Nombre y Apellido: ...................................... N° Legajo: .............

#### Recuperatorio Primer Parcial de Estructuras de Datos y Algoritmos

#### Segundo Cuatrimestre de 2016

|  |  |  |  |
| --- | --- | --- | --- |
| Ejercicio 1 | ***Ejercicio 2*** | ***Ejercicio 3*** | Nota |
|  |  |  |  |

Condición Mínima de Aprobación: Tener por lo menos dos ejercicios con B-

## Ejercicio 1

Se tiene la siguiente clase para representar nodos en un árbol binario de búsqueda, donde “small” es la referencia al subárbol izquierdo, y “large” la referencia al subárbol derecho.

**public class Node<T> {**

**private T value;**

**private Node<T> small, large;**

**public BinaryTree(T value, BinaryTree<T> small, BinaryTree<T> large) {**

**this.value = value;**

**this.small = small;**

**this.large = large;**

**}**

**...**

**}**

Esta misma estructura se podría utilizar para representar una lista circular doblemente encadenada, donde “small” apunta al nodo anterior y “large” al nodo siguiente.

Implementar un algoritmo que dado un árbol binario de búsqueda con esta representación, modifique los punteros de los nodos para convertirlo en una lista circular doblemente encadenada que contiene los nodos en orden según un recorrido in-order.

El método a implementar recibe por parámetro la raíz del árbol, y retorna la referencia al primer nodo de la lista (el menor valor). **El algoritmo no debe crear estructuras nuevas, solamente modificar los punteros de los nodos ya existentes.**

**public static <T> Node<T> treeToList(Node<T> root)**

**El algoritmo debe tener complejidad temporal O(N), siendo N la cantidad de nodos.**

Ejemplo: Dado el árbol de la izquierda, el algoritmo recibiría por parámetro la raíz, y modificaría los punteros de los nodos ya existentes para convertirlo en la lista circular doblemente encadenada de la derecha. Finalmente retorna una referencia al nodo A.

**B**

**D**

**A**

**C**

**E**

**B**

**D**

**A**

**C**

**E**

## Ejercicio 2

Implementar una cola de prioridades que cumpla con la siguiente interfaz:

/\*\*

\* Interfaz para una cola de prioridades limitada en tamaño. La implementación recibe

\* por constructor el tamaño máximo de la cola y un comparador para utilizar como

\* criterio de prioridad a la hora de comparar elementos.

\*/

**public interface PriorityQueue<T> {**

/\*\*

\* Si hay lugar para agregar un elemento, lo agrega y retorna true. En caso

\* contrario no lo agrega y retorna false.

\*/

**public boolean enqueue(T value);**

/\*\*

\* Desencola el elemento con mayor prioridad. Si la cola está vacía retorna null.

\*/

**public T dequeue();**

**}**

A continuación se muestra un ejemplo de uso:

PriorityQueue<String> q = new PriorityQueueImpl<String>(3, new Comparator<String>() {

@Override

public int compare(String s1, String s2) {

return s1.compareTo(s2);

}

});

System.out.println(q.enqueue("B")); // true

System.out.println(q.enqueue("A")); // true

System.out.println(q.enqueue("D")); // true

System.out.println(q.dequeue()); // A

System.out.println(q.enqueue("C")); // false

System.out.println(q.dequeue()); // B

System.out.println(q.dequeue()); // D

System.out.println(q.dequeue()); // null

**Los métodos enqueue y dequeue deben tener complejidad temporal O(logN), siendo N la cantidad de elementos en la cola. No se puede utilizar la API de Java.**

## Ejercicio 3

Se tiene un árbol binario sin valores repetidos donde cada nodo almacena un número entero positivo. Implementar un algoritmo que imprima los nodos del árbol por niveles y de izquierda a derecha, pero sólo aquellos donde la suma desde la raíz hasta ese nodo es menor o igual a cierto valor N recibido por parámetro.

Ejemplo: para el siguiente árbol, si se ejecuta el algoritmo con N=14 la salida es {4, 7, 1, 2, 3}, si se ejecuta con N=22 la salida es {4, 7, 1, 2, 6, 3, 10, 5}, si se ejecuta con N=8 la salida es {4, 1, 2}.