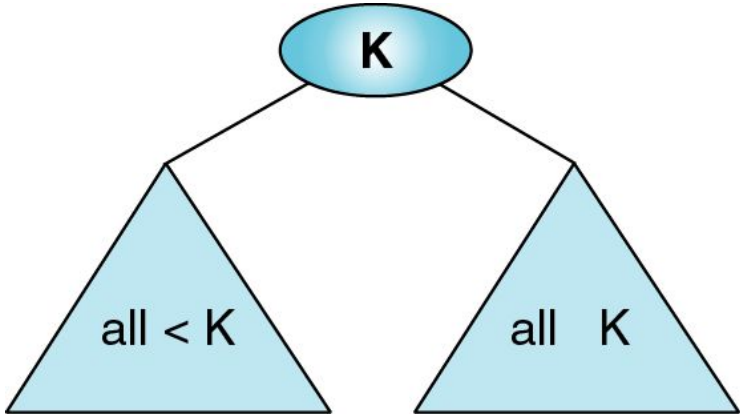
Data Structures & Algorithms

Lecture 10: Heaps

NUST
SCHOOL OF ELECTRICAL ENGINEERING &
COMPUTER SCIENCE (SEECS)

Recap

- Binary Search Trees (BST)



- BST Operations
  - Insertion
  - Search
  - Traversal
  - Deletion

- BST Functions
  - Minimum/Maximum function
  - Successor function
  - Predecessor function

Heap

- A **heap** is a specialized tree-based data structure

- Can be viewed as a nearly complete binary tree

- A **complete binary tree** is a tree in which leaves are filled from left to right on one level before moving to next level

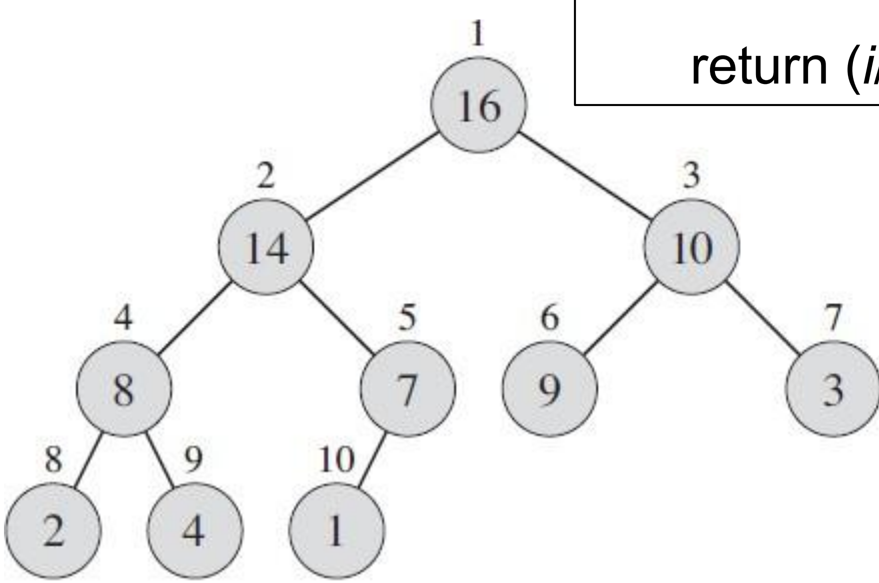
- Heap - a tree completely filled on all levels except possibly the lowest

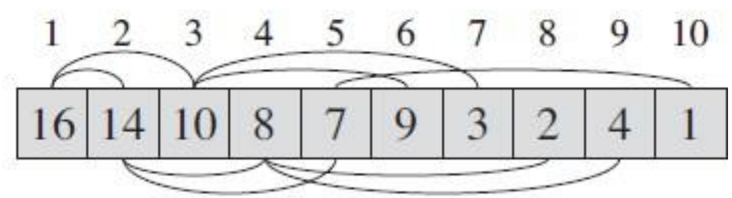
Implementation

NUST SEECS

- Heaps can be easily implemented by arrays
- Each node represents an element of the array
- Complete binary tree – if not full, then the only unfilled level is filled in from left to right

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





Heaps

NUST SEECS

- Two kinds of binary heaps
  - Max-heap
  - Min-heap
- Values in the nodes satisfy the heap properties
  - Max-heap property
  - Min-heap property

Heap Property

NUST SEECS

- The **heap property** of a tree is a condition that must be true for the tree to be considered a **heap**

- **Min-heap property**

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

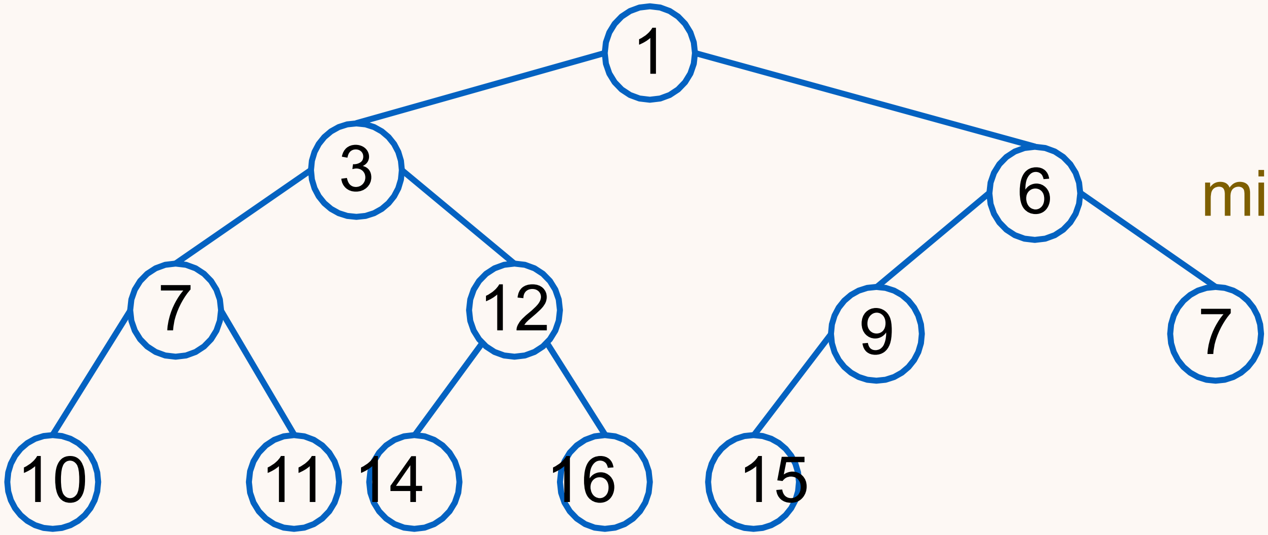
  *So, the root of any sub-tree holds the **least** value in that sub-tree*

