Sortieralgorithmen

Sortieren

- Sortieren einer (Mehrfach-) Menge von Elementen über einem geordneten Wertebereich (z. B. int, double, String) ist zentrales und intensiv studiertes algorithmisches Problem.
- Mehrfachmenge: mehrfaches Vorkommen der Elemente erlaubt
- Ziel: Berechnung einer geordneten Sequenz aus einer ungeordneten Sequenz dieser Elemente.
- Gegeben:
  - Grundmenge U (U: Universum), totale Ordnung ≤ hierauf
  - Mehrfachmenge M mit Elementen $e_i \in U$
  - Repräsentation von M als Sequenz der Elemente: $l_0 = [e_0, e_1, \dots, e_{n-1}]$ (implementiert z. B. als verkettete Liste oder als (dynamisches) Array).
- Gesucht:
  - $l_s = [e_{j0}, e_{j1}, \dots, e_{jn-1}]$: Anordnung der Elemente aus l_0 gemäß der Ordnung ≤ mit Nachbedingung $e_{j0} \le e_{j1} \le \dots \le e_{jn-1} \wedge \mathrm{perm}(l_0, l_s)$; Prädikat perm ist erfüllt, wenn Sequenz l_s eine Permutation von l_0 ist.

Klassifizierung von Sortierverfahren

- intern / extern:
  - internes Sortierverfahren:
    - Alle Datensätze können gleichzeitig im Hauptspeicher gehalten werden.
    - Direkter Zugriff auf alle Elemente ist möglich und erforderlich.
  - externes Sortierverfahren:
    - Sortieren von Massendaten, die auf externen Speichermedien gehalten werden.
    - Zugriff ist auf einen Ausschnitt der Datenelemente beschränkt.
- methodisch:
  - Sortieren durch Auswählen
  - Sortieren durch Einfügen
  - Sortieren mittels Teile-und-Herrsche/Divide-and-Conquer-Verfahren
  - Sortieren durch Fachverteilen
- nach Effizienz:
  - Einfache Verfahren haben Laufzeit $O(n^2)$.
  - Effiziente Verfahren haben Laufzeit $O(n \log n)$.
  - Es ist beweisbar, das vergleichsbasierte Sortierverfahren nicht besser sein können → untere Schranke des Sortierproblems.
  - Manche Methoden: Unterschiede in Durchschnitts- und Worst-CaseVerhalten
- im Array oder nicht:
  - Array-basierte Verfahren benötigen keinen zusätzlichen Platz für Referenzen.
  - Besonders interessant: Verfahren, die nur ein einziges Array benötigen.
    - Sprechweise: solche Verfahren sortieren in situ (auch: in place).
    - Ergebnis wird erzielt durch Vertauschungen innerhalb dieses Arrays.

Kosten-Nutzen-Analyse für Sortierung

- Wesentliche Ausgangsfrage vor einem Sortiervorhaben: Lohnt sich der Aufwand für die Sortierung überhaupt?
- Strategie: Kosten-Nutzen-Analyse durchführen
  - Sei anz_{sv} die Zahl der Suchvorgänge über die Lebensdauer der Menge.
  - Dann ist es sinnvoll die Menge zu sortieren, falls gilt:
  - $T_s + anz_{sv} \cdot T' < anz_{sv} \cdot T$
    - mit T_s: Aufwand für das Sortieren der Menge
    - T': Aufwand für das Suchen in sortierter Menge
    - T: Aufwand für das Suchen in unsortierter Menge

Sortierverfahren für Listen bzw. für Arrays

- Sortierverfahren für Listen:
  - Verfahren, die Elemente der Liste entlang ihrer Anordnung aufgreifen möchten.
- Sortierverfahren für Arrays:
  - Verfahren, die ausnutzen, dass ein Zugriff auf ein Element des Arrays in $O(1)$ erfolgen kann.
  - Verfahren zur Sortierung von Listen sind prinzipiell auch anwendbar.
  - Im Allg. verbrauchen diese jedoch mehr Speicherplatz, weil das Ergebnis in eine neue Liste geschrieben wird und erst dann die alten Listen frei gegeben werden.
  - In dieser LE beschriebene Array-Implementierungen arbeiten ohne Kopie auf ein und demselben Array (Sortierung erfolgt in situ)

Überblick über die Sortierverfahren

vergleichsbasierte Sortierverfahren

Bezeichnung	Best Case	Average Case	Worst Case	stabil	in situ
Einfache Sortierverfahren					
Sortieren durch Auswählen (SelectionSort)	$O(n^2)$	$O(n^2)$	$O(n^2)$	(leicht mgl.)	(wenn mit Array)
Sortieren durch Einfügen (InsertionSort)	O(n)	$O(n^2)$	$O(n^2)$	X	(wenn mit Array)
Blasensortierung (BubbleSort)	O(n)	$O(n^2)$	$O(n^2)$	X	X
Verfeinertes Auswählen					
Haldensortieren (HeapSort)	O(n log n)	O(n log n)	**O(n log n)**		X
Teile-&-Herrsche/Divide-&-Conquer-Verf.					
Sortieren durch Verschmelzen (MergeSort)	O(n log n)	O(n log n)	**O(n log n)**	X	(in der Regel nicht)
Sortieren durch Zerlegen (QuickSort)	O(n log n)	O(n log n)	$O(n^2)$		X

nicht-vergleichsbasierte Sortierverfahren

Sortieren durch Fachverteilen	Zeit	stabil	in situ
BucketSort	O(n + m)	X	
RadixSort	O(n · k)	X	

Sortieren durch Auswählen (SelectionSort)

Grundidee (Varianten)

- Lösche nacheinander die Maxima aus einer Liste l und füge sie vorne an eine anfangs leere Ergebnisliste ls an.. → aufsteigende Sortierung (im Folgenden betrachtet)
- Lösche nacheinander die Minima aus einer Liste l und füge sie vorne an eine anfangs leere Ergebnisliste ls an. → absteigende Sortierung
- Lösche nacheinander die Maxima aus einer Liste l und füge sie hinten an eine anfangs leere Ergebnisliste ls an. → absteigende Sortierung
- Lösche nacheinander die Minima aus einer Liste l und füge sie hinten an eine anfangs leere Ergebnisliste ls an. → aufsteigende Sortierung

SelectionSort auf verketteten Listen: Entwurf rekursiver Lösung

- Algorithmenentwurf durch Induktion:
  - Induktionsanfang:
    - Leere Liste ls und einelementige Liste ls sind sortiert.
  - Induktionsschritt:
    - Sei $(n-1)$-elementige Liste ls sortiert.
    - Nächstes Element aus l ist < als alle Elemente in ls und wird als erstes Element in ls eingefügt,
    - Daher ist n-elementige Liste ls sortiert.
  - Induktionsende:
    - Wenn l leer ist, dann befinden sich alle Elemente in ls.
- Formulierung als rekursive Methode der Klasse LinkedList<E>
- Wir wollen nicht mit jedem Rekursionsschritt eine neue (Teil-) Liste erstellen, daher benötigen wir eine äußere Methode zur Einbettung der rekursiven Methode, in der insbesondere die Liste ls vereinbart wird.

