# PRIORITY QUEUES / HEAPS

Markus Jäger (Computer Science)
Florian Beck (Artificial Intelligence)
Bernhard Anzengruber (Artificial Intelligence)

Institute of Pervasive Computing
Johannes Kepler University Linz

markus.jaeger@jku.at
florian.beck@jku.at
bernhard.anzengruber-tanase@jku.at

Algorithms and Data Structures 1
Exercise – 2023S

**JOHANNES KEPLER UNIVERSITY LINZ**
Altenberger Straße 69
4040 Linz, Austria
jku.at

# Abstract Data Type :: DEFINITION

**Abstract Data Type (ADT)**

- data type that can **only** be accessed via an interface
- set of values and a collection of operation on these values
- describes **which data** can be managed
- describes **which operations** can be performed on it

The ***interface*** **defines** data and operations

The ***implementation*** **realizes** the actual operations

The implementation is **completely separated** by the interface

- access to data elements is possible **only** via operations provided by the interface
- reusability
- exchangeability

# ADT :: ADVANTAGES

Enables programming at different **levels of abstraction**

Abstract from implementation details

- e.g., stack → `push()` `pop()`
- exact implementation is hidden

Use of **different** implementations of the **same** interface depending on the application area

Errors can be fixed separately on different levels of abstraction

- **interface remains unchanged**

# ADT :: PRIORITY QUEUE

Stores elements sorted according to a (**priority**) key

- `min()` or `max()`
- `insert()`
- `removeMin()` or `removeMax()`

**Applications**

- discrete event simulation
- job scheduler
- base for sorting algorithms

# ADT :: PRIORITY QUEUE

**Prerequisite**

- elements must be comparable

**I)** PQ implementation using a **linked list**

- **O(1)** for `min()` and `removeMin()`, as the head points to the lowest element
- **O(n)** for `insert()`, as the entire sequence may need to be traversed

head → 1 → 3 → 6 → 8 → 13 → 17 → None

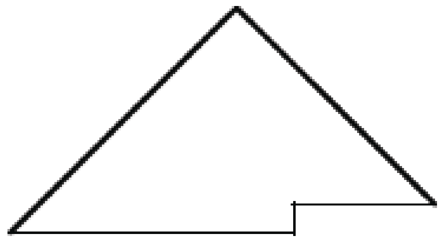
**II)** PQ implementation using a **min heap**

- **O(1)** for `min()`
- **O(log n)** for `insert()` and `removeMin()`

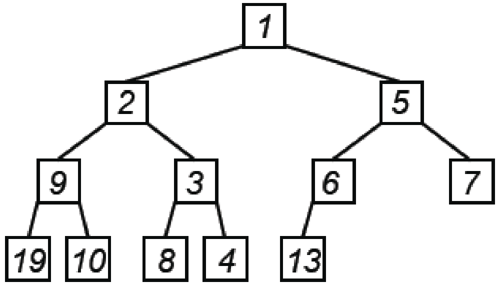
# ADT :: HEAP DEFINITION

Heap properties

- **insertion** and **removal** in **O(log n)**
- **structural property** (a heap is an **almost complete** binary tree)



- **order property** (**MinHeap**)
  - every node's value is ≤ the values of all descendants of this node
  - i.e., key(parent) ≤ key(child)



# ADT :: HEAP INSERT OPERATION

**Insertion in heap**

- create a node at the lowest level of the tree as far to the left as possible (**structural property**)
- let the new value ascend/upheap according to its weight (**order property**)

