CSE 105:
Data Structures and Algorithms-I
(Part 2)

Instructor

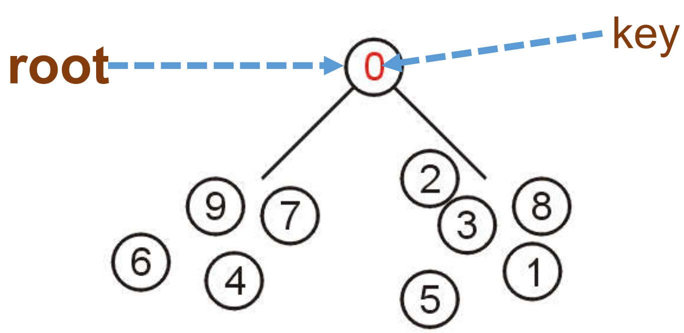
Dr Md Monirul Islam

Heap, Heapsort and Priority Queue

Definition: Heap

A non-empty binary tree is a heap if

- The key associated with the root maintains certain property with the keys associated with either of the sub-trees (if any)

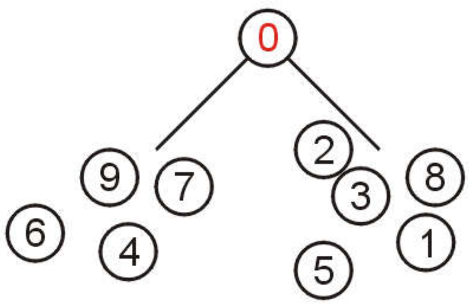


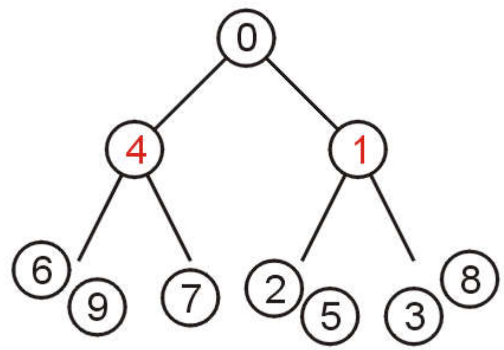
binary tree

Definition: min-Heap

A non-empty binary tree is a **min-heap** if

- The key associated with the root is **less than or equal** to the keys associated with either of the sub-trees (if any)

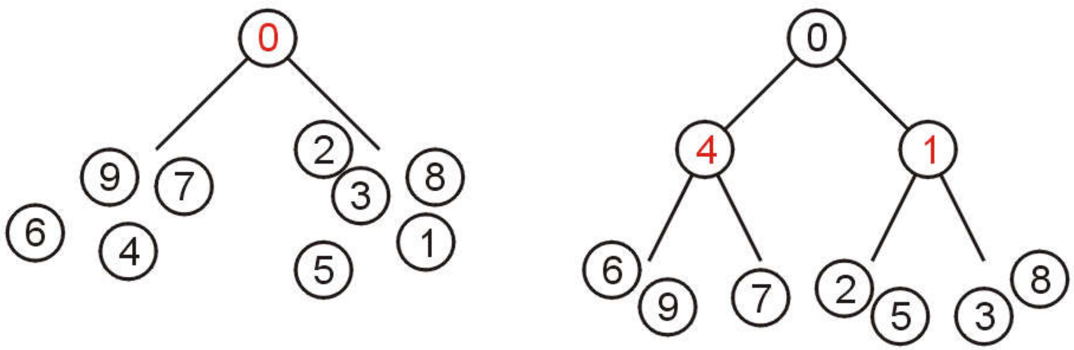




Definition: min-Heap

A non-empty binary tree is a **min-heap** if

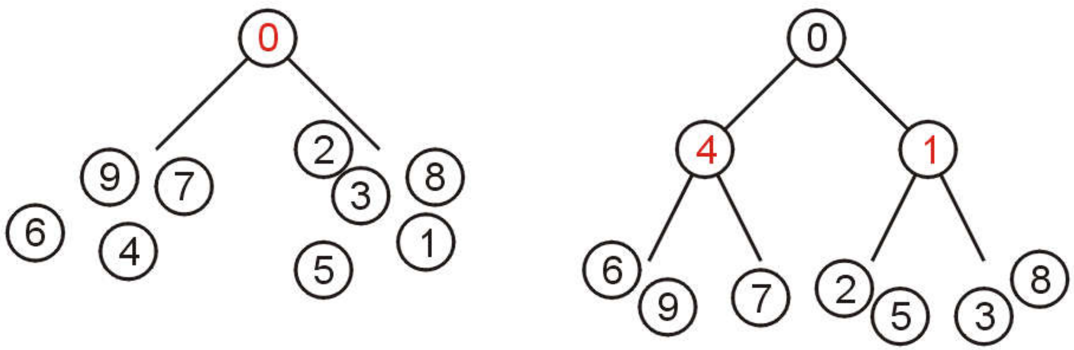
- The key associated with the root is **less than or equal** to the keys associated with either of the sub-trees (if any)
- Both of the sub-trees (if any) are also binary min-heaps



Definition: min-Heap

A non-empty binary tree is a **min-heap** if

- The key associated with the root is **less than or equal** to the keys associated with either of the sub-trees (if any)
- Both of the sub-trees (if any) are also binary min-heaps



From this definition:

- A **single node** is a **min-heap**
- **All keys** in either sub-tree are **greater** than the **root key**

Definition

Important:

There is NO other RELATIONSHIP between

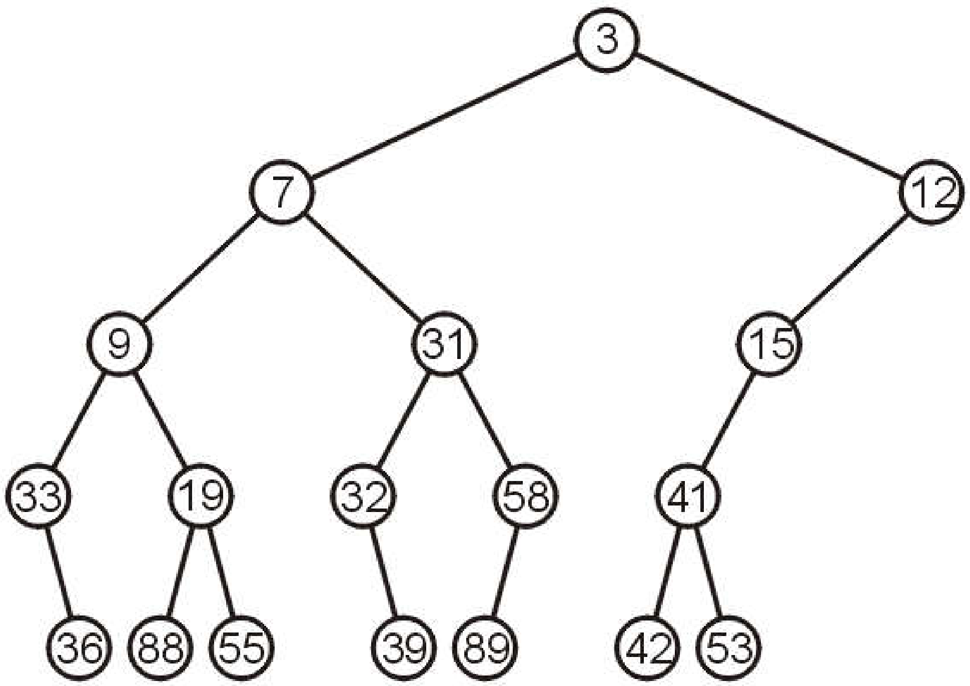
the elements in the TWO SUBTREES

[unlike the BST]

DON'T fail to understand this!

Example

This is a **binary min-heap:**

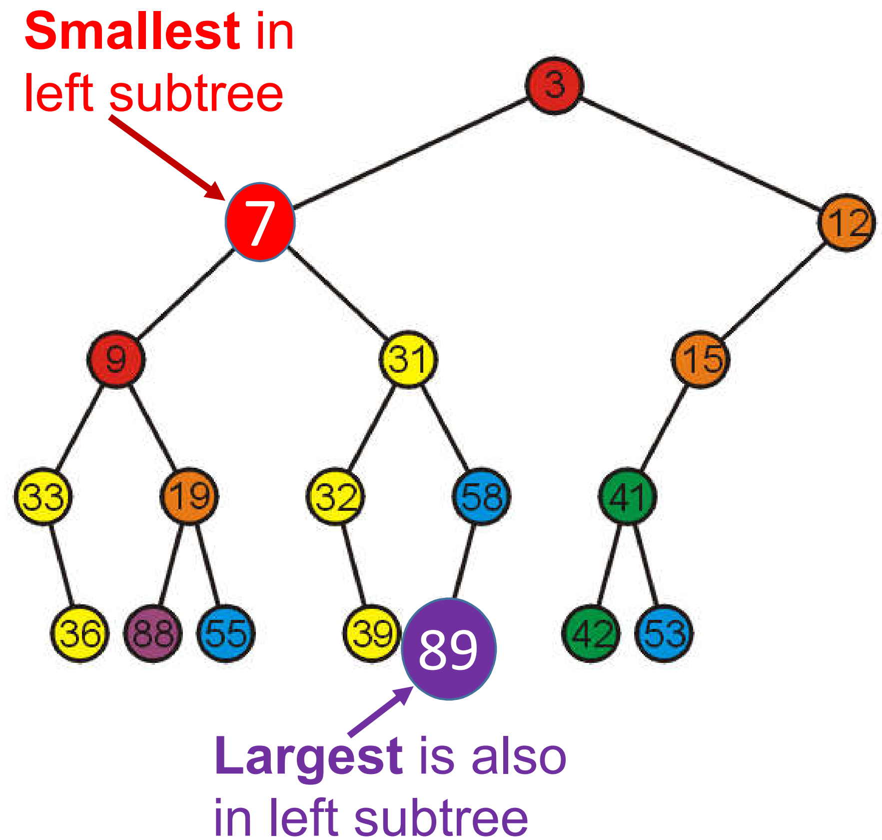


Note: Later, we will implement heap only with complete binary tree…

Example

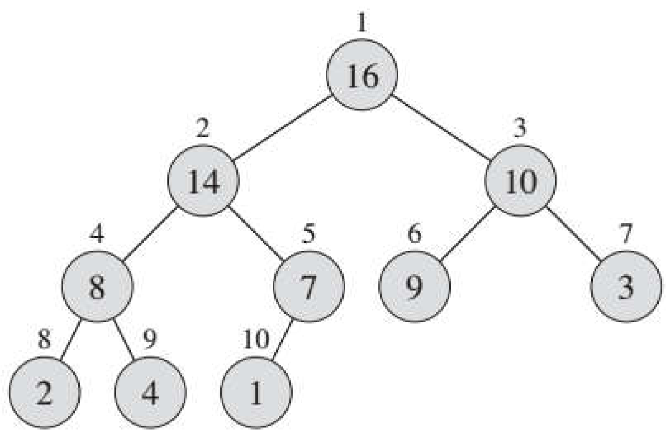
Adding color, we observe

- The left subtree has the **smallest** (7) and the **largest** (89) objects
- No relationship between items with similar priority
  - (just assume for now that the keys are priority values)



Example

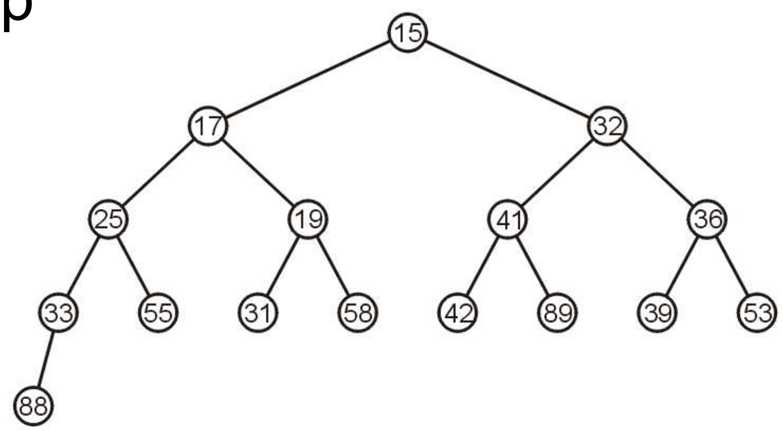
This is a **binary max-heap**:



Note: This max-heap is **implemented with complete binary tree…**

Array Implementation

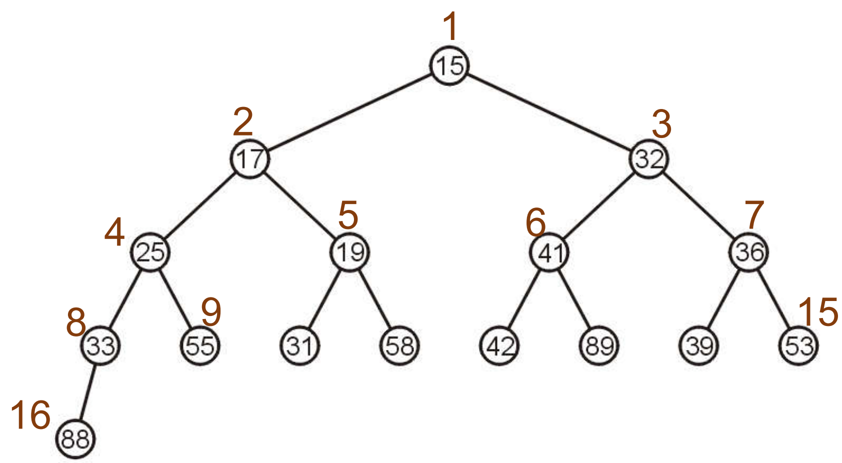
For the min-heap



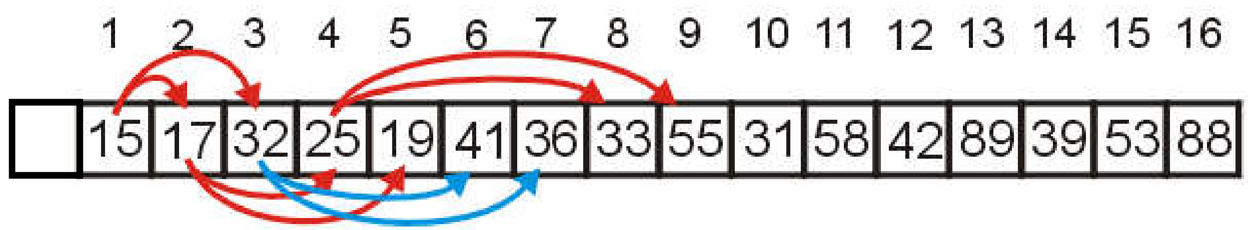
a level order traversal yields:

	15	17	32	25	19	41	36	33	55	31	58	42	89	39	53	88

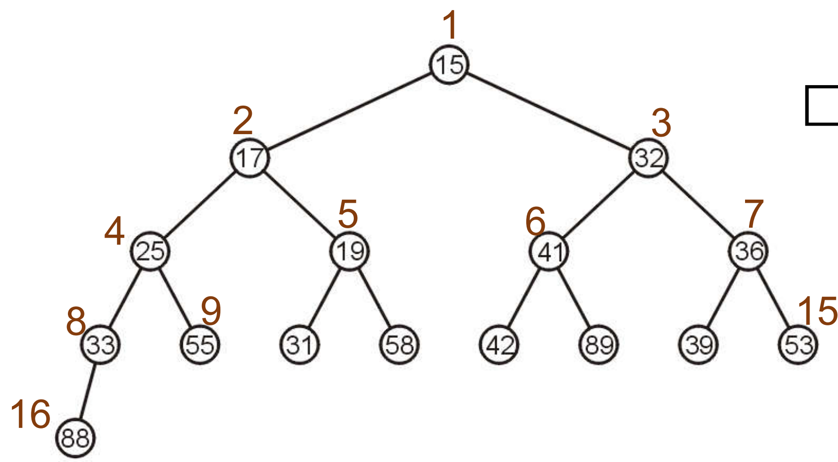
Array Implementation

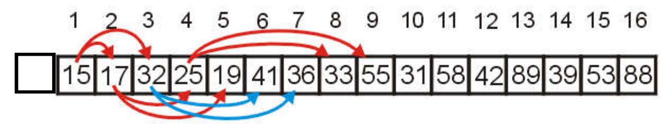


Recall that If we associate an index–starting at 1–with each entry in the breadth-first traversal, we get:



Array Implementation





Given the entry at index k, it follows that:

- The parent of node is a $k/2$
- the children are at $2k$ and $2k + 1$

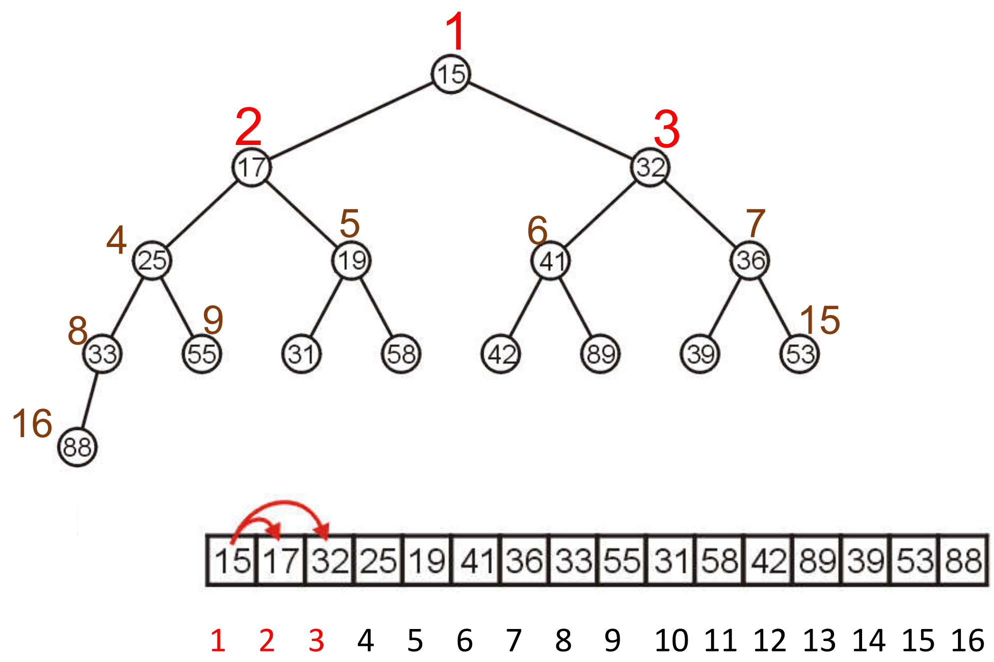
```
parent = k >> 1;
```

```
left_child = k << 1;
right_child = left_child + 1;
```

Cost (trivial): start array at position 1 instead of position 0

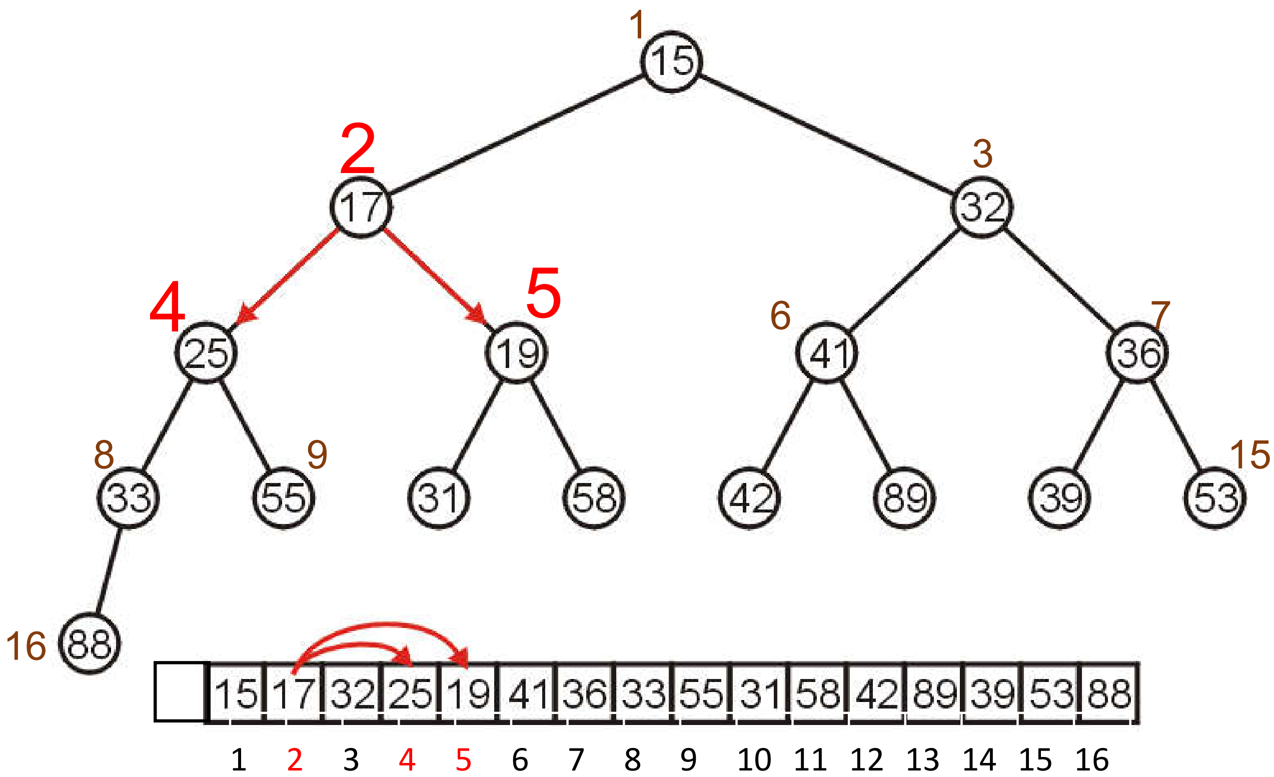
Array Implementation

The children of 15 are 17 and 32:



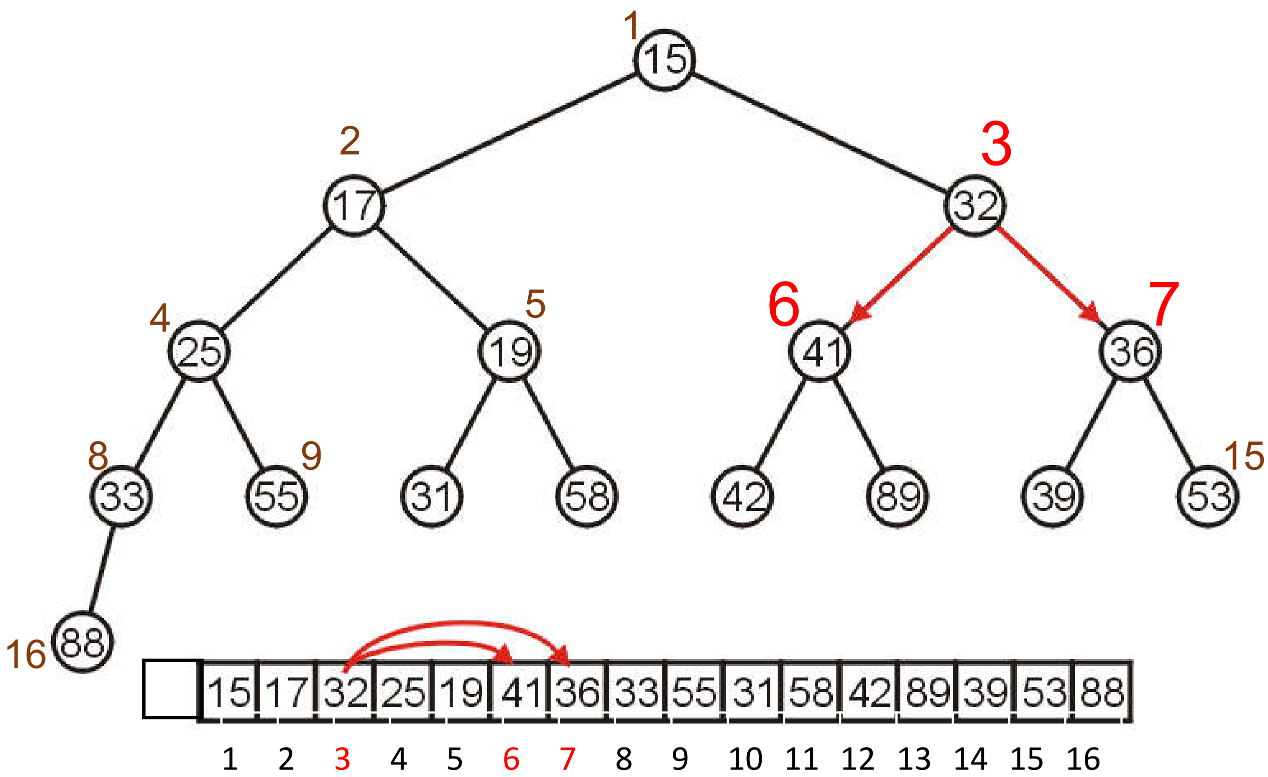
Array Implementation

The children of 17 are 25 and 19:



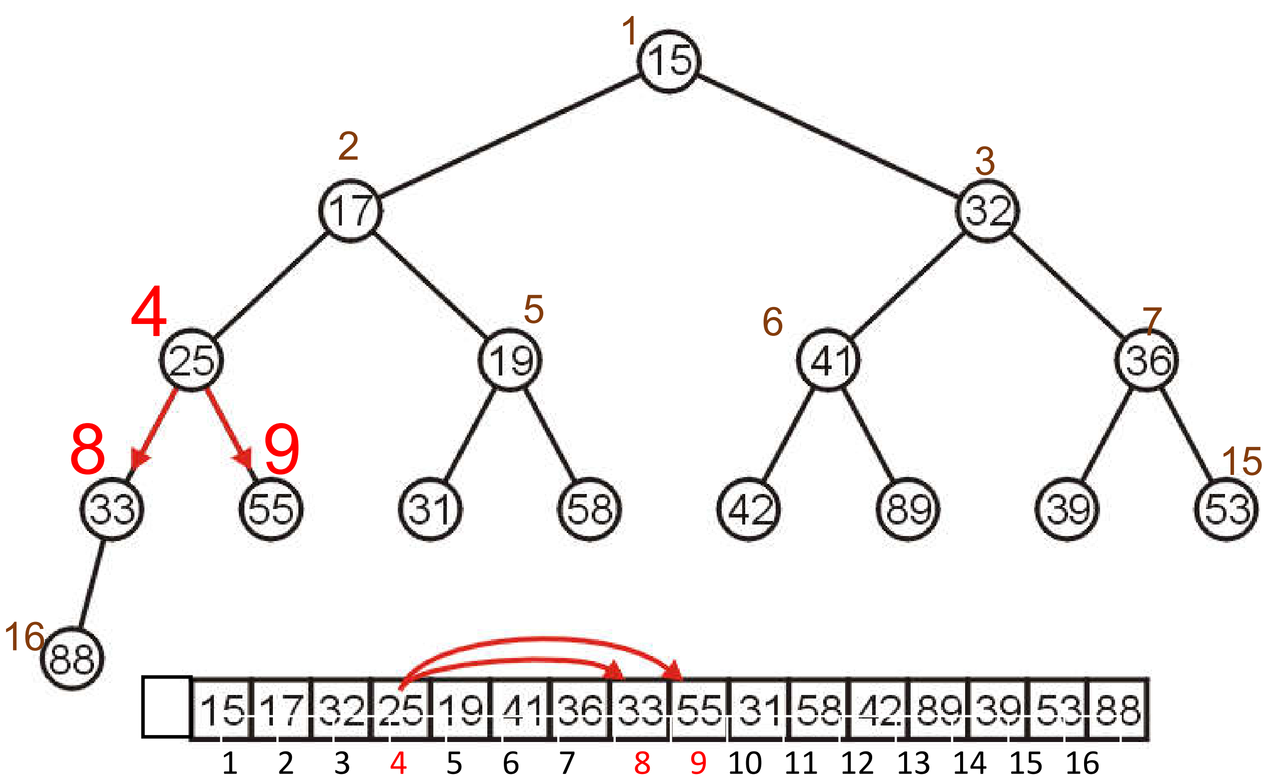
Array Implementation

The children of 32 are 41 and 36:

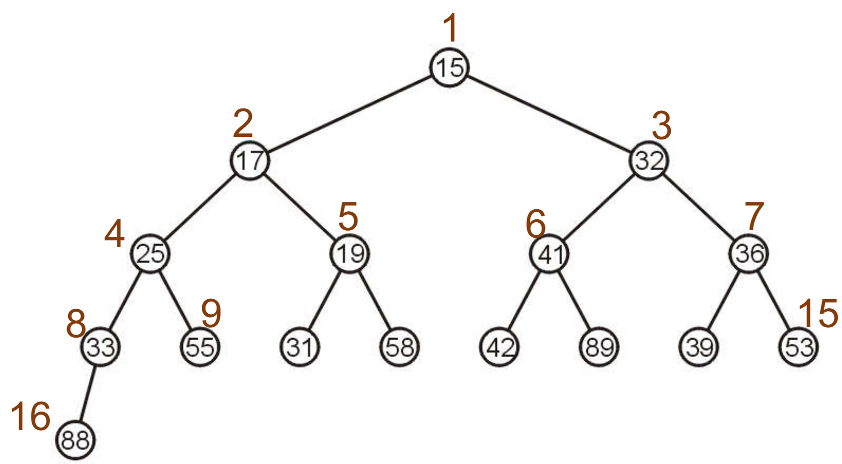


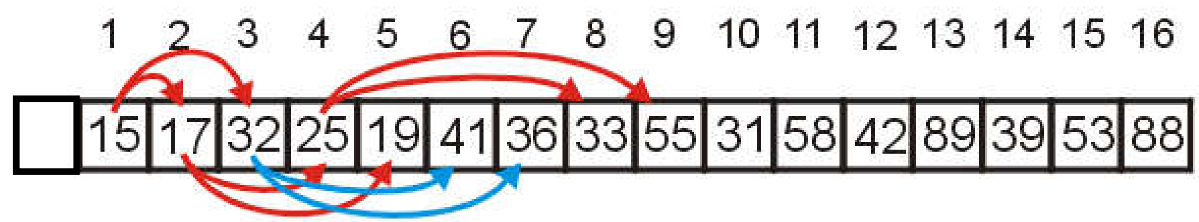
Array Implementation

The children of 25 are 33 and 55:



Array Implementation

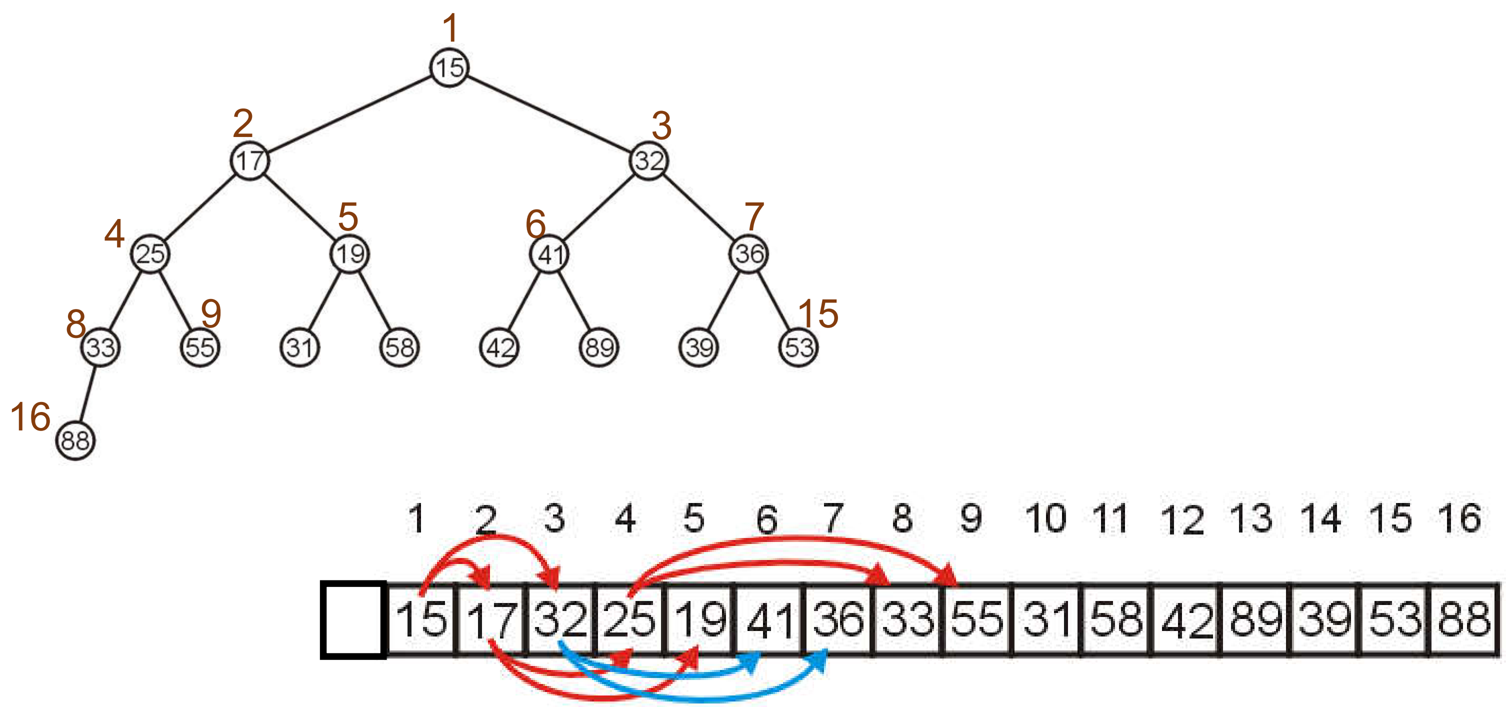




Min-heap property to be maintained:

$$Array\,[parent] \leq Array\,[child]$$

Array Implementation



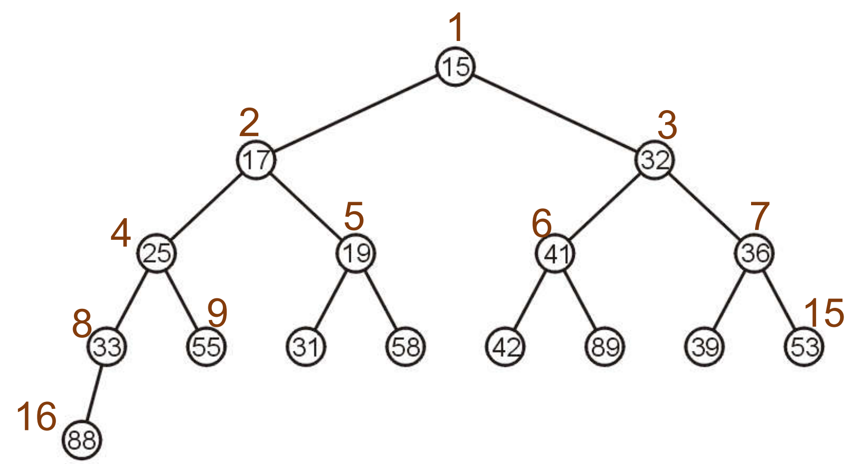
Min-heap property to be maintained:

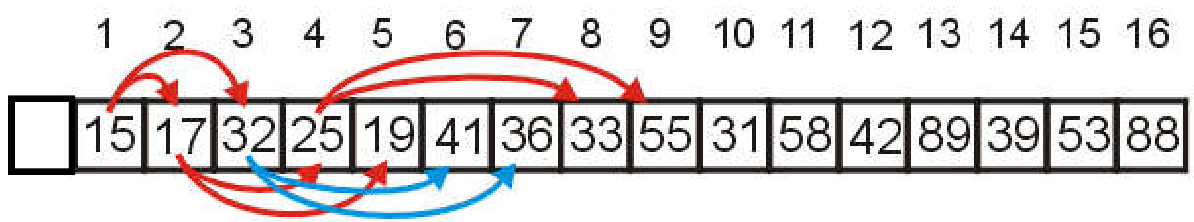
$$Array\ [parent(i)] \leq Array\ [i]$$

Operations

We will consider three operations:

- Top [actually, getMinimum in this case]
- Push [insert]
- Pop [remove or extractMinimum]

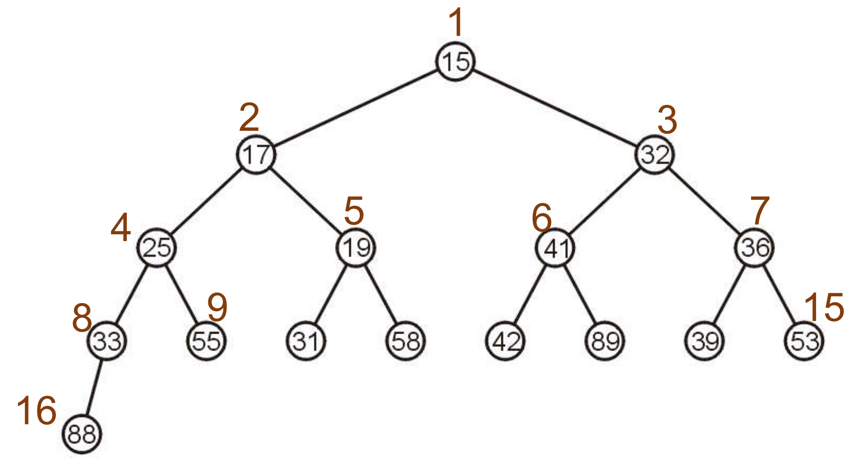


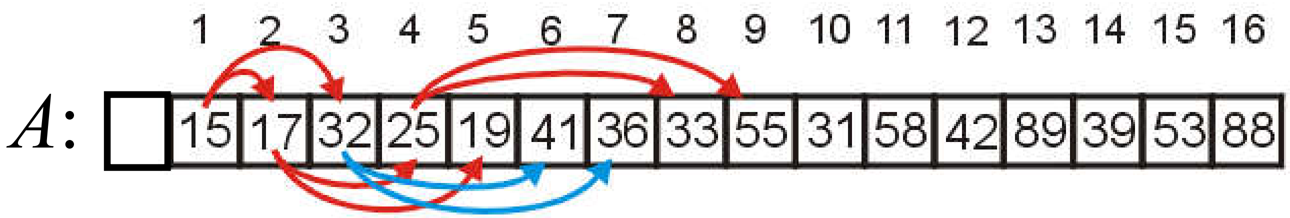


Operation: Top

- Very trivial

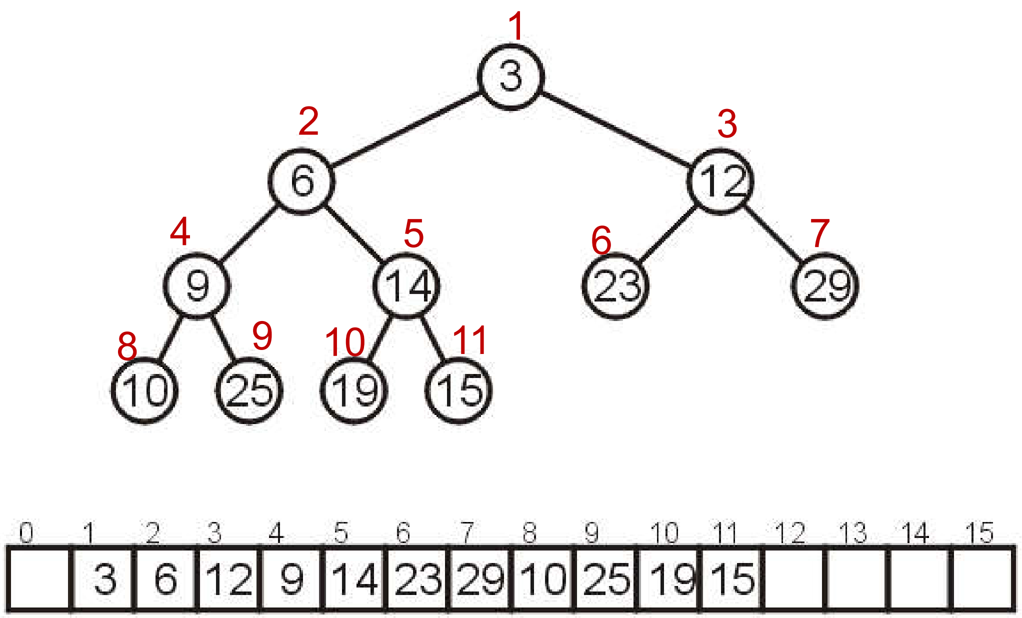
return $A[1]$





Array Implementation: Push

Consider the following heap, both as a tree and in its array representation

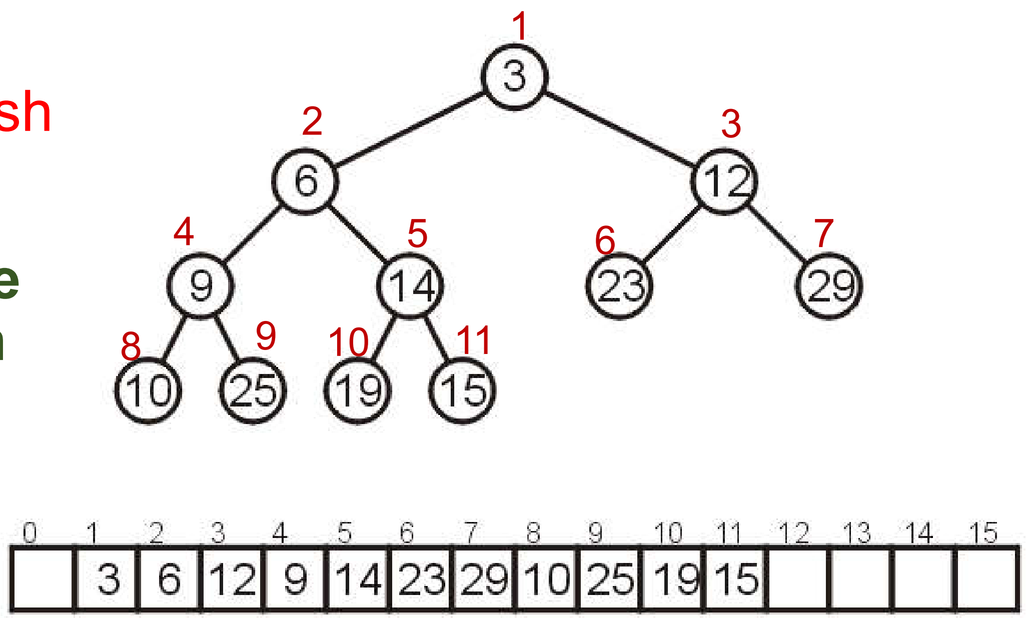


Array Implementation: Push

Consider the following heap, both as a tree and in its array representation

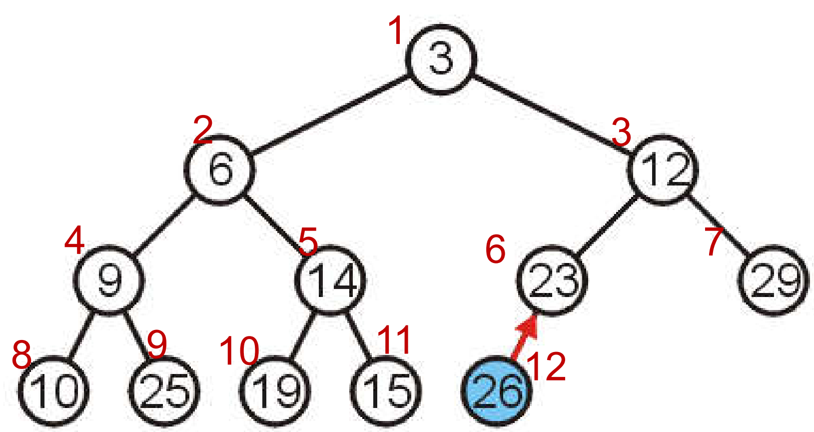
We cannot push everywhere

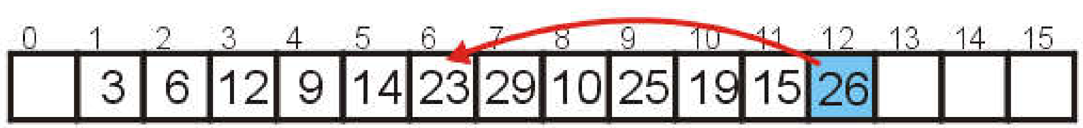
but only at the last position



Array Implementation: Push

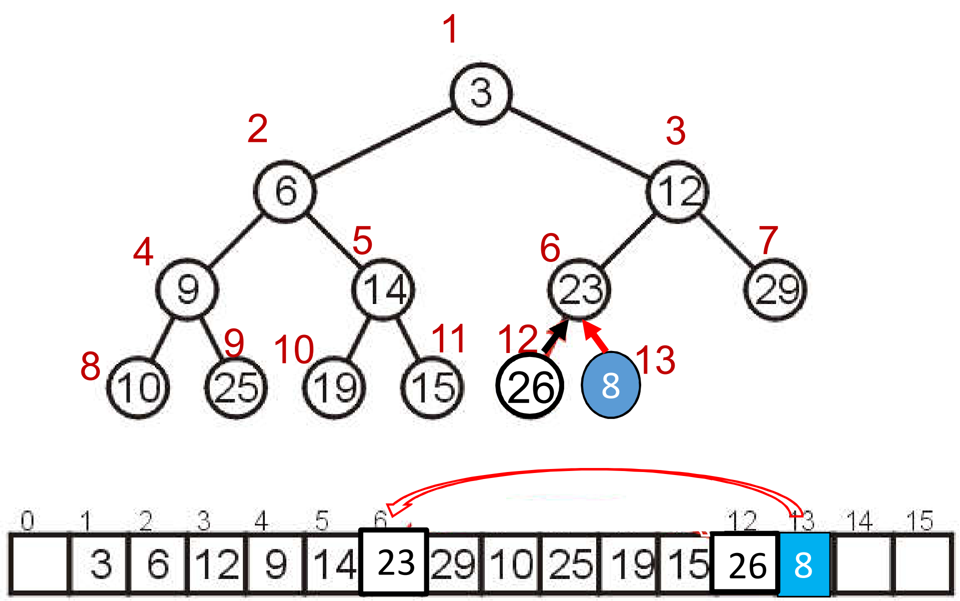
Inserting 26 requires no changes





Array Implementation: Push

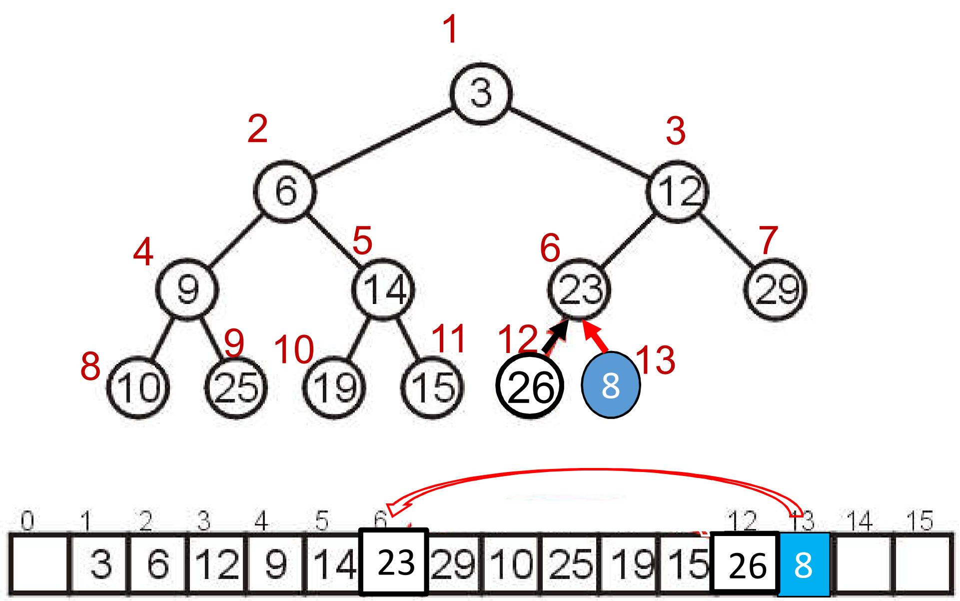
Now inserting 8 breaks heap property



Array Implementation: Push

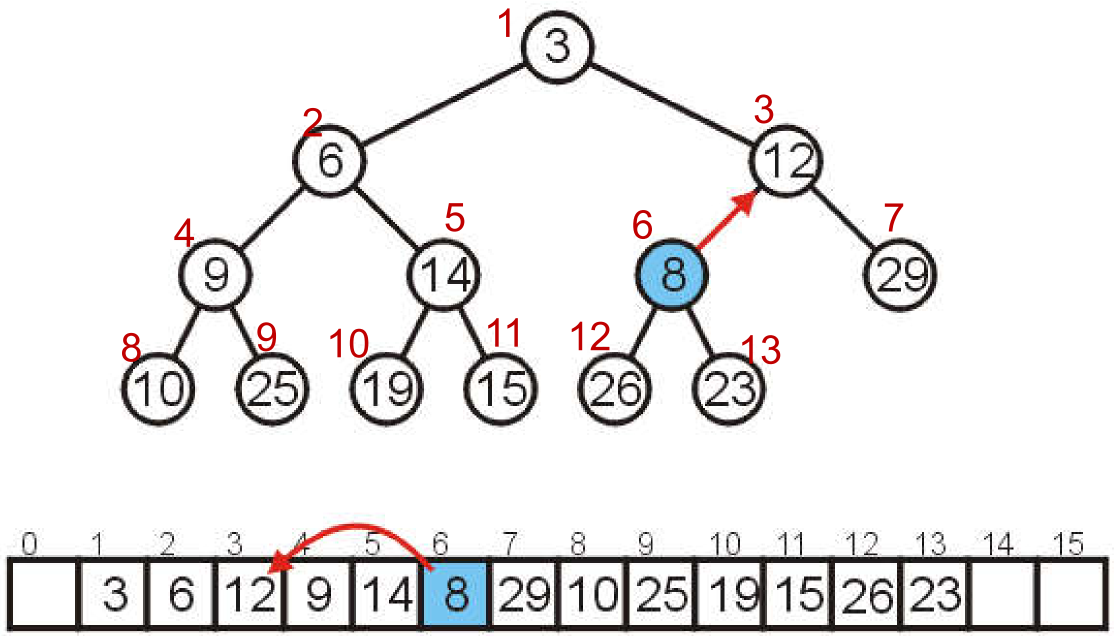
Inserting 8 requires a few **percolations**:

- Swap 8 and 23



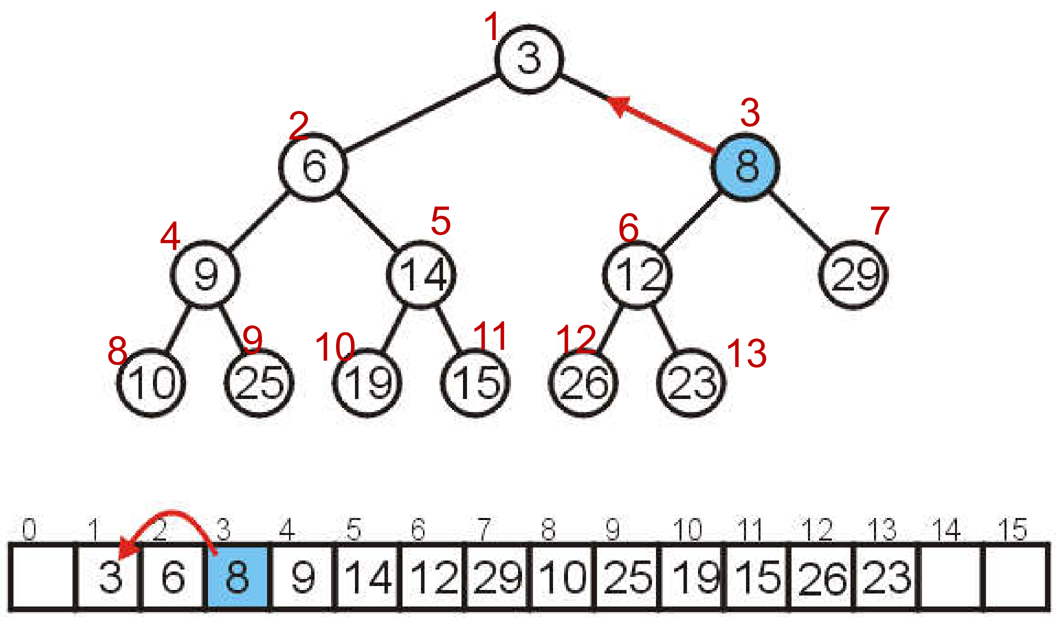
Array Implementation: Push

Swap 8 and 12



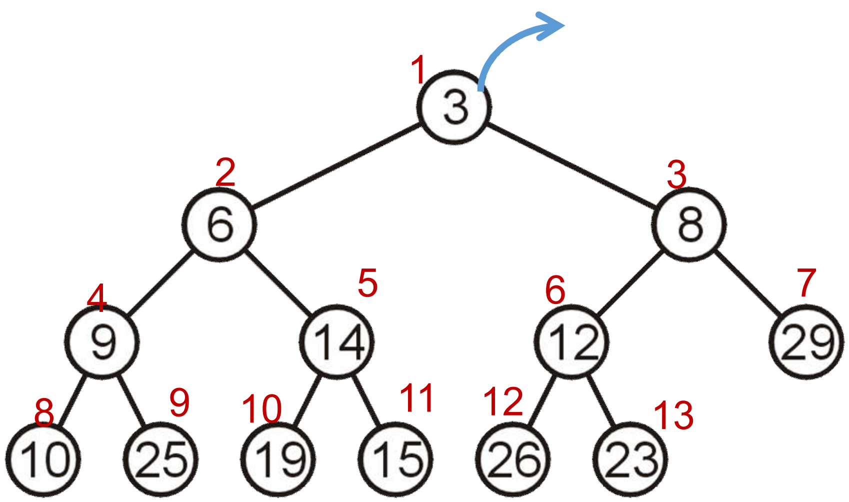
Array Implementation: Push

At this point, it is **greater than** its **parent**, so we are **finished**



Array Implementation: Pop/Extract_min

Suppose we want to pop the top entry: 3

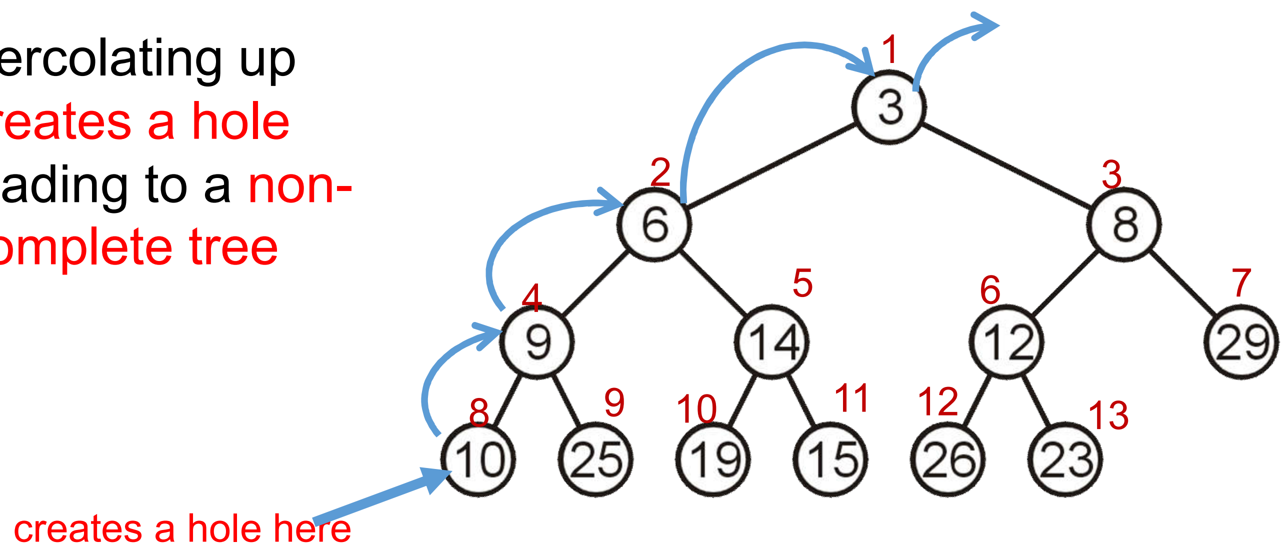


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	6	8	9	14	12	29	10	25	19	15	26	23		

Array Implementation: Pop/Extract_min

Suppose we want to pop the top entry: 3

Percolating up creates a hole leading to a non-complete tree

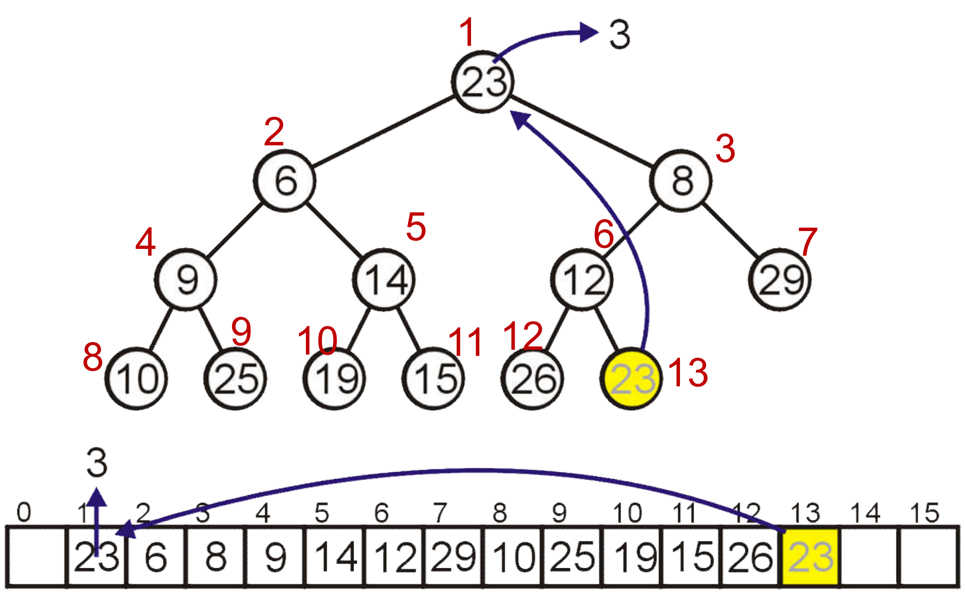


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	6	8	9	14	12	29	10	25	19	15	26	23		

Array Implementation: Pop

Instead, consider this strategy:

- Copy the last object, 23, to the root

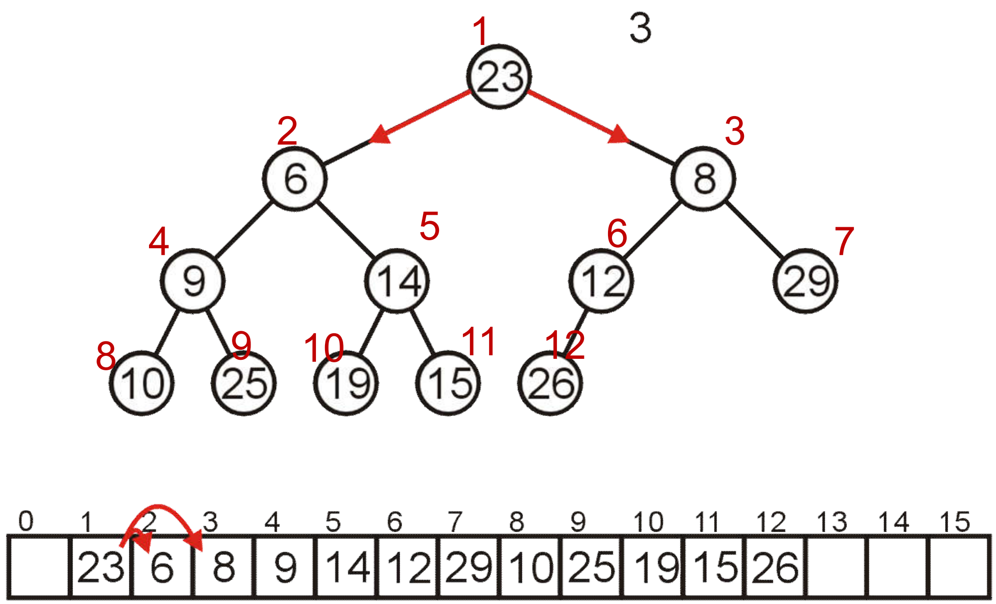


Array Implementation: Pop

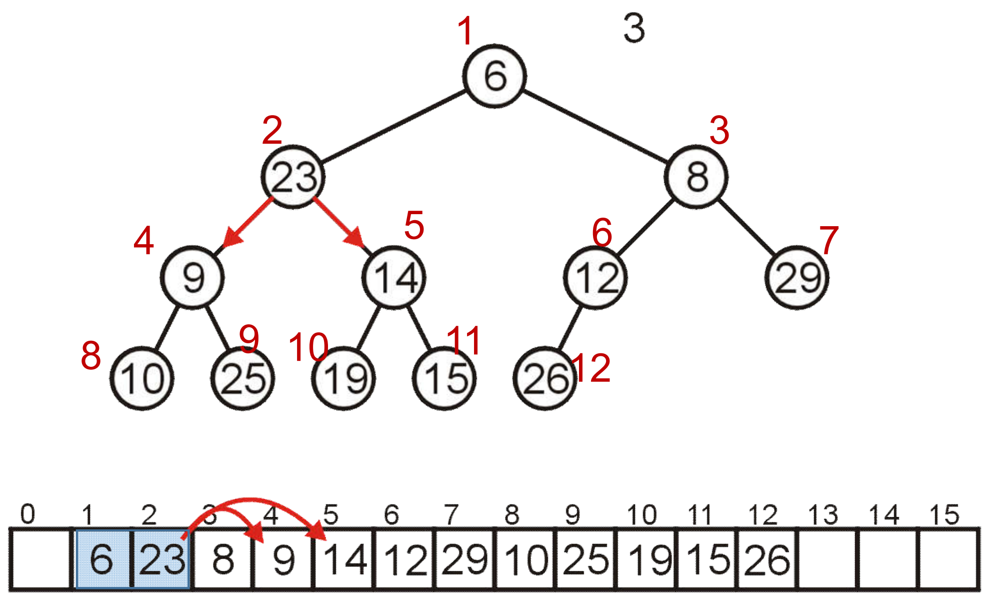
Now percolate down

Compare Node 1 with its children: Nodes 2 and 3

- Swap 23 and 6



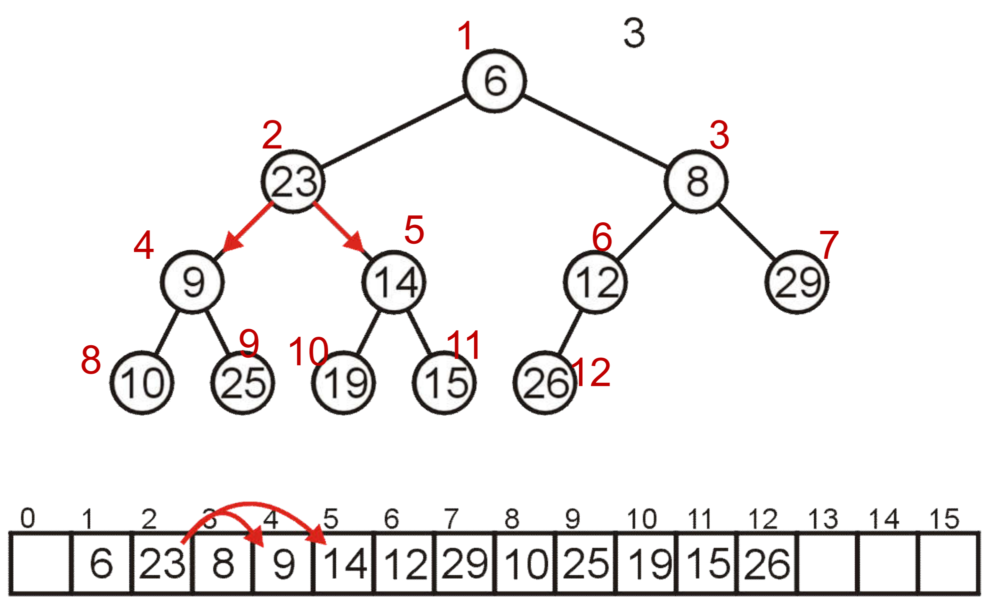
Array Implementation: Pop



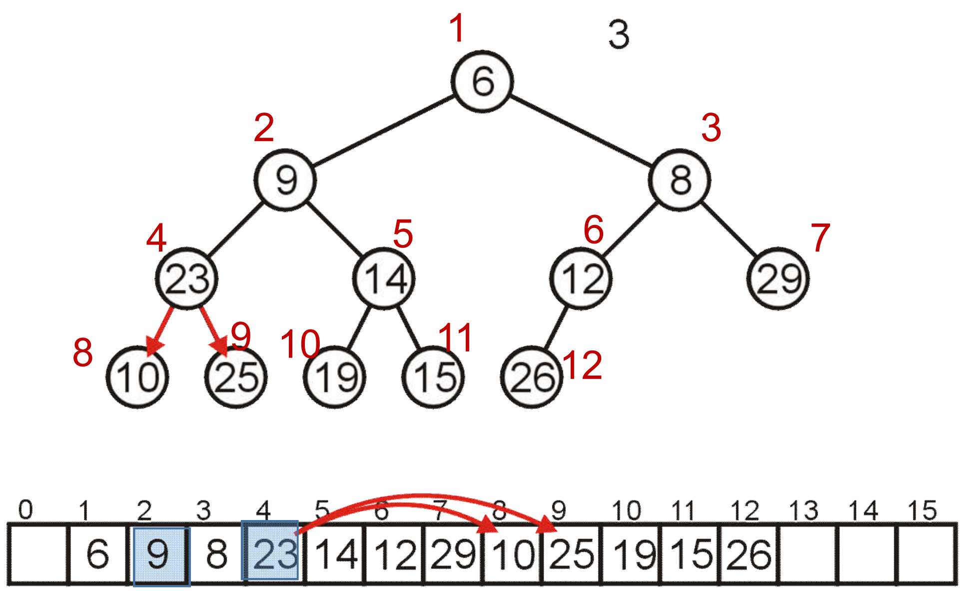
Array Implementation: Pop

Compare Node 2 with its children: Nodes 4 and 5

- Swap 23 and 9



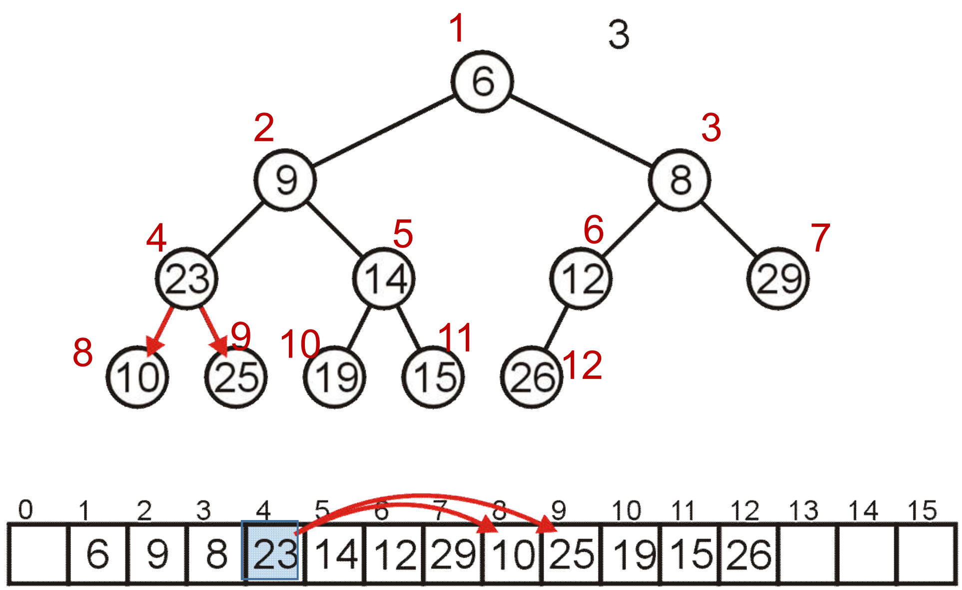
Array Implementation: Pop



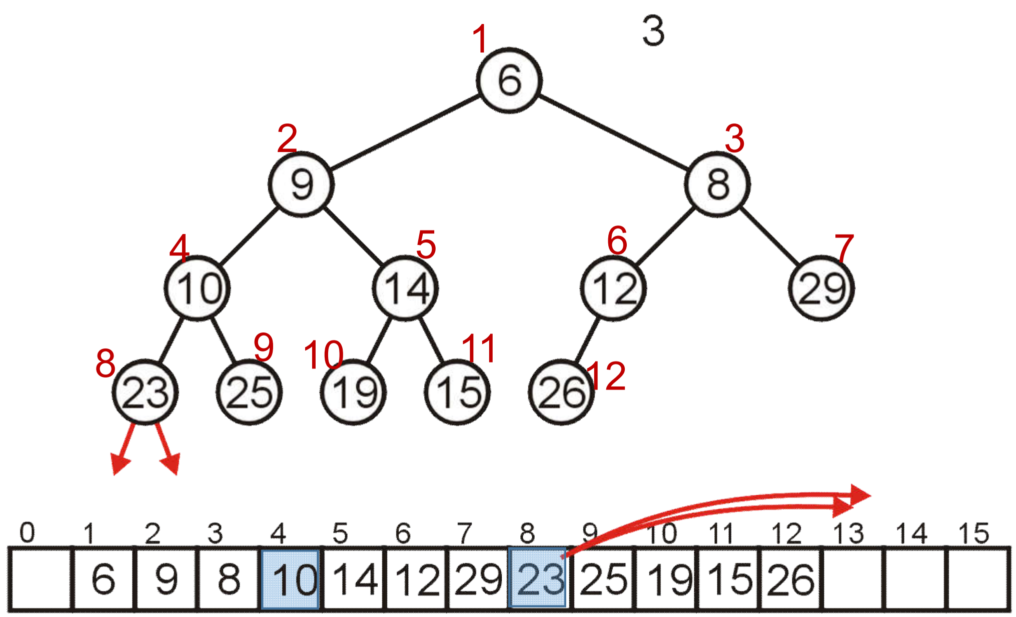
Array Implementation: Pop

Compare Node 4 with its children: Nodes 8 and 9

- Swap 23 and 10



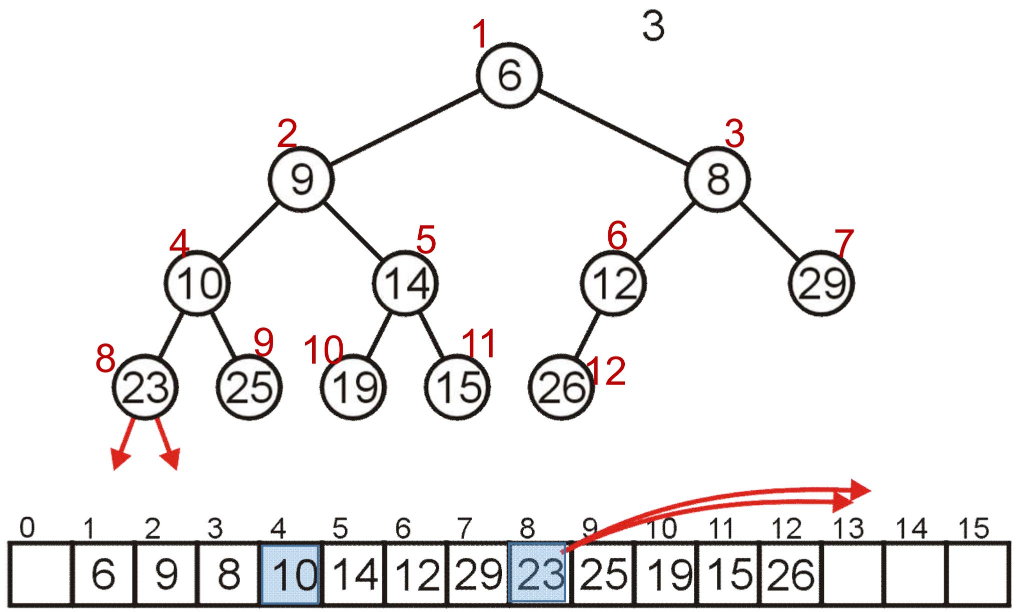
Array Implementation: Pop



Array Implementation: Pop

The children of Node 8 are beyond the end of the array:

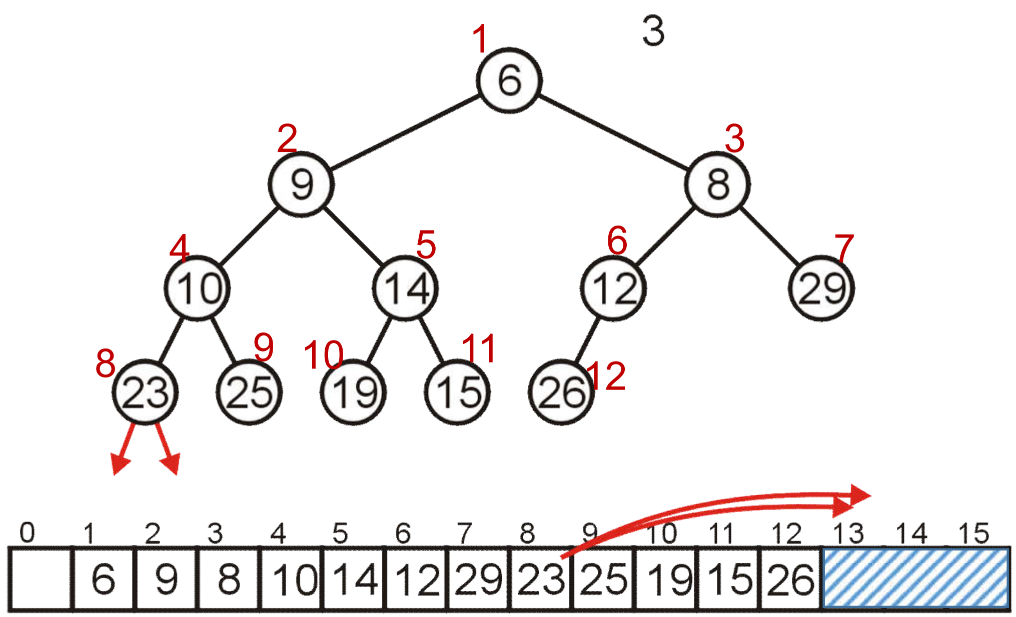
- Stop



Array Implementation: Pop

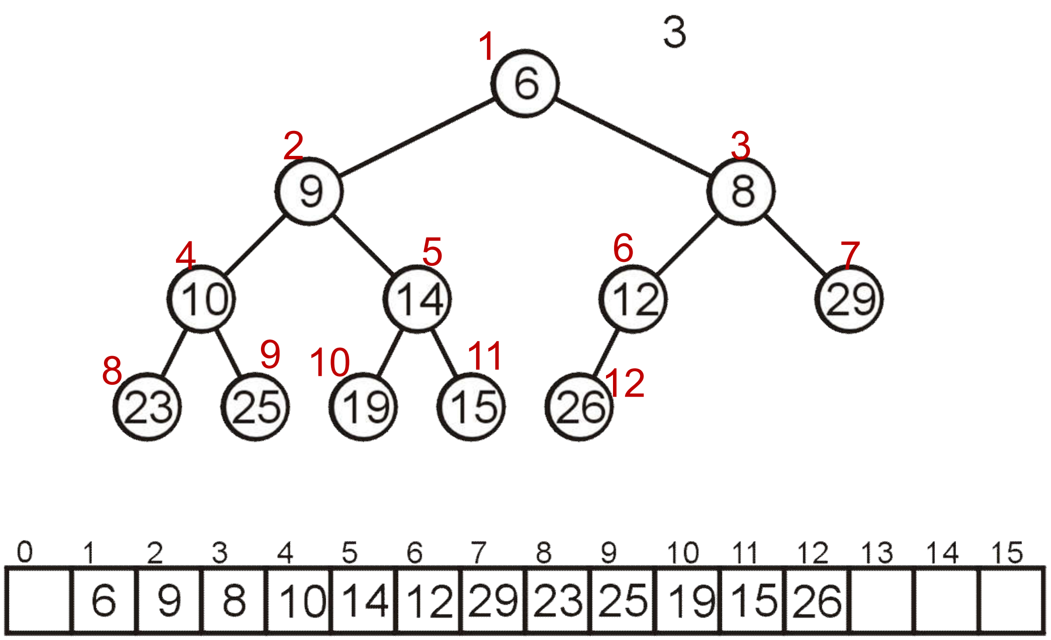
The children of Node 8 are beyond the end of the array:

- Stop



Array Implementation: Pop

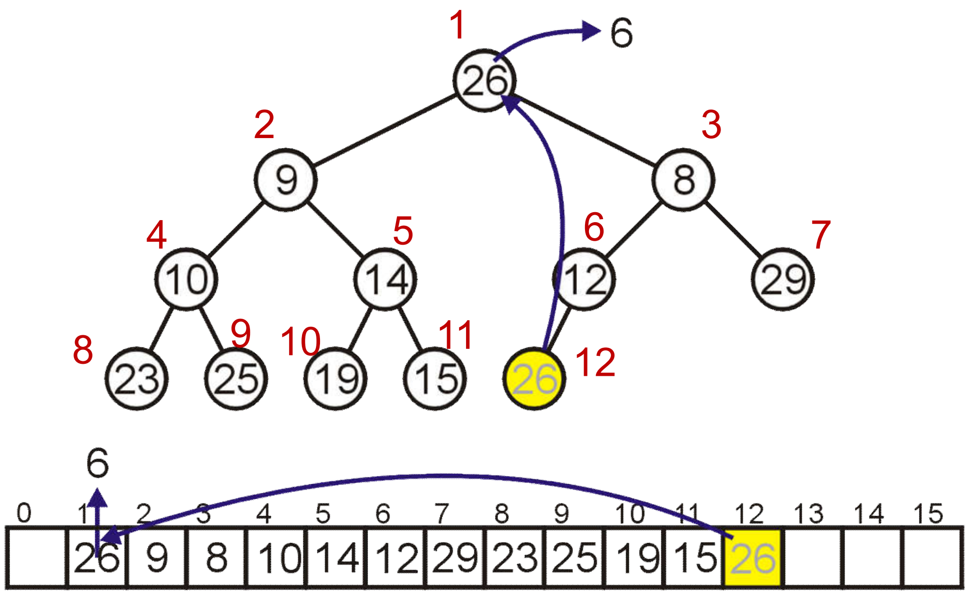
The result is a binary min-heap



Array Implementation: Pop

Dequeuing the minimum again:

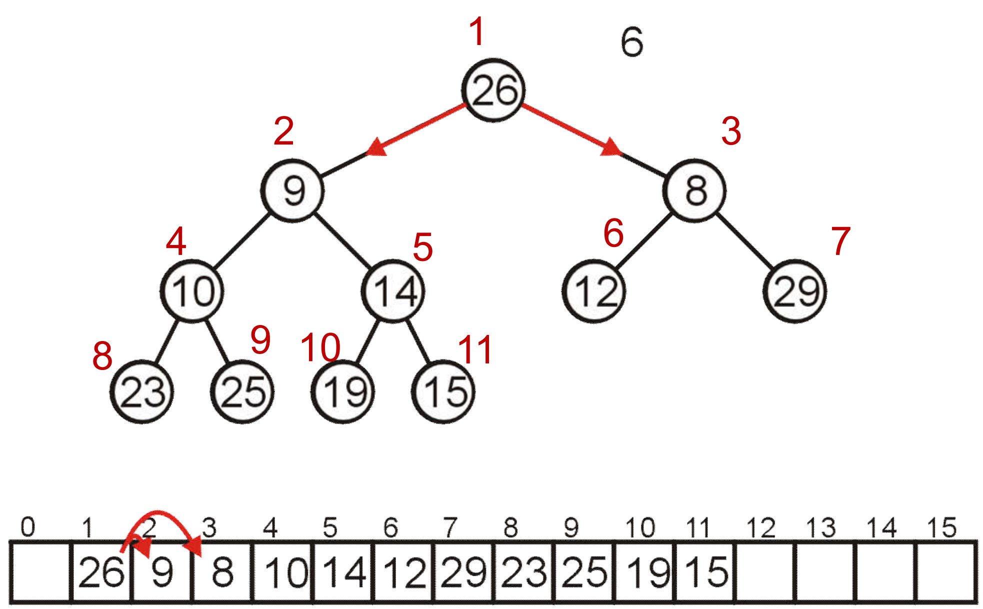
- Copy 26 to the root



Array Implementation: Pop

Compare Node 1 with its children: Nodes 2 and 3

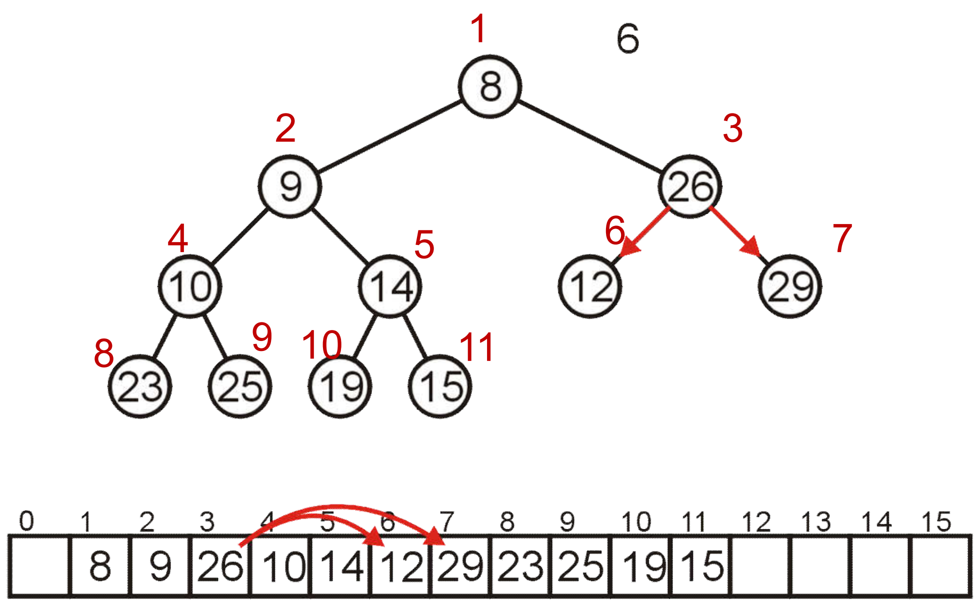
- Swap 26 and 8



Array Implementation: Pop

Compare Node 3 with its children: Nodes 6 and 7

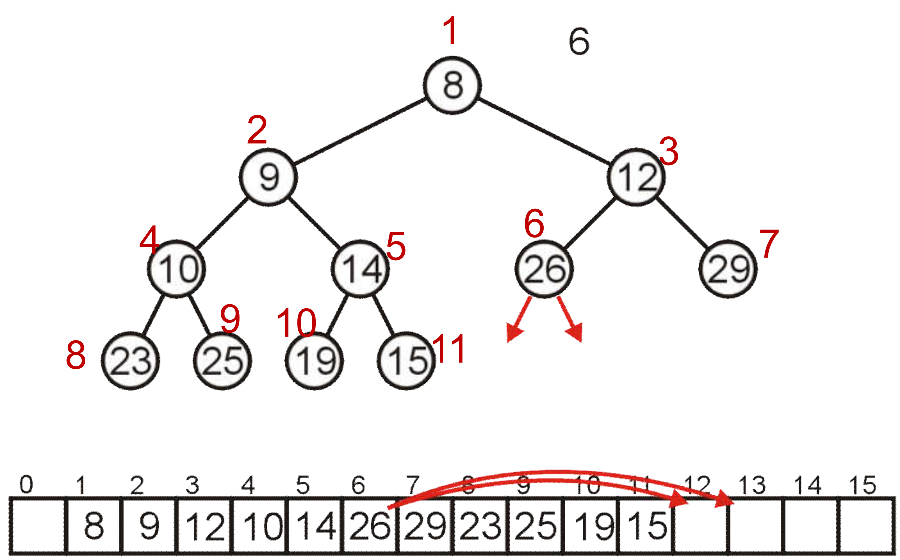
- Swap 26 and 12



Array Implementation: Pop

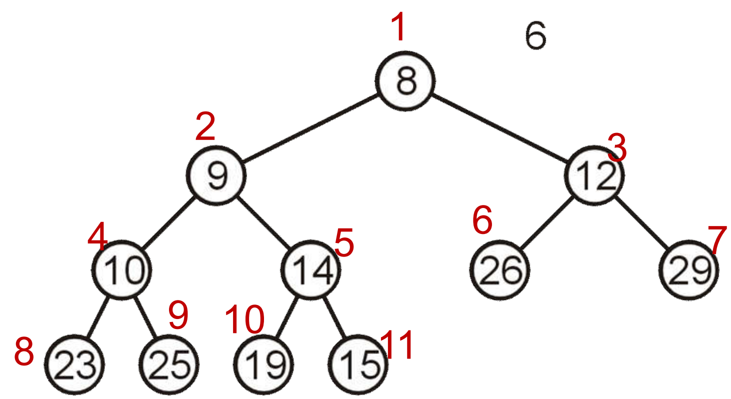
The children of Node 6, Nodes 12 and 13 are unoccupied

- Currently, `count == 11`



Array Implementation: Pop

The result is a min-heap

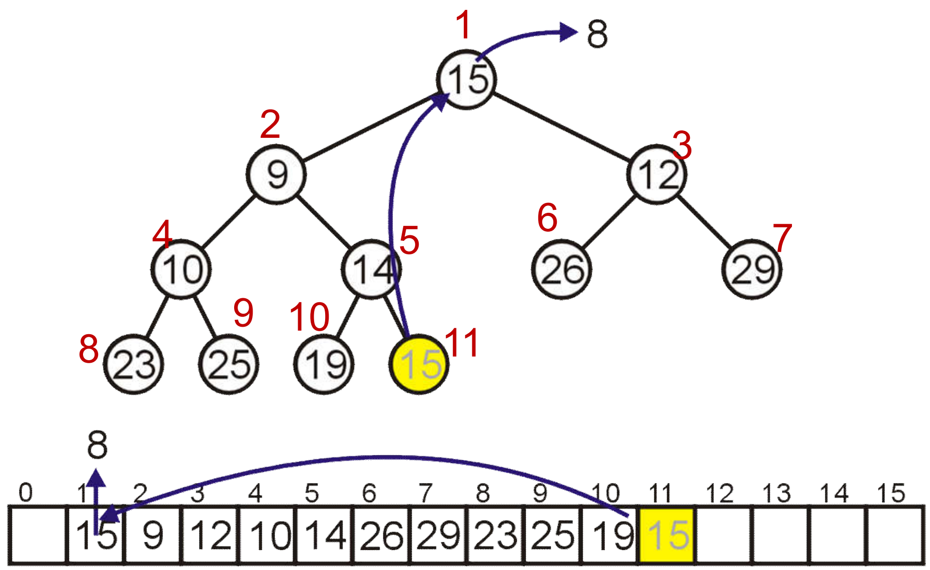


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	8	9	12	10	14	26	29	23	25	19	15				

Array Implementation: Pop

Dequeuing the minimum a third time:

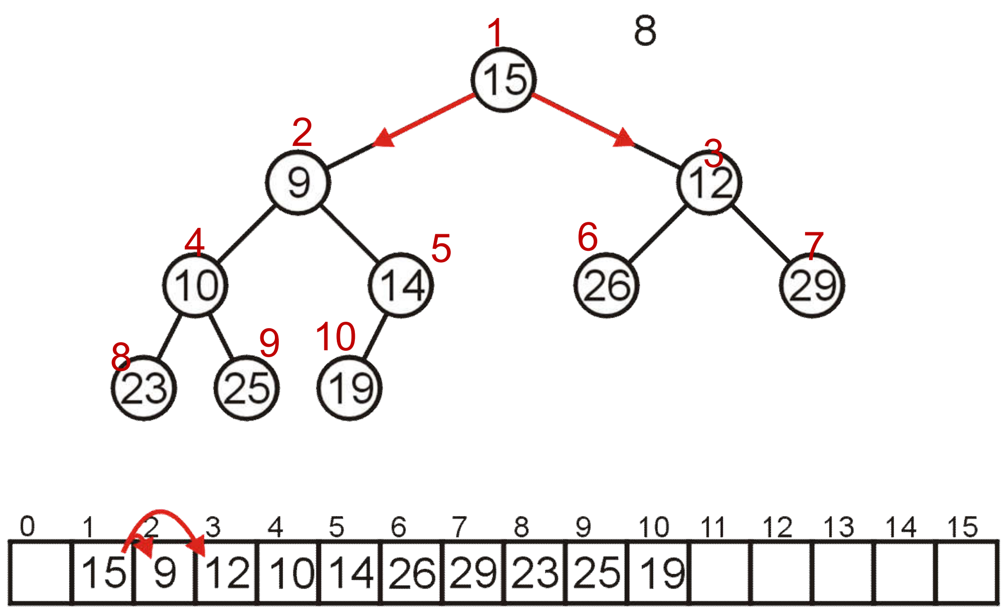
- Copy 15 to the root



Array Implementation: Pop

Compare Node 1 with its children: Nodes 2 and 3

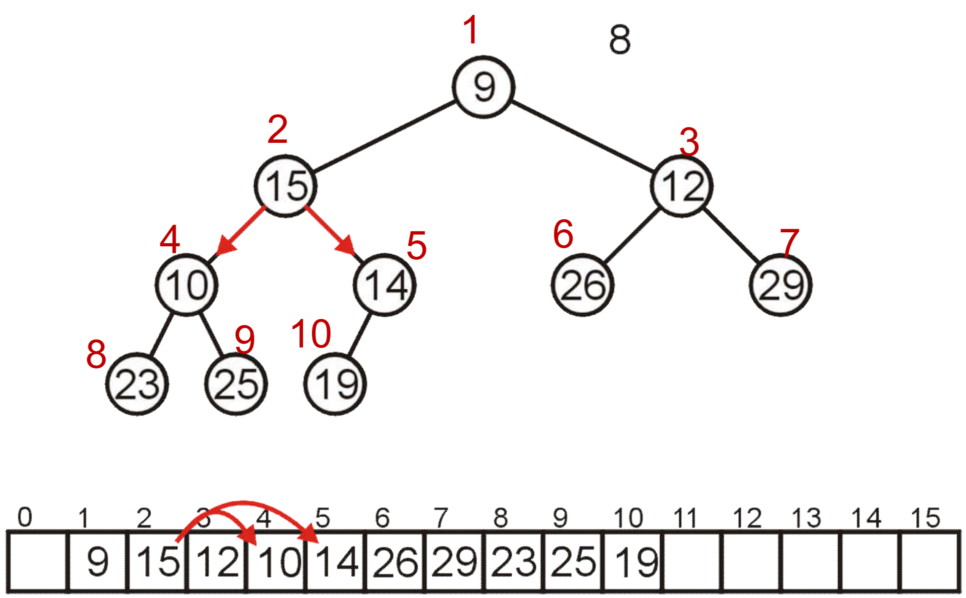
- Swap 15 and 9



Array Implementation: Pop

Compare Node 2 with its children: Nodes 4 and 5

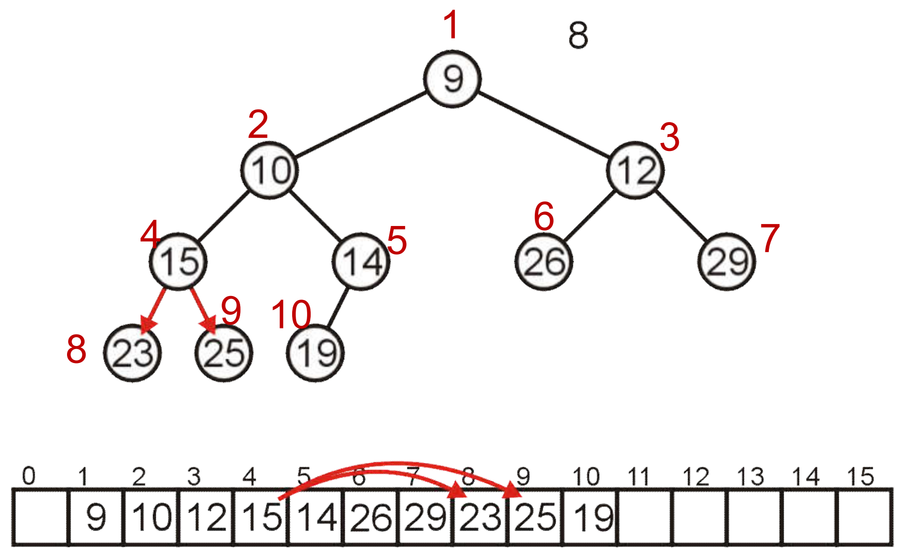
- Swap 15 and 10



Array Implementation: Pop

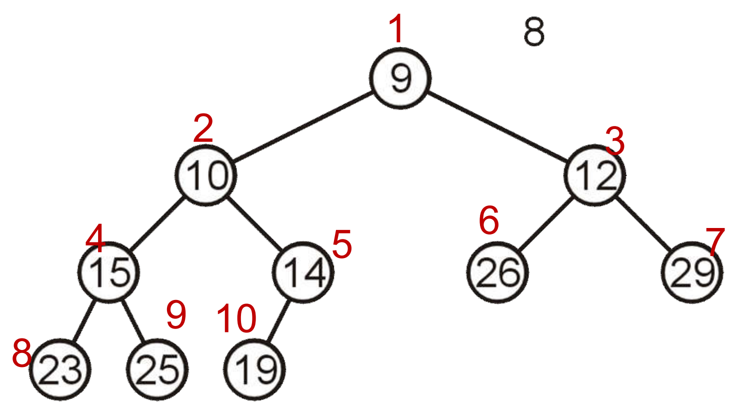
Compare Node 4 with its children: Nodes 8 and 9

- 15 < 23 and 15 < 25, so **stop**



Array Implementation: Pop

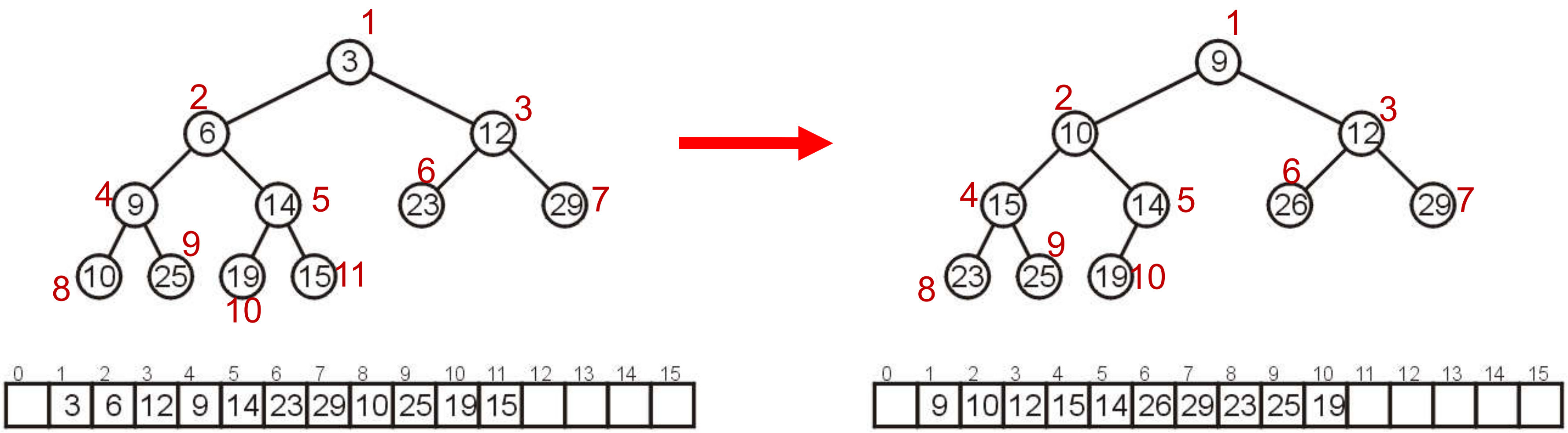
The result is a properly formed binary min-heap



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	9	10	12	15	14	26	29	23	25	19					

Array Implementation: Pop

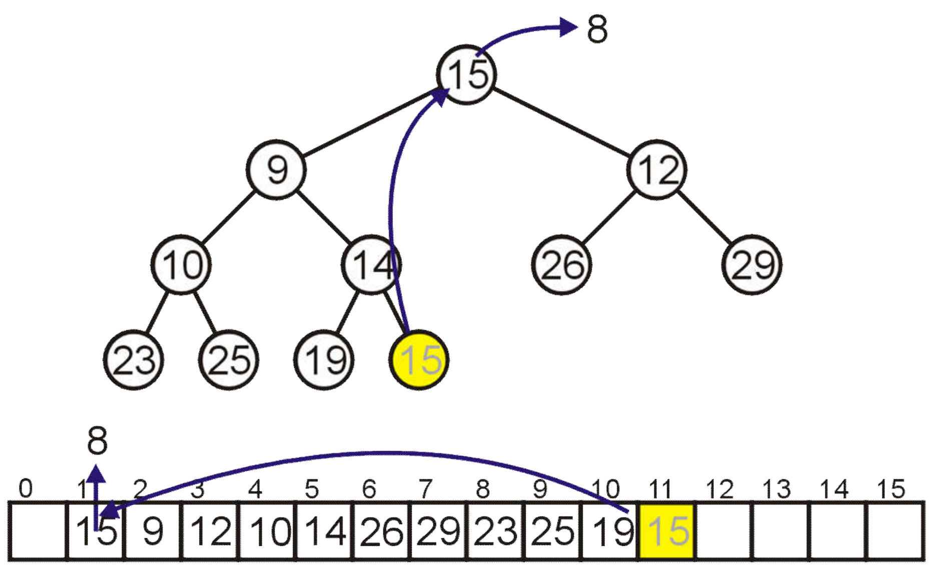
After all our modifications, the final heap is



