Heap Sort and Priority Queue

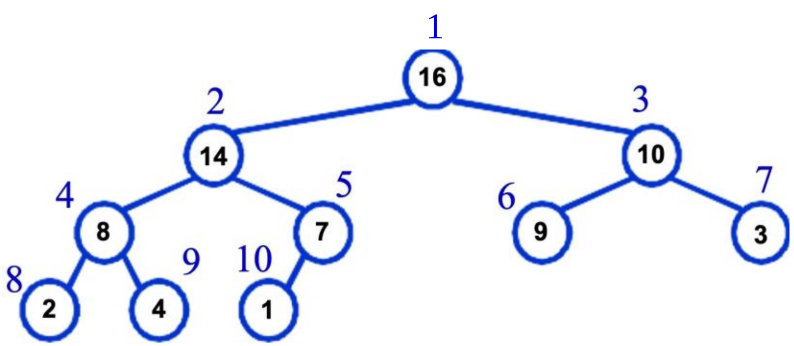
Charles Aunkan Gomes
Lecturer, Dept. of CSE
United International University
charles@cse.uiu.ac.bd

Binary Heaps

- The (binary) heap data structure is an array object that can be viewed as a complete binary tree

- Each node of the tree corresponds to an element of the array that stores the value in the node.
  - The tree is completely filled on all levels except possibly the lowest, where it is filled from the left up to a point.

0	1	2	3	4	5	6	7	8	9	10
10	16	14	10	8	7	9	3	2	4	1

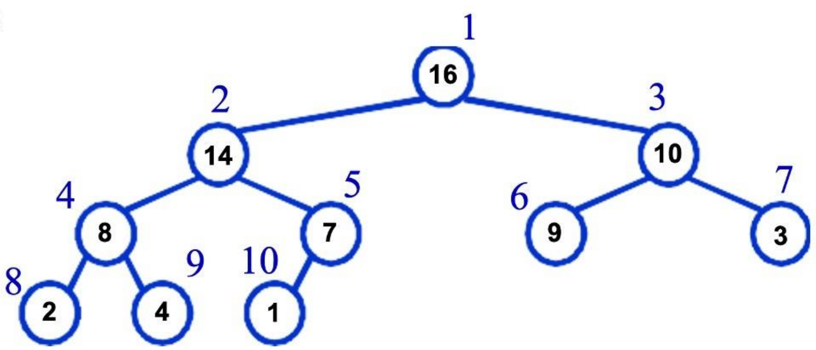


Binary Heaps

- To represent a complete binary tree as an array:
  - The root node is A[1]
  - Node i is A[i]
  - The parent of node i is A[i/2]
  - The left child of node i is A[2i]
  - The right child of node i is A[2i + 1]

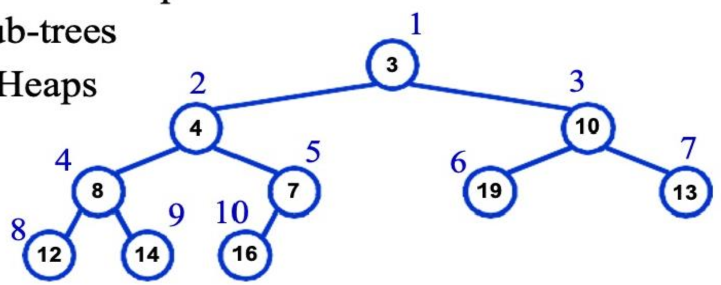
```
Parent(i)
    return floor(i/2)
Right(i)
    return 2i+1
Left(i)
    return 2i
```

