Heap Sort

Kuan-Yu Chen (陳冠宇)

2019/03/20 @ TR-310-1, NTUST

Review

- Maximum-subarray Problem
  - The brute-force solution takes $\Omega(n^2)$ time
  - A transformation approach takes $\Theta(n^2)$ time
  - The divide-and-conquer method takes $\Theta(n \log_2 n)$ time
    - Faster than the brute-force method

- Matrix Multiplication
  - The brute-force solution takes $\Theta(\mathrm{n}^3)$ time
  - The divide-and-conquer method takes $\Theta(\mathrm{n}^3)$ time
  - Strassen's method takes $\Theta(n^{\log_2 7})$ time

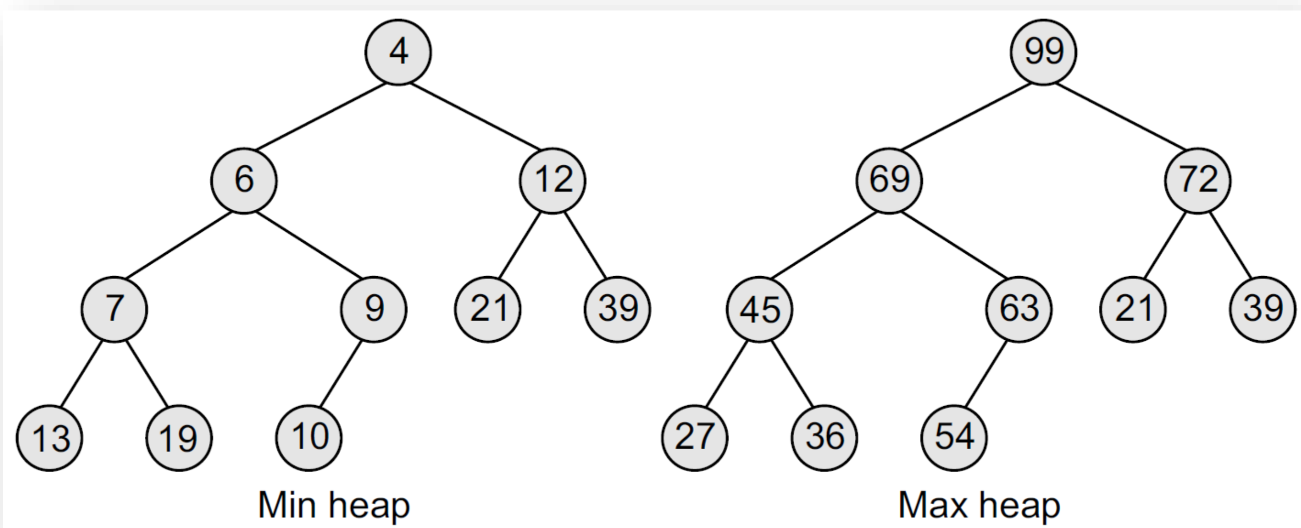
Binary Heap

- A **binary heap** is a complete binary tree in which every node satisfies the heap property
  - Min Heap

$$If\ B\ is\ a\ child\ of\ A, then\ key(B)\ \geq\ key(A)$$

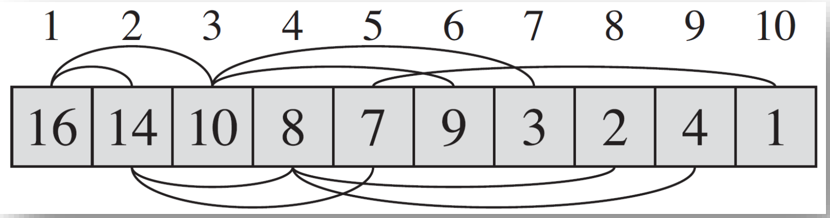
  - Max Heap

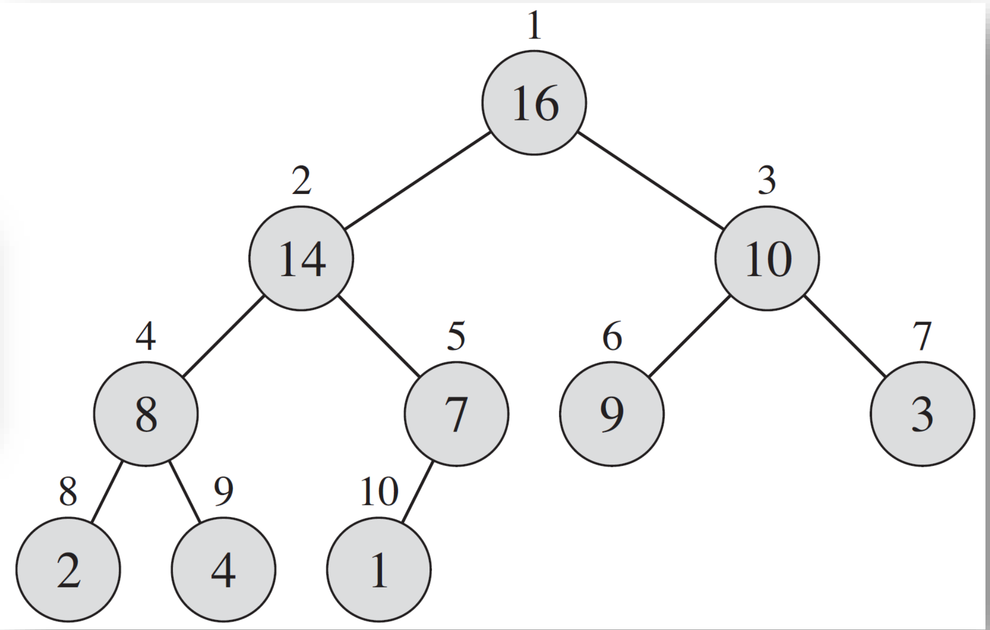
$$If\ B\ is\ a\ child\ of\ A, then\ key(A)\ \geq\ key(B)$$



