Estructuras de Datos y Algoritmos. Curso 2009/2010
Unidad Temática III: EDAs de Búsqueda y su Jerarquía Java

Tema 4. Montículo Binario: una estructura para la Representación de Cola de Prioridad y la Ordenación rápida

Mabel Galiano Natividad Prieto

Departamento de Sistemas Informáticos y Computación
Escuela Técnica Superior de Informática Aplicada

Índice

Objetivos y Bibliografía

El objetivo general de este tema es presentar una implementación jerárquica de la EDA Cola de Prioridad que, en el peor de los casos soporta la inserción de nuevos Datos y el acceso y eliminación del mínimo en tiempo logarítmico, y que utiliza como soporte para sus Datos una Representación Contigua mediante un sencillo `array`. Esta implementación eficiente de la Cola de Prioridad que recibe el nombre de **Montículo Binario** o **Binary Heap**, permitirá además retomar el problema de la Ordenación eficiente de un `array` en base a la estrategia de Selección, conduciendo al diseño del método de ordenación genérico **Heap Sort**.

Como bibliografía básica, en este tema se utilizarán los apartados del 1 al 5 del capítulo 20 del libro de Weiss, M.A. "**Estructuras de datos en Java**".

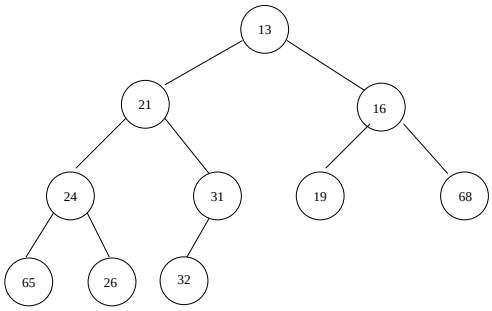
1. El Montículo Binario como soporte de Datos para la Búsqueda Dinámica del Dato de máxima prioridad: definición y propiedades

En el tema anterior se propuso el uso de un **Árbol Binario de Búsqueda** como soporte para los datos de la Cola de Prioridad (la clase `ABBColaPrioridad`) proporcionando una implementación de la interfaz en la que, bajo ciertas condiciones de equilibrado del ABB, todas sus operaciones tienen coste logarítmico. Pues bien, en este tema se estudiarán los Heaps o Montículos Binarios como un soporte jerárquico para la Representación de los Datos de la Cola de Prioridad, dando lugar a la implementación más eficiente de este modelo que, como se demostrará, tiene las siguientes ventajas: se puede implementar utilizando un sencillo `array`; permite la implementación de los métodos `insertar` y `eliminarMin` con coste logarítmico en el peor de los casos, el coste promedio de `insertar` es constante y el coste de `recuperarMin` también es constante, aún en el caso peor.

Cuestión: ¿Cuál sería el coste de las operaciones de la EDA Cola de Prioridad si se utilizara como soporte para sus datos una Lista Enlazada Genérica (LEG)? ¿Y si se utilizara una LEG Ordenada?

Un **Montículo Binario** o **Heap** se define como un Árbol Binario (AB) con una estructura y una ordenación de sus datos que favorece la Búsqueda Dinámica eficiente del Dato de máxima prioridad. En un Montículo Binario se cumplen las dos propiedades siguientes:

1. **Propiedad estructural:** es un **AB Completo**, por lo que su altura es a lo sumo $\lfloor logN \rfloor$; de este modo se asegura un coste logarítmico aún en el caso peor si los algoritmos implican la exploración de un camino de la raíz hasta una hoja. Además, este tipo de ABs admiten una representación implícita sobre array.
2. **Propiedad de orden:** la Raíz del AB contiene el Dato de máxima prioridad de la Cola, el que requiere un mínimo tiempo de espera, por lo que se puede acceder a él directamente, en tiempo constante. Dado que todo subÁrbol del Heap es a su vez un Heap, se tiene que el Dato que contiene cada Nodo es menor o igual que el que contienen sus descendientes. Nótese que esta propiedad establece que el dato en el nodo padre nunca es mayor que el Dato de sus hijos. En la figura siguiente se muestra un ejemplo de un Montículo Binario.



Nótese que los datos de un Heap deben poder compararse en base a su prioridad o tiempo de espera; es decir, deben ser de alguna clase `E` que implemente el interfaz `Comparable<E>`.

Además, todos los caminos en el Heap contienen datos que forman secuencias ordenadas ascendentemente; por ejemplo, en la figura anterior el camino desde la raíz a cualquier hoja permite obtener las siguientes secuencias ordenadas: $13, 21, 24, 65$; $13, 21, 24, 26$; $13, 21, 31, 32$; $13, 16, 19$ y $13, 16, 68$. Nótese que, si por la propiedad de orden el Dato que contiene cada Nodo es menor o igual que el que contienen sus Descendientes, la raíz del árbol tiene el valor mínimo. No obstante, independientemente del tipo de Recorrido que se realice, resulta imposible obtener la Secuencia ordenada de los Datos del Heap.

Dado que por definición un Heap es un AB Completo, admite una representación Implícita según la cuál sus Datos se representan sobre un array, `elArray` y se utiliza un entero que representa su tamaño actual, `talla`. Con esta representación, `elArray[1]` es la raíz, y por lo tanto el valor mínimo en un Montículo Minimal (o el máximo en un Montículo Maximal). Recuérdese que, dado un nodo `elArray[i]`, su hijo izquierdo es `elArray[2*i]`, si $2 * i \leq talla$, su hijo derecho `elArray[2*i+1]`, si $2 * i + 1 \leq talla$, y si $i \neq 1$, `elArray[i/2]` es su Padre. Además las componentes del array cumplen la **Propiedad de Orden de Montículo**, que se puede enunciar a través del siguiente predicado: $\forall i : 1 < i \leq talla : elArray[i/2] \leq elArray[i]$.

Cuestiones:

1. En general, ¿qué operaciones de un AB NO pueden mejorar su coste en un Heap? ¿Por qué? ¿Cuáles sí podrían hacerlo? ¿Por qué?. En particular, si se utiliza un Heap como Representación de una Cola de Prioridad ¿qué tipos de toString de un AB (InOrden, PreOrden, PostOrden y PorNiveles) son útiles? ¿Por qué? ¿Cuál resulta más eficiente y por qué?
2. ¿Se puede calcular en el orden de una constante el número de hojas de un AB? ¿Y si es un AB Completo? ¿y el de un Heap? ¿Cómo? Responder a las mismas cuestiones pero sobre altura y tamanyo.
3. ¿Se puede localizar en alguna zona o punto de un AB el Dato máximo que contiene? ¿Y el mínimo ? ¿Y si el AB es Completo? ¿y si es un Heap Minimal? ¿Cómo y con qué coste?.

