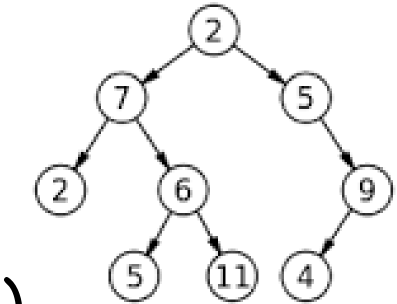
Binary Heap

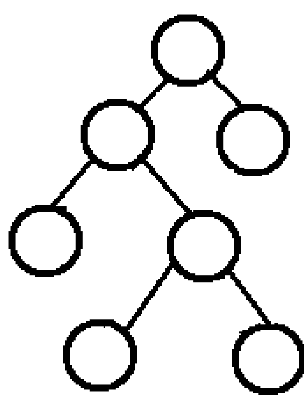
- Binary Tree
- Strictly Binary Tree (Full Binary Tree)
- Complete Binary Tree

Binary Heap

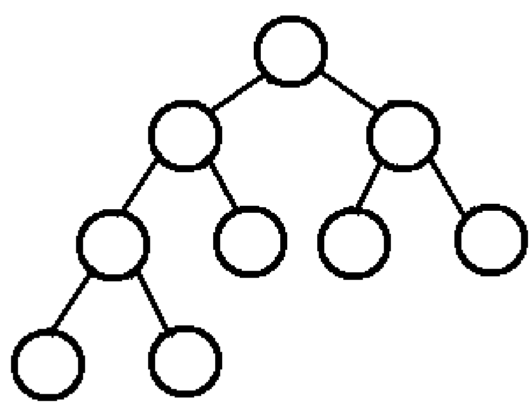
- Binary Tree
  - Any node can have at most 2 children



- Strictly Binary Tree (Full Binary Tree)
  - A binary tree is a full binary tree if every node has 0 or 2 children.
- Complete Binary Tree
  - a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.



full tree



complete tree

Binary Heap

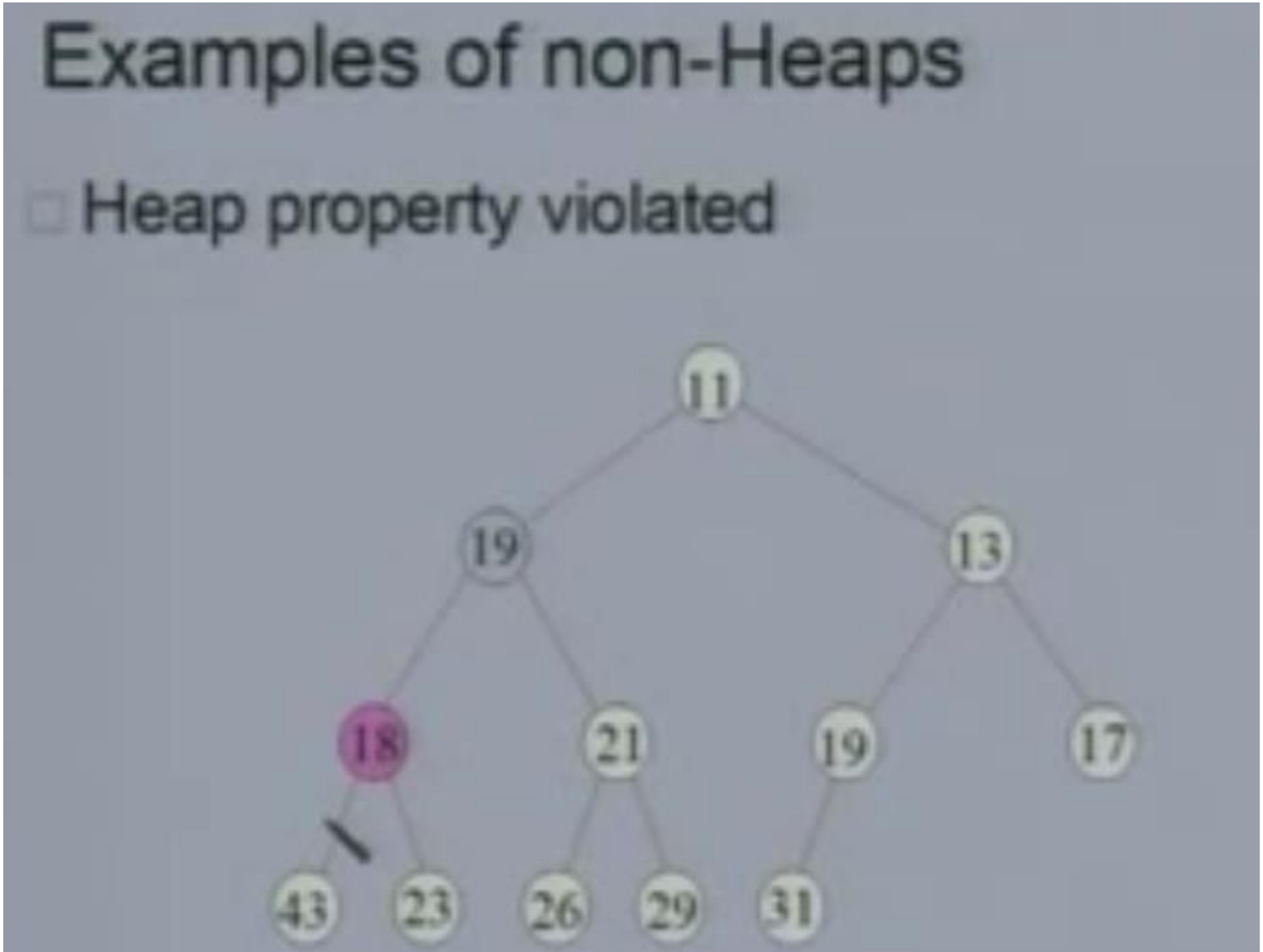
(Binary) Heaps

A binary tree that stores priorities (or priority-element) pairs at nodes

Structural property: All levels except last are full. Last level is left-filled.

Heap property: Priority of node is at least as large as that of its parent.

