Heapsort IX

Algorithmus:

Lokale Hilfsprozedur `korrigiere` lässt das Element $a[li]$ in das als Teilheap vorausgesetzte Teilarray $a[li+1], \ldots, a[r]$ an die richtige Stelle einsinken, so dass $a[li], \ldots, a[r]$ einen Heap bildet.

```
void HeapSort() {
   // Aufbau des Heap
   for (int i=n/2; i>=1; i--) korrigiere(i, n);
   // Sortierschritte
   for (int i=n; i>=2; i--) {
      vertausche(i, 1);
      korrigiere(1, i-1);
   }
}
```

Heapsort X


```
void korrigiere(int li, int r) {
   int i, j;
   Entry x;
   i = li; x = a[li-1]; // abzusenkendes Element in der Wurzel
   if ((2*li) <= r) { // rechte Grenze nicht überschritten
      j = 2*i; // a[j-1] ist linker Sohn von a[i-1]
      do { // wiederholtes Absenken
         if (j < r) { // suche größeren der beiden Söhne
            if (a[j-1].key < a[j].key) j = j+1;
         // nun: a[j-1] ist größter Sohn
         if (x.key < a[j-1].key) {
            vertausche(i-1, j-1);
            i = j; j = 2*i; // Absenken
         } else j = r+1; // halte an: Heap-Bedingung erfüllt
      } while (j <= r);
   }
}
```

Heapsort XI

Laufzeitanalyse:

Aufteilung des Algorithmus in zwei getrennte Schritte:

1. Aufbau des ‚Anfangheap‘:
   - `korrigiere` wird einmal für jeden Knoten aufgerufen, der mind. einen Sohn hat.
   - Für die Höhe h eines vollständig ausgeglichenen binären Baumes gilt: $h = \lceil \log_2(n+1) \rceil$
   - Für n gilt: $2^{h-1} \leq n < 2^h$
   - In einem derartigen Baum besitzt Level i genau 2^{i-1} Knoten für $1 \leq i \leq h-1$.
   - Für jeden dieser Knoten: Darin enthaltene Schlüssel muss maximal bis in einen Blattknoten wandern, d.h. einschließlich des Knotens selbst $h-(i-1)$ Knoten durchlaufen

Heapsort XII

- Es gilt:

$$T_{worst}^{Aufbau}(n) \leq \sum_{i=1}^{h-1} 2^{i-1}\big(h-(i-1)\big) = \sum_{i=0}^{h-2} 2^{i}(h-i) = \sum_{i=2}^{h}\big(2^{h-i}\cdot i\big) =$$

$$= \sum_{i=2}^{h}\left(\frac{2^{h-1}}{2^{i-1}}\cdot i\right) \leq n\sum_{i=2}^{h}\frac{i}{2^{i-1}} = O(n)$$

- Das letzte „=“ gilt, da $\sum_{i=2}^{h}\frac{i}{2^{i-1}}$ für $h \to \infty$ konvergiert.

Heapsort XIII

2. Sortierung durch sukzessive Entnahme des Maximums:
   - $(n-1)$-mal muss jeweils ein Schlüssel durch `korrigiere` abgesenkt werden.
   - Jedes Absenken ist auf einen Pfad von der Wurzel zu einem Blatt beschränkt.
   - $T_{worst}^{Sortierung}(n) \leq (n-1) \cdot \lceil \log_2(n-1+1) \rceil = O(n \cdot \log n)$

Schlussfolgerung:

- Die Laufzeit von Heapsort im schlechtesten Fall ist also $O(n \cdot \log n)$ und damit optimal.

Heapsort XIV

Bemerkungen:

- 50% der Knoten eines Heaps liegen auf der untersten Ebene, 25% der Knoten eines Heaps liegen auf der vorletzten Ebene, etc…
  - Wahrscheinlichkeit eines absinkenden Elementes, „weit unten“ zu landen, ist groß.
  - Durchschnittlicher Zeitbedarf nahe bei $2 \cdot n \cdot \log n + O(n)$.
  - Schlechtes Durchschnittsverhalten im Vergleich zu Quicksort $(1{,}39 \cdot n \cdot \log n + O(n))$.
- Da Heap-Aufbau nur $O(n)$ Zeit benötigt, kann Heapsort dazu benutzt werden, die k größten (kleinsten) Elemente einer Schlüsselmenge in $O(n + k \cdot \log n)$ Zeit zu finden.
  Ist $k \leq \frac{n}{\log n}$ kann dieses Problem in $O(n)$ Zeit gelöst werden.