0	1	2	3	4	5	6	7	8	9	10
10	16	14	10	8	7	9	3	2	4	1

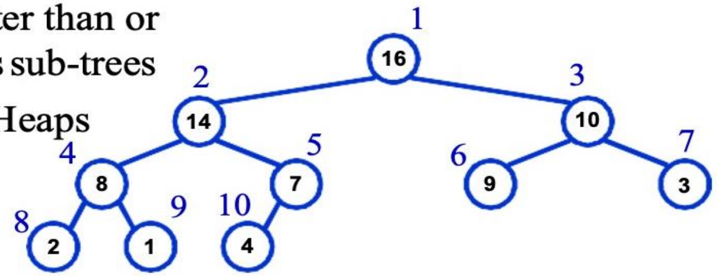


Types of Binary Heaps

- **Min-Heaps:**
  - The element in the root is less than or equal to all elements in both of its sub-trees
  - Both of its sub-trees are Min-Heaps



- **Max-Heaps:**
  - The element in the root is greater than or equal to all elements in both its sub-trees
  - Both of its sub-trees are Max-Heaps

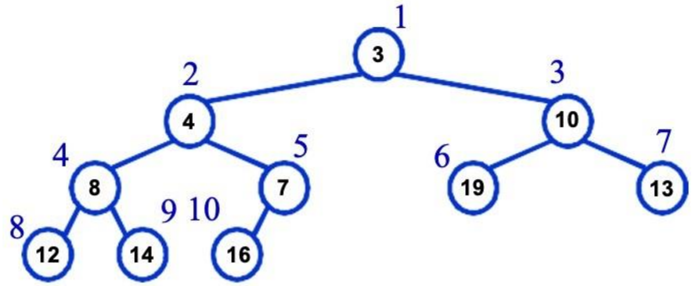


The Min-Heap Property

- Min-Heaps satisfy the *heap property*:

  A[*Parent*(*i*)] ≤ A[*i*] ; for all nodes *i* > 1

  - The value of a node is at least the value of its parent
  - The smallest element in a min-heap is stored at the root
  - Where is the largest element ???
    - Ans: At one of the leaves [leaf indices are *n*/2+1 to *n*]

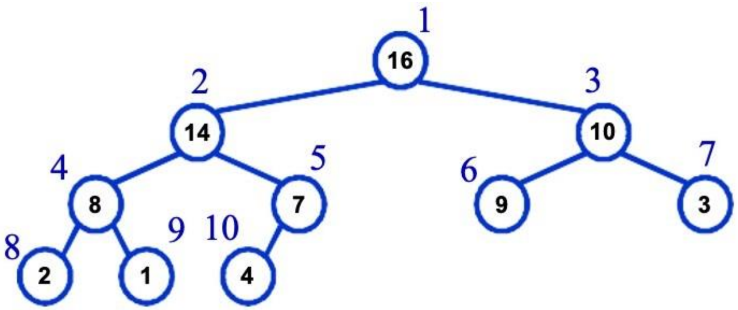


The Max-Heap Property

- Max-Heaps satisfy the *heap property*:

  A[*Parent*(*i*)] ≥ A[*i*] ; for all nodes *i* > 1

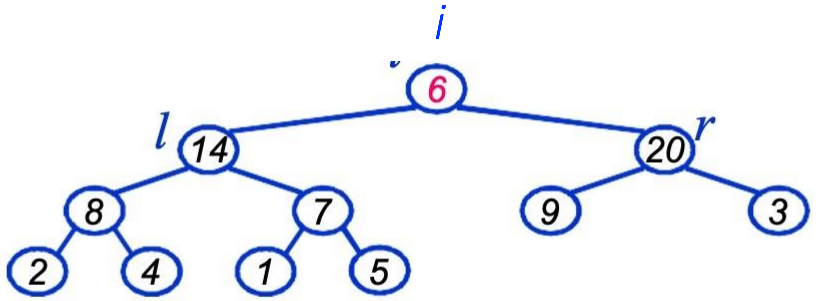
  - The value of a node is at most the value of its parent
  - The largest element in a max-heap is stored at the root
  - Where is the smallest element ???
    - Ans: At one of the leaves [leaf indices are *n*/2+1 to *n*]



Max-Heap Operations: Max-Heapify()

