Fibonacci Heaps

Prof. Rhadamés Carmona

Última revisión: 06/05/2019

Agenda

- Motivación
- Estructura de datos
- Operaciones
  - Crear, Insertar Key
  - Union, Obtener Mínimo
  - Extraer mínimo, Decrementar Key
- Conclusiones
- Tarea
- Referencias

Motivación

- Fibonacci heap es una estructura de datos similar a una cola de prioridad; se puede ver como una colección de heaps.
- El nombre de Fibonacci aparece porque en el estudio de la complejidad del algoritmos aparecen estos números de Fibonacci.
- Útiles cuando se hacen más inserciones y "decrementos" de claves $\Theta(1)$ que eliminaciones O(logn), donde Θ es el "orden amortizado".

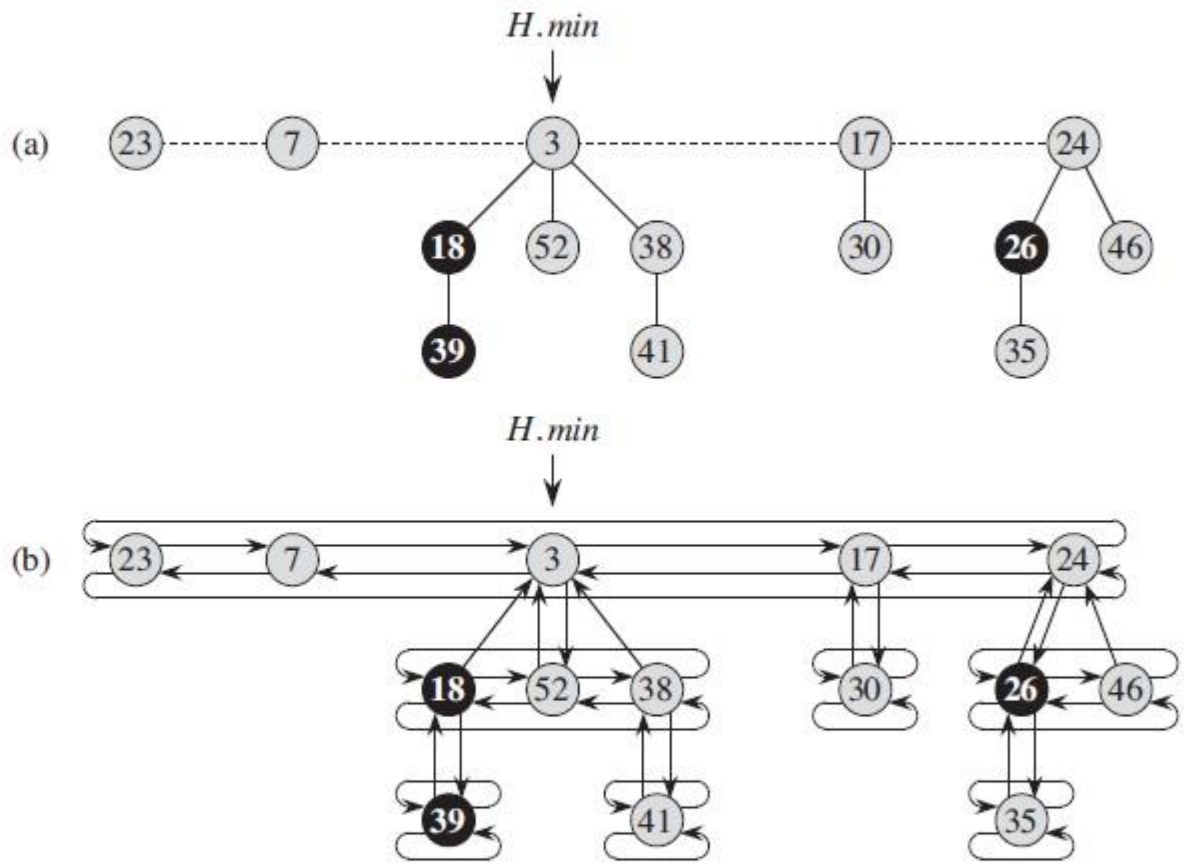
Motivación

- Mejora considerablemente el tiempo del algoritmo de camino más corto entre 2 nodos (Dijktra's algorithm), de E.logV → E+V.logV
- Mejora el tiempo del mínimo spanning tree.
- Todas estas operaciones son $\Theta(1)$: crear, insertar, buscar mínimo, decrementar clave, unión.
- Eliminar el mínimo es O(log).

Estructura de datos

- Es un bosque de heaps.
- Hay un apuntador a la raíz mínima.
- Las raíces de los heaps se conectan mediante una lista circular doblemente enlazada.
- Los hijos de cualquier nodo se conectan entre sí de manera similar.
- Adicionalmente, cada nodo tiene un apuntador a su nodo padre, y un apuntador a uno de sus hijos.

Estructura de datos



Estructura de datos

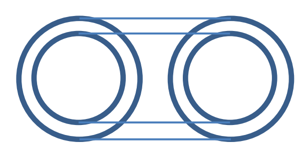
- Es una colección de árboles con la propiedad de max-heap (o min-heap).
- A diferencia del heap boinomial (compuestos de árboles binomiales) los árboles pueden tener cualquier forma.
- Ejecuta operaciones en “lazy way”, osea, perezoso. Posterga actualizaciones complejas para que la operación extraer el mínimo requiera “consolidar” árboles.

Estructura de datos

- Los nodos marcados se usarán para postergar cierto procesamiento.
  - Un nodo x se marca si ha perdido un hijo desde la última vez que x se hizo hijo de otro nodo.
  - Nodos recién creados están desmarcados.
  - Cuando x se hace hijo de otro nodo, se desmarca.
- El último nivel de los heaps no se llenan de izquierda a derecha como en un heap clásico.
- De cada nodo se guarda el “degree” o grado (número de hijos), una marca (bool) y 4 apuntadores.

Operaciones

- **Constructor**: simplemente se coloca minH = NULL y size = 0. O(1).
- **GetMin**: retorna minH. O(1).
- **Union**: h1 = h1 U h2. Une las dos listas de raíces, modificando 4 apuntadores. Luego:
  – If (minH->key > h2.minH->key) minH=h2.minH;
  – size += h2.size;
  – O(1)