Binary Heap

- Given the index i of a node, we can easily compute the indices of its parent, left child, and right child
  - Take max-heap for example
    - For a node in the max-heap with index i
    - Its parent is $\left\lfloor \frac{i}{2} \right\rfloor$
    - Its left child is $2i$
    - Its right child is $2i + 1$





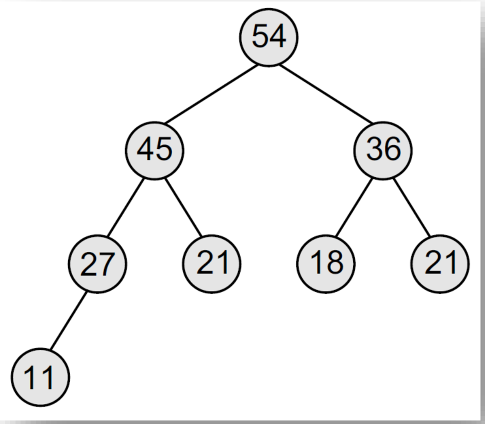
Heap – Insertion

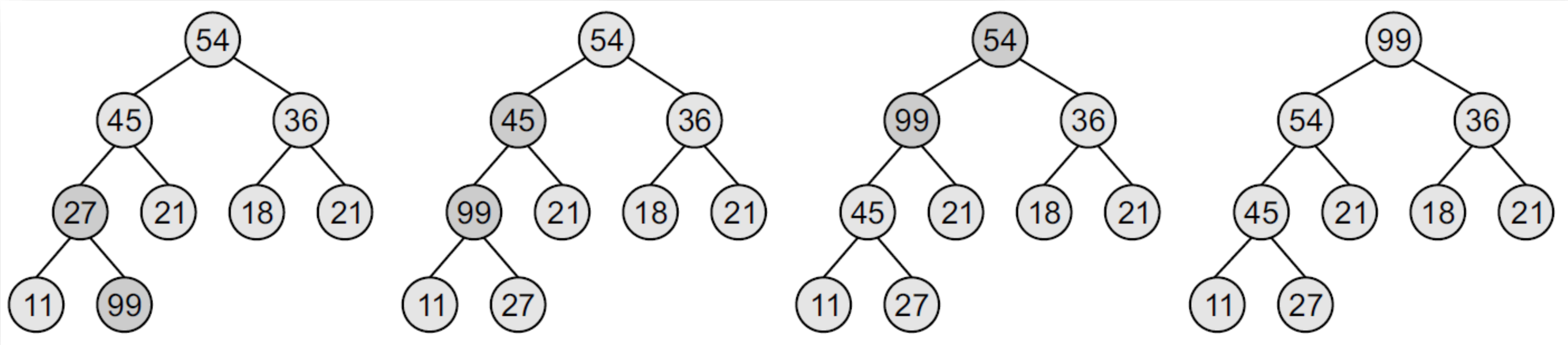
- Inserting a new value into the heap is done in the following two steps:
  - Consider a max heap H with n elements

