EL9343

Data Structure and Algorithm

Lecture 4: Introduction to sorting II: HeapSort, Quicksort

Instructor: Yong Liu

Last Lecture

- Divide-and-conquer algorithms
  - maximum subarray

- Insertion sort
  - Design approach: Incremental
  - Sorts in place: Yes
  - Best case: $\Theta(n)$
  - Worst case: $\Theta(n^2)$

Introduction to Sorting: Merge Sort

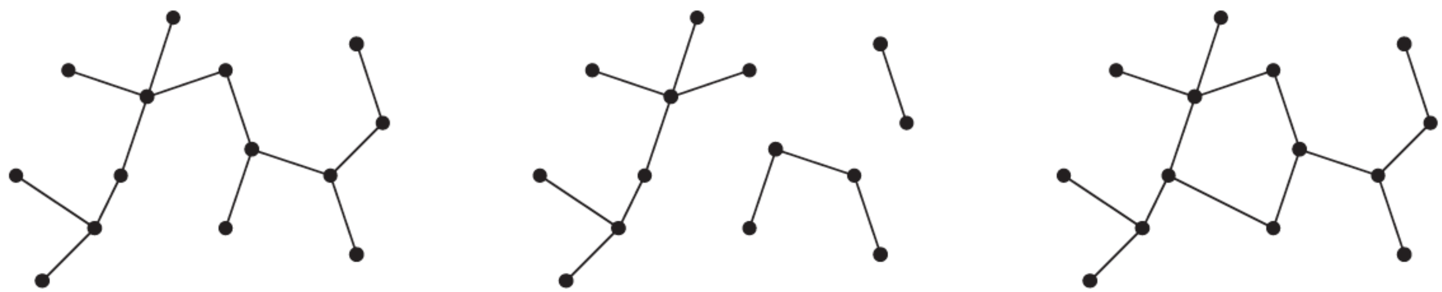
- Bubble Sort
  - Design approach: Incremental
  - Sorts in place: Yes
  - Running time: $\Theta(n^2)$
- Merge Sort
  - Design approach: divide and conquer
  - Sorts in place: No
  - Running time: $\Theta(n\log n)$

Introduction to Sorting: HeapSort

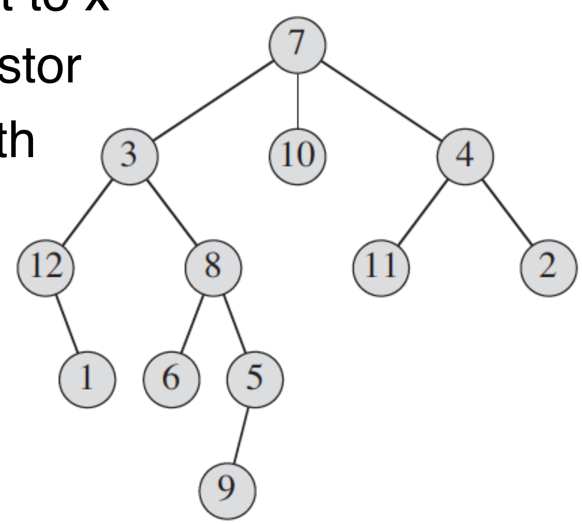
- So far we've talked about several algorithms to sort an array of numbers
  - What is the advantage of merge sort?
    - Answer: O(n lg n) worst-case running time
  - What is the advantage of insertion sort?
    - Answer: sorts in place
    - Also: When array "nearly sorted", runs fast in practice
- Next on the agenda: *Heapsort*
  - Combines advantages of both previous algorithms

Data Structure: Tree

- **(Free) Tree**: connected, acyclic, undirected graph
- **Forest:** acyclic, undirected graph, possibly disconnected

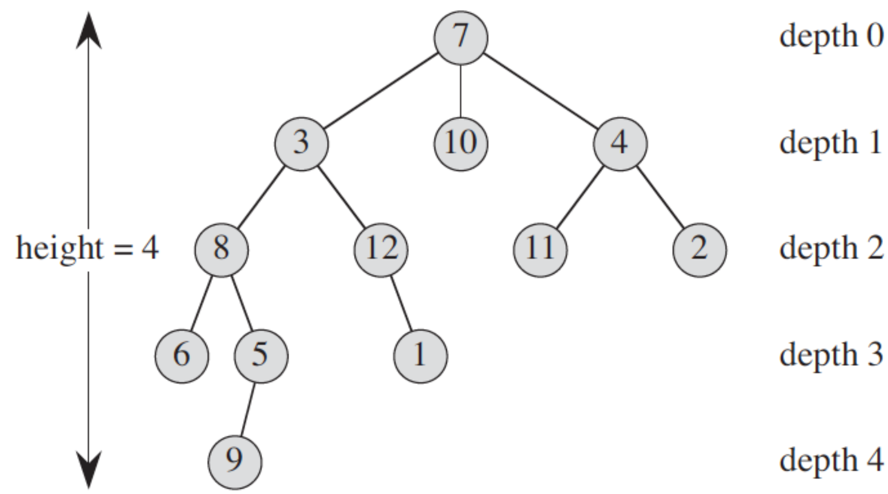


- **Rooted Tree**: a free tree with special **root** node
  - **Ancestor** of node x: any node on the path from root to x
  - **Descendant** of node x: any node with x as its ancestor
  - **Parent** of node x: node immediately before x on path from root
  - **Child** of node x: any node with x as its parent
  - **Siblings** of node x: nodes sharing parent with x
  - **Leaf/external node**: without child
  - **Internal node**: with at least one child



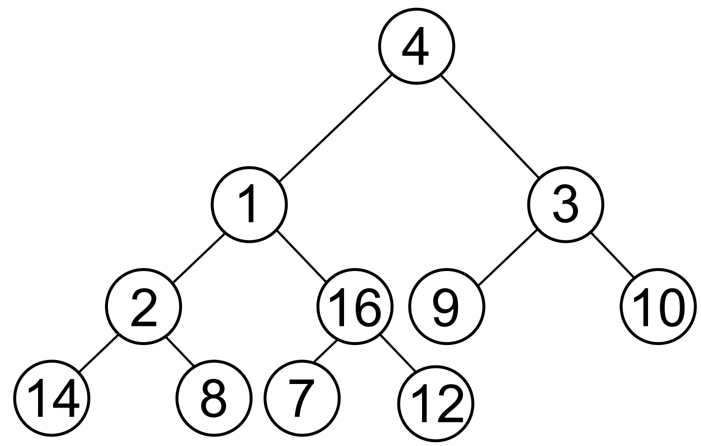
Data Structure: Tree

- **degree** of x: number of children
- **depth** of x: length of the simple path from root to x
- **level** of a tree: all nodes at the same depth
- **height** of x: length of the longes simple path from x downward to some leaf node
- **height of a tree**: height of root
- **Ordered Tree**: rooted tree in which children of each node are ordered



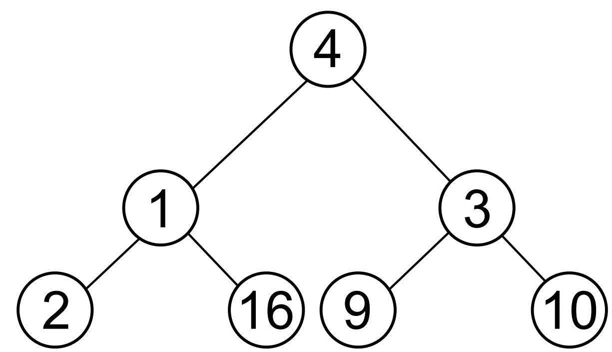
Special Types of Trees

- **Binary Tree (recursive def.)**
  - contains no node
  - root node, left subtree (binary), right subtree (binary)



Full binary tree

- **Full binary tree**
  - A binary tree in which each node is either a leaf or has degree exactly 2.



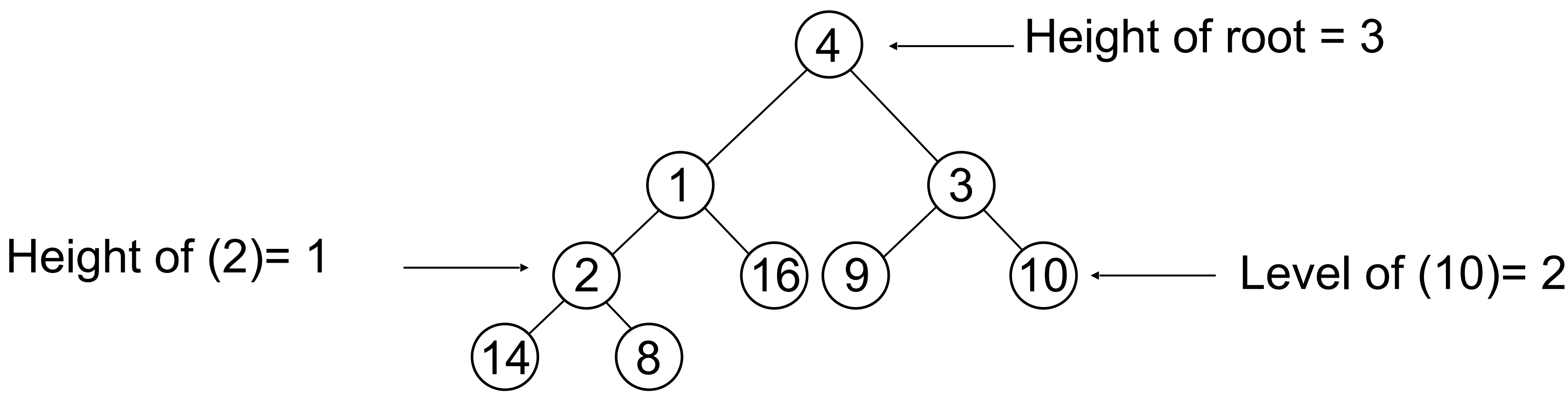
Complete binary tree

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

Useful Properties

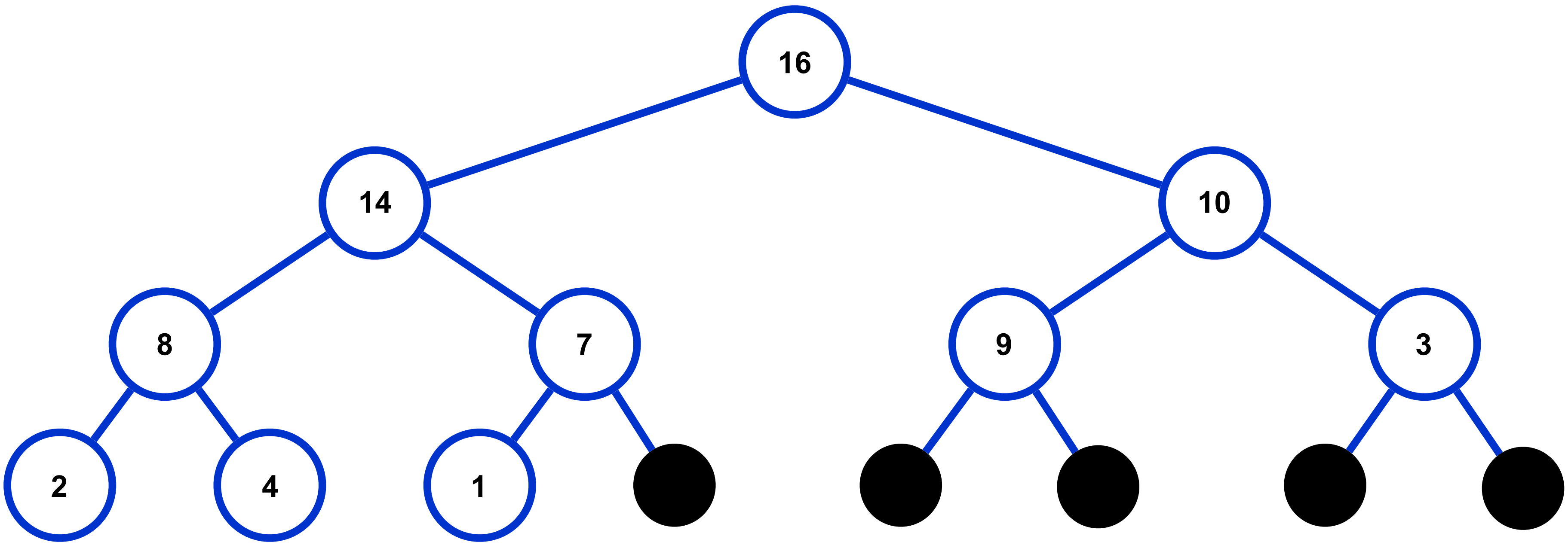
- There are at most 2^l nodes at level l of a binary tree
- A binary tree with depth d has at most 2^{d+1}-1 nodes
- A binary tree with n nodes has depth at least $\lfloor lgn \rfloor$

$$n \le \sum_{l=0}^{d} 2^l = \frac{2^{d+1}-1}{2-1} = 2^{d+1}-1$$



Data Structure: Heap

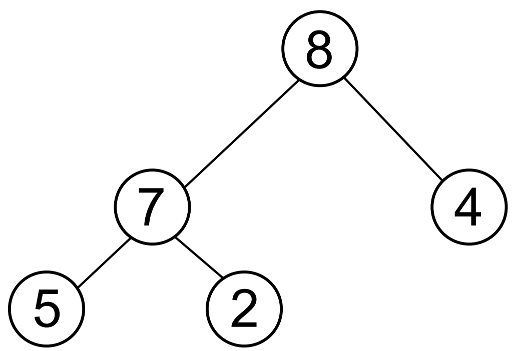
- A **heap** can be seen as a complete binary tree:



  - The book calls them “nearly complete” binary trees; can think of unfilled slots as null pointers

The Heap Data Structure

- A heap can be seen as a 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
  - **Order (heap) property:** for any node x: Parent(x) ≥ x



Heap

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

A heap is a binary tree that is filled in order

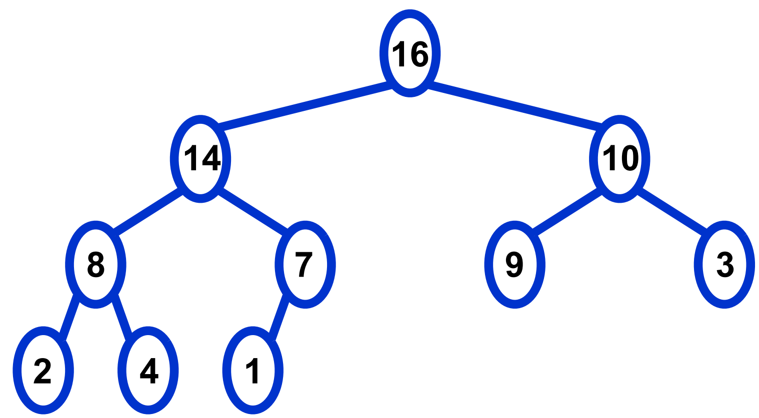
The Heap Data Structure

- In practice, heaps are usually implemented as arrays:

$A =$

16	14	10	8	7	9	3	2	4	1

$=$



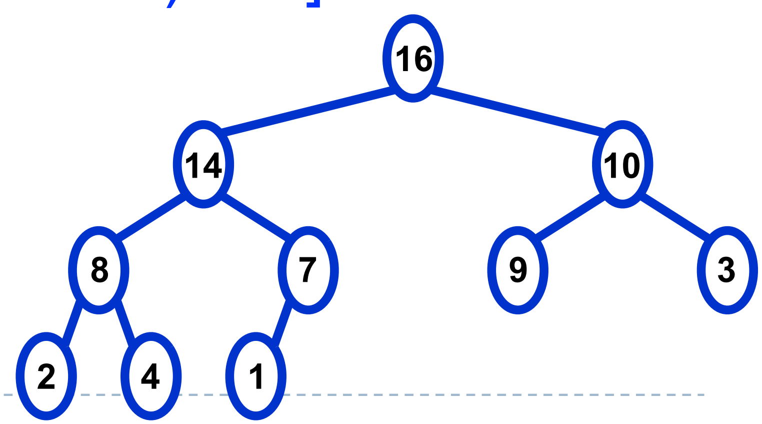
Array Representation of Heaps

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

A =

16	*14*	*10*	*8*	*7*	*9*	*3*	*2*	*4*	*1*

=



Heap Types

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

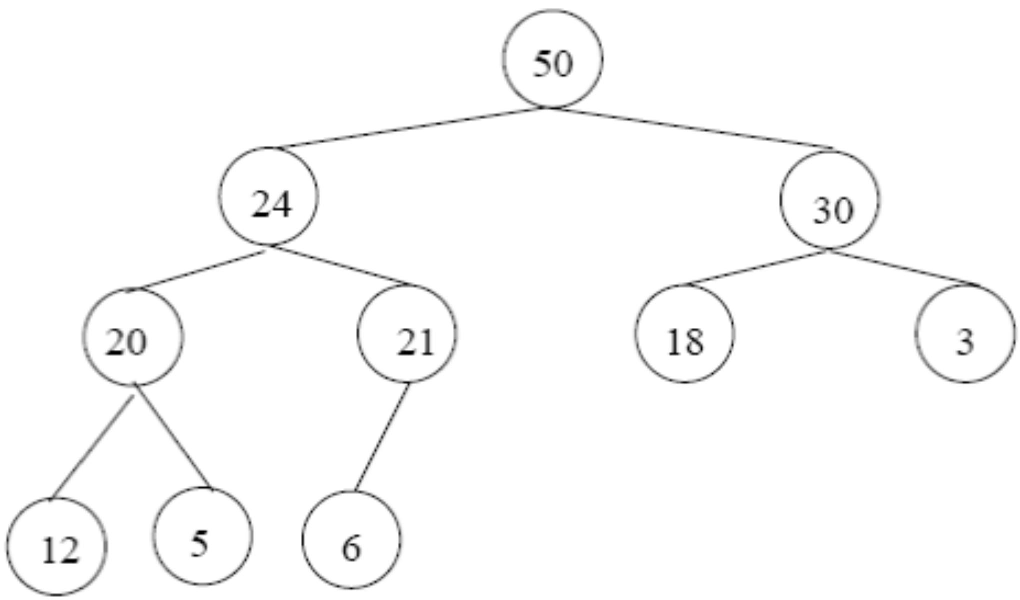
    $$A[\text{PARENT}(i)] \geq A[i]$$

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

    $$A[\text{PARENT}(i)] \leq A[i]$$

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)



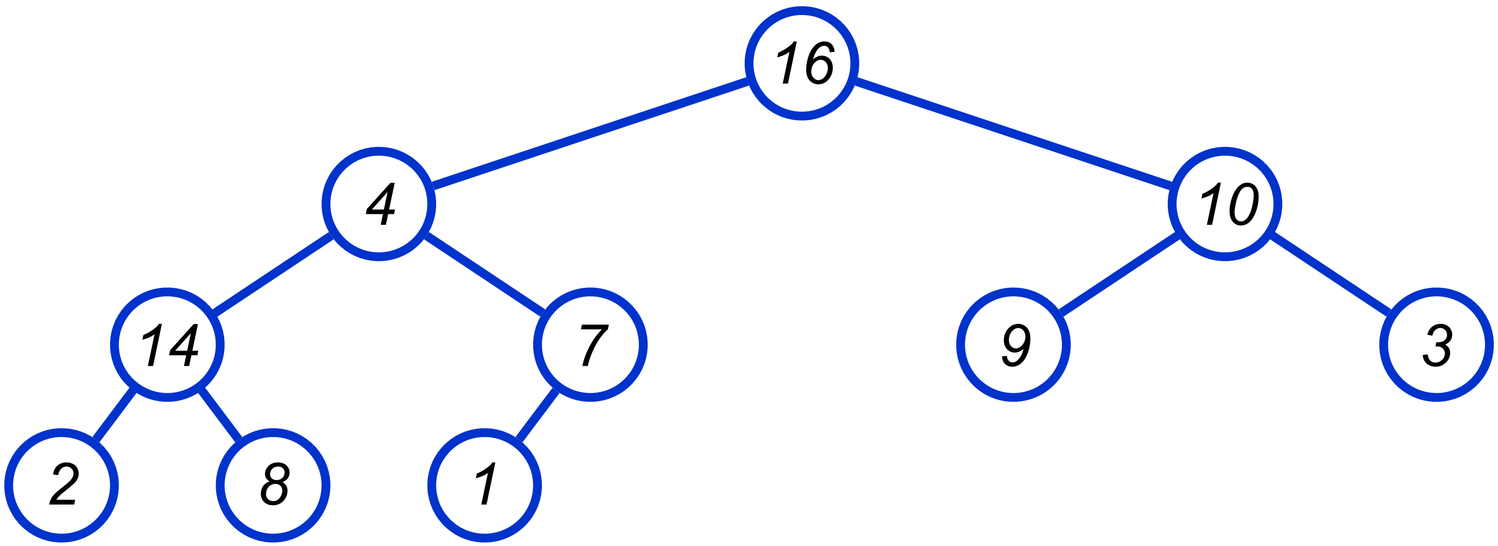
Operations on Heaps

- 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

Heap Operations: MAX-HEAPIFY

- Maintain the max-heap property: MAX-HEAPIFY
- Suppose a node is smaller than a child
  - 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

