Datenstrukturen & Algorithmen

Peppo Brambilla
Universität Bern
Frühling 2018

Übersicht

Sortieralgorithmen

- Einleitung
- Heapsort
- Quicksort

Motivation

- Sortieren ist Voraussetzung für viele Anwendungen
  - Nach Bestellnummern sortierte Aufträge, etc.
- Viele Algorithmen enthalten Sortierschritte
  - Sortieren häufig Voraussetzung für effizientes Suchen
  - Beispiel: GUIs (graphical user interfaces)
    - Zeichnen von grafischen Objekten die sich gegenseitig verdecken
    - Objekte sollen gemäss „liegt vor“ oder „liegt hinter“ Relation geordnet werden
- In kommerzieller Datenverarbeitung wird mehr als 25% der Rechenzeit mit Sortieren verbracht

Records und Keys

- Datensatz besteht aus Menge von Records
- Jeder Record enthält einen Key (Schlüssel) und zusätzliche Daten („Satellitendaten“)
- Daten werden anhand der Schlüssel sortiert

```
class Record {
 int key;
 SatelliteData data;
}
```

```
class SatelliteData{
 String name;
 String address;
 ...
}
```

Ausgangslage

- Sortiere Sequenz von ganzen Zahlen
  - Sortieren von Records mit Keys im Prinzip gleich
  - Implementierung mit Java siehe Übungen
- Eingabe: Zahlen $\langle a_1, a_2, \ldots, a_n \rangle$
- Ausgabe: Permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$
  so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$
- Beispiel
  - Eingabe: 8 2 4 9 3 6
  - Ausgabe: 2 3 4 6 8 9

Übersicht

Sortieralgorithmen

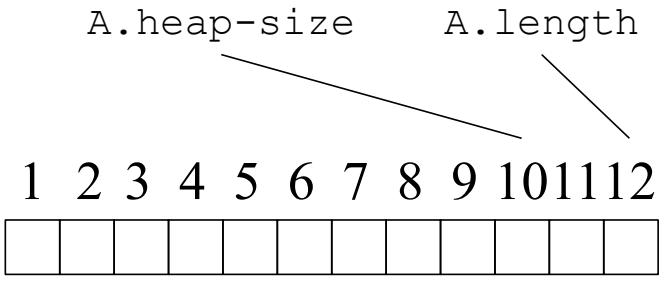
- Einleitung
- Heapsort
- Quicksort

Heapsort

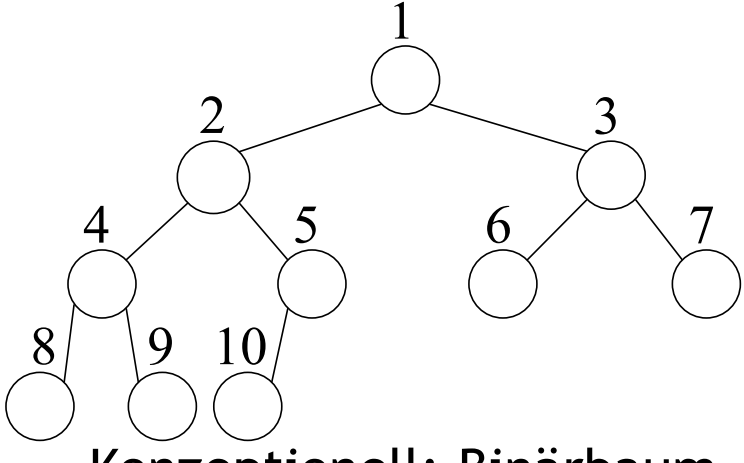
- Sortierverfahren mit Zeitkomplexität $O(n \lg n)$
- In-place Verfahren
  - Speichert nur konstante Anzahl Elemente ausserhalb des Eingabefeldes
  - Sortieren durch Mischen (Merge Sort) ist nicht in-place!
- Heap Datenstruktur zur Verwaltung der Daten
  - Heaps auch nützlich in anderen Algorithmen
  - Prioritätswarteschlangen

Heaps

- Felder mit einer zusätzlichen Struktur
  - Heap `A`
  - `A.length` Anzahl Elemente des Feldes
  - `A.heap-size` Anzahl Elemente im Heap
- Heap kann als Binärbaum angesehen werden
  - Jeder Knoten des Baumes entspricht einem Element des Feldes
  - Reihenfolge wie in Skizze



Implementierung: Feld `A`



Konzeptionell: Binärbaum

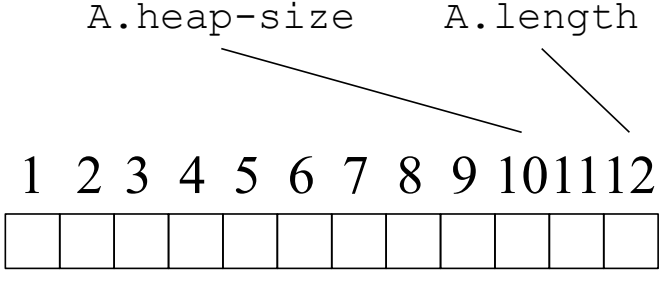
Heaps

- Wurzel: `A[1]`
- Berechnung der Indizes

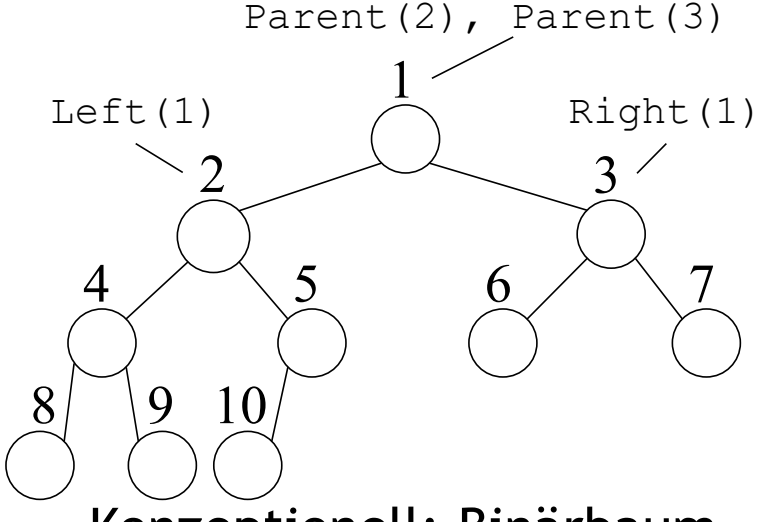
```
Parent(i){ return floor(i/2); }
Left(i)  { return 2*i;   }
Right(i) { return 2*i+1; }
```

