Chapter 10

Heapsort

Dr. Javid Ullah

Department of Computer Science
Islamabad Model College for Boys H-9, Islamabad

Lecture Outline

- Heap
- Binary Heap
- Heap Property
- Building A Heap
- The HeapSort Algorithm

Introduction

- Heapsort
  - Running time: $O(n \lg n)$
    - Like merge sort
  - Sort in place: only a constant number of array elements are stored outside the input array at any time
    - Like insertion sort
- Heap
  - A data structure used by Heapsort to manage information during the execution of the algorithm
  - Can be used as an efficient priority queue

Binary Heap

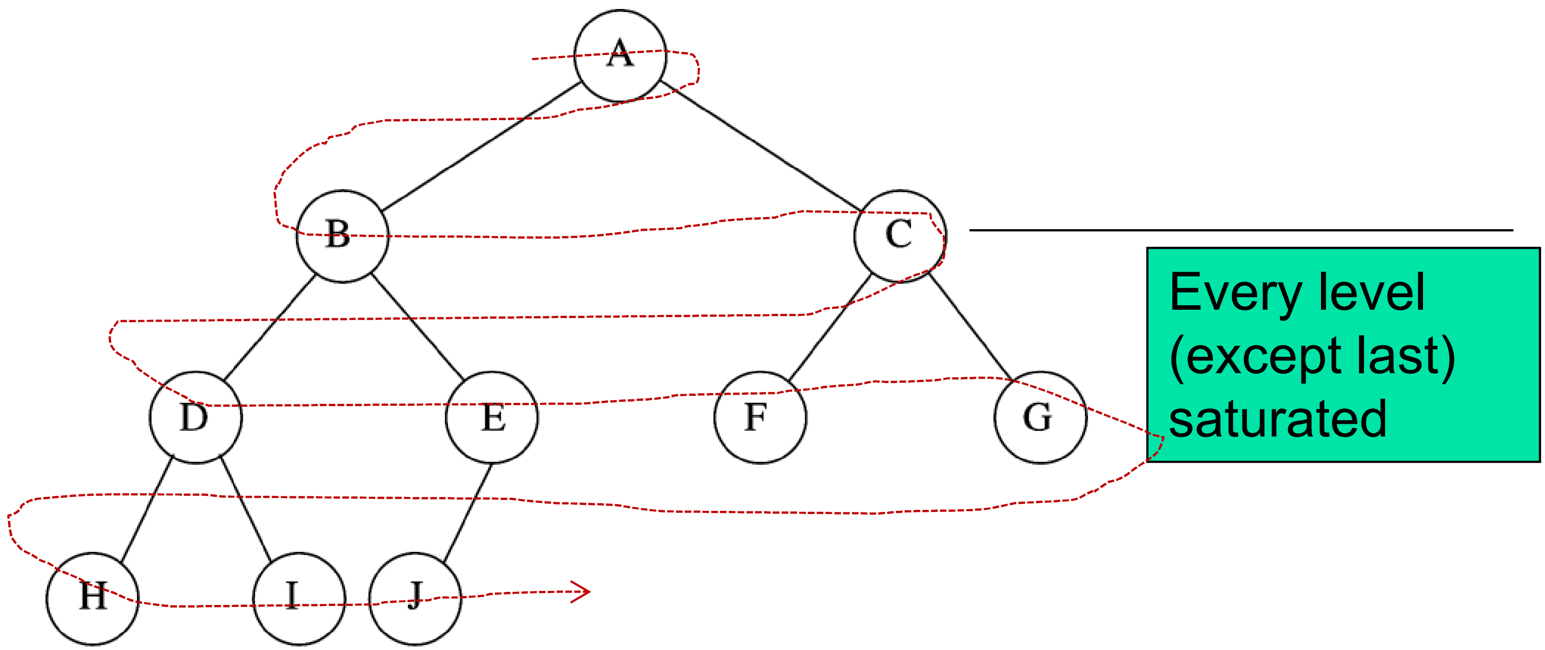
- A heap can be seen as a complete binary tree
- The tree is completely filled on all levels except possibly the lowest.
- In practice, heaps are usually implemented as arrays
- An array A that represent a heap is an object with two attributes: $A[1 \mathrel{..} length[A]]$
  - $length[A]$: # of elements in the array
  - $heap\text{-}size[A]$: # of elements in the heap stored within array A, where $heap\text{-}size[A] \leq length[A]$
  - No element past $A[heap\text{-}size[A]]$ is an element of the heap
- max-heap and min-heap

A =

16	14	10	8	7	9	3	2	4	1

Binary Heap Example

N=10



Array representation:

	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

A Max-Heap

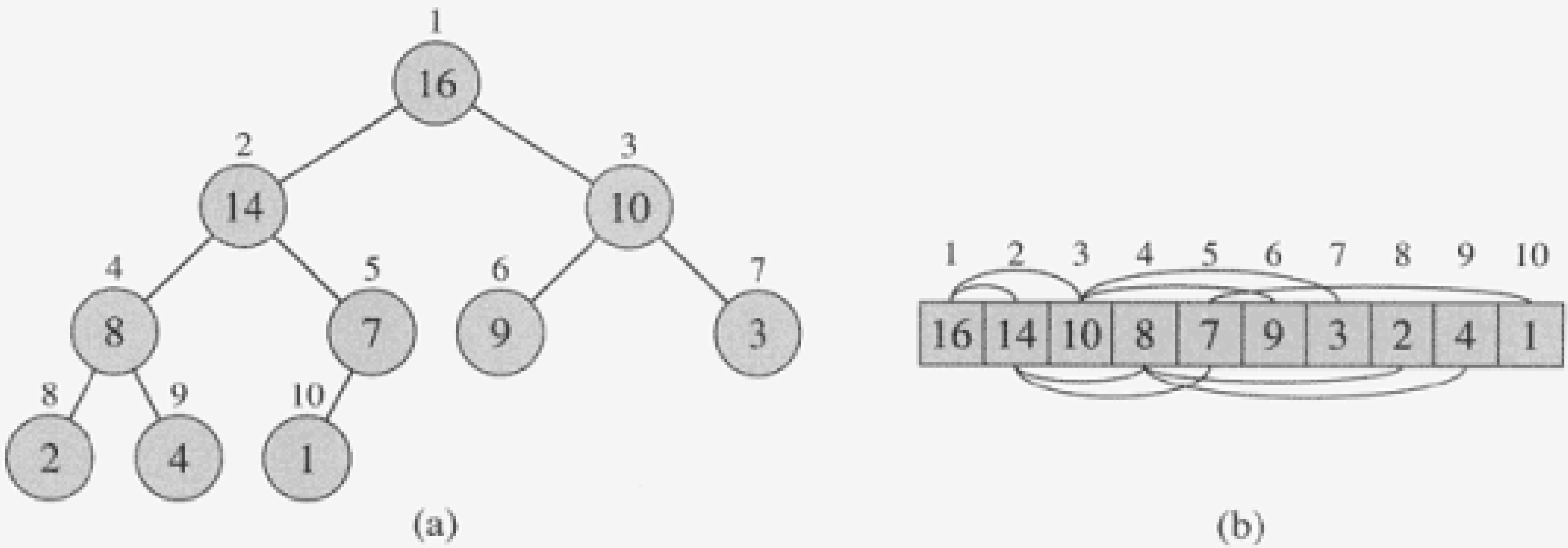
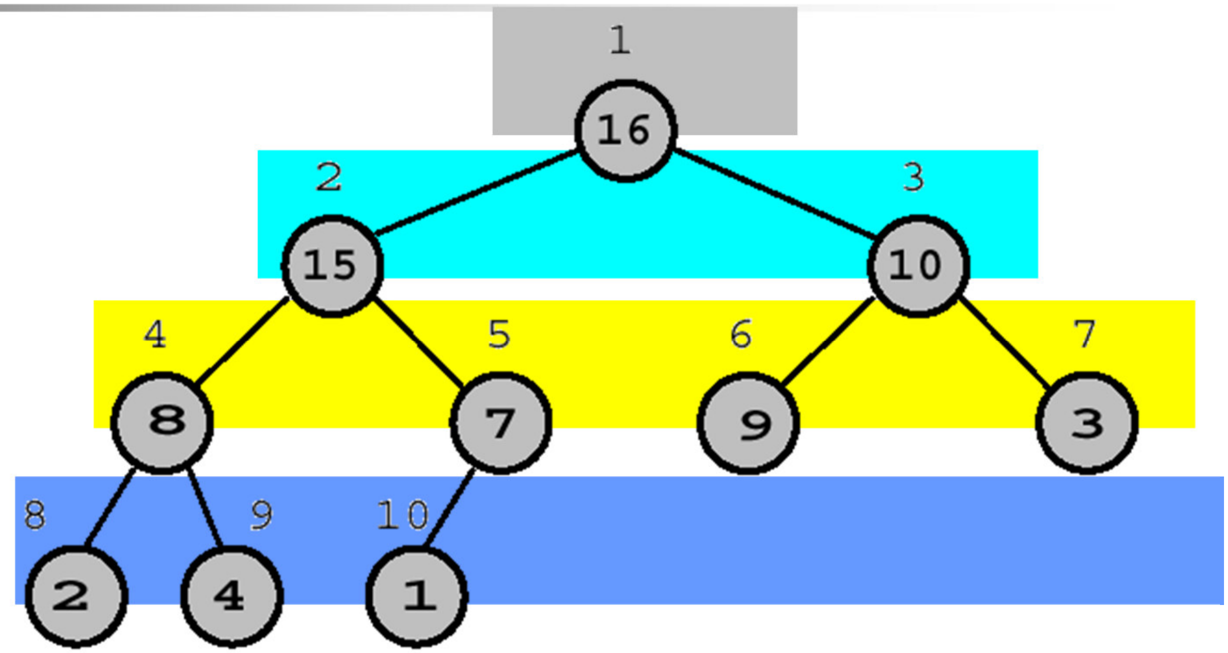


Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

Referencing Heap Elements

- The root node is $A[1]$
- Node i is $A[i]$
- **Parent(i)**
  - **return $\lfloor i/2 \rfloor$**
- **Left(i)**
  - **return $2*i$**
- **Right(i)**
  - **return $2*i + 1$**



1	2	3	4	5	6	7	8	9	10
16	15	10	8	7	9	3	2	4	1

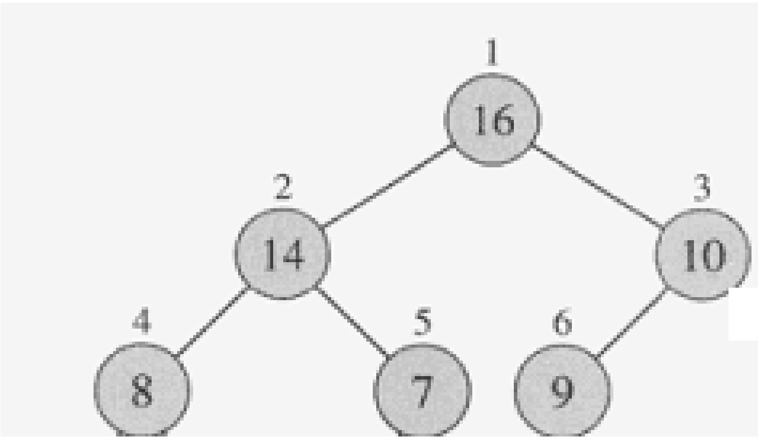
Level: 3 2 1 0

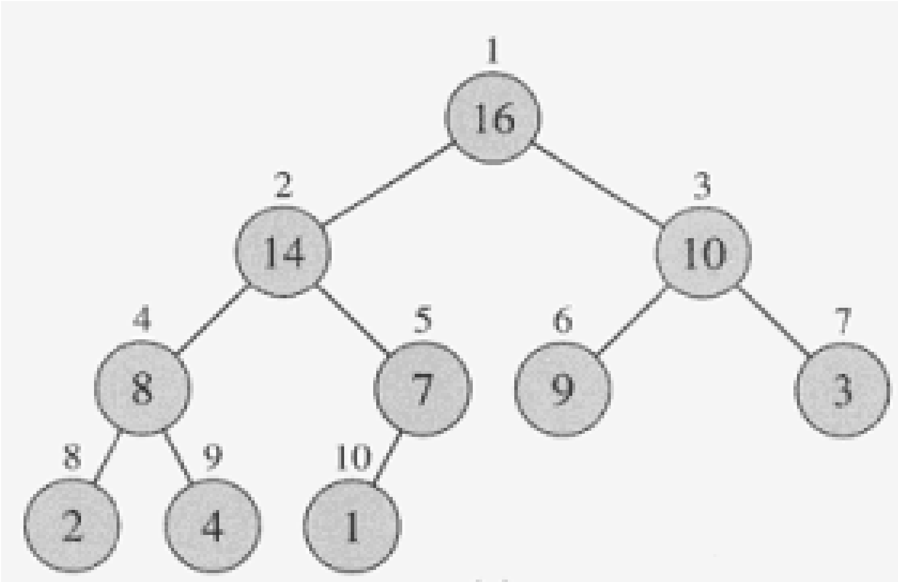
Heap Property

- Heap property – the property that the values in the node must satisfy
- Max-heap property: for every node i other than the root
  - $A[PARENT(i)] \geq A[i]$
  - The value of a node is at most the value of its parent
  - The largest element in a max-heap is stored at the root
  - The subtree rooted at a node contains values on larger than that contained at the node itself
- Min-heap property: for every node i other than the root
  - $A[PARENT(i)] \leq A[i]$

Heap Height

- The height of a node in a heap is the number of edges on the longest simple downward path from the node to a leaf
- The height of a heap is the height of its root
  - The height of a heap of n elements is $\Theta(\lg n)$





of nodes in each level

- Fact: an n-element heap has at most 2^{h-k} nodes of level k, where h is the height of the tree

- for $k = h$ (root level) $\rightarrow 2^{h-h} = 2^0 = 1$
- for $k = h$-1 $\rightarrow 2^{h-(h-1)} = 2^1 = 2$
- for $k = h$-2 $\rightarrow 2^{h-(h-2)} = 2^2 = 4$
- for $k = h$-3 $\rightarrow 2^{h-(h-3)} = 2^3 = 8$
- …
- for $k = 1$ $\rightarrow 2^{h-1} = 2^{h-1}$
- for $k = 0$ (leaves level) $\rightarrow 2^{h-0} = 2^h$

Heap Height

- A heap storing n keys has height $h = \lfloor \lg n \rfloor = \Theta(\lg n)$
- Due to heap being **complete**, we know:
  - The maximum # of nodes in a heap of height h
    - $2^h + 2^{h-1} + \ldots + 2^2 + 2^1 + 2^0 =$
    - $\sum_{i=0 \text{ to } h} 2^i = (2^{h+1} - 1)/(2-1) = 2^{h+1} - 1$
  - The minimum # of nodes in a heap of height h
    - $\mathbf{1} + 2^{h-1} + \ldots + 2^2 + 2^1 + 2^0 =$
    - $\sum_{i=0 \text{ to } h-1} 2^i + 1 = (2^{h-1+1} - 1)/(2-1) + 1 = 2^h$
  - Therefore
    - $2^h \le n \le 2^{h+1} - 1$
    - $h \le \lg n$ & $\lg(n+1) - 1 \le h$
    - $\lg(n+1) - 1 \le h \le \lg n$
  - which in turn implies:
    - $h = \lfloor \lg n \rfloor = \Theta(\lg n)$

Heap Procedures

- MAX-HEAPIFY: maintain the max-heap property
  - O(lg *n*)
- BUILD-MAX-HEAP: produces a max-heap from an unordered input array
  - O(n)
- HEAPSORT: sorts an array in place
  - O(n lg n)
- MAX-HEAP-INSERT, HEAP-EXTRACT, HEAP-INCREASE-KEY, HEAP-MAXIMUM: allow the heap data structure to be used as a priority queue
  - O(lg n)

Maintaining the Heap Property

- MAX-HEAPIFY
  - Inputs: an array A and an index i into the array
  - Assume the binary tree rooted at LEFT(i) and RIGHT(i) are max-heaps, but A[i] may be smaller than its children (violate the max-heap property)
  - MAX-HEAPIFY let the value at A[i] floats down in the max-heap

MAX-HEAPIFY

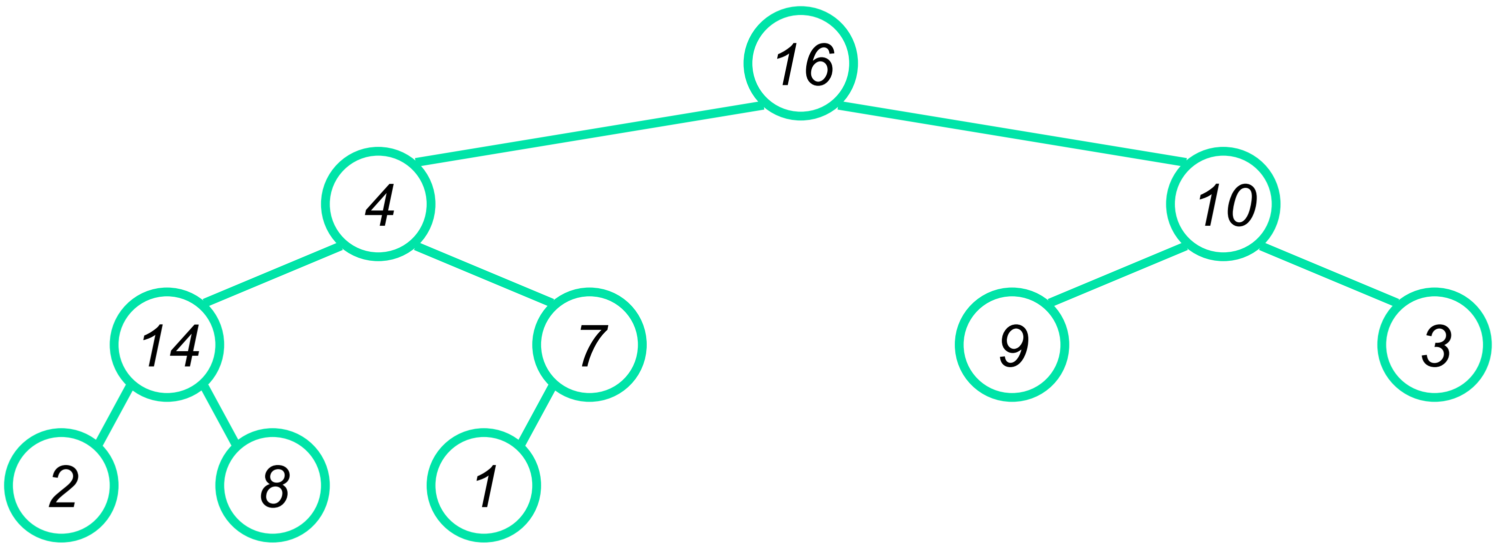
```
MAX-HEAPIFY(A, i)
  l ← LEFT(i)
  r ← RIGHT(i)
  if l ≤ heap-size[A] and A[l] > A[i]
     then largest ← l
     else largest ← i
  if r ≤ heap-size[A] and A[r] > A[largest]
     then largest ← r
  if largest ≠ i
     then exchange A[i] ↔ A[largest]
          MAX-HEAPIFY(A, largest)
```

Extract the indices of LEFT and RIGHT children of i

Choose the largest of A[i], A[l], A[r]

