Analysis and Design of Algorithms

Session 3.

Maestría en Sistemas Computacionales.

Luis Fernando Gutiérrez Preciado.

What topics will we cover today?

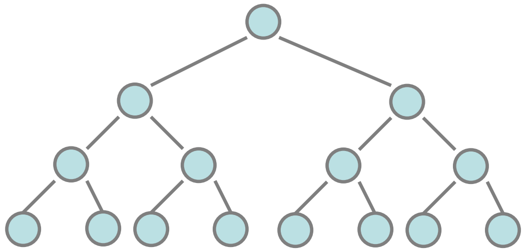
- Design and Analysis of Sorting Algorithms:
  - with quasi-linear complexity
- Which ones are they?
  - Shell (already covered)
  - Heapsort
  - Radix
- Counting Sort: linear complexity

HeapSort

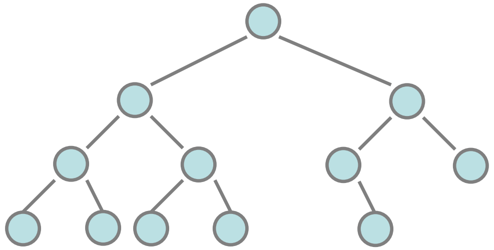
- Also called heap sort.
- Its time complexity is N log N in the best, worst, and average cases.
- Quicksort is often faster, but HeapSort performs better in critical cases.
- The first step of the algorithm involves building a heap from the array.
- The second step (which is iterative) involves removing the largest element and replacing it with the one placed at the end of the heap.

What is a heap?

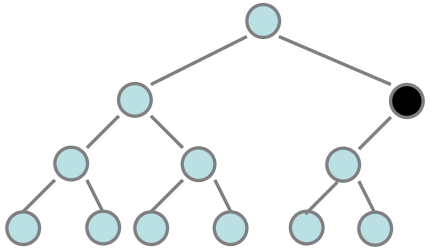
- It is a binary tree with these characteristics:
  1. Each node has a comparable value such that **no node has a value larger than its parent's**.
  2. It is **balanced**: each node has two children, except those on the last two levels.
  3. It is **left-aligned**: if a node has only one child, it must be the left child.



balanced

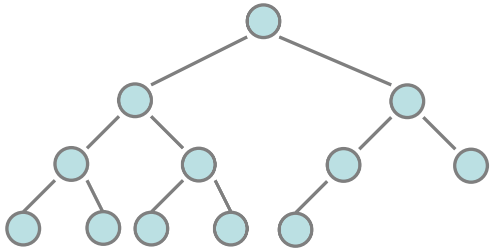


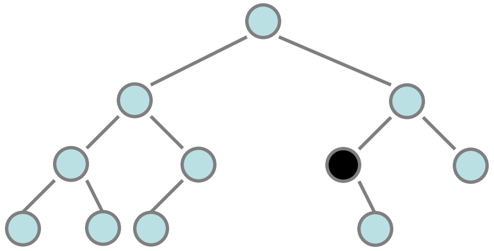
balanced

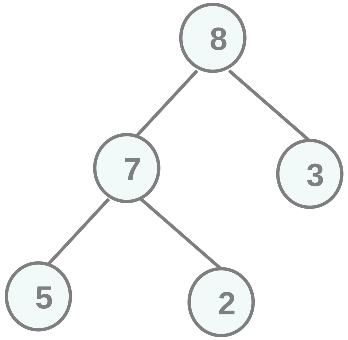


Unbalanced

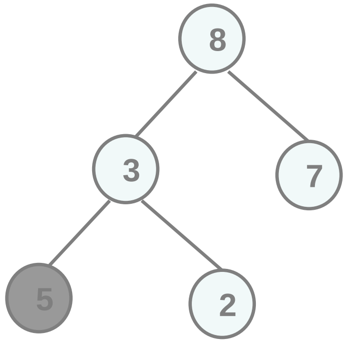
What is a heap?


