generemos la estructura de cola. El puntero llamado siguiente es un atributo privado, pero el nodo hacemos que sea friend class con Cola para poder desde allí editar su valor.

Atributos privados:

Cliente cliente;

- En esta variable almacenaré el valor del cliente que NodoCola debe contener.

NodoCola *siguiente;

- Haré apuntar este puntero al NodoCola que corresponda para formar la estructura de Cola.

Soluciones a las dificultades encontradas.

Con la intención de simplificar el código y hacerlo más visible realizo un typedef a los punteros de tipo NodoCola como pnodoCola, además hago de siguiente un atributo privado, pero permito que Cola lo edite ya que hago un friend class con ella. Por comodidad iniciaré todos los punteros a siguiente NodoCola como NULL para poder darles valores con posterioridad desde la clase Cola.

LISTA

Detalles y justificación de la implementación.

Utilizo este TAD para gestionar los pnodoLista en los que almacenaré Cliente una vez hayan comprado su entrada. La lista es doblemente enlazada para facilitar la ordenación de los Clientes en función de su código de entrada.

La lista estará gestionada por dos punteros, primero y puntero, a NodosLista, uno que apunte al primer elemento de la lista ordenada y otro que utilizo para poder recorrerla a la hora de ver dónde colocar cada elemento para que esté siempre ordenada.

Constructor:

Inicio ambos punteros, primero y puntero a NULL para indicar que la cola está vacía, también inicio clientes_en_lista a 0 por el mismo motivo.

Métodos públicos:

void insertar (Cliente c);

- Introduce un Cliente nuevo en la Lola, para ello puntero irá iterando sobre la lista hasta que o bien el NodoLista siguiente al que está apuntando contenga un Cliente con un identificador de entrada menor que el del Cliente que queremos introducir o que llegue al final de la lista.

  Si ha llegado al final de la lista puede ser por dos causas, la lista está vacía por lo que el NodoLista que contiene al Cliente nuevo será el primer elemento de la Lista o porque hemos recorrido toda la lista sin que haya otro Cliente anteriormente introducido en ella con un identificador de entrada de mayor valor en cuyo caso introduciremos nuestro Cliente detrás del NodoLista al que está apuntando puntero. Para realizar esto utilizaremos el puntero a NodoLista que apunta al elemento anterior contenido en cada NodoLista, recordamos que Lista es una esrructura tipo lista doblemente enlazada.

  Si se para por haber llegado a un NodoLista cuyo NodoLista siguente contiene un Cliente con un identificador de enetrada de menor valor comprobamos que el NodoLista al que apunta puntero contenga un Cliente con identificador de entrada mayor, de serlo insertamos detrás de él y de no serlo insertaremos el nuevo NodoLista que contiene al nuevo Cliente delante.
- Esta función al ser llamada incrementa en uno clientes_en_lista de modo que se lleva un recuento activo de ellos.

Cliente eliminar ();

- Elimina y retorna el Cliente que entró primero de todos en la Lista, para ello accede al puntero primero lo hace apuntar al siguiente elemento y elimina el que ates era el primero.
- Esta función al ser llamada decrementa en uno clientes_en_lista de modo que se lleva un recuento activo de ellos.
- Esta función se utiliza exclusivamente para resetear el programa.

void mostrar ();

- Lee la Lista de forma iterativa accediendo en cada paso al NodoLista apuntado por el NodoLista anterior por el puntero siguiente ejecutando la función show_cliente_completo() de cada cliente en cada NodoCola, mostrando sus datos junto a su identificador de entrada.

bool es_vacia ();

- Retorna primero como un booleano, primero valdrá NULL==0==false cuando la cola esté vacía, en cualquier otro caso retornará true.
- NOTA: es_vacia() devuelve FALSE cuando es vacía y TRUE cuando está llena, aunque por el nombre de la función parece que debería de devolver los resultados al revés al hacerlo de este modo me evito negar dos veces, una en el return de la función y otra al utilizarla.
- Esta función se utiliza exclusivamente para alertar de que la lista no se puede mostrar pues está vacía.

int cantidad_de_clientes();

- En esta variable almacenan dinámicamente según se añaden o se eliminan la cantidad de clientes que hay en la cola en todo momento.
- Esta función se utiliza exclusivamente para mostrar en el menú los clientes que ya hay comprado su entrada y de forma activa al mostrar la lista de entradas compradas durante los procesos de compra.

$ void vaciar();

- Elimina todos los clientes de la lista mediante llamadas a eliminar() mientras queden clientes en ella.

$ void duplicar();

- Crea un array ordenado de Clientes a partir de los clientes contenidos por los NodoLista de la estructura. Esto se realiza para poder crear el Arbol sin destruir la Lista.

Métodos privados:

void insertar_delante(pnodoLista n);

- Mediante un puntero a NodoLista auxiliar hago que el NodoLista apuntado por el puntero pasado a la función quede entre el NodoLista al que apunta puntero y el siguiente NodoLista.
- El puntero a siguiente del NodoLista apuntado por puntero apuntará al nuevo NodoLista, el anterior del siguente al nuevo y los punteros siguiente y anterior del nuevo a su siguiente y anterior en la lista de forma que quede incluído en la estructura.

void insertar_detras(pnodoLista n);

- Mediante un puntero a NodoLista auxiliar hago que el NodoLista apuntado por el puntero pasado a la función quede entre el NodoLista al que apunta puntero y el anterior NodoLista.
- El puntero a anterior del NodoLista apuntado por puntero apuntará al nuevo NodoLista, el siguiente del anterior al nuevo y los punteros siguiente y anterior del nuevo a su siguiente y anterior en la lista de forma que quede incluído en la estructura.

void insertar_primero(pnodoLista n);

- Inserta un NodoLista en la lista haciendo apuntar sus punteros a siguiente y anterior a NULL y apuntando primero y puntero al nuevo NodoLista que será el primero de la misma.

Atributos privados:

pnodoLista puntero, primero;

- Mediante estos punteros a los NodoCola en los que se almacena el primer Cliente ordenado y el cliente el el que nos estemos fijando a la hora de ordenar gestionamos la cola permitiendo introducir datos en ella o sacándolos.

int clientes_en_lista;

- En esta variable almacenan dinámicamente según se añaden o se eliminan la cantidad de clientes que hay en la cola en todo momento.

Soluciones a las dificultades encontradas.

La variable clientes_en_lista es tratada del mismo modo que en la Cola para ir almacenando en todo momento el número de clientes que haya en la Lista.
Lista está organizada como una estructura de lista doblemente enlazada, esto facilita en clan medida la inserción de datos de forma ordenada en ella.

$ Crear un array de Clientes a partir de los Clientes almacenados en los nodos que forman la estructura es algo útil pues no permite obtener un array de Clientes ordenados como lo estaban en la lista haciendo así el formato Clientes ordenados exportable a otras estructuras como en el caso del Arbol.

NODOLISTA

Detalles y justificación de la implementación.

Utilizo este TAD para almacenar los Clientes en una cola. El TAD está compuesto por un Cliente que es el dato que queremos almacenar en la Lista, un puntero que apuntará al siguiente NodoLista de la Lista y otro que apunta al NodoLista anterior, dando a la estructura una forma de lista doblemente enlazada .

A los punteros que apuntan a un NodoLista le damos el nombre de pnodoLista de forma que hacemos el código más legible.

Constructor:

Iniciamos cada NodoLista con un cliente y dos punteros de tipo NodoLista apuntando a NULL. Esto lo hacemos para poder hacer apuntar ese puntero al NodoLista que corresponda cuando generemos la estructura de lista doblemente enlazada. El puntero llamado siguiente y el llamado puntero son atributos privados, pero NodoLista hacemos que sea friend class con Lista para poder desde allí editar su valor.

Atributos privados:

Cliente cliente;

- En esta variable almacenaré el valor del cliente que NodoLista debe contener.

NodoLista *siguiente;

- Haré apuntar este puntero al NodoLista que corresponda para formar la estructura de Lista.

NodoLista *siguiente;

- Haré apuntar este puntero al NodoLista que corresponda para formar la estructura de Lista siendo esta una lista doblemente enlazada.

Soluciones a las dificultades encontradas.

Con la intención de simplificar el código y hacerlo más visible realizo un typedef a los punteros de tipo NodoLista como pnodoLista, además hago de siguiente un atributo privado, pero permito que Lista lo edite ya que hago un friend class con ella. Por comodidad iniciaré todos los punteros a siguiente NodoLista como NULL para poder darles valores con posterioridad desde la clase Lista.

El NodoLista contendrá dos punteros para hacer que la lista sea doblemente enlazada.

$ ARBOL

Detalles y justificación de la implementación.

Este TAD gestino los pnodoArbol en los que hay almacenados Clientes. Se crea a partir de un Array de Clientes y su utilidad reside en permitir a los usuarios buscar a los Clientes por su DNI para obtener su identificador de entrada de forma eficiente.
Para lograr esto el árbol será de búsqueda binario. Existirá un Cliente ficticio que nos garantizará que la disposición de los clientes en el árbol sea óptima, este Cliente ficticio tendrá el DNI 49999999-M que es el DIN “medio”. Al insertar nuevos Clientes en el árbol estos quedarán a izquierda los de menor DNI y a la derecha los de mayor. Todos los métodos realizados sobre el árbol ya sea llenarlo, como mostrarlo, dibujarlo o destruirlo se implementan de forma recursiva para simplificar el código.
Para poder usar correctamente este TAD se deberá utilizar como un puntero a la estructura, de este modo podremos iniciarla con el Cliente ficticio que necesitemos, además esto nos permite prescindir de un método para saber si el Arbol está lleno o no pues lo podremos saber directamente con el puntero a la estructura.

Constructor:

Cuando creamos el árbol lo hacemos con un Cliente inicial colocado en su raíz, de este modo mejoramos la calidad de la búsqueda binaria.

Métodos públicos:

void generar(Cliente * c, int longitud);

- Crea un Árbol de búsqueda binario a partir de un array de Clientes. Utilizo un array a partir de los clientes en la Lista (es un duplicado de ella) para no tener que destruirla.
- Para insertar un nodo nuevo comprobamos que la raíz del subárbol no esté nula, si lo está insertamos y si no vemos si debemos de hacerlo a la derecha o a la izquierda de esta y cambiamos de subárbol hasta llegar a un nodo vacío.

void vaciar();

- Elimina de forma recursiva los pnodoArbol de la estructura recorriéndola por post orden liberando de memoria cada raíz.