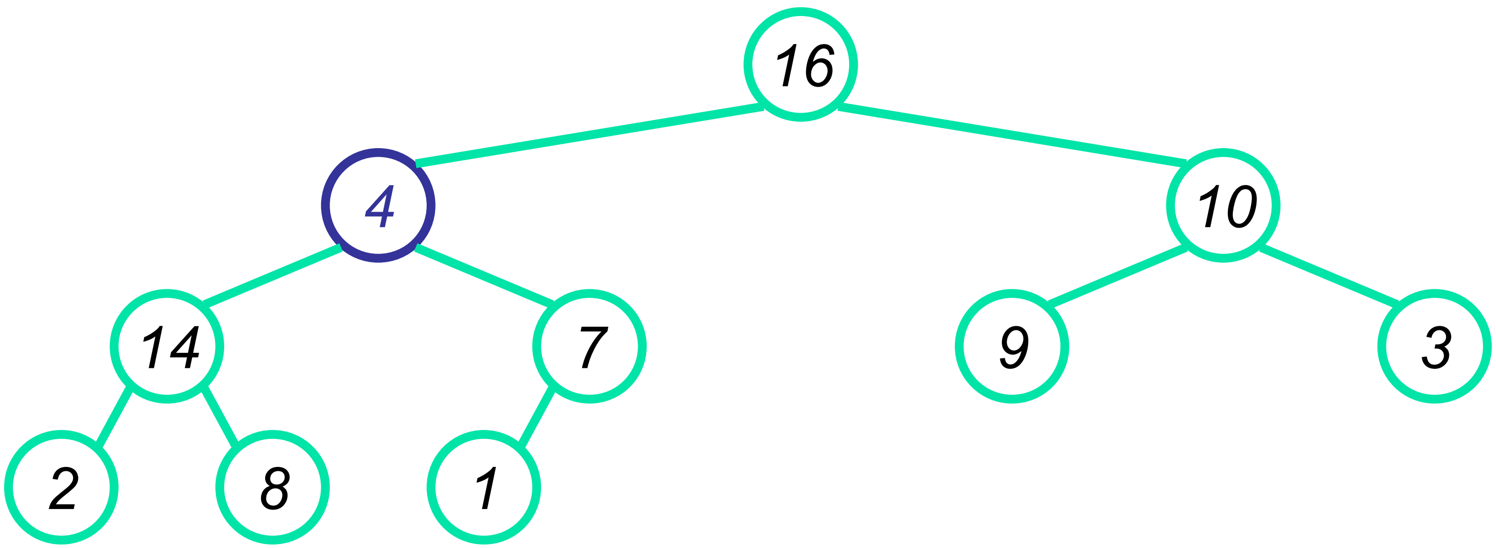
Float down A[i] recursively

MAX-HEAPIFY Example



A =										
	16	4	10	14	7	9	3	2	8	1

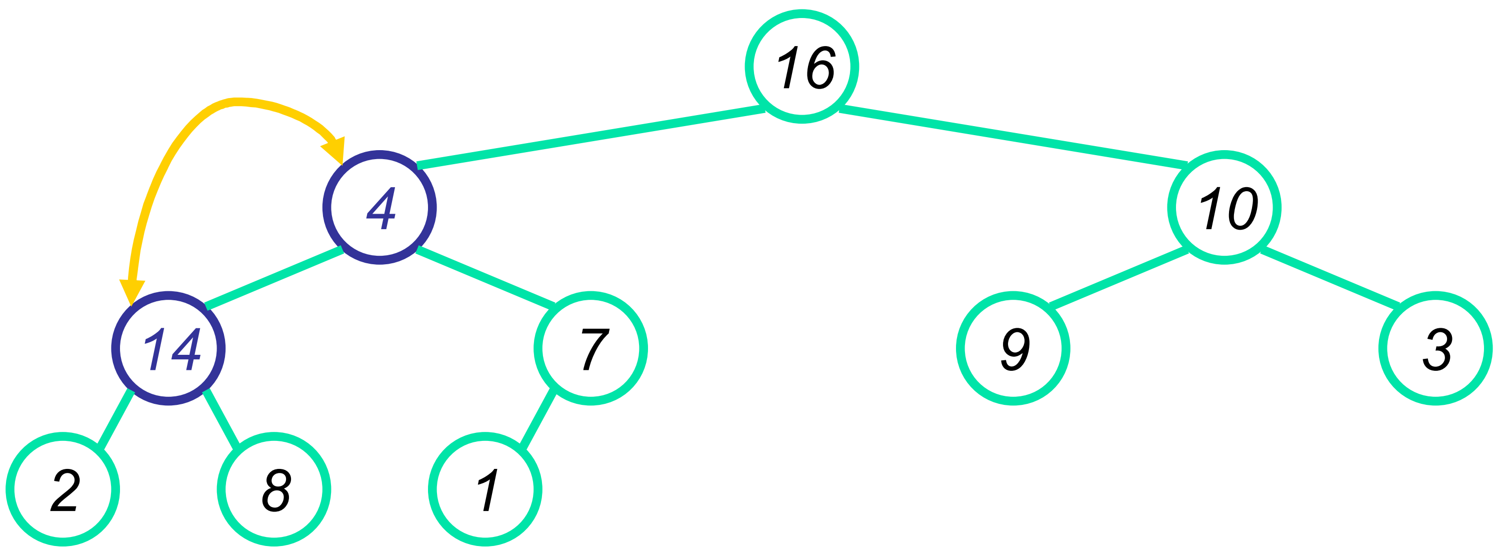
MAX-HEAPIFY Example



A =

16	4	10	14	7	9	3	2	8	1

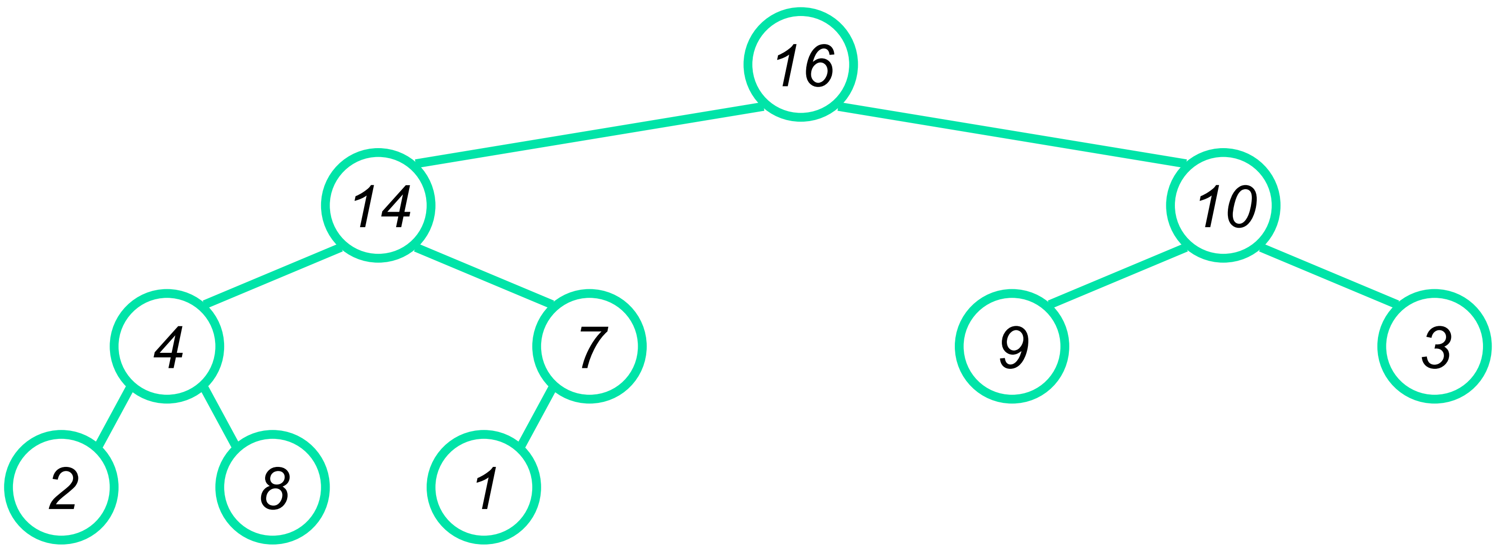
MAX-HEAPIFY Example



A =

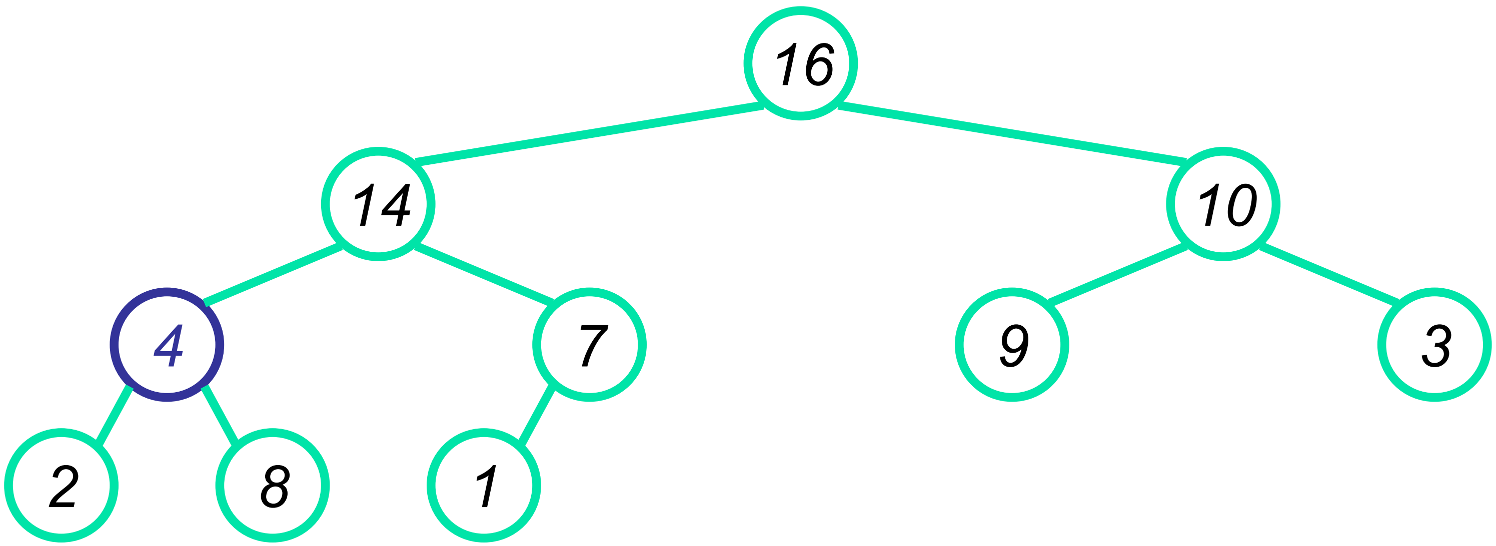
16	4	10	14	7	9	3	2	8	1

MAX-HEAPIFY Example



A =	16	14	10	4	7	9	3	2	8	1

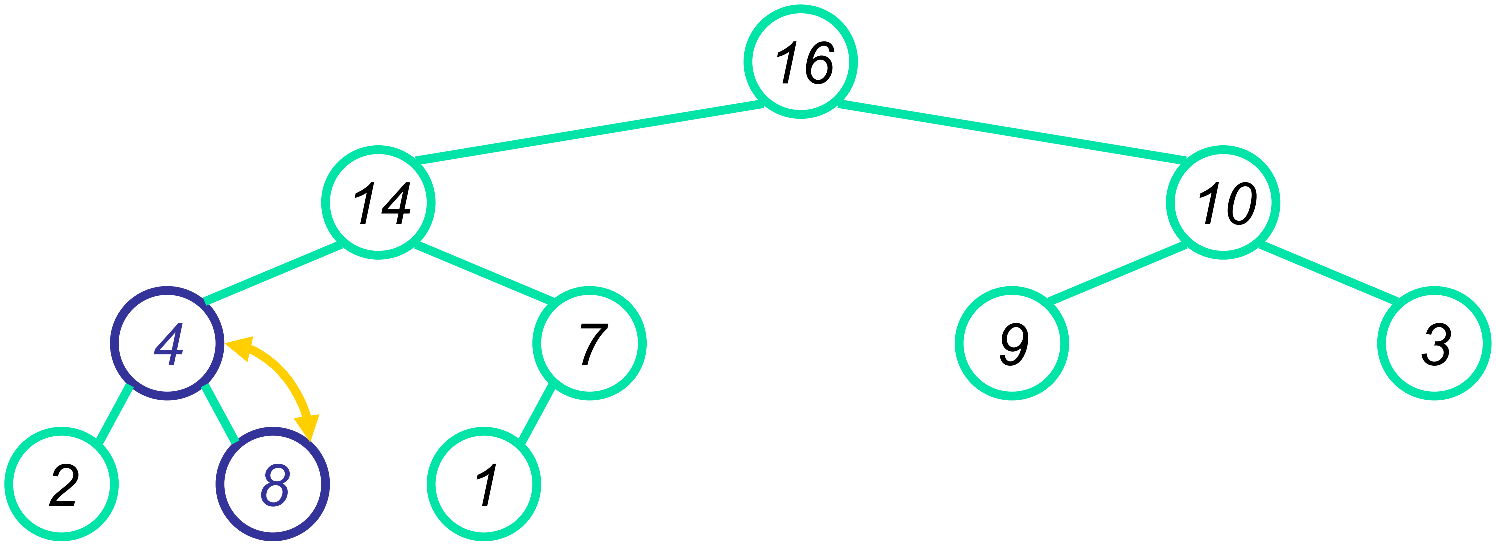
MAX-HEAPIFY Example



$A =$

16	14	10	4	7	9	3	2	8	1

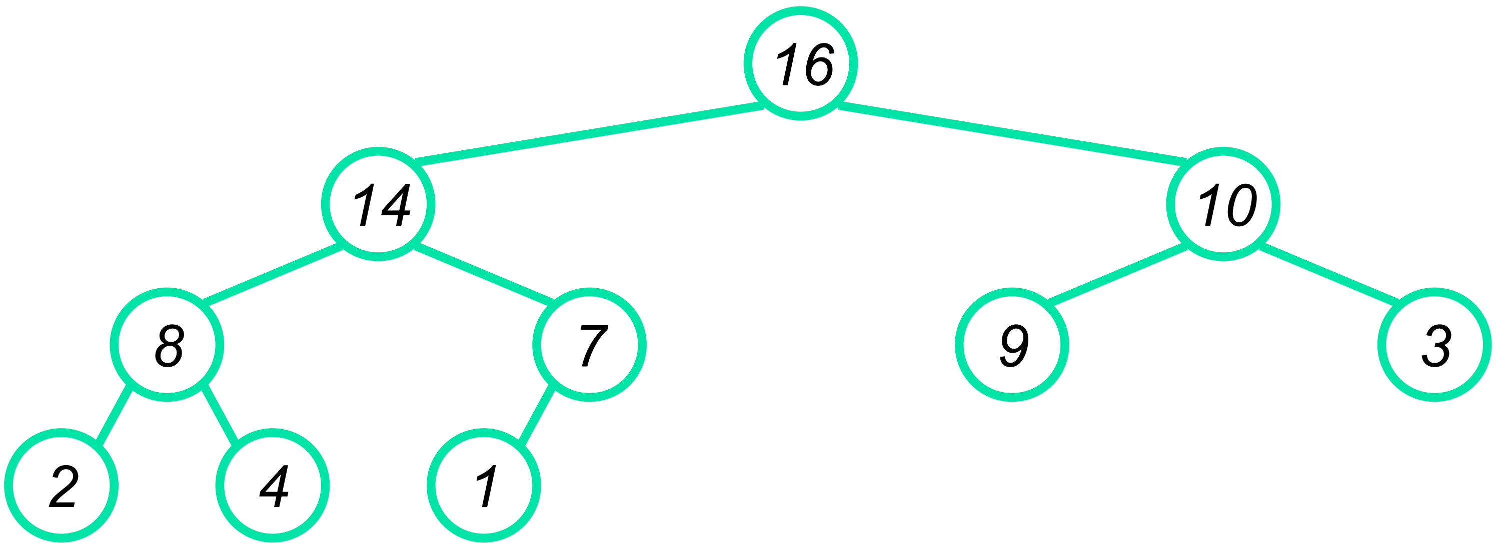
MAX-HEAPIFY Example



A =

16	14	10	4	7	9	3	2	8	1

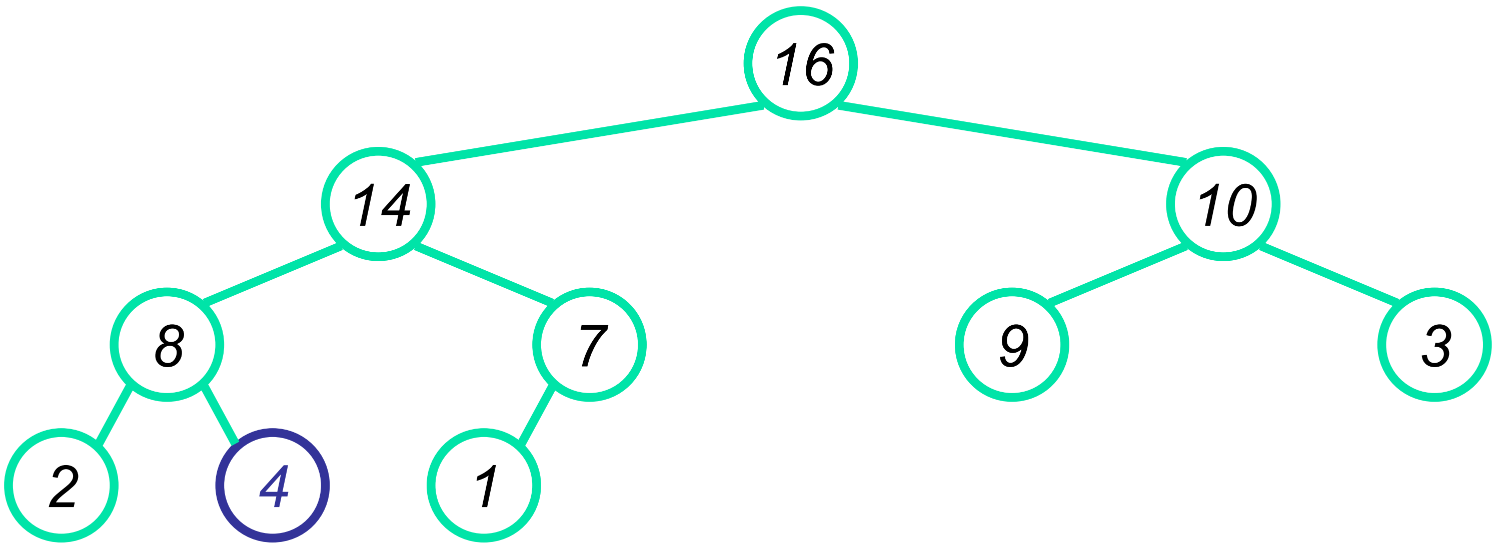
MAX-HEAPIFY Example



A =

16	14	10	8	7	9	3	2	4	1

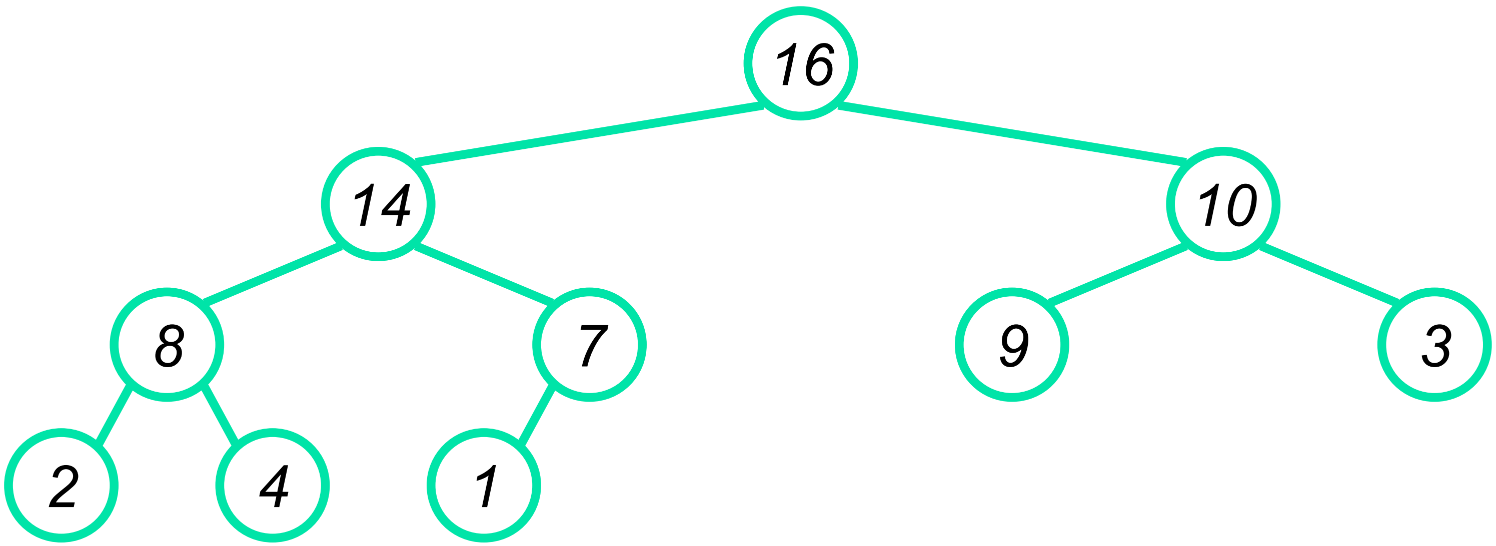
MAX-HEAPIFY Example



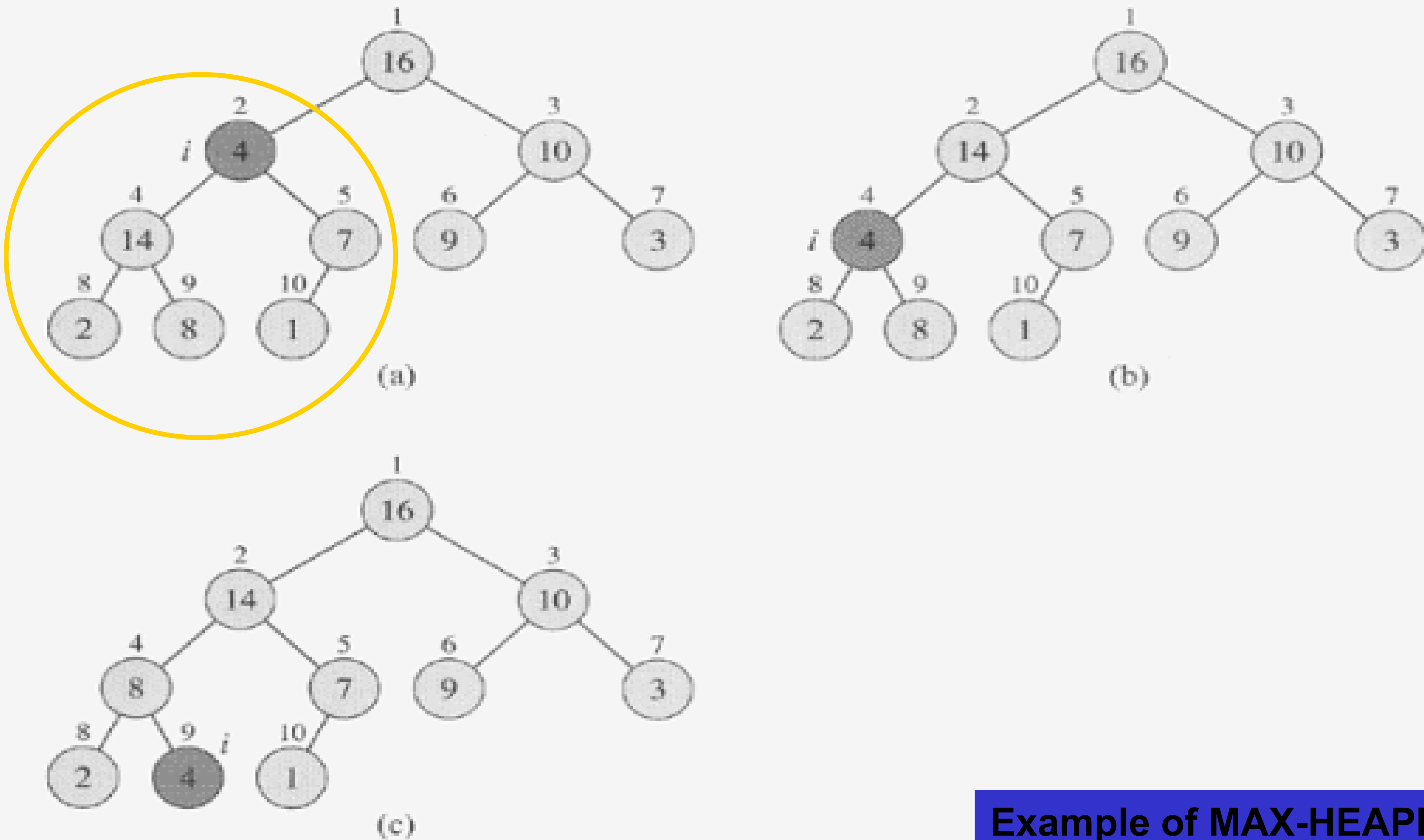
$A =$

16	14	10	8	7	9	3	2	4	1

MAX-HEAPIFY Example



$A =$	16	14	10	8	7	9	3	2	4	1

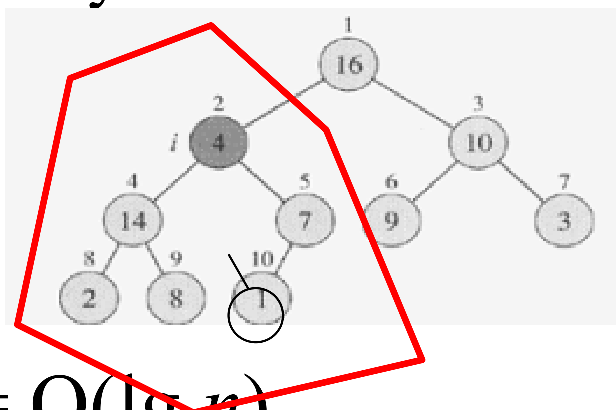


Example of MAX-HEAPIFY

Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where *heap-size*$[A] = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY$(A, 4)$ now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in **(c)**, node 4 is fixed up, and the recursive call MAX-HEAPIFY$(A, 9)$ yields no further change to the data structure.

Analyzing MAX-HEAPIFY (1/2)

- $\Theta(1)$ to find out the largest among A[i], A[LEFT(i)], and A[RIGHT(i)]
- Plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node i
  - The children's subtrees each have size at most 2n/3 – the worst case occurs when the last row of the tree is exactly half full



