SS 2021

Algorithmen und Datenstrukturen

6. Kapitel

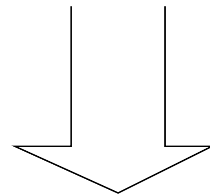
Sortieralgorithmen

Martin Dietzfelbinger

Juni 2021

6.1 Erinnerung, Grundbegriffe

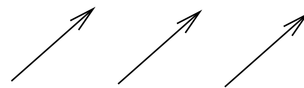
a_1	a_2	a_3		………				a_n
10	1	4	5	1	9	4	1	8
US	SO	RA	OR	RT	HM	LG	IE	IT



Item:

Schlüssel	x	"key"
Daten:	dat	"data"

1	1	1	4	4	5	8	9	10
SO	RT	IE	RA	LG	OR	IT	HM	US



"Stabilität"

Sortierproblem, abstrakt, Spezifikation: Siehe Kapitel 1.

Input: Tupel $x = (a_1, \ldots, a_n)$ von Objekten a_i
mit Schlüssel- und Datenteil: a_i`.key` und a_i`.data`.

Gegeben als **Array** `A`$[1..n]$ oder als **lineare Liste**.

Achtung: In Bildern unterdrücken wir oft den Datenteil. Objekt a_i wird also einfach als Schlüssel dargestellt.

Schon gesehen: Sortieralgorithmus **Insertionsort**.

Bei Vorkommen gleicher Sortierschlüssel: Sortierverfahren heißt **stabil**, wenn Objekte mit identischen Schlüsseln in der Ausgabe in derselben Reihenfolge stehen wie in der Eingabe.

Maßzahlen für Sortieralgorithmen (in Abhängigkeit von n)**:**

1. **Laufzeit** (in O-Notation);
2. **Anzahl** der **Vergleiche** („*wesentliche Vergleiche*“)

$$„a_i\texttt{.key} < a_j\texttt{.key}“ \quad \text{bzw. } „a_i\texttt{.key} \leq a_i\texttt{.key}“;$$

3. Anzahl der Daten**verschiebungen** oder **Kopiervorgänge** (wichtig bei sehr umfangreichen Objekten);
4. der *neben* dem Array (bzw. der linearen Liste) benötigte **Speicherplatz**; wenn dieser nur $O(1)$, also von n unabhängig, ist: Algorithmus arbeitet „**in situ**“ oder „**in-place**“.

Insertionsort: Schlechtester Fall: $\frac{1}{2}n(n-1) \approx \frac{n^2}{2}$ Vergleiche, Zeit $\Theta(n^2)$.
Bester Fall: $n-1$ Vergleiche, Zeit $\Theta(n)$. **Durchschnittlicher** Fall: Zeit $\Theta(n^2)$. **Stabil, in situ.**

6.2 Mergesort

(engl. *merge* = vermengen, reißverschlussartig zusammenfügen)

Algorithmenparadigma oder -**schema**:

„**Divide-and-Conquer**“ (**D-a-C**, „**teile und herrsche**“)

Vorgehen eines D-a-C-Algorithmus $\mathcal{A}$ für Berechnungsproblem $\mathcal{P} = (\mathcal{I}, \mathcal{O}, f)$, auf Instanz $x \in \mathcal{I}$:

0) Trivialitätstest: Wenn $\mathcal{P}$ für x einfach direkt zu lösen ist, tue dies. – Sonst:

1) Teile: Zerteile x in (zwei oder mehr) Stücke $x_1, \ldots, x_a \in \mathcal{I}$.

(Wesentlich für Korrektheit/Terminierung: Jedes der Stücke ist „kleiner“ als x.)

2) Rekursion: Löse $\mathcal{P}$ für $x_1, \ldots, x_a$, separat, durch **rekursive** Verwendung von $\mathcal{A}$.

Teillösungen: $r_1, \ldots, r_a$.

3) Kombiniere: Baue aus x und $r_1, \ldots, r_a$ und $x_1, \ldots, x_a$ eine Lösung r für $\mathcal{P}$ auf x auf.

Mergesort ist D-a-C-Algorithmus für das Sortierproblem. – Input: $x = (a_1, \ldots, a_n)$.

Eine Trivialitätsschranke $n_0 \geq 1$ ist festgelegt.

0) Trivialitätstest: Falls $n \leq n_0$, sortiere x mit einem einfachen Algorithmus (z. B. Insertionsort). – Sonst:

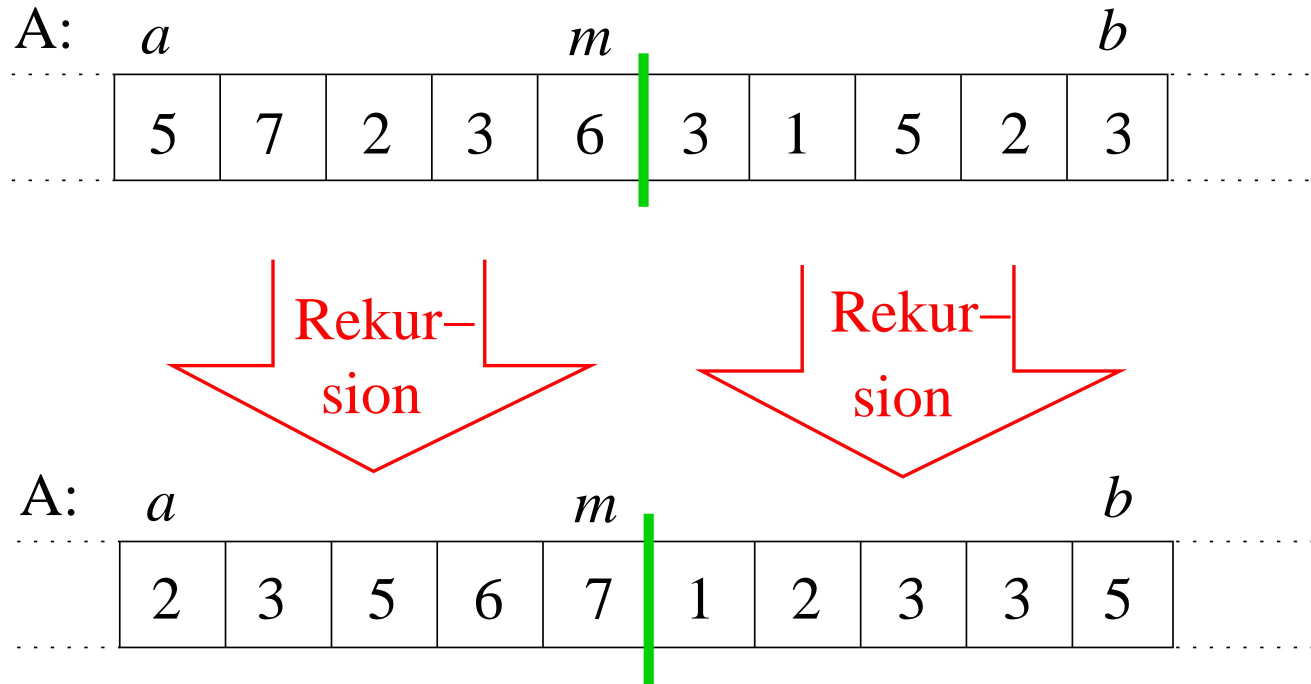
1) Teile: Setze $m := \lceil n/2 \rceil$.
Zerlege x in zwei Teilinputs x_1 (m Einträge) und x_2 ($n - m$ Einträge).

2) Rekursion: Sortiere x_1 **rekursiv**, Ergebnis r_1; sortiere x_2 **rekursiv**, Ergebnis r_2.

3) Kombiniere: „Mische“ die sortierten Folgen r_1 und r_2 zu einer sortierten Folge r zusammen. (Werden sehen: Aufwand ist $\leq n - 1$ Vergleiche.)

Für eine konkrete Formulierung nehmen wir an, dass Ein- und Ausgabe über ein Array $\texttt{A}[1..n]$ erfolgen, und geben eine **rekursive** Prozedur **r_Mergesort** an, die ein durch seine Grenzen a und b ($1 \leq a \leq b \leq n$) gegebenes Teilarray von $\texttt{A}[1..n]$ sortiert.

Rekursive **Prozedur r_Mergesort**(a, b):



1) Aufteilen von A[$a \ldots b$] in A[$a \ldots m$] und A[$m+1 \ldots b$].

2) Rekursives Sortieren der beiden Segmente.

3) **Merge**(a, m, b).

Datenstruktur: Arrays A[1..n] (Ein-/Ausgabe), B[1..n] (Hilfsarray)

```
Prozedur r_Mergesort(a, b)    // 1 ≤ a ≤ b ≤ n, sortiert A[a..b]
(1)   if b − a ≤ n_0                         // n_0 ≥ 0 ist vorausgesetzt
(2)      then Insertionsort(a, b)
(3)      else                     // Wissen: Länge von A[a..b] ist ≥ 2
(4)         m ← a + ⌊(b − a)/2⌋;
(5)         r_Mergesort(a, m);
(6)         r_Mergesort(m + 1, b);
(7)         Merge(a, m, b)  // fügt sortierte Teilfolgen zusammen;
                            // benutzt Hilfsarray B[1..n]
```

```
Prozedur Mergesort(A[1..n])    // sortiert A[1..n]
(1)   r_Mergesort(1, n).
```

Benötigt: Prozedur **Merge**(a, m, b), $1 \leq a \leq m < b \leq n$,
die folgendes tut:

Voraussetzung: $\mathtt{A}[a \mathinner{..} m]$ und $\mathtt{A}[m+1 \mathinner{..} b]$ sind sortiert.

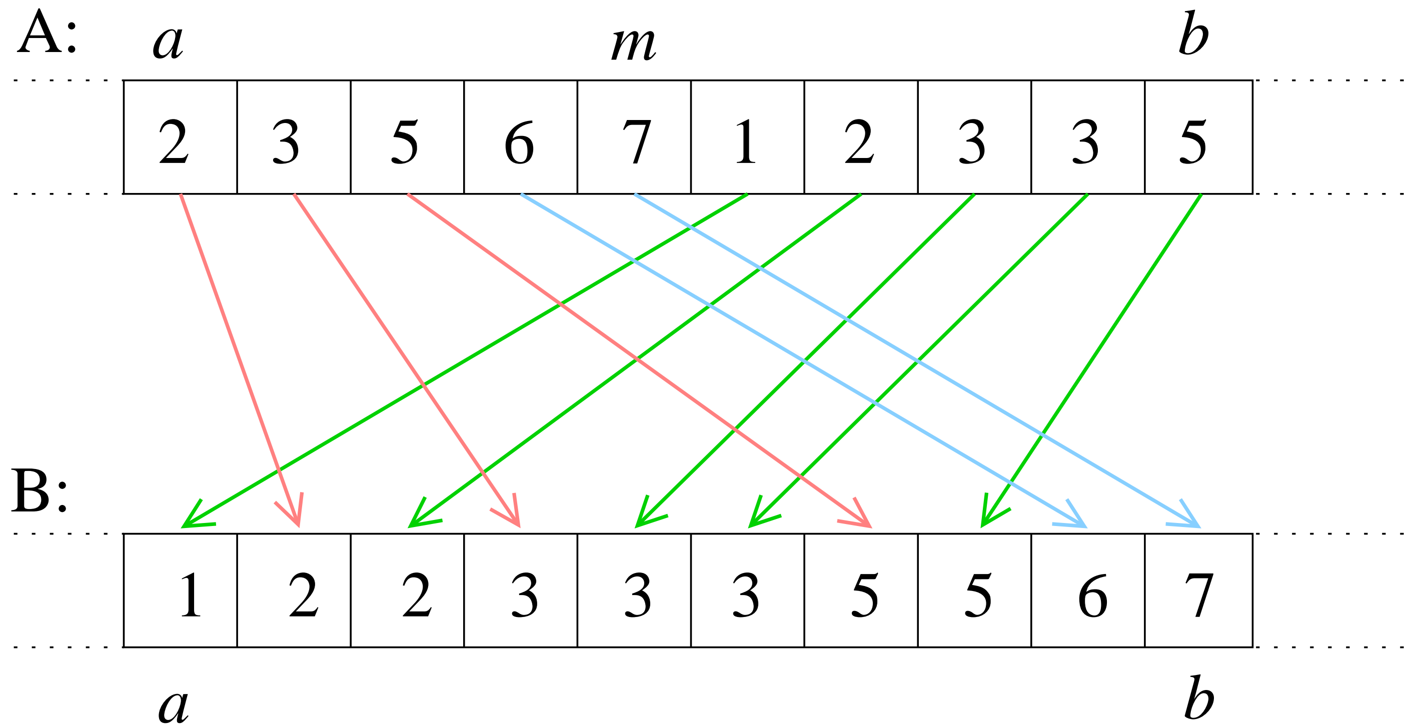
Resultat: $\mathtt{A}[a \mathinner{..} b]$ ist sortiert.

Ansatz: Quasiparalleler Durchlauf durch die Teilarrays:

„Reißverschlussverfahren“.

Benutze Hilfsarray $\mathtt{B}[a \mathinner{..} b]$ für das Ergebnis.

Vergleiche Kapitel 2, Folie 29: **union**(S_1, S_2) bei der Implementierung von **DynSets** mit sortierten Listen/Arrays.



„Mischen“ von `A`$[a \ldots m]$ und `A`$[m+1 \ldots b]$.

Rote Pfeile: `i`–`k`-Paare in Zeile (6).

Grüne Pfeile: `j`–`k`-Paare in Zeile (7).

Blaue Pfeile: `i`–`k`-Paare in Zeile (8), Kopieren.

Prozedur Merge(a, m, b)

```
            // für 1 ≤ a ≤ m < b ≤ n, sortiert A[a..b],
            // nimmt an, dass A[a..m] und A[m+1..b] sortiert sind.
(1)   i ← a;        // Index in A[a..m]
(2)   j ← m + 1;    // Index in A[m+1..b]
(3)   k ← a;        // Index in B[a..b]
(4)   while i ≤ m and j ≤ b do
(5)     if A[i] ≤ A[j]
(6)       then B[k] ← A[i]; ++i; ++k
(7)       else B[k] ← A[j]; ++j; ++k
                // verbleibendes Schlussstück kopieren:
(8)   while i ≤ m do B[k] ← A[i]; ++i; ++k;
(9)   while j ≤ b do B[k] ← A[j]; ++j; ++k;
                // B[a..b] nach A[a..b] kopieren:
(10)  for k from a to b do A[k] ← B[k].
```

Eigenschaften von **Merge**(a, m, b):

- Das Verfahren ist korrekt, d. h.: Wenn die beiden Teile sortiert sind, enthält $\texttt{A}[a..b]$ dieselben Einträge wie vorher, in aufsteigend sortierter Reihenfolge.
- Es werden höchstens $b - a$ Schlüsselvergleiche ausgeführt. (Bei jedem Vergleich wird ein Schlüssel transportiert, nach Schleife (4)–(7) ist mindestens ein Eintrag übrig.)
- Der Zeitbedarf ist $\Theta(b - a)$;
- **Merge** ist **stabil**. (Vergleichsmodus Zeile (5) bevorzugt linkes Teilarray.)

Satz 6.2.1

Mergesort$(\texttt{A}[1..n])$ sortiert $\texttt{A}[1..n]$ stabil.

Beweis: Durch Induktion über rekursive Aufrufe von **r_Mergesort**(a, b) zeigt man, dass die Prozedur das Teilarray $\texttt{A}[a..b]$ stabil sortiert. Im Induktionsschritt benutzt man die eben aufgelisteten Eigenschaften von **Merge**. $\square$

Zeitanalyse von Mergesort über „Rekurrenzgleichungen". Zähle nur Vergleiche. Vor.: $n_0 = 0$.

$C(n) :=$ maximale Anzahl von Vergleichen bei Eingabelänge n.

$$C(1) = 0, C(2) = 1, C(3) = 3.$$

$$C(n) = C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor) + n - 1. \qquad (*)$$

(Längen der Teilarrays: $\lceil n/2 \rceil$ und $\lfloor n/2 \rfloor$, **Merge** benötigt $\leq n - 1$ Vergleiche.)

Ausrechnen von Hand liefert:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
$C(n)$	0	1	3	5	8	11	14	17	21	25	29	33	37	41	...

Zunächst: Zweierpotenzen $n = 2^k$, $k \geq 0$.

n	1	2	4	8	16	32	64	...
$C(n)$	0	1	5	17	49	129	321	...

Geratene **Beobachtung:** $k \geq 0 \Rightarrow C(2^k) = (k-1) \cdot 2^k + 1$.

Beweis durch Induktion über k:
Für $k = 0$: $C(2^0) = (0-1) \cdot 2^0 + 1$.
Sein nun $k \geq 1$. Die I.V. sagt: $C(2^{k-1}) = (k-2) \cdot 2^{k-1} + 1$.
I.-Schritt: Nach Rekurrenz: $C(2^k) = 2C(2^{k-1}) + 2^k - 1$.
Nach I.V.: $C(2^k) = 2((k-2) \cdot 2^{k-1} + 1) + 2^k - 1 = (k-1) \cdot 2^k + 1$.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
$C(n)$	0	1	3	5	8	11	14	17	21	25	29	33	37	41	...

Zwischen 2^{k-1} und 2^k jeweils Zuwachs um k.

Damit **Vermutung**: Für $2^{k-1} < n \leq 2^k$, d. h. $k = \lceil \log n \rceil$:

$$C(n) = (k-2)2^{k-1} + 1 + (n - 2^{k-1})k = nk - 2^k + 1.$$

Satz 6.2.2

$C(n) = n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1$, für $n \geq 1$.

Beweis: Durch Induktion über $k \geq 1$ zeigen wir: Für $2^{k-1} \leq n \leq 2^k$ gilt $C(n) = nk - 2^k + 1$.

I.A.: $k = 1$. Für $n = 1$ gilt $C(1) = 0 = 1 \cdot 1 - 2^1 + 1$ und $C(2) = 1 = 2 \cdot 1 - 2^1 + 1$.

Nun sei $k \geq 2$ gegeben; **I.V.:** Beh. gilt für $k - 1$.

I.-Schritt:

1. Fall: $n = 2^k$. Nach Beobachtung: $C(n) = 2^k(k-1) + 1 = nk - 2^k + 1$.

2. Fall: $n = 2^{k-1}$. Nach I.V. gilt $C(n) = 2^{k-1}(k-1) - 2^{k-1} + 1 = 2^{k-1} \cdot k - 2^k + 1 = nk - 2^k + 1$.

3. Fall: $2^{k-1} < n < 2^k$.

Es gilt $2^{k-2} < n/2 < 2^{k-1}$, also $2^{k-2} \leq \lfloor n/2 \rfloor \leq \lceil n/2 \rceil \leq 2^{k-1}$.

Nach **I.V.** haben wir also

$C(\lceil n/2 \rceil) = \lceil n/2 \rceil \cdot (k-1) - 2^{k-1} + 1$ und $C(\lfloor n/2 \rfloor) = \lfloor n/2 \rfloor \cdot (k-1) - 2^{k-1} + 1$.

Aus der Rekurrenzgleichung $(*)$ erhalten wir damit:

$$\begin{aligned} C(n) &= C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor) + n - 1 \\ &= \lceil n/2 \rceil (k-1) - 2^{k-1} + 1 + \lfloor n/2 \rfloor (k-1) - 2^{k-1} + 1 + n - 1 \\ &= n(k-1) + n - 2^k + 1 = nk - 2^k + 1. \end{aligned}$$

□

Korollar 6.2.3

$C(n) \leq n \log n$, für $n \geq 1$.

Beweis: Wenn $n = 2^k$ ist, gilt nach Satz 6.2.2:

$C(n) = 2^k \cdot k - 2^k + 1 \leq nk = n \log n$.

Wenn $2^{k-1} < n < 2^k$ ist für ein $k \geq 1$, dann gilt:

$nk - 2^k + 1 = n(k-1) + (n - 2^k + 1) \leq n \log n$. □

Korollar 6.2.4

Die Rechenzeit von Mergesort ist $\Theta(n \log n)$.

Variante 1: (Sollte man immer anwenden!)

In **r_Mergesort** kleine Teilarrays ($b - a \leq n_0$) durch (z. B.) **Insertionsort** sortieren.

Vermeidet überproportional hohen Zeitaufwand für viele rekursive Aufrufe bei kleinen Teilarrays.
Das optimale n_0 auf einer konkreten Maschine kann ein Experiment liefern.
($n_0 = 8$ ist keinesfalls zu groß.)

Variante 2: Mergesort für lineare Listen.

Wenn Liste L Länge 1 hat: fertig; sonst teile L in Teillisten L_1, L_2 der halben Länge.

(Durchlaufe L, hänge die Einträge abwechselnd an L_1 und L_2 an. Kosten: Θ(Länge von L).)

Auf L_1 und L_2 wird der Algorithmus rekursiv angewendet.

Dann werden die beiden nun sortierten Listen „gemischt“, analog **Merge**.

Analyse der Vergleiche: Identisch zum Obigen.

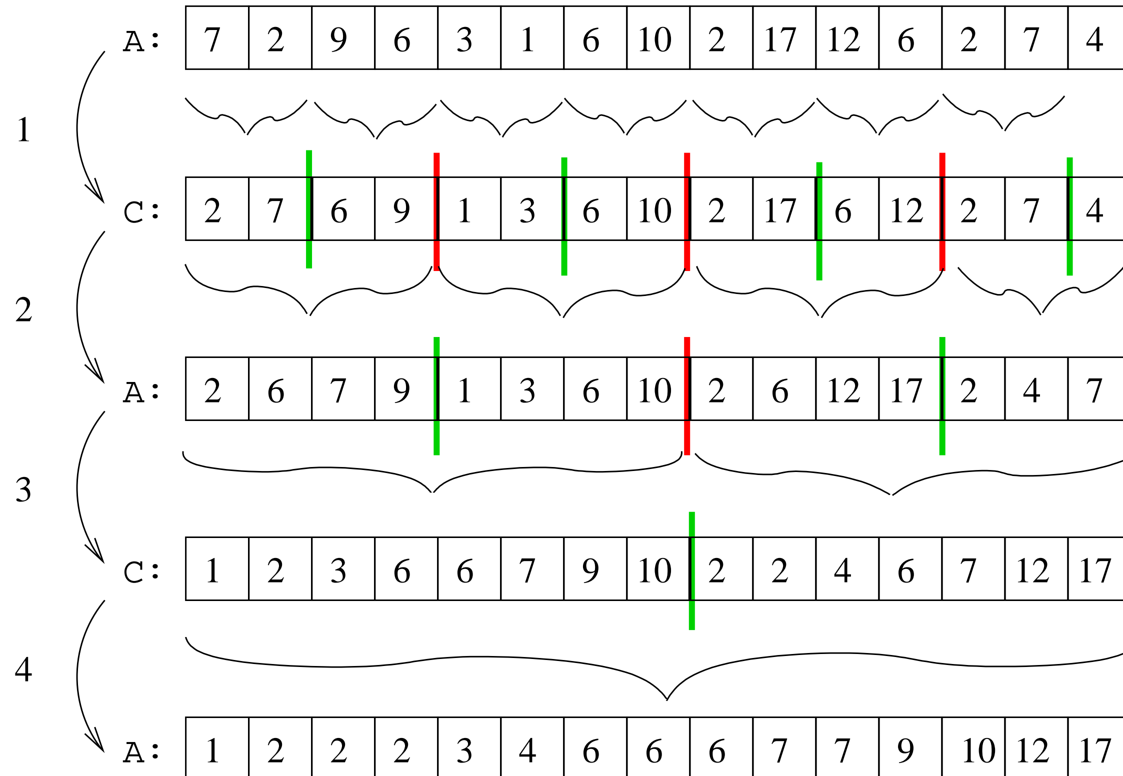
Laufzeit: Proportional zur Anzahl der Vergleiche, also auch $\Theta(n \log n)$.

Kein zusätzlicher Platz (aber Zeiger für die Liste).

Variante 3: Iteratives Mergesort, „bottom-up“

Idee: Arbeite in Runden $i = 1, 2, 3, \ldots, \lceil \log n \rceil$.

Beispiel:



Variante 3: Iteratives Mergesort, „bottom-up“

Runden $i = 1, \ldots, \lceil \log n \rceil$.

Vor Runde i: Teilarrays $\mathtt{A}[(l-1) \cdot 2^{i-1} + 1 \ldots l \cdot 2^{i-1}]$ der Länge 2^{i-1} sind sortiert. (Eventuell ist das am weitesten rechts stehende Teilarray unvollständig.)

Runde i: **merge**$((l-2) \cdot 2^{i-1} + 1, (l-1)2^{i-1}, l \cdot 2^{i-1})$ für $l = 2, 4, 6, \ldots, 2\lceil n/2^i \rceil$,

mit Hilfe eines zweiten Arrays B.

(Mögliche Sonderrolle für unvollständige Teilarrays ganz rechts.)

Trick, spart Kopierarbeit: Benutze 2 Arrays, die in jeder Runde ihre Rolle wechseln.

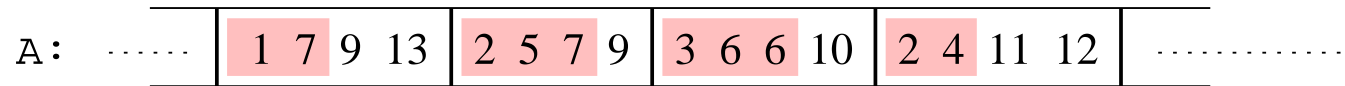
Zeitverhalten von iterativem Mergesort: wie bei rekursivem Mergesort.

Alternative, einfache Zeitanalyse für diese Variante:
Jede Runde kostet offenbar $< n$ Vergleiche und Zeit $\Theta(n)$, und es gibt $\lceil \log n \rceil$ Runden. Also ist die die Zahl der Vergleiche $< n\lceil \log n \rceil$ und die Gesamtzeit $\Theta(n\lceil \log n \rceil) = \Theta(n \log n)$.

Variante 4: Mehrweg-Mergesort

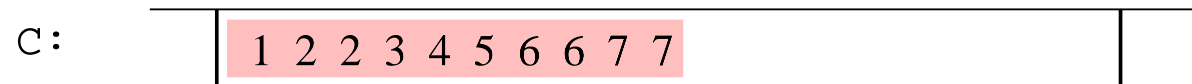
Besonders in Kombination mit dem iterativen Verfahren interessant ist die Möglichkeit, in einer `merge`-Runde nicht nur ein Array, sondern 3, 4, oder viel mehr (k viele) Teilarrays zusammenzumischen.

(Unrealistisch kleines) *Beispiel*:



4–way merge: wähle kleinstes Element aus 4 (Rest–)Arrays.

fertig



Die nächsten gewählten Elemente: "9" aus 1.Teilarray, "9" aus 2. Teilarray, "10", "11", etc.

Benötigt: Auswahl des kleinsten „aus k“. Dafür: Heap (später).

Beim Sortieren von Daten auf Hintergrundspeichern wie **Festplatten** oder **Solid-State-Disks** hat diese Variante gegenüber dem Mischen von zwei Teilfolgen Laufzeitvorteile, da die Anzahl von (zeitaufwendigen) Seitenfehlern (*page faults*) verringert wird.

Daher Empfehlung: Für Sortieren von großen Datenmengen auf Hintergrundspeichern: Mergesort-artige Verfahren benutzen!

Ein ähnlicher Effekt tritt auch beim Sortieren im Hauptspeicher mit k-Wege-Mischen ein: Die Zahl der Cachefehler (*cache misses*) sinkt.

Details, für Interessierte: Ein (Riesen-)Array (n Einträge) im Hintergrundspeicher soll sortiert werden.

