

4 Sortierverfahren

4.1 Problemstellung

Vorbedingung: Sei M eine Menge von *Schlüsseln* und \leq eine totale Ordnung auf M .

Gegeben: Folge s_1, \dots, s_n von Datensätzen

Datensatz s_i hat (Sortier-) Schlüssel $k_i \in M$

Gesucht: Permutation π auf den Indizes $\{1, \dots, n\}$, so dass

$$k_{\pi(1)} \leq k_{\pi(2)} \leq \dots \leq k_{\pi(n)}.$$

Varianten:

- *intern/extern*: Ein- und Ausgabe im Speicher (Liste, Array, ...) vs. sequentielles Medium (Festplatte, Magtape, ...)

- *in-situ Verfahren* benötigt wenig extra Speicherplatz (z.B. $o(n)$)
- *stabiles Verfahren* erhält Reihenfolge von Datensätzen mit gleichem Schlüssel

4.2 Rahmen fürs Sortieren

```
interface Sortable {
    boolean isLe (int i, int j);           //  $k_i \leq k_j$ ?
    boolean isLt (int i, int j);           //  $k_i < k_j$ ?
    boolean isEq (int i, int j);           //  $k_i = k_j$ ?
    void swap (int i, int j);              // Vertausche  $s_i$  und  $s_j$ 
}
```

Dabei gilt:

- eigentliche Datensätze versteckt in Implementierung von Sortable
- isLe ist totale Ordnung
- isEq ist Äquivalenzrelation
- $\text{isLt}(i,j) \Leftrightarrow \text{isLe}(j,i)$
- $\text{isLe}(i,j)$ und $\text{isLe}(j,i) \Leftrightarrow \text{isEq}(i,j)$

⇒ einzige Operation auf Datensätzen: *Schlüsselvergleich*

- Leistungskriterien: # Schlüsselvergleiche und # Datenbewegungen

Beispiel: Sortable IntArray

```
public class IntArray implements Sortable {
    private int [] a;
    IntArray (int [] a) { this.a = a; }
    public boolean isLe (int i, int j) { return a[i] <= a[j]; }
    public boolean isLt (int i, int j) { return a[i] < a[j]; }
    public boolean isEq (int i, int j) { return a[i] == a[j]; }
    public void swap (int i, int j) {
        int t = a[i]; a[i] = a[j]; a[j] = t;
    }
    // ... andere Methoden, zB zum Auslesen des Arrays
}
```

4.3 Elementare Sortierverfahren

4.3.1 Bubblesort

```
void bubbleSort (Sortable a, int lo, int n) {
    for (boolean sorted = false; !sorted; n--) {
        sorted = true;
        for (int i = lo; i < n; i++)
            if (a.isLt (i+1,i)) {
                a.swap (i,i+1);
                sorted = false;
            }
    }
}
```

- Sortiert $a[lo..n]$
- Mit $n = n - lo + 1$, $C(n)$ Anzahl Vergleiche, $M(n)$ Anzahl Bewegungen:

$$\begin{aligned} C_{min}(n) &= n - 1 & C_{max}(n) &= n(n - 1)/2 & \overline{C}(n) &\in O(n^2) \\ M_{min}(n) &= 0 & M_{max}(n) &= n(n - 1)/2 & \overline{M}(n) &\in O(n^2) \end{aligned}$$

4.3.2 Auswahlort^a

Idee: Für $1 \leq i \leq n$ bestimme jeweils das kleinste Element in $a[i..n]$ und tausche es an Position i

Implementierung

```
void selectionSort (Sortable a, int lo, int n) {
    int minindex;           // Index des Kandidaten für Minimum
    for (int i = lo; i<n; i++) {
        minindex = lo;
        for (int j = i+1; j<=n; j++)
            if (a.isLt (j,minindex))
                minindex = j;
        a.swap (lo, minindex);
    }
}
```

^a*selection sort*

4.3.3 Sortieren durch Einfügen^a

Idee: Für $n - 1 \geq i \geq 1$ füge jeweils Element $a[i]$ an der richtigen Stelle in sortierte Folge $a[i+1..n]$ ein.

Implementierung

```
void insertionSort (Sortable a, int lo, int n) {
    for (int i = n-1; i>=lo; i--) {
        for (int j = i; j< n; j++)
            if (a.isGt (j,j+1))
                a.swap (j,j+1);
            else
                break;
    }
}
```

^a*insertion sort*

Analyse von Insertion Sort

Für $(i = n - 1, \dots, 1)$ benötigt das Einfügen des Elementes i mindestens 1, höchstens $n - i$ Vergleiche und mindestens 0, höchstens $n - i$ Bewegungen

$$C_{min}(n) = n - 1 \quad C_{max}(n) = \sum_{i=1}^{n-1} (n - i) = \sum_{i=1}^{n-1} i = \frac{n(n - 1)}{2}$$
$$M_{min}(n) = 0 \quad M_{max}(n) = \sum_{i=1}^{n-1} (n - i) = \frac{n(n - 1)}{2}$$

Im Mittel: Die Hälfte von $a[i+1..n]$ ist größer als $a[i]$

$$\bar{C}(n) \approx \sum_{i=1}^{n-1} i/2 \in \Theta(n^2) \quad \bar{M}(n) \approx \sum_{i=1}^{n-1} i/2 \in \Theta(n^2)$$

4.3.4 Sortieren durch Verschmelzen^a

Prinzip:

- Aufteilen der Eingabefolge;
- Sortieren der Teilfolgen (rekursiv);
- Verschmelzen der sortierten Folgen

```
void mergeSort (Mergeable a, int lo, int hi) {
    if (lo == hi) return;           // Länge ist 1
    int mid = (hi + lo) / 2;
    mergeSort (a, lo, mid);         // linke Hälfte sortieren
    mergeSort (a, mid+1, hi);      // rechte Hälfte sortieren
    merge (a, lo, mid, hi);        // verschmelzen
}
```

^amerge sort

Infrastruktur für mergeSort

- extra Platz fürs Verschmelzen erforderlich (nicht in-situ)

⇒ Hilfsstruktur (Array) von gleicher Größe wie Eingabe

```
interface Mergeable extends Sortable {  
    void move (int i, int j);  
    // move from source array[i] to temporary array[j]  
    void moveBack (int i, int j);  
    // move from temporary array[i] to source array[j]  
}
```

Beispiel: Implementierung in IntArray

```
private int [] dest;  
public void move (int i, int j) { dest[j] = a[i]; }  
public void moveBack (int i, int j) { a[j] = dest[i]; }
```

Verschmelzen

von sortiertem a[lo..mid] mit a[mid+1..hi]

```
void merge (Mergeable a, int lo, int mid, int hi) {
    int i = lo;      // durchläuft a[lo..mid]      (sortiert)
    int j = mid+1;  // durchläuft a[mid+1..hi]    (sortiert)
    int out = lo;   // Ausgabezeiger: [lo..hi]
    while (i <= mid && j <= hi)
        if (a.isLe (i,j))
            a.move (i++, out++);
        else
            a.move (j++, out++);
    // Genau eine der folgenden Schleifen wird durchlaufen
    while (i <= mid) a.move (i++, out++);
    while (j <= hi) a.move (j++, out++);
    // Zurückkopieren
    for (out = lo; out<= hi; out++)
        a.moveBack (out, out);
}
```

Analyse von Mergesort

Schlüsselvergleiche für merge Setze

- $n_1 = \text{mid} - \text{lo} + 1$ und $n_2 = \text{hi} - \text{mid}$
- best-case: $C_{\min}(n_1, n_2) = \min(n_1, n_2)$
- worst-case: $C_{\max}(n_1, n_2) = n_1 + n_2 - 1$
- Mit $n = n_1 + n_2$ ergibt sich $C_{\min}(n) = n/2$ und $C_{\max}(n) = n - 1$

Wegen $m = \lceil (\text{lo} + \text{hi})/2 \rceil$ ist $n_1 \in \{n_2, n_2 + 1\}$

Schlüsselvergleiche für mergeSort folgen der Rekursionsgleichung

$$T(n) = 2T(n/2) + C_{\text{merge}}(n)$$

Da $C_{\text{merge}}(n) \in \Theta(n) = \Theta(n^{\log_2 2} \cdot \log^0 n)$, gilt

$$T(n) \in \Theta(n \log n)$$

Zusätzlicher Speicher: $O(n)$

Variante: Reines 2-Wege-Mergesort

for $i = 0 \dots \log_2 n$

verschmelze adjazente Teilfolgen der Länge 2^i

8	6	9	3	4	7	2	1
6	8	3	9	4	7	1	2
3	6	8	9	1	2	4	7
1	2	3	4	6	7	8	9

Variante: Natürliches 2-Wege-Mergesort unter Ausnutzung bereits sortierter Teilfolgen, *runs*

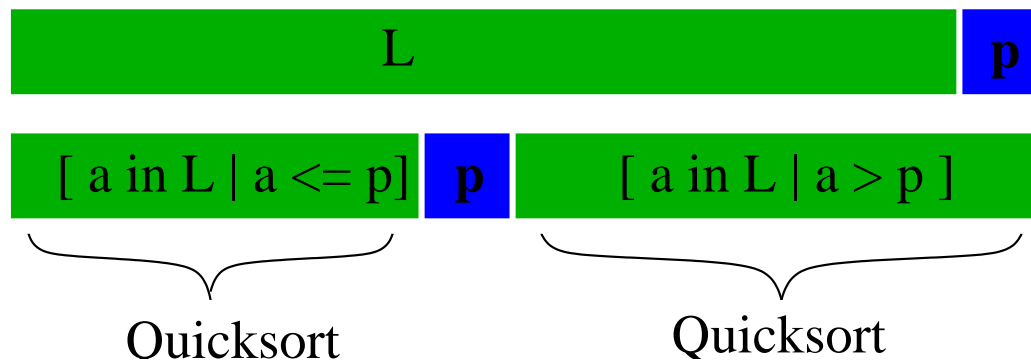
8	6	9	3	4	7	2	1
6	8	9	2	3	4	7	1
2	3	4	6	7	8	9	1
1	2	3	4	6	7	8	9

$$C_{\min}(n) = n - 1 \quad \overline{C}(n) = C_{\max}(n) = O(n \log n)$$

4.3.5 Quicksort

- Divide-and-Conquer Prinzip
- Sehr gute Laufzeit im Mittel, schlechte Laufzeit im worst case.

```
quicksort xs =  
  if length xs <= 1 then xs  
  else let (p, xs') = selectAndRemovePivot xs in  
        quicksort [ x <- xs' | x <= p ]  
  ++ [ p ]  
  ++ quicksort [ x <- xs' | x > p ]
```



