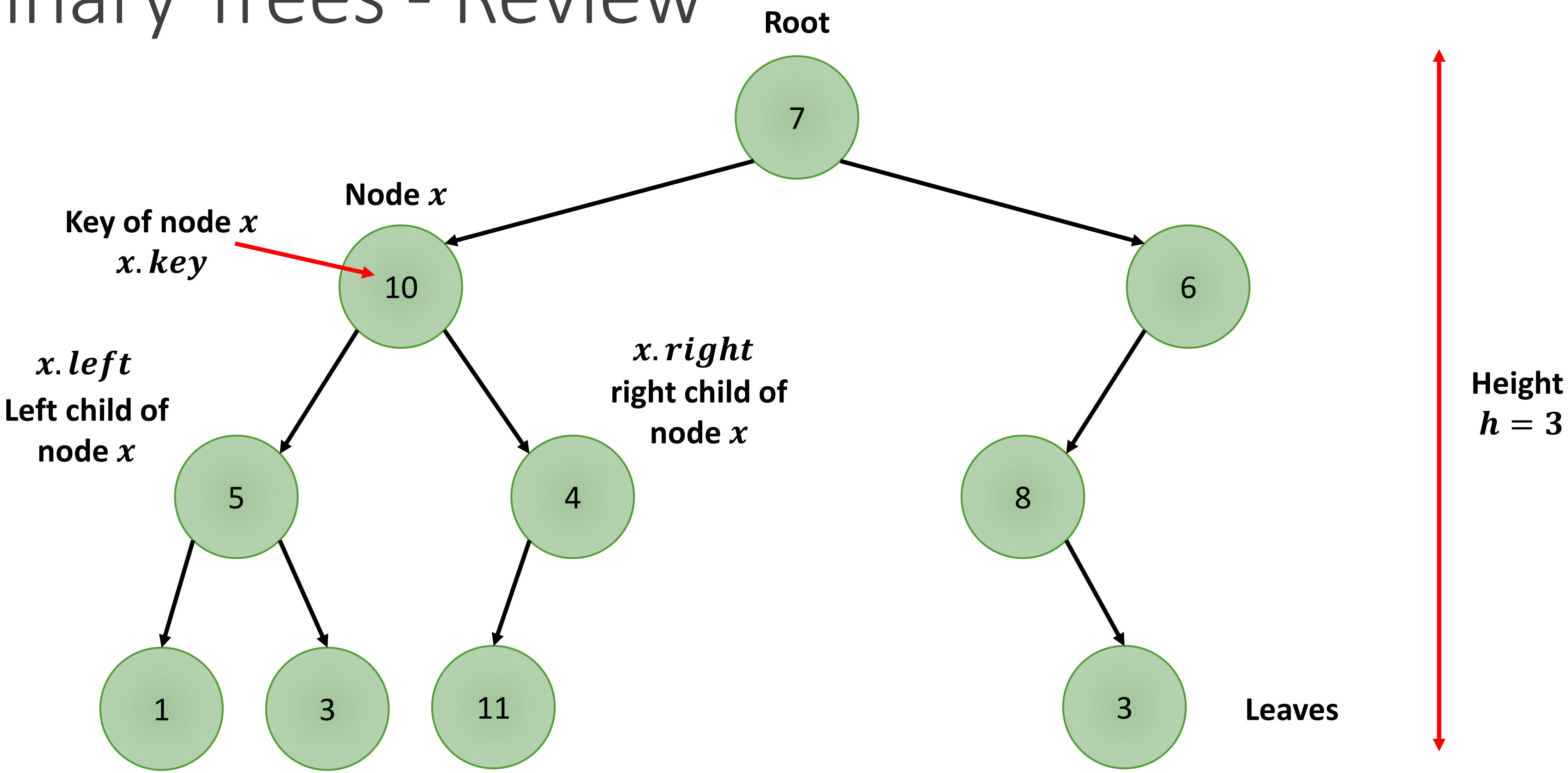
CP312
Algorithm Design and Analysis I

LECTURE 7: TREE-BASED SORTING

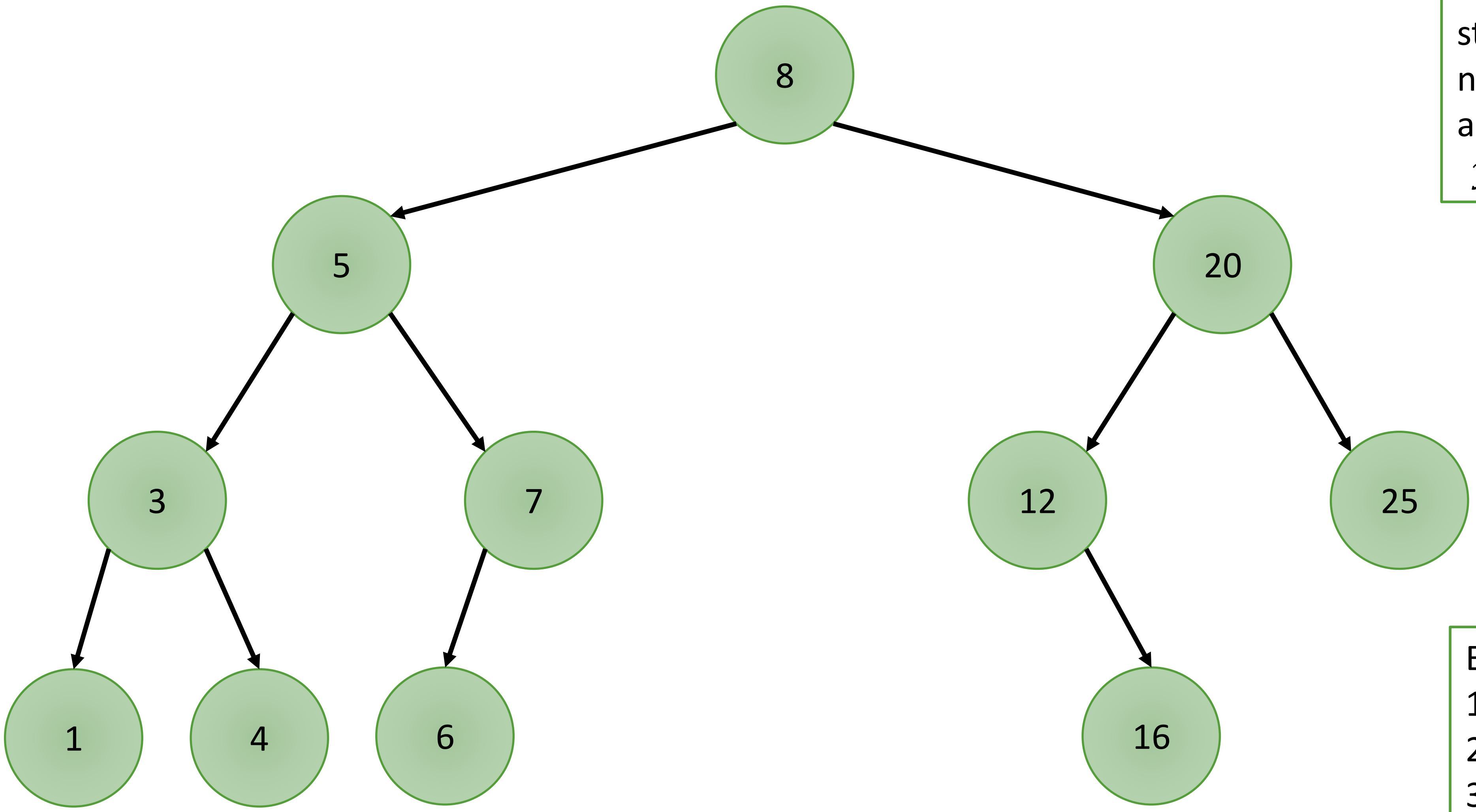
Sorting with Trees

- In precious lectures we saw how to sort using arrays.
- In this lecture, we will see the effect of changing data structures on the performance of an algorithm.
- In particular, we will use **binary trees** instead of arrays.

Binary Trees - Review



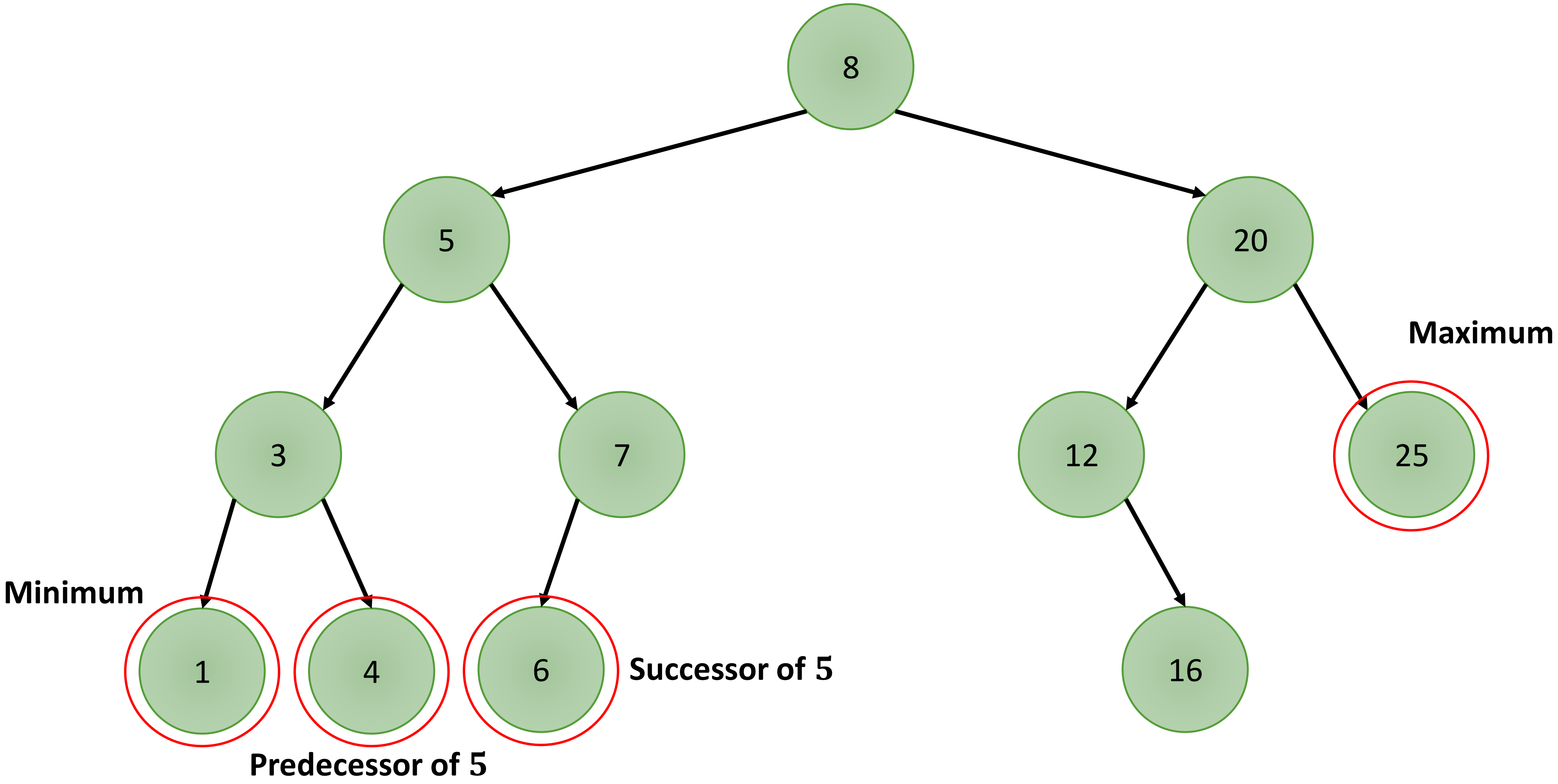
Binary **Search** Trees (BST)



Key Property:
BST is a binary tree data structure where for every node x with left child y and right child z:
$y.key < x.key < z.key$

BST operations:
1.) BST-search(k)
2.) BST-insert(k)
3.) BST-delete(k)

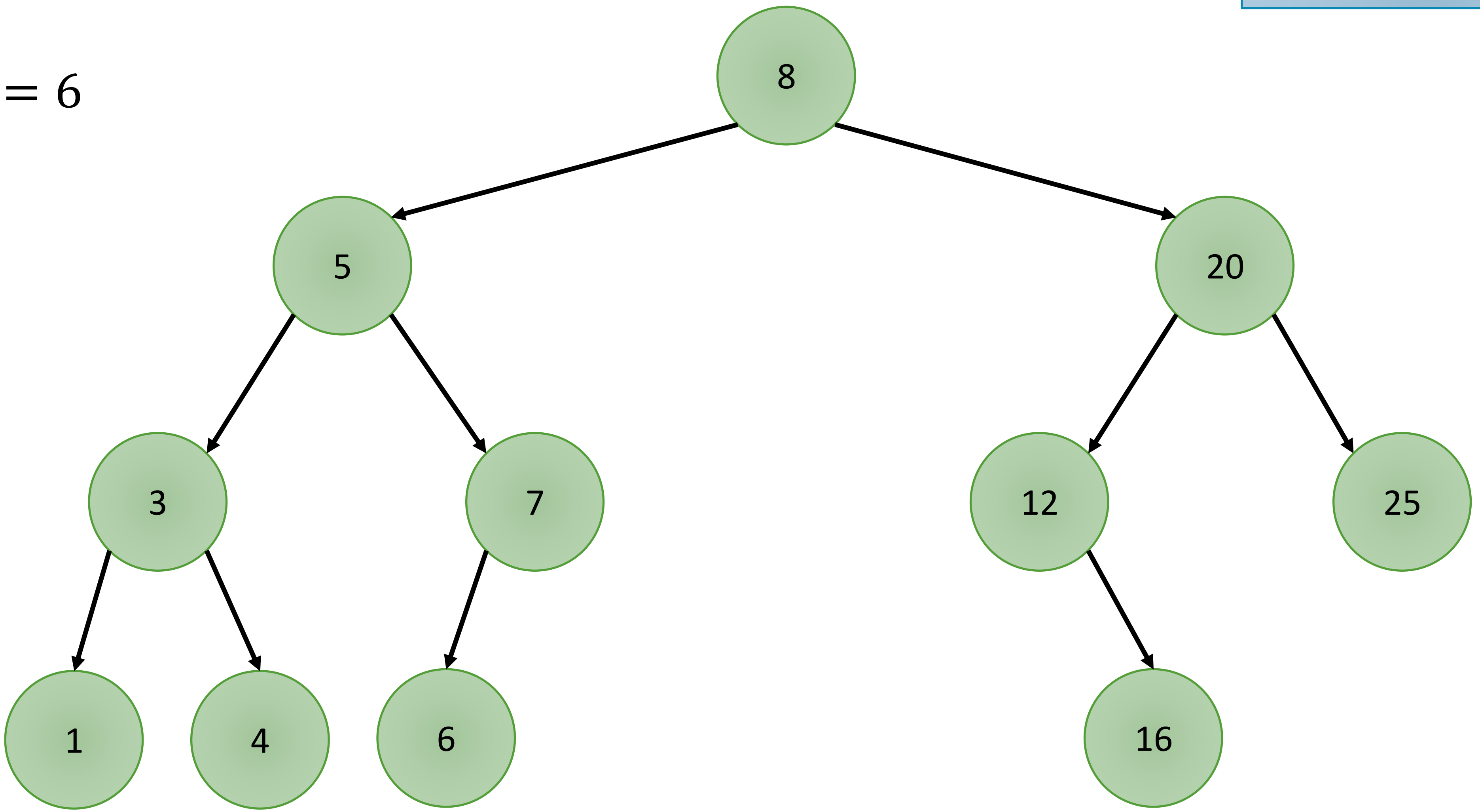
Binary Search Trees (BST)



Binary **Search** Trees (BST)

BST-Search(k)

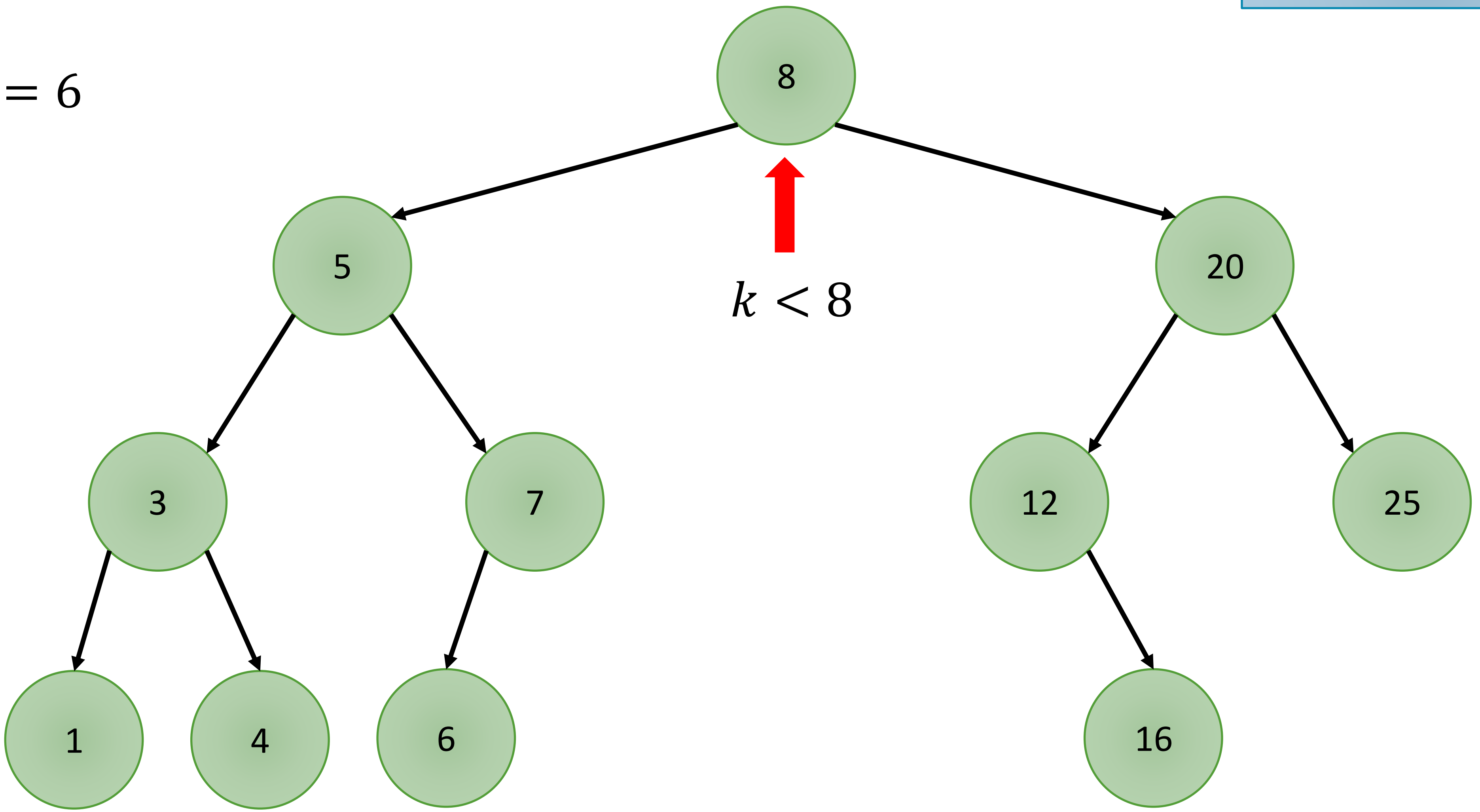
$k = 6$



Binary **Search** Trees (BST)

BST-Search(k)

$k = 6$

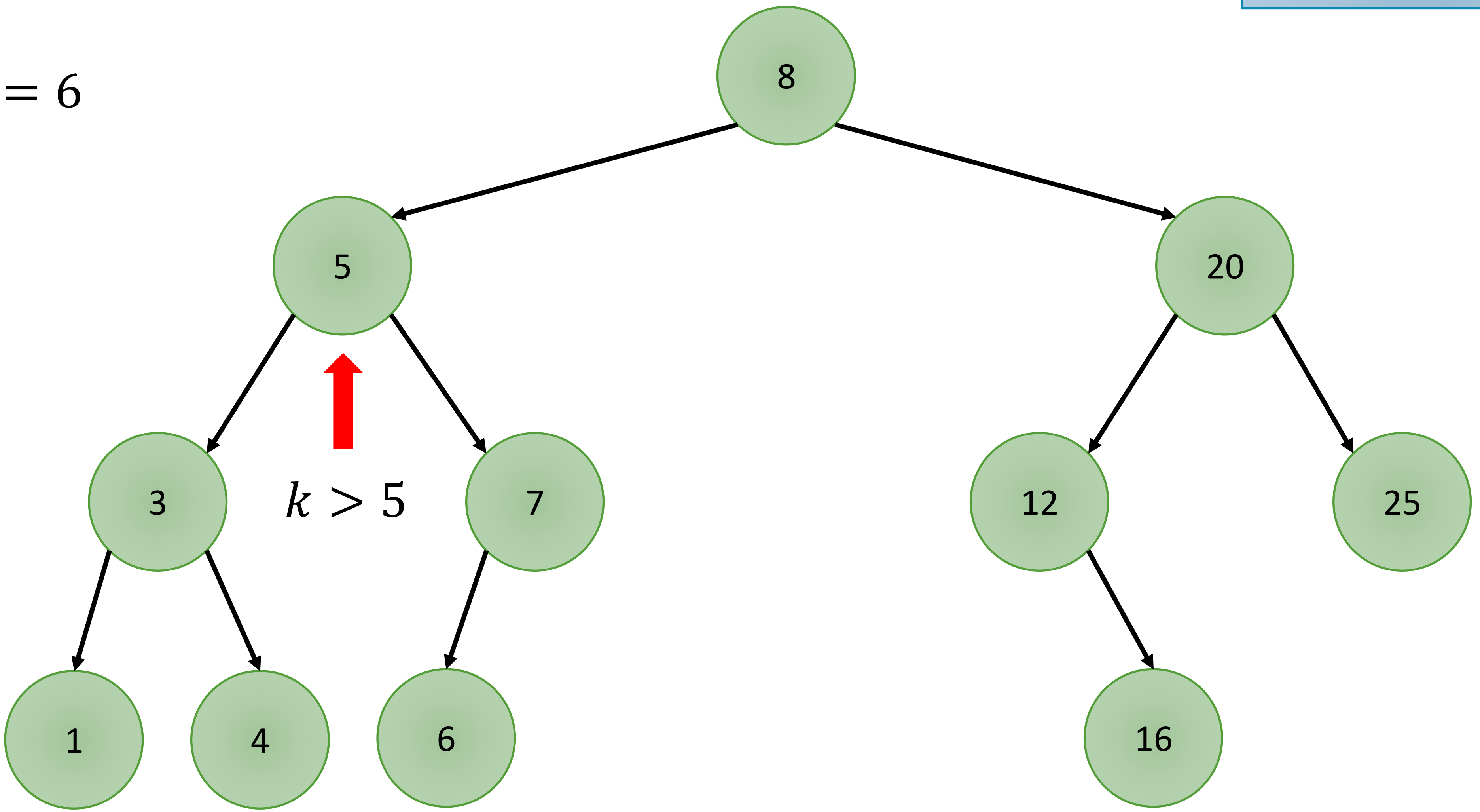


$k < 8$

Binary **Search** Trees (BST)

BST-Search(k)

$k = 6$

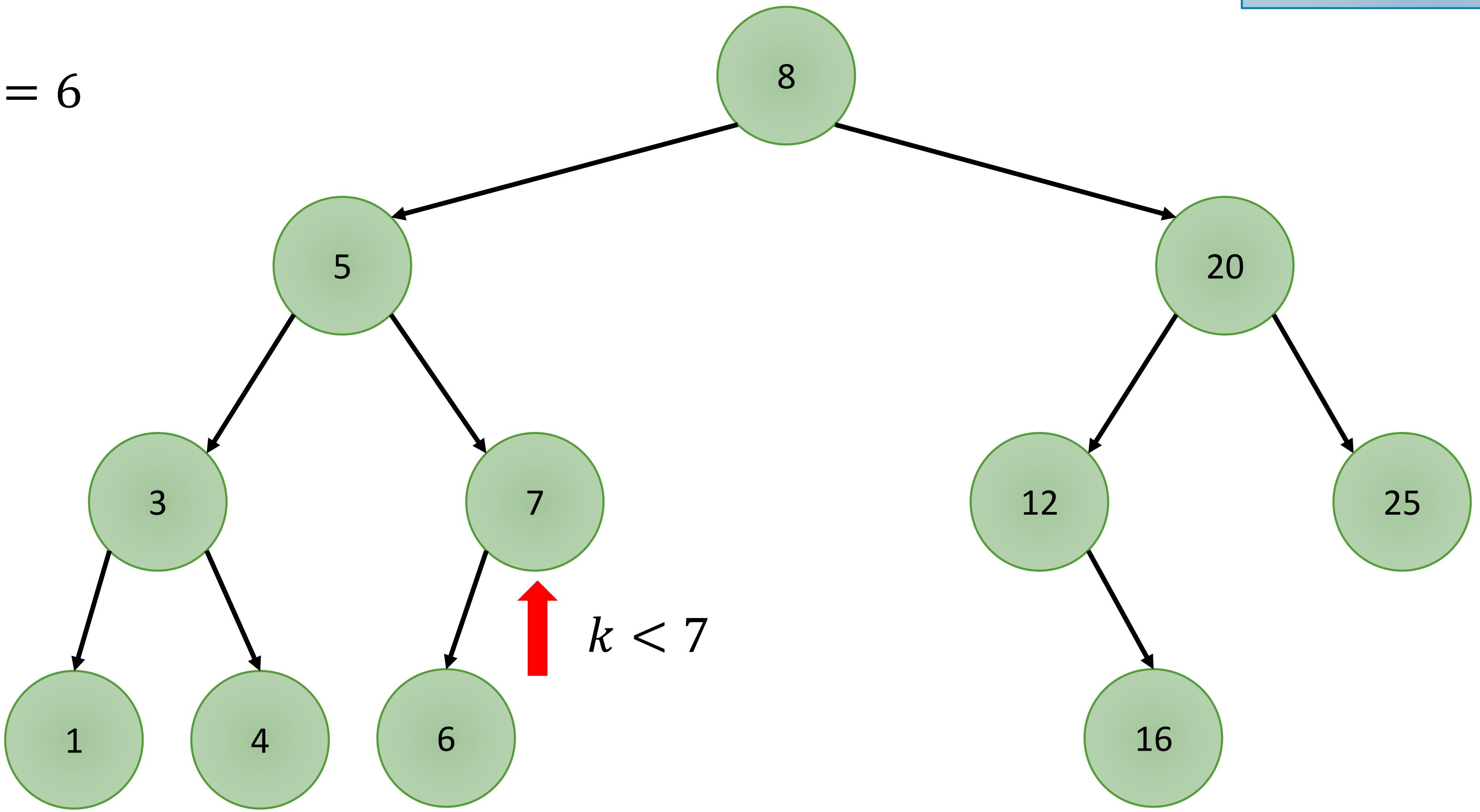


$k > 5$

Binary **Search** Trees (BST)