Wir nehmen an, dass im Hauptspeicher für den Sortieralgorithmus Platz für M viele Elemente zur Verfügung steht. Mit einem Transport können wir B viele Elemente vom Hintergrundspeicher holen oder dorthin schreiben. B heißt die Seiten- oder Blockgröße, in der „Einheit“ Arrayeinträge.

Wir sortieren zunächst Arraysegmente der Länge M mit einem „internen“ Verfahren, z. B. gewöhnlichem Mergesort, schreiben diese wieder auf den Hintergrundspeicher, das ergibt $2n/B$ viele Blocktransporte.

Dann wählen wir $k = M/B$, mischen also in einer Runde immer M/B viele sortierte Arraysegmente derselben Länge L zu einem Segment der Länge $L(M/B)$ zusammen. (k kann nicht größer als M/B sein, weil für jedes der bearbeiteten k Segmente mindestens ein Block im Hauptspeicher liegen muss.) In jeder Runde werden n/B Blöcke aus dem Hintergrundspeicher geholt und wieder dorthin geschrieben.

Was ist die Anzahl r der Runden im zweiten Teil? Es muss $M \cdot (M/B)^r \approx n$ sein, das heißt $r \approx \log_{M/B}(n/M) = \boldsymbol{\log(n/M)/\log(M/B)}$.

Dieser $(\log(n/M)/\log(M/B))$-malige Transport aller Daten aus dem Hintergrundspeicher in den Hauptspeicher und zurück kostet die Übertragung von $2(n/B)\log(n/M)/\log(M/B)$ vielen Blöcken. Wenn man naiv wie in gewöhnlichem Mergesort immer nur zwei Segmente mischt, erhält man $\boldsymbol{\log(n/M)}$ Runden und $2(n/B)\log(n/M)$ übertragene Blöcke. Der eingesparte Faktor $\log(M/B)$ kann erheblich sein! ($M = 10^6$ und $B = 1000$ ist nicht unrealistisch. Dann ist $\log(M/B) = 10$.) Der eingesparte Faktor hängt nicht von n ab.

Vorlesungsvideo:

Quicksort

6.3 Quicksort

Schnelles Sortieren im Durchschnitt/Erwartungswert [Hoare, 1960]

Standard-Sortierverfahren, bereitgestellt in praktisch allen Programmiersprachen, für Sortieren von Daten mit zahlenwertigen Schlüsseln im Hauptspeicher vorzugsweise zu benutzen.

Folgt ebenso wie Mergesort dem

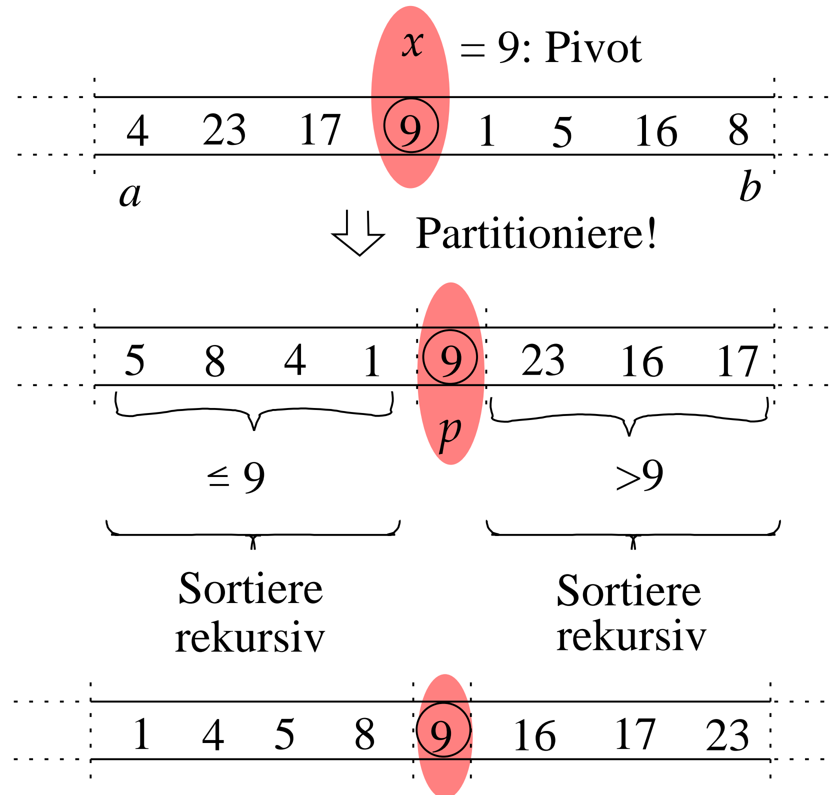
Divide-and-Conquer-Ansatz

Zentrale Teilaufgabe:

Sortiere (Teil-)Array A[a..b], wo $1 \leq a \leq b \leq n$.

Zu erledigen von einer **(rekursiven)** Prozedur **r_qsort**(a, b).

Kernstück: Partitionieren und rekursiv sortieren.



Ansatz für **r_qsort**(a, b), gleichzeitig Korrektheitsbetrachtung:

0) Trivialitätstest: Wenn $a \geq b$, ist nichts zu tun.

(Oder: Wenn $b - a < n_0$: **Insertionsort**(a, b).)

1) Teile: (Hier „**partitioniere**")
Wähle „**Pivotelement**" $x \in \{\texttt{A[}a\texttt{].key}, \ldots, \texttt{A[}b\texttt{].key}\}$.
Arrangiere die Einträge von $\texttt{A[}a..b\texttt{]}$ so um, dass für ein p mit $a \leq p \leq b$ gilt:
$\texttt{A[}p\texttt{].key} = x$ und $\texttt{A[}a\texttt{].key}, \ldots, \texttt{A[}p-1\texttt{].key} \leq x < \texttt{A[}p+1\texttt{].key}, \ldots, \texttt{A[}b\texttt{].key}$.

2) Rekursion auf Teilinstanzen $x_1 = \texttt{A[}a..p-1\texttt{]}$ und $x_2 = \texttt{A[}p+1..b\texttt{]}$:
r_qsort$(a, p-1)$; **r_qsort**$(p+1, b)$.

Nach I.V. gilt nun: $\texttt{A[}a..p-1\texttt{]}$, $\texttt{A[}p+1..b\texttt{]}$ sortiert.

3) Kombiniere: Es ist nichts zu tun: $\texttt{A[}a..b\texttt{]}$ ist sortiert.

Kernstück: 2) Partitionieren.

Möglichkeiten für Wahl des **„Pivotelements“** x:

- `A[`a`].key` (**erstes** Element des Teilarrays)
- `A[`b`].key` (**letztes** Element des Teilarrays)
- `A[`$a + \lfloor (b-a)/2 \rfloor$`].key` (**Mitte** des Teilarrays)
- **„Clever Quicksort“**: Das Pivotelement x ist der **Median** der drei Schlüssel `A[`a`].key`, `A[`b`].key`, `A[`$a + \lfloor (b-a)/2 \rfloor$`].key`
- **„Randomisiertes Quicksort“**:
  Wähle einen der $b - a + 1$ Einträge in `A[`a`..`b`]` **zufällig**.

Allgemein: Wähle ein s mit $a \le s \le b$, **vertausche** `A[`a`]` mit `A[`s`]`.

Das Pivotelement ist danach immer `A[`a`].key`.

Partitionieren mit Pivotelelement $x = \texttt{A}[a]\texttt{.key}$

Klar: $b - a$ Vergleiche genügen, um Schlüssel $\leq x$ „nach links" und Schlüssel $> x$ „nach rechts" zu schaffen und den Eintrag mit dem Pivotelement an die Stelle zwischen den Bereichen zu schreiben.

Eine sehr geschickte Methode wurde schon von **C. A. R. Hoare**[1] angegeben: kein zusätzlicher Platz, keine unnötige Datenbewegung.

Hier: Variante des in der AuP-Vorlesung angegebenen Verfahrens.

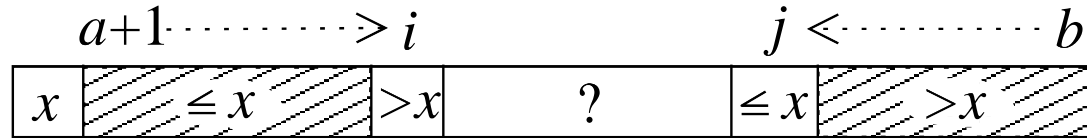
---

[1] **C. A. R. Hoare** (*1934), brit. Informatiker, erfand Quicksort & Korrektheitskalkül („Hoare-Kalkül").

„I conclude that there are two ways of constructing a software design: One way is to make it *so simple* that there are *obviously no* deficiencies and the other way is to make it *so complicated* that there are *no obvious* deficiencies. The first method is far more difficult." (Dankesrede für den Turingpreis 1980)

„I think Quicksort is the only really interesting algorithm that I've ever developed."

Quelle: Wikipedia, `https://de.wikipedia.org/wiki/Tony_Hoare#Zitate`

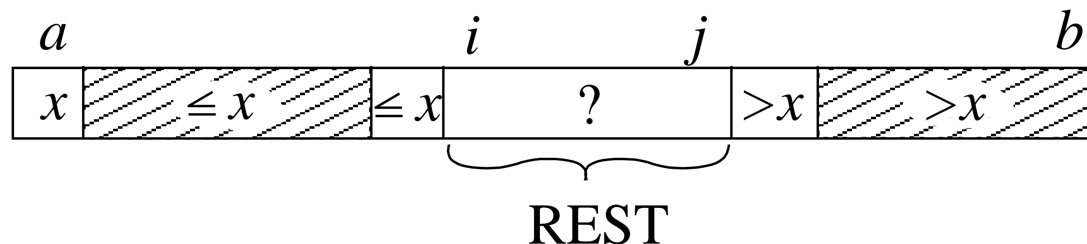


Ignoriere A[a].

Lasse ab $a+1$ einen „Zeiger“ (Index) i nach rechts wandern, bis ein Schlüssel $> x$ erreicht wird.

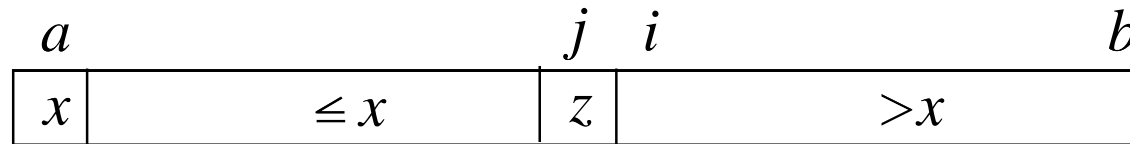
Lasse ab b einen „Zeiger“ j (Index) nach links wandern, bis ein Schlüssel $\leq x$ erreicht wird.

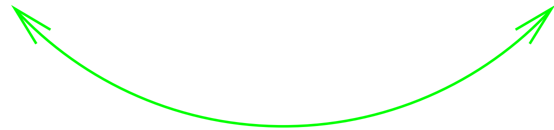
Falls $i < j$: Vertausche A[i] und A[j].

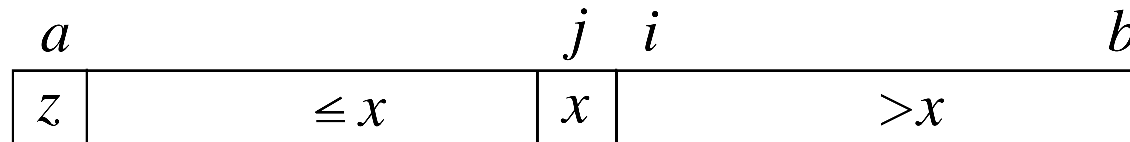


Erhöhe i um 1, erniedrige j um 1. Wiederhole . . .

. . . bis sich i und j „treffen“ und „überkreuzen“, d. h. bis $i = j + 1$ gilt.





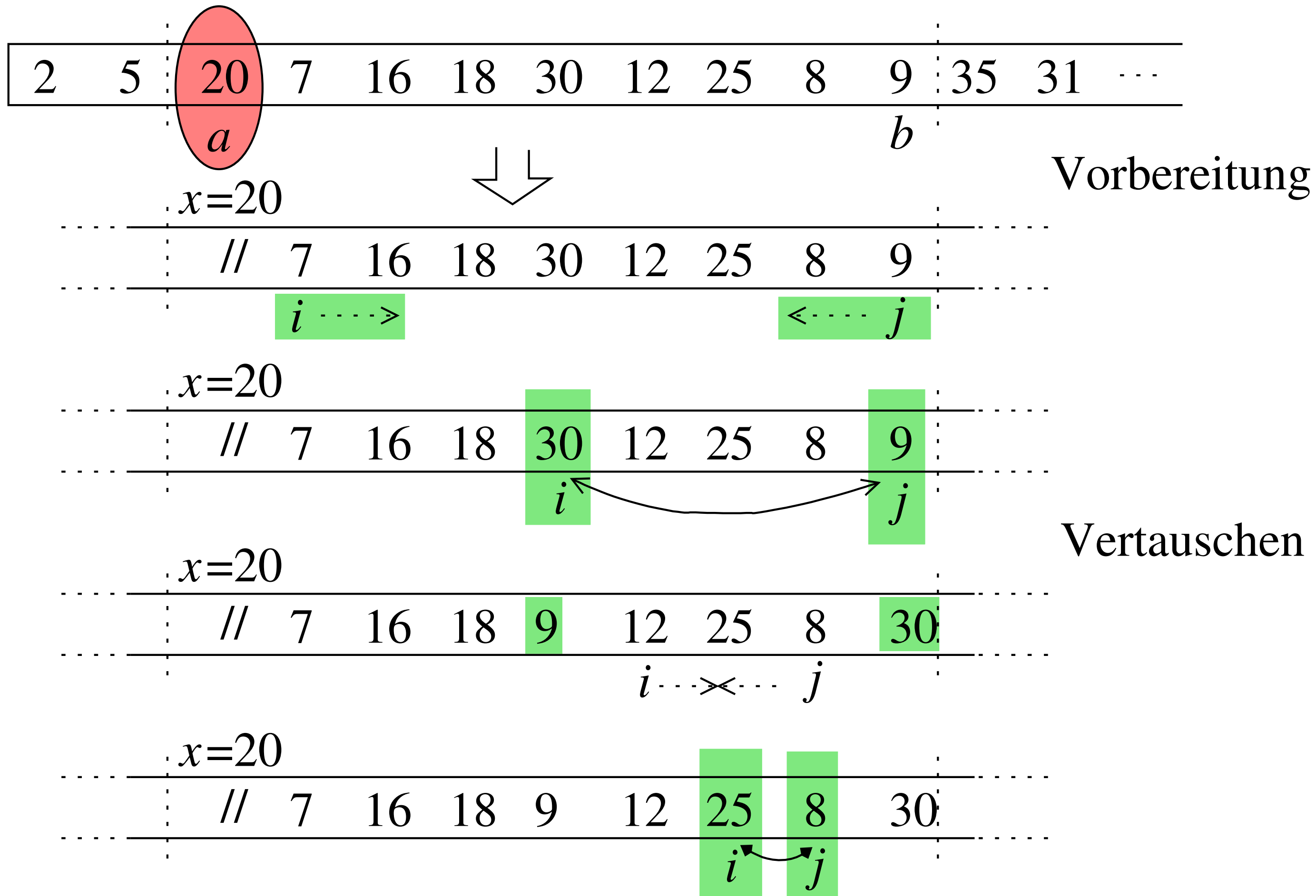


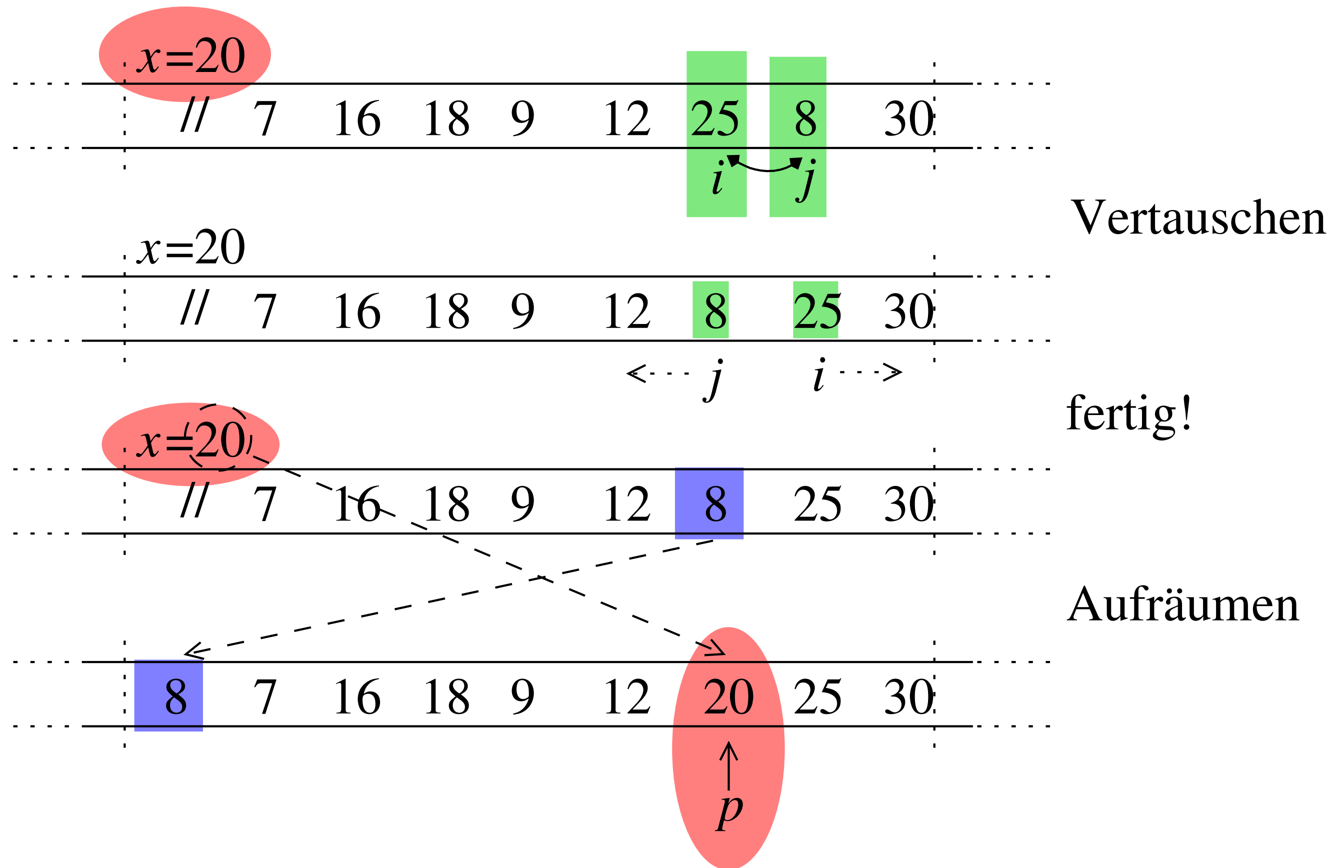
$p=j$

Es gilt jetzt: $\texttt{A}[a+1]\texttt{.key}, \ldots, \texttt{A}[j]\texttt{.key} \leq x < \texttt{A}[j+1]\texttt{.key}, \ldots, \texttt{A}[b]\texttt{.key}$.

Nun vertausche $\texttt{A}[a]$ mit $\texttt{A}[j]$. Position p des Pivotelements x ist jetzt $\texttt{A}[j]$.

Beispiel für partition:





Sorgfalt bei der Implementierung der Idee ist nötig, um Randfälle und den Abschluss korrekt zu behandeln und um unnötige Vergleiche zu verhindern. Nach einigem Tüfteln ergibt sich Folgendes.

Prozedur partition$(a, b, \mathtt{p})$ // Partitionierungsprozedur in **r_qsort**(a, b)

// Voraussetzung: $a < b$; Pivotelement: $\mathtt{A}[a]$

```
(1)   x ← A[a].key;
(2)   i ← a + 1;  j ← b;
(3)   while i ≤ j do        // noch nicht fertig; Hauptschleife
(4)     while i ≤ j and A[i].key ≤ x do ++i;
(5)     while j > i and A[j].key > x do --j;
(6)     if i = j then --j;        // Wissen: A[i].key > x
(7)     if i < j then
(8)       vertausche A[i] und A[j];
(9)       ++i;  --j;
(10)  if a < j then vertausche A[a] und A[j];
(11)  p ← j;   // Rückgabewert
```

Achtung: Diese Implementierungsdetails vermeiden Mehrfachvergleiche zwischen Schlüsseln.

Lemma 6.3.1 (Korrektheit von **partition**)

Ein Aufruf **partition**$(a, b, \mathtt{p})$, für $1 \le a < b \le n$, bewirkt Folgendes:

Sei $x = \mathtt{A}[a]\mathtt{.key}$ (vor Aufruf), p der Inhalt von $\mathtt{p}$ (nach Aufruf).

(a) $\mathtt{A}[a\,..\,b]$ wird so umarrangiert, dass $x = \mathtt{A}[p]\mathtt{.key}$ und
$\mathtt{A}[a+1]\mathtt{.key}, \ldots, \mathtt{A}[p-1]\mathtt{.key} \le x < \mathtt{A}[p+1]\mathtt{.key}, \ldots, \mathtt{A}[b]\mathtt{.key}$.

(b) An welche Stelle die Einträge $\mathtt{A}[j]$, $a \le j \le b$, gebracht werden, hängt nur von der Menge $\{i \in [a+1, b] \mid \mathtt{A}[i]\mathtt{.key} > x\}$ ab, nicht von den konkreten Schlüsselwerten.

Beweis: (a) Sorgfältige, etwas mühselige Fallunterscheidung, nur für Unerschrockene auf den nächsten Folien durchexerziert. (b) Alle Entscheidungen werden nur aufgrund von Vergleichen mit x getroffen.$\square$

Beweis von (a): **(Nicht prüfungsrelevant.)**

Wir beobachten die folgende **Schleifeninvariante** für die **Hauptschleife** (3)–(9)

für a, b und die Inhalte i, j, x von `i`, `j`, `x`:

Unmittelbar vor Beginn von Zeile (3) gilt:

(I) $a \leq i - 1 \leq j \leq b$,
(IIa) $\texttt{A}[a+1]\texttt{.key}, \ldots, \texttt{A}[i-1]\texttt{.key} \leq x$ und
(IIb) $\texttt{A}[j+1]\texttt{.key}, \ldots, \texttt{A}[b]\texttt{.key} > x$.

Beweis durch Induktion über Schleifendurchläufe, gleich.

Beweis der Korrektheit der Prozedur **partition**:

Die Hauptschleife bricht ab, wenn in Zeile (3) $j < i$ gilt.

Wegen (I) gilt dann $j = i - 1$ und damit wegen (IIa,b) auch
A[$a + 1$],...,A[j] $\leq x$ und A[$j + 1$],...,A[b] $> x$.
Wenn $j = a$ ist, gibt es also außer A[a] keine Einträge mit Schlüssel $\leq x$, es ist nichts zu tun.
Sonst stellt die Vertauschung in Zeile (10) die verlangte Ausgabe her.

Beweis der **Invariante**: Induktion über Schleifendurchläufe.

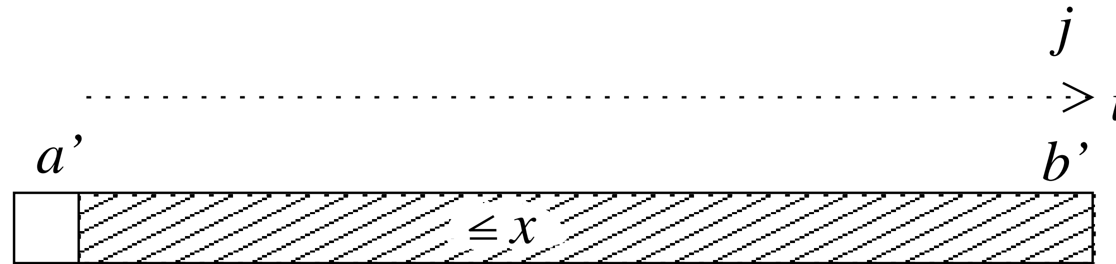
I.A.: Zu Beginn gilt die Invariante: (I) ist klar, und (IIa,b) sind Aussagen über die leere Menge.

I.-Schritt: Betrachte einen kompletten Schleifendurchlauf, unter der Annahme, dass anfangs die Invariante gilt.

Wir können $i \leq j$ annehmen (sonst bricht die Schleife ab).

Setze $a' := i - 1$ und $b' := j$. Dann gilt $a \leq a' < b' \leq b$.
Aus der Art, wie in Zeilen (4) und (5) die Variablen `i` und `j` behandelt werden, sieht man, dass die Einträge in
$\mathtt{A}[a+1\,..\,a']$ und in $\mathtt{A}[b'+1\,..\,b]$ in der Schleife weder gelesen noch verändert werden.

Wir können also so tun, als ob wir im ersten Schleifendurchlauf wären, mit Teilarray $\mathtt{A}[a'+1\,..\,b']$.



1. Fall: Die Schleife (4) bricht ab, weil $i > j$ wird.
Dann ist offenbar jetzt $i - 1 = j = b'$
(also gilt (I)) und $\texttt{A[}a' + 1\texttt{]}, \ldots, \texttt{A[}b'\texttt{]} \leq x$,
weil dies in (4) getestet wurde.

Also gilt (IIa,b) für die neuen Werte i und j.

Die Tests in Zeilen (5), (6), (7) ergeben jeweils $false$, die Anweisungen in Zeilen (5)–(9) werden nicht ausgeführt.

(Die Hauptschleife wird nun nicht mehr ausgeführt.)

2. Fall:

Die Schleife (4) bricht mit $i \leq j$ ab, weil

(∗1) $$\texttt{A[}i\texttt{].key} > x$$

ist.

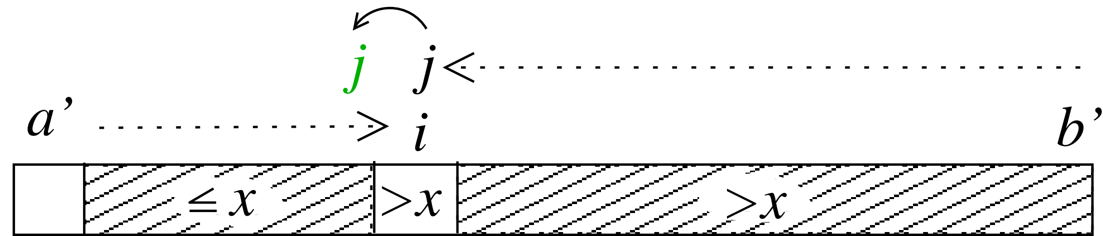
Dann gilt

(∗2) $$\texttt{A[}a' + 1\texttt{].key}, \ldots, \texttt{A[}i - 1\texttt{].key} \leq x,$$

weil dies getestet worden ist.

