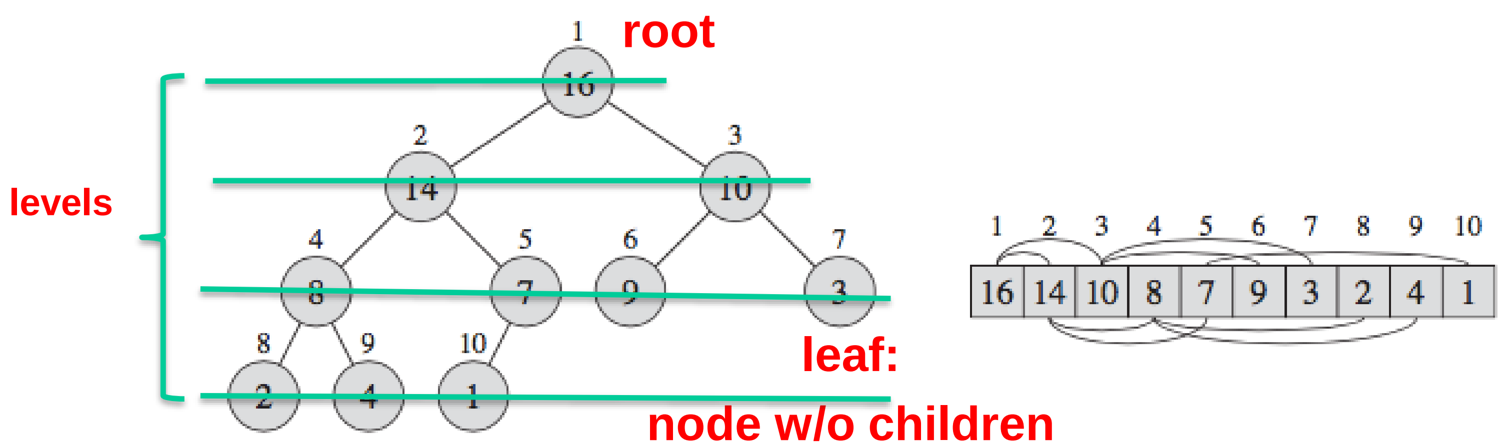
Heaps

"Teachers open the door, but you must enter by yourself. "

- *Chinese Proverb*

Binary Heaps

- A *binary heap* is a data structure that we can viewed as a mostly complete binary tree.
  - not to be confused with the *runtime heap*
    - portion of memory for dynamically allocated variables
  - all levels have maximum number of nodes, except deepest where nodes are filled in from left to right
  - implementation is usually array-based



Binary Heap

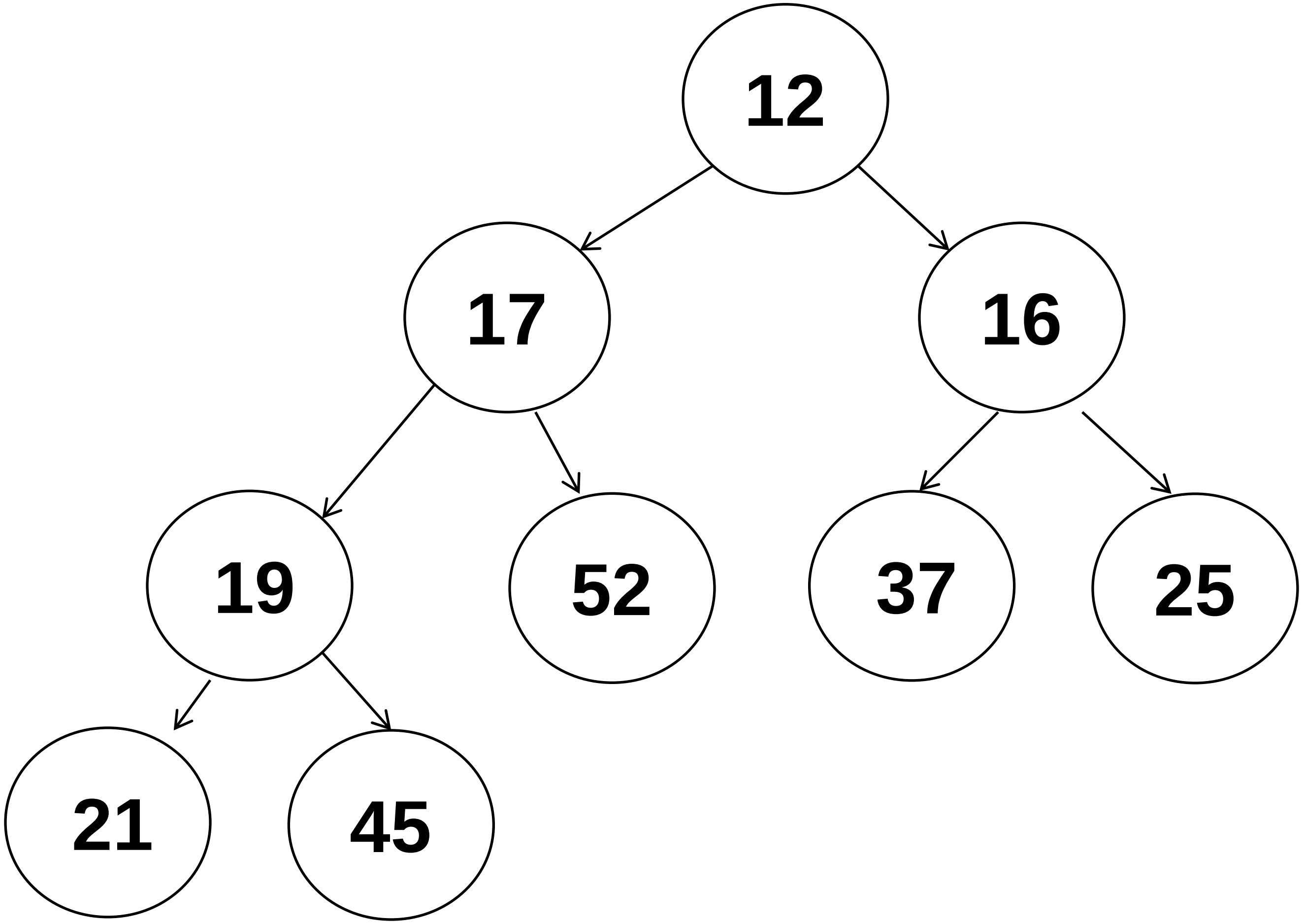
- Two kinds of binary heaps:
  - *Max-Heap* and *Min-Heap*
- Each maintains the *heap order property*
  - The Max-Heap satisfies the Max-Heap Property:

    `A[Parent[i]] ≥ A[i]`
  - The Min-Heap satisfies the Min-Heap Property:

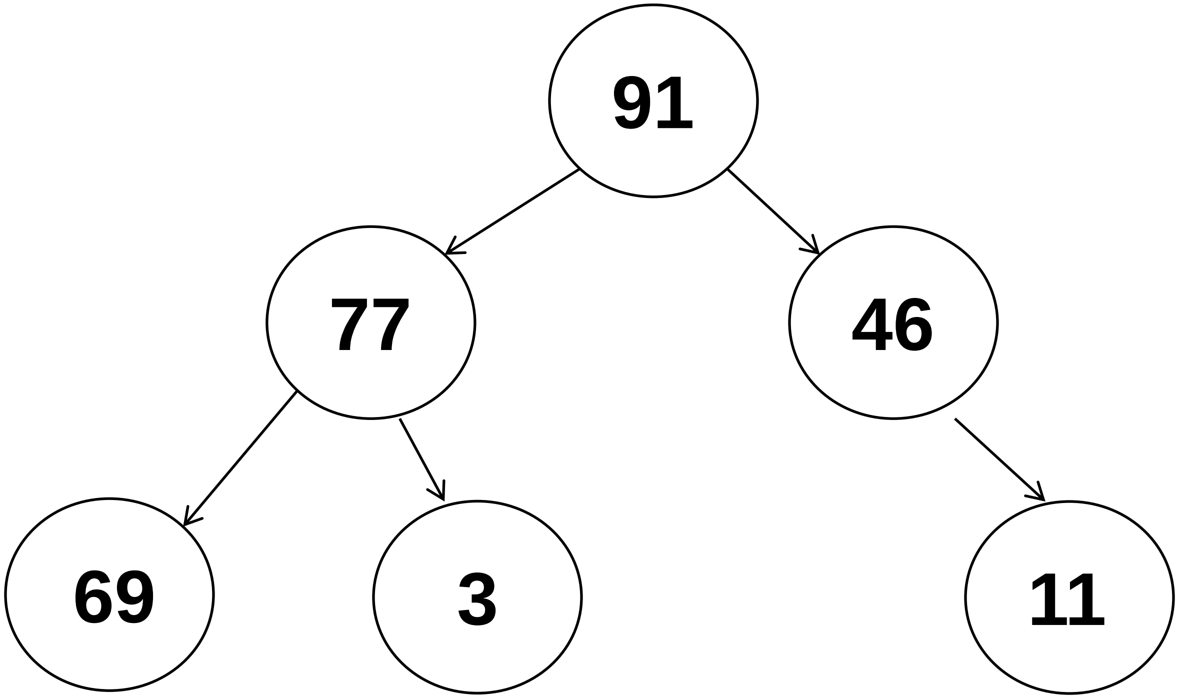
    `A[Parent[i]] ≤ A[i]`

- The Max-Heap is used in the *heapsort* algorithm
- Both types are used to implement *priority queues*

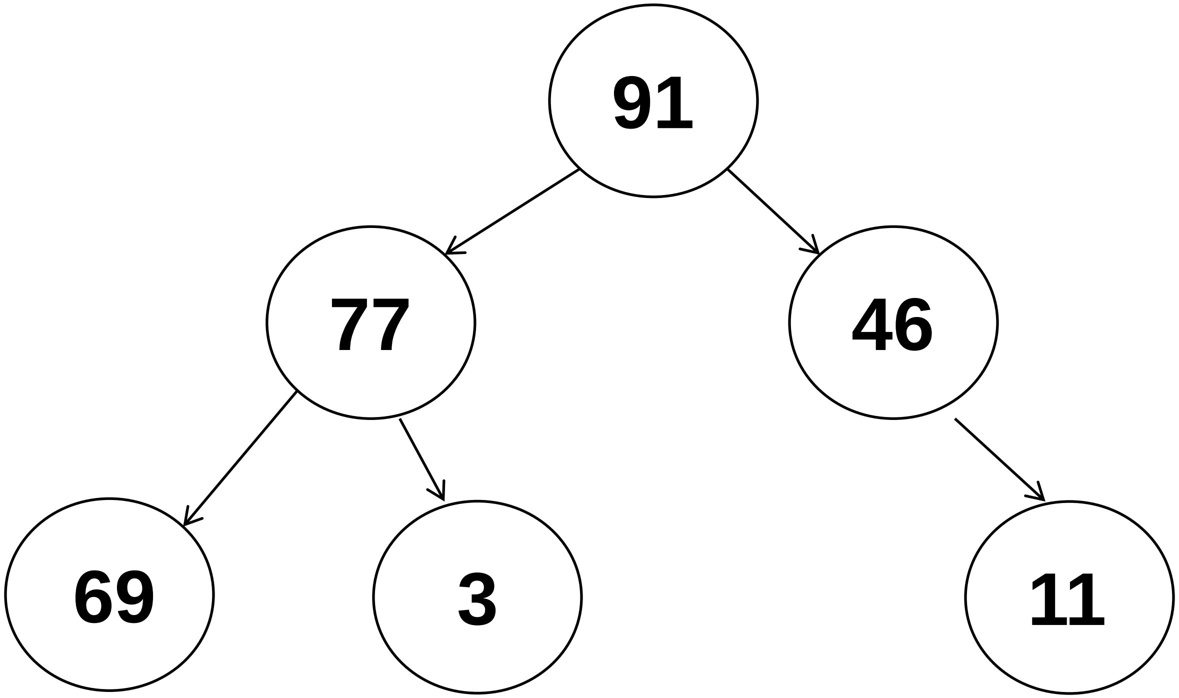
Example: Min-Heap



Is It a Max-Heap?

