Chap6. Heap Sort

- Heaps
- Maintaining the heap property
- Building a heap
- The heapsort algorithm
- Priority queues

Heapsort

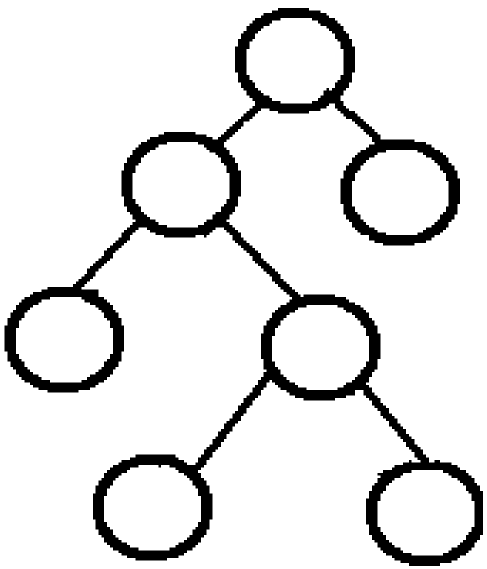
- Combines the better attributes of merge sort and insertion sort.
  - Like merge sort, but unlike insertion sort, running time is $O(n \log n)$.
  - Like insertion sort, but unlike merge sort, sorts in place.
- Introduces an algorithm design technique
  - Create data structure(*heap*) to manage information during the execution of an algorithm.
- The *heap* has other applications beside sorting.
  - Priority Queues

Full vs. Complete Binary Trees

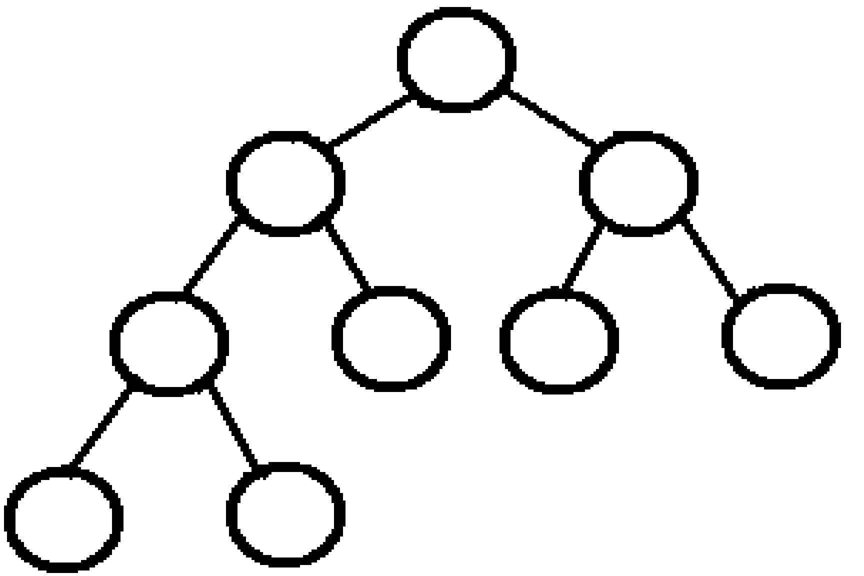
- A binary tree T is *full* if each node is either a leaf or has exactly two child nodes.

- A binary tree T with n levels is *complete* if all levels except possibly the last are completely full, and the last level has all its nodes to the left side.

