Estructura de Datos

Semana 7

Logro de la sesión

Al finalizar la unidad, el estudiante identifica, analiza y resuelve problemas algorítmicos que usan el tipo abstracto de datos: colas con prioridad, desarrollando funciones/métodos que usen esas estructuras.

Agenda

- Utilización de colas prioritarias
- El tipo abstracto de dato Cola de prioridad
- Implementación de una cola prioritaria
- Ordenamiento y comparadores

Utilización de colas prioritarias

Problemas de priorización

- **Trabajos de impresión:** las impresoras del laboratorio aceptan y completan constantemente trabajos de todo el edificio. Queremos imprimir los trabajos de los profesores de la facultad antes que los trabajos de los estudiantes, y los estudiantes graduados antes que los de pregrado, etc.

- **Sala de emergencias:** Programación de pacientes para el tratamiento en la sala de emergencias. Una víctima de un disparo debe recibir tratamiento antes que un hombre con un resfriado, independientemente de la hora de llegada. ¿Cómo elegimos siempre el caso más urgente cuando siguen llegando nuevos pacientes?

- *Operaciones clave que queremos:*
  - ***añadir (add)*** *un elemento (trabajo de impresión, paciente, etc.)*
  - ***Obtener(get)/eliminar(remove)*** *el elemento* ***más "importante"*** *o "urgente"*

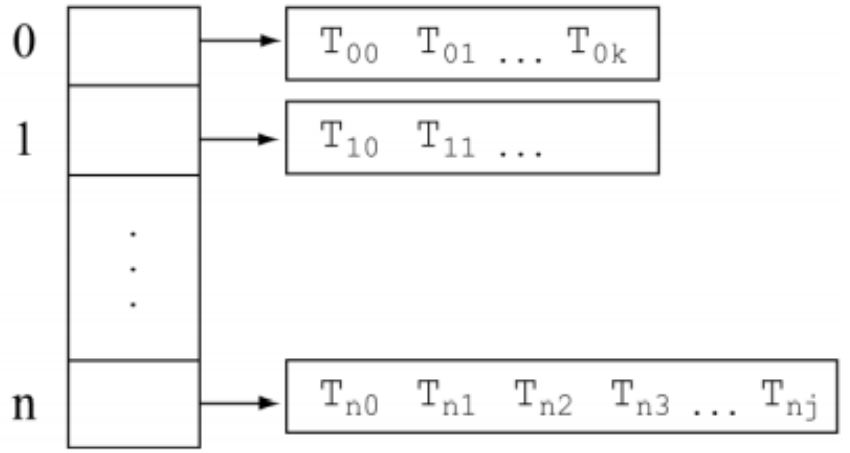
Ejemplos

Somos un banco y queremos gestionar una colección de la información de nuestros clientes:

- *Realizaremos muchas adiciones y eliminaciones*
- *Nos gustaría poder solicitar a la colección que elimine u obtenga la información sobre el cliente con el saldo mínimo/máximo de la cuenta bancaria*

Otro ejemplo: un servidor Unix compartido tiene una lista de trabajos de impresión para imprimir:

- *Quiere imprimirlos en orden cronológico, pero cada trabajo de impresión también tiene una prioridad, y los trabajos de mayor prioridad siempre se imprimen antes que los de menor prioridad*
- *los trabajos más importantes primero o los trabajos más cortos primero*



Cola de prioridades, de 0 a n prioridades

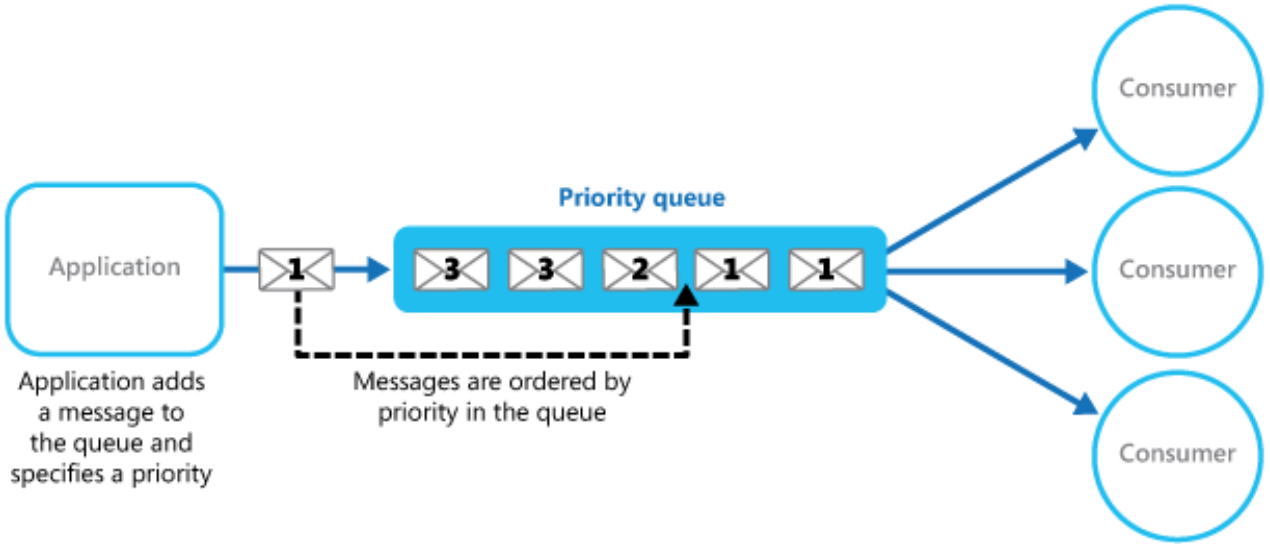
Ejemplos

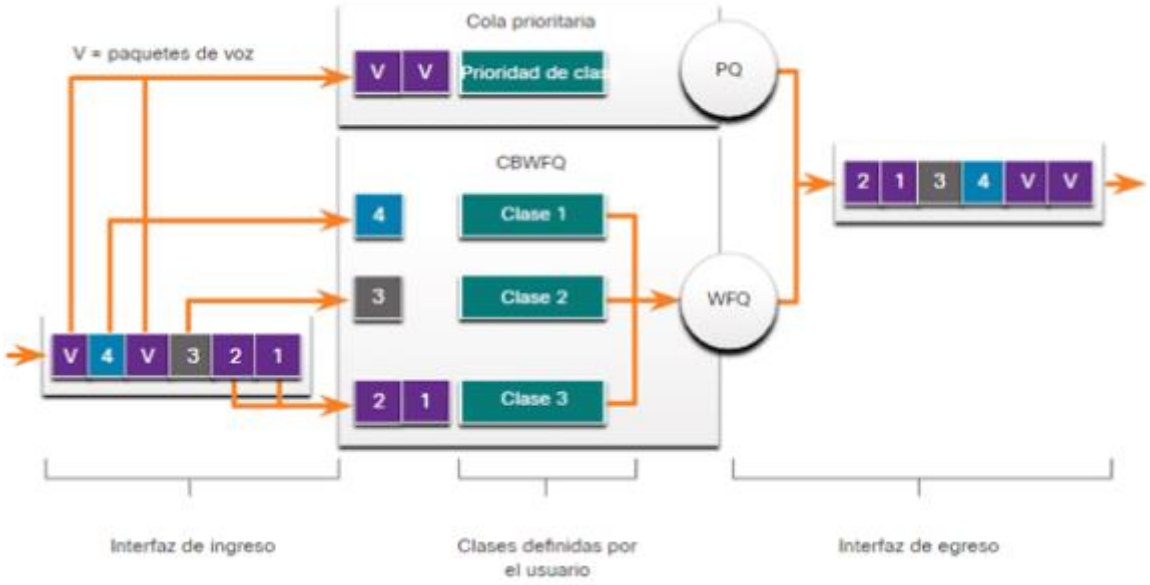
Otro ejemplo: estamos escribiendo un algoritmo de IA fantasma para Pac-Man:

- *Se necesita buscar el mejor camino para encontrar al Pac-Man; pondrá en cola todas las rutas posibles con prioridades (basadas en conjeturas sobre cuál tendrá éxito) y las probará en orden*

la gestión de un planificador de tareas en un Sistema MultiUsuario:

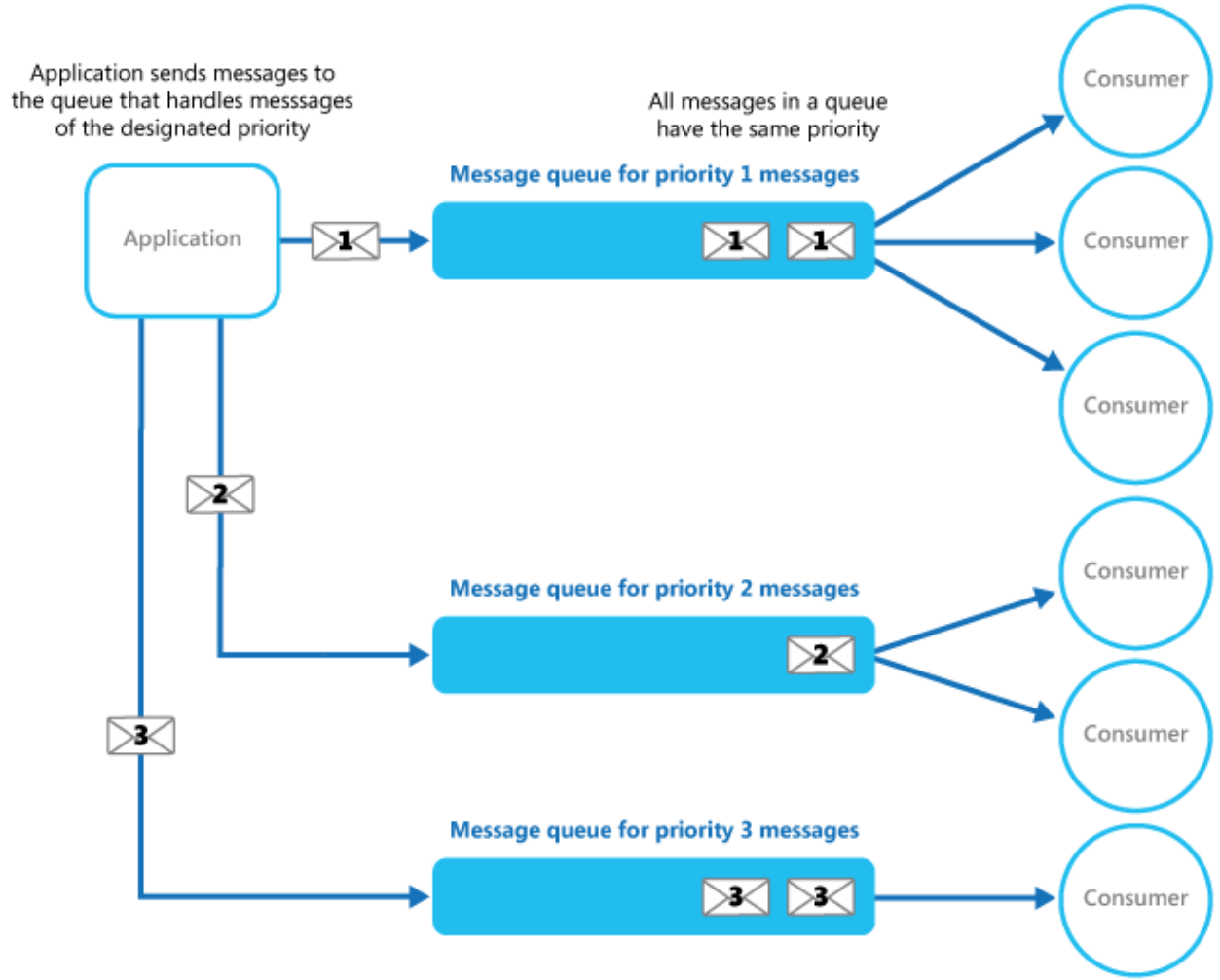
- *los trabajos que consumen menos recursos*
- *los trabajos del administrador del sistema*





Planificación de ejecución de procesos por el CPU

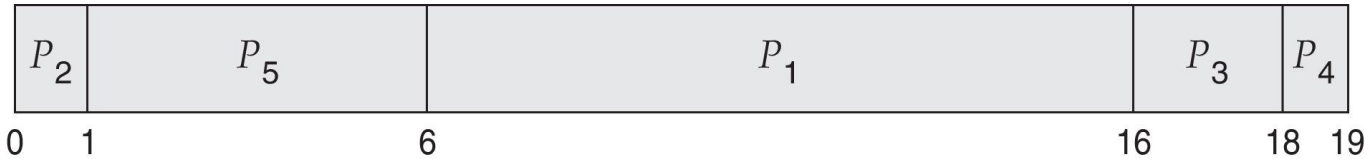
- Un número de prioridad (entero) está asociado con cada proceso.
- La CPU se asigna al proceso con la prioridad más alta (número entero más pequeño ≡ prioridad más alta)
  - Apropiativo
  - Cooperativo o no apropiativo
- Problema ≡ **Bloqueo indefinido o muerte por inanición (Starvation)** – es posible que los procesos de baja prioridad nunca se ejecuten
- Solución ≡ **Envejecimiento** – a medida que pasa el tiempo, aumenta la prioridad del proceso.



Ejemplo de Planificación por prioridades

Proceso	Tiempo de ráfaga	Prioridad
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Diagrama de Gantt de la planificación por prioridades



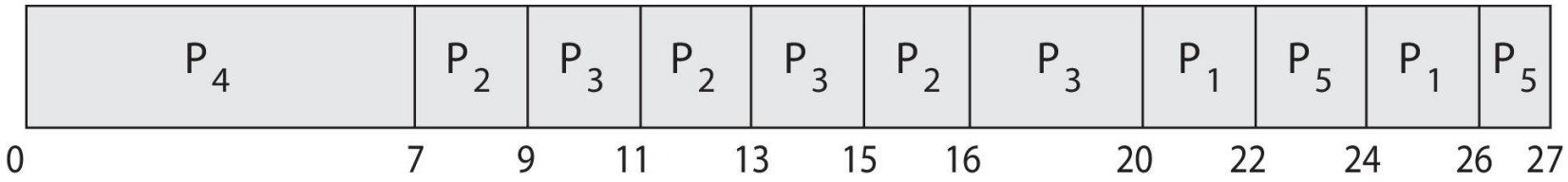
- Tiempo promedio de espera = 8.2 msec

Planificación por prioridades con Round-Robin

Proceso	Tiempo de ráfaga	Prioridad
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

- Ejecute el proceso con la máxima prioridad. Los procesos con la misma prioridad se ejecutan por turnos

- Diagrama de Gantt con cuanto de tiempo (q) de 2 ms



Ejemplo de Planificación por prioridades con Round-Robin

L	A		E		B				C				D							
U	0		1		2		3		4		5		6		7		8		9		10
E		A		E		B		B		C		B		D		D		D		D	

L																				
U		11		12		13		14		15		16		17		18		19		20
E	D		D		D		C		B		C		B		C		C		C	

L																				
U		21		22		23		24		25		26		27		28		29		20
E	C		C		A		E		A		E		A							

q=1

					T.Espera		
Proceso	T.Llegada	t	Prioridad	T.Finaliza	T	W	P
A	0	4	3	27	27	23	6.75
B	2	5	2	17	15	10	3
C	4	8	2	22	18	10	2.25
D	6	7	1	13	7	0	1
E	1	3	3	26	25	22	8.333333333
Promedios					18	13	4.266666667

Ejemplo de Planificación por prioridades

L	A		B			C									D		
U	0		1		2		3		4		5		6		7		8		9		10
E																					

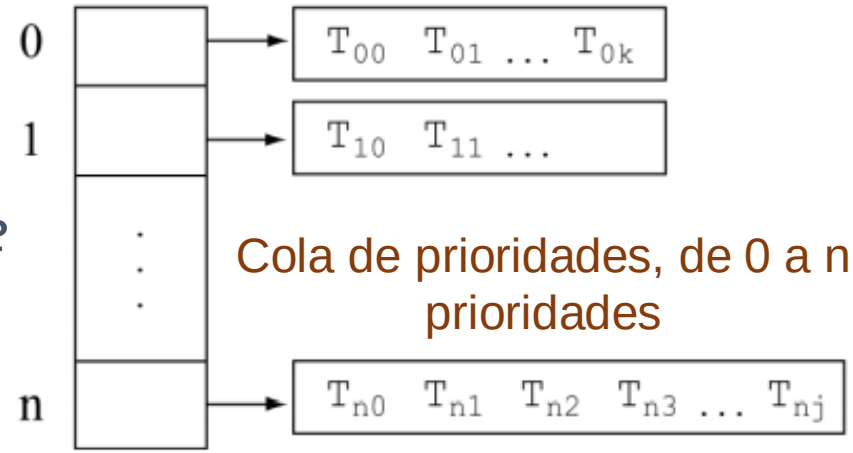
L				E															
U		11		12		13		14		15		16		17		18		19		20
E																			

Proceso	Tiempo Llegada	t	Prioridad	Tiempo Finalizac.	T	W	P
A	0	3	4	10	10	7	3.3
B	1	5	2	8	7	2	1.4
C	3	2	0	5	2	0	1.0
D	9	5	4	15	6	1	1.2
E	12	5	4	20	8	3	1.6
Promedios					6.6	4.3	1.7

Implementación de una cola de prioridad

Lo que se desea es construir y mantener una estructura de datos que contenga registros con claves numéricas (prioridades) y que cuente con algunas de las operaciones siguientes:

- *Construir una cola de prioridad a partir de N elementos*
- *Insertar un nuevo elemento*
- *Suprimir el elemento más prioritario*
- *Cambiar la prioridad de un elemento*
- *Unir dos colas de prioridad en una más grande*



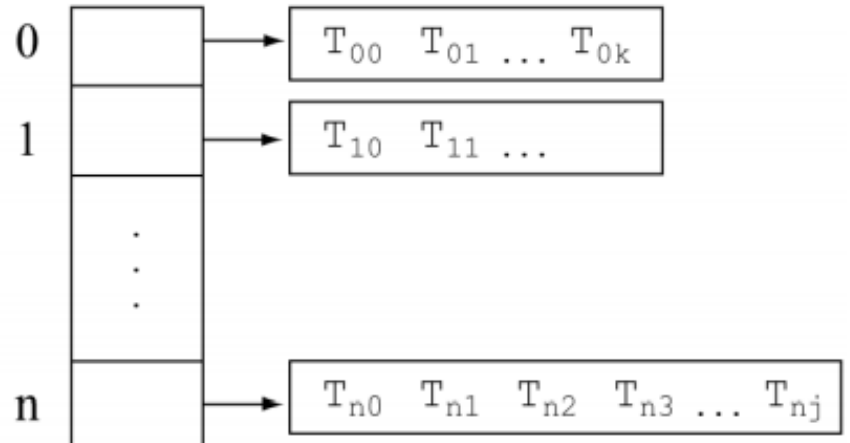
Las operaciones más importantes en una cola de prioridades se refieren aquellas que permiten repetidamente seleccionar el elemento de la cola de prioridad que tiene como clave el valor mínimo (ó máximo, según como hayamos definido quien es más prioritario).

