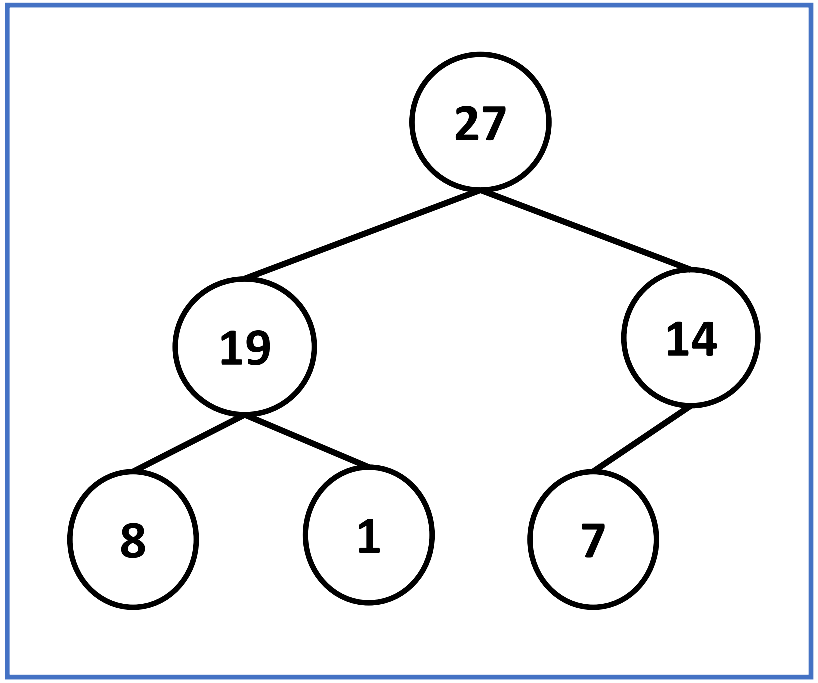
Heaps and Heapsort

Heap – Definition

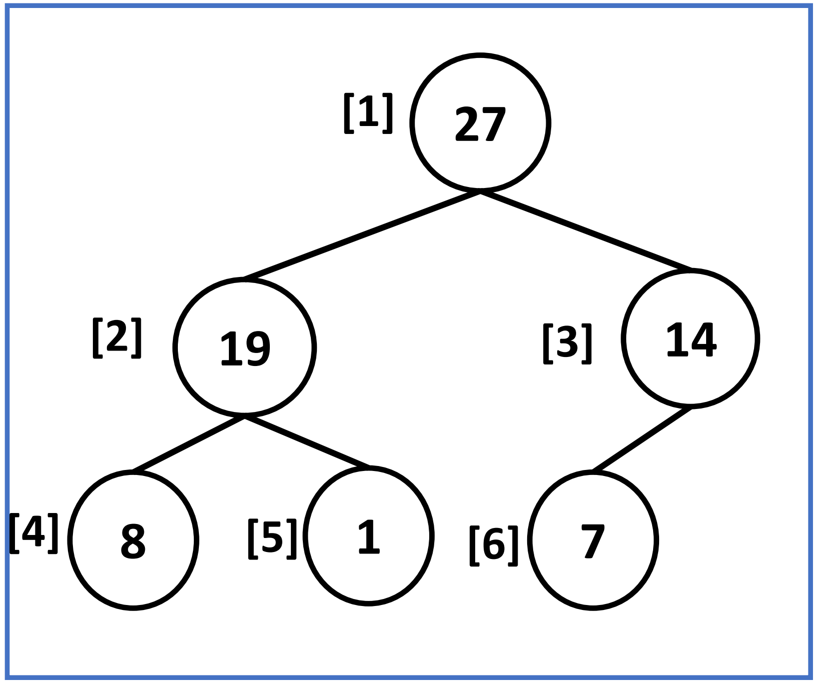
- Almost complete binary tree - completely filled, except possibly the last level (leaves)
  - Filled from left to right
- Satisfy **heap property** (**max-heap** or **min-heap**)
- **Max-heap**: Data (parent node P) ≥ Data (child node C)
- **Min-heap**: Data (parent node P) ≤ Data (child node C)
- Static or dynamic implementation
- Efficient data structure to implement Priority Queues



Heap: Static Representation

Array Representation

- **length [A]:** Size of the array A
- For an index node i
  - **parent (i):** return $\lfloor i/2 \rfloor$
  - **left (i):** return (2*i),
  - **right (i):** return (2*i + 1)
- **Max-heap property:** Value (node) >= Value (child)



	[1]	[2]	[3]	[4]	[5]	[6]
A	27	19	14	8	1	7

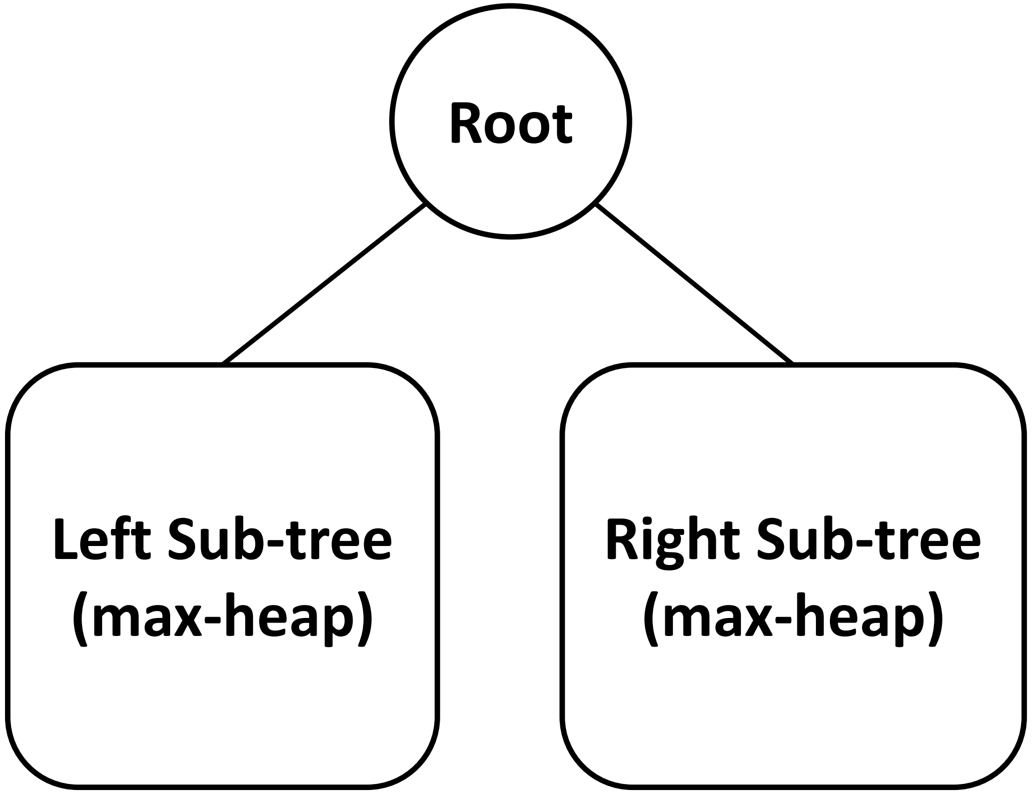
Basic Heap Operations

1. MAX-HEAPIFY procedure
2. BUILD-MAX-HEAP procedure
3. HEAPSORT procedure
4. MAX-HEAP-INSERT procedure
5. HEAP-EXTRACT-MAX procedure

