HEAP SORT AND PRIORITY QUEUE

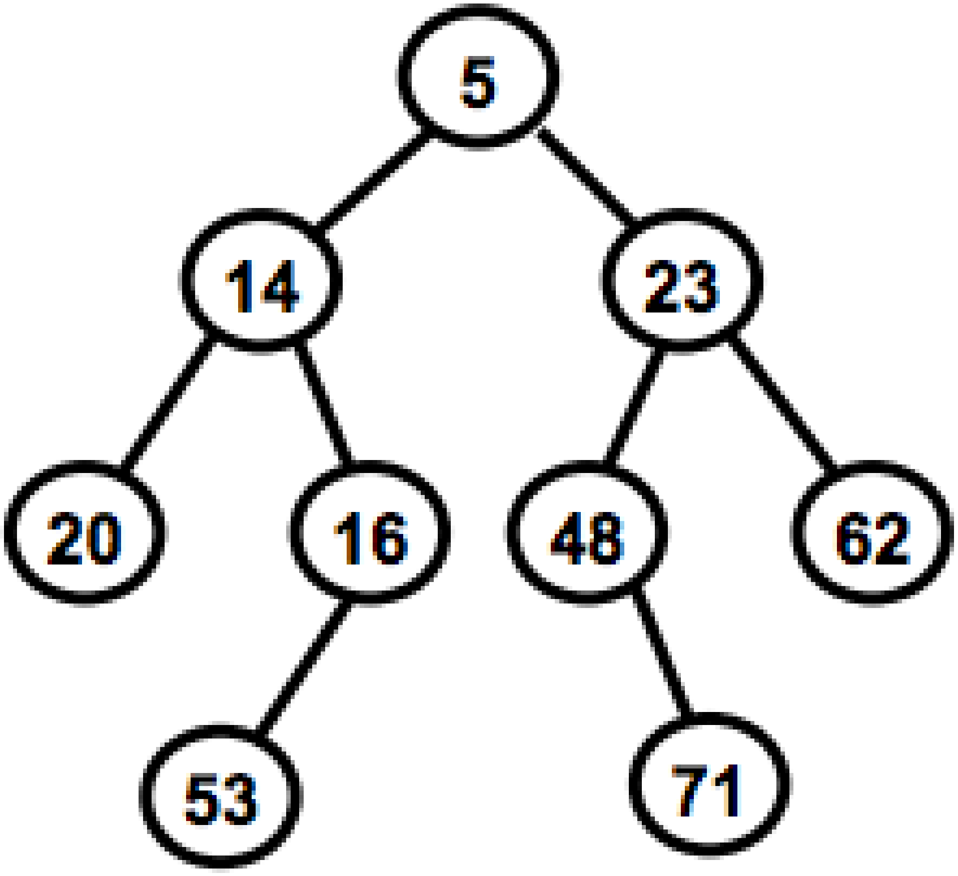
Punjab University College of Information Technology (PUCIT)
University of the Punjab, Lahore, Pakistan.

Heap

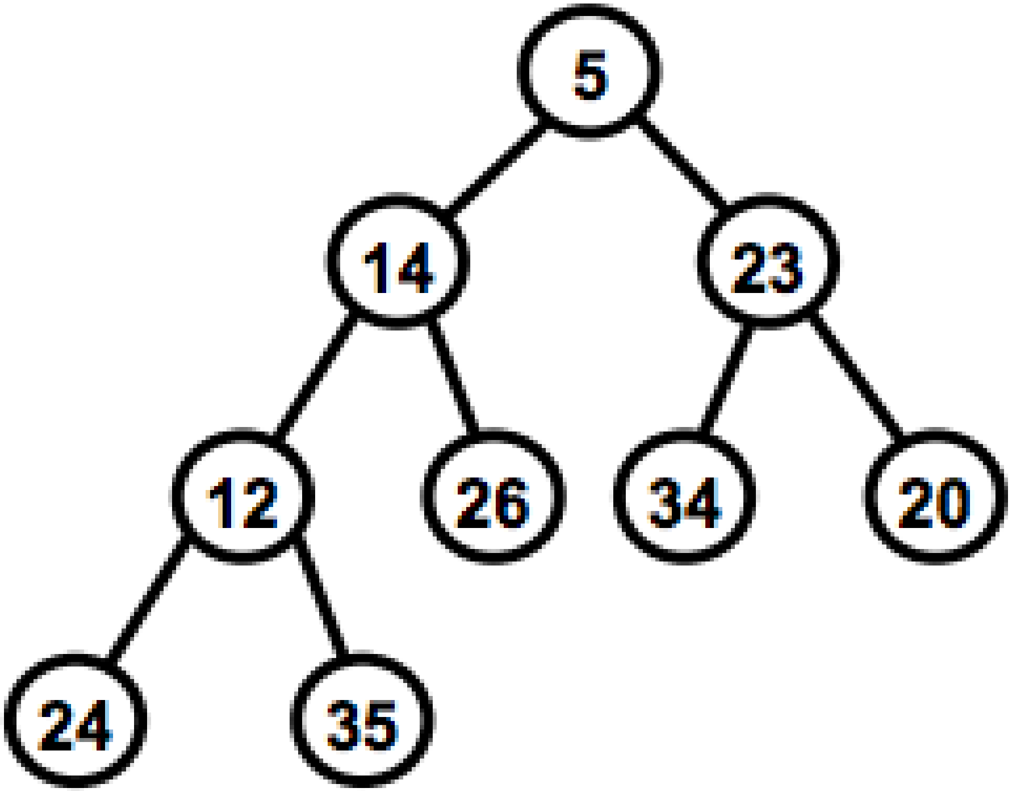
- Heap is a binary tree with two properties:
  - **Structure/Shape property**
    - Each level (except possibly the bottom most level) is completely filled
    - The bottom most level may be partially filled (from left to right)