Left-aligned


Not left-aligned



Satisfies property 1

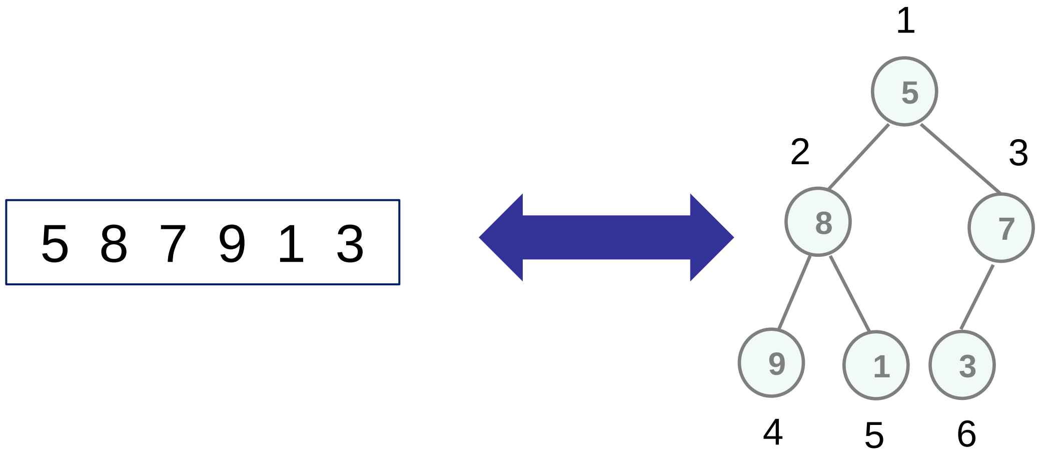


Does not satisfy property 1

Why with heaps?

1. Complying with property one will facilitate quasi-linear time sorting.
2. Complying with properties 2 and 3 will allow us to treat the received array as a tree (<u>without creating an explicit binary tree</u>).

- Correspondence between array and binary tree:
  - Note that it does not satisfy property 1.



Heap-Sort

- The first significant step of Heap-Sort involves transforming the input array into a max-heap, to satisfy property 1.
- Strategy: Convert each non-unitary subtree into a max-heap, from bottom to top.
  - The unitary subtrees are the leaves of the original tree; they trivially satisfy the max-heap properties.
- What is the position of the last non-unitary subtree?

Heap-Sort

- Convert the input array into a max-heap.
- Strategy: Convert each non-unitary subtree into a max-heap, from bottom to top.
  - The third argument of Max-Heapify indicates how far the max-heap extends.

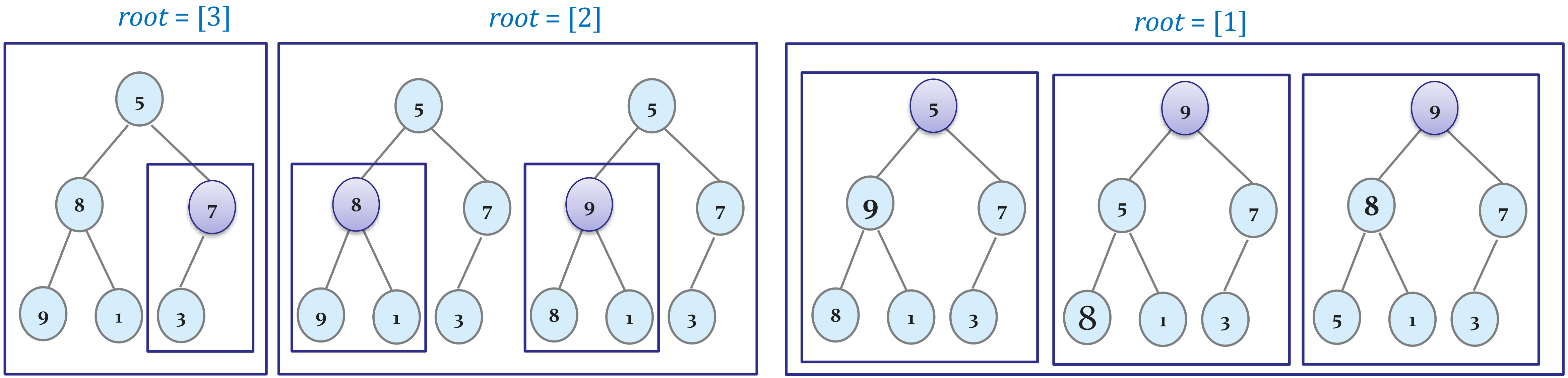
Build-Max-Heap(A):

1. For $root \leftarrow \lfloor A.length/2 \rfloor$ downto 1
    2. Max-Heapify($A, root, A.length$)

Heap-Sort

- Convert the input array into a max-heap.

