Problem Set 4

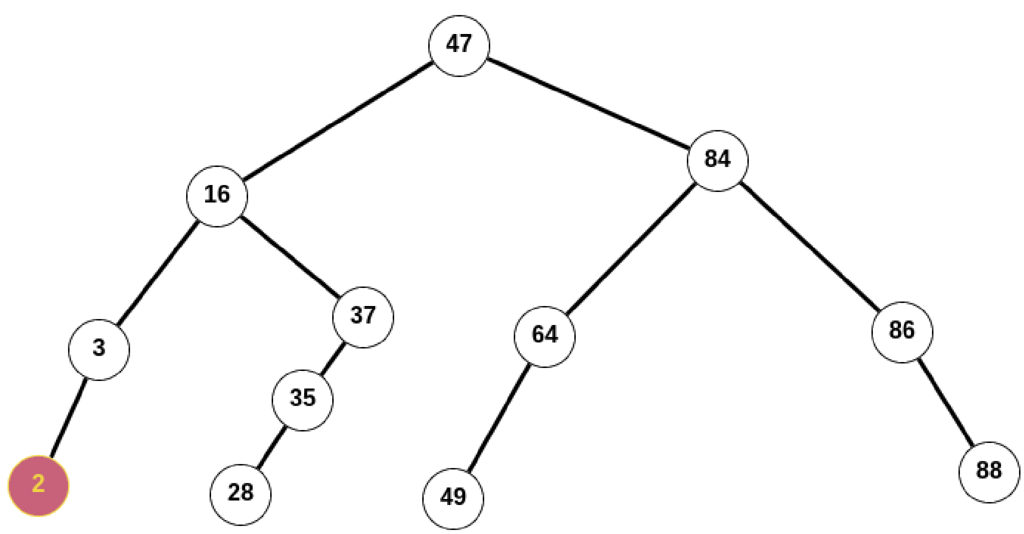
Name: Your Name

Collaborators: Name1, Name2

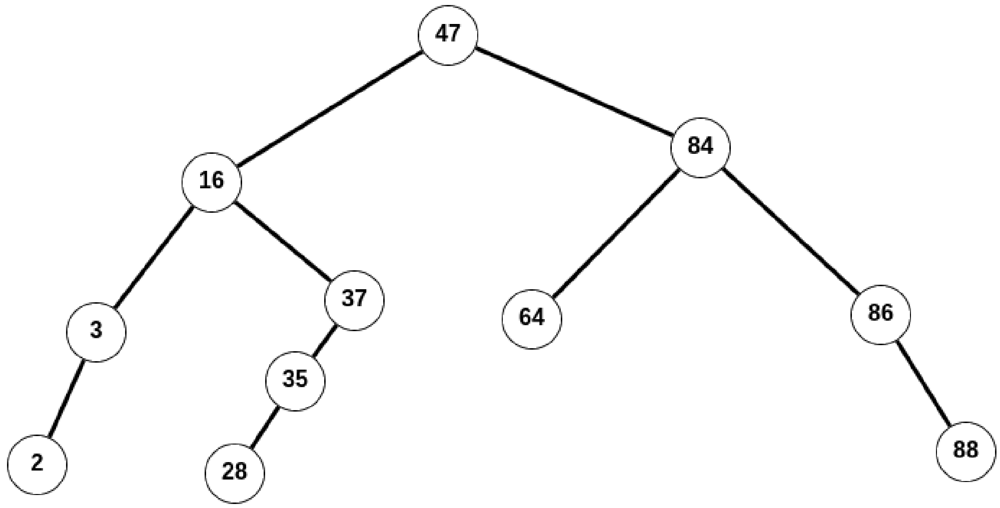
Problem 4-1.

(a) node 16 and 37 is not height balanced with skew 2 and -2 respectively.

