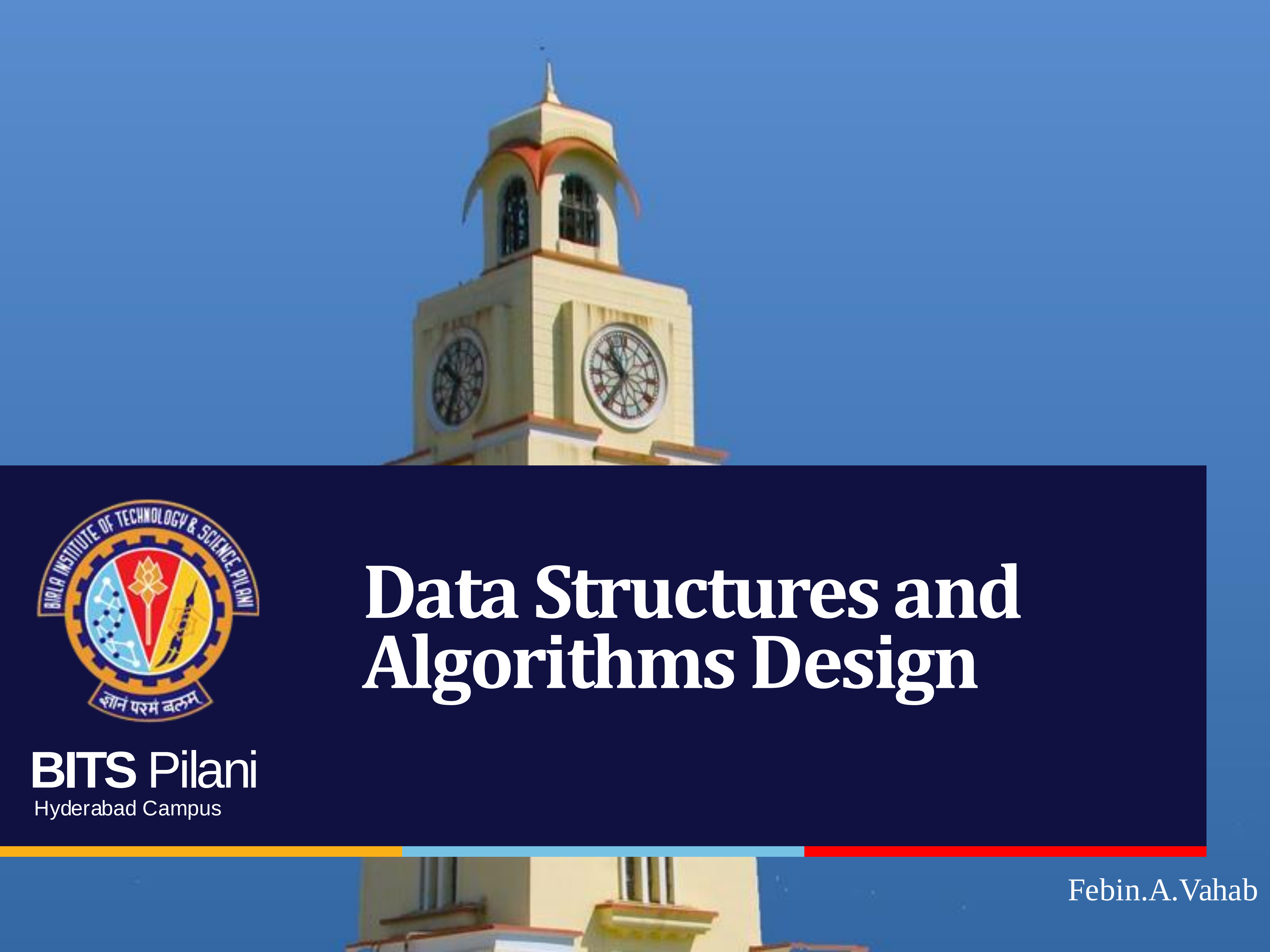


BITS Pilani
Hyderabad Campus

# Data Structures and Algorithms Design

Febin.A.Vahab

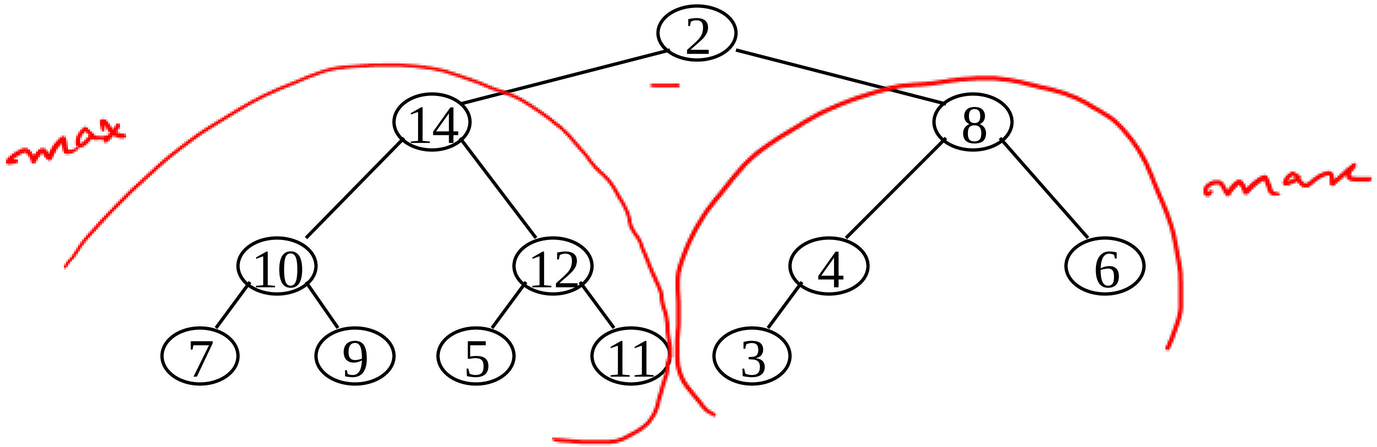
# SESSION 4 -PLAN

| Contact Sessions(#) | List of Topic Title | Text/Ref Book/external resource |
|---|---|---|
| 4 | Heap implementation of priority queue, Heap sort | T1: 2.4<br>T2:6 |

## Heapify

- Before discussing the method for building heap from an arbitrary complete binary tree, we discuss a simpler problem.
- Let us consider a binary tree in which left and right subtrees of the root satisfy the heap property but not the root.

