A Binäre Heaps (1 Punkt)

Gegeben sind die Ziffern A = {9,5,3,4}. Geben Sie alle möglichen binären Heaps aus A an. Stellen Sie dabei die Heaps als implizites Feld dar.

Musterlösung:

Folgende binäre Heaps sind möglich:

1	2	3	4
3	4	5	9

1.Möglichkeit

1	2	3	4
3	5	4	9

2.Möglichkeit

1	2	3	4
3	4	9	5

3.Möglichkeit

Aufgabe B *d*-näre Heaps

Bis jetzt kennen Sie binäre Heaps und deren implizite Repräsentation in einem Array. Entwickeln Sie nun einen d-nären Heap, in welchem statt maximal 2 Kindern jedes Element nun maximal d Kinder hat. Der Heap soll lückenlos in einem Array gespeichert werden.

Aufgabe B.1 Formel (3 Punkte)

Geben Sie für das Element mit Arrayindex j die Formeln zum Berechnen des Elterelements und der d Kindelemente an (vom 1. Kind bis zum d. Kind). Begründen Sie ihre Formeln.

Aufgabe B.2 Implementierung (3 Punkte)

Implementieren Sie *deleteMin* und *siftDown* für den d-nären Heap.

Lösung

Aufgabe B.1

- ***Behauptung:***
  - Sei $child(i, j), j \in \{1, \dots d\}$ das j-te Kind von i, sowie $parent(i)$ das Elternelement von i
  - Startet die Indizierung des Arrays bei 0, so gilt: $child(i, j) = d \cdot i + j \wedge parent(i) = \lfloor \frac{i-1}{d} \rfloor$
  - Startet die Indizierung bei 1, so gilt: $child(i, j) = d \cdot (i-1) + j + 1 \wedge parent(i) = \lceil \frac{i-1}{d} \rceil$
- ***Beweis von child (vollst. Ind über i)*** (für Indizierung bei 0 (bei 1 geht analog):
  - *Basis*:
    $child(0, j) = j = d \cdot 0 + j$✓
  - *Induktionshypothese*: $\forall i, j : child(i, j) = d \cdot i + j$
  - *Induktionsschritt*:
    Da das letzte Kind von i nach IH $d \cdot i + d$ ist (und somit die nächsten j Elemente $d \cdot i + d + j$ sind) folgt:
    $child(i + 1, j) = d \cdot i + d + j = d \cdot (i + 1) + j$✓
- Nun ist aber $parent(child(i, j)) = \lfloor \frac{d \cdot i + j - 1}{d} \rfloor = \lfloor i + \frac{j-1}{d} \rfloor = i$, da $j - 1 < d$, und damit stimmen beide Formeln ✓

Aufgabe B.2

Procedure: siftDown($i : \mathbb{N}_0$)

if $child(i, 1) \leq n$ **then**
  $k := \max\{t \in \{1, \dots d\} \mid child(i, t) \leq n\}$;
  $l :=$argmin$_{j \in \{1,\dots k\}}(child(i, j))$;
  $m := child(i, l)$;
  if $h[i] > h[m]$ **then**
    swap$(h[i], h[m])$;
    siftDown(m) ;
  end
end

Function: deleteMin(): Element

result := $h[1]$: Element ;
$h[1] := h[n]$;
$n := n - 1$;
siftDown(1) ;
return *result* ;

Übungsblatt 7 Aufgabe C

Jean-Pierre von der Heydt

13. Juni 2018

Gegeben sei ein sortiertes Array A der Länge n von ganzen Zahlen, in dem Elemente mehrfach vorkommen können. Implementieren Sie zwei unterschiedliche Varianten eines Algorithmus, welcher die Positionen des ersten und letzten Auftretens eines Elements in A ermittelt. Argumentieren Sie für die Varianten C.1 und C.2 jeweils warum Ihr Algorithmus das gewünschte Laufzeitverhalten aufweist.

C.1 Liegestuhl-Variante (1,5 Punkte)

Geben Sie einen Algorithmus an, der in $\mathcal{O}(\log n)$ Schritten die Positionen des ersten und des letzten Auftretens eines Elementes e in A ermittelt.

