5 Métodos de Ordenamiento

5.1 Métodos de Ordenamiento Internos

- 5.1.1 Burbuja
- 5.1.2 Quicksort
- 5.1.3 Heapsort
- 5.1.4 Inserción Simple
- 5.1.5 Shellsort

5 Métodos de Ordenamiento

Métodos de Ordenamiento

- Internos.- El espacio extra que se requiere para realizar el ordenamiento, es tan pequeño que se considera nulo.
- Externos.- Se requiere una cantidad de espacio extra proporcional al número de datos a ordenar.

5.1 Ordenamiento Interno

Conceptos Básicos

- **Archivo.** En este curso nos referiremos no solo al conjunto almacenado en memoria secundaria, sino incluso a cualquier conjunto de datos en memoria primaria.
- **Registro.** Cada uno de los elementos que conforman un archivo.
- **Llave.** Conjunto de campos o atributos que determina el orden del archivo.

5.1 Ordenamiento Interno

Convenciones

Para simplificar el análisis de los diversos métodos que estudiaremos

- Los archivos de ejemplo se compondrán únicamente de llaves aunque los registros de las aplicaciones prácticas casi siempre contienen otros datos aparte de la llave.
- Las llaves aparecerán como valores numéricos.
- Las llaves solo se compondrán de un dato (en los casos prácticos puede ser cualquier número de datos).
- Se describirán los métodos de ordenamiento asumiendo que se desea organizar el archivo en forma ascendente (para hacerlo a la inversa los cambios son pequeños).
- En los casos prácticos los datos que contiene cada registro (aparte de la llave) son tan importantes como la llave misma y debemos tener en cuenta que cuando hablemos de **mover la llave** de un registro a otro lugar del archivo, realmente nos referiremos a **mover el registro completo**.

5.1.1 Método de la Burbuja

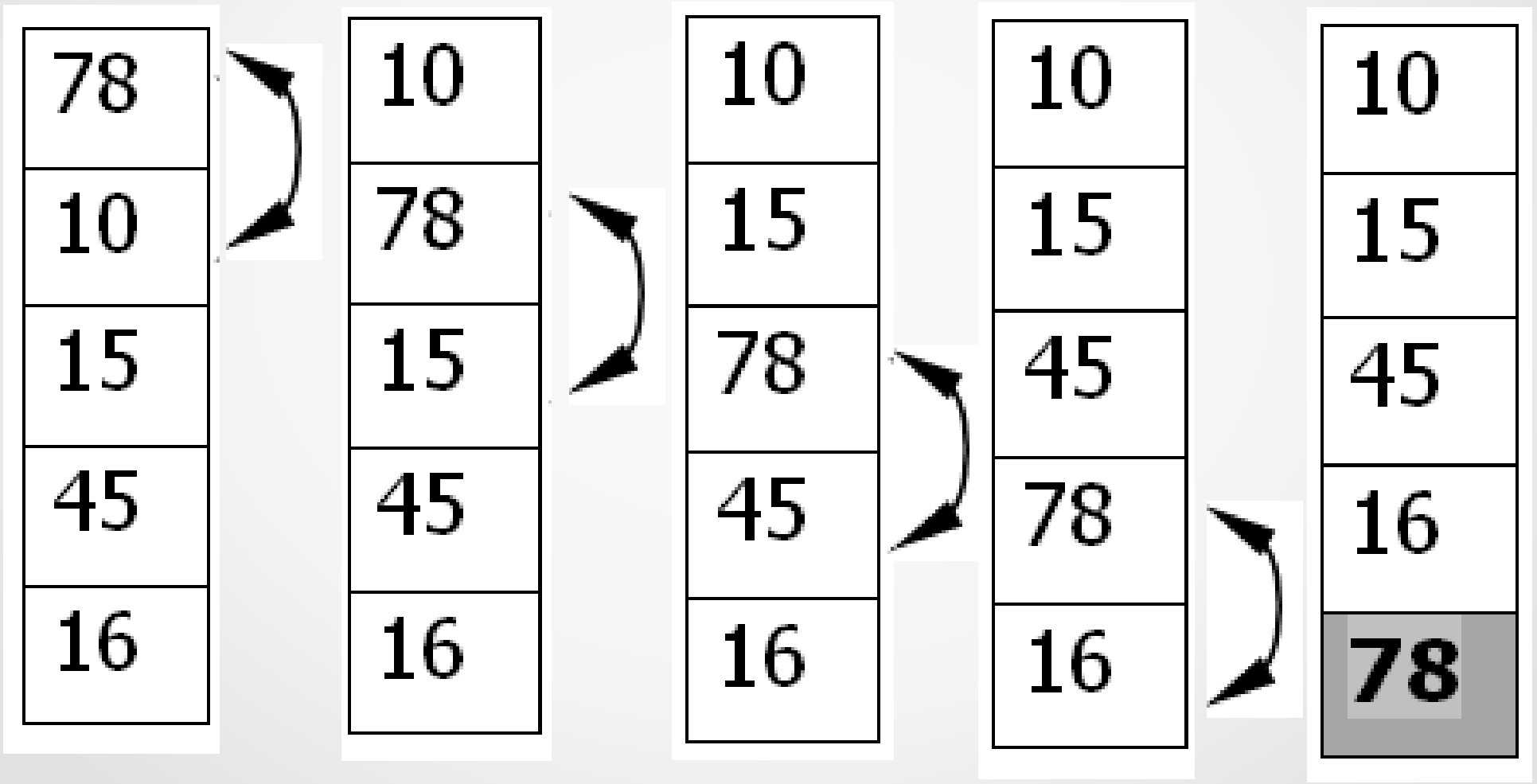
A. Logra el objetivo realizando varios **"pasos"** al archivo comparando pares de llaves en posiciones adyacentes.
B. Si la **comparación** determina que están fuera de orden, **intercambiarán sus lugares**.
C. Se inicia comparando la **1ª y 2ª llaves**.
D. Luego se compararán la **2ª y 3ª llaves** (la segunda puede ser la que antes era la primera).
E. Y así sucesivamente hasta comparar **la penúltima** con la **última** llaves.

- Los pasos anteriores logran que el **valor más grande** sea precipitado al fondo del archivo.
- Si en esta etapa, **no se realizan intercambios**, significa que el archivo está ordenado y el proceso debe terminar.
- En caso de que se presente **al menos un intercambio**, el razonamiento obliga a repetir los pasos arriba descritos.
  - Iniciando con las **llaves 1 y 2**.
  - Finalizando con la penúltima y última llaves (considerando que el archivo **lógico contiene una llave menos**).

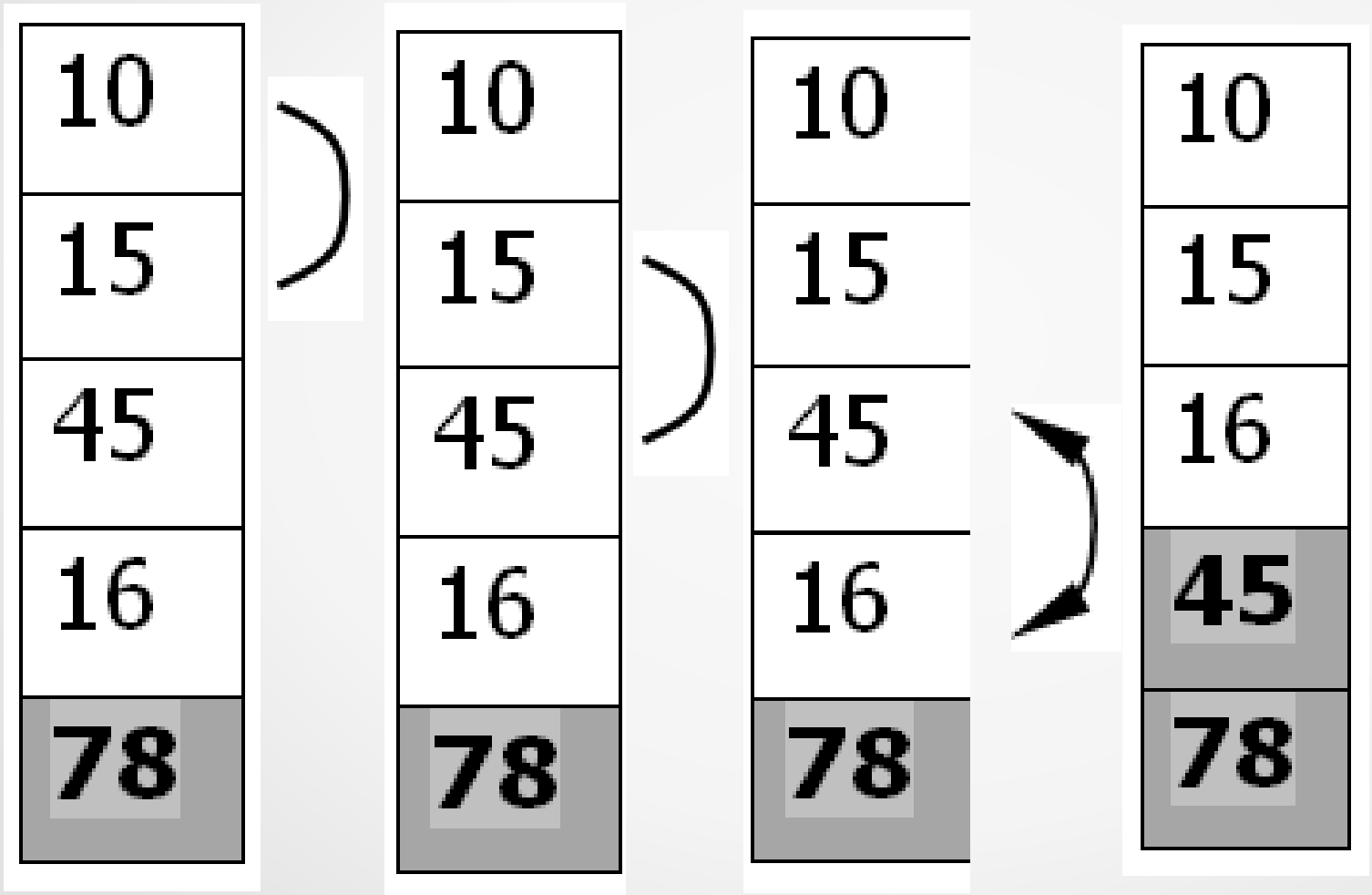
Comparación

Comparación e intercambio

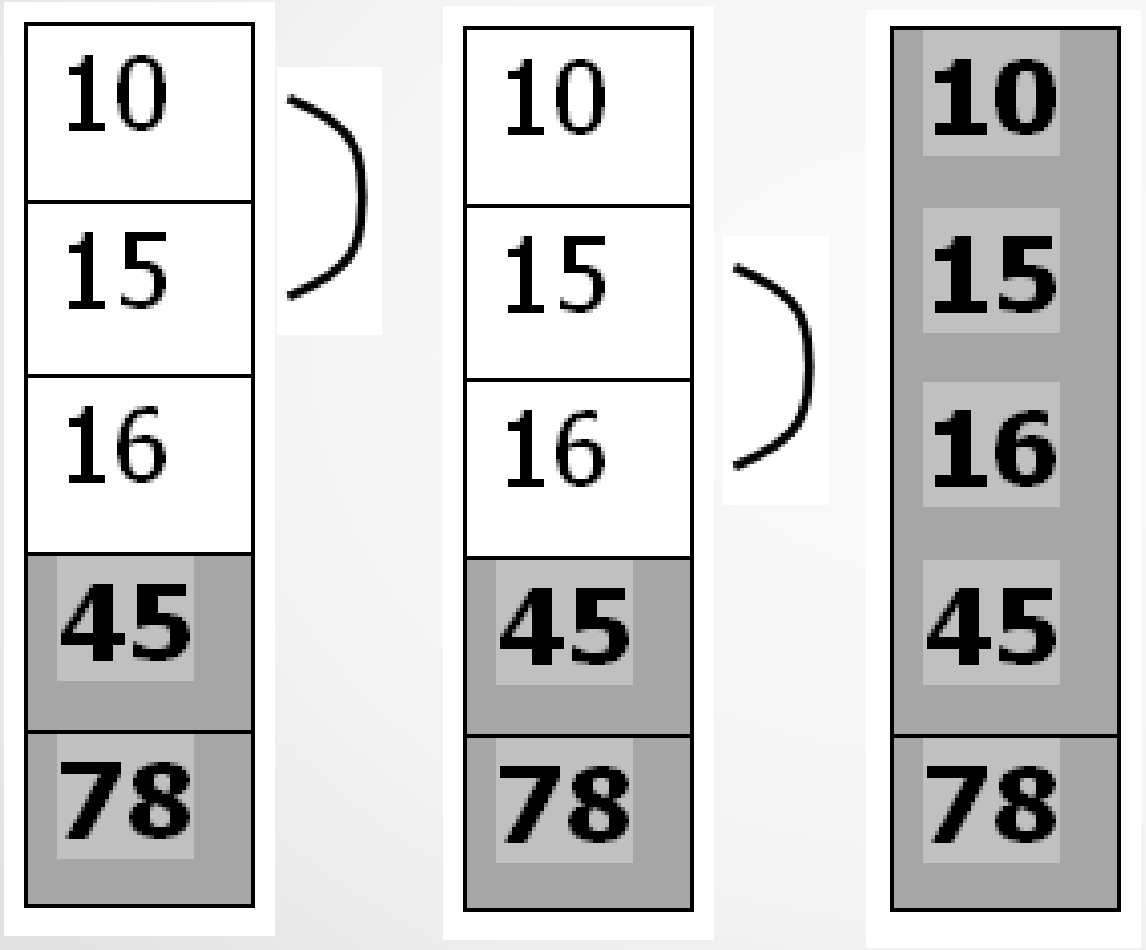
REPASO 1



REPASO 2



REPASO 3



5.1.2 Quick Sort

Los problemas de ordenamiento solucionados mediante algoritmos de orden de complejidad **$O(n^2)$**, como burbuja, para valores de **N** muy grandes, tienden a ser intratables, por lo tanto, hay que buscar otras alternativas.

Tamaño del archivo (n)	Número de comparaciones
100	4,872
500	124,740
1000	499,175
5000	12,483,304
10000	49,988,784
20000	199,971,855

5.1.2 Quick Sort

Si escribiéramos un programa que:

1) Divida los archivos grandes en cierta cantidad de *subarchivos* más pequeños.

2)Ordene los subarchivos en forma independiente.

3)Finalmente los reúna de nuevo para obtener el archivo ya ordenado.

4)Creemos que se realizara un menor esfuerzo para ordenar el archivo

5.1.2 Quick Sort

En base a la tabla anteriormente mostrada:

n = **10,000** comparaciones = **49,988,784**

100 subarchivos de 100 registros c/u = 10,000 registros

n = **100** comparaciones = **4,872**

100 subarchivos x 4,872 comparaciones = **487,200**

Se realiza el mismo trabajo con el 1% del esfuerzo.

5.1.2 Quick Sort

Quicksort aplica esta técnica (conocida comúnmente como "Divide y Vencerás"). ¿Cómo lo hace?

Observar un archivo de llaves diferentes ya **ordenado**.

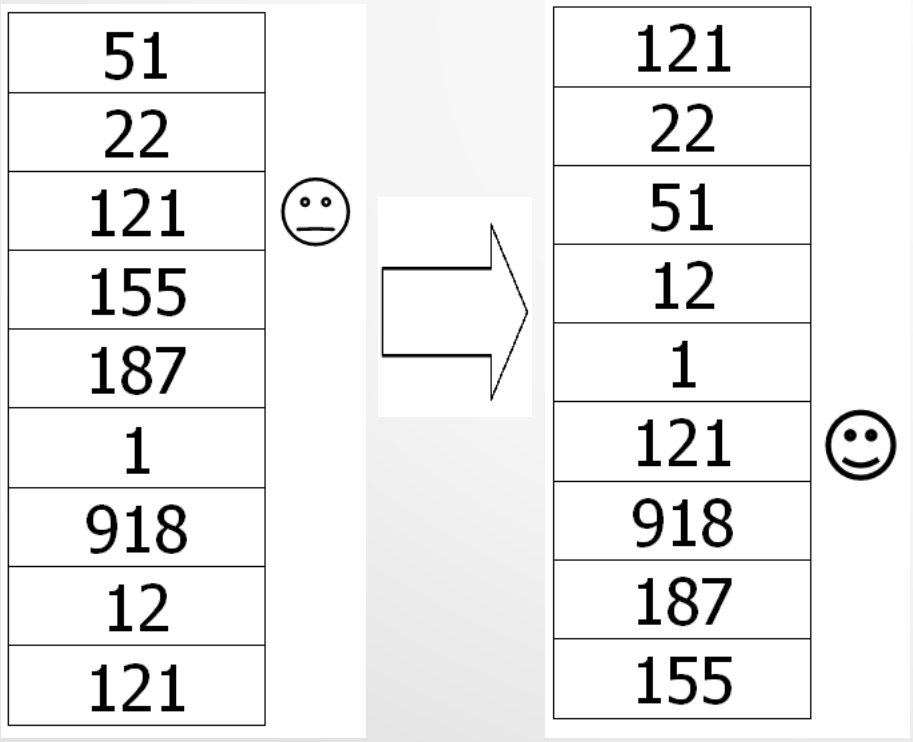
15
21
36
75
99
100
198
210
211

Elijamos una llave cualquiera.

Observemos que todas las llaves menores que la que tomamos están **arriba** y todas las mayores **abajo**.

5.1.2 Quick Sort

Por lo tanto, si tomamos ahora, una **llave cualquiera** de un archivo desordenado y logramos poner esa llave **en una posición tal** que todas las llaves menores o iguales a ella queden arriba y las mayores abajo, **sin importar el orden**, esa llave ya estará en la posición correcta.



