Priority Queues: Binary Heaps

Data Structures
Data Structures and Algorithms

Outline

1. Binary Trees
2. Basic Operations
3. Complete Binary Trees
4. Pseudocode
5. Heap Sort
6. Final Remarks

Definition

Binary max-heap is a binary tree (each node has zero, one, or two children) where the value of each node is at least the values of its children.

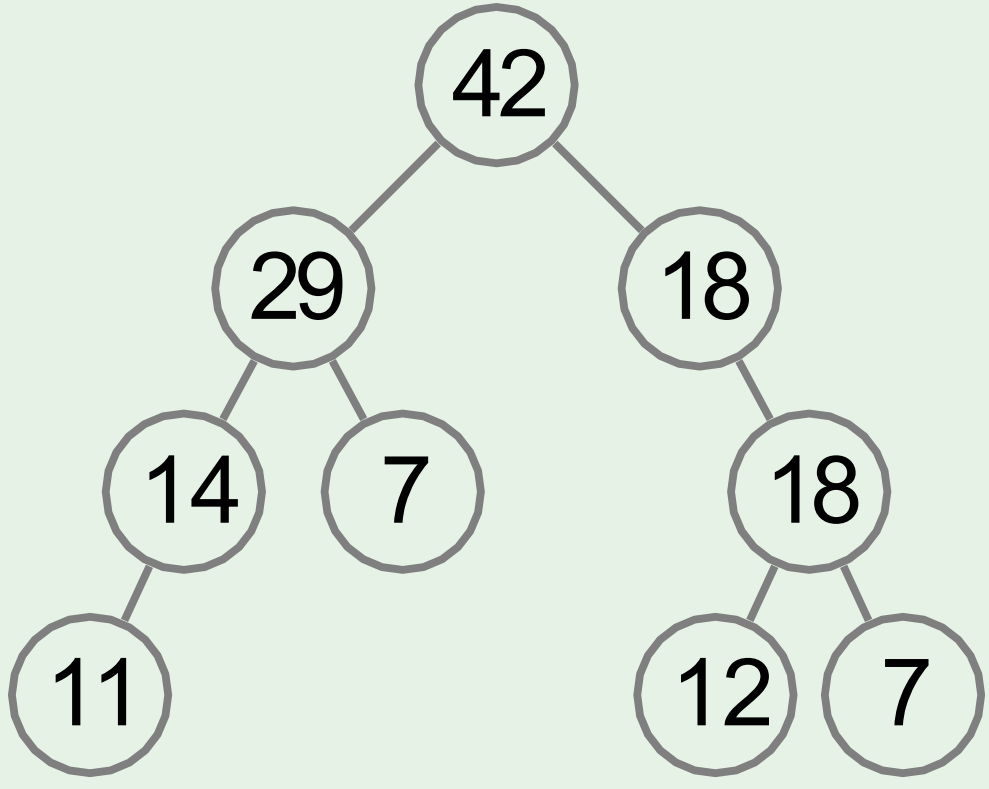
Definition

Binary max-heap is a binary tree (each node has zero, one, or two children) where the value of each node is at least the values of its children.

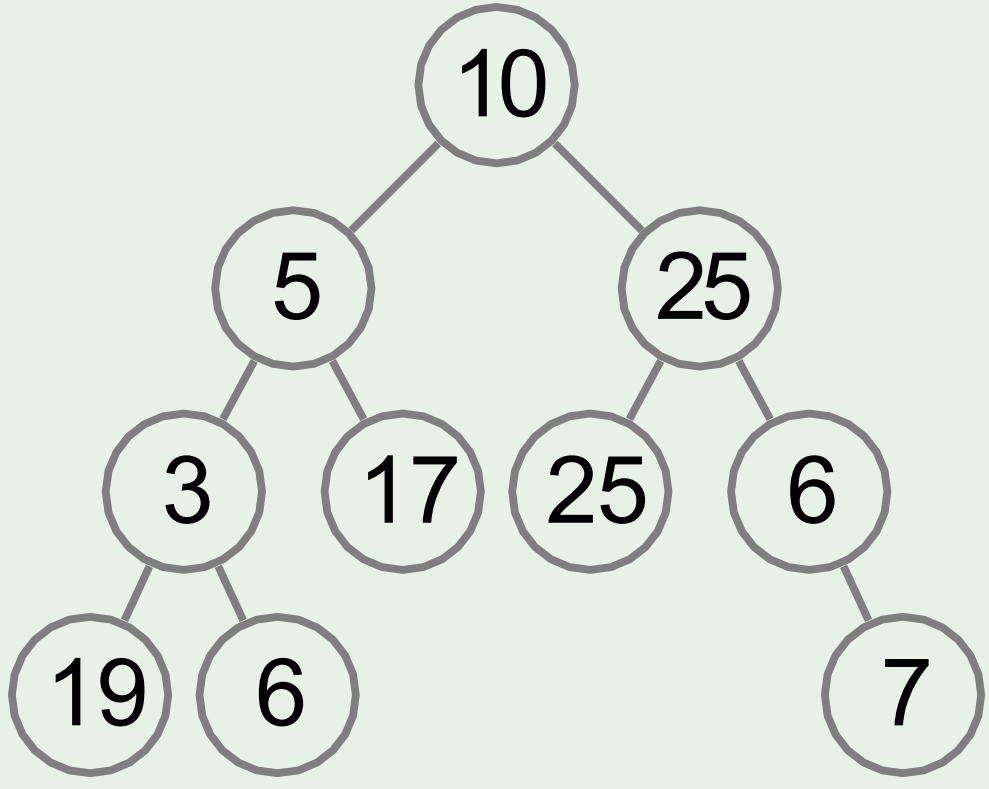
In other words

For each edge of the tree, the value of the parent is at least the value of the child.

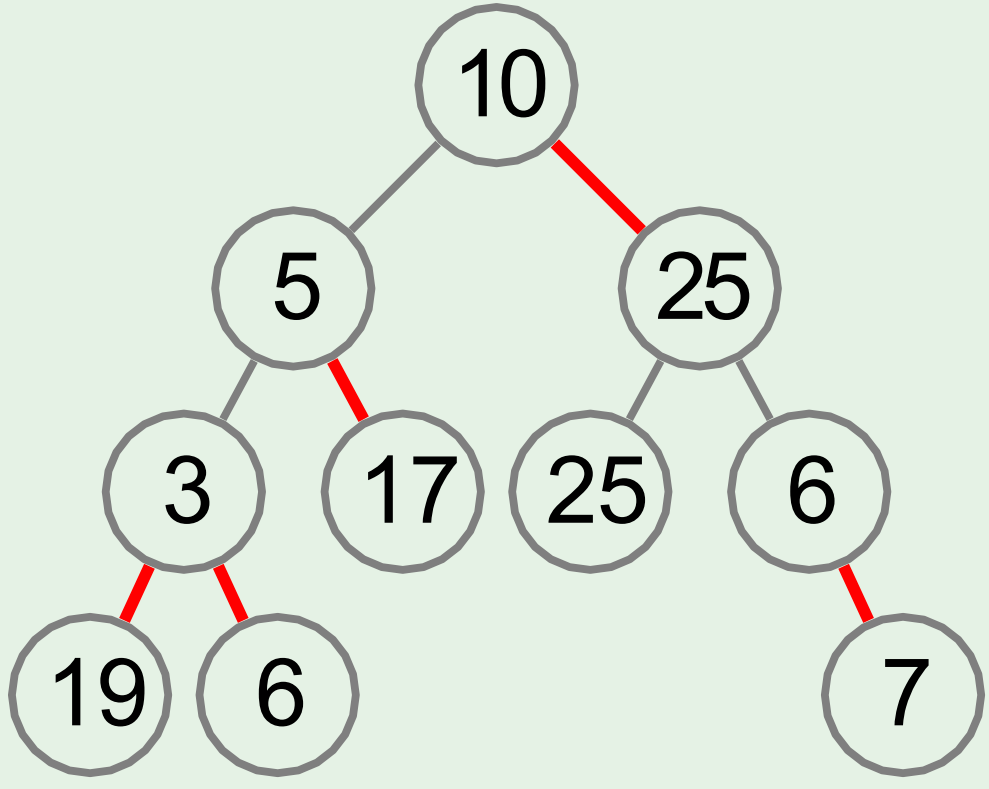
Example: heap



Example: not a heap

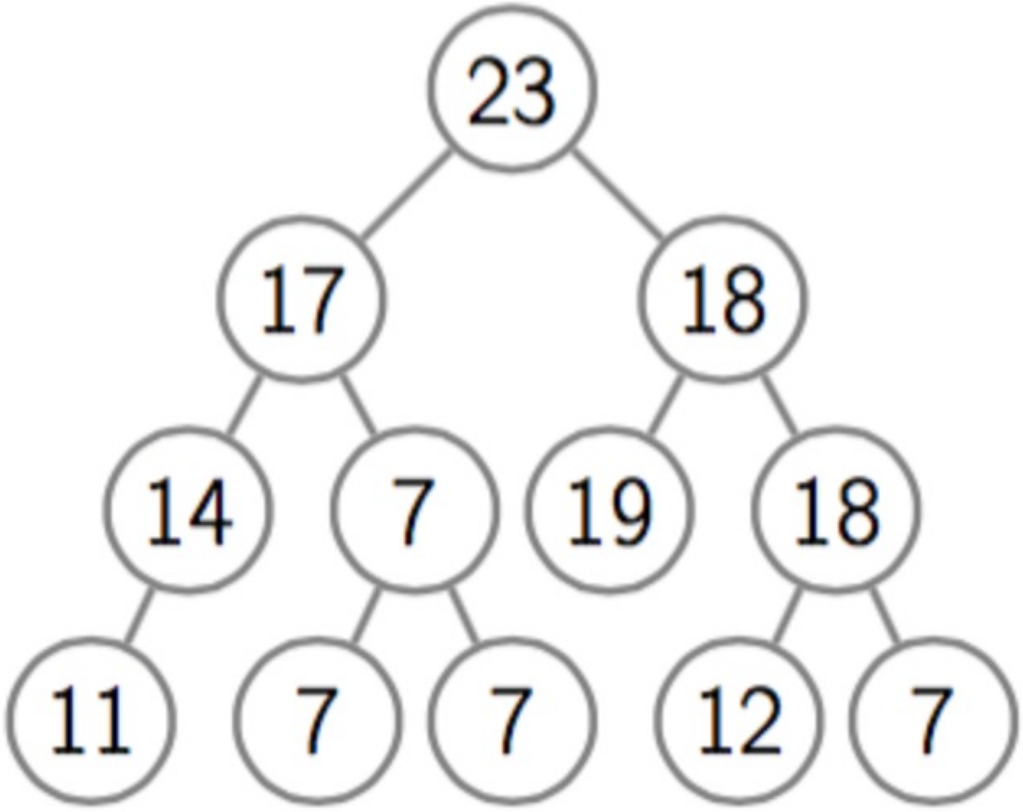


Example: not a heap



Question

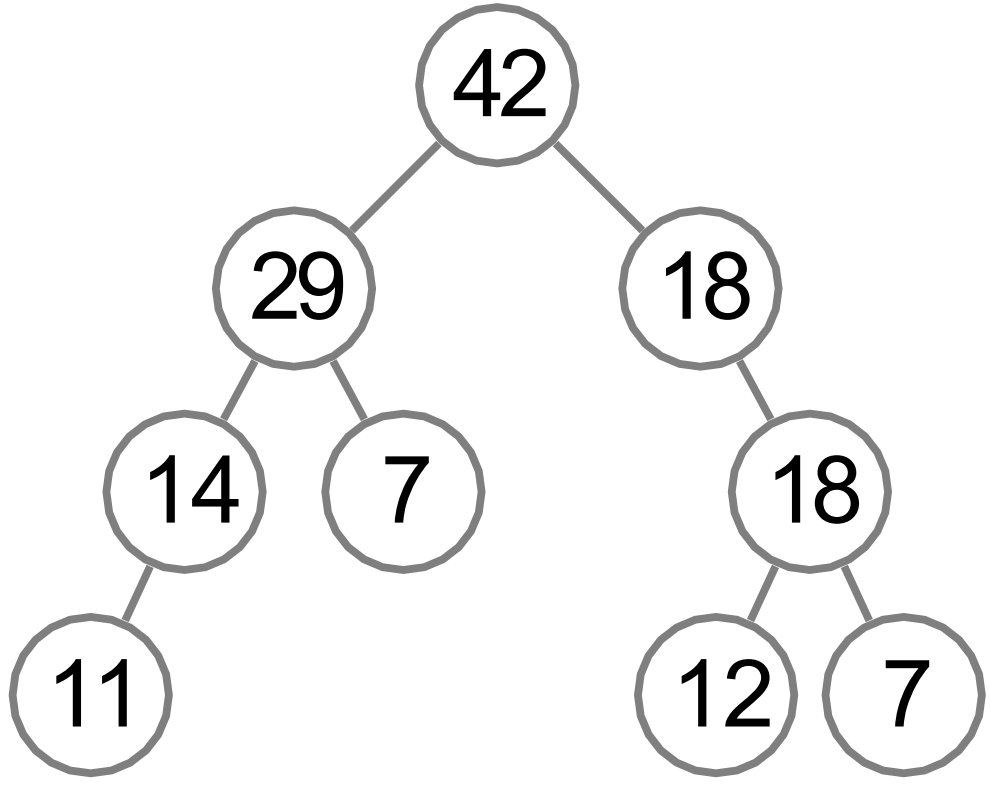
Is it a binary max-heap?



Outline

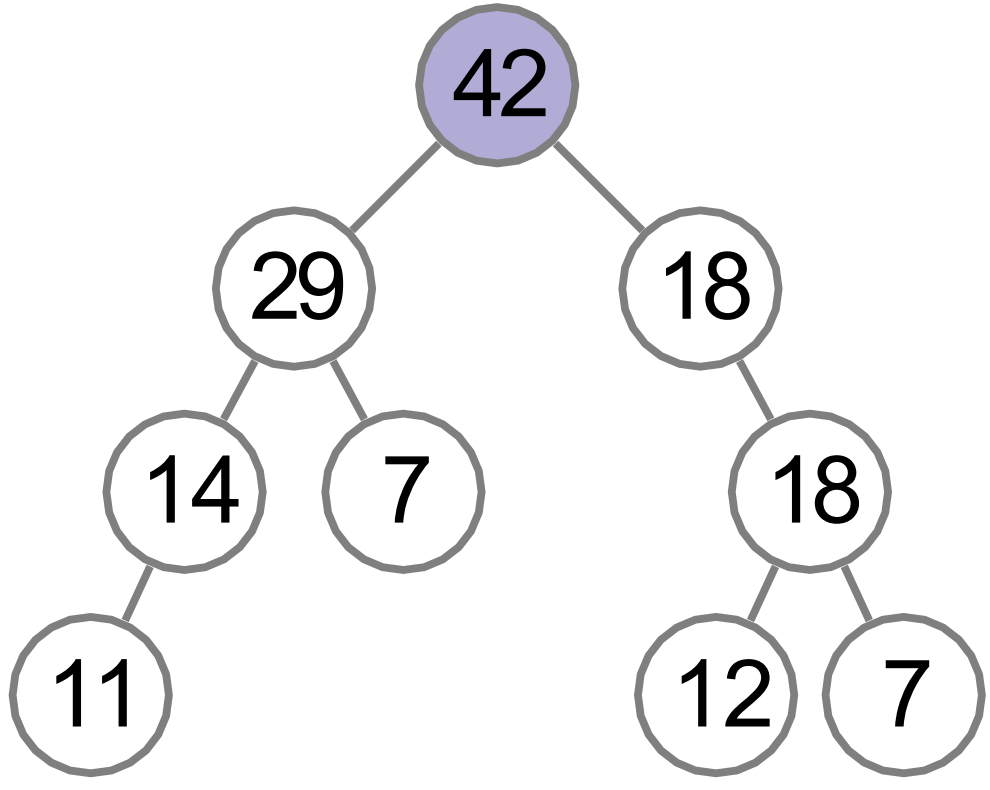
1. Binary Trees
2. Basic Operations
3. Complete Binary Trees
4. Pseudocode
5. Heap Sort
6. Final Remarks

GetMax



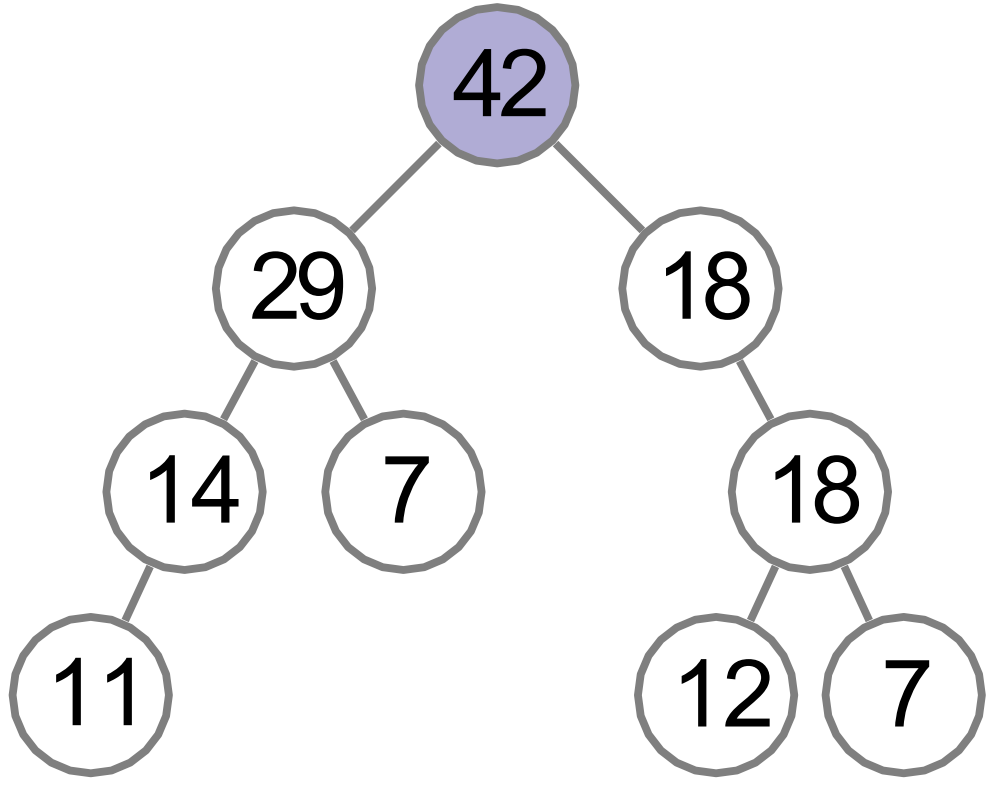
GetMax

return the root value



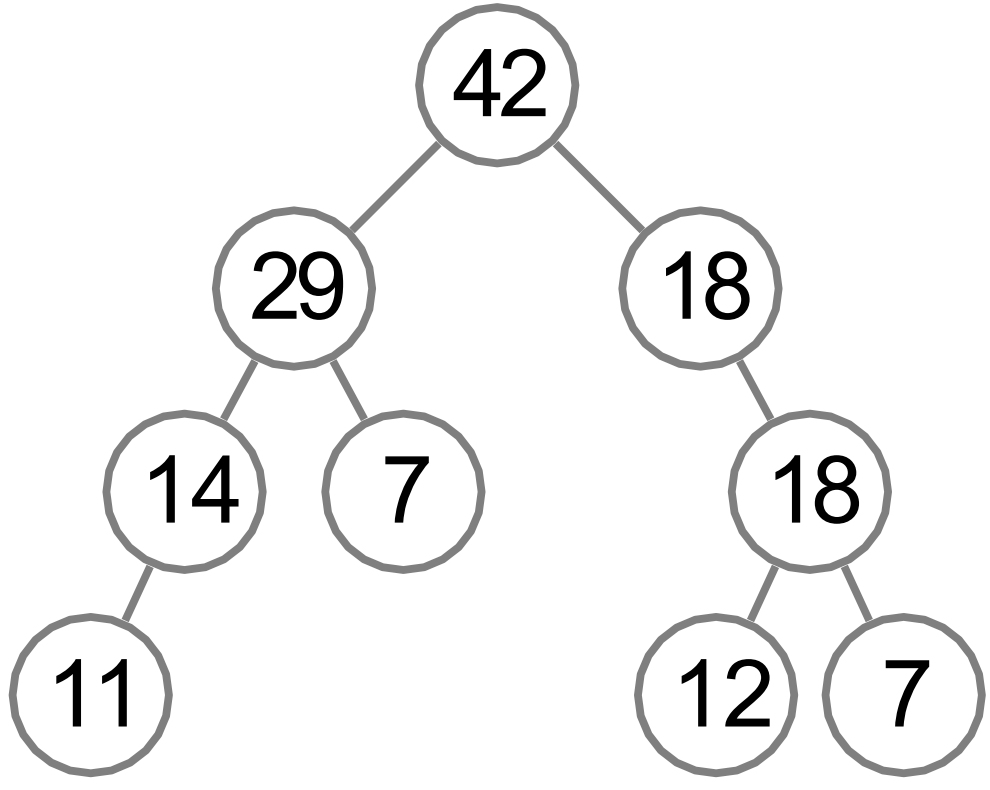
GetMax

return the root value



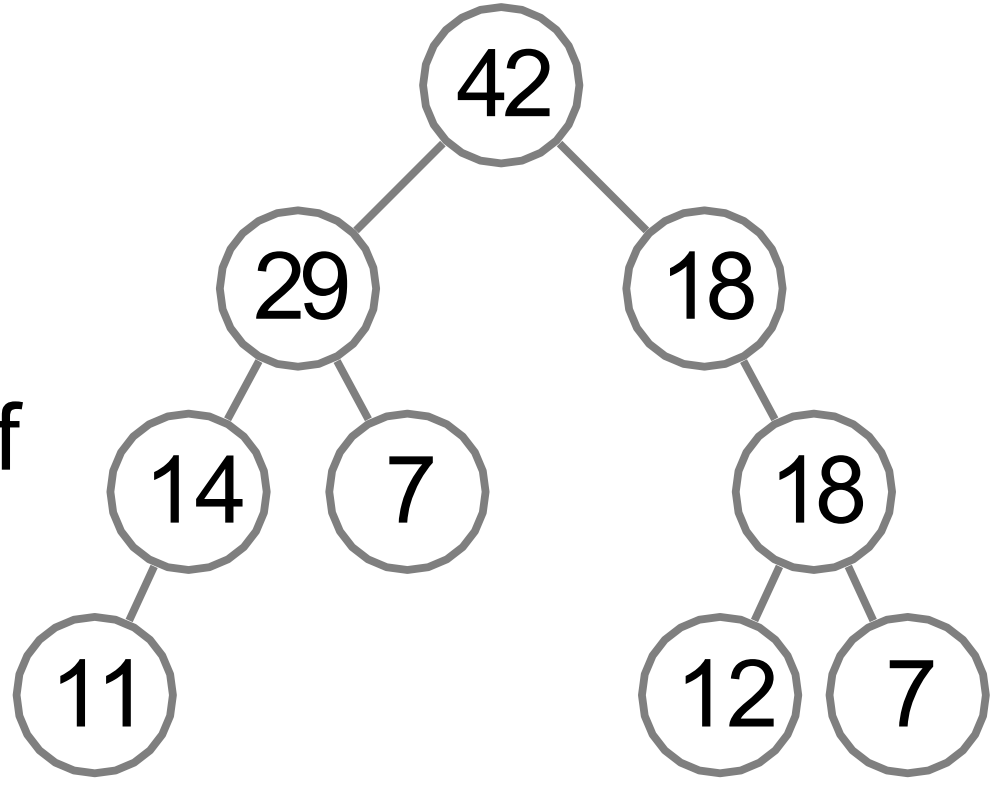
running time: $O(1)$

Insert



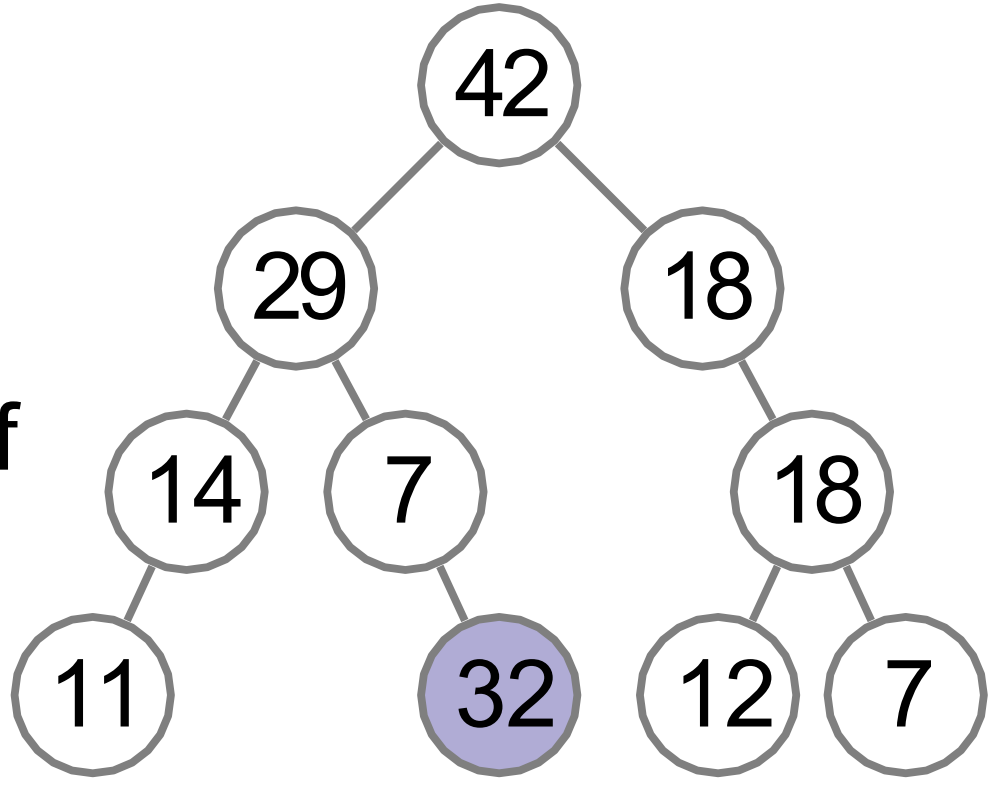
Insert

attach a new node to any leaf



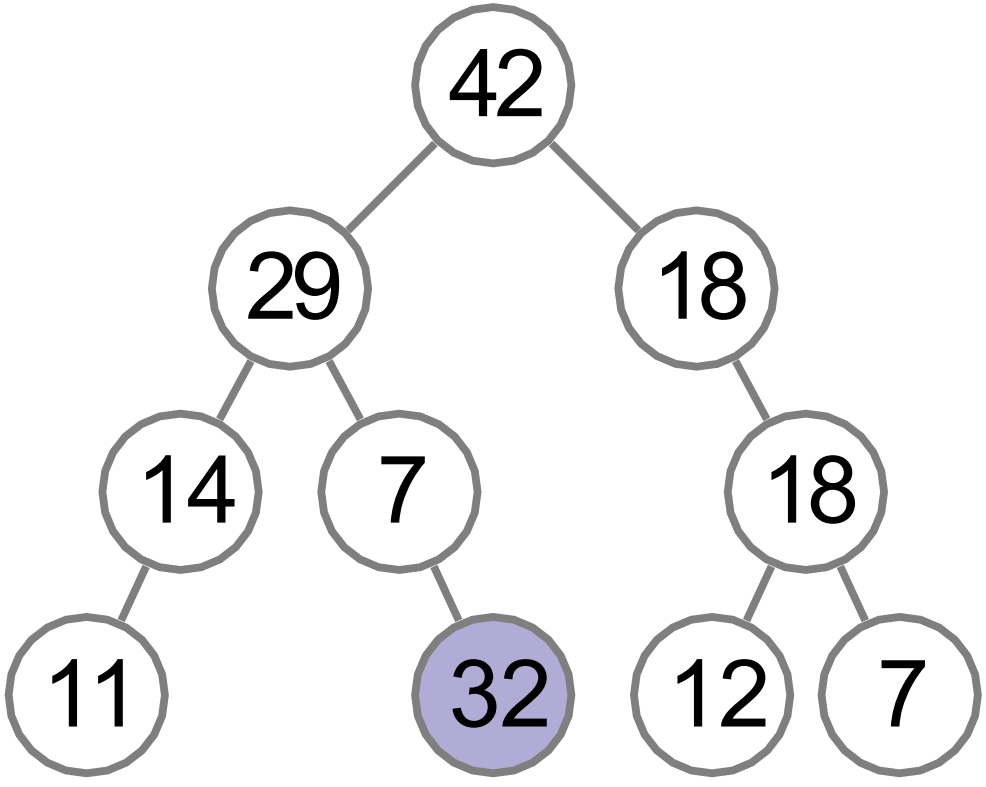
Insert

attach a new node to any leaf



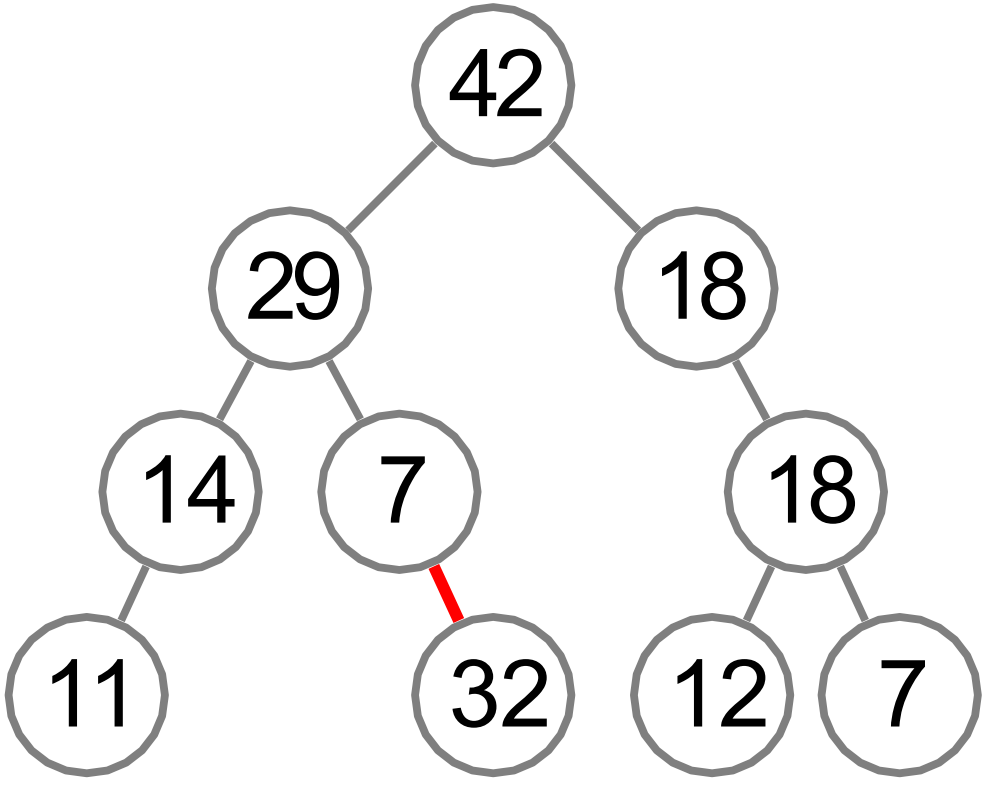
Insert

this may violate the heap property



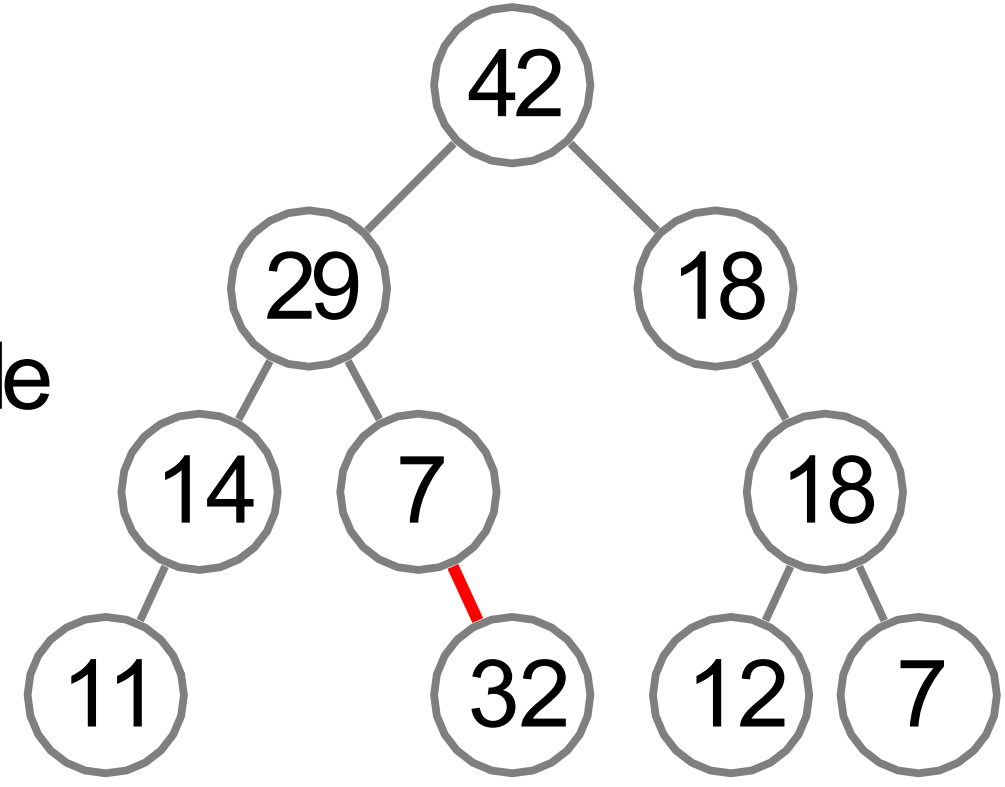
Insert

this may violate the heap property



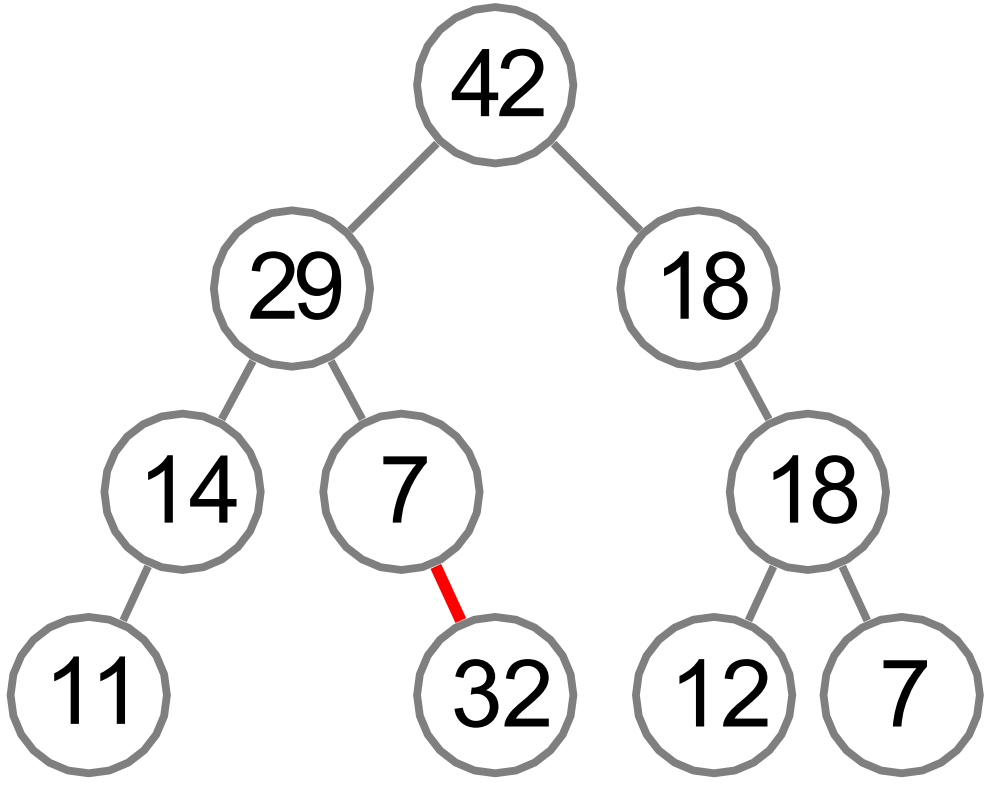
Insert

to fix this, we let the new node sift up

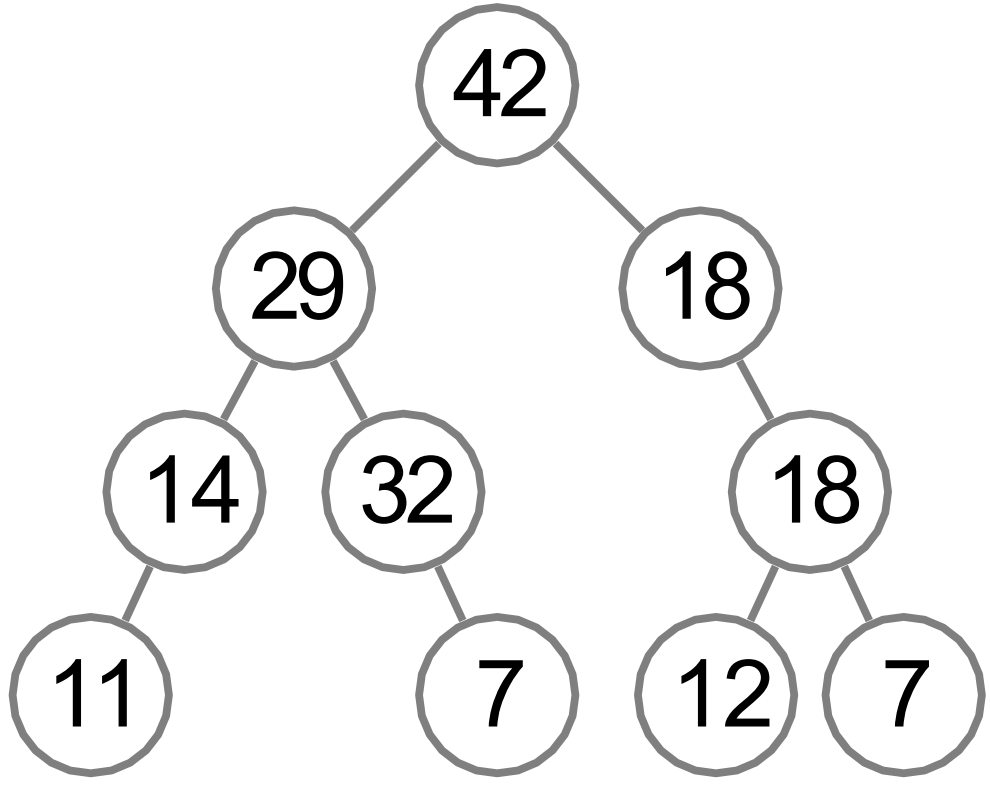


SiftUp

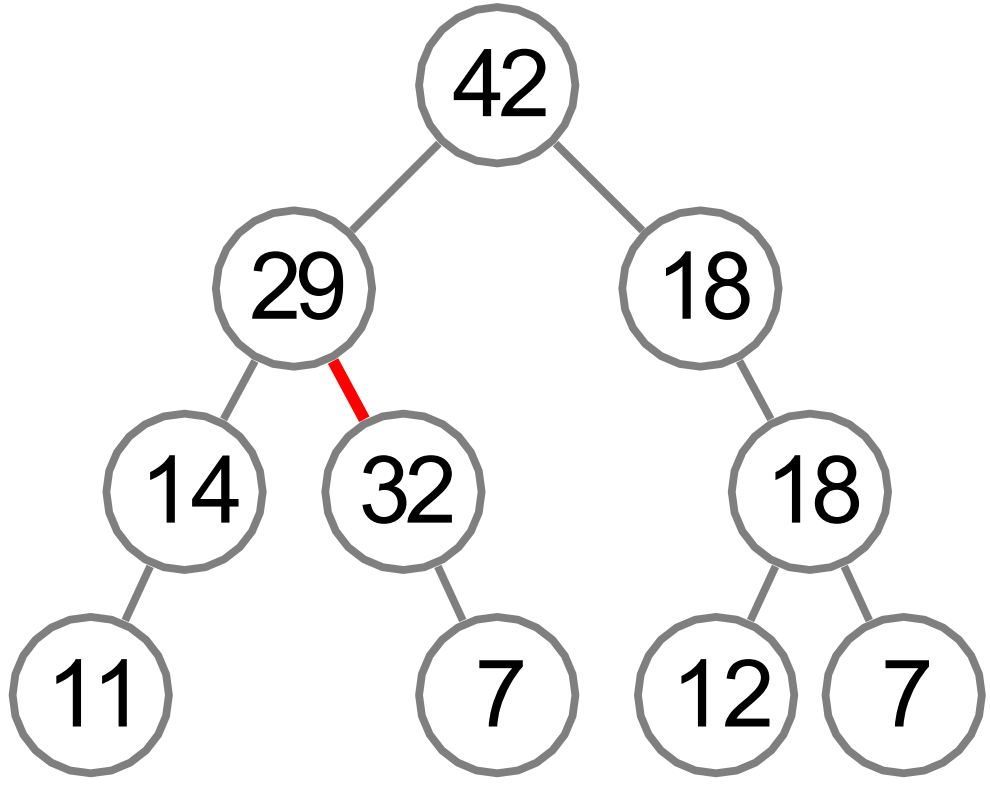
for this, we swap the problematic node with its parent until the property is satisfied



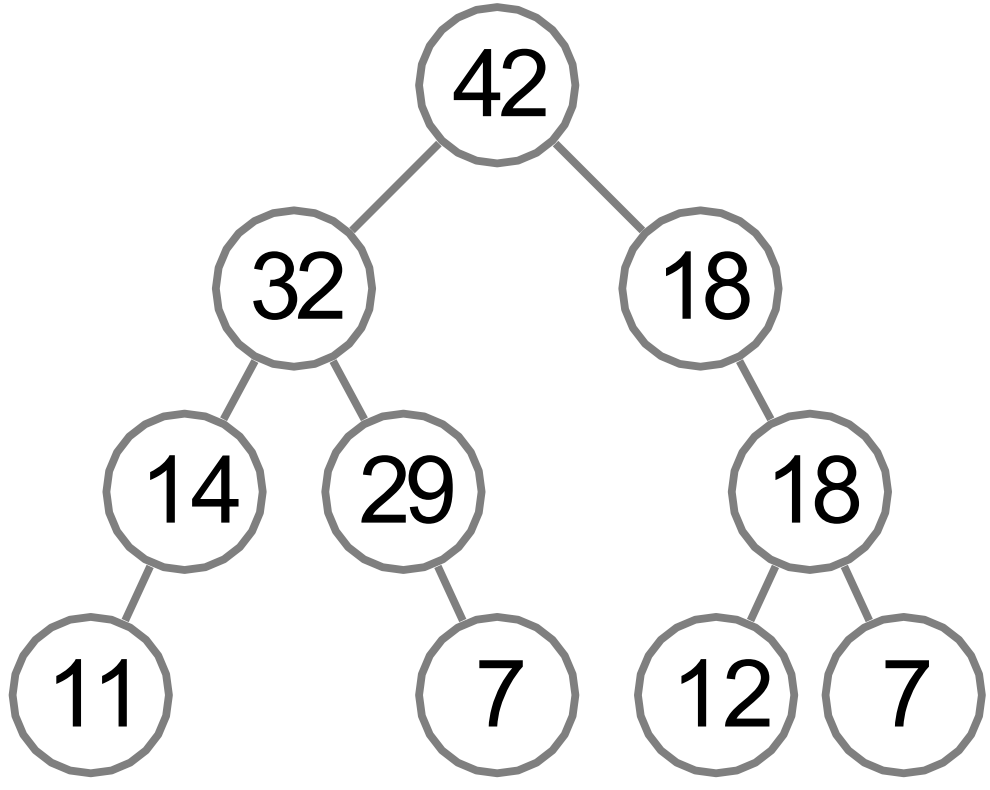
SiftUp



SiftUp

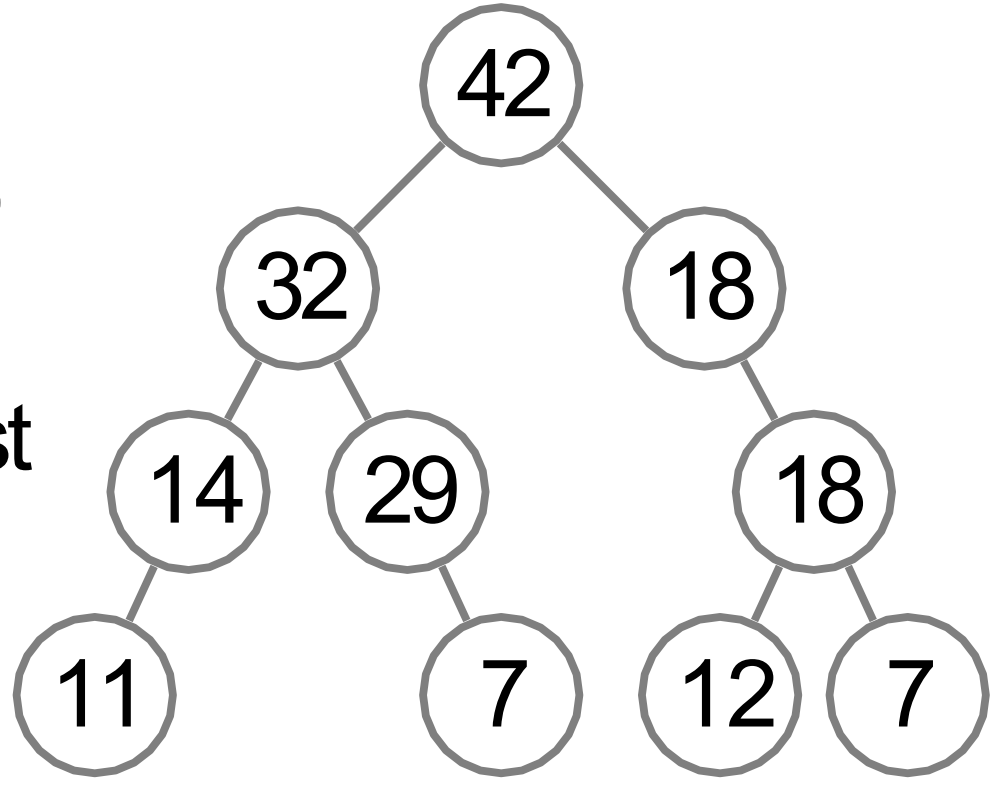


SiftUp



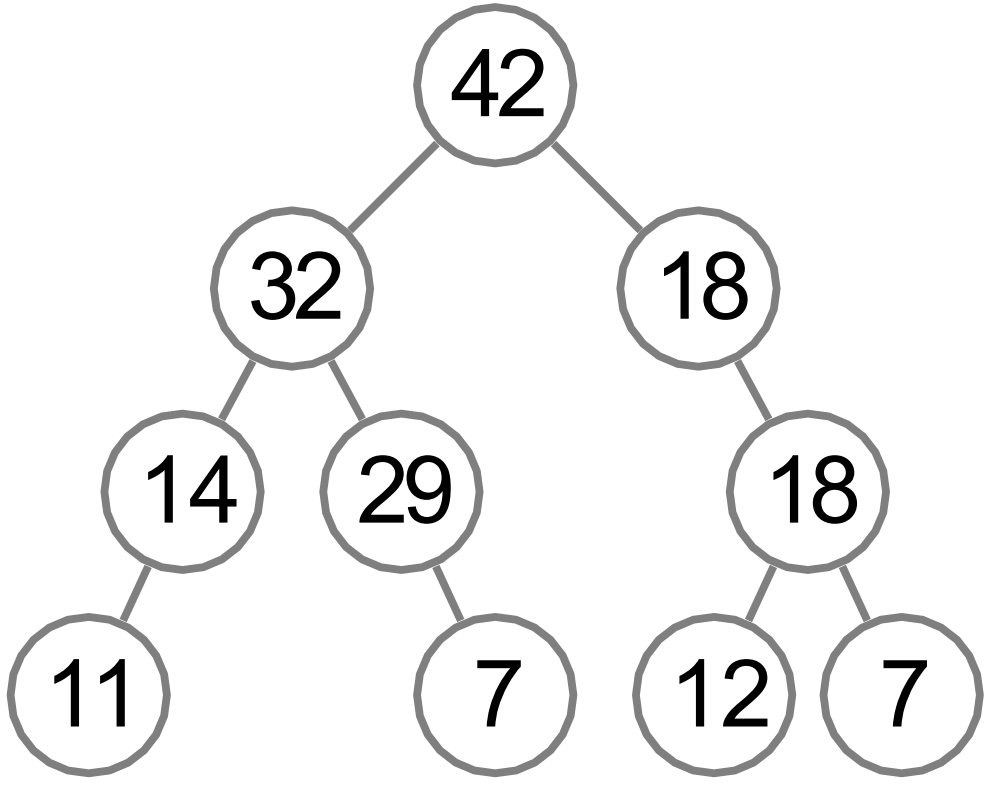
SiftUp

invariant: heap property is violated on at most one edge

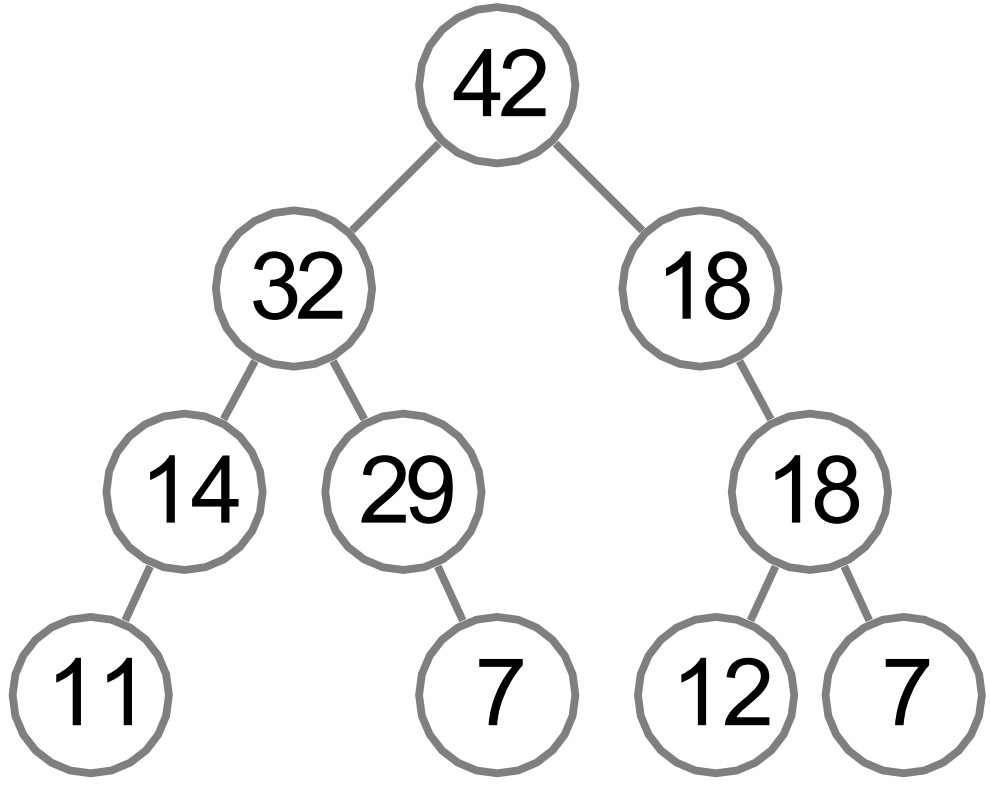


SiftUp

this edge gets closer to the root while sifting up



SiftUp



running time: O(tree height)

Question

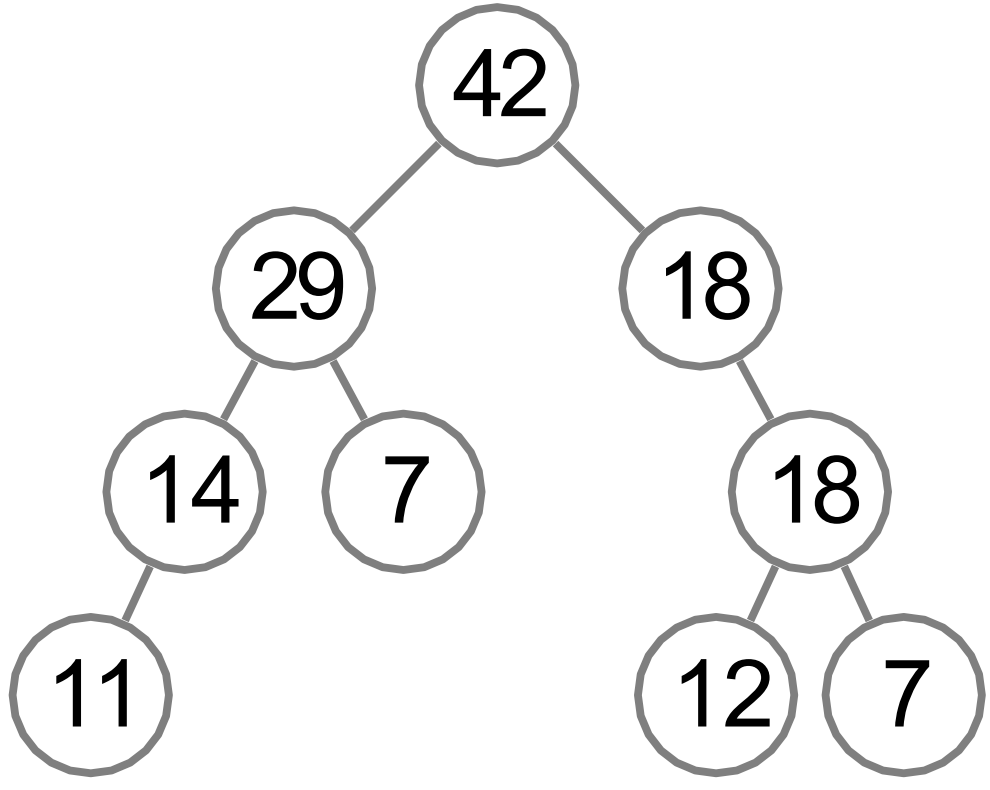
Assume that initially a binary max-heap H contains just a single node with value 0. Then, for all i from 1 to n, we insert i into H. We do this by attaching a new element to some leaf of the current tree. What will be the height of the resulting tree?