La definición de Montículo dada corresponde a lo que se denomina **Montículo Binario Minimal** (o también ***minHeap***); de forma equivalente se podría definir un **Montículo Binario Maximal** (o también ***maxHeap***) para permitir el acceso eficiente al valor máximo en lugar de al mínimo.

2. Implementación de una Cola de Prioridad según un Montículo Binario: la clase genérica MonticuloBinario

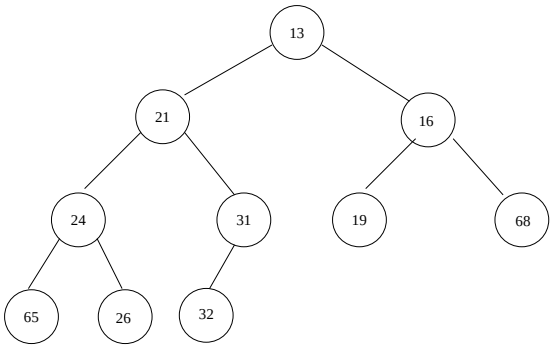
Para la implementación en Java de un Montículo Binario se tendrá como objetivo el diseño de una forma de Representación de los Datos al servicio de la Implementación de la EDA Cola de Prioridad estudiada en el tema `1` de esta misma Unidad Didáctica. Por ello, se diseñará dentro del paquete `jerarquicos` la clase `MonticuloBinario` que implementará la interfaz `ColaPrioridad`. Los atributos mínimos para esta clase son un `array` genérico de datos, `elArray`, un entero que representa su tamaño actual, `talla`, y una constante que representa su capacidad o tamaño inicial, `CAPACIDAD_POR_DEFECTO`. Nótese que los datos de un

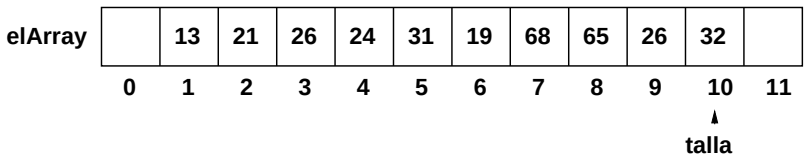
`MonticuloBinario` deben ser de una clase `E` que implemente el interfaz `Comparable<E>`. La estructura de la clase será como sigue:

```
package librerias.estructurasDeDatos.jerarquicos;
import librerias.estructurasDeDatos.modelos.*;
public class MonticuloBinario<E extends Comparable<E>> implements ColaPrioridad<E> {
   protected E [] elArray;
   protected static final int CAPACIDAD_POR_DEFECTO = 11;
   protected int talla;
   public MonticuloBinario() { ... }
   public void insertar(E x) { ... }
   public E eliminarMin() { ... }
   public E recuperarMin() ) { return elArray[1] }
   public boolean esVacia() ) { ... }
   public String toString() ) { ... }
   protected void duplicarArray ) { ... }
   ...
}
```

El método `insertar`

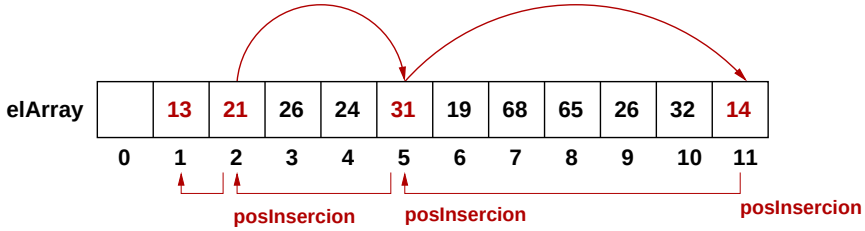
El método `insertar` se debe implementar de forma que se preserven las dos propiedades enunciadas que caracterizan a un Montículo: la propiedad estructural y la de ordenación parcial. Así, para insertar el Dato `x` en el Montículo se debe añadir en la primera posición disponible del array (`talla++`), para no violar la propiedad estructural del heap y a continuación ***reflotarlo*** sobre sus antecesores hasta situarlo en un punto en el que no se viole la propiedad de orden del heap. En las figuras siguientes se muestra un ejemplo de Montículo Binario y su representación sobre el array:



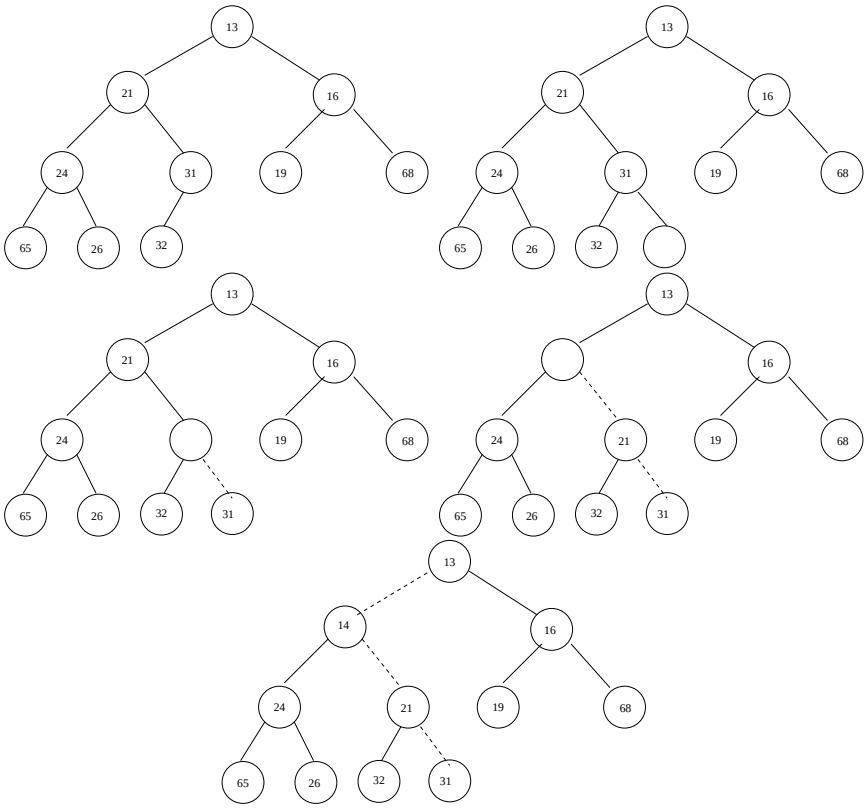


Para insertar sobre este Heap el Dato `14` se procede como sigue: se añade en la primera posición libre de `elArray`, esto es, en la posición **11** pero ahora ya no se cumple la propiedad de ordenación de montículo, puesto que `14` es un Dato menor que el contenido en su nodo