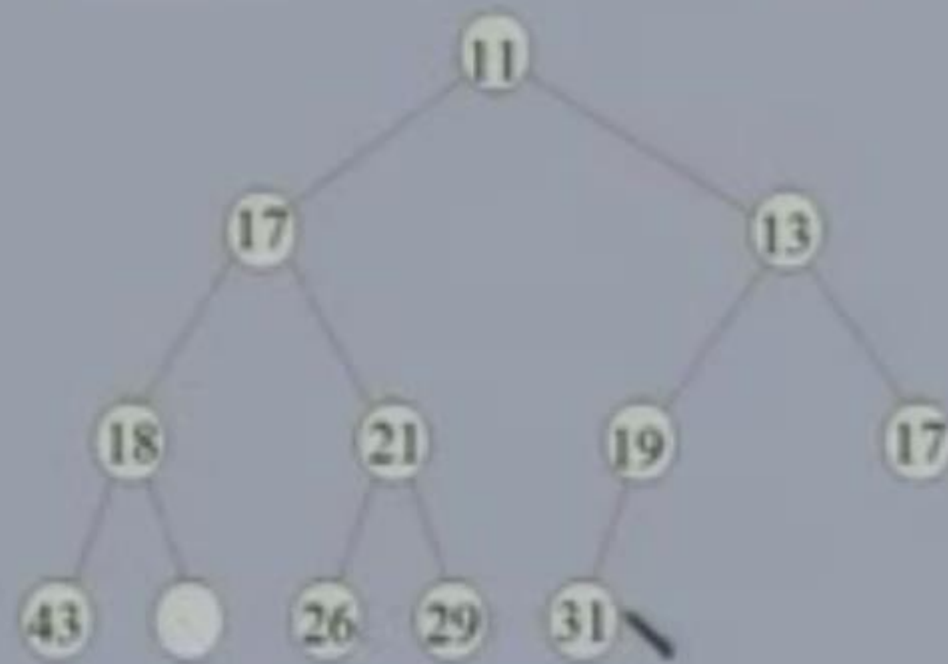
Binary Heap


Examples of non-Heaps
Heap property violated
11
19
13
18
21
19
17
43
23
26
29
31

Binary Heap

Example of non-heap

- Last level not left-filled



Binary Heap

Finding the minimum element

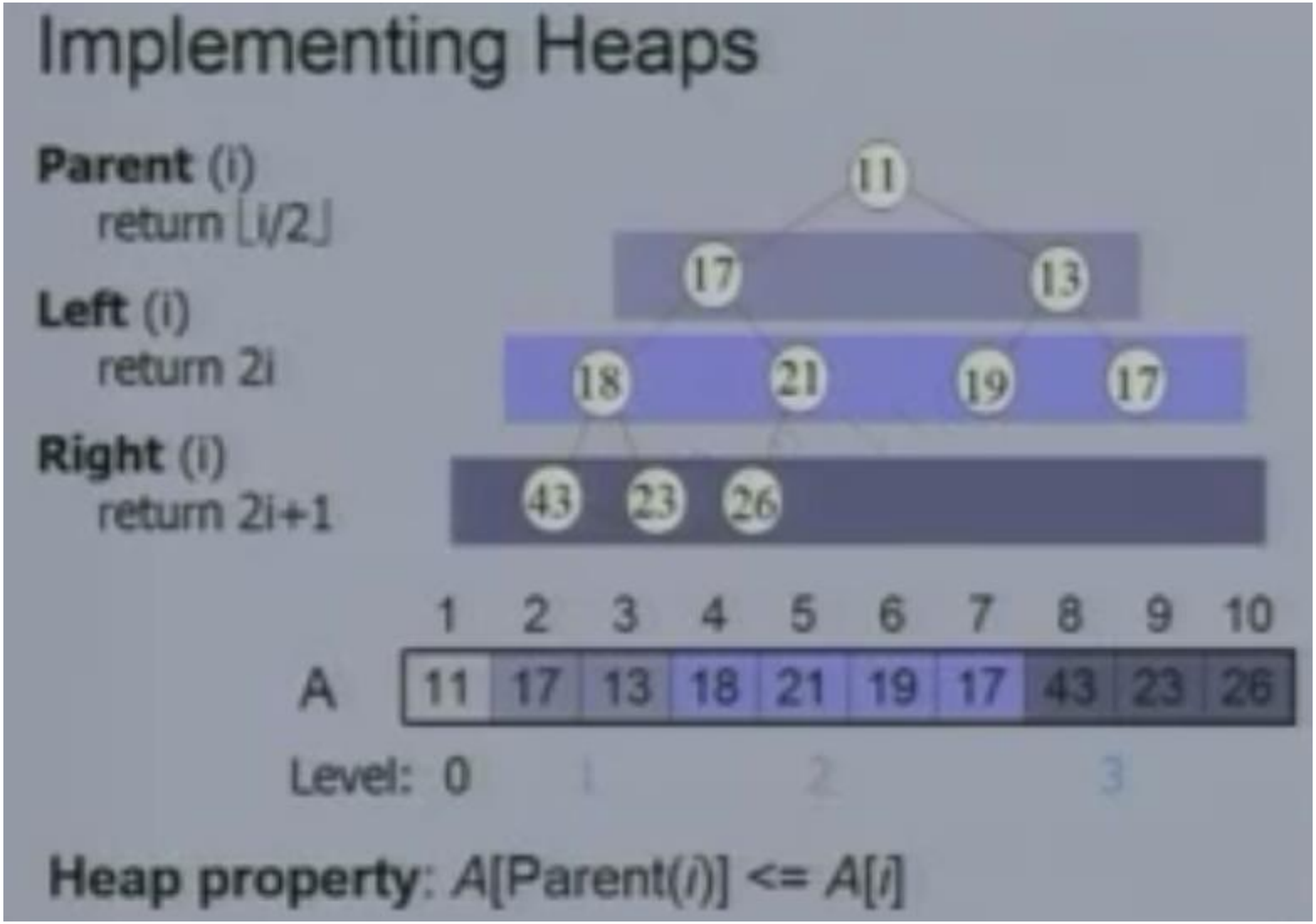
- The element with smallest priority always sits at the root of the heap.
- This is because if it was elsewhere, it would have a parent with larger priority and this would violate the heap property.
- Hence minimum() can be done in O(1) time.

Binary Heap

Height of a heap

- Suppose a heap of n nodes has height h.
- Recall: complete binary tree of height h has 2^{h+1}-1 nodes.
- Hence $2^h-1 < n <= 2^{h+1}-1$.
- $\cancel{n = \lfloor \log_2 h \rfloor}$ ← Error, $h = \lfloor log_2 n \rfloor$

Binary Heap

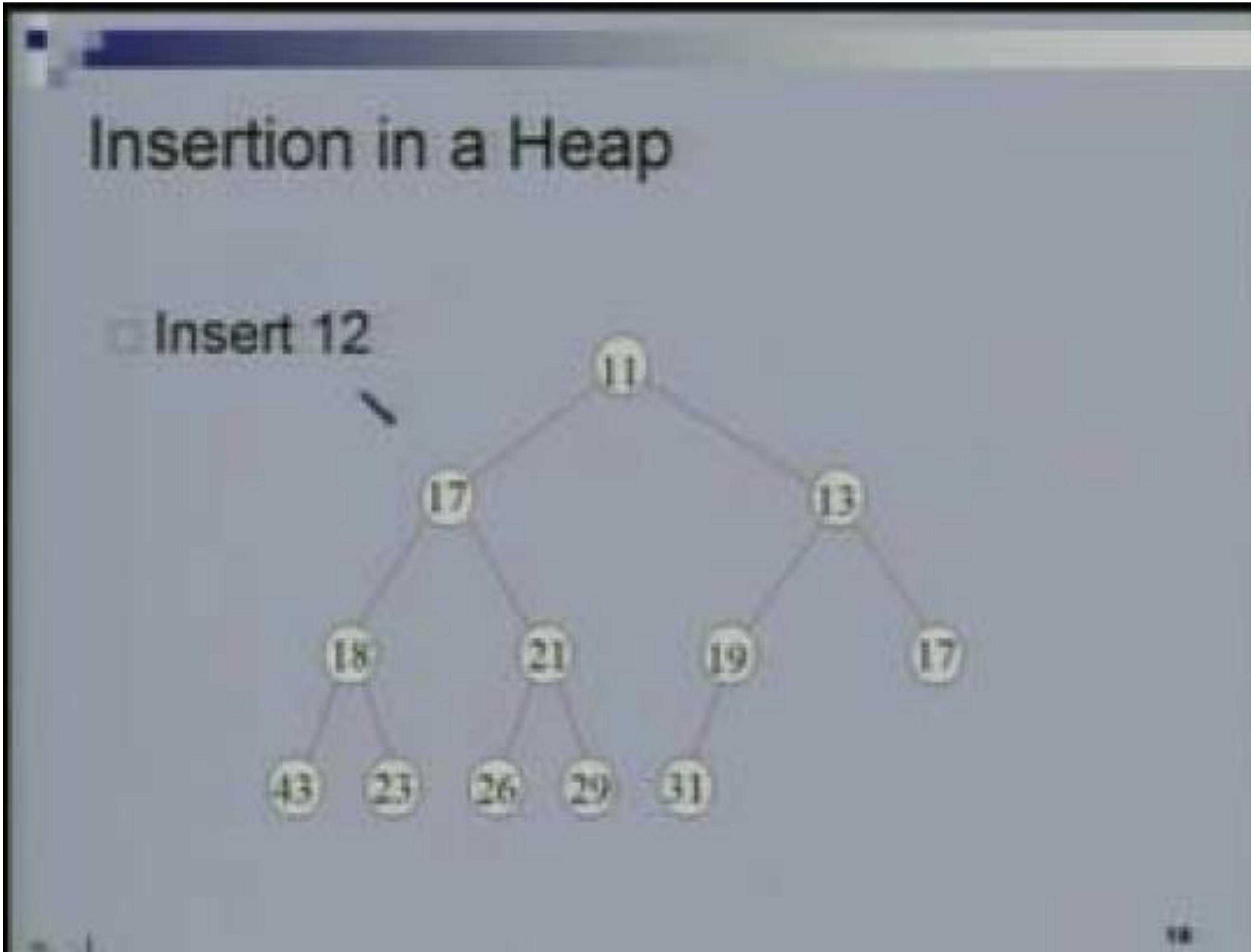

Implementing Heaps
Parent (i)
return ⌊i/2⌋
Left (i)
return 2i
Right (i)
return 2i+1
11
17
13
18
21
19
17
43
23
26
1 2 3 4 5 6 7 8 9 10
A
11 17 13 18 21 19 17 43 23 26
Level: 0 1 2 3
Heap property: A[Parent(i)] <= A[i]