Implementierung

```
void quicksort (Sortable a, int lo, int hi) {  
    if (lo < hi) {  
        int i = divide (a, lo, hi);  
        quicksort (a, lo, i-1);  
        quicksort (a, i+1, hi);  
    }  
}
```

divide (a, lo, hi) implementiert den Aufteilungsschritt

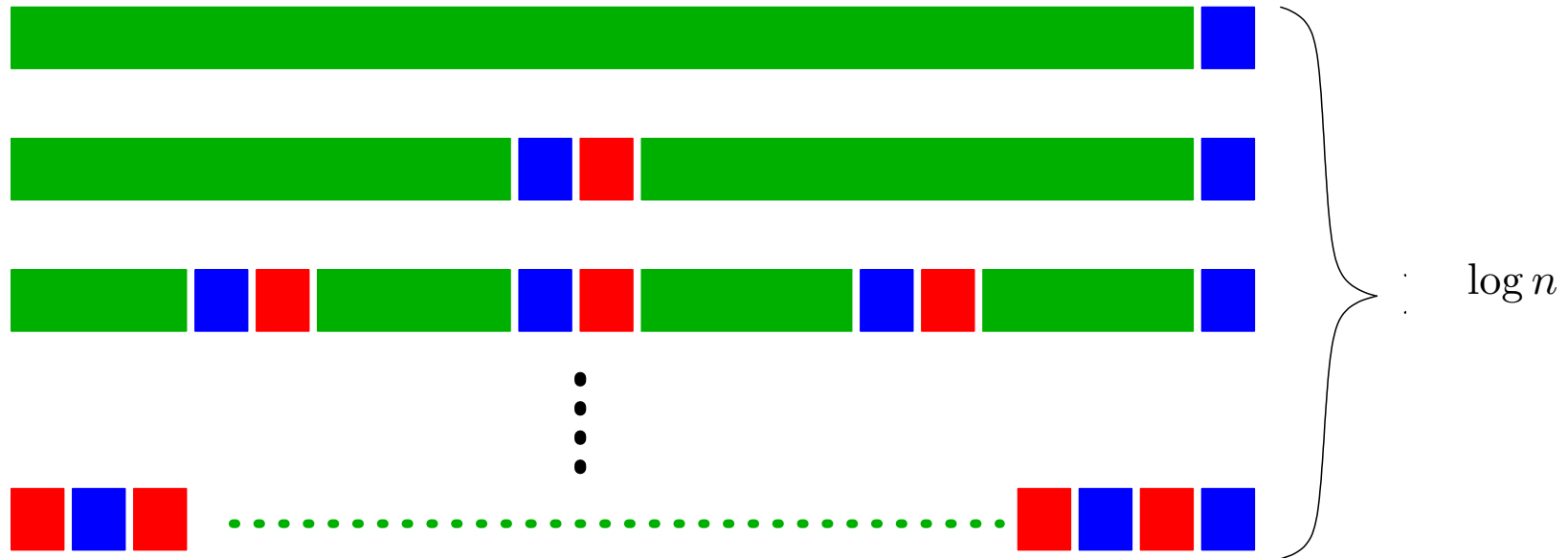
- wählt ein Pivotelement p aus a[lo..hi]
- partitioniert a entsprechend p
- liefert neue Position von p als Ergebnis

Implementierung: Aufteilungsschritt

```
int divide (Sortable a, int lo, int hi) {
    int p = hi;           // Index von Pivot
    int i = lo-1;        // a[lo..i] kleiner als Pivot
    int j = hi;          // a[j..hi-1] größer gleich Pivot
    while (i < j) {
        do i++; while (i < j && a.isLt (i, p));
        do j--; while (i < j && a.isGe (j, p));
        if (i < j)       // a[i] >= p > a[j]
            a.swap (i, j);
        else
            a.swap (i, p);
    }
    return i;
}
```

Analyse von Quicksort

Best case: Pivotelement teilt Folge jeweils genau zur Hälfte

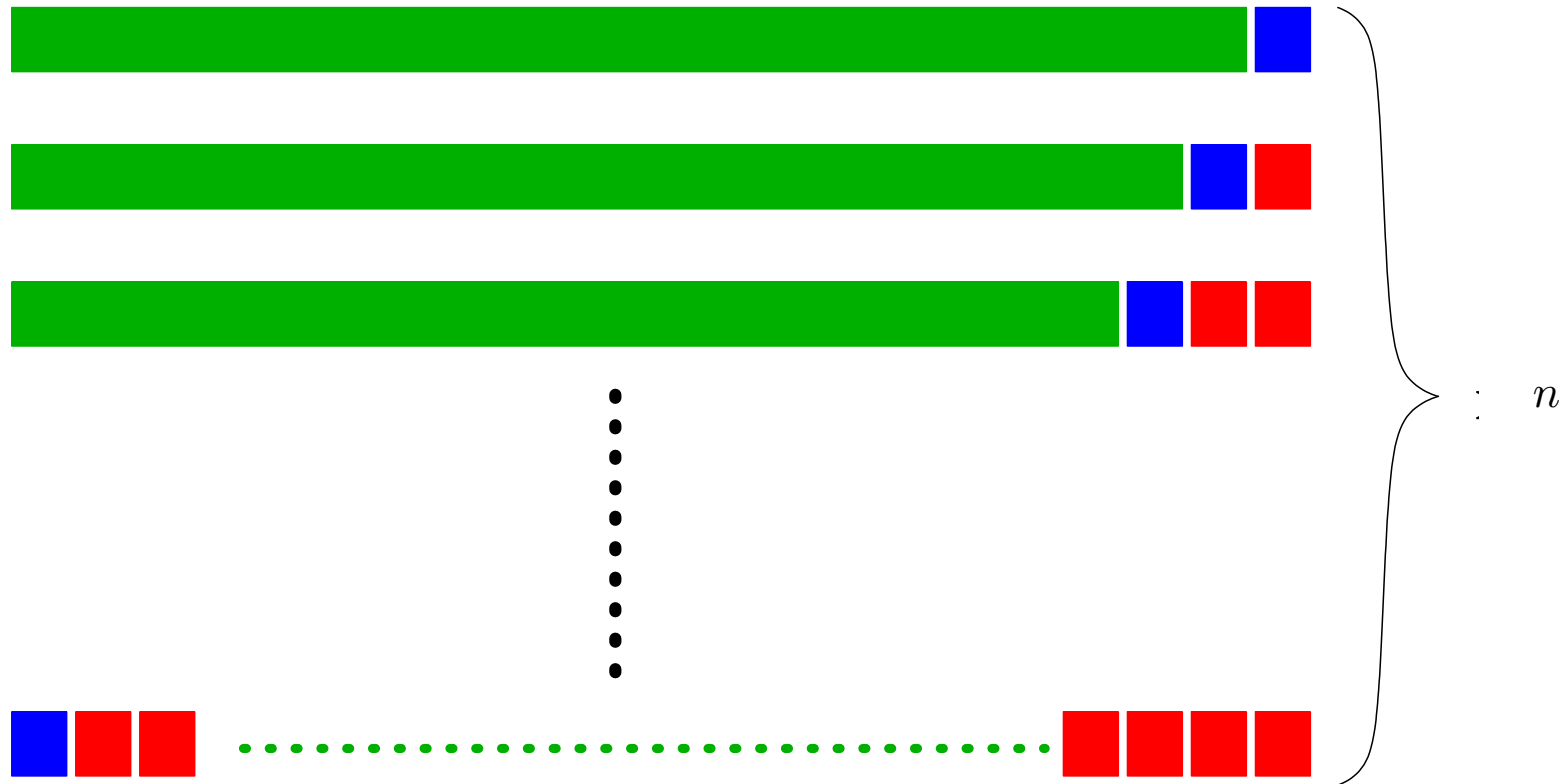


$$C_{min}(n) = 2C_{min}(n/2) + \Theta(n)$$

$$C_{min}(n) \in O(n \log n)$$

Analyse von Quicksort

Worst case: je eine der beiden Folgen ist leer; z.B. Folge aufsteigend sortiert:



$$C_{max}(n) = C_{max}(n-1) + \Theta(n)$$

$$C_{max}(n) \in O(n^2)$$

Average Case Analyse von Quicksort

Annahmen:

- n paarweise verschiedene Schlüssel
- Alle $n!$ Permutationen gleichwahrscheinlich

Fakt:

- Teilfolgen nach Aufteilungsschritt wieder zufällig

$$\Rightarrow P(\text{Pivotelement hat Rang } k) = 1/n.$$

Rekursionsgleichung:

$$\begin{aligned}\bar{C}(0) = \bar{C}(1) &= 0 \\ \bar{C}(n) &= n - 1 + \frac{1}{n} \sum_{k=1}^n (\bar{C}(k-1) + \bar{C}(n-k))\end{aligned}$$

Satz (Geschlossene Form)

$$\bar{C}(n) \approx 1.386n \log n - 2.846n + O(\log n)$$

Satz (Abschätzung) Für eine Konstante $c > 0$ und $n \geq 2$ gilt

$$\overline{C}(n) \leq c \cdot n \log n \quad (8)$$

Beweis durch vollständige Induktion (schreibe $C(n)$ statt $\overline{C}(n)$ und $\log n$ statt $\log_2 n$)

Induktionsverankerung $n = 2$ liefert $c \geq 1/2$.

