Script **generated by TTT**

Title: Täubig: GAD (27.05.2014)

Date: Tue May 27 13:53:20 CEST 2014

Duration: 124:52 min

Pages: 44

Übersicht

Sortieren schneller als $\mathcal{O}(n \log n)$

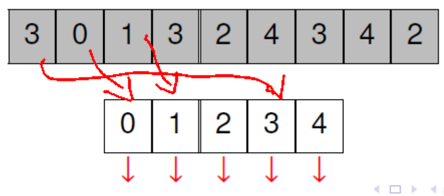
Buckets

- mit paarweisen Schlüsselvergleichen: nie besser als $\mathcal{O}(n \log n)$
- Was aber, wenn die Schlüsselmenge mehr Struktur hat? z.B. Zahlen / Strings bestehend aus mehreren Ziffern / Zeichen
- Um zwei Zahlen / Strings zu vergleichen reicht oft schon die erste Ziffer / das erste Zeichen. Nur bei gleichem Anfang kommt es auf mehr Ziffern / Zeichen an.

Sortieren schneller als $\mathcal{O}(n \log n)$

Buckets

- mit paarweisen Schlüsselvergleichen: nie besser als $\mathcal{O}(n \log n)$
- Was aber, wenn die Schlüsselmenge mehr Struktur hat? z.B. Zahlen / Strings bestehend aus mehreren Ziffern / Zeichen
- Um zwei Zahlen / Strings zu vergleichen reicht oft schon die erste Ziffer / das erste Zeichen. Nur bei gleichem Anfang kommt es auf mehr Ziffern / Zeichen an.
- Annahme: Elemente sind Zahlen im Bereich $\{0, \ldots, K-1\}$
- Strategie: verwende Feld von K Buckets (z.B. Listen)



Sortieren schneller als $\mathcal{O}(n \log n)$

Buckets

```
Sequence<Elem> kSort(Sequence<Elem> s) {
   Sequence<Elem>[] b = new Sequence<Elem>[K];
   foreach (e ∈ s)
      b[key(e)].pushBack(e);
   return concatenate(b);   // Aneinanderreihung von b[0],...,b[k-1]
}
```


Sortieren schneller als $\mathcal{O}(n \log n)$

Buckets

```
Sequence<Elem> kSort(Sequence<Elem> s) {
   Sequence<Elem>[] b = new Sequence<Elem>[K];
   foreach (e ∈ s)
      b[key(e)].pushBack(e);
   return concatenate(b);   // Aneinanderreihung von b[0],...,b[k-1]
}
```

Laufzeit: $\Theta(n + K)$ Problem: nur gut für $K \in o(n \log n)$

Speicher: $\Theta(n + K)$

Sortieren schneller als $\mathcal{O}(n \log n)$

Buckets

```
Sequence<Elem> kSort(Sequence<Elem> s) {
   Sequence<Elem>[] b = new Sequence<Elem>[K];
   foreach (e ∈ s)
      b[key(e)].pushBack(e);
   return concatenate(b);   // Aneinanderreihung von b[0],...,b[k-1]
}
```

Laufzeit: $\Theta(n + K)$ Problem: nur gut für $K \in o(n \log n)$

Speicher: $\Theta(n + K)$

- wichtig: kSort ist stabil, d.h. Elemente mit dem gleichen Schlüssel behalten ihre relative Reihenfolge

$\Rightarrow$ Elemente müssen im jeweiligen Bucket *hinten* angehängt werden

RadixSort

- verwende K-adische Darstellung der Schlüssel
- Annahme:
  Schlüssel sind Zahlen aus $\{0, \ldots, K^d - 1\}$ repräsentiert durch d Stellen von Ziffern aus $\{0, \ldots, K - 1\}$
- sortiere zunächst entsprechend der niedrigstwertigen Ziffer mit kSort und dann nacheinander für immer höherwertigere Stellen
- behalte Ordnung der Teillisten bei