Full vs. Complete Binary Trees

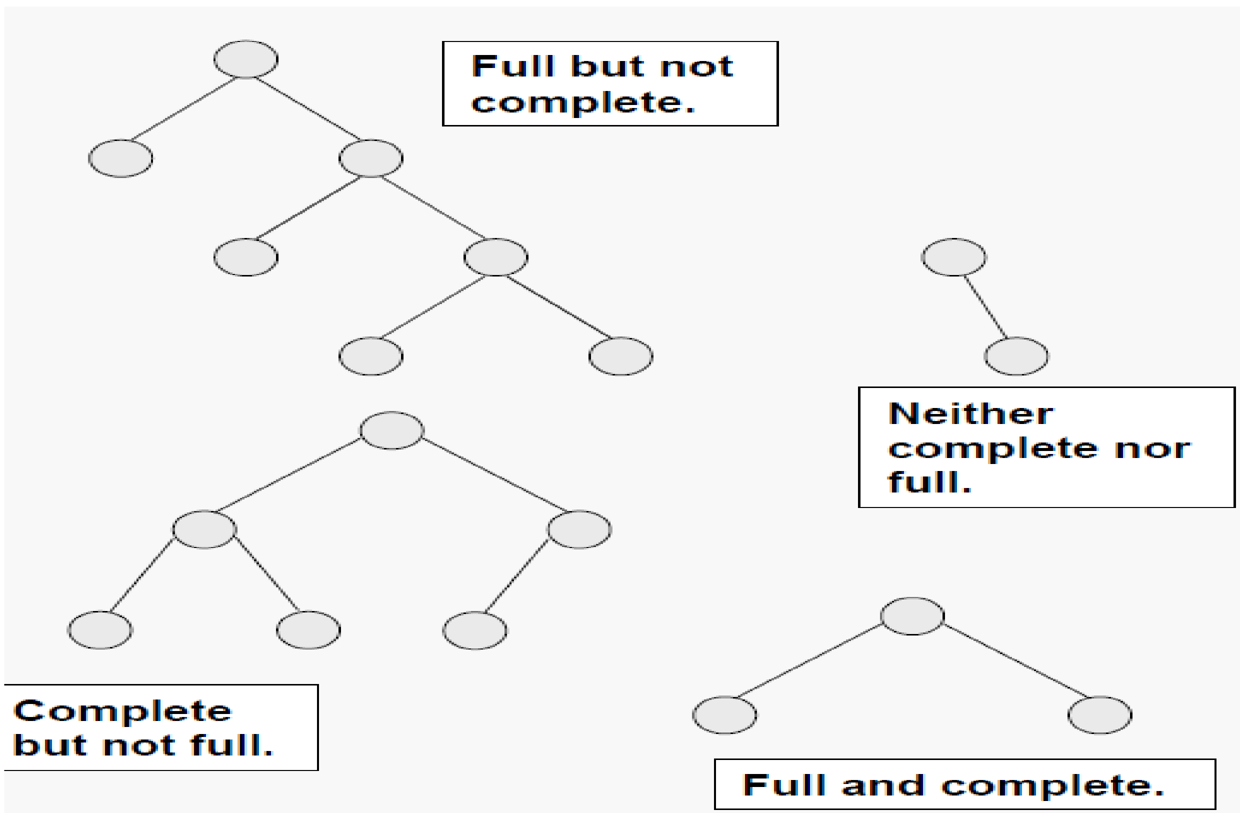


full tree



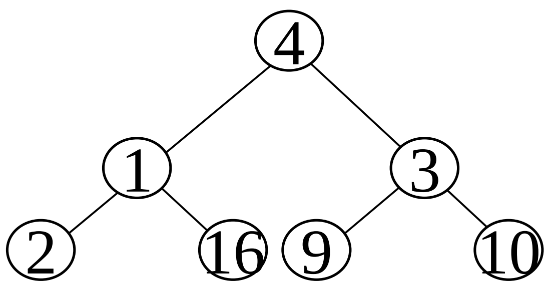
complete tree

Full vs. Complete Binary Trees



Complete vs. Nearly complete Binary Tree

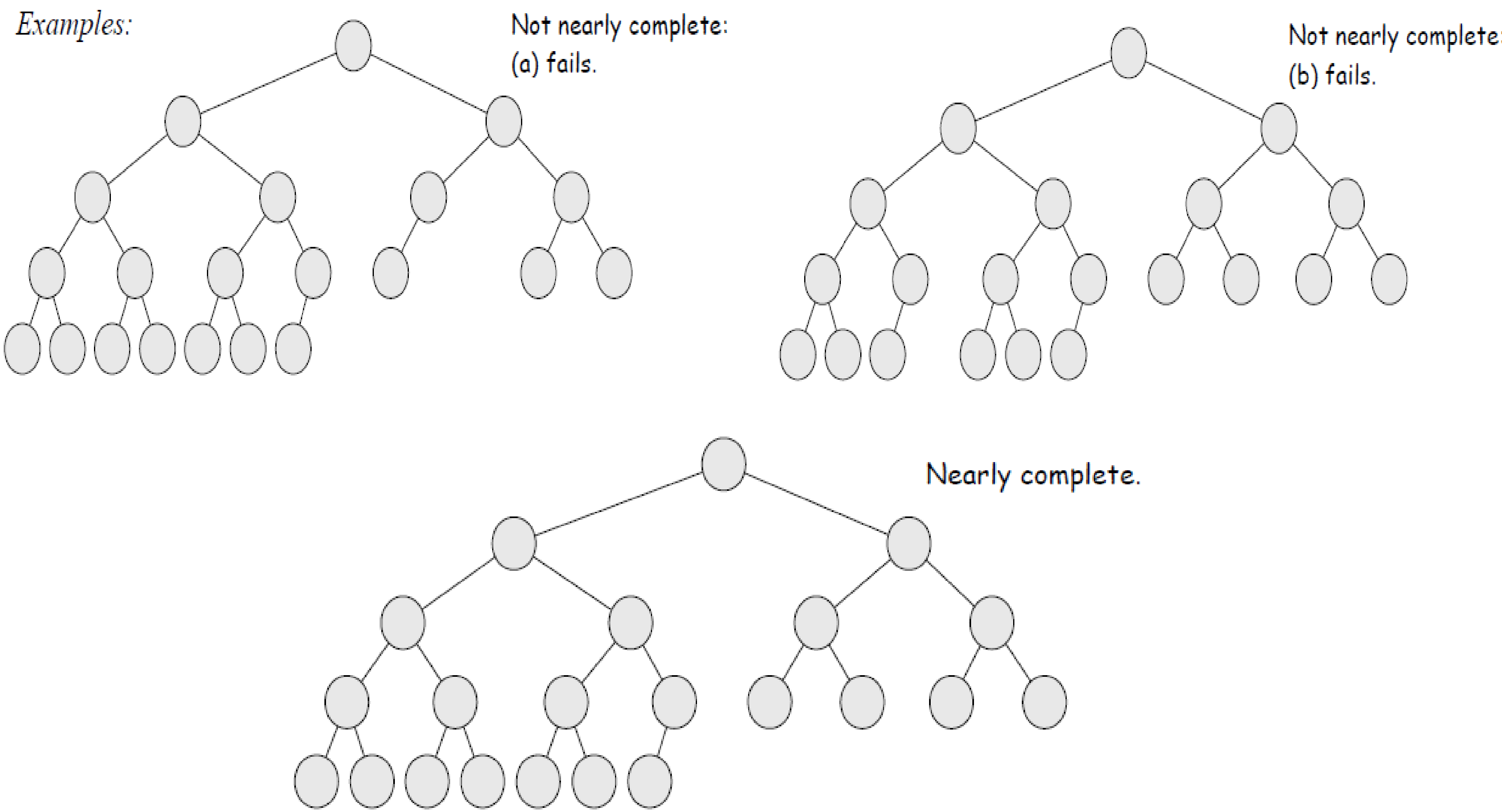
- A complete binary tree is a binary tree in which all leaves are on the same level and all internal nodes have degree 2.



Complete binary tree

- A nearly complete binary tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

Nearly Complete Binary Tree



Representation of Nearly Complete Binary Tree

- A nearly complete binary tree may be represented as an array (i.e., no pointers):
- Number the nodes, beginning with the root node and moving from level to level, left to right within a level.
- The number assigned to a node is its index in the array.

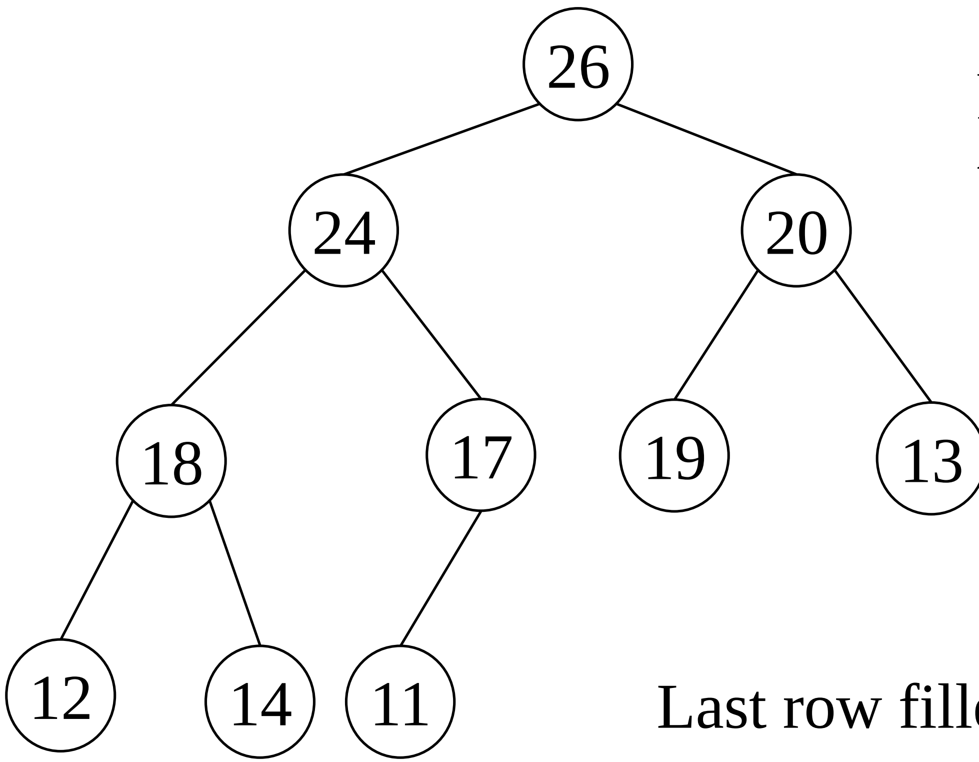
Additional Properties of Nearly Complete Binary Trees

- The root of the tree is A[1].
- If a node has index i, we can easily compute the indices of its:
  - parent $\lfloor i/2 \rfloor$
  - left child 2i
  - right child 2i + 1
- Array viewed as a nearly complete binary tree.
  - Physically linear array.
  - Logically binary tree, filled on all levels (except low est)

Heap vs. Array

1	2	3	4	5	6	7	8	9	10
26	24	20	18	17	19	13	12	14	11

heap as an array.



heap as a binary tree.

Last row filled from left to right.

Heap

- A binary tree with n nodes and of height *h* is nearly complete iff its nodes correspond to the nodes which are numbered 1 to *n* in the complete binary tree of height *h*.
- A heap is a *nearly complete binary tree* that satisfies the heap property:
  max-heap: For every node *i* other than the root*:*
  $$A[\text{Parent}(i)] \geq A[i]$$
  min-heap: For every node i other than the root:
  $$A[\text{Parent}(i)] \leq A[i]$$

Max-Heap

- A max-heap is a *nearly complete binary tree that* satisfies the heap property:
  For every node i other than the root,
  $$A[\text{PARENT}(i)] \geq A[i]$$
- What does this mean?
  - the value of a node is at most the value of its parent
  - <u>the largest element in the heap is stored in the root</u>

Height

- *Height of a node in a tree*: the number of edges on the longest simple downward path from the node to a leaf.
- *Height of a tree*: the height of the root.
- Height of a heap: $\lfloor \log n \rfloor$
  - Basic operations on a heap run in $O(\log n)$ time

Heap Characteristics

- *Height* $= \lfloor \log n \rfloor$
- \# of *leaves* $= \lceil n/2 \rceil$
- \# of nodes of height h $\leq \lceil n/2^{h+1} \rceil$

Heaps have 5 basic procedures

- HEAPIFY: maintains the heap property
- BUILD-HEAP: builds a heap from an unordered array
- HEAPSORT: sorts an array in place
- EXTRACT-MAX: selects max element
- INSERT: inserts a new element