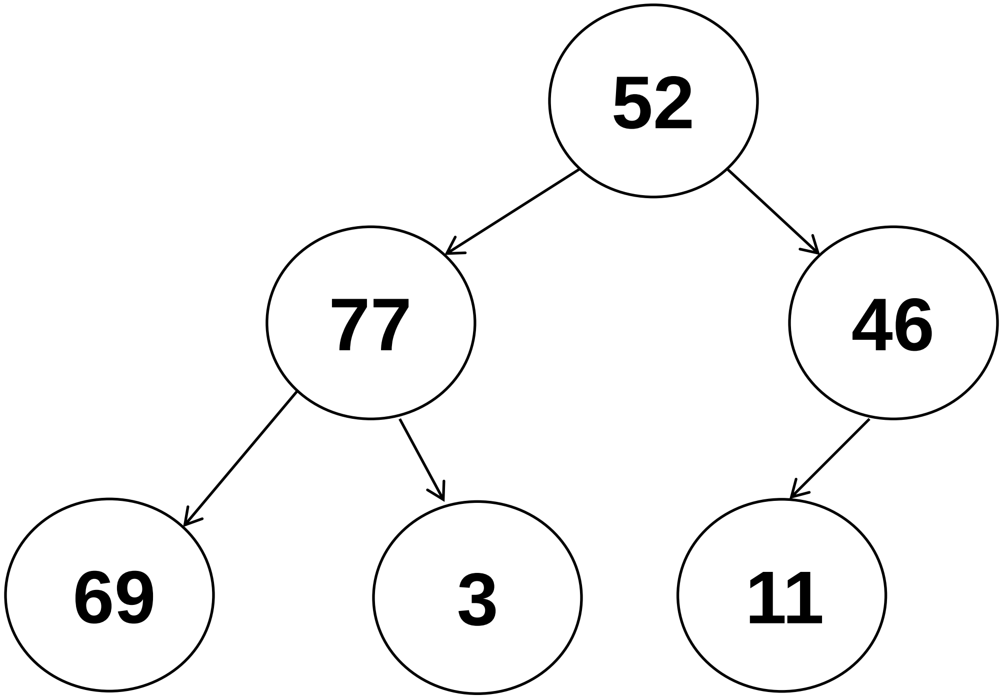


Is It a Max-Heap?

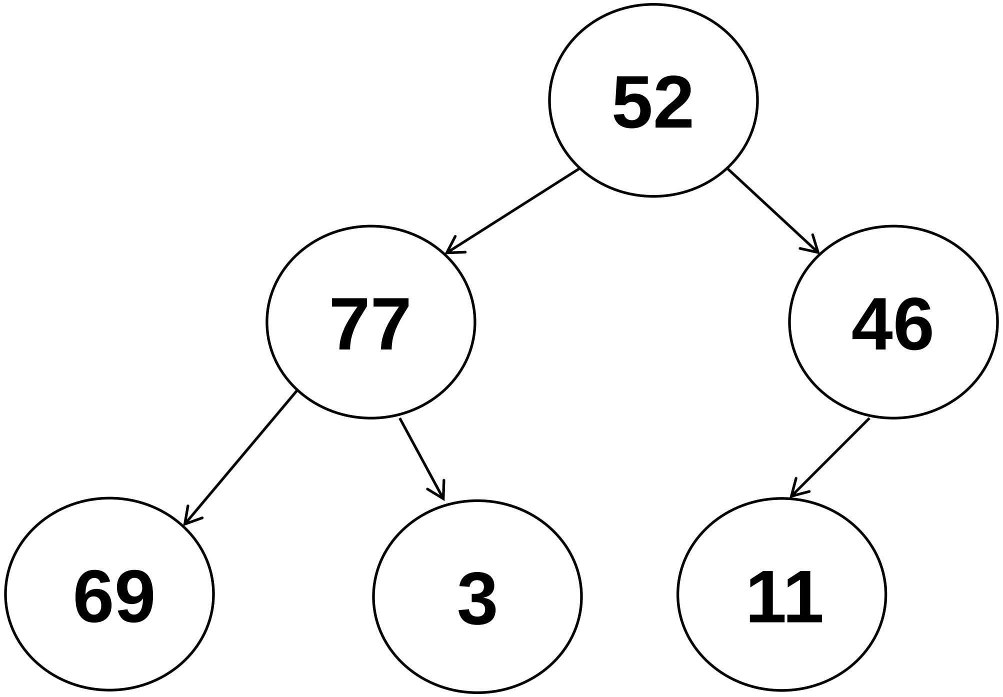


No – bottom level not filled in left to right

Is It a Max-Heap?

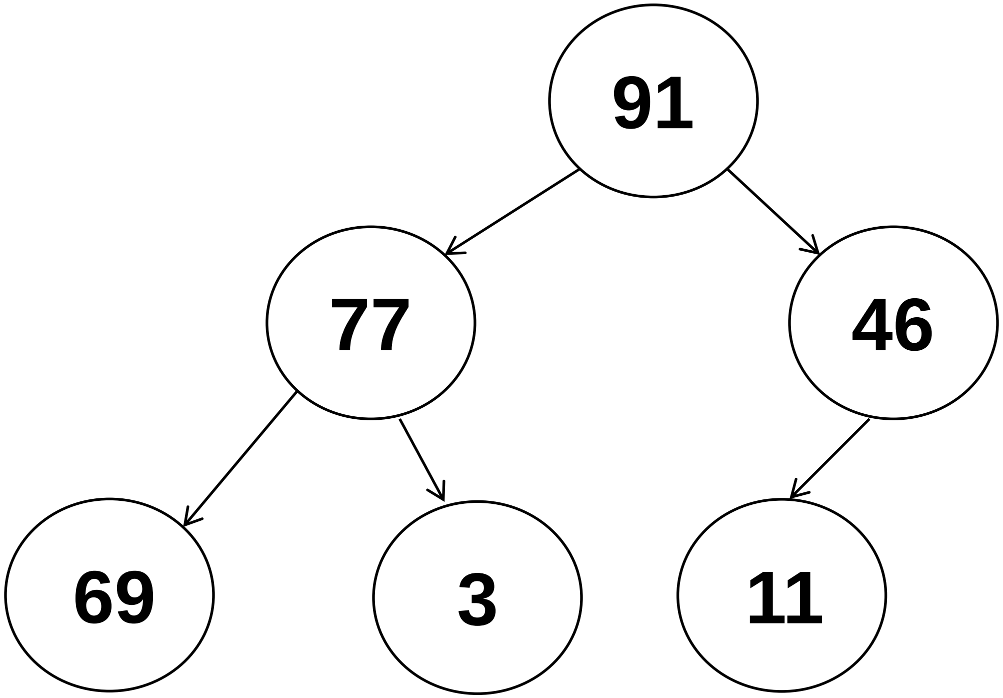


Is It a Max-Heap?

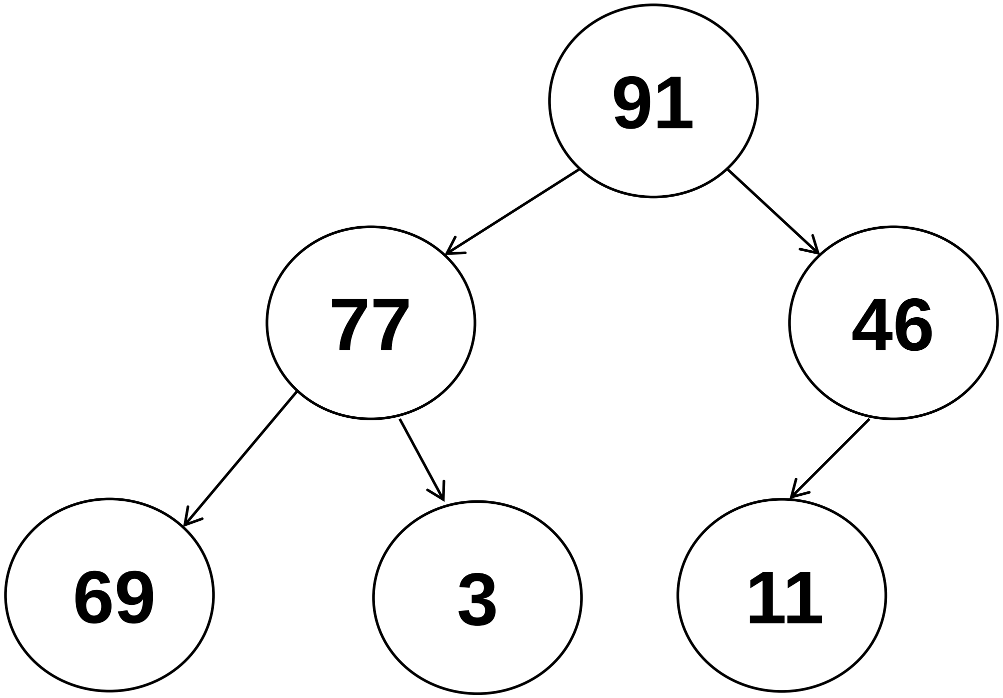


No – max value is not at the top

Is It a Max-Heap?



Is It a Max-Heap?



Yes, it is.

Binary Heap

The height of a node is the number of edges on the longest downward path from the node to a leaf.

- Not to be confused with the depth of a node, the number of edges between the node and the root.
- Also note, if the heap is not a complete tree, some nodes on the same of the level will *not* have the same height.

The height of a heap is the height of its root.