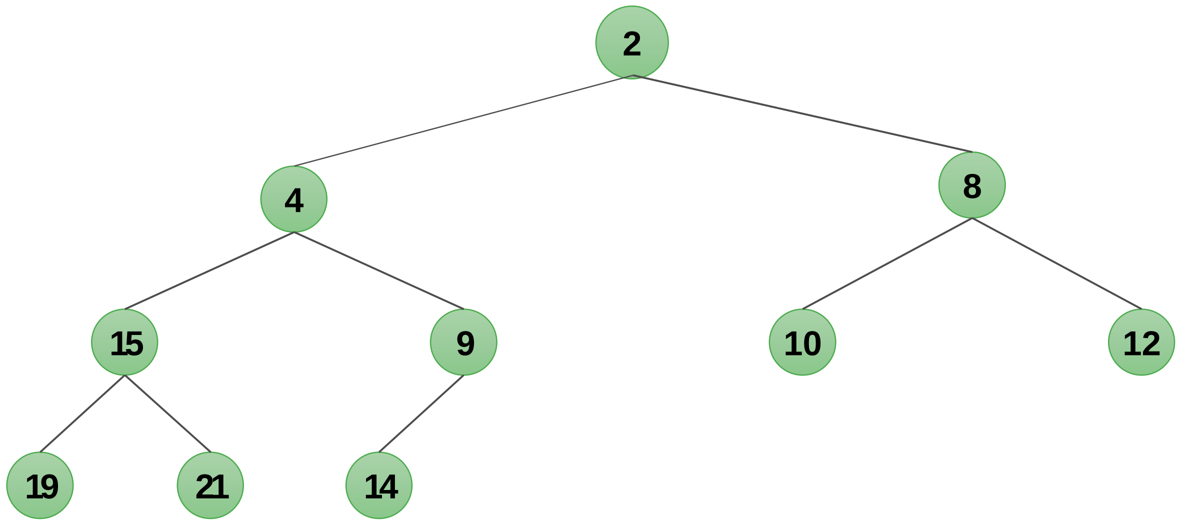
→ **UpHeap** swap values as long as

1. the **child node is smaller** than its parent and
2. **root** is not reached

# ADT :: HEAP INSERT OPERATION

Example **`insert`**`(1)`

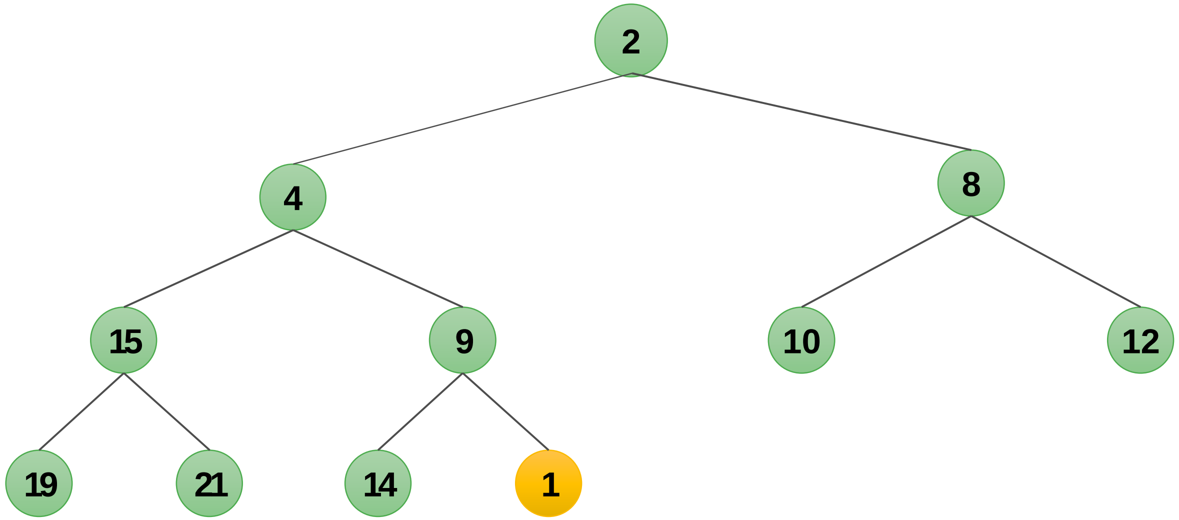
- insert 1 far left on the lowest level



# ADT :: HEAP INSERT OPERATION

Example **`insert`**`(1)`

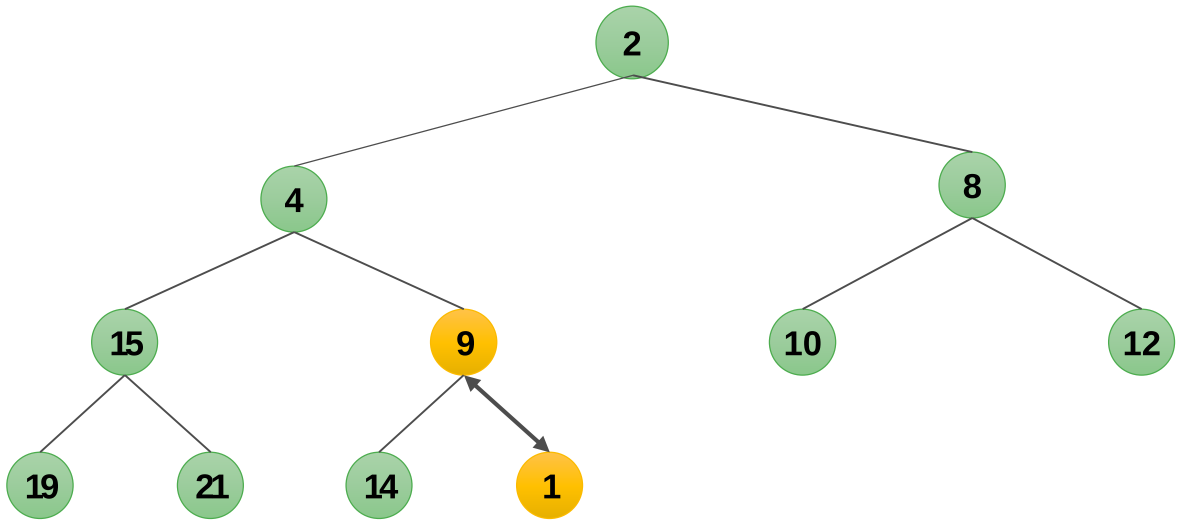