Implementación de una cola de prioridad

Esto conlleva a que una cola de prioridad P debe soportar al menos las siguiente operaciones:

ColaPrioridad(T)
Insertar(P,x): añade el elemento **x** a la cola de prioridad **EncontrarMin(P):** Devuelve el elemento de **P** con la prioridad con menor valor. **EliminarMin(P):** Quita y devuelve el elemento con la prioridad con menor valor.

Cola de prioridades, de 0 a n prioridades



Algunos tipos de implementación (no eficientes)

- *Listas*: almacene la info en una lista desordenada, elimine el mínimo/máximo buscándolo
  - Problema: la búsqueda es costosa
- *Lista ordenada*: almacene todo en una lista ordenada, luego búsquelo en el tiempo O (log n) con búsqueda binaria
  - Problema: la inserción/eliminación es costosa
- *Arbol de búsqueda binaria (Binary Search Tree)*: almacenar en un BST, buscarlo en el tiempo O (log n) para el elemento min (más a la izquierda)
  - Problema: el árbol podría estar desbalanceado
- *BST Balanceado*
  - Problema: en la práctica, si el programa tiene muchas inserciones/eliminaciones, funciona lentamente en árboles AVL y otras BST balanceados
  - Problema: la eliminación siempre se produce desde el lado izquierdo

Implementación en Java: Priority queue

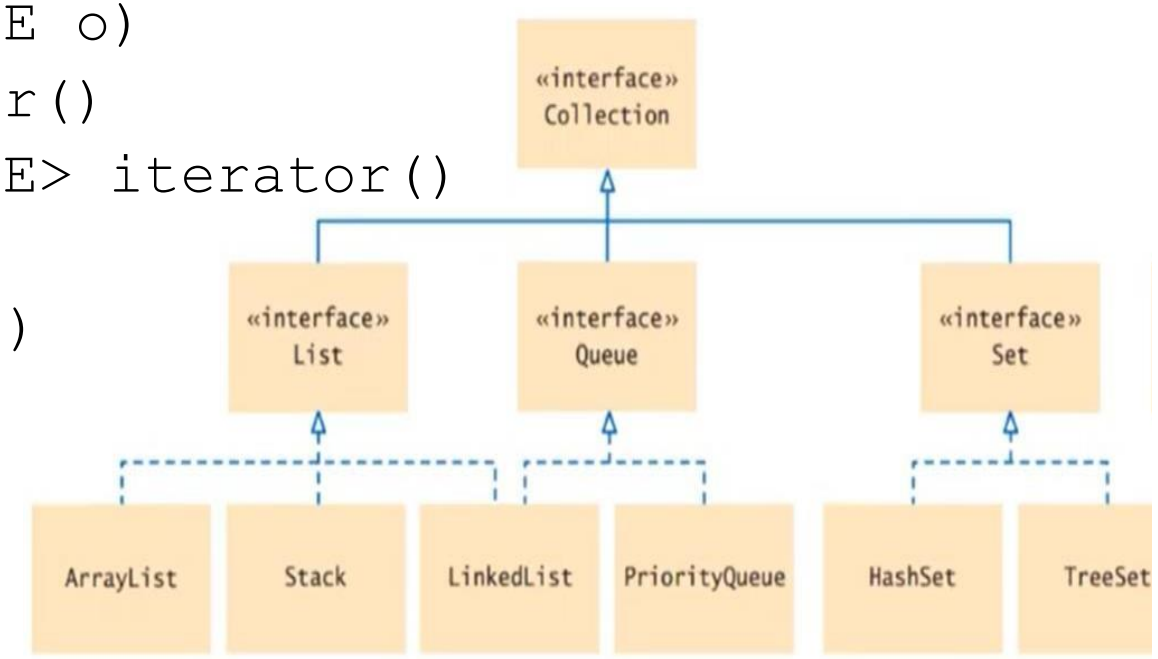
- **Priority queue**: una colección de elementos ordenados que proporciona acceso rápido al elemento mínimo (o máximo)
  - una mezcla entre una cola y un BST (binary search tree, árbol de búsqueda binaria)
  - generalmente implementado usando una estructura llamada *montículo (heap)*

- Operaciones con **priority queue**:
  - `add`
    - O(1) promedio, O(log *n*) peor-caso
  - `peek` - devuelve el elemento **mínimo**
    - O(1)
  - `remove` - devuelve y elimina el elemento **mínimo**
    - O(log *n*) peor-caso
  - `isEmpty`, `clear`, `size`, `iterator`
    - O(1)

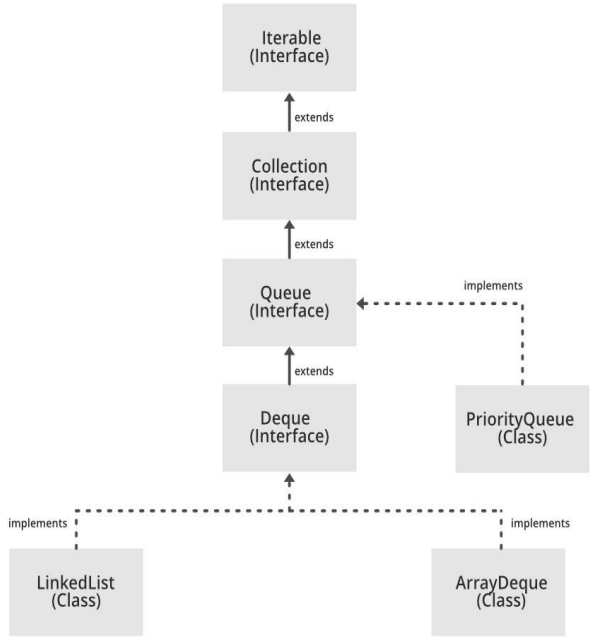
Clase Java Priority Queue

`public class PriorityQueue<E> implements Queue<E>`

- `public PriorityQueue<E>()`
  - Crea una cola de prioridad con la capacidad inicial predeterminada (11) que ordena sus elementos según su ordenamiento natural
- `public PriorityQueue (Comparator <? Super E> comparator)`
  - Crea una PriorityQueue con la capacidad inicial predeterminada y cuyos elementos se ordenan según el comparador especificado.
- `public PriorityQueue<E>(int capacity, Comparator<E> comp)`
  - Crea una cola de prioridad con la capacidad inicial especificada que ordena sus elementos según el comparador especificado.
- `public void add(E o)`
- `public void clear()`
- `public Iterator<E> iterator()`
- `public E peek()`
- `public E remove()`



Interfaces y clases en Java Collection



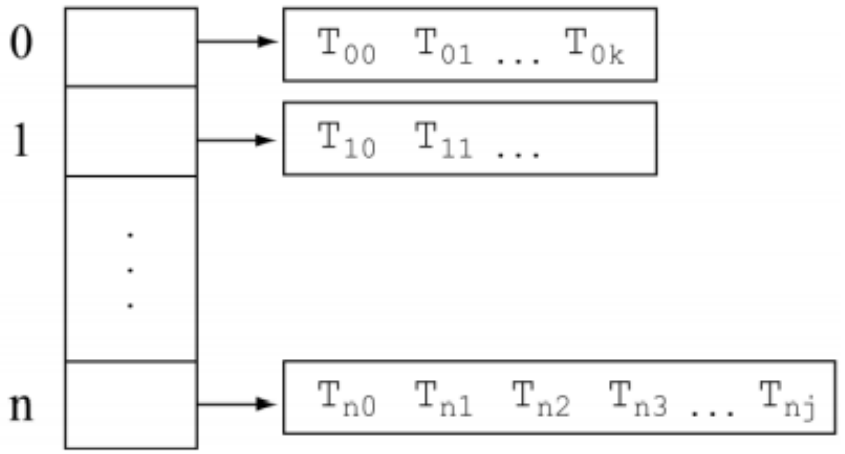
Implementación con tabla de prioridades

Quizás la forma más intuitiva de implementar una *cola de prioridades* es con una tabla o *array* de listas, donde cada elemento de la tabla se organiza a la manera de una cola *(primero en entrar-primero en salir)* y representa a los objetos con la misma prioridad. La Figura . muestra una tabla en la que cada elemento se corresponde con una prioridad y de la que emerge una cola.

La operación `inserEnPrioridad` añade un nuevo elemento `T` de prioridad `m` a la estructura, siguiendo estos pasos:

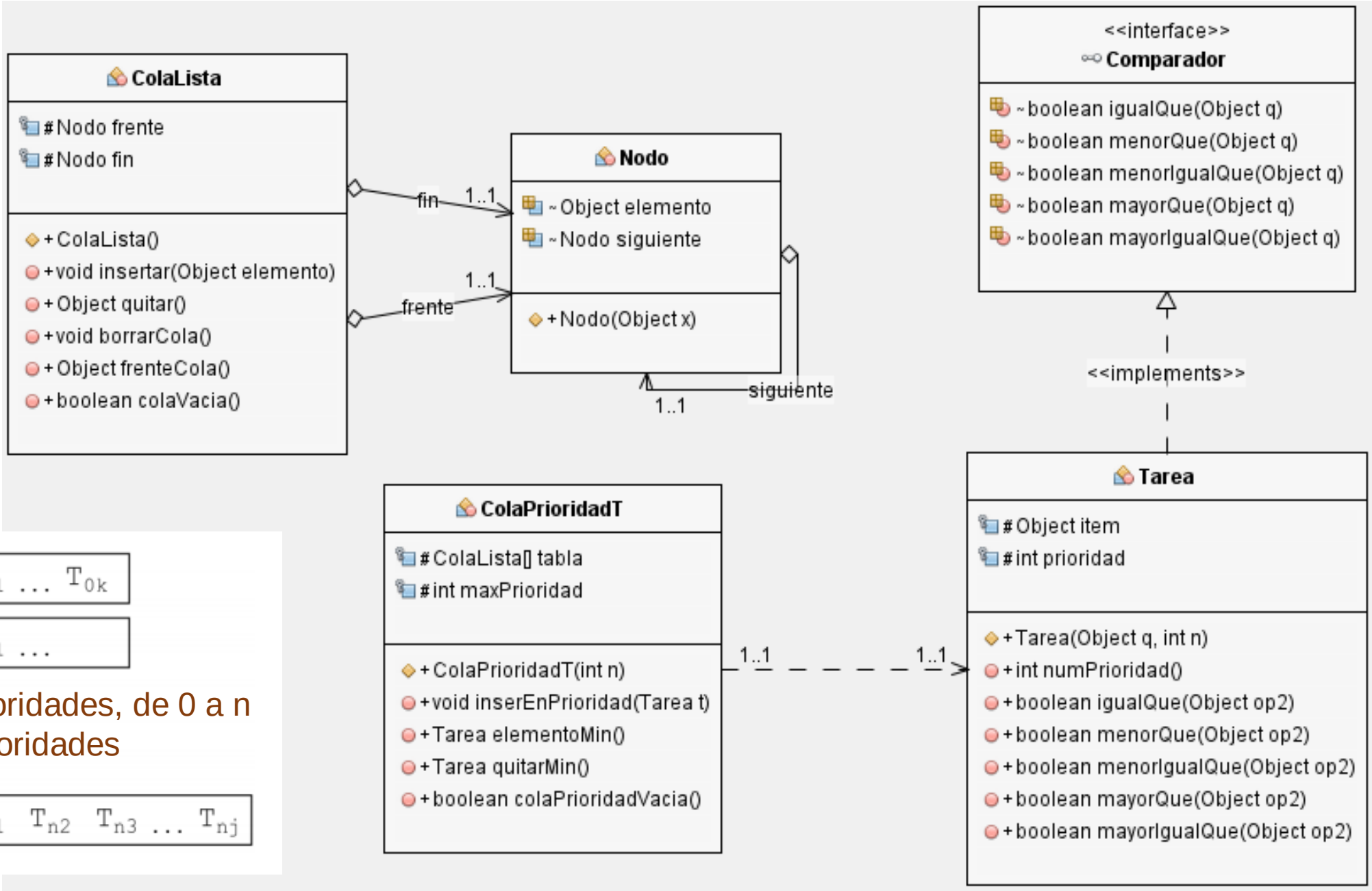
1. Buscar la prioridad `m` en la tabla.
2. Si existe, poner el elemento `T` al final de la cola `m`.
3. Si la prioridad `m` está fuera del rango de prioridades, generar un error

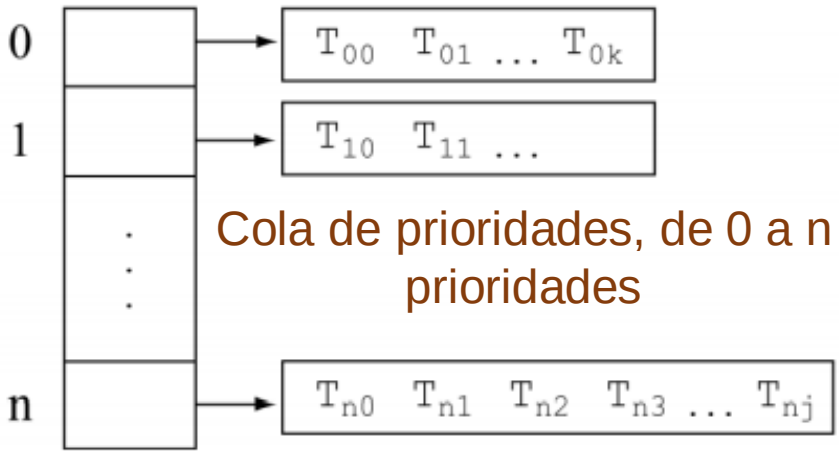
La operación `elementoMin` devuelve el elemento `frente` de la cola no vacía que tiene la máxima prioridad. La operación `quitarMin` también devuelve el elemento de máxima prioridad y, además, lo extrae de la cola.



Cola de prioridades, de 0 a n prioridades

Implementación con tabla de prioridades





Cola de prioridades, de 0 a n prioridades

Implementación con vector de prioridades

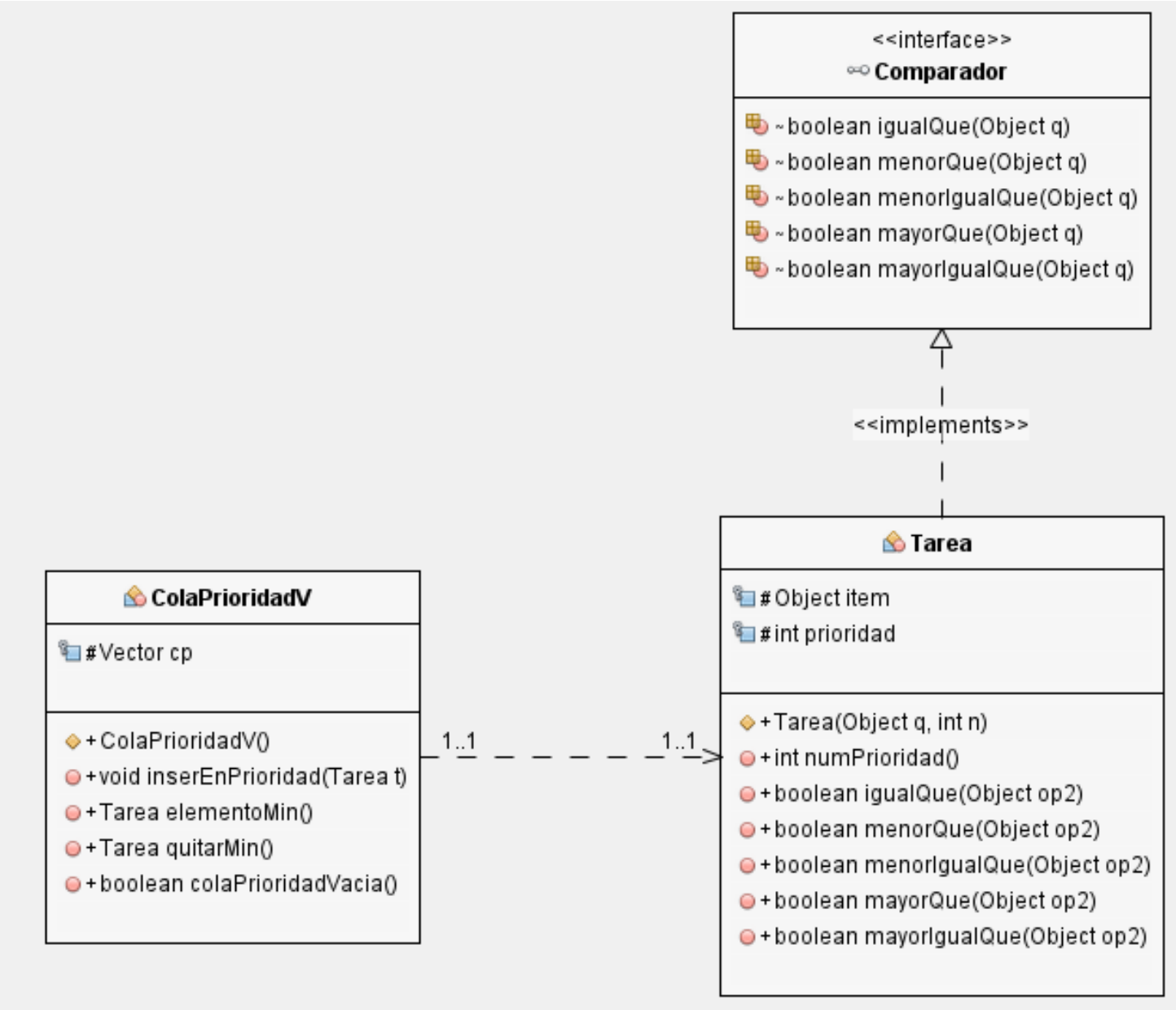
La forma mas sencilla de implementar una *cola de prioridades* es mediante un `Vector` de objetos *(Tareas)* ordenado respecto a la prioridad del objeto. El vector se organiza de tal forma que un elemento *x* precede a *y* si:

1. *Prioridad(x)* es mayor que *Prioridad(y)*.
2. Ambos tienen la misma prioridad, pero *x* se añadió antes que *y*.

Con esta organización, siempre el primer elemento del vector es el elemento de la cola de prioridades de máxima prioridad y el último elemento es el de menor prioridad y, por consiguiente, el último a procesar.

La clase que se declara para representar la *cola de prioridades* define el vector, y su constructor inicializa la estructura. Ahora, no es necesario establecer el número máximo de prioridades.

