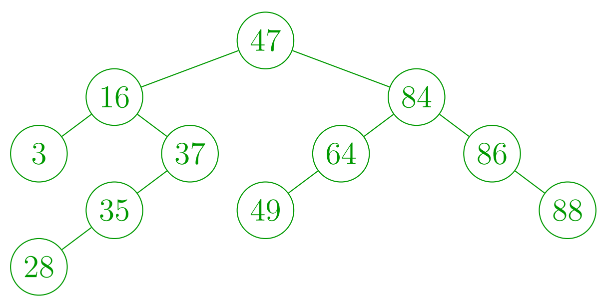
Problem Set 4

Please write your solutions in the LaTeX and Python templates provided. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct.

Problem 4-1. [10 points] **Binary Tree Practice**

(a) [2 points] The Set Binary Tree `T` below is **not height-balanced** but does satisfy the **binary search tree** property, assuming the key of each integer item is itself. Indicate the keys of all nodes that are not height-balanced and compute their skew.



Solution: The nodes containing the keys 16 and 37 are not height balanced. Their skews are 2 and -2 respectively.

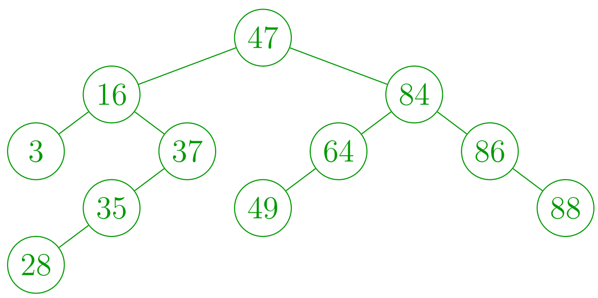
Rubric:

- 1 point for each correct node and skew

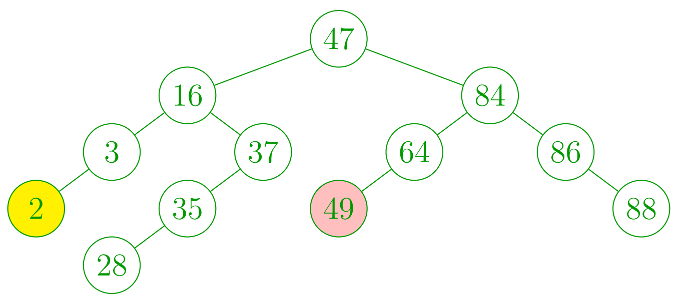
(b) [5 points] Perform the following insertions and deletions, one after another in sequence on `T`, by adding or removing a leaf while maintaining the binary search tree property (a key may need to be swapped down into a leaf). For this part, **do not** use rotations to balance the tree. Draw the modified tree after each operation.

```
T.insert(2)
T.delete(49)
T.delete(35)
T.insert(85)
T.delete(84)
```

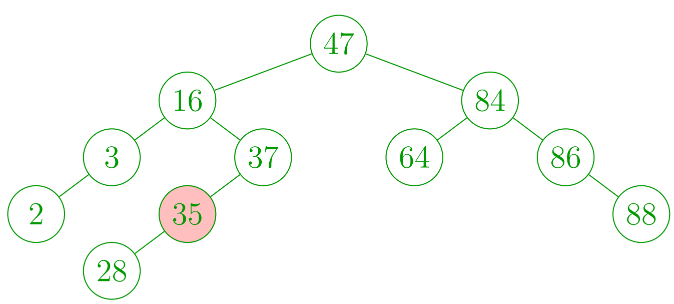
Solution:



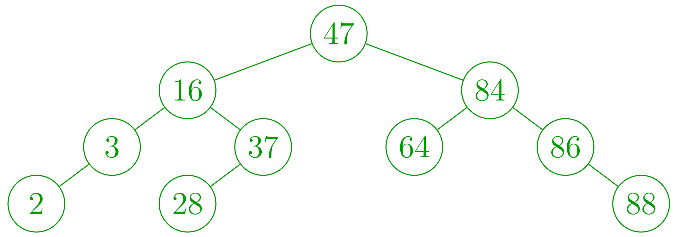
`insert(2)`



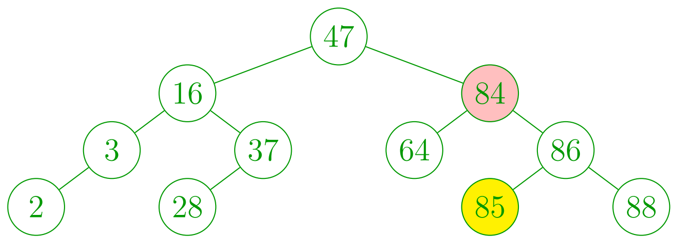
`delete(49)`



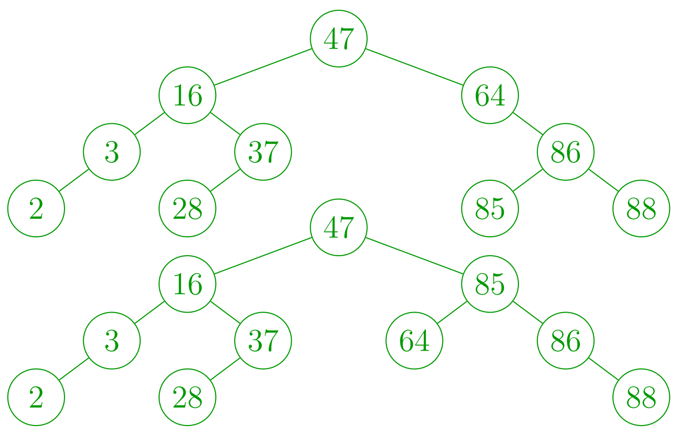
`delete(35)`



`insert(85)`



`delete(84)` (Two solutions, swap down predecessor/successor)



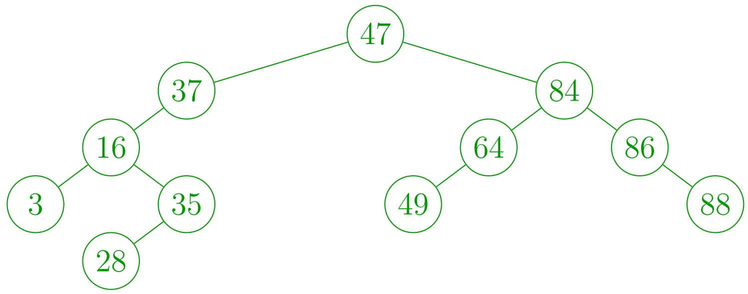
Rubric:

- 1 point for each correct operation, relative to the previous tree

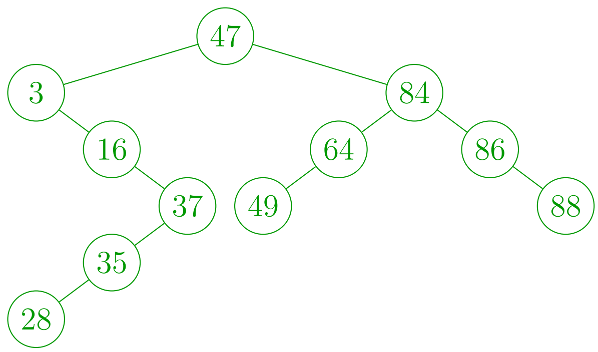
(c) [3 points] For each unbalanced node identified in part (a), draw the two trees that result from rotating the node in the **original** tree left and right (when possible). For each tree drawn, specify whether it is height-balanced, i.e., all nodes satisfy the AVL property.

Solution: Node containing 16 is not height-balanced.

- Rotating left at this node does not balance the tree:

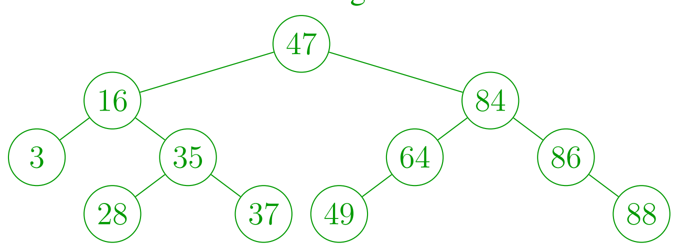


- Rotating right at this node does not balance the tree:



Node containing 37 is not height-balanced.