padre que es `31`. Para restablecer la propiedad de orden se debe buscar la posición apropiada de inserción entre sus antecesores, para ello, se comparará con los valores `31`, `21` y `13`, hasta encontrar el primer Dato menor que el que se está comparando; éste ya puede ser el padre del nuevo dato, cumpliéndose de este modo la propiedad de ordenación; al igual que se hacía en el método de inserción, también los datos de la secuencia de comparación se deben desplazar para hacer hueco al nuevo. En la figura siguiente se señalan los Datos de comparación y los desplazamientos que se realizarán. Nótese que si el Dato a añadir fuera el nuevo mínimo; es decir, fuera menor que los que hay actualmente en el Montículo, el proceso de búsqueda del lugar de inserción terminaría al encontrar la raíz, quedando el nuevo Dato en la posición 1, como cabía esperar.



En las siguientes figuras se muestra sobre la representación gráfica del Montículo como un AB, el proceso de inserción utilizado como ejemplo.



El código Java del método `insertar` de la Clase `MonticuloBinario` es el que se describe a continuación; obsérvese que antes de insertar se debe comprobar si es necesario redimensionar el array, esto se ha resuelto de la misma forma que en las implementaciones de Pila y Cola sobre array: llamando al método `duplicarArray`. La posición de inserción inicialmente está en la posición libre del array `++talla` y se inicia un bucle de búsqueda en el que la condición de fin es encontrar un Dato menor o igual al que se inserta o llegar al final del array (en este caso a la posición 1); en cada paso del bucle se compara el Dato a insertar `x` con el que hay en el padre de la actual posicion de inserción `elArray[posInsercion/2]` si éste es mayor el valor del padre se copia en `posInsercion` y `posInsercion` pasa a estar en el padre. Cuando finaliza el bucle `posInsercion` es el índice donde insertar `x`.

```
public void insertar (E x) {
   if ( talla == elArray.length-1 ) duplicarArray();
   int  posInsercion = ++talla;
   while ( (posInsercion > 1) && x.compareTo(elArray[posInsercion/2])<0 ) {
      elArray[posInsercion] = elArray[posInsercion/2];
      posInsercion = posInsercion/2;
   }
   elArray[posInsercion] = x;
}
@SuppressWarnings("unchecked")
private void duplicarArray() {
   E [] elArrayAnt = elArray;
   elArray = new Comparable[talla*2+1];
   for (int i = 1; i < elArrayAnt.length; i++) elArray[i] = elArrayAnt[i];
}
```

La Complejidad Temporal del método `insertar` se establece en función del número de Datos del Montículo, que viene dado por el valor del atributo `talla`. El método presenta instancias correspondiendo el Caso Mejor a una situación en la que el Dato que se añade es mayor que el que hay en el que a partir de ese momento será su Nodo Padre; en el ejemplo propuesto se daría cuando se inserta cualquier valor mayor que 31, por ejemplo 40; en este caso el número de comparaciones a realizar en el bucle es 1 y su coste Constante.

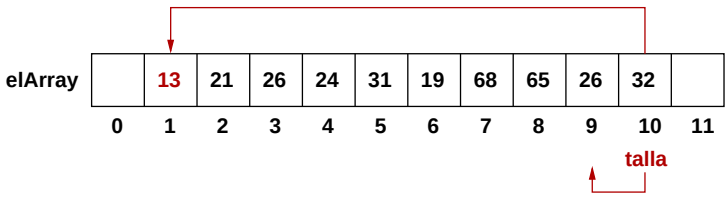
El Caso Peor se produce cuando el elemento que se inserta es el nuevo mínimo; en este caso se realizarán tantas comparaciones como Datos se encuentren en el camino hasta la raíz, que es precisamente la altura del árbol que, como se sabe, es logarítmica con el número de nodos; así, el coste del método en el Caso Peor es `O(log talla)`. Además, se ha demostrado que en promedio se requieren 2.6 comparaciones para llevar a cabo una inserción, de forma que en promedio `insertar` mueve un elemento 1.6 niveles.

Cuestiones:

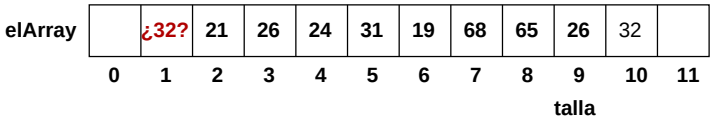
1. Hacer una traza de `insertar` el valor 3 sobre el Montículo `[0,1,4,8,2,5,6,9,15,7,12,13]`;
2. Hacer una traza de `insertar` a partir de un Montículo vacío los siguientes datos: 6, 4, 15, 2, 10, 11, 8, 1, 13, 7, 9, 12, 5, 3, 14.

El método `eliminarMin`

El acceso al elemento menor es directo ya que está en la posición 1 del array. Para eliminar este Dato sin violar la propiedad estructural de Montículo, el Dato situado en la última posición ocupada del array (`talla`), que por otra parte se debe recolocar ya que `talla` se debe decrementar en 1, se ubicará en la posición del Dato que se elimina, esto es en la posición 1 del array, como se muestra en la figura siguiente:

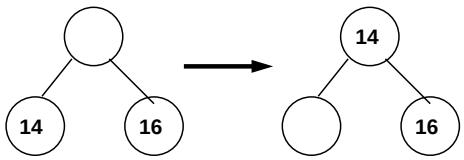


Después de este cambio se cumple la propiedad estructural pero no la de Ordenación, ya que el Dato que se coloca en la posición 1 proviene del último nivel del Árbol que es dónde se encuentran precisamente los valores mayores:

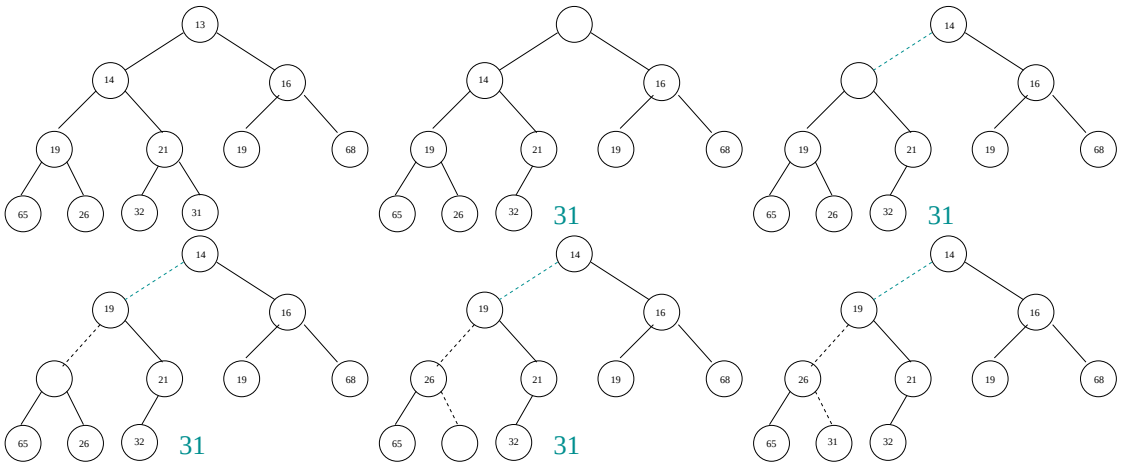


Se tiene ahora una representación que no es un Montículo debido al valor de su raíz pero que sus hijos izquierdo y derecho si que lo son; para restituir la propiedad de Orden al Montículo es suficiente con **hundir** la posición del hueco hasta encontrar la posición correcta para el dato que se debe colocar. Este es un proceso similar al realizado en el método `insertar`, la diferencia está en que ahora el hueco se debe mover hacía abajo: **se debe hundir el hueco en lugar de reflotarlo**.

Este proceso de hundimiento es más complejo que el de reflotamiento debido a que, en este caso, hay que considerar los dos hijos que puede tener (en el reflotamiento únicamente se comparaba con uno, el padre) y hundir el hueco siempre en la dirección del hijo menor tal y como se indica en la siguiente figura:



En las siguientes figuras se detalla el proceso completo para `eliminarMin` en el Montículo $[13, 14, 16, 19, 21, 19, 68, 65, 26, 32, 31]$: el hueco inicialmente está en la posición 1 y el Dato a recolocar es el 31; el hueco será ocupado bien por el elemento a recolocar, en cuyo caso terminaría el proceso, bien por el menor de los hijos del hueco, en este caso por el dato 14, desplazándose el hueco en esa dirección (subárbol izquierdo); el siguiente hueco lo ocuparía el 19 moviéndose el hueco de nuevo a la izquierda, éste sería ocupado por el 26 y el hueco se movería a la derecha siendo ocupado ya por el elemento a recolocar, el 31, terminando así el proceso.



El código en Java de `eliminarMin` es el que se muestra a continuación; este método, aparte de devolver el valor mínimo mediante una llamada al método `recuperarMin`, consiste en copiar el Dato de la última posición ocupada del array sobre la posición 1 y llamar a un método, `hundir`, que se encarga de hundir el hueco hasta colocar este Dato y restablecer la propiedad de orden. El método `hundir` se define como un método privado de la Clase, tiene un parámetro que es la posición del hueco (aunque desde `eliminarMin` el hueco siempre se genera en la posición 1) y parte del supuesto que tanto el subárbol izquierdo como el derecho de éste son Montículos. El código de `hundir` es básicamente una búsqueda desde el hueco y sobre sus descendientes de la posición correcta para el Dato que incumple la propiedad de ordenación, `elArray[hueco]`; en cada paso de la iteración, se compara el hijo derecho con el izquierdo dejando sobre la variable local `hijo` la posición del menor; nótese que esta comparación sólo es posible si el `hueco` tiene dos hijos, lo cuál se comprueba con la primera condición de la guarda del `if`. Seguidamente, este hijo menor se compara con el valor que se desea recolocar y si es menor el proceso debe concluir, lo cuál se implementa mediante la variable local `esHeap` y sino el valor en el hijo sube al hueco y el hueco continúa hundiéndose. Cuando termina el bucle el valor a recolocar se asigna sobre el hueco.

```
public E eliminarMin() {
   E elMinimo = recuperarMin();
   elArray[1] = elArray[talla--];
   hundir (1);
   return elMinimo;
}
private void hundir(int hueco) {
  E aux = elArray[hueco];
  int hijo = hueco*2; boolean esHeap = false;
  while ( hijo <= talla && !esHeap){
    if ( hijo != talla && (elArray[hijo+1]).compareTo(elArray[hijo]) < 0 ) hijo++;
    if ( (elArray[hijo]).compareTo(aux) < 0 ){
       elArray[hueco] = elArray[hijo];
       hueco = hijo; hijo = hueco*2;
    } else esHeap = true;
  }
  elArray[hueco] = aux;
}
```

La complejidad del método `eliminarMin` viene determinada por la del método `hundir`, ya que la preservación de la propiedad estructural del heap, i.e. el intercambio de los datos de las posiciones `1` y `talla` y el decremento de `talla`, se efectúa en tiempo constante. Para estudiar la complejidad temporal del método `hundir` se va a considerar como tamaño del problema la altura del Heap. Para un tamaño dado, se observan las siguientes instancias significativas:

- Mejor de los Casos: cuando `aux` ya es menor o igual que el mínimo de sus Hijos, i.e. cuando `esHeap = true` tras concluir la primer iteración del `while`. En este caso la cota de complejidad del método hundir sería $T_{hundir}(H) \in \theta(1)$
- Peor de los Casos: cuando `aux` resulta ser la última Hoja del Heap donde se hunde, i.e. cuando tras concluir el bucle `while hueco` es igual a `talla` (y nótese que `esHeap = false`). En este caso, $T^{p}_{hundir}(H) \in \theta(H)$ y $T_{insertar}(H) \in O(logtalla)$