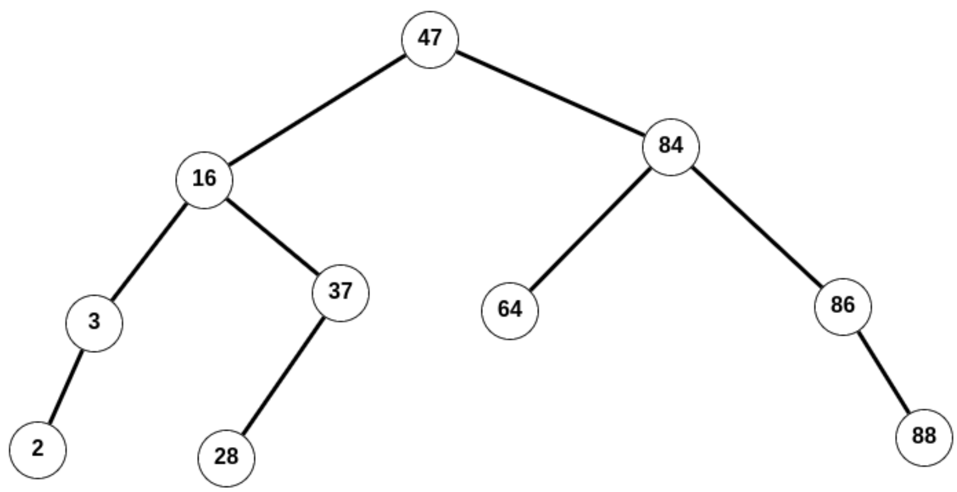
(b) `T.insert(2)`



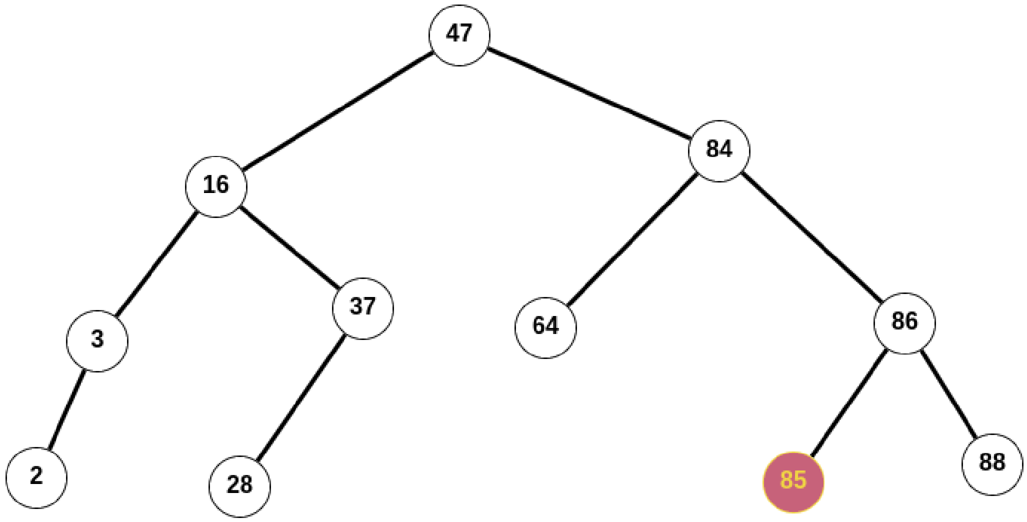
`T.delete(49)`



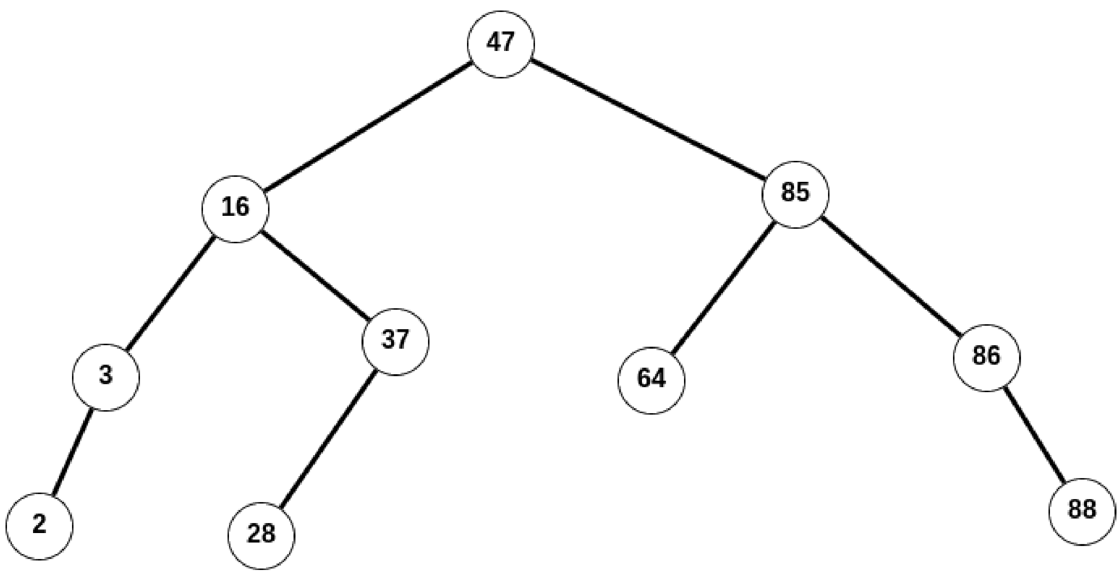

```
T.delete(35)
```



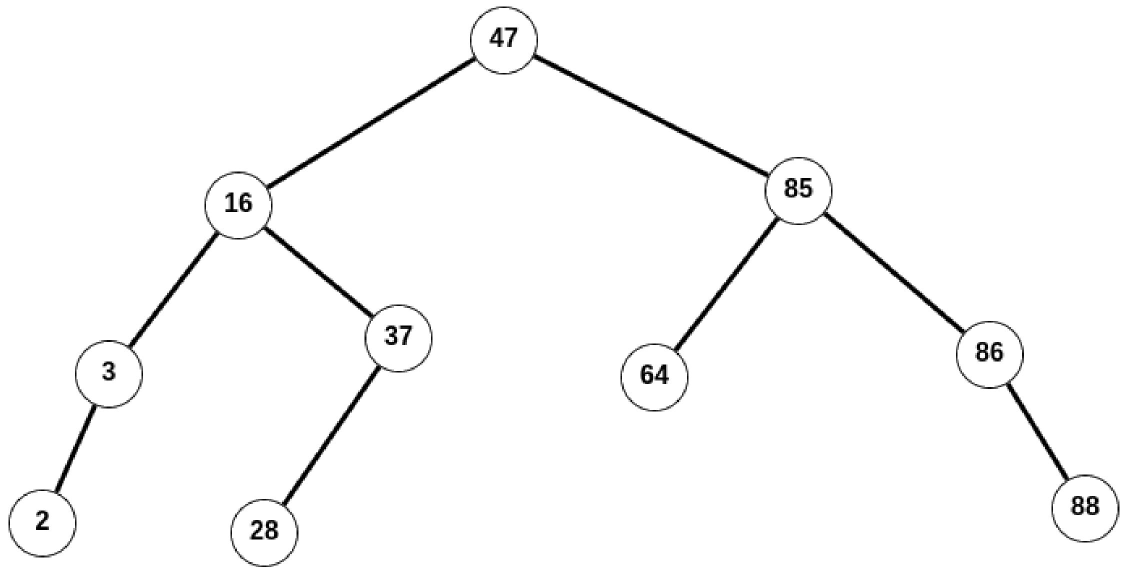