BST-Search(k)

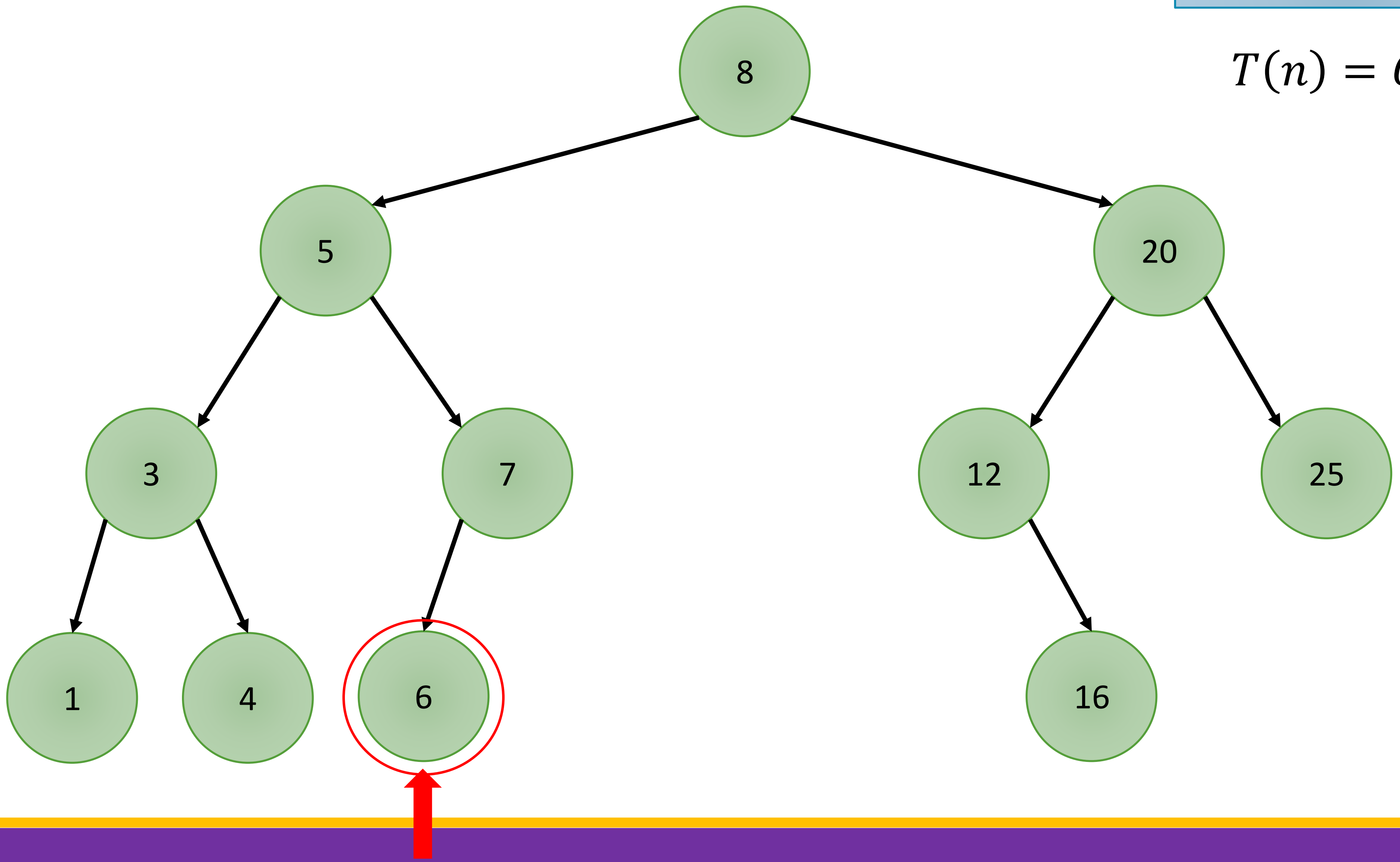
$k = 6$



Binary **Search** Trees (BST)

BST-Search(k)

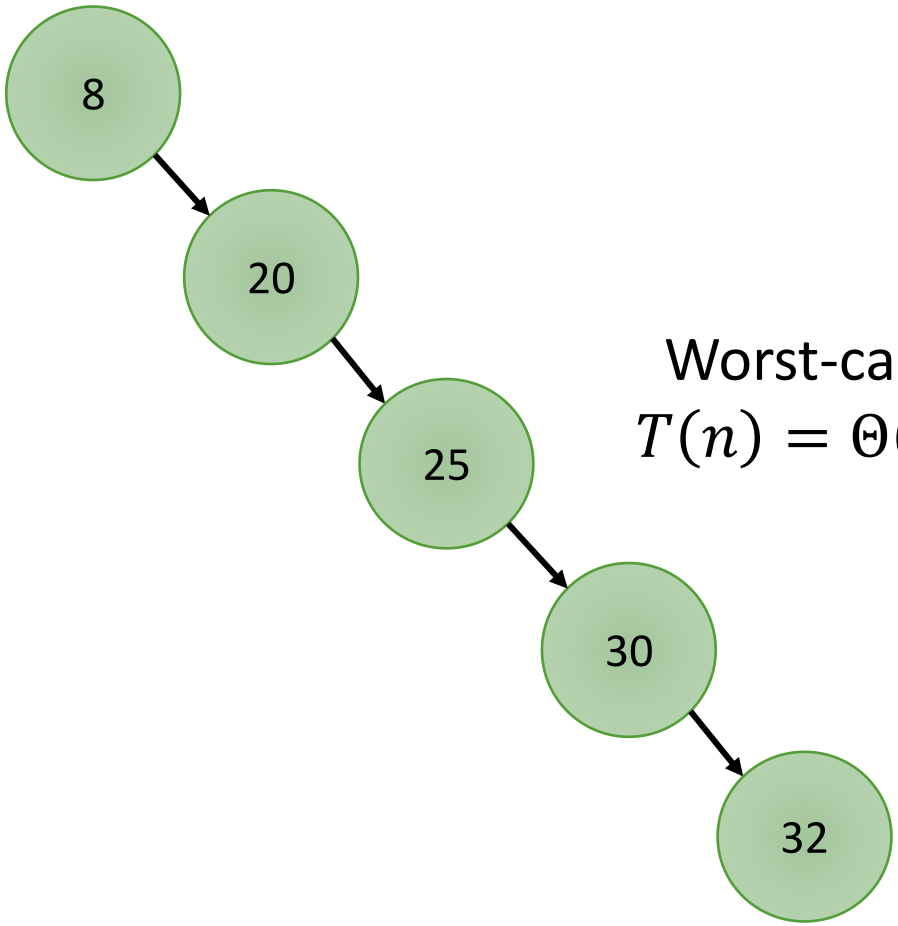
$$T(n) = O(h)$$



Binary **Search** Trees (BST)

BST-Search(k)

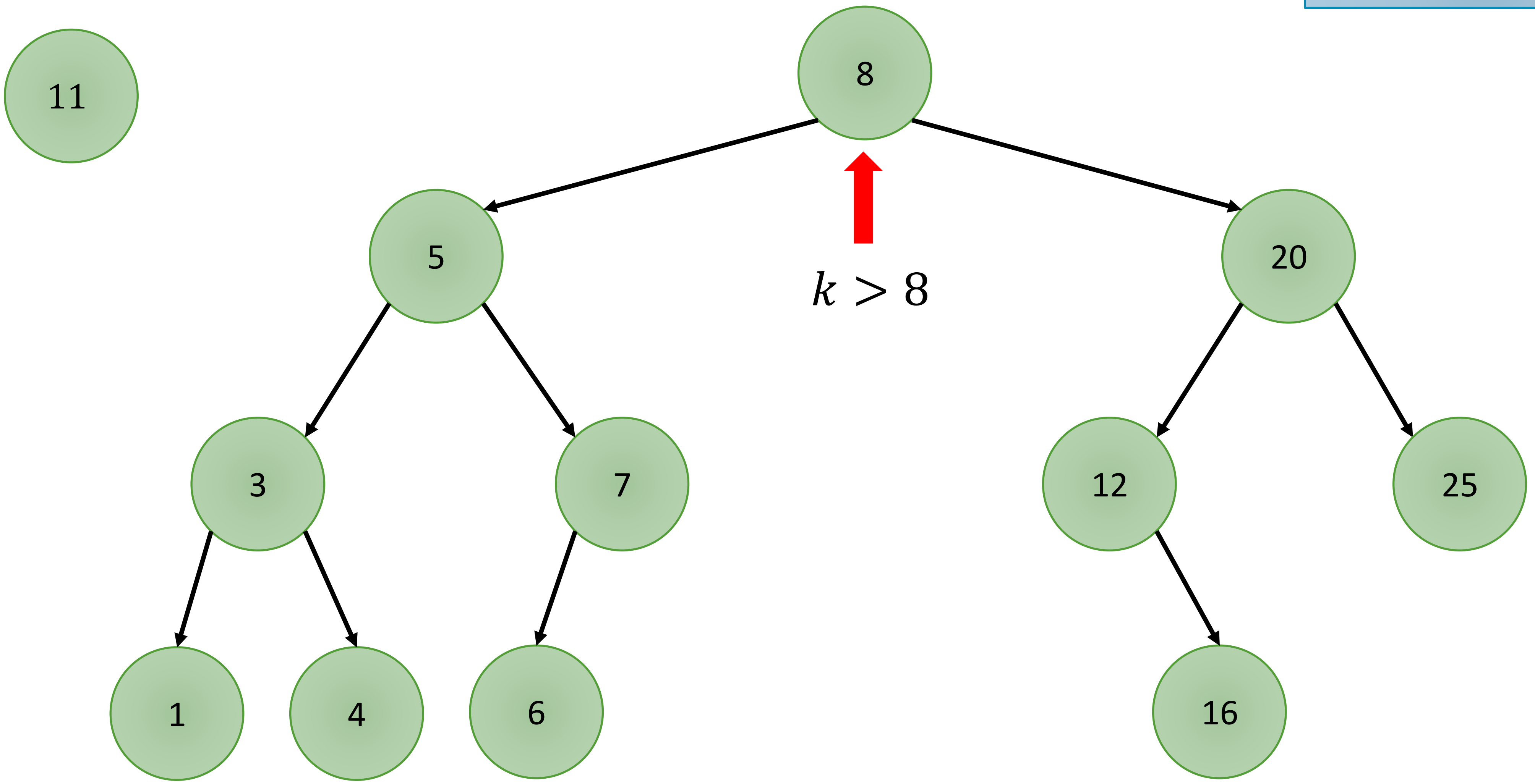
$T(n) = O(h)$



Worst-case
$T(n) = \Theta(n)$

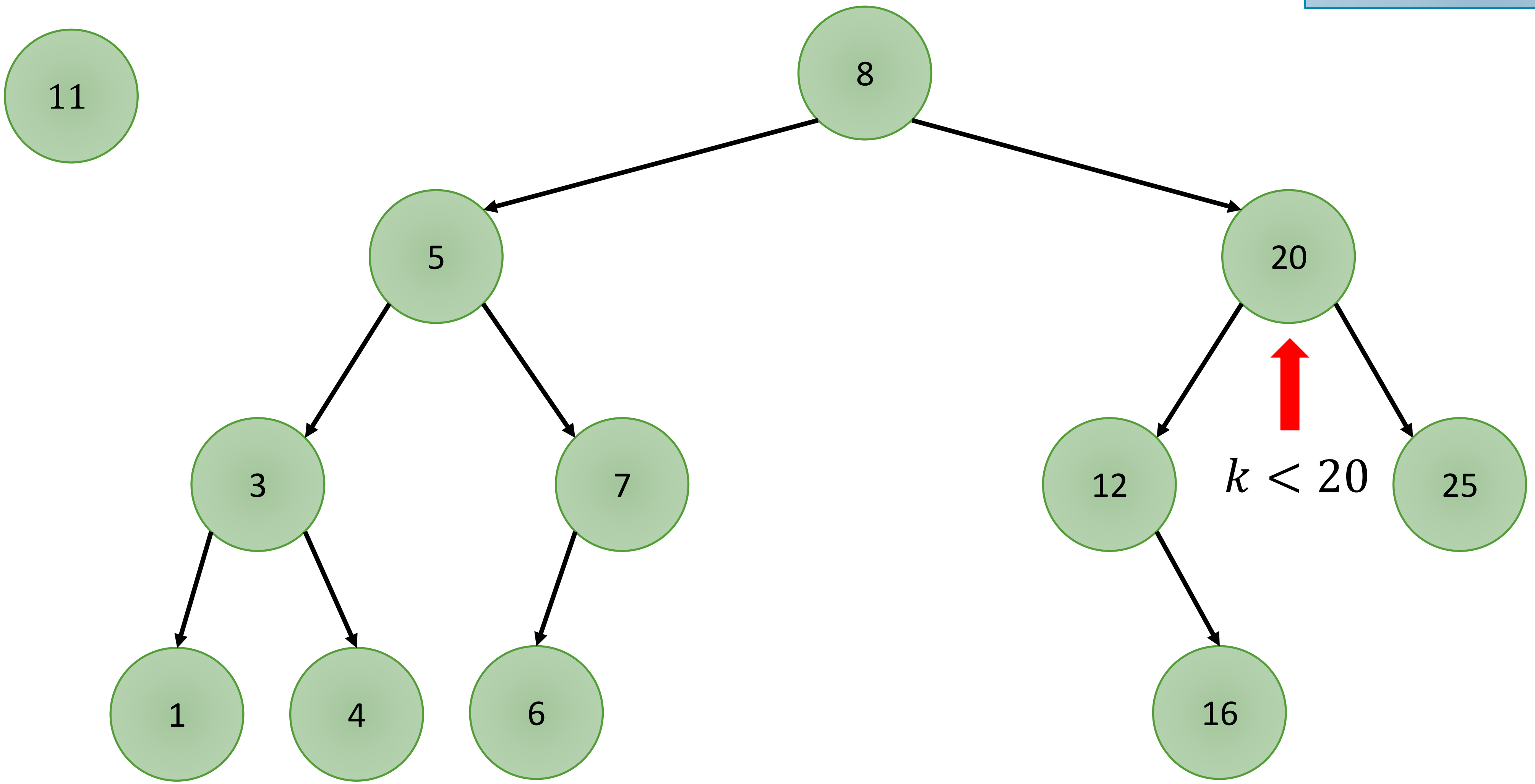
Binary **Search** Trees (BST)

BST-Insert(k)



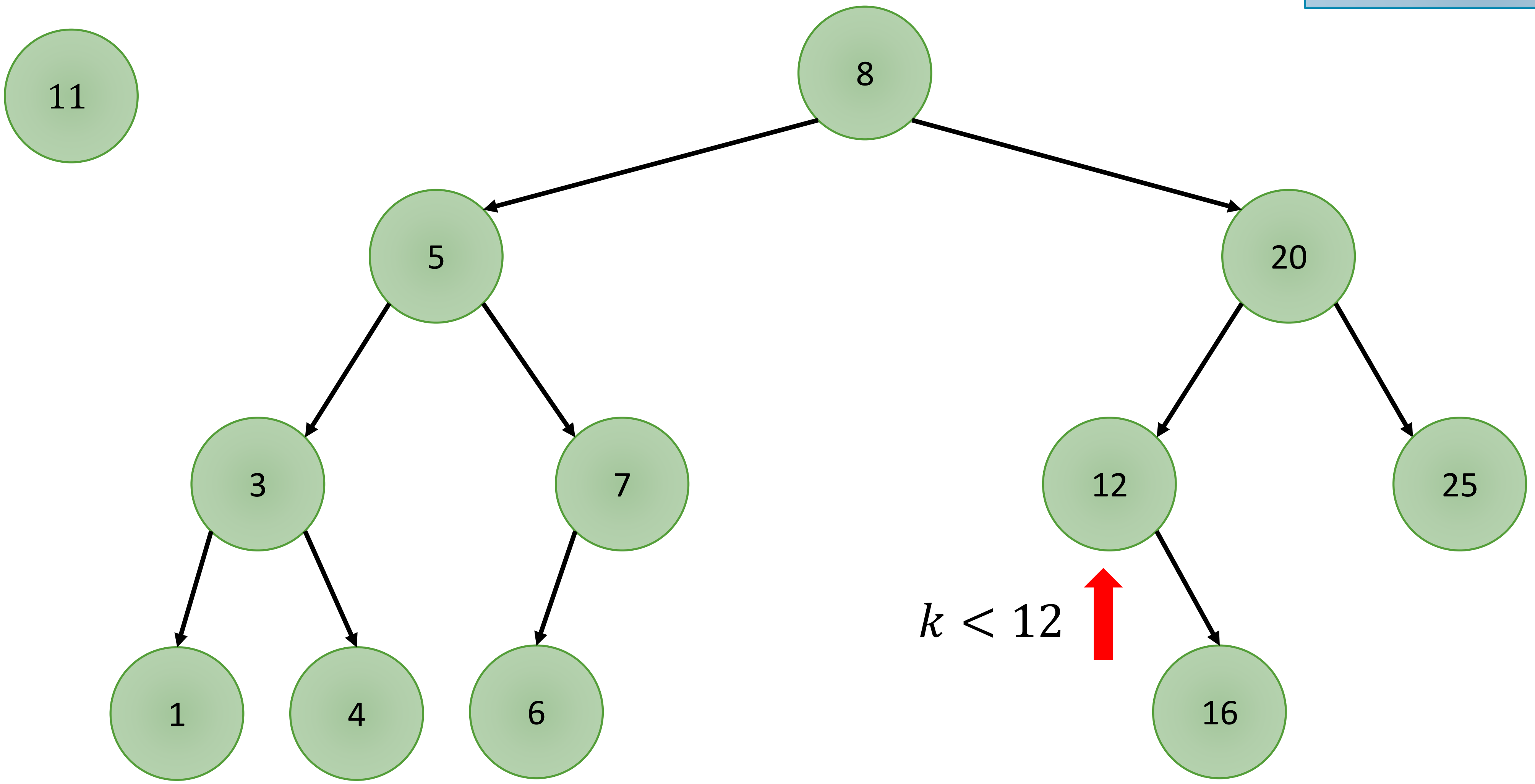
Binary **Search** Trees (BST)

BST-Insert(k)



Binary **Search** Trees (BST)

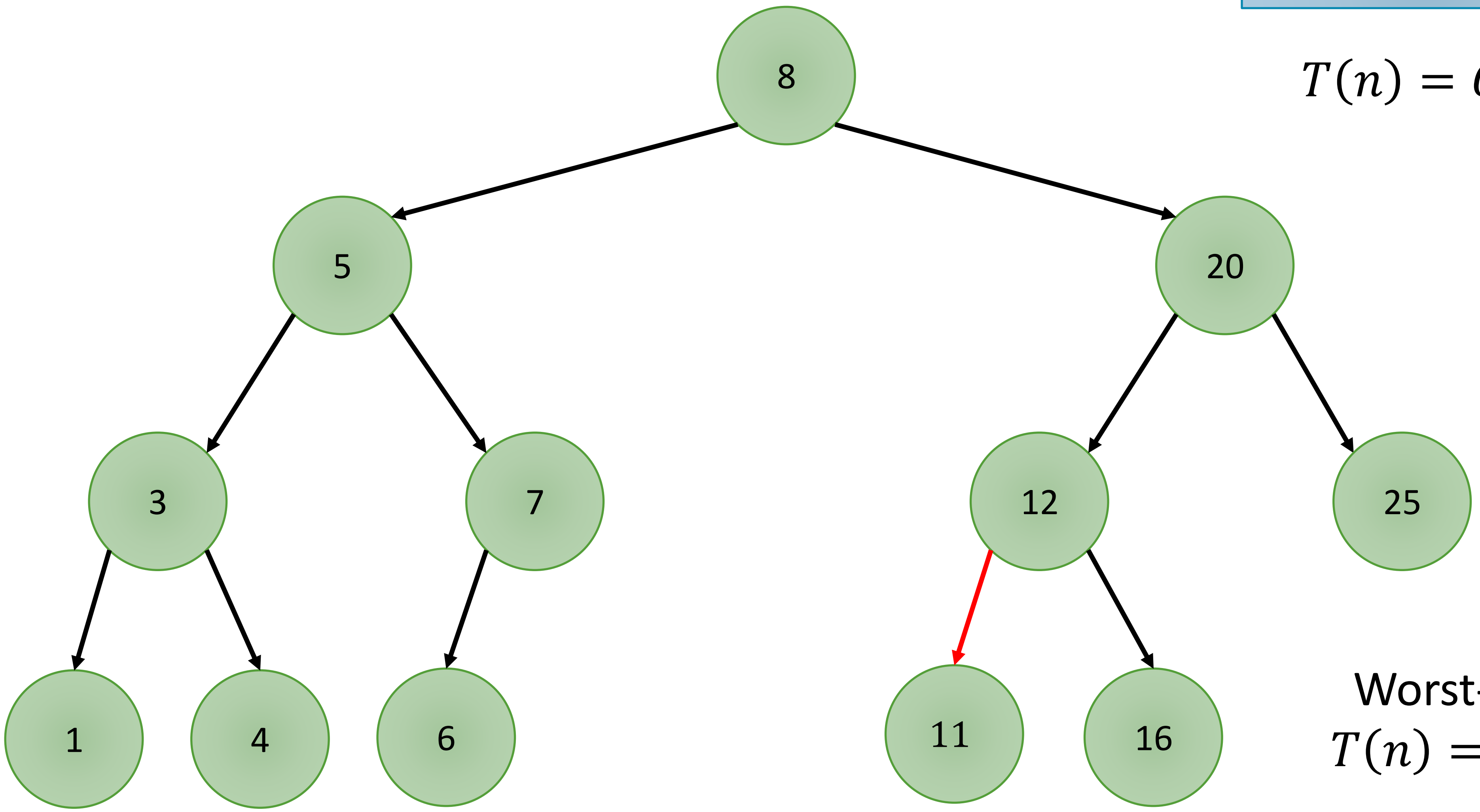
BST-Insert(k)



Binary **Search** Trees (BST)

BST-Insert(k)

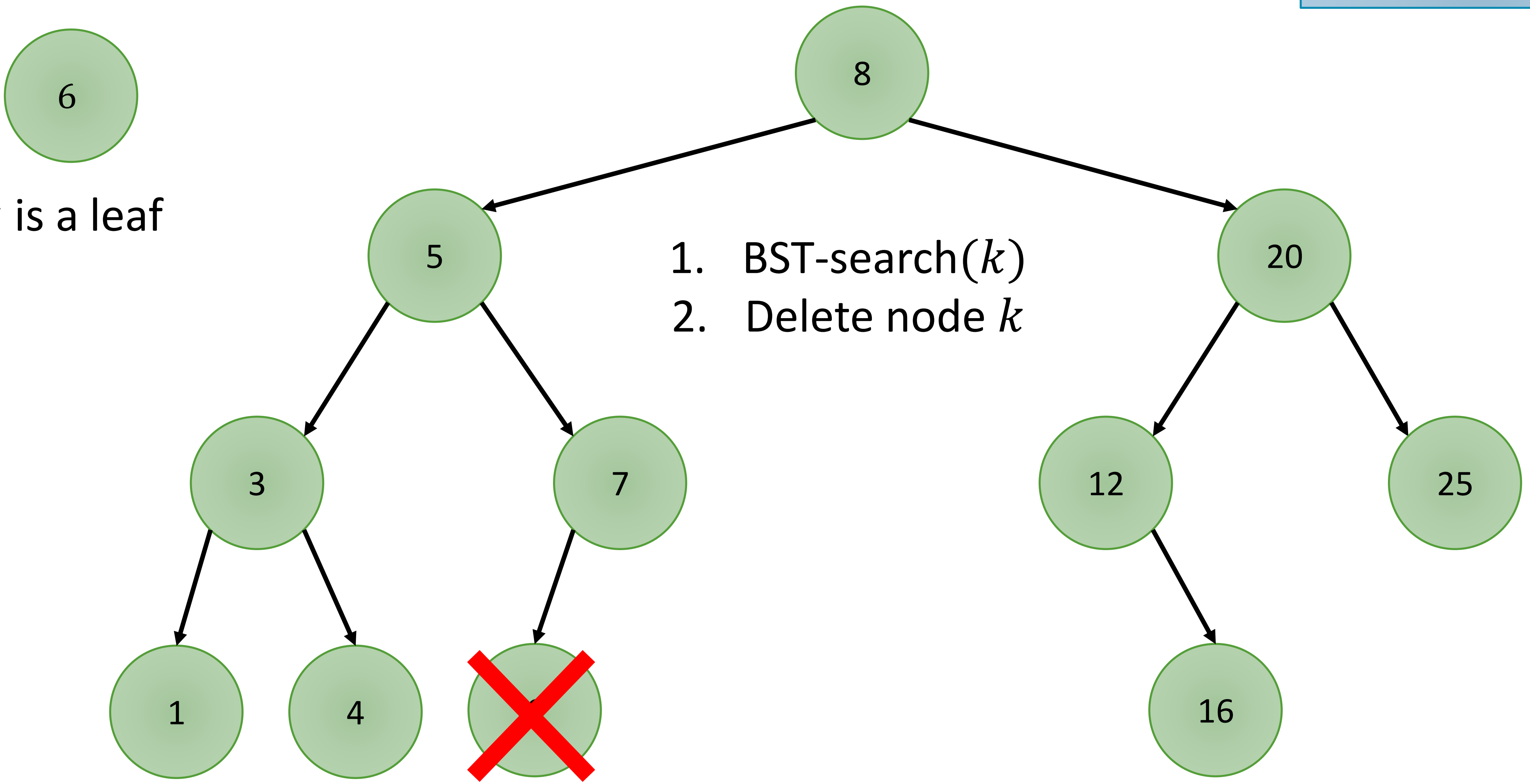
$T(n) = O(h)$



Worst-case
$T(n) = \Theta(n)$

Binary **Search** Trees (BST)