- Rotating left at this node is not possible
- Rotating right at this node results in a height-balanced tree!



Rubric:

- 1 point for each correct rotation, relative to the previous tree

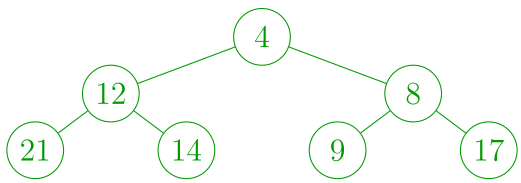
Note: Material on this page requires material that will be covered in **L08 on March 3, 2020**. We suggest waiting to solve these problem until after that lecture. All other pages of this assignment can be solved using only material from L07 and earlier.

Problem 4-2. Heap Practice [10 points]

For each array below, draw it as a **complete**[1] binary tree and state whether the tree is a max-heap, a min-heap, or neither. If the tree is neither, turn the tree into a min-heap by repeatedly swapping items that are **adjacent in the tree**. Communicate your swaps by drawing a sequence of trees, marking on each tree the pair that was swapped.

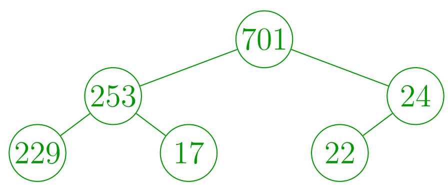
(a) `[4, 12, 8, 21, 14, 9, 17]`

Solution: Min-heap



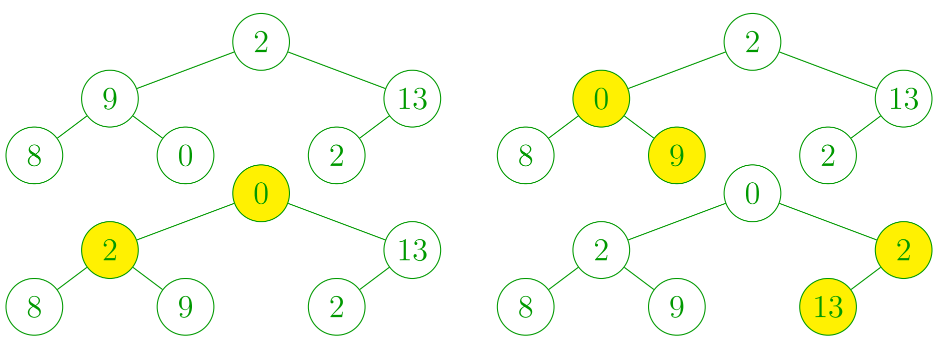
(b) `[701, 253, 24, 229, 17, 22]`

Solution: Max-heap



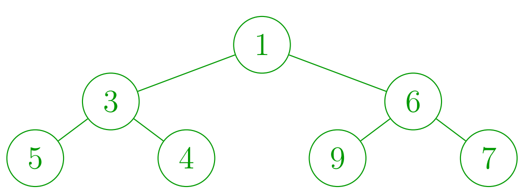
(c) `[2, 9, 13, 8, 0, 2]`

Solution: Neither: three swaps suffice to transform into a min-heap



(d) `[1, 3, 6, 5, 4, 9, 7]`

Solution: Min-heap



[1] Recall from Lecture 8 that a binary tree is ***complete*** if it has exactly 2^i nodes of depth i for all i except possibly the largest, and at the largest depth, all nodes are as far left as possible.

Rubric:

- 1 point per correct tree drawing
- 1 point per correct categorization
- 2 points for a correct sequence of swaps for part (c)

Problem 4-3. [10 points] **Gardening Contest**

Gardening company Wonder-Grow sponsors a nation-wide gardening contest each year where they rate gardens around the country with a positive integer[2] ***score***. A garden is designated by a ***garden pair*** (s_i, r_i), where s_i is the garden's assigned score and r_i is the garden's unique positive integer ***registration number***.