$\Theta(n^2)$

$\Theta(n)$

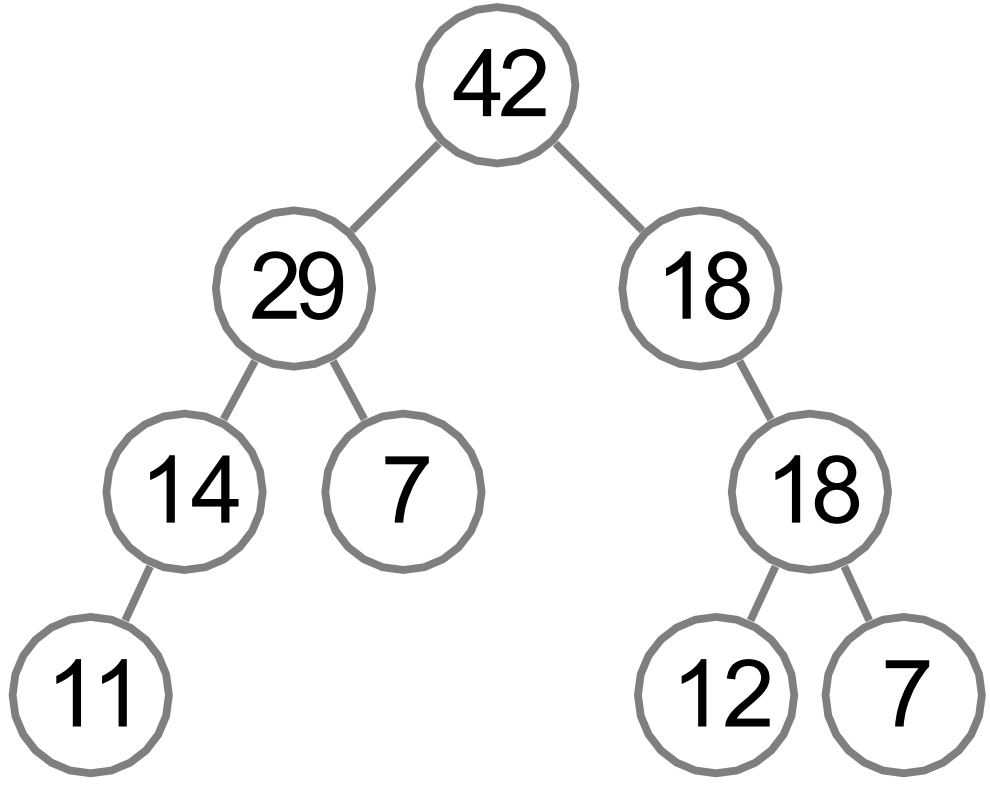
$\Theta(\log n)$

ExtractMax



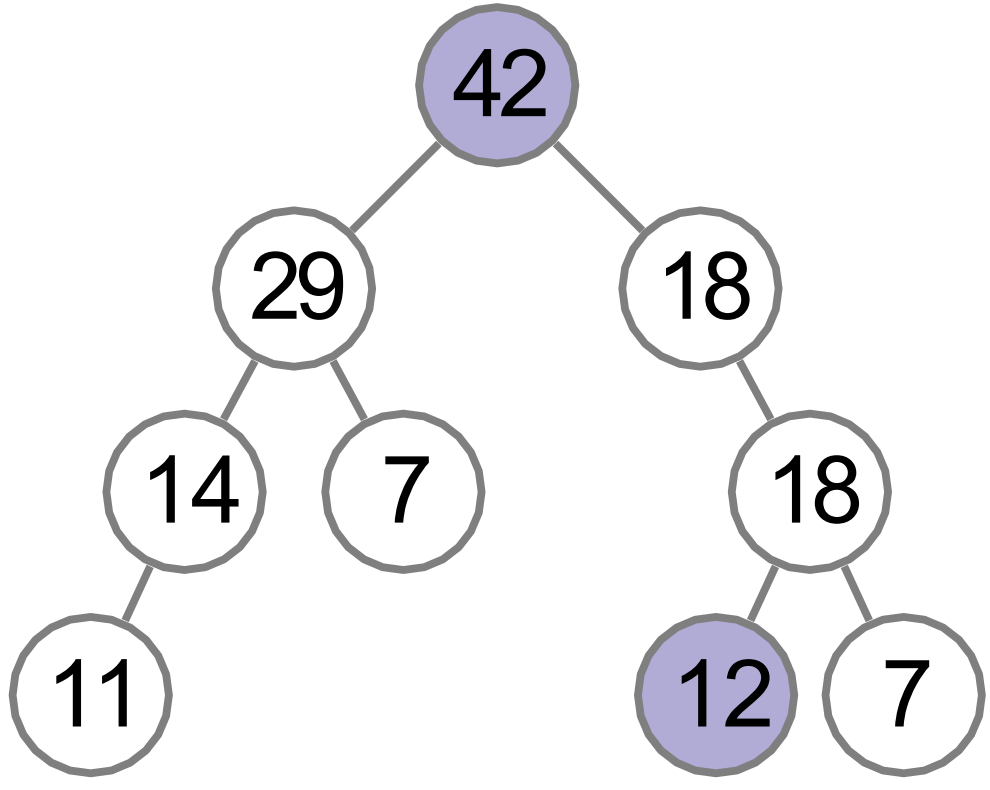
ExtractMax

replace the root
with any leaf



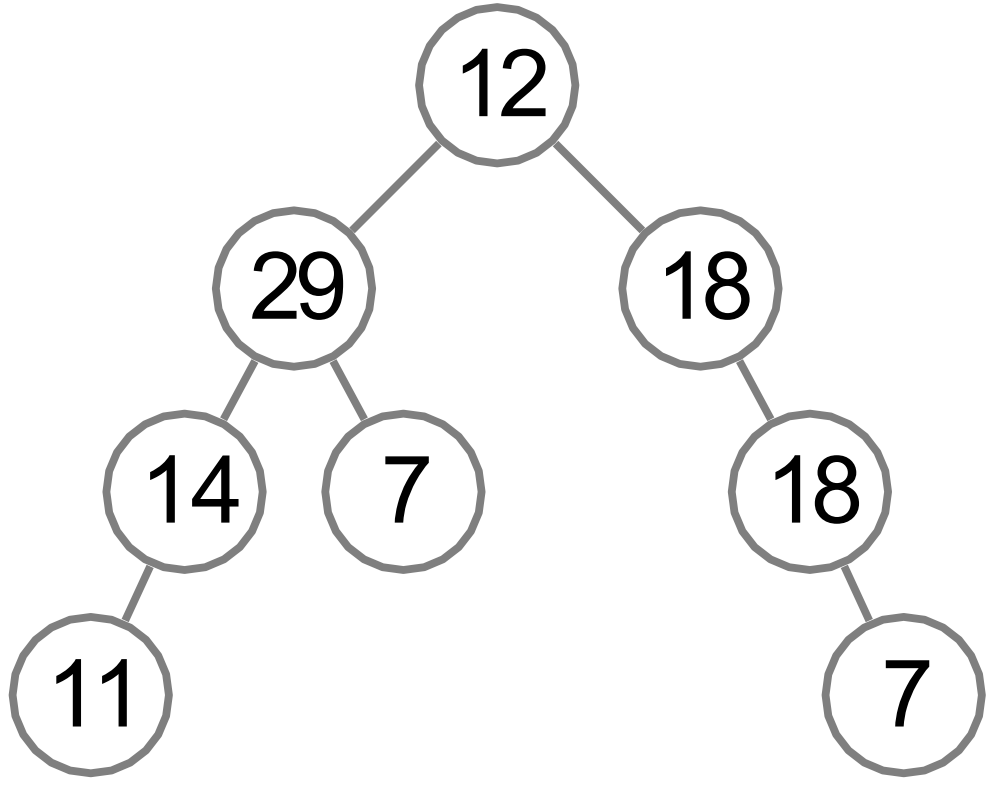
ExtractMax

replace the root
with any leaf



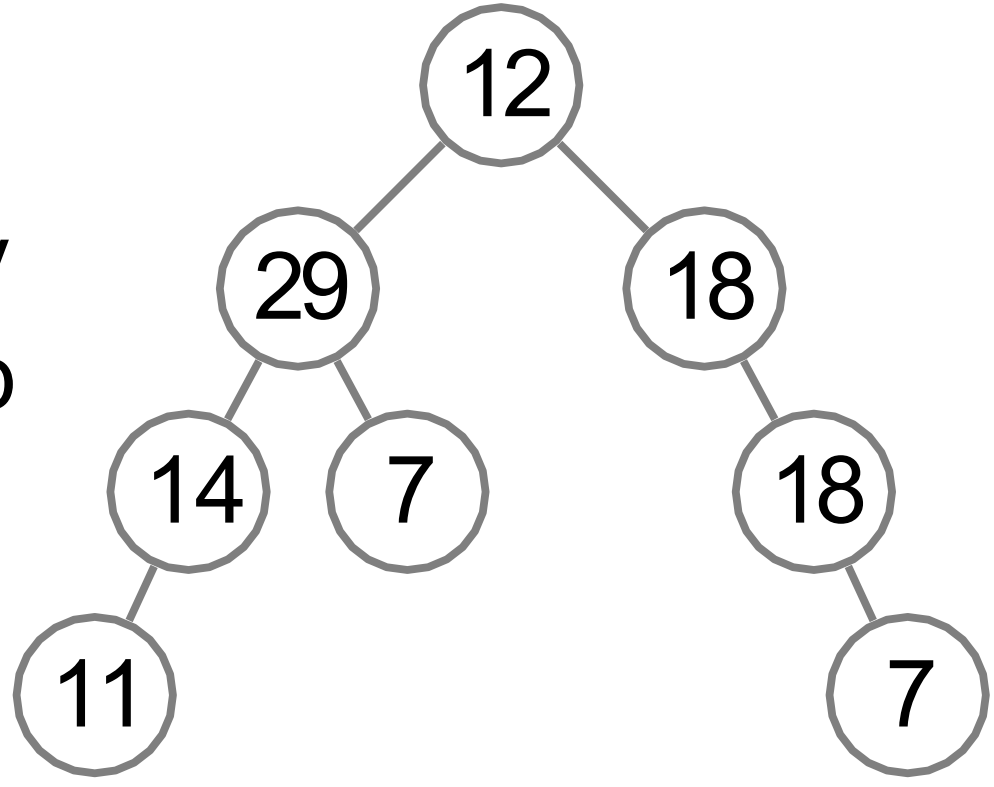
ExtractMax

replace the root
with any leaf



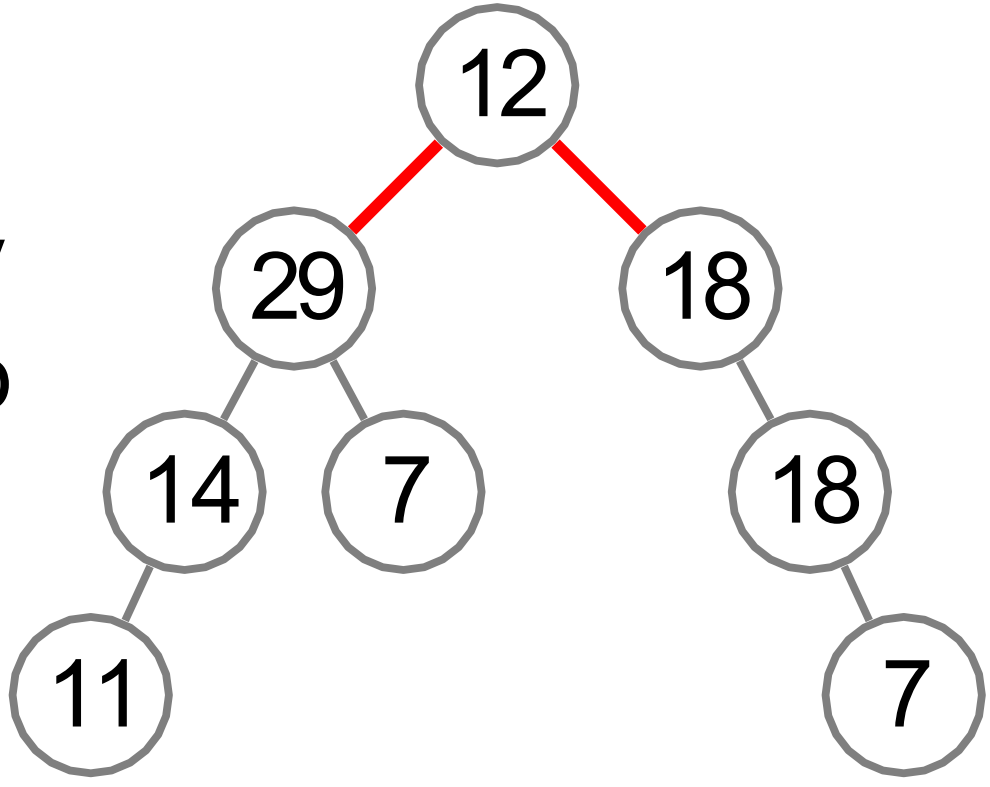
ExtractMax

again, this may violate the heap property



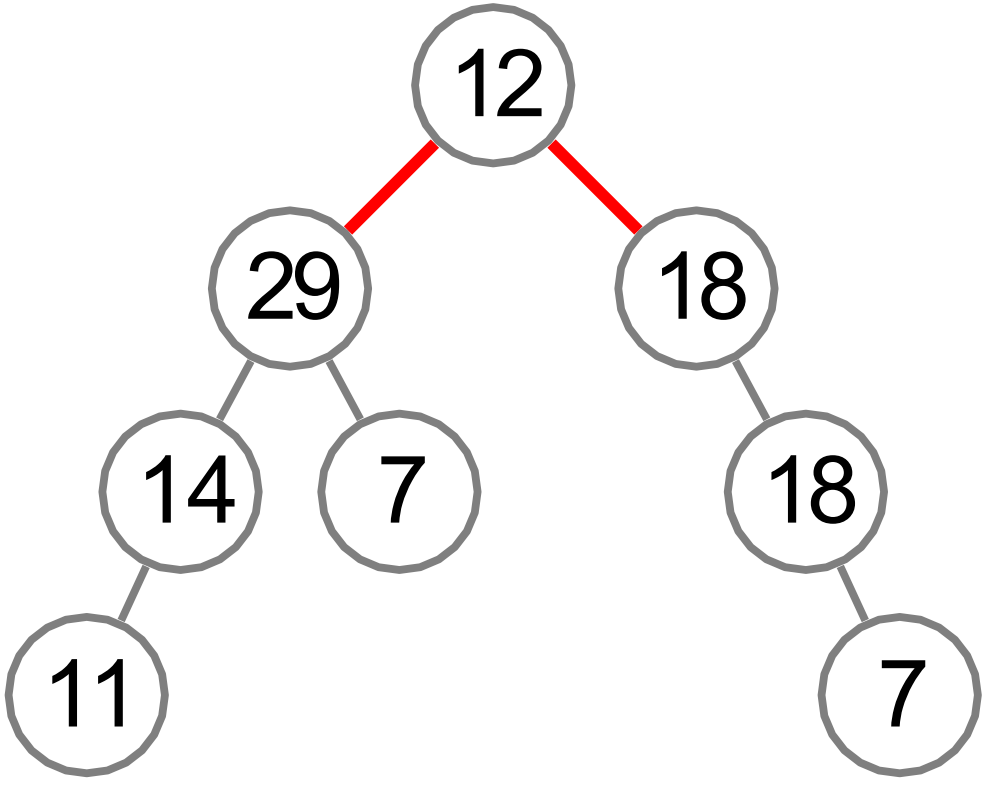
ExtractMax

again, this may violate the heap property



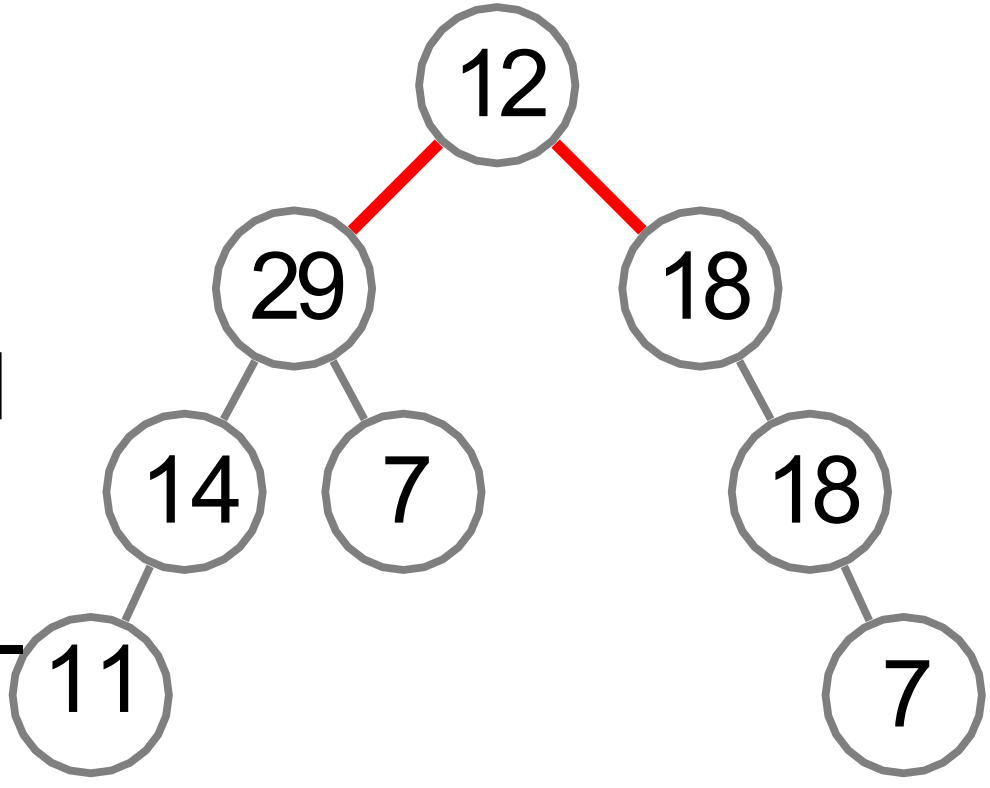
ExtractMax

to fix it, we let the problematic node sift down

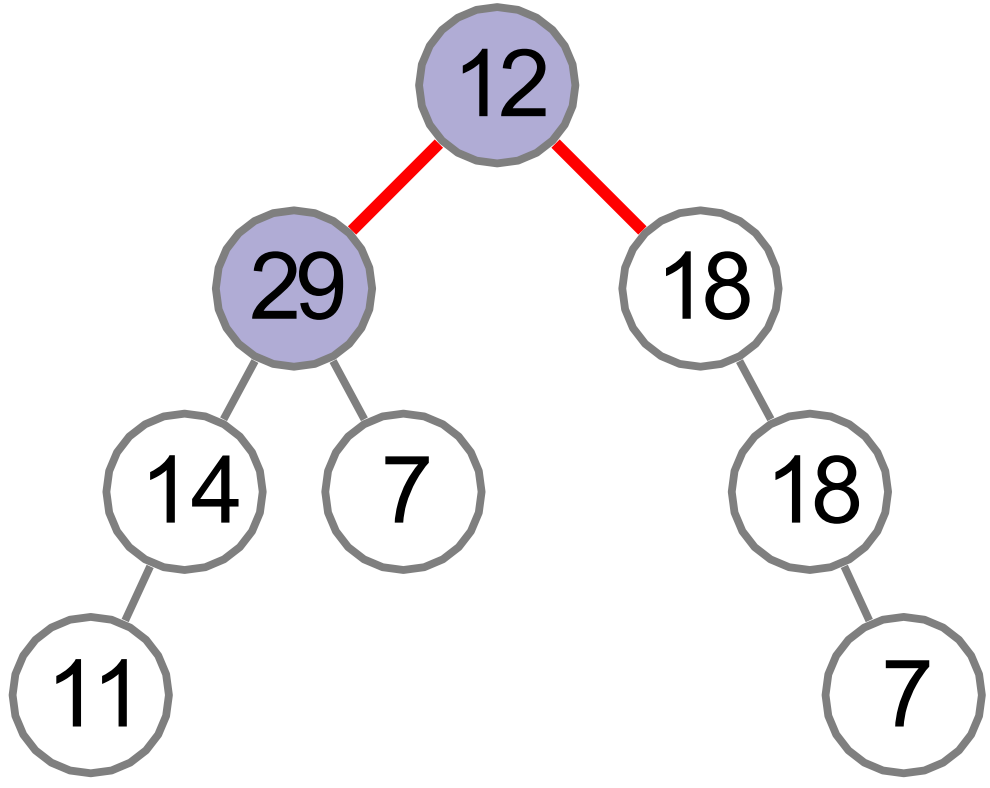


SiftDown

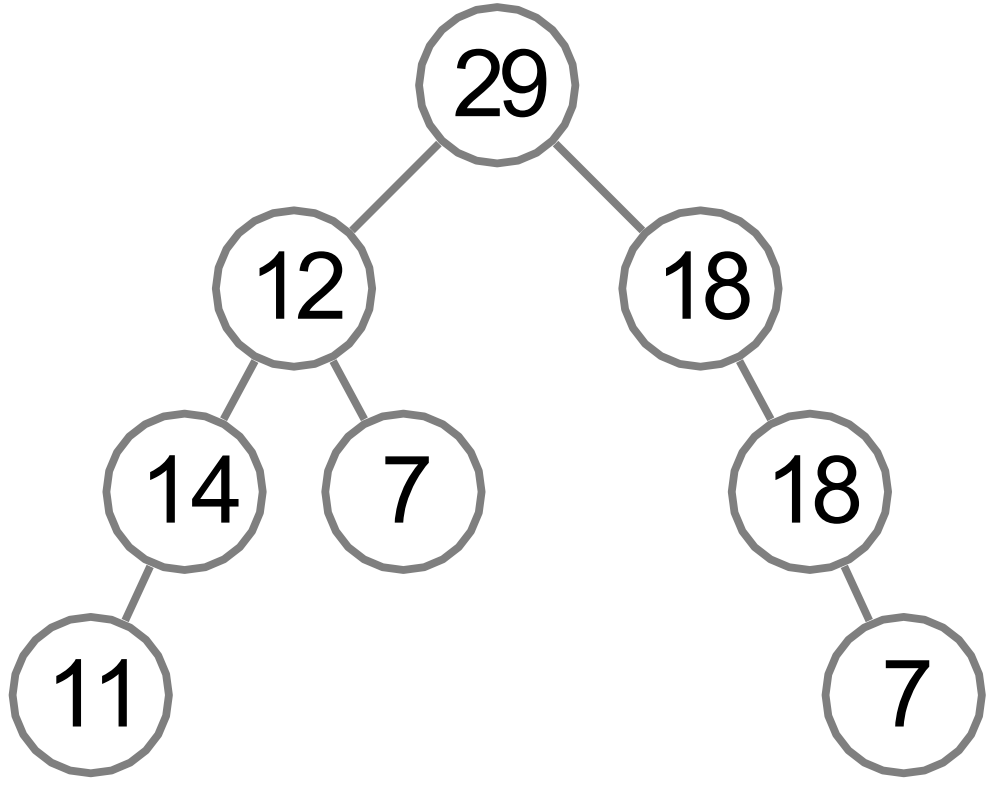
for this, we swap the problematic node with larger child until the heap property is satisfied



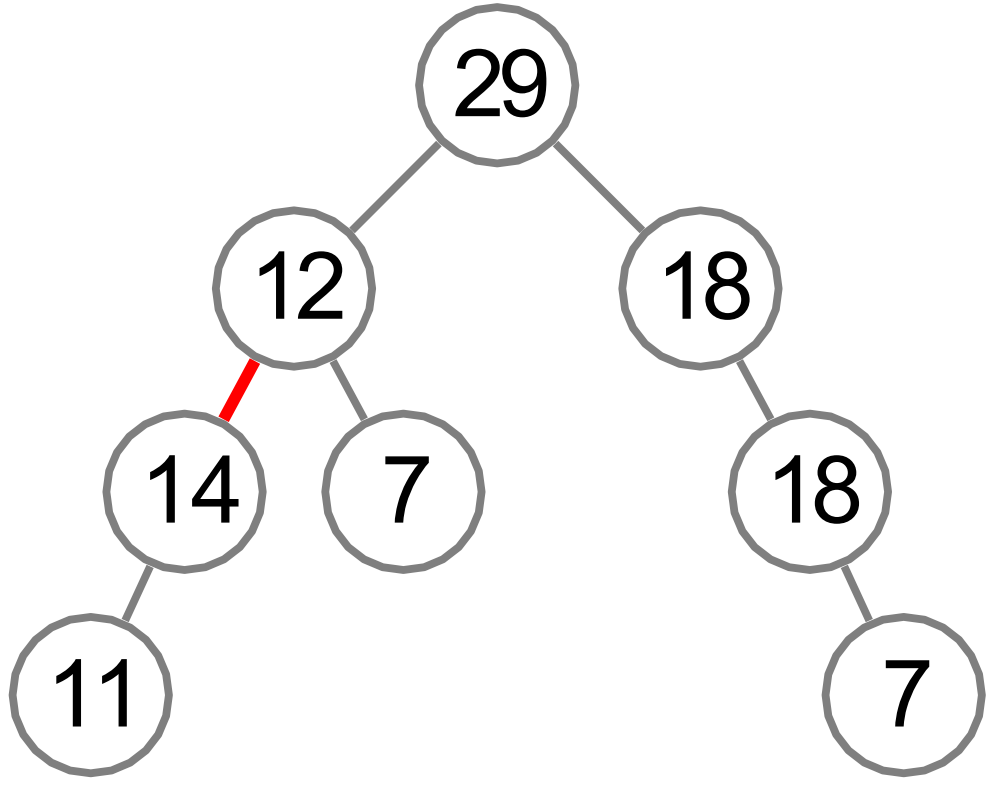
SiftDown



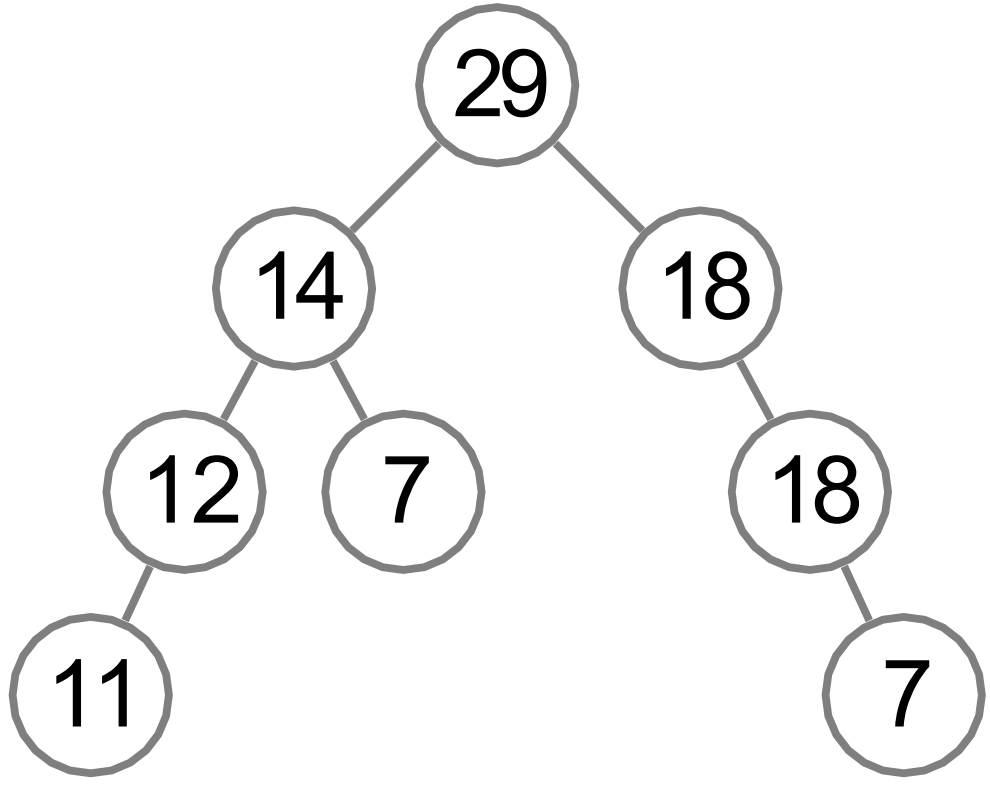
SiftDown



SiftDown

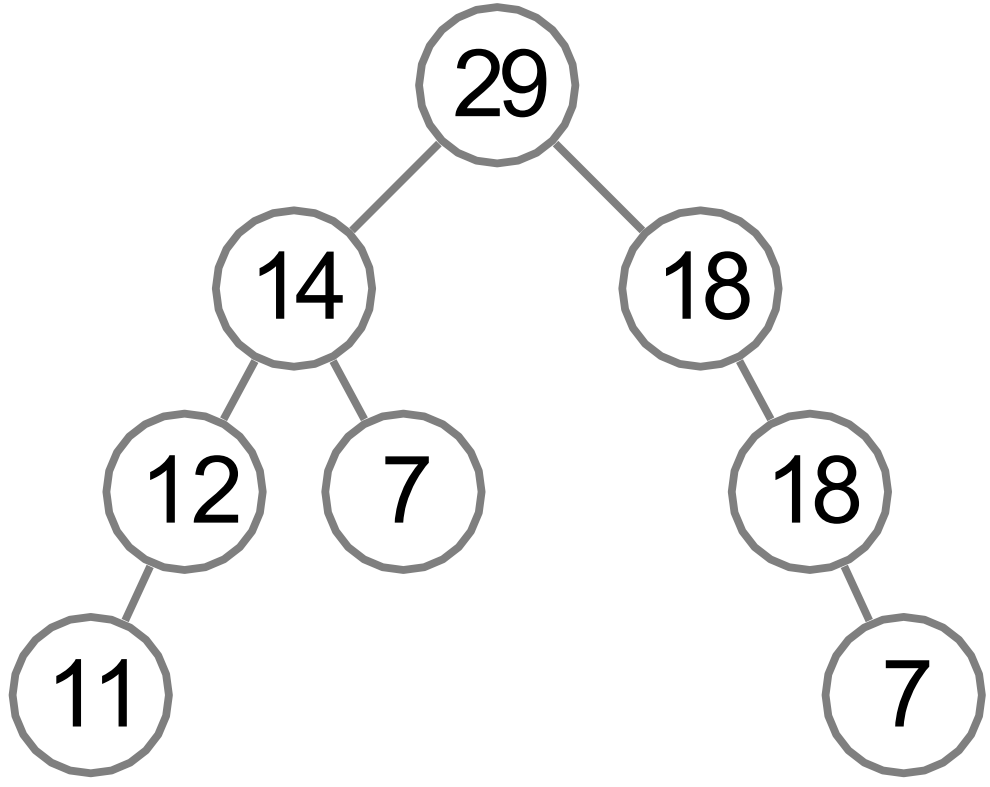


SiftDown

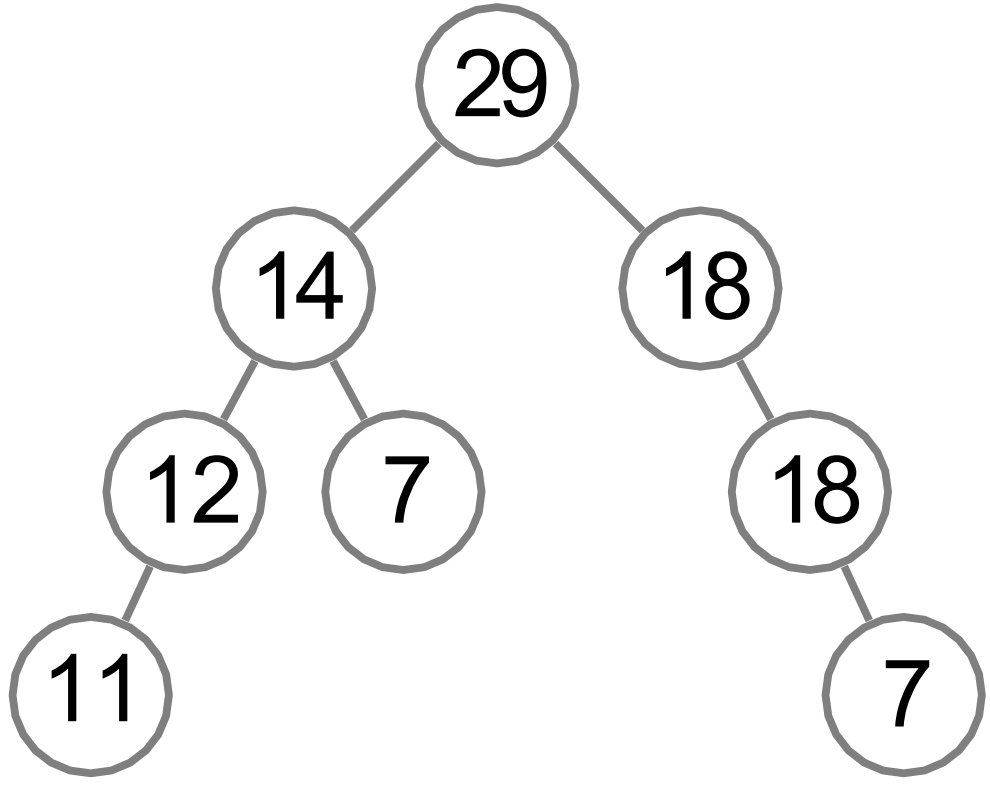


SiftDown

we swap with the larger child which automatically fixes one of the two bad edges

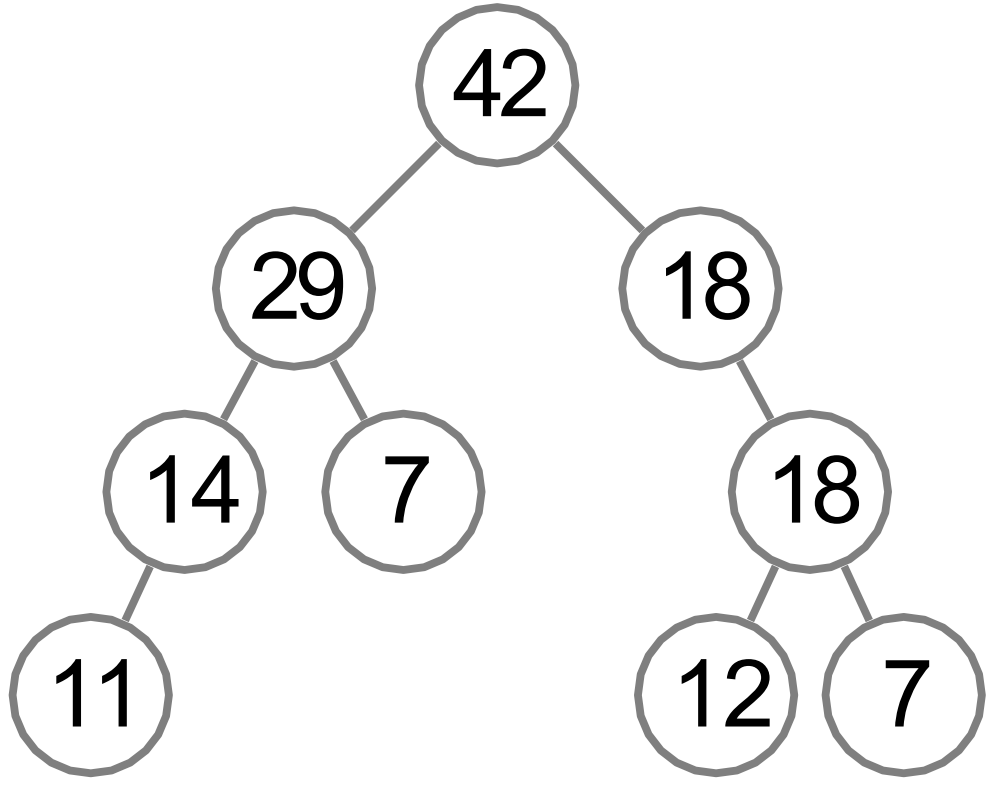


SiftDown



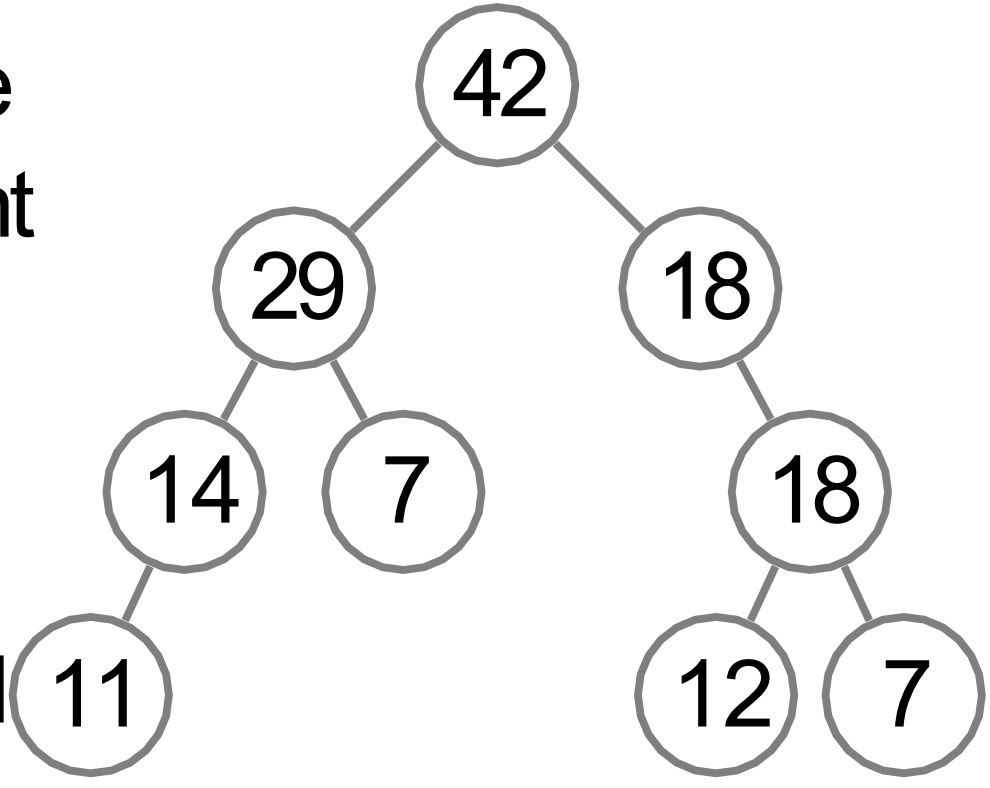
running time: O(tree height)

ChangePriority



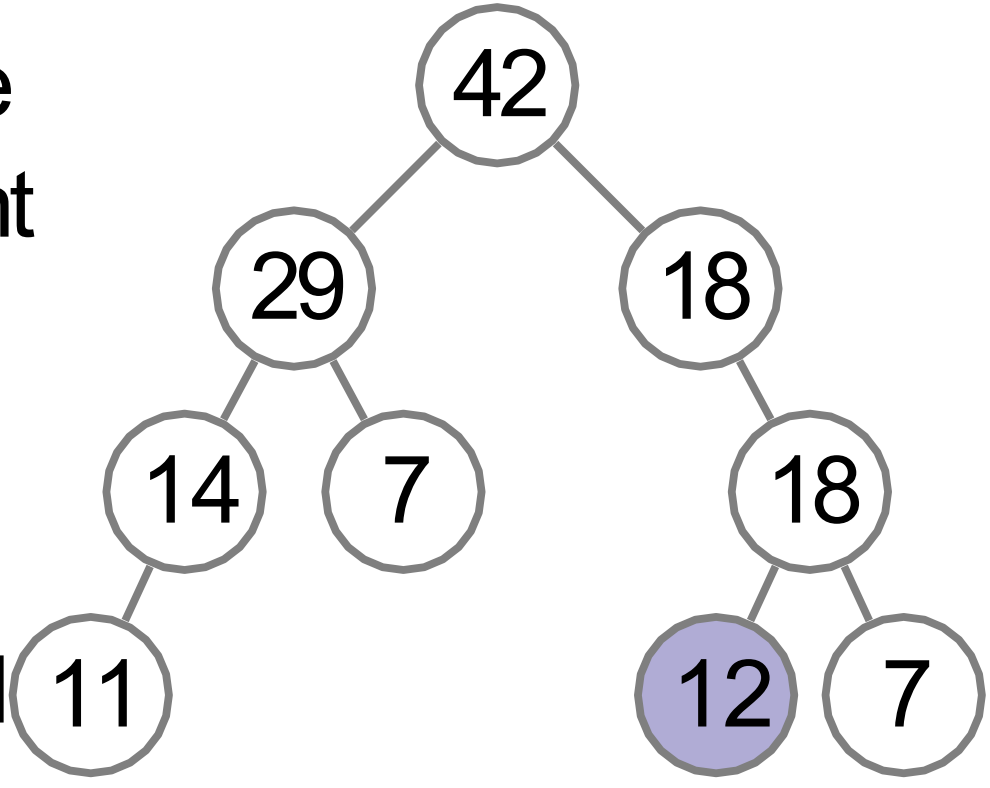
ChangePriority

change the priority and let the changed element sift up or down depending on whether its priority decreased or increased