  - **Order property order property**
    - **max-heap**, the data contained in each node is greater than (or equal to) the data in that node’s children.
    - **min-heap**, the data contained in each node is less than (or equal to) the data in that node’s children.

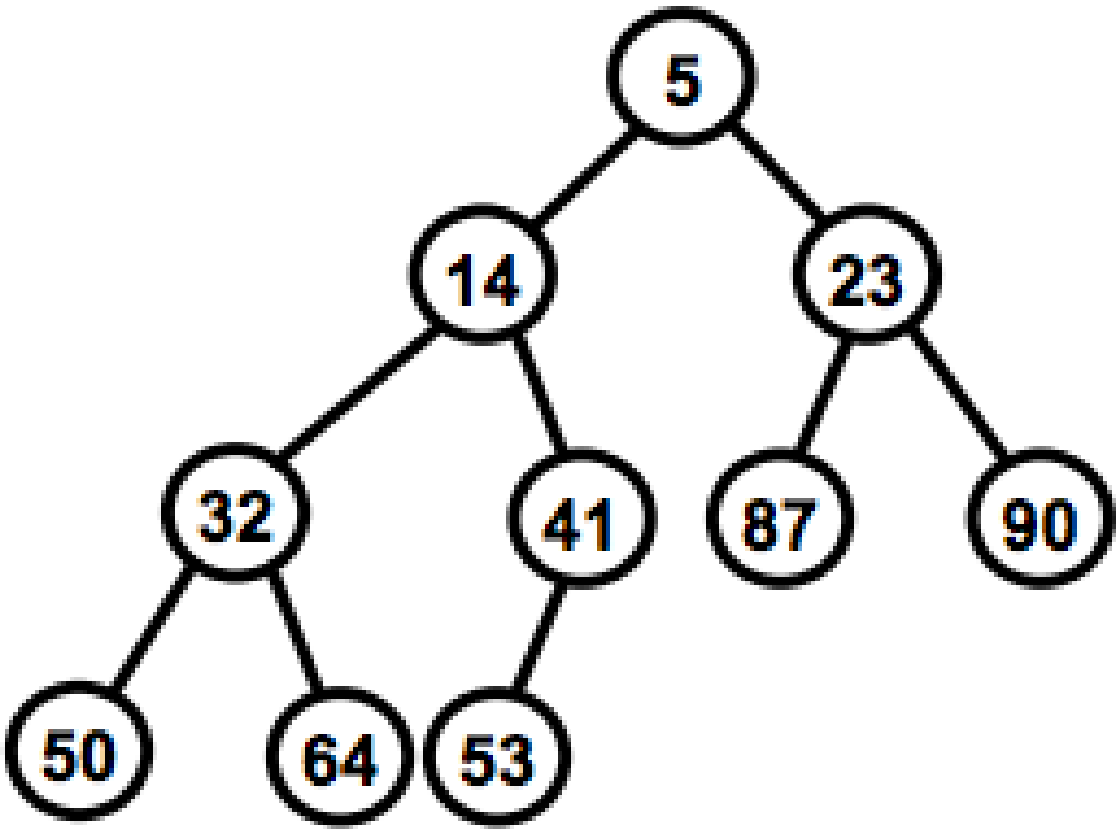
Example 1: Is it a min-heap?



Example 2: Is it a min-heap?



Example 2: Is it a min-heap?



Heap: A Simple Implementation

- Use an array to hold the data.

- Store the root in position 1.
  - We won't use index 0 for this implementation.

