4.1a) (1) Zuerst wird eine Breitensuche auf dem Knoten a ausgeführt.

(2) Aus dem entstandem Baum wird der kürzeste Weg zum Knoten c rausgeschrieben

(3) Nun wird erneut eine Breitensuche ausgeführt, auf dem Knoten c.

(4) Aus dem entstandem Baum wird der kürzeste Weg zum Knoten b rausgeschrieben

(5) Schreibe beide Wege zusammen und gebe ihn aus.

Die Laufzeit ist $O(\underbrace{2 \cdot (|V|+|E|)}_{\text{2 mal Breiten-suche}} + \underbrace{c}_{\text{2 Strings zusammenschreiben}}) \hat{=} O(|V|+|E|)$

b) [redacted]

Erstelle einen Hilfsknoten, der mit allen Knoten $s \in S$ direkt verbunden ist. Von dort aus wird eine Breitensuche gestartet, wobei offensichtlicher Weise die komplette zweite Ebene alles ausschliesslich Knoten $s \in S$ sind.

Sobald die Breitensuche den ersten Knoten $z \in Z$ erreicht, wird die Breitensuche abgebrochen und der Weg zu diesem Knoten ausgegeben, jedoch ohne dem erstellten Hilfsknoten am Anfang

-1p. Laufzeit? Dein Algo braucht etwas länger wenn |Z| = 1 !

4.4) Es sei H ein Min-Heap, der aus jeder der k Listen das kleinste Element enthält.

~~1/2P.~~ ~~(1) Sortiere H mit Heap-Sort (O(n·log(n)))~~

(2) Schreibe das erste Element (ist nun das kleinste) in die Ergebnisliste L und lösche es aus H

(3) Füge in H das nächst-größere Element aus der Liste ein, aus der zuletzt ein Element in L geschrieben wurde wie verwaltet du das?

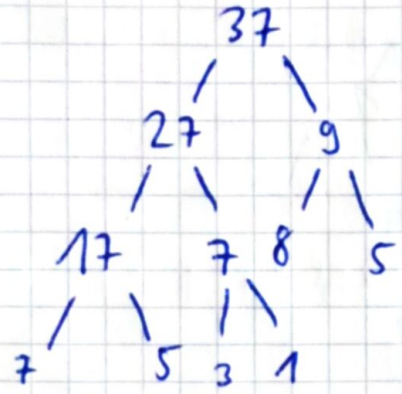
-1P. (4) Wiederhole solange (1) bis (3), bis alle Listen leer sind $\Rightarrow O(k \cdot n \cdot \log(n))$, da k-mal Heap-Sort verwendet wird.

Beispieldurchlauf:

$L_1 = [1, 4]$ $L_2 = [9, 10]$ $L_k = [7, 11]$

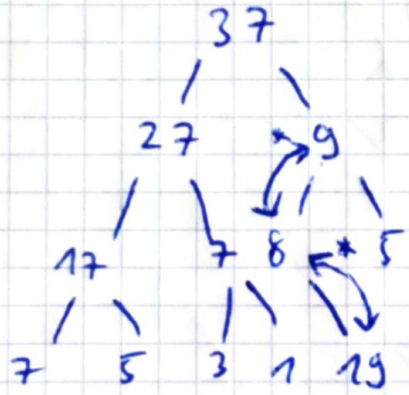
Start	$H = [1, 9, 7]$	$L = []$
(1)	$H = [1, 7, 9]$	$L = []$
(2)	$H = [7, 9]$	$L = [1]$
(3)	$H = [7, 9, 4]$	$L = [1]$
(1)	$H = [4, 7, 9]$	$L = [1]$
(2)	$H = [7, 9]$	$L = [1, 4]$
(3)	$H = [7, 9]$	$L = [1, 4]$
(1)	$H = [9]$	$L = [1, 4, 7]$
(2)	$H = [9, 10]$	$L = [1, 4, 7]$
(3)	$H = [9, 10]$	$L = [1, 4, 7]$
(1)	$H = [10]$	$L = [1, 4, 7, 9]$
(2)	$H = [10, 11]$	$L = [1, 4, 7, 9]$
(3)	$H = [10, 11]$	$L = [1, 4, 7, 9]$
(1)	$H = [11]$	$L = [1, 4, 7, 9, 10]$
⋮		

4.2a) Heap H=[-|37|27|9|17|7|8|5|7|5|3|1]



Insert (19)

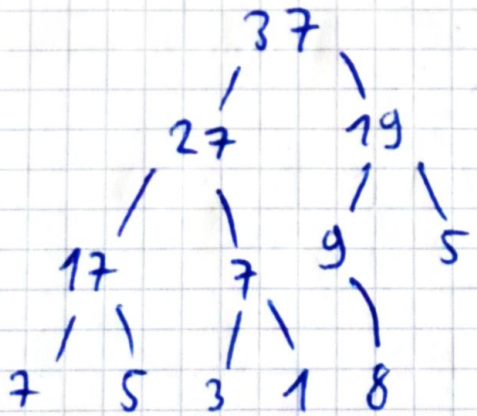
H=[-|37|27|9|17|7|8|5|7|5|3|1|19]



* Die Pfeile signalisieren notwendige Reparaturen!

