Algorithmen I
Tutorium 32

Eine Lehrveranstaltung im SS 2017 (mit Folien von Christopher Hommel)

Daniel Jungkind (ufesa@kit.edu) | 16. Juni 2017

INSTITUT FÜR THEORETISCHE INFORMATIK

Schwarzes Brett

- **Erinnerung**: Am **21.06.** statt Übungstermin **Probeklausur**! Hingehen lohnt sich!
- Nächstes Mal (23.06.) bin ich **nicht da** – werde aber **vertreten** von **Christopher** (dem Schöpfer der Vorlage dieser Folien ☺).
  Blätter kriegt ihr vermutlich von ihm.

EIMERWEISE SORTIERALGORITHMEN

Eimerweise Sortieralgorithmen

Erinnerung: Bucketsort

- n Elemente **beschränkter** Größe (also $\in \{a, ..., b\}$)
- Lege an $buckets$: **array**$[a..b]$ **of** List **of** Element $\quad (k := |buckets|)$
- Schmeiße jedes Element e in seinen Eimer: $buckets[e].pushBack(e)$ (hinten anhängen)
- Am **Ende**: „Eimer“ zusammenhängen

$\Rightarrow$ Array sortiert.

- **Laufzeit**: $O(n + k)$
- **Aufpassen** bei großen/unbeschränkten k!

CountingSort

Zahlen zählen

- Sagen wir jetzt ganz konkret: n Zahlen $\in \{1, ..., k\}$.
- **Neue Idee: Zähle** (in einem Extra-Array), welche Zahl **wie oft vorkommt** (in $\Theta(n)$)
- Bestimme dann, welche Zahl **in welchen Bereich** des sortierten Arrays gehört (in $\Theta(k)$)
- Füge die Zahlen dann dort ein (in $\Theta(n)$)
- Gesamtlaufzeit von *CountingSort*: $O(n + k)$

CountingSort

Zahlen zählen

- Sagen wir jetzt ganz konkret: n Zahlen $\in \{1, ..., k\}$.
- **Neue Idee**: **Zähle** (in einem Extra-Array), welche Zahl **wie oft vorkommt** (in $\Theta(n)$)
- Bestimme dann, welche Zahl **in welchen Bereich** des sortierten Arrays gehört (in $\Theta(k)$)
- Füge die Zahlen dann dort ein (in $\Theta(n)$)
- Gesamtlaufzeit von *CountingSort*: $O(n + k)$

CountingSort

Zahlen zählen

- Sagen wir jetzt ganz konkret: n Zahlen $\in \{1, ..., k\}$.
- **Neue Idee**: **Zähle** (in einem Extra-Array), welche Zahl **wie oft vorkommt** (in $\Theta(n)$)
- Bestimme dann, welche Zahl **in welchen Bereich** des sortierten Arrays gehört (in $\Theta(k)$)
- Füge die Zahlen dann dort ein (in $\Theta(n)$)
- Gesamtlaufzeit von *CountingSort*: $O(n + k)$

CountingSort

Zahlen zählen

- Sagen wir jetzt ganz konkret: n Zahlen $\in \{1, ..., k\}$.
- **Neue Idee**: **Zähle** (in einem Extra-Array), welche Zahl **wie oft vorkommt** (in $\Theta(n)$)
- Bestimme dann, welche Zahl **in welchen Bereich** des sortierten Arrays gehört (in $\Theta(k)$)
- Füge die Zahlen dann dort ein (in $\Theta(n)$)
- Gesamtlaufzeit von *CountingSort*: $O(n + k)$

CountingSort

Zahlen zählen

- Sagen wir jetzt ganz konkret: n Zahlen $\in \{1, ..., k\}$.
- **Neue Idee**: **Zähle** (in einem Extra-Array), welche Zahl **wie oft vorkommt** (in $\Theta(n)$)
- Bestimme dann, welche Zahl **in welchen Bereich** des sortierten Arrays gehört (in $\Theta(k)$)
- Füge die Zahlen dann dort ein (in $\Theta(n)$)
- Gesamtlaufzeit von *CountingSort*: $O(n + k)$