RadixSort

radixSort(Sequence<Elem> s) {
  for (int $i = 0$; $i < d$; $i++$)
    kSort(s,i);   // *sortiere gemäß* $key_i(x)$
          // *mit* $key_i(x) = (key(x)/K^i) \bmod K$
}

RadixSort

radixSort(Sequence<Elem> s) {
  for (int $i = 0$; $i < d$; $i++$)
    kSort(s,i);   // *sortiere gemäß* $key_i(x)$
          // *mit* $key_i(x) = (key(x)/K^i) \bmod K$
}

Verfahren funktioniert, weil kSort stabil ist:

Elemente mit gleicher i-ter Ziffer bleiben sortiert bezüglich der Ziffern $i-1 \ldots 0$ während der Sortierung nach Ziffer i

Laufzeit: $O(d(n+K))$ für n Schlüssel aus $\{0, \ldots, K^d - 1\}$

RadixSort

Beispiel

012	203	003	074	024	017	112

RadixSort

Beispiel

012	203	003	074	024	017	112

012	112	203	003	074	024	017

RadixSort

Beispiel

012 203 003 074 024 017 112

012 112 | 203 003 | 074 024 | 017

012 112 203 003 074 024 017

Sortieren Schnelleres Sortieren

RadixSort

Beispiel

012 203 003 074 024 017 112

012 112 | 203 003 | 074 024 | 017

012 112 203 003 074 024 017

203 003 | 012 112 017 | 024 | 074

Sortieren Schnelleres Sortieren

RadixSort

Beispiel

012 203 003 074 024 017 112

012 112 | 203 003 | 074 024 | 017

012 112 203 003 074 024 017

203 003 | 012 112 017 | 024 | 074

203 003 012 112 017 024 074

Sortieren Schnelleres Sortieren

RadixSort

Beispiel

012 203 003 074 024 017 112

012 112 | 203 003 | 074 024 | 017

012 112 203 003 074 024 017

203 003 | 012 112 017 | 024 | 074

203 003 012 112 017 024 074

003 012 017 024 074 | 112 | 203

RadixSort

Beispiel

012 203 003 074 024 017 112

012 112 | 203 003 | 074 024 | 017

012 112 203 003 074 024 017

203 003 | 012 112 017 | 024 | 074

203 003 012 112 017 024 074

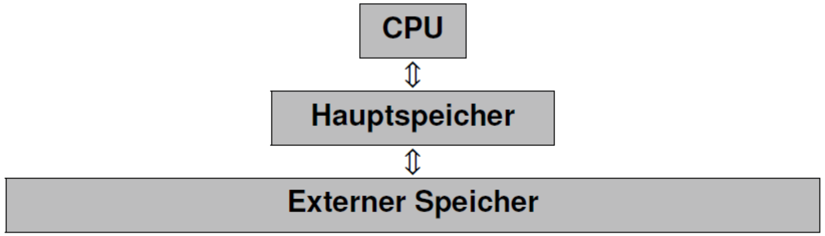
003 012 017 024 074 | 112 | 203

003 012 017 024 074 112 203

Übersicht

Externes Sortieren

Heutige Computer:



- Hauptspeicher hat Größe M
- Transfer zwischen Hauptspeicher und externem Speicher mit Blockgröße B

Externes Sortieren

Problem:
Minimiere Anzahl Blocktransfers zwischen internem und externem Speicher

Anmerkung:
Gleiches Problem trifft auch auf anderen Stufen der Hierarchie zu (Cache)

Externes Sortieren

Problem:
Minimiere Anzahl Blocktransfers zwischen internem und externem Speicher

Anmerkung:
Gleiches Problem trifft auch auf anderen Stufen der Hierarchie zu (Cache)

Lösung: Verwende MergeSort

Vorteil:

MergeSort verwendet oft konsekutive Elemente (Scanning)
(geht auf Festplatte schneller als Random Access-Zugriffe)