- $T(n) \leq T(2n/3) + \Theta(1)$
  - By case 2 of the master theorem: $T(n) = O(\lg n)$

Analyzing MAX-HEAPIFY (2/2)

- Alternately, Heapify takes $T(n) = \Theta(h)$
  - h = height of heap = lg n
    - $T(n) = \Theta(\lg n)$

Building A Heap

Build-Max-Heap (1/2)

- We can build a heap in a bottom-up manner by running Build-Max-Heap() on successive subarrays
  - Fact: for array of length n, all elements in range $A[\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2 \,..\, n]$ are heaps (*Why?*)
    - These elements are **leaves**, they do not have children
  - We also know that the leave-level has at most 2^h nodes = $\lceil \boldsymbol{n/2} \rceil$ nodes
    - and other levels have a total of $\lfloor n/2 \rfloor$ nodes
- Walk backwards through the array from n/2 to 1, calling Build-Max-Heap() on each node.

IA, P-133

Build-Max-Heap (2/2)

// given an unsorted array A, make A a heap

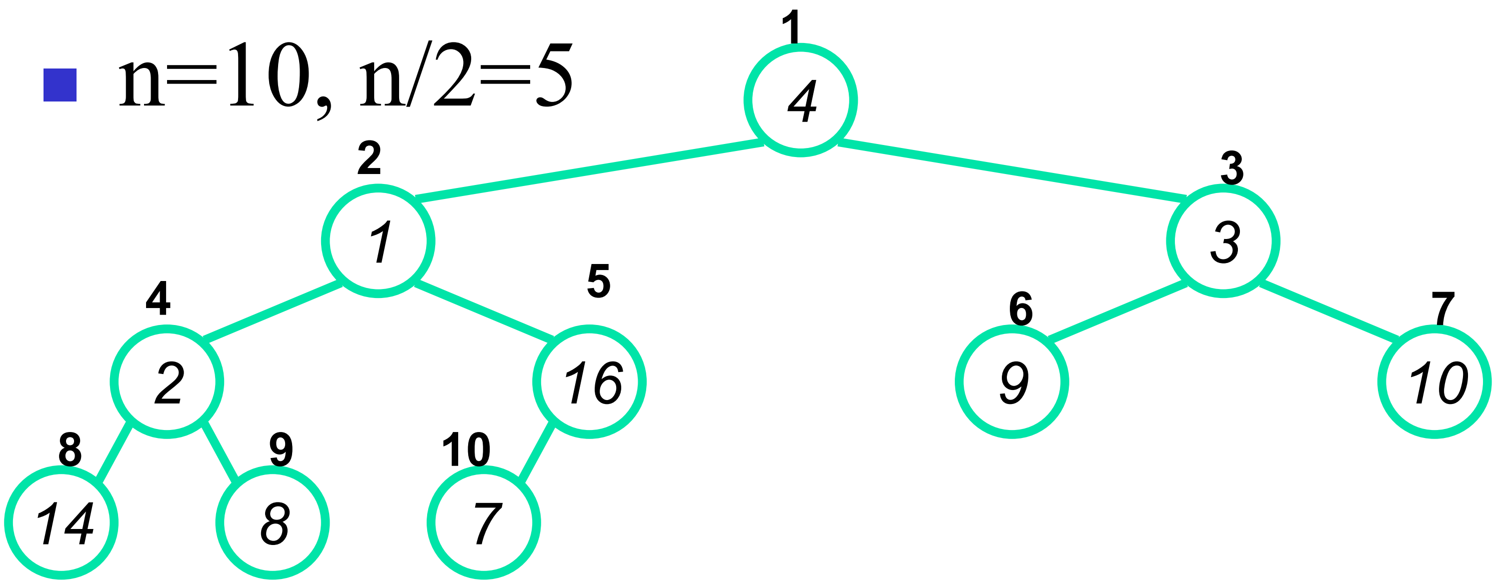
```
BUILD-MAX-HEAP(A)
   heap-size[A] ← length[A]
   for i ← ⌊length[A]/2⌋ downto 1
          do MAX-HEAPIFY(A, i)
```

- The **Build-Max-Heap ()** procedure, which runs in linear time, produces a *max-heap* from an unsorted input array.
- However, the **MAX-HEAPIFY()** procedure, which runs in *O(lg n)* time, is the key to maintaining the heap property.

IA, P-133

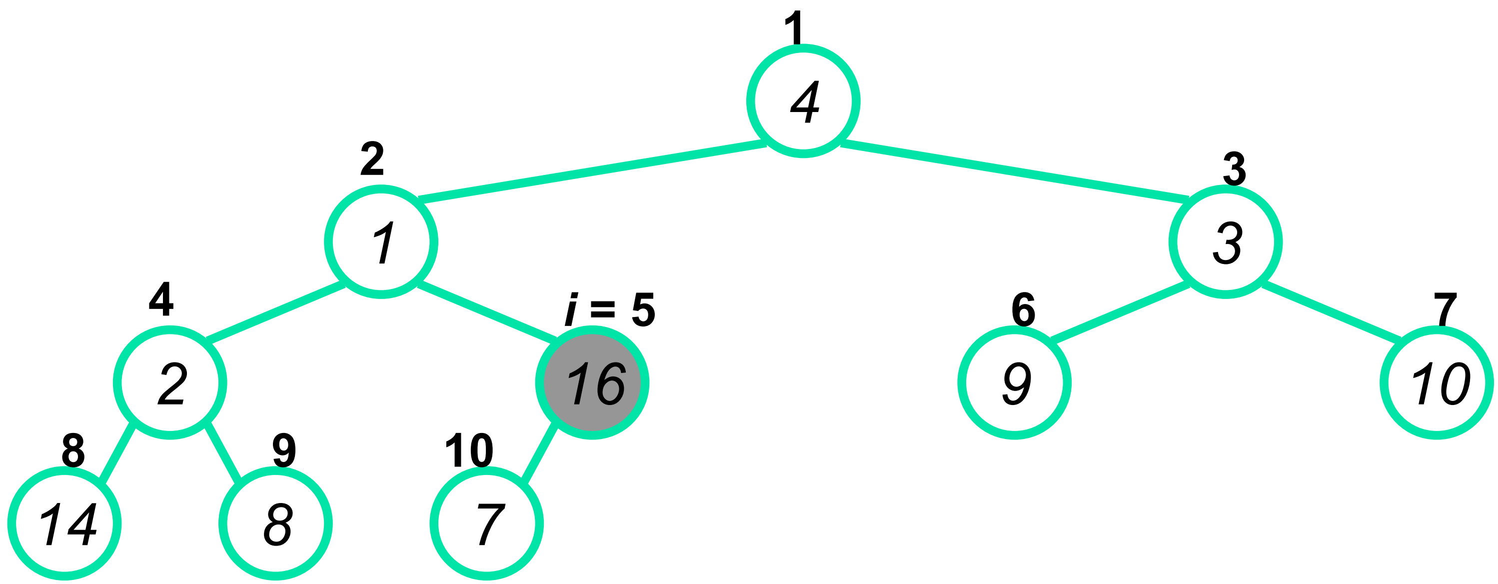
Build-Max-Heap Example

- Work through example
  A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}
- n=10, n/2=5



