Introduction to Algorithms

Mong-Jen Kao (高孟駿)

Tuesday 10:10 – 12:00

Thursday 15:30 – 16:20

Data Structures

Particular ways of storing data *to support special operations*.

Min- (Max-) Heap / Priority Queue

Storing semi-dynamic data to extract the minimum element fast.

Priority Queue

- Suppose that we want to maintain a set A of ***elements, each associated with a key,*** so as to support the following operations.
  - **Insert(A, x)** – to insert a given element x into A.
  - **Maximum(A)** – to return the largest element in A.
  - **Extract-Max(A)** – to remove and return the largest element from A.
  - **Increase-Key(A, x, k)**
    – to increase the value of the elements x's key to the new value k.

Priority Queue

With ***max-heap***,
these operations can be done in…

- Suppose that we want to maintain a set A of ***elements, each associated with a key,*** so as to support the following operations.
  - **Insert(A, x)** – to insert a given element x into A.
  - **Maximum(A)** – to return the largest element in A.
  - **Extract-Max(A)** – to remove and return the largest element from A.
  - **Increase-Key(A, x, k)**
    – to increase the value of the elements x's key to the new value k.

$O(\log n)$ time.

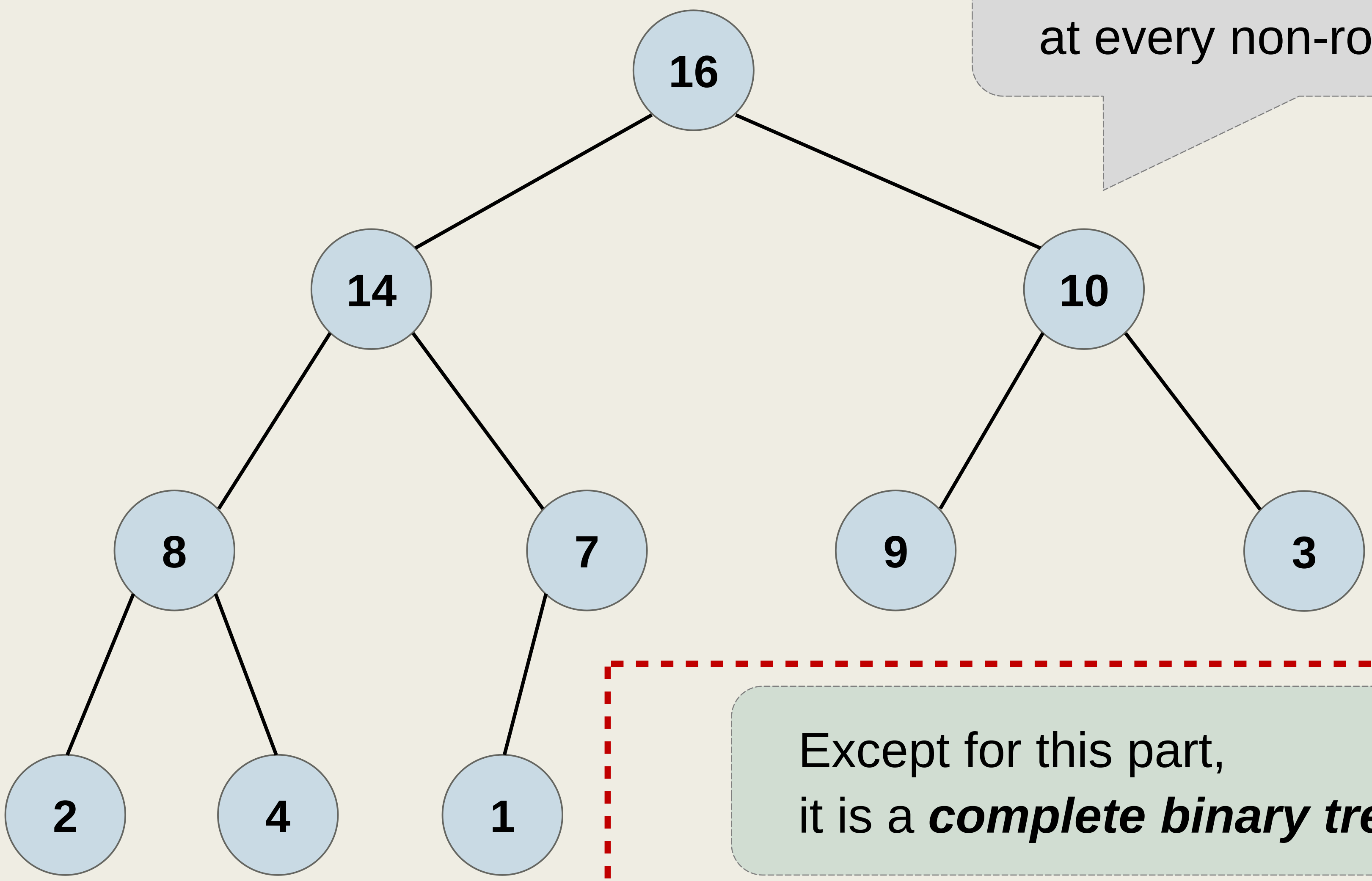
$O(1)$ time.