5 8 7 9 1 3 ⇨ 9 8 7 5 1 3



[Heap Sort Visualization (usfca.edu)]

Exercise: (individual and compare in a team)

Form the heap from an array with values from 1 to 15.

Heap-Sort

- **The second significant step of Heap-Sort**
- The first element of the array holds the largest element. Since we are sorting from smallest to largest, we exchange it with the max-heap's last element (initially the array's last element).
  - This element is now in its final position and will not be revisited.
  - The max-heap now ends at the previous position.
- By placing the last element at the beginning, property one is lost, as this element was one of the smallest. Therefore, we convert the subarray, excluding the last element, into a max-heap.
- Return to the initial step as long as the size of the max-heap is > 0.

Heap-Sort

- Algoritmo Heap-Sort completo:

  HeapSort(A):

  1. Build-Max-Heap(A)
  2. For $heapSize$ ← A.length downto 2
     3. Swap(A, 1, $heapSize$)
     4. Max-Heapify(A, 1, $heapSize$ – 1)

➢ Nota: The algorithms shown here assume that the range of positions in an array is [1 .. N].

Exercise: (individual and compare in a team)

2. From the result of the previous exercise (heap of values from 1 to 15), now sort **the 4 largest** values.

HeapSort

- How do we convert a subtree into a max-heap?
- Strategy:
  - Obtain the largest child of the root of the subtree.
  - If this child is larger than the root, swap them and repeat this process recursively, now with the subtree rooted at the larger child.
- How do we obtain the position of each child?
- To consider:
  - Before extracting the value of each child, check if its position leaves the current range of the max-heap. If yes, discard that child.

HeapSort

- How do we convert a subtree into a max-heap?
- Max-Heapify(A, *root*, *heapSize*):
    1. Let *left* = Left(*root*)
    2. Let *right* = Right(*root*)
    3. Let *max* = *root*
    4. If $left \leq heapSize$ and $A_{left} > A_{max}$ then *max* = *left*
    5. If $right \leq heapSize$ and $A_{right} > A_{max}$ then *max* = *right*
    6. If $max \neq root$ then
        7. Swap(A, *root*, *max*)
        8. Max-Heapify(A, *max*, *heapSize*)

HeapSort Analysis

- Complete HeapSort algorithm:

HeapSort(A):

1. Build-Max-Heap(A)
2. For $heapSize$ ← A.length downto 2
    3. Swap(A, 1, $heapSize$)
    4. Max-Heapify(A, 1, $heapSize$ – 1)

$$T_{\text{HeapSort}}(N) = T_{\text{BuildMaxHeap}}(\text{N}) + (\text{N} - 1)(T_{\text{MaxHeapify}}(\text{N}))$$

HeapSort Analysis

Max-Heapify(A, $root$, $heapSize$):

1. Let *left* = Left(A)
2. …
6. If $max \neq root$ $then$
    7. Swap(A, $root$, max)
    8. Max-Heapify(A, max, $heapSize$)

$\theta(1)$

HeapSort Analysis

Max-Heapify$(A, root, heapSize)$:

1. Let $left$ = Left(A)
2. ...
6. If $max \neq root$ $then$
   7. Swap$(A, root, max)$
   8. Max-Heapify$(A, max, heapSize)$

$\theta(1)$

Worst case → $max \neq root$ is always satisfied

How many recursive calls are made? The ones needed to reach the first leaf: $llamadas \leq h$, where h is the height of the heap

