Problemstellung Sortieren

Eingabe: Eine Folge

$$\langle a_1, \ldots, a_n \rangle$$

von Zahlen.

Ausgabe: Eine Permutation

$$\langle a'_1, \ldots, a'_n \rangle$$

der Zahlen $a_1, \ldots, a_n$ mit

$$a'_i \leq a'_{i+1}$$

für $0 < i < n$.

Die Struktur der Daten

- Normalerweise sortiert man nicht nur reine Zahlen, sondern **Datensätze** (records, Tupel) nach einem Feld (Attribut).
- Dieses Feld wird auch Schlüssel (key) genannt.
- Da Datensätze groß sein können, werden Zeiger auf die Datensätze benutzt.
- Das spart Kopierkosten!
- Trotzdem formulieren wir Algorithmus mit Zahlen.
- Umformulierung/Umkodierung für konkretes Problem notwendig.
- Ersichtlich: Algorithmus ist Abstraktion.

Übersicht

Wir hatten bis jetzt:

Sortier-Algorithmus	worst-case	in-place
INSERTION-SORT	$\Theta(n^2)$	yes
MERGE-SORT	$\Theta(n \lg n)$	NO

Jetzt kommen:

Sortier-Algorithmus	worst-case	in-place
HEAP-SORT	$\Theta(n \lg n)$	yes
QUICK-SORT	$\Theta(n^2)$	yes

QUICK-SORT: avg-case von $\Theta(n \lg n)$ und in praxi sehr schnell.

Übersicht

Reine Sortieralgorithmen:

- Alle oben genannten Sortierverfahren basieren auf Vergleichen.
- Wir werden die Grenzen dieser Verfahren kennenlernen.
  Untere Schranke: $\Omega(n \ln n)$.
- Zwei Sortierverfahren schlagen diese Schranke:
  - COUNTING-SORT
    Für n Zahlen $1, \ldots, k$: $O(n + k)$.
  - RADIX-SORT
    Erweitern des Zahlenbereiches.
  - BUCKET-SORT
    Reelle Zahlen, wobei Wahrscheinlichkeitsverteilung bekannt sein muß.

Übersicht

Dazu lernen wir noch Algorithmen kennen, die die i-t kleinste Zahl aus einer Sequenz von Zahlen heraussuchen.

- Mit Sortieren: $O(n \lg n)$.
- Besseres Verfahren: $O(n)$.
- Noch eins: $O(n^2)$ worst case, aber $O(n)$ avg-case.
- Und noch ein komplizierterer Algorithmus: $O(n)$.

Heapsort

Die wichtigsten Eigenschaften von Heapsort in Kürze:

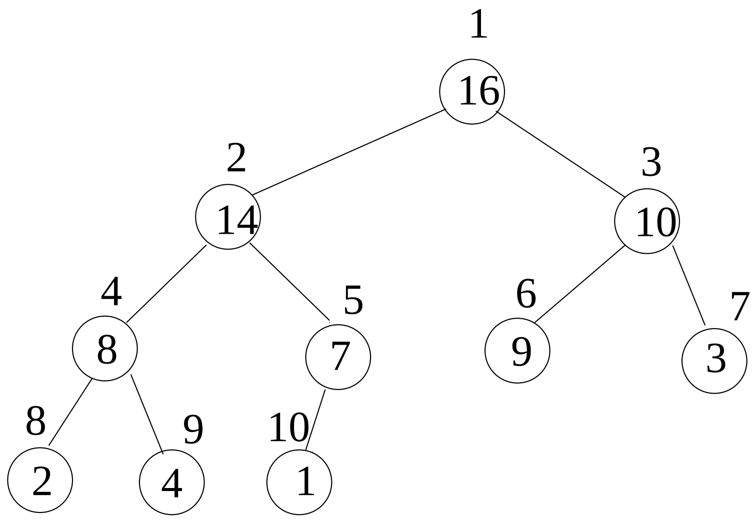
- worst-case: $O(n \lg n)$
- in-place
- benutzte Datenstruktur: Heap
  zur Verwaltung von Information

Ein Heap wird in vielen Algorithmen benötigt, z.B.:

- Prioritätsschlangen (priority queues)

(Binary) Heap

Ein binärer Heap ist ein Array, daß als "vollständiger" binärer Baum angesehen werden kann:



(a)

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

(b)

(Binary) Heap

- Knoten $\leftrightarrow$ Arrayelement
- Baum ist vollständig gefüllt, bis zu einem Knoten auf der untersten Ebene. Die restlichen Knoten rechts davon fehlen.
- Ein Array A, das einen Heap repräsentiert hat zwei Attribute:
  1. **length[A]**: die Anzahl der Elemente von A.
  2. **heap-size[A]**: die Anzahl der Elemente im Heap, der in A gespeichert wird.
     A ist Vorrat an Speicherplätzen.
- Es gilt: **heap-size[A]** $\leq$ **length[A]**
- Kein Element von A nach **A[heap-size[A]]** ist benutzt, obwohl natürlich potentiell alle Elemente **A[1** $\ldots$ **length[A]]** benutzt werden können.

Heap-Indizes

Wie numeriert man die Knoten im Baum, so daß diese Nummern dann gültige Arrayindizes ergeben?

- Die Wurzel bekommt die 1.
  d.h.: A[1] speichert die Wurzel.
- Index des linken Sohnes von Knoten i:

  LEFT(i)
  return $2i$;
- Index des rechten Sohnes von Knoten i:

  RIGHT(i)
  return $2i + 1$;
- Index des Vaters von i:

  PARENT(i)
  return $\lfloor 2i \rfloor$;

Implementierung: Als Macros using bitshifts.

Heap-Eigenschaft

Es gilt die Heap-Eigenschaft:

> Für jeden Knoten außer der Wurzel gilt
>
> $\mathrm{A[PARENT(i)]} \geq \mathrm{A[i]}$

In Worten:

> Der Wert eines Knotens (der in ihm gespeicherten Zahl) ist höchstens so hoch wie der Wert des Vaterknotens.

Heap-Höhe

Def.: Die **Höhe** eines Knotens in einem Baum definieren wir als die Länge des längsten Pfades vom Knoten bis zu einem Blatt.

Die **Höhe** eines Baumes ist dann die Höhe der Wurzel.

Bem: Die Höhe eines Heaps für n Elemente ist $\Theta(\lg n)$. (s. Übung)

Die meisten Prozeduren auf einem Heap benötigen eine Laufzeit, die maximal proportional zur Höhe ist. Also: $O(\lg n)$.

Heap-Operationen

Für das Sortieren interessant:

- HEAPIFY mit $O(\lg n)$
  erhält die Heap-Eigenschaft
- BUILD-HEAP mit $O(n)$
  baut einen Heap aus einem unsortieren Array
- HEAPSORT mit $O(n \lg n)$
  sortiert ein Array in-place

Für Prioritätsschlangen:

- EXTRACT-MAX mit $O(\lg n)$
  gibt Maximum der im Heap gespeicherten Zahlen
- INSERT mit $O(\lg n)$
  fügt Element in Heap ein

HEAPIFY

Ziel:

- Erhalten der Heap-Eigenschaft.

Diese war:

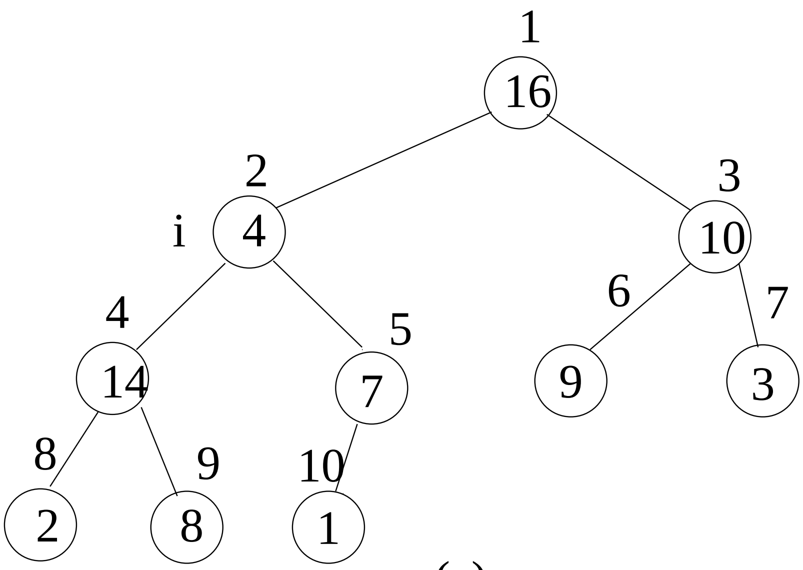
- A[PARENT(i)] $\geq$ A[i]
  für alle Knoten ausser der Wurzel

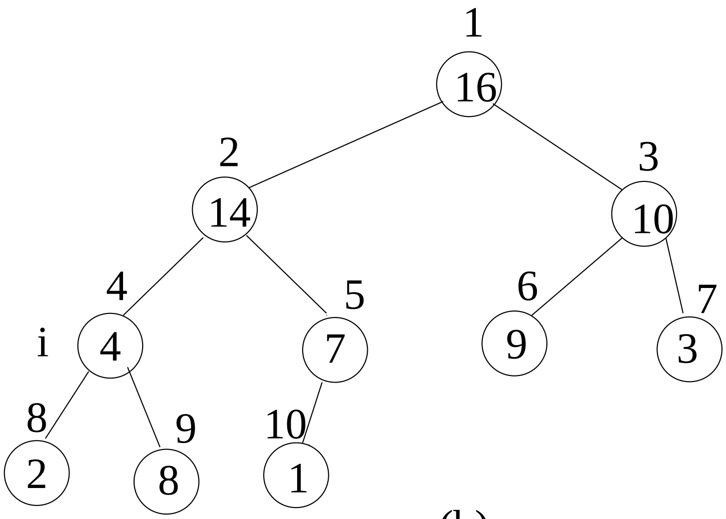
Idee:

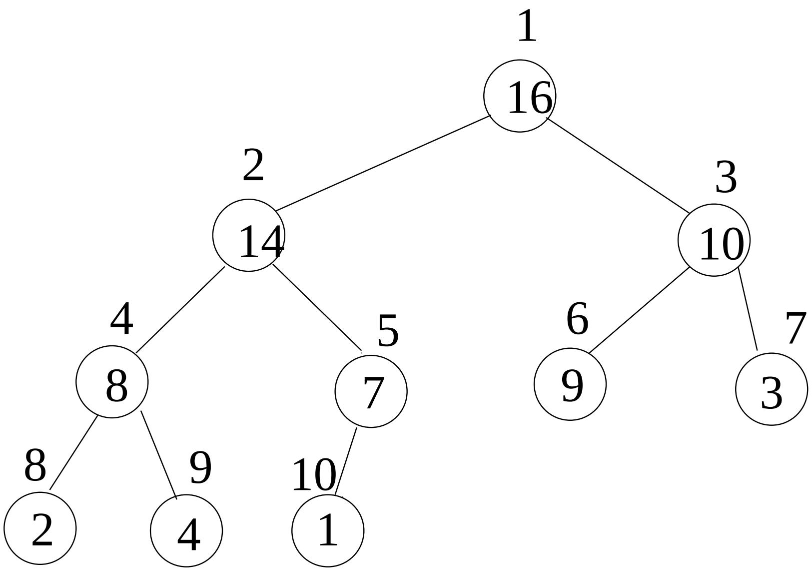
- Eingabe: ein i
- Annahme: Left(i) und Right(i) sind Heaps
- Aber: A[i] kann kleiner als Sohn sein.
- Maßnahme: lasse A[i] nach unten sinken.
- Dann: Teilbaum von i ist Heap

HEAPIFY

```
HEAPIFY (A, i)
l ← LEFT (i)
r ← RIGHT (i)
if l ≤ heap-size[A] and A[l] > A[i]
    then largest ← l
    else largest ← i
if r ≤ heap-size[A] and A[r] > A[largest]
    then largest ← r
if largest ≠ i
    then exchange A[i] ⟷ A[largest]
         HEAPIFY (A, largest)
```


1
16
2
3
i
4
10
4
6
7
5
14
7
9
3
8
9
10
2
8
1
(a)


1
16
2
3
14
10
4
5
6
7
i
4
7
9
3
8
9
10
2
8
1
(b)



(c)

HEAPIFY Beispiel

(a) $i = 2$ verletzt Heap-Eigenschaft, da $A[2]$ nicht größer als beide Söhne.

(b) Heap-Eigenschaft für $i = 2$ wird durch vertauschen des Inhalts von Knoten 2 und 4 wiederhergestellt.

Aber: Heap-Eigenschaft von $i = 4$ ist dadurch verletzt. Daher rekursiver Aufruf.

(c) Vertauschen der Inhalte der Knoten 4 und 9 stellt die Heap-Eigenschaft für $i = 4$ wieder her.

HEAPIFY Analyse

Laufzeit von HEAPIFY für Teilbaum der größe n bei Knoten i:

- max **A[i], A[LEFT(i)], A[RIGHT(i)]** suchen: $\Theta(1)$
- rekursiver Aufruf auf **einem** Unterbaum von i: maximale Größe eines solchen Unterbaums: $2n/3$
  - unterste Ebene ganz voll: $n/2$
  - unterste Ebene nur halb voll: $\frac{2}{3}n$

Es ergibt sich also für die Laufzeit folgende Rekurrenz:

$$T(n) \leq T(2n/3) + \Theta(1)$$

Die Lösung ist

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

Oder in Termini der Höhe: T(h) = O(h)

Rekurrenzen

Wie bekommt man die Lösung einer Rekurrenz?

Methoden:

1. Substitutionsmethode
2. Iterationsmethode
3. Master Theorem

Bemerkungen:

- Für obige Rekurrenz verwenden wir das Master Theorem.
- Wenn anwendbar, Master Theorem am einfachsten.

