Importante: Los ejercicios deben entregarse a través de web (**Domjudge y Blackboard**). Cada ejercicio deberá ir en un fichero con nombre:

ColaPrioridad.h

Supermercado.h

ListaContigua.h

ListaContigua.cpp

mainSupermercado.cpp

ColaPrirordidad.cpp

Supermercado.cpp

Los cinco primeros ficheros podrán ser descargados a través de **Blackboard** y no deberán modificados por el alumno. Sin embargo, ColaPrioridad.cpp y Supermercado.cpp, deberán ser desarrollados completamente por el alumno.

La fecha de entrega: consultar la página de la actividad en blackboard

Supermercado con prioridad (3 puntos): Se quiere gestionar el funcionamiento de las cajas de un supermercado para poder saber que clientes se encuentran en cada una de las cajas y en caso de cerrar una de ellas recolocar a todos los clientes en las restantes. Para este propósito se deberán implementar dos clases: la clase ColaPrioridad que permite almacenar los clientes que se encuentran esperando en una caja y que se van atendiendo según el orden de llegada y la clase Supermercado que permite gestionar varias cajas al mismo tiempo. La clase ColaPrioridad se implementará mediante un montículo para asegurar que todos los clientes sean servidos según su orden de llegada. Esta clase se compone de los siguientes atributos:

- **ListaContigua vector.** Lista contigua sobre la que se implementará la cola de prioridad (Montículo).

Y será necesario implementar los siguientes métodos:

- **void encolar(int num).** Recibe un número que representará el orden de llegada del cliente al supermercado y lo colocará al final de la cola. Sin embargo al ser una cola de prioridad este elemento irá escalando hasta la posición que le corresponda dentro del Montículo. Es decir, se irá intercambiando con su padre hasta que sea mayor que su padre o hasta que lleguemos a la raíz

- **int desencolar().** Devuelve el número que representa el orden de llegada del cliente que se encuentra en la primera posición del montículo. Al ser una cola de prioridad se deberán seguir los siguientes pasos:

    - El primer elemento será sustituido por el último elemento.

  - El último elemento de la cola será eliminado
  - Se deberá reestructurar el montículo usando el método reestructurar.

- **bool estaVacia().** Indica si la cola se encuentra vacía.
- **void reestructurar().** Después de haber desencolado un elemento, el montículo puede no ser un montículo así que esta función se encarga de reestructurar el montículo para que sea otra vez un montículo.

Por otro lado la clase Supermercado tiene los siguientes atributos:

- **ColaPrioridad *cajas.** Array de punteros a objetos de tipo ColaPrioridad que representan cada una de las cajas..
- **int n_cajas.** Número de cajas que hay en el supermercado

Y será necesario implementar los siguientes métodos::

- **Supermercado(int n)**. Constructor que se encarga de reservar memoria para las n cajas que tiene el supermercado e incializa el atributo n_cajas.
- **void nuevoUsuario(int n,int id).** Encola el usuario con el id indicado en la caja que se encuentra en la posición n del array de cajas.
- **void cerrarCaja(int n).** Esta función simula el cierre de la caja n y el reparto de los usuarios en las cajas restantes. Para ello será necesario desencolar todos los usuarios que se encuentran en la caja n en el orden de llegada, e ir encolándolos en las cajas restantes que no estén vacías. Debido a que los usuarios se deberán repartir de forma equitativa en las cajas restantes, iremos recorriendo las cajas por orden de 0 a N-1 e introduciremos un único usuario por cada caja que no esté vacía. Cuando hayamos introducido un usuario en cada una de las cajas volveremos a empezar desde la caja 0 hasta que no queden más usuarios. Debido al uso de colas de prioridad, los clientes que hayan llegado antes se ordenaran de forma correcta en cada una de las cajas restantes según su orden de llegada.
- **int atenderUsuario(int n)**; Atiende al usuario que se encuentra en la caja n y por tanto lo desencola de la cola que representa dicha caja. Esta función devolverá el id del usuario atendido.
- **bool cajaVacia(int n).** Indica si la caja n tiene o no tiene usuarios esperando.

Utilizar el fichero mainSupermercado.cpp proporcionado a través de Blackboard para realizar las pruebas necesarias y para enviar al corrector automático. Este programa simula el funcionamiento de un supermercado con las siguientes opciones:

- N: Crea un objeto de tipo supermercado que permitirá controlar varias cajas.

- U: Encola un usuario en una de las cajas.
- C: Simula el cierre de una de las cajas repartiendo los usuarios entre las cajas restantes
- A: Atiende al primer usuario que se encuentre en la caja indicada
- S: Termina.

Importante: Los ejercicios deben entregarse a través de web (**Domjudge y Blackboard**). Cada ejercicio deberá ir en un fichero con nombre:

ABB.h

Nodo.h

mainABB.cpp

ABB.cpp

Los tres primeros ficheros podrán ser descargados a través de **Blackboard** y no deberán modificados por el alumno. Sin embargo, ABB.cpp deberá ser desarrollado completamente por el alumno.

Arbol Binario de Búsqueda (ABB) (5 puntos): Desarrollar la clase que implementa un árbol binario de búsqueda que permite almacenar números enteros y realizar operaciones sobre ellos.

La clase **ABB** tiene los siguientes atributos:

- **int n**: atributo que almacena de forma privada el número de nodos del árbol.
- **Nodo *raiz.** Puntero a una variable de tipo struct Nodo que se considerará como la raíz del árbol. Si el puntero apunta a NULL indicará que el árbol está vacío. Esta estructura tiene los siguientes campos:
  - **contenido**. Elemento de tipo entero donde se guardará los números
  - **padre.** Puntero apuntando al padre o a NULL si es la raíz.
  - **hijoIzquierdo**. Apunta al hijo izquierdo, o NULL si no tiene hijo.
  - **hijoDerecho**. Apunta al hijo derecho, o NULL si no tiene hijo.

