Algorithms
Chapter 6 Heapsort

Assistant Professor: Ching-Chi Lin
林清池 助理教授
chingchi.lin@gmail.com

Department of Computer Science and Engineering
National Taiwan Ocean University

Outline

- **Heaps**
- Maintaining the heap property
- Building a heap
- The heapsort algorithm
- Priority queues

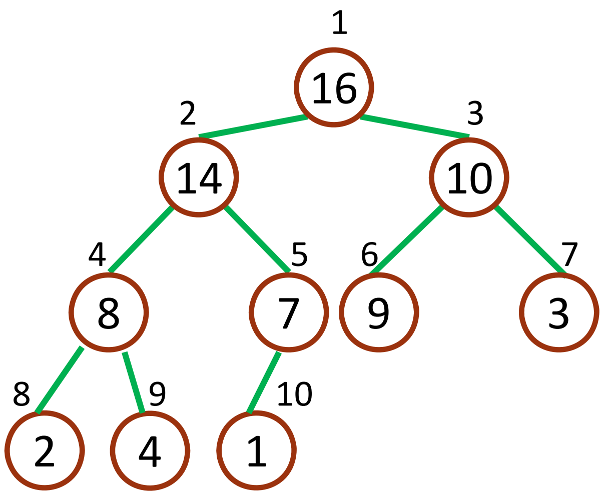
The purpose of this chapter

- In this chapter, we introduce the **heapsort** algorithm.
  - with worst case running time $O(n \lg n)$
  - an **in-place** sorting algorithm: only a constant number of array elements are stored outside the input array at any time.
  - thus, require at most $O(1)$ additional memory

- We also introduce the **heap** data structure.
  - an useful data structure for heapsort
  - makes an efficient priority queue

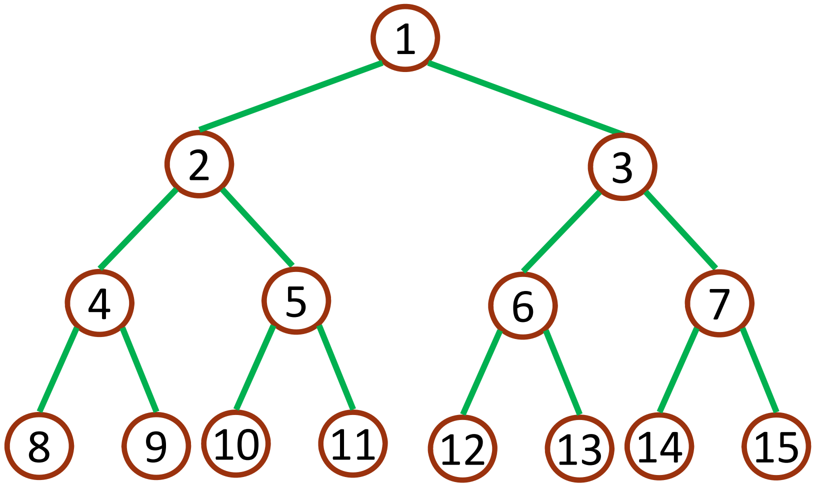
Heaps

- The (**Binary**) **heap** data structure is an **array** object that can be viewed as a nearly complete binary tree.
  - A binary tree with *n* nodes and depth *k* is **complete** iff its nodes correspond to the nodes numbered from 1 to *n* in the full binary tree of depth *k*.

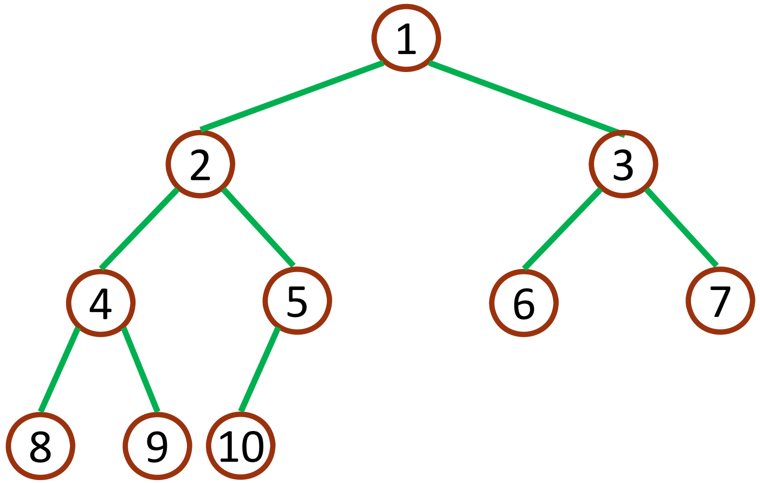


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Binary tree representations



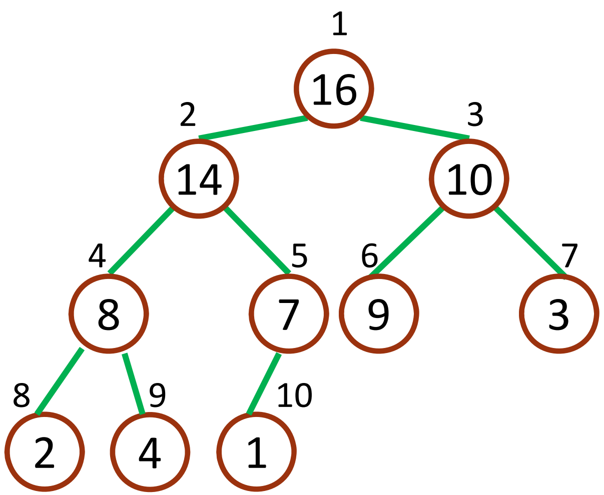
A full binary tree of height 3.



A complete binary tree with 10 nodes and height 3.