The height *h* of a heap of *n* elements is $h = \Theta(\log_2 n)$

A heap with height *h* has:

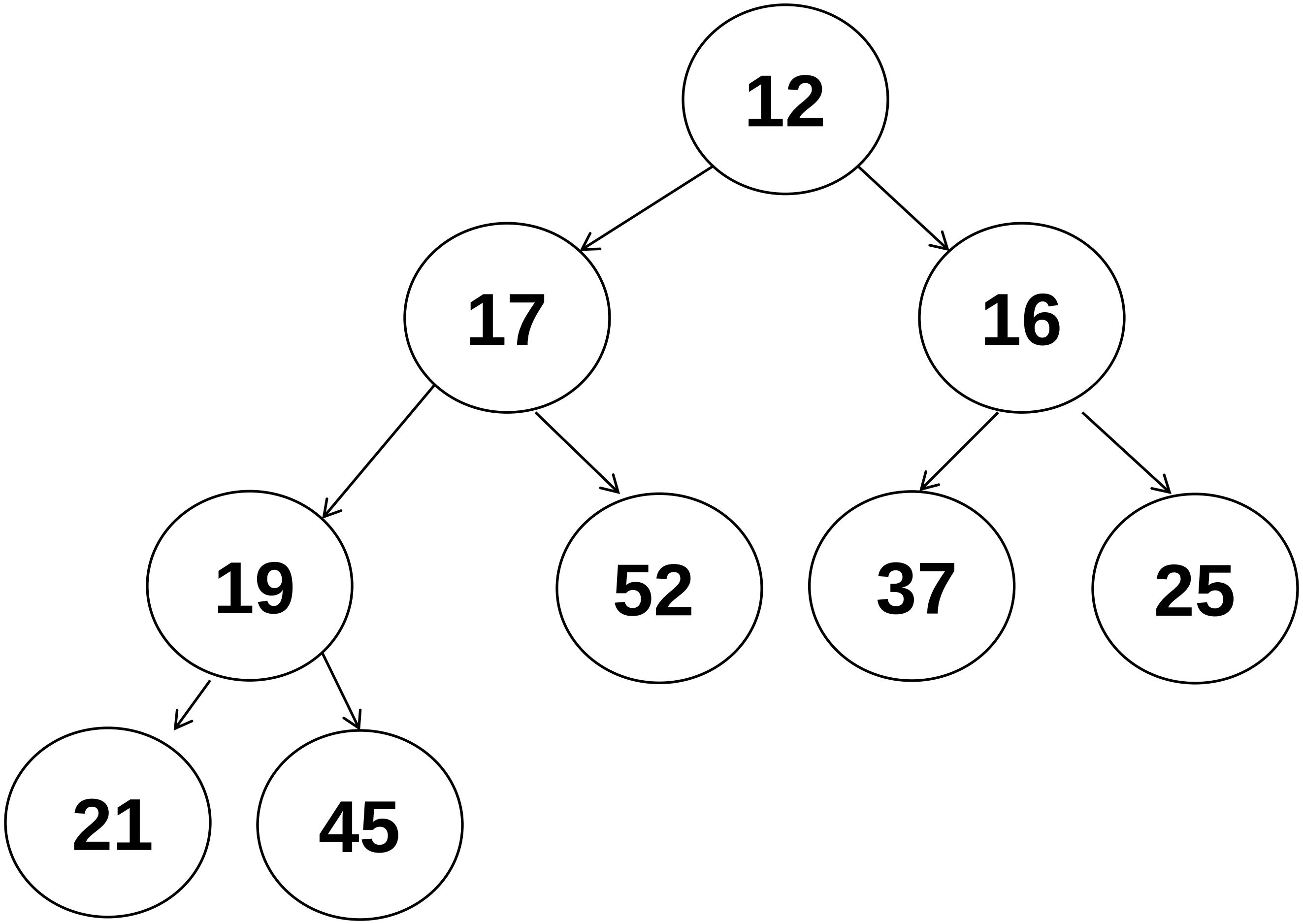
- at least 2^h elements
  - $1 + 2 + 2^2 + \ldots + 2^{h-1} + 1 = 2^h$
- at most $2^{h+1} - 1$ elements
  - $1 + 2 + 2^2 + \ldots + 2^h = 2^{h+1} - 1$

Min-Heap: Insert Operation

- Add new element to next open spot at the bottom of the heap
- If new value is less than parent, swap with parent
- Continue swapping up the tree as long as the new value is less than its parent’s value
- Procedure the same for Max-Heaps, except swap if new value is greater than its parent

