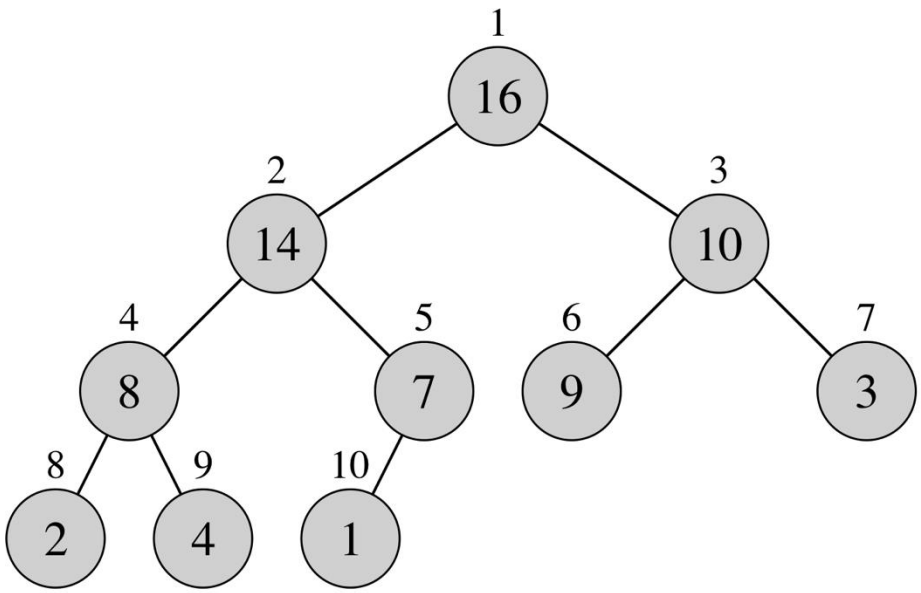
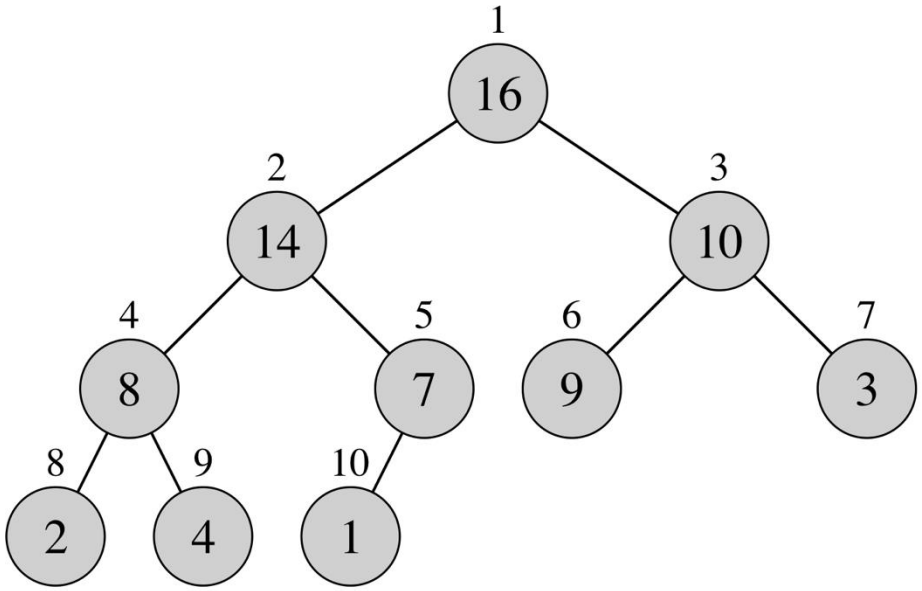
Heaps & heapsort

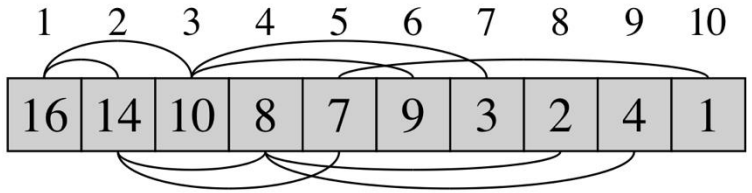
CLRS 6.1 – 6.5



(Max) Heaps

- A (binary) **heap** is an array object *A* that we can view as a nearly complete binary tree, where the **root is at $A[1]$**
- For any given node at index i, other related nodes can be found
  - *Parent*: at index $\left\lfloor \frac{i}{2} \right\rfloor$
  - *Left child*: at index $2i$
  - *Right child*: at index $2i + 1$
- **Properties**:
  - **Nearly complete binary tree**: tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point
  - **Max-heap property**: for every node i other than the root, $A[\text{Parent}(i)] \geq A[i]$