1. Add the new value at the bottom of H
2. Let the new value rise to its appropriate place in H

Example

- Consider a max heap and insert 99 in it





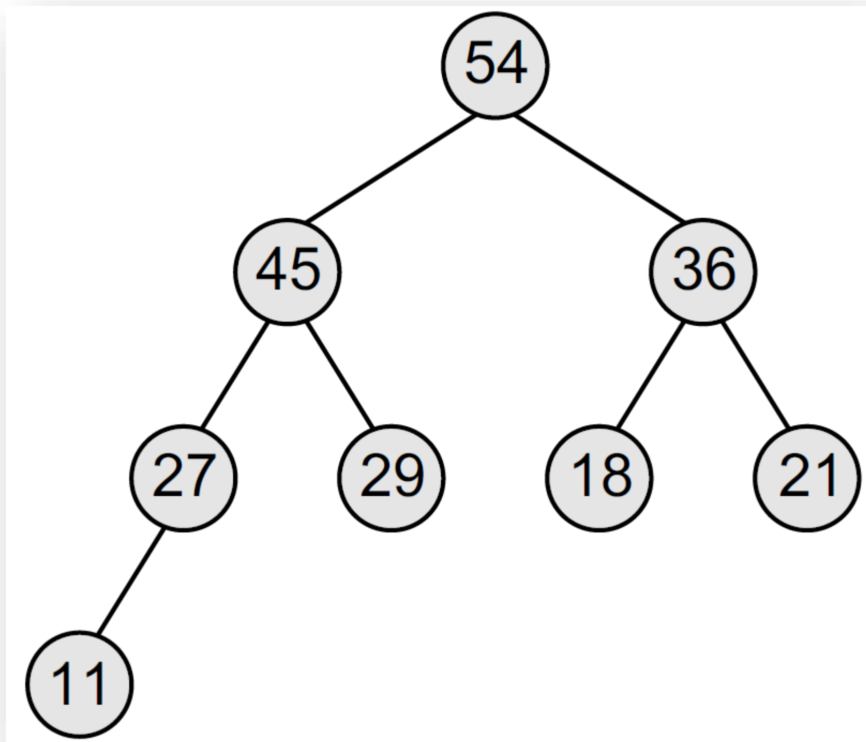
Heap – Deletion

- An element is always deleted from the root of the heap

- Consider a max heap H having n elements, deleting an element from the heap is done in the following three steps:
  1. Replace the root node's value with the last node's value
  2. Delete the last node
  3. Sink down the new root node's value so that H satisfies the heap property

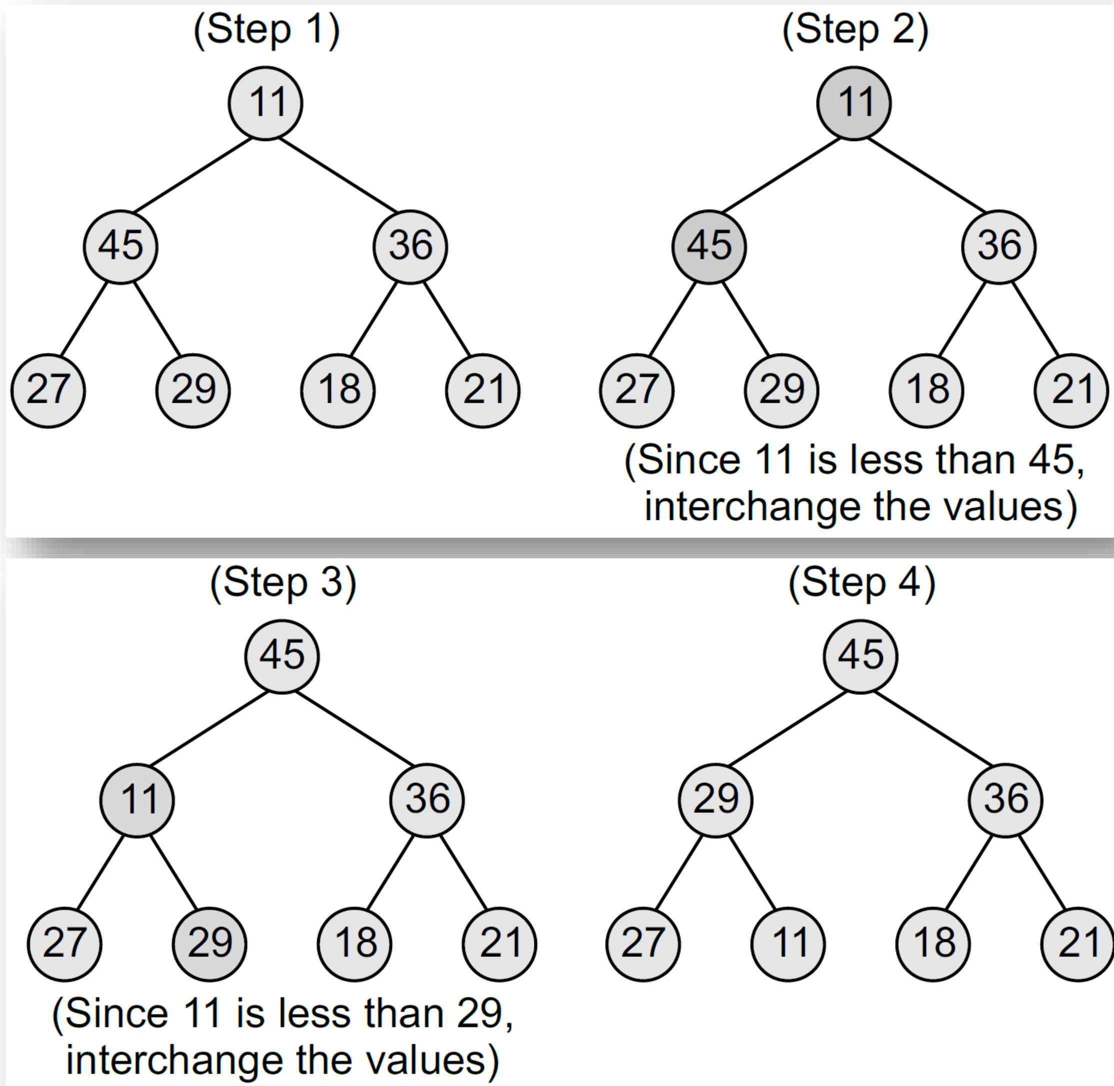
Example.