El problema ahora es ordenar 2 **subarchivos** más pequeños, uno **arriba** de la llave que quedó en su lugar y otro **abajo**.

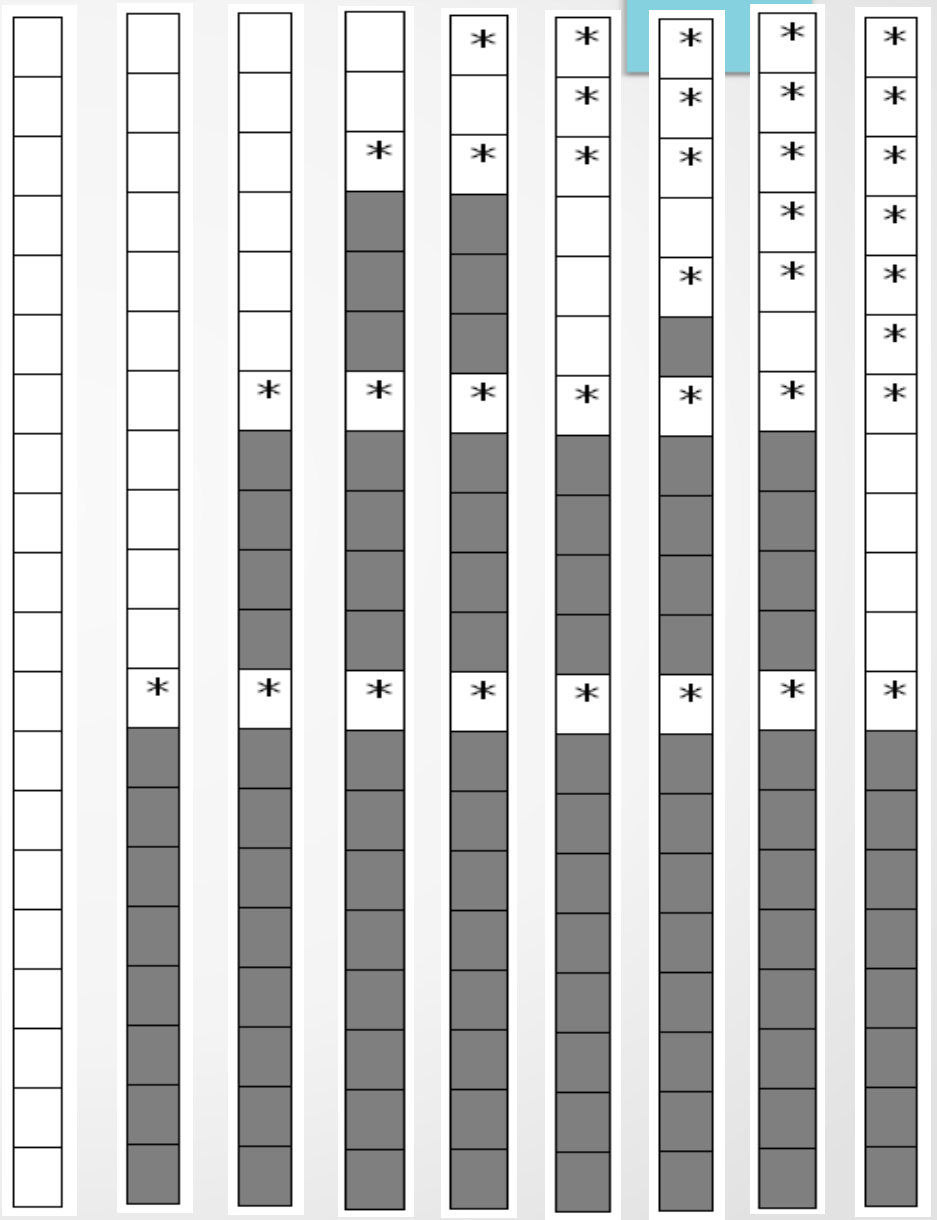
Primero se ordena uno y el otro se deja pendiente.

5.1.2 Quick Sort

- Cada vez va creciendo el número de subarchivos por ordenar, pero los subarchivos van siendo cada vez más pequeños hasta llegar a tener solo un registro y, por lo tanto, ya estar ordenados.
- En cierto momento se llega a una situación en la que ya no hay más subarchivos por ordenar y el archivo completo estará ordenado.
- En la diapositiva siguiente se muestra como avanza el proceso.

5.1.2 Quick Sort

- Cada segmento en blanco es la parte del archivo para la cual se va a colocar en su lugar una llave.
- Los segmentos en gris indican los subarchivos pendientes.
- Los asteriscos muestran registros que ya se colocaron en su lugar.



5.1.2 Quick Sort

- **Quicksort** se reduce a un procedimiento básico que coloca en su lugar una llave.
- La ejecución repetitiva de ese procedimiento ocasionará el ordenamiento del archivo completo.

Algoritmo para colocar en su lugar un registro de un subarchivo.

archivo = nombre del archivo.
k, m = límites inferior y superior de un subarchivo cualquiera.
i, j = variables índice para encontrar el lugar.
x = llave del subarchivo elegida por el algoritmo, por simplicidad se elige la primera: **archivo(k)**

5.1.2 Quick Sort

1. La llave **archivo(k)** almacenada en **x** deja un **"hueco"**.

2. Las variables **i** y **j** iniciarán en los límites del subarchivo (**i** solo se incrementará, **j** solo se decrementará).

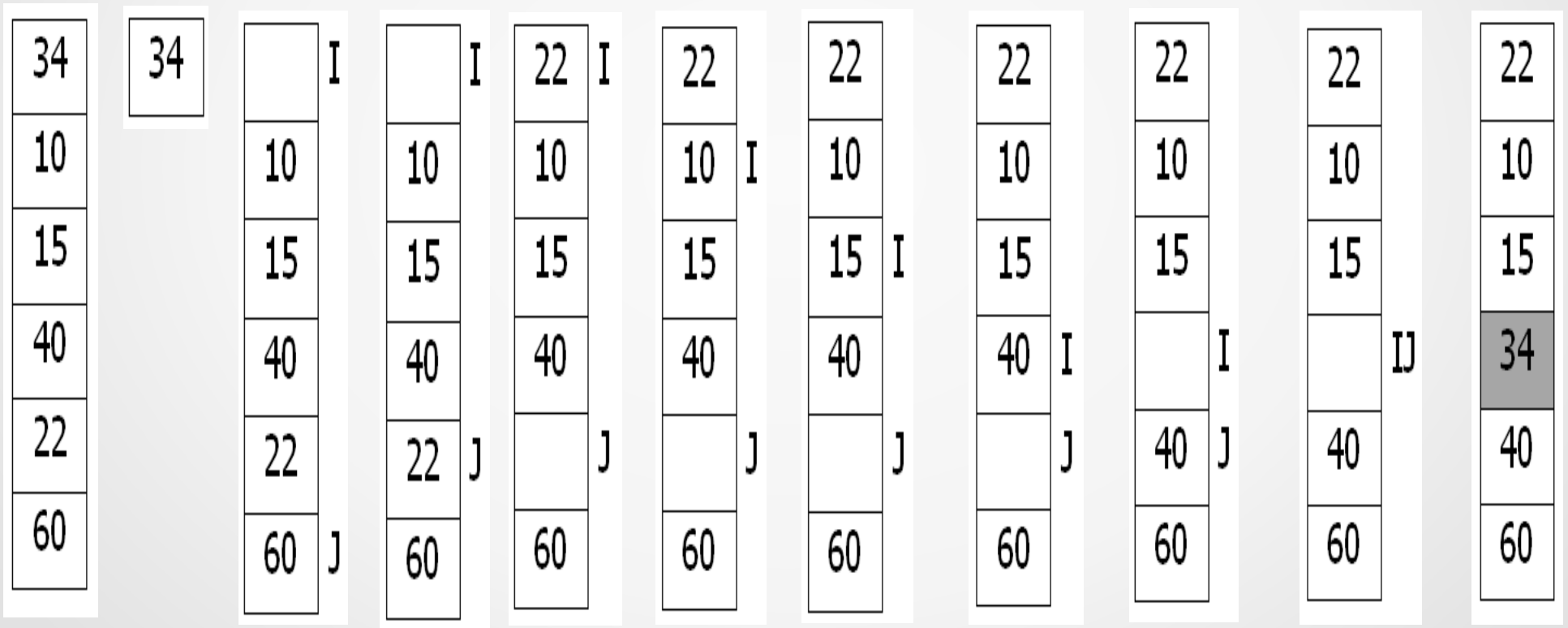
3. Los huecos de i deben llenarse con un valor **menor o igual que x** que encuentre **j** al decrementarse.

4. Los huecos de j se llenarán con un valor **mayor que x** que encuentre **i** mientras se incrementa.

5. Cuando **i y j se encuentren**, ese es el lugar que debe ocupar el valor guardado en x.

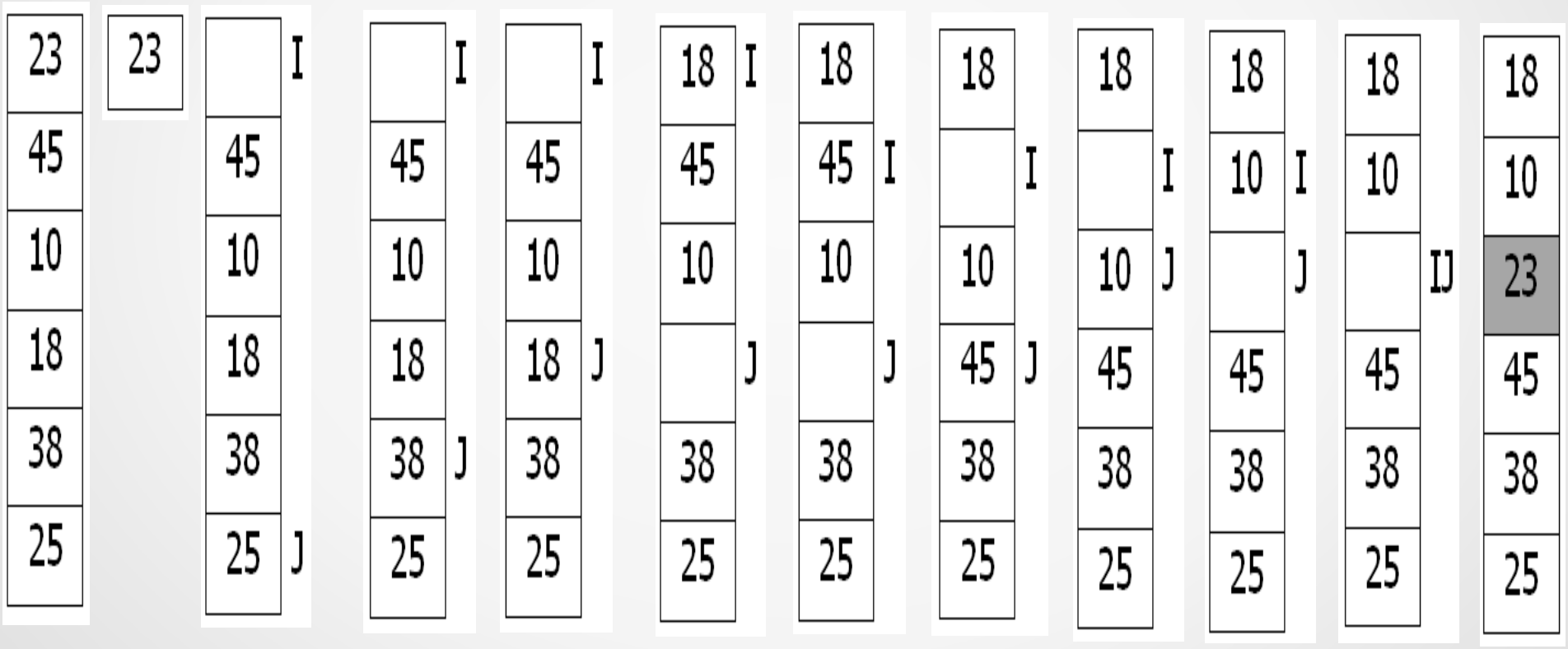
5.1.2 Quick Sort

Ejemplo de ejecución del Algoritmo



5.1.2 Quick Sort

Otro ejemplo



5.1.2 Quick Sort

Quicksort es un algortimo cuyo orden de complejidad temporal es O(n logn).
La tabla siguiente lo demuestra.

n	Movimientos	n*logn	Movimientos/ (n*logn)
15,000	85,285	62,641.37	1.3615
82,000	565,231	402,932.74	1.4028
100,000	698,099	500,000.00	1.3962
150,000	1,106,557	776,413.69	1.4252
300,000	2,306,304	1,643,136.38	1.4036
500,000	3,878,252	2,849,485.00	1.3610

Quicksort recursivo

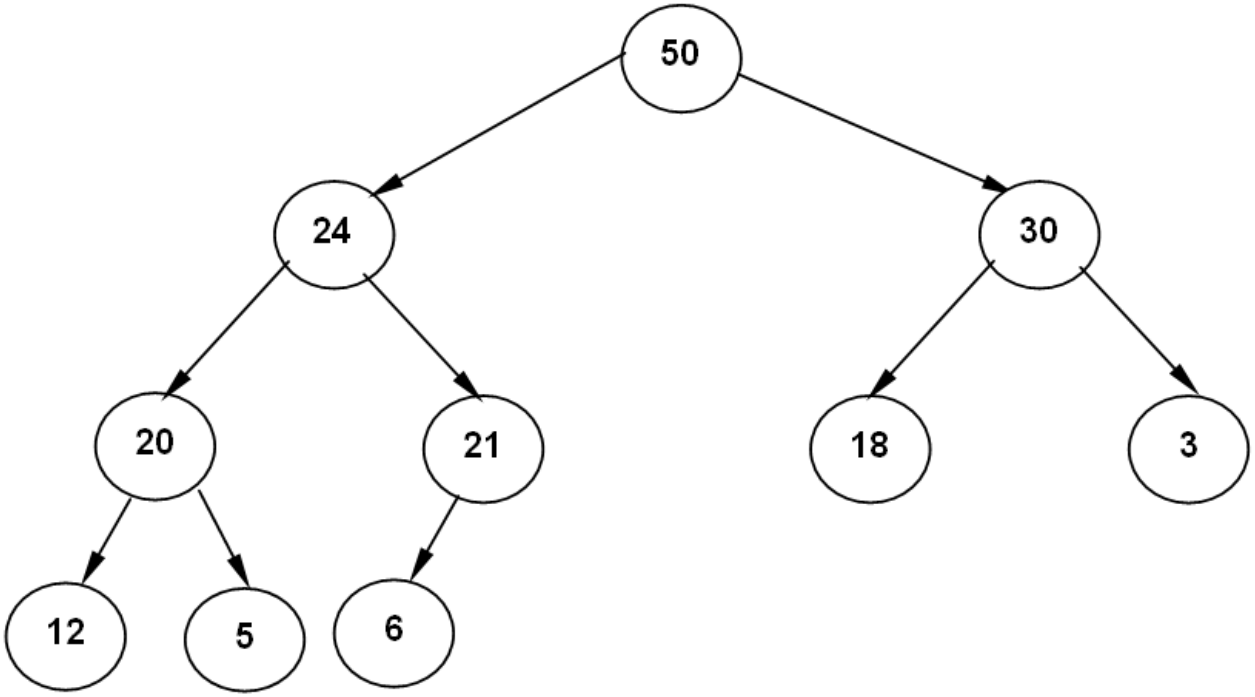
```
void IntArrayQuickSort (int[] data, int l, int r){
   int i, j;
   int x;
   i = l; j = r;
   x = data [(l + r) / 2]; // elemento de la mitad
   while (true) {
      while (data[i] < x)
        i++;
      while (x < data[j])
        j--;
      if (i <= j) {
          exchange (data, i, j);
          i++;
          j--;}
      if (i > j)
          break;
    }
    if (l < j) IntArrayQuickSort (data, l, j);
    if (i < r) IntArrayQuickSort (data, i, r);      }
```

5.1.3 Heap Sort (Ordenamiento por Montón)

Supongamos que se cuenta con un archivo como este:

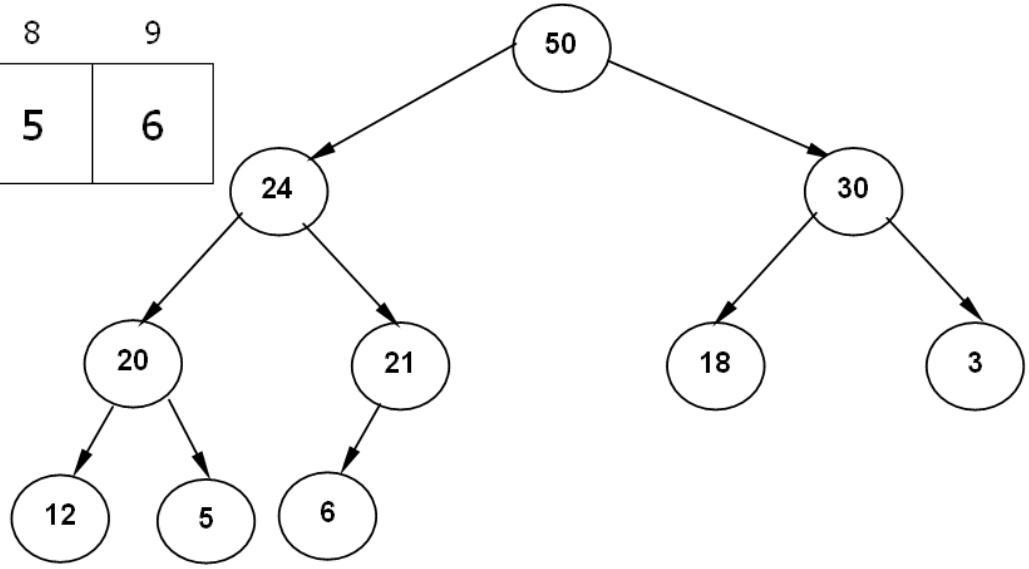
0	1	2	3	4	5	6	7	8	9
50	24	30	20	21	18	3	12	5	6

Se puede interpretar como si fuera un árbol binario:



- La dirección del hijo izquierdo de un nodo de la dirección **i** es **2i+1**.
- La dirección del hijo derecho de un nodo de la dirección **i** es **2i+2**.

