

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

ACTIVIDAD 2

Implementar operaciones sobre un Heap y heapsort

Objetivos

- Aprender a implementar un *heap* binario mínimo o máximo en un array, haciendo uso de `Comparator`.
- Aprender a utilizar esta estructura de datos e implementar funciones que realicen otra funcionalidad aunque siempre respetando la propiedad del *heap*.
- Conocer un nuevo método de ordenación, `heapSort`, utilizando el *heap* binario como estructura de datos subyacente.

Requerimientos

Para superar esta actividad se debe realizar lo siguiente:

- Dominar un *heap* binario mínimo.
- Dominar la implementación de nuevas funciones sobre un *heap* binario mínimo.
- Dominar el `heapSort` como nuevo método de ordenación.

Enunciado

En esta actividad, que consta de dos ejercicios, vamos a complementar las clases de teoría con la utilización de los *heaps* o montículos en determinados problemas.

Ejercicio 1. En este primer ejercicio se van a implementar diferentes operaciones sobre un *heap* mínimo implementado en un `ArrayList`.

En teoría hemos estudiado los *heap* (montículos) máximos y mínimos, y en prácticas hemos hecho uso de dicha implementación en la clase `class Heap<T>`. En dicha implementación hemos utilizado un `ArrayList<T> theHeap`, que tiene la raíz en la posición 0. Para obtener un *heap* mínimo de enteros (`Integer`) hemos declarado un *Comparator* de la siguiente forma `Less<Integer> less = new Less<Integer>()`, y dicho *heap mínimo* lo hemos declarado de la siguiente manera `Heap<Integer> heap = new Heap<Integer>(less)`. En esta actividad se pide implementar las siguientes operaciones sobre dicho *heap*.

- **decreaseKey(i, x)**. Esta nueva operación reduce el valor del elemento del *heap* en la posición i ($0 \leq i \leq \text{theHeap.size()} - 1$), un valor positivo x ($x > 0$). Dado que esta operación puede violar el orden del *heap*, como sugerencia, puede ser necesario realizar alguna operación del tipo `siftUp`.
- **increaseKey(i, x)**. Esta nueva operación incrementa el valor del elemento del *heap* en la posición i ($0 \leq i \leq \text{theHeap.size()} - 1$), un valor positivo x ($x > 0$). Dado que esta operación puede violar el orden del *heap*, como sugerencia, puede ser necesario realizar alguna operación del tipo `siftDown`.
- **replaceKey(i, x)**. Esta nueva operación reemplaza el valor del elemento del *heap* en la posición i ($0 \leq i \leq \text{theHeap.size()} - 1$), por un valor positivo x ($x > 0$). Dado que esta operación puede violar el orden del *heap*, como sugerencia, puede ser necesario realizar alguna operación del tipo `siftUp` o `siftDown`.
- **delete(i)**. Esta nueva operación elimina el nodo (elemento) en la posición i ($0 \leq i \leq \text{theHeap.size()} - 1$) del *heap*. A nivel orientativo y como sugerencia, una forma de implementar esta operación podría ser, realizando primero **decreaseKey(i, Δ)** y después **removeMin()**.
- **String branchMinSum()** sobre un *heap* mínimo, que *devuelve* la rama (desde la hoja hasta la raíz) con el valor mínimo de la suma de valores de los nodos desde las hojas hasta la raíz. Es decir, ésta debe ser una función *bottom-up* que procese cada una de las ramas desde cada una de las hojas hasta la raíz, acumulando el valor de cada nodo y devolviendo la que tiene el valor mínimo.
- **isHeap()**. Esta nueva función implementa una función que devuelva `true` cuando el contenido del `ArrayList theHeap` verifica la condición de orden del *heap* para todos sus elementos, y `false` en caso contrario. Destacar que dicha función se puede implementar siguiendo un recorrido *top-down* o *bottom-up* sobre el árbol binario implementado en el `ArrayList`. Si dispone de tiempo, implemente los dos algoritmos.

Para la implementación de las anteriores operaciones puede ser necesario implementar otras nuevas basadas en las conocidas `siftUp` y `siftDown`. Por ejemplo, `protected void siftUp(int start)`, `protected void siftDown(int start)` y `public T getValue(int i)`.