Maintaining Heap Property: Key Idea (1/4)

MAX-HEAPIFY:

1. **Assume** binary trees rooted at ***LEFT(i)*** and **RIGHT(i)** are max-heaps.
2. But, **A(i) might be smaller than its children**, thus violating the max-heap property.
3. **Solution**: Find the largest between root, left-child and right-child.
4. If root is not the largest, swap root with the largest (left or right child)
5. Do 3~4 **recursively** with the index of the largest



Maintaining Heap Property (1/4)

1. Find the largest between root, left-child and right-child.
2. If root is not the largest, swap root with the largest (left or right child)

MAX-HEAPIFY(A, i)

$\ell \leftarrow$ *left (i)*

$r \leftarrow$ *right (i)*

if $\ell \leq$ *heapsize of A and* $A[\ell] > A[i]$

  largest $\leftarrow \ell$

else *largest* $\leftarrow i$

if $r \leq$ *heapsize of A and* $A[r] > A[largest]$

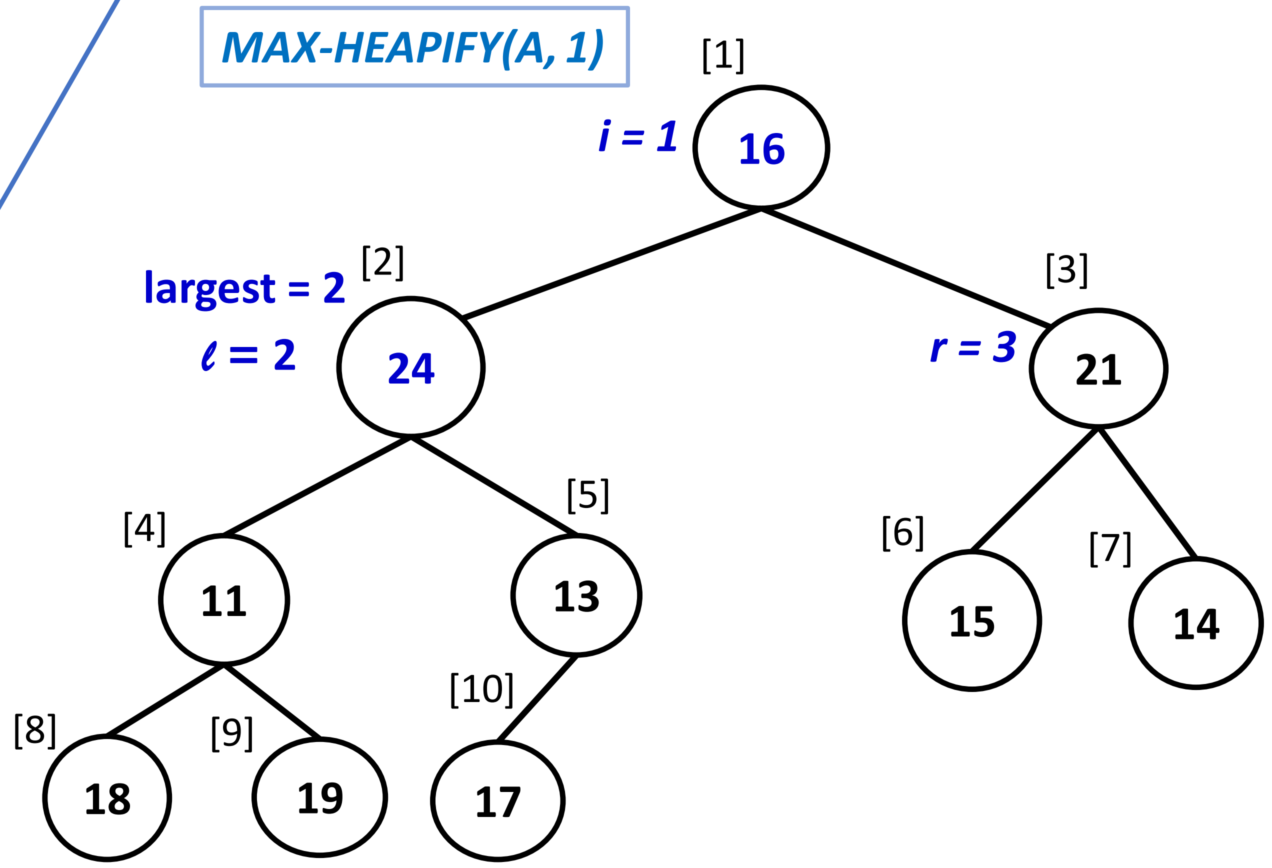
  largest $\leftarrow r$

if *largest* $\neq i$

  swap $(A[i] \leftrightarrow A[largest])$

  MAX-HEAPIFY(A, largest)

Recursive Call



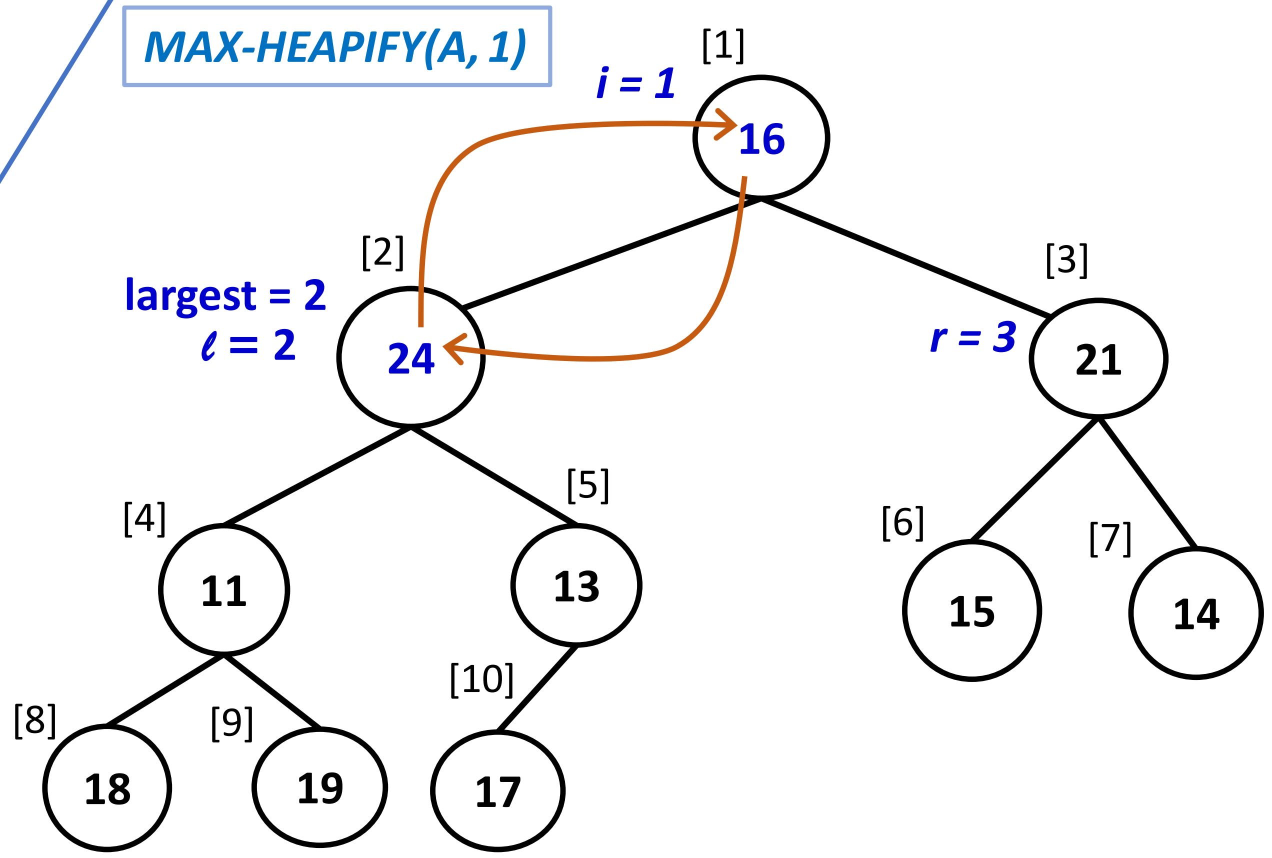
Maintaining Heap Property (2/4)