Beobachte auch:

(∗3) $$a' + 1 \leq i \leq j \leq b'.$$



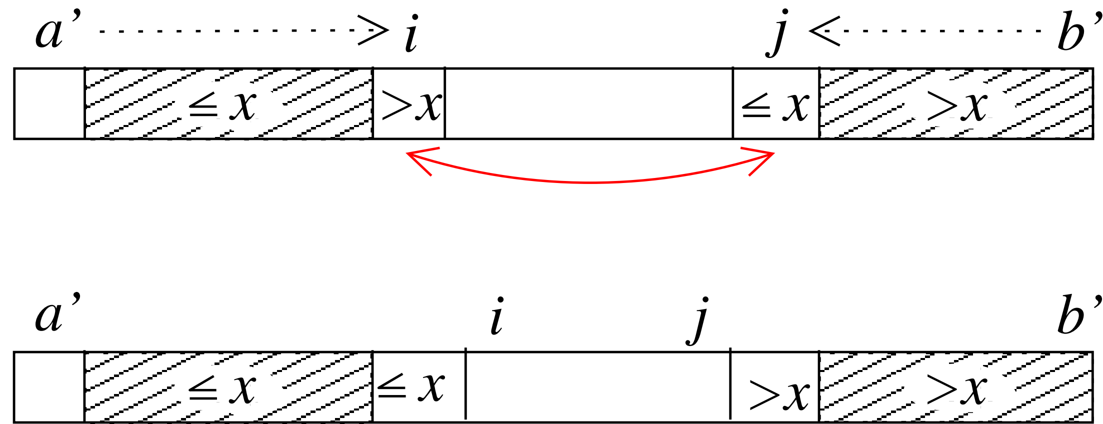
Unterfall 2a: Die Schleife in (5) bricht ab, weil $i = j$ wird.
Dann gilt (weil dies getestet wurde):

(∗4) $\texttt{A}[j+1]\texttt{.key}, \ldots, \texttt{A}[b']\texttt{.key} > x$

und $\texttt{A}[j]\texttt{.key} > x$ (wegen **(∗1)**). Also gilt (IIb) auch nach Zeile (6).
Weil sich i nicht ändert, haben wir (IIa) wegen **(∗2)**.

Zeile (6) verringert j um 1, so dass nun $a' \le j$, also (I) gilt.

(Der nächste Test in Zeile (3) beendet die Hauptschleife.)



Unterfall 2b: Die Schleife in (5) bricht mit $i < j$ ab, weil

(∗5) $\texttt{A[}j\texttt{].key} \leq x$

gilt. Wegen **(∗3)** gilt nach der Veränderung der Indizes in Zeile (9):

$a' \leq j$ und $i - 1 \leq j < b'$, also (I).

(IIa), (IIb) folgen ebenfalls unter Verwendung von **(∗1)**, **(∗2)**, **(∗4)**.

Dies beendet den Beweis der Schleifeninvarianten. □

Lemma 6.3.2 (Aufwand von **partition**)

Bei Aufruf **partition**$(a, b, \texttt{p})$ gilt:

(a) Die Anzahl der Schlüsselvergleiche ist exakt $b - a$;

(b) der (Zeit-)Aufwand ist $\Theta(b - a)$.

Beweis: Sei i der Inhalt von `i`, j der Inhalt von `j`.

(a) Schlüsselvergleiche finden nur statt, so lange $i < j$ gilt. Nach jedem Vergleich „$\texttt{A}[i]\texttt{.key} \le x$“ wird i um 1 erhöht (Zeile (4) oder (9)), nach jedem Vergleich „$\texttt{A}[j]\texttt{.key} > x$“ wird j um 1 erniedrigt (Zeile (5) oder (9)).

In der Situation $i \ge j$ wird kein Vergleich mehr ausgeführt. Daher kann kein Eintrag zweimal mit x verglichen werden.

Prozedur r_qsort(a, b)

// Rekursive Prozedur im Quicksort-Algorithmus, $1 \le a \le n+1$, $0 \le b \le n$.

(1) **if** $a < b$ **then**

(2) s $\leftarrow$ ein Element von $\{a, \dots, b\}$;

// Es gibt verschiedene Methoden, um s zu wählen; s. Folie 22

(3) vertausche A[a] und A[s];

(4) **partition**(a, b, p); // p: Ausgabeparameter

(5) **r_qsort**$(a, \mathrm{p} - 1)$;

(6) **r_qsort**$(\mathrm{p} + 1, b)$.

Effiziente Variation zum Abbruchkriterium in Zeile (1):
Für ein passendes $n_0 \ge 2$ Teilarrays der Länge $\le n_0$ mit **Insertionsort**(A[$a..b$]) sortieren.

Lemma 6.3.3 Ein Aufruf von **r_qsort**(a, b) sortiert A[$a \mathinner{..} b$]. (Beweis: s. Folie 21.)

Um A[$1 \mathinner{..} n$] zu sortieren:

Prozedur Quicksort(A[$1 \mathinner{..} n$])

(1) **r_qsort**$(1, n)$.

Rechenzeit von Quicksort$(\mathtt{A}[1..n])$:

Aufwand für einen Aufruf **r_qsort**(a,b) ist $O(1)$

plus der Aufwand für **partition**(a,b,p), proportional zu $b-a$ (falls $a<b$)

plus der Aufwand für die rekursiven Aufrufe (falls $a<b$).

$b-a$ ist genau die Anzahl der Schlüsselvergleiche, die von **partition**(a,b,p) ausgeführt werden.

Die Aufrufe **partition**(a,b,p) mit $a<b$ haben verschiedene Pivotelemente, also gibt es davon $\le n$ viele. Jeder Aufruf mit $a<b$ generiert höchstens zwei Aufrufe für ein leeres oder ein einelementiges Teilarray, also gibt es davon $\le 2n$ viele.

Die Gesamtrechenzeit für **Quicksort**$(\mathtt{A}[1..n])$ ist also:

$$O(n) + O(\underbrace{\text{Gesamtanzahl aller Schlüsselvergleiche}}_{=:\,\boldsymbol{V}}).$$

V hängt von der Anordnung der Eingabe ab! (Bei randomisiertem QS zusätzlich vom Zufall.)

Schlechtester Fall („worst case“):

Wir überlegen: Wenn zwei Schlüssel verglichen werden, ist einer das Pivotelement, der andere wandert in eines der beiden Teilarrays.

$\Rightarrow$ Es werden niemals dieselben Schlüssel später nochmals verglichen.

Also gilt immer: $V \le \binom{n}{2} = \frac{1}{2}n(n-1)$.

Der Wert $V = \binom{n}{2}$ entsteht zum Beispiel, wenn die Eingabe schon sortiert ist und man in **r_qsort**(a, b) immer `A[`a`]` als Pivotelement wählt.

(Dann ist das linke Teilarray leer, das rechte enthält $b - a$ Elemente.
Die Rekursionstiefe ist n, und $V = (n-1) + (n-2) + \cdots + 3 + 2 + 1 = \binom{n}{2}$.)

Achtung: Ein ähnlicher Effekt tritt ein, wenn das Array „fast“ sortiert ist.

Durchschnittlicher/erwarteter **Fall**:

Zur Vereinfachung nehmen wir an:
Eingabeschlüssel sind n verschiedene Zahlen $x_1 < \cdots < x_n$.

Wir machen die folgende **Wahrscheinlichkeitsannahme**:

Jede Anordnung der Eingabeschlüssel hat dieselbe Wahrscheinlichkeit.

Es gibt $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$ viele solche Anordnungen.

Als Pivotelement wählen wir den Eintrag an einer festen Stelle, z. B. `A[`a`]`.

Unter dieser Wahrscheinlichkeitsannahme wird V zu einer **Zufallsgröße**.

Wir untersuchen den Erwartungswert/Mittelwert von V.

(**Vorsicht!** Die folgende Überlegung gilt nicht für „Clever-Quicksort“ und nicht, wenn die Schlüssel nicht alle verschieden sind.)

$C_n :=$ die erwartete Anzahl der Vergleiche beim Sortieren eines Teilarrays mit n Einträgen mit **r_qsort** (unter der Annahme „zufällige Anordnung von $x_1 < \cdots < x_n$ in der Eingabe“).

$C_0 = 0$, (klar)
$C_1 = 0$, (klar) $C_2 = 1$, (ein Vergleich mit einem Pivot, dann fertig).

C_3: Wenn man beim ersten **r_qsort**(a, b)-Aufruf mit $b = a + 2$ das mittlere Element x_2 als Pivot hat, gibt es 2 Vergleiche insgesamt.
Wahrscheinlichkeit hierfür: $\frac{1}{3}$.

Wenn man x_1 oder x_3 als Pivot hat, hat man zunächst 2 Vergleiche, und einer der rekursiven Aufrufe führt zu einem dritten.
Insgesamt: 3 Vergleiche. Wahrscheinlichkeit hierfür: $\frac{2}{3}$.

Mittlere Vergleichsanzahl: $C_3 = \frac{1}{3} \cdot 2 + \frac{2}{3} \cdot 3 = \frac{8}{3}$.

C_n: Wenn man beim ersten **r_qsort**(a, b)-Aufruf das i-t kleinste Element x_i als Pivotelement wählt, dann erzeugt dieser Aufruf selbst zunächst genau $n - 1$ Vergleiche,

es ergibt sich als partitionierende Position p (in p) genau $a + i - 1$,

und es entstehen die rekursiven Aufrufe

r_qsort$(a, a + i - 2)$ (Teilarray der Länge $i - 1$) und
r_qsort$(a + i, b)$ (Teilarray der Länge $n - i$).

Diese kosten im Durchschnitt C_{i-1} und C_{n-i} Vergleiche.

Beachte: Da anfangs die Schlüssel zufällig angeordnet waren, und bei **partition** nur die Beziehung zwischen Schlüsselwert und dem Pivotelement x relevant ist (Lemma 6.3.1(b)), ist nachher die Anordnung in den beiden Teilarrays ebenfalls zufällig.

Falls x_i Pivotelement ist: Durchschnittliche Kosten sind $\underbrace{n-1}_{\textbf{partition}} + \underbrace{C_{i-1}}_{\text{1. rek. Aufruf}} + \underbrace{C_{n-i}}_{\text{2. rek. Aufruf}}$.

Die Wahrscheinlichkeit, dass x_i Pivotelement ist, ist gleich $1/n$, weil nach der Wahrscheinlichkeitsannahme jede Anordnung der Einträge $x_1, \ldots, x_n$ in $\texttt{A}[a\,..\,b]$ gleichwahrscheinlich ist (Lemma 6.3.1(b)).

Also:

$$C_n = \frac{1}{n} \cdot \sum_{i=1}^{n} (n - 1 + C_{i-1} + C_{n-i})$$
$$= n - 1 + \frac{1}{n} \cdot \sum_{i=1}^{n} (C_{i-1} + C_{n-i}).$$

$C_0 = C_1 = 0$, $C_2 = 1$, $C_n = n - 1 + \frac{1}{n} \cdot \sum_{i=1}^{n} (C_{i-1} + C_{n-i})$ für $n \geq 3$.

Abschnitt 4.2 in Kap. 4, Folien 24–32 :

Rekursionsformel für die mittlere totale Weglänge $A(n)$ in zufällig erzeugten binären Suchbäumen.

$A(0) = A(1) = 0$, $A(2) = 1$,

$A(n) = n - 1 + \frac{1}{n} \cdot \sum_{i=1}^{n} (A(i-1) + A(n-i))$.

Das ist genau die gleiche Rekursion wie die für C_n!

Bemerkung: Dies ist kein Zufall, sondern es besteht eine recht enge Verbindung zwischen Quicksort und dem randomisierten Erzeugen von Suchbäumen.

Also sind auch die Lösungen identisch.

Satz 6.3.4

Die erwartete Vergleichsanzahl bei der Anwendung von Quicksort auf ein Array mit n verschiedenen Einträgen, die zufällig angeordnet sind, ist

$$2(n+1)H_n - 4n \approx \underline{(2\ln 2)n\log n} - 2{,}846\ldots n.$$

Dabei ist $2\ln 2 =$ **1,386294...** (die „Quicksort-Konstante“). □

Korollar 6.3.5

Für die randomisierte Version, bei der das Pivotelement zufällig gewählt wird, gilt dieselbe Formel für die erwartete Anzahl von Vergleichen, **und zwar für jedes beliebige feste Eingabearray** $\mathtt{A}[1\,..\,n]$.

Randomisierung verhindert worst-case-Eingaben bei Quicksort!

Bemerkung:
Eine $O(n\log n)$-Laufzeit stellt sich bei zufälliger Eingabeanordnung bzw. bei randomisiertem Quicksort sogar mit hoher Wahrscheinlichkeit ein. („Zuverlässigkeitsaussage“.)

Bemerkungen:

- Quicksort ist in der Praxis sehr schnell. (Sequenzielle Zugriffe in **partition** sind Cache-freundlich.)
- Vermeide rekursive Aufrufe für allzu kleine Teilarrays: besser direkt mit **Insertionsort** sortieren.
- **Warnung 1:** Wenn $\mathtt{A}[1\,..\,n]$ teilweise sortiert ist und man das Pivotelement naiv wählt, erhält man schlechte Laufzeiten, bis zu $\Omega(n^2)$. Abhilfe: Clever Quicksort oder Randomisiertes Quicksort.
- **Warnung 2:** Wenn $\mathtt{A}[1\,..\,n]$ (viele) identische Schlüssel enthält, ist das beschriebene Verfahren nicht günstig.

  Einfache Abhilfe: Ersetze Schlüssel $\mathtt{A}[i]\mathtt{.key}$ durch das Paar $(\mathtt{A}[i]\mathtt{.key}, i)$, und sortiere lexikographisch mit randomisiertem Quicksort. Oder: **3-Wege-Partition**. (Für Pivot x bilde drei Blöcke mit Einträgen $< x$, $= x$ und $> x$ und wende Rekursion nur auf den ersten und den dritten Block an.)
- Quicksort ist **nicht stabil**.
- Platz (für den Rekursionsstack) kann im schlechtesten Fall bis zu $O(n)$ groß werden.

  Ausweg: Formuliere **r_qsort** um, so dass Rekursion nur für das kleinere der beiden Teilarray angewendet wird. Das größere wird in einer Iteration weiterbehandelt.

  Effekt: Deutlich weniger Prozeduraufrufe. Der Rekursionsstack hat höchstens Tiefe $\log n$ – fast „in situ“. (Siehe nächste Folie.)

Prozedur hr_qsort(a, b) // Halb-rekursive Variante

```
(1)   a ← a; b ← b;
(2)   while a < b do
(3)      s ← ein (z. B. zufälliges) Element von {a, ..., b};
(4)      vertausche A[a] und A[s];
(5)      partition(a, b, p);
(6)      if  p − a < b − p
(7)         then
(8)            hr_qsort(a, p − 1);
(9)            a ← p + 1;      // while-Schleife bearbeitet rechtes Teilarray weiter
(10)        else  // p − a ≥ b − p
(11)           hr_qsort(p + 1, b)
(12)           b ← p − 1;      // while-Schleife bearbeitet linkes Teilarray weiter
```

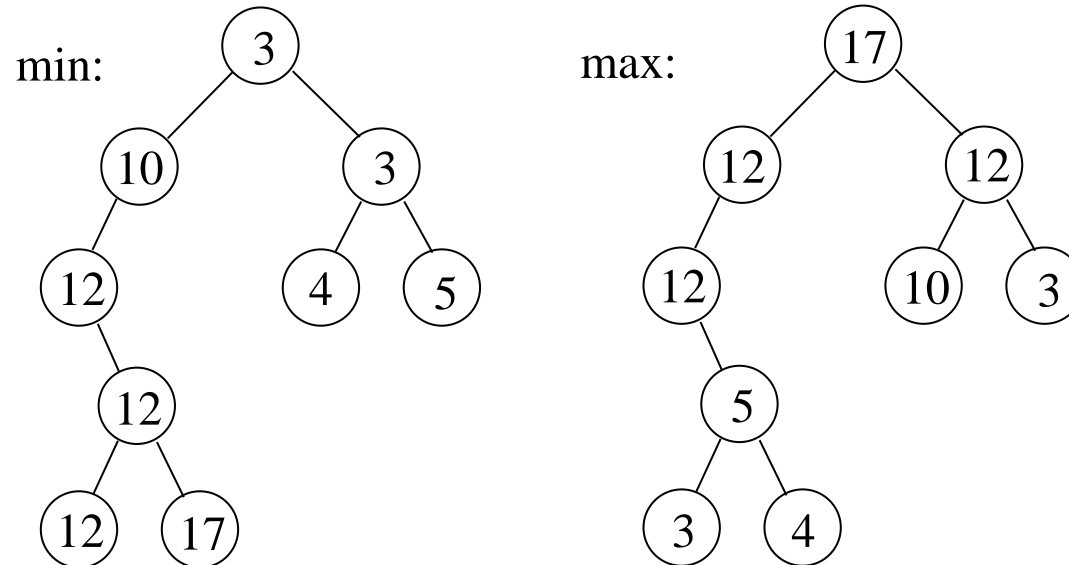
Vorlesungsvideo:

Heaps

6.4 Heapsort

Heap-Ordnung in Binärbäumen:

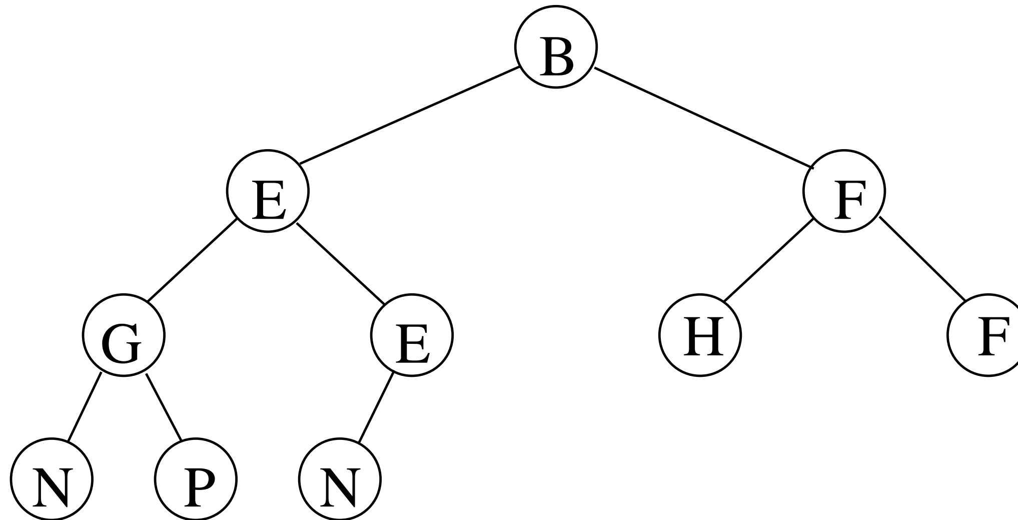
Beispiel: Hier nur interne Knoten relevant. Blätter: Knoten ohne (interne) Kinder.

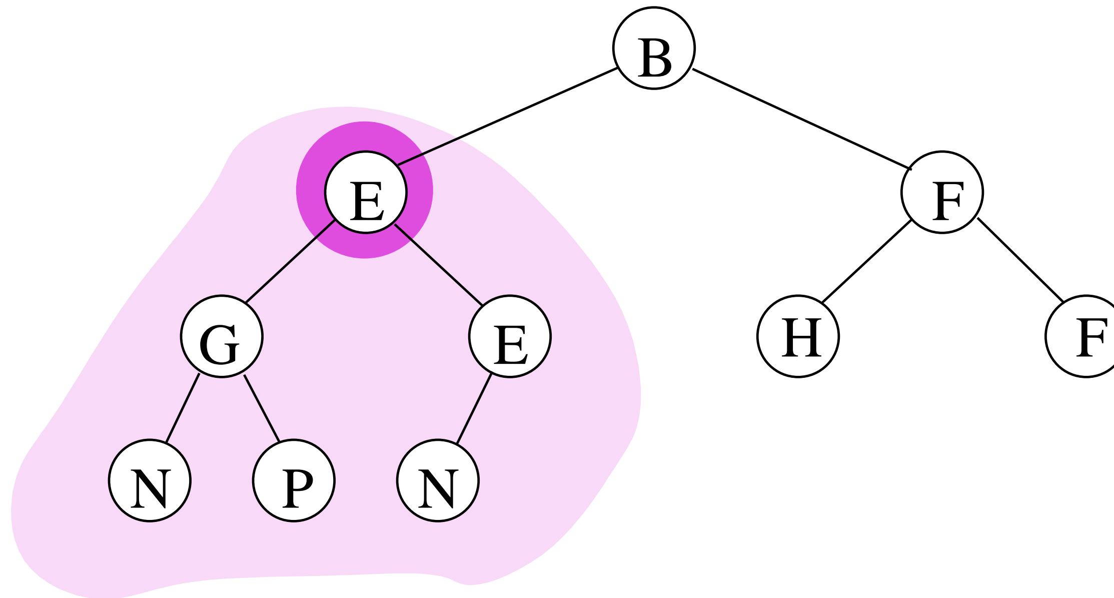


Links/Rechts (Min/Max-Heap): Entlang Blatt-Wurzel-Wegen fallen/steigen die Schlüssel.

Definition 6.4.1

Ein Binärbaum T mit Knotenbeschriftungen $m(v)$ aus $(U, <)$ heißt **min-heapgeordnet** (kurz oft: **heapgeordnet**), wenn für alle Knoten v und w gilt: v Vorgänger von $w \;\Rightarrow\; m(v) \leq m(w)$.



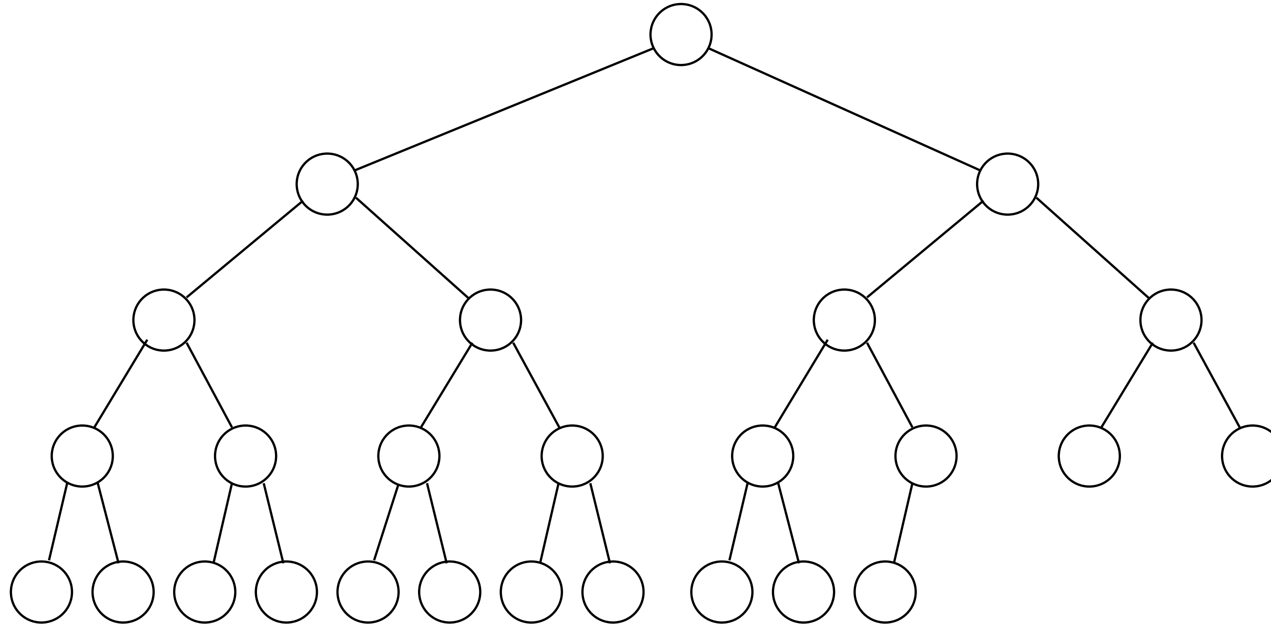


Beachte: T heapgeordnet $\Rightarrow$ in Knoten v steht der minimale Eintrag des Teilbaums T_v mit Wurzel v.

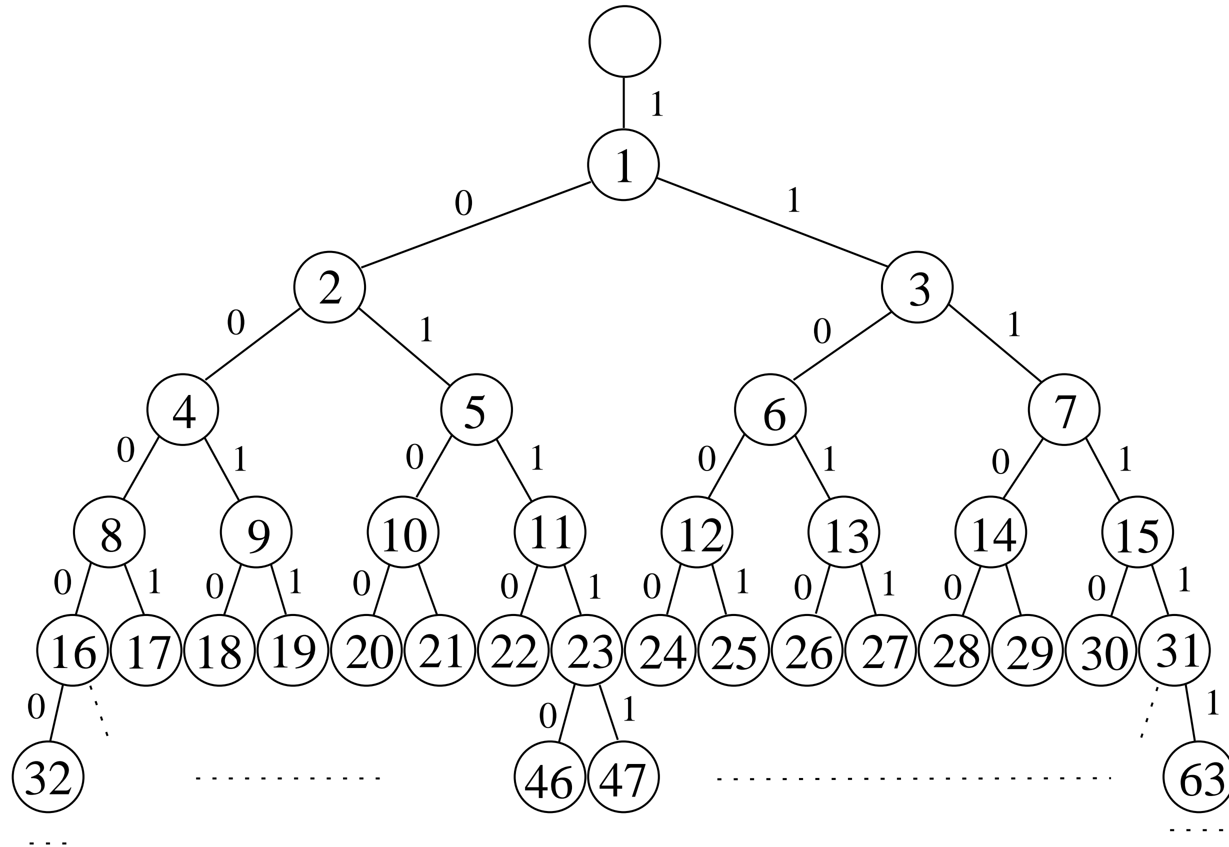
(Variante:

T heißt **max-heapgeordnet**, wenn für alle v, w gilt: v Vorgänger von w $\Rightarrow$ $m(v) \geq m(w)$.)