  See the following fig

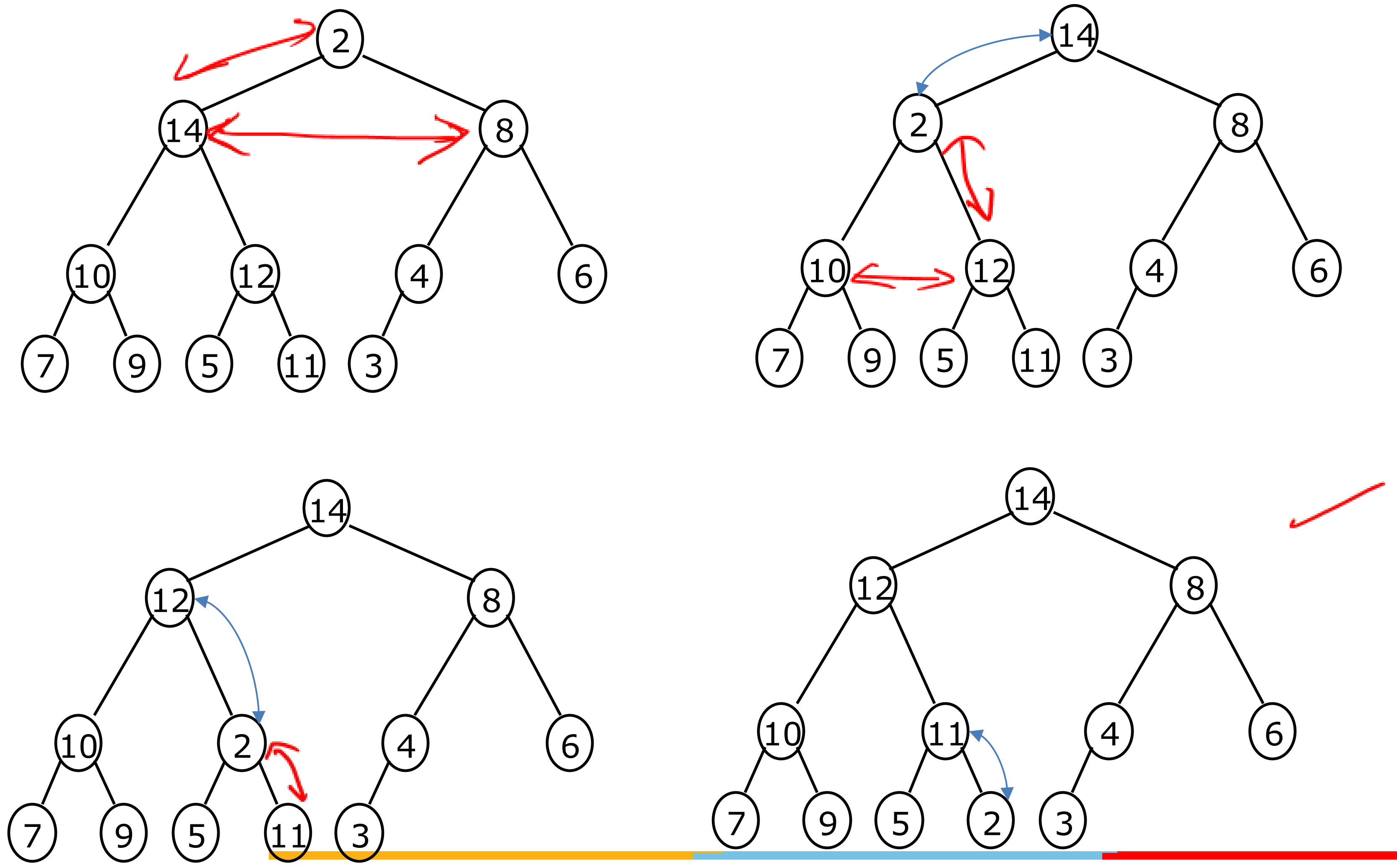


- Now the Question is how to transform the above tree into a Heap?

## Solution for the Question

- Swap the root and left child of root, to make the root satisfy the heap property.
- Then check the subtree rooted at left child of the root is heap or not. If it is we are done, if not repeat the above action of swapping the root with the maximum of its children.
- That is, push down the element at root till it satisfies the heap property.
- The following sequence of fig depicts the **heapification** process

# Heapify


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

# Max-Heapify

```
MAX-HEAPIFY(A, i)
1   l = LEFT(i)
2   r = RIGHT(i)
3   if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5   else largest = i
6   if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8   if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

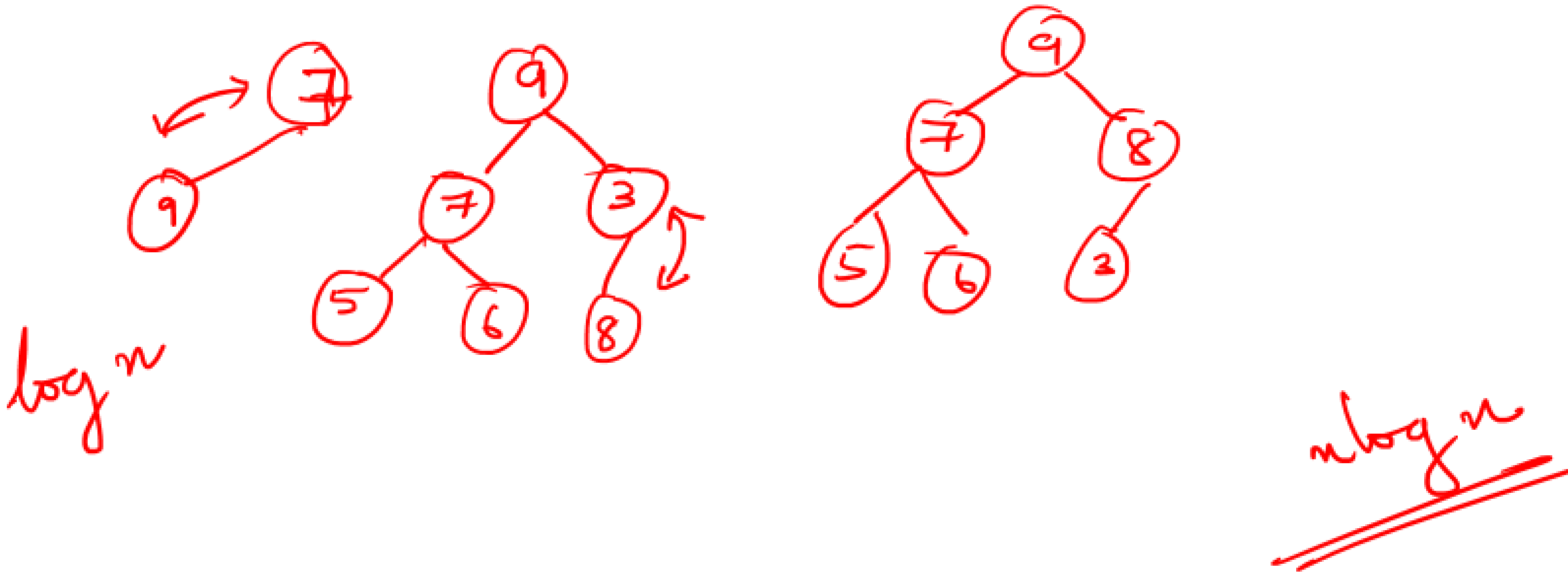
Time complexity????

***O(log n) (Height of the tree)***

# Building a MAX heap

- By means of n successive insert item operations we can construct a heap storing n key-element pairs in O (n log n) time
- Example

(7 9 3 5 6 8 13 32 12 21



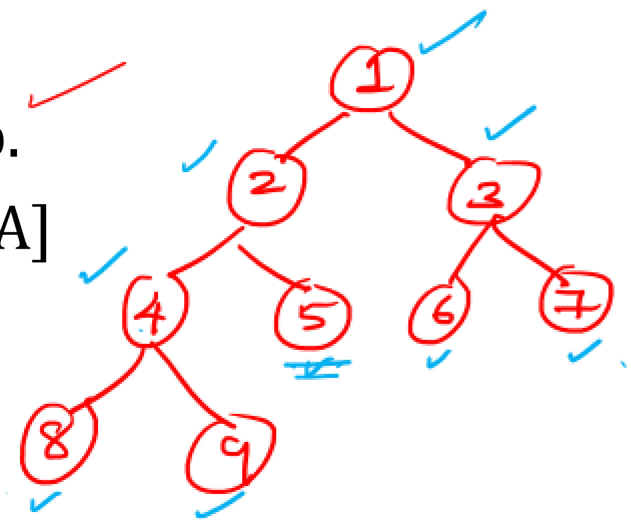
# Building a Max Heap-Bottom Up

9

**Build-Max-Heap(A)**

- // Input: A: an (unsorted) array
- // Output: A modified to represent a heap.
- // Running Time: O(n) where n = length[A]