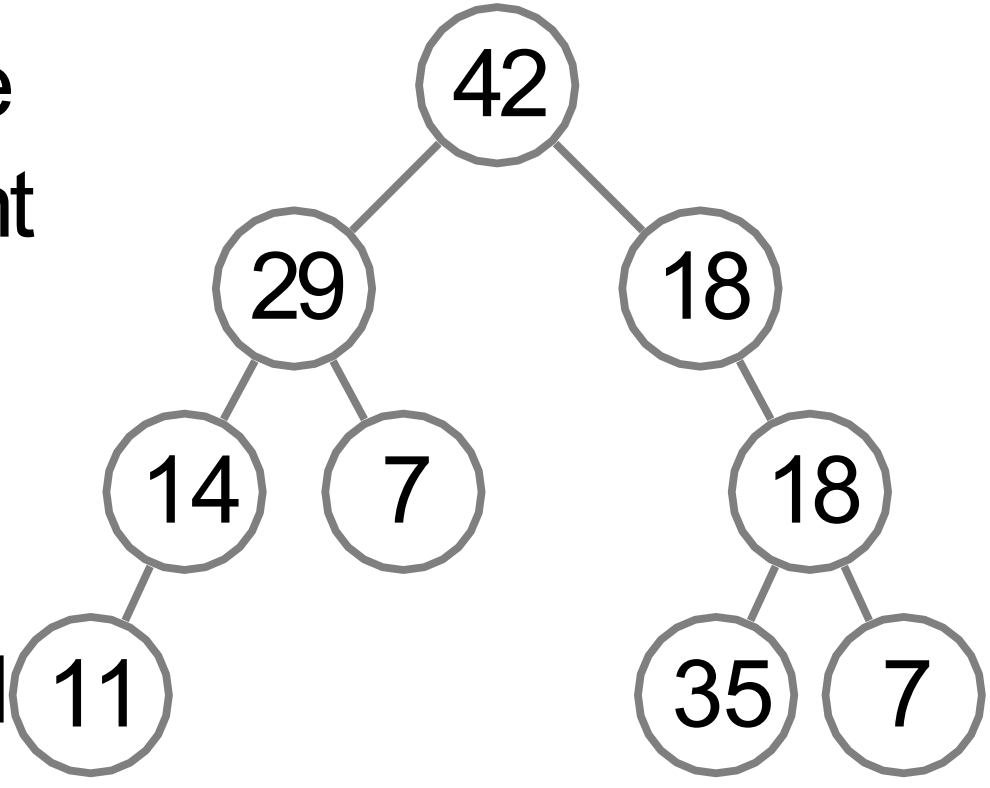
ChangePriority

change the priority and let the changed element sift up or down depending on whether its priority decreased or increased

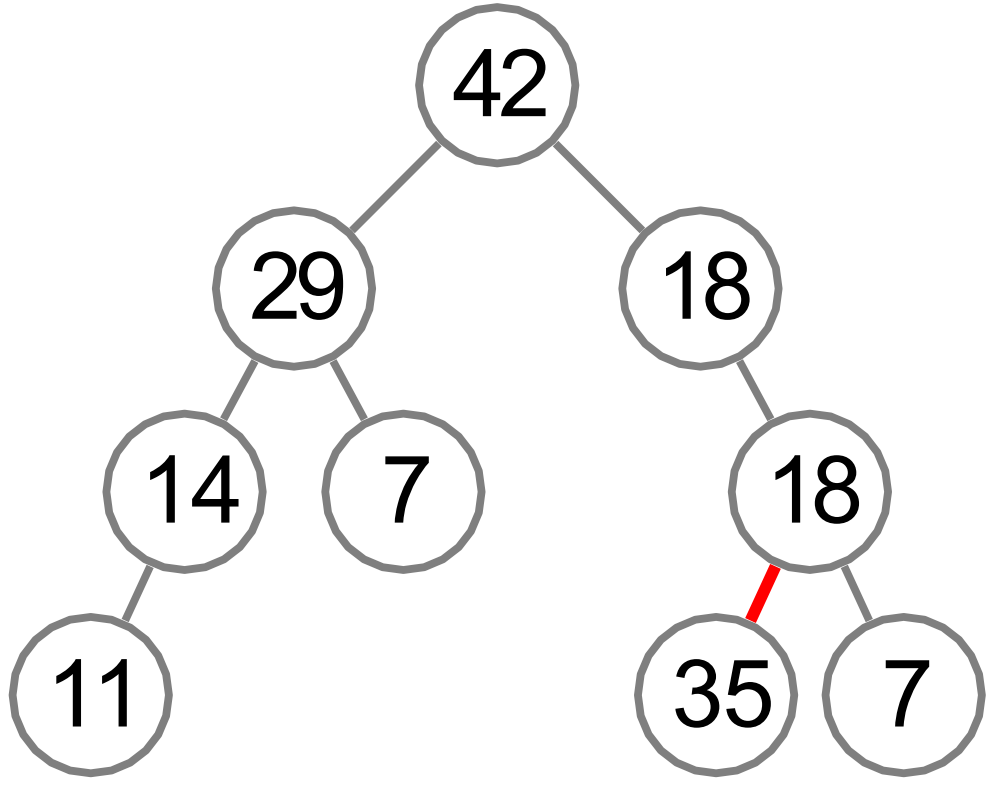


ChangePriority

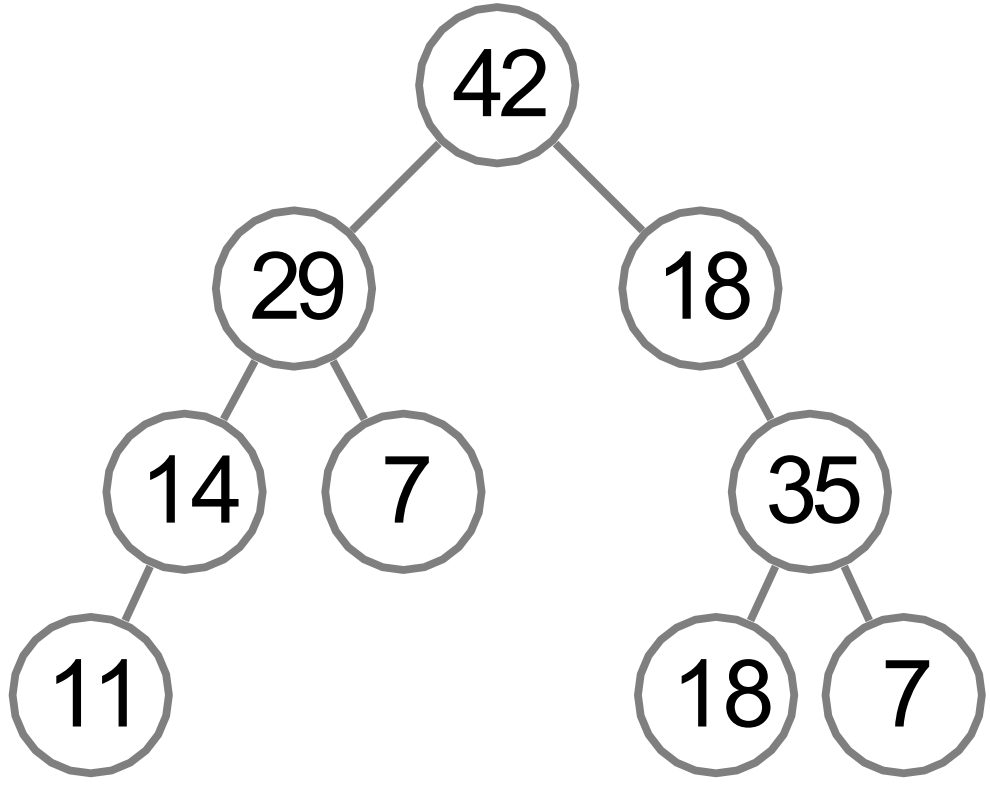
change the priority and let the changed element sift up or down depending on whether its priority decreased or increased



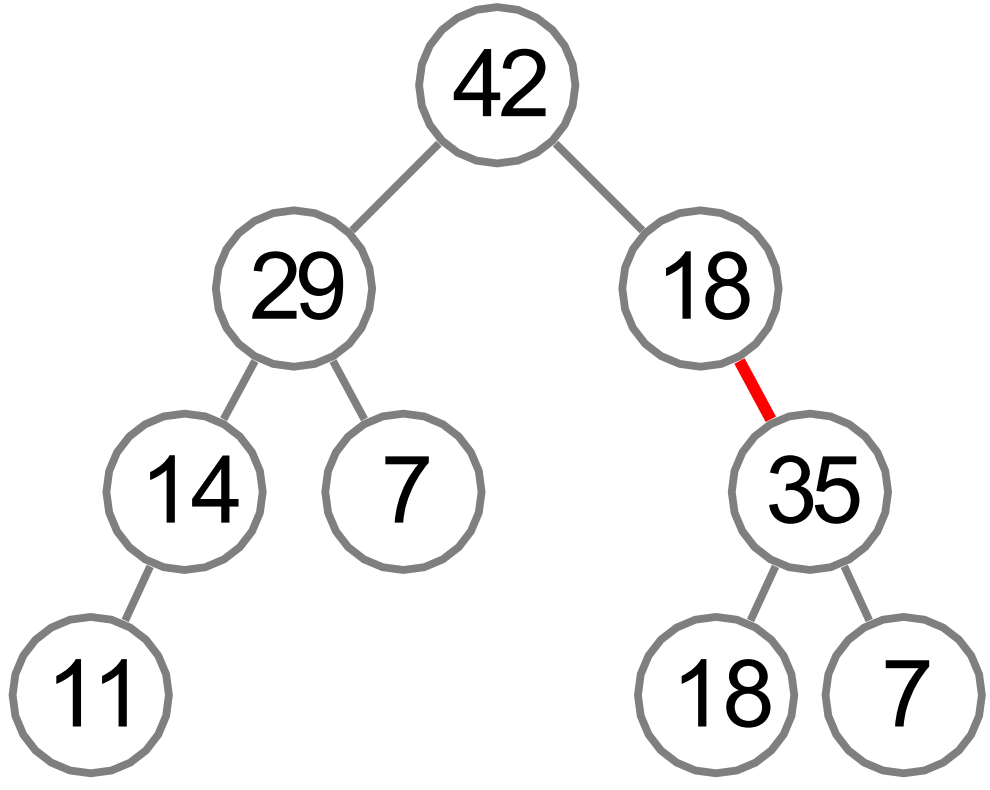
ChangePriority



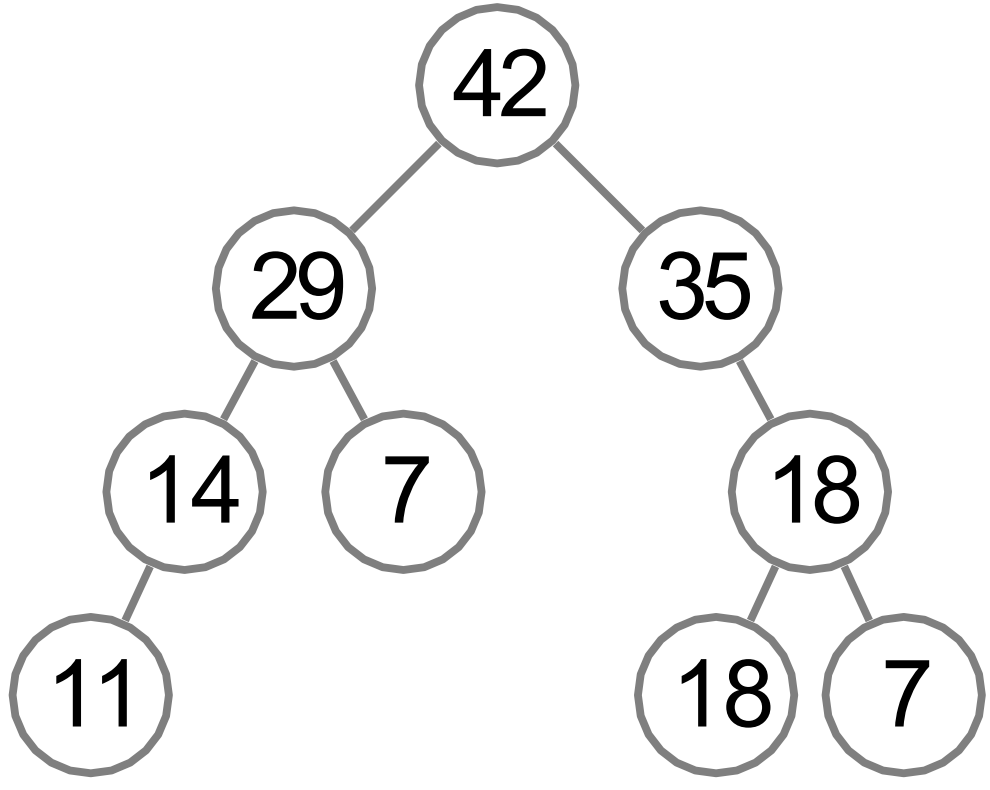
ChangePriority



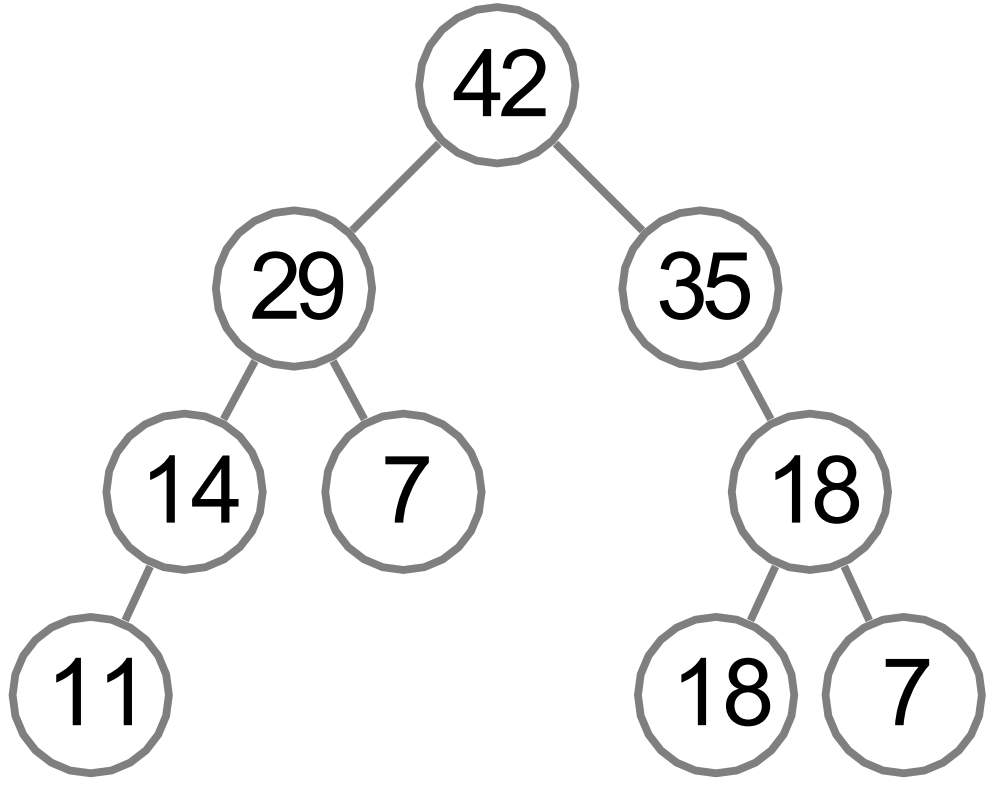
ChangePriority



ChangePriority

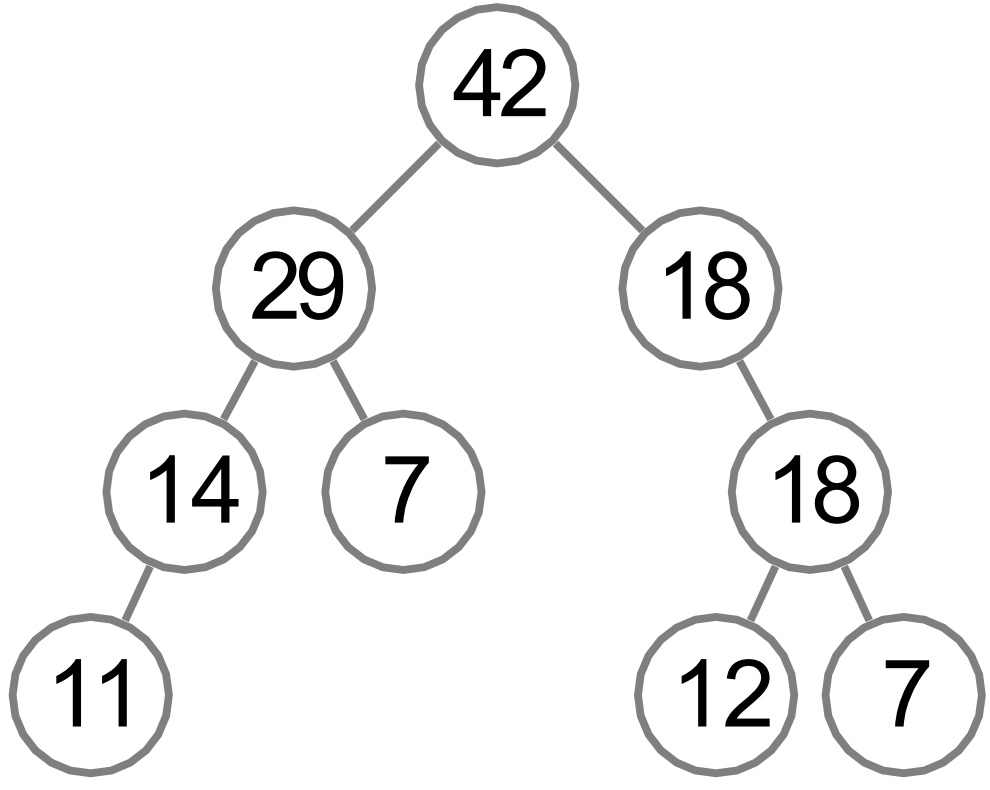


ChangePriority



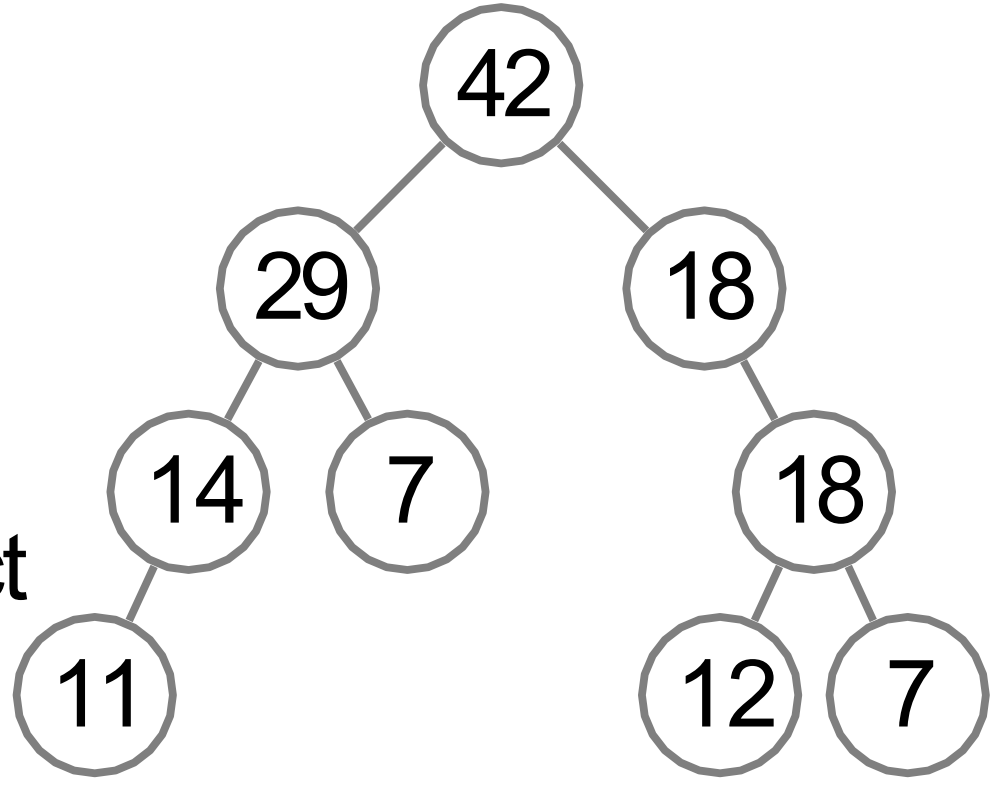
running time: O(tree height)

Remove

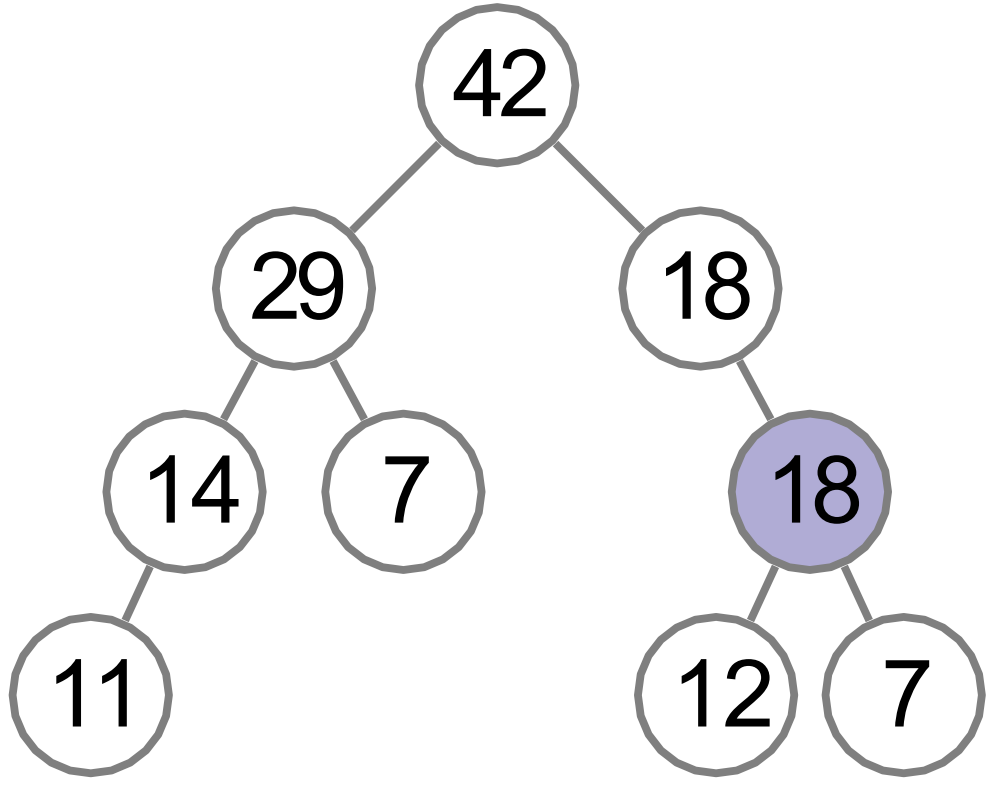


Remove

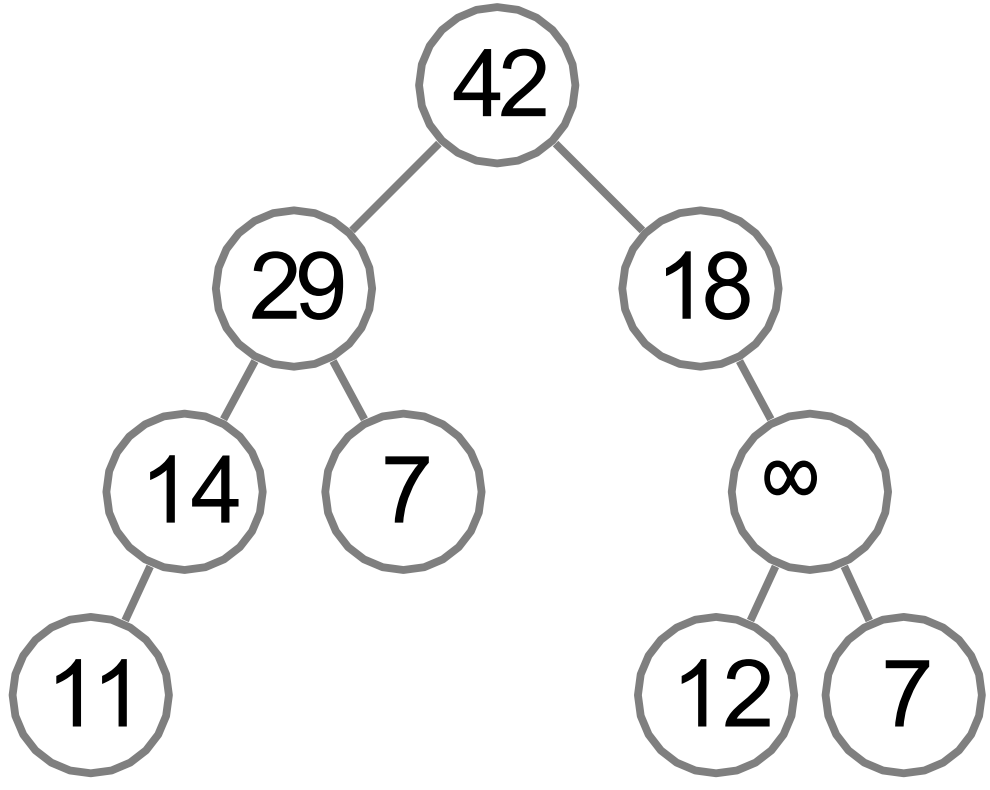
change the priority of the element to ∞, let it sift up, and then extract maximum



