Max Heap and Priority Queues Part 1

Ira A. Fulton Schools of Engineering
Arizona State University

Topics of this lecture

- The Heap Data Structure
- Heapify
- Build-Heap, Heapsort
- Max and ExtractMax
- IncreaseKey and Insertion
- Analysis of Heap Operations

The Max-Heap Data Structure: Definition

An array object with a tree view, and heap property

Tree view:

- The n elements are stored in A[1], A[2], …, A[n]
- Can be viewed as a nearly complete binary tree, the tree nodes in a complete binary tree are filled up, one level at a time, from left to right.

Heap property:

- A[parent(i)]≥A[i], ∀2≤i≤n

Tree view:

PARENT(i)
    return $\lfloor {}^{i}/_{2} \rfloor$

LEFT(i)
    return 2i

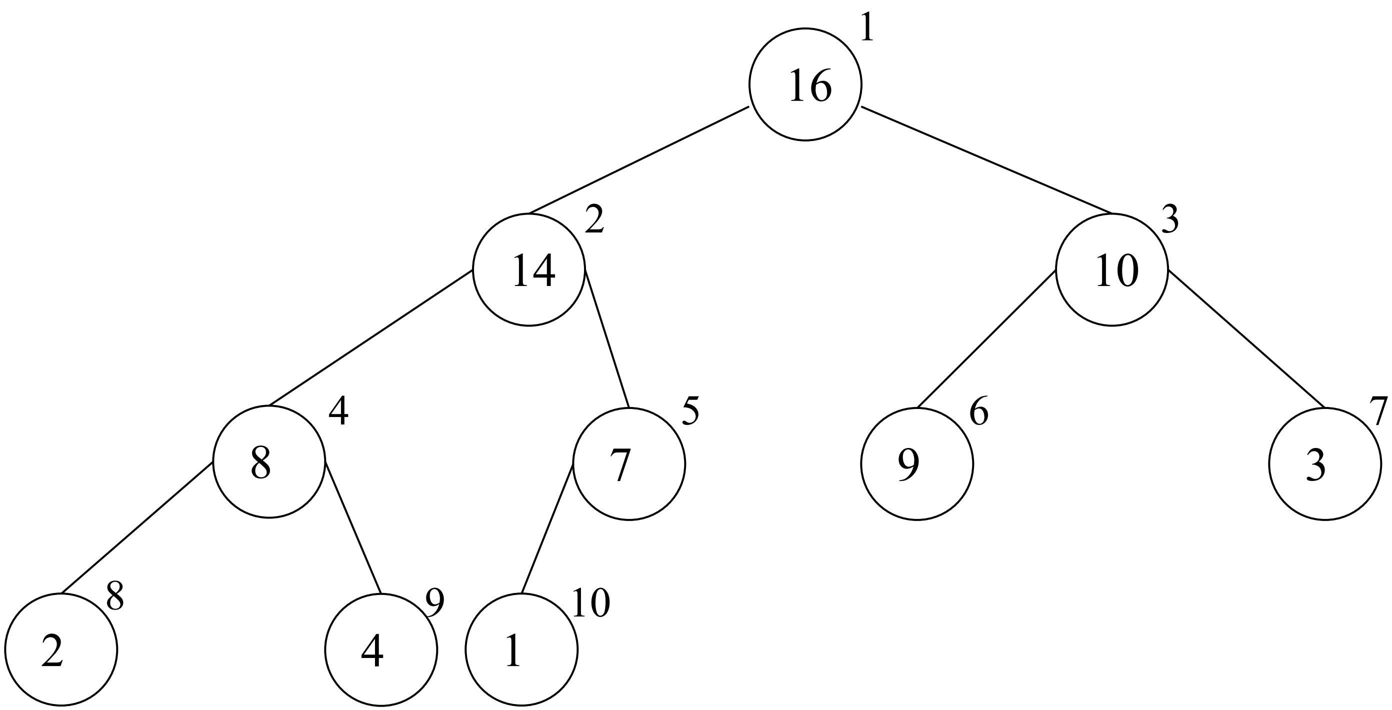
RIGHT(i)
    return (2i+1)

The Max-Heap Data Structure: Definition

- **Heap property (in the textbook):**
  - $A[\text{parent}(i)] \geq A[i], \forall 2 \leq i \leq n$
- **For ease of presentation, we will use the following equivalent heap property at node $i \in \{1, 2, \ldots, n\}$:**
  - $A[i] \geq A[\text{left}(i)]$, if $\text{left}(i) \leq n$
  - $A[i] \geq A[\text{right}(i)]$, if $\text{right}(i) \leq n$
- **We say heap property is violated at node i if one of the following happens:**
  - $\text{left}(i) \leq n$ but $A[i] < A[\text{left}(i)]$
  - $\text{right}(i) \leq n$ but $A[i] < A[\text{right}(i)]$

The Max-Heap Data Structure: Example

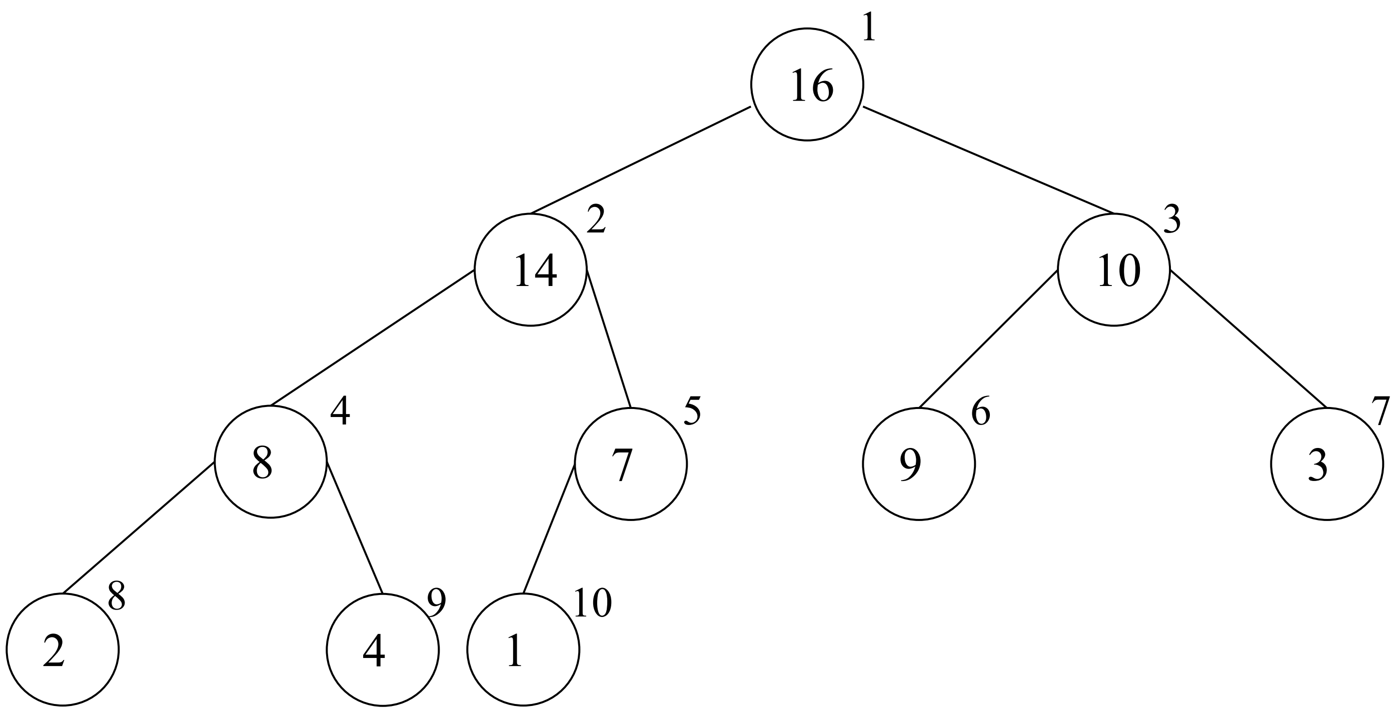
The following is a max-heap with 10 elements



16	14	10	8	7	9	3	2	4	1		
1	2	3	4	5	6	7	8	9	10	11	12

The Max-Heap Data Structure: Example

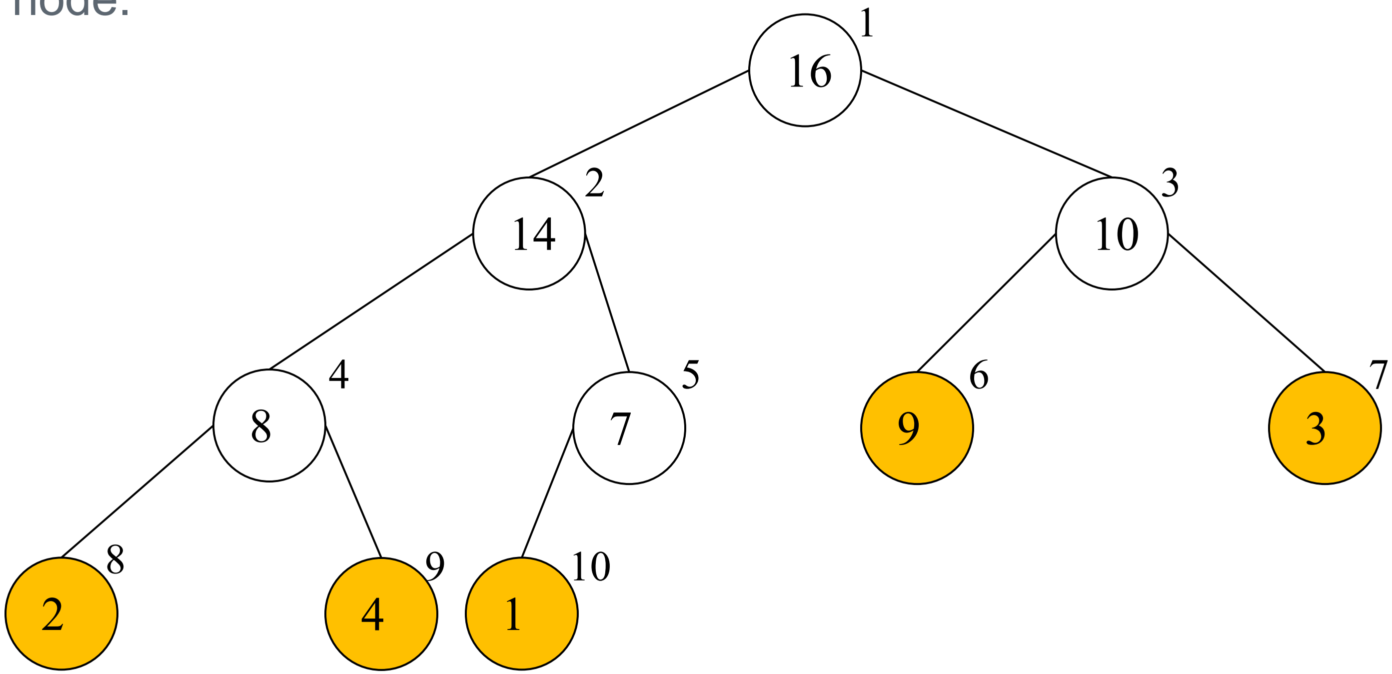
It has a tree view: A nearly complete binary tree, from A[1] to A[10], where 10 is the heap-size



16	14	10	8	7	9	3	2	4	1		
1	2	3	4	5	6	7	8	9	10	11	12

The Max-Heap Data Structure: Example

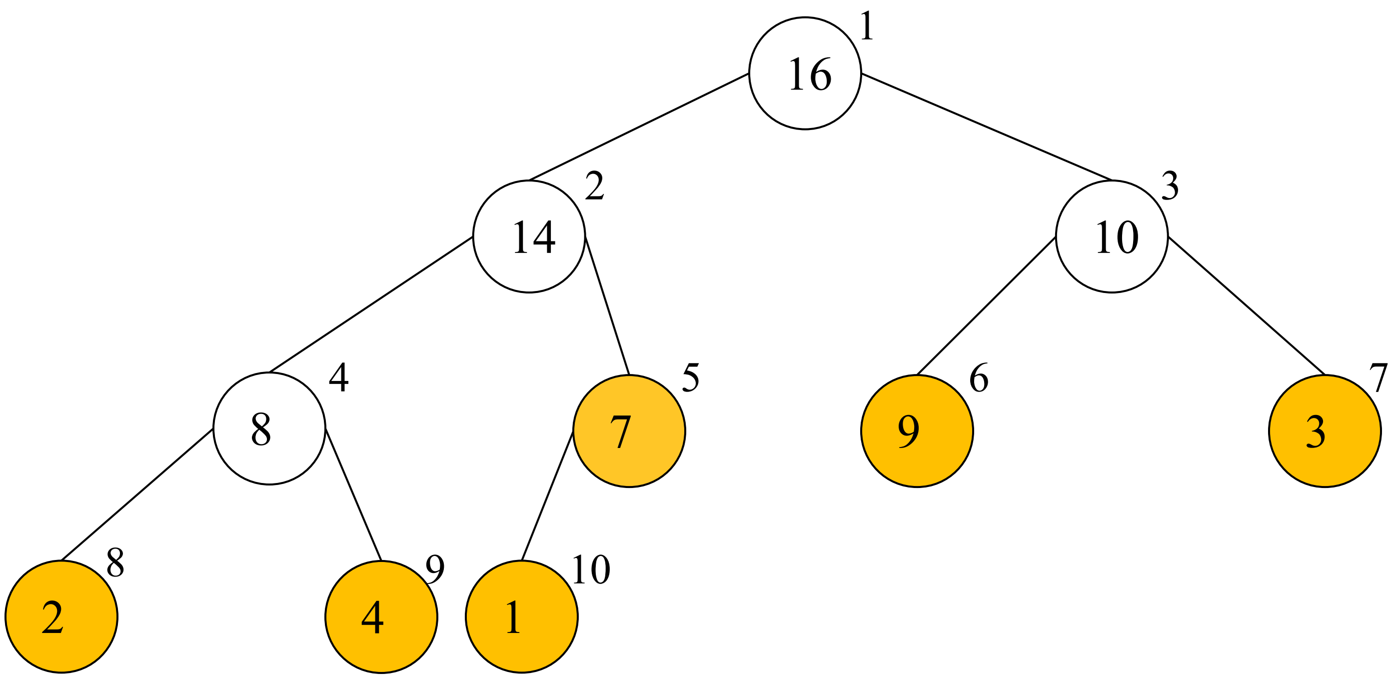
Clearly, heap property is not violated at nodes 10, 9, 8, 7, 6, as none of them has a child node.



16	14	10	8	7	9	3	2	4	1		
1	2	3	4	5	6	7	8	9	10	11	12

The Max-Heap Data Structure: Example

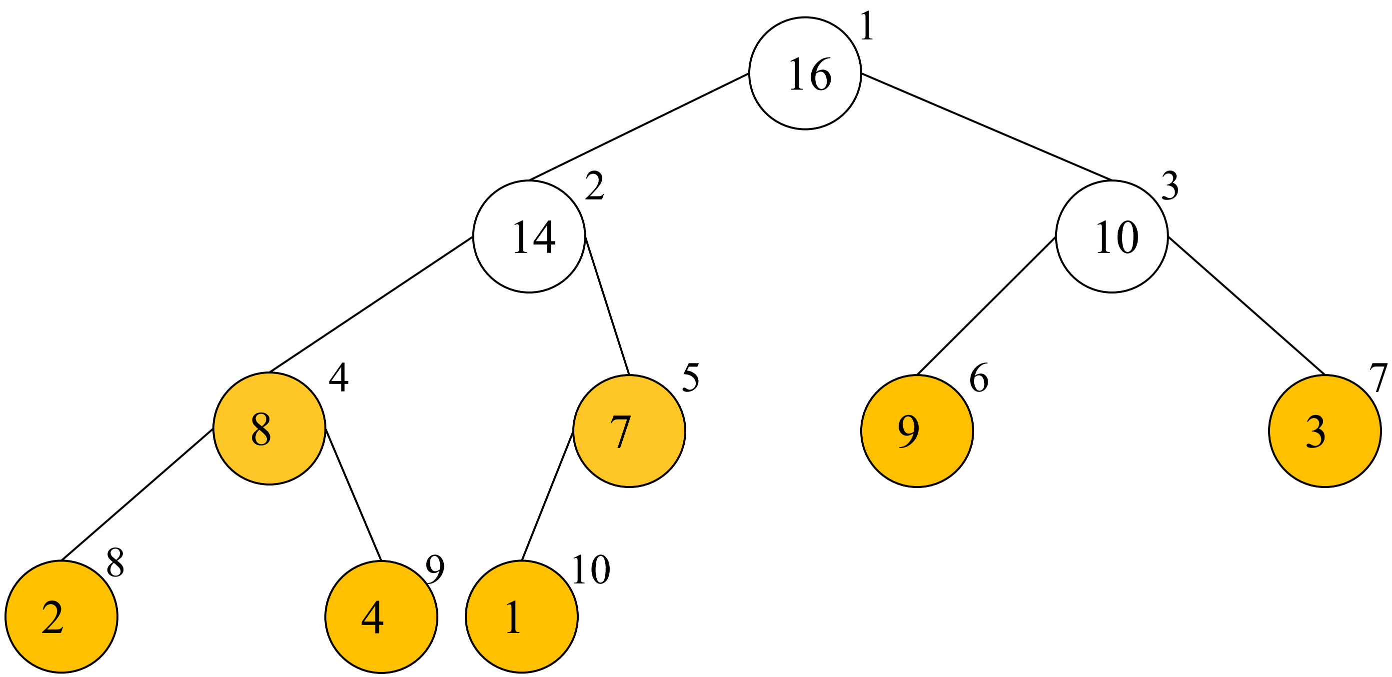
We verify that heap property is not violated at node 5.



16	14	10	8	7	9	3	2	4	1		
1	2	3	4	5	6	7	8	9	10	11	12

The Max-Heap Data Structure: Example

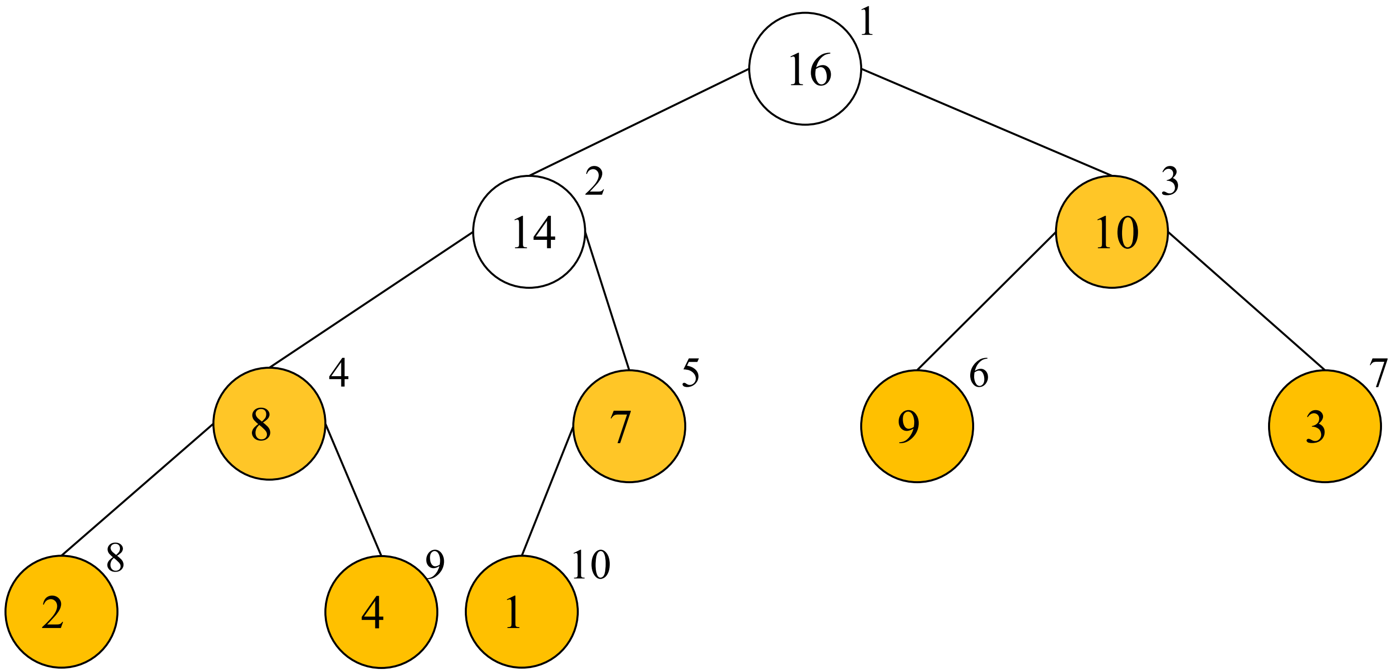
We verify that heap property is not violated at node 4.



16	14	10	8	7	9	3	2	4	1		
1	2	3	4	5	6	7	8	9	10	11	12

The Max-Heap Data Structure: Example

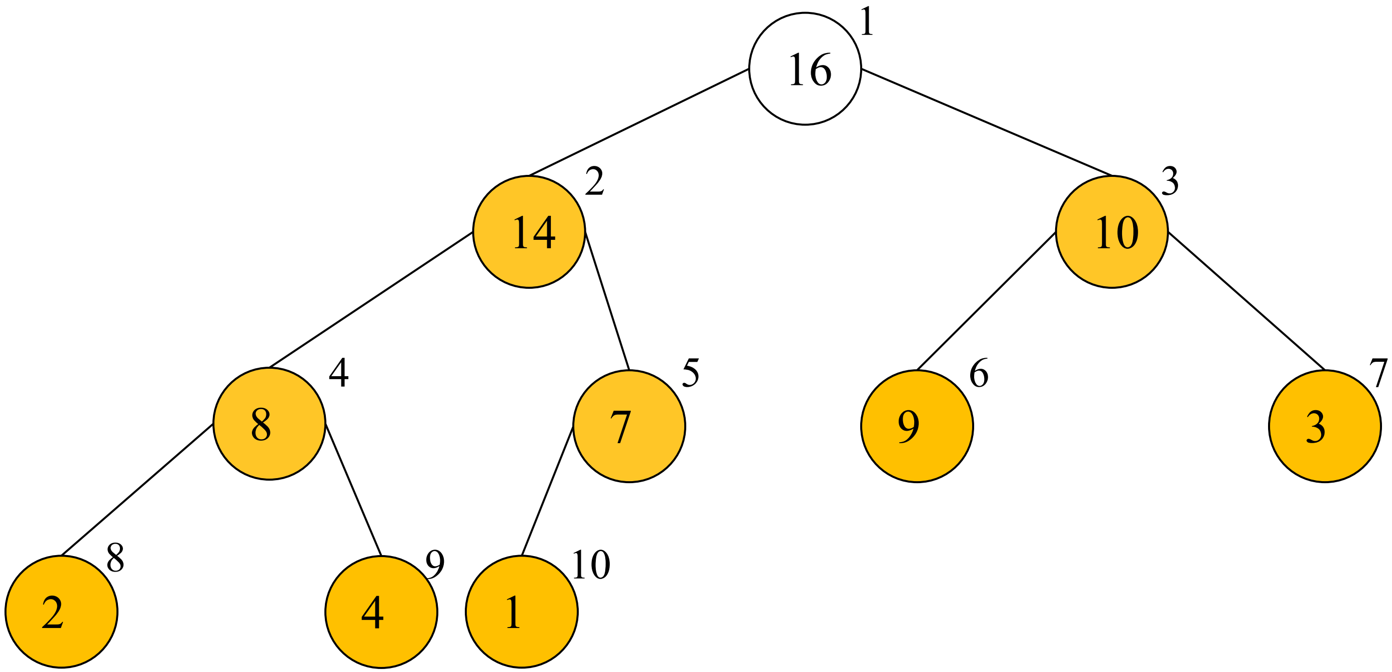
We verify that heap property is not violated at node 3.



16	14	10	8	7	9	3	2	4	1		
1	2	3	4	5	6	7	8	9	10	11	12

The Max-Heap Data Structure: Example

We verify that heap property is not violated at node 2.

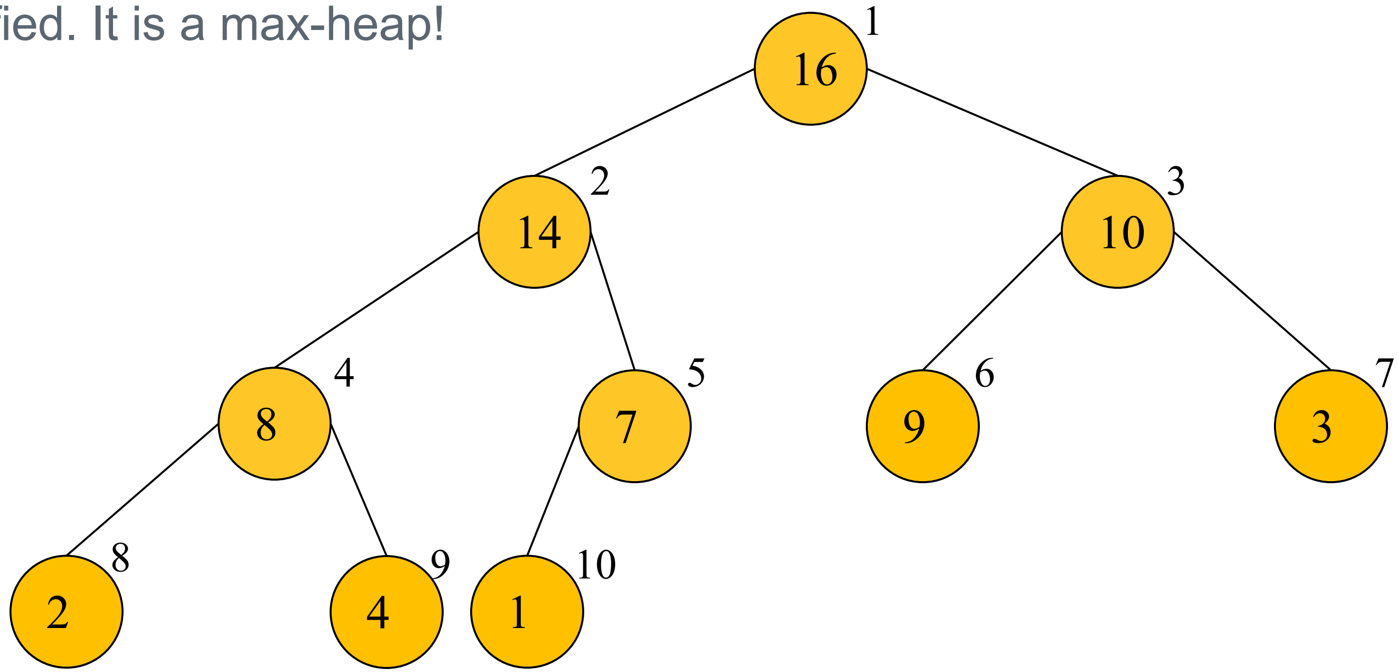


16	14	10	8	7	9	3	2	4	1		
1	2	3	4	5	6	7	8	9	10	11	12

The Max-Heap Data Structure: Example

We verify that heap property is not violated at node 1.

Verified. It is a max-heap!