(a) [5 points] To support inclusion and reduce competition, Wonder-Grow wants to award identical trophies to the top k gardens. Given an unsorted array A of garden pairs and a positive integer $k \le |A|$, describe an $O(|A| + k \log |A|)$-time algorithm to return the registration numbers of k gardens in A with highest scores, breaking ties arbitrarily.

Solution: Build a max-heap from array A keyed on the garden scores s_i, which can be done in $O(|A|)$ time. Then repeatedly remove the maximum pair k times using `delete_max()`, and return the registration numbers of the pairs extracted (say in an array of size k). Because a max-heap correctly removes **some** maximum from the heap with each deletion in $O(\log |A|)$ time, this algorithm is correct and runs in $O(|A| + k \log |A|)$ time.

(b) [5 points] Wonder-Grow decides to be more objective and award a trophy to every garden receiving a score strictly greater than a reference score x. Given a max-heap A of garden pairs, describe an $O(n_x)$-time algorithm to return the registration numbers of all gardens with score larger than x, where n_x is the number of gardens returned.

Solution: For this problem, we cannot afford $O(n_x \log |A|)$ time to repeatedly delete the maximum from A until the deleted pair has score less than or equal to x. However, we can exploit the max-heap property to traverse only an $O(n_x)$ subset of the max-heap containing the largest n_x pairs. First observe that the max-heap property implies all n_x items having key larger than x form a connected subtree of the heap containing the root (assuming any stored score is greater than x). So recursively search node v of the heap starting at the root. There are two cases, either:

- the score at v is $\le x$, so return an empty set since, by the max-heap property, no pair in v's subtree should be returned; or

[2] In this class, when an integer or string appears in an input, without listing an explicit bound on its size, you should assume that it is provided inside a constant number of machine words in the input.

- the score at v is $> x$, so recursively search the children of v (if they exist) and return the score at v together with the scores returned by the recursive calls (which is correct by induction).

This procedure visits at most $3n_x$ nodes (the nodes containing the n_x items reported and possibly each such node's two children), so this procedure runs in $O(n_x)$ time.

Rubric:

- 1 points for a description of a correct algorithm
- 1 point for analysis of correctness
- 1 point for analysis of running time
- 2 points correct algorithm is efficient
- Partial credit may be awarded

Problem 4-4. [15 points] **Solar Supply**

Entrepreneur Bonty Murns owns a set S of n solar farms in the town of Fallmeadow. Each solar farm $(s_i, c_i) \in S$ is designated by a unique positive integer ***address*** s_i and a farm ***capacity*** c_i: a positive integer corresponding to the maximum energy production rate the farm can support. Many buildings in Fallmeadow want power. A building (b_j, d_j) is designated by a unique ***name*** string b_j and a ***demand*** d_j: a positive integer corresponding to the building's energy consumption rate.

To receive power, a building in Fallmeadow must be connected to a **single** solar farm under the restriction that, for any solar farm s_i, the sum of demand from all the buildings connected to s_i may not exceed the farm's capacity c_i. Describe a database supporting the following operations, and for each operation, specify whether your running time is worst-case, expected, and/or amortized.

`initialize(`S`)`	Initialize database with a list $S = ((s_0, c_0), \dots, (s_{n-1}, c_{n-1}))$ corresponding to n solar farms in $O(n)$ time.
`power_on(`b_j`,`d_j`)`	Connect a building with name b_j and demand d_j to any solar farm having available capacity at least d_j in $O(\log n)$ time (or return that no such solar farm exists).
`power_off(`b_j`)`	Remove power from the building with name b_j in $O(\log n)$ time.
`customers(`s_i`)`	Return the names of all buildings supplied by the farm at address s_i in $O(k)$ time, where k is the number of building names returned.

Solution: Our approach will be to maintain the following data structures:

- a Priority Queue P on the solar farms, storing for each solar farm its address s_i, capacity c_i, and its available capacity a_i (initially $a_i = c_i$), keyed on available capacity;
- a Set data structure B mapping each powered building's name b_j to the address of the solar farm s_i that it is connected to and its demand d_j; and

- a Set data structure F mapping the address of each solar farm s_i to: (1) its own Set data structure B_i containing the buildings associated with that farm, and (2) a pointer to the location of s_i in P.