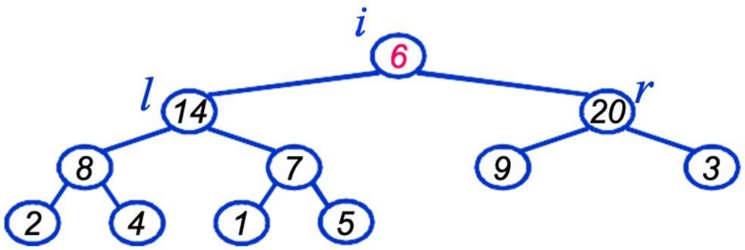
- **Max-Heapify()**: maintain the max-heap property
  - Given: a node *i* in the heap with children *l* and *r*

    : two subtrees rooted at *l* and *r*, assumed to be heaps
  - Problem: The subtree rooted at *i* may violate the heap property (*How?*)
  - Action: let the value of the parent node "float down" so subtree at *i* satisfies the heap property
    - *What will be the basic operation between i, l, and r?*

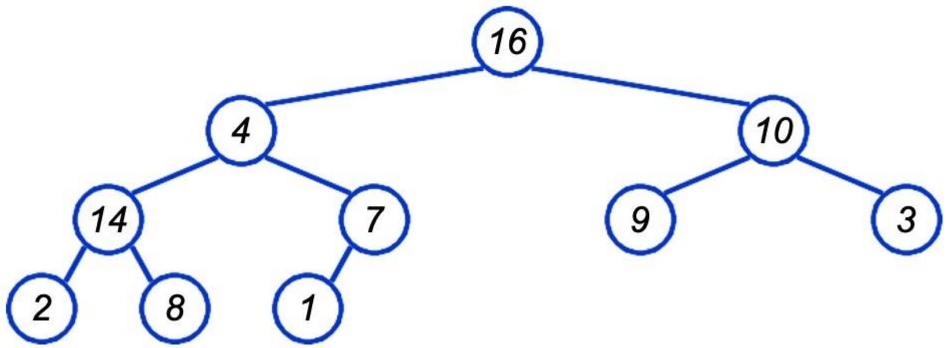


Max-Heap Operations: Max-Heapify()

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

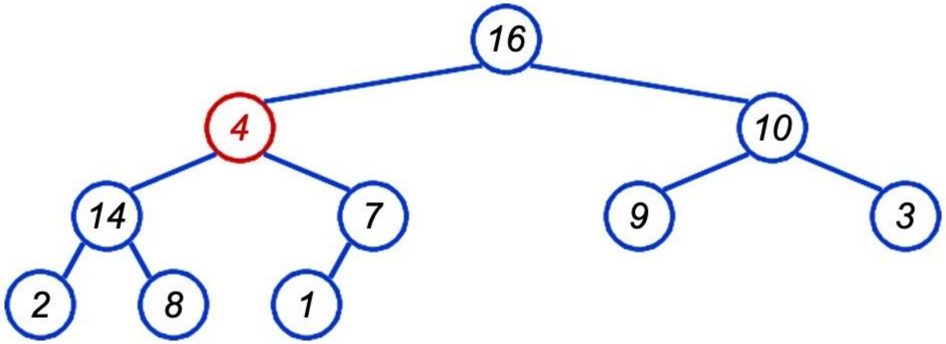


Heapify() - Example



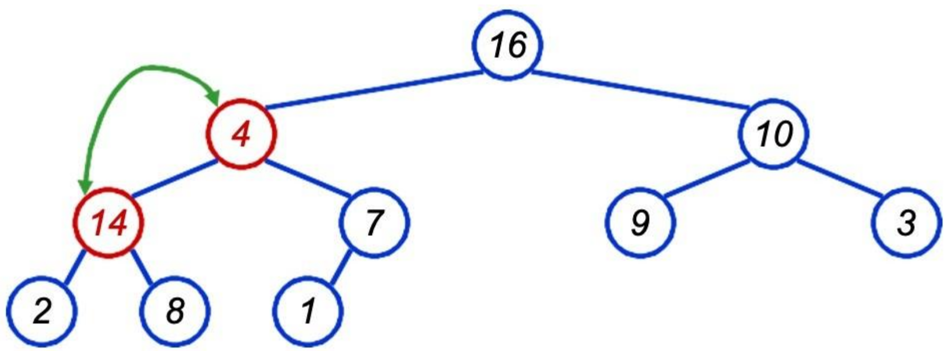
	1	2	3	4	5	6	7	8	9	10
A =	16	4	10	14	7	9	3	2	8	1