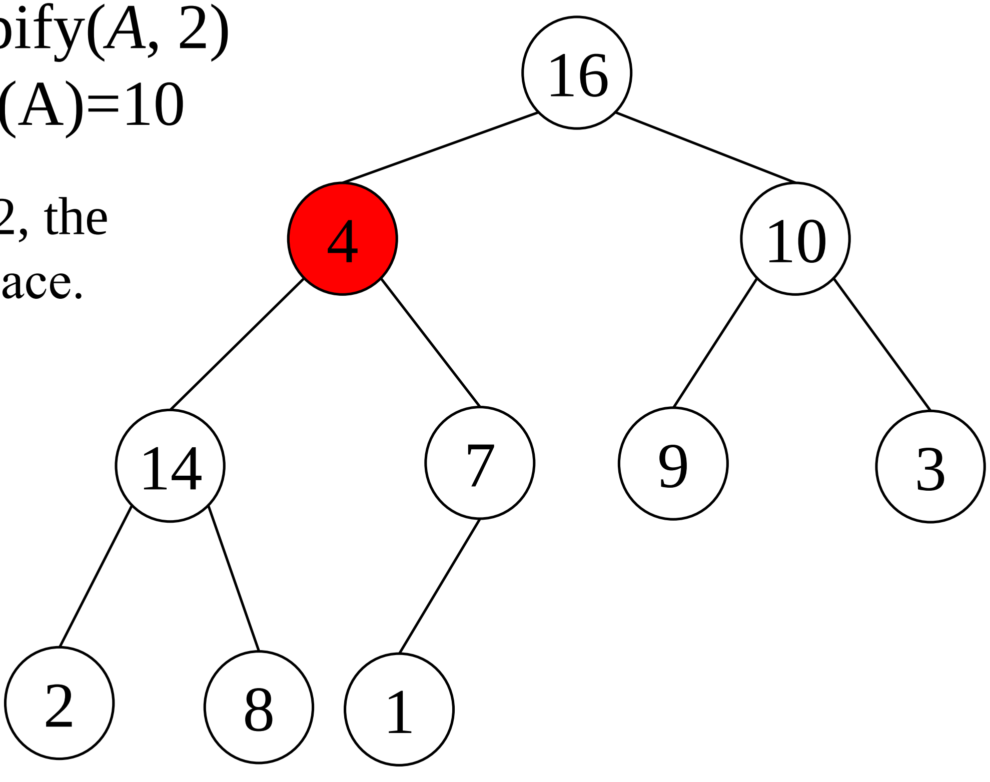
MAXHEAPIFY(A,i)

- Goal is to put the i^{th} element in the correct place in a portion of the array that “almost” has the heap property.
- The only element with index of *i* or greater that is out of place is A[*i*].
- Assume that left and right subtrees of A[*i*] have the heap property.
- “Sift” A[*i*] down to the right position.

MaxHeapify – Example

MaxHeapify(A, 2)
Heapsize(A)=10

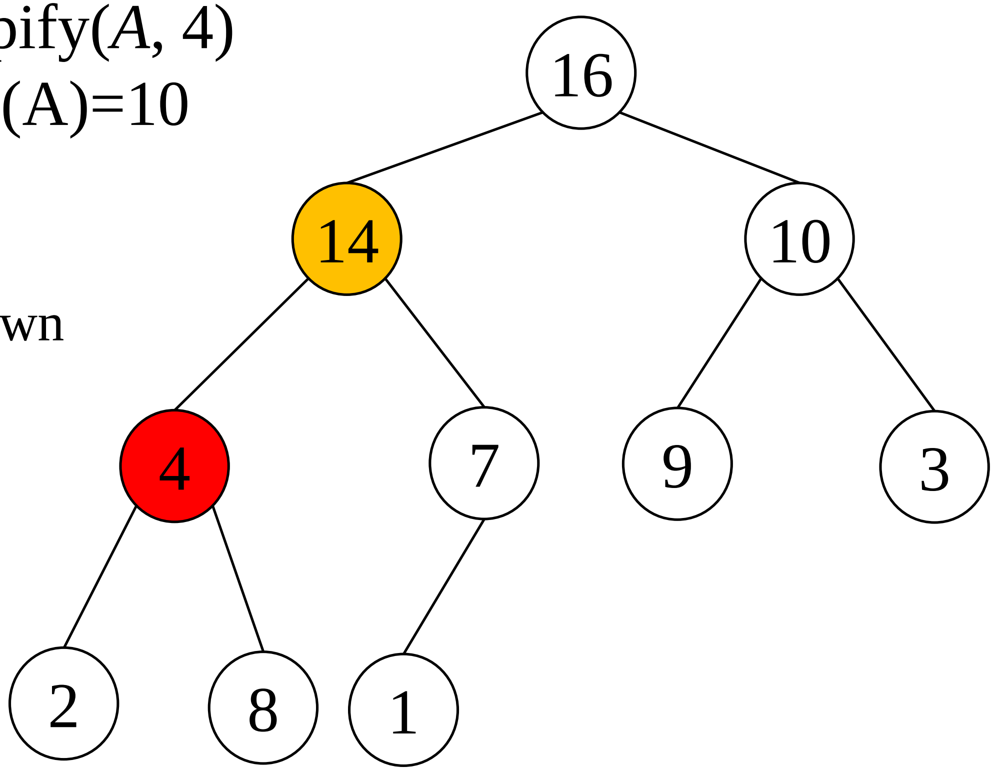
Array element 2, the "4", is out of place.



MaxHeapify – Example

MaxHeapify(A, 4)
Heapsize(A)=10

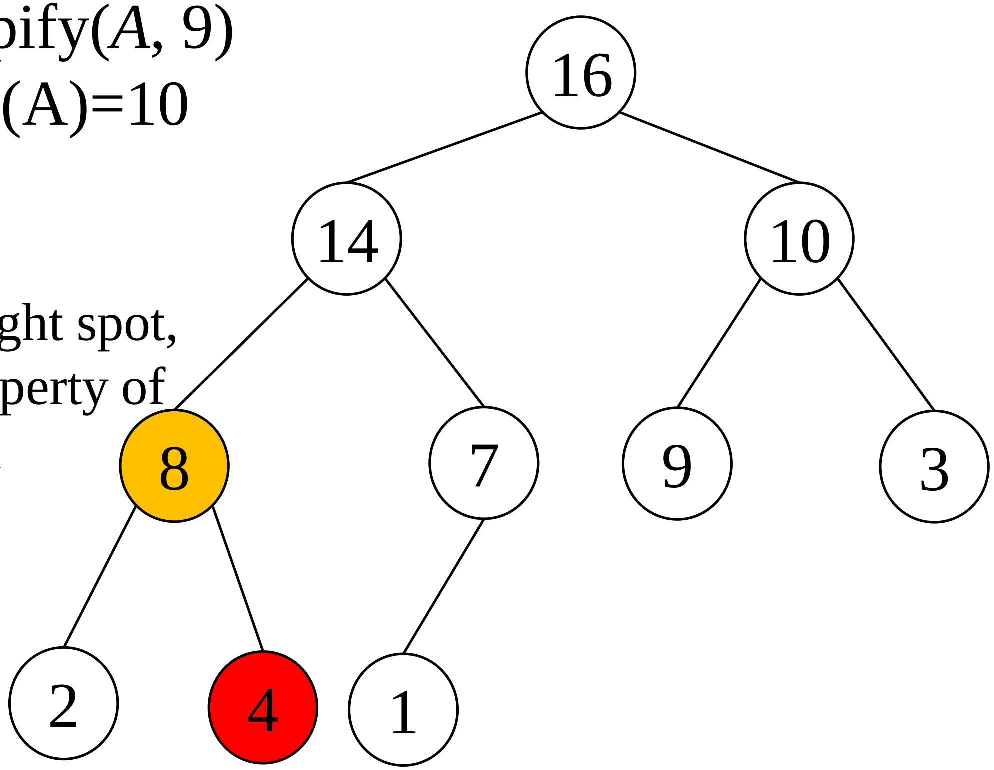
Moving the 4 down



MaxHeapify – Example

MaxHeapify(A, 9)
Heapsize(A)=10

The 4 is in the right spot, and the heap property of the tree has been restored.