SelectionSort auf verketteten Listen: rekursive Implementierung

Äußere Methode selsort

```
public LinkedList<E> selsort() {
 // Kopie der zu sortierenden Liste l0
 LinkedList<E> l = new LinkedList<E>();
 l.addAll(this);
 // Erzeuge leere Ergebnisliste ls
 LinkedList<E> ls = new LinkedList<E>();
 // {P: ∀ e ∈ l, e' ∈ ls: e ≤ e' ∧ l ∪ ls = l0 ∧ ls sortiert}
 // Aufruf der Rekursion
 return selsortRec(l, ls);
 // {Q: ∀ e ∈ l, e' ∈ ls: e ≤ e' ∧ l ∪ ls = l0 ∧ ls sortiert ∧ l leer}
}
```

SelectionSort auf verketteten Listen: rekursive Implementierung

Innere Methode selsortRec

```
LinkedList<E> selsortRec(LinkedList<E> l, LinkedList<E> ls) {
        // {I: ∀ e ∈ l, e' ∈ ls: e ≤ e' ∧ l ∪ ls = l0 ∧ ls sortiert}
        if (!l.isEmpty()) {
                E max = getMaximum(l);
                l.remove(max);
                ls.addFirst(max);
                return selsortRec(l, ls);
        } else return ls;
}
```

SelectionSort auf verketteten Listen: iterative Implementierung

Nach Entrekursivierung und Einbettung in die Außenmethode:

```
public LinkedList<E> selsort() { //O(n²)
        // Kopie der zu sortierenden Liste l0
        LinkedList<E> l = new LinkedList<E>();
        l.addAll(this);
        // Erzeuge leere Ergebnisliste ls
        LinkedList<E> ls = new LinkedList<E>();
        // {P: ∀ e ∈ l, e' ∈ ls: e ≤ e' ∧ l ∪ ls = l0
        // ∧ ls sortiert}
        while (!l.isEmpty()) {
                E max = getMaximum(l); // Maximum auswaehlen
                l.remove(max); // Maximum entfernen
                ls.addFirst(max); // Maximum an Ergebnisliste
                // vorne anfuegen
                // {I: ∀ e ∈ l, e' ∈ ls: e ≤ e' ∧ l ∪ ls = l0
                ∧ ls sortiert}
        }
        // {Q: ∀ e ∈ l, e' ∈ ls: e ≤ e' ∧ l ∪ ls = l0
        // ∧ ls sortiert
        // ∧ l leer}
        return ls;
}
```

Korrektheit und Stabilität von selsort

- selsort ist korrekt
  - $Q = I \wedge \neg b = I \wedge l$ leer
  - I gilt für die leere Liste ls
  - Bei Verlassen der Schleife gilt I mit leerer Liste l: $ls = l0 \wedge ls$ sortiert (”=” steht für Gleichheit von Mehrfachmengen)
  - I ist Schleifeninvariante: Wiederherstellen von I durch remove(…) und add(…)
  - Terminierung, da l streng monoton verkürzt wird.
- Damit selsort stabil ist, muss getMaximum(l) das in der Reihenfolge von l letzte Maximum finden und remove(…) auch dieses Element löschen.

Aufwandsabschätzung zu selsort

- Idee für Verbesserung (führt zur Haldensortierung (HeapSort)):
  - Die kritischen Operationen sind das Identifizieren und Entfernen des Maximums.
  - In einer Halde war es möglich, das Maximum in $O(1)$ zu entnehmen und in $O(\log n)$ die max-Halden-Eigenschaft wieder herzustellen

In situ SelectionSort auf Arrays: iterative Implementierung

```
public void selsort() {  //O(n2)
        E max;
        int maxIndex;
        for (int i = a.length - 1; i > 0; i--) {
              max = a[i];
              maxIndex = i;
              for (int j = 0; j < i; j++) {
                    if (a[j].compareTo(max) >= 0) {
                          max = a[j];
                          maxIndex = j;
                    }
              }
              swap(a, i, maxIndex);
        }
}
protected void swap(E[] a, int n, int m) {
        E tmp = a[n];
        a[n] = a[m];
        a[m] = tmp;
}
```

Sortieren durch Einfügen (InsertionSort)

- Grundidee
- Typisches Verfahren beim Sortieren von Spielkarten:
  - Starte mit der ersten Karte vom Stapel eine neue Kartensequenz auf der Hand.
  - Nimm jeweils die nächste Karte vom Kartenstapel und füge diese an der richtigen Stelle in die Kartensequenz auf der Hand ein.
- Angenommen wir können n – 1 Werte sortieren. Dann können wir den n-ten Wert einsortieren, indem wir seinen Platz in der sortierten Liste finden und die restlichen Elemente nach hinten verschieben.
- Nimm das nächste Element aus der Liste l und füge es an der richtigen Stelle in die (anfangs leere) Liste ls ein.

InsertionSort auf Listen: iterative Implementierung

```
public LinkedList<E> insort() {
        LinkedList<E> l = new LinkedList<E>();
        l.addAll(this);
        LinkedList<E> ls = new LinkedList<E>();
        // {P: l ∪ ls = l0 ∧ ls sortiert ∧ ls leer}
        while (!l.isEmpty()) {
              E e = l.get(0);
              l.remove(e);
              // suche in ls e', e" aufeinanderfolgend
              // mit e' ≤ e < e";
              // fuege in ls e zwischen e', e" ein
        }
        // {Q: l ∪ ls = l0 ∧ ls sortiert ∧ l leer}
        return ls;
}
```

Bei alternativer Implementierung auf Basis von Arrays entsteht analog zu SelectionSort wieder ein in situ-Verfahren

InsertionSort auf Listen: Aufwandsabschätzung

- Sequentielle Suche:
- mittlerer Aufwand $O(k/2)$
- schlimmster Fall $O(k)$ wenn k Länge von ls ist
- Günstigster Fall:
  - l ist bereits entgegengesetzt geordnet (jew. Entnahme des Maximums).
  - Innere Schleife verursacht dann konstanten Aufwand c', weil immer vorne angefügt werden kann.
  - Gesamtaufwand ist dann im besten Fall in $O(n)$.