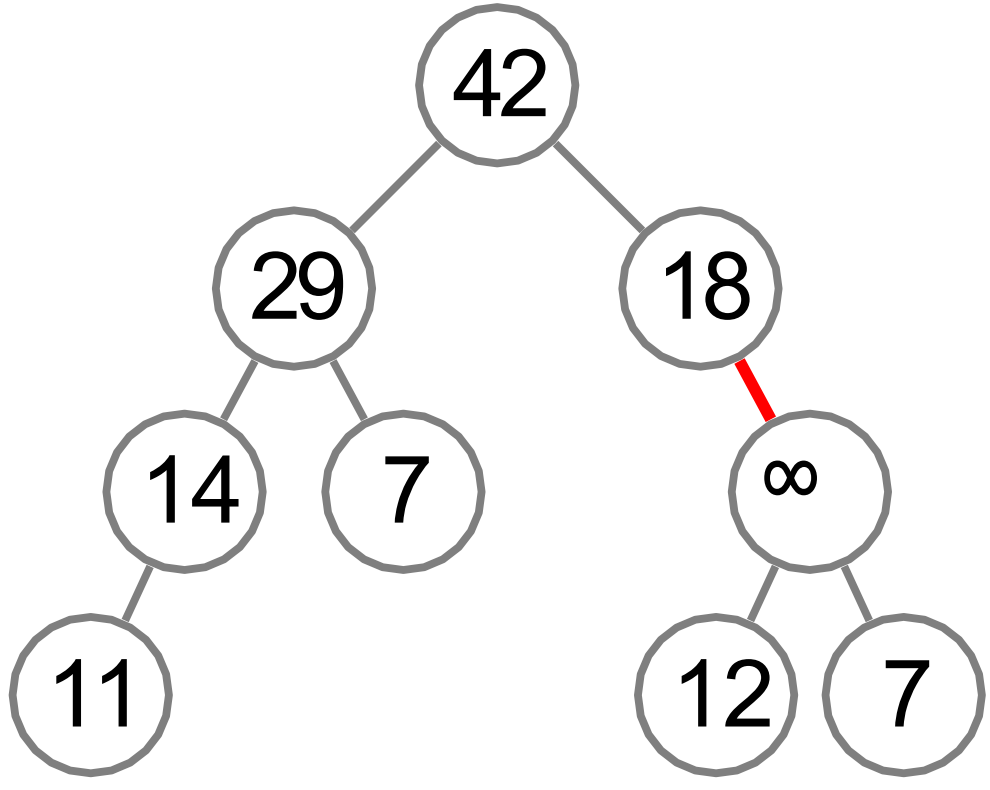
Remove



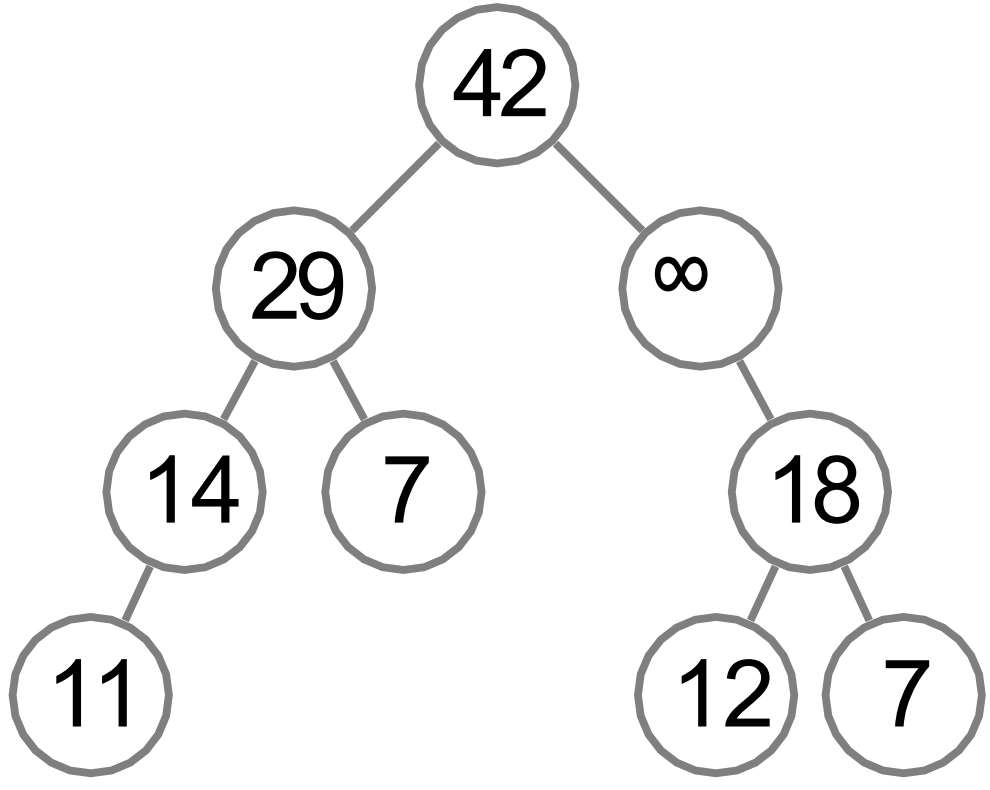
Remove



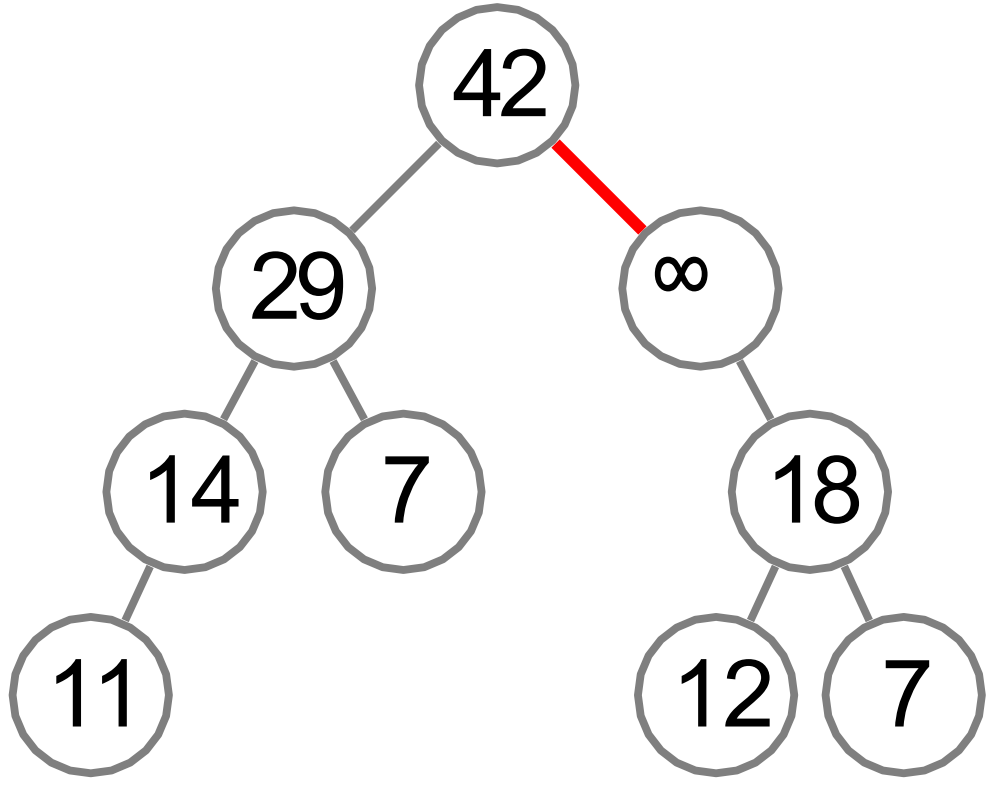
Remove



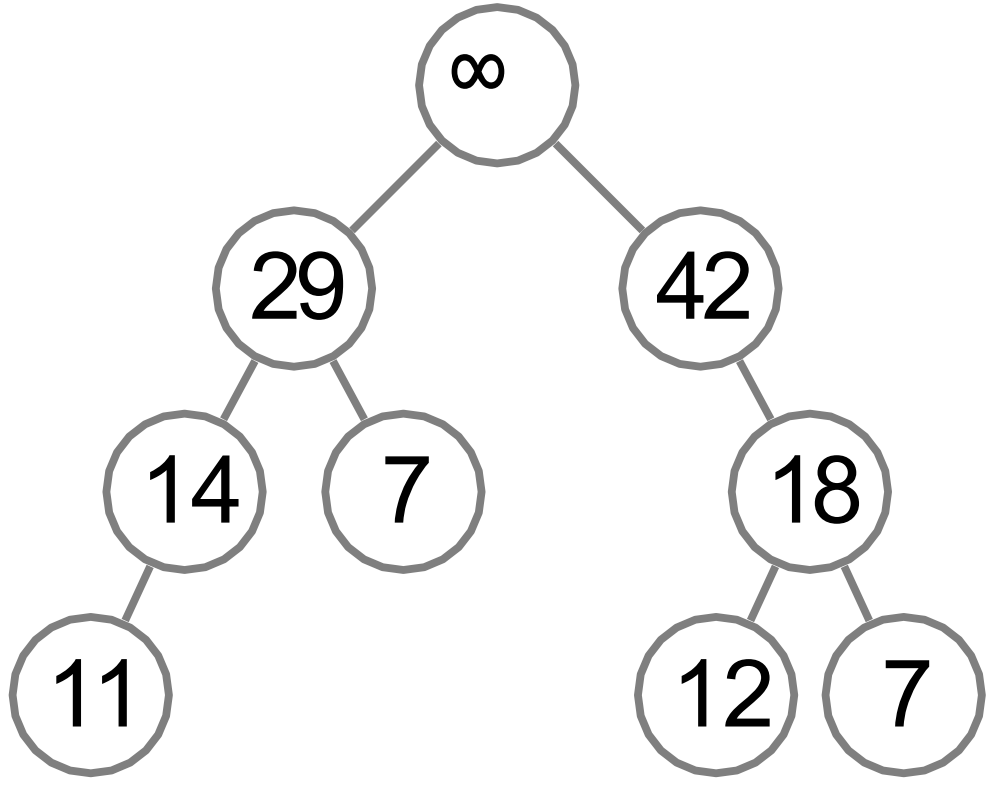
Remove



Remove



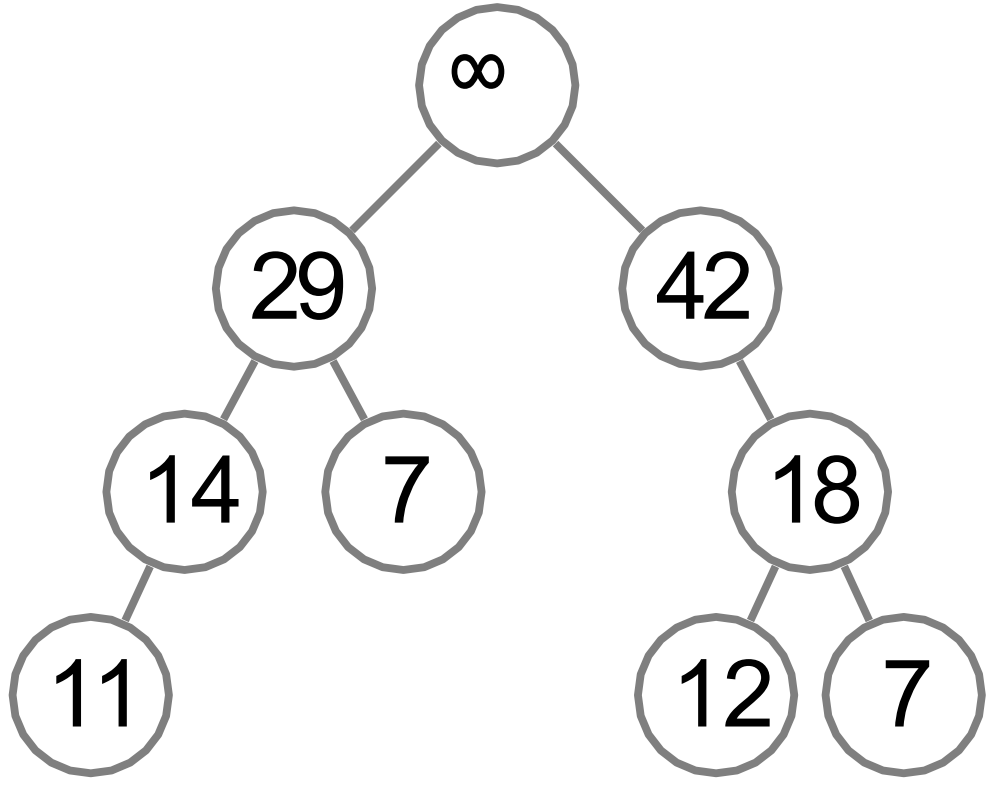
Remove



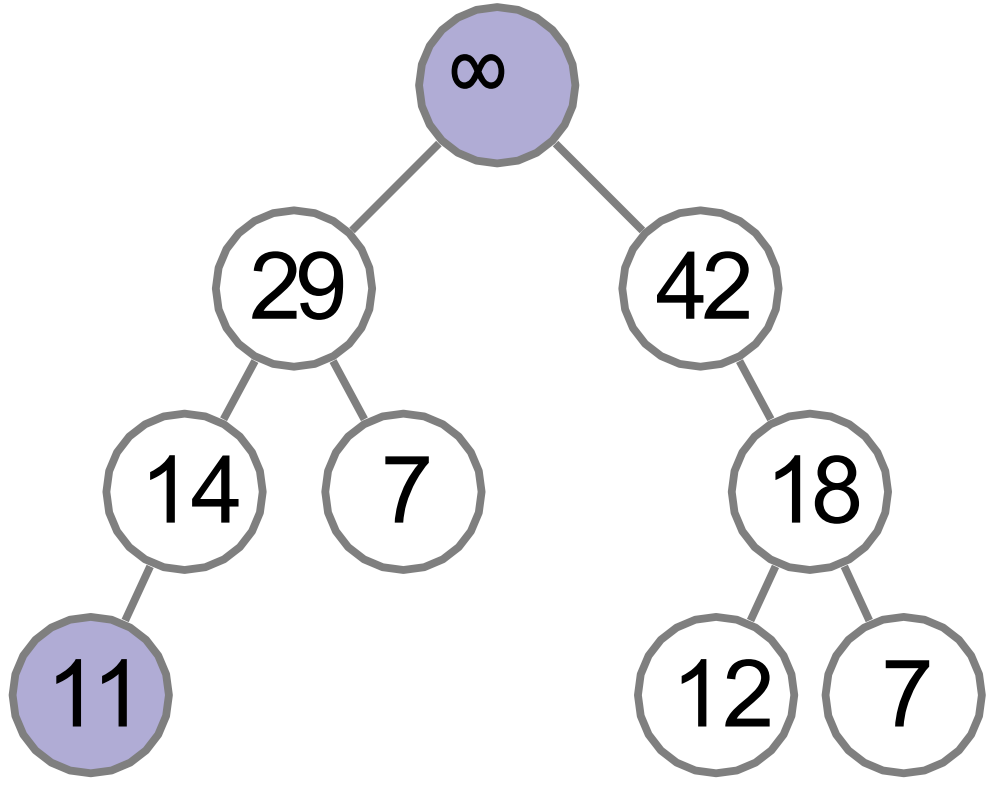
Remove

now, call

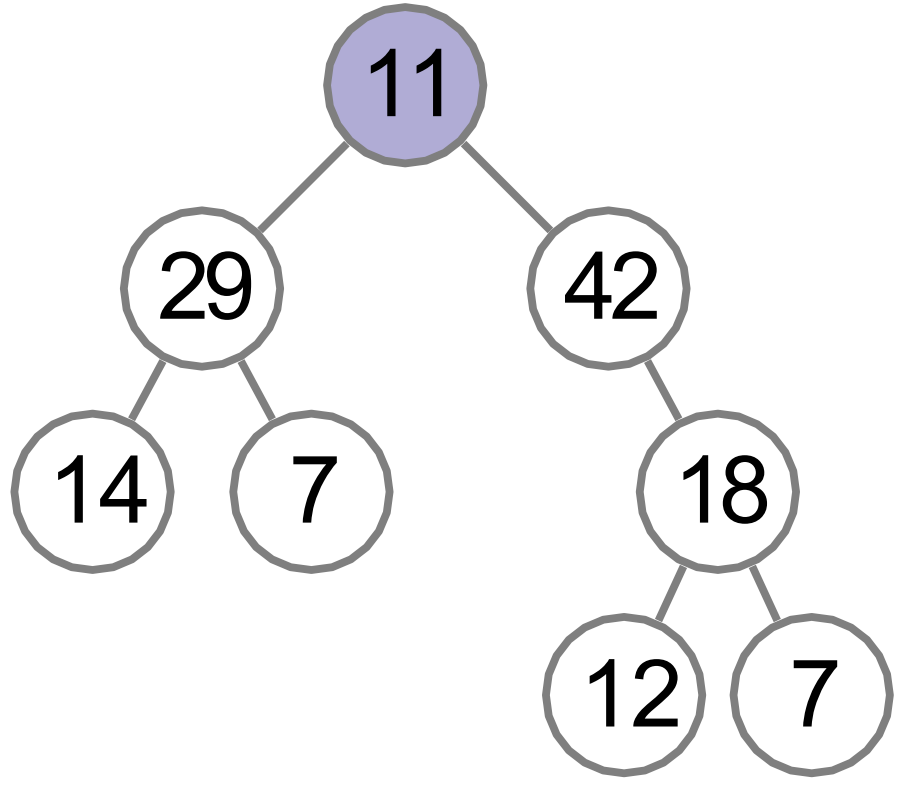
`ExtractMax()`



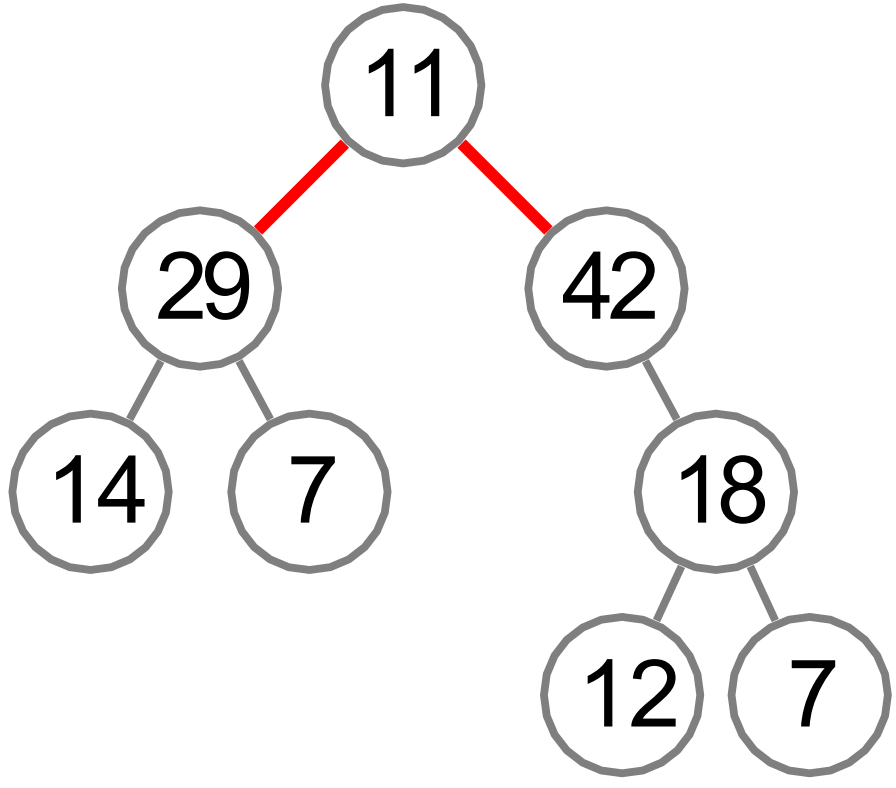
Remove



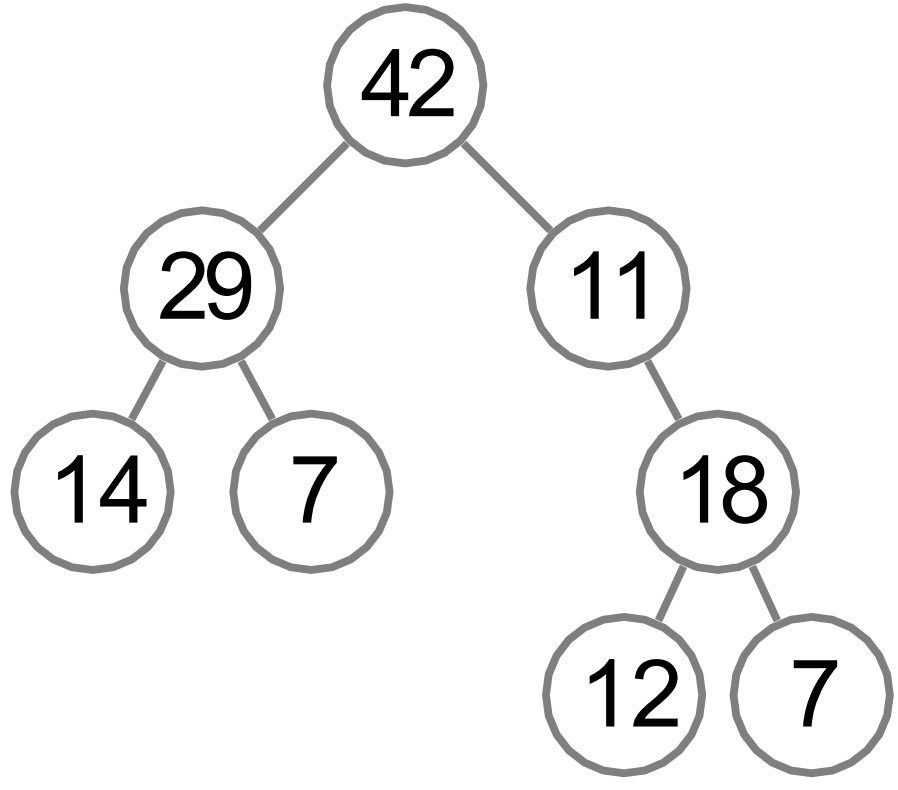
Remove



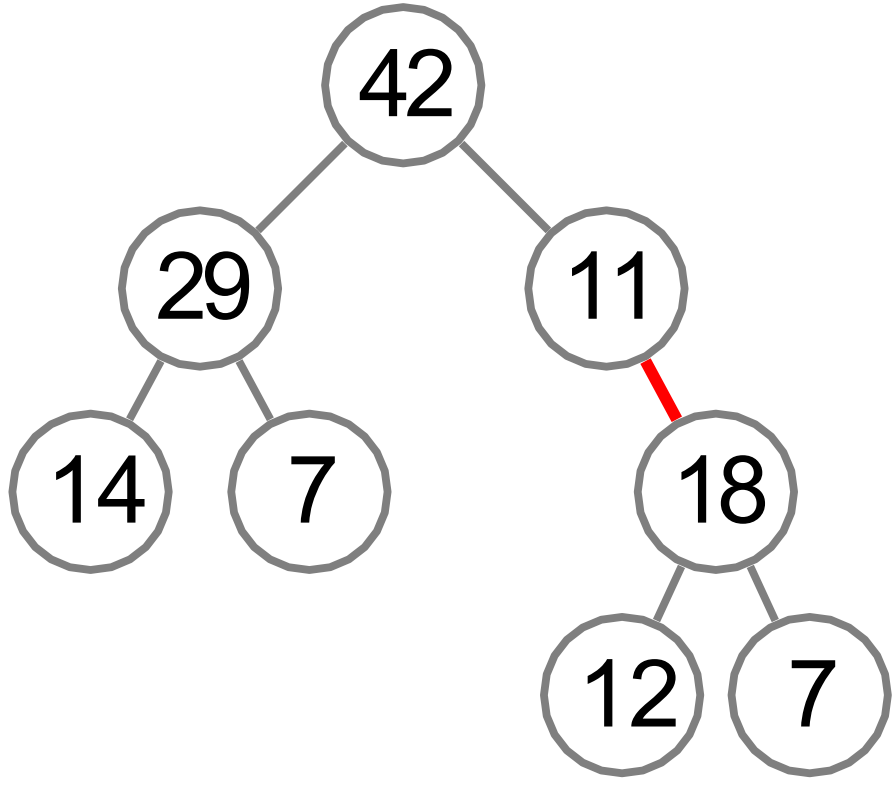
Remove



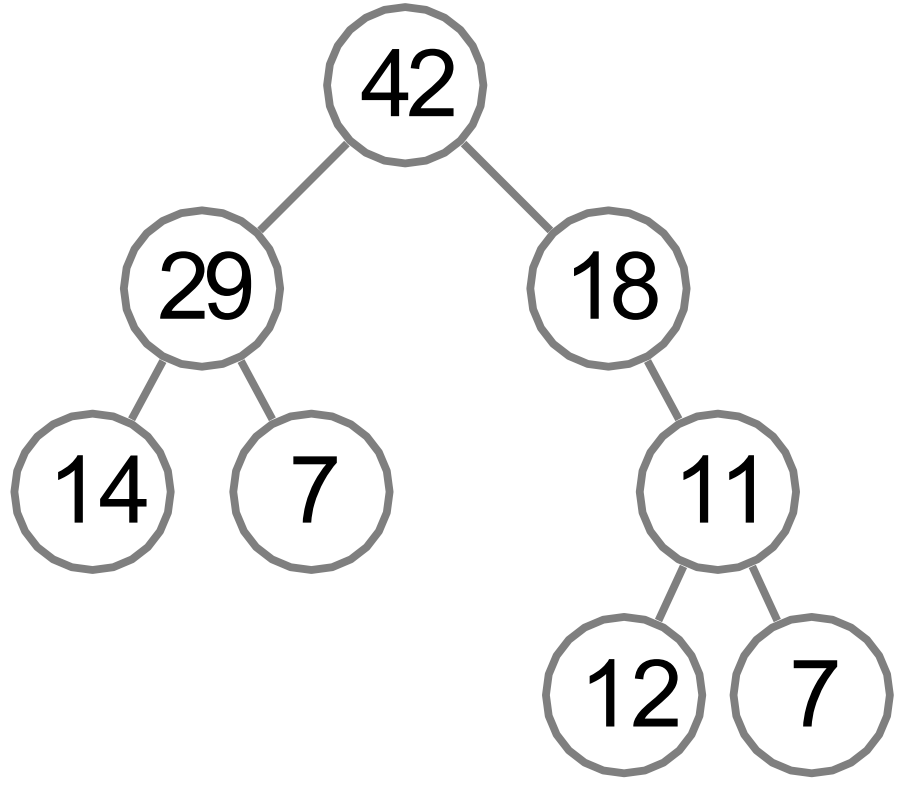
Remove



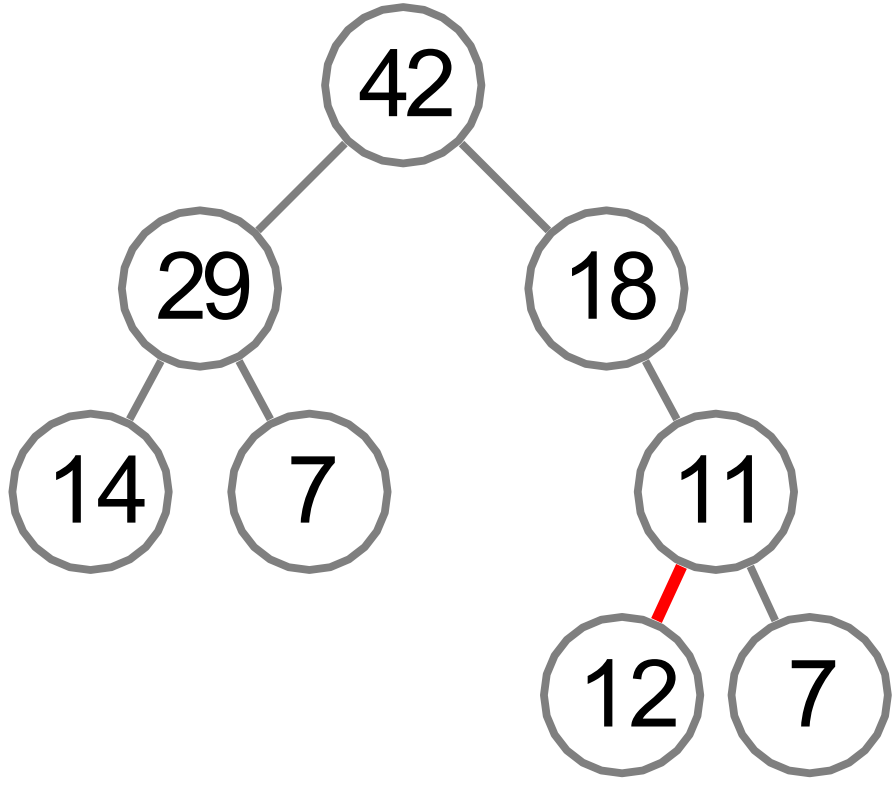
Remove



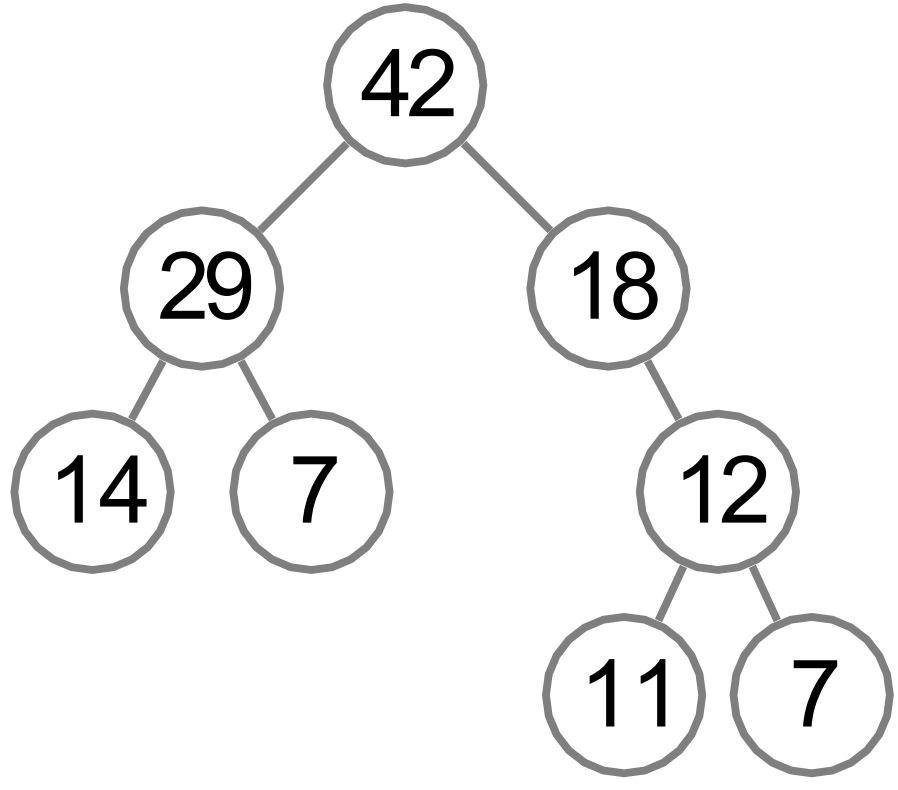
Remove



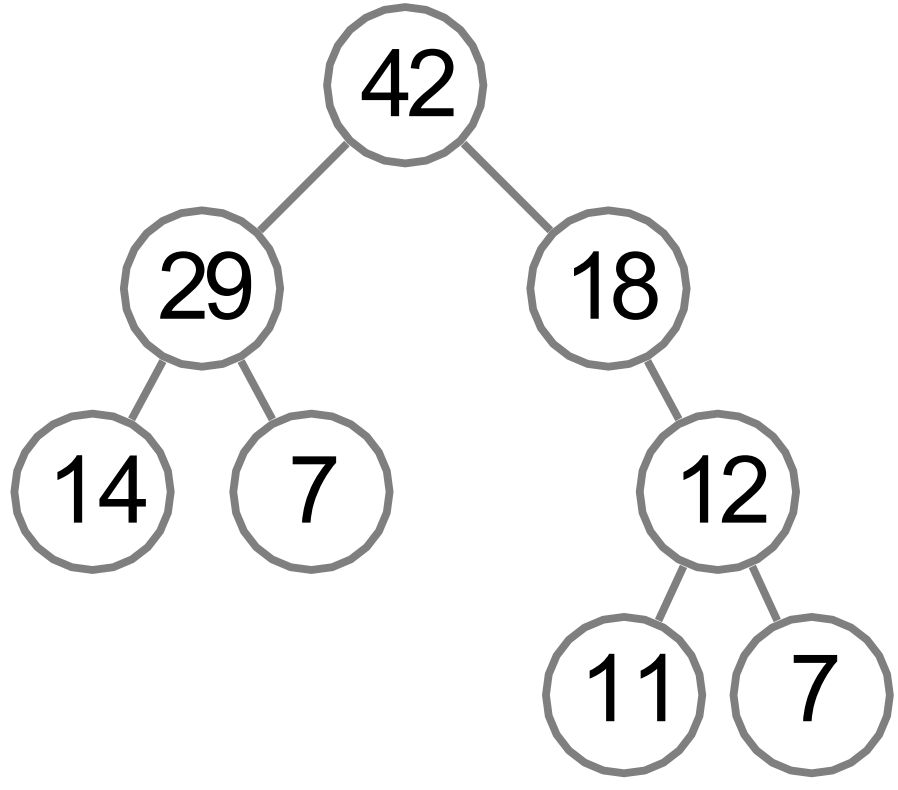
Remove



Remove



Remove



running time: O(tree height)

Summary

- `GetMax` works in time $O(1)$, all other operations work in time O(tree height)

Summary

- `GetMax` works in time $O(1)$, all other operations work in time O(tree height)
- we definitely want a tree to be shallow

Outline

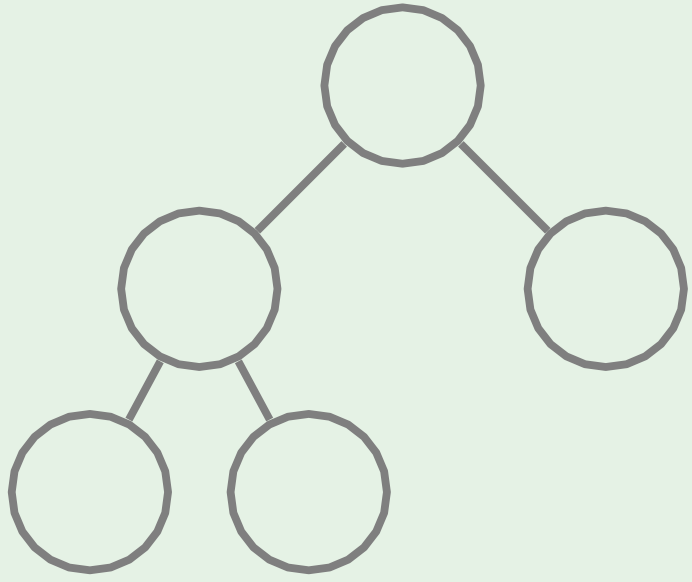
1. Binary Trees
2. Basic Operations
3. Complete Binary Trees
4. Pseudocode
5. Heap Sort
6. Final Remarks

How to Keep a Tree Shallow?

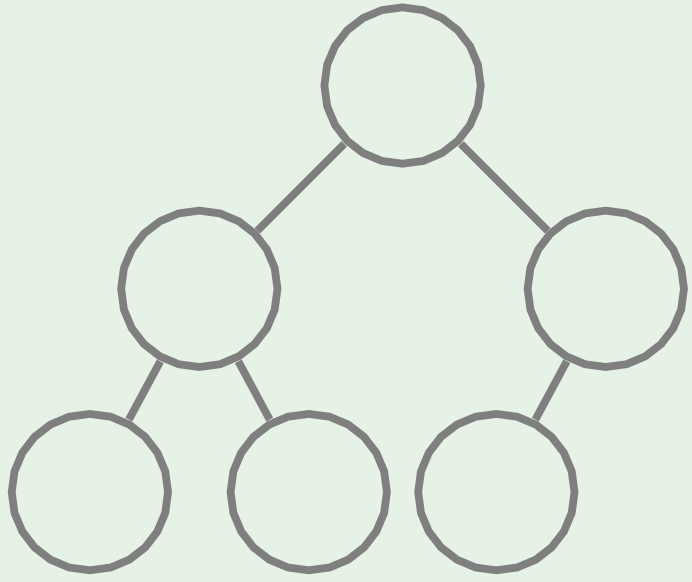
Definition

A binary tree is complete if all its levels are filled except possibly the last one which is filled from left to right.

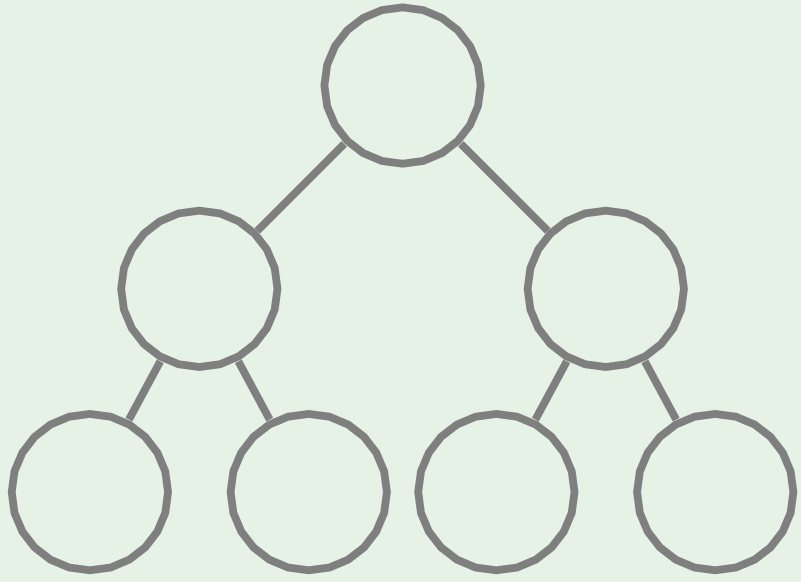
Example: complete binary tree



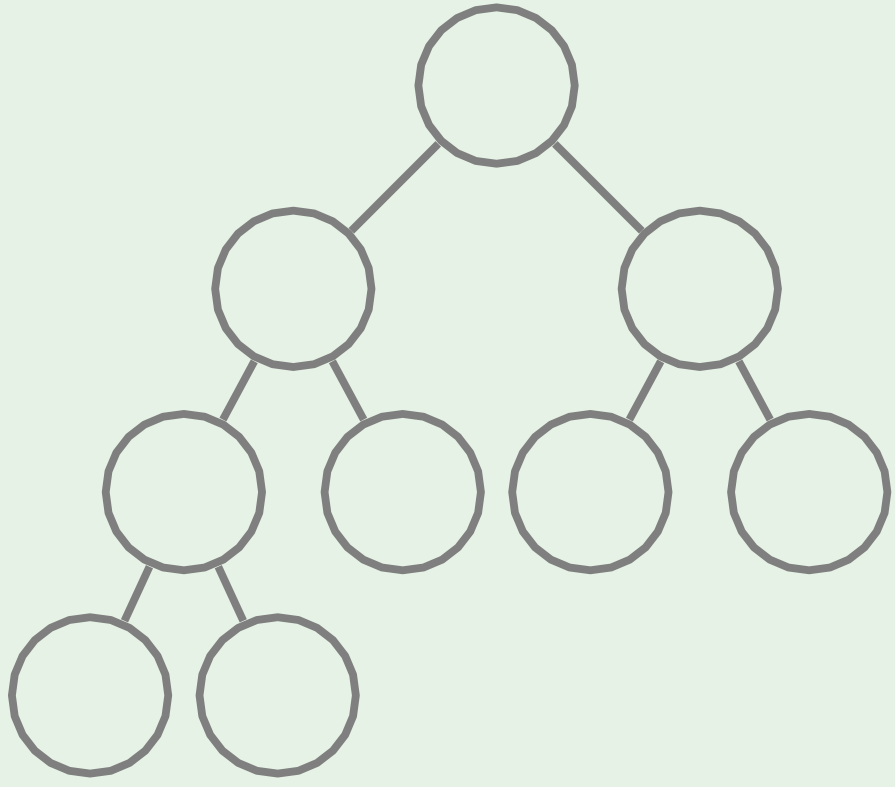
Example: complete binary tree



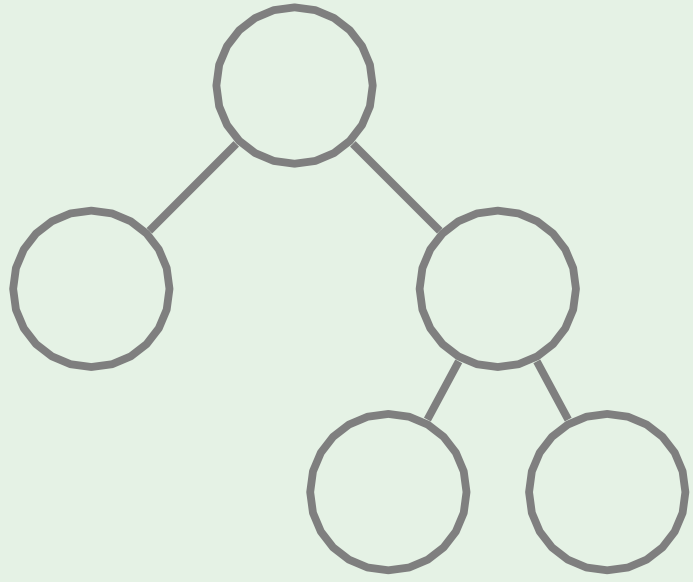
Example: complete binary tree



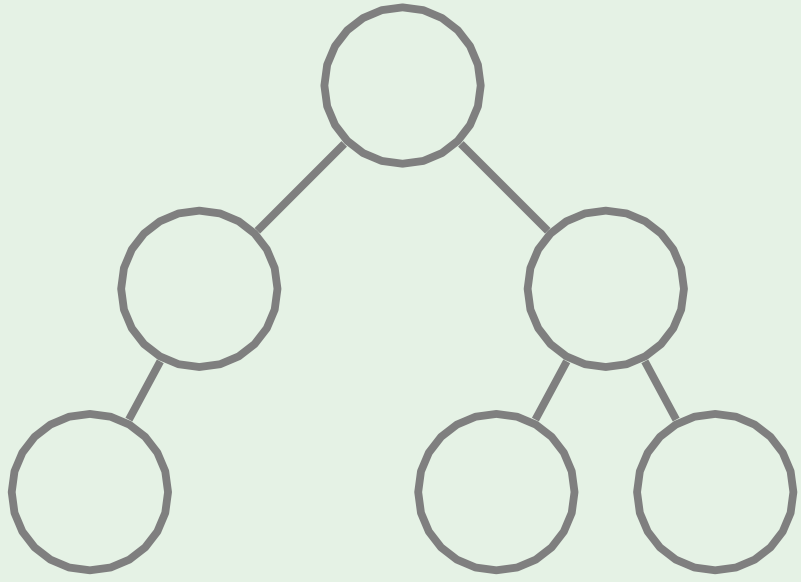
Example: complete binary tree



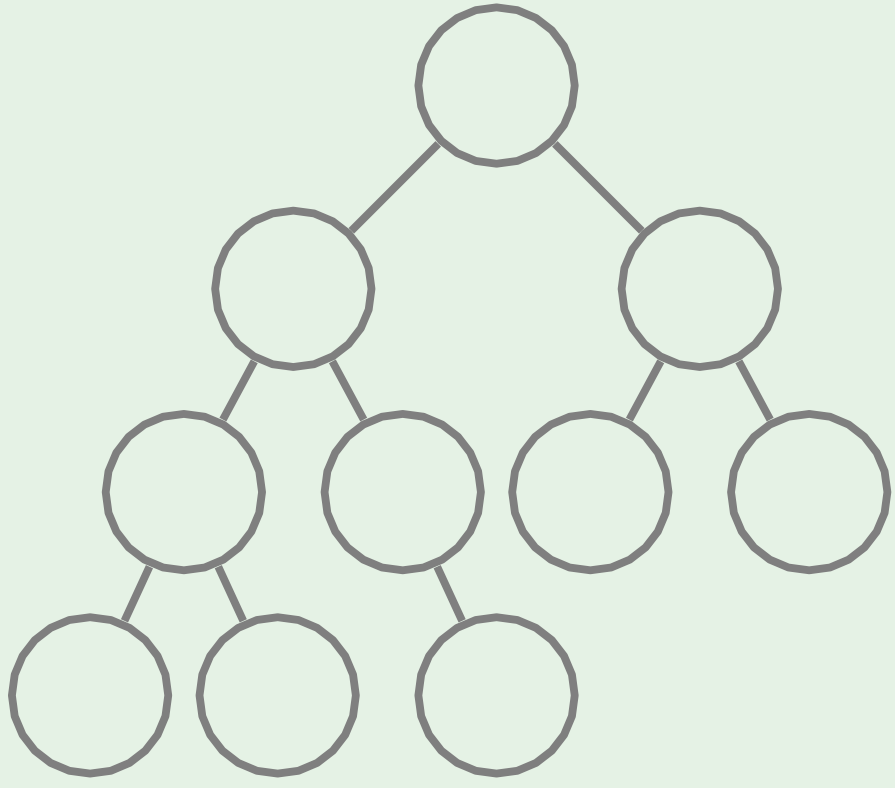
Example: not complete binary tree



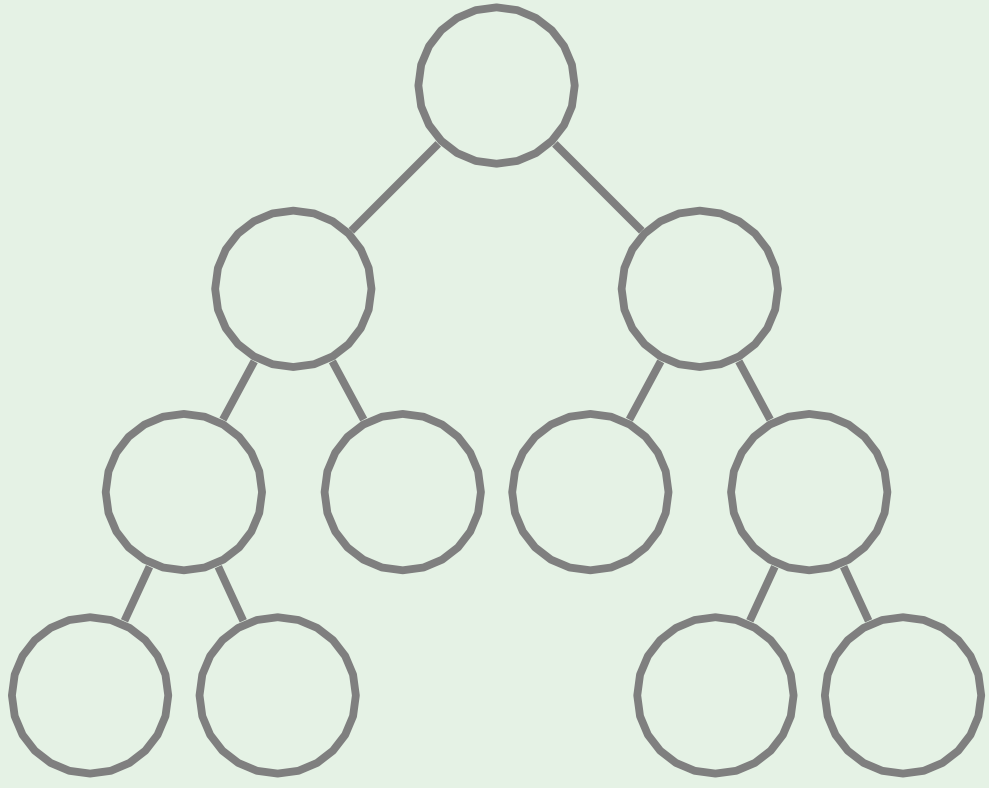
Example: not complete binary tree



Example: not complete binary tree



Example: not complete binary tree

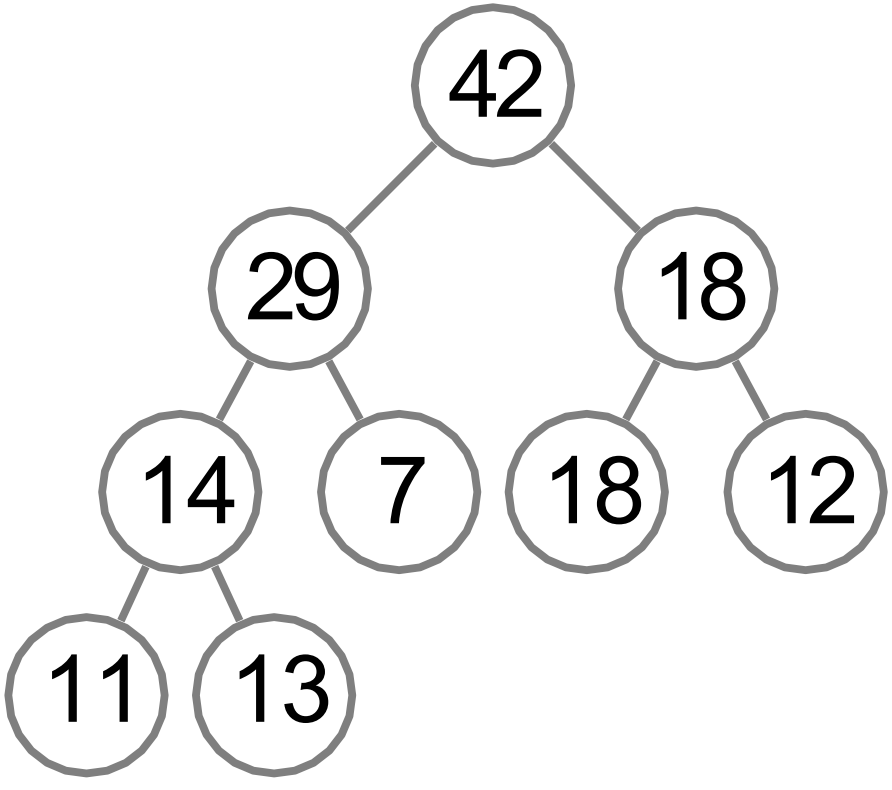


First Advantage: Low Height

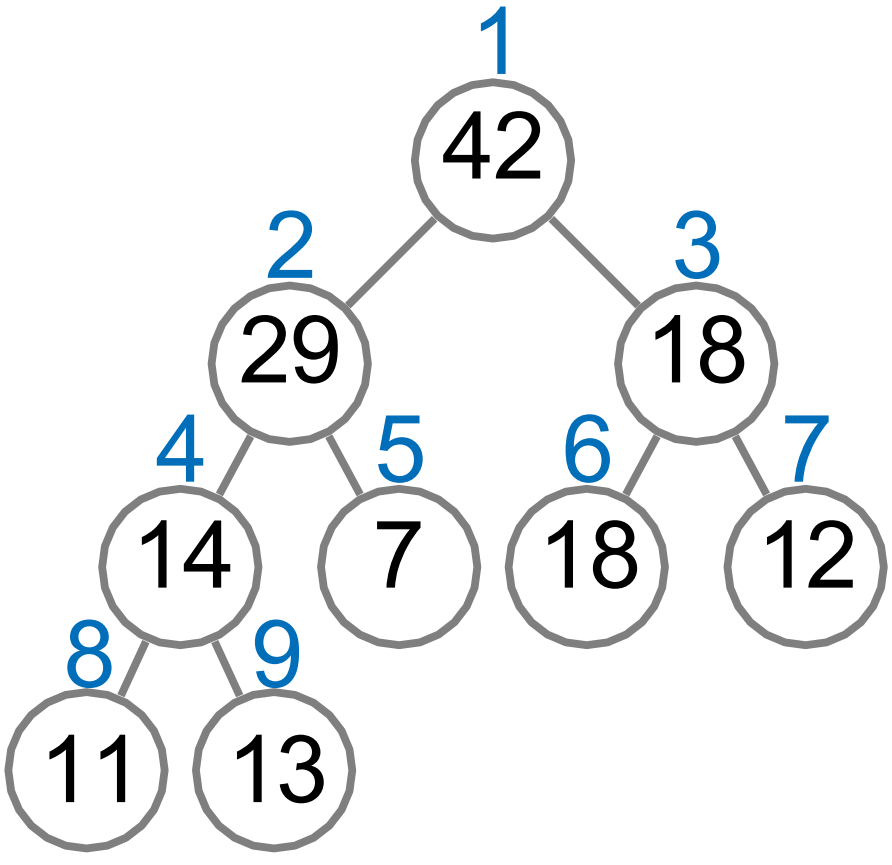
Lemma

A complete binary tree with n nodes has height at most $O(\log n)$.

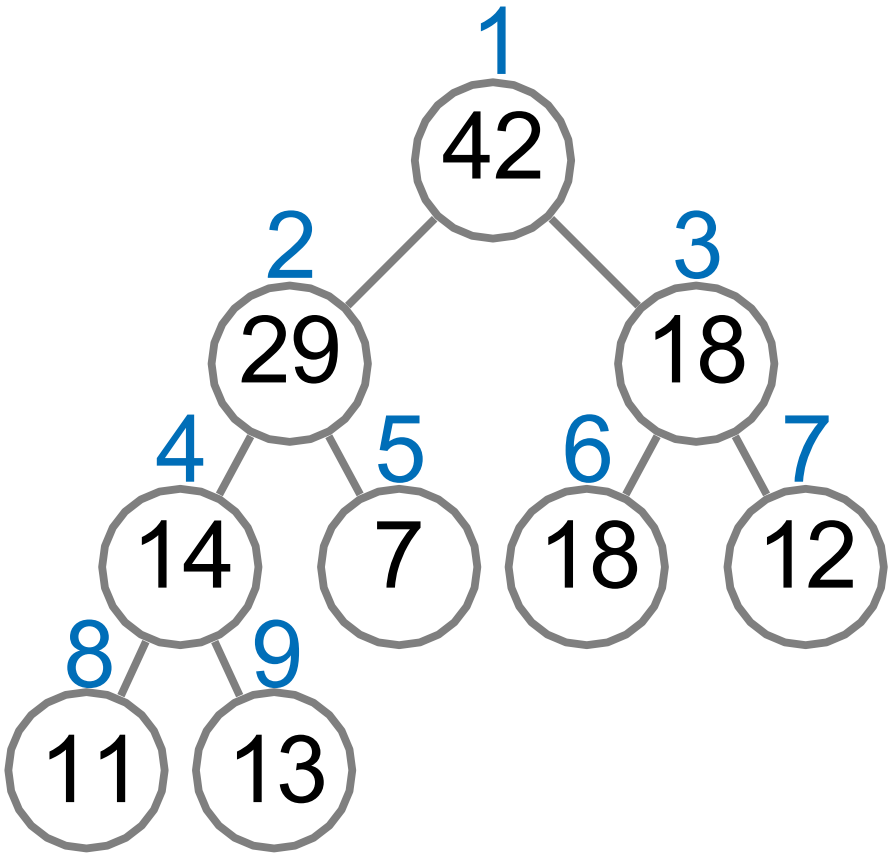
Second Advantage: Store as Array



Second Advantage: Store as Array



Second Advantage: Store as Array

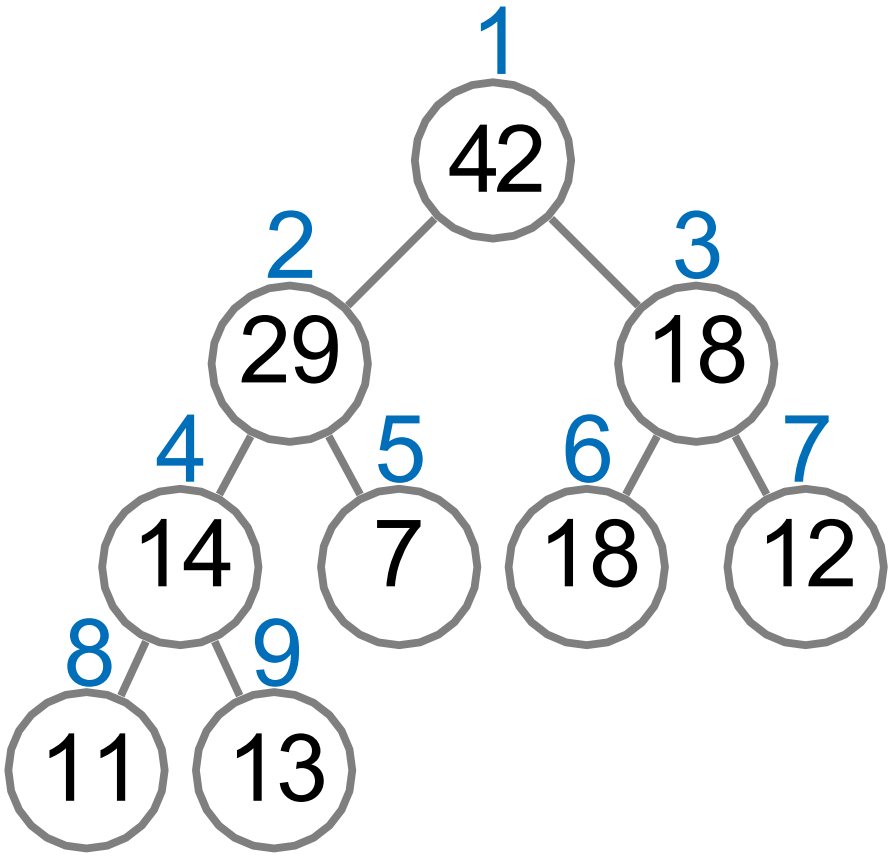


parent(i) = $\lfloor \frac{i}{2} \rfloor$

leftchild(i) = $2i$

rightchild(i) = $2i + 1$

Second Advantage: Store as Array



parent(i) = $\lfloor \frac{i}{2} \rfloor$

leftchild(i) = $2i$

rightchild(i) = $2i + 1$

1	2	3	4	5	6	7	8	9
42	29	18	14	7	18	12	11	13

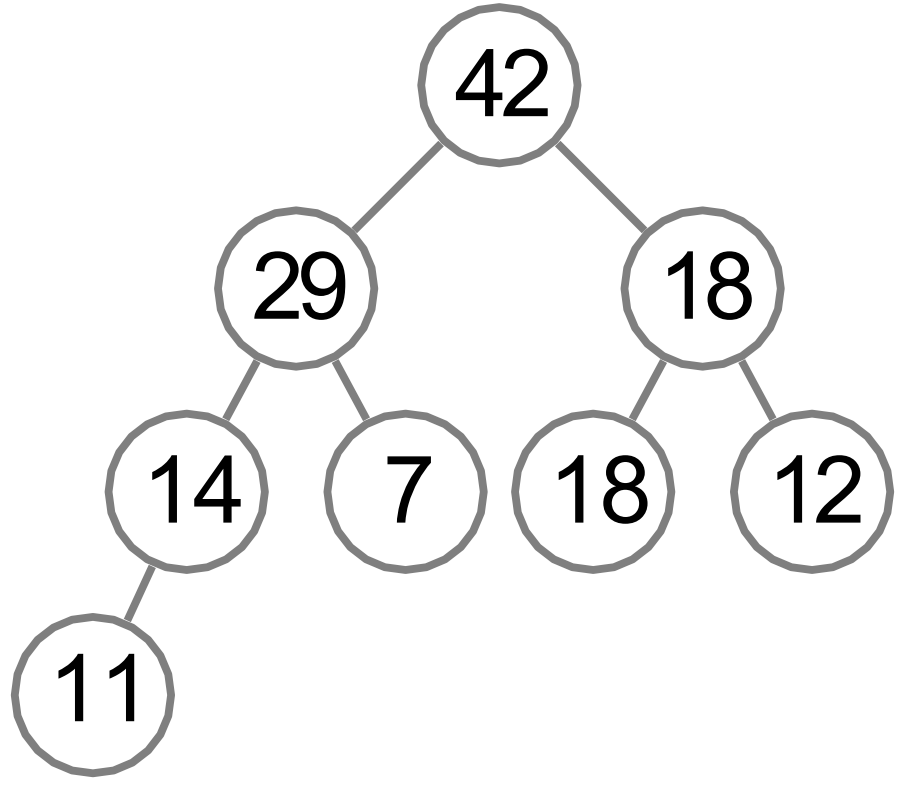
- What do we pay for these advantages?

- What do we pay for these advantages?
- We need to keep the tree complete.

- What do we pay for these advantages?
- We need to keep the tree complete.
- Which binary heap operations modify the shape of the tree?

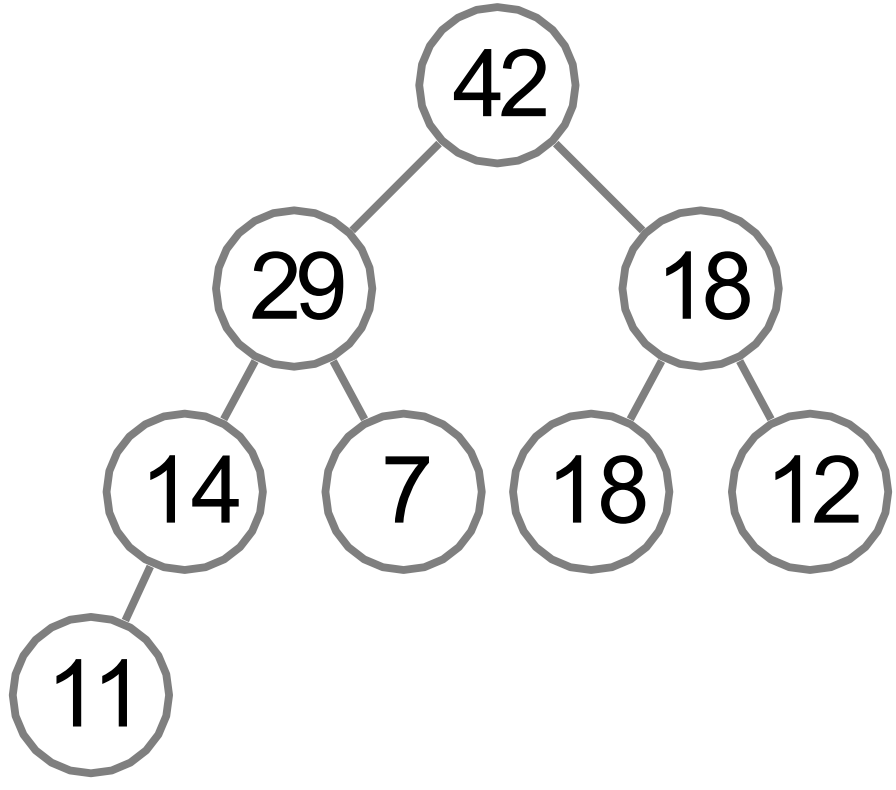
- What do we pay for these advantages?
- We need to keep the tree complete.
- Which binary heap operations modify the shape of the tree?
- Only `Insert` and `ExtractMax` (`Remove` changes the shape by calling `ExtractMax`).

Keeping the Tree Complete



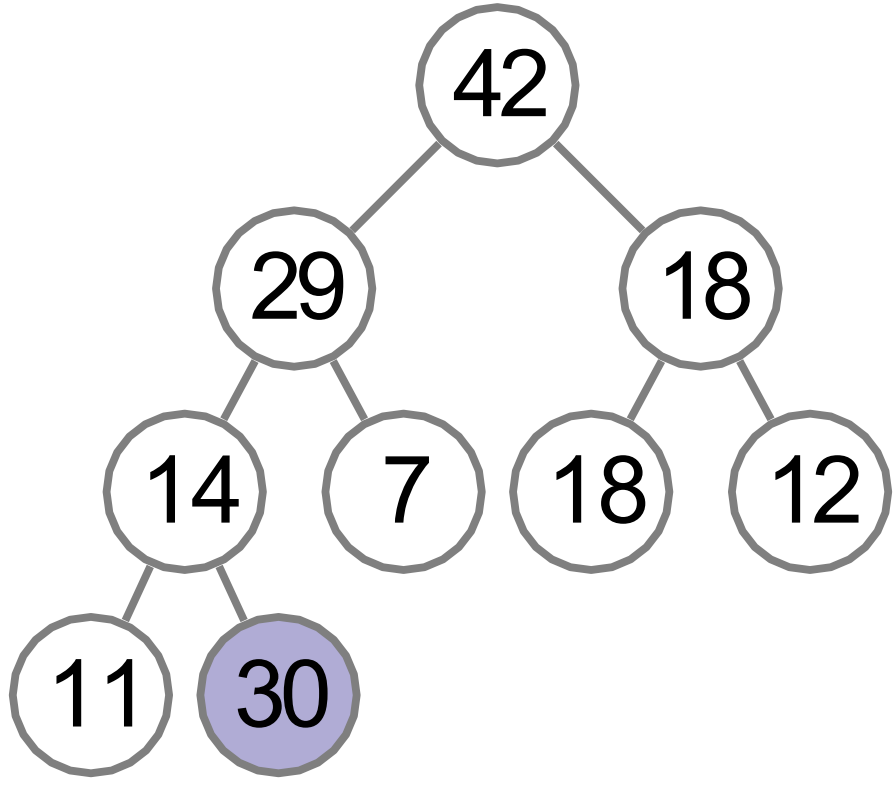
Keeping the Tree Complete

to insert an element, insert it as a leaf in the leftmost vacant position in the last level and let it sift up



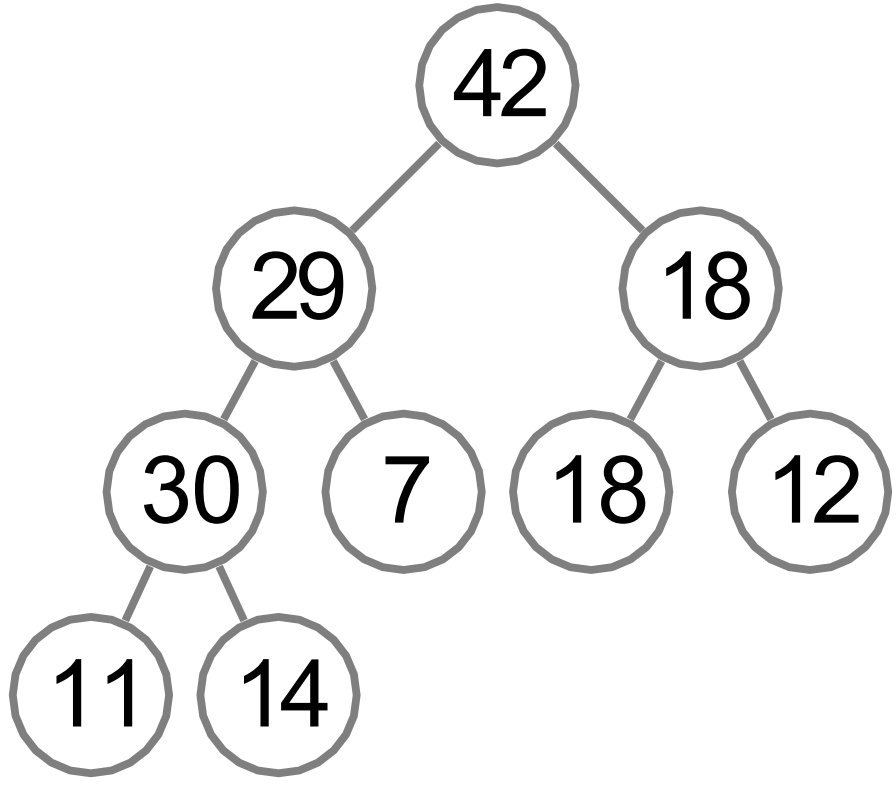
Keeping the Tree Complete

to insert an element, insert it as a leaf in the leftmost vacant position in the last level and let it sift up



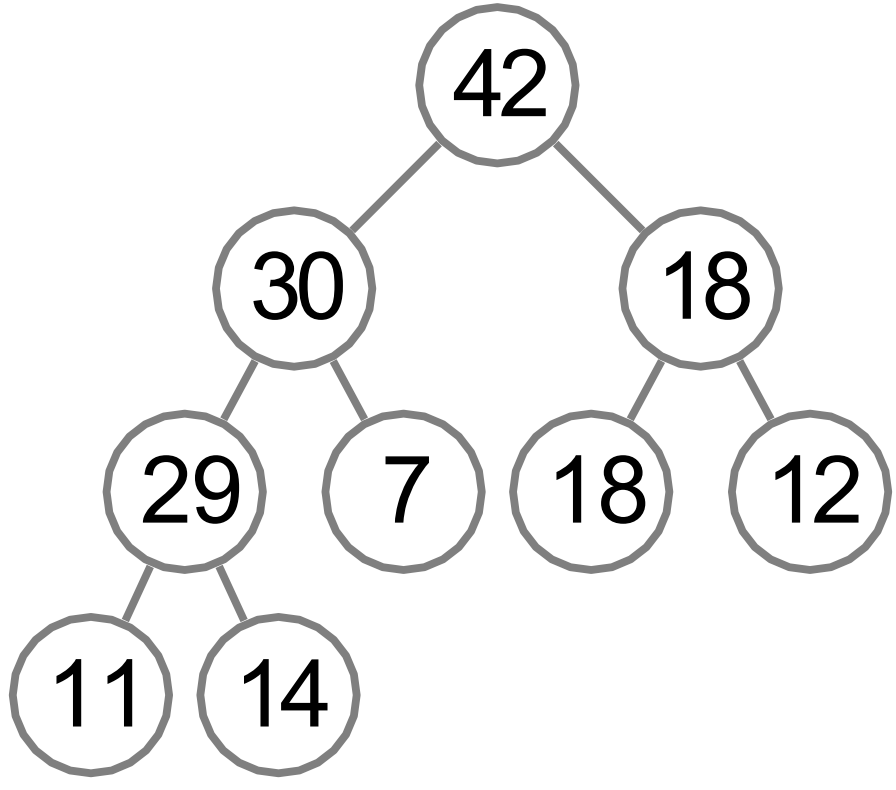
Keeping the Tree Complete

to insert an element, insert it as a leaf in the leftmost vacant position in the last level and let it sift up