Binary Heap

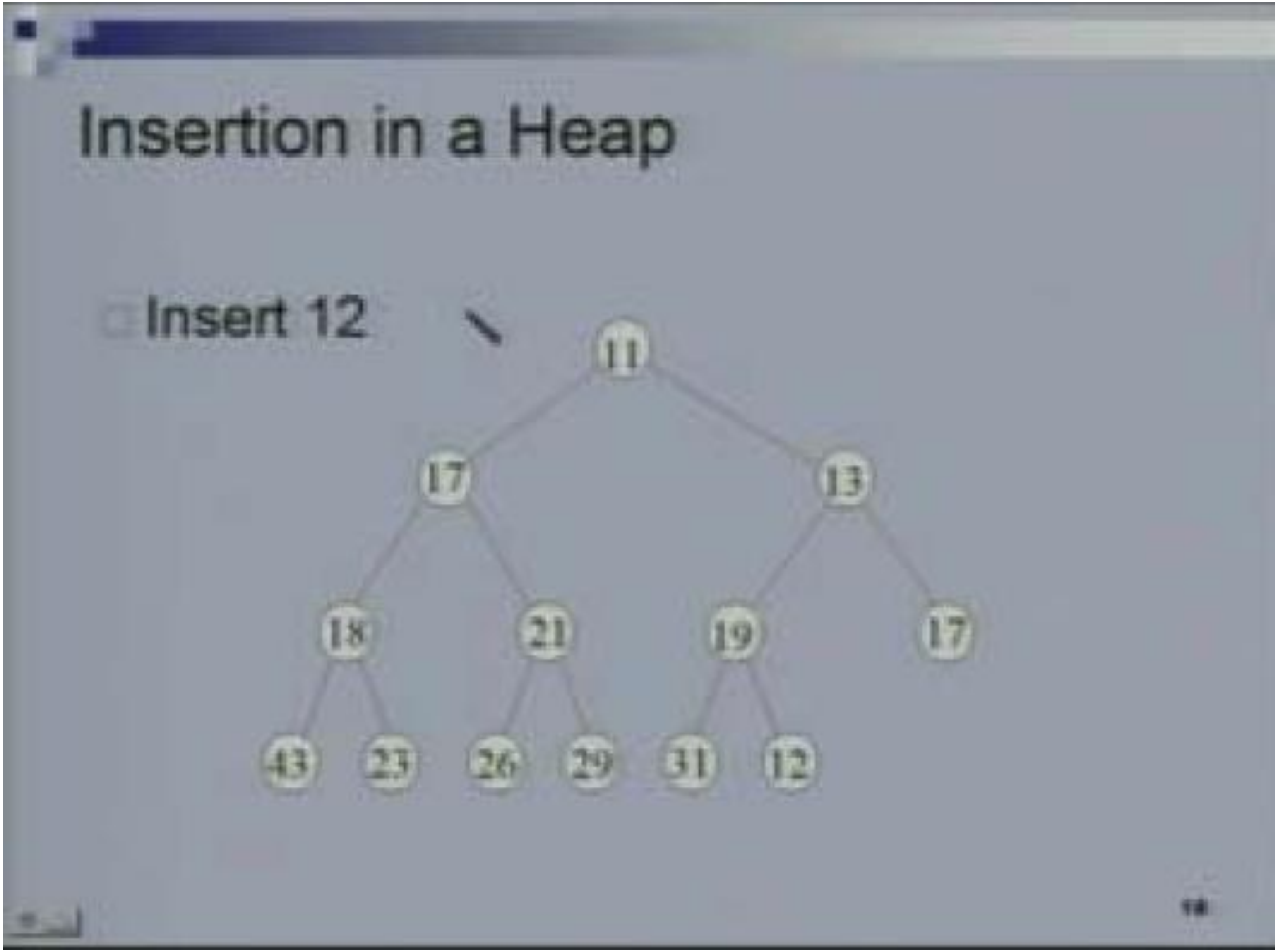
Implementing Heaps (2)

- Notice the implicit tree links; children of node *i* are 2*i* and 2*i*+1
- Why is this useful?
  - In a binary representation, a multiplication/division by two is left/right shift
  - Adding 1 can be done by adding the lowest bit