Heapsort XV

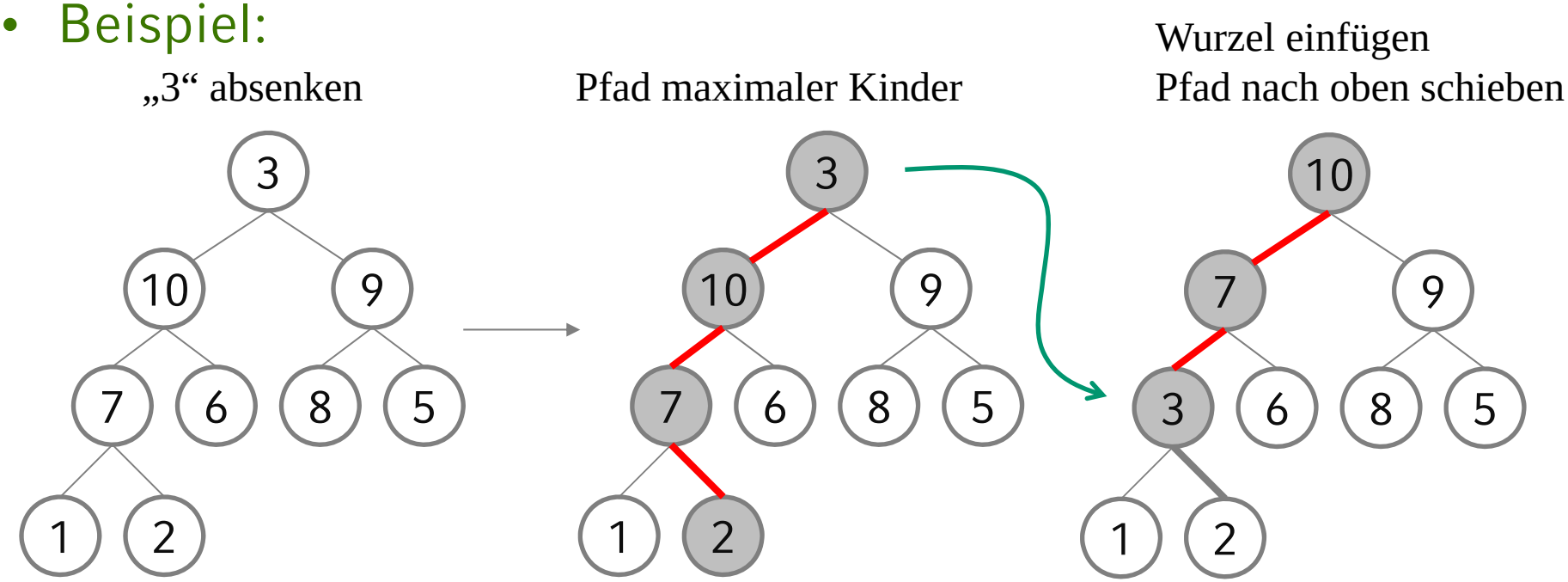
- Verbesserung: **Bottom-Up-Heapsort** (Wegener 1990)
  - **Bisher:** Beim Absenken eines Schlüssels jeweils zwei Vergleiche:
    - „Welcher Sohn ist größer?“
    - „Ist dieser größer als der abzusenkende Schlüssel?“ → meistens „ja“.
  - **Jetzt:** Betrachte nur noch die 1. Frage (Zielpfad von Wurzel zu Blatt)
  - Algorithmus (Veränderung der Absenkprozedur von Heapsort):
    - Wurzelelement x (in Knoten p_0) soll abgesenkt werden
    - Bestimme Pfad $p = <p_0,p_1,..,p_n>$ (Wurzel->Blatt) durch jeweilige Auswahl des größten Kindes → Pfad maximaler Kinder
    - Durchlauf des resultierenden Pfades p von unten (hinten) nach oben (vorne) (Bottom-Up), dabei:
      falls $p_i > x$
      - Schiebe Wurzelelement x an die Stelle von Pfadelement p_i;
        Schiebe Teilpfad $<p_1,p_2,..,p_i>$ um eins nach oben, d.h. p_1 bildet neue Wurzel.