- Delete the root node's value from a given max heap H



Example..

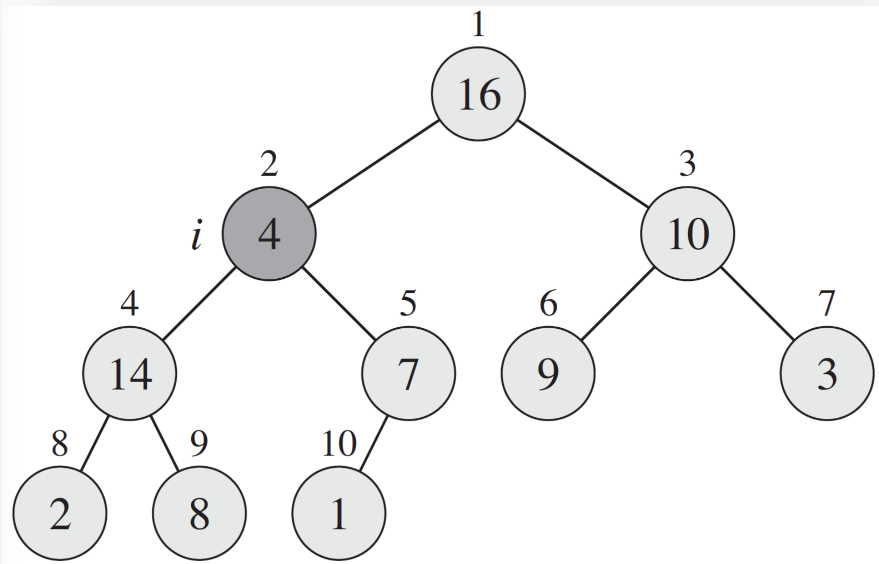
- Delete the root node's value from a given max heap H



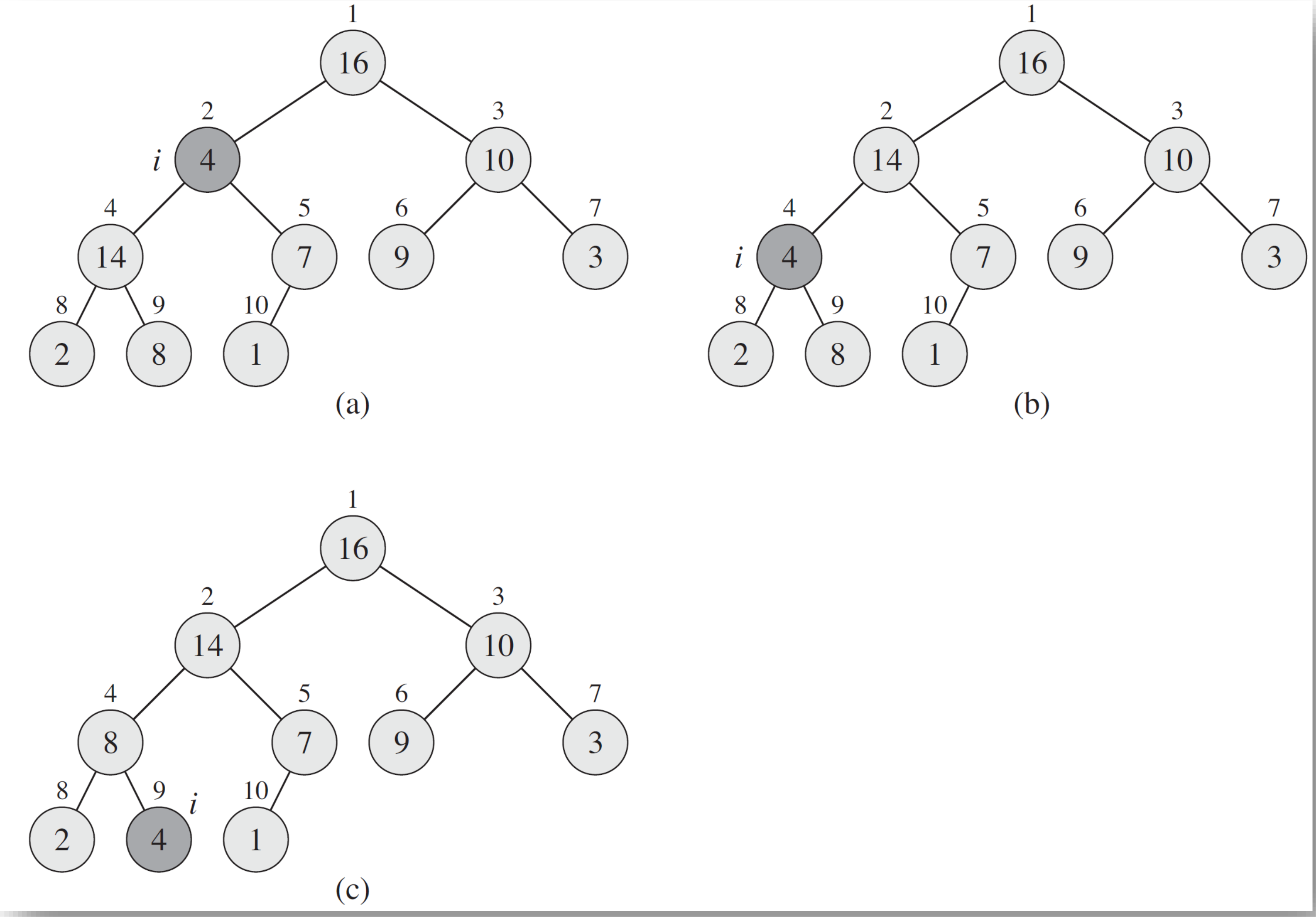
Max-Heapify.