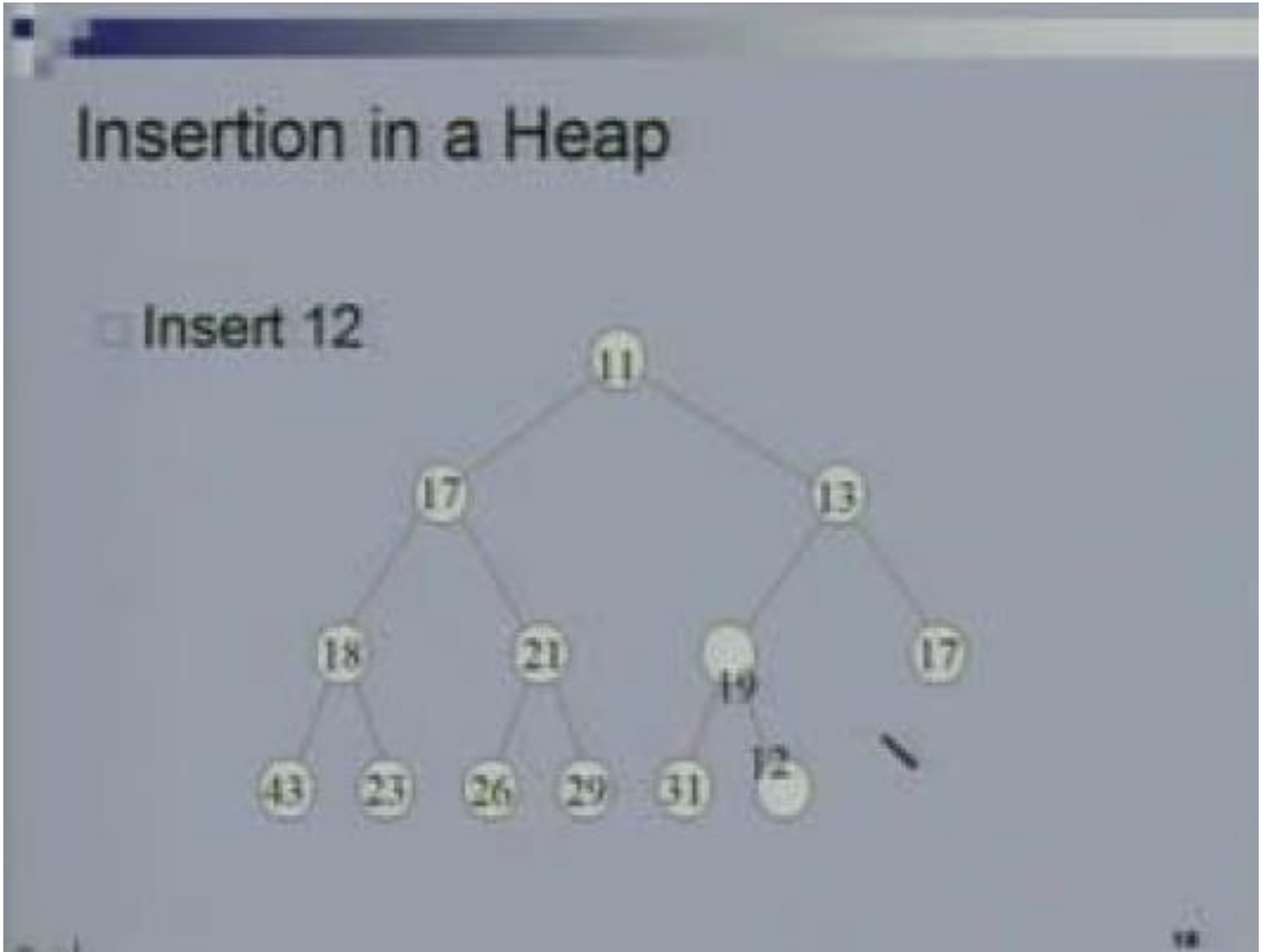
Binary Heap



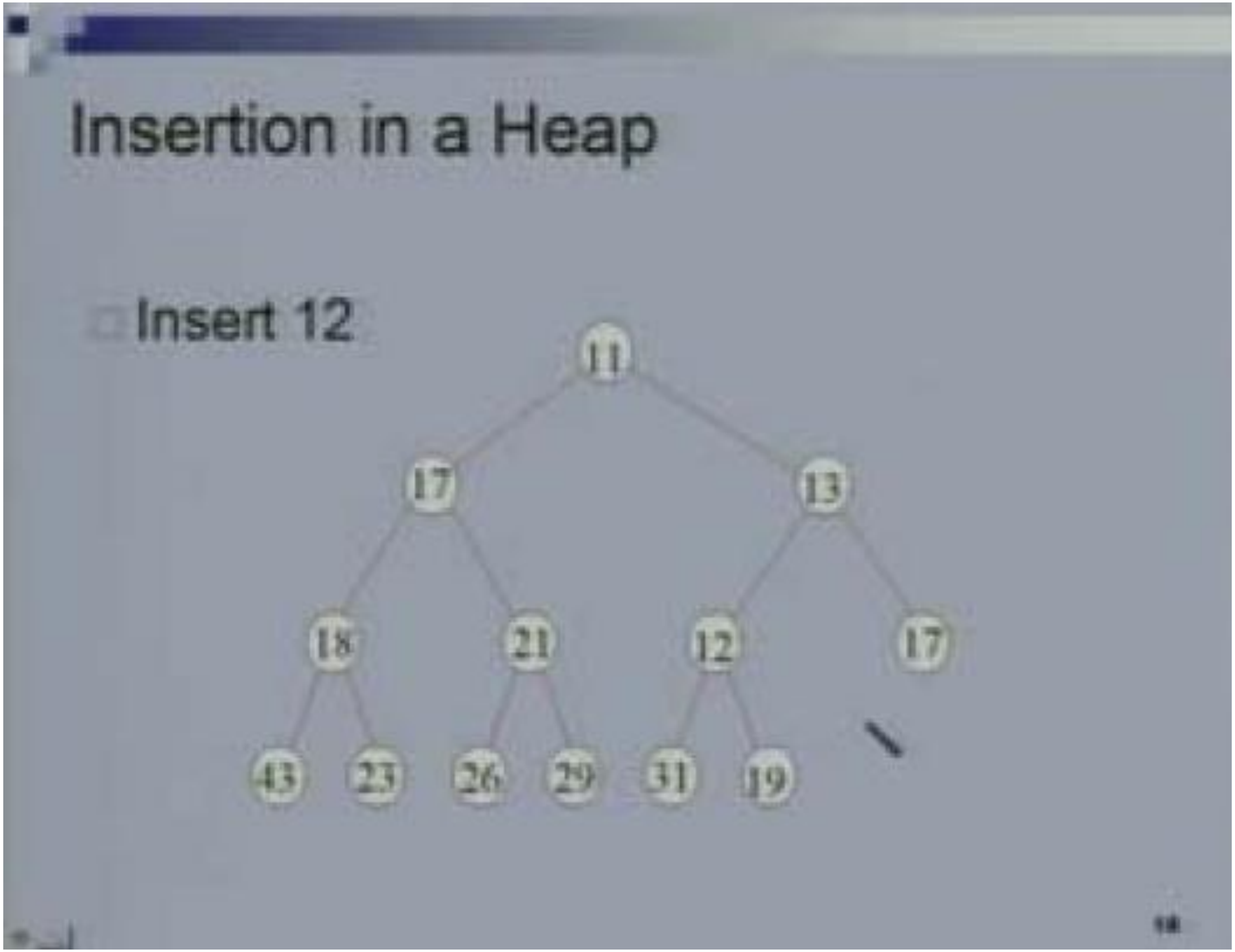
Binary Heap


Insertion in a Heap
Insert 12
11
17
13
18
21
19
17
43
23
26
29
31
12

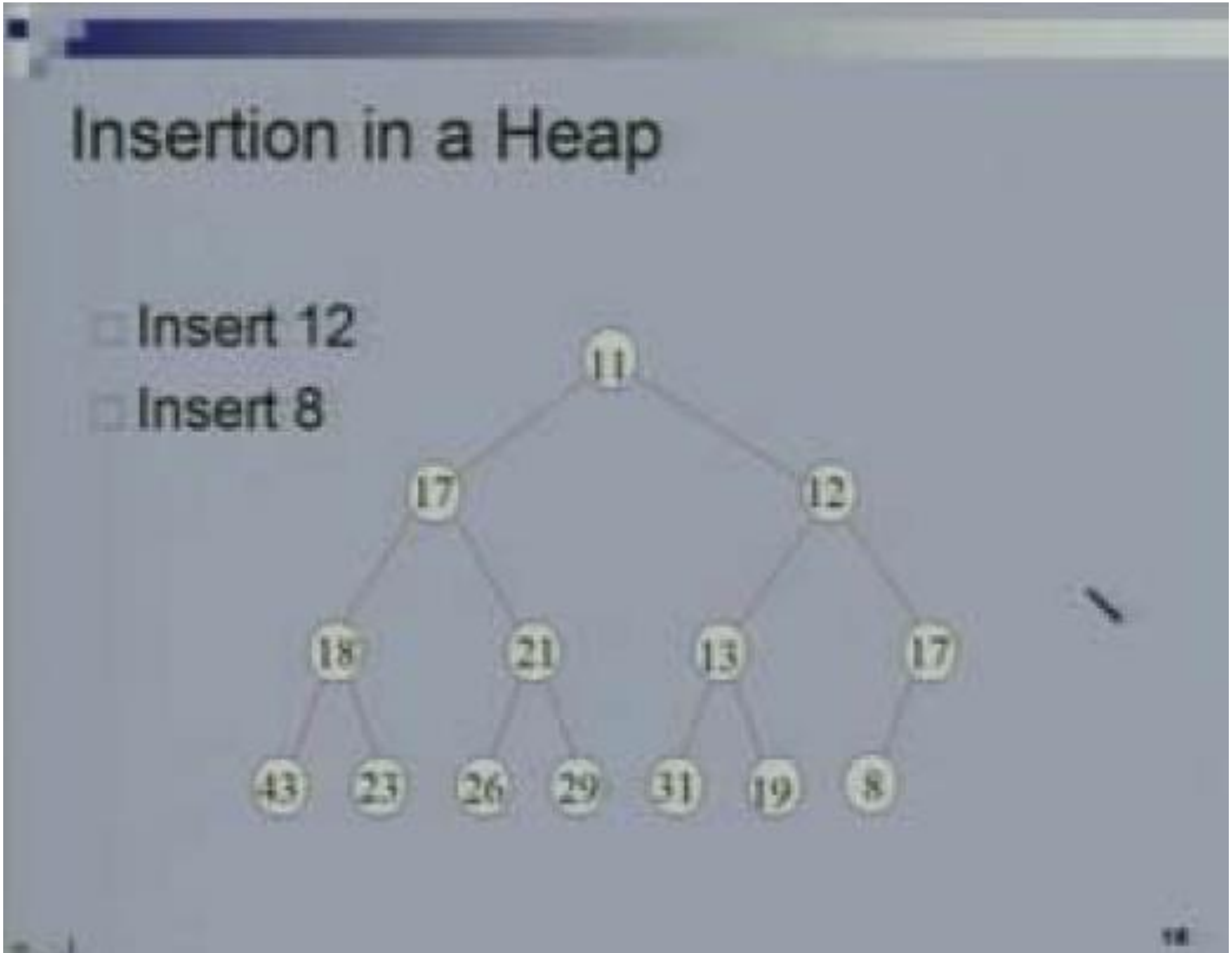
Binary Heap


Insertion in a Heap
Insert 12
11
17
13
18
21
17
19
12
43
23
26
29
31

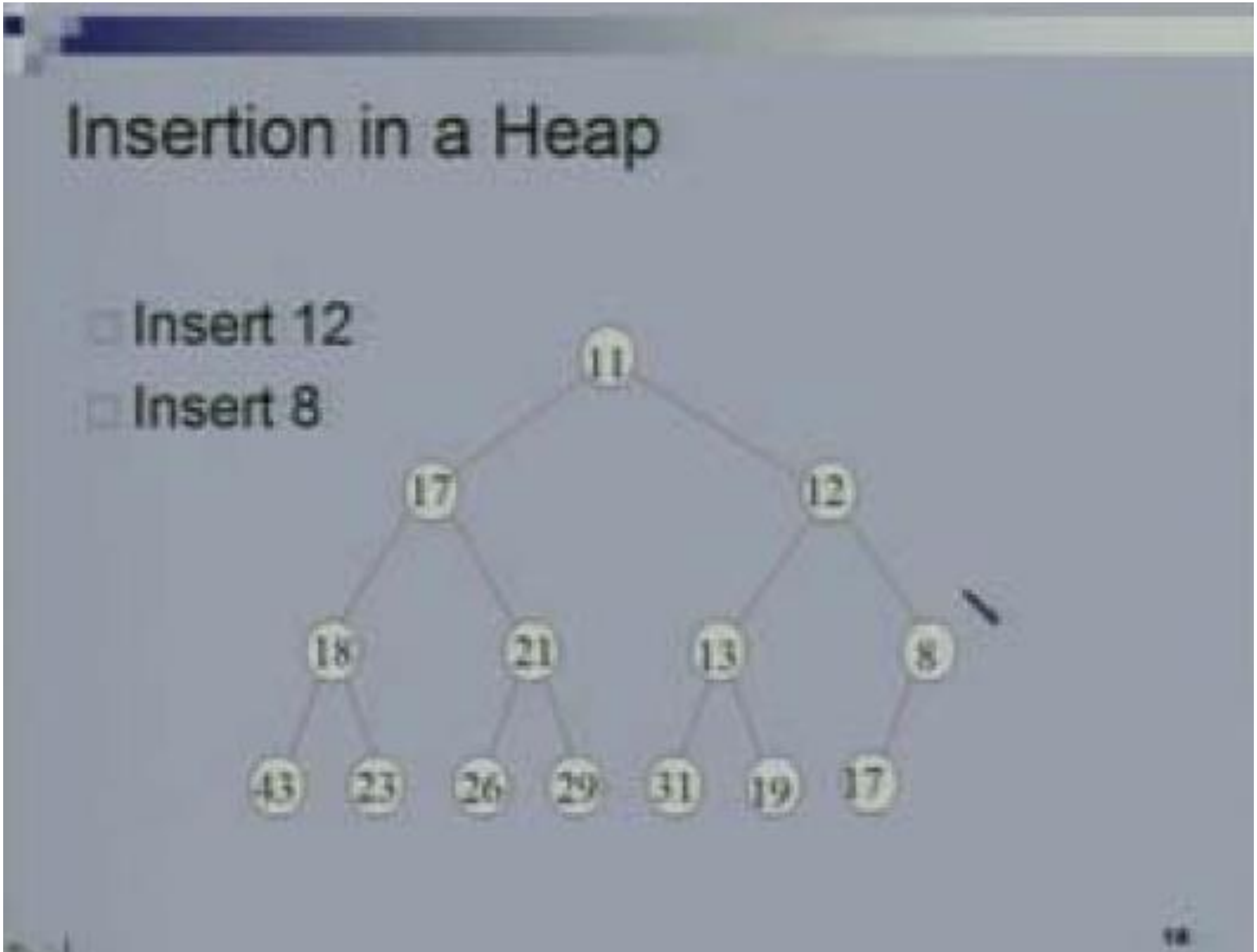
Binary Heap


Insertion in a Heap
Insert 12
11
17
13
18
21
12
17
43
23
26
29
31
19

Binary Heap


Insertion in a Heap
Insert 12
Insert 8
11
17
12
18
21
13
17
43
23
26
29
31
19
8

Binary Heap


Insertion in a Heap
Insert 12
Insert 8
11
17
12
18
21
13
8
43
23
26
29
31
19
17

Binary Heap

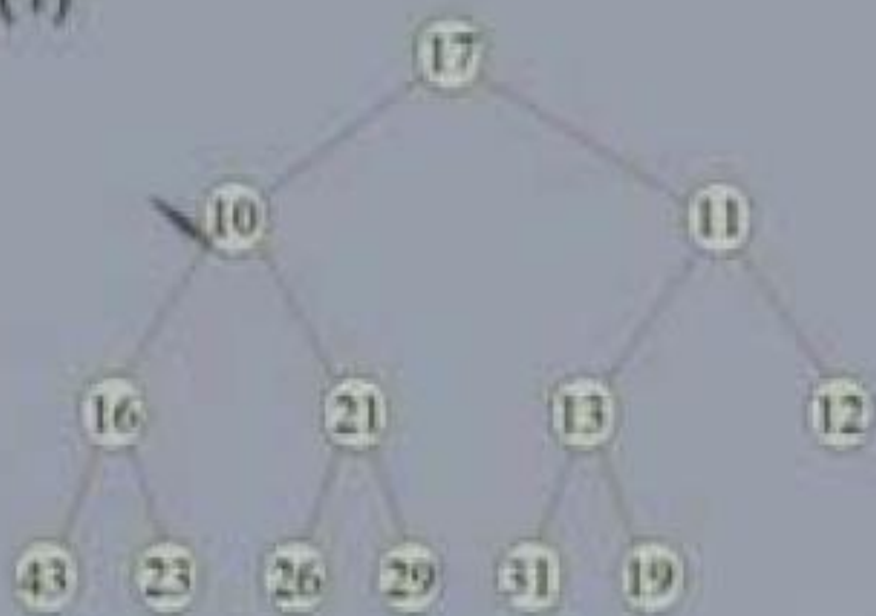
Heapify

- *i* is index into the array *A*
- Binary trees rooted at Left(*i*) and Right(*i*) are heaps
