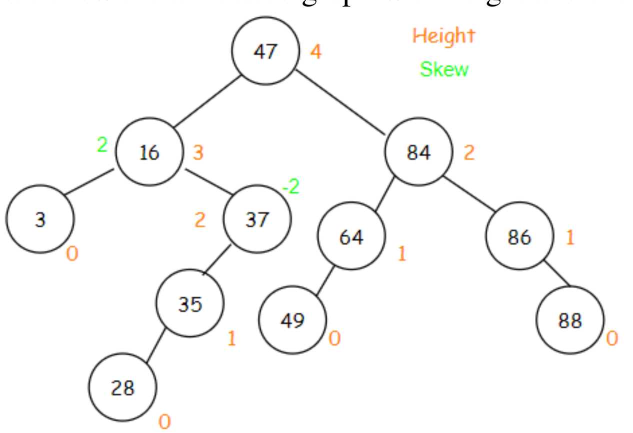
Problem Set 4

Name: Thomas Ren

Collaborators: None

Problem 4-1.

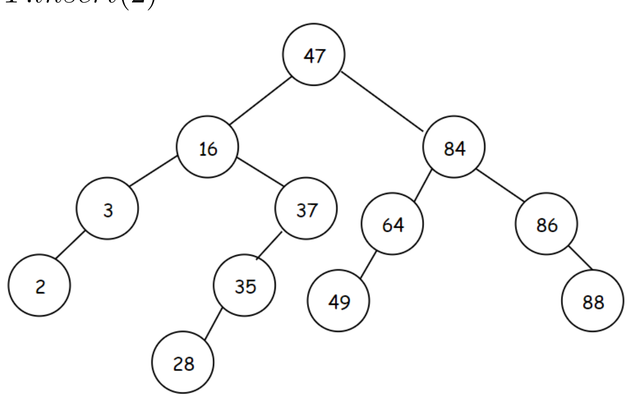
(a) We show the annotated graph with height and skew.



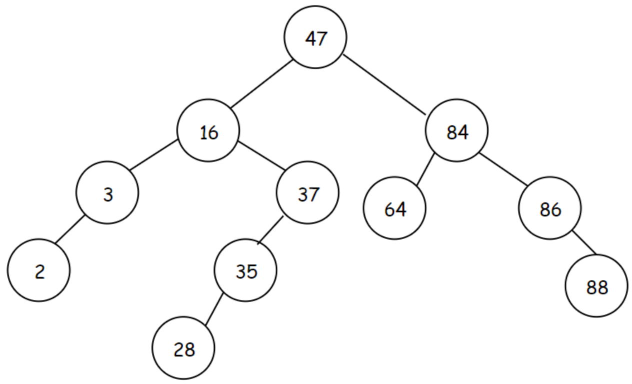
From the annotation, we can easily tell that **Node(16) and Node(37)** are not height-balanced, which means they have skews higher than 1.

(b) In this part, I'll do the following steps separately and show the graph after each operation.

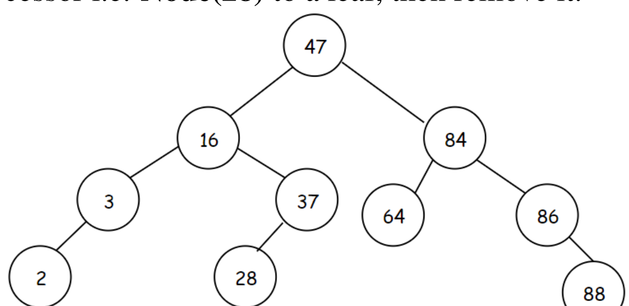
1. $T.insert(2)$



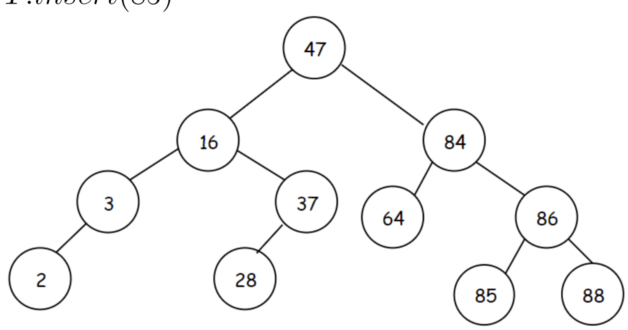
2. $T.delete(49)$. Since **Node(49)** is a leaf, we can just delete it by removing it directly.