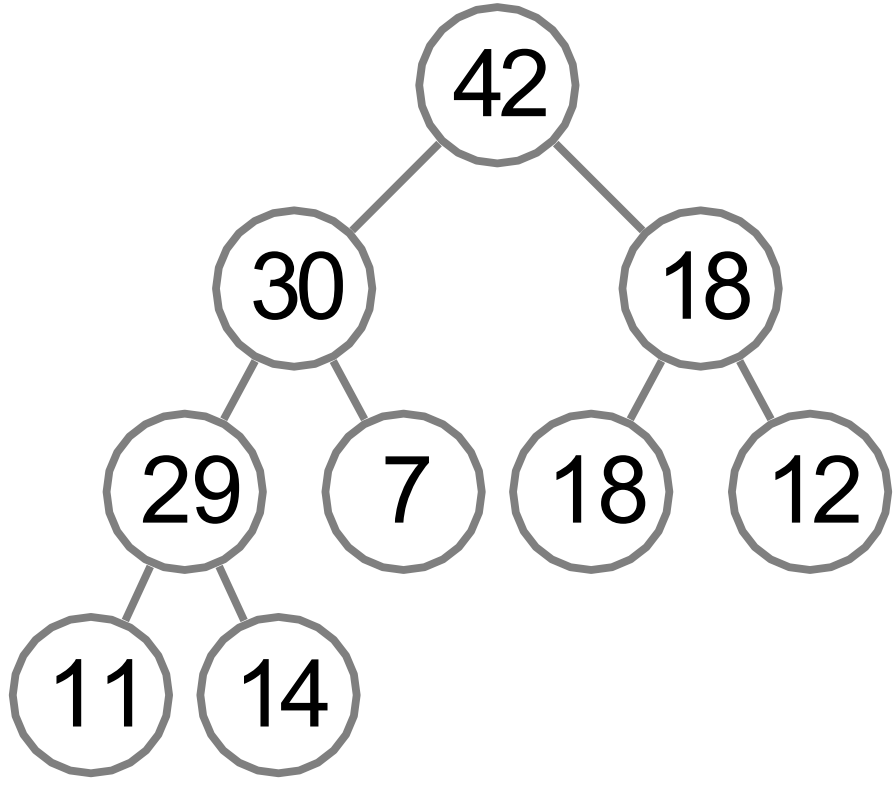
Keeping the Tree Complete

to insert an element, insert it as a leaf in the leftmost vacant position in the last level and let it sift up



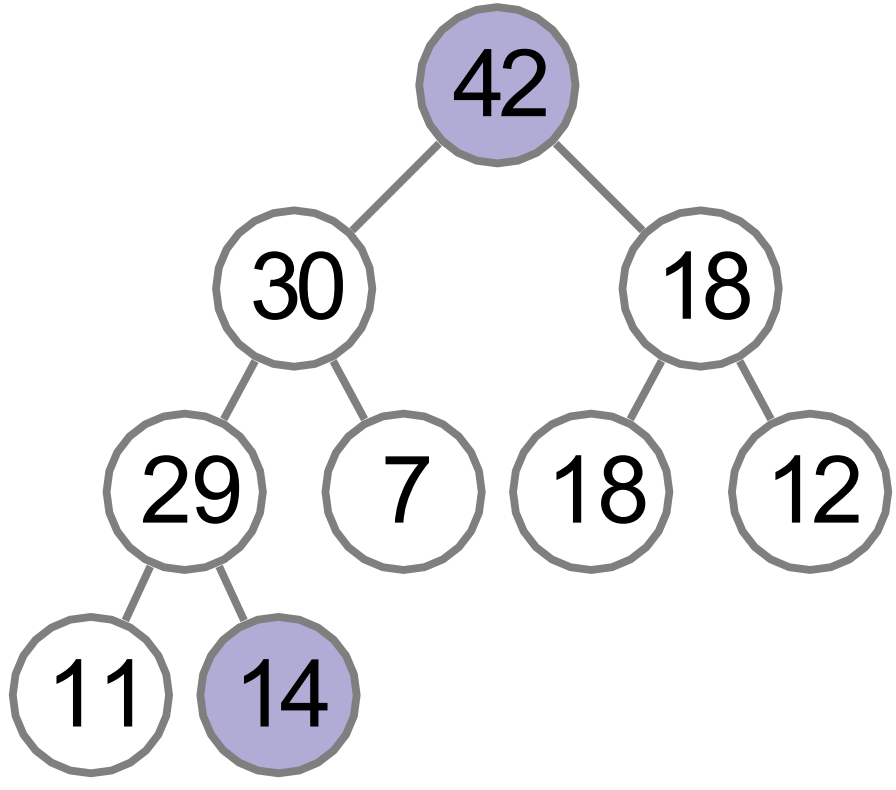
Keeping the Tree Complete

to extract the maximum value, replace the root by the last leaf and let it sift down



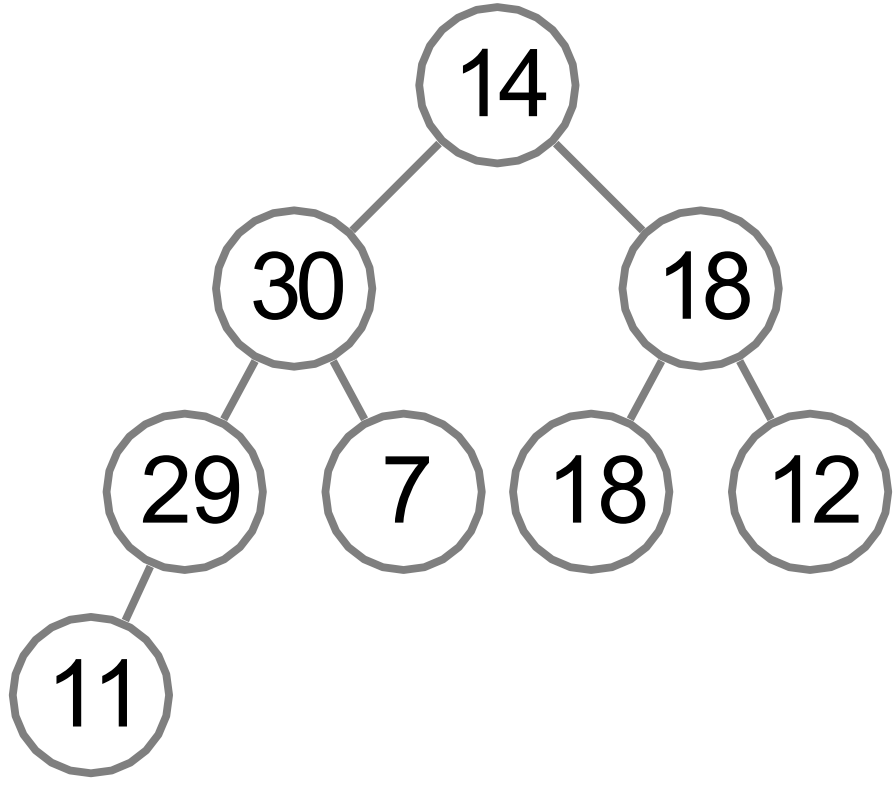
Keeping the Tree Complete

to extract the maximum value, replace the root by the last leaf and let it sift down



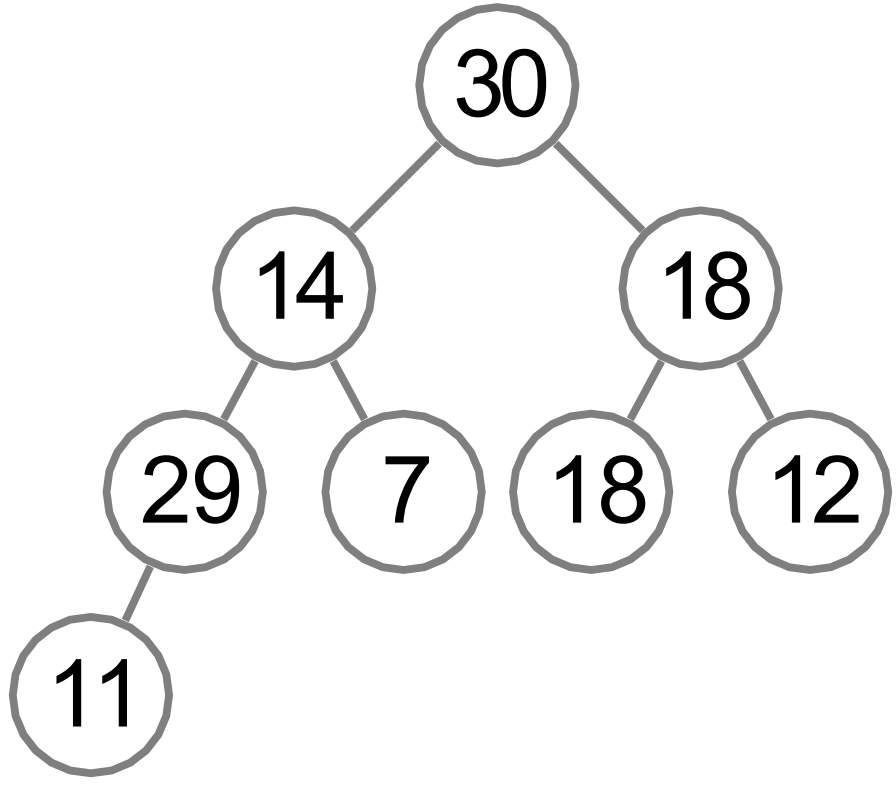
Keeping the Tree Complete

to extract the maximum value, replace the root by the last leaf and let it sift down



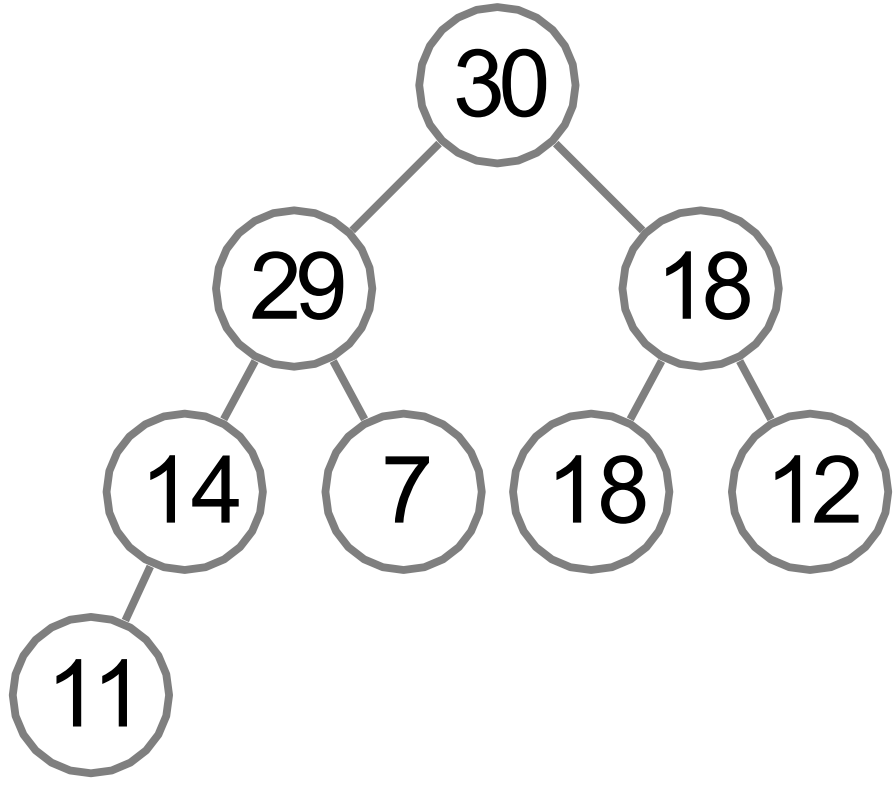
Keeping the Tree Complete

to extract the maximum value, replace the root by the last leaf and let it sift down



Keeping the Tree Complete

to extract the maximum value, replace the root by the last leaf and let it sift down

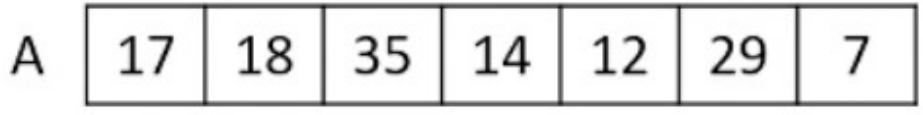


Outline

1. Binary Trees
2. Basic Operations
3. Complete Binary Trees
4. Pseudocode
5. Heap Sort
6. Final Remarks

Sort Using Priority Queues

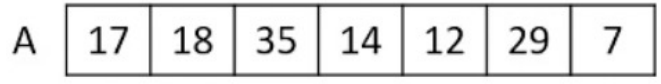
```
HeapSort(A[1...n])
create an empty priority queue
for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



Sort Using Priority Queues

HeapSort($A[1 \ldots n]$)

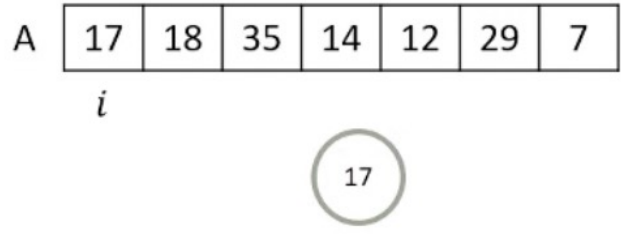
```
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```

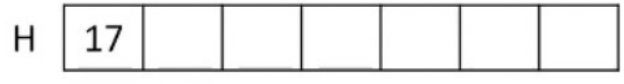


Sort Using Priority Queues

HeapSort($A[1 \ldots n]$)

```
create an empty priority queue
for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```

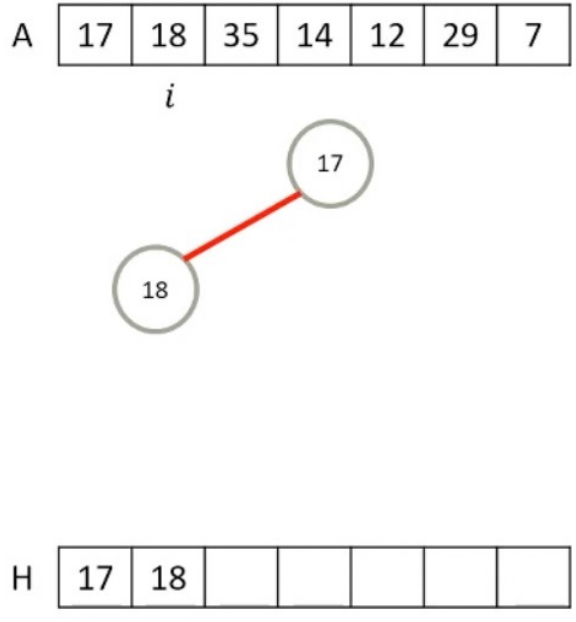




Sort Using Priority Queues

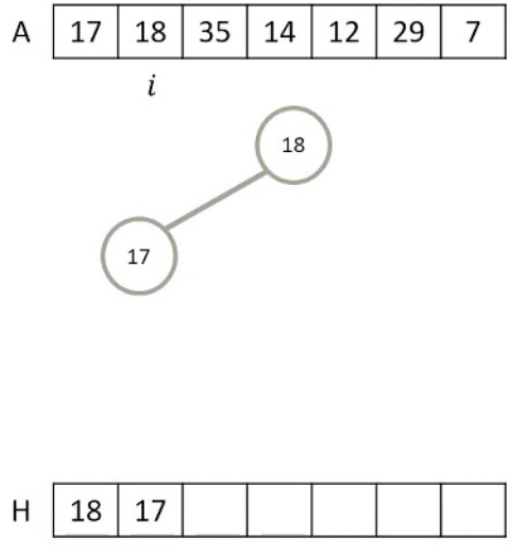
HeapSort($A[1 \ldots n]$)

```
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



Sort Using Priority Queues

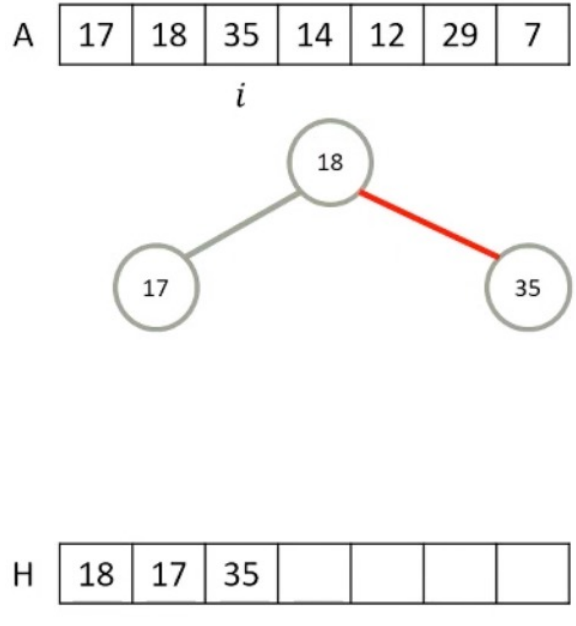
```
HeapSort(A[1...n])
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



Sort Using Priority Queues

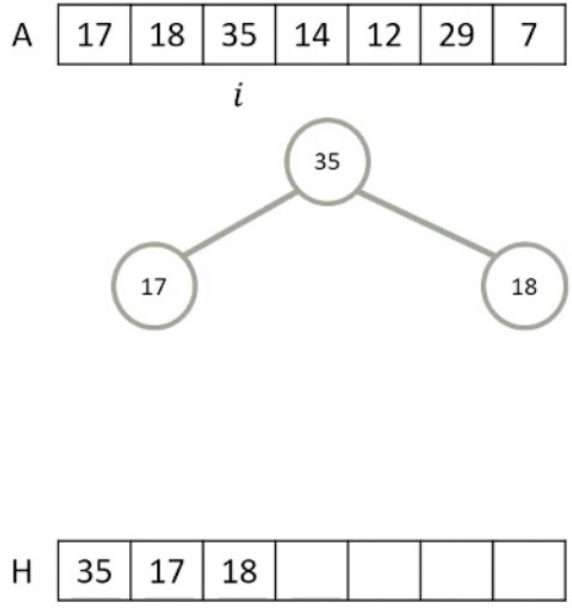
HeapSort($A[1 \ldots n]$)

```
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



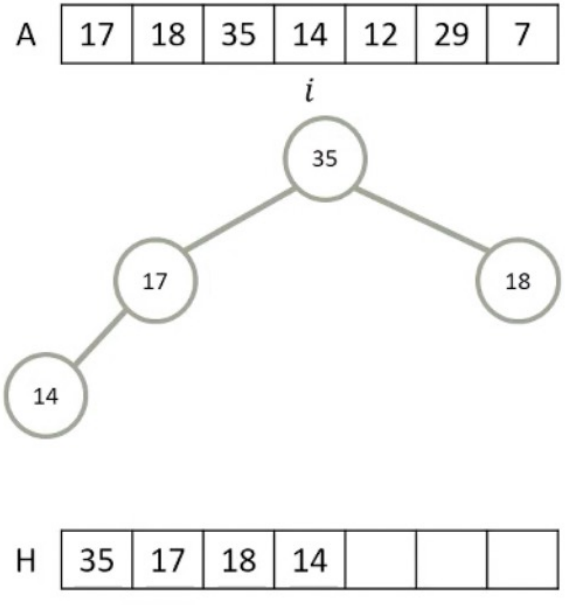
Sort Using Priority Queues

```
HeapSort(A[1...n])
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



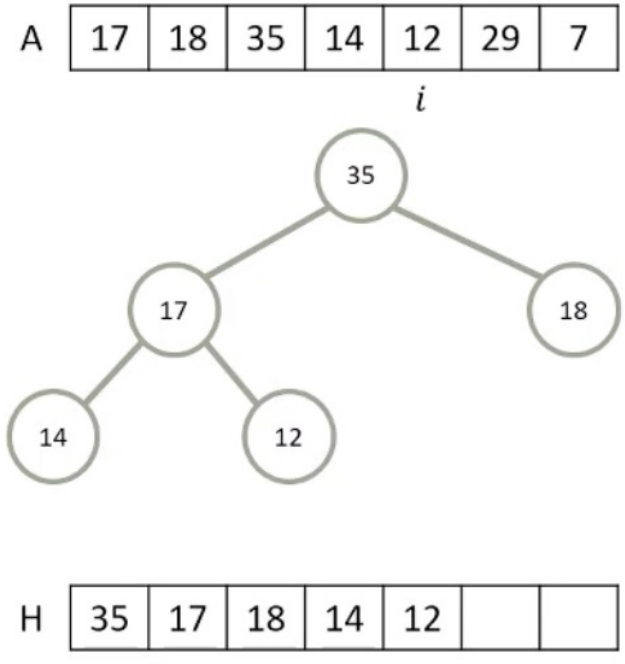
Sort Using Priority Queues

```
HeapSort(A[1...n])
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



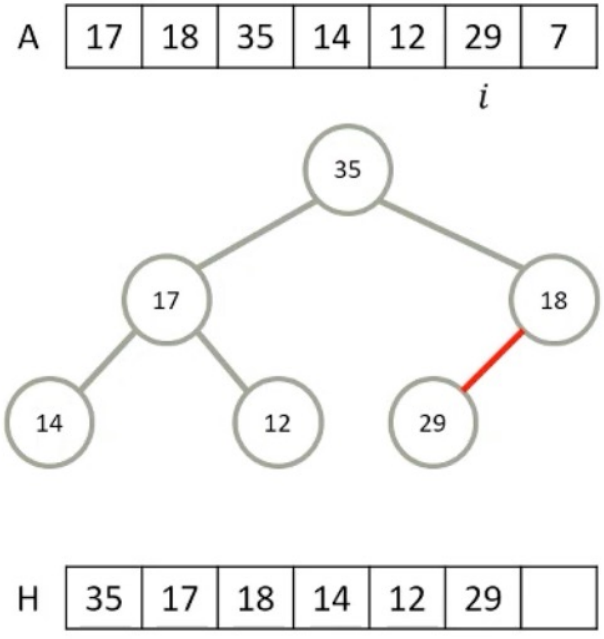
Sort Using Priority Queues

```
HeapSort(A[1...n])
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



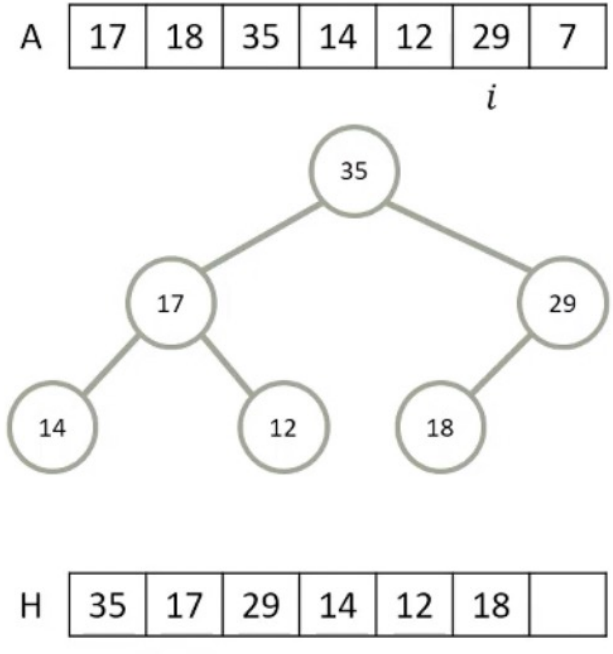
Sort Using Priority Queues

```
HeapSort(A[1...n])
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



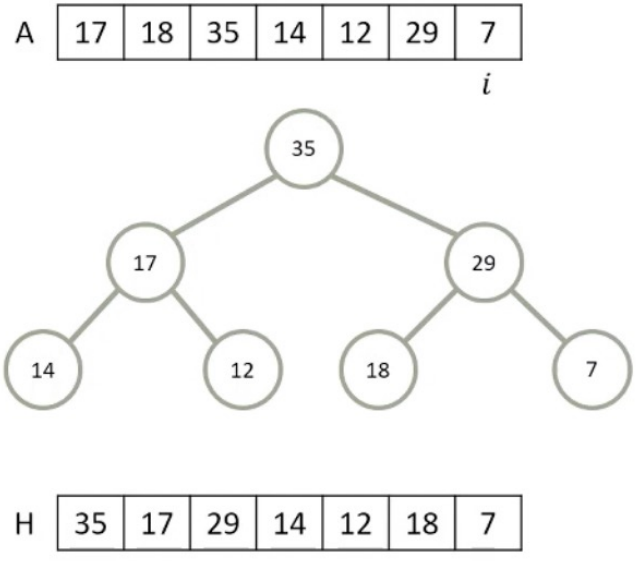
Sort Using Priority Queues

```
HeapSort(A[1...n])
create an empty priority queue
for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



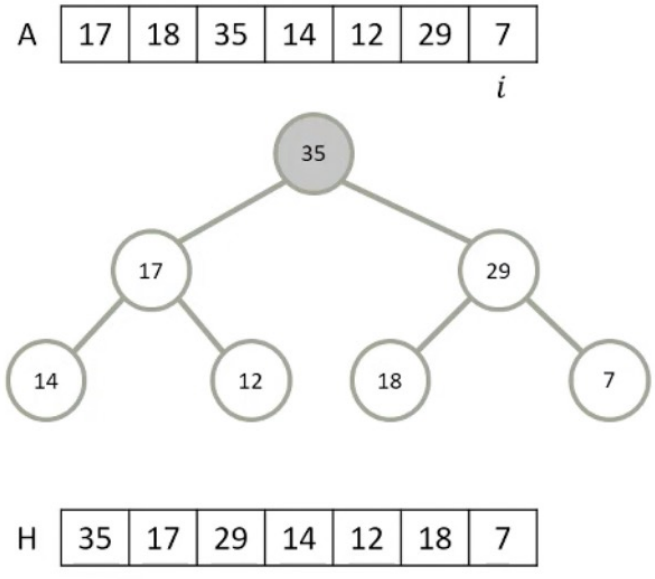
Sort Using Priority Queues

```
HeapSort(A[1...n])
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



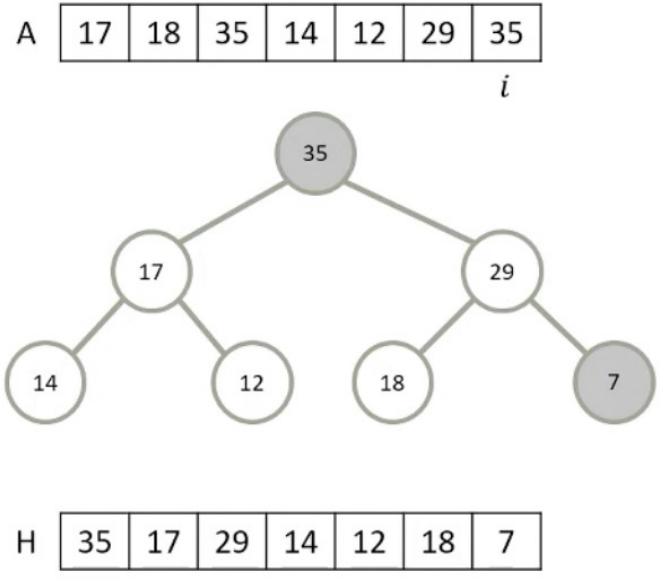
Sort Using Priority Queues

```
HeapSort(A[1...n])
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



Sort Using Priority Queues

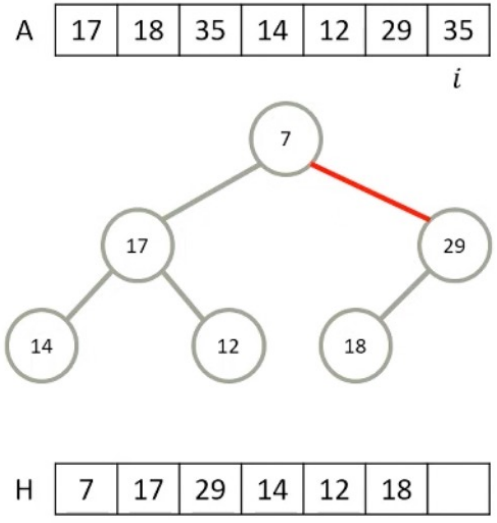
```
HeapSort(A[1...n])
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



Sort Using Priority Queues

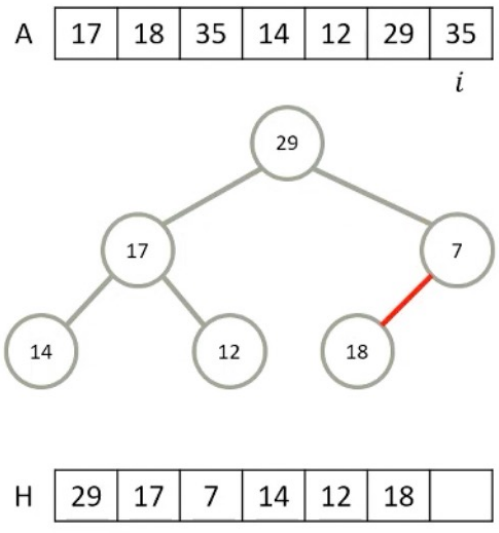
HeapSort($A[1 \ldots n]$)

```
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



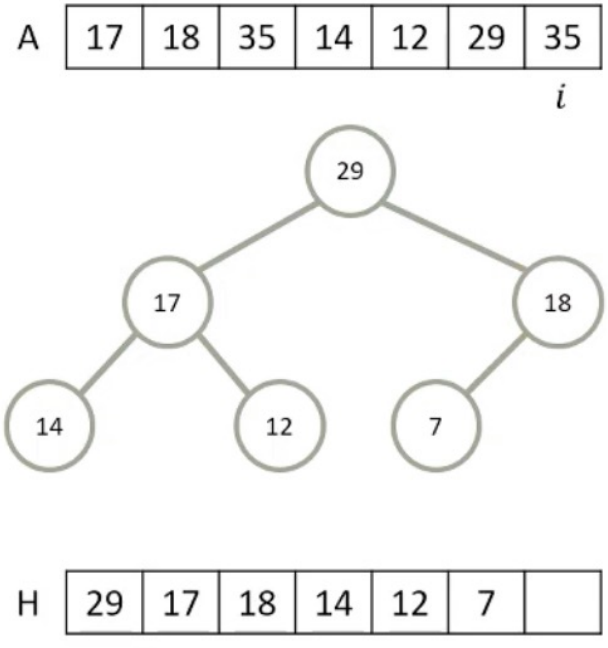
Sort Using Priority Queues

```
HeapSort(A[1...n])
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



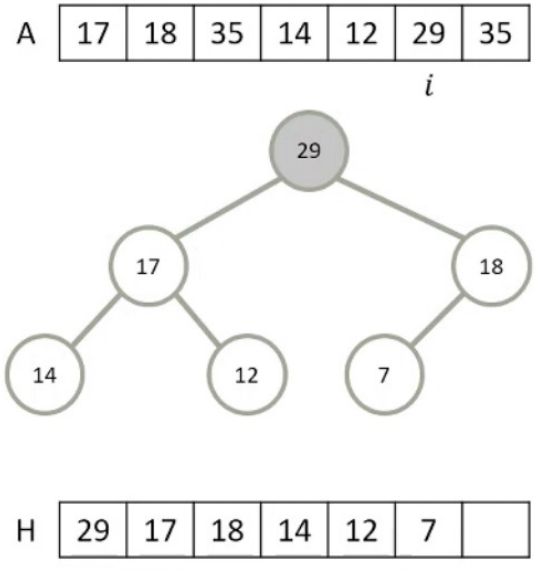
Sort Using Priority Queues

```
HeapSort(A[1...n])
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



Sort Using Priority Queues

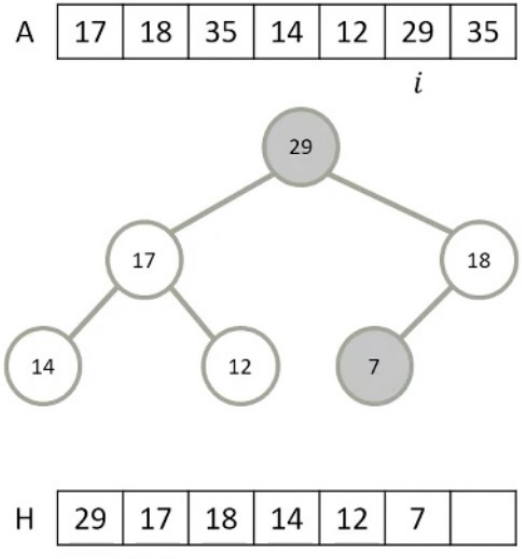
```
HeapSort(A[1...n])
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



Sort Using Priority Queues

HeapSort($A[1 \ldots n]$)

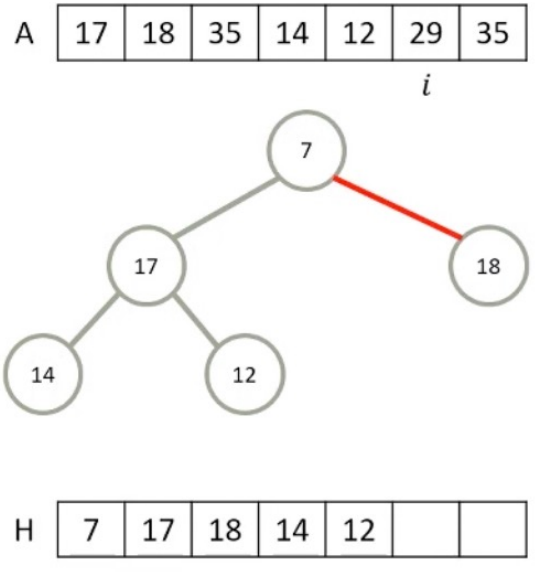
```
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



Sort Using Priority Queues

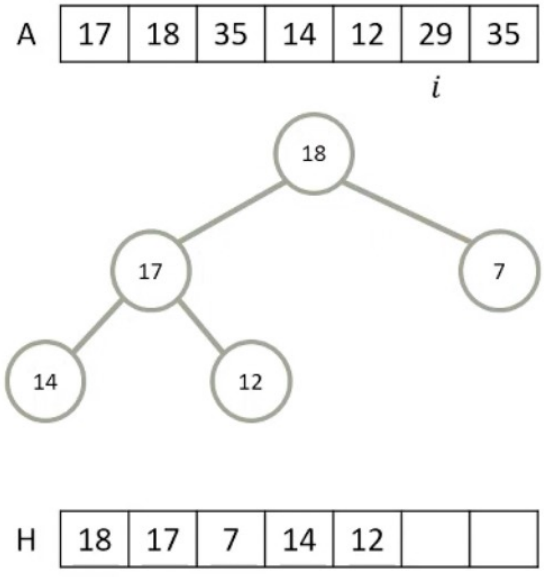
HeapSort($A[1 \ldots n]$)

```
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



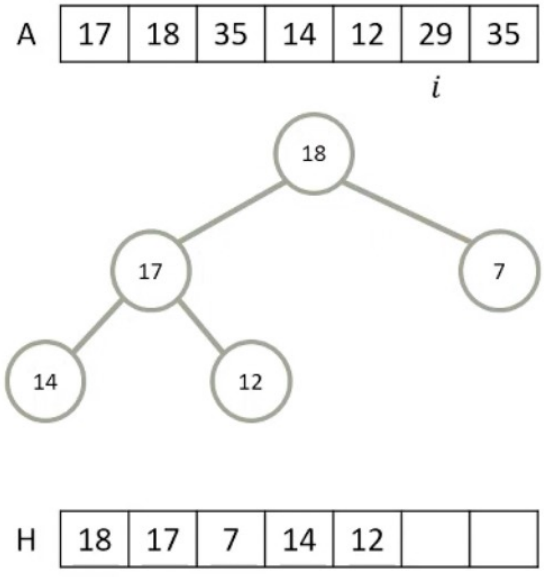
Sort Using Priority Queues

```
HeapSort(A[1...n])
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



Sort Using Priority Queues

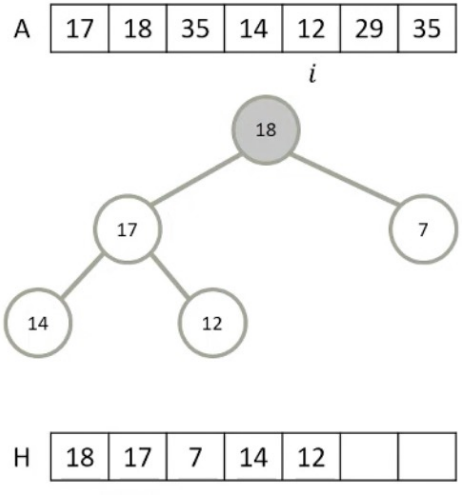
```
HeapSort(A[1...n])
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



Sort Using Priority Queues

HeapSort($A[1 \ldots n]$)

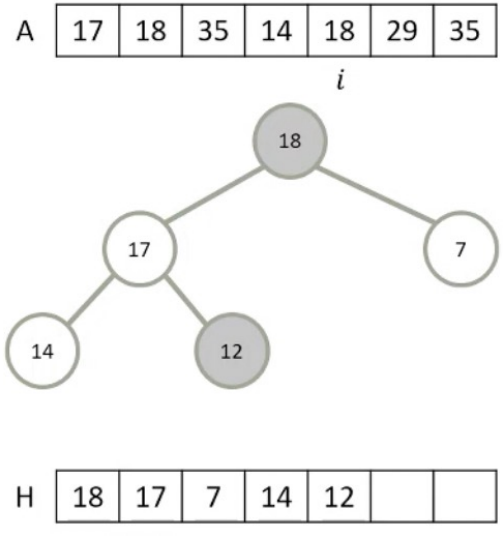
```
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



Sort Using Priority Queues

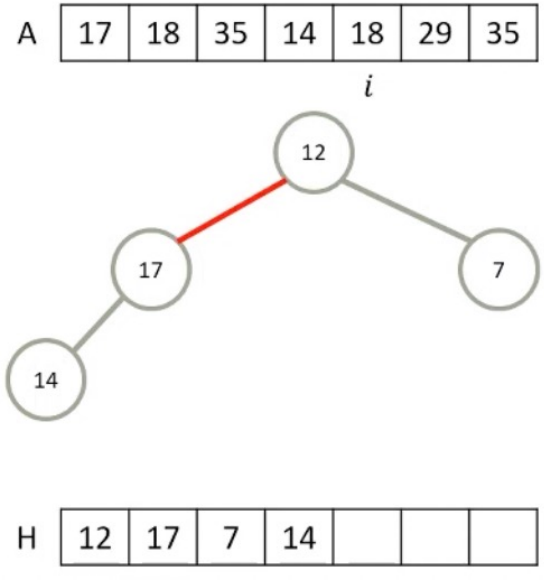
HeapSort($A[1 \ldots n]$)

```
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



Sort Using Priority Queues

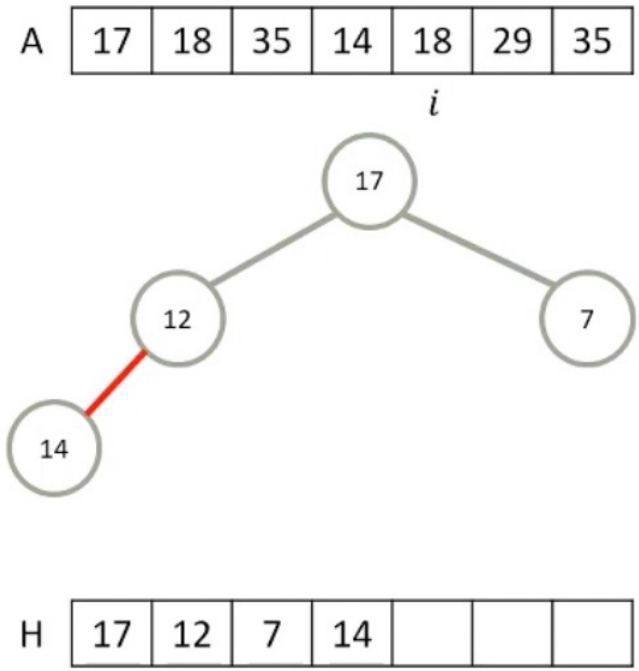
```
HeapSort(A[1...n])
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



Sort Using Priority Queues

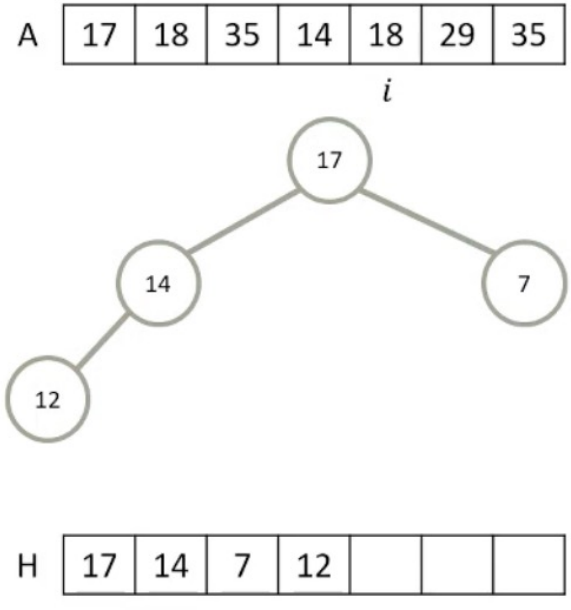
HeapSort($A[1 \ldots n]$)

```
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



Sort Using Priority Queues

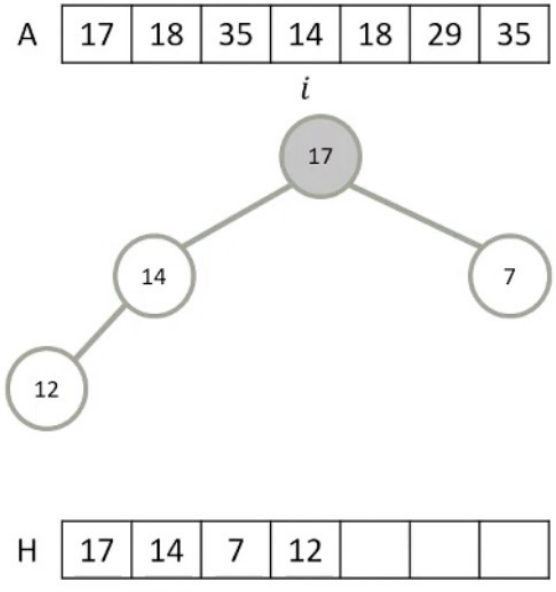
```
HeapSort(A[1...n])
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



Sort Using Priority Queues

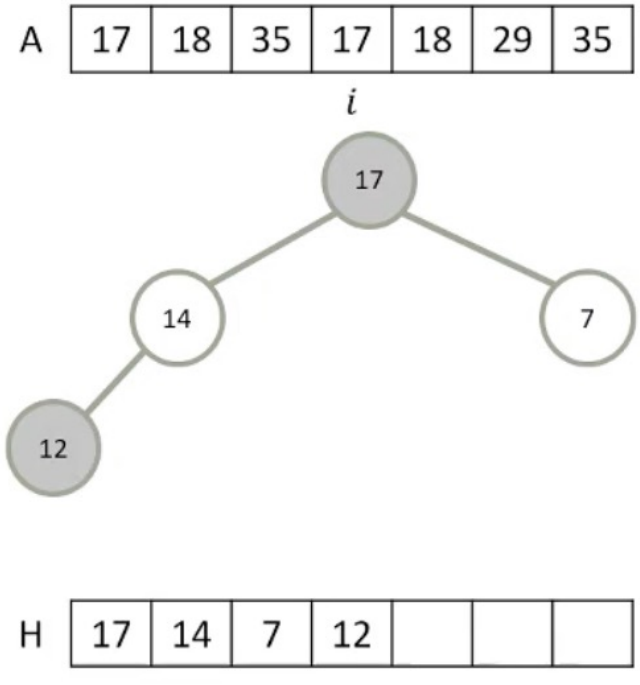
HeapSort($A[1 \ldots n]$)

```
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



Sort Using Priority Queues

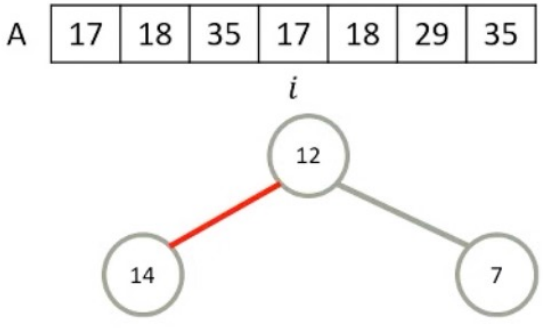
```
HeapSort(A[1...n])
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```

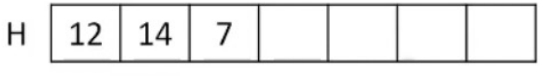


Sort Using Priority Queues

HeapSort($A[1 \ldots n]$)

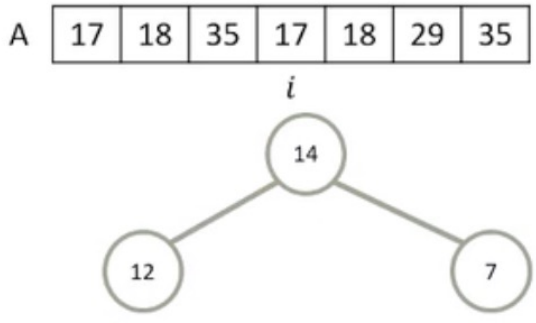
```
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```

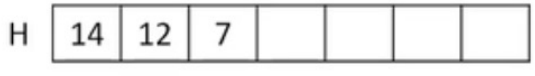




Sort Using Priority Queues

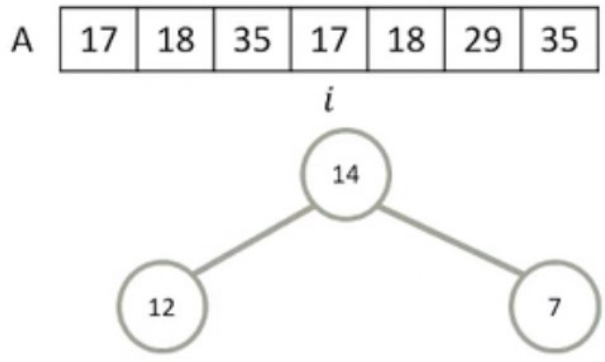
```
HeapSort(A[1...n])
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```

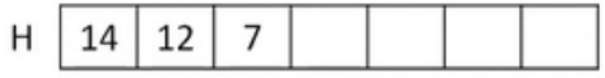




Sort Using Priority Queues

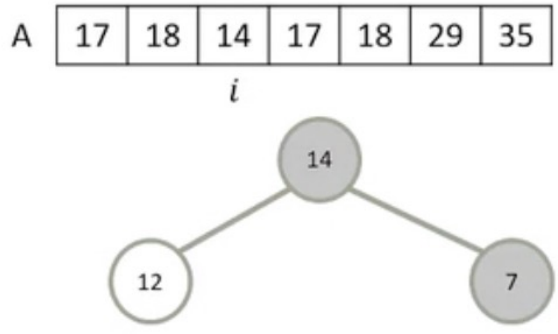
```
HeapSort(A[1...n])
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```

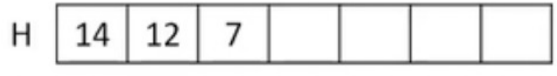




Sort Using Priority Queues

```
HeapSort(A[1...n])
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```

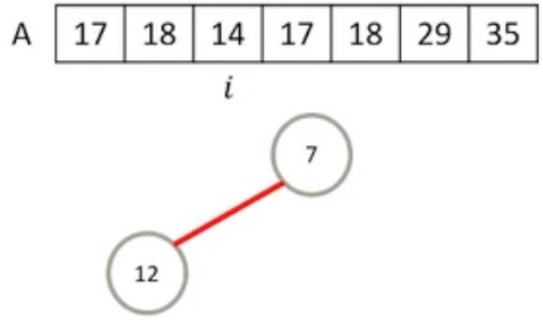


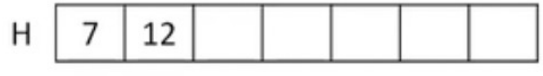


Sort Using Priority Queues

HeapSort($A[1 \ldots n]$)

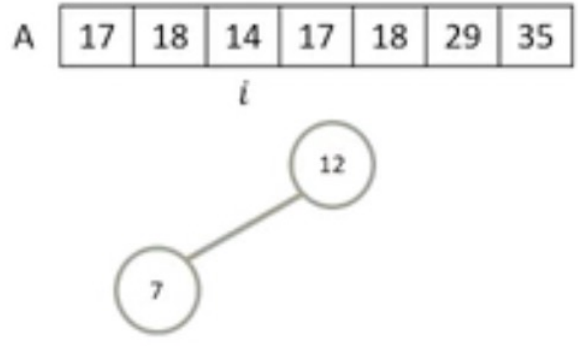
```
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```