CountingSort

CountingSort

function CountingSort(A : **array**[1...n] **of** $\mathbb{N}$)

- $C[1...k] := (0, ..., 0)$: **array of** $\mathbb{N}$ *// das Zählerarray*
- $B[1...n]$: **array of** $\mathbb{N}$ *// das sortierte Array (in VL ein Parameter)*
- **for** $i := 1$ **to** n **do**
  - $C[A[i]]++$
- $C[0] := 1$
- **for** $i := 2$ **to** k **do**
  - $C[i] += C[i-1]$
- *// $C[\ell - 1]$ sagt jetzt, wo in B der Bereich für die Zahl $\ell \in A$ beginnt*
- **for** $i := 1$ **to** n **do**
  - $B\left[C\left[A[i]-1\right]\right] := A[i]$ *// $A[i]$ in den zugehörigen Bereich einfügen*
  - $C\left[A[i]-1\right]++$ *// verschiebe Start des Bereiches um eins nach rechts*
- **return** B

CountingSort

Karlsruher Institut für Technologie

Eigenschaften

$\Rightarrow$ **Spezialfall** von Bucketsort: Eimer $\hat{=}$ Zähler

CountingSort

Eigenschaften

$\Rightarrow$ **Spezialfall** von Bucketsort: Eimer $\hat{=}$ Zähler

Ist CountingSort stabil? **?**

CountingSort

Eigenschaften

$\Rightarrow$ **Spezialfall** von Bucketsort: Eimer $\hat{=}$ Zähler

Ist CountingSort stabil? **Ja.**

CountingSort

Eigenschaften

$\Rightarrow$ **Spezialfall** von Bucketsort: Eimer $\hat{=}$ Zähler

Ist CountingSort stabil? **Ja.**

Ist Countingsort in-place? **?**

CountingSort

Eigenschaften

$\Rightarrow$ **Spezialfall** von Bucketsort: Eimer $\hat{=}$ Zähler

Ist CountingSort stabil? **Ja.**

Ist Countingsort in-place? **Nein.**

Eimerweise Sortieralgorithmen

Aufgabe 1: Das zählt nicht

Gegeben sei $A \in \left(\bigcup_{i=1}^{k} \left\{ i,\ i + \frac{1}{2} \right\} \right)^n$. Gebt ein Verfahren an, mit dem A in $O(n + k)$ sortiert werden kann.

Eimerweise Sortieralgorithmen

Lösung zu Aufgabe 1

(A ist ein **Tupel** (da Element eines kartesischen Produktes).)
Sortiere $A' := 2 \cdot A$ durch *CountingSort* mit $k' := 2k + 1$ und teile jeden Wert im sortierten Array wieder durch 2.
Laufzeit: Da $|A'| = |A| = n$ in
$O(n + k')$ = $O(n + 2k + 1)$ = $O(n + k)$

Radixsort

Ein neuer Ansatz

- **Geg.**: n Zahlen $\in \mathbb{N}_0$ in Darstellung zur Basis K mit jeweils d Stellen pro Zahl
- **Idee:** $\forall$ Stelle von niedrigstwertig nach höchstwertig:
  Sortiere mit **Bucketsort** (Buckets von 0 bis $K-1$) nach **dieser** Stelle
- **Warum** geht das?
  $\Rightarrow$ **Stabilität** von Bucketsort
- **Laufzeit** von (LSD-)Radixsort: $O(d \cdot (n+K))$
  (LSD: **L**owest **s**ignificant **d**igit)

Radixsort

Ein neuer Ansatz

- **Geg.**: n Zahlen $\in \mathbb{N}_0$ in Darstellung zur Basis K mit jeweils d Stellen pro Zahl
- **Idee**: $\forall$ Stelle von niedrigstwertig nach höchstwertig:
  Sortiere mit **Bucketsort** (Buckets von 0 bis $K-1$) nach **dieser** Stelle
- **Warum** geht das?
  $\Rightarrow$ **Stabilität** von Bucketsort
- **Laufzeit** von (LSD-)Radixsort: $O(d \cdot (n+K))$
  (LSD: Lowest significant digit)