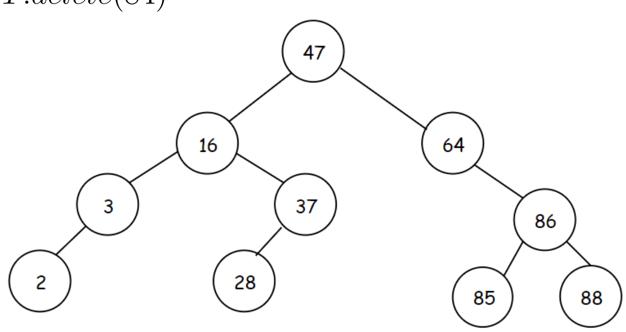
3. $T.delete(35)$. Since **Node(35)** is not a leaf, we should first swap it with its predecessor i.e. **Node(28)** to a leaf, then remove it.



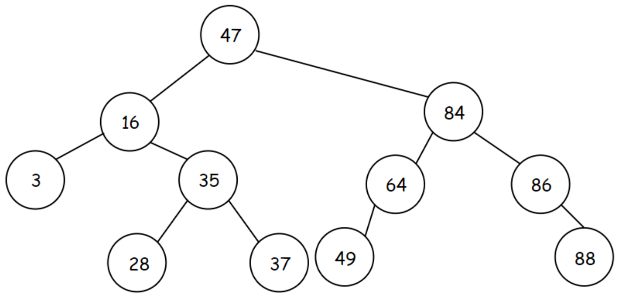
4. $T.insert(85)$



5. $T.delete(84)$

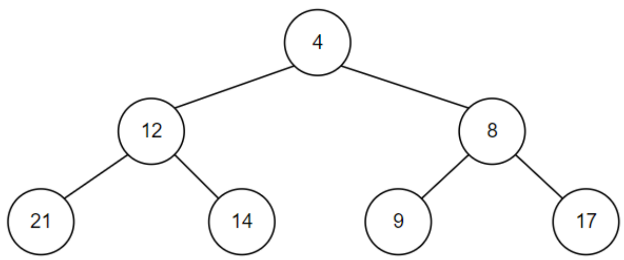


(c) Recall that **Node(16) and Node(37)** are not height-balanced, so we will use rotate to build an AVL Tree and count the height as a Set Interface property to validate the balance.
$rotate\_right(37)$ will balance the whole Binary Tree.

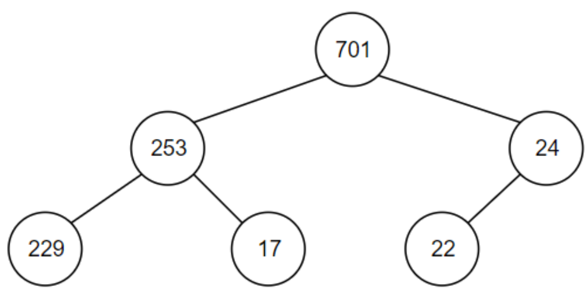


Problem 4-2.

(a) First, we draw the heap according to the given array. Then we state whether it's a max-heap or min-heap or neither.
State: Min-heap