- But, *A*[*i*] might be ~~smaller~~ larger than its children, thus violating the heap property
- The method **Heapify** makes binary tree rooted at *i* a heap by moving *A*[*i*] down the heap.

Binary Heap

Heapify

- Heap property violated at node with index 1 but subtrees rooted at 2, 3 are heaps.
- heapify(1)



Binary Heap

Running time Analysis

- A heap of n nodes has height O(log n).
- While inserting we might have to move the element all the way to the top.
- Hence at most O(log n) steps required.
- In Heapify, the element might be moved all the way to the last level.
- Hence Heapify also requires O(log n) time.

34

Binary Heap

Binary Heaps (A Few More Operations)

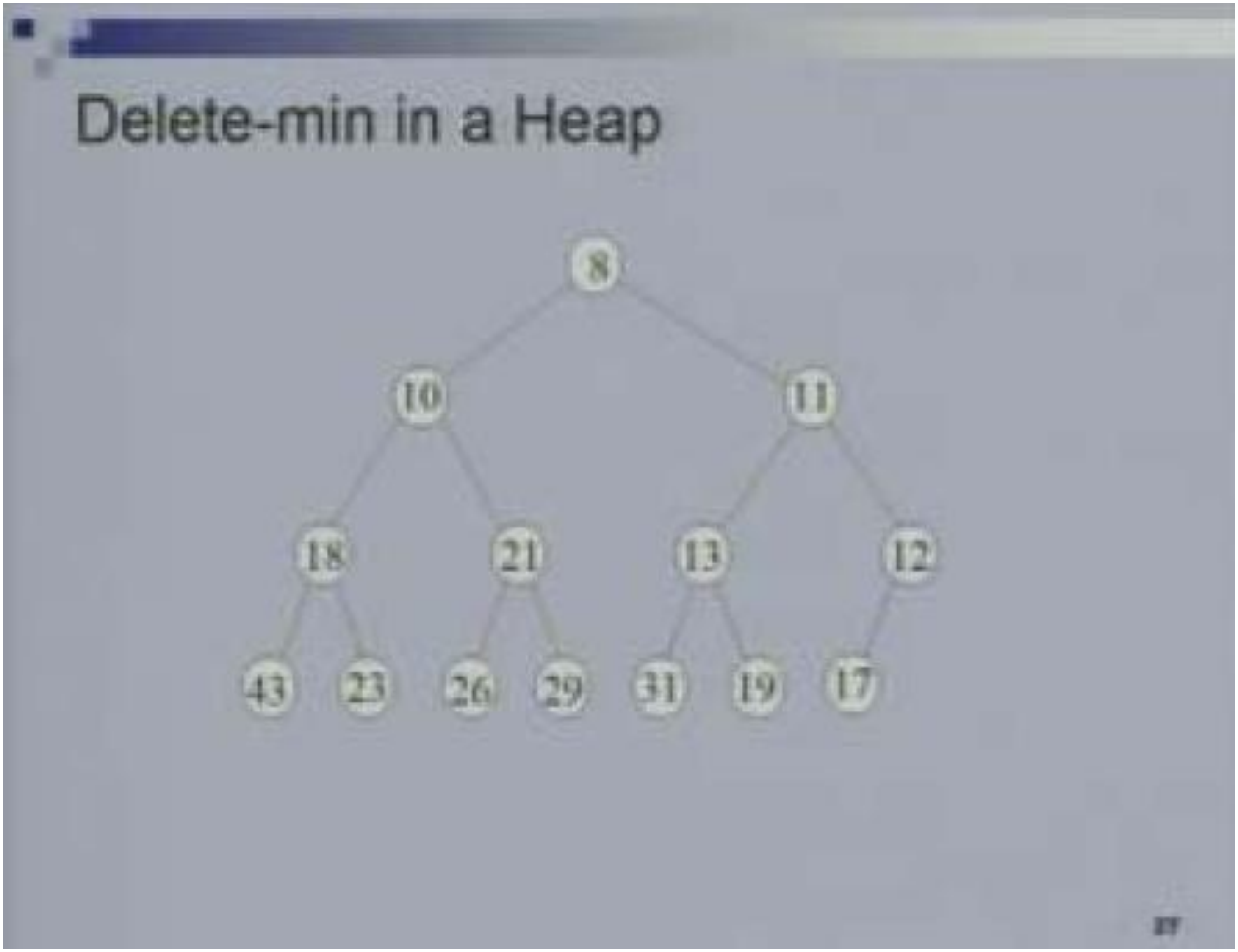
- Delete-min
- Building a heap in O(n) time
- Heap Sort