```
heap-size[A] ← length[A]
for i ← floor(length[A]/2) downto 1
    Max-Heapify(A, i)
```



Why start at n/2?

– Because elements A[n/2 + 1 … n] are all leaves of the tree

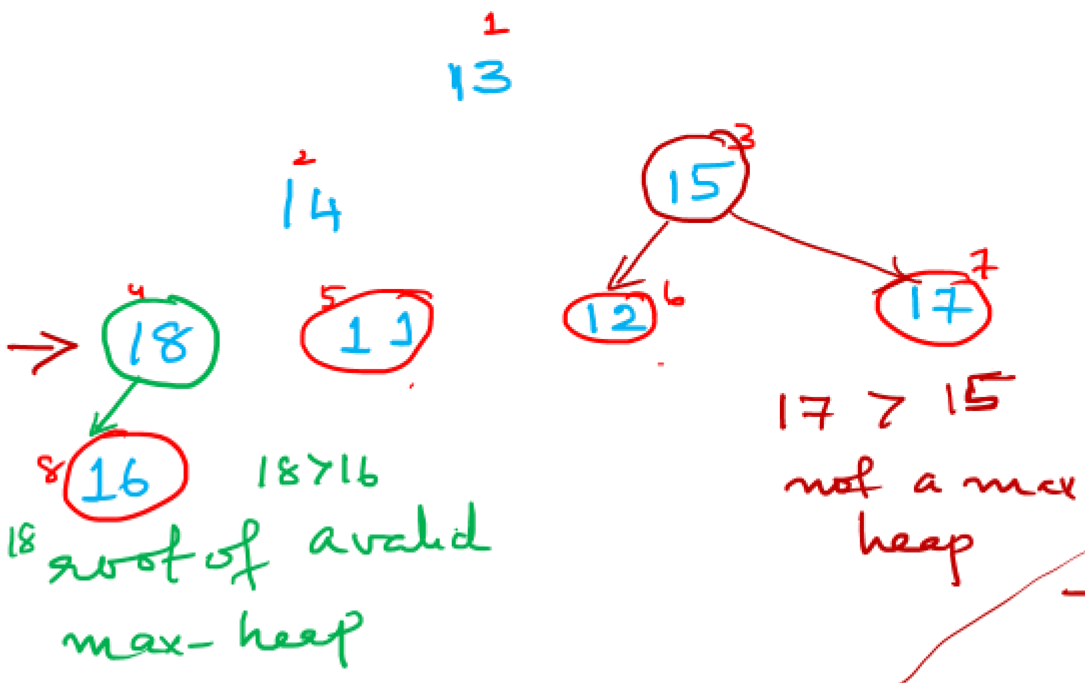
– 2i > n, for i > n/2 + 1

## Building a Max Heap-Bottom Up-Steps

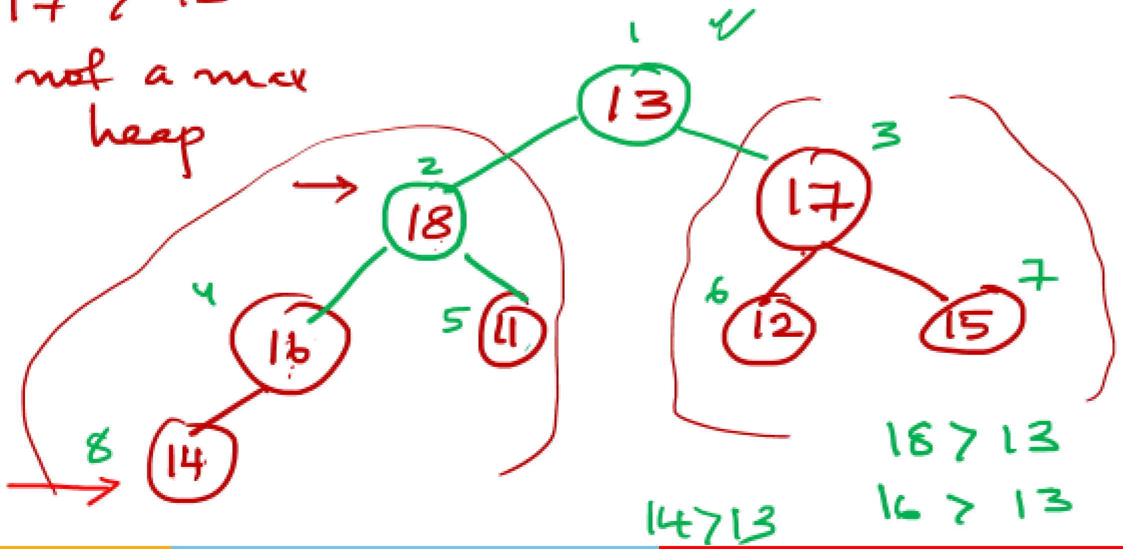
- Step 0: Initialize the structure with keys in the order given
- Step 1: Starting with the last (rightmost) parental node, fix the heap rooted at it, if it doesn’t satisfy the heap condition: keep exchanging it with its largest child until the heap condition holds
- Step 2: Repeat Step 1 for the preceding parental node

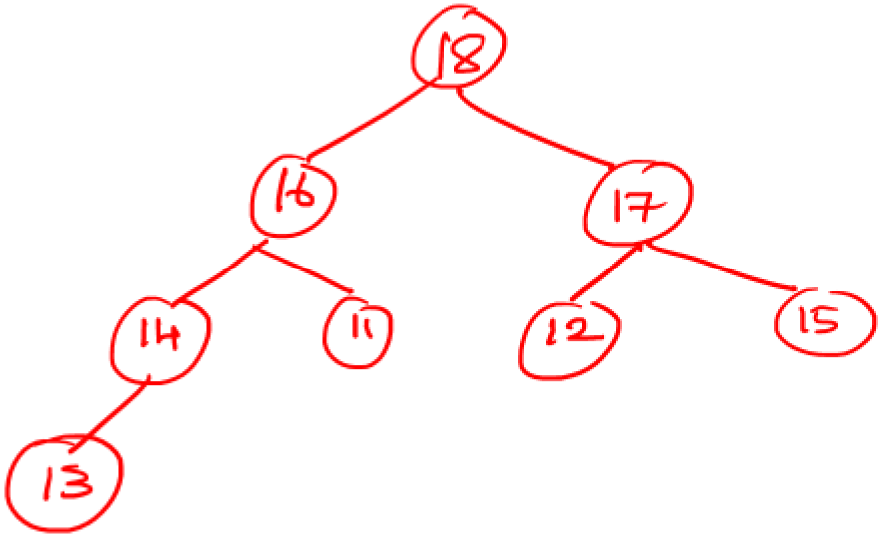
O(L)

13 14 15 18 11 12 17 16



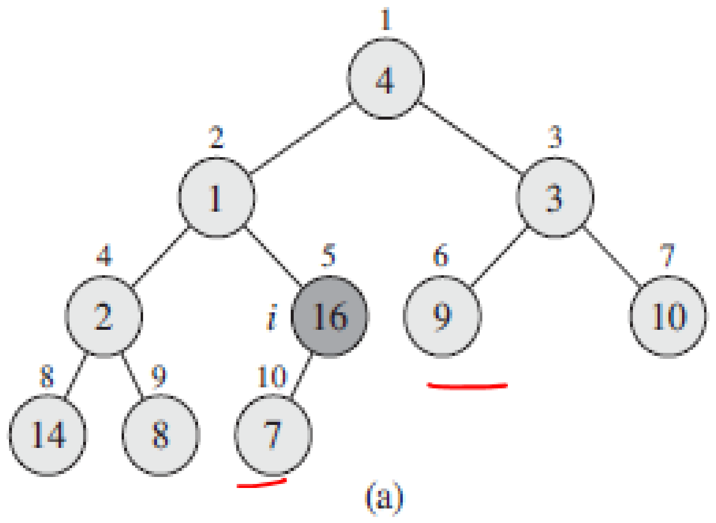
→ leaf nodes are heap by themselves

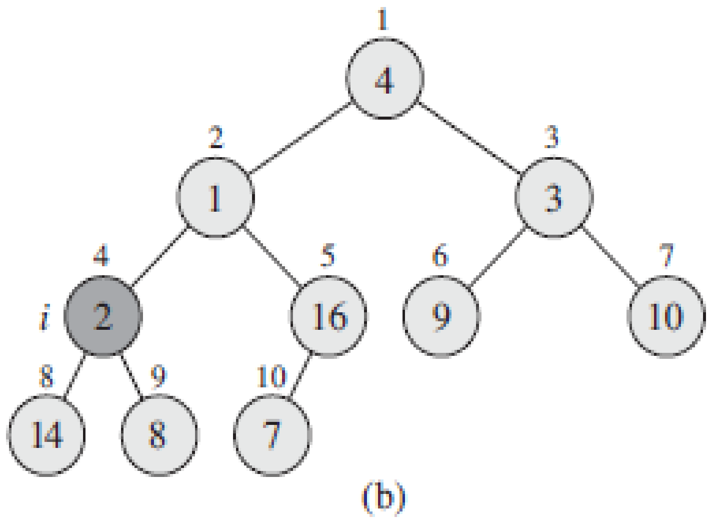



18
16
17
14
11
12
15
13

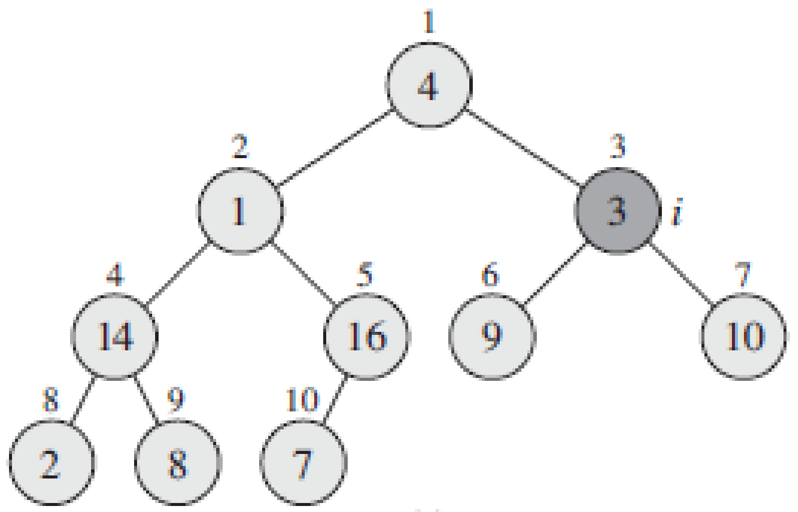
# Building a Max Heap-Bottom Up

| A | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |

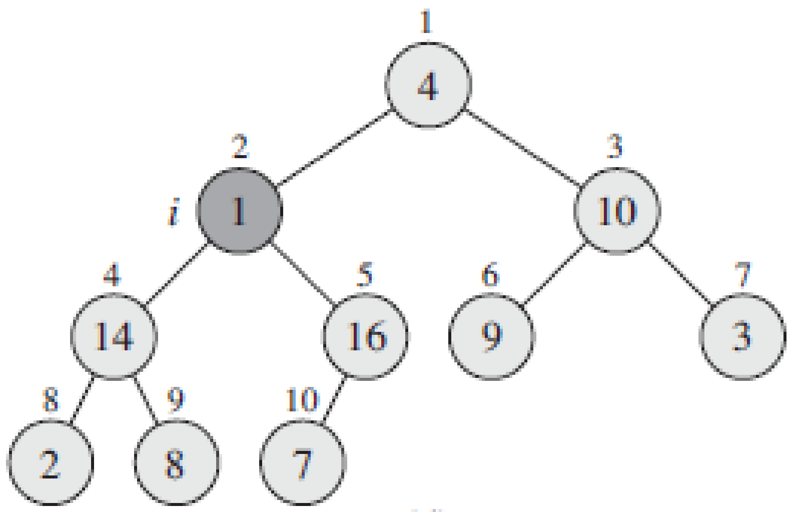




# Building a Max Heap-Bottom Up

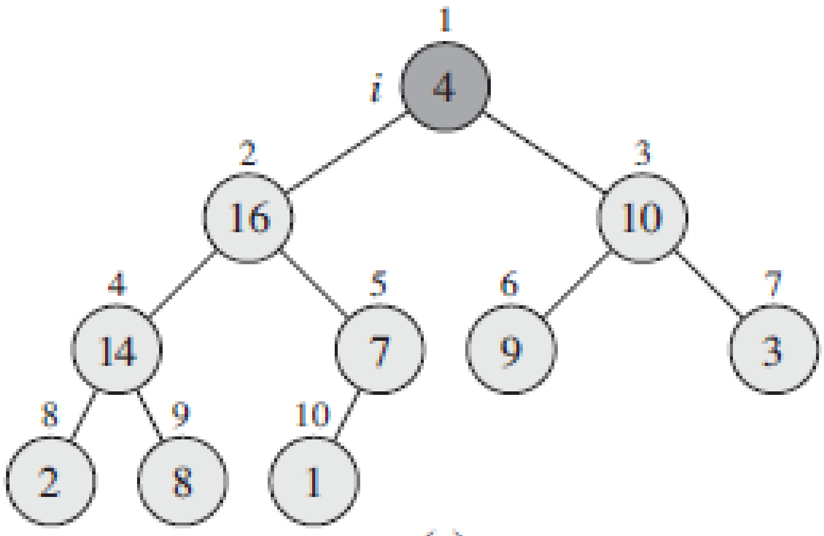


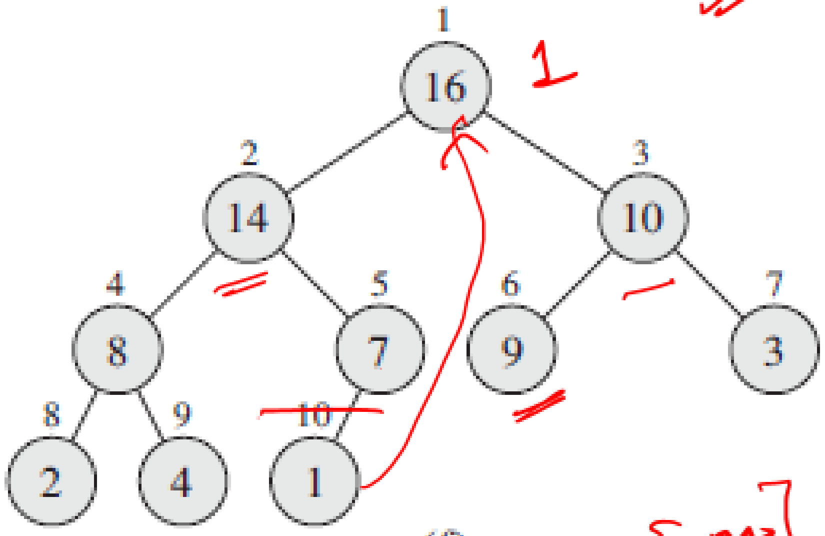
(c)



(d)

# Building a  Max Heap-Bottom Up


(e)


(f)

A[n] ⟷ A[1

# Building a Max Heap-Bottom Up-Analysis

- The time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree
- The heights of most nodes are small
- Our tighter analysis relies on the properties that an
  - n-element heap has height lg n