Heapify() - Example



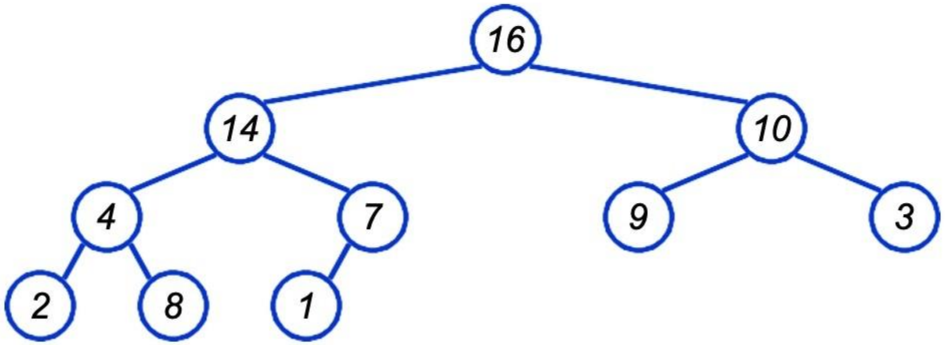
	1	2	3	4	5	6	7	8	9	10
A =	16	4	10	14	7	9	3	2	8	1

Heapify() - Example



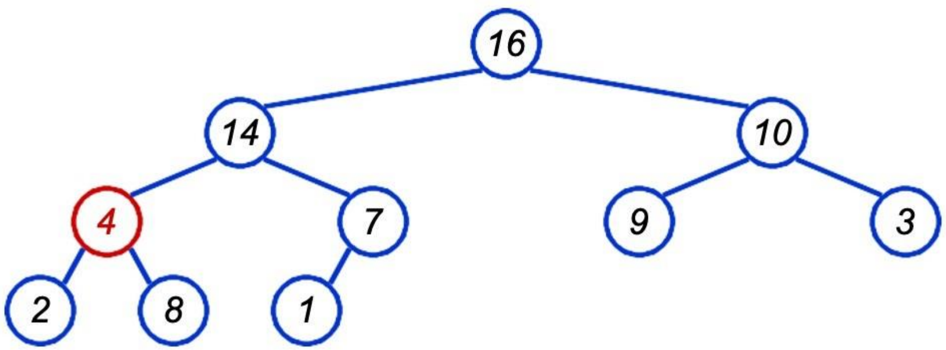
	1	2	3	4	5	6	7	8	9	10
A =	16	4	10	14	7	9	3	2	8	1

Heapify() - Example



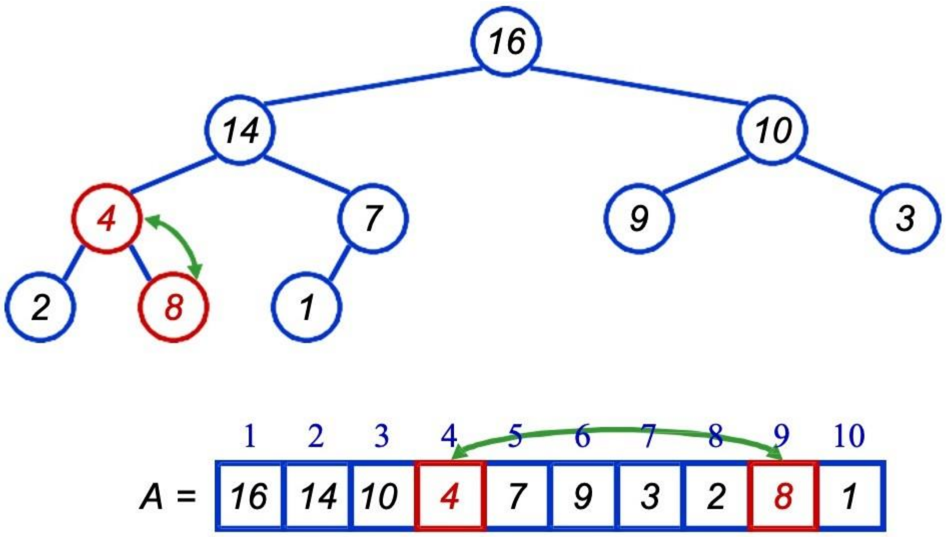
	1	2	3	4	5	6	7	8	9	10
A =	16	14	10	4	7	9	3	2	8	1