InsertionSort auf Arrays: iterative Implementierung

```
public void insort() {
        E e;
        int j;
        for (int i = 1; i < a.length; i++) {
                e = a[i]; // entnommenes Element merken
                j = i - 1;
                while (j >= 0 && e.compareTo(a[j]) < 0) {
                        a[j + 1] = a[j];
                        j--;
                }
                a[j + 1] = e; // entnommenes Element
                              // einsetzen
        }
}
```

- i: vordere Grenze des Arraybereichs, aus dem erstes Element entnommen wird.
- j: Durchlaufen des vorderen Arraybereiches, um Einfügestelle zu finden; bis dahin: Elemente von oben kommend um eine Position nach rechts verschieben.

Blasensortierung (BubbleSort)

- Grundidee
  - Das Array wird immer wieder durchlaufen und dabei werden benachbarte Elemente in die richtige Reihenfolge gebracht.
  - Größere Elemente überholen so die kleineren und drängen an das Ende der Folge („wie aufsteigende (Luft-) Blasen“)

BubbleSort auf Arrays

```
static <E extends Comparable<E>> void bubblesort(E[] a) { // O(n^2)
        // Variable zum merken, ob beim aktuellen Durchlauf der Sequenz eine
        // Vertauschung stattfand
        boolean swapped;
        // oberster Index des Arrays a, bis zu dem noch korrekte Ordnung geprueft
        // werden muss
        int upper = a.length - 1;
        do {
                swapped = false;
                for (int i = 0; i < upper; i++) {
                        if (a[i].compareTo(a[i + 1]) > 0) {
                                // tausche im Array a die
                                // Eintraege an den Indizes
                                // i und i + 1
                                swap(a, i, i + 1);
                                // merke: es wurde getauscht
                                swapped = true;
                        }
                }
                upper--;
        } while (swapped);
}
```

BubbleSort auf Arrays: Aufwandsabschätzung

- Im ersten Durchlauf werden n Elementpaare verglichen.
- Im zweiten Durchlauf werden n – 1 Elementpaare verglichen (upper ist um 1 reduziert).
- Gesamtaufwand: $O(n^2)$
- Durch die Optimierung mit swapped sinkt der Aufwand im bestenFall auf $O(n)$.

Haldensortierung (HeapSort)

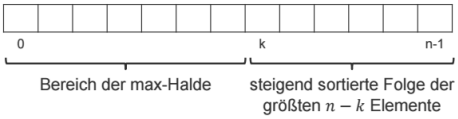
- Sortieren durch Auswählen bisher: SelectionSort
  - Je Schritt das Maximum (Minimum) der Werte auswählen und aus Ursprungsliste entnehmen: Aufwand $O(n)$.
  - Gesamtaufwand: $O(n^2)$
- Strategie:
  - Maximumsauswahl (Minimumsauswahl) durch geeignete Datenstruktur verbessern: Halde (Heap).
  - Auswahl des Maximums (Minimums) und Entfernung aus einer Halde hat (nur) den Aufwand $O(\log n)$.
- Grundidee der Haldensortierung (HeapSort):
  - Die n zu sortierenden Elemente werden in eine max-Halde (minHalde) eingefügt: Aufwand $O(n \log n)$. Optimiert sogar nur $O(n)$.
  - Es wird n-mal das Maximum (Minimum) aus der Halde entnommen: Aufwand $O(n \log n)$.
  - Gesamtaufwand damit $O(n \log n)$

Haldensortierung (HeapSort)

- Ziel: Aufbau der Halde optimieren und in situ sortieren.
- Teil-Array $a[i..kk]$, $0 \le i \le k < n$, heißt Teil-max-Halde (Teil-maxHeap), gdw.: $\forall j \in [i, \dots, k]\ a[j] \ge a[2j+1]$ falls $2j+1 \le k$
- und $a[j] \ge a[2j+2]$ falls $2j+2 \le k$
- Wenn $a[0..n-1]$ eine Teil-max-Halde ist, dann ist $a[0..n-1]$ auch eine max-Halde, d. h. die Array-Einbettung eines partiell geordneten Baums.
- Ablauf Haldensortierung (HeapSort):
  - Aufbau der max-Halde: Die hintere Hälfte des Arrays $a[n/2..n-1]$ ist bereits eine Teil-max-Halde (da diese Knoten keine Kinder mehr haben und somit die Bedingung trivialerweise erfüllt ist).

```
for i := n/2 - 1 downto 0 do {
        erweitere die Teil-max-Halde a[i+1..n] zu einer
        Teil-max-Halde a[i..n] durch „Einsinken lassen“ von a[i]
}
```

  - Baue die sortierte Folge rückwärts am hinteren Ende des Arrays auf



    - In jedem Schritt wird $a[0]$ mit $a[k-1]$ vertauscht und damit der Haldenbereich auf $a[0..k-2]$ reduziert.
    - $a[1..k-2]$ ist weiterhin eine Teil-max-Halde.
    - Durch Einsinken von $a[0]$ wird $a[0..k-2]$ wieder zu einer (Teil-max-) Halde

reheap: Element in max-Halde „einsinken“ lassen

Ziel: Wiederherstellen der max-Halden-Eigenschaft, indem a[i] in die Halde „einsinkt“.

```
protected void reheap(E[] a, int i, int k) {
        int leftKidIdx = 2 * i + 1;
        int rightKidIdx = leftKidIdx + 1;
        int kidIdx;
        if (leftKidIdx <= k && rightKidIdx > k) {
                // nur ein (= linkes) Kind
                if (a[leftKidIdx].compareTo(a[i]) > 0) {
                        swap(a, leftKidIdx, i);
                }
        } else {
                if (rightKidIdx <= k) {
                        // in kidIdx groesseren der beiden Kinder erfassen
                        kidIdx = a[leftKidIdx].compareTo(a[rightKidIdx]) > 0
                                        ? leftKidIdx : rightKidIdx;
                        if (a[kidIdx].compareTo(a[i]) > 0) { // gfs. tauschen
                                swap(a, i, kidIdx);
                                reheap(a, kidIdx, k);
                        }
                }
        }
}
```

Aufwand für Aufbau der Halde

- Grobe Abschätzung:
  - Jeder Aufruf von reheap hat Aufwand $O(\log n)$.
  - $O(n)$ Aufrufe von reheap haben den Gesamtaufwand $O(n \log n)$.
- Genauere Abschätzung:
  - Laufzeit für reheap ist abhängig von der Höhe h der Halde.
  - Diese ist aber meist viel kleiner als $\log n$.
  - Z.B. haben die ersten $(n/2)$ eingefügten Werte die Höhe 0.
  - Allgemein: es gibt $(n)/(2^{h+1})$ Knoten der Höhe h. Für jeden von diesen hat reheap den Aufwand $O(h)$, fast immer also weniger als $O(n \log n)$.
  - Es lässt sich damit für den Gesamtaufwand zeigen: $\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(n)$