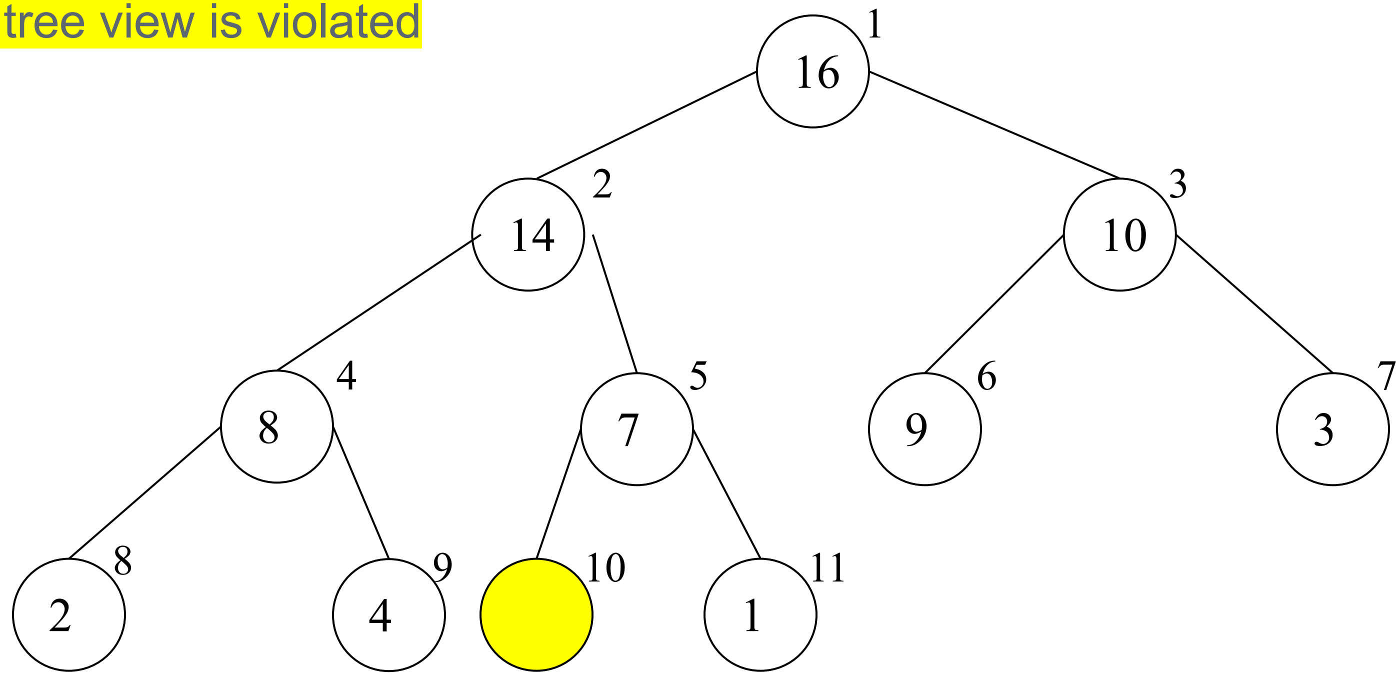


16	14	10	8	7	9	3	2	4	1		
1	2	3	4	5	6	7	8	9	10	11	12

The Max-Heap Data Structure

This is NOT a max-heap.

The tree view is violated

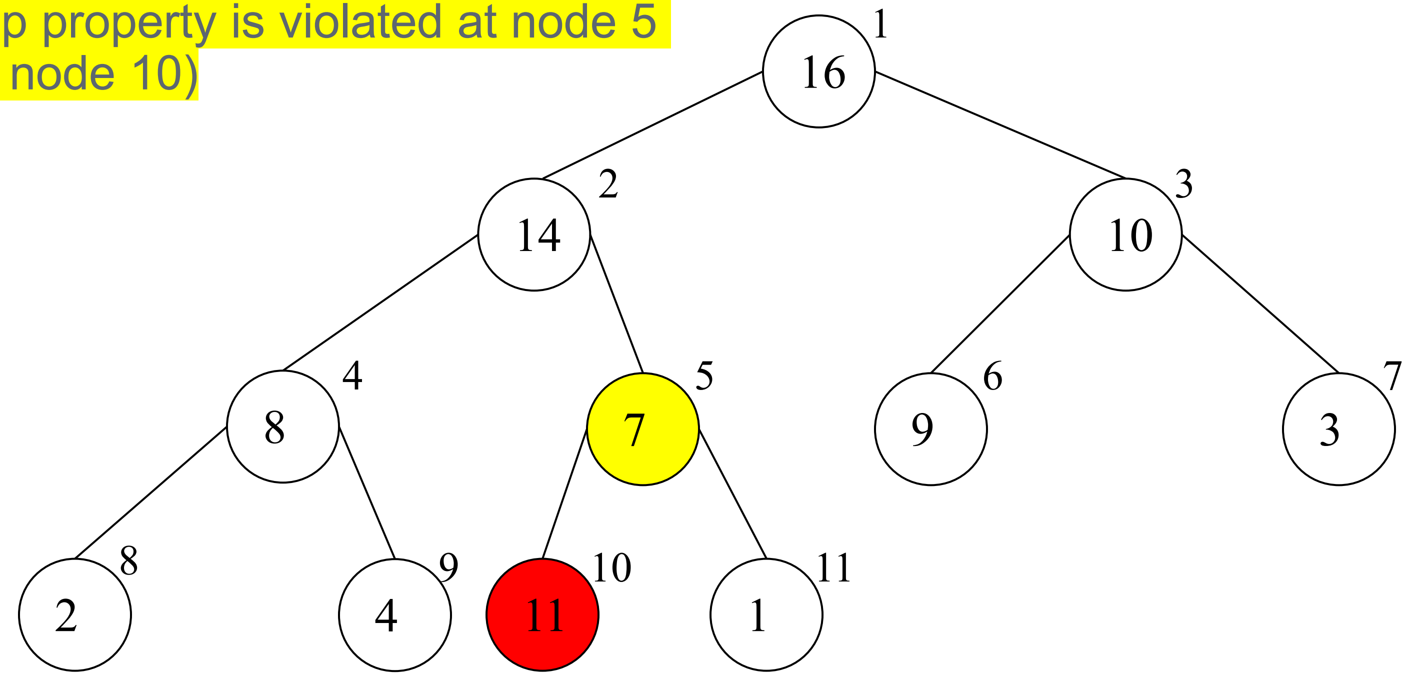


16	14	10	8	7	9	3	2	4		1	
1	2	3	4	5	6	7	8	9	10	11	12

The Max-Heap Data Structure

This is NOT a max-heap.

Heap property is violated at node 5 (not node 10)



16	14	10	8	7	9	3	2	4	11		
1	2	3	4	5	6	7	8	9	10	11	12

Summary

- **The max-heap data structure is an array object that has**
  - A nearly completely binary tree view
  - Heap property
- **It should be implemented as an array object, not a binary tree**

Ira A. Fulton Schools of Engineering

Arizona State University

Max Heap and Priority Queues Part 2

Ira A. Fulton Schools of Engineering
Arizona State University

Topics of this lecture

- The Heap Data Structure
- Heapify
- Build-Heap, Heapsort
- Max and ExtractMax
- IncreaseKey and Insertion
- Analysis of Heap Operations