BST-delete(k)

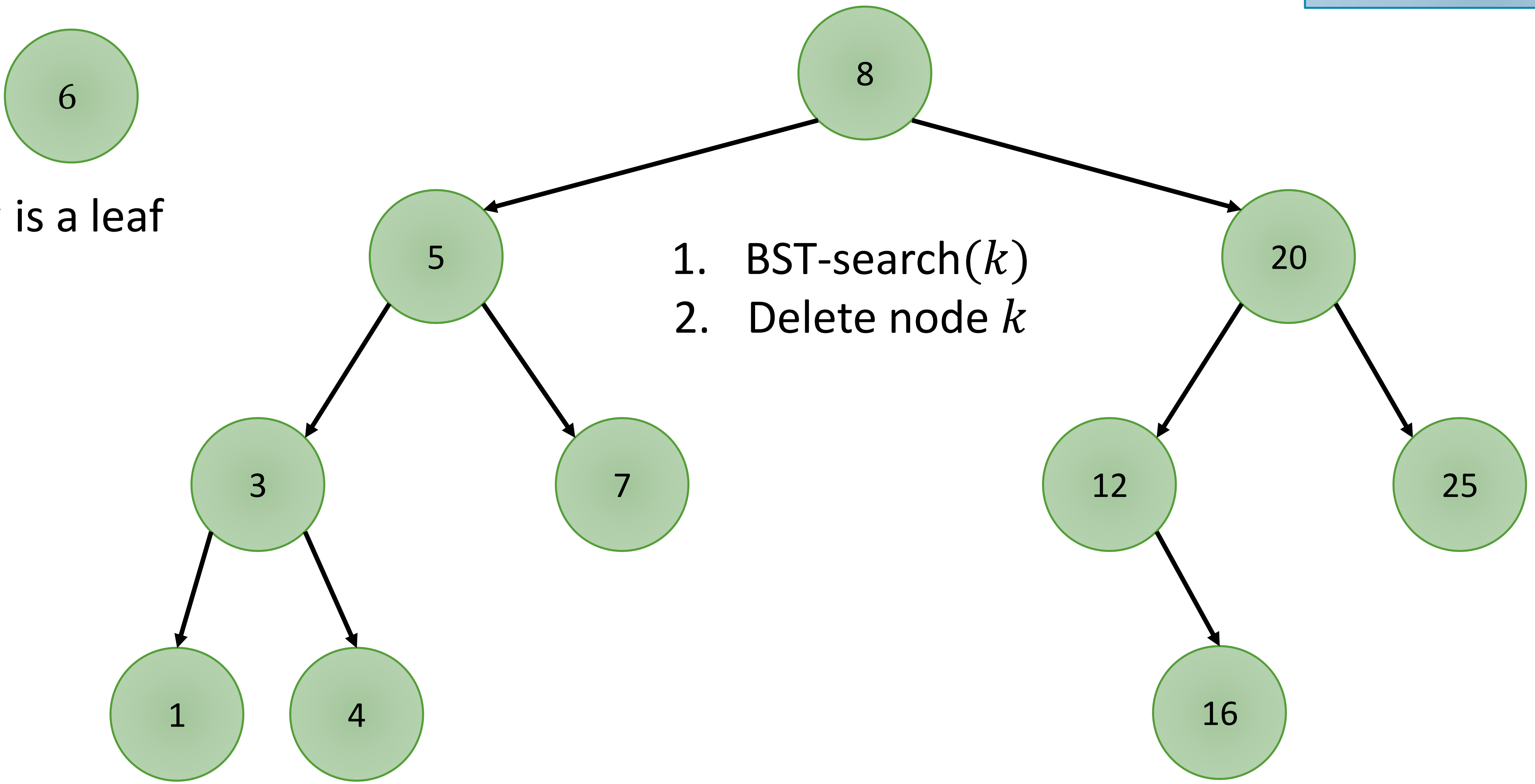


k is a leaf

1. BST-search(k)
2. Delete node k

Binary **Search** Trees (BST)

BST-delete(k)



k is a leaf

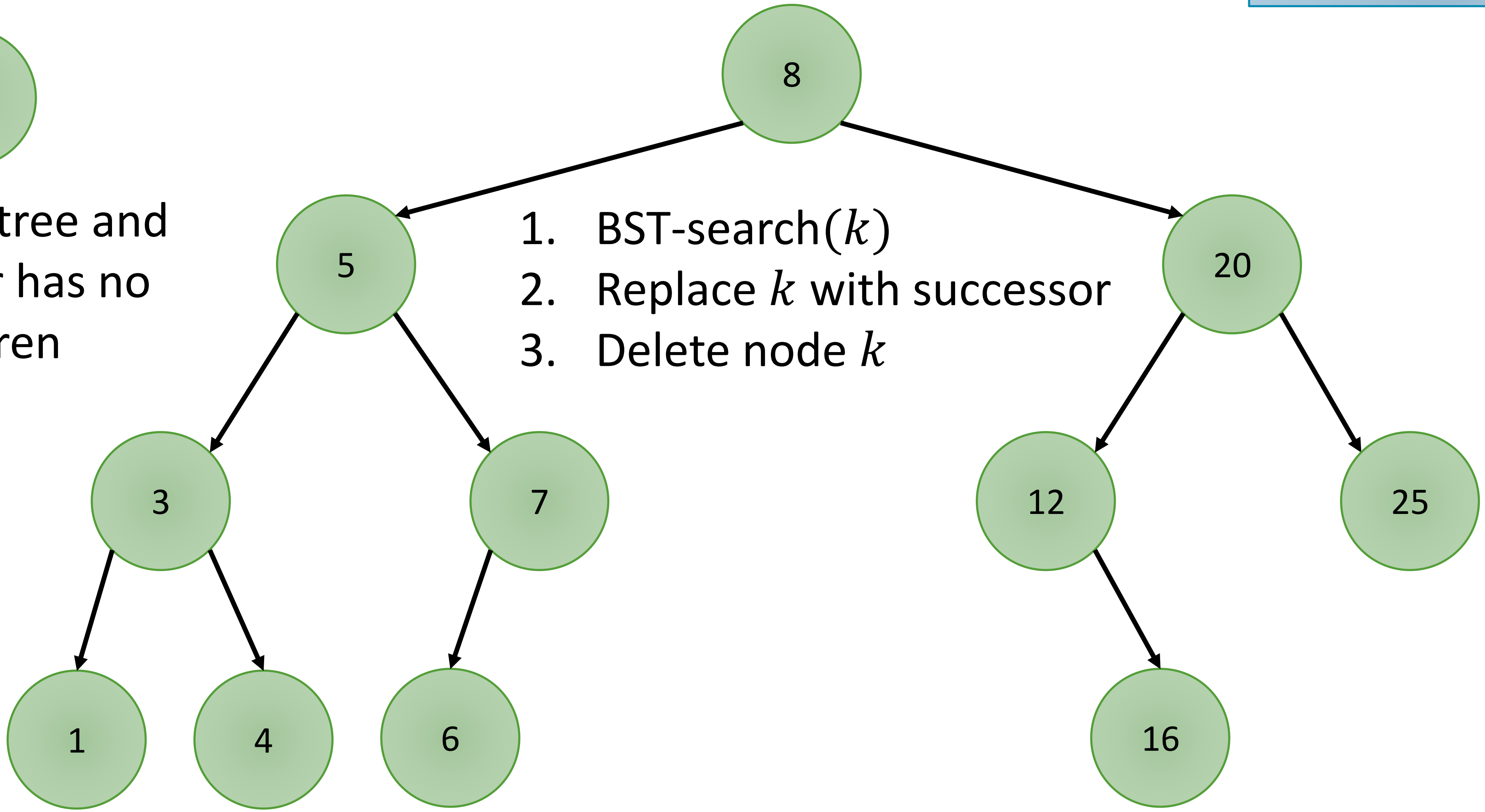
1. BST-search(k)
2. Delete node k

Binary **Search** Trees (BST)

BST-delete(k)

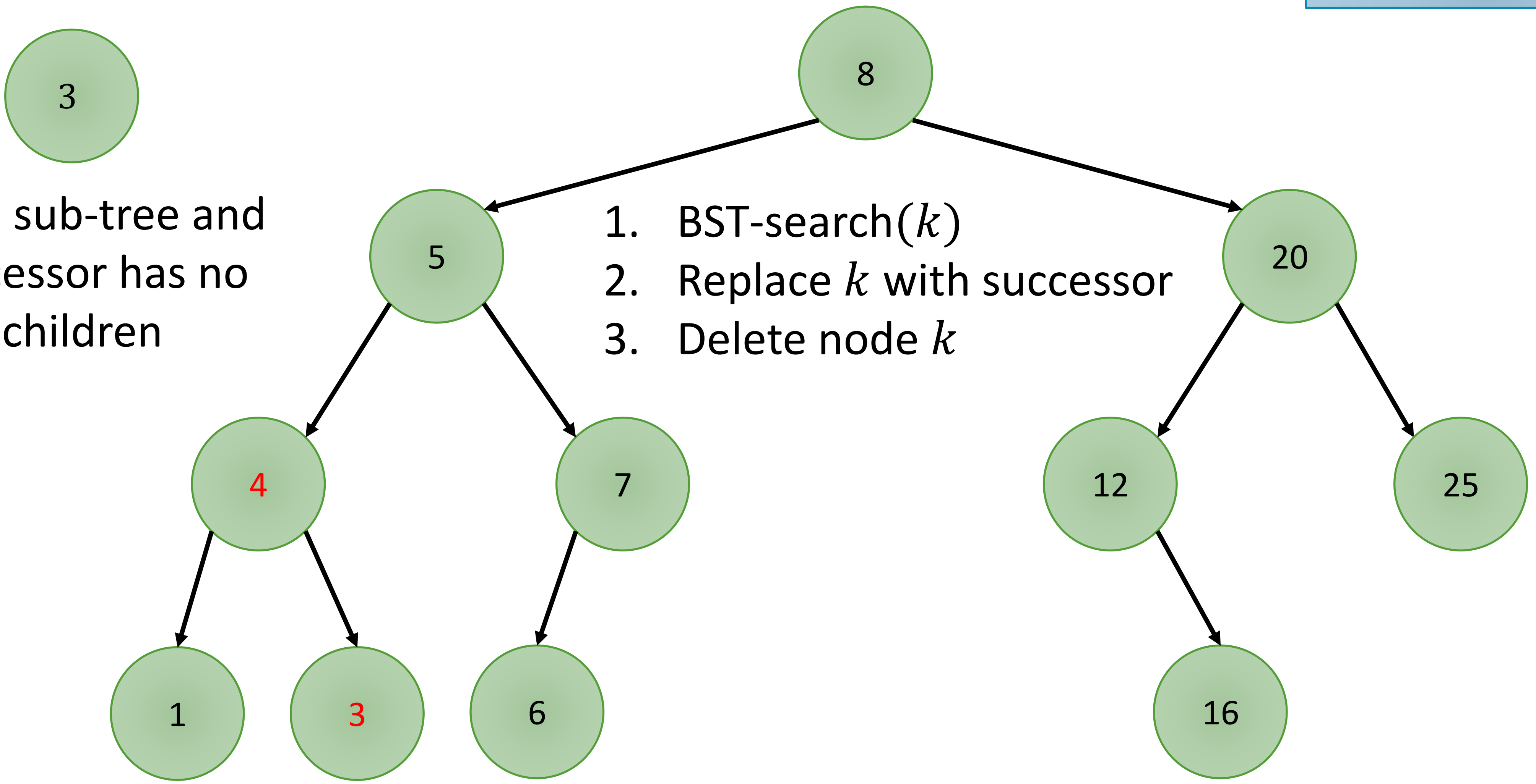
k is a sub-tree and successor has no children

1. BST-search(k)
2. Replace k with successor
3. Delete node k



Binary **Search** Trees (BST)

BST-delete(k)



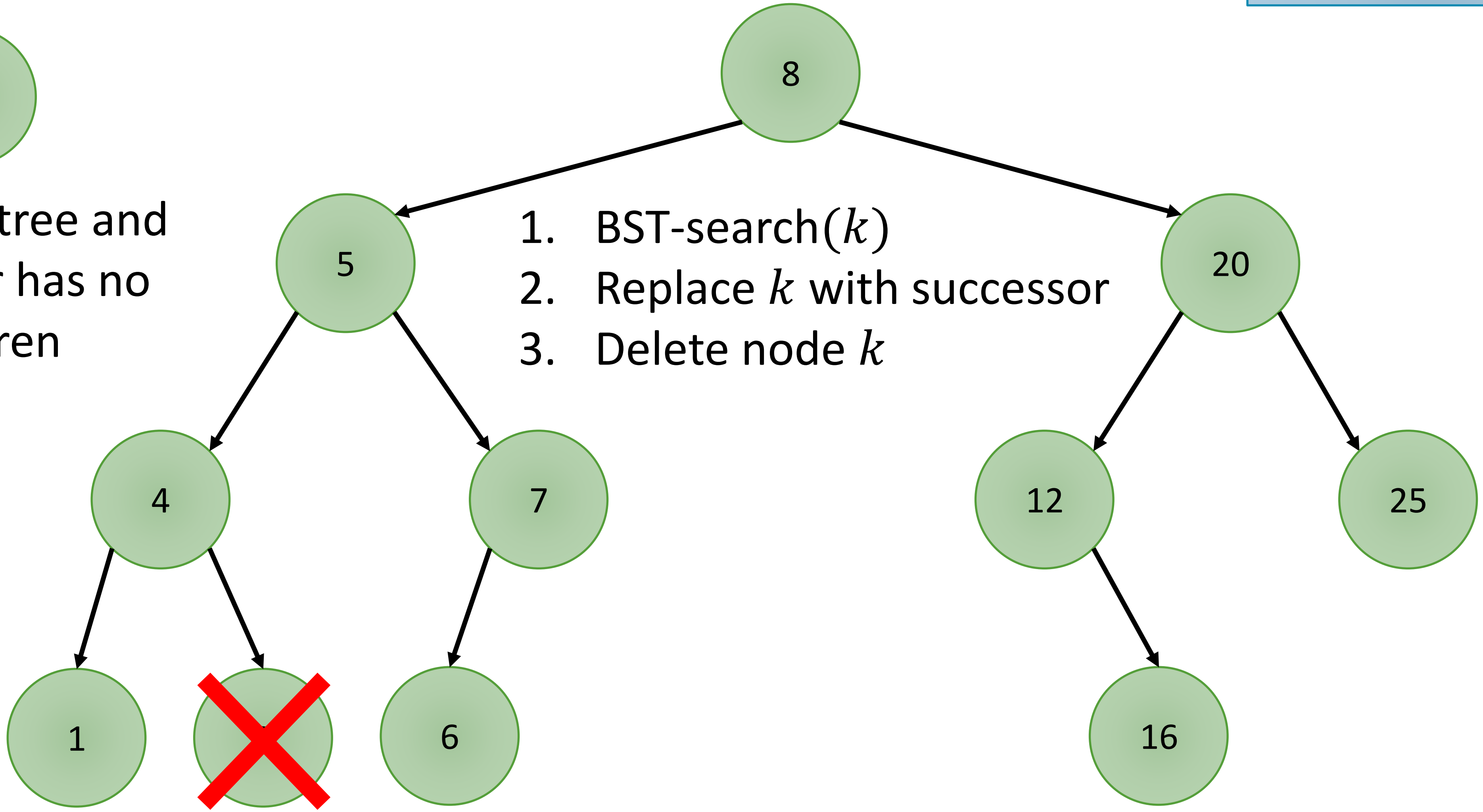
k is a sub-tree and successor has no children

1. BST-search(k)
2. Replace k with successor
3. Delete node k

Binary **Search** Trees (BST)

BST-delete(k)

k is a sub-tree and successor has no children

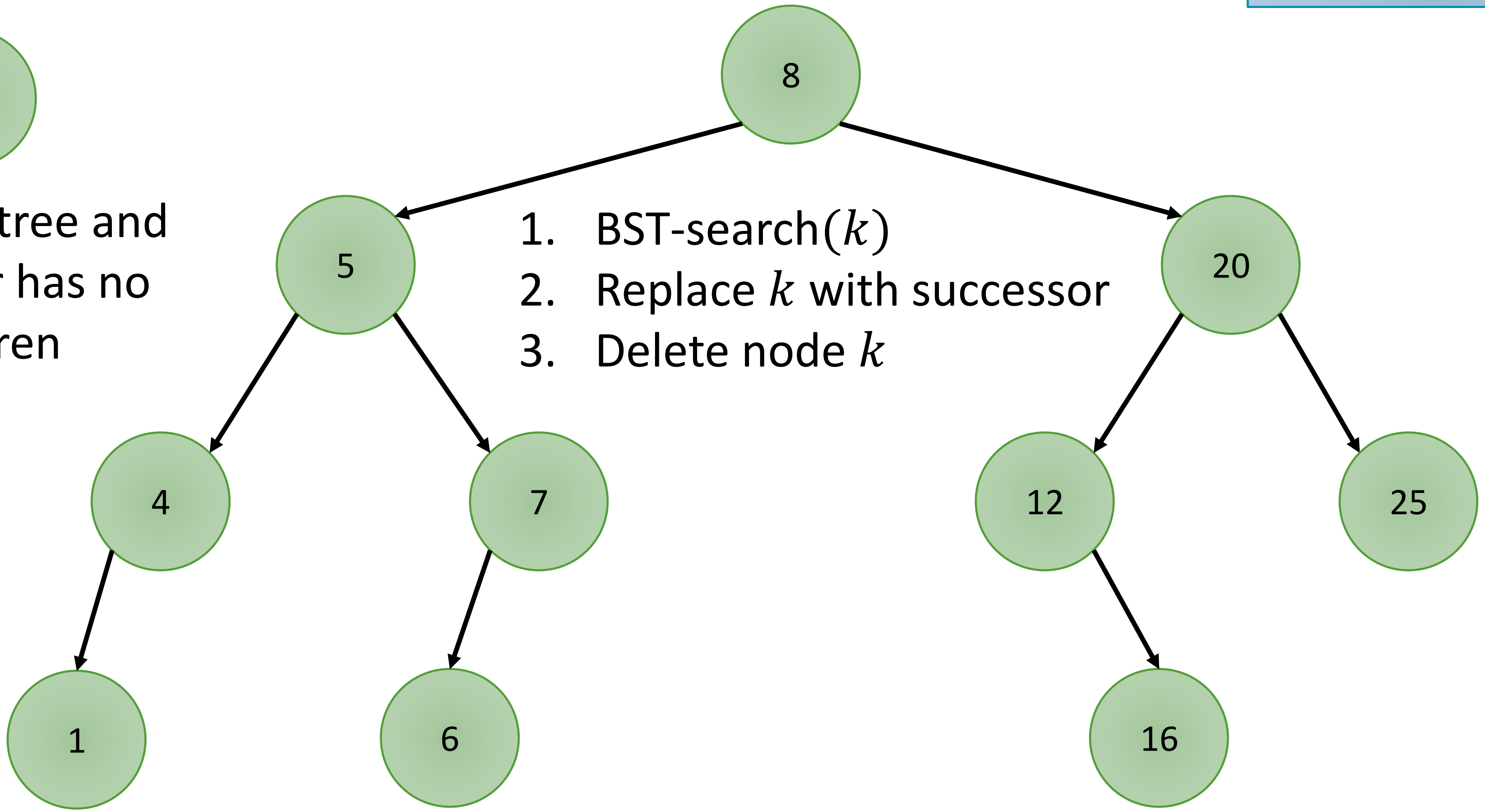


1. BST-search(k)
2. Replace k with successor
3. Delete node k

Binary **Search** Trees (BST)

BST-delete(k)

k is a sub-tree and successor has no children

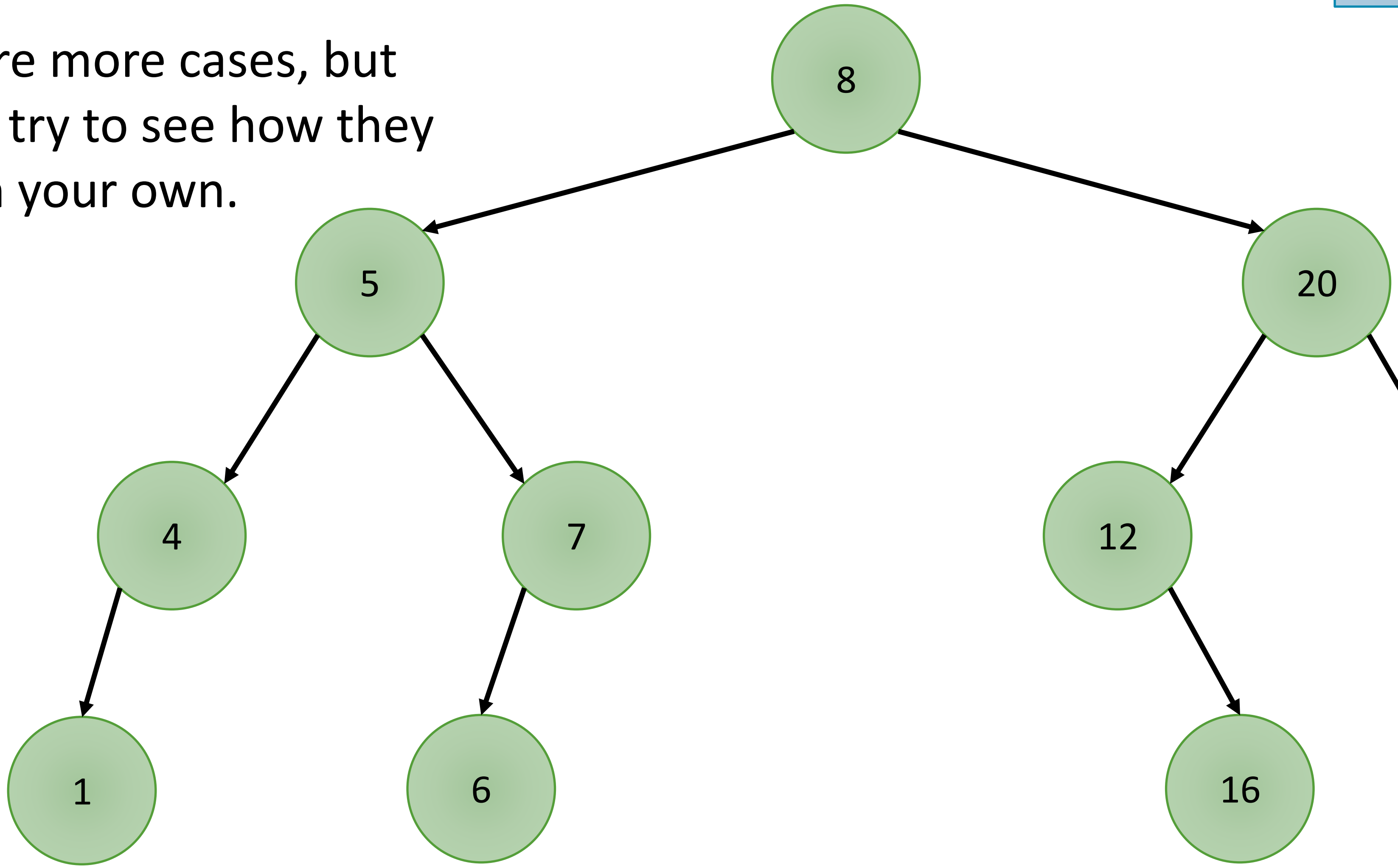


1. BST-search(k)
2. Replace k with successor
3. Delete node k

Binary **Search** Trees (BST)

BST-delete(k)

There are more cases, but you can try to see how they work on your own.