  → Zahl der Vergleiche wird reduziert

  → beste bekannte Sortierverfahren für große n ($n > 16.000$)

Heapsort XV

- Beispiel:

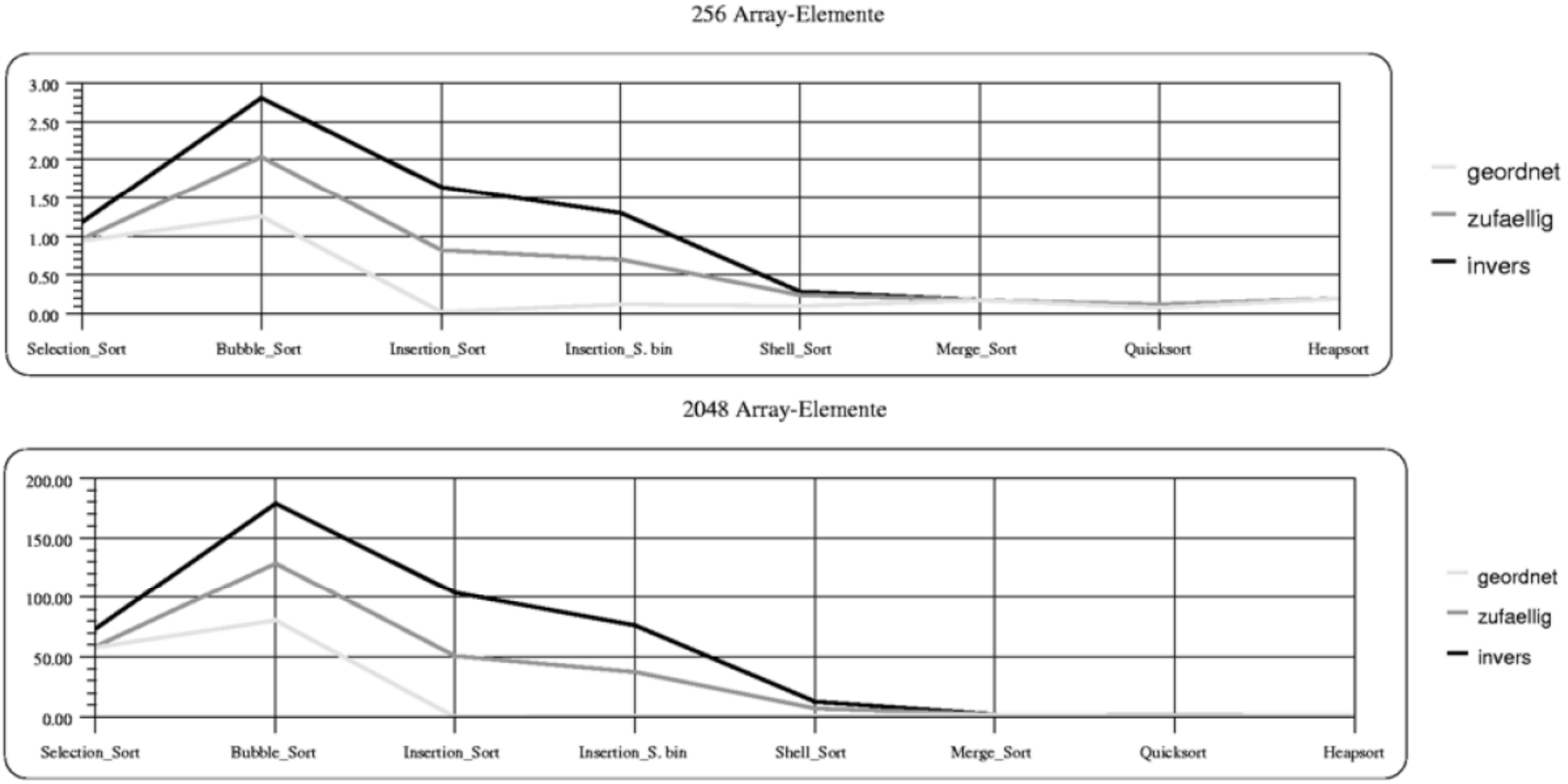


- Fazit (Bottom-Up Heapsort):
  - → Zahl der Vergleiche wird reduziert
  - → beste bekannte Sortierverfahren für große n ($n > 16.000$)

3.4. Experimentell ermittelte Laufzeit von Sortierverfahren

Vergleich Sortierverfahren I

- Experimenteller Vergleich der Laufzeit verschiedener Sortierverfahren (Wirth)



Vergleich Sortierverfahren II

	Anzahl der Elemente = 256			Anzahl der Elemente = 2048		
	geordnet	zufällig	invers	geordnet	zufällig	invers
Selection_Sort	0,94	0,96	1,18	58,18	58,34	73,46
Bubble_Sort	1,26	2,04	2,80	80,18	128,84	178,66
Insertion_Sort	0,02	0,82	1,64	0,22	50,74	103,80
Insertion_Sort (binär)	0,12	0,70	1,30	1,16	37,66	76,06
Shell_Sort	0,10	0,24	0,28	0,80	7,08	12,34
Merge_Sort	0,18	0,18	0,18	1,98	2,06	1,98
Quicksort	0,08	0,12	0,08	0,72	1,22	0,76
Heapsort	0,20	0,20	0,20	2,32	2,22	2,12

Vergleich Sortierverfahren III

- **Leistungsvergleich**
  Bottom-Up-Heapsort ⇔ Median-basierter Clever Quicksort

n	Clever Quicksort	Bottom-Up-Heapsort	Verhältnis
100	5,74	6,94	0,827
200	6,88	7,99	0,861
500	8,42	9,31	0,904
1000	9,60	10,31	0,931
2000	10,78	11,31	0,953
5000	12,35	12,66	0,976
10000	13,54	13,66	0,991
20000	14,72	14,67	1,003
30000	15,42	15,24	1,012