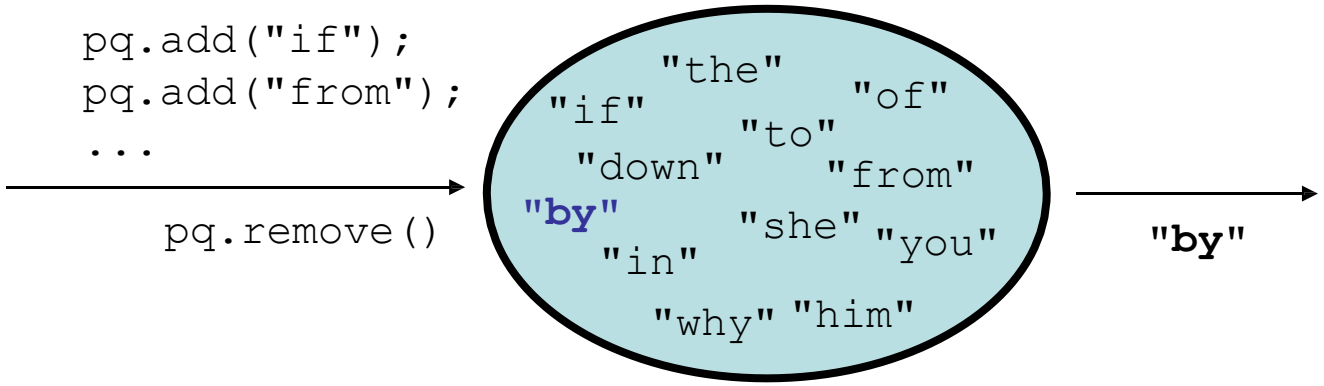
Implementación con vector de prioridades



TAD Cola de Prioridad

TAD de cola de prioridad

- **cola de prioridad** : una colección de elementos ordenados que proporciona un acceso rápido al elemento mínimo (o máximo).
  - `add` agrega en orden
  - `peek` devuelve el valor **mínimo** o de "prioridad más alta"
  - `remove` elimina/devuelve el valor **mínimo**
  - `isEmpty` , `clear` , `size` , `iterador` O(1)



priority queue (cola de prioridad)

Arreglo vacío?

- Considere usar un arreglo vacío para implementar una cola de prioridad.
  - `add` : Guárdelo en el siguiente índice disponible, como en una lista.
  - `peek` : Recorra los elementos para encontrar el elemento mínimo.
  - `remove` : Bucle sobre los elementos para encontrar min. mover al eliminar.

```
queue.add(9);
queue.add(23);
queue.add(8);
queue.add(-3);
queue.add(49);
queue.add(12);
queue.remove();
```

índice	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
valor	9	23	8	**-3**	49	12	0	0	0	0
size	6									

  - ¿Qué tan eficiente es `add`? `peek`? `remove`?
    - O(1), O(N), O(N)
    - (`peek` debe recorrer la matriz; `remove` debe mover elementos)

Arreglo ordenado?

- Considere usar una matriz *ordenada* para implementar una cola de prioridad.
  - `add`: Guárdelo en el índice adecuado para mantener el orden.
  - `peek`: el elemento mínimo está en el índice [0].
  - `remove`: Desplazar elementos para eliminar min del índice [0].

```
queue.add(9);
queue.add(23);
queue.add(8);
queue.add(-3);
queue.add(49);
queue.add(12);
queue.remove();
```

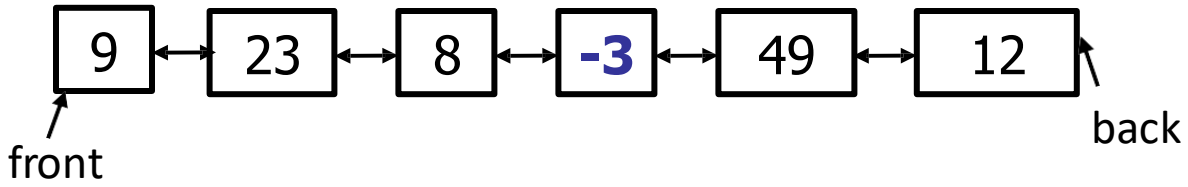
índice	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
valor	**-3**	8	9	12	23	49	0	0	0	0
size	6									

  - ¿Qué tan eficiente es `add`? `peek`? `remove`?
    - O(N), O(1), O(N)
- (`add` y `remove` debe mover elementos)

¿Lista enlazada?

- Considere usar una lista doblemente enlazada para implementar una cola de prioridad.
  - `add`: Guárdelo al final de la lista enlazada.
  - `peek`: Recorra los elementos para encontrar el elemento mínimo.
  - `remove`: Bucle sobre los elementos para encontrar min. Desvincular para eliminar.

```
queue.add(9);
queue.add(23);
queue.add(8);
queue.add(-3);
queue.add(49);
queue.add(12);
queue.remove();
```

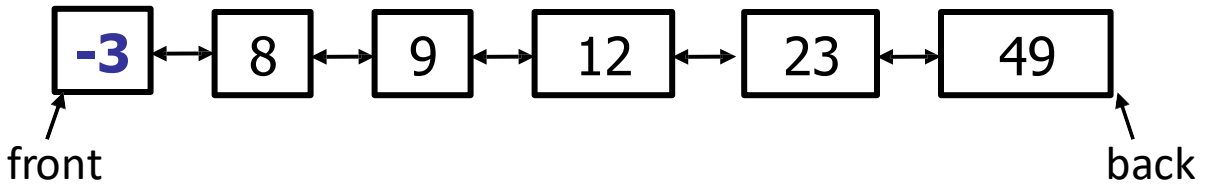


  - ¿Qué tan eficiente es `add`? `peek`? `remove`?
    - O(1), O(N), O(N)
    - (`peek` y `remove` debe recorrer la lista enlazada)

¿Lista enlazada ordenada?

- Considere usar una lista enlazada *ordenada* para implementar una cola de prioridad.
  - `add` : guárdelo en el lugar adecuado para mantener el orden.
  - `peek` : El elemento mínimo está en la parte delantera.
  - `eliminar` : desvincular el elemento frontal para eliminar.

```
queue.add(9);
queue.add(23);
queue.add(8);
queue.add(-3);
queue.add(49);
queue.add(12);
queue.remove();
```

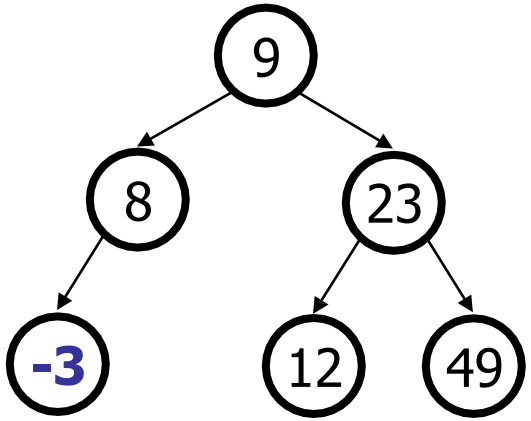


  - ¿Qué tan eficiente es `add`? `peek`? `remove`?
    - O(N) , O(1), O(1)
    - (`add` debe recorrer la lista enlazada para encontrar el punto de inserción adecuado)

¿Árbol de búsqueda binario?

- Considere usar un árbol de búsqueda binario para implementar un PQ (priority queue).
  - `add`: Guárdelo en el lugar adecuado BST L/R - ordenado.
  - `peek`: el elemento mínimo está en el extremo izquierdo del árbol.
  - `remove`: desvincular el elemento del extremo izquierdo para eliminarlo.

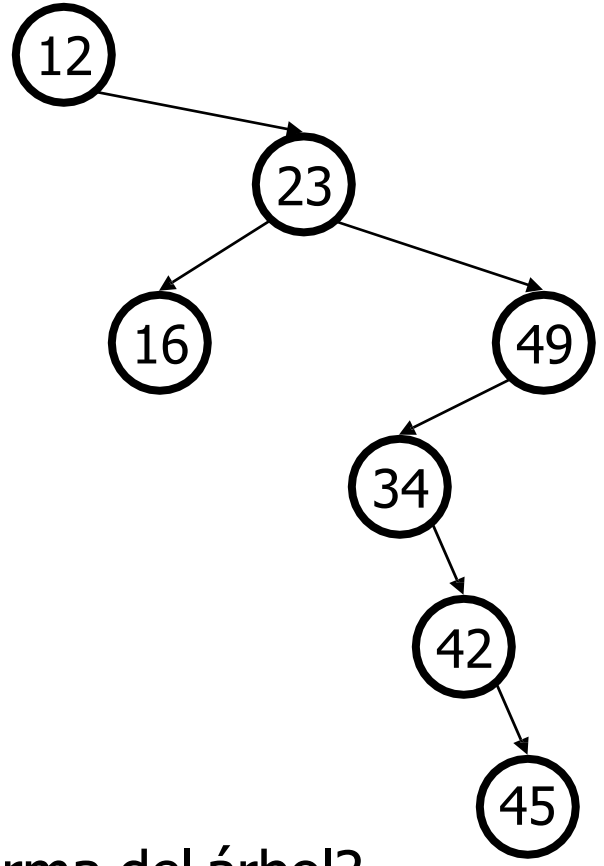
```
queue.add(9);
queue.add(23);
queue.add(8);
queue.add(-3);
queue.add(49);
queue.add(12);
queue.remove();
```



  - ¿Qué tan eficiente es `add`? `peek`? `remove`?
    - O(log *N*), O(log *N*), O(log *N*)...?
    - (bueno en teoría, pero el árbol tiende a desequilibrarse hacia la derecha)

Árbol binario desequilibrado

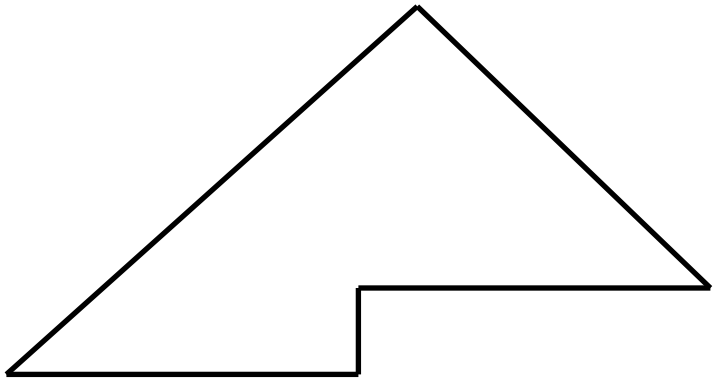
```
queue.add(9);
queue.add(23);
queue.add(8);
queue.add(-3);
queue.add(49);
queue.add(12);
queue.remove();
queue.add(16);
queue.add(34);
queue.remove();
queue.remove();
queue.add(42);
queue.add(45);
queue.remove();
```

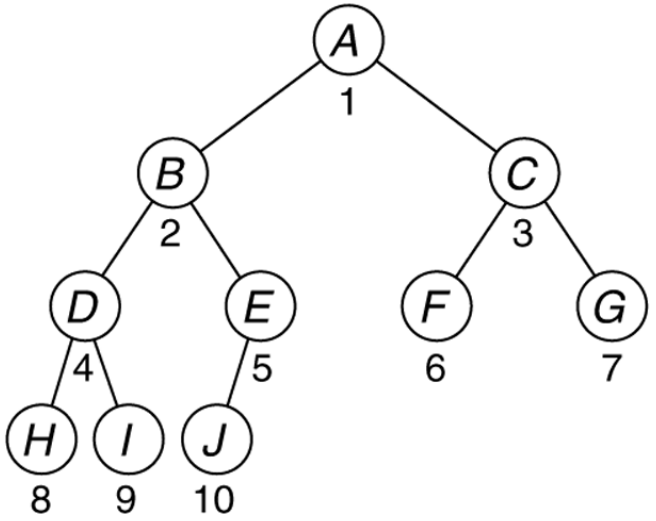


- Simule estas operaciones. ¿Cuál es la forma del árbol?
- Un árbol que está *desequilibrado* tiene una altura cercana a *N* en lugar de log *N*, lo que rompe el tiempo de ejecución esperado de muchas operaciones.

Montículos

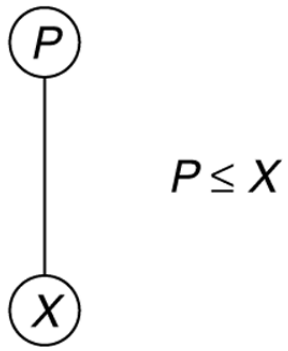
- **montículo**: un árbol binario *completo con* ordenamiento *vertical .*
  - **Árbol completo** : todos los niveles están llenos, excepto posiblemente el nivel más bajo, que debe completarse de izquierda a derecha.
    - (es decir, un nodo puede no tener hijos hasta que existan todos los hermanos posibles)



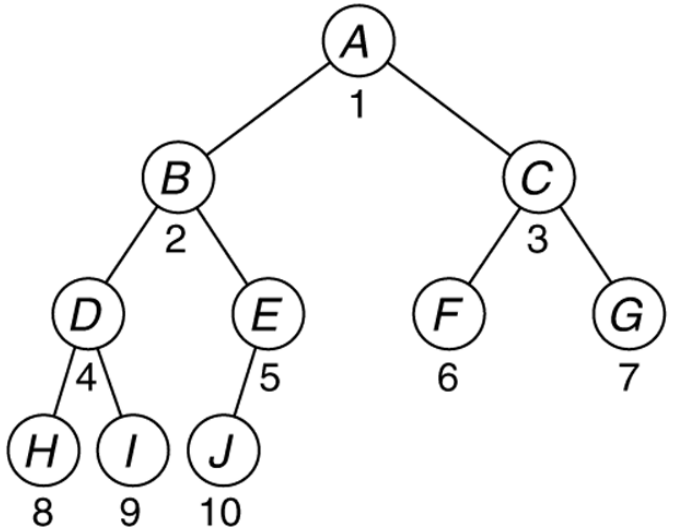


Ordenamiento de un montículo

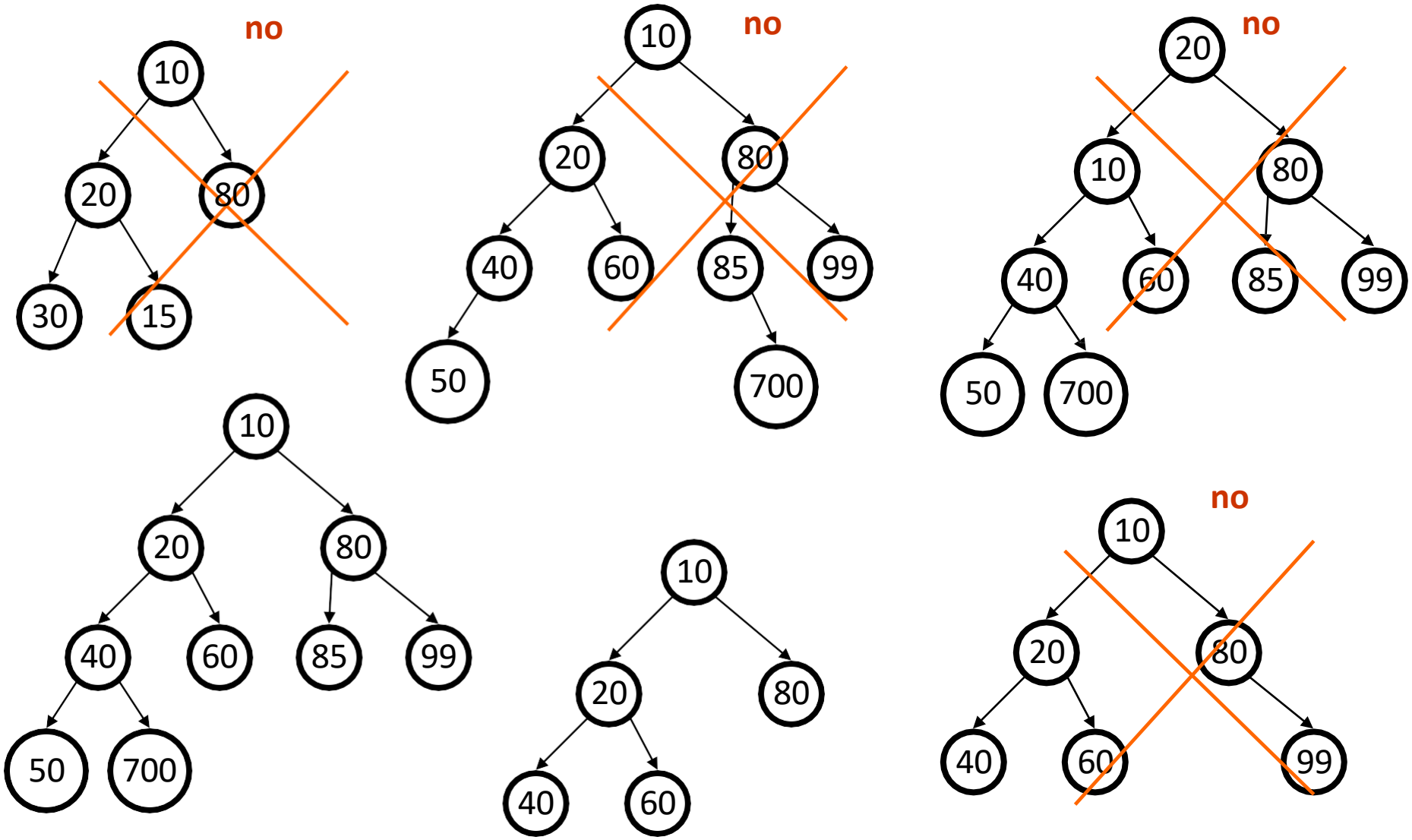
- **Ordenamiento de un montículo (Heap ordering)** : si P ≤ X para cada elemento X con padre P.
  - Los valores de los padres son siempre menores que los de sus hijos.
  - Implica que el elemento mínimo es siempre la raíz (un "min-heap").
    - variación: "max-heap" almacena el elemento más grande en la raíz, orden inverso
  - ¿Es un montículo un BST? ¿Como están relacionados?



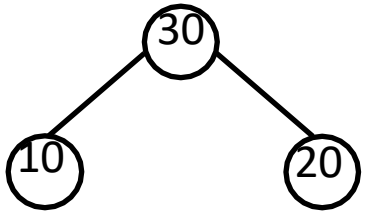
$P \leq X$

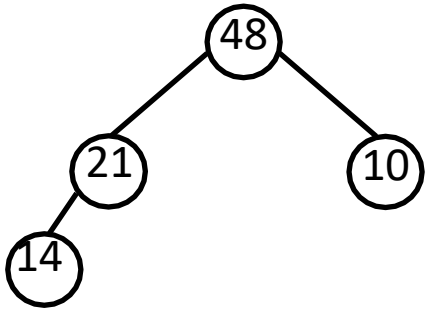


¿Cuáles son min-heaps?