Master Theorem

Seien

- $a \geq 1$ und $b > 1$ Konstanten,
- $f(n)$ eine Funktion,
- $T(n)$ definiert für die natürlichen Zahlen durch

$$T(n) = aT(n/b) + f(n)$$

wobei n/b interpretiert wird als $\lceil n/b \rceil$ oder $\lfloor n/b \rfloor$.

Dann ergeben sich für $T(n)$ folgende asymptotischen Grenzen:

1. Falls
$$f(n) = O(n^{\log_b a - \epsilon})$$
für ein $\epsilon > 0$
dann
$$T(n) = \Theta(n^{\log_b a})$$

2. Falls
$$f(n) = \Theta(n^{\log_b a})$$
dann
$$T(n) = \Theta(n^{log_b a} \lg n)$$

3. Falls
$$f(n) = \Omega(n^{\log_b a + \epsilon})$$
für ein $\epsilon > 0$ und
$$af(n/b) \leq cf(n)$$
für ein $c < 1$ und genügend große n
dann
$$T(n) = \Theta(f(n))$$

Master Theorem Anwendung von 2

Für HEAPIFY:

$$T(n) \leq T(2n/3) + \Theta(1)$$

Fall 2 des Master Theorems:

(2) Falls

$$f(n) = \Theta(n^{\log_b a})$$

dann

$$T(n) = \Theta(n^{log_b a} \lg n)$$

Zu zeigen:

$$\Theta(1) = \Theta(n^{\log_{2/3}(1)})$$

das ist richtig, da

$$n^{\log_{2/3}(1)} = n^0 = 1$$

Also

$$T(n) = O(n^{\log_{2/3}(1)} \lg n) = O(\lg n)$$

q.e.d.

BUILD-HEAP

- BUILD-HEAP baut einen Heap aus einen Array
- BUILD-HEAP verwendet HEAPIFY
  - von den Blättern aufwärts zur Wurzel
- In $A[(\lfloor n/2 \rfloor + 1) \ldots n]$ sind nur Blätter.
- Blätter sind Bäume aus einem Knoten und diese sind bereits Heaps.
- Anwendung von HEAPIFY auf $A[1 \ldots \lfloor n/2 \rfloor]$ von rechts nach links.
- Letzteres garantiert für i
  - die linken und rechten Teilbäume von i sind bereits Heaps bevor HEAPIFY auf i angewendet wird.

BUILD-HEAP

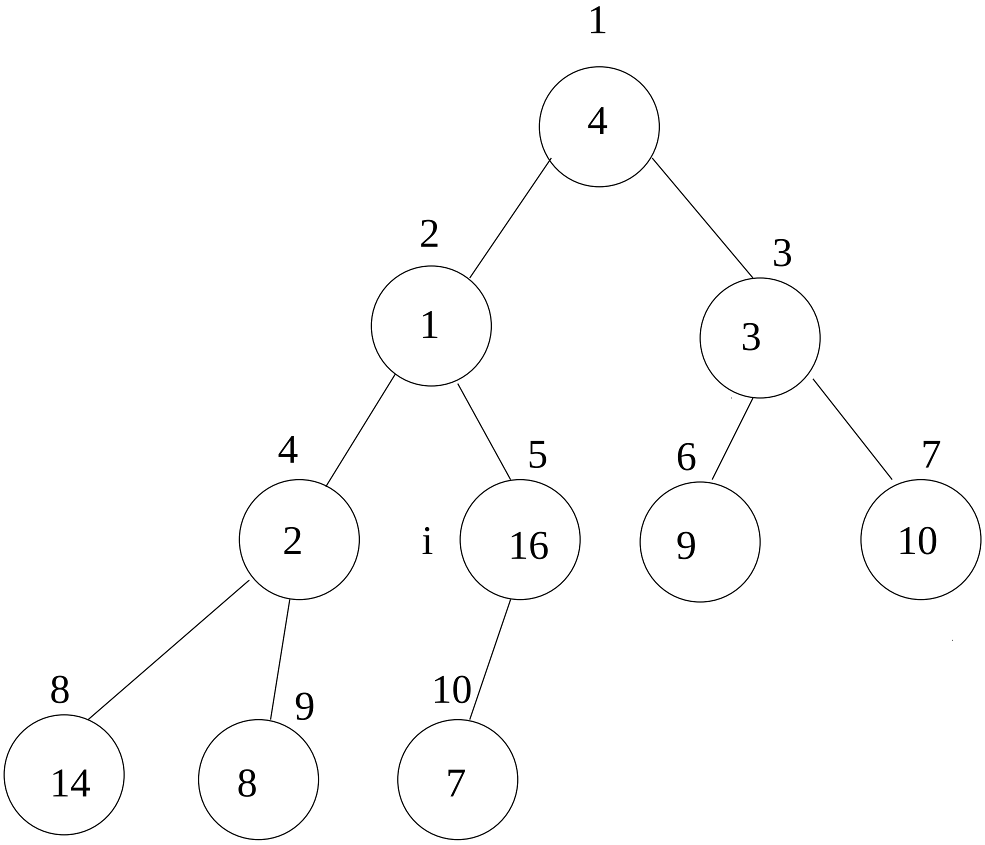
BUILD-HEAP(**A**)

heap-size$[A] \leftarrow length[A]$

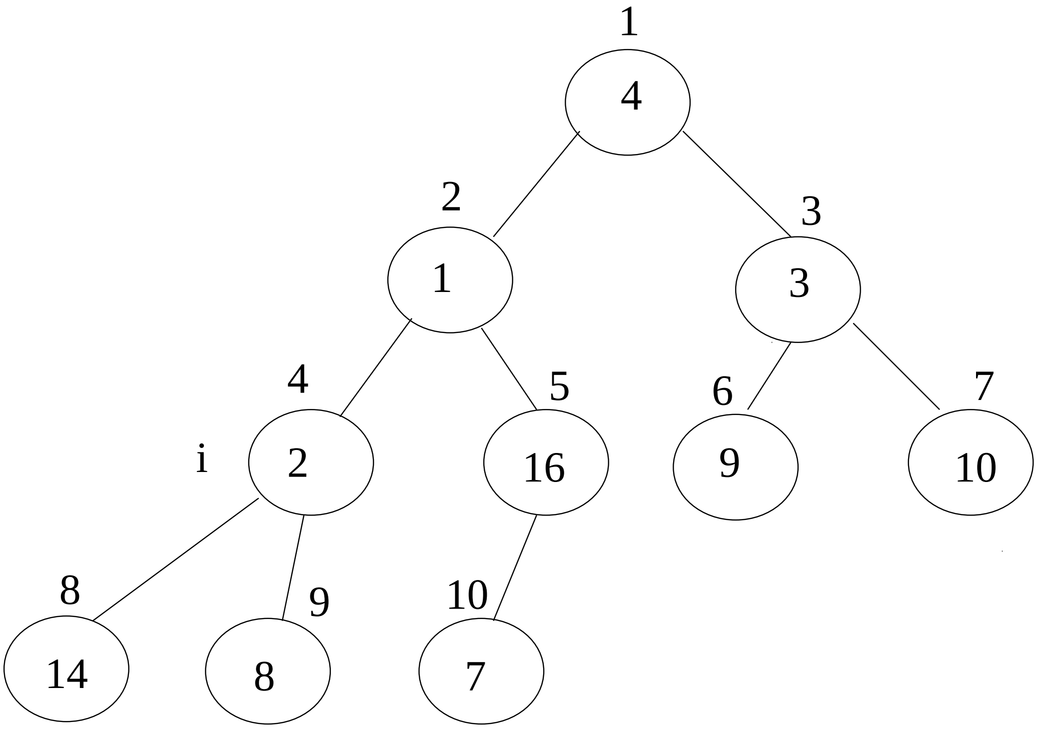
for $i \leftarrow \lfloor$ **length**$[A]/2\rfloor$ **downto** 1

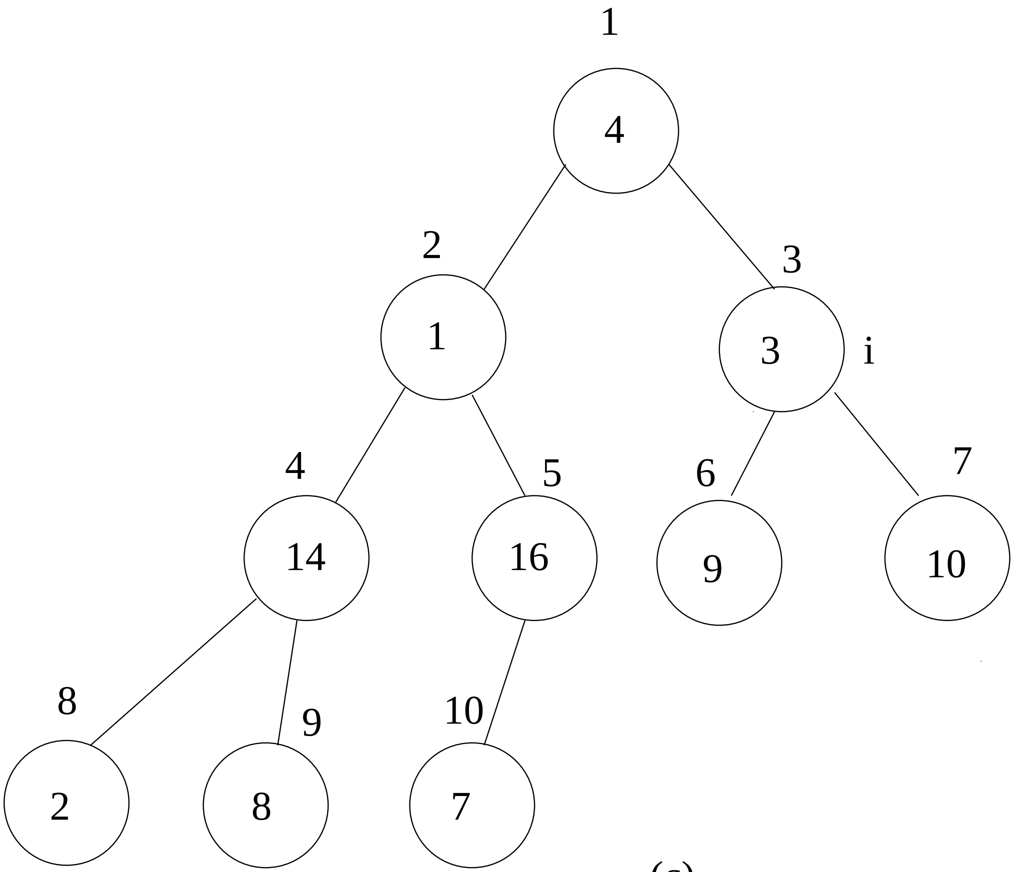
  do HEAPIFY(A, i)

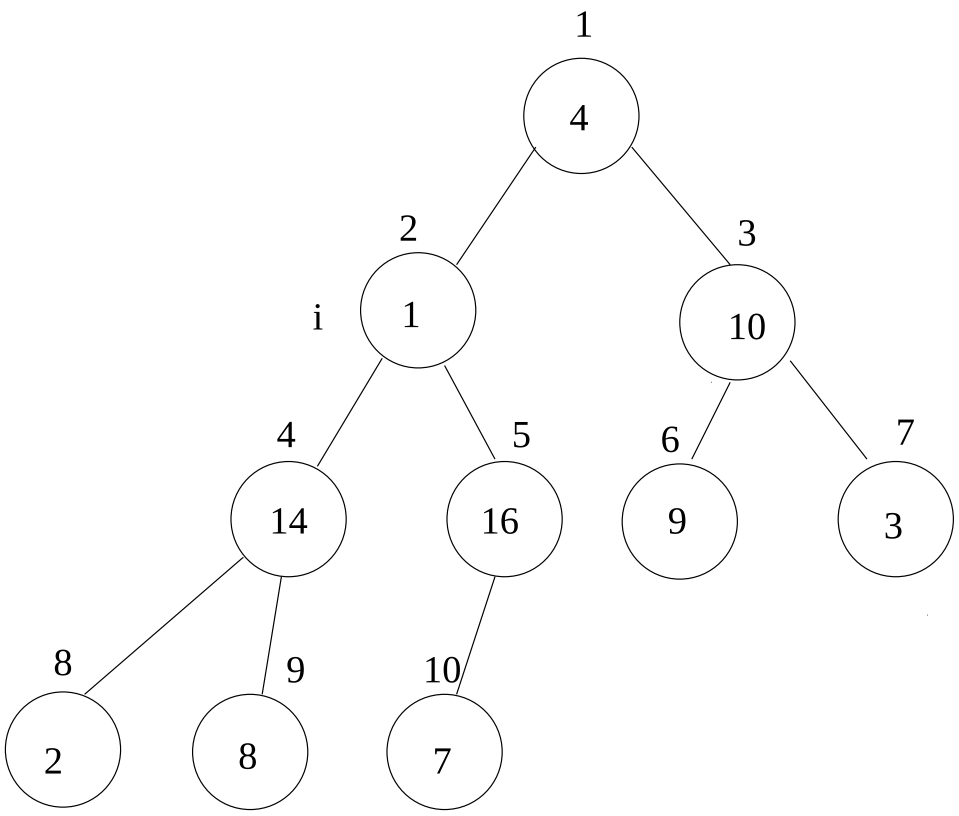
A										
	4	1	3	2	16	9	10	14	8	7



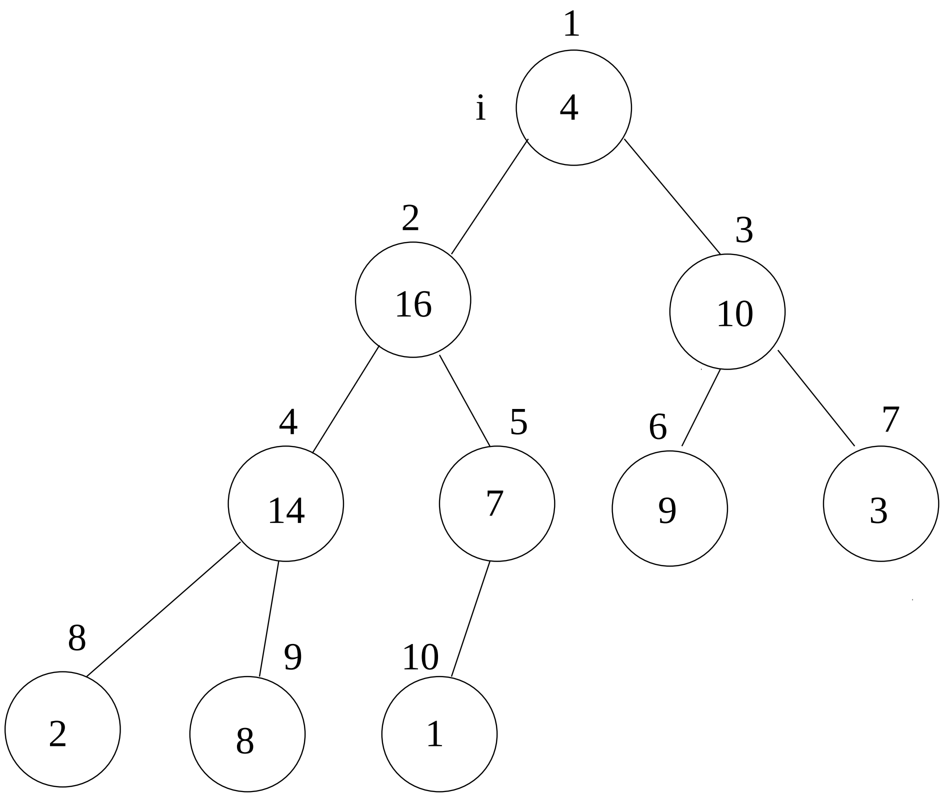
(a)