C.2 Chefsessel-Variante (1,5 Punkte)

Geben Sie einen Algorithmus an, der in $\mathcal{O}(n)$ Schritten die Positionen des ersten und des letzten Auftretens eines Elementes e in A ermittelt. Sei die Anzahl der Vorkommen einzelner Elemente nach oben durch k beschränkt, dann soll dieser Algorithmus für $k < \log n$ weniger Vergleiche benötigen als der Algorithmus in C.1.

1 Lösung

Wir werden für beide unserer Algorithmen die Funktion `locate(A[1..n], k)` verwenden. Diese ist auf der 6. Vorlesungsfolie vom Foliensatz zum 30.5.18 zu finden. Sie gibt uns für ein sortiertes Array A den Index des ersten Elements $\geq k$ zurück.

Lösung C.1

Algorithm 1: Liegestuhl-Variante

Data: $A[1...n]$: sorted Array, $e : \mathbb{N}$

```
1 /* Assert that e ∈ A                                          */
2 return (locate(A, e), locate(A, e + 1) − 1)
```

Dieser Algorithmus erfüllt offensichtlich das gewünschte Laufzeitverhalten, da er lediglich zwei mal die Funktion `locate` aufruft, welche in $\mathcal{O}(\log n)$ liegt.

Lösung C.2

Algorithm 2: Chefsessel-Variante

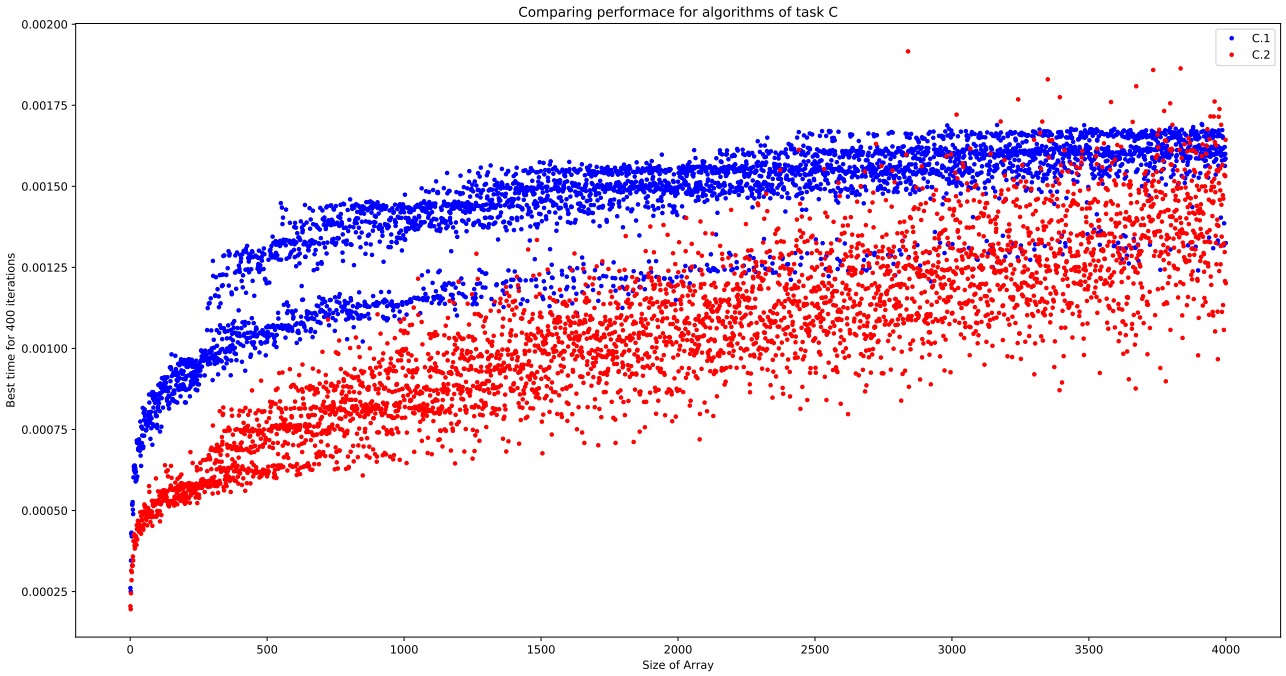
Data: $A[1...n]$: sorted Array, $e : \mathbb{N}$