- In order to maintain the max-heap property, we call the procedure MAX-HEAPIFY
  - Its inputs are an array A and an index i into the array
  - The node $A[i]$ has two children $LEFT(i)$ and $RIGHT(i)$
  - If $A[i]$ is smaller than its children, the procedure can make it correct

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```



Example

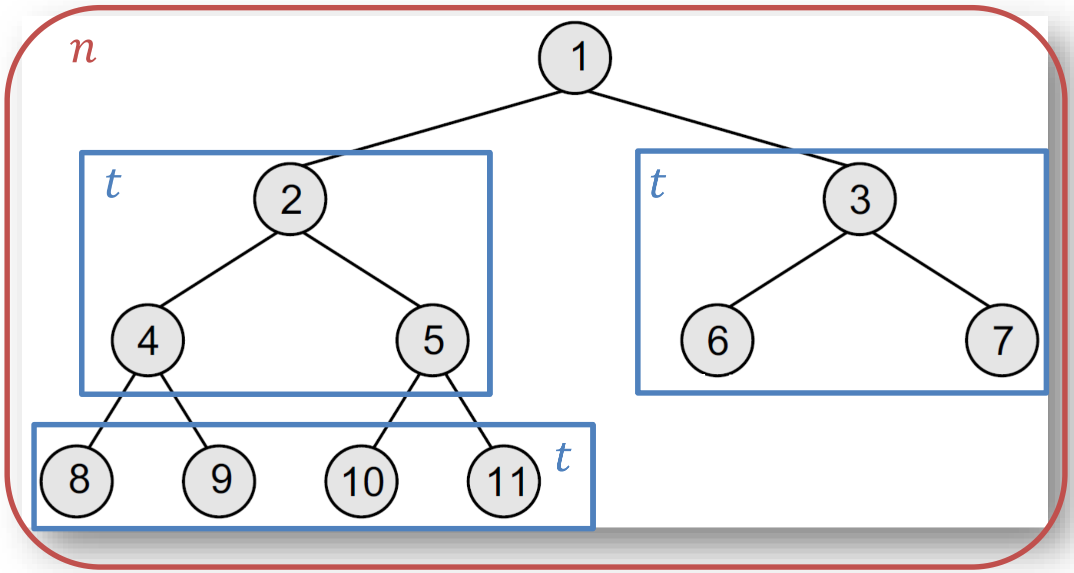


1
16
2
i
4
3
10
4
14
5
7
6
9
7
3
8
2
9
8
10
1
(a)
1
16
2
14
3
10
4
i
4
5
7
6
9
7
3
8
2
9
8
10
1
(b)
1
16
2
14
3
10
4
8
5
7
6
9
7
3
8
2
9
i
4
10
1
(c)

Max-Heapify..

- Analyze the MAX-HEAPIFY procedure
  - The running time of MAX-HEAPIFY on a subtree of size n, i.e., $T(n)$, rooted at a given node i is $\Theta(1) + T\left(\frac{2n}{3}\right)$
    - To fix up the relationships among the elements $A[i]$, $A[LEFT(i)]$ and $A[RIGHT(i)]$
    - To do recursive calls on the subtrees
  - By the master theorem, $T(n) \leq \Theta(1) + T\left(\frac{2n}{3}\right) = O(\log_2 n)$
    - We can characterize the running time of MAX-HEAPIFY on a node of height h as $O(h)$