Externes Sortieren

- Eingabe: großes Feld auf der Festplatte
- Annahme: Anzahl der Elemente n ist durch B teilbar ($B \mid M$)
  (sonst z.B. Auffüllen mit maximalem Schlüssel)

Externes Sortieren

- Eingabe: großes Feld auf der Festplatte
- Annahme: Anzahl der Elemente n ist durch B teilbar ($B \mid M$)
  (sonst z.B. Auffüllen mit maximalem Schlüssel)

Run Formation Phase:

- Lade wiederholt Teilfeld der Größe M in den Speicher,
- sortiere es mit einem in-place-Sortierverfahren,
- schreibe sortiertes Teilfeld (Run) wieder zurück auf die Festplatte

Externes Sortieren

- Eingabe: großes Feld auf der Festplatte
- Annahme: Anzahl der Elemente n ist durch B teilbar ($B \mid M$)
  (sonst z.B. Auffüllen mit maximalem Schlüssel)

Run Formation Phase:

- Lade wiederholt Teilfeld der Größe M in den Speicher,
- sortiere es mit einem in-place-Sortierverfahren,
- schreibe sortiertes Teilfeld (Run) wieder zurück auf die Festplatte

$\Rightarrow$ benötigt n/B Blocklese- und n/B Blockschreiboperationen
Laufzeit: $2n/B$ Transfers

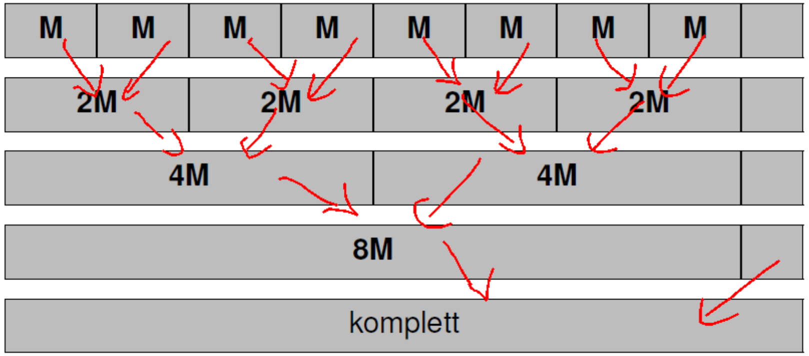
- ergibt sortierte Bereiche (Runs) der Größe M

M	**M**	**M**	**M**	**M**	**M**	**M**	**M**	

Externes Sortieren

Merge Phasen

- Merge von jeweils 2 Teilfolgen in $\lceil \log_2(n/M) \rceil$ Phasen
- dabei jeweils Verdopplung der Größe der sortierten Teile



Merge von zwei Runs

- von jedem der beiden Runs und von der Ausgabesequenz bleibt ein Block im Hauptspeicher (3 Puffer: 2× Eingabe, 1× Ausgabe)
- Anfang: beide Eingabepuffer mit B Elementen (1 Block) laden, Ausgabepuffer leer
- Dann: jeweils führende Elemente der beiden Eingabepuffer vergleichen und das kleinere in den Ausgabepuffer schreiben
- Wenn Eingabepuffer leer $\Rightarrow$ neuen Block laden
- Wenn Ausgabepuffer voll $\Rightarrow$ Block auf Festplatte schreiben und Ausgabepuffer leeren

Merge von zwei Runs

- von jedem der beiden Runs und von der Ausgabesequenz bleibt ein Block im Hauptspeicher (3 Puffer: 2× Eingabe, 1× Ausgabe)
- Anfang: beide Eingabepuffer mit B Elementen (1 Block) laden, Ausgabepuffer leer
- Dann: jeweils führende Elemente der beiden Eingabepuffer vergleichen und das kleinere in den Ausgabepuffer schreiben
- Wenn Eingabepuffer leer $\Rightarrow$ neuen Block laden
- Wenn Ausgabepuffer voll $\Rightarrow$ Block auf Festplatte schreiben und Ausgabepuffer leeren