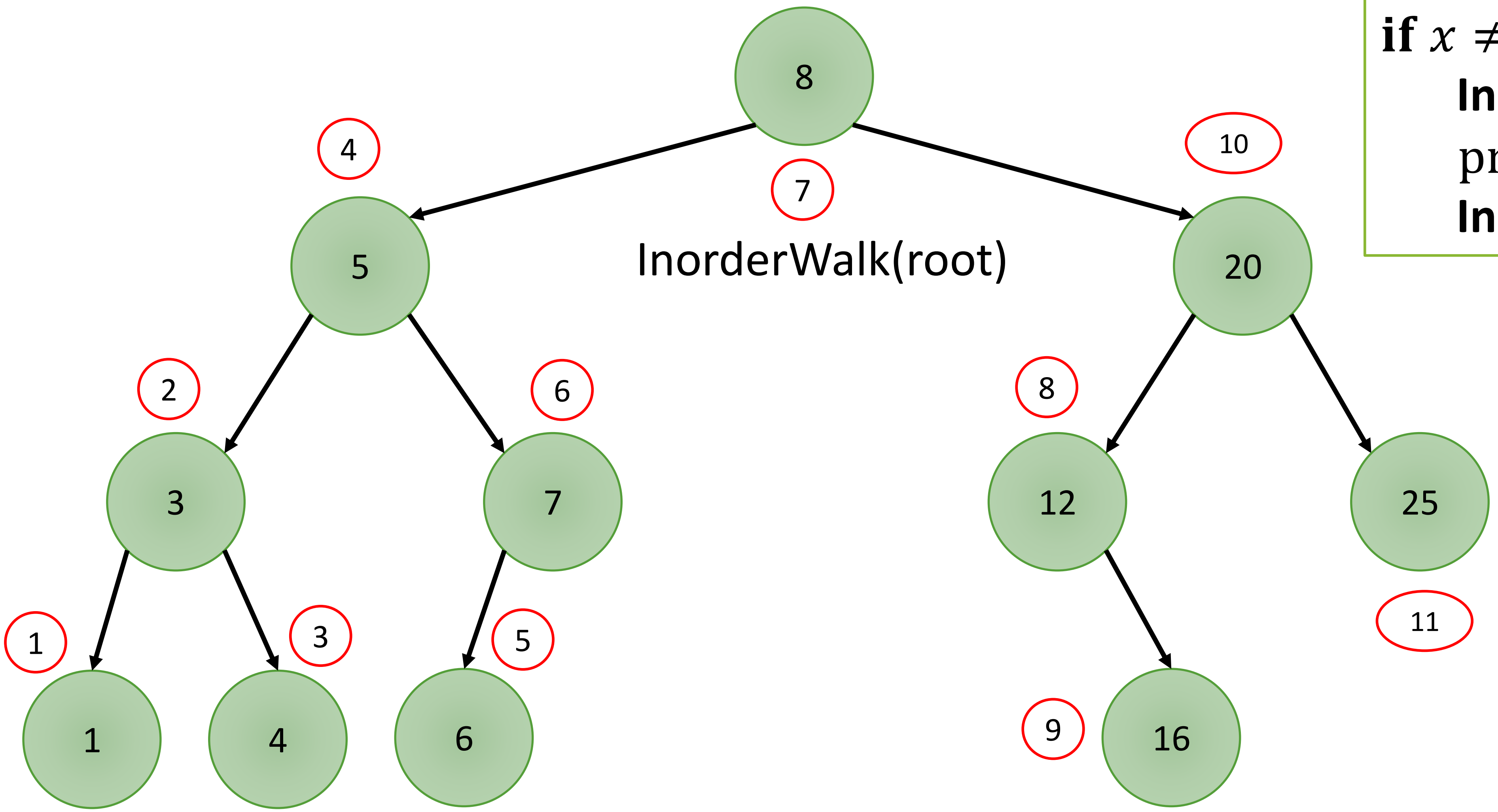
$T(n) = O(h)$

Worst-case
$T(n) = \Theta(n)$

Binary **Search** Trees (BST)

In-Order Traversal

InorderWalk(x):
if $x \neq \text{NULL}$
    InorderWalk$(x.left)$
    $\text{print}(x.key)$
    InorderWalk$(x.right)$



$$T(n) = O(n)$$

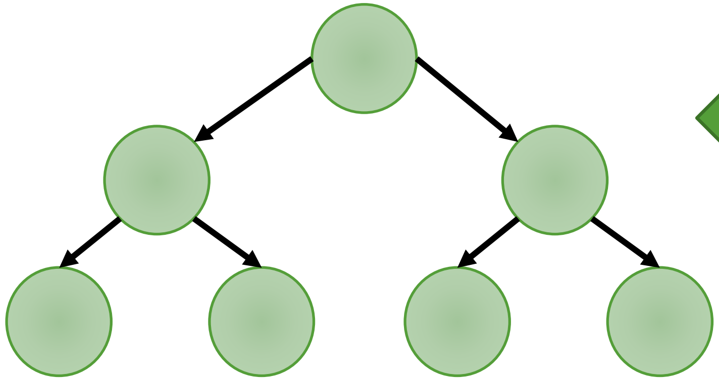
Assuming Tree is Balanced!

BST Sort

$$T(n) = \Theta(n \lg n) + \Theta(n) = \Theta(n \lg n)$$

A								
	2	8	7	1	3	5	6	4

Create BST
(n BST-inserts)
$\Theta(n \lg n)$



In-order Traversal
$\Theta(n)$

1	2	3	4	5	6	7	8

BST Sort

- In order to make sure that the tree is balanced, we can use various other tree-based data structures:
  - Red-black Trees
  - AVL Trees (better than RB-trees for search-intensive applications)
  - Splay Trees (pushes frequently accessed elements closer to the root; useful for caching)
  - Treap (a combination of a tree and a heap)
  - B-Trees (for large multi-degree data)
  - 2-3 / 2-3-4 trees (special cases of B-trees)

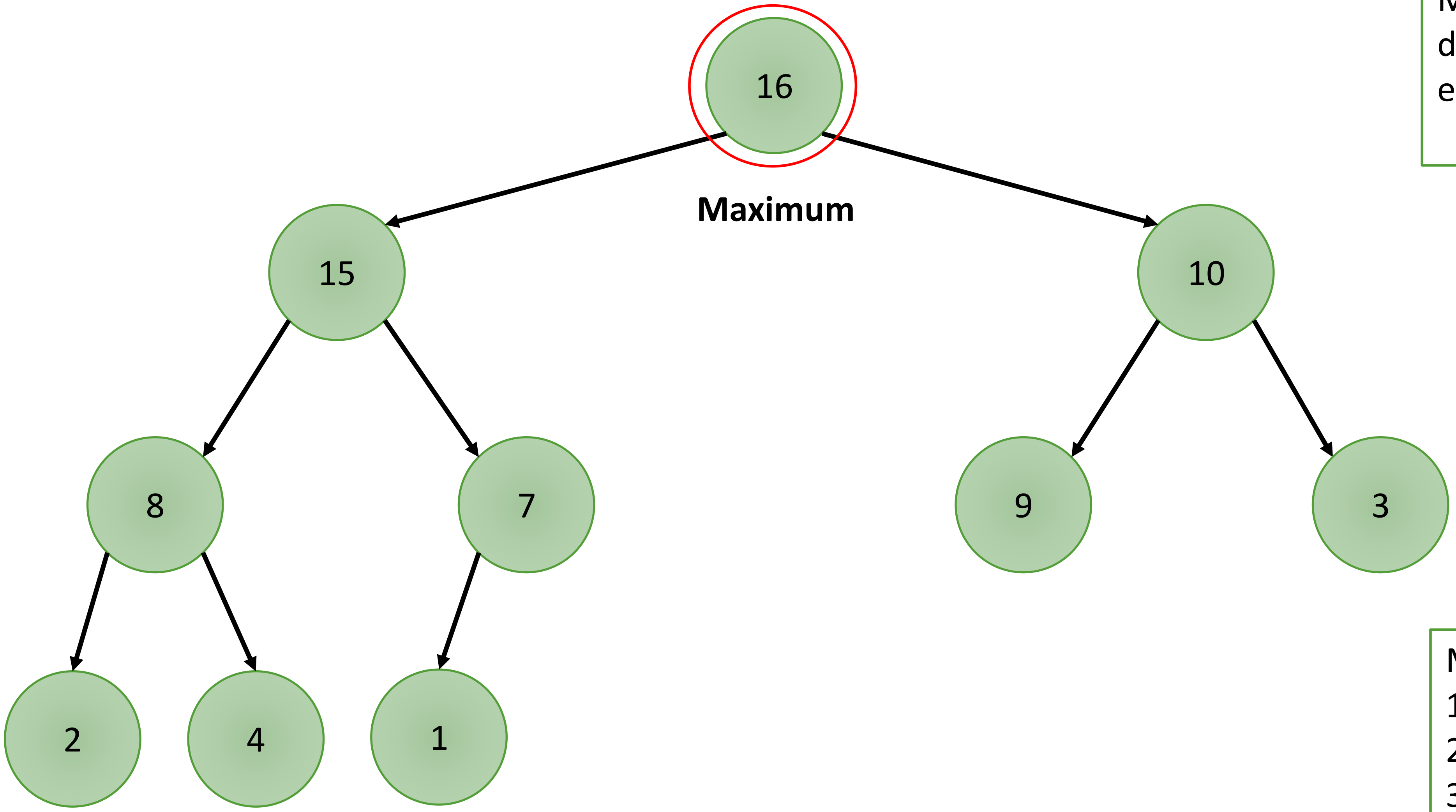
- But on average, randomly inserting elements in a standard BST gives $O(n \lg n)$

Heapsort: At a glance

- Worst-case $O(n \lg n)$ - like merge-sort, but unlike insertion sort
- Sorts in place – like insertion, but unlike merge-sort

- Heapsort combines the best attributes of these deterministic sorting algorithms.

Binary Max-Heap

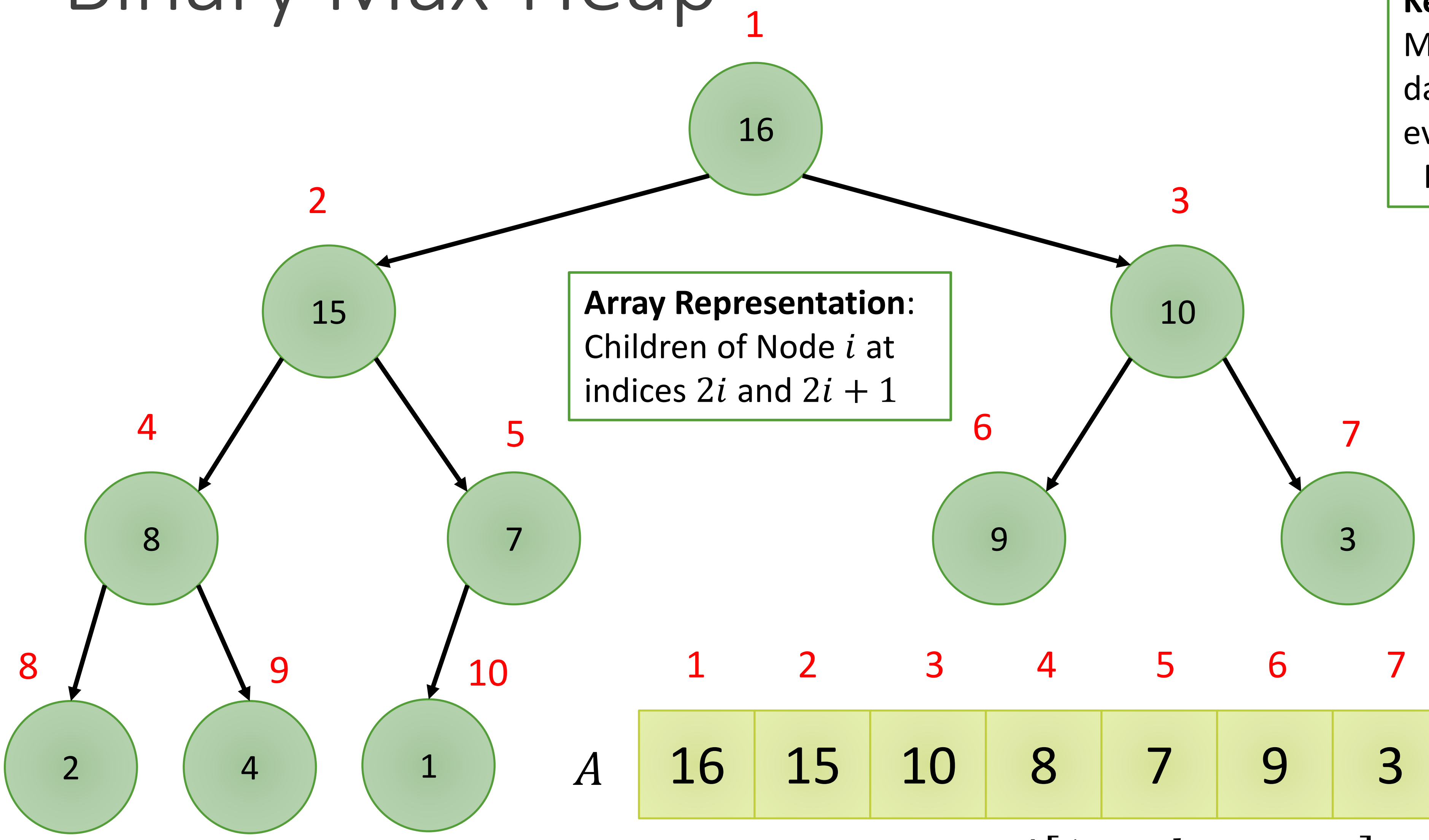


Key Property:
Max-Heap a binary tree data structure where for every node x:
Parent$(x).key \geq x.key$

Max-Heap operations:
1.) HEAPIFY(i)
2.) HEAP-getmax
3.) HEAP-insert(k)

Binary Max-Heap

Key Property:
Max-Heap a binary tree data structure where for every node x:
Parent$(x).key \geq x.key$

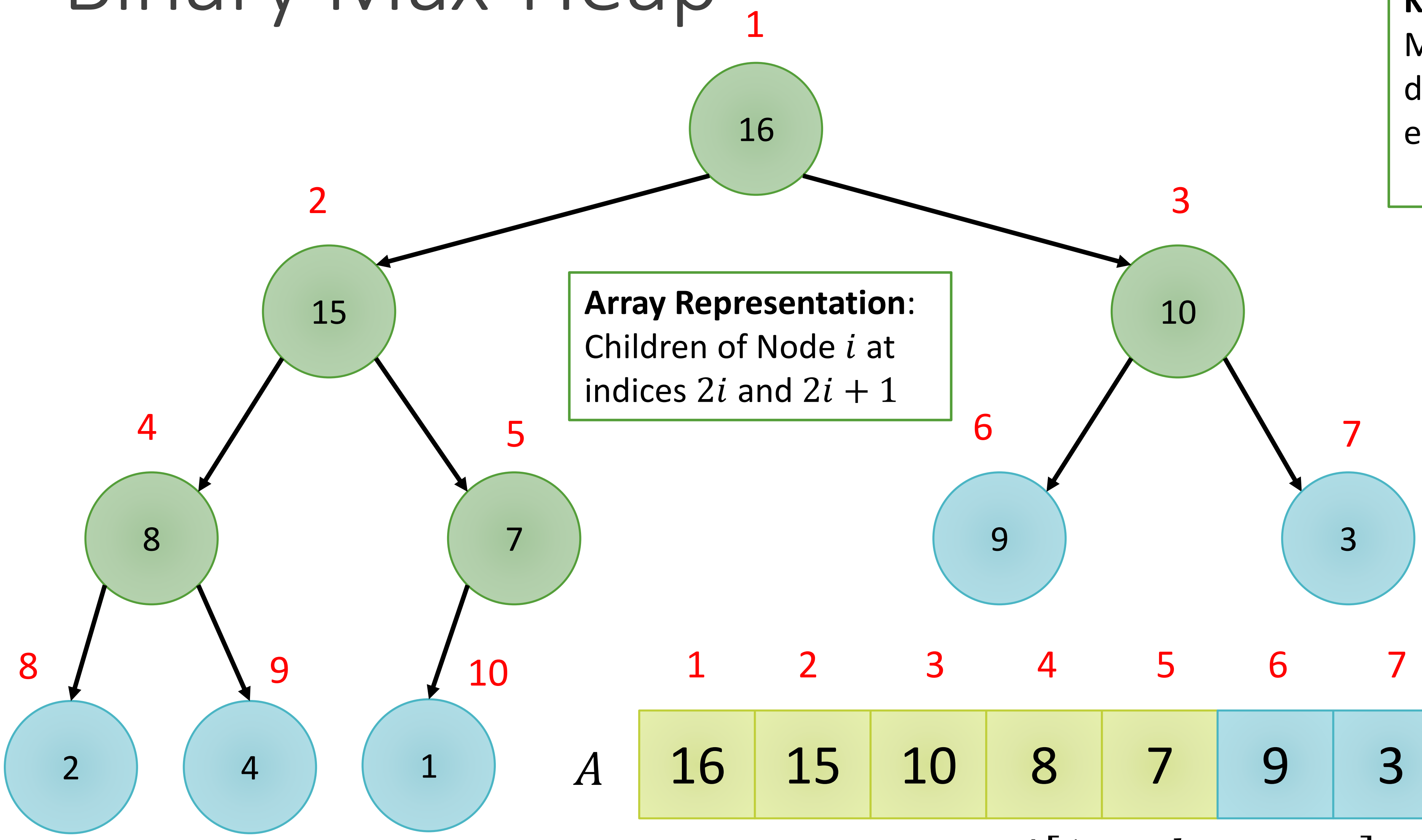


Array Representation:
Children of Node i at indices $2i$ and $2i+1$

	1	2	3	4	5	6	7	8	9	10
A	16	15	10	8	7	9	3	2	4	1

$A[1, \ldots, heapsize]$

Binary Max-Heap



Array Representation: Children of Node i at indices $2i$ and $2i+1$

Key Property: Max-Heap a binary tree data structure where for every node x:
 Parent$(x).key \geq x.key$

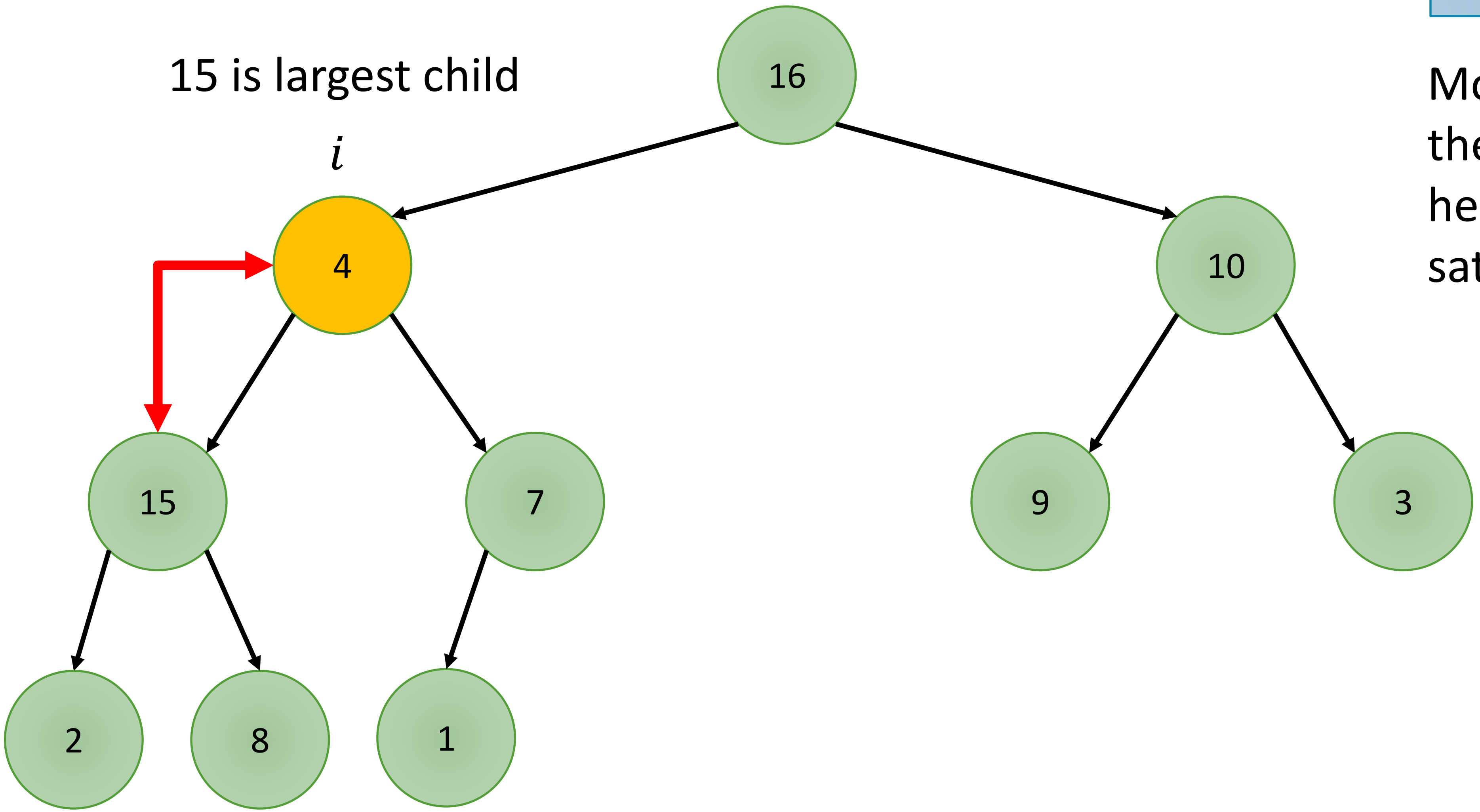
Number of leaves in a heap is $\lceil n/2 \rceil$

	1	2	3	4	5	6	7	8	9	10
A	16	15	10	8	7	9	3	2	4	1

$A[1, \ldots, heapsize]$

Max-Heap

HEAPIFY(i)

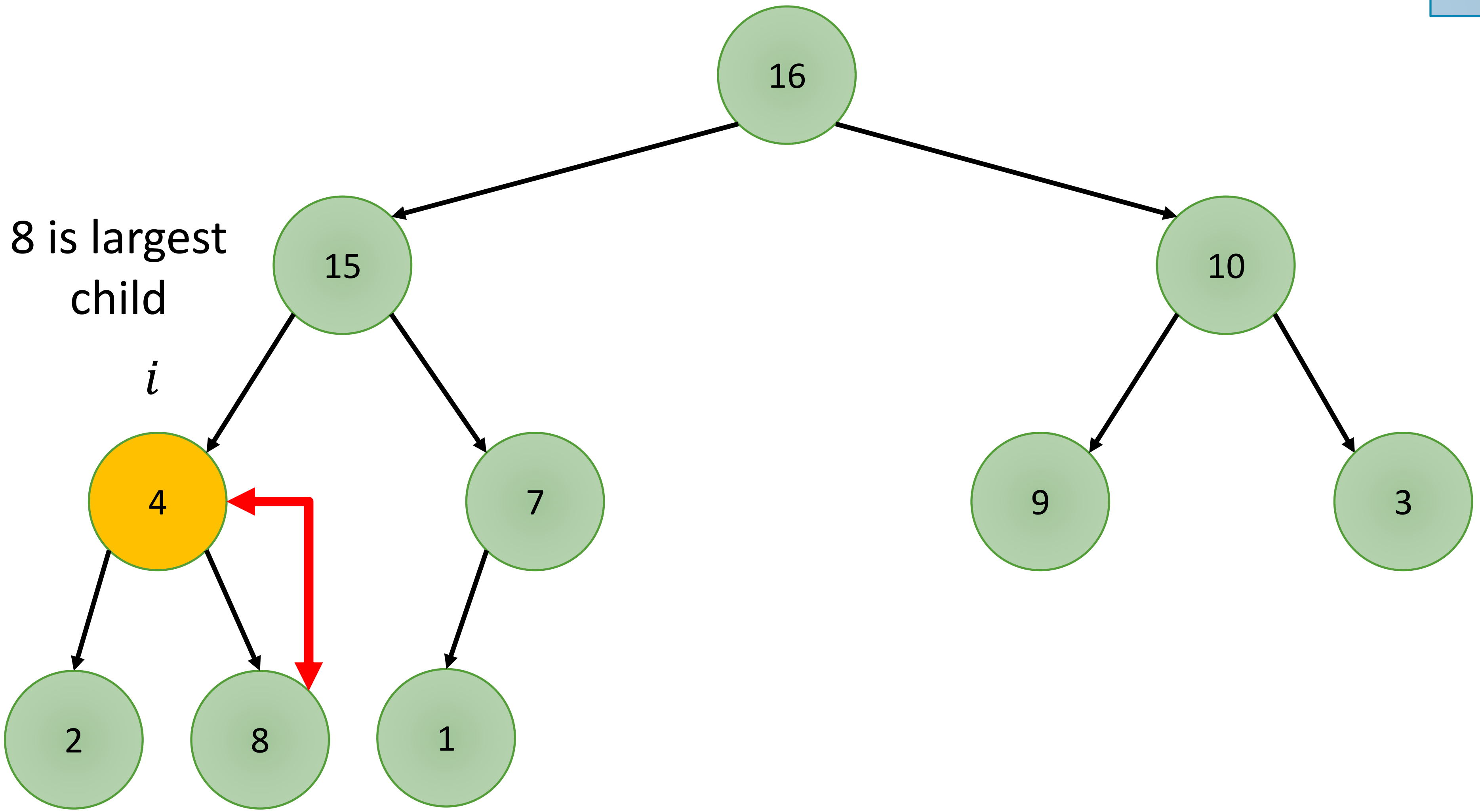


Move node i down the tree until the heap property is satisfied.

