The Heap Data Structure

Instructor: Ashok Singh Sairam

Lecture Plan

- Heap
  - Representation, Types, ….
- Operations on Heaps
  - Max-heapify
  - Build Heap
  - Heapsort
  - Priority Queue

Review

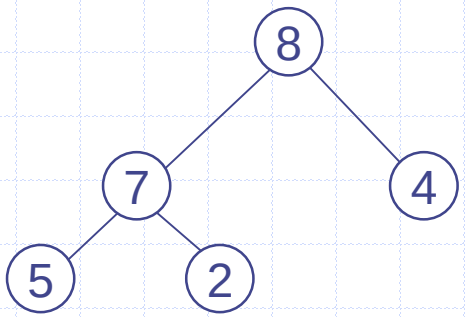
- Show by induction that any binary tree of height h has at most 2^h leaves
-

Heap

- A **heap** is a nearly complete binary tree with the following two properties:
  - **Structural property:** all levels are full, except possibly the last one, which is filled from left to right

    A heap is a binary tree that is filled in order
  - **Order (heap) property:** for any node x

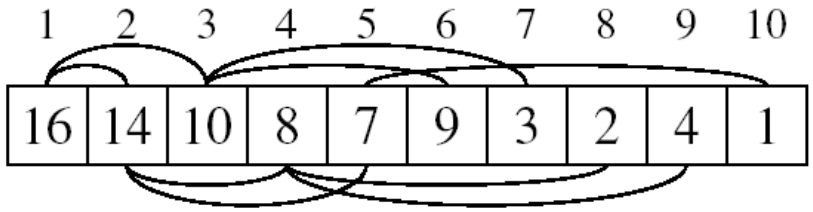
    $Parent(x) \geq x$ or $Parent(x) \leq x$



From the heap property, it follows that:
"The root is either the maximum or the minimum element of the heap!"

Array Representation of Heaps

- A heap can be stored as an array *A*.
  - Root of tree is $A[1]$
  - Left child of $A[i] = A[2i]$
  - Right child of $A[i] = A[2i + 1]$
  - Parent of $A[i] = A[\lfloor i/2 \rfloor]$
  - Heapsize[A] ≤ length[A]
- The elements in the subarray $A[(\lfloor n/2 \rfloor+1) .. n]$ are leaves



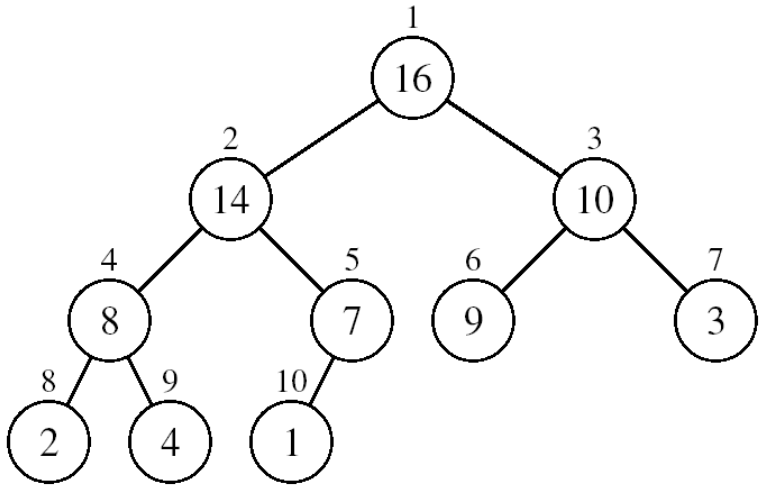


Fig: Max heap

Heap Types

- **Max-heaps** (largest element at root), have the *max-heap property:*
  - for all nodes i, excluding the root:

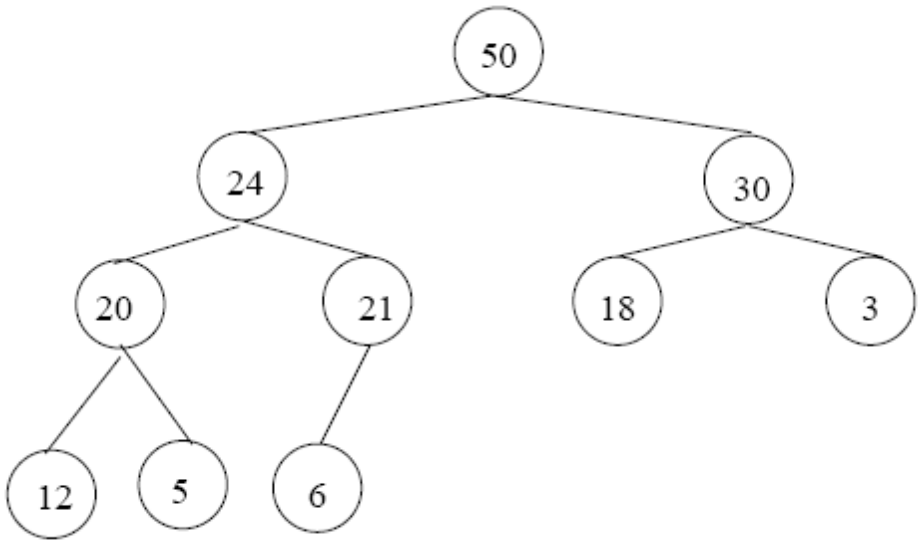
$$A[PARENT(i)] \geq A[i]$$

- **Min-heaps** (smallest element at root), have the *min-heap property:*
  - for all nodes i, excluding the root:

$$A[PARENT(i)] \leq A[i]$$

Adding/Deleting Nodes

- New nodes are always inserted at the bottom level (left to right)
- Nodes are removed from the bottom level (right to left)

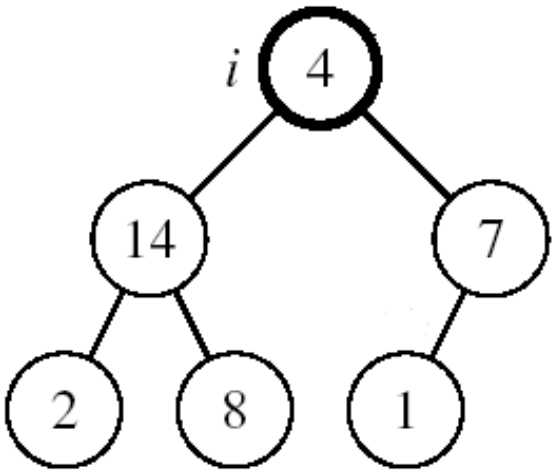


Operations on Heaps

- Maintain/Restore the max-heap property
  - MAX-HEAPIFY
- Create a max-heap from an unordered array
  - BUILD-MAX-HEAP
- Sort an array in place
  - HEAPSORT
- Priority queues

Maintaining the Heap Property

- Correct a single violation of the heap property in a subtree's root A[i]
  - Left and Right subtrees of *i* are max-heaps
- To eliminate the violation:
  - Exchange with larger child
  - Move down the tree
  - Continue until node is not smaller than children



Procedure Max-Heapify

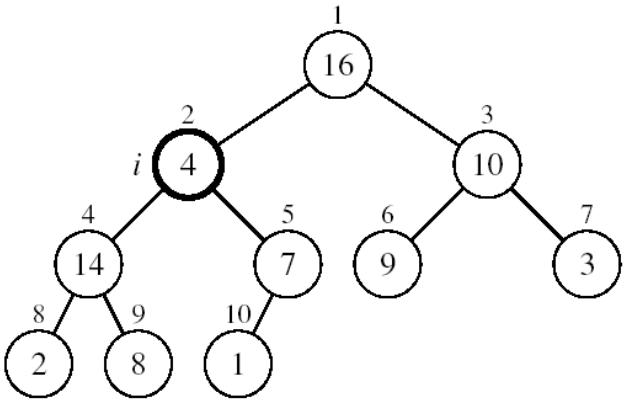
Assumption:

i. Array representation

ii. Assumes that A[i] violates max-heap, but the left and right subtree are max heaps