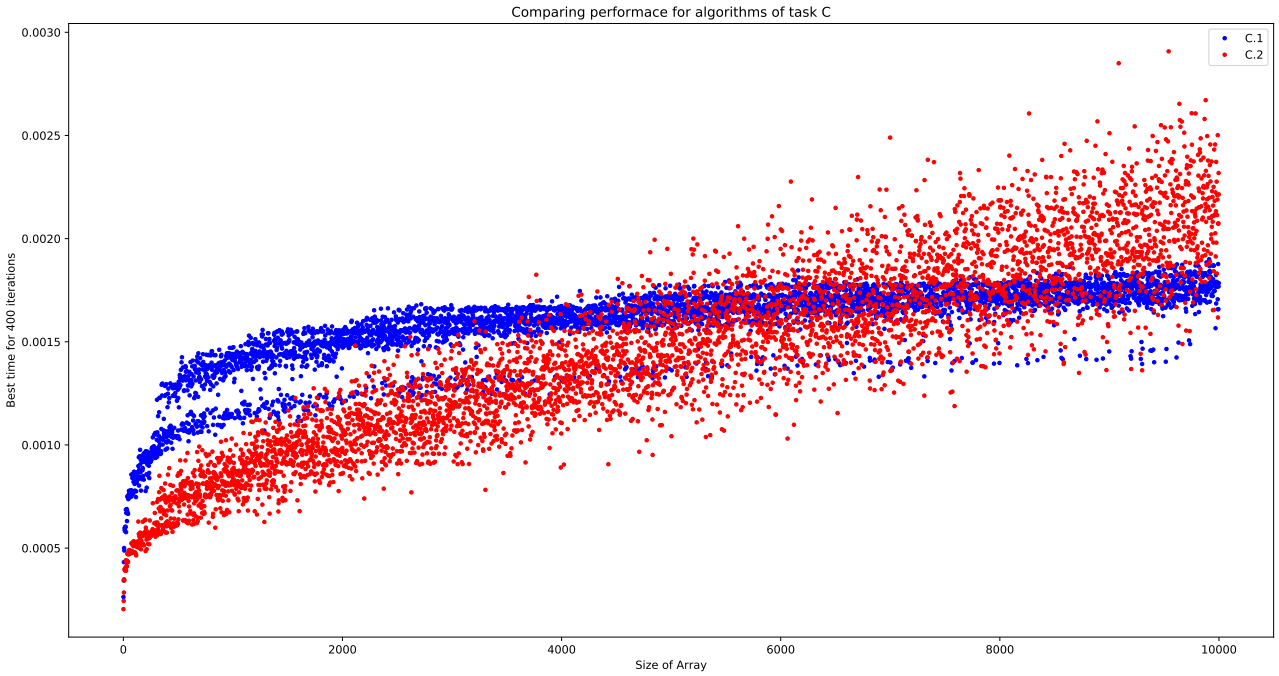
```
1 /* Assert that e ∈ A                                          */
2 s := locate(A, e)
3 t := s + 1
4 while A[t] = e do                          // Assuming A[n+1]=∞
5     t := t + 1
6 return (s, t − 1)
```