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] and A[largest]
            MAX-HEAPIFY(A, largest)
```

O(1)

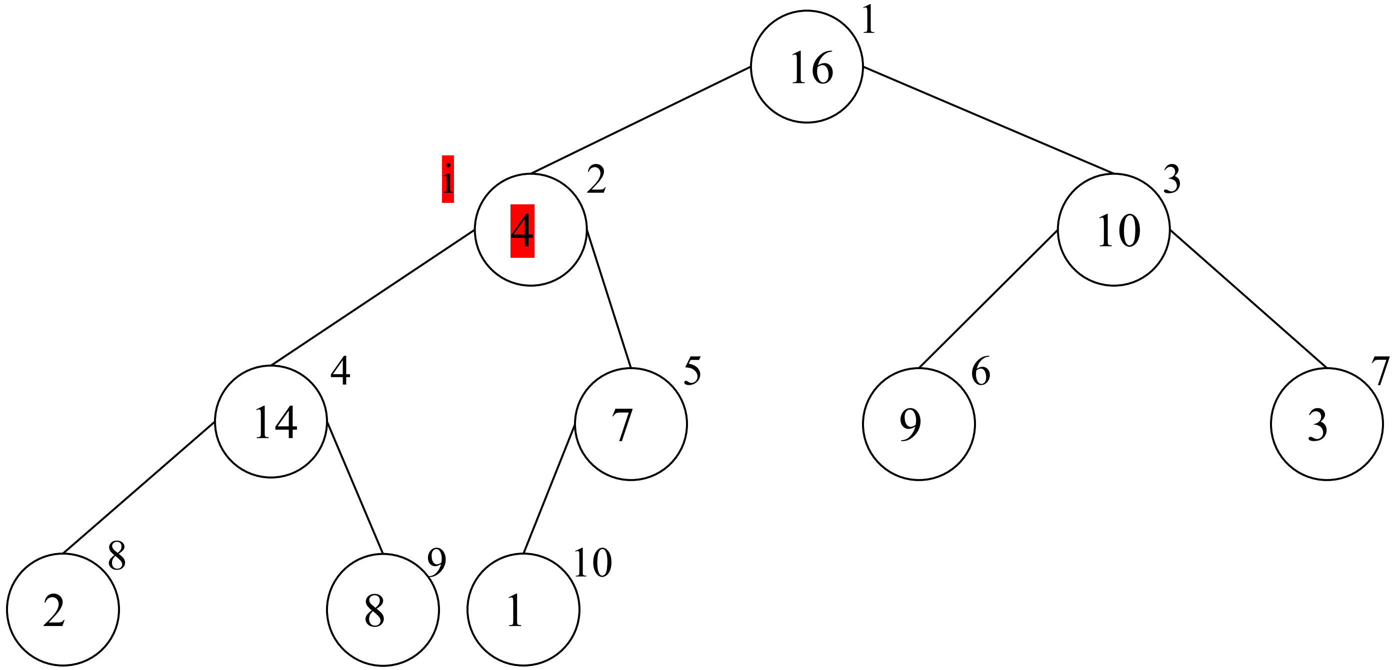
$T(^{2n}/_3)$

- T(n) = worst-case running time of HEAPIFY(A, i) on a heap with n elements is proportional to the height of the tree: O(log n).

- Assumption: LEFT(i) and RIGHT(i) are max-heaps before the call.

Example: Max-Heapify(A, 2)

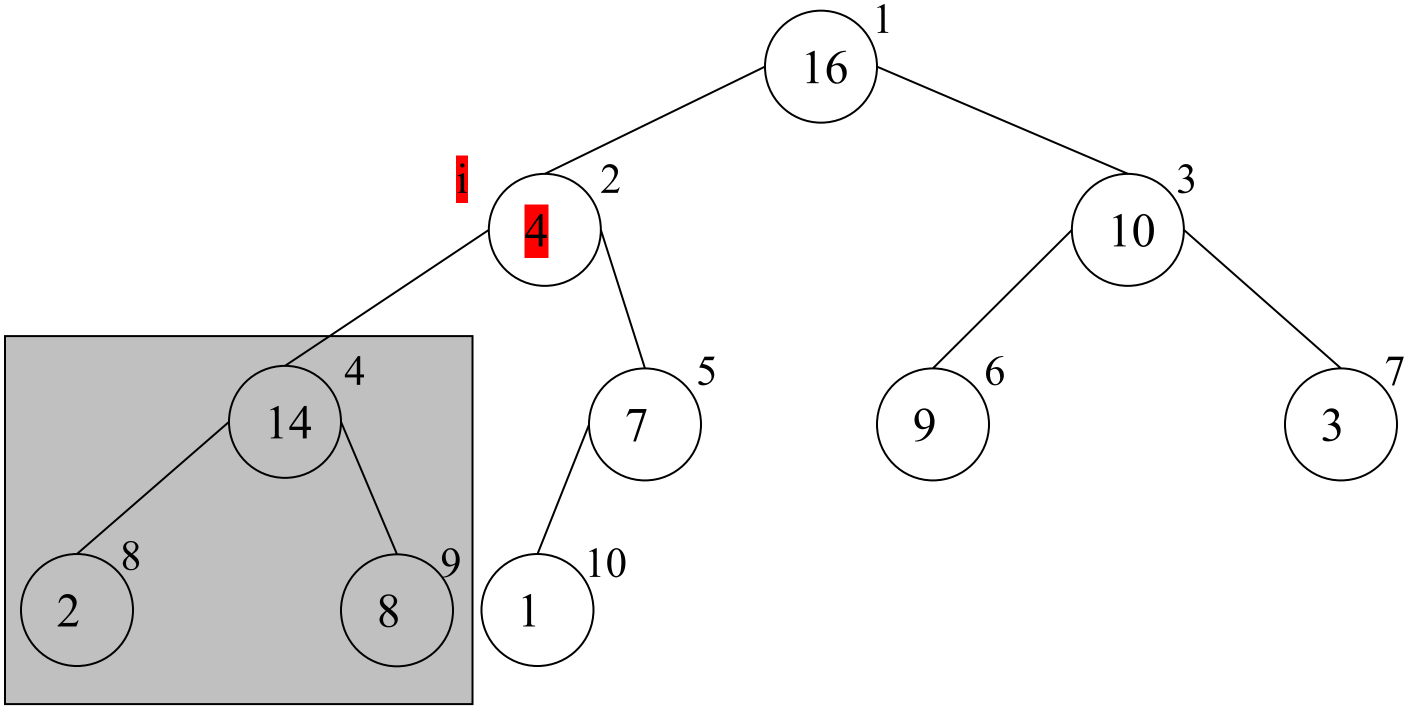
We illustrate the steps of max-heapify(A, 2)



16	4	10	14	7	9	3	2	8	1		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Max-Heapify(A, 2)

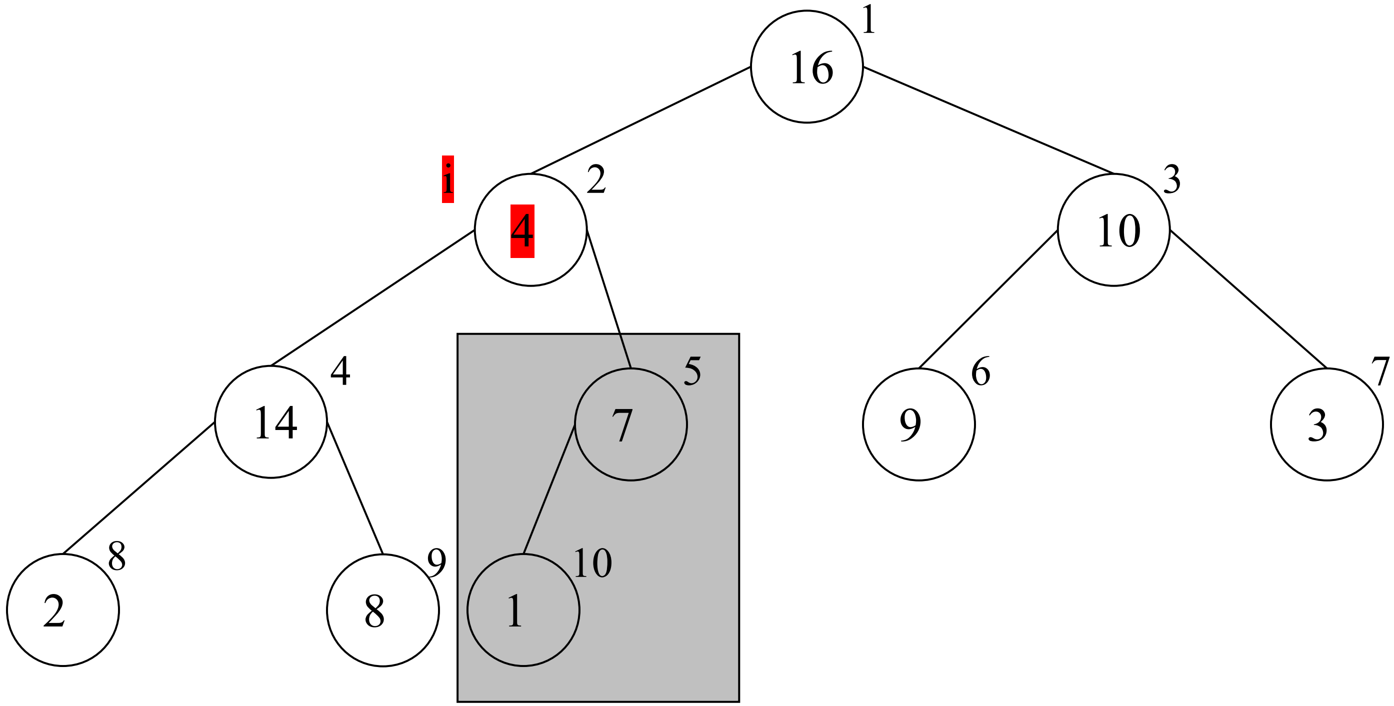
The tree rooted at left(i) is a max-heap



16	4	10	14	7	9	3	2	8	1		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Max-Heapify(A, 2)

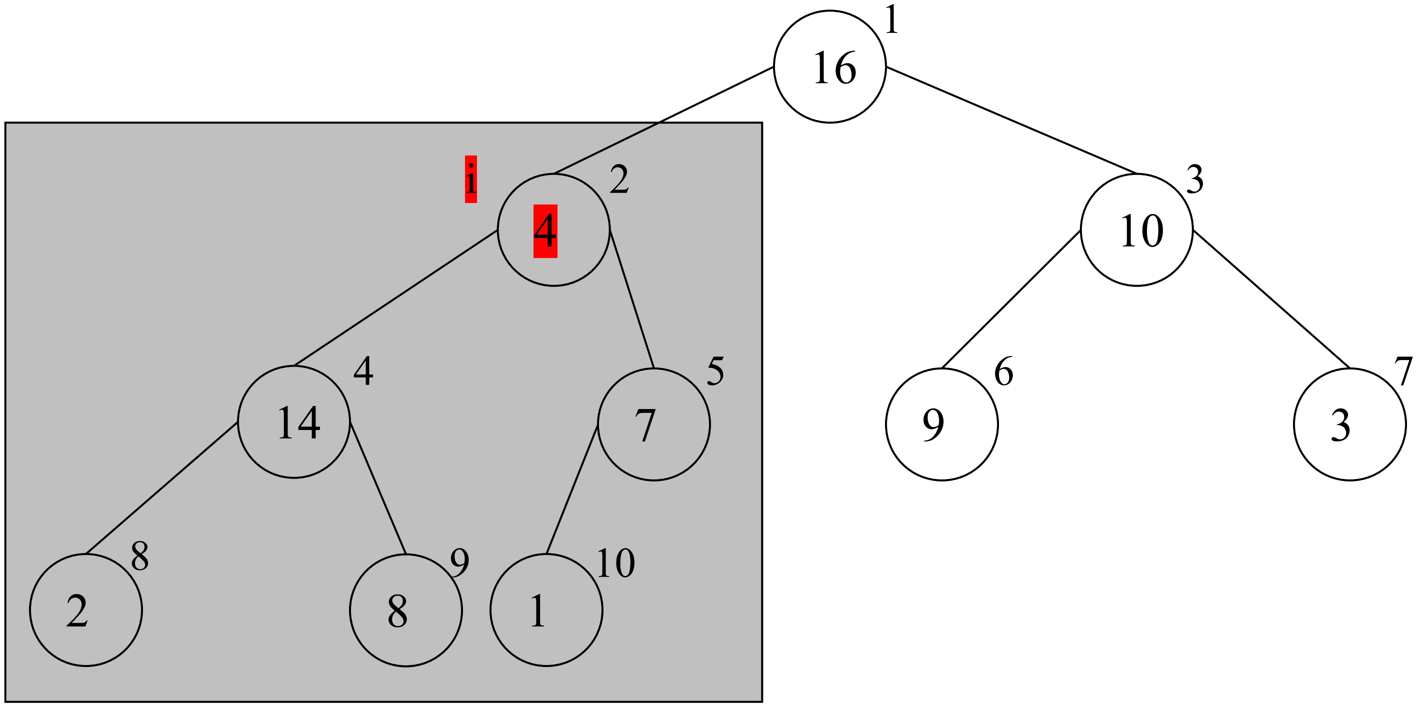
The tree rooted at right(i) is a max-heap



16	4	10	14	7	9	3	2	8	1		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Max-Heapify(A, 2)

We do not need to know whether the tree rooted at node i is a max-heap or not. In this particular case, it is not.

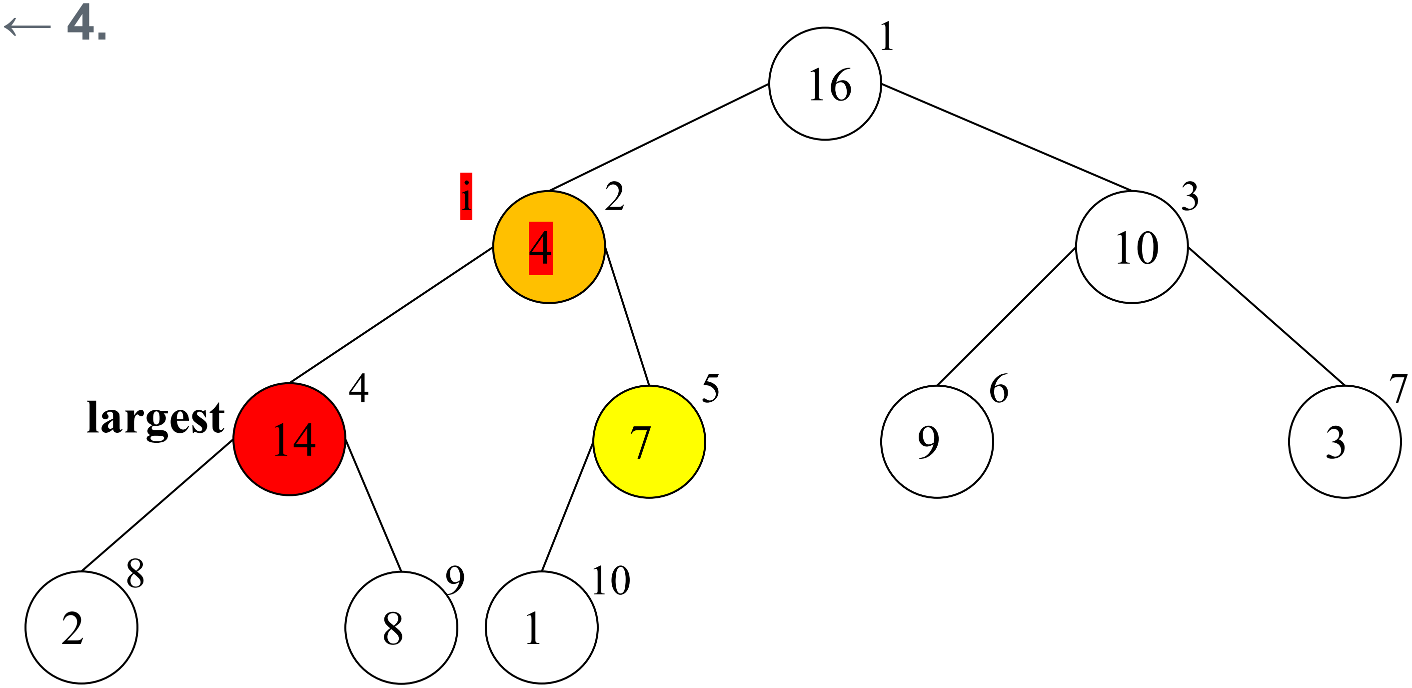


16	4	10	14	7	9	3	2	8	1		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Max-Heapify(A, 2)

Max-heapify(A, 2), i ← 2

largest ← 4.

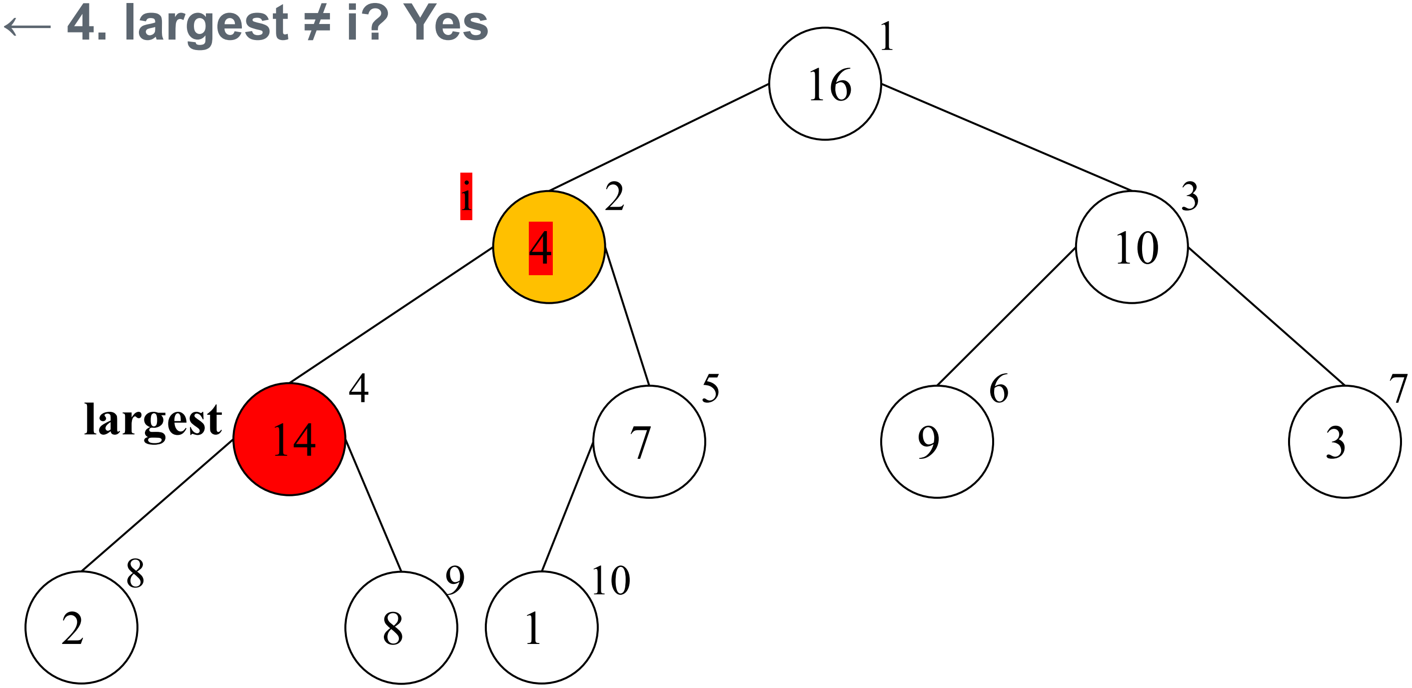


16	4	10	14	7	9	3	2	8	1		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Max-Heapify(A, 2)

Max-heapify(A, 2), ii ← 4

largest ← 4. largest ≠ i? Yes



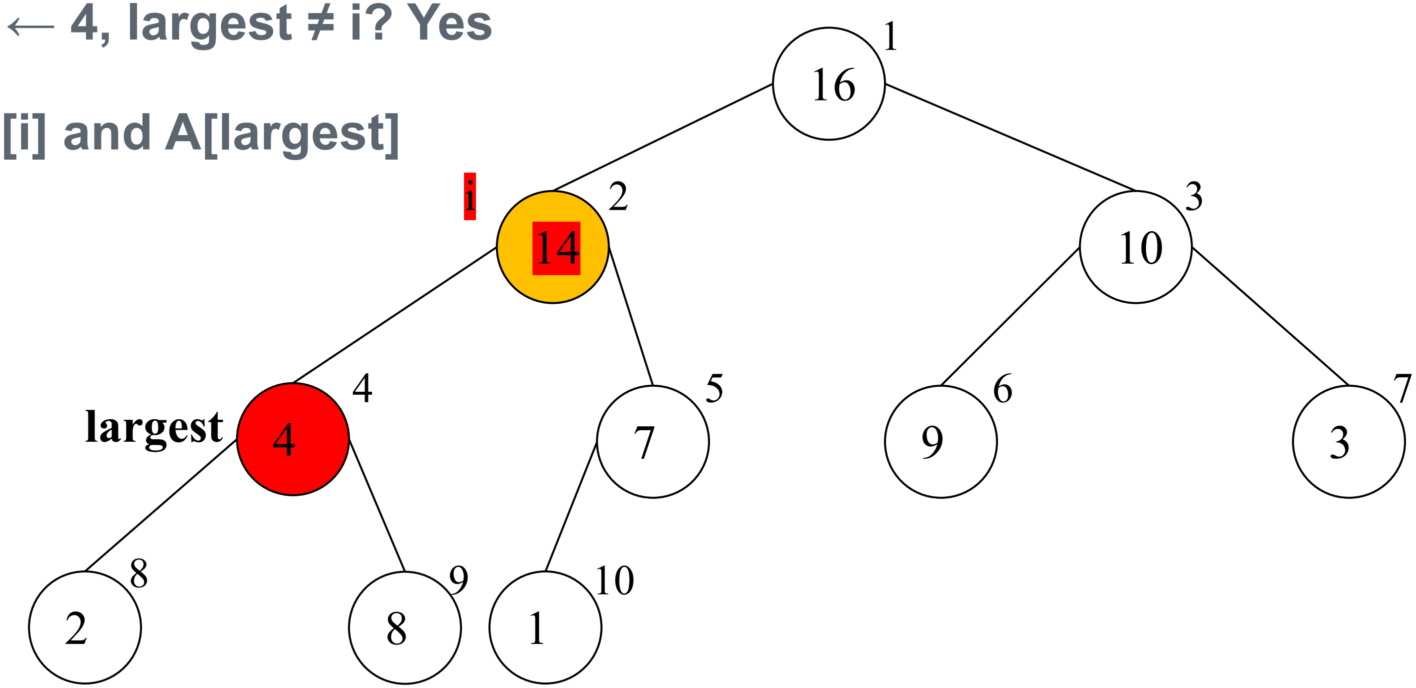
16	14	10	4	7	9	3	2	8	1		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Max-Heapify(A, 2)

Max-heapify(A, 2)

largest ← 4, largest ≠ i? Yes

swap A[i] and A[largest]

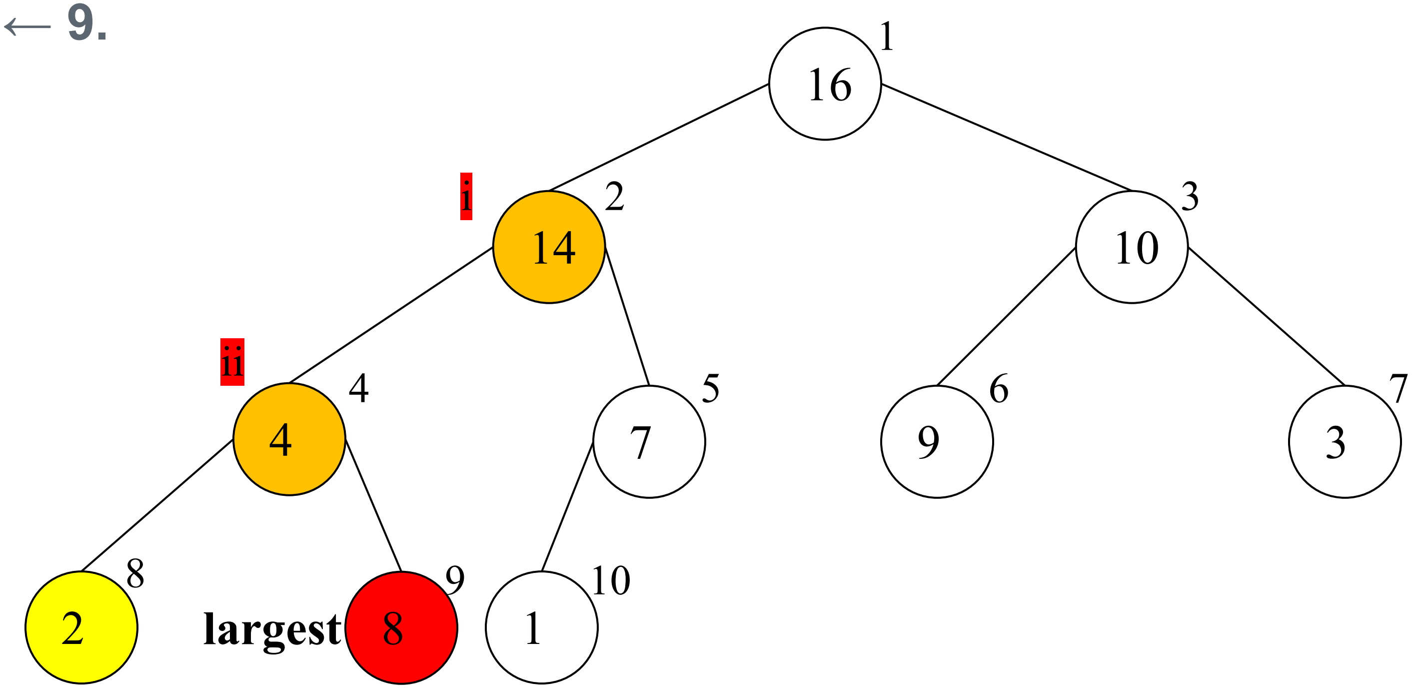


16	14	10	4	7	9	3	2	8	1		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Max-Heapify(A, 2)

Max-heapify(A, 4), ii ← 4

largest ← 9.

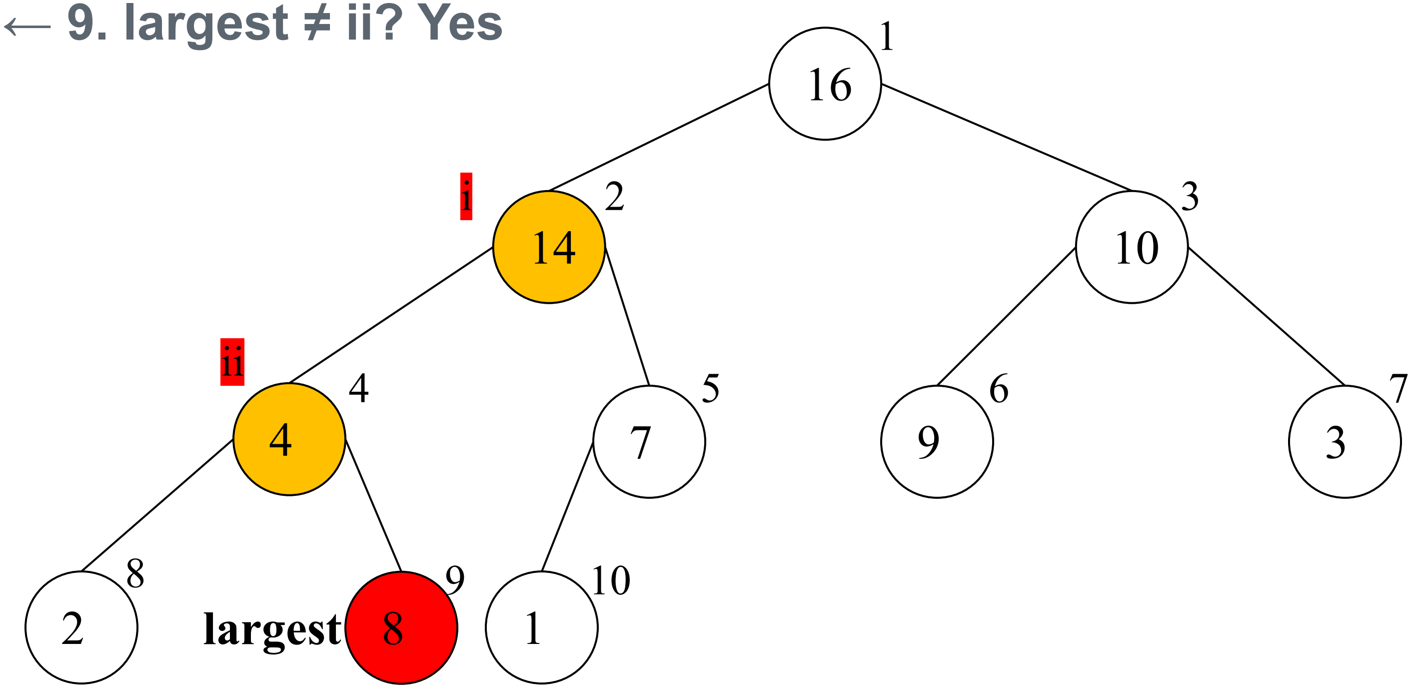


16	14	10	4	7	9	3	2	8	1		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Max-Heapify(A, 2)

Max-heapify(A, 4), ii ← 4

largest ← 9. largest ≠ ii? Yes



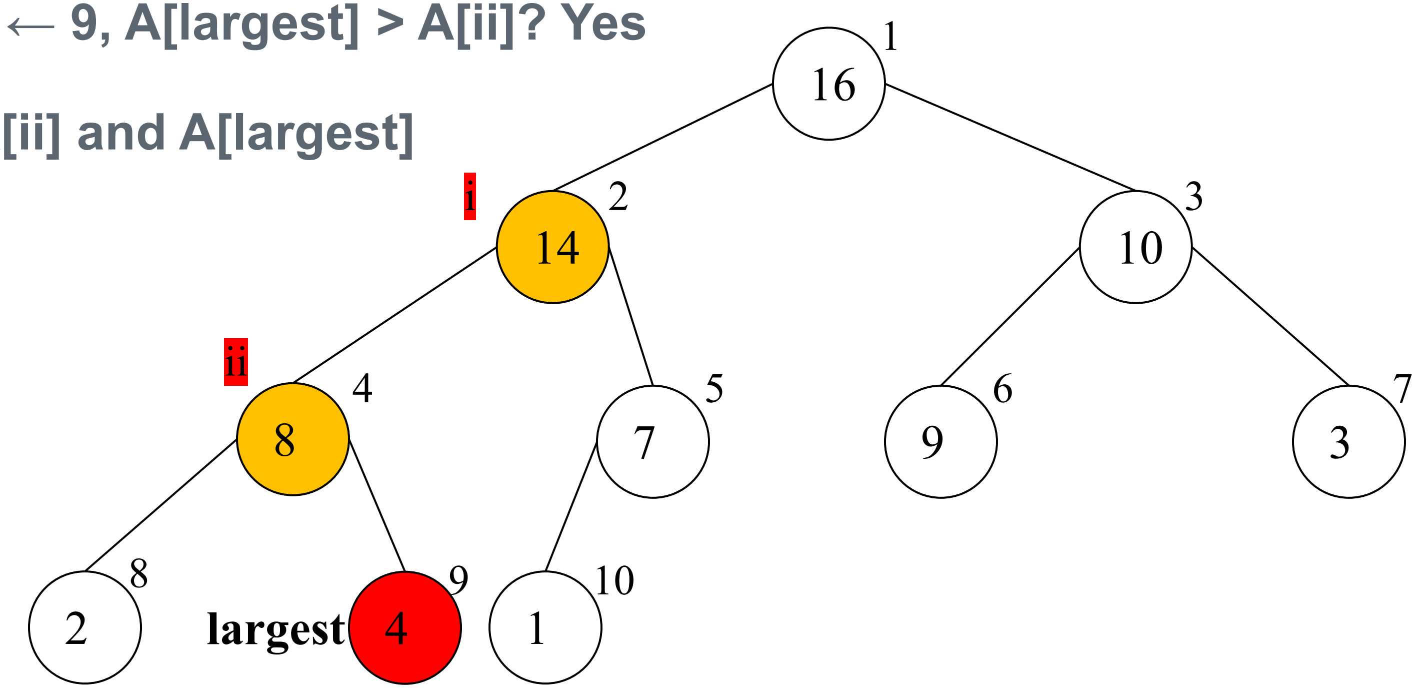
16	14	10	4	7	9	3	2	8	1		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Max-Heapify(A, 2)

Max-heapify(A, 4), ii ← 4