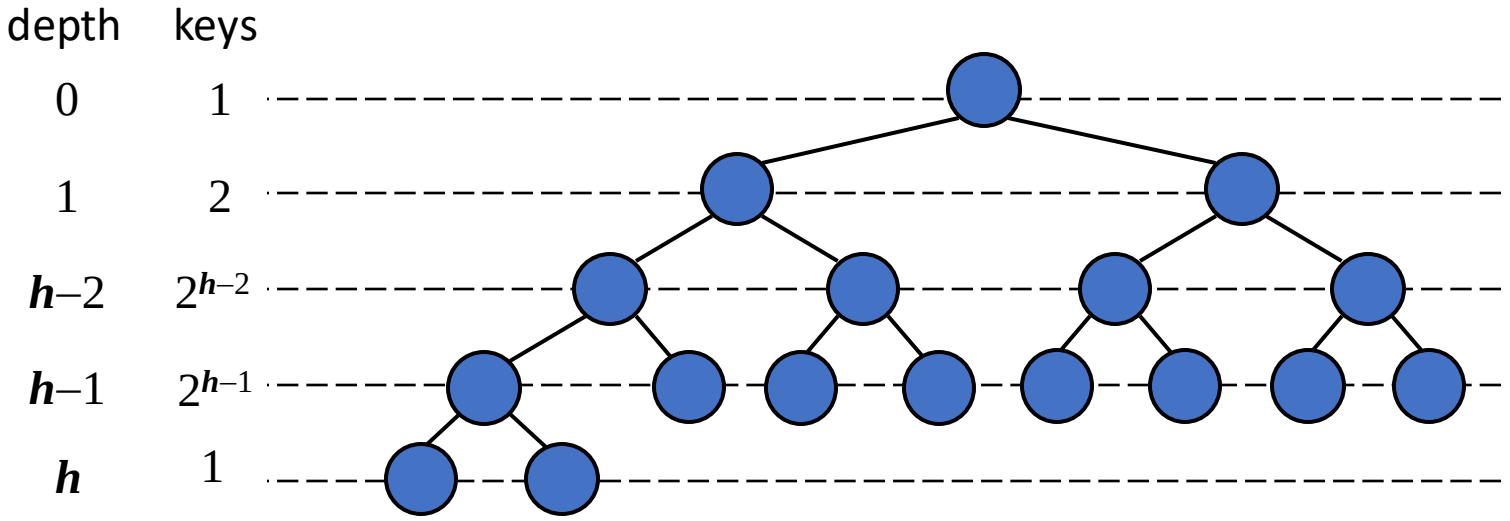


Heap height

Theorem: A heap containing $\boldsymbol{n}$ elements has height $O(\log n)$.

Proof:

- Let h be the height of a heap storing n keys.
- Since there are 2^i keys at depth $i = 0, \ldots, h-1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$.
- Thus, $n \geq 2^h$ and therefore $h \leq \log n$.



Heap operations

$O(\log n)$
- Max-Heap-Insert: *insert into heap*
- Heap-Extract-Max: *remove and return item with max key*
- Heap-Increase-Key: *increase value of particular key*
- **Max-Heapify**: *maintain max-heap property*

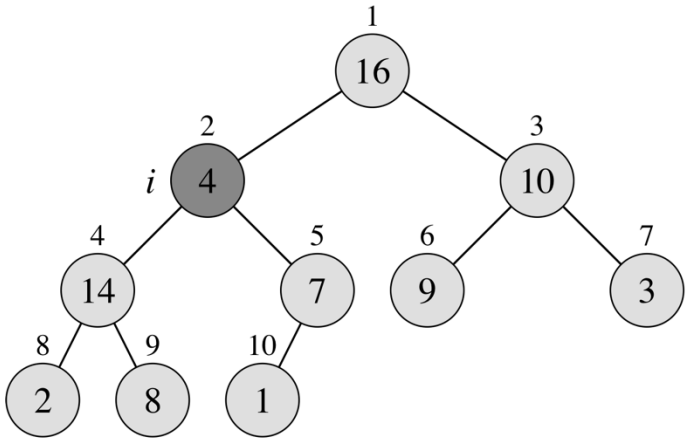
$O(1)$
- Heap-Maximum: *return (but do not remove) item with max key*

