# Priority Queues: Binary Heaps

Alexander S. Kulikov

Steklov Institute of Mathematics at St. Petersburg
Russian Academy of Sciences

Data Structures
Data Structures and Algorithms

# Outline

### 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.

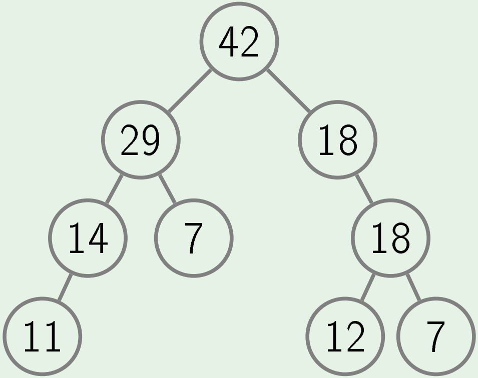
## Definition

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.

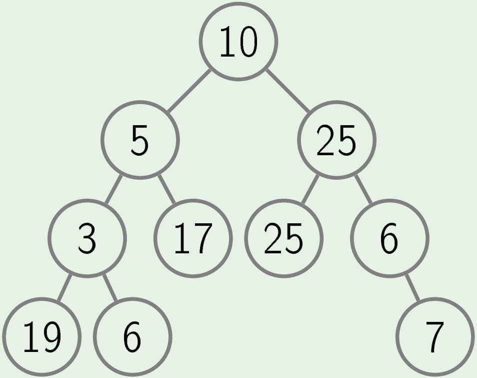
## In other words

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

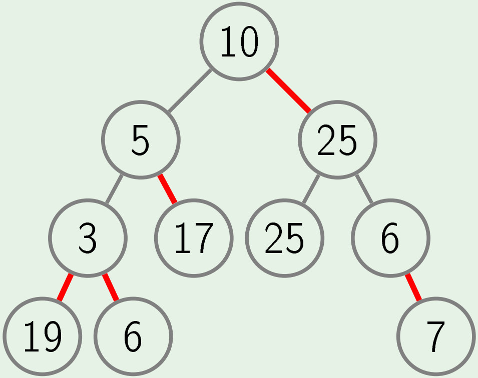
## Example: heap



## Example: not a heap

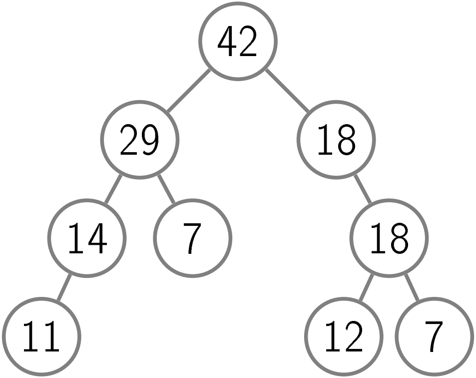


# Example: not a heap



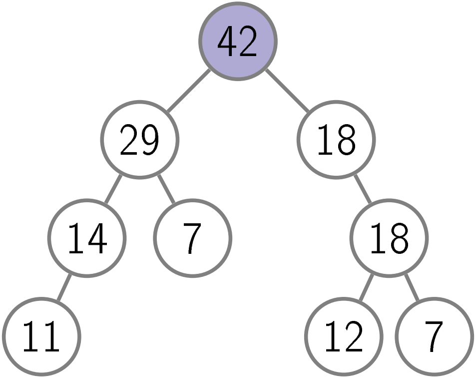
# Outline

# GetMax

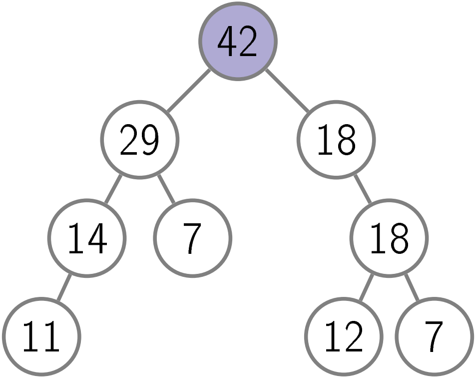


# GetMax

return the root value



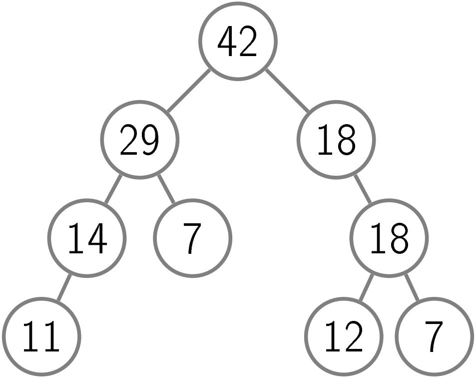
# GetMax




return the root value

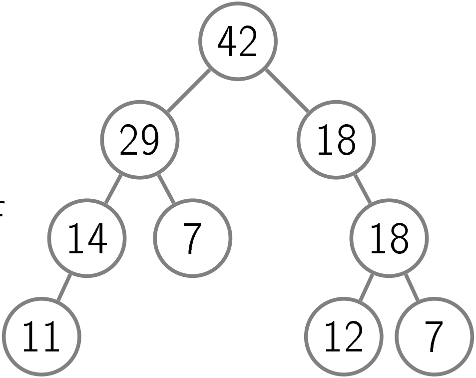
running time: $O(1)$

# Insert



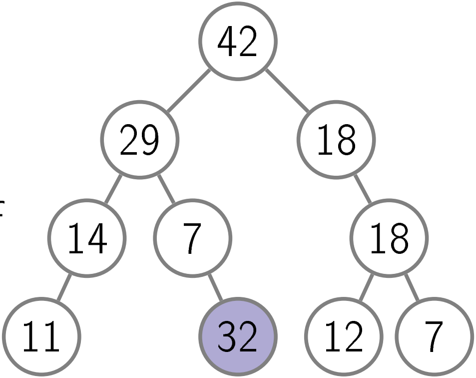
# Insert

attach a new
node to any leaf