0	1	2	3	4	5	6	7	8	9
50	24	30	20	21	18	3	12	5	6



Si un árbol binario es un **Montón** cumplirá con las siguientes condiciones:

1. Si el número total de nodos en el árbol es **impar**, todos los nodos internos tienen dos hijos.
2. Si el número total de nodos en el árbol es **par**, el último nodo interno solo tendrá un hijo (el nodo interno más extremo hacia la derecha, del penúltimo nivel).
3. Todas las **hojas** están en el **último nivel** y en caso de que las haya en el penúltimo, estarán **a la derecha de los nodos internos**.
4. La llave de cada nodo es **mayor o igual** que las llaves de sus hijos.

hoja = nodos sin hijos.
nodo interno = nodo con hijos.

5.1.3 Heap Sort (Ordenamiento por Montón)

Eliminación de nodos del montón

- El archivo del ejemplo es un Montón.
- La construcción de un Montón a partir de un archivo cualquiera la estudiaremos un poco después.
- La eliminación de nodos de un montón ocasiona el ordenamiento del archivo.

5.1.3 Heap Sort (Ordenamiento por Montón)

Eliminar un nodo de un Montón consiste en:

- Retirar la llave de la raíz (la mayor del montón).
- Reubicar el resto de las llaves de manera que la estructura siga siendo un montón.
- Para hacerlo, se debe eliminar un nodo.
- El nodo elegido es el único que se permite para que la estructura siga siendo montón: la última hoja (extrema derecha del último nivel).
- El único lugar disponible para colocar la llave del nodo eliminado es el lugar vacante dejado por la raíz.
- Una vez ahí, la llave debe ir bajando hasta el lugar en que su valor sea mayor que el de sus hijos.

5.1.3 Heap Sort (Ordenamiento por Montón)

El algoritmo para bajar la llave que pertenecía a la última hoja (llamémosla K) es el siguiente:

1. Si K es mayor o igual que el mayor de sus "supuestos hijos" entonces es el lugar adecuado y el proceso termina.

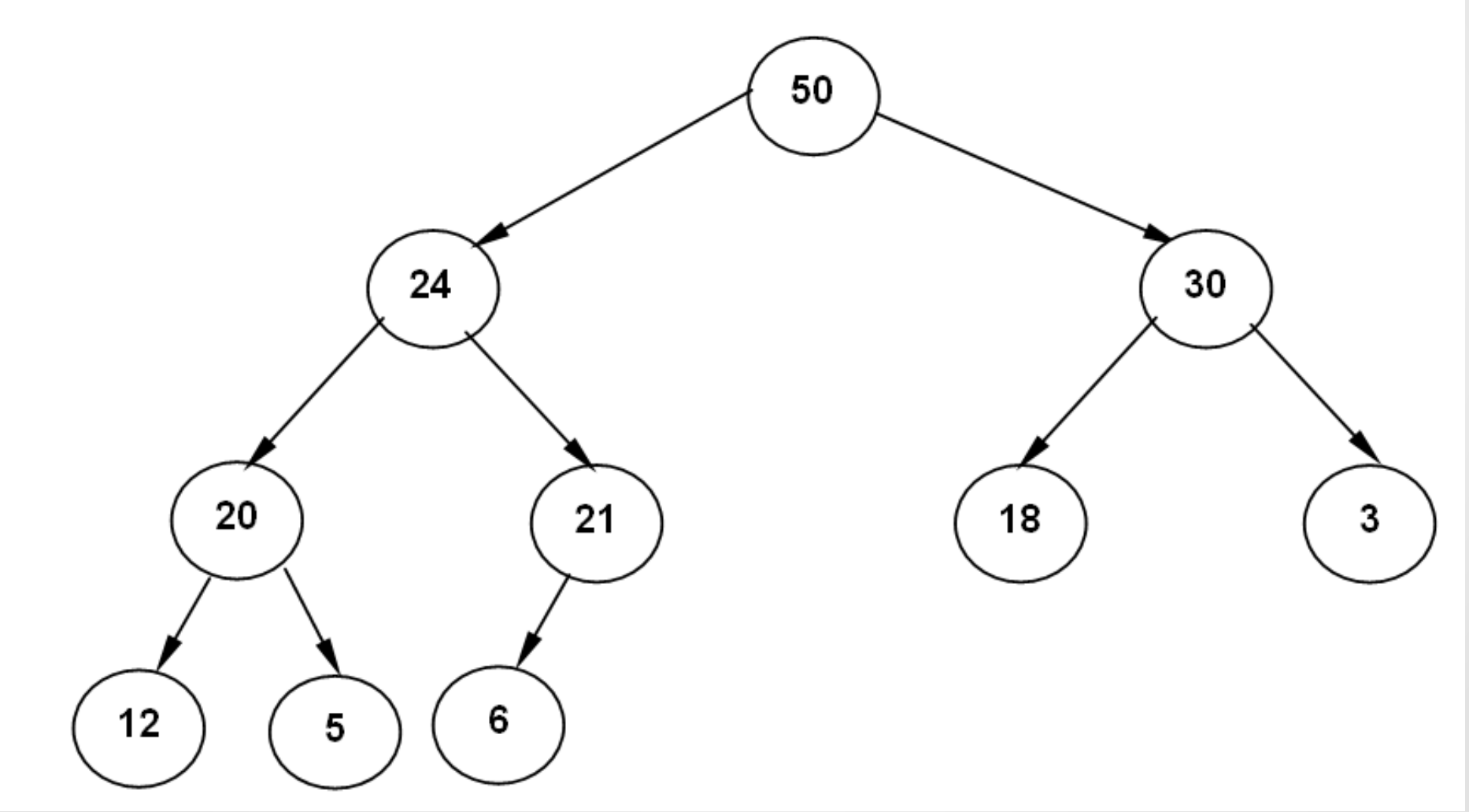
2. De lo contrario, intercambia su lugar con el mayor de sus supuestos hijos.

3. Se repite el paso 1.

Este algoritmo requiere que existan al menos dos nodos en el árbol.

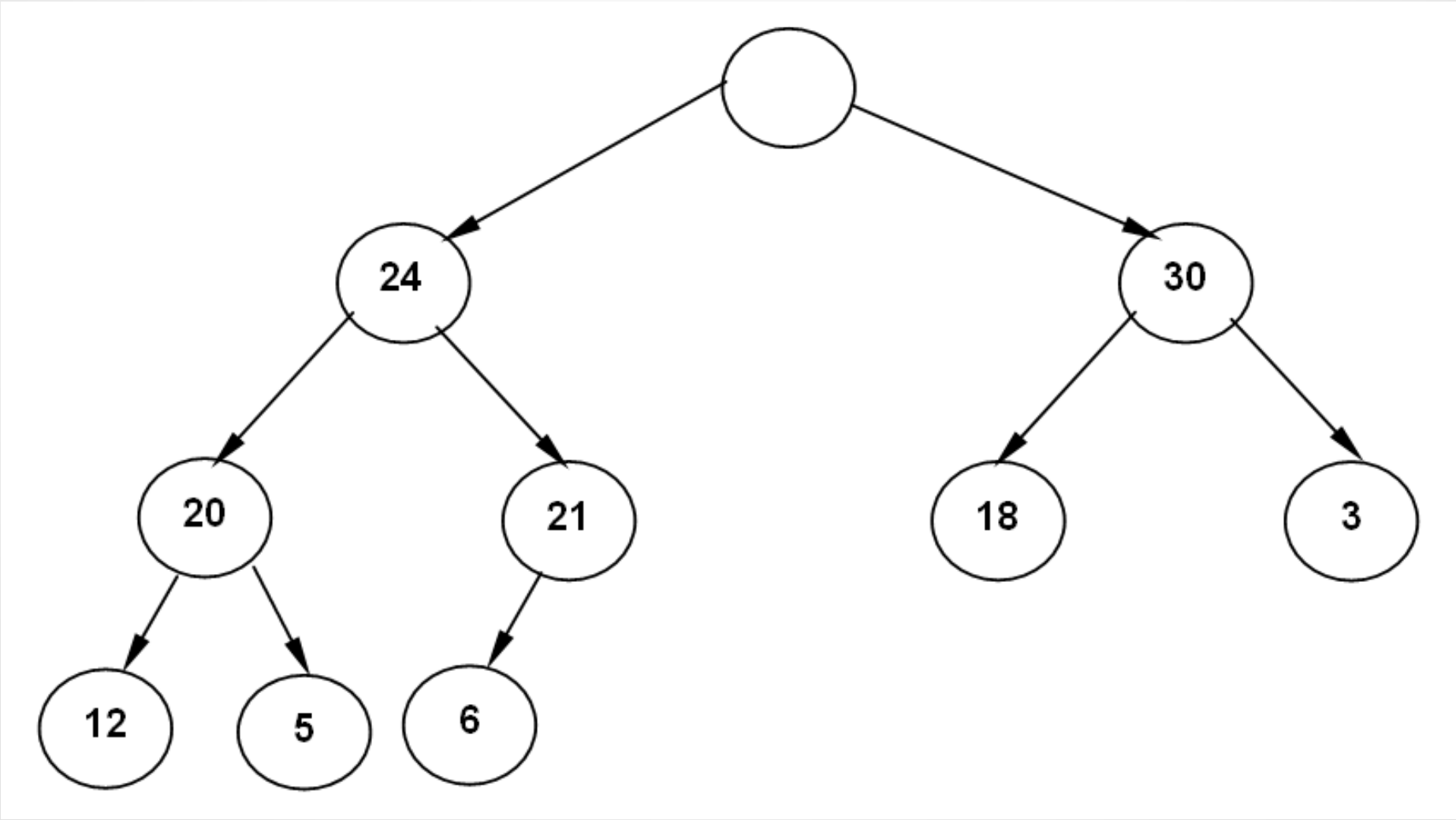
5.1.3 Heap Sort (Ordenamiento por Montón)

Se removerá un nodo del ejemplo



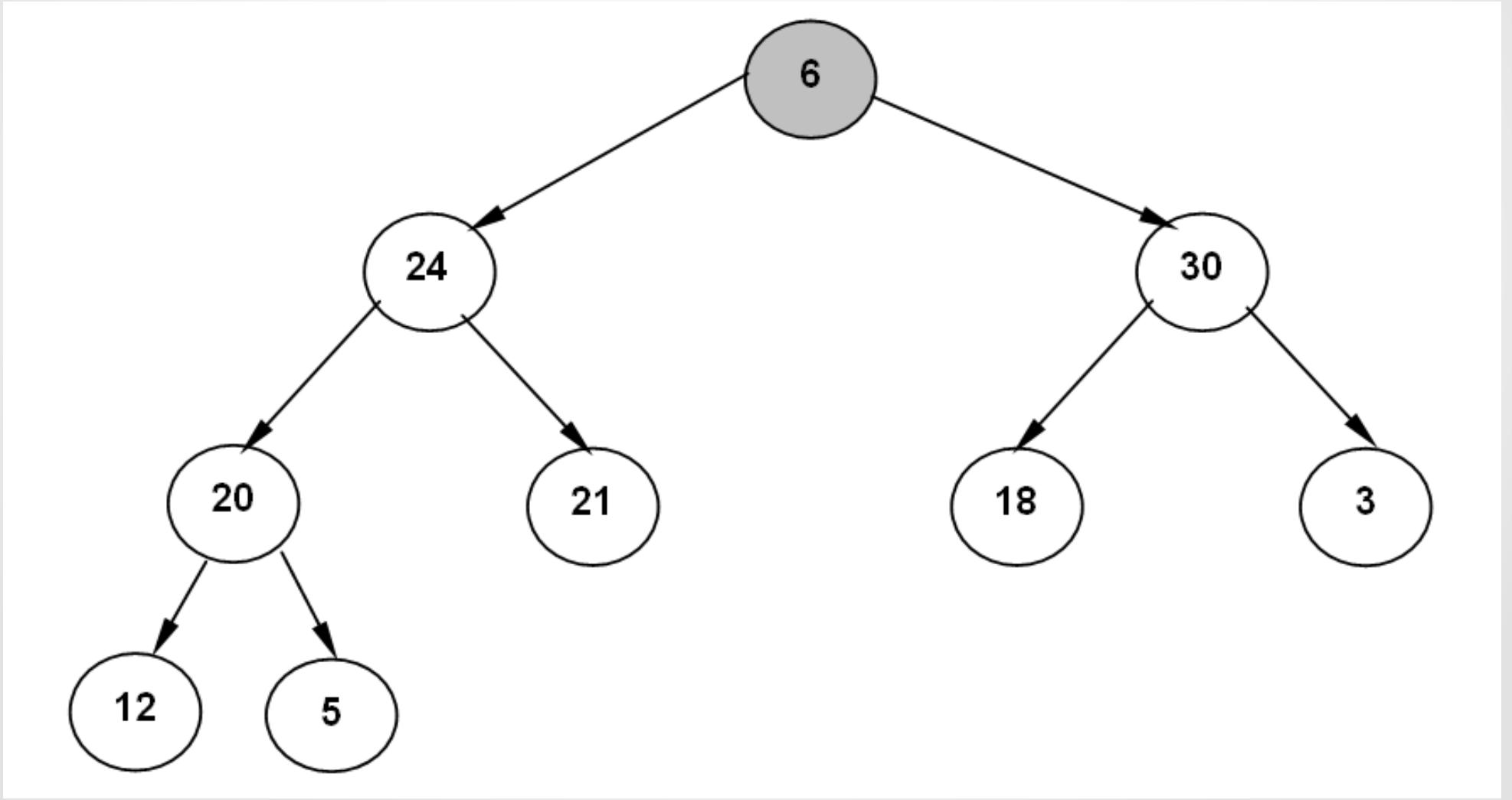
5.1.3 Heap Sort (Ordenamiento por Montón)

Sale la llave de la raíz



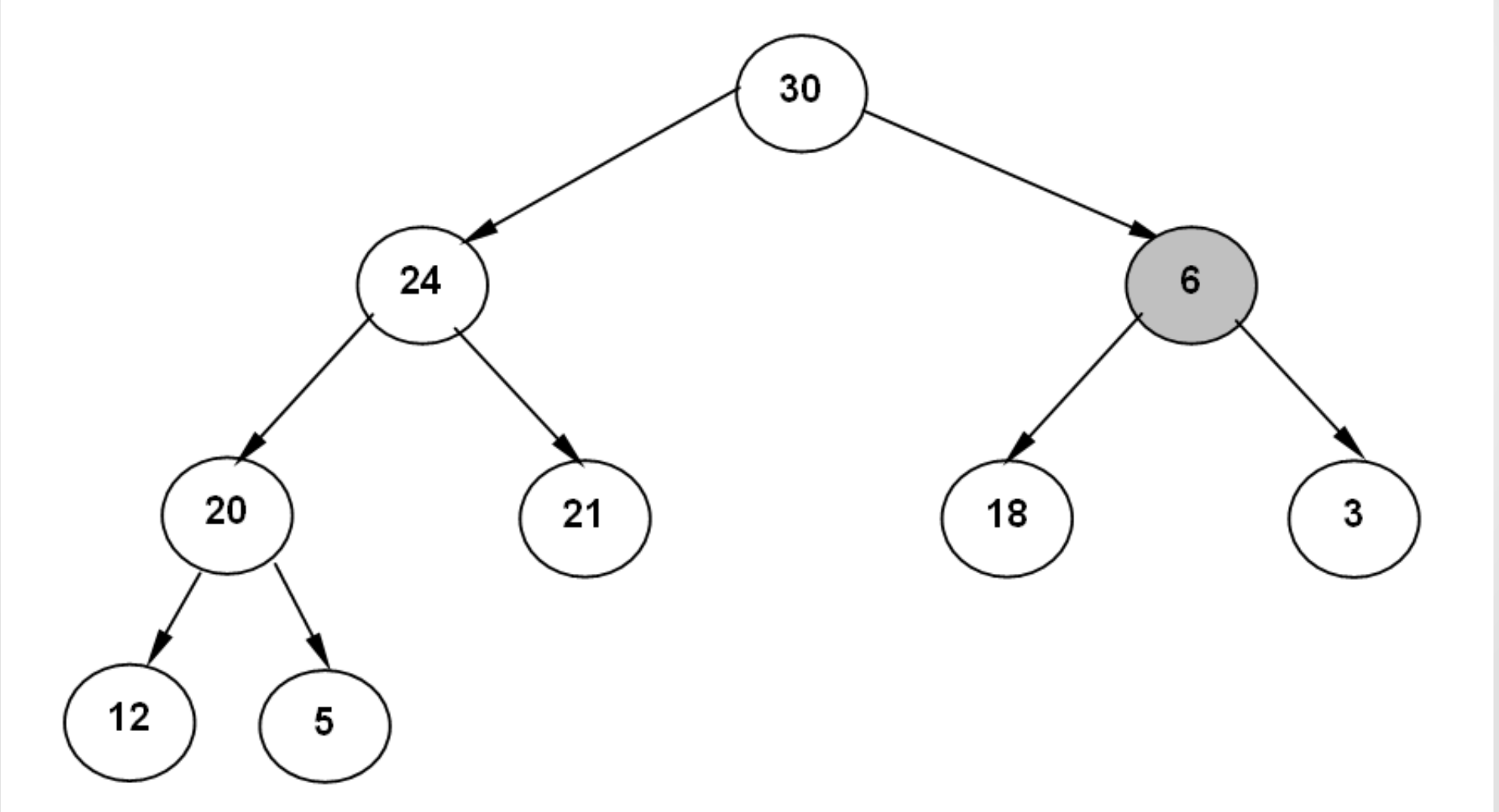
5.1.3 Heap Sort (Ordenamiento por Montón)