largest ← 9, A[largest] > A[ii]? Yes

swap A[ii] and A[largest]

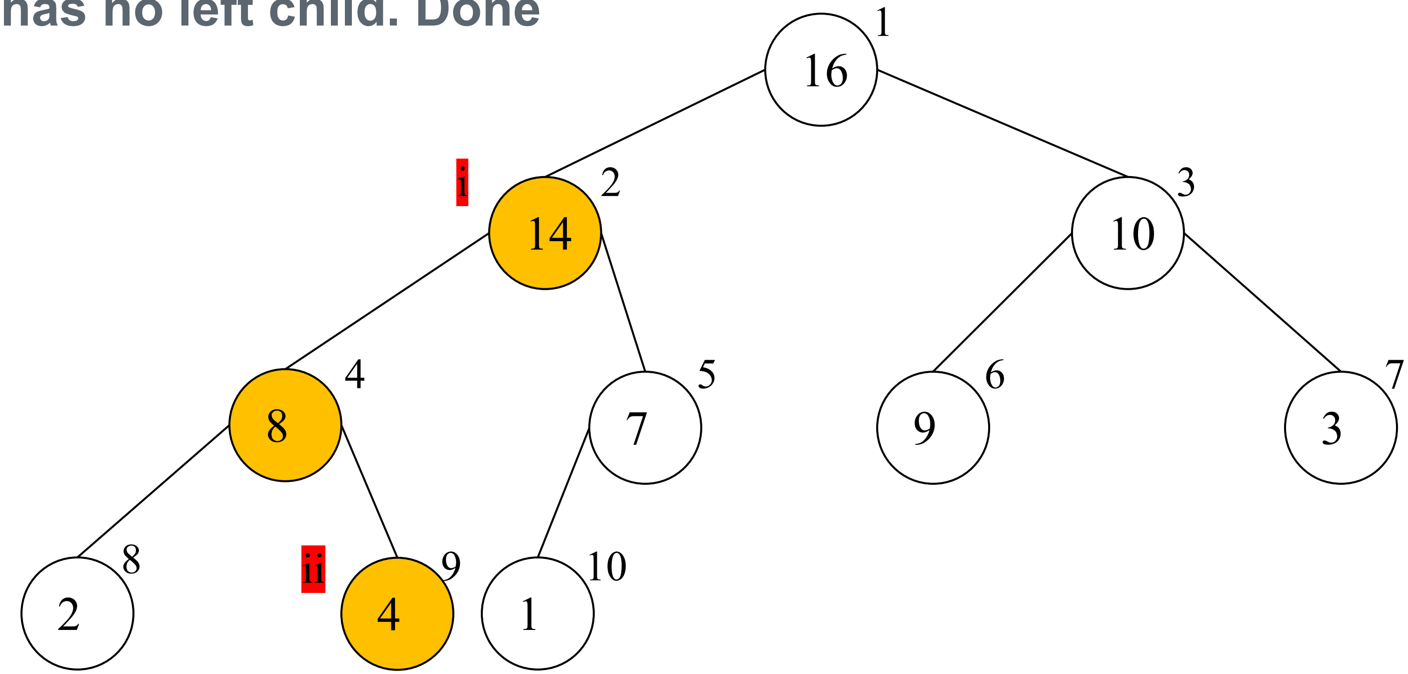


16	14	10	8	7	9	3	2	4	1		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Max-Heapify(A, 2)

Max-heapify(A, 9), ii $\leftarrow$ 9

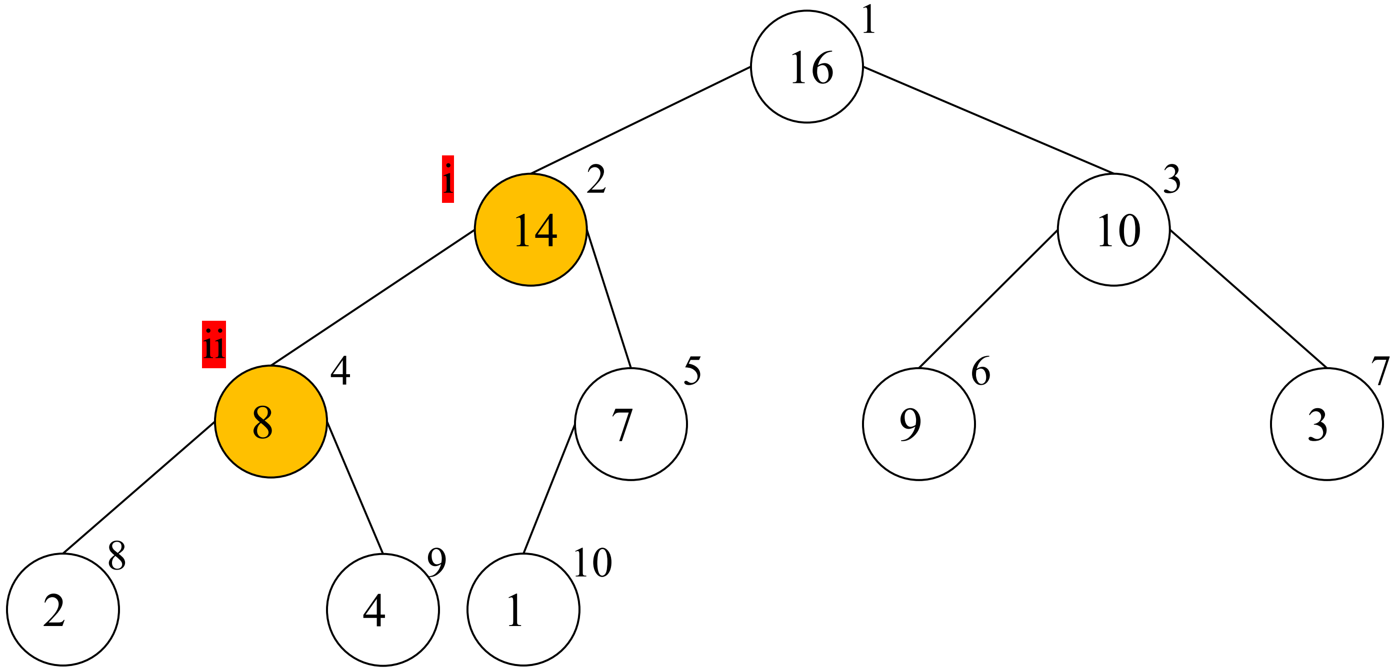
Node ii has no left child. Done



16	14	10	8	7	9	3	2	4	1		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Max-Heapify(A, 2)

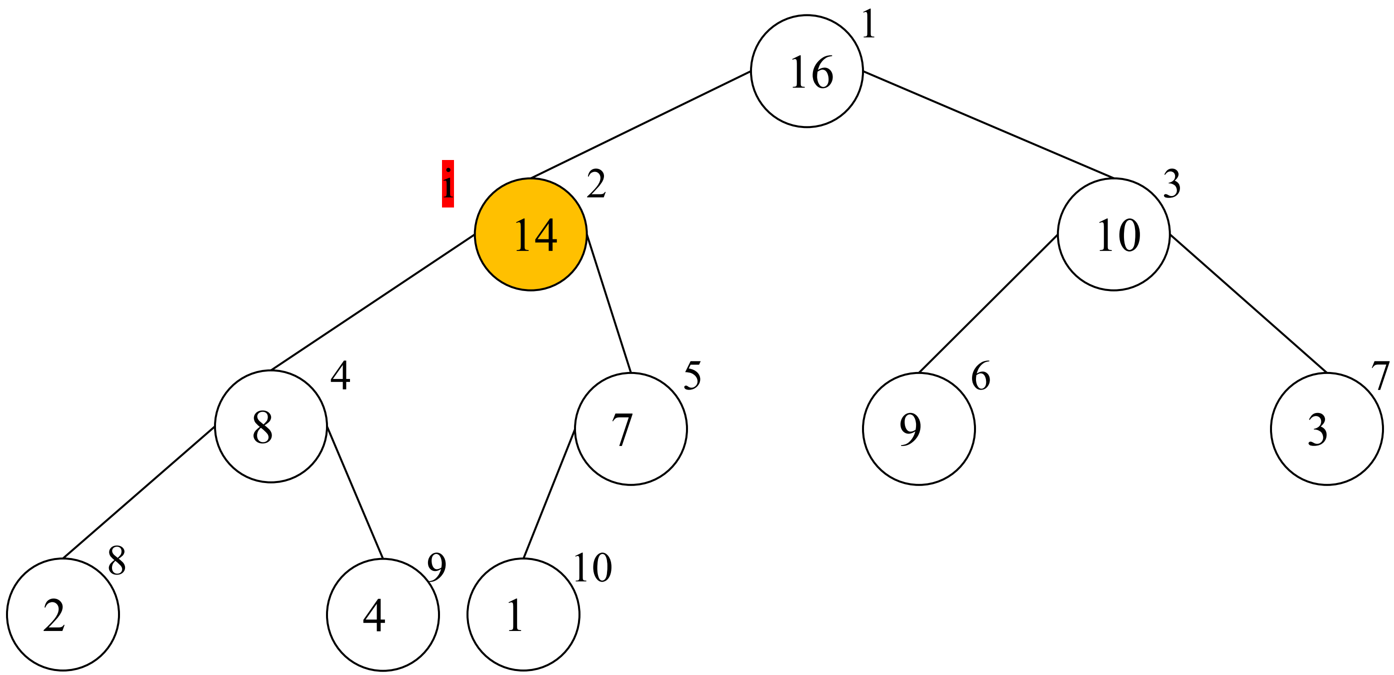
Max-heapify(A, 4), Done



16	14	10	8	7	9	3	2	4	1		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Max-Heapify(A, 2)

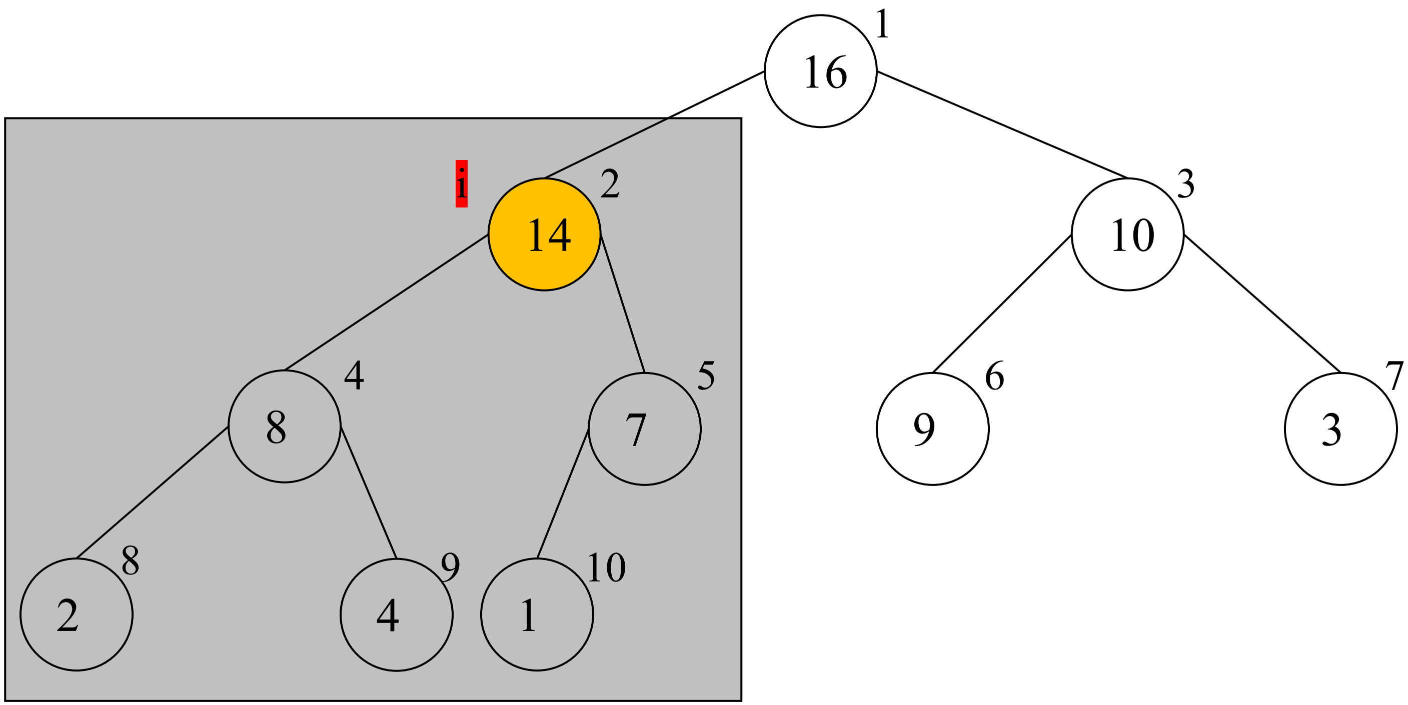
Max-heapify(A, 2), Done



16	14	10	8	7	9	3	2	4	1		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Max-Heapify(A, 2)

The tree rooted at node 2 is a max-heap.



16	14	10	8	7	9	3	2	4	1		
1	2	3	4	5	6	7	8	9	10	11	12

Summary

- **Pre-condition of Max-Heapify(A, i)**
  - If left(i)<=heap-size, the subtree rooted at left(i) is a max-heap
  - If right(i)<=heap-size, the subtree rooted at right(i) is a max-heap
- **Post-condition of Max-Heapify(A, i)**
  - The subtree rooted at i is a max-heap
- **We will prove that the time complexity of Heapify is $O(n)$, where n is the heap-size.**

Ira A. Fulton Schools of
Engineering
Arizona State University

Max Heap and Priority Queues Part 3

Ira A. Fulton Schools of Engineering
Arizona State University

Topics of this lecture

- The Heap Data Structure
- Heapify
- Build-Heap, Heapsort
- Max and ExtractMax
- IncreaseKey and Insertion
- Analysis of Heap Operations

Build Heap

= **Building a Heap** =

How can we build a heap from $\underbrace{\text{an arbitrary array}}_{\text{i.e. it may not necessarily be a heap.}}$ A[1..n] ?

- **bottom-up manner, using HEAPIFY.**

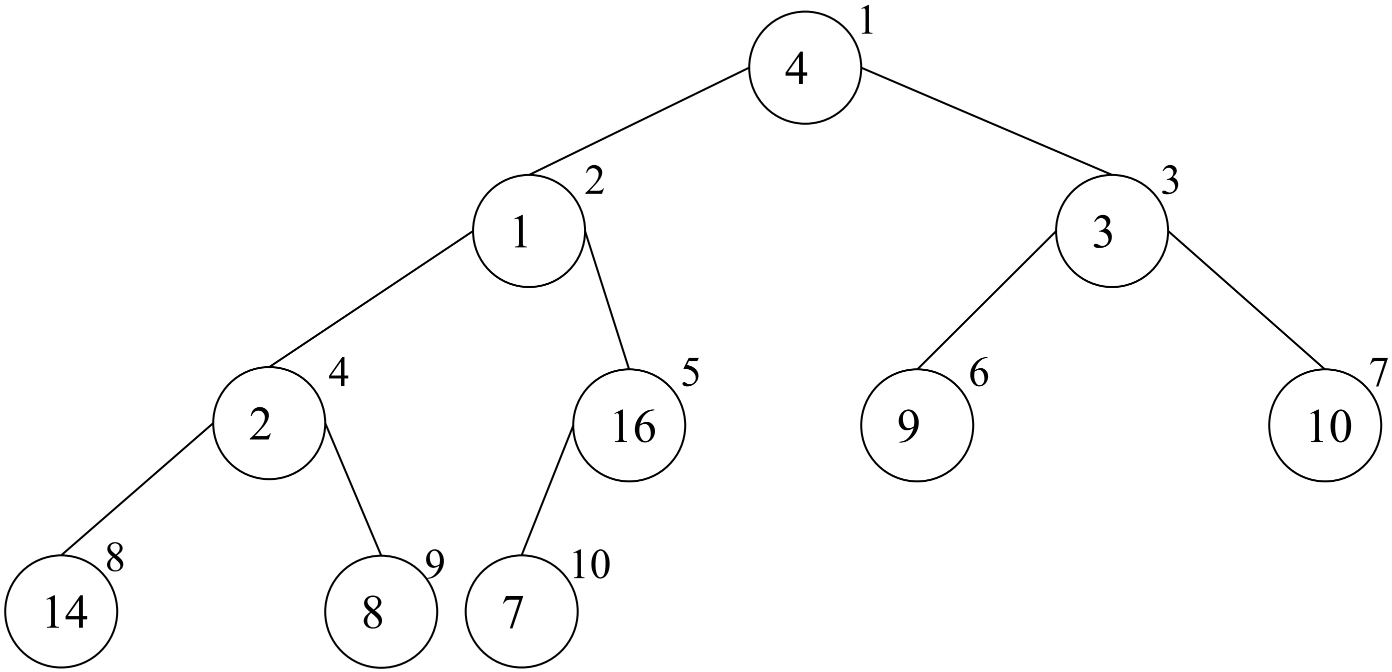
- **The elements in the array $[\lfloor {}^{n}/_{2} \rfloor + 1 \,..\, n]$ are all leaves of the tree, each is a 1-element heap to begin with. The procedure BUILD-HEAP goes through the remaining nodes of the tree and runs HEAPIFY on each node.**

Build Heap

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

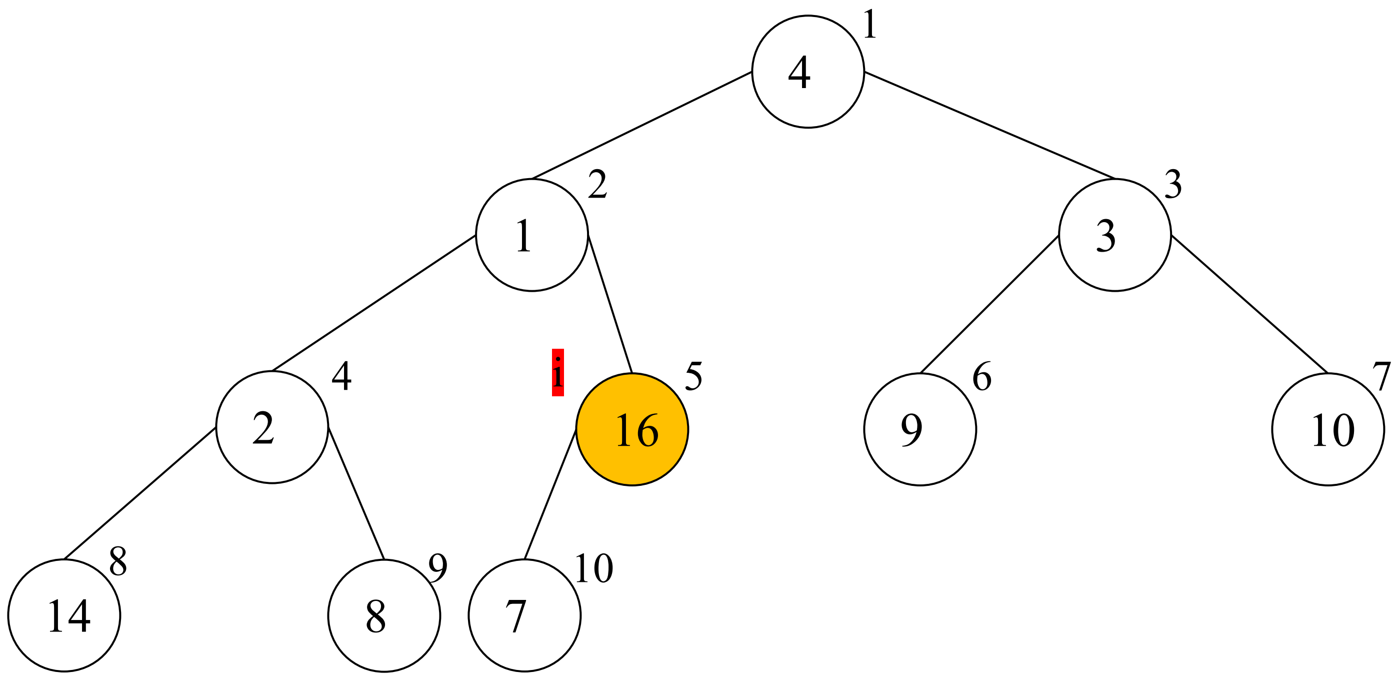
- **Does the order in which the nodes are processed matter?**
