Práctica 1

Parcial 2. Temas 4. Heapsort

Objetivo

Implementar el código **original** para resolver la tarea planteada, demostrando que se alcanza el siguiente resultado de aprendizaje de la asignatura:

- **RAE2.** Entender estrategias algorítmicas clásicas (divide y vencerás, avance rápido y vuelta atrás), y ser capaz de utilizarlas a la hora de implementar soluciones a problemas concretos.
- **RAE5.** Comprender las principales estructuras de datos relacionales (grafos), su utilidad, su funcionamiento y su implementación. Ser capaz de emplearlas para la resolución de problemas concretos y como mecanismo para plantear soluciones más óptimas.

Para ello se pide codificar el algoritmo de ordenación ***Heapsort***, a partir de una implementación estática de un *Heap*.

Introducción

Como se ha estudiado en la teoría, un árbol binario se puede implementar de manera estática o de manera dinámica. En el primer caso, se utiliza un array como estructura de almacenamiento. Así, en una versión sencilla, se puede almacenar cada dato del árbol en una posición del array. Si enumerásemos cada elemento del árbol de 1 a N, desde la raíz y hacia las hojas, y dentro de cada nivel de izquierda a derecha, podríamos tener algo así:

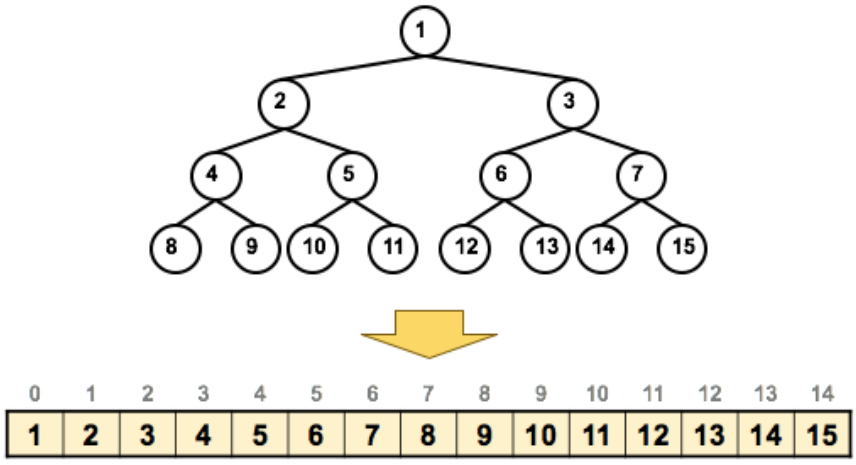

Figura 1. Implementación estática de un árbol binario

La principal ventaja de este tipo de representación es que es muy sencillo ascender y descender por la estructura. Con simples multiplicaciones y divisiones se puede descender a los hijos de un nodo, o ascender al padre, respectivamente.

Entre sus desventajas, está el problema del espacio. Un árbol sesgado de profundidad 4, requeriría crear un array como el de la figura, donde realmente sólo se almacenarían 4 datos.

Un Heap es un árbol completo en el que los datos están ordenados por prioridad. Por su propiedad de árbol completo, la implementación estática es ideal para un Heap, ya que no se desperdiciaría tanto espacio, y las operaciones de inserción y extracción se simplifican por la mencionada navegabilidad dentro de la estructura.

Descripción del trabajo

En la presente actividad, los/las estudiantes deberán trabajar **en parejas** para desarrollar una versión propia y original de código en Java, que implemente el algoritmo de ordenación del Heapsort, a partir de una implementación estática de un árbol binario.

Por simplicidad, los datos que almacenará el heap serán **números enteros mayores que cero (>0)**, y la **prioridad** será mayor cuanto más pequeño sea el elemento.

