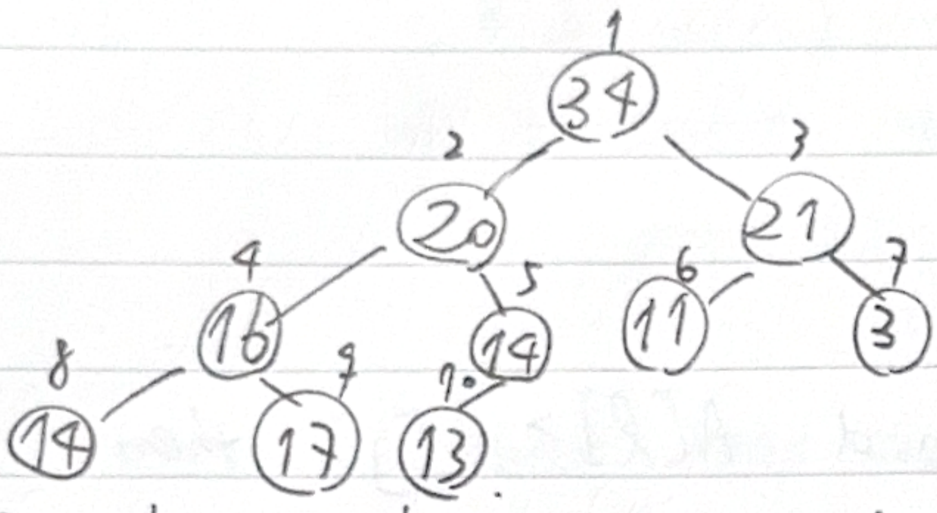
NO.

DATE

Data Structures and Algorithm Analysis (H1) Lab 04.

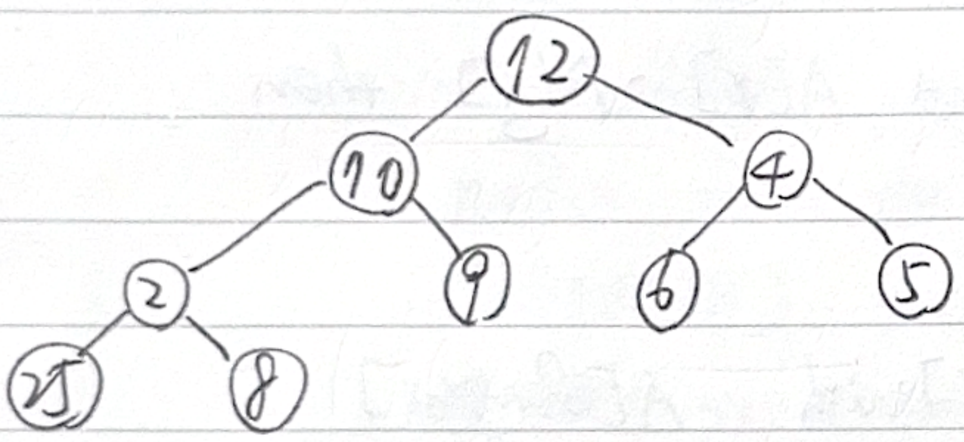
Q.4.1.

NO.