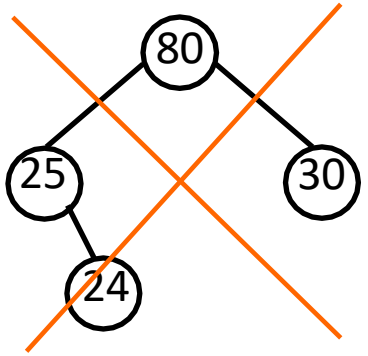


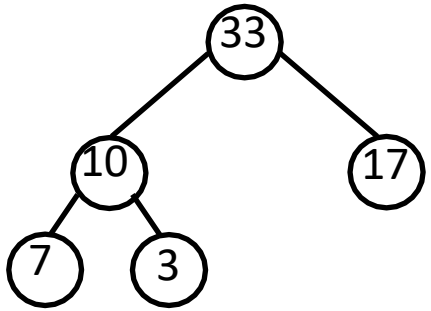
¿Cuáles son max-heaps?

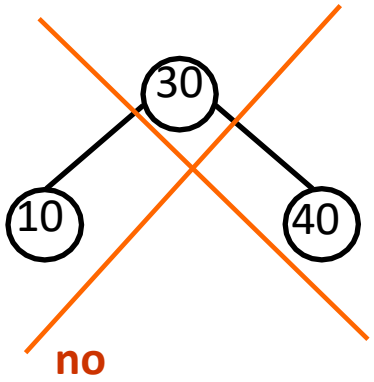




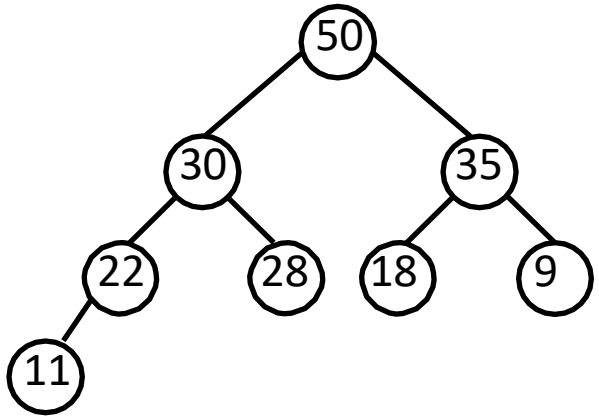
no







no

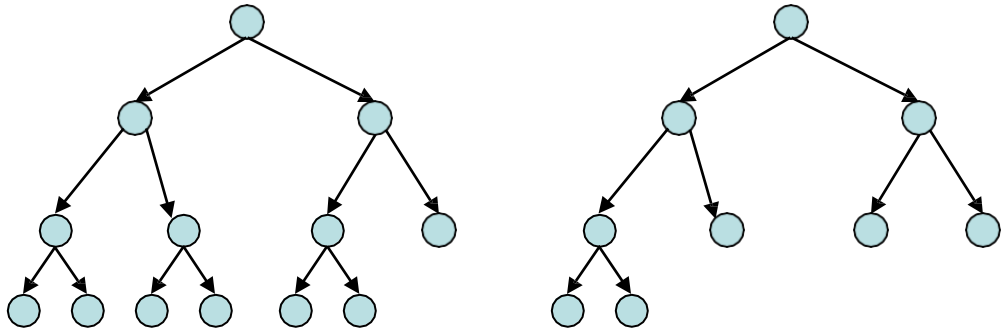


22

Altura del montículo y tiempo de ejecución

- La altura de un árbol completo es siempre log *N.*
  - ¿Cómo sabemos esto con certeza?

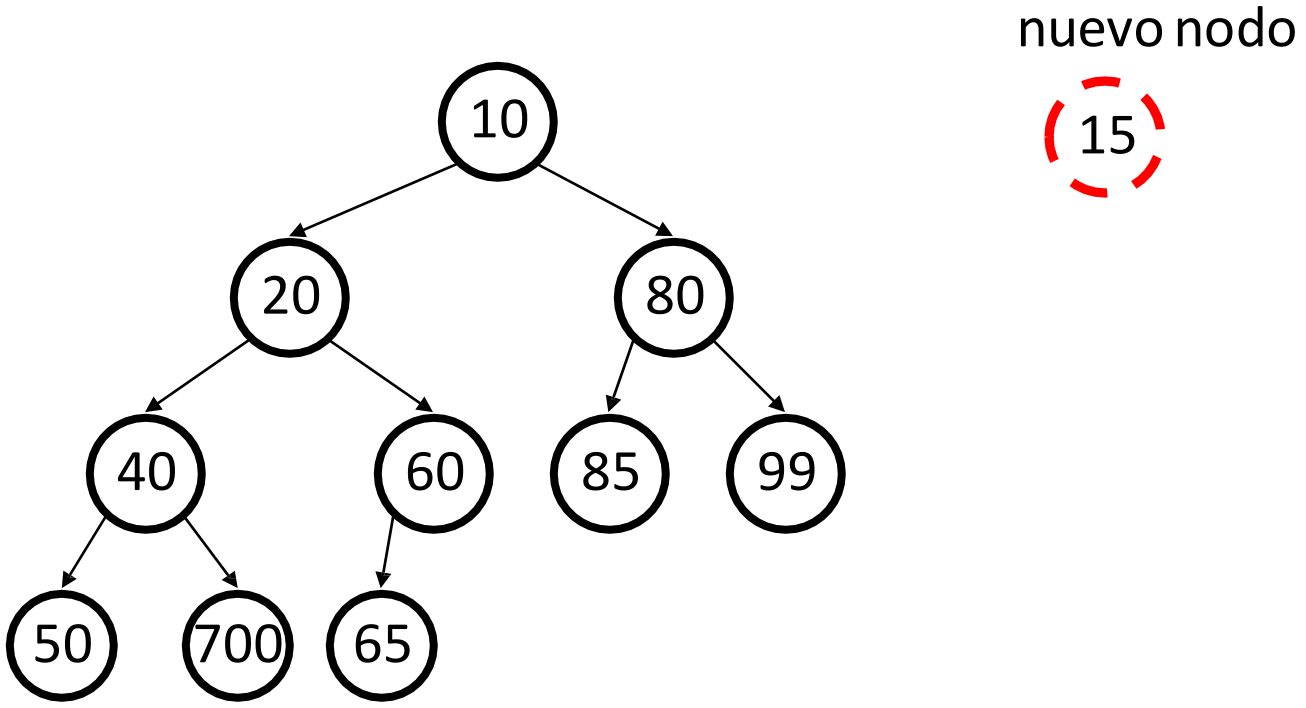
- Debido a esto, si implementamos una cola de prioridad usando un montículo, podemos proporcionar las siguientes garantías de tiempo de ejecución:
  - `add`: O(log *N*)
  - `peek`: O(1)
  - `remove`: O(log *N*)



árbol completo de *n nodos*
de altura h:
$2^{h} \leq n \leq 2^{h+1} - 1$
$h = \lfloor \log n \rfloor$

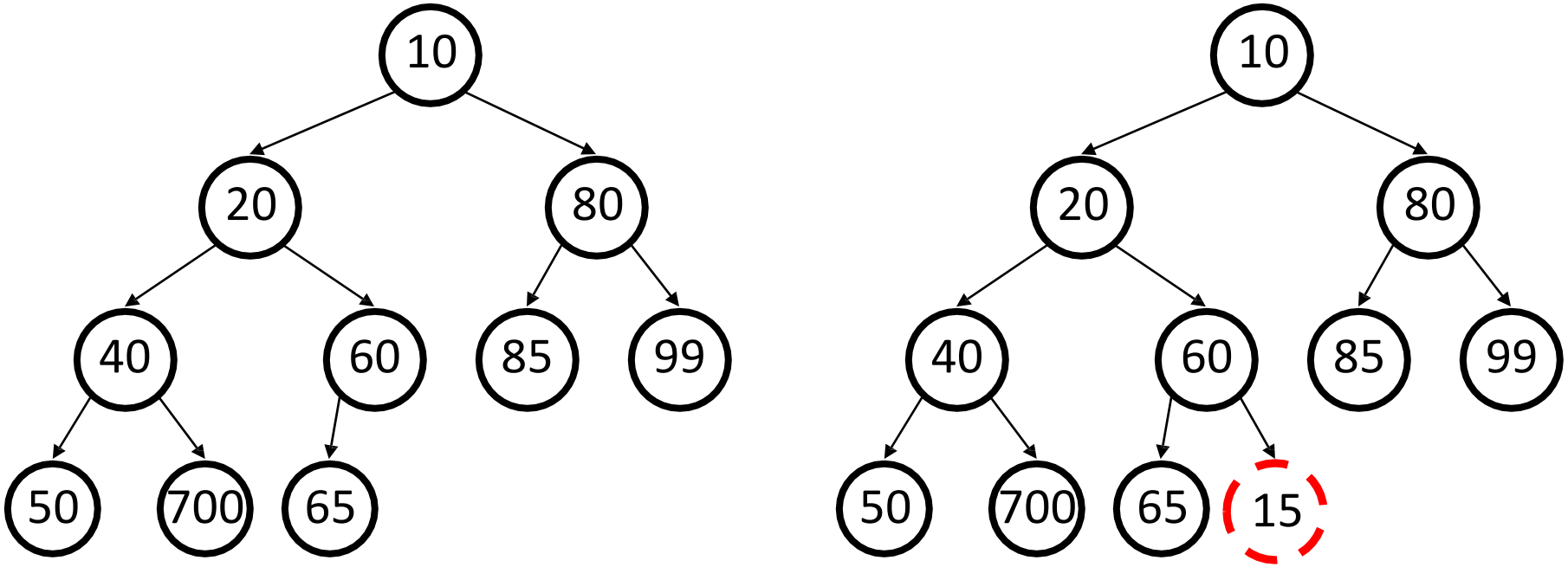
La operación add

- Cuando se agrega un elemento a un montículo, ¿dónde debe ir?
  - Debe insertar un nuevo nodo manteniendo las propiedades del montículo.
  - `queue.add(15);`



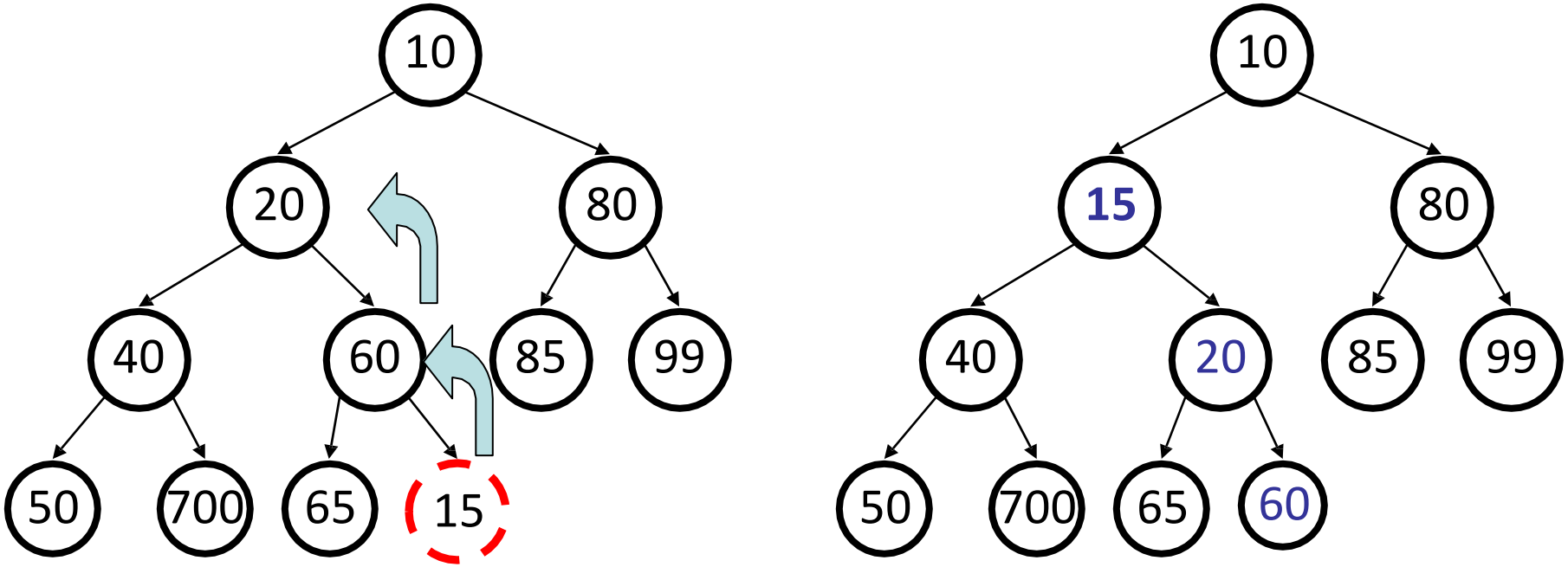
La operación add

- Cuando se agrega un elemento a un montículo, debe colocarse inicialmente como la *hoja más a la derecha* (para mantener la propiedad de integridad).
  - ¡Pero la propiedad de ordenamiento del montículo se rompe!



"Burbujeando" un nodo

- **Burbujear (bubble up)** : para restaurar el orden del montículo, el elemento recién agregado se desplaza ("burbujea") hacia arriba en el árbol hasta que alcanza su lugar adecuado.
  - Weiss: *"filtrarse" (percolate up)* intercambiando con su padre
  - ¿Cuántos bubble-ups son necesarios, como máximo?



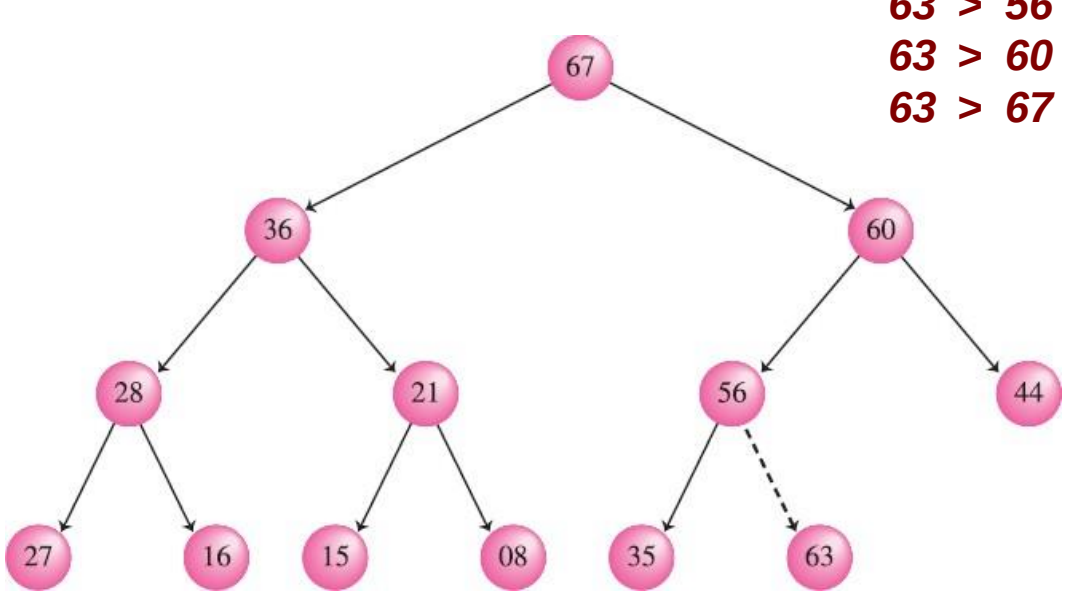
Inserción de un elemento en un montículo (max)

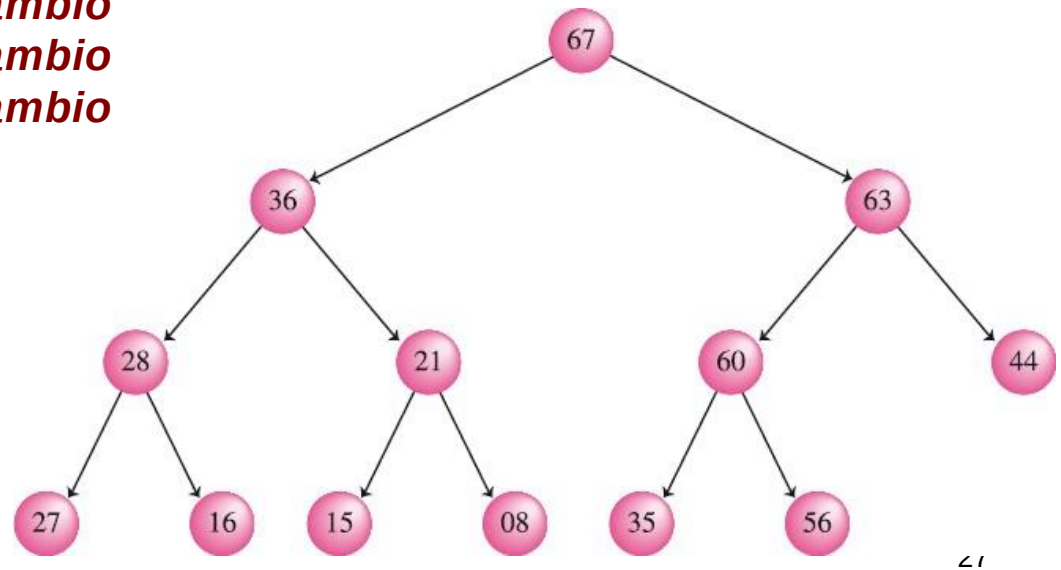
La inserción de un elemento en un montículo se lleva a cabo por medio de los siguientes pasos:

1. *Se inserta el elemento en la primera posición disponible.*
2. *Se verifica si su valor es mayor que el de su padre. Si se cumple esta condición entonces se efectúa el intercambio. Si no se cumple esta condición entonces el algoritmo se detiene y el elemento queda ubicado en la posición correcta en el montículo.*

Supongamos que se quiere incorporar al montículo de la figura el elemento 63, las comparaciones que realizamos son:

63 > 56 si hay intercambio
63 > 60 si hay intercambio
63 > 67 no hay intercambio





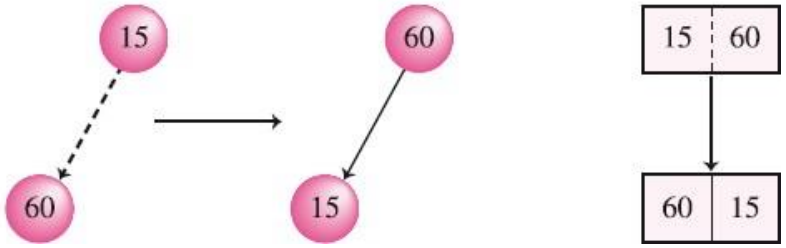
Inserción de un elemento en un montículo (max)

Supongamos que se desea insertar las siguientes claves en un montículo que se encuentra vacío:

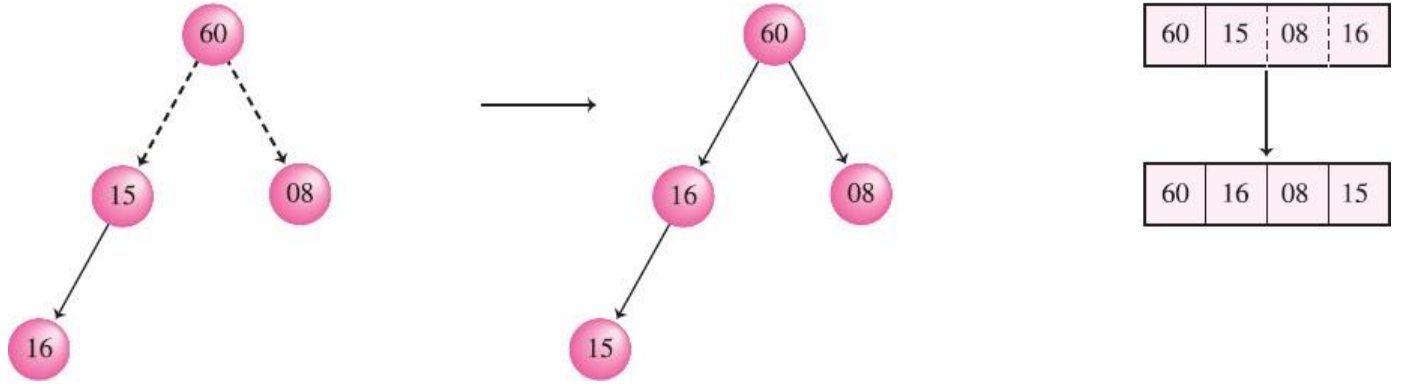
15 60 08 16 44 27 12 35

a) INSERCIÓN: CLAVE 15

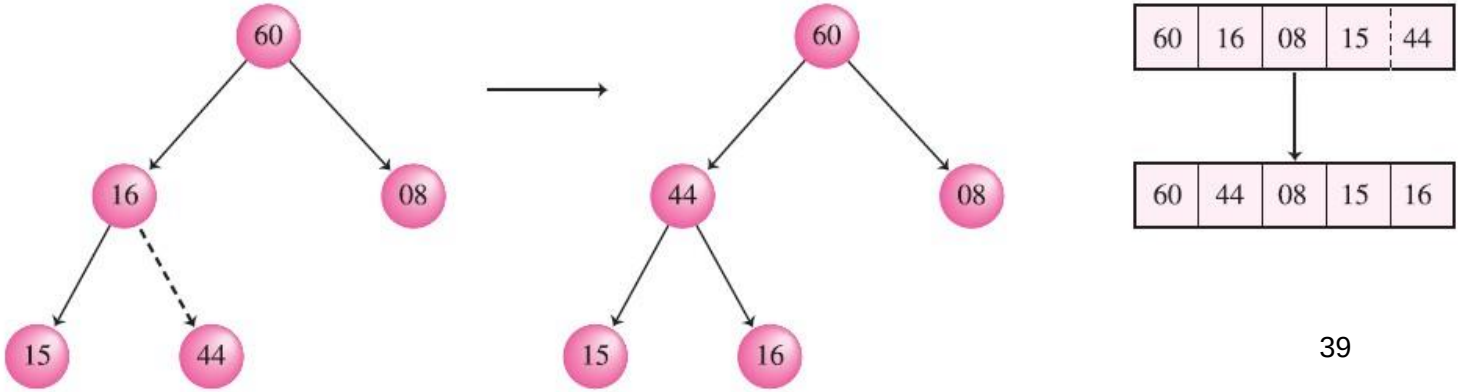
b) INSERCIÓN: CLAVE 60



c) INSERCIÓN: CLAVES 08 y 16



d) INSERCIÓN: CLAVE 44

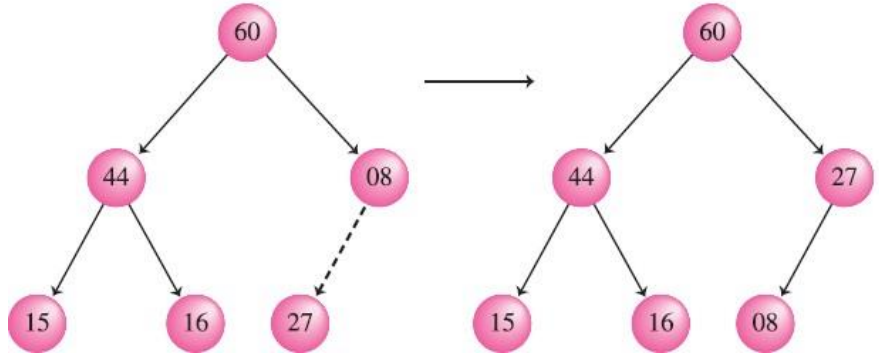


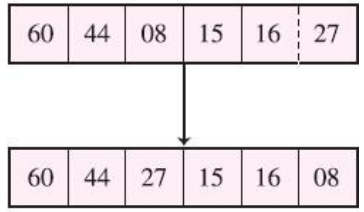
Inserción de un elemento en un montículo (max)

Supongamos que se desea insertar las siguientes claves en un montículo que se encuentra vacío:

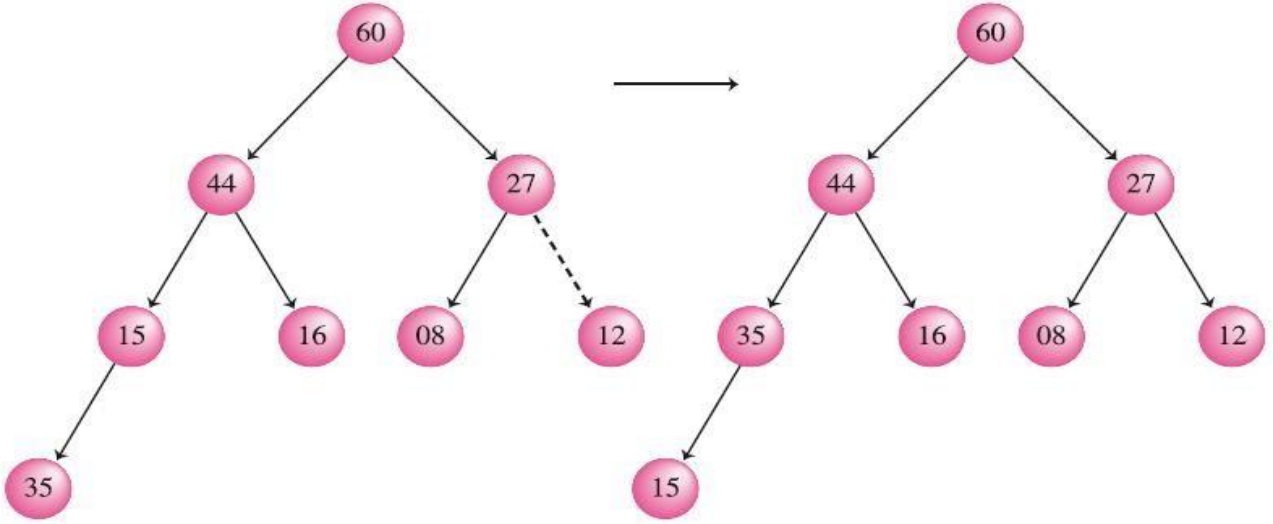
15 60 08 16 44 27 12 35

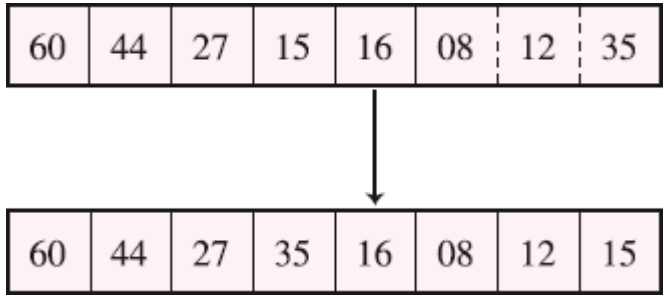
e) INSERCIÓN: CLAVE 27





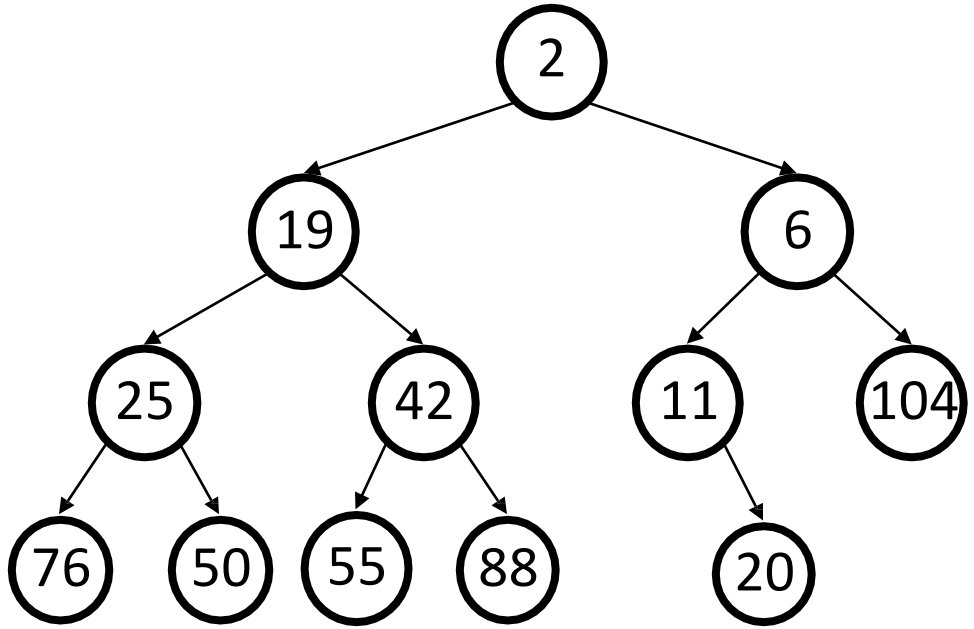
f) INSERCIÓN: CLAVES 12 y 35





Ejercicio

- Dibuje el estado del árbol de un montículo mínimo después de agregar estos elementos:
  - 6, 50, 11, 25, 42, 20, 104, 76, 19, 55, 88, 2



Inserción de un elemento en un montículo (max)

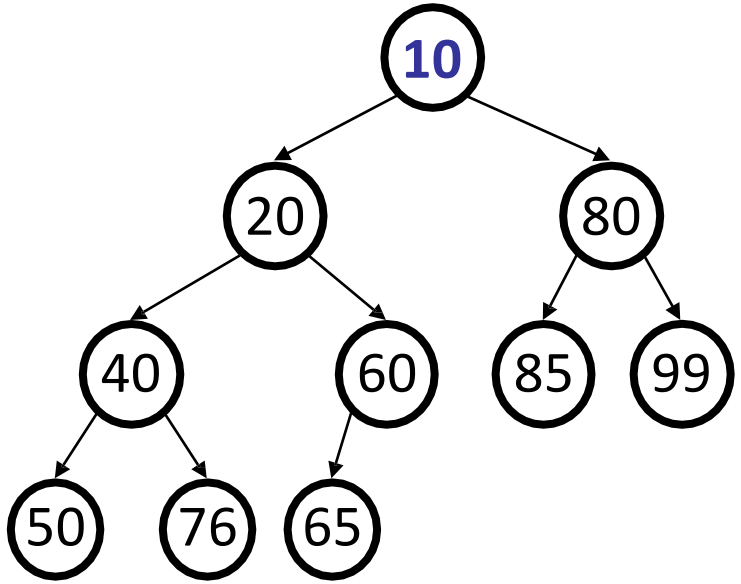
Ejercicios

Dado el montículo de la figura, verifique como queda luego de insertar las siguientes claves

56 21 13 28 67 36 07 10

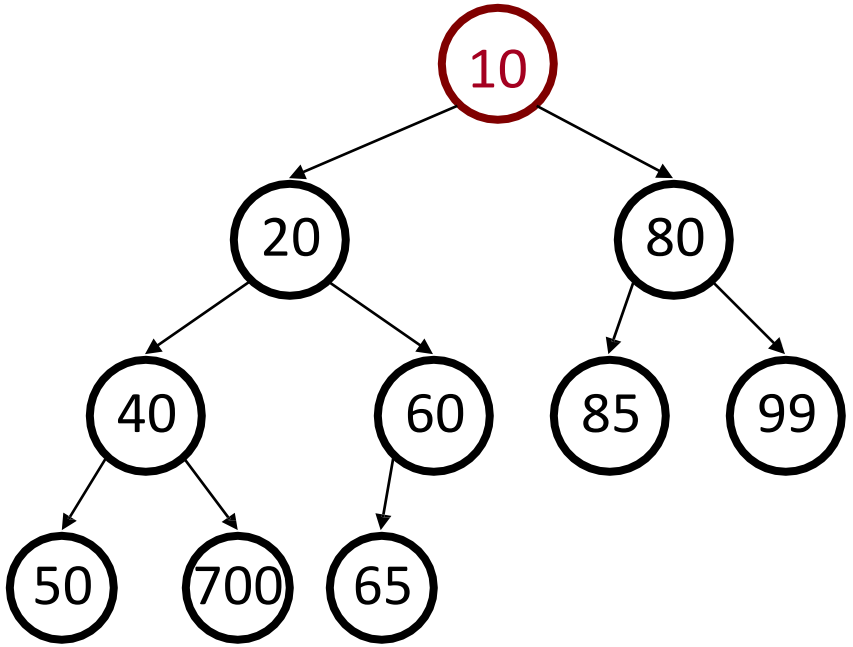
La operación peek

- Un "peek" a un montículo mínimo es trivial de realizar.
  - debido a las propiedades del montículo, el elemento mínimo es siempre la raíz
  - Tiempo de ejecución de O(1)
- El "peek" en un montículo máximo también sería O(1) (devuelve el máximo, no el mínimo)



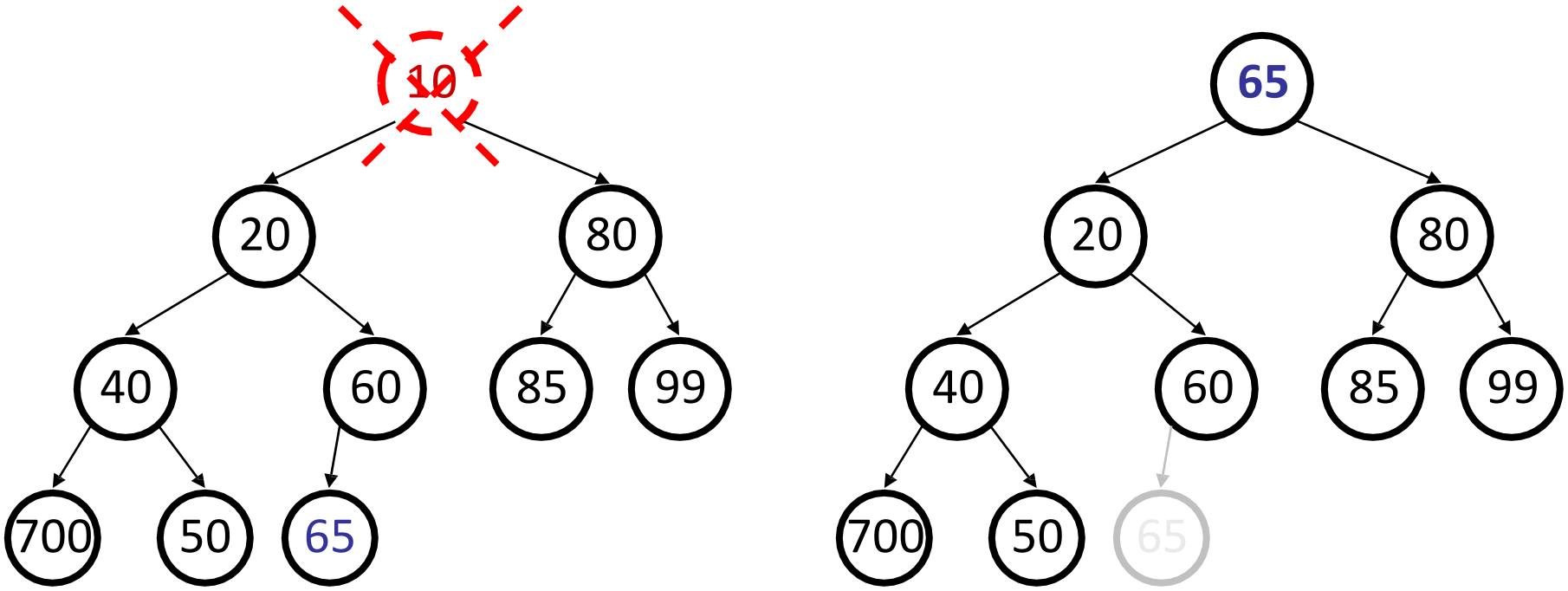
La operación remove

- Cuando se elimina un elemento de un montículo, ¿qué debemos hacer?
  - La raíz es el nodo a eliminar. ¿Cómo alteramos el árbol?
  - `queue.remove();`



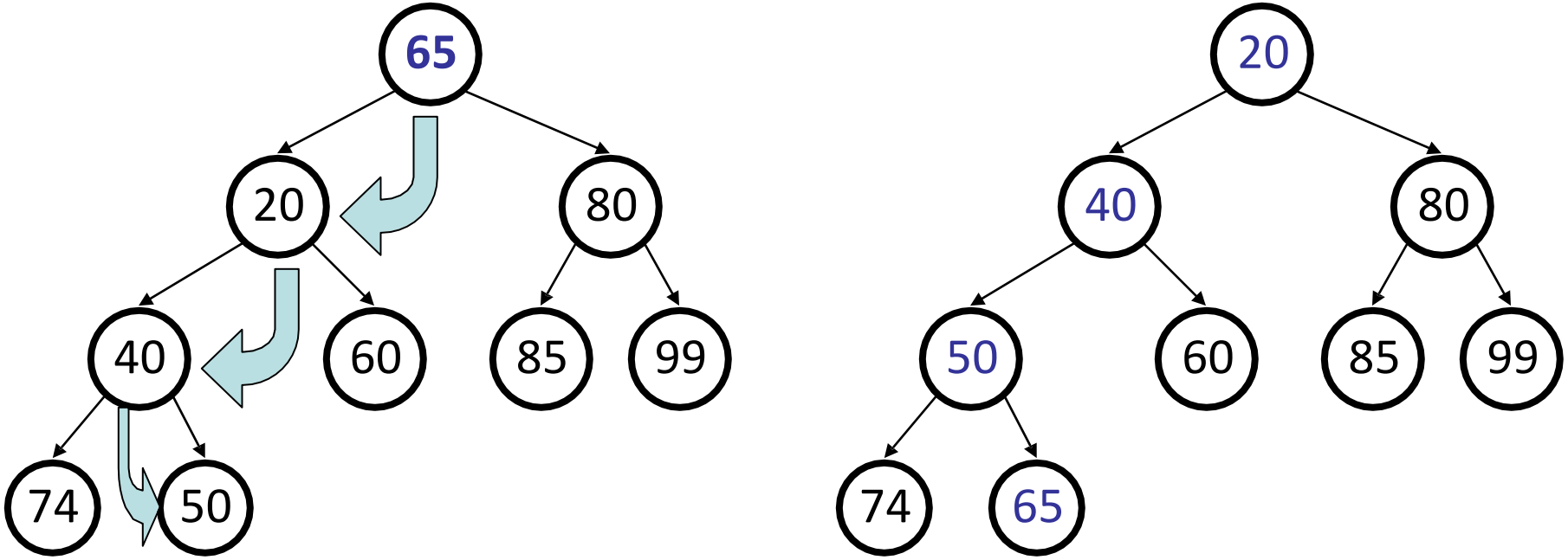
La operación remove

- Cuando la raíz se elimina de un montículo, debe reemplazarse inicialmente por la *hoja más a la derecha* (para mantener la integridad).
  - ¡Pero la propiedad de ordenamiento del montículo se rompe!



