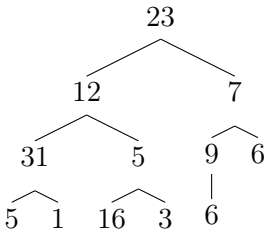
# Grundzüge DS & Alg (WS14/15)

Lösungsvorschlag zu Aufgabenblatt 3

## Aufgabe 1

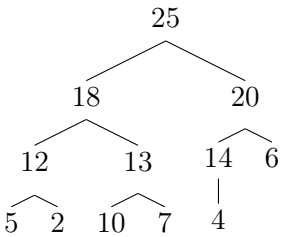
(a)

nicht-Heap



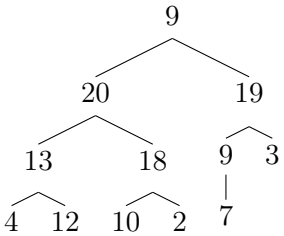

(b)

Heap



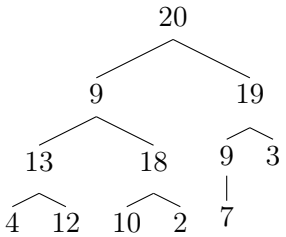

(c)

Beinahe-Heap

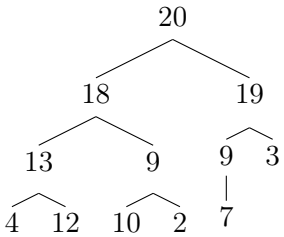



Heapify

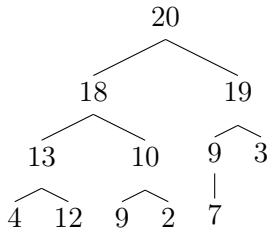
1. Iteration




2. Iteration




3. Iteration



# Grundzüge DS & Alg (WS14/15)

Lösungsvorschlag zu Aufgabenblatt 3

## Aufgabe 2

### Erwartete Lösung in $h + \lceil \log_2 h \rceil$

Wir bauen den "Pfad der Größten", wobei für jedes Element garantiert wird: Wenn es einen Sibling hat, so ist es größer als dieser, und wenn es ein Kindknoten hat, so ist es nicht das Ende des Pfades. Wenn wir den Pfad schon hätten, müssten wir nur noch heraus finden, *wo* in dieser Liste wir einfügen müssen, und sind prinzipiell fertig. Also:

```
procedure Sift-Down(H)
      path ← PathOfBiggest(H)
      position ← PositionInSorted(H[1], path[2...h], H)
      InsertPushUp(H[0], path[position], H)

// Return the path of the biggest elements in the heap
PathOfBiggest(H):
      path ← new array of size h
      path[1] = 1 // Der Pfad beginnt immer bei der Wurzel
      parent = 1 // Der ‘‘aktuelle Parent’’ ist die Wurzel
      for i from 2 through h do
            if RightChild(parent) < LeftChild(parent)
                  nextParent ← LeftChild(parent)
            else
                  nextParent ← RightChild(parent)

            path[i] = nextParent // Pfad führt durch das größere Kind
            parent = nextParent
      return path

// Find where value would be inserted in given path
// Can return any value from a − 1 through b
PositionInSorted(value, path[a...b], H)
      if a > b
            return b
      m ← ⌊(a+b)/2⌋
      // Heap ist von groß nach klein, also der Pfad auch
      if value > H[path[m]]
            return PositionInSorted(value, path[a...m], H)
      else
            return PositionInSorted(value, path[m + 1...b], H)

// Insert at the specified position, pushing everything
// upwards in the heap, popping out the root
```

```
InsertPushUp(what, where, H)
      popped ← H[where]
      H[where = what
      if where != 0
            InsertPushUp(popped, Parent(where), H)
```

Dieser Ansatz braucht für den Fall $h = 2$ maximal 4 Vergleiche, für $h = 3$ maximal 6.

Die Komplexität der Vergleiche bei diesem Ansatz beträgt $h + \lceil \log_2 h \rceil <_{\text{fü}} 2h$.

### Etwas schnellere Lösung in $h + \log^*(h) + O(1) <_{\text{fü}} h + \lceil \log_2 h \rceil$

Anstatt den gesamten Pfad von Anfang an zu berechnen, können auch die ersten $h - \log_2(h)$ Elemente des Pfades berechnet werden, gefolgt von einem Vergleich mit dem letzten Element:

$\in$ Pfad: Dann sind nur noch $\log_2(h - \log_2(h))$ Vergleiche (zur Binärsuche) notwendig. Die Vergleiche zur Festlegung des restlichen Pfades entfallen.

$\notin$ Pfad: In diesem Fall kann der Algorithmus rekursiv absteigen, und das Ende des Pfades als neue Wurzel betrachten. Selbst wenn hier der obige "langsame Algorithmus verwendet wird, wurden einige Vergleiche gespart.