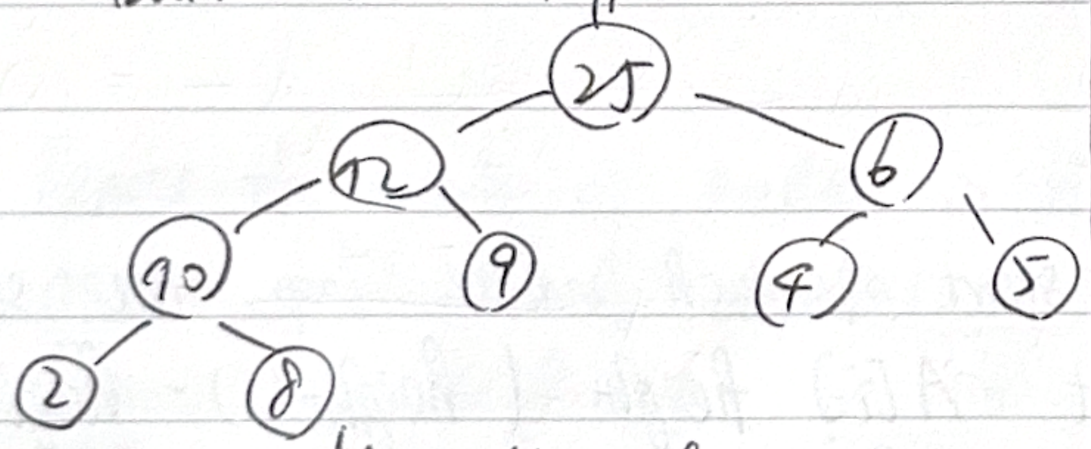


16 < 17 but it's the value of a father node.

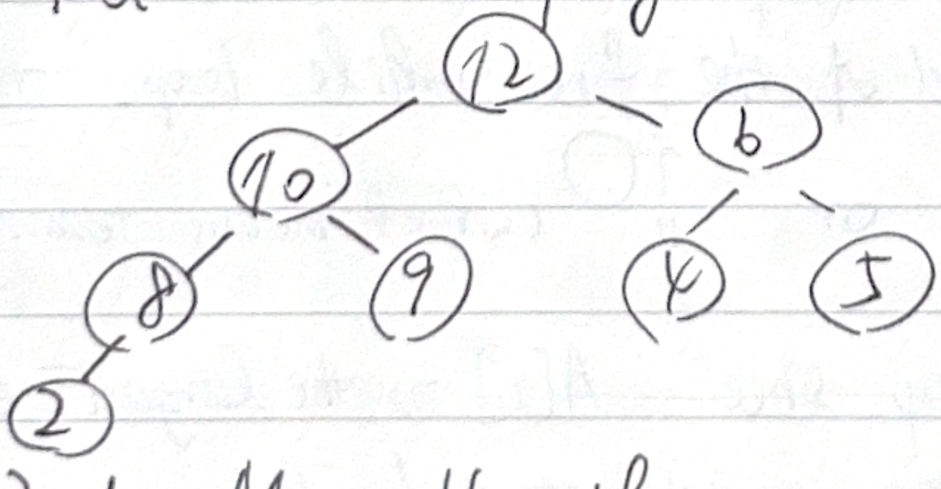
Q4.2.



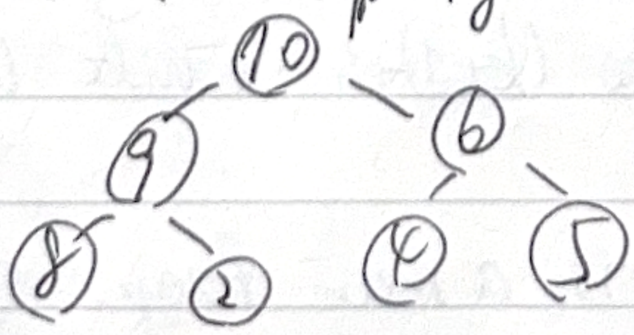
After Build-Max-Heap:



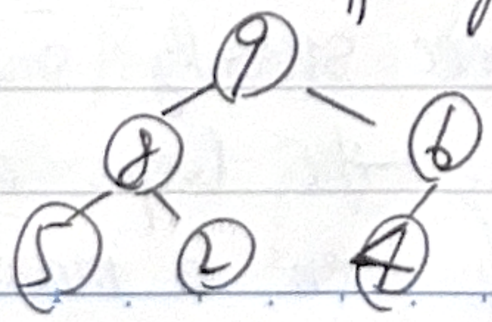
After first Max-Heapify:



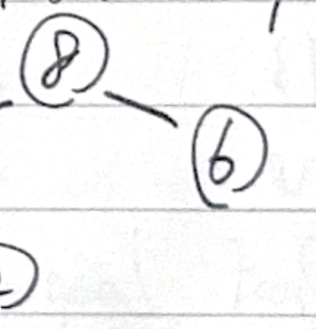
After 2nd Max-Heapify:



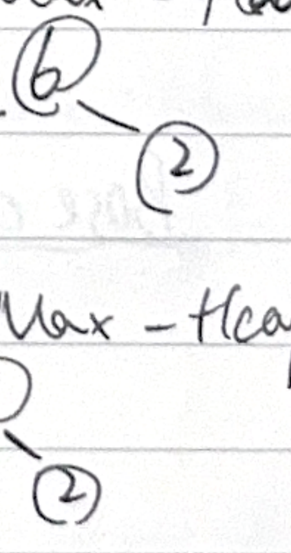
After 3rd Max-Heapify:



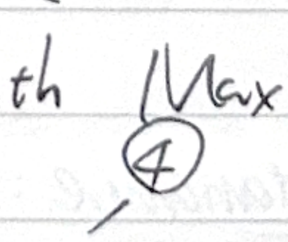
After 4th Max-Heapify:



After 5th Max-Heapify:

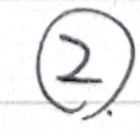


After 6th Max-Heapify:



After 7th Max-Heapify:

After 8th Max-Heapify.



NO.

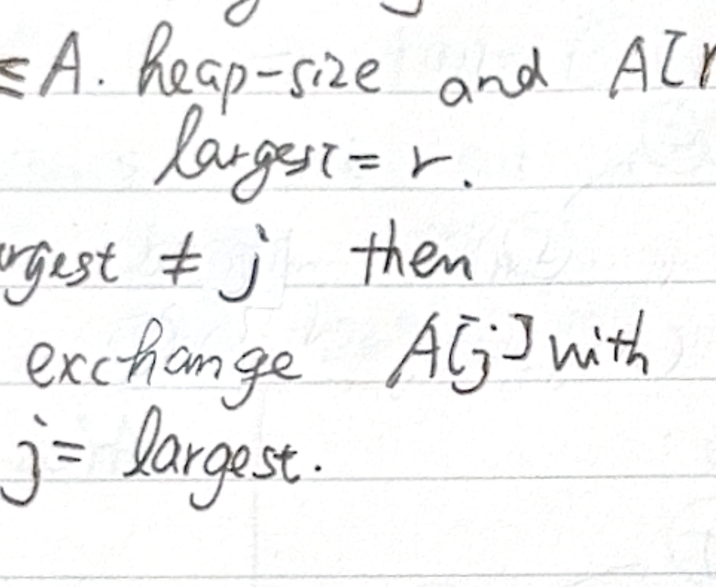
DATE

Q.4.3

```
1  1.   j = i
   2.  while TRUE do
   3.        l = Left (j)
   4.        r = Right (j)
   5.        if l ≤ A.heap-size and A[l] > A[j] then
   6.              largest = l.
   7.        else
   8.              largest = j
   9.        if r ≤ A.heap-size and A[r] > A[j] then
   10.             largest = r.
   11.       if largest ≠ j then
   12.             exchange A[j] with A[largest]
   13.             j = largest.
   14.       else
   15.             break.
```

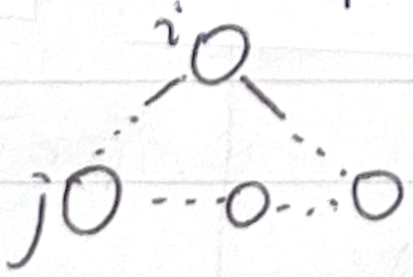
2. Proof: Loop-invariant: At the end of each while loop, the subtree with root i of height (height(i) - height(j)) is a max heap.

Initialization: At the end of the first while loop, the subtree



is a max heap since A[i] is the largest element or it's a degenerated tree (only one root).

Maintenance: At the end of the (k+1)th while loop, the subtree

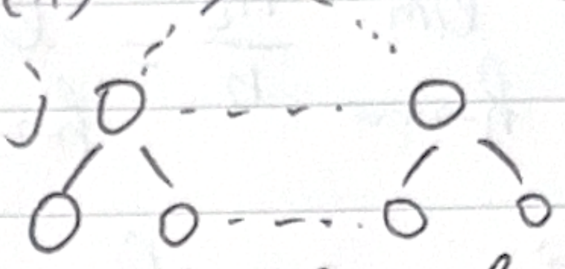


is a max heap, since the last row of leaf nodes are strictly smaller than their parents, guaranteed by the loop, and the rest of the tree has already been a max heap at the end of the last loop.