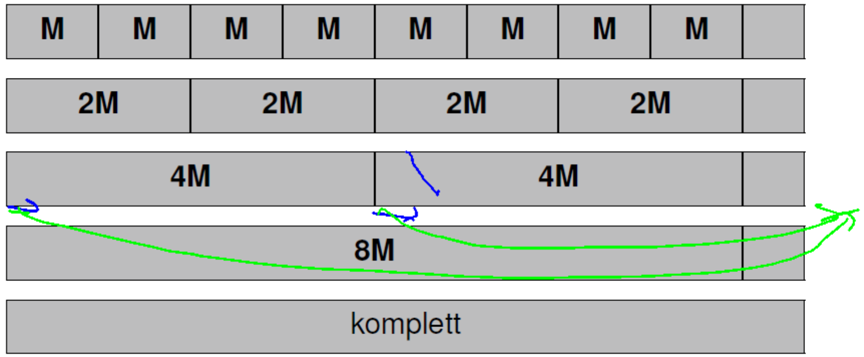
- In jeder Merge-Phase wird das ganze Feld einmal gelesen und geschrieben

$\Rightarrow$ $(2n/B)(1 + \lceil \log_2(n/M) \rceil)$ Block-Transfers

Externes Sortieren

Merge Phasen

- Merge von jeweils 2 Teilfolgen in $\lceil \log_2(n/M) \rceil$ Phasen
- dabei jeweils Verdopplung der Größe der sortierten Teile



Multiway-MergeSort

- Verfahren funktioniert, wenn 3 Blöcke in den Speicher passen
- Wenn mehr Blöcke in den Speicher passen, kann man gleich $k \geq 2$ Runs mergen.
- Benutze Prioritätswarteschlange (Priority Queue) zur Minimumermittlung, wobei die Operationen $O(\log k)$ Zeit kosten
- $(k+1)$ Blocks und die PQ müssen in den Speicher passen

$\Rightarrow$ $(k+1)B + O(k) \leq M$, also $k \in O(M/B)$

- Anzahl Merge-Phasen reduziert auf $\lceil \log_k (n/M) \rceil$

$\Rightarrow$ $(2n/B)\left(1 + \left\lceil \log_{M/B}(n/M) \right\rceil\right)$ Block-Transfers

- In der Praxis: Anzahl Merge-Phasen gering
- Wenn $n \leq M^2/B$: nur eine einzige Merge-Phase (erst M/B Runs der Größe M, dann einmal Merge)

Übersicht

Prioritätswarteschlangen

M: Menge von Elementen

prio(e): Priorität von Element e

Operationen:

- M.build($\{e_1, \ldots, e_n\}$): $M = \{e_1, \ldots, e_n\}$
- M.insert(Element e): $M = M \cup e$
- Element M.min(): gib ein e mit minimaler Priorität prio(e) zurück
- Element M.deleteMin(): entferne Element e mit minimalem Wert prio(e) und gib es zurück

Adressierbare Prioritätswarteschlangen

Zusätzliche Operationen für adressierbare Priority Queues:

- Handle insert(Element e): wie zuvor, gibt aber ein Handle (Referenz / Zeiger) auf das eingefügte Element zurück
- remove(Handle h): lösche Element spezifiziert durch Handle h
- decreaseKey(Handle h, int k): reduziere Schlüssel / Priorität des Elements auf Wert k (je nach Implementation evt. auch um Differenz k)
- M.merge(Q): $M = M \cup Q$; $Q = \emptyset$;

Prioritätswarteschlangen mit Listen

Priority Queue mittels unsortierter Liste:

- build($\{e_1, \dots, e_n\}$): Zeit $O(n)$
- insert(Element e): Zeit $O(1)$
- min(), deleteMin(): Zeit $O(n)$

Priority Queue mittels sortierter Liste:

- build($\{e_1, \dots, e_n\}$): Zeit $O(n \log n)$
- insert(Element e): Zeit $O(n)$
- min(), deleteMin(): Zeit $O(1)$

$\Rightarrow$ Bessere Struktur als eine Liste notwendig!