- insert 1 far left on the lowest level



# ADT :: HEAP INSERT OPERATION

Example `insert(1)`

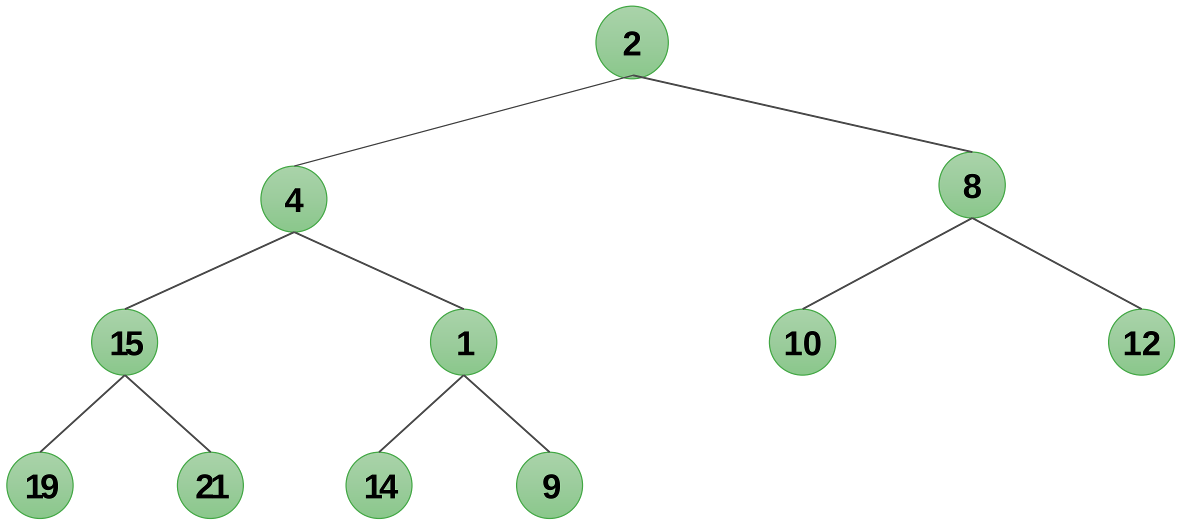
- insert 1 far left on the lowest level
- $1^{st}$ `Upheap()`



# ADT :: HEAP INSERT OPERATION

Example `insert(1)`

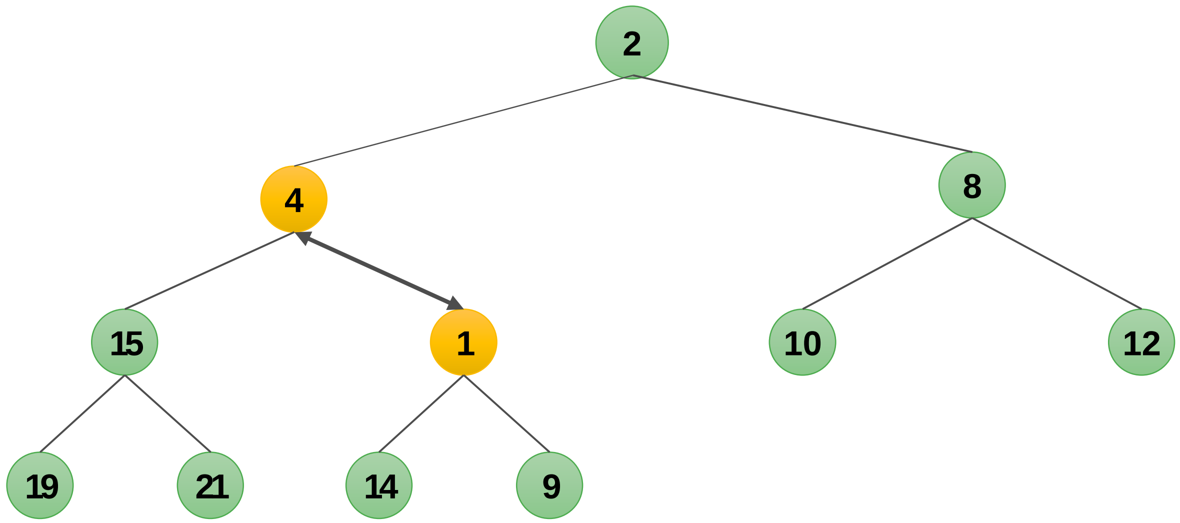
- insert 1 far left on the lowest level
- `1`$^{st}$ `Upheap()`