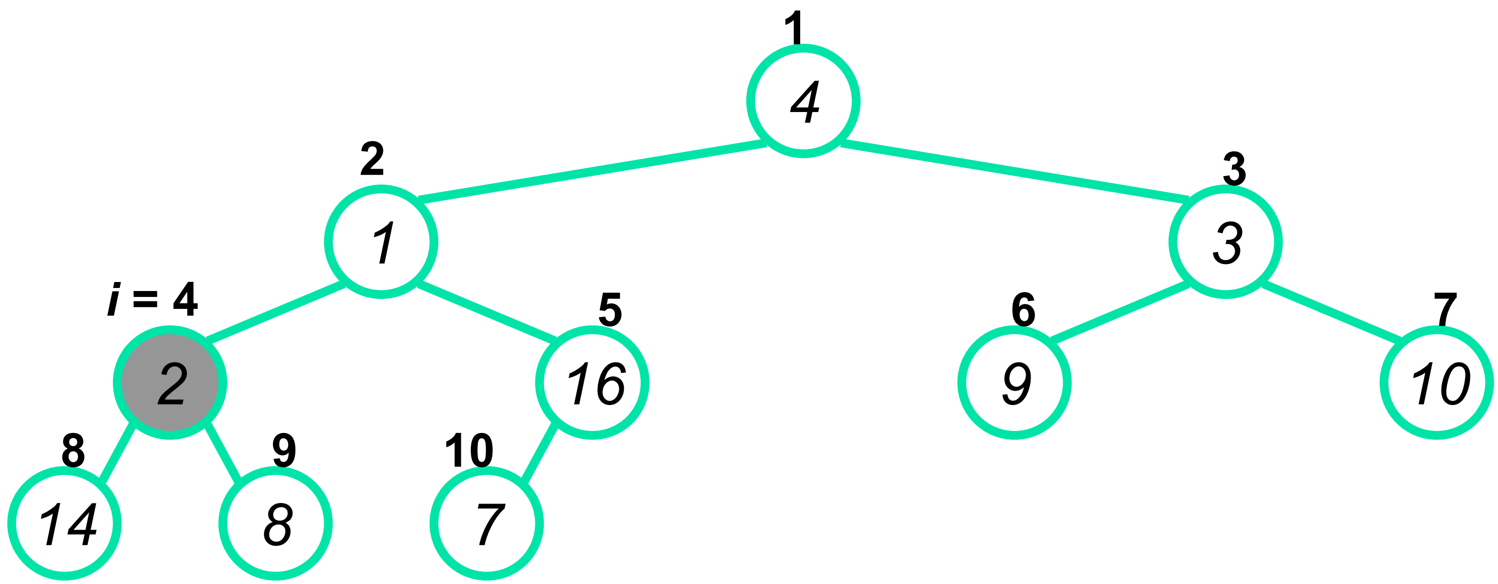
Build-Max-Heap Example

- A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}



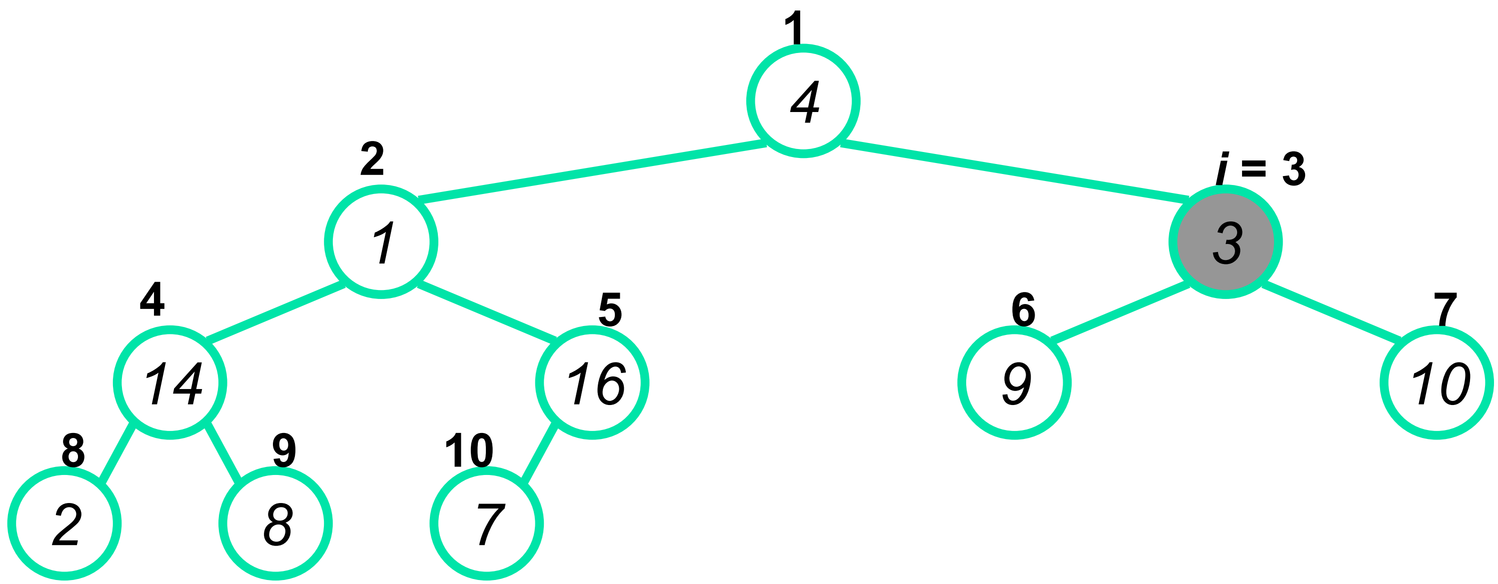
Build-Max-Heap Example

- A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}



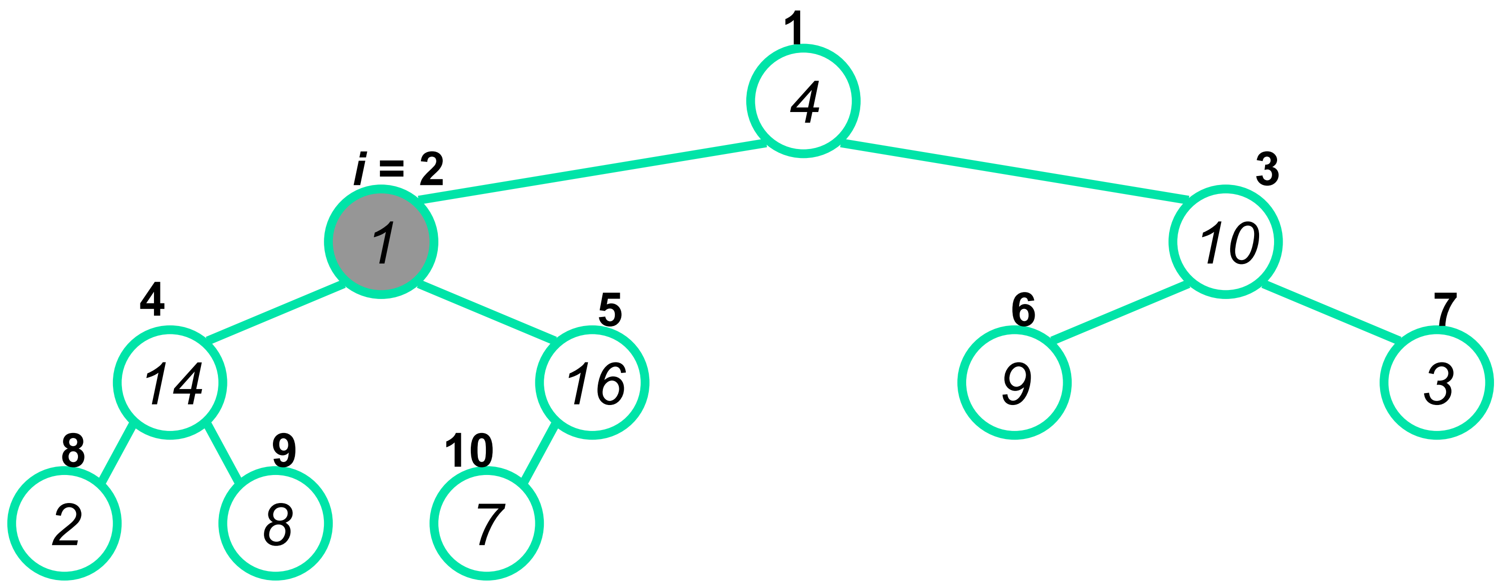
Build-Max-Heap Example

- A = {4, 1, 3, **14**, 16, 9, 10, **2**, 8, 7}



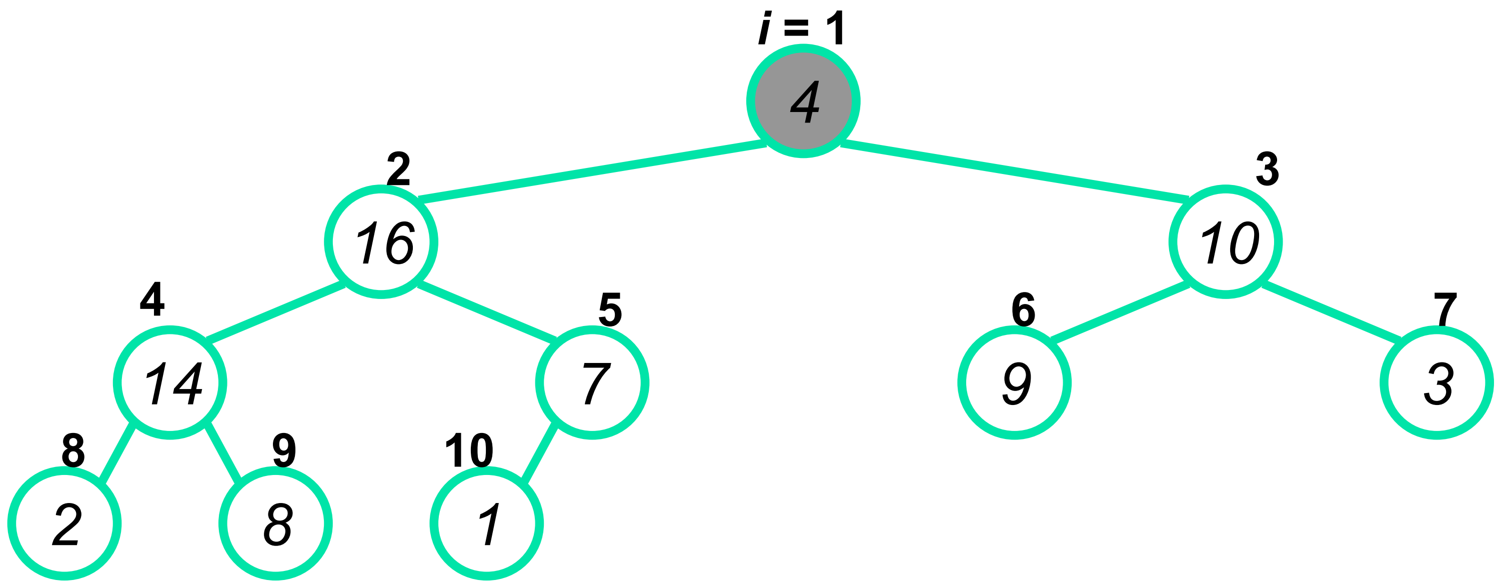
Build-Max-Heap Example

- A = {4, 1, **10**, 14, 16, 9, **3**, 2, 8, 7}



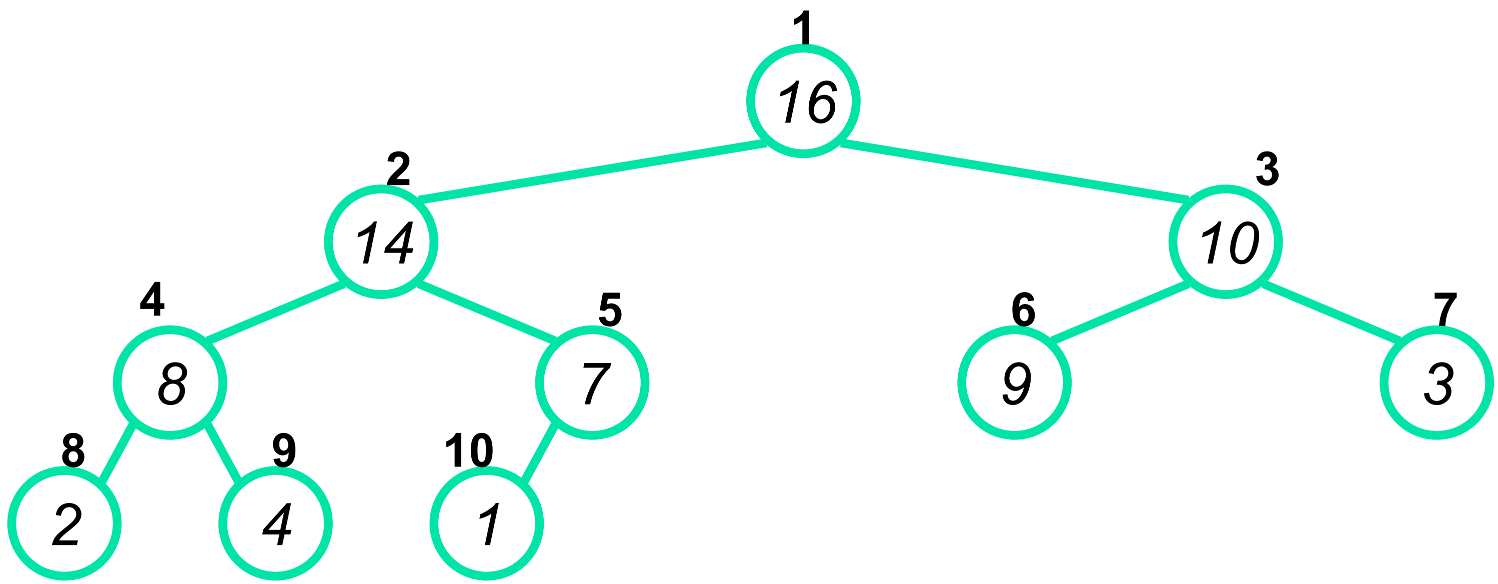
Build-Max-Heap Example

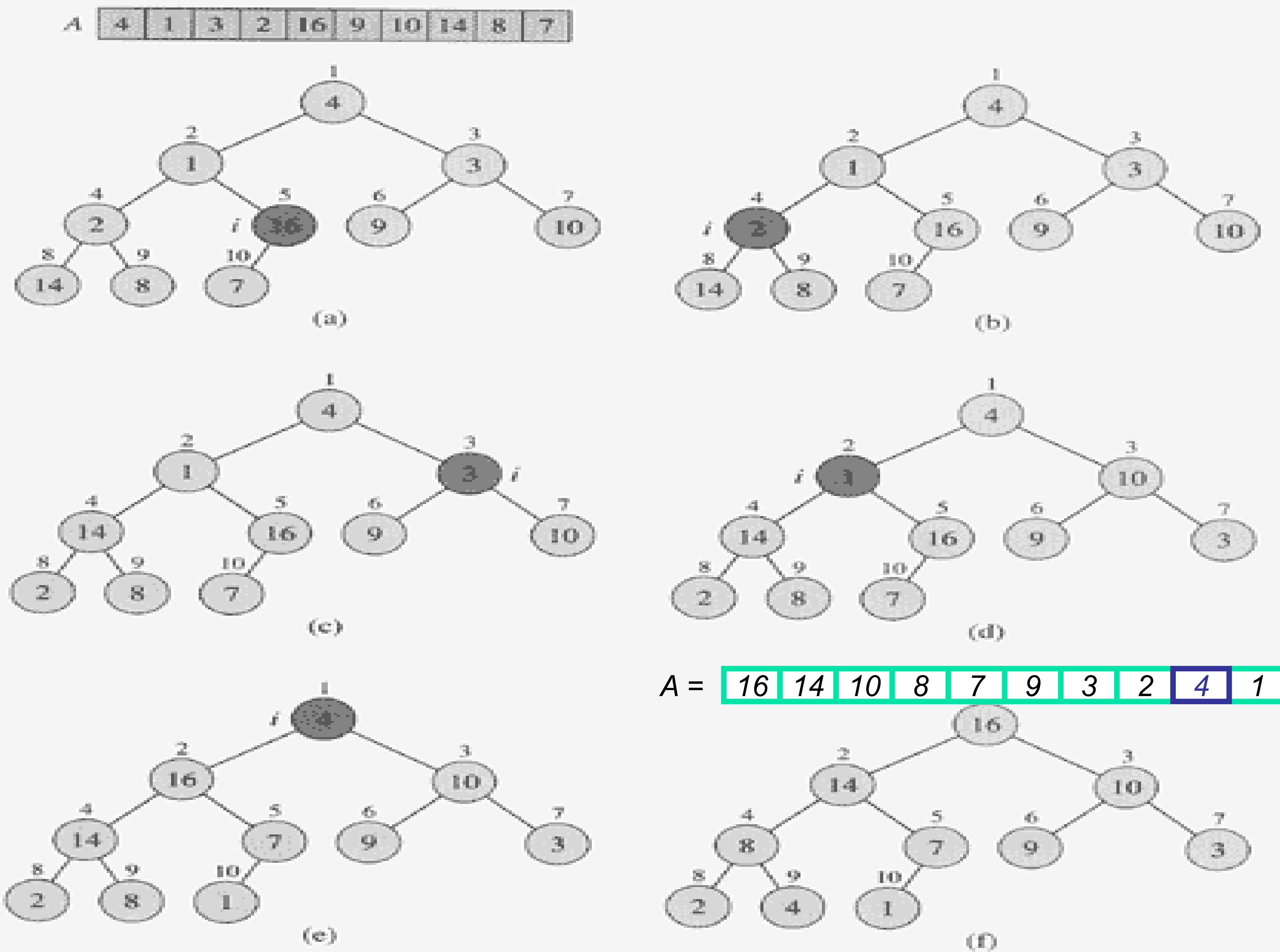
- A = {4, **16**, 10, 14, **7**, 9, 3, 2, 8, **1**}



Build-Max-Heap Example

- A = {**16**, **14**, 10, **8**, 7, 9, 3, 2, **4**, 1}




A 4 1 3 2 16 9 10 14 8 7
(a)
(b)
(c)
(d)
(e)
A = 16 14 10 8 7 9 3 2 4 1
(f)

Analyzing Build-Max-Heap (1/2)

- Each call to **`MAX-HEAPIFY`** takes O(lg *n*) time
- There are O(*n*) such calls (specifically, $\lfloor n/2 \rfloor$)
- Thus the running time is O(*n* lg *n*)
  - *Is this a correct asymptotic upper bound?*
    - *YES*
  - *Is this an* ***asymptotically tight bound?***
    - *NO*
- A tighter bound is O(*n*)
  - *How can this be? Is there a flaw in the above reasoning?*
  - *We can derive a tighter bound by observing that the time for MAX-HEAPIFY to run at a node* **varies** *with the height of the node in the tree, and the heights of most nodes are small.*
- Fact: an *n*-element heap has at most 2^{h-k} nodes of level k, where h is the height of the tree.

Analyzing Build-Max-Heap (2/2)

- The time required by *MAX-HEAPIFY* on a node of height k is $O(k)$. So we can express the total cost of *Build-Max-Heap* as

$$\sum_{k=0 \text{ to } h} 2^{h-k} O(k) = O(2^h \sum_{k=0 \text{ to } h} k/2^k)$$
$$= O(n \sum_{k=0 \text{ to } h} k(½)^k)$$

From: $\sum_{k=0 \text{ to } \infty} k\, x^k = x/(1-x)^2$ *where* $x = 1/2$

So, $\sum_{k=0 \text{ to } \infty} k/2^k = (1/2)/(1 - 1/2)^2 = 2$

Therefore, $O(n \sum_{k=0 \text{ to } h} k/2^k) = O(n)$

- So, we can bound the running time for building a heap from an unordered array in linear time

IA, P-135

The HeapSort Algorithm

Heapsort

- Given Build-Max-Heap an in-place sorting algorithm is easily constructed:
  - Maximum element is at A[1]
  - Discard by swapping with element at A[n]
    - Decrement heap_size[A]
    - A[n] now contains correct value
  - Restore heap property at A[1] by calling MAX-HEAPIFY
  - Repeat, always swapping A[1] for A[heap_size(A)]

Heapsort

HEAPSORT(A)

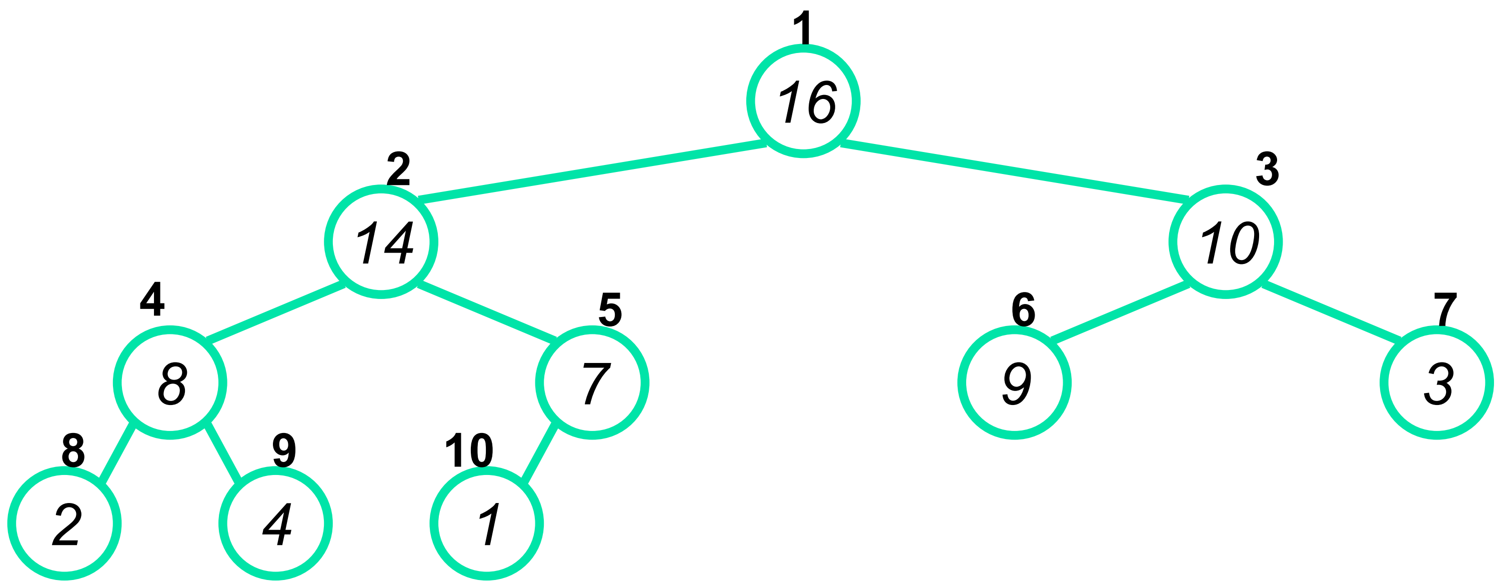
{

1. Build-MAX-Heap(A)
2. for $i \leftarrow length[A]$ downto 2
3. do exchange $A[1] \leftrightarrow A[i]$
4. $heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1$
5. MAX- Heapify(A, 1)