Vergleich Sortierverfahren I

Folgerungen:

- Clever Quicksort ist schneller für kleine n.
- Leistungsfähigkeit von Bottom-Up-Heapsort nimmt für wachsendes n zu.
- Ab ca. $n = 16.000$ ist Bottom-Up-Heapsort besser als Clever Quicksort.
- Für weiter wachsendes n nimmt dessen leistungsvorteil weiter zu
  → Bottom-Up-Heapsort wird 15% besser für $n = 10^{20}$.

3.5. Sortieren in linearer Zeit

Sortieren in linearer Zeit I

Aussage: Sortieren auf Basis von Schlüsselvergleichen:
Im schlechtesten Fall mindestens $O(n \cdot \log n)$ Operationen

Aber: Wenn Schlüsselwerte bestimmte Einschränkungen unterliegen und andere Operationen als Vergleiche angewendet werden können
→ Möglichkeit, in schnellerer Zeit als $O(n \cdot \log n)$ zu sortieren (z.B. in $O(n)$).

Folgende Annahmen:

- Die auftretenden Schlüssel sind a-priori bekannt
- Wertebereich der Schlüssel ist bekannt und relativ klein

Sortieren in linearer Zeit II

Beispiel:

Bekannt, dass die Schlüsselfolge von $a_1, a_2, \ldots, a_n$ eine Permutation von $\{1, 2, \ldots, n\}$ darstellt.

→ Folgende Prozedur „Sort“ zur Sortierung:

```
void Sort() {
   for (int i=1; i<=n; i++) {
      while (a[i].key) != i)
         vertausche(i, a[i].key);
      }
   }
}
```

- Benötigt nur $O(n)$ Zeit für die Sortierung von n Elementen.
- **Bucket Sort** als Verallgemeinerung dieses Verfahrens.