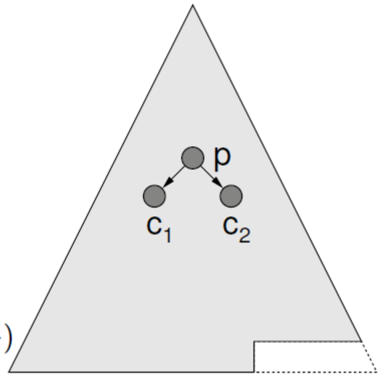
Binärer Heap

Idee: verwende Binärbaum

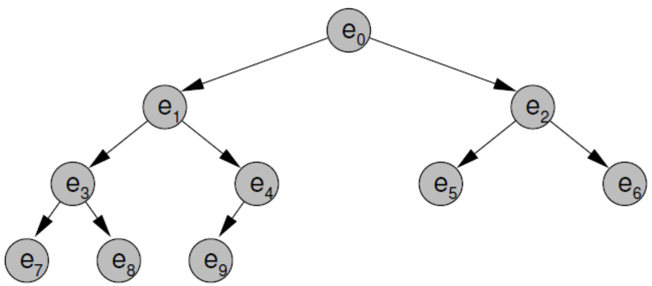
Bewahre zwei Invarianten:

- Form-Invariante: fast vollständiger Binärbaum
- Heap-Invariante:

  $\text{prio}(p) \leq \min\{\text{prio}(c1), \text{prio}(c2)\}$



Binärer Heap als Feld



e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9

- Kinder von Knoten $H[i]$ in $H[2i+1]$ und $H[2i+2]$
- Form-Invariante: $H[0] \dots H[n-1]$ besetzt
- Heap-Invariante: $H[i] \leq \min\{H[2i+1], H[2i+2]\}$

Binärer Heap als Feld

insert(e)

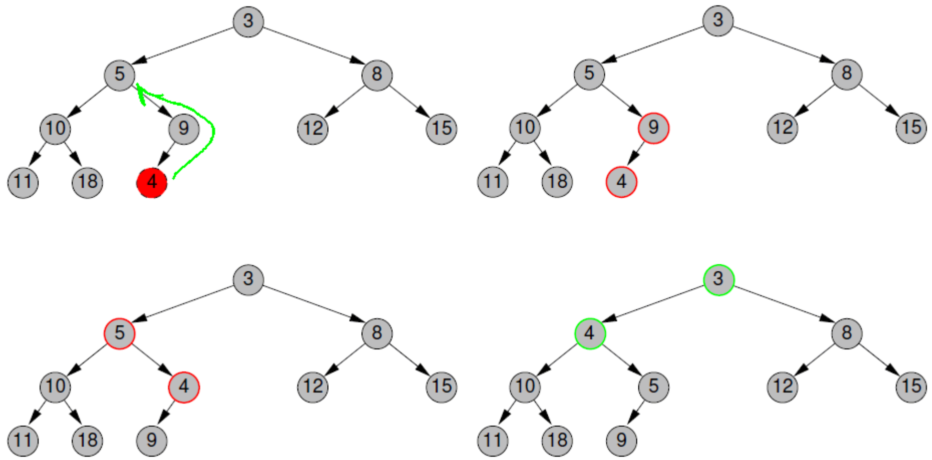
- Form-Invariante: $H[n] = e$; siftUp(n); $n{+}{+}$;
- Heap-Invariante:

  vertausche e mit seinem Vater bis
  $\text{prio}(H[\lfloor (k-1)/2 \rfloor]) \leq \text{prio}(e)$ für e in $H[k]$ (oder e in $H[0]$)

```
siftUp(i) {
   while (i > 0 ∧ prio(H[⌊(i−1)/2⌋]) > prio(H[i])) {
      swap(H[i], H[⌊(i−1)/2⌋]);
      i = (i−1)/2;
   }
}
```