Attributes of a Heap

- An array A that presents a heap with two attributes:
  - **length[A]:** the number of elements in the array.
  - **heap-size[A]:** the number of elements in the heap stored with array A.
  - **length[A] $\geq$ heap-size[A]**

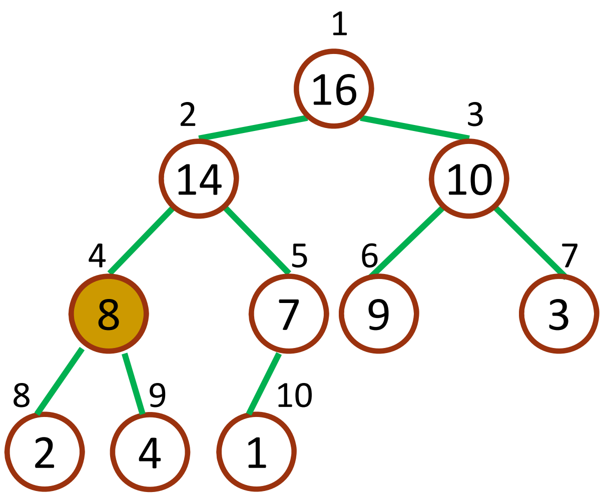


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

length[A] = heapsize[A] = 10

Basic procedures$_{1/2}$

- If a complete binary tree with *n* nodes is represented sequentially, then for any node with index *i*, $1 \le i \le n$, we have
  - A[1] is the **root** of the tree
  - the parent **PARENT(*i*)** is at $\lfloor i/2 \rfloor$ if $i \neq 1$
  - the left child **LEFT(*i*)** is at $2i$
  - the right child **RIGHT(*i*)** is at $2i+1$



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Basic procedures$_{2/2}$

- The **LEFT** procedure can compute 2*i* in one instruction by simply shifting the binary representation of *i* left one bit position.
- Similarly, the **RIGHT** procedure can quickly compute 2*i*+1 by shifting the binary representation of *i* left one bit position and adding in a 1 as the low-order bit.
- The **PARENT** procedure can compute $\lfloor i/2 \rfloor$ by shifting *i* right one bit position.

Heap properties

- There are two kind of binary heaps: max-heaps and min-heaps.
  - In a **max-heap**, the **max-heap property** is that for every node *i* other than the root,

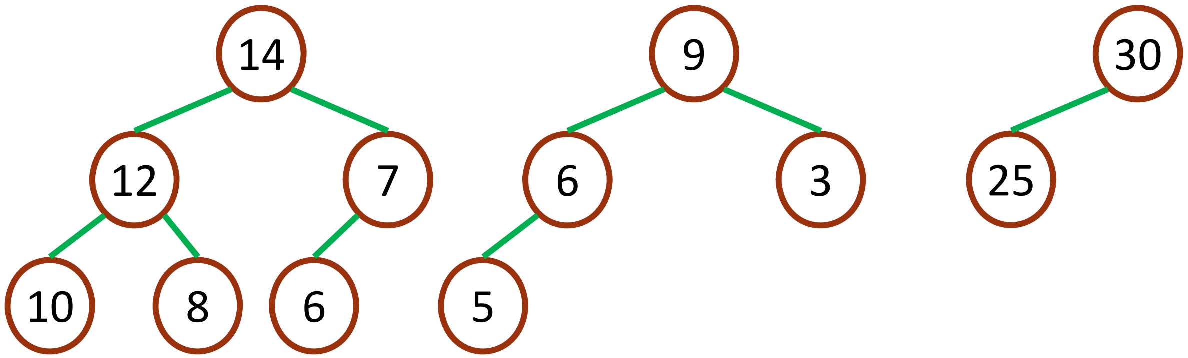
    $$A[\textsc{Parent}(i)] \geq A[i] .$$

    - the largest element in a max-heap is stored at the root
    - the subtree rooted at a node contains values no larger than that contained at the node itself
  - In a **min-heap**, the **min-heap property** is that for every node *i* other than the root,

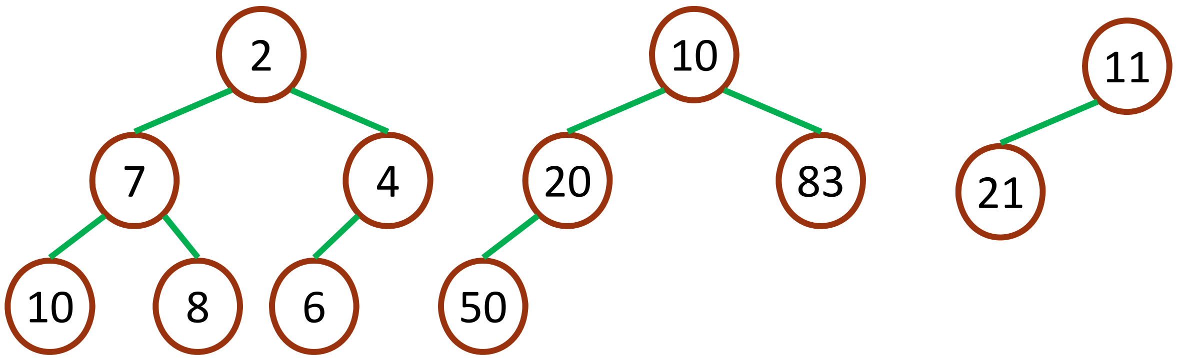
    $$A[\textsc{Parent}(i)] \leq A[i] .$$

    - the smallest element in a min-heap is at the root
    - the subtree rooted at a node contains values no smaller than that contained at the node itself

Max and min heaps



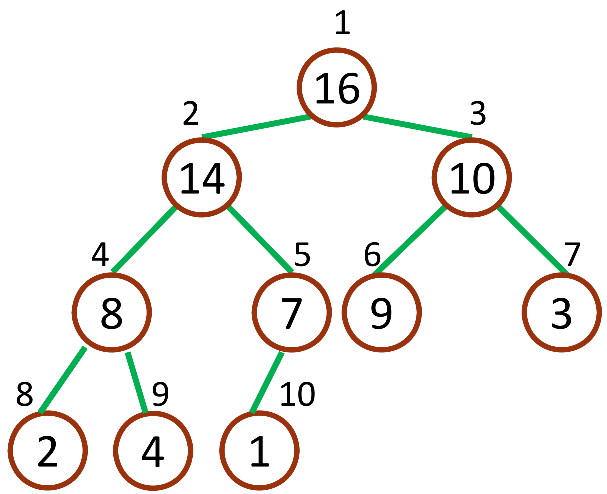
Max Heaps



Min Heaps

The height of a heap

- The **height** of a node in a heap is the number of edges on the longest simple downward path from the node to a leaf, and the height of the heap to be the height of the root, that is $\Theta(\lg n)$.
- For example:
  - the height of node 2 is 2
  - the height of the heap is 3