Cliente buscar(char* DNI);

- Nos permite buscar un Cliente de forma recursiva por el Árbol con la finalidad de obtener su identificador de entrada.
- Vemos si la raíz del subárbol es vacía, si lo es el Cliente no está, si el Cliente que contiene es el que buscamos lo devolvemos y si es otro miramos si el que buscamos estría a la derecha o a la izquierda de él y reducimos el subárbol hasta llegar a uno de los casos de parada.

void show_pre_orden();

- De forma recursiva mostramos la estructura: raíz, izquierda y derecha.

void show_post_orden();

- De forma recursiva mostramos la estructura: izquierda, derecha, y raíz.

void show_in_orden();

- De forma recursiva mostramos la estructura: izquierda, raíz y derecha.

void draw();

- Dibuja el árbol de forma horizontal utilizado una cadena tipo string como auxiliar para saber qué símbolos preceden a qué Cliente. Muestra los símbolos que correspondan delante del Cliente mientras recorre el Arbol en modo pre-orden.

Métodos Privados:

void generar_recursivo(pnodoArbol nuevoNodo, char* nuevoDNI, pnodoArbol arbol);

void draw_recursivo(pnodoArbol arbol, string indent, bool is_tail);

void eliminar_recursivo_post_orden(pnodoArbol nodo);

Cliente buscar_recursivo(pnodoArbol arbol, char* DNI);

void show_pre_orden_recursivo(pnodoArbol arbol);

void show_post_orden_recursivo(pnodoArbol arbol);

void show_in_orden_recursivo(pnodoArbol arbol);

Todos ellos son llamados por el método público que les corresponde el cual los inicia con la raíz como primer nodo para comenzar las llamadas recursivas.

Atributos privados:

pnodoArbol raiz;

- Contiene al Cliente ficticio y es el nodo de inicio de la estructura a partir del cual se sostiene el Arbol.

Soluciones a las dificultades encontradas.

Las mayores dificultades surgieron entorno a iniciar el Cliente ficticio, estas fueron resueltas accediendo al Arbol desde gestor mediante un puntero y llamando al constructor del Arbol solo cuando fuera preciso iniciarlo.

$ NODOARBOL

Detalles y justificación de la implementación.

Utilizo este TAD para almacenar los Clientes en un Arbol. El TAD está compuesto por un Cliente que es el dato que queremos almacenar en el Arbol, un puntero que apuntará al nodo derecho que formará el subárbol derecho y otro que hará lo propio con el izquierdo.
A los punteros que apuntan a un NodoArbol le damos el nombre de pnodoArbol de forma que hacemos el código más legible.

Constructor:

Inicio cada pnodoArbol con un Cliente y con sus punteros derecho e izquierdo a NULL. Los punteros son atributos privados, pero NodoArbol hacemos que sea friend class con Arbol para poder desde allí editar su valor.

Atributos Privados:

Cliente cliente;

- En esta variable almacenaré el valor del cliente que NodoArbol debe contener.

NodoArbol *izquierda;

- Haré apuntar este puntero al subArbol izquierdo para formar la estructura de Arbol de búsqueda binario.

NodoArbol *derecha;

- Haré apuntar este puntero al subArbol derecho para formar la estructura de Arbol de búsqueda binario.

Soluciones a las dificultades encontradas.

Con la intención de simplificar el código y hacerlo más visible realizo un typedef a los punteros de tipo NodoArbol como pnodoArbol, además hago de siguiente un atributo privado, pero permito que Arbol lo edite ya que hago un friend class con él. Por comodidad iniciaré todos los punteros a siguiente NodoArbol como NULL para poder darles valores con posterioridad desde la clase Lista.
El NodoArbol contendrá dos punteros para hacer que sea binario de búsqueda.

$ COLORES

Detalles y justificación de la implementación.

Utilizo este TAD para dar color al texto que la aplicación muestra por pantalla, abstraer este proceso resulta útil en cualquier situación pues son recursos que se utilizarán de forma constante cada vez que se quiera mostrar información al usuario, pero en este caso lo es más aún ya que permite la compatibilidad WINDOWS-LINUX-MAC.
La forma de colorear el texto es distinta en cada plataforma y este TAD une bajo una misma interface este proceso. Para ello de definen los colores como instrucciones de preprocesador como: #define ROJO 'R' lo cual permite interactuar con la clase de forma intuitiva.

Métodos Públicos:

void colorear(char color);

- Convierte el color de texto al pasado al método.

void blanco(void);

- Retorna el color del texto al predeterminado (BLANCO).

CLIENTE

Detalles y justificación de la implementación.

Cliente será un TAD que se corresponderá con una persona que quiere comprar una entrada por internet, esta persona tendrá un nombre, un DNI y será VIP o no en la página de compra de entradas. De ser VIVP tendrá ventajas pues podrá comenzar a comprar su entrada dos horas antes que los clientes normales. Cada persona comenzará a comprar su entrada en una hora determinada y tardará una cantidad de tiempo en hacerlo, en este caso estos datos se corresponderán con valores aleatorios que cumplan los requisitos dados por los profesores, pero podrían ser datos reales.
Los clientes dependiendo de si son VIP o no recorrerán ciertos servidores de venta de entradas u otros hasta acabar todos en una lista ordenados por el número identificador de su entrada, el cual se generará una vez hayan salido de los servidores, es decir una vez hayan finalizado el proceso de compra de su entrada.

Constructor:

El TAD cliente contará con dos constructores distintos, uno en el que no se iniciarán ninguno de sus atributos y otro en el que se iniciarán la hora_de_inicio_de_compra y si el Cliente es registrado o no como parámetros y el DNI y el tiempo de compra como valores aleatorios.
En caso de que el Cliente sea no registrado su hora de inicio de compra se incrementará en dos horas ayudándonos de una función auxiliar que hará esto.
Existen dos constructores por comodidad a la hora de declarar Clientes.

Métodos públicos:

void show_cliente();

- Llama a otros métodos privados que explicaremos próximamente de modo tal que muestra por pantalla todos los datos del cliente excepto su identificador de entrada aplicándoles un formato.

void show_cliente_completo();
void show_cliente_completo_formato_reducido();

- Llama a los mismos métodos que show_cliente() y les aplica el mismo formato a la hora de mostrar los atributos del Cliente y además muestra con formato el identificador de entrada.

void generar_identificador_de_entrada();

- Dota al cliente de identificador de entrada, 4 letras aleatorias, este método es utilizado cuando los clientes salen de los servidores.

$ char*_identificador_de_entrada();

- Devuelve el identificador_de_entrada para poder ordenar la lista.

int get_hora_de_inicio_de_compra();

- Devuelve la hora de inicio de compra del cliente de modo que se pueda saber cuándo este debe de entrar en un servidor para comprar su entrada.

int get_tiempo_de_compra();

- Devuelve el tiempo que elCliente tardad en comprar su entrada, de este modo podremos saber cuándo sacarlo del servidor.

$ char* get_DNI();

- Devuelve el identificador de entrada para poder ordenar el Arbol.

Atributos privados:

char DNI [LEN_DNI];

- Aquí se almacenan las ocho letras y el número de DNI que son generados de forma aleatoria. El número es generado a partir de las letras como en los DNI reales.

int hora_de_inicio_de_compra;

- Aquí se almacena la hora a la que el Cliente debe entrar en los servidores a comprar su entrada.

int tiempo_de_compra;

- Aquí se almacena el tiempo que el Cliente debe permanecer dentro de un servidor, es lo que tarda en comprar su entrada.

bool registrado;

- Dice si el cliente está registrado o no, a la hora de la implementación estar registrado supone tener un tiempo de inicio de compra dos horas menor y entrar en un servidor formado por una cola única en vez de en uno formado por cuatro colas.

char identificador_de_entrada [LEN_IDENTIFICADOR];

- Cuatro letras aleatorias que representan la identificación de la entrada.

Métodos Privados:

void show_identificador_de_entrada();

- Muestra las 4 letras del identificador de entrada en pantalla sin formato.
- Esta función es llamada solo por las que muestran el cliente con formato.

void generar_DNI();

- Genera 8 letras aleatorias y las guarda en DNI, con ellas genera un número tal y como se hace con los DNI reales y guarda todo en DNI.
- Esta función es llamada solo por el constructor.

void show_DNI();

- Muestra el DNI sin formato.
- Esta función es llamada solo por las que muestran el cliente con formato.

$ void set_DNI();

- Permite crear un Cliente ficticio con DNI 49999999-M al pasar un tiempo negativo como inicio de compra.

void generar_hora_de_inicio_de_compra();

- Suma dos horas a la hora de inicio de compra en el caso de que el Cliente sea no registrado.
- Esta función es llamada solo por el constructor.

void show_hora_de_inicio_de_compra();

- Muestra la hora de inicio de compra con formato parcial, en vez de mostrar el número en el que se guardan los minutos del inicio de la compra la muestra como hh:mm.
- Esta función es llamada solo por las que muestran el cliente con formato.

void generar_tiempo_de_compra();

- Da un número aleatorio de 1 a 10 al tiempo de compra.
- Esta función es llamada solo por el constructor.

void show_tiempo_de_compra();

- Muestra sin formato el tiempo que el cliente deberá de pasar en el servidor.
- Esta función es llamada solo por las que muestran el cliente con formato.

void show_tipo();

- Muestra el tipo de cliente VIP o NO_REGISTRADO

Soluciones a las dificultades encontradas.

Por comodidad todos los procesos que se realizan dentro del Cliente están subdivididos en funciones que se llaman unas a otras, de esta forma resulta más fácil ir probando cada apartado de código por separado, también resulta más fácil realizar modificaciones sobre él.

Evito crear dos clases distintas de Cliente (VIP y NO_REGISTRADO) mediante el uso de un booleano que los diferencia, así no es necesario realizar herencia. Esta variable se emplea para saber si en cliente debe de ir a servidor VIP o a los cuatro servidores habilitados para los no registrados.

La hora a la que los clientes no registrados comienzan a comprar entradas y el rango de duración de la compra de la misma está definido como:

#define HORA_DE_INICIO_NO_REGISTRADOS 120

#define TIEMPO_MAXIMO_DE_COMPRA 10

de modo que resultaría fácil modificarlo, pero sin que estén dentro de ninguna variable.

GESTOR

Detalles y justificación de la implementación.

Utilizo el TAD Gestor como interface para controlar los servidores mediante la ejecución de funciones desde el main. En gestor se localiza una Cola que actúa como servidor para los Clientes VIP y otra Cola que los almacena antes de entrar al servidor. También se encuentra un array con otras cuatro Colas para representar a los cuatro servidores de los clientes NO REGISTRADOS, con otra Cola más para almacenarlos antes de hacerlos pasar por los servidores. Por último, hay una Lista que como ya hemos visto es doblemente enlazada y se encuentra siempre ordenada por los identificadores de entrada de los Clientes que contiene.
Gestor permite generar una cantidad determinada (está declarada en un define) de Clientes VIP y otra de Clientes NO REGISTRADOS, se pueden ver o borrar ambos tipos. También se puede hacer una simulación de su paso a través de los servidores. Se iniciará un reloj y los Clientes entrarán en ellos cuando llegue su hora y saldrán cuando hayan estado su tiempo de compra siendo los primeros de la cola. En el caso de los Clientes NO REGISTRADOS hay un algoritmo que determina en qué Cola colocarse, la primera menos ocupada siguiendo el orden del array. Cuando los Clientes salen de las Colas se van a la Lista la cual se puede visualizar.