Induktionsschritt: Es gelte die Ungleichung (8) für alle $i < n$.

$$\begin{aligned} C(n) &= \frac{1}{n} \left[\sum_{k=1}^n (C(k-1) + C(n-k)) \right] + n - 1 \\ &= \frac{2}{n} \left[\sum_{k=1}^{n-1} C(k) \right] + n - 1 \\ &\leq \frac{2c}{n} \left[\sum_{k=1}^{n-1} k \cdot \log k \right] + n - 1 \quad (\text{Ind.-Hypothese angewendet}) \\ &= \frac{2c}{n} \left[\sum_{k=1}^{n/2} k \cdot \log k + \sum_{k=n/2+1}^{n-1} k \cdot \log k \right] + n - 1 \end{aligned}$$

Für $1 \leq k \leq n/2$: $\log k \leq \log n - 1$

Für $n/2 + 1 \leq k \leq n - 1$: $\log k \leq \log n$

$$\begin{aligned}
C(n) &= \dots \\
&\leq \frac{2c}{n} \left[\sum_{k=1}^{n/2} k(\log n - 1) + \sum_{k=n/2+1}^{n-1} (k \cdot \log n) \right] + n - 1 \\
&= \frac{2c}{n} \left[\frac{n}{4} \left(\frac{n}{2} + 1 \right) \log n - \frac{n^2}{8} - \frac{n}{4} + \left(\frac{3n^2}{8} - \frac{3n}{4} \right) \log n \right] + n - 1 \\
&= \frac{2c}{n} \left[\left(\frac{n^2}{2} - \frac{n}{2} \right) \log n - \frac{n^2}{8} - \frac{n}{4} \right] + n - 1 \\
&= c \cdot n \log n - c \cdot \log n - \frac{c \cdot n}{4} - \frac{c}{2} + n - 1 \\
&\leq c \cdot n \log n - \frac{c \cdot n}{4} - \frac{c}{2} + n - 1 \\
&\leq c \cdot n \log n - n - 2 + n - 1 && \text{für } c \geq 4 \\
&\leq c \cdot n \log n.
\end{aligned}$$

4.4 Schnelle Sortierverfahren

4.4.1 Stand der Wissenschaft

Untere Schranke: $C_{max}(n) > \bar{C}(n) \geq \lceil \log(n!) \rceil - 1 \approx n \log n - 1.4427n$,

Ziel: $C_{max}(n)$ bzw. $\bar{C}(n) \leq b \cdot n \log n + cn$ für kleine c, b

Quicksort-Varianten:

<i>QUICKSORT</i>	(Hoare 1962)	$\bar{C}(n) \approx 1.386n \log n - 2.846n + O(\log n)$
<i>CLEVER-QUICKSORT</i>	(Hoare 1962)	$\bar{C}(n) \approx 1.188n \log n - 2.255n + O(\log n)$
<i>QUICK-HEAPSORT</i>	(Cantone/ Cincotti 2000)	$\bar{C}(n) = n \log n + 3n + o(n)$
<i>QUICK-WEAK-HEAPSORT</i>		$\bar{C}(n) = n \log n + 0.2n + o(n)$

Heapsort-Varianten:

<i>HEAPSORT</i>	(Williams/Floyd 1964)	$C_{max}(n) = 2n \log n + O(n)$
<i>BOTTOM-UP-HEAPSORT</i>	(Wegener 1993)	$C_{max}(n) = 1.5n \log n + O(n)$ allerdings $\bar{C}(n) = n \log n + O(n)$
<i>WEAK-HEAPSORT</i>	(Dutton 1993)	$C_{max}(n) = n \log n + 0.1n$
<i>RELAXED-WEAK-HEAPSORT</i>		$C_{max}(n) = n \log n - 0.9n$

4.4.2 Clever-Quicksort

Auswahl des Pivotelementes (Median-of-3 Strategie):

- a) $m = (\text{left} + \text{right})/2$ oder b) $m = \text{left} + 1$
- Pivot: mittleres Element von $\{a[\text{left}], a[m], a[\text{right}]\}$
- vertausche $a[\text{left}]$ mit Pivot, weiter wie bisher

Worst-case:

- existiert immer noch, aber
- wird durch a) vermieden für auf- bzw. absteigende Sortierung

Satz: Schlüsselvergleiche im Mittel

$$\overline{C}(n) \approx 1.188 n \log n - 2.255 n + O(\log n)$$

```

void cleverQuickSort(Sortable a, int left, int right) {
    if (right-left >= 3) {
        if(a.isLt (right, left+1)) a.swap(left+1, right);
        if(a.isLt (right, left))    a.swap(left, right);
        if(a.isLt (left, left+1))  a.swap(left+1, left);
        // pivotelement in a[left]
        int i = left+1, j = right;
        while (true) {
            do { i++; } while (a.isLt (i, left));
            do { j--; } while (a.isLt (left, j));
            if (i < j) a.swap (i,j);
            else break;
        }
        a.swap(left,j);
        cleverQuickSort (a, left, j-1);
        cleverQuickSort (a, i, right);
    } else
        sortLessThanThreeElements (a, left, right);
}

```

4.4.3 Heapsort: Effizientes Sortieren durch Auswählen

Prinzip von Heapsort:

- Sortieren durch wiederholtes Auswählen des Maximums (Auswahlort).
- Verwendet Struktur (Halde bzw. Heap), die Bestimmung des Maximums effizient unterstützt.

```
heapSort (folge) =  
  folge.verwandleInHeap ();  
  while (!folge.empty())  
    entferne maximales Element m aus folge  
    gebe m aus  
    verwandle folge wieder in einen Heap.
```

Definition: Heap

Ein Folge $F = (k_1, k_2, \dots, k_n)$ von Schlüsseln heißt (Max-)Heap, wenn für alle $i \in \{1, 2, \dots, n/2\}$ gilt:

$$k_i \geq k_{2i} \quad \text{und} \quad k_i \geq k_{2i+1} \quad (9)$$

(Für einen *Min-Heap* ersetze \geq durch \leq in (9))

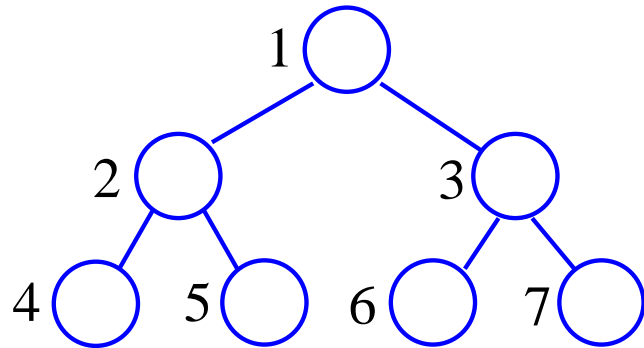
Beispiel

1	2	3	4	5	6	7
47	17	43	15	8	4	2

47	15	17	8	43	4	2
----	----	----	---	----	---	---

Veranschaulichung

1	2	3	4	5	6	7
47	17	43	15	8	4	2



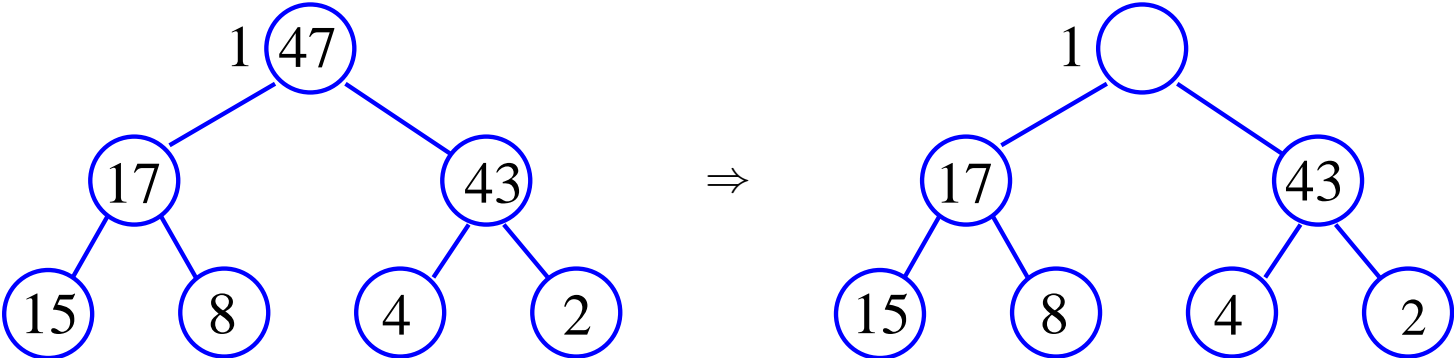
Heapbedingung

- $\text{key}(t) \geq \text{key}(\text{left}(t))$
 - $\text{key}(t) \geq \text{key}(\text{right}(t))$
- ⇒ Maximum an Wurzel (Position 1)

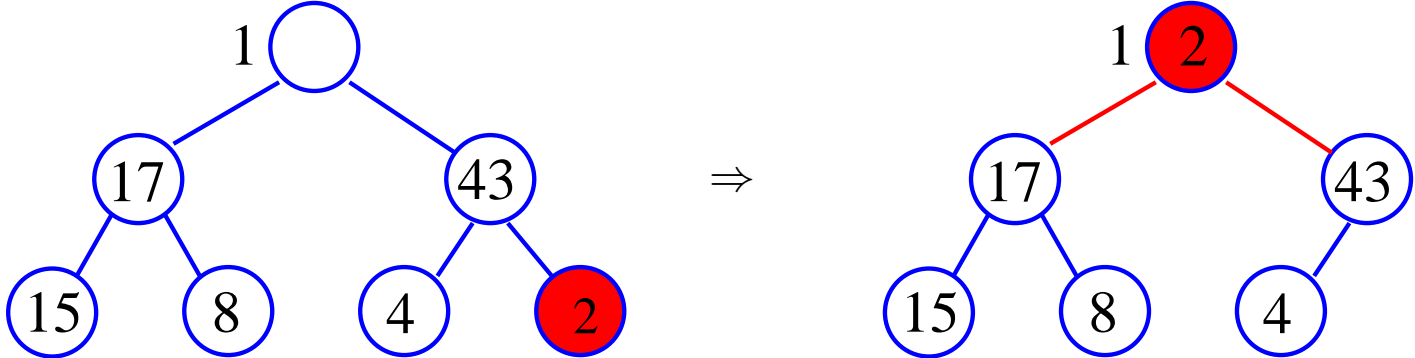
Vollständiger Binärbaum mit Positionsnummern

