EI1013/MT1013 Estructuras de datos
Examen final
Viernes, 21 de Junio de 2013

1. **(3 puntos)** Vamos a implementar una tabla de dispersión con resolución de colisiones mediante encadenamiento, con la peculiaridad de que se permitirán elementos con la misma clave. Para ello definimos la siguiente clase genérica:

```
public class ChainHashTable<K,V> {
    private class Node {
        public K key;
        public V value;
        public int count=1;

        // Constructor de un nodo
        Node(K key, V value) {...}
    }

    private int REHASH_THRESHOLD;
    private List<Node>[] table;
    private int size;
    private int unique;

    private void rehash();

    ChainHashTable (int initial_size);
    public int addEntry(K key, V value);
    public int removeEntry(K key);
    public K getEntry(K key();
    public int getCount(K key);
    ...
}
```

   - `K` es el tipo de la clave, y `V` el de los valores asociados.
   - La clase interna `Node` almacena un par clave, valor y en `count` el número de veces que se ha insertado un par con esa clave. El constructor del nodo inicializa la cuenta a 1.
   - `table` contiene las listas de cada posición de la tabla de dispersión. Si en una posición de la tabla no se han insertado elementos, la posición correspondiente tiene el valor `null`.
   - `unique` contiene el número de elementos almacenados en la tabla sin tener en cuenta los repetidos, y `size` el número de elementos total en la teniendo en cuenta los repetidos.
   - Se realiza *rehashing* cuando el valor de `unique` en mayor que el de `REHASH_THRESHOLD`. El tamaño de `table` se dobla y el valor `REHASH_THRESHOLD` se multiplica por dos.
   - `addEntry` añade un par clave/valor nuevo. En caso de que ya existiera un par con en la misma clave, reemplaza el valor. Devuelve el número de elementos que se han insertado con esa misma clave.
   - `removeEntry` elimina una ocurrencia del para clave/valor. Devuelve cero si sólo había un par con esa clave, o no había ninguno.

   Implementa los métodos `addEntry` y `rehash`.

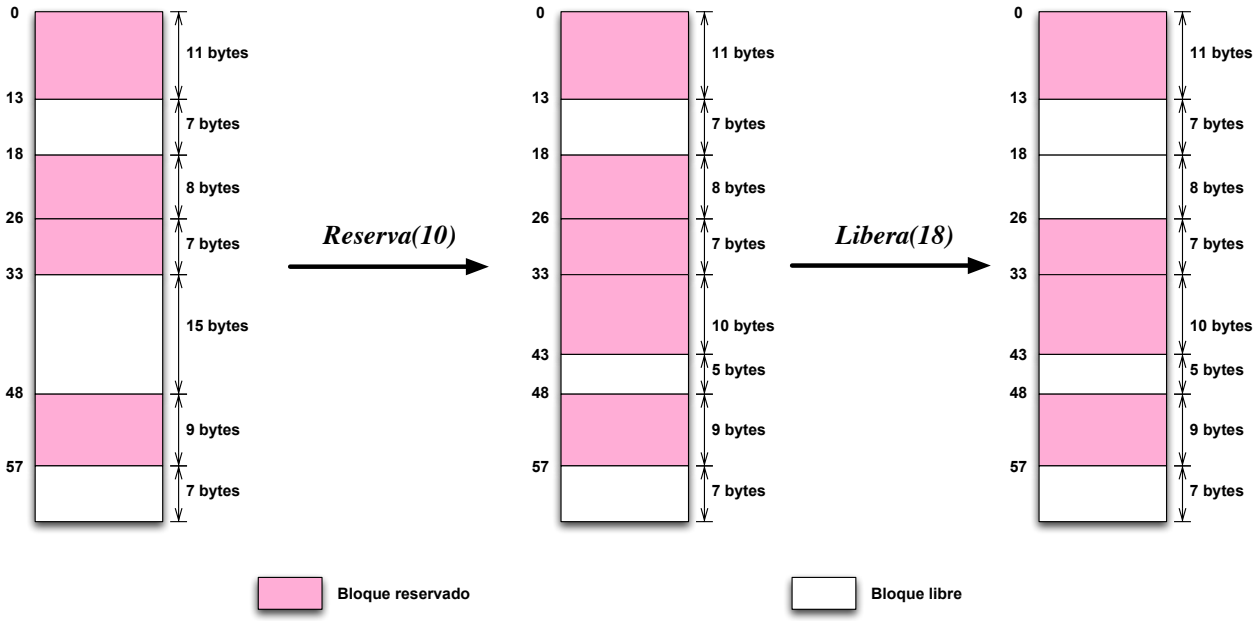
2. **(2 puntos)** Implementar un método estático que determine dado un vector de enteros si es un **MinHeap** (montículo a mínimos).

```
public static boolean esMinHeap (int v[]);
```

