Análisis y Diseño de Algoritmos.

Sesión 3. 3 de septiembre de 2015.

Maestría en Sistemas Computacionales.

Por: Hugo Iván Piza Dávila.

¿Qué veremos hoy?

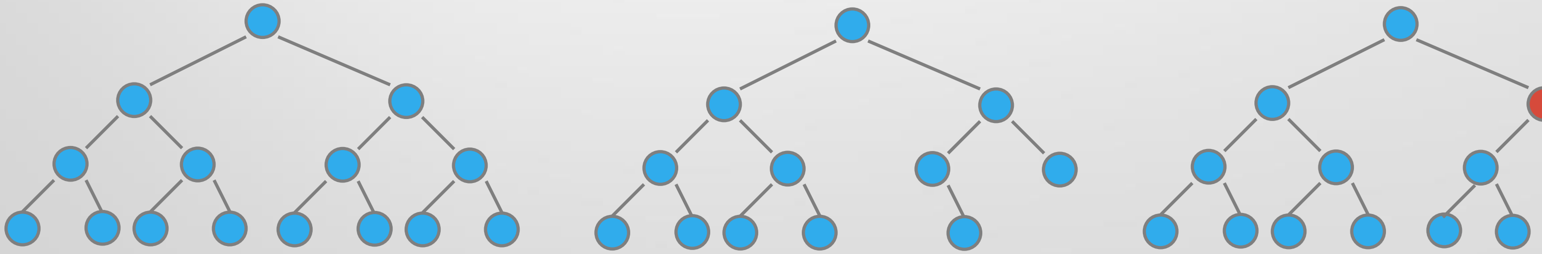
- Diseño y análisis de algoritmos de ordenamiento:
  - de complejidad cuasi-lineal
  - iterativos
- ¿Cuáles son?
  - Shell (tarea optativa)
  - Heapsort
  - Radix
- Ordenamiento por conteo: complejidad lineal

Heap Sort

- También llamado ordenamiento por *montículos*.
- Su complejidad temporal es N log N en los casos mejor, peor y promedio.
- Quicksort es con frecuencia más rápido, pero HeapSort es mejor en los casos críticos.
- El primer paso del algoritmo consiste en construir un montículo (heap) a partir del arreglo.
- El segundo paso (e iterativo) es eliminar el elemento más grande y sustituirlo por el que está colocado al final del montículo.

¿Qué es un montículo?

- Es un árbol binario con estas características:
  1. Cada nodo tiene un valor comparable tal que ningún nodo tiene un valor más grande que el de su padre.
  2. Está balanceado: cada nodo tiene 2 hijos, excepto los de los últimos dos niveles.
  3. Está alineado a la izquierda: si un nodo sólo tiene un hijo, debe ser el izquierdo.

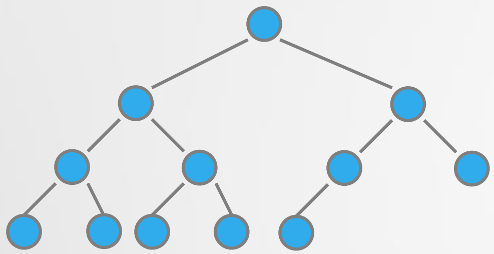


Balanceado

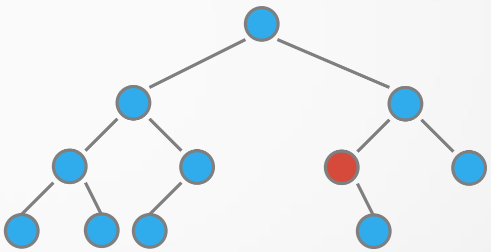
Balanceado

No balanceado

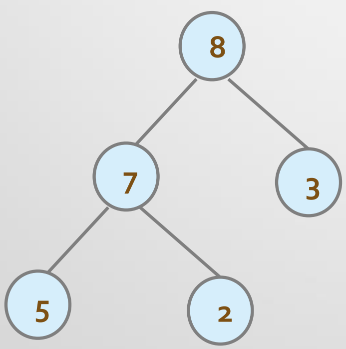
¿Qué es un montículo?