MAX-HEAPIFY

MaxHeapify(A, i)

$l \leftarrow$ left(i)
$r \leftarrow$ right(i)
if $l \leq$ *heap-size*[A] and $A[l] > A[i]$
    then *largest* $\leftarrow l$
    else *largest* $\leftarrow i$
if $r \leq$ *heap-size*[A] **and** $A[r] > A[largest]$
    then *largest* $\leftarrow r$
if *largest* $\neq i$
    then exchange $A[i] \leftrightarrow A[largest]$
            MaxHeapify(A, *largest*)

Assumption:
Left(i) and Right(i) are max-heaps.

Running time of MaxHeapify

<u>*MaxHeapify*(*A*, *i*)</u>

1. $l \leftarrow$ left(i)
2. $r \leftarrow$ right(i)
3. **if** $l \leq$ *heap-size*[A] and $A[l] > A[i]$
4.   **then** *largest* $\leftarrow l$
5.   **else** *largest* $\leftarrow i$
6. **if** $r \leq$ *heap-size*[A] **and** $A[r] > A[$*largest*$]$
7.   **then** *largest* $\leftarrow r$
8. **if** *largest* $\neq i$
9.   **then** exchange $A[i] \leftrightarrow A[$*largest*$]$
10.   *MaxHeapify*(*A*, *largest*)

Line1-Line9:
Time to fix node *i* and its children = **$\Theta(1)$**

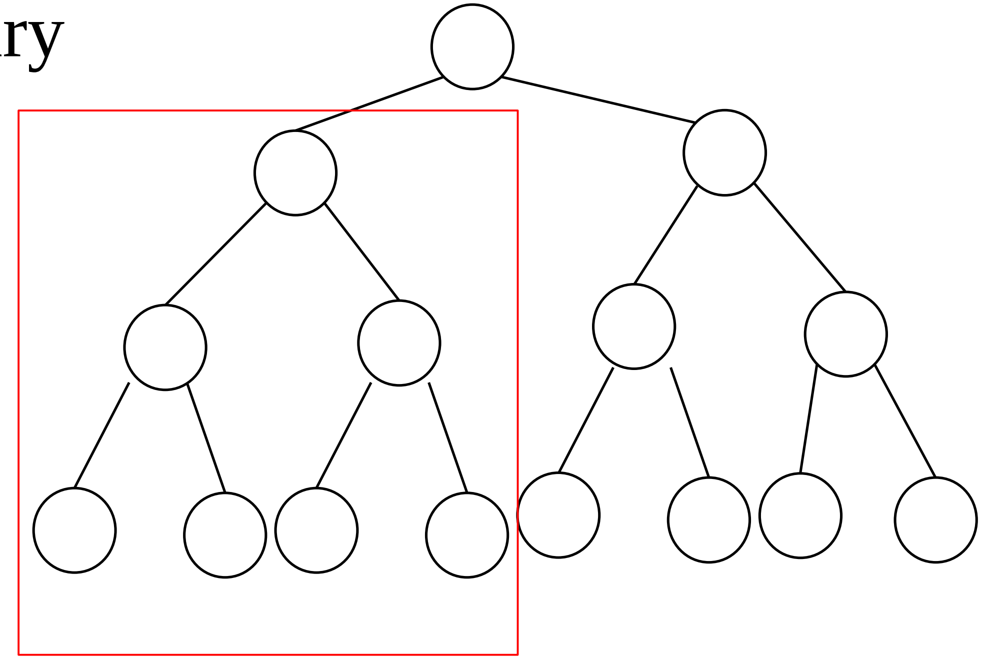
+

?

Running time of MaxHeapify

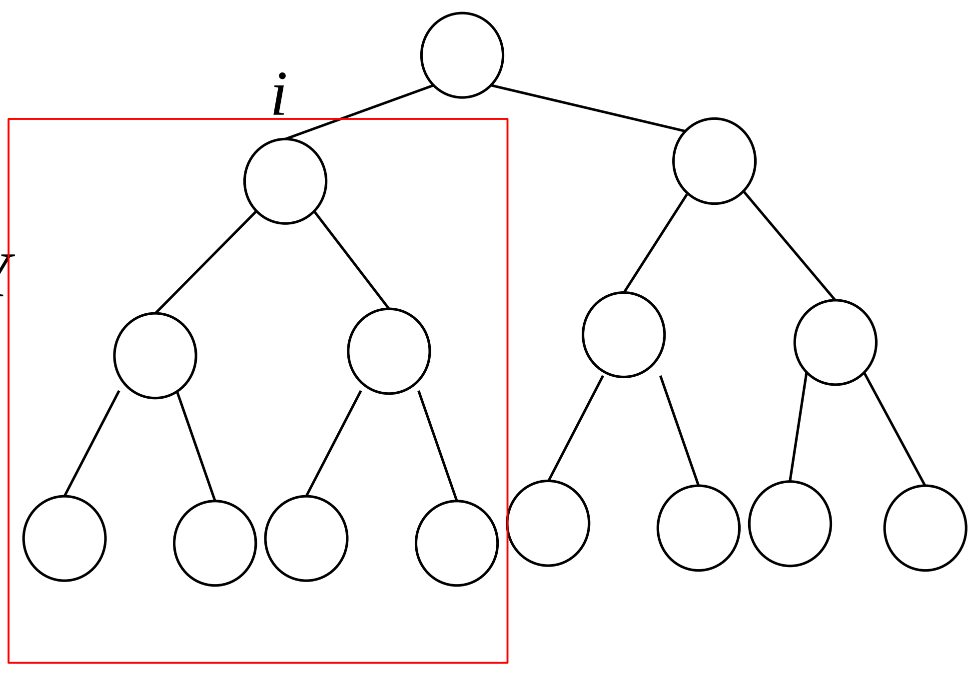
- **How many nodes might be involved?**

In the case of a full binary tree, <u>about half </u>of the tree might be involved.



Running time of MaxHeapify

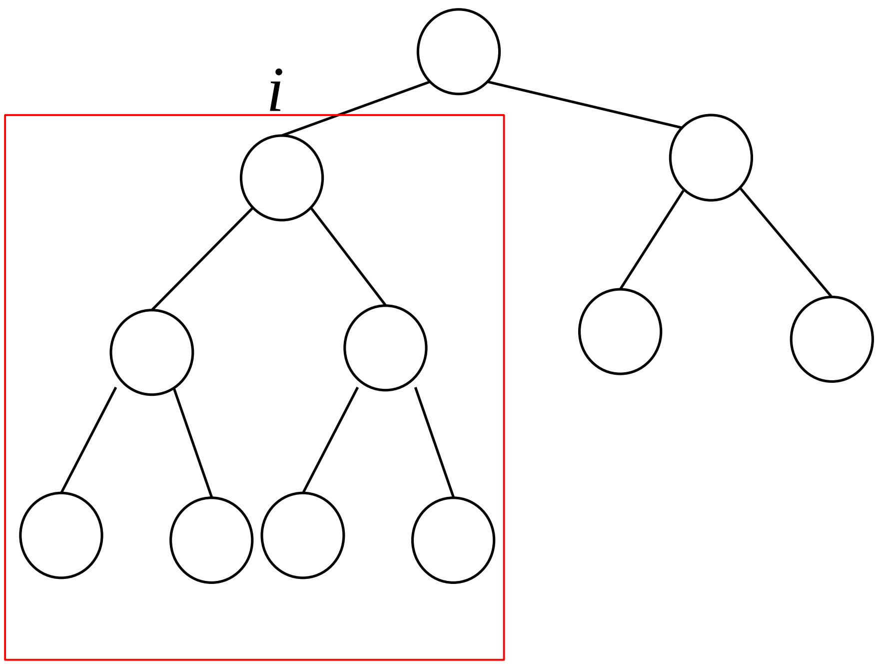
In a complete binary tree with 15 nodes, 8 of those nodes are leaves at the bottom level.



If we perform MAXHEAPIFY on node *i*, 7 of the 15 nodes will be involved – about ½ of the nodes.

Running time of MaxHeapify

- What is the worst case?
  When the last row of the tree is half full.