1
4
2
1
3
3
4
5
6
7
i
2
16
9
10
8
9
10
14
8
7
(b)

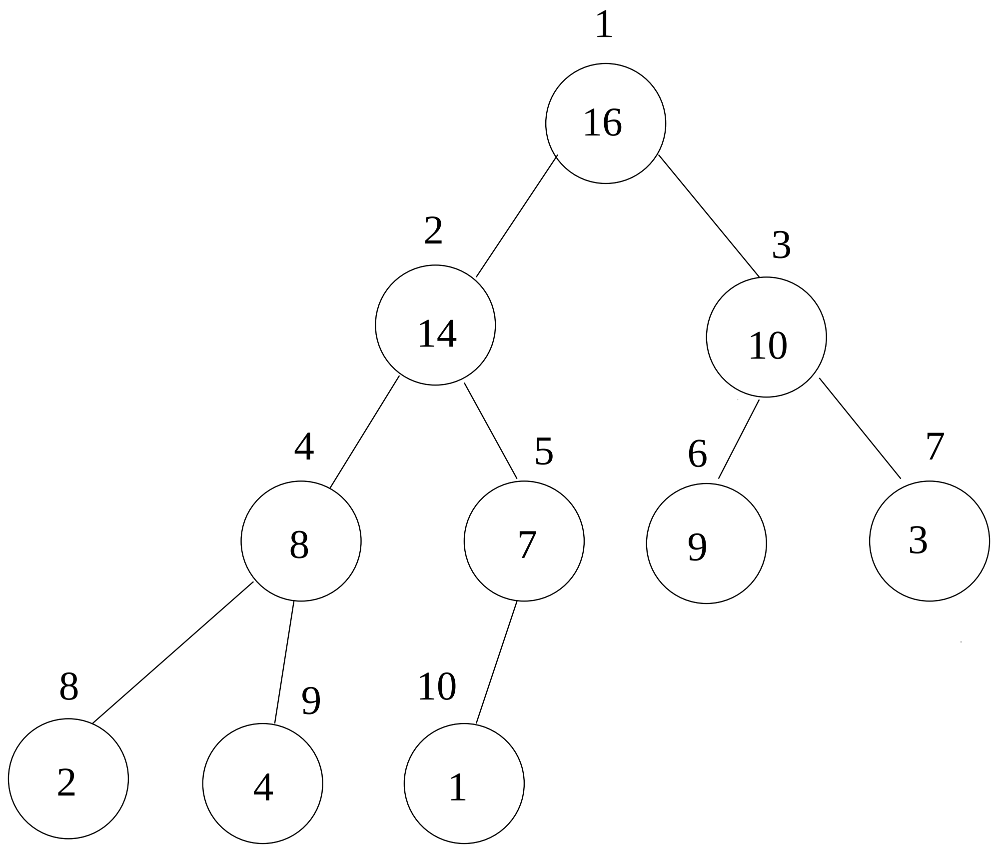

1
4
2
3
1
3
i
4
5
6
7
14
16
9
10
8
9
10
2
8
7
(c)



(d)



(e)



(f)

BUILD-HEAP Analyse

Eine einfache obere Schranke:

- Jeder Aufruf von HEAPIFY kostet $O(\lg n)$
- Es gibt $O(n)$ solche Aufrufe
- Laufzeit: $O(n \lg n)$

Dieses ist zwar eine korrekte obere Schranke, jedoch keine enge.

Also suchen wir eine bessere.

Knackpunkt:

- Laufzeit von HEAPIFY variiert mit der Höhe

BUILD-HEAP Analyse

Eigenschaft eines Heaps:

- In einem n-elementigen Heap gibt es höchstens

  $$\lceil n/2^{h+1} \rceil$$

  Knoten der Höhe h.
  (s. Übung)

BUILD-HEAP Analyse

HEAPIFY braucht für Knoten der Höhe h $O(n)$ Zeit.

Es ergeben sich also folgende Kosten für BUILD-HEAP:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h})$$

Zwischenrechnung

Es gilt (3.6):

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

mit x=1/2 erhalten wir

$$\sum_{h=0}^{\infty} hx^h = \frac{1/2}{(1-1/2)^2} = 2$$

BUILD-HEAP Analyse

Aus
$$\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h})$$
ergibt sich also
$$\begin{aligned} O(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}) &= O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) \\ &= O(n) \end{aligned}$$

Also:

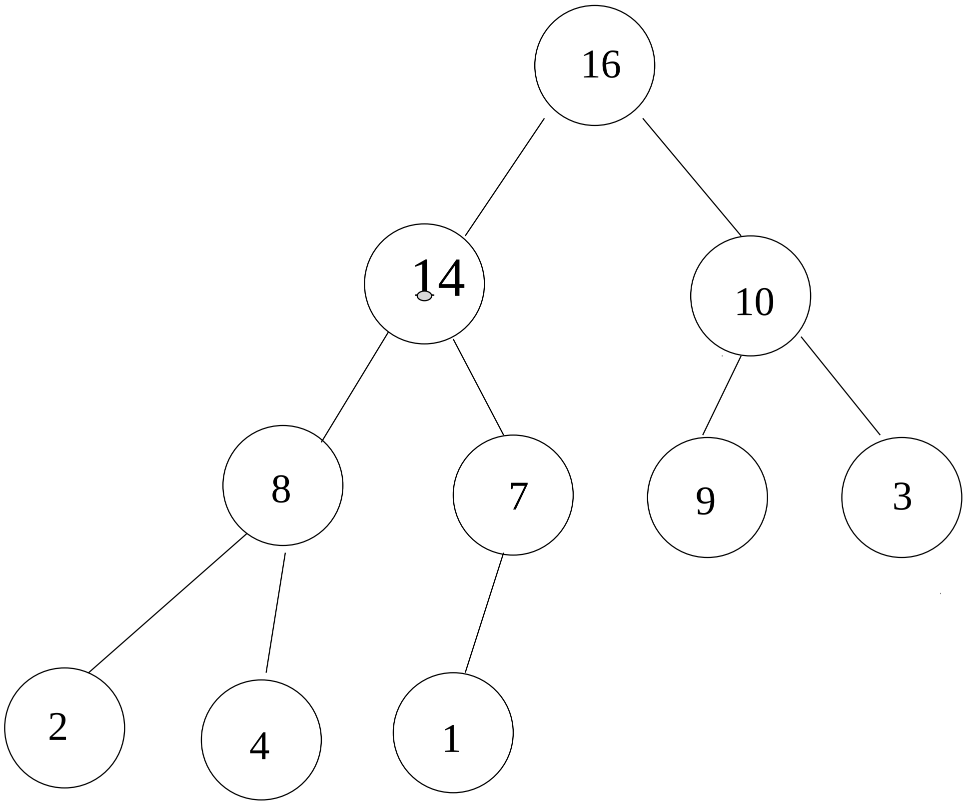
- BUILD-HEAP läuft in $O(n)$

HEAP-SORT

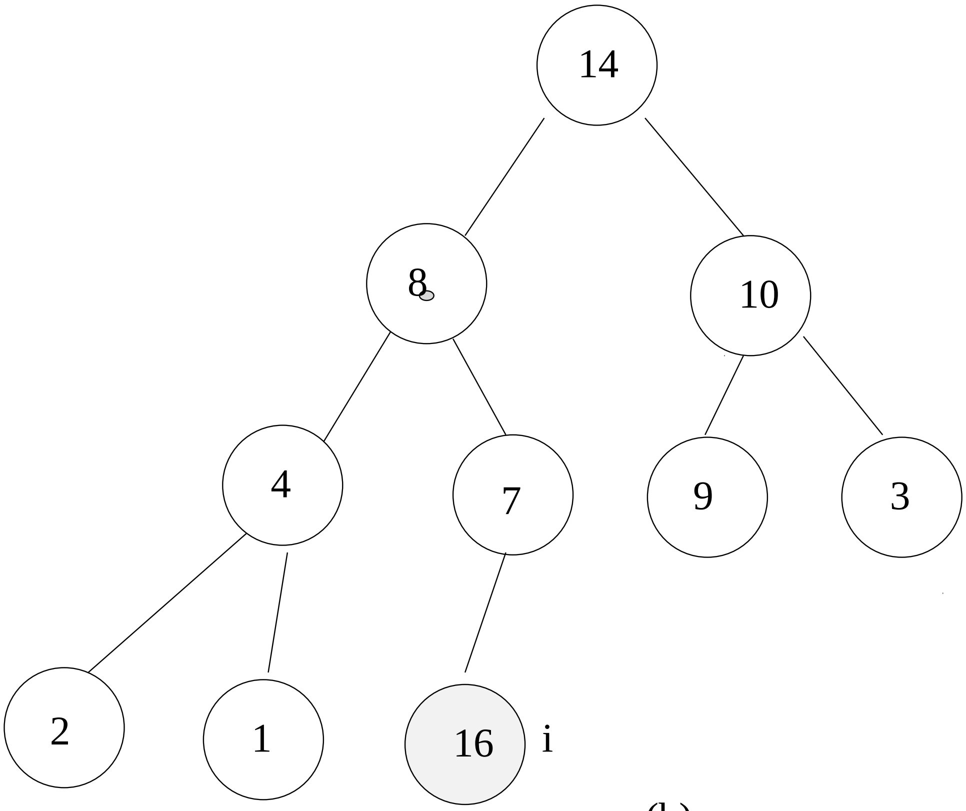
1. Wir bauen zunächst einen Heap.
2. Damit steht das größte Element an der Wurzel.
3. Tausche Wurzel ($= A[1]$) mit letztem Element in Heap (**A[heap-size]**).
4. Wiederherstellen der Heap-Eigenschaft.

HEAP-SORT

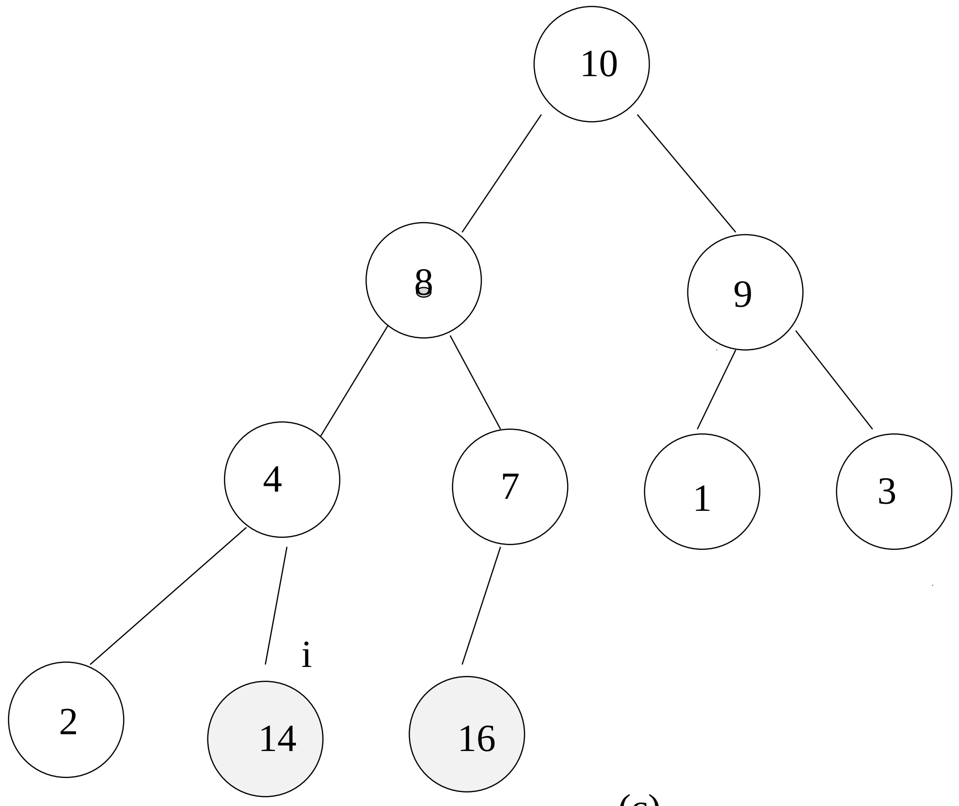
```
HEAPSORT( A)
    BUILD-HEAP( A)
    for i ← length[A] downto 2
        do exchange A[1] ⟷ A[i]
           heap-size[A] ← heap-size[A] − 1
           HEAPIFY( A,1)
```