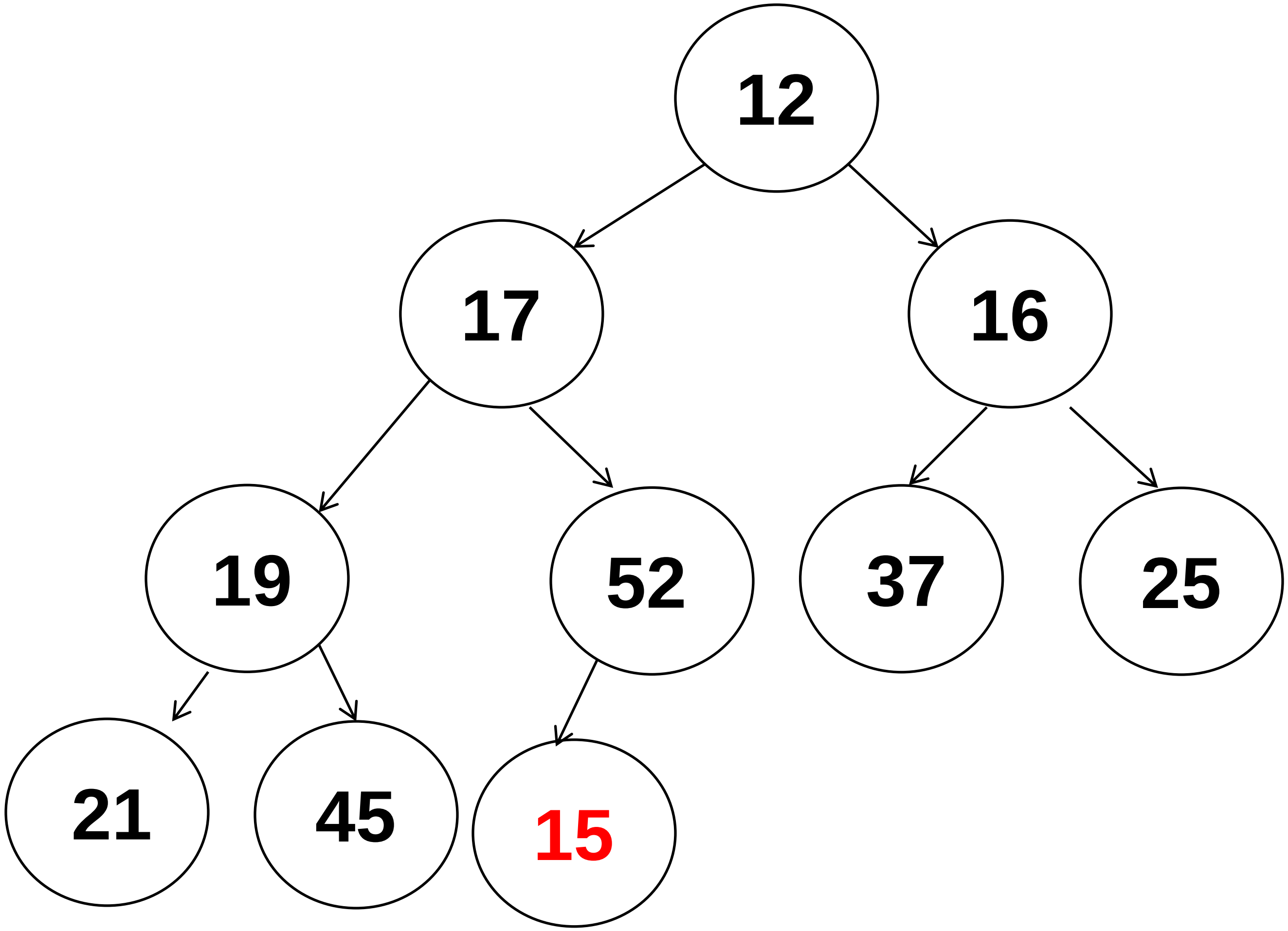
Example: Min-Heap Insert

- Heap before inserting 15



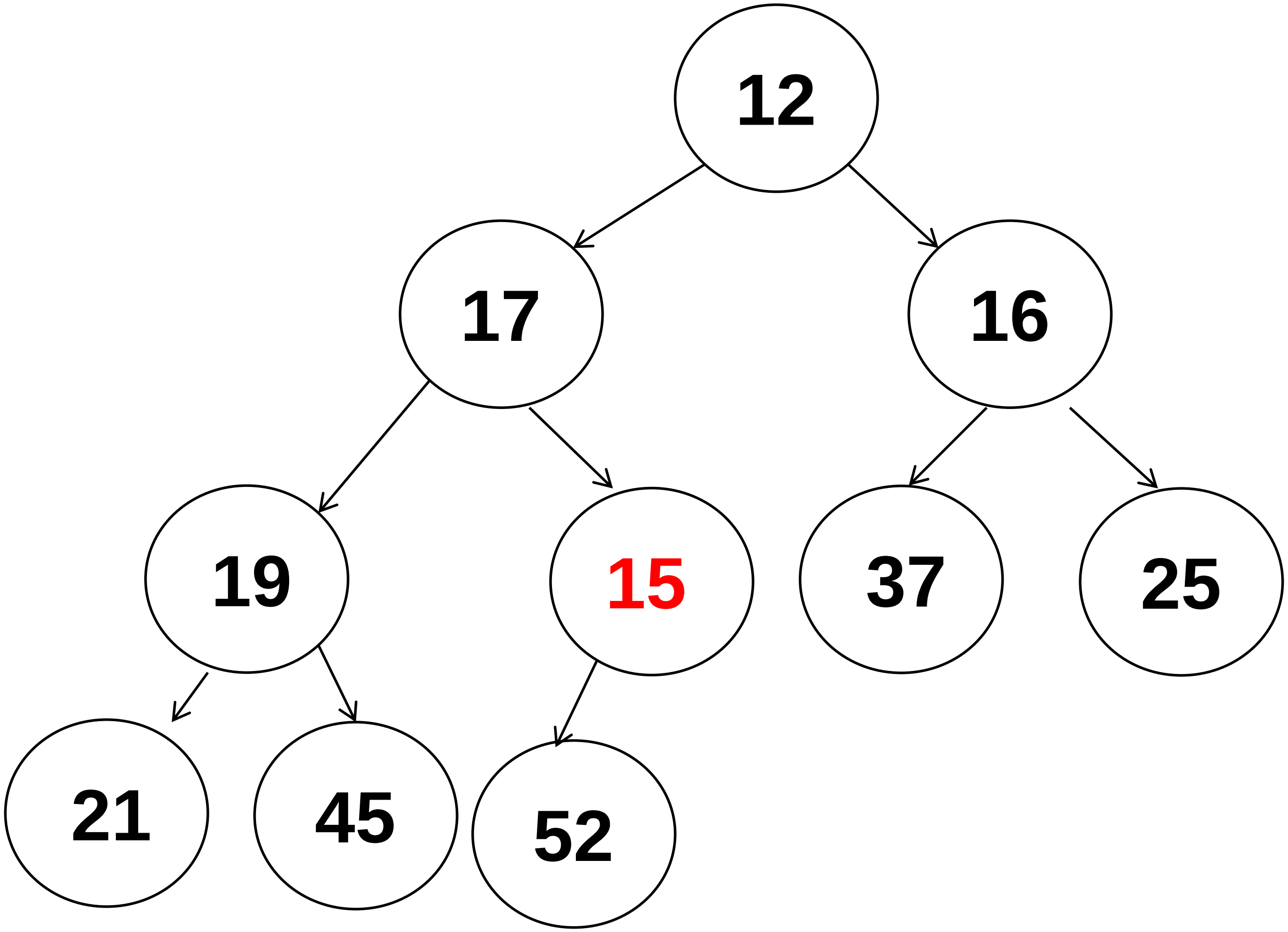
Example: Min-Heap Insert

- Add 15 as next left-most node



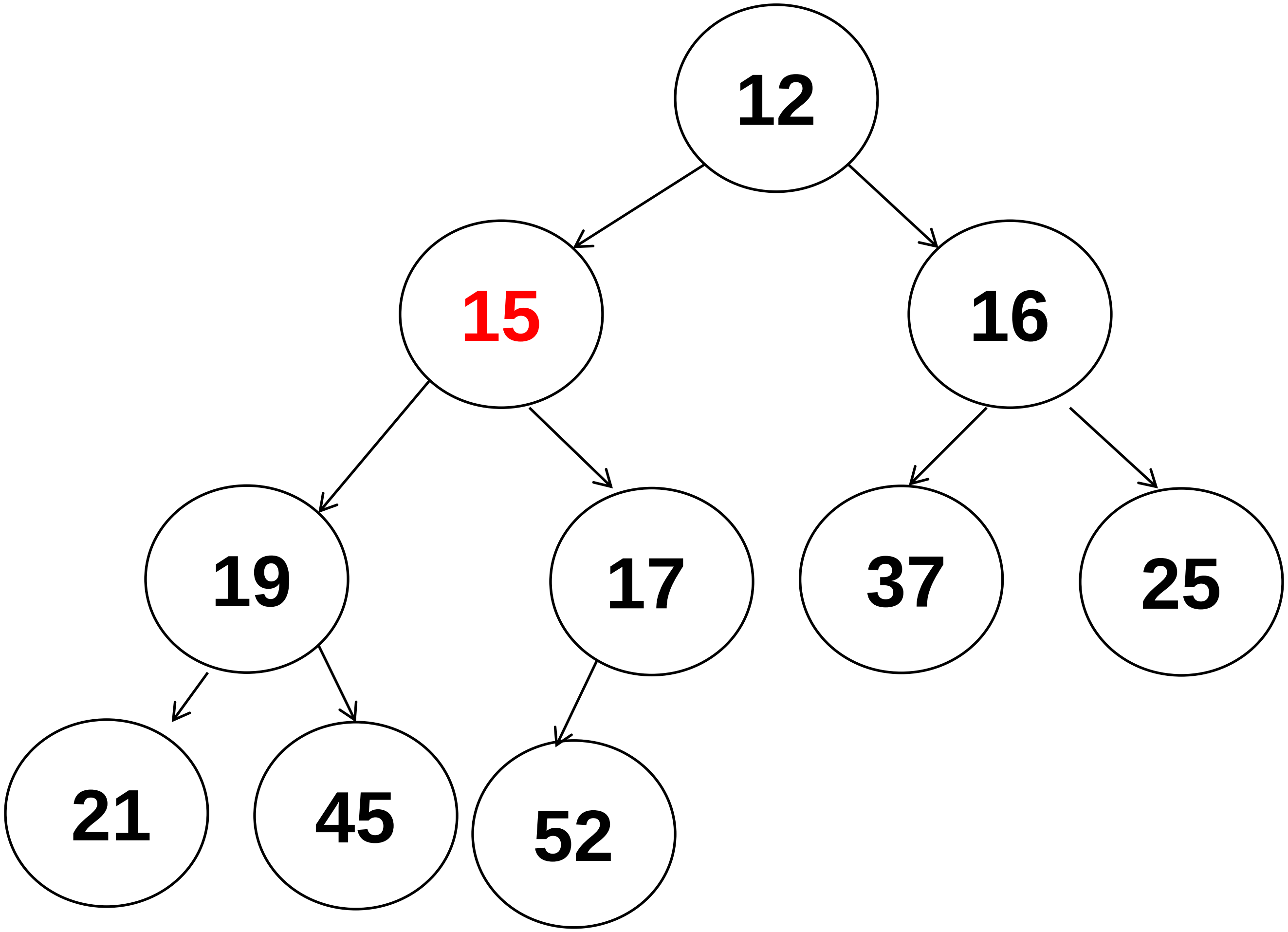
Example: Min-Heap Insert

- Because 15 < 52, swap



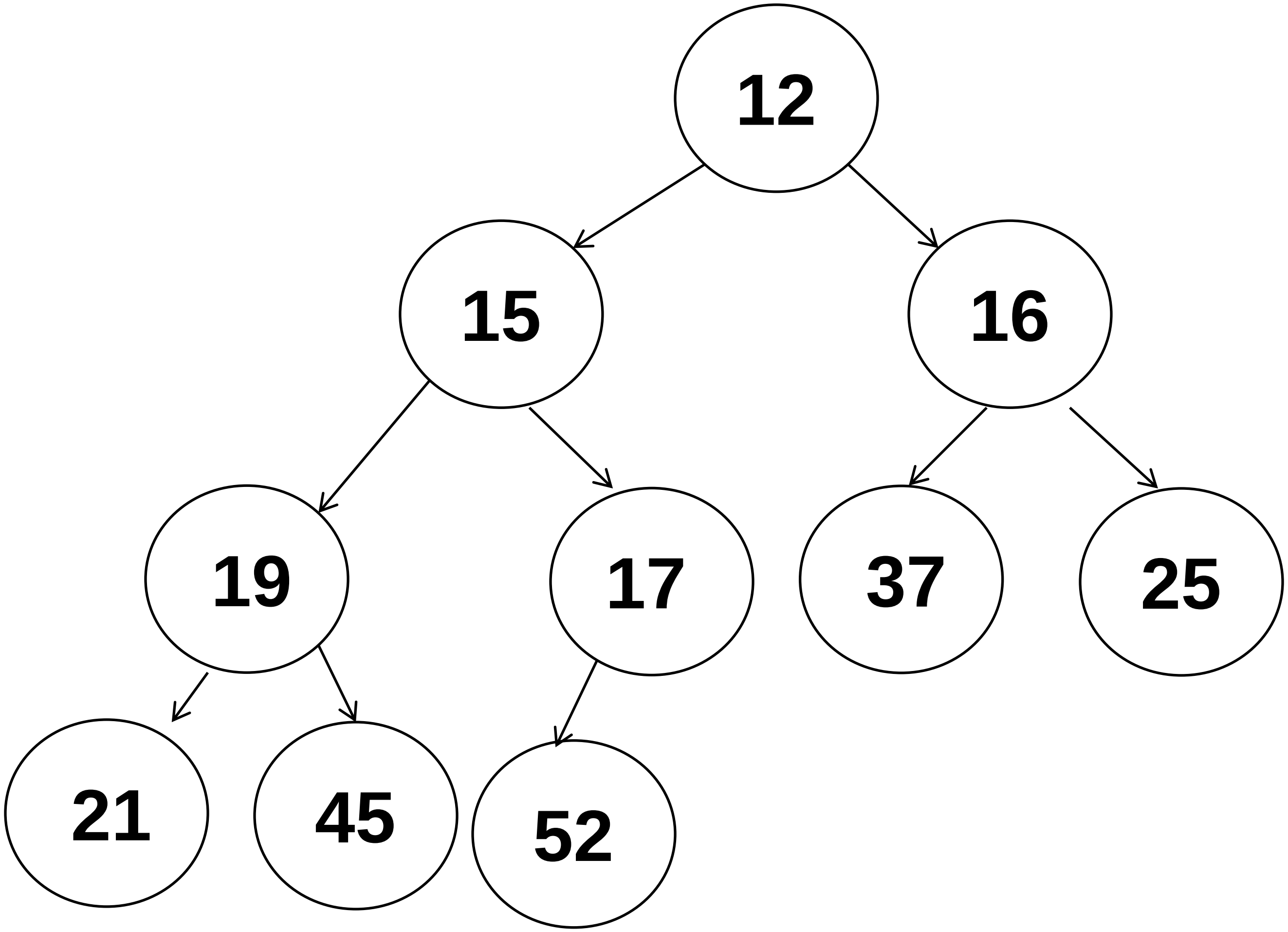
Example: Min-Heap Insert

- Because 15 < 17, swap



Example: Min-Heap Insert

- 15 > 12, so stop swapping

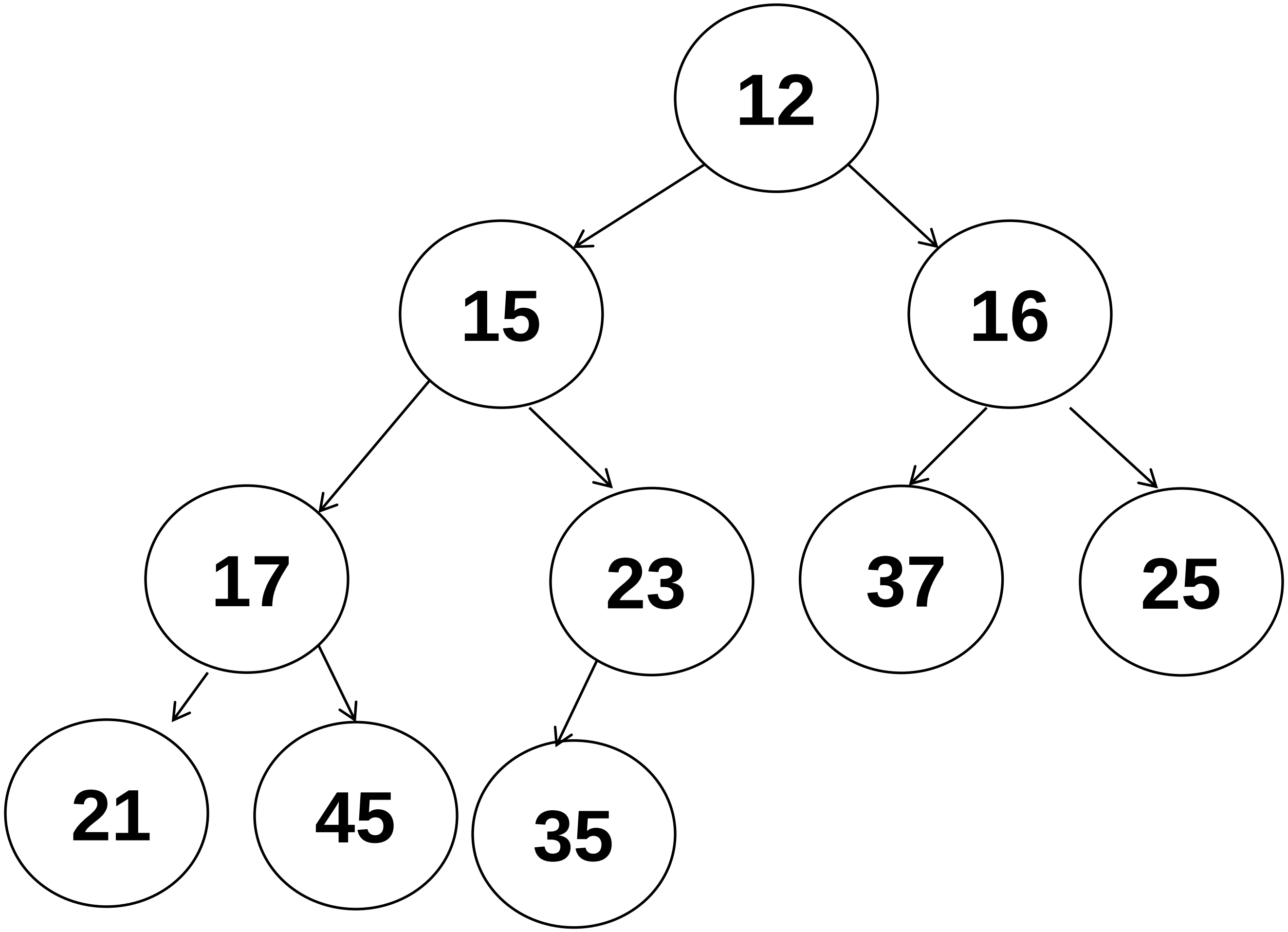


Min-Heap: Extract Operation

- Removes minimum value in the heap
  - Store value at the root (min value) for return
  - Replace value at root with last value in the heap; remove that last node
  - If new root is larger than its children, swap that value with its smaller child.
  - Continue swapping this value down the heap, until neither child is smaller than it is
- Procedure the same for Max-Heap, except replace value with larger of its two children