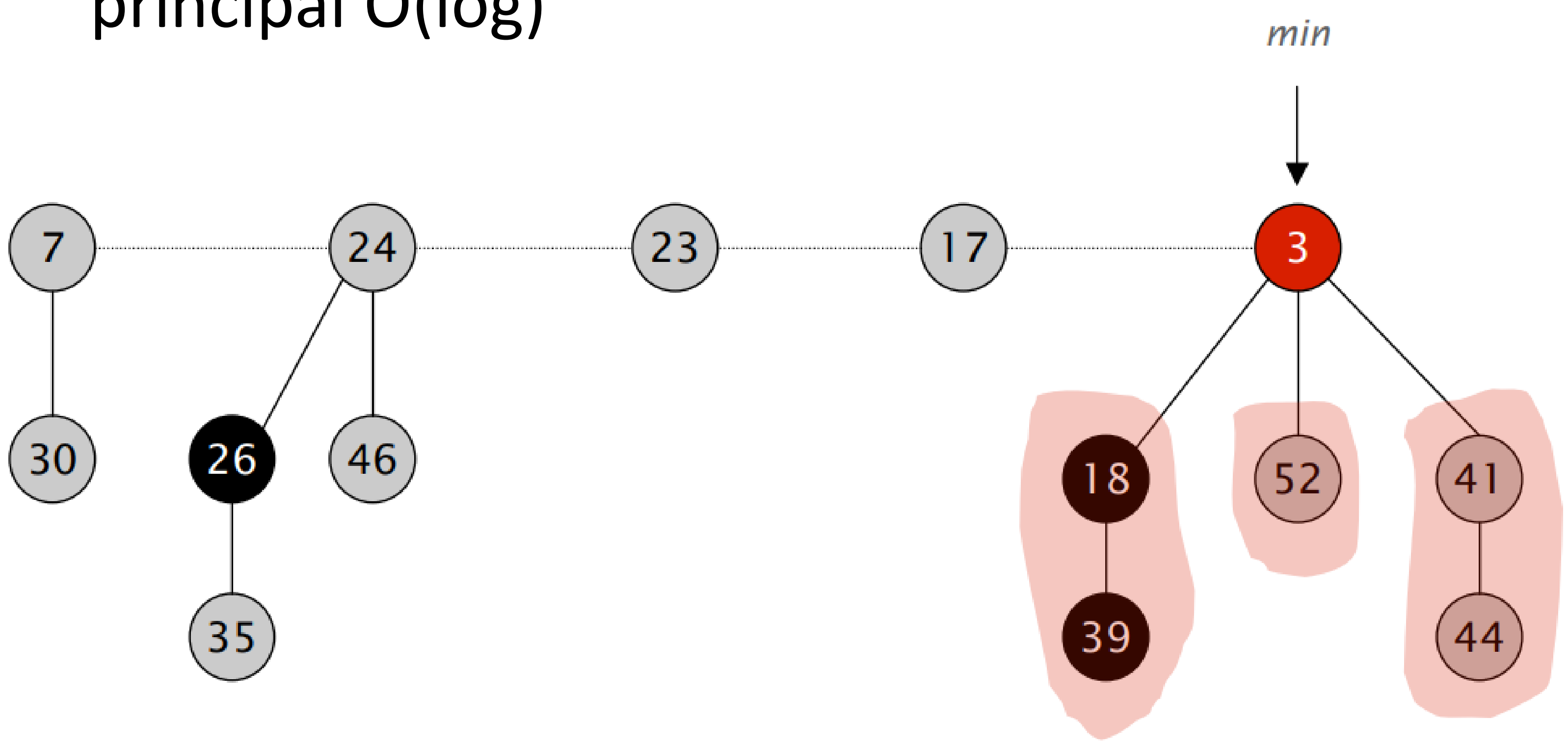


Operaciones

- **Insertar** (key):
  - Crea un nodo x
  - Lo inserta en la lista circular de raíces
  - Actualiza minH=(minH->key < x->key) ? minH:x;
  - Actualiza size++;
  - O(1)

Operaciones

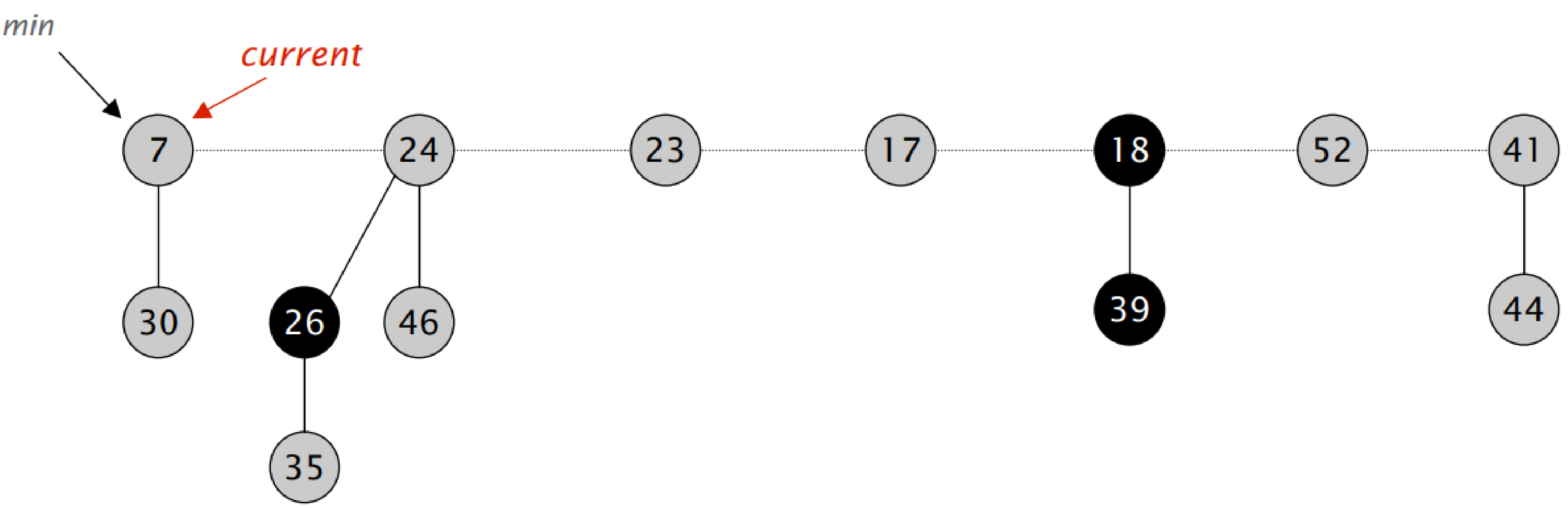
- **Extraer/Eliminar** mínimo:
  (1) Elimina el nodo, y coloca sus hijos en la lista principal O(log)



Operaciones

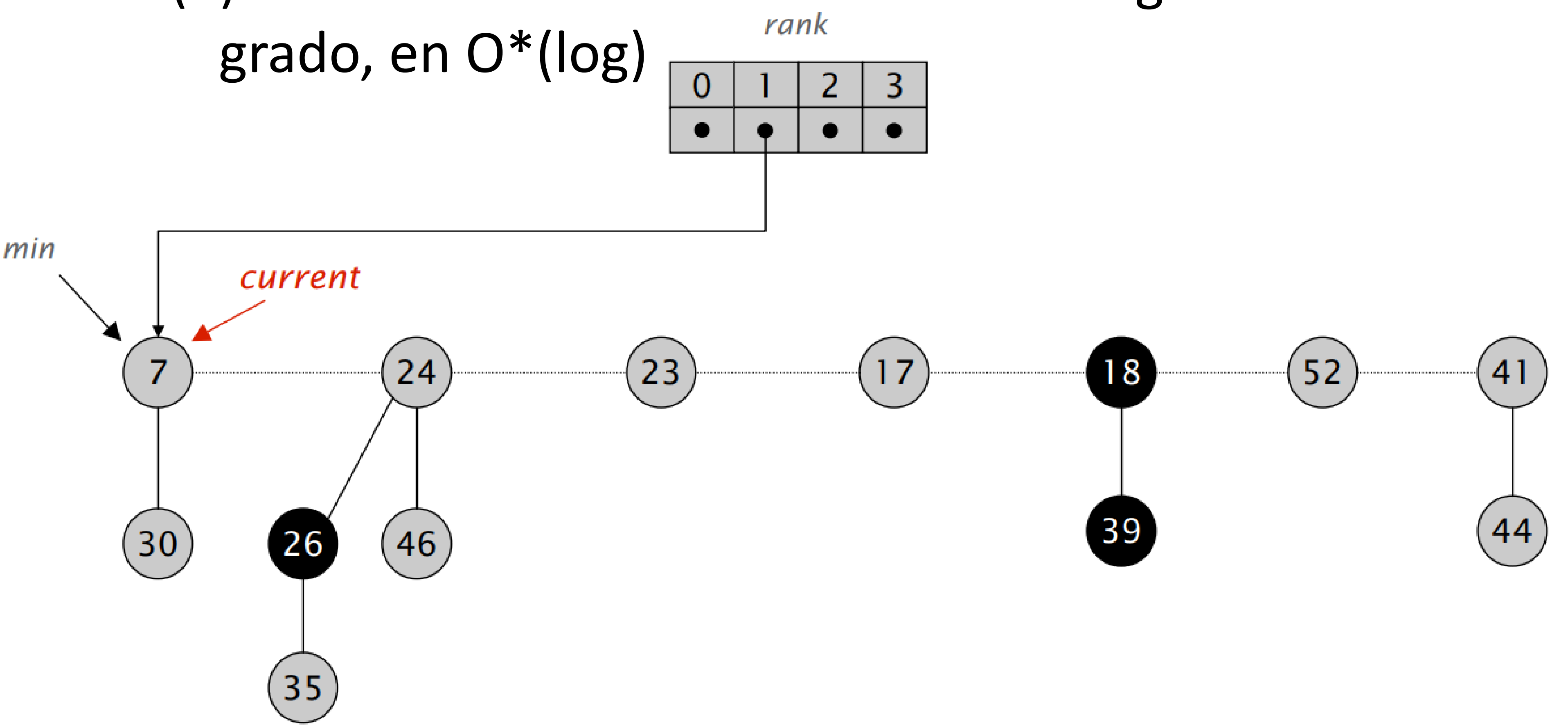
- **Extraer/Eliminar** mínimo:

  (2) Actualiza el mínimo chequeando los O(log) nuevas raíces insertadas contra en viejo mínimo.