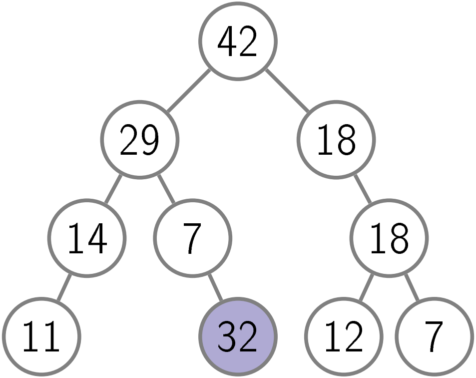
# Insert

attach a new
node to any leaf



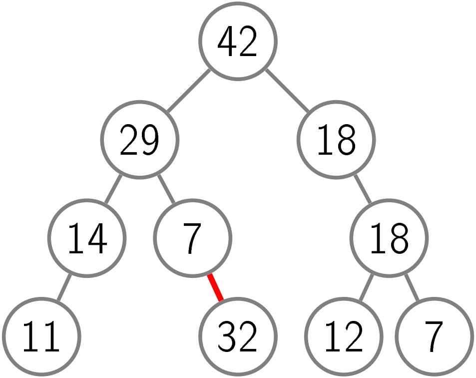
# Insert

this may violate the heap property



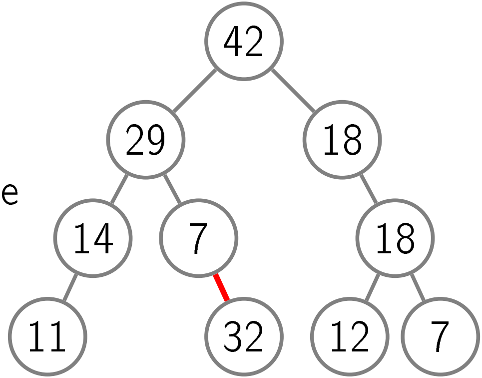
# Insert

this may violate the heap property



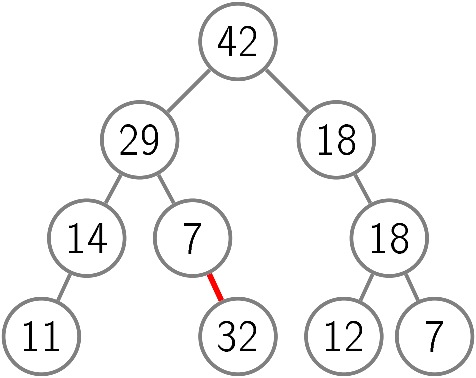
# Insert

to fix this, we let the new node sift up

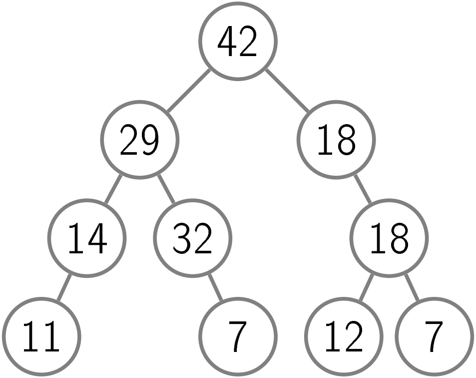


# SiftUp

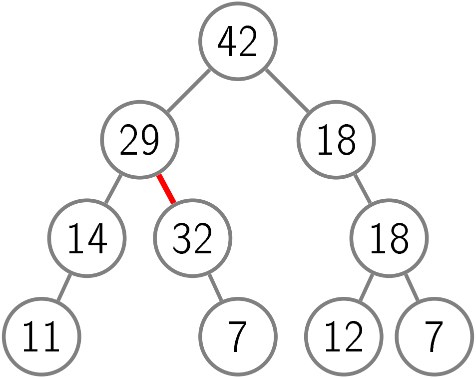
for this, we swap the problematic node with its parent until the property is satisfied



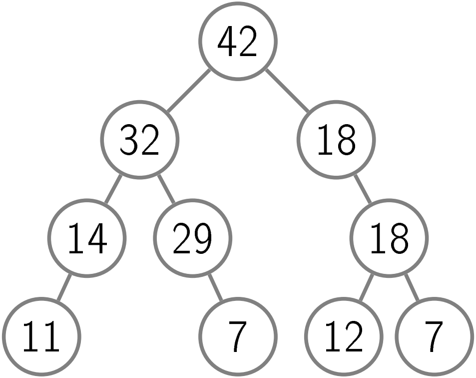
# SiftUp



# SiftUp

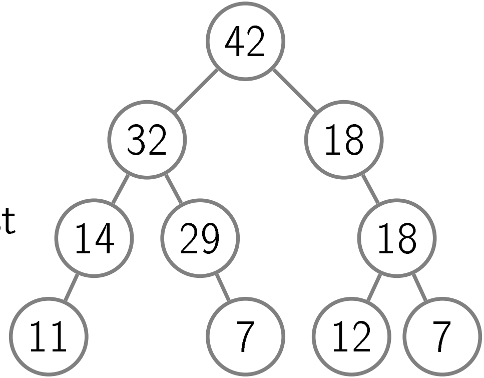


# SiftUp



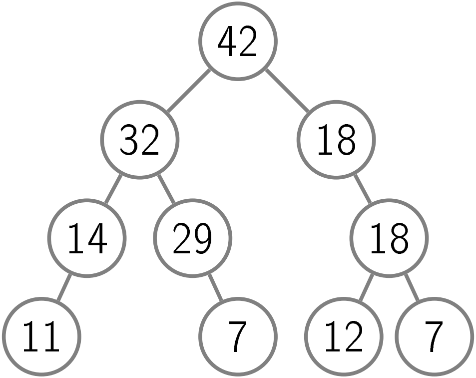
# SiftUp

invariant: heap property is violated on at most one edge

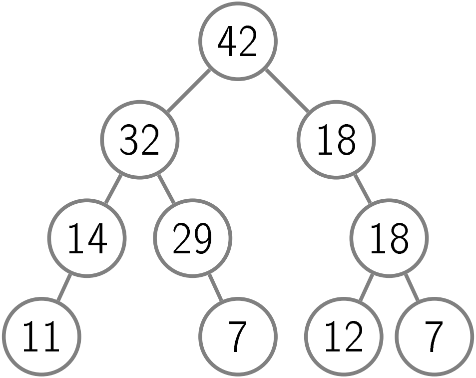


# SiftUp

this edge gets closer to the root while sifting up

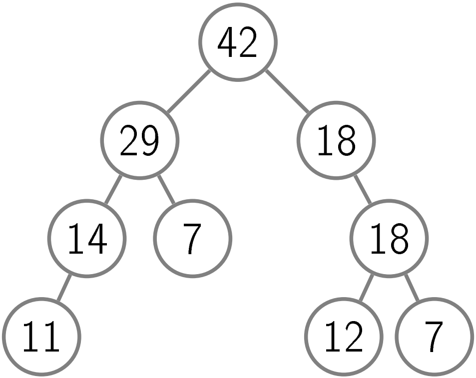


# SiftUp



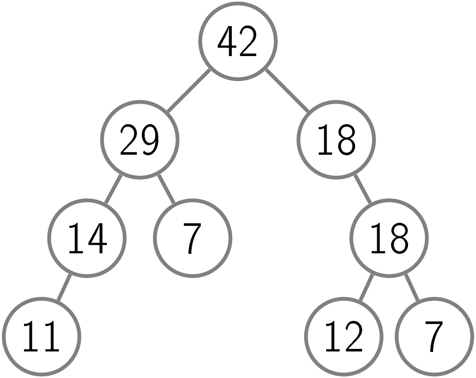

running time: $O$(tree height)