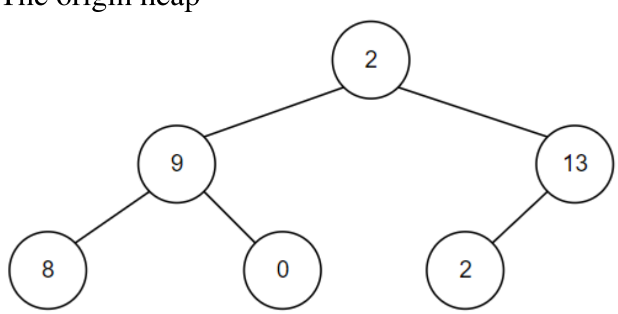


(b) **State: Max-heap**

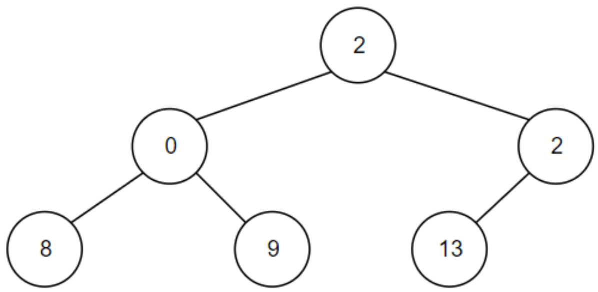


(c) **State: Neither**

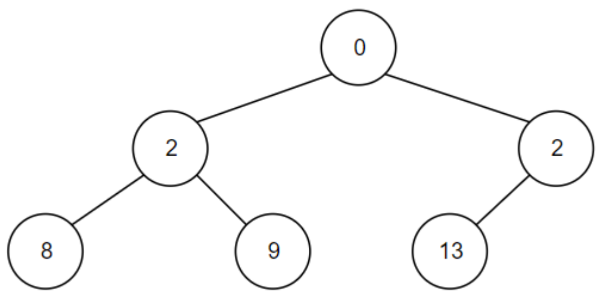
1. The origin heap



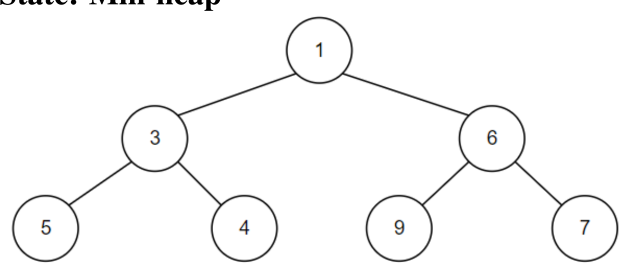
2. Swap form bottom. i.e. $height\ 1$



3. Recurse up.



(d) State: Min-heap



Problem 4-3.

(a) Since the only unique integers are the registration number, we can only build our **Set Data Structure** keyed on registration numbers r_i.
Note that we have to build our data structure in at least $O(|A|)$-time, and that leaves us $O(k \log |A|)$-time to get the k-highest scores, which means that we have to get and delete the highest score while maintaining our data structure in $O(\log |A|)$-time.
Speaking about $O(\log |A|)$-time searching in a set interface, we have **Trees** to support our operations. Nevertheless, neither **Binary Trees** nor **AVL Tree** can be build in $O(|A|)$-time, so we turn to **Binary Heap**.
Remember that we can $max\_heapify\_down()$ to build a binary heap in **linear time**, and $delete\_max()$ while returning the maximum value in $O(\log |A|)$-time, thus satisfying our time limit.

(b) The time limit $O(n_x)$ tells that we have to return a satisfying value in $O(1)$-time.
Considering our data structure is max-heap, we are only guaranteed a node's value greater or equal than its children, so we can only traverse from root and append satisfying value to a list or other data structure that can be easily built in linear time.
The traversing algorithm is as follows:

```
satisfying_garden = []
for v in range(root to leaf):
    if v.value >x:
        satisfying_garden.append(v)
        recurse on v.left
        recurse on v.right
    else:
        break
```

For this algorithm, we only need to compare at most $3n_x$ nodes' values to get the final list of gardens, thus satisfying the requirements.

Problem 4-4.

To guarantee that we can reach s_i in $O(\log n)$-time, we need to maintain a balanced tree so the height of the tree is $O(\log n)$.
Besides, we need to achieve the building name b_j in constant time, so a set interface is needed.
We give our data structure as follows:

1. A **Max-heap** H that is keyed on available capacity, which takes $O(n)$-time.
2. Augment the max-heap with a pointer to a hash table or Python dictionary A. This Python dictionary stores the buildings' name-demand pairs, so that we can reach a building's name-demand pair by firstly searching tho the solar farm in the heap and then reach the name-demand pair.
3. To get to the certain node in the heap in $O(\log n)$-time, we also need to have a dictionary B mapping the buildings to the farms.
4. A dictionary C mapping s_i to a pointer to the node in the heap.