Bucket Sort I

Bucket Sort:

Prinzip:

- Wertebereich der Schlüssel gegeben als $[0, 1, \ldots, m-1]$
- Duplikate der Schlüssel seien erlaubt
- Gegeben sei eine Folge von Behältern (buckets) $B_0, B_1, \ldots, B_{m-1}$.
- Jeder Bucket kann eine Liste von Elementen aufnehmen
- Wir benutzen gleiche Array-Datenstruktur wie bei offenen Hashverfahren, d.h. mit Überlauflisten.
- Neue Elemente werden am Ende der Überlaufliste eingefügt

Bucket Sort II

Algorithmus:

```
ALGORITHMUS BucketSort; // Feld a sei globale Variable
BEGINN
   FOR i := 1 TO n DO
      füge a[i].key in das Bucket B_{a_1.key} ein
   END;
   FOR i := 0 TO m-1 DO
      schreibe die Elemente von B_i in die Ergebnisfolge
   END;
END BucketSort;
```

- Dieser Algorithmus wird auch als „**Sortieren durch Fachverteilung**“ bezeichnet.

Bucket Sort III

Laufzeitanalyse:

- Einfügen eines Elementes in ein Bucket: $O(1)$ Zeit
  (z.B. lineare Liste mit Zeiger auf letztes Element)
- Bucket Sort benötigt $O(n + m)$ Zeit,
  falls $m = O(n) \rightarrow O(n)$ Zeit

Bucket Sort IV

Bemerkung:

- Zuordnen eines Schlüssels a_i zum ‚richtigen' Bucket $B_{a_i.key}$ stellt eine m-wertige Vergleichsoperation dar.
- Gleichwertig zu $\log_2 m$ binären Vergleichsoperationen
- Algorithmus steht nicht im Widerspruch zur ‚theoretischen Komplexität' von Sortierverfahren von $O(n \cdot \log n)$.

- Häufig haben Schlüssel einen sehr großen Wertebereich
  → zu viele Buckets notwendig!
  → **Bucket Sort in mehreren Phasen**
  (k Phasen, falls Wertebereich der Schlüssel $[0,1,\ldots,m^k-1]$)

Bucket Sort V

Beispiel:

Sei $k = 2$ und der Wertebereich der Schlüssel $[0,1, \ldots, m^2 - 1]$.

Dann werden die beiden folgenden Sortierphasen durchgeführt:

1. Phase: Sortierung *innerhalb* der Buckets

Bucket Sort mit m Buckets, wobei a_i in das Bucket $B_{a_i.key\ MOD\ m}$ eingefügt wird.