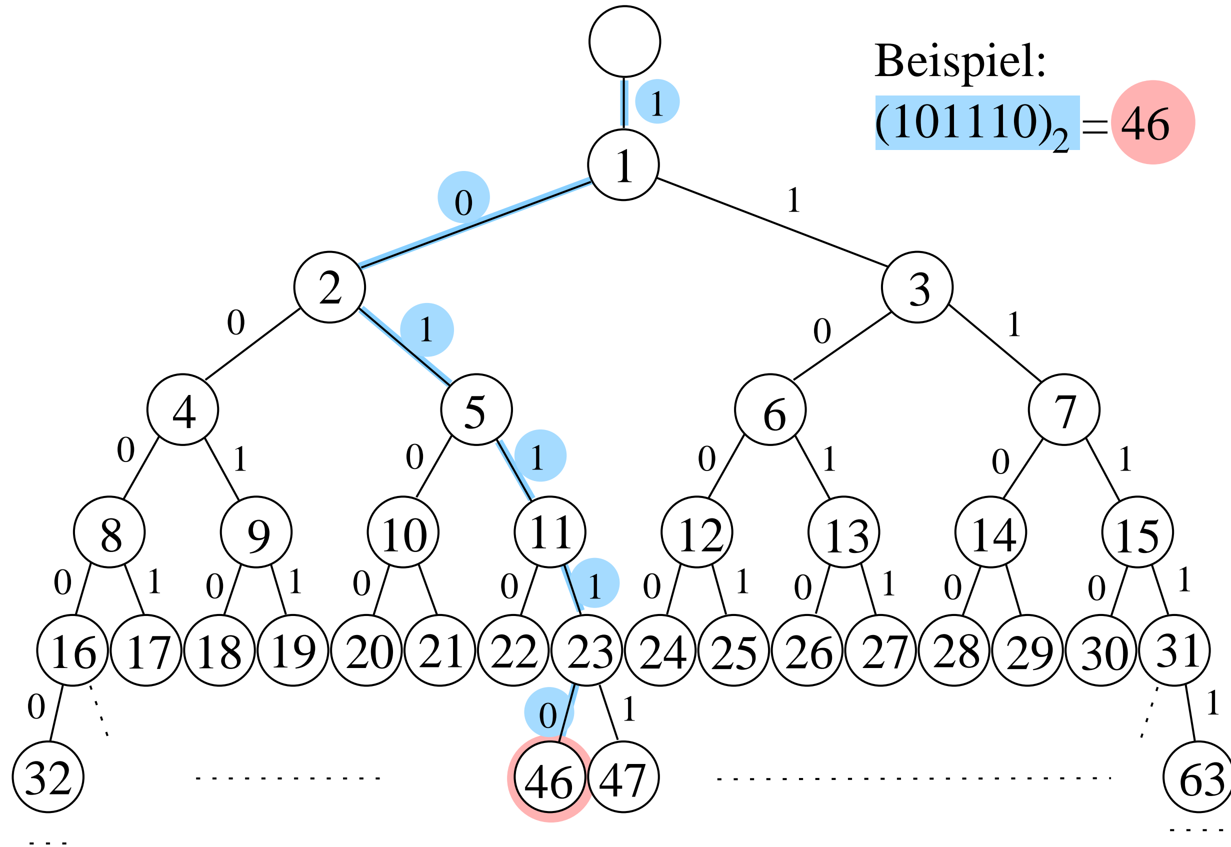
Ein **„linksvollständiger“** Binärbaum:
Alle Levels $j = 0, 1, \ldots$ voll (jeweils 2^j Knoten) bis auf das tiefste.
Das tiefste Level l hat von links her gesehen eine ununterbrochene Folge von Knoten.



(Unendlicher) vollständiger Binärbaum (mit Kunstwurzel), Kante zu linkem/rechtem Kind mit $0/1$ beschriftet, Knoten gemäß **Leveldurchlauf-Anordnung** nummeriert:



(Unendlicher) vollständiger Binärbaum (mit Kunstwurzel), Kante zu linkem/rechtem Kind mit $0/1$ beschriftet, Knoten gemäß **Leveldurchlauf**-**Anordnung** nummeriert:



Ist $s_1 \cdots s_l \in \{0,1\}^l$ die Markierung auf dem Weg von der Wurzel zu Knoten i, so ist $1 s_1 \cdots s_l$ die **Binärdarstellung** von i.

Damit können linksvollständige Binärbaume **ohne Zeiger** als Arrays dargestellt werden: Speichere Eintrag zu Knoten Nummer i im Baum in `A`[i]!

Dies spart den Speicherplatz für die Zeiger ein.

Dennoch können wir leicht zu Kindern und Vorgängern navigieren:

- Knoten 1 ist die Wurzel;
- Knoten $2i$ ist linkes Kind von i;
- Knoten $2i+1$ ist rechtes Kind von i;
- Knoten $i \geq 2$ hat Vorgänger $\lfloor i/2 \rfloor$.

Kombiniere Heapbedingung, Array-Darstellung, Daten:

Definition 6.4.2

Sei $(U, <)$ Totalordnung. Ein (Teil-)Array $\texttt{A[}1\,..\,k\texttt{]}$ mit Einträgen aus $U \times D$ heißt ein **Min-Heap** (oder **Heap**), wenn der entsprechende linksvollständige Binärbaum mit k Knoten (Knoten i mit $\texttt{A[}i\texttt{].key} \in U$ und $\texttt{A[}i\texttt{].data} \in D$ beschriftet) bezüglich der key-Komponenten (min-)heapgeordnet ist, d. h. wenn gilt:

$$\boxed{\texttt{A[}\lfloor i/2 \rfloor\texttt{].key} \leq \texttt{A[}i\texttt{].key}, \text{ für } 1 < i \leq k.}$$

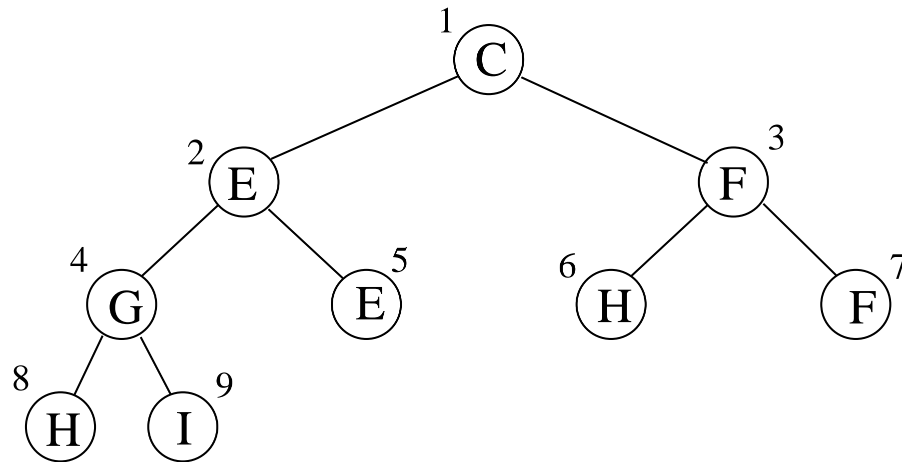
Die Charakterisierung im Kasten erwähnt den (gedachten) Binärbaum nicht mehr explizit.

Analog: **Max-Heap** (benutze Maxheap-Ordnung).

Beispiel: $U = \{\mathrm{A, B, C}, \ldots, \mathrm{Z}\}$ mit der Standardordnung. (Daten weggelassen.)

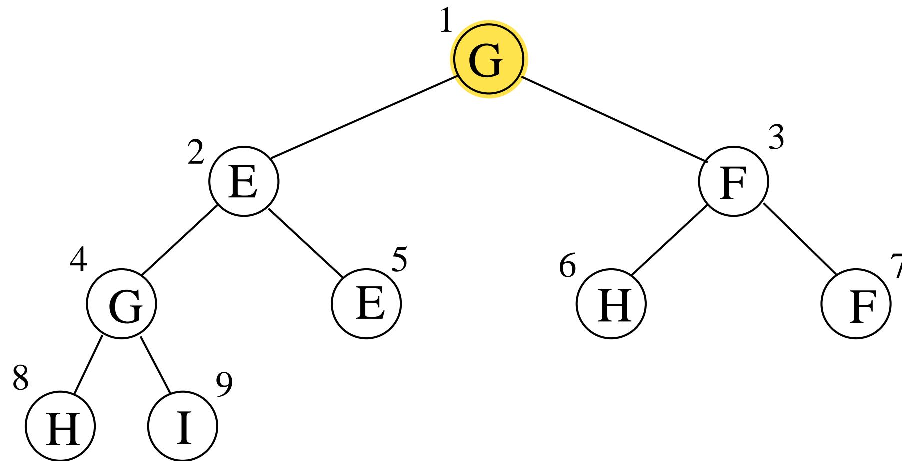
Ein Min-Heap und der zugehörige Baum ($k = 9$):

1	2	3	4	5	6	7	8	9	10	11	12
C	E	F	G	E	H	F	H	I	*	*	*



Aufgabe: Heaps reparieren

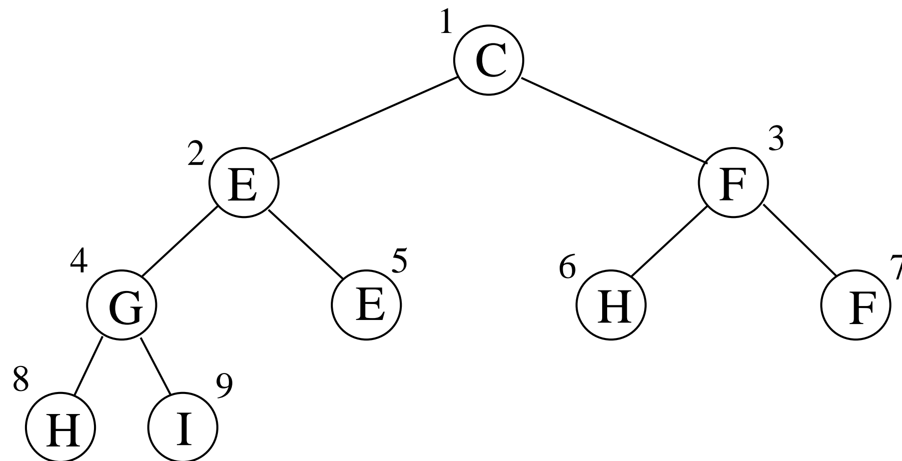
1	2	3	4	5	6	7	8	9	10	11	12
G	E	F	G	E	H	F	H	I	*	*	*



Beobachte: Wurzelschlüssel G z. B. durch C ersetzen liefert Heap.

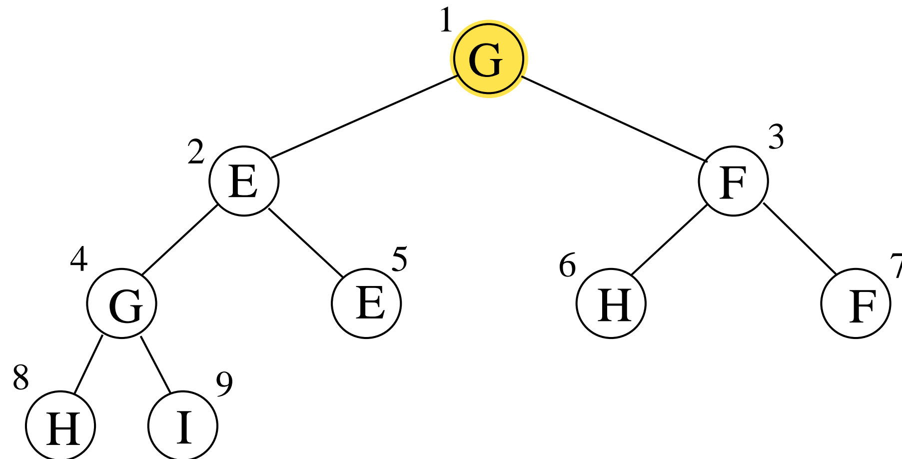
Aufgabe: Heaps reparieren

1	2	3	4	5	6	7	8	9	10	11	12
C	E	F	G	E	H	F	H	I	*	*	*



Beobachte: Wurzelschlüssel G z. B. durch C ersetzen liefert Heap.

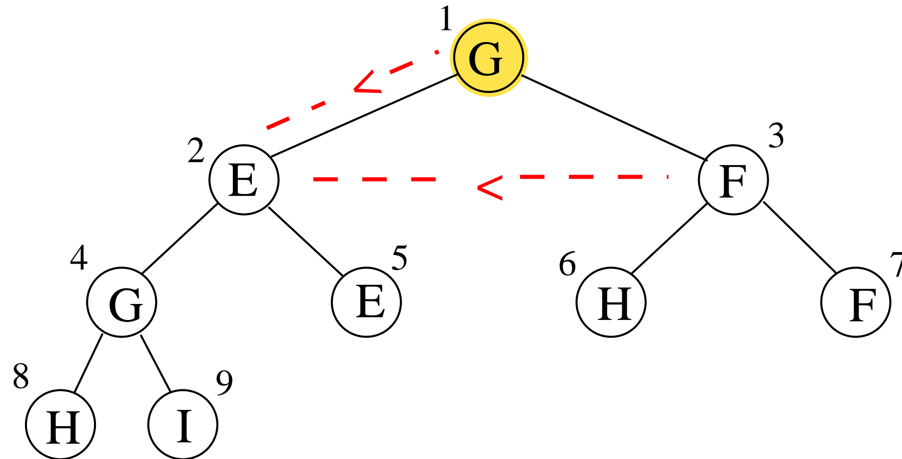
Aufgabe: Heaps reparieren



Beobachte: Wurzelschlüssel G z. B. durch C ersetzen liefert Heap.

Definition: Höchstens `A[j]` **ist zu groß** :⇔ es gibt einen Schlüssel $x \leq$ `A[j].key`, so dass ein Heap entsteht, wenn man `A[j].key` durch x ersetzt.

Heaps reparieren



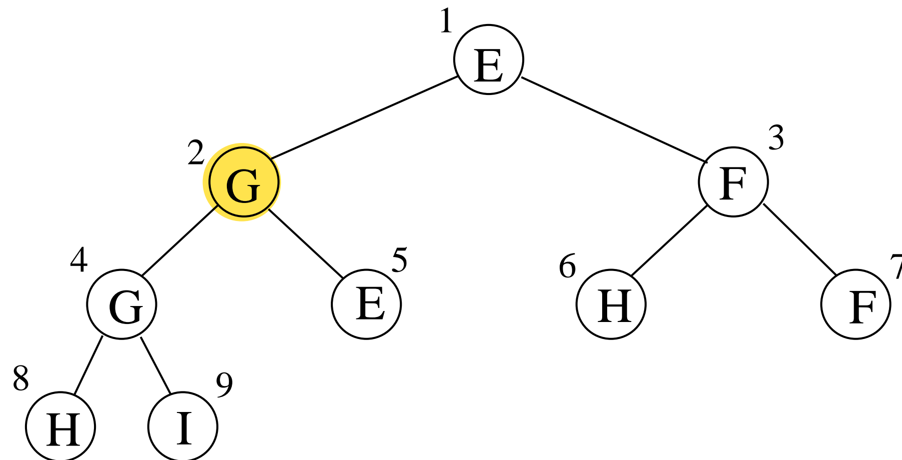
Höchstens `A[1]` ist zu groß: 1 ist „aktueller Knoten“.

Vergleiche die Kinder des aktuellen Knotens.

Vergleiche „kleineres“ Kind mit aktuellem Knoten.

Falls kleiner: **Vertausche** „kleineres“ Kind (E) mit dem aktuellen Knoten (G).

Heaps reparieren



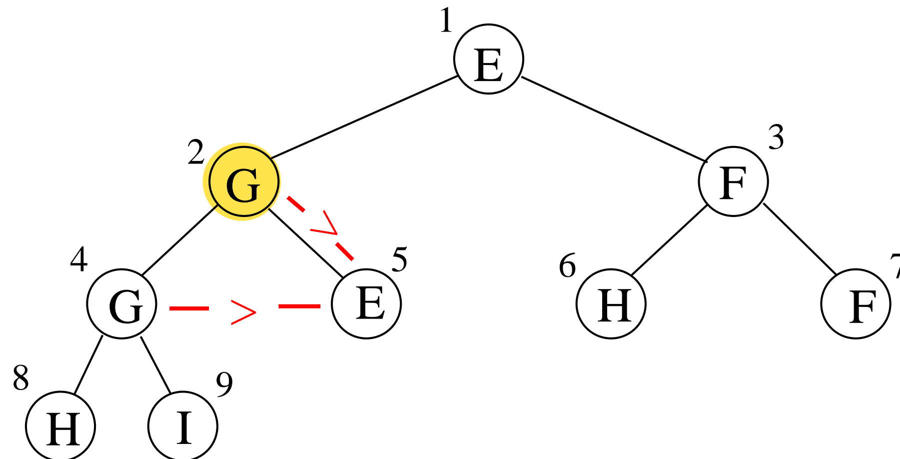
Vergleiche „kleineres“ Kind mit aktuellem Knoten.

Falls kleiner: **Vertausche** „kleineres“ Kind (E) mit dem aktuellen Knoten (G).

Nun: Höchstens `A[2]` ist zu groß. (Ersetze G durch E!)

Heaps reparieren

1	2	3	4	5	6	7	8	9	10	11	12
E	G	F	G	E	H	F	H	I	*	*	*



Höchstens A[2] ist zu groß. (Ersetze G durch E!)

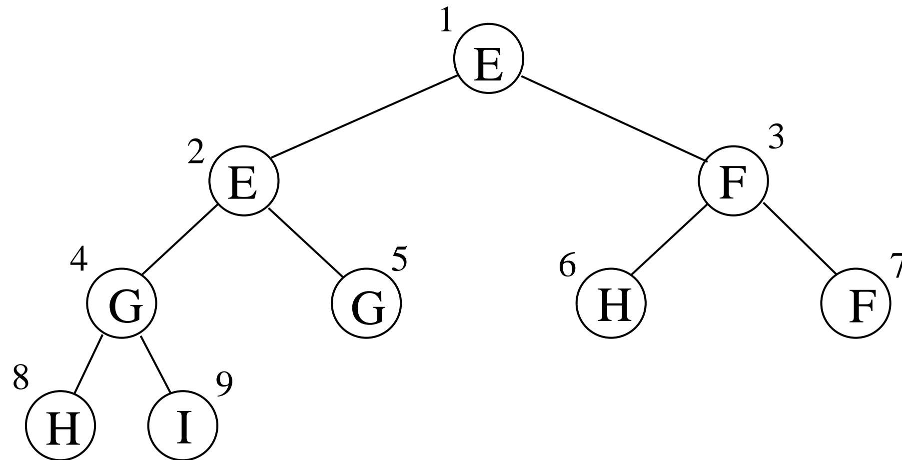
Vergleich der beiden Kinder von Knoten 2.

Vergleich des „kleineren“ Kinds (Knoten 5) mit dem aktuellen Knoten 2.

Vertauschen des „kleineren“ Kinds (Knoten 5) mit dem „aktuellen“ Knoten 2.

Heaps reparieren

1	2	3	4	5	6	7	8	9	10	11	12
E	E	F	G	G	H	F	H	I	*	*	*



Höchstens `A[5]` ist zu groß. (Ersetze G durch E!)

Aktueller Knoten 5 hat keine Kinder $\Rightarrow$ Kein Fehler mehr: Reparatur beendet.

Andere Möglichkeit für Ende:

Eintrag im aktuellen Knoten ist nicht größer als die Einträge in seinen Kindern.

```
bubbleDown(1, k)  // Heap-Reparatur in A[1 . . k]
                  // Vorbedingung: Höchstens A[1] ist zu groß
(1)   j ← 1;        // „aktueller Knoten“
(2)   done ← (2 · j > k);
(3)   while not done do    // (∗) höchstens A[j] ist zu groß, A[j] hat Kind A[2 · j] im Baum
(4)      m ← 2 · j;
(5)      if m + 1 ≤ k and A[m+1].key < A[m].key
(6)         then m ← m + 1;    // nun gilt: A[m] ist einziges oder „kleineres“ Kind von A[j]
(7)      if A[m].key < A[j].key
(8)         then vertausche A[j] mit A[m];
(9)              j ← m;        // neuer „aktueller Knoten“
(10)             done ← (2 · j > k);    // wenn done = false, gilt wieder (∗)
(11)        else done ← true.  // fertig, kein Fehler mehr.
```

Anmerkung: Im Programmtext wird auf die Binärbaumstruktur nicht Bezug genommen (außer indirekt über die Indexberechnungen).

Korrektheit:

Wir wollen zeigen: Wenn höchstens `A[1]` zu groß ist und **bubbleDown**$(1, k)$ durchgeführt wird, dann entsteht ein Heap mit denselben Einträgen wie vorher.

Dass sich die Menge der Einträge nicht ändert, folgt daraus, dass Elemente nur vertauscht werden.

Für die Heapeigenschaft zeigen wir die folgende Induktionsbehauptung **durch Induktion über die Schleifendurchläufe**:

(IB$_j$) Zu Beginn des Durchlaufs, in dem `j` die Zahl j enthält, ist **höchstens** `A[`j`]` **zu groß**.

Wir formulieren den Beweis in der Baum-Sprechweise.

Der Ind.-Anfang ($j = 1$) gilt nach Voraussetzung.

Betrachte Schleifendurchlauf für Knoten j.

1. Fall: In Zeile (8) wird $\mathtt{A}[j]$ mit $\mathtt{A}[2j]$ vertauscht.

Es seien u, v, w die Schlüssel in $\mathtt{A}[j]$, $\mathtt{A}[2j]$ bzw. $\mathtt{A}[2j+1]$ **vor** der Vertauschung.
Nach (IB_j) können wir ein $x \leq u$ wählen, so dass

$(*)$ in $\mathtt{A}[1..k]$ ein Heap entsteht, wenn man das u in $\mathtt{A}[j]$ durch x ersetzt.

Wir wissen nach dem Algorithmus: $v \leq w$ und $v < u$.

Nach der Vertauschung steht u in $\mathtt{A}[2j]$ und v in $\mathtt{A}[j]$.

Für **I.-Schritt** zu zeigen: Es entsteht ein Heap, wenn wir u in $\mathtt{A}[2j]$ durch v ersetzen.

(Weil $v < u$ gilt, liefert das (IB_{2j}).)

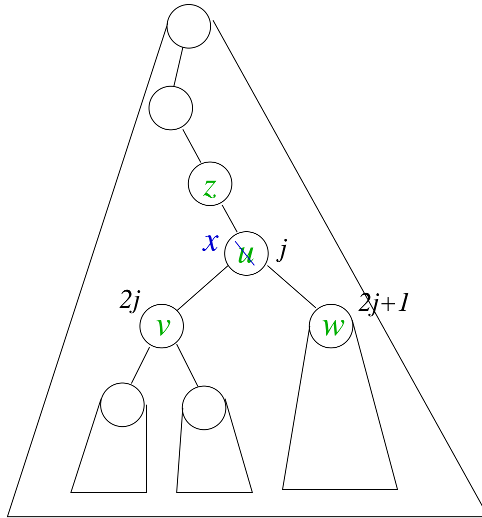
Behauptung: Aus $\texttt{A}[1..k]$ (nach der Vertauschung) entsteht ein Heap, wenn wir $\texttt{A}[2j]$ Schlüssel u durch v ersetzen.

(A)

Vorher, nach I.V.:

Ein Heap

(mit x in $\texttt{A}[j]$).

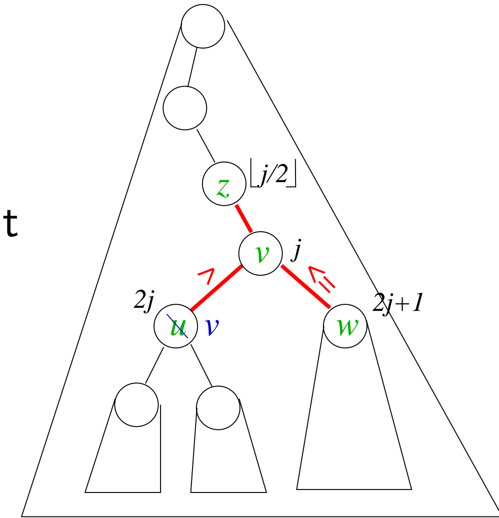


(B)

Nachher:

Ein Heap (mit

v in $\texttt{A}[2j]$)?



Situationen (A) und (B) (mit Ersetzungen $u \rightarrow x$ bzw. $u \rightarrow v$) unterscheiden sich nur in Knoten j.
$\rightarrow$ Müssen in (B) nur Relationen zwischen j, seinem Vorgänger und seinen Kindern überprüfen.

(1) „j vs. $\lfloor j/2 \rfloor$“: Nach I.V. in (A) gilt $z \leq x \leq v$.

(2) „j vs. $2j$“: Die Schlüssel sind beide gleich v.

(3) „j vs. $2j+1$“: Im Algorithmus, Zeile (8), wurden v und w verglichen und es wurde $2j$ als „kleineres Kind“ gewählt. Also gilt $v \leq w$.

2. Fall: In Zeile (7) wird $\texttt{A}[j]$ mit $\texttt{A}[2j+1]$ vertauscht: analog zum 1. Fall.

In jedem Schleifendurchlauf wird der Inhalt von `j` (mindestens) verdoppelt. Daher gilt:
Nach einer Reihe von Schleifendurchläufen endet die Schleife. Dann steht ein j in `j`, für das gilt:

(IB$_j$) und Knoten j hat keine Kinder oder die Schlüssel in den Kindern sind $\geq u = \texttt{A}[j]\texttt{.key}$.

Sei $x \leq u$ so, dass ein Heap entsteht, wenn u durch x ersetzt wird.
Dann ist auch $\texttt{A}[1\,..\,k]$ ohne diese Ersetzung ein Heap, weil die Schlüssel in den Kindern von Knoten j nicht kleiner als u sind und im Fall $j > 1$ der Schlüssel in Knoten $\lfloor j/2 \rfloor$ höchstens $x \leq u$ ist.

Kosten: Der Inhalt j der Variablen `j` ist anfangs 1 und verdoppelt sich in jeder Runde (mindestens); ein Schleifendurchlauf findet nur statt, wenn $2j \leq k$ ist.
Daher: Wenn es s Durchläufe durch die **while**-Schleife gibt, muss $2^s \leq k$ sein, d. h. $s \leq \log k$.
In jedem Durchlauf gibt es maximal zwei Schlüsselvergleiche.
Also ist der Zeitaufwand $O(\log k)$, bei nicht mehr als $2\log k$ Vergleichen. □

Wir haben damit das folgende Ergebnis gezeigt.

Lemma 6.4.3

Wenn anfangs höchstens `A[1]` zu groß ist, stellt **bubbleDown**$(1, k)$ die Heapeigenschaft her. Die Prozedur führt maximal $2 \log k$ Vergleiche durch und hat Laufzeit $O(\log k)$.

Vorlesungsvideo:

Heapsort

Die Idee von Heapsort für ein Array A$[1..n]$ ist nun folgende:

1. (Heapaufbau) Arrangiere A$[1..n]$ so um, dass ein Heap entsteht.

2. (Auswahlphase) Für k von n abwärts bis 2:

Entnimm kleinsten Eintrag A$[1]$ aus dem Heap A$[1..k]$. Stelle mit den in A$[2..k]$ verbliebenen Einträgen in A$[1..k-1]$ wieder einen Heap her.
(In A$[k]$ ist dann Platz für den entnommenen Eintrag.)

Dadurch werden die Arrayeinträge in steigender Reihenfolge entnommen.

Wenn wir den in Runde k entnommenen Eintrag in A$[k]$ speichern, ist am Ende das Array A$[1..n]$ fallend sortiert.