The remainder of this chapter

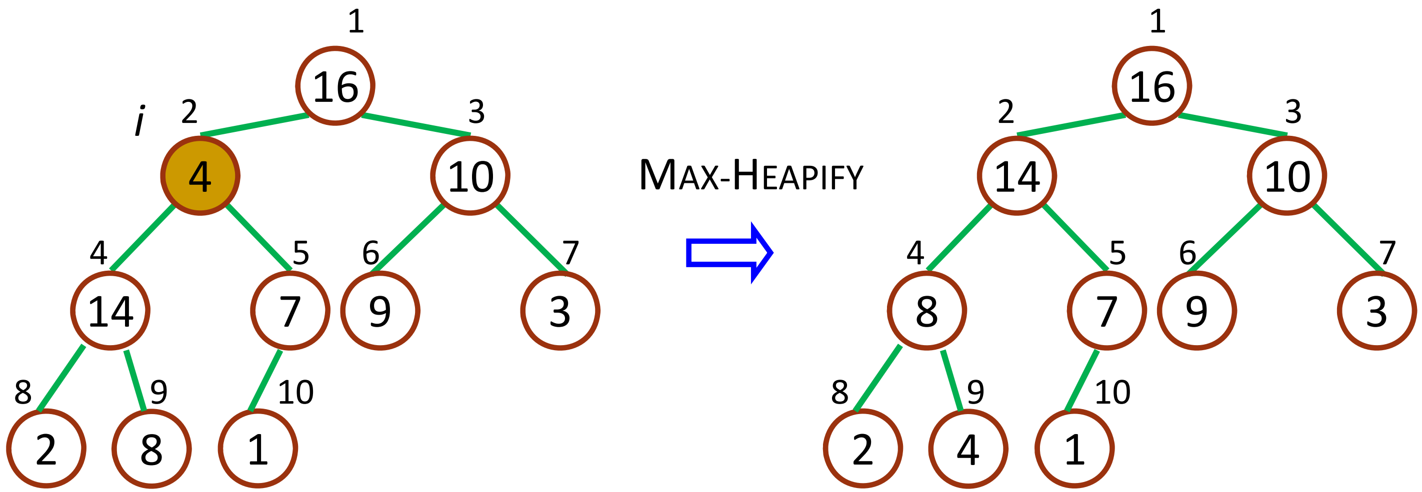
- We shall presents some basic procedures in the remainder of this chapter.
  - The **MAX-HEAPIFY** procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
  - The **BUILD-MAX-HEAP** procedure, which runs in $O(n)$ time, produces a max-heap from an unordered input array.
  - The **HEAPSORT** procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
  - The **MAX-HEAP-INSERT**, **HEAP-EXTRACT-MAX**, **HEAP-INCREASE-KEY**, and **HEAP-MAXIMUM** procedures, which run in $O(\lg n)$ time, allow the heap data structure to be used as a priority queue.

Outline

- Heaps
- **Maintaining the heap property**
- Building a heap
- The heapsort algorithm
- Priority queues

The MAX-HEAPIFY procedure$_{1/2}$

- MAX-HEAPIFY is an important subroutine for manipulating max heaps.
  - **Input**: an array A and an index *i*
  - **Output**: the subtree rooted at index *i* becomes a max heap
  - **Assume**: the binary trees rooted at LEFT(*i*) and RIGHT(*i*) are max-heaps, but A[i] may be smaller than its children
  - **Method**: let the value at A[i] "float down" in the max-heap

