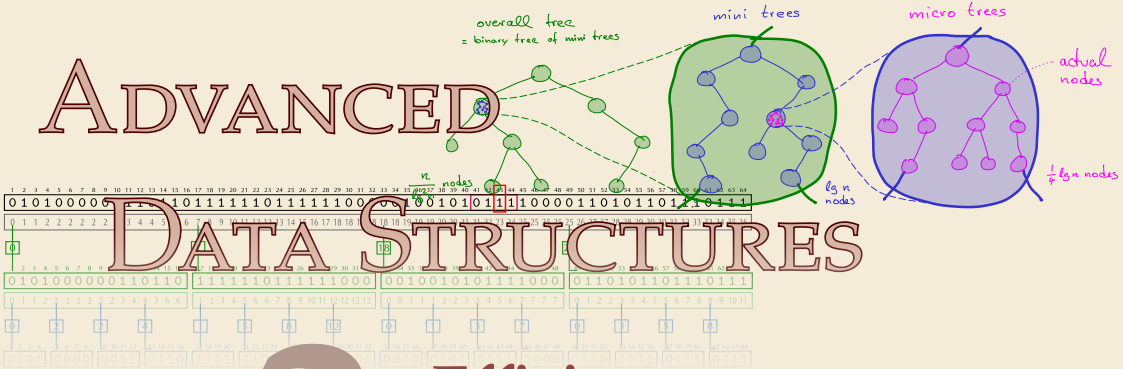



# Advanced Data Structures

# 3 Efficient Priority Queues

*Advanced Data Structures · Summer 2026*

Prof. Dr. Sebastian Wild

CS650 (Summer 2026)
Philipps-Universität Marburg
*version 2026-05-25 15:15*

# Outline

## 3 Efficient Priority Queues


3.1 Tournament Trees
3.2 Lazy Partition Heaps
3.3 Fibonacci Heaps – Informal
3.4 Quake Heaps

# Priority Queue ADT

**(Min-oriented) Priority Queue (MinPQ/PQ):**

- `construct`$(A)$
  Construct from from elements in array $A$.
- `insert`$(x, p)$
  Insert item $x$ with priority $p$ into PQ.
- `min`$()$
  Return item with smallest priority. (Does not modify the PQ.)
- `delMin`$()$
  Remove the item with smallest priority and return it.
- `decreaseKey`$(x, p')$
  Update $x$'s priority to $p' \leq p$.
- `meld`$(Q_1, Q_2)$
  Build a PQ containing the union of $Q_1$ and $Q_2$

Fundamental building block in many applications.

# Elementary Solutions

*Binary heaps* can realize all operations efficiently (except meld).

**Binary heaps**

| Operation | Running Time |
|---|---|
| `construct(`$A[1..n]$`)` | $O(n)$ |
| `insert(`$x$`,`$p$`)` | $O(\log n)$ |
| `delMax()` | $O(\log n)$ |
| `decreaseKey(`$x$`,`$p'$`)` | $O(\log n)$ |
| `max()` | $O(1)$ |
| `isEmpty()` | $O(1)$ |
| `size()` | $O(1)$ |
| `meld(`$Q_1$`,`$Q_2$`)` | $O(n)$ |

# Elementary Solutions

*Binary heaps* can realize all operations efficiently (except meld).

**Binary heaps**

| Operation | Running Time |
|---|---|
| `construct(`$A[1..n]$`)` | $O(n)$ |
| `insert(`$x$`,`$p$`)` | $O(\log n)$ |
| `delMax()` | $O(\log n)$ |
| `decreaseKey(`$x$`,`$p'$`)` | $O(\log n)$ |
| `max()` | $O(1)$ |
| `isEmpty()` | $O(1)$ |
| `size()` | $O(1)$ |
| `meld(`$Q_1, Q_2$`)` | $O(n)$ |

- apart from faster `construct`,
  BSTs always as good as binary heaps

**Balanced binary search tree**

| Operation | Running Time |
|---|---|
| `construct(`$A[1..n]$`)` | $O(n \log n)$ |
| `put(`$k$`,`$v$`)` | $O(\log n)$ |
| `get(`$k$`)`, `contains(`$k$`)` | $O(\log n)$ |
| `delete(`$k$`)` | $O(\log n)$ |
| `isEmpty()` | $O(1)$ |
| `size()` | $O(1)$ |
| `min()`, `max()` | $O(\log n) \rightsquigarrow O(1)$ |
| `floor(`$x$`)`, `ceiling(`$x$`)` | $O(\log n)$ |
| `rank(`$x$`)` | $O(\log n)$ |
| `select(`$i$`)` | $O(\log n)$ |
| `split(`$x$`)` | $O(\log n)$ |
| `join(`$T_1, T_2$`)` (for $T_1 \leq T_2$) | $O(\log n)$ |

# Elementary Solutions

*Binary heaps* can realize all operations efficiently (except meld).