# ADT :: HEAP INSERT OPERATION

Example `insert(1)`

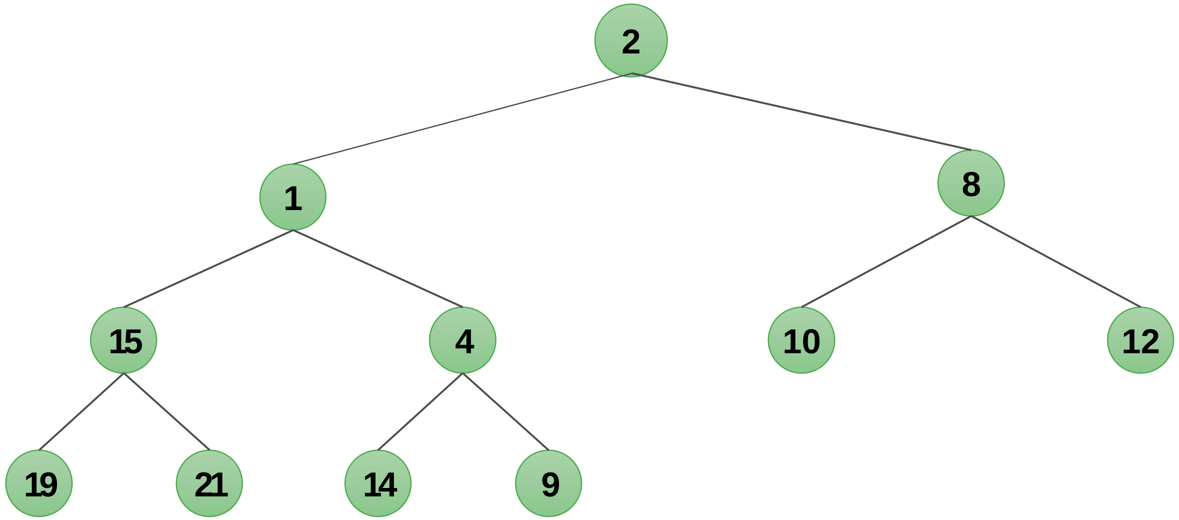
- insert 1 far left on the lowest level
- $1^{st}$ `Upheap()`
- $2^{nd}$ `Upheap()`



# ADT :: HEAP INSERT OPERATION

Example `insert(1)`

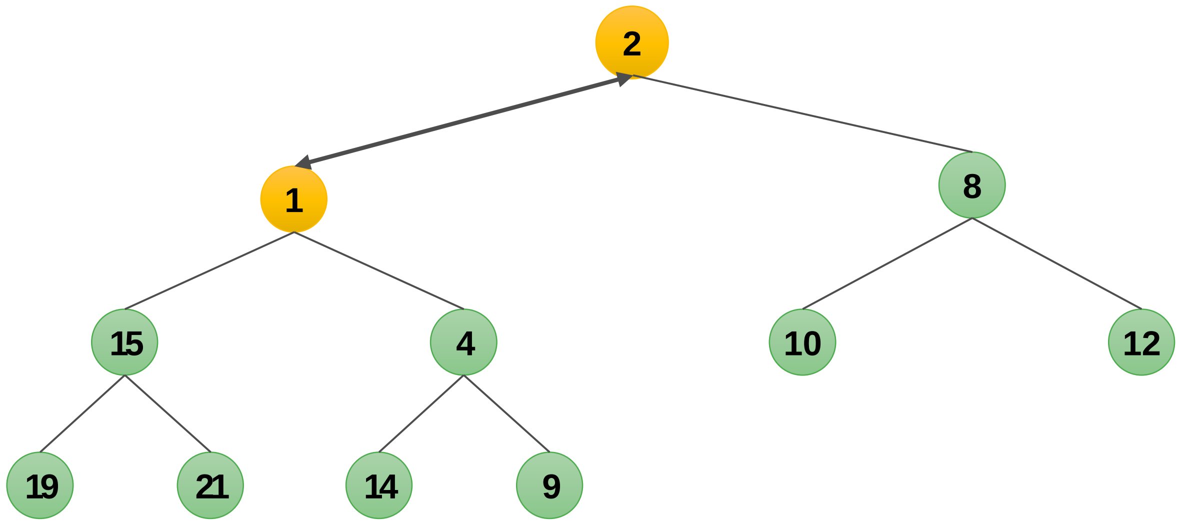
- insert 1 far left on the lowest level
- $1^{st}$ `Upheap()`
- $2^{nd}$ `Upheap()`