El código se descompondrá en las siguientes clases:

- Clase Heap.java
- Clase HeapSort.java

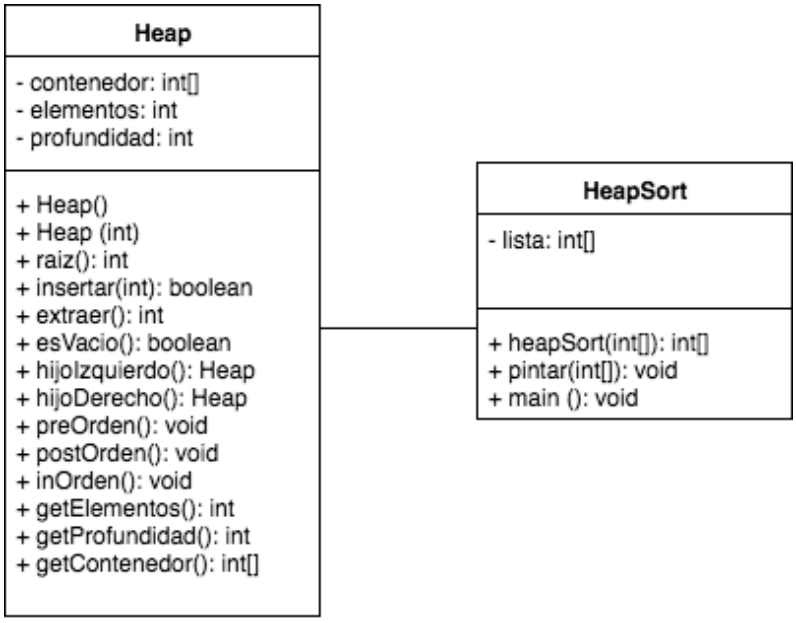

Figura 2. Diagrama de clases

Clase Heap

Implementará el heap sobre un array, y se compondrá de, al menos, los siguientes elementos:

- **Atributos (privados)**

Atributo	Descripción
- contenedor	Array de tamaño N que almacenará los datos del árbol
- elementos	Contador elementos que contiene el árbol.
- profundidad	Número de niveles máximo que se haya definido para el árbol, y que marcará la longitud máxima del array "contenedor".

- **Métodos (públicos)**

Método	Descripción
+ Heap()	Crea un árbol nulo: aquel cuyo array es nulo, y sus atributos de elementos y profundidad son cero (0).
+ Heap(int)	Crea un árbol vacío con una determinada profundidad máxima (determinada por el parámetro de entrada). El array contenedor tendrá como longitud el número exacto de posiciones para almacenar los datos de un árbol lleno de la profundidad indicada. Para indicar que una posición no está ocupada, se le asignará por defecto un valor negativo.
+ raiz(): int	Si existe, devuelve el valor almacenado en la raíz del árbol (posición 0 del array contenedor).
+ insertar(int): boolean	Añade al heap el elemento (número entero positivo) proporcionado como parámetro de entrada, devolviendo true si la operación se ha podido realizar. Tras la operación, el árbol debe seguir siendo un heap.
+ extraer(): int	Elimina del heap el elemento almacenado en la raíz, y lo devuelve como resultado. Tras la operación, el árbol debe seguir siendo un heap.
+ esVacio(): boolean	Devuelve verdadero si el árbol no tiene elementos, y falso en caso contrario.
+ hijoIzquierdo(): Heap	Devuelve un nuevo heap, que es el subárbol izquierdo del heap que invoca a la operación.
+ hijoDerecho(): Heap	Devuelve un nuevo heap, que es el subárbol derecho del heap que invoca a la operación.

+ preOrden(): void	Aplica un recorrido preOrden (izq→dcha) al árbol. La operación "visitar" simplemente escribirá por pantalla el contenido de la raíz, dejando un espacio en blanco entre un dato y otro (no escribiremos cada nodo en una línea diferente para facilitar la visibilidad).
+ postOrden(): void	Igual que el recorrido anterior, pero aplicando una estrategia postOrden (izq→dcha).
+ inOrden(): void	Igual que el recorrido anterior, pero aplicando una estrategia inOrden (izq→dcha).
+ getElementos(): int	Devuelve del valor del atributo elementos.
+ getProfundidad(): int	Devuelve del valor del atributo profundidad.
+ getContenedor(): int[]	Devuelve el array contenedor.

Los métodos hijoIzquierdo() y hijoDerecho() deben crear un nuevo árbol. Tendrán un nivel menos que su padre, y por tanto sus atributos (contenedor, elementos y profundidad) serán diferentes.