- Level i hat 2^i Knoten (außer dem letzten Level)
- Knoten von oben nach unten und von links nach rechts nummeriert
- Knoten i hat Knoten $2i$ als linken und Knoten $2i + 1$ als rechten Sohn

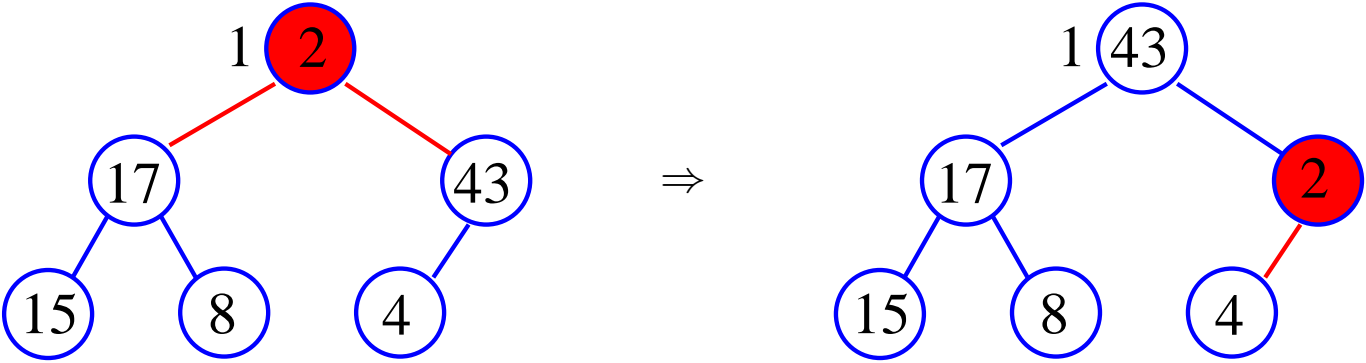
1. Entfernen des Maximums k_1



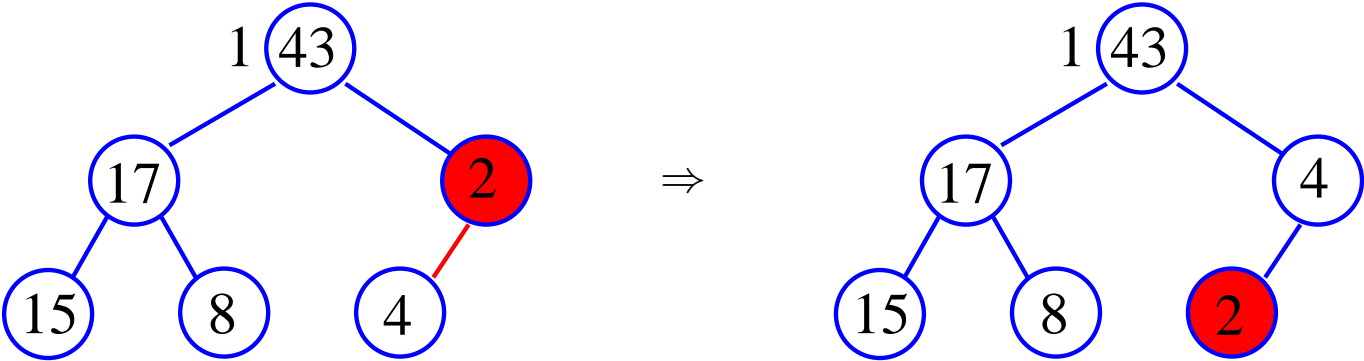
2. Übertrage k_n an Wurzel



3. Versickere k_1

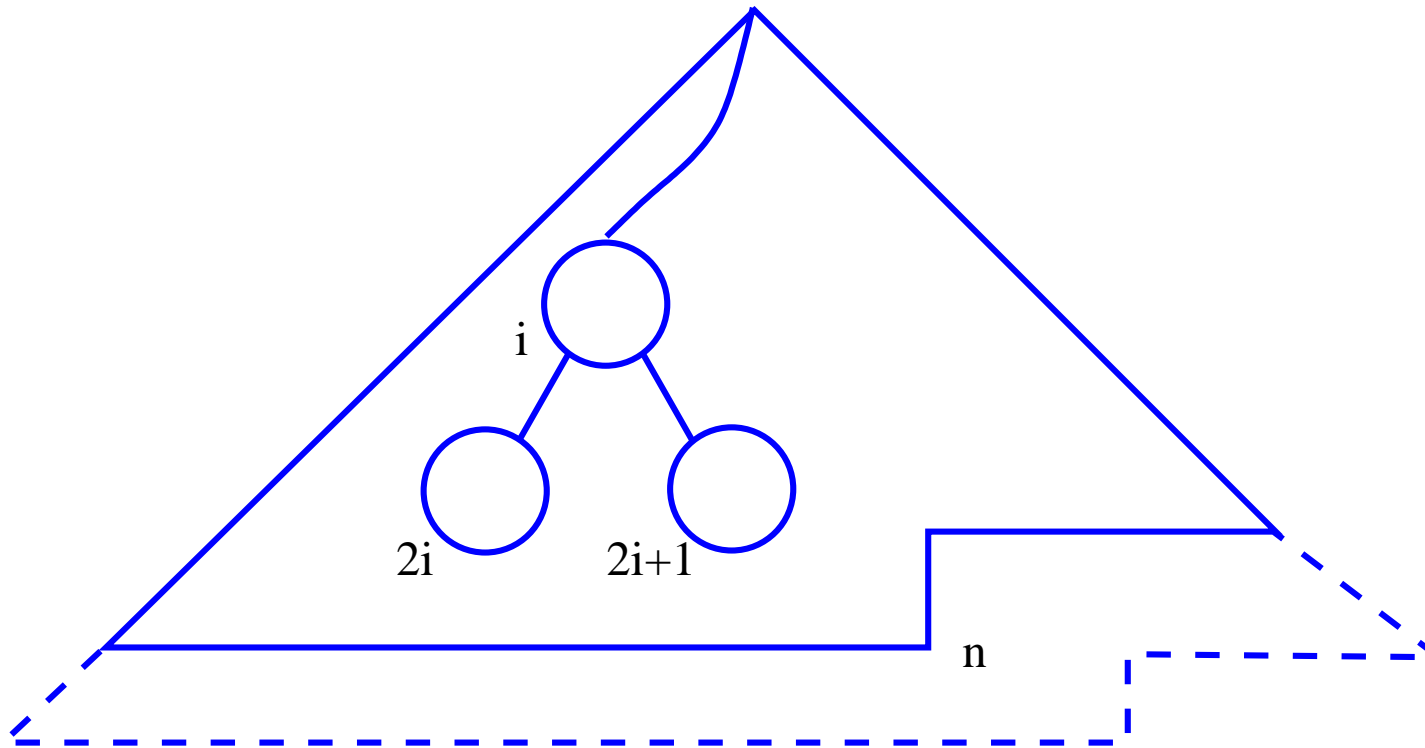
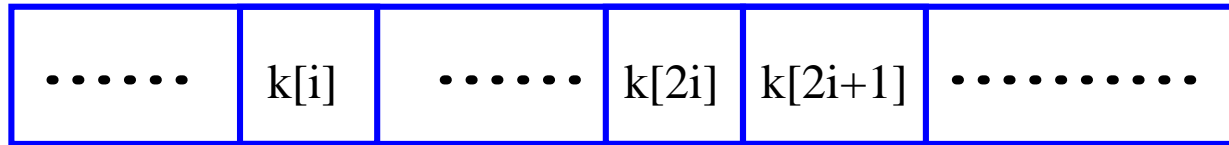


4. Versickere k_3



Versickern

Allgemein: Versickere k_i in $a[i..n]$

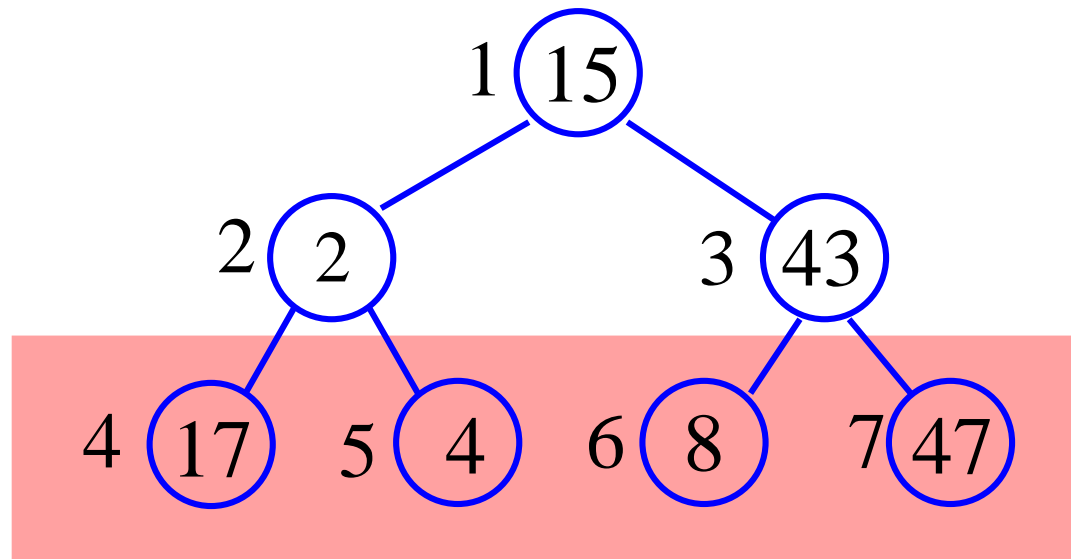


Versickern: Implementierung

```
void pushdown(Sortable a, int i, int n) {
    while (2*i <= n) {                               // i hat linken Sohn
        int j = 2*i;
        if (j < n && a.isLt (j, j+1))
            j = j + 1;                               // 2i+1 ist größerer Sohn
        if (a.isLt (i, j)) {                         // Heapbedingung verletzt
            a.swap(i, j);
            i = j;                                    // versickere weiter beim Sohn
        }
        else
            return;                                  // fertig
    }
}
```

Erstellen eines Heaps

Beobachtung: Die Schlüssel $k_{\lfloor n/2 \rfloor + 1}, \dots, k_n$ erfüllen immer die Heap-Bedingung, da keine Kinder vorhanden!



