# Datenstrukturen und Algorithmen

## Vorlesung 8: Heapsort (K6)

Joost-Pieter Katoen

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

http://www-i2.rwth-aachen.de/i2/dsal10/

11. Mai 2010





## Übersicht







## Übersicht







## Heaps

**Heap (Haufen)**

Ein Heap ist ein Binärbaum, der Elemente mit Schlüsseln enthält und in ein Array eingebettet ist. Die Heap-Bedingung für *Max-Heap*s fordert:

- Der Schlüssel eines Knotens ist stets größer als (bzw. mindestens so groß wie) die Schlüssel seiner Kinder.

Weiter gilt:

- Alle Ebenen, abgesehen von evtl. der untersten, sind komplett gefüllt.
- Die Blätter befinden sich damit alle auf einer, höchstens zwei, Ebenen.
- Die Blätter der untersten Ebene sind linksbündig angeordnet.

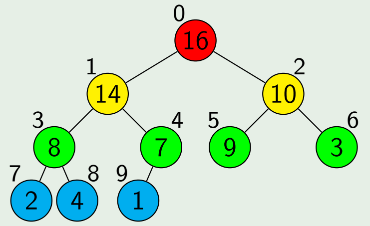
# Arrayeinbettung eines Heaps

## Arrayeinbettung

Das Array a wird wie folgt als Binärbaum aufgefasst:

- Die Wurzel liegt in `a[0]`.
- Das linke Kind von `a[i]` liegt in `a[2 * i + 1]`.
- Das rechte Kind von `a[i]` liegt in `a[2 * i + 2]`.

## Beispiel




| 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

- Durch die möglichst vollständige Füllung der Ebenen werden „Löcher“ im Array vermieden.
- Vergrößert man den Baum um ein Element, so wird das Array gerade um ein Element länger.





# Heaps – Eigenschaften

## Lemma

*Vergrößert man den Schlüssel der Wurzel, dann bleibt der Baum ein Heap.*

## Lemma

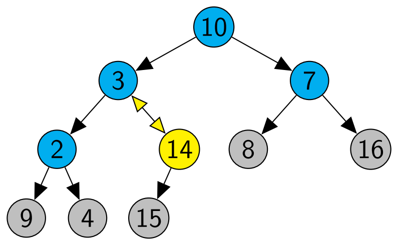
*Jedes Array ist ein Heap ab Position* $\lfloor \frac{n}{2} \rfloor$.

- Ein Heap hat $\lfloor \frac{n}{2} \rfloor$ innere Knoten.





# Naiver Heapaufbau




| 10 | 3 | 7 | 2 | 14 | 8 | 16 | 9 | 4 | 15 |
|---|---|---|---|---|---|---|---|---|---|

## Heapaufbau, naiv

Der Heap wird von oben nach unten (top-down) aufgebaut, indem

- ein neues Element möglichst weit links angefügt wird und
- rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist.





# Naiver Heapaufbau – Algorithmus und Analyse

```
void bubble(int E[], int pos) {
  while (pos > 0) {
    int parent = (pos - 1) / 2;
    if (E[parent] > E[pos]) {
      break;
    }
    swap(E[parent], E[pos]);
    pos = parent;
  }
}
```

Die Höhe $k$ eines Heaps mit $n$ Elementen ist beschränkt durch:

$$n \leqslant 2^{k+1} - 1 \quad \Rightarrow \quad k = \lfloor \log n \rfloor$$

- Damit kostet jedes Einfügen $k \approx \log n$ Vergleiche.

$\Rightarrow$ Zum Aufbau eines Heaps mit $n$ Elementen benötigt man $\Theta(n \cdot \log n)$ Vergleiche.

Es geht effizienter: `sink` (auch: heapify, fixheap) [Floyd 1964]

## Heapify – Strategie

Betrachte `E[i]` unter der Annahme, der rechte und linke Teilbaum ist bereits ein Heap.

- `E[i]` kann kleiner als seine Kinder sein.
- Wir wollen die beiden Teilbäume – Heaps – zusammen mit `E[i]` zu einem (Gesamt-)Heap verschmelzen.
- Dazu lassen wir `E[i]` in den Heap hineinsinken, so dass der Teilbaum mit Wurzel `E[i]` ein Heap ist.