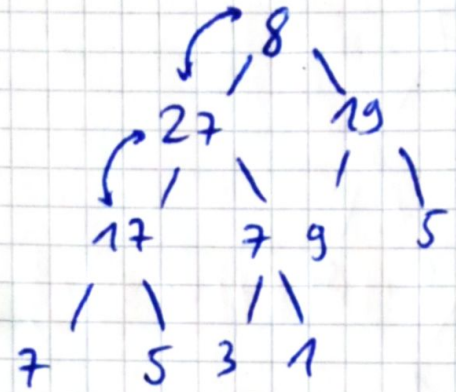
repairUp()

H=[37|27|19|17|7|9|5|7|5|3|1|8]



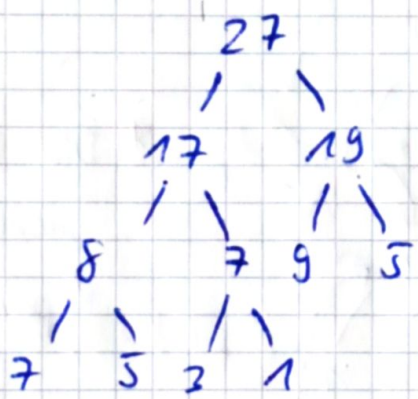
delete_max()

H=[-|8|27|19|17|7|9|5|7|5|3|1]



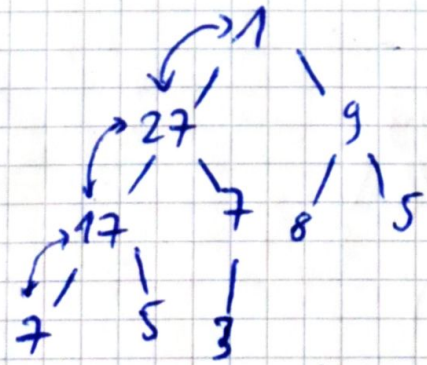
repairDown()

H=[-|27|17|19|8|7|9|5|7|5|3|1]



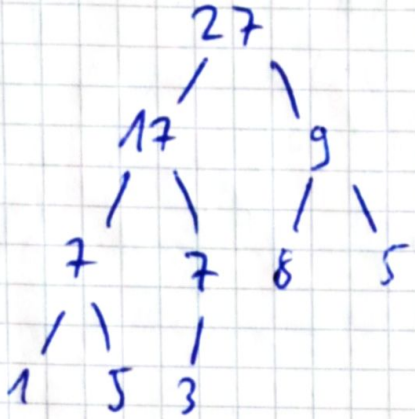
b) delete_max()

H=[-|11|27|9|17|7|8|5|7|5|3]



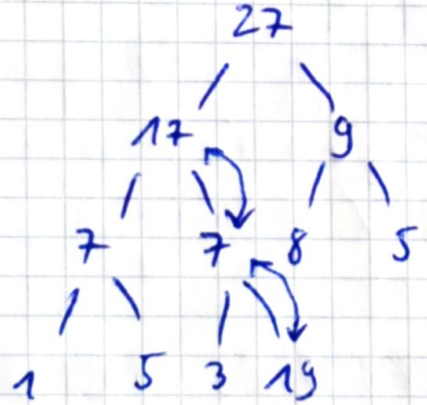
repairDown()

H=[-|27|17|9|7|7|8|5|1|5|3]



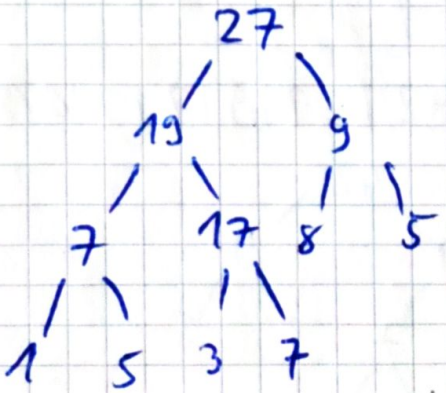
insert (19)

H=[-|27|17|9|7|7|8|5|1|5|3|19]