$O(\log n)$ time.

Maximum Heap

- The maximum heap is a ***nearly complete binary tree*** such that
  - *The nodes* in the tree *are comparable* to each other.
  - (*Max-Heap property*)
    For any non-root node v and its parent $p(v)$,
    we always have
    $$p(v) \geq v .$$

Maximum Heap



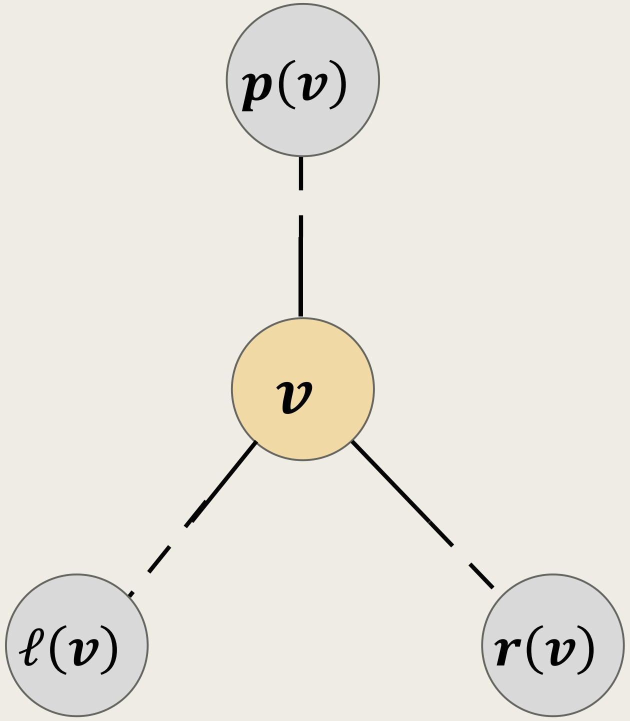
The max-heap property holds at every non-root vertex.

Except for this part,
it is a ***complete binary tree***.

Representing a Binary Tree

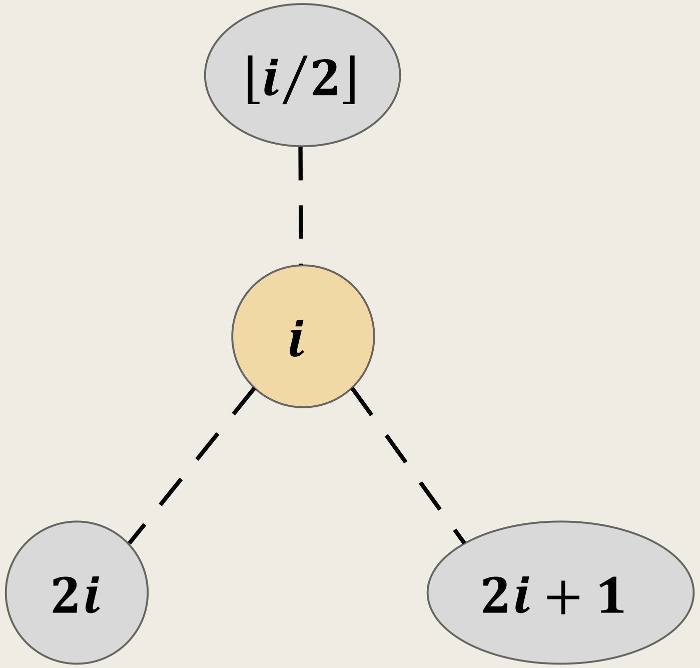
- **In general**, to record the structure of a binary tree $T = (V, E)$, for each node $v \in V$, we need to store the following information.
  - The parent node of v, denoted $p(v)$.
  - The left- and right- children nodes of v, denoted $\ell(v)$ and $r(v)$, respectively.

```
struct node {
    int val;
    node *p, *l, *r;
};
```