  - There are at most **n/2 $^{(h+1)}$** nodes at any height h
  - ie
  - Max_Heapify takes O(1) for time for nodes that are one level above the leaves, and in general, O(L) for the nodes that are L levels above the leaves.

$\frac{n}{2}$ $\frac{n}{4}$

# Building a Max Heap-Bottom Up-Analysis

- The total cost of BUILD-MAX-HEAP

$$\rightarrow T(n) = \sum_{h=0}^{logn} n/2^{h+1} * O(h) =$$

$O(n)$

$$= O(n/2 * \sum_{h=0}^{logn} h/2^h)$$

$$<= O(n/2 * \sum_{h=0}^{inf} h/2^h)$$

$$<= O(n * [1 + 1/2 + 1/4 + ....]) \; infinte \; GP$$

$$<= O(n * [1/(1-(1/2))]$$

$$<= O(2n)$$

$$<= O(n))$$

# Building a Max Heap-Bottom Up-Analysis

**Running Time of Build-Max-Heap**

**Trivial Analysis**: Each call to Max-Heapify requires log(n) time, we make n such calls ⇒ O(n log n).

**Tighter Bound**: Each call to Max-Heapify requires time O(h) where h is the height of node i. Therefore running time is

$$\sum_{h=0}^{\log n} \frac{n}{(2^{h+1})} * O(h) = \mathbf{O(n)}$$

# Priority Queue ADT

- A priority queue is a container of elements, each having an associated key that is provided at the time the element is inserted.
- The name "priority queue“ comes from the fact that keys determine the "priority" used to pick elements to be removed
- An item is a pair (key, element)
- Main methods of the Priority Queue ADT
  - insertItem(k, o)
    inserts an item with key k and element o
  - removeMin()
    removes the item with smallest key and returns its element

1

## Priority Queue ADT

- Additional methods