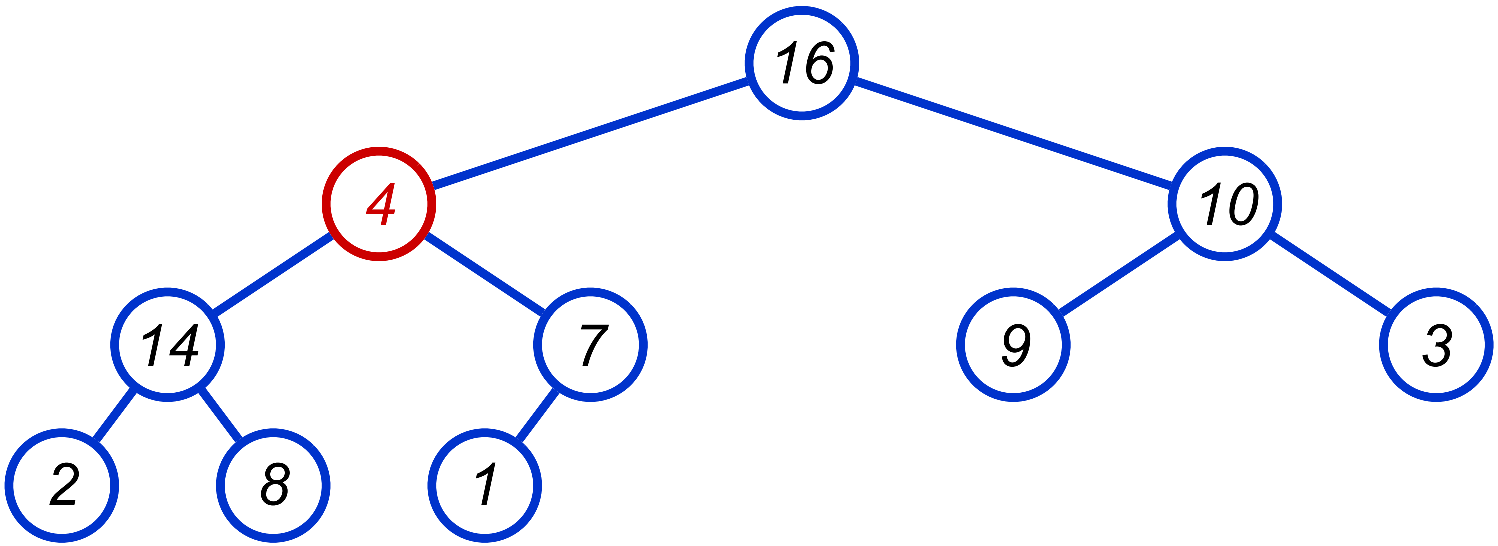
MAX-HEAPIFY Example



A =

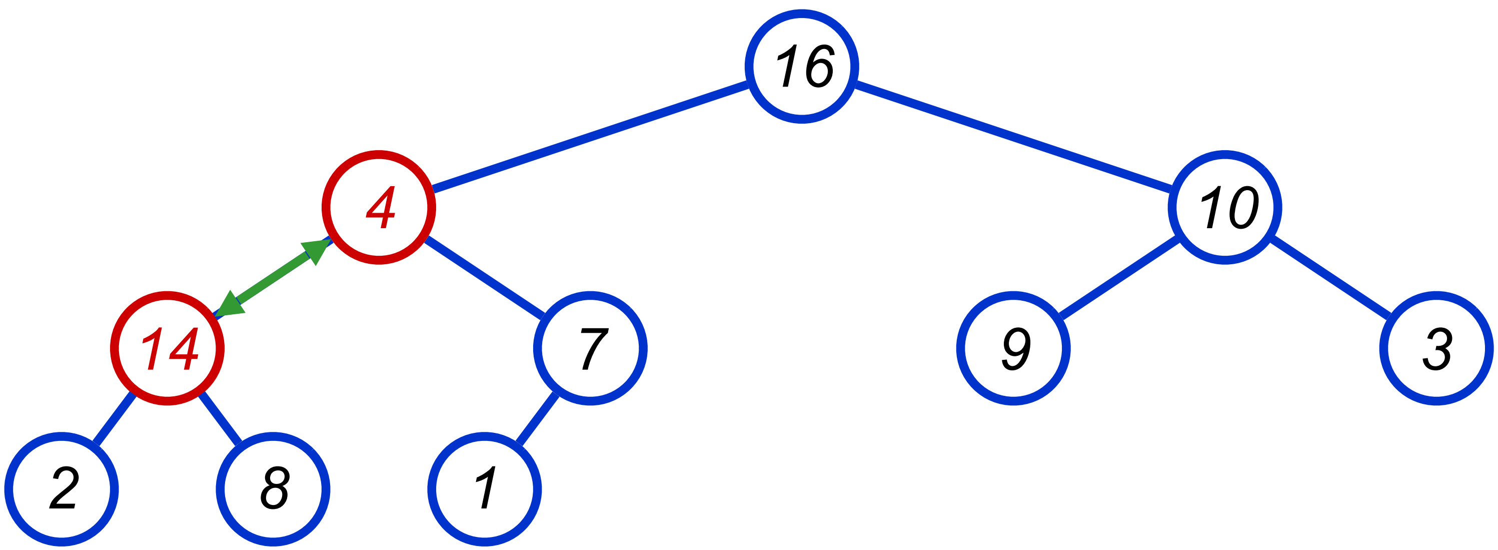
16	4	10	14	7	9	3	2	8	1

MAX-HEAPIFY Example



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

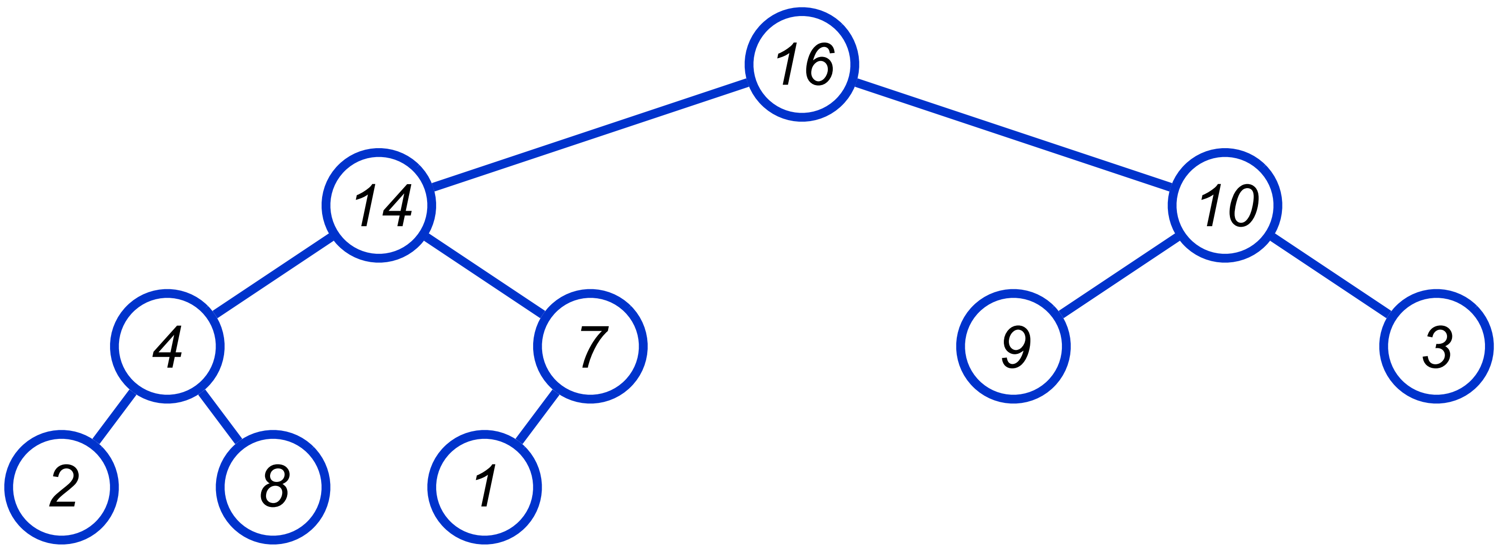
MAX-HEAPIFY Example



A =

16	4	10	14	7	9	3	2	8	1

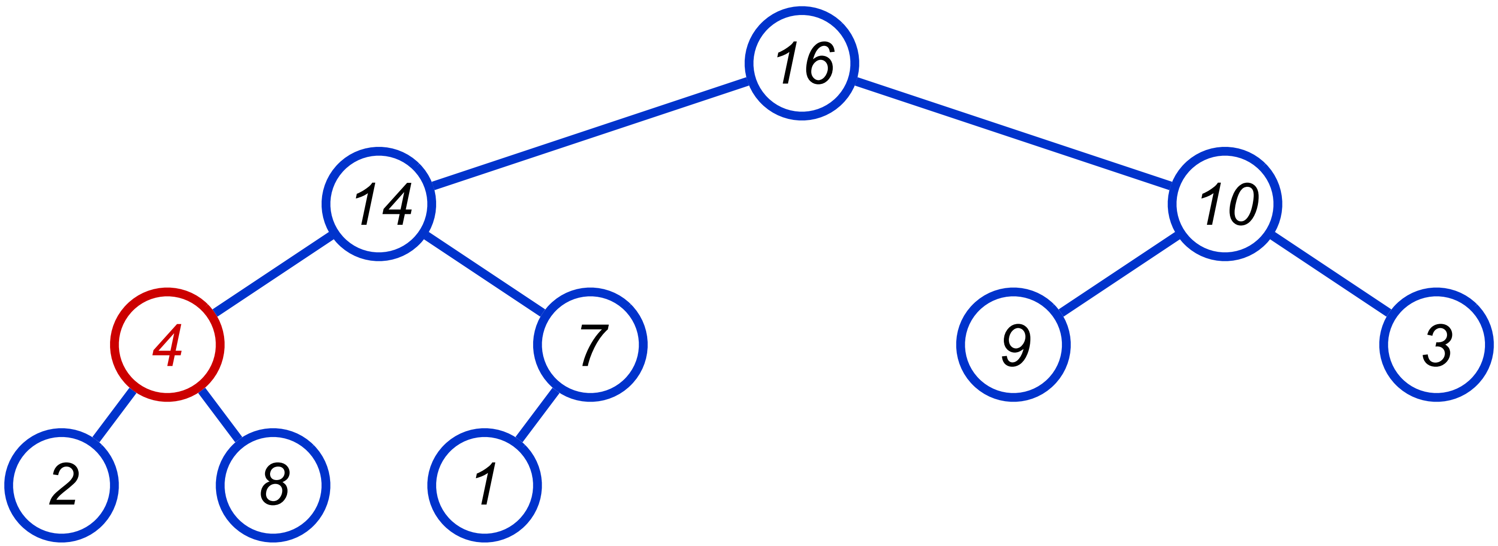
MAX-HEAPIFY Example



$A =$

16	14	10	4	7	9	3	2	8	1

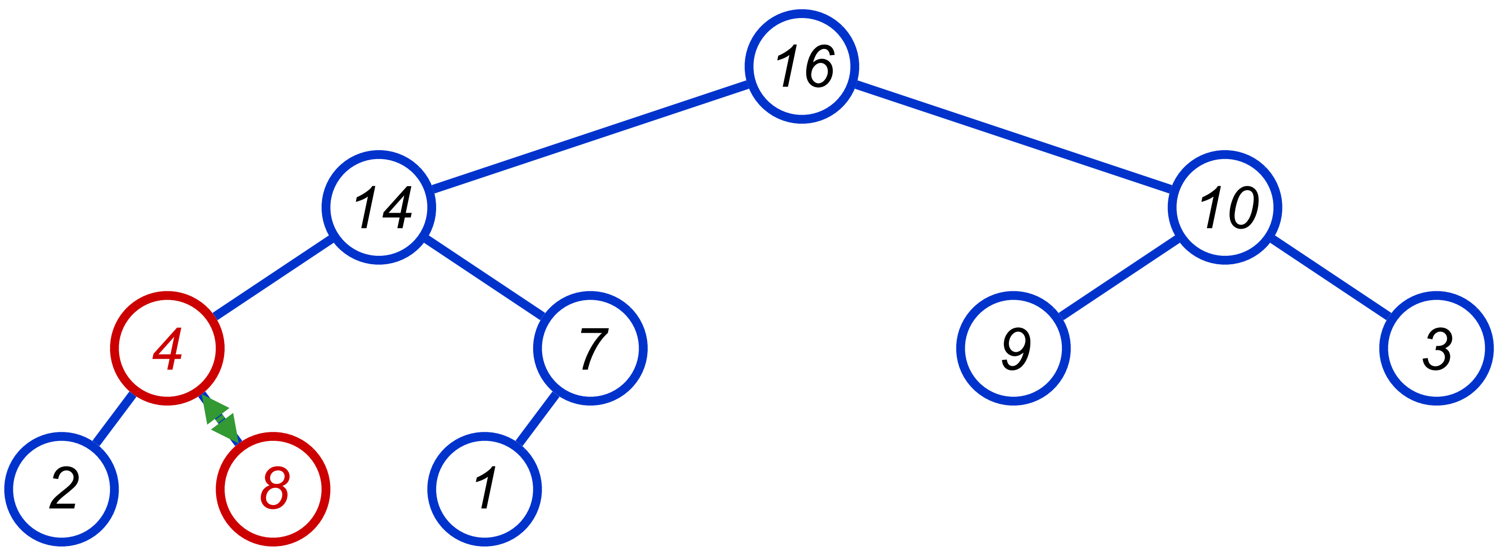
MAX-HEAPIFY Example



$A =$

16	14	10	4	7	9	3	2	8	1

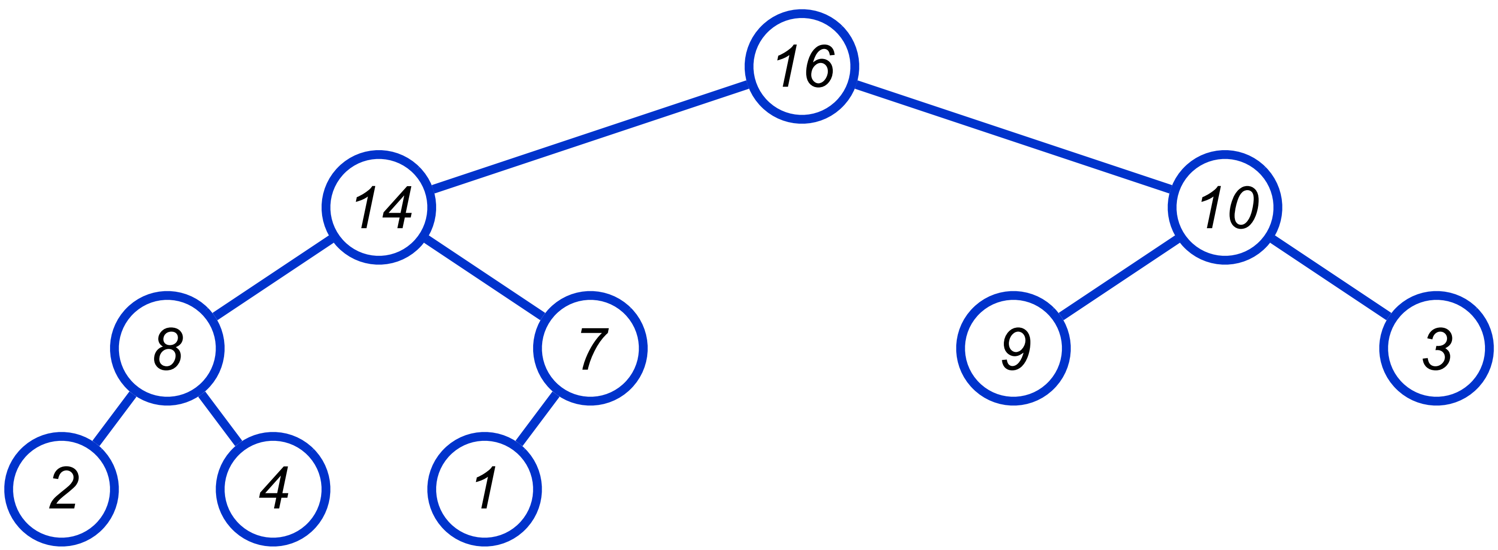
MAX-HEAPIFY Example



$A =$

16	14	10	4	7	9	3	2	8	1

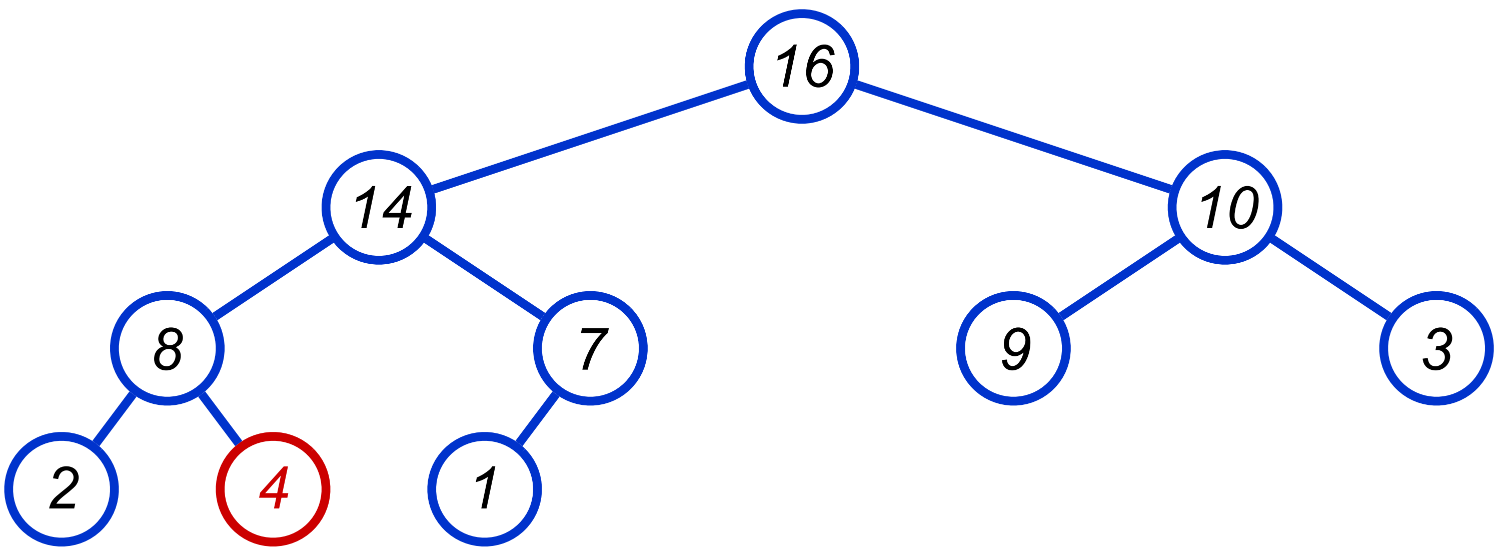
MAX-HEAPIFY Example



A =

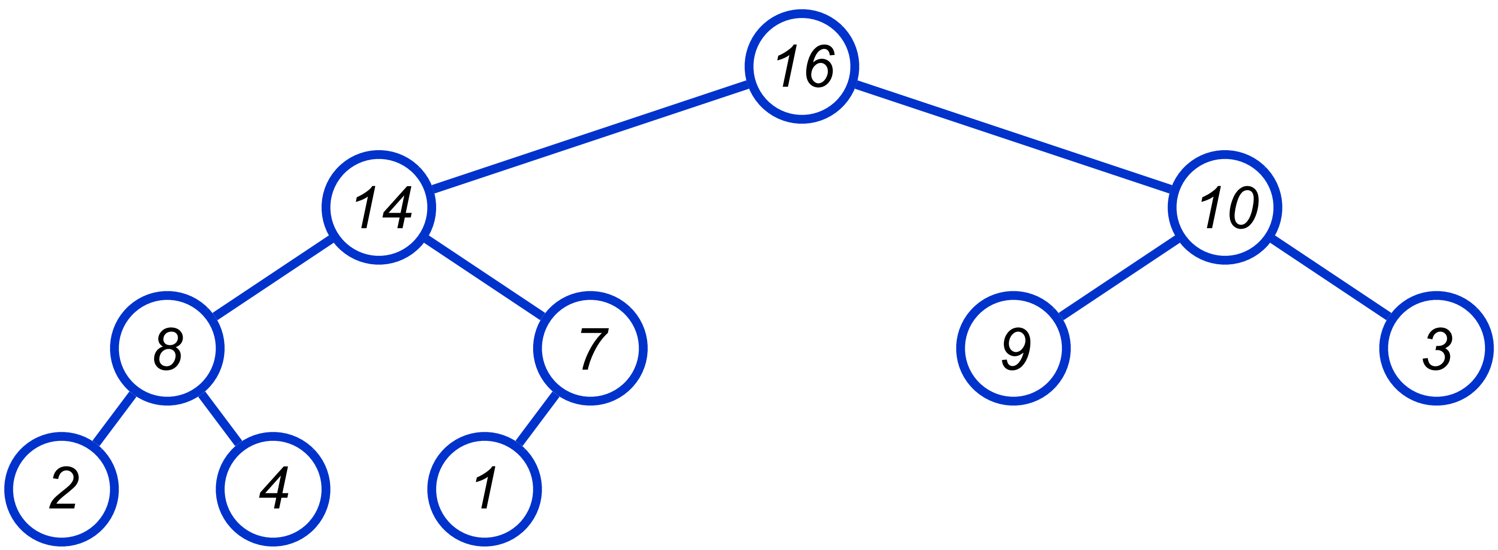
16	14	10	8	7	9	3	2	4	1

MAX-HEAPIFY Example



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

MAX-HEAPIFY Example



$A =$

16	14	10	8	7	9	3	2	4	1

Heap Operations: MAX-HEAPIFY

```
Max-Heapify(A, i)
{
      l = Left(i); r = Right(i);
      if (l <= heap_size(A) && A[l] > A[i])
            largest = l;
      else
            largest = i;
      if (r <= heap_size(A) && A[r] > A[largest])
            largest = r;
      if (largest != i)
            Swap(A, i, largest);
            Heapify(A, largest);
}
```

Assumptions:

- Left and Right subtrees of i are max-heaps
- A[i] may be smaller than its children

Analyzing MAX-HEAPIFY - Informal

- Intuitively
  - It trace a path from the root to a leaf (longest path length: h
  - At each level, it makes exactly 2 comparisons
  - Total number of comparison 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 $\lfloor \lg n \rfloor$

Analyzing MAX-HEAPIFY - Formal

- Fixing up relationships between *i*, *l*, and *r* takes $\Theta(1)$ time
- *If the heap at i has n elements, how many elements can the subtrees at l or r have?*
  - Answer: $2n/3$ (worst case: bottom row 1/2 full)
- So time taken by MAX-HEAPIFY is given by
  - $T(n) \le T(2n/3) + \Theta(1)$

Analyzing MAX-HEAPIFY - Formal

- So we have
  - $T(n) \leq T(2n/3) + \Theta(1)$
- By case 2 of the Master Theorem,
  - $T(n) = O(\lg n)$
- Thus, MAX-HEAPIFY takes logarithmic time

Building a Heap

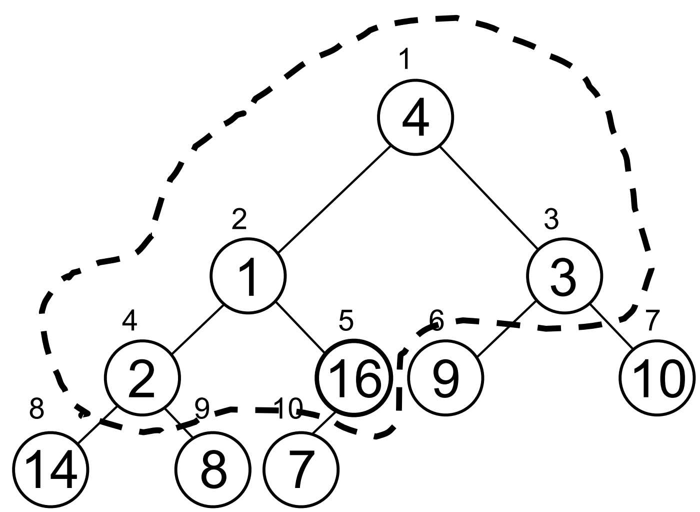
- We can build a heap in a bottom-up manner by running MAX-HEAPIFY on successive subarrays
  - Convert an array A[1 … n] into a max-heap (n = length[A])
  - The elements in the subarray A[(⌊n/2⌋ +1) .. n] are leaves
  - Apply MAX-HEAPIFY on elements between 1 and ⌊n/2⌋

Alg: <u>BUILD-MAX-HEAP(*A*)</u>

n = length[A]

for i ← ⌊n/2⌋ **downto** 1

  do MAX-HEAPIFY(*A*, i, n)



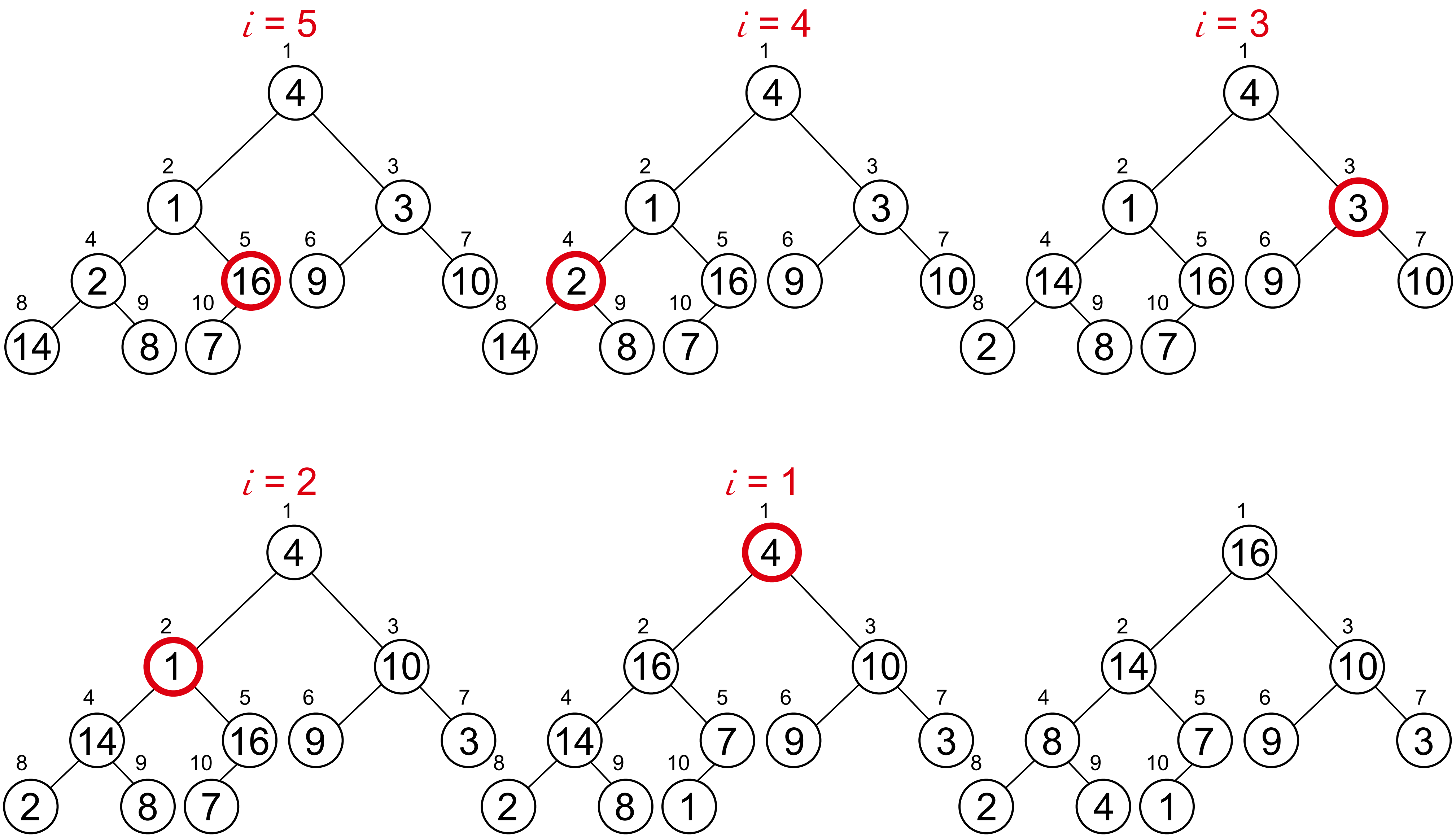
A:

4	1	3	2	16	9	10	14	8	7

Example:

A

4	1	3	2	16	9	10	14	8	7



Analyzing BUILD MAX HEAP

Alg: <u>BUILD-MAX-HEAP(*A*)</u>

1. n = length[A]
2. **for** i ← $\lfloor n/2 \rfloor$ **downto** 1
3. **do** MAX-HEAPIFY(*A*, i, n)

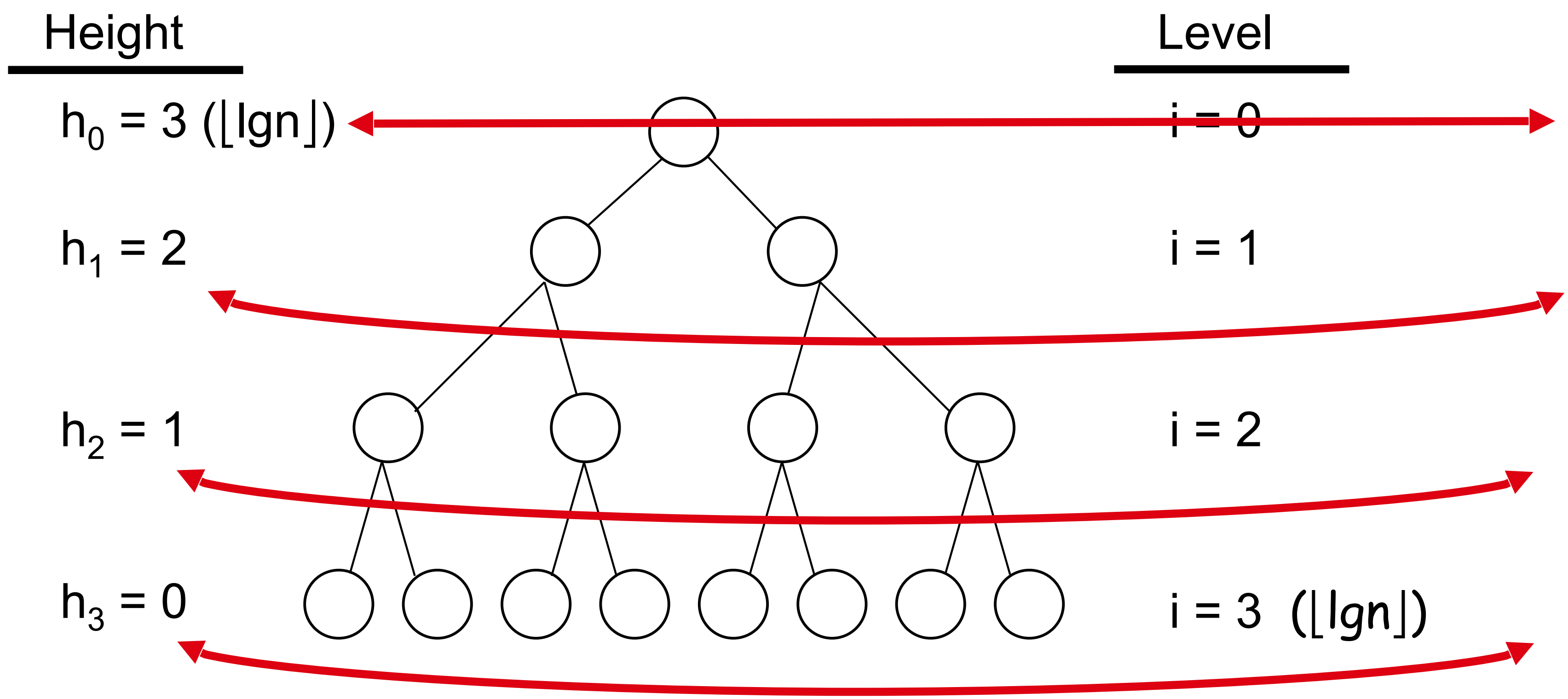
$$\left.\begin{array}{r} \\ O(\lg n) \end{array}\right\} O(n)$$

- Each call to MAX-HEAPIFY takes O(lg *n*) time
- There are O(*n*) such calls (specifically, $\lfloor n/2 \rfloor$)
- Thus the running time is O(*n* lg *n*)
  - *Is this a correct asymptotic upper bound? YES*
  - *Is this an asymptotically tight bound? NO*

- A tighter bound is O(*n*)

Running Time of BUILD MAX HEAP

- HEAPIFY takes O(h) ⇒ the cost of HEAPIFY on a node i is proportional to the height of the node i in the tree



$h_i = h - i$ height of the heap rooted at level i

$n_i = 2^i$ number of nodes at level i

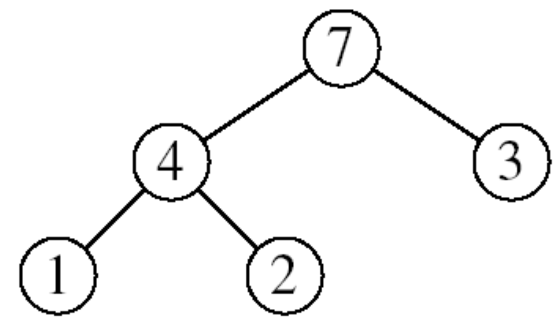
Running Time of BUILD MAX HEAP

$T(n) = \sum_{i=0}^{h} n_i h_i$	Cost of HEAPIFY at level i $*$ number of nodes at that level
$= \sum_{i=0}^{h} 2^i (h-i)$	Replace the values of n_i and h_i computed before
$= \sum_{i=0}^{h} \frac{h-i}{2^{h-i}} 2^h$	Multiply by 2^h both at the nominator and denominator and write 2^i as $\frac{1}{2^{-i}}$
$= 2^h \sum_{k=0}^{h} \frac{k}{2^k}$	Change variables: k = h - i
$\le n \sum_{k=0}^{\infty} \frac{k}{2^k}$	The sum above is smaller than the sum of all elements to ∞ and h = lgn
$= O(n)$	The sum above is 2

Running time of BUILD-MAX-HEAP: T(n) = O(n)

Heapsort

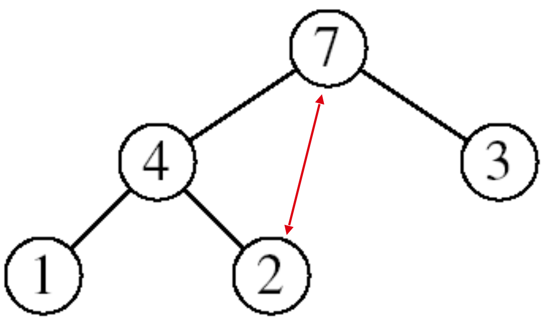
- Goal:
  - Sort an array using heap representations



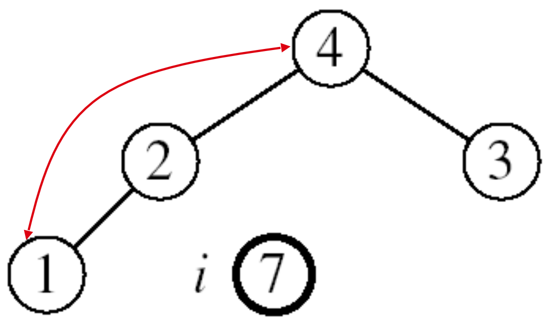
- Idea:
  - Build a **max-heap** from the array
  - Swap the root (the maximum element) with the last element in the array
  - "Discard" this last node by decreasing the heap size
  - Call MAX-HEAPIFY on the new root
  - Repeat this process until only one node remains

Example

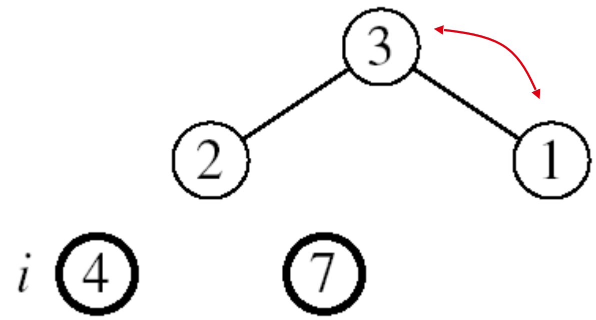
A=[7, 4, 3, 1, 2]