Se intenta llenar el nodo "vacante" con la llave de la última hoja



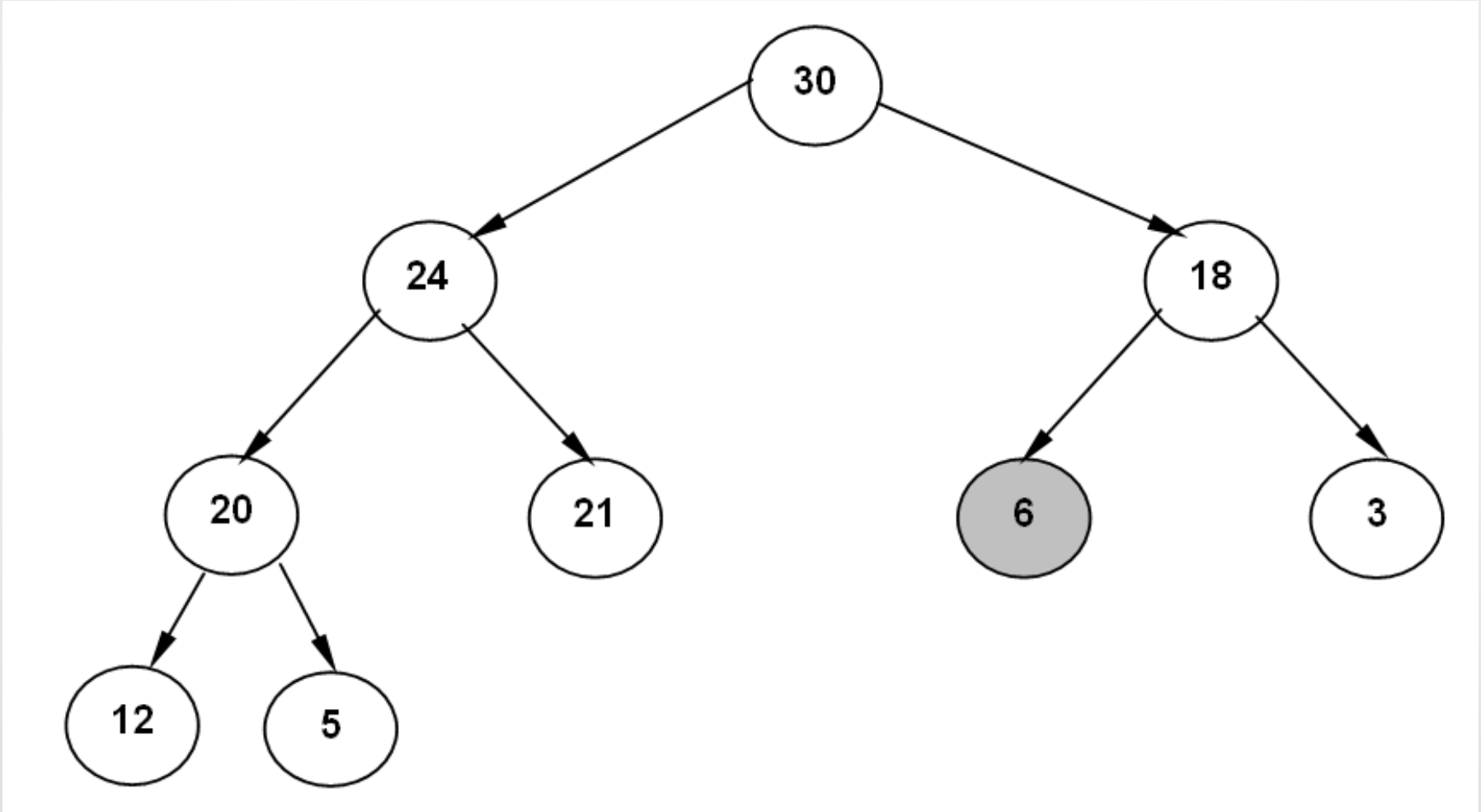
5.1.3 Heap Sort (Ordenamiento por Montón)

Como no cumple con la condición básica de "llave mayor a sus hijos", se intercambia con el mayor de ellos



5.1.3 Heap Sort (Ordenamiento por Montón)

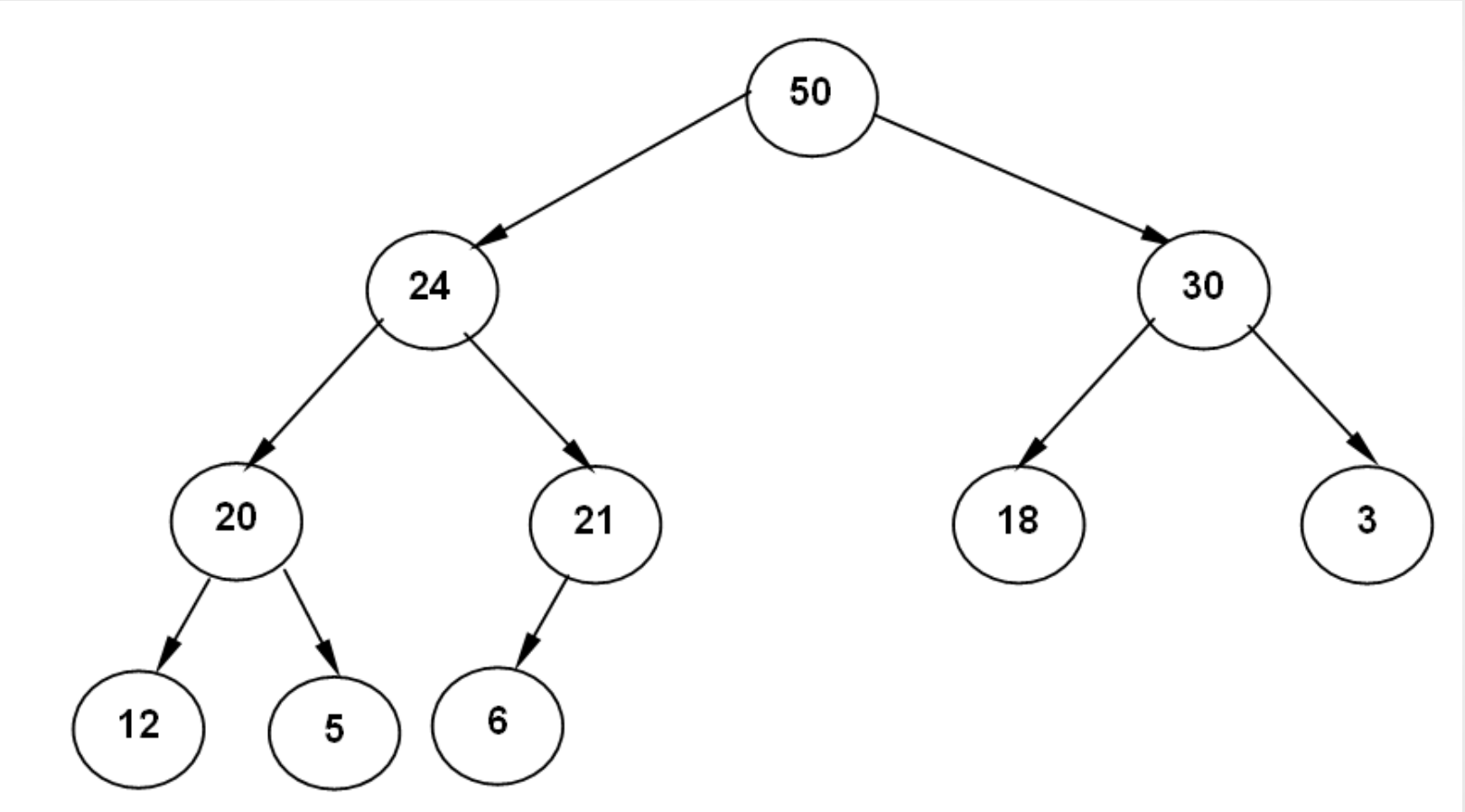
Sigue sin cumplir con la condición de "llave mayor a sus hijos" por lo que se intercambia con el mayor de ellos



La llave, al llegar a una hoja, cumple con la condición

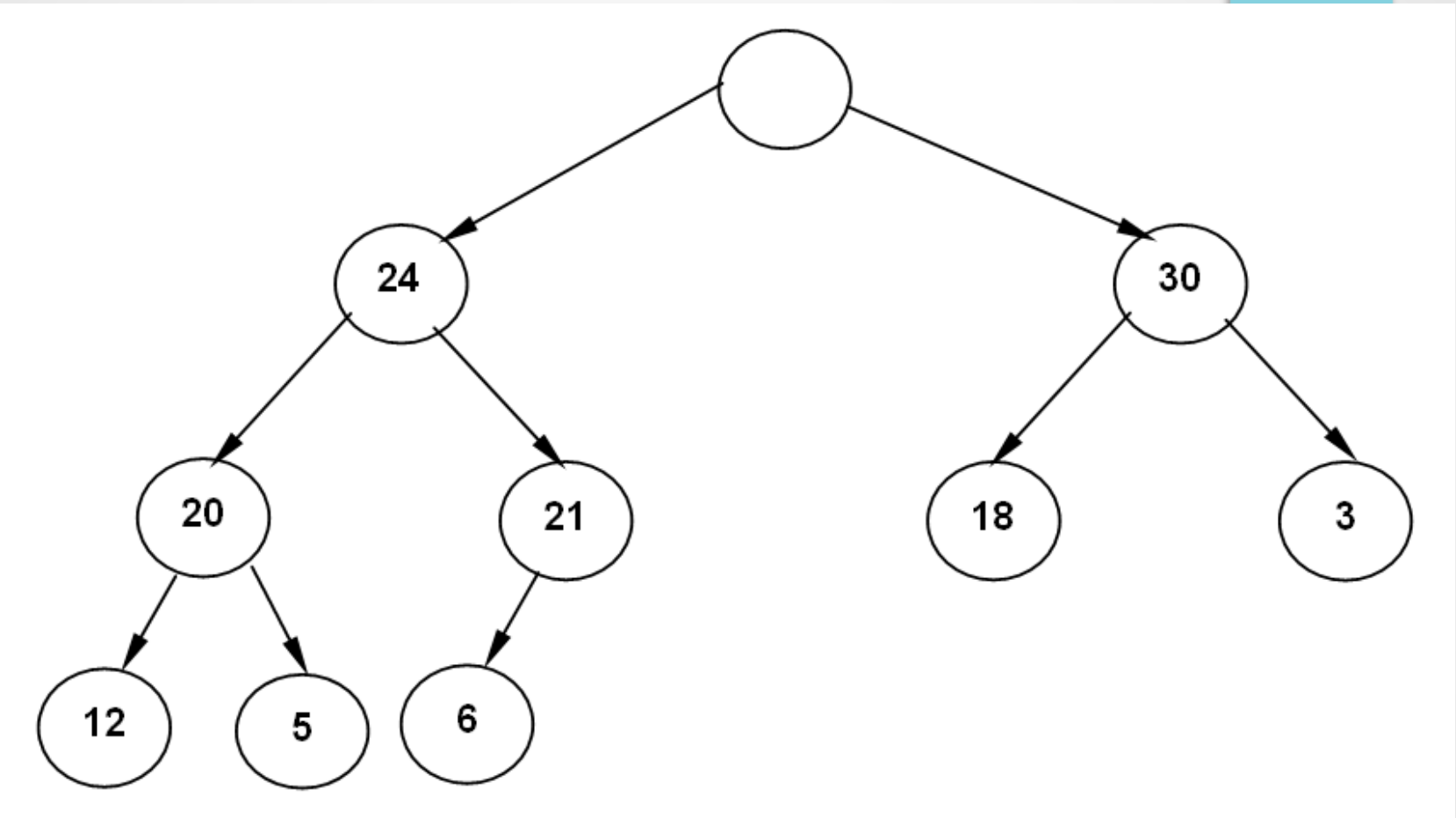
5.1.3 Heap Sort (Ordenamiento por Montón)