"Burbujeando hacia abajo" un nodo

- **bubble down** : para restaurar el orden del montículo, la nueva raíz impropia se desplaza ("burbujea") hacia abajo en el árbol hasta que alcanza su lugar adecuado.
  - Weiss: *"filtrarse" (percolate down)* intercambiando con su hijo *más* pequeño (¿por qué?)
  - ¿Cuántos bubble-down son necesarios, como máximo?



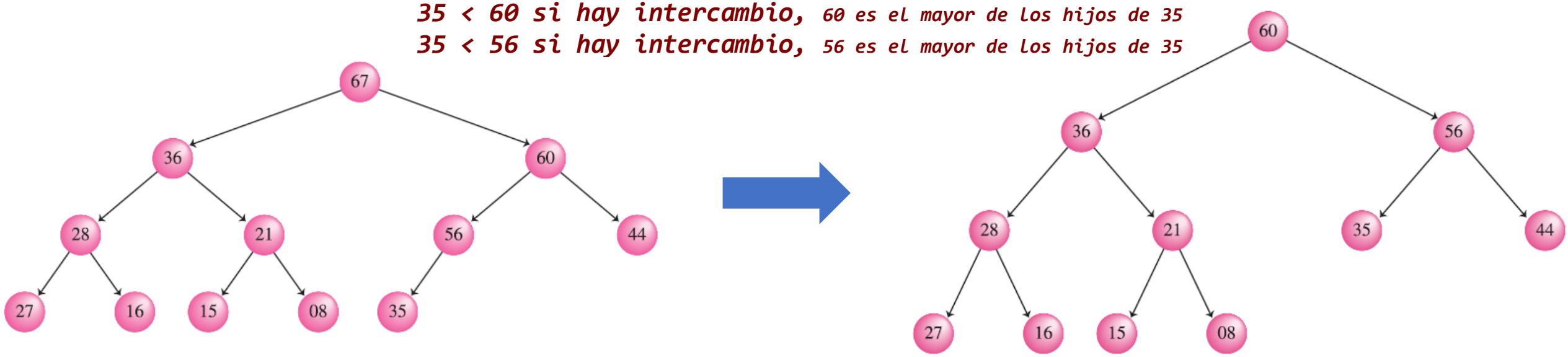
Eliminación de un montículo (max)

Los pasos necesarios para lograr la eliminación de la raíz de un montículo son:

1. Se reemplaza la raíz con el elemento que ocupa la última posición del montículo.
2. Se verifica si el nuevo valor de la raíz es menor que el valor mas grande de sus hijos. Si se cumple la condición, entonces se efectúa el intercambio. Si no se cumple la condición entonces el algoritmo se detiene y el elemento queda ubicado en la posición correcta en el montículo.
Se repite desde 1.

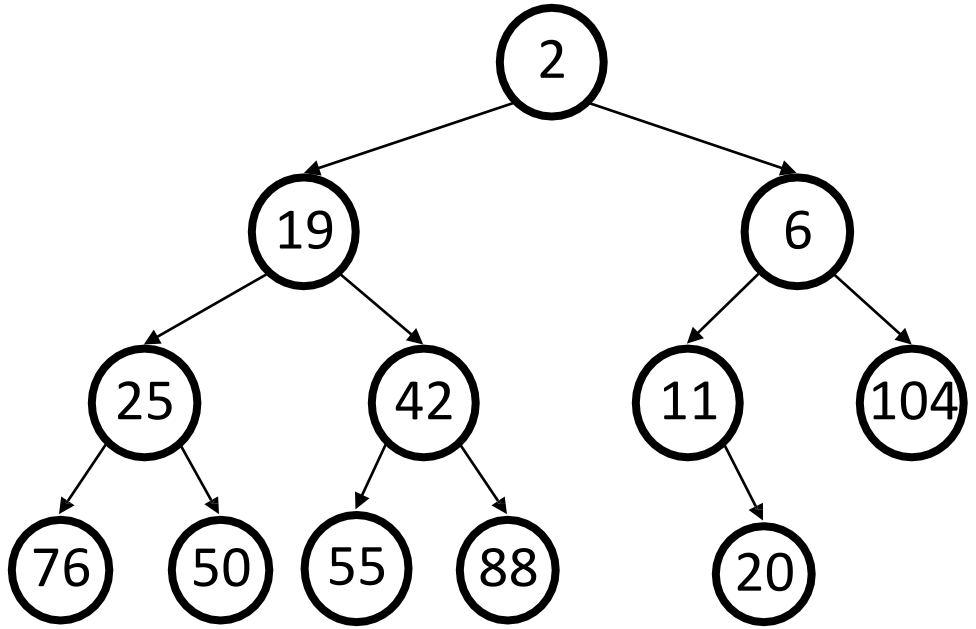
Supongamos que se desea eliminar la raíz del montículo (67) de la figura:

35 < 60 si hay intercambio, *60 es el mayor de los hijos de 35*
35 < 56 si hay intercambio, *56 es el mayor de los hijos de 35*



Ejercicio

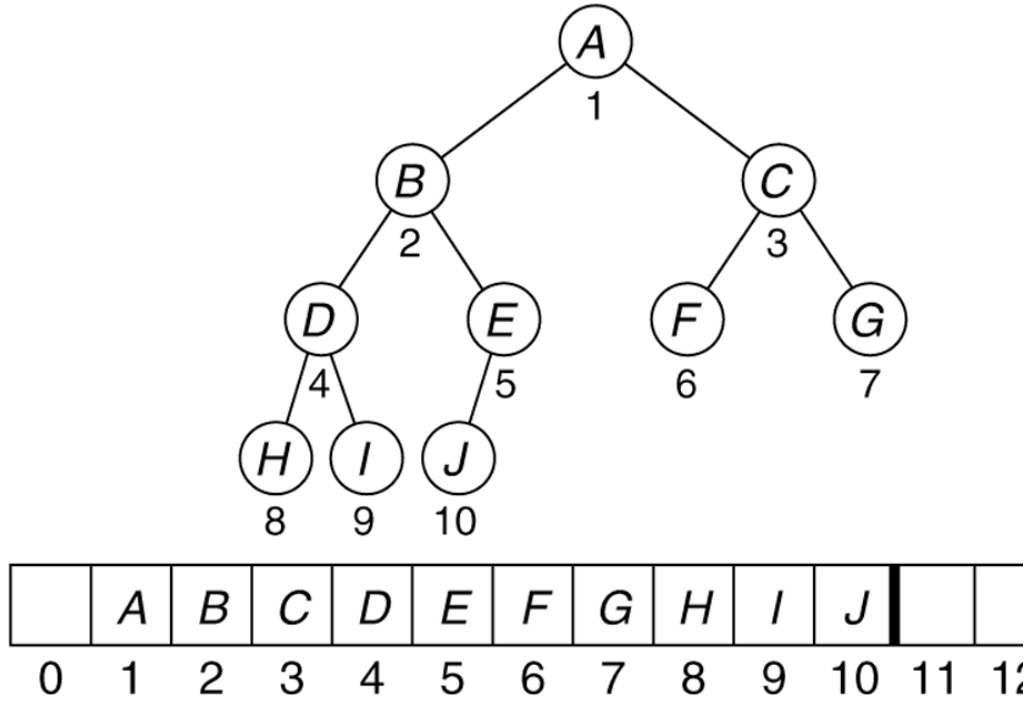
- Supongamos que tenemos el montículo mínimo que se muestra a continuación.
- Muestre el estado del árbol del montículo después de que se haya llamado ejecutado la operación remove 3 veces, y qué elementos son devueltos por la eliminación.



Implementación de una Cola prioritaria

Implementación de montículo con arreglos

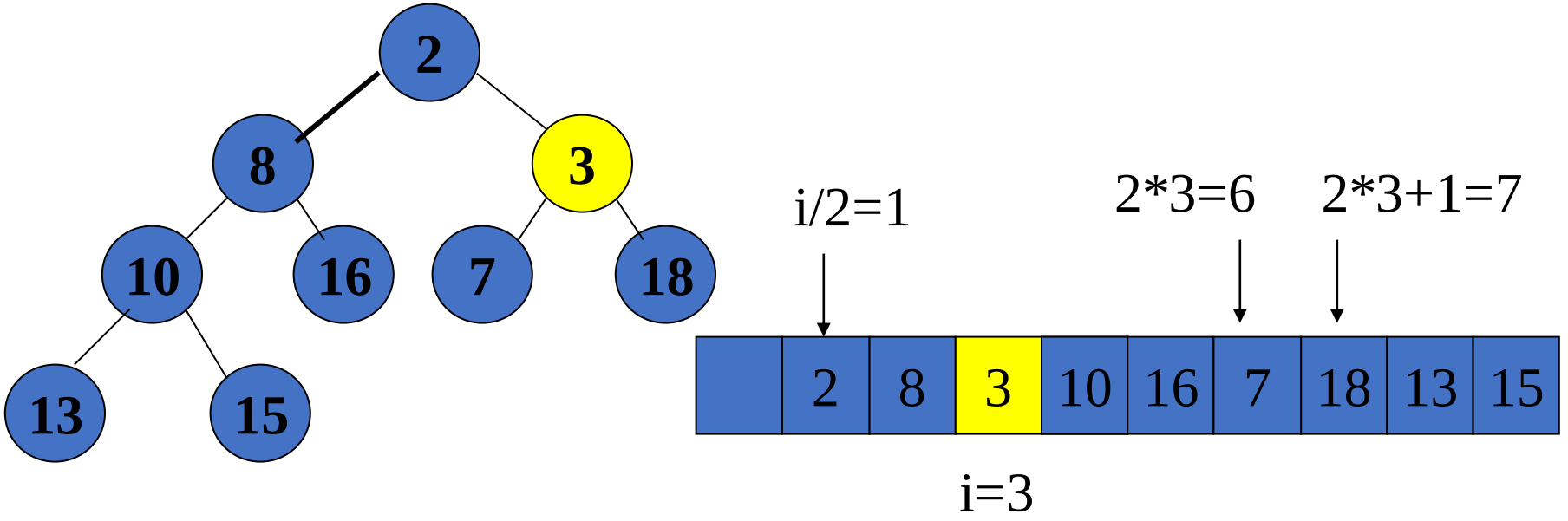
- Aunque un montículo es conceptualmente un árbol binario, ya que es un árbol *completo* , cuando lo implementamos, en realidad podemos "hacer trampa" y simplemente *usar un arreglo* .
  - índice de raíz = 1 (deje 0 vacío para simplificar las matemáticas)
  - para cualquier nodo *n* en el índice *i* :
    - índice de *n*.left = 2 *i*
    - índice de *n*.right = 2 *i* + 1
    - índice padre de *n* ?
  - Esta representación de matriz es elegante y eficiente (O(1)) para operaciones de árbol comunes.



Ejemplos

Si un nodo está almacenado en la posición i (cuando empieza de 1):

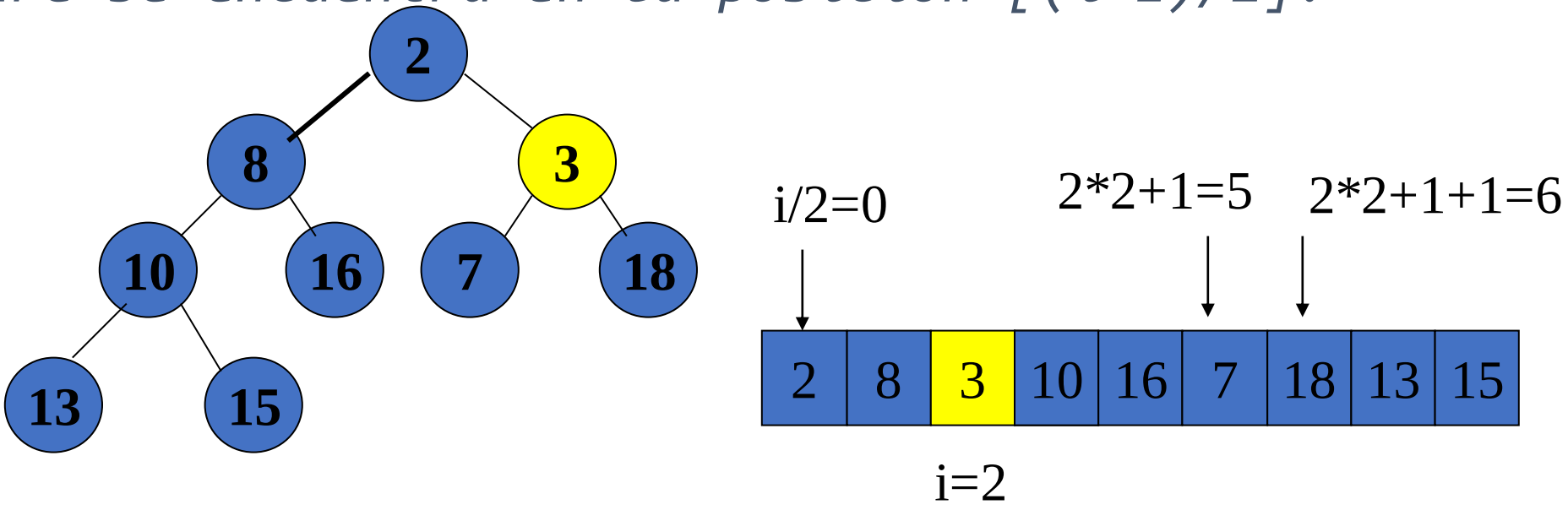
- *Su hijo izquierdo, si existe, se encuentra en la posición 2i.*
- *Su hijo derecho, si existe, se encuentra en la posición 2i+1.*
- *El padre se encuentra en la posición [i/2].*



Ejemplos

Si un nodo está almacenado en la posición i (cuando empieza de 0):

- *Su hijo izquierdo, si existe, se encuentra en la posición 2i+1.*
- *Su hijo derecho, si existe, se encuentra en la posición (2i+1)+1.*
- *El padre se encuentra en la posición [(i-1)/2].*



La forma secuencial de un montículo de `n` elementos implica que si `2*i+1 ≥ n` entonces `i` no tiene hijo izquierdo (tampoco hijo derecho), y si `(2*i+1)+1 ≥ n` entonces `i` no tiene hijo derecho.

Implementando HeapPQ

- Implementemos una cola de prioridad `int` usando un arreglo min-heap.

```
public class HeapIntPriorityQueue
        implements IntPriorityQueue {
    private int[] elements;
    private int size;

    // construye una nueva cola de prioridad vacía
    public HeapIntPriorityQueue() {
        elements = new int[10];
        size = 0;
    }

    ...
}
```

Métodos auxiliares

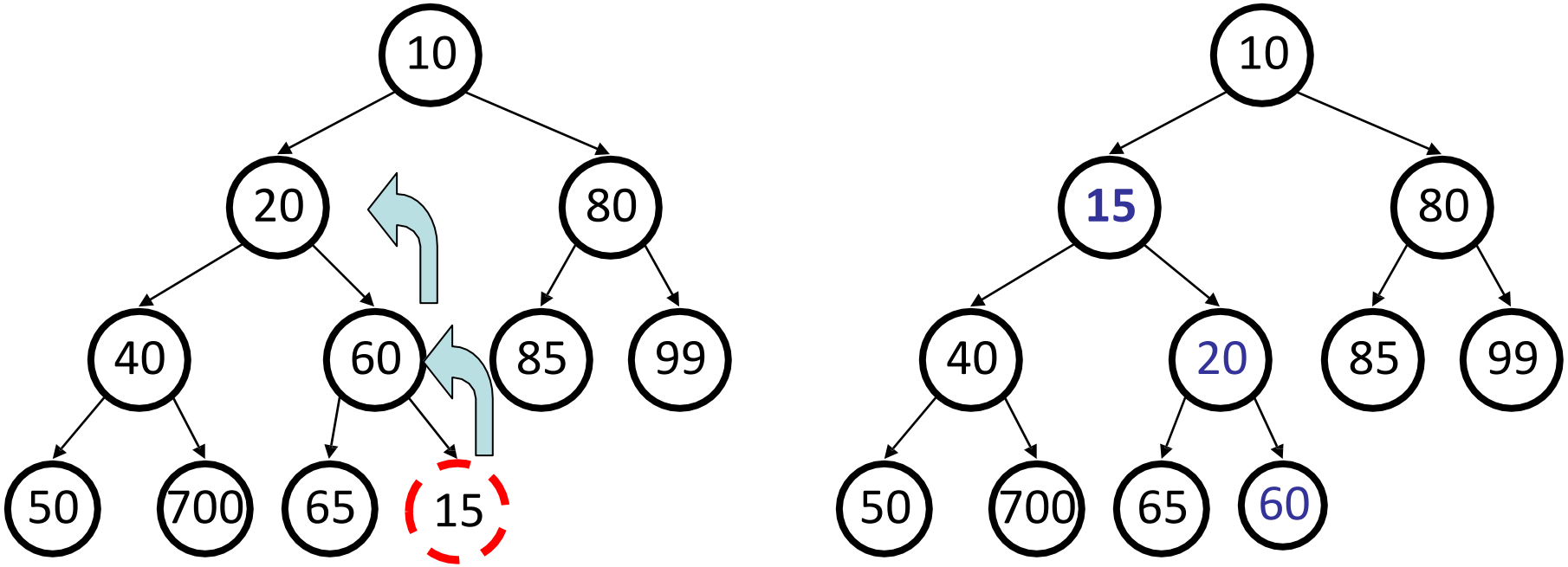
- Dado que trataremos el arreglo como un árbol/montículo completo, y subiremos/bajaremos entre padres/hijos, estos métodos son útiles:

```
// ayudas para navegar por los índices arriba/abajo del árbol
private int parent(int index)          { return index/2; }
private int leftChild(int index)       { return index*2; }
private int rightChild(int index)      { return index*2 + 1; }
private boolean hasParent(int index) { return index > 1; }
private boolean hasLeftChild(int index) {
    return leftChild(index) <= size;
}
private boolean hasRightChild(int index) {
    return rightChild(index) <= size;
}
private void swap(int[] a, int index1, int index2) {
    int temp = a[index1];
    a[index1] = a[index2];
    a[index2] = temp;
}
```