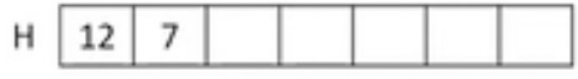




Sort Using Priority Queues

```
HeapSort(A[1...n])
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```

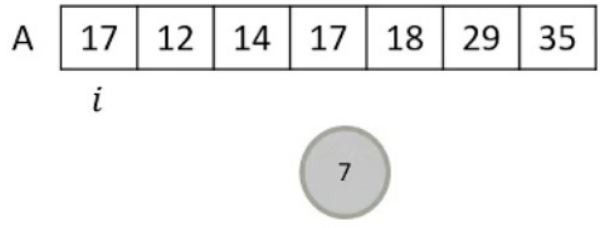




Sort Using Priority Queues

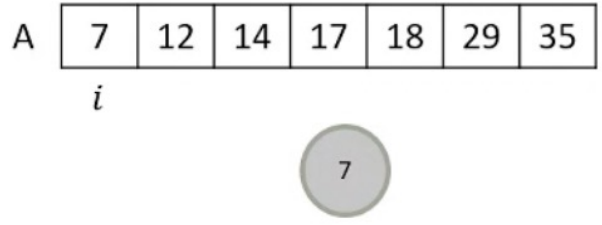
HeapSort($A[1 \ldots n]$)

```
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



Sort Using Priority Queues

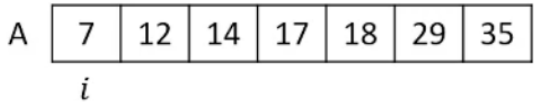
```
HeapSort(A[1...n])
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



Sort Using Priority Queues

HeapSort($A[1 \ldots n]$)

```
create an empty priority
queue for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] ← ExtractMax()
```



- The resulting algorithms is comparison-based and has running time $O(n \log n)$ (hence, asymptotically optimal!).

This lesson

In-place heap sort algorithm. For this, we will first turn a given array into a heap by permuting its elements.

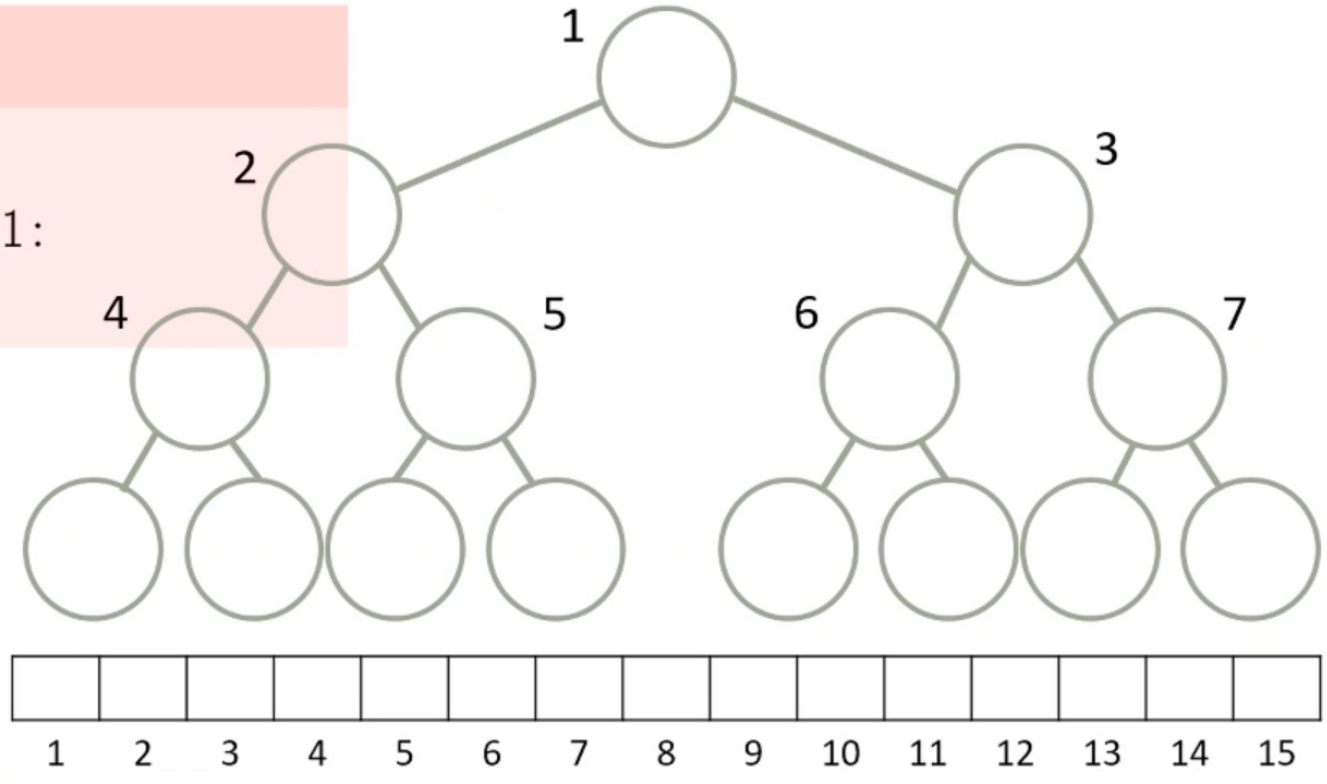
Turn Array into a Heap

```
BuildHeap(A[1...n])

size ← n
for i from ⌊n/2⌋ downto 1:
    SiftDown(i)
```

```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:
size = 15

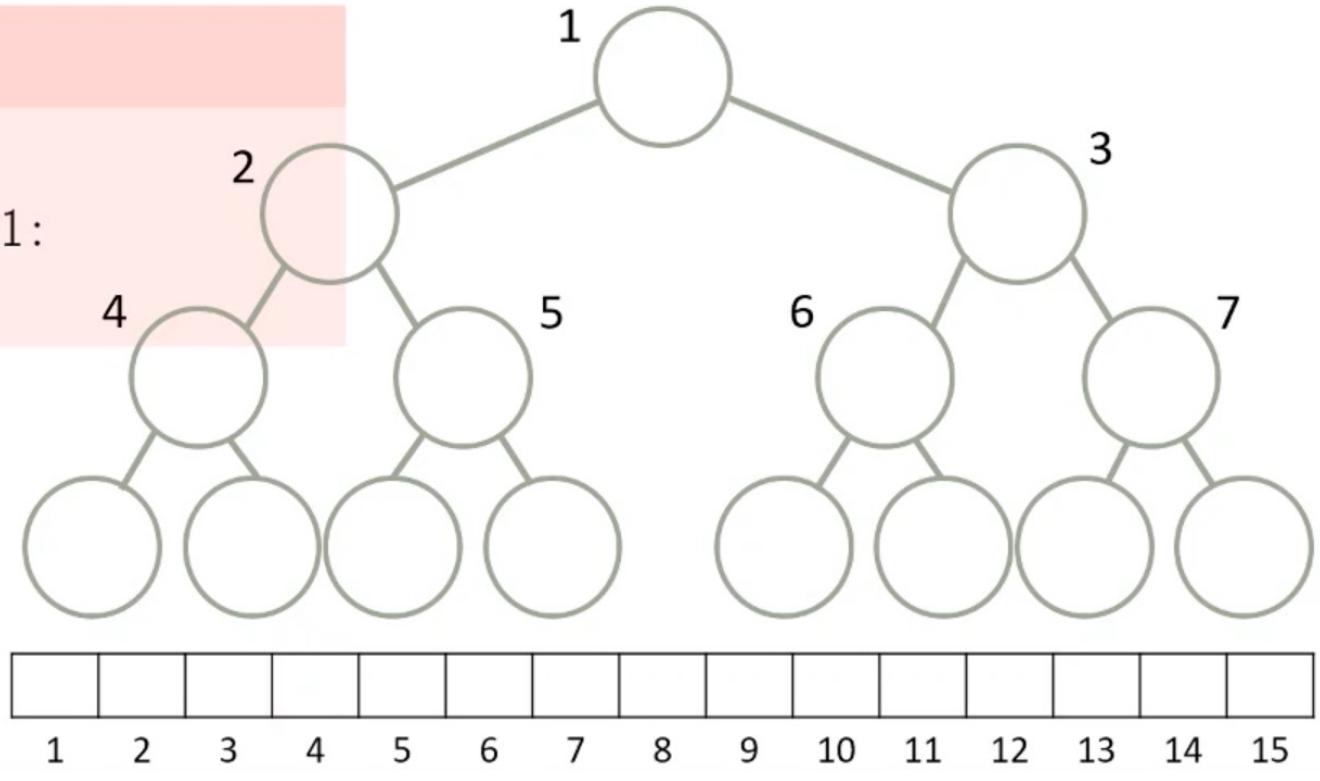


```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

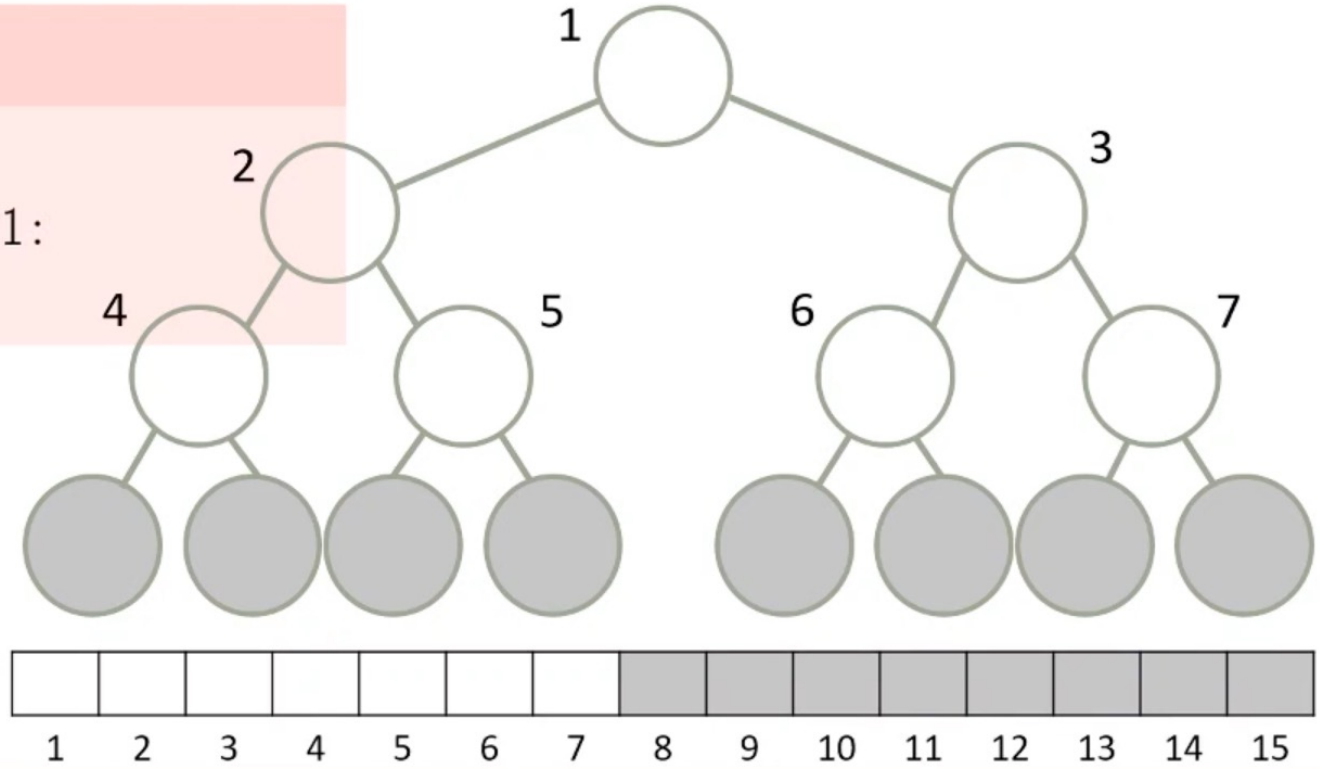


```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

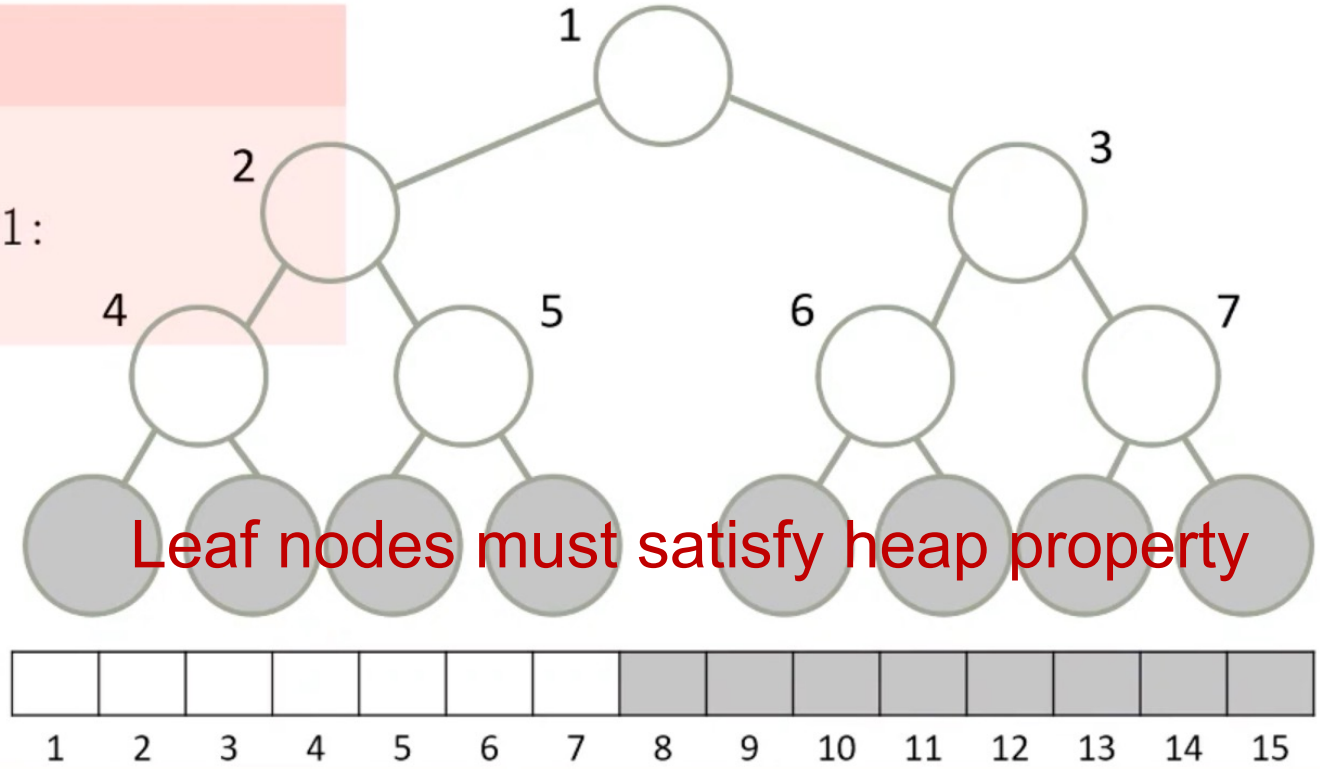


```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15



BuildHeap($A[1 \ldots n]$)

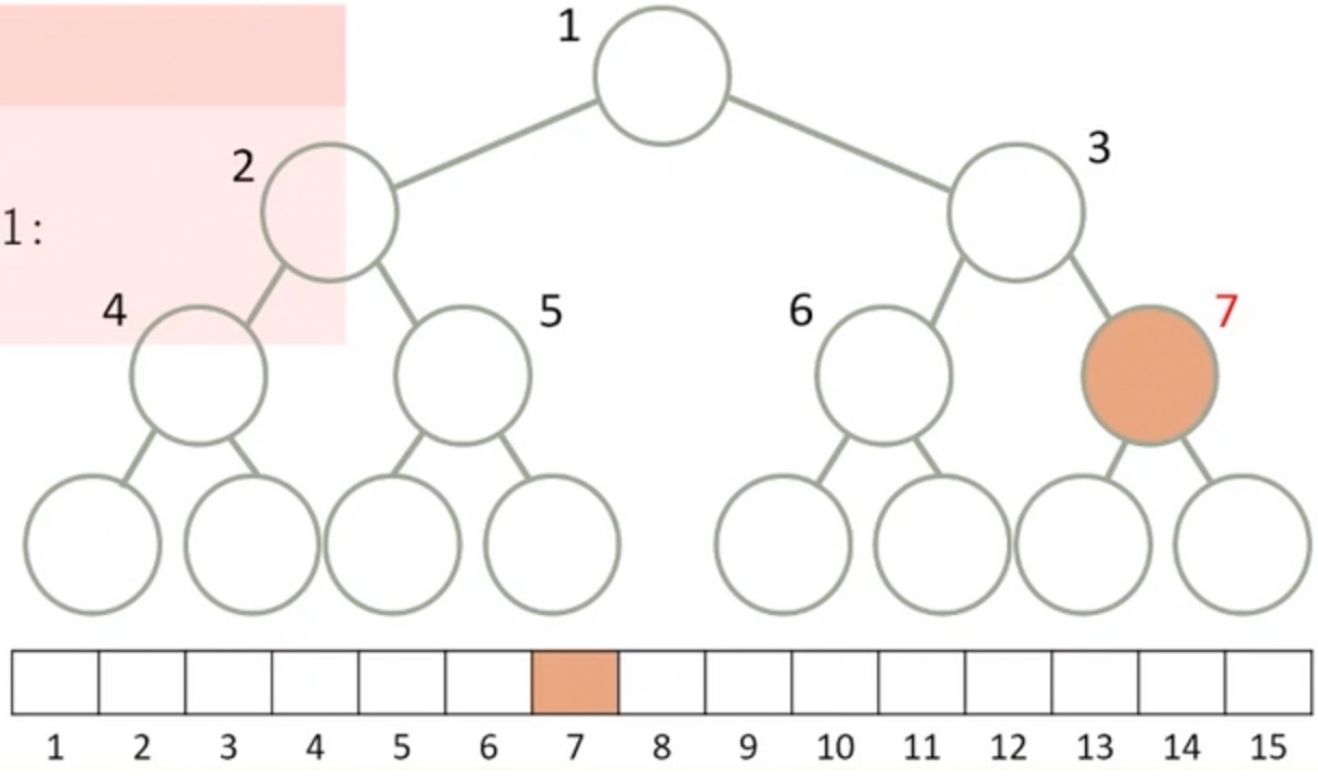
```
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

First potential node violation



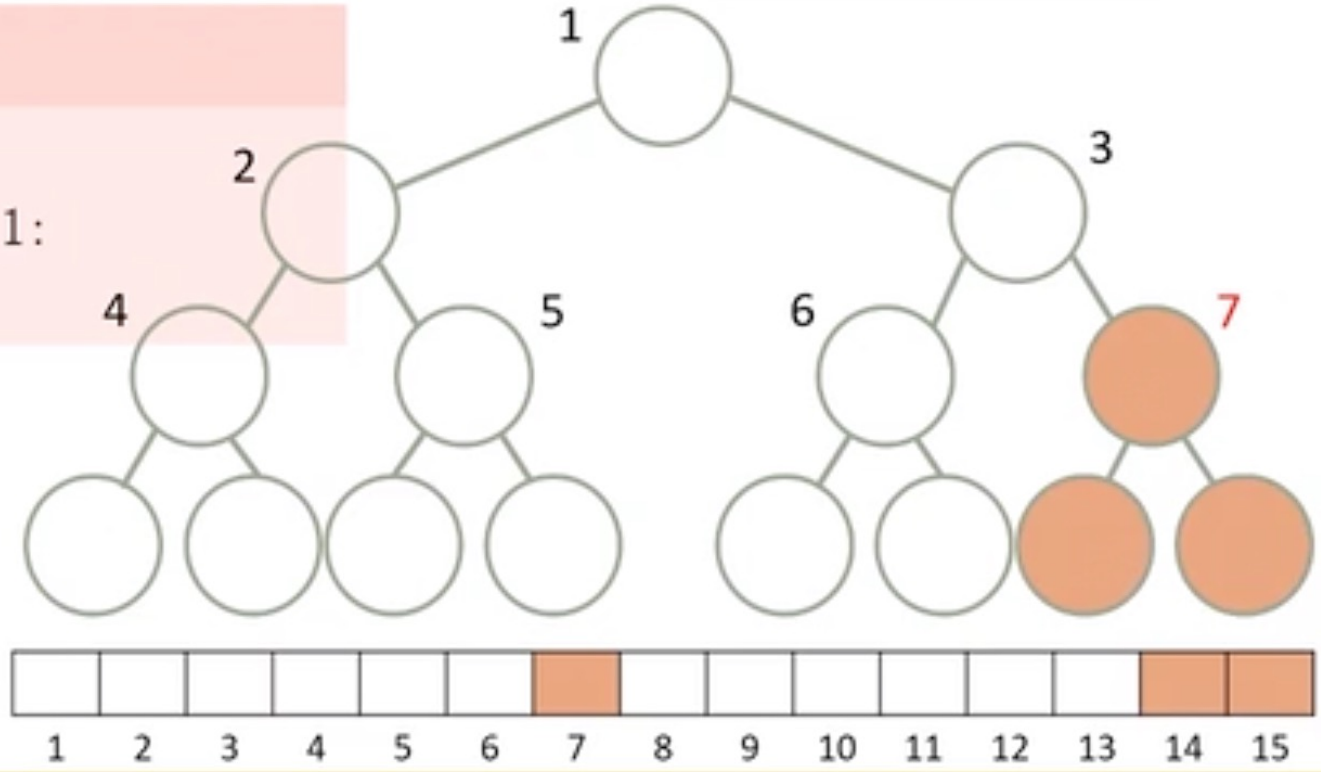
```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

First potential tree violation



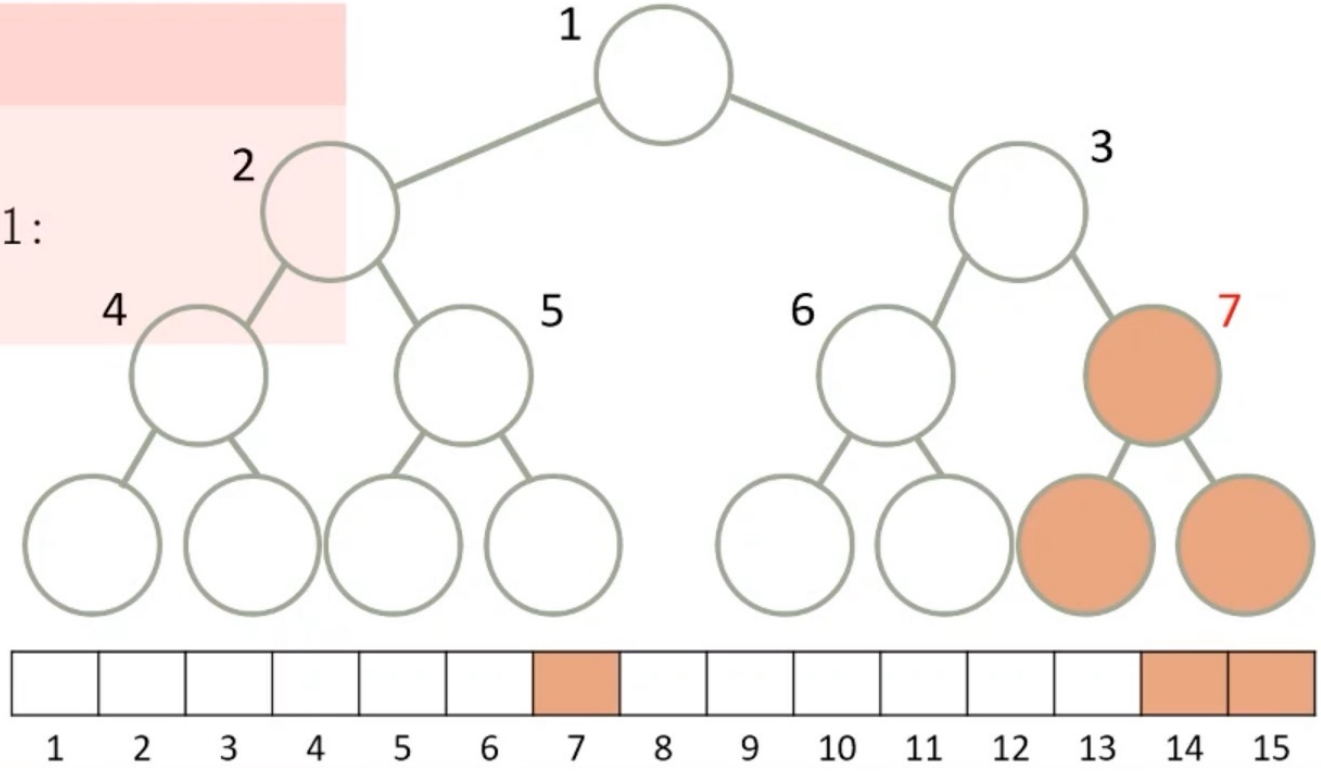
```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

Call $SiftDown$ for node 7



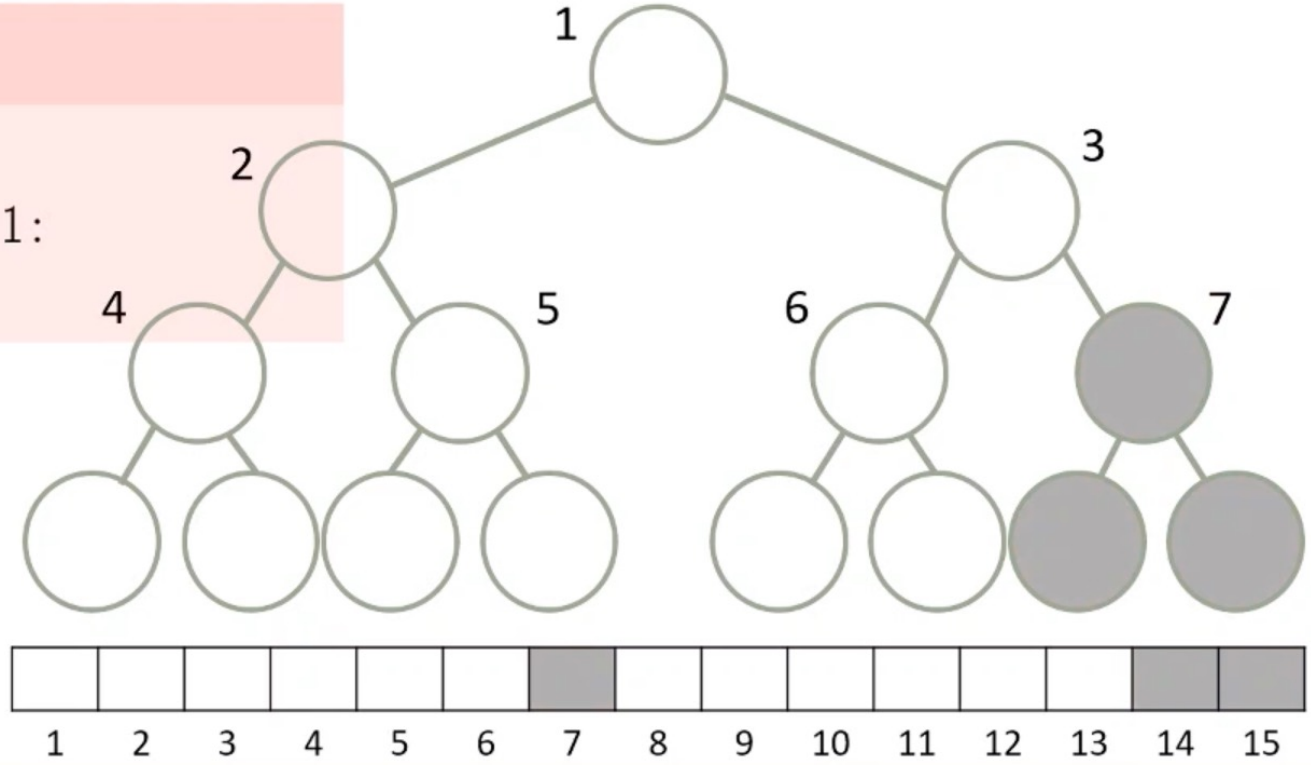
```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

Heap Property satisfied



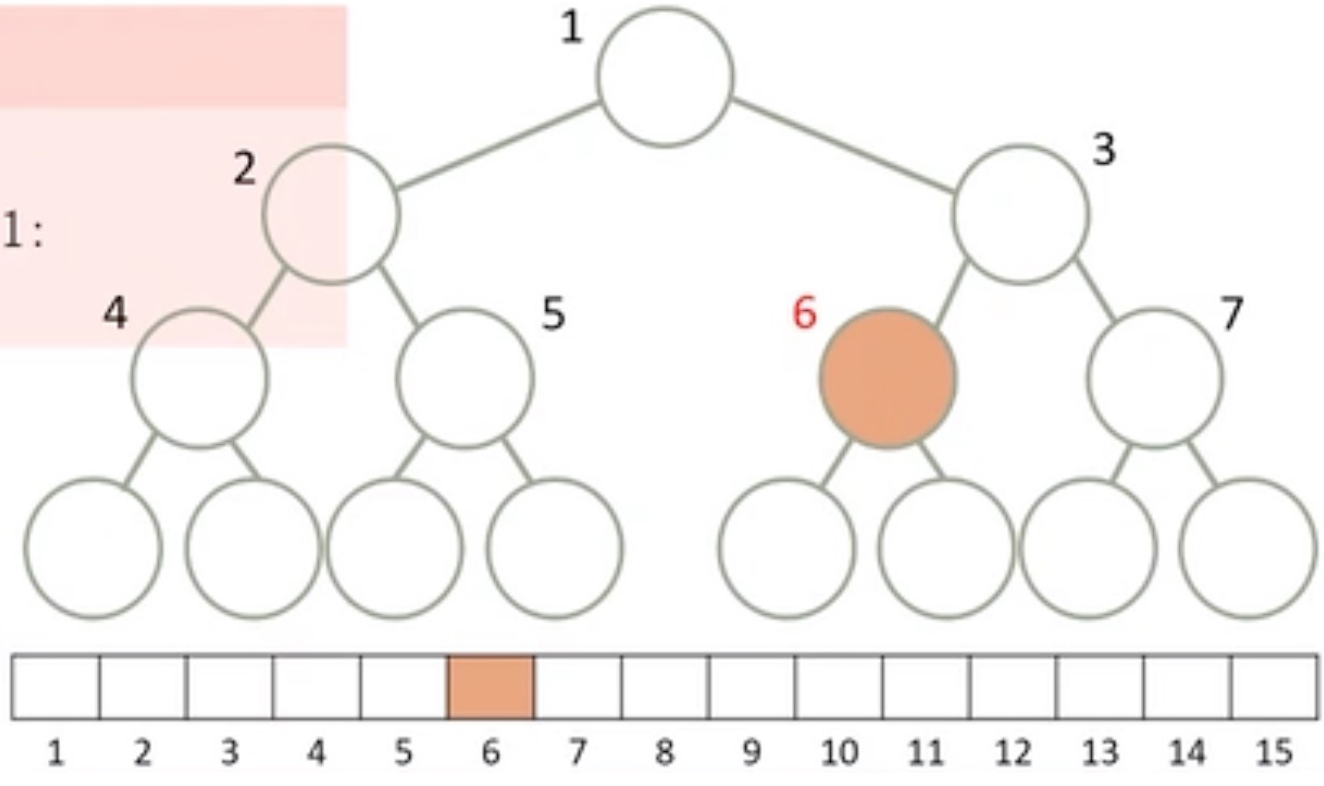
```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

Next potential node violation



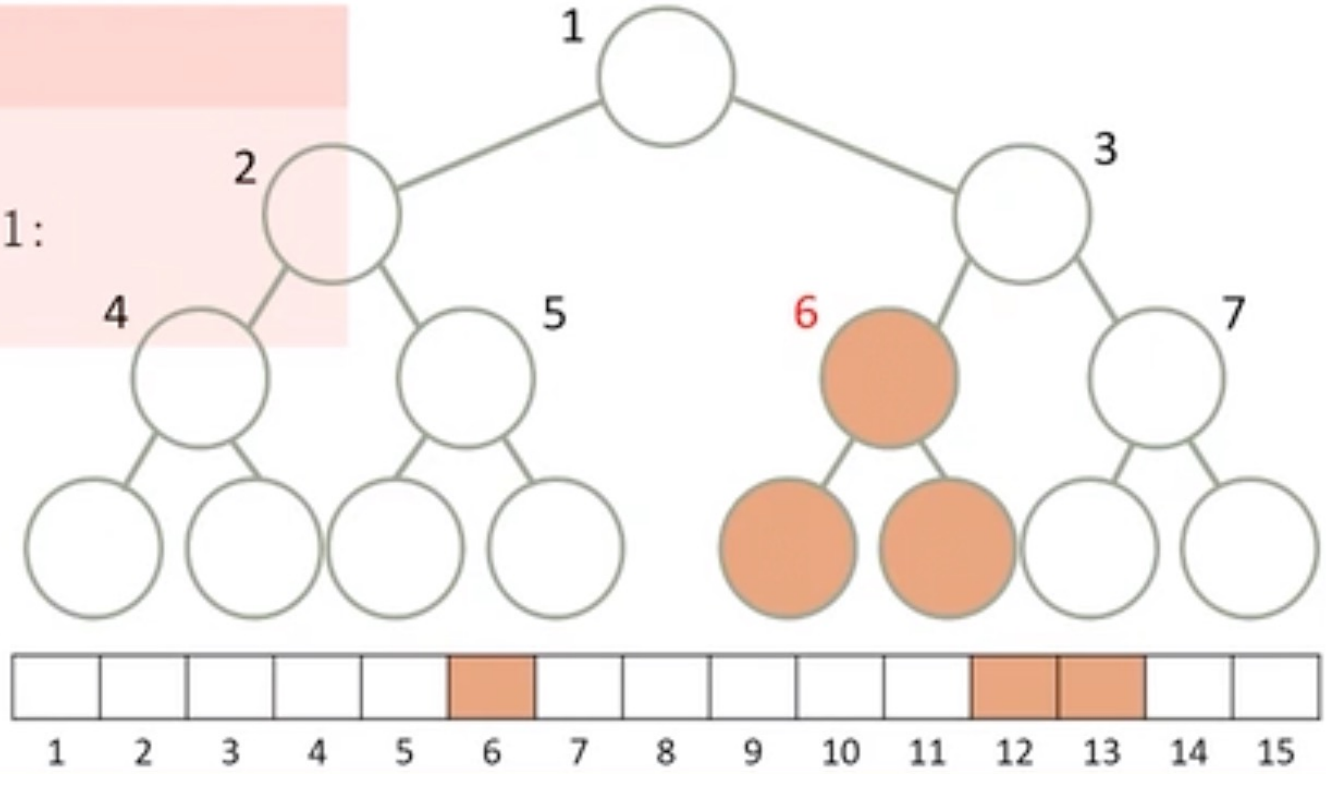
```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

Next potential tree violation



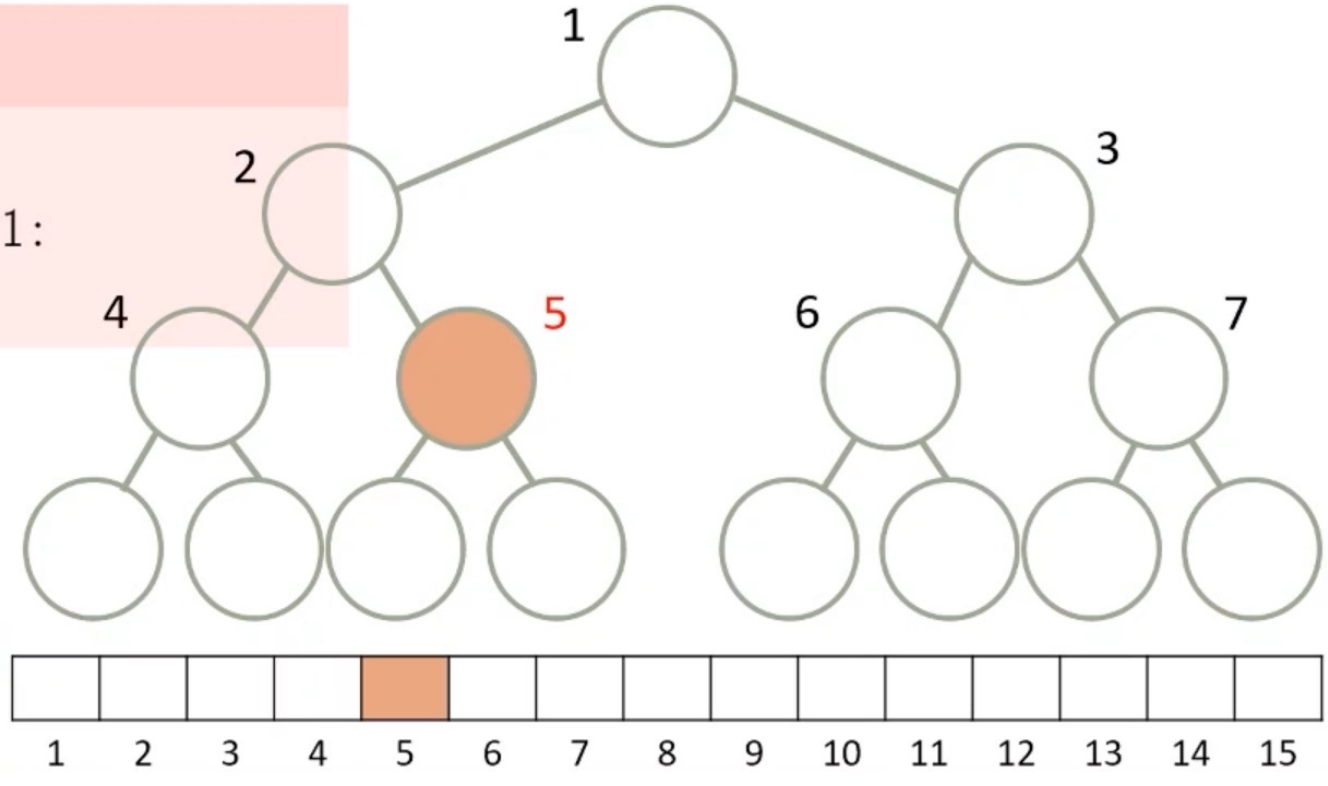
```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

Next potential node violation



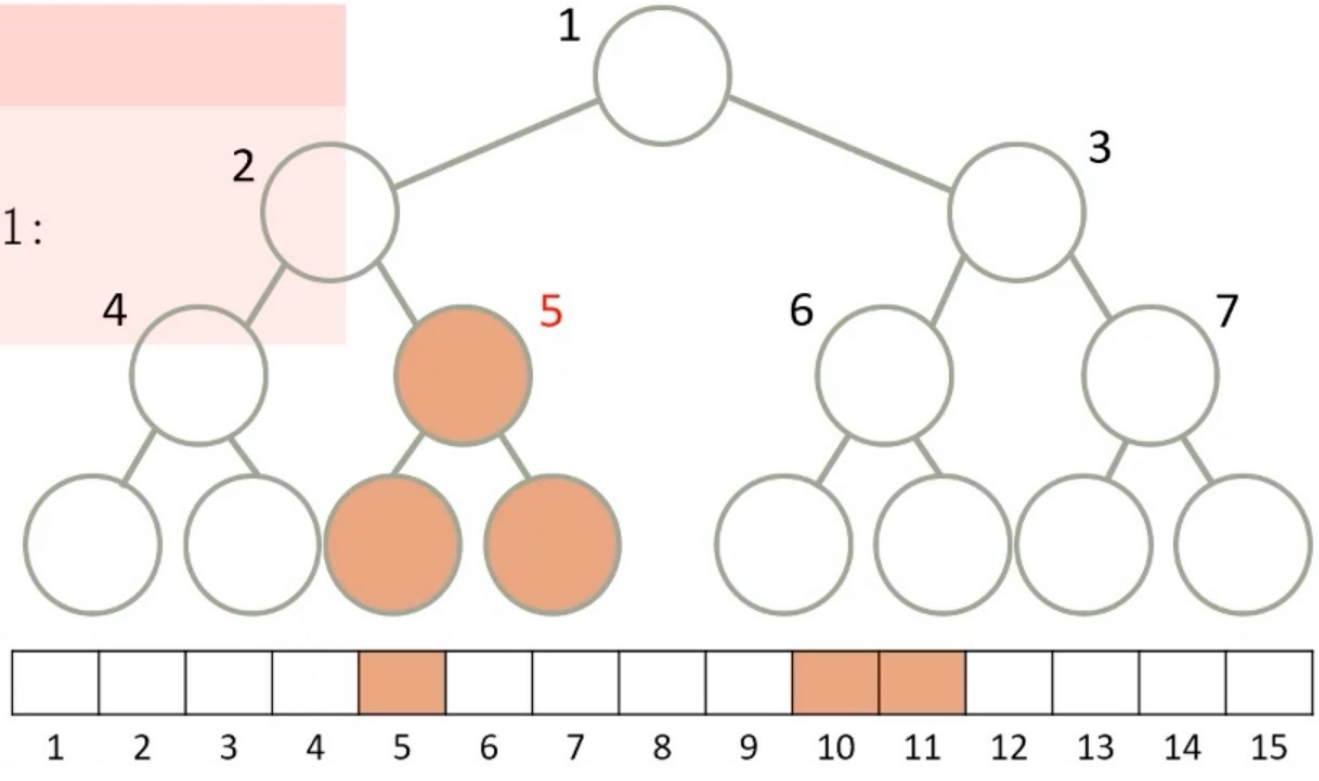
```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

Next potential tree violation



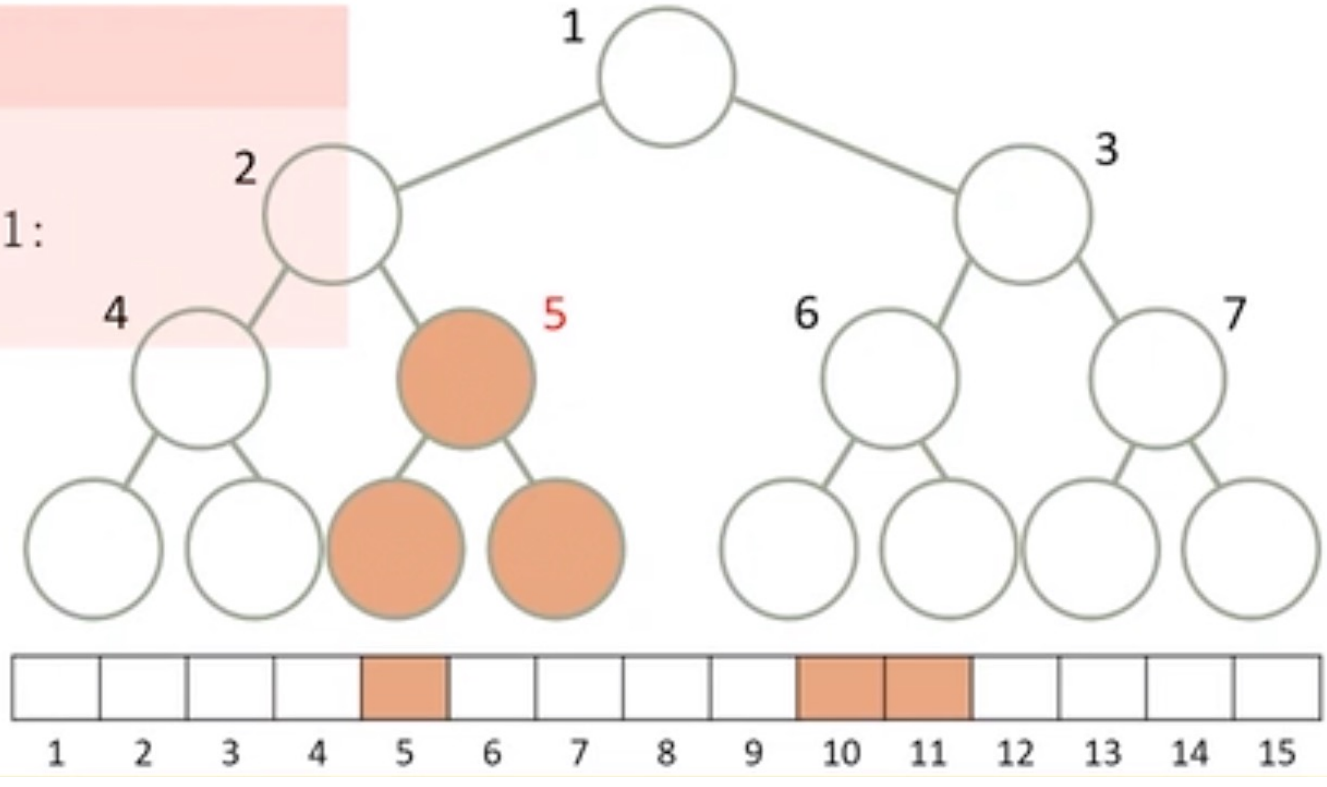
```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