Radixsort

Ein neuer Ansatz

- **Geg.**: n Zahlen $\in \mathbb{N}_0$ in Darstellung zur Basis K mit jeweils d Stellen pro Zahl
- **Idee**: $\forall$ Stelle von niedrigstwertig nach höchstwertig:
  Sortiere mit **Bucketsort** (Buckets von 0 bis $K-1$) nach **dieser** Stelle
- **Warum** geht das?
  $\Rightarrow$ **Stabilität** von Bucketsort
- **Laufzeit** von (LSD-)Radixsort: $O(d \cdot (n+K))$
  (LSD: Lowest significant digit)

Radixsort

Ein neuer Ansatz

- **Geg.**: n Zahlen $\in \mathbb{N}_0$ in Darstellung zur Basis K mit jeweils d Stellen pro Zahl
- **Idee**: $\forall$ Stelle von niedrigstwertig nach höchstwertig:
  Sortiere mit **Bucketsort** (Buckets von 0 bis $K - 1$) nach **dieser** Stelle
- **Warum** geht das?
  $\Rightarrow$ **Stabilität** von Bucketsort
- **Laufzeit** von (LSD-)Radixsort: $O(d \cdot (n + K))$
  (LSD: **L**owest **s**ignificant **d**igit)

Eimerweise Sortieralgorithmen

Aufgabe 2: Radixchensalat

Sortiert die folgende Liste mit *LSD-Radixsort*:

$\langle 36, 78, 50, 1, 92, 15, 43, 99, 64 \rangle$

Eimerweise Sortieralgorithmen

Lösung zu Aufgabe 2

Eingabe: $\langle 36, 78, 50, 01, 92, 15, 43, 99, 64 \rangle$

Nach der ersten Stelle (von rechts):
$\langle 5\mathbf{0}, 0\mathbf{1}, 9\mathbf{2}, 4\mathbf{3}, 6\mathbf{4}, 1\mathbf{5}, 3\mathbf{6}, 7\mathbf{8}, 9\mathbf{9} \rangle$

Nach der zweiten Stelle:
$\langle \mathbf{0}1, \mathbf{1}5, \mathbf{3}6, \mathbf{4}3, \mathbf{5}0, \mathbf{6}4, \mathbf{7}8, \mathbf{9}2, \mathbf{9}9 \rangle$

(LSD-)Radixsort

Eigenschaften

- **Laufzeit**: linear (für **konstante** d **und** K!)
  (für große d: Verfahren in $\Theta(n \log n)$ geeigneter)
- Ebenfalls **Spezialfall** von *BucketSort*
- ... und ebenfalls **stabil**, aber **nicht in-place**.
- *MSD-Radixsort* gibt's auch, wird hier nicht behandelt

(LSD-)Radixsort

Eigenschaften

- **Laufzeit**: linear (für **konstante** d **und** K!)
  (für große d: Verfahren in $\Theta(n \log n)$ geeigneter)
- Ebenfalls **Spezialfall** von *BucketSort*
- ... und ebenfalls **stabil**, aber **nicht in-place**.
- *MSD-Radixsort* gibt's auch, wird hier nicht behandelt

(LSD-)Radixsort

Karlsruher Institut für Technologie

Eigenschaften

- **Laufzeit**: linear (für **konstante** d **und** K!)
  (für große d: Verfahren in $\Theta(n \log n)$ geeigneter)
- Ebenfalls **Spezialfall** von *BucketSort*
- ... und ebenfalls **stabil**, aber **nicht in-place**.
- *MSD-Radixsort* gibt's auch, wird hier nicht behandelt

(LSD-)Radixsort

Eigenschaften

- **Laufzeit**: linear (für **konstante** d **und** K!)
  (für große d: Verfahren in $\Theta(n \log n)$ geeigneter)
- Ebenfalls **Spezialfall** von *BucketSort*
- ... und ebenfalls **stabil**, aber **nicht in-place**.
- *MSD-Radixsort* gibt's auch, wird hier nicht behandelt