Then we will show how to complete the operations in required time.

1. $initialize(S)$: here we maintain a max-heap by calling max-heap.build to build a max-heap keyed on capacity c_i, since the initial available capacity is the capacity c_i.
2. $power\_on(b_j, d_j)$: we directly append b_j to the root of the heap and maintain the max-heap by $max\_heapify\_down()$ in $O(\log n)$-time. And assign c_i to the dictionary mapping b_j to c_i.
3. $power\_off(b_j)$: search the c_i contains b_j in $O(\log n)$-time. Delete b_j from A and delete c_i from B. Maintain max-heap by $max\_heapify\_up()$ in $O(\log n)$-time. And other operations to maintain the whole data structure can be done in $O(1)$-time.
4. $customers(s_i)$: search the node through dictionary C in $O(1)$-time, then return A in $O(k)$-time, where $k = |A|$.

Problem 4-5.
To build a database that satisfies the requirements, we first need to identify some of the properties of the situation.

1. To change a item while maintaining the whole data structure in $O(\log n)$-time, we need a **Set AVL Tree** keyed on the joints' indices.
2. The matrix multiplication is not commutative, but the order of matrix multiplication is **the same as the traversal order**.
3. All transformation matrices are of shape (4,4), so for every node in the AVL Tree, the full transformation matrix is also of shape **(4,4)**.
4. The full transformation of the arm should be returned in $O(1)$-time, so we should augment the partial transformation for each node in AVL Tree.

We sum the aforementioned properties to build a database:

1. A **Set AVL Tree** designated as AVL keyed on the joints' indices so that the traversal order of AVL Tree corresponds to the matrix multiplication order.
2. For AVL_ Node, we augment the partial transformation using the following code:

```
    self.trans = self.left.trans.dot(self.mat).dot(self.right.trans)
```

   To do the matrix multiplication, we store the matrices as **numpy arrays** and do matrix multiplication by calling $np.dot(a, b)$ method.

At last, we illustrate how to implement the operations.

1. $initialize(M)$: Normally, it takes $O(n \log n)$-time to build a Set AVL Tree, but M here is a sorted array, so we can build the Set AVL Tree by traversing through the whole array in $O(n)$-time.
2. $update\_joint(k, M_k\prime)$: Since we have augment the partial transformation, we can recompute the partial transformation matrix in $O(1)$-time. Maintaining the data structure takes $O(\log n)$-time to update all the partial transformation matrices that are ancestors of k.
3. $full_transformation()$: This can be easily down by calling our augmentation using the following code:

```
    AVL.root.trans
```

Problem 4-6.

(a) Since $t \neq 0$, there is at least 1 topping on the slice $(x\prime, y\prime)$. So there exists a topping (x_i, y_i), where $i \in \{0, 1, \cdots, n-1\}$.
For every topping in the slice, the Cartesian coordinate (x_i, y_i) satisfies $x_i \leq x\prime, \quad y_i \leq y\prime$.
To get the same **tastiness**, our slice point (x, y) should include all toppings on that slice, which requires $x_i \leq x, y_j \leq y, \forall i, j \in \{0, 1, \cdots, n-1\}$. This means our slice point should be the largest x_i, y_j.
Then $x = max(x_i), y = max(y_j), \quad i, j \in \{0, 1, \cdots, n-1\}$

(b) To get access of the whole tree's max_prefix_sum in $O(1)$-time, we need to store the max_prefix_sum in an attribute that can be reached from the tree class directly, which is the **root**.
We can compute the $prefix\_sum$ and the max_prefix_sum by aggregating from its $left\_subtree$ and $right\_subtree$. **Note that** the right subtree only compute prefix sum in the right subtree part without including the left subtree part, cause the left subtree part doesn't belong to the tree when standing at the right subtree.
The key here is to compare whether the max_prefix_sum is in the left part or right part or exactly in the middle and then decide which part has the max_prefix_sum and pass the key follows the sum.

