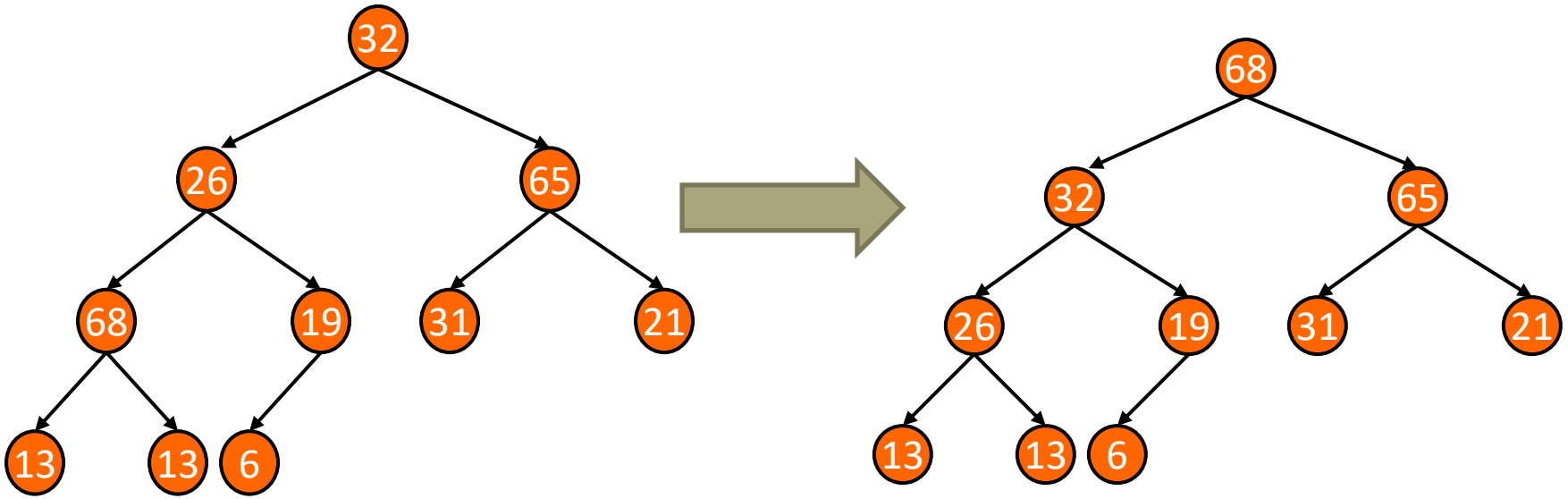
1) Ya sabemos cómo construir maxHeap a partir de un vector



	1	2	3	4	5	6	7	8	9	10	
	32	26	65	68	19	31	21	13	13	6	....

Array talla=10

	1	2	3	4	5	6	7	8	9	10	
	68	32	65	26	19	31	21	13	13	6	....

maxHeap talla=10

2) ¿Cómo podemos aprovechar ahora el heap para ordenar la secuencia?

4. HeapSort – *Ordenación rápida según HeapSort*

- El coste de HeapSort es $O(N*\log_2 N)$
  - *QuickSort* tiene un coste $O(N^2)$ en el peor de los casos
  - *MergeSort* requiere un vector auxiliar
- Este algoritmo de ordenación se basa en las propiedades de los Heaps
  - Primer paso: almacenar todos los elementos del vector a ordenar en un montículo (heap)
  - Segundo paso: extraer el elemento raíz del montículo (el mínimo) en sucesivas iteraciones, obteniendo el conjunto ordenado

4. HeapSort - *Inserción de los datos del vector en el Heap*

- La forma más eficiente de insertar los elementos de un vector en un *Heap* es mediante el método *arreglarMonticulo*:

```
@SuppressWarnings("unchecked")      // Constructor a partir de un vector
public MonticuloBinario(E v[]) {
  talla = v.length;                 // Copiamos los datos del vector
  elArray = (E[]) new Comparable[talla+1];
  System.arraycopy(v, 0, elArray, 1, talla);
  arreglarMonticulo();              // Arreglamos la propiedad de orden
}
```

- El coste de este constructor es O(N), siendo N la talla del vector

4. HeapSort – *Algoritmo*

```
public class Ordenacion {
  public static <E extends Comparable<E>> void heapSort(E v[]) {
    // Creamos el heap a partir del vector
    MonticuloBinario<E> heap = new MonticuloBinario<E>(v);
    // Vamos extrayendo los datos del heap de forma ordenada
    for (int i = 0; i < v.length; i++)
      v[i] = heap.eliminarMin();
  }
}
```

- Coste HeapSort = coste constructor + N * coste de *eliminarMin*

  $T_{heapSort}(N) \in O(N) + N*O(\log_2 N) = O(N*\log_2 N)$

- HeapSort puede modificarse fácilmente para ordenar sólo los *k* primeros elementos del vector con coste $O(N + k*\log_2 N)$