Implementierung von HeapSort

```
public void heapsort(E[] a) {
        int n = a.length;
        // Phase 1: Halde aufbauen
        for (int i = n / 2; i >= 0; i--) {
              reheap(a, i, n - 1);
        }

        // Phase 2: jeweils Maximum entnehmen und sortierte Liste am Ende aufbauen
        for (int i = n - 1; i > 0; i--) {
               // Maximum ans Ende des Haldenbereichs tauschen
               swap(a, 0, i);
               // nach vorne getauschtes Element einsinken lassen und
               // max-Halden-Eigenschaft wieder herstellen
               reheap(a, 0, i - 1);
        }
}
```

Sortieren unter Verwendung der Teile-und-Herrsche-Strategie

- Zur Erinnerung: Teile-und-Herrsche/Divide-and-Conquer
- Wenn die Objektmenge klein genug ist, dann löse das Problem direkt sonst
  - Divide: Zerlege die Menge in mehrere Teilmengen (möglichst ähnlicher/gleicher Größe).
  - Conquer: Löse das Problem rekursiv für jede Teilmenge.
  - Merge: Berechne aus den für die Teilmengen erhaltenen Lösungen eine Lösung des Gesamtproblems.
- im Folgenden betrachtete Strategien:
  - Sortieren durch Verschmelzen (MergeSort)
  - Sortieren durch Zerlegen (QuickSort)

Sortieren durch Verschmelzen (MergeSort)

- Grundidee
- gegeben sei eine Sequenz l = [e0, e1, … , en−1].
- Algorithmus MergeSort(l):
  - Wenn Länge(l) <= 1, dann gib l zurück, sonst
  - Divide:
    - l1 := [e0,…,e⌊n/2⌋-1];
    - l2 := [e⌊n/2⌋ ,…,en-1];
  - Conquer:
    - l1' := MergeSort(l1);
    - l2' := MergeSort(l2);
  - Merge:
  - gib merge(l1',l2') zurück
- Sortierung erfolgt beim Verschmelzen (merge)
- Rekursionstiefe ist in O(log2 n).
- Verfahren arbeitet (auch) auf Listen.

Aufwandsabschätzung zu MergeSort

- Gegeben: Liste der Länge n
- Rekursive Aufrufe lassen sich als Baum darstellen, dessen Knoten Aufrufe von mergesortRec repräsentieren.
- Wir benennen die Ebenen von oben kommend mit k = 0, 1, …
- Gesamtaufwand ist Summe der Aufrufe aller Knoten im Rekursionsbaum.
- Aufwand eines Knotens auf Ebene k ist jeweils in $O(n/2^k)$ für das Zerlegen und Verschmelzen der sortierten Hälften (Länge ca. $n/2^k$).
- Anzahl der Knoten/Listen auf Ebene k ist 2^k, also in $O(2^k)$.
- Summe der Aufwände auf Ebene k ist demnach $O(2^k)\cdot O(n/2^k) = O(n)$, unabhängig von k.
- Anzahl der Ebenen ist in $O(\log_2 n)$.
- Gesamtaufwand im schlechtesten Fall: $O(n\cdot\log_2 n)$

MergeSort auf Listen: rekursive Implementierung

äußere Methode mergesort

```
static <E extends Comparable<? super E>> LinkedList<E>
                        mergesort(LinkedList<E> l) {
        // Basisfall
        if (l.size() < 2) return l;
        // zerlege l in 2 ungefaehr gleich lange Teillisten
        LinkedList<E> left = new LinkedList<>(l.subList(0, l.size() / 2));
        LinkedList<E> right = new LinkedList<>(l.subList(l.size() / 2, l.size()));
        // sortiere beide Teillisten und verschmelze sie
        return merge(mergesort(left), mergesort(right));
}
```

Verschmelzen (sog. Reißverschlussverfahren)

```
static <E extends Comparable<? super E>> LinkedList<E>
              merge(LinkedList<E> l, LinkedList<E> r) {
        LinkedList<E> ret = new LinkedList<>();
        while (l.size() > 0 && r.size() > 0) { // Reissverschluss
               E left = l.getFirst(), right = r.getFirst();
               if (left.compareTo(right) <= 0) {
                      ret.addLast(left);
                      l.removeFirst();
               } else {
                      ret.addLast(right);
                      r.removeFirst();
               }
        }
        while (l.size() > 0) { // uebrige Werte von links
               ret.addLast(l.getFirst());
               l.removeFirst();
        }
        while (r.size() > 0) { // uebrige Werte von rechts
               ret.addLast(r.getFirst());
               r.removeFirst();
        }
        return ret;
}
```

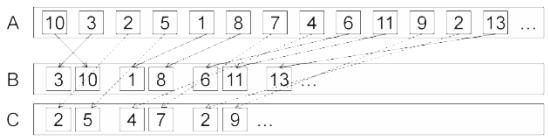
Wichtige Eigenschaften von MergeSort

- mergesort(…) ist stabil.
- Kein wahlfreier Zugriff notwendig.
- Schnellstes Verfahren auf verketteten Listen wegen der geringsten Zahl an Vergleichen.
- Sequenzieller Zugriff auf Teillisten, daher verbreitetes Sortierverfahren für große Listen, die nicht mehr in den Hauptspeicher passen (externes Sortierverfahren).
- Durch geschickte Listenimplementierung auch in-situ (auch ohne Kopieroperationen der einzelnen Elemente) möglich.

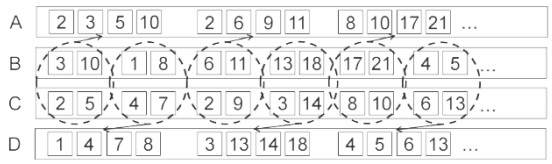
MergeSort als externes Sortierverfahren

Bottom-Up statt Top-Down.

- 4 „Bänder“ A-D (= sequentiell zugreifbare Dateien, Magnetbänder), Die Eingabe befindet sich auf Band A.
- 1. Schritt: Zweier-Tupel von Band A verschmelzen und abwechselnd auf Band B und Band C schreiben.
- Auf Band C und D liegen nun jeweils sortierte Zweier-Tup



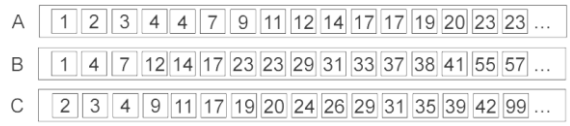
- 2. Schritt: Zweier-Tupel von Band B und C verschmelzen, die entstehenden Vierer-Tupel abwechselnd auf Band A und D speichern.



- 3. Schritt: Vierer-Tupel von Band A/D verschmelzen, Ergebnis abwechselnd auf Band B/C speichern.
- k. Schritt: Tupel der Länge 2^{k-1} verschmelzen, Ergebnis abwechselnd schreiben (dabei alterniert A/D und B/C als Eingabe).