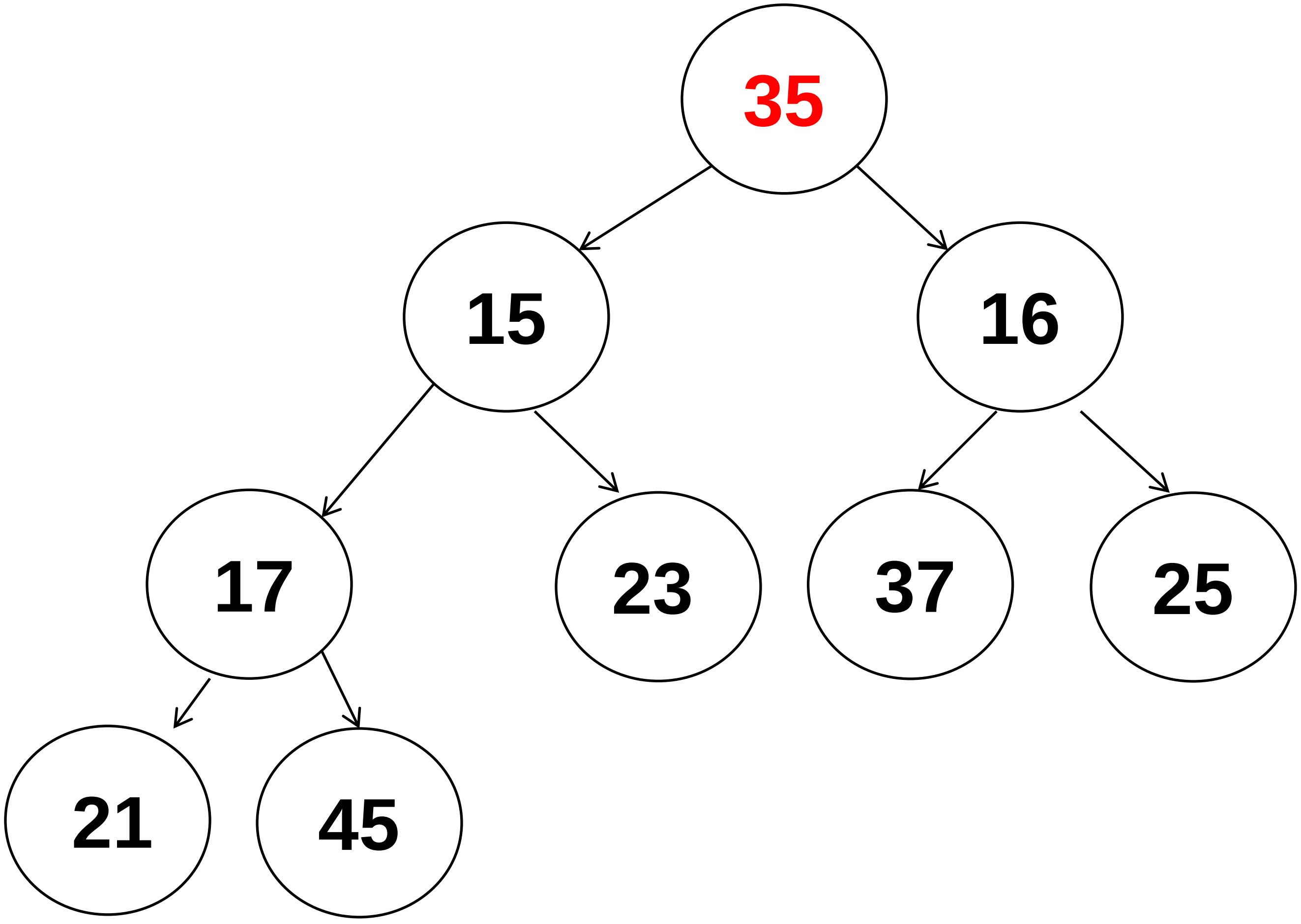
Example: Min-Heap Extract

- Original heap



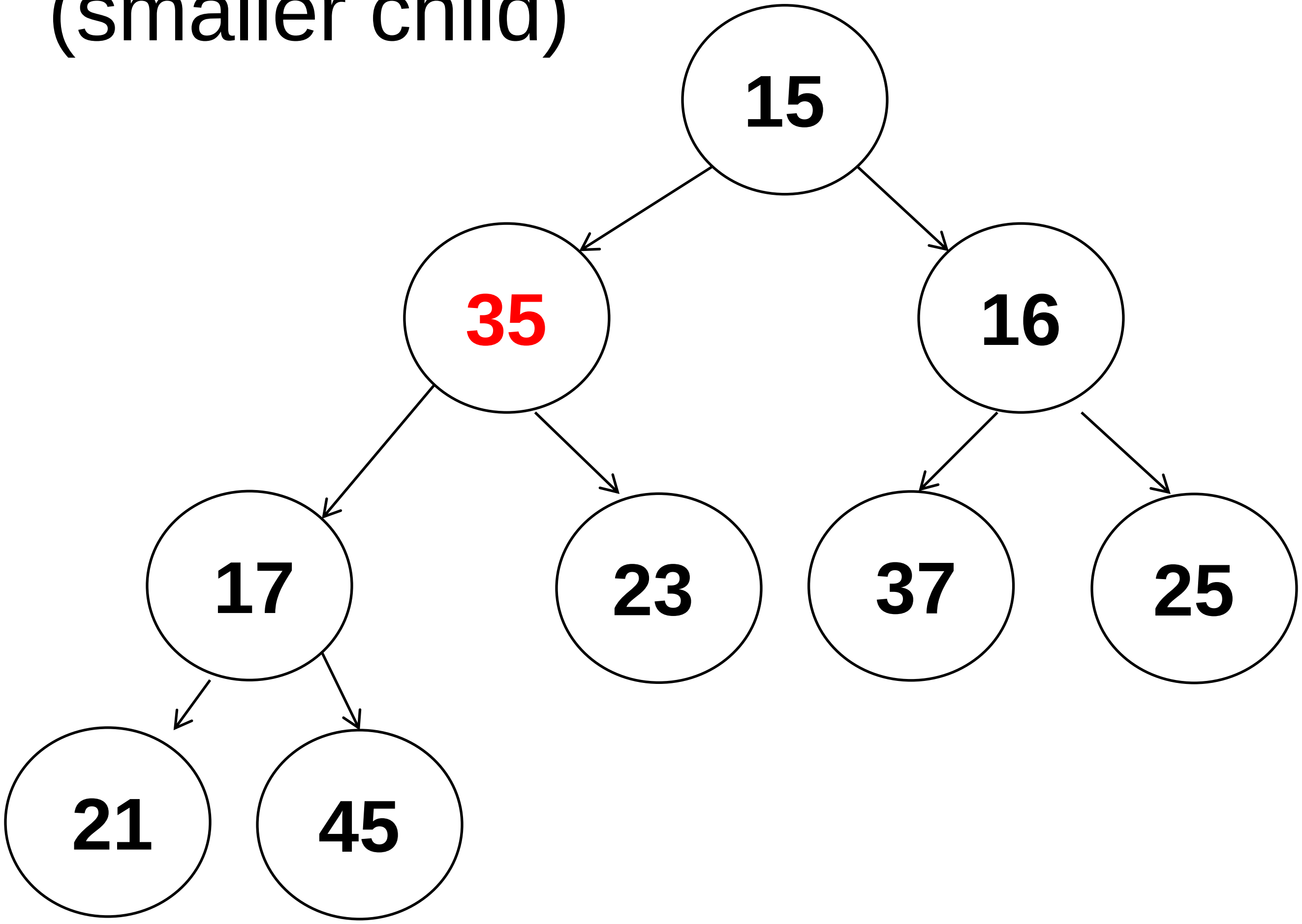
Example: Min-Heap Extract

- Replace 12 with 35, remove last node



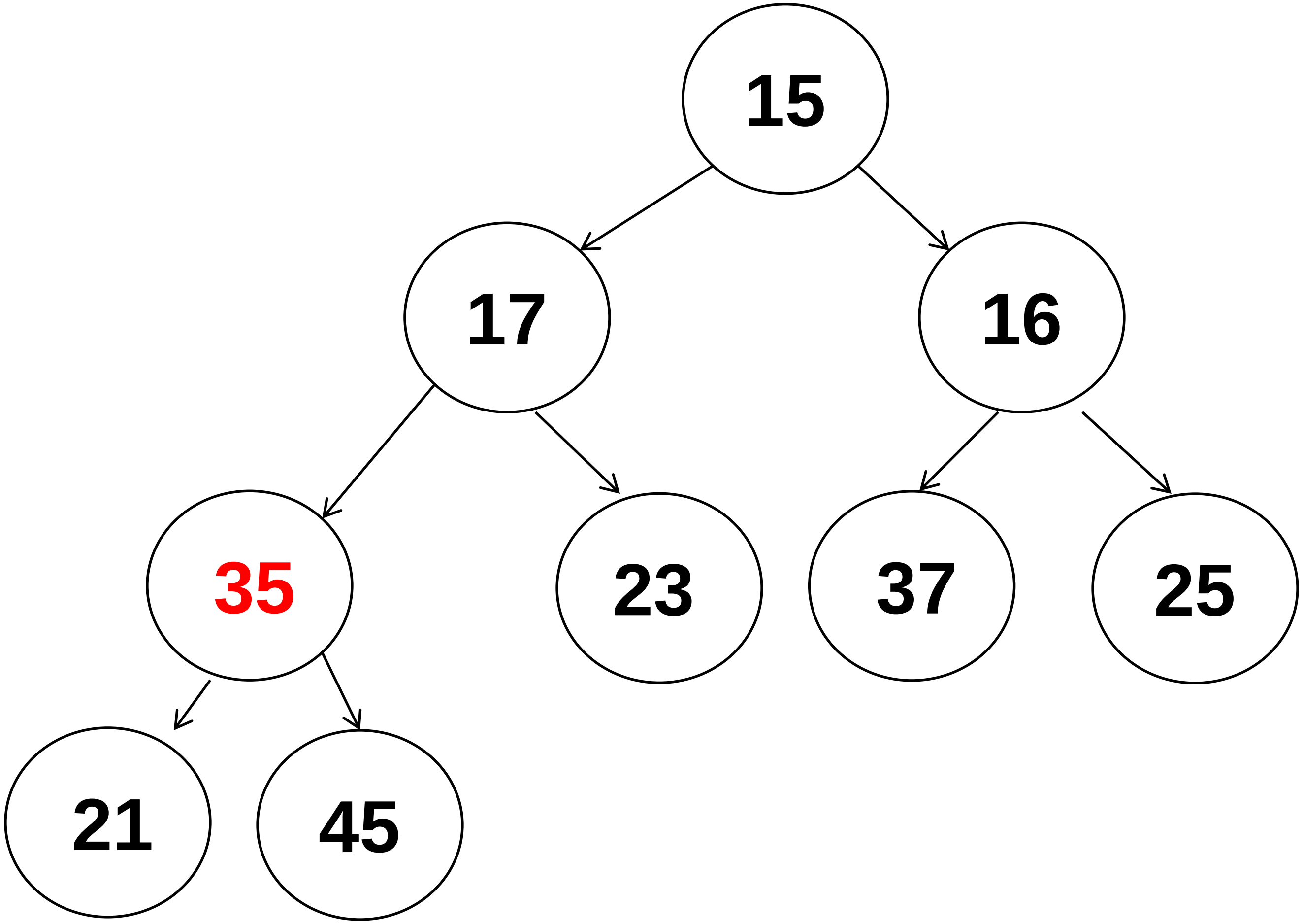
Example: Min-Heap Extract

- Because 35 > 15 and 16, swap with 15 (smaller child)



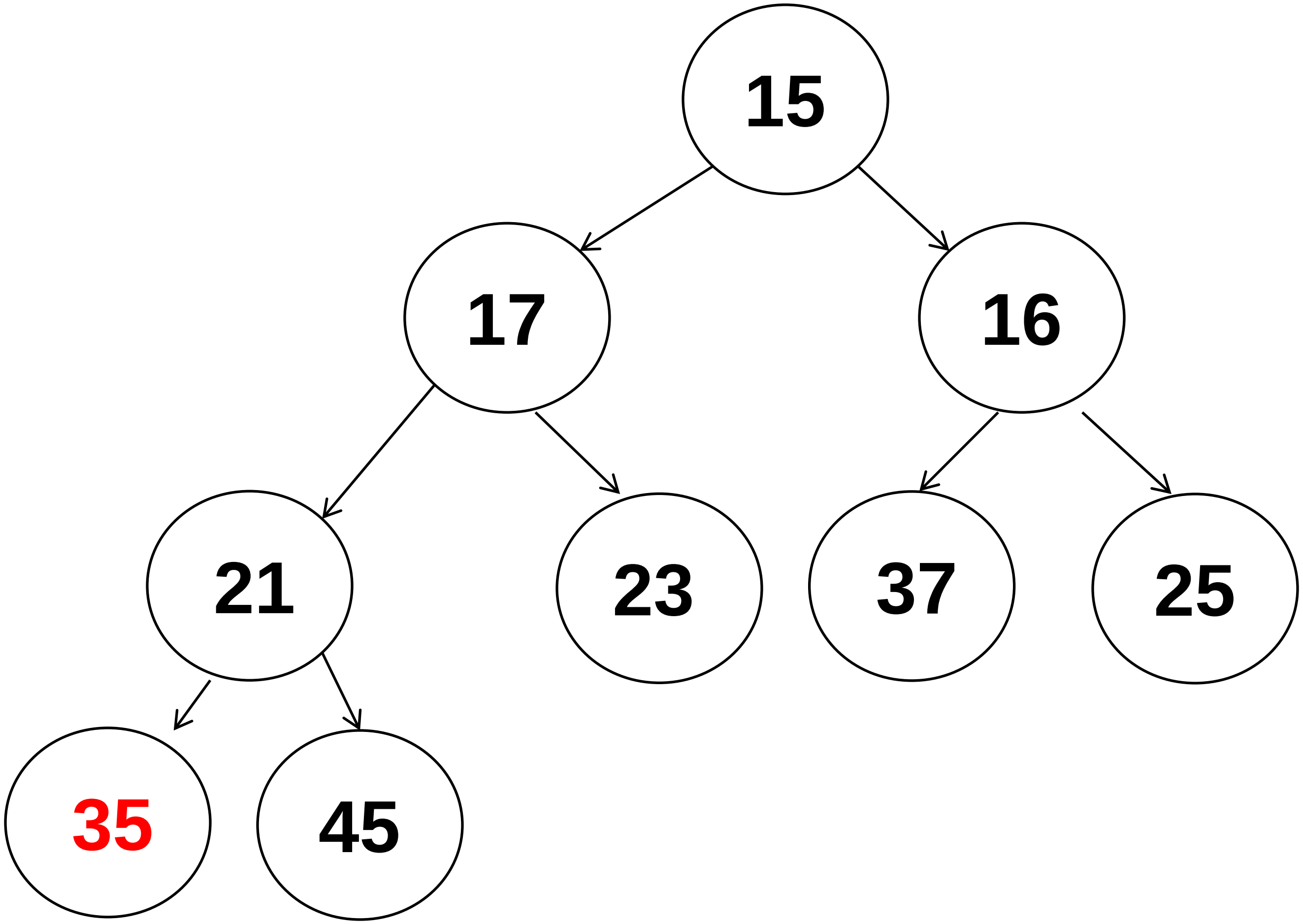