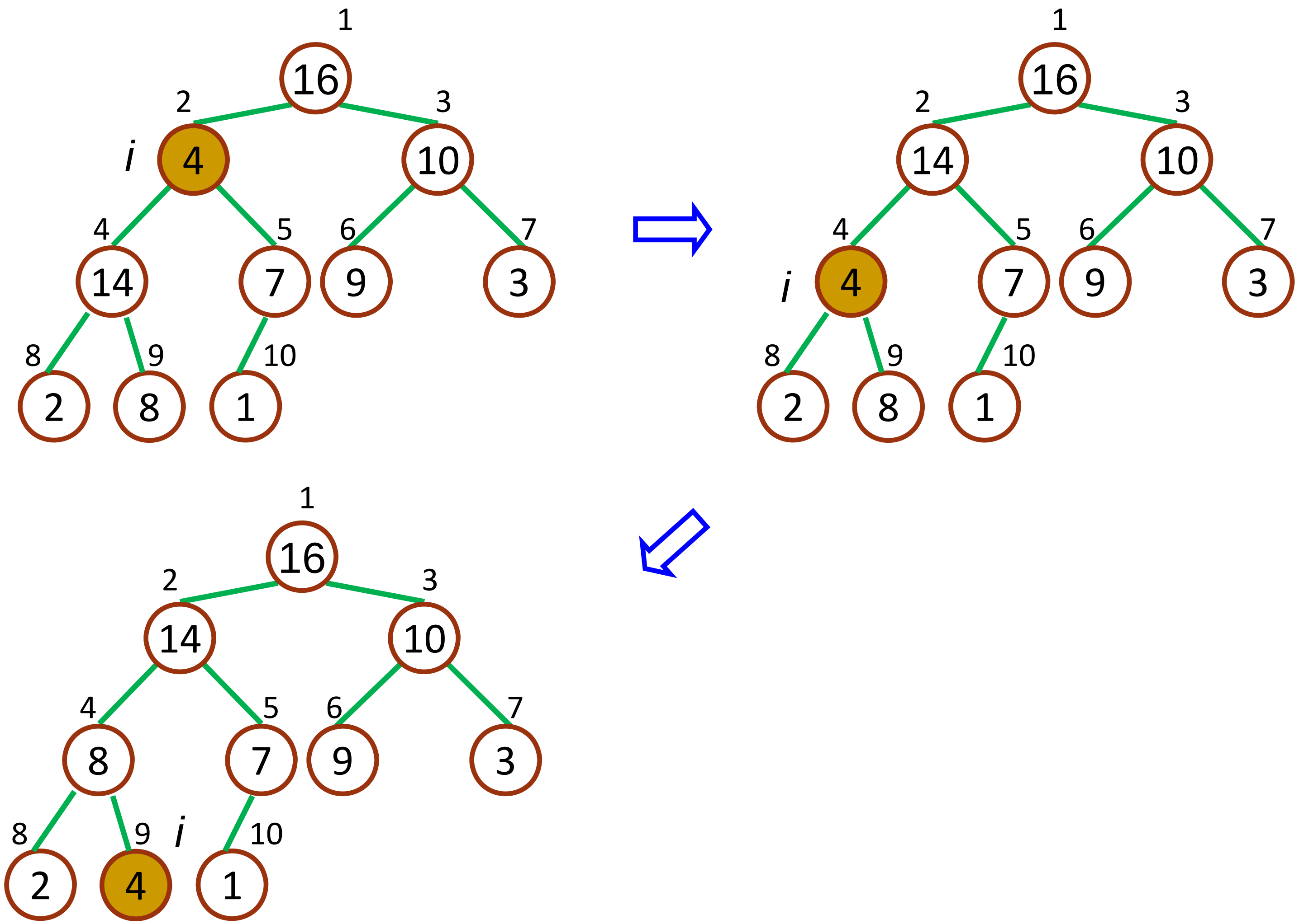


The MAX-HEAPIFY procedure$_{2/2}$

MAX-HEAPIFY(A, *i*)

$\ell \leftarrow$ LEFT(*i*)
r $\leftarrow$ RIGHT(*i*)
if $\ell \leq$ *heap-size*[A] and A[ℓ] > A[*i*]
    then *largest* $\leftarrow \ell$
    else *largest* $\leftarrow i$
if $r \leq$ *heap-size*[A] and a[*r*] > A[*largest*]
    then *largest* $\leftarrow r$
if *largest* $\neq i$
    then exchange A[*i*] $\leftrightarrow$ A[*largest*]
        MAX-HEAPIFY (A, *largest*)

An example of MAX-HEAPIFY procedure



The time complexity

- It takes $\Theta(1)$ time to fix up the relationships among the elements A[*i*], A[LEFT(*i*)], and A[RIGHT(*i*)].
- Also, we need to run MAX-HEAPIFY on a subtree rooted at one of the children of node *i*.
- The children's subtrees each have size at most $2n/3$
  - worst case occurs when the last row of the tree is exactly half full
- The running time of MAX-HEAPIFY is

$$\begin{aligned} T(n) &= T(2n/3) + \Theta(1) \\ &= O(\lg n) \end{aligned}$$

  - solve it by case 2 of the master theorem
- Alternatively, we can characterize the running time of MAX-HEAPIFY on a node of height h as $O(h)$.

Outline

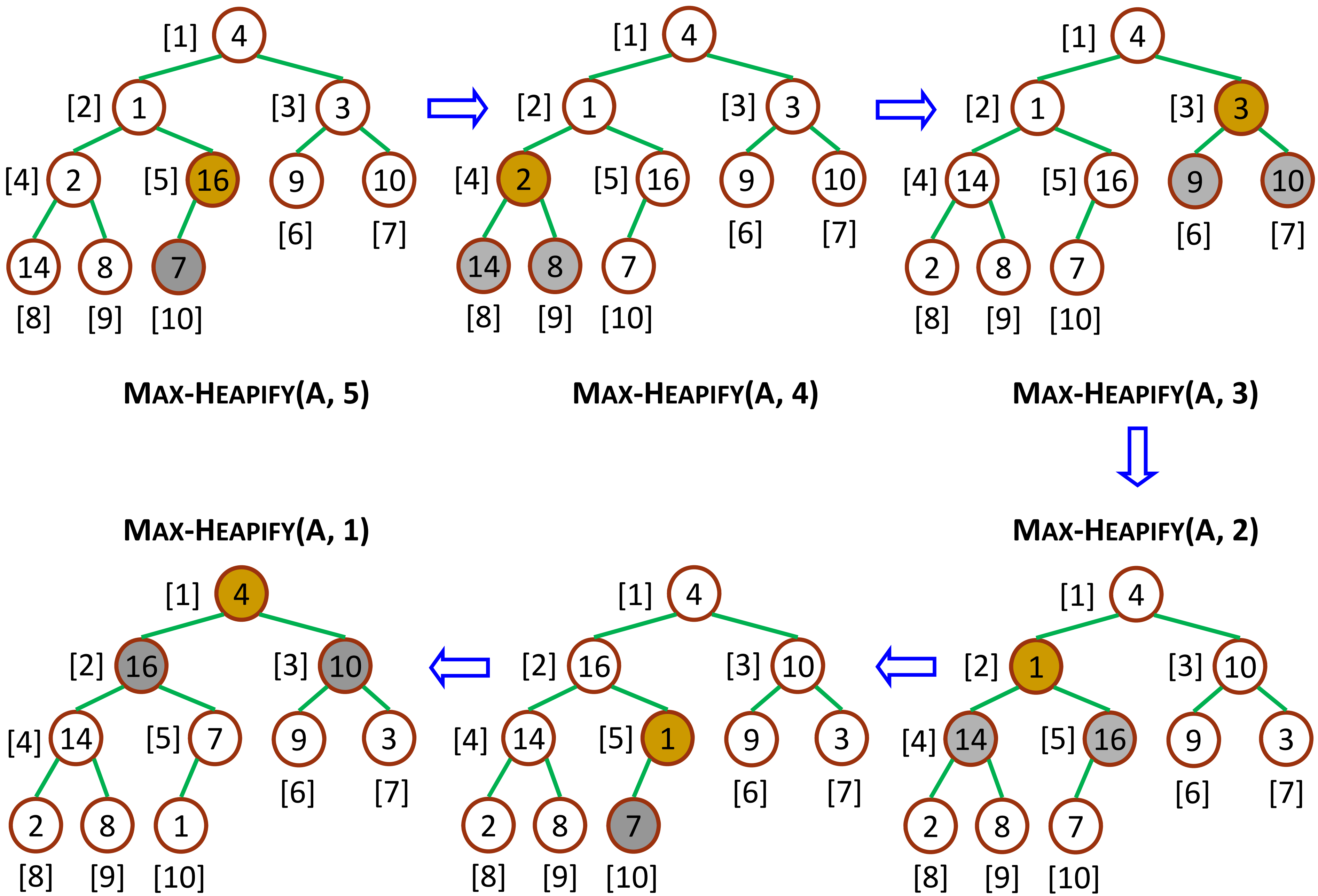
- Heaps
- Maintaining the heap property
- **Building a heap**
- The heapsort algorithm
- Priority queues