Here 7 out of 11 nodes are involved.
In general, $\leq$ 2/3 of the tree might be involved in the worst case.

Running time of MaxHeapify

- Remember that, in a complete binary tree, <u>more than half of the nodes in the entire tree</u> are the leaf nodes on the bottom level of the tree.
- But the only nodes involved in MAX-HEAPIFY are the descendants of A[i], which must be in A[i]'s half of the tree.
- So worst case is when the last row of the tree is half full on the left side and A[i] is their ancestor.

Running time of MaxHeapify

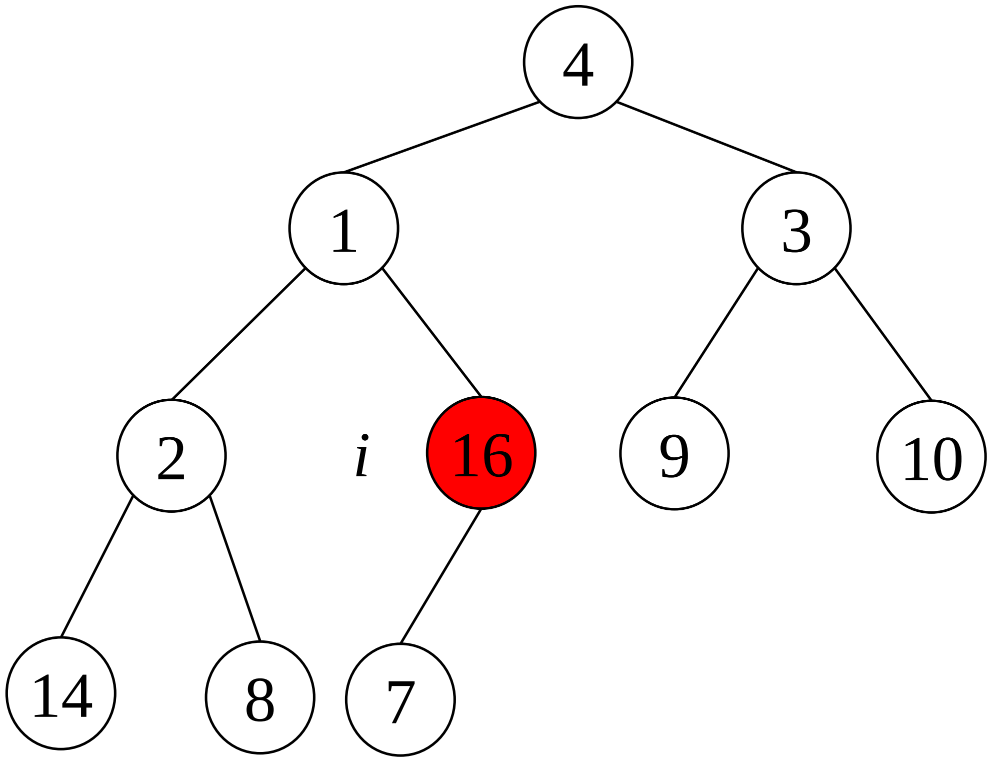
- The subtrees of the children of our current node have size at most $2n/3$.
- The running time of MAX_HEAPIFY can be described by the recurrence:
  $T(n) \le T(2n/3) + \Theta(1)$
- This is Case 2 by the master method, so:
  $T(n) = O(\log n)$
- We could also describe the running time of MAX-HEAPIFY for a node of height h as $O(h)$. *(This is useful only if we know* the height of a specific node.)

BUILD-MAX-HEAP

- Use MAX-HEAPIFY in a bottom-up manner to convert an array A[1..n] into a heap.
- Each leaf is initially a one-element heap.
  Elements A[$\lfloor n/2 \rfloor$ + 1..n] are leaves.
- MAX-HEAPIFY is called on all interior nodes.