Example: Min-Heap Extract

- Because 35 > 17 and 23, swap 17



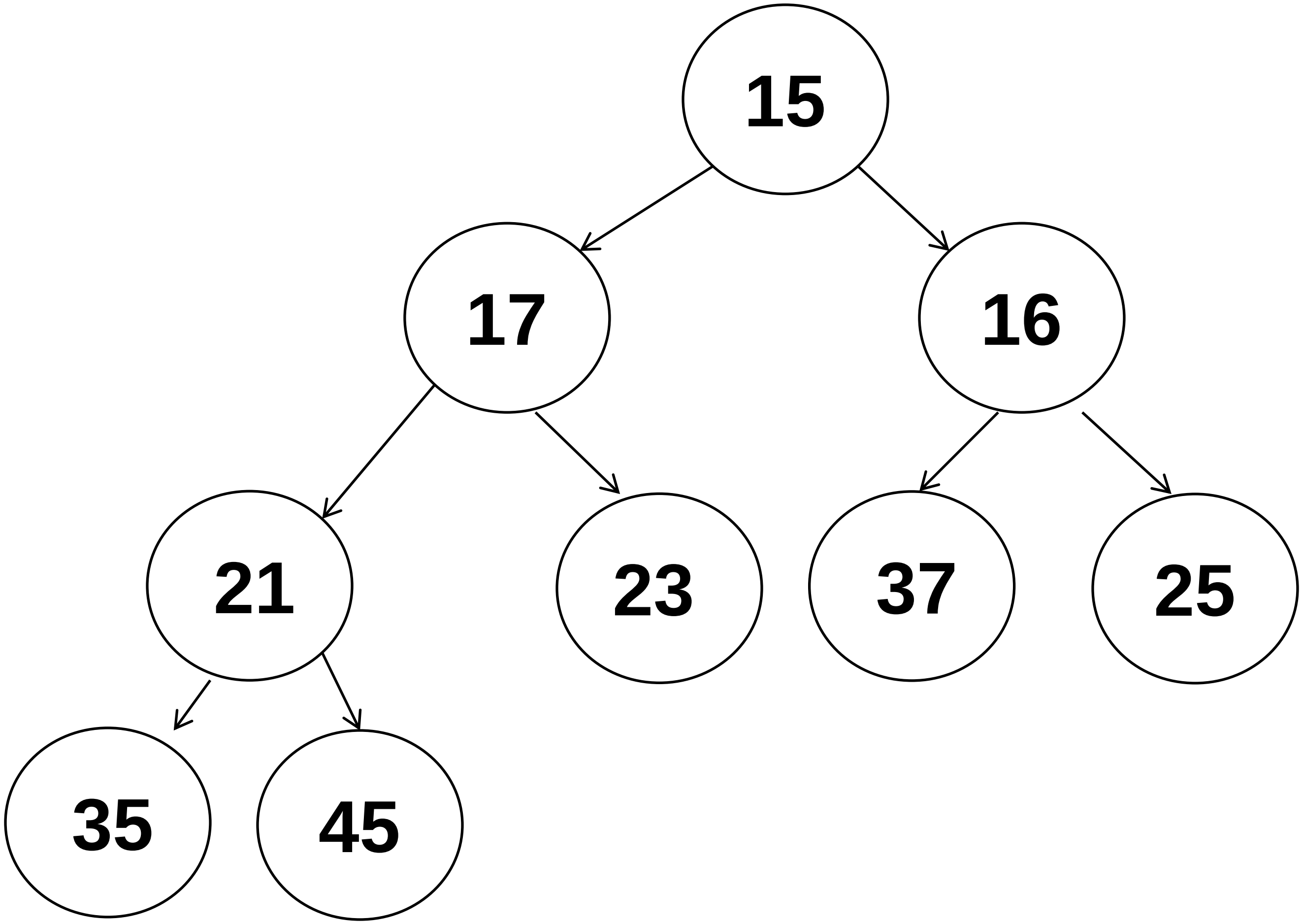
Example: Min-Heap Extract

- Because 35 > 21, swap



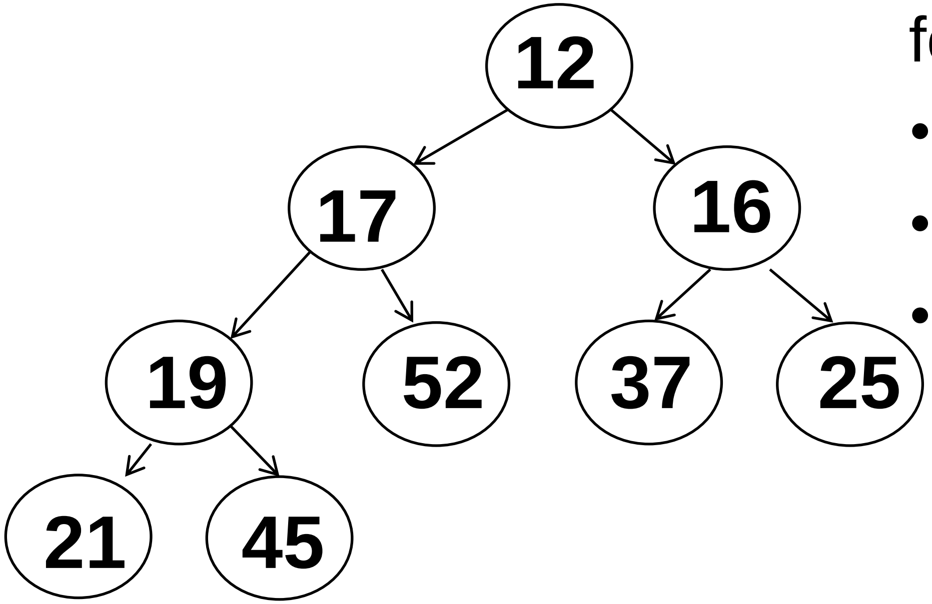
Example: Min-Heap Extract

- 35 now at leaf, stop



Internal Storage

- Interestingly, heaps are often implemented with an array instead of nodes



for element at position `i`:

- parent index: `i / 2`
- left child index: `i * 2`
- right child index: `i * 2 + 1`

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
12	17	16	19	52	37	25	21	45	-	-	-	-	-	-	-

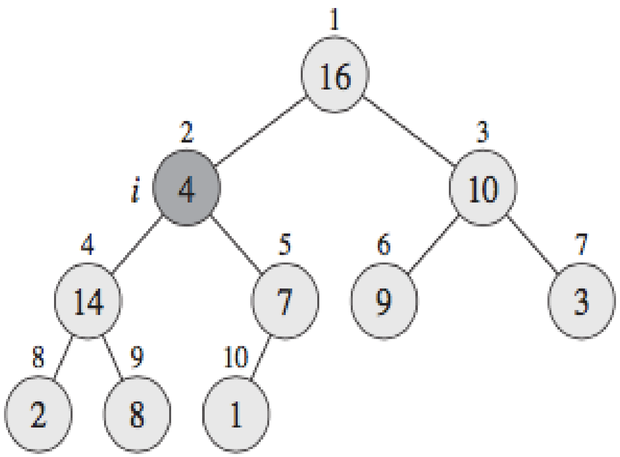
Max-Heap Algorithms

- `MAX-HEAPIFY(A, i)`
  - recursive procedure for maintaining Max-Heap property
  - assume binary trees rooted at `Left(i)` and `Right(i)` are max-heaps, but `A[i]` violates the Max-Heap property

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

Example: Maintaining Max-Heap Property

MAX-HEAPIFY(A, 2)



(a)

MAX-HEAPIFY(A, i)

```
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

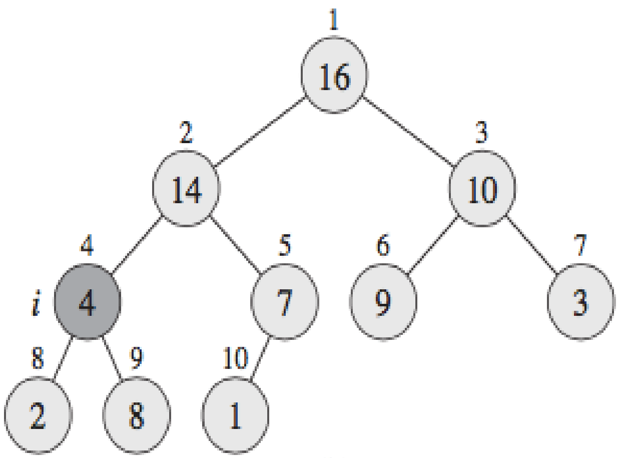
```
l = 2 * 2
r = 2 * 2 + 1
A[4] > A[2]
largest = 4

A[9] < A[4]
no change
4 ≠ 2
exchange A[2], A[4]
Max-Heapify(A, 4)
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	4	10	14	7	9	3	2	8	1	-	-	-	-	-	-

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	14	10	4	7	9	3	2	8	1	-	-	-	-	-	-

Example: Maintaining Max-Heap Property

MAX-HEAPIFY(A, 4)



(b)

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

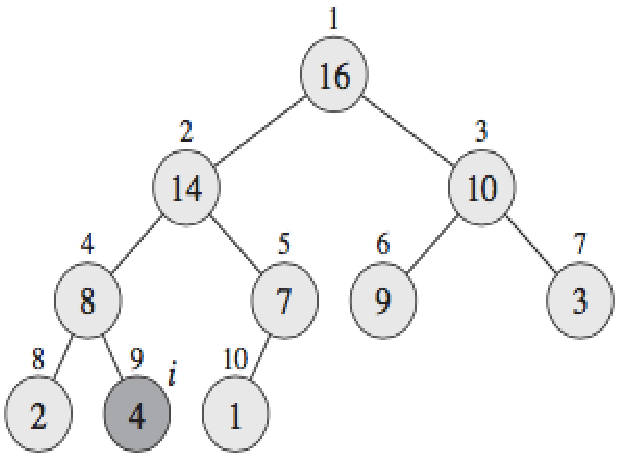
```
l = 4 * 2
r = 4 * 2 + 1
A[4] > A[8]

largest = 4
A[9] > A[4]
largest = 9
9 ≠ 4
exchange A[4], A[9]
Max-Heapify(A, 9)
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	14	10	4	7	9	3	2	8	1	-	-	-	-	-	-

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	14	10	8	7	9	3	2	4	1	-	-	-	-	-	-

Example: Maintaining Max-Heap Property

MAX-HEAPIFY(A, 9)



(c)

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
      largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
      largest = r
if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest)
```

```
l = 9 * 2
r = 9 * 2 + 1
18 > 16 (A.heap-size)

largest = 9
19 > 16
no change
9 = 9

return
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	14	10	8	7	9	3	2	4	1	-	-	-	-	-	-

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	14	10	8	7	9	3	2	4	1	-	-	-	-	-	-

Runtime Max-Heapify

- Runtime based on number of recursive calls
- In the worst case, start at root and have to move value to a leaf.
  – At most, height of heap *h* calls
- So `Max-Heapify(A, i) = O(h)`
  – but $h = \log_2(n)$
- Therefore,

  $\texttt{Max-Heapify(A, i)} = O(\log_2(n))$

Building a Max-Heap

- The leaves are the nodes indexed by:

$$\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n.$$

- We can convert an array `A` into a Max-Heap by using the following procedure

BUILD-MAX-HEAP(A)

$A.heap\text{-}size = A.length$

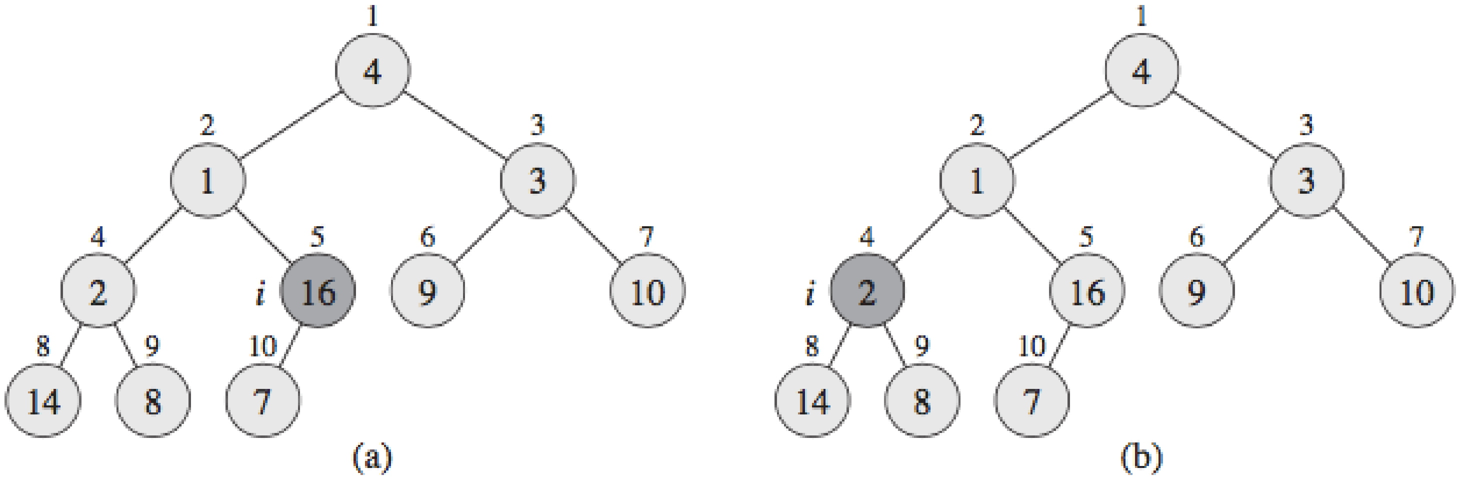
for $i = \lfloor A.length/2 \rfloor$ **downto** 1

    MAX-HEAPIFY(A, i)

Example: Building a Max-Heap

`i = n / 2 = 5` `i = 5 - 1 = 4`

`MAX-HEAPIFY(A, 5)` `MAX-HEAPIFY(A, 4)`



No change **Swap 2 and 14**

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
4	1	3	14	16	9	10	2	8	7	-	-	-	-	-	-

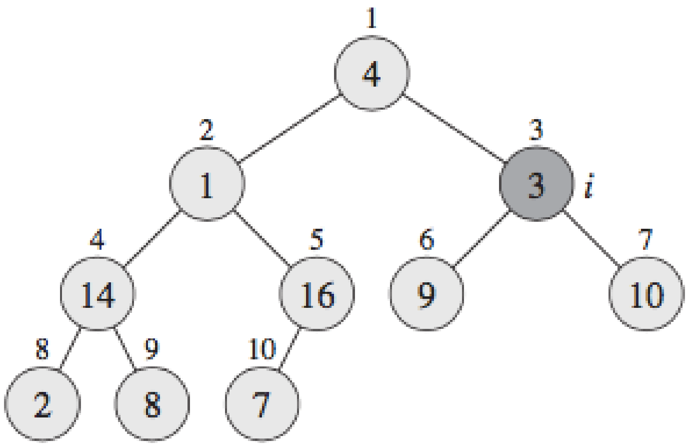
Example: Building a Max-Heap

`i = 4 - 1 = 3`

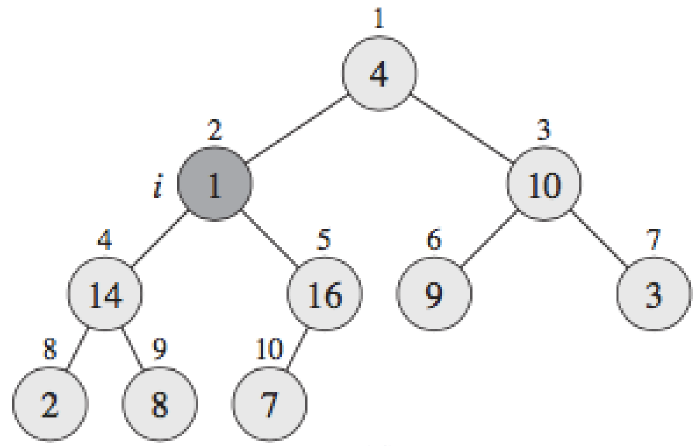
`MAX-HEAPIFY(A, 3)`

`i = 3 - 1 = 2`

`MAX-HEAPIFY(A, 2)`



(c)



(d)

Swap 3 and 10

Swap 1 and 16

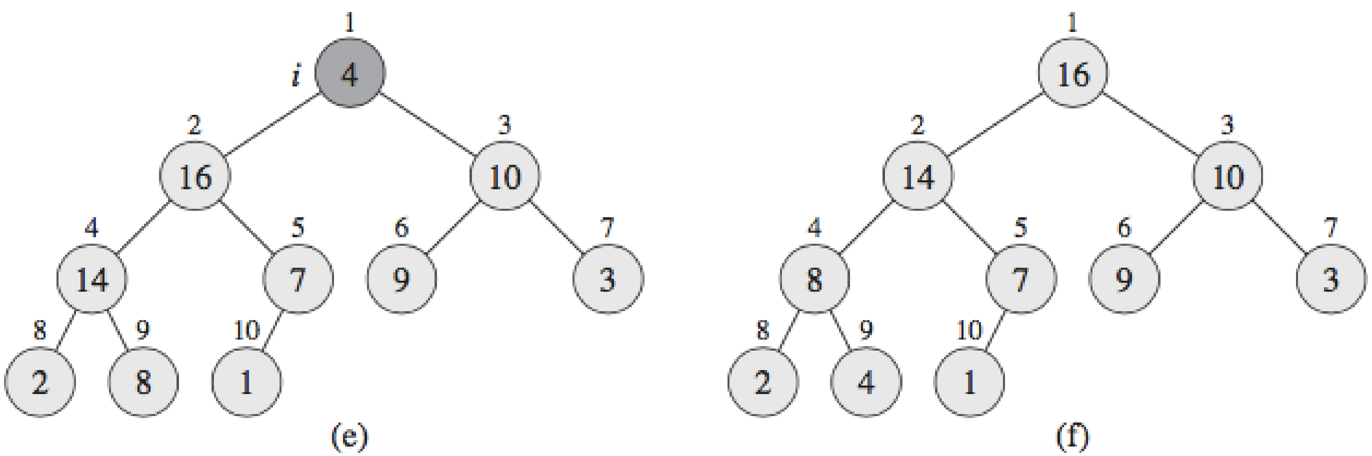
Swap 1 and 7

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
4	16	10	14	7	9	3	2	8	1	-	-	-	-	-	-

Example: Building a Max-Heap

