Laboratorio 3

Heap Sort

11/12/2022

2º GEI

Estructura de Dades

Universitat de Lleida

David Carreras y Dídac Cayuela

Índice

Nuestra solución para el problema del HeapSort está basada siguiendo el modelo de funcionamiento explicado en el PDF. En primer lugar, implementamos la estructura de la clase BinaryHeap, donde desarrollamos funciones básicas como la obtención del elemento padre, hijo derecho e izquierdo dado un índice. A partir de ahí hemos generado algunas funciones auxiliares que nos han sido de gran utilidad para el desarrollo del HeapSort y las comentaremos a continuación:

Función parent

Para obtener el elemento padre dado un índice, podemos usar la fórmula (i-1) / 2 pasada por la función Math.floor(). En caso de que el índice que calculemos dentro de la función no sea válido lanzaremos una excepción de IndexOutOfBounds.

```
int parentIdx = (index - 1) / 2;
parentIdx = (int)Math.floor(parentIdx);

if (parentIdx < 0) {
   throw new IndexOutOfBoundsException("Invalid parent
index. This function must not be called using a root
index");
}
```

Función hasParent:

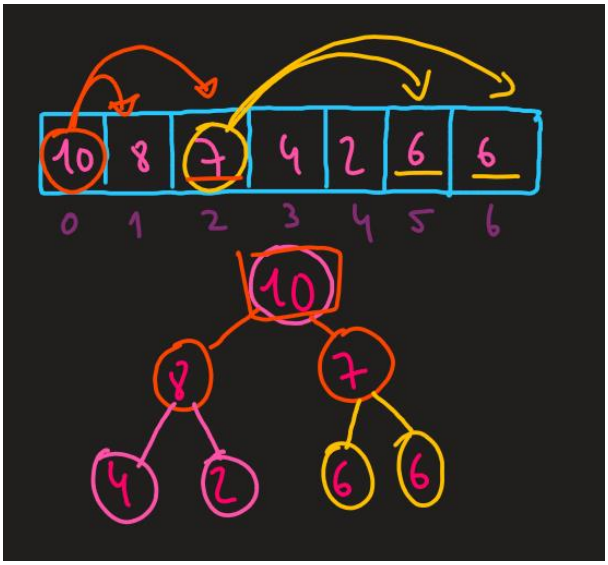
Si queremos saber si un cierto índice tiene un elemento padre, nos basta con comprobar que dicho índice es como mínimo uno, pues en dicho caso, su padre sería el elemento cero.

```
boolean hasParent(int index) {
   return index > 0;
}
```

Función left

Para obtener el hijo izquierdo dado un índice, basta con multiplicar el índice del elemento que es el padre de dicho izquierdo por dos y añadirle 1

```
return 2 * index + 1
```



Función right

Para obtener el hijo derecho dado un índice el proceso es prácticamente idéntico que el comentado anteriormente, con la única diferencia de que le añadiremos dos al índice. Los índices hijos siempre son consecutivos

```
return 2 * index + 2;
```

Función hasLeft y hasRight

Para saber si un elemento tiene un hijo izquierdo, es suficiente con comprobar que el índice de su hijo izquierdo esté dentro de los valores del vector o de la parte del vector considerada para el heap, que en nuestro caso es heapSize y obviamente cero.

Caso de right (y para left):

```
right(index) < heapSize && index >= 0;
```

Funciones Auxiliares:

Para poder entender el funcionamiento de *addElement y deleteRoot,* así como el funcionamiento de la función *heapSort* en sí, primero debemos comentar las funciones auxiliares que nos han permitido implementarlas.