(LSD-)Radixsort

Eigenschaften

- **Laufzeit**: linear (für **konstante** d **und** K!)
  (für große d: Verfahren in $\Theta(n \log n)$ geeigneter)
- Ebenfalls **Spezialfall** von *BucketSort*
- ... und ebenfalls **stabil**, aber **nicht in-place**.
- *MSD-Radixsort* gibt's auch, wird hier nicht behandelt

Eimerweise Sortieralgorithmen

Aufgabe 3: Liste der bedrohten Sortierarten

Gegeben seien n Zahlen im Bereich von 0 bis $n^3 - 1$. Gebt ein Verfahren an, mit dem diese in $\Theta(n)$ sortiert werden können.

Eimerweise Sortieralgorithmen

Lösung zu Aufgabe 3

Betrachte die Zahlen in Darstellung zur Basis n, d.h. jede Zahl hat in dieser Darstellung 3 Stellen.
Wende *RadixSort* an $\Rightarrow$ die Zahlen werden in $\Theta(3 \cdot (n + n)) = \Theta(n)$ sortiert.

Eimerweise Sortieralgorithmen

Karlsruher Institut für Technologie

Aufgabe 4: PancakeSort

Für eine Familienfeier habt ihr euch großzügigerweise bereiterklärt, einen großen Stapel Pfannkuchen zu liefern. Ärgerlicherweise bemerkt ihr zu spät, dass eure Schöpfkelle ein Loch hat, so dass jedesmal eine unterschiedliche Menge Teig in der Pfanne gelandet ist – somit ist jeder Pfannkuchen unterschiedlich groß und der Stapel sieht völlig chaotisch aus, weshalb *bestimmt* die Welt untergeht. Glücklicherweise steht euch euer Vetter Donald mit Rat und Tat zur Seite, denn er ist der unbestrittene Weltmeister in der Kunst des Pfannkuchenwendens. Phänomenalerweise ist er sogar in der Lage, einen beliebig großen (Teil-)Stapel Pfannkuchen in einem Schwung komplett umzudrehen! Leider ist er seit einem Zwischenfall beim Pfannenwenderweitwurf etwas beschränkt, so dass er auf eure Anleitung angewiesen ist. Gebt ein In-place-Verfahren an, mit dem der Pfannkuchenstapel in möglichst geringer Zeit (bzgl. Anzahl der Pfannkuchen) sortiert werden kann.

Eimerweise Sortieralgorithmen

Lösung zu Aufgabe 4

- Zu Beginn sieht der Stapel so aus: [bottom...biggest...top]
- Wende Teilstapel [biggest..top] *// Größter nach oben*
- Wende ganzen Stapel *// Größter ganz unten*
- Für alle Teilstapel über dem nun korrekt einsortierten: **Wiederhole**.

$\Rightarrow$ Pro Pfannkuchen höchstens 2-mal wenden $\Rightarrow O(n)$.

BINÄRE HEAPS

And now for something completely different...

Binäre Heaps

- **Binärbaum**: Baum, wobei jeder Knoten **max. zwei Kinder** hat.
- Ein Baum T erfüllt die **Heap-Eigenschaft** $:\Leftrightarrow$
  $\forall v \in T: \quad parent(v) \leqslant v$
  (je näher an Wurzel = Minimum, desto kleiner die Werte)
  Achtung: Das heißt **nicht** „~~T ist sortiert~~“!
- Ein **binärer Heap** ist ein Binärbaum, der die Heap-Eigenschaft erfüllt.
- **Wofür?** $\Rightarrow$ (un/beschränkte) *PriorityQueues* (brauchen wir später noch!)