ist ein shift-right 1001 → 100

shift-left 1001 → 10010



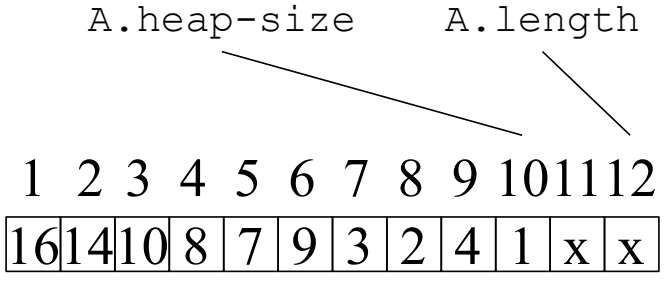
Implementierung: Feld `A`



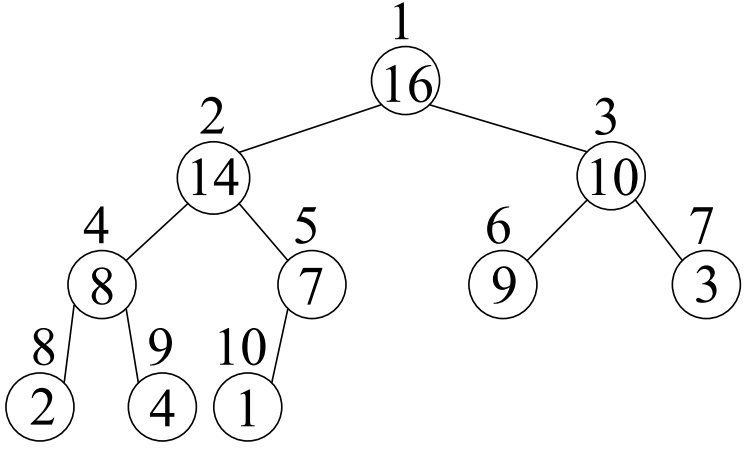
Konzeptionell: Binärbaum

Heap Eigenschaft

- Max-Heap `A[Parent(i)]>=A[i]`
- Min-Heap `A[Parent(i)]<=A[i]`



Implementierung: Feld `A`



Konzeptionell: Binärbaum

Beispiel: Max-Heap

Aufrechterhaltung der Heapeigenschaft

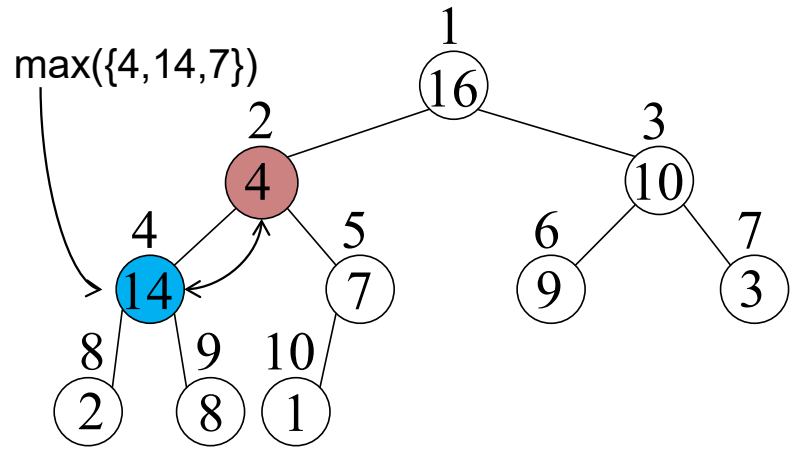
- Ausgangslage
  - Gegeben Index `i`
  - Bäume ausgehend von `Left(i)` und `Right(i)` sind Heaps
  - Knoten `i` verletzt Heapeigenschaft
- Ziel: Heapeigenschaft wieder herstellen

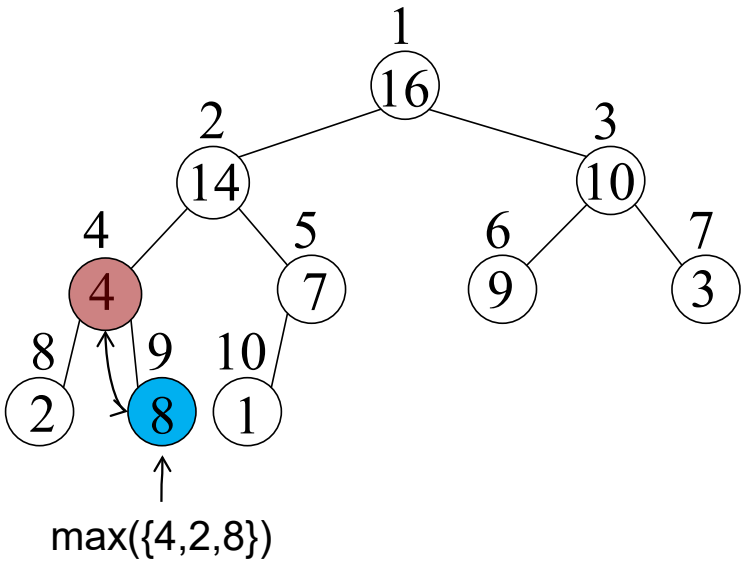
MAX-HEAPIFY(A, i)