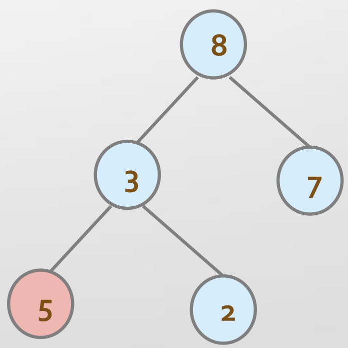
Alineado a la izquierda



No alineado a la izquierda



Cumple propiedad 1



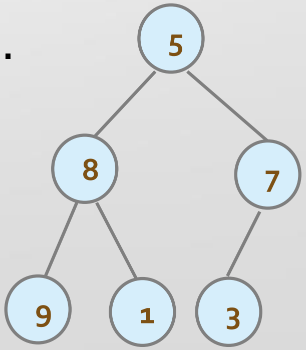
No cumple propiedad 1

¿Por qué con montículos?

1. El cumplimiento de la propiedad 1 nos facilitará realizar el ordenamiento en tiempo quasi-lineal.
2. El cumplimiento de las propiedades 2 y 3 nos permitirán tratar al arreglo recibido como árbol (sin necesidad de crear un árbol binario explícito).

- Correspondencia entre arreglo y árbol binario:
  - Nótese que no cumple con la propiedad 1.

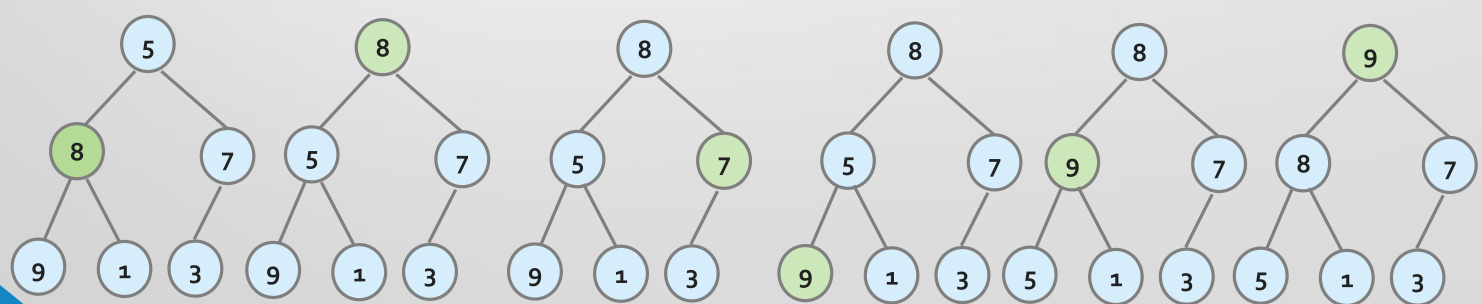
5	8	7	9	1	3



Primer paso

- Convertir el arreglo en montículo:
  1. Comenzar en el 2º elemento: hijo izquierdo de la raíz.
  2. Obtener el índice del padre… ¿cuál es la fórmula?
  3. Si el elemento actual es mayor que su padre:
     a) Intercambiarlos
     b) Regresar al paso 3 hasta llegar a la raíz
  4. Regresar al paso 1 con el siguiente elemento.

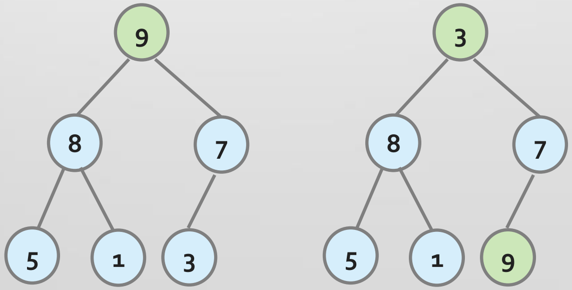
5 <u>8</u> 7 9 1 3 → 9 8 7 5 1 3



Segundo paso

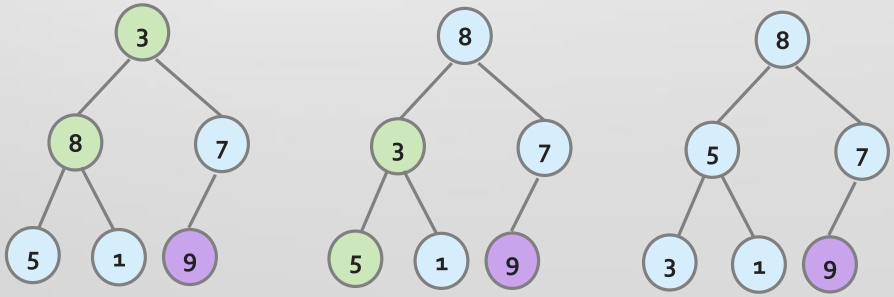
- Eliminar iterativamente el elemento más grande y sustituirlo por el que está al final del montículo.