Binäre Heaps

- **Binärbaum**: Baum, wobei jeder Knoten **max. zwei Kinder** hat.
- Ein Baum T erfüllt die **Heap-Eigenschaft** $:\Leftrightarrow$
  $\forall v \in T : \quad \mathit{parent}(v) \leqslant v$
  (je näher an Wurzel $=$ Minimum, desto kleiner die Werte)
  Achtung: Das heißt **nicht** ~~„T ist sortiert“~~!
- Ein **binärer Heap** ist ein Binärbaum, der die Heap-Eigenschaft erfüllt.
- **Wofür?** $\Rightarrow$ (un/beschränkte) *PriorityQueues* (brauchen wir später noch!)

Binäre Heaps

- **Binärbaum**: Baum, wobei jeder Knoten **max. zwei Kinder** hat.
- Ein Baum T erfüllt die **Heap-Eigenschaft** $:\Leftrightarrow$
  $\forall v \in T: \quad parent(v) \leqslant v$
  (je näher an Wurzel $=$ Minimum, desto kleiner die Werte)
  Achtung: Das heißt **nicht** ~~„T ist sortiert“~~!
- Ein **binärer Heap** ist ein Binärbaum, der die Heap-Eigenschaft erfüllt.
- Wofür? $\Rightarrow$ (un/beschränkte) *PriorityQueues* (brauchen wir später noch!)

Binäre Heaps

- **Binärbaum**: Baum, wobei jeder Knoten **max. zwei Kinder** hat.
- Ein Baum T erfüllt die **Heap-Eigenschaft** $:\Leftrightarrow$
  $\forall v \in T: \quad parent(v) \leqslant v$
  (je näher an Wurzel $=$ Minimum, desto kleiner die Werte)
  Achtung: Das heißt **nicht** ~~„T ist sortiert“~~!
- Ein **binärer Heap** ist ein Binärbaum, der die Heap-Eigenschaft erfüllt.
- **Wofür**? $\Rightarrow$ (un/beschränkte) *PriorityQueues* (brauchen wir später noch!)

Binäre Heaps

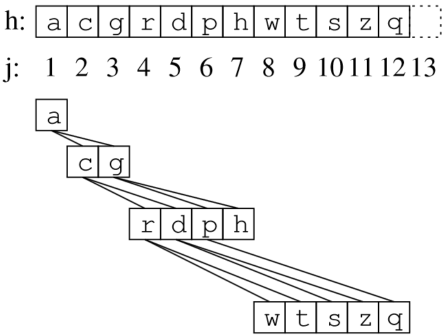
Implementierung

- Repräsentiere binären Baum als **array**[1…*n*]
- Die Ebenen des Baumes liegen von **oben** ⇝ **unten** und von **links** ⇝ **rechts** nacheinander im Array
- Von Knoten *j* kriegt man **Eltern** und **Kinder** wie folgt:

  $parent(j) = \left\lfloor \frac{j}{2} \right\rfloor$

  $leftChild(j) = 2j$

  $rightChild(j) = 2j + 1$



Binäre Heaps

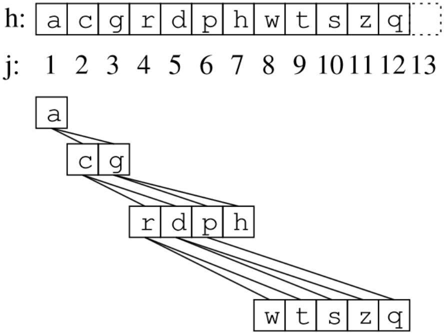
Implementierung

- Repräsentiere binären Baum als **array**[1...n]
- Die Ebenen des Baumes liegen von **oben** $\rightsquigarrow$ **unten** und von **links** $\rightsquigarrow$ **rechts** nacheinander im Array
- Von Knoten j kriegt man **Eltern** und **Kinder** wie folgt:

$$parent(j) = \left\lfloor \frac{j}{2} \right\rfloor$$

$$leftChild(j) = 2j$$

$$rightChild(j) = 2j + 1$$



Binäre Heaps

Einfügen von Elementen (insert)

- Setze Element *e* in die **unterste** Ebene, so weit **rechts** wie möglich
- Heap-Eigenschaft **fixen**: mit *siftUp*
  Vertausche *e* solange mit seinem Parent, bis wieder erfüllt
- Laufzeit: $O(\log n)$