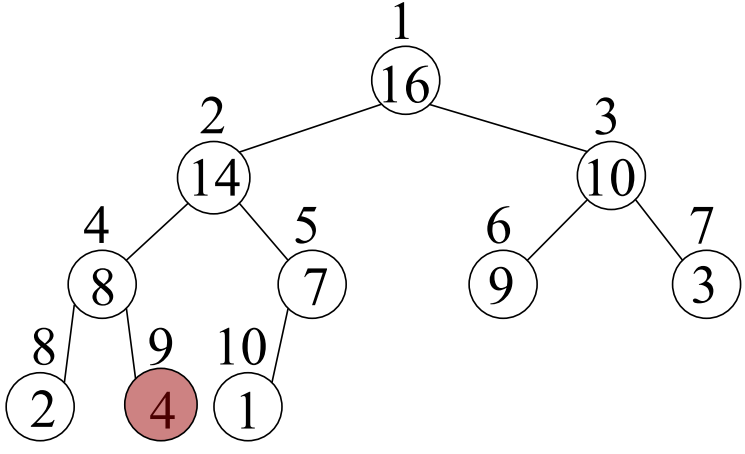
```
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4        largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7        largest = r
8  if largest ≠ i
9        exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```

Aufrechterhaltung der Heapeigenschaft

- `Max-Heapify(A,2)`







Laufzeit von `Max-Heapify`

- Zugriff auf Knoten `A[i]`, `A[Left(i)]` und `A[Right(i)]` in $\Theta(1)$
- Von Kindern ausgehende Bäume haben höchstens $2n/3$ Knoten
- Lösung von $T(n) \leq T(2n/3) + \Theta(1)$ mit Mastertheorem ergibt $T(n) = O(\lg n)$

Heap Konstruktion

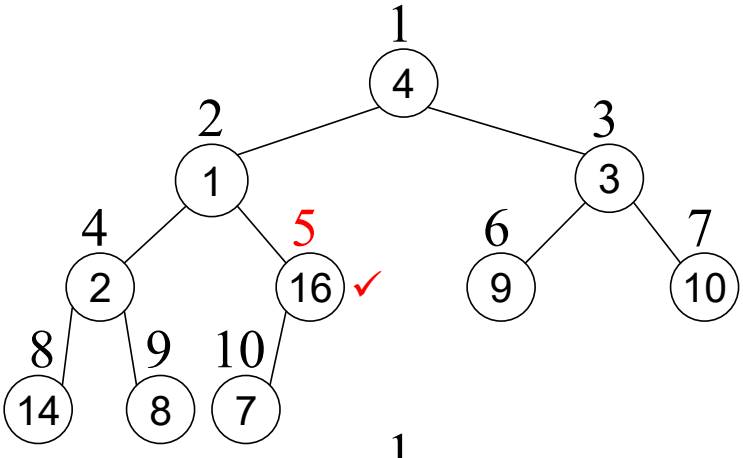
- Problem: gegeben beliebiges Feld, etabliere Heapeigenschaft
- „Bottom-up“ Max-Heapify
  - Durchlaufe alle Knoten (ausser Blätter) und führe auf jedem `Max-Heapify` aus

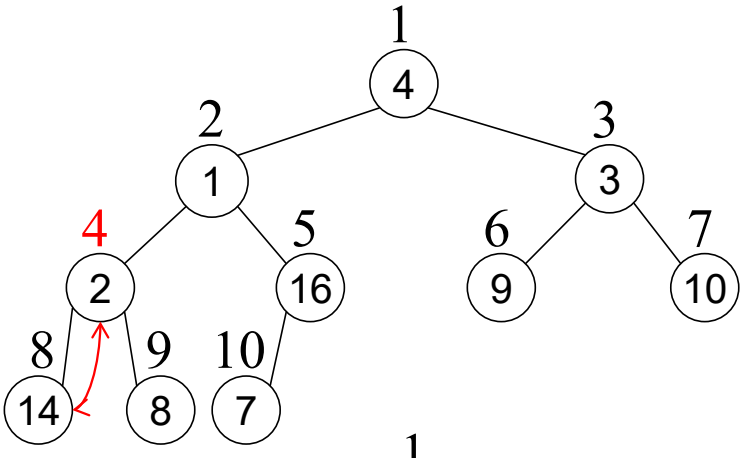
$$
\begin{array}{l}
\textsc{Build-Max-Heap}(A) \\
1 \quad A.heap\text{-}size = A.length \\