Erstellen eines Heaps: Implementierung

```
void heapify (Sortable a, int lo, int n) {  
    // verwandle a[lo..n] in einen Heap  
    for (int i = n/2; i >= lo; i--) {  
        pushdown(a, i, n);  
    }  
}
```

Hauptfunktion

```
void heapSort (Sortable a, int n) {  
    heapify (a, 1, n);  
    for (int i = n; i > 1; i--) {  
        // max (a[1..i]) an Wurzel in a[1]  
        a.swap (1, i);  
        pushdown (a, 1, i-1);  
    }  
}
```

Analyse Heapsort

Sei a Heap mit n Schlüsseln

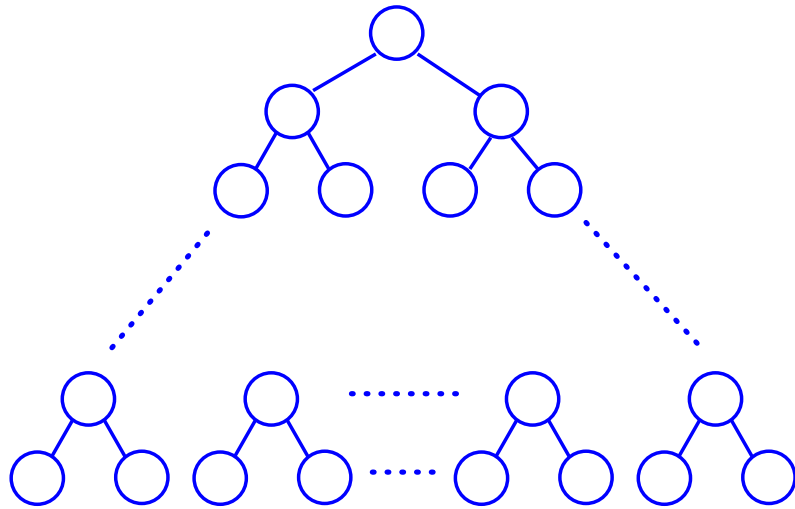
Platzbedarf

- Eingabe $O(n)$
- Hilfsvariable, Prozedurkeller, etc $O(1)$

Laufzeit

- `maximum (a)` in Zeit $O(1)$
- `deletemax (a)` in $2 \log n$ Vergleichen (n -mal durchgeführt)
- `heapify (a)` in $2n$ Vergleichen (s. nächste Seite)
- Gesamtzahl Vergleiche: $\leq 2n \log n + 2n \in O(n \log n)$

Analyse Iteriertes Versickern



Tiefe	# Elemente	# Vergleiche
$(k-1) - (k-1)$	$2^{k-1}/2^{k-1}$	$2(k-1)$
\vdots	\vdots	\vdots
$(k-1) - i$	$2^{k-1}/2^i$	$2i$
\vdots	\vdots	\vdots
$(k-1) - 2$	$2^{k-1}/2^2$	$2 \cdot 2$
$(k-1) - 1$	$2^{k-1}/2^1$	$2 \cdot 1$
$(k-1) - 0$	$2^{k-1}/2^0$	$2 \cdot 0$

Gesamtzahl der Vergleiche

$$\begin{aligned}
 2^{k-1} \sum_{i=0}^{k-1} 2 \frac{i}{2^i} &= 2 \cdot 2^k - 2(k-1) - 4 \\
 &= 2n - 2(\log n - 1) - 4 \\
 &\leq 2n \quad \text{für } n > 2
 \end{aligned}$$

4.4.4 Bottom-Up Heapsort

Schlimmster Fall in Heapsort

Es wird immer das kleinste Element versickert

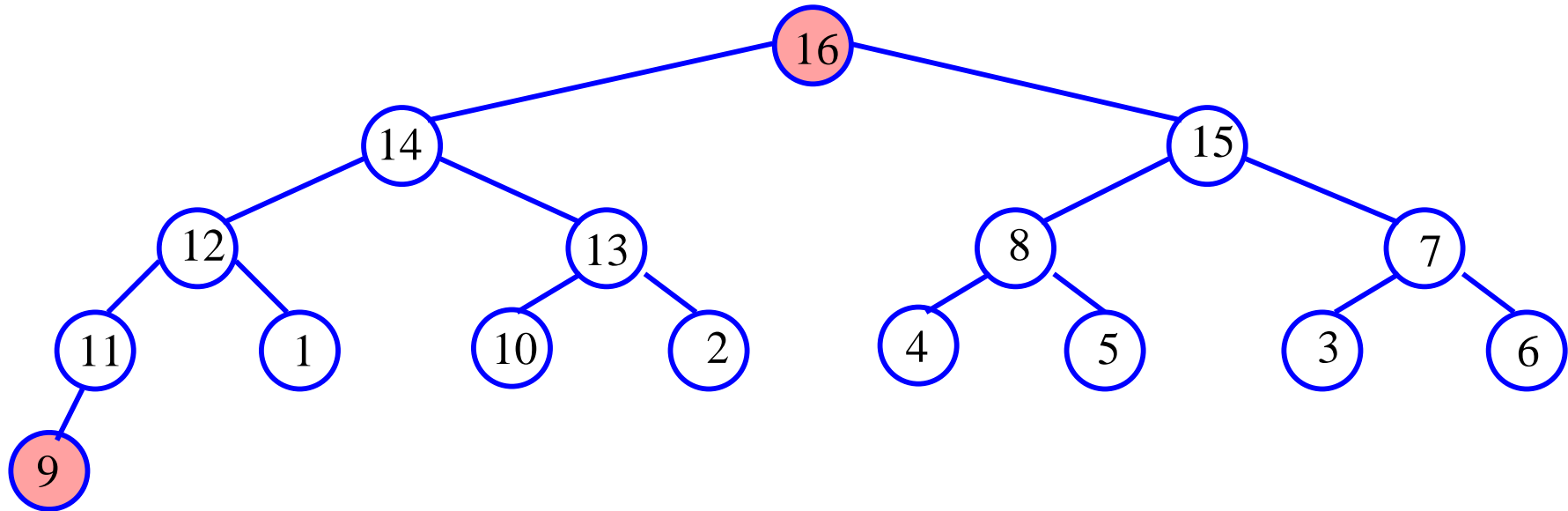
Beobachtung

Die erwartete Tiefe eines versickerten Elementes im Heap ist groß.

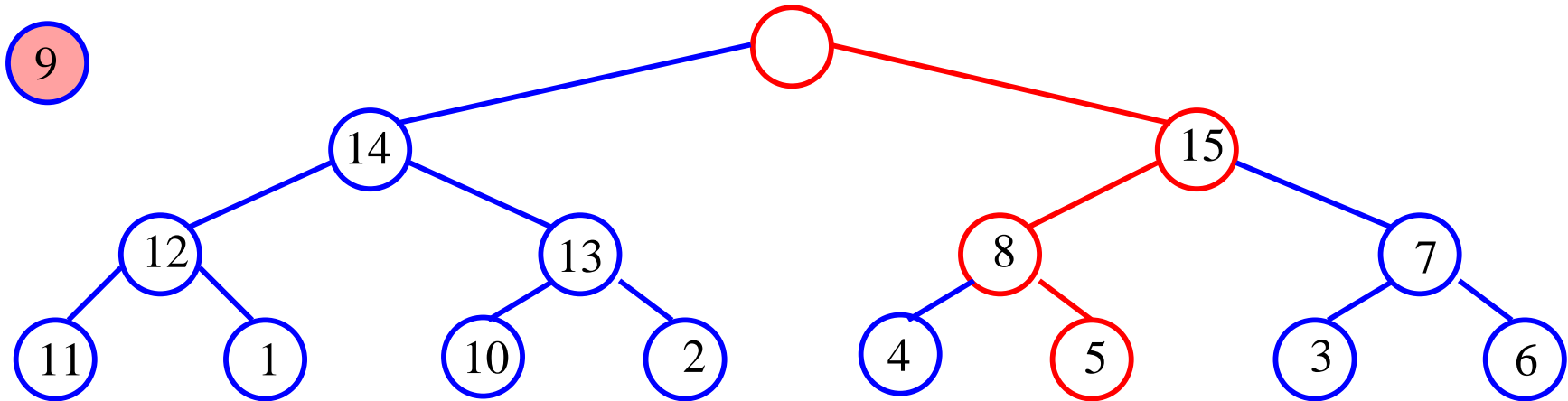
Idee: Bottom-Up-Heapsort

1. Gehe zum größeren der beiden Söhne mit **einem** Schlüsselvergleich pro Niveau
2. Wiederhole 1 bis bei Blatt angelangt (search-leaf)
3. Steige dann wieder auf bis zur Einfügestelle (bottom-up-search)
4. Ringtausch entlang des Pfades zur Wurzel