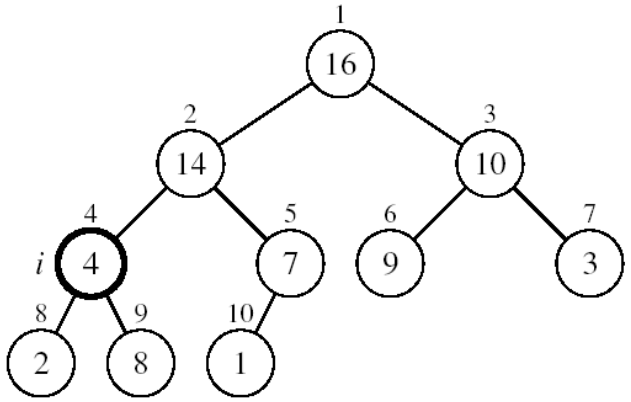
```
MAX-HEAPIFY(A, i)
1   l = LEFT(i)
2   r = RIGHT(i)
3   if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5   else largest = i
6   if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8   if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

Example: Max-Heapify(A,2)



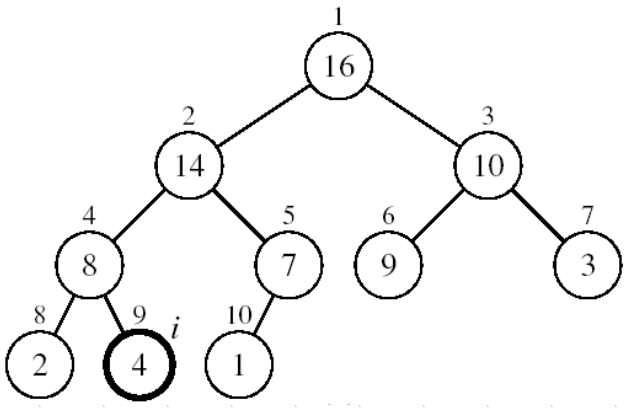
A[2] violates the heap property

$A[2] \leftrightarrow A[4]$



A[4] violates the heap property

$A[4] \leftrightarrow A[9]$



Heap property restored

Running Time of Max-Heapify

- Intuitively:
  - It traces a path from the root to a leaf (longest path length: h)
  - At each level, it makes exactly 2 comparisons
  - Total number of comparisons is 2h
  - Running time is O(h) or $O(lgn)$
- Running time of MAX-HEAPIFY is $O(lgn)$
- Can be written in terms of the height of the heap, as being $O(h)$