Posteriormente se podrá crear un Arbol a partir de un array que se obtiene de la Lista, este se podrá visualizar de cuatro formas distintas y sobre él se podrán buscar los Clientes en función de su DNI

Constructor.

Se inicia el puntero arbol a NULL, todo lo demás, el llenado de las Colas y la generación o borrado de Clientes se realiza por parte del usuario desde el main.

Métodos públicos:

void generar_solicitudes_de_clientes_VIP(void);

- Genera clientes tipo VIP asignándoles a cada uno un tiempo de inicio de compra que será igual al inicio de compra del cliente anterior más un aleatorio entre cero y cinco, guarda los clientes en una cola ordenados por su hora de inicio. SI la Cola de Clientes VIP en espera está llena y se vuelve a solicitar su llenado esta se reescribe.

void mostrar_clientes_VIP(void);

- Muestra la cola de los Clientes VIP que todavía no han entrado al servidor.

void borrar_clientes_VIP(bool texto);

- Elimina la Cola de los Clientes VIP que todavía no han entrado al servidor.

void generar_solicitudes_de_clientes_no_registrados(void);

- Genera un número definido de Clientes NO REGISTRADOS y los almacena en una Cola, si la Cola está llena la reescribe. El proceso de llenado de cola es igual que para los VIP, el ajuste de la hora de inicio de compra se hace dentro del TAD Cliente.

void mostrar_clientes_no_registrados(void);

- Muestra la cola de los Clientes NO REGISTRADOS que todavía no han entrado al servidor.

void borrar_clientes_no_registrados(bool texto);

- Elimina la Cola de los Clientes NO REGISTRADOS que todavía no han entrado al servidor.

void simular_venta_a_clientes_VIP(void);

- Hace pasar a los Clientes VIP por su servidor dedicado, cuando han estado tanto tiempo como indica su tiempo de compra siendo los primeros de la Cola les saca de servidor, asigna un identificador de entrada e introduce en la Lista ordenada.

void simular_venta_a_clientes_no_registrados(void);

- Selecciona el servidor correcto de los cuatro disponibles para los Clientes NO REGISTRADOS según los criterios indicados y cuando han estado tanto tiempo como indica su tiempo de compra siendo los primeros de la Cola les saca de servidor, asigna un identificador de entrada e introduce en la Lista ordenada.

void mostrar_entradas_vendidas(void);

- Muestra la Lista ordenada de los Clientes con su identificador de entrada.

void reiniciar(void);

- Borra todas las Colas de los Clientes en espera y la Lista de clientes con entradas compradas si es que no estban vacías ya.

int cantidad_no_registrados();

- Devuelve el número de Clientes NO REGISTRADOS en espera.

int recuento_VIP();

- Devuelve el número de Clientes VIP en espera.

int recuento_entradas();

- Devuelve el número de Clientes que ya han comprado su entrada.

$ void generar_arbol(void);

- Extrae de la Lista un array de clientes y con él genera un Arbol de búsqueda binario, de este modo no se elimina la lista.

$ void buscar_cliente(void);

- Pide un DNI válido al usuario y lo busca en el Arbol, si etá muestra el Cliente y si no notifica su ausencia.

$ void mostrar_arbol_en_pre_orden(void);

$ void mostrar_arbol_en_post_orden(void);

$ void mostrar_arbol_en_in_orden(void);

$ void dibujar_arbol(void);

- Llama a los métodos que dibujan el árbol, cada método se explica en el TAD Arbol

$ int cantidad_no_registrados_total();

- Dice cuantos Clientes NO_REGISTRADOS activos totales hay en el sistema.

$ int recuento_VIP();

- Dice cuantos Clientes VIP activos totales hay en el sistema.

$ bool estado_arbol();

- Notifica si el Arbol está vacío o no.

Atributos privados:

Cola VIP_en_espera;

- En esta cola se almacenan los Clientes VIP cuando se generan a la espera de entrar a su servidor, está ordenada por hora de inicio de compra.

Cola servidor_VIP;

- Esta Cola representa al servidor en el que los Clientes VIP pueden comprar las entradas.

Cola no_registrado_en_espera;

- En esta cola se almacenan los Clientes NO REGISTRADOS cuando se generan a la espera de entrar a su servidor, está ordenada por hora de inicio de compra.

Cola servidor_no_registrado[SERVIDORES_NO_REGISTRADOS];

- Con estas 4 Colas se representan los servidores en los que los Clientes NO REGISTRADOS pueden comprar sus entradas.

Lista clientes_terminados;

- En esta Lista se almacenen los Clientes sin importar su tipo que ya hayan comprado su entrada, está ordenada por el identificador de entrada.

$ Arbol *arbol;

- Puntero a la estructura Arbol en la que se puede buscar a un Cliente por su DNI para obtener su identificador de entrada de forma eficiente.

Métodos privados:

void show_hora(int t);

- Se utiliza esta función para dar formato al reloj que controla cuando los CLientes entran o salen del servidor antes de mostrarloe en pantalla. El formato dado será hh:mm que coincide con el formato en el que se muestr el tiempo de inicio de compra de los Clientes.

int recuento_no_registrados();

- Cuenta Cuantos Clientes NO REGISTRADOS hay en total entre los cuatro servidores en los que pueden comprar las entradas. Esta función se utiliza para saber cuándo todos los Clientes han terminado de comprar su entrada y poder dar por terminada la venta de las mismas.

void actualizar_pantalla_VIP(int tiempo, int tiempo_en_proceso);

- Muestra el estado de la Cola de Clientes VIP en espera, del servidor dedicado y de la Lista en un tiempo concreto. Esta función existe por hacer más legible el código y separar los cálculos de si los Clientes deben entrar o salir de los servidores de la fase de mostrar el estado del proceso.

void actualizar_pantalla_no_registrado(int tiempo, int * tiempo_en_proceso);

- Muestra el estado de la Cola de Clientes NO REGISTRADOS en espera, de sus servidores y de la Lista en un tiempo concreto. Esta función existe por hacer más legible el código y separar los cálculos de si los Clientes deben entrar o salir de los servidores de la fase de mostrar el estado del proceso.

$ void pedir_DNI_validado(char*);

$ bool validar_formato(char*);

$ bool validar_numeros(char*);

$ bool validar_letra(char*);

- Funciones auxiliares que gestionan la obtención de un DNI válido por parte de el usuario.

Soluciones a las dificultades encontradas.

Para saber cuándo un Cliente lleva tanto tiempo como marca su tiempo de compra siendo el primero de la Cola que implementa el servidor en el que está utilizo una variable auxiliar que llamo tiempo en proceso. Esta variable la incremento en uno si hay un Cliente en la Cola y la pongo a cero cuando ha alcanzado a tiempo de compra y por tanto cuando he sacado al Cliente del servidor.

En el caso de los Clientes NO REGISTRADOS en vez de utilizar solo una variable auxiliar utilizo un array de tantas variables como servidores haya para realizar el mismo proceso.

El número de Clientes de cada tipo, la velocidad a la que pasa el tiempo, el número de servidores para Clientes NO REGISTRADOS o la distancia entre el tiempo de compra de un Cliente y otro están determinadas por constantes en defines, por lo que son fácilmente modificables antes de ejecutar el programa.

Las funciones de actualizar pantalla, aunque sean una solución un poco fea ya que les paso los tiempos por parámetro son útiles porque simplifican el código, lo hacen más legible y fácil de modificar y mantener. Hacer de los temporizadores Atributos de la Clase no me pareció una solución adecuada ya que son específicos de las funciones de simulación y no definen al Gestor.

COMPORTAMIENTO DEL PROGRAMA

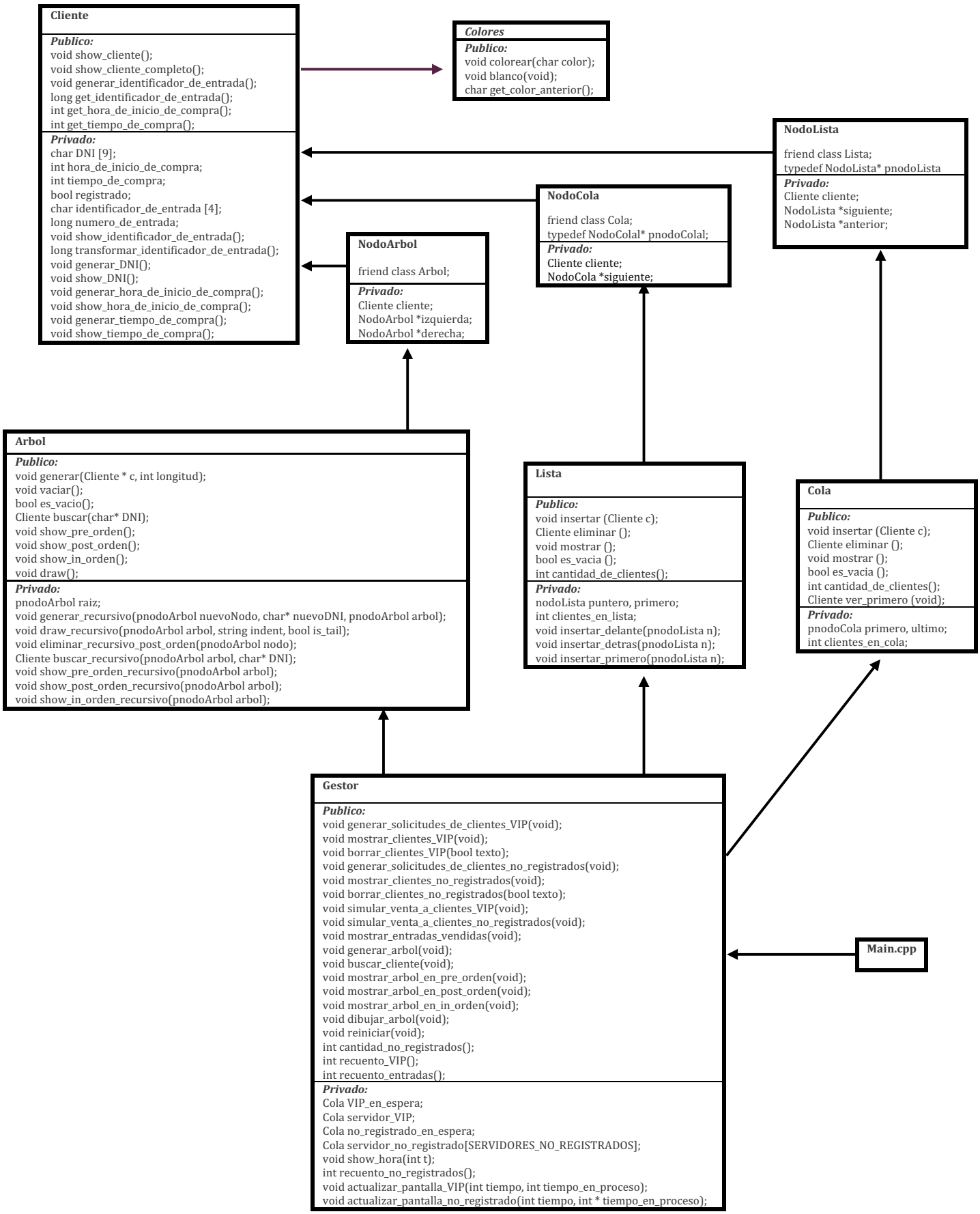
El usuario se encuentra con un conjunto de acciones que puede realizar. Cada acción corresponde a una letra que introducirá por pantalla. De este modo el usuario generará Clientes VIP o NO REGISTRADOS pudiendo borrarlos, volverlos a generar o pudiendo visualizarlos. Simulará el proceso de compra de los Clientes haciendo pasar a los VIP por una Cola de la que tendrán que ser primeros tanto tiempo como indique su tiempo de comprar para poder salir. En el caso de los NO REGISTRADOS habrá cuatro colas y se colocarán en la primera más vacía. Una vez finalizado el proceso de compra de cada Cliente a este se le asignará un identificador de entrada y será introducido en una Lista ordenada por él. Finalmente, el usuario podrá decidir si ver la Lista, si reiniciar el programa o si generar más Clientes para repetir el proceso.