  - Yes, since we need to guarantee that the subtrees rooted at the children of a node i are heaps *before* HEAPIFY is called at that node.
- **BUILD-HEAP runs in O(*n* log *n*) time. Why?**
  - Is this a tight bound? No. The tight upper bound is Θ(n).

Example: Build-Max-Heap(A)



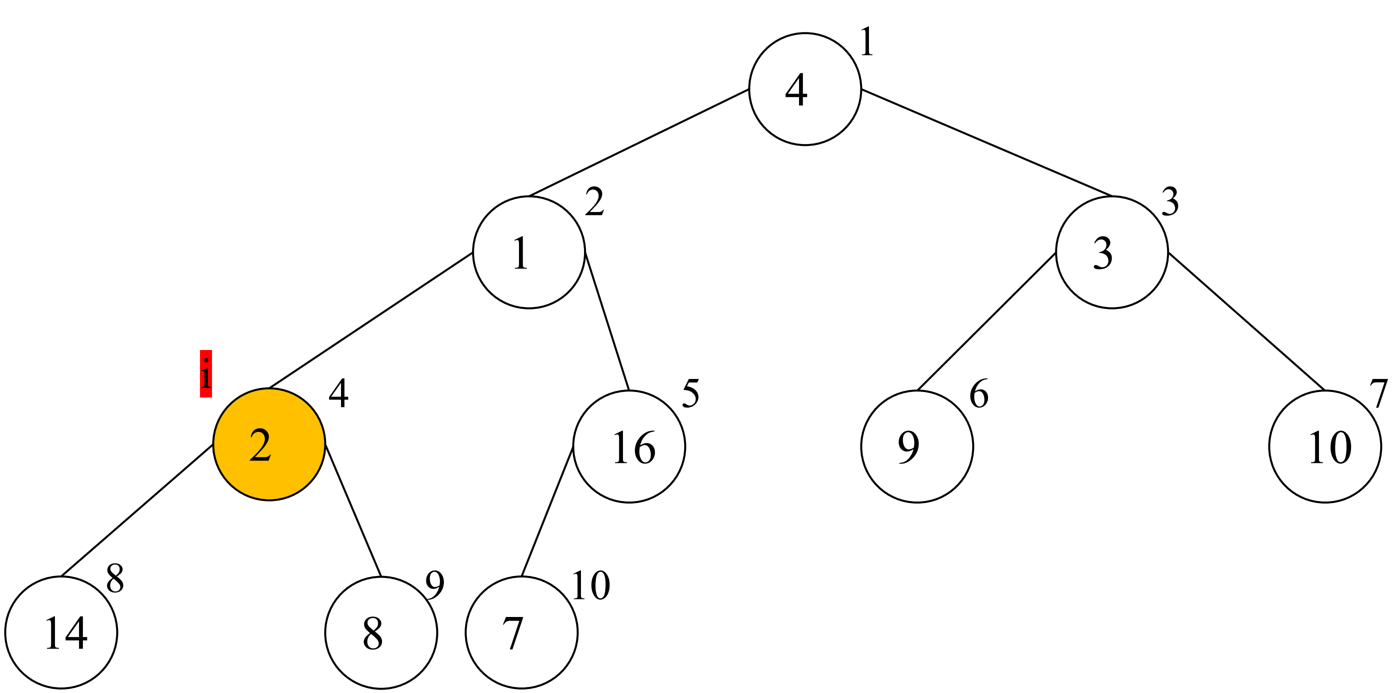
4	1	3	2	16	9	10	14	8	7		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Build-Max-Heap(A)



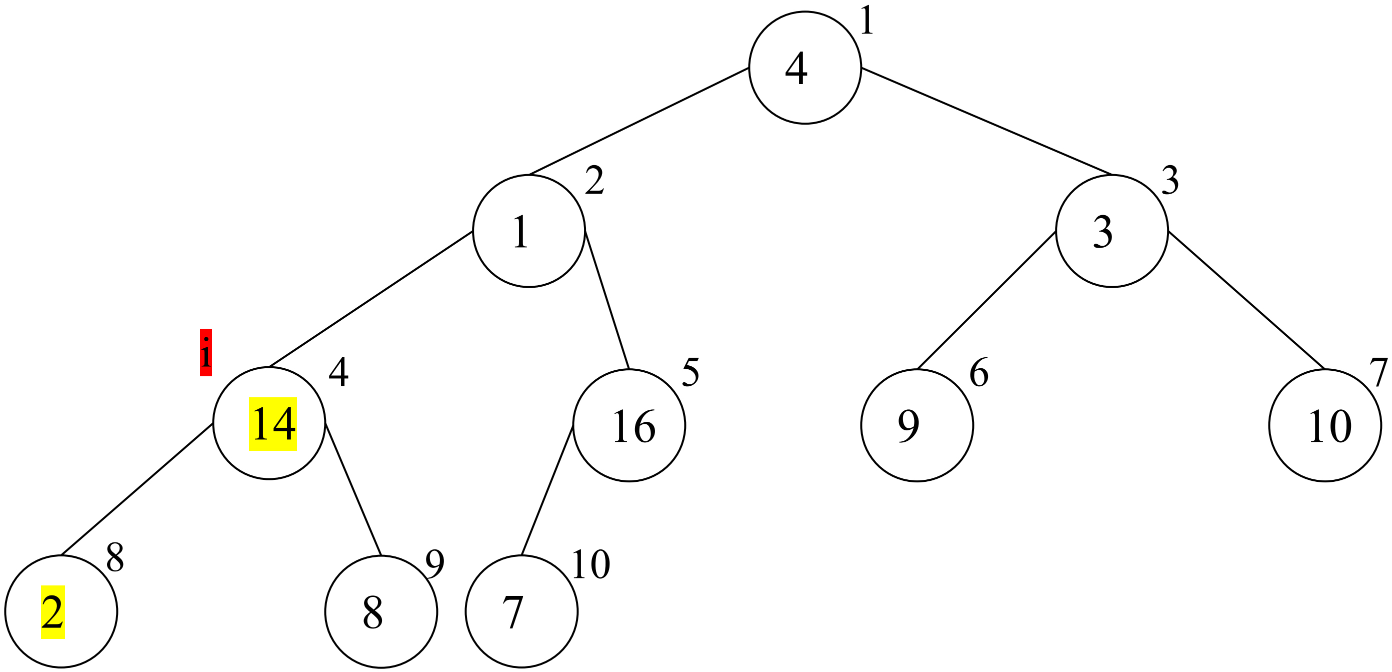
4	1	3	2	16	9	10	14	8	7		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Build-Max-Heap(A)



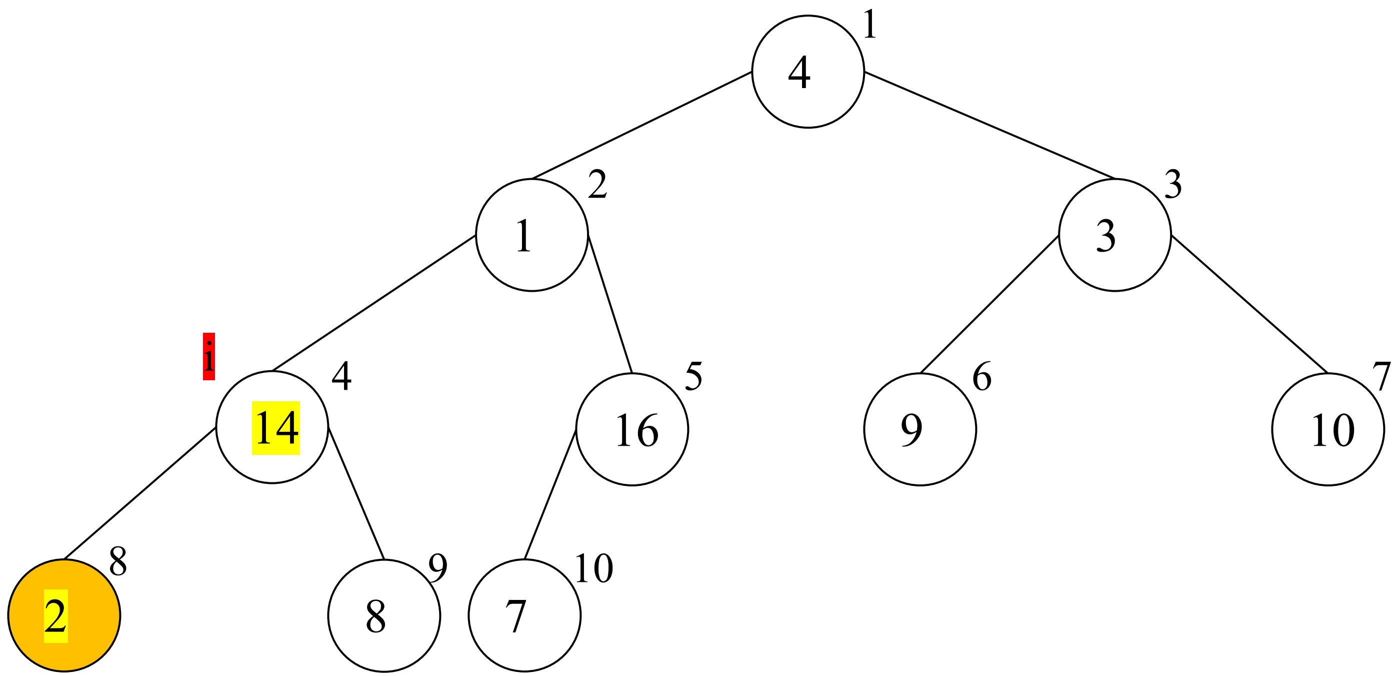
4	1	3	2	16	9	10	14	8	7		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Build-Max-Heap(A)



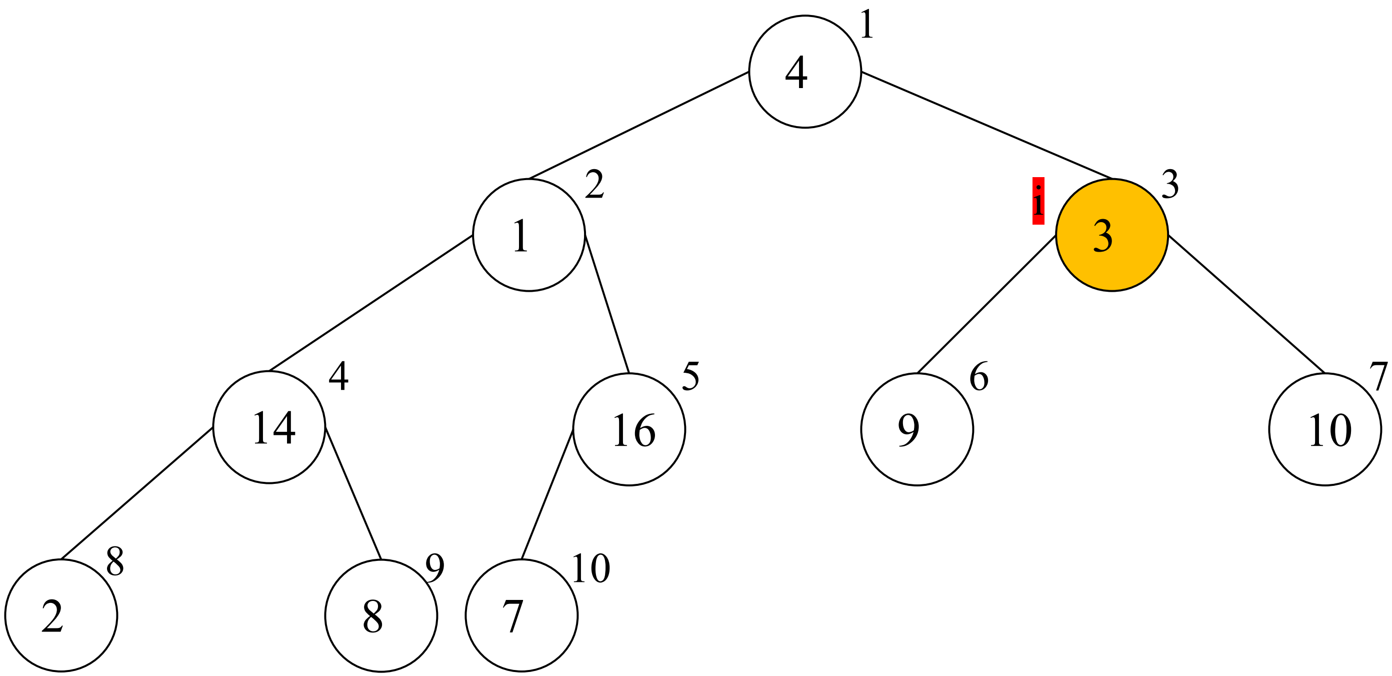
4	1	3	14	16	9	10	2	8	7		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Build-Max-Heap(A)



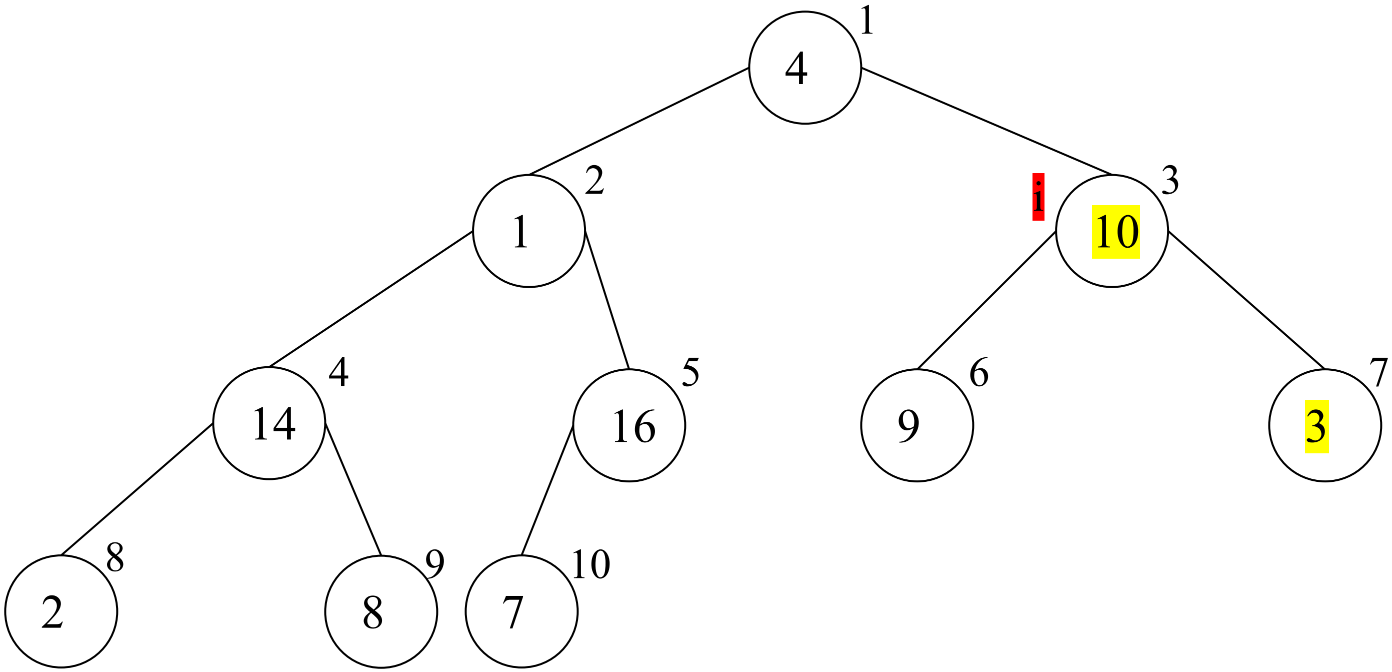
4	1	3	14	16	9	10	2	8	7		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Build-Max-Heap(A)



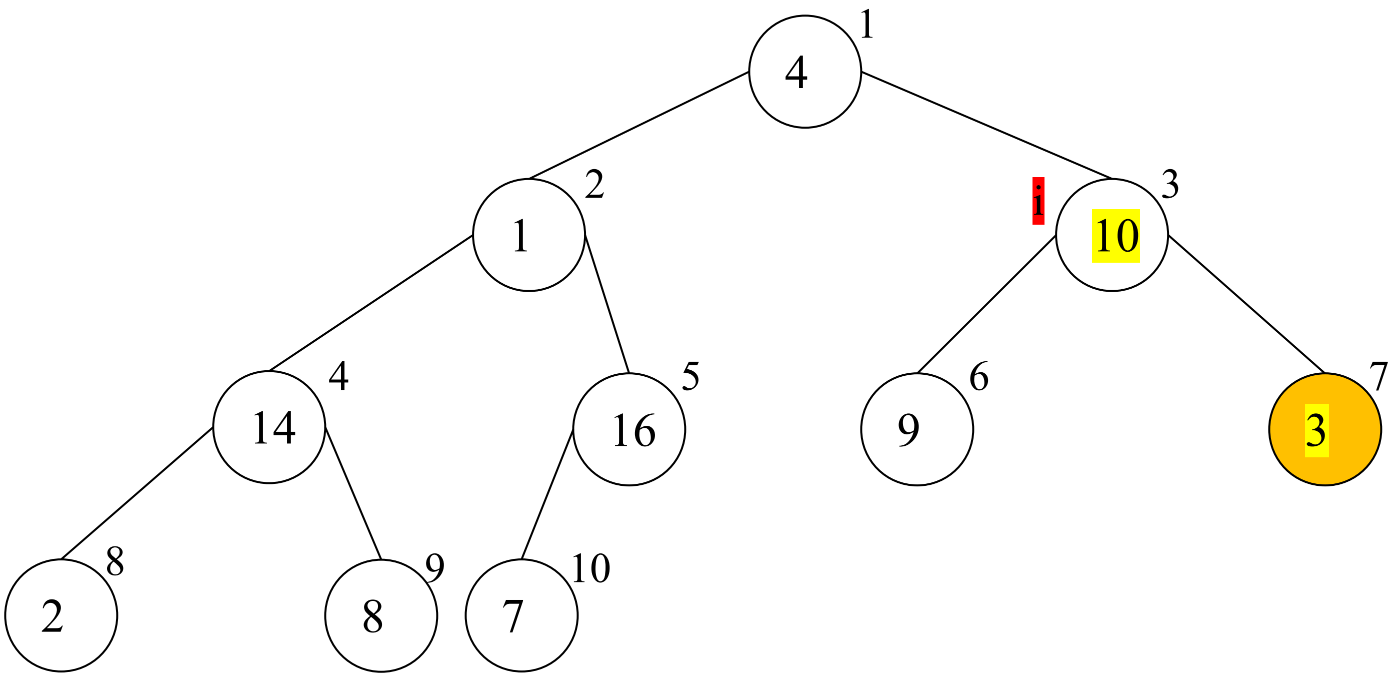
4	1	3	14	16	9	10	2	8	7		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Build-Max-Heap(A)



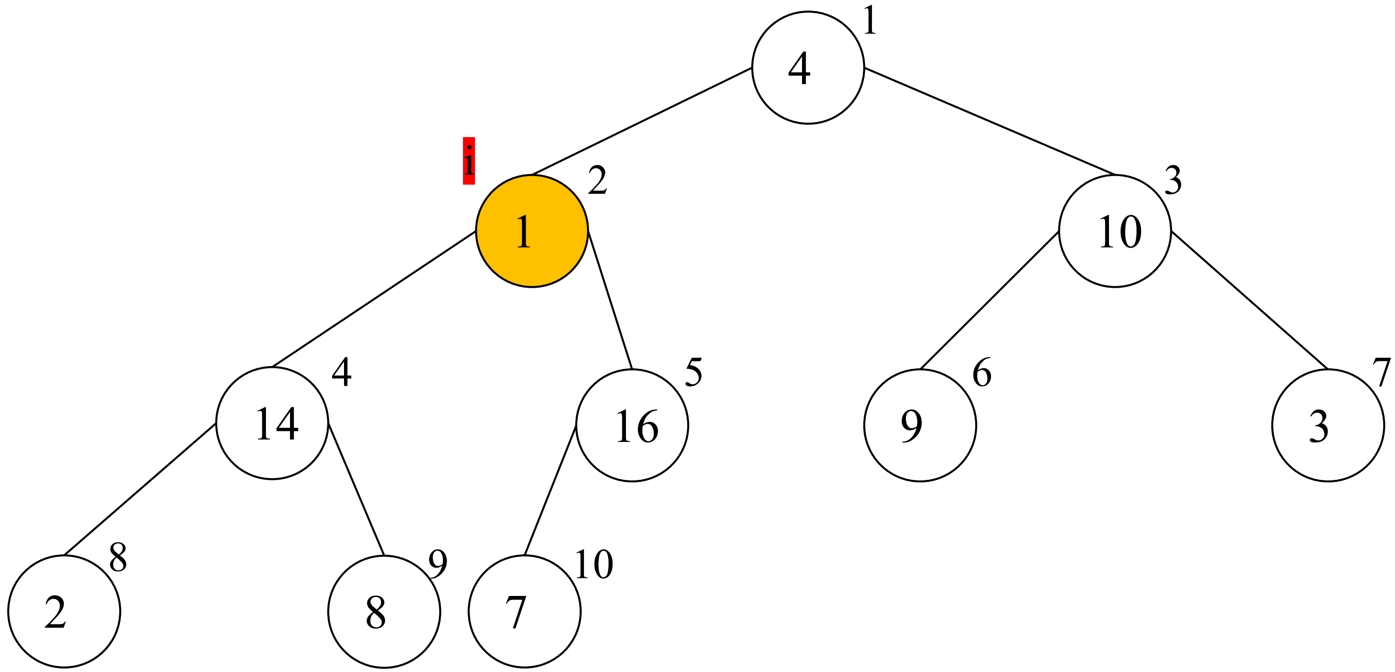
4	1	10	14	16	9	3	2	8	7		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Build-Max-Heap(A)



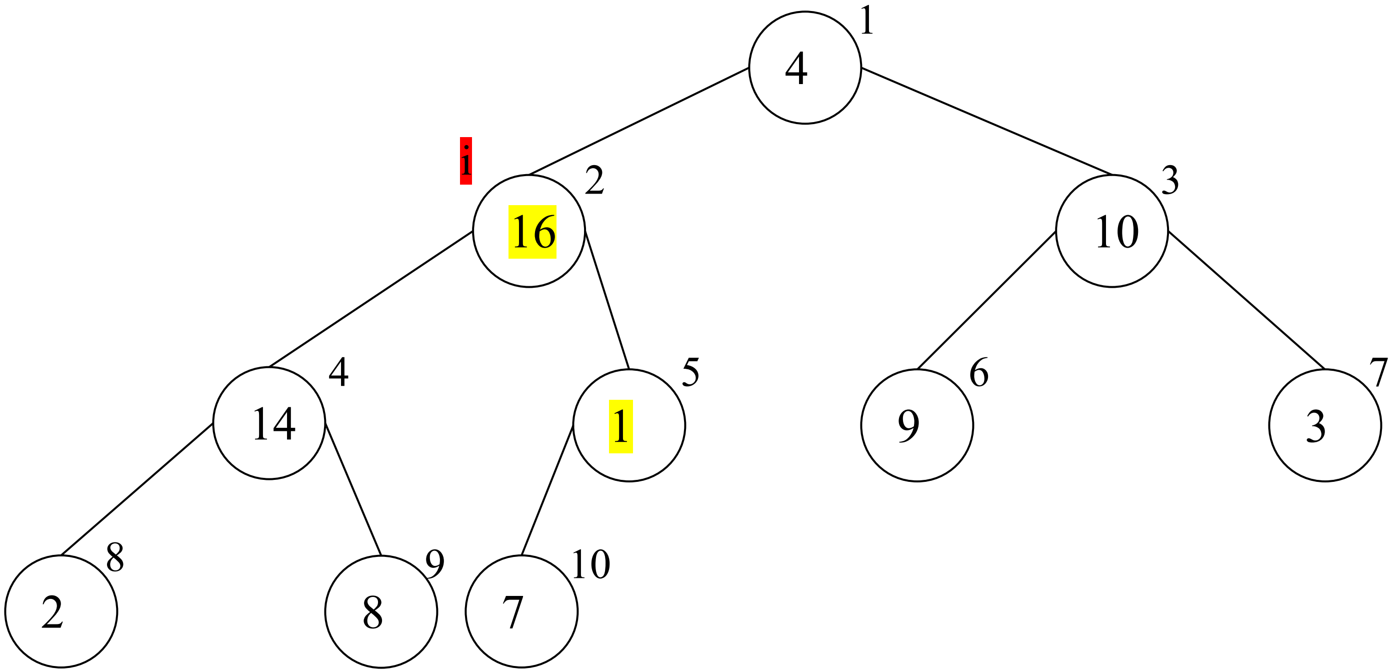
4	1	10	14	16	9	3	2	8	7		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Build-Max-Heap(A)



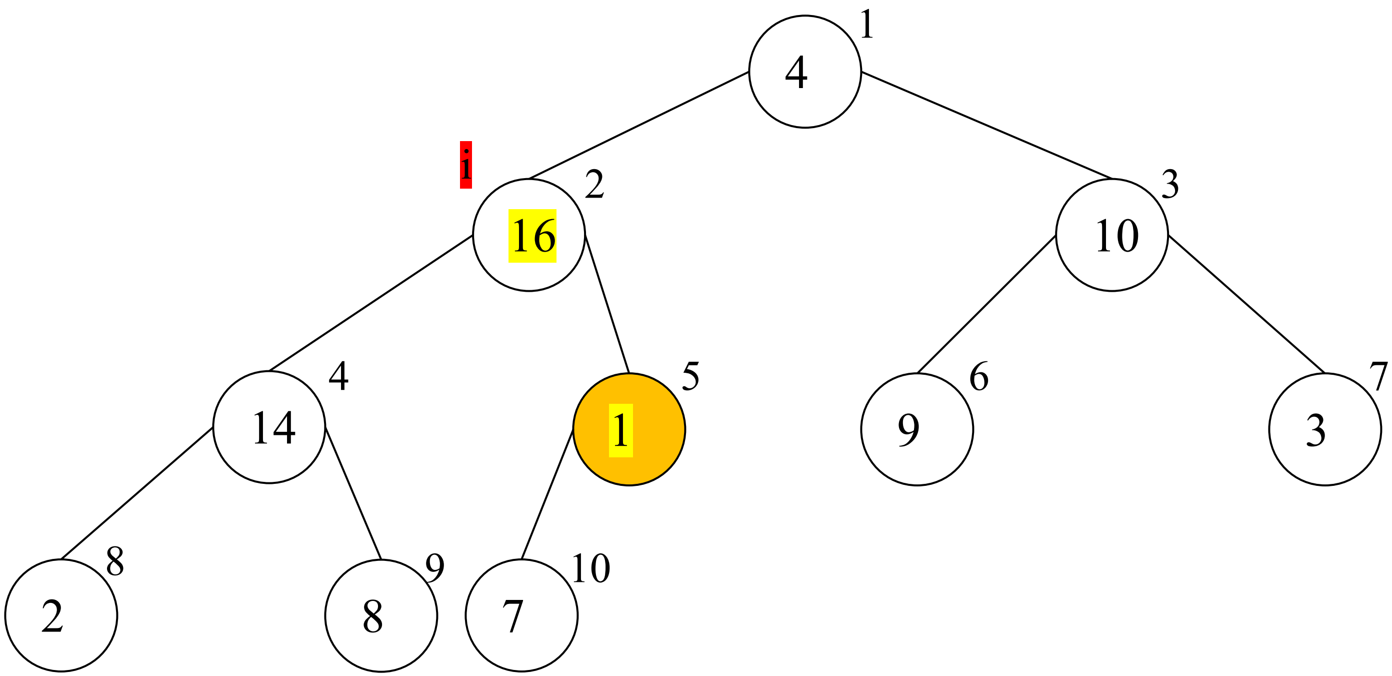
4	1	10	14	16	9	3	2	8	7		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Build-Max-Heap(A)



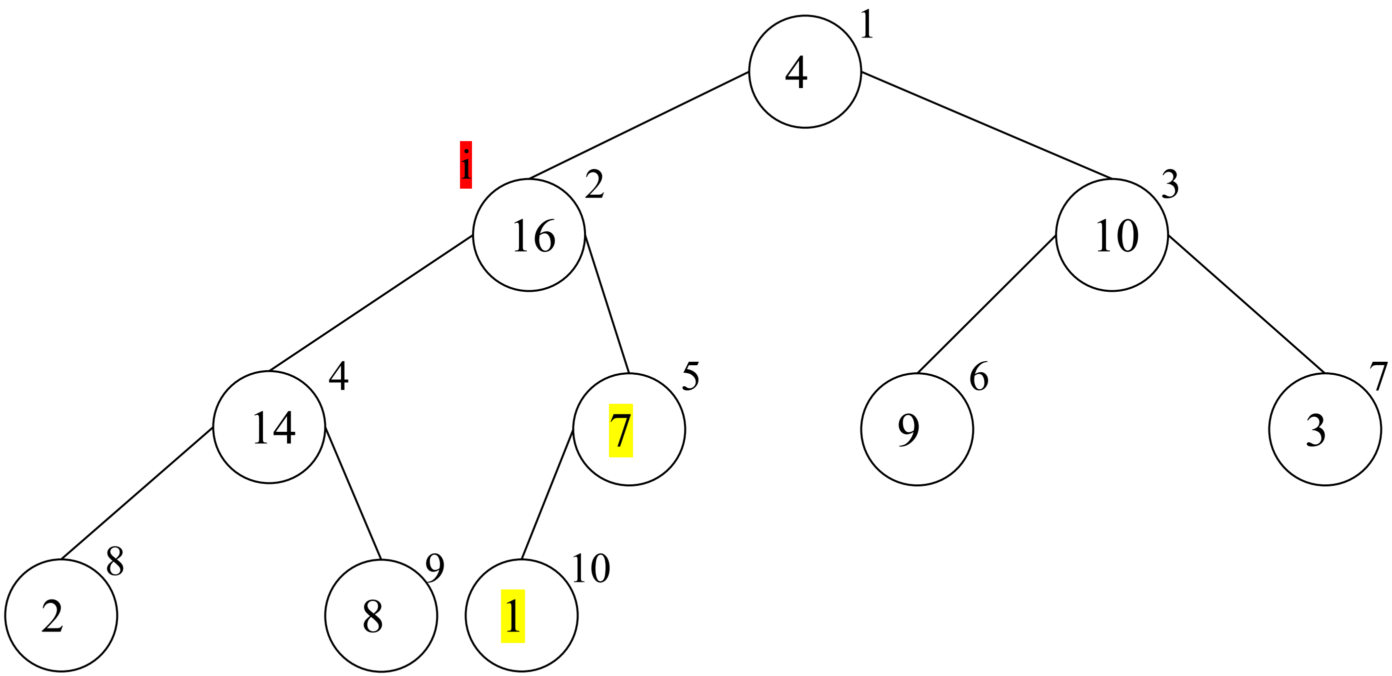
4	16	10	14	1	9	3	2	8	7		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Build-Max-Heap(A)



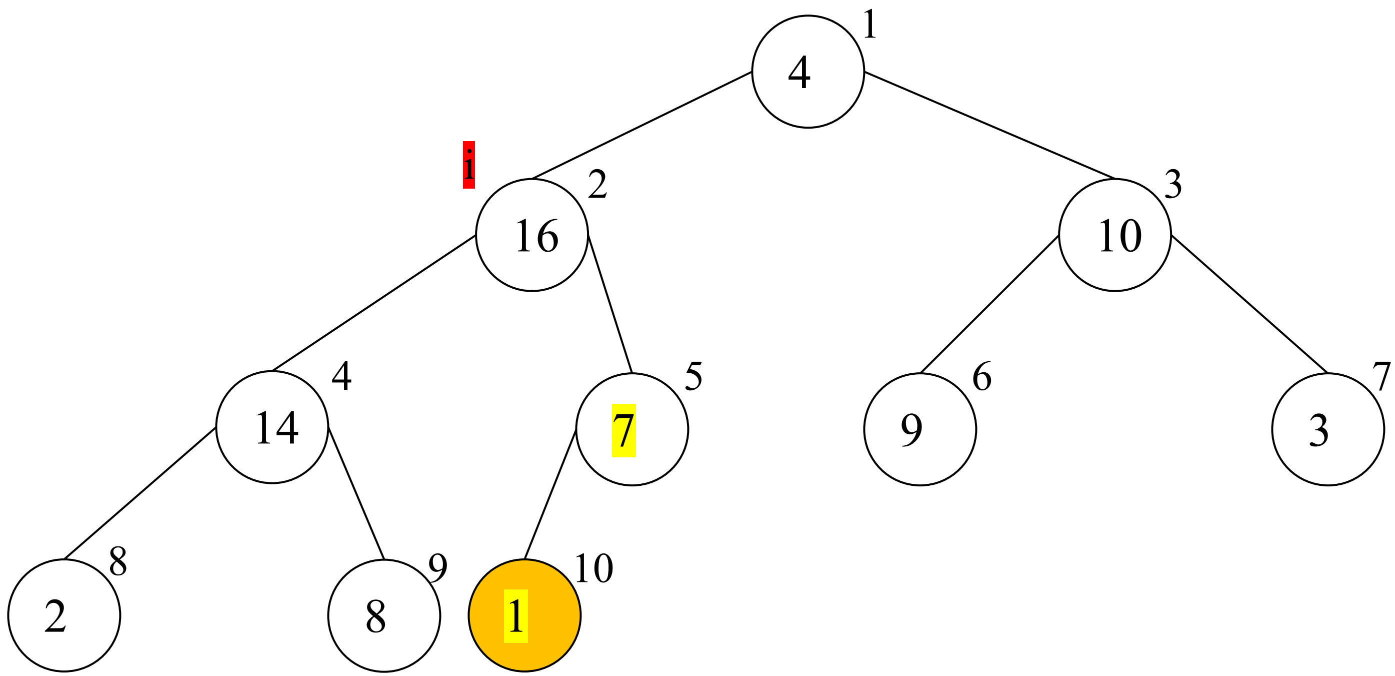
4	16	10	14	1	9	3	2	8	7		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Build-Max-Heap(A)



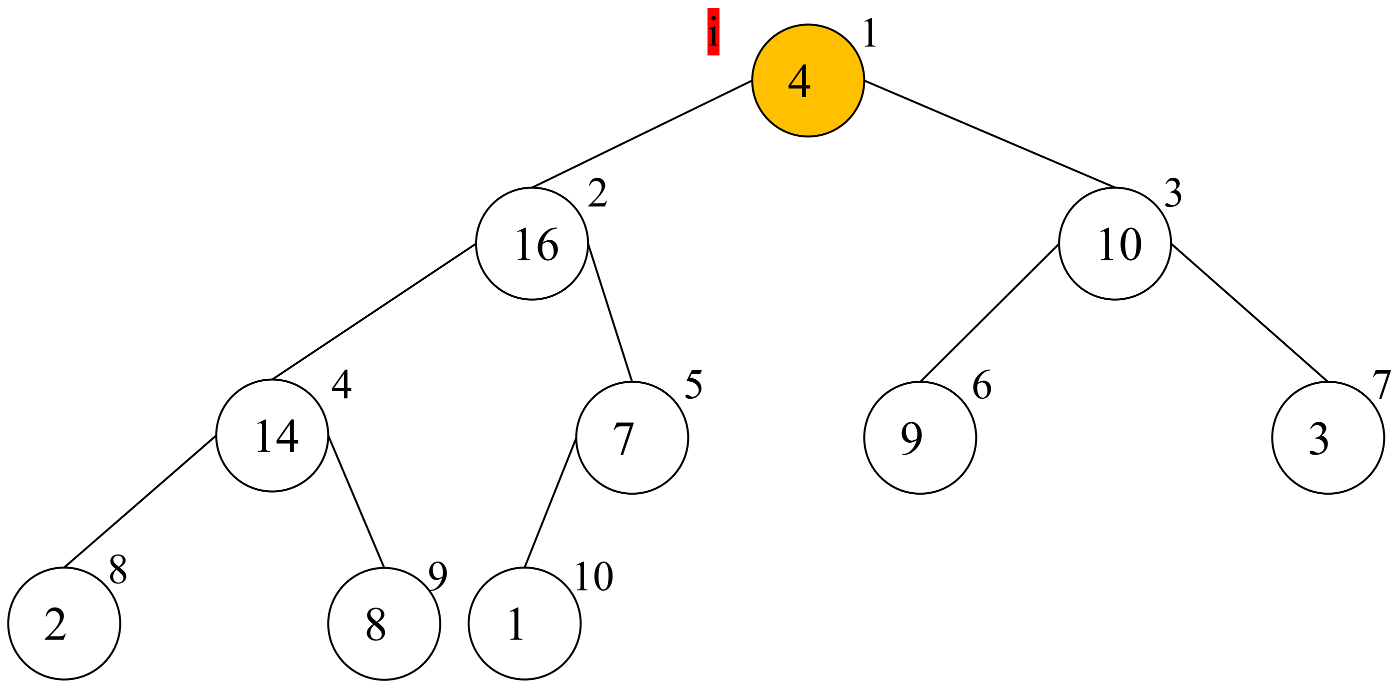
4	16	10	14	7	9	3	2	8	1		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Build-Max-Heap(A)



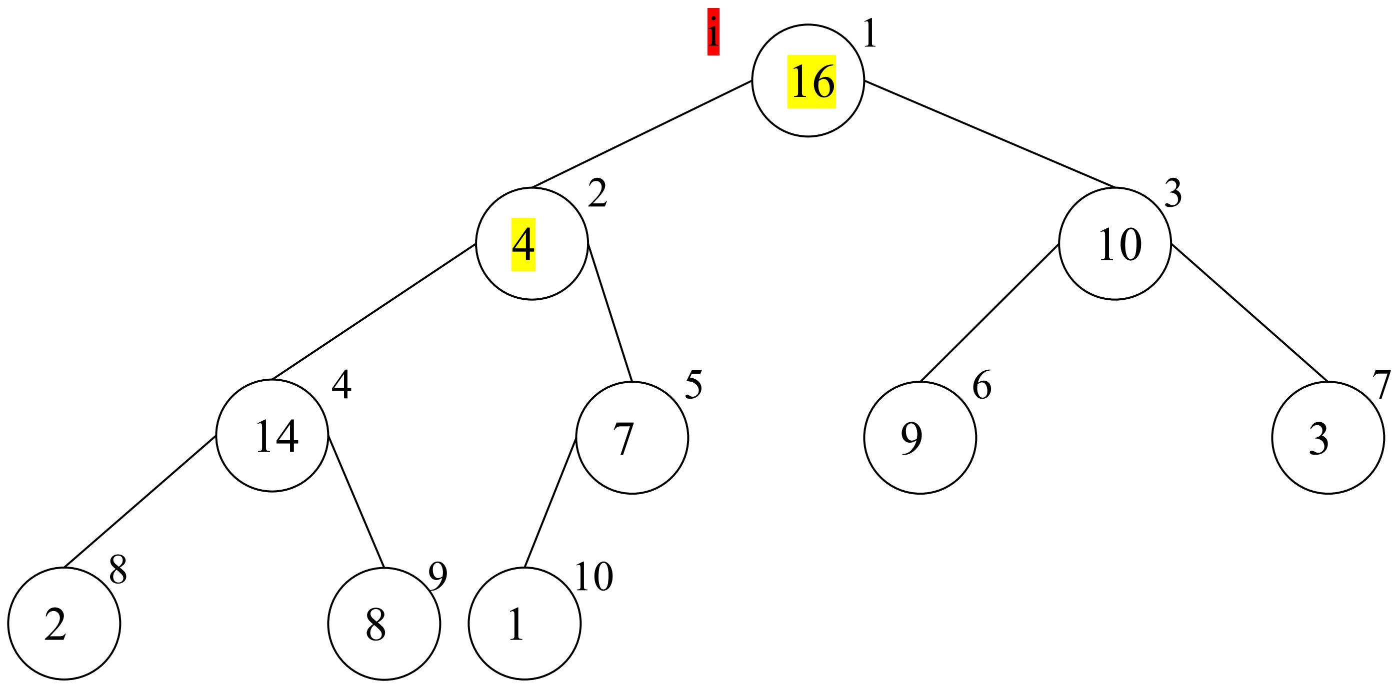
4	16	10	14	7	9	3	2	8	1		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Build-Max-Heap(A)



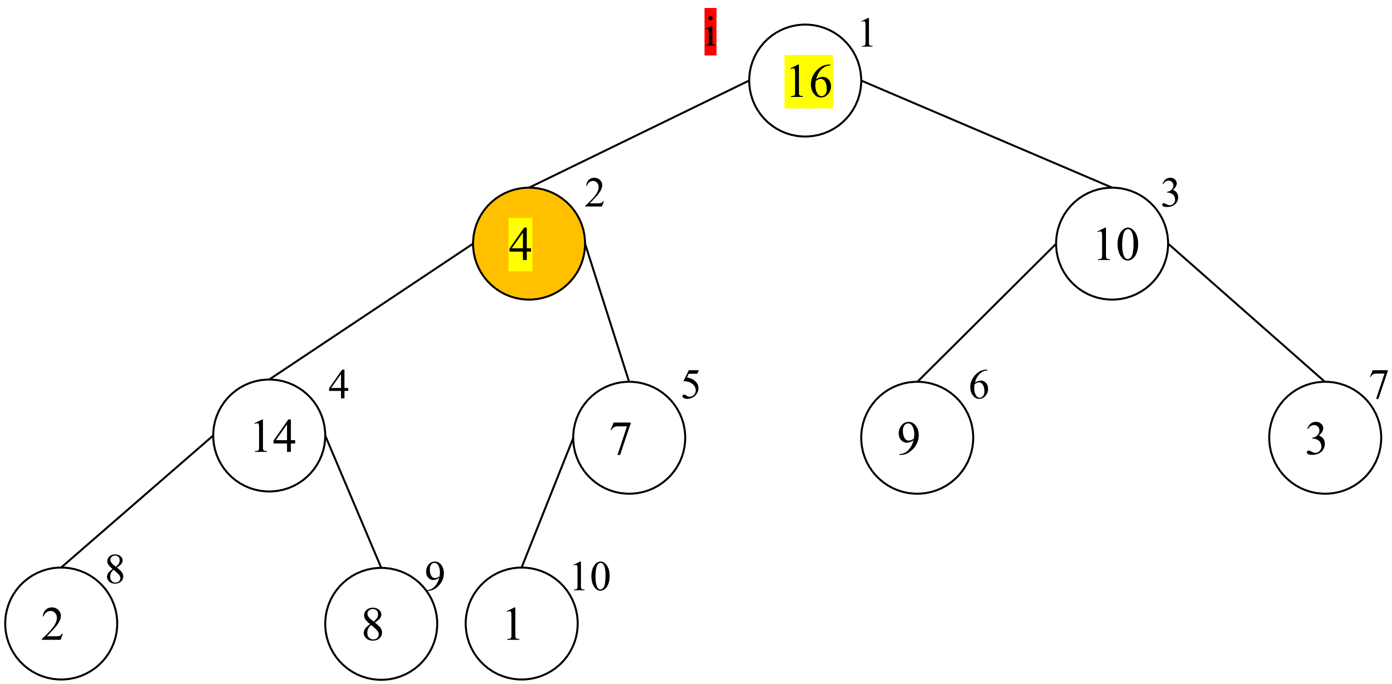
4	16	10	14	7	9	3	2	8	1		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Build-Max-Heap(A)