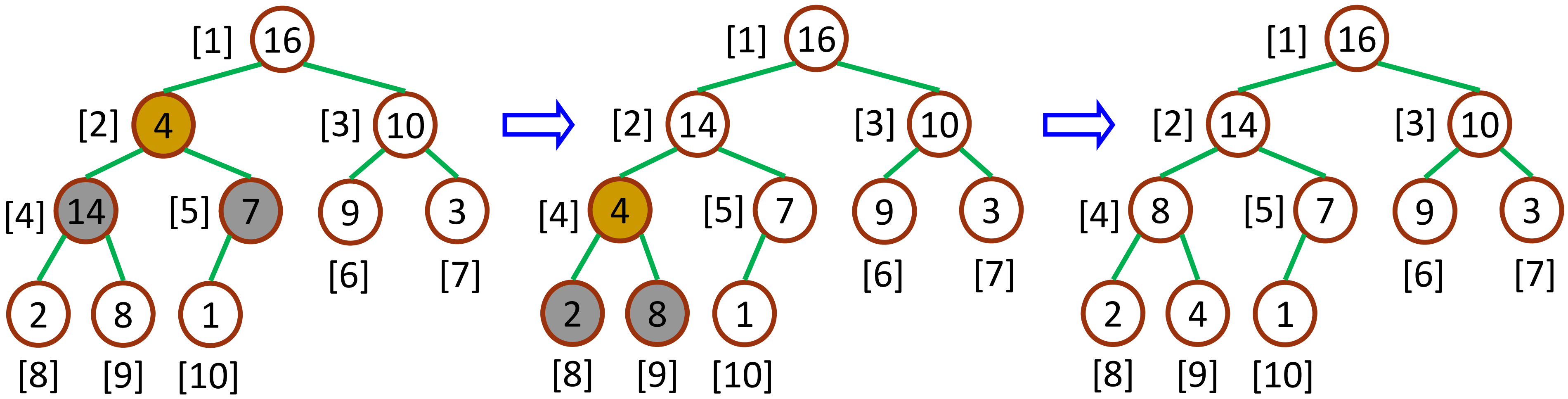
Building a Heap

- We can use the MAX-HEAPIFY procedure to convert an array A=[1..n] into a max-heap in a **bottom-up** manner.
- The elements in the subarray $A[(\lfloor n/2 \rfloor+1)\ldots n]$ are all **leaves** of the tree, and so each is a 1-element heap.
- The procedure BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one.

BUILD-MAX-HEAP(A)

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


MAX-HEAPIFY(A, 5)
MAX-HEAPIFY(A, 4)
MAX-HEAPIFY(A, 3)
MAX-HEAPIFY(A, 2)
MAX-HEAPIFY(A, 1)



[1] 16
[2] 4
[3] 10
[4] 14
[5] 7
9
3
[6]
[7]
2
8
1
[8]
[9]
[10]
[1] 16
[2] 14
[3] 10
[4] 4
[5] 7
9
3
[6]
[7]
2
8
1
[8]
[9]
[10]
[1] 16
[2] 14
[3] 10
[4] 8
[5] 7
9
3
[6]
[7]
2
4
1
[8]
[9]
[10]

max-heap

Correctness$_{1/2}$

- To show why BUILD-MAX-HEAP work correctly, we use the following **loop invariant:**
  - At the start of each iteration of the for loop of lines 2-3, each node $i+1$, $i+2$, …, n is the root of a max-heap.

BUILD-MAX-HEAP(A)

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

- We need to show that
  - this invariant is true prior to the first loop iteration
  - each iteration of the loop maintains the invariant
  - the invariant provides a useful property to show correctness when the loop terminates.

Correctness$_{2/2}$

- **Initialization**: Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. $\lfloor n/2 \rfloor+1, \ldots n$ is a leaf and is thus the root of a trivial max-heap.
- **Maintenance**: By the loop invariant, the children of node i are both roots of max-heaps. This is precisely the condition required for the call MAX-HEAPIFY(A, i) to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes $i + 1, i + 2, \ldots, n$ are all roots of max-heaps.
- **Termination**: At termination, i=0. By the loop invariant, each node 1, 2, ..., n is the root of a max-heap. In particular, node 1 is.

Time complexity$_{1/2}$

- **Analysis 1**:
  - Each call to MAX-HEAPIFY costs $O(\lg n)$, and there are $O(n)$ such calls.
  - Thus, the running time is $O(n \lg n)$. This upper bound, through correct, is **not asymptotically tight**.
- **Analysis 2**:
  - For an n-element heap, height is $\lfloor \lg n \rfloor$ and at most $\lceil n / 2^{h+1} \rceil$ nodes of any height h.
  - The time required by MAX-HEAPIFY when called on a node of height h is $O(h)$.
  - The total cost is $\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$.

Time complexity$_{2/2}$

- The last summation yields

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

- Thus, the running time of BUILD-MAX-HEAP can be bounded as

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n)$$

- We can build a max-heap from an unordered array in **linear time**.