NO.

DATE

Termination: The loop terminates when "break" is activated, which means largest = j in this loop.
⇒ it implies the subtree (*): i is a



max heap that doesn't require any further modification about the violation node j!
By our assumption before our algorithm, Left(i) and Right(i) are already max-heaps, which means the subtrees with root being node j and all other nodes with the same height as node j in subtree (*) are max-heaps.
⇒ (*) is a max heap! □.

Q4.4

1. Proof: Denote the number of nodes in the left subtree of the root as n_L. Similarly define n_R. then $n = 1 + n_L + n_R$.
Denote the height of the tree is h.
Then, by the def of a heap, all leaf nodes should be put in the left subtree of the root with higher priority than be put in the right one.
⇒ $n_R \le n_L \le n_R + 2^{h-1} = n - 1 - n_L + 2^{h-1}$.
↑ extreme case: the left subtree has its last level full but the right one has its empty.

WTS: $\max\{n_R, n_L\} \le \frac{2}{3}n$.
⇐ $n_L \le \frac{2}{3}n$.
⇐ $\sup_{h\in\mathbb{N}} n_L \le \frac{2}{3}n$.
Only need to show: $\sup_{h\in\mathbb{N}} n_L \le \frac{2}{3}n$.
In fact, for $n_L \le n_R + 2^{h-1}$:
"=" holds ⇔ $\begin{cases} n_L = 2^h - 1. \\ n_R = 2^{h-1} - 1. \end{cases}$ [since left subtree is a full tree with height $h-1$. $n_L = 1 + 2 + \cdots + 2^{h-1} = 2^h - 1$. and $n_R = n_L - 2^{h-1}$]

NO.

DATE

$$\Rightarrow \frac{n_L}{n} = \frac{2^h - 1}{2^h - 1 + 2^{h-1} - 1 + 1} = \frac{2^h - 1}{2^h + 2^{h-1} - 1.}$$

$$\Rightarrow \lim_{h \to +\infty} \frac{n_L}{n} = \lim_{h \to +\infty} \frac{1 - \frac{1}{2^h}}{1 + \frac{1}{2} - \frac{1}{2^h}} = \frac{1}{1 + \frac{1}{2}} = \frac{2}{3}.$$

i.e. $\sup_{h \in \mathbb{N}} \frac{n_L}{n} = \frac{2}{3} \Rightarrow \sup_{h \in \mathbb{N}} n_L = \frac{2}{3} n.$ □.

2. Proof: $T(n) \leq T(\frac{2}{3}n) + \Theta(1)$.

$a = 1, \ b = \frac{3}{2}. \quad f(n) = \Theta(1).$

$n^{\log_b a} = n^0 = 1. \Rightarrow f(n) = \Theta(n^{\log_b a}). = \Theta(n^{\log_b a} \log^0 n)$

But be cautious! We got a "$\leq$" here!

Thus, by the Master Thm,

$$T(n) = O(n^{\log_b a} \log^{0+1} n) = O(\log n).$$

Q.45.

Argue: ① During Build-Max-Heap: we've proved that total time of Build-Max-Heap is $O(n \cdot \log n)$.

② For calls for Max-Heapify(A, 1):

The key problem is that every time we exchange A[1] and A[2], we got the largest one to the back but we have a relatively small number to the front.

⇒ Max-Heapify must keep it "floating down" to the bottom again. And the floating down path's length is proportional to the height of the heap - which is $\Omega(\log i)$.

During the first $\frac{n}{2}$ iterations, such cost has very high possibility.

$$\Rightarrow \text{cost} \geq \sum_{i=\frac{n}{2}+1}^{n} \Omega(\log(i-1)) \geq \sum_{i=\frac{n}{2}+1}^{n} \Omega(\log(\tfrac{n}{2})) = \sum_{i=\frac{n}{2}+1}^{n} \Omega(\log n)$$

$$= \frac{n}{2} \cdot \Omega(\log n) = \Omega(n \log n).$$

Thus, $\text{cost} \geq O(n \log n) + \Omega(n \log n) = \Omega(n \log n)$ #