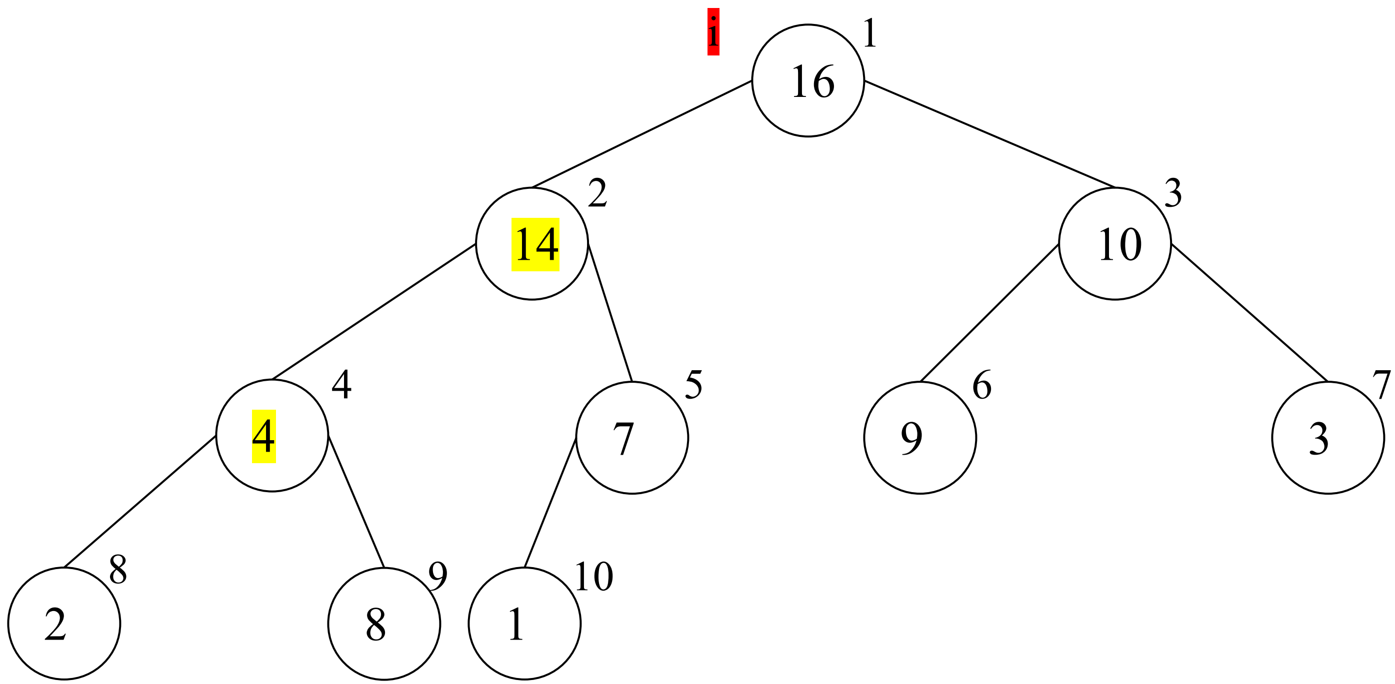
16	4	10	14	7	9	3	2	8	1		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Build-Max-Heap(A)



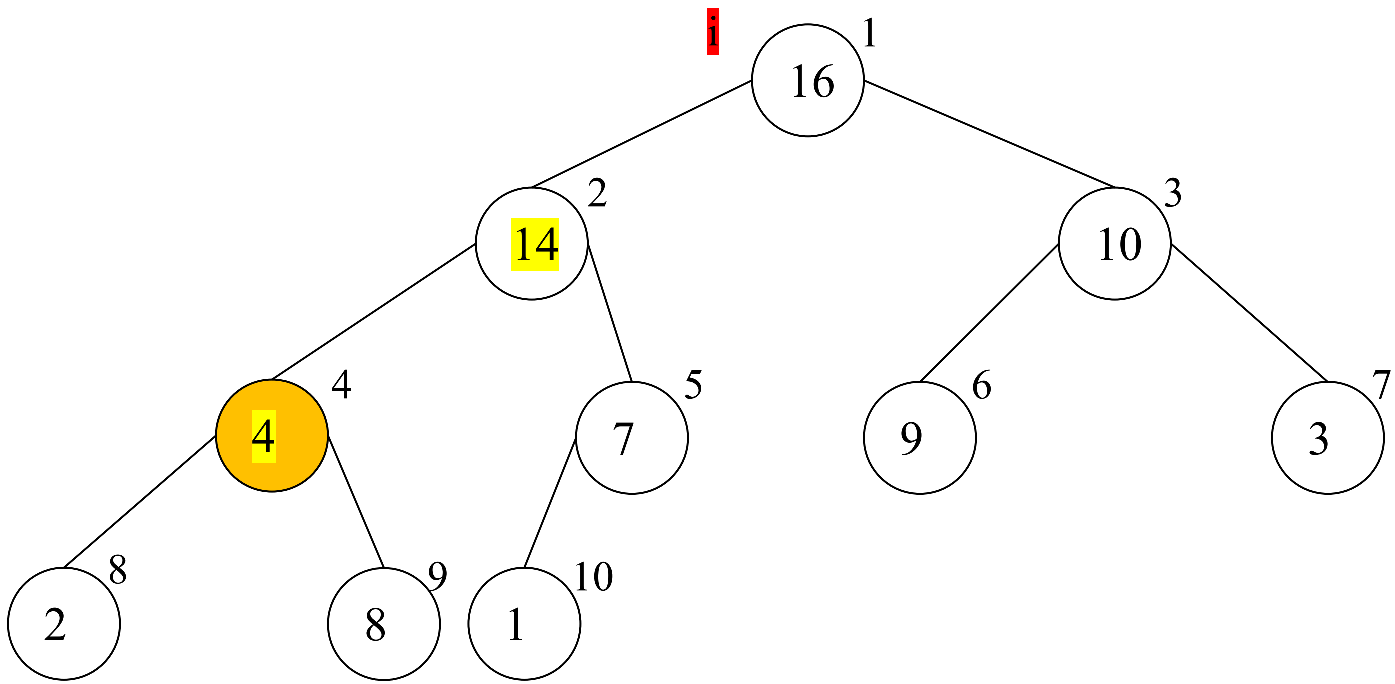
16	4	10	14	7	9	3	2	8	1		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Build-Max-Heap(A)



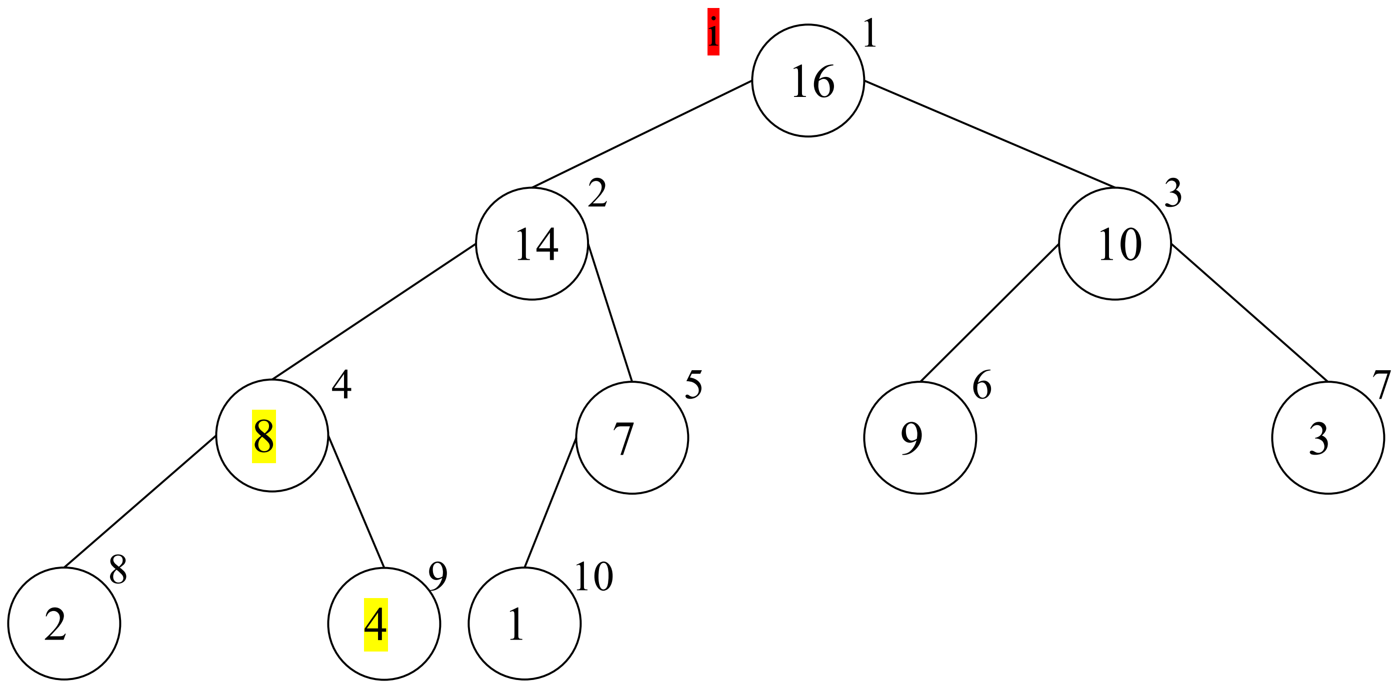
16	14	10	4	7	9	3	2	8	1		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Build-Max-Heap(A)



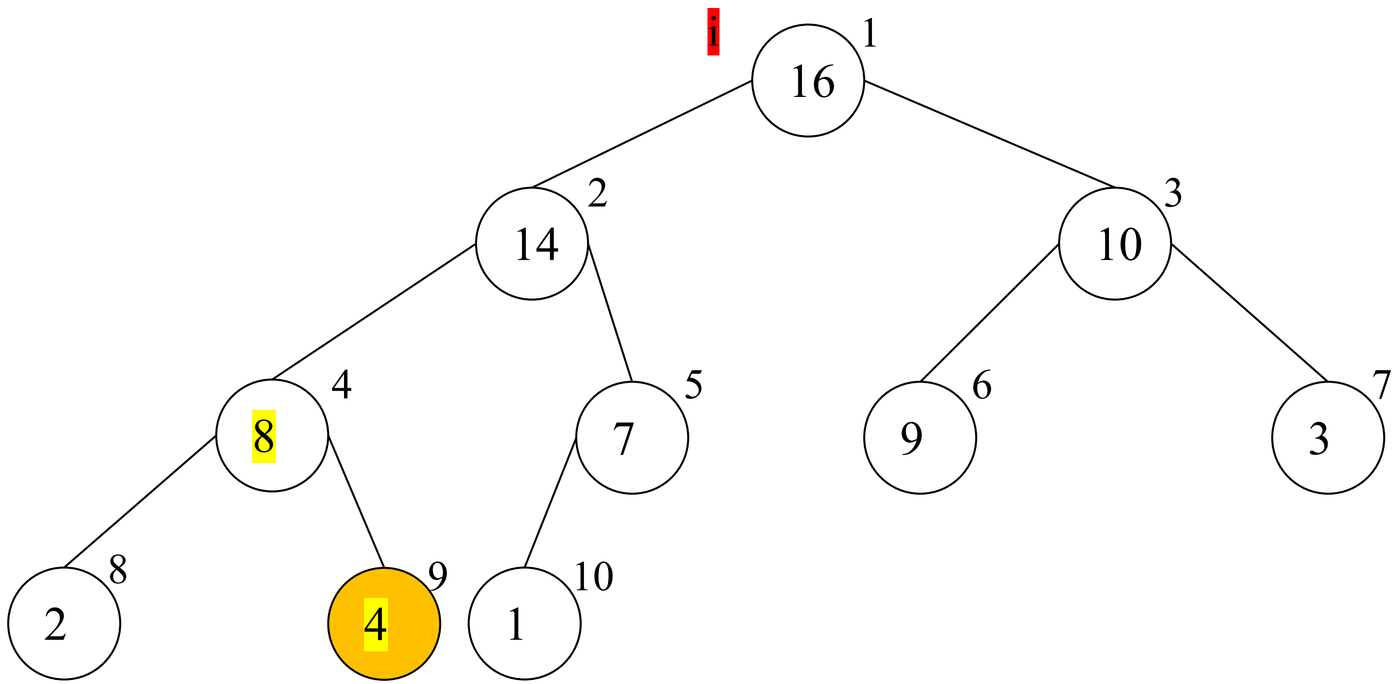
16	14	10	4	7	9	3	2	8	1		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Build-Max-Heap(A)



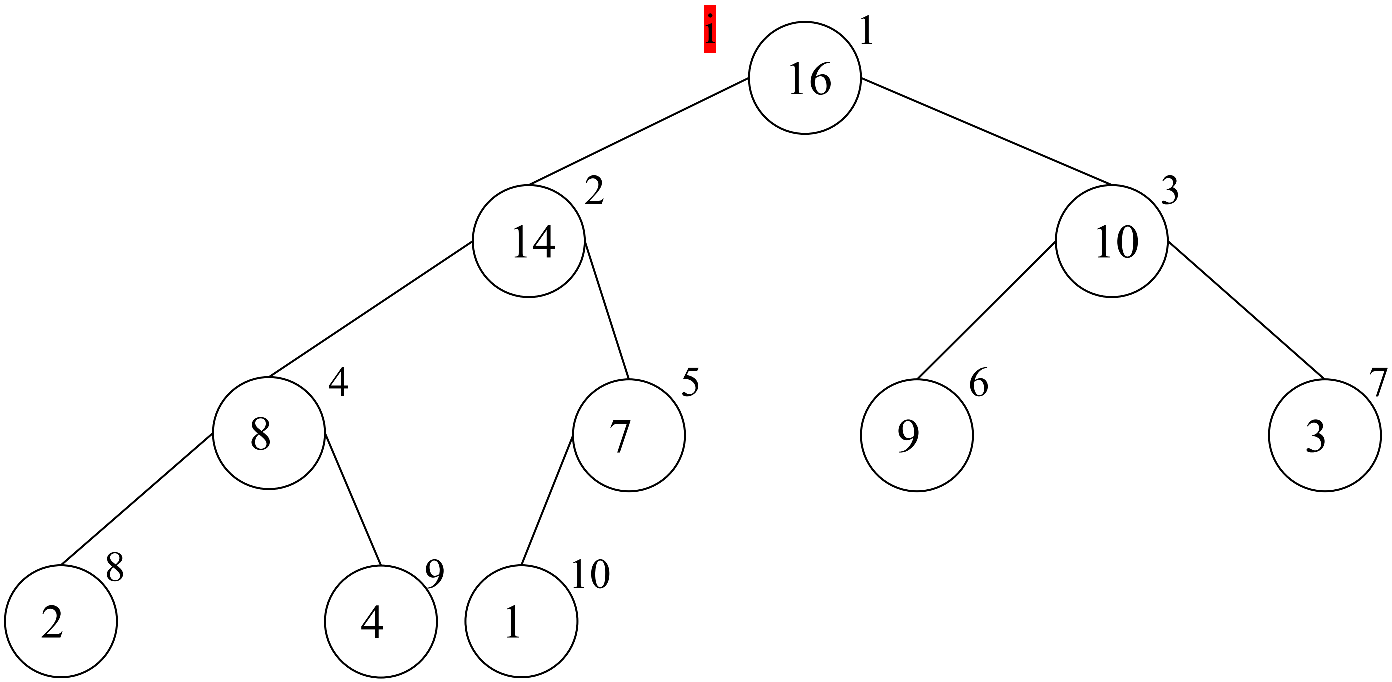
16	14	10	8	7	9	3	2	4	1		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Build-Max-Heap(A)



16	14	10	8	7	9	3	2	4	1		
1	2	3	4	5	6	7	8	9	10	11	12

Example: Build-Max-Heap(A)



16	14	10	8	7	9	3	2	4	1		
1	2	3	4	5	6	7	8	9	10	11	12

Heapsort

HEAPSORT(A)

$\theta(n)$ BUILD-HEAP(A)

<u>for</u> i = length(A) <u>to</u> 2 <u>do</u>

$\theta(n)$ exchange A[1] and A[i]

heap-size[A]—

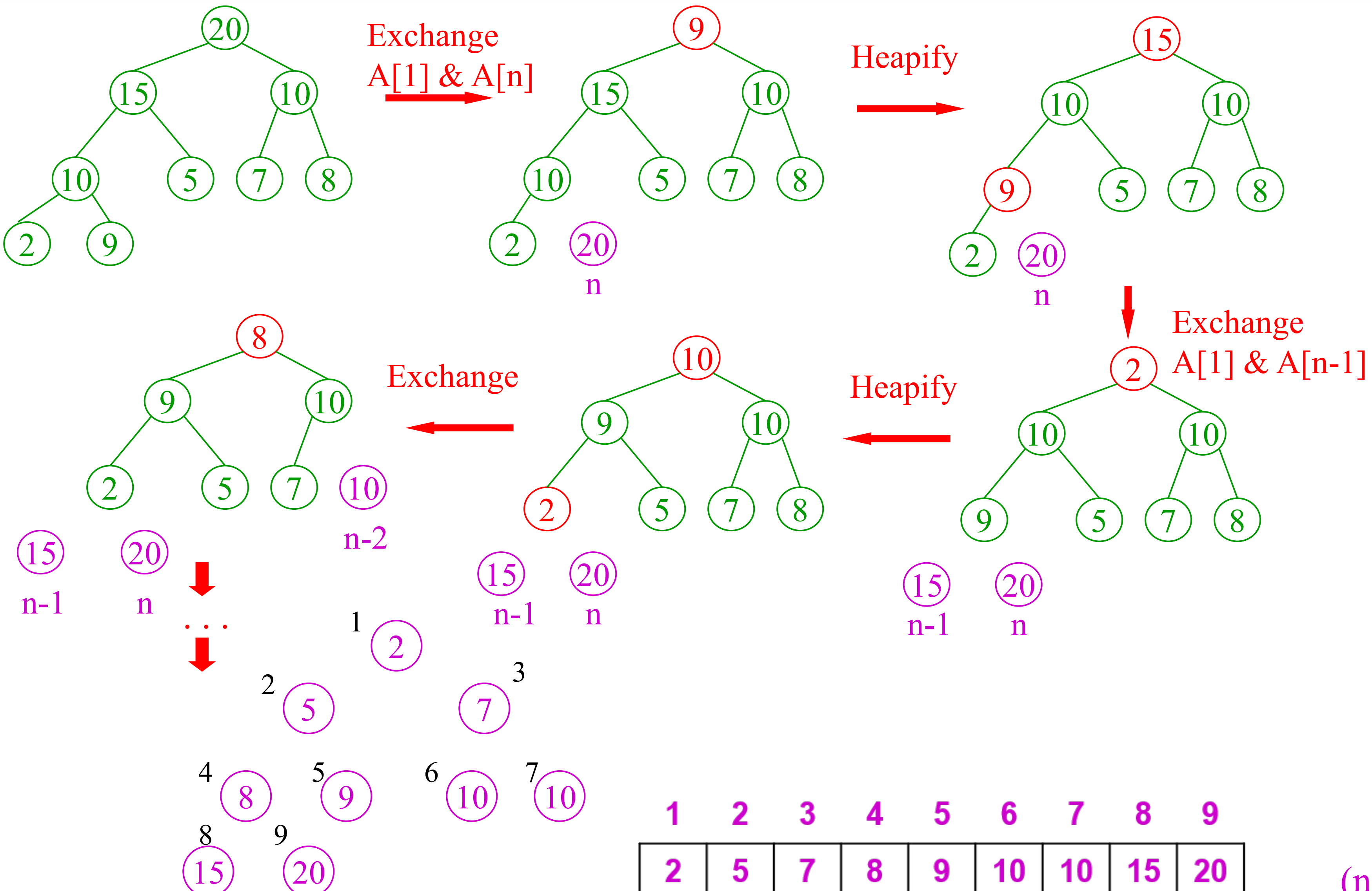
$O(n \log n)$ HEAPIFY(A, 1) ($O(\log n)$ per call to HEAPIFY)

$O(n \log n)$

- The maximum element of A is always stored at the root A[1] (whenever this property is violated, we immediately fix it with HEAPIFY). Thus, it can be correctly put in place by exchanging A[1] with A[heap-size[A]]. Let n = length(A).

A[n] contains maximum element in A[1..n]
A[n-1] contains second maximum element in A[1..n]

Heapsort



1	2	3	4	5	6	7	8	9
2	5	7	8	9	10	10	15	20

(n = 9)

Summary

- We have studied BuildHeap and Heapsort
- We will prove later that BuildHeap has time complexity $\mathbf{O(n)}$ and Heapsort has time complexity $O(n \log n)$, where n is the heap-size.

ASU Ira A. Fulton Schools of Engineering
Arizona State University

Max Heap and Priority Queues Part 4

Ira A. Fulton Schools of Engineering
Arizona State University

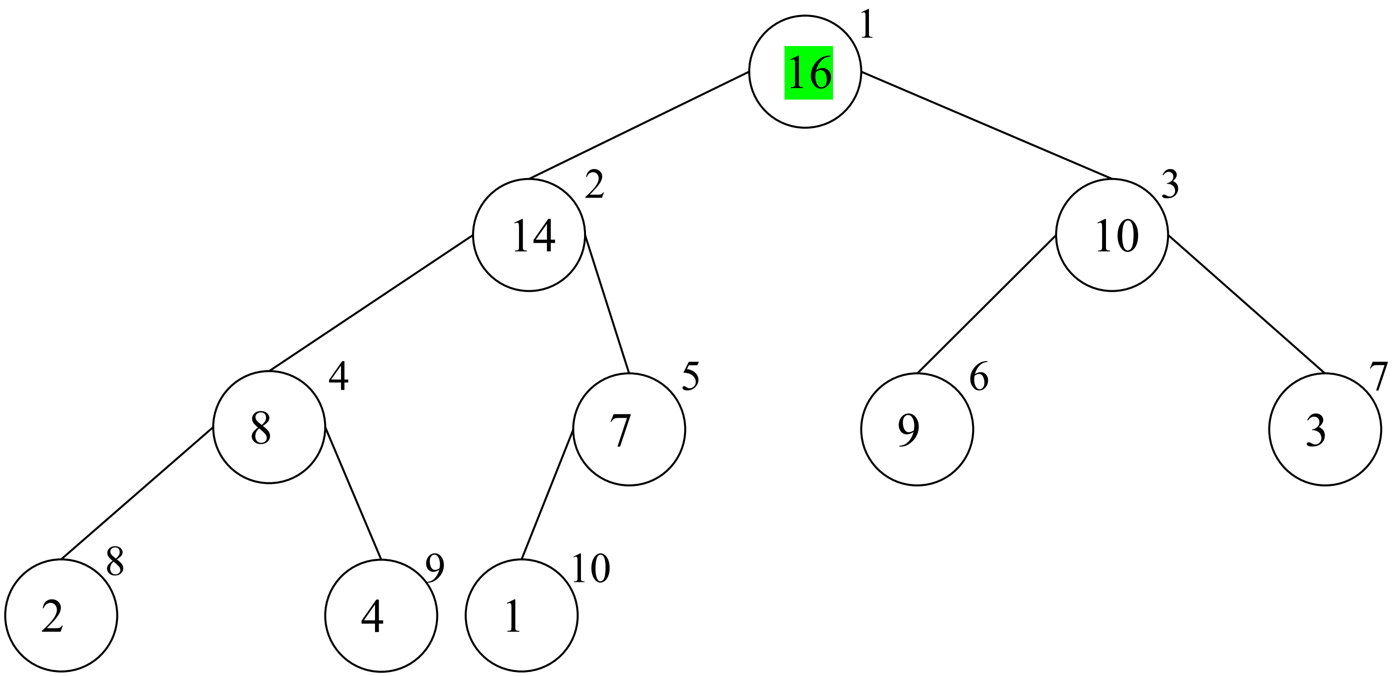
Topics of this lecture

- The Heap Data Structure
- Heapify
- Build-Heap, Heapsort
- Max and ExtractMax
- IncreaseKey and Insertion
- Analysis of Heap Operations

MAX and EXTRACT-MAX

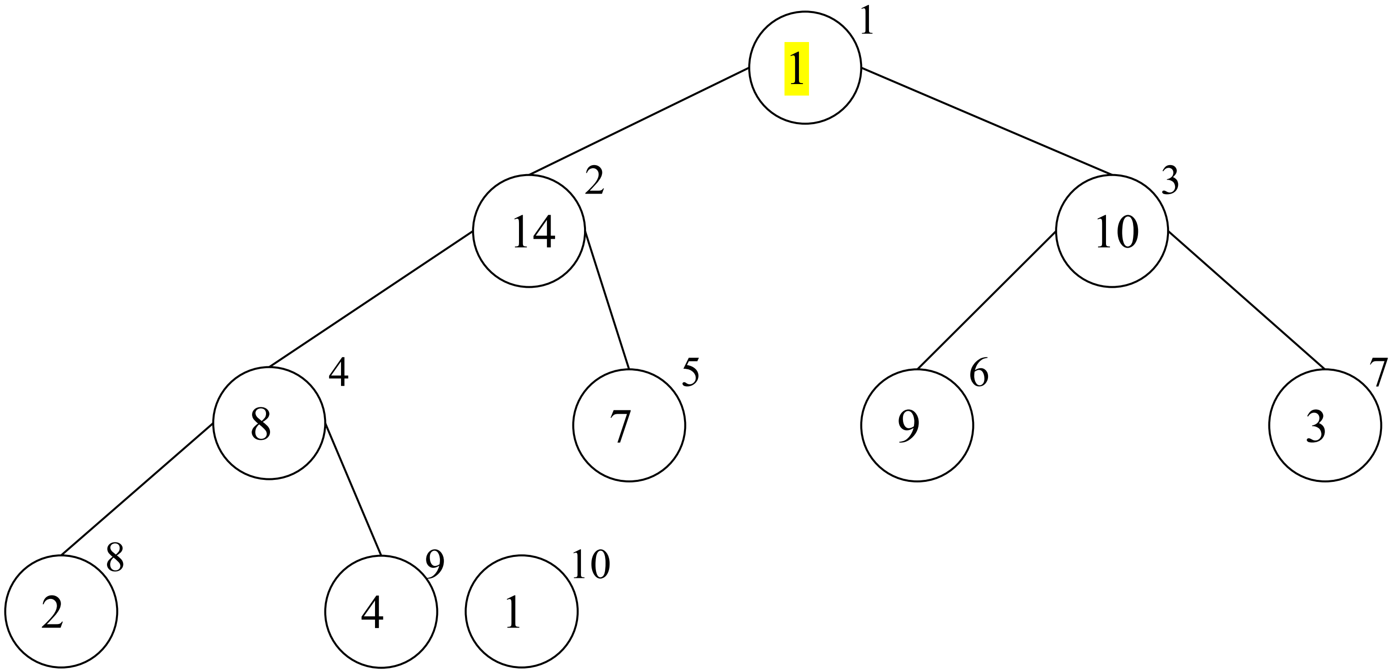
```
HEAP-MAXIMUM(A)
        return A[1]