Binary Heap

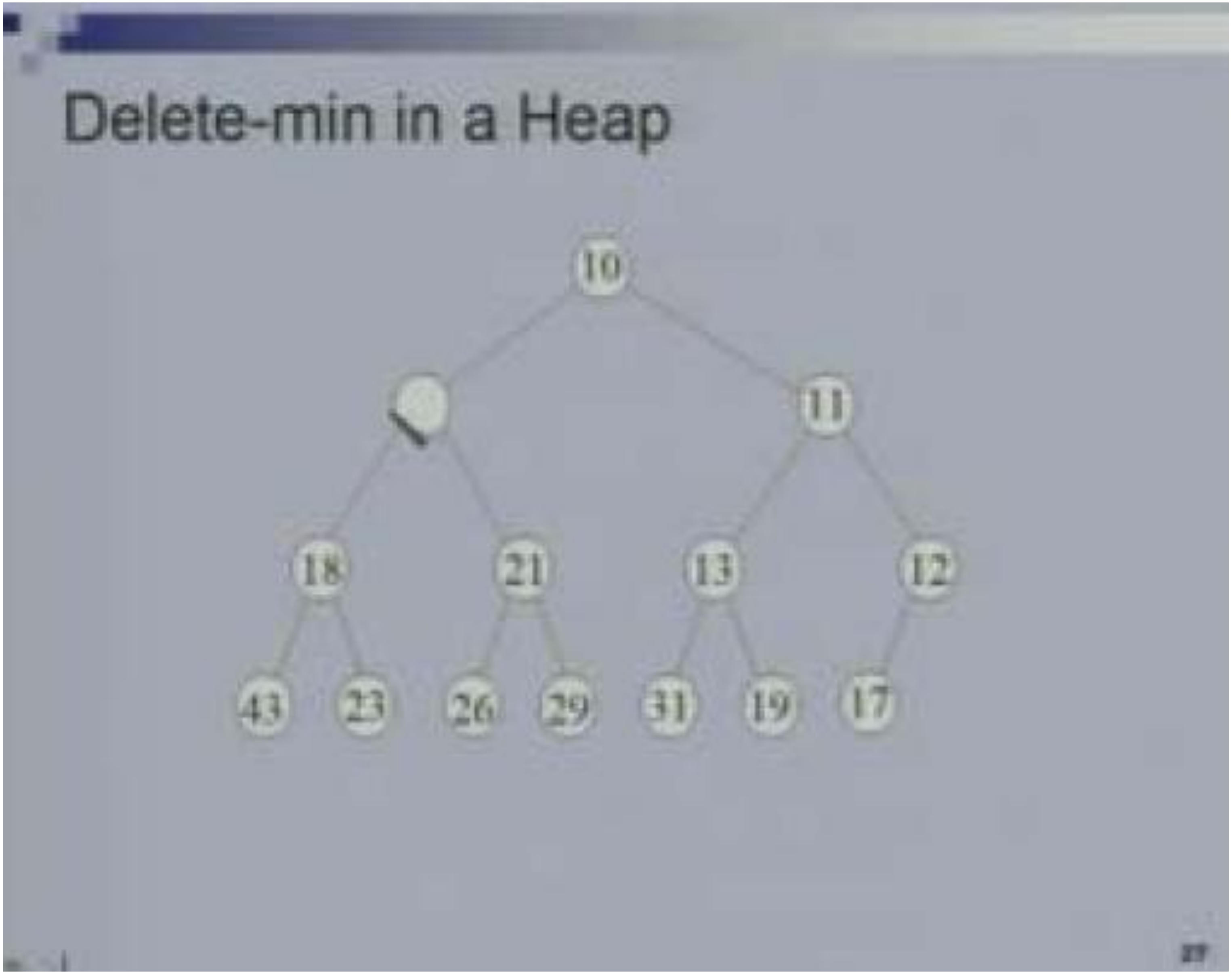
Delete-min

- The minimum element is the one at the top of the heap.
- We can delete this and move one of its children up to fill the space.
- Empty location moves down the tree.
- Might end up at any position on last level.
- Resulting tree would not be left filled.