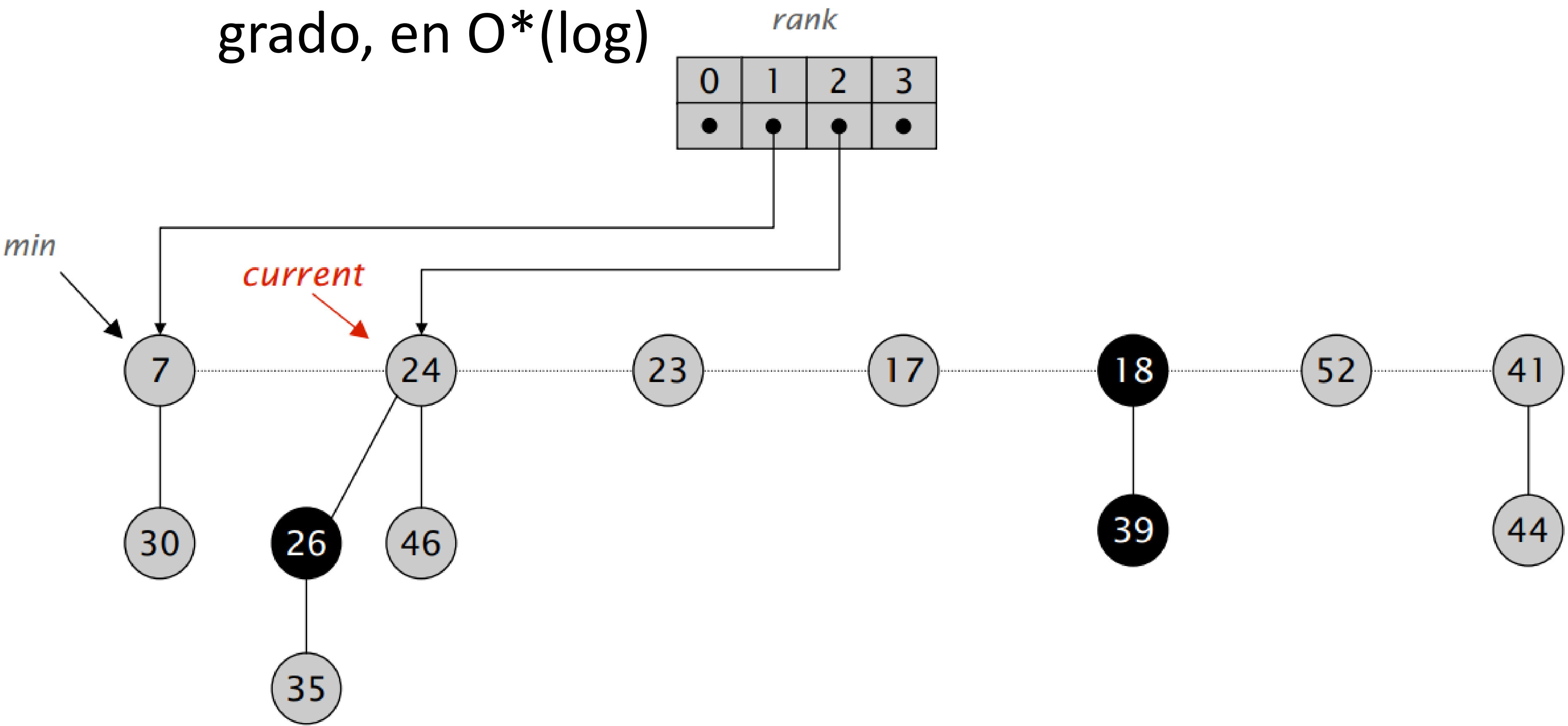
Operaciones

- **Extraer/Eliminar** mínimo:
  - (3) Consolida árboles--> 2 raíces no tengan el mismo grado, en O*(log)



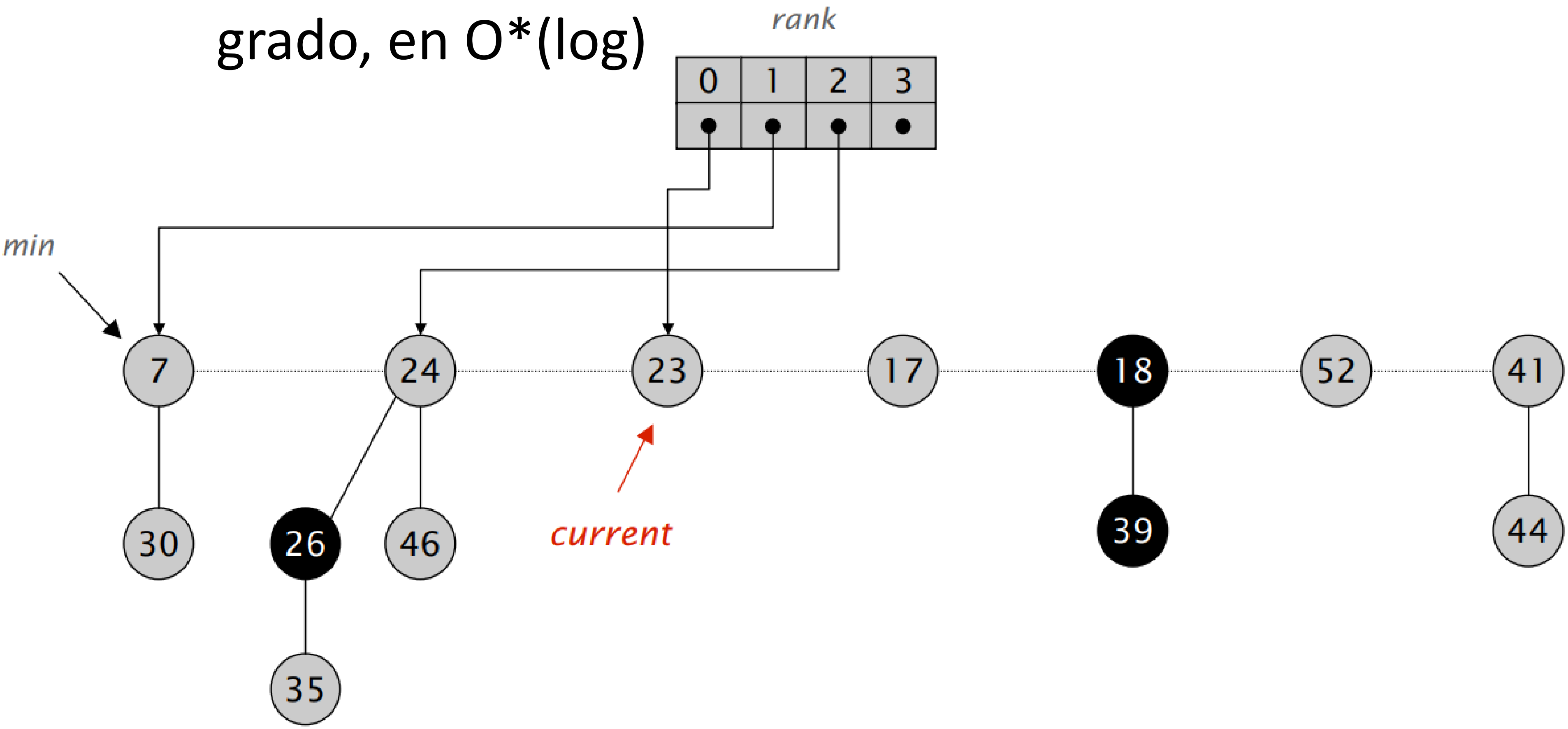
Operaciones

- **Extraer/Eliminar** mínimo:
  (3) Consolida árboles--> 2 raíces no tengan el mismo grado, en O*(log)