1. Find the largest between root, left-child and right-child.
2. If root is not the largest, swap root with the largest (left or right child)

MAX-HEAPIFY(A, i)

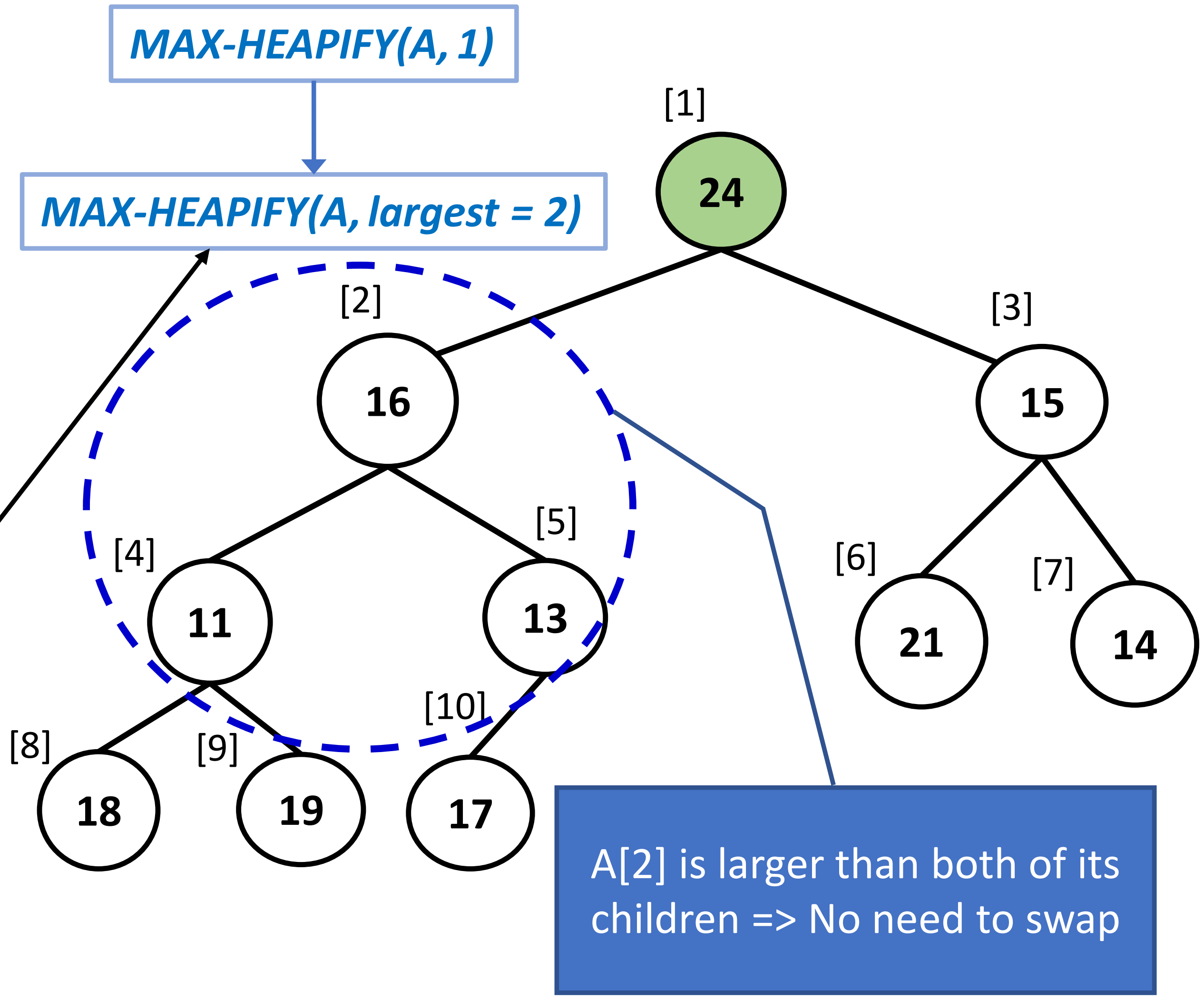
$\ell \leftarrow$ *left (i)*
$r \leftarrow$ *right (i)*
if $\ell \leq$ *heapsize of A and* $A[\ell] > A[i]$
  largest $\leftarrow \ell$
else *largest* $\leftarrow i$
if $r \leq$ *heapsize of A and* $A[r] > A[largest]$
  largest $\leftarrow r$
if *largest* $\neq i$
  swap $(A[i] \leftrightarrow A[largest])$
  MAX-HEAPIFY(A, largest)

Recursive Call



Maintaining Heap Property (3/4)

MAX-HEAPIFY(A, i)
ℓ ← *left (i)*
r ← right (i)
if *ℓ ≤ heapsize of A and A[ℓ] > A[i]*
largest ← ℓ
else
largest ← i
if *r ≤ heapsize of A and A[r] > A[largest]*
largest ← r
if *largest ≠ i*
swap (A[i] ↔ A[largest])
MAX-HEAPIFY(A, largest)