Heapify() - Example

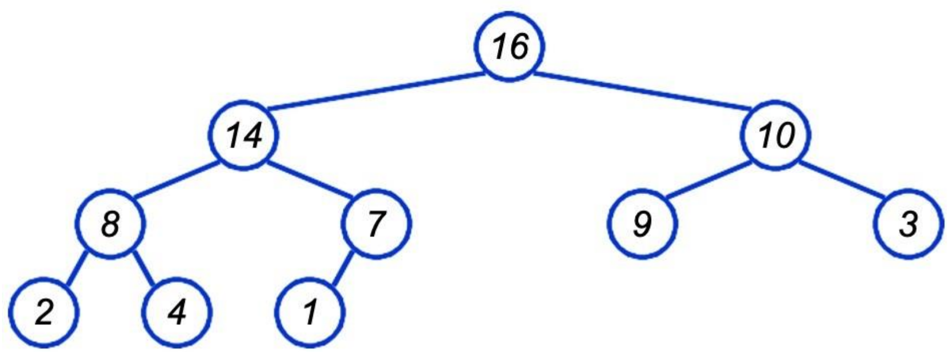


	1	2	3	4	5	6	7	8	9	10
A =	16	14	10	4	7	9	3	2	8	1

Heapify() - Example

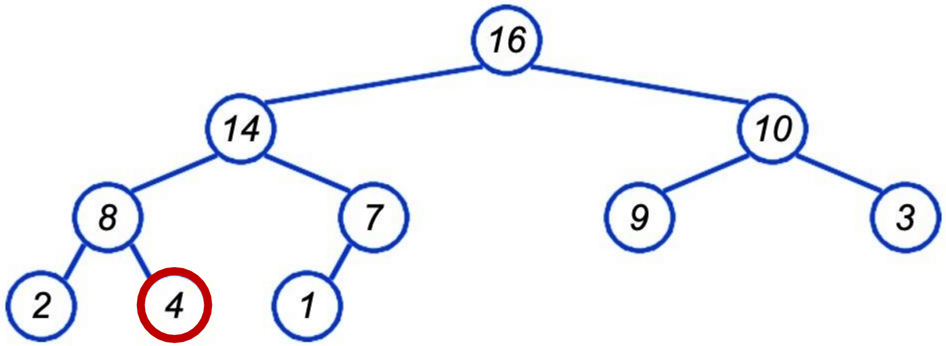


Heapify() - Example



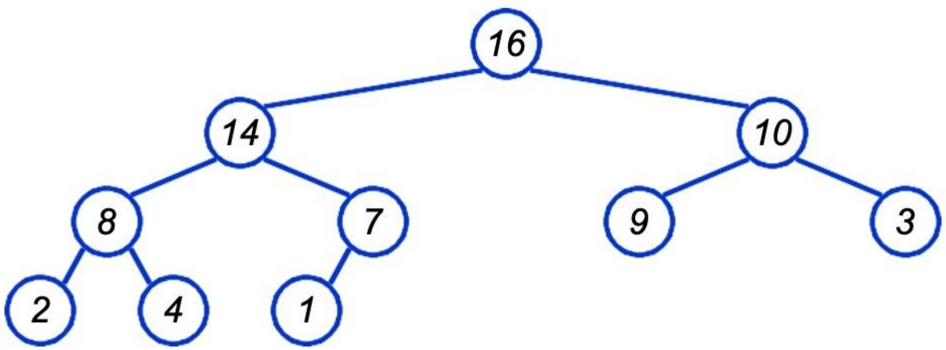
	1	2	3	4	5	6	7	8	9	10
A =	*16*	*14*	*10*	*8*	*7*	*9*	*3*	*2*	*4*	*1*