Binäre Heaps

Einfügen von Elementen (insert)

- Setze Element e in die **unterste** Ebene, so weit **rechts** wie möglich
- Heap-Eigenschaft **fixen**: mit *siftUp*
  Vertausche e solange mit seinem Parent, bis wieder erfüllt
- Laufzeit: $O(\log n)$

Binäre Heaps

Einfügen von Elementen (insert)

- Setze Element e in die **unterste** Ebene, so weit **rechts** wie möglich
- Heap-Eigenschaft **fixen**: mit *siftUp*
  Vertausche e solange mit seinem Parent, bis wieder erfüllt
- Laufzeit: $O(\log n)$

Binäre Heaps

Entfernen des Minimums (deleteMin)

- Einfach: Minimum $= A[1]$ „**oben wegnehmen**“
- **Lücke schließen**: Letztes Element u aus **unterster** Ebene nach oben holen ($A[1] := A[n] = u$).
- Heap-Eigenschaft **fixen**: mit *siftDown*
  Vertausche u solange mit dem jew. **kleinsten** Kind, bis wieder erfüllt
- Laufzeit: $O(\log n)$

Minimum abfragen (min)

- return $A[1]$
- Laufzeit: $O(1)$

Binäre Heaps

Entfernen des Minimums (deleteMin)

- Einfach: Minimum $= A[1]$ „**oben wegnehmen**“
- **Lücke schließen**: Letztes Element u aus **unterster** Ebene nach oben holen ($A[1] := A[n] = u$).
- Heap-Eigenschaft **fixen**: mit *siftDown*
  Vertausche u solange mit dem jew. **kleinsten** Kind, bis wieder erfüllt
- Laufzeit: $O(\log n)$

Minimum abfragen (min)

- return $A[1]$
- Laufzeit: $O(1)$

Binäre Heaps

Entfernen des Minimums (deleteMin)

- Einfach: Minimum $= A[1]$ „**oben wegnehmen**“
- **Lücke schließen**: Letztes Element u aus **unterster** Ebene nach oben holen ($A[1] := A[n] = u$).
- Heap-Eigenschaft **fixen**: mit *siftDown*
  Vertausche u solange mit dem jew. **kleinsten** Kind, bis wieder erfüllt
- Laufzeit: $O(\log n)$

Minimum abfragen (min)

- return $A[1]$
- Laufzeit: $O(1)$

Binäre Heaps

Entfernen des Minimums (deleteMin)

- Einfach: Minimum $= A[1]$ „**oben wegnehmen**“
- **Lücke schließen**: Letztes Element u aus **unterster** Ebene nach oben holen ($A[1] := A[n] = u$).
- Heap-Eigenschaft **fixen**: mit *siftDown*
  Vertausche u solange mit dem jew. **kleinsten** Kind, bis wieder erfüllt
- Laufzeit: $O(\log n)$

Minimum abfragen (min)

- return $A[1]$
- Laufzeit: $O(1)$

Binäre Heaps

Entfernen des Minimums (deleteMin)

- Einfach: Minimum $= A[1]$ „**oben wegnehmen**"
- **Lücke schließen**: Letztes Element u aus **unterster** Ebene nach oben holen ($A[1] := A[n] = u$).
- Heap-Eigenschaft **fixen**: mit *siftDown*
  Vertausche u solange mit dem jew. **kleinsten** Kind, bis wieder erfüllt
- Laufzeit: $O(\log n)$

Minimum abfragen (min)

- **return** $A[1]$
- Laufzeit: $O(1)$

Binäre Heaps

Karlsruher Institut für Technologie

Aufbau aus einem chaotischen Array (buildHeap)

- Haben chaotisches Array A, wollen **Heapstruktur** auf A herstellen

$\Rightarrow$ Systematisch richtigrum tauschen:

foreach *Ebene* $\in$ *Zweittiefste ... Oberste* **do**
    for *elem* $\in$ *Ebene* **from** *right* **to** *left* **do**
        if *elem too high* **then** siftDown(*elem*)