El programa está a prueba de fallos de ejecución, no hay posibles acciones de usuario que lleven a sacar un Cliente o a mostrarlo cuando la Lista está vacía o acciones que pueda realizar que sean incompatibles con el programa.

$ Una vez tenemos la Lista podemos generar un Arbol que será de búsqueda binario a partir de una copia de los Clientes de la lista que introducimos en un array para evitar eliminarla. Este Arbol podremos mostrarlo de tres formas: in-orde, post-orden y pre-orden, también podremos dibujarlo. La utilidad del Arbol es que nos permite buscar por DNI y de una forma eficiente a los Clientes para obtener su identificador de entrada.

Los procesos más detallados sobre la implementación de cada acción están reflejados en el TAD correspondiente en su método correspondiente.

$ DIAGRAMA UML



CÓDIGO FUENTE

NodoLista.hpp

```
#ifndef NODOLISTA_HPP
#define NODOLISTA_HPP

#include "Cliente.hpp"

class NodoLista
{
	public:
		NodoLista(Cliente c, NodoLista *sig = NULL, NodoLista *ant = NULL);
		~NodoLista();
	private:
		Cliente cliente;
		NodoLista *siguiente;
		NodoLista *anterior;

		friend class Lista;
};

typedef NodoLista* pnodoLista;

#endif // NODOLISTA_HPP
```

NodoLista.cpp

```
#include "NodoLista.hpp"

NodoLista::NodoLista(Cliente c, NodoLista* sig, NodoLista* ant)
{
	cliente = c;
	siguiente = sig;
	anterior = ant;
}

NodoLista::~NodoLista()
{
}
```

Lista.hpp

```
#ifndef LISTA_HPP
#define LISTA_HPP

#include "NodoLista.hpp"

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

		void insertar (Cliente c);
		Cliente eliminar ();
		void mostrar ();
		bool es_vacia ();
		int cantidad_de_clientes();
		void vaciar();
		void duplicar(Cliente * c, int longitud);

	private:
		pnodoLista puntero, primero;
		int clientes_en_lista;
		void insertar_delante(pnodoLista n);
		void insertar_detras(pnodoLista n);
		void insertar_primero(pnodoLista n);

};

#endif // LISTA_HPP
```

Lista.cpp

```
#include "Lista.hpp"

Lista::Lista()
{
	primero = NULL;
	puntero = NULL;
	clientes_en_lista = 0;
}

void Lista::insertar(Cliente c)
{
	pnodoLista nuevo = new NodoLista(c);
	char* identificador_cliente = c.get_identificador_de_entrada();
	puntero = primero;

	if (!primero)
		primero = nuevo;
	else
	{
		while(puntero->siguiente and (strcmp(puntero->siguiente-
>cliente.get_identificador_de_entrada(), identificador_cliente) < 0))
			puntero = puntero->siguiente;

		if(!puntero->siguiente)
			if(strcmp(puntero->cliente.get_identificador_de_entrada(),
identificador_cliente) > 0)
				insertar_detras(nuevo);
			else
				insertar_primero(nuevo);

		else
			if(strcmp(puntero->cliente.get_identificador_de_entrada(),
identificador_cliente) > 0)
				insertar_detras(nuevo);
			else
				insertar_delante(nuevo);
	}

	clientes_en_lista ++;
}

void Lista::insertar_delante(pnodoLista n)
{
	pnodoLista aux;

	aux = puntero->siguiente;
	puntero->siguiente = n;
	n->siguiente = aux;
	n->anterior = puntero;
}
```

```
void Lista::insertar_detras(pnodoLista n)
{
	pnodoLista aux;

	aux = puntero->anterior;
	puntero->anterior = n;
	n->anterior = aux;
	n->siguiente = puntero;
	primero = n;
}

void Lista::insertar_primero(pnodoLista n)
{
	puntero->siguiente = n;
	n->anterior = puntero;
}

Cliente Lista::eliminar()
{
	pnodoLista nodo;
	Cliente c;
	nodo = primero;

	primero = nodo -> siguiente;
	c = nodo -> cliente;
	delete nodo;

	clientes_en_lista --;

	return c;
}

void Lista::vaciar()
{
	while (primero)
	{
		eliminar();
	}
}

bool Lista::es_vacia()
{
	return primero;
}

void Lista::mostrar()
{
	pnodoLista aux = primero;
	while(aux)
	{
		aux -> cliente.show_cliente_completo();
```

```
                aux = aux -> siguiente;
        }
}

int Lista::cantidad_de_clientes()
{
        return clientes_en_lista;
}

void Lista::duplicar(Cliente* c, int longitud)
{
        puntero = primero;

        for (int x = 0; x < longitud; x++)
        {
                c[x] = puntero->cliente;
                puntero = puntero->siguiente;
        }
}

Lista::~Lista()
{
        vaciar();
}
```

NodoCola.hpp

```
#ifndef NODO_HPP
#define NODO_HPP

#include "Cliente.hpp"

class NodoCola
{
	public:
		NodoCola(Cliente c, NodoCola *sig = NULL);
		~NodoCola();
	private:
		Cliente cliente;
		NodoCola *siguiente;

		friend class Cola;

};

typedef NodoCola* pnodoCola;

#endif // NODO_HPP
```

NodoCola.cpp

```
#include "NodoCola.hpp"

NodoCola::NodoCola(Cliente c, NodoCola* sig)
{
	cliente = c;
	siguiente = sig;
}

NodoCola::~NodoCola()
{
}
```

Cola.hpp

```
#ifndef COLA_HPP
#define COLA_HPP
#include "NodoCola.hpp"

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

	void insertar (Cliente c);
	Cliente eliminar ();
	void mostrar ();
	bool es_vacia ();
	int cantidad_de_clientes();
	Cliente ver_primero (void);
	void vaciar();

private:
	pnodoCola primero, ultimo;
	int clientes_en_cola;
};

#endif // COLA_HPP
```

Cola.cpp

```
#include "Cola.hpp"

Cola::Cola()
{
	primero = NULL;
	ultimo = NULL;
	clientes_en_cola = 0;
}

void Cola::insertar(Cliente c)
{
	pnodoCola nuevo = new NodoCola(c);
	if (ultimo)
		ultimo -> siguiente = nuevo;
	ultimo = nuevo;
	if (!primero)
		primero = nuevo;
	clientes_en_cola ++;
}

bool Cola::es_vacia()
{
	return primero;
}

Cliente Cola::eliminar()
{
	pnodoCola nodo;
	Cliente c;
	nodo = primero;

	primero = nodo -> siguiente;
	c = nodo -> cliente;
	delete nodo;

	if (!primero)
		ultimo = NULL;

	clientes_en_cola --;

	return c;
}

void Cola::vaciar()
{
	while (primero)
	{
		eliminar();
	}
}
```

```
Cliente Cola::ver_primero()
{
	return primero->cliente;
}

void Cola::mostrar()
{
	pnodoCola aux = primero;
	while(aux)
	{
		aux -> cliente.show_cliente();
		aux = aux -> siguiente;
	}
}

int Cola::cantidad_de_clientes()
{
	return clientes_en_cola;
}

Cola::~Cola()
{
	vaciar();
}
```

NodoArbol.hpp

```
#ifndef NODOARBOL_HPP
#define NODOARBOL_HPP

#import "Cliente.hpp"

class NodoArbol
{
	public:
		NodoArbol(Cliente c, NodoArbol* izq = NULL, NodoArbol* der = NULL);
		~NodoArbol();

	private:
		Cliente cliente;
		NodoArbol *izquierda;
		NodoArbol *derecha;

		friend class Arbol;

};

typedef NodoArbol* pnodoArbol;

#endif // NODOARBOL_HPP
```

NodoArbol.cpp

```
#include "NodoArbol.hpp"

NodoArbol::NodoArbol(Cliente c, NodoArbol* izq, NodoArbol* der)
{
	cliente = c;
	izquierda = izq;
	derecha = der;
}

NodoArbol::~NodoArbol()
{
}
```

Arbol.hpp

```
#ifndef ARBOL_HPP
#define ARBOL_HPP

#include "NodoArbol.hpp"

class Arbol
{
	public:
		Arbol(pnodoArbol r = NULL);
		~Arbol();
		void generar(Cliente * c, int longitud);
		void vaciar();
		bool es_vacio();
		Cliente buscar(char* DNI);
		void show_pre_orden();
		void show_post_orden();
		void show_in_orden();
		void draw();

	private:
		pnodoArbol raiz;

		void generar_recursivo(pnodoArbol nuevoNodo, char* nuevoDNI, pnodoArbol
arbol);
		void draw_recursivo(pnodoArbol arbol, string indent, bool is_tail);

		void eliminar_recursivo_post_orden(pnodoArbol nodo);

		Cliente buscar_recursivo(pnodoArbol arbol, char* DNI);
		void show_pre_orden_recursivo(pnodoArbol arbol);
		void show_post_orden_recursivo(pnodoArbol arbol);
		void show_in_orden_recursivo(pnodoArbol arbol);
};

#endif // ARBOL_HPP
```