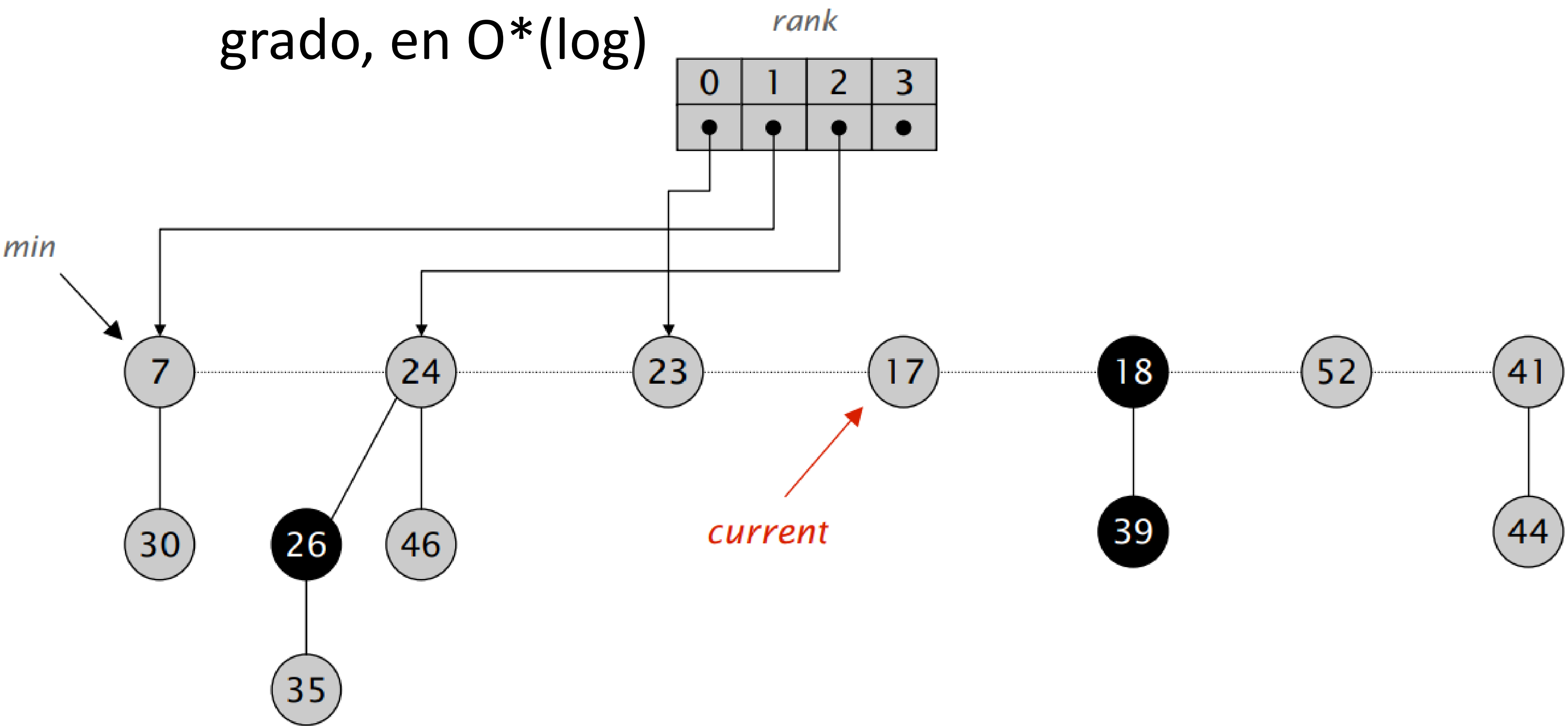
Operaciones

- **Extraer/Eliminar** mínimo:
  (3) Consolida árboles--> 2 raíces no tengan el mismo grado, en O*(log)



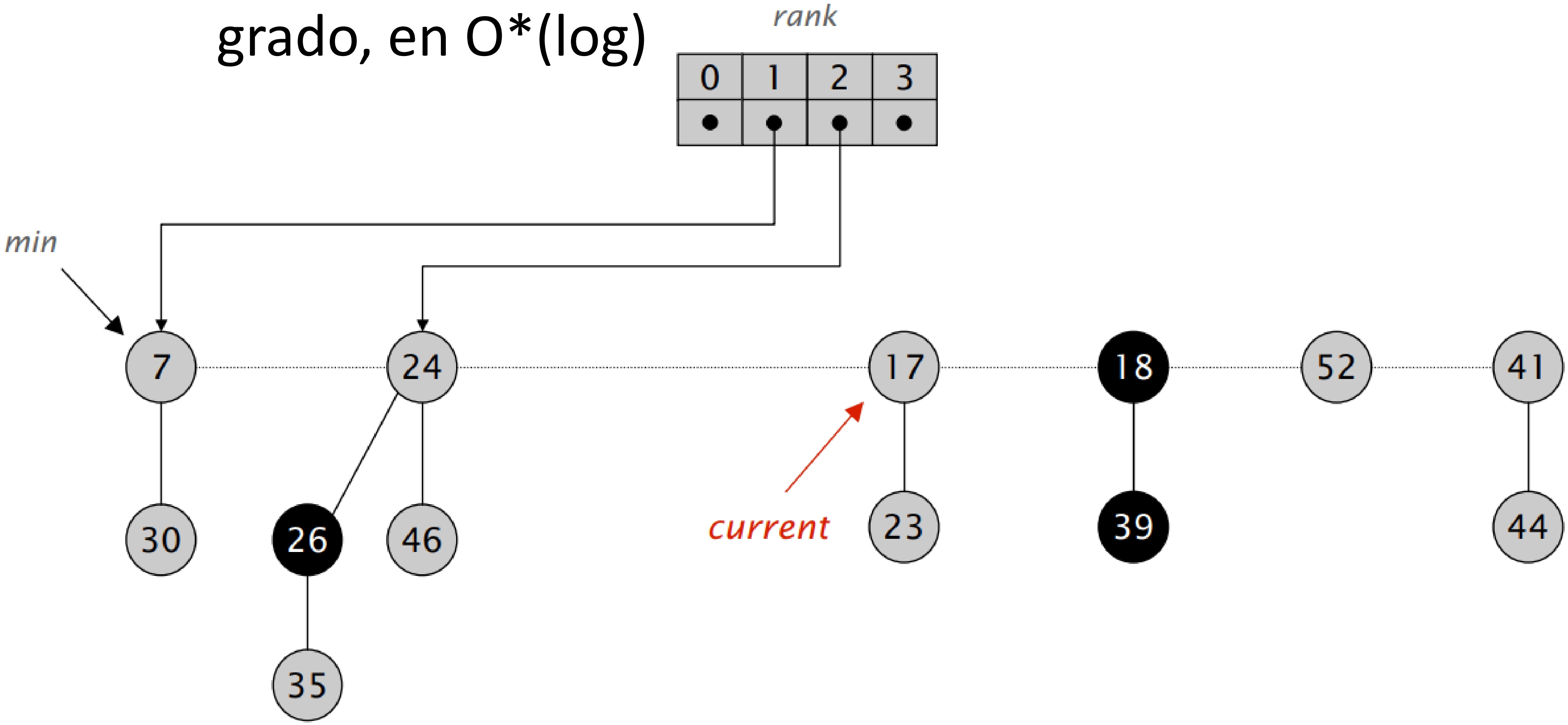
Operaciones

- **Extraer/Eliminar** mínimo:
  - (3) Consolida árboles--> 2 raíces no tengan el mismo grado, en O*(log)