1. El elemento más grande siempre está en la raíz.
2. Intercambiar el elemento de la raíz por el del final.
    - El elemento más grande ya quedó en su posición definitiva: esa posición ya no se visitará y el nuevo final del montículo es la posición anterior.
    - Se pierde la propiedad 1 de los montículos.



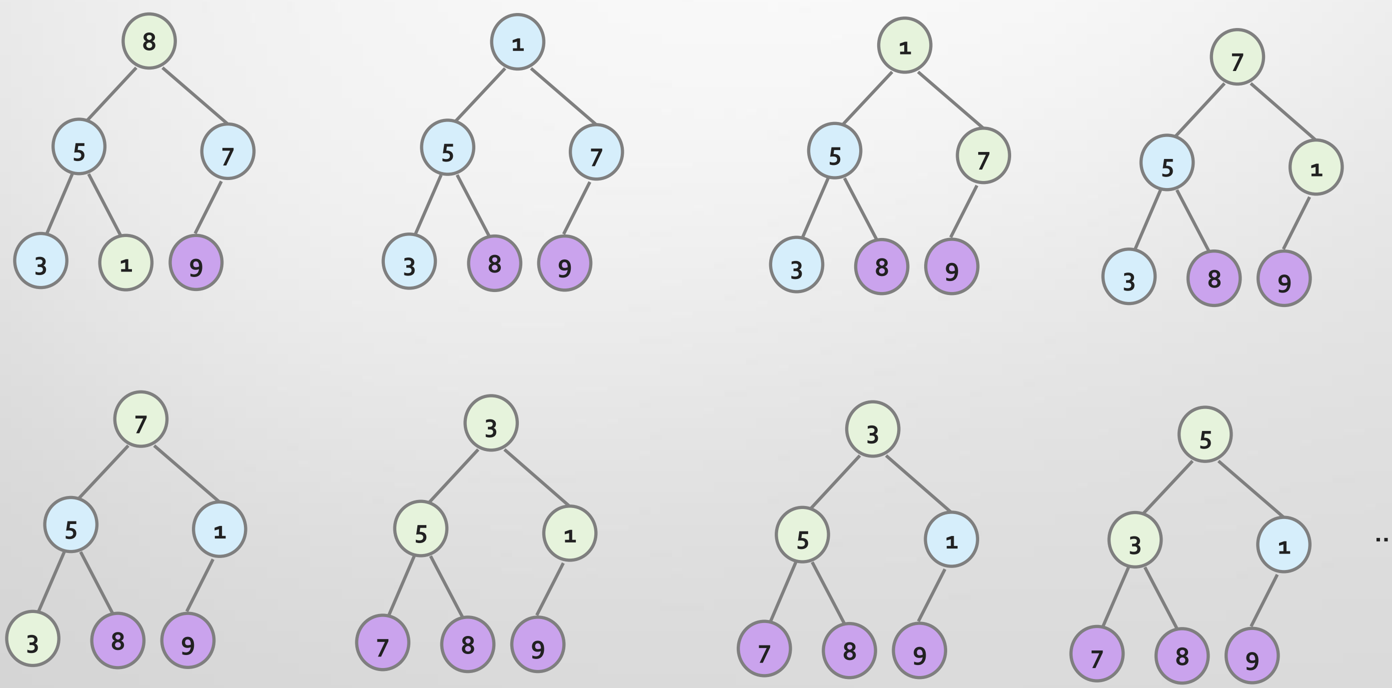
Segundo paso

3. Llevar la nueva raíz tan abajo como sea necesario hasta que se recupere la propiedad 1 de los montículos: *push-down*.
   a. Compararlo con el mayor de los hijos.
      ¿Cuál es la fórmula para acceder a los dos hijos?
      - Si el hijo derecho no existe, el mayor será el hijo izquierdo.
      - No se puede dar que el hijo izquierdo no existe y el derecho sí.
   b. Si es menor,
      - Intercambiarlos.
      - Regresar al paso 3.a hasta llegar a las hojas del montículo.