Arbol.cpp

```
#include "Arbol.hpp"

Arbol::Arbol(pnodoArbol r)
{
	raiz = r;
}

void Arbol::generar(Cliente * c, int longitud)
{
	pnodoArbol nuevoNodo = NULL;
	char* nuevoDNI;
	for (int x = 0; x < longitud; x++)
	{
		nuevoNodo = new NodoArbol(c[x]);
		nuevoDNI = c[x].get_DNI();

		generar_recursivo(nuevoNodo,nuevoDNI,raiz);
	}

}

void Arbol::generar_recursivo(pnodoArbol nuevoNodo, char* nuevoDNI, pnodoArbol arbol)
{
	if (strcmp(nuevoDNI, arbol->cliente.get_DNI()) <= 0)
		if (arbol->izquierda)
			generar_recursivo(nuevoNodo,nuevoDNI,arbol->izquierda);
		else
			arbol->izquierda = nuevoNodo;
	else
		if (arbol->derecha)
			generar_recursivo(nuevoNodo,nuevoDNI,arbol->derecha);
		else
			arbol->derecha = nuevoNodo;
}

Cliente Arbol::buscar(char* DNI)
{
	return buscar_recursivo(raiz,DNI);
}

Cliente Arbol::buscar_recursivo(pnodoArbol arbol, char* DNI)
{
	if (arbol == nullptr)
		return Cliente(-1,1);
	else
	{
		if (!strcmp(arbol->cliente.get_DNI(),DNI))
		{
			return arbol->cliente;
```

```
                }
                else if (strcmp(arbol->cliente.get_DNI(),DNI) > 0)
                        return buscar_recursivo(arbol->izquierda,DNI);
                else
                        return buscar_recursivo(arbol->derecha,DNI);
        }
        return Cliente(-1,1);
}

void Arbol::draw()
{
        cout << endl << endl;
        draw_recursivo(raiz,"\t\t",true);
        cout << endl << endl;
}

void Arbol::draw_recursivo(pnodoArbol arbol, string indent, bool is_tail)
{
        Colores C;
        if (arbol == raiz)
        {
                C.colorear(ROJO);
                cout << "\t(DNI: 49999999-M)"<< endl << indent << "│   "<< endl;
                C.blanco();
        }
        else
        {
                C.colorear(AMARILLO);
                cout << indent << (is_tail ? "└─": "├─");
                C.blanco();
                if (arbol == nullptr)
                {
                        C.colorear(ROJO);
                        cout << "*" << endl;
                        C.blanco();
                        return;
                }
                else
                {
                        C.colorear(GRIS);
                        arbol->cliente.show_cliente_completo_formato_reducido();
                        C.blanco();
                }
        }

        if (arbol->izquierda != nullptr or arbol->derecha != nullptr)
        {
                if (arbol!= raiz)
                        indent.append(is_tail ? "   " : "│   ");
                draw_recursivo(arbol->derecha, indent, false);
                if (arbol->izquierda)
                        draw_recursivo(arbol->izquierda, indent, true);
```

```
                else
                        draw_recursivo(arbol->izquierda, indent, false);
        }
}

void Arbol::show_pre_orden()
{
        cout << endl << endl;
        show_pre_orden_recursivo(raiz);
        cout << endl << endl;
}

void Arbol::show_pre_orden_recursivo(pnodoArbol arbol)
{
        if (arbol != NULL)
        {
                if (arbol == raiz)
                {
                        Colores C;
                        C.colorear(MORADO);
                        cout << "\t\tRAIZ_SIMBOLICA" << endl;
                        C.colorear(GRIS);
                }
                else
                        arbol->cliente.show_cliente_completo();
                show_pre_orden_recursivo(arbol->izquierda);
                show_pre_orden_recursivo(arbol->derecha);
        }
}

void Arbol::show_post_orden()
{
        cout << endl << endl;
        show_post_orden_recursivo(raiz);
        cout << endl << endl;
}

void Arbol::show_post_orden_recursivo(pnodoArbol arbol)
{
        if (arbol != NULL)
        {
                show_post_orden_recursivo(arbol->izquierda);
                show_post_orden_recursivo(arbol->derecha);
                if (arbol == raiz)
                {
                        Colores C;
                        C.colorear(MORADO);
                        cout << "\t\tRAIZ_SIMBOLICA" << endl;
                        C.colorear(GRIS);
                }
                else
                        arbol->cliente.show_cliente_completo();
```

```
	}
}

void Arbol::show_in_orden()
{
	cout << endl << endl;
	show_in_orden_recursivo(raiz);
	cout << endl << endl;
}

void Arbol::show_in_orden_recursivo(pnodoArbol arbol)
{
	if (arbol != NULL)
	{
		show_in_orden_recursivo(arbol->izquierda);
		if (arbol == raiz)
		{
			Colores C;
			C.colorear(MORADO);
			cout << "\t\tRAIZ_SIMBOLICA" << endl;
			C.colorear(GRIS);
		}
		else
			arbol->cliente.show_cliente_completo();
		show_in_orden_recursivo(arbol->derecha);
	}
}

void Arbol::vaciar()
{
	eliminar_recursivo_post_orden(raiz);
}

void Arbol::eliminar_recursivo_post_orden(pnodoArbol nodo)
{
	if (nodo)
	{
		eliminar_recursivo_post_orden(nodo->izquierda);
		eliminar_recursivo_post_orden(nodo->derecha);
		delete nodo;
		nodo = NULL;
	}
}

Arbol::~Arbol()
{
	vaciar();
}
```

Colores.hpp

```
#ifndef COLORES_HPP
#define COLORES_HPP

#include <iostream>
#define WINDOWS 0
#if WINDOWS
#include "windows.h"
#endif

#define ROJO 'R'
#define VERDE 'V'
#define AMARILLO 'A'
#define AZUL 'Z'
#define GRIS 'G'
#define MORADO 'M'
#define BLANCO 'B'

using namespace std;

class Colores
{
	public:
		Colores();
		~Colores();
		void colorear(char color);
		void blanco(void);
	private:
		#if WINDOWS
			HANDLE hConsole;
		#endif
};

#endif // COLORES_HPP
```

Colores.cpp

```
#include "Colores.hpp"

Colores::Colores()
{
	#if WINDOWS
	hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
	#endif
}

Colores::~Colores()
{
}

void Colores::colorear (char color)
{
	#if WINDOWS
	switch(color)
	{
		case ROJO:
			SetConsoleTextAttribute(hConsole,12);
		break;
		case VERDE:
			SetConsoleTextAttribute(hConsole,10);
		break;
		case AZUL:
			SetConsoleTextAttribute(hConsole,9);
		break;
		case AMARILLO:
			SetConsoleTextAttribute(hConsole,14);
		break;
		case GRIS:
			SetConsoleTextAttribute(hConsole,8);
		break;
		case MORADO:
			SetConsoleTextAttribute(hConsole,5);
		break;
	}
	#else
	switch(color)
	{
		case ROJO:
			cout << "\033[91m";
		break;
		case VERDE:
			cout << "\033[92m";
		break;
		case AZUL:
			cout << "\033[94m";
		break;
		case AMARILLO:
```

```
                        cout << "\033[93m";
                break;
                case GRIS:
                        cout << "\033[37m";
                break;
                case MORADO:
                        cout << "\033[35m";
                break;
        }
        #endif
}

void Colores::blanco ()
{
        #if WINDOWS
                SetConsoleTextAttribute(hConsole,15);
        #else
                cout << "\033[0m";
        #endif
}
```

Cliente.hpp

```
#ifndef CLIENTE_HPP
#define CLIENTE_HPP

#define HORA_DE_INICIO_NO_REGISTRADOS 120
#define TIEMPO_MAXIMO_DE_COMPRA 10
#define LEN_IDENTIFICADOR 5
#define LEN_DNI 11
#define DNI_FIN 10
#define DNI_LETRA 9
#define DNI_GUION 8

#include "Colores.hpp"
#include <ctime>
#include <math.h>
#include <stdlib.h>
#include <string>

class Cliente
{
	public:

		Cliente();
		Cliente(int t,bool r);
		~Cliente();

		void show_cliente();
		void show_cliente_completo();
		void show_cliente_completo_formato_reducido();
		void generar_identificador_de_entrada();
		char* get_identificador_de_entrada();
		int get_hora_de_inicio_de_compra();
		int get_tiempo_de_compra();
		char* get_DNI();

	private:
		char DNI [LEN_DNI];
		int hora_de_inicio_de_compra;
		int tiempo_de_compra;
		bool registrado;
		char identificador_de_entrada [LEN_IDENTIFICADOR];

		//Identificador de entrada
		void show_identificador_de_entrada();

		//DNI
		void generar_DNI();
		void show_DNI();
		void set_DNI();

		//Hora de inicio de compra
```

```
            void generar_hora_de_inicio_de_compra();
            void show_hora_de_inicio_de_compra();

            //Tiempo de compra
            void generar_tiempo_de_compra();
            void show_tiempo_de_compra();

            void show_tipo();
};

#endif // CLIENTE_HPP
```