```
T.insert(85)
```



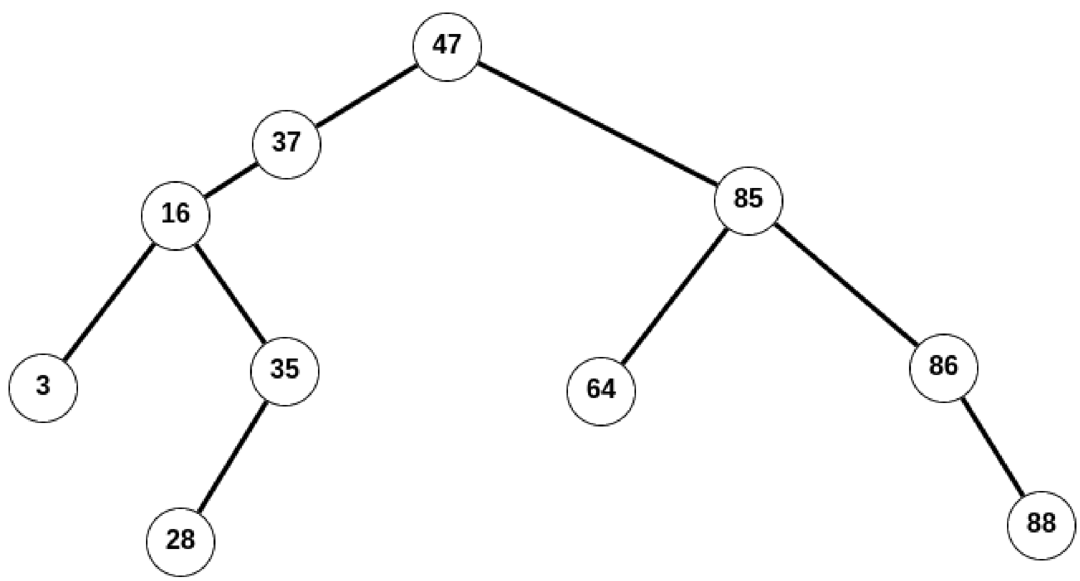
`T.delete(84)`



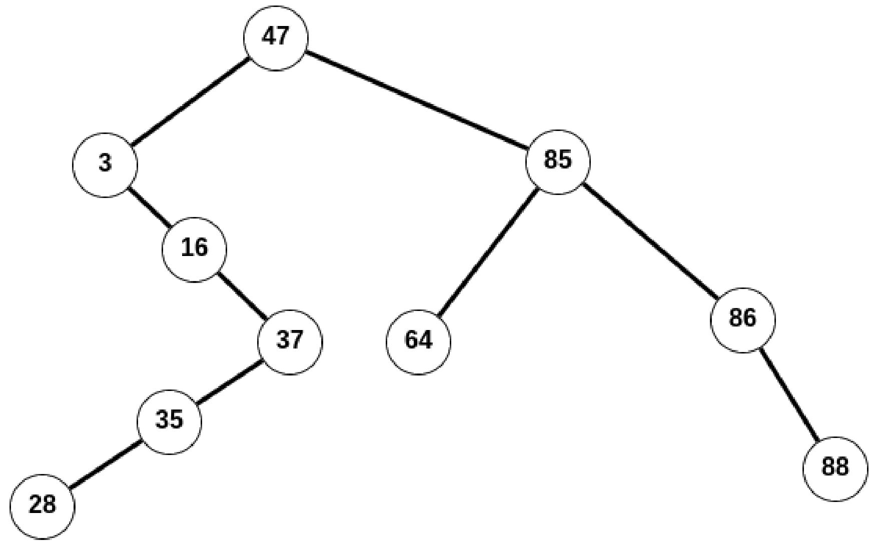
(c) • rotating right at node 16 - **Not height balanced**