- **Max-heap property**

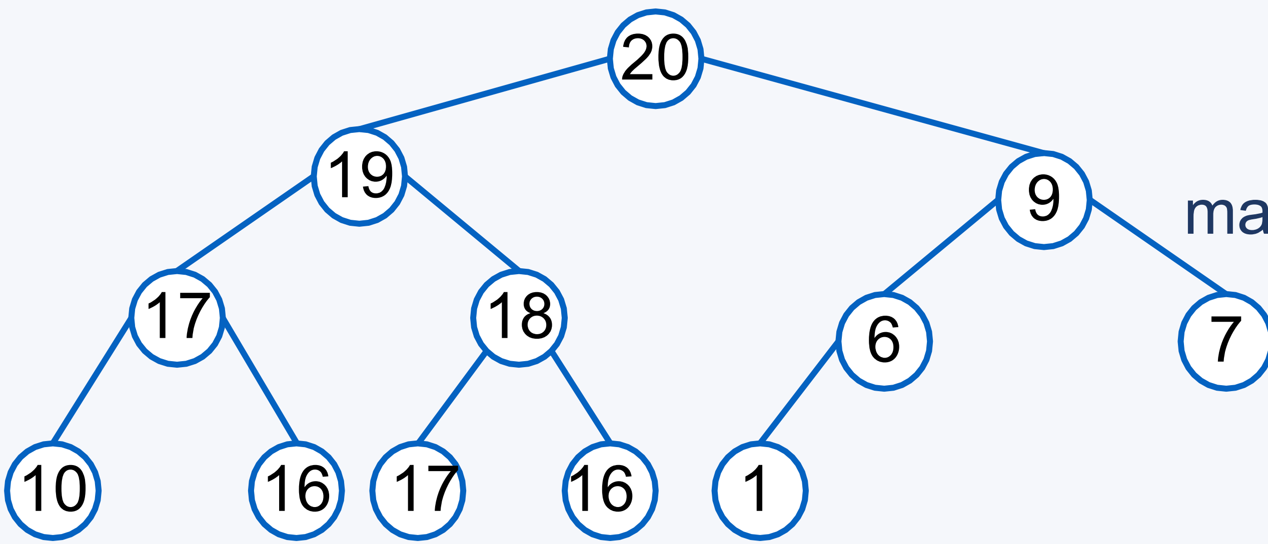
  A[parent(i)] $\geq$ A[i]

  *The root of any sub-tree holds the **greatest** value in the sub-tree*

Heap Property



Min-heap satisfying min-heap-property



Max-heap satisfying max-heap-property

Operations on Heaps

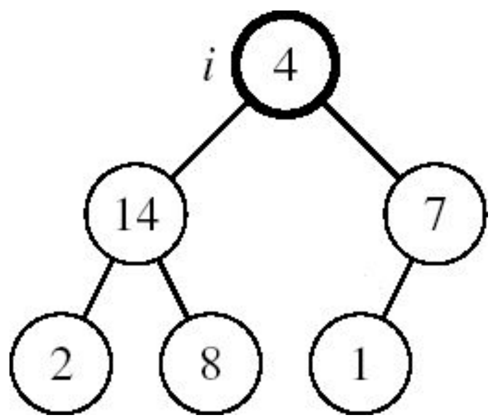
- Maintain/Restore the max-heap property
  - MAX-HEAPIFY
- Create a max-heap from an unordered array
  - BUILD-MAX-HEAP
- Sort an array in place
  - HEAPSORT
- Priority queues

Maintaining the Heap Property

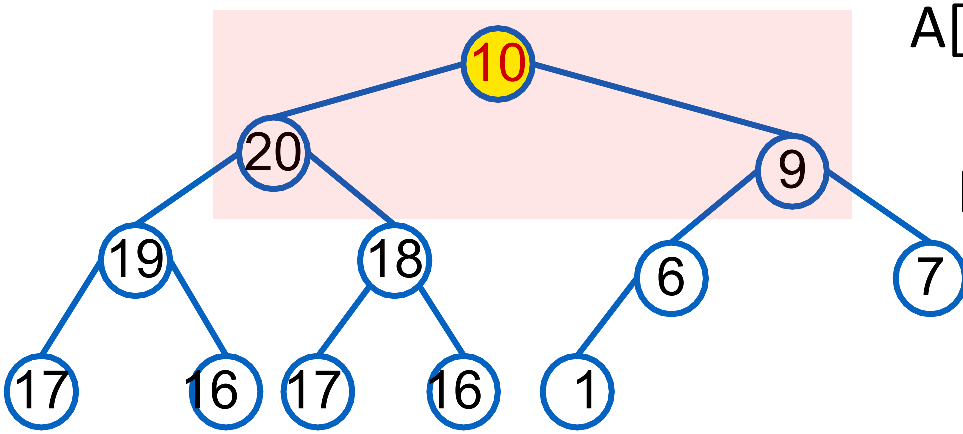
NUST SEECS

- Suppose a node is smaller than a child
  - Left and Right subtrees of *i* are max-heaps

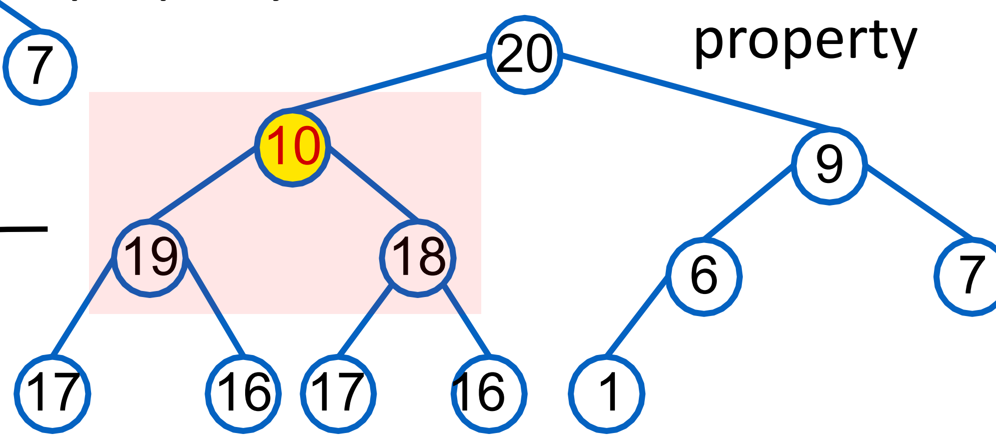
- To eliminate the violation:
  - Exchange with larger child
  - Move down the tree
  - Continue until node is not smaller than children



Max-Heapify(A,1,n)