Cliente.cpp

```
#include "Cliente.hpp"

Cliente::Cliente()
{
}

Cliente::Cliente(int t, bool r)
{
	this->hora_de_inicio_de_compra = t;
	this->registrado = r;

	if (t >= 0)
		generar_DNI();
	else
		set_DNI();
	generar_hora_de_inicio_de_compra();
	generar_tiempo_de_compra();
}

//DNI
void Cliente::set_DNI()
{
	char dni [LEN_DNI] = {'4','9','9','9','9','9','9','9','-','M'};
	for (int x = 0; x < LEN_DNI; x++)
		DNI[x] = dni [x];
}
void Cliente::generar_DNI()
{
	char letra [23] =
{'T','R','W','A','G','M','Y','F','P','D','X','B','N','J','Z','S','Q','V','H','L','C','K','E'};
	long dni = 0;


	for (int x = 0; x < (DNI_GUION); x++)
	{
		DNI[DNI_GUION-1-x]=(rand()%10);
		dni+=DNI[DNI_GUION-1-x]*pow(10,x);
		DNI[DNI_GUION-1-x] += 48;
	}
	dni %=23;
	DNI[DNI_GUION] = '-';
	DNI[DNI_LETRA] = letra [dni];
	DNI[DNI_FIN] = '\0';
}
void Cliente::show_DNI()
{
	cout << DNI;
}

//HORA DE INICIO DE COMPRA
```

```
void Cliente::generar_hora_de_inicio_de_compra()
{
	if (!registrado)
		hora_de_inicio_de_compra += HORA_DE_INICIO_NO_REGISTRADOS;
}
void Cliente::show_hora_de_inicio_de_compra()
{
	int hora = hora_de_inicio_de_compra/60, minutos = hora_de_inicio_de_compra%60;
	if (hora < 10)
		cout << '0';
	cout << hora << ':';
	if (minutos < 10)
		cout << '0';
	 cout << minutos;
}
int Cliente::get_hora_de_inicio_de_compra()
{
	return hora_de_inicio_de_compra;
}

//TIEMPO DE COMPRA
void Cliente::generar_tiempo_de_compra()
{
	tiempo_de_compra = 1 + rand()%TIEMPO_MAXIMO_DE_COMPRA;
}
void Cliente::show_tiempo_de_compra()
{
	cout << tiempo_de_compra;
	if (tiempo_de_compra < 10)
		cout << " ";
}
int Cliente::get_tiempo_de_compra()
{
	return tiempo_de_compra;
}

void Cliente::show_tipo()
{
	if (registrado)
		cout << "VIP";
	else
		cout << "NO REGISTRADO";
}

//CLIENTE
void Cliente::show_cliente()
{
	Colores C;
	C.colorear(GRIS);
	cout << "\t\tDNI: ";
	C.blanco();
	show_DNI();
```

```
	C.colorear(MORADO);
	cout << "    -    ";
	C.colorear(GRIS);
	cout << "Hora de inicio de compra: ";
	C.colorear(AMARILLO);
	show_hora_de_inicio_de_compra();
	C.colorear(MORADO);
	cout << "    -    ";
	C.colorear(GRIS);
	cout << "Tiempo de compra: ";
	C.colorear(AMARILLO);
	show_tiempo_de_compra();
	C.colorear(MORADO);
	cout << "    -    ";
	C.colorear(ROJO);
	show_tipo();
	C.blanco();
	cout << endl;
}

//IDENTIFICADOR DE ENTRADA
void Cliente::generar_identificador_de_entrada()
{
	for(int x = 0; x < (LEN_IDENTIFICADOR-1); x++)
		identificador_de_entrada[x] = rand()%26 + 65;

	identificador_de_entrada[LEN_IDENTIFICADOR-1] = '\0';
}
void Cliente::show_identificador_de_entrada()
{
	cout << identificador_de_entrada;
}

char* Cliente::get_identificador_de_entrada()
{
	return identificador_de_entrada;
}

void Cliente::show_cliente_completo()
{
	Colores C;
	C.colorear(GRIS);
	cout << "\t\tDNI: ";
	C.blanco();
	show_DNI();
	C.colorear(MORADO);
	cout << "    -    ";
	C.colorear(GRIS);
	cout << "Hora de inicio de compra: ";
	C.colorear(AMARILLO);
	show_hora_de_inicio_de_compra();
	C.colorear(MORADO);
```

```
	cout << "    -    ";
	C.colorear(GRIS);
	cout << "Tiempo de compra: ";
	C.colorear(AMARILLO);
	show_tiempo_de_compra();
	C.colorear(MORADO);
	cout << "    -    ";
	C.colorear(GRIS);
	cout << "Identificador de entrada: ";
	C.colorear(VERDE);
	show_identificador_de_entrada();
	C.colorear(MORADO);
	cout << "    -    ";
	C.colorear(ROJO);
	show_tipo();
	C.blanco();
	cout << endl;
}

void Cliente::show_cliente_completo_formato_reducido()
{
	Colores C;
	C.colorear(AMARILLO);
	cout << "DNI: ";
	C.blanco();
	show_DNI();
	C.colorear(MORADO);
	cout << "    -    ";
	C.colorear(GRIS);
	cout << "Hora de inicio de compra: ";
	C.colorear(AMARILLO);
	show_hora_de_inicio_de_compra();
	C.colorear(MORADO);
	cout << "    -    ";
	C.colorear(GRIS);
	cout << "Tiempo de compra: ";
	C.colorear(AMARILLO);
	show_tiempo_de_compra();
	C.colorear(MORADO);
	cout << "    -    ";
	C.colorear(GRIS);
	cout << "Identificador de entrada: ";
	C.colorear(VERDE);
	show_identificador_de_entrada();
	C.colorear(MORADO);
	cout << "    -    ";
	C.colorear(ROJO);
	show_tipo();
	C.blanco();
	cout << endl;
}
```

```
char* Cliente::get_DNI()
{
        return DNI;
}

Cliente::~Cliente()
{

}
```

Gestor.hpp

```
#ifndef GESTOR_HPP
#define GESTOR_HPP

#include "Cola.hpp"
#include "Lista.hpp"
#include "Arbol.hpp"

#define DIFERENCIA_LLEGADA 5
#define VIP 10
#define NO_REGISTRADOS 25
#define SERVIDORES_NO_REGISTRADOS 4

#if WINDOWS
	#define TIEMPO 1000
#else
	#define TIEMPO 400000
	#include <unistd.h>
#endif

class Gestor
{
	public:
		Gestor();
		~Gestor();
		void generar_solicitudes_de_clientes_VIP(void);
		void mostrar_clientes_VIP(void);
		void borrar_clientes_VIP(bool texto);
		void generar_solicitudes_de_clientes_no_registrados(void);
		void mostrar_clientes_no_registrados(void);
		void borrar_clientes_no_registrados(bool texto);
		void simular_venta_a_clientes_VIP(void);
		void simular_venta_a_clientes_no_registrados(void);
		void mostrar_entradas_vendidas(void);
		void generar_arbol(void);
		void buscar_cliente(void);
		void mostrar_arbol_en_pre_orden(void);
		void mostrar_arbol_en_post_orden(void);
		void mostrar_arbol_en_in_orden(void);
		void dibujar_arbol(void);
		void reiniciar(void);

		int cantidad_no_registrados();
		int recuento_VIP();
		int cantidad_no_registrados_total();
		int recuento_VIP_total();
		int recuento_entradas();
		bool estado_arbol();

	private:
		int vip;
```

```
        int no_registrados;
        Cola VIP_en_espera;
        Cola servidor_VIP;

        Cola no_registrado_en_espera;
        Cola servidor_no_registrado[SERVIDORES_NO_REGISTRADOS];

        Lista clientes_terminados;
        Arbol *arbol;

        void pedir_DNI_validado(char*);
        bool validar_formato(char*);
        bool validar_numeros(char*);
        bool validar_letra(char*);
        void show_hora(int t);
        int recuento_no_registrados();
        void actualizar_pantalla_VIP(int tiempo, int tiempo_en_proceso);
        void actualizar_pantalla_no_registrado(int tiempo, int * tiempo_en_proceso);

        Colores C;
};

#endif // GESTOR_HPP
```