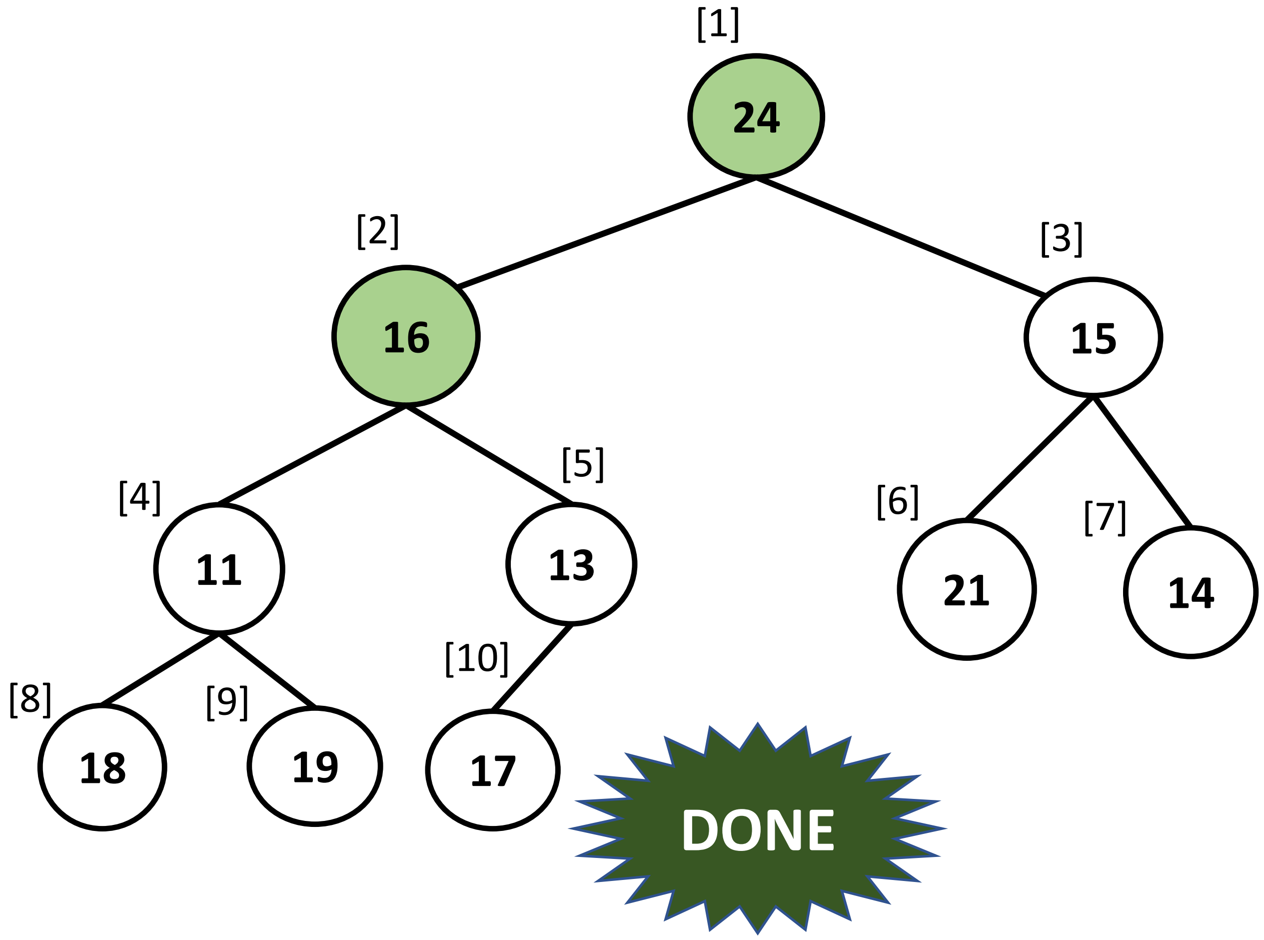


Maintaining Heap Property (4/4)

MAX-HEAPIFY(A, i)

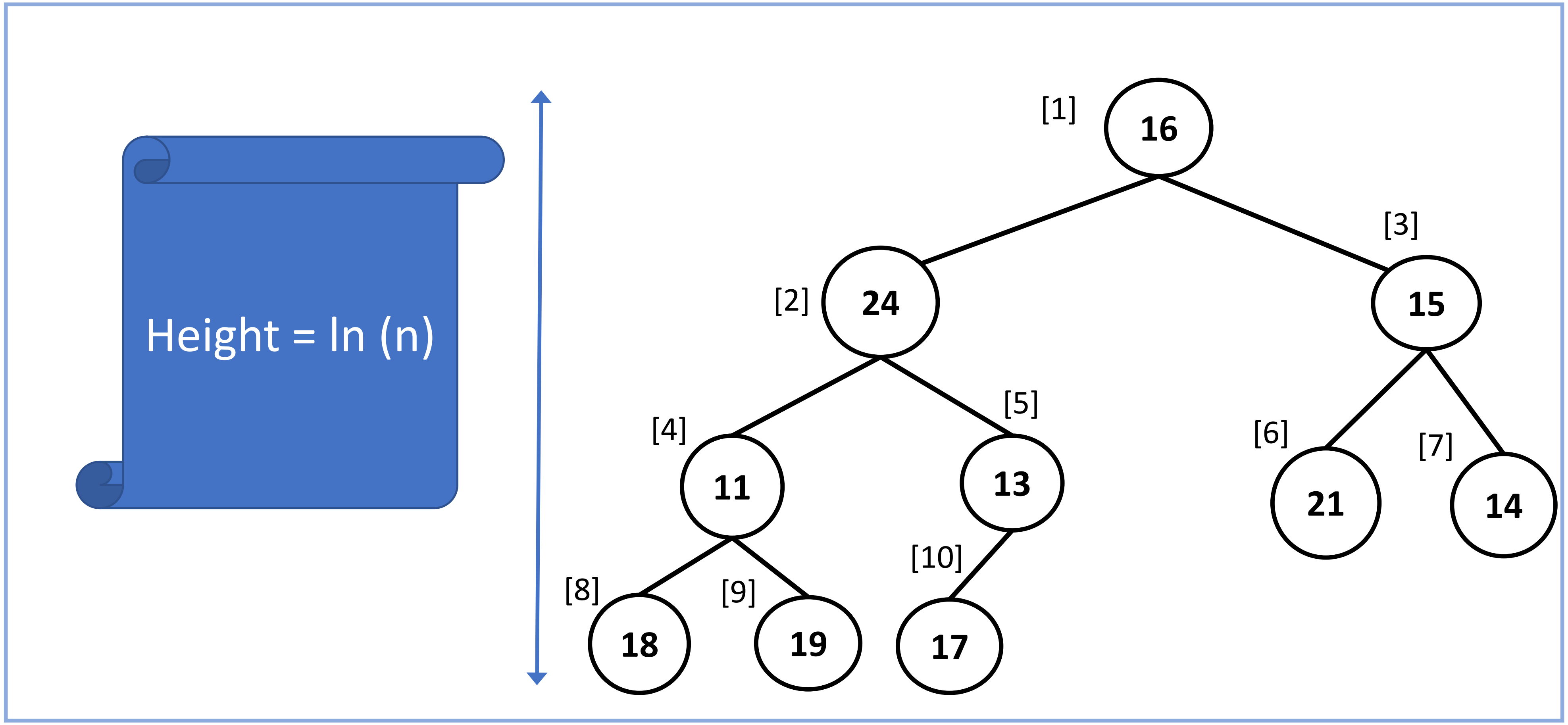
$\ell \leftarrow$ *left (i)*
$r \leftarrow$ *right (i)*
if $\ell \leq$ *heapsize of A and* $A[\ell] > A[i]$
  largest $\leftarrow \ell$
else *largest* $\leftarrow i$
if $r \leq$ *heapsize of A and* $A[r] > A[largest]$
  largest $\leftarrow r$
if *largest* $\neq i$
  swap $(A[i] \leftrightarrow A[largest])$
  MAX-HEAPIFY(A, largest)

MAX-HEAPIFY(A, 1)



Maintaining Heap: Complexity

Complexity: O(ln(n))



Building a Heap

- A Bottom-up Approach,
- From middle of array down to the first element
- With repeated calls to *MAX-HEAPIFY (A, i)*

```
BUILD_MAX-HEAP (A){
for i ← ⌊length[A]/2⌋ downto 1 do
      MAX-HEAPIFY(A, i)
}
```