Segundo paso

4. Regresar al paso 2 tomando como último elemento del montículo, el anterior al elegido en la pasada anterior.



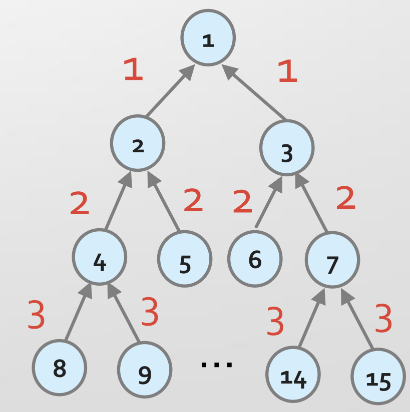
Análisis de Heapsort

- Los pasos 1 y 2 llevan la misma lógica en el recorrido del arreglo, pero en sentido contrario: analizaremos uno y multiplicaremos por 2 el resultado.
- Nos interesa saber cuánto es lo más que puede tardar: peor caso.
  - El peor caso es cuando cada elemento actual tiene que recorrerse siempre hasta la raíz o hasta la hoja.
- Tomaremos dos casos extremos y calcularemos promedio:
  1. El último nivel del árbol está lleno.
  2. El último nivel del árbol tiene una hoja.
- ¿Por qué sí podemos calcular promedio?
  - La longitud del recorrido de todos los nodos del mismo nivel es igual.

Análisis de Heapsort

1. El último nivel está lleno.
   - N = 3, 7, 15, 31, 63, … *[N = 1 es un caso trivial, f(1) = 0]*
   - # de niveles de un árbol binario de N elementos: $\lfloor \mathrm{Log}_2(N) \rfloor + 1$

N	f(N)
3	2(1)
7	4(2) + f(3)
15	8(3) + f(7)
31	16(4) + f(15)
63	32(5) + f(31)
N	**½(N + 1) (lg(N + 1) – 1) + f(½ N)**



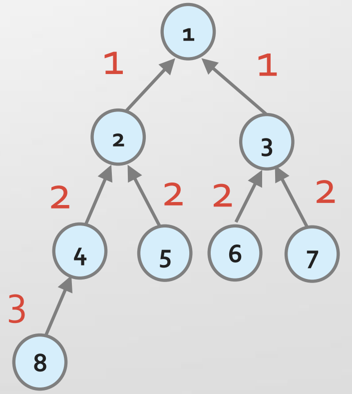
Análisis de Heapsort

N	f(N)	f(N) / N	Log (N + 1) – 2
3	2	0.66	0
7	10	1.42	1
15	34	2.26	2
31	98	3.16	3
63	258	4.09	4
127	642	5.05	5
255	1,538	6.03	6
511	3,586	7.01	7
N	f(N) ≈ N (lg(N + 1) – 2)		

Análisis de Heapsort

2. El último nivel tiene una hoja.
   - N = 2, 4, 8, 16, 32, 64, …
   - Estará en función del caso anterior.

N	f(N)
2	1
4	f(3) + 2
8	f(7) + 3
16	f(15) + 4
32	f(31) + 5
N	**f(N – 1) + lg(N)**



Análisis de Heapsort

N	f(N)	f(N) / N	Log(N) – 2
2	1	0.5	-1
4	4	1	0
8	13	1.62	1
16	38	2.38	2
32	103	3.21	3
64	264	4.12	4
128	649	5.07	5
256	1546	6.04	6
512	3595	7.02	7
N	f(N) ≈ N (lg(N) – 2)		

Análisis de Heapsort

- Falta calcular el promedio de los dos resultados obtenidos, y luego multiplicar por dos debido a los dos pasos del algoritmo; por tanto, sólo hay que sumar los dos resultados:
  1. $N(\lg(N+1)-2)$
  2. $N(\lg N-2)$
  - $N(\lg(N+1)+\lg N-4) \approx \mathbf{2N(\lg N-4)} \in O(N \lg N)$
    - Tienden a ser iguales conforme N crece.
    - Esta ecuación representa el promedio de los peores casos.
    - Nota: este análisis no contempla comparaciones entre los hijos.
- Comprobar de manera práctica que un arreglo aleatorio efectúa menos del 60% de comparaciones, uno ordenado efectúa poco más del 100%, y uno invertido es el mejor caso de los 3 ($N \geq 10^6$).