Auswahlphase als Programm:

Prozedur HeapSelect$(1, n)$

(1) **for** `k` **from** n **downto** 2 **do**

// in `A[1]` steht das Minimum von `A[`$1..k$`]`

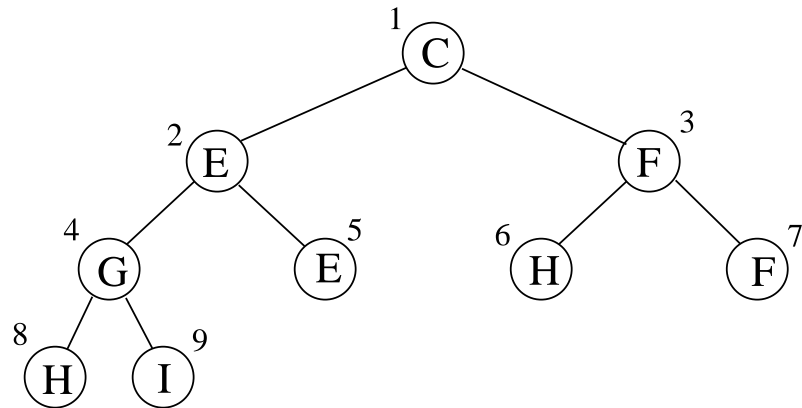
(2) vertausche `A[1]` mit `A[k]`;

// in `A[`$1..k-1$`]` ist höchstens `A[1]` zu groß

(3) **bubbleDown**$(1, \texttt{k} - 1)$; // Heapreparatur

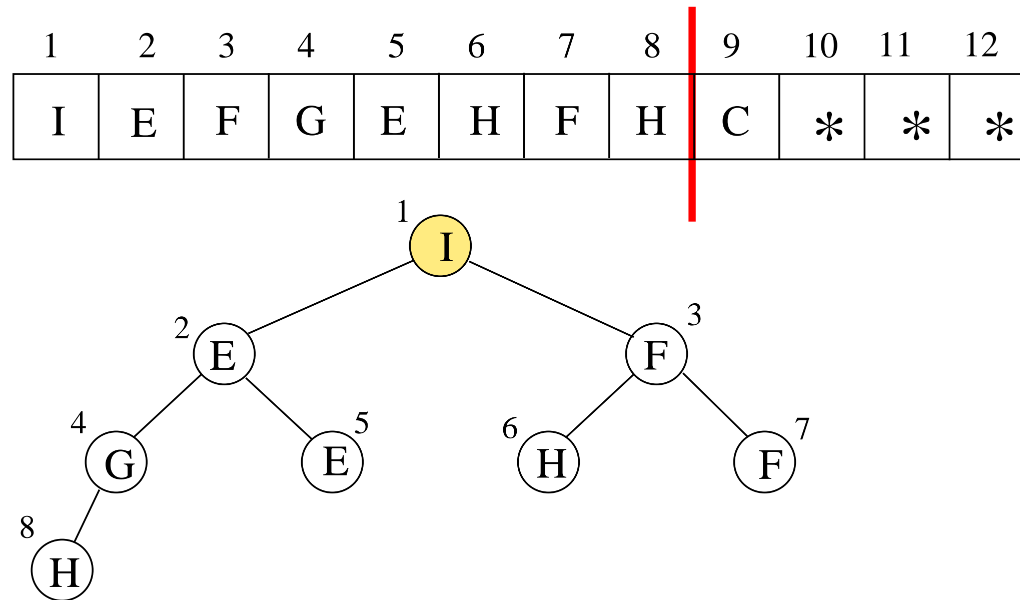
Auswahlphase

1	2	3	4	5	6	7	8	9	10	11	12
C	E	F	G	E	H	F	H	I	*	*	*



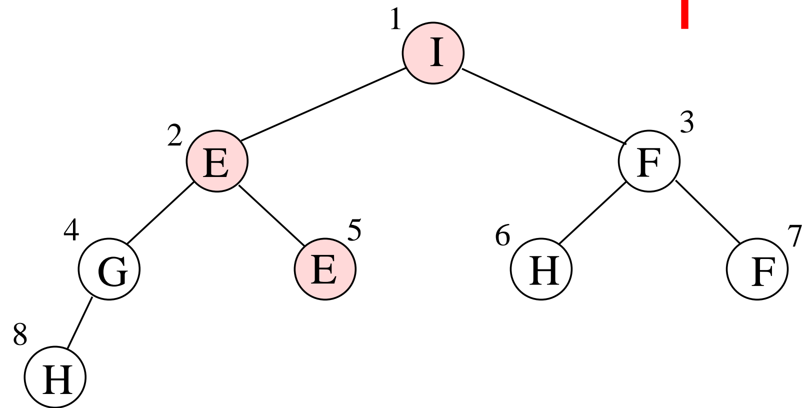
Ein Heap: `A[1..9]`.

Auswahlphase



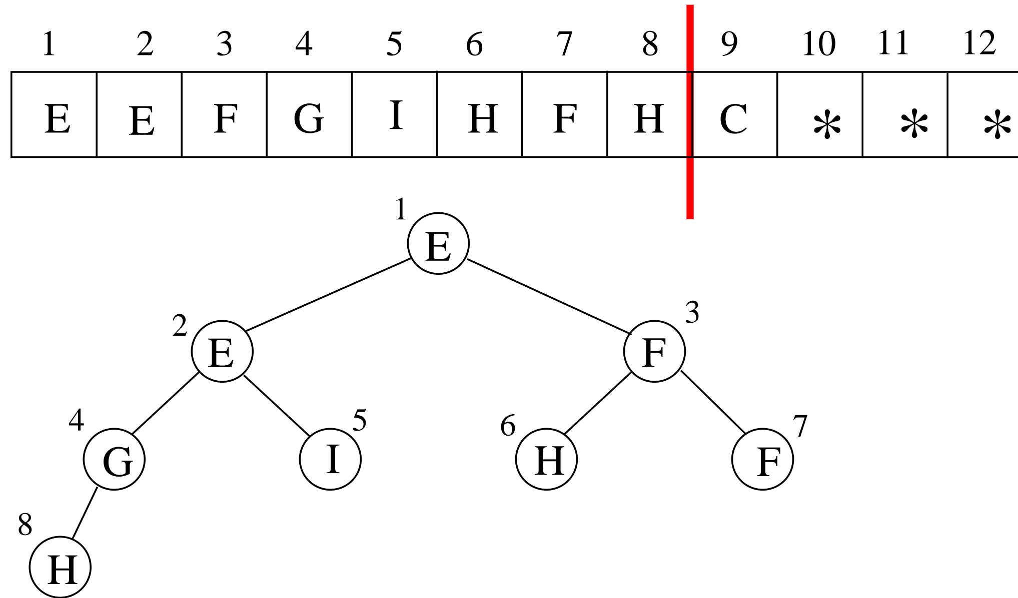
Minimum aus A[1] nach A[9], altes A[9] in die Wurzel.
In A[1..8] ist höchstens A[1] zu groß.

Auswahlphase



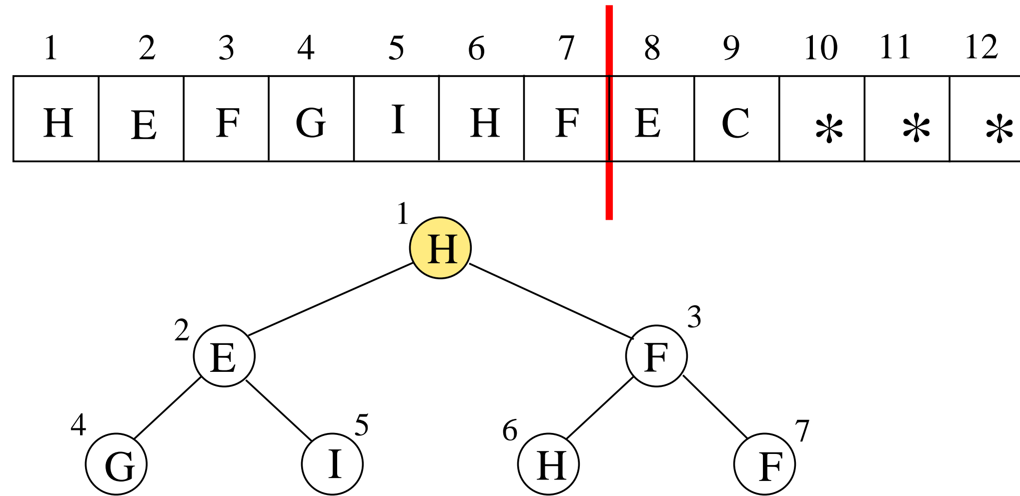
Von **bubbleDown** verfolgter Weg.

Auswahlphase



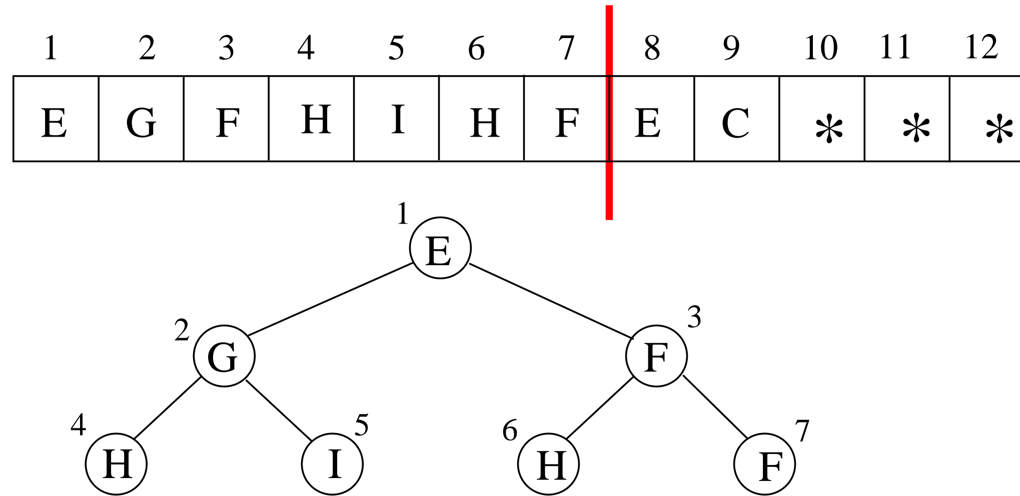
`A`[1..8] ist Heap.

Auswahlphase



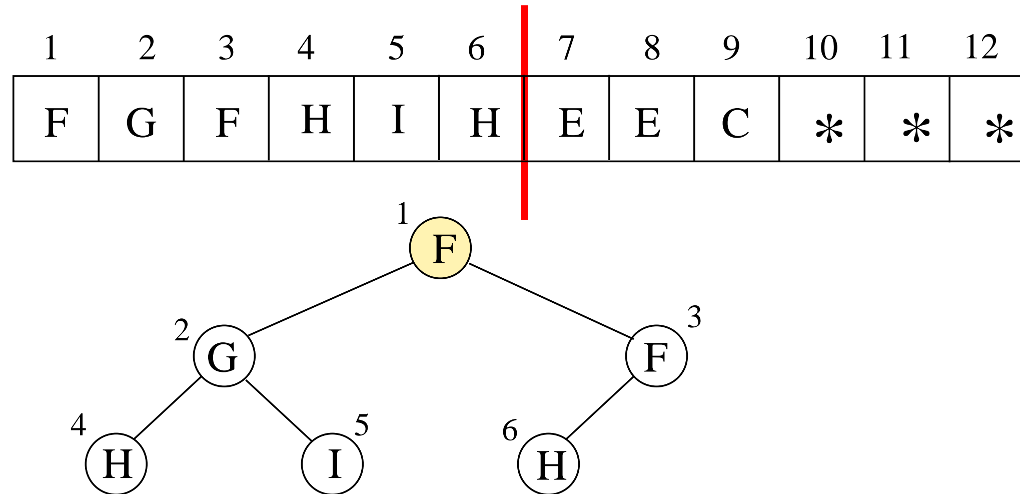
Minimum aus A[1] nach A[8], altes A[8] in die Wurzel.
In A[1..7] ist höchstens A[1] zu groß.

Auswahlphase



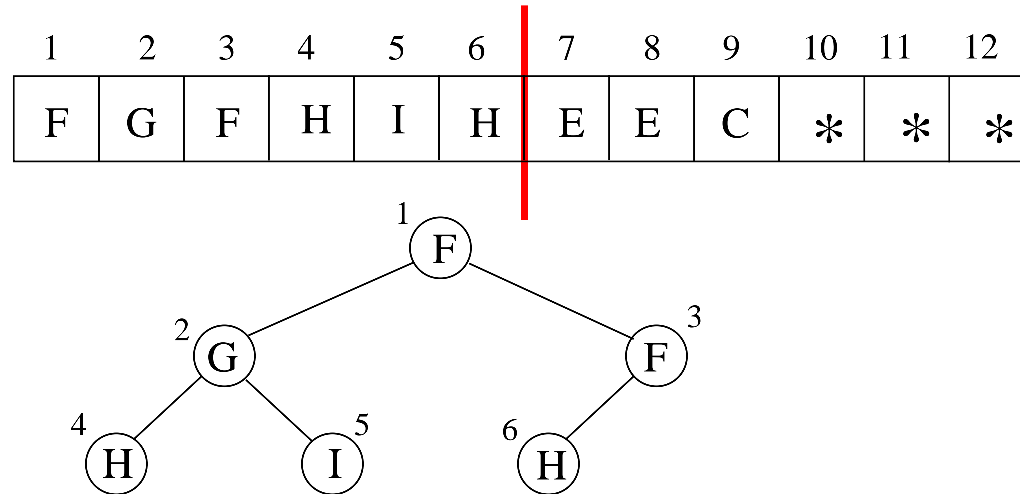
A[1..7] ist Heap.

Auswahlphase



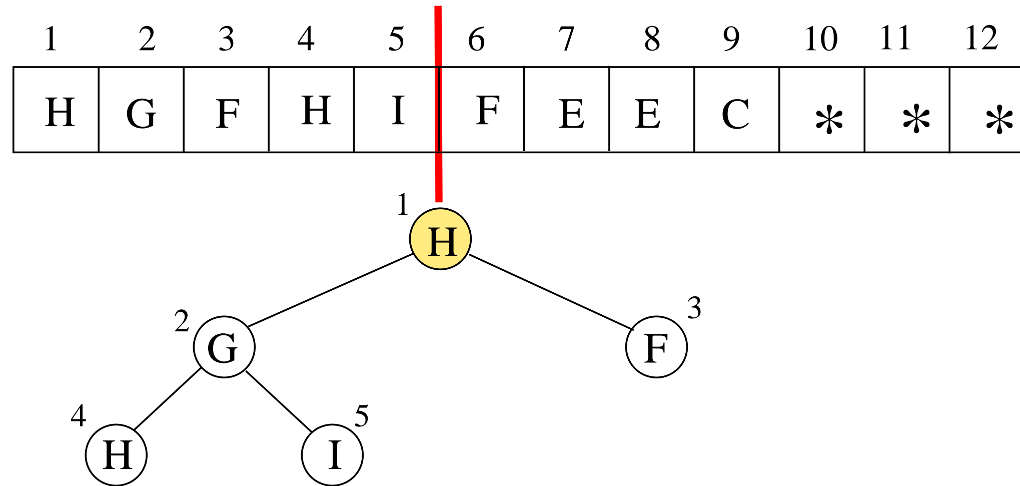
Minimum aus `A[1]` nach `A[7]`, altes `A[7]` in die Wurzel.
In `A[1..6]` ist höchstens `A[1]` zu groß.

Auswahlphase



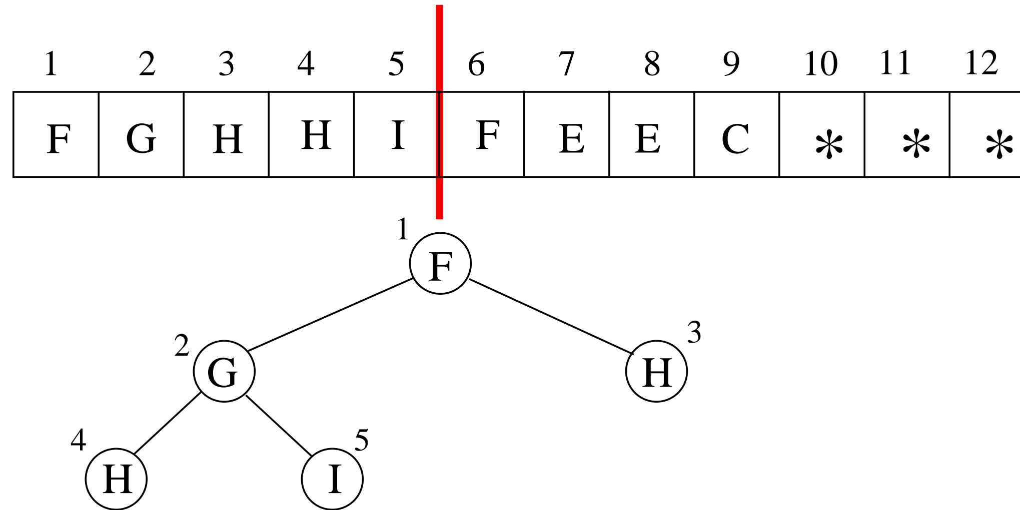
A[1..6] ist Heap.

Auswahlphase



Minimum aus A[1] nach A[6], altes A[6] in die Wurzel.
In A[1..5] ist höchstens A[1] zu groß.

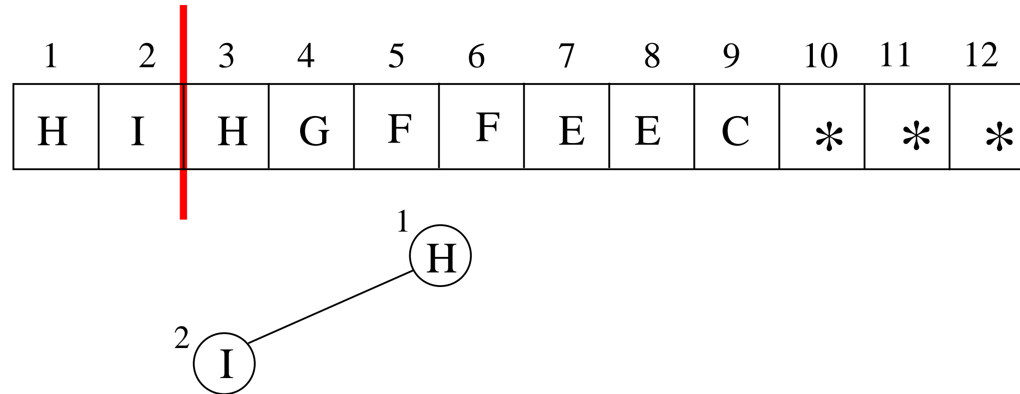
Auswahlphase



A[1..5] ist Heap.

usw.

Auswahlphase



`A`[1..2] ist Heap.

Auswahlphase

1	2	3	4	5	6	7	8	9	10	11	12
I	H	H	G	F	F	E	E	C	*	*	*

Vertauschen von `A[1]` und `A[2]` vollendet fallende Sortierung.

bubbleDown$(1, 1)$ hat keinen Effekt.

Satz 6.4.4

Die Prozedur **HeapSelect**$(1, n)$ führt höchstens $2n \log n$ Vergleiche durch und hat Rechenzeit $O(n \log n)$.

Beweis:

Anzahl Vergleiche: höchstens $\sum_{2 \le k \le n} 2 \log k < 2n \log n$.

Kosten: $\sum_{2 \le k \le n}(O(1) + O(\log k)) \; = O(n) + O(n \log n) = O(n \log n)$. $\square$

Es fehlt noch: **„1. Heapaufbau“.**

Wie verwandelt man ein beliebiges Array $\mathtt{A}[1\,..\,n]$ in einen Heap?

Idee: Betrachte im linksvollständigen Baum mit n Knoten die Teilbäume T_ℓ mit Wurzel ℓ, und stelle in diesen „von unten nach oben“, also in der Reihenfolge $\ell = n, n-1, \ldots, 3, 2, 1$, die Heapbedingung her.

Beobachtung: Wenn $n \geq \ell > n/2$, dann ist $2\ell > n$, also besteht der Teilbaum T_ℓ mit Wurzel ℓ nur aus dem Knoten ℓ und ist daher heapgeordnet.

Nun arbeitet man iterativ die Werte $\ell = \lfloor n/2 \rfloor, \lfloor n/2 \rfloor - 1, \ldots, 2, 1$ ab.

Sei ein solches ℓ gegeben, und seien $T_{2\ell}$ und $T_{2\ell+1}$, die Unterbäume an den Kindern von ℓ, schon heapgeordnet. Das bedeutet: Teilbaum T_ℓ ist heapgeordnet, außer dass eventuell im Wurzelknoten ℓ ein zu großer Schlüssel steht.

Wir wissen schon, wie man das korrigiert: Man wendet **bubbleDown** an, startend bei der Wurzel, die hier ℓ ist.

Also löst die folgende Variante von **bubbleDown** (s. Folie 52) die Aufgabe.

Prozedur bubbleDown(ℓ, n) // Heap-Reparatur im Teilbaum T_ℓ
// Vorbedingung: T_ℓ ist heapgeordnet, außer dass höchstens `A[`ℓ`]` zu groß ist

(1) j $\leftarrow$ ℓ;
(2) done $\leftarrow$ $(2 \cdot \mathtt{j} > n)$;
(3) **while not** done **do**
(4) . . .

Lemma 6.4.6

Wenn in `A[`$\ell + 1\,..\,n$`]` (d. h. in allen Teilbäumen mit Wurzel $> \ell$) die Heapbedingung gilt, dann gilt sie nach Ausführung von **bubbleDown**(ℓ, n) in `A[`$\ell\,..\,n$`]`. Die Anzahl der Vergleiche ist nicht größer als $2\log(n/\ell)$, die Laufzeit $O(\log(n/\ell))$.

Beweis: Korrektheit wie bei Lemma 6.4.3. Anzahl Schlüsselvergleiche ähnlich wie bei Lemma 6.4.3: Wenn es in **bubbleDown**(ℓ, n) s Schleifendurchläufe gibt, muss $2^s \cdot \ell \le n$ sein, d. h. $s \le \log(n/\ell)$. Also gibt es maximal $2\log(n/\ell)$ Vergleiche bei einem Zeitaufwand von $O(\log(n/\ell))$. □

Prozedur makeHeap$(1, n)$ // Transformiert Array A[$1..n$] in einen Heap

```
(1)   for ell from ⌊n/2⌋ downto 1 do
(2)         bubbleDown(ell, n);
```

Lemma 6.4.7

Ein Aufruf **makeHeap**$(1, n)$ wandelt A[$1..n$] in einen Heap um. Die Anzahl der Vergleiche ist nicht größer als $2n$; die Rechenzeit ist $O(n)$.

Beweis der Korrektheit: Mit Induktion über $\ell = \lfloor n/2 \rfloor, \lfloor n/2 \rfloor - 1, \ldots, 2, 1$:
Nach Durchlauf mit ℓ in ell gilt Heapbedingung in A[$\ell..n$]. □

Kosten: Die Anzahl der Vergleiche ist höchstens (mit Lemma 6.4.6):

$$2 \cdot \sum_{1 \le \ell \le n/2} \lfloor \log(n/\ell) \rfloor = 2 \cdot \sum_{1 \le \ell \le n/2} \quad \sum_{j\,:\, j \ge 1 \wedge \ell \cdot 2^j \le n} 1$$

(weil $\lfloor \log(n/\ell) \rfloor = |\{j \mid j \le \log(n/\ell)\}| = |\{j \mid 2^j \le n/\ell\}|$)

$$= 2 \cdot \sum_{1 \le j \le \log n} \sum_{1 \le \ell \le n/2^j} 1$$

(Vertauschung der Summationsreihenfolge)

$$\le 2 \cdot \sum_{1 \le j \le \log n} \frac{n}{2^j} \; < \; 2n \cdot \sum_{1 \le j \le \log n} \frac{1}{2^j}$$

(Geometrische Reihe: $\sum_{j \ge 1} 2^{-j} = 1$.)

$$< 2n.$$

Algorithmus HeapSort($\mathtt{A}[1\,..\,n]$)
(1) **makeHeap**$(1,n)$;
(2) **HeapSelect**$(1,n)$;

Satz 6.4.8

Der Aufruf **HeapSort**($\mathtt{A}[1\,..\,n]$) sortiert $\mathtt{A}[1\,..\,n]$ in fallende Reihenfolge. Die gesamte Rechenzeit ist $O(n\log n)$, die Gesamtzahl der Vergleiche ist kleiner als $2n(\log n+1)$. **HeapSort** arbeitet **in situ**/**in-place** (ist aber **nicht stabil**).

Beweis: Korrektheit und Zeitbedarf folgt aus der Analyse der Teilprozeduren. Die Gesamtzahl der Vergleiche ist nach Satz 6.4.4 und Lemma 6.4.7 höchstens:

$2n+\sum_{2\le k\le n} 2\log k \le 2n+2\cdot\sum_{2\le k\le n}\log k \le 2n+2n\log n.$ $\square$

Anmerkung: Man kann sogar zeigen: $<2n\log n$ Vergleiche genügen, für nicht ganz kleine n.

Bemerkung 1: Wenn Sortierung **in aufsteigende Reihenfolge** gewünscht wird, ersetze in allen Prozeduren und Programmen Schlüsselvergleiche „$\texttt{A}[i] \le \texttt{A}[j]$“ durch „$\texttt{A}[i] \ge \texttt{A}[j]$“ und umgekehrt. Die in $\texttt{A}[1..n]$ entstehende Struktur ist dann ein **Max-Heap**: in der Wurzel des Binärbaumes steht immer das Maximum.

Bemerkung 2: Die **mittlere** Anzahl von Vergleichen, unter der Annahme, dass jede Anordnung der Eingabeobjekte in $\texttt{A}[1..n]$ dieselbe Wahrscheinlichkeit $1/n!$ hat, ist $2n \log n - O(n)$.

Bemerkung 3: Es gibt eine Variante namens „Bottom-Up-Heapsort“, die $n \log n + O(n)$ Vergleiche im **mittleren** Fall hat. – **Idee:** Bei **bubbleDown**$(1, k)$ laufe gesamten „Weg der kleineren Kinder“ bis zum Blatt, ziehe dabei die „kleineren Kinder“ um eine Ebene nach oben. (Nur ein Vergleich pro Ebene.) Füge am Ende das kritische Element aus der Wurzel *von unten nach oben* an die richtige Stelle auf diesem Weg ein. Im durchschnittlichen Fall sind hier nur konstant viele Schritte nach oben nötig.

Bemerkung 4: Bei Variante **„d-Way-Heapsort“**, für $d \geq 3$, sind die Arraypositionen mit $0, 1, \ldots, k-1$ nummeriert. Jeder Knoten $i \in \{0, 1 \ldots, k-1\}$ hat bis zu d viele Kinder, nämlich $di+1, di+2, \ldots, di+d$ (soweit diese Zahlen $< k$ sind).
Der Vorgänger des Knotens $i > 0$ ist $\lfloor (i-1)/d \rfloor$.

Die Heapbedingung in $\texttt{A}[0..k-1]$ wird modifiziert:

$$\boxed{\texttt{A}[\lfloor (i-1)/d \rfloor] \leq \texttt{A}[i], \text{ für } 1 \leq i < k.}$$

Bei **bubbleDown**$(0, k-1)$ wird $\texttt{A}[j]$ mit dem *kleinsten Eintrag in einem Kind* von Knoten j vertauscht, wenn der Schlüssel in diesem Kind kleiner ist als $\texttt{A}[j]\texttt{.key}$.
(Die Entscheidung erfordert bis zu d viele Vergleiche pro Ebene.)

Vorteil: Der Baum hat nur Tiefe $\log_d n = (\log n)/(\log d)$. Daher gibt es bei **bubbleDown** Zugriff nur auf $(\log n)/(\log d)$ zusammenhängende Segmente im Array. Das führt zu viel weniger Cache-Fehlern als bei gewöhnlichem Heapsort und ist daher schneller. (Experimentell stellt man fest, dass besonders $d = 4$ zu attraktiven Verbesserungen führt.)

Übung: Man formuliere die Details und beweise die Korrektheit von **bubbleDown**$(0, k-1)$.

Vorlesungsvideo:

Prioritätswarteschlangen

6.5 Datentyp: Prioritätswarteschlange

oder **Vorrangwarteschlangen**

In einer Prioritätswarteschlange werden Objekte aufbewahrt, die mit einem Schlüssel aus einem totalgeordneten Universum $(U, <)$ versehen sind. (Oft sind die Schlüssel Zahlen.) Objekte werden **eingefügt** und **entnommen**.