Max-Heap

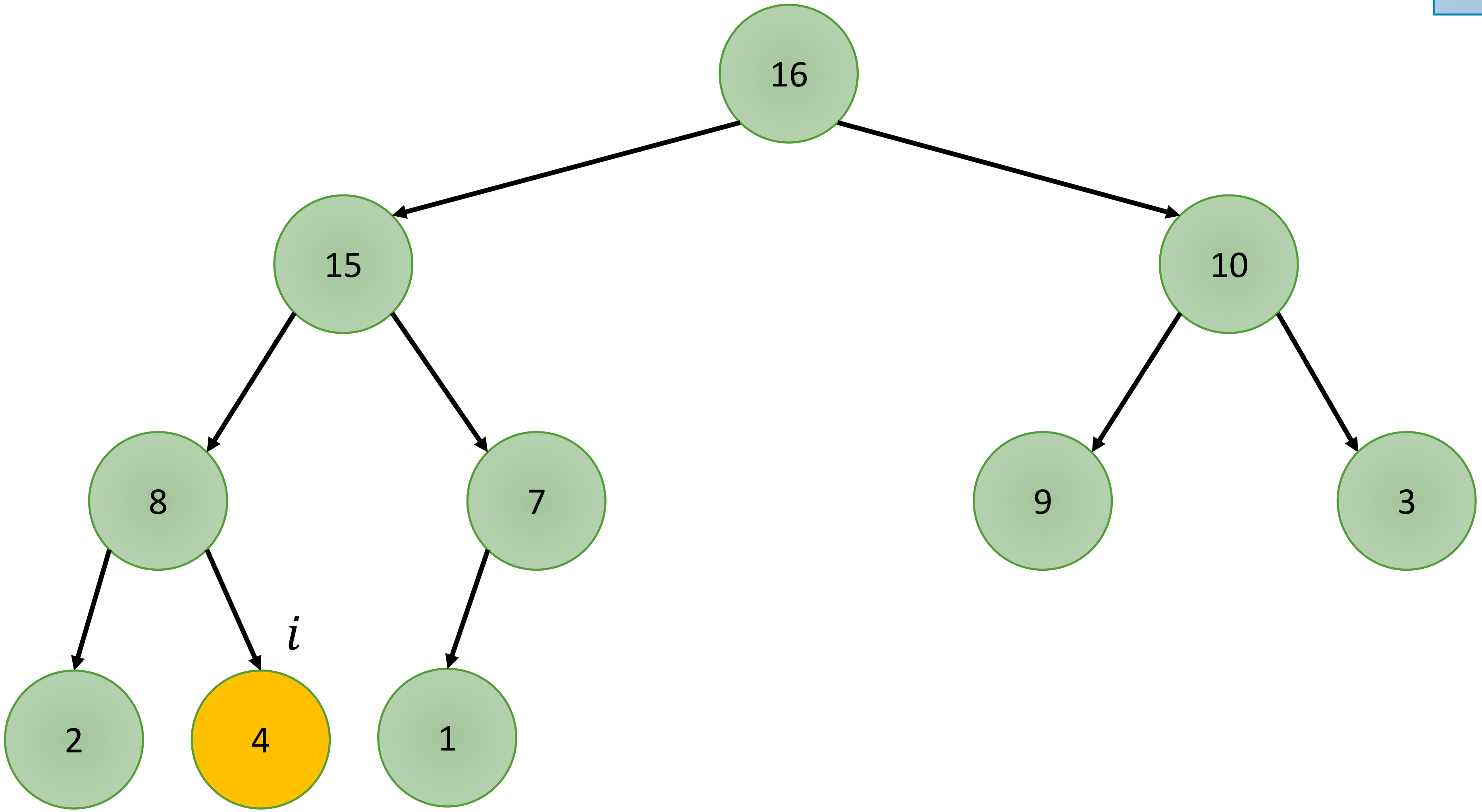
HEAPIFY(i)

8 is largest child



Max-Heap

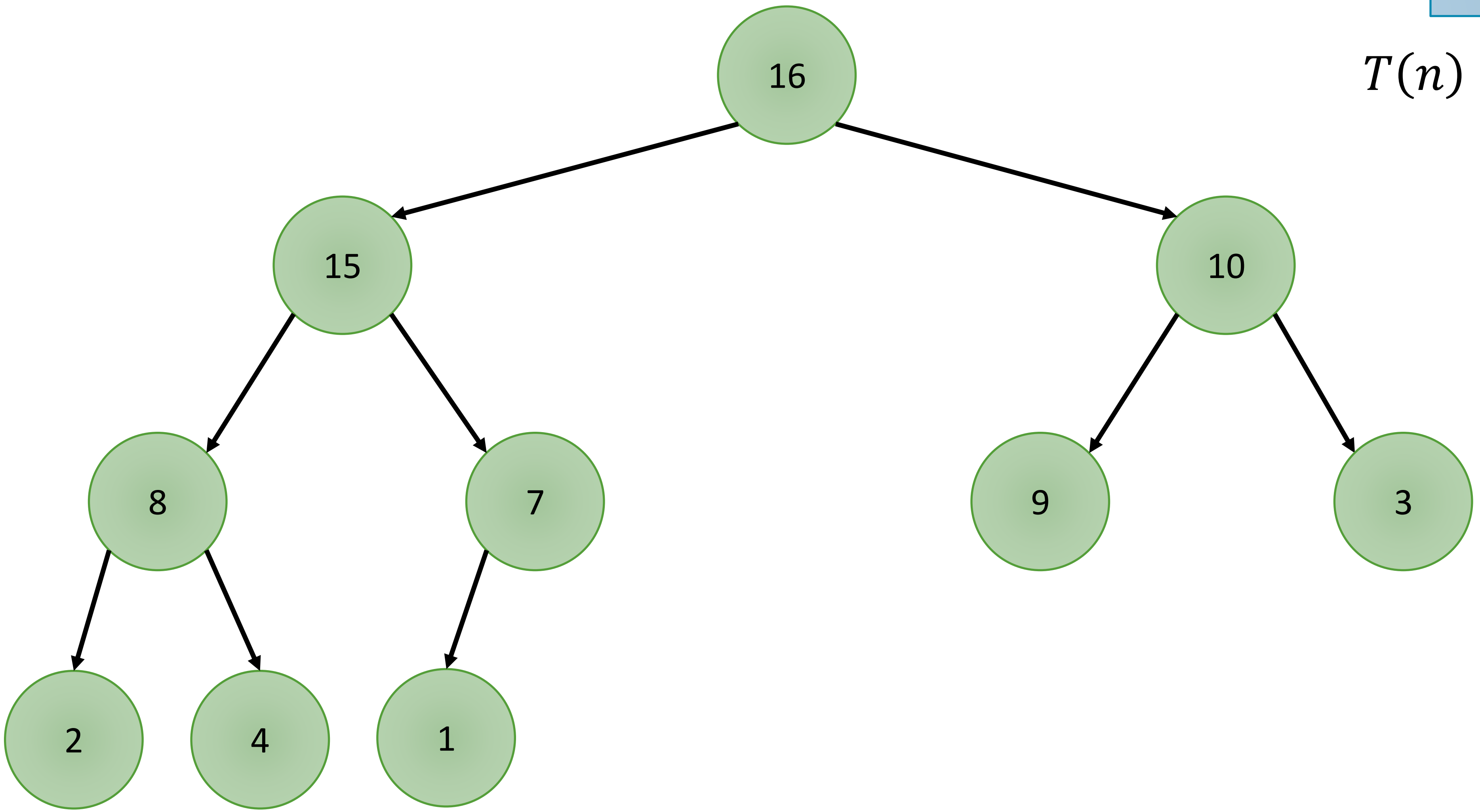
HEAPIFY(i)



Max-Heap

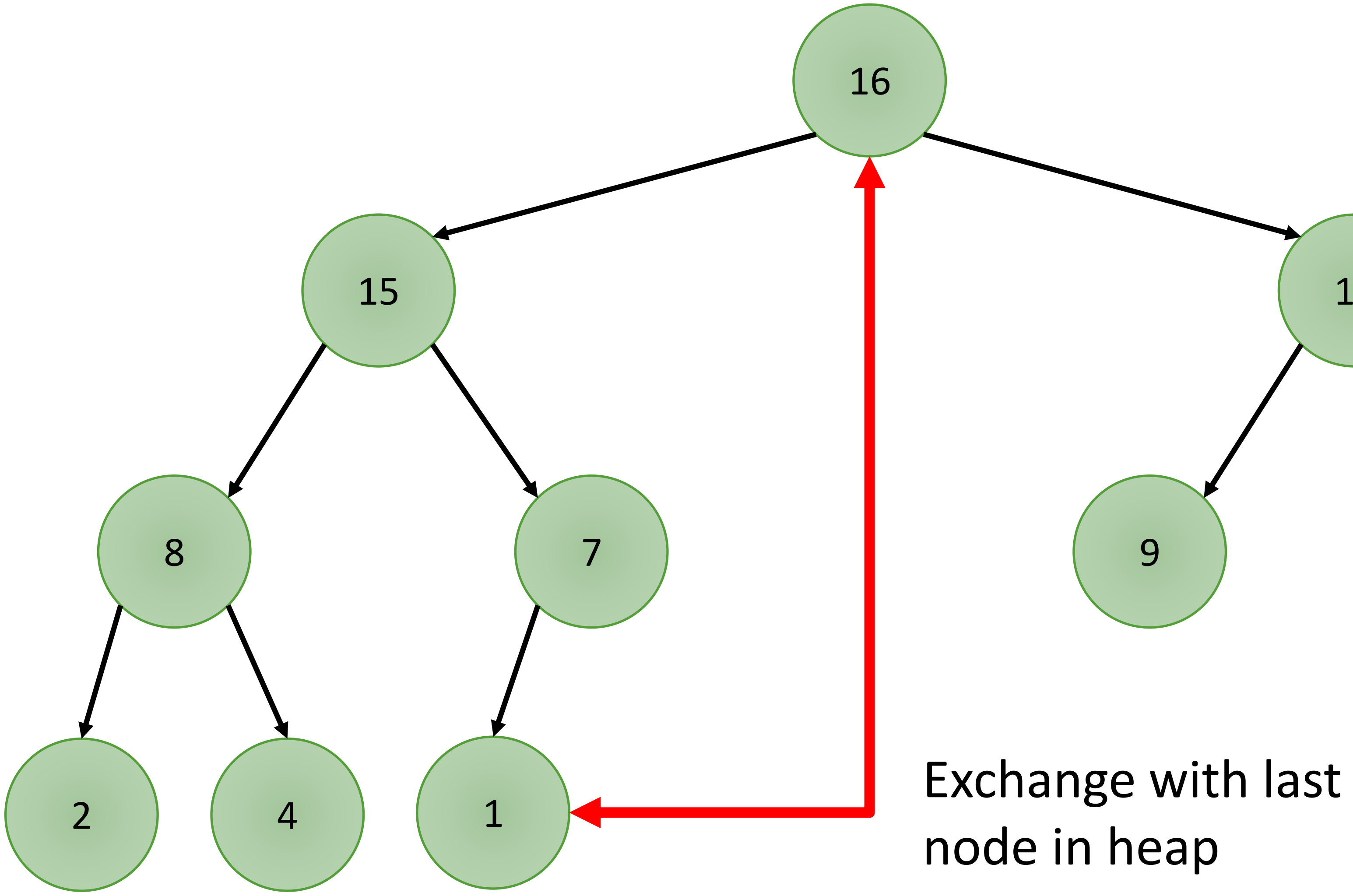
HEAPIFY(i)

$$T(n) = O(h) = O(\lg n)$$



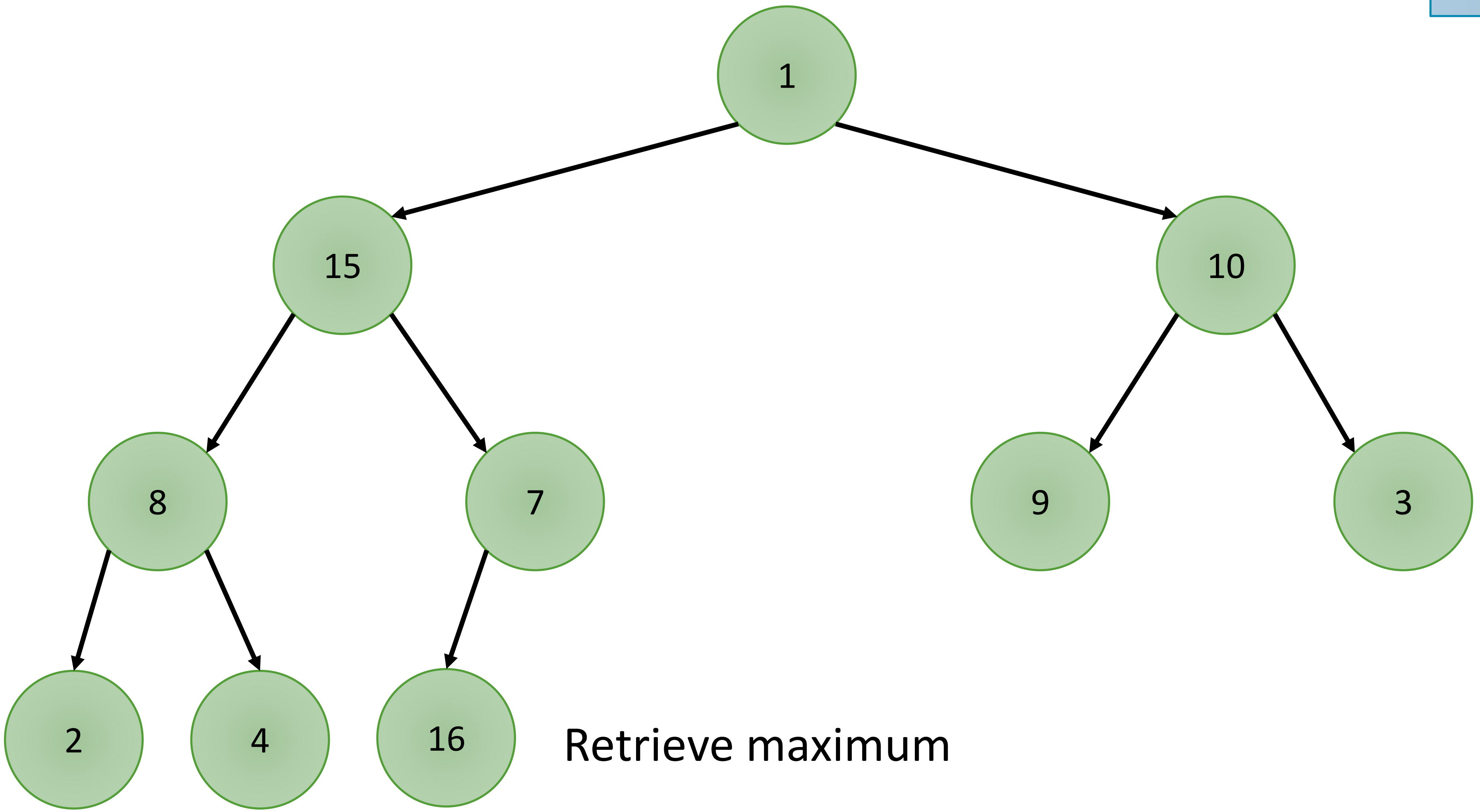
Max-Heap

HEAP-getmax



Max-Heap

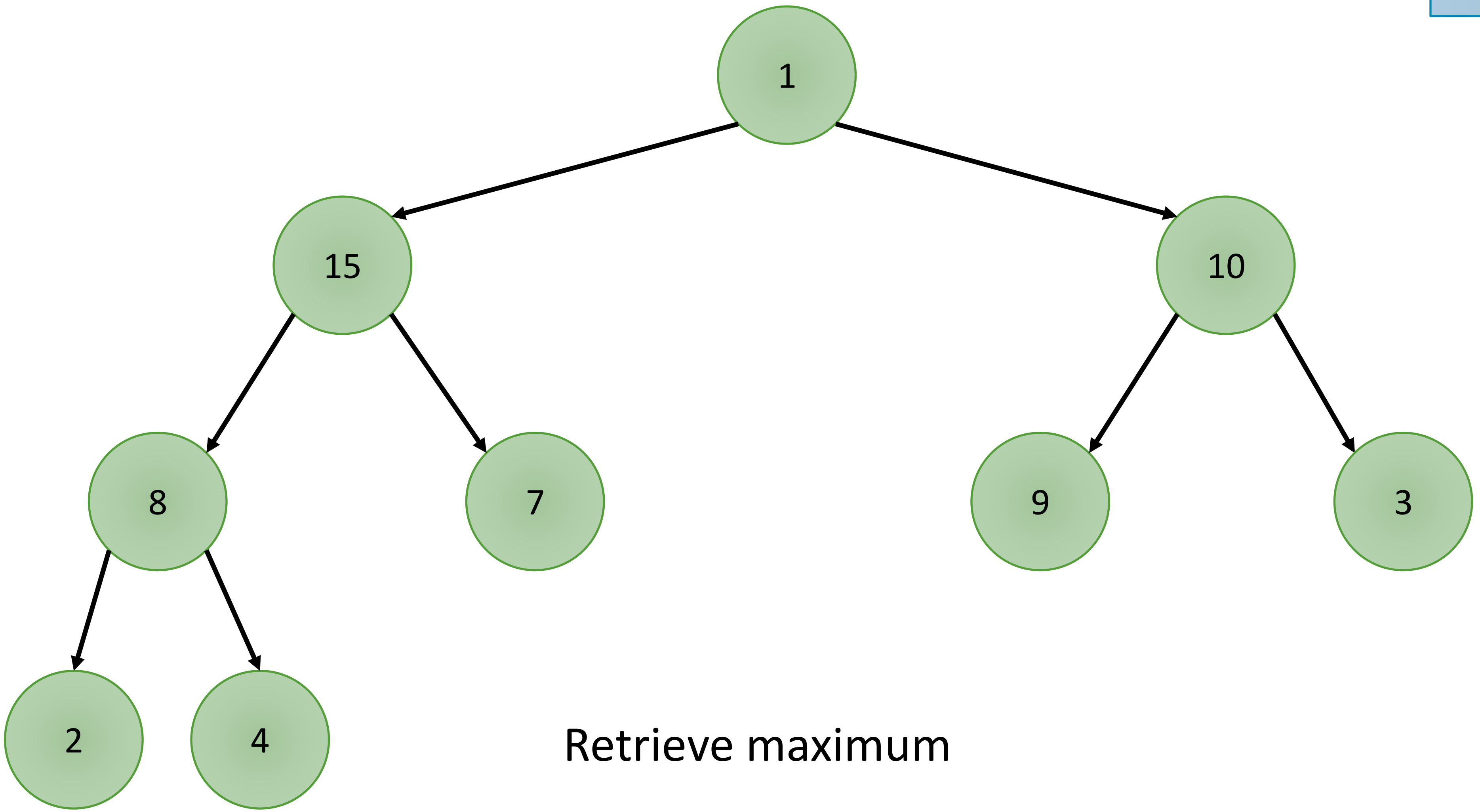
HEAP-getmax



Retrieve maximum

Max-Heap

HEAP-getmax

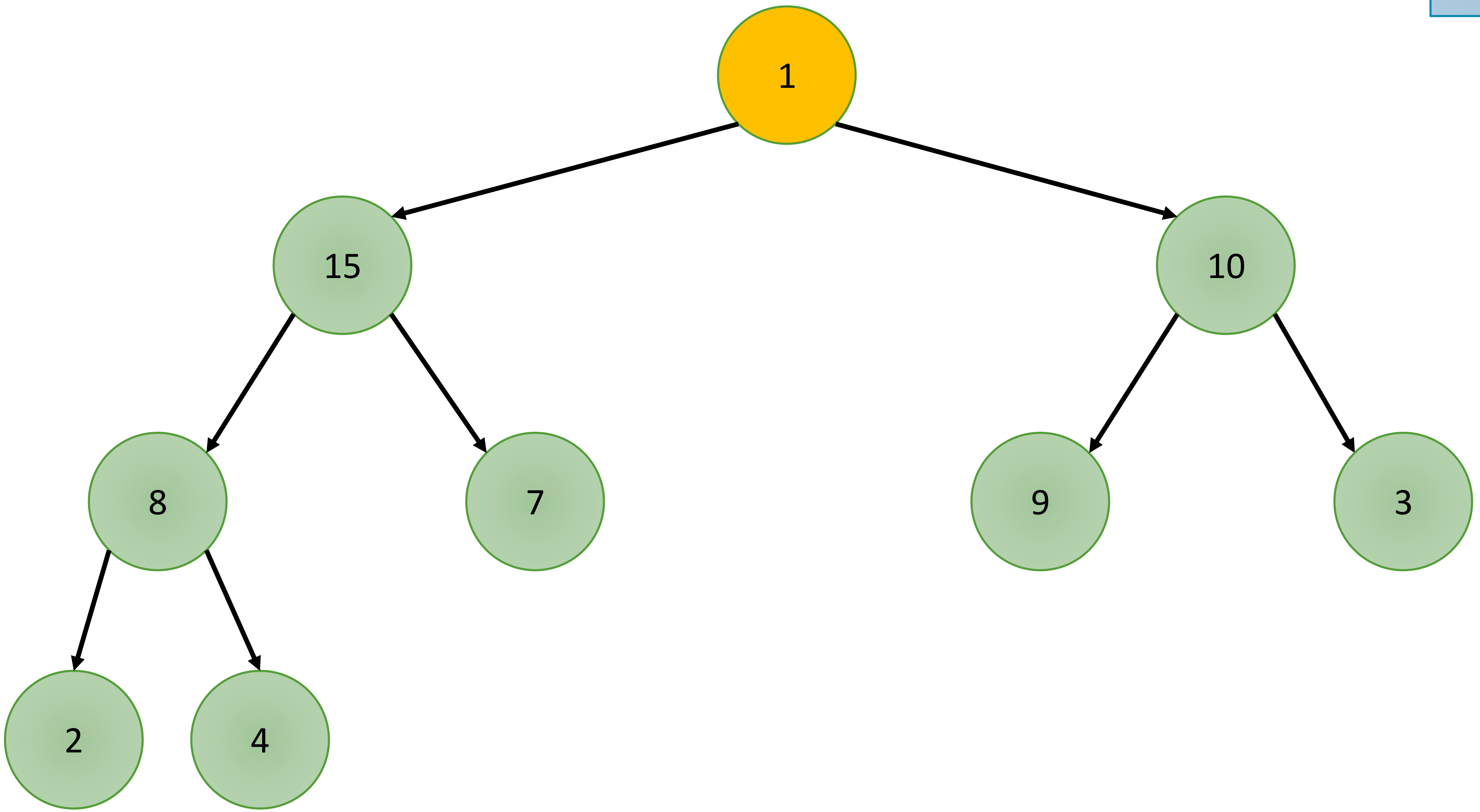


Retrieve maximum

Max-Heap

Call HEAPIFY(1)

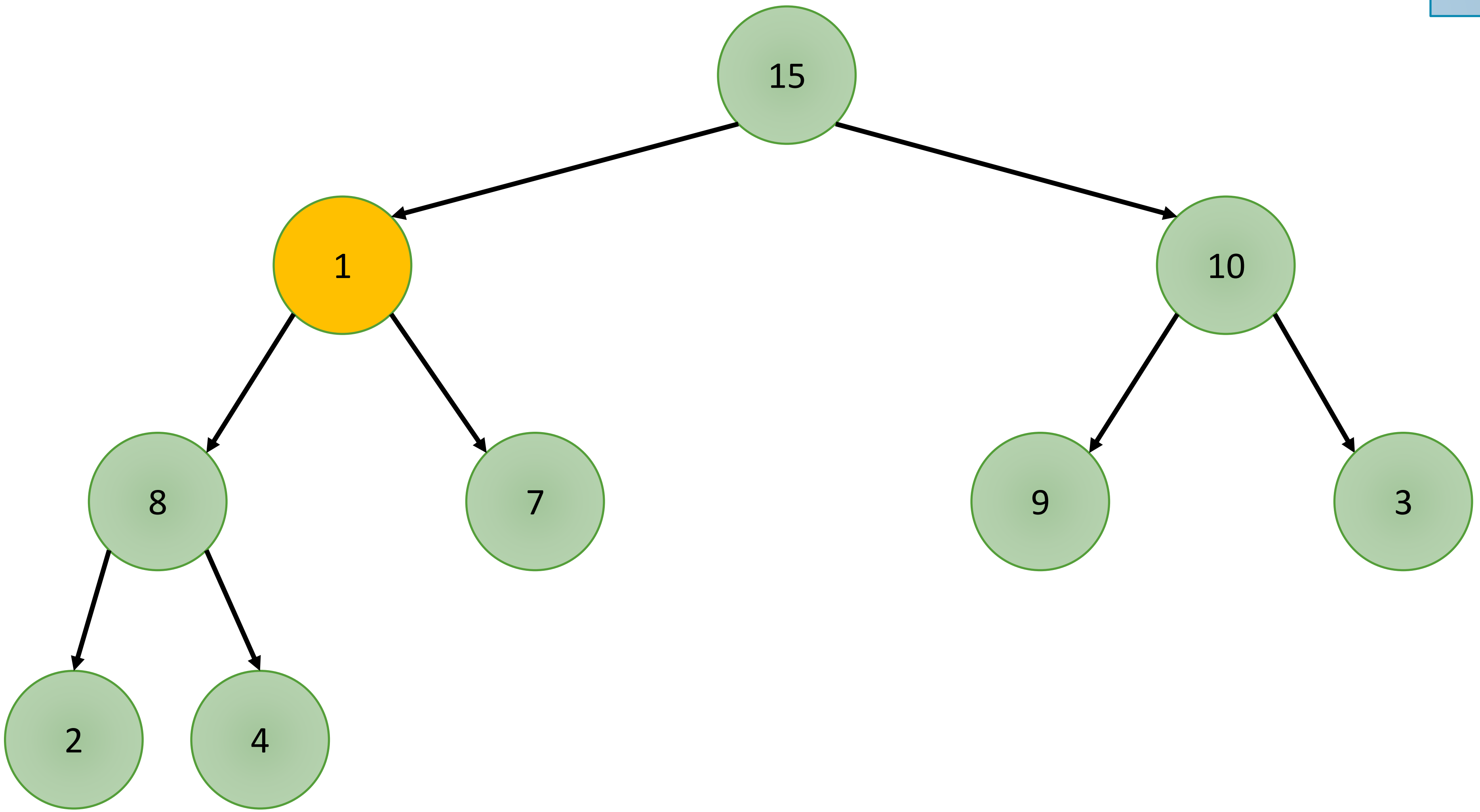
HEAP-getmax



Max-Heap

Call HEAPIFY(1)

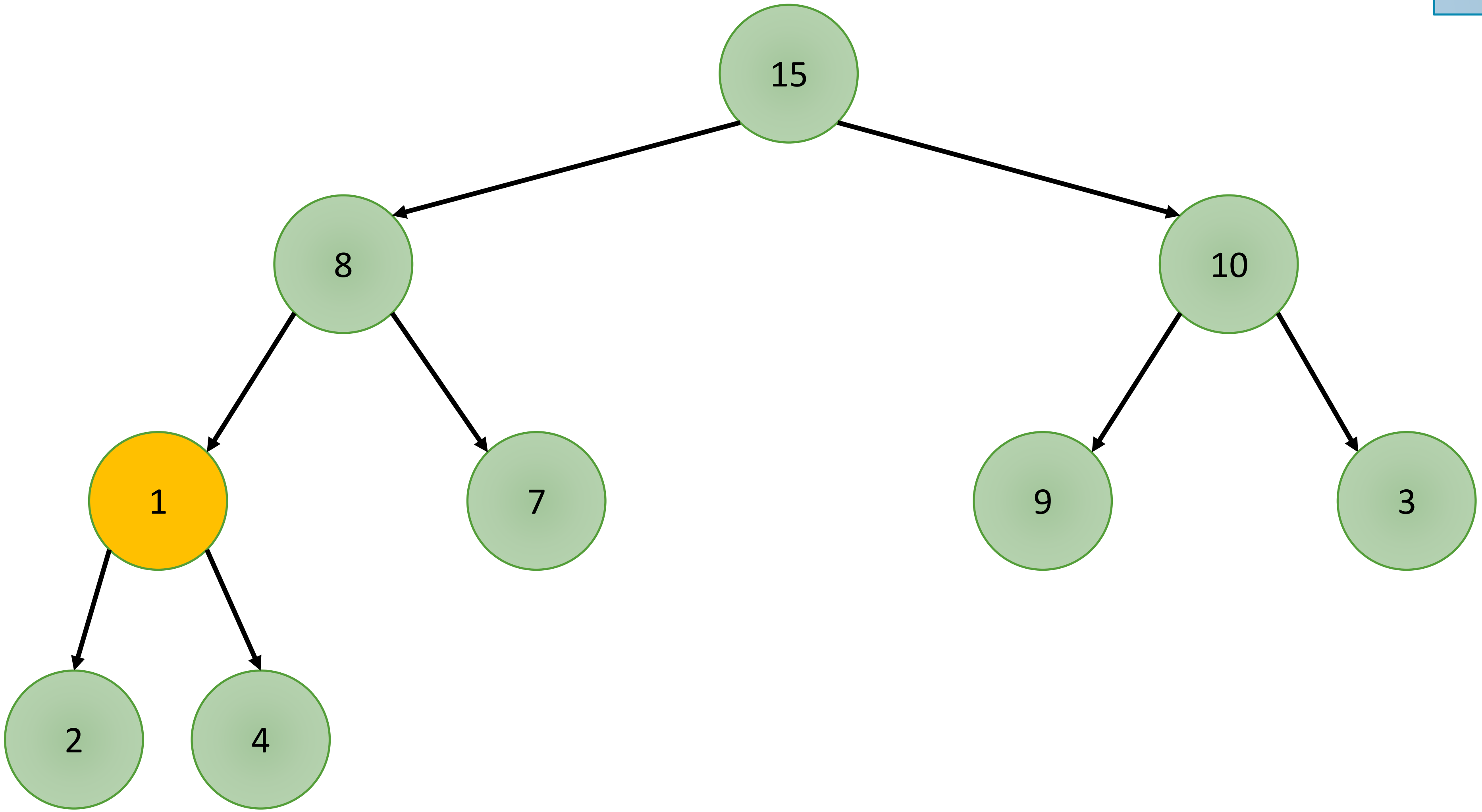
HEAP-getmax



Max-Heap

Call HEAPIFY(1)