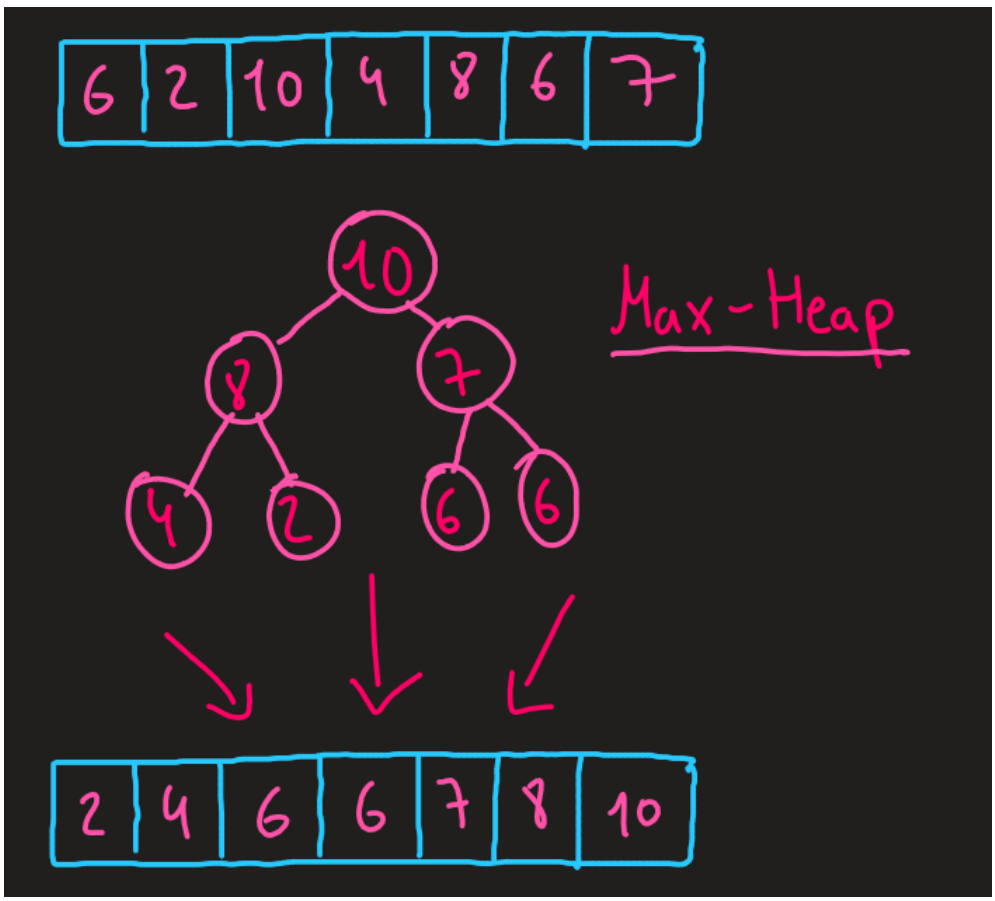
upHeapify

La función *upHeapify* nos permite asegurar que se cumplen los invariantes del heap, haciendo comprobaciones de abajo a arriba del árbol, empezando desde el índice de un hijo en concreto.

Hemos implementado esta función de manera recursiva. Mientras el índice dado tenga un padre comprobamos que este no sea mayor a su padre, en caso afirmativo los intercambiaremos para mantener la propiedad del heap y posteriormente volveremos a llamar *upHeapify,* pero esta vez con el índice del padre como parámetro.

De esta manera iremos corrigiendo el árbol moviéndonos desde un elemento dado hasta llegar a la raíz del árbol, que es la única que no tiene padre.

```
if (hasParent(index)) {
   if (comparator.compare(elements[index], elements[parent(index)]) > 0) {
       swapByIndex(index, parent(index));
       upHeapify(parent(index));
   }
}
```



downHeapify

De forma similar a *upHeapify,* la función *downHeapify* nos garantiza una vez más que cumplimos los invariantes del heap, con la diferencia de que las comprobaciones se hacen moviéndonos hacia abajo del árbol.