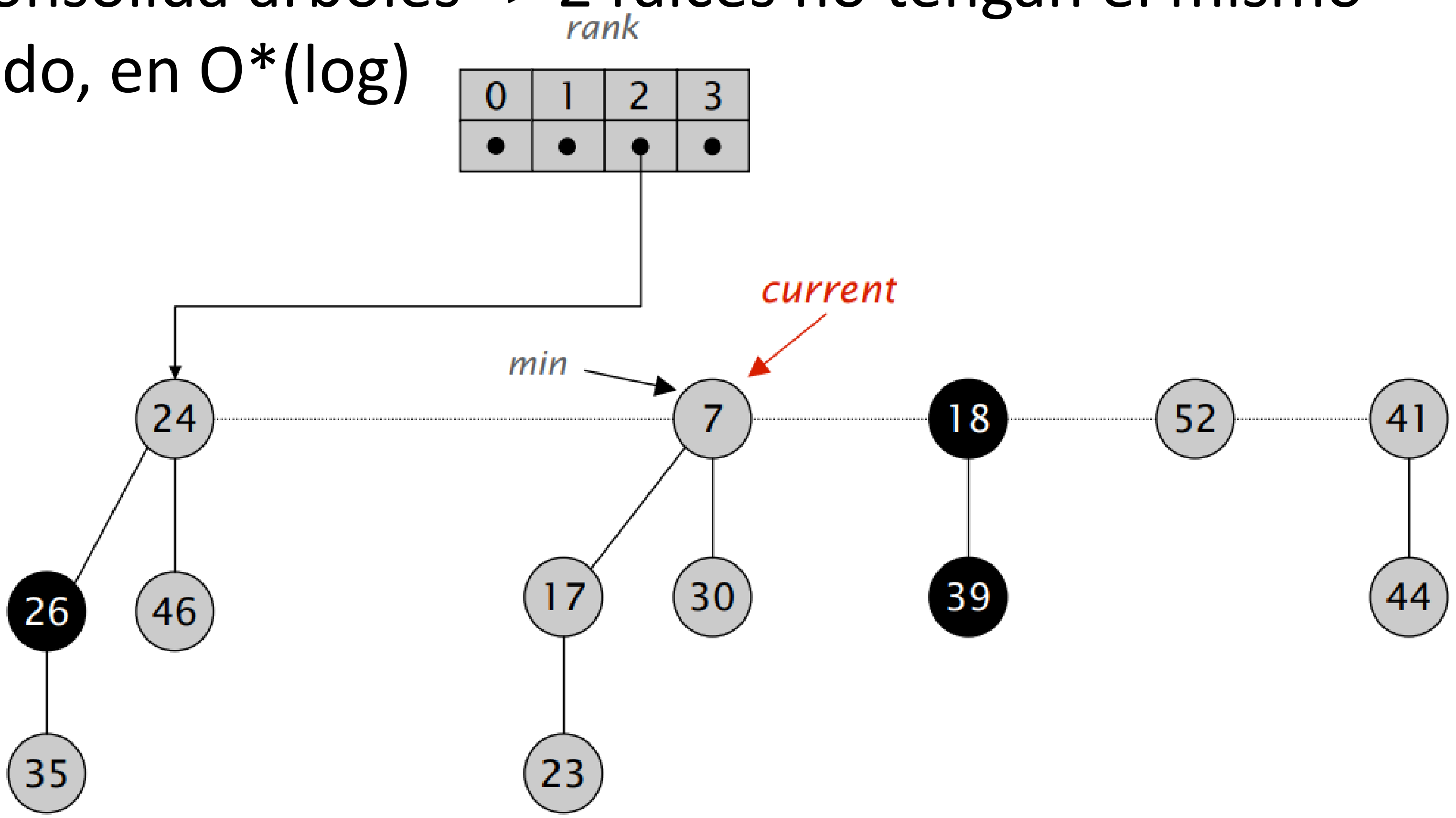
Operaciones

- **Extraer/Eliminar** mínimo:
  (3) Consolida árboles--> 2 raíces no tengan el mismo grado, en O*(log)



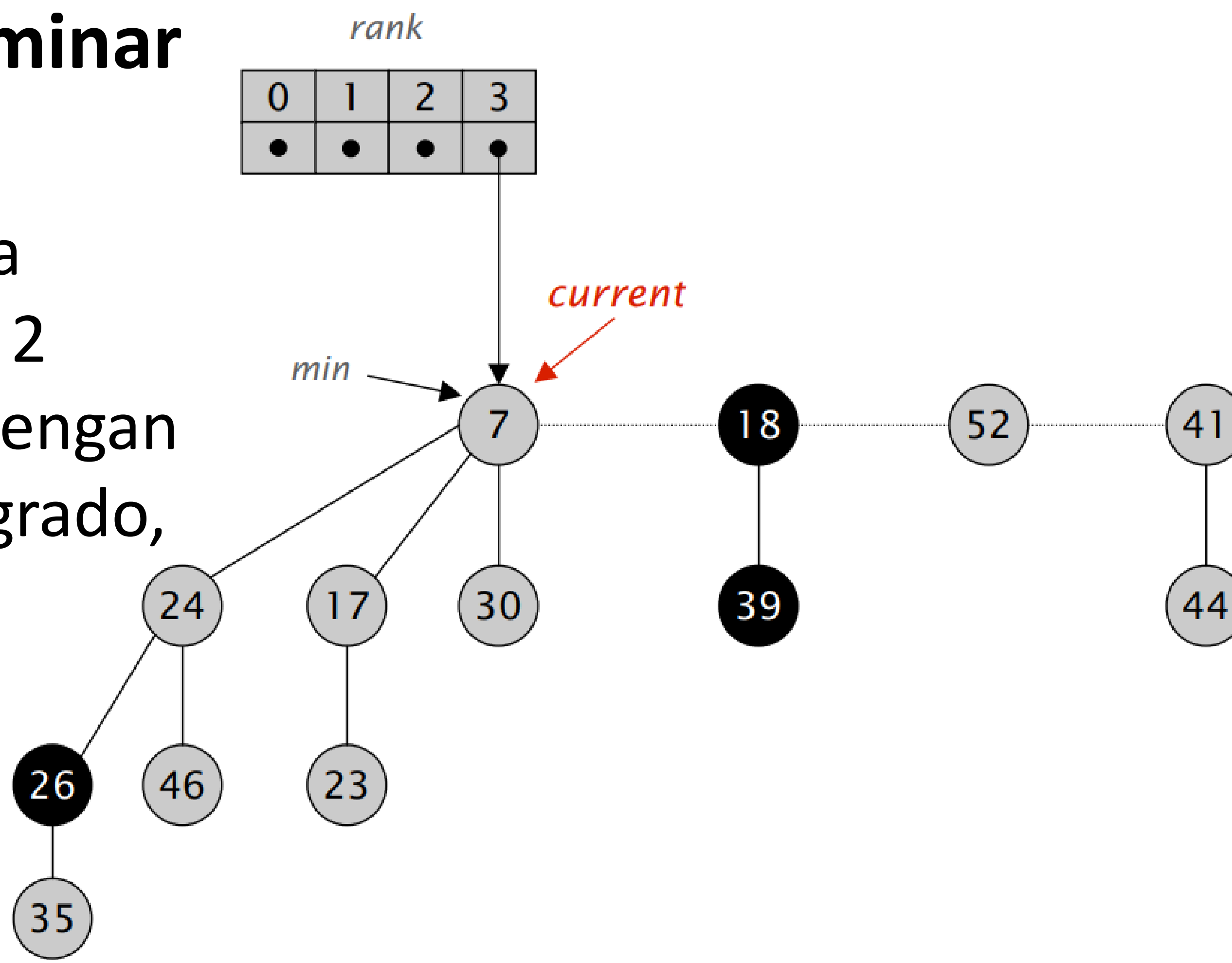
Operaciones

- **Extraer/Eliminar** mínimo:
  (3) Consolida árboles--> 2 raíces no tengan el mismo grado, en O*(log)