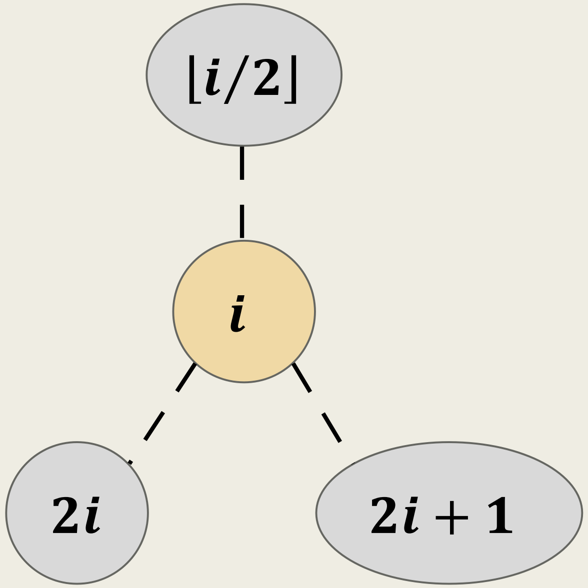


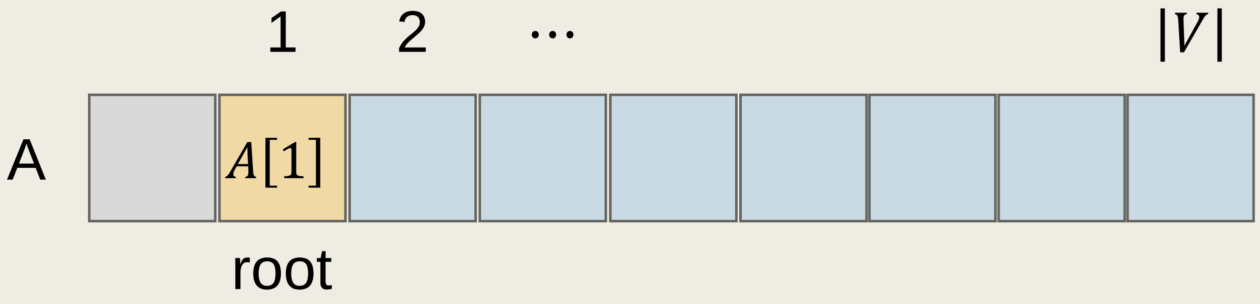
Representing a **Nearly-Complete** Binary Tree

- For a nearly-complete binary tree $T = (V, E)$,
  we can use **an array A of size $O(|V|)$** to represent it.
  - The root is $A[1]$.
  - Given an index $i \geq 1$,
    - Parent$(i) \coloneqq \lfloor i/2 \rfloor$.
    - Left$(i) \coloneqq 2i$.
    - Right$(i) \coloneqq 2i + 1$.



- For a nearly-complete binary tree $T = (V, E)$,
  we can use ***an array A of size $O(|V|)$*** to represent it.
  - The root is $A[1]$.
  - Given an index $i \geq 1$,
    - Parent(i) $\coloneqq \lfloor i/2 \rfloor$.
    - Left(i) $\coloneqq 2i$.
    - Right(i) $\coloneqq 2i + 1$.





Properties of Array-Representation

- Let A be an array representation of a nearly complete binary tree $T = (V, E)$ and let $\boldsymbol{n} = |\boldsymbol{V}|$. We have the following properties.
  - Each of the nodes at

    $$\lfloor n/2 \rfloor + 1, \qquad \lfloor n/2 \rfloor + 2, \qquad \cdots\cdots, \qquad n$$

    is a leaf node.
  - For any $1 \leq h \leq \lfloor \log n \rfloor + 1$, there are at most

    $$\left\lceil n/2^h \right\rceil$$

    nodes at height h.

In the following,
we assume array representation.

Maintain the Heap Property

- We introduce a procedure for maintaining a max-heap.
- The Max-Heapify(A, i) procedure takes as input
  - A nearly complete binary tree T with root i, where
  - Both of Left(i) and Right(i), if not empty, are both max-heaps.
- The Max-Heapify procedure guarantees that T is a max-heap after execution in $O(\log|T|)$ time.

- Max-Heapify(A, i)

  -- To assure the heap property for the tree rooted at i.