# ADT :: HEAP INSERT OPERATION

Example **`insert`**`(1)`

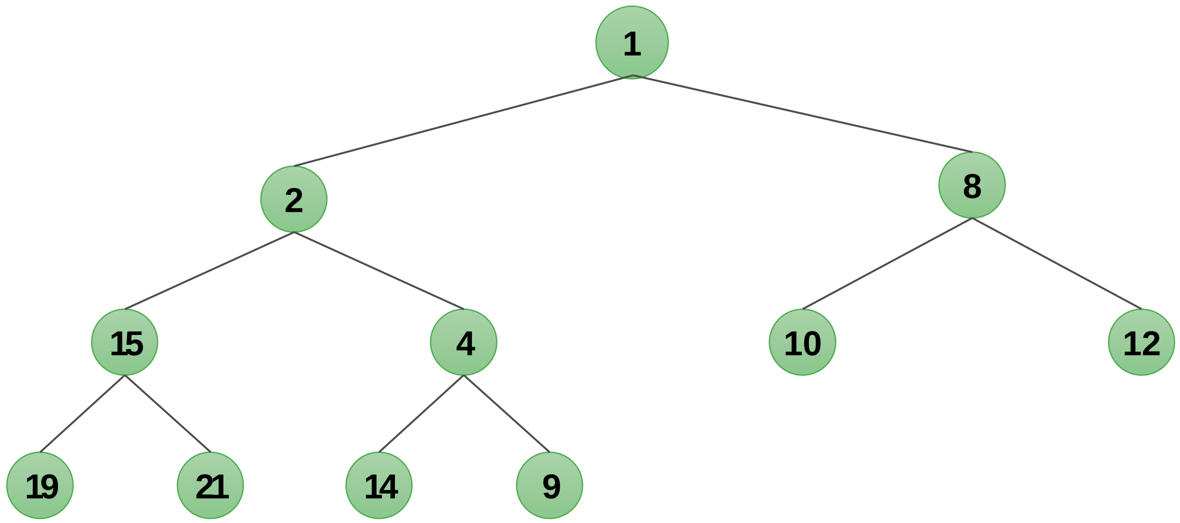
- insert 1 far left on the lowest level
- `1`$^{st}$ `Upheap()`
- `2`$^{nd}$ `Upheap()`
- `3`$^{rd}$ `Upheap()`



# ADT :: HEAP INSERT OPERATION

Example `insert(1)`

- insert 1 far left on the lowest level
- `1st Upheap()`
- `2nd Upheap()`
- `3rd Upheap()`



# ADT :: HEAP REMOVE OPERATION

**Removal** in MinHeap

- smallest element always in the root (**order property**) → **`min()`** returns element in O(1)
- **`removeMin()`** removes the root → new root must be found

**Remove procedure**

- remove the lowest far-right node and make it the new root (**structural property**)
- sink the root value downwards/downheap (**order property**)

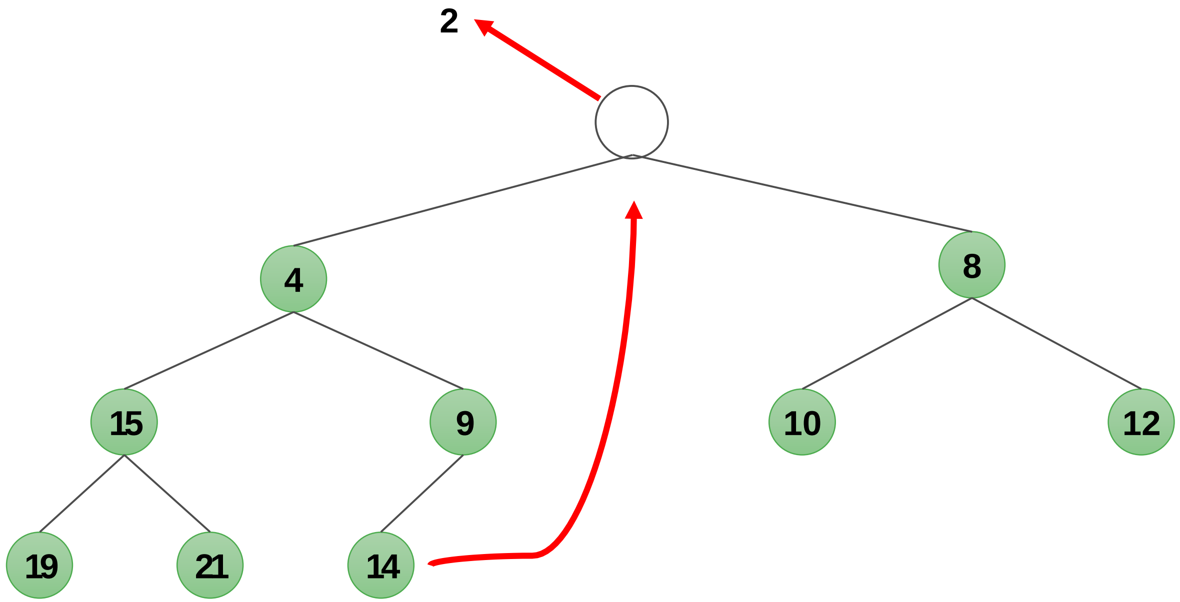
→ **DownHeap** compares a node with its **smallest child** and swaps values as long as

