PS 4

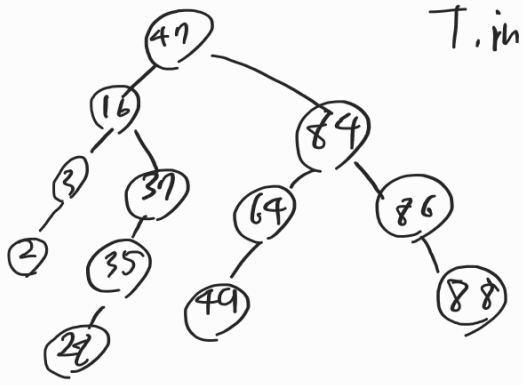
4-1.

(a) (37), (16)

skew((16)) = height((37)) - height((3)) = 3 - 1 = 2

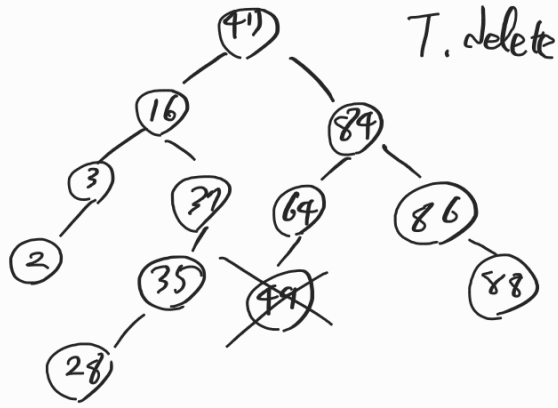
skew((37)) = height(∅) - height((35)) = 0 - 2 = -2

(b)



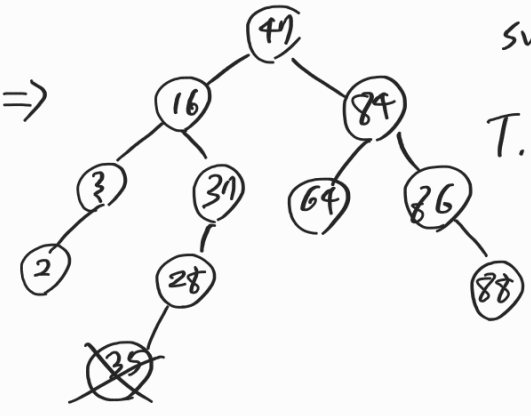
T.insert((2))

⇒



T.delete((49))

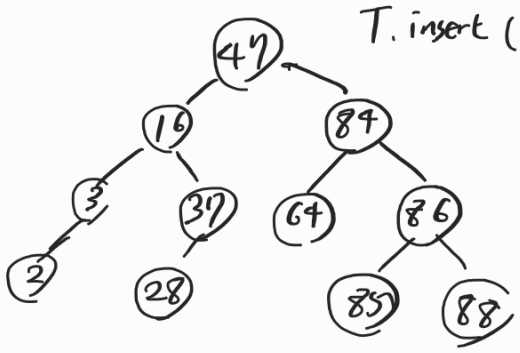
⇒



swap((35),(28))

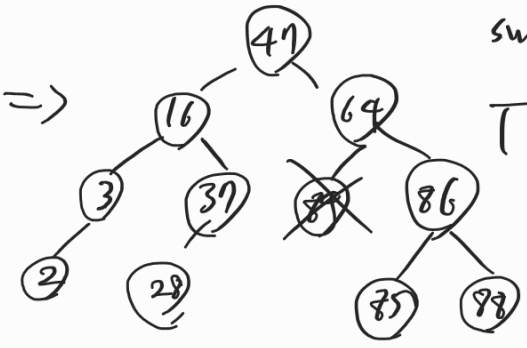
T.delete((35))

⇒



T.insert((85))

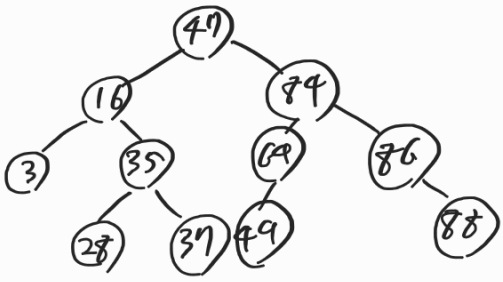
⇒



swap((64),(84))

T.delete((84))

(c) rotate_right((37))



skew((35)) = 0 skew((16)) = 1

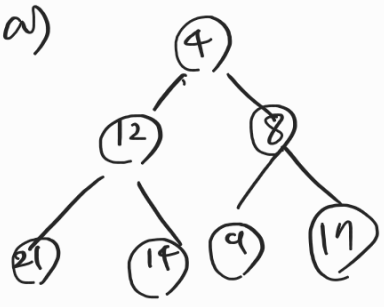
skew((47)) = 0 skew((84)) = 0

skew((64)) = -1 skew((86)) = 1

For every result of skew operation for every internal nodes, there is no result $\notin \{-1, 0, 1\}$. So the tree is balanced.