}

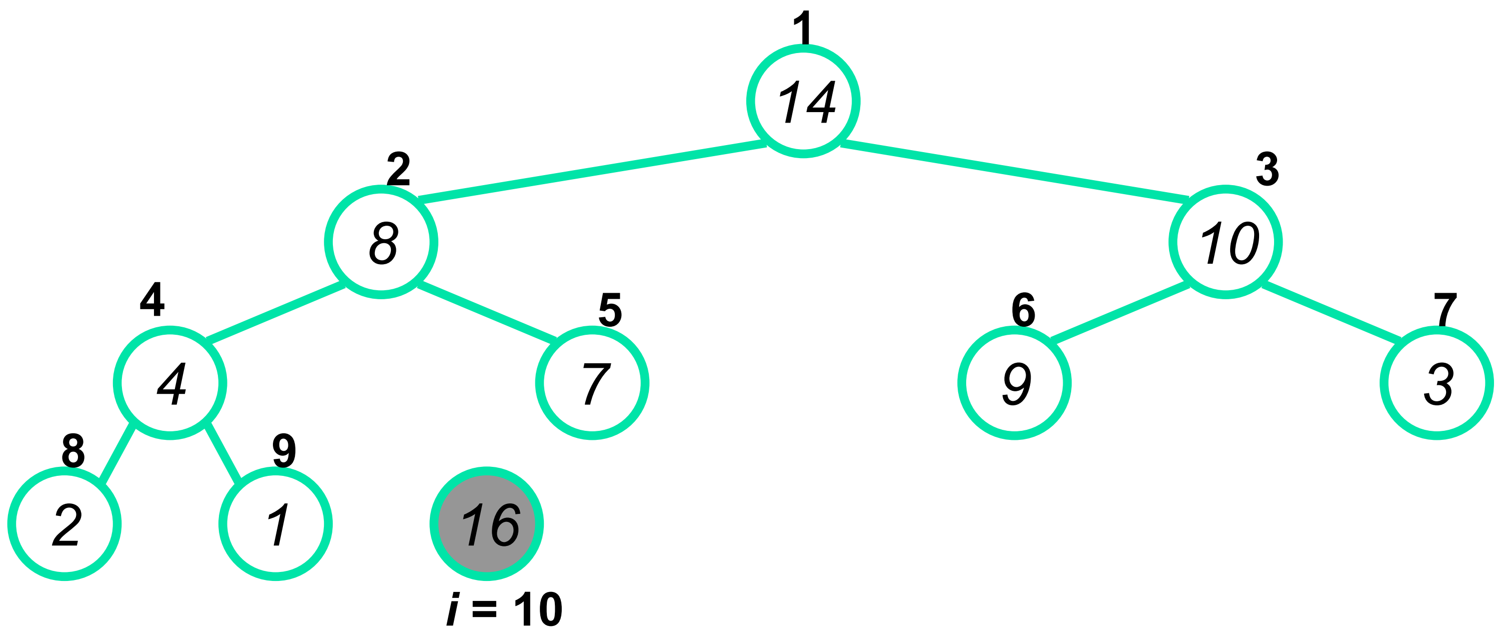
HeapSort Example

- A = {16, 14, 10, 8, 7, 9, 3, 2, 4, 1}



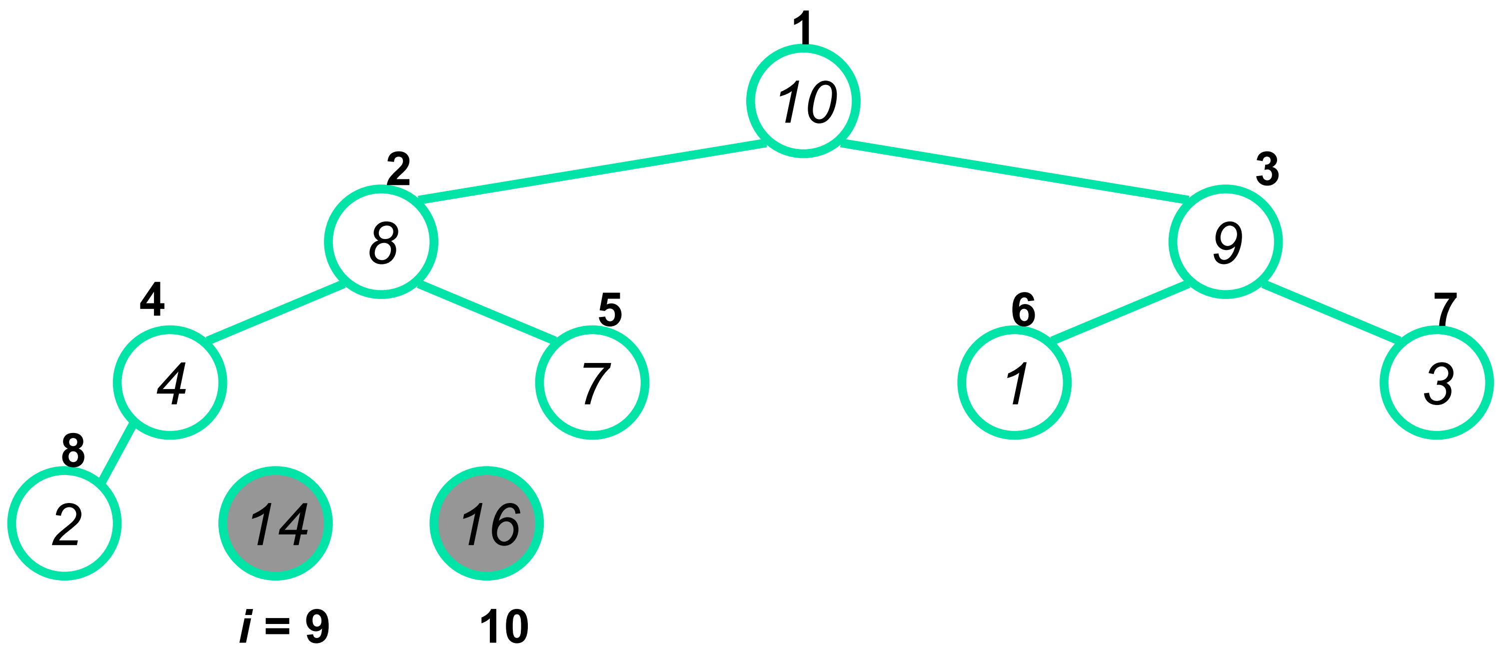
HeapSort Example

- A = {14, 8, 10, 4, 7, 9, 3, 2, 1, **16**}



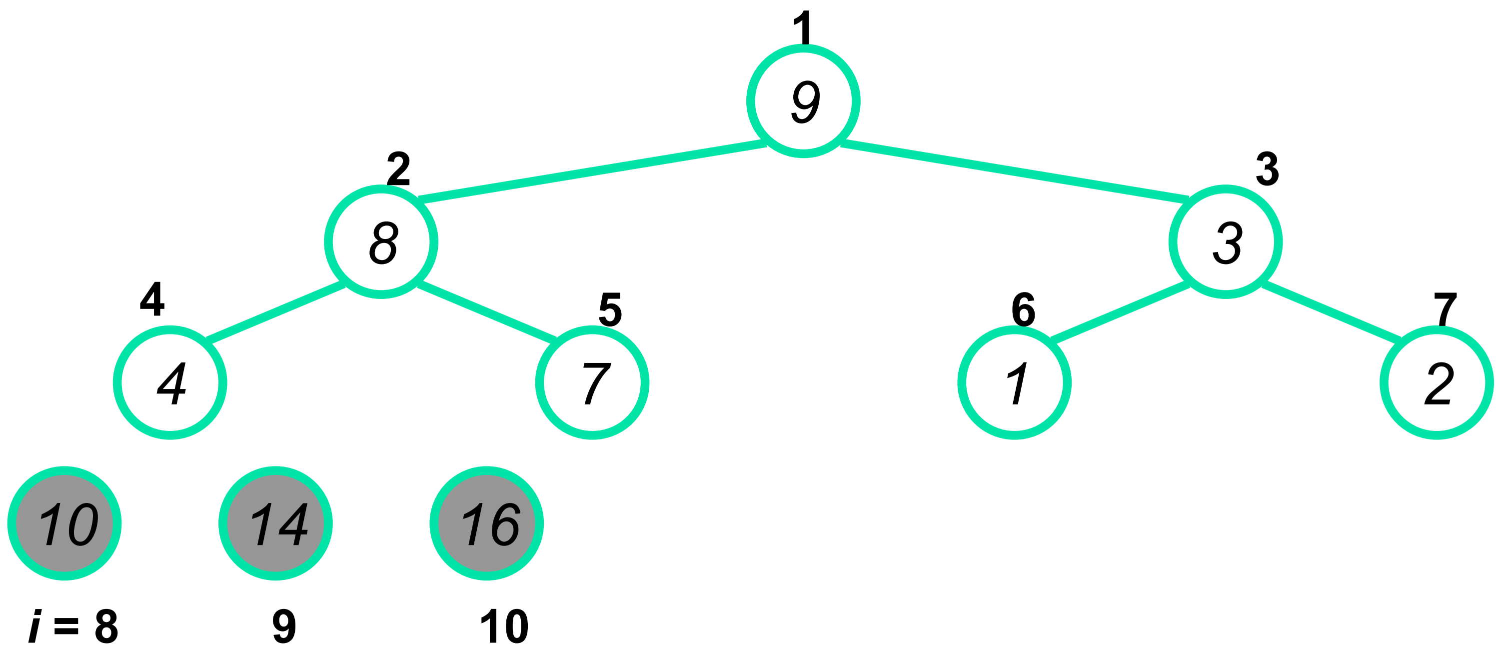
HeapSort Example

- A = {10, 8, 9, 4, 7, 1, 3, 2, **14**, **16**}



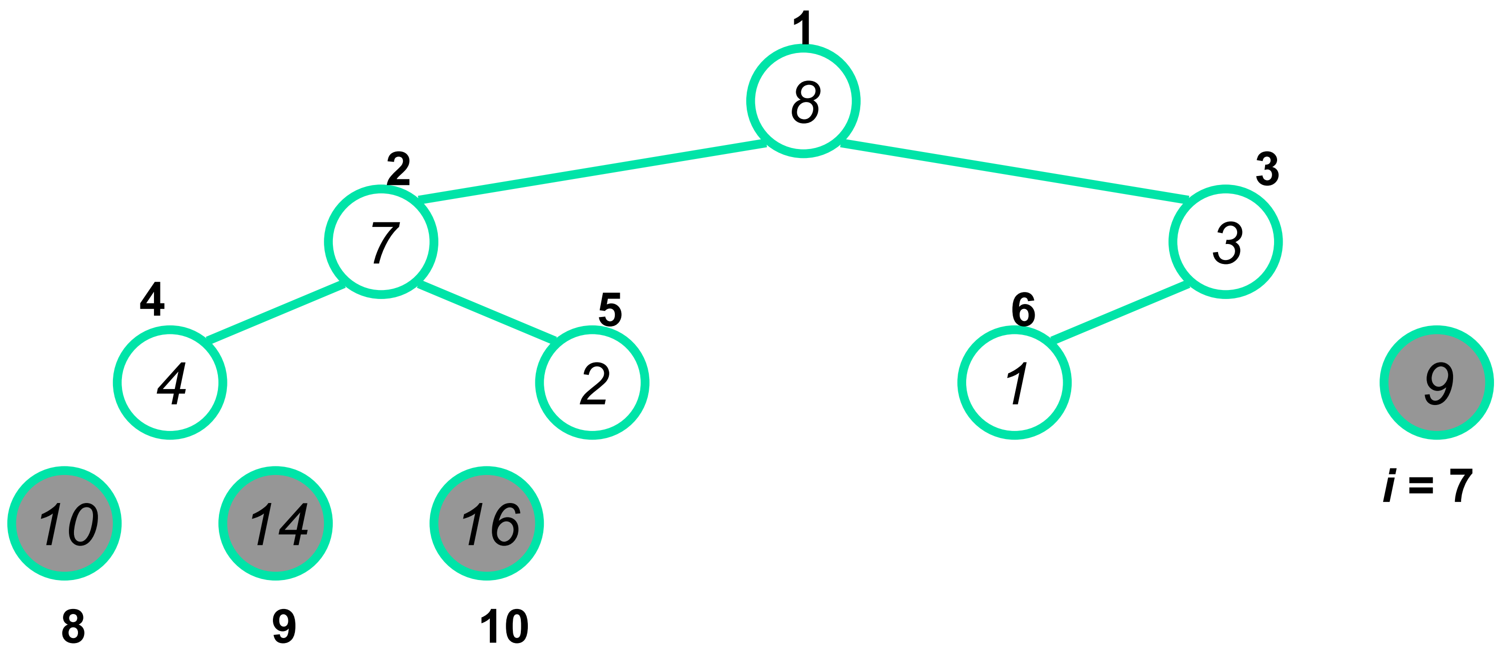
HeapSort Example

- A = {9, 8, 3, 4, 7, 1, 2, **10**, **14**, **16**}



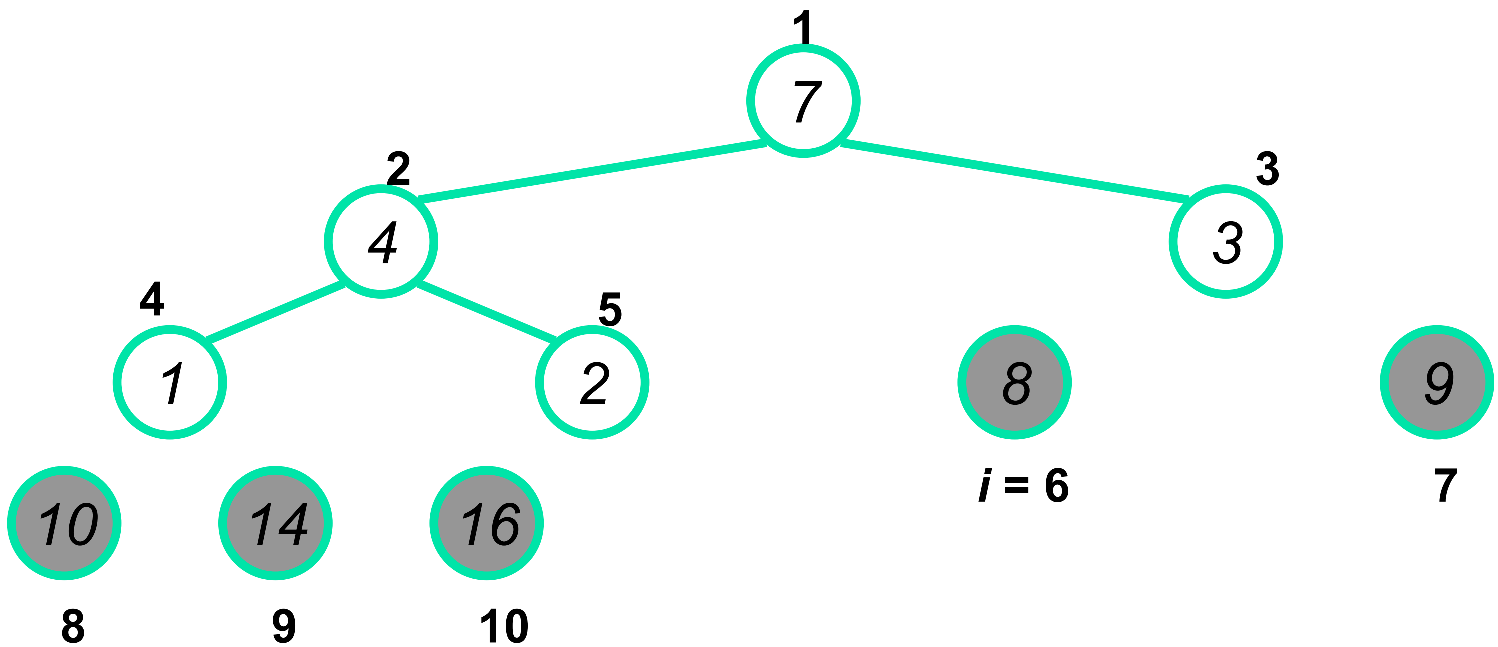
HeapSort Example

- A = {8, 7, 3, 4, 2, 1, **9**, **10**, **14**, **16**}



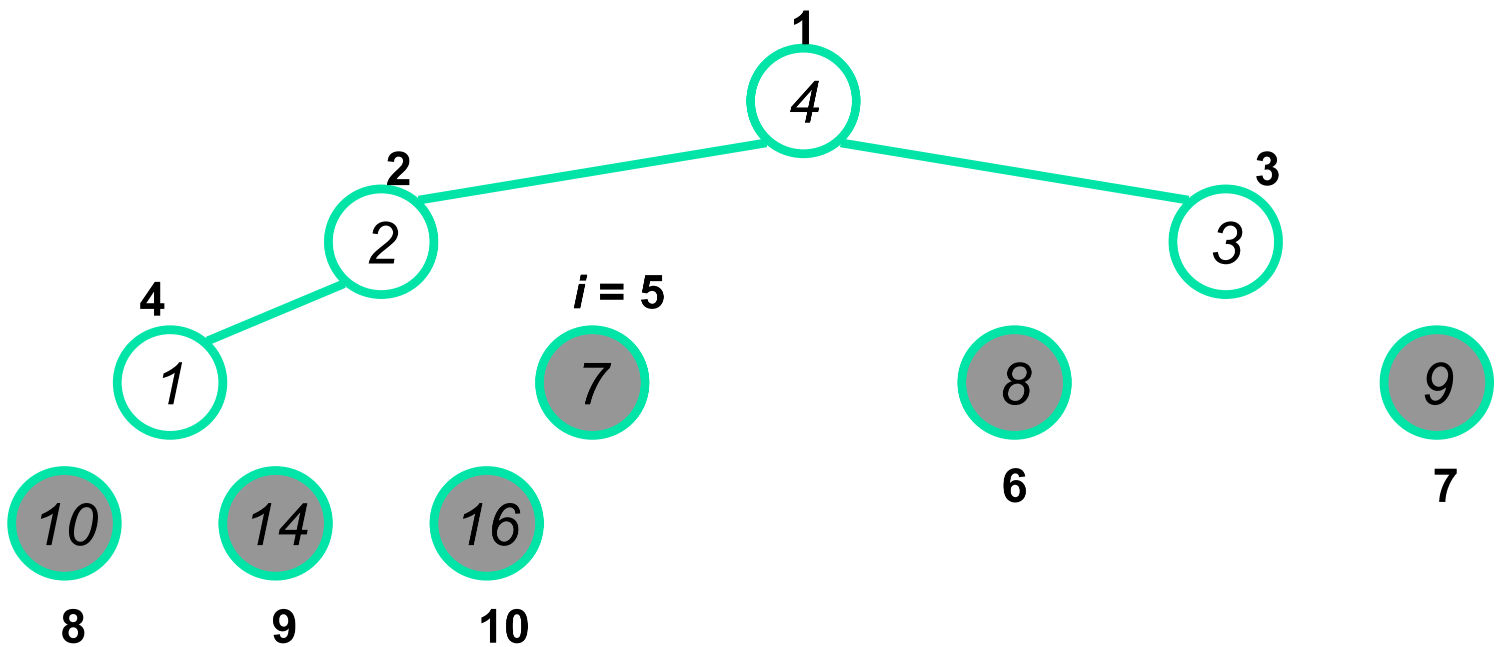
HeapSort Example

- A = {7, 4, 3, 1, 2, **8, 9, 10, 14, 16**}



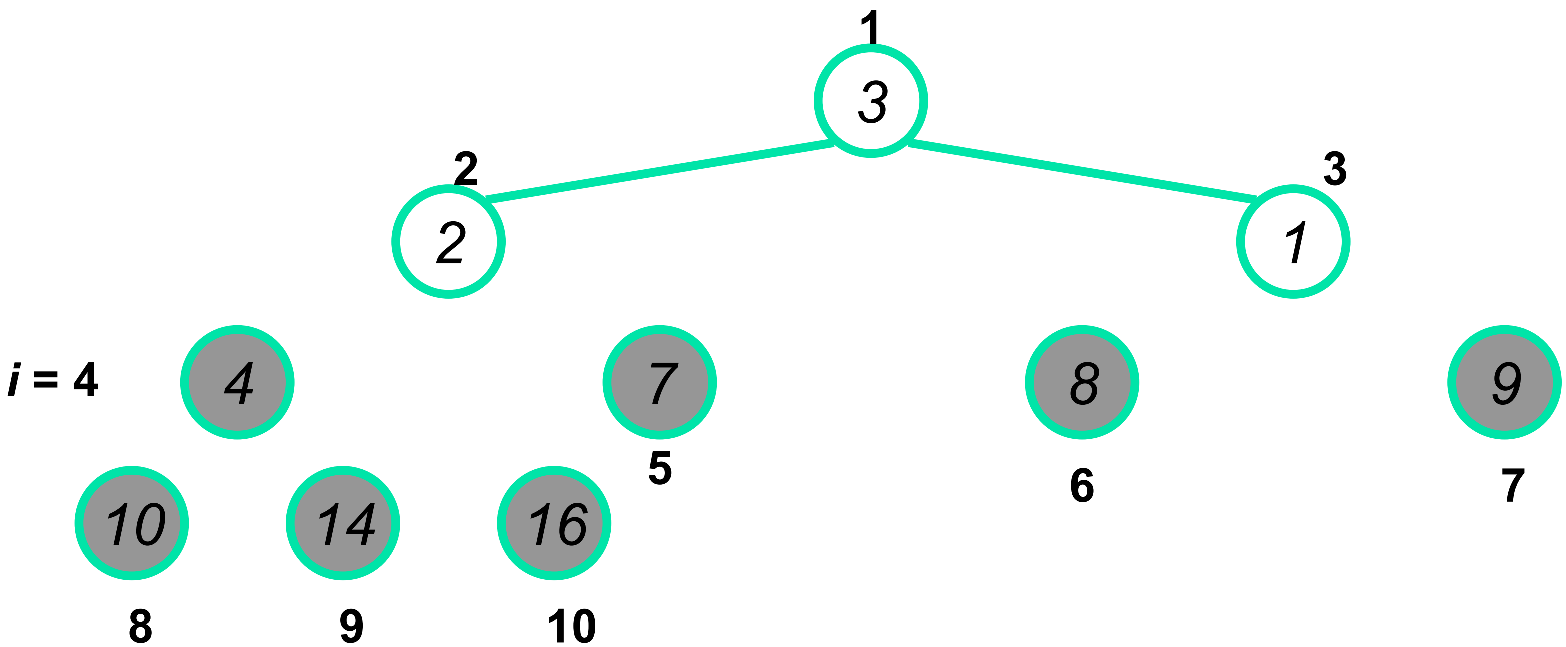
HeapSort Example

- A = {4, 2, 3, 1, **7**, **8**, **9**, **10**, **14**, **16**}



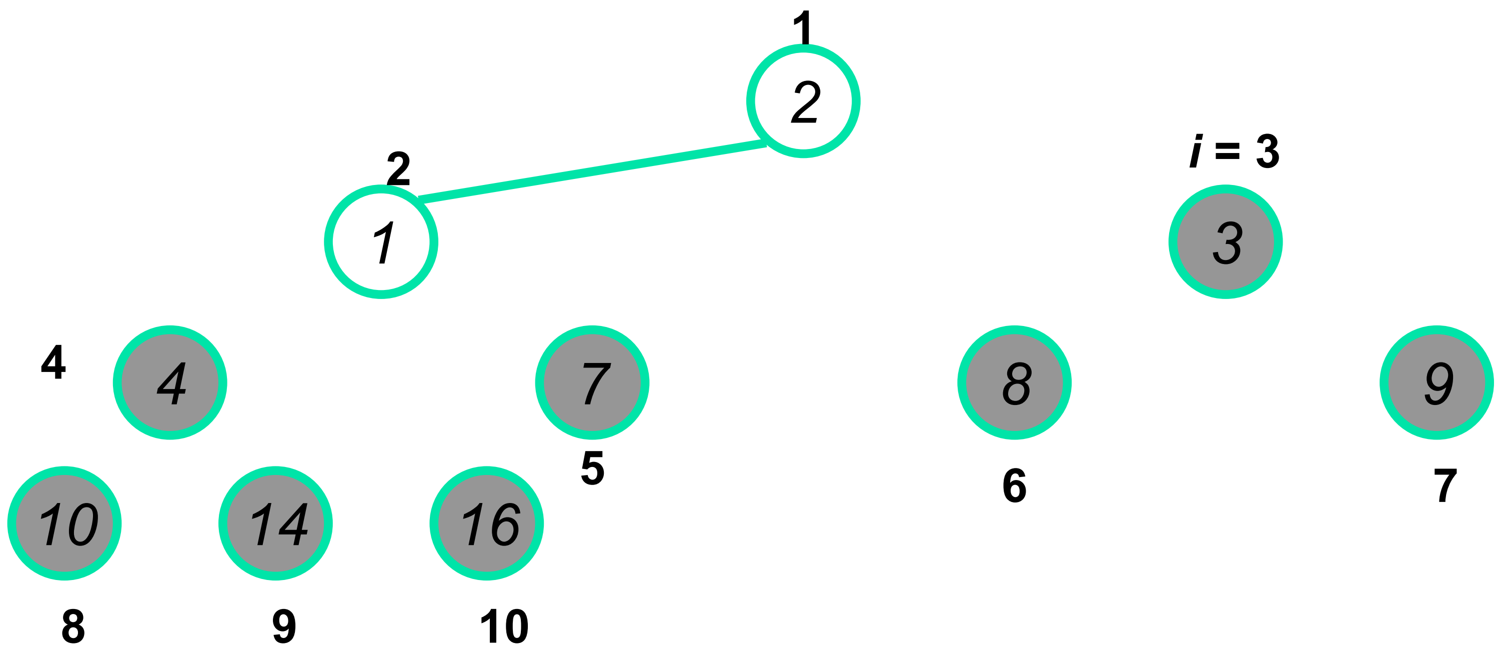
HeapSort Example

- A = {3, 2, 1, **4**, **7**, **8**, **9**, **10**, **14**, **16**}



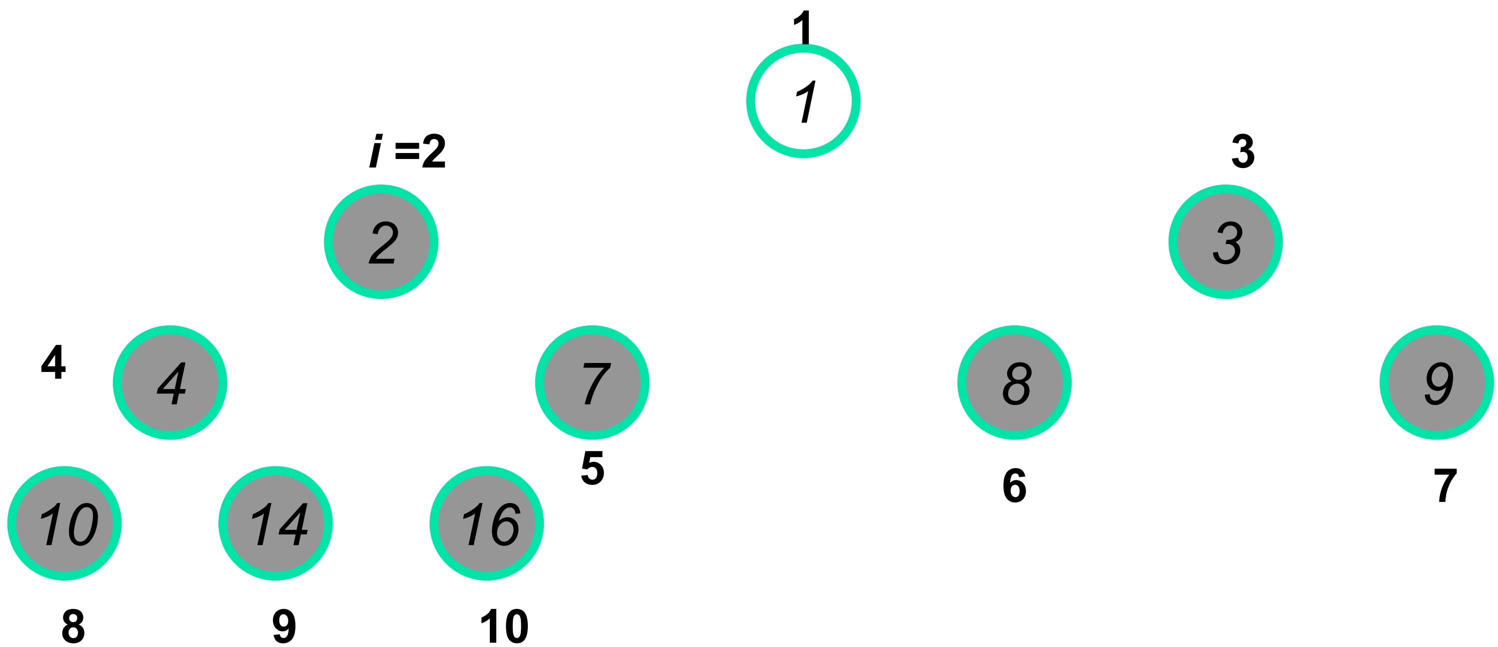
HeapSort Example

- A = {2, 1, **3**, **4**, **7**, **8**, **9**, **10**, **14**, **16**}



HeapSort Example

- A = {1, **2, 3, 4, 7, 8, 9, 10, 14, 16**}



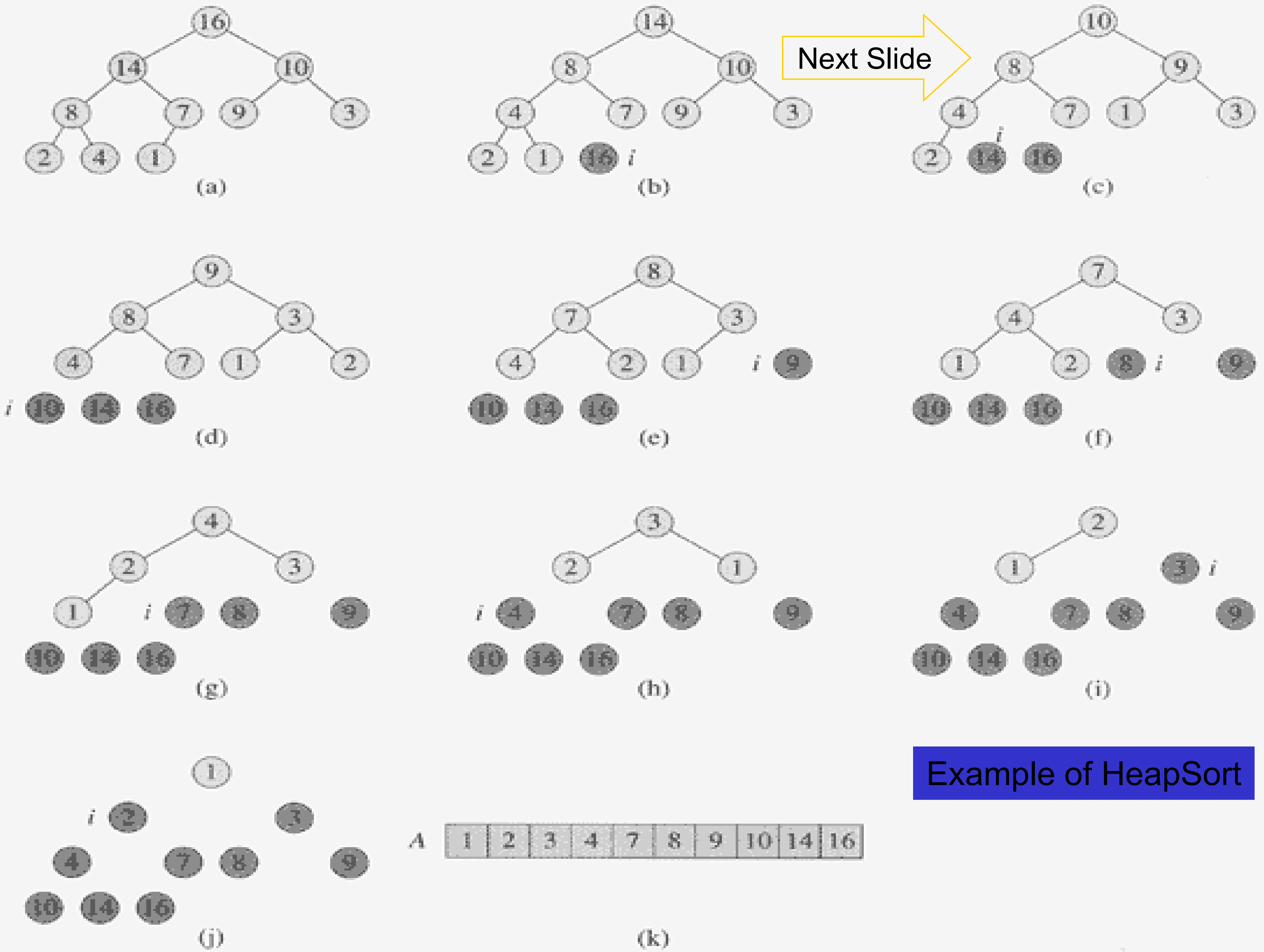
Analyzing Heapsort (1/2)

- The call to **BUILD-MAX-HEAP`()`** takes $O(n)$ time
- Each of the $n - 1$ calls to **MAX- HEAPIFY()** takes $O(\lg n)$ time
- Thus the total time taken by **`HEAPSORT()`**
  $= O(n) + (n - 1)\, O(\lg n)$
  $= O(n) + O(n \lg n)$
  $= O(n \lg n)$

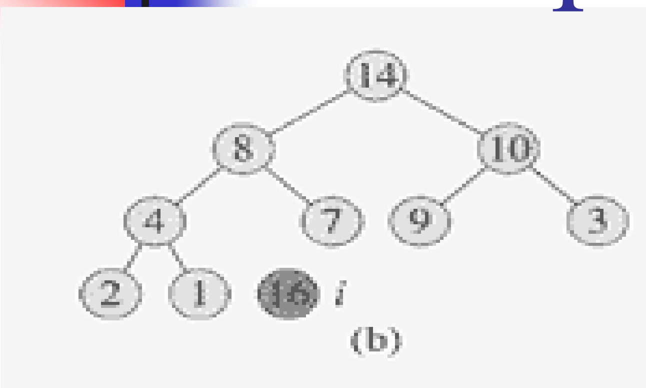
Analyzing Heapsort (2/2)

- The O(n log n) run time of heap-sort is much better than the $O(n^2)$ run time of selection and insertion sort
- Although, it has the same run time as Merge sort, but it is better than Merge Sort regarding memory space
