UNIVERSIDAD EUROPEA DE MADRID

ESCUELA DE ARQUITECTURA, INGENIERÍA Y DISEÑO

GRADO EN INGENIERÍA INFORMÁTICA

Técnicas de Programación Avanzada

Práctica 1 Parcial 2. Temas 4. Heapsort

Manuel Araújo Baño

Daniel Gutiérrez Torres

CURSO 2024-2025

Universidad Europea

Índice

1 - Introducción **2**
2 - Objetivos **2**
3 - Fundamento teórico **3**
4 - Diseño de clases **4**
4.1 Clase Heap 4
4.2 Clase HeapSort 5
5 - Explicación del código **5**
HEAP.JAVA 5
HEAPSORT.JAVA 10
6 - Pruebas realizadas **11**
7 - Conclusiones **14**

1 - Introducción

En esta práctica, abordaremos la implementación del algoritmo de ordenación Heapsort utilizando una estructura de datos tipo heap representada mediante un array. De esta forma, mediante el uso de árboles binarios completos se ordenarán un conjunto de números enteros positivos aplicando la estrategia de divide y vencerás.

En la práctica se implementarán dos clases principales:

- Heap, clase encargada de gestionar la estructura del heap, así como de implementar sus operaciones fundamentales como: insertar elementos, eliminar el máximo o mínimo.
- HeapSort: clase para aplicar el algoritmo de ordenación HeapSort, utilizando la estructura de datos implementada en la clase Heap conjunto de números enteros positivos.

2 - Objetivos

El objetivo principal de esta práctica es desarrollar una implementación funcional del algoritmo Heapsort utilizando un heap estático basado en arrays.

De forma más concreta, se persiguen los siguientes objetivos:

- Implementar un árbol binario completo de forma estática mediante un array, que sea capaz de simular el comportamiento de un heap.
- Desarrollar las operaciones de inserción, extracción de la raíz y recorridos preOrden, postOrden e inOrden.
- Desarrollar habilidades en el diseño y la programación de clases en Java, respetando principios de encapsulamiento y modularidad.

3 - Fundamento teórico

Un árbol binario es una estructura de datos jerárquica en la que cada nodo tiene como máximo dos hijos: uno izquierdo y otro derecho. Cuando un árbol de esta condición está completamente lleno a excepción del último nivel y se organiza de una forma regida por unas propiedades de prioridad, se denominará heap.

Existen dos tipos de Heaps:

- Max-Heap: la raíz de cada subárbol contiene el valor máximo respecto a sus hijos, provocando que cada nodo es mayor o igual que sus descendientes.

- Min-Heap: la raíz de cada subárbol contiene el valor mínimo respecto a sus hijos, provocando que la prioridad es mayor cuanto más pequeño es el valor del elemento, por lo que los valores menores tienen preferencia y aparecen en la parte superior de la estructura.

Un Heap puede implementarse utilizando una estructura estática como un array, lo cual resulta bastante eficiente al permitir recorrer el árbol de forma sencilla, ya que las relaciones entre padres e hijos pueden calcularse mediante fórmulas simples sobre los índices del array. Gracias a ello, se facilita la implementación de operaciones básicas como la inserción, la extracción del elemento raíz de cada subárbol y el mantenimiento de la propiedad de prioridad del hea:

- Acceder al hijo izquierdo de un nodo en la posición i mediante 2*i + 1.

- Acceder al hijo derecho mediante 2*i + 2.

- Acceder al padre mediante (i - 1) / 2.

El algoritmo HeapSort lo podremos utilizar para ordenar números enteros positivos de manera eficiente. El proceso comienza insertando todos los elementos en una estructura llamada heap, donde los números se organizan en función de su prioridad.A continuación, se extrae repetidamente el elemento con mayor prioridad y se coloca en el array resultante. Tras cada extracción, el heap se reorganiza para mantener su estructura de prioridad. Este proceso continúa hasta que no queden elementos en el heap.El resultado final es un array ordenado. HeapSort es un algoritmo rápido y eficiente en cuanto a uso de memoria, ya que no requiere estructuras adicionales complejas.

Por otro lado, implementaremos tres tipos de recorridos sobre la estructura del heap los cuales permitirán explorar la estructura del heap desde diferentes perspectivas:

- Recorrido Preorden: visitaremos primero la raíz, luego el subárbol izquierdo y finalmente el subárbol derecho.

- Recorrido Inorden: visitaremos primero el subárbol izquierdo, luego la raíz y finalmente el subárbol derecho.