2 \quad \textbf{for } i = \lfloor A.length/2 \rfloor \textbf{ downto } 1 \\
3 \qquad \textsc{Max-Heapify}(A, i)
\end{array}
$$

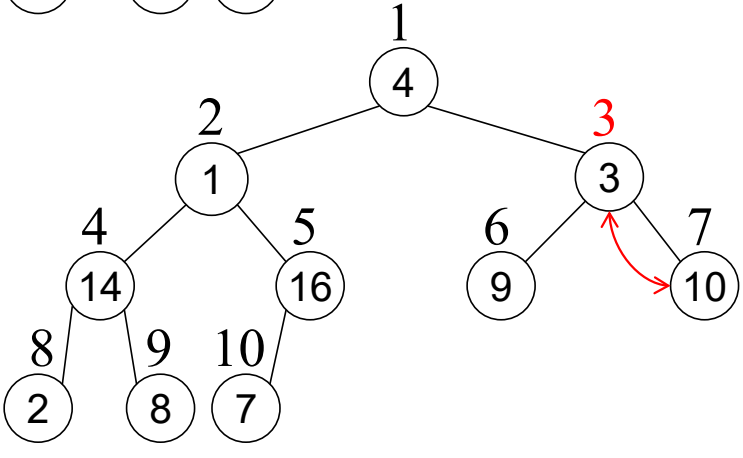
- Korrektheit: Schleifeninvariante „jeder Knoten $i + 1, i + 2, \ldots, n$ ist Wurzel eines Max-Heap“
- Laufzeit $O(n)$ (siehe Buch)

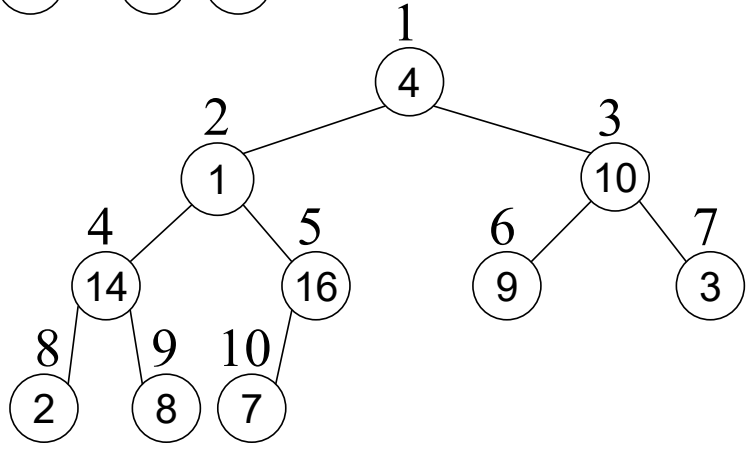
Heap Konstruktion

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

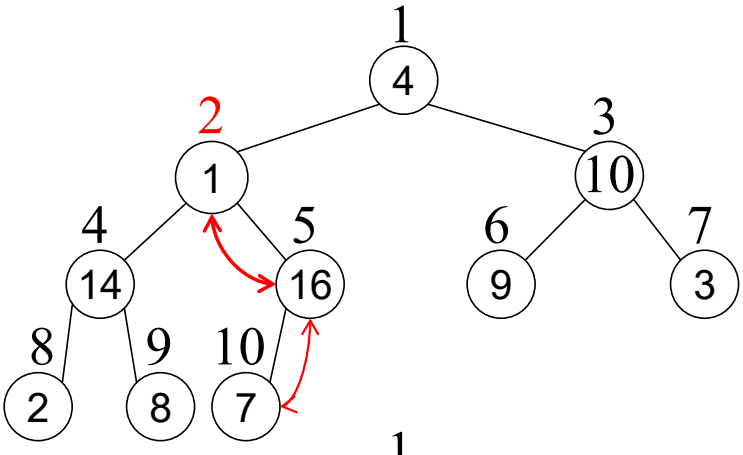


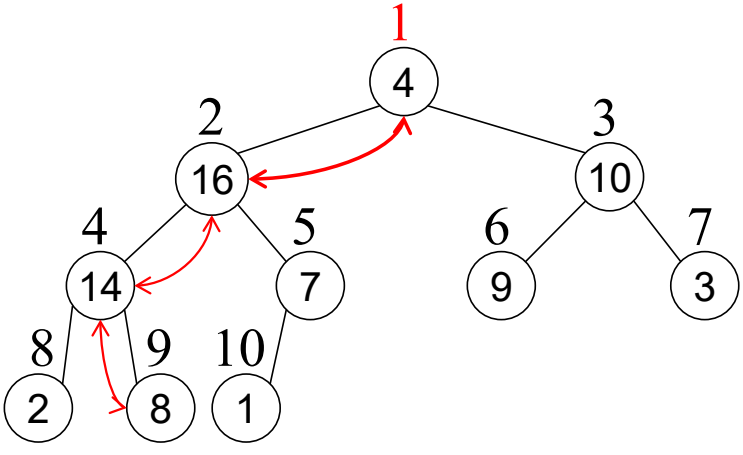


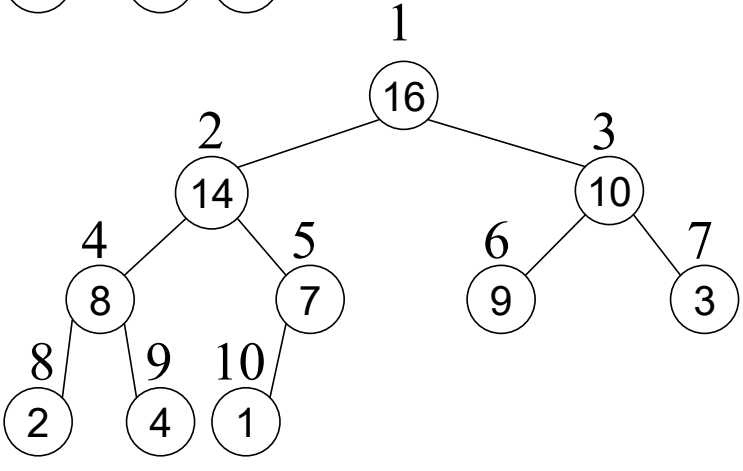




Heap Konstruktion







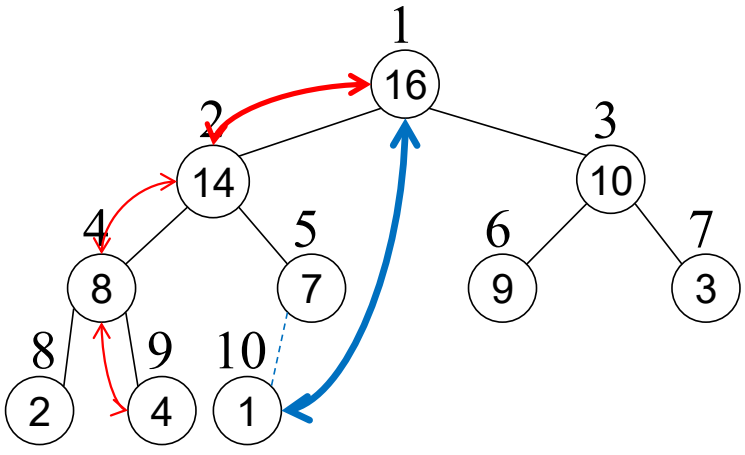
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

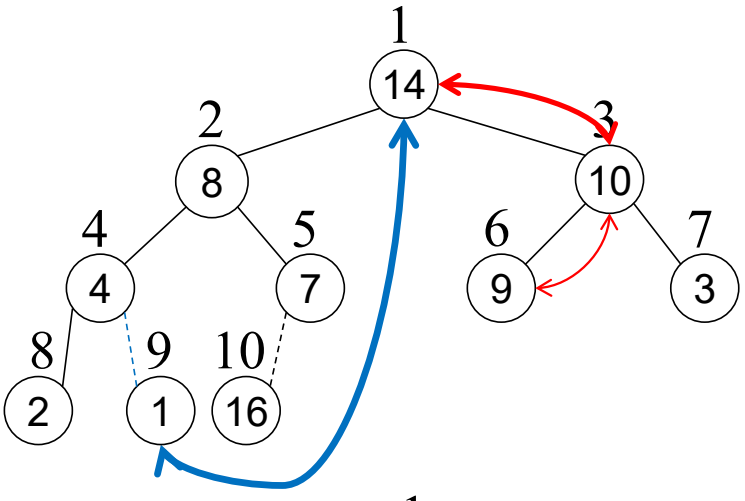
Heap Sort

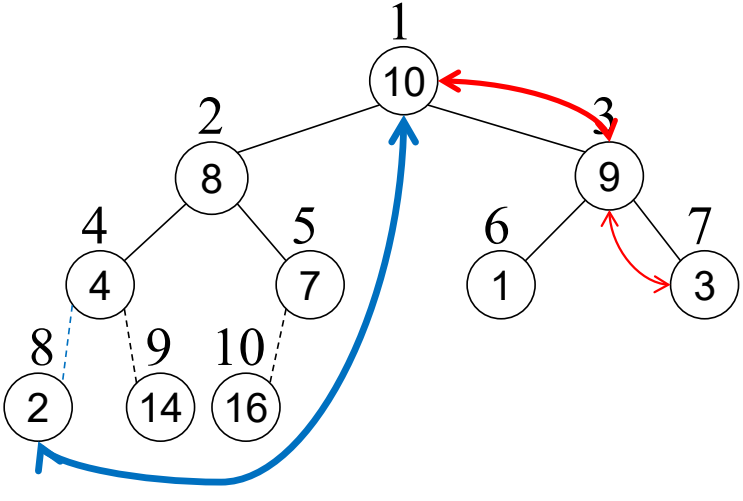
1. Konstruiere Heap
2. Verschiebe grösstes Element im Heap an sortierte Position bis Heap leer
   - Nach jeder Verschiebung Heap Eigenschaft aufrechterhalten

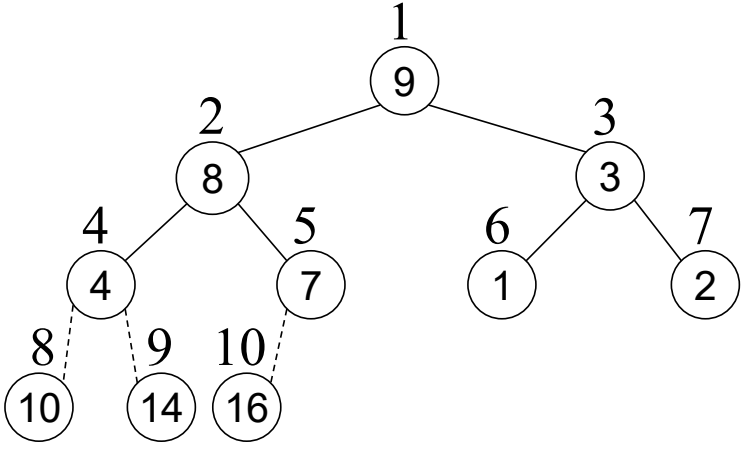
```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