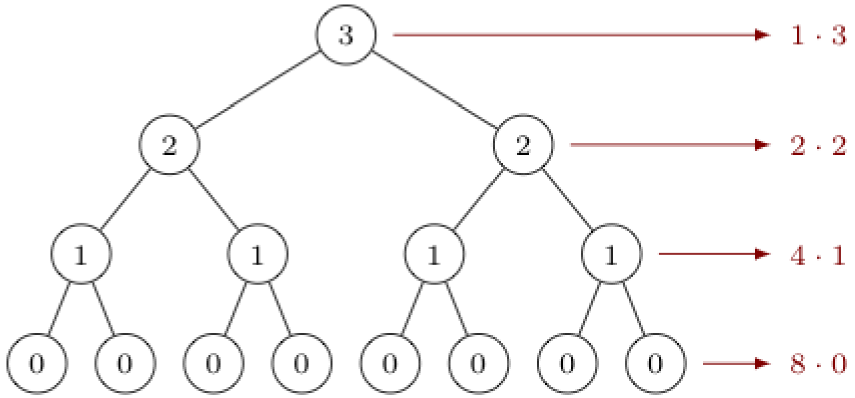
A heap with h levels has no more than N = 1 + 2 + 4 + … 2^{h-1} = $2^h - 1$ elements.

.: A heap with N elements has $h = \lg N$ levels. Thus: $T_{MaxHeapify}(N) = O(\lg N)$

HeapSort Análisis

Build-Max-Heap(A):

1. For $root \leftarrow \lfloor A.length/2 \rfloor$ downto 1
    2. Max-Heapify(A, *root*, *A.length*)



$$T(n) = 2^h \cdot 0 + 2^{h-1} \cdot 1 + 2^{h-2} \cdot 2 + 2^{h-3} \cdot 3 + \cdots + 2^{h-h} \cdot h$$

$$= \sum_{i=0}^{h} 2^{h-i} i \qquad\qquad \sum_{i=0}^{\infty} i x^i = \frac{x}{(1-x)^2}$$

$$= 2^h \sum_{i=0}^{h} \frac{i}{2^i} = \frac{1/2}{(1-(1/2))^2} = \frac{1/2}{1/4} = 2 \qquad T(n) < 2^h \cdot 2 = 2^{h+1}$$

Análisis de Heap-Sort

- Complete HeapSort algorithm:

HeapSort(A):

1. Build-Max-Heap(A) $O(N)$
2. For *heapSize* ← A.length downto 2 $N-1$
    3. Swap(A, 1, *heapSize*) $O(1)$
    4. Max-Heapify(A, 1, *heapSize* – 1) $O(\lg N)$

$$T_{HeapSort}(N) = O(N) + (N - 1)(O(\lg N)] = \mathbf{O(N \lg N)}$$

RadixSort

- Sorting is based on processing the **digits** that compose each number individually.
- As many passes are made as the number of digits in the largest number.
  - Ten variable-sized lists are created.
  - In the first list, all those ending in 0 are stored; the second list contains those ending in 1, and so on...
  - In the next pass, ten new lists are created.
  - In the new first list, the numbers with 0 as their penultimate digit are stored, following the order they had in the 10 lists from the previous pass.

RadixSort

- List = {5, 67, 58, 34, 25, 31, 19, 20, 9, 24, 26, 17, 10, 16, 52}
  - $List_0$ = {20,10} $List_1$ = {31}
  - $List_2$ ={52} $List_3$ ={}
  - $List_4$ ={34, 24} $List_5$ = {5,25}
  - $List_6$ = {26, 16} $List_7$ = {67, 17}
  - $List_8$ = {58} $List_9$ = {19, 9}

  - $List_0$ = {05, 09} $List_1$ = {10, 16, 17, 19}
  - $List_2$ ={20, 24, 25, 26} $List_3$ ={31, 34}
  - $List_4$ ={} $List_5$ = {52, 58}
  - $List_6$ = {67} $List_7$ = {}
  - $List_8$ = {} $List_9$ = {}

http://cs.armstrong.edu/liang/animation/web/RadixSort.html

Análisis de RadixSort

- Let:
  - d be the digits
  - n be the input size
- $O(dn)$
  - b be the base (decimal b=10)
  - k be the maximum number in the input
  - How many digits can k have?
- $O(\mathrm{n}\log_b k)$