Radixsort

- Basa su funcionamiento en partir cada elemento de la lista en componentes atómicos.
  - Para números enteros positivos, lo divide en dígitos.
  - Para cadenas de texto, lo divide en caracteres.
- Construye una cola por cada valor diferente que puede tener un componente.
  - Para enteros, construye 10 colas.

Radixsort

- En una primera pasada, inserta cada elemento del arreglo en la cola que corresponde a su dígito menos significativo.
- En las pasadas siguientes, saca cada elemento de la cada cola y lo inserta en la cola que corresponde al siguiente dígito menos significativo.
    - Cada cola quedará ordenado de acuerdo al dígito anterior.
- Regresa el contenido de las colas al arreglo original.

Radixsort

1. Construir un arreglo/lista de 10 colas (C++: List, Java: LinkedList).
2. Copiar cada elemento del arreglo a la cola que corresponda al dígito menos significativo del elemento.
3. Registrar el tamaño actual de cada cola.

Lista = {5, 67, 26, 58, 20, 34, 25, 31, 19, 9, 24, 17}

$Cola_0$ = {20}

$Cola_1$ = {}

$Cola_2$ ={}

$Cola_3$ ={}

$Cola_4$ ={34}

$Cola_5$ = {5}

$Cola_6$ = {26}

$Cola_7$ = {67}

$Cola_8$ = {58}

$Cola_9$ = {}

Radixsort

- Calcular el número de dígitos D que tiene el elemento más grande.
- Por cada dígito *d* en [2..D], hacer:
  - Por cada cola *c* en [0..9], hacer:
    - Obtener el tamaño T que tenía *c* en la <u>pasada anterior</u>
    - Desde t = 1 hasta T,
      - Sacar el elemento al frente.
      - Colocarlo al final de la cola c' que corresponda al dígito *d*.
  - Recalcular el tamaño de cada cola.

Radixsort

- $Cola_0$ = 20 ← {}
- $Cola_1$ = 31 ← {}
- $Cola_2$ = {} ← 20
- $Cola_3$ = {} ← 31 34
- $Cola_4$ = 34 ← {24}

$Cola_5$ = {5, 25}

$Cola_6$ = {26}

$Cola_7$ = {67, 17}

$Cola_8$ = {58}

$Cola_9$ = {19, 9}

Radixsort

- Copiar el contenido de las listas de regreso al arreglo original.
- Por cada cola *c* en [0..9], hacer:
  - Obtener el tamaño T de *c*
  - Desde t = 1 hasta T,
    - Sacar el elemento al frente.
    - Colocarlo en la posición siguiente del arreglo.

- $Cola_0$ = {5, 9}
- $Cola_1$ = {10, 16, 17, 19}
- $Cola_2$ ={20, 24, 25, 26}
- $Cola_3$ ={31, 34}
- $Cola_4$ ={}

$Cola_5$ = {52, 58}
$Cola_6$ = {67}
$Cola_7$ = {}
$Cola_8$ = {}
$Cola_9$ = {}

- Lista = {5, 9, 10, 16, 17, 19, 20, 24, 25, 26, 31, 34, 52, 58, 67}

Ordenamiento por Conteo

- Veamos un caso especial:
  - Ordenar una lista de N enteros diferentes con valores de 0 a $N - 1$:
  - 4, 3, 5, 1, 6, 0, 2 $\Rightarrow$ 0, 1, 2, 3, 4, 5, 6
  - ¿Se puede lograr con una complejidad temporal menor a N lg N?
  - El valor estará en función de la posición: O(N).
- Otro caso especial (más interesante):
  - Ordenar una lista de N enteros con valores de 0 a $M - 1$, $M < N$:
  - 3, 1, 3, 1, 2, 1, 0 $\Rightarrow$ 0, 1, 1, 1, 2, 3, 3