  -- Assumption: Left(i) and Right(i), if not empty, are max-heaps.

A. Let $k \coloneqq i$.

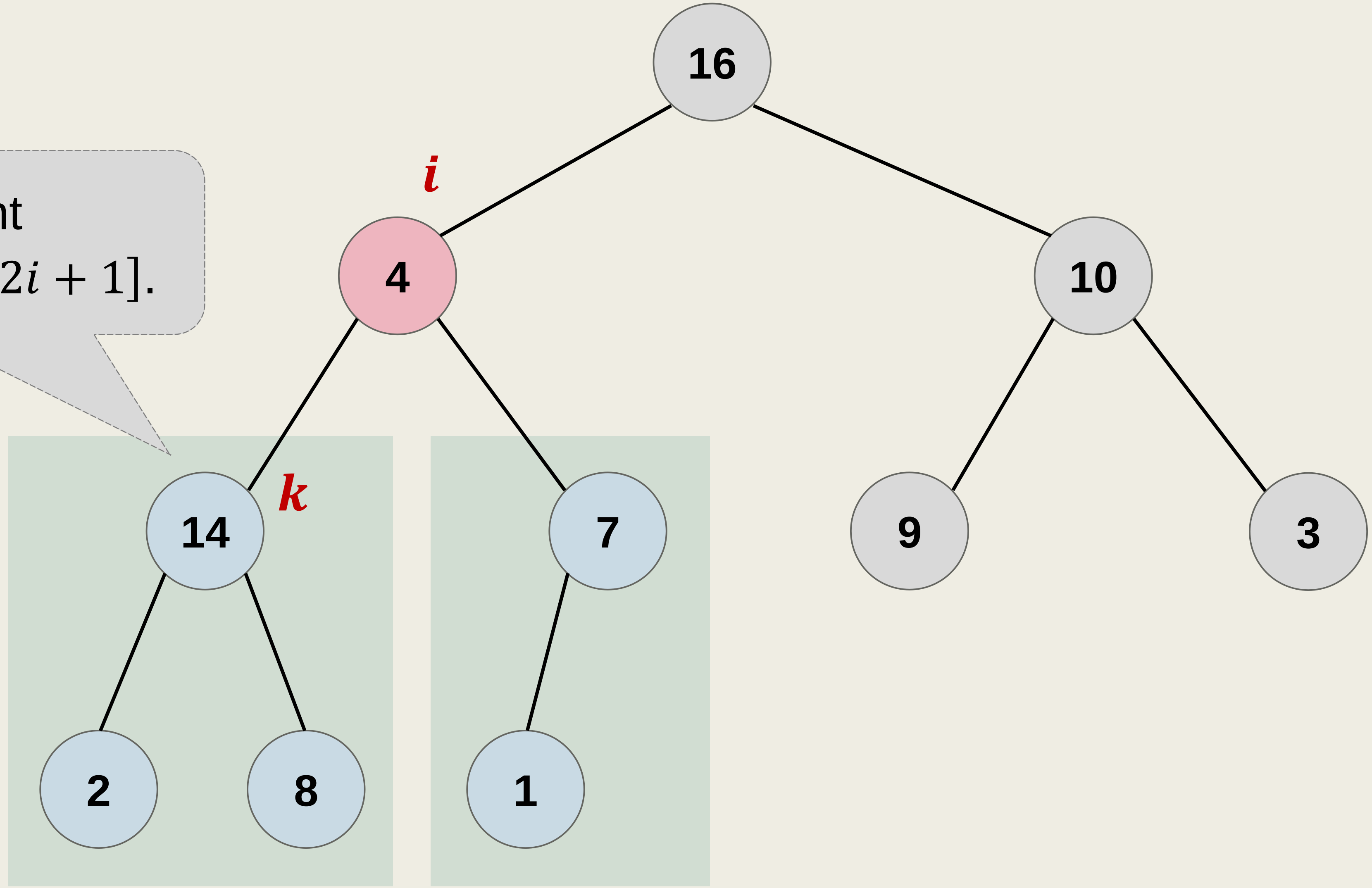
B. If $2i \leq heap\_size[A]$ and $A[2i] > A[k]$, then $k \coloneqq 2i$.