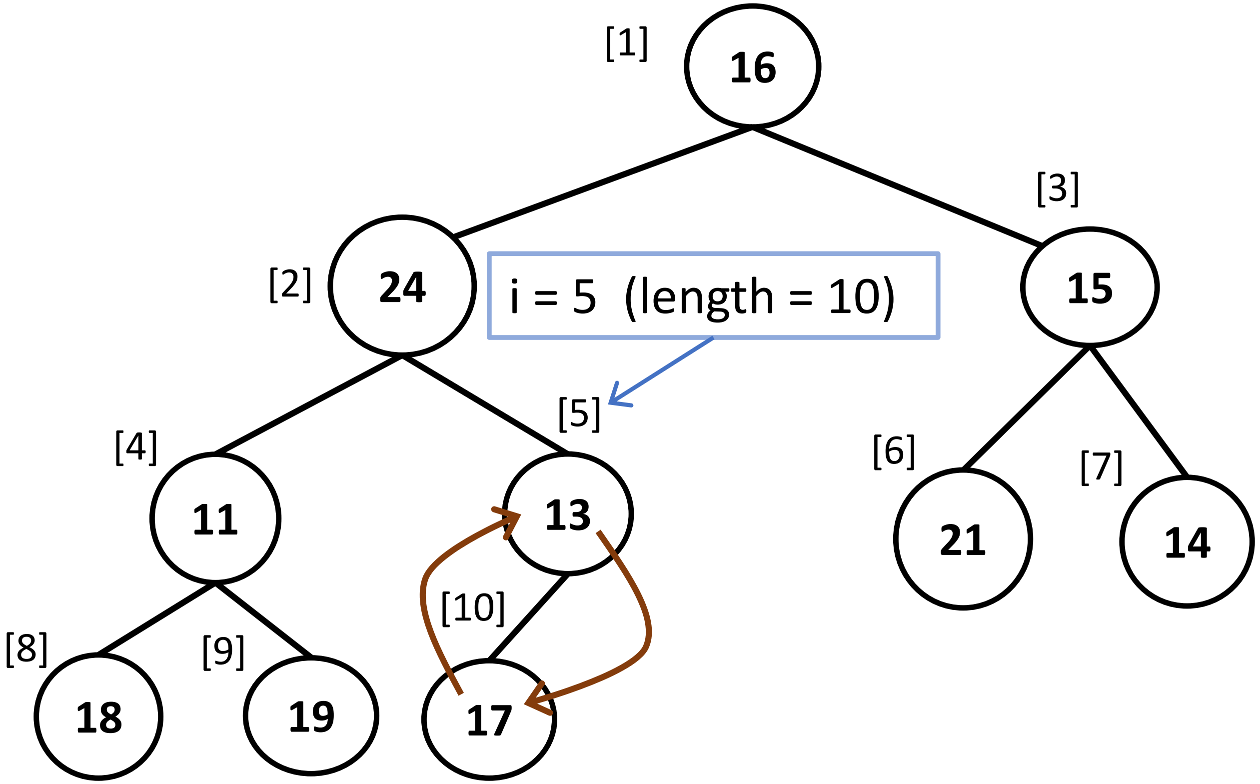
Building Heap: An Example (1/8)

BUILD_MAX-HEAP (A) {

for $i \leftarrow \lfloor length[A]/2 \rfloor$ downto 1

MAX-HEAPIFY(A, i)

}



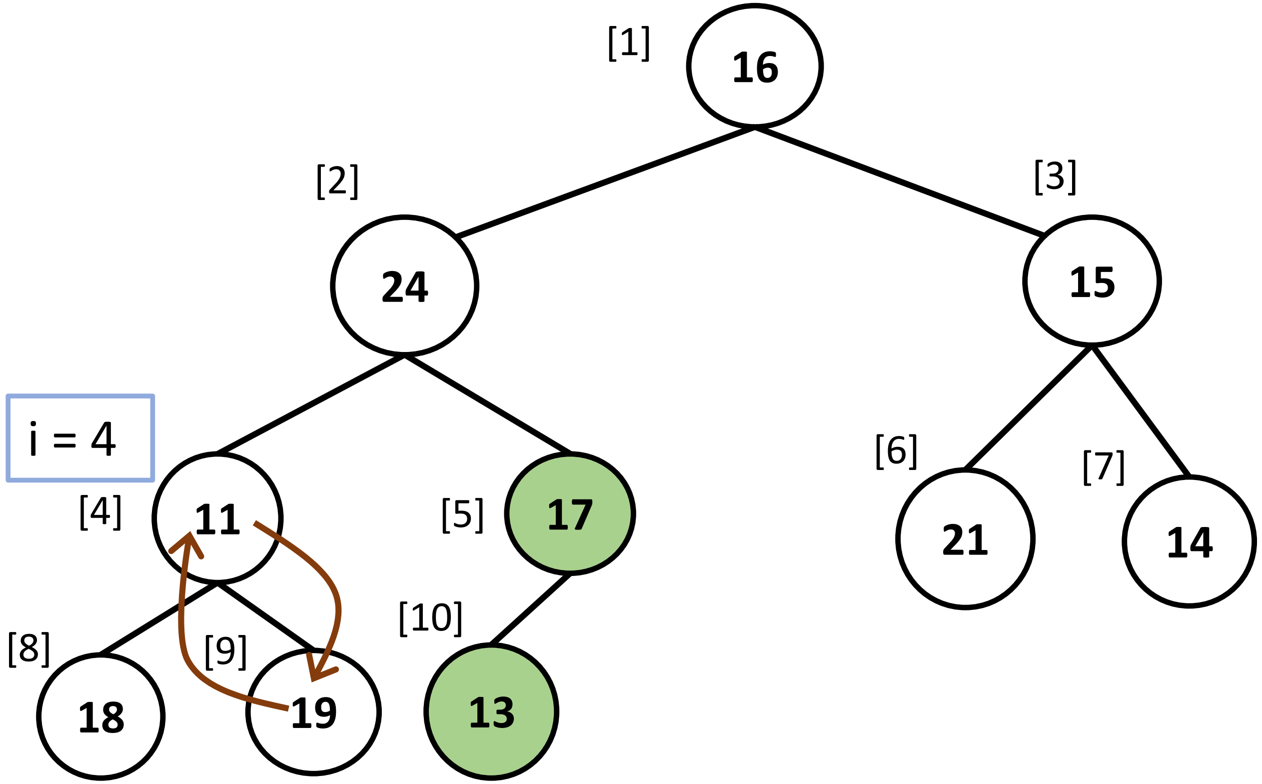
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
A	**16**	**24**	**15**	**11**	**17**	**21**	**14**	**18**	**19**	**13**

Building a Heap: An Example (2/8)

BUILD_MAX-HEAP (A){

for $i \leftarrow \lfloor length[A]/2 \rfloor$ downto 1
    MAX-HEAPIFY(A, i)