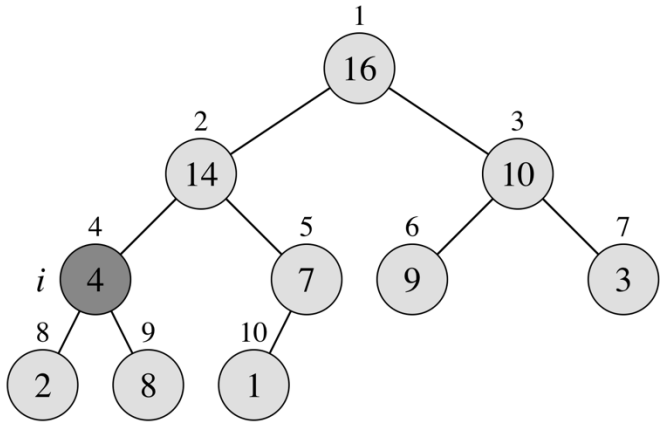
$O(n)$
- Build-Max-Heap: *construct a max-heap from an array of keys*

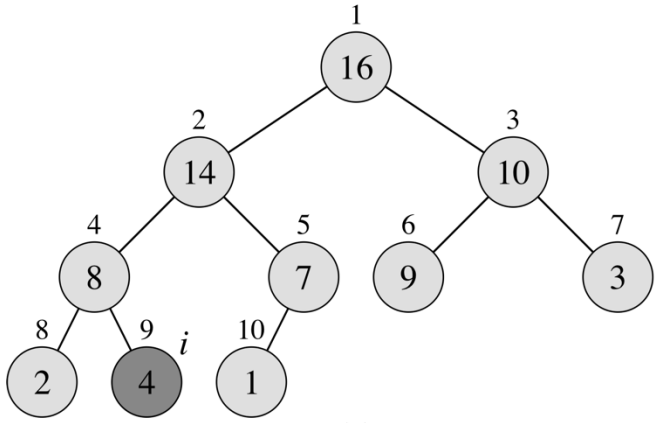
$O(n \log n)$
- Heapsort: *use a heap to sort an array of keys*

Max-Heapify

- Works on a particular node at index i.
- Assumption: Binary trees rooted at Left(i) and Right(i) are max-heaps, but possibly the node at index i might violate the max-heap property.
- **Idea**: While the max-heap property is violated, fix it by **floating down** the node







MAX-HEAPIFY(A, i) $O(\log n)$

```
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)
```

Inserting a single element

- Place it at the end of the array (next empty node of tree)
- While the max-heap property is violated, fix it by **floating up** the node.

```
MAX-HEAP-INCREASE-KEY(A, x, k)
1  if k < x.key                                          6.5 Priority queues
2        error "new key is smaller than current key"
3  x.key = k
4  find the index i in array A where object x occurs
5  while i > 1 and A[PARENT(i)].key < A[i].key
6        exchange A[i] with A[PARENT(i)], updating the information that maps
               priority queue objects to array indices
7        i = PARENT(i)

MAX-HEAP-INSERT(A, x, n)
1  if A.heap-size == n
2        error "heap overflow"
3  A.heap-size = A.heap-size + 1
4  k = x.key
5  x.key = –∞
6  A[A.heap-size] = x
7  map x to index heap-size in the array
8  MAX-HEAP-INCREASE-KEY(A, x, k)
```

Visualization ("insert"): https://gallery.selfboot.cn/en/algorithms/heap

Extracting the maximum

- Remove the maximum (known to be the root node at A[1])
- Exchange it with the last item
- Fix the max-heap property

$$O(\log n)$$

```
MAX-HEAP-EXTRACT-MAX(A)
1  max = MAX-HEAP-MAXIMUM(A)
2  A[1] = A[A.heap-size]
3  A.heap-size = A.heap-size – 1
4  MAX-HEAPIFY(A, 1)
5  return max
```