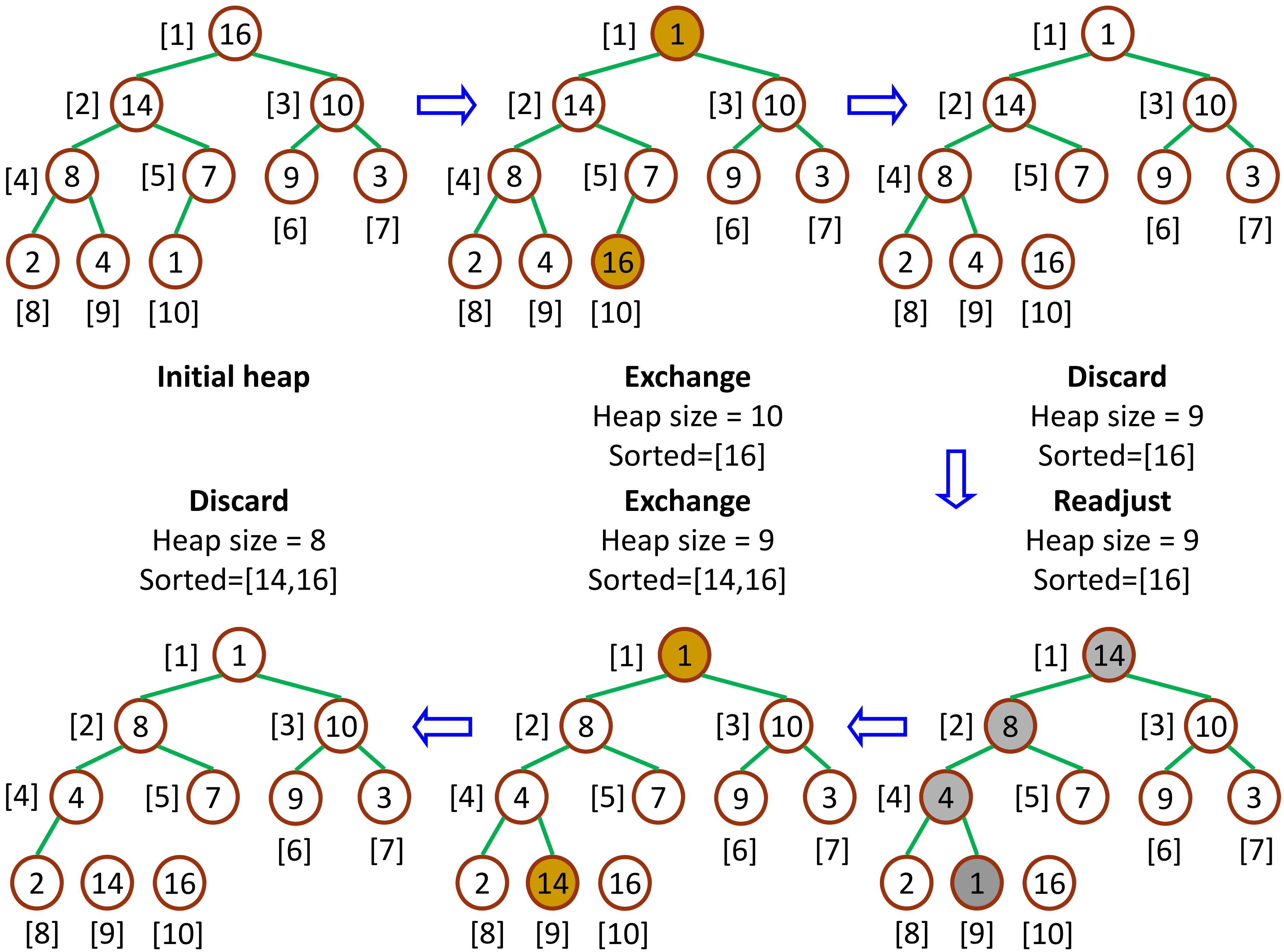
Outline

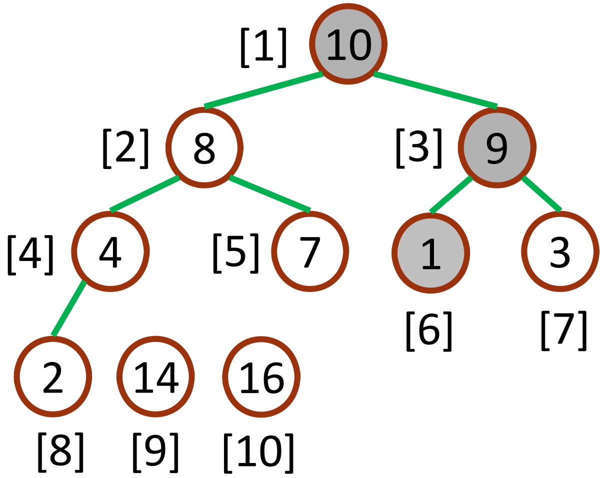
- Heaps
- Maintaining the heap property
- Building a heap
- **The heapsort algorithm**
- Priority queues