Heapify() - Example



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

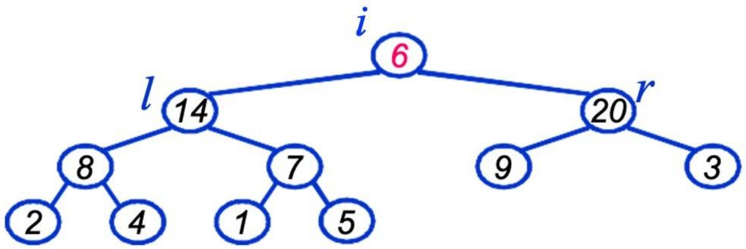
Heapify() - Example



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

Analyzing Heapify()

- Fixing up relationships among the elements $A[i]$, $A[l]$, and $A[r]$ takes O(1) time
- *If the heap at i has n elements, at most how many elements can the subtrees at l or r have?*



- Answer: $2n/3$ (worst case: bottom row half full)
- So time taken by **Heapify()** is given by

  $T(n) \leq T(2n/3) + \Theta(1)$

Analyzing Heapify()

- So we have
  - $T(n) \leq T(2n/3) + \Theta(1)$

- Solving the recurrence, we have
  - $T(n) = O(\log n)$

- Thus, **Heapify()** takes $O(h)$ time for a node at height h.

Heap Operations: BuildHeap()

- We can build a heap in a bottom-up manner by running **Heapify()** on successive subarrays

- Fact: for array of length *n*, all elements in the range A[$\lfloor n/2 \rfloor$ + 1 ... *n*] already satisfies Heap Property (*Why?*)
- So
  - Walk backwards through the array from *n*/2 to 1, calling **Heapify()** on each node.
  - Order of processing guarantees that the children of node *i* are heaps when *i* is processed

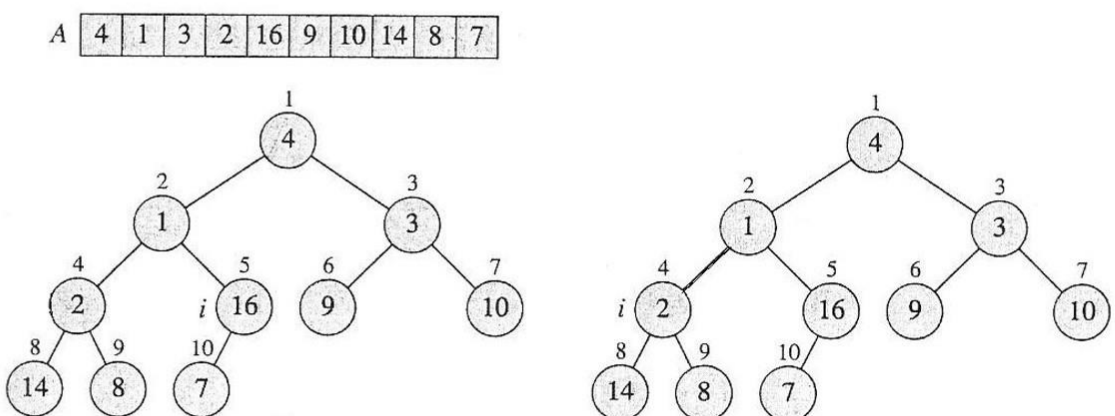
Heap Operations: BuildHeap()

- Converts an unorganized array A into a max-heap.