1. the **child node is smaller** and
2. a **leaf** is not reached

# ADT :: HEAP REMOVE OPERATION

Example **`removeMin()`**

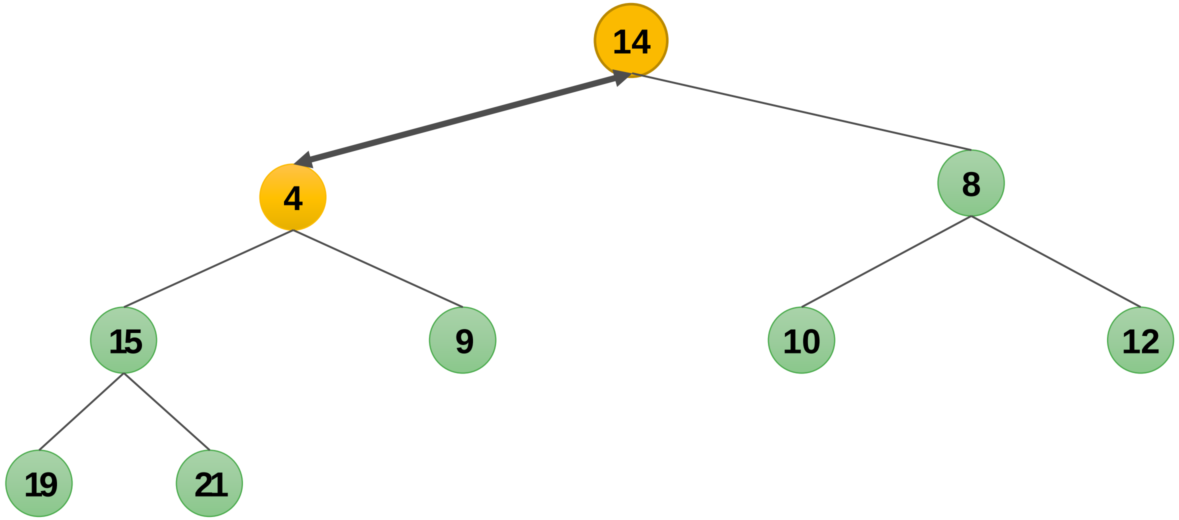
- move lowest far right node to the root



# ADT :: HEAP REMOVE OPERATION

Example `removeMin()`

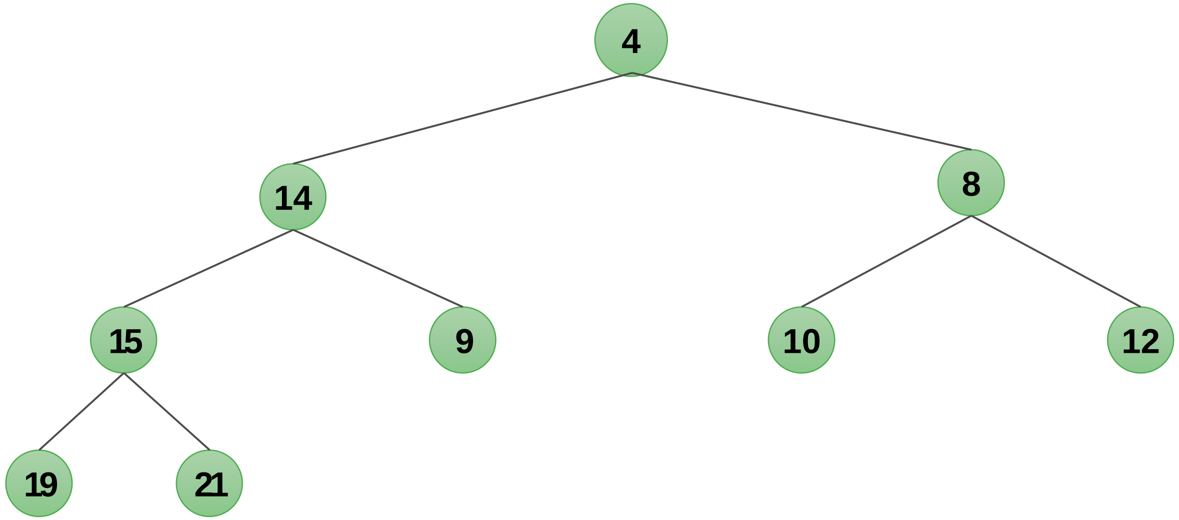
- move lowest far right node to the root
- $1^{st}$ `Downheap()`