MergeSort als externes Sortierverfahren

- Letzter Schritt: jeweils nur noch je ein Tupel mit ca. n/2 Elementen auf zwei Bändern (o.b.d.A. Band B/C, Ausgabe erfolgt auf A).

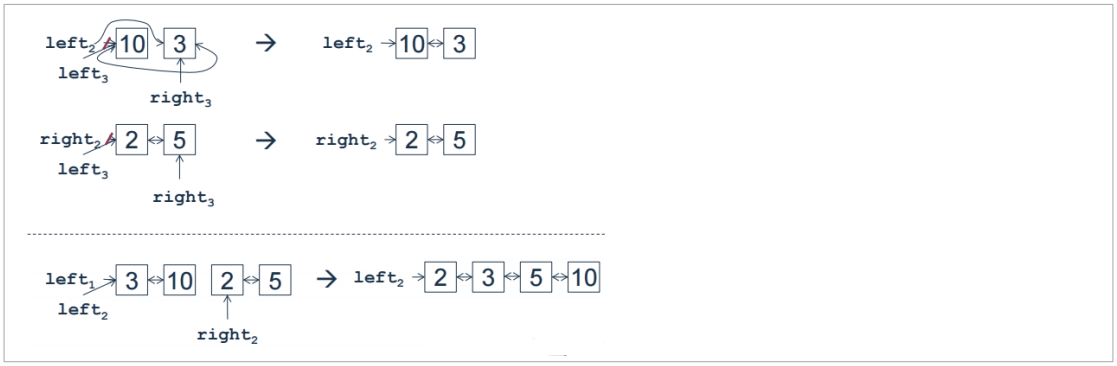


- Falls Band A vor dem letzten Schritt belegt ist, muss noch umkopiert werden.
- Insgesamt sind $\log_2 n$ Schritte notwendig. In jedem Schritt werden alle Elemente einmal bearbeitet Gesamtaufwand: $O(n \cdot \log_2 n)$.

MergeSort in-situ

- Mittels geschickter Listenoperationen ist es möglich, MergeSort in-situ zu implementieren.
- Zuerst rekursiv aufteilen (nur Referenzen "umbiegen" notwendig).

- Beim Verschmelzen wieder "Umbiegen" von Referenzen.



Sortieren durch Zerlegen (QuickSort)

Grundidee

- Gegeben sei eine Sequenz l = [$e0$, $e2$, … , en−1].
- Algorithmus QuickSort(l):
- Wenn Länge(l) <= 1, dann gib l zurück, sonst
  - Divide:
    - Wähle irgendeinen Wert p = ej aus l aus. Berechne eine Teilfolge l1 aus l mit den Elementen, deren Wert ≤ p ist, und eine Teilfolge l2 mit Elementen > p.
  - Conquer:
    - l1' := QuickSort(l1);
    - l2' := QuickSort(l2);
  - Merge: gib concat(l1',l2') zurück
- Sortierung erfolgt durch die Art der Zerlegung (divide).
- Verfahren arbeitet (vor allem) auf Arrays in situ.

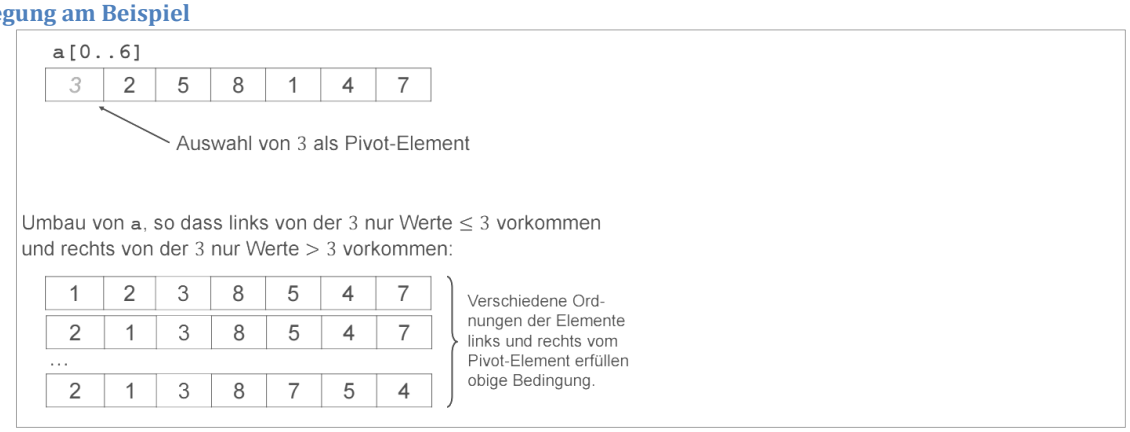
Zerlegung

- Gegeben:
  - Array a von Elementen mit Totalordnung ≤ und
  - Ausschnitt $a[m. . n]$ zwischen Indizes m, nn(einschließlich).
  - In diesem Ausschnitt vorkommendes beliebiges Element p, sogenanntes Pivot-Element (pivot (franz.): Dreh-/Angelpunkt).

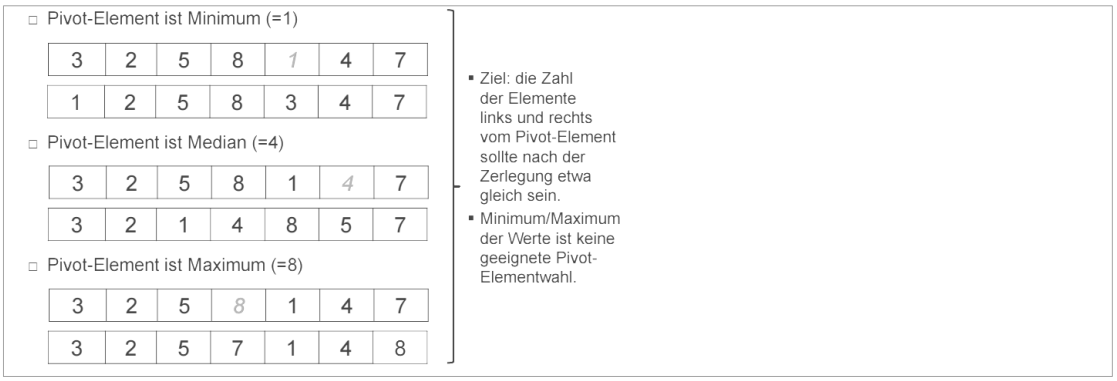
Zerlegung

- Aufgabe:
  - Umbau des Ausschnitts durch Platztauschen, so dass für einen Index r (r ist Index des Pivot-Elements; p: = $a[r]$ ist Pivot) mit $m \le r \le n$ gilt:
    - $a[k] \le p$ für $m \le k \le r$ und
    - $a[k] > p$ für $r < k \le n$.
  - Der Index r soll zurückgegeben werden, während der Umbau von $a[m..n]$ als Seiteneffekt erfolgt.