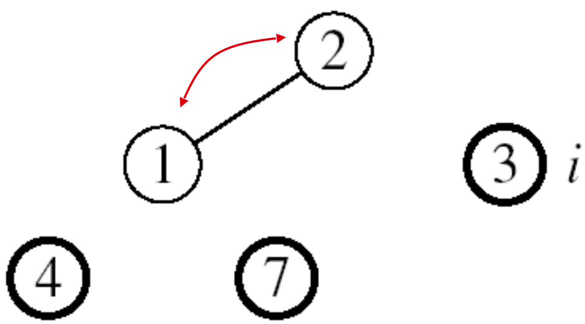
MAX-HEAPIFY(A, 1, 4)

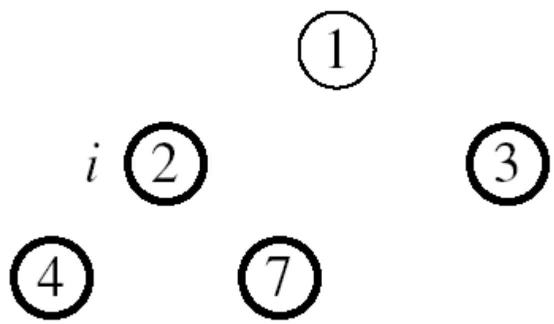


MAX-HEAPIFY(A, 1, 3)



MAX-HEAPIFY(A, 1, 2)





A	1	2	3	4	7

MAX-HEAPIFY(A, 1, 1)

Analyzing Heapsort

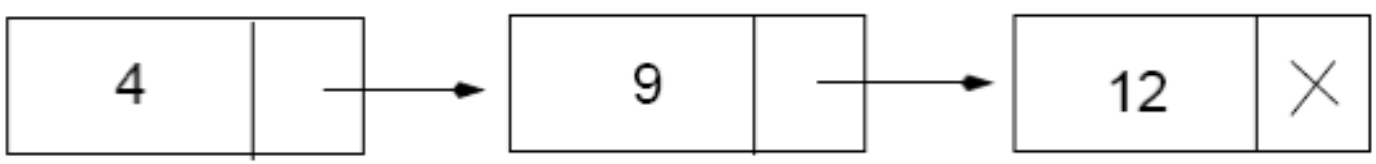
1. BUILD-MAX-HEAP(A) $O(n)$
2. **for** i ← length[A] **downto** 2
3. **do** exchange A[1] ↔ A[i]
4. MAX-HEAPIFY(A, 1, i - 1) $O(\lg n)$

n-1 times

- The call to **BUILD-MAX-HEAP** takes O(*n*) time
- Each of the *n* - 1 calls to **MAX-HEAPIFY** takes O(lg *n*) time
- Thus the total time taken by **HeapSort**
  $= O(n) + (n - 1)\, O(\lg n) = O(n) + O(n \lg n) = O(n \lg n)$

Priority Queues

- The heap data structure is incredibly useful for implementing *priority queues,* which maintains a set of elements.
- Properties of priority queues
  - Each element is associated with a value (priority)
  - The key with the highest (or lowest) priority is extracted first



Operations on Priority Queues

- Max-priority queues support the following operations:
  - INSERT(S, x): <u>inserts</u> element x into set S
  - EXTRACT-MAX(S): <u>removes and returns</u> element of S with largest key
  - MAXIMUM(S): <u>returns</u> element of S with largest key
  - INCREASE-KEY(S, x, k): <u>increases</u> value of element x's key to k (Assume k ≥ x's current key value)

HEAP-MAXIMUM

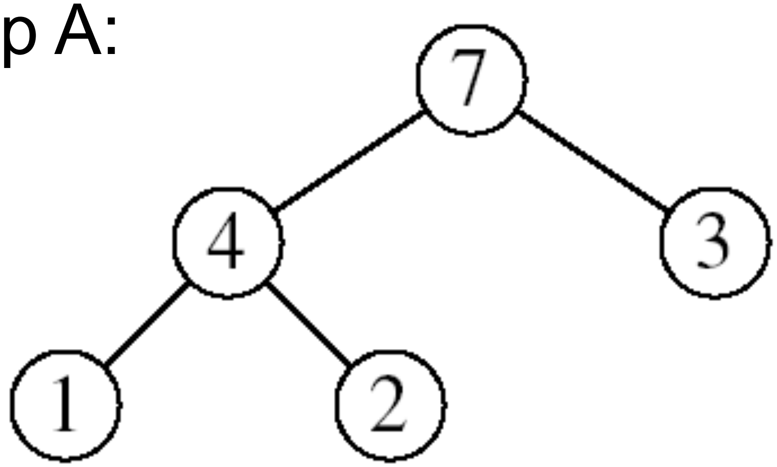
Goal: Return the largest element of the heap

Alg: HEAP-MAXIMUM(A)

Running time: O(1)

1. **return** A[1]

Heap A:

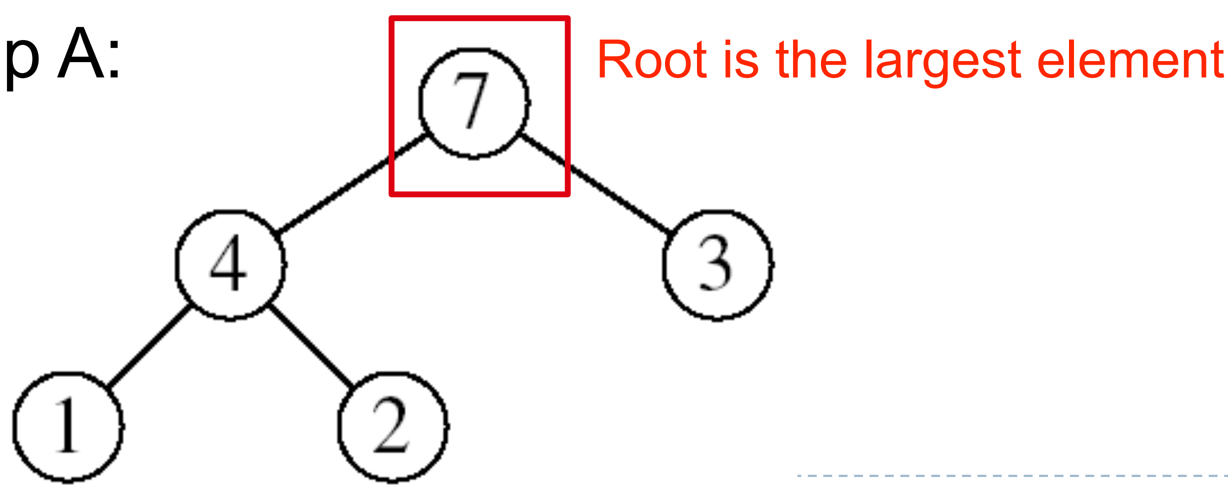


Heap-Maximum(A) returns 7

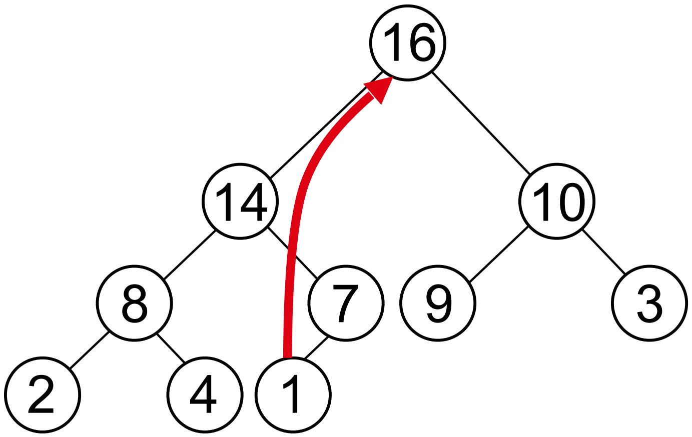
HEAP-EXTRACT-MAX

- Goal
  - Extract the largest element of the heap (i.e., return the max value and also remove that element from the heap
- Idea
  - Exchange the root element with the last
  - Decrease the size of the heap by 1 element
  - Call MAX-HEAPIFY on the new root, on a heap of size n-1

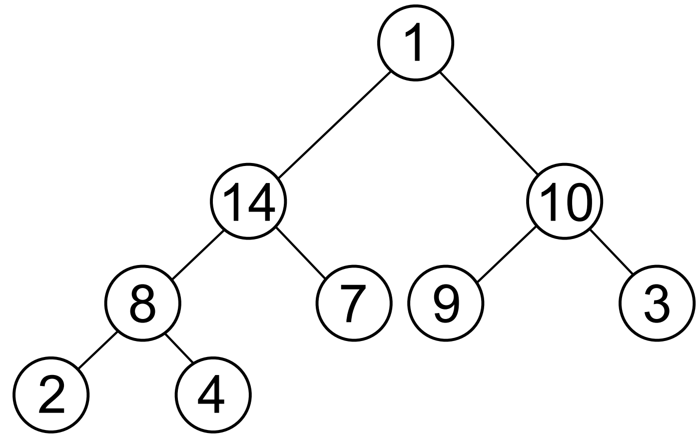
Heap A:



Example: HEAP-EXTRACT-MAX

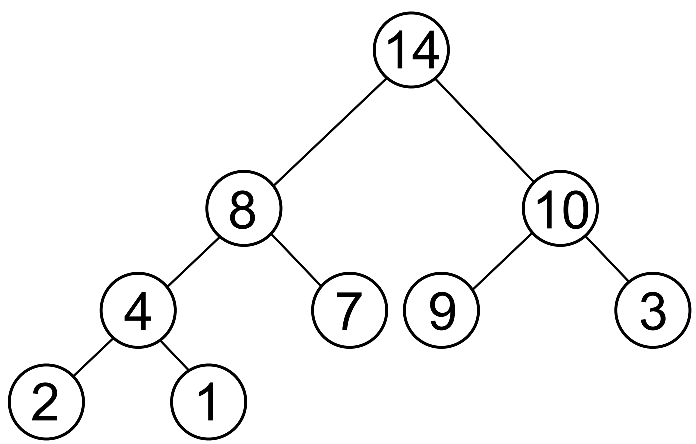


max = 16



Heap size decreased with 1

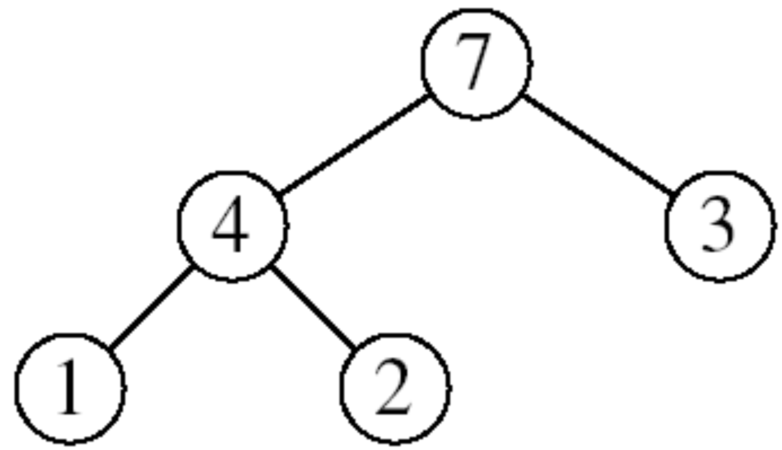
Call MAX-HEAPIFY(A, 1, n-1)



HEAP-EXTRACT-MAX

Alg: HEAP-EXTRACT-MAX(A, n)

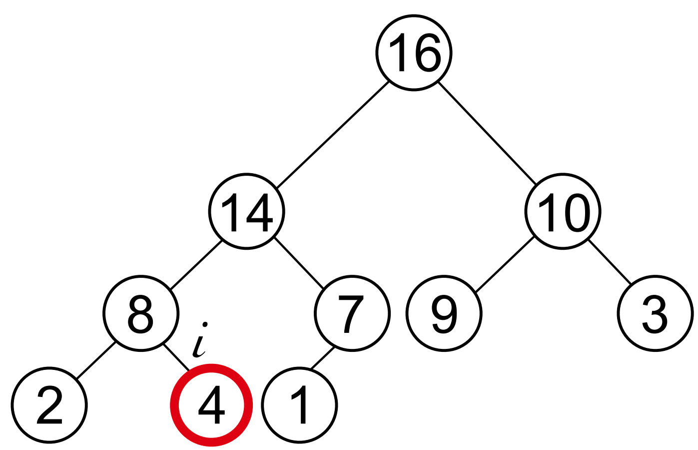
1. **if** n < 1
2. **then error** “heap underflow”
3. max ← A[1]
4. A[1] ← A[n]
5. MAX-HEAPIFY(A, 1, n-1)
6. **return** max



