Colas de prioridad

Colas de prioridad

- Si pensamos en la cola que se forma en una caja de supermercado con prioridad para mujeres embarazadas.
- En general, en este tipo de cajas también se admiten hombres y mujeres no embarazadas que serán atendidos según el orden de llegada.
- Sin embargo, si llega una mujer embarazada la cajera le dará el primer lugar y la atenderá inmediatamente.
- A este tipo de colas se les llama “colas jerarquizadas” o “colas por prioridad”.

- Muchas aplicaciones requieren que procesemos elementos con claves en orden, pero no necesariamente en orden completo y no necesariamente todos a la vez.

- A menudo, recopilamos un conjunto de elementos, luego se procesa el que tenga la clave más grande, luego tal vez se recolecten más elementos, luego se procesa el que tiene la clave más grande actual, y así sucesivamente.

- Por ejemplo, es probable tener una computadora (o un teléfono celular) que sea capaz de ejecutar varias aplicaciones al mismo tiempo.
- Esto se logra típicamente asignando una prioridad a los eventos asociados con las aplicaciones, y luego siempre se elige procesar a continuación el evento de mayor prioridad.
- Por ejemplo, es probable que la mayoría de los teléfonos móviles procesen una llamada entrante con mayor prioridad que una aplicación de juego.

- Una cola de prioridad es una estructura de datos para mantener un conjunto S de elementos, cada uno con un valor asociado llamado clave.

- Admite dos operaciones: eliminar el máximo e insertar nuevos elementos.

- Usando colas de prioridad es similar a usar colas (eliminar los elementos más antiguos) y pilas (eliminar los elementos más nuevos), pero implementarlos de manera eficiente es más desafiante.

- Un algoritmo importante de ordenamiento conocido como **heapsort** también se deriva de la implementación de una cola de prioridad.

- Las colas de prioridad vienen en dos formas: colas de máxima prioridad y colas de mínima prioridad.

```
PARENT(i)
1  return ⌊i/2⌋

LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4       largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7       largest = r
 8  if largest ≠ i
 9       exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3       MAX-HEAPIFY(A, i)
```

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3       exchange A[1] with A[i]
4       A.heap-size = A.heap-size − 1
5       MAX-HEAPIFY(A, 1)
```

FUNCIONES PARA CREAR UN MONTÍCULO - HEAPS

```
PARENT(i)
1  return ⌊i/2⌋

LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```

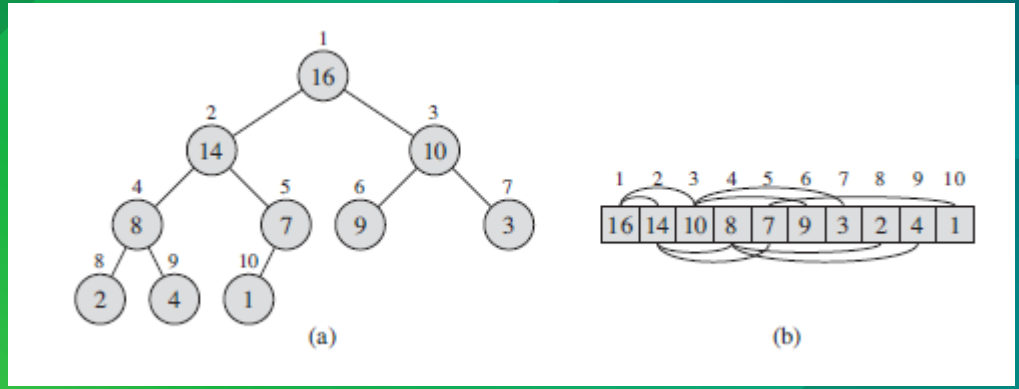
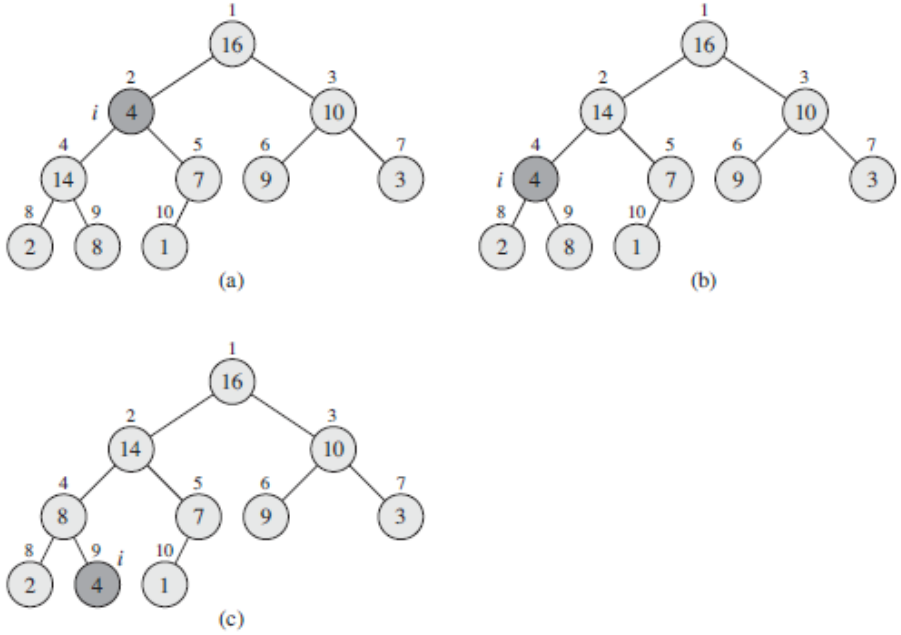


Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

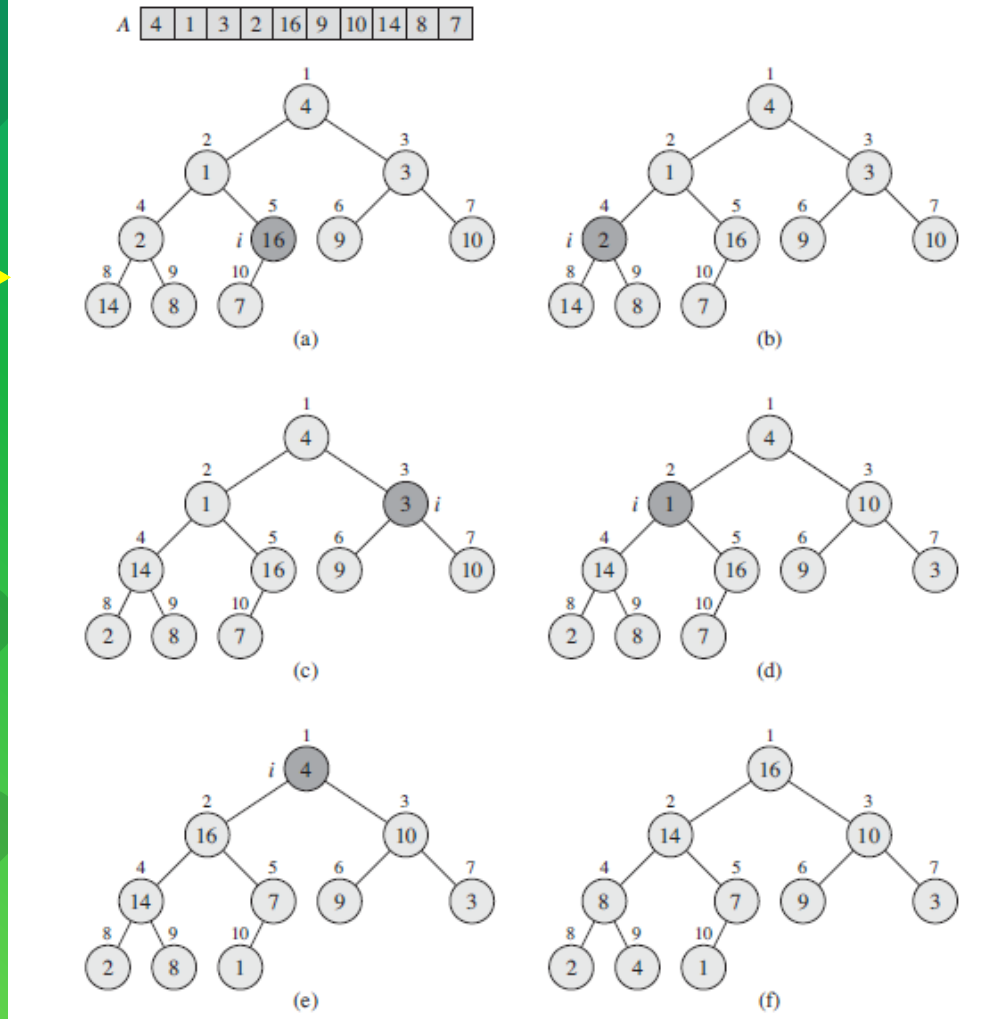



```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8  if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. (**a**) The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in (**b**) by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY$(A, 4)$ now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in (**c**), node 4 is fixed up, and the recursive call MAX-HEAPIFY$(A, 9)$ yields no further change to the data structure.

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

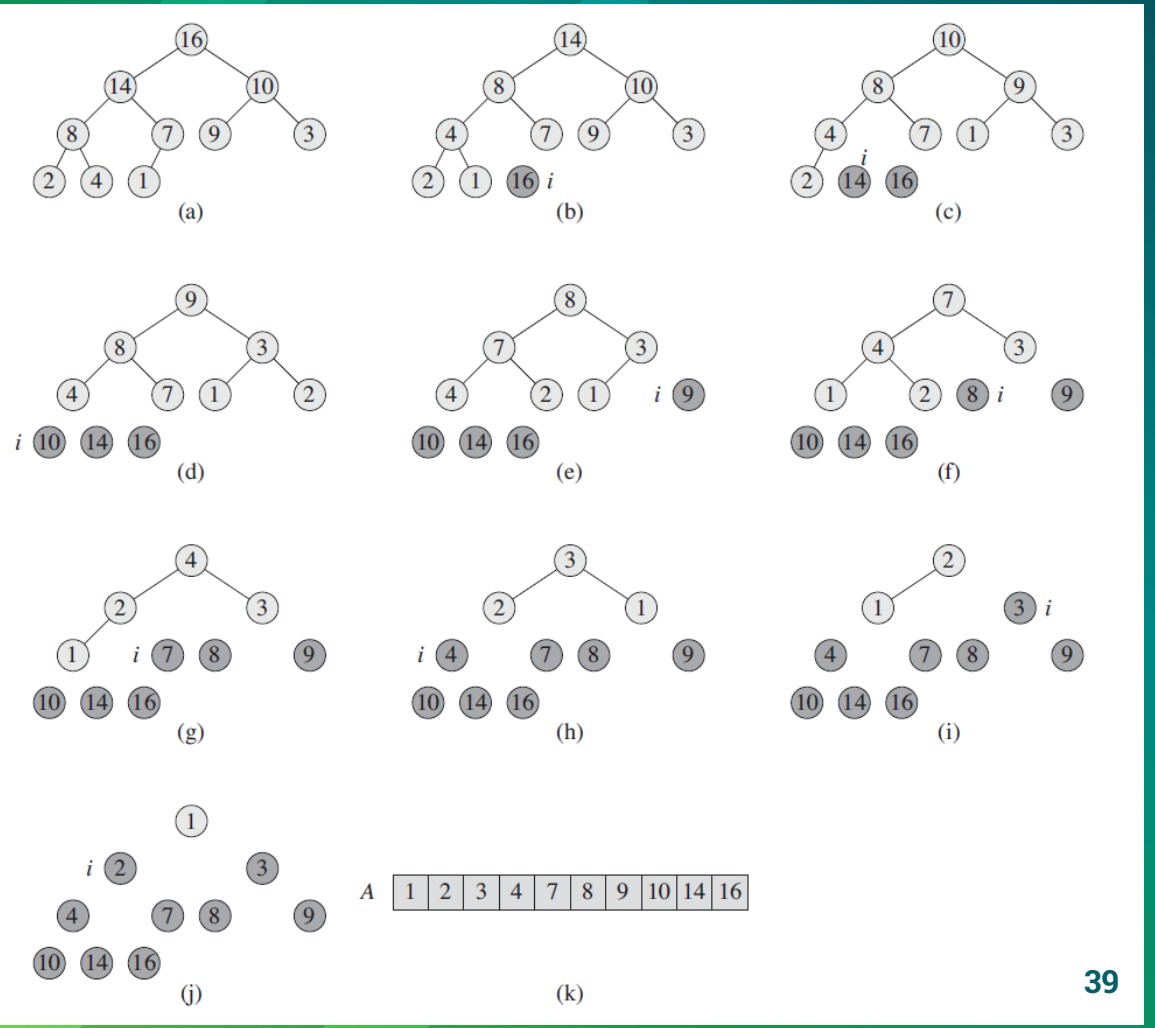
Los elementos en el subarreglo **A[**(⌊n/2⌋ +1) ... n**]** son hojas del árbol.



FUNCIÓN DEL ALGORITMO DE ORDENAMIENTO HEAPSORT

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

The operation of HEAPSORT.
(a) The max-heap data structure just after BUILD-MAXHEAP has built it in line 1.
(b)–(j) The max-heap just after each call of MAX-HEAPIFY in line 5, showing the value of i at that time. Only lightly shaded nodes remain in the heap.
(k) The resulting sorted array A.



- Una cola de máxima prioridad admite las siguiente operaciones:
  - INSERT(S, x) inserta el elemento x en el conjunto S, que es equivalente a la operación $S = S \cup \{x\}$.
  - MÁXIMO(S) devuelve el elemento de S con la clave más grande.
  - EXTRACT-MAX(S) elimina y devuelve el elemento de S con la clave más grande.
  - INCREMENT-KEY(S, x, k) aumenta el valor de la clave del elemento x al nuevo valor k, que se supone que es al menos tan grande como el valor clave actual de x.

FUNCIONES DE UNA COLA DE PRIORIDAD