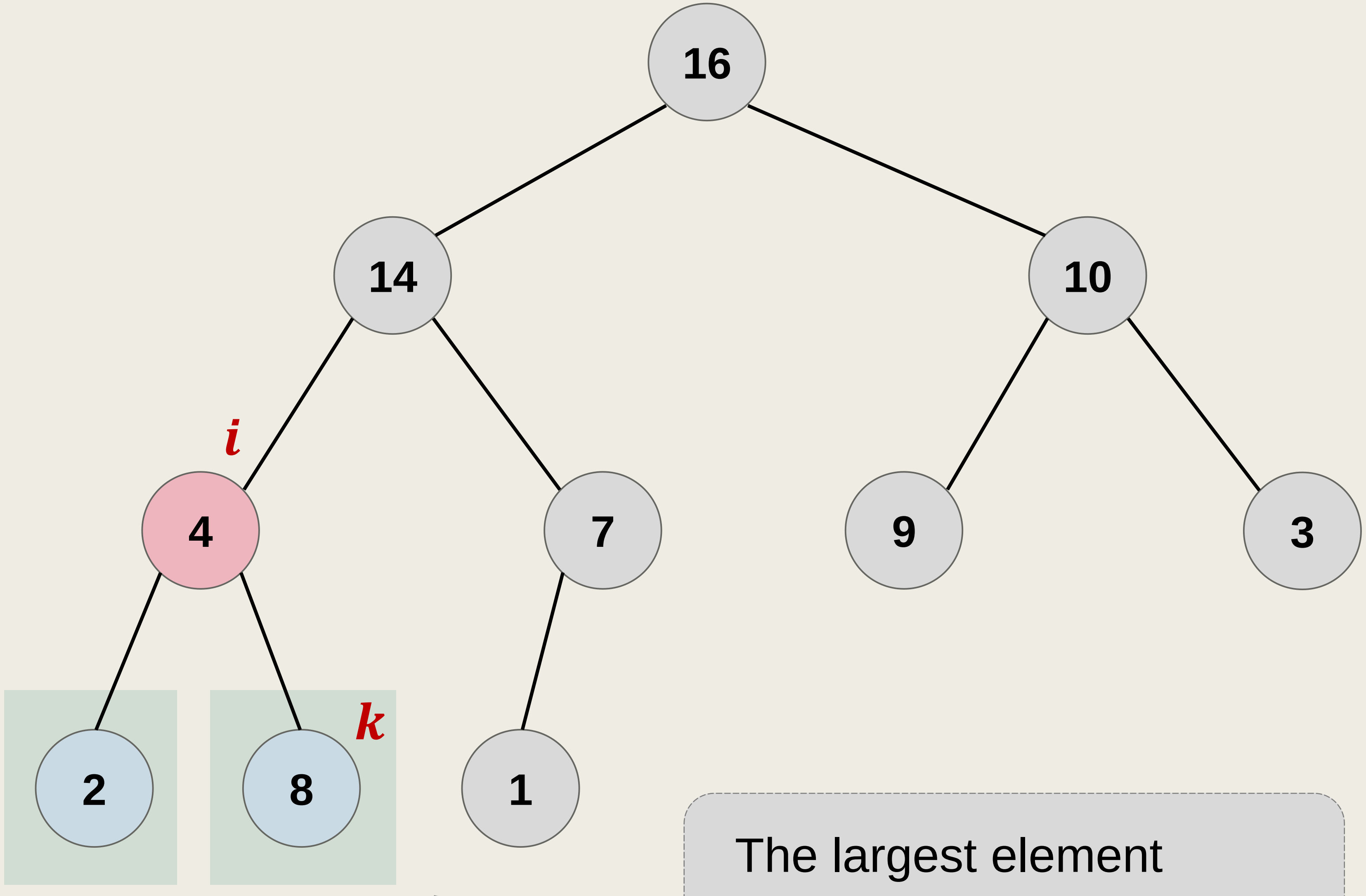
   If $2i + 1 \leq heap\_size[A]$ and $A[2i + 1] > A[k]$, then $k \coloneqq 2i + 1$.