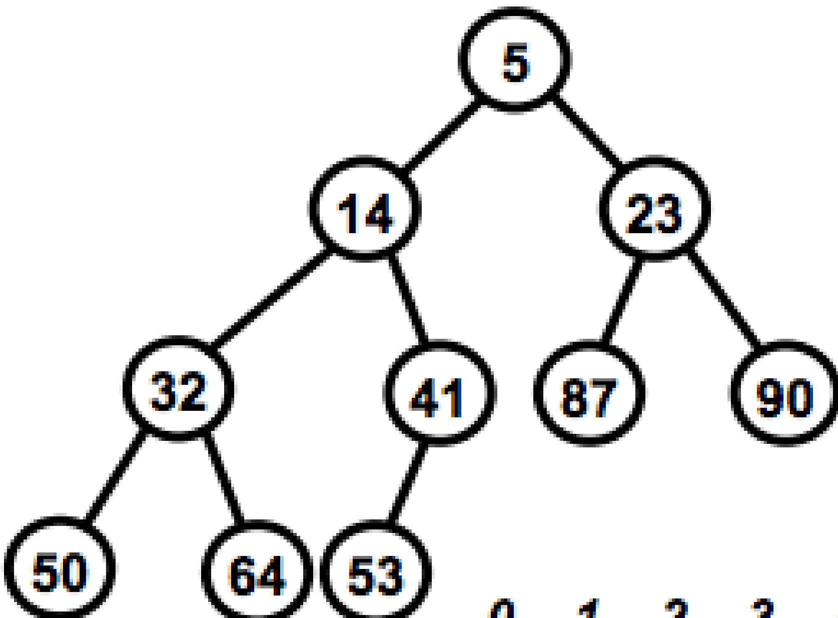
- For any node in position i,
  - its **left child** (if any) is in position **2i**
  - its **right child** (if any) is in position **2i + 1**
  - its **parent** (if any) is in position **i/2**

Heap: A Simple Implementation (Cont.)

PARENT(i)
1 **return** $\lfloor i/2 \rfloor$

LEFT(i)
1 **return** $2i$

RIGHT(i)
1 **return** $2i + 1$



For node at i:
Left child is at 2i
Right child is at 2i12
Parent is at $\lfloor$i/2$\rfloor$

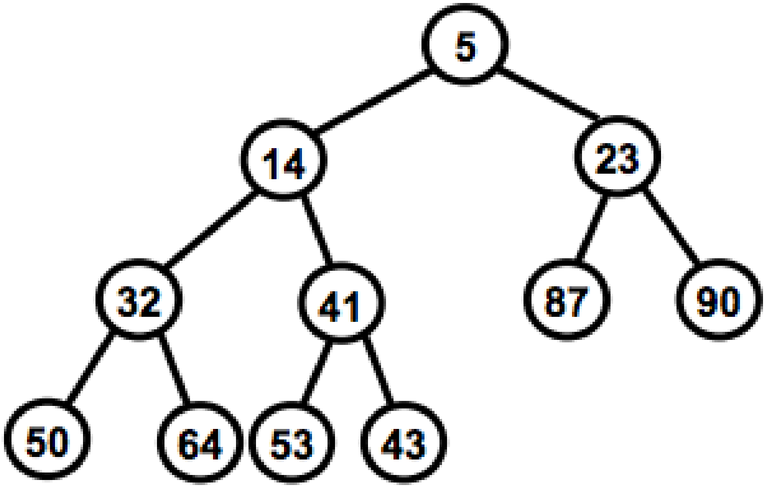
0	1	2	3	4	5	6	7	8	9	10
	5	14	23	32	41	87	90	50	64	53

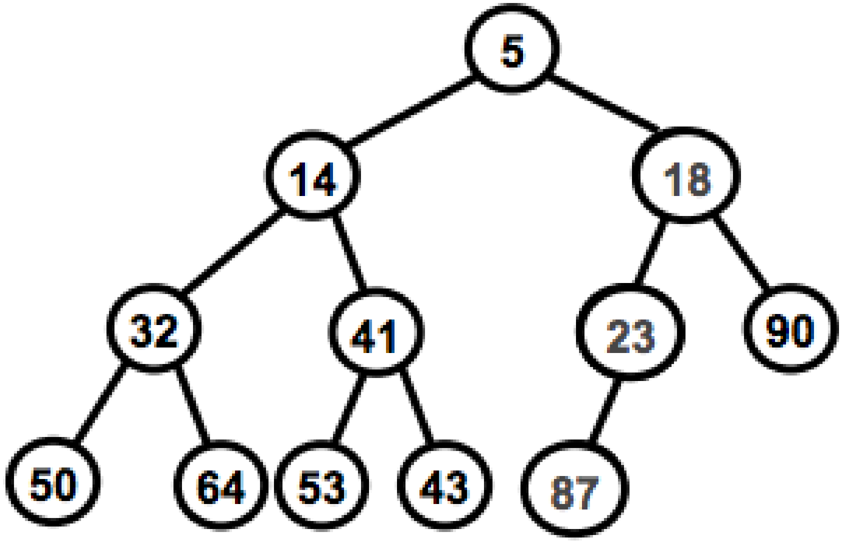
Heap: Insertion

- Place the new element in the next available position in the array.
- Compare the new element with its parent. If the new element is smaller, than swap it with its parent.
- Continue this process until either the new element's parent is smaller than or equal to the new element, or the new element reaches the root.