  - minKey()
    returns, but does not remove, the smallest key of an item
  - minElement()
    returns, but does not remove, the element of an item with smallest key
  - size(), isEmpty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

# Priority Queue ADT -Total Order Relation

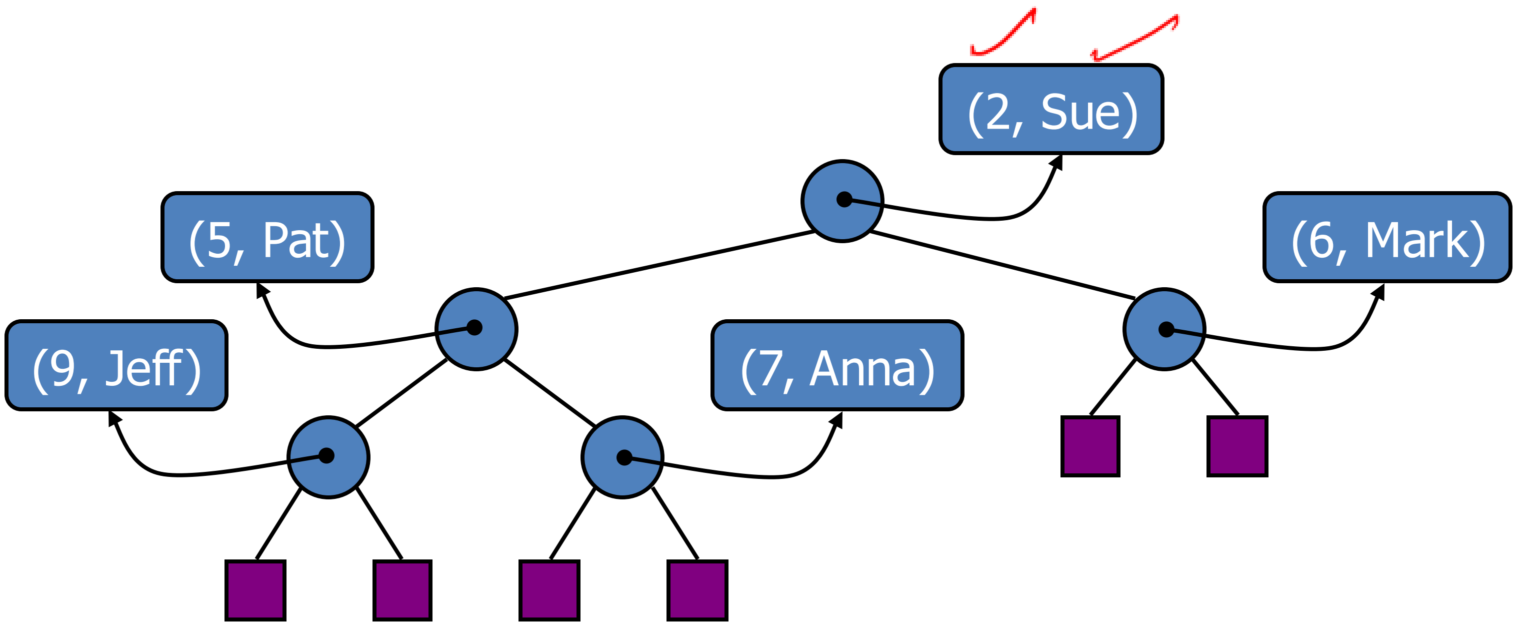
- Keys in a priority queue can be arbitrary objects on which an order is defined.
- Two distinct items in a priority queue can have the same key
- Total order relation

# Sorting with a Priority Queue

- We can use a priority queue to sort a set of comparable elements
- Given a collection C of n elements that can be compared according to a total order relation, and we want to rearrange them in increasing order (or at least in nondecreasing order if there are ties).
  - Insert the elements into a PQ one by one with a series of **insertItem(e, e)** operations
  - Remove the elements in sorted order with a series of **removeMin()** operations putting them back into C
- The running time of this sorting method depends on the priority queue implementation

# Heaps and Priority Queues

- We can use a heap to implement a priority queue
- A (key, element) item is stored at each internal node
- **Keep track of the position of the last node**



# Heap-Sort

**Heap-Sort(A)**

**// Input: A: an (unsorted) array**

**// Output: A modified to be sorted from smallest to largest**

**// Running Time: O(n log n) where n = length[A]**