```
HEAP-MAXIMUM(A)
1   return A[1]
```

```
HEAP-EXTRACT-MAX(A)
1   if A.heap-size < 1
2         error "heap underflow"
3   max = A[1]
4   A[1] = A[A.heap-size]
5   A.heap-size = A.heap-size − 1
6   MAX-HEAPIFY(A, 1)
7   return max
```

Similar a HeapSort

```
HEAP-INCREASE-KEY(A, i, key)
1   if key < A[i]
2         error "new key is smaller than current key"
3   A[i] = key
4   while i > 1 and A[PARENT(i)] < A[i]
5         exchange A[i] with A[PARENT(i)]
6         i = PARENT(i)
```

```
MAX-HEAP-INSERT(A, key)
1   A.heap-size = A.heap-size + 1
2   A[A.heap-size] = −∞
3   HEAP-INCREASE-KEY(A, A.heap-size, key)
```

FUNCIÓN HEAP-INCREASE-KEY

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2       error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5       exchange A[i] with A[PARENT(i)]
6       i = PARENT(i)
```

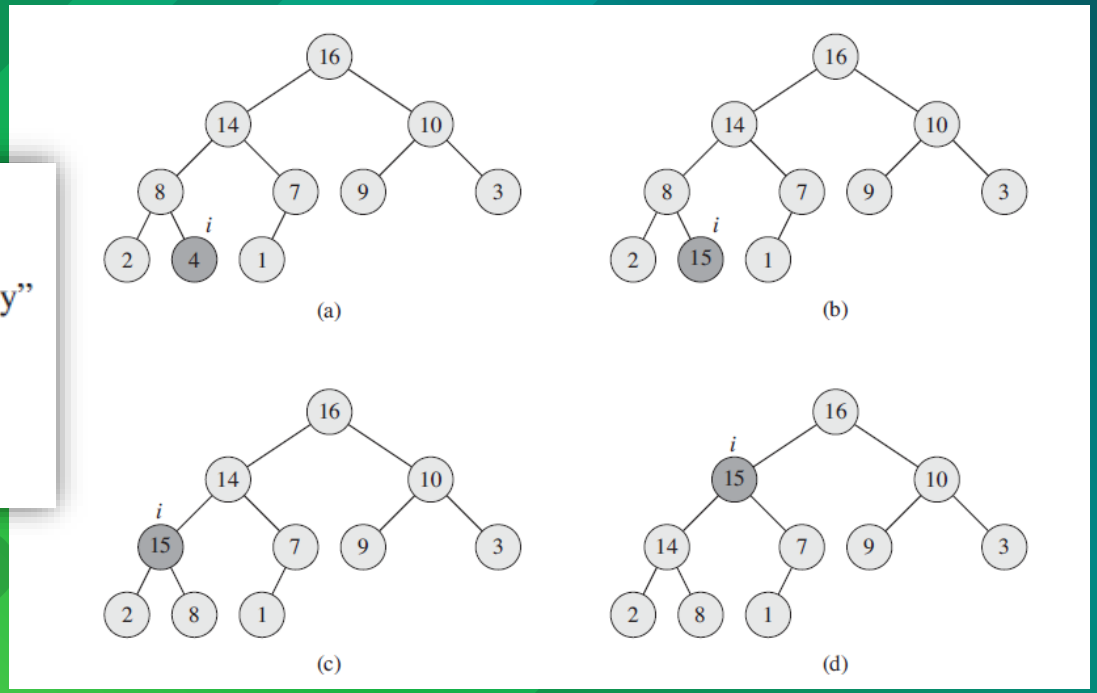


Figure 6.5 The operation of HEAP-INCREASE-KEY. **(a)** The max-heap of Figure 6.4(a) with a node whose index is i heavily shaded. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 4–6, the node and its parent have exchanged keys, and the index i moves up to the parent. **(d)** The max-heap after one more iteration of the **while** loop. At this point, $A[\text{PARENT}(i)] \geq A[i]$. The max-heap property now holds and the procedure terminates.

FUNCIONES DE UNA COLA DE PRIORIDAD

```
MAX-HEAP-INSERT(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = −∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

El procedimiento MAX-HEAP-INSERT implementa la operación de inserción. Toma como entrada la llave (key) del nuevo elemento que será insertado dentro de **max-heap A**.
El procedimiento primero expande el **max-heap** agregando al árbol una nueva hoja cuya clave es -∞.
Luego llama **HEAP-INCREASE-KEY** para establecer la clave de este nuevo nodo a su valor correcto y mantener la propiedad max-heap.

- Alternativamente, una cola de prioridad mínima admite las operaciones:
  - INSERT MINIMUM
  - EXTRACT-MIN
  - DECREASE-KEY.