Heap: Insertion

Inset 18

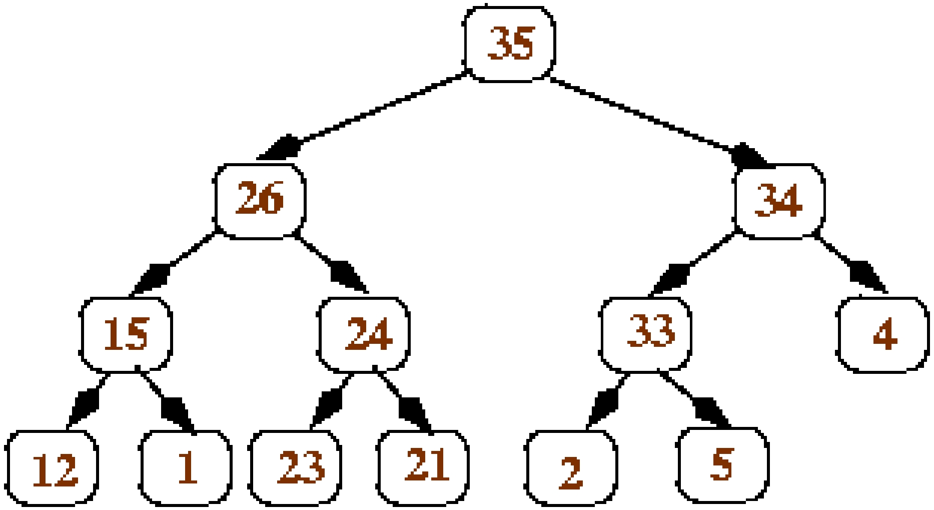




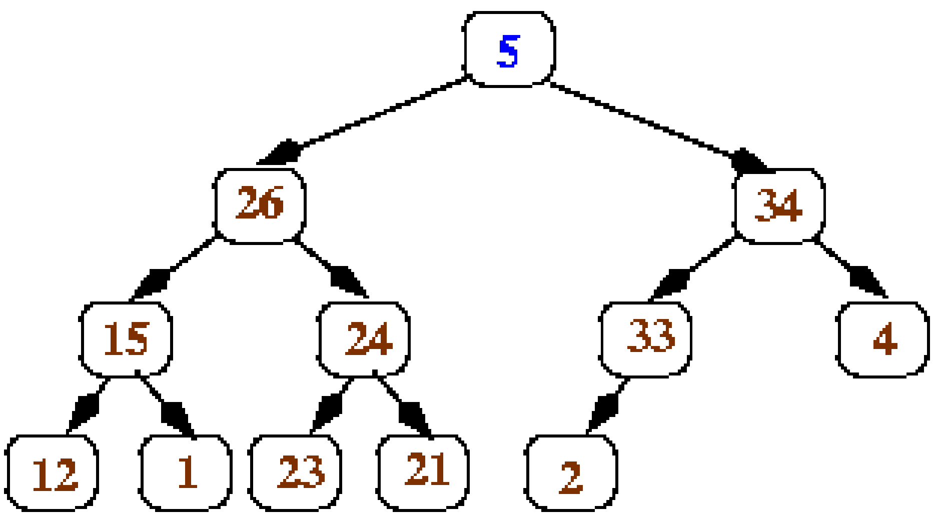
Heap: Remove

- Replace the value in the node you want to delete with the value at the end of the array (which corresponds to the heap's rightmost leaf at depth d). Remove that leaf from the tree.

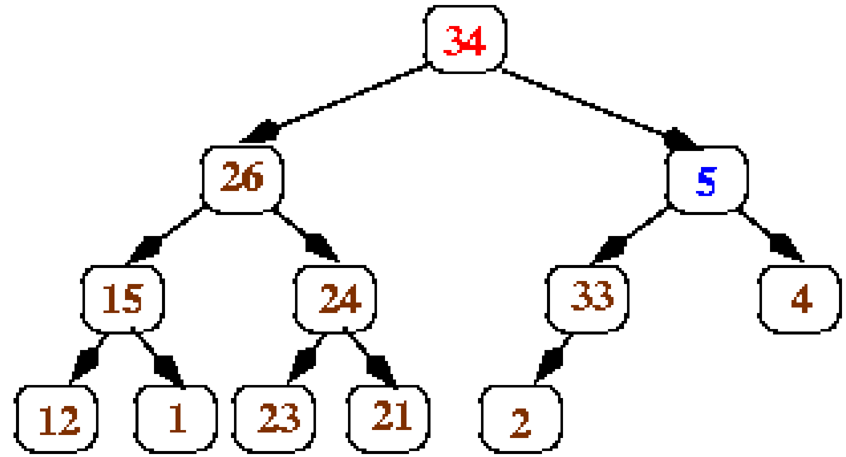
- Now work your way down the tree, swapping values to restore the order property:
  - each time, if the value in the current node is less than one of its children (if heap is max-heap), then swap its value with the larger child (that ensures that the new root value is larger than both of its children)