3. **(3 puntos)** Un gestor de memoria se encarga gestionar y repartir la cantidad de memoria disponible entre los distintos programas que se lo soliciten. Deseamos implementar uno cuyo funcionamiento básico responda a las siguiente reglas.

   - La memoria se divide en bloques de tamaño variable, que comienzan en una dirección de memoria *base* y tienen un *tamaño* en bytes determinado.
   - Los bloques pueden estar libres u ocupados, dependiendo de si han sido reservados o no.
   - Inicialmente sólo existirá un único bloque libre que empezará en la dirección cero y ocupará toda la memoria.
   - Cuando se realiza un reserva de memoria, el usuario indica la cantidad de bytes que quiere reservar. El gestor examina los bloques libres en busca de uno con suficiente tamaño para la cantidad solicitada. Si encuentra uno con el tamaño justo lo marca como ocupado. Si es mayor de lo necesario, el gestor divide él bloque en dos, uno con la cantidad solicitada y el otro con la restante. El primero es marcado como ocupado, el segundo sigue libre.
   - Cuando se realiza la liberación de un bloque éste es marcado como libre *libre*.

   A continuación podéis ver un ejemplo gráfico en el que sobre una memoria de 64 bytes ya segmentada se realizan una operación de reserva y liberación.

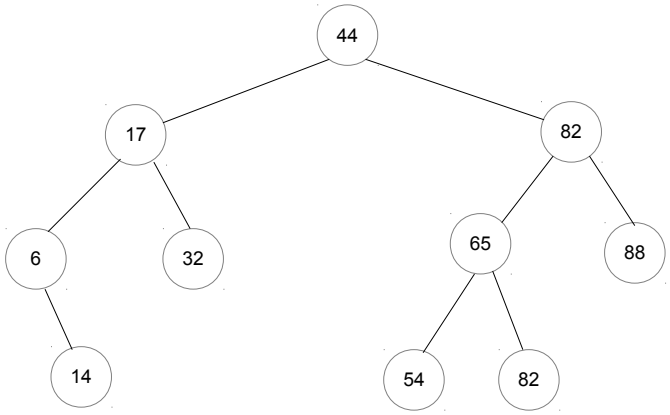


   Implementa la clase `GestorMemoria`. Define los campos y las estructuras auxiliares necesarias (*pista: utilizar una estructura de datos para los bloques libres y otra para los bloques ocupados*).

   La clase contará con los siguientes métodos públicos.

   - `GestorMemoria(int cantidad)`
     El constructor tomará el tamaño de la memoria y la pondrá toda como libre en un único bloque.
   - `int Reserva(int cantidad) throws ExcepcionMemoriaLlena`
     Reservará un bloque con la cantidad de memora indicada. Devolverá la dirección de memoria donde empiece el bloque reservado, o la excepción `ExcepcionMemoriaLlena` si no ha sido posible reservarla.
   - `void Libera(int direccion) throws ExcepcionDireccionInvalida`
     Liberará el bloque que comience en la dirección indicada. Si no existe un bloque que comience en esa dirección lanzará una excepción de tipo `ExcepcionDireccionInvalida`.

4. **(1 punto)** Implemetar un método estático recursivo que dado un árbol binario de búsqueda (de la clase `BinaryTree`) como el de la figura, escriba por pantalla la secuencia de elementos que contiene **ordenados de mayor a menor**.



5. **(1 punto)** Dado el siguiente grafo realizar un recorrido en anchura desde el nodo etiquetado con la letra *d*. La respuesta debe mostrar claramente la traza del algoritmo de recorrido en anchura y debe quedar claro qué nodos se visitan en cada iteración.