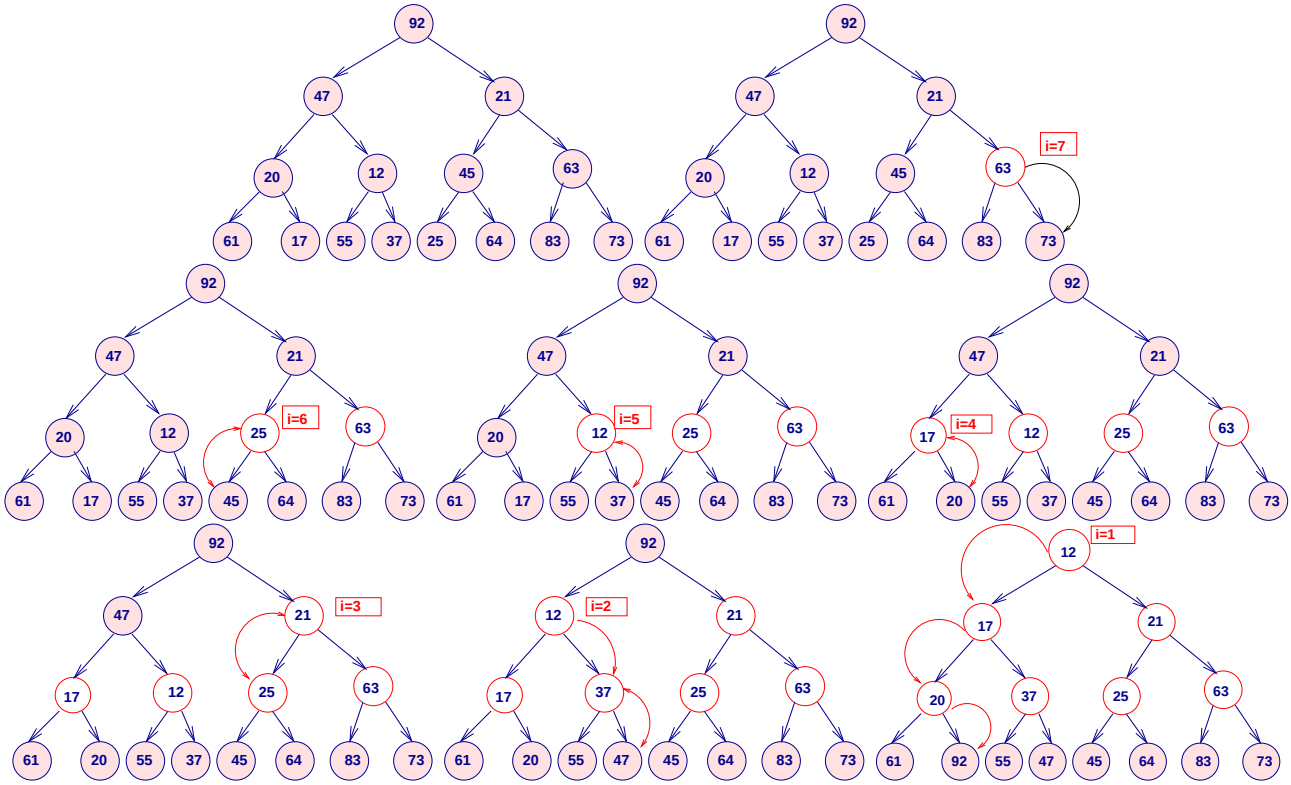
Cuestión: Una Cola de Prioridad implementada como Montículo Minimal, ¿funciona como Cola para Datos de igual prioridad? ¿Por qué?. Para responder a esta cuestión, conviene observar el orden en el que saldrán los Datos de prioridad 1 del Montículo que resulta de insertar, en el orden mostrado, los Pares (1,a), (2,a), (1,b), (3,a), (4,a), (1,c), (6,a), (1,d) (donde la primer componente es la prioridad del Dato que figura como segunda componente).

El método `arreglar`

Para inicializar un Montículo Binario con un conjunto de `N` datos se podría seguir alguna de las dos estrategias siguientes: añadir los N Datos al Montículo utilizando el método `insertar` o almacenar de forma contigua los N Datos sobre `elArray`, actualizando `talla`, y al finalizar modificar la estructura para que se cumpla la propiedad de Montículo. Para completar esta segunda estrategia se debe diseñar un método `arreglar` que debe restablecer la propiedad de orden sobre un AB Completo ya almacenado sobre el array, para ello es suficiente con hundir sus nodos en orden inverso al recorrido por niveles; como no hace falta hacer `hundir` sobre los nodos hojas, porque éstos ya son Montículo, se comenzará a `hundir` el nodo de mayor índice que no sea una hoja. El código Java para este método sería:

```
private void arreglar () {
   for (int i = v.length/2; i > 0; i--) hundir(i);
}
```

A continuación se muestra una traza de `arreglar` sobre los Datos 92, 47, 21, 20, 12, 45, 63, 61, 17, 55, 37, 25, 64, 83, 73.



Para establecer la Función de Complejidad en términos del número de Nodos del Montículo (que viene dado por el valor del atributo `talla`, i.e x = talla), es necesario estudiar la Complejidad temporal del método `arreglar` que viene dada por la suma de las alturas de todos los Nodos del Montículo, que es $O(talla)$. Para establecer este cálculo se recurrirá al siguiente teorema:

> **Teorema:** Dado un Árbol Binario Lleno de altura H que contiene $x = 2^{H+1} - 1$ nodos, se cumple que la suma de las alturas de sus nodos es $x - H - 1$. Dado que un Árbol Binario Completo tiene entre 2^H y $x = 2^{H+1} - 1$ nodos, también la suma de la alturas de sus nodos está acotada por $O(x)$.