remakes heap

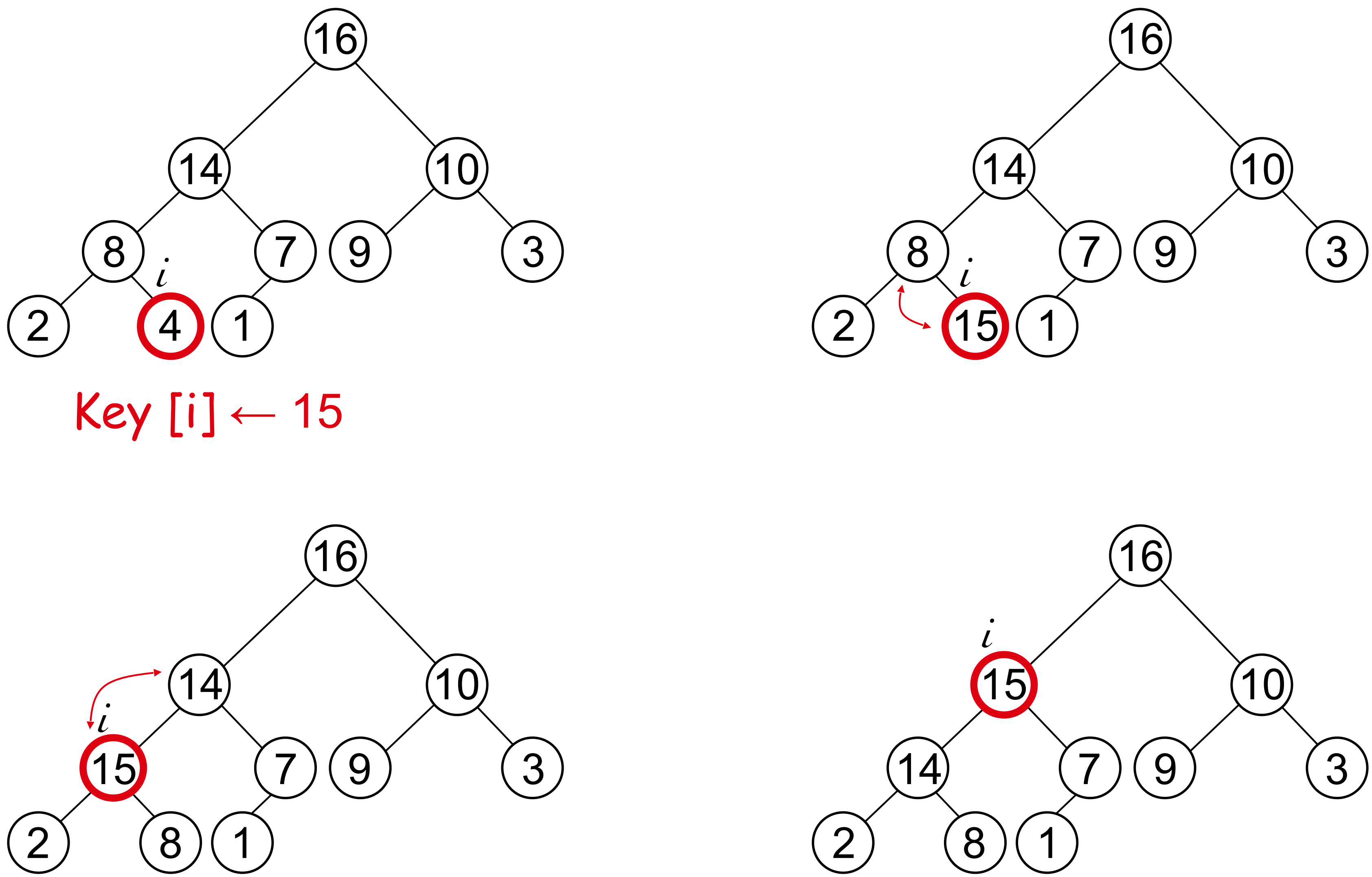
HEAP-INCREASE-KEY

- Goal
  - Increases the key of an element i in the heap
- Idea
  - Increment the key of A[i] to its new value
  - If the max-heap property does not hold anymore: traverse a path toward the root to find the proper place for the newly increased key



Key [i] ← 15

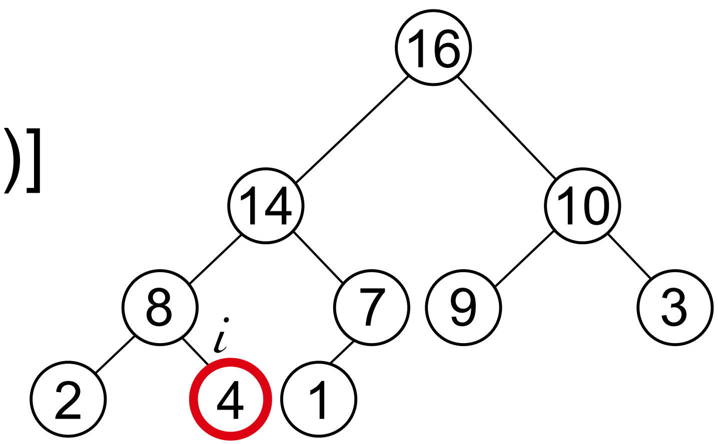
Example: HEAP-INCREASE-KEY



Analyzing HEAP-INCREASE-KEY

Alg: HEAP-INCREASE-KEY(A, i, key)

1. **if** key < A[i]
2. **then error** "new key is smaller than current key"
3. A[i] ← key
4. **while** i > 1 and A[PARENT(i)] < A[i]
5. **do** exchange A[i] ↔ A[PARENT(i)]
6. i ← PARENT(i)

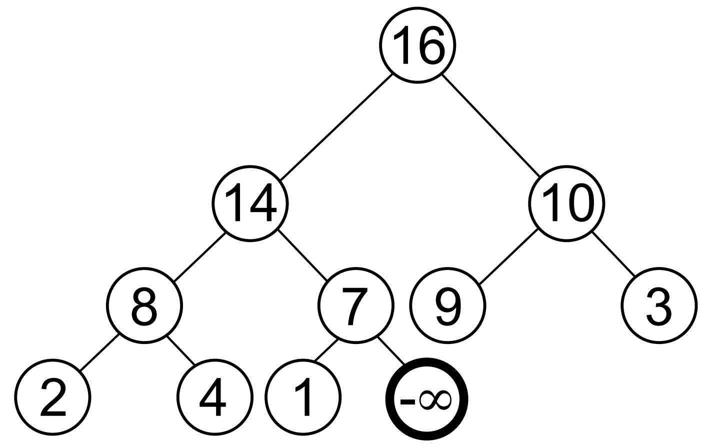


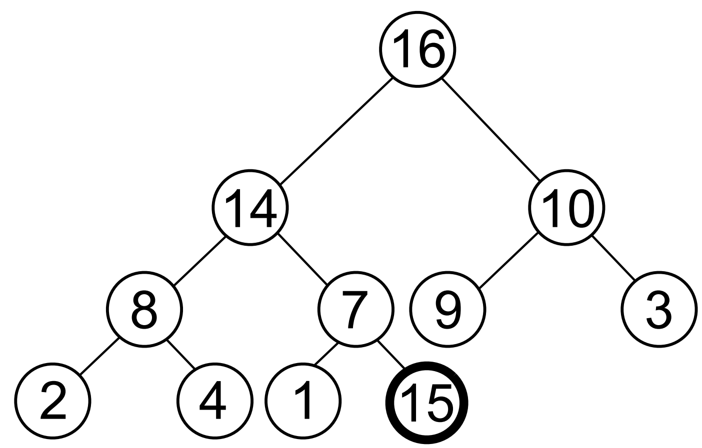
Key [i] ← 15

- Running time: O(lgn)

MAX-HEAP-INSERT

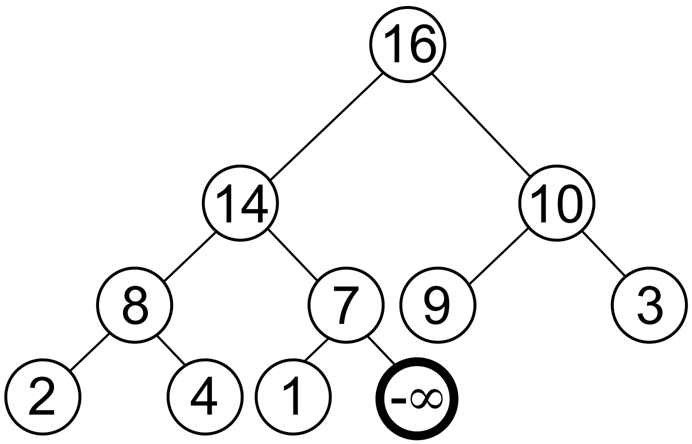
- Goal
  - Inserts a new element into a max-heap
- Idea
  - Expand the max-heap with a new element whose key is -∞
  - Calls HEAP-INCREASE-KEY to set the key of the new node to its correct value and maintain the max-heap property



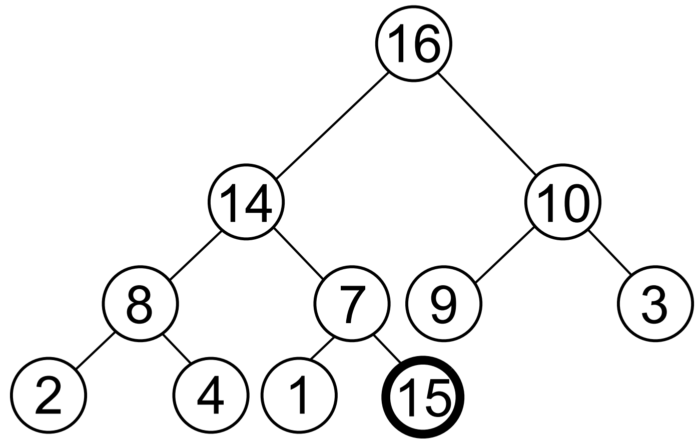


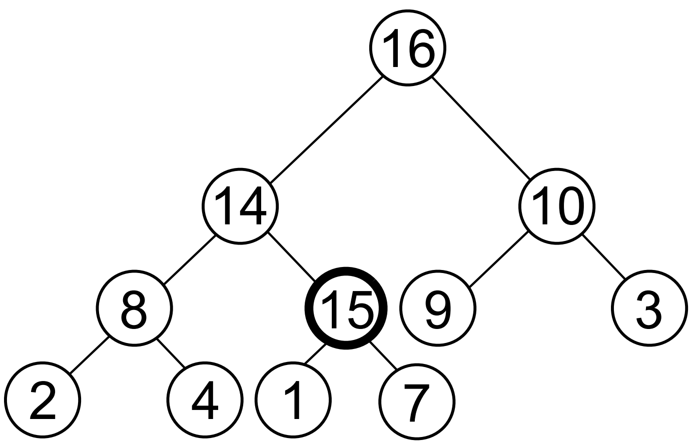
Example: MAX-HEAP-INSERT

Insert value 15:
- Start by inserting -∞

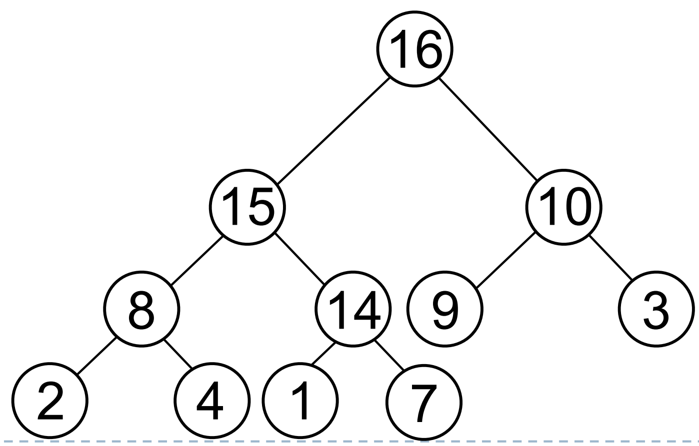


Increase the key to 15
Call HEAP-INCREASE-KEY on A[11] = 15





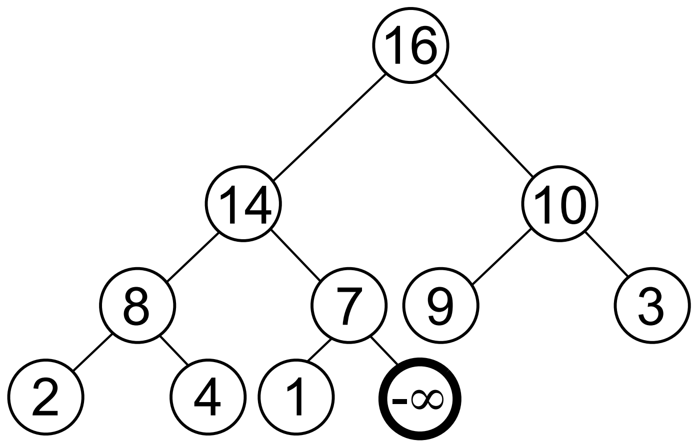
The restored heap containing
the newly added element



Analyzing MAX-HEAP-INSERT

Alg: MAX-HEAP-INSERT(A, key, n)

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



Running time: O(lgn)

Summary

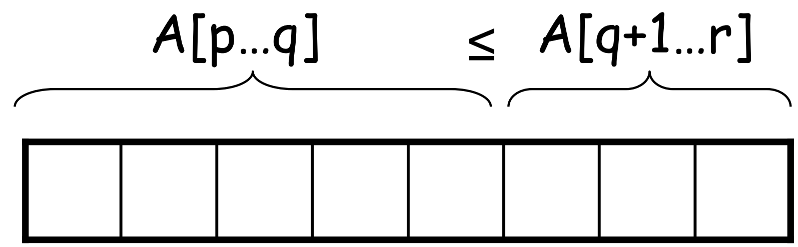
- We can perform the following operations on heaps:
  - MAX-HEAPIFY $O(\lg n)$
  - BUILD-MAX-HEAP $O(n)$
  - HEAP-SORT $O(n\lg n)$
  - MAX-HEAP-INSERT $O(\lg n)$
  - HEAP-EXTRACT-MAX $O(\lg n)$
  - HEAP-INCREASE-KEY $O(\lg n)$
  - HEAP-MAXIMUM $O(1)$

Quicksort: Brief Review

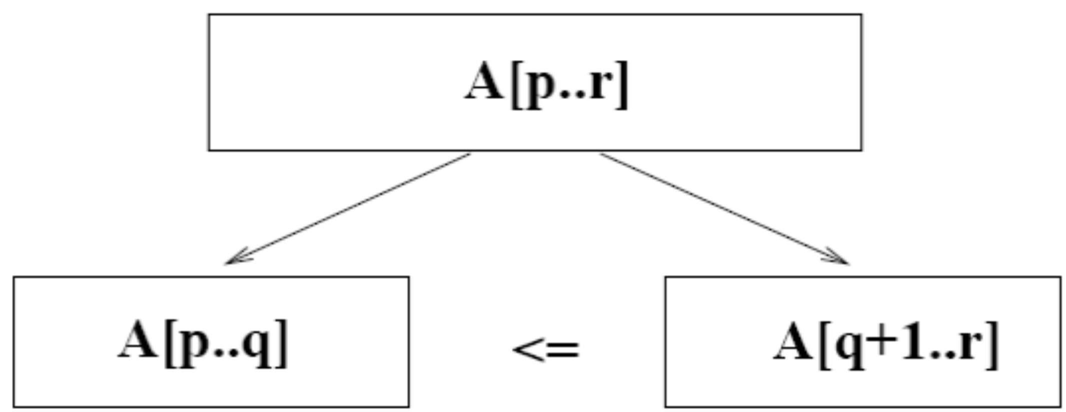
- Sorts in place
- Sorts O(nlgn) in the average case
- Sorts $O(n^2)$ in the worst case
  - But the worst case doesn't happen often (more on this later…)