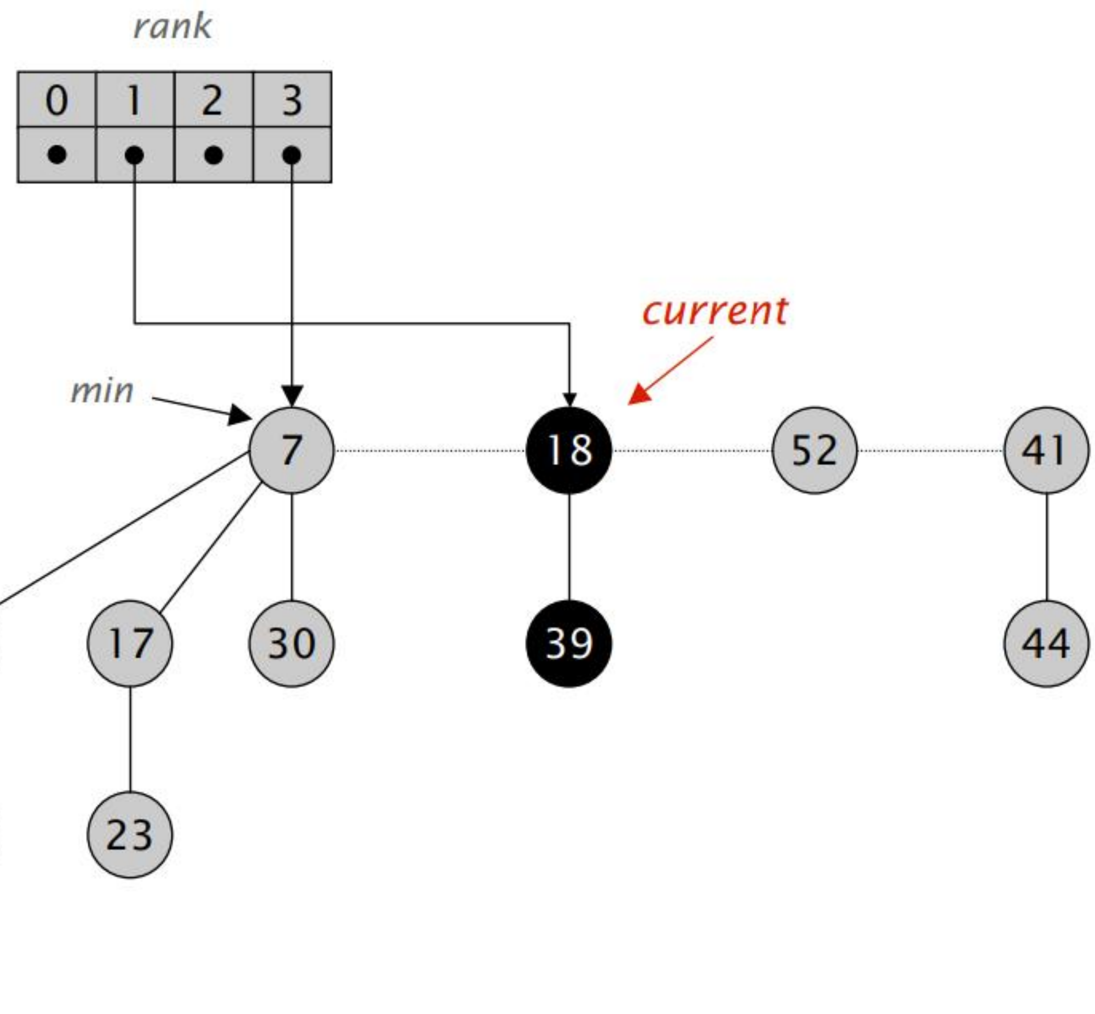
Operaciones

- **Extraer/Eliminar** mínimo:
  (3) Consolida árboles--> 2 raíces no tengan el mismo grado, en O*(log)



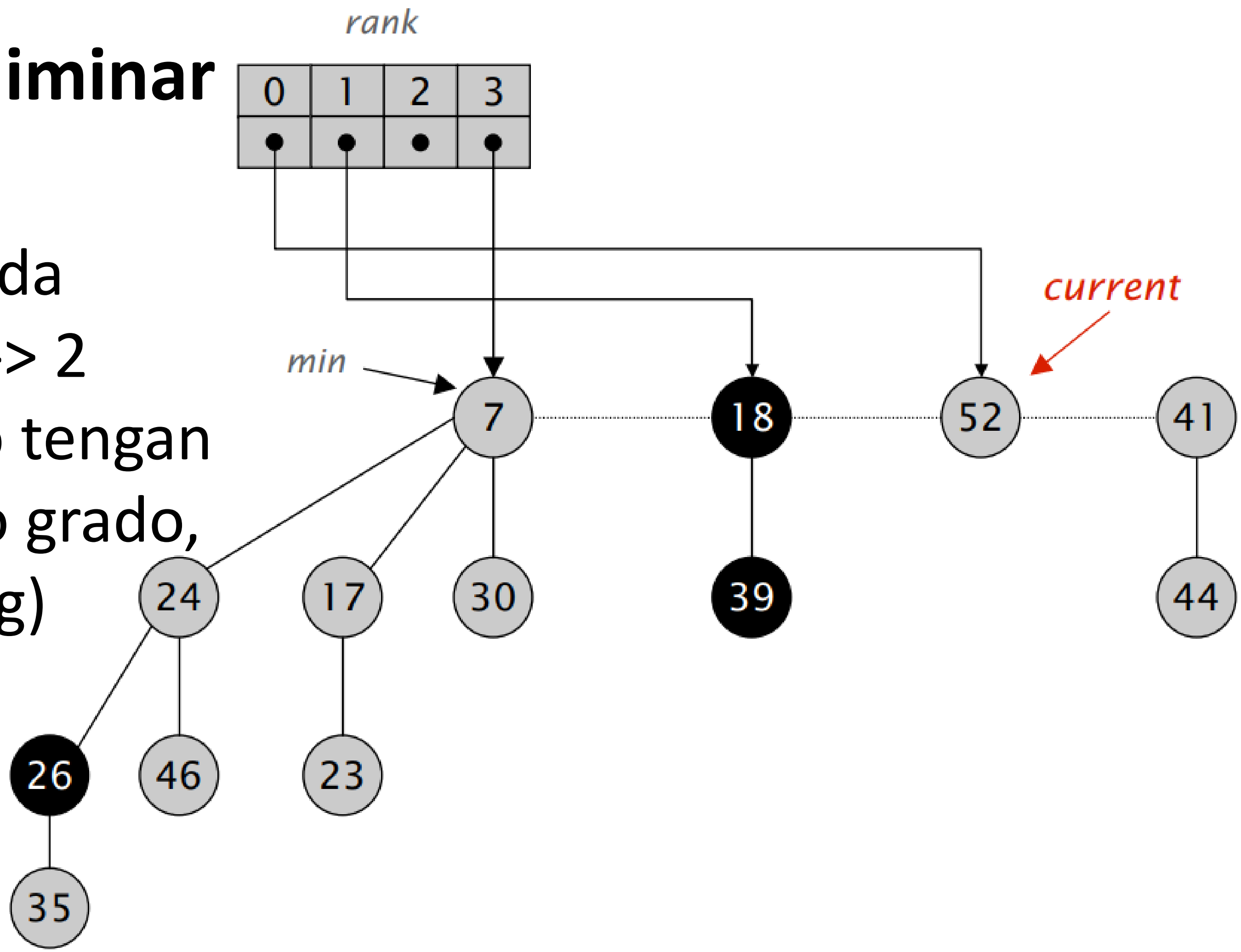
Operaciones

- **Extraer/Eliminar** mínimo:
  - (3) Consolida árboles--> 2 raíces no tengan el mismo grado, en O*(log)



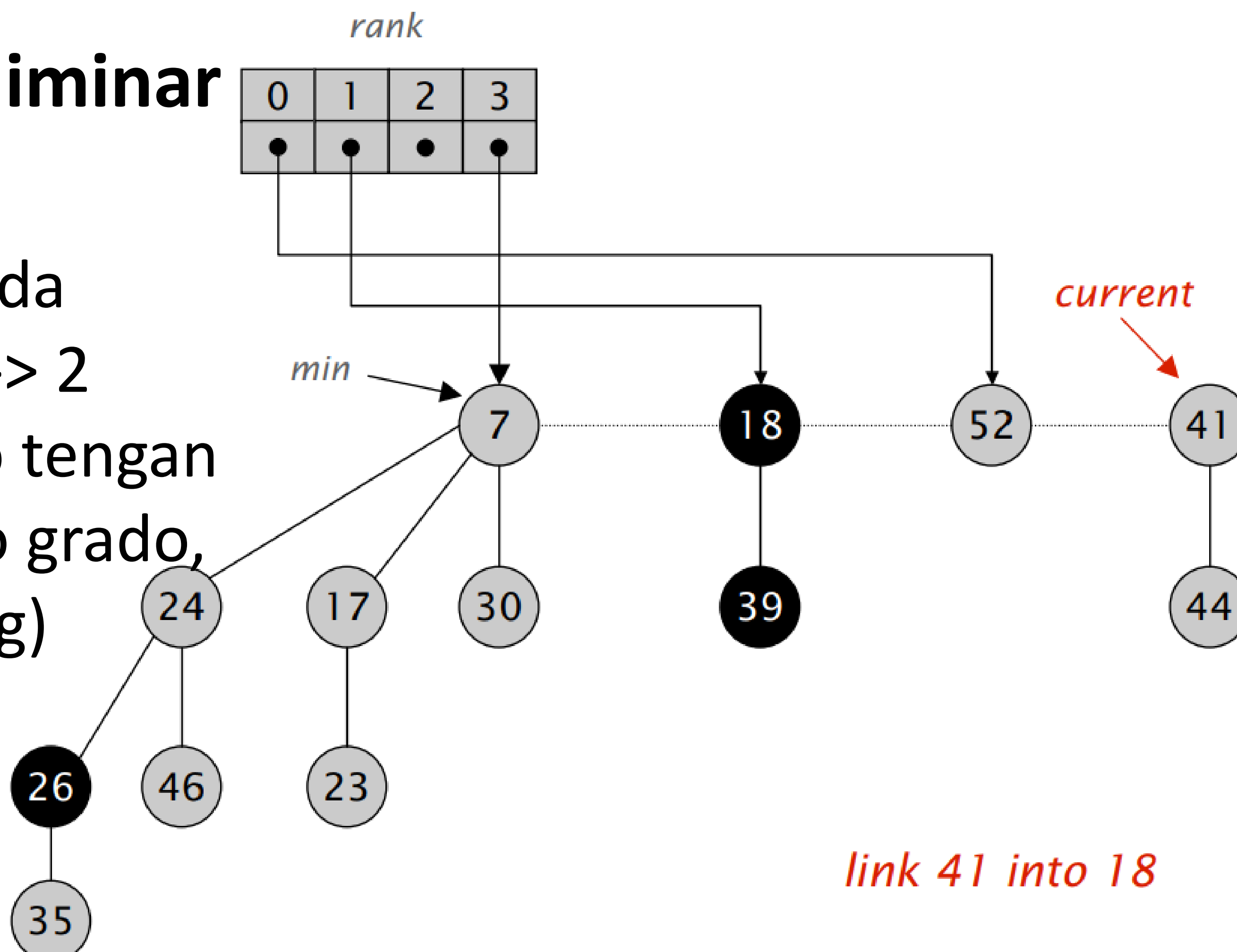
Operaciones

- **Extraer/Eliminar** mínimo:
  - (3) Consolida árboles--> 2 raíces no tengan el mismo grado, en O*(log)