}



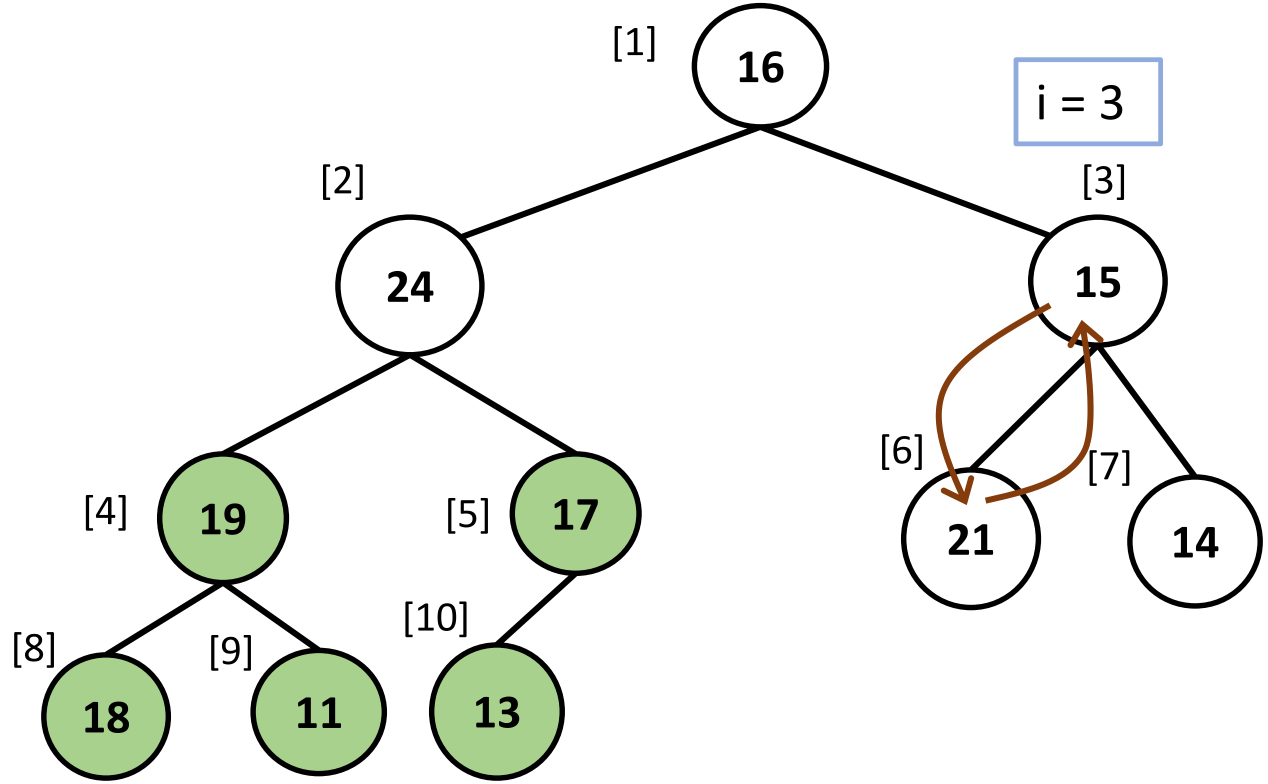
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
A	**16**	**24**	**15**	**19**	**17**	**21**	**14**	**18**	**11**	**13**

Building a Heap: An Example (3/8)

```
BUILD_MAX-HEAP (A){

for i ← ⌊length[A]/2⌋ downto 1
   MAX-HEAPIFY(A, i)

}
```



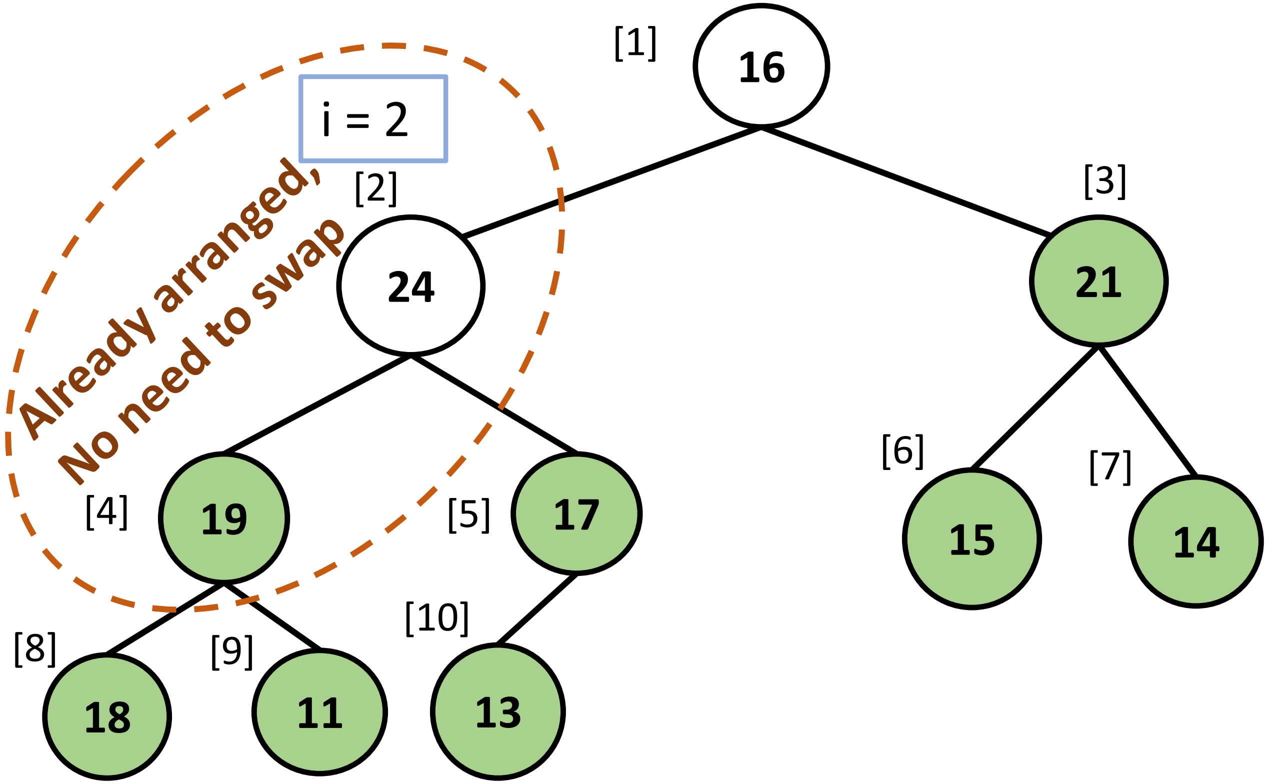
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
A	**16**	**24**	**21**	**19**	**17**	**15**	**14**	**18**	**11**	**13**

Building a Heap: An Example (4/8)

BUILD_MAX-HEAP (A){

for $i \leftarrow \lfloor length[A]/2 \rfloor$ downto 1
    MAX-HEAPIFY(A, i)