Gestor.cpp

```
#include "Gestor.hpp"

Gestor::Gestor()
{
	arbol = NULL;
	vip = 0;
	no_registrados = 0;
}

Gestor::~Gestor()
{
}

void Gestor::generar_solicitudes_de_clientes_VIP()
{
	int tiempo_cliente_anterior = 0;
	if (recuento_VIP() == 0)
		vip+=VIP;

	borrar_clientes_VIP(false);

	for(int x = 0; x < VIP; x++)
	{
		VIP_en_espera.insertar(Cliente (tiempo_cliente_anterior,true));
		tiempo_cliente_anterior += rand()%6;
	}

	C.colorear(VERDE);
	cout << endl << "\tClientes VIP generados!" << endl << endl;
	C.blanco();
}

void Gestor::mostrar_clientes_VIP()
{
	if (VIP_en_espera.es_vacia())
	{
		C.colorear(VERDE);
      cout << endl<< "\tClientes VIP: " << "\t\t\t"<< VIP_en_espera.cantidad_de_clientes() <<
endl << endl;
		C.colorear(GRIS);
		C.colorear(GRIS);
		VIP_en_espera.mostrar();
		C.blanco();
	}
	else
	{
		C.colorear(ROJO);
		cout << endl << "\tNo hay Clientes VIP en espera!" << endl << endl;
		C.blanco();
  }
```

```
}

void Gestor::borrar_clientes_VIP(bool texto)
{
	if (recuento_VIP() == VIP)
		vip -= VIP;

	VIP_en_espera.vaciar();

	if (texto)
	{
		C.colorear(MORADO);
		cout << endl << "\tClientes VIP borrados!" << endl << endl;
		C.blanco();
	}
}

void Gestor::generar_solicitudes_de_clientes_no_registrados()
{
	int tiempo_cliente_anterior = 0;
	if (cantidad_no_registrados() == 0)
		no_registrados += NO_REGISTRADOS;

	borrar_clientes_no_registrados(false);

	for(int x = 0; x < NO_REGISTRADOS; x++)
	{
		no_registrado_en_espera.insertar(Cliente (tiempo_cliente_anterior,false));
		tiempo_cliente_anterior += rand()%6;
	}

	C.colorear(VERDE);
	cout << endl << "\tClientes NO REGISTRADOS generados!" << endl << endl;
	C.blanco();
}

void Gestor::mostrar_clientes_no_registrados()
{
	if (no_registrado_en_espera.es_vacia())
  {
		C.colorear(VERDE);
     cout << endl<< "\tClientes NO REGISTRADOS: " << "\t\t\t"<<
no_registrado_en_espera.cantidad_de_clientes() << endl << endl;
		C.colorear(GRIS);
		no_registrado_en_espera.mostrar();
		C.blanco();
  }
	else
  {
		C.colorear(ROJO);
		cout << endl << "\tNo hay Clientes NO REGISTRADOS en espera!" << endl <<
endl;
```

```
                C.blanco();
    }
}

void Gestor::borrar_clientes_no_registrados(bool texto)
{
        if (cantidad_no_registrados() == NO_REGISTRADOS)
                no_registrados -= NO_REGISTRADOS;

        no_registrado_en_espera.vaciar();

        if (texto)
        {
                C.colorear(MORADO);
                cout << endl << "\tClientes NO REGISTRADOS borrados!" << endl << endl;
                C.blanco();
    }
}

void Gestor::simular_venta_a_clientes_VIP()
{
        int tiempo = -1;
        int tiempo_en_proceso = 0;

        if(VIP_en_espera.cantidad_de_clientes()>0)
        {
                Cliente c;

                do
                {
                        if (servidor_VIP.cantidad_de_clientes()>0)
                        {
                                if (servidor_VIP.ver_primero().get_tiempo_de_compra() ==
(tiempo_en_proceso + 1))
                                {
                                        tiempo_en_proceso = 0;
                                        c = servidor_VIP.eliminar();
                                        c.generar_identificador_de_entrada();
                                        clientes_terminados.insertar(c);
                                }
                                else
                                        tiempo_en_proceso++;
                        }

                        while((VIP_en_espera.cantidad_de_clientes() > 0) and
(VIP_en_espera.ver_primero().get_hora_de_inicio_de_compra() == tiempo))
                                servidor_VIP.insertar(VIP_en_espera.eliminar());

                        actualizar_pantalla_VIP(tiempo, tiempo_en_proceso);

                        tiempo ++;
```

```
                        #if WINDOWS
                                Sleep(TIEMPO);
                                system("cls");
                        #else
                                usleep(TIEMPO);
                                cout << "\033c";
                        #endif

                }while(servidor_VIP.cantidad_de_clientes() > 0 or
VIP_en_espera.cantidad_de_clientes() > 0);

                C.colorear(VERDE);
                cout << "\t\tVENTA DE ENTRADAS VIP TERMINADA" << endl;
                actualizar_pantalla_VIP(tiempo, tiempo_en_proceso);
                C.blanco();
        }
        else
   {
                C.colorear(ROJO);
                cout << endl << "\tNo hay clientes VIP" << endl << endl;
                C.blanco();
   }
}

void Gestor::actualizar_pantalla_VIP(int tiempo, int tiempo_en_proceso)
{
        C.colorear(AMARILLO);
        cout << endl <<"\t\t*****************************" << endl;
        C.colorear(ROJO);
        cout << "\t\t      Tiempo:";
        C.colorear(AMARILLO);
        show_hora(tiempo);
        C.colorear(AMARILLO);
        cout << endl;
        cout << "\t\t*****************************" << endl<<endl;
        cout <<"-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------"<< endl;

        C.colorear(MORADO);
        cout << "\tClientes en espera: " << "\t\t\t";C.colorear(AMARILLO); cout <<
VIP_en_espera.cantidad_de_clientes() << endl << endl;

        C.colorear(GRIS);
        VIP_en_espera.mostrar();

        C.colorear(AMARILLO);
        cout << endl <<"-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------"<< endl;

        C.colorear(AZUL);
        cout << "\tServidor VIP: " << "\t\t\t"<< " Tiempo en proceso:
";C.colorear(AMARILLO);cout <<  tiempo_en_proceso << endl << endl;
```

```
	C.blanco();
	servidor_VIP.mostrar();

	C.colorear(AMARILLO);
	cout << "-------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------"<< endl;

	C.colorear(VERDE);
	cout << "\tEntradas vendidas: "<< "\t\t\t";C.colorear(AMARILLO);cout <<
clientes_terminados.cantidad_de_clientes() << endl << endl;

	C.colorear(GRIS);
	clientes_terminados.mostrar();

	C.colorear(AMARILLO);
	cout <<"-------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------"<< endl;
	C.blanco();
}

void Gestor::simular_venta_a_clientes_no_registrados()
{
	int tiempo = 119;
	int tiempo_en_proceso[SERVIDORES_NO_REGISTRADOS];

	for (int x = 0; x < SERVIDORES_NO_REGISTRADOS; x++)
		tiempo_en_proceso[x] = 0;

	if(no_registrado_en_espera.cantidad_de_clientes()>0)
	{
		Cliente c;

		do
		{
			for (int x = 0; x < SERVIDORES_NO_REGISTRADOS; x++)
			{
				if (servidor_no_registrado[x].cantidad_de_clientes() > 0)
				{
					if
(servidor_no_registrado[x].ver_primero().get_tiempo_de_compra() == (tiempo_en_proceso[x]
+ 1))
					{
						tiempo_en_proceso[x] = 0;
						c = servidor_no_registrado[x].eliminar();
						c.generar_identificador_de_entrada();
						clientes_terminados.insertar(c);
					}
					else
						tiempo_en_proceso[x]++;
				}
			}
```

```
                        while((no_registrado_en_espera.cantidad_de_clientes() > 0) and
(no_registrado_en_espera.ver_primero().get_hora_de_inicio_de_compra() == tiempo))
                        {
                                int mejor_servidor;
                                int menor = NO_REGISTRADOS+1;

                                for(int x = 0; x < SERVIDORES_NO_REGISTRADOS; x ++)
                                {
                                        if (servidor_no_registrado[x].cantidad_de_clientes() <
menor)
                                        {
                                                menor =
servidor_no_registrado[x].cantidad_de_clientes();
                                                mejor_servidor = x;
                                        }
                                }


        servidor_no_registrado[mejor_servidor].insertar(no_registrado_en_espera.eliminar());
                        }

                        actualizar_pantalla_no_registrado(tiempo, tiempo_en_proceso);

                        tiempo ++;

                        #if WINDOWS
                                Sleep(TIEMPO);
                                system("cls");
                        #else
                                usleep(TIEMPO);
                                cout << "\033c";
                        #endif

                }while(recuento_no_registrados() > 0 or
no_registrado_en_espera.cantidad_de_clientes() > 0);

                C.colorear(VERDE);
                cout << "\t\tVENTA DE ENTRADAS NO REGISTRADOS TERMINADA" << endl;
                C.blanco();
     actualizar_pantalla_no_registrado(tiempo, tiempo_en_proceso);
        }
        else
   {
                C.colorear(ROJO);
                cout << endl << "\tNo hay clientes NO_REGISTRADOS" << endl << endl;
                C.blanco();
   }
}

void Gestor::actualizar_pantalla_no_registrado(int tiempo, int * tiempo_en_proceso)
{
```

```
	C.colorear(AMARILLO);
	cout << endl <<"\t\t*****************************" << endl;
	C.colorear(ROJO);
	cout << "\t\t      Tiempo:";
	C.colorear(AMARILLO);
	show_hora(tiempo);
	cout << endl;
	C.colorear(AMARILLO);
	cout << "\t\t*****************************" << endl<<endl;
	cout << endl <<"-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------"<< endl;

	C.colorear(MORADO);
	cout << "\tClientes en espera: " << "\t\t\t"; C.colorear(AMARILLO); cout <<
no_registrado_en_espera.cantidad_de_clientes() << endl << endl;

	C.colorear(GRIS);
	no_registrado_en_espera.mostrar();

	for (int x = 0; x < SERVIDORES_NO_REGISTRADOS; x++)
	{
		C.colorear(AMARILLO);
		cout <<"-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------"<< endl;
		C.colorear(AZUL);
		cout << "\tServidor: "<< x << "\t\t\tTiempo en proceso: ";
C.colorear(AMARILLO); cout << tiempo_en_proceso[x] << endl << endl;

		C.blanco();
		servidor_no_registrado[x].mostrar();
   }
	C.colorear(AMARILLO);
	cout <<"-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------"<< endl;

	C.colorear(VERDE);
	cout << "\tEntradas vendidas: "<< "\t\t\t"; C.colorear(AMARILLO); cout <<
clientes_terminados.cantidad_de_clientes() << endl << endl;

	C.colorear(GRIS);
	clientes_terminados.mostrar();
	C.colorear(AMARILLO);
	cout <<"-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------"<< endl;
}

void Gestor::mostrar_entradas_vendidas()
{
	if(clientes_terminados.cantidad_de_clientes())
	{
		C.colorear(VERDE);
		cout << endl << "\tEntradas vendidas: " << endl << endl;
```

```
                C.colorear(GRIS);
                clientes_terminados.mostrar();
                C.blanco();
        }
        else
        {
                C.colorear(ROJO);
                cout << endl << "\tNo hay ENTRADAS vendidas" << endl;
                C.blanco();
        }
}

void Gestor::generar_arbol()
{
        if (clientes_terminados.es_vacia())
        {
                int numero_clientes = clientes_terminados.cantidad_de_clientes();
                arbol = new Arbol( new NodoArbol (Cliente(-1,0)));

                Cliente c [numero_clientes];
                clientes_terminados.duplicar(c, numero_clientes);
                arbol->generar(c,numero_clientes);

                C.colorear(VERDE);
                cout << endl << "\tArbol generado!" << endl << endl;
                C.blanco();
        }
        else
        {
                C.colorear(ROJO);
                cout << endl << "\tNO HAY ENTRADAS VENDIDAS" << endl << endl;
                C.blanco();
        }
}

void Gestor::buscar_cliente()
{
        if (arbol)
        {
                char DNI [LEN_DNI];
                pedir_DNI_validado(DNI);

                if (!strcmp(DNI,"xxxxxxxxxx"))
                {
                        C.colorear(ROJO);
                        cout << "\t\tDNI ERRONEO!" << endl << endl;
                        C.blanco();
                }
                else
                {
                        Cliente c = arbol->buscar(DNI);
```

```
                        if (c.get_hora_de_inicio_de_compra() < 0)
                        {
                                C.colorear(MORADO);
                                cout << endl<<endl << "\t\tCLIENTE NO EXISTE!" << endl <<
endl;
                                C.blanco();
                        }
                        else
                        {
                                C.colorear(VERDE);
                                cout << endl << "\t\tSu cliente es:" << endl;
                                c.show_cliente_completo();
                                C.blanco();
                                cout << endl << endl;
                        }
                }
        }
        else
        {
                C.colorear(ROJO);
                cout << endl << "\tARBOL VACIO" << endl << endl;
                C.blanco();
        }
}

void Gestor::mostrar_arbol_en_post_orden()
{
        if(arbol)
        {
                C.colorear(VERDE);
                cout << endl << "\t\tPOST_ORDEN:";
                C.colorear(GRIS);
                arbol->show_post_orden();
                C.blanco();
        }
        else
        {
                C.colorear(ROJO);
                cout << endl << "\tARBOL VACIO" << endl << endl;
                C.blanco();
        }
}

void Gestor::mostrar_arbol_en_pre_orden()
{
        if(arbol)
        {
                C.colorear(VERDE);
                cout << endl << "\t\tPRE_ORDEN:";
                C.colorear(GRIS);
                arbol->show_pre_orden();
                C.blanco();
```

```
	}
	else
	{
		C.colorear(ROJO);
		cout << endl << "\tARBOL VACIO" << endl << endl;
		C.blanco();
	}
}

void Gestor::mostrar_arbol_en_in_orden()
{
	if(arbol)
	{
		C.colorear(VERDE);
		cout << endl << "\t\tIN_ORDEN:";
		C.colorear(GRIS);
		arbol->show_in_orden();
		C.blanco();
	}
	else
	{
		C.colorear(ROJO);
		cout << endl << "\tARBOL VACIO" << endl << endl;
		C.blanco();
	}
}

void Gestor::dibujar_arbol()
{
	if(arbol)
	{
		C.colorear(VERDE);
		cout << "\n\t\tARBOL:";
		C.colorear(GRIS);
		arbol->draw();
		C.blanco();
	}
	else
	{
		C.colorear(ROJO);
		cout << endl << "\tARBOL VACIO" << endl << endl;
		C.blanco();
	}
}

void Gestor::reiniciar()
{
	vip = 0;
	no_registrados = 0;
	borrar_clientes_VIP(true);
	borrar_clientes_no_registrados(true);
```

```
	C.colorear(MORADO);
	servidor_VIP.vaciar();

	for (int x = 0; x < SERVIDORES_NO_REGISTRADOS; x++)
		servidor_no_registrado[x].vaciar();

	clientes_terminados.vaciar();


	cout << endl << "\tClientes LISTA DE ENTRADAS borrada!" << endl << endl;

	arbol->vaciar();
	arbol = NULL;
	cout << endl << "\tClientes ARBOL borrado!" << endl << endl;

	C.colorear(VERDE);
	cout << endl << "\tPrograma reiniciado!" << endl << endl;
	C.blanco();
}

int Gestor::recuento_no_registrados()
{
	int acumulador = 0;
	for (int x = 0; x < SERVIDORES_NO_REGISTRADOS; x++)
		acumulador += servidor_no_registrado[x].cantidad_de_clientes();

	return acumulador;
}

void Gestor::show_hora(int t)
{
	int hora = t/60, minutos = t%60;
	if (hora < 10)
		cout << '0';
	cout << hora << ':';
	if (minutos < 10)
		cout << '0';
	 cout << minutos;
}

int Gestor::recuento_VIP()
{
	return VIP_en_espera.cantidad_de_clientes();
}

int Gestor::recuento_entradas()
{
	return clientes_terminados.cantidad_de_clientes();
}

int Gestor::cantidad_no_registrados()
{
```

```
	return no_registrado_en_espera.cantidad_de_clientes();
}

void Gestor::pedir_DNI_validado(char* DNI)
{
	cout << endl << endl << "\t\tIntroduce el DNI que quieres buscar: ";
	cin.ignore();
	cin.getline(DNI,LEN_DNI);

	if (!validar_formato(DNI))
		strcpy(DNI,"xxxxxxxxxx");
}

bool Gestor::validar_formato(char* DNI)
{
	return strlen(DNI) == DNI_FIN and validar_numeros(DNI) and validar_letra(DNI) and
DNI[DNI_GUION] == '-';
}

bool Gestor::validar_letra(char* DNI)
{
	char letra [23] =
{'T','R','W','A','G','M','Y','F','P','D','X','B','N','J','Z','S','Q','V','H','L','C','K','E'};
	long dni = 0;

	for (int x = 0; x < DNI_GUION; x++)
		dni+=(DNI[DNI_GUION-x-1] - 48)*pow(10,x);

	dni %=23;

	return letra[dni] == DNI[DNI_LETRA];
}

bool Gestor::validar_numeros(char* DNI)
{
	bool valido = true;

	for (int x = 0; x < DNI_GUION; x++)
	{
		if(DNI[x] - 48 > 9)
			valido = false;
	}

	return valido;
}

bool Gestor::estado_arbol()
{
	return arbol;
}

int Gestor::recuento_VIP_total()
```

```
{
	return vip;
}

int Gestor::cantidad_no_registrados_total()
{
	return no_registrados;
}
```