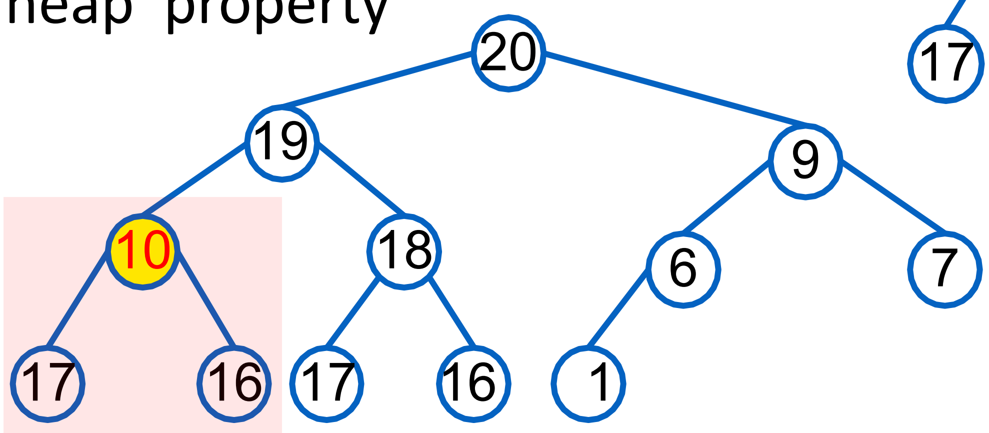


A[1] violates heap property

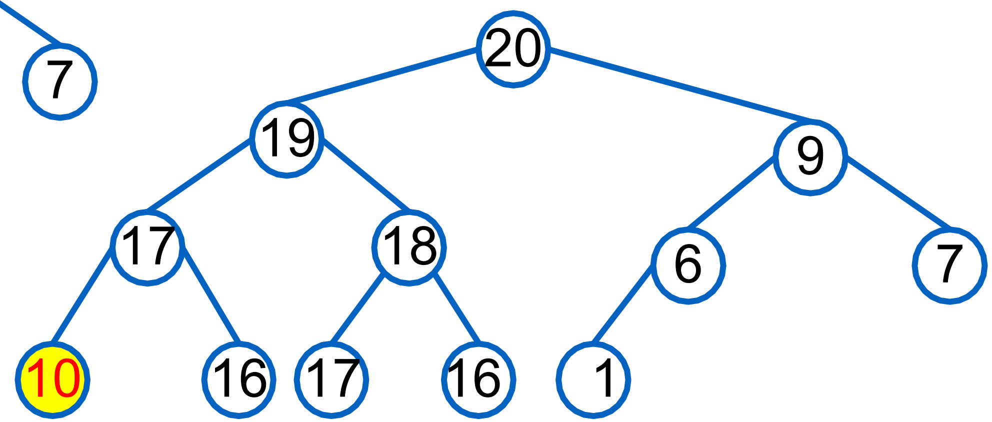


A[2] violates heap property

A[4] violates heap property



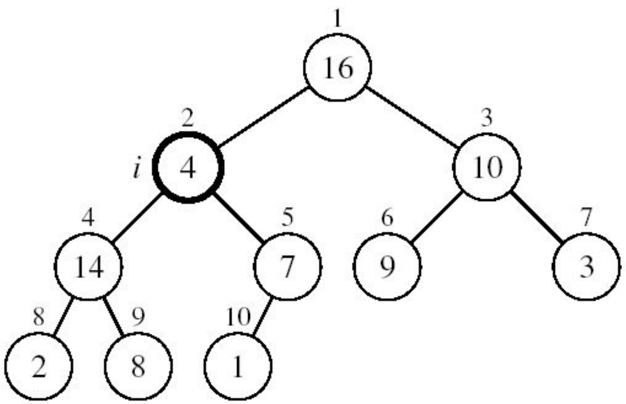
Assume we are given a tree represented by an array A



Another Example

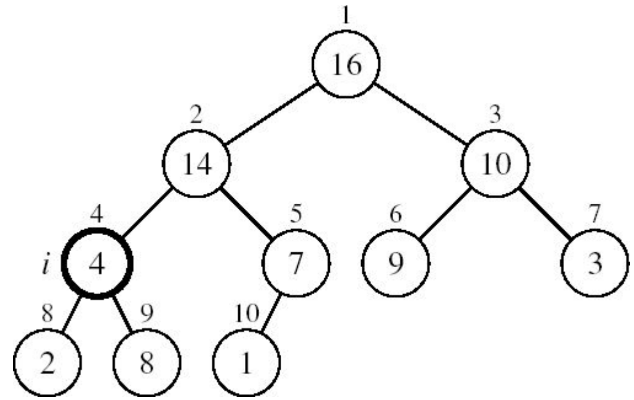
NUST SEECS

MAX-HEAPIFY(A, 2, 10)



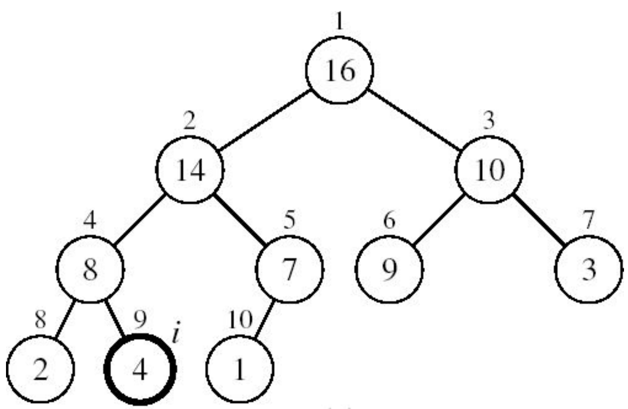
A[2] violates the heap property

A[2] ↔ A[4]



A[4] violates the heap property

A[4] ↔ A[9]



Heap property restored

Max-Heapify

NUST SEECS

- 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

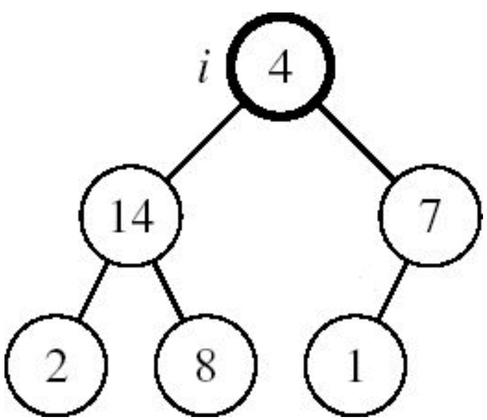
  - Exchange with bigger of the two children and keep sifting down
    - So, A[i] **moves down** in the heap
    - The move of A[i] may have caused a violation of the max-heap property at its new location. So, we must recursively call Max-Heapify(A,i) at the location i where the node ***"lands"***

  - This is a **top down** approach

Max-Heapify

Assumptions:

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



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

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

Running time of Max-Heapify

Intuitively

- In worst case, it traces a path from the root to a leaf (longest path length: d)
- At each level, it makes exactly 2 comparisons
- Total number of comparisons is $2d$
- Running time is **O(d)** or since $\boldsymbol{d = \log_2 n - 1}$, so **O(log n)**

Building a Heap

- How to build a heap from scratch?
  - Convert an **arbitrary array** into a **max-heap**
    - We call Max-Heapify(A, i, n) for every i starting at last node and going to the root
    - I.e., follow **bottom-up** strategy

Build-Max-Heap(A)

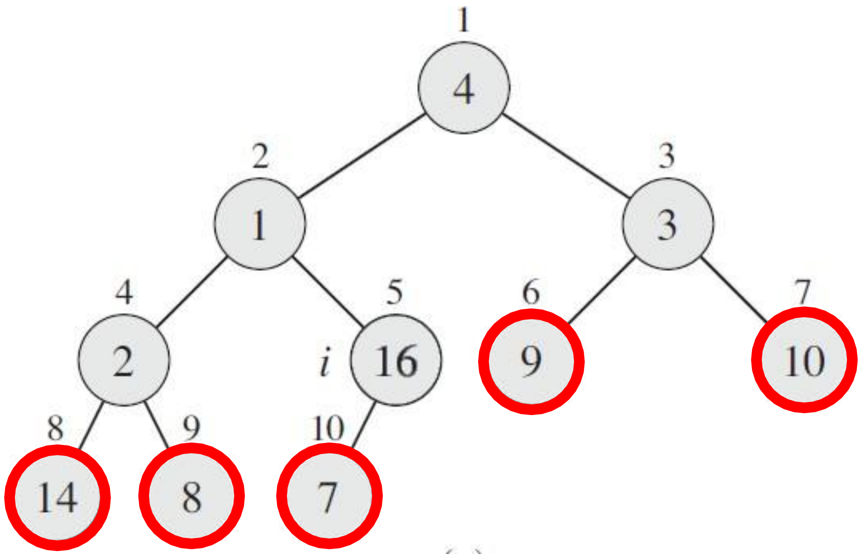
1. n = *length*[A]
2. for i = n **down to** 1
3. Max-Heapify(A, i, n)

How Build-Max-Heap(A) works?