C. If $k \neq i$, then

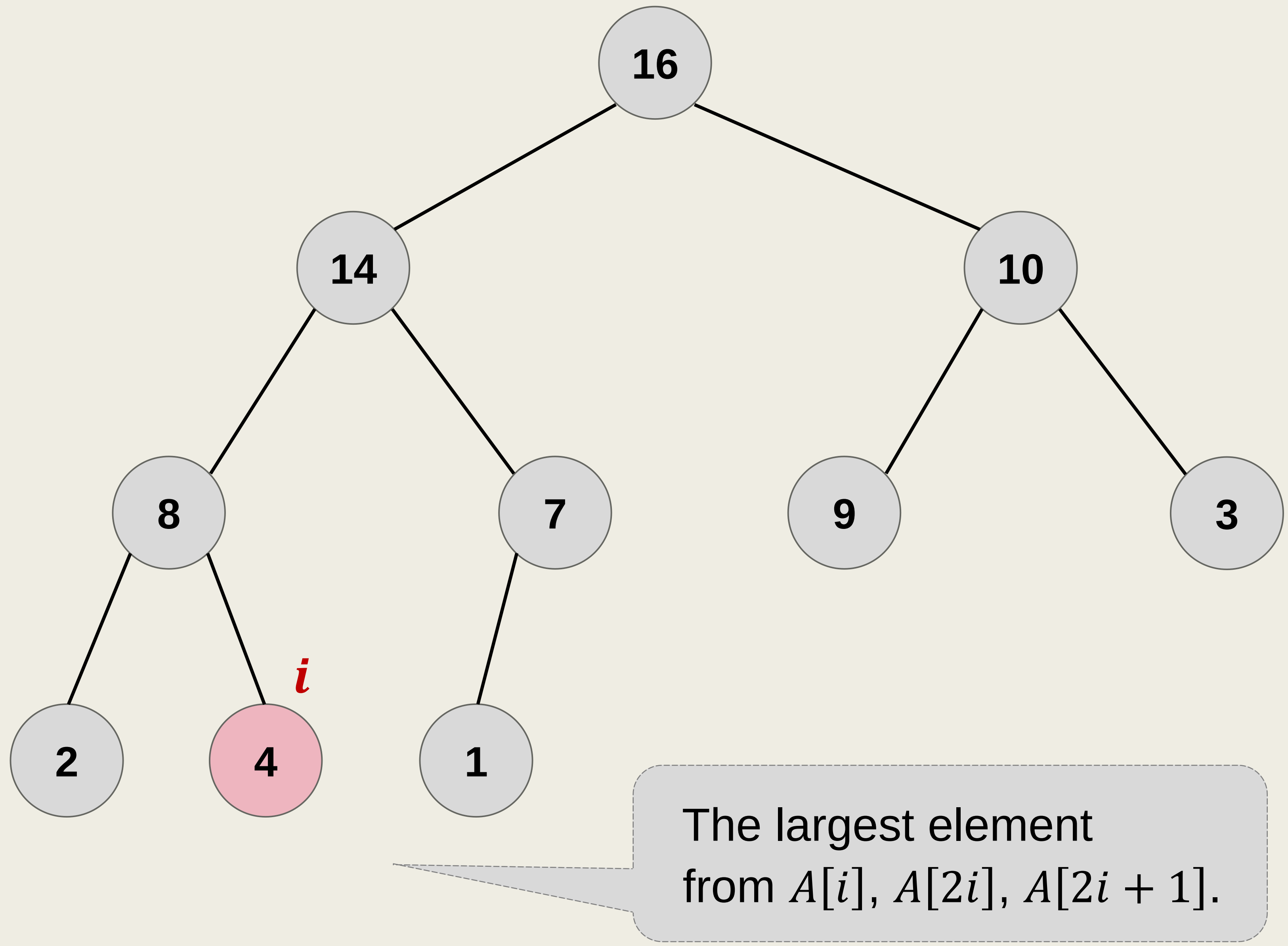
   - Exchange $A[i]$ with $A[k]$.
   - Max-Heapify(A, k).