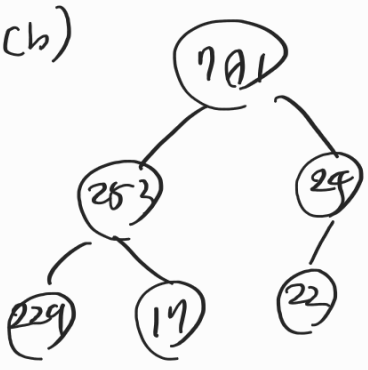
4-2

(a)



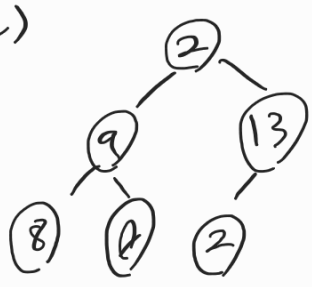
This tree is a min-heap, because for every $0 \le i \le |A| - 1$, $A[\text{parent}(i)] \le A[i]$

(b)

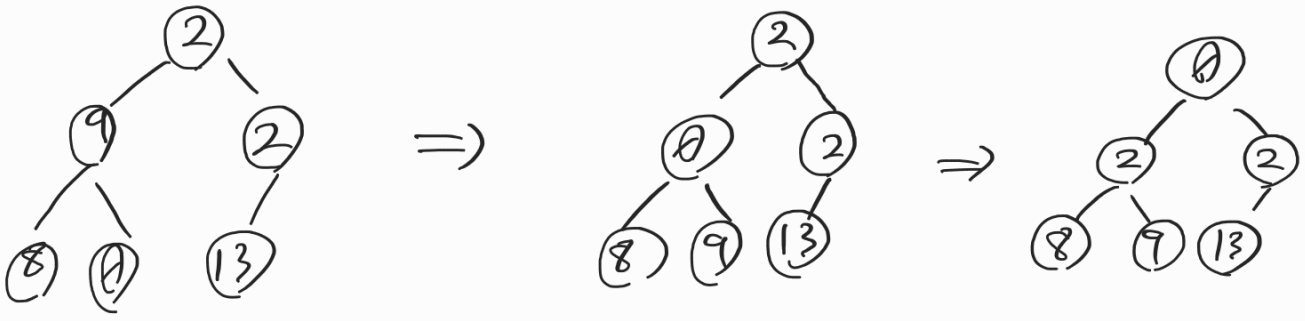


This tree is min-heap.

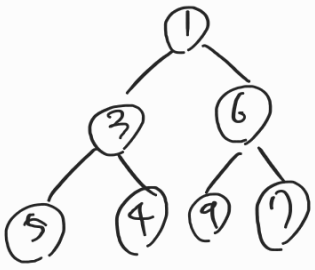
(c)



neither.



(d)



This tree is min-heap.

4-3.

(a)

Let's make A a binary heap. The height of the binary heap is $\lfloor \log(|A|) \rfloor$. Assume that max_heapify_down has been defined out of this problem context. We can max_heap by apply max_heapify_down for every node in A. Then building a heap takes,

$$\Theta\left(\sum_{i=0}^{|A|-1} [\log(|A|) - \log(i)]\right) = \Theta(|A|\log|A| - \log(|A|!)) = \Theta\left(\log \frac{|A|^{|A|}}{|A|!}\right)$$

$$= \Theta\left(\log\left(\frac{|A|^{|A|}}{\sqrt{|A|}\left(\frac{|A|}{e}\right)^{|A|}}\right)\right) = \Theta(|A|\log(e) - \log(\sqrt{|A|})) = \Theta(|A|).$$

Then we can use the heap as a priority queue. Deleting maximum in heap-priority queue takes $O(\lfloor \log|A| \rfloor)$ time. Getting k-highest scores takes O(|A| + k*log(|A|)) time.

(b)

Starting at the root node.

Traverse left subtree returning registration numbers of gardners with score larger than x. If encounter while traversing, stop walking down and traverse right subtree. And execute this procedure recursively.

```
    Alg greater_than_x(A, x, T, i)
1   input: A max_heap A of garden pairs. Return a list T containing the registration
numbers of gardners with score larger than x. i is a node(index).
2   if A[i] <= x
3       return
4   greater than x(A, x, T, left(i))
5   greater than x(A, x, T, right(i))
6   T.add(A[i])
```

The running time is

$$O(2^{\log(n_x)}) = O(n_x).$$

4-4.
Construct two data structures:
S: stores nodes with (s_i, c_i) of a solar farm, a list of customer buildings. S is a max-heap, where the root node has the largest capacity.
B: stores nodes with (b_i, d_i) and a pointer to a node in S which it is connected to.
Initialize(S): Building a max-heap takes O(n) time. In intialization phase, B is empty.
power_on(b_i, d_i): Make a building and connect it to the root node in S. Add this building to the list the connected node in S conatins. Let the integer pair in the connected node in S (s_i, c_i), be (s_i, c_i - b_i). and heapify such that S keeps max-heap property. And insert the building into B.
If the capacity of the root node in S is lower than d_i, just return to terminate.
Entirely it takes O(log(n)) time.