Zerlegung am Beispiel



Auswirkungen der Wahl des Pivot-Elements



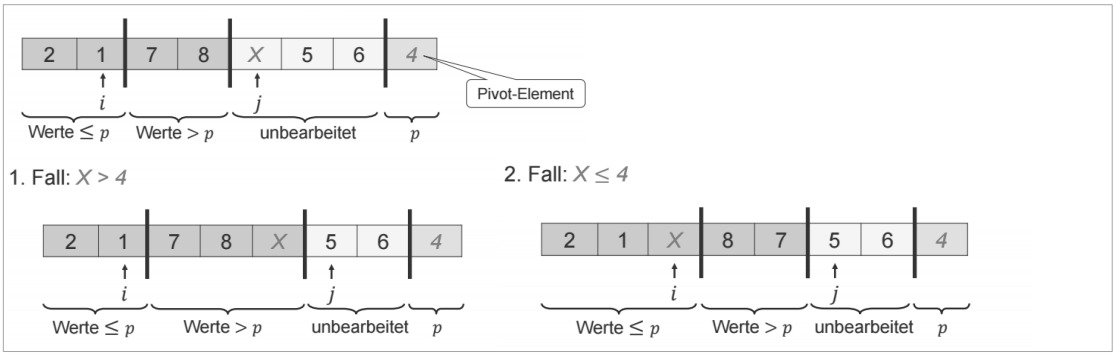
Spezifikation der Methode partition Vor-/Nachbedingung

- Methodensignatur:
  - private static <E extends Comparable<E>> int
  - partition(E[] a, int m, int n)
- Spezifikation der Methode:
  - Vorbedingung
    - P: $(0 \le m \le n < a.\,\mathrm{length})$
  - Nachbedingung
    - r: = Rückgabewert der Methode partition
    - a' : = „Belegung“ von a beim Betreten der Methode
    - Q: $(m \le r \le n) \wedge \mathrm{perm}(a[m..n], a'[m..n]) \wedge \forall k: ((m \le k \le r \Rightarrow a[k] \le a[r] \wedge (r < k \le n \Rightarrow a[r] < a[k]))$
  - Prädikat $\mathrm{perm}(x, y)$ ist wahr, falls die Sequenzen x und y Permutationen von einander sind.

Algorithmische Idee zum Finden der „Nahtstelle“

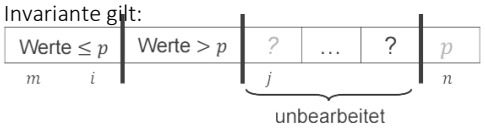
- Wähle das letzte Element des Arrays als Pivot-Element p.
- Zerlege den Array-Bereich logisch in 4 Bereiche: Sammelbereich für Werte ≤ p (anfangs leer), Sammelbereich für Werte > p (anfangs leer), Bereich noch zu bearbeitender Werte, Pivot-Element selbst.
- Grenzen zwischen den ersten drei Bereichen verschieben sich während des Verfahrens; Zeiger i und j zum Markieren der Grenzen.

Algorithmische Idee zum Finden der „Nahtstelle“



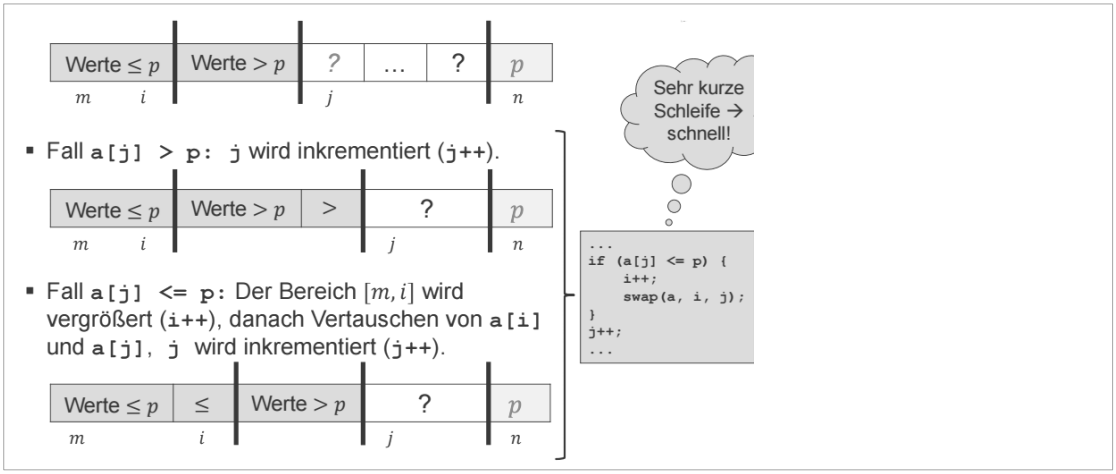
Spezifikation der Methode partition : Invariante

- Durch Wahl des Pivot-Elements am rechten Rand entstehen vier Bereiche $[m, i]$, $]i,j[$, $[j, n[$ und $[n, n]$, für deren Werte die folgende Invariante gilt:
  - grafisch:

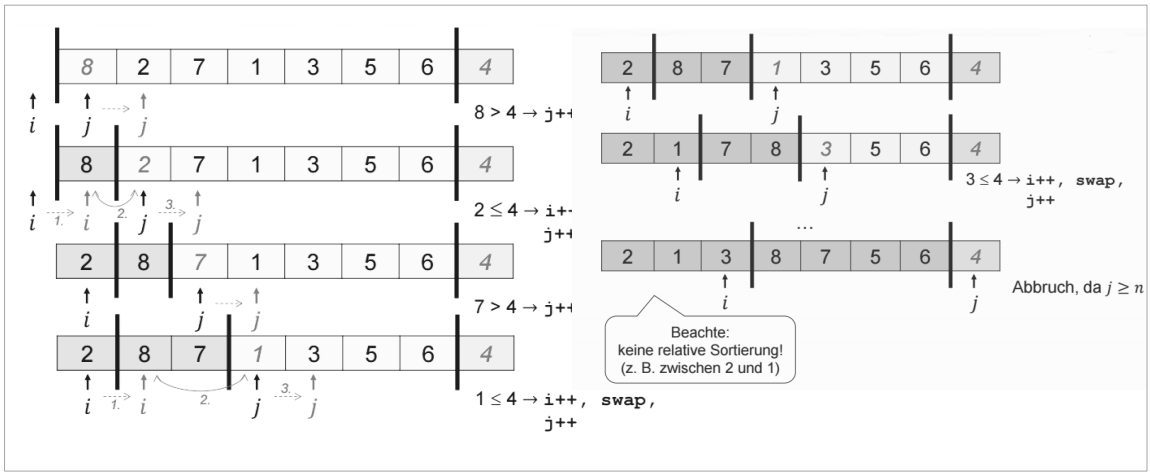


  - als Formel: I: $m - 1 \le i < j \le n \wedge \text{perm}(a[m..n]) \wedge \forall k$: $((m \le k \le i \Rightarrow a[k] \le p) \wedge (i < k < j \Rightarrow p < a[k]))$
  - Nachbedingung der Schleife: S: = $I \wedge \neg b$ S: $m - 1 \le i < j \le n \wedge \text{perm}(a[m..n], a'[m..n]) \wedge \forall k$: $((m \le k \le i \Rightarrow a[k] \le p) \wedge (i < k < j \Rightarrow p < a[k])) \wedge (j \ge n)$