# ADT :: HEAP REMOVE OPERATION

Example `removeMin()`

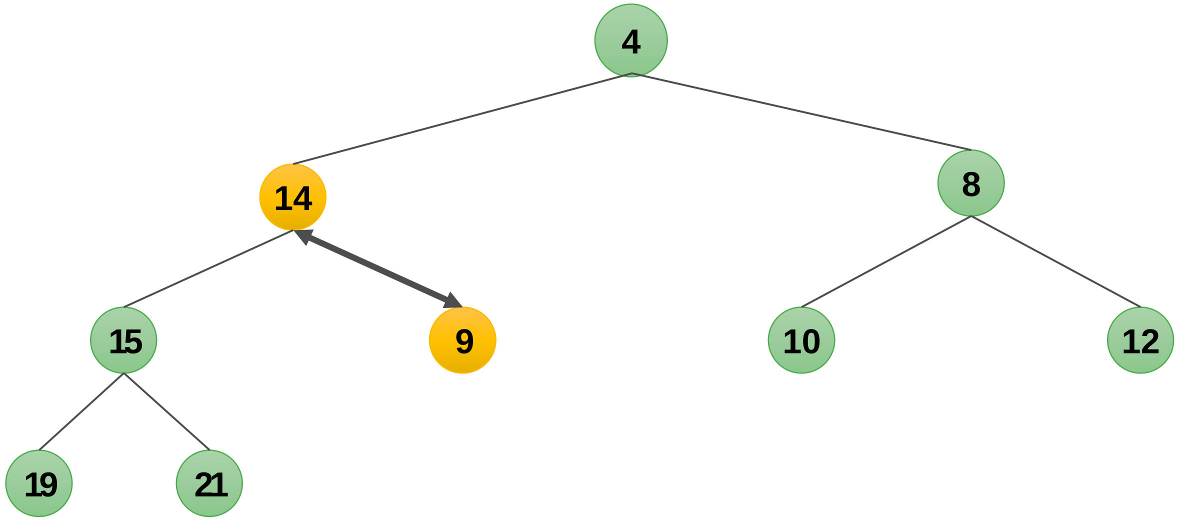
- move lowest far right node to the root
- 1$^{st}$ `Downheap()`



# ADT :: HEAP REMOVE OPERATION

Example **`removeMin()`**

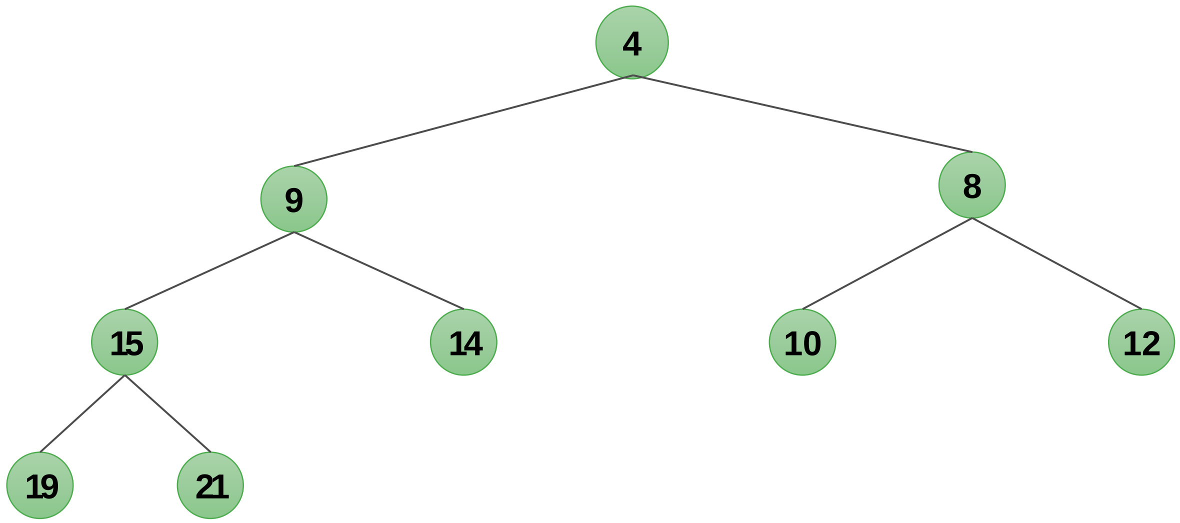
- move lowest far right node to the root
- `1st Downheap()`
- `2nd Downheap()`