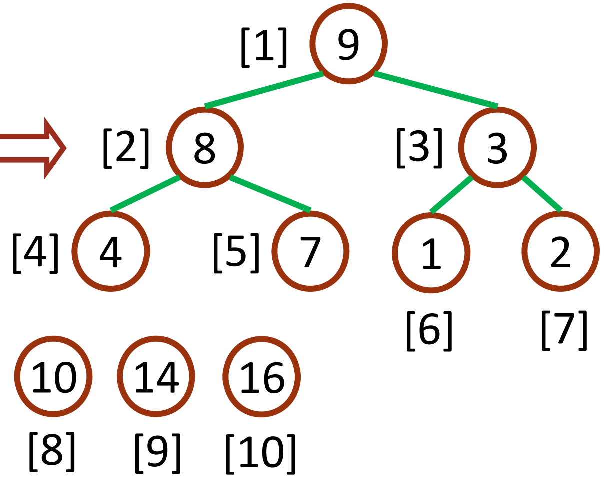
The heapsort algorithm

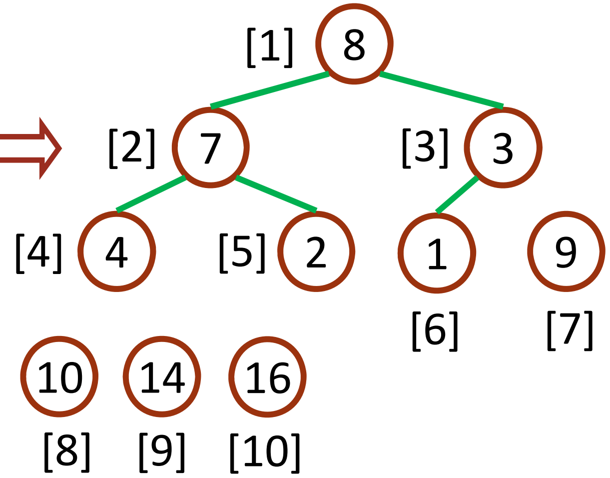
- Since the maximum element of the array is stored at the root, A[1] we can **exchange** it with $A[n]$.
- If we now "**discard**" $A[n]$, we observe that $A[1...(n-1)]$ can easily be made into a max-heap.
- The children of the root $A[1]$ remain max-heaps, but the new root $A[1]$ element may violate the max-heap property, so we need to **readjust** the max-heap. That is to call MAX-HEAPIFY(A, 1).

```
HEAPSORT(A)
    BUILD-MAX-HEAP(A)
    for i ← length[A]  downto 2
        do exchange A[1] ↔ A[i]
            heap-size[A] ← heap-size[A] –1
            MAX-HEAPIFY(A, 1)
```


Initial heap
Exchange
Heap size = 10
Sorted=[16]
Discard
Heap size = 9
Sorted=[16]
Readjust
Heap size = 9
Sorted=[16]
Exchange
Heap size = 9
Sorted=[14,16]
Discard
Heap size = 8
Sorted=[14,16]







Readjust
Heap size = 8
Sorted=[14,16]

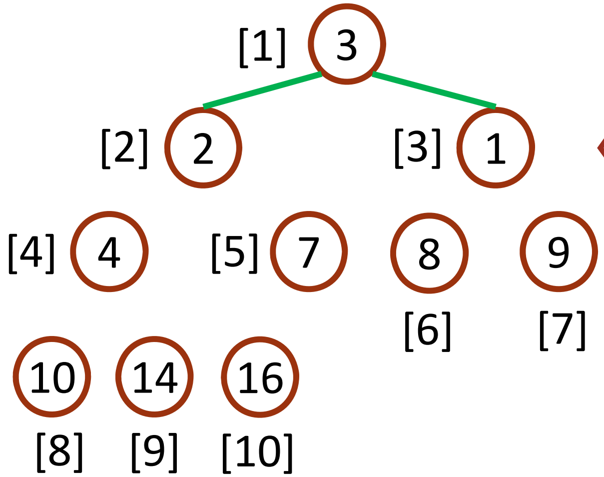
Heap size = 7
Sorted=[10,14,16]

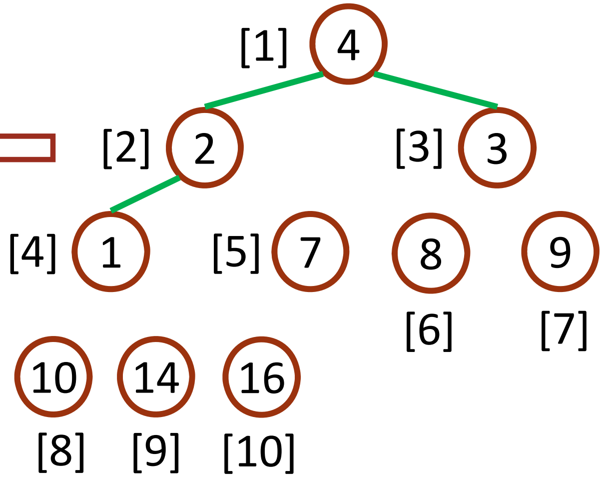
Heap size = 6
Sorted=[9,10,14,16]

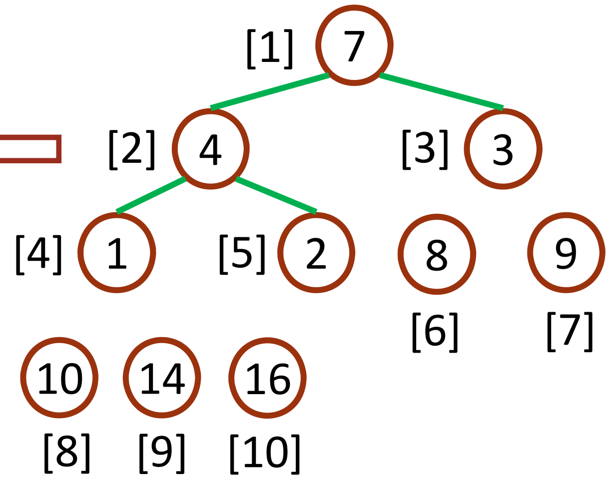
Heap size = 3
Sorted=[4,7,8,9,10,14,16]

Heap size = 4
Sorted=[7,8,9,10,14,16]

Heap size = 5
Sorted=[8,9,10,14,16]







Time complexity

- The HEAPSORT procedure takes ***O*(*n*lg*n*)** time
  - the call to BUILD-MAX-HEAP takes $O(n)$ time
  - each of the $n-1$ calls to MAX-HEAPIFY takes $O(\lg n)$ time

Outline

- Heaps
- Maintaining the heap property
- Building a heap
- The heapsort algorithm
- **Priority queues**

Heap implementation of priority queues

- Heaps efficiently implement priority queues.
- There are two kinds of priority queues: 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.
- A **priority queue** is a data structure for maintaining a set *S* of elements, each with an associated value called a **key**.

Priority queues

- A **max-priority queue** supports the following operations.
  - INSERT(S, x): inserts the element x into the set S.
  - 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 value of element x's key to the new value k. Assume $k \geq x$'s current key value.

Finding the maximum element

- MAXIMUM(S): returns the element of S with the largest key.
- Getting the maximum element is easy: it's the root.

HEAP-MAXIMUM(A)

1. return A[1]

- The running time of HEAP-MAXIMUM is $\Theta(1)$.