- Klingt nach $O(n \log n)$, aber Vorlesung sagt:
  Laufzeit $O(n)$

Binäre Heaps

Karlsruher Institut für Technologie

Aufbau aus einem chaotischen Array (buildHeap)

- Haben chaotisches Array A, wollen **Heapstruktur** auf A herstellen

$\Rightarrow$ Systematisch richtigrum tauschen:

foreach *Ebene* $\in$ *Zweittiefste ... Oberste* **do**
  for *elem* $\in$ *Ebene* **from** *right* **to** *left* **do**
    if *elem too high* **then** siftDown(*elem*)

- Klingt nach $O(n \log n)$, aber Vorlesung sagt:
  Laufzeit $O(n)$

Haufenweise Sortieralgorithmen

Heapsort

- buildHeap(A)   in $O(n)$
- n-mal: deleteMin()   jeweils in $O(\log n)$

  Nach jedem deleteMin() wird ein **Platz hinten frei**: Schmeiß es **dorthin**! ($\Rightarrow$ liefert **absteigende** Sortierung)

$\Rightarrow$ **Gesamt-Laufzeit**: $O(n \log n)$

− **nicht** stabil   (wg. siftUp/Down)

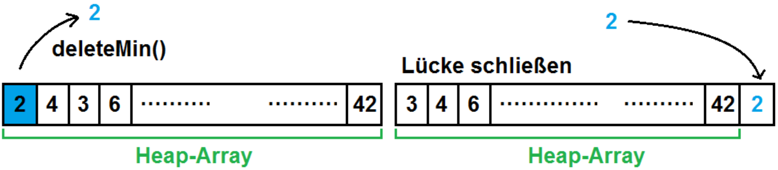
+ in-place

+ Cache-effizient

Haufenweise Sortieralgorithmen

Heapsort

- buildHeap(A) in $O(n)$
- n-mal: deleteMin() jeweils in $O(\log n)$
  Nach jedem deleteMin() wird ein **Platz hinten frei**: Schmeiß es **dorthin**! ($\Rightarrow$ liefert **absteigende** Sortierung)

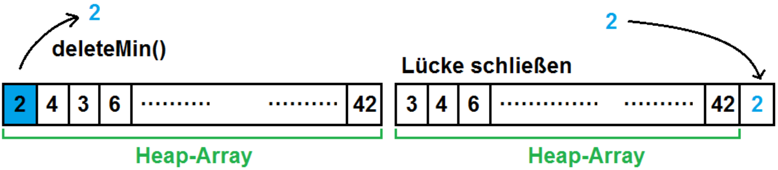


$\Rightarrow$ Gesamt-Laufzeit: $O(n \log n)$
- − nicht stabil (wg. siftUp/Down)
- + in-place
- + Cache-effizient

Haufenweise Sortieralgorithmen

Heapsort

- buildHeap(A)   in $O(n)$
- n-mal: deleteMin()   jeweils in $O(\log n)$
  Nach jedem deleteMin() wird ein **Platz hinten frei**: Schmeiß es **dorthin**! ($\Rightarrow$ liefert **absteigende** Sortierung)



$\Rightarrow$ **Gesamt-Laufzeit**: $O(n \log n)$

➖ **nicht** stabil   (wg. siftUp/Down)

➕ in-place

➕ Cache-effizient

SORTING ALGORITHMS

– Literally –

Sorting Algorithms

Sortiere Mergesort, Radixsort, Heapsort, InsertionSort, SelectionSort, CountingSort, Quicksort (mit partition), (simples) Bucketsort nach **Stabilität**.

InsertionSort SelectionSort Mergesort CountingSort Bucketsort Radixsort	**Ja**
Heapsort Quicksort	**Nein**

Sorting Algorithms

Sortiere Mergesort, Radixsort, Heapsort, InsertionSort, SelectionSort, CountingSort, Quicksort (mit partition), (simples) Bucketsort nach **Stabilität**.