# ADT :: HEAP REMOVE OPERATION

Example **`removeMin()`**

- move lowest far right node to the root
- 1st `Downheap()`
- 2nd `Downheap()`



# ADT :: HEAP REALIZATION

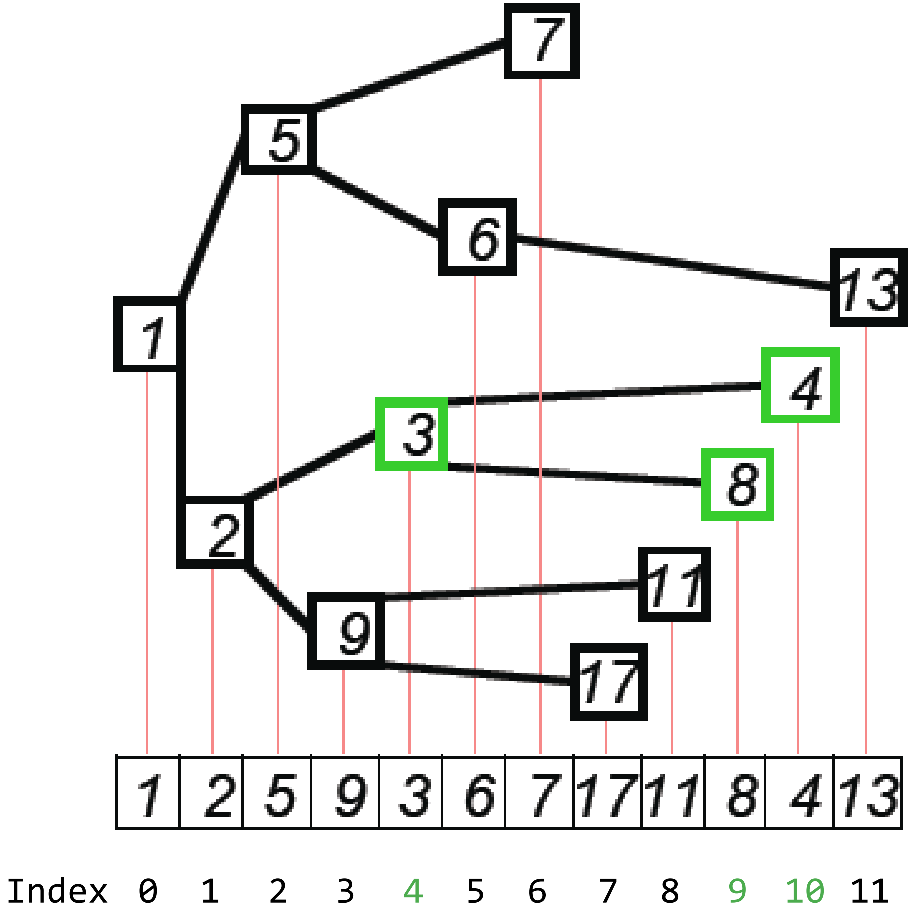
**Array representation**

- heap transformation into 1D array/vector
- sequential top→down, left→right

Element access/**indexing**

- **children** of node with index *i* have the indices 2**i*+1 and 2**i*+2

- **parent** node of a node with index *j* has index *(j-1)/2*



# PRIORITY QUEUES / HEAPS

Institute of
Pervasive Computing

Markus Jäger (Computer Science)
Florian Beck (Artificial Intelligence)
Bernhard Anzengruber (Artificial Intelligence)

Institute of Pervasive Computing
Johannes Kepler University Linz

markus.jaeger@jku.at
florian.beck@jku.at
bernhard.anzengruber-tanase@jku.at

**JOHANNES KEPLER UNIVERSITY LINZ**
Altenberger Straße 69
4040 Linz, Austria
jku.at

Algorithms and Data Structures 1
Exercise – 2023S