A	4	1	3	2	16	9	10	14	8	7

Important observation

- There is no need to call Max-Heapify() on leaf nodes
- Since, each is a 1-element heap to begin with, this call always returns without any change to original Heap
- For array of length *n*, all elements in range A[*n*/2+1 ... *n*] are leaves of the tree



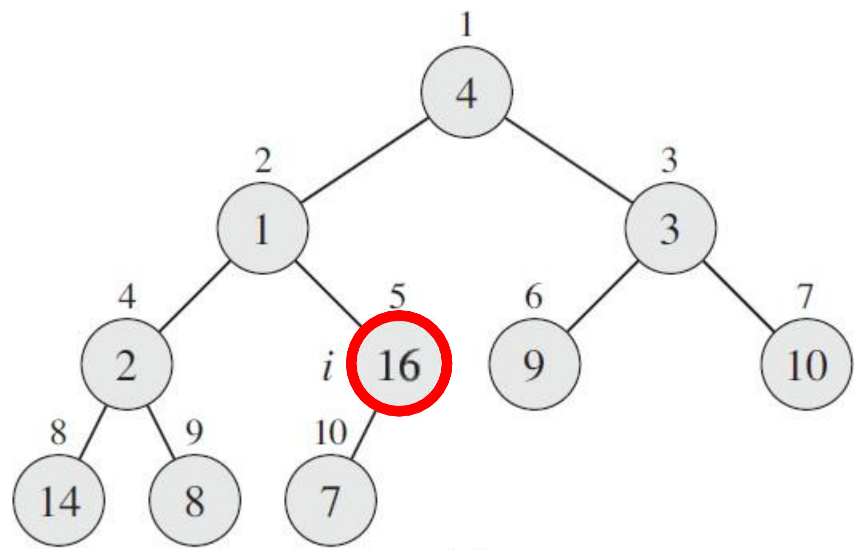
So, what will be better array index to start with?

Start at internal node

How Build-Max-Heap(A) works?

A									
4	1	3	2	16	9	10	14	8	7

- At most, the internal node with the largest index is equal to *n*/2

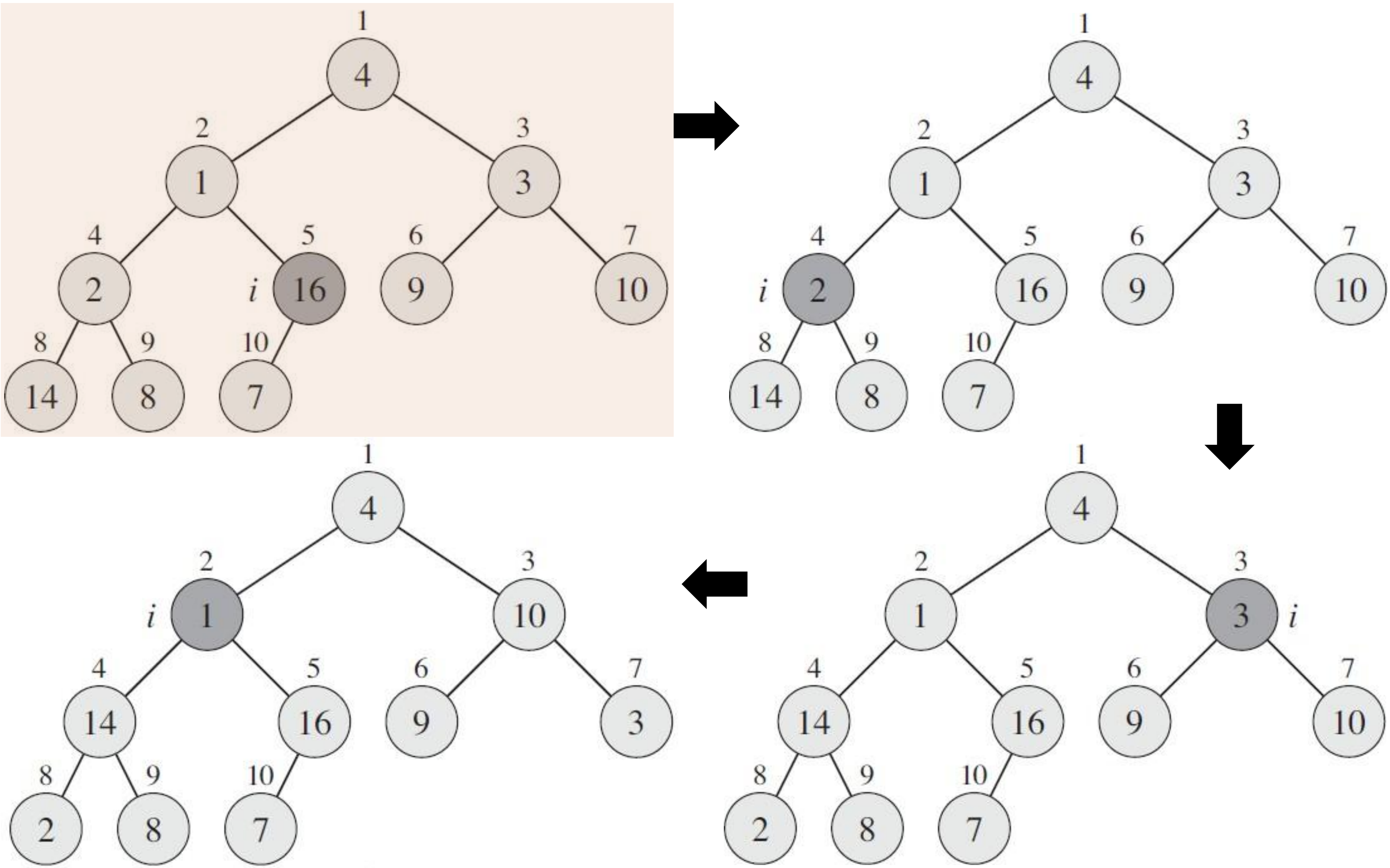


Build-Max-Heap(A)