# ExtractMax



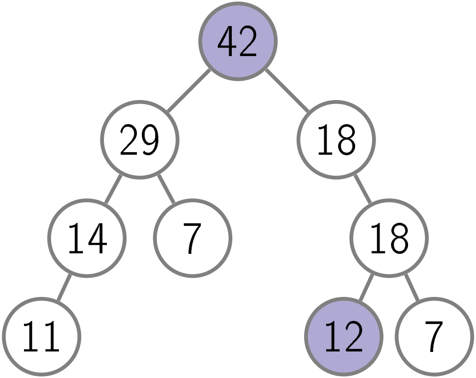
# ExtractMax

replace the root
with any leaf



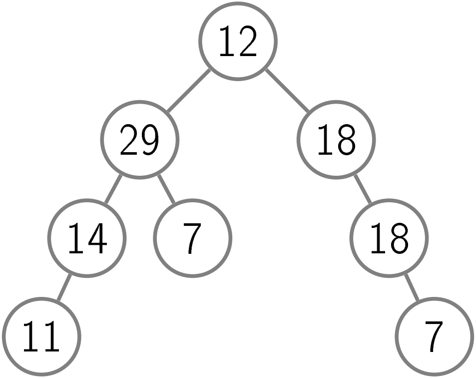
# ExtractMax

replace the root
with any leaf



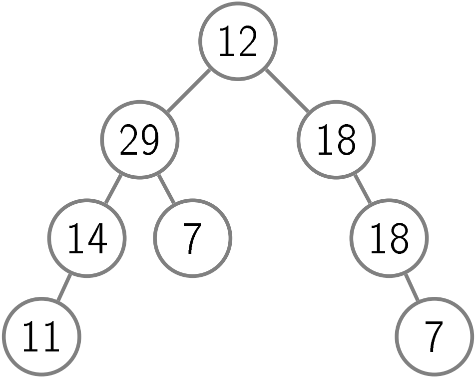
# ExtractMax

replace the root
with any leaf



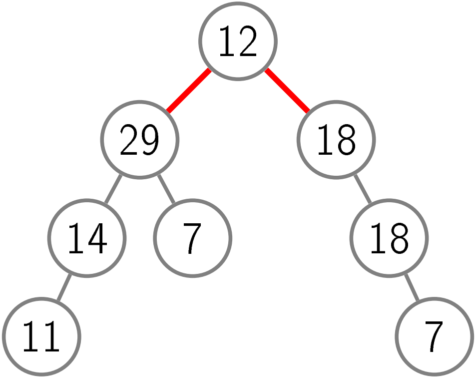
# ExtractMax

again, this may violate the heap property



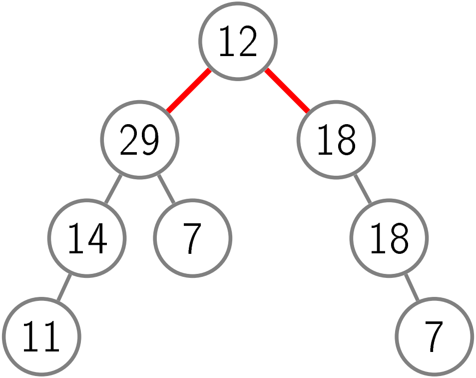
# ExtractMax

again, this may violate the heap property



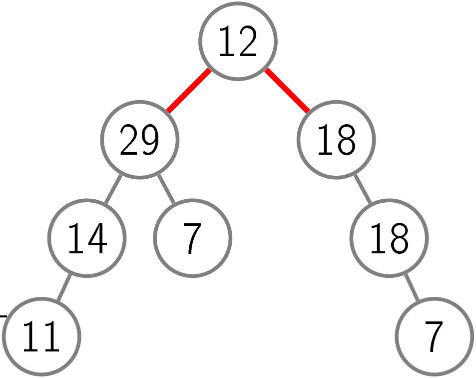
# ExtractMax

to fix it, we let the problematic node sift down

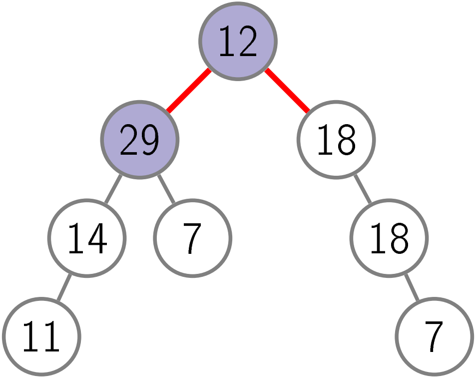


# SiftDown

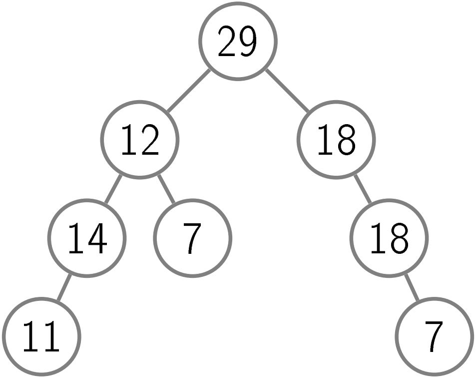
for this, we swap the problematic node with larger child until the heap property is satisfied