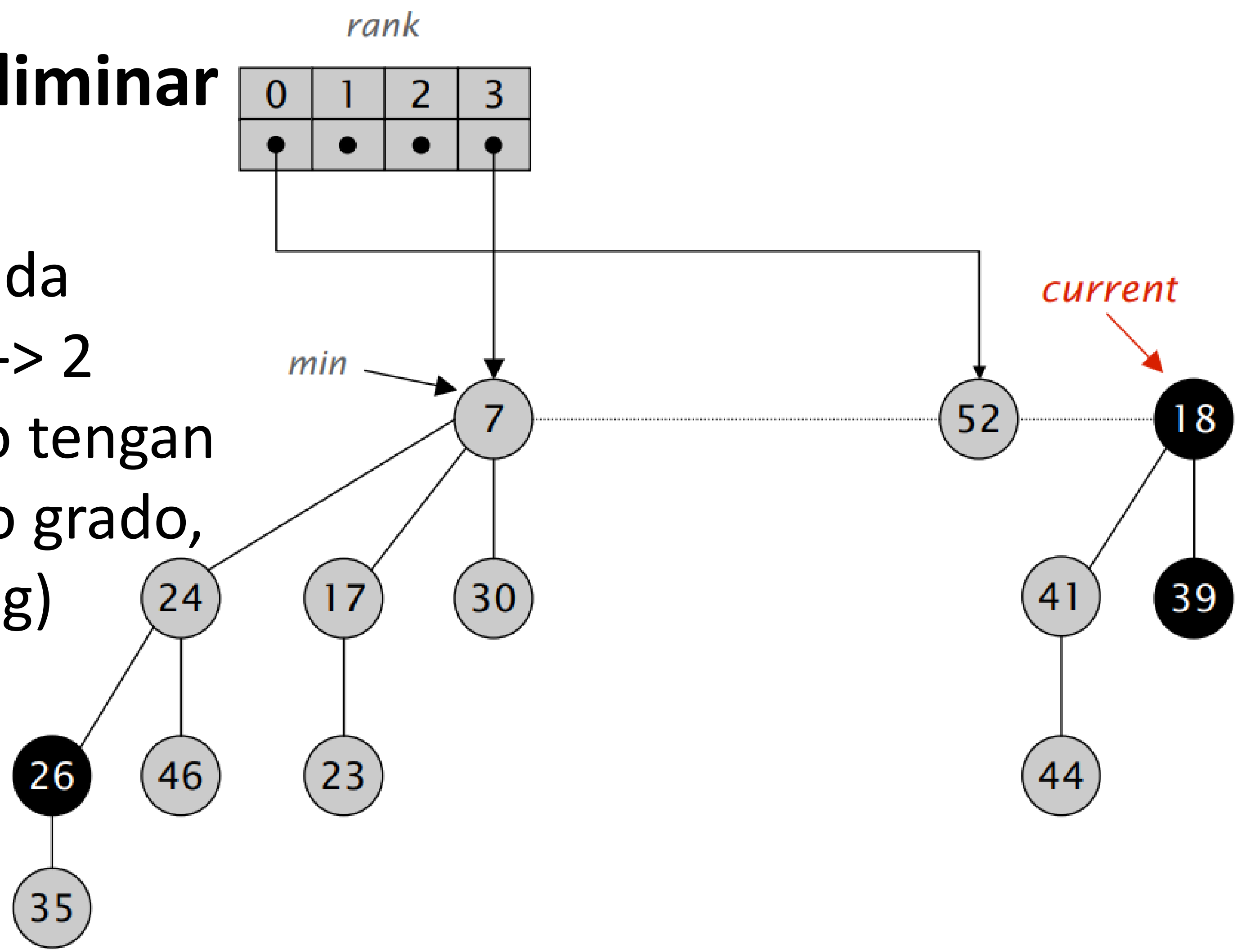
Operaciones

- **Extraer/Eliminar** mínimo:
  - (3) Consolida árboles--> 2 raíces no tengan el mismo grado, en O*(log)



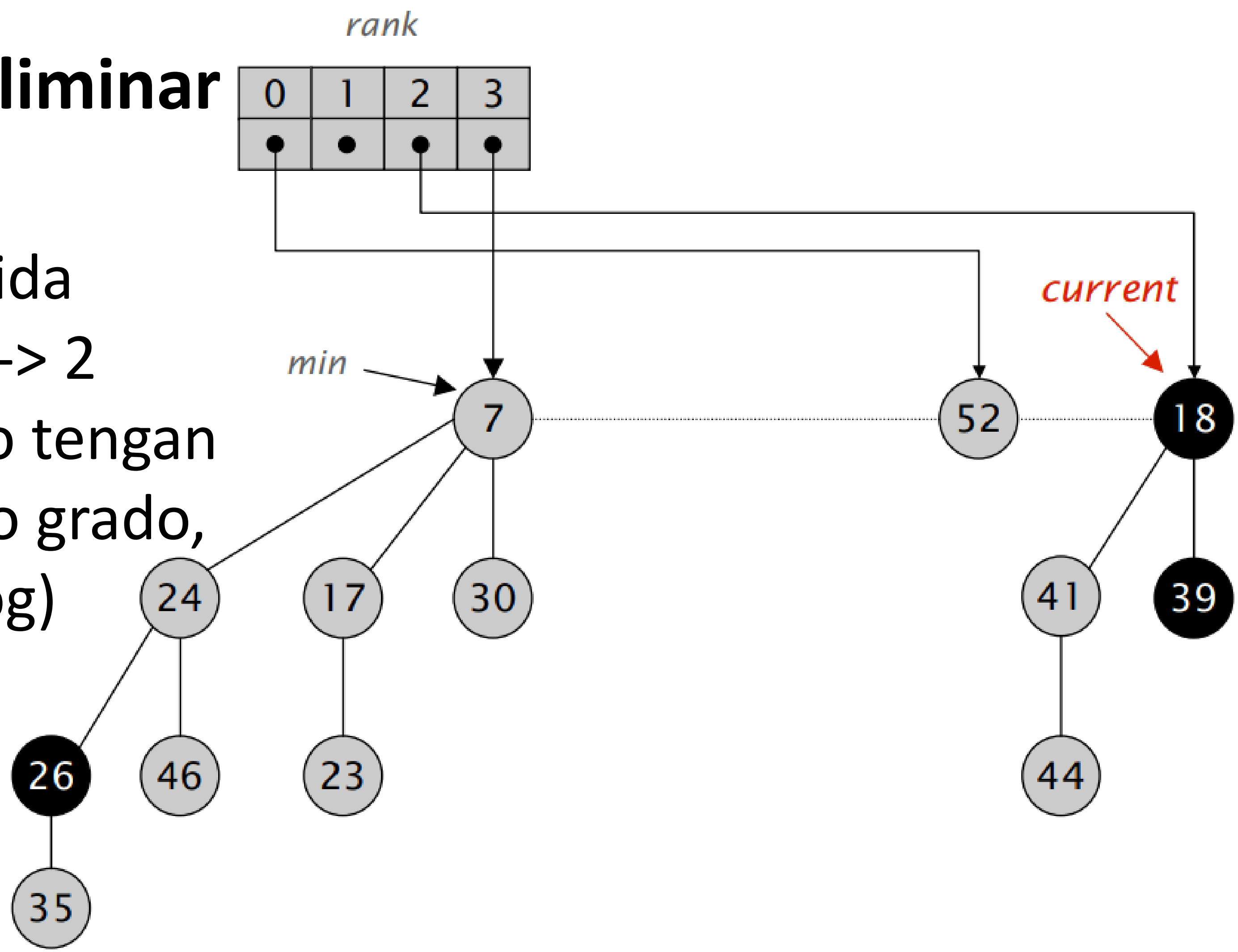
Operaciones

- **Extraer/Eliminar** mínimo:
  - (3) Consolida árboles--> 2 raíces no tengan el mismo grado, en O*(log)



