

# Algorithmen und Datenstrukturen

## B5. Heaps und Heapsort

Marcel Lüthi and Gabriele Röger

Universität Basel

28. März 2019

# Algorithmen und Datenstrukturen

28. März 2019 — B5. Heaps und Heapsort

## B5.1 Einführung

## B5.2 Heaps

## B5.3 Warteschlangen mit Heaps

## B5.4 Heapsort

## B5.1 Einführung

## Ausblick auf Vorlesung

- ▶ Die Datenstruktur Heap
- ▶ Heaps zur Implementation von Priorityqueues
- ▶ Heapsort

## Informatiker des Tages



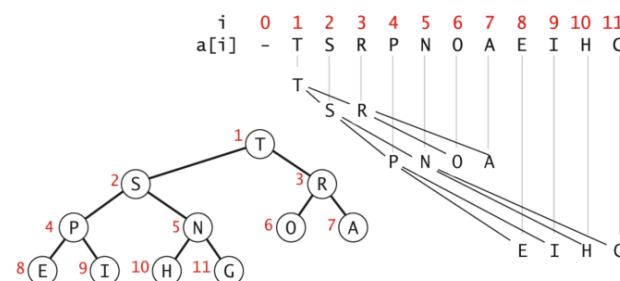
Robert W. Floyd

- ▶ Gewinner Turing Award (1978)
  - ▶ U.a. für Arbeit an Analyse von Algorithmen
- ▶ Entwickler des Treesort Algorithmus (Vorgänger von Heapsort)
  - ▶ Verbesserung von Heapsort, nachdem dieser von J. Williams entwickelt wurde.
- ▶ Auch bekannt für: Floyd-Warshall Algorithmus
  - ▶ Findung von kürzesten Pfaden in Graphen.

Floyd, R. W. (1979). "The paradigms of programming".  
Communications of the ACM. 22 (8).

## Bijektion - Array / Vollständiger Binärbaum

- ▶ Jedes Array kann als vollständiger Binärbaum interpretiert werden:
  - ▶ Linker Teilbaum: Index Wurzel \* 2
  - ▶ Rechter Teilbaum: Index Wurzel \* 2 + 1



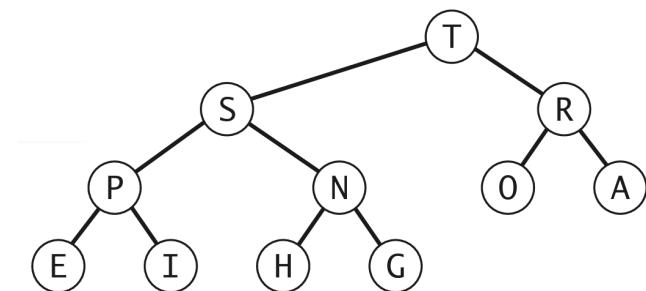
Quelle: Abbildung 2.26, Algorithms, Sedgewick & Wayne

## B5.2 Heaps

## Heap

### Definition: Heap

Ein binärer Baum / Array ist Heap geordnet, wenn der Schlüssel in jedem Knoten grösser gleich dem Schlüssel seiner beiden Kindern (sofern vorhanden) ist.



Quelle: Abbildung 2.25, Algorithms, Wayne & Sedgewick

## Heap Ordnung

### Theorem

Der grösste Schlüssel in einem Heap-geordneten Binärbaum befindet sich an der Wurzel.

### Beweis.

Induktion über die Baumgrösse

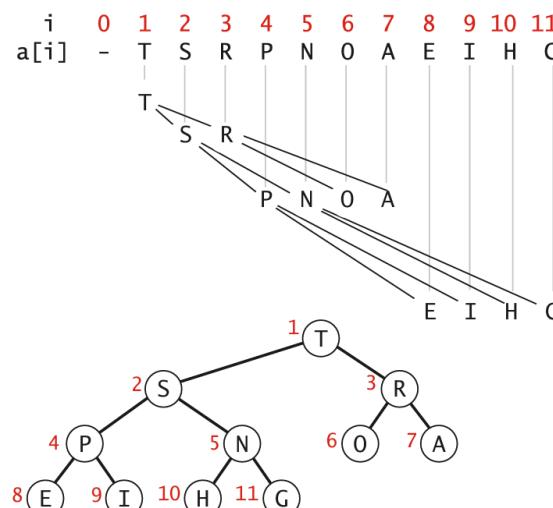


## Binärer Heap

### Definition: Binärer Heap

Ein binärer Heap ist eine Sammlung von Schlüsseln, die in einem vollständigen Heap-geordneten Binärbaum angeordnet sind und in einem Array ebenenweise repräsentiert werden (das erste Feld des Arrays wird nicht verwendet).