2. Phase: Sortierung *zwischen* den Buckets

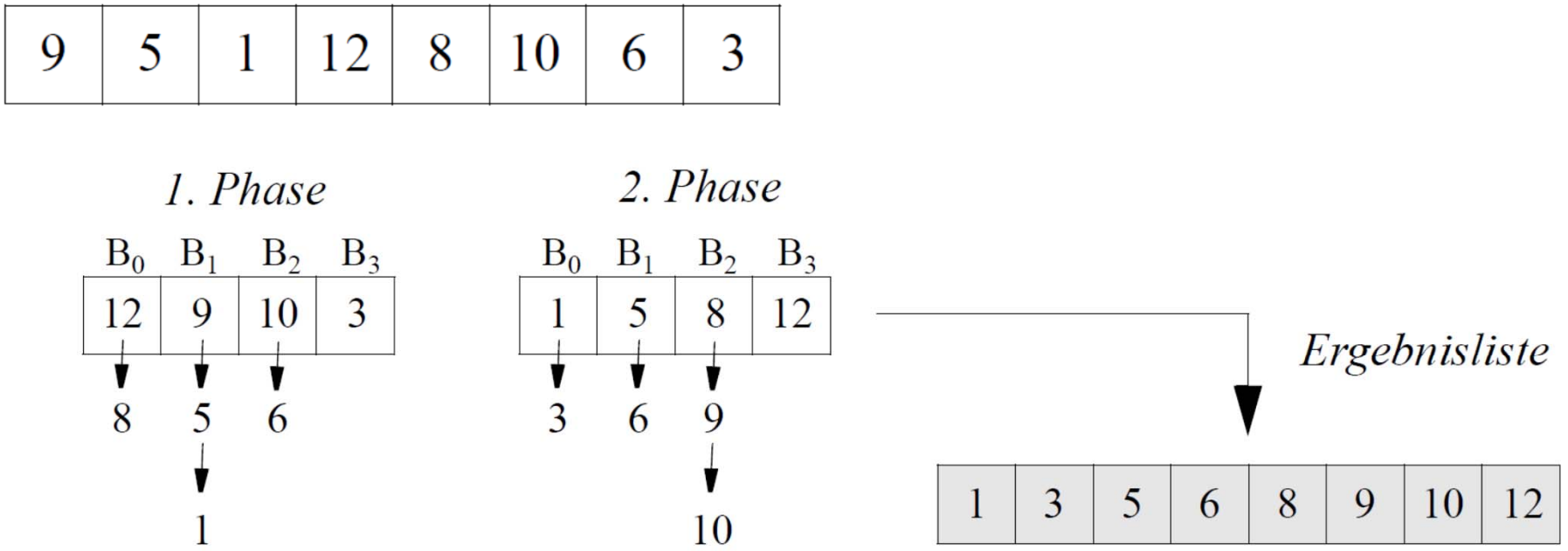
Durchläuft die Buckets der 1. Phase und hängt a_i jeweils an die Liste von Bucket $B_{a_i.key\ DIV\ m}$ an.

Ergebnisse der 2. Phase werden in die Ergebnisliste geschrieben

Bucket Sort VI

Beispiel: (veranschaulicht)

Sei $m = 4$, Schlüsselintervall $= [0, \ldots, 15]$



Nimmt man k als Konstante an → Verfahren in $O(n)$ Zeit, falls das Anhängen an eine Liste in $O(1)$ Zeit erfolgt.

Radix Sort I

Radix Sort: (allgemeines Verfahren, bei dem die Schlüsselwerte als Ziffernfolge zur Basis m aufgefasst werden)

Prinzip:

- Seien die Schlüsselwerte Zeichenfolgen über einem Alphabet von m Buchstaben mit gleicher Länge l, die als Zahlen zur Basis m interpretiert werden.
- Wertebereich der Schlüssel entspricht also $[0, 1, \ldots, m^l - 1]$
- Gegeben sei eine Folge von Buckets $B_0, B_1, \ldots, B_{m-1}$.
- Neue Elemente werden immer „am Ende“ des Buckets eingefügt.