**Heapify**

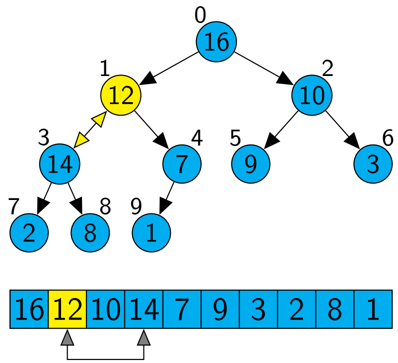
- Finde das Maximum der Werte `E[i]` und seiner Kinder.
- Ist `E[i]` bereits das größte Element, dann ist dieser gesamte Teilbaum auch ein Heap. Fertig.
- Andernfalls tausche `E[i]` mit dem größten Element und führe Heapify in diesem Unterbaum weiter aus.





## Heapify – Algorithmus und Beispiel

```
void sink(int E[], int n, int pos) {
  int next = 2 * pos + 1;
  while (next < n) {
    if (next + 1 < n &&
        E[next + 1] > E[next]) {
      next = next + 1;
    }
    if (E[pos] > E[next]) {
      break;
    }
    swap(E[pos], E[next]);
    pos = next;
    next = 2 * pos + 1;
  }
}
```

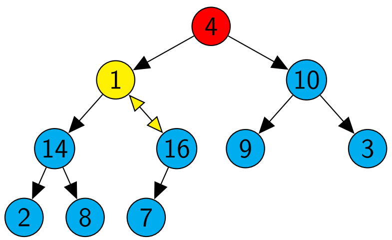







## Heapaufbau – Algorithmus und Beispiel

Strategie: Wandle das Array von unten nach oben (bottom-up) in einen Heap um.




```
void buildHeap(int E[]) {
  for (int i = E.length / 2 - 1; i >= 0; i--) {
    sink(E, E.length, i);
  }
}
```

Nach jedem Aufruf von `sink(E, E.length, i)` sind die Knoten `i, ..., E.length - 1` schon Wurzeln von Heaps.





## Konstruktion eines Heaps

**Lemma**

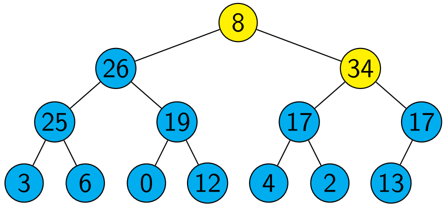
*Der Algorithmus* `buildHeap` *ist korrekt und terminiert.*

- Initialisierung: Jeder Knoten $i = \lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1$ ist ein Blatt und damit Wurzel eines trivialen Heaps.
- Schleifeninvariante: Zu Beginn der `for`-Schleife ist jeder Knoten $i{+}1, \ldots, E.length$ die Wurzel eines Heaps.
- In jeder Iteration sind alle Kinder des Knotens $i$ bereits Wurzeln eines Heaps (Schleifeninvariante).

$\Rightarrow$ Bedingung für den Aufruf von `sink` ist erfüllt.

- Dekrementierung von $i$ stellt Schleifeninvariante wieder her.
- Terminierung: Bei $i = 0$ ist gemäß Schleifeninvariante jeder Knoten $1, 2, \ldots, n$ die Wurzel eines Heaps.

# Heapsort – Algorithmus und Beispiel




| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 26 | 34 | 25 | 19 | 17 | 17 | 3 | 6 | 0 | 12 | 4 | 2 | 13 | 41 | 69 |

```java
void heapSort(int E[]) {
  buildHeap(E);
  for (int i = E.length - 1; i > 0; i--) {
    swap(E[0], E[i]);
    sink(E, i, 0);
  }
}
```





# Heapsort – Analysis

- Die Worst-Case Komplexität von *Heapify* ist maximal $\lfloor 2 \cdot \log n \rfloor$ für $n$ Knoten.
  - für einen Heap mit Level $k$, gibt es $2 \cdot k$ Vergleiche im Worst-Case
- Die Worst-Case Komplexität von *buildHeap* ist $\Theta(n)$ (Beweis: nächste Folie)
- Für den Heapsort erhalten wir somit:

$$W(n) = (\sum_{i=1}^{n-1} 2 \cdot \lfloor \log n \rfloor) + n \leqslant 2 \cdot \int_1^n (\log e) \ln x dx + n \leqslant 2 \cdot n \cdot \log n + c \cdot n$$

$$\Rightarrow W(n) \in \mathcal{O}(n \cdot \log n)$$

- Es wird kein zusätzlicher Speicherplatz benötigt.