Dieses Verhalten führt zu folgender Rekurrenz-Gleichung bei den Kosten:

$$C(h) \leq h - \log_2(h) + 1 + \max\{\log_2(h - \log_2(h)), C(\log_2(h))\}$$

Wir definieren [1] $\log^\star(h) = 0$, wenn $h \leq 1$, und $\log^\star(h) = 1 + \log^\star(\log(h))$, wenn $h > 1$. Dann $C(h) = h + \log^*(h) + O(1)$

Diese Optimierung ist aber eher marginal. Kurz ausgerechnet: Für 8000000 (und $C(0) = 0$) ergibt diese Rekurrenz 8000005. Der obige "langsame" Ansatz braucht 8000023. Heaps dieser Höhe (also mit $2^{8\text{Mio.}}$ Elementen) haben da ganz andere Probleme. Optimierungen werden an anderen Stellen eher gebraucht.

### Zu schnelle Lösungen

Sublineare Ansätze sind grundsätzlich falsch: Damit ließe sich ein Heapsort in $o(n \log n)$ basteln, Widerspruch.

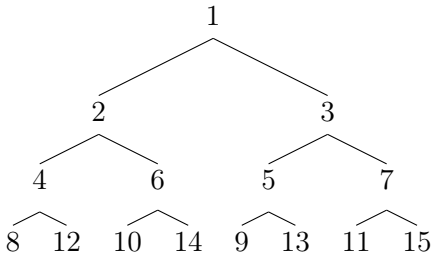
[1] `http://en.wikipedia.org/wiki/Iterated_logarithm`

# Grundzüge DS & Alg (WS14/15)

Lösungsvorschlag zu Aufgabenblatt 3

## Aufgabe 3

(a)




(b)

Wir zeigen dies durch Induktion über die Höhe des Baumes:

IA: Für einen Baum der Höhe 2 ist die Bedingung offensichtlich erfüllt (siehe Skizze aus a).

IV: Für ein Baum der Höhe $h$ gilt: jeder Position im Array $1 \dots n$ mit $n < 2^h$ ist genau eine Position im Baum zugeordnet.

IS: $h \longrightarrow h+1$ :
Alle Positionen der Werte im Array sind auf Ebene $h$ des ergebenden Baumes nach IV eindeutig. Sie lassen sich als $v_{2^{h-1}}$ bis $v_{2^h-1}$ darstellen. Nach Definition von $t(x)$ ist damit
$\forall v : t(v) =$ Anzahl der Knoten auf Ebene von Element v und
$\forall v_i, v_j : t(v_i) = t(v_j)$ mit $i, j \in [2^{h-1}, 2^h - 1]$.
Sei $a = \log t(v_{2^{h-1}})$. Die Knoten der Höhe $h+1$ erhalten die Positionen $2^a$ bis $2^a + 2a$, somit können maximal $2a$ Knoten adressiert werden, was genau der maximalen Anzahl an Knoten in der Höhe $h+1$ entspricht.
Da jeder Knoten in der Höhe h eindeutig zugeordnet ist, und wir jeweils nur $a$ oder $2a$ aufaddieren, erhalten wir wieder eine eindeutige Zuordnung. Damit ist der entstehende Baum der Höhe $h+1$ ebenfalls ein gültiger, eindeutiger Baum.

(c)

parent$(v) =$ if $v - \frac{t(v)}{2} < t(v)$ then return $v - \frac{t(v)}{2}$ else return $v - t(v)$

(d)

wenn $v + t(v) > n$ gilt, dann ist $v$ Blatt des Baumes in $1 \dots n$

(e)

Ja kann man, es müssen lediglich ein paar Funktionen angepasst werden:

leftchild$(v) = v + t(v)$
rightchild$(v) = v + 2t(v)$
parent$(v) =$ siehe (c)