```
Build-Max-Heap(A)
for i = length[A] downto 2
    exchange A[1] and A[i]
    heap-size[A] ← heap-size[A] – 1
    Max-Heapify(A, 1)
```

## Heap Sort-Steps

- Given an array of n element, first we build the heap
- The largest element is at the root, but its position in sorted array should be at the end. So swap the root with the last.
- We have placed the highest element in its correct position we left with an array of n-1 elements. Repeat the same of these remaining n-1 element to place the next largest elements in its correct position.
- Repeat the above step till an elements are placed in their correct positions.

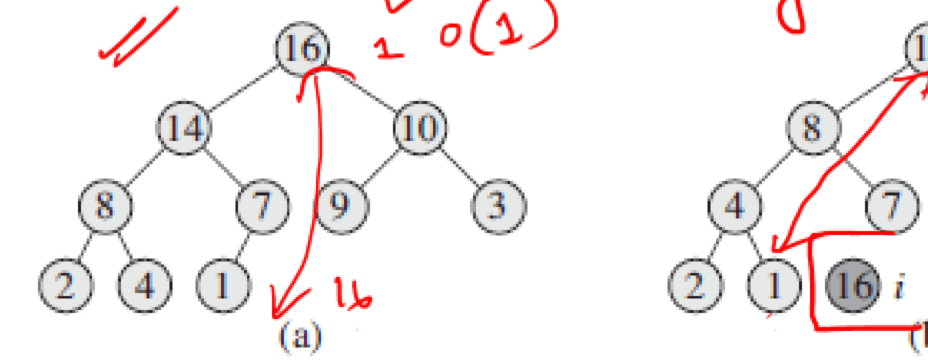
# Heap Sort-Example

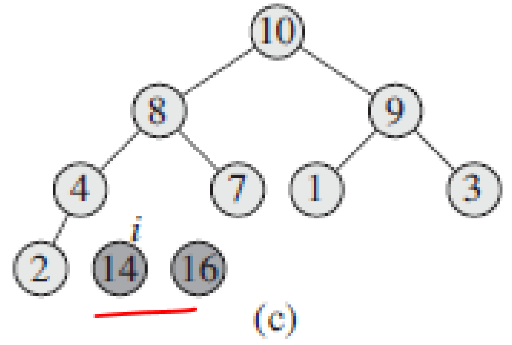
$O(n)$

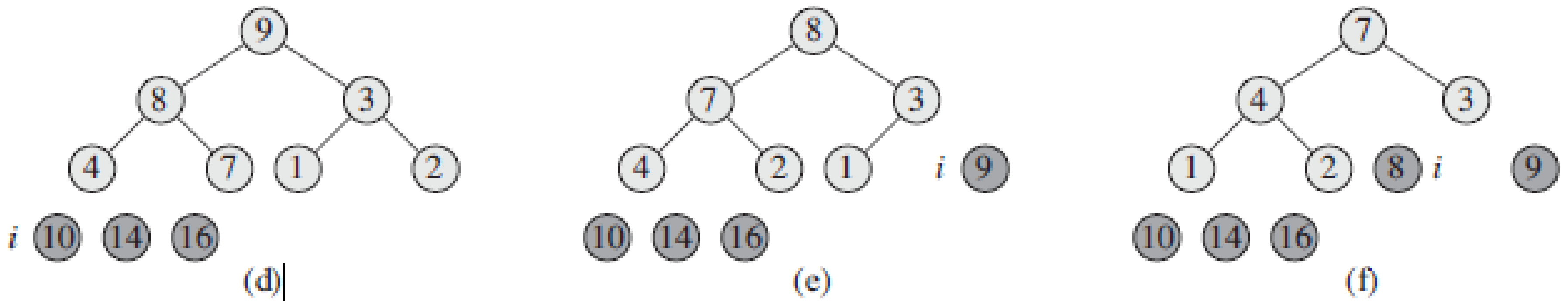
$\log n$

$1 \quad O(1)$

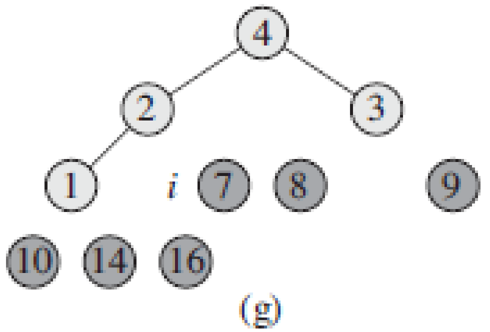
$\boxed{14\,|\,16}$



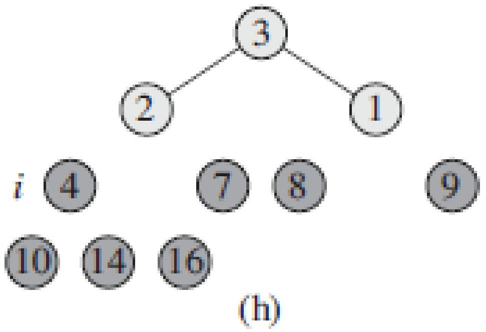




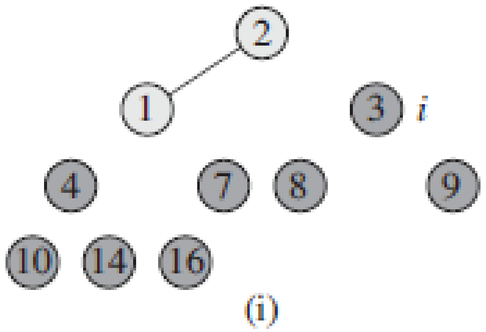
## Heap Sort-Example



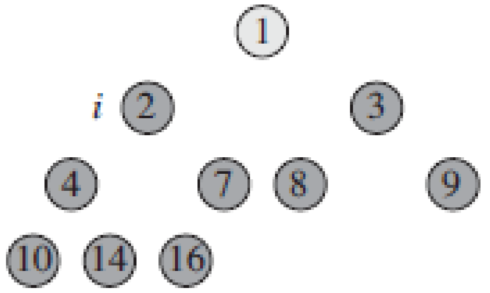
(g)



(h)



(i)



(j)

| A | 1 | 2 | 3 | 4 | 7 | 8 | 9 | 10 | 14 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|

(k)

$O(n \log n)$

# Heap-Sort

- Consider a priority queue with ***n*** items implemented by means of a heap
  - the space used is ***O***(***n***)
  - methods insertItem and removeMin take ***O***(log ***n***) time
  - methods size, isEmpty, minKey, and minElement take time ***O***(1) time
- Using a **heap-based priority queue**, we can sort a sequence of ***n*** elements in ***O***(***n*** log ***n***) time
- The resulting algorithm is called **heap-sort**
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

# Heap-Algorithms

**Parent(A, i)**