Calculamos el hijo izquierdo y derecho del elemento pasado como parámetro y marcamos el índice actual como el mayor de los tres en la variable *largest.* si el hijo izquierdo es válido, miramos si es mayor que el índice actual, en caso afirmativo éste será el nuevo *largest.* hacemos lo mismo con el hijo derecho, pero comparándolo con *largest.* De esta manera, cuando acabamos de ejecutar estas comprobaciones tenemos guardado en *largest* el elemento con más valor de los 3.

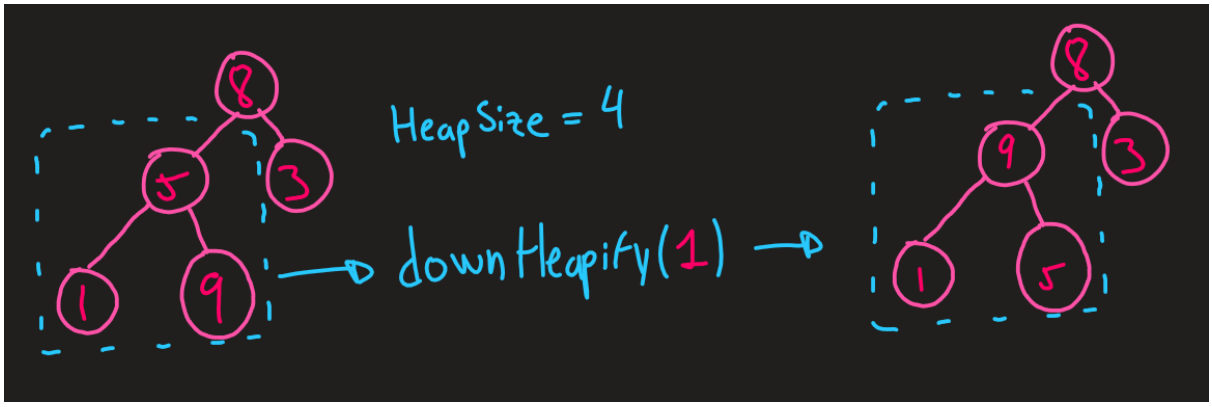
```
if (hasLeft(index) && (comparator.compare(elements[index], elements[left]) < 0)) {
   largest = left;}

if (hasRight(index) && (comparator.compare(elements[largest], elements[right]) < 0)) {
   largest = right;}
```

Si alguno de los hijos es mayor, es decir, si *largest* es distinto al índice del padre, entonces intercambiamos *largest* con este para mantener la propiedad del heap y posteriormente llamamos de manera recursiva a *downHeapify* con largest como parámetro.

```
if (largest != index) {
   swapByIndex(largest, index);
   downHeapify(largest);}
```

De esta manera recorremos el árbol de arriba a abajo, empezando desde un elemento padre dado hasta acabar en una hoja y comprobando solo esas partes del árbol que precisan de cambios.



addElement

Con la ayuda de las funciones mencionadas anteriormente, añadir un elemento al heap es una tarea relativamente sencilla.

añadimos el elemento pasado como parámetro al array elements en la posición con valor maxHeap y que coincide con el final del prefijo, por tanto, en realidad lo que estamos haciendo es añadir el nuevo elemento al final del árbol.

Debido a esta operación de inserción ahora debemos llamar a la función *upHeapify* que se asegurará de que la rama del árbol que contiene este nuevo elemento mantiene los invariantes del heap.

También debemos acordarnos de aumentar el *heapSize,* ya que hemos insertado un nuevo elemento al prefijo del array que contiene los elementos ordenados como un *max-heap*.

```
heapSize++;
elements[lastPosition] = element;
upHeapify(lastPosition);
```

deleteRoot

Para borrar la raíz del árbol primero debemos intercambiar el elemento raíz con el único nodo que al quitarlo no deja el árbol hueco, este elemento es el último del prefijo *heapSize.*

Luego debemos decrementar *heapSize* debido a que ahora hay un elemento menos en el prefijo, y entonces llamamos a *downHeapify* para arreglar la estructura del árbol empezando desde la nueva raíz, y así devolver esta y el resto del árbol en posiciones que mantengan el heap.

```
heapSize--;
E element = elements[0];
swapByIndex(0, heapSize);
downHeapify(0);
```

heapSort

Para realizar el HeapSort hay que entender que nuestro array consistirá de dos zonas, el prefix(HeapSize) donde tendremos el max heap con todos los elementos que forman el árbol en este momento y el sufix, que considerará los elementos que en este momento no forman parte de dicha heap (o bien porque todavía no han sido añadidos o porque ya han sido eliminados del árbol).