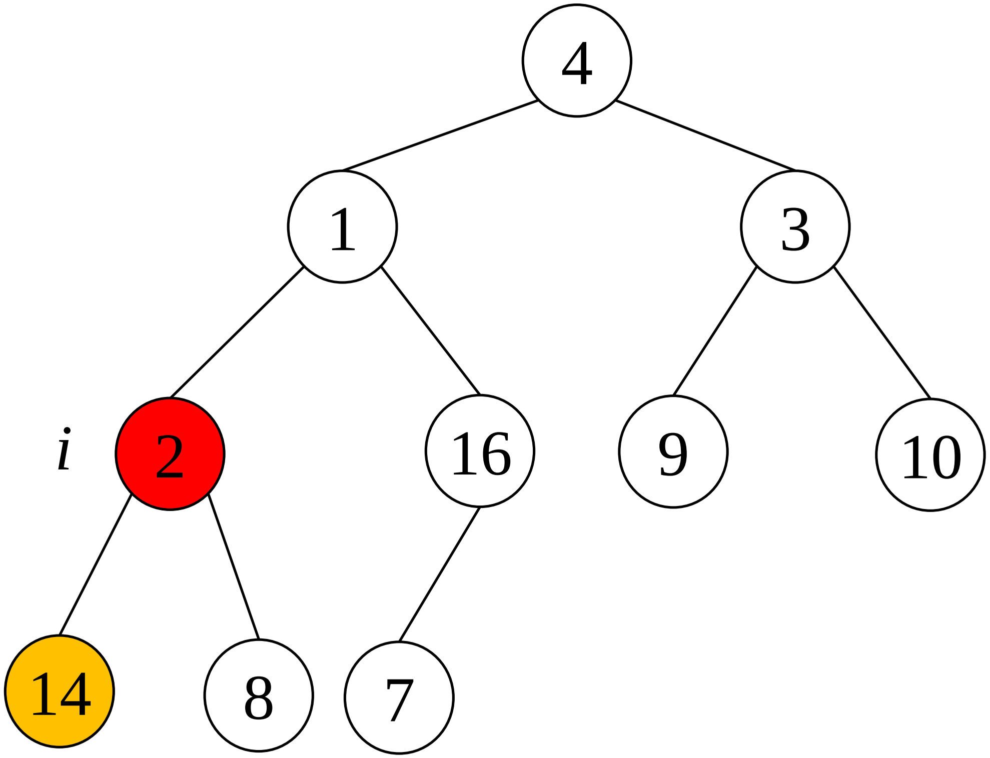
BUILD-MAX-HEAP

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

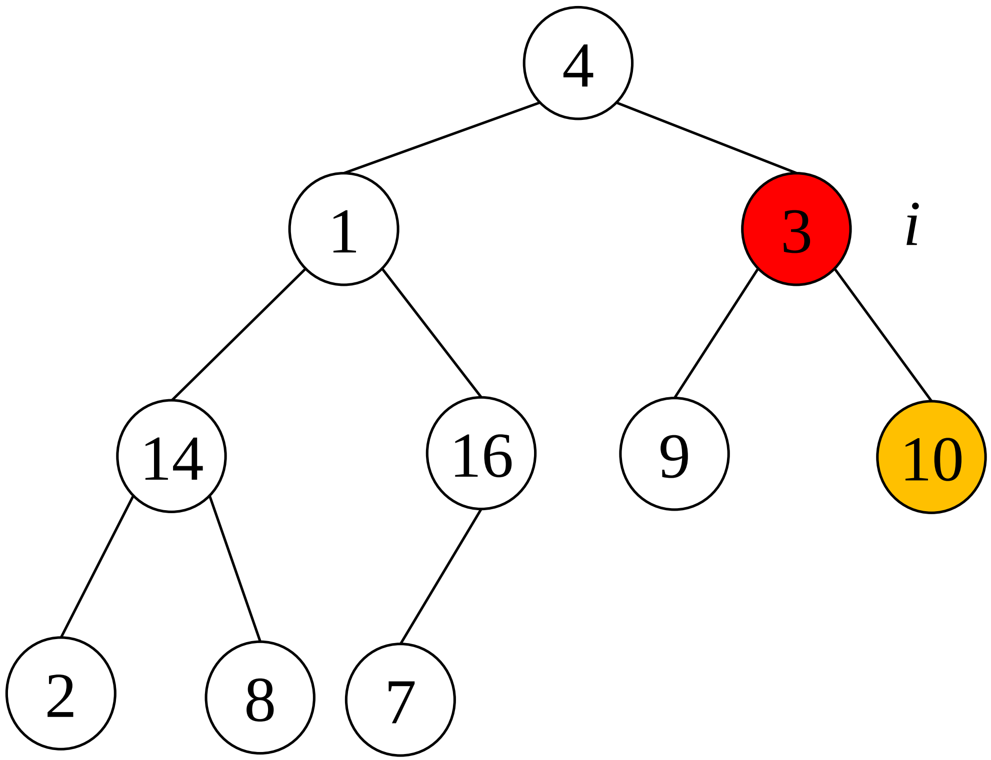


length(A) = 10
floor(length(A)/2) = 5
process from 5 to 1

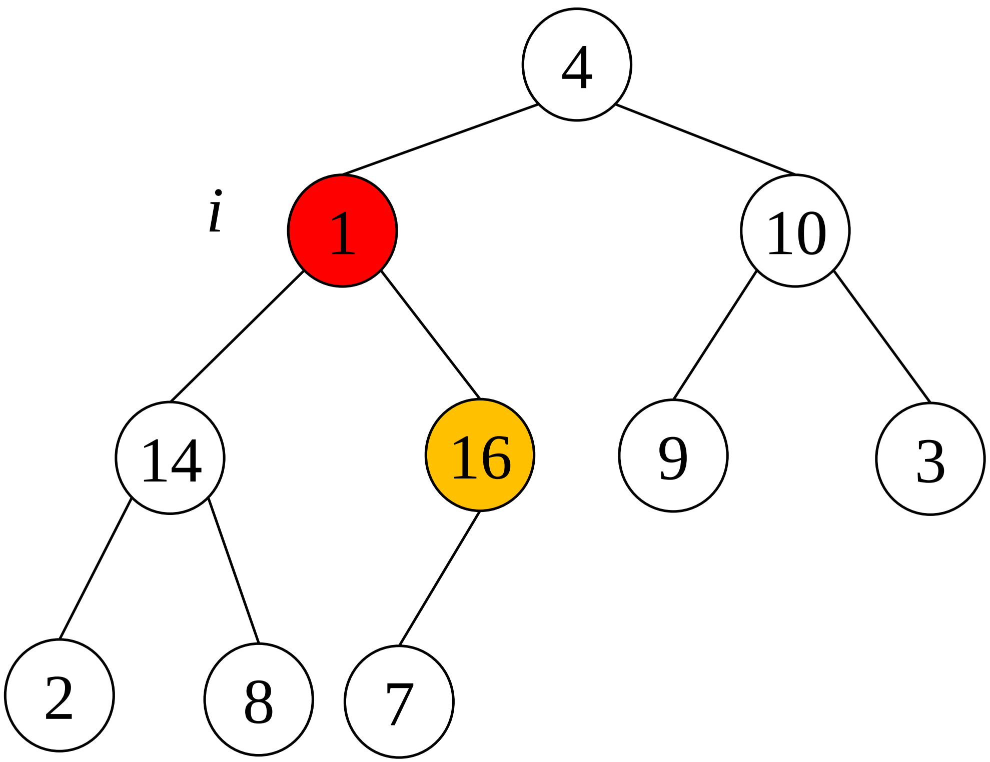
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7



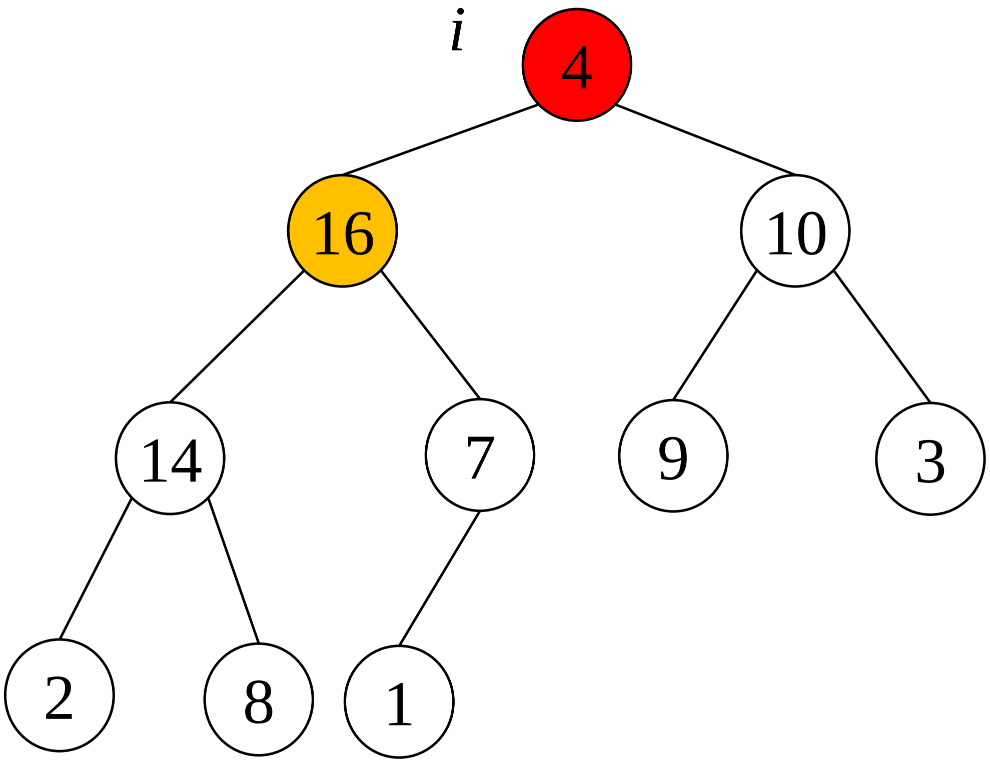
1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7



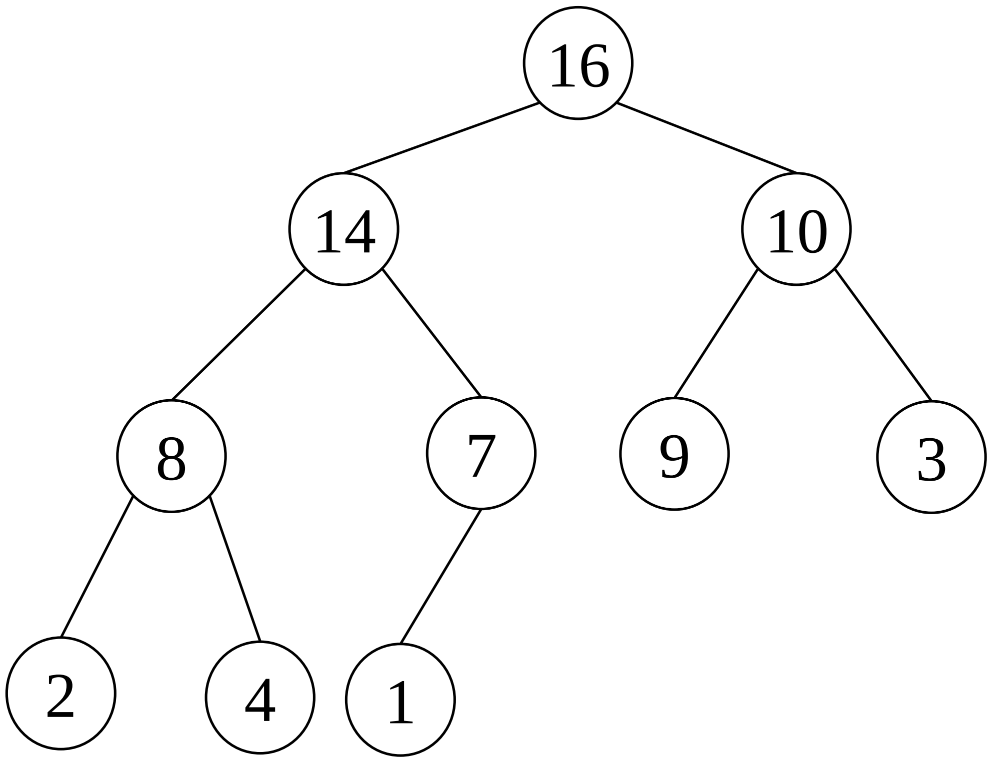
1	2	3	4	5	6	7	8	9	10
4	1	3	14	16	9	10	2	8	7