Ordenamiento por Conteo

- La lista a ordenar es: 3, 1, 3, 1, 2, 1, 0

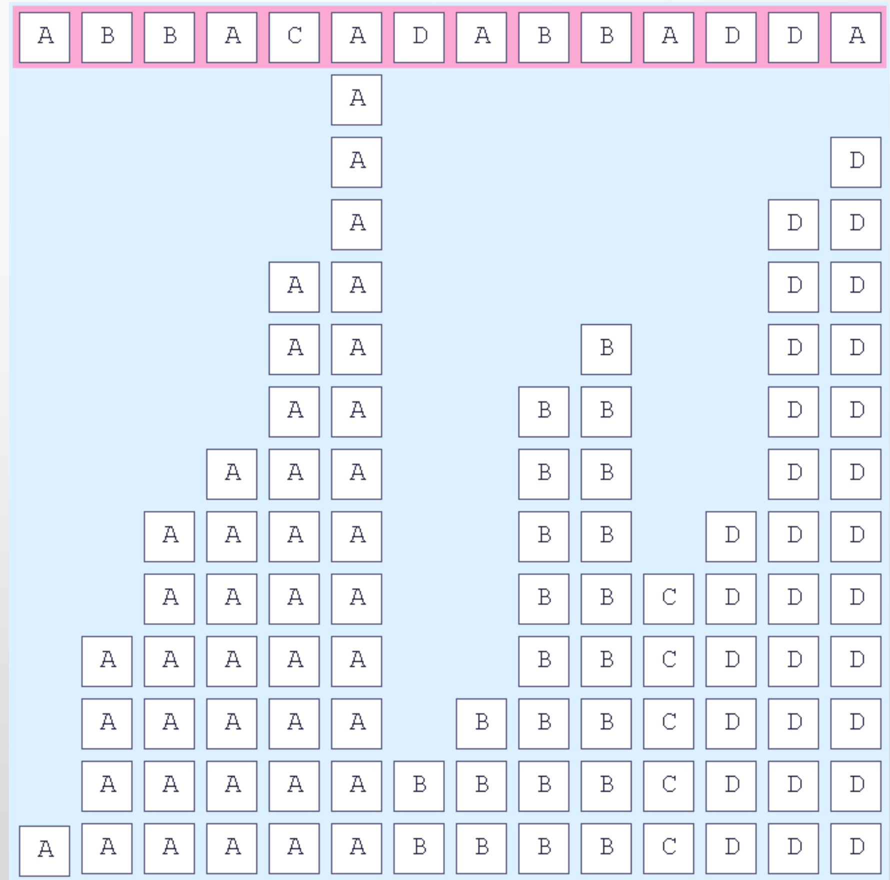
```
           0  1  2  3
```

1. Contar ocurrencias de cada valor: `Conteos = {1, 3, 1, 2}`
2. Acumular los conteos: `Conteos = {1, 4, 5, 7}`
3. Proceder de derecha a izquierda y escribir en una nueva lista:

```
[0] Conteos = {0, 4, 5, 7}. Lista' = {0, , , , , , }.
[1] Conteos = {0, 3, 5, 7}. Lista' = {0, , ,1, , , }.
[2] Conteos = {0, 3, 4, 7}. Lista' = {0, , ,1,2, , }.
[1] Conteos = {0, 2, 4, 7}. Lista' = {0, ,1,1,2, , }.
[3] Conteos = {0, 2, 4, 6}. Lista' = {0, ,1,1,2, ,3}.
[1] Conteos = {0, 1, 4, 6}. Lista' = {0,1,1,1,2, ,3}.
[3] Conteos = {0, 1, 4, 5}. Lista' = {0,1,1,1,2,3,3}.
```

Ordenamiento por Conteo

- ¿Qué complejidad tiene el algoritmo?
  - Temporal
  - Espacial
- Este algoritmo funciona cuando las claves son números enteros
- ¿Y si fueran números reales o letras?
  - Mapear los valores a índices del arreglo en tiempo constante



Tarea

- ¿Qué quisieran hacer como proyecto de obtención de grado para el cual esta materia pueda ayudar?
  - Desde el punto de vista de las técnicas que se verán o líneas de investigación de interés.
  - O desde el punto de vista de la aplicación o problema que se desea resolver.