InsertionSort SelectionSort Mergesort CountingSort Bucketsort Radixsort	**Ja**
Heapsort Quicksort	**Nein**

Sorting Algorithms

Sortiere Mergesort, Radixsort, Heapsort, InsertionSort, SelectionSort, CountingSort, Quicksort (mit partition), (simples) Bucketsort nach **Cache-Effizienz**.

InsertionSort SelectionSort Heapsort CountingSort Quicksort	Ja
Bucketsort Mergesort Radixsort	Nein

Sorting Algorithms

Sortiere Mergesort, Radixsort, Heapsort, InsertionSort, SelectionSort, CountingSort, Quicksort (mit partition), (simples) Bucketsort nach **Cache-Effizienz**.

InsertionSort SelectionSort Heapsort CountingSort Quicksort	**Ja**
Bucketsort Mergesort Radixsort	**Nein**

Sorting Algorithms

Sortiere Mergesort, Radixsort, Heapsort, InsertionSort, SelectionSort, CountingSort, Quicksort (mit partition), (simples) Bucketsort nach **Platzverbrauch**.

InsertionSort SelectionSort Mergesort (ohne Rekursionsoverhead) Quicksort (ohne Rekursionsoverhead)	$O(1)$
Heapsort	$O(n)$
CountingSort Bucketsort	$O(n+k)$
Radixsort	$O(n+K)$

Sorting Algorithms

Sortiere Mergesort, Radixsort, Heapsort, InsertionSort, SelectionSort, CountingSort, Quicksort (mit partition), (simples) Bucketsort nach **Platzverbrauch**.

InsertionSort SelectionSort Mergesort (ohne Rekursionsoverhead) Quicksort (ohne Rekursionsoverhead)	$O(1)$
Heapsort	$O(n)$
CountingSort Bucketsort	$O(n+k)$
Radixsort	$O(n+K)$

Sorting Algorithms

Sortiere Mergesort, Radixsort, Heapsort, InsertionSort, SelectionSort, CountingSort, Quicksort (mit partition), (simples) Bucketsort nach **Worst-Case-Laufzeit**.

Mergesort Heapsort	$O(n \log n)$
Quicksort InsertionSort SelectionSort	$O(n^2)$

Radixsort	$O(d \cdot (n + K))$ (K: Basis, d: Digits)
Bucketsort Countingsort	$O(n + k)$ (k: „maxValue“)

Sorting Algorithms

Sortiere Mergesort, Radixsort, Heapsort, InsertionSort, SelectionSort, CountingSort, Quicksort (mit partition), (simples) Bucketsort nach **Worst-Case-Laufzeit**.

Mergesort Heapsort	$O(n \log n)$
Quicksort InsertionSort SelectionSort	$O(n^2)$

Radixsort	$O(d \cdot (n + K))$ (K: Basis, d: Digits)
Bucketsort Countingsort	$O(n + k)$ (k: „maxValue“)

Sorting Algorithms

Sortiere Mergesort, Radixsort, Heapsort, InsertionSort, SelectionSort, CountingSort, Quicksort (mit partition), (simples) Bucketsort nach **„Standard-Laufzeit"**.

Mergesort Heapsort Quicksort (erwartet)	$O(n \log n)$
InsertionSort SelectionSort	$O(n^2)$

Radixsort	$O(d \cdot (n + K))$ (K: Basis, d: Digits)
Bucketsort Countingsort	$O(n + k)$ (k: „maxValue")

Sorting Algorithms

Sortiere Mergesort, Radixsort, Heapsort, InsertionSort, SelectionSort, CountingSort, Quicksort (mit partition), (simples) Bucketsort nach **„Standard-Laufzeit“**.

Mergesort Heapsort Quicksort (erwartet)	$O(n \log n)$
InsertionSort SelectionSort	$O(n^2)$

Radixsort	$O(d \cdot (n + K))$ (K: Basis, d: Digits)
Bucketsort Countingsort	$O(n + k)$ (k: „maxValue“)

Schönes Wochenende noch! ☺