- rotating left at node 16 - **Not height balanced**



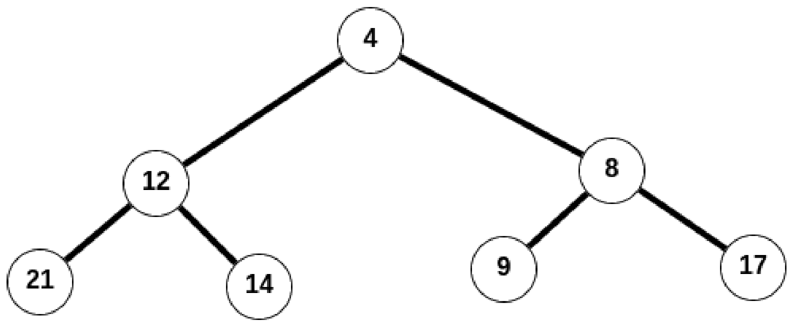
- rotating left at node 37 - **height balanced**



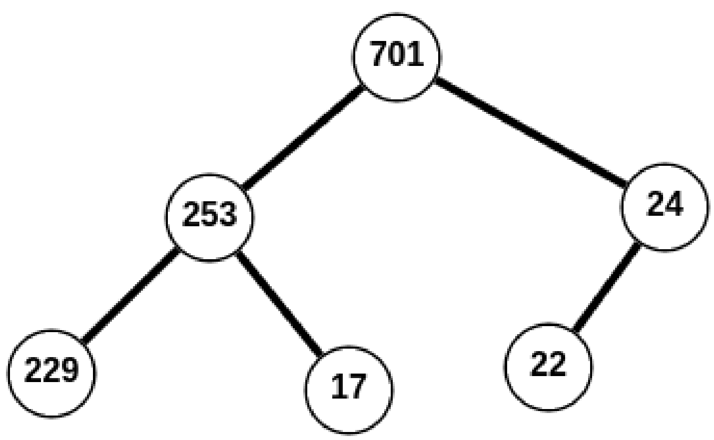
- rotating right at 37 is not possible.

Problem 4-2.