Heap Sort

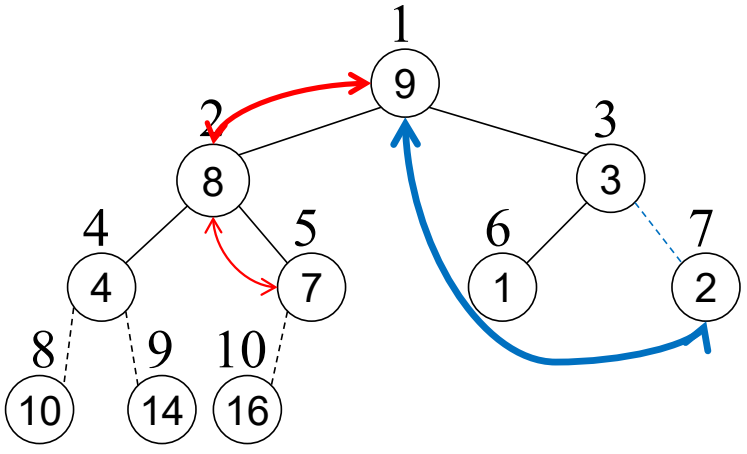

1
16
2
14
3
10
4
8
5
7
6
9
7
3
8
2
9
4
10
1

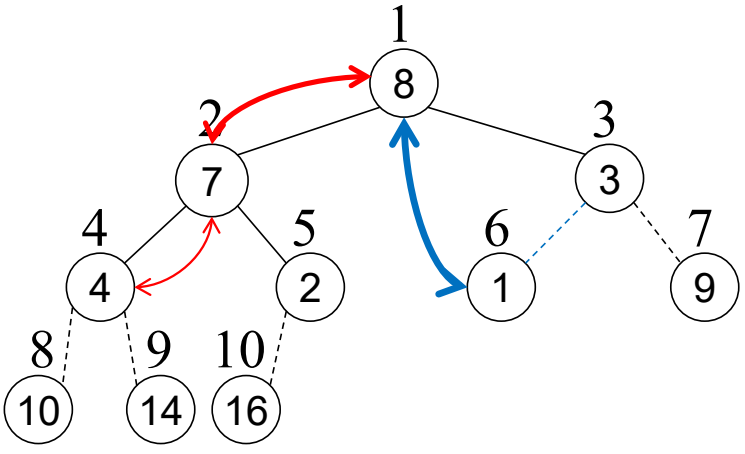

1
14
2
8
3
10
4
4
5
7
6
9
7
3
8
2
9
1
10
16

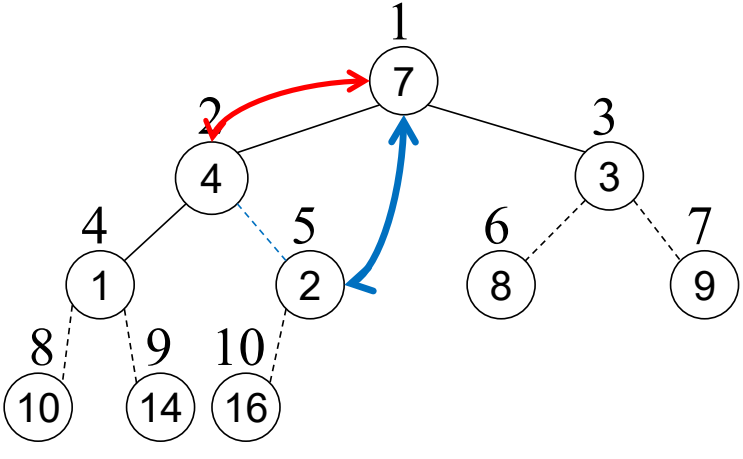

1
10
2
8
3
9
4
4
5
7
6
1
7
3
8
2
9
14
10
16

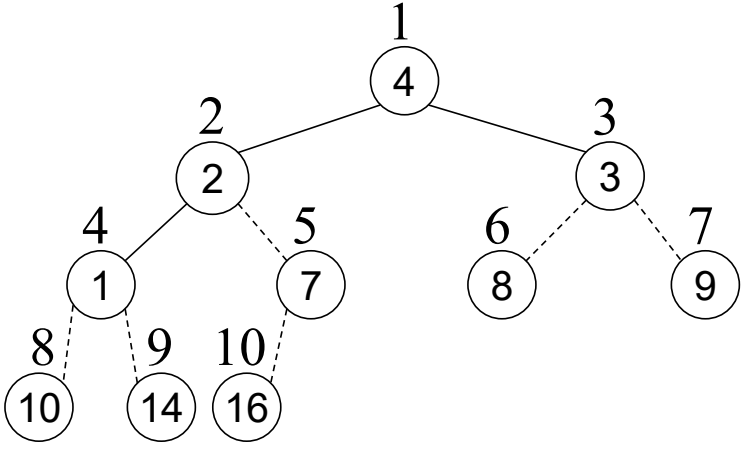

1
9
2
8
3
3
4
4
5
7
6
1
7
2
8
10
9
14
10
16

Heap Sort

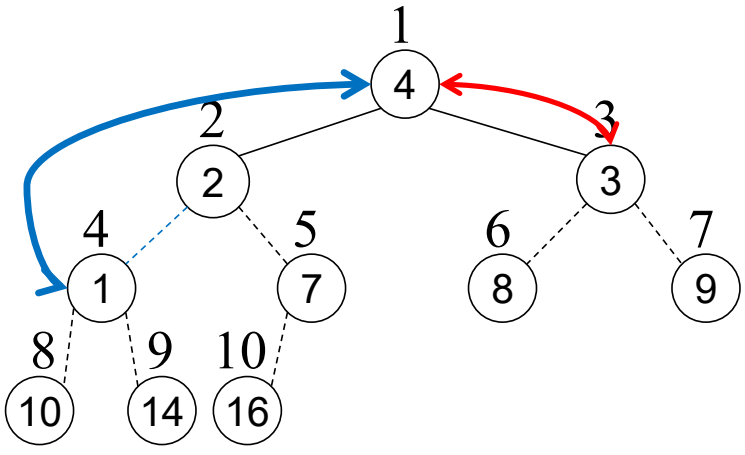

1
9
2
8
3
3
4
4
5
7
6
1
7
2
8
10
9
14
10
16

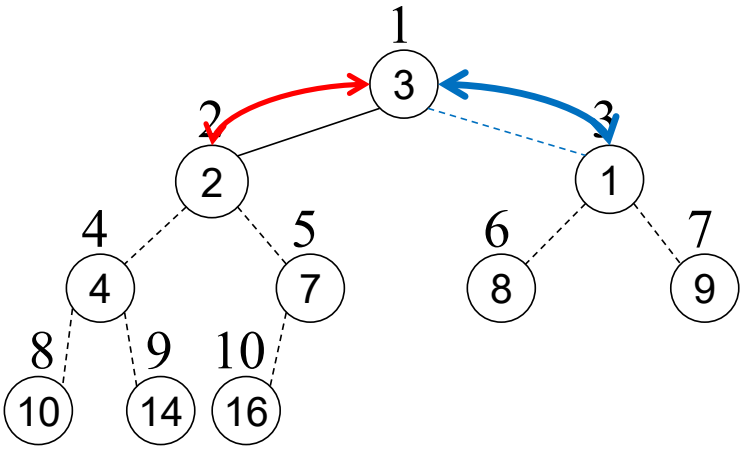

1
8
2
7
3
3
4
4
5
2
6
1
7
9
8
10
9
14
10
16

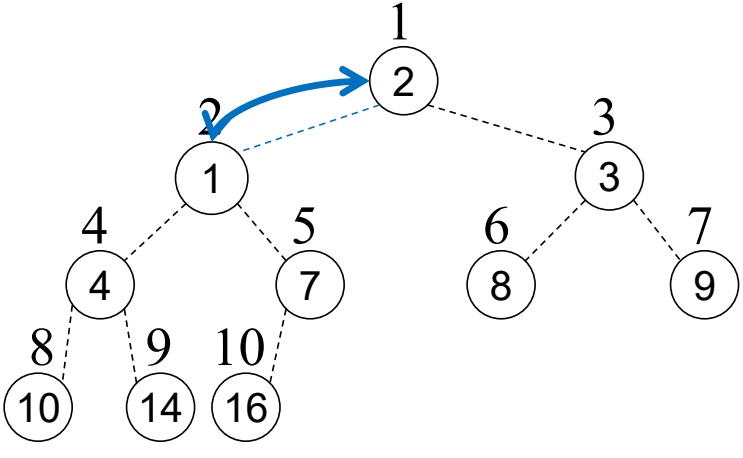

1
7
2
4
3
3
4
1
5
2
6
8
7
9
8
10
9
14
10
16


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

Heap Sort







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

Aufwand

- Heap Konstruktion: $O(n)$
- Aufrechterhaltung der Heap Eigenschaft:

$$(n-1)O(\lg n) = O(n \lg n)$$

- Total: Konstruktion + Aufrechterhaltung

$$O(n) + O(n \lg n) = O(n \lg n)$$

Prioritätswarteschlangen

- (Priority queues)
- Datenstruktur zur Verwaltung einer Menge von Elementen, wobei jedes Element einen Schlüssel hat
- Operationen
  - Insert: Einfügen eines neuen Elements
  - Maximum: Abfragen des Elements mit grösstem (kleinstem) Schlüssel
  - Extract-Max: Entfernen des Elements mit grösstem (kleinsten) Schlüssel
  - Increase-Key: Verändern des Schlüssels eines Elements

Anwendungen

- Verwaltung von Arbeitsaufträgen geordnet nach Prioritäten (Zeitplanung, Scheduling)
  - Rechenaufträge auf gemeinsam genutztem Rechner
  - Prozessverwaltung in Multitasking
- Implementation mit Heap Datenstruktur

Implementation mit Heaps

```
HEAP-MAXIMUM(A)
1  return A[1]

HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2        error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

Implementation mit Heaps

HEAP-INCREASE-KEY(A, i, key)

```
1  if key < A[i]
2        error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5        exchange A[i] with A[PARENT(i)]
6        i = PARENT(i)
```

MAX-HEAP-INSERT(A, key)

```
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = −∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

Übersicht

Sortieralgorithmen

- Einleitung
- Heapsort
- Quicksort

Quicksort

- Entwickelt von C. A. R. Hoare in 1962
- Teile-und-beherrsche (divide-and-conquer) Schema
- Sortiert „in-place“
- Beliebt in Praxis
  - Gilt als einer der schnellsten Sortier-algorithmen für grosse Datensätze

Teile-und-beherrsche

- Sortiere ein Feld A mit n Elementen

1. Teile: Zerlege Feld in zwei Teilfelder um ein Pivot q, so dass {Elemente links von q} $\leq A[q] \leq$ {Element rechts von q}

$x = A[q]$

2. Beherrsche: Sortiere Teilfelder rekursiv
3. Verbinde: Teilfelder zusammenfügen, trivial

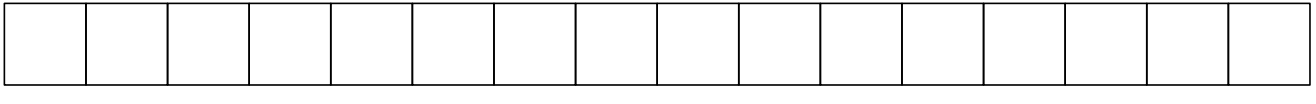
- Schlüssel zum Erfolg: Zerlegung des Felds in linearer Zeit