Extracting max element

- EXTRACT-MAX(S): removes and returns the element of S with the largest key.

HEAP-EXTRACT-MAX(A)

if *heap-size*[A] < 1
    then error "heap underflow"
max $\leftarrow$ A[1]
A[1] $\leftarrow$ A[*heap-size*[A]]
heap-size[A] $\leftarrow$ *heap-size*[A] $-$ 1
MAX-HEAPIFY(A, 1)
return *max*

- **Analysis**: constant time assignments + time for MAX-HEAPIFY.

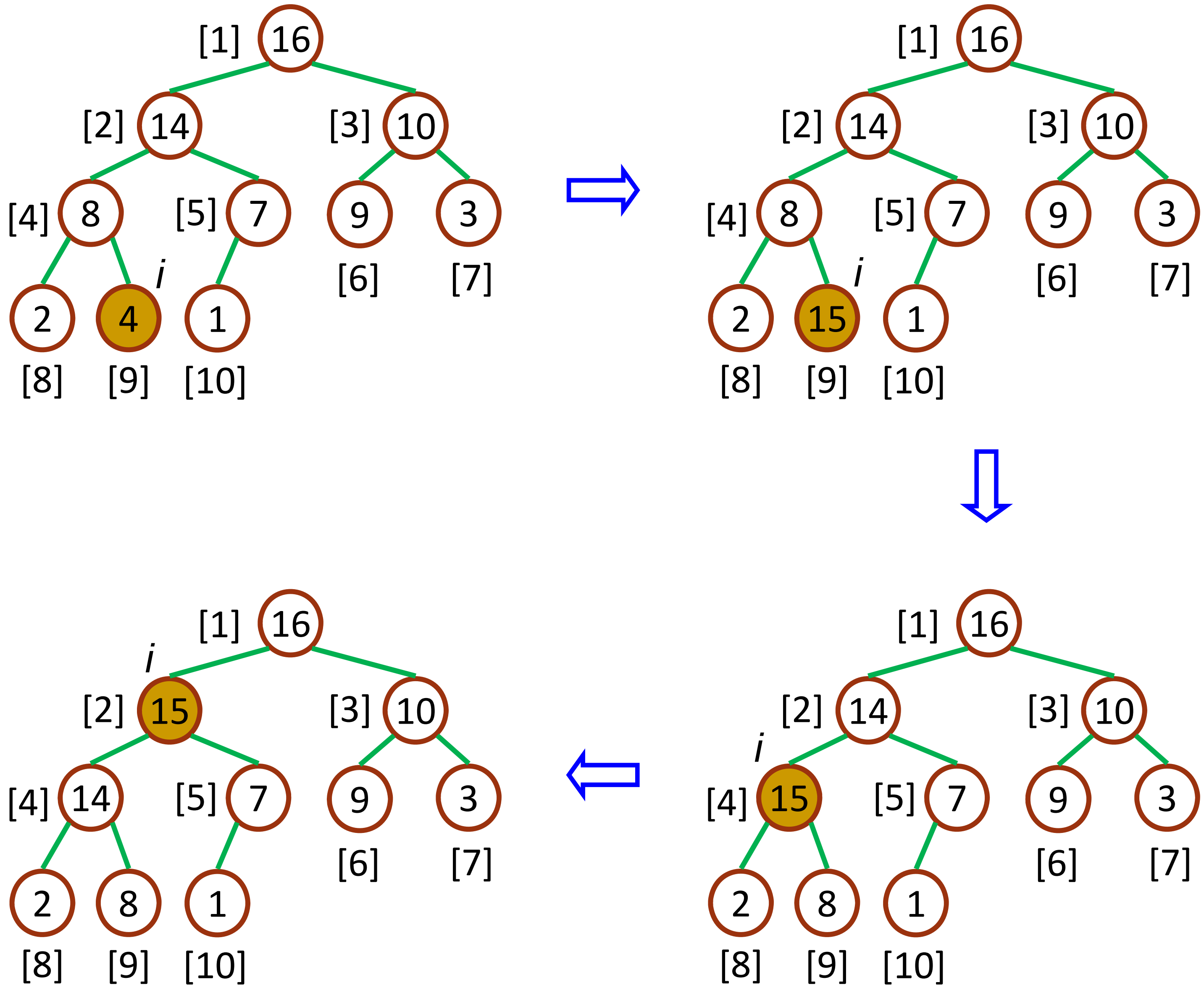
- The running time of HEAP-EXTRACT-MAX is $O(\lg n)$.

Increasing key value

- INCREASE-KEY(S, x, k): increases value of element x's key to k.
  Assume $k \geq x$'s current key value.

```
HEAP-INCREASE-KEY (A, i, key)
    if key < A[i]
        then error "new key is smaller than current key"
    A[i] ← key
    While i > 1 and A[PARENT(i)] < A[i]
        do exchange A[i] ↔ A[PARENT(i)]
            i ← PARENT(i)
```

- **Analysis**: the path traced from the node updated to the root has length $O(\lg n)$.
- The running time is $O(\lg n)$.



[1] 16
[2] 14
[3] 10
[4] 8
[5] 7
9
3
[6]
[7]
i
2
4
1
[8]
[9]
[10]
[1] 16
[2] 14
[3] 10
[4] 8
[5] 7
9
3
[6]
[7]
i
2
15
1
[8]
[9]
[10]
[1] 16
i
[2] 15
[3] 10
[4] 14
[5] 7
9
3
[6]
[7]
2
8
1
[8]
[9]
[10]
[1] 16
[2] 14
[3] 10
i
[4] 15
[5] 7
9
3
[6]
[7]
2
8
1
[8]
[9]
[10]

Inserting into the heap

- INSERT(S, x): inserts the element x into the set S.

MAX-HEAP-INSERT(A, *key*)

1. *heap-size*[A] $\leftarrow$ *heap-size*[A]+1
2. A[*heap-size*[A] $\leftarrow -\infty$
3. HEAP-INCREASE-KEY(A, *heap-size*[A], *key*)

- **Analysis**: constant time assignments + time for HEAP-INCREASE-KEY.
- The running time is $O(\lg n)$.