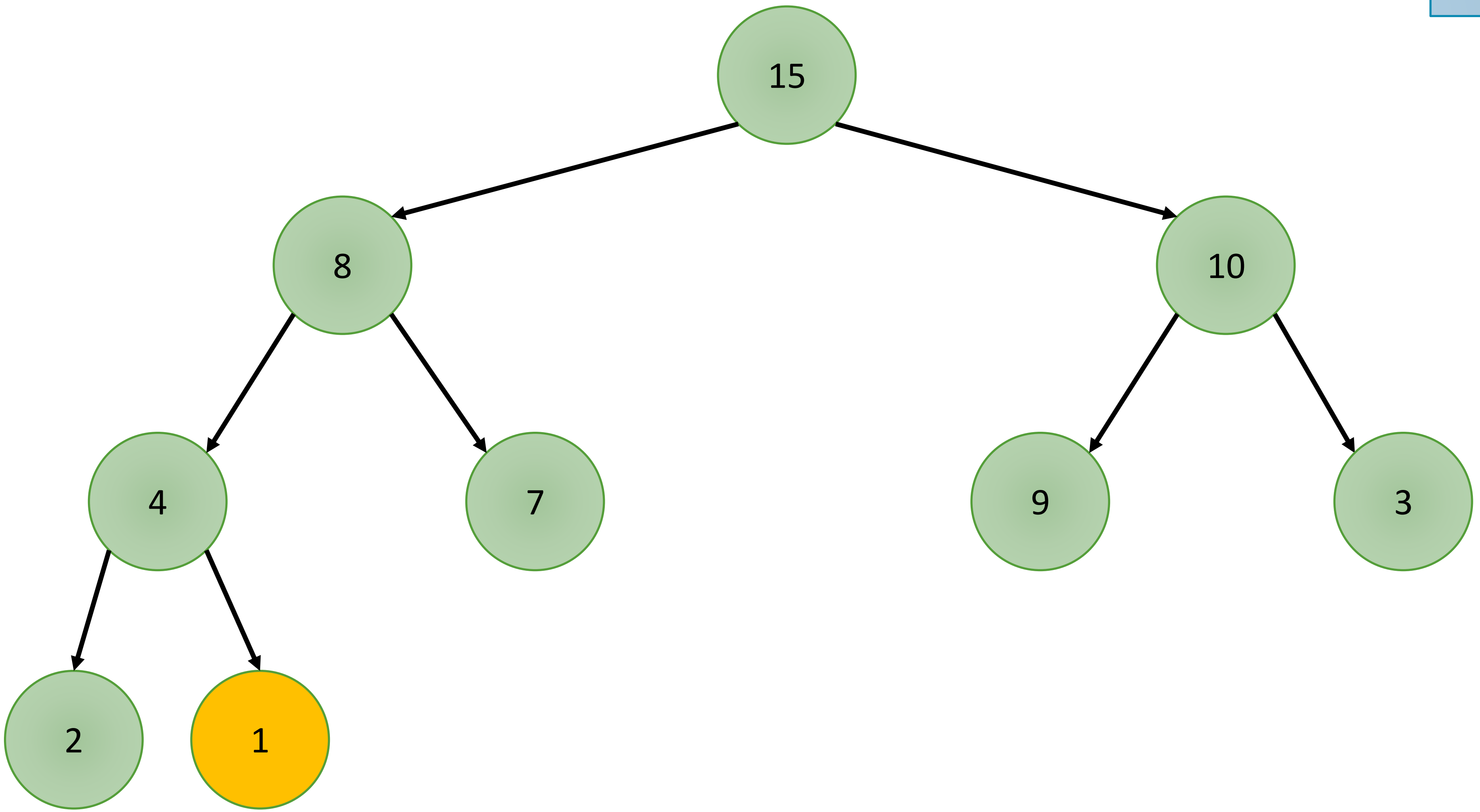
HEAP-getmax



Max-Heap

Call HEAPIFY(1)

HEAP-getmax

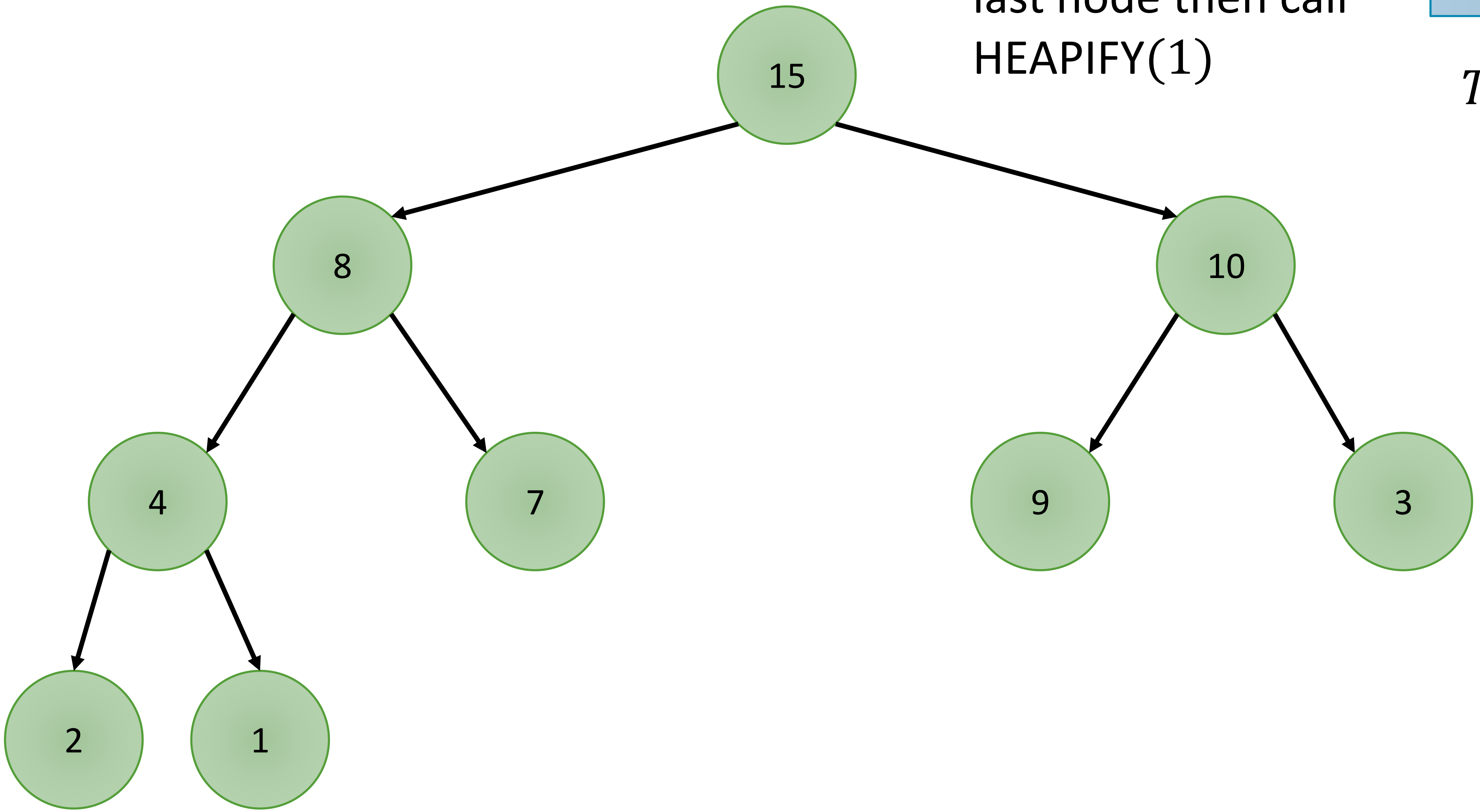


Max-Heap

Exchange $A[1]$ with last node then call HEAPIFY(1)

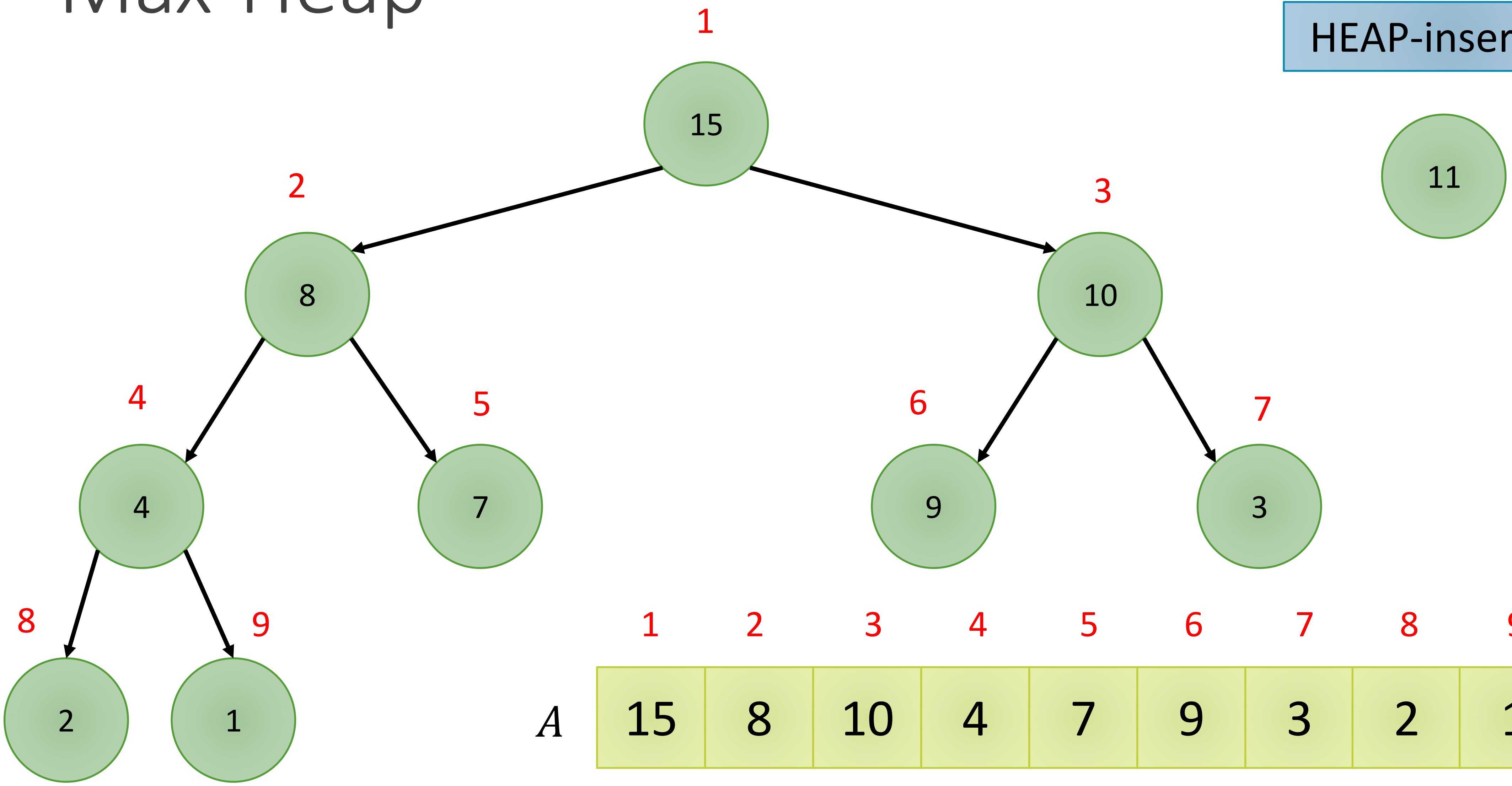
HEAP-getmax

$$T(n) = O(\lg n)$$



Max-Heap

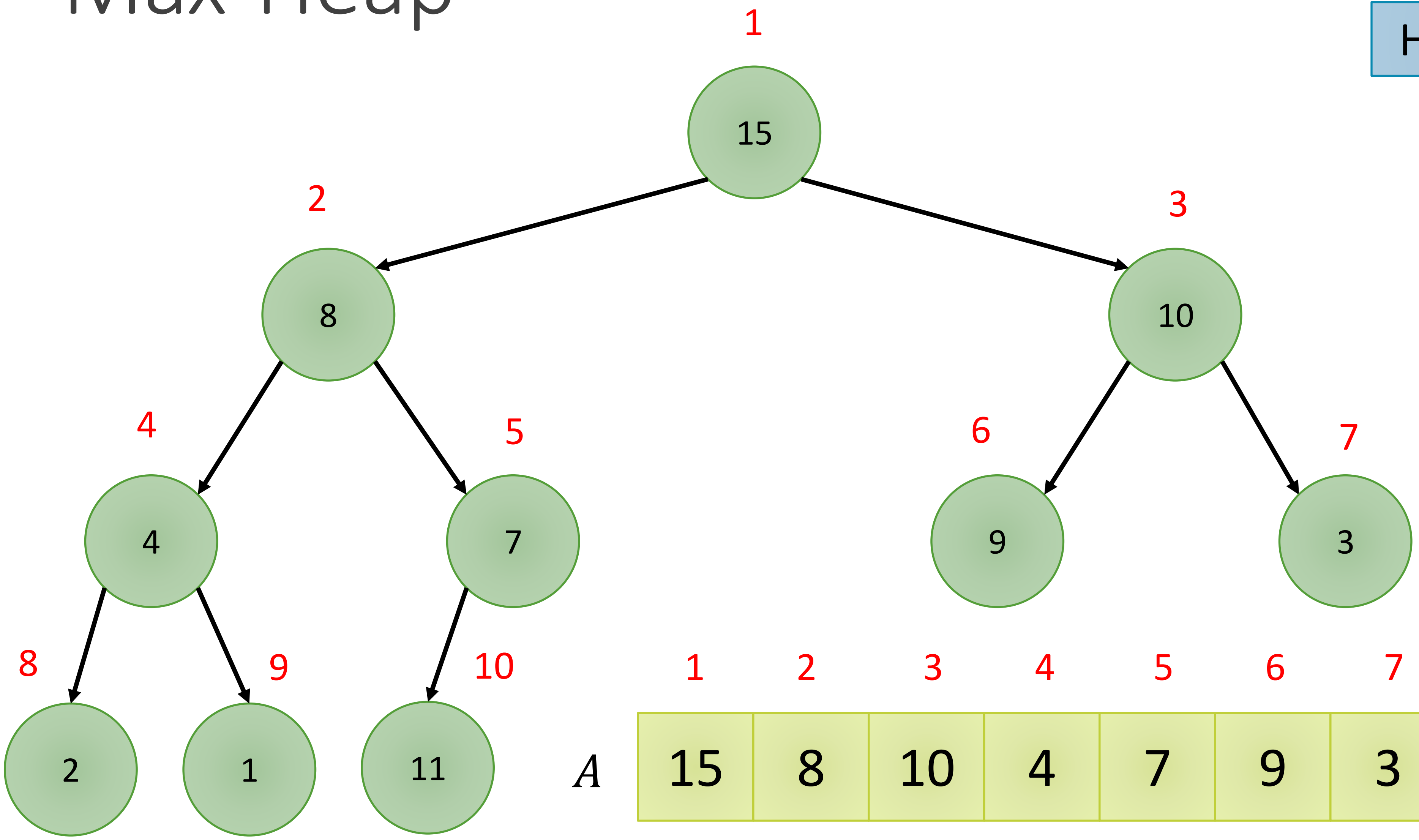
HEAP-insert(k)



	1	2	3	4	5	6	7	8	9
A	15	8	10	4	7	9	3	2	1

Max-Heap

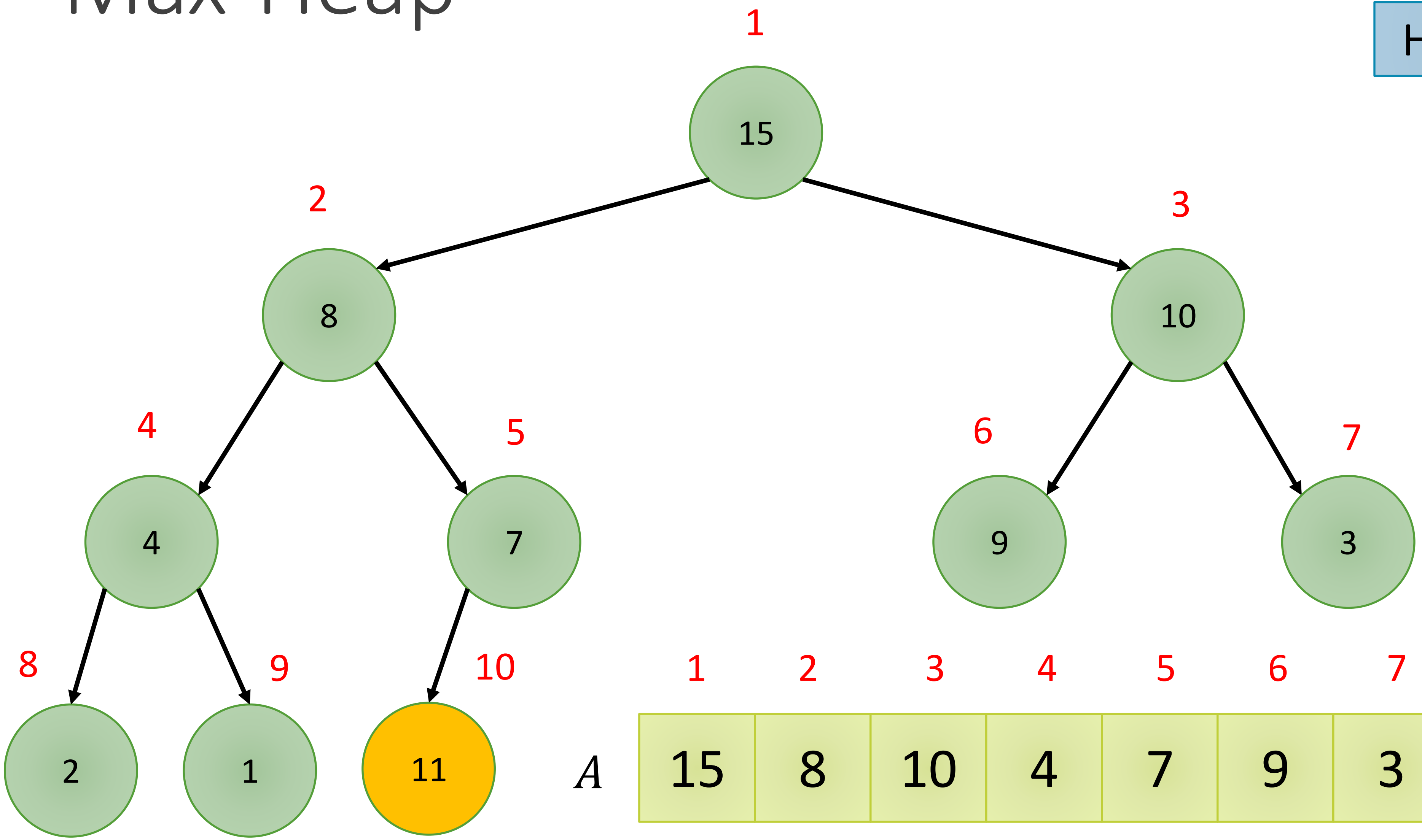
HEAP-insert(k)



	1	2	3	4	5	6	7	8	9	10
A	15	8	10	4	7	9	3	2	1	11

Max-Heap

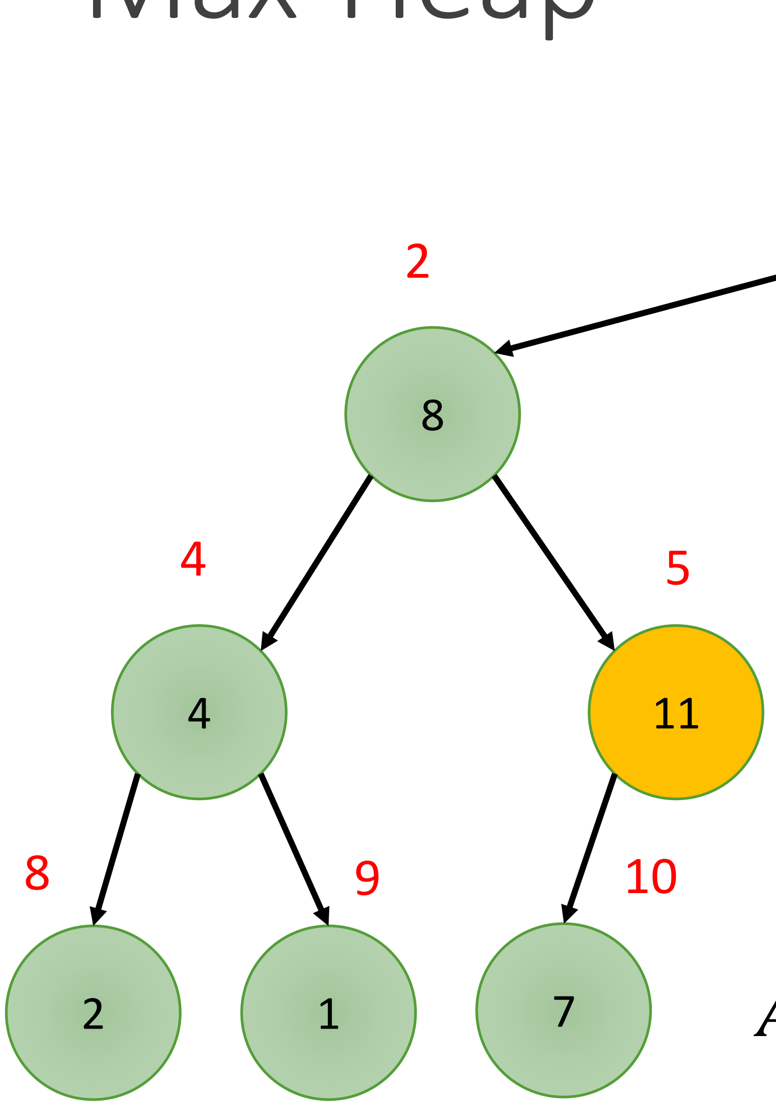
HEAP-insert(k)



	1	2	3	4	5	6	7	8	9	10
A	15	8	10	4	7	9	3	2	1	11

Max-Heap

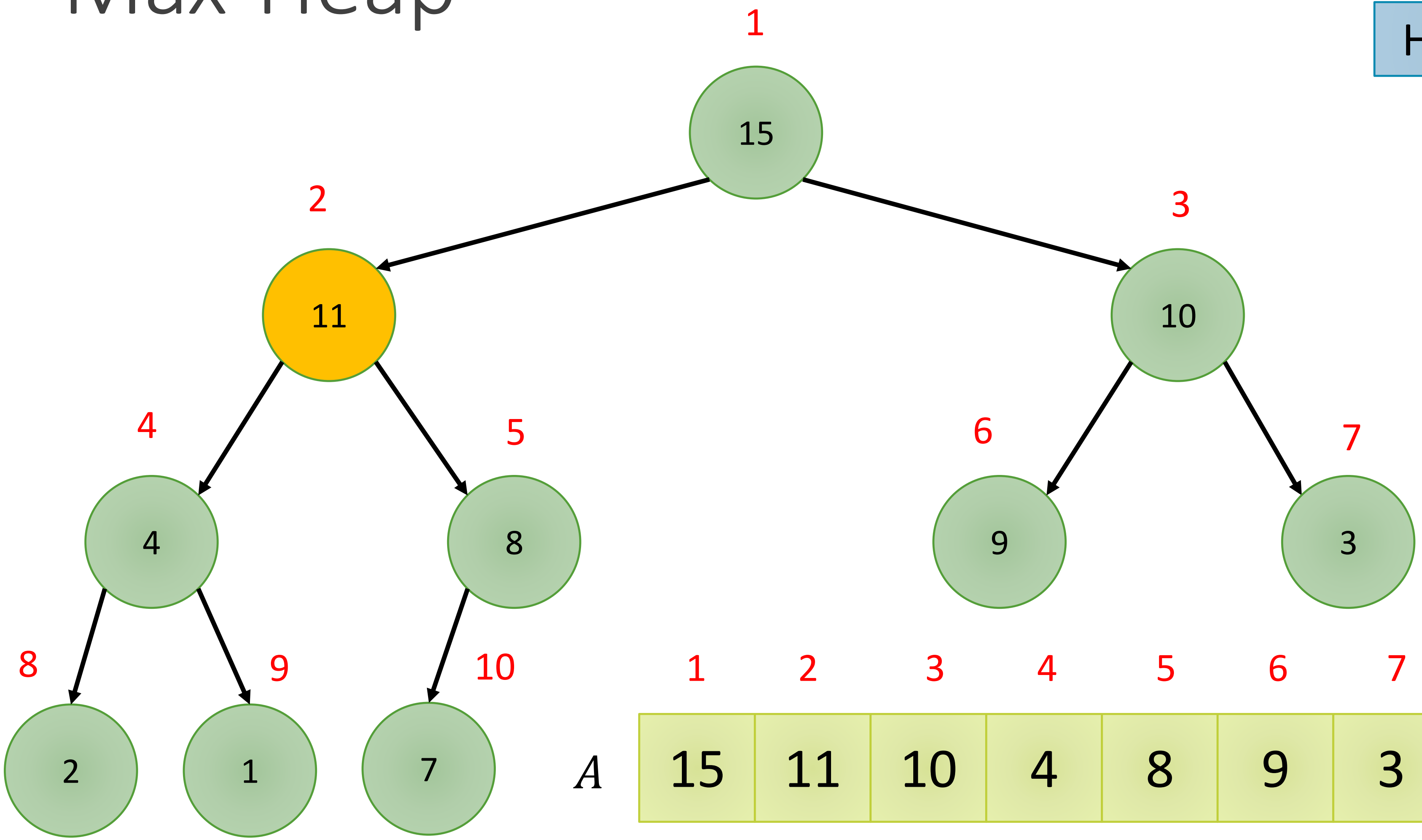
HEAP-insert(k)