```
i = 2 - 1 = 1
MAX-HEAPIFY(A, 1)
```

Done



Swap 4 and 16

Swap 4 and 14

Swap 4 and 8

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	14	10	8	7	9	3	2	4	1	-	-	-	-	-	-

Runtime Build-Max-Heap

BUILD-MAX-HEAP(A)

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

- A simple upper bound on the running time of Build-Max-Heap is $O(n*\log_2 n)$
- However, we can prove that an asymptotically tighter bound is $O(n)$

Build Max-Heap Running Time

- An *n*-element heap has:
  - height $\lfloor \lg n \rfloor$
  - at most $\lceil n/2^{h+1} \rceil$ nodes at height *h*
  - runtime for `Max-Heapify = O(h)`
- Then, the running time is:

$$\mathbf{T(n)} = \underbrace{\sum_{h=0}^{\lfloor \lg n \rfloor}}_{\text{over each level of heap}} \underbrace{\left\lceil \frac{n}{2^{h+1}} \right\rceil}_{\text{max number of nodes at each level}} \underbrace{O(h)}_{\text{runtime for Max-Heapify}} = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$$

Build-Max-Heap Running Time

- However, we can simplify this expression. Using summation formula (A.8) , we get

$$\mathbf{T(n)} = O\left(n\underbrace{\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}}_{=2}\right)$$

$$= \mathbf{O(2n)} = \mathbf{O(n)}$$

- Therefore, we can build a heap from an unordered array in `O(n)` time.

Heapsort

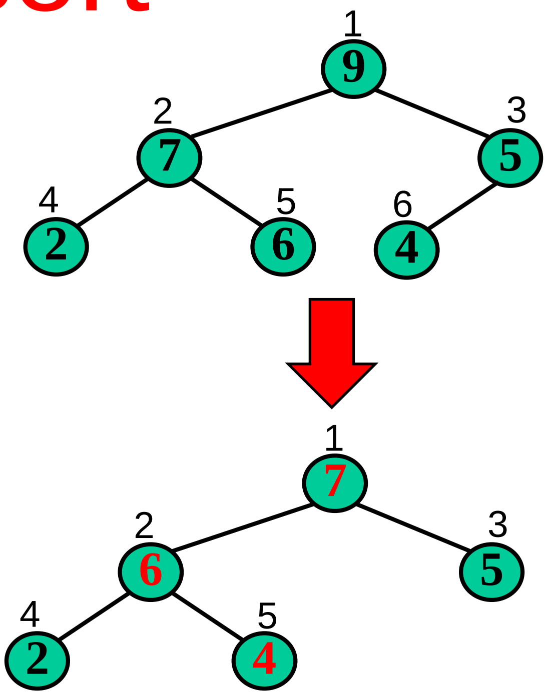
```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  for i = A.length downto 2
      exchange A[1] with A[i]
      A.heap-size = A.heap-size - 1
      MAX-HEAPIFY(A, 1)
```

Example: Heapsort

Begin after Build-Max-Heap

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

```
i = 6
swap A[1], A[6]
A.heap-size = 5
Max-Heapify(A, 1)
```



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
9	7	5	2	6	4	-	-	-	-	-	-	-	-	-	-

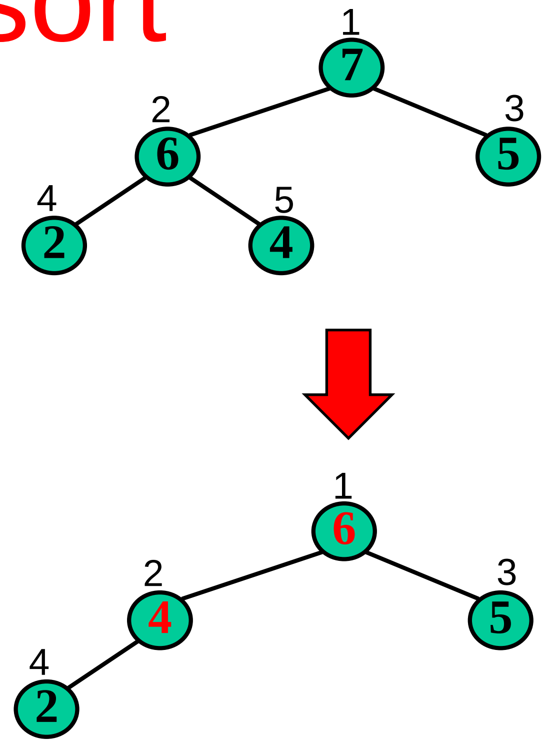
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
7	6	5	2	4	**9**	-	-	-	-	-	-	-	-	-	-

Example: Heapsort

Begin after Build-Max-Heap

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

```
i = 5
swap A[1], A[5]
A.heap-size = 4
Max-Heapify(A, 1)
```



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
7	6	5	2	4	**9**	-	-	-	-	-	-	-	-	-	-

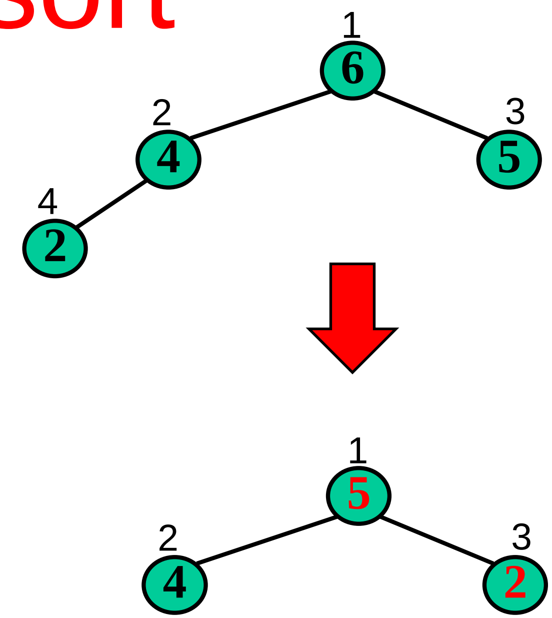
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
6	4	5	2	**7**	**9**	-	-	-	-	-	-	-	-	-	-

Example: Heapsort

Begin after Build-Max-Heap

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

```
i = 4
swap A[1], A[4]
A.heap-size = 3
Max-Heapify(A, 1)
```

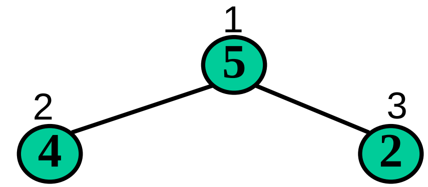


1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
6	4	5	2	**7**	**9**	-	-	-	-	-	-	-	-	-	-

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
5	4	2	**6**	**7**	**9**	-	-	-	-	-	-	-	-	-	-

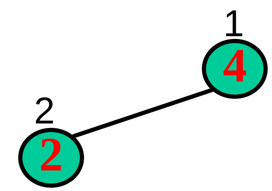
Example: Heapsort

Begin after Build-Max-Heap




```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

```
i = 3
swap A[1], A[3]
A.heap-size = 2
Max-Heapify(A, 1)
```

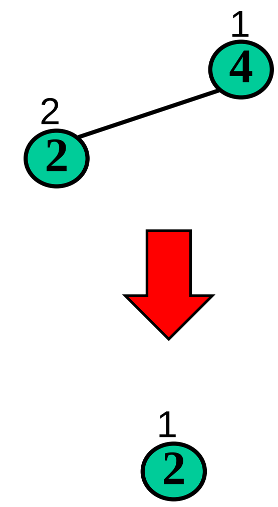


1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
5	4	2	**6**	**7**	**9**	-	-	-	-	-	-	-	-	-	-

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
4	2	**5**	**6**	**7**	**9**	-	-	-	-	-	-	-	-	-	-

Example: Heapsort

Begin after Build-Max-Heap




```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size - 1
    MAX-HEAPIFY(A, 1)
```

```
i = 2
swap A[1], A[2]
A.heap-size = 1
Max-Heapify(A, 1)
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
4	2	5	6	7	9	-	-	-	-	-	-	-	-	-	-

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	4	5	6	7	9	-	-	-	-	-	-	-	-	-	-

Heapsort Running Time

- The heap-sort algorithm takes time $O(n*\log_2 n)$
  - Build-Max-Heap takes $O(n)$ time
  - We do n calls to Max-Heapify which takes $O(\log_2 n)$