  - Since the height of the heap is $\lfloor lgn \rfloor$

Exercise

6.2-1

Using Figure 6.2 as a model, illustrate the operation of MAX-HEAPIFY$(A, 3)$ on the array $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.

6.2-2

Starting with the procedure MAX-HEAPIFY, write pseudocode for the procedure MIN-HEAPIFY(A, i), which performs the corresponding manipulation on a min-heap. How does the running time of MIN-HEAPIFY compare to that of MAX-HEAPIFY?

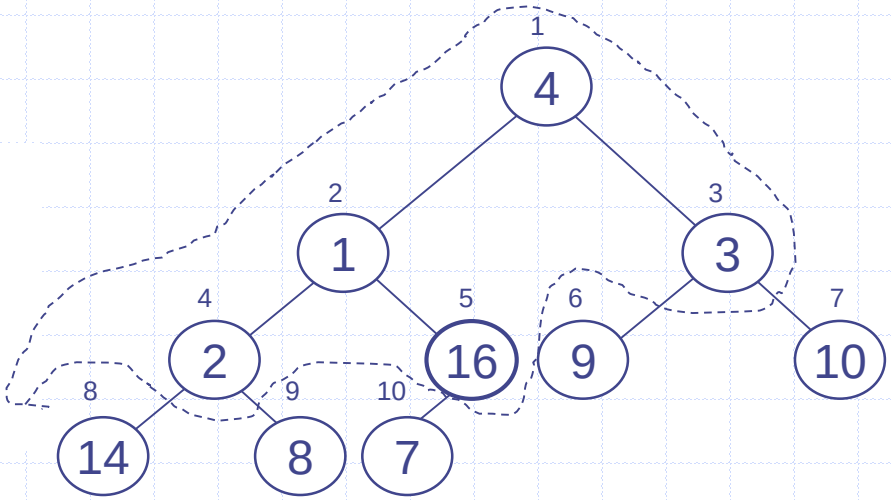
6.2-3

What is the effect of calling MAX-HEAPIFY(A, i) when the element $A[i]$ is larger than its children?

Building a Heap

- Convert an array $A[1 \ldots n]$ into a max-heap ($n = len[A]$)
- The elements in the subarray $A[(\lfloor n/2 \rfloor+1) .. n]$ are leaves
  - Each is a 1-element heap
- Apply MAX-HEAPIFY on elements between 1 and $\lfloor n/2 \rfloor$

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