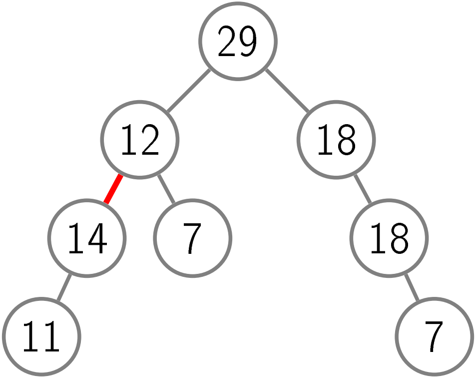
# SiftDown



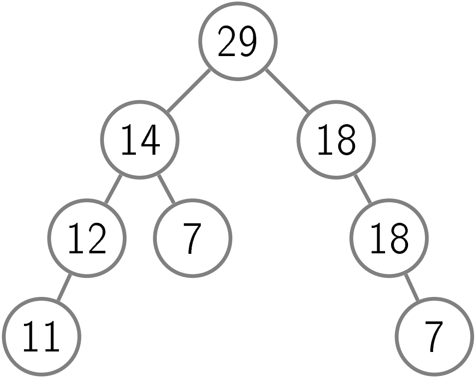
# SiftDown



# SiftDown

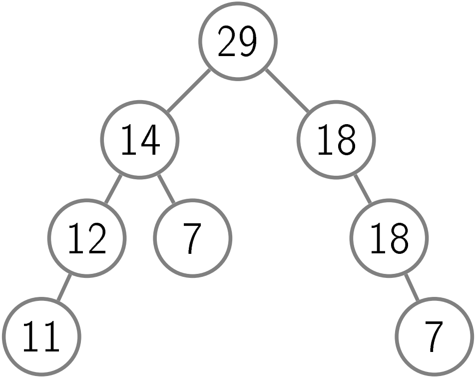


# SiftDown

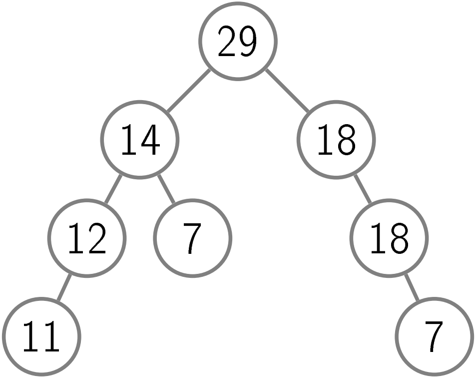


# SiftDown

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

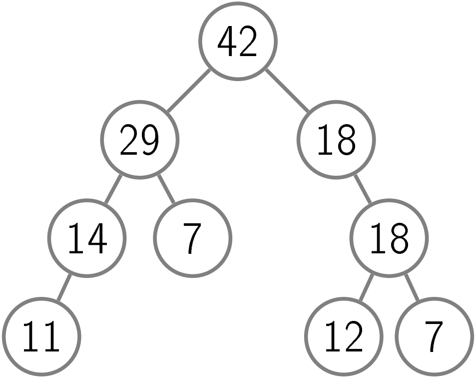


# SiftDown



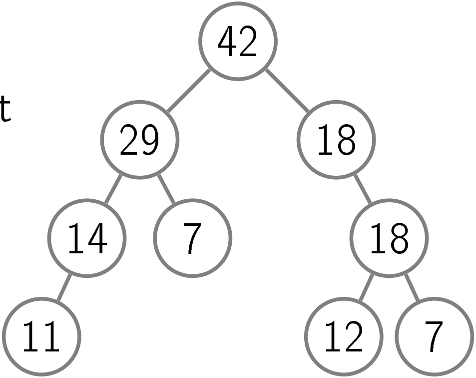

running time: $O(\text{tree height})$

# ChangePriority



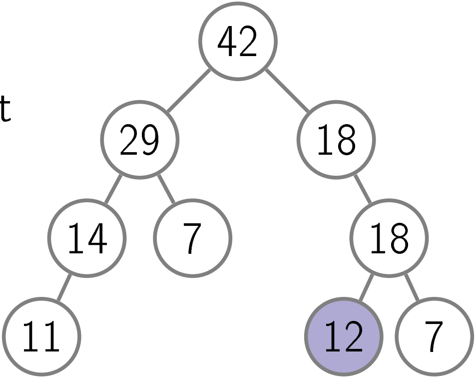
# ChangePriority

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



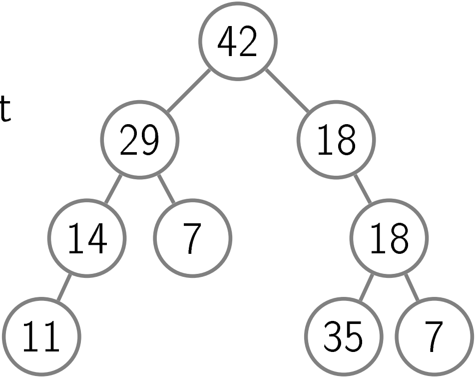
# ChangePriority

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

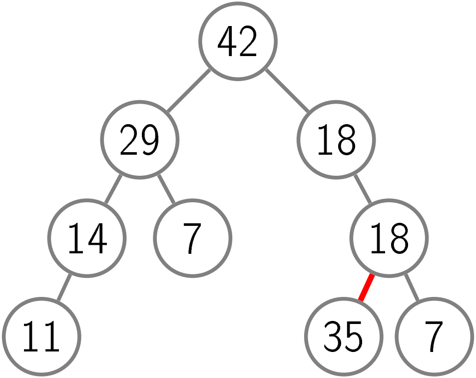


# ChangePriority

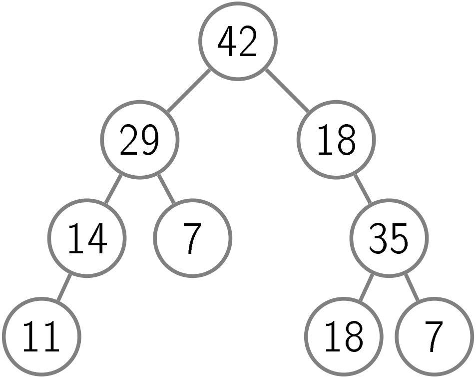
change the priority and let the changed element sift up or down depending on whether its priority decreased or increased



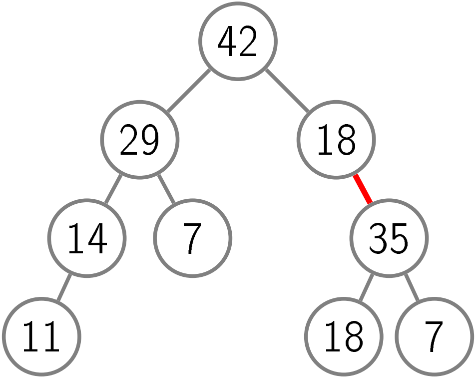
# ChangePriority



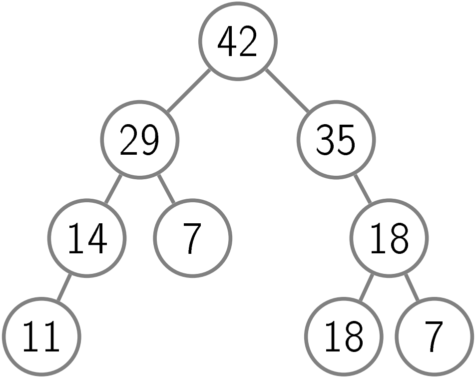
# ChangePriority



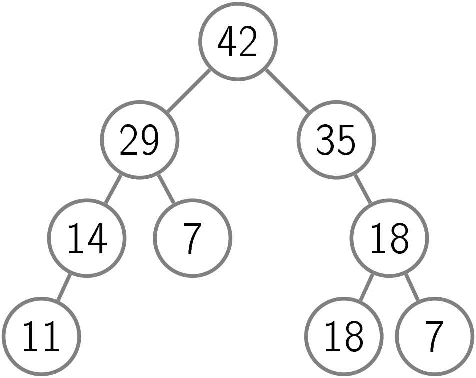
# ChangePriority



# ChangePriority

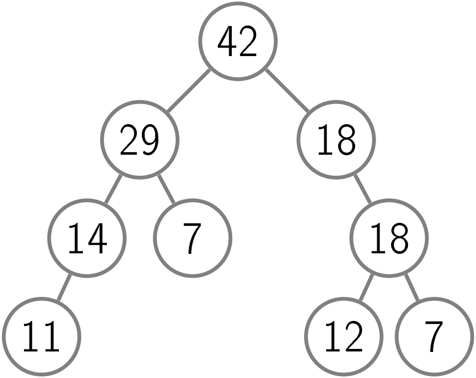


# ChangePriority



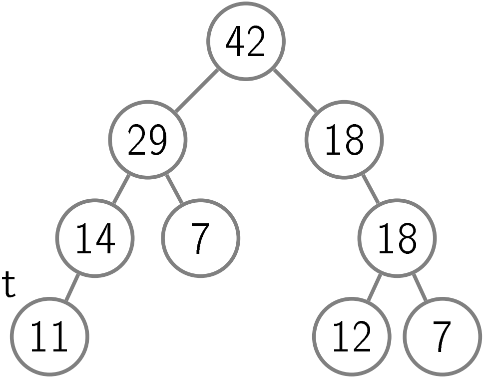

running time: $O$(tree height)

# Remove

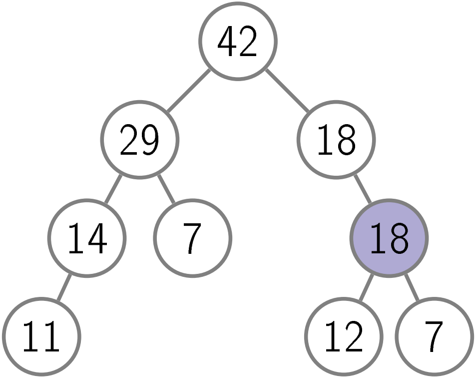


# Remove

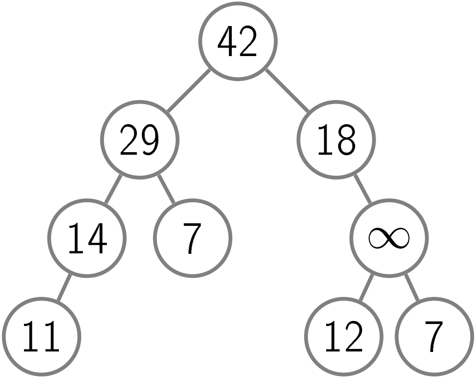
change the priority of the element to $\infty$, let it sift up, and then extract maximum