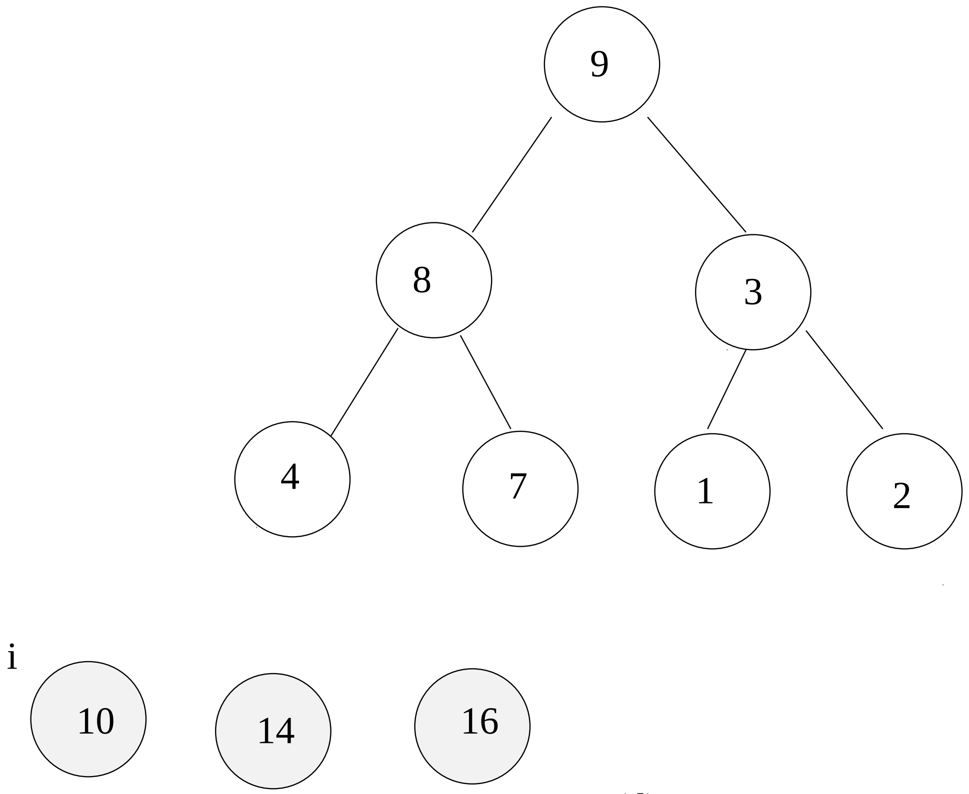
(a)



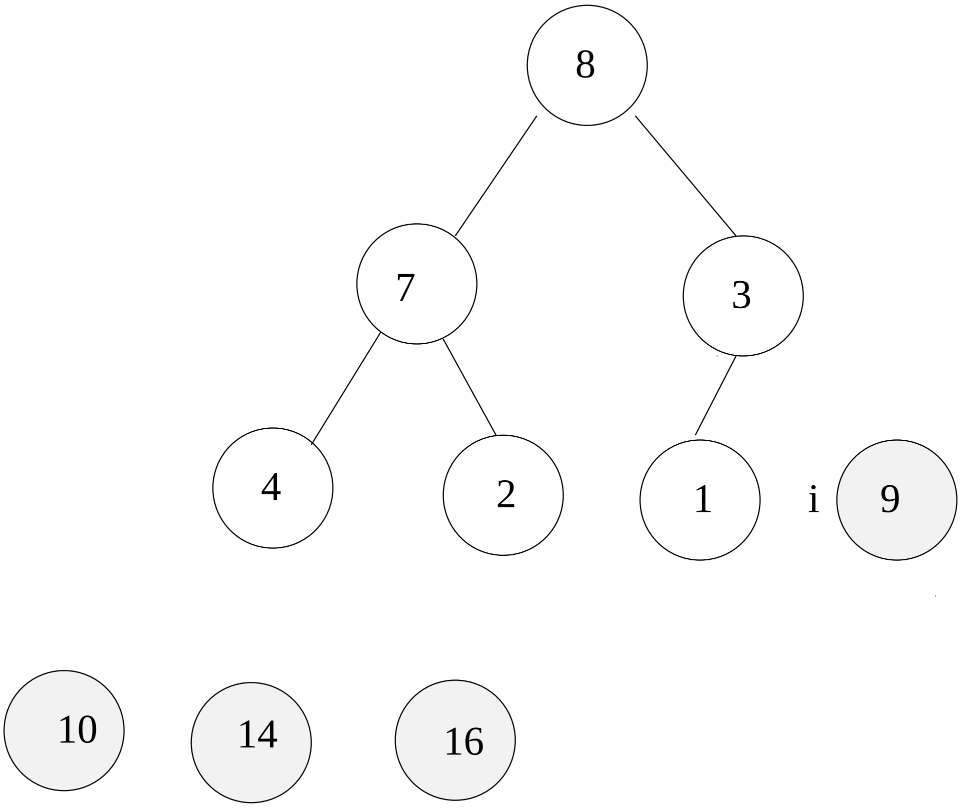
(b)



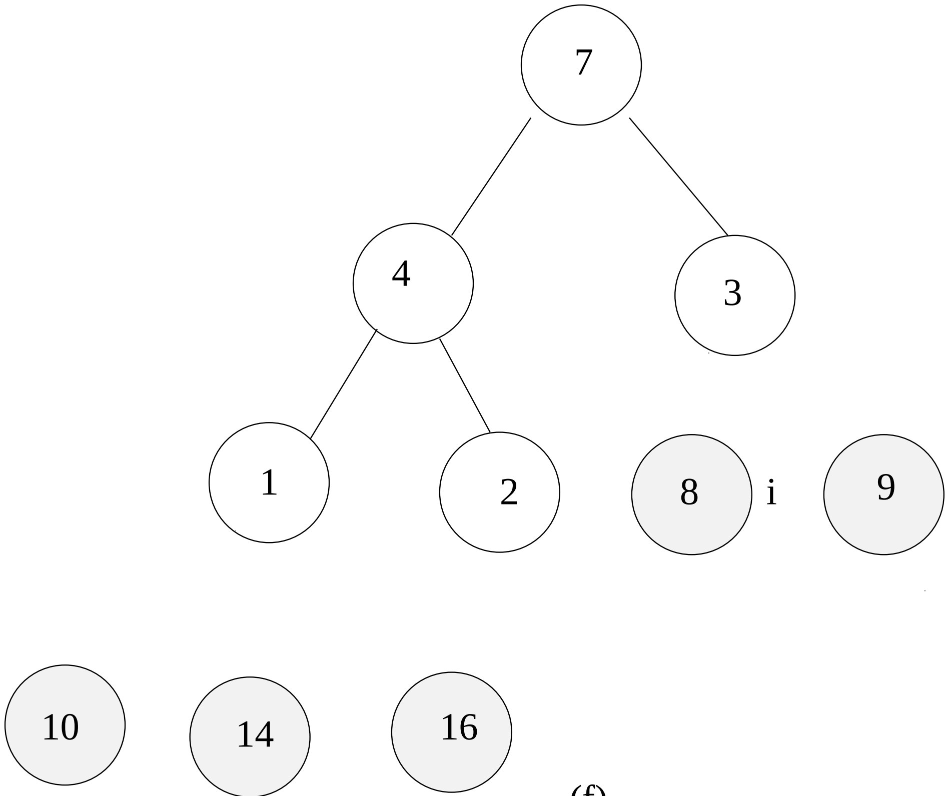
(c)



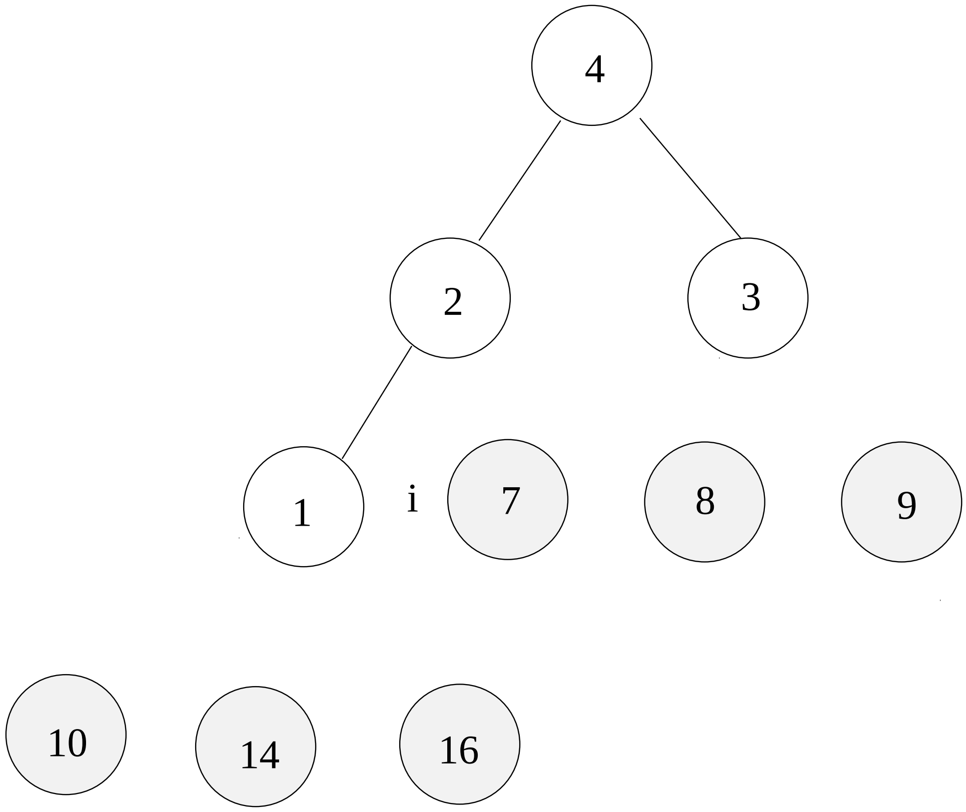
(d)



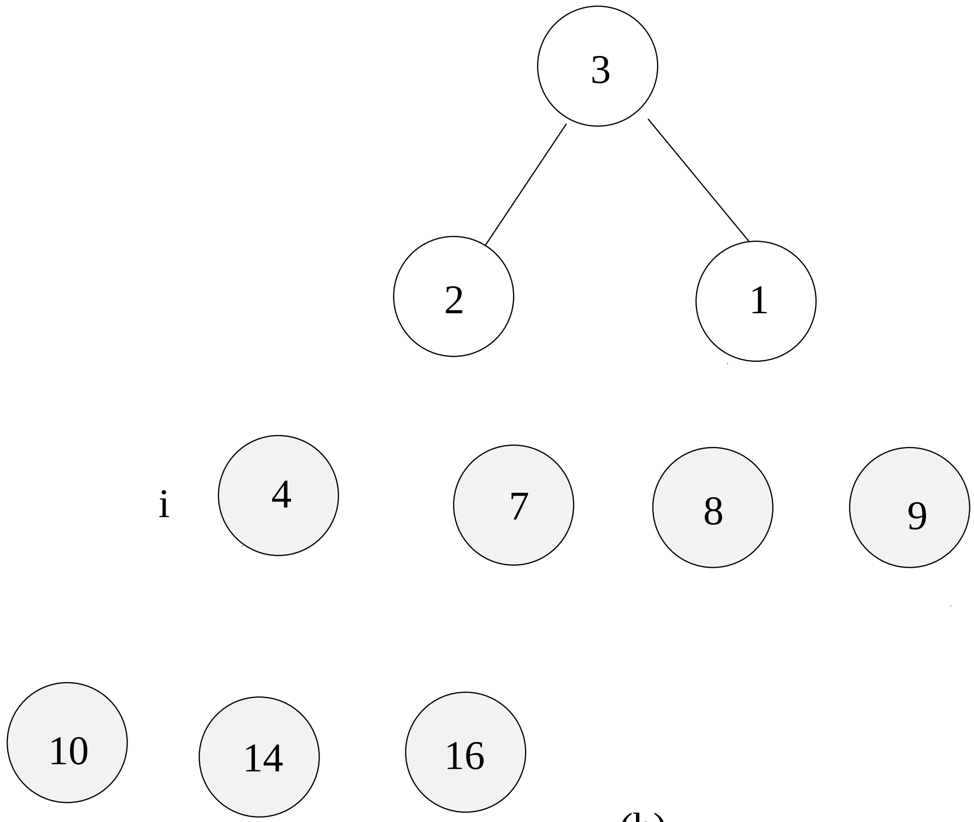
(e)