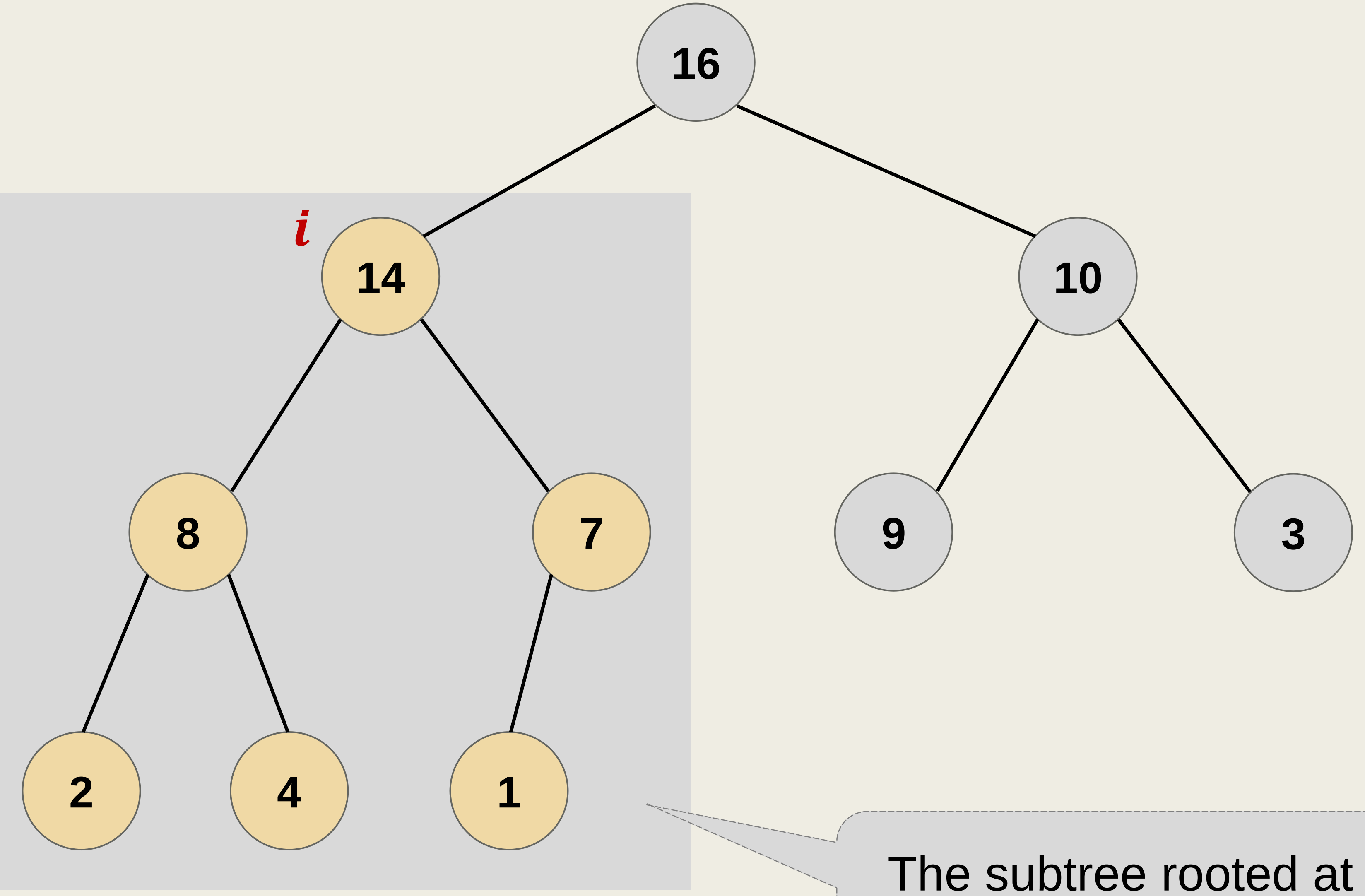
The largest element from $A[i]$, $A[2i]$, $A[2i+1]$.





The largest element
from $A[i]$, $A[2i]$, $A[2i+1]$.


16
14
10
8
7
9
3
i
2
4
1
The largest element
from $A[i]$, $A[2i]$, $A[2i+1]$.



The subtree rooted at i is now a max-heap.

Building a Heap in $O(n)$ Time

Building the Heap in $O(n)$ Time

- The Build-Max-Heap(A) procedure takes an array A as input and ***builds a max-heap*** for the elements in A ***in place***.
  - This procedure proceeds in a bottom-up manner and uses the Max-Heapify procedure to guarantee the heap property.

- Build-Max-Heap(A)

A. $heap\_size[A] := \mathrm{length}[A]$.

B. for $i = \mathrm{length}[A]$ down to 1, do
   Max-Heapify(A, i).

Analysis of Build-Max-Heap

- Recall that,

  the call to Max-Heapify on an element at height h takes $O(h)$ time.

- For any $1 \le h \le \lfloor \log n \rfloor + 1$, there are at most $\lceil n/2^h \rceil$ nodes at height h.