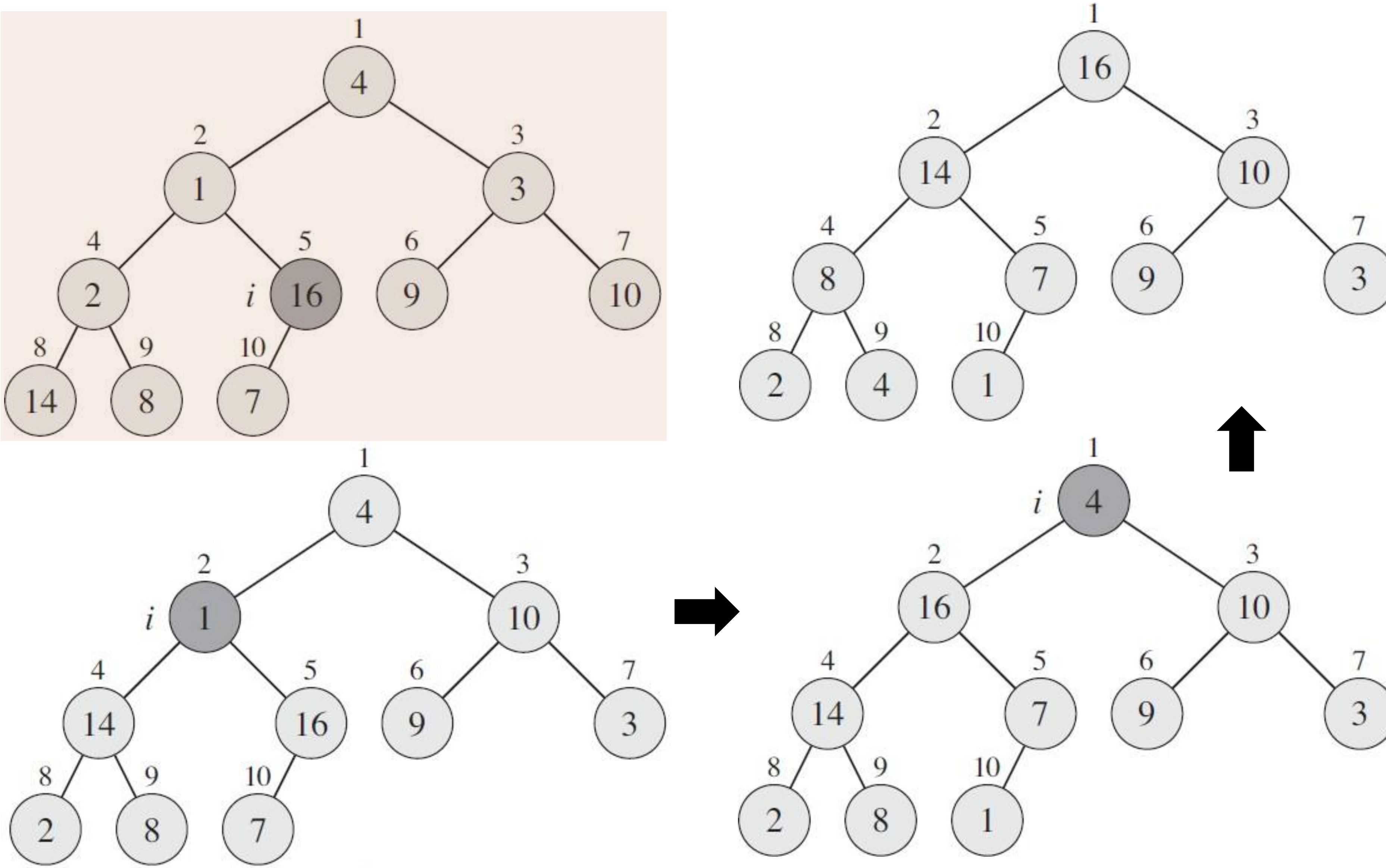
n = *length*[A]
for *i* = ***n/2*** **down to** 1
    Max-Heapify(A, *i, n*)

Working of Build-Max-Heap

NUST SEECS



Working of Build-Max-Heap



How Build-Max-Heap(A) works?

NUST SEECS

Build-Max-Heap(A)

1. n = *length*[A]
2. for i = **$n/2$** **down to** 1
3. Max-Heapify(A, i, n)

O(log n)

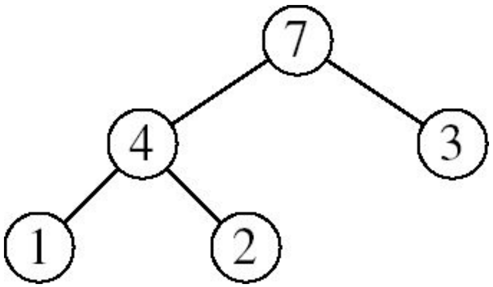
O(n)

$\Rightarrow$ Running time: **O($n$log$n$)**

- This is correct upper bound, however, this is not an asymptotically tight upper bound

Heapsort

- **Goal:**
  - Sort an array using heap representations



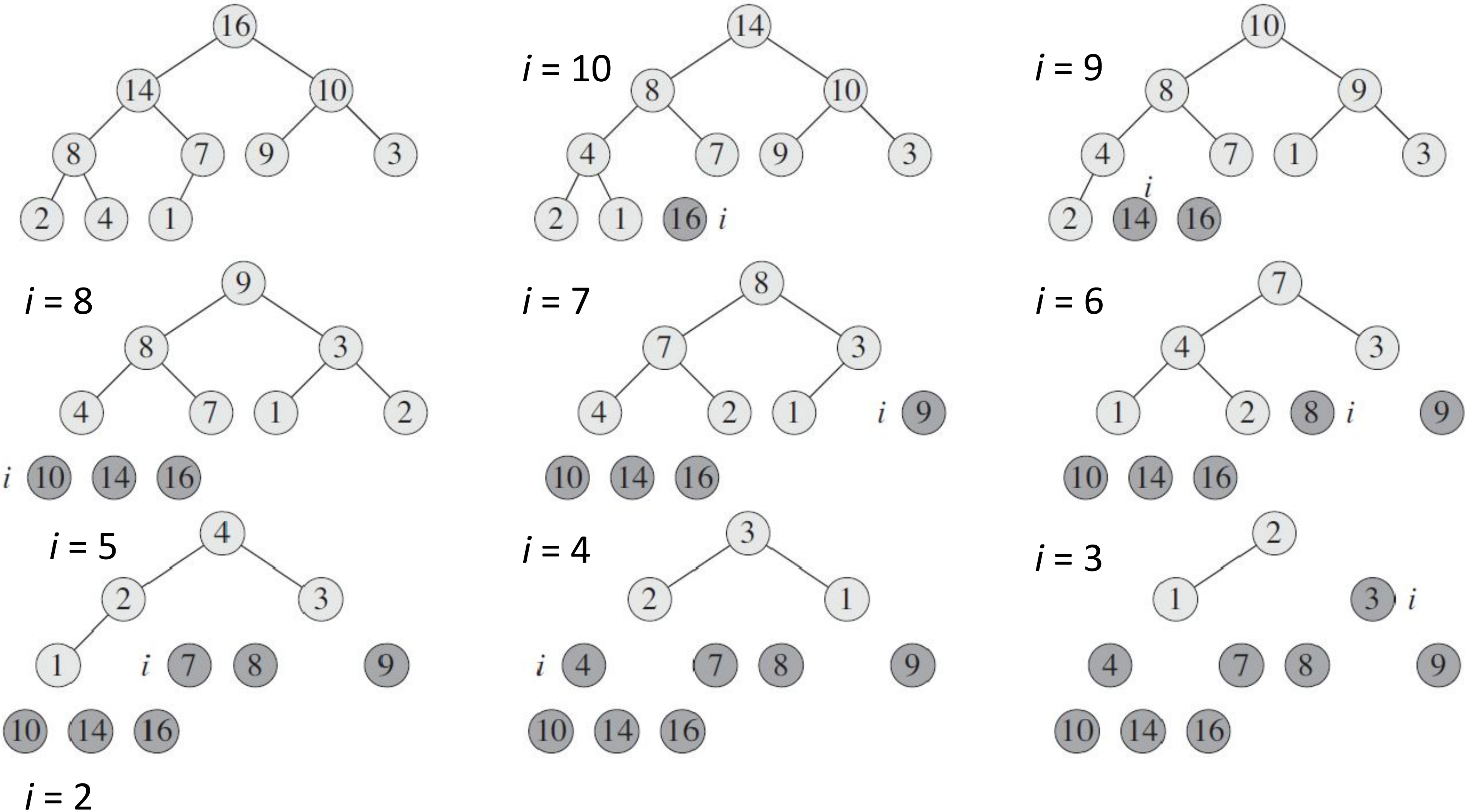
- **Idea:**
  - Build a **max-heap** from the array
  - Swap the **root** (the maximum element) with the **last** element in the array
  - **"Discard"** this last node by decreasing the heap size
  - Call MAX-HEAPIFY on the new root
  - Repeat this process until only one node remains