## Binärer Heap



Quelle: Abbildung 2.26, Algorithmen, Wayne & Sedgewick

## B5.3 Warteschlangen mit Heaps

## Priority Queue ADT

```
class MaxPQ[Item]:
    # Element einfuegen
    def insert(k : Item) -> None

    # Grossestes Element zurueckgeben
    def max() -> Item

    # Grossestes Element entfernen und zurueckgeben
    def delMax() -> Item

    # Ist die Queue leer?
    def isEmpty() -> bool

    # Anzahl Elemente in der Priority Queue
    def size() -> int
```

## Beobachtung (2)

- ▶ Array implementation erlaubt uns in konstanter Zeit zu jedem Kind den Elternknoten und von jedem Elternknoten alle Kinder finden ...  
... ohne dabei explizite Verweise verwalten zu müssen .
- ▶ Der Baum hat die Höhe  $\lfloor \log_2(N) \rfloor$

### Plan

Durch geschicktes Vertauschen der Eltern/Kinder in  $O(\log_2(N))$  Operationen nach Entfernen oder Einfügen eines Elements die Heapbedingung wiederherstellen.

## Beobachtung

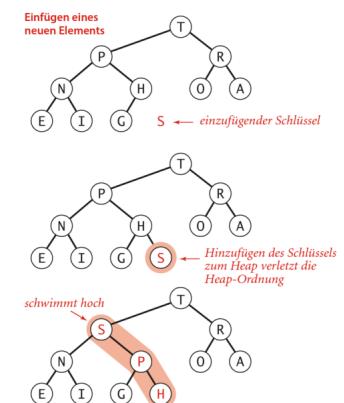
Array implementation von Max-heap hat grösstes Element immer an Stelle 1 .

- ▶ Implementation von `max` ist trivial

Problem: Wir müssen wenn wir beim `insert` und `delMax` die Heapbedingung erfüllen können.

## Element einfuegen

- ▶ Blatt wird an letzter Stelle im Array eingefügt
  - ▶ entspricht Blatt ganz rechts
- ▶ Heap Bedingung wird durch ausführen von `swim` wiederhergestellt

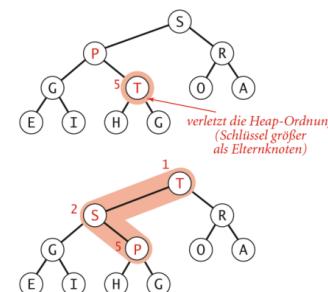


Quelle: Abbildung 2.29: Algorithmen, Sedgewick & Wayne

## Die Operation swim

- ▶ Knoten an Position  $k$  in Array  $a$  schwimmt nach oben bis Heap Bedingung wieder erfüllt ist.
- ▶ Braucht maximal  $\log_2(N) + 1$  Vergleiche.

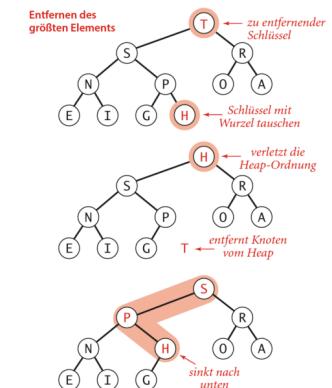
```
def swim(a, k):
    while k > 1 and a[k/2] < a[k]:
        a[k/2], a[k] = a[k], a[k/2]
        k = k/2
```



Quelle: Abbildung 2.29: Algorithmen, Sedgewick & Wayne

## Grösstes Element entfernen

- ▶ Wurzel (grösstes Element) wird entfernt
- ▶ Blatt ganz rechts wird an Wurzel gesetzt
- ▶ Heap Bedingung wird durch ausführen von sink wiederhergestellt

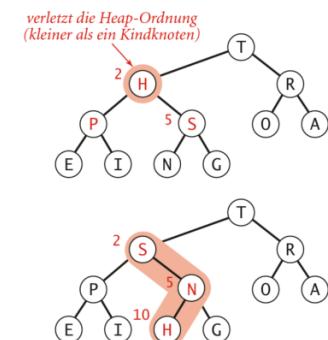


Quelle: Abbildung 2.29: Algorithmen, Sedgewick & Wayne

## Die Operation sink

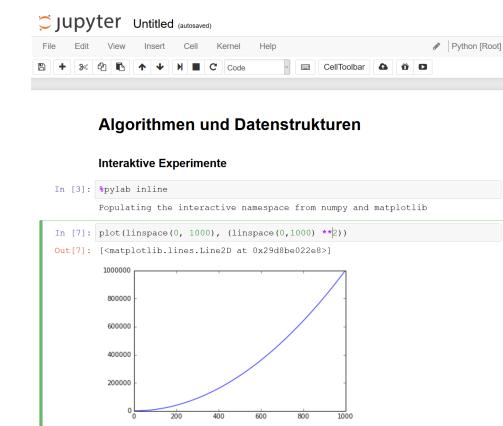
- ▶ Knoten an Position  $k$  in Array  $a$  sinkt nach unten bis Heap Bedingung wieder erfüllt ist.
- ▶ Element wird mit grösserem Kind vertauscht.
- ▶ Braucht maximal  $2 \log_2(N)$  Vergleiche.

```
def sink(a, k):
    while 2 * k <= len(a):
        j = 2 * k
        if j < len(a) and a[j] < a[j+1]:
            j += 1
        if not a[k] < a[j]:
            break
        a[j], a[k] = a[k], a[j]
        k = j
```