Una vez implementadas estas nuevas operaciones se deben de comprobar su correcto funcionamiento sobre el *heap* representado en el `ArrayList theHeap`, pudiendo necesitar la ayuda de otras operaciones previamente implementadas, como por ejemplo `isHeap()`. Para la realización de este ejercicio se proporcionara el **test** que se deberá pasar correctamente.

En la siguiente figura podemos observar el diagrama de clases UML para la clase *Heap* con todos los métodos necesarios para poder realizar este ejercicio.

org.eda1.actividad02.ejercicio01.Heap<T>
◆ <u>DEFAULT_INITIAL_CAPACITY: int</u>
◆ theHeap: ArrayList<T>
◆ comparator: Comparator<T>
● Heap(initialCapacity: int, comp: Comparator<T>): void
● Heap(): void
● Heap(initialCapacity: int): void
● Heap(comp: Comparator<T>): void
● Heap(otherHeap: Heap<T>): void
● size(): int
● isEmpty(): boolean
● add(element: T): void
◆ siftUp(): void
◆ compare(element1: T, element2: T): int
◆ swap(parent: int, child: int): void
● getMin(): T
● getValue(i: int): T
● removeMin(): T
◆ siftDown(start: int): void
● toString(): String
◆ siftDown(start: int, end: int): void
● assign(index: int, value: T): void
● makeHeap(): void
● isHeap1(): boolean
● isHeap(): boolean
● branchMinSum(): String
● showHeap(): void
● increaseKey(i: int, x: Integer): boolean
◆ siftUp(start: int): void
● decreaseKey(i: int, x: Integer): boolean
● replaceKey(i: int, x: T): boolean
● delete(i: int): boolean
● displayHeap(): void

Ejercicio 2. En este segundo ejercicio se van a implementar los dos algoritmos de ordenación utilizando los *heaps* estudiados en clase: *sortHeap* y *heapSort*.

Para implementar **sortHeap** se realizan los siguientes pasos

- Crear un montículo vacío de tamaño n
- Insertar cada uno de los n elementos del array en el montículo
- Extraer cada uno de los elementos y almacenarlos en el array

El problema que tiene este algoritmo de inserción es que necesita un array, más un montículo, ambos de tamaño n , es decir, recursos de memoria necesarios para la implementación de **sortHeap** = $2 * n$

La solución al problema es realizar la ordenación en el mismo vector de tamaño n , obteniendo **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 (ordenación ascendente, de menor a mayor)
- Recorrer el array `arr[]` desde el final (posición n) hasta la posición 1 (sin incluirla), y para cada posición i hacer:
 - Intercambiar `arr[0]` con `arr[i-1]`
 - Reparar heap, sin modificar la parte ya ordenada (desde i hasta el final el vector ya queda ordenado). Llamar a una función **siftDown** (hundir) modificada, indicando la posición hasta la que se quiere hundir el elemento (**siftDown(arr, 0, i - 1)**)
- Invertir el array, ya que el resultado utilizando un **Comparator Less** genera un array de mayor a menor (el **Comparator Greater** es justo lo contrario). Como aclaración de esto último se presentan a continuación ambos Comparators.

```
package org.edal.actividad02.ejercicio02;

import java.util.Comparator;






public class Greater<T> implements Comparator<T>
{
    public int compare(T x, T y) {
        return -((Comparable<T>)x).compareTo(y);
    }
}

package org.edal.actividad02.ejercicio02;

import java.util.Comparator;

public class Less<T> implements Comparator<T>
{
    public int compare(T x, T y) {
        return ((Comparable<T>)x).compareTo(y);
    }
}
```

Para la realización de este ejercicio se proporcionará el **test** que se deberá pasar correctamente. Y en la siguiente figura se muestra el diagrama de clases UML para la clase *HeapSort*, que incluye también el método `sortHeap`.

 org.eda1.actividad02.ejercicio02.HeapSort	
	<u>main(args: String[]): void</u>
	<u>sortHeap(aList: ArrayList<T>, comp: Comparator<T>): void</u>
	<u>siftDown(aList: ArrayList<T>, first: int, last: int, comp: Comparator<T>): void</u>
	<u>heapSort(aList: ArrayList<T>, comp: Comparator<T>): void</u>