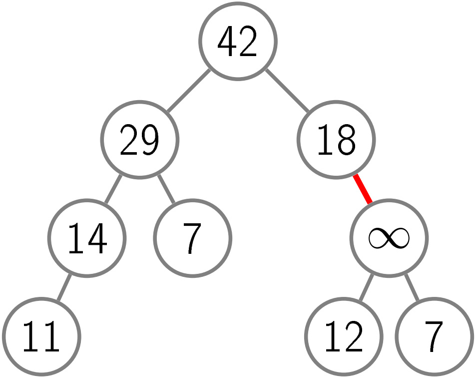
# Remove



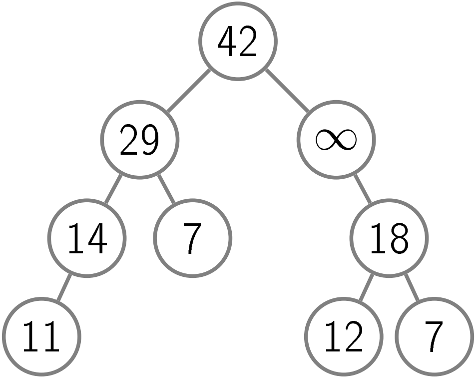
# Remove



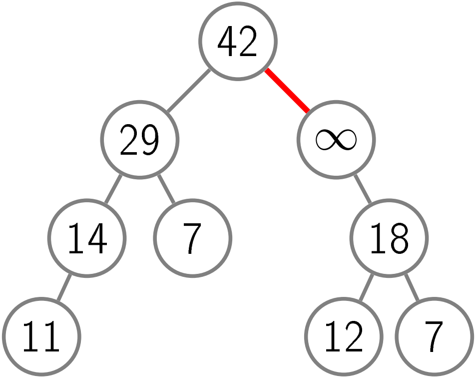
# Remove



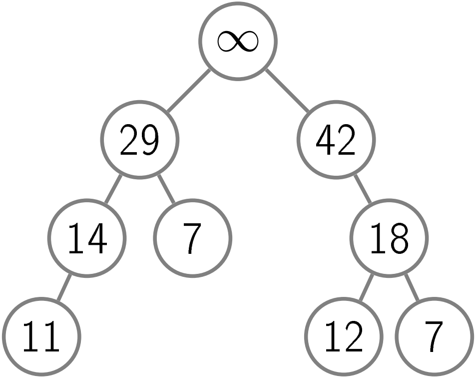
# Remove



# Remove



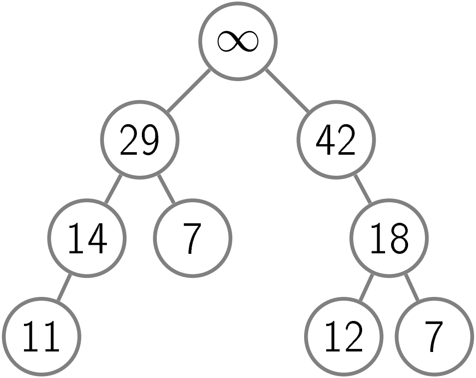
# Remove



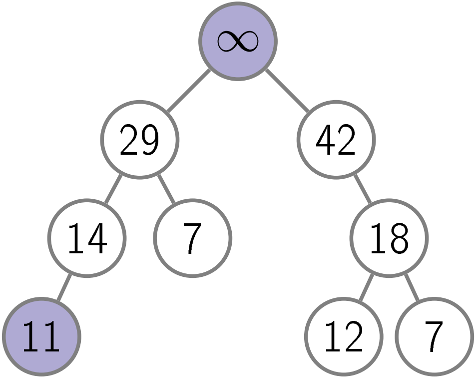
# Remove

now, call

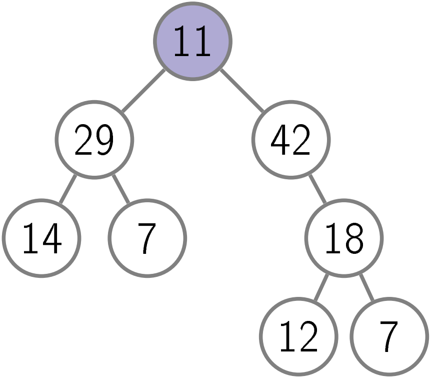
`ExtractMax()`



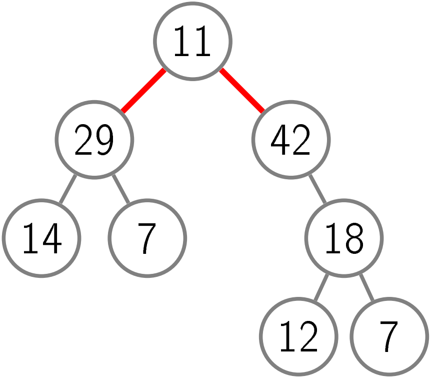
# Remove



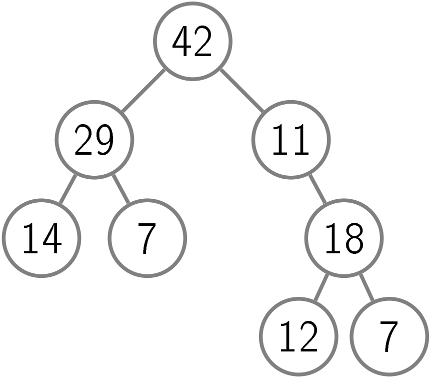
# Remove



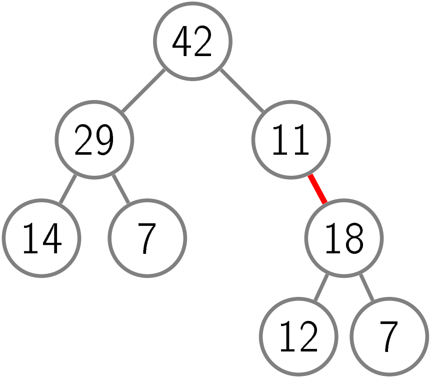
# Remove



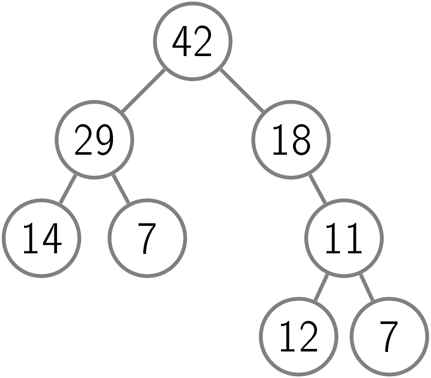
# Remove



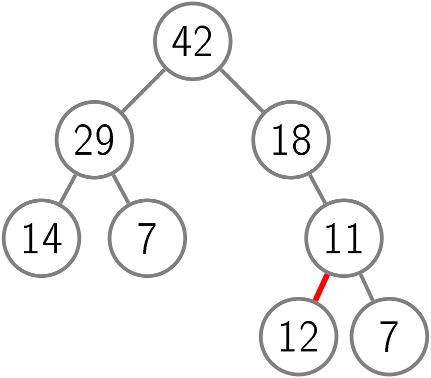
# Remove



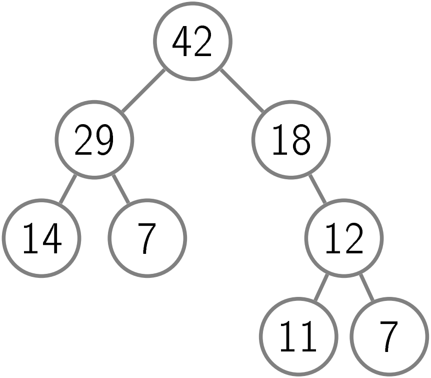
# Remove



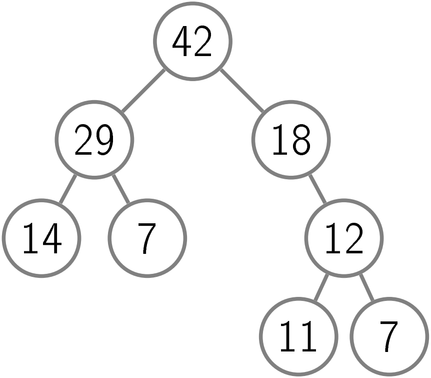
# Remove



# Remove



# Remove




running time: $O(\text{tree height})$

# Summary

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

# Summary

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

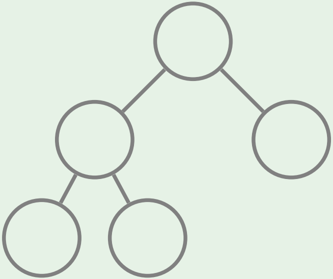
# Outline

# How to Keep a Tree Shallow?

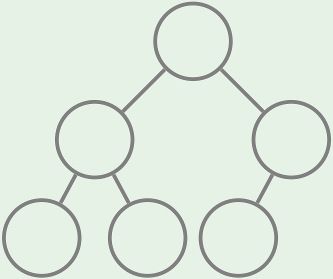
## Definition

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

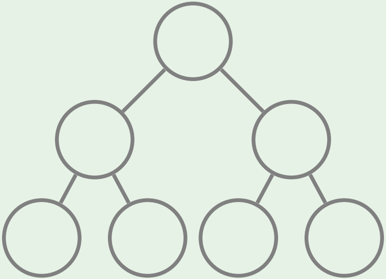
# Example: complete binary tree



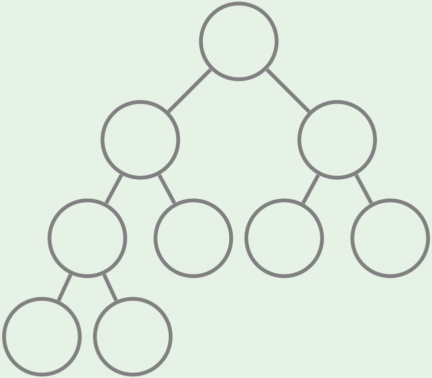
# Example: complete binary tree



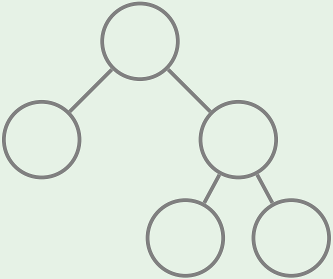
# Example: complete binary tree



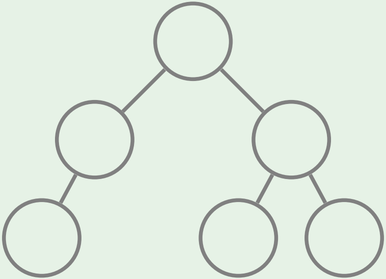
# Example: complete binary tree



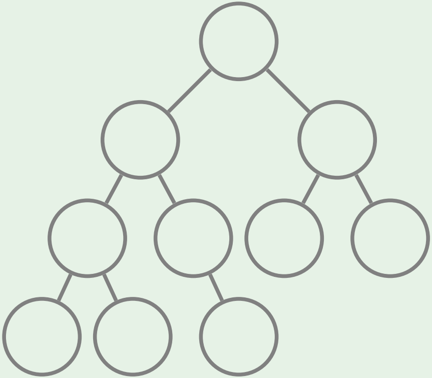