El primer paso consiste en generar la heap a partir de los elementos que forman el sufix, por tanto, iremos añadiendo tantos elementos al árbol como elementos tengamos en el sufix. Como en un inicio heapSize= 0 y sufix es elements.length, podemos realizar la operación de añadir tantas veces como elementos tengamos en el array. De esta forma nos aseguramos añadir todos y cada uno de los elementos del sufix al heap.

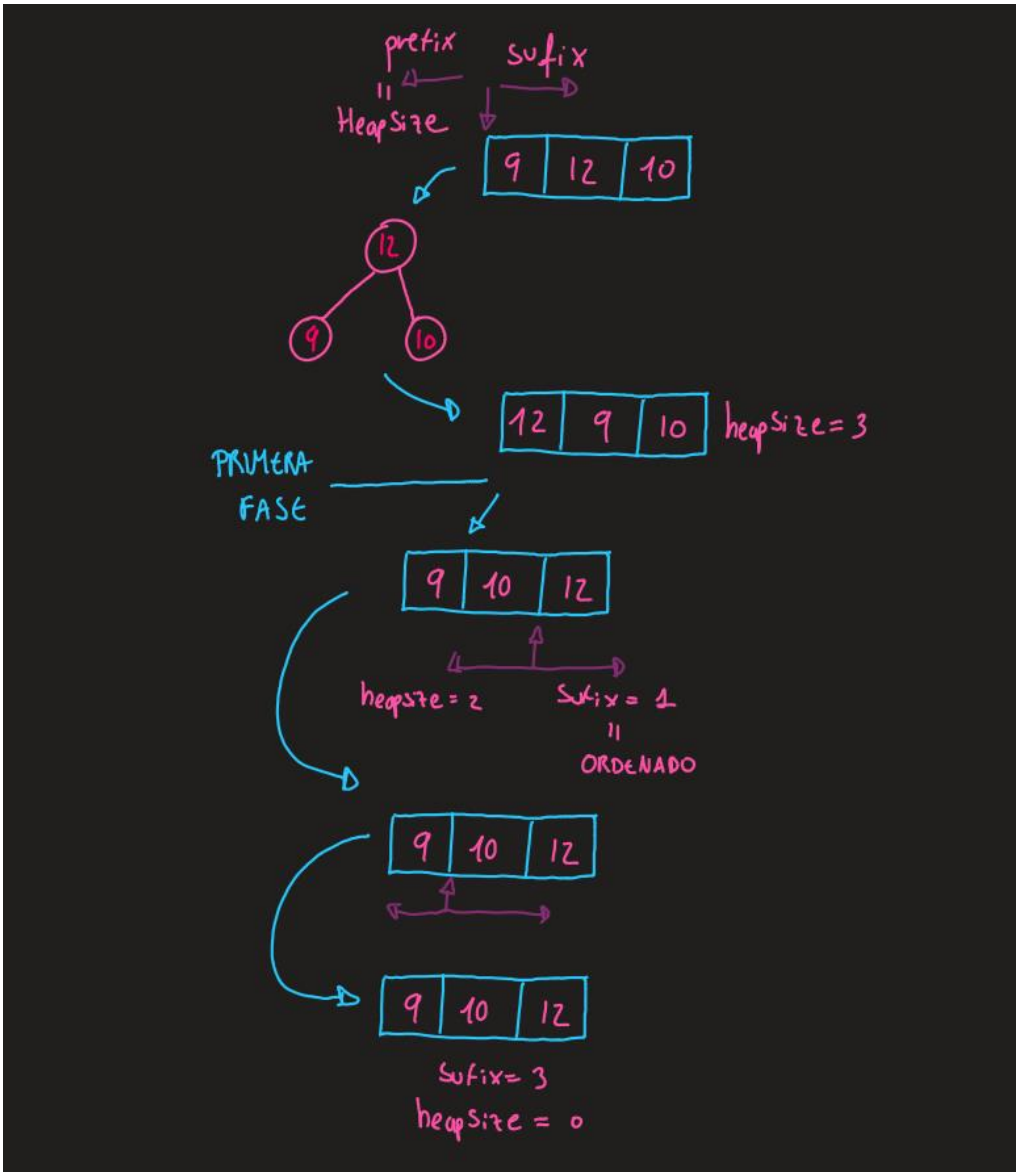
La función add, como se ha explicado con anterioridad, se encargará de que la propiedad de ser un max-heap se cumpla pese a añadir un nuevo elemento (un elemento padre siempre es mayor o igual que sus hijos). Resulta evidente comentar que, por cada elemento que añadimos en el heap, nuestro heapSize incrementa, hasta el punto que cuando finalice el bucle heapSize será elements.length y el sufix 0, pues todo el array será un árbol con la propiedad del max-heap.

```
for (E element : elements) {
    addElement(element);
}
```