Appendix



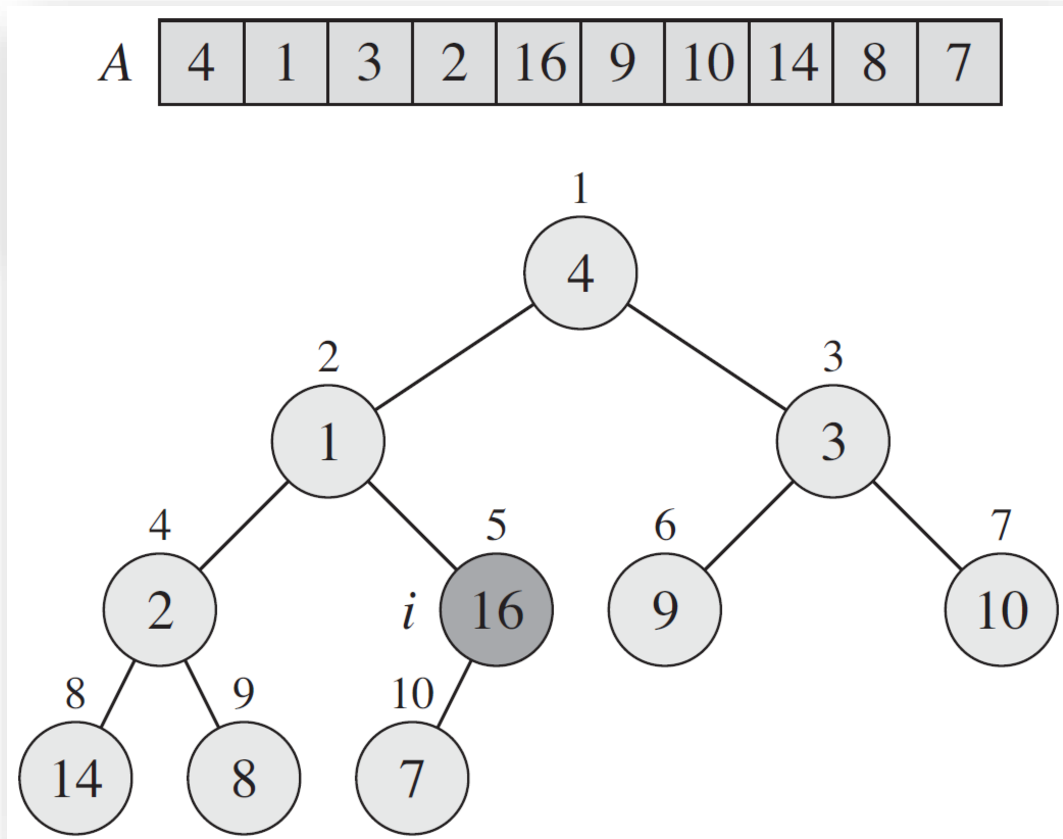
$$n \approx 3t$$

The children's subtrees each have size **at most** $= 2t = \dfrac{2n}{3}$

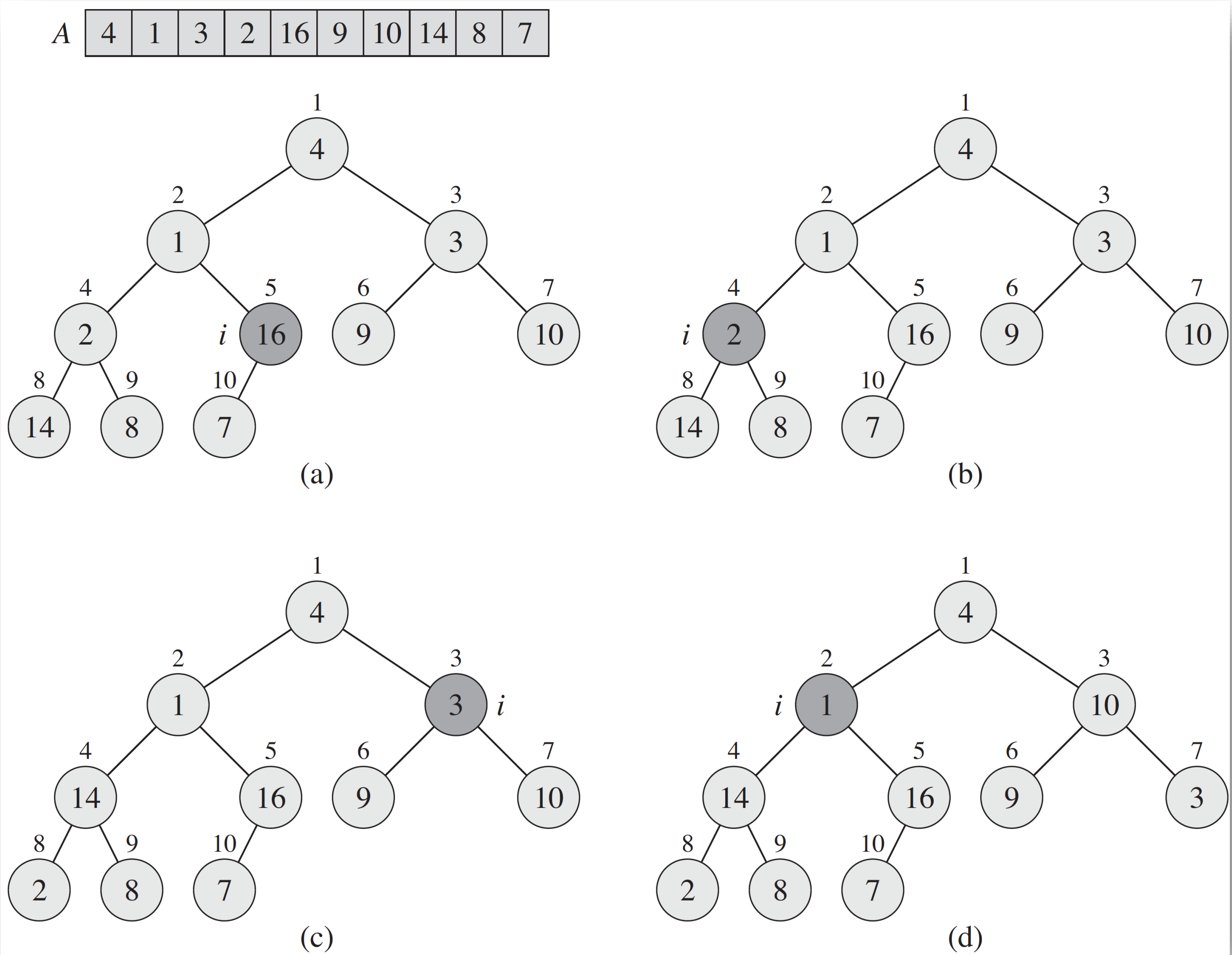
Build-Max-Heap.