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



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



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

Running Time of BUILD-MAX-HEAP

- Simple upper bound:
  - each call to MAX-HEAPIFY costs O(log n)
  - O(n) such calls
  - running time at most O($nlogn$)
- Previous bound is not tight:
  - lots of the elements are leaves
  - most elements are near leaves (small height)

Tighter Bound for BUILD-MAX-HEAP

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

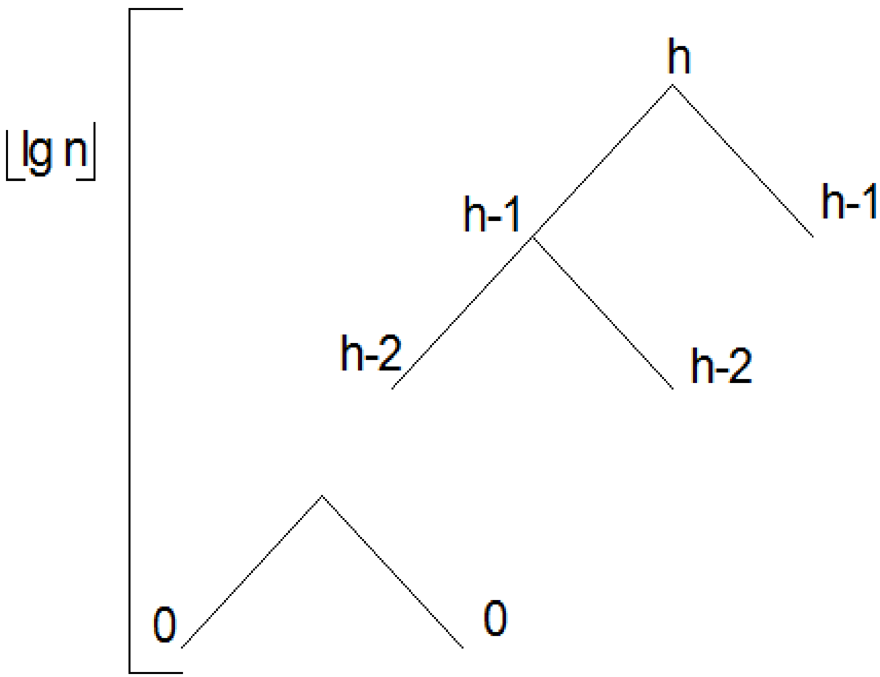
$$\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}$$
$$\le \sum_{h=0}^{\infty} \frac{h}{2^h}$$
$$= \frac{1/2}{(1-1/2)^2}$$
$$= 2$$

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

Thus the running time is bounded by O(n)

Therefore, we can build a heap from an unordered array in linear time

Recursion tree method for calculating Build-Max-Heap cost



Height	Nodes	Cost
h	2^0	$h*2^0$
h-1	2^1	$(h-1)*2^1$
h-2	2^2	$(h-2)*2^2$
0	2^h	$0*2^h$

$$\sum_{h=0}^{\lfloor \lg n\rfloor} h*2^{\lfloor \lg n\rfloor-h} = \sum_{h=0}^{\lfloor \lg n\rfloor} h*\frac{2^{\lfloor \lg n\rfloor}}{2^h} \le \sum_{h=0}^{\lfloor \lg n\rfloor} h*\frac{2^{\lg n}}{2^h} = \sum_{h=0}^{\lfloor \lg n\rfloor} h*\frac{n^{\lg 2}}{2^h}$$

$$= \sum_{h=0}^{\lfloor \lg n\rfloor} h*\frac{n}{2^h} = n\sum_{h=0}^{\lfloor \lg n\rfloor}\frac{h}{2^h} \le n\sum_{h=0}^{\infty}\frac{h}{2^h} = 2n = O(n)$$

Heap Sort

- First build a heap.
- Then successively remove the biggest element from the heap and move it to the first position in the sorted array.
- The element currently in that position is then placed at the top of the heap and sifted to the proper position.

Heap Sort

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2 do
        exchange A[1] ↔ A[i]
        heap-size[A] ← heap-size[A] – 1
        MAX-HEAPIFY(A, 1)
```

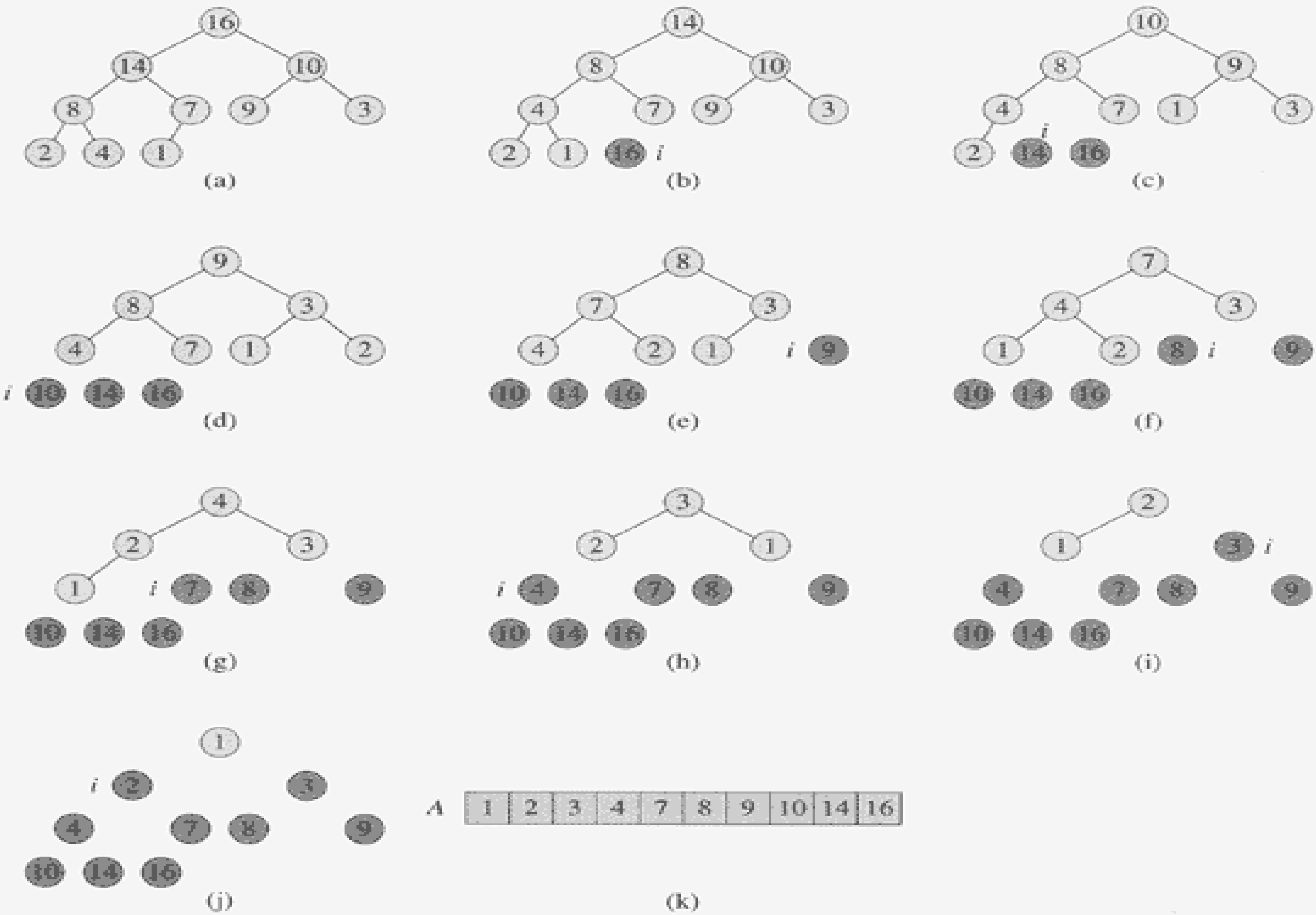


Figure 6.4 The operation of HEAPSORT. **(a)** The max-heap data structure just after it has been built by BUILD-MAX-HEAP. **(b)–(j)** The max-heap just after each call of MAX-HEAPIFY in line 5. The value of i at that time is shown. Only lightly shaded nodes remain in the heap. **(k)** The resulting sorted array A.

Running time of Heapsort

HEAPSORT(A)
1 BUILD-MAX-HEAP(A) O(n)
2 for i ← length[A] downto 2 do O(n-1)
3 exchange A[1] ↔ A[i] O(1)
4 heap-size[A] ← heap-size[A] – 1 O(1)
5 MAX-HEAPIFY(A, 1) O(log n)

- Total time is:
  O(n) + O(n-1) * [O(1) + O(1) + O(log n)]
  which is approximately O(*n) + O(nlogn)*
  or just O(*nlogn)*

Running time of Heapsort

- BUILD-MAX-HEAP takes O(*n*).
- We have a loop. Each of the *n-1* calls to MAX-HEAPIFY takes O(log *n*) time.
- Total time is O(*nlogn*).

Space requirements of Heapsort

- Heapsort uses an array as its data structure.
- Heapsort sorts “in place”.
- Any extra storage needed?
- Only a negligible amount – one extra storage location is needed as temporary storage when swapping two array elements

Priority Queues

- Popular & important application of heaps.
- Max and min priority queues.
- Maintains a *dynamic* set *S* of elements.
- Each set element has a *key* – an associated value.
- Goal is to support insertion and extraction efficient ly
- Applications:
  - Ready list of processes in operating systems by their priorities (the list is highly dynamic)
  - In event-driven simulators to maintain the list of events to be simulated in order of their time of occurrence.

Basic Operations

- Operations on a max-priority queue:
  - Insert(S, x) : inserts the element x into the set S
    $S \leftarrow S \cup \{x\}$.
  - Maximum(S) : returns the element of S with the largest key.
  - 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.

- Min-priority queue supports Insert, Minimum, Extract-Min, and Decrease-Key.
- Heap gives a good compromise between fast insertion but slow extraction and vice versa.

HEAP-MAXIMUM

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

- Returns the item at the top of the heap
- Runs in $\Theta(1)$ time

Heap-Extract-Max(*A*)

<u>*Heap-Extract-Max*(*A*)</u>

1. if *heap-size*[*A*] < 1
2. then error "heap underflow"
3. *max* $\leftarrow$ *A*[1]
4. *A*[1] $\leftarrow$ *A*[*heap-size*[*A*]]
5. *heap-size*[*A*] $\leftarrow$ *heap-size*[*A*] - 1
6. MaxHeapify(*A*, 1)
7. return max

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

Heap-Insert(A, key)