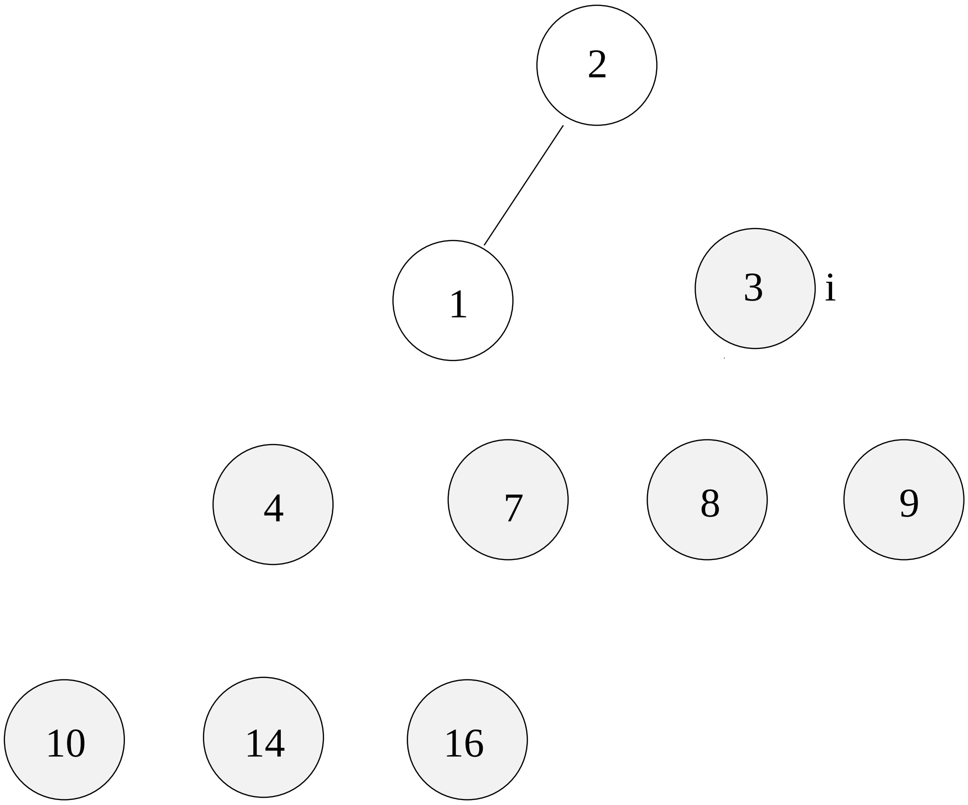
(f)



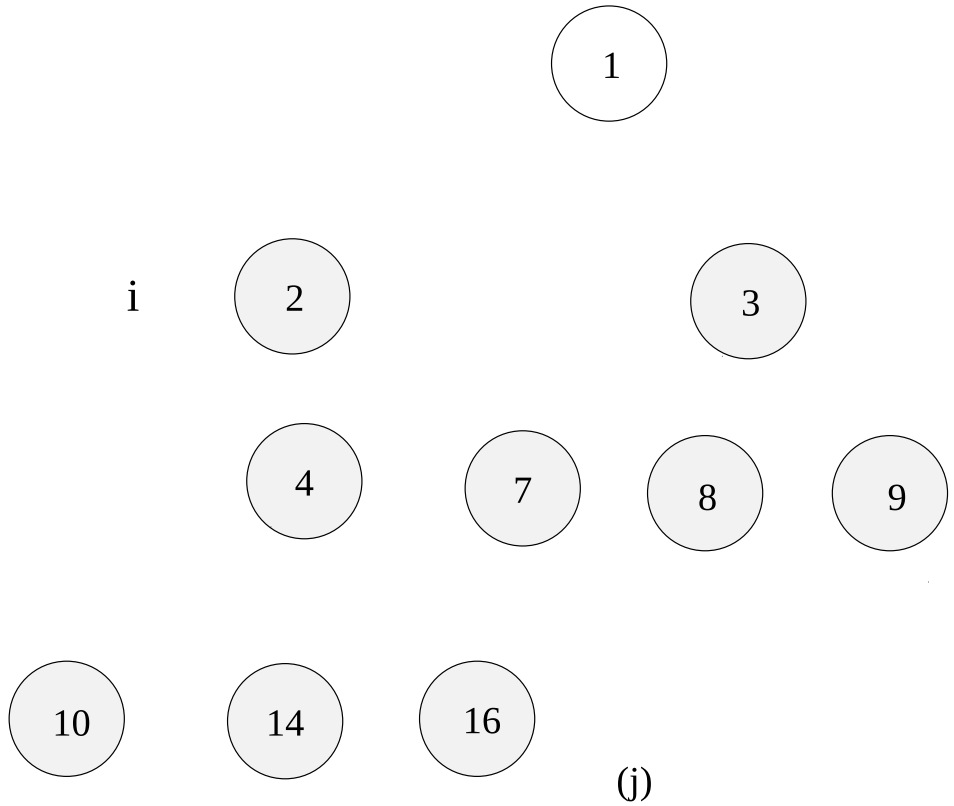
(g)



(h)



(i)



(j)

A	1	2	3	4	7	8	9	10	14	16

(k)

HEAP-SORT Analyse

HEAP-SORT benötigt

$$O(n \lg n)$$

Zeit, da

- Aufruf von BUILD-HEAP $O(n)$ braucht
- jeder der $n-1$ Aufrufe von HEAPIFY $O(\lg n)$ braucht

QUICK-SORT wird schlechteres worst-case Verhalten haben, ist aber in der Praxis meistens schneller als HEAP-SORT.

Ausflug: Prioritätsschlangen

- Der Heap ist eine nützliche Datenstruktur.
- Verwendet z.B. für Prioritätsschlangen.
- Die Datenstruktur der Prioritätsschlangen braucht man z.B. für
  - Job Scheduling
    key = Priorität
  - Simulatoren
    key = Zeitstempel

Anmerkungen

- Heap ist eine Datenstruktur
- Prioritätsschlangen sind eine Datenstruktur
- Wir bauen Prioritätsschlangen mit Heaps
- Ergo: man kann Datenstrukturen mit anderen Datenstrukturen bauen.

Prioritätsschlange

Eine Prioritätsschlange unterstützt folgende Operationen:

- INSERT(S,x)
  fügt ein Element x in Menge S ein
  $S \leftarrow S \cup \{x\}$
- MAXIMUM(S)
  gibt das Maximum der Elemente von S zurück
- EXTRACT-MAX(S)
  gibt das Maximum von S zurück und entfernt es aus S.

Dual: MINIMUM(S) und EXTRACT-MIN(S).

Implementierung mit Heaps.

HEAP-MAXIMUM

HEAP-MAXIMUM(A)
  return A[1]

Laufzeit: $O(1)$

HEAP-EXTRACT-MAX

```
HEAP-EXTRACT-MAX(A)
if heap-size[A] < 1
    then error "heap underflow"
max ← A[1]
A[1] ← A[heap-size[A]]
heap-size[A] ← heap-size[A] − 1
HEAPIFY(A,1)
return max
```

Laufzeit: $O(\lg n)$

HEAP-INSERT

Idee:

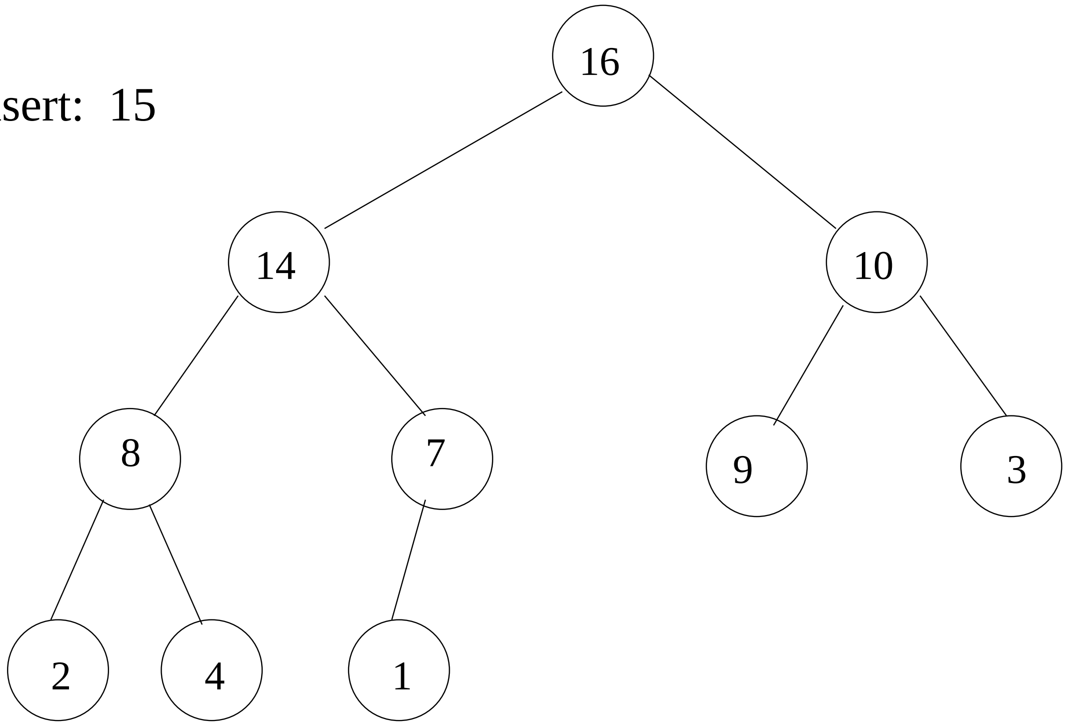
1. Einfügen des neuen Elementes ans Ende des Arrays
2. Dann wie INSERTION-SORT (l 5-7) auf dem Pfad bis zur Wurzel richtigen Platz suchen

HEAP-INSERT

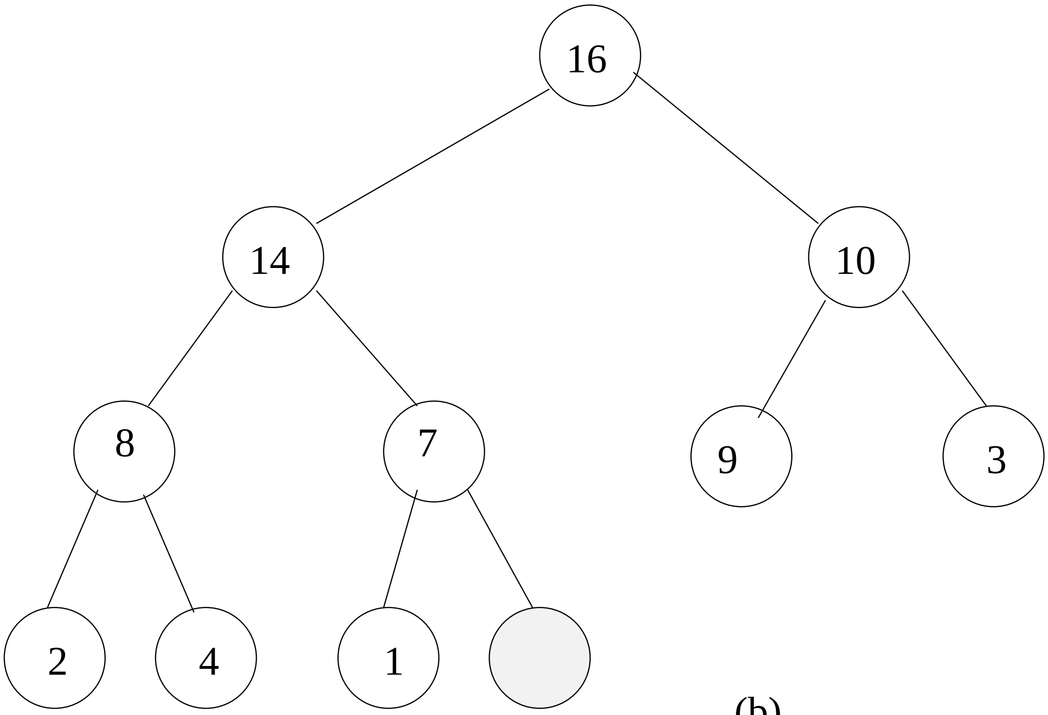
```
HEAP-INSERT(A, key)
    heap-size[A] ← heap-size[A] + 1
    i ← heap-size[A]
    while i > 1 and A[PARENT(i)] < key
        do A[i] ← A[PARENT(i)]
           i ← PARENT(i)
    A[i] ← key
```

Laufzeit: $O(\lg n)$
(Pfadlänge)