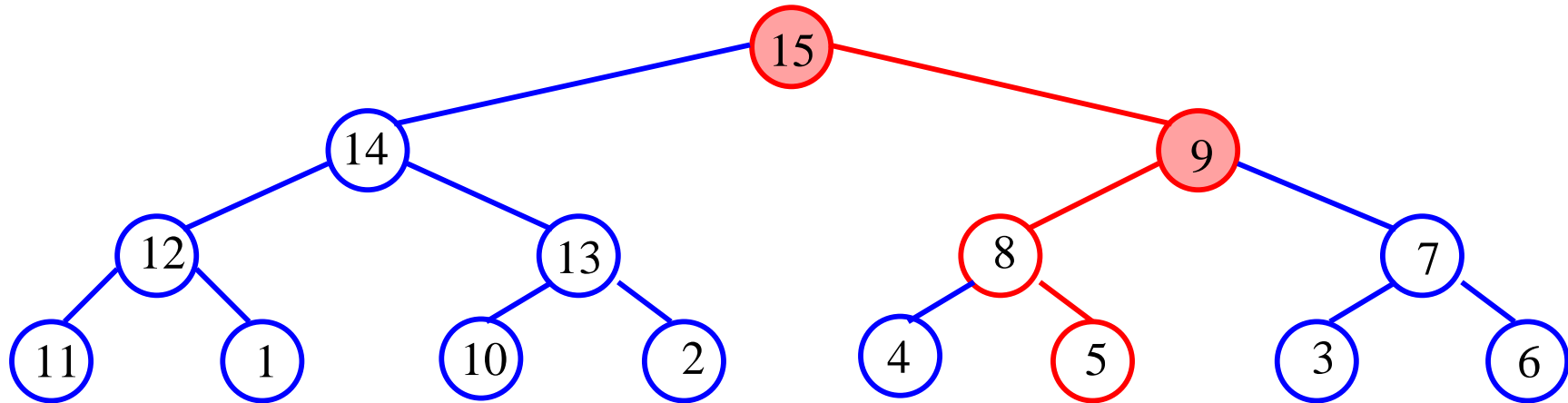
Bottom-Up-Heapsort: Maximum entfernen



Bottom-Up-Heapsort: Pfad durch größere Söhne



Bottom-Up-Heapsort: Wiederaufstieg mit Tausch



Bottom-Up-Heapsort: Implementierung

```
void pushdown(Sortable a, int root, int n) {
    int p = root;
    while (2*p < n) {                // p hat zwei Söhne
        if (a.isLt (2*p, 2*p+1))
            p = 2*p + 1;            // 2*p+1 ist größerer Sohn
        else
            p = 2*p;                // 2*p ist größerer Sohn
    }
    if (2*p == n) p = n;            // nur noch ein Sohn
    // p ist gesuchtes Blatt; suche Vorfahren von p mit a[p] > a[root]
    while (p != root && a.isLe (p, root))
        p /= 2;
    // ‘Ringtausch’ a[root] => a[p] => a[p/2] => ... => a[root]
    for (;p != root; p /= 2)
        a.swap (root, p);
}
```

- Echter Ringtausch effizienter

Ergebnisse

Satz: Bottom-up Heapsort führt zum Sortieren einer Folge von n Schlüsseln im schlechtesten Fall nur $1.5n \log n + (2 - c(n))n + O(\log^2 n)$ Schlüsselvergleiche aus (Wegener 1993).

Fleischer (1991) sowie Schaffer und Sedgewick (1993) haben worst-case Beispiele angegeben, bei denen die Anzahl der wesentlichen Vergleiche für BOTTOM-UP-HEAPSORT gleich $1.5n \log n - o(n \log n)$ ist.

Satz: Im Mittel benötigt Bottom-up Heapsort (nur) $n \log n + O(n)$ Schlüsselvergleiche (Li & Vitányi 1993).

Experimente:

Sei $d(n)$ so gewählt, daß $n \log n + d(n)n$ die erwartete Anzahl an Schlüsselvergleichen von BOTTOM-UP-HEAPSORT ist. Dann liegt $d(n)$ im Intervall von $[0.34, 0.39]$. Diese Zahl ist groß für $n \approx 2^k$ und klein für $n \approx 1.4 \cdot 2^k$.

4.4.5 Quick-Heapsort

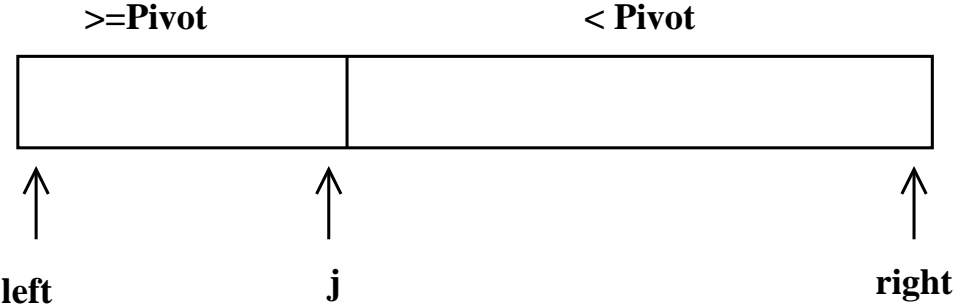
Hybrid-Algorithmus: Divide-and-Conquer Ansatz von *QUICKSORT* mit *HEAPSORT*.

1. Aufteilung des Elementarrays in umgekehrter Richtung, d.h. in linker Hälfte $a[1..j-1]$ *größer* als Pivot $a[j]$ und in rechter Hälfte $a[j+1..n]$ *kleiner-gleich* dem Pivotelement.
2. Nur in der jeweils kleineren Hälfte wird ein Heap aufgebaut und zwar in der linken ein Max-Heap *oder* in der rechten ein Min-Heap.
3. Bei Max-Heap:
 - Maximum ans Ende der rechten Hälfte
 - Einfügen des verdrängten Elements in Max-Heap (als Blatt, da kleiner als Pivot) \Rightarrow aufsteigend sortierte Folge am Ende der rechten Hälfte

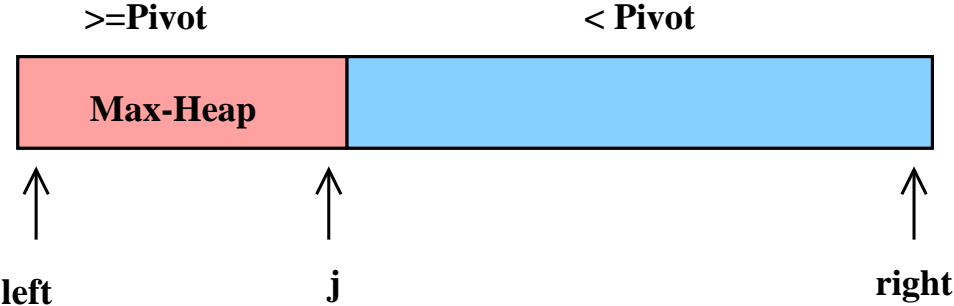
Satz: Im Mittel sortiert *QUICK-HEAPSORT* n Elemente in $\leq n \log n + 3n + o(n)$ Vergleichen.

4.4.6 Quick-Heapsort: Veranschaulichung

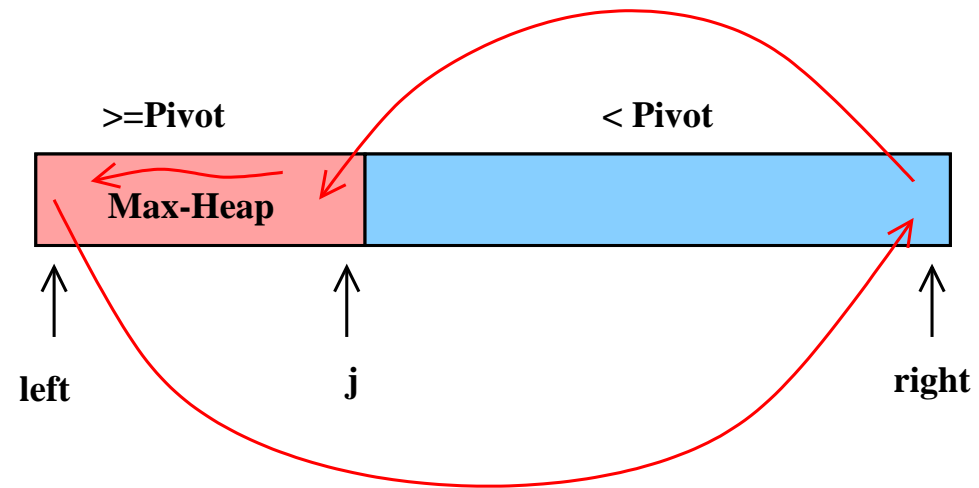
Nach Aufteilung



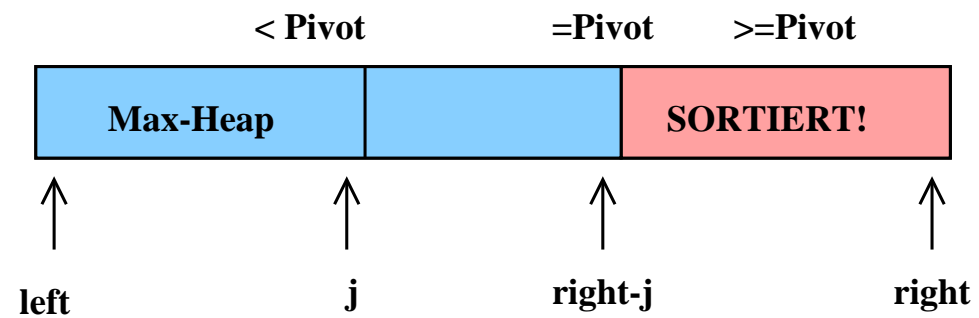
Nach Heapaufbau



Externes Versickern



Ergebnis



Aufteilen

```
static void quickHeapSort (Sortable a, int left, int right) {
    while (right-left > 0) {
        int i = left, j = right+1;
        do {
            do j--; while (j>=i && a.isLt (j, left));
            do i++; while (i<=j && a.isLe (left, i));
            if (j > i) a.swap(i,j);
        } while(j >= i);
        // all x in a[j+1..right] : x < left
        // all y in a[left..i-1]   : x >= left
        a.swap(left, right - (j-left));    // move pivot to final position
        if( (j-left) >= (right-j) ) {
            external_minheap_sort( a, j+1, right, left );
            left = right - (j-left) + 1;
        } else {
            external_maxheap_sort( a, left+1, j, right );
            right = right - (j-left) - 1;
        }
    }
}
```

