AlgoDat: 3.Hausaufgabe (03.01.23) - Cora Zeitler

Donnerstag, 27. April 2023 08:09

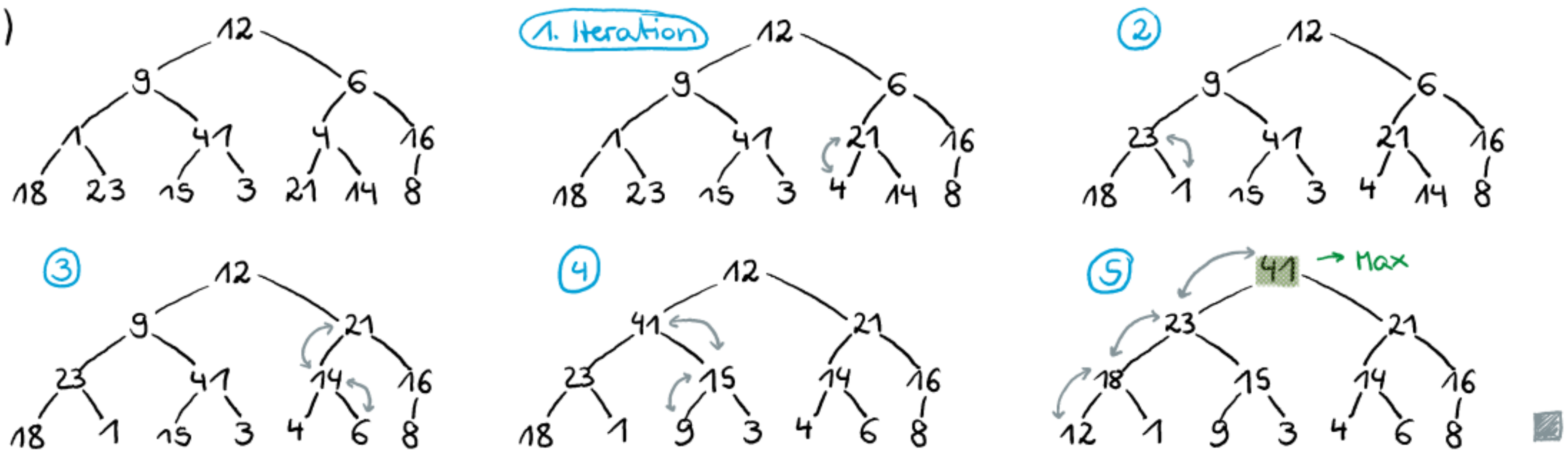
Σ: 12/24

Aufgabe 2:

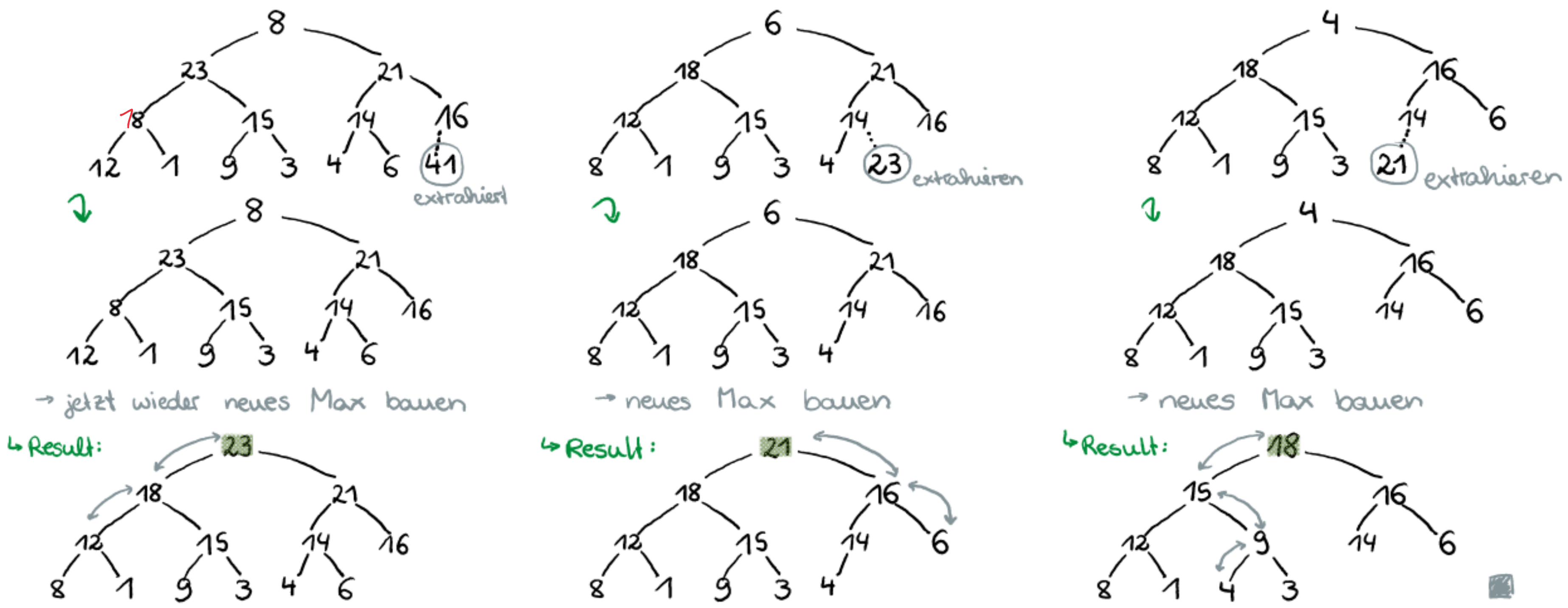
(a) Wenden Sie BUILD-MAX-HEAP(A) auf das Array
$A = (12, 9, 6, 1, 41, 4, 16, 18, 23, 15, 3, 21, 14, 8)$ an.
(Bitte Baumdarstellung zur Veranschaulichung.)