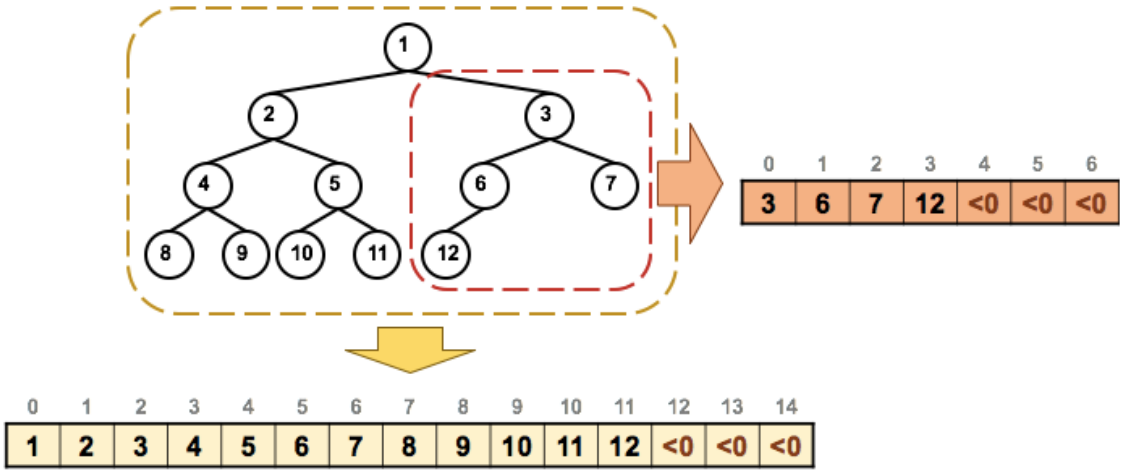

Figura 3. Organización de la información

Clase HeapSort

La clase HeapSort implementará el método de ordenación para un array desordenado de números enteros positivos. Además incluirá el método pintar, que podrá ayudar a trazar los diferentes pasos del proceso.

Método	Significado
+ heapSort(int[]): int[]	Método que implementa el algoritmo de ordenación heapsort. Recibe como entrada un array de números enteros mayores que cero, y devuelve como resultado otro array con los datos ordenados. En su interior debe utilizar un heap para realizar la ordenación.
+ pintar(int[]): void	Escribe por pantalla, en una sola línea, todos los datos del array de enteros que recibe como parámetro.

Requisitos formales de la actividad

Además de la revisión del código y la documentación a entregar, se realizarán una serie de pruebas automáticas, motivo por el que **deberá seguirse fielmente la especificación indicada** (nombres de las clases, atributos, métodos, tipos de datos, etc.)

Ambas clases no deben pertenecer a ningún paquete.

Es necesario, pero no suficiente, que el código compile para poder aprobar la actividad. Igualmente, es necesario utilizar los bucles adecuados y abandonarlos de manera estructurada para poder aprobar la actividad.

Ante cualquier duda o ambigüedad que pudiera surgir, puede recurrirse a los canales habituales de resolución de dudas: tutorías, foros de la asignatura en el Campus Virtual, etc.

Normas de entrega

Deben subirse al Campus Virtual **3 ficheros**: dos ficheros **.java** y un fichero **.pdf**:

1. Heap.java
2. HeapSort.java
3. PDF con la documentación

El documento en formato **PDF** describirá el trabajo realizado, la organización del mismo, las decisiones tomadas, las dificultades encontradas y cómo se han solucionado, la bibliografía consultada, etc. Este documento seguirá el siguiente formato al ser nombrado, siguiendo el orden alfabético de los apellidos de los estudiantes:

Apellidos_Estudiante1-Apellidos_Estudiante2.TPA.P3.PDF

Ejemplo:

Estudiante 1: Marcos García Ramos

Estudiante 2: Patricia Álvarez Torres

Álvarez_Torres-García-Ramos.TPA.P3.PDF

Es muy importante respetar los formatos y nombres anteriores. No hacerlo puede suponer la no aceptación de la entrega e imposibilitar aprobar el trabajo.

La entrega debe realizarse **antes** de las 23:55h del día fijado como límite de entrega.