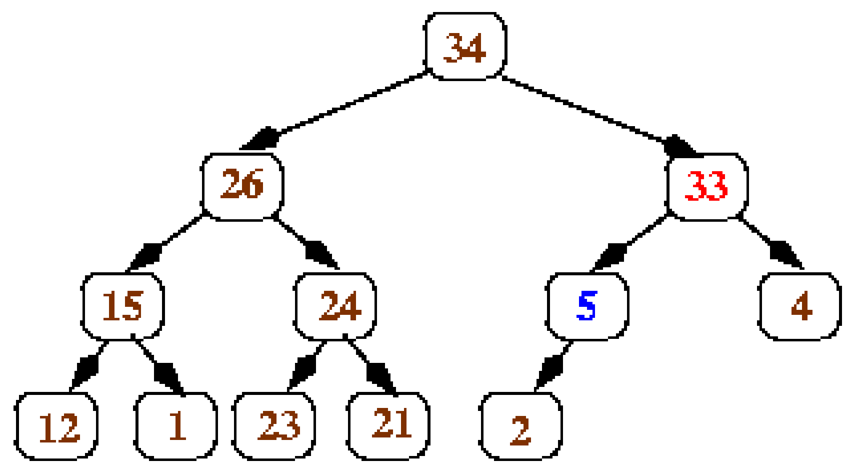
original heap



step 1: extract root value and replace with last lea



step 2: swap with larger child



step 2: swap with larger child again

All done!

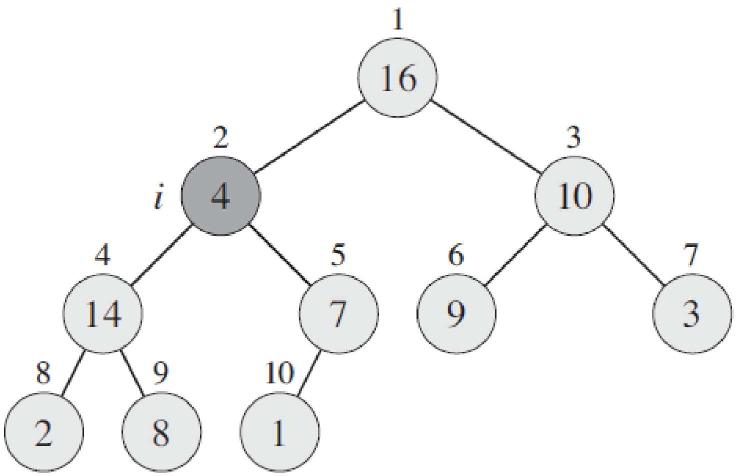
Maintaining the heap property

In order to maintain the max-heap property, we call the procedure MAX-HEAPIFY. Its inputs are an array A and an index i into the array. When it is called, MAX-HEAPIFY assumes that the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps, but that $A[i]$ might be smaller than its children, thus violating the max-heap property. MAX-HEAPIFY lets the value at $A[i]$ "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

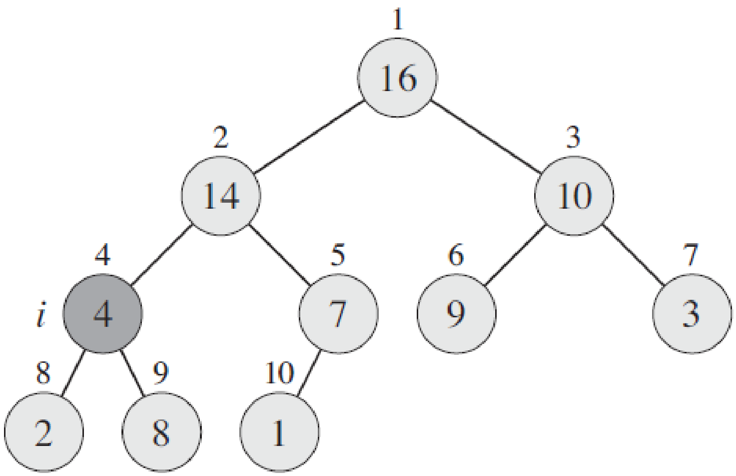
```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