- We can use the procedure MAX-HEAPIFY in a **bottom-up** manner to convert a tree into a max-heap

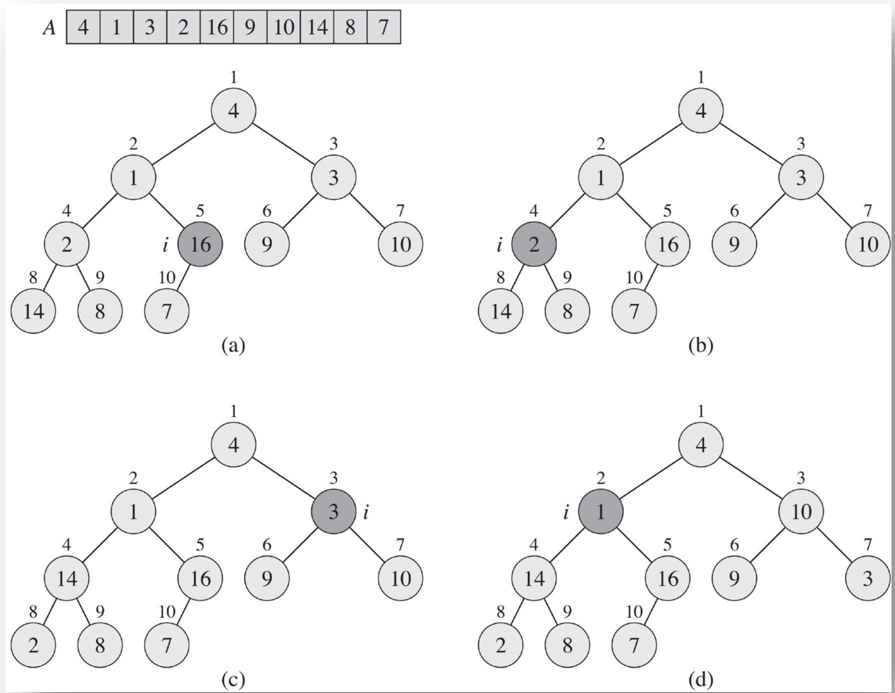
```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

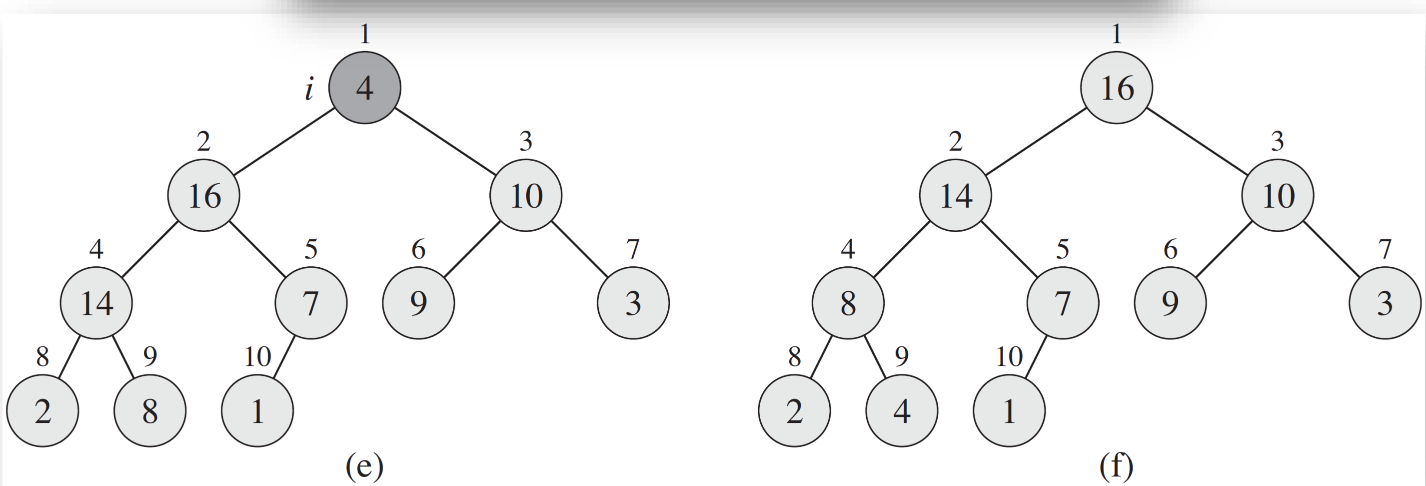


Example.



Example..





Build-Max-Heap..

- We can compute a simple upper bound on the running time of BUILD-MAX-HEAP
  - Each call to MAX-HEAPIFY costs $O(\log_2 n)$ time
  - BUILD-MAX-HEAP makes $O(n)$ such calls
  - Thus, the running time is $O(n \log_2 n)$
  - This upper bound, though correct, is not asymptotically tight!

- A tighter analysis relies on the properties that an n-element heap has height $\lfloor \log_2 n \rfloor$ and at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes of any height h

$n = 15$

$height = \lfloor \log_2 15 \rfloor = 3$

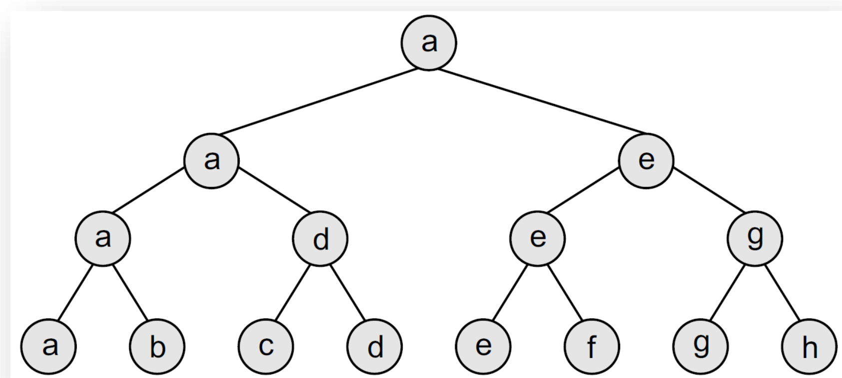
$h = 1, |node_{h=1}| = \left\lceil \frac{15}{2^2} \right\rceil = 4$

$h = 3$

$h = 2$

$h = 1$

$h = 0$



Build-Max-Heap…

- To put everything together
  - The time required by MAX-HEAPIFY when called on a node of height h is $O(h)$
  - Thus, the total cost of BUILD-MAX-HEAP as being bounded by