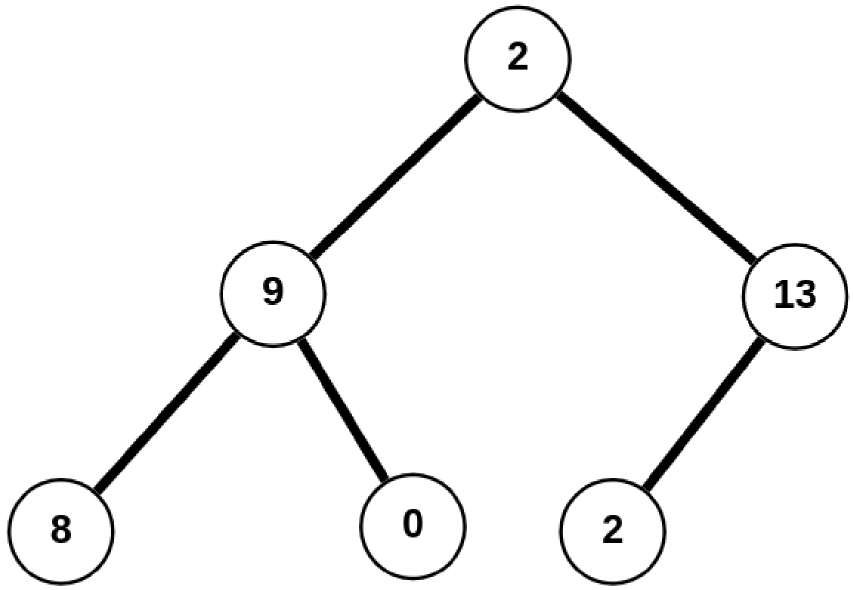
(a) min-heap

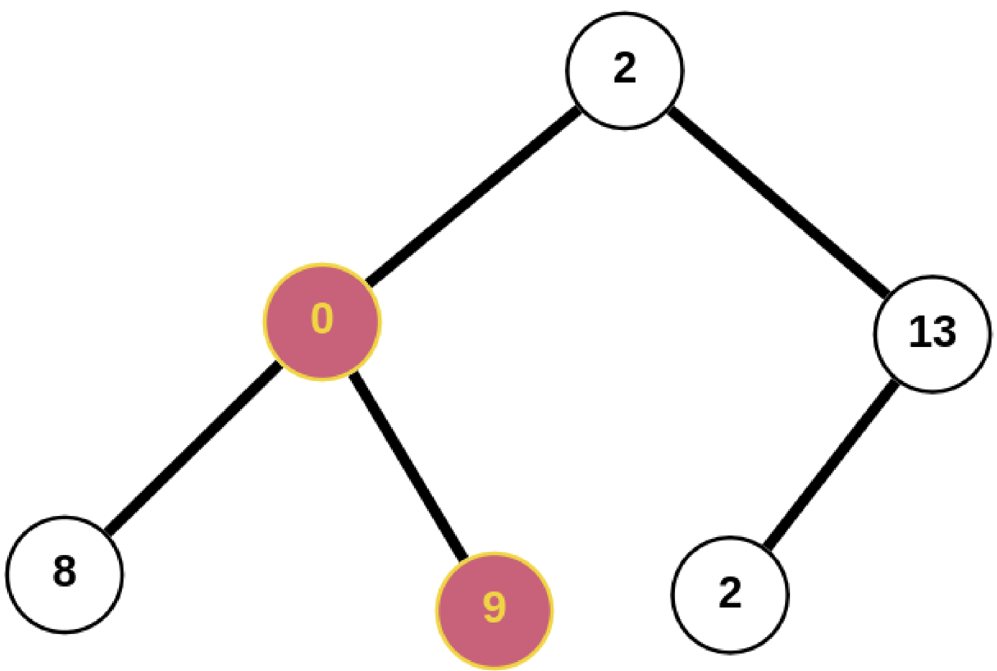


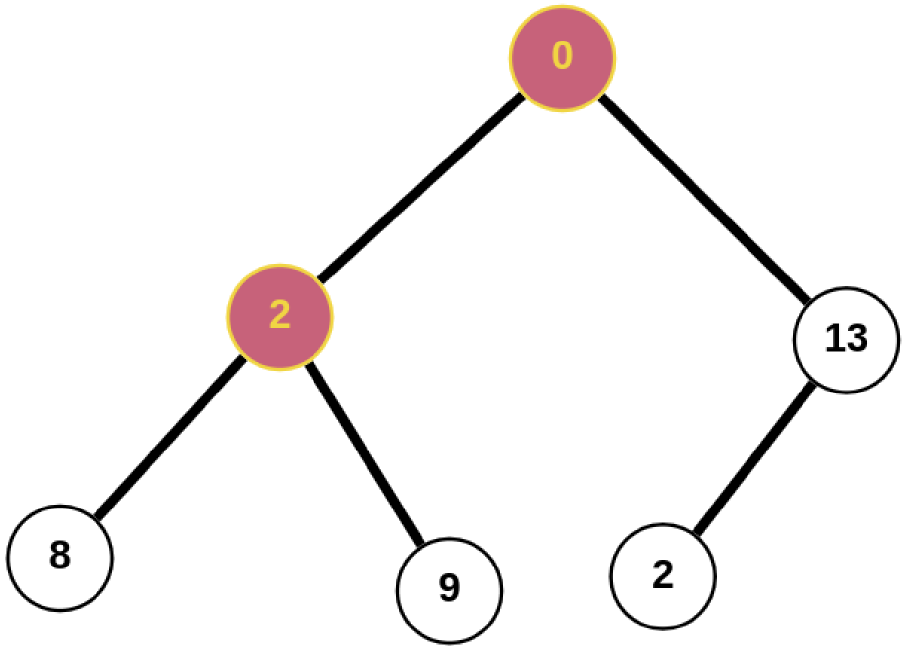
(b) max-heap

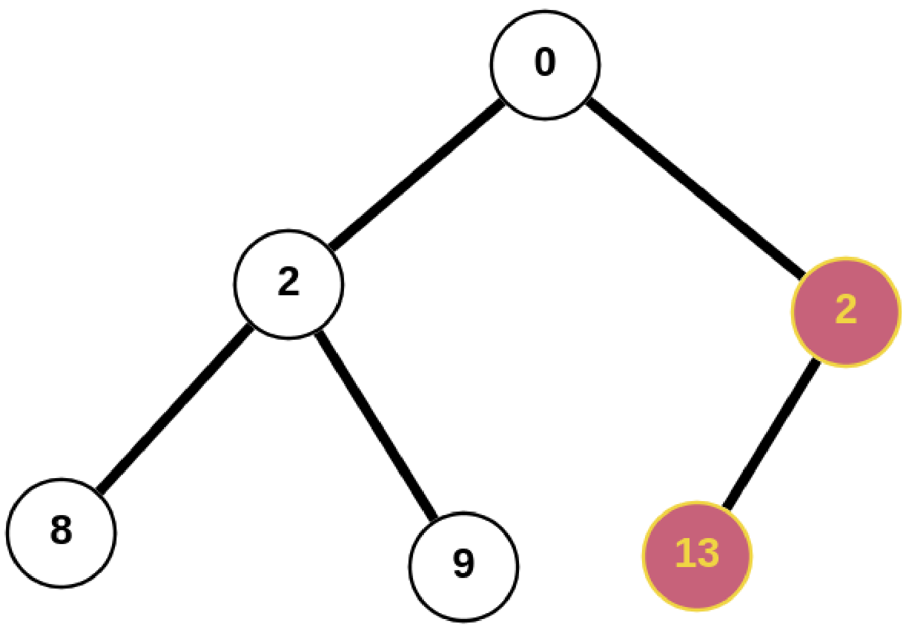


(c) neither min-heap nor max-heap

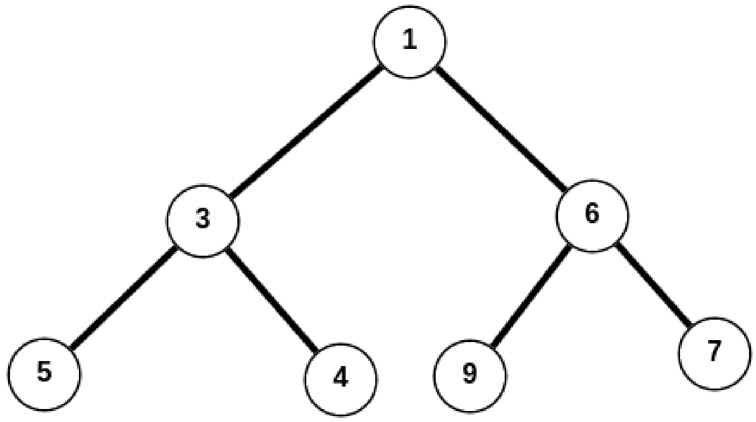

2
9
13
8
0
2


2
0
13
8
9
2





(d) min-heap


1
3
6
5
4
9
7

Problem 4-3.