Operaciones

- **Extraer/Eliminar** mínimo:
  (3) Consolida árboles--> 2 raíces no tengan el mismo grado, en O*(log)

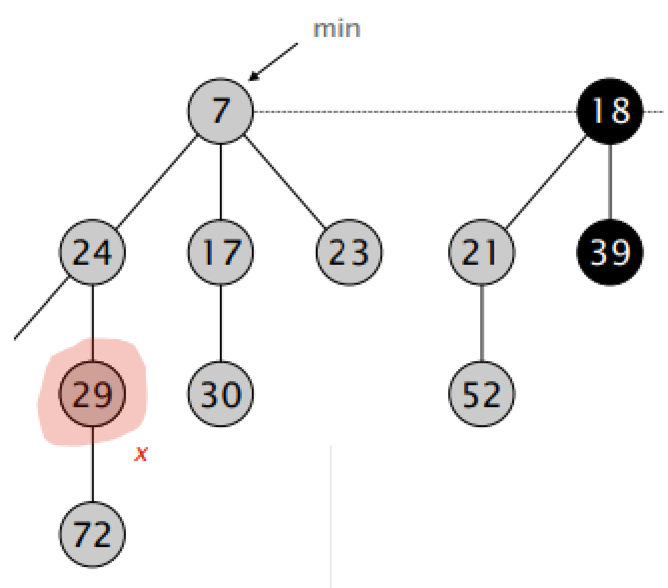


Operaciones

- **Decrementar clave x**:

  (1) Caso base: no viola la propiedad heap. Se podría actualizar "min" si es una raíz, O(1). Los nodos marcados son los que han perdido un hijo en otras operaciones.

Decrementamos x de 40 a 29



Operaciones

- **Decrementar clave x**:

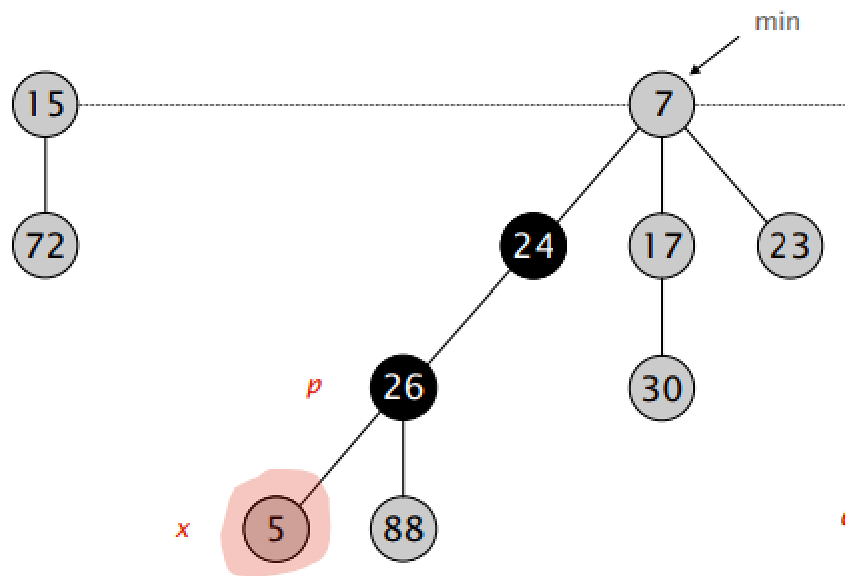
  (2) Caso que viola la propiedad heap. Se mueve el sub-árbol rooteado en x a la lista de raíces y se desmarca x si estaba marcado. Luego se chequea el padre de x. Si no estaba marcado, se marca y “fin”. Sino, se mueve y=padre(x) a la lista de raíces, y se desmarca. Esto se repite para y=padre(y), hasta llegar a la raíz (fin), o hasta llegar a un ancestro desmarcado (en cuyo caso, se marca por perder un hijo **si no es raíz**, y “fin”). Siempre se actualiza min.

Operaciones

- **Decrementar clave x**:

  (2) Caso que viola la propiedad heap.

•X se decrementa de 70 a 5.
•Su padre y abuelo ya venían marcados.
•Note que 5 no es menor que 26.

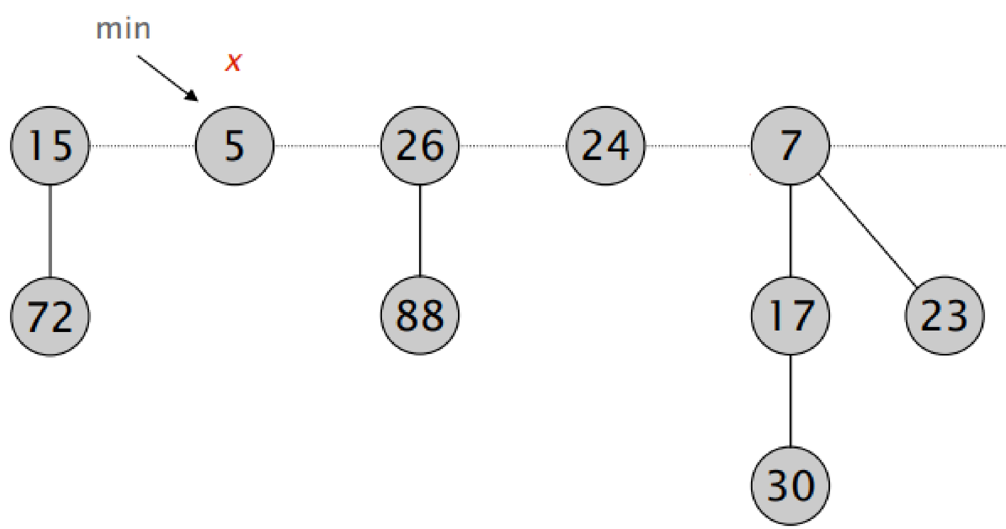


Operaciones

- **Decrementar clave x**:

  (2) Caso que viola la propiedad heap.

  - El 26 y 24 pasan a la lista de raíces, y se desmarcan.
  - La raíz (7) no se marca por ser raíz.
  - El mínimo se va actualizando.
  - Aunque sea O(log), al realizar n decrementos, en promedio queda en O*(1)



Operaciones

- **Eliminar x:** se decrementa la clave en –INF (llamada el método decrementar) y luego se elimina (llamada al método eliminar min), quedando la operación en O*(log)

Tarea

- Implementar los 3 tipos de heaps (minHeaps) vistos en clases: binary, binomial, Fibonacci.
- Hacer pruebas sobre las operaciones de insertar, eliminar mínimo, consultar mínimo y unión (mezclar) para los 3 heaps.
- Empiece primero llenando los heaps con N enteros aleatorios. Mida el tiempo de llenado de cada heap.
- Luego mida una secuencia de 100*N operaciones tomadas aleatoriamente de insertar, eliminar mínimo, consultar mínimo y unión para los 3 heaps y mida el tiempo por operación y compare.

Tarea

- Luego, genere y aplique 100*N operaciones seguidas de inserción, 100*N consultar mínimo, 100*N de eliminación de mínimo, y 100*N operaciones de unión, y compare el tiempo de cada tipo de operación en los 3 heaps.
- Qué concluye?
- Entregar la implementación de las 3 clases, y el programa principal que las usa. Hacer un informe de a lo sumo 3 páginas explicando los resultados.

Conclusiones

- Todas las operaciones son O(1) u O*(1) salvo eliminar min o eliminar que son O*(log).
- Tienen un mejor tiempo amortizado que los heaps binomiales.
- Se requiere de varios apuntadores por nodo, lo cual genera un costo adicional en memoria.
- Las operaciones no son tan triviales de implementar como las de un heap común, por lo que no son tan ampliamente utilizados.

Conclusiones

operación	*Heaps* binarios (peor caso)	*Heaps* binomiales (peor caso)	*Heaps* de Fibonacci (amortizado)
`crearHeap()`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`minimo()`	$\Theta(1)$	$O(\log n)$	$\Theta(1)$
`insertar(x)`	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
`eliminarMinimo()`	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
`mezclar(H1, H2)`	$\Theta(n)$	$O(\log n)$	$\Theta(1)$
`disminuirClave(x,k)`	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
`eliminar(x)`	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$

Referencias

- T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, “Fibonacci Heaps” in *Introduction to Algorithms*, 2nd ed., Cambridge, MA: The MIT Press, 2001, ch. 20, pp.476-497