HEAP-EXTRACT-MAX(A)
        if heap-size[A] < 1 then
            "error: heap empty"
        else
            max = A[1]
            A[1] = A[heap-size[A]]
            heap-size[A]--
            MAX-HEAPIFY(A, 1)
            return max
```

Example: Heap-Extract-Max(A)



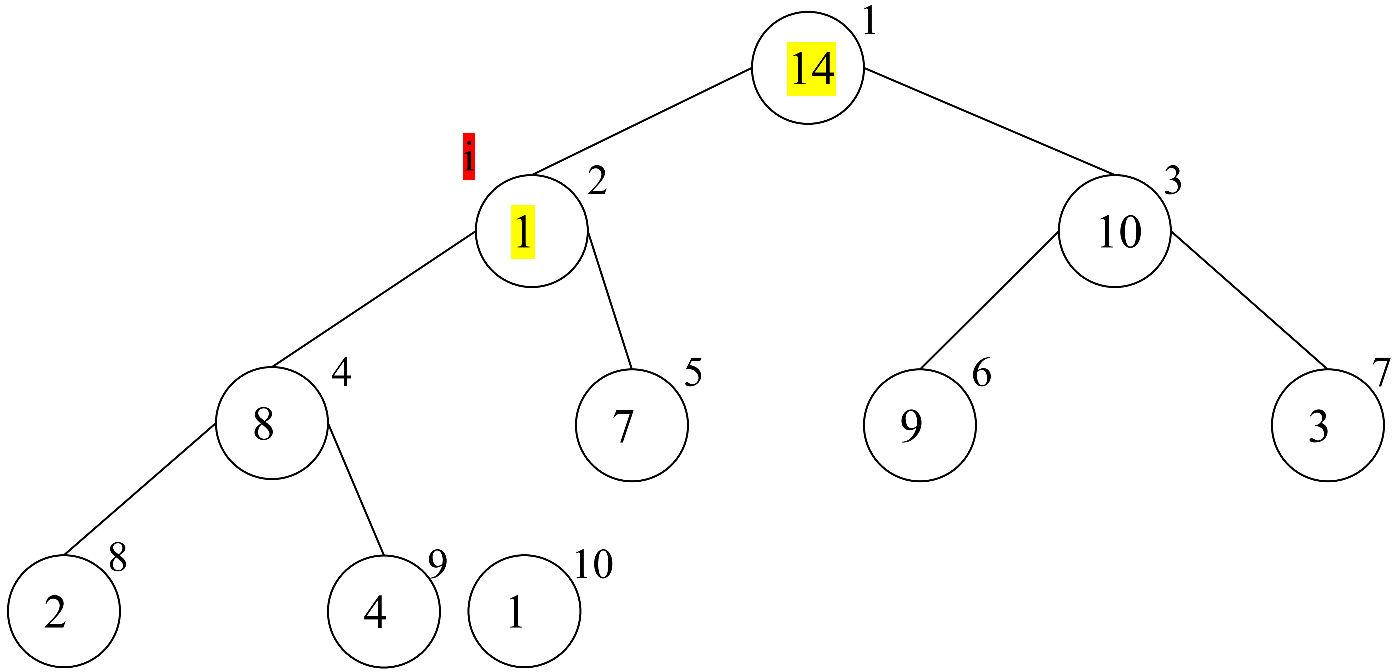
16	14	10	8	7	9	3	2	4	1		
1	2	3	4	5	6	7	8	9	10		

Example: Heap-Extract-Max(A)



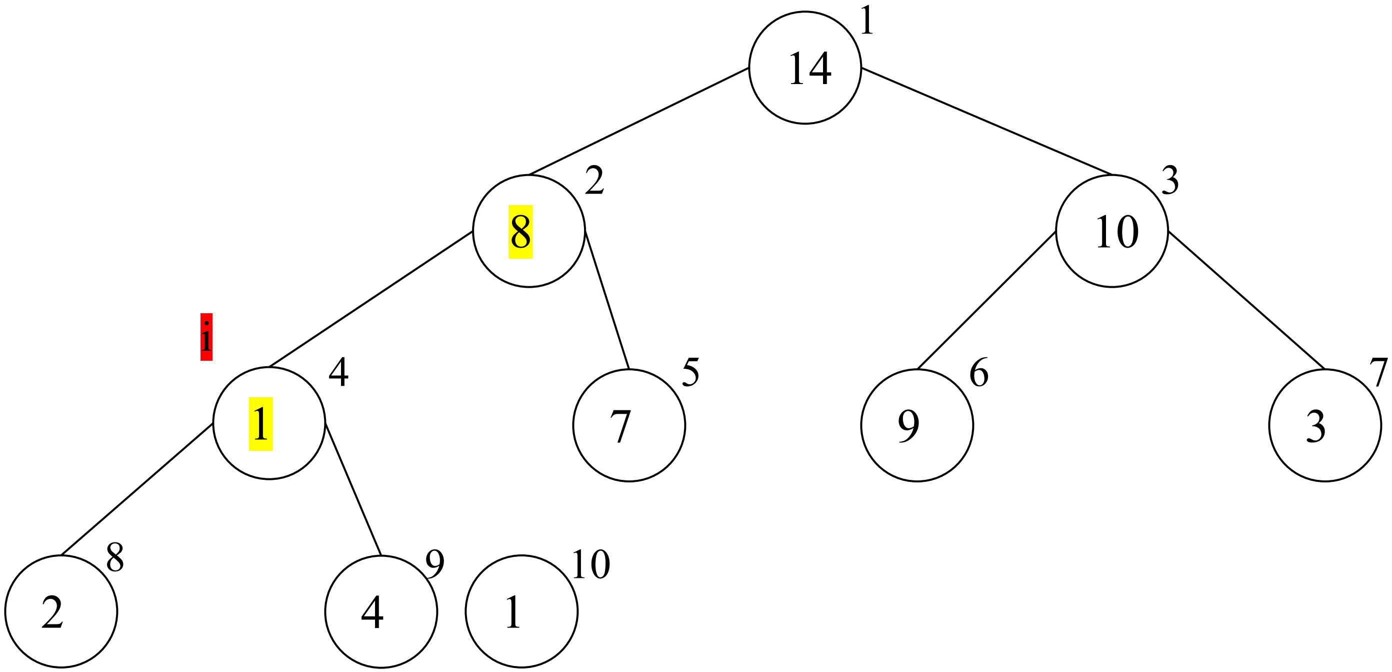
1	14	10	8	7	9	3	2	4	1		
1	2	3	4	5	6	7	8	9			

Example: Heap-Extract-Max(A)



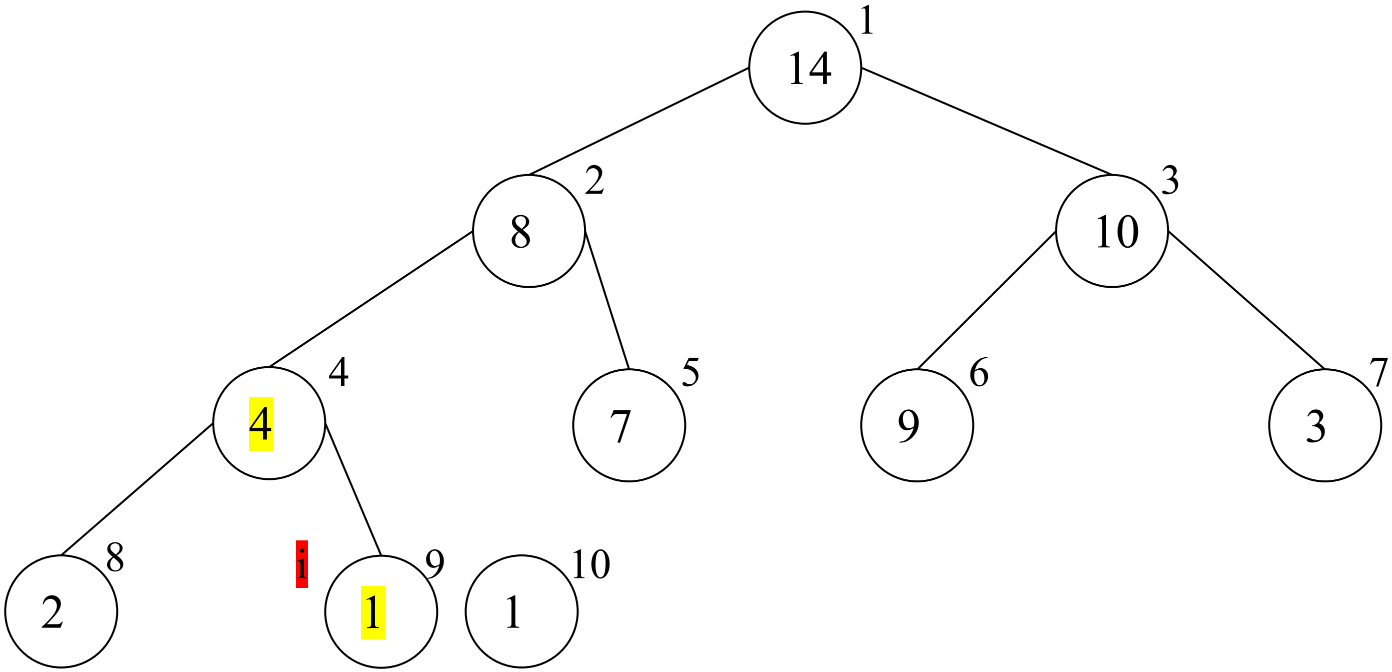
14	1	10	8	7	9	3	2	4	1		
1	2	3	4	5	6	7	8	9			

Example: Heap-Extract-Max(A)



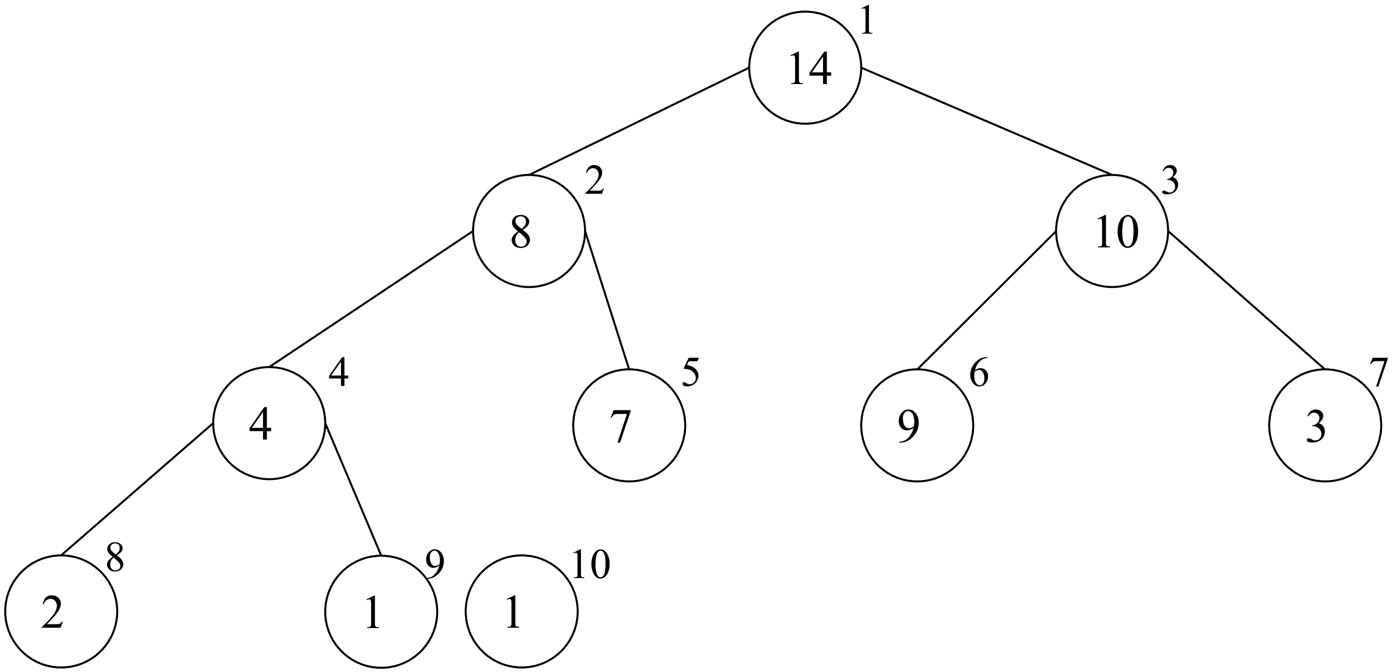
14	8	10	1	7	9	3	2	4	1		
1	2	3	4	5	6	7	8	9			

Example: Heap-Extract-Max(A)



14	8	10	4	7	9	3	2	1	1		
1	2	3	4	5	6	7	8	9			

Example: Heap-Extract-Max(A)



14	8	10	4	7	9	3	2	1	1		
1	2	3	4	5	6	7	8	9			

Topics of this lecture

- **We have studied ExtractMax.**
- **We will prove that the time complexity of ExtractMax is $O(\log n)$, where n is heap-size.**

Ira A. Fulton Schools of
Engineering
Arizona State University

Max Heap and Priority Queues Part 5

Ira A. Fulton Schools of Engineering
Arizona State University

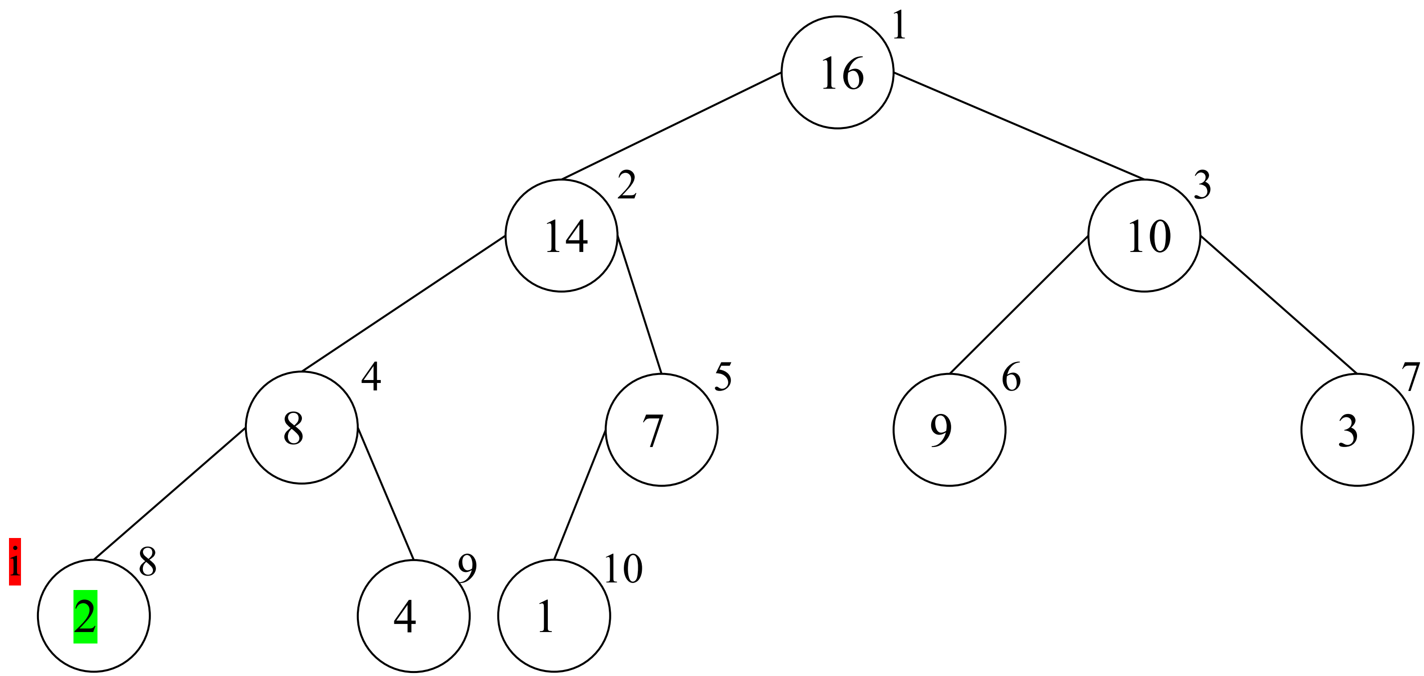
Topics of this lecture

- The Heap Data Structure
- Heapify
- Build-Heap, Heapsort
- Max and ExtractMax
- IncreaseKey and Insertion
- Analysis of Heap Operations

INCREASE-KEY

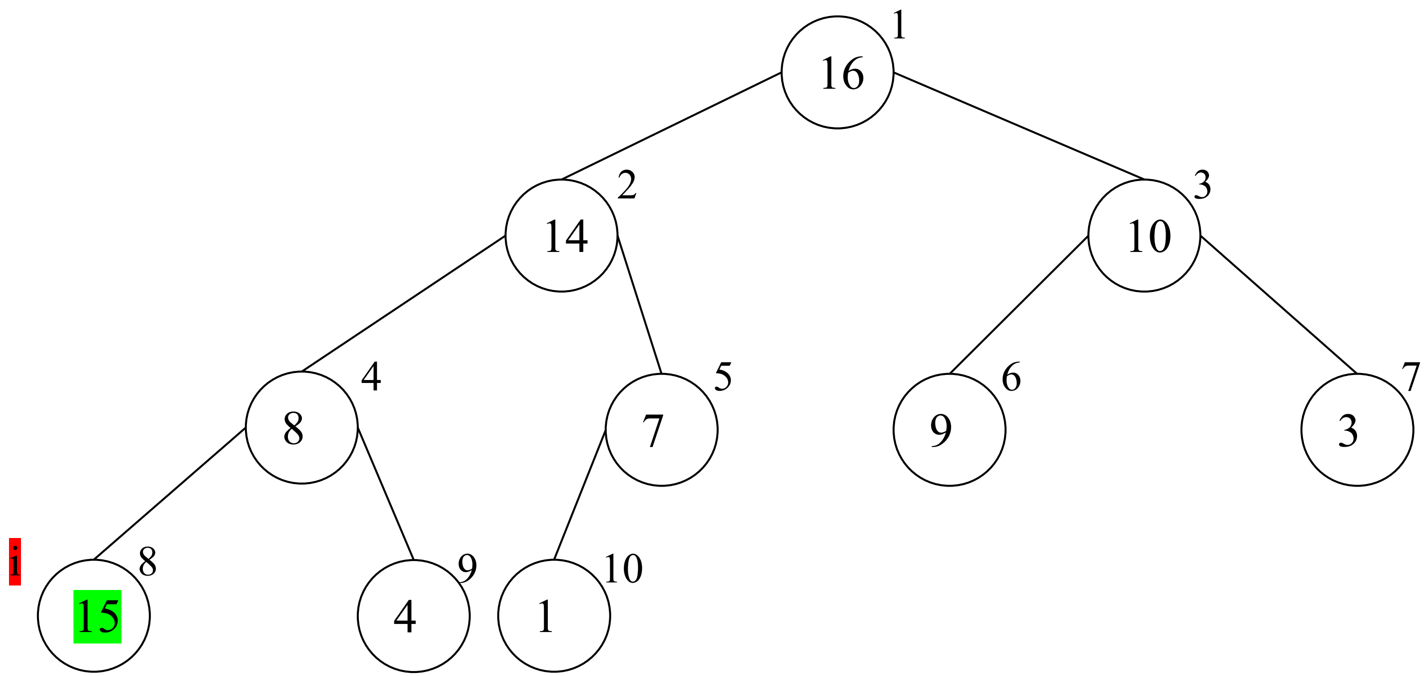
```
HEAP-INCREASE-KEY(A, i, key)
  if key < A[i] then
          "error: new key is smaller than current key"
  else
          A[i] := key
          while i > 1 and A[PARENT(i)] < A[i]
                  exchange A[i] with A[PARENT(i)]
                  i := PARENT(i)
```

Example: Max-Heap-Increase-Key(A, 8, 15)



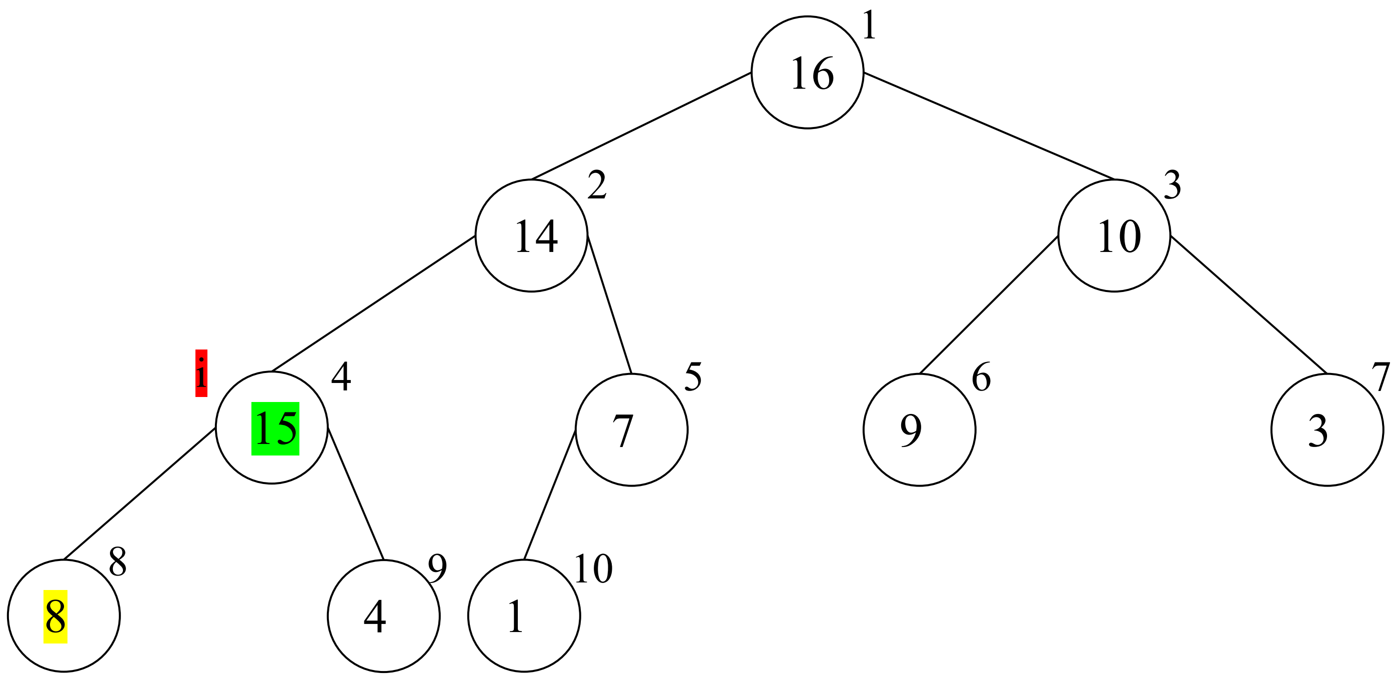
16	14	10	8	7	9	3	2	4	1		
1	2	3	4	5	6	7	8	9	10		

Example: Max-Heap-Increase-Key(A, 8, 15)



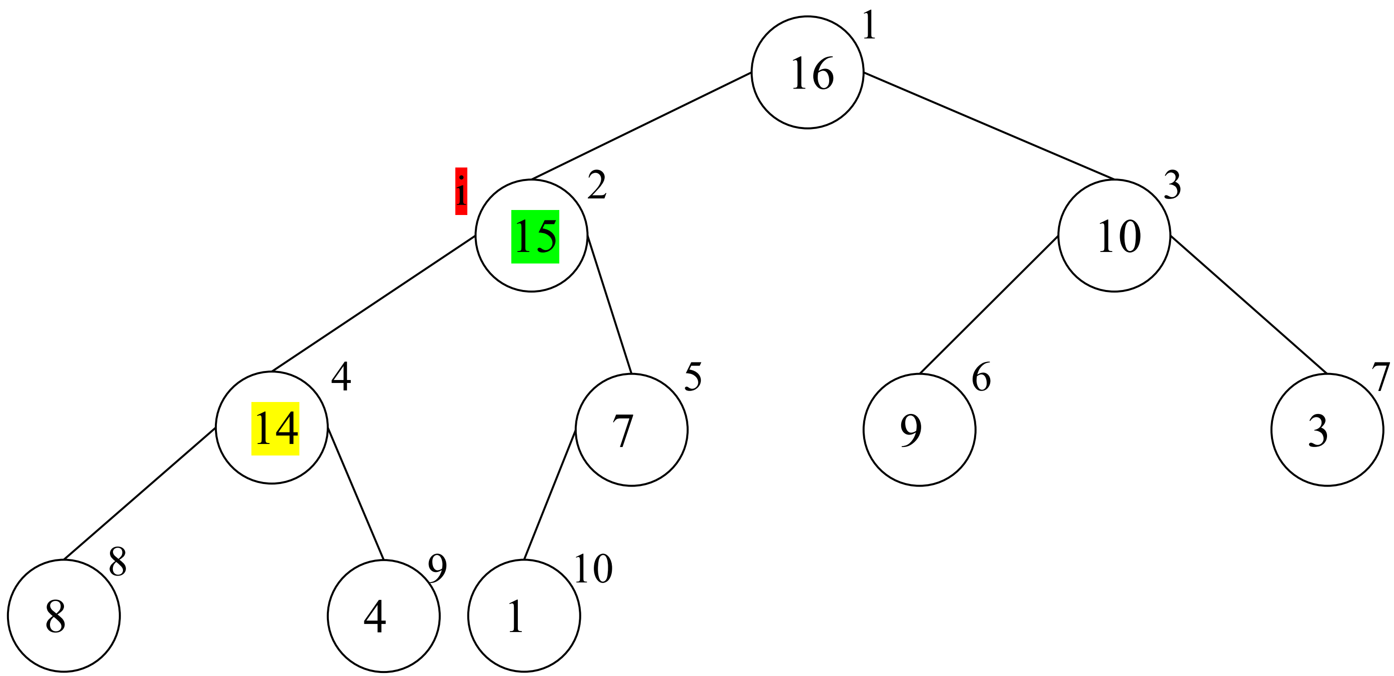
16	14	10	8	7	9	3	15	4	1		
1	2	3	4	5	6	7	8	9	10		

Example: Max-Heap-Increase-Key(A, 8, 15)



16	14	10	15	7	9	3	8	4	1		
1	2	3	4	5	6	7	8	9	10		

Example: Max-Heap-Increase-Key(A, 8, 15)

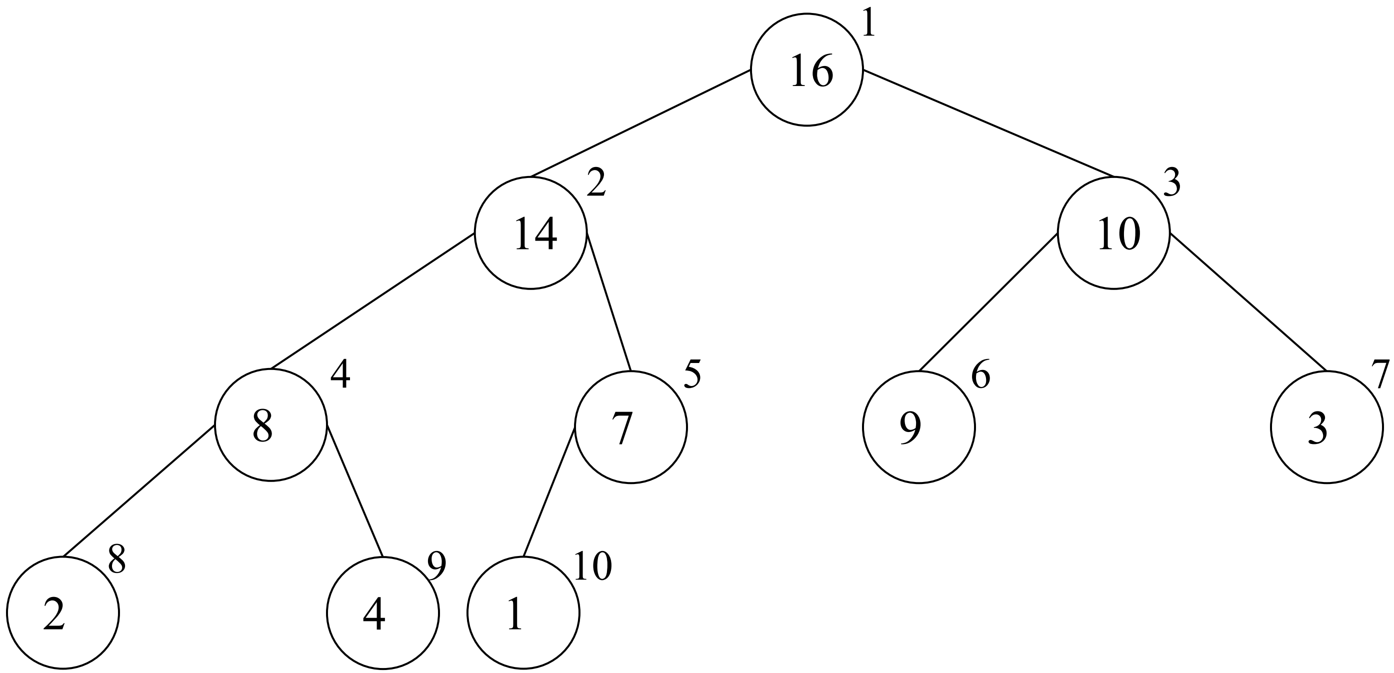


16	15	10	14	7	9	3	8	4	1		
1	2	3	4	5	6	7	8	9	10		

INSERTION

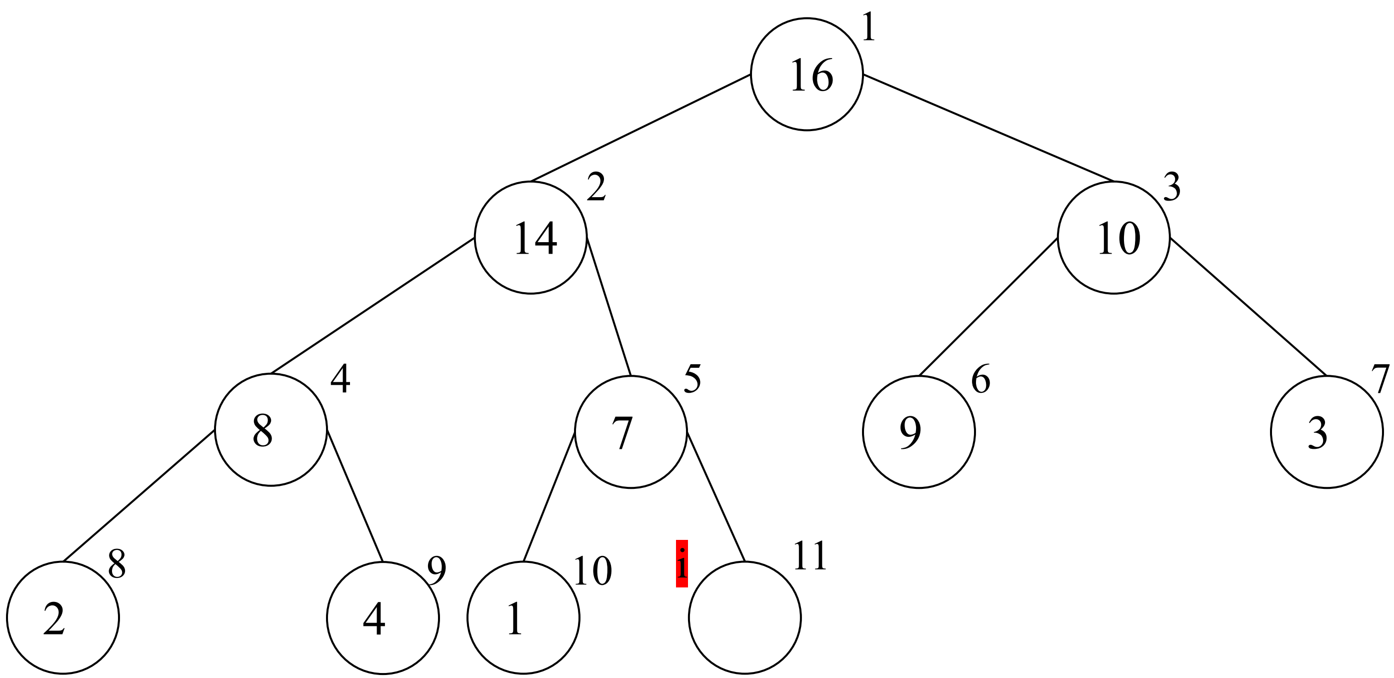
```
MAX-HEAP-INSERT(A, key)
        heap-size[A]++
        i := heap-size[A]