External Maxheap Sort (Minheap Sort analog)

```
static void external_maxheap_sort
  (Sortable a, int heapleft, int heapright, int workright) {
  build_max_heap(a, heapleft, heapright);
  for(int j=heapright; j>=heapleft; j--) {
    int l = max_special_leaf(a, heapleft, heapright);
    a.swap (l, workright--);
  }
}
```

Blattsuche (analog in Minheap)

```
static int max_special_leaf (Sortable a, int left, int right) {
    int offset = left - 1;
    int n = right - left + 1;
    int father = 1, i = 2; // left son of root
    while( i < n ) {
        if (a.isLt (offset + i, offset + i+1))
            i++; // right son is bigger
        a.swap (offset + father, offset + i); // just move
        father = i; i *= 2;
    }
    if (i == n) {
        a.swap (offset + father, offset + i);
        father = i;
    }
    return left + father;
}
```

4.5 Radixsort

- Nutzt Zahldarstellung der Schlüssel anstelle von Vergleichen
- Schlüssel k ist Wort über Alphabet mit m Elementen

$$k = (k_l, k_{l-1}, \dots, k_1, k_0)_m = \sum_{i=0}^l k_i m^i \quad \text{wobei} \quad k_i \in \{0, \dots, m-1\}$$

- $m = 10$ Dezimalzahlen
- $m = 2$ Binärzahlen
- $m = 26$ Zeichenketten über $\{a, \dots, z\}$
- Definiere Funktion: $z_m(k, i) = k_i$
- **Annahme:** $z_m(k, i)$ braucht konstante Zeit

4.5.1 Radix-exchange-sort

Sortieren durch rekursive Aufteilung nach höchstwertigen Ziffern (Bits)

Eingabe:

- Folge xs von binären Schlüsseln mit Bits an Positionen $0, \dots, l$,
- Bitposition $b \leq l$

Ausgabe: Nach Bitpositionen $\leq b$ sortierte Folge xs

```
radixExchangeSort (xs, b) =  
  if length xs <= 1 || b < 0  
  then xs  
  else let xs0 = [ x <- xs | z2 (b, x) == 0 ]  
        xs1 = [ x <- xs | z2 (b, x) == 1 ]  
        in radixExchangeSort (xs0, b - 1)  
        ++ radixExchangeSort (xs1, b - 1)
```

Beispiel

$$\begin{aligned} \mathbf{xs} &= (6, 7, 4, 2, 3) \\ &\hat{=} (110, 111, 100, 010, 011) \end{aligned}$$

	110	111	100	010	011
$b = 2$	011	010	100	111	110
$b = 1$	011	010	100	111	110
$b = 0$	010	011		110	111

Problem: Informationsgewinn pro Operation nur ein Bit

Definition 4 Die *unterscheidenden Präfixe* einer Menge $\{a_1, \dots, a_n\}$ von Zeichenketten sind die kürzesten Präfixe von $\{a_1, \dots, a_n\}$, die paarweise verschieden sind.

Das unterscheidende Präfix einer Zeichenkette a_i , die Präfix einer anderen Zeichenkette ist, ist a_i selbst.

Beispiel:

SOCKEL

SOCRATES

SODA

SOFORT

SOFA

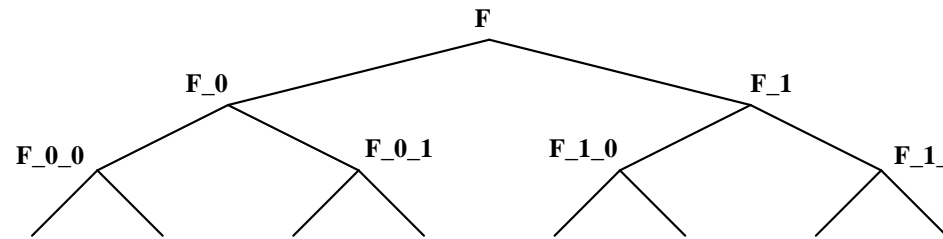
SORT

SORTIEREN

$s_i =$ unterscheidendes Präfix von a_i

Definiere $S = \sum_{i=1}^n |s_i|$

Analyse von Radix-exchange-sort



Laufzeit von Radix-exchange-sort:

- konstanter Aufwand pro Aufteilungsschritt für a_i (Bit b testen, tauschen)
- # Aufteilungsschritte für $a_i = |s_i|$

$$T_{RES}(a_1, \dots, a_n) = \sum_{i=1}^n |s_i| = O(n + S)$$

Alternative Rechnung:

$$\begin{aligned} T(a_1, \dots, a_n) &= |\{(a_i, A) \mid a_i \text{ beteiligt an Aufteilungsschritt } A\}| \\ &= \sum_{i=1}^n |\{A \mid a_i \text{ beteiligt an Aufteilungsschritt } A\}| \end{aligned}$$

4.5.2 Sortieren durch Fachverteilung

Idee: zuerst nach dem letzten Zeichen sortieren, dann nach vorletztem ...

Beispiel: $F = (434, 528, 154, 176, 783, 204, 351, 218, 900)$

Verteilen nach Ziffer 3:

				204					
				154				218	
<u>900</u>	<u>351</u>	___	<u>783</u>	<u>434</u>	___	<u>176</u>	___	<u>528</u>	___
F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9

Sammeln: 900, 351, 783, 434, 154, 204, 176, 528, 218

Aufgesammelt: 900, 351, 783, 434, 154, 204, 176, 528, 218

Verteilen nach Ziffer 2 :

	204					154			
	<u>900</u>	<u>218</u>	<u>528</u>	<u>434</u>	—	<u>351</u>	—	<u>176</u>	<u>783</u>
	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8
									F_9

Sammeln: 900, 204, 218, 528, 434, 351, 154, 176, 783

Verteilen nach Ziffer 1:

		176	218						
	—	<u>154</u>	<u>204</u>	<u>351</u>	<u>434</u>	<u>528</u>	—	<u>783</u>	—
	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8
									F_9

Sammeln: 154, 176, 204, 218, 351, 434, 528, 783, 900

Sortieren durch Fachverteilung—Analyse

n Anzahl der Schlüssel | l maximale Schlüssellänge in Bits | m Anzahl der Fächer

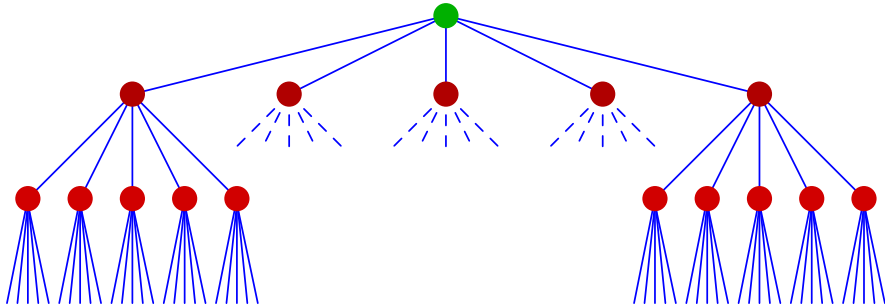
- Anzahl Durchläufe $l/\log m$
- Aufwand in Durchlauf i : $O(m + n)$
 - Für jeden Schlüssel: i -tes Zeichen ermitteln, Anhängen an Fachliste, Aufsammeln — $O(n)$
 - Für jedes Fach: Testen auf Leerheit — $O(m)$

⇒ Gesamtaufwand: $O((m + n)l/\log m)$

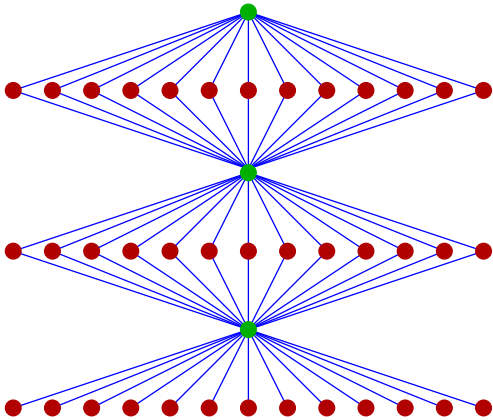
- Speicherplatz: $O(m + n)$

Vergleich der Aufteilungen

Radix-exchange-sort:

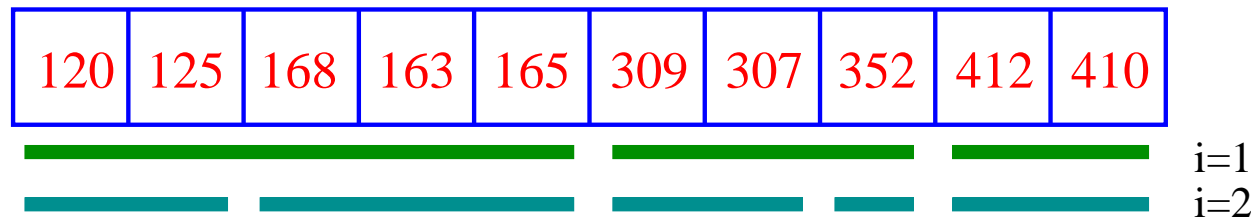


Sortieren durch Fachverteilung:



4.5.3 Forward-Radixsort

- Verwendung von m Fächern
- Fachverteilung nach **höchstwertigen** Ziffern
- Verteilung **aller** Schlüssel in einem Schritt
- Verwendung von **Gruppen**
- **Invariante:** Nach dem i -ten Durchlauf sind die Elemente nach den ersten i Zeichen sortiert
- **Gruppe G :** aufeinanderfolgende Elemente, wobei erste i Zeichen gleich sind
 - fertig, falls $|G| = 1$
 - unfertig, falls $|G| \geq 2$



Forward-Radixsort

Eingabe: Folge $F = (a_1, \dots, a_n)$ von Elementen | **Ausgabe:** Sortierte Folge

```
Gruppe[1] = F      // alle Elemente, beginne bei 1. Zeichen
for ( i = 1; es gibt eine unfertige Gruppe; i++ )
  for (g = 1; g <= Anzahl Gruppen; g++)
    if (Gruppe[g] unfertig)
      Verteile Gruppe[g] nach dem  $i$ -ten Zeichen (markiert mit Gruppennummer)
      auf die  $m$  Fächer

Durchlaufe Fächer in aufsteigender Reihenfolge und
  sammele Elemente in ihren Gruppen

for (g = 1; g <= Anzahl Gruppen; g++)
  Durchlaufe Gruppe[g]:
    if ( $z_m(a_j, i) \neq z_m(a_{j-1}, i)$ )
      beginne neue Gruppe mit Nummer  $j$ 
```