## Example: not complete binary tree



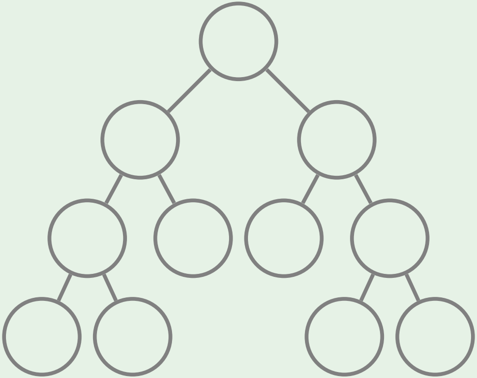
# Example: not complete binary tree



# Example: not complete binary tree



# Example: not complete binary tree



# First Advantage: Low Height

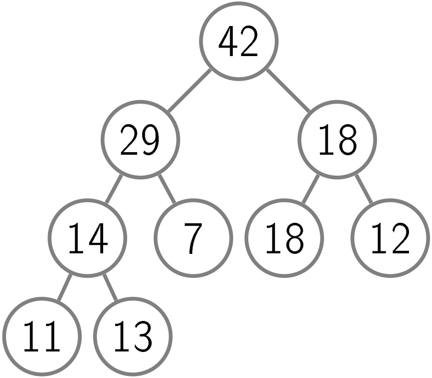
**Lemma**

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

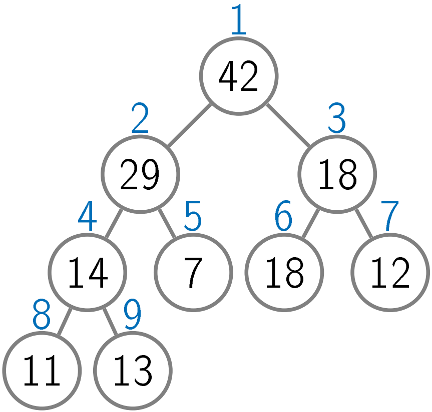
## Proof

- Complete the last level to get a full binary tree on $n' \geq n$ nodes and the same number of levels $\ell$.
- Note that $n' \leq 2n$.
- Then $n' = 2^\ell - 1$ and hence $\ell = \log_2(n'+1) \leq \log_2(2n+1) = O(\log n)$. □

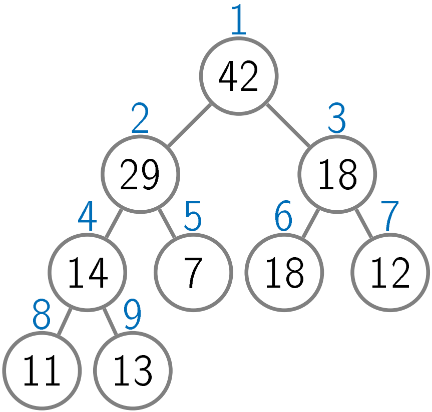
# Second Advantage: Store as Array



# Second Advantage: Store as Array



# Second Advantage: Store as Array

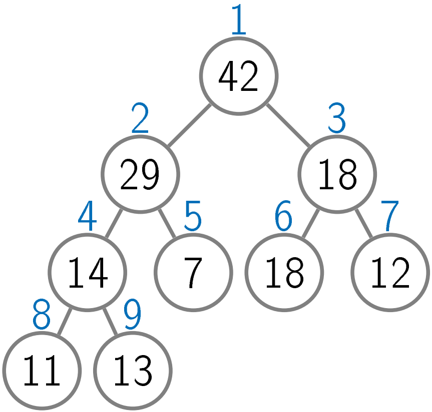



$$\text{parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor$$

$$\text{leftchild}(i) = 2i$$

$$\text{rightchild}(i) = 2i + 1$$

# Second Advantage: Store as Array




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

$$\text{leftchild}(i) = 2i$$

$$\text{rightchild}(i) = 2i + 1$$

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

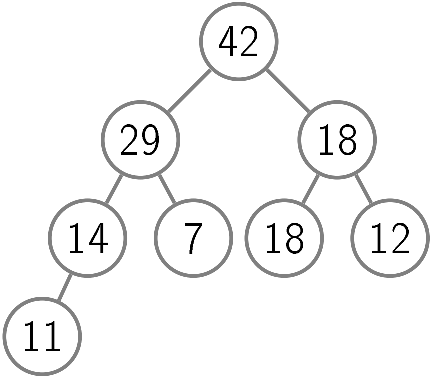
- What do we pay for these advantages?

- 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?

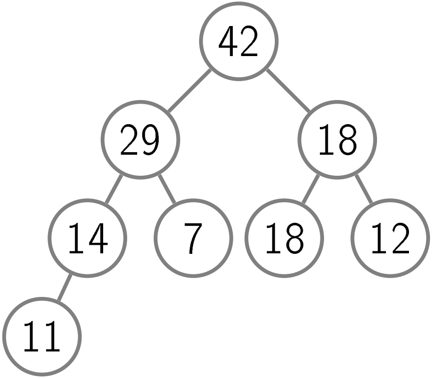
- What do we pay for these advantages?
- We need to keep the tree complete.
- Which binary heap operations modify the shape of the tree?
- Only `Insert` and `ExtractMax` (`Remove` changes the shape by calling `ExtractMax`).