  - **Heap sort is in-place sorting algorithm**
  - But **not stable**
    - **Does not preserve the relative order of elements with equal keys**
    - Sorting algorithm (stable) if 2 records with same key stay in original order

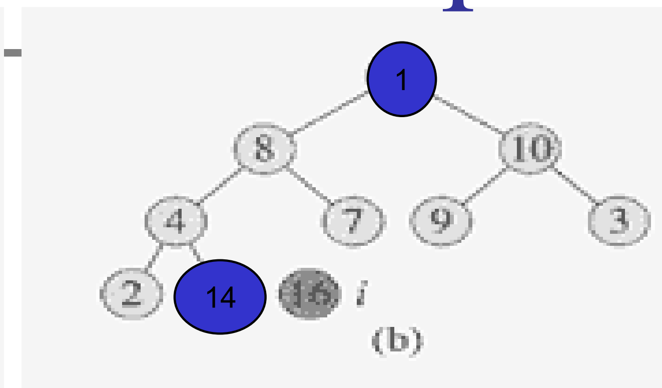
Example of HeapSort



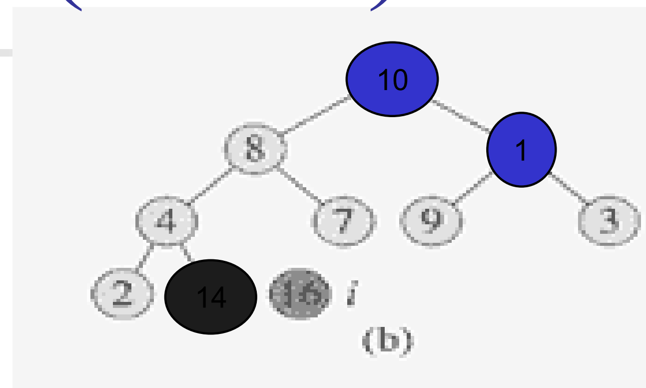
Example of HeapSort (Cont.)



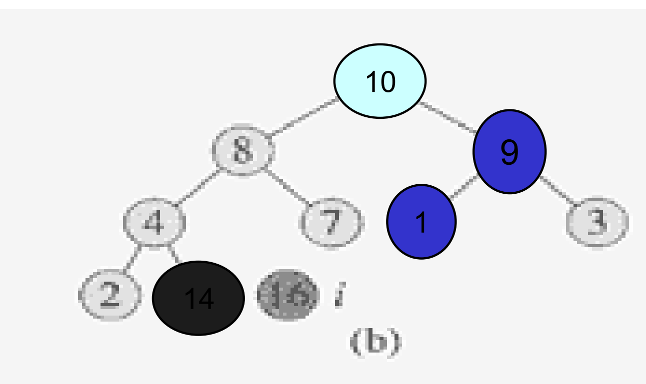
(b)



(b)



(b)



(b)

Priority Queues

Max-Priority Queues

- A data structure for maintaining a set *S* of elements, each with an associated value called a *key*.
- Applications:
  - scheduling jobs on a shared computer
  - prioritizing events to be processed based on their predicted time of occurrence.
  - Printer queue
- Heap can be used to implement a max-priority queue

Max-Priority Queue: Basic Operations

- Maximum(S): $\longrightarrow$ $return\ A[1]$
  - returns the element of S with the largest key (value)
- Extract-Max(S):
  - removes and returns the element of S with the largest key
- Increase-Key(S, x, k):
  - increases the value of element x's key to the new value k, $x.value \leq k$
- Insert(S, x):
  - inserts the element x into the set S, i.e. $S \rightarrow S \cup \{x\}$

HEAP-MAXIMUM

```
HEAP-MAXIMUM(A)
    return A[1]
```

$\Theta(1)$

HEAP-Extract-Max(A)

1. if *heap-size*[A] < 1 // zero elements
2. **then error** "heap underflow"
3. *max* ← A[1] // max element in first position
4. A[1] ← A[*heap-size*[A]]
   // value of last position assigned to first position
5. *heap-size*[A] ← *heap-size*[A] – 1
6. Heapify(A, 1)
7. return *max*

Running time : Dominated by the running time of MaxHeapify
= $O(\lg n)$

HEAP-Extract-MIN(A)