(a) create a max-heap keyed by the score s_i of each node in $O(|A|)$ and start deleting the maximum element s_i in $O(\log |A|)$ for k times, so it takes $O(k \log |A|)$. this algorithm takes $O(|A| + k \log |A|)$ time.

(b) we should traverse the n_x nodes and return their registration numberes using max-heap property, so it takes $O(n_x)$ time.

- if whe current node is a greater than x return its registration number, and check its children as a recursive call.
- if the current node is not greater than x you should stop because all children of it is less than x by max-heap property.

Problem 4-4. we can achieve this using the following data structures:

- max-heap stores all solar farms (s_i, c_i) keyed by their capacity c_i.
- create a hash table B stores all buildings (b_i, d_i) hashed by their address d_i, each building saves the solar farm s_i in the max-heap it is connected to.
- create a hash table SA contains all solar farms hashed by their address s_i, which points to another hash table SAC which contains all buildings connected to the solar farm hashed by its address b_i.

- `initailiz(S)`: create the max-heap (priority queue), the hash table, and the hash table of the buildings, and the hash table of the solar farms, in $O(n)_a$ time worst case.
- `power_on(`b_i, d_j`)`: pick the building from the hash table in $O(1)_a$ time, and connect it to the solar farm with maximum capacity by subtract d_i from c_i then maintain the max-heap property in $O(logn)$ time worst case, finally assign this solar farm to this building in B in $o(1)_e$ and add the building to its solar farm in SA in the hash table SAC in $o(1)_e$ time.
- `power_off(`b_i`)`: simply we can reverse the process of `power_on(`b_i, d_j`)`, find the building in B and add its d_i to the connected solar farm in the max-heap and and maintain the max-heap property in $O(logn)$ time worst case, remove this building from the hash table SA in $O(1)_e$ time, and finally remove the building from the hash table SAC in $O(1)_e$ time.
- `customers(`s_i`)`: find the solar farm in SA in $O(1)_e$ time, and return all buildings connected to this solar farm in $O(k)$ time worst case.

Problem 4-5.

Problem 4-6.

(a) lazy to

(b) write the solution

(c) I've just implemented it.

(d) solution:

```
from Set_AVL_Tree import BST_Node, Set_AVL_Tree

class Key_Val_Item:
    def __init__(self, key, val):
        self.key = key
        self.val = val

    def __str__(self):
        return "%s,%s" % (self.key, self.val)

class Part_B_Node(BST_Node):
    def subtree_update(A):
        super().subtree_update()
        A.sum = A.item.val
        if A.left:
            A.sum += A.left.sum
        if A.right:
            A.sum += A.right.sum

        middle= A.item.val
        if A.left:
            left = A.left.max_prefix
            middle += A.left.sum
        else:
            left = -10000000000
        if A.right:
            right = middle + A.right.max_prefix
        else:
            right = -1000000000

        A.max_prefix = max(left, right, middle)
        if left == A.max_prefix:
            A.max_prefix_key= A.left.max_prefix_key
        elif right == A.max_prefix:
            A.max_prefix_key = A.right.max_prefix_key
        else:
            A.max_prefix_key = A.item.key

class Part_B_Tree(Set_AVL_Tree):
    def __init__(self):
        super().__init__(Part_B_Node)

```

```
    def max_prefix(self):
        '''
        Output: (k, s) | a key k stored in tree whose
                       | prefix sum s is maximum
        '''
        k, s = 0, 0
        if self.root:
            k, s = self.root.max_prefix_key, self.root.max_prefix
        return (k, s)

def tastiest_slice(toppings):
    '''
    Input:  toppings | List of integer tuples (x,y,t) representing
                     | a topping at (x,y) with tastiness t
    Output: tastiest | Tuple (X,Y,T) representing a tastiest slice
                     | at (X,Y) with tastiness T
    '''
    B = Part_B_Tree()   # use data structure from part (b)
    X, Y, T = 0, 0, 0
    toppings.sort(key=lambda x: x[1])
    for x, y, t in toppings:
        B.insert(Key_Val_Item(x, t))
        if B.max_prefix()[1] > T:
            X, Y, T =  B.max_prefix()[0], y, B.max_prefix()[1]
    return (X, Y, T)
```