```
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)
```

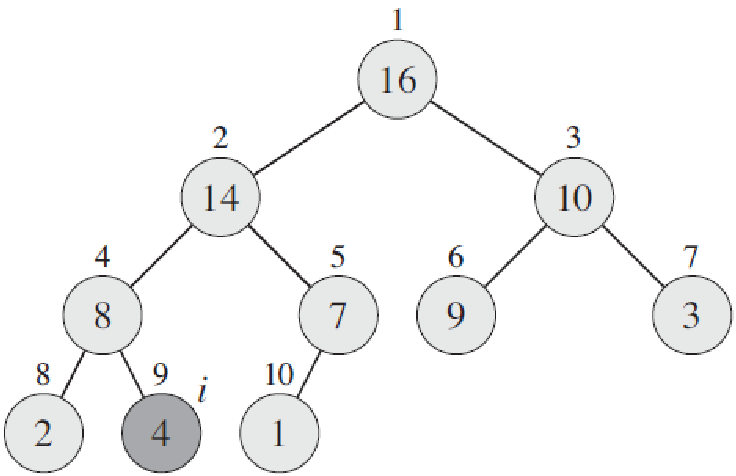
MAX-HEAPIFY(A, 2) , where A.*heap-size = 10.*



(a)



(b)



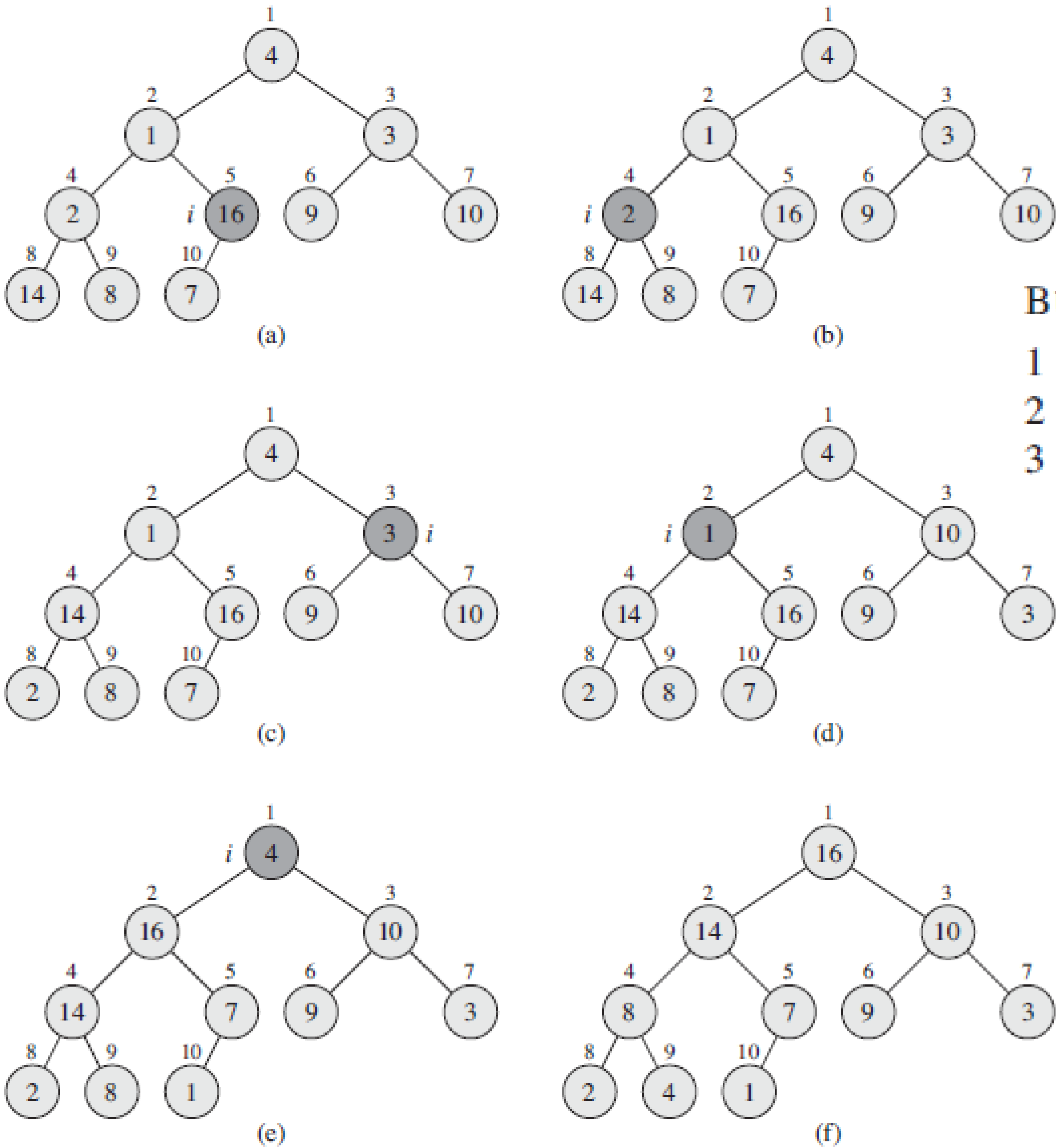
(c)

Building a heap

We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array A[1 … n], where n = A.*length, into a max-heap.*