Now we support the operations:

- `initialize(`S`)`: build Set data structures P and then F from S, and initialize all other data structures above as empty. This operation directly maintains the invariants of our database by reducing to build for F and P. There are $O(n)$ empty data structures and pointers constructed, so if we implement P and F with data structures that can build in $O(n)$ time, this operation will also take $O(n)$ time.
- `power_on(`b_j`, `d_j`)`: assume that b_j is not already connected to power (the operation is unspecified otherwise). First, find a solor farm to connect by deleting a solar farm s_i from P having largest available capacity c_i (`delete_max`) and checking whether it's capacity is at least d_j. There are two cases:
  - $d_j > c_i$, so reinsert the solar farm back into P (relinking a pointer from F to a location in P) and return that no solar farm can currently support the building.
  - $d_j \leq c_i$, so subtract d_j from c_i and reinsert it back into P (relinking a pointer). Then, add b_j to B mapping to s_i, and then find the B_i in F associated with s_i and add b_j to B_i.

  This operation directly maintains the invariants of our database and takes time asymptotically upperbounded by the sum of: one delete max and one insert operation on P, an insert operation on B, a find on F, an insertion into B_i, and constant additional work (to maintain pointers and perform arithmetic).
- `power_off(`b_j`)`: assume that b_j is already connected to power (the operation is unspecified otherwise). Lookup the s_i and d_j associated with b_j in B, lookup B_i in F using s_i, and remove b_j from B_i. Lastly, go to s_i's location in P and remove s_i from P, increase c_i by d_j, and reinsert s_i into P. This operation directly maintains the invariants of our database, and takes time asymptotically upperbounded by the sum of: one lookup in B, one lookup in F, one delete from B_i, one removal by location from P, one insertion into P, and constant additional work.
- `customers(`s_i`)`: lookup B_i in F using s_i, and return all names stored in B_i. This operation is correct based on the invariants maintained by the data structure (B_i contains the buildings associated with s_i), and takes time asymptotically upperbounded by the sum of: one lookup in F and one iteration through B_i.

We have shown this database can correctly support the operations. Now we choose implementations of the data structures in the database that will allow the operations to be efficient. We need to be able to build B and F in $O(n)$ time with $O(\log n)$ lookups, so we must use hash tables for these, leading to expected bounds on all operations, and amortized bounds on `power_on` and `power_off`. For each B_i, we need $O(\log n)$ lookup, insert, and delete, so can implement with either a Set AVL Tree or a hash table. Lastly, P requires $O(n)$ build, and $O(\log n)$ insert and delete

delete max, so either a Max-Heap or a Sequence AVL Tree augmented with subtree max items can be used. Note that removing an item from a Sequence AVL Tree via a pointer to its node is exactly the same as deleting the item after being found by index. Removing an item from a Max-Heap by index is not natively supported, but uses the same technique as removing the root: swap the item with the last leaf (the last item in the array), remove the item, and then swap the moved item up or down the tree to restore the Max-Heap property.

Rubric:

- 2 points for a description of a correct database
- 1 point for correct algorithm for first and last operation (2)
- 2 points for correct algorithm for middle two dynamic operations (2)
- 1 points for correct argument of correctness
- 2 points for correct argument of running times
- 4 points if correct database is efficient
- Partial credit may be awarded

Problem 4-5. [15 points] **Robot Wrangling**

Dr. Squid has built a robotic arm from $n+1$ rigid bars called ***links***, each connected to the one before it with a rotating joint (n joints in total). Following standard practice in robotics[3], the orientation of each link is specified locally relative to the orientation of the previous link. In mathematical notation, the change in orientation at a joint can be specified using a 4×4 ***transformation matrix***. Let $\mathcal{M} = (M_0, \dots, M_{n-1})$ be an array of transformation matrices associated with the arm, where matrix M_k is the change in orientation at joint k, between links k and $k+1$.