Array Implementation: Pop

Dequeuing the minimum a third time: **Look at this example again**

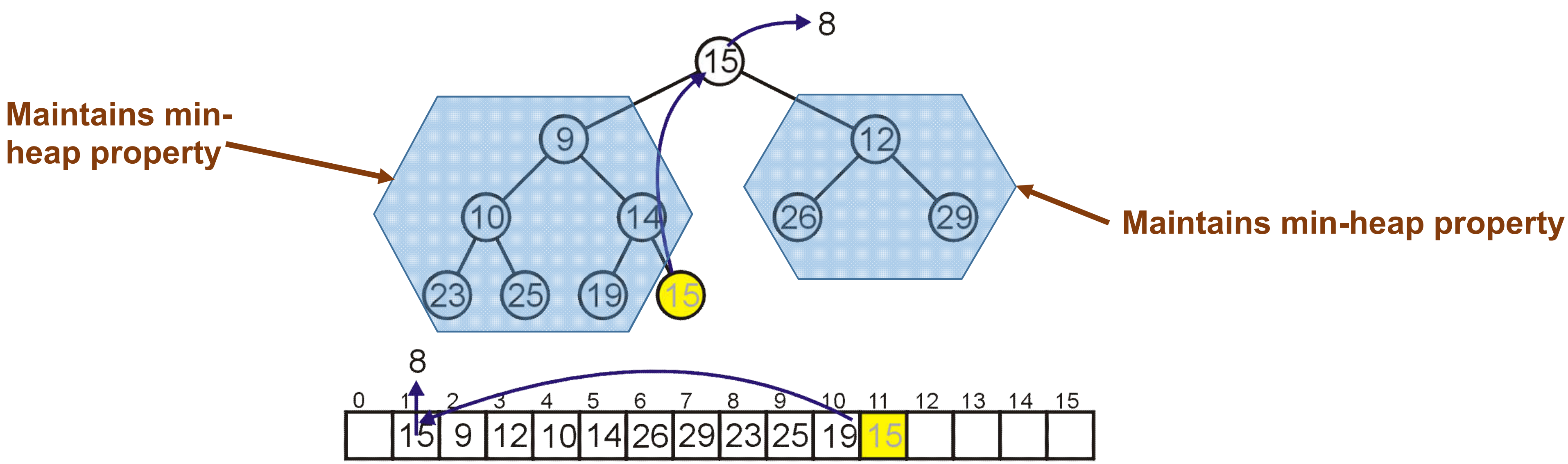
- Copy 15 to the root



Array Implementation: Pop

Dequeuing the minimum a third time: **Look at this example again**

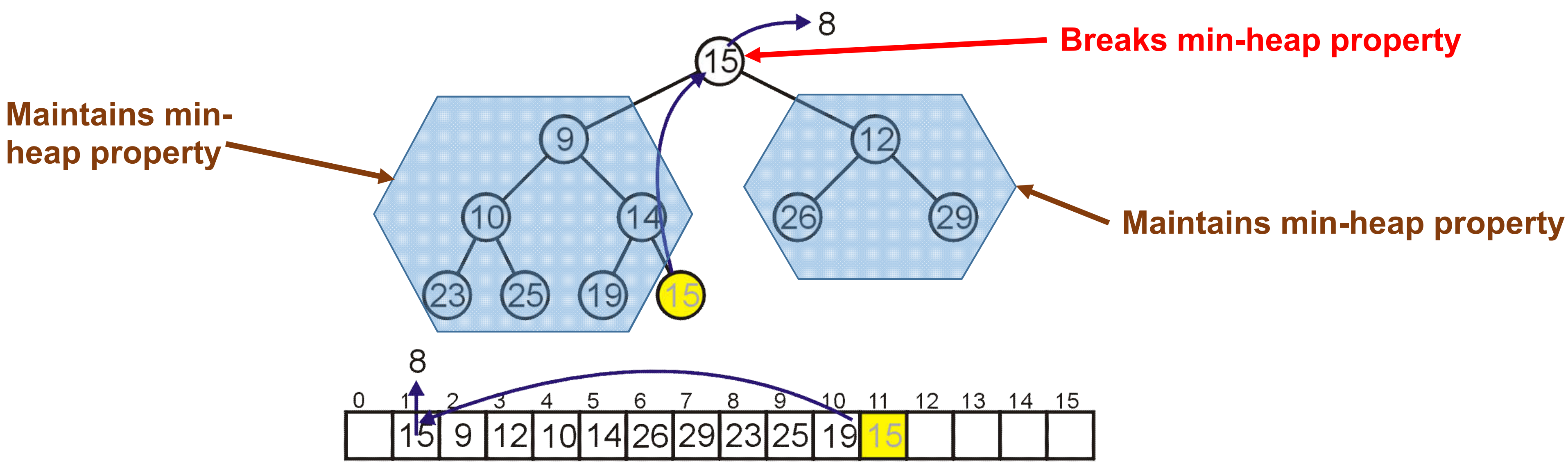
- Copy 15 to the root



Array Implementation: Pop

Dequeuing the minimum a third time: **Look at this example again**

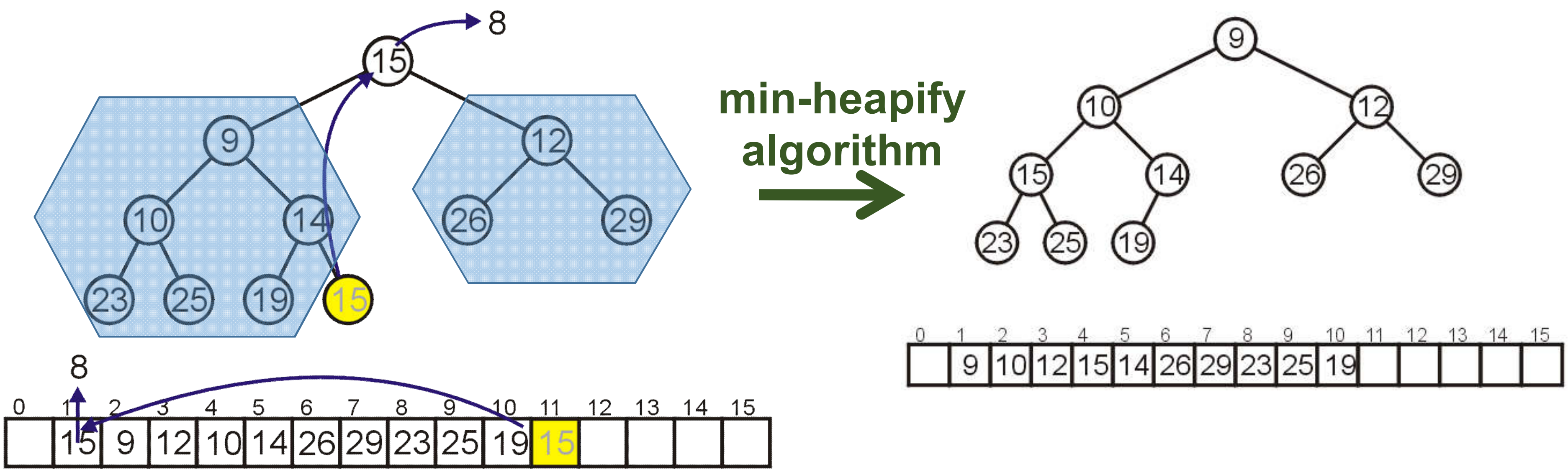
- Copy 15 to the root



Array Implementation: Pop

Dequeuing the minimum a third time: **Look at this example again**

- Copy 15 to the root



MIN-HEAPIFY

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

MIN-HEAPIFY

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

```
LEFT(i)
    return 2i

RIGHT(i)
    return 2i + 1
```

- Its inputs are an array A and an index i into the array.

MIN-HEAPIFY

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

```
LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```

- Its inputs are an array A and an index i into the array.
- When it is called, MIN-HEAPIFY **assumes**
  - the binary trees rooted at LEFT(i) and RIGHT(i) are min-heaps,
  - but that $A[i]$ might be larger than its children
    - thus violating the min-heap property.

MIN-HEAPIFY

MIN-HEAPIFY (A, i)

$l = \text{LEFT}(i)$
$r = \text{RIGHT}(i)$
if $l \leq A.\mathit{heap\text{-}size}$ and $A[l] < A[i]$
    $\mathit{smallest} = l$
else $\mathit{smallest} = i$
if $r \leq A.\mathit{heap\text{-}size}$ and $A[r] < A[\mathit{smallest}]$
    $\mathit{smallest} = r$
if $\mathit{smallest} \neq i$
    exchange $A[i]$ with $A[\mathit{smallest}]$
        MIN-HEAPIFY $(A, \mathit{smallest})$

LEFT(i)
return $2i$

RIGHT(i)
return $2i + 1$

- Its inputs are an array A and an index i into the array.
- When it is called, MIN-HEAPIFY **assumes**
  - the binary trees rooted at LEFT(i) and RIGHT(i) are min-heaps,
  - but that $A[i]$ might be larger than its children
    - thus violating the min-heap property.
- MIN-HEAPIFY lets the value at $A[i]$ "float down" in the min-heap so that the subtree rooted at index i obeys the min-heap property.

MIN-HEAPIFY

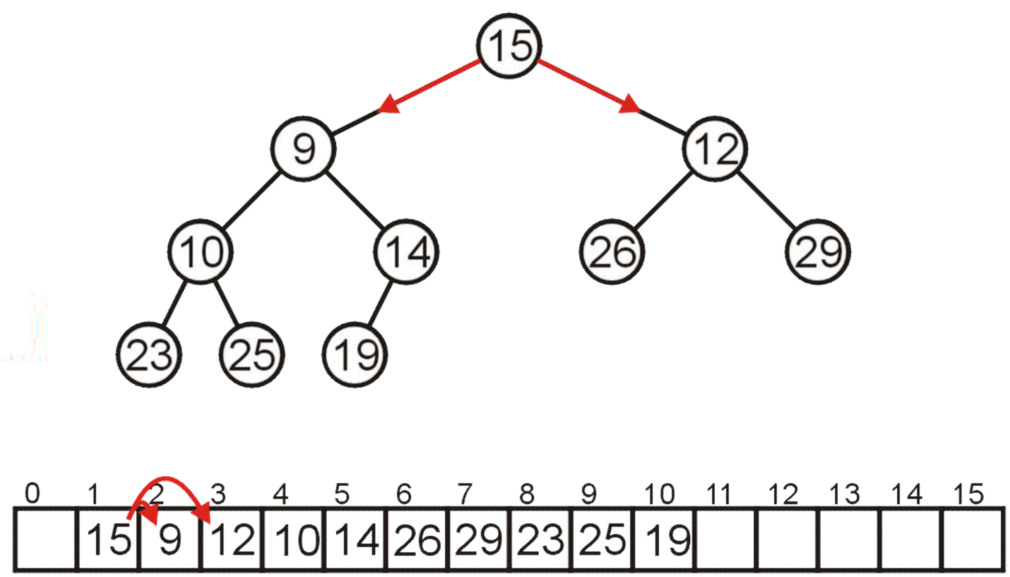
MIN-HEAPIFY (A, i)

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

```
LEFT(i)
    return 2i

RIGHT(i)
    return 2i + 1
```

Call MIN-HEAPIFY $(A, 1)$



MIN-HEAPIFY

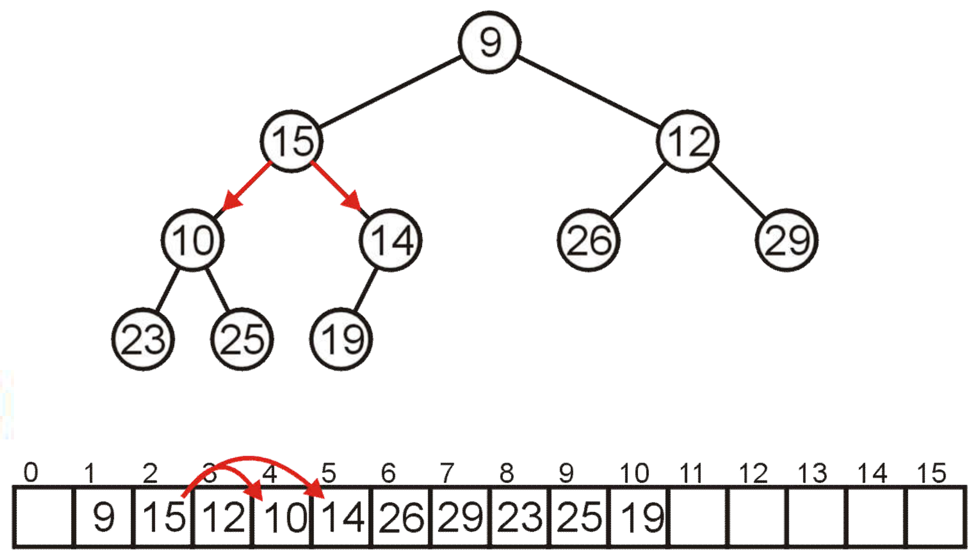
MIN-HEAPIFY (A, i)

l = LEFT(i)
r = RIGHT(i)
if $l \leq A.heap\text{-}size$ and $A[l] < A[i]$
    $smallest = l$
else $smallest = i$
if $r \leq A.heap\text{-}size$ and $A[r] < A[smallest]$
    $smallest = r$
if $smallest \neq i$
    exchange $A[i]$ with $A[smallest]$
        MIN-HEAPIFY $(A, smallest)$

LEFT(i)
1 **return** $2i$

RIGHT(i)
1 **return** $2i + 1$

Recursive Call
MIN-HEAPIFY $(A, 2)$



MIN-HEAPIFY

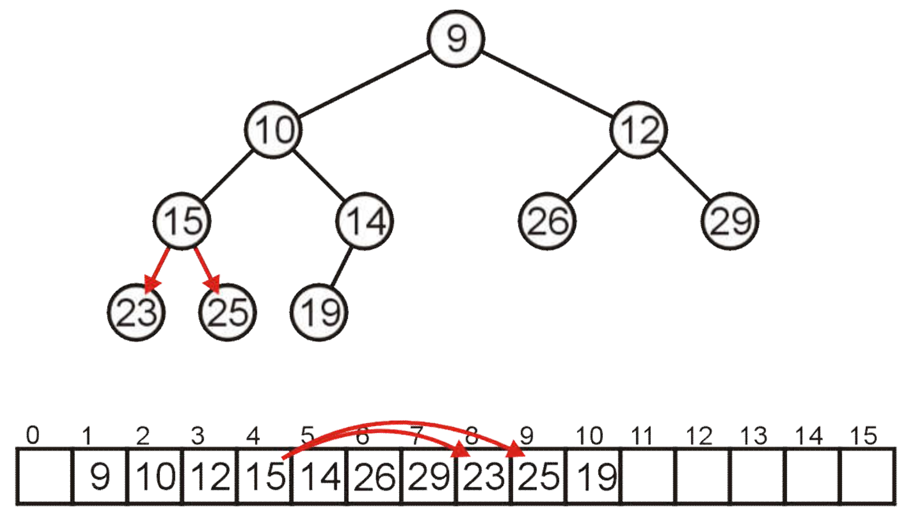
```
MIN-HEAPIFY (A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] < A[i]
    smallest = l
else smallest = i
if r ≤ A.heap-size and A[r] < A[smallest]
    smallest = r
if smallest ≠ i
    exchange A[i] with A[smallest]
        MIN-HEAPIFY (A, smallest)
```

```
LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```

Recursive Call

MIN-HEAPIFY (A, 4)

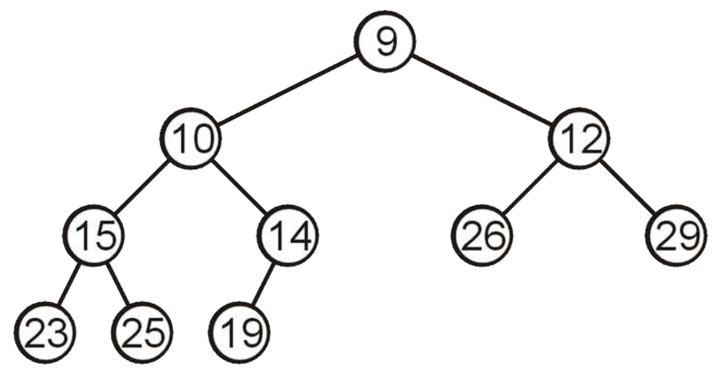


MIN-HEAPIFY

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

```
LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	9	10	12	15	14	26	29	23	25	19					

MAX-HEAPIFY

```
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)
```

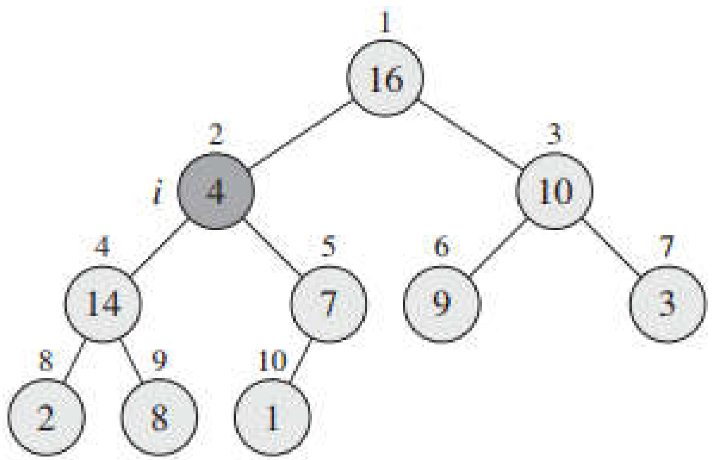
```
LEFT(i)
    return 2i