# Heapsort – Heapeigenschaften

**Lemma**

*Ein n-elementiger Heap hat die Höhe* $\lfloor \lg n \rfloor$.

**Lemma**

*Ein Heap hat maximal* $\lceil n/2^{h+1} \rceil$ *Knoten in der Höhe h.*

Beweise siehe Übung 2.





# Heapsort – Komplexitätsanalyse

Die Worst-Case Komplexität von *buildHeap* ist $\Theta(n)$
Beweis:

- Die Laufzeit von *Heapify* für einen Knoten der Höhe $h$ ist in $\mathcal{O}(h)$.
- $\lceil n/2^{h+1} \rceil$ =Anzahl der Knoten in Höhe $h$. Daraus folgt für *buildHeap*:

$$\begin{aligned} \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \mathcal{O}(h) &= \mathcal{O}\left( n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) \qquad \mid \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2 \\ &= \mathcal{O}\left( n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= \mathcal{O}(n) \end{aligned}$$

# Heapsort - Zusammenfassung

- Heapsort sortiert in $\mathcal{O}(n \cdot \log n)$
- Heapsort ist ein in-place Algorithmus.
- Heapsort ist nicht stabil.





# Erinnerung: Die Prioritätswarteschlange (I)

- Betrachte Elemente, die mit einem Schlüssel (key) versehen sind.
- Jeder Schlüssel sei höchstens an einem Element vergeben.
- Schlüssel werden als Priorität betrachtet.
- Die Elemente werden nach ihrer Priorität sortiert.





# Erinnerung: Die Prioritätswarteschlange (II)

**Prioritätswarteschlange (priority queue)**

- `void insert(PriorityQueue pq, int e, int k)` fügt das Element `e` mit dem Schlüssel `k` in `pq` ein.
- `int getMin(PriorityQueue pq)` gibt das Element mit dem kleinsten Schlüssel zurück; benötigt nicht-leere `pq`.
- `void delMin(PriorityQueue pq)` entfernt das Element mit dem kleinsten Schlüssel; benötigt nicht-leere `pq`.
- `int getElt(PriorityQueue pq, int k)` gibt das Element `e` mit dem Schlüssel `k` aus `pq` zurück; `k` muss in `pq` enthalten sein.
- `void decrKey(PriorityQueue pq, int e, int k)` setzt den Schlüssel von Element `e` auf `k`; `e` muss in `pq` enthalten sein.
  `k` muss außerdem kleiner als der bisherige Schlüssel von `e` sein.

Mit Heaps ist eine effiziente Implementierung möglich.





# Drei Prioritätswarteschlangenimplementierungen

| | Implementierung | | |
|---|---|---|---|
| Operation | unsortiertes Array | sortiertes Array | Heap |
| `isEmpty(pq)` | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ |
| `insert(pq,e,k)` | $\Theta(1)$ | $\Theta(n)^*$ | $\Theta(\log n)$ |
| `getMin(pq)` | $\Theta(n)$ | $\Theta(1)$ | $\Theta(1)$ |
| `delMin(pq)` | $\Theta(n)^*$ | $\Theta(1)$ | $\Theta(\log n)$ |
| `getElt(pq,k)` | $\Theta(n)$ | $\Theta(\log n)^\dagger$ | $\Theta(n)$ |
| `decrKey(pq,e,k)` | $\Theta(n)$ | $\Theta(\log n)^\dagger$ | $\Theta(\log n)$ |

$^*$Beinhaltet das Verschieben aller Elemente „rechts“ von `k`.
$^\dagger$Mittels binärer Suche.