(c) This problem is quite similar to a **2-dimensional search**, we cannot simply search for the minimum value twice and directly output our search because we won't localize the global minimum this way. Instead, we can search at one dimension with our Set_AVL_Tree in $O(\log n)$-time to insert a node while maintaining the structure and output the max_prefix_sum in $O(1)$-time, on the other hand, we will increment on the other dimension to make sure we won't miss the **global minimum**.
Specifically, we first use **comparison model** to sort the x coordinate using **Merge Sort**, which achieve the best $O(n \log n)$-time. Then we can insert each node **keyed on** y with an increment order of x so that toppings always satisfies the slice principle. After traversing the whole toppings' list, our output of max_prefix_sum keyed on y should be the best solution and the x that comes along with it should be the global minimum.

(d) Submit your implementation to `alg.mit.edu`.

```
from Set_AVL_Tree import BST_Node, Set_AVL_Tree
########################################
# DO NOT REMOVE THIS IMPORT STATEMENT #
# DO NOT MODIFY IMPORTED CODE         #
########################################
import numpy as np


def merge(L,R):
    '''
```

```
    :param L: the left sorted list
    :param R: the right sorted list
    :return: a sorted list combining the left and right in increasing order
    '''
    Merged_list = []
    L_len = len(L)
    R_len = len(R)
    i,j = 0,0
    while i<L_len and j<R_len:
        if L[i][0]<=R[j][0]:
            Merged_list.append(L[i])
            i += 1
        else:
            Merged_list.append(R[j])
            j += 1
    if i == L_len:
        while j<R_len:
            Merged_list.append(R[j])
            j += 1
    else:
        while i<L_len:
            Merged_list.append(L[i])
            i += 1

    return Merged_list


def Merge_Sort(A):
    '''
    :param A: an unsorted list
    :return: a sorted version of A in increasing order
    '''

    if len(A) == 1:
        return A
    elif len(A) == 2:
        if A[0][0]<=A[1][0]:
            return A
        else:
            return A[::-1]
    else:
        half = len(A)//2
        L = Merge_Sort(A[:half])
        R = Merge_Sort(A[half:])
        return merge(L,R)

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)

def prefix_sum(A):
    if A:
        return A.prefix_sum
    else:
        return int(0)

def max_prefix_sum(A):
    if A:
        return A.max_prefix_sum
    else:
        return -float('inf')

class Part_B_Node(BST_Node):


    def subtree_update(A):
        super().subtree_update()
        #########################################
        # ADD ANY NEW SUBTREE AUGMENTATION HERE #
        #########################################

        A.prefix_sum = A.item.val
        if A.left:
            A.prefix_sum += A.left.prefix_sum
        if A.right:
            A.prefix_sum += A.right.prefix_sum

        left, right = -float('inf'), -float('inf')
        middle = A.item.val

        if A.left:
            left = A.left.max_prefix_sum
            middle += A.left.prefix_sum
        if A.right:
            right = middle + A.right.max_prefix_sum
        A.max_prefix_sum = max(left,middle,right)
        if A.max_prefix_sum == left:
            A.max_prefix_key = A.left.max_prefix_key
        elif A.max_prefix_sum == middle:
            A.max_prefix_key = A.item.key
        else:
            A.max_prefix_key = A.right.max_prefix_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
        ##################
        # YOUR CODE HERE #
        ##################

        k = self.root.max_prefix_key
        s = self.root.max_prefix_sum

        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
    ##################
    # YOUR CODE HERE #
    ##################
    # C = Part_B_Tree()
    # X_kv_pair = []
    # Y_kv_pair = []
    # for i, item in enumerate(toppings):
    #     X_kv_pair.append(Key_Val_Item(key=item[0], val=item[2]))
    # B.build(X_kv_pair)
    # X,_ = B.max_prefix()
    # for i, item in enumerate(toppings):
    #     if item[0]<=X:
    #         Y_kv_pair.append(Key_Val_Item(key=item[1],val=item[2]))
    # C.build(Y_kv_pair)
    # Y,T = C.max_prefix()

    toppings = Merge_Sort(toppings)

    for topping in toppings:
        B.insert(Key_Val_Item(key=topping[1],val=topping[2]))
        (y,t) = B.max_prefix()
        if T<t:
            X,Y,T = topping[0],y,t
```

```
    return (X, Y, T)
```