Repasemos el proceso



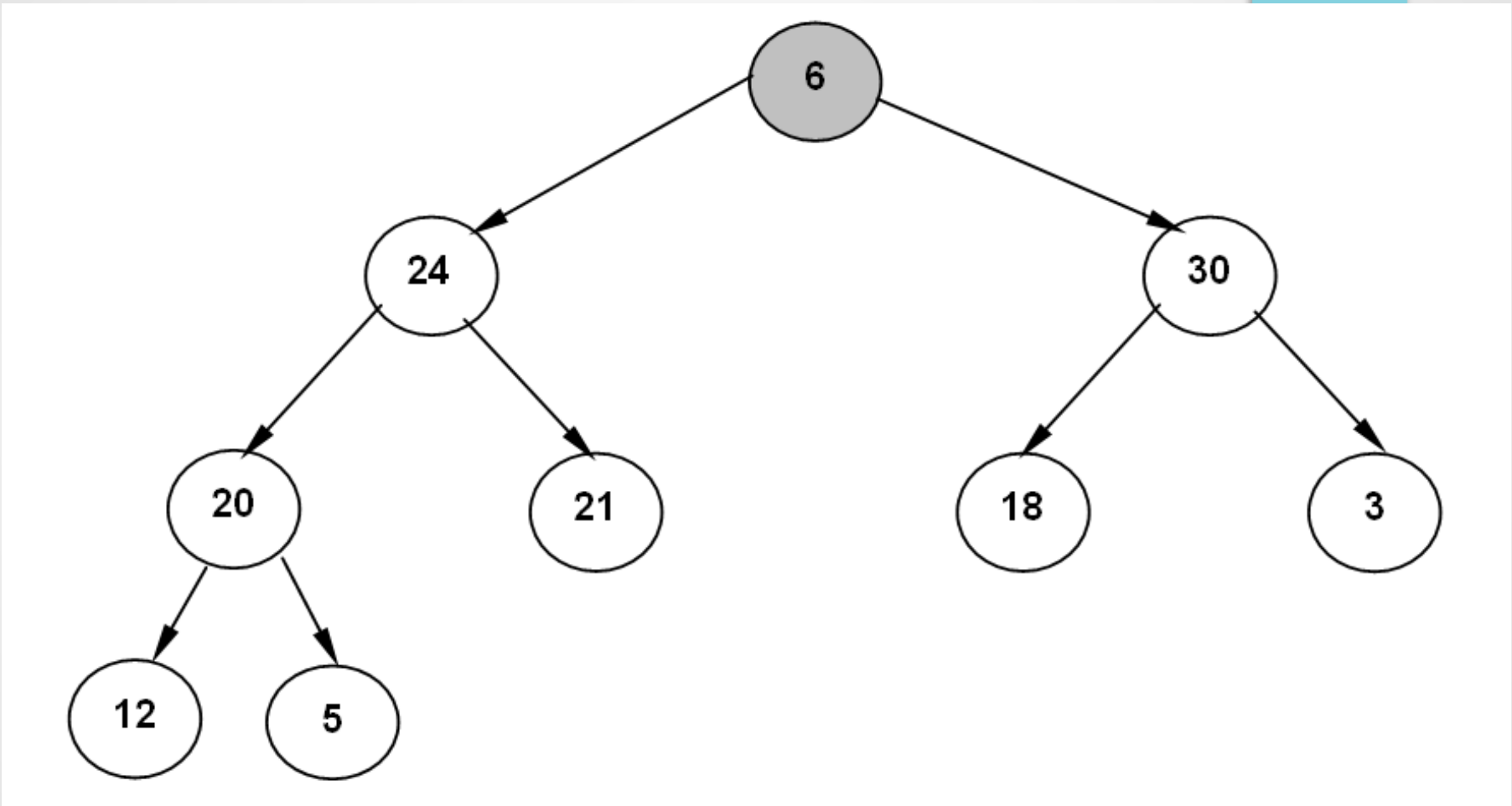
50	24	30	20	21	18	3	12	5	**6**

5.1.3 Heap Sort (Ordenamiento por Montón)



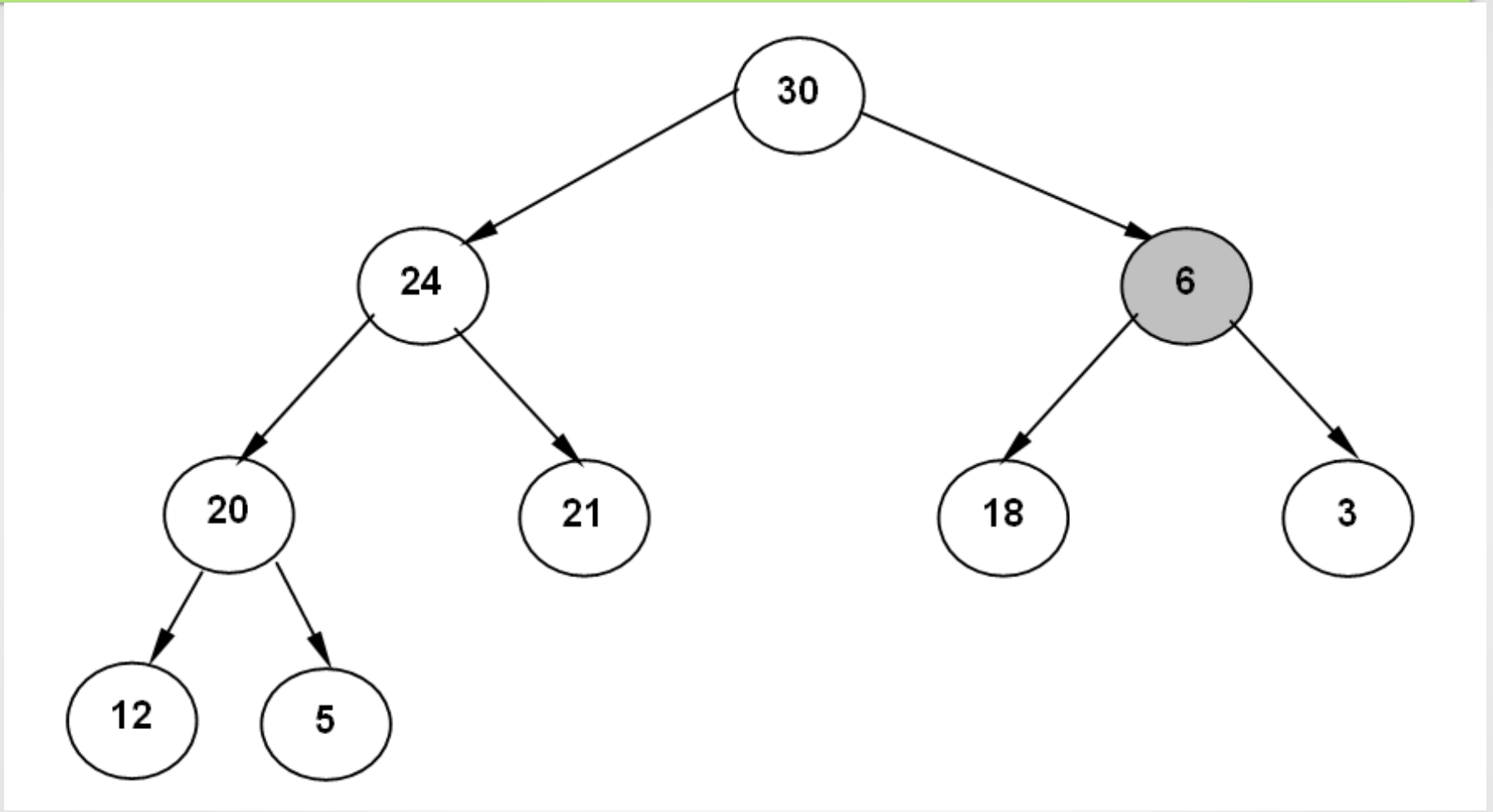
	24	30	20	21	18	3	12	5	**6**

5.1.3 Heap Sort (Ordenamiento por Montón)



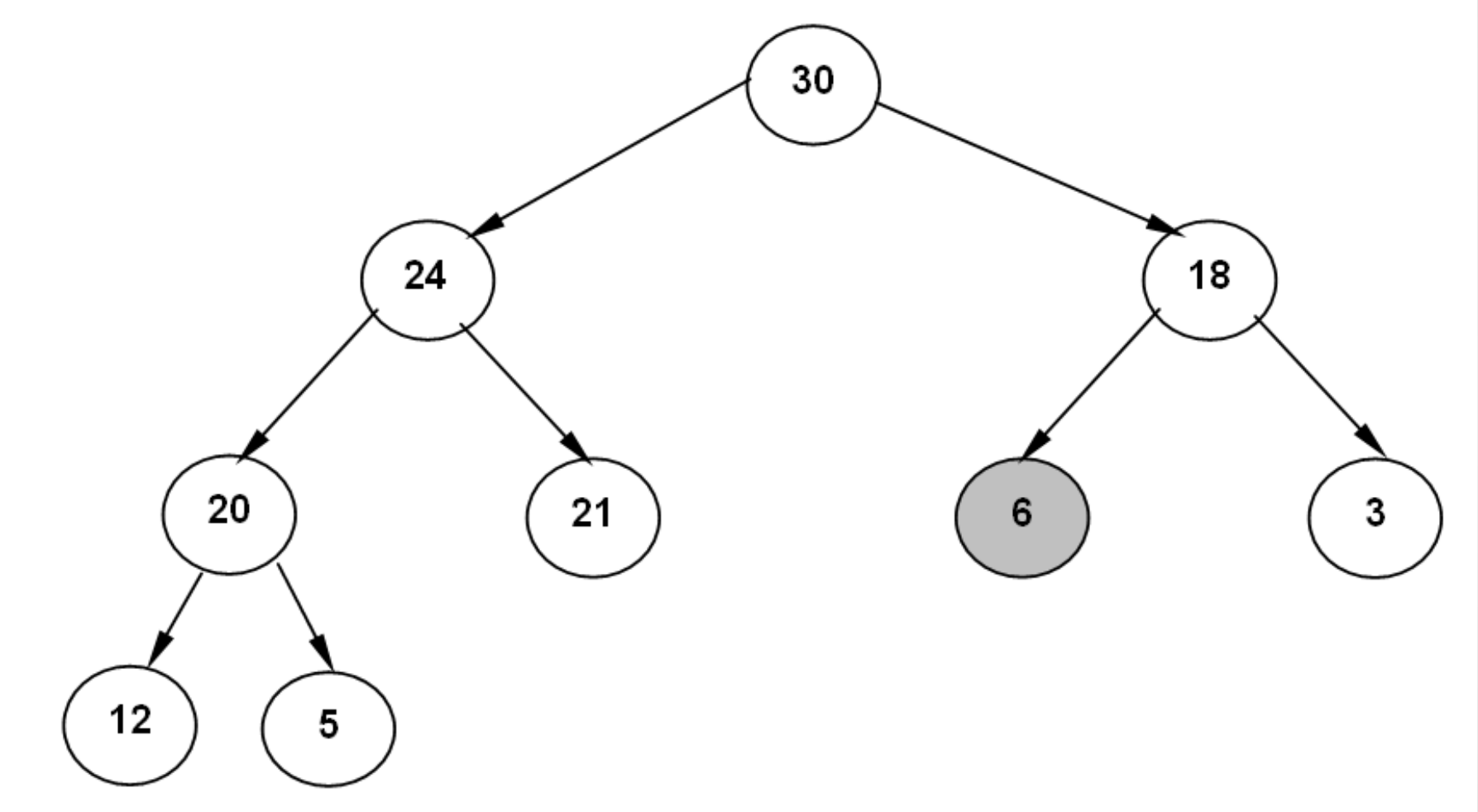
6	24	30	20	21	18	3	12	5	50

5.1.3 Heap Sort (Ordenamiento por Montón)



30	24	6	20	21	18	3	12	5	50

5.1.3 Heap Sort (Ordenamiento por Montón)



30	24	18	20	21	**6**	3	12	5	**50**

5.1.3 Heap Sort (Ordenamiento por Montón)

Del procedimiento que se acaba de conocer, se desprenden los siguientes fundamentos para la CONSTRUCCIÓN DE UN MONTÓN

- A partir de un nodo que tiene como hijos a las raíces de dos montones, se puede construir un nuevo montón.
- Cada hoja es un montón.

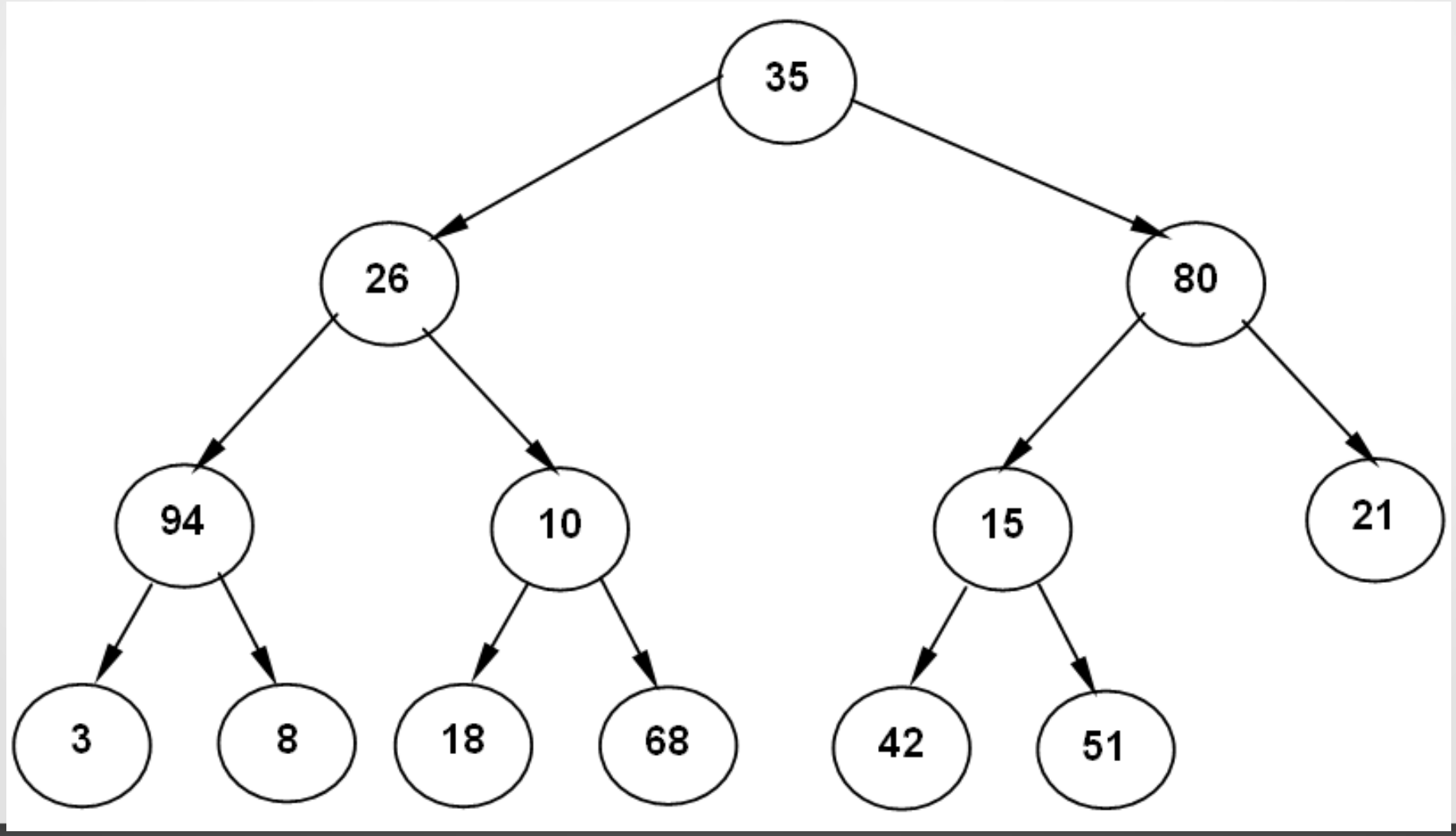
5.1.3 Heap Sort (Ordenamiento por Montón)

Algoritmo:

- Se inicia el proceso con el último nodo interno (penúltimo nivel).
- Como sus hijos (o hijo) son montones solo hay que verificar si su llave es mayor que la de ellos.
- Si cumple con esa condición, significa que la llave está en su lugar.
- Si no cumple con la segunda condición, se debe bajar la llave hasta el lugar adecuado.
- El proceso continúa con el siguiente nodo a la izquierda, si ya terminó un nivel, se pasa al nivel anterior con el primer nodo de la derecha.
- Una vez terminado el proceso (cuando se acomoda la raíz), el archivo completo es un montón.

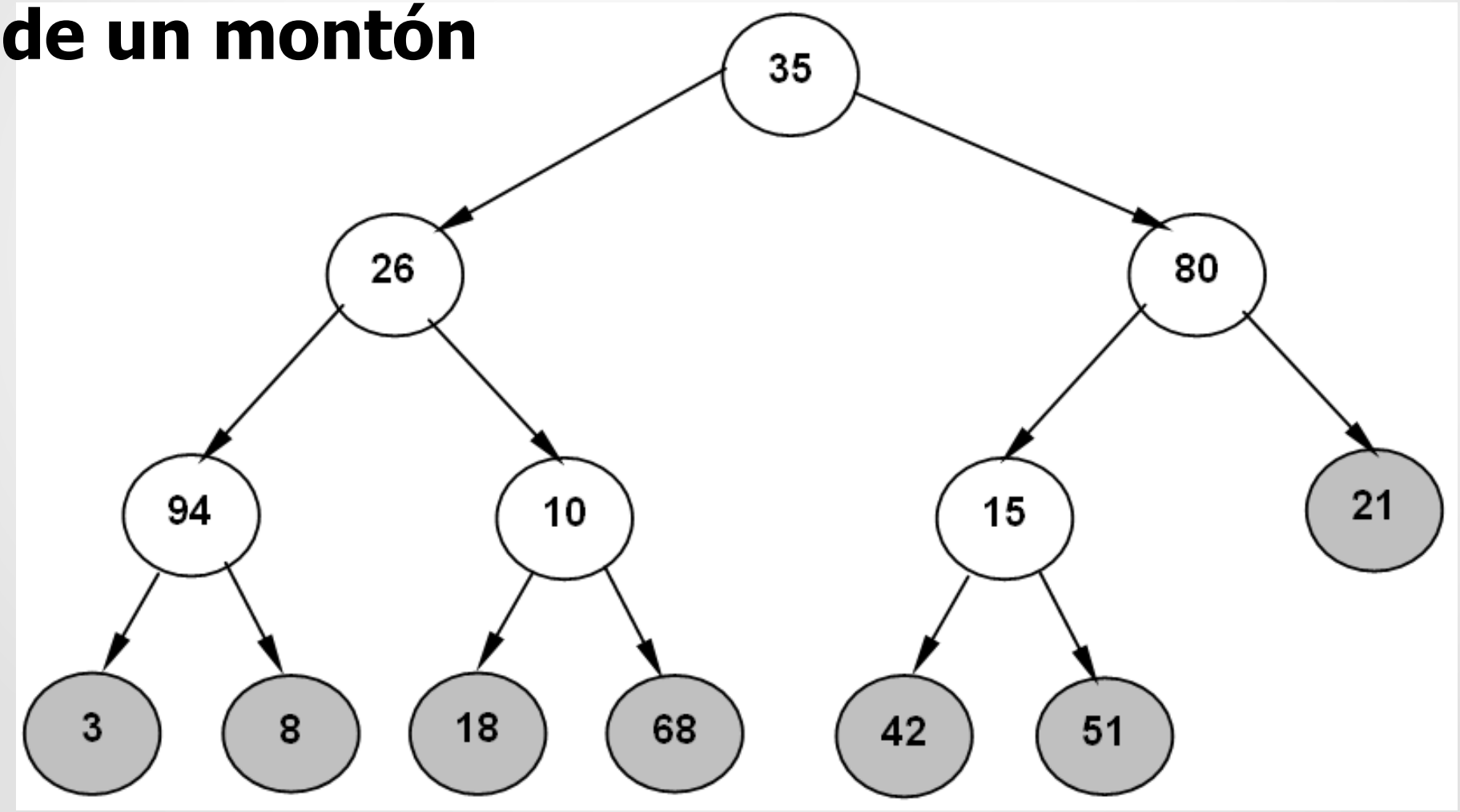
5.1.3 Heap Sort (Ordenamiento por Montón)