Counting Sort

- Let's consider a special case:
  - Sorting a list of N different integers with values from 0 to N – 1:
  - 4, 3, 5, 1, 6, 0, 2 ⇒ 0, 1, 2, 3, 4, 5, 6
    - Can this be achieved with a time complexity less than N lg N?
    - The value will depend on the position: O(N).
- Another special case (more interesting):
  - Sorting a list of N integers with values from 0 to M – 1, where M < N:
    3, 1, 3, 1, 2, 1, 0 ⇒ 0, 1, 1, 1, 2, 3, 3

Ordenamiento por Conteo

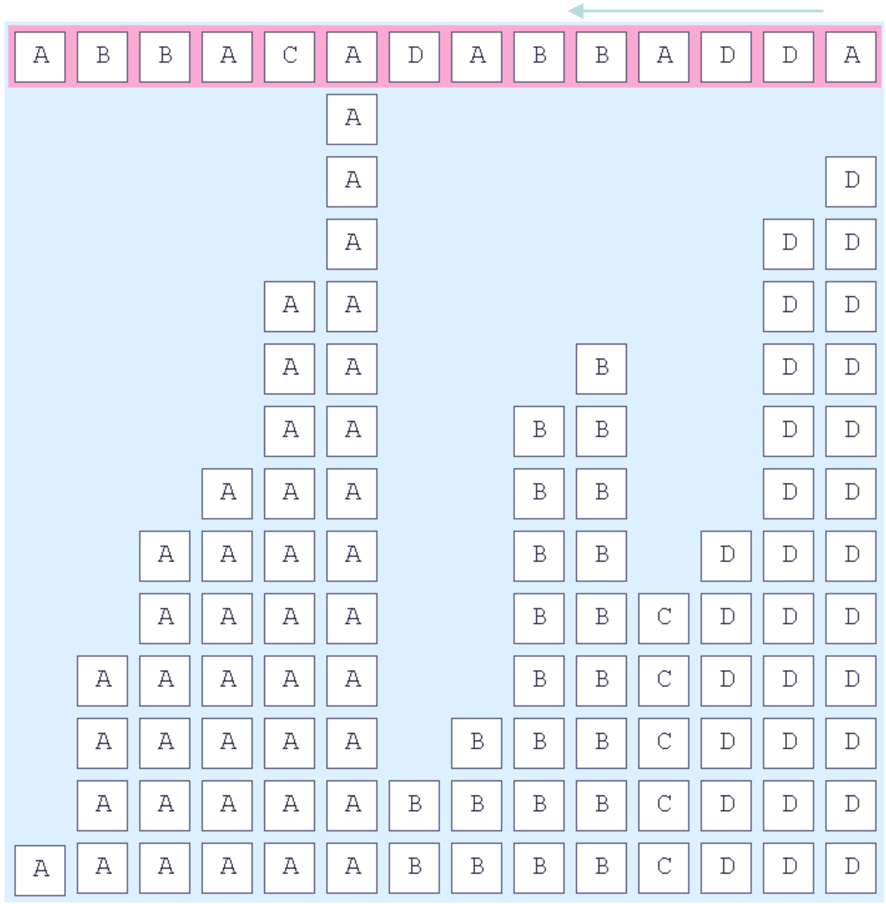
- The list to be sorted is: 3, 1, 3, 1, 2, 1, 0
  1. Count occurrences of each value:
  2. Accumulate the counts:
  3. Proceed from right to left and write to a new list:

```
                  #0 #1 #2 #3
Counts      = [1, 3, 1, 2]
Accumulated= [1, 4, 5, 7]
```

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

Ordenamiento por Conteo

- What complexity does the algorithm have?
  - Temporal
  - Spatial
- This algorithm works when the keys are integers.
- What if they were real numbers or letters?
  - Not applicable for real numbers.
  - We need to map the values to array indices in constant time.



Conclusions

- We understood, implemented, and analyzed the Heap Sort algorithm, a quasi-linear algorithm.
- We grasped the logic of the radix sort, another quasi-linear algorithm.
- We understood the logic of a linear sorting algorithm (counting sort).