Call $SiftDown$ for node 5



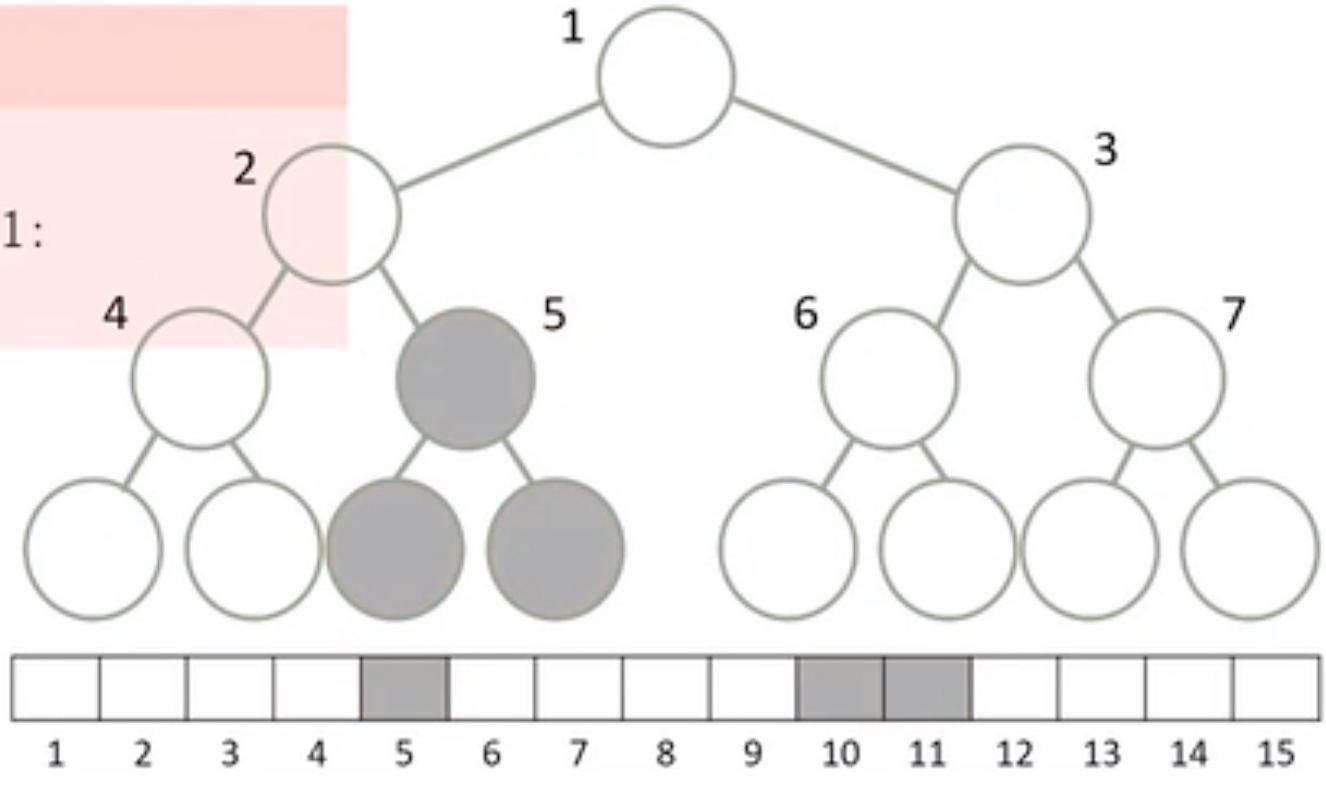
```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

Heap Property satisfied



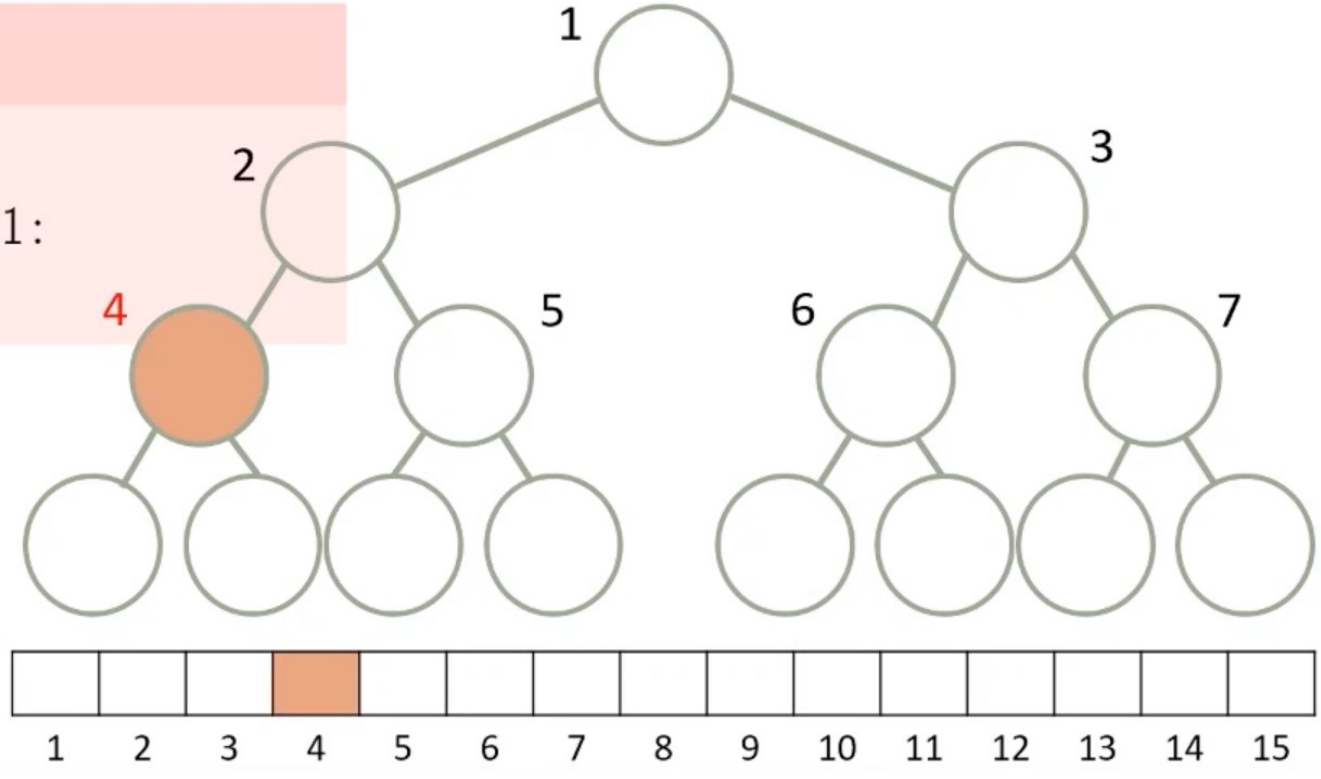
```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

Next potential node violation



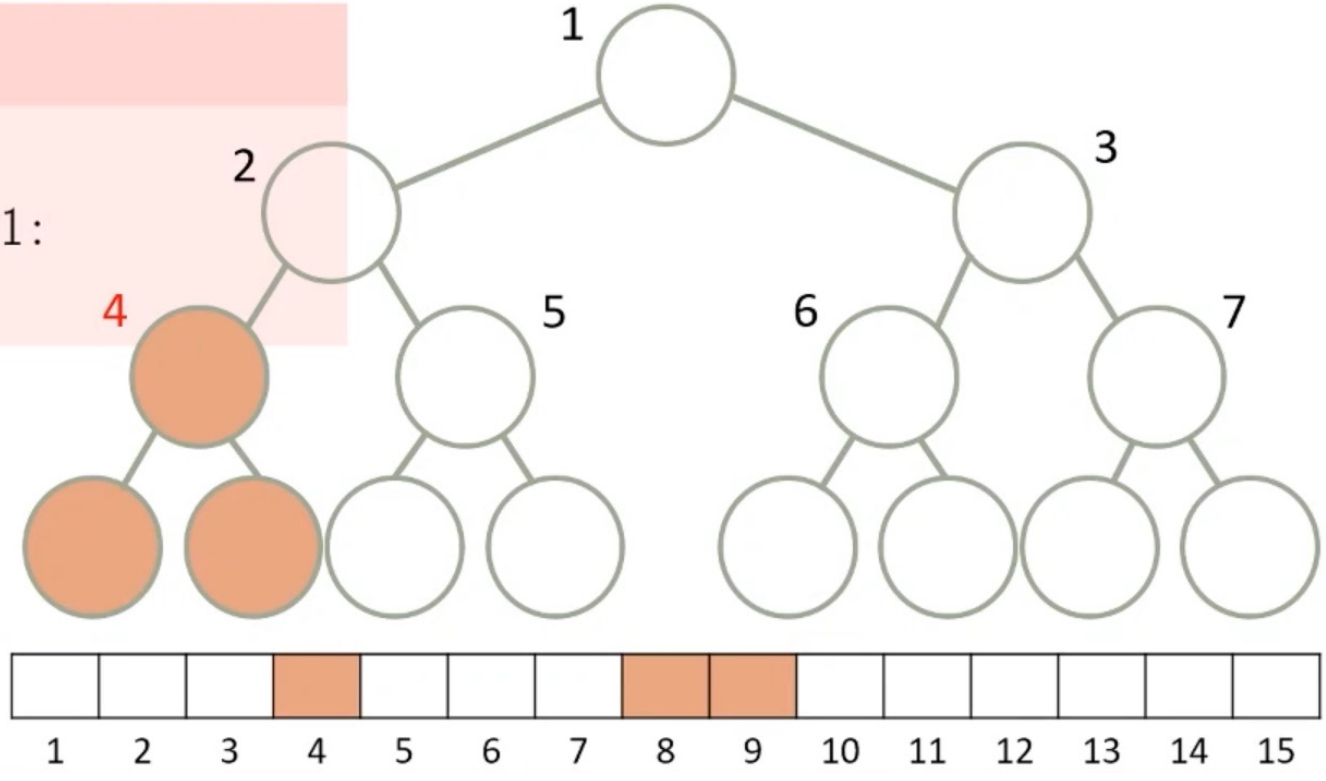
```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

Next potential tree violation



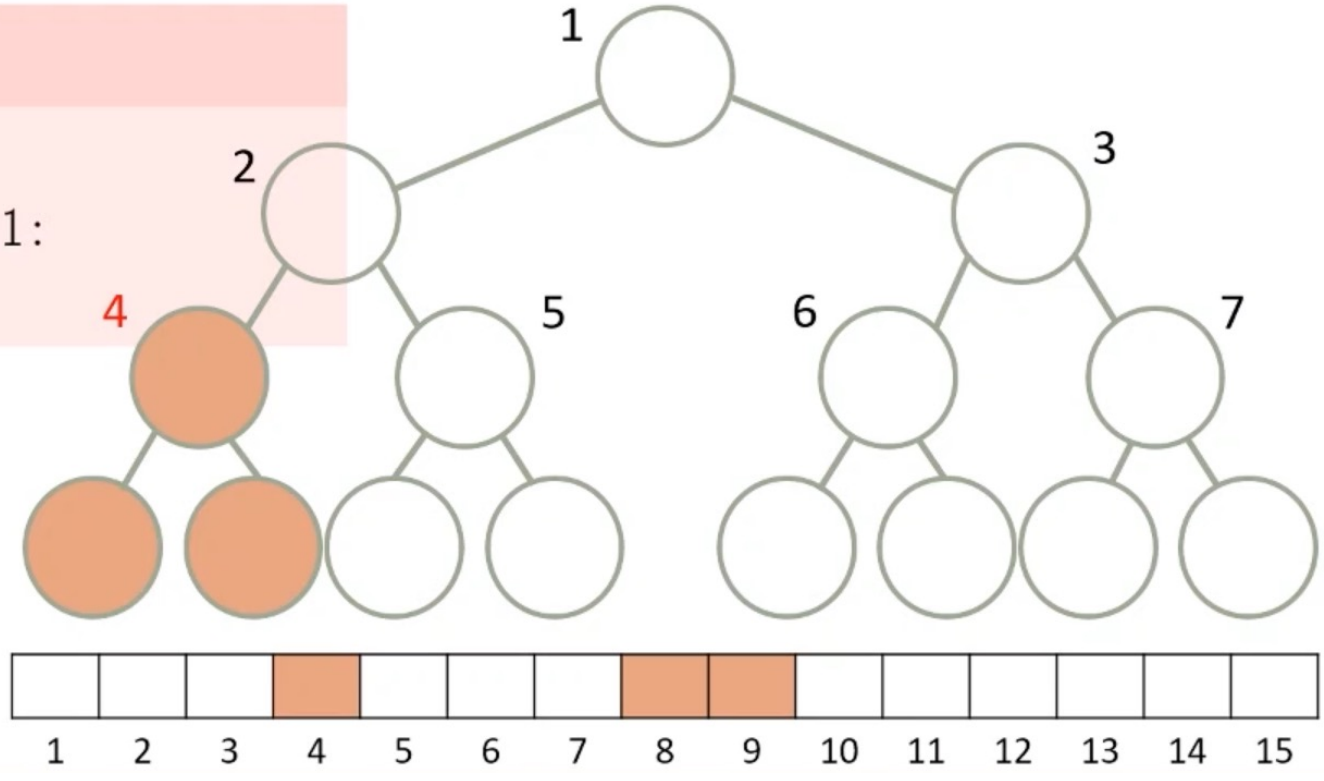
```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

Call $SiftDown$ for node 4



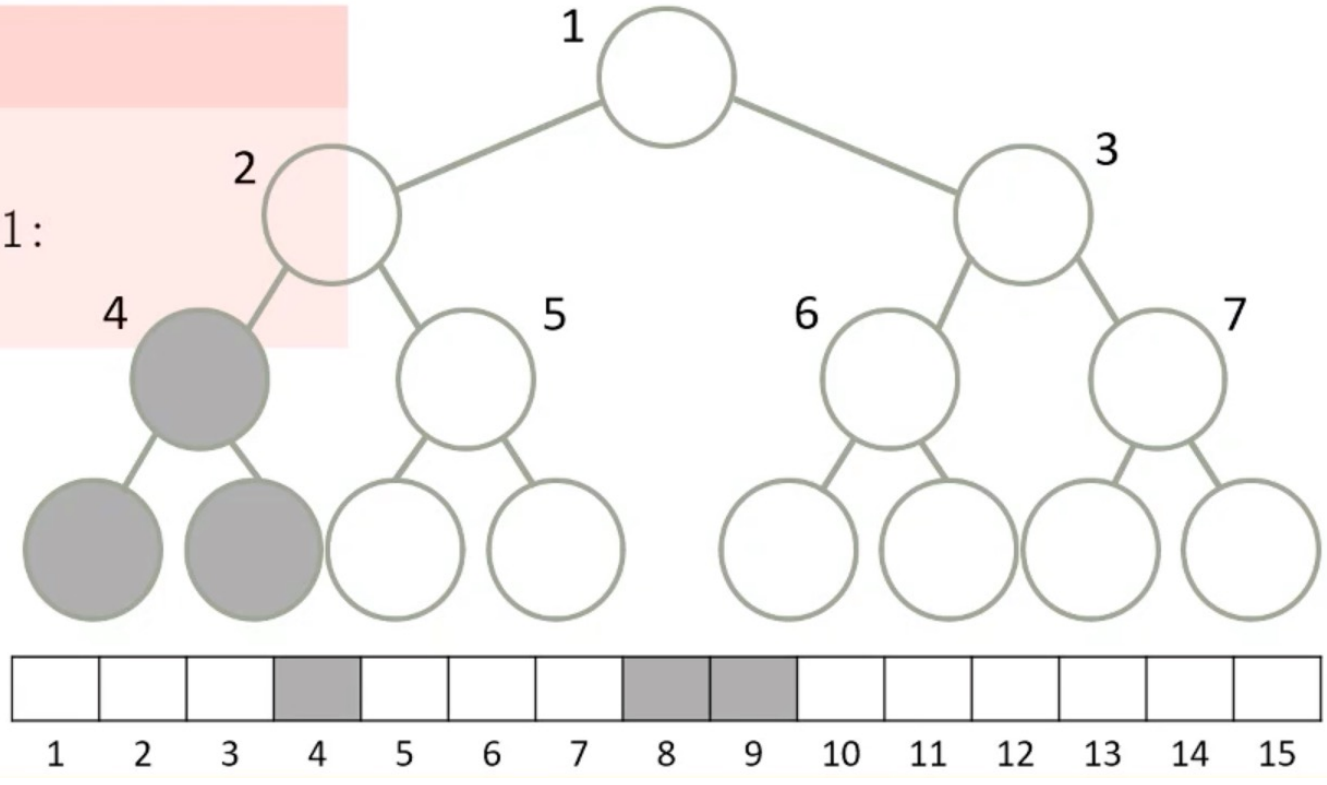
```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

Heap Property satisfied

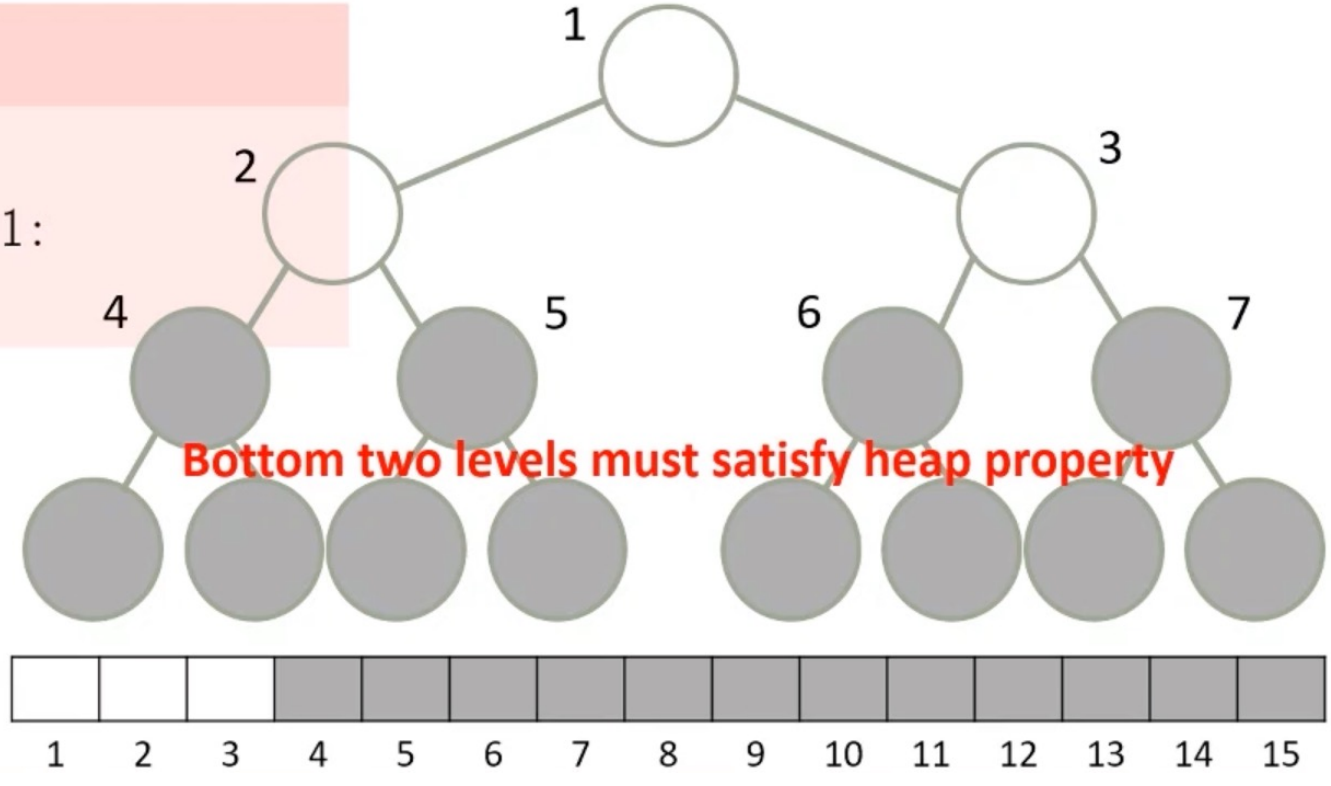


```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15



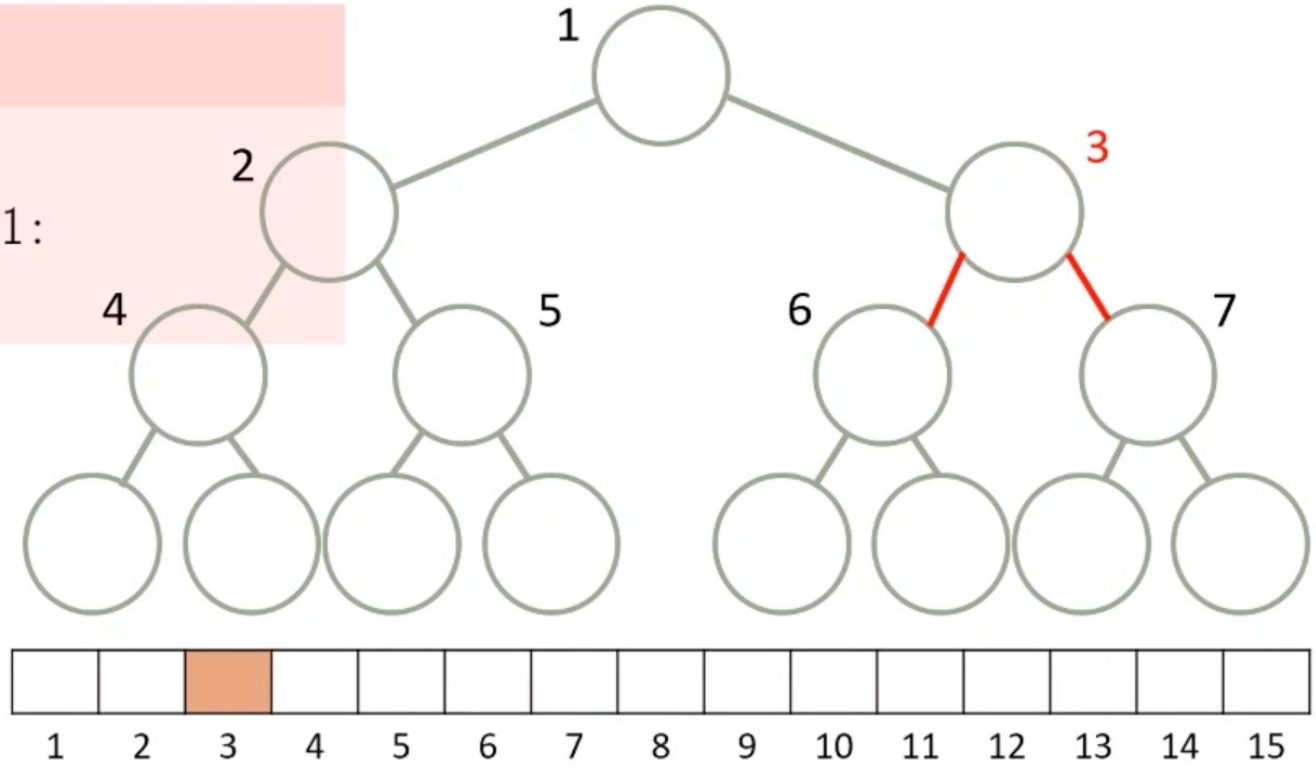
```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

Next potential node violation



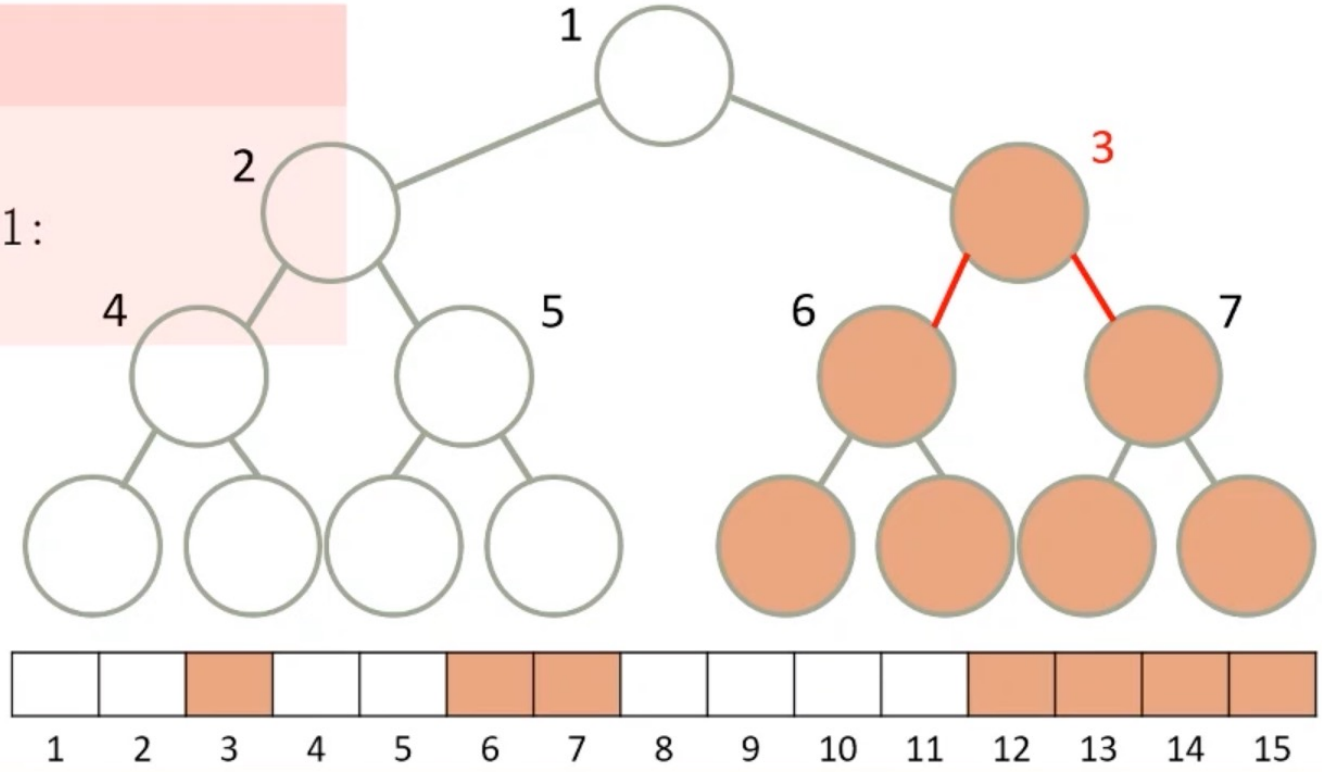
```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

Call $SiftDown$ for node 3



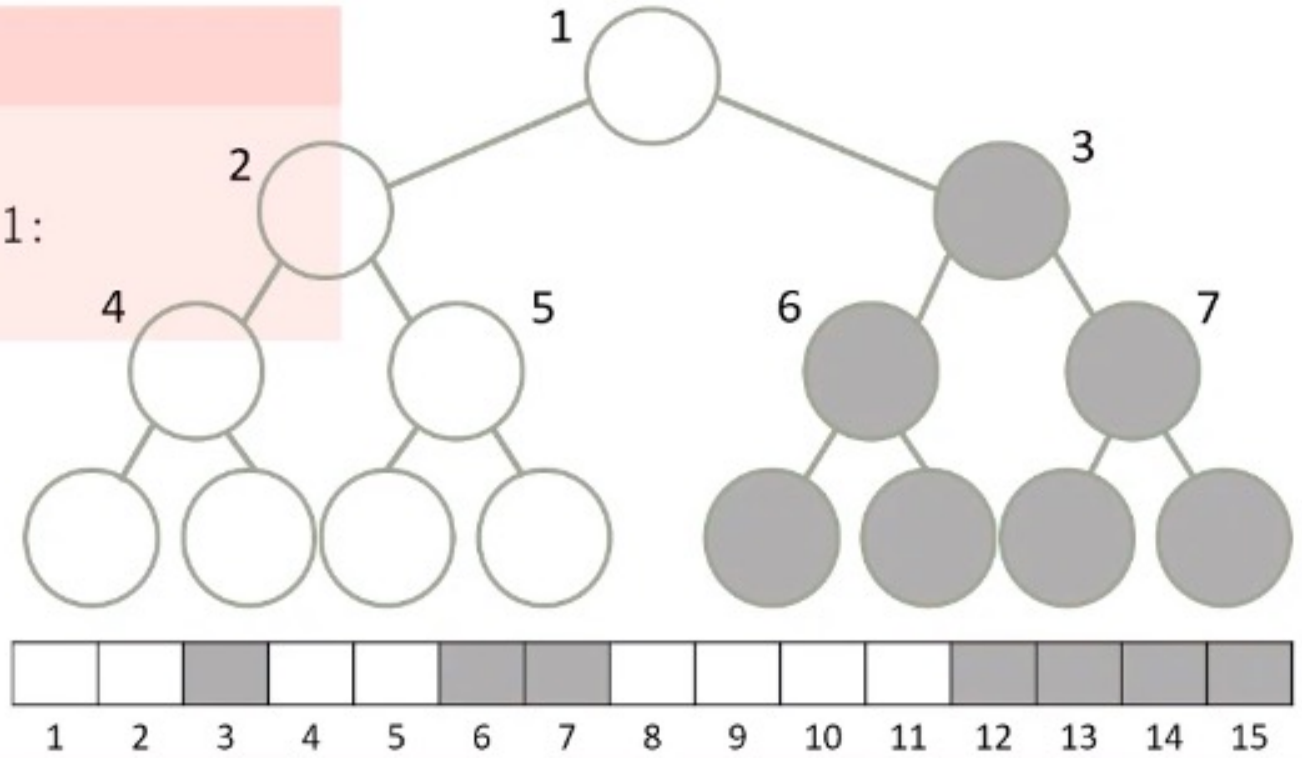
```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

Heap Property satisfied



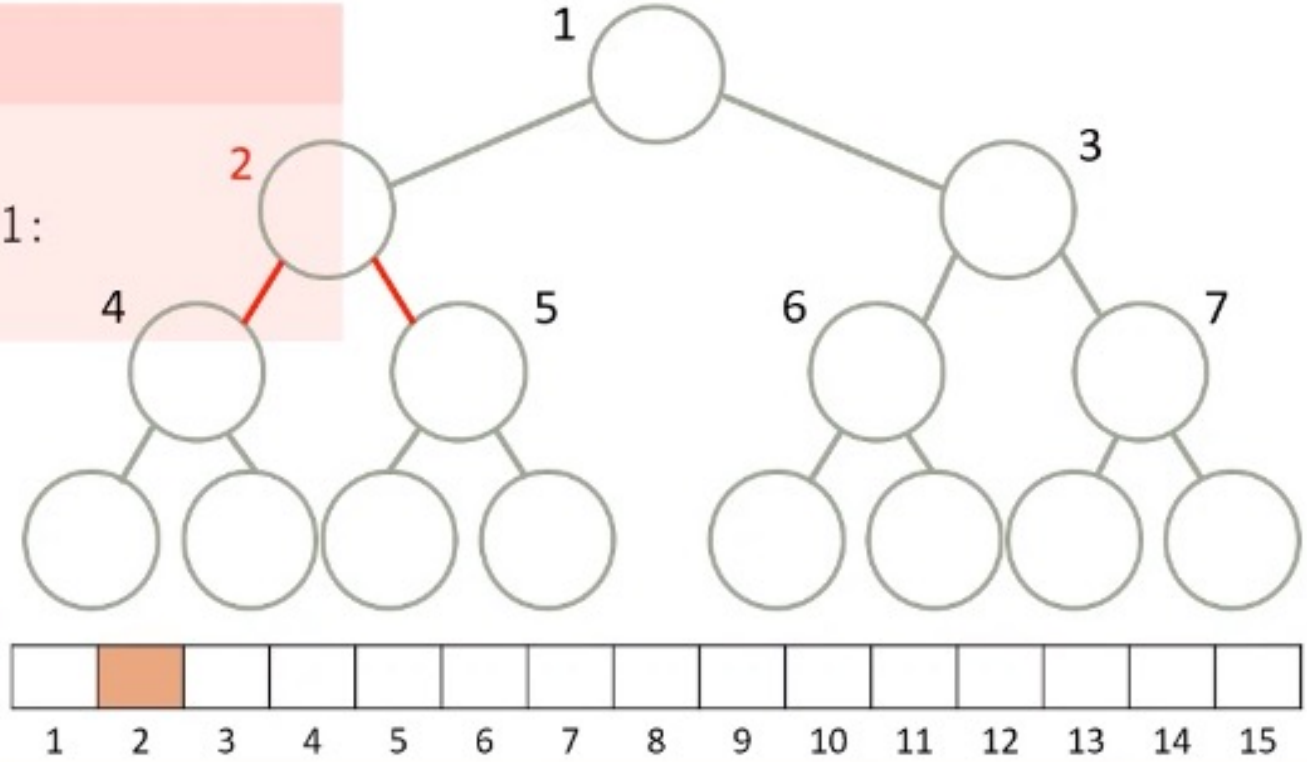
```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

Next potential node violation



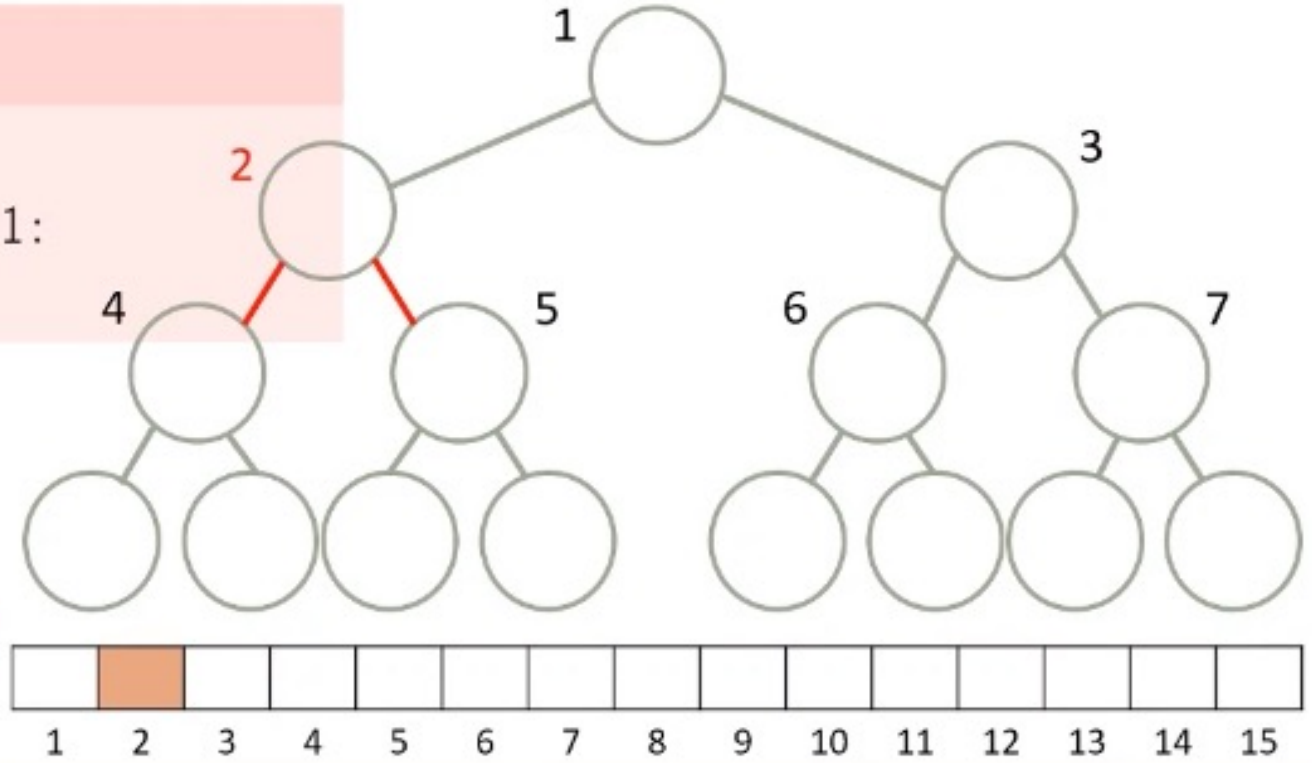
```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

Next potential node violation



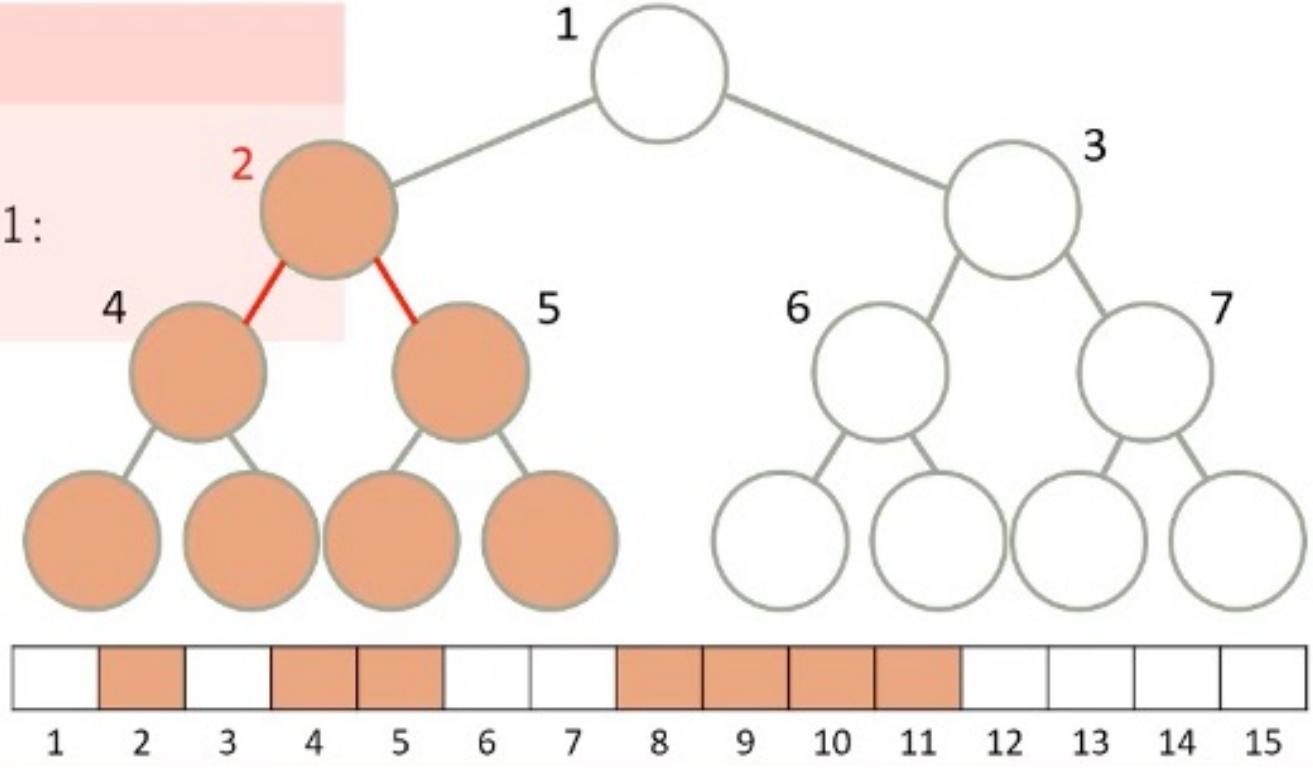
```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

Call $SiftDown$ for node 2



BuildHeap($A[1 \ldots n]$)

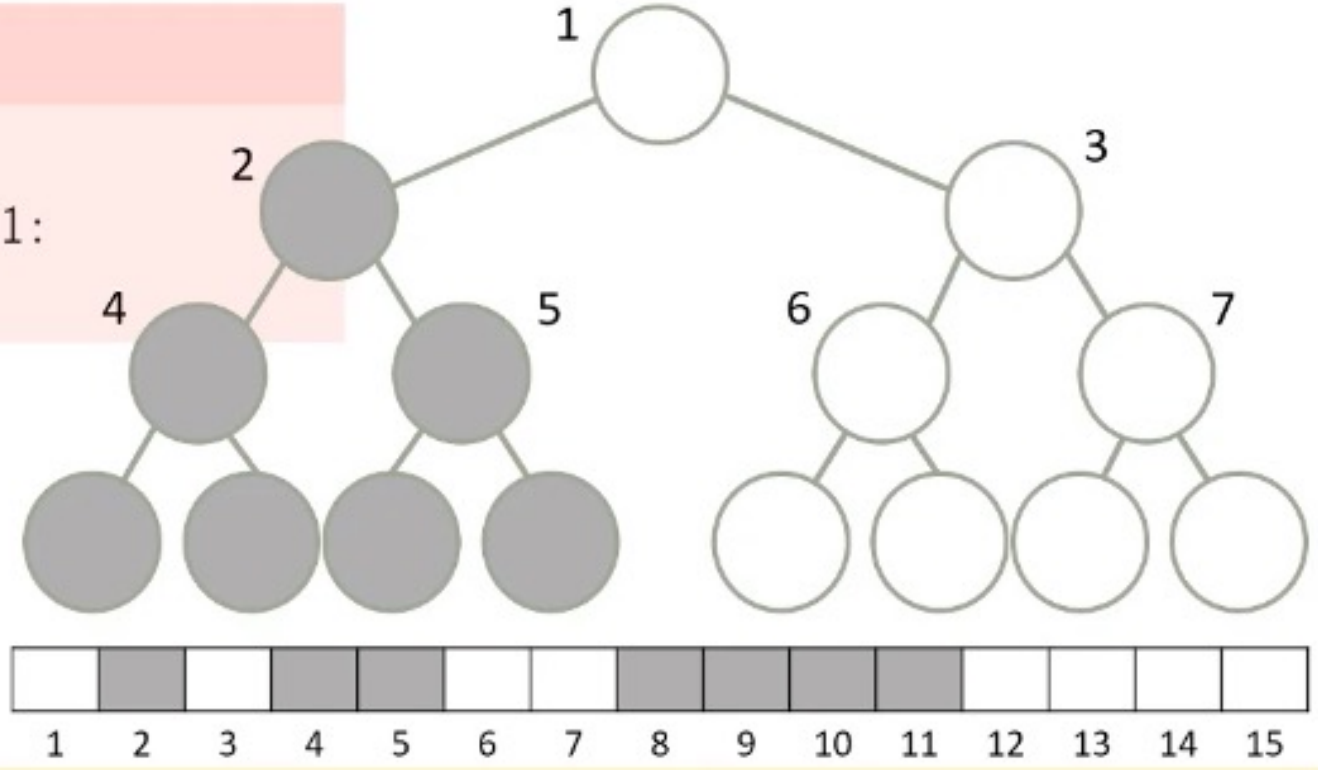
```
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

Heap Property satisfied

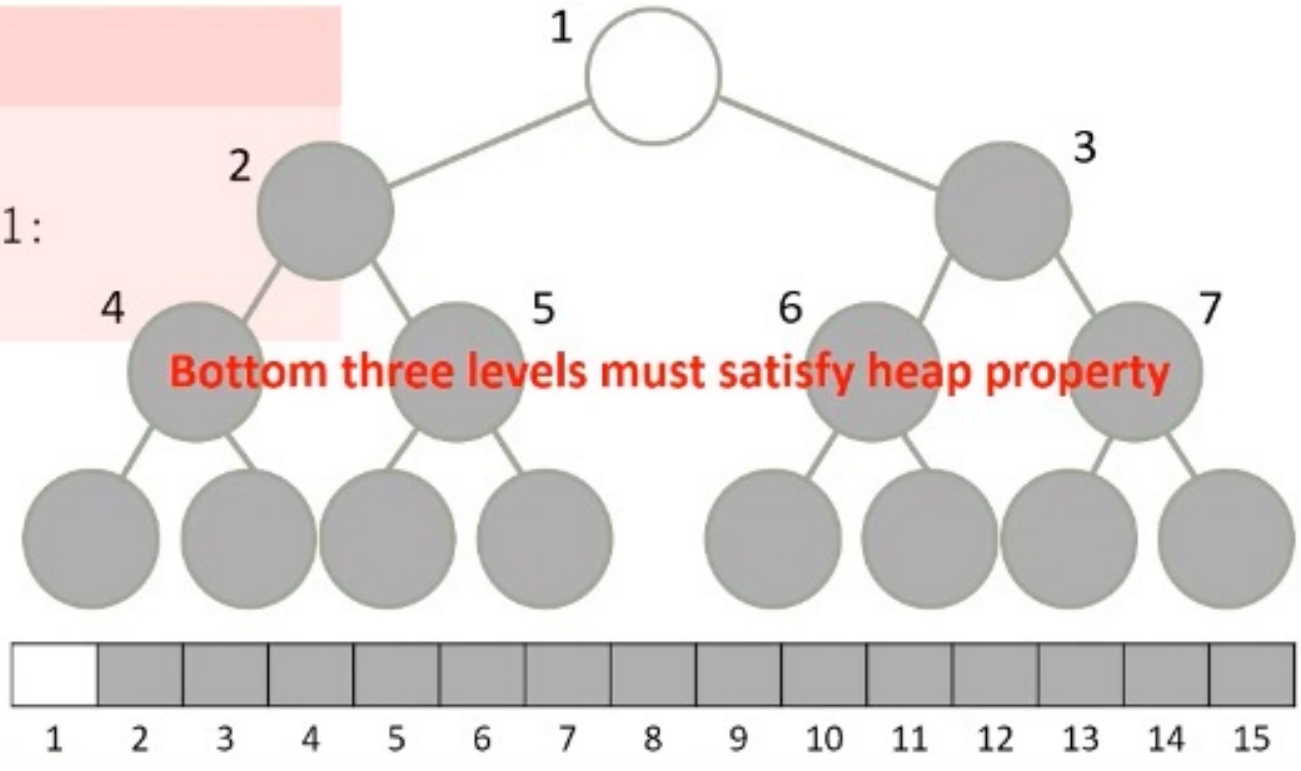


```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15



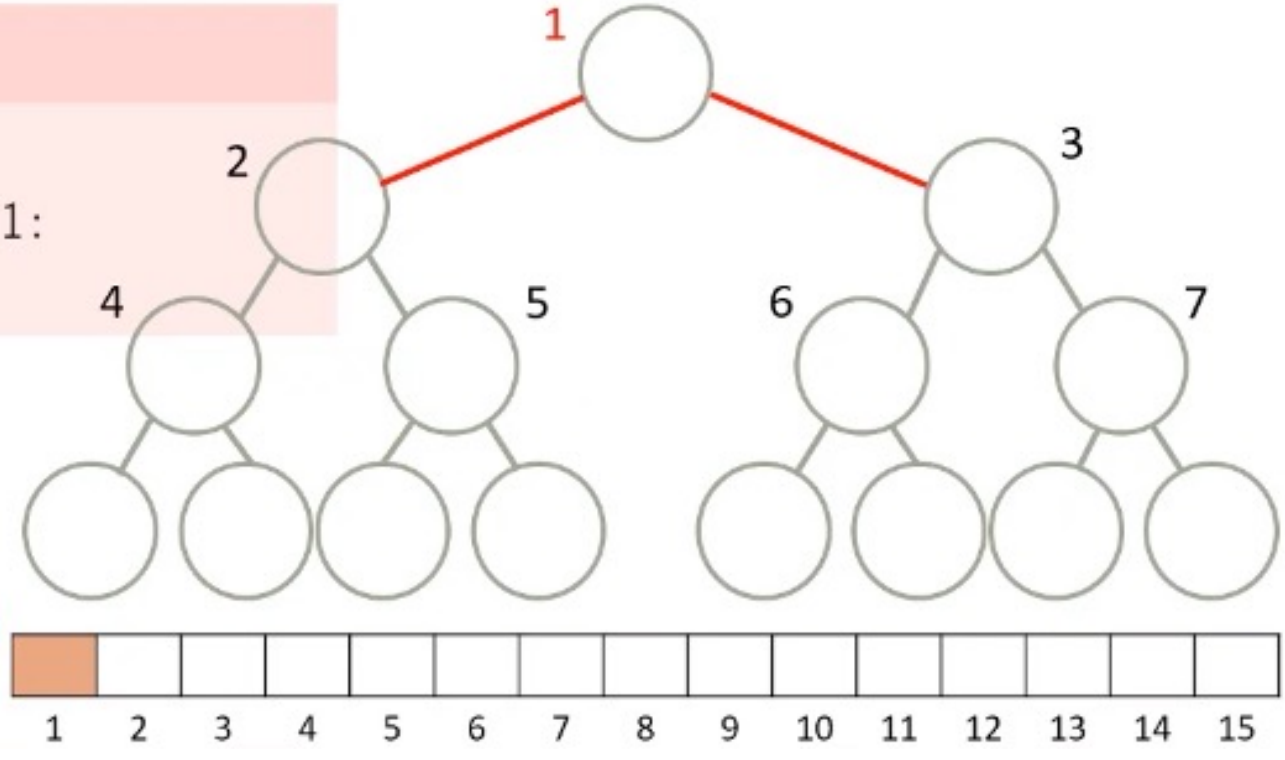
```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

Only potential node violation



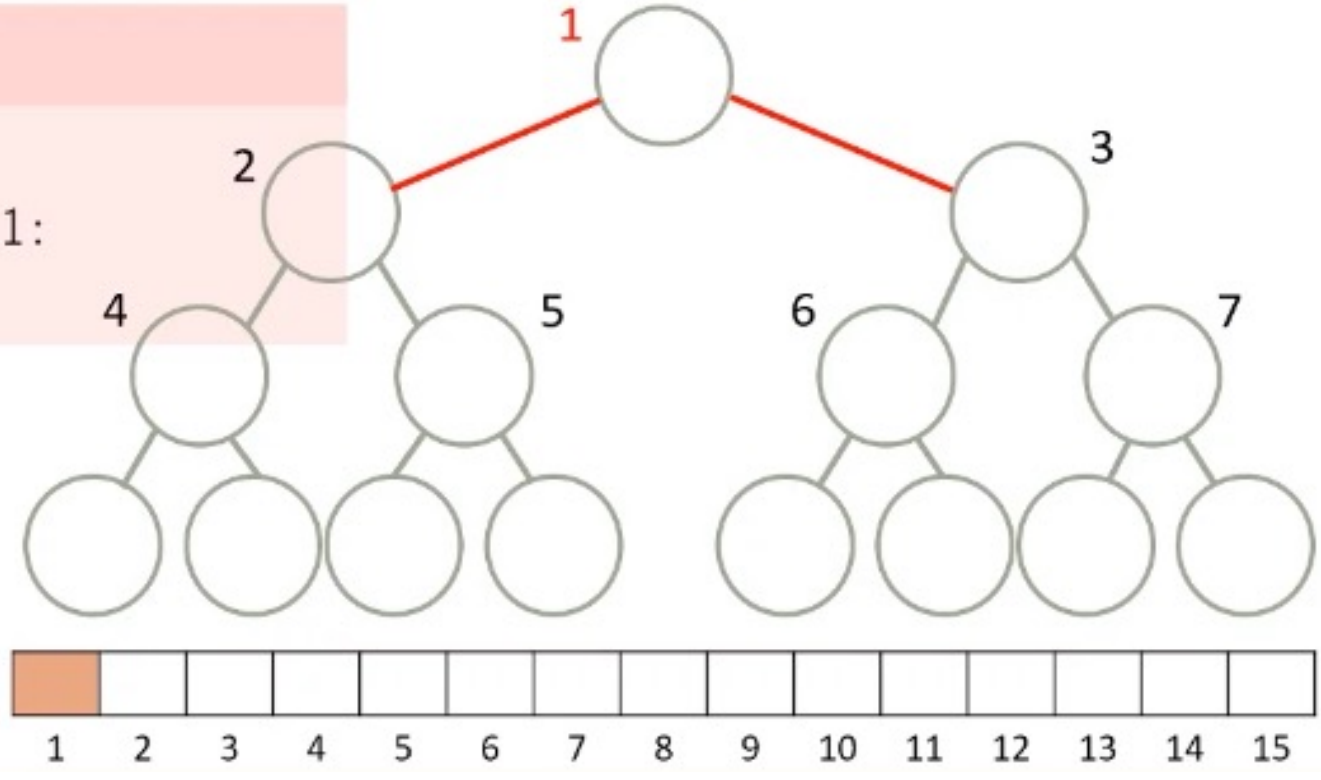
```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15