El siguiente paso consiste en eliminar el mayor elemento, que siempre será la raíz del Heap. Tras esto dicho máximo se intercambia con el último elemento ubicado en el array y se decrementa el tamaño del prefix(HeapSize). Posteriormente para volver a conseguir la estructura del Max Heap se realiza una ordenación del árbol hasta obtener el max heap una vez eliminada la raiz e intercambiada por el último elemento. De todas estas tareas se encargará nuestra función deleteRoot, que será llamada mientas todavía queden elementos por eliminar del heap, o lo que es lo mismo, mientras el heapSize sea 1 o más querrá decir que todavía hay elementos por eliminar y ser insertados en el sufix.

```
while (heapSize > 0) {
    deleteRoot();
}
```

De esta forma a medida que habremos ido desmontando el heap, en el resto del array(sufix) quedarán los elementos ordenados de forma ascendente.



Test que añadimos:

emptySort() → Ordenación de un array vacío

oneElementSort() → Ordenación con tan solo un elemento

twoElementSort() → Ordenación con dos elementos

samplePdfSort() → Test inspirado en el ejemplo de sort explicado en el pdf

add() → Test para comprobar que la función add funciona correctamente

delete() →Test para comprobar la función deleteRoot

sort1(), sort2(), sort3() → Ordenaciones de arrays de Integers

Sobre la complejidad de HeapSort:

Una ordenación mediante el algoritmo HeapSort es *O(n log(n))* debido a la estructura de datos utilizada (árbol binario en este caso) donde cada operación requiere a lo sumo de log x pasos, donde x sea la cantidad de elementos que faltan por mirar. Dicha estructura y funcionamiento es la base para dicho algoritmo. Destacar que en el peor de los casos el algoritmo podría llegar a tener una complejidad de $O(n^2)$, pero una media sería la que hemos comentado *O(n log(n))*.

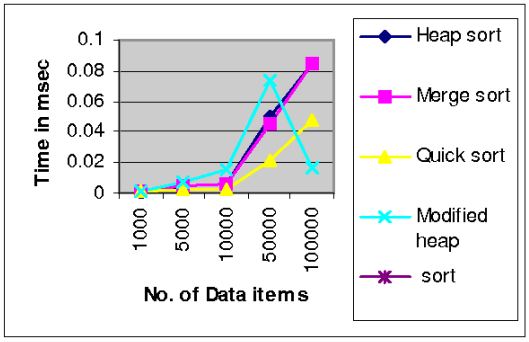


Fig 2 .Comparison of sorting algorithms performance