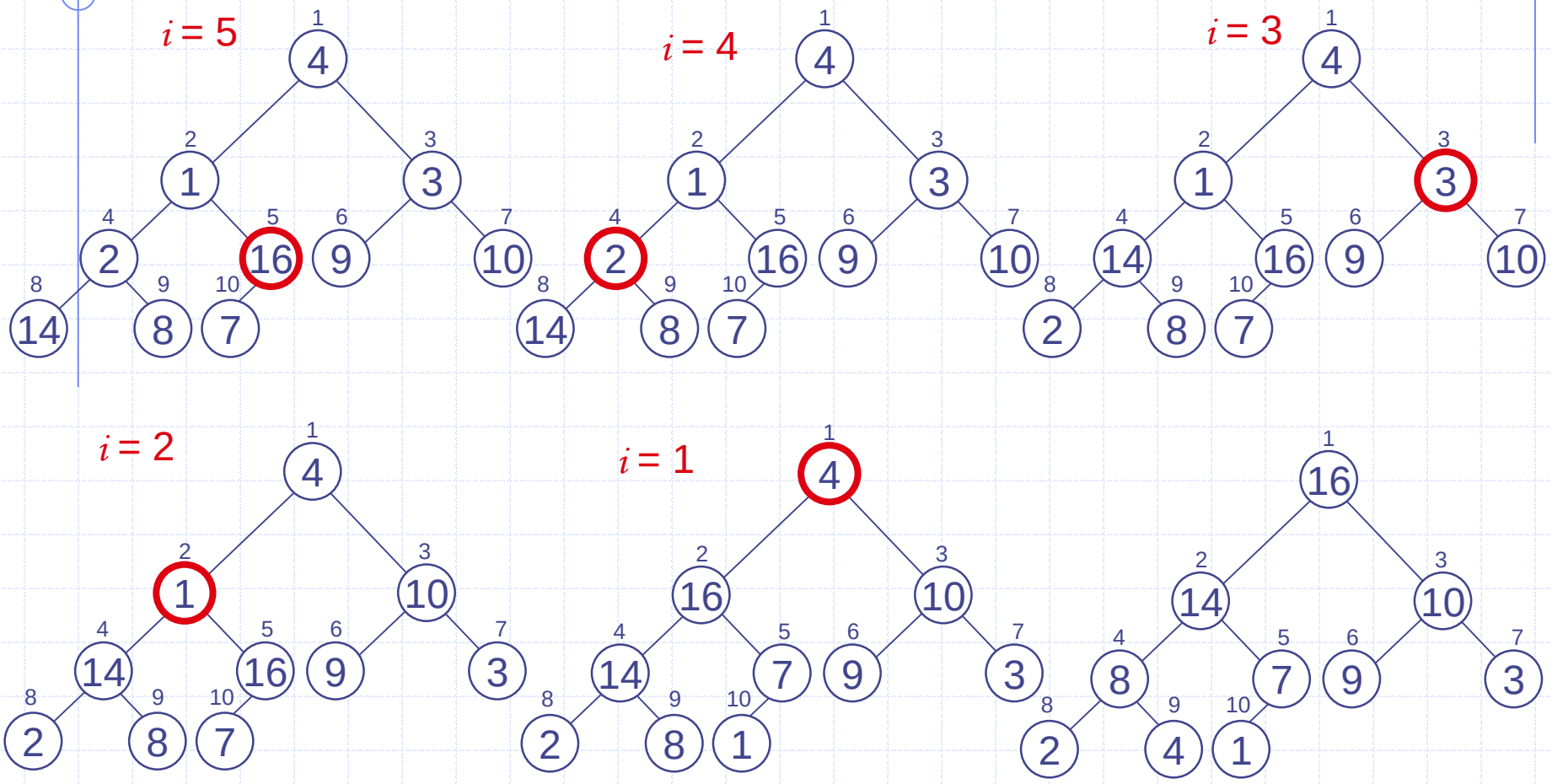
A:

4	1	3	2	16	9	10	14	8	7

Example:

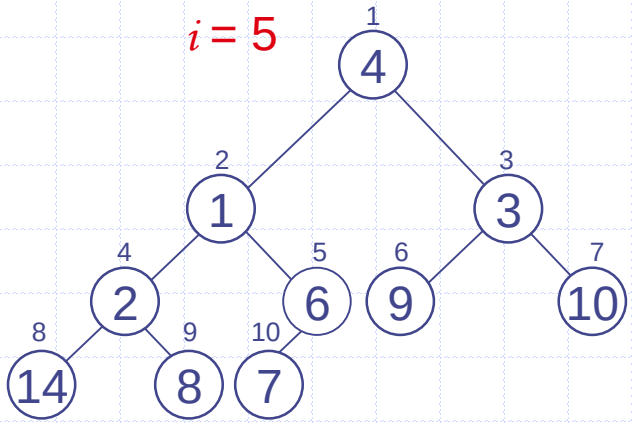
A

4	1	3	2	16	9	10	14	8	7



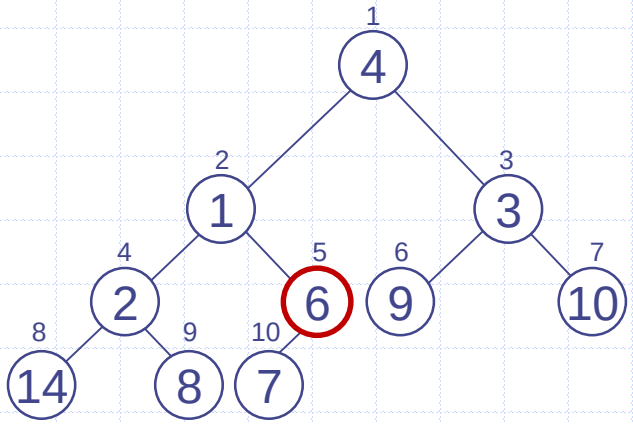
Correctness: Build-Max-Heap

- Loop invariant: Nodes *i+1* to *n* is the root of a max-heap
- Initialization: Before the loop starts $\mathrm{i} = \lfloor n/2 \rfloor$
  - Nodes $\lfloor n/2 \rfloor$+1, $\lfloor n/2 \rfloor$+2, .. , n is a leaf and thus is the root of a trivial max-heap



Correctness: Build-Max-Heap

- Maintenance:
  - For loop starts from i=$\lfloor n/2 \rfloor$ and move backwards to 1
  - Children of i are numbered higher than i and by loop invariant they are already root of max-heap
  - Max-heapify requires both the right and left child of i to be roots of max heap, in order to max i a max heap root
    - The above condition is true as i decrements in the for loop