Zerlegung des Felds

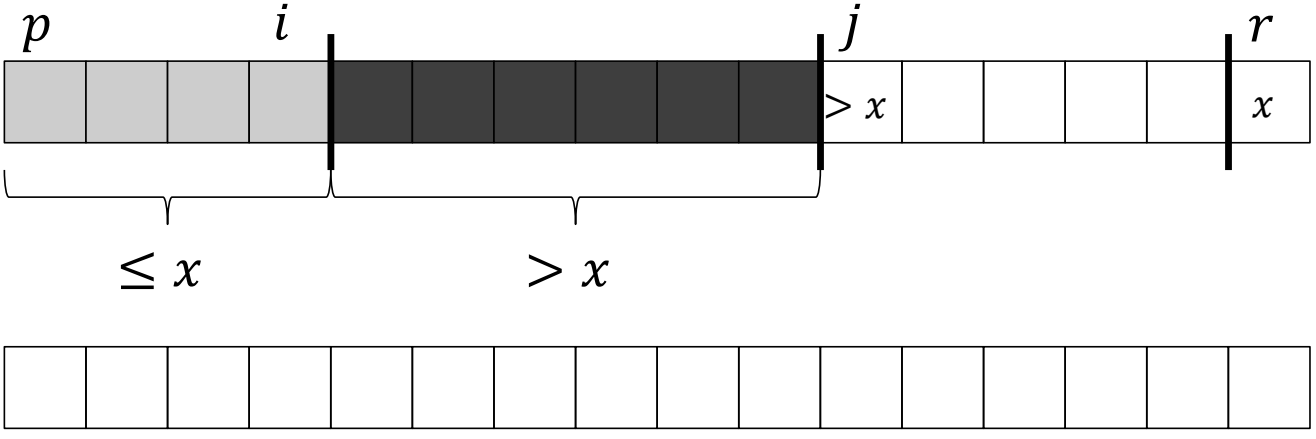
```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4        if A[j] ≤ x
5              i = i + 1
6              exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

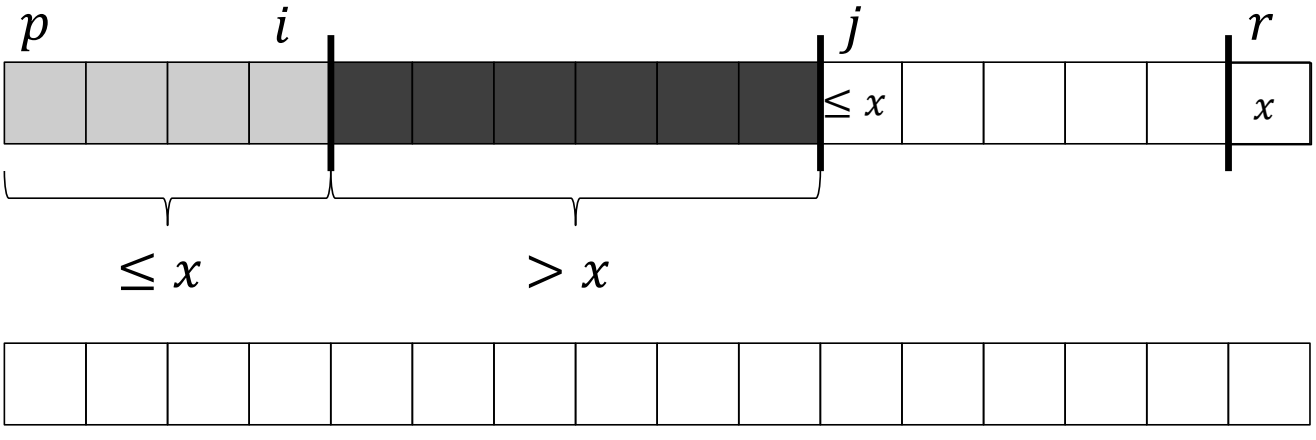
- Schleifeninvariante (vgl. Buch)

1. If $p \leq k \leq i$, then $A[k] \leq x$.
2. If $i + 1 \leq k \leq j - 1$, then $A[k] > x$.
3. If $k = r$, then $A[k] = x$.

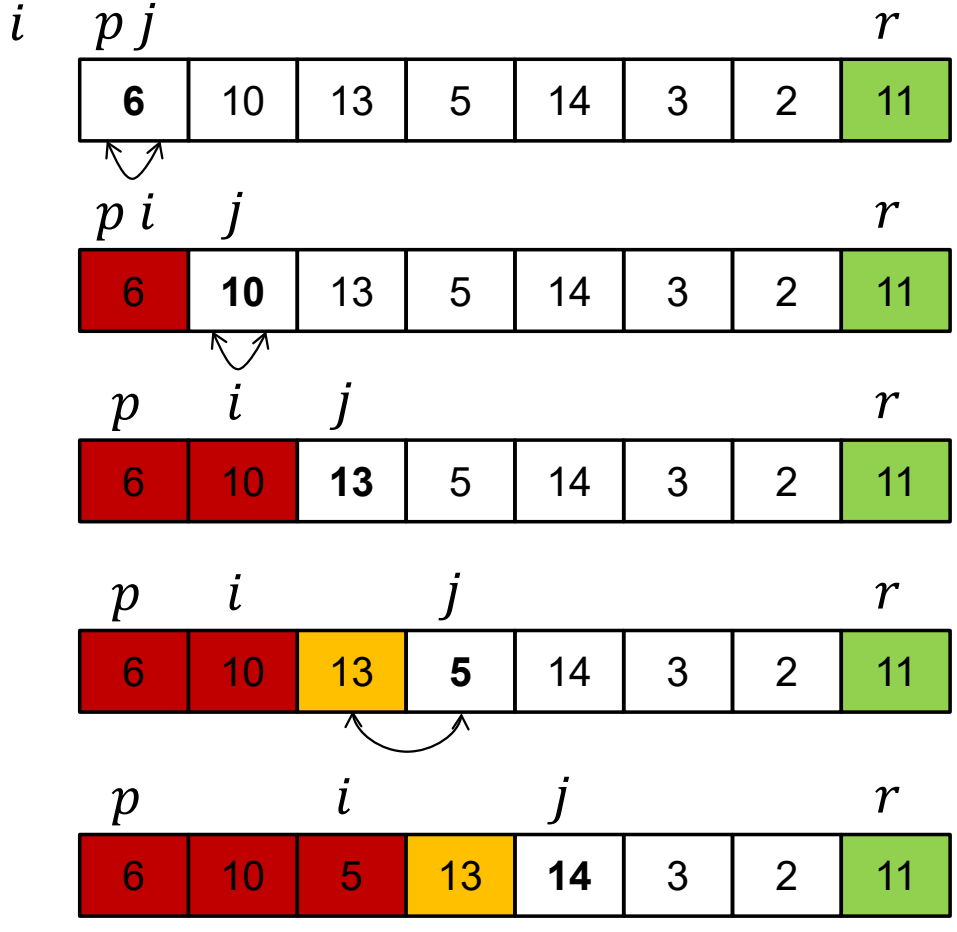


Schleifeninvariante

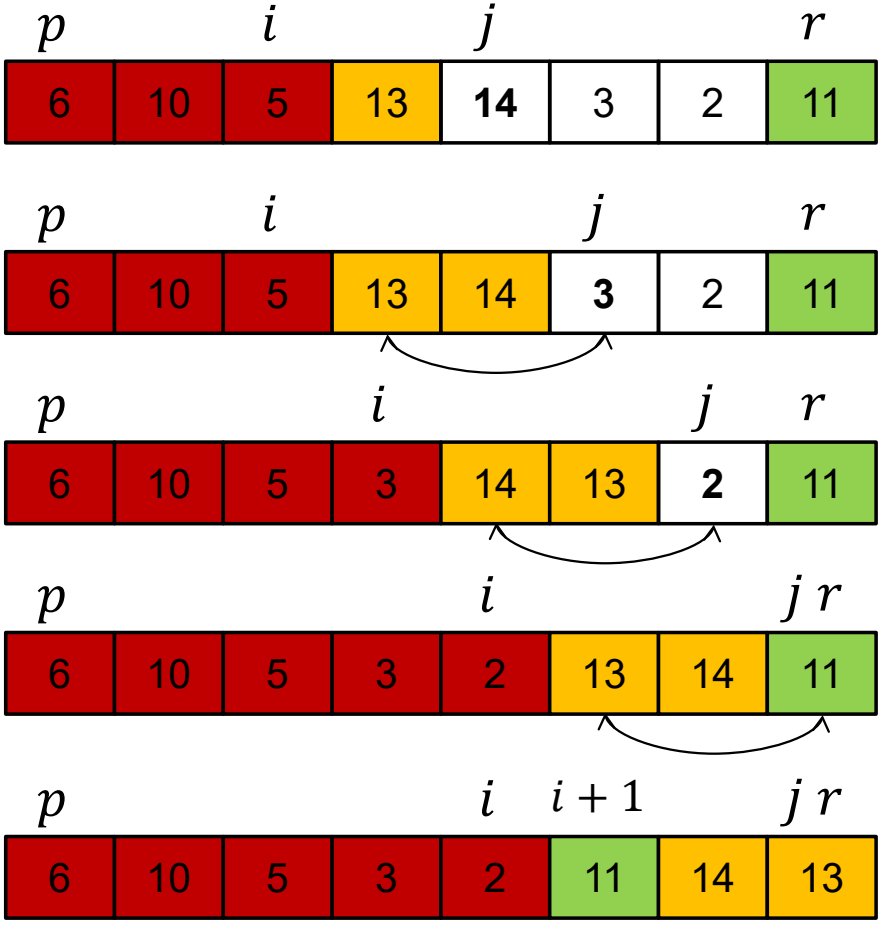




Beispiel


i p j r
6 10 13 5 14 3 2 11
p i j r
6 10 13 5 14 3 2 11
p i j r
6 10 13 5 14 3 2 11
p i j r
6 10 13 5 14 3 2 11
p i j r
6 10 5 13 14 3 2 11

Beispiel


p i j r
6 10 5 13 14 3 2 11
p i j r
6 10 5 13 14 3 2 11
p i j r
6 10 5 3 14 13 2 11
letzte Ausführung der Schleife
p i j r
6 10 5 3 2 13 14 11
exchange A[i+1] with A[r]
p i i + 1 j r
6 10 5 3 2 11 14 13
nach Verschieben des Pivots A[r]

Quicksort

$\textsc{Quicksort}(A, p, r)$

1 **if** $p < r$
2 $\quad q = \textsc{Partition}(A, p, r)$
3 $\quad \textsc{Quicksort}(A, p, q-1)$
4 $\quad \textsc{Quicksort}(A, q+1, r)$

- Aufruf