Cuestiones y Ejercicios:

1. Comparar los costes de las dos estrategias para la inicialización de un Montículo apuntadas en el apartado anterior.
2. Hacer una traza de `eliminarMin` sobre el Montículo $[0, 1, 4, 8, 2, 5, 6, 9, 15, 7, 12, 13]$.
3. Hacer una traza de `hundir(3)` sobre el Montículo $[0, 1, 4, 8, 2, 5, 6, 9, 15, 7, 12, 13]$.
4. ¿Qué modificaciones se deberían hacer en la clase `MonticuloBinario` si los datos se encuentran en las posiciones del vector comprendidas entre 0 y $talla - 1$?.
5. Dada una Colección de Integer representada como un Montículo Minimal, se pide escribir un método que obtenga su máximo con el mínimo número de comparaciones

6. Sea h un Montículo Minimal vacío. Se pide realizar la traza del proceso de inicialización de h con la siguiente Colección de Integer: 32,26,65,68,19,31,21,13,13,6. Indíquese así mismo el proceso de inicialización seguido y su coste
7. Impleméntese un Montículo Binario Maximal.
8. Diseñar en la clase `MonticuloBinario` los métodos siguientes y estudiar su coste:

   a) El método `borrarHojasEnRango` debe borrar aquellas hojas del Montículo que estén dentro de un rango no vacío [x,y] dado; el método debe tener como máximo un coste Temporal lineal con el número de datos del Montículo. Nota: el orden de los Nodos en el Heap resultante no es importante si sigue siendo Heap.

   b) El método `estaEn` debe comprobar si un Elemento e dado está o no en el Montículo.

   c) El método `igualesAlMinimo` que calcule, con el menor coste Temporal posible, el número de Elementos de un Montículo iguales al mínimo (éste incluido).

   d) El método `eliminarK` para eliminar el k-ésimo dato (por niveles) de un Montículo.

   e) El método `recuperarKMenor` que debe devolver el k-ésimo Dato menor.

   f) El método `comprobarPO` que devuelve true si los elementos depositados en su atributo elArray cumplen la Propiedad de Ordenación de un Montículo Minimal y false en caso contrario.

3. Ordenación rápida según un Montículo

Heap Sort es un método de Ordenación Rápida basado en el principio de Selección que utiliza un Montículo Binario para estructurar el subarray sobre el que se efectúa la Selección. Antes de pasar a la descripción y codificación del método resulta conveniente recordar en qué consiste la estrategia de Ordenación por Selección, estudiada ya en primer curso. Además, y puesto que Heap Sort se implementa sobre un array en el que los Datos se almacenan a partir de la posición 0 y hasta la dimensión del array menos 1, y se utiliza un Montículo Binario Maximal, es necesario plantear las modificaciones que implica el cambio de representación sobre el array, así como las implicaciones que se derivan de la utilización de un Montículo Maximal en lugar de un Minimal, que es lo que se ha implementado hasta el momento.

3.1. La estrategia de selección revisitada

Según la estrategia de Ordenación por Selección, para ordenar ascendentemente las componentes del `array v[0 ... v.length-1]` se procede de forma iterativa recorriéndolo de forma ascendente y seleccionando en cada iteración i (con $0 \leq i < v.length$) el elemento menor de entre los que forman el **subarray** `v[i ... v.length-1]` para colocarlo en la posición que le corresponde que en este caso es i; de este modo, en cada paso de la iteración se tiene que el **subarray** `v[0... i-1]` está ya ordenado y el **subarray** `v[i... v.length-1]` contiene las $v.length - i$ componentes pendientes de ordenar. El proceso de ordenación concluye cuando el subarray ordenado se extiende por todo el array, esto es cuando $i == v.length$. La transcripción a Java de esta estrategia da lugar al conocido método `seleccionDirecta`:

```
public static <T extends Comparable<T>> void seleccionDirecta(T  v[]){
    for ( int i=0; i < v.length-1; i++ ) {
        int posMinimo = buscarMinimo (v, i);
        intercambiar (v, i, posMinimo);
    }
}
```

donde el método `buscarMinimo` debe devolver el índice entre i y $v.length - 1$ cuya componente corresponde al valor mas pequeño. Una implementación equivalente de la estrategia de Selección Directa consiste en realizar un Recorrido Descendente del array Seleccionando en cada paso el elemento mayor; esta implementación se escribiría en Java como sigue:

```
private static <T extends Comparable<T>> void seleccionDirectaMax(T  v[]){
    for ( int i = v.length-1; i >0; i-- ) {
        int posMaximo = buscarMaximo (v, i);
        intercambiar (v, i, posMaximo);
    }
}
```

donde el método `buscarMaximo` debe devolver ahora el índice entre 0 e i cuya componente corresponde al valor mas grande. Ahora, en cada paso de Ordenación se cumple que el **subarray** `v[i... v.length-1]` está ya ordenado y el **subarray** `v[0... i-1]` contiene las i componentes pendientes de ordenar. El proceso de ordenación concluye en este caso cuando el subarray ordenado se extiende por todo el array, esto es cuando $i == 0$.

Como ya se estudió en su momento, la Complejidad Temporal del método de Ordenación por Selección Directa es cuadrática con la talla del array, x=v.length; i.e. $\Theta(x^2)$, que viene

dada por la necesidad de realizar $x-1$ veces el recorrido para buscar el máximo que tiene un coste $\Theta(x)$. Si la zona del array no tratada o pendiente de ordenar **subarray** `v[0 ... i]` se organiza como un Montículo Binario Maximal, el método `buscarMaximo` es concretamente el método `eliminarMax` que como ya se ha estudiado tiene un coste logarítmico con el número de elementos del Montículo, en este caso $O(logx)$. De este modo, el coste de la Ordenación se reduce a $O(x*logx)$ dando lugar al método Rápido de Ordenación según un Montículo o Heap Sort que se pasa a describir de forma inmediata.

3.2. La estrategia Heap Sort

El método Heap Sort supone los datos a ordenar en un cierto array v comenzando en la posición 0 del mismo, de igual modo que se planteó en la implementación del resto de métodos de ordenación estudiados hasta el momento y procede como sigue: inicialmente se construye un Montículo Maximal con los elementos del array y seguidamente se recorre el vector en sentido descendente tal y como se hace en el método `seleccionDirecta`, desde el final hasta la posición 1. Para cada posición i se intercambia el valor máximo, que por definición de Montículo estará en la posición 1, $v[1]$, con el que se encuentra en la posición i; al realizar este intercambio, el valor introducido en la raíz $v[1]$ puede violar la propiedad de ordenación del Montículo y para ajustarlo se debe hundir el hueco generado en la raíz sobre el Montículo que ahora ocupará el **subarray** `v[0 ... i-1]`; y se pasa a tratar la posición anterior con lo que se avanza haciendo decrecer en 1 la zona por ordenar o zona problema e incrementando en 1 la la zona ya ordenada.