Beispiel Forward-Radixsort

Beispiel: $F = 434, 528, 154, 176, 783, 204, 351, 218$

Verteilen nach Ziffer 1:

	176	218							
	154	204	351	434	528		783		900
F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9

Sammeln:

1	2	3	4	5	6	7	8	9
154	176	204	218	351	434	528	783	900
1	1	3	3	5	6	7	8	9

Verteilen nach Ziffer 2:

$$\begin{array}{cccccccc} \frac{204(3)}{F_0} & \frac{218(3)}{F_1} & \frac{\quad}{F_2} & \frac{\quad}{F_3} & \frac{\quad}{F_4} & \frac{154(1)}{F_5} & \frac{\quad}{F_6} & \frac{176(1)}{F_7} \end{array}$$

Sammeln:

1	2	3	4	5	6	7	8	9
154	176	204	218	351	434	528	783	900
1	2	3	4	5	6	7	8	9

Analyse Forward-Radixsort

$$S_{max} = \max\{|s_i| \mid 1 \leq i \leq n\}$$

Laufzeit von Forward-Radixsort:

- konstanter Aufwand pro Aufteilungsschritt für a_i :
 - verteilen nach Ziffer i
 - sammeln nach Gruppennummer
 - Gruppennummer ändern
- # Iterationen für a_i : $|s_i|$
- # Iterationen insgesamt: $S_{max}/\log m$
- Aufwand pro Iteration: m

$$\begin{aligned} T_{FR}(a_1, \dots, a_n) &= \sum_{i=1}^n |s_i| / \log m + m S_{max} / \log m \\ &= O(n + S / \log m + m S_{max} / \log m) \end{aligned}$$

4.5.4 Vergleich

	Operationen
Radix-exchange-sort	$O(n + m \cdot S / \log m)$
Fachverteilung	$O(n \cdot l / \log m + m \cdot l / \log m)$
Forward Radixsort	$O(n + S / \log m + m \cdot S_{max} / \log m)$
Quicksort	$O(n \log n (l/w))$

$w =$ Maschinenwortlänge, $S \leq n \cdot l$

4.6 Zusammenfassung

Name	Schlecht	Durchschn.	in situ	stabil
Bubblesort	$O(n^2)$	$O(n^2)$	ja	ja
Auswahlsort	$O(n^2)$	$O(n^2)$	ja	nein
	wenig Transfers: $O(n)$			
Einfügesort	$O(n^2)$	$O(n^2)$	ja	ja
	als Sub-sort z.B. in Quicksort			
Mergesort	$O(n \log n)$	$O(n \log n)$	nein	ja
Quicksort	$O(n^2)$	$O(n \log n)$	nein	nein
	meistbenutzt, viele Varianten,			
Heapsort	$O(n \log n)$	$O(n \log n)$	ja	nein
	$2n \log n$ gegenüber QS: $1.3n \log n$			
Bottom-Up Heaps.	$O(n \log n)$	$O(n \log n)$	ja	nein
Quick-Heapsort	$O(n^2)$	$O(n \log n)$	ja	nein
	$1n \log n$ gegenüber QS: $1.3n \log n$			
Radix-Exchange-Sort	$O(n + S)$	$O(n + S)$	nein	nein

4.6.1 Untere Schranke für allgemeine Sortierverfahren

Satz Zum Sortieren einer Folge von n Schlüsseln mit einem allgemeinen Sortierverfahren (nur Schlüsselvergleiche) sind im Worst-case ebenso wie im Mittel wenigstens $\Omega(n \log n)$ Vergleichsoperationen zwischen zwei Schlüsseln erforderlich.

Modellierung von allgemeinen Sortierverfahren:

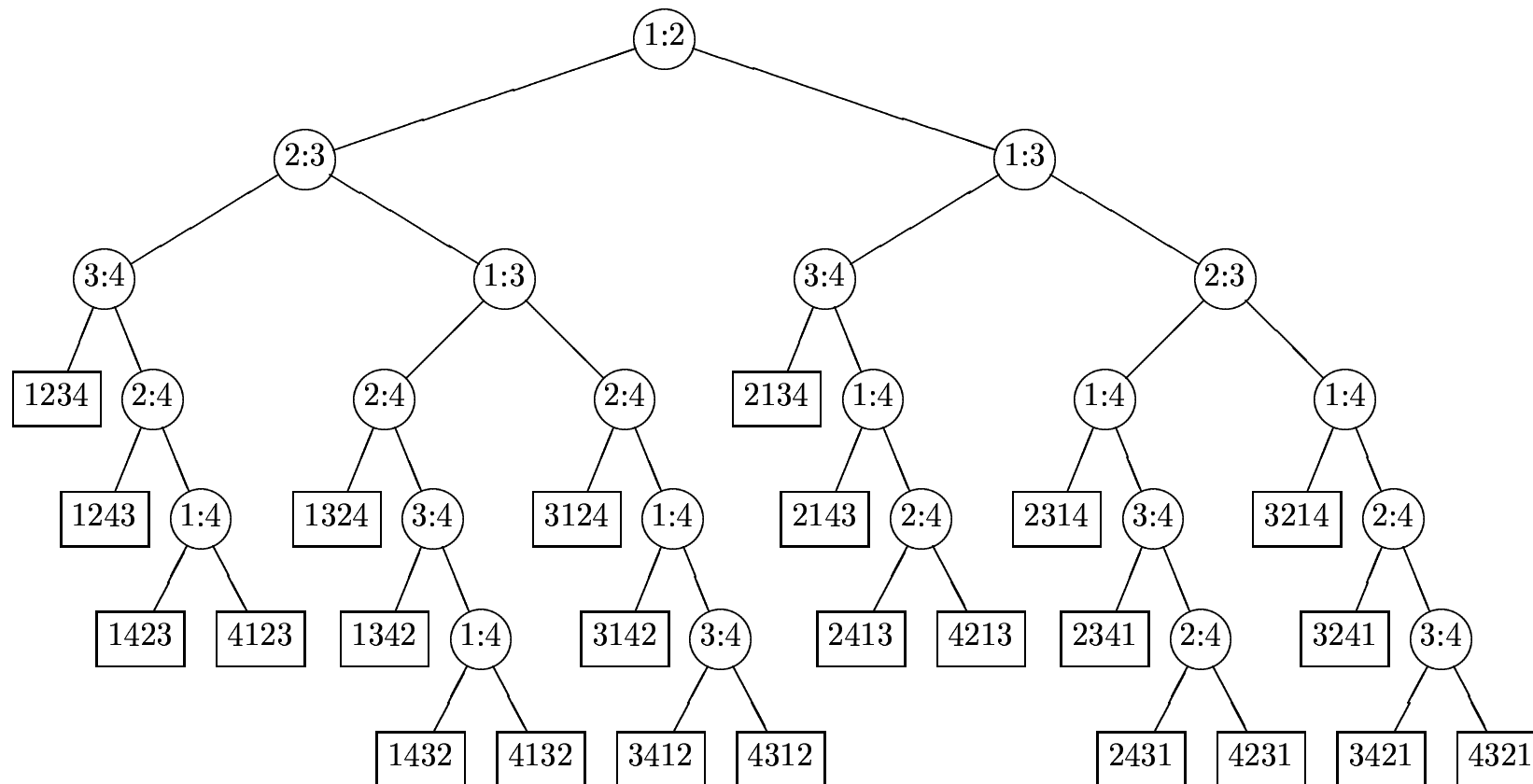
⇒ **Entscheidungsbäume.**

Entscheidungsäume

- Sortierverfahren A
- Entscheidungsbaum für den Ablauf von A auf Folgen der Länge n enthält
 - für jede der $n!$ Permutationen ein Blatt
 - jeder innere Knoten repräsentiert eine Vergleichsoperation und hat zwei Söhne
 - Weg W von der Wurzel zu einem Blatt v :
 - * die Vergleiche an den Knoten von W identifizieren die Permutation π_v von v
 - * entsprechen den von A durchgeführten Vergleichen, falls die Eingabe π_v ist
 - Längster/mittlerer Weg von der Wurzel zu einem Blatt
 - $\hat{=}$ max. bzw. mittlere Anzahl von Vergleichen.

Beispiel Entscheidungsbaum

Sortieren durch Einfügen für Folge $F = (k_1, k_2, k_3, k_4)$ von 4 Schlüsseln



Maximale und mittlere Tiefe von Binärbäumen

(Tiefe eines Knotens = Weglänge von Wurzel)

Satz Die maximale und die mittlere Tiefe eines Blattes in einem Binärbaum mit k Blättern ist mindestens $\log k$.

Beweis

1. Sei $d(k)$ die maximale Tiefe eines Binärbaums mit k Blättern.

Zeige $d(k) \geq \log k$ mit Induktion über k .

Induktionsverankerung $k = 1$: $d(1) = 0 = \log 1$

Induktionsschritt: für $2k > 1$ gilt

$$\begin{aligned}d(2k) &= 1 + \max\{\max(d(k_1), d(k_2)) \mid k_1, k_2 > 0, 2k = k_1 + k_2\} \\ &\geq 1 + \max(d(k), d(k)) \\ &= 1 + d(k) \\ &\geq 1 + \log(k) \\ &= \log 2k\end{aligned}$$

2. Mittlere Tiefe $\geq \log k$: Mit Induktion.

Beweis der unteren Schranke

Beachte: $n! \geq (n/2)^{n/2}$.

Mit dieser Abschätzung folgt, dass die Entscheidungsbäume (die den Ablauf der Sortieralgorithmen modellieren) eine maximale und mittlere Tiefe von mindestens

$$\log(n!) \geq n/2 \log(n/2) \in \Omega(n \log n)$$

haben.