```
Quicksort(A, 1, A.length)
```

Analyse

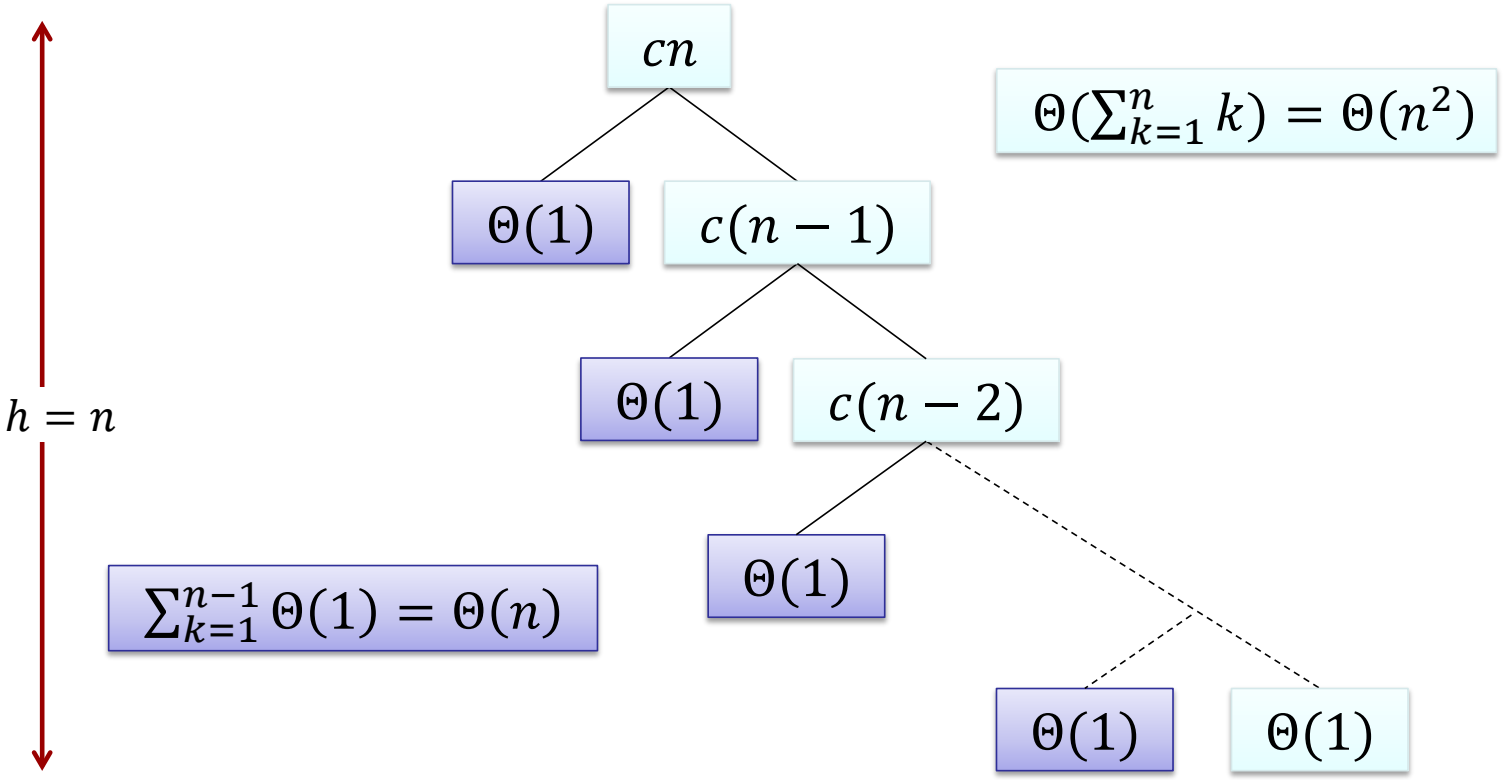
- Annahme: keine doppelten Elemente
  - Alle Elemente sind verschieden
- Worst case
  - Eingabe ist sortiert
  - Zerlegung immer um grösstes oder kleinstes Element (je nachdem ob Eingabe aufsteigend oder absteigend sortiert war)
  - Eine Seite der Zerlegung ist leer

$$T(n) = T(0) + T(n-1) + \Theta(n)$$

  - → Rekursionsbaum

Worst case

$$T(n) = T(0) + T(n-1) + \Theta(n)$$



$$T(n) = \Theta(n) + \Theta(n^2) = \Theta(n^2)$$

Analyse

- Best case
  - Nur für Intuition
  - Gleichmässige Zerlegung

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$
$$= \Theta(n \lg n)$$

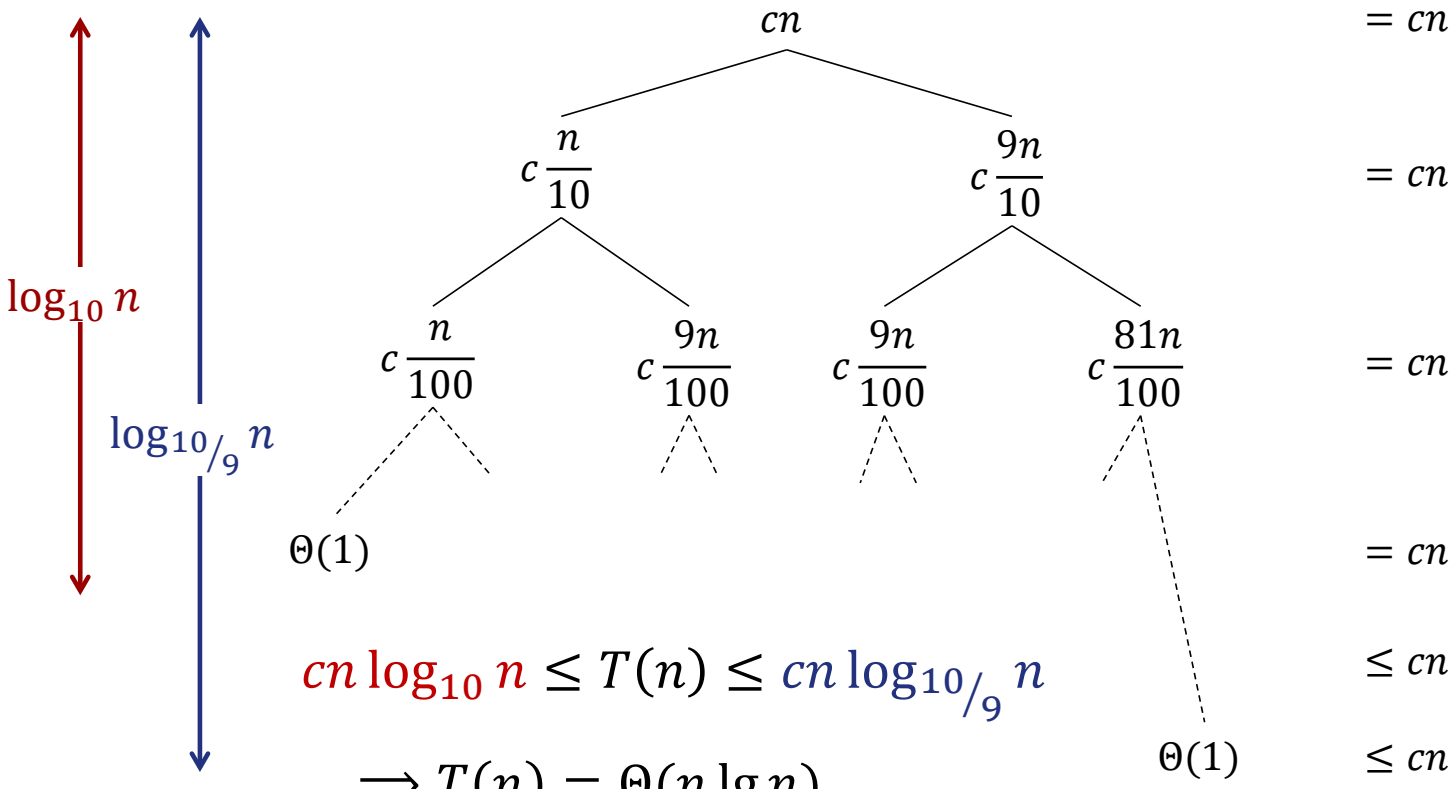
  - Was wenn Zerlegung im Verhältnis $\frac{1}{10} : \frac{9}{10}$?

$$T(n) = T\left(\frac{1}{10}n\right) + T\left(\frac{9}{10}n\right) + \Theta(n)$$

  - → Rekursionsbaum

„Fast“ best case Rekursionsbaum

$$T(n) = T\left(\frac{1}{10}n\right) + T\left(\frac{9}{10}n\right) + \Theta(n)$$



$$cn \log_{10} n \leq T(n) \leq cn \log_{10/9} n$$

$$\Longrightarrow T(n) = \Theta(n \lg n)$$

Randomisierter Quicksort

Idee: Zerlegung um ein zufälliges Element

RANDOMIZED-PARTITION(A, p, r)