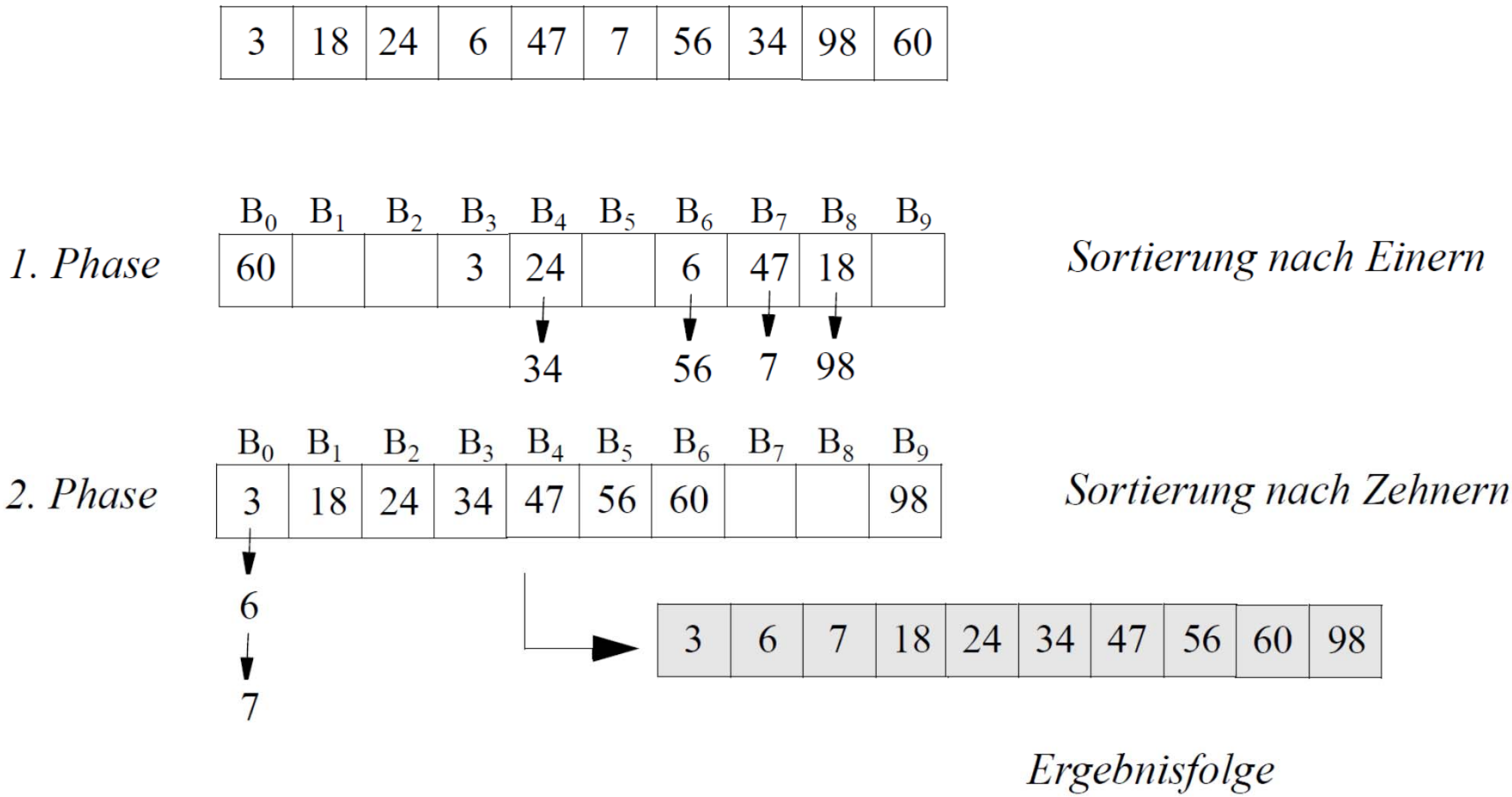
Radix Sort II

Prinzip: (Fortsetzung)

- Es werden l Phasen durchgeführt:
  - Jede Phase abhängig von der jeweils betrachteten Ziffer an Position j der m-adischen Schlüssel, wobei j mit der niedrigstwertigen Position 0 beginnt und
  - Jede Phase durchläuft alle Positionen bis $l - 1$
- In Phase j: Verteilung der Datensätze auf die Buckets, so dass B_i alle Datensätze enthält, deren Schlüssel an j-ter Position das i-te Zeichen besitzt.
  - Dabei bleibt die relative Anordnung der Datensätze innerhalb eines jeden Buckets, die aus den früheren Phasen stammt, unverändert.
- Abschließend wird die Ergebnisfolge mit einem Durchlauf der Buckets in aufsteigender Folge ihrer Nummern $(B_0, B_1, \ldots, B_{m-1})$ generiert

Radix Sort III

Veranschaulichtung: $m = 10$, $l = 2$, Schlüsselintervall $= [0, \ldots, 99]$



Radix Sort IV

Laufzeitanalyse:

- Falls l konstant und $m < n$: Laufzeit von $O(n)$
- Falls alle n Schlüssel verschieden, gilt $l \geq \lceil \log_m n \rceil$. Solange $l = c \cdot \lceil \log_m n \rceil$ mit kleiner Konstanten c: Laufzeit von $O(n \cdot \log n)$
- Diese Laufzeiten gelten auch im schlechtesten Fall.