main.cpp

```
#include <iomanip>
#include "Gestor.hpp"


int main(int argc, char **argv)
{
	Gestor gestor;
	char opcion;
	Colores C;

	#if WINDOWS
			system("cls");
	#else
		cout << "\033c";
	#endif

	#if WINDOWS
		ShowWindow(GetConsoleWindow(),SW_MAXIMIZE);
	#else
	cout << "\e[9;1t";
	#endif

	C.colorear(ROJO);
	cout << "LOADING";
	#if WINDOWS
		Sleep(TIEMPO/2);
	#else
		usleep(TIEMPO/2);
	#endif
	cout << ".";
	#if WINDOWS
		Sleep(TIEMPO/2);
	#else
		usleep(TIEMPO/2);
	#endif
	cout << ".";
	#if WINDOWS
		Sleep(TIEMPO/2);
	#else
		usleep(TIEMPO/2);
	#endif
	cout << ".";
	#if WINDOWS
		Sleep(TIEMPO/2);
	#else
		usleep(TIEMPO/2);
	#endif

	srand(time(NULL));
```

```
	#if WINDOWS
			system("cls");
	#else
		cout << "\033c";
	#endif

	do
	{
		C.colorear(AMARILLO);
		cout << endl << endl;
		cout << "\t-------------------------------------------------------------------------------------
-----\t\t\t" << endl;
		cout << "\t                              MENU                                  \t\t\t" << endl;
		cout << "\t-------------------------------------------------------------------------------------
-----\t\t\t" << endl;

		C.colorear(VERDE);

		cout << "\tNumero de clientes VIP: "; C.colorear(MORADO); cout << setw(4)<<
gestor.recuento_VIP_total() << endl;C.colorear(VERDE);
		cout << "\tNumero de clientes VIP EN ESPERA: "; C.colorear(MORADO); cout <<
setw(4)<< gestor.recuento_VIP() << endl;C.colorear(VERDE);
		cout << "\tNumero de clientes no registrados: " ;C.colorear(MORADO); cout <<
setw(4)<< gestor.cantidad_no_registrados_total() << endl;C.colorear(VERDE);
		cout << "\tNumero de clientes no registrados EN ESPERA:
" ;C.colorear(MORADO); cout << setw(4)<< gestor.cantidad_no_registrados() <<
endl;C.colorear(VERDE);
		cout << "\tEntradas vendidas: "; C.colorear(MORADO); cout << setw(4) <<
gestor.recuento_entradas() << endl; C.colorear(VERDE);
		cout << "\tLista: ";
		if (gestor.recuento_entradas())
			cout << "OK" << endl;
		else
		{
			C.colorear(ROJO);
			cout << "OFF" << endl;
			C.colorear(VERDE);
		}
		cout << "\tArbol: ";
		if (gestor.estado_arbol())
			cout << "OK" << endl << endl;
		else
		{
			C.colorear(ROJO);
			cout << "OFF" << endl << endl;
		}
		C.colorear(AMARILLO);

		cout << "\t\t A. ";C.colorear(GRIS); cout << "Generar solicitudes de entrada de
los clientes VIP." << endl;C.colorear(AMARILLO);
		cout << "\t\t B. ";C.colorear(GRIS); cout << "Mostrar la cola de solicitudes en
espera de los clientes VIP." << endl;C.colorear(AMARILLO);
```

```
		cout << "\t\t C. ";C.colorear(GRIS); cout << "Borrar la cola de solicitues en espera de los clientes VIP." << endl;C.colorear(AMARILLO);
		cout << "\t\t D. ";C.colorear(GRIS); cout << "Generar solicitudes de entrada de los clientes no registrados." << endl;C.colorear(AMARILLO);
		cout << "\t\t E. ";C.colorear(GRIS); cout << "Mostrar la cola de solicitudes en espera de los clientes no registrados." << endl;C.colorear(AMARILLO);
		cout << "\t\t F. ";C.colorear(GRIS); cout << "Borrar la cola de solicitues en espera de los clientes no registrados." << endl;C.colorear(AMARILLO);
		cout << "\t\t G. ";C.colorear(GRIS); cout << "Simular el proceso de compra de los clientes VIP." << endl;C.colorear(AMARILLO);
		cout << "\t\t H. ";C.colorear(GRIS); cout << "Simular el proceso de compra de los clientes no registrados." << endl;C.colorear(AMARILLO);
		cout << "\t\t I. ";C.colorear(GRIS); cout << "Mostrar la lista de entradas vendidas." << endl;C.colorear(AMARILLO);
		cout << "\t\t J. ";C.colorear(GRIS); cout << "Reiniciar el programa." << endl << endl;C.colorear(AMARILLO);

		cout << "\t\t K. ";C.colorear(GRIS); cout << "Crear un arbol binario de busqueda ordenado por el DNI." << endl;C.colorear(AMARILLO);
		cout << "\t\t L. ";C.colorear(GRIS); cout << "Buscar un cliente en el ABB, dado su DNI y mostrar sus datos." << endl;C.colorear(AMARILLO);
		cout << "\t\t M. ";C.colorear(GRIS); cout << "Mostrar el ABB en pre-orden." << endl;C.colorear(AMARILLO);
		cout << "\t\t N. ";C.colorear(GRIS); cout << "Mostrar el ABB en post-orden." << endl;C.colorear(AMARILLO);
		cout << "\t\t O. ";C.colorear(GRIS); cout << "Mostrar el ABB en in-orden." << endl;C.colorear(AMARILLO);
		cout << "\t\t P. ";C.colorear(GRIS); cout << "Dibujar el ABB." << endl << endl;C.colorear(AMARILLO);

		cout << "\t\t S. ";C.colorear(ROJO); cout << "Salir del programa" << endl << endl;C.colorear(VERDE);

		cout << "\tIndique la opcion deseada: ";
		C.colorear(AMARILLO);

		cin >> opcion ;
		opcion = toupper(opcion);

		#if WINDOWS
			system("cls");
		#else
			cout << "\033c";
		#endif

		switch(opcion)
		{
			case 'A':
				gestor.generar_solicitudes_de_clientes_VIP();
			break;
			case 'B':
				gestor.mostrar_clientes_VIP();
```

```
break;
case 'C':
        gestor.borrar_clientes_VIP(true);
break;
case 'D':
        gestor.generar_solicitudes_de_clientes_no_registrados();
break;
case 'E':
        gestor.mostrar_clientes_no_registrados();
break;
case 'F':
        gestor.borrar_clientes_no_registrados(true);
break;
case 'G':
        gestor.simular_venta_a_clientes_VIP();
break;
case 'H':
        gestor.simular_venta_a_clientes_no_registrados();
break;
case 'I':
        gestor.mostrar_entradas_vendidas();
break;
case 'J':
        gestor.reiniciar();
break;
case 'K':
        gestor.generar_arbol();
break;
case 'L':
        gestor.buscar_cliente();
break;
case 'M':
        gestor.mostrar_arbol_en_pre_orden();
break;
case 'N':
        gestor.mostrar_arbol_en_post_orden();
break;
case 'O':
        gestor.mostrar_arbol_en_in_orden();
break;
case 'P':
        gestor.dibujar_arbol();
break;
case 'S':
        C.colorear(MORADO);
        cout << "Saliendo del programa..." << endl << endl;
        C.blanco();
break;
default:
        C.colorear(ROJO);
        cout << "Opcion incorrecta!" << endl << endl;
break;
```

```
                }
        }
        while(opcion != 'S');

        return 0;
}
```