Quicksort

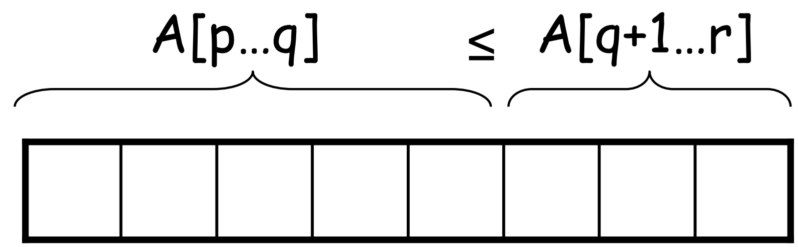
Sort an array A[p…r]



- **Divide**
  - Partition the array A into 2 subarrays A[p..q] and A[q+1..r], such that each element of A[p..q] is smaller than or equal to each element in A[q+1..r]
  - Need to find index q to partition the array



Quicksort



- **Conquer**
  - Recursively sort A[p..q] and A[q+1..r] by calls to Quicksort
- **Combine (unlike merge sort)**
  - Trivial: the arrays are sorted in place
  - No additional work is required to combine them
  - The entire array is now sorted

Quicksort

Alg.: QUICKSORT(A, p, r)

    if $p < r$

      then $q \leftarrow$ PARTITION(A, p, r)

            QUICKSORT (A, p, q)

            QUICKSORT (A, q+1, r)

Initially: $p=1$, $r=n$

Recurrence: $T(n) = T(q) + T(n - q) + f(n)$

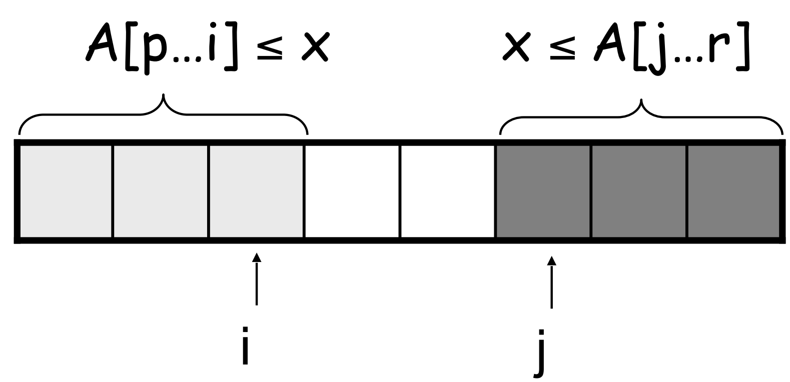
f(n) depends on partition()

Partition

- Clearly, all the action takes place in the **PARTITION()** function
  - Rearranges the subarray in place
  - End result:
    - Two subarrays
    - All values in first subarray ≤ all values in second
  - Returns the index of the "pivot" element separating the two subarrays
- *How should you implement this?*

Partition

- Choosing PARTITION()
  - There are different ways to do this
  - Each has its own advantages/disadvantages
- Hoare partition
  - Select a pivot element **x** around which to partition
  - Starts from both ends
  - Grows two regions
    - $A[p\ldots i] \leq \mathbf{x}$
    - $\mathbf{x} \leq A[j\ldots r]$



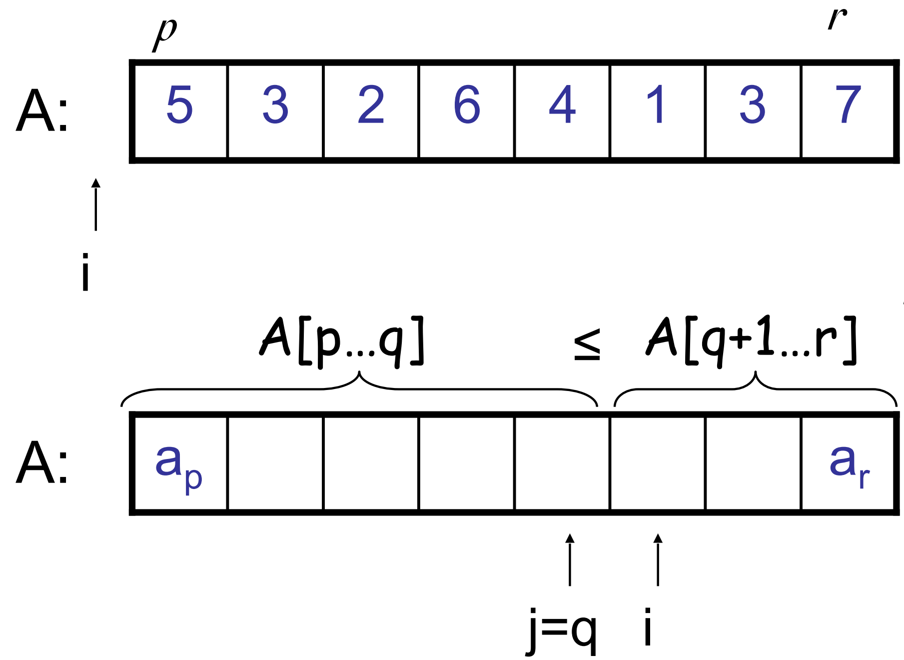
Partition in Words

- Partition(A, p, r):
  - Select an element to act as the “pivot” (*which?*)
  - Grow two regions, A[p..i] and A[j..r]
    - All elements in A[p..i] <= pivot
    - All elements in A[j..r] >= pivot
  - Increment i until A[i] >= pivot
  - Decrement j until A[j] <= pivot
  - Swap A[i] and A[j]
  - Repeat until i >= j
  - Return j

Partition Code

Alg. PARTITION (A, p, r)

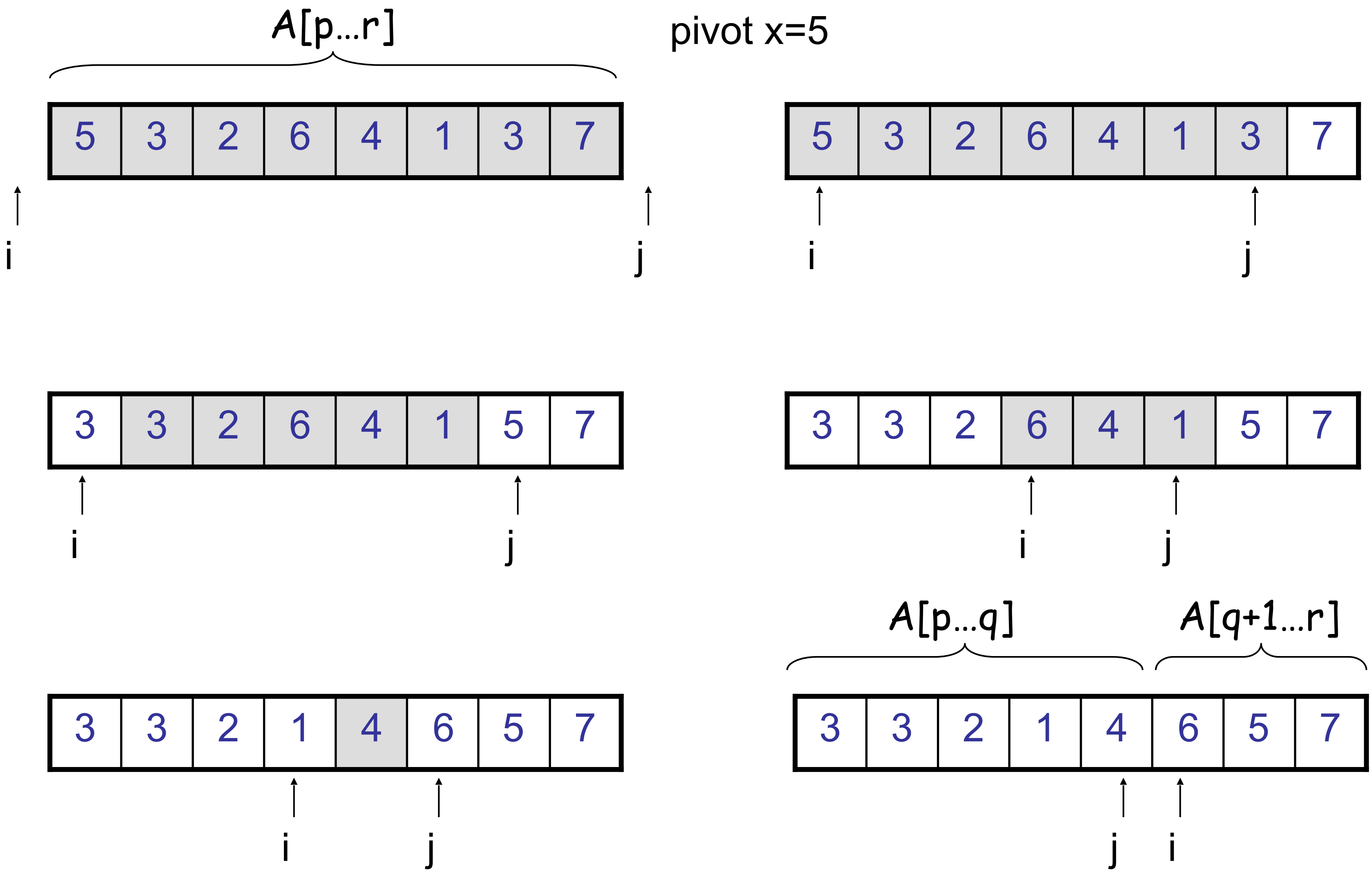
```
x ← A[p]
i ← p – 1
j ← r + 1
while TRUE
        do repeat j ← j – 1
              until A[j] ≤ x
        do  repeat i ← i + 1
              until A[i] ≥ x
         if i < j
              then exchange A[i] ↔ A[j]
        else return j
```



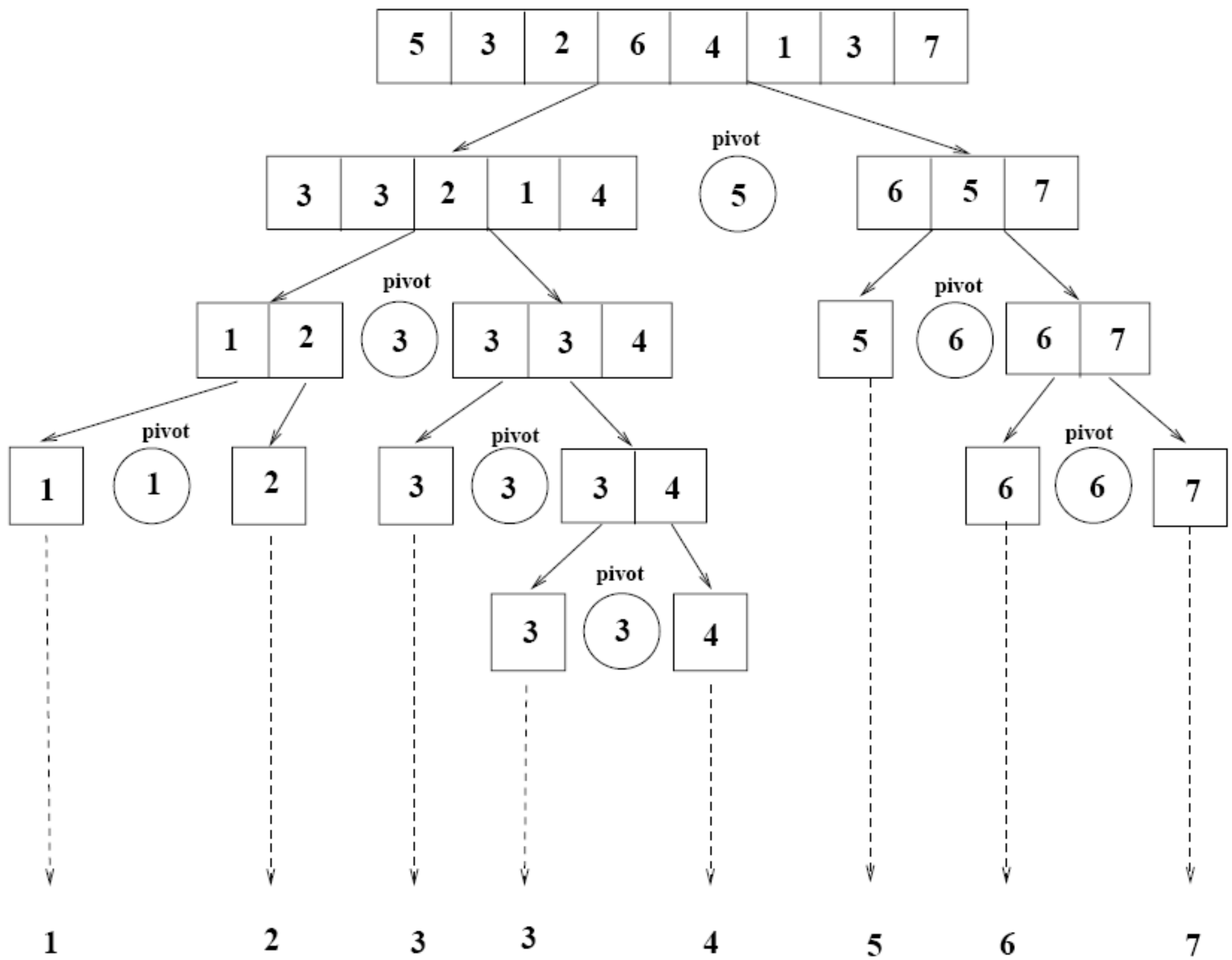
Each element is visited once!

