Estructuras de Datos 2022/23 – Segundo Parcial
17 de enero de 2023
15:00 – 17:00
Aulas: P3.02 y P3.09

Nombre, Apellidos y DNI:

Pregunta 1. (3 puntos)

Dado el array de seis posiciones de la Figura 1, se pide ordenarlo (de menor a mayor) por medio del algoritmo HeapSort.

No se pide realizar la implementación, se pide explicar textual y gráficamente (diagramas) el proceso, siendo ambas explicaciones necesarias. Teniendo esto en cuenta, **explica las diversas situaciones que se dan en las inserciones y eliminaciones del heap**.

19	25	22	1	17	2
0	1	2	3	4	5

Figura 1. Array correspondiente a la pregunta 1

Pregunta 2. (3,5 puntos)

Dadas dos tablas hash vacías (**tabla 1** y **tabla 2**), representadas por un array de 10 posiciones, **dibuja paso a paso**, cómo quedan las tablas con las operaciones indicadas en la Figura 2. Es **necesario explicar brevemente qué ocurre y cómo se actúa en cada nueva situación**, es decir, la primera vez que ocurre una nueva situación, deberá ser explicada pero no en las siguientes veces que ocurra.

Información común de ambas tablas:

- Función de dispersión a utilizar:
  $h(k) = ((\sum \text{dígitos de } k) * 2 + k + 3) \bmod 10$

  Por ejemplo:
  $h(22) = ((\sum \text{dígitos de } 2) * 2 + 22 + 3) \% 10 = (8 + 22 + 3) \% 10 = 33 \% 10 = 3$

Información sobre la **tabla 1**:

- La tabla de dispersión se representa por medio de un vector de nodos enlazados de parejas (clave, valor).
- Estrategia de dispersión utilizada: dispersión abierta.

Información sobre la **tabla 2**:

- La tabla de dispersión se representa por medio de un vector de tripletas (clave, valor, marca).
- Estrategia de dispersión utilizada: dispersión cerrada.

  $h_i(k) = (h(k) + i * h(k) + 1) \bmod 10$

  h(k) es la función de dispersión previamente comentada

  i es el número de intentos para buscar una nueva posición libre (0, 1, 2, ...)

  - Por ejemplo, para la pareja con clave 22 la secuencia de posiciones en las que buscar una posición libre será la siguiente:
    1. $h_0(22) = (h(22) + 0 * h(22) + 1) \% 10 = (3 + 0 + 1) \% 10 = 4 \% 10 = 4$
    2. $h_1(22) = (h(22) + 1 * h(22) + 1) \% 10 = (3 + 3 + 1) \% 10 = 7 \% 10 = 7$
    3. $h_2(22) = (h(22) + 2 * h(22) + 1) \% 10 = (3 + 6 + 1) \% 10 = 10 \% 10 = 0$
    4. …

1ª Operación -> Insertar (100, 1000)	6ª Operación -> Insertar (1, 10)
2ª Operación -> Insertar (110, 1100)	7ª Operación -> Insertar (104, 1040)
3ª Operación -> Insertar (0, 0)	8ª Operación -> Eliminar 0
4ª Operación -> Eliminar 100	9ª Operación -> Insertar (2, 22)
5ª Operación -> Insertar (2, 20)	10ª Operación -> Insertar (10, 1100)
	La pareja de cada operación tiene la forma **(clave, valor)**

Figura 2. Operaciones de la pregunta 2

Pregunta 3. (3,5 puntos)

Implementa un método llamado *removeSiblingLeaves* que reciba un *BinaryTree<E>* (llamado *original*) y devuelva un *LinkedBinaryTree<E>* que será el resultado de **eliminar las hojas del árbol *original* cuyo padre sea un nodo interior de grado 2**. Si el árbol *original* es vacío, devolverá un *LinkedBinaryTree<E>* vacío. A continuación, en la Figura 3, se muestra un ejemplo para un *BinaryTree >Integer>*.

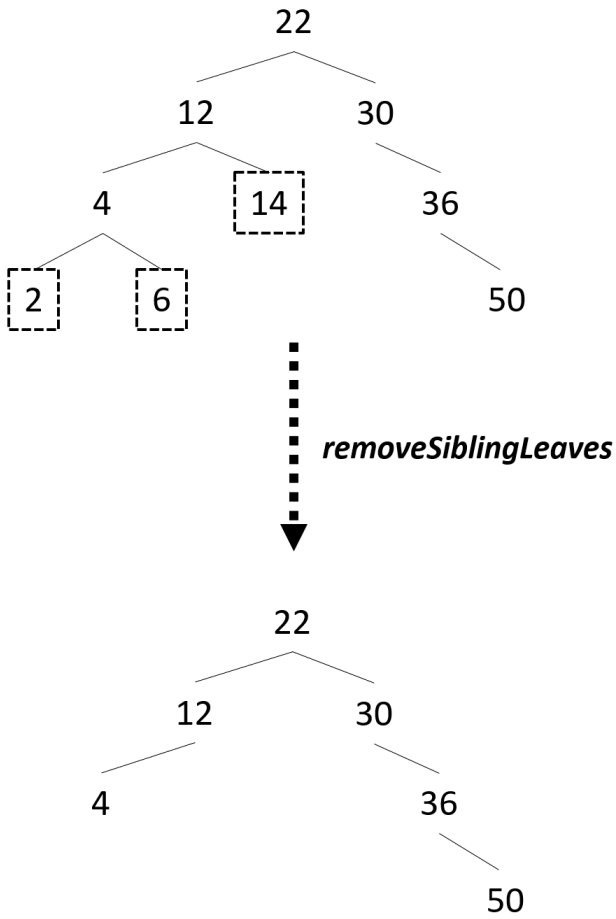

Figura 3. Ejemplos de la Pregunta 3 para el caso de un BinaryTree<Integer>

Información adicional

```
public interface BinaryTree<E> extends Collection<E> {
    BinaryTree<E> left();
    BinaryTree<E> right();
    E root();
    void removeLeft();
    void removeRight();
    int height();
    iterator<E> iterator();
    BinaryTreeIterator<E> iteratorPre();
    BinaryTreeIterator<E> iteratorIn();
    BinaryTreeIterator<E> iteratorPost();
    BinaryTreeIterator<E> iteratorLevels();
    void clear();
    boolean contains(Object o);
    boolean containsAll(Collection<?> c);
    boolean equals(Object o);
    int size();
    Object[] toArray();
    boolean isEmpty();
}
```