**~~Binary heaps~~** *stay tuned*

| Operation | Running Time |
|---|---|
| `construct(`$A[1..n]$`)` | $O(n)$ |
| `insert(`$x$`,`$p$`)` | ~~$O(\log n)$~~ $O(1)$ |
| `delMax()` | $O(\log n)$ |
| `decreaseKey(`$x$`,`$p'$`)` | ~~$O(\log n)$~~ $O(1)$ |
| `max()` | $O(1)$ |
| `isEmpty()` | $O(1)$ |
| `size()` | $O(1)$ |
| `meld(`$Q_1$`,`$Q_2$`)` | ~~$O(n)$~~ $O(1)$ |

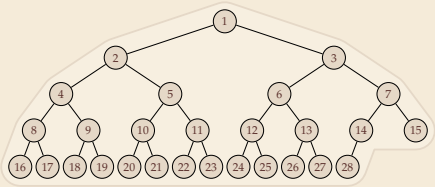
- apart from faster `construct`, BSTs always as good as binary heaps
- PQ abstraction still helpful
- faster heaps exist!

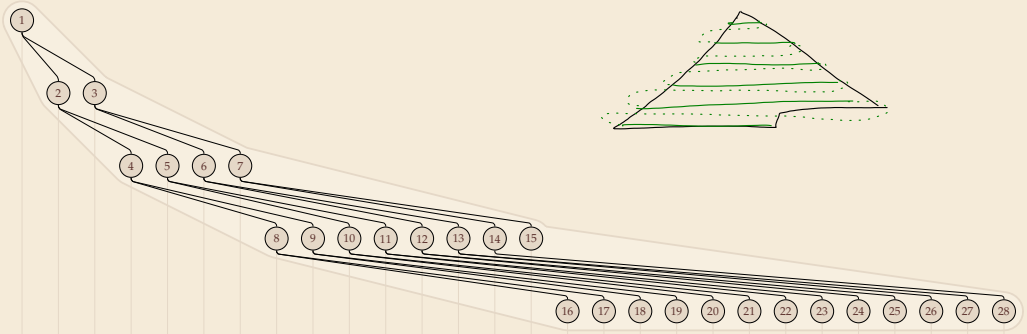
**Balanced binary search tree**

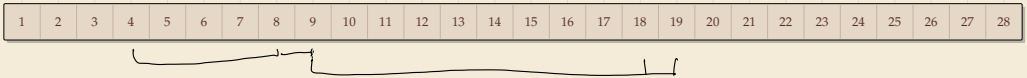
| Operation | Running Time |
|---|---|
| `construct(`$A[1..n]$`)` | $O(n \log n)$ |
| `put(`$k$`,`$v$`)` | $O(\log n)$ |
| `get(`$k$`),contains(`$k$`)` | $O(\log n)$ |
| `delete(`$k$`)` | $O(\log n)$ |
| `isEmpty()` | $O(1)$ |
| `size()` | $O(1)$ |
| `min(),max()` | $O(\log n) \rightsquigarrow O(1)$ |
| `floor(`$x$`),ceiling(`$x$`)` | $O(\log n)$ |
| `rank(`$x$`)` | $O(\log n)$ |
| `select(`$i$`)` | $O(\log n)$ |
| `split(`$x$`)` | $O(\log n)$ |
| `join(`$T_1$`,`$T_2$`)` (for $T_1 \leq T_2$) | $O(\log n)$ |

## 3.1 Tournament Trees

# Implicit Complete Binary Trees



## Del Min

remove root

swap last-root

sink down

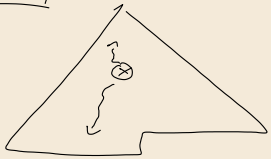
$\leq \log n + 1$

$\#\text{cmps} \leq 2 \log n \pm O(1)$

## Insert

$\leq \log n \pm O(1)$

## Change Key



$O(\log n)$