(b) Führen Sie die Operation HEAP-EXTRACT-MAX(A) drei mal aus. (8 Punkte)

2a)



2b) ① Output 41: → 41 mit 8 getauscht → 41 extrahieren
② Output 23: → 23 mit 6 getauscht → 23 extrahieren
③ Output 21: → 21 mit 4 getauscht → 21 extrahieren



8/8

Aufgabe 3:
Betrachten Sie folgende Funktion zum Aufbau eines Heaps:

```
BH'(A)
  heap_size[A] = 1
  for i=2 to length(A) do    // O(n)          } O(n log n)
    HEAP_INSERT (A, A[i])    // O(log n) ,nach VL
```

// zum Vergleich

algorithm BUILDMAXHEAP(A)
heap_größe[A] = länge[A]
for $i = \lfloor \frac{\text{länge}[A]}{2} \rfloor$ down to 1 do
MAXHEAPIFY(A, i)
end for
end algorithm // siehe Aufg. 2a

Beantworten Sie folgende Fragen und begründen Sie Ihre Antwort:

(a) Erzeugen BUILD-MAX-HEAP(A) und BH'(A) für jedes Feld A den gleichen Heap?

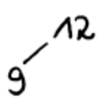
(b) Zeigen Sie, dass die asymptotische Laufzeit von BH'(A) im schlechtesten Fall $\Omega(n \log n)$ beträgt.

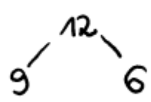
(8 Punkte)

3a) $A = \{12, 9, 6, 1, 41, 4, 16, 18, 23, 15, 3, 21, 14, 8\}$

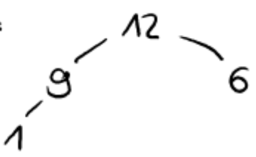
↳ mit BH: → 1. Initialisiere die Heap Größe von A auf 1
2. Iteriere über die Elemente von A von i = 2 bis zur Länge von A
3. Füge jedes Element A[i] in den Heap ein, indem die HEAP_INSERT Funktion aufgerufen wird

⓪ - Heap Größe von A auf 1 gesetzt
- wir beginnen mit 2. Element von A, also 9

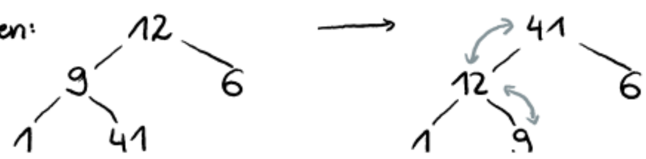
① 
// - 9 wird am Ende des Heaps eingefügt
- es gibt ein Element im Heap, also bleibt Heap-Eigenschaft erhalten