Heapsort

Heapsort(A)

1. Build-Max-Heap(A)
2. **for** *i* = *length*[A] **downto** 2
3. exchange A[1] ↔ A[*i*]
4. *heap-size*[A] = *heap-size*[A]-1
5. Max-Heapify(A,1, *heap-size*[A])

Example: Heapsort



NUST SEECS

Heapsort(A)

1. Build-Max-Heap(A) — O(n)
2. for i = *length*[A] **downto** 2
3. exchange A[1] ↔ A[i]
4. *heap-size*[A] = *heap-size*[A]-1
5. Max-Heapify(A,1, *heap-size*[A]) — O(log n)

n - 1 times

- Each of the n - 1 calls to Max-Heapify() takes O(log n) time => **O(n log n)**
- Uses the very useful heap data structure
  - Complete binary tree
  - Heap property: parent key >= children's keys
- Sorts in place

Priority Queues

- Heapsort is an excellent algorithm, but a good implementation of quicksort usually beats it in practice
- Nevertheless, the heap data structure itself has many uses
- In this lecture, we present one of the most popular applications of a heap i.e., as an efficient **priority queue**
  - As with heaps, priority queues come in two forms: max-priority queues and min-priority queues

  - We will focus here on how to implement max-priority queues, which are in turn based on max-heaps

Priority Queues

Properties

- Each element is associated with a value (priority)
- The key with highest (or lowest) priority is extracted first

Priority Queues

NUST SEECS

- A ***priority queue*** is a data structure for maintaining a set S of elements, each with an associated value called a ***key***
- Max-priority queues support the following operations:
  - HEAP-MAXIMUM(S): returns element of S with largest key
  - HEAP-EXTRACT-MAX(S): removes and returns element of S with largest key (DEQUEUE)
  - HEAP-INCREASE-KEY(S, x, k): increases value of element x's key to k (Assume k ≥ x's current key value)
  - MAX-HEAP-INSERT(S, x): inserts element x into set S (ENQUEUE)

Max-Priority Queues

- Among their other applications, we can use **max-priority queues** to schedule jobs on a shared computer
  - The max-priority queue keeps track of the jobs to be performed and their relative priorities
  - When a job is finished or interrupted, the scheduler selects the highest-priority job from among those pending by calling EXTRACT-MAX
  - The scheduler can add a new job to the queue at any time by calling INSERT

HEAP-MAXIMUM

Goal:

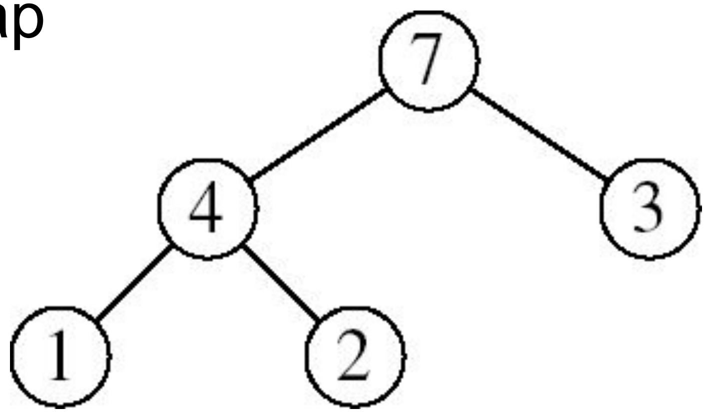
- Return the largest element of the heap

HEAP-MAXIMUM(A)

1. **return** A[1]

Running time: O(1)

Heap A:



Heap-Maximum(A) returns 7

HEAP-EXTRACT-MAX

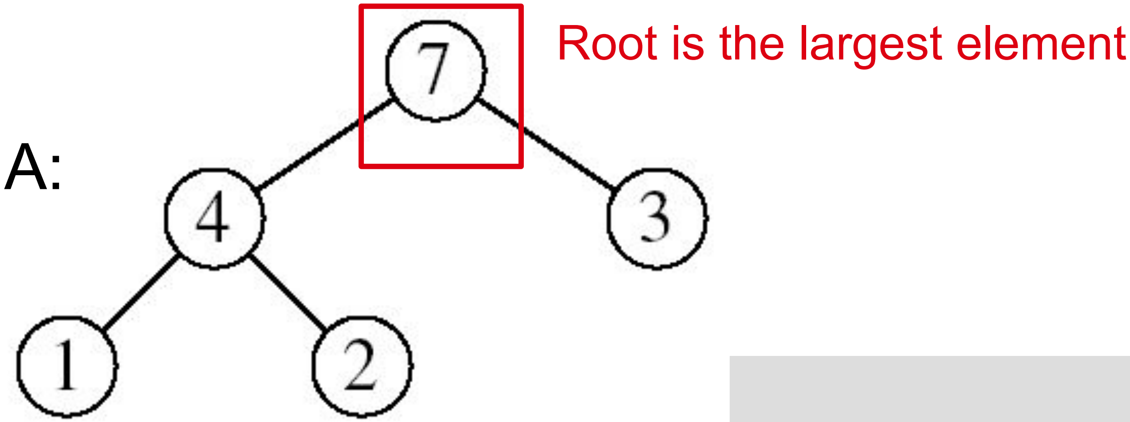
Goal:

- Extract the largest element of the heap i.e., return the max value and also remove that element from the heap

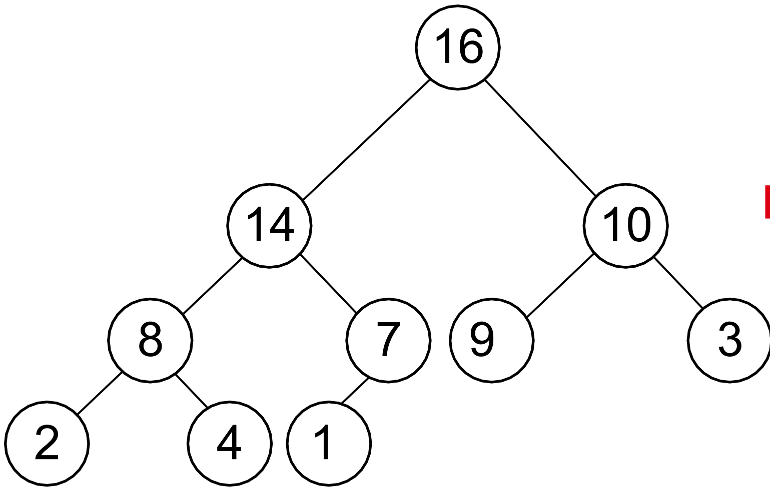
Idea:

- 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

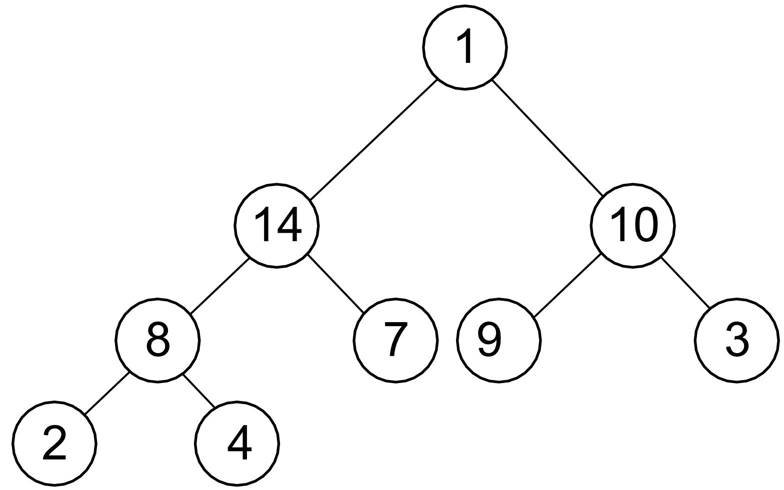
Heap A:



Example: HEAP-EXTRACT-MAX

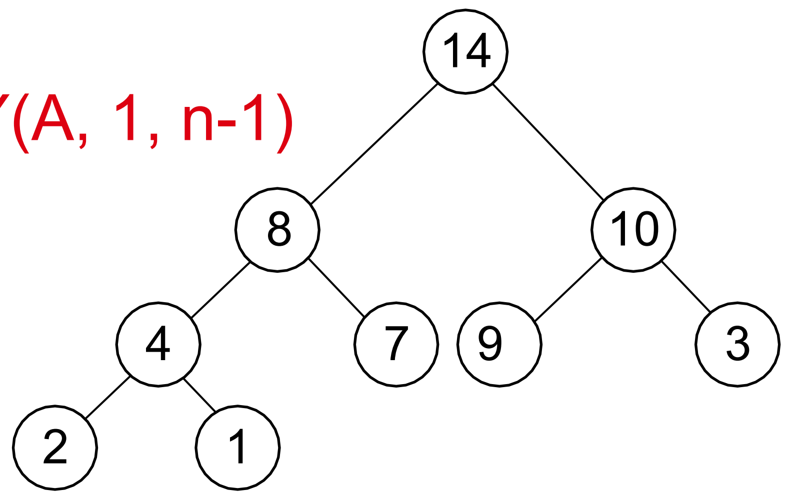


max = 16



Heap size decreased with 1

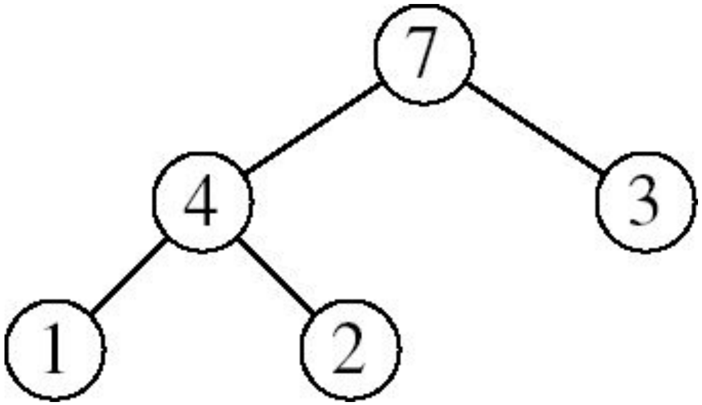
Call MAX-HEAPIFY(A, 1, n-1)



HEAP-EXTRACT-MAX

HEAP-EXTRACT-MAX(A, n)

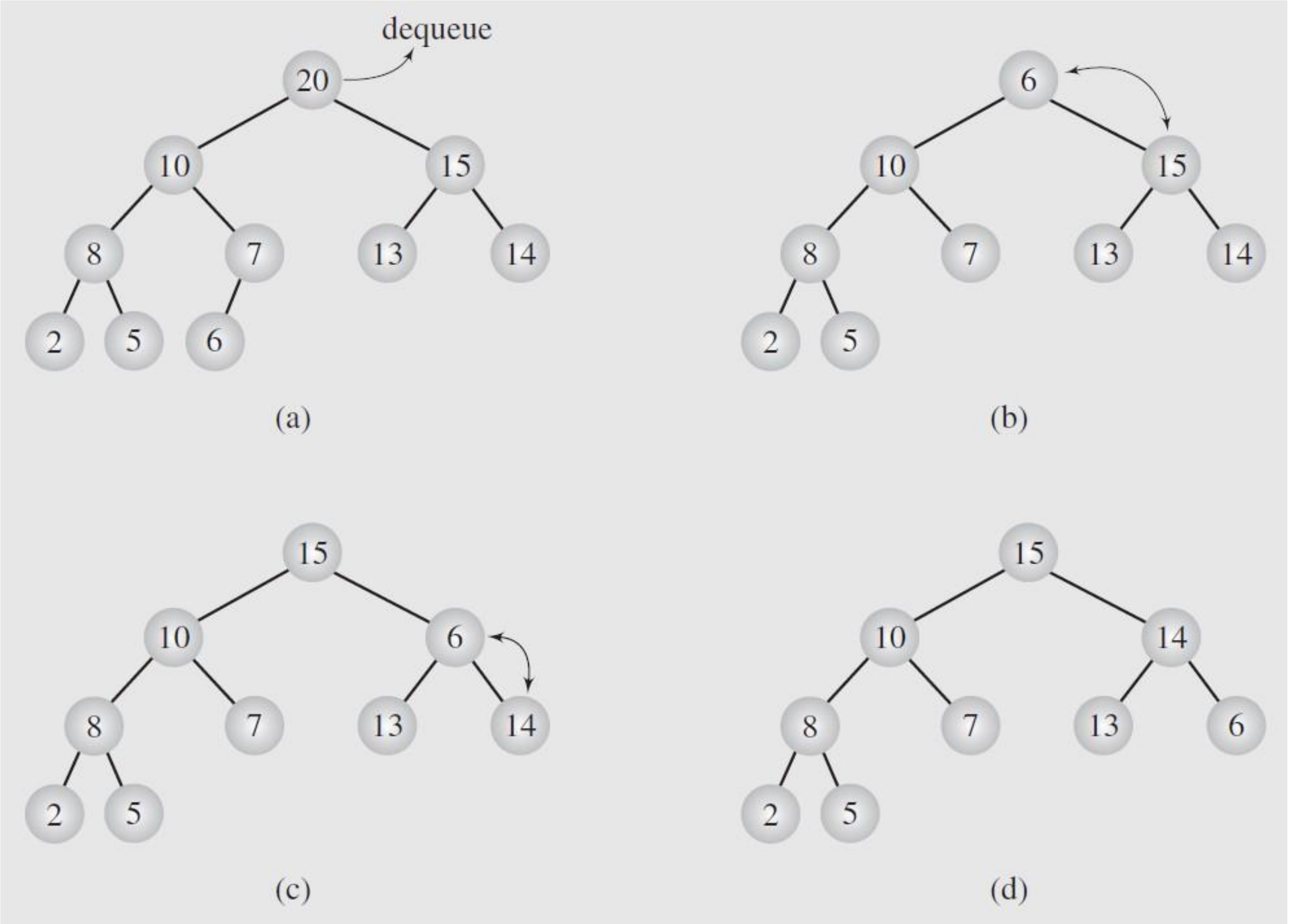
1. **if** $n < 1$
2. **error** "heap underflow"
3. $max = A[1]$
4. $A[1] = A[n]$
5. MAX-HEAPIFY(A, 1, n-1) // remakes heap
6. **return** max