- Hence, the total running time of Build-Max-Heap is

$$\sum_{1 \le h \le \lfloor \log n \rfloor + 1} \left\lceil \frac{n}{2^h} \right\rceil \cdot O(h) \;=\; O\left(n \cdot \sum_{h \ge 0} \frac{h}{2^h} \right).$$

Analysis of Build-Max-Heap

- To bound $\sum_{h\geq 0} h/2^h$, observe that

$$\sum_{i\geq 0} x^i = \frac{1}{1-x}$$

  holds for all x with $|x| < 1$.

- Differentiating both sides of the equation on x, we obtain that

$$\sum_{i\geq 1} i \cdot x^{i-1} = \frac{1}{(1-x)^2} \quad \text{holds for any } |x| < 1.$$

Analysis of Build-Max-Heap

- Differentiating both sides of the equation w.r.t. x, we obtain that

$$\sum_{i \geq 1} i \cdot x^{i-1} = \frac{1}{(1-x)^2} \quad \text{holds for any } |x| < 1.$$

- Taking $x = 1/2$, we obtain that

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

- Hence,

$$\sum_{1 \leq h \leq \lfloor \log n \rfloor + 1} \left\lceil \frac{n}{2^h} \right\rceil \cdot O(h) = O(n) \cdot \sum_{h \geq 0} \frac{h}{2^h} = O(n).$$

Extracting the Maximum Element

Extracting the Maximum Element

- To extract the maximum element from a max-heap A, we swap the root with the last element, and perform Max-Heapify.
  - The time it takes is $O(\log n)$.

- Extract-Max(A)

A. Exchange $A[1]$ with $A[heap\_size[A]]$.
B. Decrease $heap\_size[A]$ by 1 and call Max-Heapify$(A, 1)$.
C. Return $A[heap\_size[A] + 1]$.

The Heapsort Algorithm

Heapsort

- With the procedure we have so far,
  we can do sorting in $O(n \log n)$ time with max-heap.

- Heapsort(A)

A. Build-Max-Heap(A).

B. For $i = \text{length}[A]$ down to 2, do

    Extract-Max(A).

Other Operations

Increase the Value of an Element

- We can change the value of an element. After that, we need to ensure the heap property. Overall it takes $O(\log n)$ time.
  - Perform Max-Heapify if the value is decreased.
  - Otherwise, we proceed upward if the value is increased.

- Max-Heap-Increase-Key(A, i, key) -- Assumption: $key > A[i]$.

A. $A[i] \leftarrow key$.

B. While $i > 1$ and $A[i/2] < A[i]$, do

- Exchange $A[i]$ with $A[i/2]$ and set $i \leftarrow i/2$.

Insert a new Element

- To insert an element, we insert it at the end of the heap and perform the increase-key operation.
  - The time it takes is $O(\log n)$.

- Max-Heap-Insert(A, key)

A. Increase $heap\_size[A]$ by 1.

B. Call Max-Heap-Increase-Key$(A, heap_size[A], key)$.

Priority Queues

Priority Queue

With ***max-heap***,
these operations can be done in…

- Suppose that we want to maintain a set A of ***elements, each associated with a key,*** so as to support the following operations.
  - **Insert(A, x)** – to insert a given element x into A.
  - **Maximum(A)** – to return the largest element in A.
  - **Extract-Max(A)** – to remove and return the largest element from A.
  - **Increase-Key(A, x, k)**
    – to increase the value of the elements x's key to the new value k.

$O(\log n)$ time.

$O(1)$ time.

$O(\log n)$ time.

Mergeable Heaps – A Note

Mergeable Heaps

- Mergeable Heaps refer to the data structures that supports the following operations.
  - Make_Heap() – to create and return an empty heap.
  - Insert(H, x) – to insert a given element x into H.
  - Minimum(H) – to return the smallest element in H.
  - Extract-Min(H) – to remove and return the smallest element from H.
  - **Union($\boldsymbol{H_1}$, $\boldsymbol{H_2}$)** – to create and return the union of H_1 and H_2. The heaps H_1 and H_2 are destroyed by this operation.

Mergeable Heaps

- This type of structures often supports the following two operations as well.
  - Decrease-Key(H, x, k) – to assign the element x a smaller key k.
  - Delete (H, x) – to delete a given node x from H.

Mergeable Heaps

Procedure	Binary Heap (worst-case)	Binomial Heap (worst-case)	Fibonacci Heap (amortized/average)
Make-Heap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
Minimum	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$
Extract-Min	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Union	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
Decrease-Key	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
Delete	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

As the semesters are shortened,
we may not be able to examine them in this semester.