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

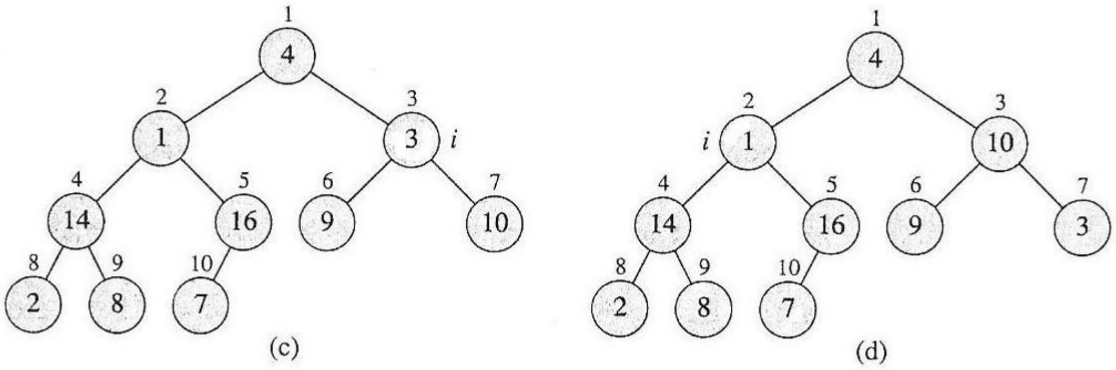
BuildHeap(): Example

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

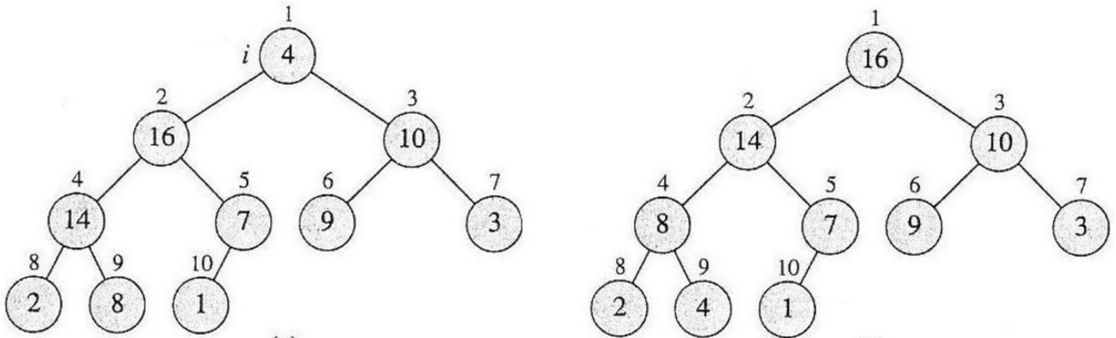


(a) (b)

BuildHeap(): Example



BuildHeap(): Example



(e) (f)

BuildHeap(): Example 2

- Work through example
  A = {14, 11, 33, 22, 56, 49, 30, 24, 18, 37}
  Construct a Min-Heap from the given array using Build-Heap function.

Analyzing BuildHeap()

- Each call to **Heapify()** takes $O(\log n)$ time
- There are $O(n)$ such calls (specifically, $\lfloor n/2 \rfloor$)
- Thus the running time is $O(n \log n)$
  - *Is this a correct asymptotic upper bound?*
  - *Is this an asymptotically tight bound?*

- A tighter bound is $O(n)$
  - *How can this be? Is there a flaw in the above reasoning?*

Analyzing BuildHeap(): Tight

- To **Heapify()** a subtree takes $O(h)$ time, where h is the height of the subtree
  - $h = O(\log m)$, m = # nodes in the subtree
  - The height of most subtrees is small
- Fact: an n-element heap has at most $\lceil n/2^{h+1} \rceil$ nodes of height h
- Prove that **BuildHeap()** takes $O(n)$ time