```
// Input: A: an array representing a heap, i: an array index
// Output: The index in A of the parent of i
// Running Time: O(1)
if i == 1 return NULL
return floor(i/2)
```

## Heap-Algorithms

**Left(A, i)**

```
// Input: A: an array representing a heap, i: an array index
// Output: The index in A of the left child of i
// Running Time: O(1)
if $2 * i \leq$ heap-size[A]
return $2 * i$
else return NULL
```

# Heap-Algorithms

```
Right(A, i)
// Input: A: an array representing a heap, i: an array index
// Output: The index in A of the right child of i
// Running Time: O(1)
if $2 * i + 1 \le heap\text{-}size[A]$
return $2 * i + 1$
else return NULL
```

## Heap-Algorithms

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

// Input: A: an array representing a heap, i: an array index, key: a new key greater than A[i]

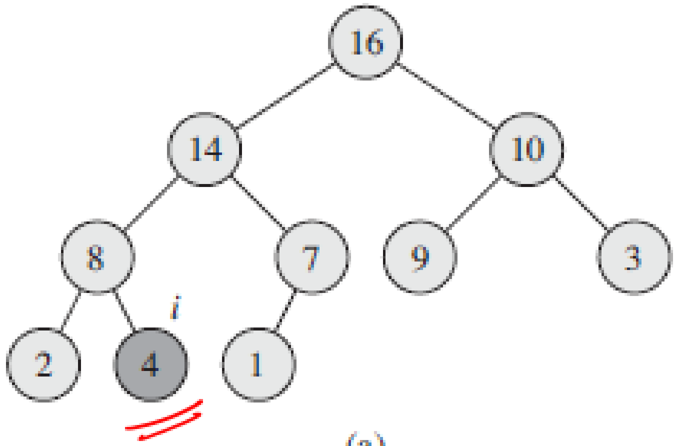
// Output: A still representing a heap where the key of A[i] was increased to key

// Running Time: O(log n) where n =heap-size[A]

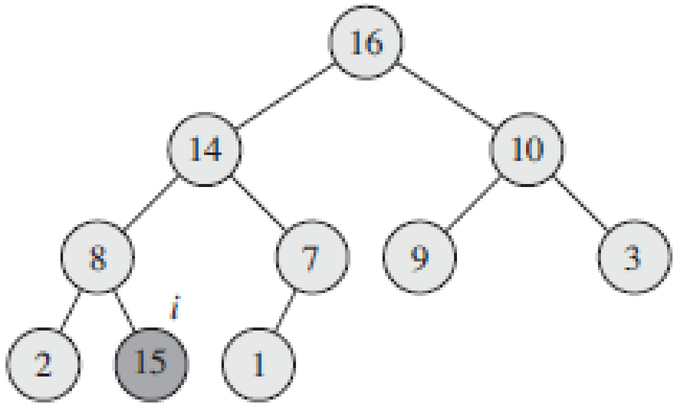
```
if key < A[i]
        error("New key must be larger than current key")
A[i] ← key
while i > 1 and A[Parent(i)] < A[i]
        exchange A[i] and A[Parent(i)]
        i ← Parent(i)
```

# Heap-Increase-Key

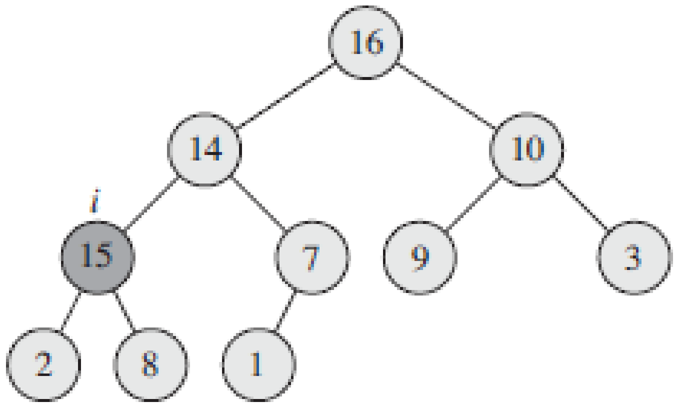
*increase 4 to 15*



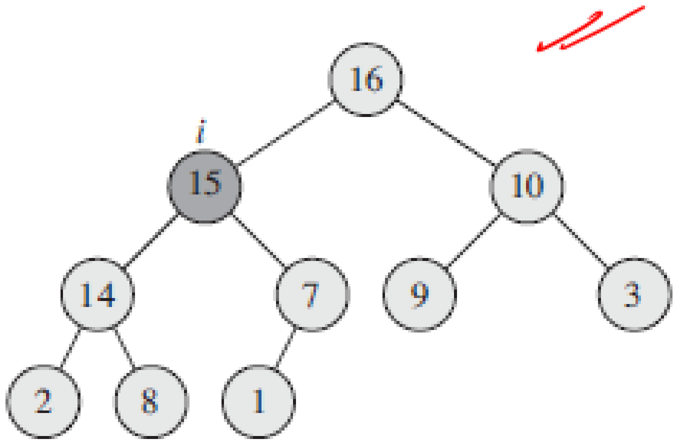
(a)



(b)



(c)



(d)

# Heap-Algorithms