Correctness: Build-Max-Heap

- Termination: At termination i =0. By loop invariant, each node 1,2,..,n is the root of a max-heap

Running Time of Build-Max-Heap

- A simple upper bound

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

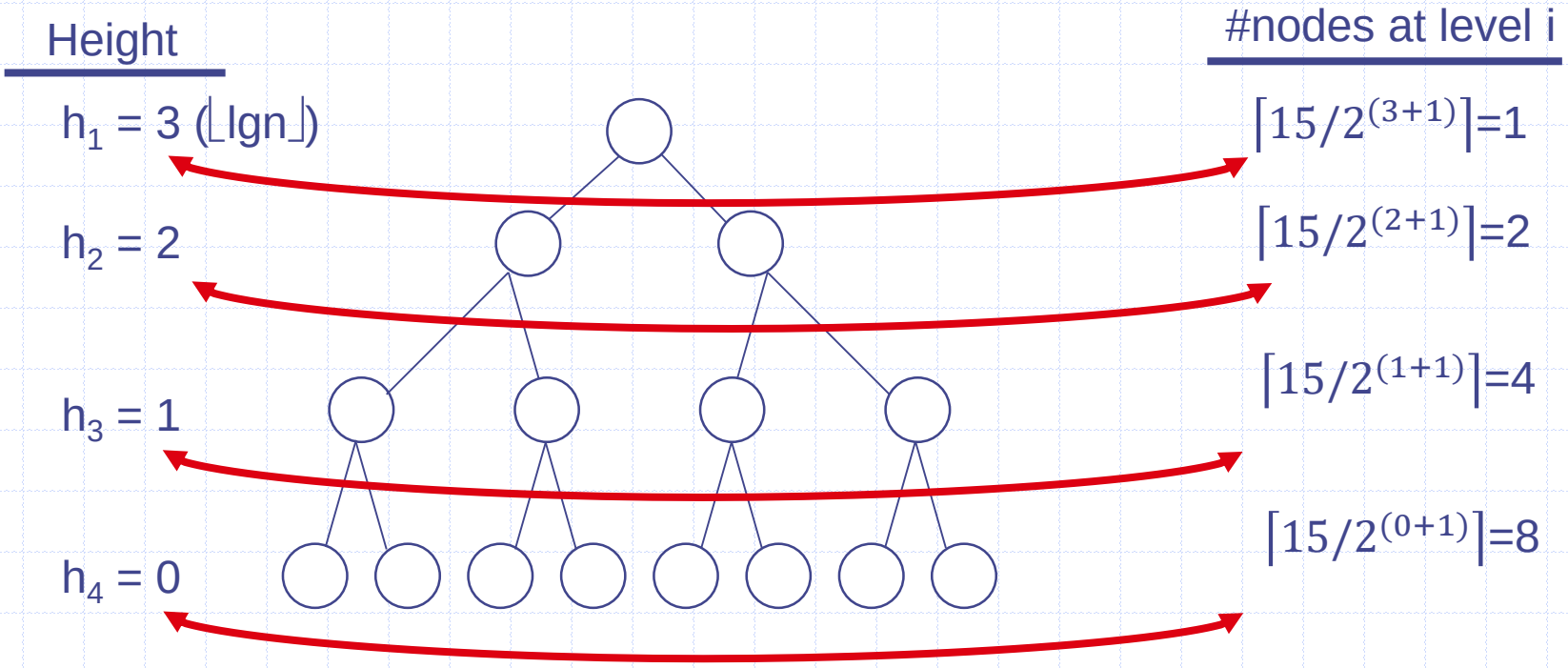
O(lgn) } O(n)

$\Rightarrow$ Running time: $O(n \lg n)$

- This is not an asymptotically tight upper bound

Running Time of Build-Max-Heap

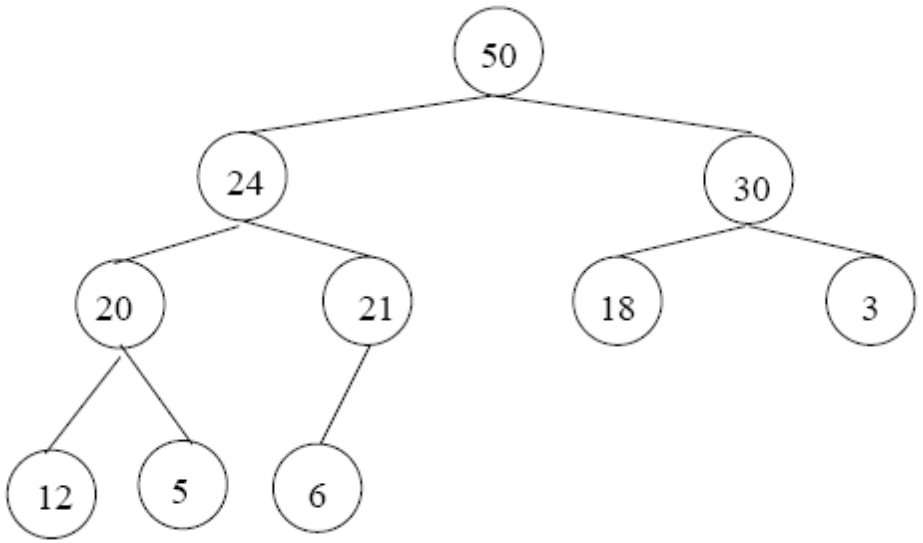
- Consider two properties. An n-element heap
  - has height $\lfloor \lg n \rfloor$
  - has at most $\lceil n/2^{h+1} \rceil$ nodes at height h



Deleting a Node

Two possibilities

- Last node - Trivial
- Any node(i) – Replace A[i] with last node; max-heapify(A,i)



Acknowledgement

- Dr George Bebis, Foundation Professor, Dept of Computer Science and Engineering, University of Nevada Reno