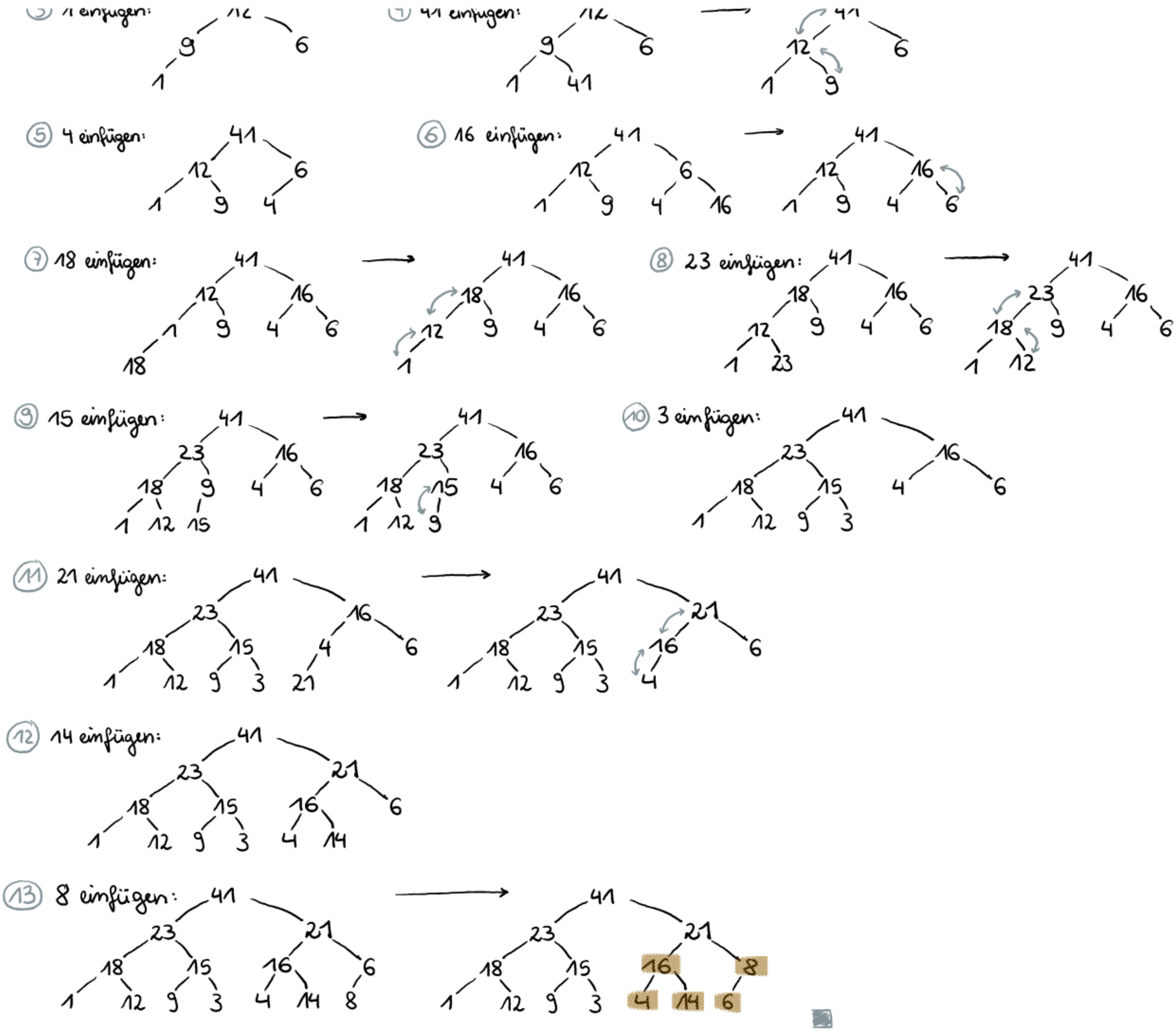
② 
// - Iterationsvorgang wird fortgesetzt
→ wenn Heap-Eigenschaft nicht erfüllt → Baum anpassen (Build-Max-Heap) f

③ 1 einfügen:



④ 41 einfügen:





→ im Vergleich zu 2a sind die Heaps unterschiedlich
→ Somit erzeugen beide Algorithmen nicht den gleichen Heap

3b) Mit Worst-Case prüfen:

Im schlechtesten Fall ist das Eingabefeld A vollständig unsortiert und hat keine besonderen Eigenschaften. In diesem Fall muss HEAP_INSERT für jedes Element in A aufgerufen werden, um den Max-Heap aufzubauen. Na, ja Heap_Insert wird jedes Mal aufgerufen

In welchem Fall muss wirklich jedes Element nach oben getauscht werden?