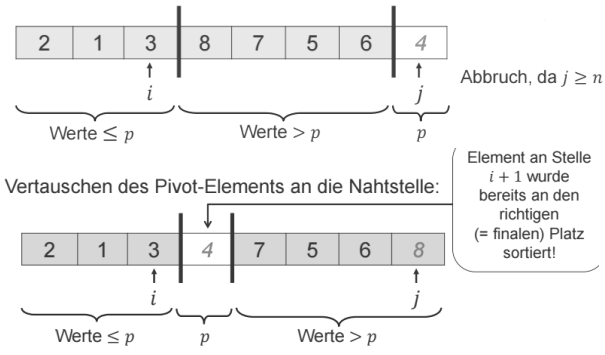
Verarbeitung eines Elements



Zerlegung am Beispiel



Zerlegung am Beispiel



Implementierung der Methode partition

```
protected int partition(E[] a, int m, int n) {
        // {P: (0 ≤ m ≤ n < a.length)}
        // a' := java.util.Arrays.copyOf(a, a.length); <- „virtual copy"
        E p = a[n];
        int i = m - 1;
        int j = m;
        while (j < n) {
                // {I: m - 1 ≤ i < j ≤ n ∧ perm(a[m..n],a'[m..n]) ∧
                // ∀ k: ((m ≤ k ≤ i ⇒ a[k] ≤ p) ∧ (i < k < j ⇒ p < a[k]))}
                if (a[j].compareTo(p) <= 0) {
                        i++;
                        swap(a, i, j);
                }
                j++;
        }
        // {S: I ∧ ¬b}
        int r = i + 1;
        swap(a, r, n);
        // {Q: (m ≤ r ≤ n) ∧ perm(a[m..n],a'[m..n]) ∧
        // ∀ k:((m ≤ k ≤ r ⇒ a[k] ≤ a[r]) ∧ (r < k ≤ n ⇒ a[r] < a[k]))}
        return r;
}
```

Korrektheit und Terminierung

- Korrektheit:
  - Die Bedingung {I: m - 1 ≤ i < j ≤ n ∧ perm(a[m..n],a'[m..n]) ∧ ∀ k: ((m ≤ k ≤ i ⇒ a[k] ≤ p) ∧ (i < k < j ⇒ p < a[k]))} ist Invariante. Beweis ist nicht schwierig.
- Terminierung:
  - Solange $j < n$ ist, wird der Abstand zwischen j und n pro Schleifendurchlauf um 1 verringert (Schleifenvariante z. B. V: = $n - j$).
- Nachbedingung der Methode partition:
  - Zu zeigen: $I \wedge \neg b \Rightarrow wp$("r = i + 1; swap(a, r, n);" ,Q) mit: Q: $(m \leq r \leq n) \wedge \mathrm{perm}(a[\mathrm{m}...\mathrm{n}]a'[m..n]) \wedge \forall k$: $((m \leq k \leq r \Rightarrow a[\mathrm{k}] \leq a[\mathrm{r}]) \wedge (r < k \leq n \Rightarrow a[\mathrm{r}] < a[\mathrm{k}]))$ Beweis ist nicht schwierig

QuickSort auf Arrays: rekursive Implementierung

äußere Methode quicksort

```
public void quicksort(E[] a) {
        quicksortRec(a, 0, a.length - 1);
}
```

innere Methode quicksortRec

```
protected void quicksortRec(E[] a, int m, int n) {
        if (n > m) {
                int r = partition(a, m, n);
                quicksortRec(a, m, r - 1);
                quicksortRec(a, r + 1, n);
        }
}
```

Nachbedingung von partition(…) Q

```
protected void quicksortRec(E[] a, int m, int n) {
        if (n > m) {
                int r = partition(a, m, n);
                // {Qpartition: (m ≤ r ≤ n) ∧ perm(a[m..n], a'[m..n]) ∧
                // ∀k:((m ≤ k ≤ r ⇒ a[k] ≤ a[r]) ∧
                // r < k ≤ n ⇒ a[r] < a[k]))}
                quicksortRec(a, m, r - 1);
                quicksortRec(a, r + 1, n);
        }
        // {Q: perm(a[m..n], a'[m..n]) ∧ a[m..n] sortiert}
}
```

- Für partition(…) lautete die Nachbedingung: {Qpartition: (m ≤ r ≤ n) ∧ perm(a[m..n], a'[m..n]) ∧ ∀k:((m ≤ k ≤ r ⇒ a[k] ≤ a[r]) ∧ r < k ≤ n ⇒ a[r] < a[k]))}
- Diese gilt nach dem Aufruf von partition(…).

Nachweis der Gültigkeit der Nachbedingung Q

- Induktionsanfang:
  - Rekursion so lange, bis partition(…) auf einelementiger Liste.
  - Diese ist sortiert.
- Induktionsschritt:
  - Sind beide Teile sortiert (Q), so ist wegen Qpartition auch die Verbindung der Teile sortiert

Aufwandsabschätzung zu QuickSort

- Bester Fall:
  - Wenn das Pivot-Element den zu sortierenden Ausschnitt genau halbiert (also die Hälfte der Werte kleiner ist als das Pivot-Element und die andere Hälfte der Werte größer ist als das Pivot-Element), dann hat QuickSort den Aufwand $O(n \log 2n)$.
- Schlechtester Fall:
  - Im schlechtesten Fall wird das Pivot-Element stets so gewählt, dass es das größte oder das kleinste Element der Liste ist.
  - Dies ist etwa der Fall, wenn als Pivot-Element stets das Element am Ende der Liste gewählt wird und die zu sortierende Liste bereits sortiert vorliegt.
  - Die zu untersuchende Liste wird dann in jedem Rekursionsschritt nur um eins kleiner und die Laufzeit wird beschrieben durch $n + (n-1) + (n-2) + \cdots + 1 \in O(n^2)$

Verbesserung der Wahl des Pivot-Elements

- Für jedes Verfahren, das nach einer feststehenden Regel das Pivot-Element aussucht (z. B. in der Mitte des Ausschnitts, am Ende des Ausschnitts, …), kann eine Eingabe gefunden werden, die zu quadratischem Aufwand führt.
- „Normalerweise“ tritt der $O(n^2)$-Fall aber nicht auf; dann hat QuickSort im Mittel den Aufwand $O(n \log 2n)$.
- Pivotwahl, um Wahrscheinlichkeit des $O(n^2)$-Falls zu reduzieren:
  - 3-Median-Strategie: Wähle drei Elemente vom linken und rechten Rand und aus der Mitte des zu sortierenden Ausschnitts. Wähle als Pivot-Element das mittlere dieser drei Elemente.
  - Zufallsstrategie: Wähle mit Zufallszahlengenerator (gleichverteilt) die Stelle des Pivot-Elements im zu sortierenden Ausschnitt aus.