# Keeping the Tree Complete



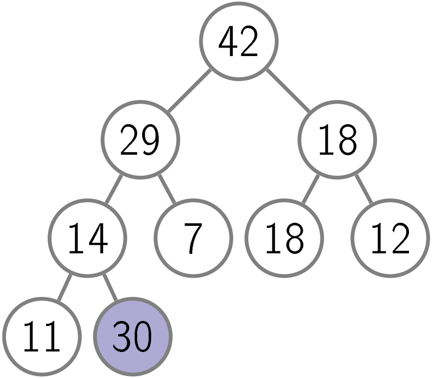
# Keeping the Tree Complete

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



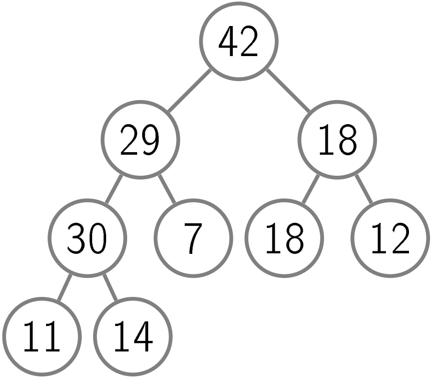
# Keeping the Tree Complete

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



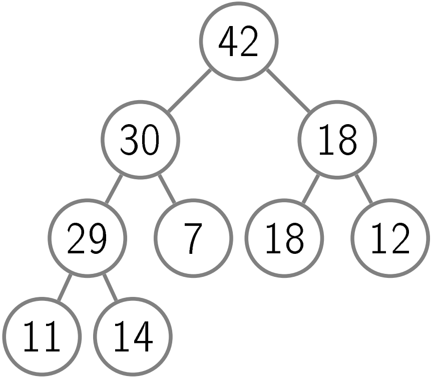
# Keeping the Tree Complete

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



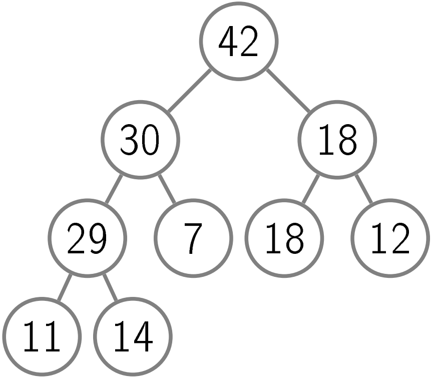
# Keeping the Tree Complete

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



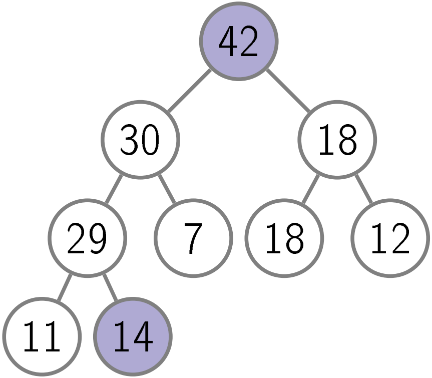
# Keeping the Tree Complete

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



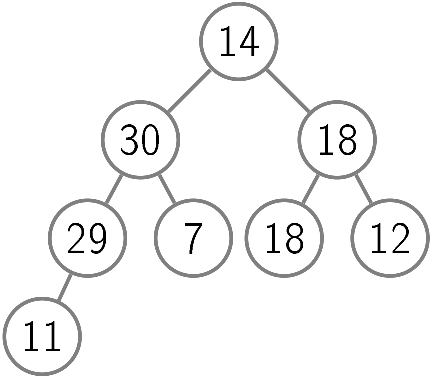
# Keeping the Tree Complete

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



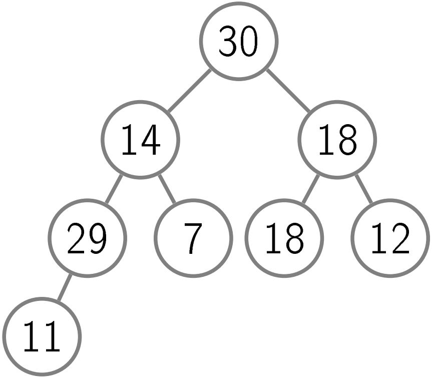
# Keeping the Tree Complete

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



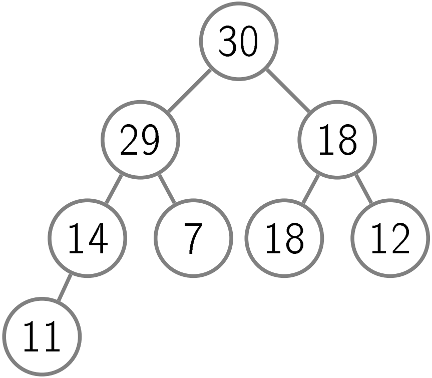
# Keeping the Tree Complete

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

# General Setting

- *maxSize* is the maximum number of elements in the heap

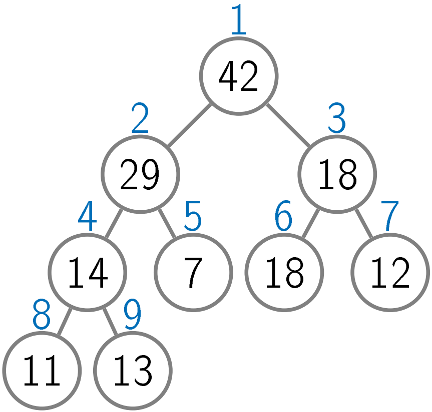
# General Setting

- *maxSize* is the maximum number of elements in the heap
- *size* is the size of the heap

# General Setting

- *maxSize* is the maximum number of elements in the heap
- *size* is the size of the heap
- $H[1 \dots maxSize]$ is an array of length *maxSize* where the heap occupies the first *size* elements

# Example




$size = 9$

$maxSize = 13$

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $H$ | 42 | 29 | 18 | 14 | 7 | 18 | 12 | 11 | 5 | 30 | 29 | 2 | 8 |

**Parent($i$)**

return $\lfloor \frac{i}{2} \rfloor$

**LeftChild($i$)**

return $2i$

**RightChild($i$)**

return $2i + 1$

## SiftUp($i$)

```
while i > 1 and H[Parent(i)] < H[i]:
    swap H[Parent(i)] and H[i]
    i ← Parent(i)
```

## SiftDown($i$)

```
maxIndex ← i
ℓ ← LeftChild(i)
if ℓ ≤ size and H[ℓ] > H[maxIndex]:
    maxIndex ← ℓ
r ← RightChild(i)
if r ≤ size and H[r] > H[maxIndex]:
    maxIndex ← r
if i ≠ maxIndex:
    swap H[i] and H[maxIndex]
    SiftDown(maxIndex)
```

## Insert($p$)

```
if size = maxSize:
  return ERROR
size ← size + 1
H[size] ← p
SiftUp(size)
```

## ExtractMax()

$result \leftarrow H[1]$

$H[1] \leftarrow H[size]$

$size \leftarrow size - 1$

```
SiftDown(1)
```

return $result$

## `Remove`$(i)$

$H[i] \leftarrow \infty$
`SiftUp`$(i)$
`ExtractMax`$()$

## ChangePriority$(i, p)$

```
oldp ← H[i]
H[i] ← p
if p > oldp:
  SiftUp(i)
else:
  SiftDown(i)
```

# Summary

The resulting implementation is

- fast: all operations work in time $O(\log n)$ (`GetMax` even works in $O(1)$)

# Summary

The resulting implementation is

- fast: all operations work in time $O(\log n)$ (`GetMax` even works in $O(1)$)
- space efficient: we store an array of priorities; parent-child connections are not stored, but are computed on the fly

# Summary

The resulting implementation is

- fast: all operations work in time $O(\log n)$ (`GetMax` even works in $O(1)$)
- space efficient: we store an array of priorities; parent-child connections are not stored, but are computed on the fly
- easy to implement: all operations are implemented in just a few lines of code

# Outline

# Sort Using Priority Queues

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

```
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!).

- The resulting algorithms is comparison-based and has running time $O(n \log n)$ (hence, asymptotically optimal!).
- Natural generalization of selection sort: instead of simply scanning the rest of the array to find the maximum value, use a smart data structure.

- The resulting algorithms is comparison-based and has running time $O(n \log n)$ (hence, asymptotically optimal!).
- Natural generalization of selection sort: instead of simply scanning the rest of the array to find the maximum value, use a smart data structure.
- Not in-place: uses additional space to store the priority queue.

## 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($A[1 \ldots n]$)**

$size \leftarrow n$
for $i$ from $\lfloor n/2 \rfloor$ downto 1:
  SiftDown($i$)

- 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($A[1 \dots n]$)**

```
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.