Einfügungen sind beliebig möglich.

Beim Entnehmen wird immer ein Eintrag mit **minimalem Schlüssel** gewählt.

Idee: Schlüssel entsprechen hier **„Prioritäten“** – je *kleiner* der Schlüssel, desto *höher* die Priorität. Wähle stets einen Eintrag mit höchster Priorität!

Der Datentyp Prioritätswarteschlage ist mit **Heaps** effizient zu realisieren.

Wir diskutieren hier nur **Binärheaps**, aber man könnte genauso gut auch Bäume mit größerem Ausgangsgrad benutzen, etwas 4-Wege-Heaps oder allgemein k-Wege-Heaps.

Beispiel: Datensatz: (L, k), $L \in \{\mathtt{A}, \ldots, \mathtt{Z}\}$, $k \in \mathbb{N}$, Schlüssel ist der Buchstabe L.

Operation	innerer Zustand	Ausgabe
empty	$\emptyset$	–
isempty	$\emptyset$	$true$
insert$(\mathtt{M}, 1)$	$\{(\mathtt{M}, 1)\}$	–
insert$(\mathtt{K}, 2)$	$\{(\mathtt{M}, 1), (\mathtt{K}, 2)\}$	–
insert$(\mathtt{M}, 3)$	$\{(\mathtt{M}, 1), (\mathtt{K}, 2), (\mathtt{M}, 3)\}$	–
insert$(\mathtt{R}, 4)$	$\{(\mathtt{M}, 1), (\mathtt{K}, 2), (\mathtt{M}, 3), (\mathtt{R}, 4)\}$	–
insert$(\mathtt{T}, 5)$	$\{(\mathtt{M}, 1), (\mathtt{K}, 2), (\mathtt{M}, 3), (\mathtt{R}, 4), (\mathtt{T}, 5)\}$	–
extractMin	$\{(\mathtt{M}, 1), (\mathtt{M}, 3), (\mathtt{R}, 4), (\mathtt{T}, 5)\}$	$(\mathtt{K}, 2)$
extractMin	$\{(\mathtt{M}, 1), (\mathtt{R}, 4), (\mathtt{T}, 5)\}$	$(\mathtt{M}, 3)$
insert$(\mathtt{R}, 6)$	$\{(\mathtt{M}, 1), (\mathtt{R}, 4), (\mathtt{T}, 5), (\mathtt{R}, 6)\}$	–
insert$(\mathtt{M}, 2)$	$\{(\mathtt{M}, 1), (\mathtt{R}, 4), (\mathtt{T}, 5), (\mathtt{R}, 6), (\mathtt{M}, 2)\}$	–
extractMin	$\{(\mathtt{R}, 4), (\mathtt{T}, 5), (\mathtt{R}, 6), (\mathtt{M}, 2)\}$	$(\mathtt{M}, 1)$
extractMin	$\{(\mathtt{R}, 4), (\mathtt{T}, 5), (\mathtt{R}, 6)\}$	$(\mathtt{M}, 2)$
isempty	$\{(\mathtt{R}, 4), (\mathtt{T}, 5), (\mathtt{R}, 6)\}$	$false$

Beispiel: Datensatz: (L, k), $k \in \mathbb{N}$, $L \in \{\mathtt{A}, \ldots, \mathtt{Z}\}$, Schlüssel ist der Buchstabe L.

Operation	innerer Zustand	Ausgabe
empty	$\emptyset$	–
isempty	$\emptyset$	*true*
insert$(\mathtt{M}, 1)$	$\{(\mathtt{M}, 1)\}$	–
insert$(\mathtt{K}, 2)$	$\{(\mathtt{M}, 1), (\mathtt{K}, 2)\}$	–
insert$(\mathtt{M}, 3)$	$\{(\mathtt{M}, 1), (\mathtt{K}, 2), (\mathtt{M}, 3)\}$	–
insert$(\mathtt{R}, 4)$	$\{(\mathtt{M}, 1), (\mathtt{K}, 2), (\mathtt{M}, 3), (\mathtt{R}, 4)\}$	–
insert$(\mathtt{T}, 5)$	$\{(\mathtt{M}, 1), (\mathtt{K}, 2), (\mathtt{M}, 3), (\mathtt{R}, 4), (\mathtt{T}, 5)\}$	–
extractMin	$\{(\mathtt{M}, 1), (\mathtt{M}, 3), (\mathtt{R}, 4), (\mathtt{T}, 5)\}$	$(\mathtt{K}, 2)$
extractMin	$\{(\mathtt{M}, 1), (\mathtt{R}, 4), (\mathtt{T}, 5)\}$	$(\mathtt{M}, 3)$
insert$(\mathtt{R}, 6)$	$\{(\mathtt{M}, 1), (\mathtt{R}, 4), (\mathtt{T}, 5), (\mathtt{R}, 6)\}$	–
insert$(\mathtt{M}, 2)$	$\{(\mathtt{M}, 1), (\mathtt{R}, 4), (\mathtt{T}, 5), (\mathtt{R}, 6), (\mathtt{M}, 2)\}$	–
extractMin	$\{(\mathtt{R}, 4), (\mathtt{T}, 5), (\mathtt{R}, 6), (\mathtt{M}, 2)\}$	$(\mathtt{M}, 1)$
extractMin	$\{(\mathtt{R}, 4), (\mathtt{T}, 5), (\mathtt{R}, 6)\}$	$(\mathtt{M}, 2)$
isempty	$\{(\mathtt{R}, 4), (\mathtt{T}, 5), (\mathtt{R}, 6)\}$	*false*

Einfügereihenfolge hat keine festgelegte Beziehung zur Ausgabereihenfolge.

Anwendungen:

1) **Prozessverwaltung in Multitask-System**
Jeder Prozess hat einen Namen (eine „ID“) und eine Priorität (Zahl in $\mathbb{N}$).
(**Kleinere Zahlen** bedeuten **höhere Prioritäten**.)
Aktuell rechenbereite Prozesse befinden sich in der **Prozesswarteschlange**. Bei Freiwerden eines Prozessors (z. B. durch Unterbrechen eines Prozesses) soll einer der Prozesse **mit höchster Priorität** aus der Warteschlange entnommen und weiter ausgeführt werden.

2) **„Discrete Event Simulation“**
System von „Aktionen“ oder „Ereignissen“ soll auf dem Rechner simuliert werden.
(Anwendungsbeispiel: Technisches System mit mehreren kommunizierenden Komponenten.)
Jeder **ausführbaren** Aktion A ist ein Zeitpunkt $t_A \in [0, \infty)$ zugeordnet.
Die Ausführung einer Aktion A kann neue Aktionen A' erzeugen oder auch schon vorhandene Aktionen A' früher ausführbar machen, mit neuen Zeitpunkten $t_{A'} > t_A$.
Ein Schritt: Wähle diejenige noch nicht ausgeführte Aktion *mit dem frühesten Ausführungszeitpunkt* und führe sie aus.

3) Innerhalb von **Algorithmen** (Dijkstra, Jarník/Prim (später), geometrische Algorithmen („scan-line“)).

Datentyp: SimplePriorityQueue (Einfache Prioritätswarteschlange)

1. Signatur:

Sorten: *Keys*
Data
PrioQ // „Priority Queues“
Boolean

Operationen: $empty\colon \to PrioQ$
$isempty\colon PrioQ \to Boolean$
$insert\colon PrioQ \times Keys \times Data \to PrioQ$
$extractMin\colon PrioQ \to PrioQ \times Keys \times Data$

Mathematisches Modell:

Sorten:

Keys : $(U, <)$ // totalgeordnetes „Universum“

Data : D // Menge von „Datensätzen“

Boolean : $\{false, true\}$

PrioQ : die Menge aller endlichen **Multimengen*** $P \subseteq U \times D$

Operationen:

empty$(\,) = \emptyset$ // die *leere Multimenge*

$$isempty(P) = \begin{cases} true, & \text{für } P = \emptyset \\ false, & \text{für } P \neq \emptyset \end{cases}$$

* **Multimenge**: Elemente dürfen mehrfach vorkommen, wie z. B. in $\{1, 1, 4, 5, 5, 5, 9\}$.

Mathematisches Modell (Forts.):

insert$(P, x, d) = P \cup \{(x, d)\}$ (als Multimenge),

$$\textit{extractMin}(P) = \begin{cases} \textit{undefiniert}, & \text{wenn } P = \emptyset \\ (P', x_0, d_0), & \text{wenn } P \neq \emptyset, \end{cases}$$

für ein Element (x_0, d_0) in P mit minimalem Schlüssel x_0, und $P' = P - \{(x_0, d_0)\}$ (als Multimenge).

Standardimplementierung: (Binäre) Heaps

Andere Implementierungen sind möglich!

Implementierung: Objekt mit Array $\texttt{A}[1..m]$ (für die Einträge) und Pegel $\texttt{n}$ für die aktuelle Zahl n von Einträgen.

Überlaufprobleme werden ausgeklammert. (Verdoppelungsstrategie, falls nötig.)

empty(m):
Lege Array $\texttt{A}[1..m]$ an;
Jeder Eintrag kann ein Paar $(\texttt{key}, \texttt{data})$ aufnehmen.
$\texttt{n} \leftarrow 0$.

// Zeitaufwand: $O(1)$ oder $O(m)$.

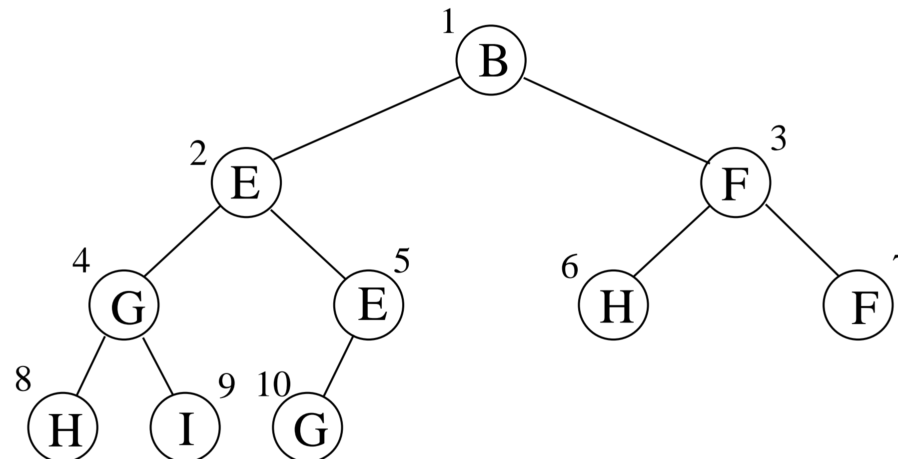
isempty(): **return** $(\texttt{n} = 0)$; // Zeitaufwand: $O(1)$

extractMin: Implementierung von *extractMin*(P):

Ein Eintrag mit minimalem Schlüssel steht in der Wurzel, d. h. in Arrayposition 1.

Entnehme `A[1]` (hier „B“) und gib es aus.

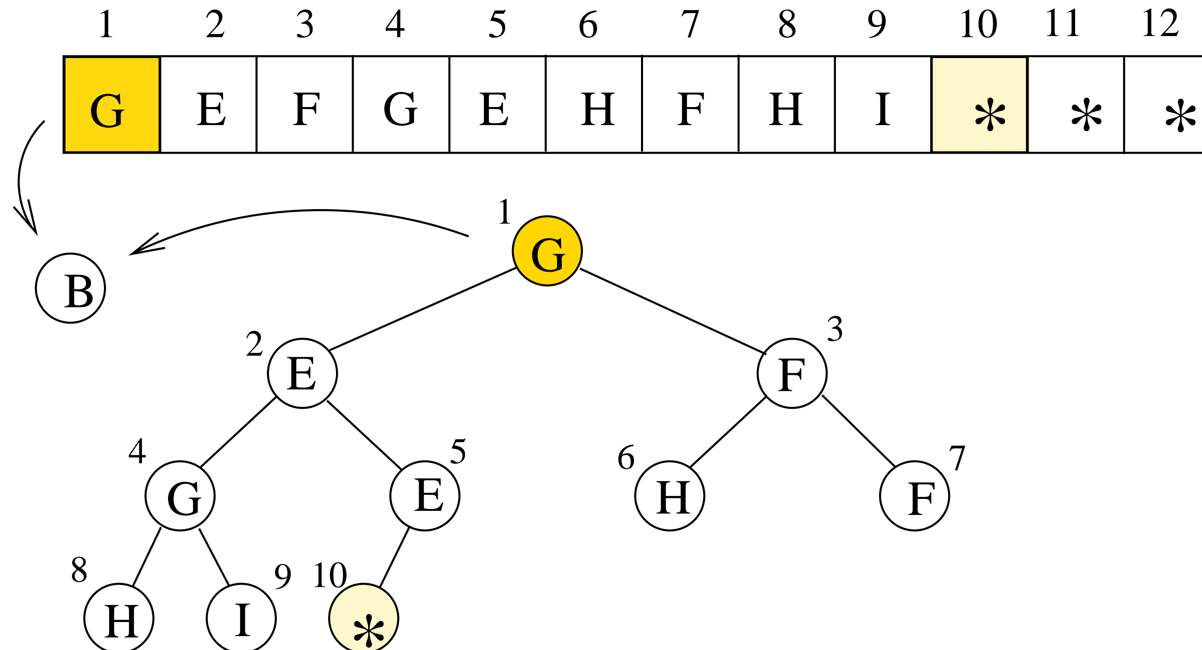
1	2	3	4	5	6	7	8	9	10	11	12
B	E	F	G	E	H	F	H	I	G	*	*



extractMin: Implementierung von *extractMin*(P):

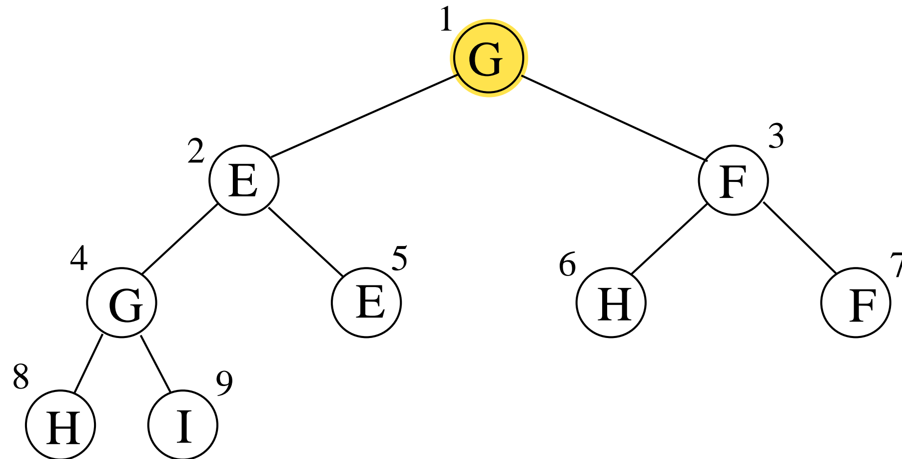
Ein Eintrag mit minimalem Schlüssel steht in der Wurzel, d. h. in Arrayposition 1.

Entnehme `A[1]` (hier „B“) und gib es aus.



„Loch“ im Binärbaum – „Stopfen“ mit dem letzten Eintrag.

Bleibt Aufgabe: Heap reparieren



Höchstens `A[1]` ist zu groß, also: **bubbleDown** hilft hier!

Prozedur extractMin()

// Entnehmen eines minimalen Eintrags aus Priority-Queue
// Ausgangspunkt: Pegelstand n (in n), $\texttt{A}[1..n]$ ist Heap, $n \geq 1$

(0) **if** $\texttt{n} = 0$ **then** „Fehlerbehandlung“;
(1) $\texttt{x} \leftarrow \texttt{A[1].key}$; $\texttt{d} \leftarrow \texttt{A[1].data}$;
(2) $\texttt{A[1]} \leftarrow \texttt{A[n]}$;
(3) n--;
(4) **if** $\texttt{n} > 1$ then **bubbleDown**$(1, \texttt{n})$;
(5) **return** $(\texttt{x}, \texttt{d})$;

Korrektheit: klar wegen Korrektheit von **bubbleDown**$(1, \texttt{n})$.

Zeitaufwand: $O(\log(n))$, wobei n die aktuelle Größe des Heaps ist.

Beachte: Benutzer der Datenstruktur hat keinen Einfluss auf die Reihenfolge der Ausgabe von Objekten mit identischem Schlüssel.

Implementierung von **insert**(x, d):

Voraussetzung: A[$1..n$] ist Heap; $1 \leq n < m$; n enthält n.

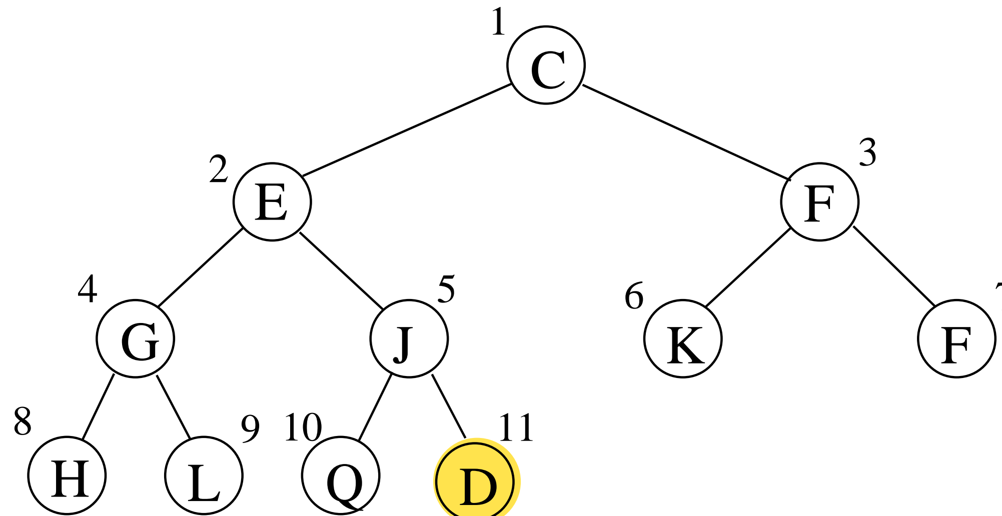
n++;
A[n] $\leftarrow (x, d)$.

An der Stelle $n' = n + 1$ (neuer Pegelstand) ist nun eventuell die Heapeigenschaft gestört, **weil** x **zu klein ist**.

Definition

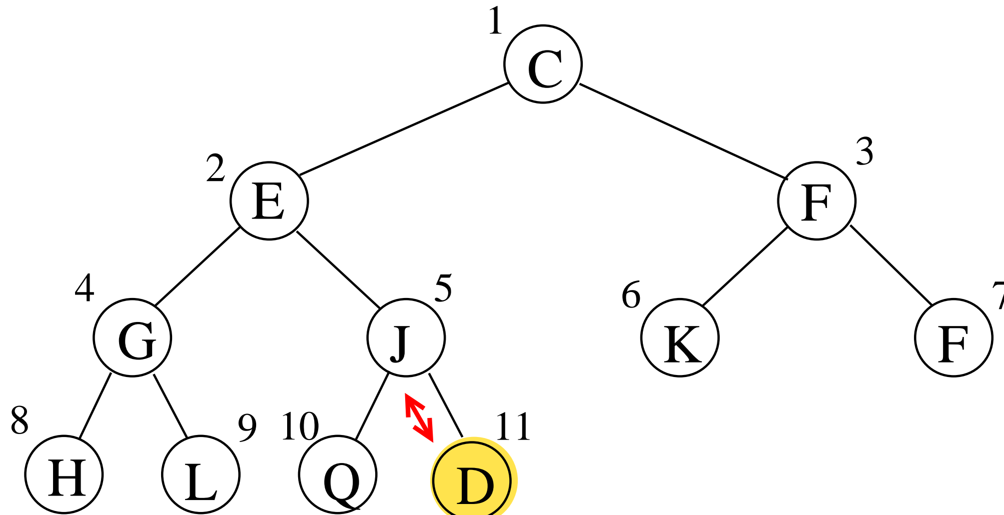
Höchstens A[j] ist zu klein (in A[$1..n$]) :$\Leftrightarrow$
in A[$1..n$] entsteht ein Heap, wenn man $u =$ A[j].key durch einen geeigneten Schlüssel $x \geq u$ ersetzt.

Einfügen von „D“ an Stelle $n = 11$.



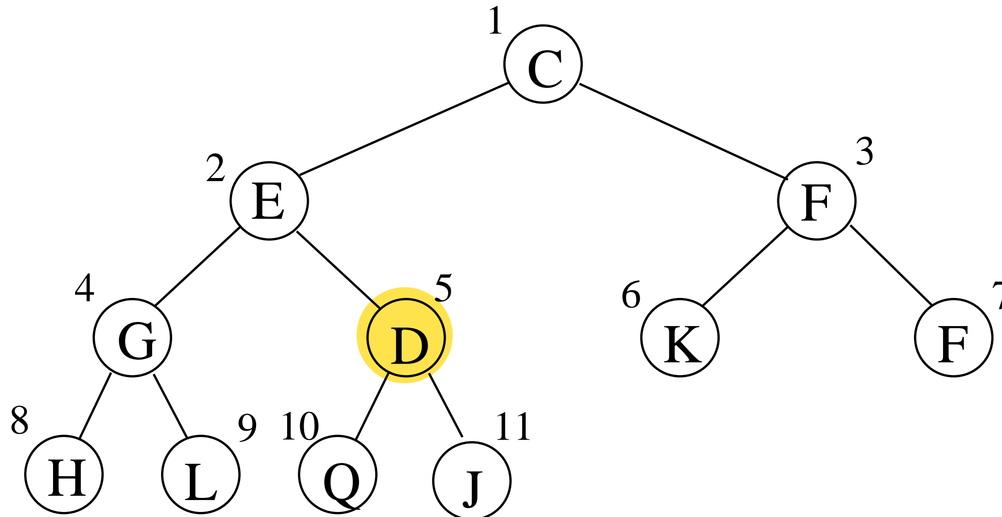
Reparatur, wenn höchstens ein Schlüssel zu klein: **bubbleUp**.

1	2	3	4	5	6	7	8	9	10	11	12
C	E	F	G	J	K	F	H	L	Q	D	*



Reparatur, wenn höchstens ein Schlüssel zu klein: **bubbleUp**.

1	2	3	4	5	6	7	8	9	10	11	12
C	E	F	G	D	K	F	H	L	Q	J	*



und so weiter . . .

Prozedur bubbleUp(i) // Heapreparatur ab `A[`i`]` nach oben

(1) $\texttt{j} \leftarrow i$;

(2) $\texttt{h} \leftarrow \lfloor \texttt{j}/2 \rfloor$;

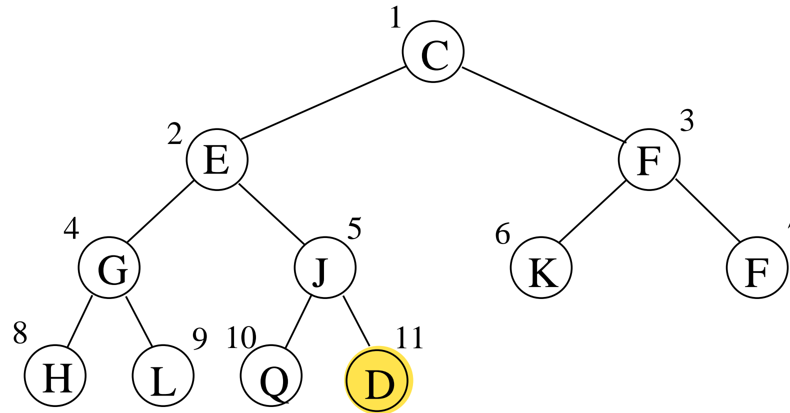
(3) **while** $\texttt{h} \geq 1$ **and** $\texttt{A[h].key} > \texttt{A[j].key}$ **do**

(4) vertausche `A[j]` mit `A[h]`;

(5) $\texttt{j} \leftarrow \texttt{h}$;

(6) $\texttt{h} \leftarrow \lfloor \texttt{j}/2 \rfloor$.

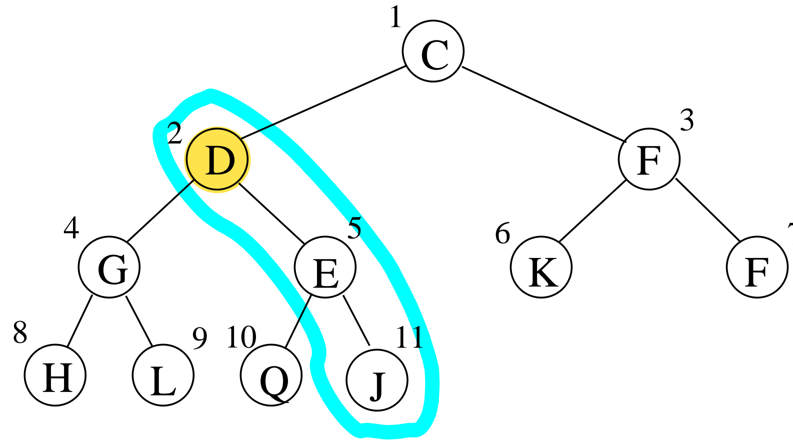
1	2	3	4	5	6	7	8	9	10	11	12
C	E	F	G	J	K	F	H	L	Q	D	*



Anschaulich: Auf dem Weg von `A[`i`]` zur Wurzel werden alle Elemente, deren Schlüssel größer als der (alte) Eintrag in `A[`i`]`) ist, um eine Position (auf dem Weg) nach unten gezogen.

Eintrag `A[`i`]` landet in der freigewordenen Position.

Man kann dies auch effizienter programmieren (wie in Insertionsort, Folie 7 in Kapitel 1).



Anschaulich: Auf dem Weg von `A`[i] zur Wurzel werden alle Elemente, deren Schlüssel größer als der (alte) Eintrag in `A`[i]) ist, um eine Position (auf dem Weg) nach unten gezogen.

Eintrag `A`[i] landet in der freigewordenen Position.

Man kann dies auch effizienter programmieren (wie in Insertionsort, Folie 7 in Kapitel 1).

Behauptung 6.5.1

In `A`$[1..n]$ ist höchstens `A`$[i]$ zu klein

$\Rightarrow$ nach Ausführung von **bubbleUp**(i) ist `A`$[1..n]$ ein Heap.

Der Zeitaufwand ist $O(\log i)$, die Anzahl der Schlüsselvergleiche ist höchstens das Level von i im Baum, also die Anzahl der Bits in der Binärdarstellung $\text{bin}(i)$, d. h. $\lceil \log(i+1) \rceil$.

Beweis: Ähnlich wie bei **bubbleDown** (siehe Druckfolien).

Beweis der Korrektheit von **bubbleUp**(i): Durch Induktion über die Runden zeigt man:

(IB$_j$) Zu Beginn des Schleifendurchlaufs für j ist höchstens $\texttt{A}[j]$ zu klein.

Zu Beginn, für $j = i$, stimmt dies nach Annahme über die Eingabe.
Nun betrachte einen Schleifendurchlauf der **while**-Schleife. Die I.V. sagt, dass ein Heap entsteht, wenn man u in Knoten j durch einen geeigneten Schlüssel $x \geq u$ ersetzt.

1. Fall: $j \geq 2$ und $u = \texttt{A}[j]\texttt{.key} < v = \texttt{A}[\lfloor j/2 \rfloor]\texttt{.key}$; es wird getauscht.
Nun steht u in Knoten $\lfloor j/2 \rfloor$ und v im Kind j.