Para esta clase deberán implementarse los siguientes métodos públicos:

- **Constructor.** Crea un árbol vacío.
- **Destructor.** Se encargará de liberar la memoria que fue reservada de forma dinámica para almacenar todos los nodos del árbol, llamará de forma interna al método recursivo **eliminarSubArbol.**
- **void leerArbol().** Este método lee un árbol a partir de la entrada llamando internamente al método **leerSubarbol**. (Este método no es necesario implementarlo)

- **void imprimir()**. Este método realiza una representación del recorrido en in orden del árbol actual usando el método recursivo **imprimirRecorrido.** (Este método no es necesario implementarlo)
- **void insertar(int nuevoElemento).** Inserta un nuevo elemento en el árbol como una hoja en la posición que le corresponde para ello utiliza internamente el método **buscarHueco.**
- **Nodo *buscar(int elementoABuscar).** Método público que busca un elemento en el árbol binario de búsqueda que coincida con el valor indicado en **elementoABuscar.** Internamente utilizada el método privado **buscarRecursivo**.
- **void eliminar(int elementoAEliminar).** Elimina el elemento del árbol que coincide con el valor indicado por **elementoAEliminar**. Si el elemento se encuentra en una hoja se deberá eliminar únicamente dicho elemento. En el caso de que el nodo sea interno deberá, intercambiarse por otro nodo de la siguiente manera:
  - Si el nodo sólo tiene un hijo izquierdo, se deberá buscar el máximo elemento del subárbol cuyo padre es el hijo izquierdo, para ello se utilizará el método interno **buscarMaximo**.
  - Si el nodo sólo tiene un hijo derecho, se deberá buscar el mínimo elemento del subárbol cuyo padre es el hijo derecho, para ello se utilizará el método interno **buscarMinimo**.
  - Si el nodo tiene ambos hijos, elegiremos el hijo que tenga una mayor altura mediante el método **alturaNodo** y a continuación realizaremos las operaciones indicadas anteriormente.

  Debido a que puede ocurrir que el máximo o el mínimo no sean nodos hojas será necesario realizar un método recursivo **eliminarNodo** que implemente este proceso para ir intercambiando nodos hasta llegar a un nodo hoja.
- **bool esABB().** Indica si el árbol cumple las condiciones para ser un árbol binario de búsqueda. Este método llamará internamente al método recursivo **esSubABB.**
- **bool esAVL().** Indica si el árbol cumple las condiciones para ser un árbol AVL. Este método llamará internamente al método recursivo **esSubAVL.**

Para esta clase deberán implementarse los siguientes métodos privados que implementan diferentes funciones recursivas:

- **void eliminarSubarbol(Nodo *raizSubarbol).** Método que elimina de forma recursiva los nodos que cuelgan de raizSubarbol.
- **Nodo *leerSubarbol(Nodo *padre).** Método que permite crear un árbol de forma recursiva a partir de la entrada. (Este método no es necesario implementarlo).
- **void imprimirRecorrido(Nodo *raizSubarbol).** Método recursivo que representa el árbol mediante su recorrido en in orden (Este método no es necesario implementarlo).

- **Nodo *buscarHueco(Nodo *raizSubarbol, int elementoAInsertar).** Método que busca recursivamente el padre del hueco donde podemos insertar un nuevo elemento dado a partir del subárbol que tiene como raíz **raizSubarbol.**
- **Nodo *buscarRecursivo (Nodo *raizSubarbol, int elementoABuscar).** Método que busca recursivamente el elemento que coincide con **elementoABuscar** a partir del subárbol que tiene como raíz **raizSubarbol.**
- **Nodo *buscarMaximo(Nodo *raizSubarbol).** Método que busca el máximo de un subárbol (ir por la rama derecha hasta llegar a la última hoja). Si no hay ningún hijo derecho más devolverá la raíz.
- **Nodo *buscarMinimo(Nodo *raizSubarbol).** Método que busca el mínimo de un subárbol (ir por la rama izquierda hasta llegar a la última hoja). Si no hay ningún hijo izquierdo más devolverá la raíz.
- **void eliminarNodo(Nodo *nodoParaEliminar).** Este método implementa el proceso indicado anteriormente en eliminar pero de forma recursiva hasta que el nodo eliminado sea una hoja.
- **Int alturaNodo(Nodo *raizSubarbol).** Método que permite calcular la altura de un nodo en un árbol. Si el árbol es vacío se considera que la altura es -1 y si es un nodo hoja 0.
- **bool esSubABB(Nodo *raizSubarbol).** Comprueba de forma recursiva si el subárbol cumple las condiciones para ser un árbol binario de búsqueda.
- **bool esSubAVL(Nodo *raizSubarbol).** Comprueba de forma recursiva si el subárbol cumple las condiciones para ser un árbol de tipo AVL.

Utilizar todos los ficheros proporcionado a través de Blackboard para realizar las pruebas necesarias y para enviar al corrector automático. Este programa permite realizar las siguientes operaciones:

- N: permite crear un nuevo árbol.
- A: permite insertar un valor en el árbol.
- E: permite eliminar un valor del árbol.
- I: imprime el recorrido en in orden del árbol.
- L: lee un árbol por la entrada.
- B: comprueba si es un árbol binario de búsqueda
- V: comprueba si es un árbol AVL
- F: liberar la memoria ocupada por el árbol.
- S: salir.

NOTA: Para cada uno de los métodos implementados se deberá incluir una pequeña descripción de su funcionamiento, sus precondiciones mediante assertdomjudge si las hubiera y el análisis de su complejidad temporal y espacial. Esta información deberá incluirse en la cabecera de cada función.