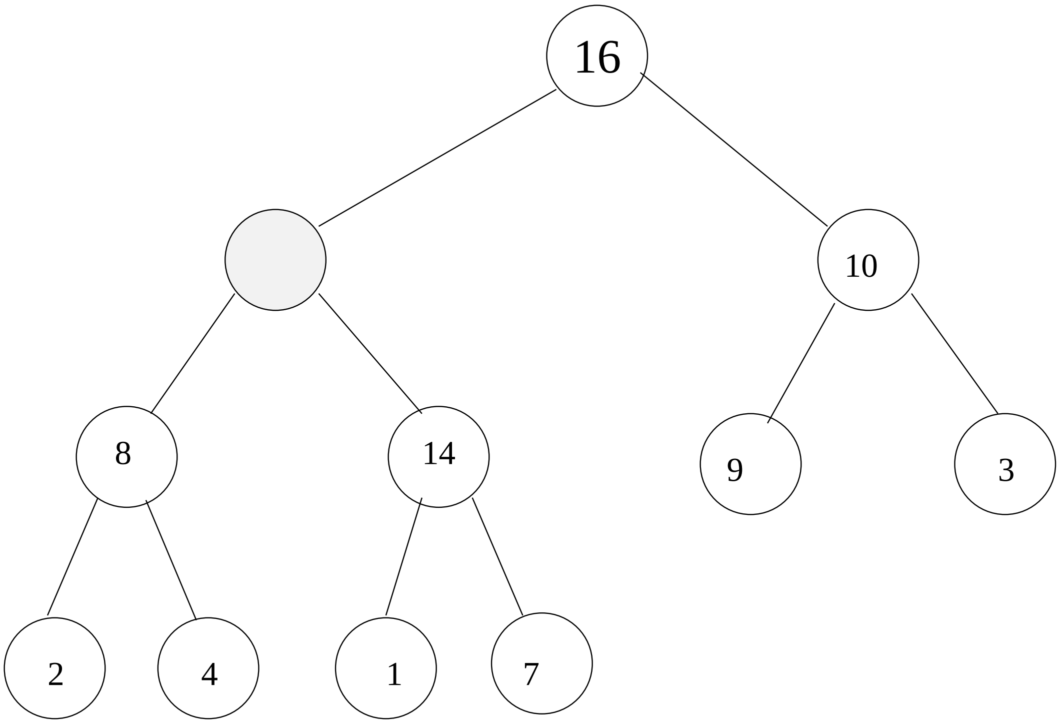
insert: 15



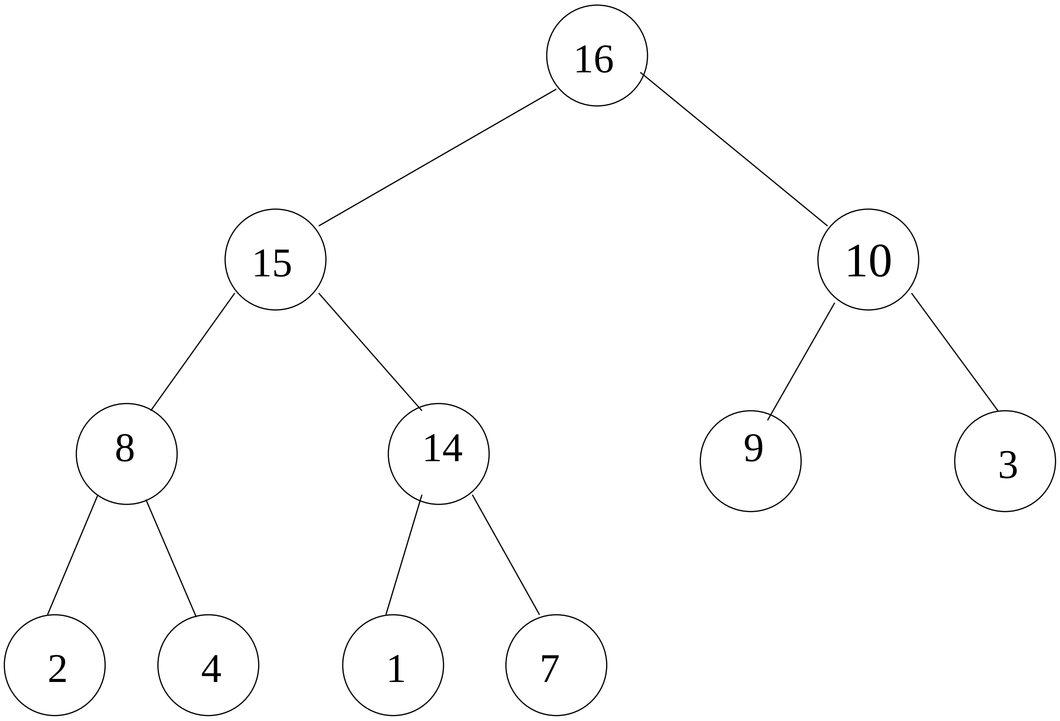
(a)



(b)



(c)



(d)

Prioritätsschlangen auf HEAP-Basis

Zusammenfassend:

- Jede Operation einer Prioritätsschlange kann in

  $$O(\lg n)$$

  mit einem Heap realisiert werden.

Quicksort

Eigenschaften von Quicksort:

- divide-and-conquer-Algorithmus
- worst-case: $\Theta(n^2)$
- avg-case: $\Theta(n \lg n)$
- versteckten Konstanten in $\Theta(n \lg n)$ sind sehr klein
- sortiert in-place

Idee:

- teile nach Elementgrößen

(MERGE-SORT teilt Array nach Lage (Mitte).)

Quicksort

Divide-and-conquer-Schritte um $A[p \ldots r]$ zu sortieren:

Divide $A[p \ldots r]$ wird umgeordnet in zwei nichtleere Teilarrays $A[p \ldots q]$ und $A[q+1 \ldots r]$, so daß jedes Element von $A[p \ldots q]$ kleiner ist als jedes Element von $A[q+1 \ldots r]$.

q wird von der Umordnungsprozedur mitberechnet

Conquer Die Teilarrays $A[p \ldots q]$ und $A[q+1 \ldots r]$ werden rekursiv durch Quicksort sortiert.

Combine Da in-place sortiert wird, ist in diesem Schritt keine Arbeit notwendig.

QUICKSORT

```
QUICKSORT(A, p, r)
if p < r
   then q ← PARTITION(A, p, r)
        QUICKSORT(A, p, q)
        QUICKSORT(A, q + 1, r)
```

Ein Aufruf QUICKSORT(A,1,length[A]) sortiert das ganze Array A.

PARTITION

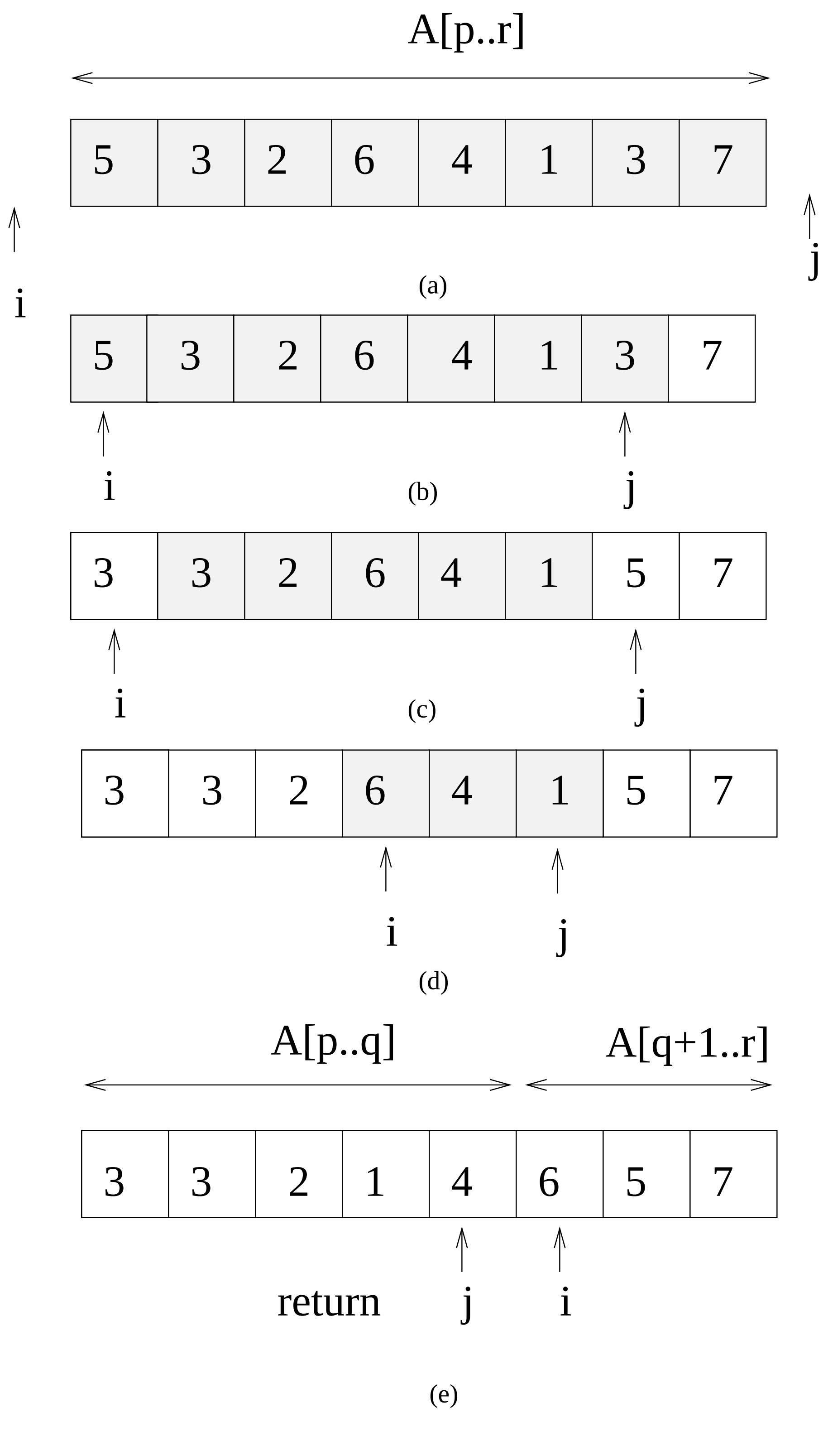
Idee:

- suche ein Pivot-Element x
- alle Elemente kleiner als x kommen dann in $A[p \dots q]$ und alle Elemente größer als x kommen in $A[q + 1 \dots r]$.
- dazu wird A von links und rechts gleichzeitig durchlaufen.
- indizes: i von links und j von rechts
- durchlaufen, bis man zwei Elemente $A[i]$ und $A[j]$ findet mit
  - $A[i] \geq x$
  - $A[j] \leq x$

  diese werden dann vertauscht
- wenn sich i und j treffen ist man fertig.

PARTITION

```
PARTITION( A,p,r)
x ← A[p]
i ← p − 1
j ← r + 1
while TRUE
   do repeat j ← j − 1
         until A[j] ≤ x
      repeat i ← i + 1
         until A[i] ≥ x
      if i < j
         then exchange A[i] ⟷ A[j]
         else return j
```



A[p..r]
5 3 2 6 4 1 3 7
i
j
(a)
5 3 2 6 4 1 3 7
i
j
(b)
3 3 2 6 4 1 5 7
i
j
(c)
3 3 2 6 4 1 5 7
i
j
(d)
A[p..q]
A[q+1..r]
3 3 2 1 4 6 5 7
return
j
i
(e)

PARTITION

- konzeptionell einfach:
  schiebe alle Elemente $\geq x$ in den unteren Teil von A und alle $\leq x$ in den oberen Teil von A.
- Pseudocode ist aber trickreich:
  - i und j bleiben immer in $A[p \dots r]$.
    (kein range-Fehler) nicht offensichtlich im Pseudocode.
  - wichtig: $A[p]$ wird benutzt.
    falls $A[r]$ benutzt und $A[r]$ ist Maximum: Endlosschleife in QUICKSORT, da PARTITION $q = r$ zurückgibt.
- LAUFZEIT: $\Theta(n)$, falls $n = r - p + 1$.

QUICKSORT Analyse

Feststellung:

- Laufzeit abhängig von Balancierung (Verhältnis der Größen beider Partitionen):
  - balanciert: asymptotische Laufzeit $O(n \lg n)$
  - unbalanciert: asymptotische Laufzeit $O(n^2)$

QUICKSORT worst-case

Behauptung:

- worst-case, falls eine Region $n - 1$ Elemente, die andere 1 Element beinhaltet

Annahme:

- Dies geschieht in jedem Schritt

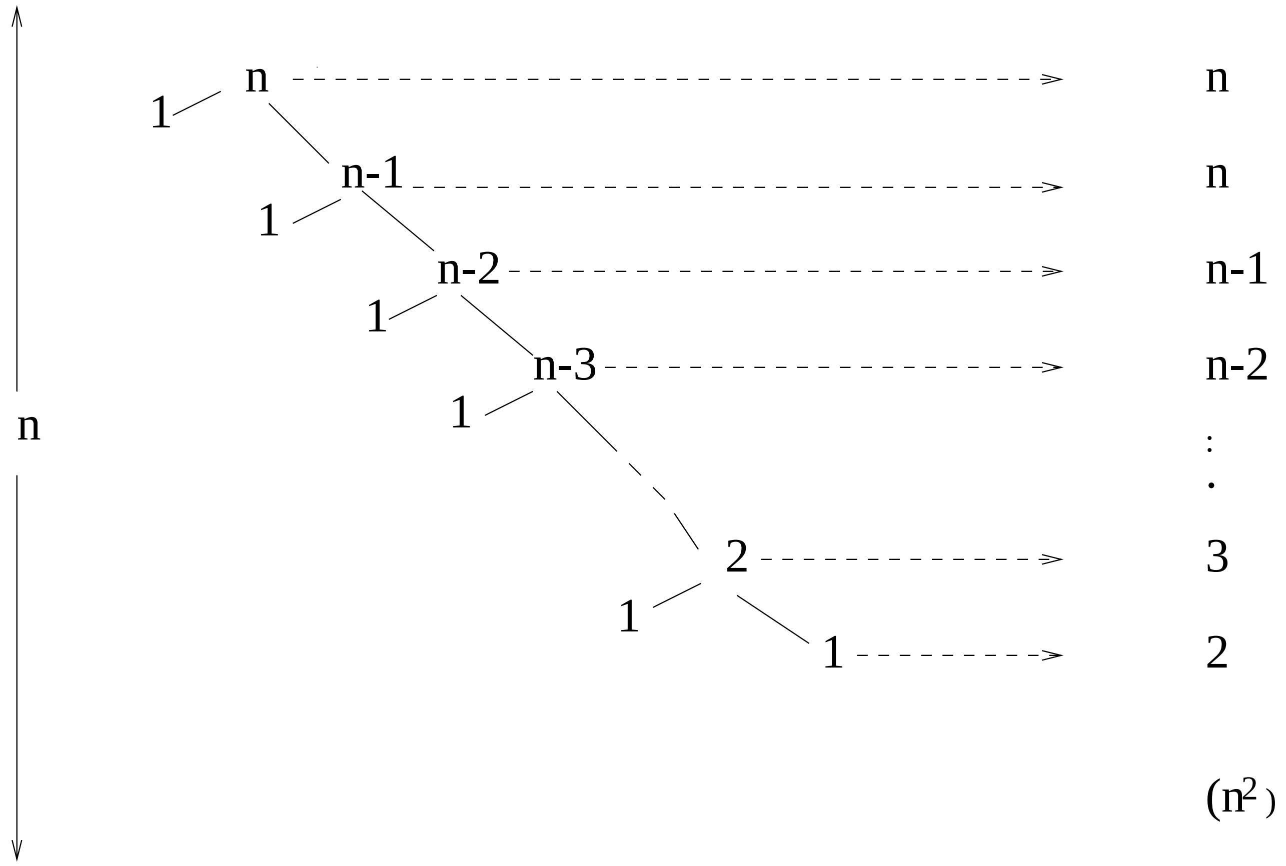
Laufzeitkosten:

- Partitionierungen: $\Theta(n)$ und $\Theta(1)$.