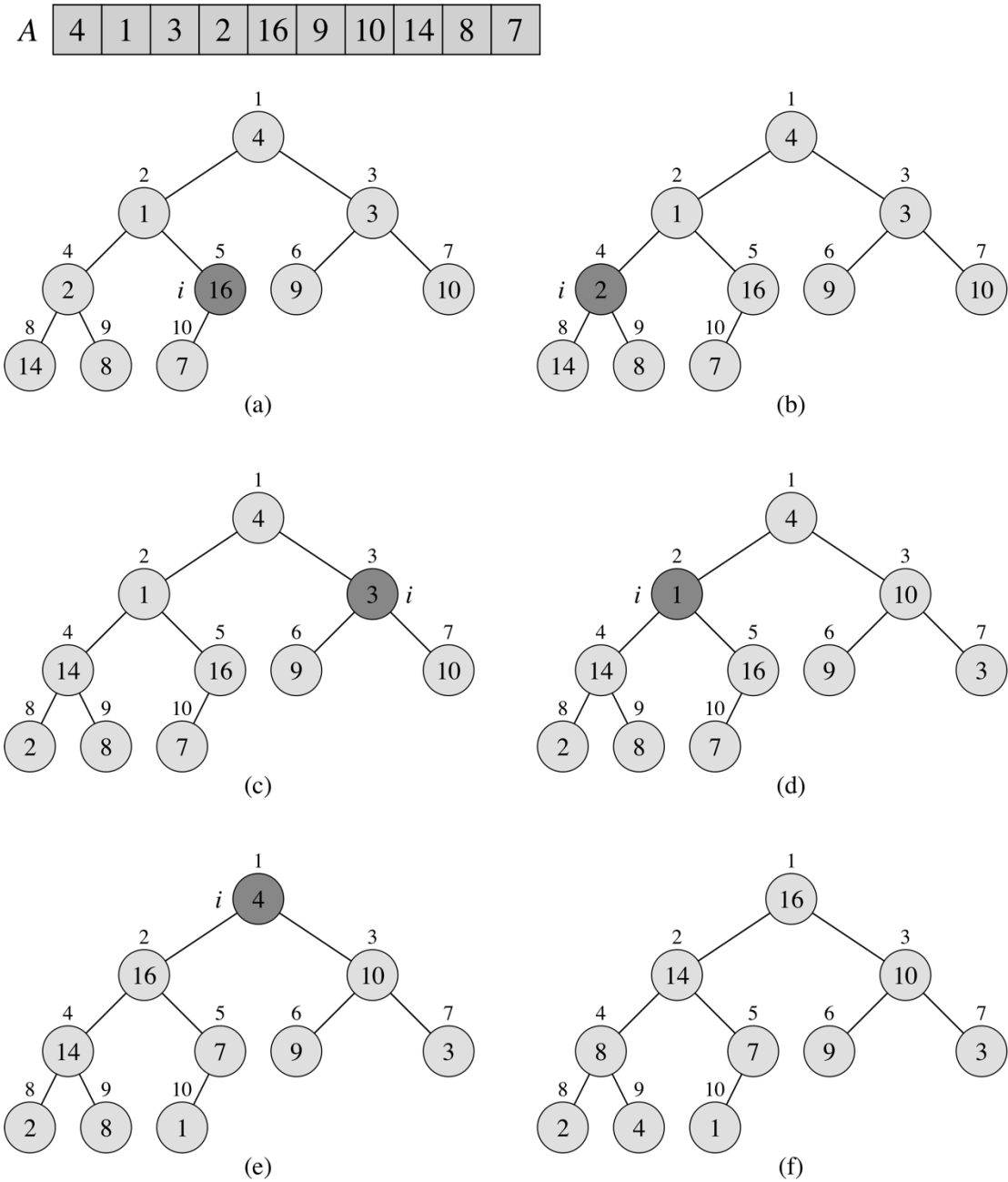
Visualization (“extract max”): https://gallery.selfboot.cn/en/algorithms/heap

Constructing a heap from an array of elements

- Take advantage of the fact that all elements are known in advance
- Repeatedly use max-heapify

$O(n)$

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



Visualization (“build heap”): https://visualizer.siddhartha-chatterjee.com/visualize/Heap

Heapsort

- Efficiently build a heap
- Repeatedly remove the maximum item, placing it at the end of the array
- Repeat process with remaining part of the heap

$O(n \log n)$

HEAPSORT(A, n)

```
1  BUILD-MAX-HEAP(A, n)
2  for i = n downto 2
3        exchange A[1] with A[i]
4        A.heap-size = A.heap-size – 1
5        MAX-HEAPIFY(A, 1)
```

Visualization ("heap sort"): https://www.cs.usfca.edu/~galles/visualization/HeapSort.html

Application to Priority Queues

- A **priority queue** stores a collection of (key, element) pairs and supports
  - Insert
  - Maximum (Minimum)
  - Extract-Max (Extract-Min)

- **Easy to sort using a priority queue as auxiliary data structure**
  - Insert all items into priority queue
  - One-by-one, call extract-max and place item at the beginning of list

- This generic priority-queue approach to sorting encapsulates common sorting algorithms, depending on the **implementation** of the priority queue
  - Use heap → heapsort
  - Use unsorted list → selection sort
  - Use sorted list → insertion sort