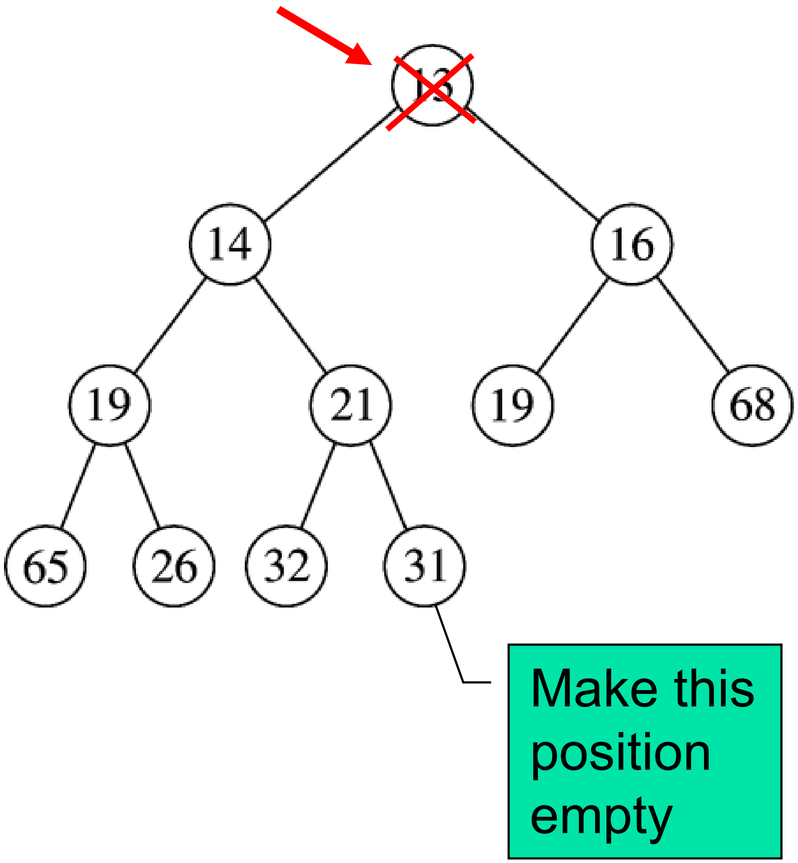
- Write a pseudo-code code similar to HEAP-Extract-MIN(A)

HEAP-Extract-MIN

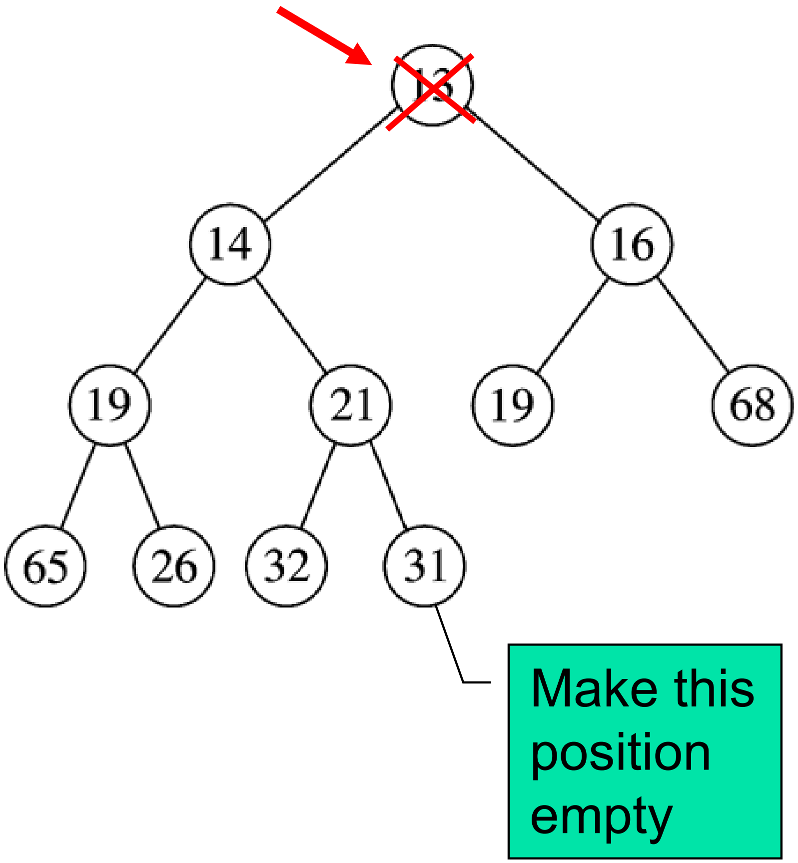
- Minimum element is always at the root in min-heap
- Heap decreases by one in size
- Move last element into hole at root
- *Percolate down* while heap-order property not satisfied

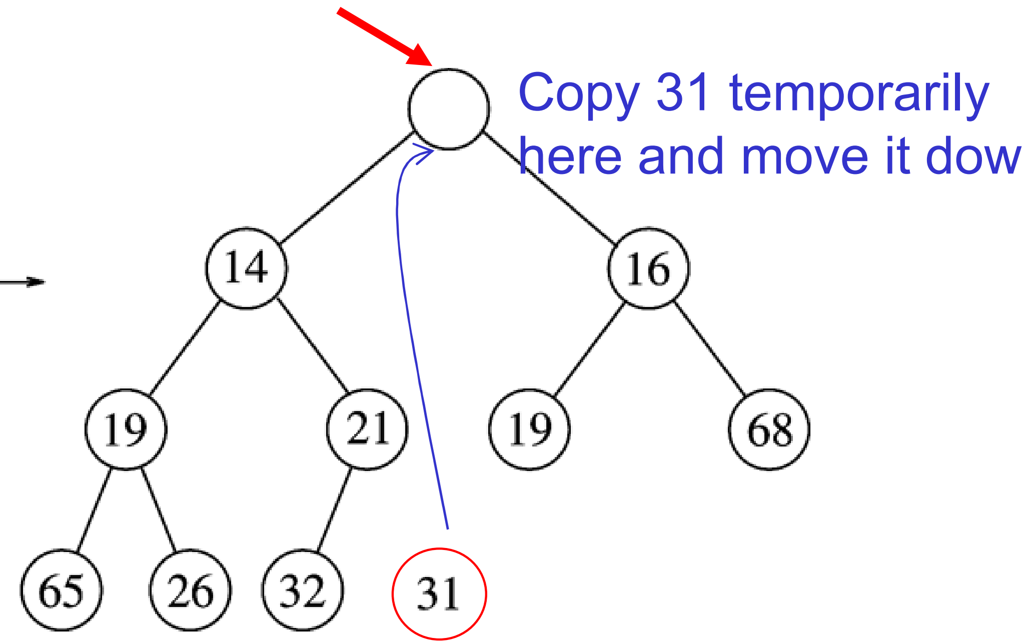
Percolating down…

HEAP-Extract-MIN : Example



HEAP-Extract-MIN : Example

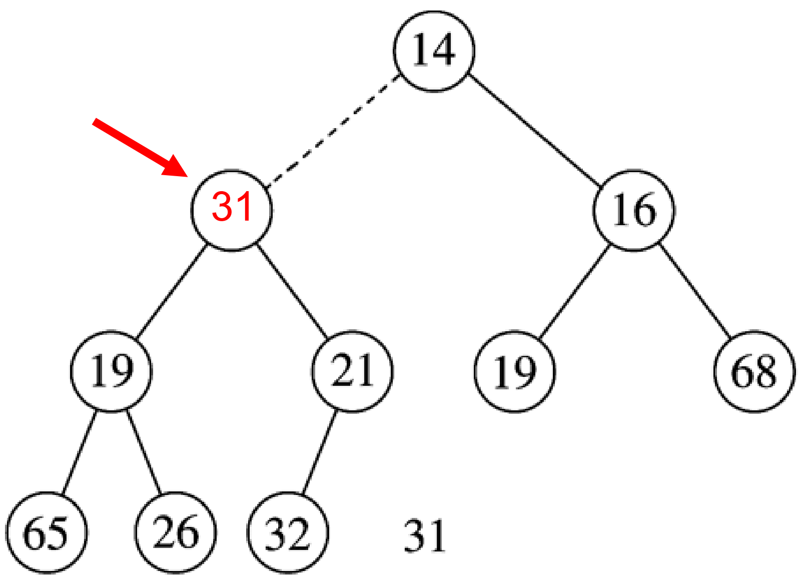




Is 31 > min(14,16)?

•Yes - swap 31 with min(14,16)

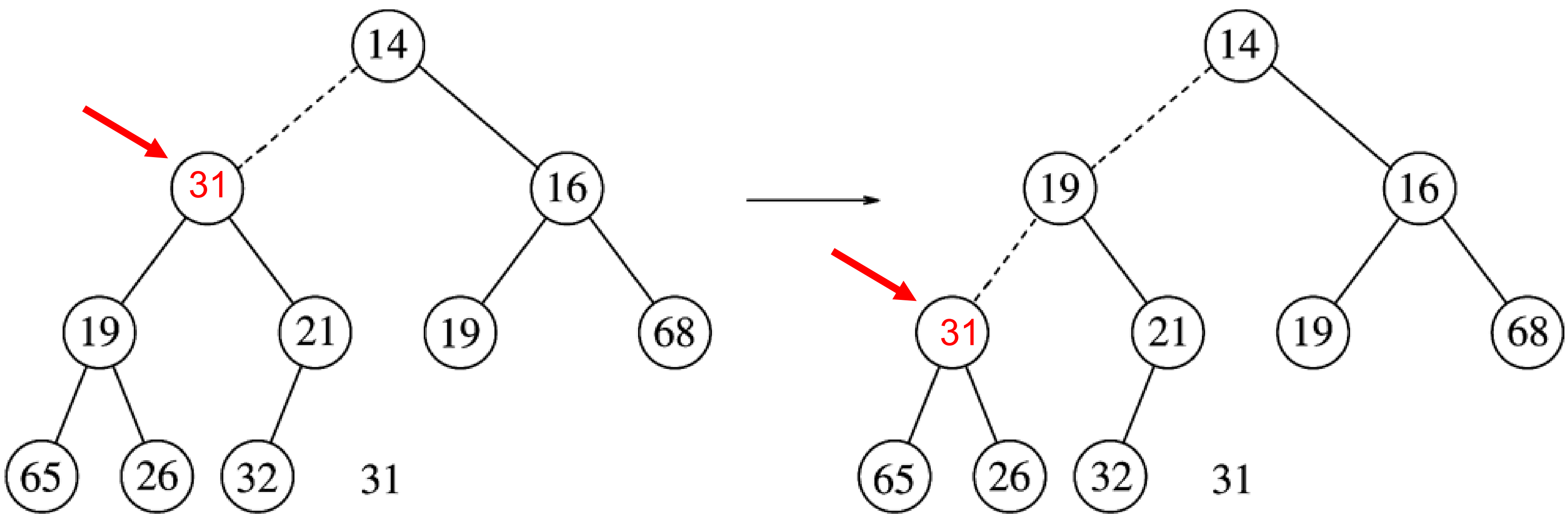
HEAP-Extract-MIN : Example



Is 31 > min(19,21)?

•Yes - swap 31 with min(19,21)

HEAP-Extract-MIN : Example

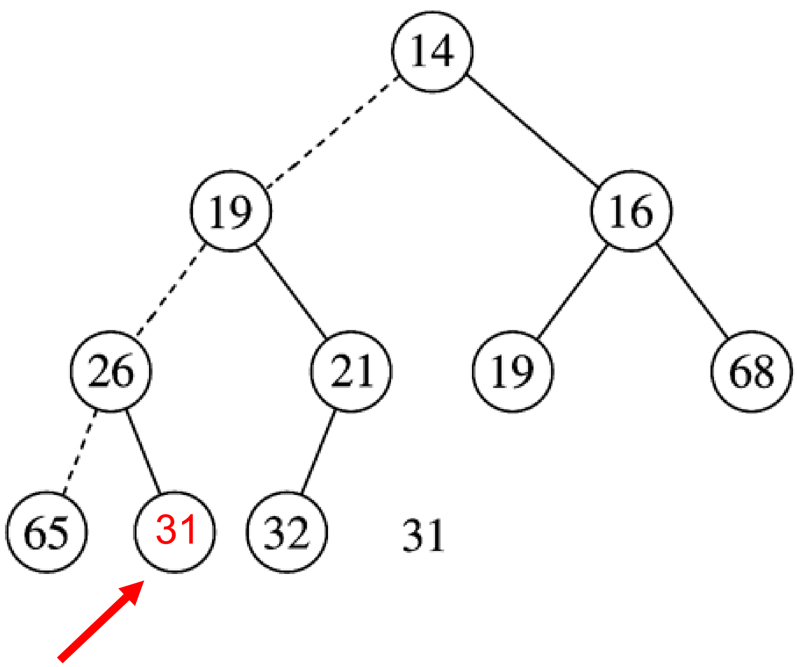


Is 31 > min(19,21)?
•Yes - swap 31 with min(19,21)

Is 31 > min(65,26)?
•Yes - swap 31 with min(65,26)

Percolating down…

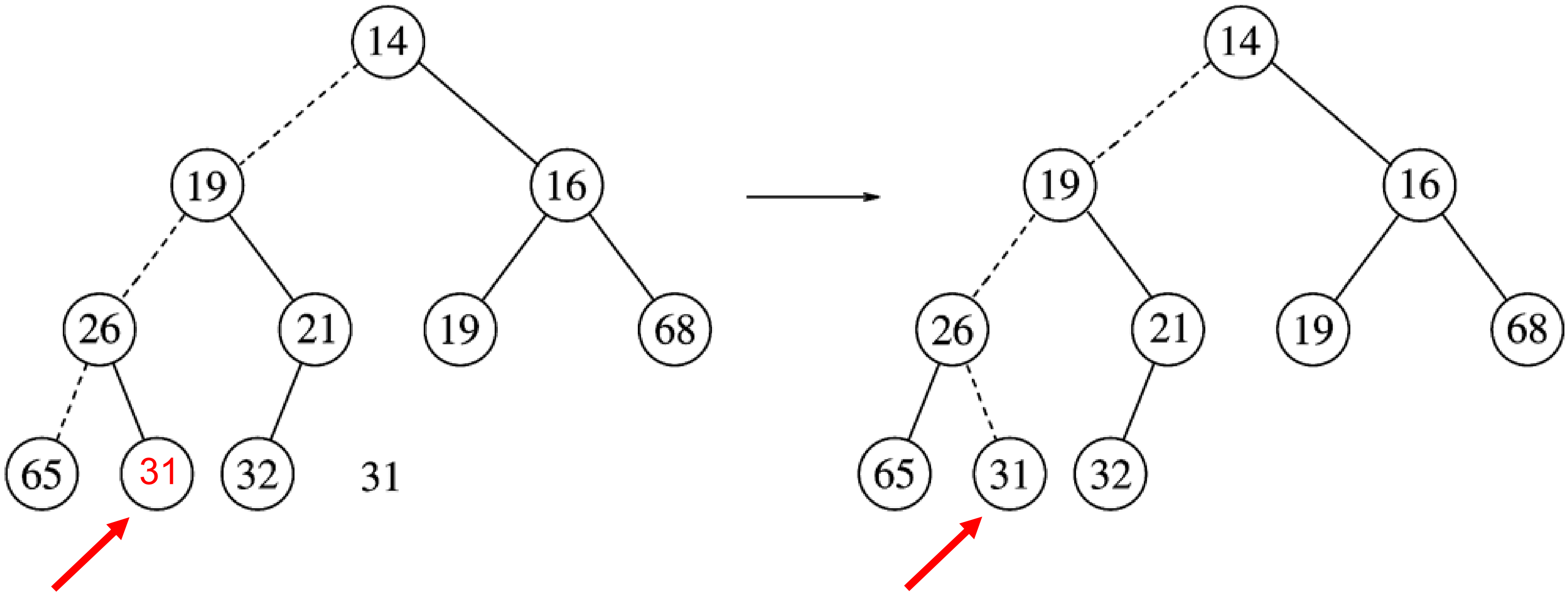
HEAP-Extract-MIN : Example



Percolating down…

Percolating down…

HEAP-Extract-MIN : Example



✓ Heap order prop

✓ Structure prop

HEAP-INCREASE-KEY

- Increase the job priority
- Steps
  - Update the key of A[i] to its new value
    - May violate the max-heap property
  - Traverse a path from A[i] toward the root to find a proper place for the newly increased key

HEAP-INCREASE-KEY(A, i, key)

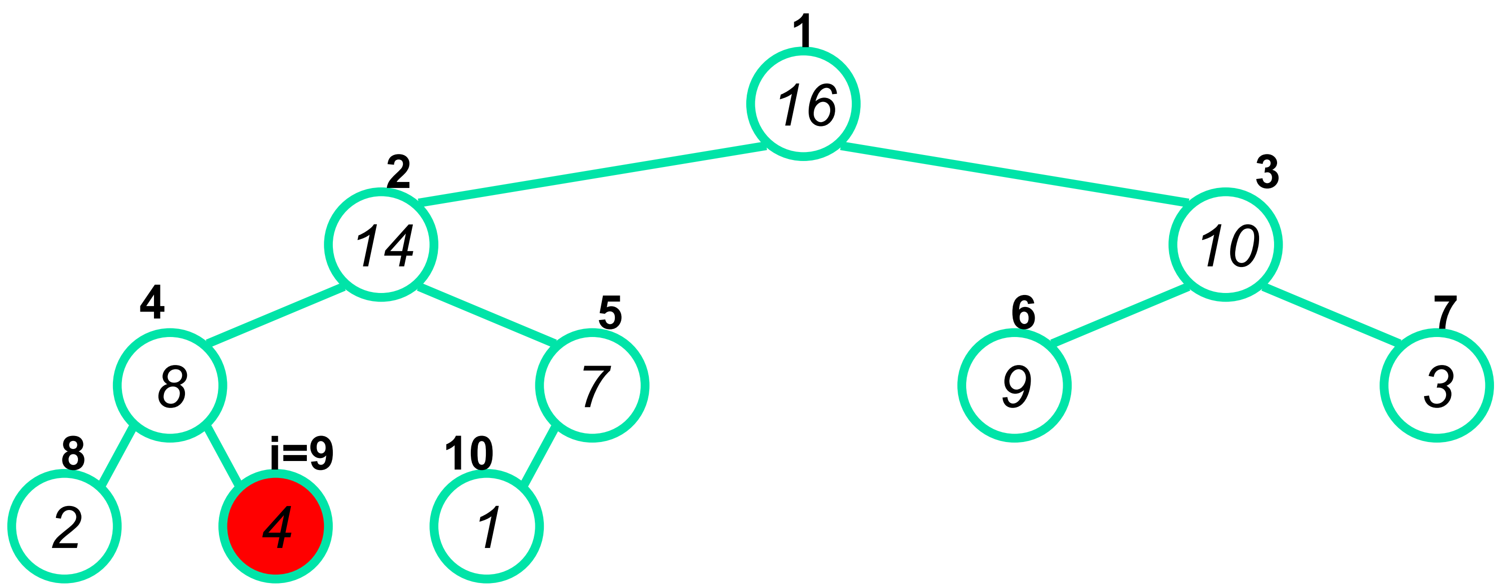
// increase a value (key) in the array

1. **if** $key < A[i]$
2. **then error** "new key is smaller than current key"
3. $A[i] \leftarrow key$
4. **while** $i > 1$ and $A[\text{Parent}(i)] < A[i]$
5. do exchange $A[i] \leftrightarrow A[\text{Parent}(i)]$
6. $i \leftarrow \text{Parent}(i)$ // move index up to parent

O(lg n)