Rekurrenz:

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

Rekursionsbaum:



n
1
n
n-1
1
n-2
1
n-3
1
2
1
1
n
n
n-1
n-2
:
.
3
2
(n2)

QUICKSORT worst-case

- Wir lösen diese Rekurrenz durch Iteration.
- Dazu benötigen wir Beobachtung: $T(1) = \Theta(1)$

$$
\begin{aligned}
T(n) &= T(n-1) + \Theta(n) \\
&= \sum_{k=1}^{n} \Theta(k) \\
&= \Theta(\sum_{k=1}^{n} k) \\
&= \Theta(n^2)
\end{aligned}
$$

QUICKSORT worst-case

Anmerkungen:

- worst-case tritt auf, falls A schon sortiert
- in diesem Fall braucht INSERTION-SORT $\Theta(n)$

QUICKSORT best-case

Behauptung:

- best-case, falls beide Partitionen $n/2$ groß sind

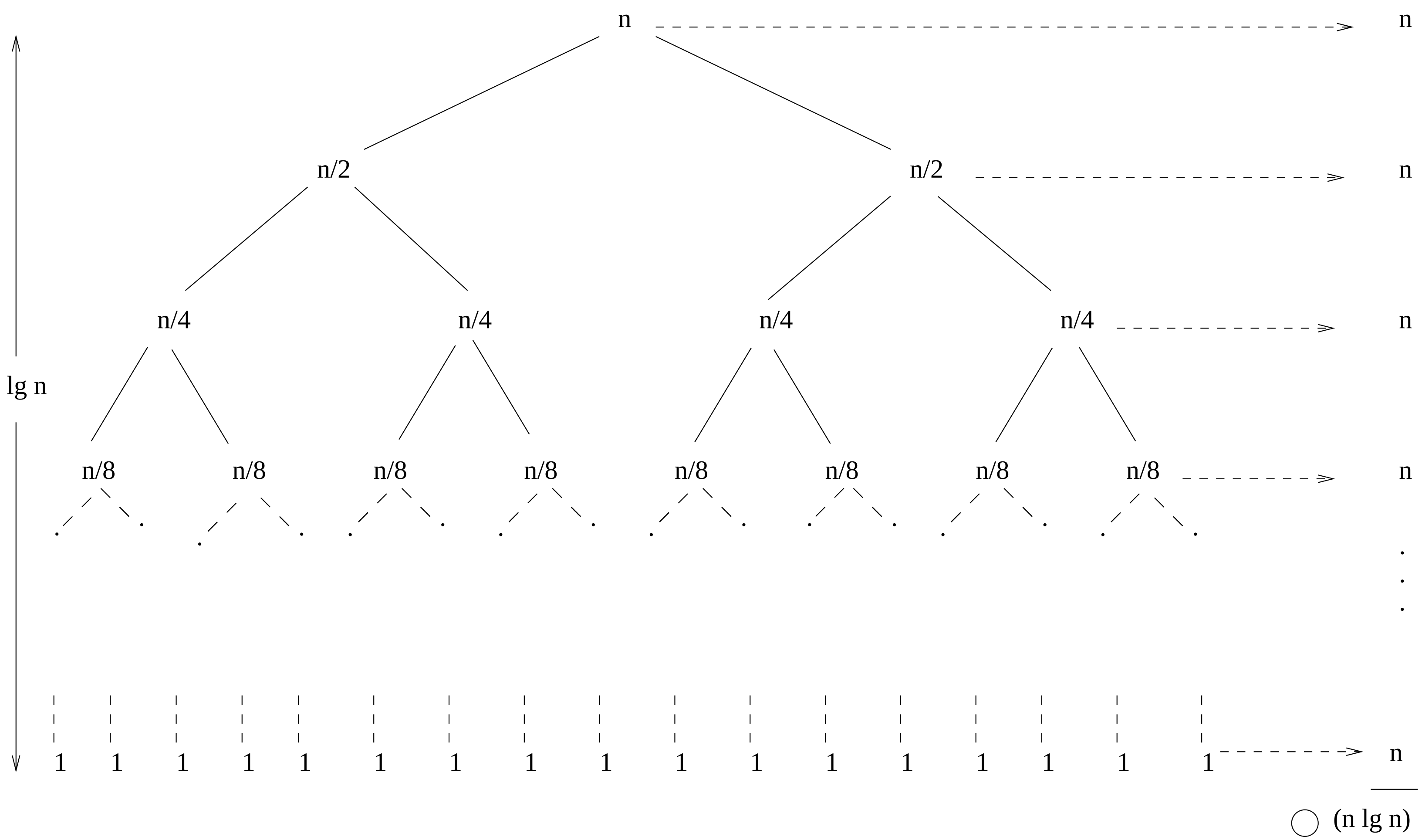
Annahme:

- Dieser Fall kommt bei jedem Aufruf vor

Rekurrenz:

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

Rekursionsbaum:



n
n/2
n/2
n/4
n/4
n/4
n/4
n/8
n/8
n/8
n/8
n/8
n/8
n/8
n/8
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
lg n
n
n
n
n
n
Θ (n lg n)

QUICKSORT best-case

Lösen durch Fall 2 des Master-Theorems: Seien

- $a \geq 1$ und $b > 1$ Konstanten,
- $f(n)$ eine Funktion,
- $T(n)$ definiert für die natürlichen Zahlen durch

  $$T(n) = aT(n/b) + f(n)$$

  wobei n/b interpretiert wird als $\lceil n/b \rceil$ oder $\lfloor n/b \rfloor$.

(2) Falls

$$f(n) = \Theta(n^{\log_b a})$$

dann

$$T(n) = \Theta(n^{log_b a} \lg n)$$

QUICKSORT best-case

Mit $a = b = 2$:

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

- Also ist QUICKSORT hier viel schneller.
- Also ist eine “geschickte” Partitionierung sehr wichtig.
- Aber: avg-case nah an best-case und nicht an worst-case

Einfluß Partitionierung auf Rekurrenz

Annahme:

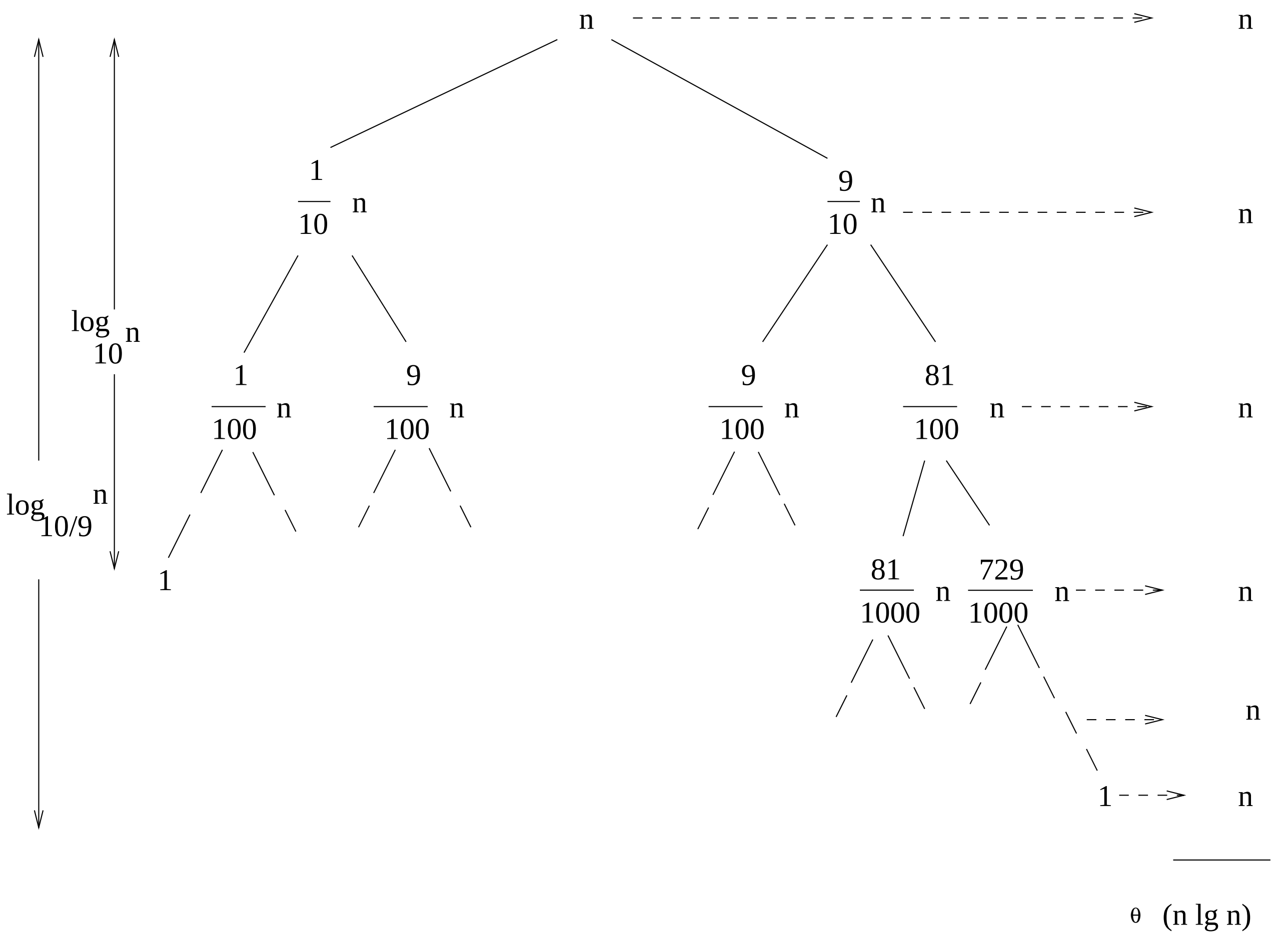
- Partition partitioniert im Verhältnis 9:1

Rekurrenz:

$$T(n) = T(9n/10) + T(n/10) + n$$

Rekursionsbaum:

x


n
n
1/10 n
9/10 n
n
log_10 n
1/100 n
9/100 n
9/100 n
81/100 n
n
log_10/9 n
1
81/1000 n
729/1000 n
n
n
1
n
θ (n lg n)

Einfluß Partitionierung auf Rekurrenz

Mit

- $\log_{10/9} n = \Theta(\lg n)$

haben wir

- Laufzeit immer noch $\Theta(n \lg n)$

Ähnlich:

- 99/1
- 999/1

Allgemein:

- Falls Verhältnis durch Konstante beschränkt ist, so ist die Laufzeit von QUICKSORT $\Theta(n \lg n)$

Intuition avg-case

Normalerweise:

- in einem Rekursionsbaum gibt es:
  - gute Partitionierungen und
  - schlechte Partitionierungen

Wie sind diese verteilt?

Ausflug Gleichverteilung

Was ist Gleichverteilung (von n Zahlen im Bereich $[1, k]$)?

- Wenn jede Zahl von $[1, k]$ gleich häufig vorkommt.

Wie kommt Gleichverteilung zustande?

- Durch Würfeln.

Was ist keine Gleichverteilung?

- Telefonbucheintragungen: Meier häufiger als Moerkotte

Intuition avg-case

- Annahme: Eingaben gleichverteilt
- Dann:
  - mehr als 80% besser als 9:1
  - weniger als 20% schlechter als 9:1
- Gute und schlechte Partitionierungen zufällig auf Rekursionsbaum verteilt.
- vereinfachende Annahme:
  - gute und schlechte Partitionierungen abwechselnd.
  - gute sind 1:1 und schlechte n-1:1.

  Basiskosten für $n = 1$ seien 1.

Intuition avg-case

Die Folge von einer guten und einer schlechten Partitionierung liefert

- 3 Teilarrays der Größen 1, $(n-1)/2$ und $(n-1)/2$
- mit Kosten $2n-1=\Theta(n)$
- Also: dieser Fall ist "fast" der optimale Fall.
- Nur: Konstanten für Partitionierung werden größer

QUICKSORT: weiteres Vorgehen

- RANDOMIZED-QUICKSORT
  damit wir nicht in eine Falle laufen
  (alles oder weite strecken schon sortiert)
- Analyse von RANDOMIZED-QUICKSORT

Randomized Quicksort

Beobachtung (Ex 13.4-4):

- Für jede Konstante $k > 0$ gilt:
  - Für alle bis auf $O(1/n^k)$ der $n!$ Permutationen des Eingabearrays hat QUICKSORT die Laufzeit $\Theta(n \lg n)$.

Konsequenz:

- permutiere Zahlen im Array zufällig, da dann Chance auf worst-case sehr gering

Dies ändert nichts an der asymptotischen Laufzeit.

Realisierung:

- Funktion RANDOM(a,b): liefert eine Zahl zwischen a und b.
- die Folge dieser Zahlen sieht zufällig aus.

RANDOMIZED QUICKSORT

RANDOMIZED-PARTITION(A, p, r)

i $\leftarrow$ RANDOM(p,r)

exchange A[p] $\leftrightarrow$ A[i]

return PARTITION(A,p,r)

RANDOMIZED-QUICKSORT(A, p, r)

if $p < r$

  then $q \leftarrow$ RANDOMIZED-PARTITION(A, p, r)

    RANDOMIZED-QUICKSORT(A, p, q)

    RANDOMIZED-QUICKSORT$(A, q + 1, r)$

Randomized Quicksort worst-case

Vorher gesehen:

- worst-case split ergibt $O(n^2)$ Laufzeit
- unter der Annahme, daß dies tatsächlich der schlechteste Fall ist

Jetzt:

- Genauere Analyse

Randomized Quicksort worst-case

Sei $T(n)$ worst-case Laufzeit von RANDOMIZED-QUICKSORT. Dann

$$T(n) = \max_{1 \leq q \leq n-1} (T(q) + T(n-q)) + \Theta(n)$$

wobei q über $1, \ldots, n-1$ läuft, da PARTITION zwei nicht-leere Regionen erzeugt.