Call $SiftDown$ for root

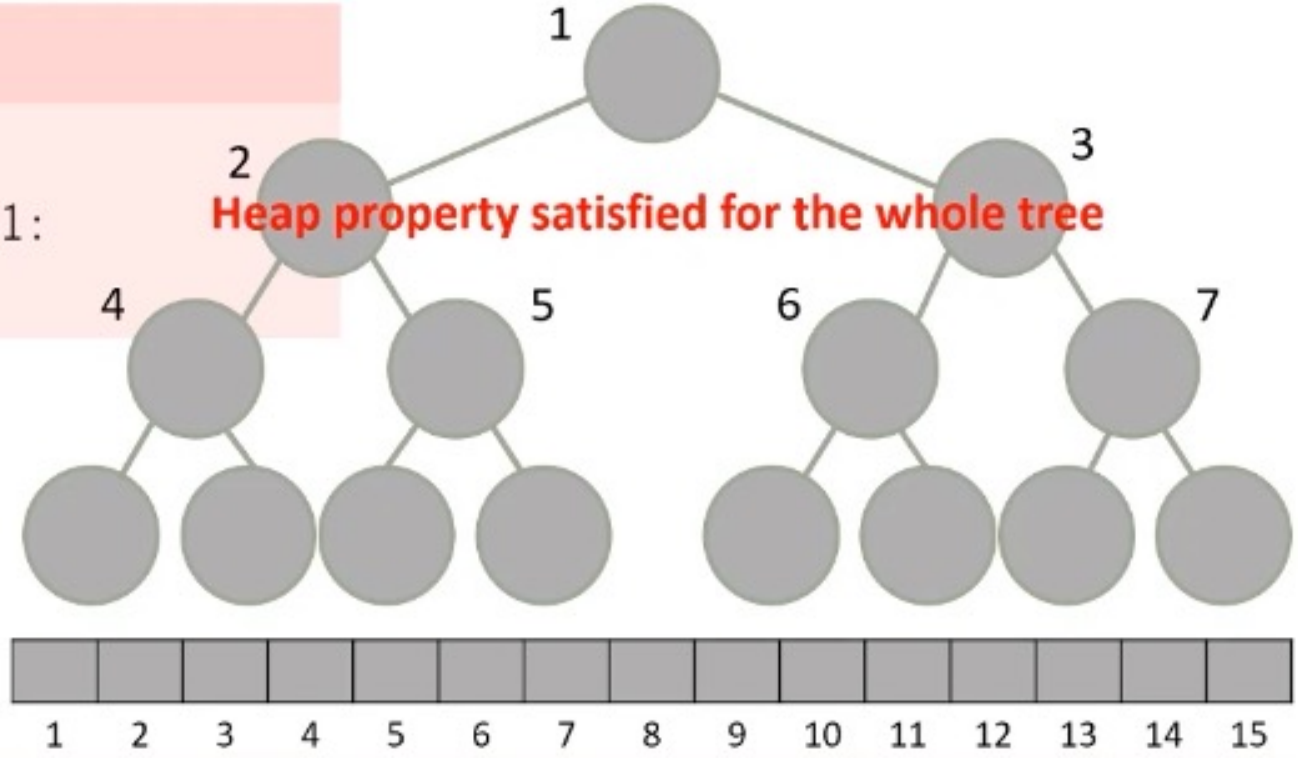


```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

Heap:

size = 15

nodes = 15, sub-trees = 15



- We repair the heap property going from bottom to top.

- We repair the heap property going from bottom to top.
- Initially, the heap property is satisfied in all the leaves (i.e., subtrees of depth 0).

- We repair the heap property going from bottom to top.
- Initially, the heap property is satisfied in all the leaves (i.e., subtrees of depth 0).
- We then start repairing the heap property in all subtrees of depth 1.

- We repair the heap property going from bottom to top.
- Initially, the heap property is satisfied in all the leaves (i.e., subtrees of depth 0).
- We then start repairing the heap property in all subtrees of depth 1.
- When we reach the root, the heap property is satisfied in the whole tree.

- We repair the heap property going from bottom to top.
- Initially, the heap property is satisfied in all the leaves (i.e., subtrees of depth 0).
- We then start repairing the heap property in all subtrees of depth 1.
- When we reach the root, the heap property is satisfied in the whole tree.
- Online visualization

- We repair the heap property going from bottom to top.
- Initially, the heap property is satisfied in all the leaves (i.e., subtrees of depth 0).
- We then start repairing the heap property in all subtrees of depth 1.
- When we reach the root, the heap property is satisfied in the whole tree.
- Online visualization
- Running time: $O(n \log n)$

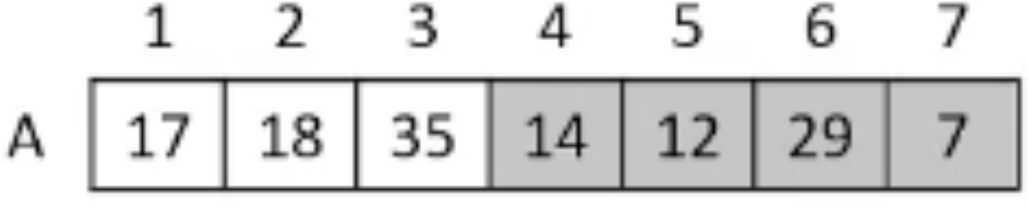
In-place Heap Sort

```
HeapSort(A[1...n])
BuildHeap(A)                                  {size = n}
repeat (n – 1) times:
   swap A[1] and A[size]
   size ← size – 1
   SiftDown(1)
```

	1	2	3	4	5	6	7
A	17	18	35	14	12	29	7

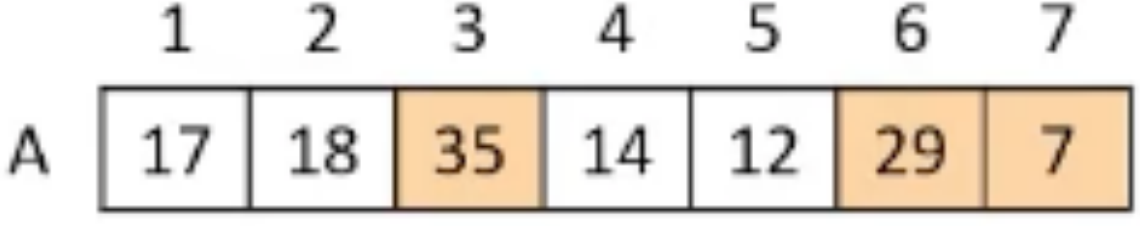
In-place Heap Sort

```
HeapSort(A[1 . . . n])
BuildHeap(A)                                  {size = n}
repeat (n − 1) times:
    swap A[1] and A[size]
    size ← size − 1
    SiftDown(1)
```



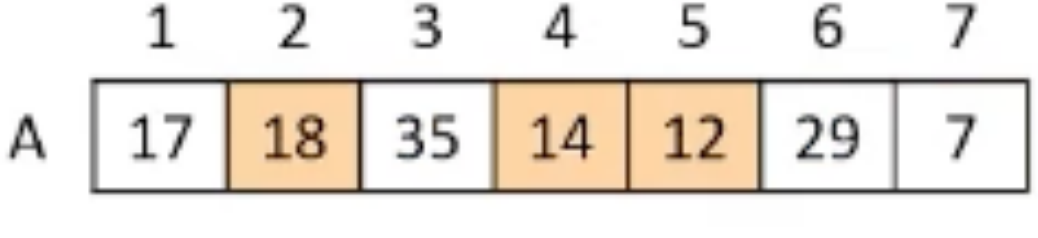
In-place Heap Sort

```
HeapSort(A[1 . . . n])
BuildHeap(A)                                  {size = n}
repeat (n − 1) times:
    swap A[1] and A[size]
    size ← size − 1
    SiftDown(1)
```



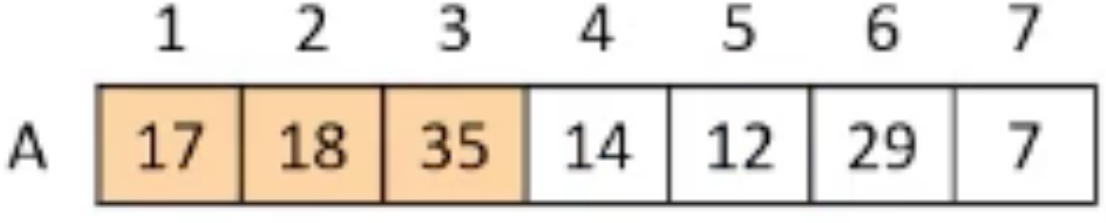
In-place Heap Sort

```
HeapSort(A[1...n])
BuildHeap(A)                                {size = n}
repeat (n − 1) times:
    swap A[1] and A[size]
    size ← size − 1
    SiftDown(1)
```



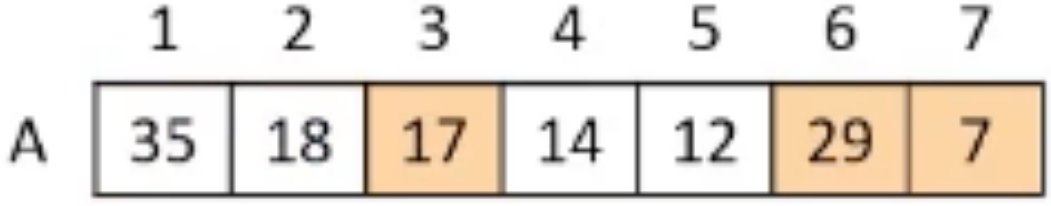
In-place Heap Sort

```
HeapSort(A[1 . . . n])
BuildHeap(A)                                  {size = n}
repeat (n − 1) times:
    swap A[1] and A[size]
    size ← size − 1
    SiftDown(1)
```



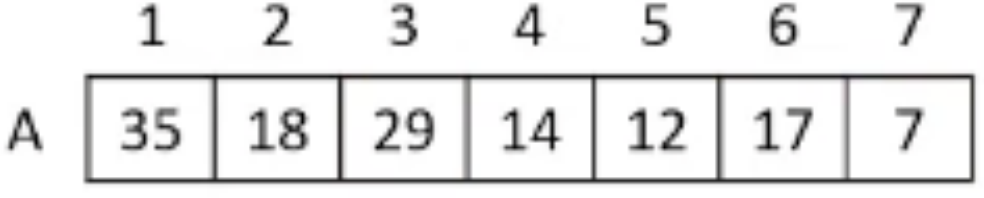
In-place Heap Sort

```
HeapSort(A[1 . . . n])
BuildHeap(A)                                  {size = n}
repeat (n − 1) times:
    swap A[1] and A[size]
    size ← size − 1
    SiftDown(1)
```



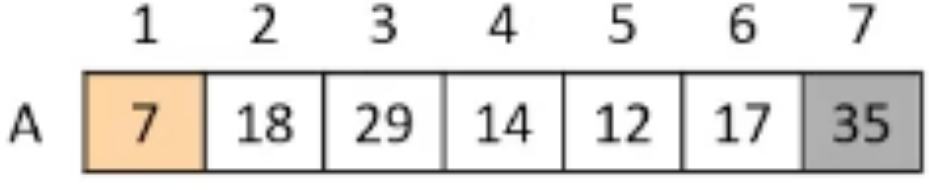
In-place Heap Sort

```
HeapSort(A[1 . . . n])
BuildHeap(A)                                {size = n}
repeat (n – 1) times:
   swap A[1] and A[size]
   size ← size – 1
   SiftDown(1)
```



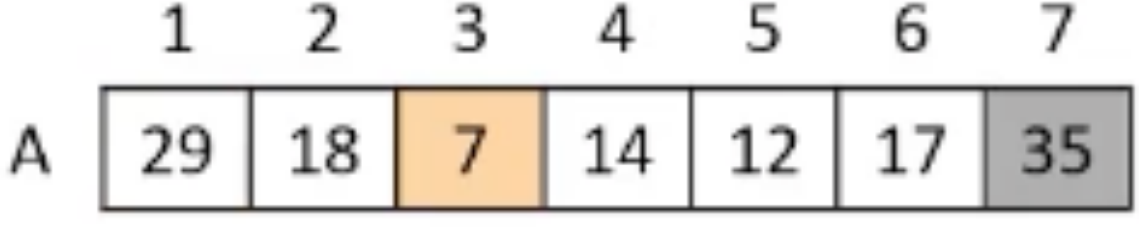
In-place Heap Sort

```
HeapSort(A[1 . . . n])
BuildHeap(A)                                  {size = n}
repeat (n − 1) times:
    swap A[1] and A[size]
    size ← size − 1
    SiftDown(1)
```



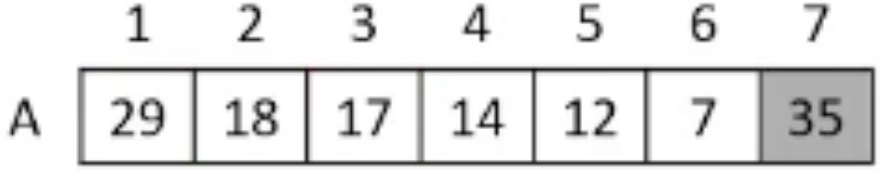
In-place Heap Sort

```
HeapSort(A[1 . . . n])
BuildHeap(A)                                  {size = n}
repeat (n − 1) times:
   swap A[1] and A[size]
   size ← size − 1
   SiftDown(1)
```



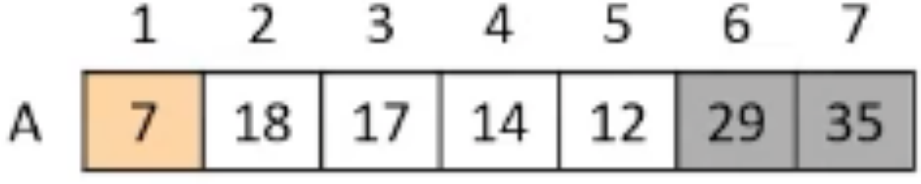
In-place Heap Sort

```
HeapSort(A[1 . . . n])
BuildHeap(A)                                  {size = n}
repeat (n − 1) times:
   swap A[1] and A[size]
   size ← size − 1
   SiftDown(1)
```



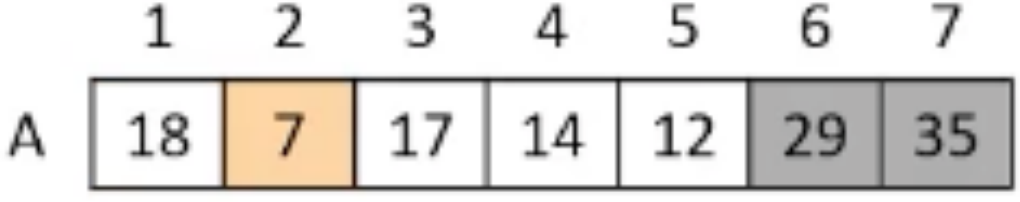
In-place Heap Sort

```
HeapSort(A[1 . . . n])
BuildHeap(A)                                  {size = n}
repeat (n − 1) times:
    swap A[1] and A[size]
    size ← size − 1
    SiftDown(1)
```



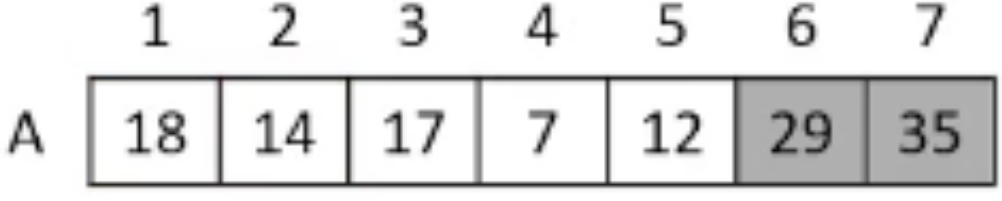
In-place Heap Sort

```
HeapSort(A[1 . . . n])
BuildHeap(A)                                  {size = n}
repeat (n − 1) times:
    swap A[1] and A[size]
    size ← size − 1
    SiftDown(1)
```



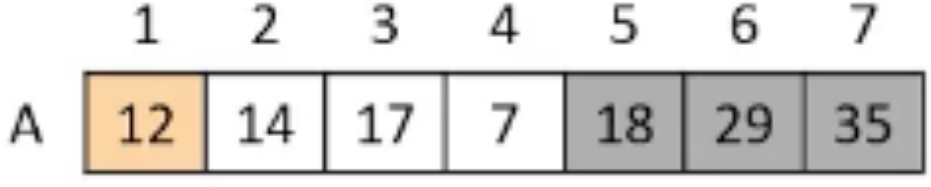
In-place Heap Sort

```
HeapSort(A[1 . . . n])
BuildHeap(A)                                  {size = n}
repeat (n − 1) times:
    swap A[1] and A[size]
    size ← size − 1
    SiftDown(1)
```



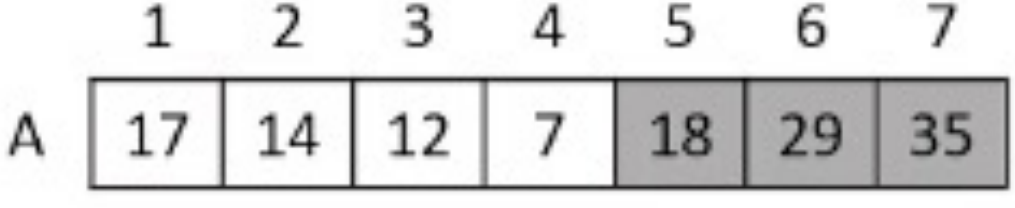
In-place Heap Sort

```
HeapSort(A[1...n])
BuildHeap(A)                                {size = n}
repeat (n − 1) times:
    swap A[1] and A[size]
    size ← size − 1
    SiftDown(1)
```



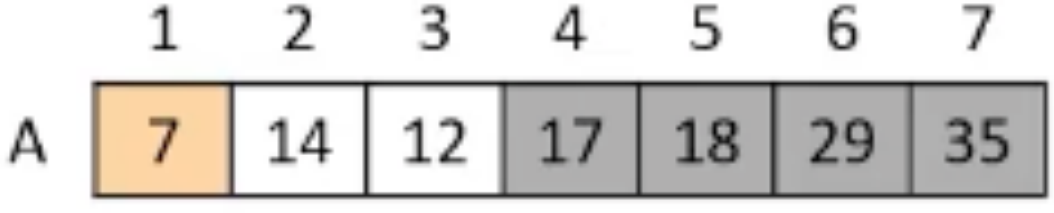
In-place Heap Sort

```
HeapSort(A[1...n])
BuildHeap(A)                                {size = n}
repeat (n − 1) times:
    swap A[1] and A[size]
    size ← size − 1
    SiftDown(1)
```



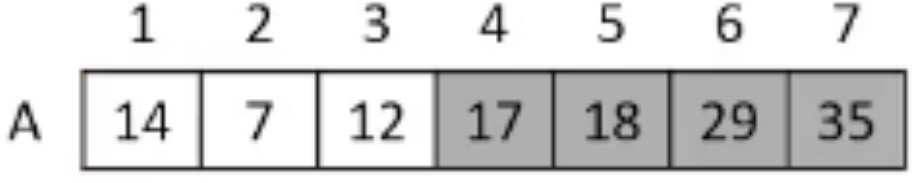
In-place Heap Sort

```
HeapSort(A[1...n])
BuildHeap(A)                                  {size = n}
repeat (n – 1) times:
   swap A[1] and A[size]
   size ← size – 1
   SiftDown(1)
```



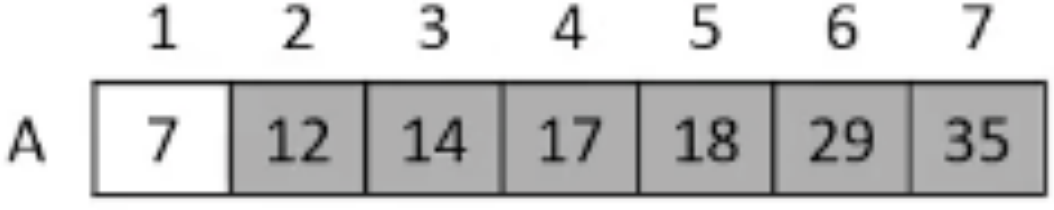
In-place Heap Sort

```
HeapSort(A[1 . . . n])
BuildHeap(A)                                  {size = n}
repeat (n − 1) times:
    swap A[1] and A[size]
    size ← size − 1
    SiftDown(1)
```



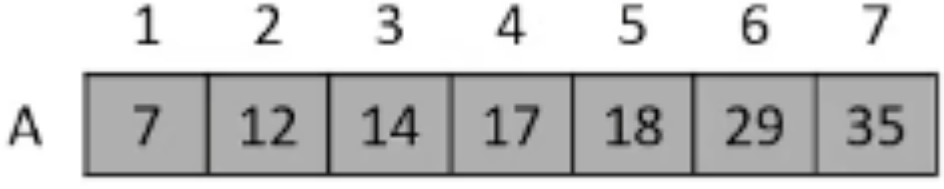
In-place Heap Sort

```
HeapSort(A[1 . . . n])
BuildHeap(A)                                {size = n}
repeat (n − 1) times:
    swap A[1] and A[size]
    size ← size − 1
    SiftDown(1)
```



In-place Heap Sort

```
HeapSort(A[1...n])
BuildHeap(A)                                  {size = n}
repeat (n − 1) times:
    swap A[1] and A[size]
    size ← size − 1
    SiftDown(1)
```



Building Running Time

- The running time of `BuildHeap` is $O(n \log n)$ since we call `SiftDown` for $O(n)$ nodes.

Building Running Time

- The running time of `BuildHeap` is $O(n \log n)$ since we call `SiftDown` for $O(n)$ nodes.
- If a node is already close to the leaves, then sifting it down is fast.

Building Running Time

- The running time of `BuildHeap` is $O(n \log n)$ since we call `SiftDown` for $O(n)$ nodes.
- If a node is already close to the leaves, then sifting it down is fast.
- We have many such nodes!

Building Running Time

- The running time of `BuildHeap` is $O(n \log n)$ since we call `SiftDown` for $O(n)$ nodes.
- If a node is already close to the leaves, then sifting it down is fast.
- We have many such nodes!
- Was our estimate of the running time of `BuildHeap` too pessimistic?

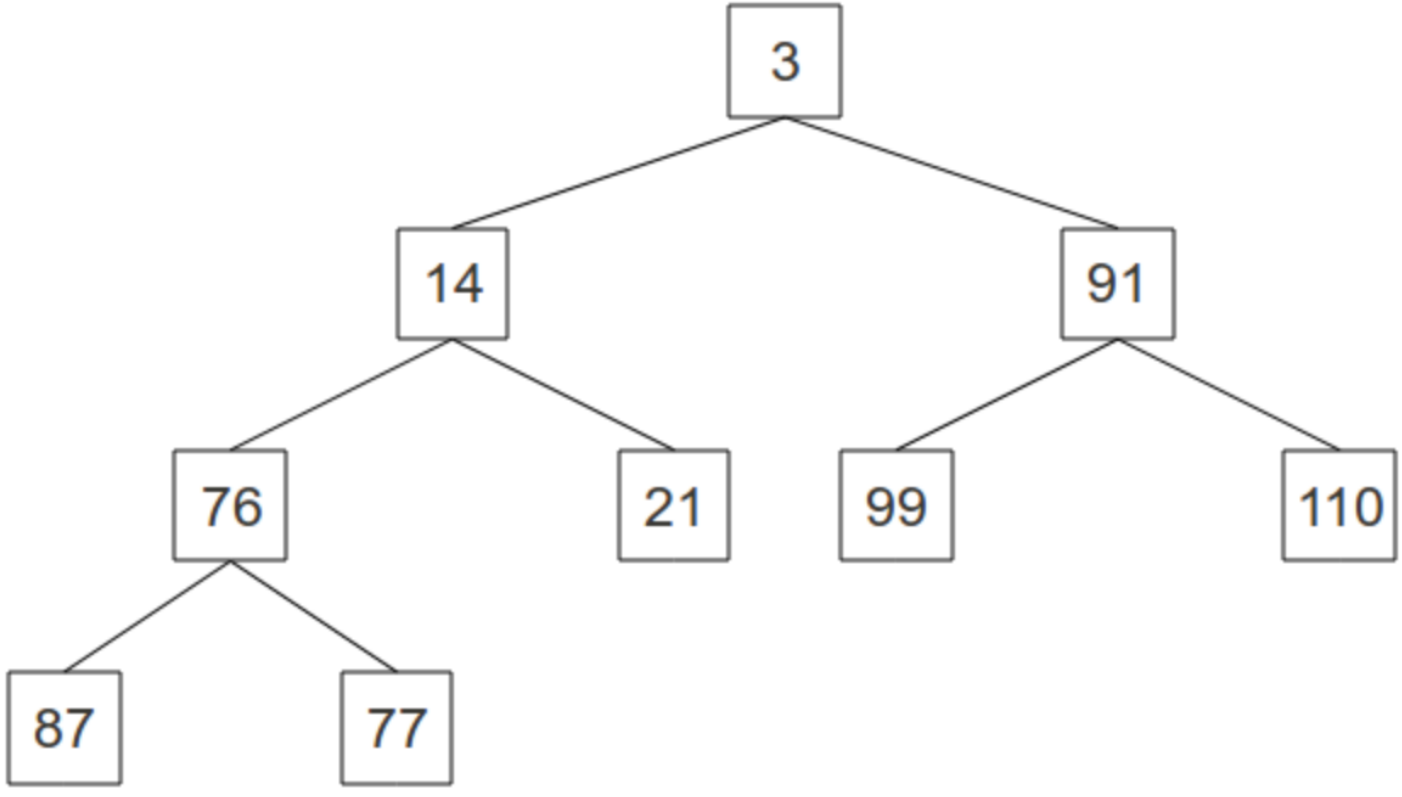
Question

The array A below represents a binary heap implementation of a priority queue of items. Only the integer key values of the items are shown. The priority queue is arranged with the lowest-value key first.

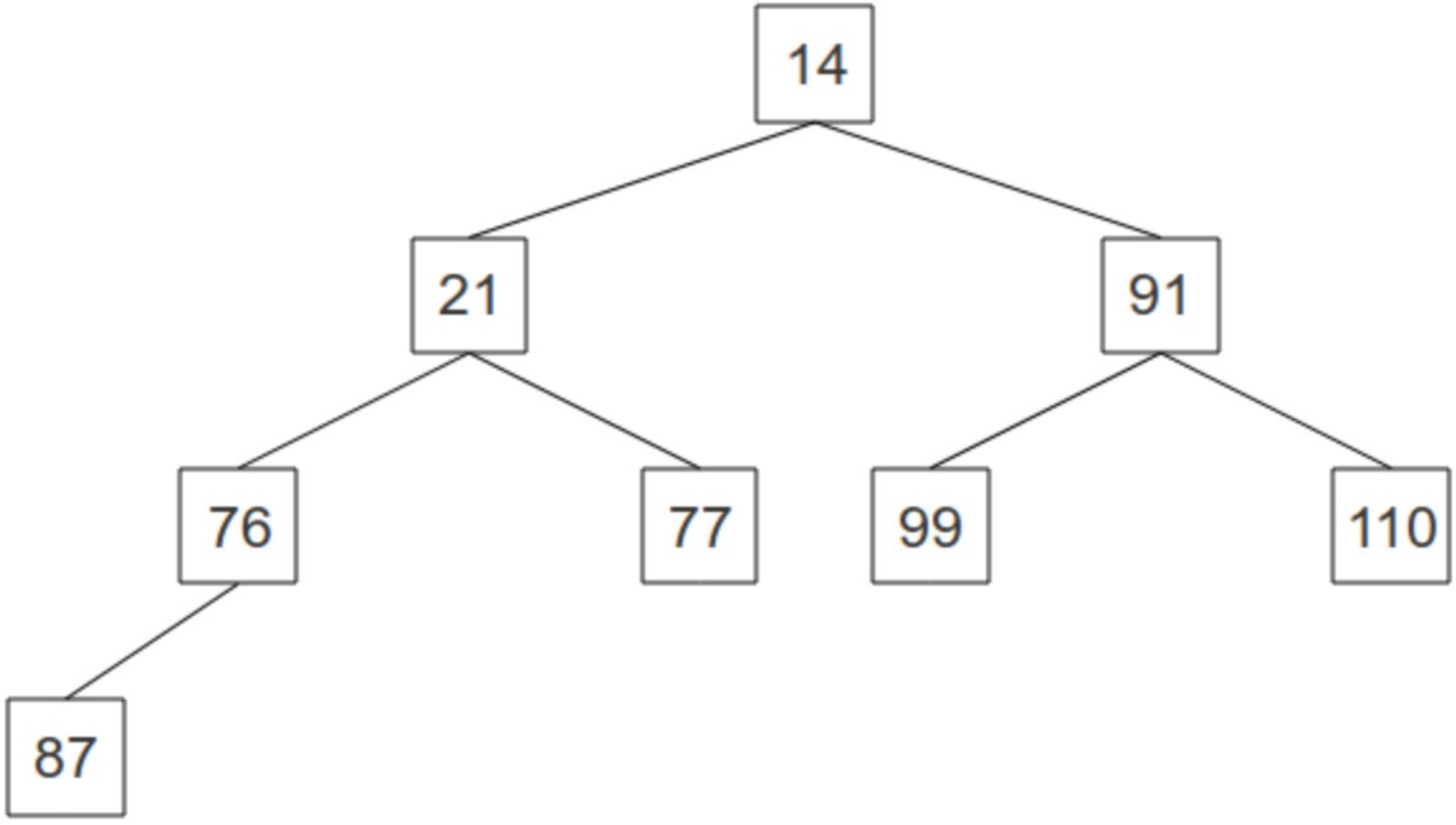
	0	1	2	3	4	5	6	7	8	9
A		3	14	91	76	21	99	110	87	77

(a) Draw the binary heap structure (tree) represented by A.
(b) Draw the heap structure after the lowest-key item has been removed.
(c) Draw the heap structure after a new item with key **1** has been added.

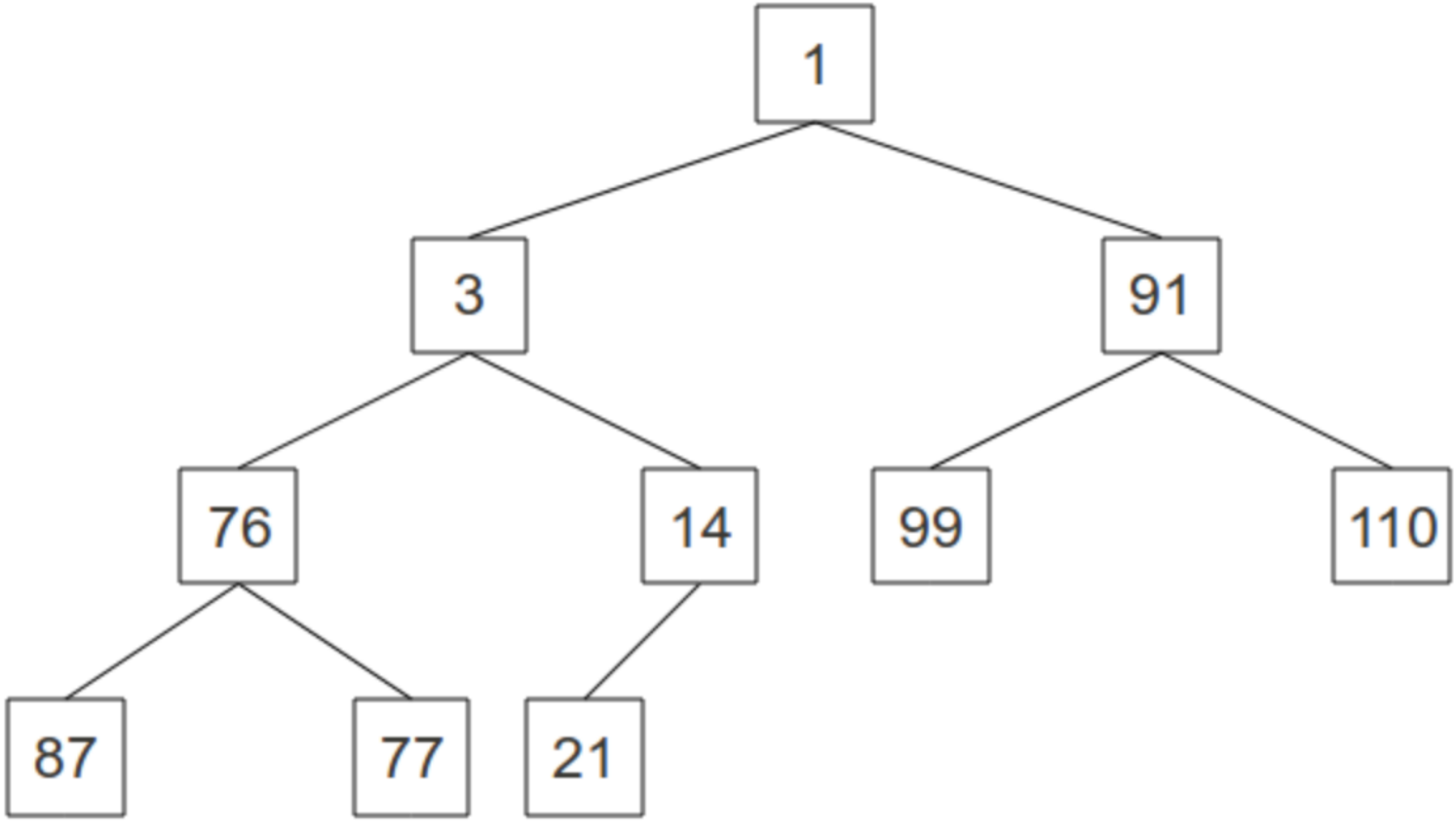
Answer (a)



Answer (b)

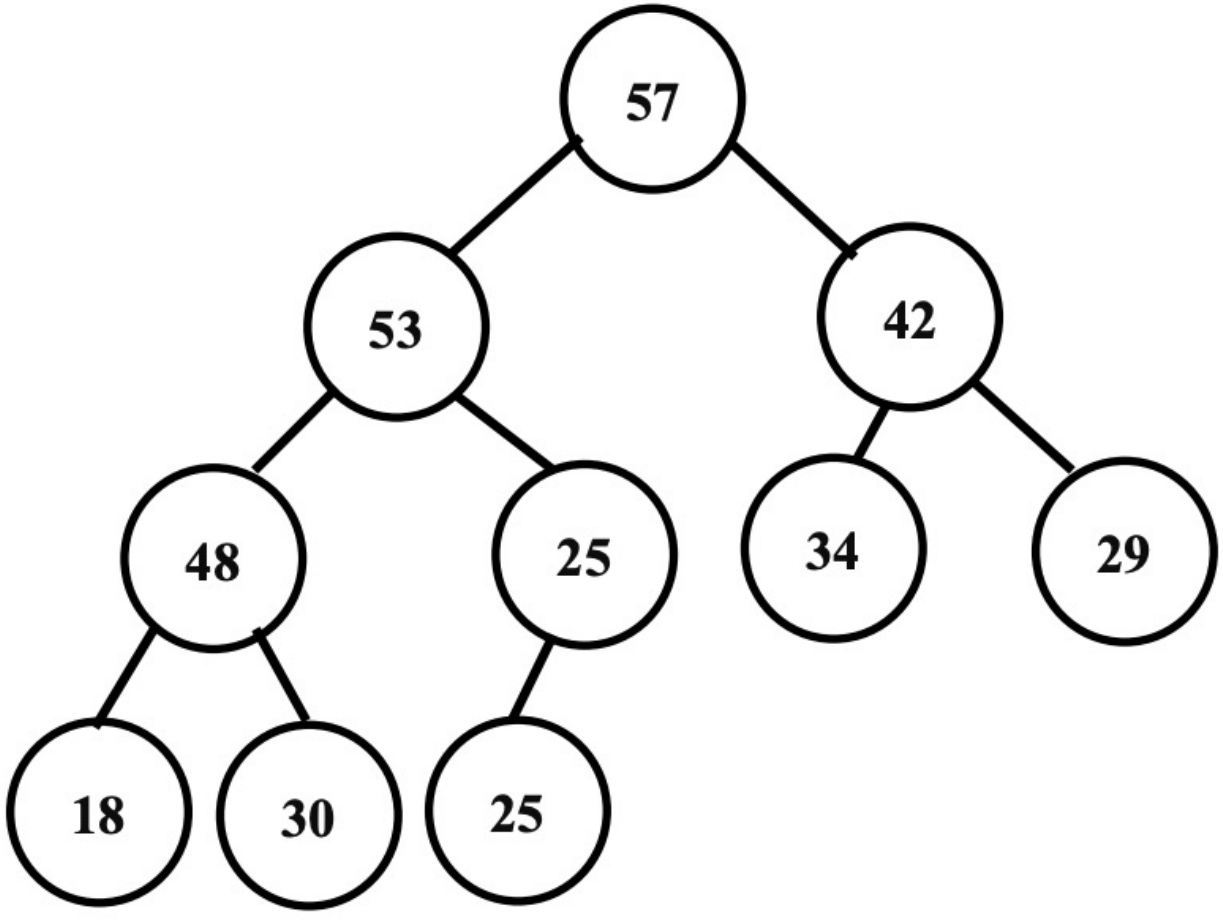


Answer (c)

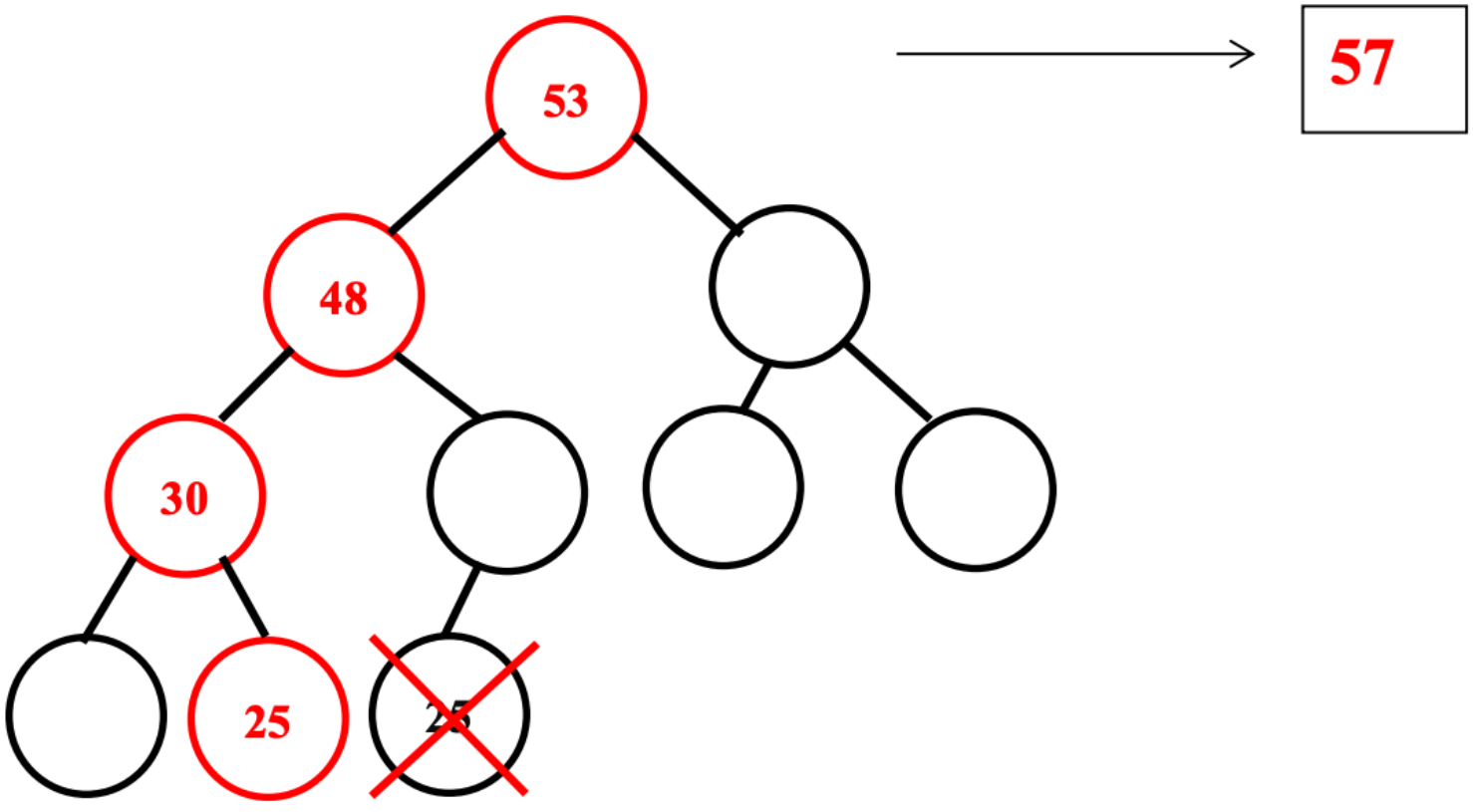


Question

Perform one remove() operation on the heap below.

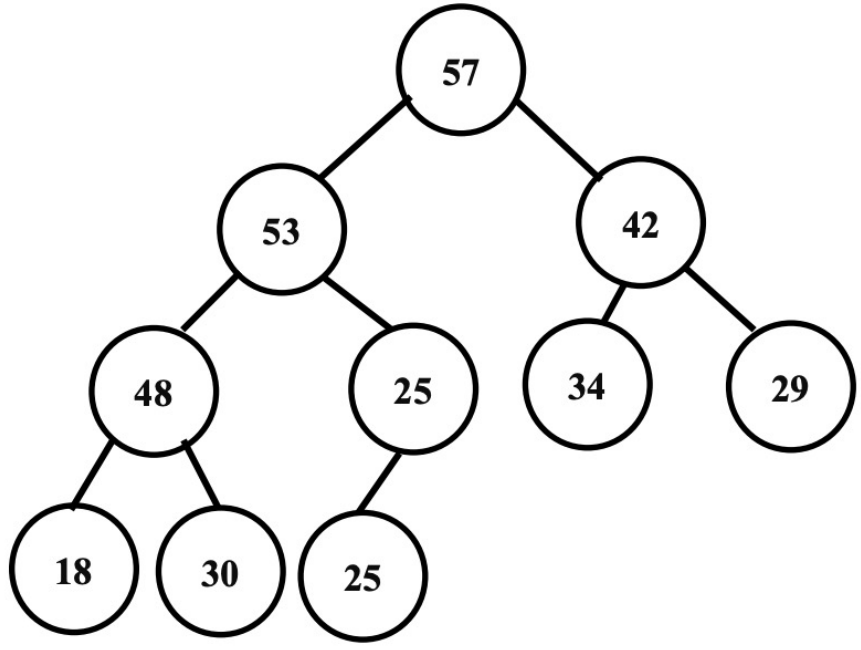


Solution



Question

(a) Insert 55 in the max-heap below. Show all the changes to the heap.

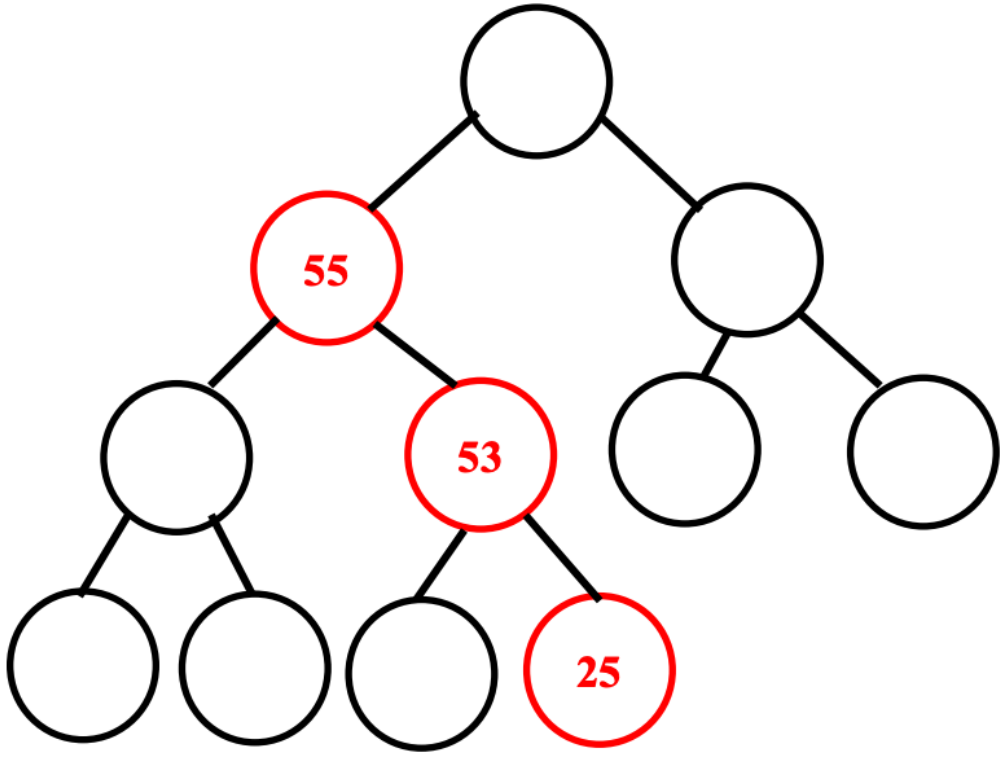


(b) Show the array that stores the above heap. Start at index 1.

(c) Rewrite the array (for the above heap) after each change it goes through during the insert 55 operation.

Solution

Insert 55 in the max-heap below. Show all the changes to the heap.



Solution

(b) Show the array that stores the above heap. Start at index 1.

(c) Rewrite the array (for the above heap) after each change it goes through during the insert 55 operation.

	0	1	2	3	4	5	6	7	8	9	10	11
Orig		57	53	42	48	25	34	29	18	30	25	
Step 1												**55**
Step 2						**55**						**25**
Step 3			**55**			**53**						