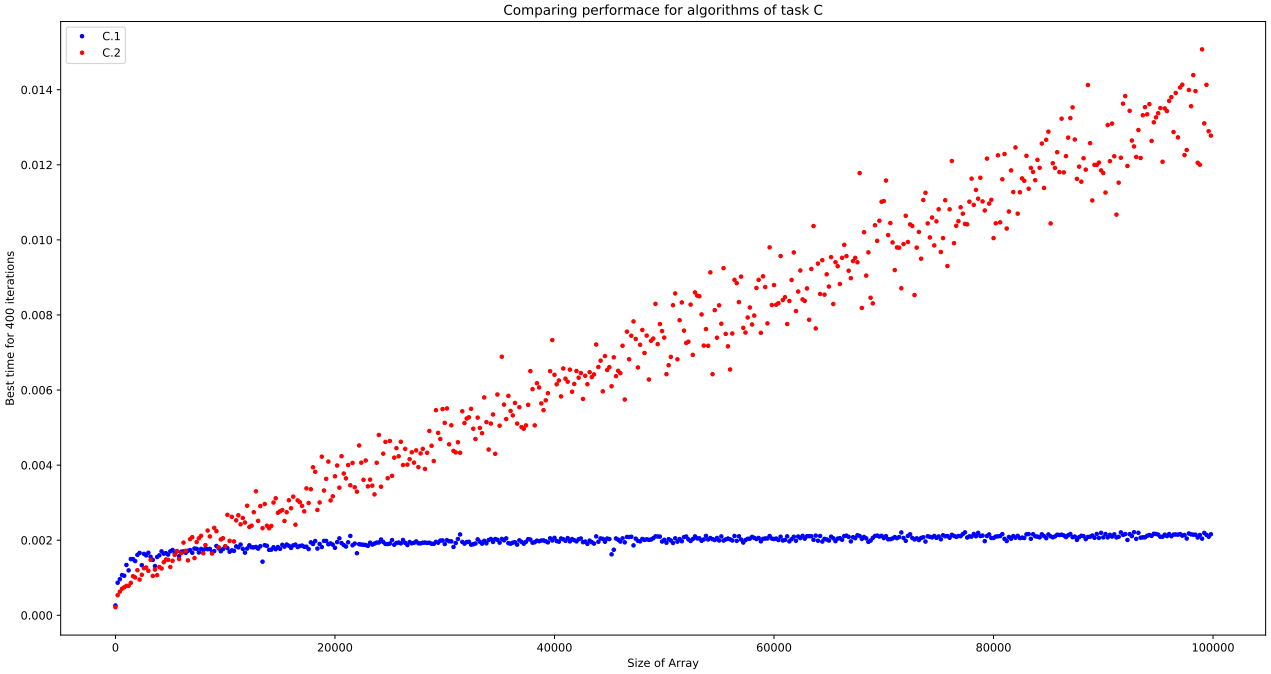
Im schlimmsten Fall durchläuft der Algorithmus das gesamt Array ($k = n$) und hat damit eine Laufzeit von $\mathcal{O}(\log n + k) = \mathcal{O}(n)$. Für $k < \log n$ muss unser Algorithmus weniger Vergleiche Ausführen als der Algorithmus in 1.

2 Zusatz

Im Folgenden werden die Laufzeiten der beiden Algorithmen verglichen. Dazu wurde ein Python Skript erstellt, welches den Pseudocode möglichst nah simuliert.
Die Algorithmen werde 300 mal auf einem Array der Größe n ausgeführt und die Laufzeit gestoppt. Um Störfaktoren zu minimieren wird die Laufzeit (der 300 Wiederholungen) für beide Algorithmen und jedes Array 30 mal berechnet und anschließend das Minimum weiterverwendet. Die Anzahl der unterschiedlichen Elemente in jedem Array beträgt konstant 400.
Dem geübten Auge ist es überlassen die vorherigen/folgenden Grafiken zu interpretieren.


Comparing performace for algorithms of task C
C.1
C.2
Best time for 400 iterations
0.00200
0.00175
0.00150
0.00125
0.00100
0.00075
0.00050
0.00025
0
500
1000
1500
2000
2500
3000
3500
4000
Size of Array


Comparing performace for algorithms of task C
C.1
C.2
Best time for 400 iterations
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0
2000
4000
6000
8000
10000
Size of Array


Comparing performace for algorithms of task C
C.1
C.2
Best time for 400 iterations
0.000
0.002
0.004
0.006
0.008
0.010
0.012
0.014
0
20000
40000
60000
80000
100000
Size of Array