```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

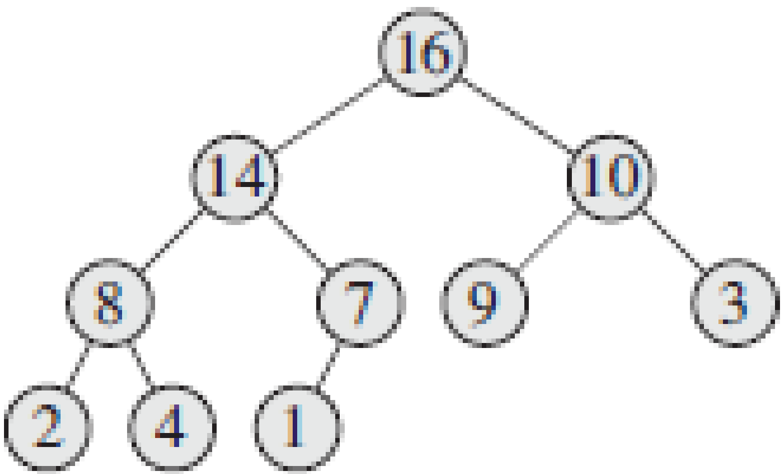
A	4	1	3	2	16	9	10	14	8	7




```
BUILD-MAX-HEAP(A)
    A.heap-size = A.length
    for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

```
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)
```

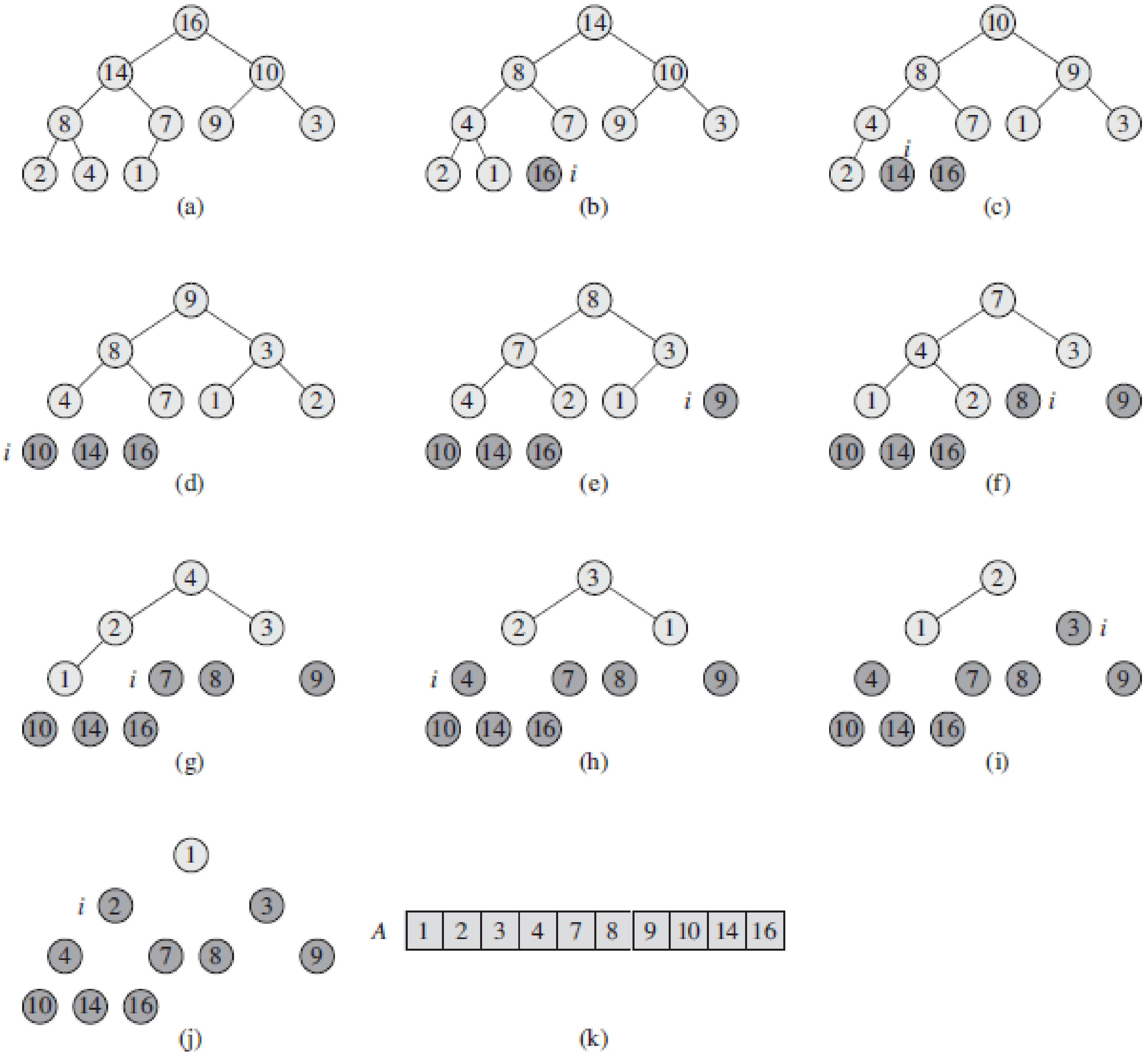
Lets try to see how HEAPSORT works!



(a)