         A[i] = -∞
        HEAP-INCREASE-KEY(A, i, key)
```

Example: Max-Heap-Insert(A, 13)



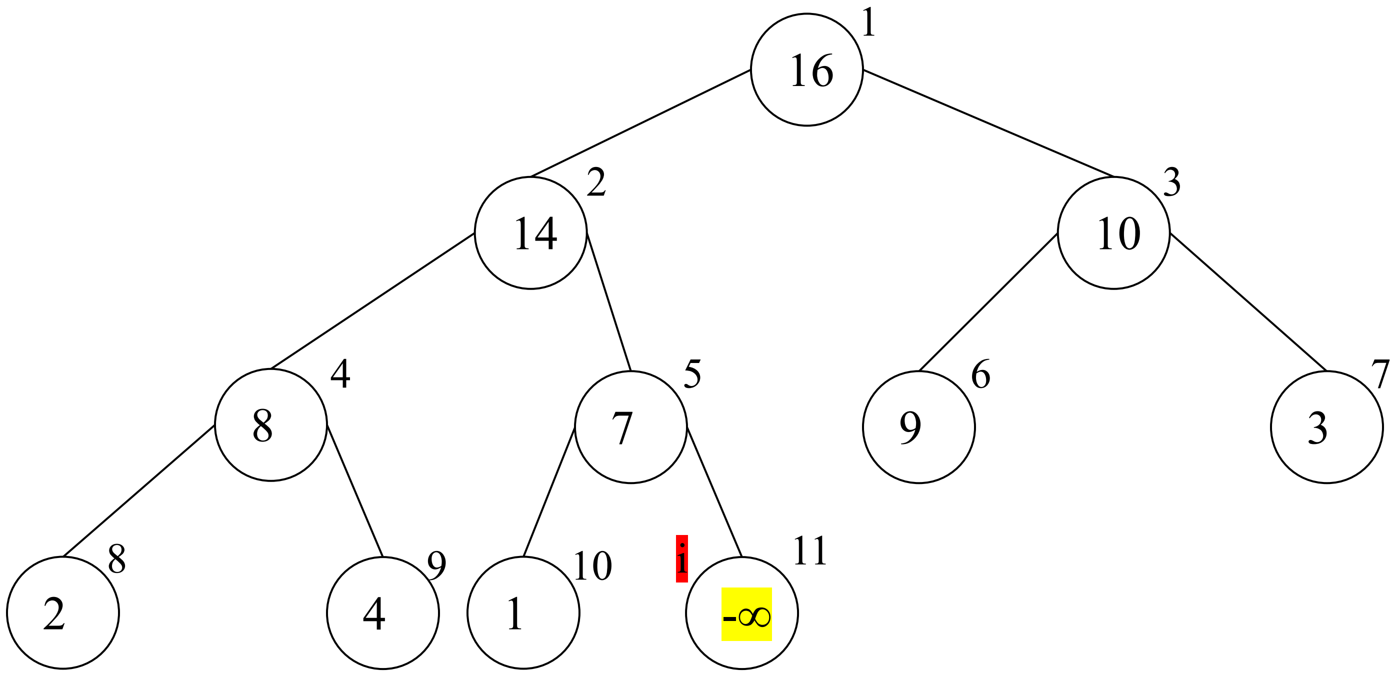
16	14	10	8	7	9	3	2	4	1		
1	2	3	4	5	6	7	8	9	10		

Example: Max-Heap-Insert(A, 13)



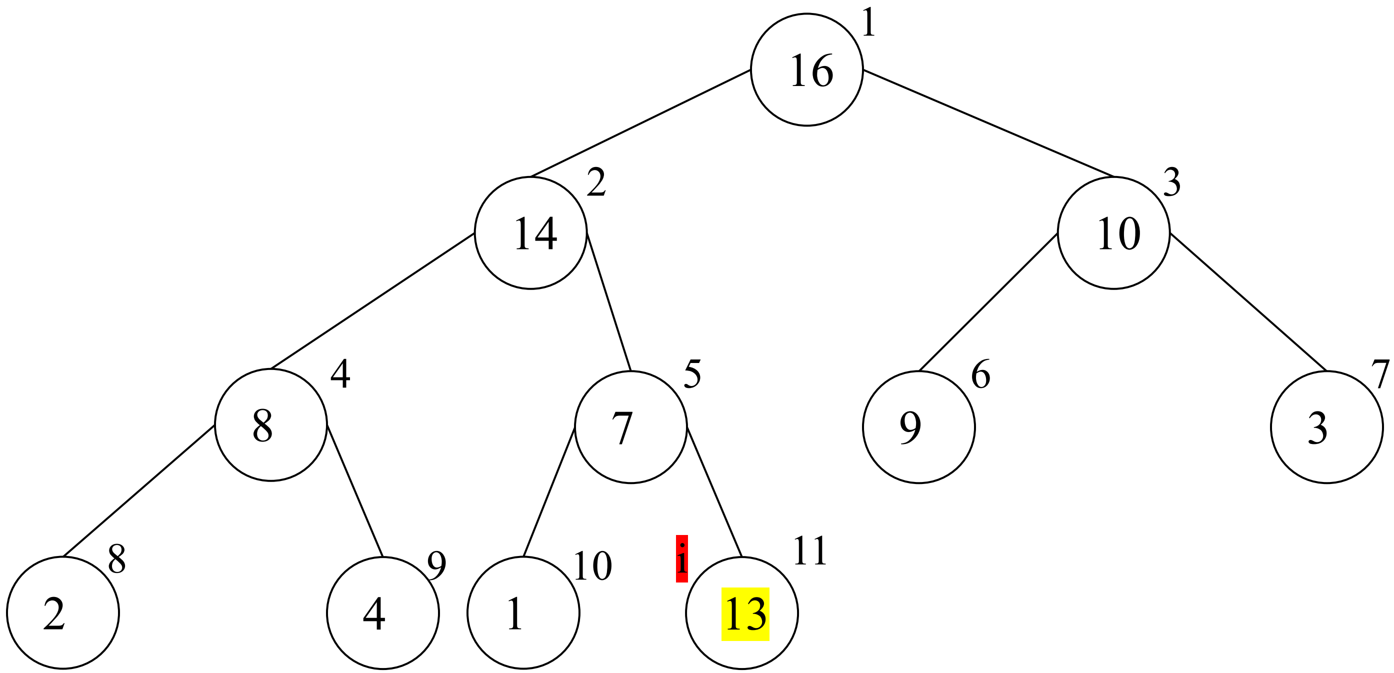
16	14	10	8	7	9	3	2	4	1		
1	2	3	4	5	6	7	8	9	10	11	

Example: Max-Heap-Insert(A, 13)



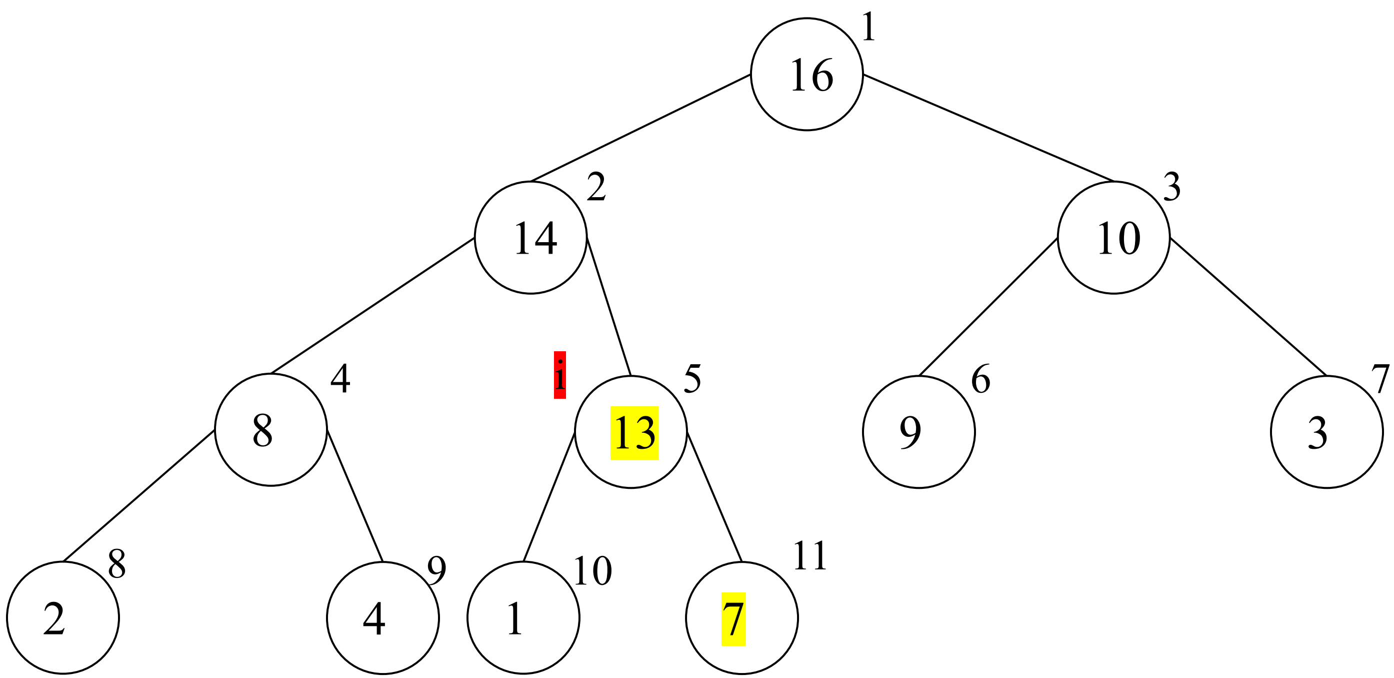
16	14	10	8	7	9	3	2	4	1	-∞	
1	2	3	4	5	6	7	8	9	10	11	

Example: Max-Heap-Insert(A, 13)



16	14	10	8	7	9	3	2	4	1	13	
1	2	3	4	5	6	7	8	9	10	11	

Example: Max-Heap-Insert(A, 13)



16	14	10	8	13	9	3	2	4	1	7	
1	2	3	4	5	6	7	8	9	10	11	

Summary

- We have studied IncreaseKey and Insertion
- We will show that both operations have time complexity $O(\log n)$, where n is heap-size.

Ira A. Fulton Schools of Engineering

Arizona State University

Max Heap and Priority Queues Part 6

Ira A. Fulton Schools of Engineering
Arizona State University

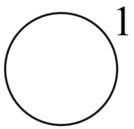
Topics of this lecture

- The Heap Data Structure
- Heapify
- Build-Heap, Heapsort
- Max and ExtractMax
- IncreaseKey and Insertion
- Analysis of Heap Operations

Analysis of Heap Operations

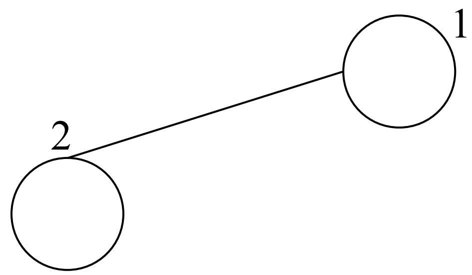
- Observation: each of the following operations has time complexity O(H), where H is the height of the tree
- We need to analyze the time complexities of Buildheap and Heapsort separately.
- What is the height of a nearly-complete binary tree with n nodes?
- What is the number of nodes in a nearly-complete binary tree with height H?

Analysis of Heap Operations



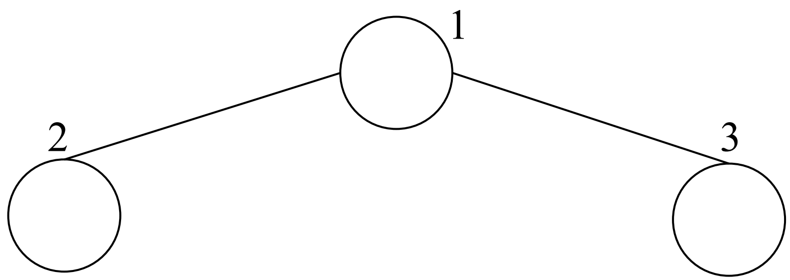
- H=0
- n=1

Analysis of Heap Operations



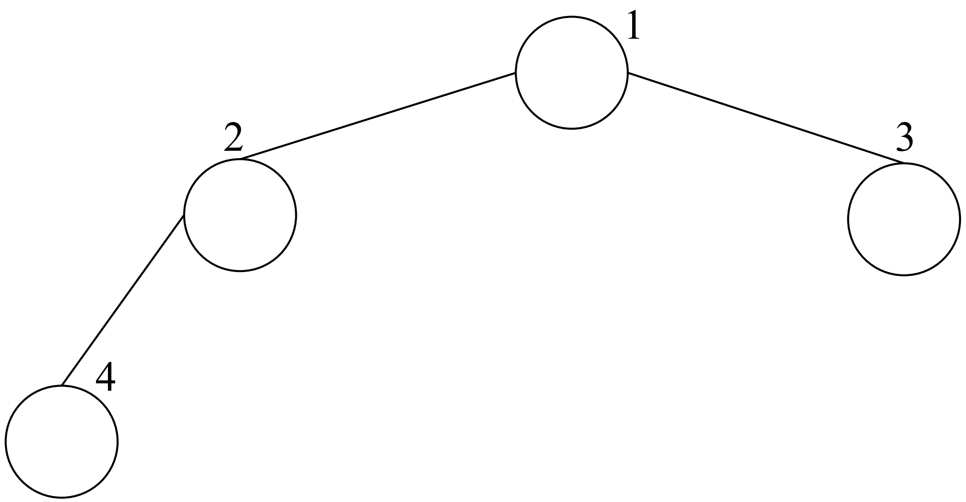
- H=1
- 2 ≤ n

Analysis of Heap Operations



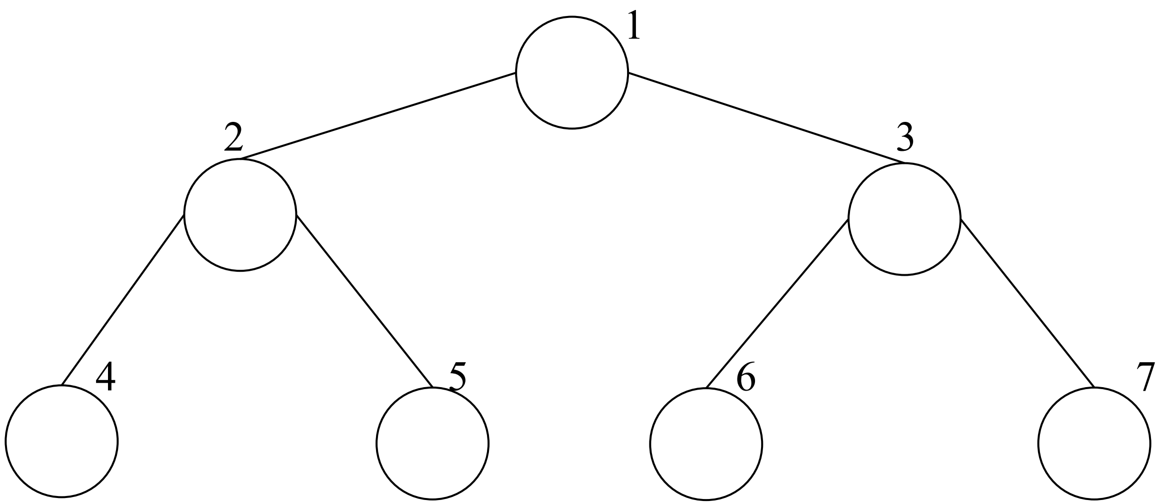
- H=1
- 2 ≤ n ≤ 3

Analysis of Heap Operations



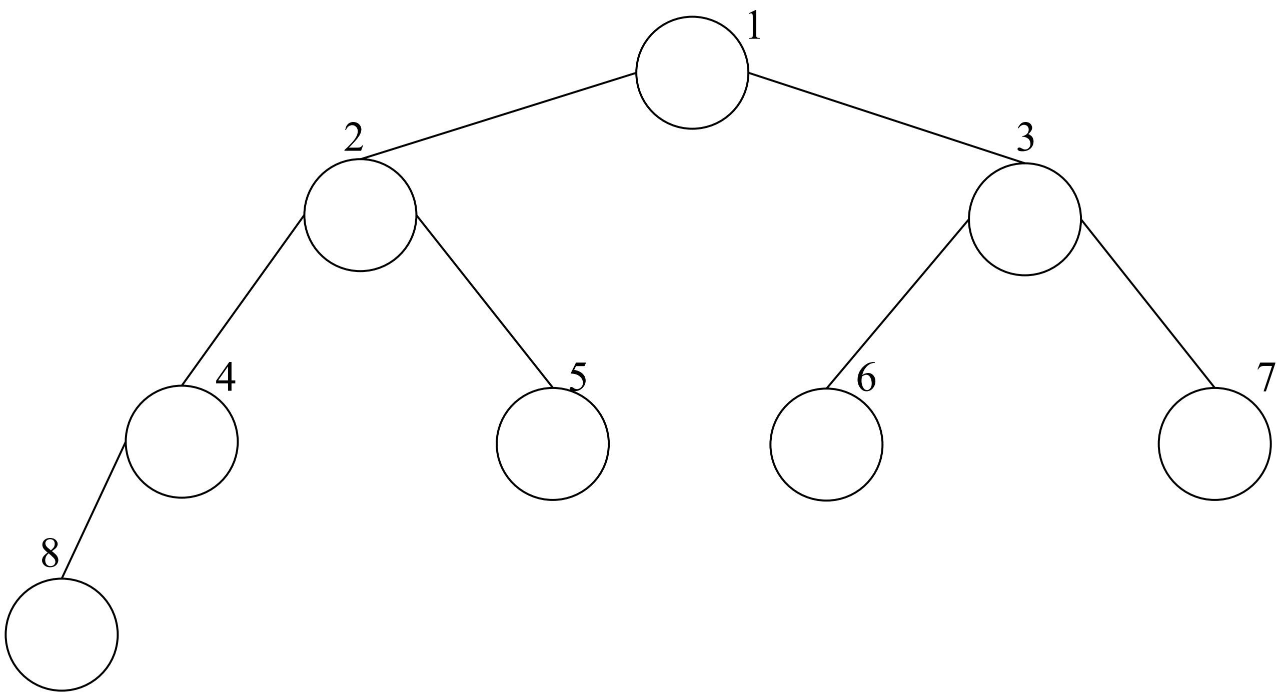
- H=2
- 4 ≤ n

Analysis of Heap Operations



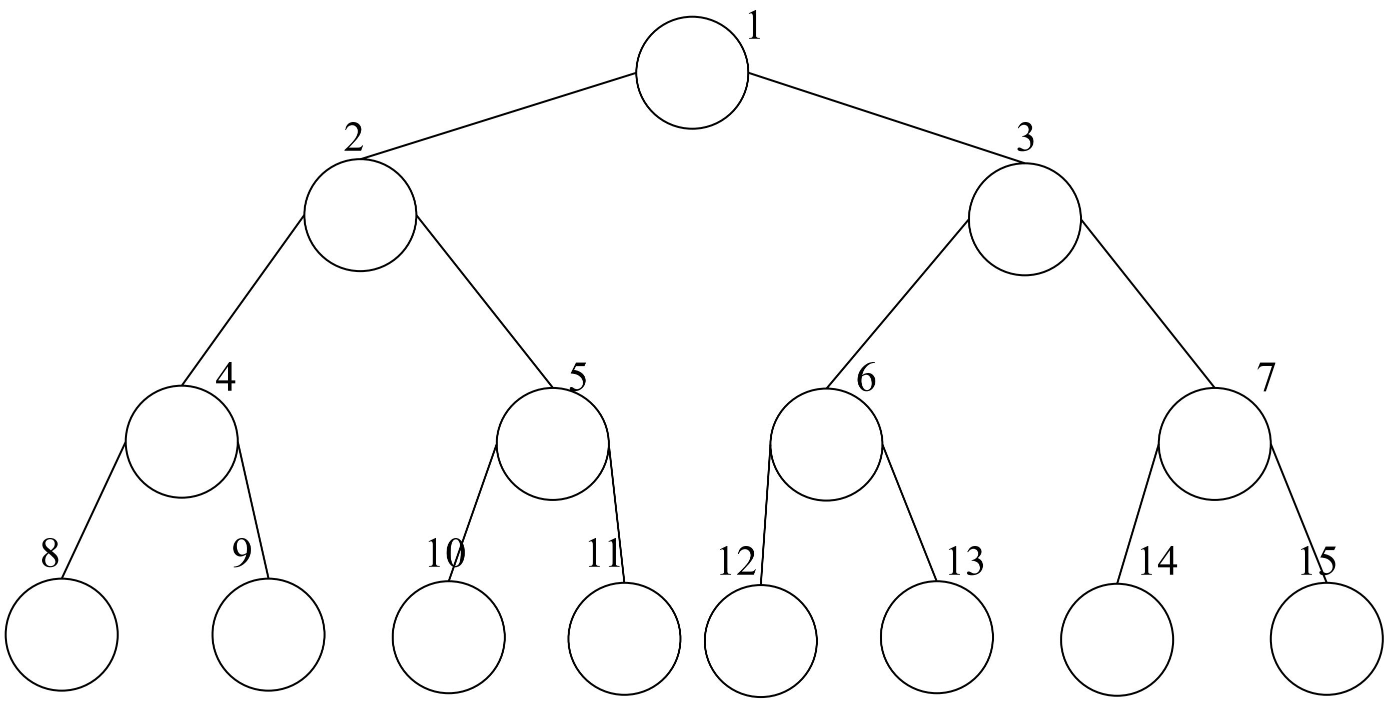
- H=2
- $4 \leq n \leq 7$

Analysis of Heap Operations



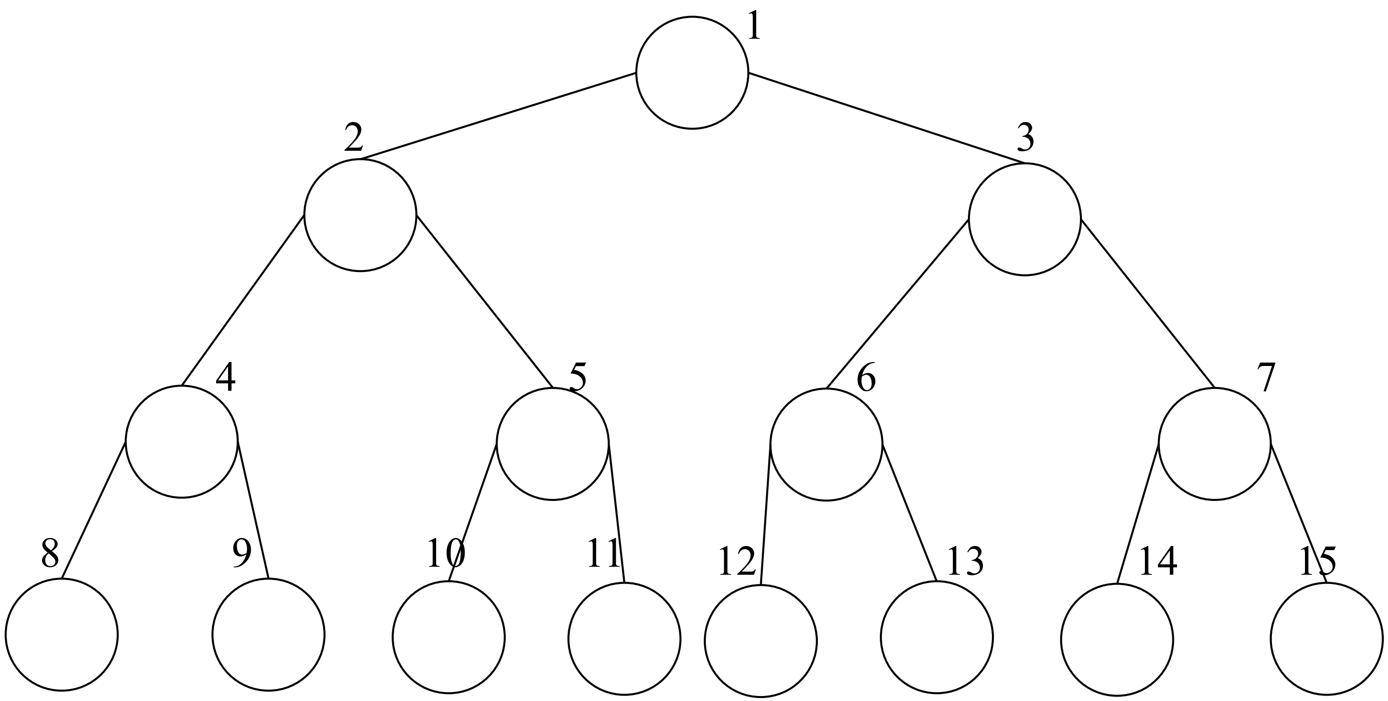
- H=3
- 8 ≤ n

Analysis of Heap Operations



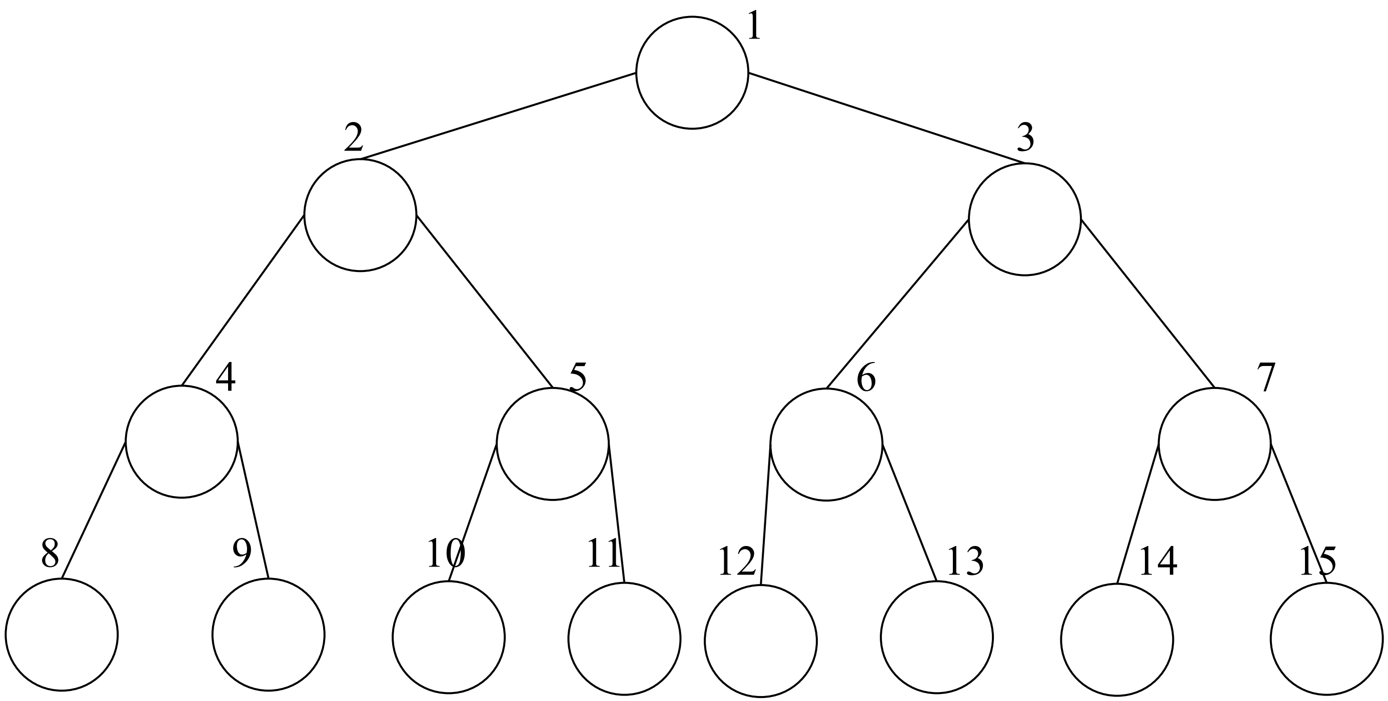
- H=3
- 8 ≤ n ≤ 15

Analysis of Heap Operations



- H=4, $16 \leq n \leq 31$
- $2^H \leq n \leq 2^{H+1}-1$, $\log_2(n+1)-1 \leq H \leq \log_2 n$
- $H \in \Theta(\log n)$

Analysis of Heap Operations



- H=4
- 16 ≤ n ≤ 31

Analysis of Heapify

The worst-case running time of heapify is $O(\log n)$

Note that heapify may stop early.

Analysis of Extract-Max

- **The worst-case running time of Extract-Max is $O(\log n)$**
- **Note that Extract-Max may stop early.**

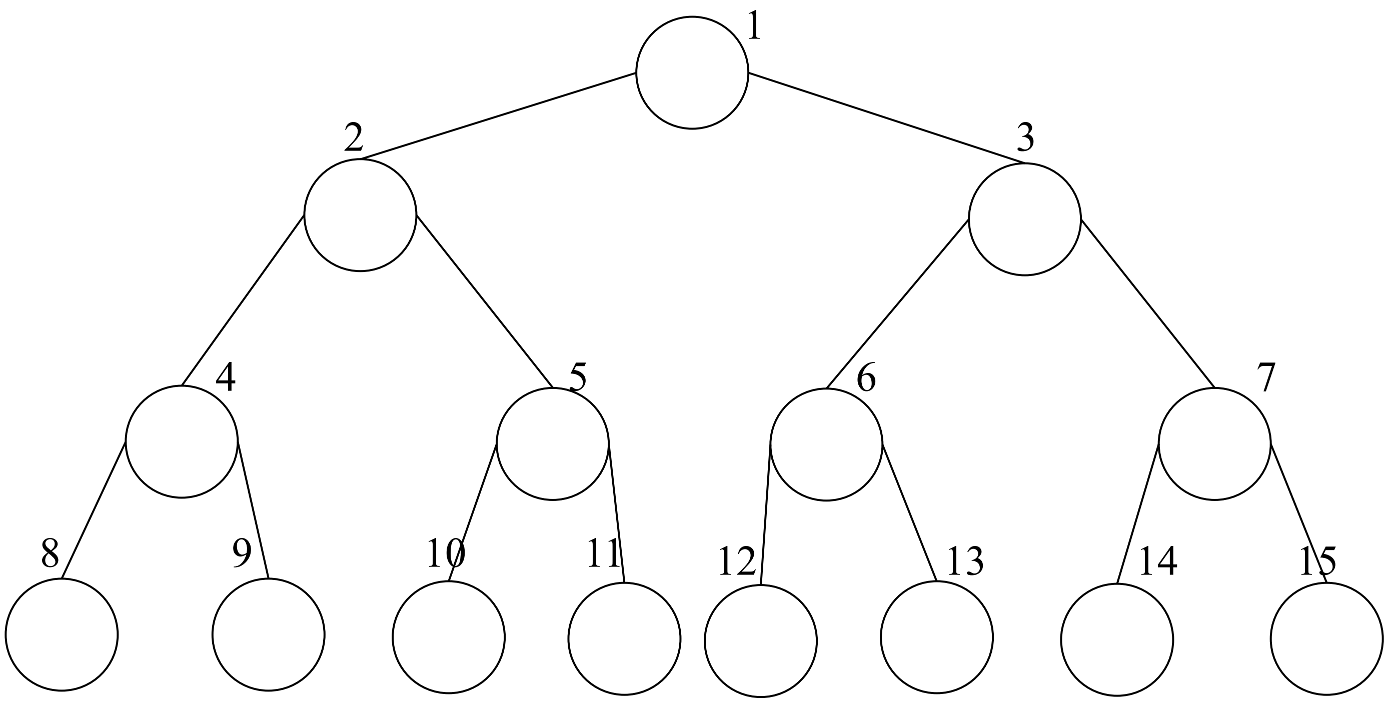
Analysis of Insertion

- The worst-case running time of Insertion is $O(\log n)$
- Note that Insertion may stop early.

Analysis of IncreaseKey

- The worst-case running time of IncreaseKey is $O(\log n)$
- Note that IncreaseKey may stop early.

Analysis of BuildHeap



- $2^H \le n \le 2^{H+1}-1$
- 2^{H-1} trees of height 1
- 2^{H-k} trees of height k, k=0, 1, 2, …, H

Analysis of BuildHeap

- $2^H \le n \le 2^{H+1}-1$
- 2^{H-1} trees of height 1
- 2^{H-k} trees of height k, k=0, 1, 2, …, H
- The worst-case time complexity of BuildHeap with n elements ($2^H \le n \le 2^{H+1}-1$) is
- $\sum_{k=1}^{H} k \times 2^{H-k} = 2^{\mathrm{H}} \sum_{k=1}^{H} \frac{k}{2^k} \le 2^{\mathrm{H}} \sum_{k=1}^{\infty} \frac{k}{2^k}$
- $\sum_{k=1}^{\infty} kx^k = \frac{x}{(1-x)^2}$ for $|x| < 1$.
- Hence $\sum_{k=1}^{H} k \times 2^{H-k} \le 2^{H+1} \le 2n$

Analysis of BuildHeap

- Proof of $\sum_{k=1}^{\infty} kx^k = \frac{\mathrm{x}}{(1-\mathrm{x})^2}$, for $|x| < 1$.
- Let $f(x) = \frac{1}{1-x} = 1 + \sum_{k=1}^{\infty} x^k$
- Since $f(x) = \frac{1}{1-x}$, $f'(x) = \frac{1}{(1-x)^2}$
- Since $f(x) = 1 + \sum_{k=1}^{\infty} x^k$, $f'(x) = \sum_{k=1}^{\infty} k \times x^{k-1}$
- Hence $\sum_{k=1}^{\infty} kx^k = x \times f'(x) = \frac{x}{(1-x)^2}$
- Therefore, the worst-case time complexity of BuildHeap is $O(n)$.

Analysis of Heapsort

- BuildHeap takes $O(n)$ time
- We perform $O(n)$ heapify operations, with a total time complexity of $O(n \log n)$.
- $O(n) + O(n \log n) = O(n \log n)$.

Priority Queue

Priority Queue is a general term of a data structure that supports operations

- Insertion
- ExtractMax (or ExtractMin)
- IncreaseKey (or DecreaseKey)

- Max-Heap is a special priority queue
- Fibonacci Heap is more efficient

Summary of Max-Heap as a Priority Queue

- BuildHeap takes $O(n)$ time
- Heapify takes $O(\log n)$ time
- Extract-Max takes $O(\log n)$ time
- Insertion takes takes $O(\log n)$ time
- IncreaseKey takes $O(\log n)$ time
- Arbitrary deletion can be done in $O(\log n)$ time

- Max-heap does not support search. However, since it is an array, search can be done in $O(n)$ time

Summary

- We have analyzed the time complexities of all heap operations.
- A key point is that the height of a nearly complete binary tree of n nodes is $O(\log n)$.
- The linear time complexity of BuildHeap is obtained by sophisticated analysis.
- Min Heap is very similar to Max Heap. In a Min Heap, the key at a node is not smaller than the key at its parent.

Ira A. Fulton Schools of
Engineering
Arizona State University