35	26	80	94	10	15	21	3	8	18	68	42	51

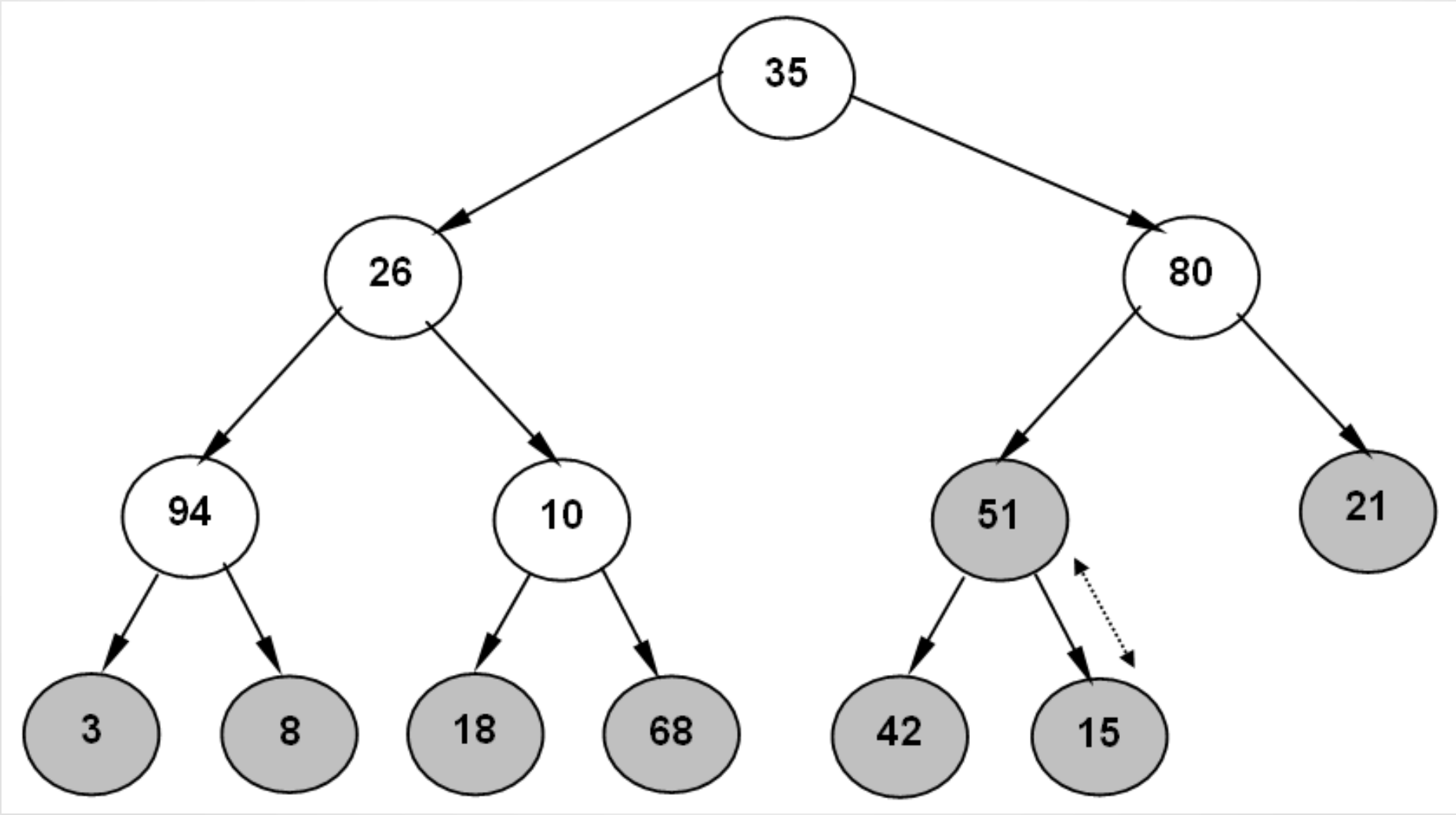


5.1.3 Heap Sort (Ordenamiento por Montón)

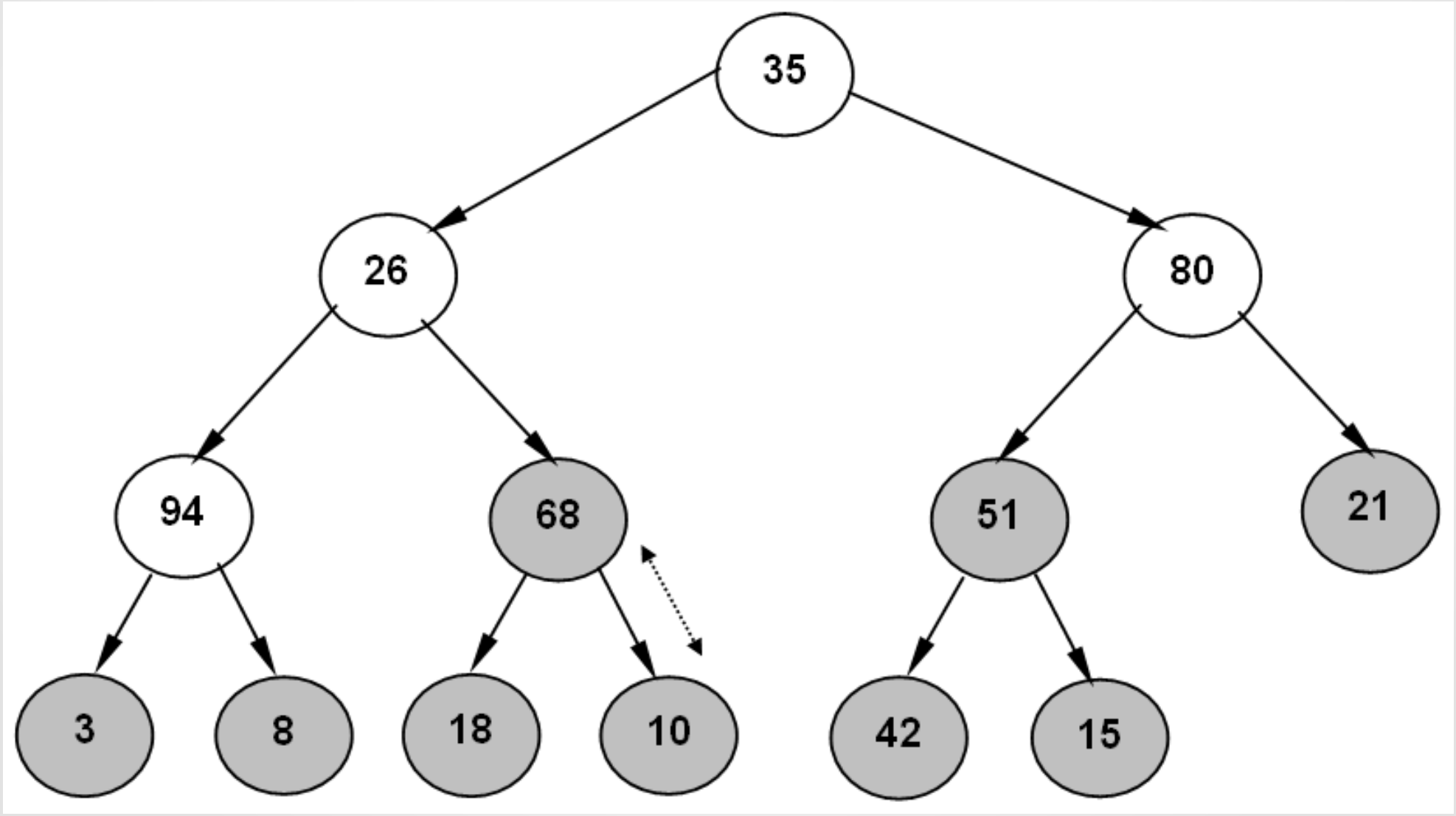
Cada nodo sombreado representa a la raíz de un montón



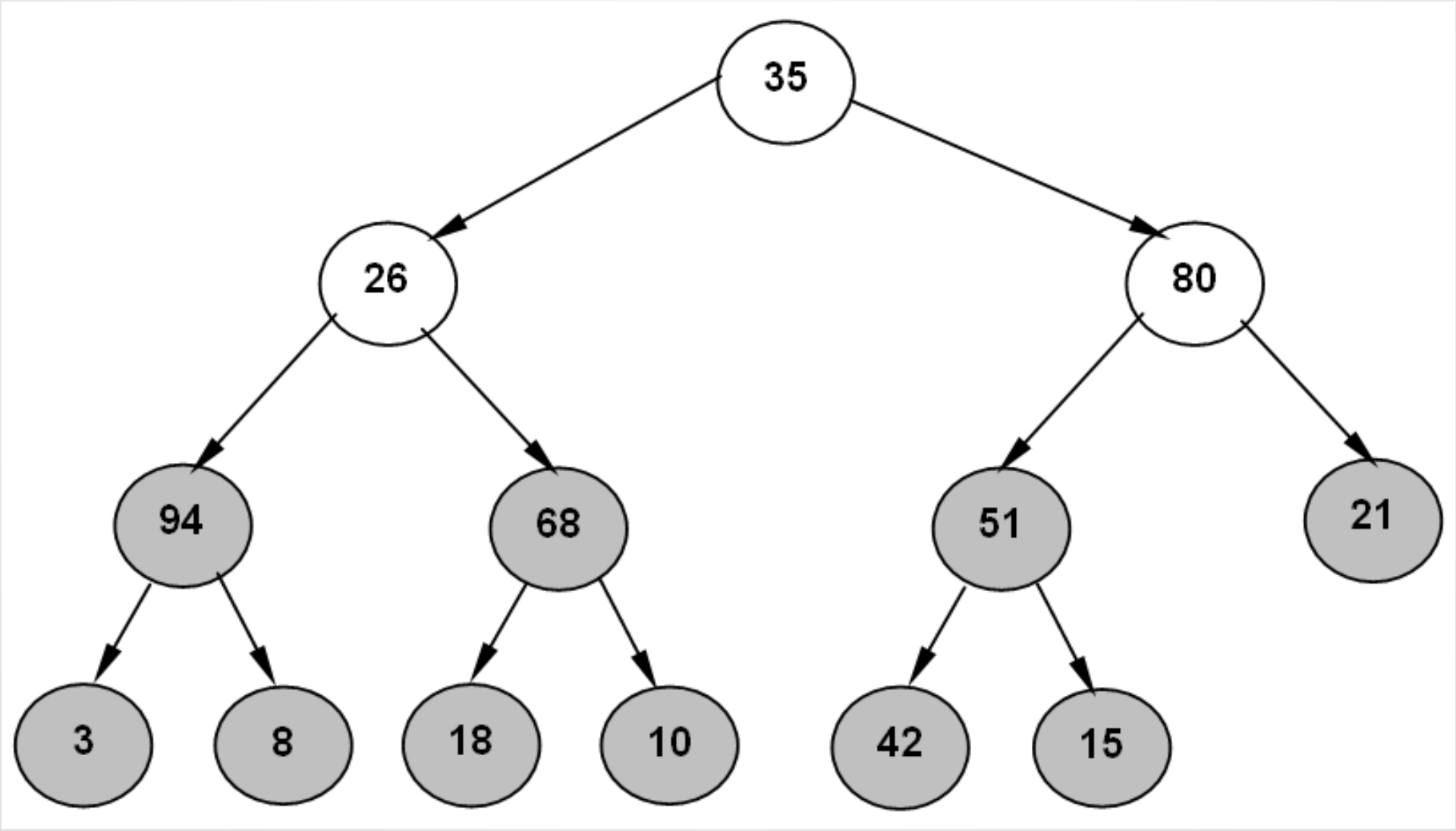
5.1.3 Heap Sort (Ordenamiento por Montón)



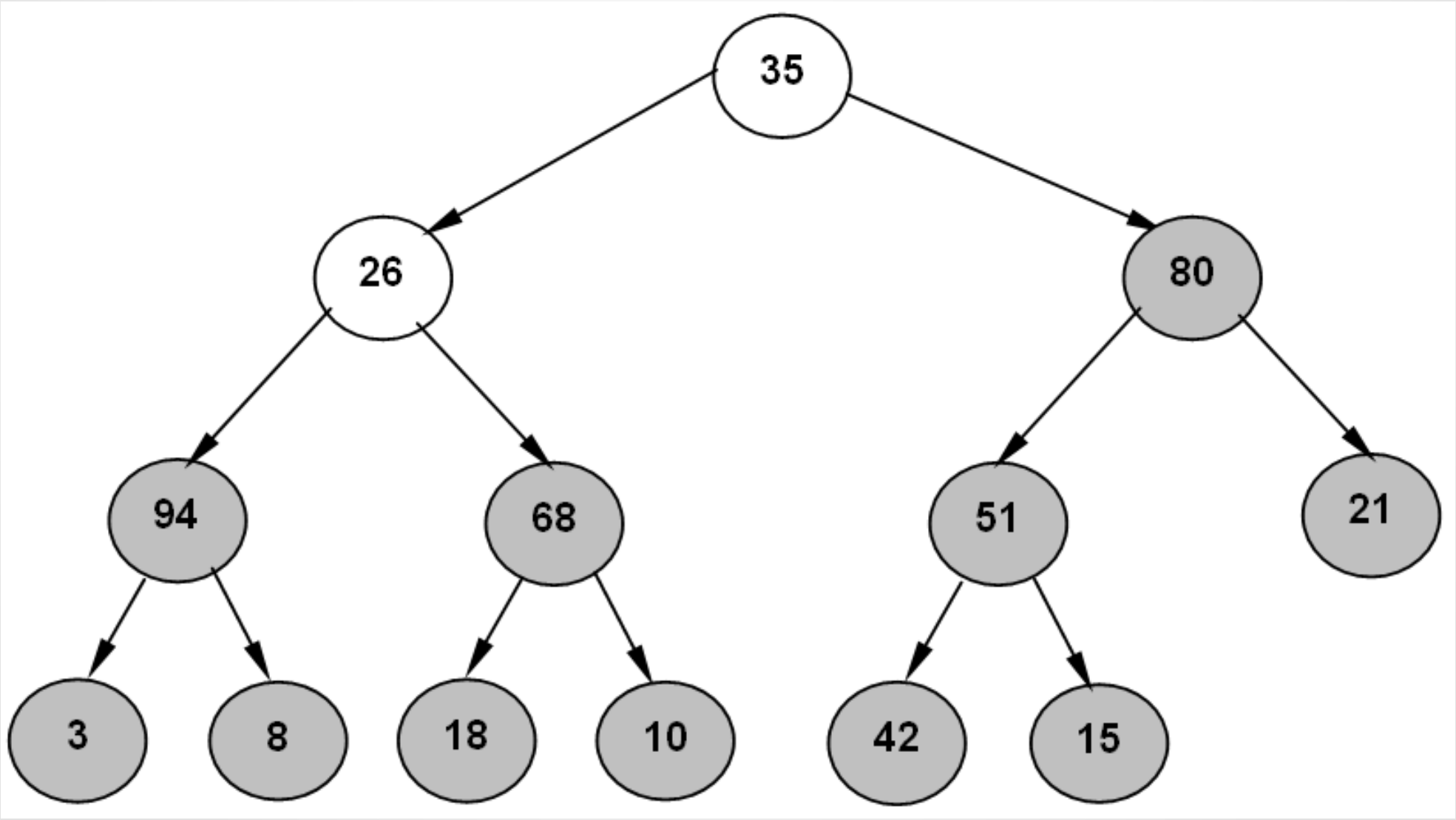
5.1.3 Heap Sort (Ordenamiento por Montón)



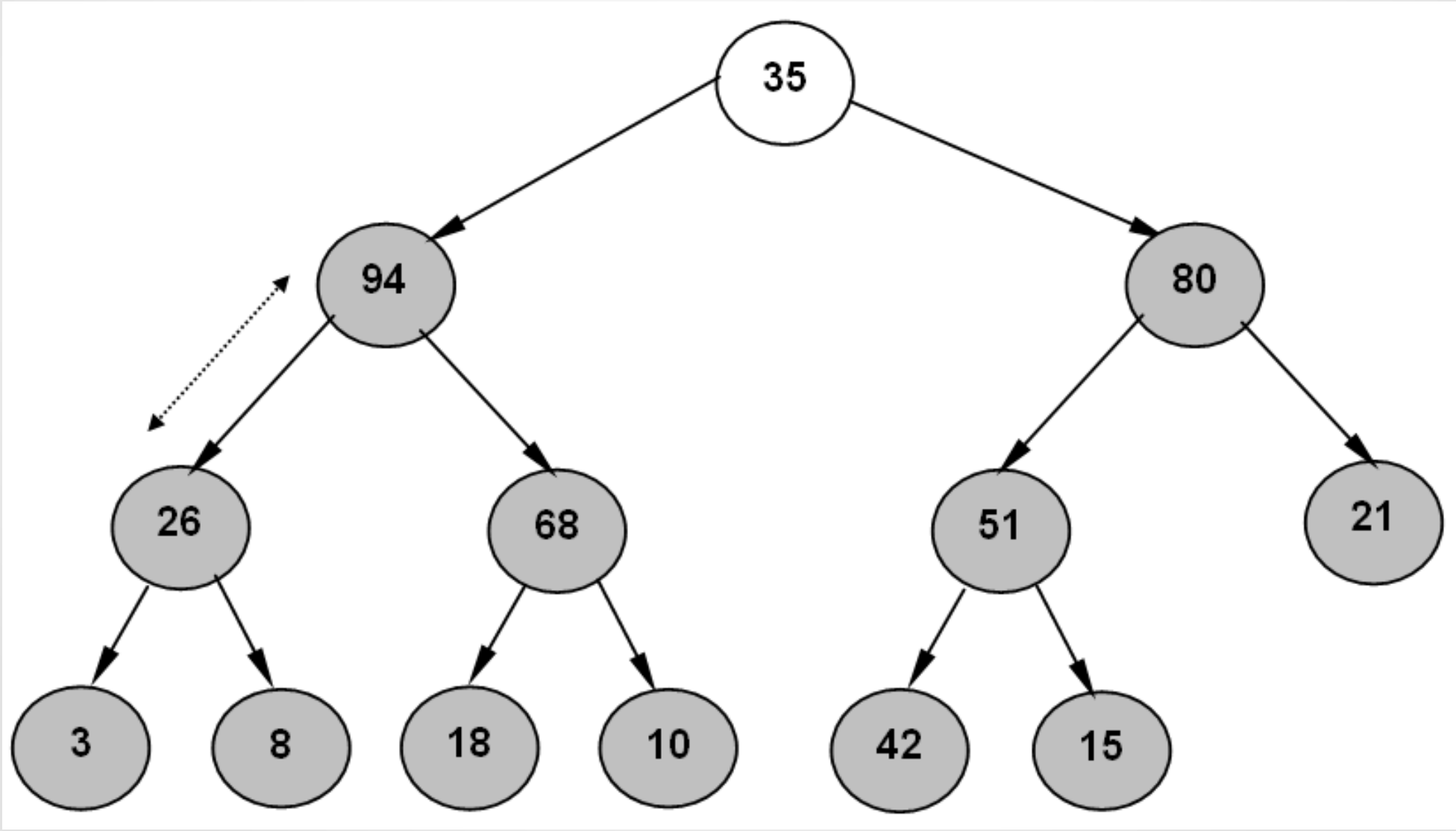
5.1.3 Heap Sort (Ordenamiento por Montón)