```
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)
```

Heap Sort works in: **O(nlogn)**


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
A 1 2 3 4 7 8 9 10 14 16
(k)

Priority Queue

- Priority Queue is one of the most popular application of Heap!

- A Priority Queue is different from a normal queue, because instead of being a “first-in-first-out”, values come out in order by priority.

A ***priority queue*** is a data structure for maintaining a set S of elements, each with an associated value called a ***key***. A ***max-priority queue*** supports the following operations:

INSERT(S, x) inserts the element x into the set S, which is equivalent to the operation $S = S \cup \{x\}$.

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, which is assumed to be at least as large as x's current key value.

- Key is used to maintain priority in the set S

Priority Queue

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

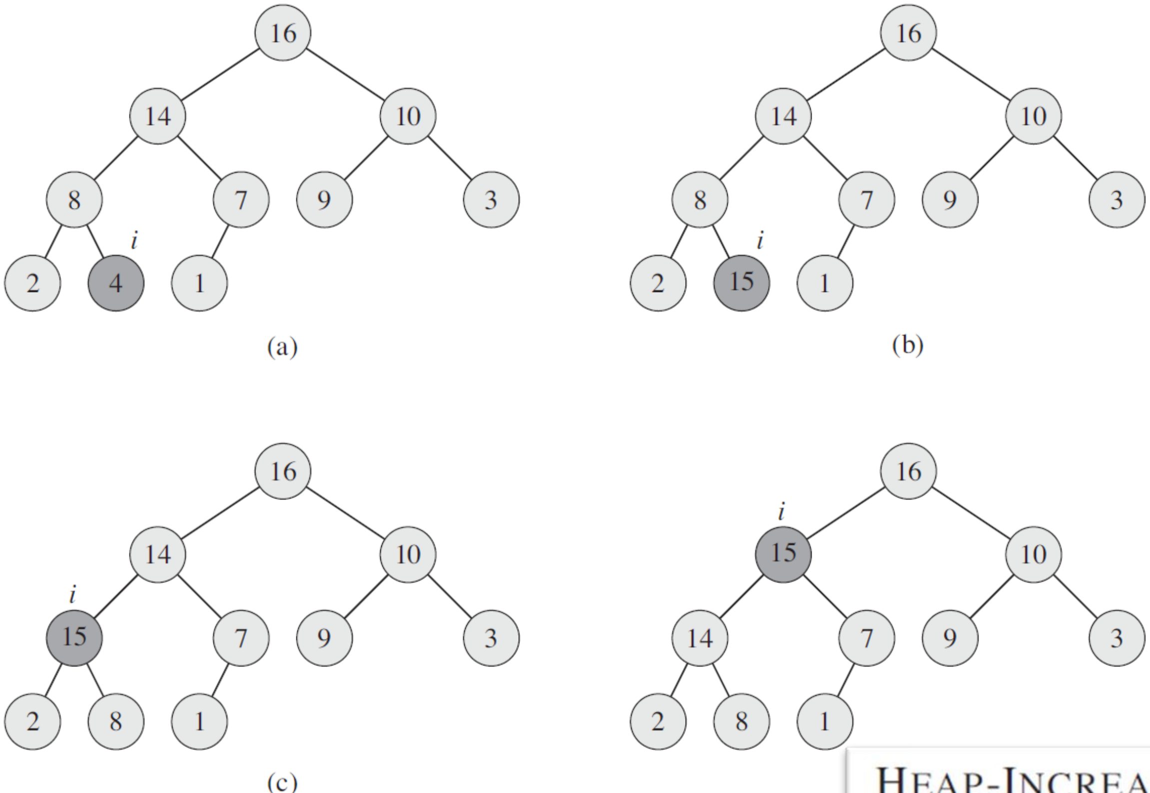
```
HEAP-EXTRACT-MAX(A)
    if A.heap-size < 1
        error "heap underflow"
    max = A[1]
    A[1] = A[A.heap-size]
    A.heap-size = A.heap-size - 1
    MAX-HEAPIFY(A, 1)
    return max
```

Priority Queue

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

HEAP-INCREASE-KEY(A,9,15)

Remember i = 9 which is index and key=15 new key to update!




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

Priority Queue

MAX-HEAP-INSERT(A, key)

1 $A.heap\text{-}size = A.heap\text{-}size + 1$
2 $A[A.heap\text{-}size] = -\infty$
3 HEAP-INCREASE-KEY$(A, A.heap\text{-}size, key)$

Very simple idea, create a new node (line 1 and 2)
Then call adjust the key using HEAP-INCREASE-KEY method!