Quelle: Abbildung 2.29: Algorithmen, Sedgewick & Wayne

## Implementation



Jupyter Notebooks: Heap.ipynb

## Komplexität

### Theorem

In einer Vorrangwarteschlange mit  $N$  Elementen benötigen die Heap-Algorithmen zum Einfügen eines neuen Elements nicht mehr als  $1 + \log_2(N)$  Vergleiche und zum Entfernen des grössten Elements nicht mehr als  $2 \log_2(N)$  Vergleiche.

## B5.4 Heapsort

## Ein Sortieralgorithmus

- ▶ Gegeben, ein unsortiertes Array der Länge  $N$ .
- ▶ Füge alle Elemente der Reihe nach in einen Heap ein.
- ▶ Entferne  $N$  mal das grösste Element und schreibe es zurück ins Array.

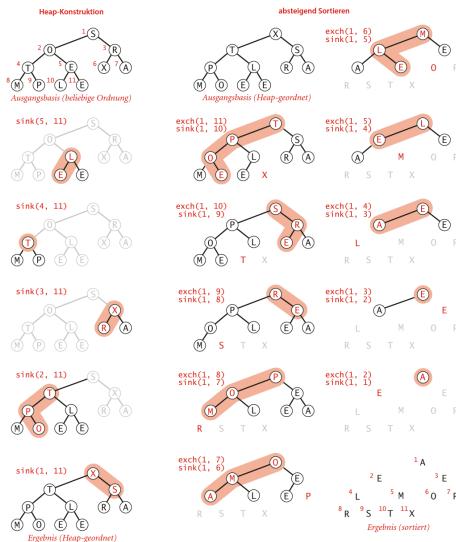
### Komplexität

Die Prozedur hat garantierte Laufzeitkomplexität von  $O(n \log_2(n))$ .

- ▶ Idee: Geschicktes verwenden von swim und sink lässt uns heapsort in-place verwenden.
- ▶ Prozedur verläuft in zwei Phasen:
  - ① Heap Konstruktion (rechts nach Links)
  - ② Absteigendes Sortieren durch sukzessives Tauschen von grösstem Element

```
def heapsort(a):
    N = len(a) - 1
    for k in range(int(N/2), 0, -1):
        sink(a, k)
    while N > 1:
        a[1], a[N] = a[N], a[1]
        N -= 1
        sink(a, 1, N)
```

## Heapsort

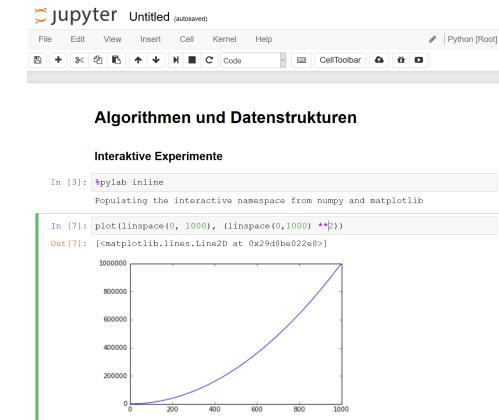


Quelle: Abbildung 2.31, Algorithmen, Wayne & Sedgewick

## Bemerkungen

- ▶ Heapsort ist theoretisch wichtig:
  - ▶ Optimal hinsichtlich Zeit und Speichernutzung
  - ▶ Laufzeit  $O(n \log n)$ .
  - ▶ Zusätzlicher Speicher ( $O(1)$ )
- ▶ Praktische Bedeutung eher klein
  - ▶ Nutzt CPU Cache nicht effizient, da entfernte Elemente ausgetauscht werden.
- ▶ Heaps sind aber für Priority Queues sehr wichtig!

## Implementation



Jupyter Notebooks: Heap.ipynb

## Zusammenfassung

- ▶ Heap-sort Algorithmus von Datenstruktur "getrieben"
- ▶ Nutzt nicht triviale Zwischenschritte und Hilfsstrukturen
  - ▶ Nutzung von Eigenschaften vollständiger binärer Bäume
  - ▶ Effiziente Implementation mittels Arrays
  - ▶ Heap Bedingung um größtes Element zu erhalten
- ▶ Verständnis von Heap ist zentral für Algorithmus
  - ▶ Danach ist Algorithmus einfach zu verstehen
  - ▶ Laufzeitanalyse trivial

Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won't usually need your code ; it'll be obvious

Fred Brooks