Representación alternativa de un Montículo Binario

Tal y como se ha comentado, para tratar el problema de la Ordenación según Montículo de forma homogénea con el resto de métodos de Ordenación es necesario trabajar sobre el array comenzando en la posición 0 y hasta la posición i. Así, si la representación implícita del Montículo comienza en la posición 0 del array, se tiene que, dado un nodo `v[k]`: si $2*k+1 < i-1$, `v[2*k+1]` es el nodo hijo izquierdo; si $2*k+2 < i-1$, `v[2*k+2]` es el nodo hijo derecho y si $k \neq 0$, `v[k/2]` es el nodo padre.

Montículo Binario Maximal

Un Montículo Binario Maximal de define como un Árbol Binario Completo con la siguiente propiedad de orden: para cada nodo X con padre P, se cumple que el dato en P es **mayor o igual** que el dato en X. La propiedad de ordenación de los montículos establece que el dato en el nodo padre nunca es **menor** que el Dato de sus hijos. Si el Árbol Binario Completo se almacena en posiciones consecutivas del array `v` comenzando desde 0, esta propiedad de orden instanciada para el `array v[0 ... i-1]` se puede expresar como $\forall k : 0 < k < i-1 : v[k/2] \geq v[k]$

El método `arreglar` para Montículo Maximal

Si se tiene un `array v[0 ... v.length-1]` y se desea estructurarlo como un Montículo Binario Maximal, se puede proceder de forma análoga a como se hizo en el método `arreglar`; ahora, se debe rediseñar un método análogo a `hundir`, `hundirMax`, pero que trabaje sobre un Montículo Binario Maximal. El método `hundirMax` se define como un método estático y privado de la clase `Ordenación` con tres parámetros: el array sobre el que se trabaja, la posición del

hueco y la del último elemento que forma parte del Montículo Binario Maximal. Su código en Java es:

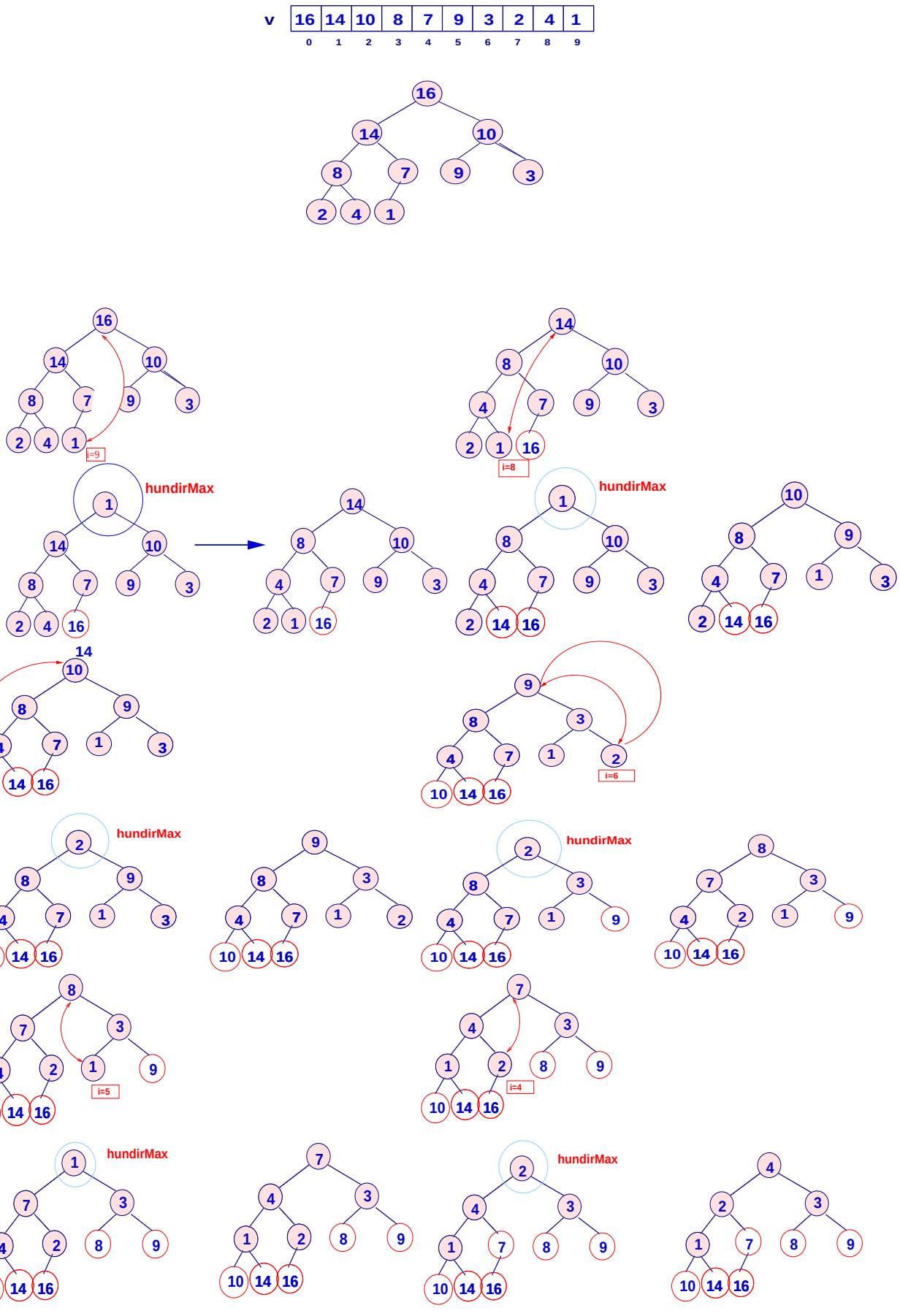
```
private static <T extends Comparable<T>> void hundirMax (T v[], int hueco, int fin) {
   int hijo = hueco*2+1;  T aux = v[hueco];
   boolean esHeap = false;
   while ( hijo<fin  && !esHeap ) {
     if ( hijo != fin-1 && ( v[hijo+1].compareTo(v[hijo])>0 ) hijo++;
     if ( v[hijo].compareTo(aux) > 0 ) {
         v[hueco] = v[hijo]; hueco = hijo; hijo = hueco*2+1;
     } else esHeap = true;
   }
   v[hueco] = aux;
}
```