	1	2	3	4	5	6	7	8	9	10
A	15	8	10	4	11	9	3	2	1	7

Max-Heap

HEAP-insert(k)

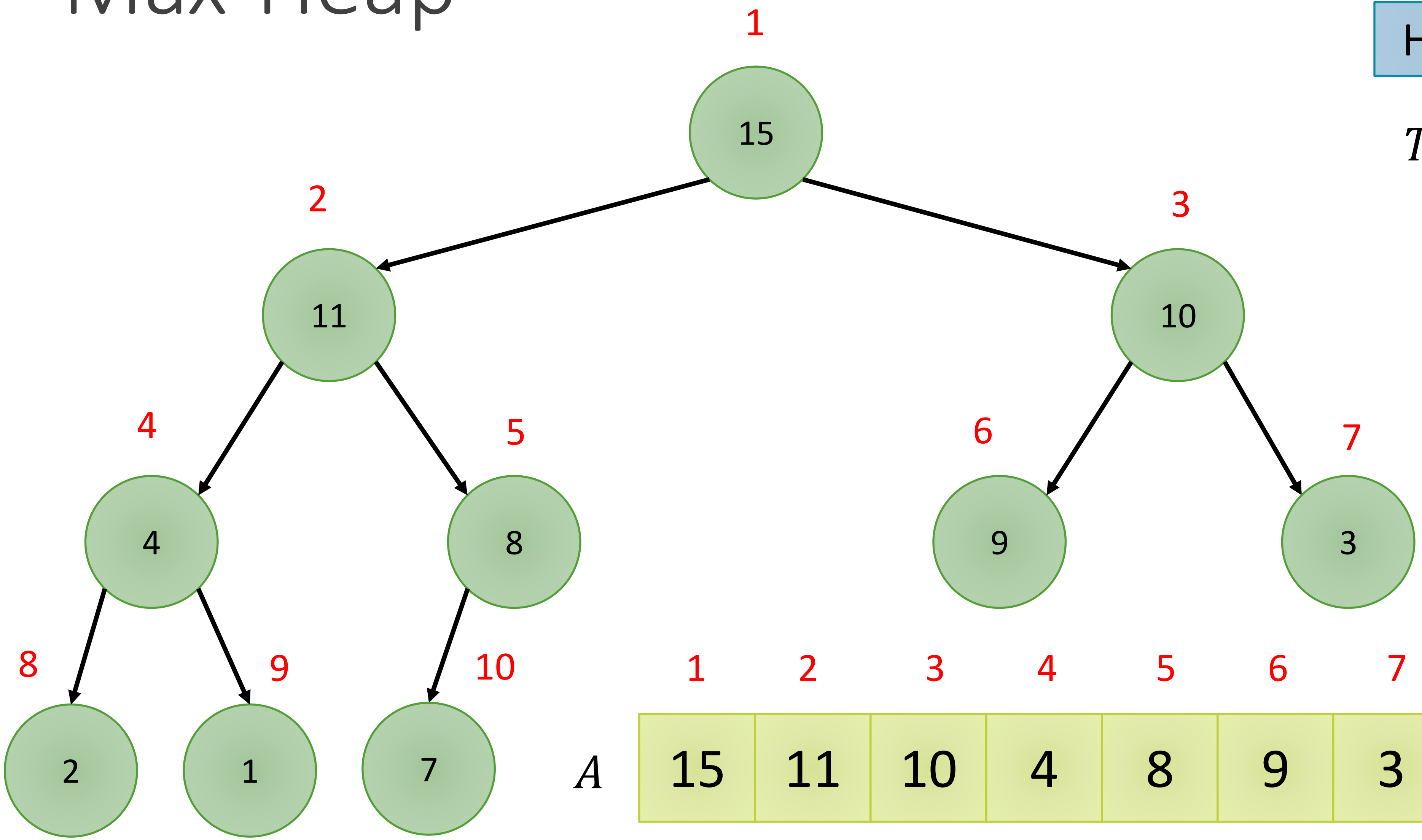


	1	2	3	4	5	6	7	8	9	10
A	15	11	10	4	8	9	3	2	1	7

Max-Heap

HEAP-insert(k)

$$T(n) = O(\lg n)$$

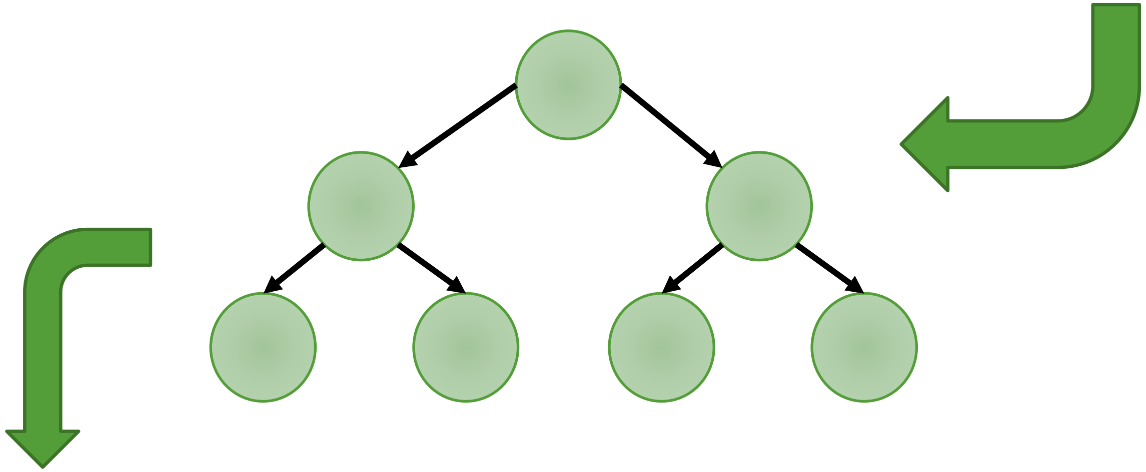


A	1	2	3	4	5	6	7	8	9	10
	15	11	10	4	8	9	3	2	1	7

Heapsort

$$T(n) = \Theta(n \lg n)$$

A	2	8	7	1	3	5	6	4



n HEAP-insert

$\Theta(n \lg n)$

n HEAP-getmax

$\Theta(n \lg n)$

1	2	3	4	5	6	7	8

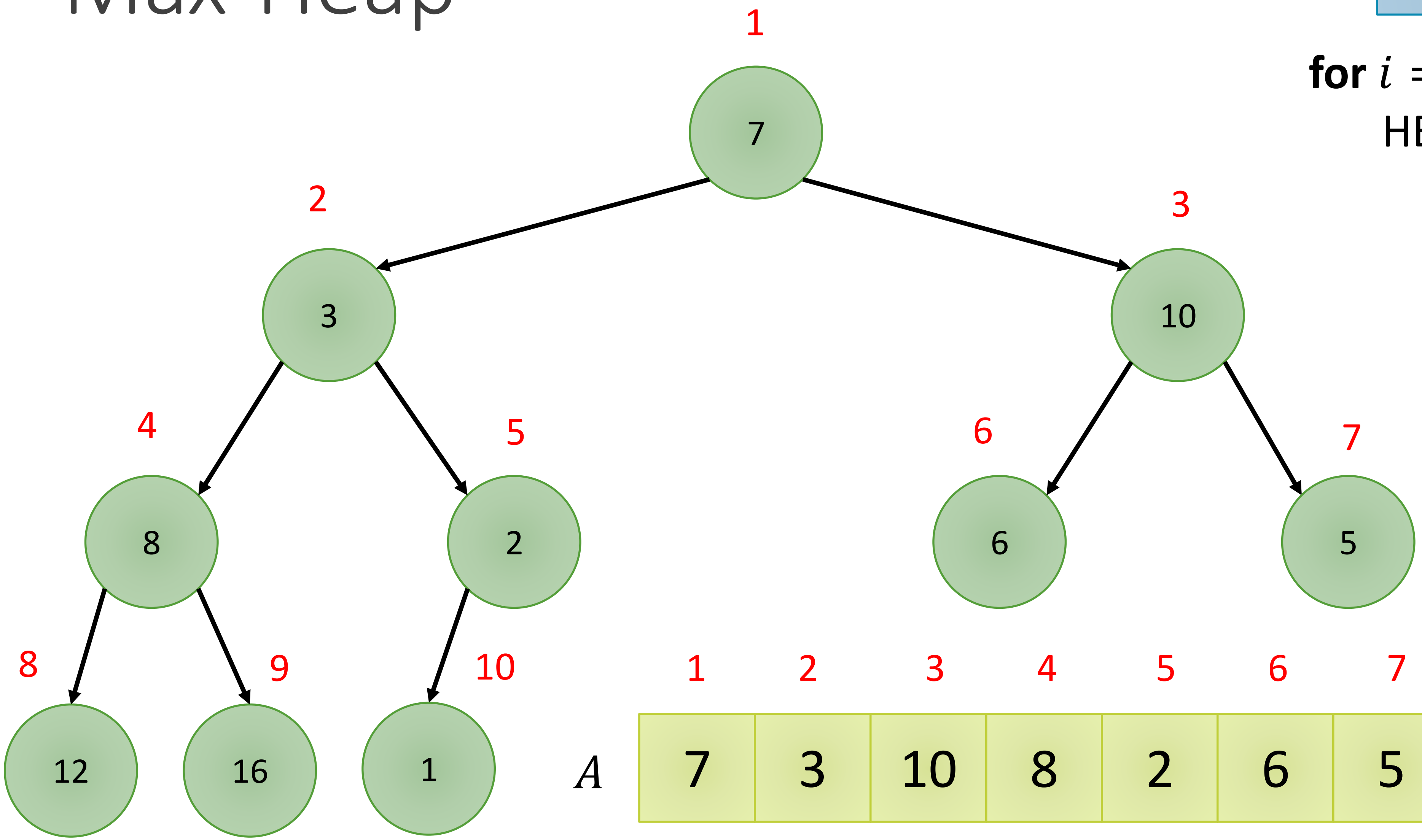
Reverse Order

$\Theta(n)$

Max-Heap

BUILD-HEAP(A)

for $i = \lfloor n/2 \rfloor$ **down to** 1
    HEAPIFY(i)

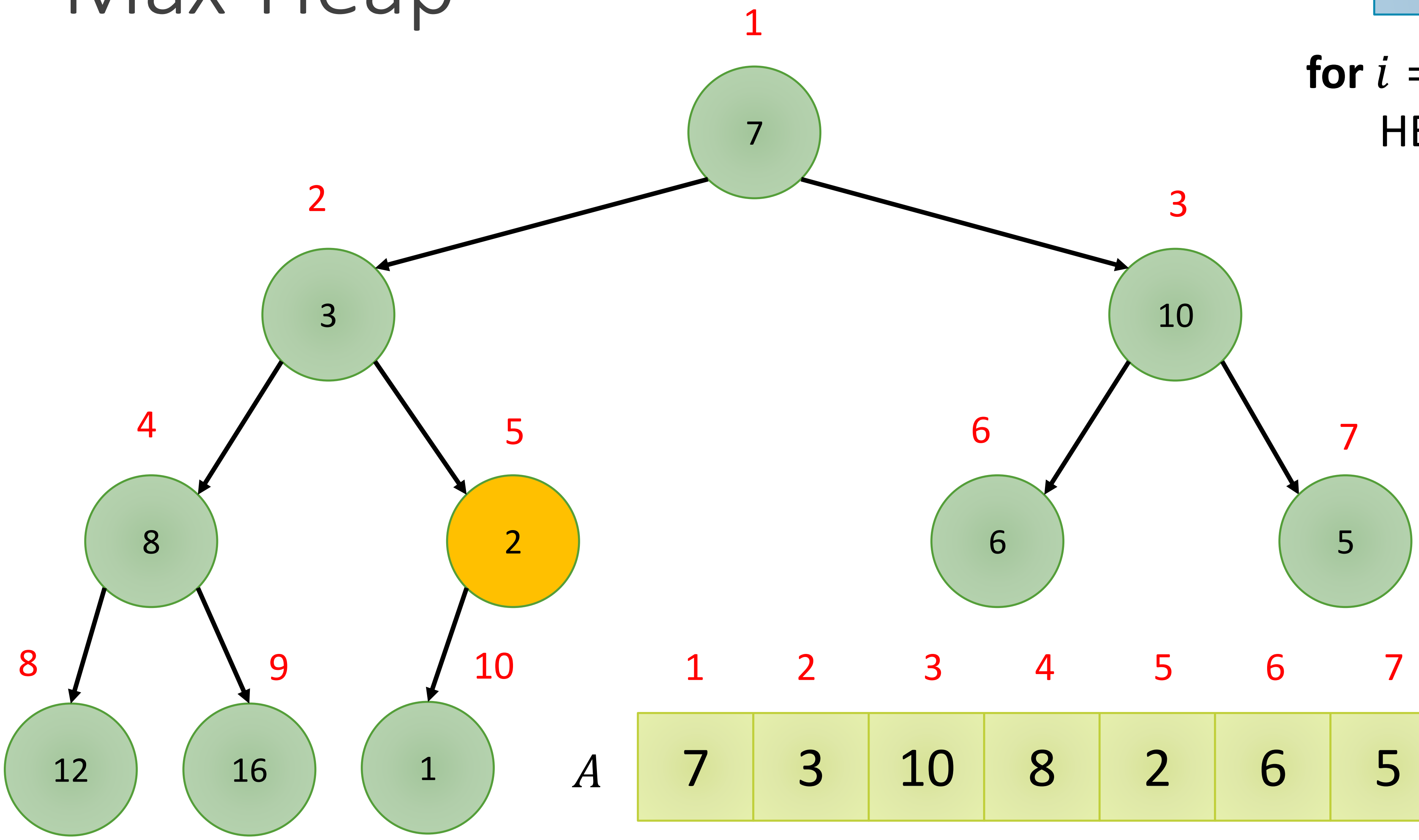


A	1	2	3	4	5	6	7	8	9	10
	7	3	10	8	2	6	5	12	16	1

Max-Heap

BUILD-HEAP(A)

for $i = \lfloor n/2 \rfloor$ **down to** 1
HEAPIFY(i)

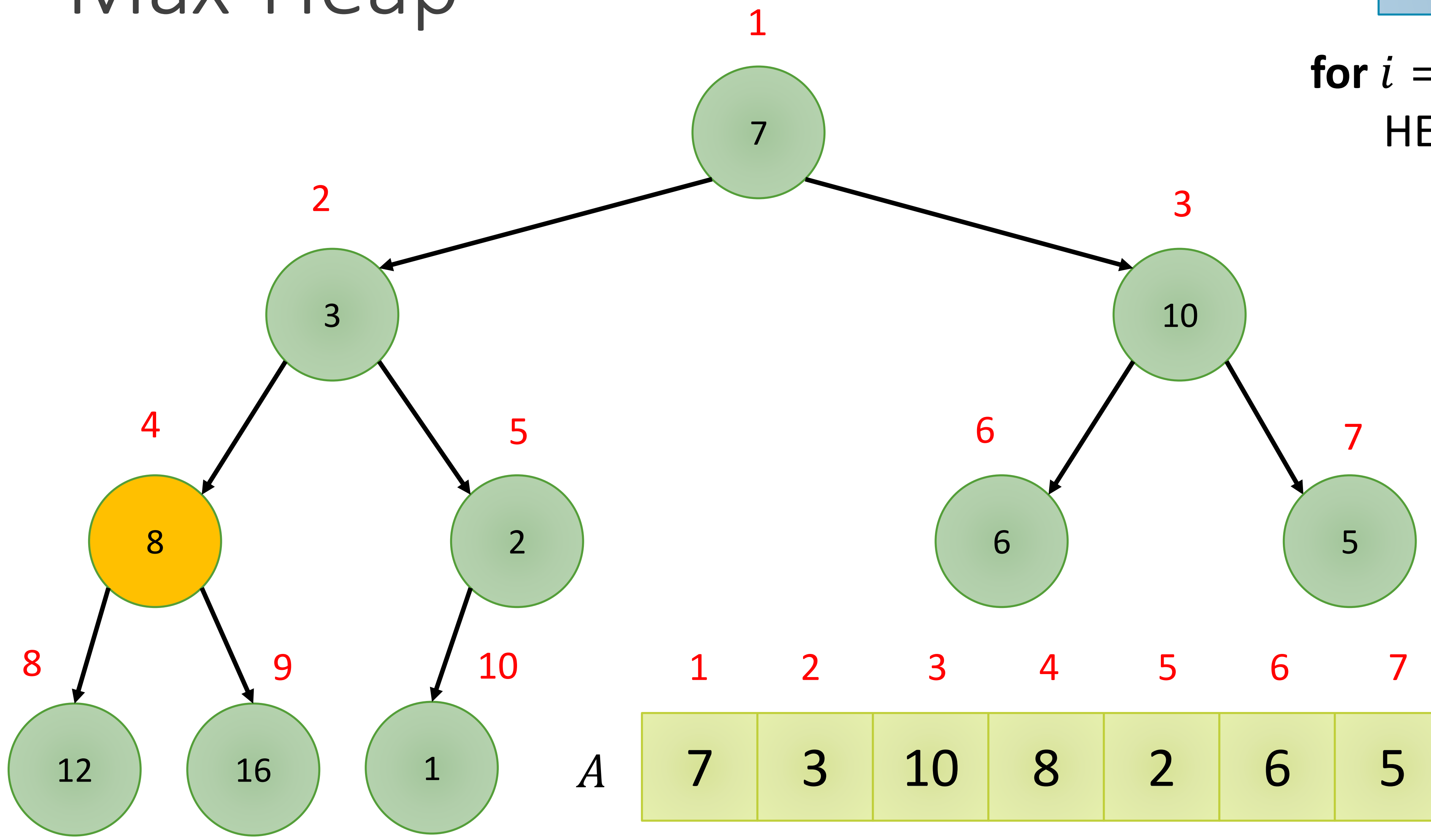


	1	2	3	4	5	6	7	8	9	10
A	7	3	10	8	2	6	5	12	16	1

Max-Heap

BUILD-HEAP(A)

for $i = \lfloor n/2 \rfloor$ **down to** 1
HEAPIFY(i)

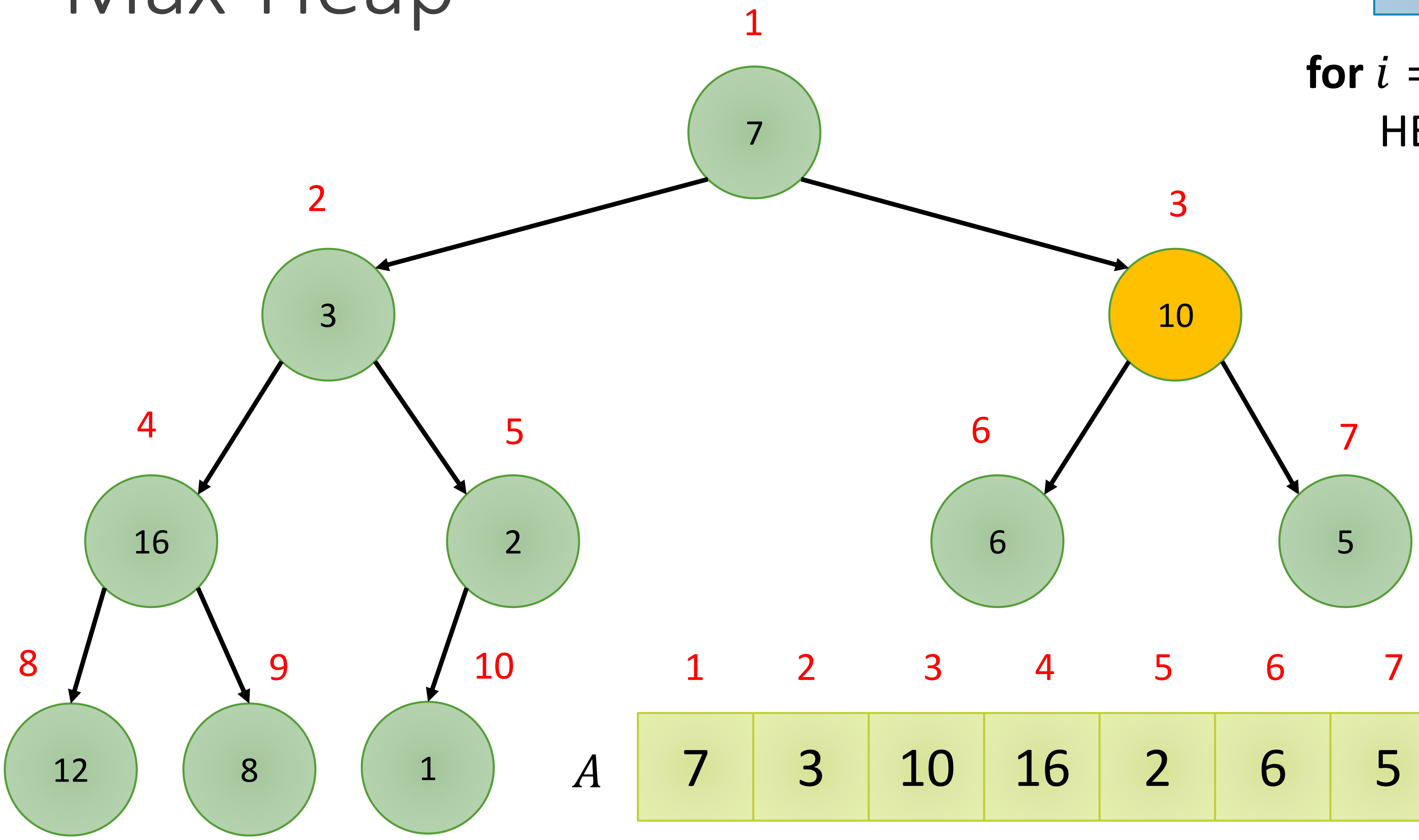


	1	2	3	4	5	6	7	8	9	10
A	7	3	10	8	2	6	5	12	16	1

Max-Heap

BUILD-HEAP(A)

for $i = \lfloor n/2 \rfloor$ **down to** 1
    HEAPIFY(i)



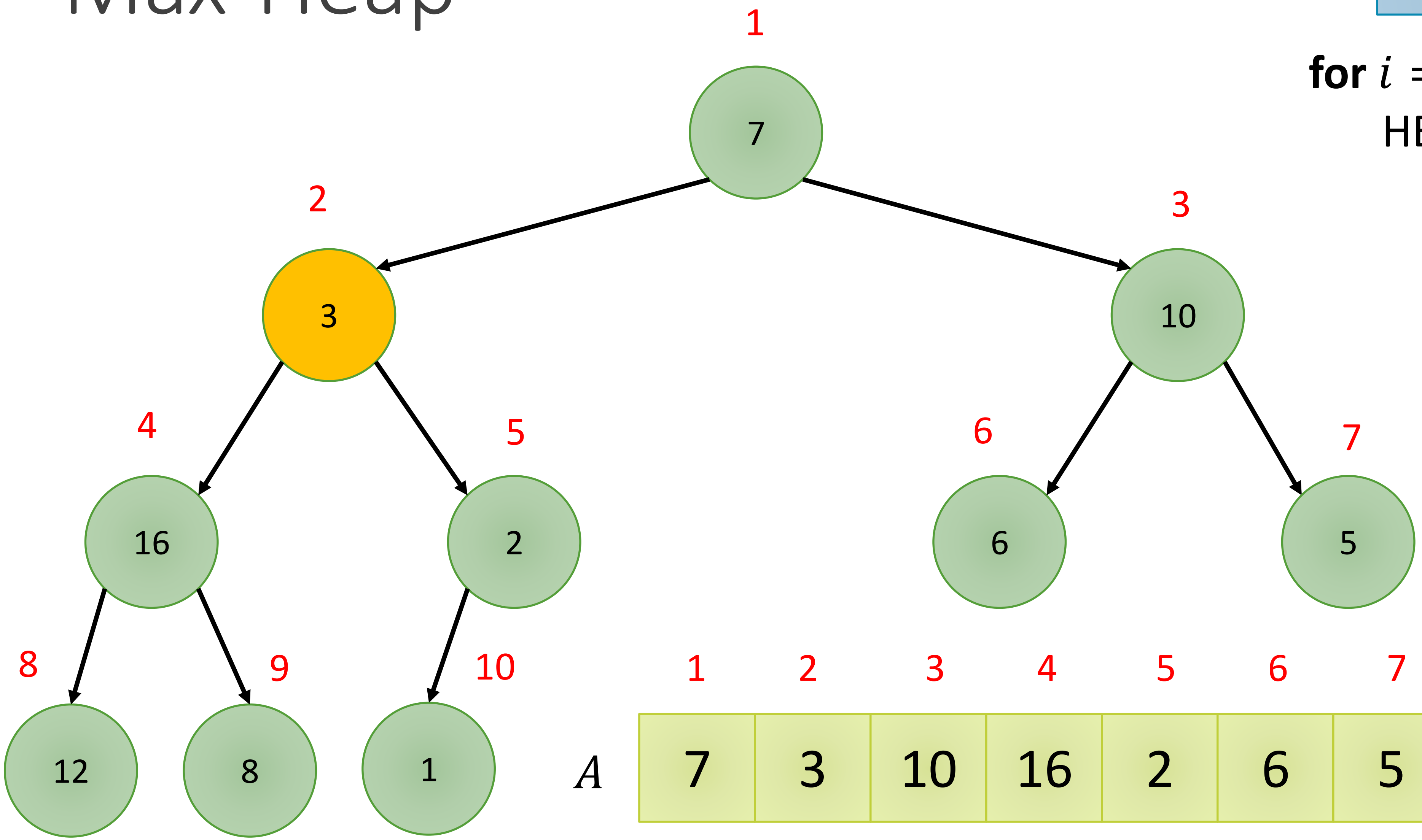
	1	2	3	4	5	6	7	8	9	10
A	7	3	10	16	2	6	5	12	8	1