- Recorrido Postorden: visitaremos primero los subárboles izquierdo y derecho, y por último la raíz.

4 - Diseño de clases

4.1 Clase Heap

Esta clase representará la estructura de datos tipo heap utilizando un array como base con el objetivo de gestionar las operaciones básicas de inserción, extracción, recorridos y acceso a elementos sobre el árbol binario completo.

- Atributos principales:

  - **contenedor**: array de enteros que almacena los elementos del heap.
  - **elementos**: contador que indica cuántos elementos hay actualmente en el heap.
  - profundidad: número máximo de niveles del árbol, que define la longitud del array contenedor.

- Métodos:

  - **Heap()**: Constructor que crea un heap nulo sin datos.
  - **Heap(int profundidad)**: Constructor que crea un heap vacío con una profundidad máxima concreta definida.
  - **insertar(int valor)**: Inserta un nuevo número entero positivo en el heap, reorganizando si es necesario.
  - **extraer()**: Extrae el valor de mayor prioridad y reorganiza el heap.
  - **raiz()**: Devuelve el valor actual en la raíz del heap.
  - **esVacio()**: Indica si el heap está vacío.
  - **hijoIzquierdo()**: Devuelve el subárbol izquierdo como un nuevo objeto Heap.
  - **hijoDerecho()**: Devuelve el subárbol derecho como un nuevo objeto Heap.
  - **preOrden()**: Recorrido preOrden del árbol.
  - **inOrden()**: Recorrido inOrden del árbol.
  - **postOrden()**: Recorridos postOrden del árbol.
  - **getElementos()**: Devuelve todos los elementos almacenados en el heap.
  - **getProfundidad()**: Retorna la profundidad del Heap
  - **heapgetContenedor()**: Devuelve la estructura del heap que contiene los elementos del heap.

4.2 Clase HeapSort

Esta clase contiene el método que aplica el algoritmo Heapsort utilizando la clase Heap, así como un método auxiliar para visualizar resultados.

- Métodos:
  - **heapSort(int[] datos)**: recibe un array desordenado de enteros positivos y devuelve otro array con los datos ordenados.
  - **pintar(int[] datos)**: muestra en pantalla todos los elementos del array recibido.

5 - Explicación del código

HEAP.JAVA

Atributos de la clase

Los atributos de la clase están formados por el contenedor de elementos del heap, la cantidad de elementos y la profundidad de ete.

```
private int[] contenedor;
private int elementos;
private int profundidad;
```

Constructor vacío

Constructor usado para inicializar los atributos del heap como estructura vacía.

```
public Heap() {
   this.contenedor = null;
   this.elementos = 0;
   this.profundidad = 0;
}
```

Constructor por parámetros

Constructor para inicializar el heap con una profundidad especificada, asignando el tamaño adecuado al contenedor según la fórmula del número de nodos en un heap completo.

```
public Heap(int profundidad) {
   this.profundidad = profundidad;
   int tamaño = (int) Math.pow(2, profundidad) - 1;
```

```
    this.contenedor = new int[tamaño];
    for (int i = 0; i < tamaño; i++) {
        contenedor[i] = -1;
    }
    this.elementos = 0;
}
```

Función raíz

Método para devolver el valor de la raíz del heap.

```
public int raiz() {
    if (!esVacio()) {
        return contenedor[0];
    }
    return -1;
}
```

Función insertar

Método para insertar un valor en el heap, asegurando que se mantenga la propiedad del heap mediante el proceso de *heapify up*.

```
public boolean insertar(int valor) {
    if (valor <= 0 || elementos >= contenedor.length) {
        return false;
    }
    contenedor[elementos] = valor;
    int i = elementos;
    while (i > 0) {
         int padre = (i - 1) / 2;
         if (contenedor[padre] > contenedor[i]) {
             int temp = contenedor[padre];
             contenedor[padre] = contenedor[i];
             contenedor[i] = temp;
             i = padre;
         } else {
             break;
         }
    }
    elementos++;
    return true;
}
```

Función extraer

Método para extraer el valor con mayor prioridad y reorganizar el heap usando *heapify down*.

```
public int extraer() {
    if (esVacio()) return -1;
    int raiz = contenedor[0];
    contenedor[0] = contenedor[elementos - 1];
    contenedor[elementos - 1] = -1;
    elementos--;
    heapify(0);
    return raiz;
}
```