RIGHT(i)
    return 2i + 1
```

- Its inputs are an array *A* and an index *i* into the array.
- When it is called, MAX-HEAPIFY assumes
  - the binary trees rooted at LEFT(*i*) and RIGHT(*i*) are max-heaps,
  - but that *A*[*i*] might be smaller than its children
    - thus violating the max-heap property.
- MAX-HEAPIFY lets the value at *A*[*i*] "float down" in the max-heap so that the subtree rooted at index *i* obeys the max-heap property.

MAX-HEAPIFY(A,2)

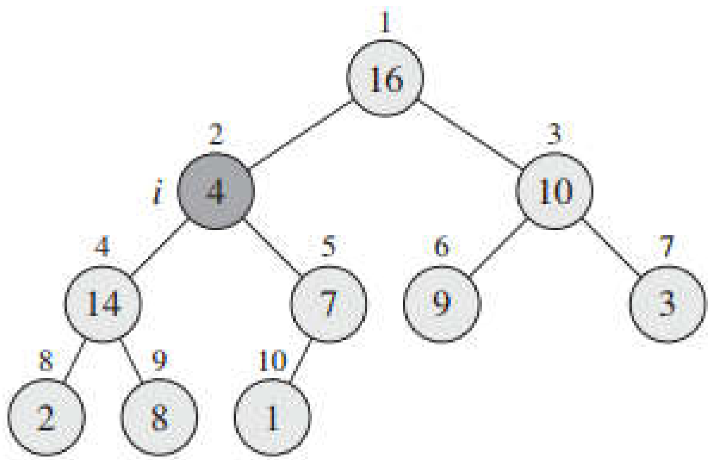
1	2	3	4	5	6	7	8	9	10
16	4	10	14	7	9	3	2	8	1

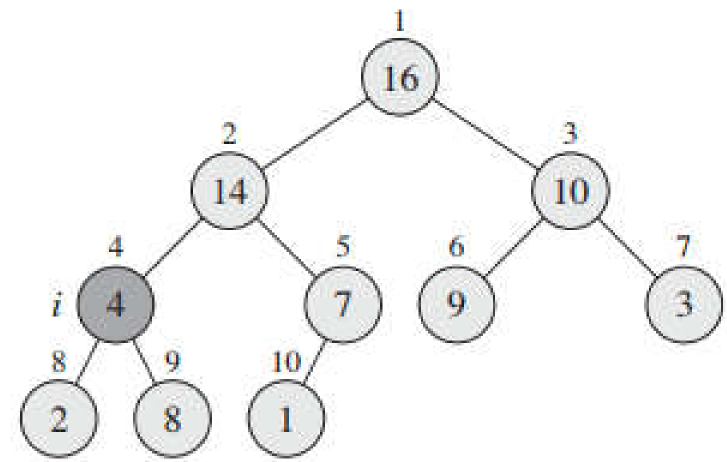



```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

MAX-HEAPIFY(A,2)

1	2	3	4	5	6	7	8	9	10
16	14	10	4	7	9	3	2	8	1

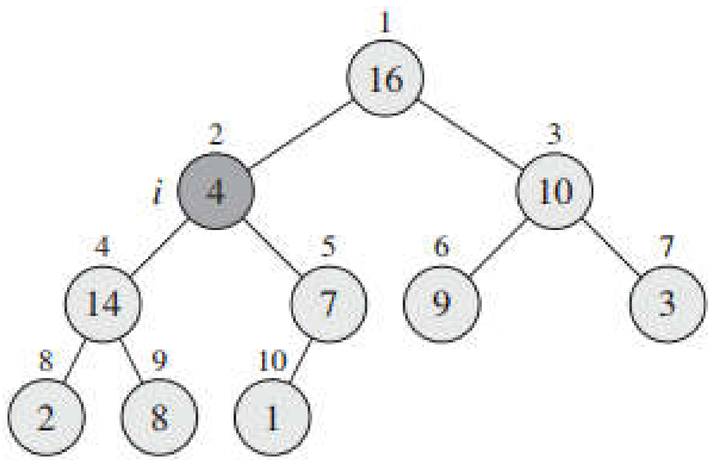


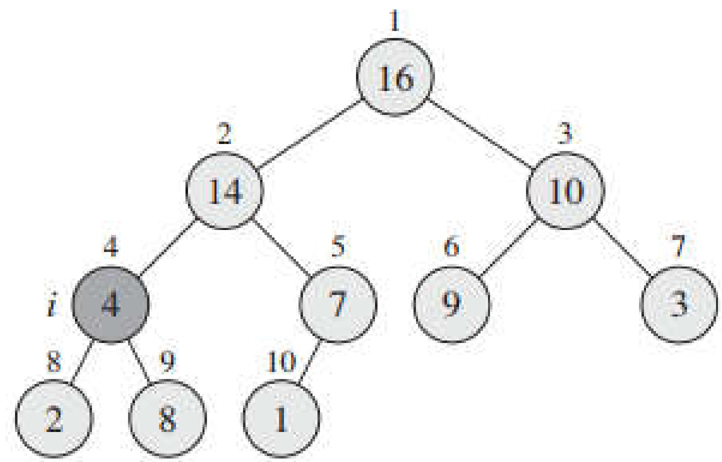



```
MAX-HEAPIFY(A, i)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ A.heap-size and A[l] > A[i]
      largest = l
  else largest = i
  if r ≤ A.heap-size and A[r] > A[largest]
      largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest)
```

MAX-HEAPIFY(*A*, 4)

1	2	3	4	5	6	7	8	9	10
16	14	10	4	7	9	3	2	8	1



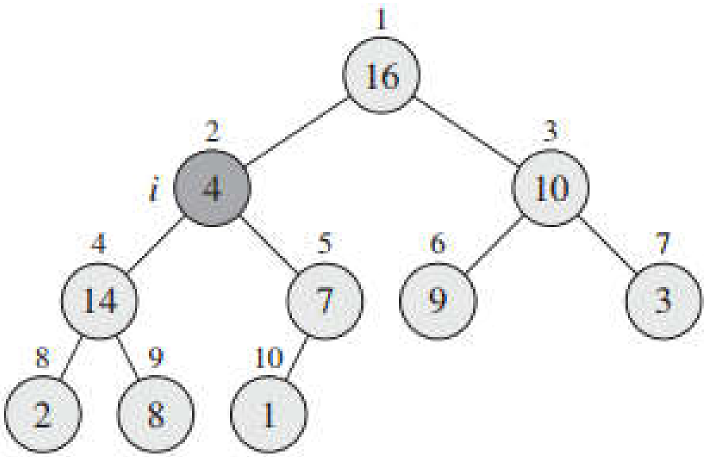


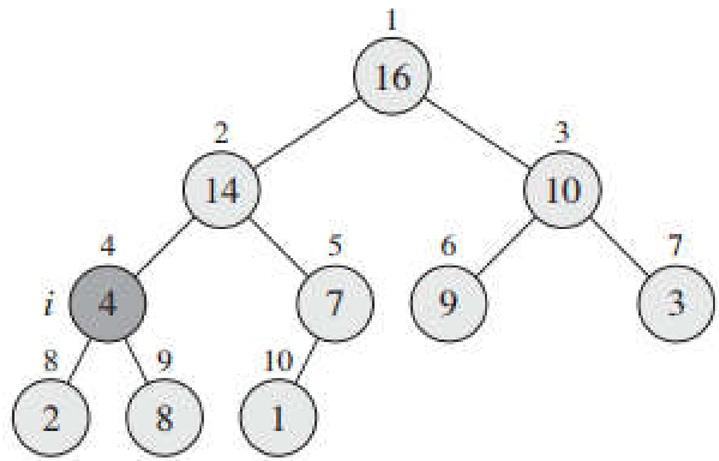
MAX-HEAPIFY(A, i)

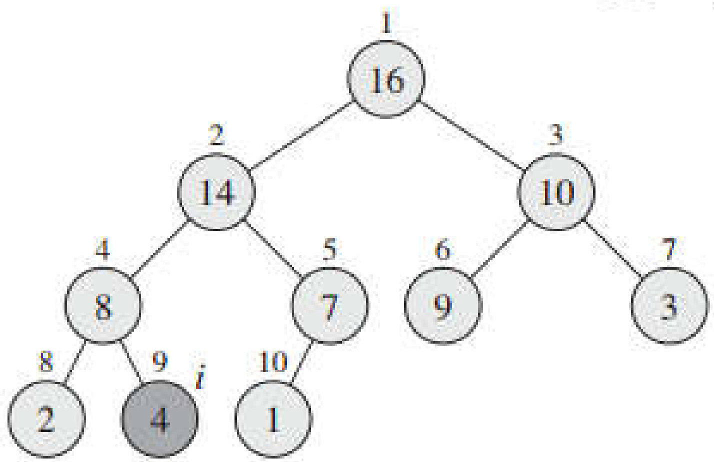
```
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

MAX-HEAPIFY(A, 4)

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1





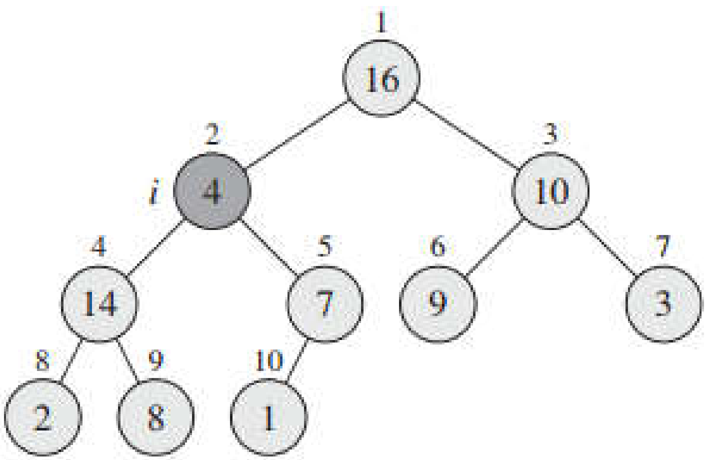


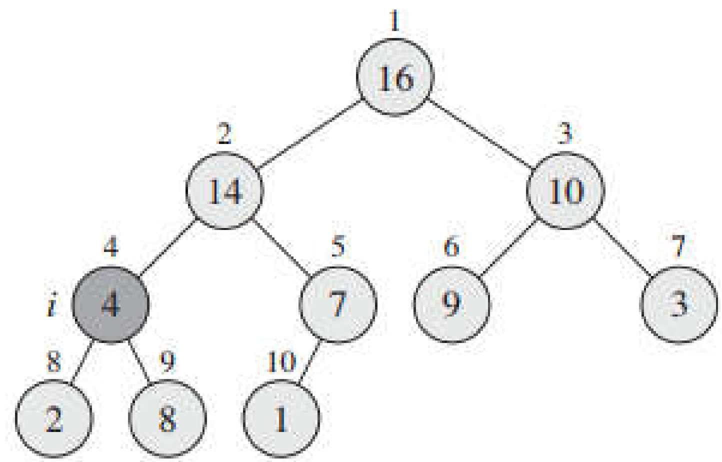
MAX-HEAPIFY(A, i)

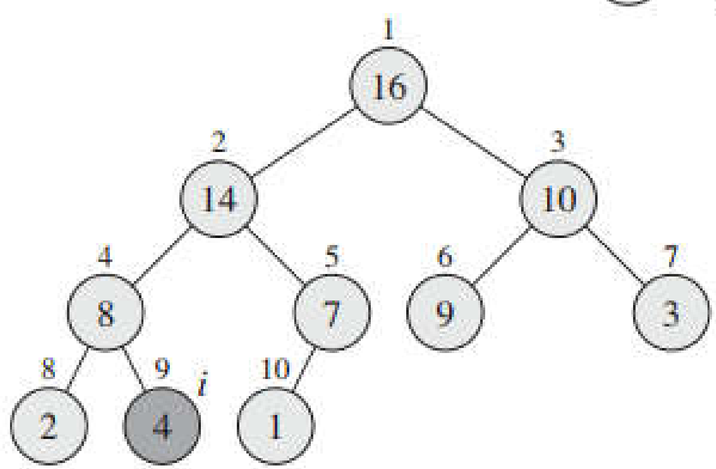
$l = \text{LEFT}(i)$
$r = \text{RIGHT}(i)$
if $l \le A.\mathit{heap\text{-}size}$ and $A[l] > A[i]$
    $\mathit{largest} = l$
else $\mathit{largest} = i$
if $r \le A.\mathit{heap\text{-}size}$ and $A[r] > A[\mathit{largest}]$
    $\mathit{largest} = r$
if $\mathit{largest} \ne i$
    exchange $A[i]$ with $A[\mathit{largest}]$
    MAX-HEAPIFY$(A, \mathit{largest})$

MAX-HEAPIFY(A, 9)

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1





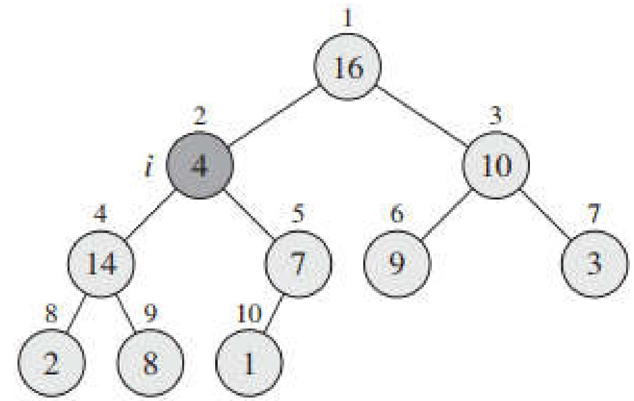



```
MAX-HEAPIFY(A, i)
   l = LEFT(i)
   r = RIGHT(i)
   if l ≤ A.heap-size and A[l] > A[i]
       largest = l
   else largest = i
   if r ≤ A.heap-size and A[r] > A[largest]
       largest = r
   if largest ≠ i
       exchange A[i] with A[largest]
       MAX-HEAPIFY(A, largest)
```

(MAX/MIN)-HEAPIFY: Running time

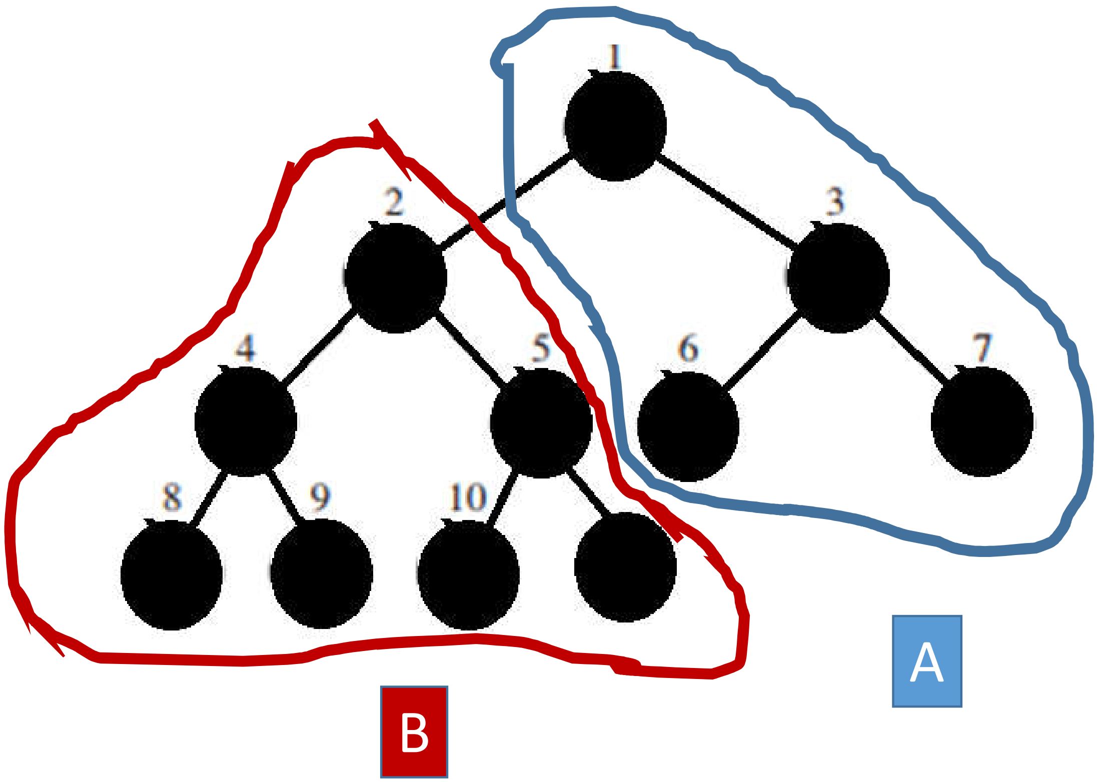
```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

$\Theta(1)$



What is the worst case scenario?

If MAX-HEAPIFY is called on an array/tree of *n* nodes how many nodes are there in the sub- tree/array on which the recursive call is made?



- The children's subtrees each have size at most 2n/3
  - A + B = n

  ⇒B ≈ 2A

  ⇒A ≈ n/3

  ⇒B ≈ 2n/3
- The worst case occurs when the bottom level of the tree is exactly half full

(MAX/MIN)-HEAPIFY: Running time

```
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)
```

$\Theta(1)$

$$T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1)$$

$\Rightarrow$ T(n) = O(log n)

Master Theorem