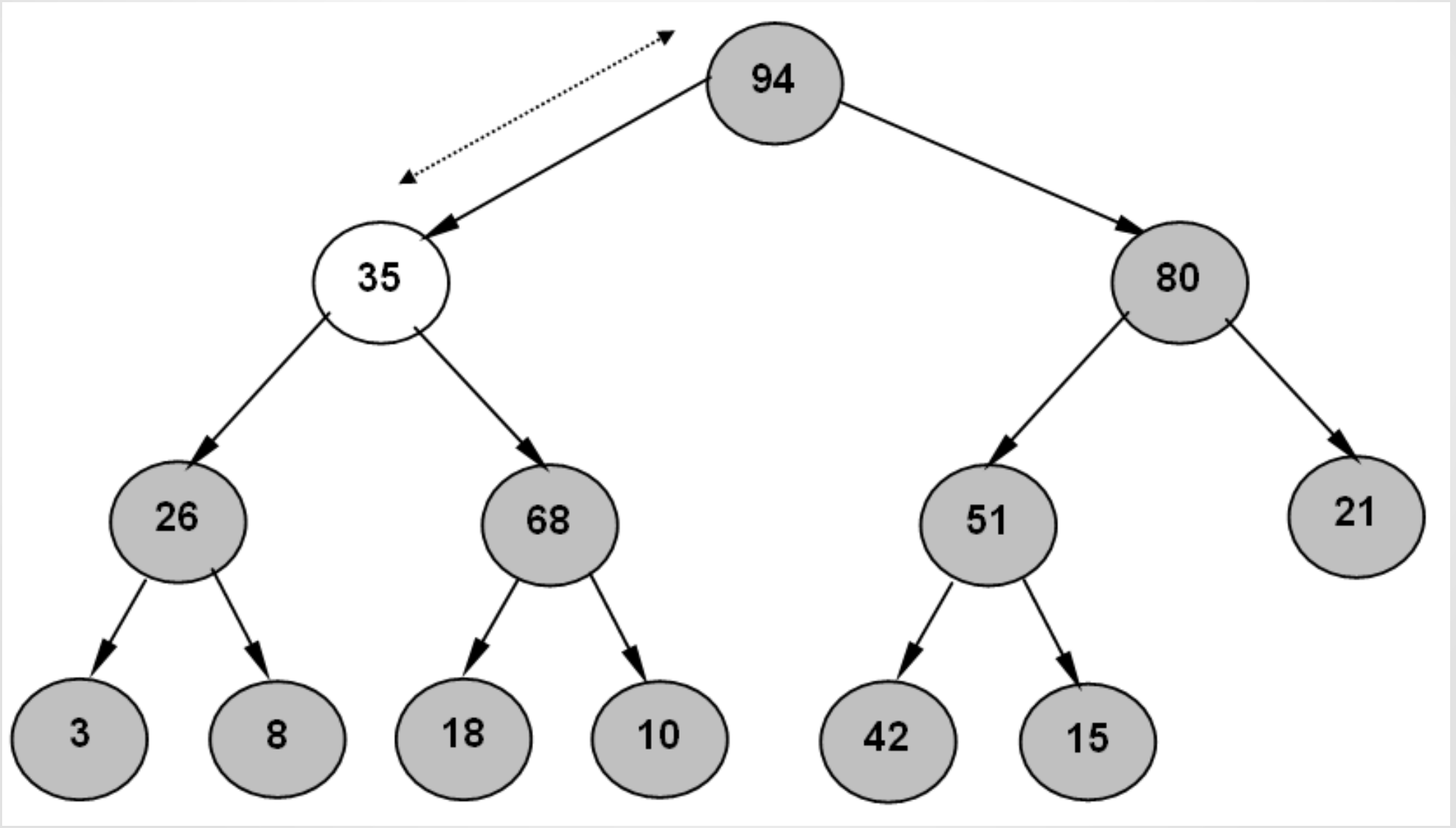
5.1.3 Heap Sort (Ordenamiento por Montón)



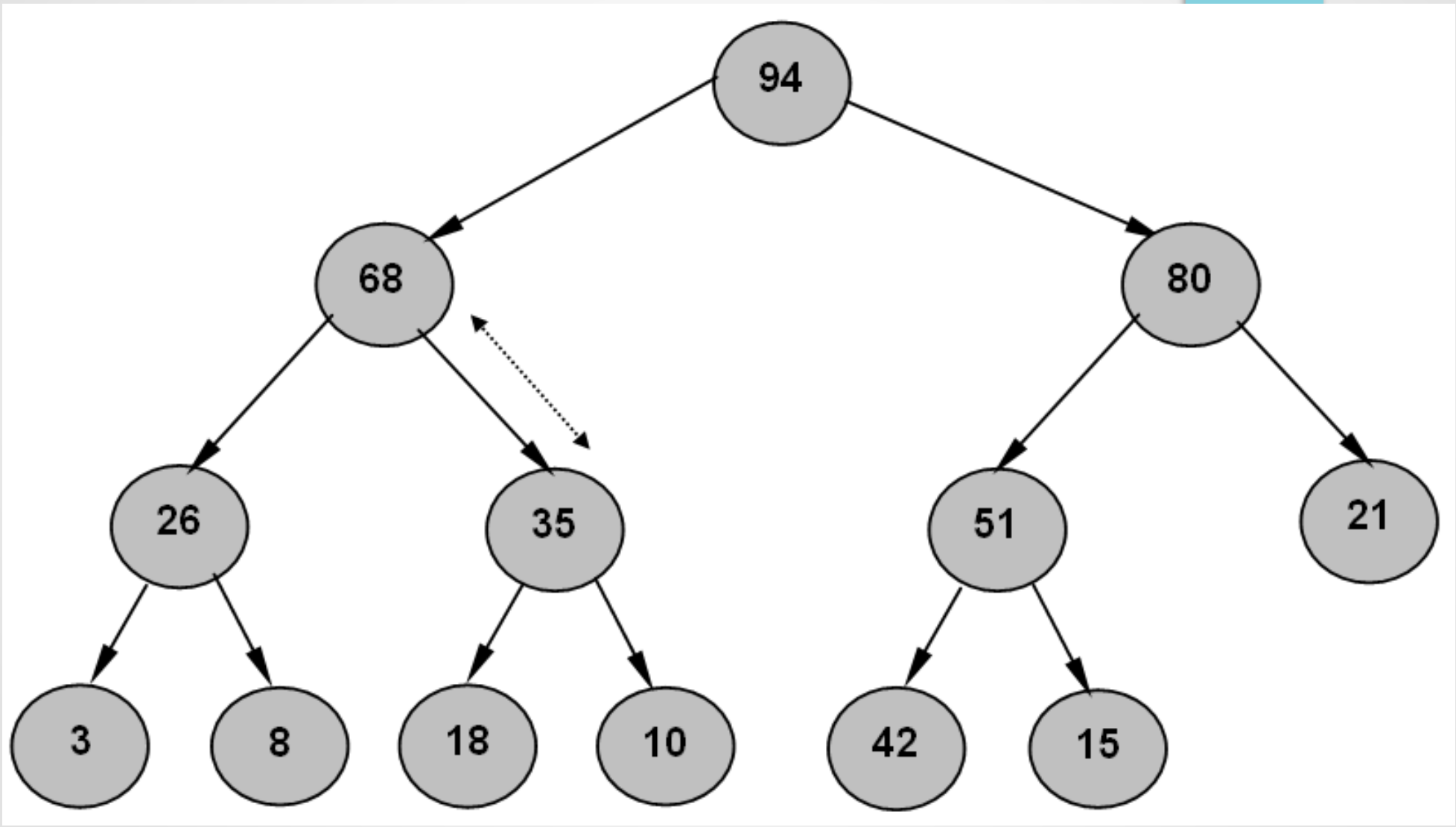
5.1.3 Heap Sort (Ordenamiento por Montón)



5.1.3 Heap Sort (Ordenamiento por Montón)



5.1.3 Heap Sort (Ordenamiento por Montón)



El archivo ya es un montón

5.1.3 Heap Sort (Ordenamiento por Montón)

Observaciones de utilidad para la programación

Si n es par	el último nodo interno tiene un hijo
Si n es impar	el último nodo interno tiene dos hijos

Posiciones importantes en el vector

0	dirección de la raíz
n-1	dirección de la última hoja
int(n/2)-1	dirección del último nodo interno
2i+1	dirección del hijo izquierdo del nodo i
2i+2	dirección del hijo derecho del nodo i

5.1.3 Heap Sort (Ordenamiento por Montón)

Cantidad de trabajo realizado por HeapSort (solo la construcción del montón) y comprobación del orden de complejidad temporal n log n

n	Número de intercambios	n*logn	Intercambios / (n*logn)
10,000	7,386	40,000.00	0.1847
50,000	37,115	234,948.50	0.1580
100,000	73,819	500,000.00	0.1476
200,000	148,118	1,060,206.00	0.1397
350,000	258,096.00	1,940,423.82	0.1330
500,000	369,440.00	2,849,485.00	0.1297

5.1.4 Inserción Simple

Se basa en la premisa de que un archivo que contiene solo un registro, ya está ordenado.

Razonamiento:

Se supone que el primer registro de un archivo de tamaño **n**, conforma un subarchivo ya ordenado y los restantes **n–1** registros forman un archivo desordenado.

5.1.4 Inserción Simple

Archivo Original

34
10
15
40
22
20

Según el razonamiento de Inserción Simple

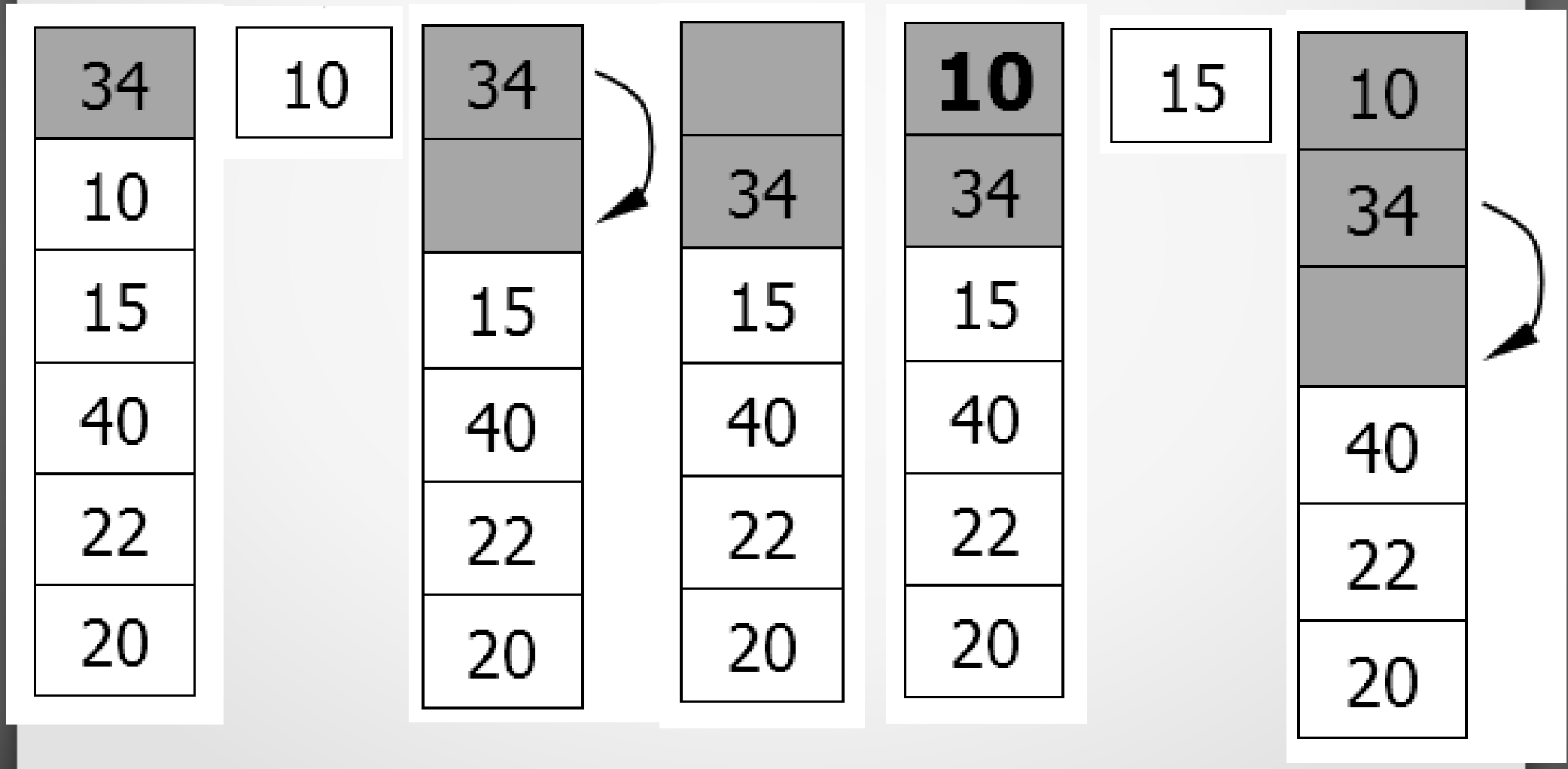
34	Subarchivo ordenado
10	Subarchivo desordenado
15	
40	
22	
20	

5.1.4 Inserción Simple

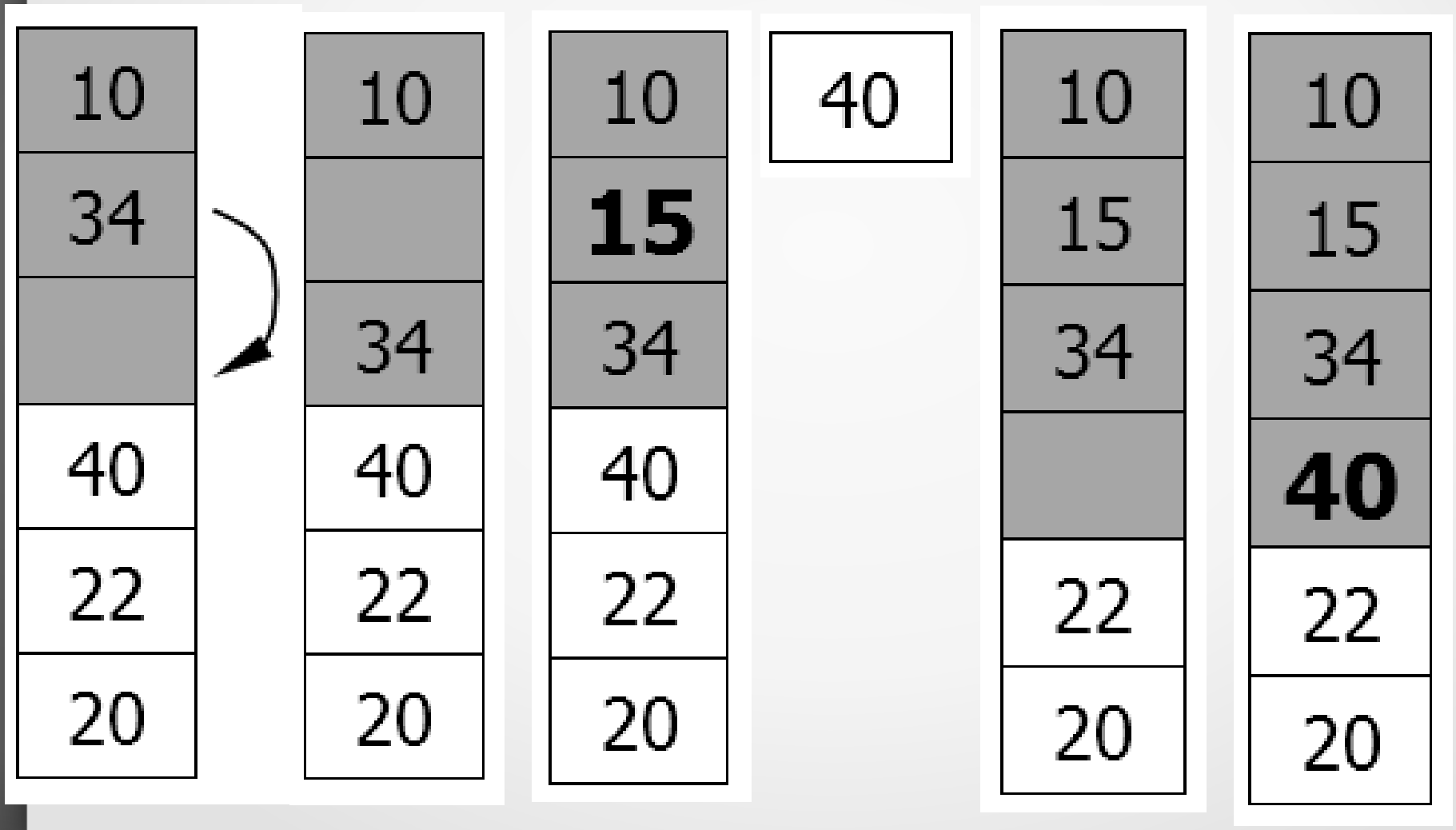
Procedimiento

- Se toma el primer registro del subarchivo desordenado y se coloca dentro del subarchivo ordenado que ahora tendrá un registro más pero deberá seguir ordenado.
- Ahora el subarchivo desordenado reducirá su tamaño a **n–2** registros.
- Los procesos anteriores se repiten hasta que el subarchivo ordenado sea de tamaño **n**.
- El nombre del método se toma del proceso de **insertar** uno de los registros del segmento desordenado dentro del ordenado.