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil h \right) = O\left(\sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{nh}{2^h} \right) = O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} h \left(\frac{1}{2}\right)^h \right)$$

$$= O\left(n \sum_{h=0}^{\infty} h \left(\frac{1}{2}\right)^h \right) = O\left(n \frac{\frac{1}{2}}{\left(1 - \frac{1}{2}\right)^2} \right) = O(2n) = O(n)$$

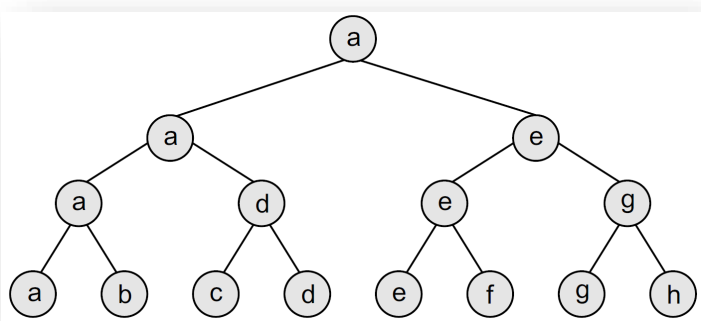
- A max-heap can be built from an unordered array in **linear time**

$h = 3$

$h = 2$

$h = 1$

$h = 0$



$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} = f(x)$$

$$f'(x) = \sum_{k=0}^{\infty} k x^{k-1} = \frac{1}{(1-x)^2}$$

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

HeapSort

- The heapsort algorithm starts by using BUILD-MAX-HEAP to build a max-heap on the input array A

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

  - The HEAPSORT procedure takes time $O(n \log_2 n)$
    - BUILD-MAXHEAP takes time $O(n)$
    - Each of the $n-1$ calls to MAX-HEAPIFY takes time $O(\log_2 n)$
    - $O(n) + (n-1) \times O(\log_2 n) \leq O(n \log_2 n)$

Questions?

kychen@mail.ntust.edu.tw