Función heapify

Método para asegurarse que la propiedad del heap se mantiene al reorganizar los elementos a partir de una posición dada.

```
private void heapify(int i) {
    int menor = i;
    int izq = 2 * i + 1;
    int der = 2 * i + 2;

    if (izq < elementos && contenedor[izq] < contenedor[menor]) {
        menor = izq;
    }
    if (der < elementos && contenedor[der] < contenedor[menor]) {
        menor = der;
    }
    if (menor != i) {
        int temp = contenedor[i];
        contenedor[i] = contenedor[menor];
        contenedor[menor] = temp;
        heapify(menor);
    }
}
```

Función esVacio

Método para verificar si el heap está vacío y no contiene elementos.

```
public boolean esVacio() {
    return elementos == 0;
}
```

Función hijoIzquierdo

Método para obtener el subheap correspondiente al hijo izquierdo del heap actual.

```
public Heap hijoIzquierdo() {
    Heap hijo = new Heap(this.profundidad - 1);
    int inicio = 1;
    int fin = Math.min(contenedor.length, 1 + (int) Math.pow(2, this.profundidad - 1) - 1);
    for (int i = inicio; i < fin; i++) {
        if (contenedor[i] > 0) hijo.insertar(contenedor[i]);
    }
    return hijo;
}
```

Función hijoDerecho

Método para obtener el subheap correspondiente al hijo derecho del heap actual.

```
public Heap hijoDerecho() {
    Heap hijo = new Heap(this.profundidad - 1);
    int inicio = 1 + (int) Math.pow(2, this.profundidad - 1) - 1;
    int fin = Math.min(contenedor.length, inicio + (int) Math.pow(2,
this.profundidad - 1) - 1);
    for (int i = inicio; i < fin; i++) {
        if (contenedor[i] > 0) hijo.insertar(contenedor[i]);
    }
    return hijo;
}
```

Función imprimir preOrden

Método para recorrer los elementos del heap siguiendo un recorrido en preorden (raíz, izquierdo, derecho).

```
private void postOrden(int i) {
    if (i >= contenedor.length || contenedor[i] <= 0) return;
    postOrden(2 * i + 1);
    postOrden(2 * i + 2);
    System.out.print(contenedor[i] + " ");
}
```

Función inOrden

Método para recorrer los elementos del heap siguiendo un recorrido en inorden (izquierdo, raíz, derecho).

```
private void inOrden(int i) {
     if (i >= contenedor.length || contenedor[i] <= 0) return;
     inOrden(2 * i + 1);
     System.out.print(contenedor[i] + " ");
     inOrden(2 * i + 2);
}
```

Función postOrden

Método para recorrer los elementos del heap siguiendo un recorrido en postorden (izquierdo, derecho, raíz).

```
private void postOrden(int i) {
   if (i >= contenedor.length || contenedor[i] <= 0) return;
   postOrden(2 * i + 1);
   postOrden(2 * i + 2);
   System.out.print(contenedor[i] + " ");
}
```

Función getElementos

Método para obtener la cantidad de elementos actualmente almacenados en el heap.

```
public int getElementos() {
    return elementos;
}
```

Función getProfundidad

Método para obtener la profundidad del heap.

```
public int getProfundidad() {
    return profundidad;
}
```

Función getContenedor

Método para obtener el contenedor interno del heap.

```
public int[] getContenedor() {
    return contenedor;
}
```

HEAPSORT.JAVA

Función heapSort

Método para ordenar un arreglo de números enteros, utilizando el algoritmo de ordenación heap sort con los métodos definidos en la clase anterior.

```
public static int[] heapSort(int[] array) {
   int profundidad = (int) (Math.floor(Math.log(array.length) / Math.log(2)) + 1);
   Heap heap = new Heap(profundidad);
   for (int valor : array) {
       heap.insertar(valor);
   }

   int[] ordenado = new int[array.length];
   for (int i = 0; i < array.length; i++) {
        ordenado[i] = heap.extraer();
   }
   return ordenado;
}
```

Main

Desde el main de la clase probaremos el algoritmo de ordenación heap sort.

```
public static void main(String[] args) {
      int[] datos = {9, 3, 7, 1, 8, 2, 5};

      System.out.print("Array original: ");
      pintar(datos);

      int[] ordenado = heapSort(datos);

      System.out.print("Array ordenado: ");
      pintar(ordenado);
}
```

6 - Pruebas realizadas

Para comprobar que el HeapSort funciona correctamente se han realizado 7 pruebas diferentes:

Caso 1: Array desordenado (caso básico)

```
// Caso 1: Array desordenado (caso básico)
int[] datos1 = {9, 3, 7, 1, 8, 2, 5};
System.out.println("1. Array desordenado:");
System.out.print("   Original: ");
pintar(datos1);
int[] ordenado1 = heapSort(datos1);
System.out.print("   Ordenado: ");
pintar(ordenado1);
System.out.println();
```

Resultado:

Original: 9 3 7 1 8 2 5

Ordenado: 1 2 3 5 7 8 9

Resultado: Éxito

Caso 2: Array ya ordenado

```
// Caso 2: Array ya ordenado
int[] datos2 = {1, 2, 3, 4, 5, 6, 7};
System.out.println("2. Array ya ordenado:");
System.out.print("   Original: ");
pintar(datos2);
int[] ordenado2 = heapSort(datos2);
System.out.print("   Ordenado: ");
pintar(ordenado2);
System.out.println();
```

Resultado:

Original: 1 2 3 4 5 6 7

Ordenado: 1 2 3 5 7 8 9

Resultado: Éxito

Caso 3: Array en orden inverso

```
// Caso 3: Array en orden inverso
int[] datos3 = {7, 6, 5, 4, 3, 2, 1};
System.out.println("3. Array en orden inverso:");
System.out.print("   Original: ");
pintar(datos3);
int[] ordenado3 = heapSort(datos3);
System.out.print("   Ordenado: ");
pintar(ordenado3);
System.out.println();
```

Resultado:

Original: 7 6 5 4 3 2 1

Ordenado: 1 2 3 5 7 8 9

Resultado: Éxito

Caso 4: Array con elementos repetidos

```
// Caso 4: Array con elementos repetidos
int[] datos4 = {5, 2, 5, 3, 2, 8, 5};
System.out.println("4. Array con elementos repetidos:");
System.out.print("   Original: ");
pintar(datos4);
int[] ordenado4 = heapSort(datos4);
System.out.print("   Ordenado: ");
pintar(ordenado4);
System.out.println();
```

Resultado:

Original: 5 2 5 3 2 8 5

Ordenado: 2 2 3 5 5 5 8

Resultado: Éxito

Caso 5: Array con un solo elemento

```
// Caso 5: Array con un solo elemento
int[] datos5 = {42};
System.out.println("5. Array con un solo elemento:");
System.out.print("   Original: ");
pintar(datos5);
int[] ordenado5 = heapSort(datos5);
System.out.print("   Ordenado: ");
```

```
pintar(ordenado5);
System.out.println();
```

Resultado:

Original: 42

Ordenado: 42

Resultado: Éxito

Caso 6: Array grande

```
// Caso 6: Array grande
int[] datos6 = {25, 17, 36, 2, 8, 19, 42, 30, 11, 13, 5, 27, 22, 39, 14};
System.out.println("6. Array grande:");
System.out.print("   Original: ");
pintar(datos6);
int[] ordenado6 = heapSort(datos6);
System.out.print("   Ordenado: ");
pintar(ordenado6);
System.out.println();
```

Resultado:

Original: 25 17 36 2 8 19 42 30 11 13 5 27 22 39 14

Ordenado: 2 5 8 11 13 14 17 19 22 25 27 30 36 39 42

Resultado: Éxito

Caso 7: Array Vacio

```
// Caso 7: Array vacío
int[] datos7 = {};
System.out.println("7. Array vacío:");
System.out.print("   Original: ");
pintar(datos7);
try {
    int[] ordenado7 = heapSort(datos7);
    System.out.print("   Ordenado: ");
    pintar(ordenado7);
} catch (Exception e) {
    System.out.println("   Error: " + e.getMessage());
}
System.out.println();
```

Resultado:

Original: ∅

Error: -1

Resultado: Éxito

7 - Conclusiones

La presente práctica tuvo como objetivo implementar el algoritmo de ordenación Heapsort mediante una estructura de datos tipo heap binario estático, representado internamente con un array. Para ello, se desarrollaron dos clases: Heap, encargada de la gestión del árbol (inserciones, extracciones y recorridos), y HeapSort, responsable del proceso de ordenación.

El algoritmo consistió en construir un heap a partir de una lista de elementos y luego extraer iterativamente el elemento máximo para obtener un arreglo ordenado de forma ascendente. Esta implementación permitió comprender tanto el funcionamiento interno del algoritmo Heapsort como la aplicación práctica de estructuras de datos y conceptos de programación orientada a objetos en Java.

En conclusión, la actividad reforzó el entendimiento de algoritmos de ordenación eficientes y promovió el desarrollo de habilidades técnicas en el manejo estructurado de datos.