Running time: Θ(n)
n = r – p + 1

Example


A[p…r]
pivot x=5
5 3 2 6 4 1 3 7
i
j
5 3 2 6 4 1 3 7
i
j
3 3 2 6 4 1 5 7
i
j
3 3 2 6 4 1 5 7
i
j
A[p…q]
A[q+1…r]
3 3 2 1 4 6 5 7
i
j
3 3 2 1 4 6 5 7
j
i

Example



5 3 2 6 4 1 3 7
pivot
3 3 2 1 4
5
6 5 7
pivot
1 2
3
3 3 4
pivot
5
6
6 7
pivot
1
1
2
pivot
3
3
3 4
pivot
6
6
7
pivot
3
3
4
1 2 3 3 4 5 6 7

Quicksort: Recurrence

Alg.: QUICKSORT(A, p, r)

if p < r

then q ← PARTITION(A, p, r)

QUICKSORT (A, p, q)

QUICKSORT (A, q+1, r)

Initially: p=1, r=n

Recurrence: $T(n) = T(q) + T(n - q) + n$

Analyzing Quicksort

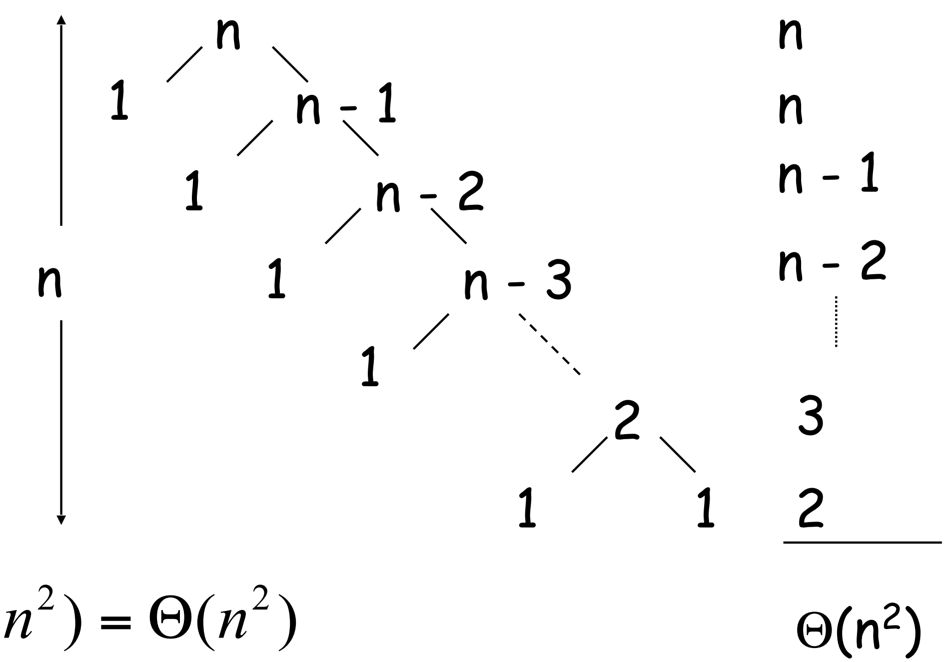
- *What will be the worst case for the algorithm?*
  - Partition is always unbalanced
- *What will be the best case for the algorithm?*
  - Partition is perfectly balanced
- *Which is more likely?*
  - The latter, except...
- *Will any particular input elicit the worst case?*
  - Yes: Already-sorted input

Analyzing Quicksort: Worst Case Partitioning

- Worst-case partitioning
  - One region has one element and the other has n – 1 elements
  - Maximally unbalanced
- Recurrence: q=1

T(n) = T(1) + T(n – 1) + n,

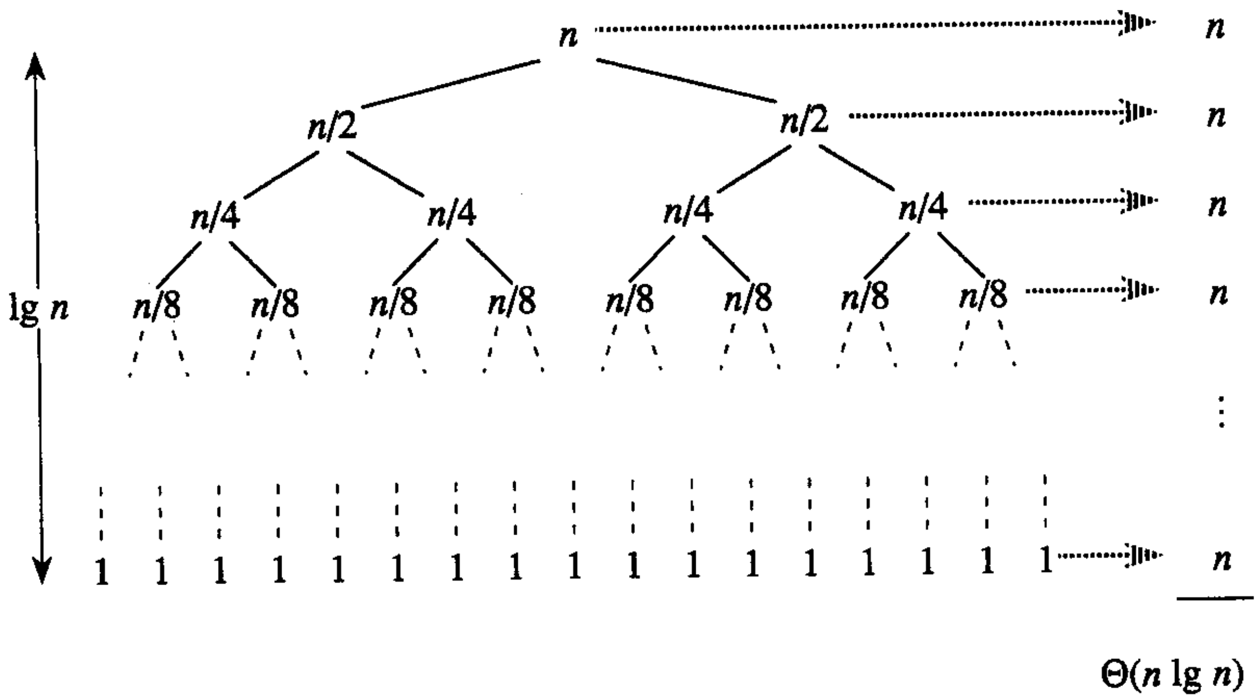
  - T(1) = Θ(1)
  - T(n) = $n + \left(\sum_{k=1}^{n} k\right) - 1 = \Theta(n) + \Theta(n^2) = \Theta(n^2)$



When does the worst case happen?

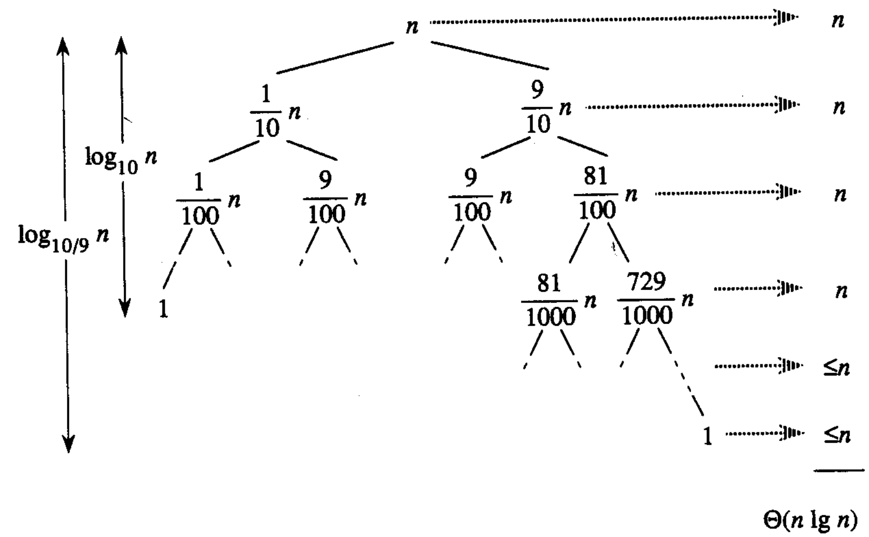
Analyzing Quicksort: Best Case Partitioning

- Best-case partitioning
  - Partitioning produces two regions of size n/2
- Recurrence: q=n/2
  - T(n) = 2T(n/2) + Θ(n)
  - T(n) = Θ(nlgn) (Master theorem)



Case Between Worst and Best

- An intuitive explanation/example:
  - Suppose that partition() always produces a 9-to-1 split. This looks quite unbalanced!
  - The recurrence is thus:
    - T(n) = T(9n/10) + T(n/10) + n
    - *How deep will the recursion go?*



- Using the recursion tree:

$$\text{longest path: } Q(n) \le n \sum_{i=0}^{\log_{10/9} n} 1 = n(\log_{10/9} n + 1) = c_2 n lgn$$

$$\text{shortest path: } Q(n) \ge n \sum_{i=0}^{\log_{10} n} 1 = n \log_{10} n = c_1 n lgn$$

Thus, $Q(n) = \Theta(nlgn)$

How does partition affect performance?

- **Any splitting of constant proportionality** yields $\Theta(nlgn)$ time !!!

- Consider the $(1 : n-1)$ splitting:

$$\text{ratio}=1/(n-1) \text{ not a constant !!!}$$

- Consider the $(n/2 : n/2)$ splitting:

$$\text{ratio}=(n/2)/(n/2) = 1 \text{ it is a constant !!}$$

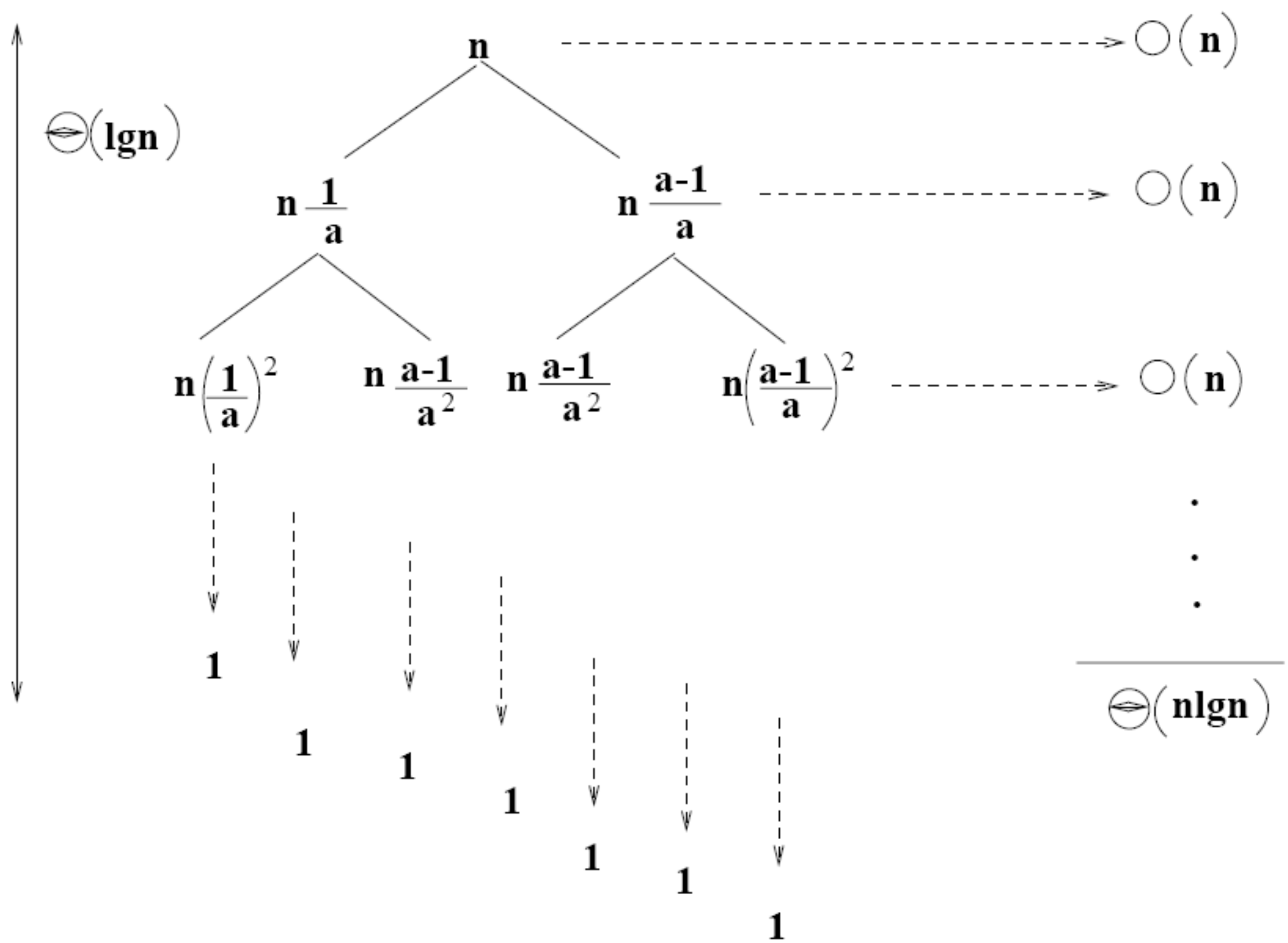
- Consider the $(9n/10 : n/10)$ splitting:

$$\text{ratio}=(9n/10)/(n/10) = 9 \text{ it is a constant !!}$$

How does partition affect performance?

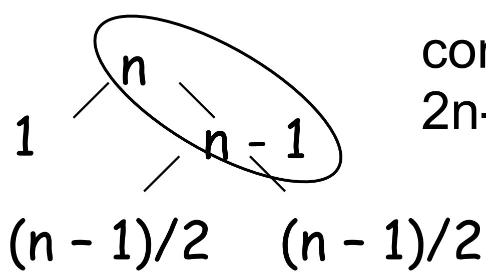
- Any $((a-1)n/a : n/a)$ splitting:

  ratio=$((a-1)n/a)/(n/a) = a-1$ it is a constant !!



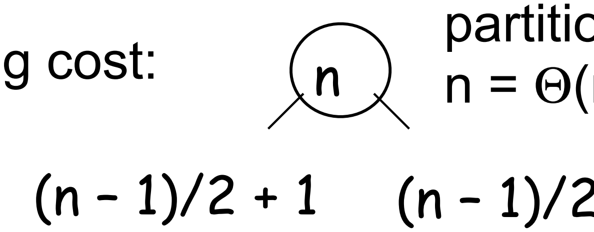
Analyzing Quicksort: Average Case Partitioning

- Average case
  - All permutations of the input numbers are equally likely
  - On a random input array, we will have a **mix** of well balanced and unbalanced splits
  - Good and bad splits are randomly distributed across throughout the tree



combined partitioning cost:
$2n-1 = \Theta(n)$

Alternate of a good and a bad split



partitioning cost:
$n = \Theta(n)$

Nearly well balanced split

Running time of Quicksort when levels alternate between good and bad splits is O(nlgn)

Randomizing Quicksort

- Randomly permute the elements of the input array before sorting
- OR ... modify the PARTITION procedure
  - At each step of the algorithm we exchange element A[p] with an element chosen at random from A[p…r]
  - The pivot element x = A[p] is equally likely to be any one of the r – p + 1 elements of the subarray

Randomizing PARTITION

Alg.: RANDOMIZED-PARTITION(A, p, r)

i ← RANDOM(p, r)

exchange A[p] ↔ A[i]

return PARTITION(A, p, r)

Randomizing Quicksort

Alg.: RANDOMIZED-QUICKSORT(A, p, r)

if p < r

then q ← RANDOMIZED-PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, q)

RANDOMIZED-QUICKSORT(A, q + 1, r)

What's next...

- Randomized Quick Sort
- Sorting Lower Bound
- Order Statistics & Selection