Running time: **O(log *n*)** since it performs only a constant amount of work on top of the O(log n) time for MAX-HEAPIFY

Dequeue

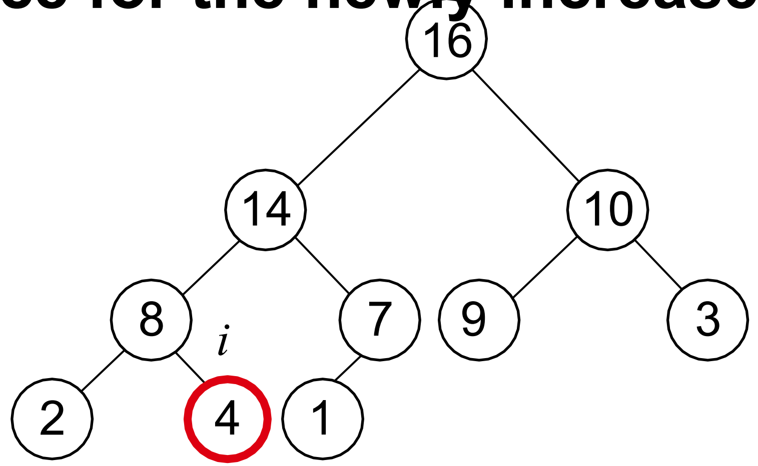
NUST SEECS



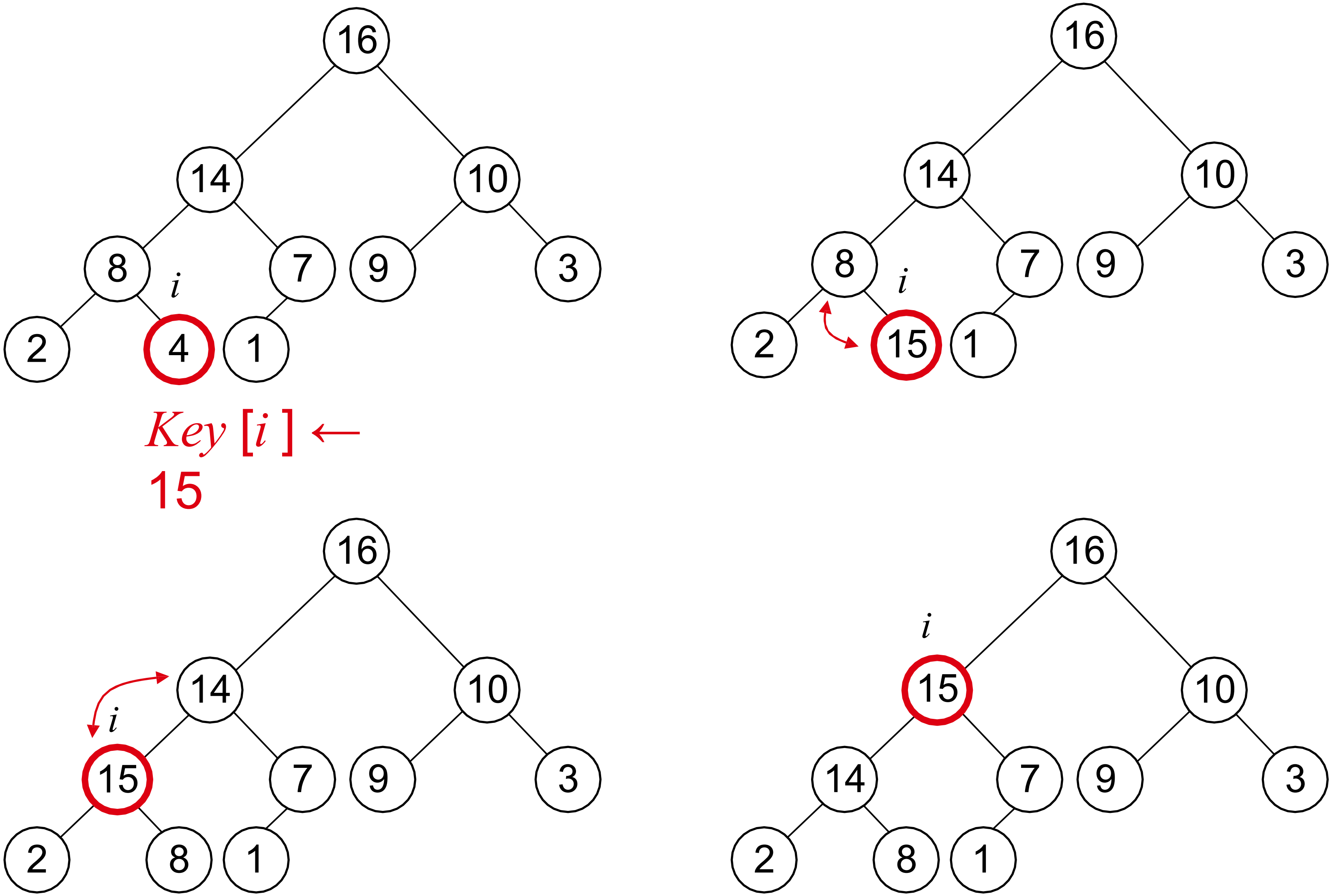
HEAP-INCREASE-KEY

- **Goal:**
  - Increases the key of an element *i* in the heap we wish to increase
- **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



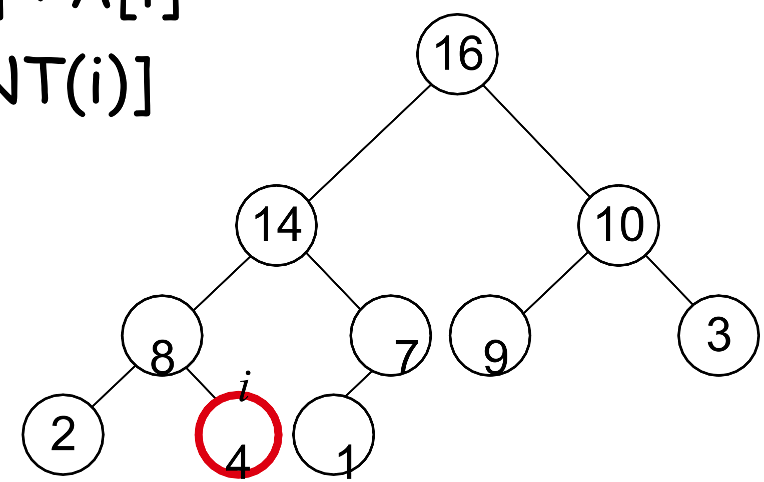
Example: HEAP-INCREASE-KEY



HEAP-INCREASE-KEY

HEAP-INCREASE-KEY(*A*, i, key)

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



Key [i] ← 15

Running time: **O(log *n*)** since the path traced from the node updated in line 3 to the root has length O(log *n*)