Beh.: Wenn man den Schlüssel u in $\lfloor j/2 \rfloor$ auf v erhöht, entsteht ein Heap (also gilt (IB$_{\lfloor j/2 \rfloor}$)).

„Alte Situation": x in Knoten j und v in Knoten $\lfloor j/2 \rfloor$: Ist Heap nach I.V., also gilt $x \geq v$.
„Neue Situation": v in Knoten j und v in Knoten $\lfloor j/2 \rfloor$. (Zu zeigen: Ist Heap.)

Kann durch die Änderung von x auf v in Knoten j die Heapeigenschaft verloren gehen? Sicher nicht zwischen j und $\lfloor j/2 \rfloor$ (derselbe Schlüssel v!). Die Schlüssel in Knoten $2j$ und $2j+1$ (falls sie existieren) sind $\geq x$, und es gilt $x \geq v$ (nach I.V.), also gilt die Heapbedingung auch hier.

2. Fall: $j = 1$ oder $u = \texttt{A}[j]\texttt{.key} \geq v = \texttt{A}[\lfloor j/2 \rfloor]\texttt{.key}$, der Algorithmus endet.
(IB$_j$) besagt, dass wir $u = \texttt{A}[j]\texttt{.key}$ durch ein $x \geq u$ ersetzen können, so dass $\texttt{A}[1..k]$ Heap wird. Dann liegt aber auch mit Schlüssel u in Knoten j schon ein Heap vor, weil es keinen Konflikt mit dem Vorgänger von Knoten j gibt. □

Prozedur insert(x, d)

// Einfügen eines neuen Eintrags in Priority-Queue

// Ausgangspunkt: Pegel `n` enthält n, `A`$[1\,..\,n]$ ist Heap.

(1) **if** `n` $= m$ **then** „Überlaufbehandlung“; // z. B. Verdopplung

(2) `n++`;

(3) `A[n]` $\leftarrow (x, d)$;

(4) **bubbleUp**(`n`).

Korrektheit: Klar wegen Korrektheit von **bubbleUp**. **Zeitaufwand:** $O(\log n)$.

Wir können **bubbleUp**(i) sogar für eine etwas allgemeinere Operation verwenden (Korrektheitsbeweis gilt weiter):

Wir ersetzen einen beliebigen Schlüssel im Heap (Position i) durch einen kleineren.

Mit **bubbleUp**(i) kann die Heapeigenschaft wieder hergestellt werden.

Prozedur decreaseKey(x, i)
// (Erniedrigen des Schlüssels an Arrayposition i auf x)
(1) **if** $\texttt{A}[i]\texttt{.key} < x$ **then** Fehlerbehandlung;
(2) $\texttt{A}[i]\texttt{.key} \leftarrow x$;
(3) **bubbleUp**(i).

// Zeitaufwand: $O(\log(i))$

SimplePriorityQueue plus „decreaseKey“:

Prioritätswarteschlange

Satz 6.5.2

Der Datentyp “Prioritätswarteschlange” kann mit Hilfe eines Heaps implementiert werden.

Dabei erfordern *empty* und *isempty* (und das **Ermitteln** des kleinsten Eintrags) konstante Zeit

und *insert*, *extractMin* und *decreaseKey* benötigen jeweils Zeit $O(\log n)$.

Dabei ist n jeweils der aktuelle Pegelstand, also die Anzahl der Einträge in der Prioritätswarteschlange.

Technisches Problem, noch zu klären:

Wie soll man der Datenstruktur „mitteilen“, **welches Objekt gemeint ist**, wenn *decreaseKey* auf einen Eintrag angewendet werden soll?

Bei (binärem) Heap: Die Positionen der Einträge im Array ändern sich die ganze Zeit, durch die durch **insert** und **extractMin** verursachten Verschiebungen im Array.

Technisch unsauber wäre es, dem Benutzer der Datenstruktur stets mitzuteilen, an welcher Stelle im Array ein Eintrag sitzt.
Dies widerspräche eklatant dem Prinzip der Kapselung eines Datentyps. (Im Kontext von Algorithmen wird allerdings manchmal so verfahren, dass der benutzende Algorithmus Zugriffsrechte in die Datenstruktur hinein hat.)

Damit zusammenhängend: Die **Spezifikation** des Datentyps „Prioritätswarteschlange“ ist nicht ganz offensichtlich.

Besser: Erweiterung der Datenstruktur: Objekt (x, d) erhält bei Ausführung von **insert**(x, d) eine **eindeutige** und **unveränderliche „Identität“** p zugeteilt, über die dieses Objekt angesprochen werden kann.
(Damit können sogar **verschiedene Kopien** ein und desselben Datensatzes **unterschieden** werden!)

p wird dem Benutzer als Reaktion auf die Einfügung mitgeteilt.

Beispiel: Datensatz = Schlüssel = Eintrag aus $\{\mathtt{A}, \ldots, \mathtt{Z}\}$.

Wir vergeben **Nummern** $1, 2, 3, \ldots$ als Identitäten. Der **Benutzer** der Datenstruktur muss sicherstellen, dass Identitäten **nicht mehr benutzt** werden, nachdem das entsprechende Objekt (durch ein *extractMin*) wieder aus der Datenstruktur verschwunden ist.

Runde	Operation	innerer Zustand	Ausgabe
0	*empty*	$\emptyset$	–
1	*insert*(D)	{(1,D)}	1
2	*insert*(B)	{(1,D), (2,B)}	2
3	*insert*(D)	{(1,D), (2,B), (3,D)}	3
4	*isempty*	{(1,D), (2,B), (3,D)}	*false*
5	*insert*(E)	{(1,D), (2,B), (3,D), (4,E)}	4
6	*insert*(G)	{(1,D), (2,B), (3,D), (4,E), (5,G)}	5
7	*decreaseKey*(5,D)	{(1,D), (2,B), (3,D), (4,E), (5,D)}	–
⋮	⋮	⋮	⋮

Runde	Operation	innerer Zustand	Ausgabe
⋮	⋮	⋮	⋮
		$\{(1,\texttt{D}), (2,\texttt{B}), (3,\texttt{D}), (4,\texttt{E}), (5,\texttt{D})\}$	
8	*extractMin*	$\{(1,\texttt{D}), (3,\texttt{D}), (4,\texttt{E}), (5,\texttt{D})\}$	$(2,\texttt{B})$
9	*extractMin*	$\{(1,\texttt{D}), (4,\texttt{E}), (5,\texttt{D})\}$	$(3,\texttt{D})$
10	*decreaseKey*$(4,\texttt{B})$	$\{(1,\texttt{D}), (4,\texttt{B}), (5,\texttt{D})\}$	–
11	*extractMin*	$\{(1,\texttt{D}), (5,\texttt{D})\}$	$(4,\texttt{B})$
12	*insert*$(\texttt{F})$	$\{(1,\texttt{D}), (5,\texttt{D}), (3,\texttt{F})\}$	3
13	*decreaseKey*$(3,\texttt{C})$	$\{(1,\texttt{D}), (5,\texttt{D}), (3,\texttt{C})\}$	–
14	*extractMin*	$\{(1,\texttt{D}), (5,\texttt{D})\}$	$(3,\texttt{C})$
15	*extractMin*	$\{(1,\texttt{D})\}$	$(5,\texttt{D})$
16	*extractMin*	$\emptyset$	$(1,\texttt{D})$
17	*isempty*	$\emptyset$	true.

Die in Runde 9 freigewordene Identität 3 wird in Runde 12 wieder vergeben. In Runde 13 wäre zum Beispiel die Operation *decreaseKey*$(4,\texttt{A})$ **nicht legal**, da das (vorher existierende) Objekt mit Identität 4 in Runde 11 aus der Datenstruktur verschwunden ist.

Das Beispiel ist als Vorbild für ein mathematisches Modell geeignet.

Technische Realisierung (für Interessierte):

Allgemein: Jedes im Heap gespeicherte Objekt ((Daten, Schlüssel)-Paar plus Zusatzinformationen) steht während seiner gesamten Lebensdauer an einer festen, unveränderlichen Stelle im Speicher. Zugriffe auf ein solches Objekt erfolgen über eine Referenz (bzw. einen Zeiger). Im Heaparray werden nur Referenzen/Zeiger auf diese Objekte gehalten.

Version 1: Die Identität p **ist** die Referenz/der Zeiger auf das „echte" Objekt im Speicher.
Das Objekt enthält seine aktuelle Position im Heap-Array als Komponente. (Diese muss bei **bubbleUp** und **bubbleDown** stets mit aktualisiert werden.) – **Nachteil:** Die Kapselung der Datenstruktur ist durchbrochen, da der Benutzer über den Zeiger/die Referenz direkt auf die Objekte zugreifen kann.

Version 2: Jedes Objekt erhält eine beliebige (laufende) Nummer (z. B. durch Durchzählen der Einfügungen). Diese Nummer ist dann die Identität p des Objekts. Die Datenstruktur hält in einem („privaten", also internen) Hilfsarray für jede dieser Nummern die Referenz auf das entsprechende Objekt. Das Objekt selbst enthält neben Daten und Schlüssel auch seine aktuelle Position im Heaparray. – **Nachteil:** Der Platzbedarf des Hilfsarrays entspricht der Anzahl der Einfügungen, nicht der maximalen Größe der Prioritätswarteschlange.

Version 3: Wie Version 2, aber die Identität p eines durch **extractMin** aus dem Heap gelöschten Eintrags kann wieder vergeben werden. Dann muss die Datenstruktur zusätzlich die verfügbaren Identitäten verwalten, zum Beispiel mit einem Stack, und Heapeinträge enthalten die Identität.

Datentyp: PriorityQueue (Prioritätswarteschlange)

1. Signatur:

Sorten: $Keys$
$Data$
Id // „Identitäten"
$PrioQ$
$Boolean$

Operationen: $empty\colon\ \to PrioQ$
$isempty\colon PrioQ \to Boolean$
$insert\colon PrioQ \times Keys \times Data \to PrioQ \times Id$
$extractMin\colon PrioQ \to PrioQ \times Id \times Keys \times Data$
$decreaseKey\colon PrioQ \times Id \times Keys \to PrioQ$

Mathematisches Modell:

Sorten:

Keys	:	$(U,<)$	// totalgeordnetes „Universum“
Data	:	D	// Menge von „Datensätzen“
Id	:	$\mathbb{N} = \{0,1,2,\ldots\}$	// unendliche Menge möglicher „Identitäten“
Boolean	:	$\{false, true\}$	
PrioQ	:	die Menge aller Funktionen $f\colon X \to U \times D$, wobei $X = \mathit{Def}(f) \subseteq \mathbb{N}$ endlich ist	

Operationen:

empty$() = \emptyset$ // die *leere Funktion*

isempty$(f) = \begin{cases} true, & \text{für } f = \emptyset \\ false, & \text{für } f \neq \emptyset \end{cases}$

insert$(x, d, f) = (f \cup \{(p, (x, d))\}, p)$, für ein $p \in \mathbb{N} - \mathrm{Def}(f)$

extractMin$(f) = \begin{cases} \textit{undefiniert}, & \text{für } f = \emptyset \\ (f', p_0, x_0, d_0), & \text{für } f \neq \emptyset, \end{cases}$ wobei im zweiten Fall
$x_0 = \min\{x \mid \exists p\, \exists d\colon f(p) = (x, d)\}$
und $f(p_0) = (x_0, d_0)$
und $f' := f - \{(p_0, (x_0, d_0))\}$.

decreaseKey$(p, x) = (f - \{(p, (y, d))\}) \cup \{(p, (x, d))\}$,
falls $f(p) = (y, d)$ mit $x \leq y$ (sonst Fehler)

Implementierung: Größere Übungsaufgabe.

Vorlesungsvideo:

Untere Schranken

6.6 Untere Schranke für Sortieren

Ein Sortieralgorithmus $\mathcal{A}$ heißt ein **Schlüsselvergleichsverfahren**, wenn

in $\mathcal{A}$ auf Schlüssel x, y aus U nur Operationen

$$x < y\ ? \quad x \leq y\ ? \quad x = y\ ?$$

sowie Verschieben und Kopieren angewendet werden. Schlüssel sind also „atomare Objekte“, die als Semantik nur ihre Rolle in der Ordnung $(U, <)$ haben.

Gegensatz wäre: Die Binärdarstellung eines zahlenwertigen Schlüssels wird gelesen und verwendet.

Mergesort, Quicksort, Heapsort, Insertionsort sind Schlüsselvergleichsverfahren.

Wir haben gesehen: Diese Verfahren benötigen $O(n \log n)$ oder $O(n^2)$ Vergleiche.

Ziel in diesem Abschnitt:

Schlüsselvergleichsverfahren können **nicht schneller als** in $\boldsymbol{\Omega(n \log n)}$ Vergleichen sortieren.

Sei $\mathcal{A}$ ein Schlüsselvergleichsverfahren.

Wir denken uns $\mathcal{A}$ auf die $n!$ verschiedenen Eingaben $(a_1, 1), \ldots, (a_n, n)$ angewendet, wobei $\sigma = (a_1, \ldots, a_n)$ eine Permutation von $\{1, \ldots, n\}$ ist.

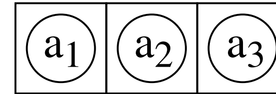
Der Sortierschlüssel ist dabei die erste Komponente. (Die zweite wird nur „mitgeschleift".)

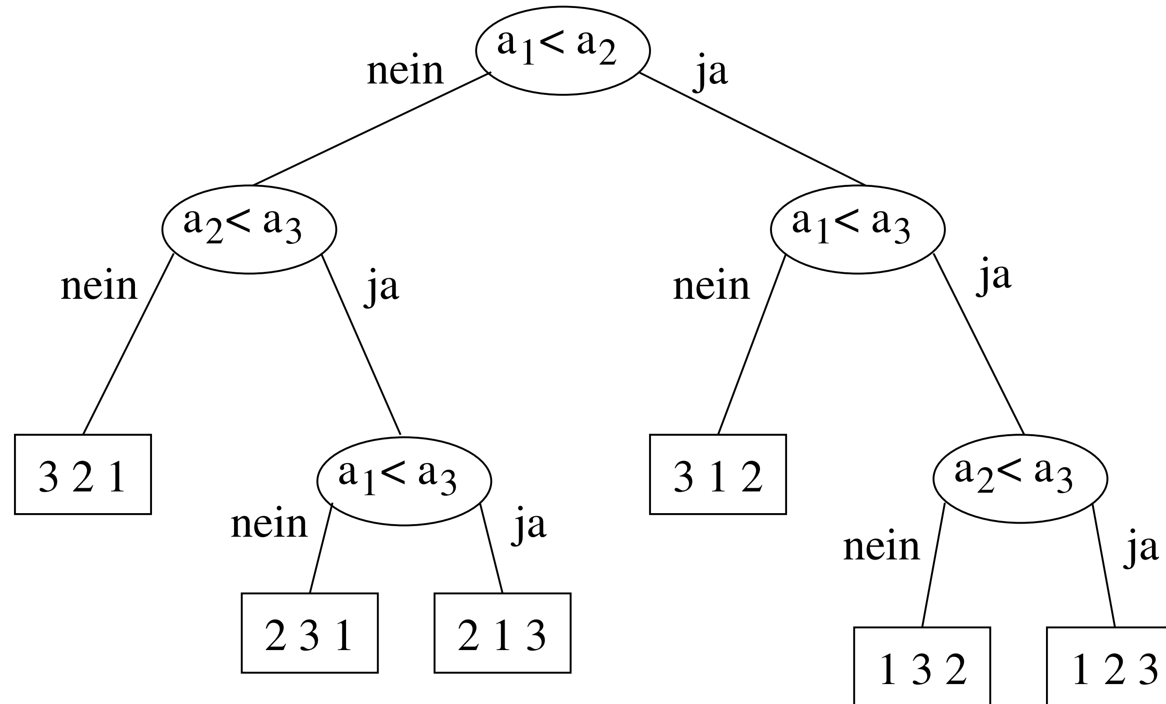
Beispiel: Aus $(2, 1), (4, 2), (1, 3), (3, 4)$ wird $(1, 3), (2, 1), (3, 4), (4, 2)$.

Effekt: In den zweiten Komponenten steht die Permutation $\pi = \sigma^{-1}$ (im Beispiel $\pi = (3, 1, 4, 2)$), die die Eingabe $(a_1, \ldots, a_n)$ sortiert: $a_{\pi(1)} < \cdots < a_{\pi(n)}$.

Daraus folgt: Algorithmus $\mathcal{A}$ erzeugt auf den genannten $n!$ verschiedenen Eingaben $n!$ verschiedene Ausgaben, eine für jede Anordnung σ der Eingabeobjekte.

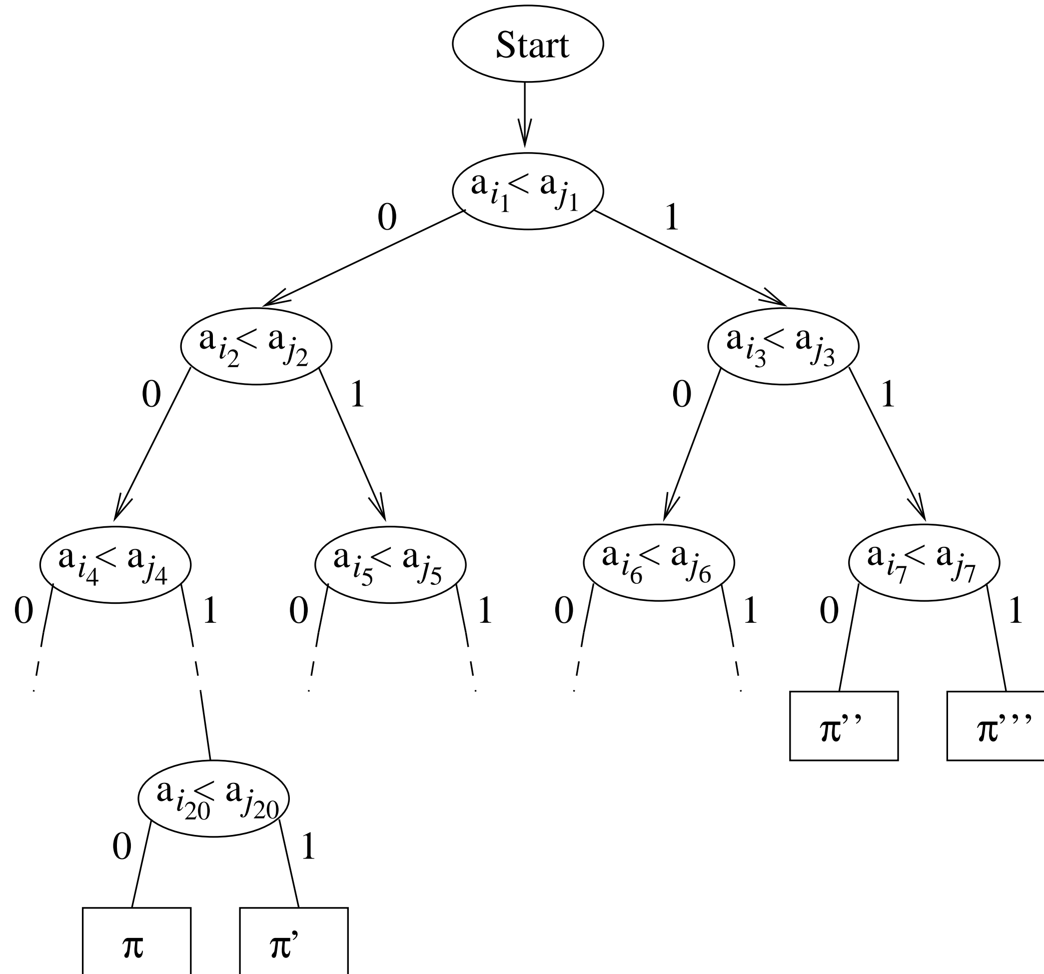
Beispiel: Alle Verläufe für **Mergesort** mit drei Elementen.





In Blättern: die zweiten Komponenten

Allgemein: Ausführen von Algorithmus $\mathcal{A}$ auf allen diesen Inputs führt zu einem **Vergleichsbaum** $\boldsymbol{T_{\mathcal{A},n}}$ mit genau $n!$ Blättern. („0“ bedeutet „nein“, „1“ bedeutet „ja“.)



Allgemein: Jeder vergleichsbasierte Sortieralgorithmus $\mathcal{A}$ erzeugt für jedes n einen **Vergleichsbaum** $\boldsymbol{T_{\mathcal{A},n}}$ mit genau $n!$ Blättern. Die Wege im Baum von der Wurzel zu den Blättern entsprechen den Berechnungen von $\mathcal{A}$ auf den $n!$ genannten Eingaben; die Knoten auf einem solchen Weg entsprechen den in der Berechnung ausgeführten Vergleichen.

Nur Vergleiche „$a_i < a_j$?“ sind überhaupt relevant, wenn wir uns auf die $n!$ vielen vorher beschriebenen Eingaben mit n verschiedenen Komponenten konzentrieren.

Für jede Teilmenge J aller dieser Inputs definieren wir durch Induktion einen Baum T_J, wie folgt.

Wenn J nur ein Element $((a_1, 1), \ldots, (a_n, n))$ hat, dann ist T_J ein mit π beschriftetes Blatt, wobei π die Permutation ist, die $(a_1, \ldots, a_n)$ sortiert.

Wenn J zwei oder mehr Elemente hat, lassen wir $\mathcal{A}$ auf allen Inputs aus J ablaufen, bis zum ersten Vergleich „$a_i < a_j$?“, der nicht bei allen Eingaben in J zu demselben Ergebnis in $\{\text{ja}, \text{nein}\}$ führt. Weil $\mathcal{A}$ vergleichsbasiert ist, gibt es keine anderen Verzweigungen, also arbeitet $\mathcal{A}$ auf allen Eingaben in J bis zu diesem Punkt genau gleich.

Wir bilden $J_0 := \{x \in J \mid \text{Test „}a_i < a_j\text{?“ bei Input } x \text{ liefert „nein“}\}$ und $J_1 := J - J_0$. Baum T_J besteht aus einer Wurzel, die mit „$a_i < a_j$?“ beschriftet ist, sowie T_{J_0} als linkem Unterbaum und T_{J_1} als rechtem Unterbaum.

Man sieht dann, dass jeder Input $(a_1, 1), \ldots, (a_n, n)$ zu genau einem Blatt gehört, und die Knoten auf dem Weg von der Wurzel zu diesem Blatt den Vergleichen entsprechen, die auf diesem Input ausgeführt werden.

Satz 6.6.1

Wenn $\mathcal{A}$ ein Schlüsselvergleichsverfahren ist, das das Sortierproblem für beliebige Inputs mit n Komponenten löst, dann gibt es eine Eingabe $(a_1, \ldots, a_n)$ (eine Permutation von $\{1, \ldots, n\}$), auf der $\mathcal{A}$ mindestens $\lceil \log(n!) \rceil$ Vergleiche ausführt.
Dies sind $n \log n - O(n)$ viele.

Beweis: Der Vergleichsbaum, den $\mathcal{A}$ erzeugt, hat $n!$ (externe) Blätter, also hat er Tiefe $\geq \lceil \log(n!) \rceil$ (s. Prop. 3.3.2(d)). Wir zeigen gleich: $\log(n!) = n \log n - O(n)$.$\square$

Bemerkung:

$n! \geq (n/e)^n$. *Beweis*: $n^n/n! < \sum_{i\geq 0} \frac{n^i}{i!} = e^n$. □

Also: $\log(n!) \geq \log((n/e)^n) = n \log n - n \log e = \boldsymbol{n \log n} - 1{,}44269\ldots n$.

Andersherum: $\ln(n!) = \sum_{1\leq i\leq n} \ln i \leq \ln n + \int_1^n \ln x \, dx =$
$\ln n + [x(\ln x - 1)]_1^n = \ln n + n \ln n - n + 1$, also $n! \leq (en)(n/e)^n$.

Mitteilung:

Für $n \geq 1$ gilt (**Stirlingsche Formel**):

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \leq n! \leq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\frac{1}{12n}}$$

Beispiel: $3598695{,}62 < 10! = 3628800 < 3628810{,}05$.

Bemerkung: Die obere Schranke ist der bessere Schätzwert.

Satz 6.6.2

Sei $\mathcal{A}$ ein Schlüsselvergleichsverfahren, das das Sortierproblem für alle Inputs mit n Komponenten löst, dann gilt für die **mittlere Vergleichsanzahl $\overline{V}_n$** (gemittelt über die $n!$ verschiedenen Anordnungen von $\{1, \ldots, n\}$, jede gleich wahrscheinlich):

$$\overline{V}_n \geq \log(n!).$$

Beweis: Der Vergleichsbaum $T_{\mathcal{A},n}$, den $\mathcal{A}$ erzeugt, hat $N = n!$ (externe) Blätter.

$T_{\mathcal{A},n}$ hat N externe Knoten $\Rightarrow$ **TEPL**$(T_{\mathcal{A},n}) \geq N \log N$, nach Prop. 3.3.5.

Also ist die **mittlere äußere Weglänge** $\frac{1}{N}\text{TEPL}(T_{\mathcal{A},n})$ mindestens $\log N$.

Daraus: Die mittlere Anzahl von Vergleichen ist $\geq \log N = \log(n!)$. □

Der nächste Satz besagt, dass randomisierte Sortierverfahren wie randomisiertes Quicksort zwar Vorteile haben, insbesondere worst-case-Inputs vermeiden können, aber keine Handhabe bilden, um an der unteren Schranke $\log(n!)$ vorbeizukommen.

Satz 6.6.3

Wenn $\mathcal{A}$ ein Schlüsselvergleichsverfahren ist, das das Sortierproblem für alle Inputs mit n Komponenten löst und dabei Randomisierung benutzt (d. h. Zufallsexperimente ausführt wie etwa Randomisiertes Quicksort), dann **gibt es eine Eingabe** $a = (a_1, \ldots, a_n)$ (nämlich eine Permutation von $\{1, \ldots, n\}$), so dass die **erwartete** Vergleichsanzahl von $\mathcal{A}$ auf a (**gemittelt über die Zufallsexperimente** von $\mathcal{A}$) mindestens $\log(n!)$ ist.

Beweis: (Nicht prüfungsrelevant, für Interessierte.) Man kann sich vorstellen, dass man alle Zufallsexperimente, die in $\mathcal{A}$ überhaupt vorkommen können, vor Beginn des Ablaufs in einem großen zusammenfassenden Zufallsexperiment ausführt. (Beispiel **randomisiertes Quicksort**: Für jedes Paar (a, b), $1 \leq a < b \leq n$, wird **vorab** zufällig eine Position $s_{(a,b)} \in [a..b]$ gewählt. Wenn dann im Ablauf des Algorithmus im Teilarray $\mathtt{T}[a..b]$ ein Pivot gewählt werden muss, benutzt man $s_{(a,b)}$.)

Ergebnis ist ein (möglicherweise sehr langes) Zufallswort α.
Jedes solche α hat eine Wahrscheinlichkeit p_α. Dabei gilt $\sum_\alpha p_\alpha = 1$.

$\mathcal{A}_\alpha$: Algorithmus $\mathcal{A}$, der mit dem Ergebnis α des anfänglichen Experiments läuft.
$\mathcal{A}_\alpha$ ist ein gewöhnlicher deterministischer Algorithmus!

$T_\alpha(\sigma) := \#(\text{Vergleiche bei } \mathcal{A}_\alpha \text{ auf Input } \sigma)$. ($\sigma = (a_1, \ldots, a_n)$: Permutation von $\{1, \ldots, n\}$.)
Nach Satz 6.6.2 gilt: $\frac{1}{n!} \sum_\sigma T_\alpha(\sigma) \geq \log(n!)$.

