Estructuras de Datos y Algoritmos I
Grado en Ingeniería Informática, Curso 2º

ACTIVIDAD 01

Ordenación: MergeSort, SortHeap y HeapSort.

Objetivos

- Aprender a hacer uso, implementar y comparar métodos de ordenación sobre `ArrayList<T>`.
- Hacer uso apropiado de las interfaces `Compare<T>` y `Comparator<T>`.
- Conocer nuevos métodos de ordenación basados cola de prioridad y heap.
- Conocer dos métodos nuevos de ordenación, `sortHeap()` y `heapSort()`, utilizando la estructura `PriorityQueue<T>`, así como una implementación propia de un *heap* binario.
- Obtener tiempos de ejecución de distintos métodos como base comparativa y evaluación.

Requisitos

Para superar esta actividad se debe realizar lo siguiente:
- Dominar un *heap* binario mínimo.
- Dominar la implementación de métodos sobre un *heap* binario mínimo.
- Dominar la implementación y uso de los métodos de ordenación `sort()`, `heapSort()` y `sortHeap()`.
- Dominar la implementación de un objeto `Comparator<T>` parametrizado.

Enunciado

En esta actividad vamos a hacer uso de lo aprendido en teoría relativo a la utilización de *heaps* (montículos) como base para la resolución de determinados problemas.

Vamos a hacer uso del método `sort()` ya implementado en la librería de Java y vamos a compararlo con dos nuevos algoritmos de ordenación que, tal y como hemos visto en clase, hace uso de *heaps* como estructura de datos subyacente: `sortHeap()` y `heapSort()`.

Para implementar `sortHeap()` se realizan los siguientes pasos:
- Crear una cola de prioridad vacía.
- Insertar cada uno de los *n* elementos del array en la cola.
- Extraer cada uno de los elementos de la cola y almacenarlos de forma consecutiva en el array.

El problema que tiene este algoritmo de inserción es que necesita un array más una cola de prioridad, ambos de tamaño *n*, es decir, que los recursos de memoria necesarios para la implementación de este método va a ser *2 * n*. ¿Qué ocurre si n es muy grande?

La solución al problema consiste en realizar la ordenación sobre **el mismo array**, aplicando el método conocido como `heapSort()`. Tal y como hemos visto en teoría, este algoritmo consiste en los siguientes pasos:

- Organizar el array `arr[]` a ordenar como un *heap* binario de mínimos.
- Recorrer el array `arr[]` desde el final (posición *n*) hasta la posición 1 (sin incluirla) y, para cada posición, hacer:
  - Intercambiar `arr[0]` con `arr[i-1]`.
- Reparar el heap sin modificar la parte ya ordenada (desde la posición i hasta el final del array ya queda ordenado). Para ello, haremos uso del método `siftDown()` (hundir) modificada, indicando la posición hasta la que se quiere hundir el elemento.
- Invertir el array, ya que el resultado va a estar ordenado en orden inverso al especificado en el comparador.

Para la realización de esta actividad se proporciona el **test** que, como ya sabéis, se deberá pasar correctamente. Para la implementación del programa, seguid el orden lineal en el que se han escrito los distintos subtest (que sigue un diseño bottom-up).

Como aspecto destacable, vamos a hacer uso de un `Comparator<T>` parametrizado, cuyo esqueleto es el que se muestra a continuación:

```
package org.eda1.actividad01;

import java.util.Comparator;

public class ComparatorPersona implements Comparator<Persona> {

    private enum Atributo {NOMBRE, COD};
    private Atributo atName;
    private boolean ordenAsc;

    public ComparatorPersona(String atributo, boolean ordenAsc){
    }

    @Override
    public int compare(Persona o1, Persona o2) {
    }

    private int compare(Persona o1, Persona o2, String cad1, String cad2){
    }
}
```

Aunque en Java se permite un uso más extendido y funcional de los enumerados (consultad documentación), en esta actividad vamos a hacer un uso sencillo y común de los mismos. Como podéis observar, en el constructor de `ComparatorPersona` le indicaremos el atributo (NOMBRE o COD) sobre el que vamos a realizar la ordenación, así como el orden del mismo (ascendente o descendente). De esta forma, con un único objeto de tipo `Comparator<T>` vamos a resumir la funcionalidad de 4 `Comparator<T>` independientes. Como último detalle comentad que, cuando dos objetos son iguales según el criterio de ordenación especificado en el comparador, el criterio secundario que se escoge es el de la ordenación natural en orden ascendente. Es decir, si yo le indico que me ordene el array en orden descendente según el atributo COD, si dos objetos tienen el mismo código, el orden secundario estará establecido por el atributo nombre, de menor a mayor (orden natural). Para más detalles, consultad el test, así como la base del programa que se adjunta en el repositorio de la asignatura.

Por ejemplo, si tenemos 5 objetos de tipo Persona:

```
Persona p1 = new Persona("Abraham", "5");
Persona p2 = new Persona("Julián", "3");
Persona p3 = new Persona("Abraham","1");
Persona p4 = new Persona("Jaime", "6");
Persona p5 = new Persona("Elisa", "3");
```

Y las añadimos a la estructura `aList` de tipo `ArrayList<Persona>`, el orden de inserción (sin ordenar) será:

```
Abraham [5], Julián [3], Abraham [1], Jaime [6], Elisa [3]
```

Si ordenamos por nombre en orden ascendente (orden natural), el resultado será:

```
Abraham [1], Abraham [5], Elisa [3], Jaime [6], Julián [3]
```

Si el orden es según el atributo código en sentido descendente, el resultado deberá ser:

```
Jaime [6], Abraham [5], Elisa [3], Julián [3], Abraham [1]]
```

En el caso de Elisa y Julián, como ambos tienen el mismo código, el orden será establecido de menor a mayor según el nombre (orden natural).