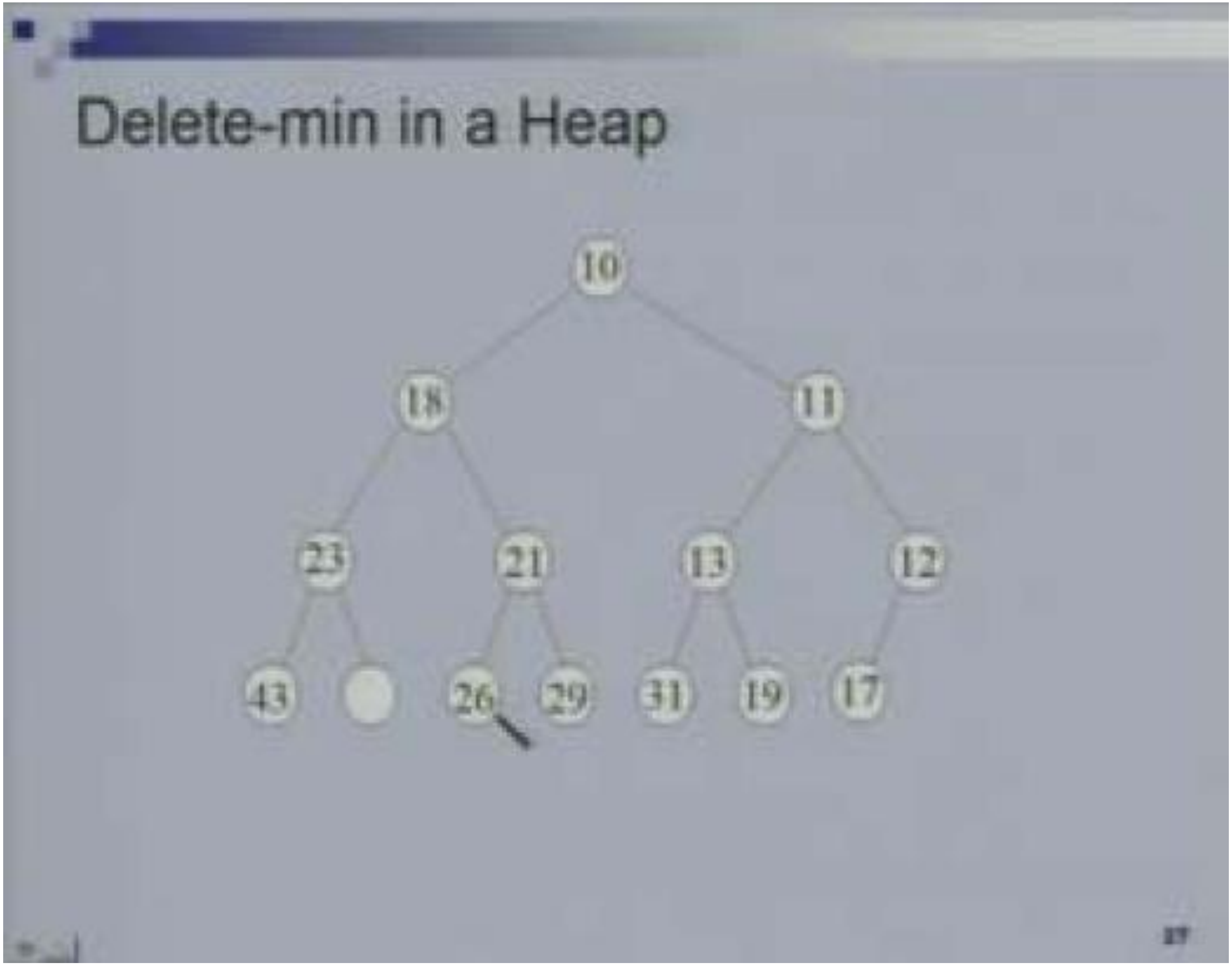
Binary Heap



Binary Heap



Binary Heap

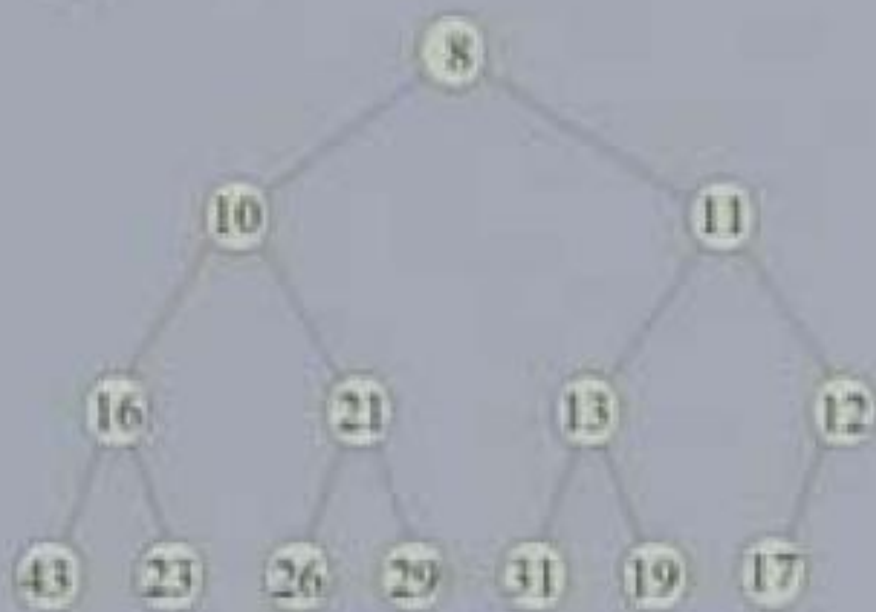


Binary Heap

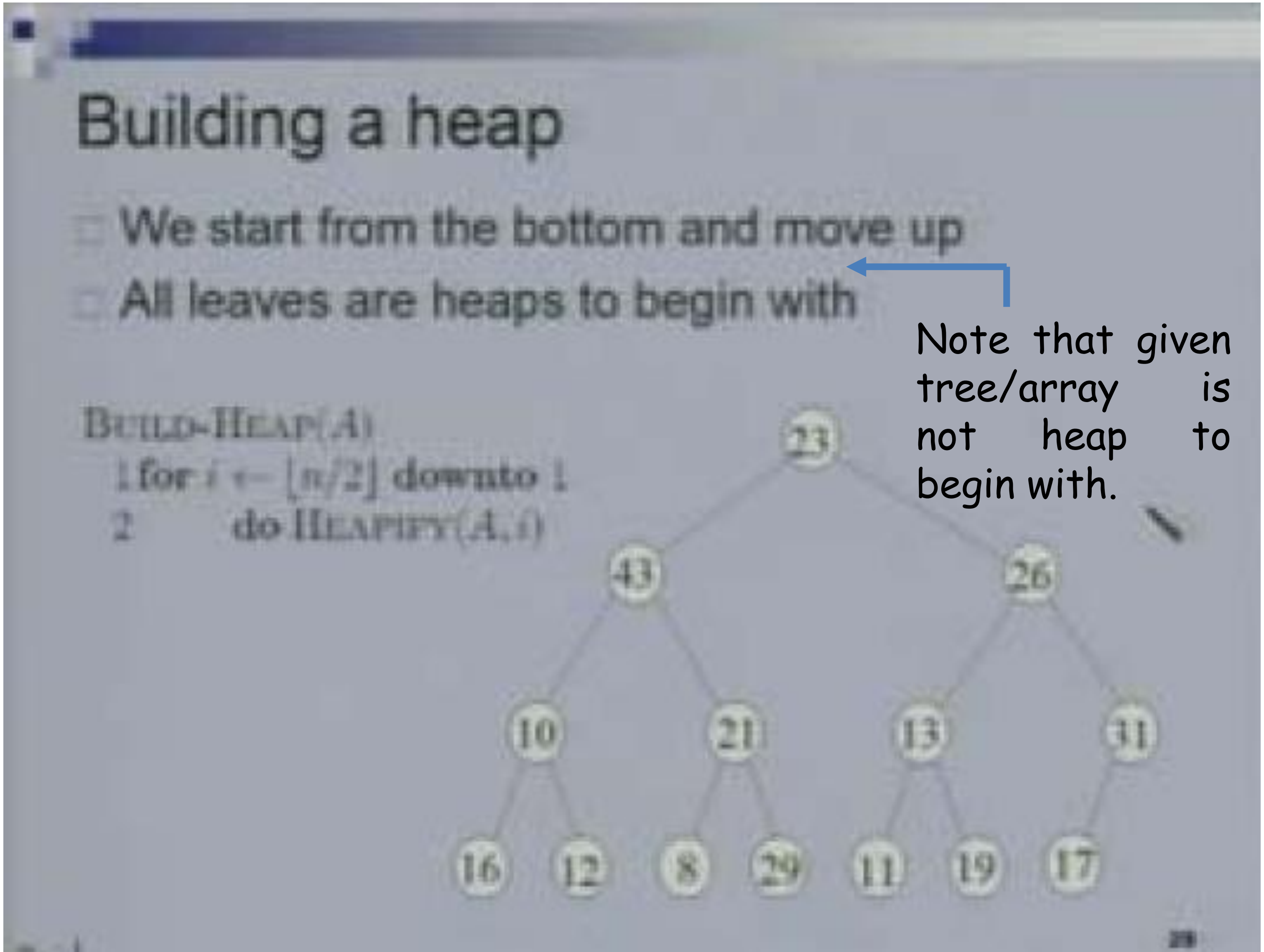
Delete-min in a Heap (2)

- Replace top element with last element of heap.

And then Heapify(1)



Binary Heap



Note that given tree/array is not heap to begin with.

Binary Heap

Building a Heap: Analysis

- Correctness: induction on *i*, all trees rooted at *m* > *i* are heaps
- Running time: ~~*n*~~ $\left\lfloor \frac{n}{2} \right\rfloor$ calls to Heapify = *n* O(*lg n*) = O(*n lg n*)
- We can provide a better O(n) bound.
  - Intuition: for most of the time Heapify works on smaller than *n* element heaps

Binary Heap

- We can provide a better O(n) bound
  - Intuition
    - We can derive a tighter bound by observing that the time for HEAPIFY to run at a node varies with the height of the node in the tree, and the heights of most nodes are small.

    - Our tighter analysis relies on the properties that an n-element heap has height $\lfloor \lg n \rfloor$ and at most $\lceil n/2^{h+1} \rceil$ nodes of any height h.

    - The time required by HEAPIFY when called on a node of height h is O(h), and so we can express the total cost of BUILD-HEAP as being bounded from above by

Binary Heap

- Total Number of swaps required

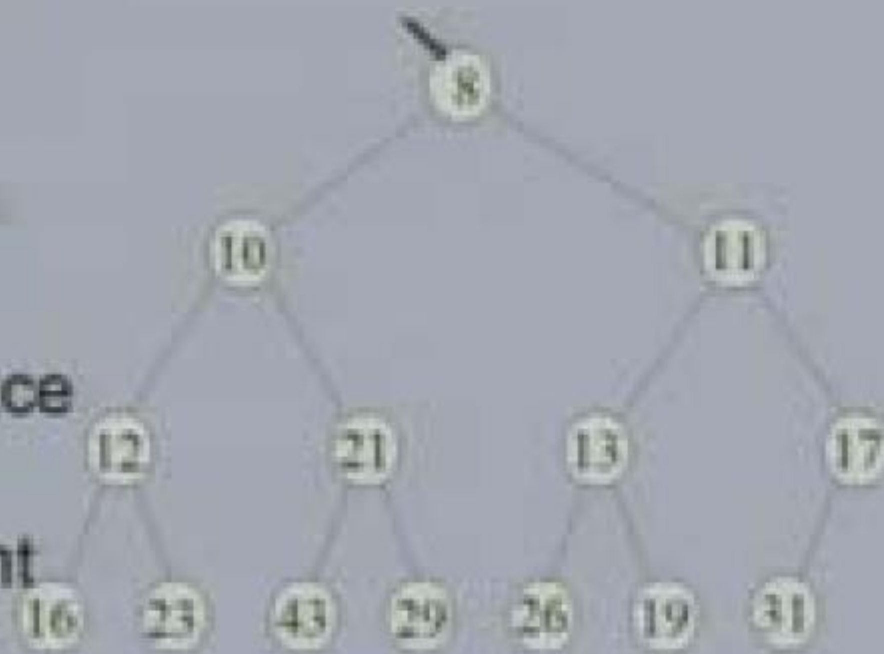
$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) \rightarrow \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2 \,. \quad \left(\because \sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2} \text{ for } |x| < 1. \right)$$

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n) \,.$$

Binary Heap

Heap Sort

- Create a heap.
- Do delete-min repeatedly till heap becomes empty.
- To do an in place sort, we move deleted element to end of heap.