```
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)
```

RANDOMIZED-QUICKSORT(A, p, r)

```
1  if p < r
2        q = RANDOMIZED-PARTITION(A, p, r)
3        RANDOMIZED-QUICKSORT(A, p, q − 1)
4        RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Randomisierter Quicksort

Idee: Zerlegung um ein zufälliges Element

- Laufzeit unabhängig von Eingabe
- Keine Annahmen über Eingabe nötig
- Keine spezielle Eingabe führt zu worst case
- Worst case tritt immer noch auf bei „unglücklicher“ Folge von zufälligen Elementen

Analyse

- Analyse hier folgt Problemstellung 7-2 im Buch (Werte nicht paarweise verschieden)
  - Anders als Analyse in Abschnitt 7.4.2
  - Natürlich mit gleichem Resultat
- Grundlagenmaterial in Abschnitt 5.2, 5.3, und Anhang C nachlesen

Analyse

- Indikatorfunktionen

$$X_k = \begin{cases} 1 \text{ if Partition generates a } k : n-k-1 \text{ split} \\ 0 \text{ otherwise} \end{cases}$$

- Erwartungswert: $E[X_k] = \Pr\{X_k = 1\} = 1/n$
- Kosten

$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n) & \text{if } 0 : n-1 \text{ split} \\ T(1) + T(n-2) + \Theta(n) & \text{if } 1 : n-2 \text{ split} \\ \vdots & \\ T(n-1) + T(0) + \Theta(n) & \text{if } n-1 : 0 \text{ split} \end{cases}$$

$$= \sum_{k=0}^{n-1} X_k \cdot \big(T(k) + T(n-k-1) + \Theta(n)\big)$$

Erwartungswert

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k \cdot (T(k) + T(n-k-1) + \Theta(n))\right]$$

Erwartungswert ist linear

$$= \sum_{k=0}^{n-1} E\left[X_k \cdot (T(k) + T(n-k-1) + \Theta(n))\right]$$

Zufallsvariablen sind unabhängig

$$= \sum_{k=0}^{n-1} \underbrace{E[X_k]}_{1/n} \cdot E[T(k) + T(n-k-1) + \Theta(n)]$$

Erwartungswert ist linear

$$= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} E[\Theta(n)]$$

Erwartungswert

$$E[T(n)] = \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} E[\Theta(n)]$$

identische Terme in Summen

$$= \frac{2}{n}\sum_{k=0}^{n-1} E[T(k)] + \Theta(n)$$

$$= \frac{2}{n}\sum_{k=2}^{n-1} E[T(k)] + \underbrace{\frac{2}{n}\big(E[T(0)] + E[T(1)]\big) + \Theta(n)}_{\Theta(n)}$$

$$= \frac{2}{n}\sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

Rekursionsgleichung

$$E[T(n)] = \frac{2}{n} \sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

- Beweise

$$E[T(n)] \leq an \lg n \text{ für Konstante } a > 0$$

- Benutze

$$\sum_{k=2}^{n-1} k \lg k \leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2$$

Substitutionsmethode

$$
\begin{aligned}
E[T(n)] &= \frac{2}{n}\sum_{k=2}^{n-1} E[T(k)] + \Theta(n) \\
&\leq \frac{2}{n}\sum_{k=2}^{n-1} ak\lg(k) + \Theta(n) \\
&= \frac{2a}{n}\sum_{k=2}^{n-1} k\lg(k) + \Theta(n) \\
&\leq \frac{2a}{n}\left(\frac{1}{2}n^2\lg(n) - \frac{1}{8}n^2\right) + \Theta(n) \\
&= \underbrace{an\lg(n)}_{\text{gewünscht}} - \underbrace{\left(\frac{an}{4} - \Theta(n)\right)}_{\text{Residuum}} \\
&\leq an\lg(n) \qquad \text{wenn } a \text{ gross genug gewählt}
\end{aligned}
$$

Zusammenfassung

- Max-Heap
  - Feld `A` mit `A.heap-size` $\leq$ `A.length`
  - Konzept: binärer Baum
  - Eigenschaft: `A[Parent(i)]` $\geq$ `A[i]`
  - $\textsc{Max-Heapify}(A, i)$ (repariert Heap)
    - Voraussetzung: Unterbäume sind Heaps
    - $O(\lg n)$
  - Heap Konstruktion
    - Bottom-up $\textsc{Max-Heapify}$
    - $O(n)$

Zusammenfassung

- Heapsort
  - In-place Verfahren
  - Kombiniert Datenstrukturen und Algorithmen
  - $O(n \lg n)$
- Prioritätswarteschlangen
  - Operationen:
    Insert, Maximum, Extract-Max, Increase-Key
  - Effiziente Implementation mit Heaps

Zusammenfassung

- Quicksort
  - In-place Verfahren
  - Teile-und-beherrsche Schema
  - Zerlegen des Feldes um Pivot Element
  - Vertauschungen: keine Blöcke werden kopiert (Folge: Sortierverfahren ist instabil)
  - Worst case: $\Theta(n^2)$, average case: $\Theta(n \lg n)$
- Randomisierter Quicksort
  - Pivot-Element zufällig auswählen und mit letztem Element vertauschen

Nächste Vorlesung

- Sortieren in linearer Zeit, Kapitel 8