To compute the position of the ***end effector***[4], Dr. Squid will need the arm's ***full transformation***: the ordered matrix product of the arm's transformation matrices, $\prod_{k=0}^{n-1} M_k = M_0 \cdot M_1 \cdot \dots \cdot M_{n-1}$. Assume Dr. Squid has a function `matrix_multiply`(M_1, M_2) that returns the matrix product[5] $M_1 \times M_2$ of any two 4×4 transformation matrices in $O(1)$ time. While tinkering with the arm changing one joint at a time, Dr. Squid will need to re-evaluate this matrix product quickly. Describe a database to support the following **worst-case** operations to accelerate Dr. Squid's workflow:

`initialize`($\mathcal{M}$)	Initialize from an initial input configuration $\mathcal{M}$ in $O(n)$ time.
`update_joint`(k, M)	Replace joint k's matrix M_k with matrix M in $O(\log n)$ time.
`full_transformation()`	Return the arm's current full transformation in $O(1)$ time.

Solution: Store the matrices in a Sequence AVL tree T, where every node `v` stores a matrix `v.M` and is augmented with `v.P`: the ordered product of all matrices in `v`'s subtree. This property at a node `v` can be computed in $O(1)$ time from the augmentations of its children. Specifically, let `PL`

[3] More on forward kinematic robotics computation here: https://en.wikipedia.org/wiki/Forward_kinematics

[4] i.e., the device at the end of a robotic arm: https://en.wikipedia.org/wiki/Robot_end_effector

[5] Recall, matrix multiplication is not commutative, i.e., $M_1 \cdot M_2 \neq M_2 \cdot M_1$, except in very special circumstances.

and `PR` be `v.left.P` and `v.right.P` respectively (or the 4×4 identity matrix if the respective left or right child is missing); then `v.P = PL · v.M · PR` which can be computed in $O(1)$ time using the provided `matrix_multiply` function, so this augmentation can be maintained. (Note that, because the number of items and traversal order never changes, AVL behavior is not needed here, since no operation will change the structure of the tree. So a static binary tree or even an implicitly represented complete binary tree stored in an array, as in a binary heap, would suffice.)

Now we support the operations:

- `initialize(`$\mathcal{M}$`)`: build `T` from the matrices in worst-case $\mathcal{M}$ in $O(|\mathcal{M}|) = O(n)$ time, maintaining the new augmentation from the leaves to the root.
- `update_joint(`k`, `M`)`: find the node `v` containing matrix k in the traversal order using `get_at(`k`)` at the root of T in $O(\log n)$ time, and replace `v.M` with M. Then recompute the augmentations up the tree in $O(\log n)$ time.
- `full_transformation()`: the augmentation stored at the root of `T` corresponds exactly to the arm's full transformation, so simply return `T.root.P` in $O(1)$ time.

Rubric:

- 2 points for a description of a correct database
- 1 point for a correct augmentation
- 2 points for description of a correct algorithm for each operation (3)
- 1 point for correct argument of correctness
- 2 points for correct argument of running times
- 3 points if correct database is efficient
- Partial credit may be awarded

Problem 4-6. [40 points] $\boldsymbol{\pi z^2 a}$ **Optimization**

Liza Pover has found a Monominos pizza left over from some big-TEX recruiting event. The pizza is a disc[6] with radius z, having n ***toppings*** labeled $0, \dots, n-1$. Assume z fits in a single machine word, so integer arithmetic on $O(1)$ such integers can be done in $O(1)$ time. Each topping i:

- is located at Cartesian coordinates (x_i, y_i) where x_i, y_i are integers from range $R = \{-z, \dots, z\}$ (you may assume that **all coordinates are distinct**), and
- has integer ***tastiness*** $t_i \in R$ (note, topping tastiness can be negative, e.g., if it's pineapple[7]).

Liza wants to pick a point (x', y') and make a pair of cuts from that point, one going straight down and one going straight left, and take the resulting ***slice***, i.e., the intersection of the pizza with the two half-planes $x \leq x'$ and $y \leq y'$. The tastiness of this slice is the sum of all t_i such that $x_i \leq x'$ and $y_i \leq y'$. Liza wants to find a ***tastiest*** slice, that is, a slice of maximum tastiness. Assume there exists a slice with **positive tastiness**.