- Laufzeit: $O(\log n)$

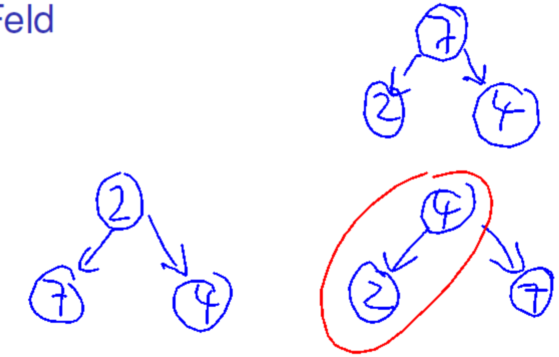
Heap - siftUp()



Binärer Heap als Feld

deleteMin()

- Form-Invariante:
  $e = H[0]$;
  $n - -$;
  $H[0] = H[n]$;
  siftDown(0);
  return e;

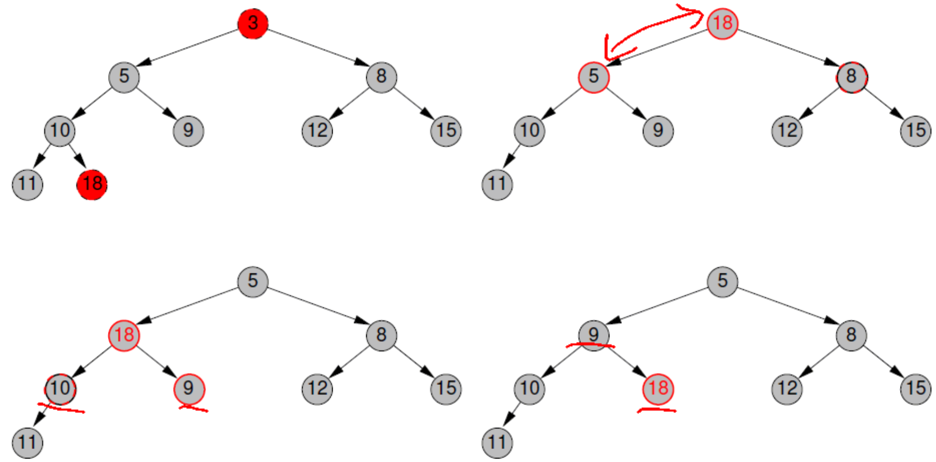


- Heap-Invariante: (siftDown)
  vertausche e (anfangs Element in $H[0]$) mit dem Kind, das die kleinere Priorität hat, bis e ein Blatt ist oder
  $\text{prio}(e) \leq \min\{\text{prio}(c_1(e)), \text{prio}(c_2(e))\}$.
- Laufzeit: $\mathcal{O}(\log n)$

Binärer Heap als Feld

```
siftDown(i) {
   int m;
   while (2i + 1 < n) {
      if (2i + 2 ≥ n)
         m = 2i + 1;
      else
         if (prio(H[2i + 1]) < prio(H[2i + 2]))
            m = 2i + 1;
         else m = 2i + 2;
      if (prio(H[i]) ≤ prio(H[m]))
         return;
      swap(H[i], H[m]);
      i = m;
   }
}
```


Heap - siftDown()



Binärer Heap / Aufbau

build($\{e_0, \ldots, e_{n-1}\}$)

- naiv:

  Für alle $i \in \{0, \ldots, n-1\}$:
  insert(e_i)