Implementando add

- Escribamos el código para agregar un elemento al montículo:

```
public void add(int value) {
      ...
  }
```



Implementando add

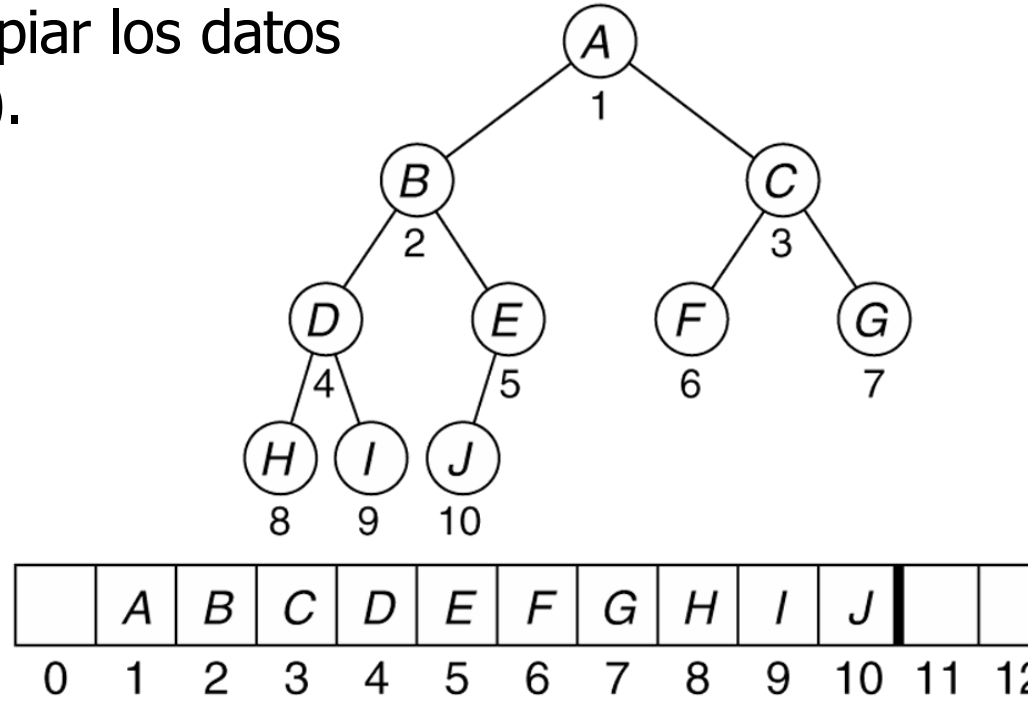
```
// Agrega el valor dado a esta cola de prioridad en orden.
public void add(int value) {
    elements[size + 1] = value; // añadir como hoja más a la derecha
// burbujear hacia arriba según sea necesario para arreglar el orden
    int index = size + 1;
    boolean found = false;
    while (!found && hasParent(index)) {
        int parent = parent(index);
        if (elements[index] < elements[parent]) {
            swap(elements, index, parent(index));
            index = parent(index);
        } else {
            found = true; // encontró la ubicación adecuada; detener
        }
    }

    size++;
}
```

Cambiar el tamaño de un montículo

- ¿Qué pasa si nuestro montículo de arreglo se queda sin espacio?
  - Debemos agrandarlo.
  - ¿Qué debemos hacer aquí?

  - (Simplemente podemos copiar los datos en una matriz más grande).



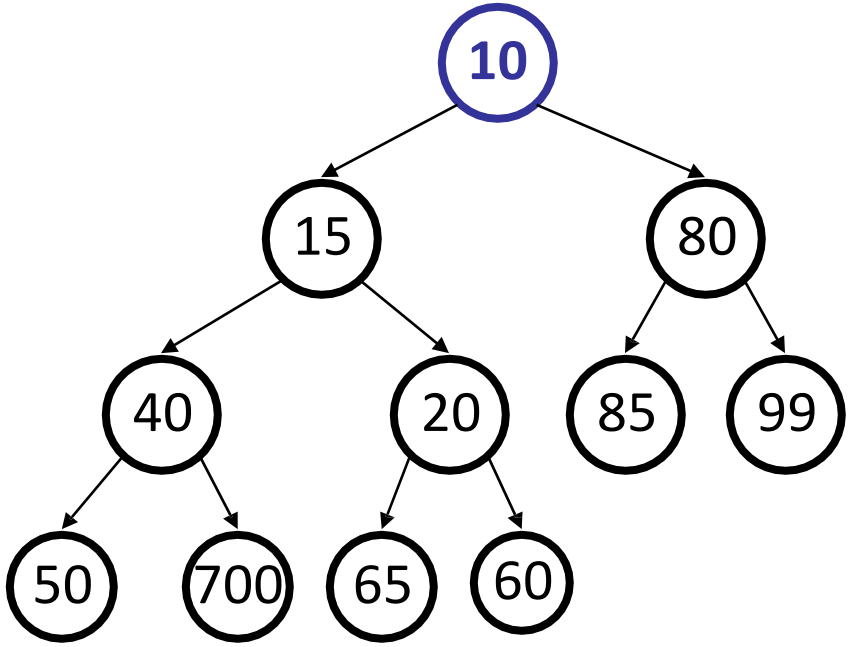
Código add modificado

```
// Agrega el valor dado a esta cola de prioridad en orden. public void add(int value) {
// cambiar el tamaño para agrandar el montículo si es necesario     if (size ==
  elements.length - 1) {
        elements = Arrays.copyOf(elements,
                                 2 * elements.length);
    }
    ...
}
```

Implementando peek

- Escribamos código para recuperar el elemento mínimo en el montículo:

```
public int peek() {
      ...
  }
```



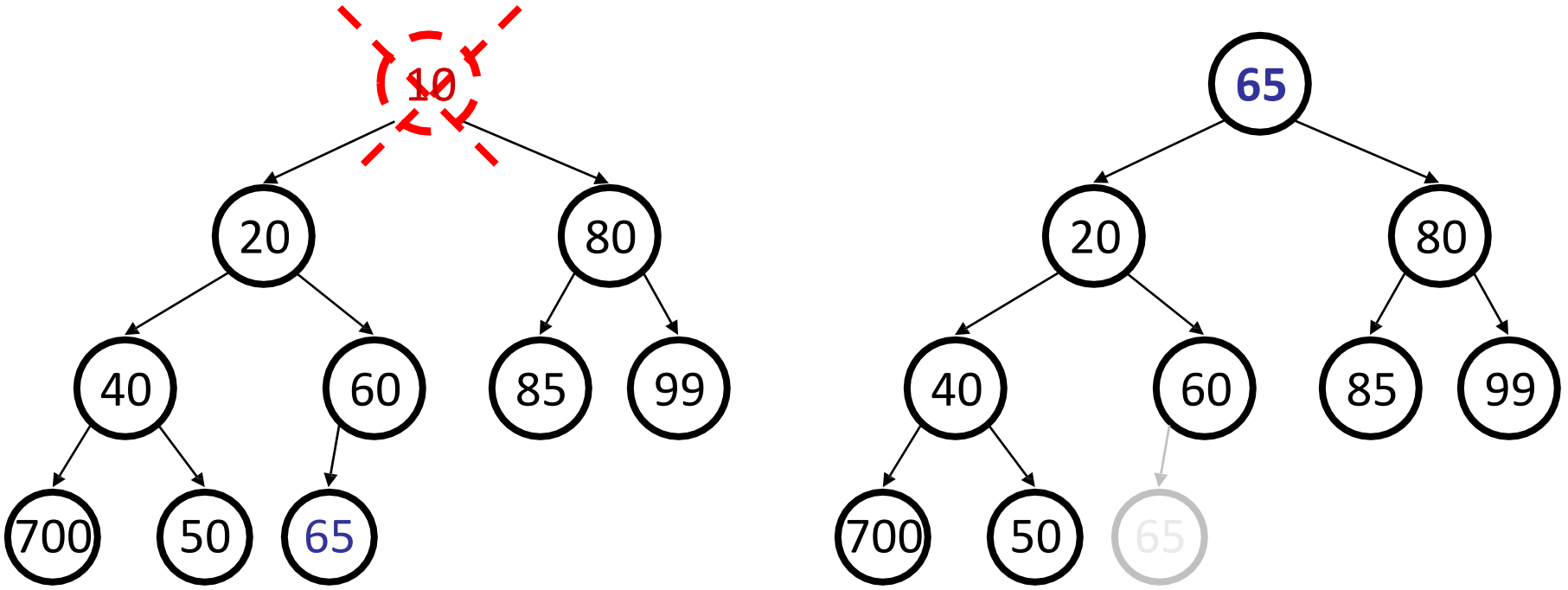
Implementando peek

```
// Devuelve el elemento mínimo en esta cola de prioridad.
// condición previa: la cola no está vacía
public int peek() {
    return elements[1];
}
```

Implementando remove

- Escribamos código para eliminar el elemento mínimo en el montículo:

```
public int remove() {
    ...
}
```



Implementando remove

```
public int remove() {   // precondicion: la cola no está vacía
    int result = elements[1];       // última hoja -> raíz
    elements[1] = elements[size];
    size--;
    int index = 1; // "burbujear hacia abajo" para arreglar el orden
    boolean found = false;
    while (!found && hasLeftChild(index)) {
        int left = leftChild(index);
        int right = rightChild(index);
        int child = left;
        if (hasRightChild(index) &&
                elements[right] < elements[left]) {
            child = right;
        }
        if (elements[index] > elements[child]) {
            swap(elements, index, child);
            index = child;
        } else {
            found = true; encontró la ubicación adecuada;
        }
    }
    return result;
}
```

Interfaz ADT PQ int.

- Escribamos nuestra propia implementación de una cola de prioridad.
  - Para simplificar el problema, solo almacenamos `int`s en nuestro conjunto por ahora.
  - Como se hace (generalmente) en Java Collection Framework, definiremos conjuntos como un ADT creando una interfaz de conjunto.
  - Las operaciones principales son: add, peek (en min), remove (min).

```
public interface IntPriorityQueue {
        void add(int value);
        void clear();
        boolean isEmpty();
        int peek(); // devuelve el elemento mínimo
        int remove();// elimina/devuelve el elemento mínimo
        int size();
    }
```

ADT PQ genérico

- Modifiquemos nuestra cola de prioridad para que pueda almacenar cualquier tipo de datos.
  - Al igual que con las colecciones anteriores, usaremos genéricos de Java (un parámetro de tipo).

```
public interface PriorityQueue<E> {
        void add(E value);
        void clear();
        boolean isEmpty();
        E peek(); // devuelve el elemento mínimo
        E remove(); // elimina/devuelve el elemento mínimo
        int size();
    }
```

Clase HeapPQ genérica

- Podemos modificar nuestra clase HeapPriorityQueue para usar genéricos como de costumbre...

```
public class HeapPriorityQueue<E>
        implements PriorityQueue<E> {
    private E[] elements;
    private int size;

    // construye una nueva cola de prioridad vacía
    public HeapPriorityQueue() {
        elements = (E[]) new Object[10];
        size = 0;
    }

    ...
}
```

Problema: ordenar elementos

```
// Agrega el valor dado a esta cola de prioridad en orden.
public void add(E value) {
    ...
    int index = size + 1;
    boolean found = false;
    while (!found && hasParent(index)) {
        int parent = parent(index);
        if (elements[index] < elements[parent]) {   // error
            swap(elements, index, parent(index));
            index = parent(index);
        } else {
            found = true; // encontró la ubicación adecuada; detener
        }
    }
}
```

- Incluso cambiar el `<` a una llamada `compareTo` no funciona.
  - Java no puede estar seguro de que el tipo `E` tenga un método `compareTo` .

Comparando objetos

- Los montículos dependen de poder *ordenar* sus elementos.
- Los operadores como `<` y `>` no funcionan con objetos en Java.
  - Pero pensamos que algunos tipos tienen un orden (por ejemplo , `Date`).
  - (En otros lenguajes, podemos habilitar <, > con *sobrecarga de operadores*).

- **Ordenación natural** : Reglas que gobiernan la ubicación relativa de todos los valores de un tipo dado.
  - Implica una noción de igualdad (como `equals`) pero también < y > .
  - **Ordenación total** : todos los elementos se pueden organizar en A ≤ B ≤ C ≤ ... orden.
  - La interfaz `Comparable` proporciona un ordenamiento natural.

La interfaz comparable

- La forma estándar para que una clase Java defina una función de comparación para sus objetos es implementar la interfaz `Comparable` .

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

- Una llamada de **A**`.compareTo(`**B**`)` debería devolver: un valor < 0 si **A** viene "antes" de **B** en el ordenamiento, un valor > 0 si **A** viene "después" de **B** en el ordenamiento, o exactamente 0 si **A** y **B** se consideran "iguales" en el ordenamiento.
- **Consejo** : Considere implementar `Comparable` .

Parámetros de tipo acotado

< **Tipo** `extends` **SuperTipo** >

- Un límite superior; acepta el supertipo dado o cualquiera de sus subtipos.
- Funciona para múltiples superclases/interfaces con `&` :

< **Tipo** `extends` **ClassA** `&` **InterfaceB** `&` **InterfaceC** `&` **...** >

< **Tipo** `super` **Supertipo** >

- Un límite inferior; acepta el supertipo dado o cualquiera de sus supertipos.

- Ejemplo:

```
// puede ser instanciado con cualquier tipo de animal
public class Nest<T extends Animal> {
    ...
}
...
Nest<Bluebird> nest = new Nest<Bluebird>();
```

Clase HeapPQ corregida

```
public class HeapPriorityQueue<E extends Comparable<E>>
        implements PriorityQueue<E> {
    private E[] elements;
    private int size;

 // construye una nueva cola de prioridad vacía
    public HeapPriorityQueue() {
        elements = (E[]) new Object[10];
        size = 0;
    }
    ...
    public void add(E value) {
        ...
        while (...) {
            if (elements[index].compareTo(
                    elements[parent]) < 0) {
                swap(...);
            }
        }
    }
}
```

Ordenamiento y comparadores

¿Qué es el orden "natural"?

```
public class Rectangle implements Comparable<Rectangle> {
    private int x, y, width, height;

    public int compareTo(Rectangle other) {
        // ...?
    }
}
```

- ¿Cuál es el "ordenamiento natural" de los rectángulos?
  - ¿Por x, rompiendo lazos por y?
  - ¿Por ancho, rompiendo lazos por alto?
  - ¿Por área? por perímetro?

- ¿Los rectángulos tienen algún orden "natural"?
  - ¿Podríamos querer colocar los rectángulos en algún orden de todos modos?

Interfaz Comparator

```
public interface Comparator<T> {
    public int compare(T first, T second);
}
```

- Interface `Comparator` es un objeto externo que especifica una función de comparación sobre algún otro tipo de objetos.
  - Le permite definir varios formas de ordenar un mismo tipo.
  - Le permite definir una(s) ordenación(es) específica(s) para un tipo incluso si no hay una ordenación "natural" obvia para ese tipo.
  - Le permite definir externamente un orden para una clase que, por cualquier motivo, no puede modificar para que sea Comparable:
    - una clase que es parte de las bibliotecas de clases de Java
    - una clase que es final y no se puede extender
    - una clase de otra biblioteca o autor, que no controlas
    - ...

Ejemplos Comparator

```
public class RectangleAreaComparator
        implements Comparator<Rectangle> {
 // comparar en orden ascendente por área (AnchoxAlto)
    public int compare(Rectangle r1, Rectangle r2) {
        return r1.getArea() - r2.getArea();
    }
}

public class RectangleXYComparator
        implements Comparator<Rectangle> {
 // comparar por x ascendente, romper empates por y
    public int compare(Rectangle r1, Rectangle r2) {
        if (r1.getX() != r2.getX()) {
            return r1.getX() - r2.getX();
        } else {
            return r1.getY() - r2.getY();
        }
    }
}
```

Usando Comparator

- `TreeSet`, `TreeMap`, `PriorityQueue` pueden usar `Comparator`:

```
Comparator<Rectangle> comp = new RectangleAreaComparator();
Set<Rectangle> set = new TreeSet<Rectangle>(comp);
Queue<Rectangle> pq = new PriorityQueue<Rectangle>(10,comp);
```

- Métodos de búsqueda y clasificación pueden aceptar `Comparator`s.

```
Arrays.binarySearch(array, value, comparator)
Arrays.sort(array, comparator)
Collections.binarySearch(list, comparator)
Collections.max(collection, comparator)
Collections.min(collection, comparator)
Collections.sort(list, comparator)
```

- Se proporcionan métodos para invertir el orden de un `Comparator`:

```
public static Comparator Collections.reverseOrder()
 public static Comparator Collections.reverseOrder(comparator)
```

PQ y Comparator

- Nuestra cola de prioridad de almacenamiento dinámico actualmente se basa en el ordenamiento natural `Comparable` de sus elementos:

```
public class HeapPriorityQueue<E extends Comparable<E>>
        implements PriorityQueue<E> {
    ...
    public HeapPriorityQueue() {...}
}
```

- Para permitir otros ordenamientos, podemos agregar un constructor que acepte un `Comparator` para que los clientes puedan organizar los elementos en cualquier orden:

```
...
    public HeapPriorityQueue(Comparator<E> comp) {...}
```

Ejercicio PQ Comparator

- Escriba código que almacene cadenas en una cola de prioridad y las vuelva a leer en orden ascendente *por longitud* .
  - Si dos cuerdas tienen la misma longitud, rompa el empate *en orden ABC* .

```
Queue<String> pq = new PriorityQueue<String>(...);
pq.add("you");
pq.add("meet");
pq.add("madam");
pq.add("sir");
pq.add("hello");
pq.add("goodbye");
while (!pq.isEmpty()) {
    System.out.print(pq.remove() + " ");
}

// sir you meet hello madam goodbye
```

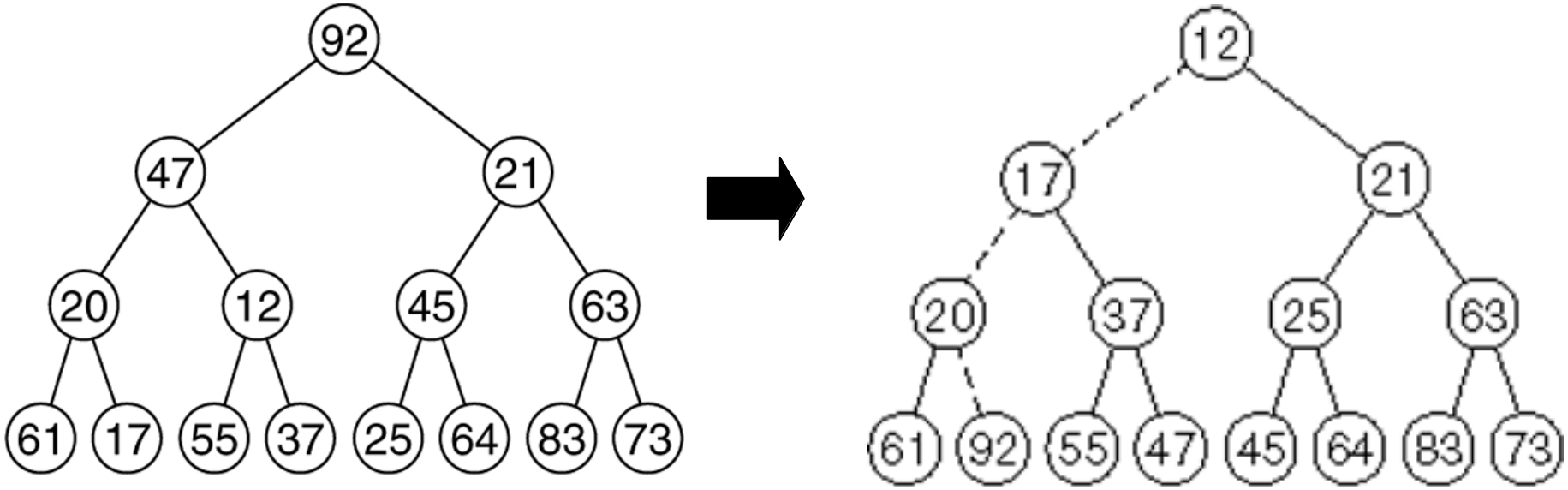
Respuesta PQ Comparator

- Utilice la siguiente clase de comparación para organizar las cadenas:

```
public class LengthComparator
        implements Comparator<String> {
    public int compare(String s1, String s2) {
        if (s1.length() != s2.length()) {
 // si las longitudes son desiguales, comparar por longitud
            return s1.length() - s2.length();
        } else {
 // romper empates por orden ABC
            return s1.compareTo(s2);
        }
    }
}
...
Queue<String> pq = new PriorityQueue<String>(100,
                          new LengthComparator());
```

Heap sort

- **heap sort** : un algoritmo para ordenar una matriz de *N* elementos convirtiendo la matriz en un montículo y luego llamando a `remove` *N* veces.
  - Los elementos saldrán ordenados.
  - Podemos ponerlos en un nuevo arreglo ordenado.
  - ¿Cuál es el tiempo de ejecución?