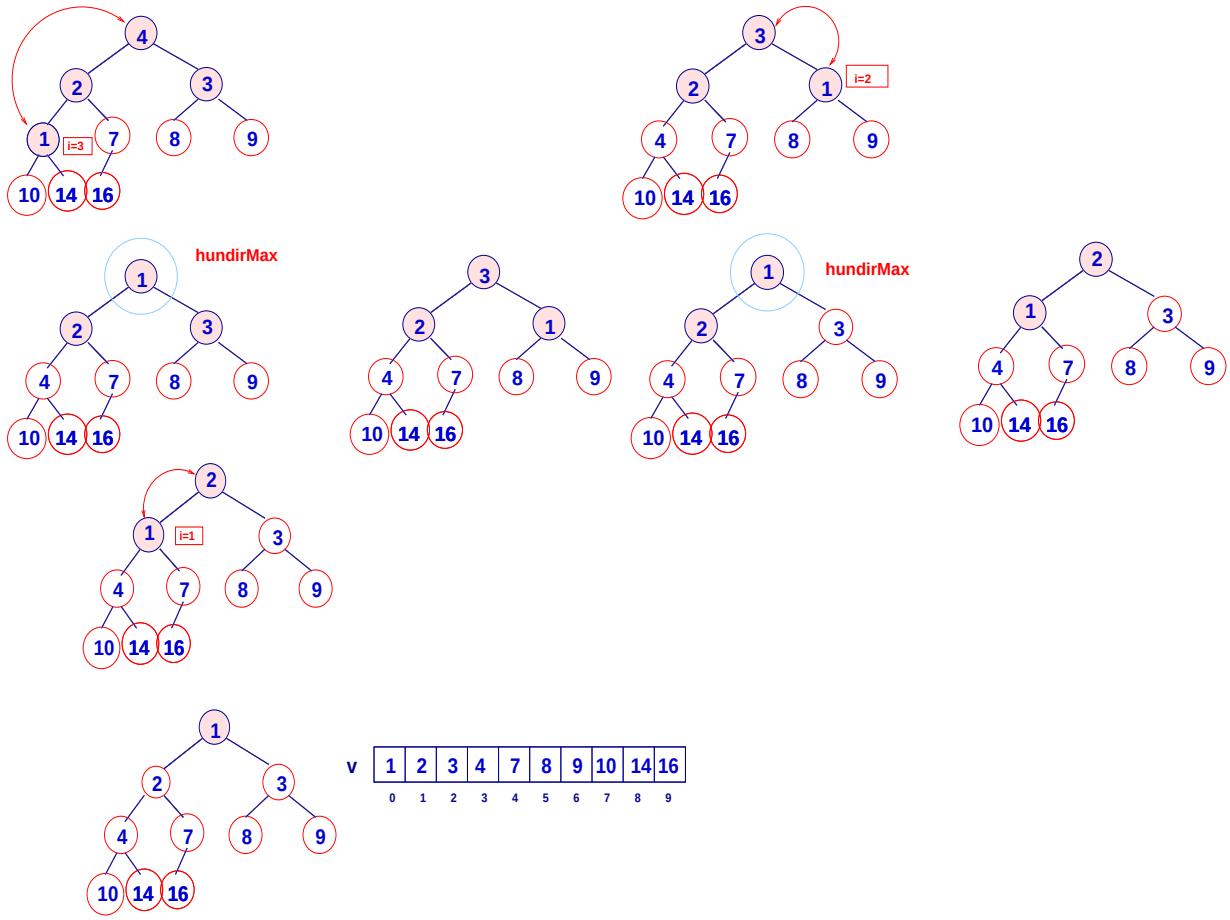
3.3. El método `heapSort` de la clase `Ordenacion` y su coste

En resumen, el método de ordenación via Montículo o `heapSort` se basa en la estructuración de la zona problema como un Montículo o Heap; así se puede encontrar el máximo con coste logarítmico; inicialmente la zona problema se extiende sobre todo el array y esta es la razón por la que la primera acción del método de Ordenación `heapSort` será `arreglarMax`. A continuación se realiza un proceso iterativo en el que en cada paso se intercambia el último Dato del Montículo (el situado en la posición `i`) con el de la Raíz; de este modo se ordena este dato, a continuación se debe hundir el hueco desde la posición `0` hasta el final del Montículo, que viene determinado por `i-1`. El código en Java para el método `heapSort` quedaría:

```
package librerias.util.operacionesArray;
public class Ordenacion {
   ...
   public static <T extends Comparable<T>> void heapSort (T v[]) {
     for ( int i = v.length/2; i > 0; i-- ) hundirMax (v, i, v.length);
     // v[0..v.length-1] es un Montículo
     for (int i = v.length-1; i > 0; i--) {
        //v[0..i] es un Montículo, zona problema
        //v[i+1..v.length-1] está ordenado de forma creciente con los datos mayores de v
        intercambiar(v,0,i);
        hundirMax(v,0,i);
        //i forma parte de la zona ordenada
     }
   }
   private static <T extends Comparable<T>> void hundirMax (T v[], int hueco, int fin) { ... }
}
```

A continuación se muestra una traza del método `heapSort`, en la primera figura se muestra el contenido del array y su representación en forma de Árbol después de construir el Montículo Binario Maximal inicial; el contenido del array en ese momento es $[16, 14, 10, 8, 7, 4, 3, 2, 4, 1]$. A partir de esta situación inicial se realiza el bucle `for` de `heapSort` comenzando con $i = 9$ y terminando cuando $i == 0$.


v
16 14 10 8 7 9 3 2 4 1
0 1 2 3 4 5 6 7 8 9
i=9
hundirMax
i=8
hundirMax
i=7
i=6
hundirMax
i=5
i=4
hundirMax
hundirMax



Análisis de la Complejidad Temporal de `heapSort`

La complejidad temporal del método **`heapSort`** viene determinada por el número de veces que se realiza el método **`hundirMax`**, que como se ha explicado, tiene un coste logarítmico con el número de nodos del árbol sobre el que se efectúa *hundir* el hueco; nótese que las llamadas a **`hundirMax`** desde el segundo bucle **`for`** siempre se realizan desde la raíz y sobre un Montículo de tamaño i; así el coste de cada una de estas llamadas será precisamente $\Theta(logi)$. Por lo tanto, si se denota por **`x`** el número de Datos a ordenar o longitud del array, se tiene que la función de coste para **`heapSort`** es la siguiente: $\sum_{i=0}^{i=x-1} logi \in \Theta(x * logx)$.

Cuestión: ¿Es heapSort un método de Ordenación estable? Es decir, si hay Datos duplicados ¿retienen dichos Datos la ordenación inicial entre ellos?