Lösung der Rekurrenz:

- Mit der Substitutionsmethode

Wir raten:

$$T(n) \leq cn^2$$

Damit:

$$\begin{aligned} T(n) &= \max_{1\le q\le n-1}(T(q)+T(n-q))+\Theta(n) \\ &\le \max_{1\le q\le n-1}(cq^2+c(n-q)^2)+\Theta(n) \\ &= c * \max_{1\le q\le n-1}(q^2+(n-q)^2)+\Theta(n) \end{aligned}$$

Da 2. Ableitung von $q^2+(n-q)^2$ nach q positiv (Übung):

- Maximum bei 1 oder $n-1$

Beide Werte gleich!

Also:

$$\begin{aligned} &\max_{1\le q\le n-1}(q^2+(n-q)^2) \\ &\qquad = 1^2+(n-1)^2 \\ &\qquad = 1+(n^2-2n+1) \\ &\qquad = n^2-2(n-1) \end{aligned}$$

Also

$$
\begin{aligned}
T(n) &\leq c * n^2 - 2c(n-1) + \Theta(n) \\
&\leq c * n^2
\end{aligned}
$$

falls c so groß, daß $-2c(n-1)$ $+\Theta(n)$ dominiert.

Also:

- worst case $O(n^2)$

gilt auch für QUICKSORT

RANDOMIZED-QUICKSORT avg-case

Vorgehen:

- Analyse von RANDOMIZED-PARTITION
- Erstellen der Rekurrenz
- Lösen der Rekurrenz

RANDOMIZED-PARTITION Analyse

vereinfachende Annahme:

- Alle Elemente sind unterschiedlich.
- Dies macht Analyse einfacher.
- Ergebnis $O(n \lg n)$ avg-case für RANDOMIZED-QUICKSORT gilt aber auch sonst

RANDOMIZED-PARTITION Analyse

Feststellungen:

- Wenn PARTITION aufgerufen wird, hat RANDOMIZED-PARTITION $A[p]$ mit einem zufälligen Element $A[i]$ vertauscht.
- Return-Wert q von PARTITION hängt nur vom **Rang** von $x = A[i]$ ab.
  (Rang(x) = k, falls x die k-t kleinste Zahl ist.
  anders ausgedrückt: es gibt k Elemente, die $\leq x$ sind.
- Falls $A[p \ldots r]$ $n = r - p + 1$ Elemente hat, so ist $rank(x) = i$ $(1 \leq i \leq n)$ mit Wahrscheinlichkeit $1/n$.

Wir berechnen jetzt die Wahrscheinlichkeit, der verschiedenen Partitionierungen (Größen).

$rank(x) = 1$**:** Dann hält PARTITION mit $i = j = p$.
Die linke Partition enthält nur $A[p]$.

Die Wahrscheinlichkeit hierfür ist $1/n$.
(= Wahrscheinlichkeit $rank(x) = 1$)

$rank(x) \geq 2$**:** Dann gibt es mindestens ein Element $\leq x = A[p]$.

Der erste Durchlauf durch die **while**-Schleife hält bei $i = p$, $j > p$.

$A[p]$ wird also in die rechte Partition geschoben.

Wenn PARTITION anhält, ist jedes Element links von x echt kleiner als x.

Also: für $i = 1, \ldots, n-1$, $rank(x) \geq 2$:
linke Partition hat mit Wahrscheinlichkeit $1/n$ Größe i

Zusammenfassend:

Die Größe $q-p+1$ der linken Partition ist 1 mit Wahrscheinlichkeit $2/n$ und i ($i = 2, \ldots, n-1$) mit Wahrscheinlichkeit $1/n$

Erstellen der Rekurrenz

Sei $T(n)$ die avg-case Laufzeit für das Sortieren eines Arrays der Größe n.

Es gilt $T(1) = \Theta(1)$.

PARTITION benötigt $\Theta(n)$ und gibt q zurück.

Rekursives sortieren von Teilarrays der Länge q und $n - q$.

Also $T(n) =:$

$$\frac{1}{n}\left(T(1) + T(n-1) + \sum_{q=1}^{n-1}(T(q) + T(n-q))\right) + \Theta(n)$$

(q ist gleichverteilt, wie oben ausgerechnet, ausser daß $q = 1$ zweimal vorkommt; einmal aus Σ herausgezogen.)

Mit $T(1) = \Theta(1)$, $T(n-1) = O(n^2)$ (aus worst-case):

$$\begin{aligned}\frac{1}{n}(T(1) + T(n-1)) &= \frac{1}{n}(\Theta(1) + O(n^2)) \\ &= O(n)\end{aligned}$$

$\Theta(n)$ absortiert $O(n)$, also:

$$\begin{aligned}T(n) &= \frac{1}{n}\sum_{q=1}^{n-1}(T(q) + T(n-q)) + \Theta(n) \\ &= \frac{2}{n}\sum_{q=1}^{n-1}(T(q)) + \Theta(n) \\ &= \frac{2}{n}\sum_{k=1}^{n-1}(T(k)) + \Theta(n)\end{aligned}$$

Lösen der Rekurrenz: Substitutionsmethode

Annahme: $T(n) \leq an \lg n + b$ für Konst. $a, b > 0$

Weiter: a, b so groß, daß $an \lg n + b > T(1)$

Dann: Für $n > 1$:

$$\begin{aligned} T(n) &= \frac{2}{n} \sum_{k=1}^{n-1} (T(k)) + \Theta(n) \\ &\leq \frac{2}{n} \sum_{k=1}^{n-1} (ak \lg k + b) + \Theta(n) \\ &= \frac{2a}{n} \sum_{k=1}^{n-1} (k \lg k) + \frac{2b}{n}(n-1) + \Theta(n) \end{aligned}$$

Wir zeigen unten:

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

Damit:

$$
\begin{aligned}
T(n) &\le \frac{2a}{n}\sum_{k=1}^{n-1}(k\lg k) + \frac{2b}{n}(n-1) + \Theta(n)\\
&\le \frac{2a}{n}(\frac{1}{2}n^2\lg n - \frac{1}{8}n^2) + \frac{2b}{n}(n-1) + \Theta(n)\\
&\le an\lg n - \frac{a}{4}n + 2b + \Theta(n)\\
&= an\lg n + b + (\Theta(n) + b - \frac{a}{4}n)\\
&\le an\lg n + b
\end{aligned}
$$

dabei a so groß, daß $a/4n$ majorisiert $\Theta(n) + b$.

Also: $T(n) = O(n\lg n)$

zu zeigen:

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

leicht:

$$\sum_{k=1}^{n-1}(k \lg k) \leq n^2 \lg n$$

da jedes $k \lg k \leq n \lg n$.

Dies reicht aber nicht, da wir für die Lösung der Rekurrenz eine Schranke der Form

$$\frac{1}{2}n^2 \lg n - \Omega(n^2)$$

benötigen.

Also berechnen wir strengere Schranke.

TRICK: spalten der Summe.

$$\sum_{k=1}^{n-1}(k \lg k) = \sum_{k=1}^{\lceil n/2 \rceil -1}(k \lg k) + \sum_{k=\lceil n/2 \rceil}^{n-1}(k \lg k)$$

- erste Summe: $\lg k \leq \lg(n/2) = \lg n - 1$.
- zweite Summe: $\lg k \leq \lg n$.

Also:

$$\begin{aligned}
\sum_{k=1}^{n-1}(k \lg k) &\leq (\lg n - 1) \sum_{k=1}^{\lceil n/2 \rceil -1} k + (\lg n) \sum_{k=\lceil n/2 \rceil}^{n-1} k \\
&= \lg n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil -1} k \\
&\leq \frac{1}{2}n(n-1)\lg n - \frac{1}{2}(\frac{n}{2}-1)\frac{n}{2} \\
&\leq \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2
\end{aligned}$$

für $n \geq 2$. qed.

Zusammenfassung

Bis jetzt:

- mehrere Algorithmen in $\Theta(n \lg n)$ worst-case oder avg-case.
- alle Algorithmen basierten auf Vergleichen

Jetzt:

- Beweis einer unteren Schranke für vergleichbasierte Algorithmen
- schnellere Algorithmen, die nicht auf vergleichen basieren

Untere Schranken für Sortieren

Annahme:

- Es werden nur $\leq, <, >, \geq, =$ genutzt um zwei Elemente einer Folge zu sortieren.
- Keine andere Information über die Elemente wird benutzt.

Einschränkung:

- Alle Elemente der Folge sind unterschiedlich.

Daher:

- $<$ reicht aus.

Entscheidungsbaum

Beobachtung:

- Es existieren $n!$ Fakultät Permutationen der Eingabefolge.
- Nur eine ergibt sortierte Folge.

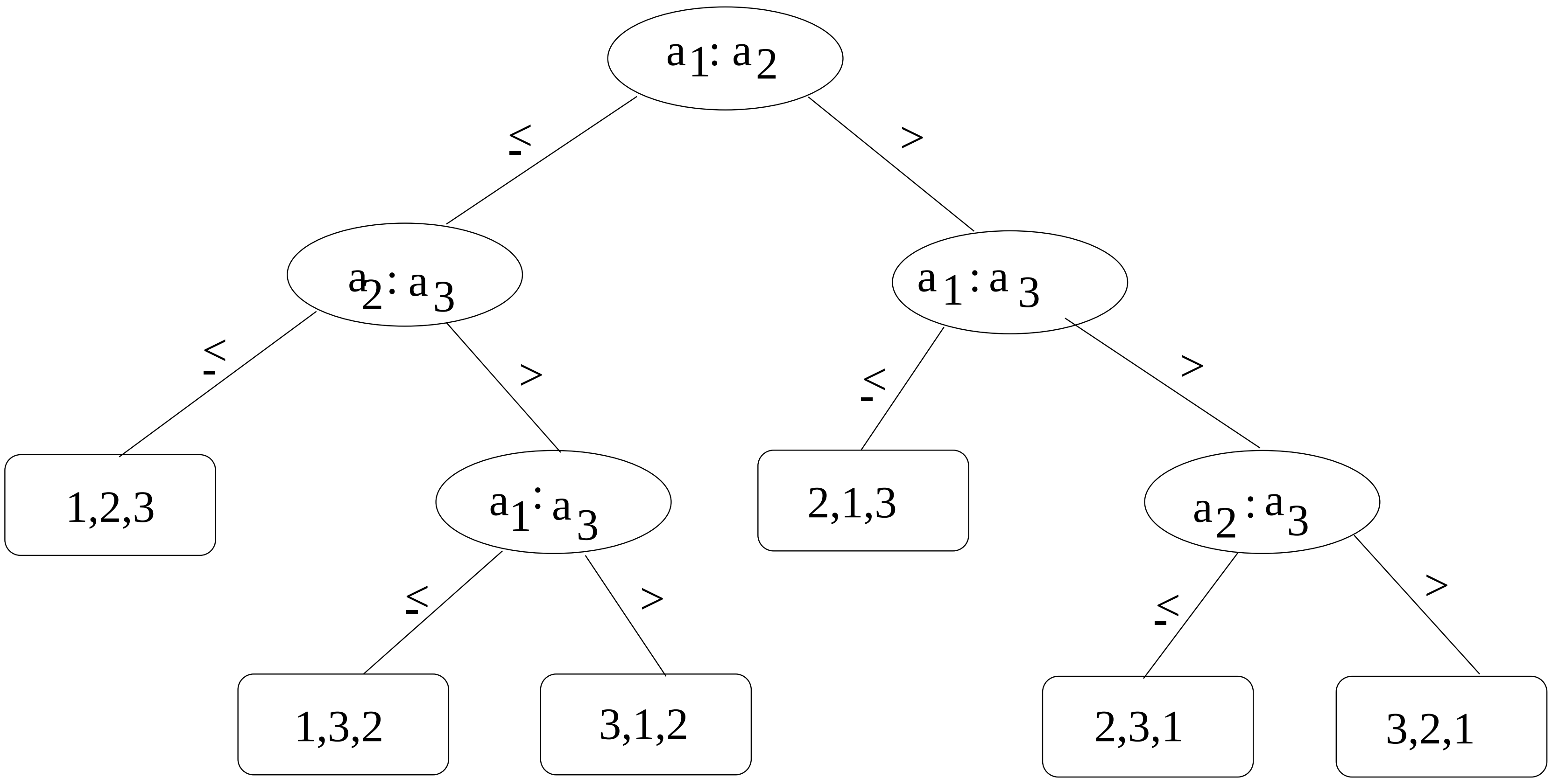
Entscheidungsbaum:

- Jeder interne Knoten ist mit einem $a_i : a_j$ markiert. $1 \leq i, j \leq n$.
- Jedes Blatt ist mit einer Permutation markiert.

Entscheidungsbaum

Zusammenhang Algorithmen:

- nur interessant: Vergleiche
  vernachlässige: Kontrolle, Zuweisungen, usw.
- jeder vom Algorithmus durchgeführter Vergleich entspricht einem Vergleich im Entscheidungsbaum:
  - Wurzel=erster Vergleich
  - Nachfolgeknoten: abhängig vom Ausgang vorheriger Vergleiche.
- Jedes Blatt im Entscheidungsbaum ist genau eine der möglichen Permutationen.
- Alle Permutationen müssen vorkommen.



$a_1 : a_2$
$\leq$
$>$
$a_2 : a_3$
$a_1 : a_3$
$\leq$
$>$
$\leq$
$>$
1,2,3
$a_1 : a_3$
2,1,3
$a_2 : a_3$
$\leq$
$>$
$\leq$
$>$
1,3,2
3,1,2
2,3,1
3,2,1