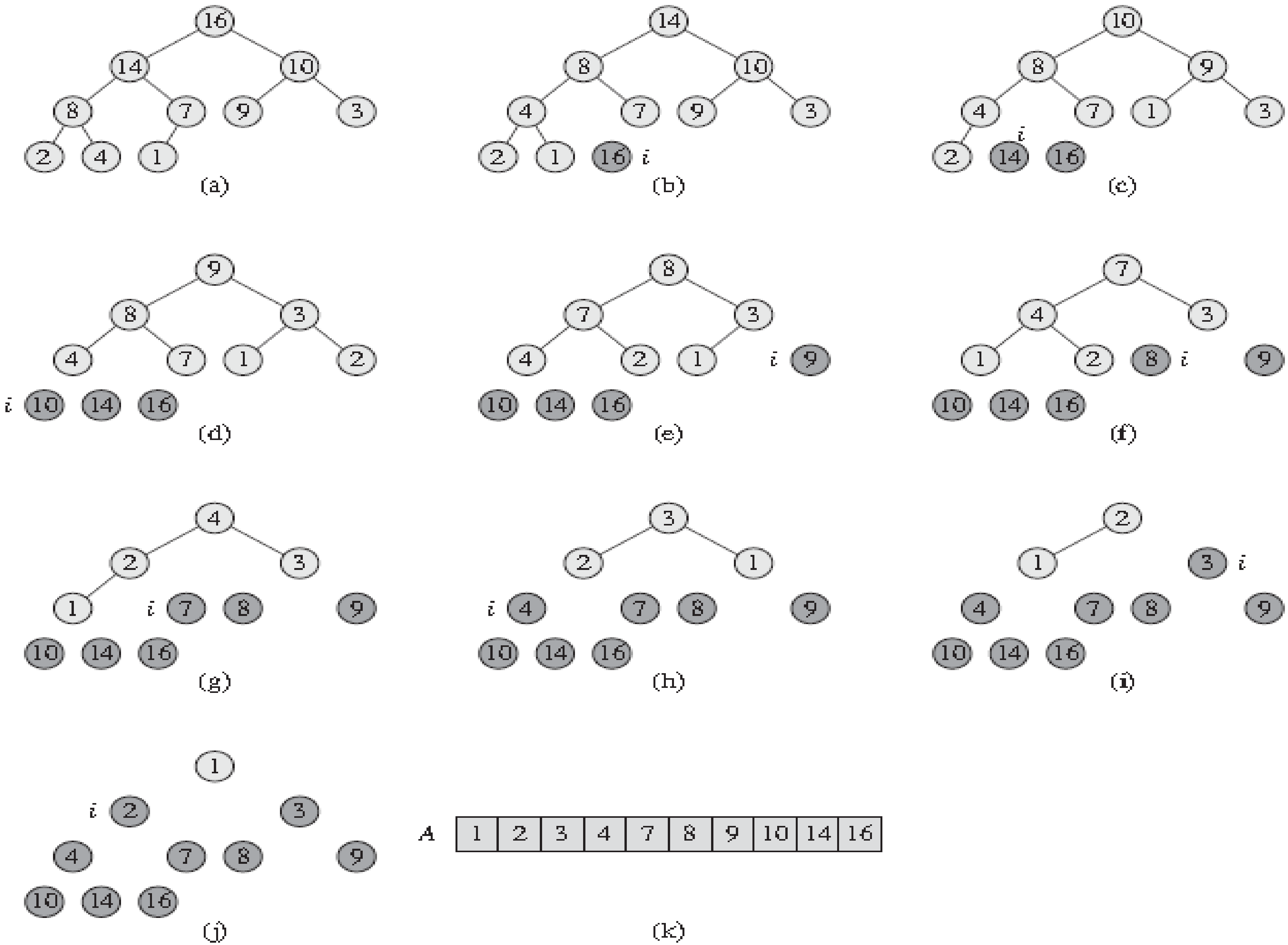


Remember that swap can be performed without temporary variable a = a + b, b = a -b, a = a - b

Binary Heap

```
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)
```

Binary Heap



Binary Heap

```
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)
```

Binary Heap

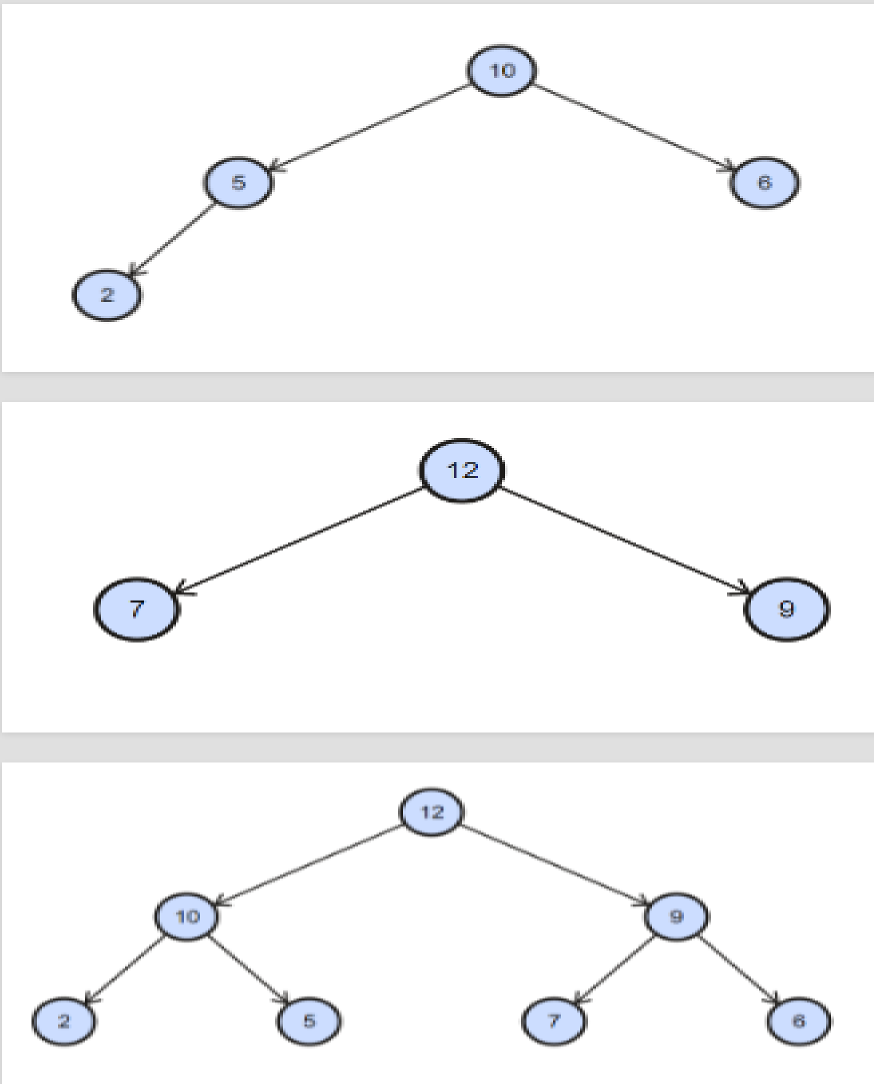
Running times of heap operations

- Insert: O(log n)
- Heapify: O(log n)
- Find minimum: O(1)
- Delete-min: O(log n)
- Building a heap: O(n)
- Heap Sort: O(nlog n)

Binary Heap

- Merging two Binary Heaps

```
Input  : a = {10, 5, 6, 2},
         b = {12, 7, 9}
Output : {12, 10, 9, 2, 5, 7, 6}
```



Just put the two arrays together and create a new heap out of them which takes O(n).

Binomial Heap

- The main application of Binary Heap is in implementation of priority queue.

- Binomial Heap is an extension of Binary Heap that provides faster union or merge operation together with other operations provided by Binary Heap.

-

Operation	Binary[1]	Binomial[1]	Fibonacci[1][2]
find-min	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$
delete-min	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$[b]
insert	$O(\log n)$	$\Theta(1)$[b]	$\Theta(1)$
decrease-key	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$[b]
merge	$\Theta(n)$	$O(\log n)$[d]	$\Theta(1)$

Binomial Heap

- A Binomial Heap is a collection of Binomial Trees