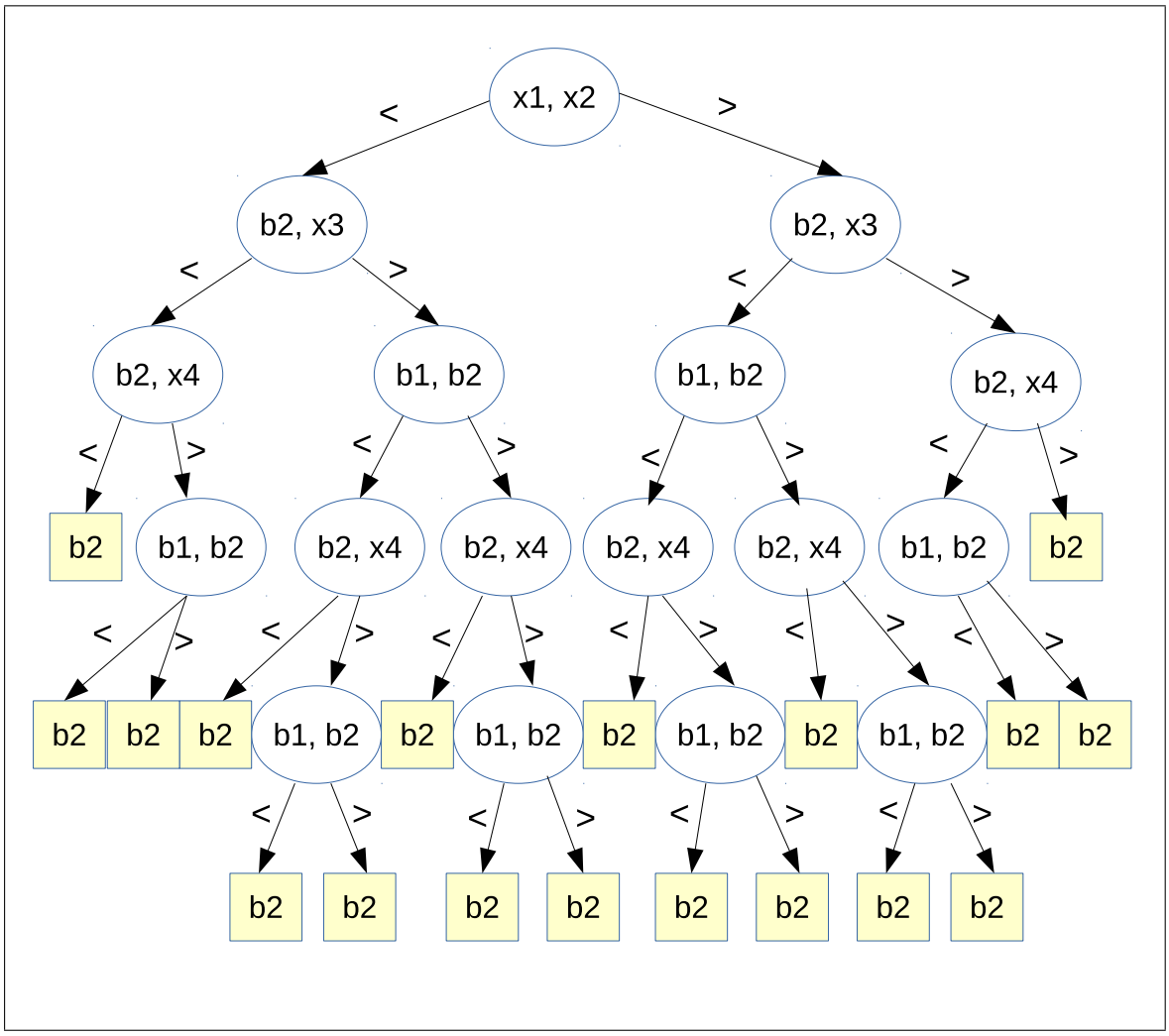

Als Vorteil ergibt sich, das zuerst alle rechtseitigen und danach erst die linkseitigen Blätter wegfallen. Dadurch gibt es mehr Knoten, die nur ein Kind haben. Somit entfällt in diesen Fällen der Verglich zwischen den Kindern eines Knotens, z.B. bei heapify. Nachteilig ist, dass die oben genannten Funktionen nun etwas komplexer sind.

## Aufgabe 4

### a)

```
1  Input: x1, x2, x3, x4
2  Output: Das zweitkleinste Element.
3
4  if (x1>x2)
5      swap(x1,x2);
6      // In x1 und x2 werden nun immer
7      // die zwei kleinsten bisher betrachteten
8      // Elemente eingetragen.
9
10
11 if (x3<x2){
12     x2 = x3;
13     // Hier wird x3 ggf. einsortiert.
14     if (x2<x1)
15         swap(x1,x2);
16 }
17
18 if (x4<x2){
19     x2 = x4;
20     // Hier wird x4 ggf. einsortiert.
21     if (x2<x1)
22        swap(x1,x2);
23 }
24
25
26 // Da in x2 immer der zweitkleinste
27 // betrachtete Wert eingetragen ist,
28 // und da alle Werte schon betrachtet wurden,
29 // ist x2 der Rueckgabewert.
30 return x2;
```

### b)




Mit b1/b2 ist das Element gemeint, das an der Stelle von x1/x2 zu diesem Zeitpunkt steht. Die Vertauschungen sind wie in Aufgabenteil a) vorgenommen.

### c)

Das Programm macht im schlechtesten Fall 5 Vergleiche. Das Programm macht im besten Fall 3 Vergleiche. Unter der Annahme, dass die Elemente zufällig Verteilt sind, ist die Durchschnittliche Anzahl der vergleiche $4 + \frac{1}{6}$.

Es gibt auch Algorithmen, bei denen in jedem Fall 4 Vergleiche durchgeführt werden.


## d)

Nein. Man muss auch nichts ändern. Nur im Entscheidungsbaum muss in einem der beiden Fälle ein Kleiner- bzw. Größergelichzeichen eingefügt werden.