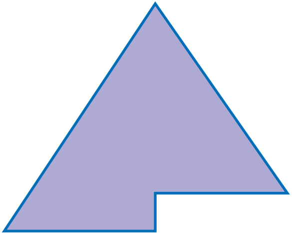
# Building Running Time

- 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

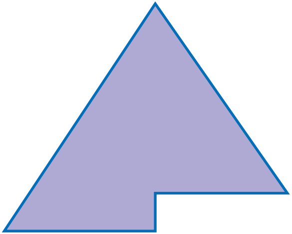
- The running time of `BuildHeap` is $O(n \log n)$ since we call `SiftDown` for $O(n)$ nodes.
- If a node is already close to the leaves, then sifting it down is fast.
- We have many such nodes!
- Was our estimate of the running time of `BuildHeap` too pessimistic?

# Building Running Time



| # nodes | $T(\texttt{SiftDown})$ |
|---|---|
| 1 | $\log_2 n$ |
| 2 | |
| $\vdots$ | $\vdots$ |
| $\leq n/4$ | 2 |
| $\leq n/2$ | 1 |

# Building Running Time

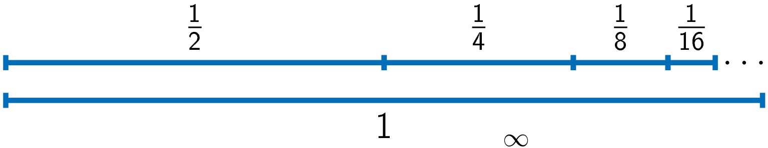


| # nodes | $T(\texttt{SiftDown})$ |
| --- | --- |
| 1 | $\log_2 n$ |
| 2 | |
| $\vdots$ | $\vdots$ |
| $\leq n/4$ | 2 |
| $\leq n/2$ | 1 |

$$T(\texttt{BuildHeap}) \leq \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \dots$$

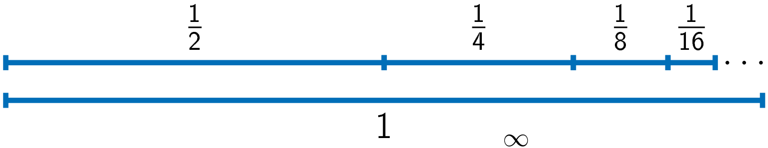
$$\leq n \cdot \sum_{i=1}^{\infty} \frac{i}{2^i} = 2n$$

# Estimating the Sum

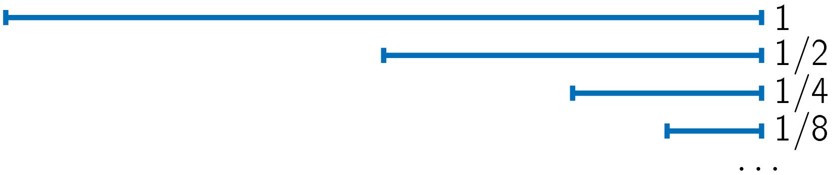



$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots = \sum_{k=1}^{\infty} \frac{1}{2^k} = 1$$

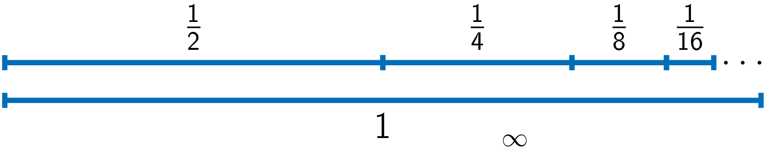
# Estimating the Sum



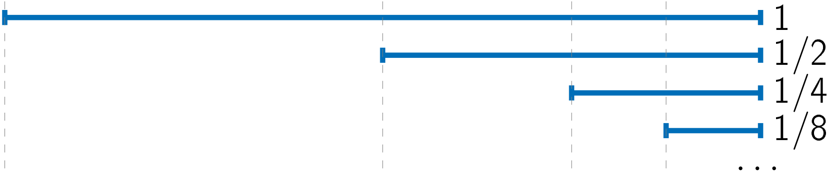

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots = \sum_{k=1}^{\infty} \frac{1}{2^k} = 1$$



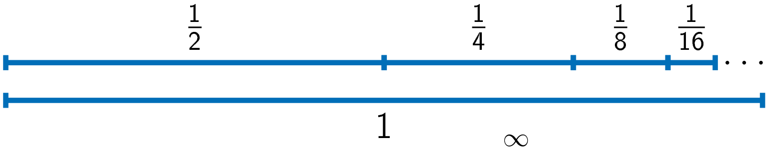
# Estimating the Sum



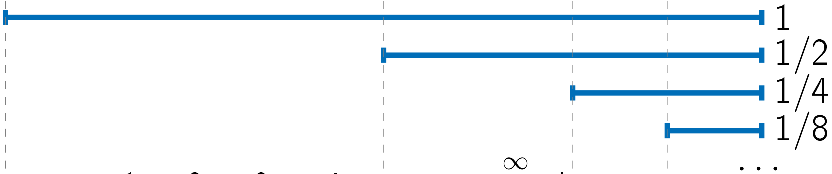

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots = \sum_{k=1}^{\infty} \frac{1}{2^k} = 1$$



# Estimating the Sum




$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots = \sum_{k=1}^{\infty} \frac{1}{2^k} = 1$$




$$\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \dots = \sum_{k=1}^{\infty} \frac{k}{2^k} = 2$$

## Partial sorting

Input: An array $A[1 \ldots n]$, an integer $1 \leq k \leq n$.

Output: The last $k$ elements of a sorted version of $A$.

## Partial sorting

**Input:** An array $A[1 \dots n]$, an integer $1 \leq k \leq n$.

**Output:** The last $k$ elements of a sorted version of $A$.

Can be solved in $O(n)$ if $k = O(\frac{n}{\log n})$!

**PartialSorting($A[1 \dots n], k$)**

```
BuildHeap(A)
for i from 1 to k:
  ExtractMax()
```

**PartialSorting($A[1 \ldots n], k$)**

```
BuildHeap(A)
for i from 1 to k:
  ExtractMax()
```

Running time: $O(n + k \log n)$

# Summary

Heap sort is a time and space efficient comparison-based algorithm: has running time $O(n \log n)$, uses no additional space.

# Outline

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

# 0-based Arrays

**Parent($i$)**

return $\lfloor \frac{i-1}{2} \rfloor$

**LeftChild($i$)**

return $2i + 1$

**RightChild($i$)**

return $2i + 2$

# Binary Min-Heap

## Definition

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

Can be implemented similarly.

# $d$-ary Heap

- In a $d$-ary heap nodes on all levels except for possibly the last one have exactly $d$ children.

# $d$-ary Heap

- In a $d$-ary heap nodes on all levels except for possibly the last one have exactly $d$ children.
- The height of such a tree is about $\log_d n$.

# $d$-ary Heap

- In a $d$-ary heap nodes on all levels except for possibly the last one have exactly $d$ children.
- The height of such a tree is about $\log_d n$.
- The running time of `SiftUp` is $O(\log_d n)$.

# $d$-ary Heap

- In a $d$-ary heap nodes on all levels except for possibly the last one have exactly $d$ children.
- The height of such a tree is about $\log_d n$.
- The running time of `SiftUp` is $O(\log_d n)$.
- The running time of `SiftDown` is $O(d \log_d n)$: on each level, we find the largest value among $d$ children.

# Summary

- Priority queue supports two main operations: `Insert` and `ExtractMax`.

# Summary

- Priority queue supports two main operations: `Insert` and `ExtractMax`.
- In an array/list implementation one operation is very fast ($O(1)$) but the other one is very slow ($O(n)$).

# Summary

- Priority queue supports two main operations: `Insert` and `ExtractMax`.
- In an array/list implementation one operation is very fast ($O(1)$) but the other one is very slow ($O(n)$).
- Binary heap gives an implementation where both operations take $O(\log n)$ time.

# Summary

- Priority queue supports two main operations: `Insert` and `ExtractMax`.
- In an array/list implementation one operation is very fast ($O(1)$) but the other one is very slow ($O(n)$).
- Binary heap gives an implementation where both operations take $O(\log n)$ time.
- Can be made also space efficient.