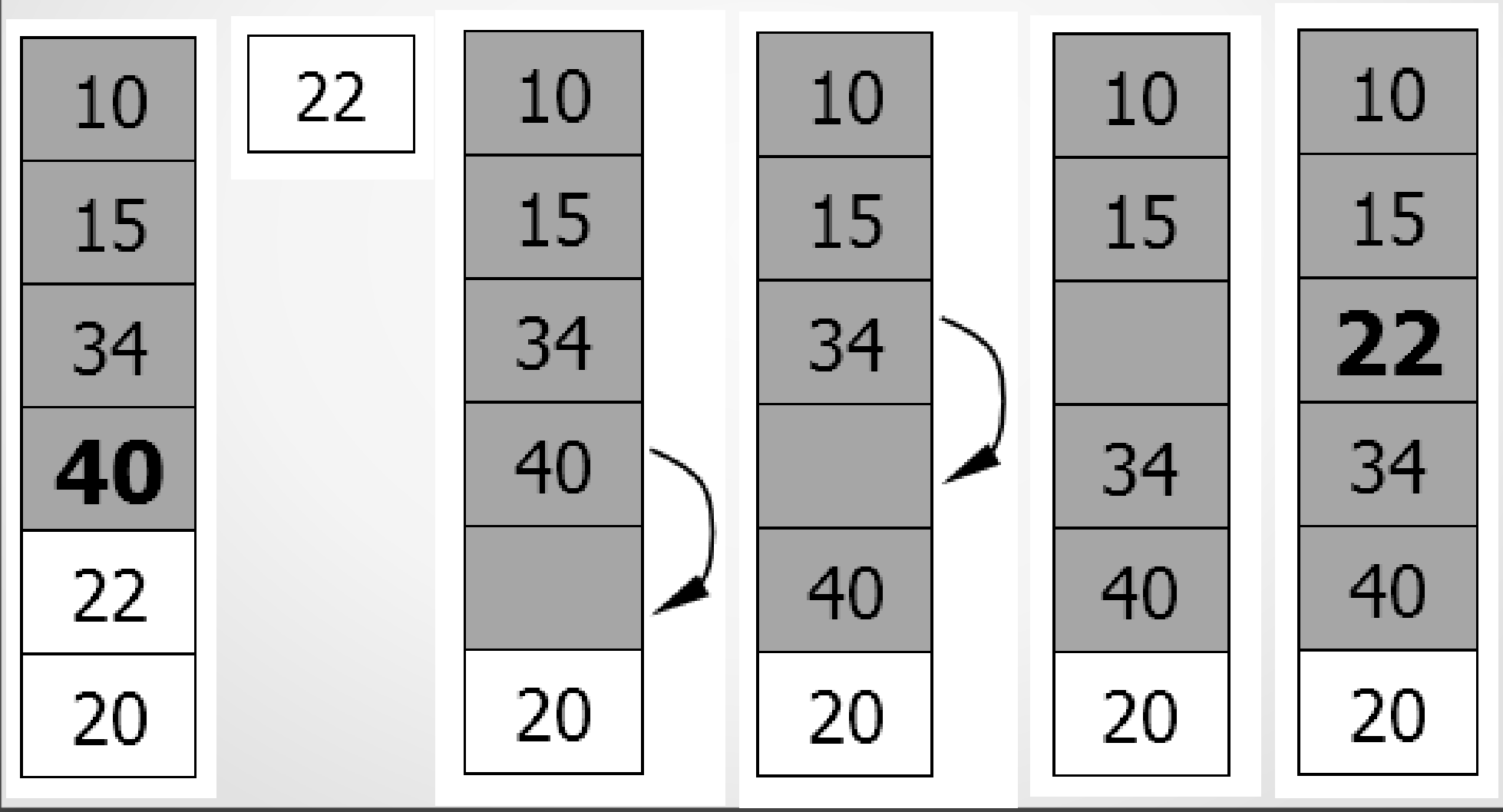
5.1.4 Inserción Simple



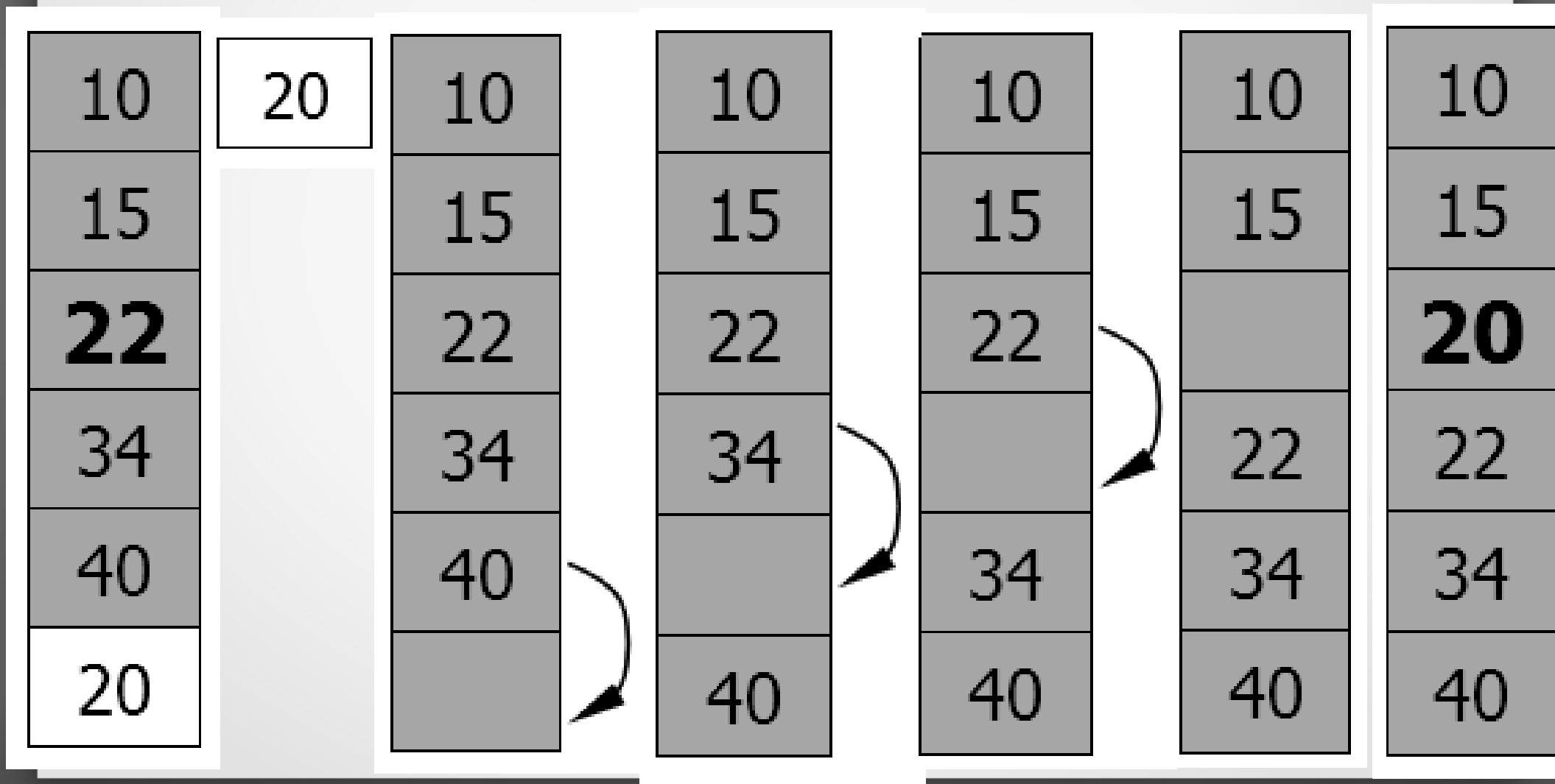
5.1.4 Inserción Simple



5.1.4 Inserción Simple



5.1.4 Inserción Simple



5.1.4 Inserción Simple

Código C++ para inserción simple

```
for(int j=1;j<n;j++){
     int temp=archivo[j];
     int i=j-1;
     while ((archivo[i]>temp)and(i>=0)){
          archivo[i+1]=archivo[i];
          i--;
     }
     archivo[i+1]=temp;
}
```

5.1.5 Ordenamiento Shell

- No es un método de ordenamiento sino un refinamiento para otros.
- El Dr. Donald Shell sugirió que el archivo a ordenar se divida en forma lógica.
- El número y tamaño de los subarchivos se determina por un concepto llamado ***Incremento:*** *distancia que hay entre 2 registros de un mismo subarchivo*.
- Se aplica una serie de incrementos en forma descendente pero el último debe ser 1.

5.1.5 Ordenamiento Shell

- Cada subarchivo se ordenará por separado de los demás.
- Se diseñó inicialmente para Inserción Simple ya que la cantidad de cantidad de trabajo de ese método se debe a que las comparaciones se aplican a datos en direcciones consecutivas de memoria.

5.1.5 Ordenamiento Shell

Ejemplo:

44	55	12	42	94	18	6	67

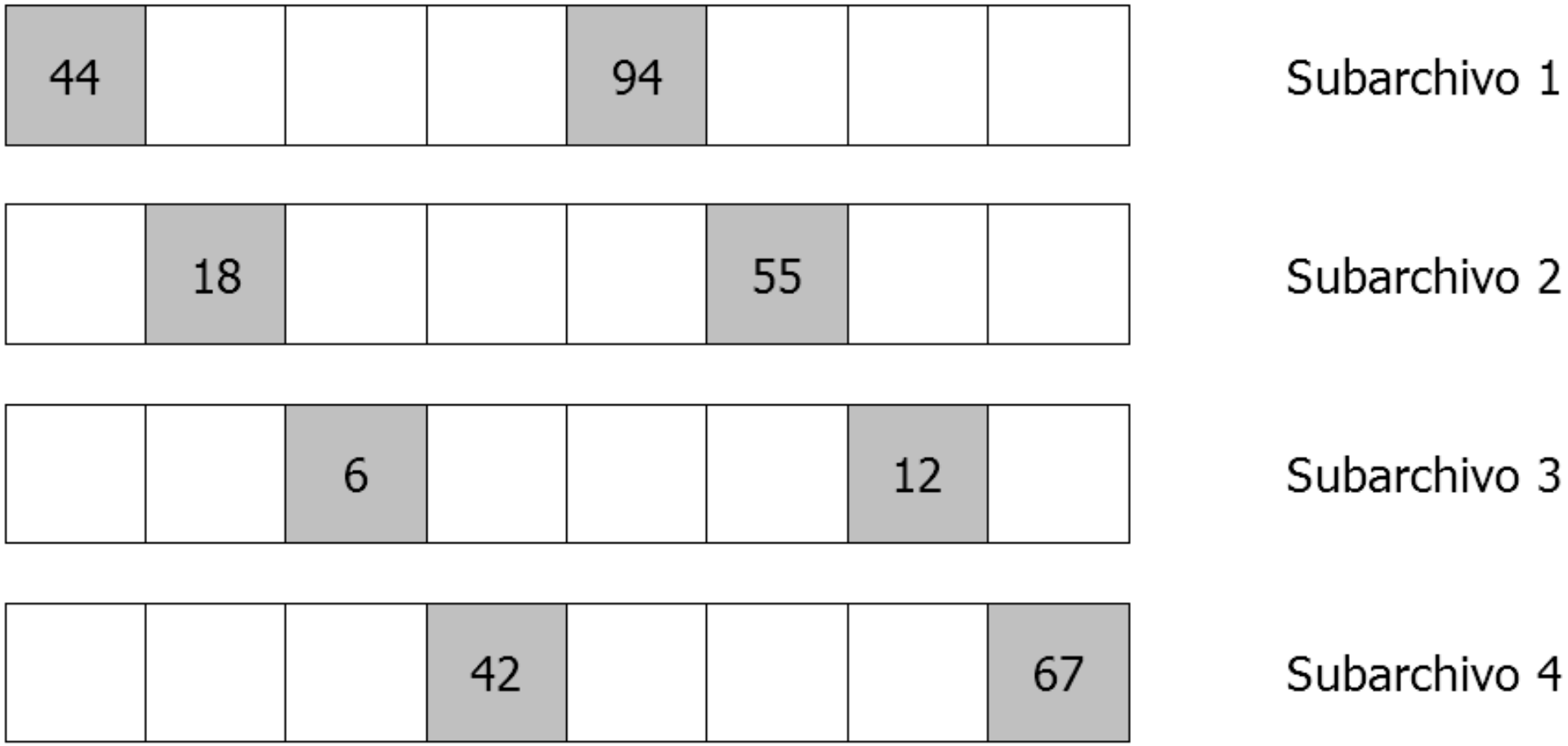
44	55	12	42	94	18	6	67

Supongamos que conviene aplicar un **incremento de 4** (el número de subarchivos que se obtiene es 4).

44				94				Subarchivo 1
	55				18			Subarchivo 2
		12				6		Subarchivo 3
			42				67	Subarchivo 4

5.1.5 Ordenamiento Shell

Los subarchivos se ordenan independientemente:



5.1.5 Ordenamiento Shell

El archivo antes de aplicado el **incremento 4:**

44	55	12	42	94	18	6	67

Una vez aplicado el incremento 4 y ordenados por separado:

44	18	6	42	94	55	12	67

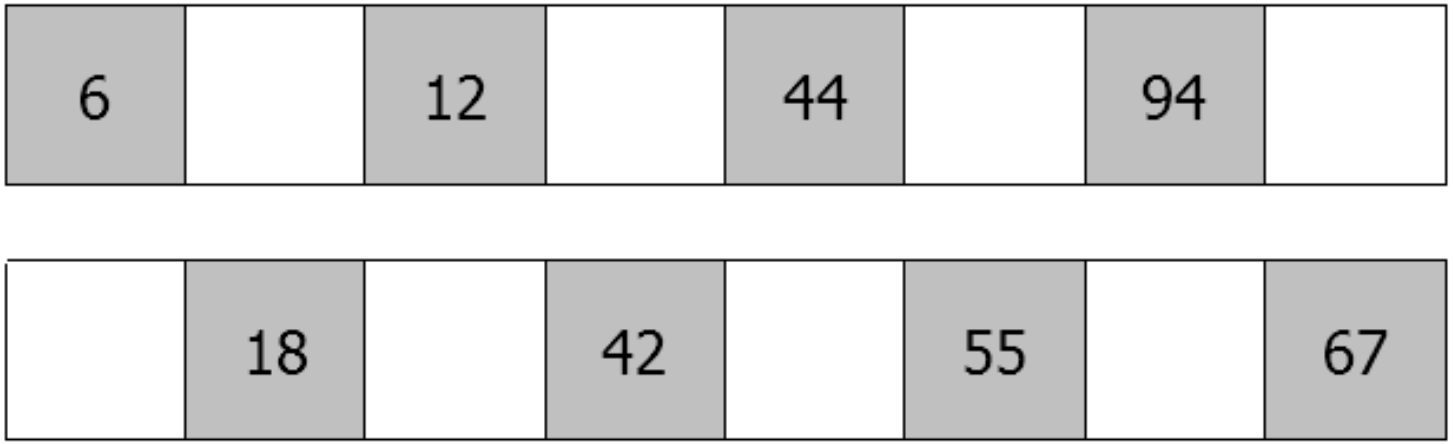
El archivo no quedó ordenado, pero se ha demostrado, que los registros queden más cerca de su lugar definitivo.

5.1.5 Ordenamiento Shell

Se aplica el siguiente incremento, dos (2) por ejemplo.

5.1.5 Ordenamiento Shell

Se ordenan los subarchivos



5.1.5 Ordenamiento Shell

El archivo completo:

6	18	12	42	44	55	94	67

¿Cómo se encuentra el archivo?, ¿con las llaves en una mejor posición?

5.1.5 Ordenamiento Shell

Finalmente se debe usar incremento 1:

6	18	12	42	44	55	94	67

Subarchivo 1

Una vez ordenado el único subarchivo, el archivo completo quedará:

6	12	18	42	44	55	67	94

Comparación de Inserción Simple para un archivo completo y para un subarchivo (SHELL).

```
for(int j=1; j<n; j++){
        int temp=archivo[j];
        int i=j-1;
        while ((archivo[i]>temp)and(i>=0)){
                archivo[i+1]=archivo[i];
                i--;}
        archivo[i+1]=temp;
}
```

```
for(long j=subarchivo+increm-1; j<n; j=j+increm){
        long temp=archivo[j];
        long i=j-increm;
        while ((archivo[i]>temp)and(i>=subarchivo-1)){
                archivo[i+increm]=archivo[i];
                i = i - increm; }
        archivo[i+increm]=temp;
}
```

El subarchivo 1 empieza en la posición 0, el subarchivo 2 empieza en la posición 1 y así sucesivamente.

j toma como valor inicial el segundo elemento del subarchivo (la dirección "increm" es el 2o elemento del subarchivo 1)

5.1.5 Ordenamiento Shell

Análisis del Algoritmo Shell

- Los archivos pequeños requieren menos trabajo para ordenarse.
- La **descomposición lógica** de un archivo en segmentos más pequeños, ocasiona que el esfuerzo total sea menor.
- Los movimientos de los datos cuando el incremento es grande, acercan los datos a su lugar definitivo.
- La secuencia de incrementos podría ser seleccionada arbitrariamente.
- Única recomendación: los incrementos no deben ser múltiplos unos de otros.

5.1.5 Ordenamiento Shell

Donald Knuth recomienda las dos secuencias de incrementos siguientes (escritas en ORDEN INVERSO):

1, 4, 13, 40, 121, ... donde $\mathbf{H_{i-1} = 3H_i + 1}$

y

1, 3, 7, 15, 31, ... donde $\mathbf{H_{i-1} = 2H_i + 1}$

Empíricamente se ha comprobado que el orden de complejidad temporal del proceso Shell es $\mathbf{n^{1.2}}$ (enseguida se muestra una tabla de demostración). El código C++ se puede descargar del sitio www.felipealanis.org

5.1.5 Ordenamiento Shell

Cantidad de trabajo realizado por ShellSort y comprobación del orden de complejidad temporal $n^{1.2}$

n	Número de movimientos	$n^{1.2}$	movimientos / $n^{1.2}$
10,000	106,039	63,095.73	1.6806
50,000	744,459	435,275.28	1.7103
100,000	1,737,630	1,000,000.00	1.7376
200,000	3,824,850	2,297,396.71	1.6649
250,000	5,247,470.00	3,002,811.08	1.7475
300,000	5,991,380.00	3,737,192.82	1.6032