Implementación heapsort

```
public static void heapSort(int[] a) {
    PriorityQueue<Integer> pq =
            new HeapPriorityQueue<Integer>();
    for (int n : a) {
        pq.add(a);
    }
    for (int i = 0; i < a.length; i++) {
        a[i] = pq.remove();
    }
}
```

- Este código es correcto y se ejecuta en tiempo O(*N* log *N*) pero desperdicia memoria.
- Hace una copia completa del arreglo `a` en el montículo interno de la cola de prioridad.
- ¿Podemos realizar heap sort sin hacer una copia de `a`?

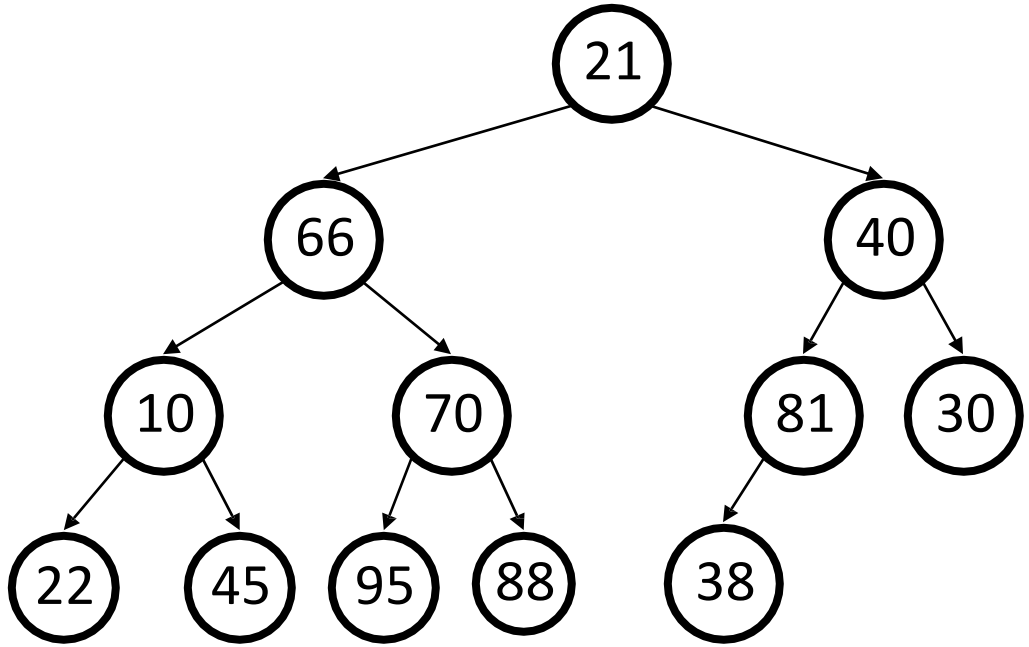
Mejora del código

- *Idea:* Trátese `a` sí mismo como un montículo máximo, cuyos datos comienzan en 0 (no en 1).
  - `a` no está realmente en orden de montículo.
  - Pero si "burbujea hacia abajo" repetidamente cada nodo que *no sea hoja* , comenzando desde el último, eventualmente tendrá un montículo adecuado.

- Ahora que `a` es un montículo máximo válido:
  - Llame a `remove` repetidamente hasta que el montículo esté vacío.
  - Pero hágalo de modo que cuando se "elimine" un elemento, se mueva al final del arreglo en lugar de desalojarlo por completo del arreglo.
  - Cuando termines, ¡voilá! El arreglo está ordenado.

Paso 1: Cree un montículo

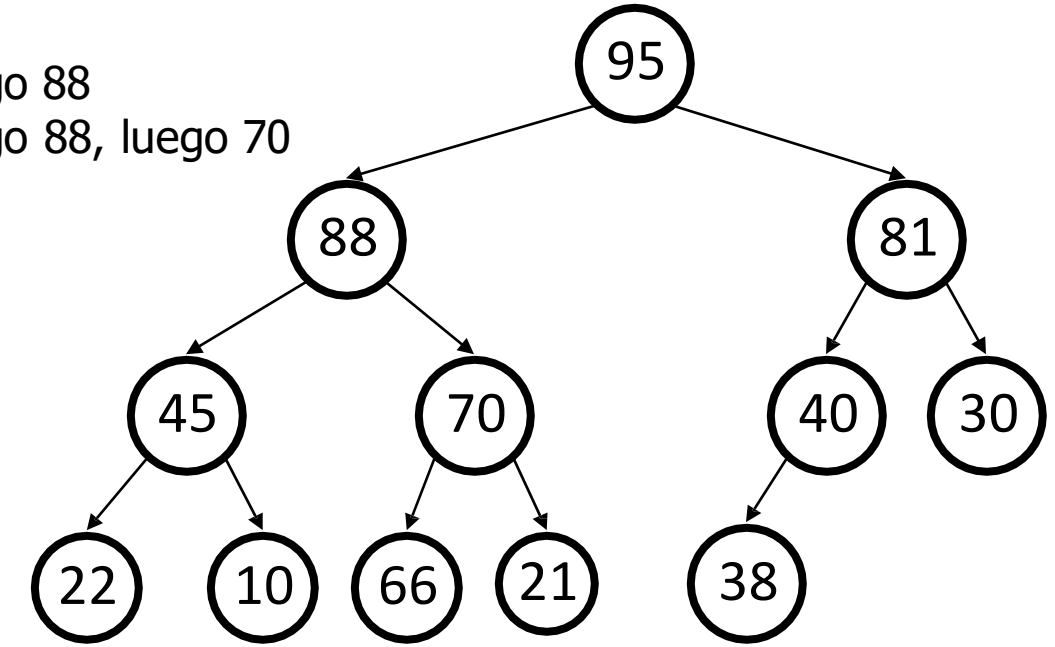
- "Burbujee" hacia abajo los nodos que no sean hoja hasta que la matriz sea un montículo *máximo* :
  - `int[] a = {21, 66, 40, 10, 70, 81, 30, 22, 45, 95, 88, 38};`
  - Intercambie cada nodo con su hijo mayor según sea necesario.



índice	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*0*	*1*	*2*	...
valor	21	66	40	10	70	81	30	22	45	95	88	38	0	...
size	12													

Construir el montículo correctamente

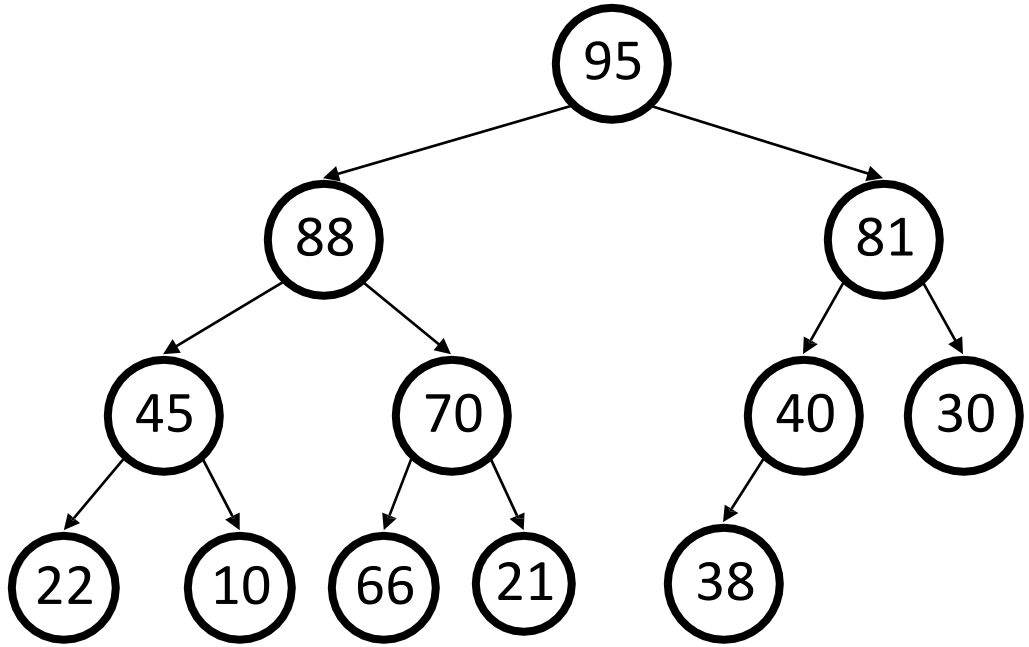
- 30: nada que hacer
- 81: nada que hacer
- 70: intercambiar con 95
- 10: intercambiar con 45
- 40: intercambiar con 81
- 66: intercambiar con 95, luego 88
- 21: intercambiar con 95, luego 88, luego 70



índice	0	1	2	3	4	5	6	7	8	9	0	1	2	...
valor	95	88	81	45	70	40	30	22	10	66	21	38	0	...
size	12													

Eliminar para ordenar

- Ahora que tenemos un montículo máximo, elimine elementos repetidamente hasta que tengamos un arreglo ordenado.
  - Mueva cada elemento eliminado hasta el final, en lugar de tirarlo.

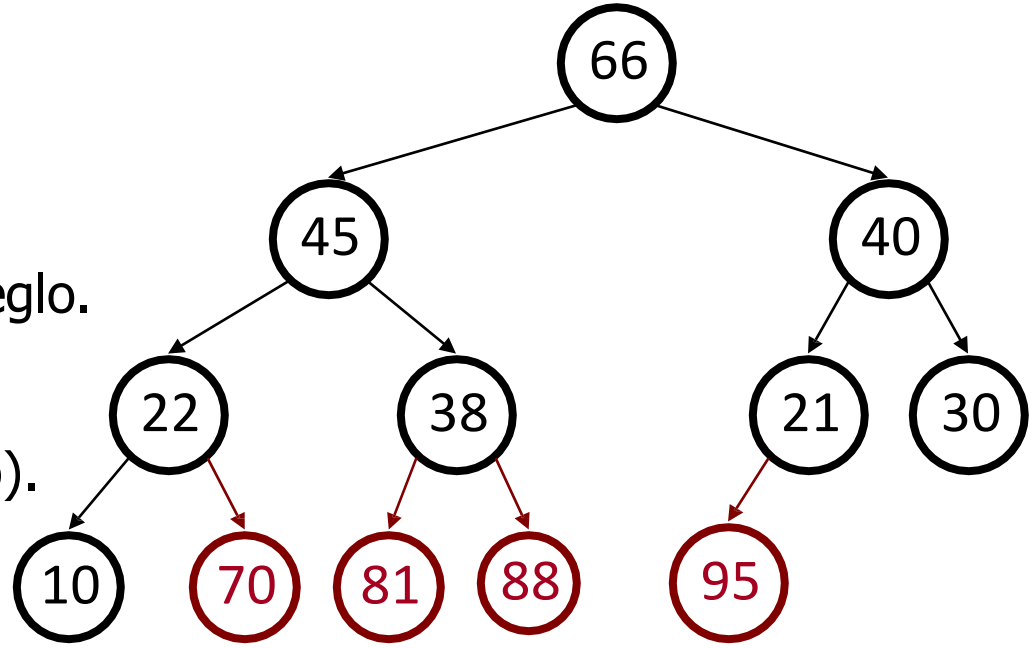


índice	0	1	2	3	4	5	6	7	8	9	0	1	2	...
valor	95	88	81	45	70	40	30	22	10	66	21	38	0	...
Talla	12													

Remove para ordenar - La respuesta

- 95: mover 38 hacia arriba, intercambiar con 88, 70, 66
- 88: mover 21 hacia arriba, intercambiar con 81, 40
- 81: mover 38 hacia arriba, intercambiar con 70, 66
- 70: mover 10 hacia arriba, intercambia con 66, 45, 22
- …

- (Observe que después de 4 eliminaciones, se ordenan los últimos 4 elementos del arreglo. Si eliminamos todos los elementos, se ordenará el arreglo completo).



índice	0	1	2	3	4	5	6	7	8	9	0	1	2	…
valor	66	45	40	22	38	21	30	10	70	81	88	95	0	…
Talla	12													

Eliminación de un montículo

Supongamos que se desea eliminar la raíz del montículo, presentado como arreglo, en forma repetida.

67	56	60	44	21	28	36	15	35	16	13	08	27	12	07	10
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Cabe aclarar que al reemplazar la raíz por el último elemento del montículo, ésta se coloca en la posición del último elemento del arreglo. Es decir, la primera vez la raíz será colocada en el índice n-1 del arreglo, la segunda vez en el índice n-2, la tercera vez en el índice n-3 y así sucesivamente hasta llegar al índice 1 y 0.

Primera eliminación:

Se intercambia la raíz, 67 con el elemento que ocupa la última posición del montículo, 10. Las comparaciones que se realizan son:
A[0] < A[2] (10 < 60) sí hay intercambio, A[2] es el mayor de los hijos de A[0]
A[2] < A[6] (10 < 36) sí hay intercambio, A[6] es el mayor de los hijos de A[2]
A[6] < A[13] (10 < 12) sí hay intercambio, A[13] es el mayor de los hijos de A[6]

Luego de eliminar la primera raíz, el montículo queda así:

60	56	36	44	21	28	12	15	35	16	13	08	27	10	07	67
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

!El mayor se ubicó en la última posición!

Eliminación de un montículo

Segunda eliminación:

Se intercambia la raíz, 60 con el elemento que ocupa la última posición del montículo, 07. Las comparaciones que se realizan son:
A[0] < A[2] (07 < 56) sí hay intercambio, A[1] es el mayor de los hijos de A[0]
A[1] < A[3] (07 < 44) sí hay intercambio, A[3] es el mayor de los hijos de A[1]
A[3] < A[8] (07 < 35) sí hay intercambio, A[8] es el mayor de los hijos de A[3]

Luego de eliminar la segunda raíz, el montículo queda así:

56	44	36	35	21	28	12	15	07	16	13	08	27	10	60	67
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

!El mayor se ubicó en la última posición!

Tercera eliminación:

Se intercambia la raíz, 56 con el elemento que ocupa la última posición del montículo, 10. Las comparaciones que se realizan son:
A[0] < A[1] (10 < 44) sí hay intercambio, A[1] es el mayor de los hijos de A[0]
A[1] < A[3] (10 < 35) sí hay intercambio, A[3] es el mayor de los hijos de A[1]
A[3] < A[7] (10 < 15) sí hay intercambio, A[7] es el mayor de los hijos de A[3]

Luego de eliminar la tercera raíz, el montículo queda así:

44	35	36	15	21	28	12	10	07	16	13	08	27	56	60	67
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Eliminación de un montículo

Se presenta el resultado de las restantes eliminaciones. Luego de eliminar la raíz del montículo, en forma repetida, el arreglo queda ordenado

Eliminación	Montículo															
4	36	35	28	15	21	27	12	10	07	16	13	08	44	56	60	67
5	35	21	28	15	16	27	12	10	07	08	13	36	44	56	60	67
6	28	21	27	15	16	13	12	10	07	08	35	36	44	56	60	67
7	27	21	13	15	16	08	12	10	07	28	35	36	44	56	60	67
8	21	16	13	15	07	08	12	10	27	28	35	36	44	56	60	67
9	16	15	13	10	07	08	12	21	27	28	35	36	44	56	60	67
10	15	12	13	10	07	08	16	21	27	28	35	36	44	56	60	67
11	13	12	08	10	07	15	16	21	27	28	35	36	44	56	60	67
12	12	10	08	07	13	15	16	21	27	28	35	36	44	56	60	67
13	10	07	08	12	13	15	16	21	27	28	35	36	44	56	60	67
14	08	07	10	12	13	15	16	21	27	28	35	36	44	56	60	67
15	07	08	10	12	13	15	16	21	27	28	35	36	44	56	60	67

¿Preguntas?

Resumiendo y Repasando…

- Una cola de prioridad permite insertar elementos en cualquier orden y eliminarlos en orden de prioridad (por ejemplo ascendente o descendente).
- A menudo se utiliza una estructura denominada montículo para implementar colas de prioridad. Un monticulo es un árbol con un orden vertical, donde los padres almacenan valores más pequeños que sus hijos (min-heap) o valores más grandes que sus hijos (max-heap).
- Al agregar un valor a un montículo, se agrega como una hoja y luego se "burbujea" (se intercambia) en el árbol hasta que se encuentra en el lugar adecuado en el ordenamiento vertical.

Resumiendo y Repasando…

- Al eliminar un valor de un montículo, la hoja más a la derecha del montículo se mueve a la raíz y luego "burbujea" hacia abajo en el árbol hasta que se encuentra en un lugar adecuado en el ordenamiento vertical.
- Debido a su estructura completa, un montículo se puede implementar de manera eficiente utilizando un arreglo, en lugar de un árbol enlazado de objetos nodo.
- Los montículos se pueden usar para ayudar a ordenar los datos, un algoritmo llamado heapsort.

FIN