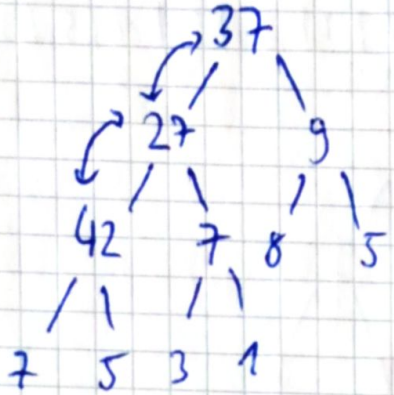
repairUp()

H=[-|27|19|9|7|17|8|5|1|5|3|7]



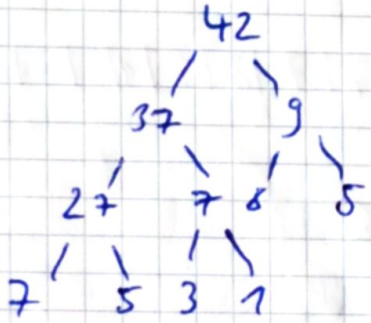
c) change-priority (4, 42)

H = [- | 37 | 27 | 9 | 42 | 7 | 8 | 5 | 7 | 5 | 3 | 1]



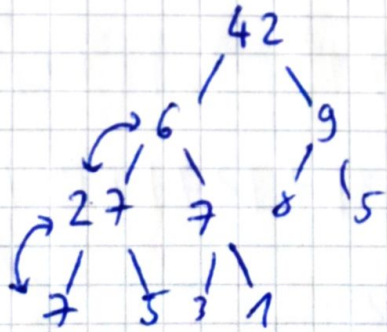
repairUp ()

H = [- | 42 | 37 | 9 | 27 | 7 | 8 | 5 | 7 | 5 | 3 | 1]

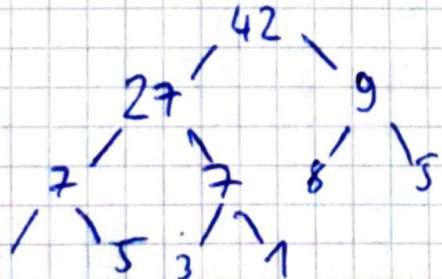


change-priority (2, 6)

[- | 42 | 6 | 9 | 27 | 7 | 8 | 5 | 7 | 5 | 3 | 1]



repairUp () H = [- | 42 | 27 | 9 | 7 | 7 | 8 | 5 | 6 | 5 | 3 | 1]



4.3a)

```
repairUp (Knoten wo)        //Knoten ist ein Tupel aus
int p = H[snd(wo)]          // (i, p_i), gemäß Aufgabe
while ((snd(wo) > 1) && (H[snd(wo)/2] < p))
    { H[snd(wo)] = H[snd(wo)/2]
      int hilf = A[fst(wo)]                  //Modifizierung:
      A[fst(wo)] = A[fst(wo/2)]              // Die Einträge im
      A[fst(wo/2)] = hilf                    // Array A werden
      snd(wo) = snd(wo/2)                    // an Stelle fst(wo)
H[snd(wo)] = p                               // mit fst(wo/2)
                                             // vertauscht
```

1:1 aus dem Skript

```
repairDown (Knoten wo)
Knoten kind
int p = H[snd(wo)]
while (snd(wo) ≤ n/2)
    { kind = 2 * snd(wo)
      if ((kind) < n) && (H[(kind)] < H[(kind)+1]) kind++
      if (p ≥ H[snd(kind)]) break
      H[snd(wo)] = H[snd(kind)]
      int hilf = A[fst(wo/2)]                //Hier wird das
      A[fst(wo/2)] = A[fst(wo)]              // Array synchron
      A[fst(wo)] = hilf                      // gehalten
      snd(wo) = snd(kind)
    }
H[snd(wo)] = p
```

Skript

```
insert (Knoten knoten)
    A[fst(knoten)] = |A|+1;    repairUp();
delete_max()
    A[fst(H[1])] = -1
    A[fst(H[|H|])] = 1
    repairDown();
```

// Das sind nur die anzupassenden Zeilen Quellcode, damit die Einträge im Array A synchron zum Heap bleiben.

Unter Wahrung dieser Algorithmen sollte lookup so aussehen:

```
lookup(i)
    return A[i]
```

b) Idee: Max-Heaps haben immer das größte Element an der Wurzel. Es reicht also, die Wurzel zu betrachten, insofern der Heap sortiert ist.

Algorithmus: (1) Ist root $\geq p$? $\rightarrow$ (2) Nein $\rightarrow$ return.
(2) Gebe die Wurzel aus
(3) delete_max(); c) sollte aber enthalten sein...
(4) (repairDown();) Wiederhole (1)

Dauert $\log(n)$

Laufzeit: $O(|\{x \in H : x \geq p\}|)$, wenn man die Ausgaben betrachtet. $-1\frac{1}{2}$ p.