Max-Heap

BUILD-HEAP(A)

for $i = \lfloor n/2 \rfloor$ **down to** 1

HEAPIFY(i)

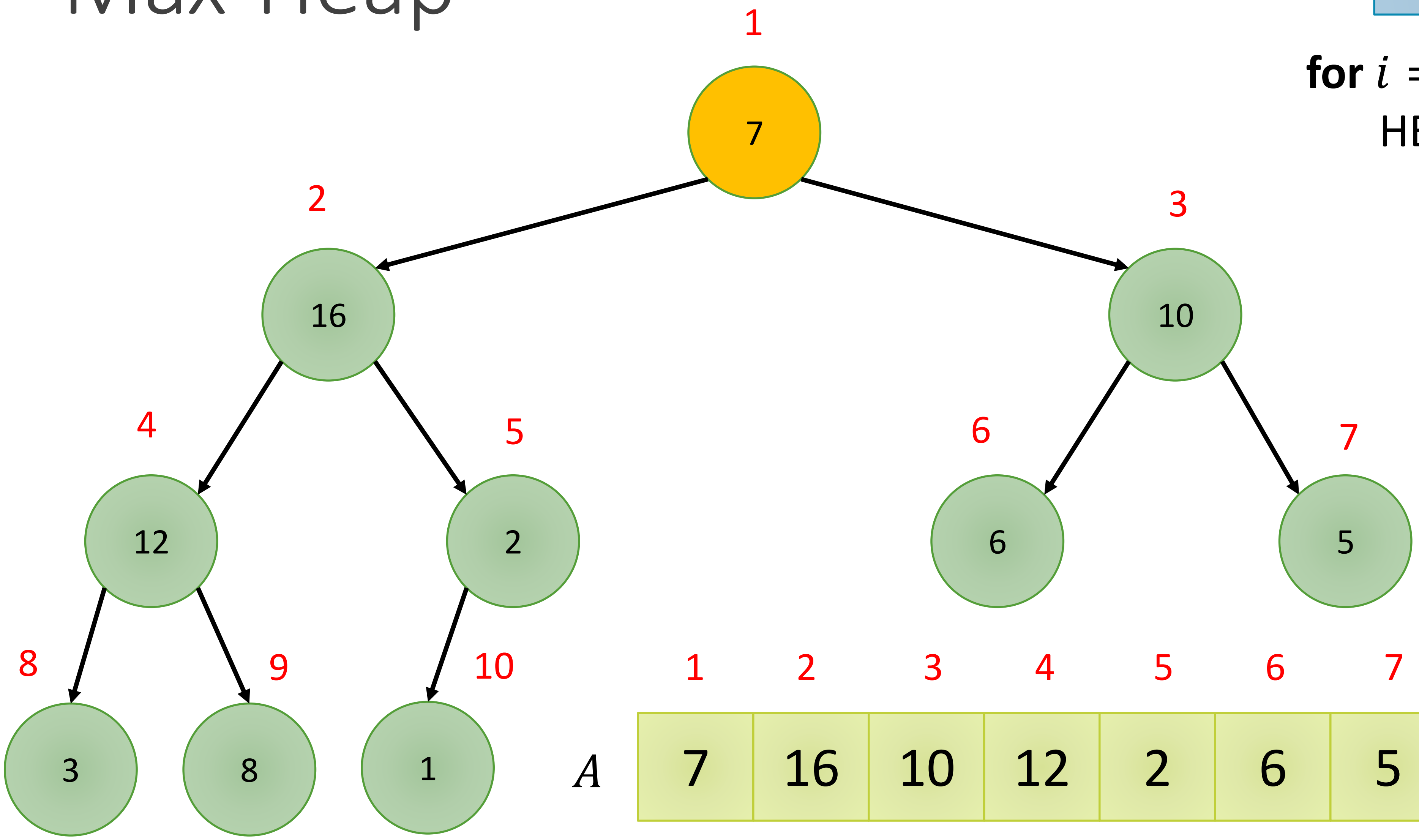


	1	2	3	4	5	6	7	8	9	10
A	7	3	10	16	2	6	5	12	8	1

Max-Heap

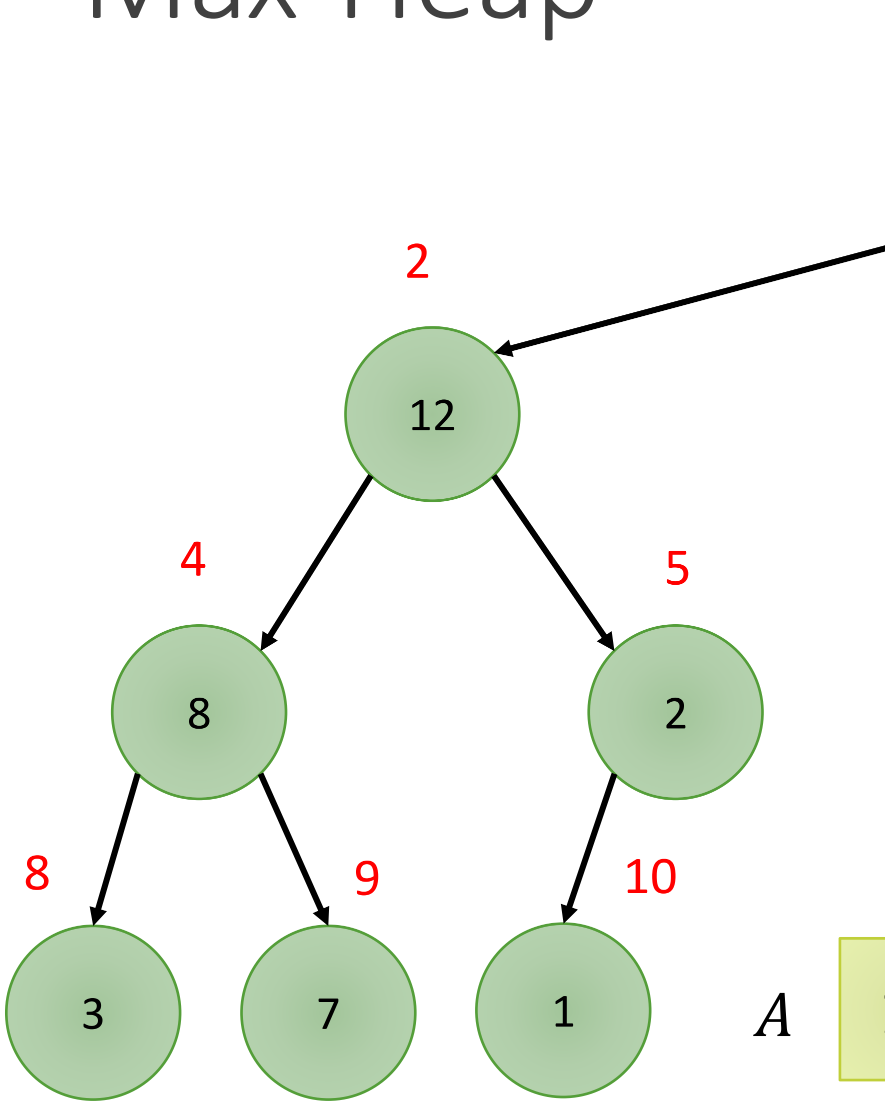
BUILD-HEAP(A)

for $i = \lfloor n/2 \rfloor$ **down to** 1
    HEAPIFY(i)



	1	2	3	4	5	6	7	8	9	10
A	7	16	10	12	2	6	5	3	8	1

Max-Heap



	1	2	3	4	5	6	7	8	9	10
A	16	12	10	8	2	6	5	3	7	1

BUILD-HEAP(A)

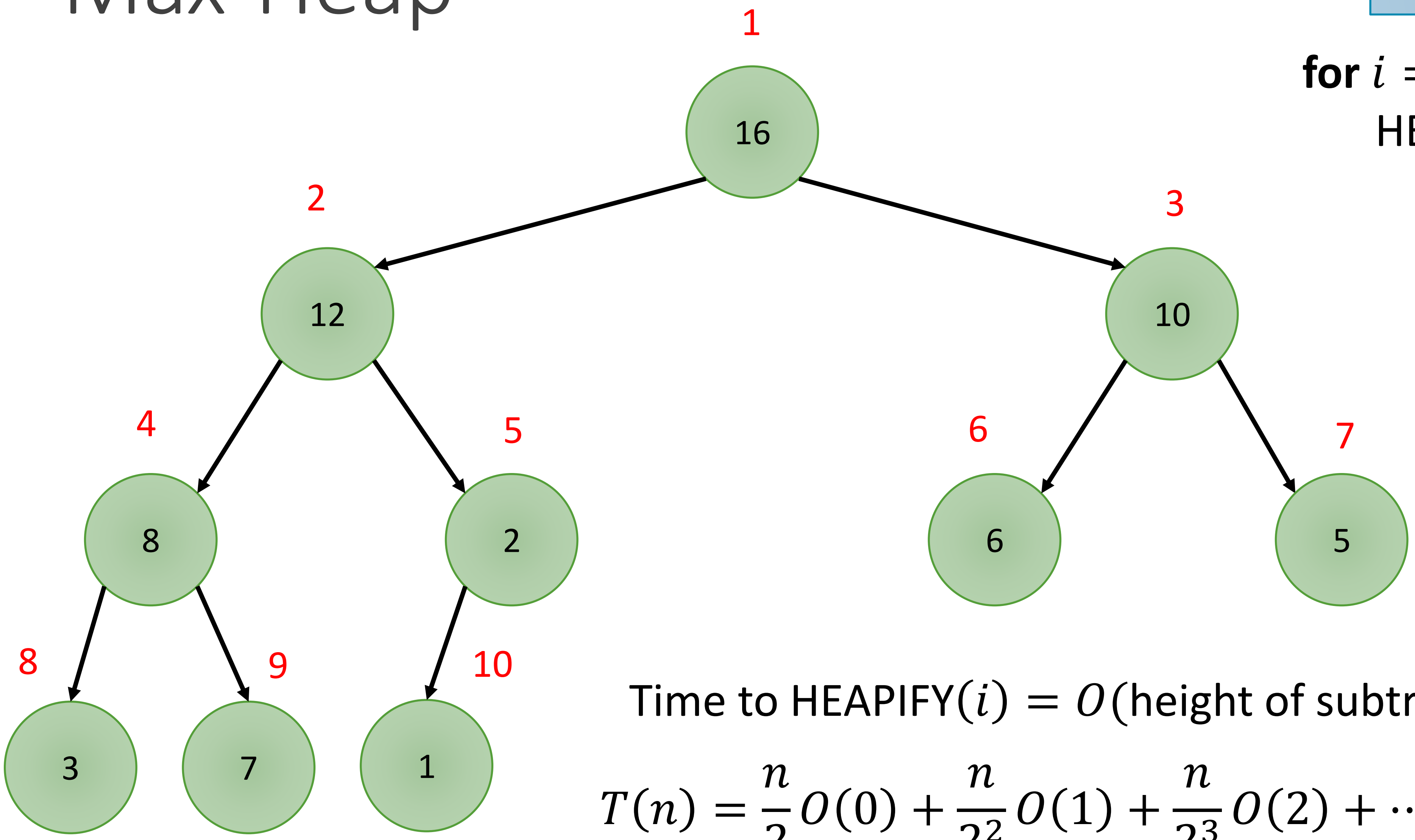
for $i = \lfloor n/2 \rfloor$ **down to** 1
    HEAPIFY(i)

Looks like $\mathrm{T}(n) = O(n \lg n)$
But we can do a better analysis

Max-Heap

BUILD-HEAP(A)

for $i = \lfloor n/2 \rfloor$ **down to** 1
  HEAPIFY(i)



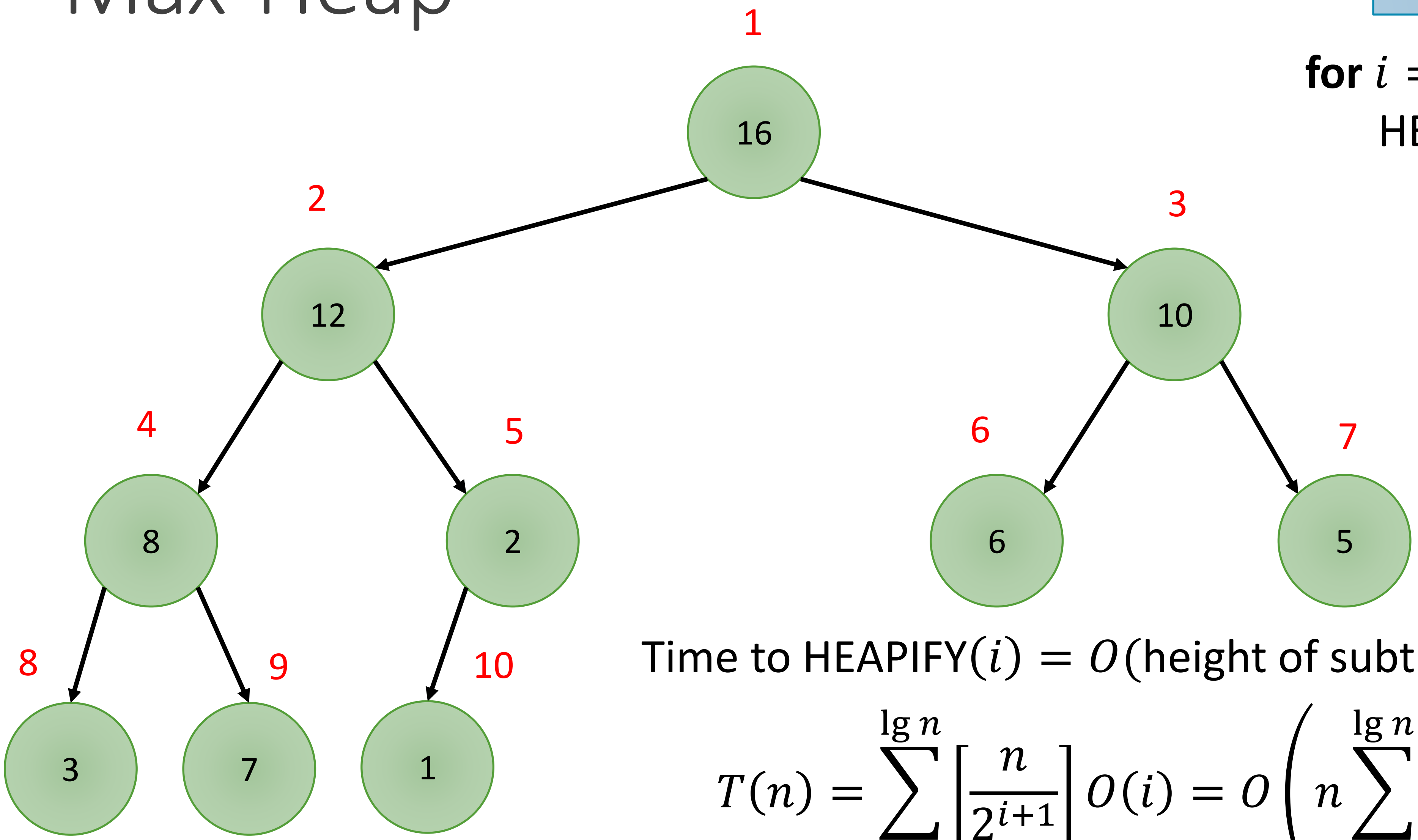
Time to HEAPIFY(i) $= O($height of subtree rooted at $i)$

$$T(n) = \frac{n}{2}O(0) + \frac{n}{2^2}O(1) + \frac{n}{2^3}O(2) + \cdots + \frac{n}{2^{\lg n+1}}O(\lg n)$$

Max-Heap

BUILD-HEAP(A)

for $i = \lfloor n/2 \rfloor$ **down to** 1
    HEAPIFY(i)



Time to HEAPIFY(i) $= O$(height of subtree rooted at i)

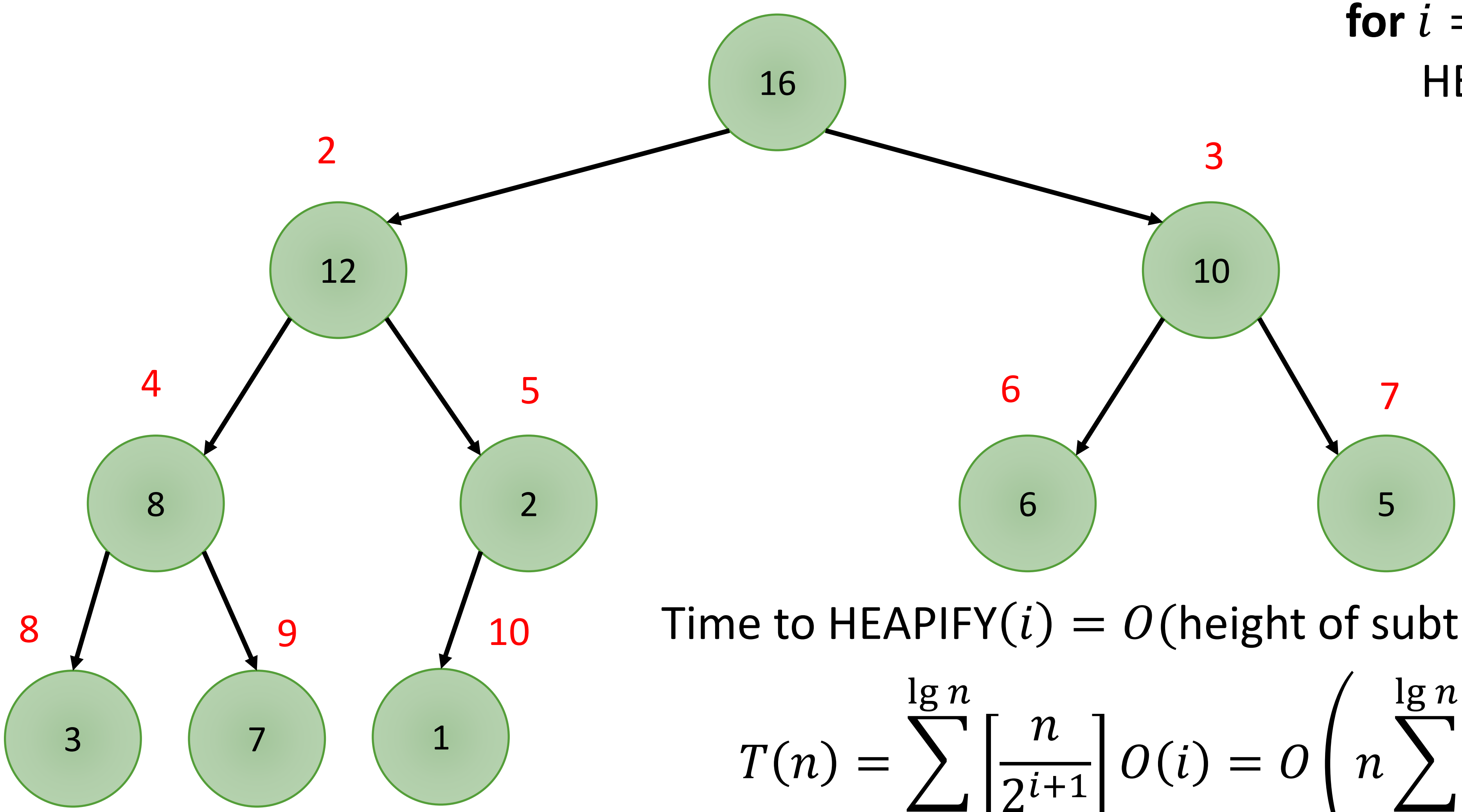
$$T(n) = \sum_{i=0}^{\lg n} \left\lceil \frac{n}{2^{i+1}} \right\rceil O(i) = O\left(n \sum_{i=0}^{\lg n} \left\lceil \frac{i}{2^i} \right\rceil \right)$$

Max-Heap

$\sum_{i=1}^{\infty} kx^k \leq \frac{x}{(1-x)^2}$ for $|x| < 1$

BUILD-HEAP(A)

for $i = \lfloor n/2 \rfloor$ **down to** 1
    HEAPIFY(i)



Time to HEAPIFY(i) $= O$(height of subtree rooted at i)

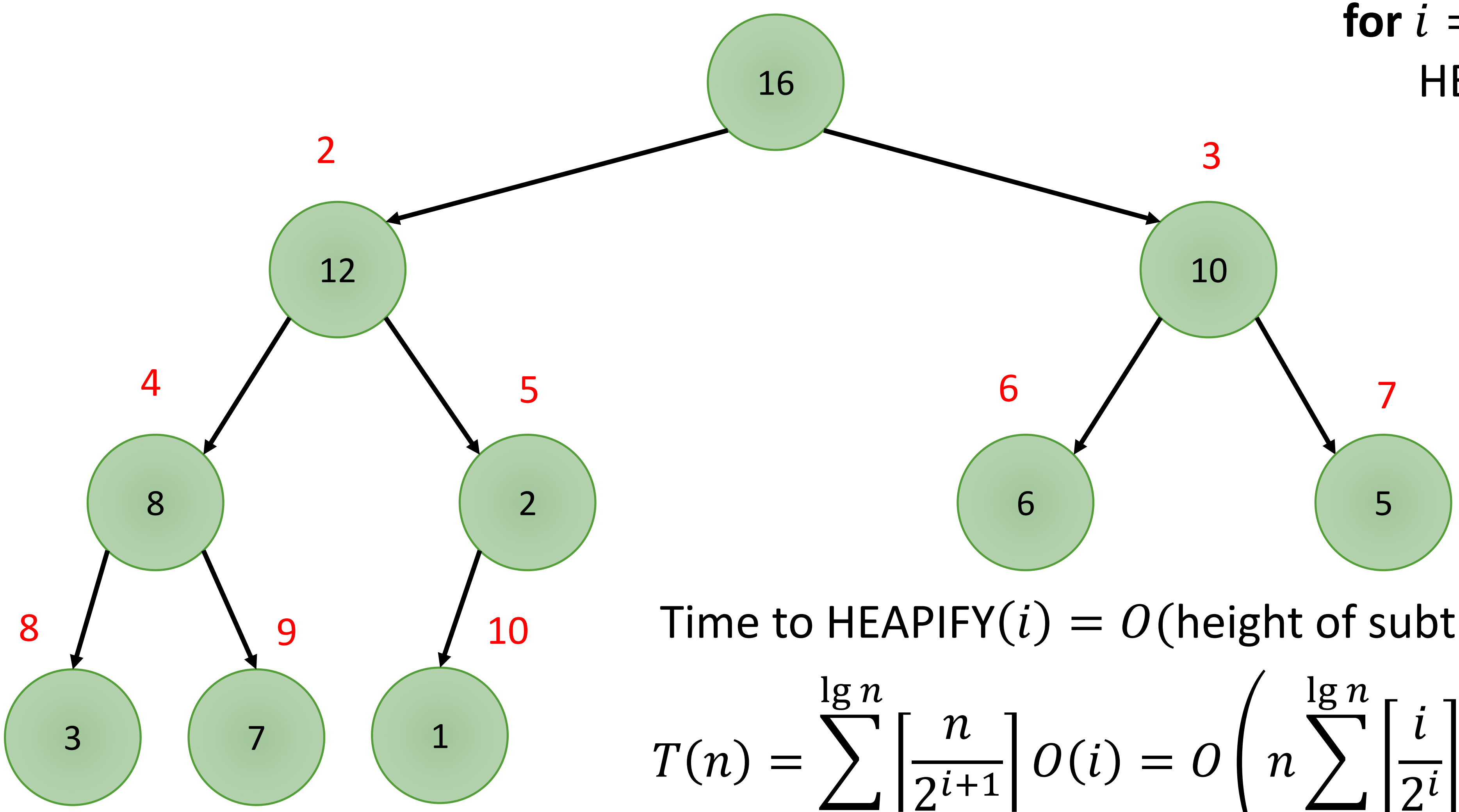
$$T(n) = \sum_{i=0}^{\lg n} \left\lceil \frac{n}{2^{i+1}} \right\rceil O(i) = O\left(n \sum_{i=0}^{\lg n} \left\lceil \frac{i}{2^i} \right\rceil \right)$$

Max-Heap

$$\sum_{i=1}^{\infty} kx^k \leq \frac{x}{(1-x)^2} \text{ for } |x| < 1$$

BUILD-HEAP(A)

for $i = \lfloor n/2 \rfloor$ **down to** 1
    HEAPIFY(i)



Time to HEAPIFY$(i) = O($height of subtree rooted at $i)$

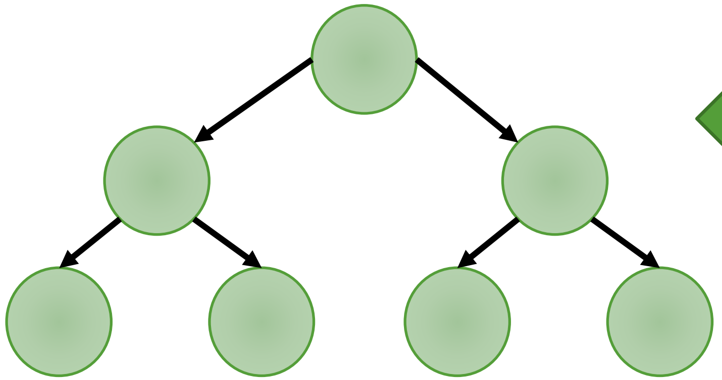
$$T(n) = \sum_{i=0}^{\lg n} \left\lceil \frac{n}{2^{i+1}} \right\rceil O(i) = O\left(n \sum_{i=0}^{\lg n} \left\lceil \frac{i}{2^i} \right\rceil\right) = O(n)$$

Heapsort

$$T(n) = \Theta(n \lg n)$$

A								
	2	8	7	1	3	5	6	4

BUILD-HEAP(A)
$O(n)$



n HEAP-getmax
$\Theta(n \lg n)$

1	2	3	4	5	6	7	8

Reverse Order
$\Theta(n)$