}



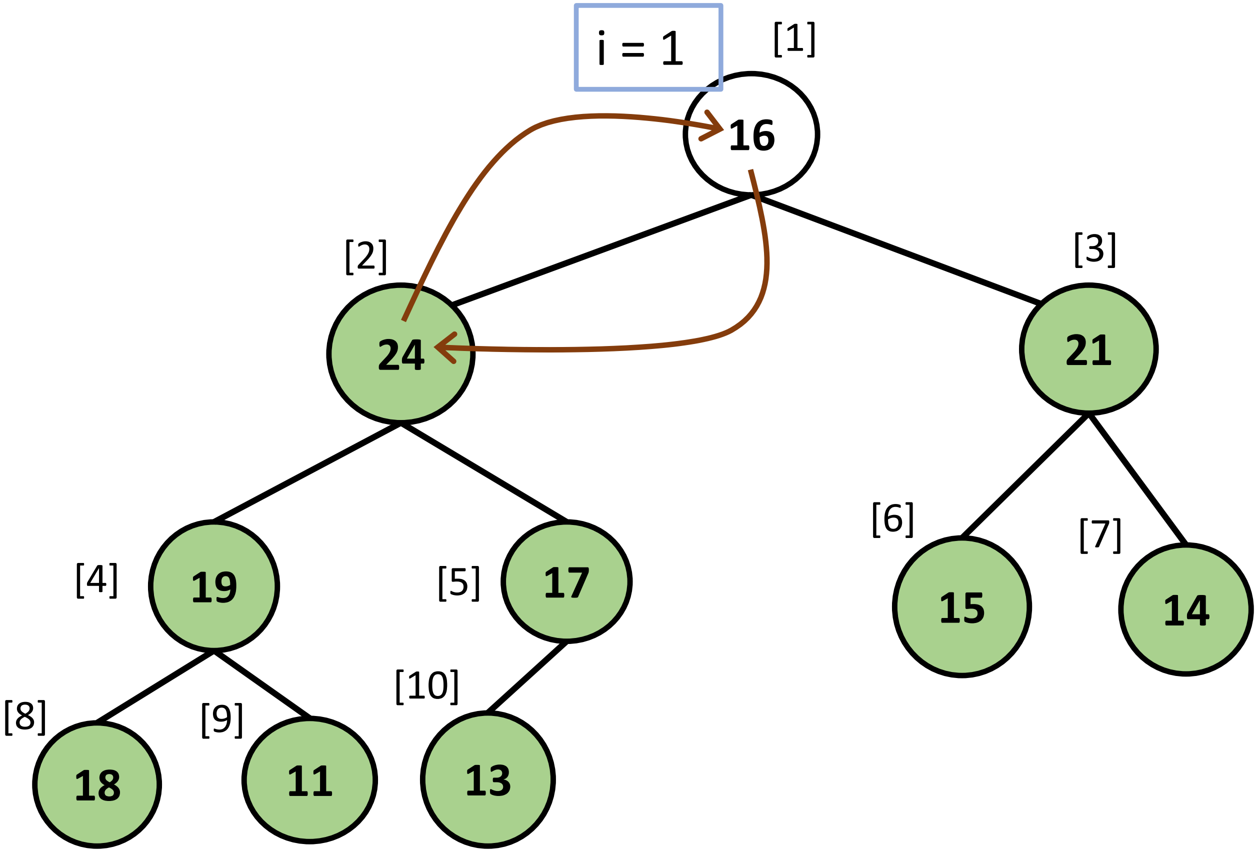
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
A	**16**	**24**	**21**	**19**	**17**	**15**	**14**	**18**	**11**	**13**

Building a Heap: An Example (5/8)

BUILD_MAX-HEAP (A){

for $i \leftarrow \lfloor length[A]/2 \rfloor$ downto 1
    MAX-HEAPIFY(A, i)

}



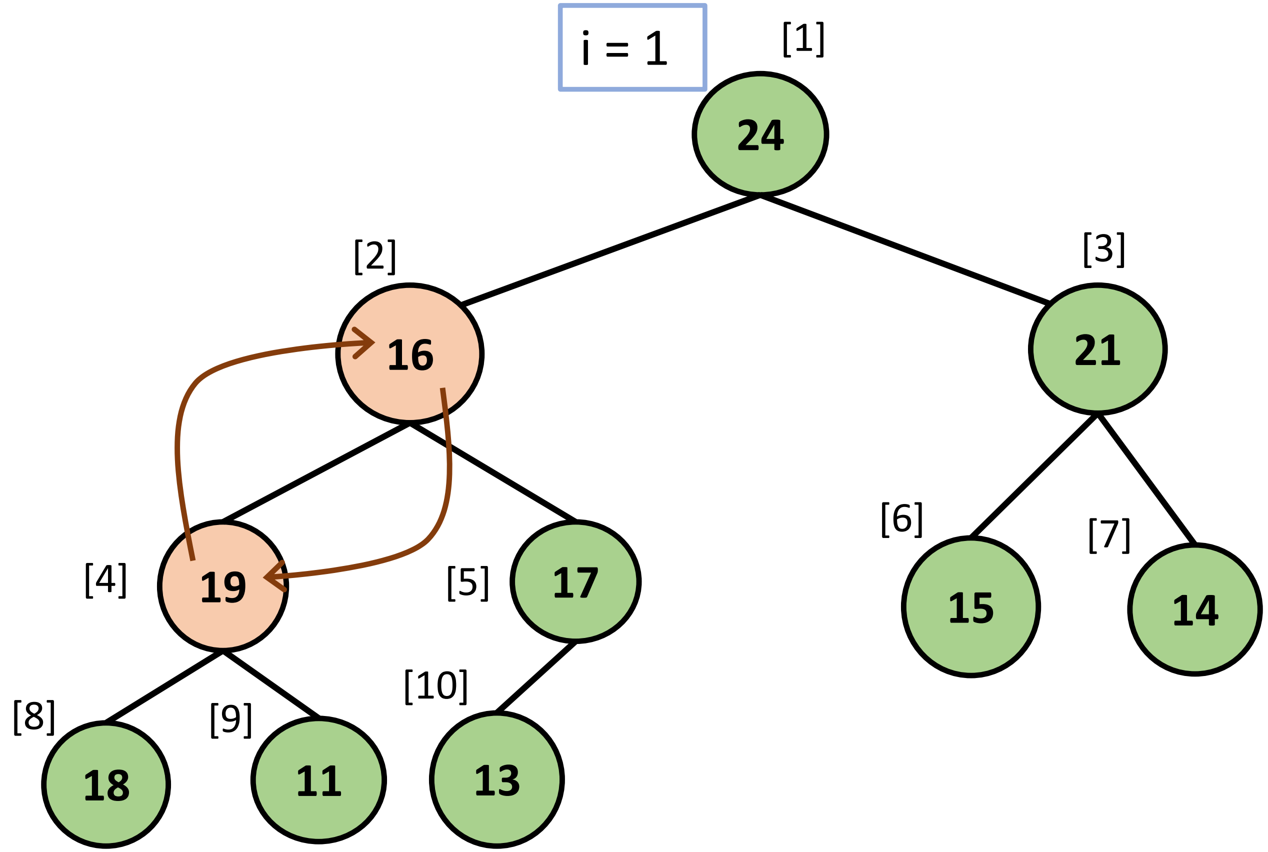
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
A	**24**	**16**	**21**	**19**	**17**	**15**	**14**	**18**	**11**	**13**

Building a Heap: An Example (6/8)

BUILD_MAX-HEAP (A){

for $i \leftarrow \lfloor length[A]/2 \rfloor$ downto 1
    MAX-HEAPIFY(A, i)