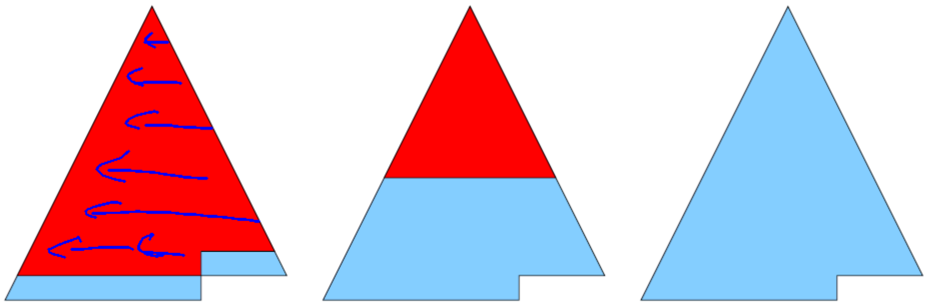
$\Rightarrow$ Laufzeit: $\Theta(n \log n)$

Binärer Heap / Aufbau

build($\{e_0, \ldots, e_{n-1}\}$)

effizient:

- Für alle $i \in \{0, \ldots, n-1\}$:
  $H[i] := e_i$.
- Für alle $i \in \left\{\left\lfloor \frac{n}{2} \right\rfloor - 1, \ldots, 0\right\}$:
  siftDown(i)



Binärer Heap / Aufbau

Laufzeit:

- $k = \lfloor \log n \rfloor$: Baumtiefe (gemessen in Kanten)
- siftDown-Kosten von Level ℓ aus proportional zur Resttiefe $(k - \ell)$
- Es gibt $\leq 2^\ell$ Knoten in Tiefe ℓ.

$$O\left(\sum_{0 \leq \ell < k} 2^\ell (k - \ell)\right) \subseteq O\left(2^k \sum_{0 \leq \ell < k} \frac{k - \ell}{2^{k-\ell}}\right) \subseteq O\left(2^k \sum_{j \geq 1} \frac{j}{2^j}\right) \subseteq O(n)$$

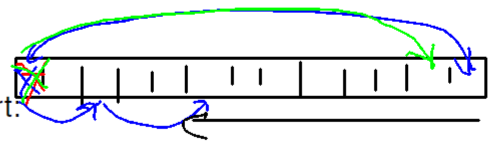
$$\begin{aligned}\sum_{j \geq 1} j \cdot 2^{-j} &= \sum_{j \geq 1} 2^{-j} + \sum_{j \geq 2} 2^{-j} + \sum_{j \geq 3} 2^{-j} + \ldots \\ &= 1 \cdot \sum_{j \geq 1} 2^{-j} + \frac{1}{2} \cdot \sum_{j \geq 1} 2^{-j} + \frac{1}{4} \cdot \sum_{j \geq 1} 2^{-j} + \ldots \\ &= (1 + 1/2 + 1/4 + \ldots) \sum_{j \geq 1} 2^{-j} = 2 \cdot 1 = 2\end{aligned}$$

Laufzeiten des Binären Heaps

- min(): $O(1)$
- insert(e): $O(\log n)$
- deleteMin(): $O(\log n)$
- build($e_0, \ldots, e_{n-1}$): $O(n)$
- M.merge(Q): $\Theta(n)$

Adressen bzw. Feldindizes in array-basierten Binärheaps können nicht als Handles verwendet werden, da die Elemente bei den Operationen verschoben werden
$\Rightarrow$ ungeeignet als adressierbare PQs (kein remove bzw. decreaseKey)

HeapSort



Verbesserung von SelectionSort:

- erst build($e_0, \ldots, e_{n-1}$): $O(n)$
- dann $n \times$ deleteMin():
  vertausche in jeder Runde erstes und letztes Heap-Element,
  dekrementiere Heap-Größe und führe siftDown(0) durch:
  $O(n \log n)$

$\Rightarrow$ sortiertes Array entsteht von hinten,
ansteigende Sortierung kann mit Max-Heap erzeugt werden

- in-place, aber nicht stabil
- Gesamtlaufzeit: $O(n \log n)$