```
Heap-Insert(A, key)
1.  heap-size[A] ← heap-size[A] + 1
2.   i ← heap-size[A]
4.  while i > 1 and A[Parent(i)] < key
5.        do A[i] ← A[Parent(i)]
6.              i ← Parent(i)
7.  A[i] ← key
```

Running time is $O(\log n)$

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

Heap-Increase-Key(*A*, *i*, *key*)

Heap-Increase-Key(A, i, key)

If $key < A[i]$
  then error "new key is smaller than the current key"
$A[i] \leftarrow key$
while $i > 1$ **and** $A[\text{Parent}[i]] < A[i]$
    do exchange $A[i] \leftrightarrow A[\text{Parent}[i]]$
      $i \leftarrow \text{Parent}[i]$

Heap-Insert(A, key)

$heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$
$A[heap\text{-}size[A]] \leftarrow -\infty$
Heap-Increase-Key(A, $heap\text{-}size[A]$, key)

Examples

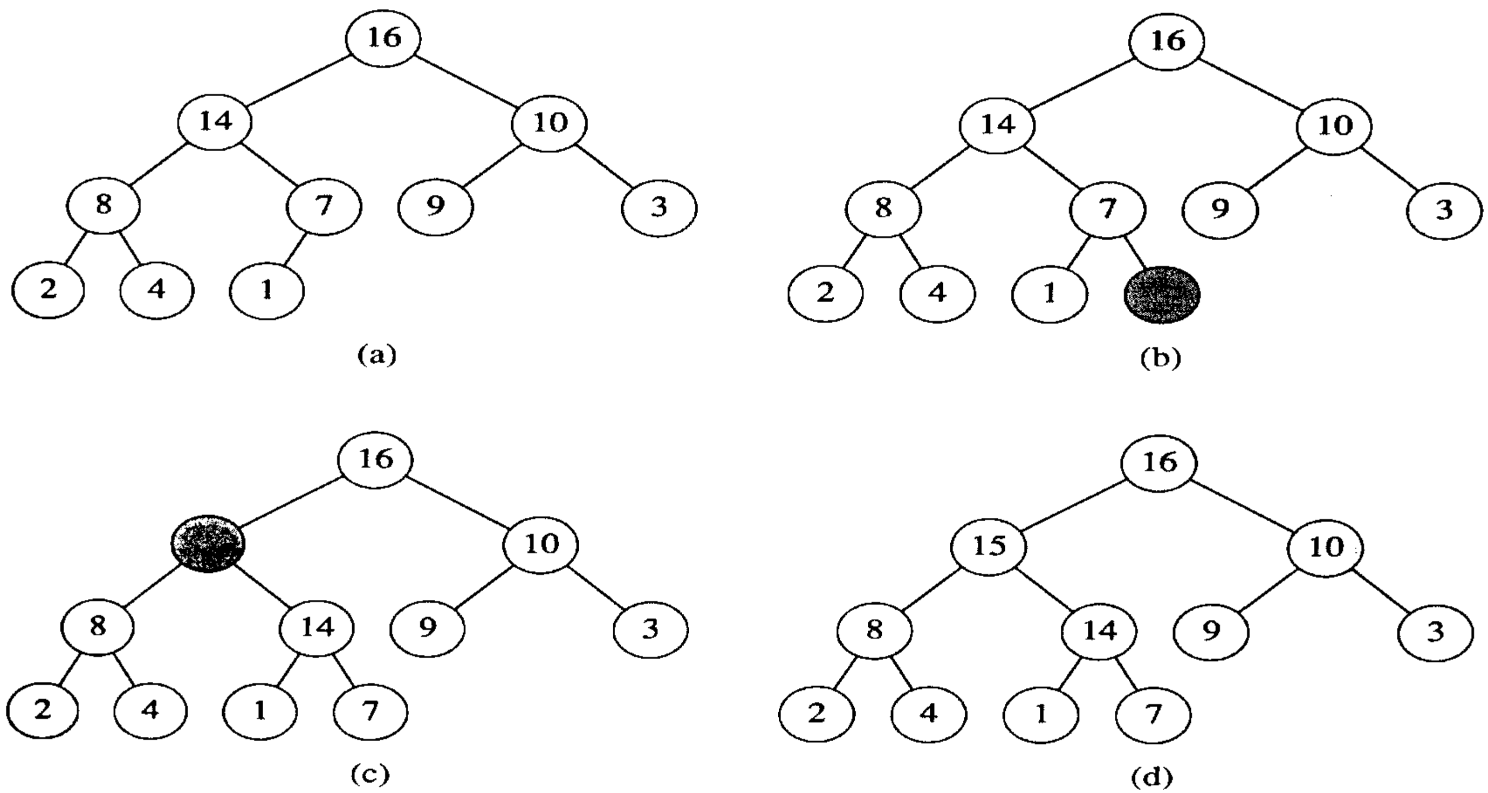


Figure 7.5 The operation of HEAP-INSERT. **(a)** The heap of Figure 7.4(a) before we insert a node with key 15. **(b)** A new leaf is added to the tree. **(c)** Values on the path from the new leaf to the root are copied down until a place for the key 15 is found. **(d)** The key 15 is inserted.

Conclusion

- what a heap is
- how to build a heap
- how to use a heap for sorting
- how to analyze heapsort's running time
- how to use a heap for priority queues