```
**Heap-Extract-Max(A)**
// Input: A: an array representing a heap
// Output: The maximum element of A and A as a heap with
this element removed
// Running Time: O(log n) where n =heap-size[A]
max ← A[1]
A[1] ← A[heap-size[A]]
heap-size[A] ←heap-size[A] – 1
Max-Heapify(A, 1)
return max
```

# Heap-Algorithms

```
**Max-Heap-Insert(A, key)**
// Input: A: an array representing a heap, key: a key to insert
// Output: A modified to include key
// Running Time: O(log n) where n =heap-size[A]
heap-size[A] ← heap-size[A] + 1
A[heap-size[A]] ← −∞
Heap-Increase-Key(A[heap-size[A]], key)
```

## HW

**Max-Heapify and Build-Max-Heap**

Demonstrate how Build-Max-Heap turns it into a heap

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 7 | 9 | 3 | 5 | 6 | 8 | 13 | 32 | 12 | 22 |

## HW

**Heap-Increase-Key**

Show what happens when you use HeapIncrease-Key to increase key 5 to 33

## HW

**Heap-Sort**

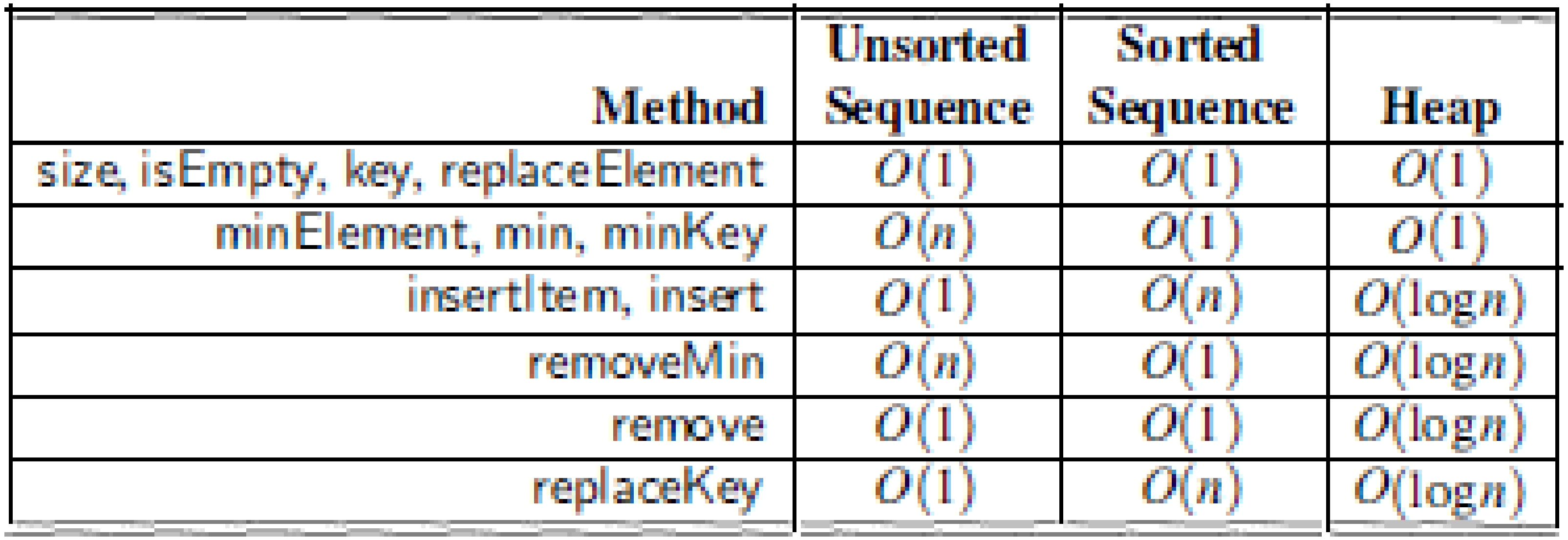
Given the heap show how you use it to sort .Explain why the runtime of this algorithm is O(n log n)

## HW

**Group 4:**

Priority Queue implementation using an Unordered Sequence (implemented using an array/linked list)

# Comparison of Different Priority Queue Implementations



| Method | Unsorted Sequence | Sorted Sequence | Heap |
|---|---|---|---|
| size, isEmpty, key, replaceElement | $O(1)$ | $O(1)$ | $O(1)$ |
| minElement, min, minKey | $O(n)$ | $O(1)$ | $O(1)$ |
| insertItem, insert | $O(1)$ | $O(n)$ | $O(\log n)$ |
| removeMin | $O(n)$ | $O(1)$ | $O(\log n)$ |
| remove | $O(1)$ | $O(1)$ | $O(\log n)$ |
| replaceKey | $O(1)$ | $O(n)$ | $O(\log n)$ |