Summiere über alle α's: $\displaystyle\sum_\alpha p_\alpha \cdot \frac{1}{n!} \sum_\sigma T_\alpha(\sigma) \geq \log(n!)$.

Umstellen der Summation: $\displaystyle\frac{1}{n!} \sum_\sigma \Big(\sum_\alpha p_\alpha \cdot T_\alpha(\sigma) \Big) \geq \log(n!)$.

Daraus folgt: **Es gibt ein** σ mit $\sum_\alpha p_\alpha \cdot T_\alpha(\sigma) \geq \log(n!)$.
$\sum_\alpha p_\alpha \cdot T_\alpha(\sigma)$ ist aber gerade die erwartete Anzahl von Vergleichen von $\mathcal{A}$ auf Eingabe σ. $\square$

Vorlesungsvideo:

Sortieren in Linearzeit

6.7. Sortieren in Linearzeit

Wenn wir Schlüssel nicht als atomares Objekt, sondern z. B. als **Index** benutzen, dann gilt die untere Schranke aus 6.6 nicht.

Wir wollen (für besondere Schlüsselformate, ganze Zahlen oder Strings) in Linearzeit sortieren!

Sei zunächst $U = \{0, 1, \ldots, m-1\} = [m]$.

Gegeben: n Objekte mit Schlüsseln $a_1, \ldots, a_n$ aus U. Sortiere!

Präziser: Gegeben sind in $\texttt{A}[1..n]$ n Datenobjekte

$$(a_1, d_1), \ldots, (a_n, d_n)$$

mit Schlüsseln $a_1, \ldots, a_n$ aus U. Sortiere **(stabil)**!

(A) Countingsort – Sortieren durch Zählen

Idee, drei Phasen:

1. **Zähle** in Komponente C[i] eines Arrays C[$0..m-1$], wie oft der Schlüssel i vorkommt.

2. Benutze das Ergebnis von 1., um (wieder in C[i]) zu berechnen, wie viele Einträge in A einen Schlüssel $\leq i$ haben.

3. Übertrage die Einträge mit Schlüssel i aus A in die Positionen C[$i-1$]$+1, \ldots,$C[i] in Array B, **stabil**.

Beispiel:

A:

1	2	3	4	5	6	7	8	9
5 $_1$	**0** $_1$	**6** $_1$	**3** $_1$	**5** $_2$	**6** $_2$	**3** $_2$	**2** $_1$	**3** $_3$

C:

0	1	2	3	4	5	6

B:

1	2	3	4	5	6	7	8	9

1. **Zähle** in Komponente $\texttt{C}[i]$ eines Arrays $\texttt{C}[0..m-1]$, wie oft der Schlüssel i vorkommt.

(1) **for** i **from** 0 **to** $m-1$ **do** C[i] $\leftarrow 0$; // Zeitaufwand: $\Theta(m)$
(2) **for** j **from** 1 **to** n **do** C[A[j].key]++; // Zeitaufwand: $\Theta(n)$

2. Benutze das Ergebnis von 1., um (wieder in $\texttt{C}[i]$) zu berechnen, wie viele Einträge in A einen Schlüssel $\leq i$ haben.

(3) **for** i **from** 1 **to** $m-1$ **do** C[i] $\leftarrow$ C[i$-$1] $+$ C[i]; // Zeitaufwand: $\Theta(m)$

3. Übertrage die Einträge mit Schlüssel i aus A in die Positionen C[$i-1$]+1, ..., C[i] in Array B, **stabil**.

Geschickt: In A[1..n] einmal von hinten nach vorne laufen.

Zum Verständnis beobachte: Wenn Eintrag mit Schlüssel i, der in A am weitesten rechts steht, an Stelle A[j] steht, muss dieser an die Stelle B[C[i]] geschrieben werden. Der **nächste** Eintrag mit diesem Schlüssel muss in das Fach unmittelbar links daneben – daher: Eintrag übertragen, dann C[i] herunterzählen. Dies wird iteriert. Man erhält folgendes Vorgehen.

```
(4)  for j from n downto 1 do
(5)      i ← A[j].key;
(6)      B[C[i]] ← A[j];        // aktueller Platz für Schlüssel i (in i)
(7)      C[i]--;                // auf linken Nachbarn umsetzen
                                                  // Zeitaufwand: Θ(n)
```

	1	2	3	4	5	6	7	8	9
A:	$\mathbf{5}_1$	$\mathbf{0}_1$	$\mathbf{6}_1$	$\mathbf{3}_1$	$\mathbf{5}_2$	$\mathbf{6}_2$	$\mathbf{3}_2$	$\mathbf{2}_1$	$\mathbf{3}_3$

	0	1	2	3	4	5	6
C:							

	1	2	3	4	5	6	7	8	9
B:									

Eingabe, Datenstruktur. Kleine Ziffern in der Eingabe: Reihenfolge identischer Schlüssel.

	1	2	3	4	5	6	7	8	9
A:	$\mathbf{5}_1$	$\mathbf{0}_1$	$\mathbf{6}_1$	$\mathbf{3}_1$	$\mathbf{5}_2$	$\mathbf{6}_2$	$\mathbf{3}_2$	$\mathbf{2}_1$	$\mathbf{3}_3$

	0	1	2	3	4	5	6
C:	*1*	*0*	*1*	*3*	*0*	*2*	*2*

	1	2	3	4	5	6	7	8	9
B:									

Nach Zeilen (1)–(2): Einträge gezählt.

A:

1	2	3	4	5	6	7	8	9
$\mathbf{5}_1$	$\mathbf{0}_1$	$\mathbf{6}_1$	$\mathbf{3}_1$	$\mathbf{5}_2$	$\mathbf{6}_2$	$\mathbf{3}_2$	$\mathbf{2}_1$	$\mathbf{3}_3$

C:

0	1	2	3	4	5	6
1	*1*	*2*	*5*	*5*	*7*	*9*

B:

1	2	3	4	5	6	7	8	9

Nach Zeile (3): Endpunkte der Segmente gleicher Schlüssel ermittelt.

	1	2	3	4	5	6	7	8	9
A:	**5** $_1$	**0** $_1$	**6** $_1$	**3** $_1$	**5** $_2$	**6** $_2$	**3** $_2$	**2** $_1$	**3** $_3$

	0	1	2	3	4	5	6
C:	*1*	*1*	*2*	*5*	*5*	*7*	*9*

	1	2	3	4	5	6	7	8	9
B:					**3** $_3$				

Zeile (6): Eintrag kopieren.

	1	2	3	4	5	6	7	8	9
A:	$\mathbf{5}_1$	$\mathbf{0}_1$	$\mathbf{6}_1$	$\mathbf{3}_1$	$\mathbf{5}_2$	$\mathbf{6}_2$	$\mathbf{3}_2$	$\mathbf{2}_1$	$\mathbf{3}_3$

	0	1	2	3	4	5	6
C:	*1*	*1*	*2*	*4*	*5*	*7*	*9*

	1	2	3	4	5	6	7	8	9
B:					$\mathbf{3}_3$				

Zeile (7): Index heruntersetzen.

	1	2	3	4	5	6	7	8	9
A:	$\mathbf{5}_1$	$\mathbf{0}_1$	$\mathbf{6}_1$	$\mathbf{3}_1$	$\mathbf{5}_2$	$\mathbf{6}_2$	$\mathbf{3}_2$	$\mathbf{2}_1$	$\mathbf{3}_3$

	0	1	2	3	4	5	6
C:	*1*	*1*	*2*	*4*	*5*	*7*	*9*

	1	2	3	4	5	6	7	8	9
B:					$\mathbf{3}_3$				

A:

1	2	3	4	5	6	7	8	9
$\mathbf{5}_1$	$\mathbf{0}_1$	$\mathbf{6}_1$	$\mathbf{3}_1$	$\mathbf{5}_2$	$\mathbf{6}_2$	$\mathbf{3}_2$	$\mathbf{2}_1$	$\mathbf{3}_3$

C:

0	1	2	3	4	5	6
1	*1*	*2*	*4*	*5*	*7*	*9*

B:

1	2	3	4	5	6	7	8	9
	$\mathbf{2}_1$			$\mathbf{3}_3$				

Zeile (6): Eintrag kopieren.

A:

1	2	3	4	5	6	7	8	9
$\mathbf{5}_1$	$\mathbf{0}_1$	$\mathbf{6}_1$	$\mathbf{3}_1$	$\mathbf{5}_2$	$\mathbf{6}_2$	$\mathbf{3}_2$	$\mathbf{2}_1$	$\mathbf{3}_3$

C:

0	1	2	3	4	5	6
1	*1*	*1*	*4*	*5*	*7*	*9*

B:

1	2	3	4	5	6	7	8	9
	$\mathbf{2}_1$			$\mathbf{3}_3$				

Zeile (7): Index heruntersetzen.

	1	2	3	4	5	6	7	8	9
A:	$\mathbf{5}_1$	$\mathbf{0}_1$	$\mathbf{6}_1$	$\mathbf{3}_1$	$\mathbf{5}_2$	$\mathbf{6}_2$	$\mathbf{3}_2$	$\mathbf{2}_1$	$\mathbf{3}_3$

	0	1	2	3	4	5	6
C:	*1*	*1*	*1*	*4*	*5*	*7*	*9*

	1	2	3	4	5	6	7	8	9
B:		$\mathbf{2}_1$			$\mathbf{3}_3$				

	1	2	3	4	5	6	7	8	9
A:	$\mathbf{5}_1$	$\mathbf{0}_1$	$\mathbf{6}_1$	$\mathbf{3}_1$	$\mathbf{5}_2$	$\mathbf{6}_2$	$\mathbf{3}_2$	$\mathbf{2}_1$	$\mathbf{3}_3$

	0	1	2	3	4	5	6
C:	*1*	*1*	*1*	*4*	*5*	*7*	*9*

	1	2	3	4	5	6	7	8	9
B:		$\mathbf{2}_1$		$\mathbf{3}_2$	$\mathbf{3}_3$				

Zeile (6): Eintrag kopieren.

A:

1	2	3	4	5	6	7	8	9
$\mathbf{5}_1$	$\mathbf{0}_1$	$\mathbf{6}_1$	$\mathbf{3}_1$	$\mathbf{5}_2$	$\mathbf{6}_2$	$\mathbf{3}_2$	$\mathbf{2}_1$	$\mathbf{3}_3$

C:

0	1	2	3	4	5	6
1	*1*	*1*	*3*	*5*	*7*	*9*

B:

1	2	3	4	5	6	7	8	9
	$\mathbf{2}_1$		$\mathbf{3}_2$	$\mathbf{3}_3$				

Zeile (7): Index heruntersetzen.

A:

1	2	3	4	5	6	7	8	9
5 $_1$	**0** $_1$	**6** $_1$	**3** $_1$	**5** $_2$	**6** $_2$	**3** $_2$	**2** $_1$	**3** $_3$

C:

0	1	2	3	4	5	6
1	*1*	*1*	*3*	*5*	*7*	*9*

B:

1	2	3	4	5	6	7	8	9
	2 $_1$		**3** $_2$	**3** $_3$				

	1	2	3	4	5	6	7	8	9
A:	$\mathbf{5}_1$	$\mathbf{0}_1$	$\mathbf{6}_1$	$\mathbf{3}_1$	$\mathbf{5}_2$	$\mathbf{6}_2$	$\mathbf{3}_2$	$\mathbf{2}_1$	$\mathbf{3}_3$

	0	1	2	3	4	5	6
C:	*0*	*1*	*1*	*2*	*5*	*5*	*7*

	1	2	3	4	5	6	7	8	9
B:	$\mathbf{0}_1$	$\mathbf{2}_1$	$\mathbf{3}_1$	$\mathbf{3}_2$	$\mathbf{3}_3$	$\mathbf{5}_1$	$\mathbf{5}_2$	$\mathbf{6}_1$	$\mathbf{6}_2$

Resultat nach Beendigung der Schleife (6)–(9).

Satz 6.7.1 (Countingsort – Sortieren durch Zählen)

n Objekte mit Schlüsseln aus $[m]$ (gegeben in Array) können in Zeit $\Theta(n+m)$ und mit Zusatzplatz $\Theta(n+m)$ sortiert werden.

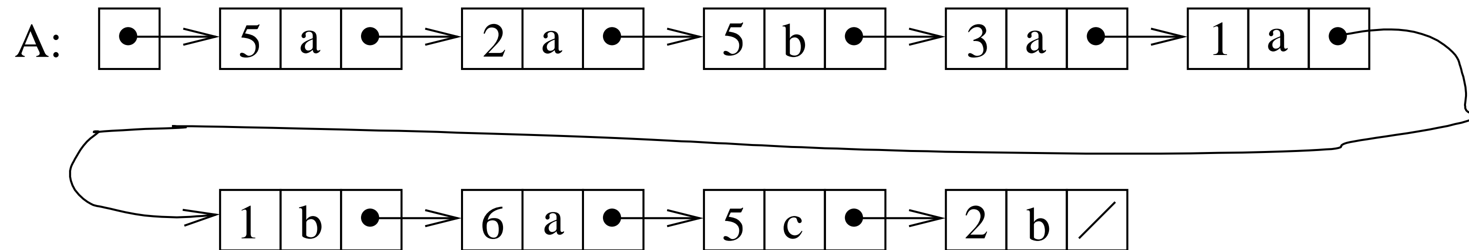
Das Sortierverfahren ist stabil.

Linearzeit und linearer Platz, falls $m \le cn$, c konstant.

Countingsort ist auch in der **praktischen** Anwendung **sehr schnell** (wenn auch überhaupt nicht Cache-freundlich).

(B) Bucketsort – Fachsortieren

Nun seien die zu sortierenden Objekte mit Schlüsseln aus $[m]$ als (einfach verkettete) lineare **Liste** gegeben.

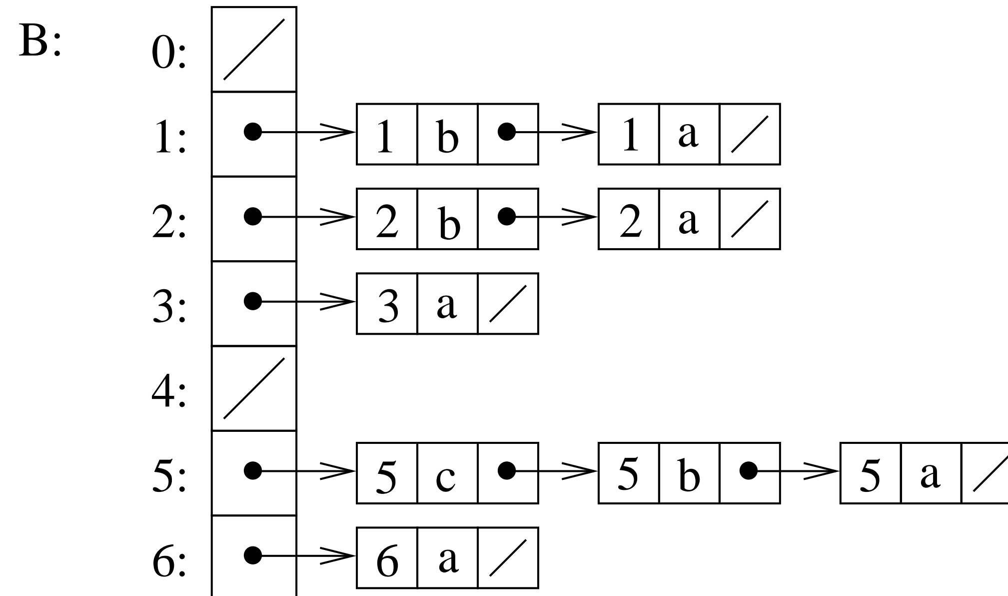


Datenstruktur: Array $\texttt{B}[0..m-1]$ von Listen $L_0 \ldots, L_{m-1}$.

Initialisierung mit NULL-Zeigern: Zeit $\Theta(m)$.

Durchlaufe Eingabeliste, übertrage Einträge mit Schlüssel i an den Anfang der Liste L_i (angehängt in B[i]): Zeit $\Theta(n)$.

Nach Transport in das Listenarray:

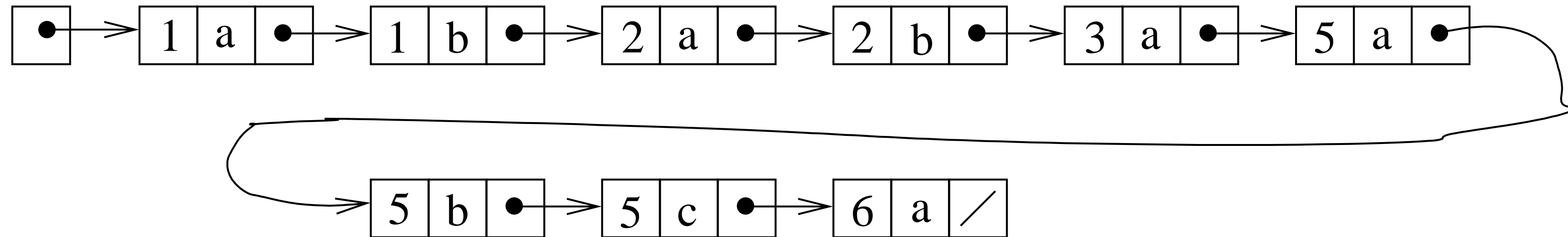


Bemerke: Identische Schlüssel in umgekehrter Reihenfolge.

Durchlaufe Array `B`$[0..m-1]$ **von hinten nach vorn**, baue neue sortierte Liste `A` (anfangs leer).

Für i: lies Elemente der Liste L_i nacheinander aus, hänge sie vorne in die Liste `A` ein.

Nach Auslesen:



Algorithmus Bucketsort (Fachsortieren)

(1) Bearbeite Elemente der Liste `A` nacheinander.

Füge Listenelement | i | data | • |→ ····· **vorn** in Liste B[i] ein.

NB: Reihenfolge von Elementen mit demselben Schlüssel wird umgedreht.

(2) Für $i = m-1, \ldots, 1, 0$: lies B[i] von vorn nach hinten.
Füge Element | i | data | • |→ ····· **vorn** in Ausgabeliste ein.

NB: Reihenfolge wird erneut umgedreht $\rightarrow$ **Stabilität**.

Korrektheit: klar.

Laufzeit:

(0) Anlegen von `B`, Initialisieren mit NULL-Zeigern: $\Theta(m)$.

(1) Verteilen auf die Listen: $\Theta(n)$

(2) Zusammenfügen: $\Theta(m) + \Theta(n)$
(Jeder Listenanfang $\texttt{B}[i]$ muss inspiziert werden.)

Gesamt: $\Theta(m+n)$.

Linearzeit und linearer Platz, falls $m \le cn$, c konstant.

Alternative:

Bei `B`$[i]$-Listen Endezeiger benutzen, in Phase (1) **hinten** anfügen.

In Phase (2): Teillisten konkatenieren.

Vorteil: in Phase (2) nur $O(m)$ Zeigerbewegungen, Zeit $O(m)$.

Nachteil: Zusätzlicher Speicherplatz $O(m)$ für Endezeiger.

Günstig insbesondere dann, wenn $m \ll n$ ist.

Satz 6.7.2 (Bucketsort/Fachsortieren)

n Objekte mit Schlüsseln aus $[m]$ (gegeben als lineare Liste) können in Zeit $\Theta(n+m)$ und mit Zusatzplatz für m Zeiger **stabil** sortiert werden.

Was tun, wenn die Anzahl m der möglichen Schlüssel begrenzt, aber deutlich größer als n ist? (Beispiel: $m = n^2$.)

(C) Radixsort – Mehrphasen-Counting-/Bucketsort

Anwendbar in verschiedenen Situationen:

1. Schlüssel sind **Folgen** $x = (x_1, \ldots, x_k) \in U^k$, $U = [m]$, $m \le n$.

   Anordnungskriterium: **lexikographisch**, d. h.

$$(x_1, \ldots, x_k) < (y_1, \ldots, y_k)$$
$$\Leftrightarrow \exists j \in \{1, \ldots, k\} : (x_1, \ldots, x_{j-1}) = (y_1, \ldots, y_{j-1}) \wedge x_j < y_j.$$

2. Schlüssel sind **Zahlen** x in $U \subseteq \{0, 1, \ldots, m^k - 1\}$, $m \le n$.

3. k (möglicherweise verschiedene) geordnete Mengen $(U_1, <_1), \ldots, (U_k, <_k)$

   Schlüssel sind **Folgen** $x = (x_1, \ldots, x_k) \in U_1 \times \cdots \times U_k$,
   mit lexikographischer Ordnung.

   Beispiele:
   (Kalender-)Daten: $\{1, \ldots, 31\} \times \{1, \ldots, 12\} \times \{1801, \ldots, 2090\}$.

   Besser: $\{1801, \ldots, 2090\} \times \{1, \ldots, 12\} \times \{1, \ldots, 31\}$.

   Spielkarten: $\{\diamondsuit, \heartsuit, \spadesuit, \clubsuit\} \times \{2, \ldots, 10, \text{Bube}, \text{Dame}, \text{König}, \text{Ass}\}$.

4. $\Sigma^{<\infty}$. Dabei: Σ ist „Alphabet", d. h. nichtleere endliche Menge (z. B. ein ISO/IEC 8859-Alphabet) mit Ordnung $<$.

(4a) **Lexikographische Ordnung** auf $\Sigma^{<\infty}$
(wie im Lexikon, ab $<$ abacus $<$ abend $<$ aber $<$ ar $<$ ararat):

$$(x_1, \ldots, x_k) <_{\text{lex}} (y_1, \ldots, y_\ell)$$

$$\Leftrightarrow \quad \big(k < \ell \wedge (x_1, \ldots, x_k) = (y_1, \ldots, y_k)\big) \quad \vee$$
$$\big(\exists j \in \{1, \ldots, \min\{k, \ell\}\} : (x_1, \ldots, x_{j-1}) = (y_1, \ldots, y_{j-1}) \wedge x_j < y_j\big).$$

(4b) **Kanonische Ordnung** auf $\Sigma^{<\infty}$
(kürzere Strings zuerst, ab $<$ ar $<$ aber $<$ abend $<$ abacus $<$ ararat):

$$(x_1, \ldots, x_k) <_{\text{kan}} (y_1, \ldots, y_\ell)$$

$$\Leftrightarrow \quad k < \ell \quad \vee$$
$$\big(k = \ell \wedge \exists j \in \{1, \ldots, k\} : (x_1, \ldots, x_{j-1}) = (y_1, \ldots, y_{j-1}) \wedge x_j < y_j\big).$$

Zunächst: Situation 1, Schlüssel $x = (x_1, \ldots, x_k) \in U^k$, $U = [m]$.

Idee: **sortiere k-mal mittels Counting-/Bucketsort**
und zwar gemäß Komponente k, dann $k-1$, . . . , dann 1
(die „**am wenigsten signifikante**“ Komponente **zuerst**).

Eingabe:
Array/Liste `A`: Einträge mit Schlüsseln $(x_1, \ldots, x_k)$ aus $[m]^k$.

Methode:
für $l = k, k-1, \ldots, 1$ tue:
  sortiere `A` mittels **Counting**- bzw. **Bucketsort** für $U = [m]$
  mit Komponente x_l aus $(x_1, \ldots, x_k)$ als Sortierschlüssel.

Zeit insgesamt: $\Theta(k \cdot (n + m))$

Zusatzplatz: $\Theta(m + n)$ bzw. $\Theta(m)$.

Beispiel:

$U = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}\}$

Eingabe `A`: BCC ABC BAC CAD ABA

Nach Runde $l = 3$: ABA BCC ABC BAC CAD

Nach Runde $l = 2$: BAC CAD ABA ABC BCC

Nach Runde $l = 1$: ABA ABC BAC BCC CAD

Beobachte: Nach Runde für l sind die Objekte gemäß $x_l, \ldots, x_k$ sortiert.

Satz 6.7.3

Der skizzierte Algorithmus **Radixsort** sortiert n Objekte mit Schlüsseln aus $[m]^k$ in Zeit $\Theta(k(n+m))$ und Platz $\Theta(m)$ (bzw. $\Theta(n+m)$ für Arrays). Das Verfahren ist stabil.

Beweis: **(Nicht prüfungsrelevant.)** Nur Korrektheit. Man zeigt durch Induktion über $l = k, k-1, \ldots, 1$ die folgende Schleifeninvariante:

Nach Schleifendurchlauf für l ist das Array/die Liste `A`
gemäß den Teilschlüsseln $(x_l, x_{l+1}, \ldots, x_k) \in [m]^{k-l+1}$ lexikographisch sortiert.

Für Induktionsschritt $l+1 \to l$ benutzt man, dass Counting-/Bucketsort **stabil** ist, und die schon hergestellte Ordnung bezüglich der weniger signifikanten Komponenten nicht zerstört.

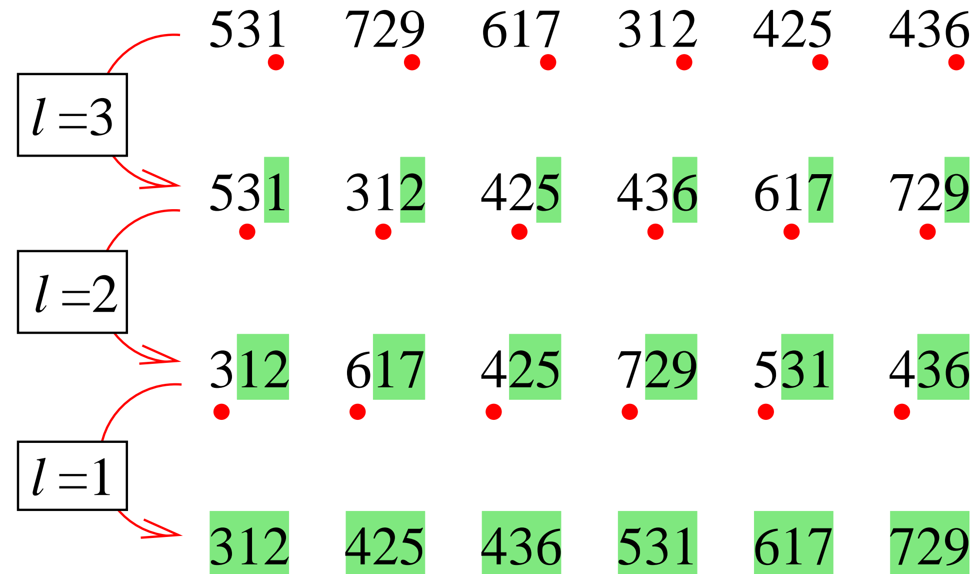
Das Verfahren für $U_1 \times \cdots \times U_k$ (Situation 3) ist im Wesentlichen dasselbe, nur benötigt man verschiedene B-Arrays (oder Arrayabschnitte) in den einzelnen Durchgängen.

Zeit: $O(kn + |U_1| + \cdots + |U_k|)$; Platz: $O(n + \max_{1 \le i \le k} |U_i|)$.

Situation 2: Schlüssel sind Zahlen in $\{0, 1, \ldots, m^k - 1\}$:

Repräsentiere Zahlen in m-ärer Darstellung (mit k Ziffern); wende Radixsort an.

Beispiel: $m = 10$ (Dezimaldarstellung)



So kleine m sind **nicht typisch**.
Sinnvoll: $m \approx n$, m Zweierpotenz, z. B. $m = 2^8$ oder 2^{16}.

Dann ist eine „Ziffer“ x_l ein Segment der Binärdarstellung von x, die leicht aus x zu extrahieren ist.

Satz 6.7.4 **Radixsort** für Zahlen sortiert n Objekte mit Schlüsseln aus $[m^k]$ in Zeit $\Theta(k(n+m))$ und Platz $\Theta(m)$ (bzw. $\Theta(n+m)$ für Arrays). Das Verfahren ist stabil.