}



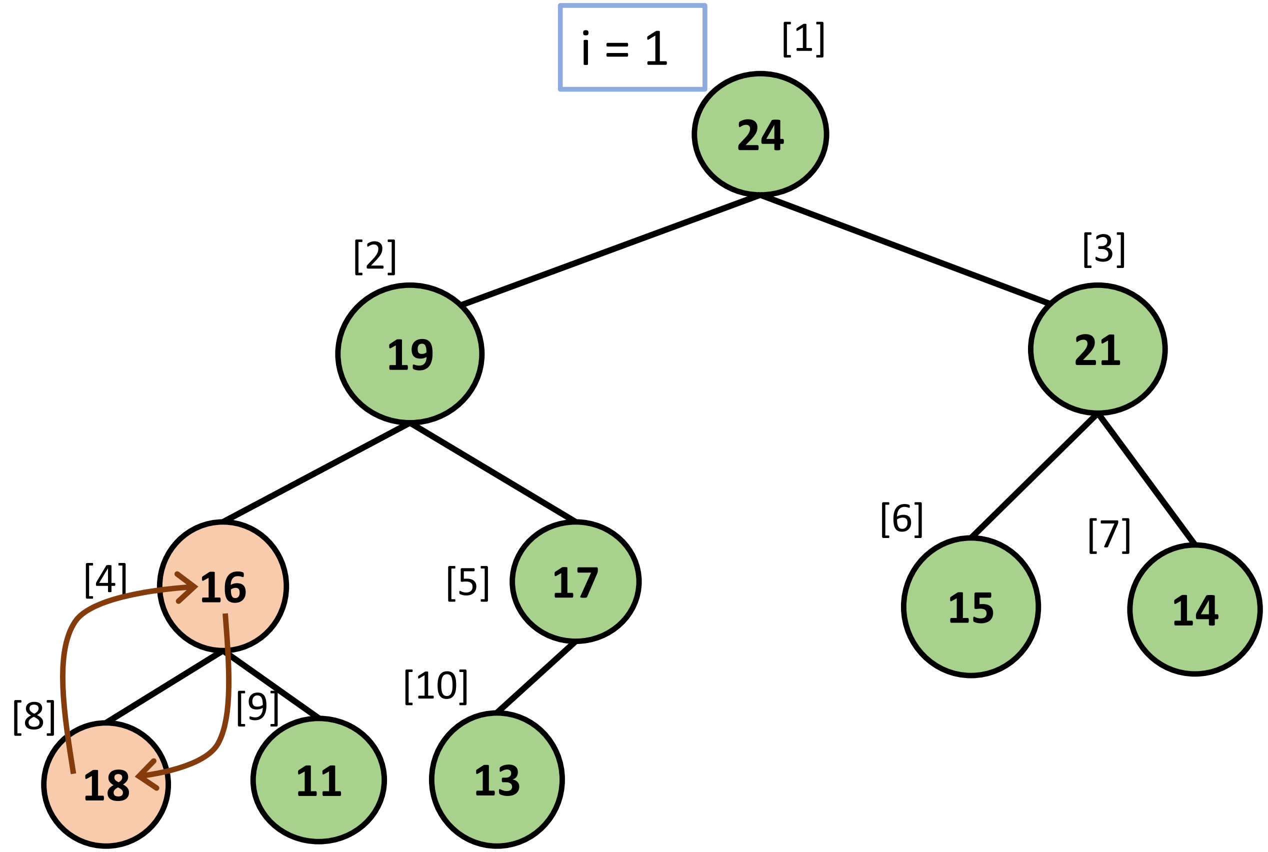
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
A	24	19	21	16	17	15	14	18	11	13

Building a Heap: An Example (7/8)

BUILD_MAX-HEAP (A){

for $i \leftarrow \lfloor length[A]/2 \rfloor$ downto 1
    MAX-HEAPIFY(A, i)

}



	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
A	**24**	**19**	**21**	**18**	**17**	**15**	**14**	**16**	**11**	**13**

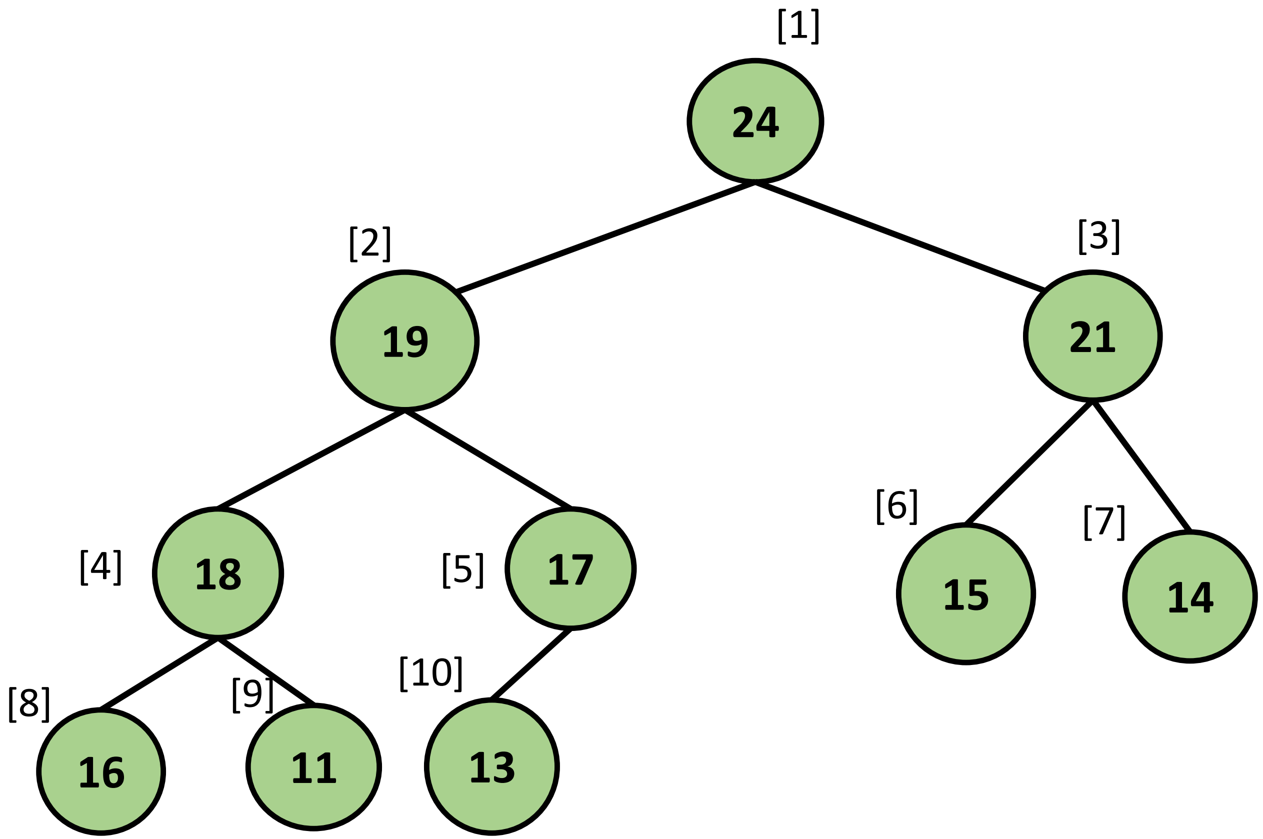
Building a Heap: An Example (8/8)

BUILD_MAX-HEAP (A){

for $i \leftarrow \lfloor length[A]/2 \rfloor$ down to 1

  MAX-HEAPIFY(A, i)

}



	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
A	24	19	21	18	17	15	14	16	11	13

DONE

Building a Heap: Complexity

BUILD_MAX-HEAP (A){

for $i \leftarrow \lfloor length[A]/2 \rfloor$ downto 1

MAX-HEAPIFY(A, i)

}

= $O(n)$

= $O(\ln(n))$

Overall Complexity: O(n ln (n))

Basic Heap Operations

1. MAX-HEAPIFY procedure
2. BUILD-MAX-HEAP procedure
3. **HEAPSORT procedure**
4. **MAX-HEAP-INSERT procedure**
5. **HEAP-EXTRACT-MAX procedure**

Thank you!