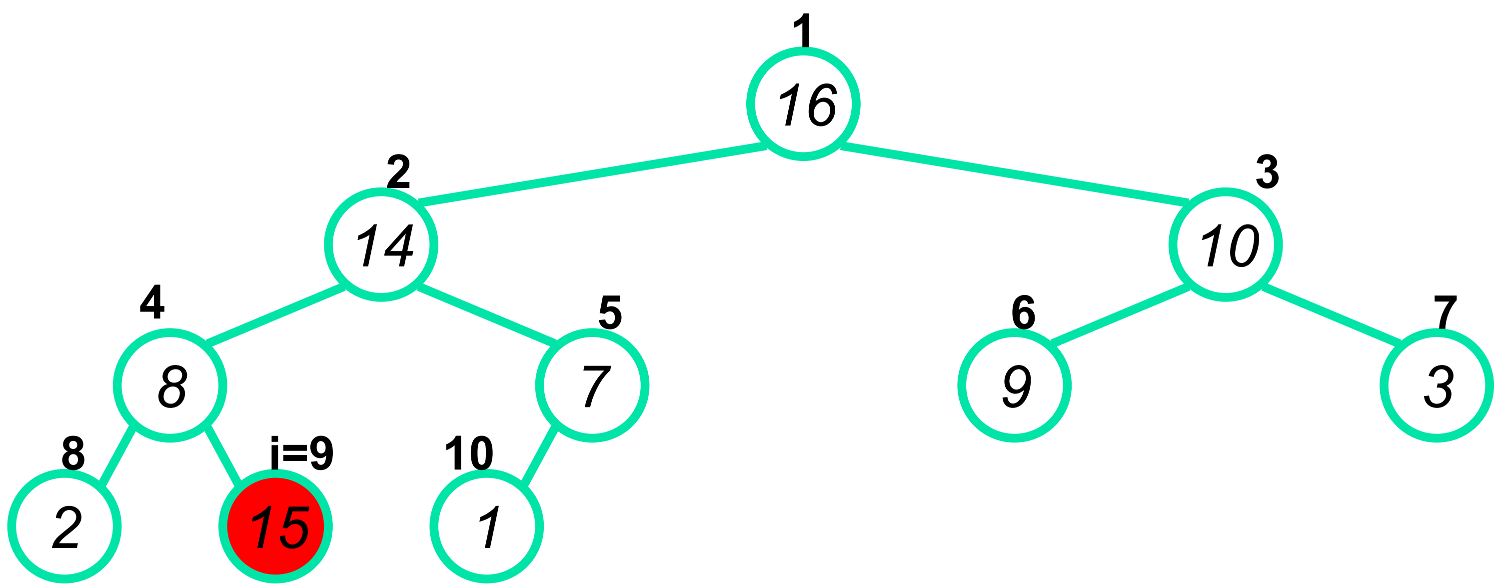
HEAP-INCREASE-KEY() Example

- A = {16, 14, 10, 8, 7, 9, 3, 2, 4, 1}



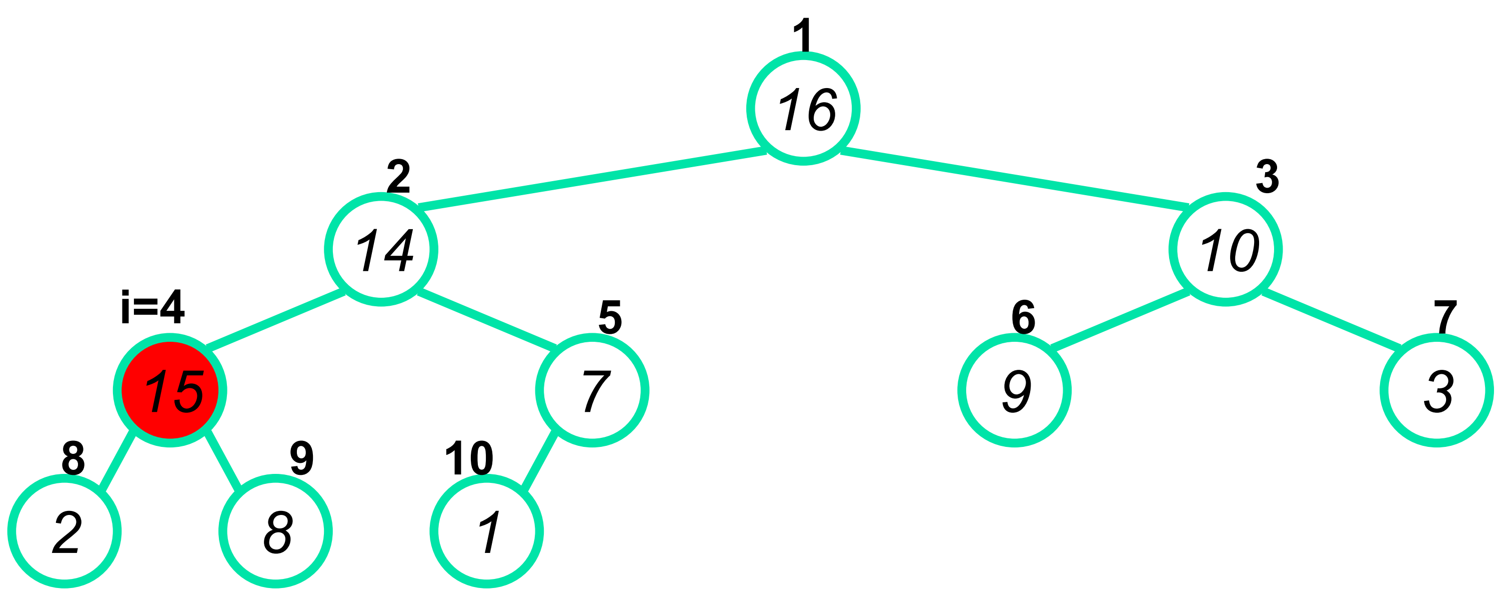
HEAP-INCREASE-KEY() Example

- A = {16, 14, 10, 8, 7, 9, 3, 2, 15, 1}
- The index i=9 increased to 15.



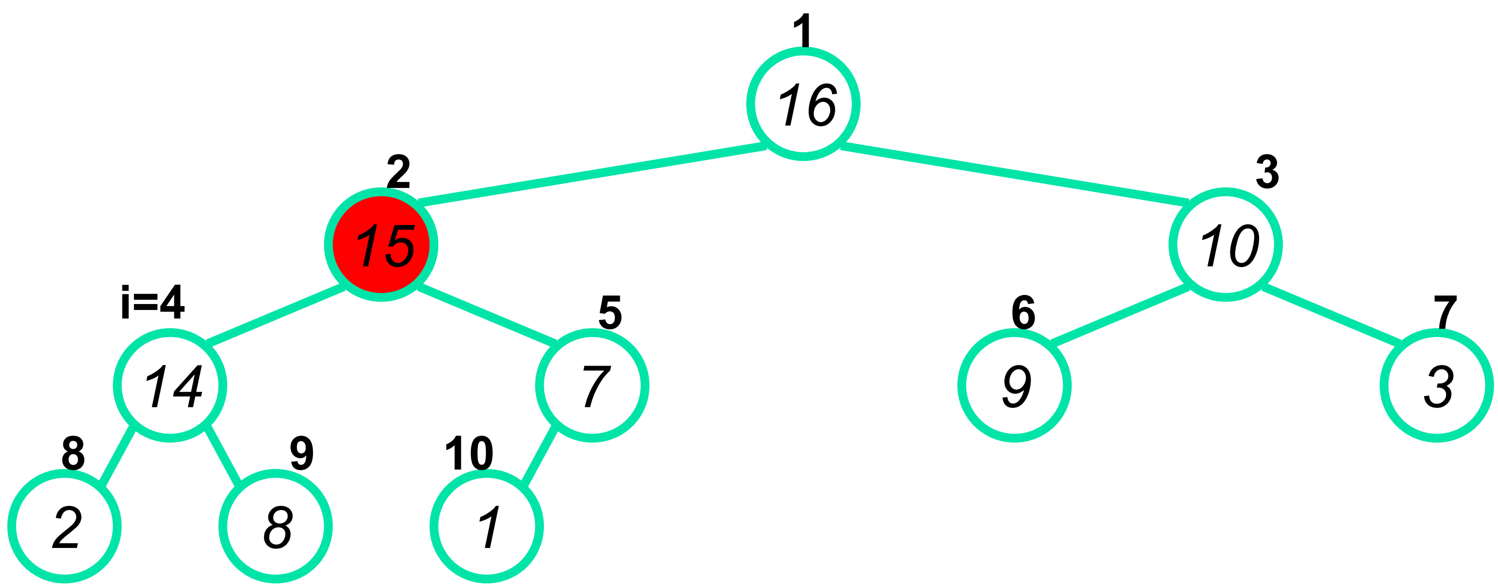
HEAP-INCREASE-KEY() Example

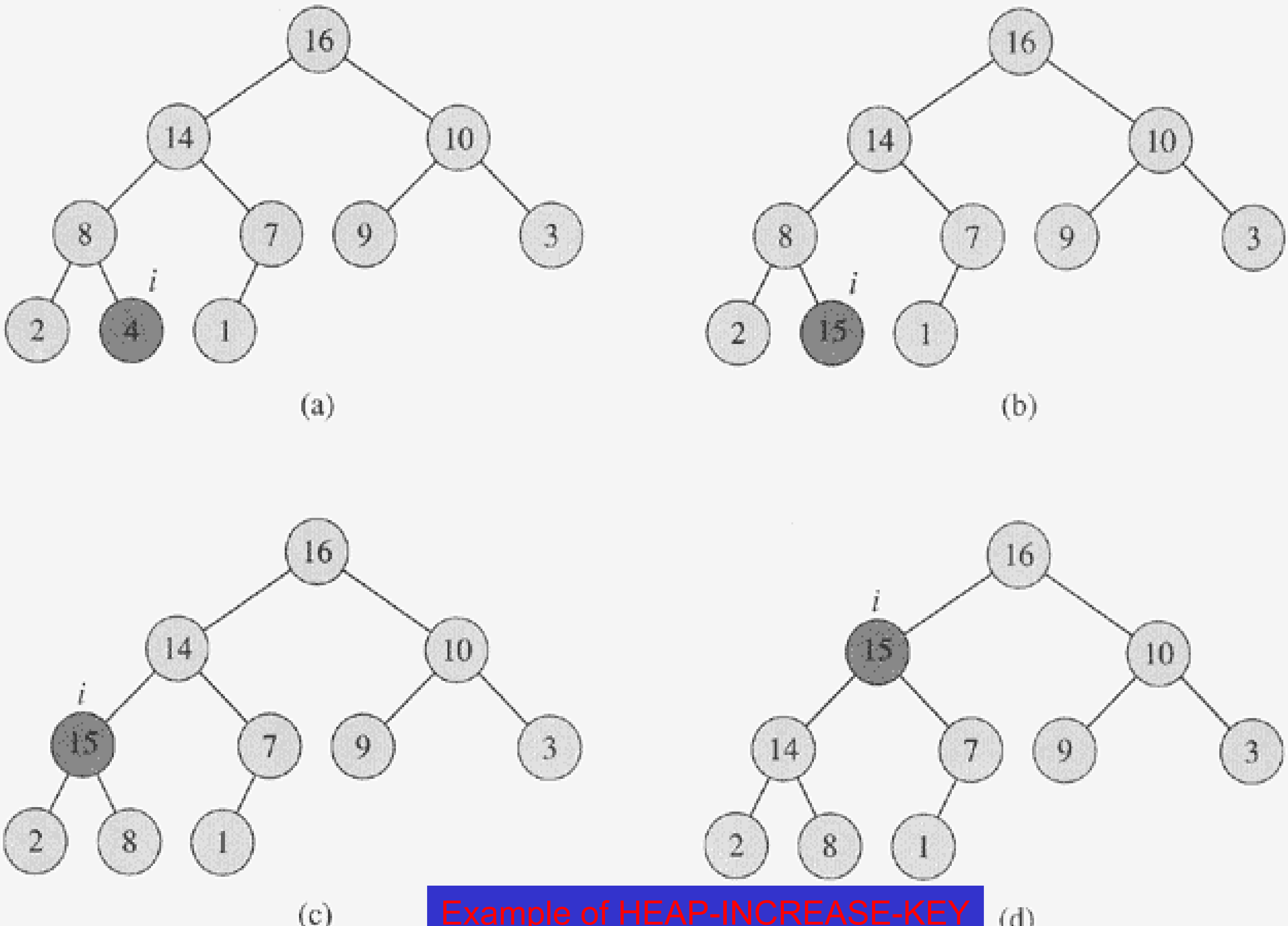
- A = {16, 14, 10, 15, 7, 9, 3, 2, 8, 1}
- After one iteration of the while loop of lines 4-6, the node and its parent have exchanged keys (values), and the index i moves up to the parent.



HEAP-INCREASE-KEY() Example

- A = {16, 15, 10, 14, 7, 9, 3, 2, 8, 1}
- After one more iteration of the while loop.
- The max-heap property now holds and the procedure terminates.





Example of HEAP-INCREASE-KEY

MAX-HEAP-INSERT

MAX-HEAP-INSERT(A, key)

1 $heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$

2 $A[heap\text{-}size[A]] \leftarrow -\infty$

3 HEAP-INCREASE-KEY(A, $heap\text{-}size[A]$, key)

O(lg n)

Running time is $O(\lg n)$

The path traced from the new leaf to the root has length $O(\lg n)$

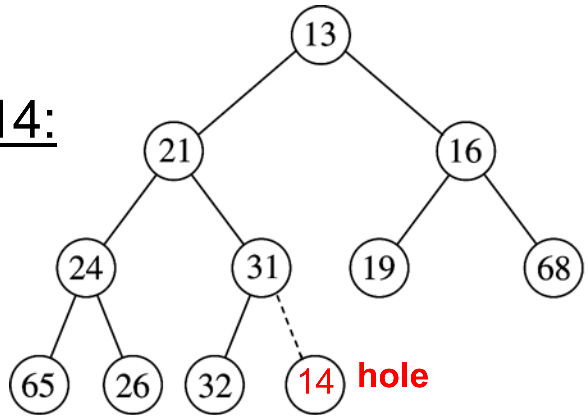
MIN-HEAP-INSERT

- Insert new element into the heap at the next available slot (“hole”)
  - According to maintaining a complete binary tree
- Then, “percolate” the element up the heap while heap-order property not satisfied

Percolating Up

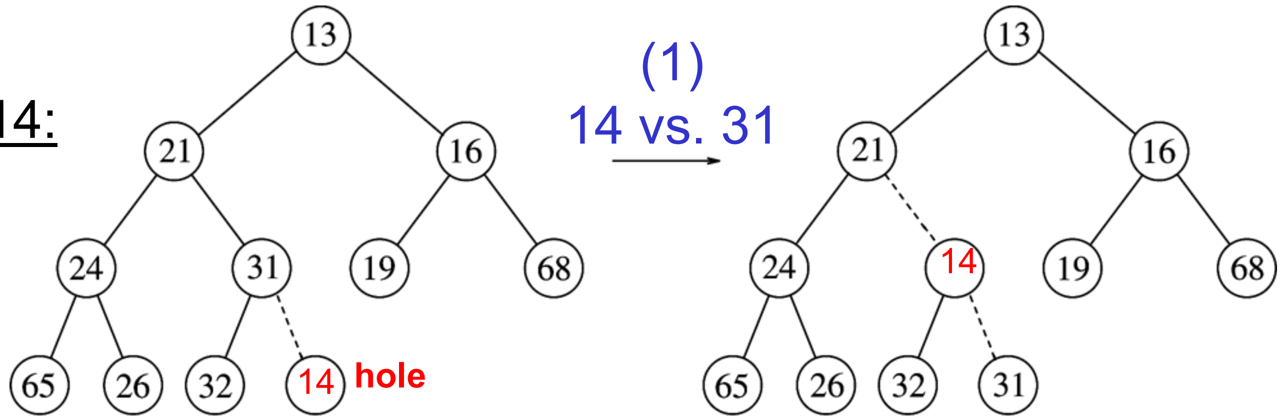
MIN-HEAP-INSERT : Example

Insert 14:



MIN-HEAP-INSERT : Example

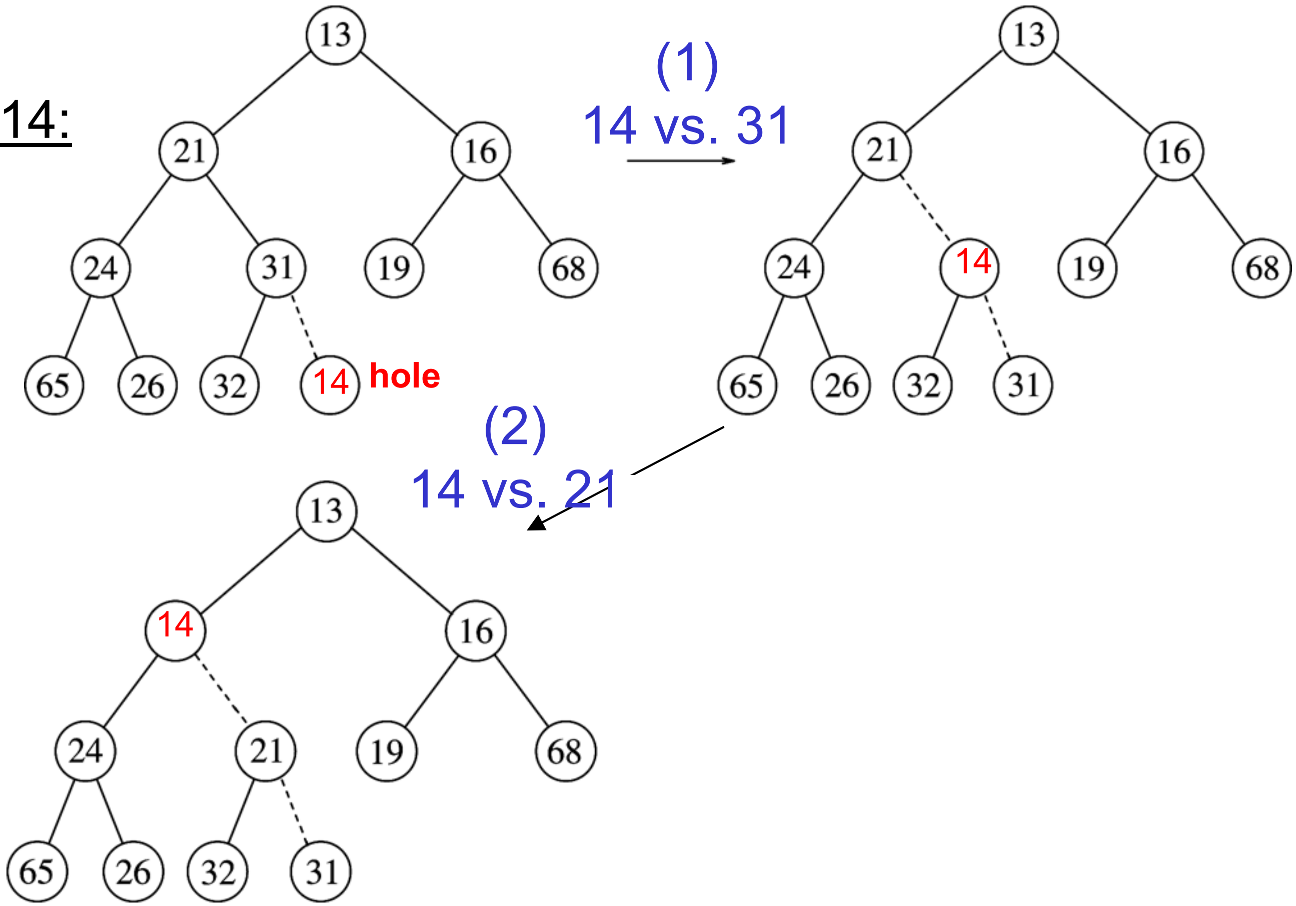
<u>Insert 14:</u>



Percolating Up

MIN-HEAP-INSERT : Example

<u>Insert 14:</u>



MIN-HEAP-INSERT : Example