HeapSort

- Given **BuildHeap()**, a sorting algorithm can easily be 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 **Heapify()**
  - Repeat, always swapping A[1] for A[heap_size(A)]

HeapSort

```
Heapsort(A)
{
   BuildHeap(A);
   for (i = length(A) downto 2)
   {
      Swap(A[1], A[i]);
      heap_size(A) = heap_size(A) - 1;
      Heapify(A, 1);
   }
}
```

HeapSort

- Work through example

  A = {14, 11, 33, 22, 56, 49, 30, 24, 18, 37}

Analyzing HeapSort

- The call to **BuildHeap()** takes $O(n)$ time
- Each of the $(n - 1)$ calls to **Heapify()** takes $O(\log n)$ time
- Thus the total time taken by **HeapSort()**

$$= O(n) + (n - 1)\, O(\log n)$$

$$= O(n) + O(n \log n)$$

$$= O(n \log n)$$

Priority Queue

- A queue that is ordered according to some priority value
- The heap data structure is incredibly useful for implementing *priority queues*
  - A data structure for maintaining a set *S* of elements, each with an associated value or *key*
  - Supports the operations **Insert()**, **Maximum()**, and **ExtractMax()**

Priority Queue Operations

- Insert(S, x) – Inserts element x into set S, according to its priority
- Maximum(S) – Returns, but does not remove, 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

- *How could we implement these operations using a heap?*

Priority Queue Operations

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

```
HEAP-EXTRACT-MAX(A)
if heap-size[A] < 1
   then error "heap underflow"
max ← A[1]
A[1] ← A[heap-size[A]]
heap-size[A] ← heap-size[A] − 1
MAX-HEAPIFY(A, 1)
return max
```

Priority Queue Operations

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

Priority Queue Operations

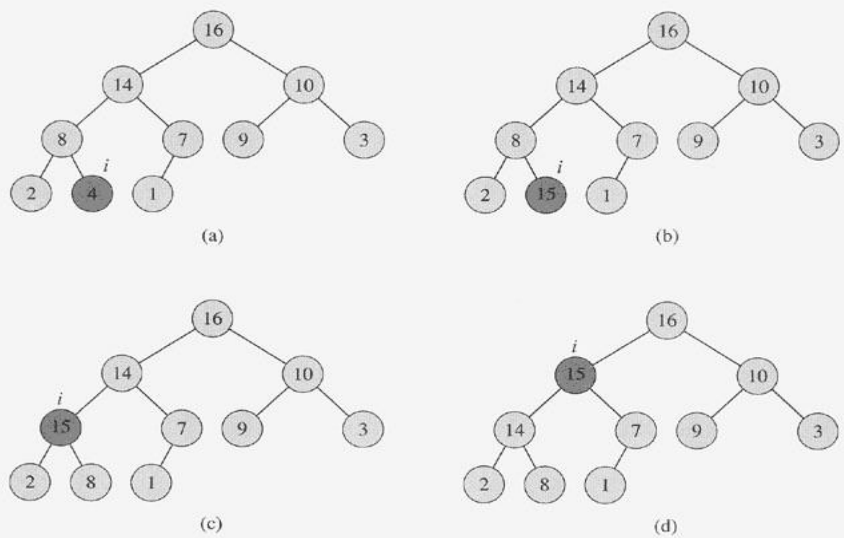


Figure 6.5 The operation of HEAP-INCREASE-KEY. **(a)** The max-heap of Figure 6.4(a) with a node whose index is i heavily shaded. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 4–6, the node and its parent have exchanged keys, and the index i moves up to the parent. **(d)** The max-heap after one more iteration of the **while** loop. At this point, $A[\text{PARENT}(i)] \geq A[i]$. The max-heap property now holds and the procedure terminates.

Priority Queue Operations

MAX-HEAP-INSERT(A, key)

```
1  heap-size[A] ← heap-size[A] + 1
2  A[heap-size[A]] ← −∞
3  HEAP-INCREASE-KEY(A, heap-size[A], key)
```


THANK YOU