Jeder Aufruf von HEAP_INSERT hat eine Laufzeit von O(log n), da das Element von der Blattebene nach oben verschoben werden muss, um die Max-Heap-Eigenschaft wiederherzustellen. Da wir für jedes Element in A HEAP_INSERT aufrufen, beträgt die Gesamtlaufzeit von BH'(A) im schlechtesten Fall:

$$T(n) = O(\log 2) + O(\log 3) + \ldots + O(\log n)$$

Da $\log k \leq \log n$ für $k \leq n$ gilt, können wir die obere Schranke der Laufzeit abschätzen:

$$T(n) \leq O(\log n) + O(\log n) + \ldots + O(\log n) = O(n \log n)$$

$O \neq \Omega$?

Daraus folgt, dass die asymptotische Laufzeit von BH'(A) im schlechtesten Fall $\Omega(n \log n)$ beträgt, da die tatsächliche Laufzeit mindestens proportional zu $n \log n$ ist.

4/8

Aufgabe 1:
Merge-Sort kann beschleunigt werden, indem man für kurze Eingaben auf einen einfachen Sortieralgorithmus Simple-Sort umschaltet. Für unser Beispiel betrage die Laufzeit bei einer Eingabe von $n = r - p + 1$ Elementen:

- $\frac{n(n-1)}{2}$ für SIMPLE-SORT(A,p,r) und
- $8n$ für die Prozedur MERGE(A,p,q,r).

Der Zeitbedarf aller übrigen Teile des Algorithmus soll zur Vereinfachung vernachlässigt werden. Der modifizierte Merge-Sort Algorithmus hat dann die Form:

```
MERGE-SORT(A,p,r)                 // wenn n ≤ L, dann Simplesort
if $(r-p+1<= L)$ then
    SIMPLE-SORT(A,p,r)
else                              // wenn n > L, dann Mergesort
    q:=(p+r/2)
    MERGE-SORT(A,p,q)
    MERGE-SORT(A,q+1,r)
    MERGE(A,p,q,r)
```

(a) Welcher Wert ist optimal für L?

(b) Wie groß ist der zeitliche Gewinn gegenüber L=1 in Abhängigkeit von der Eingabelänge n. Dabei seien für n nur Zweierpotenzen zugelassen.

(8 Punkte)

1a) geg: $f(n) = 8n$, $g(n) = \frac{n\cdot(n-1)}{2}$

→ ab welchen Wert L ist $f(n)$ größer als $g(n)$?

$$\begin{aligned} 8n &= \frac{n\cdot(n-1)}{2} & &|\cdot 2 \\ 16n &= n\cdot(n-1) & &|:n \\ 16 &= n-1 & &|+1 \\ 17 &= n \end{aligned}$$

Es geht nicht darum Simple-Sort mit dem ursprünglichen Merge-Sort Algorithmus zu vergleichen.

↳ optimalste Wert ist 17 (bei 17 sind beide Fkt. gleichschnell) f

1b) $n = 2$: $8\cdot 2 = \frac{2\cdot(2-1)}{2}$

$16 = \frac{4-2}{2}$

$16 = 1$ Warum = ? $16 \neq 1$

$n = 4$: $8\cdot 4 = \frac{4\cdot(4-1)}{2}$

$32 = \frac{16-4}{2}$

$32 = 6$

$n = 8$: $8\cdot 8 = \frac{8\cdot(8-1)}{2}$

$64 = \frac{64-8}{2}$

$64 = 18$

$n = 16$: $8\cdot 16 = \frac{16\cdot(16-1)}{2}$

$128 = \frac{256-16}{2}$

$128 = \frac{240}{2}$

$128 = 120$

$n = 32$: $8\cdot 32 = \frac{32\cdot(32-1)}{2}$

$256 = \frac{1024-32}{2}$

$256 = 496$

→ Die Laufzeit des ursprünglichen Algorithm. mit $L = 1$ beträgt: $8n$ und $\frac{n\cdot(n-1)}{2}$ f

→ Daher ist der zeitliche Gewinn gegenüber $L=1$ abhängig von der Eingabelänge n wie folgt:

↳ Wenn $n \leq L$, ist zeitlicher Gewinn 1? Warum?

↳ Wenn $n > L$, ist der zeitliche Gewinn für modif. Algorithmus:

→ für $L = 32$ → siehe oben

→ für $L = 64$: $8\cdot 64 = \frac{64\cdot(64-1)}{2}$

$512 = \frac{4.096-64}{2}$

$512 = 2016$

Für jedes $n > L$ die exakt gleiche Ersparnis?

⇒ Daher zeigt sich das der Zeitliche Gewinn durch die Verwendung des modifiz. Merge-Sort mit einem optimalen Schwellen $L = 32$ oder $L = 64$ für große Eingaben erheblich sein kann

0/8