Vorteile von QuickSort trotz Worst-Case-Verhaltens

- MergeSort und HeapSort scheinen überlegen, weil sie immer O(n log2n) Aufwand haben.
- Aber (1): QuickSort ist in der Praxis bei nicht pathologischer Pivotwahl schneller als MergeSort und HeapSort, weil die wesentliche Schleife weniger Instruktionen hat.
- Aber (2):
  - Bei wenigen Elementen (10 - 20) ist Sortieren durch Einfügen schneller als QuickSort.
  - Ursache: Im O-Kalkül sind die konstanten Faktoren von QuickSort größer, so dass bei kleinem n der quadratische Algorithmus trotzdem schneller ist als der $n \log 2n$-Algorithmus.
  - Daher die Rekursion nicht bis zum leeren oder einelementigen Ausschnitt laufen lassen, sondern rechtzeitig umsteigen.Umstieg ist einfacher zu machen als bei Merge- und HeapSort.

Sortieren durch Fachverteilen (BucketSort)

- Wenn die zu sortierenden Werte bestimmten Einschränkungen unterliegen und auch andere Operationen als Vergleiche angewendet werden können, so ist es möglich schneller zu sortieren, z. B. in $O(n)$.

Sortieren durch Fachverteilen (BucketSort)

- Beispiel:
  - Sei l= [e0, e1, … , en−1] wobei ei ganze Zahlen zwischen 0 und n – 1 sind und l keine Duplikate enthält.
  - Benutze zwei n-elementige Arrays: eines für l und eines für die sortierte Folge ls
  - Erzeugung der sortierten Folge in ls in $O(n)$
- Seien die zu sortierende Werte nun ganze Zahlen zwischen 0 und m − 1 und seien Duplikate erlaubt.
- Nutzung einer Menge von Behältern B0, … , Bm−1:
  - Jeder Behälter lässt sich als Liste von Elementen implementieren.
  - Die Behälter werden über Array verwaltet.
  - Datenstruktur entspricht der des offenen Hashing

```
Algorithmus BucketSort(l):
        für i := 0 bis m – 1 führe aus:
                Bi := ∅;
        für i := 0 bis n – 1 führe aus:
                füge ei in Bei ein;
        für i := 0 bis m – 1 führe aus:
                schreibe die Elemente von Bi in die
                Ergebnisfolge;
```

- Beispiel: Poststellen
  - Pro Person wird ein Postfach angelegt, insgesamt m Fächer.
  - Jedes Element der eingehenden Post (Menge mit n Elementen) wird in das Fach des Adressaten gelegt.
  - Jeder der n Briefe ist anzufassen, seine Fachnummer ist zu bestimmen, dann ist er in dieses Fach zu legen.
  - Am Schluss müssen die m Fächer geleert werden.
- Eigenschaften
  - Einfügen eines Elements in einen Behälter ist in $O(1)$ möglich.
  - Die Laufzeit von BucketSort ist in $O(n + m)$:
  - Wenn $m = O(n)$, dann sogar in $O(n)$
  - Bei $m \gg n$ aber nicht praktikabel (es gibt dann viel mehr mögliche Werte, als in der zu sortierenden Menge tatsächlich vorkommen).
  - Verfahren ist stabil, wenn zur Implementierung der Behälter Schlangen verwendet werden.

Verallgemeinertes Sortieren durch Fachverteilen (RadixSort)

- Statt für jeden in Frage kommenden Wert ein Fach anzulegen, wird der Wert in (kürzere) Segmente aufgeteilt.
- Für den kleinen Wertebereich des Segments gibt es Fächer, z. B.
  - einzelne Ziffern der Postleitzahl (d = 10 Fächer)
  - Buchstaben eines Worts fester Länge (d = 26 Fächer)
- Die Segmente befinden sich in einer definierten Reihenfolge, z. B. Stellen der Postleitzahl: 9-1-0-5-8
- Ablauf
  - Die Elemente der Menge werden gemäß des betrachteten Segments per BucketSort sortiert.
  - Für die Sortierung nach den restlichen Segmenten wird BucketSort rekursiv angewendet.
  - Dabei: Segmentbetrachtung von rechts nach links
- Wichtig: Realisierung der Behälter als Schlangen.

Korrektheit von RadixSort

- Induktionsanfang: ein Segment/vorderstes Segment: Fächersortierung mit min. d Fächern erzeugt korrekte Sortierung.
- Induktionshypothese: Werte mit weniger als k Segmenten, die jeweils maximal d mögliche Werte einnehmen können, können sortiert werden.
- Induktionsschluss:
  - Fall: Zwei Elemente unterscheiden sich im k-ten Segment. Dann wird der k-te Sortierschritt nach Induktionsvoraussetzung zur korrekten Sortierung der Elemente führen.
  - Fall: Zwei Elemente sind im k-ten Segment gleich.
    - Dann sind sie nach Induktionsvoraussetzung in den k − 1 Segmenten rechts davon korrekt sortiert.
    - Ein stabiles Sortierverfahren behält die relative Ordnung der beiden Elemente bei.
    - Damit sind sie auch noch korrekt sortiert, wenn nach dem k-ten Segment sortiert worden ist

Aufwandsabschätzung zu RadixSort

- Pro Segment der Radix-Sortierung werden alle n Werte untersucht.
- Bei k Segmenten ergibt sich ein Gesamtaufwand von $O(n \cdot k)$.

Variante: Radix-Exchange-Sortierung

- Radix-Exchange-Sortierung am Beispiel:
  - Sortiere Post nach der ersten Stelle der Postleitzahl in 10 Fächer.
  - Sortiere jedes Fach (rekursiv) nach der zweiten Stelle der Postleitzahl in wiederum 10 Fächer.
  - Sortiere jedes dieser Fächer nach der dritten Stelle der Postleitzahl in wiederum 10 Fächer.
  - Man benötigt keine 100 000 sondern nur rund 50 Fächer zum Sortieren!