[6] The pizza has thickness a, so it has volume $\pi z^2 a$.

[7] If you believe that Liza's Pizza preferences are objectively wrong, feel free to assert your opinions on Piazza.

(a) [2 points] If point (x', y') results in a slice with tastiness $t \neq 0$, show there exists $i, j \in \{0, 1, \dots, n-1\}$ such that point (x_i, y_j) results in a slice of equal tastiness t (i.e., a tastiest slice exists resulting from a point that is both vertically and horizontally aligned with toppings).

Solution: Pick the largest $x_i \leq x'$ and the largest $y_j \leq y'$. Because $t \neq 0$, the slice contains at least one topping, so there exists such toppings i and j. And since the slice S corresponding to point (x_i, y_j) contains exactly the same toppings as the slice corresponding to (x', y'), then slice S also has tastiness t.

Rubric:

- 2 points for a correct algorithm
- Partial credit may be awarded

(b) [8 points] To make finding a tastiest slice easier, show how to modify a Set AVL Tree so that:

- it stores **key–value items**, where each item x contains a value $x.val$ (in addition to its key $x.key$ on which the Set AVL is ordered);
- it supports a new tree-level operation `max_prefix()` which returns in **worst-case** $O(1)$ time a pair $(k^*, \text{prefix}(k^*))$, where k^* is any key stored in the tree T that maximizes the ***prefix sum***, $\text{prefix}(k) = \sum\{x.val \mid x \in T \text{ and } x.key \leq k\}$ (that is, the sum of all values of items whose keys are $\leq k$); and
- all other Set AVL Tree operations maintain their running times.

Solution: We augment the Set AVL Tree so that each node `v` stores three additional subtree properties:

- `v.sum`: the sum of all item values stored in `v`'s subtree, which can be computed in $O(1)$ time by: `v.sum = v.left.sum + v.item.val + v.right.sum`, or with zeros for the left and right sums if the respective children do not exist.
- `v.max_prefix`: $\max\{\text{prefix}(k) \mid k \in \text{subtree}(v)\}$. We can compute the max prefix in the subtree in $O(1)$ time by comparing three values:

```
v.max_prefix = max(
    v.left.max_prefix,                              # left
    v.left.sum + v.item.val,                        # middle
    v.left.sum + v.item.val + v.right.max_prefix)   # right
```

  where augmentations are considered zero on non-existent nodes.
- `v.max_prefix_key`: $\arg\max\{\text{prefix}(k) \mid k \in \text{subtree}(v)\}$. We can compute the maximizing key in $O(1)$ time based on which of the three cases is maximum in the above computation: key is `v.left.max_prefix_key` if the left is maximizing, `v.item.key` if the middle is maximizing, and `v.right.max_prefix_key` if the right is maximizing.

Because these augmentations can be computed locally in $O(1)$ time, they can be maintained without effecting the running times of the normal Set AVL Tree operations. To

support `T.max_prefix()`, simply return
`(T.root.max_prefix, T.root.max_prefix_key)`
in $O(1)$ time via the augmentations stored at the root.

Rubric:

- 3 points for correct augmentations
- 1 point for description of algorithm to support new operation
- 1 point for correct argument of correctness
- 1 point for correct argument of running time
- 2 points if correct implementation is efficient
- Partial credit may be awarded

(c) [5 points] Using the data structure from part (b) as a black box, describe a **worst-case** $O(n \log n)$-time algorithm to return a triple (x, y, t), where point (x, y) corresponds to a slice of maximum tastiness t.

Solution: Sort the input topping points by their x coordinates in $O(n \log n)$ time (e.g., using merge sort), and initialize the data structure T from part (b), initially empty. Then for each topping (x_i, y_i, t_i), insert it into T as a key-value item with key y_i and value t_i in $O(\log n)$ time, and then evaluate the max prefix (y^*, t^*) in T. The max prefix t^* is then by definition the maximum tastiness of any slice with x coordinate x_i, specifically the slice corresponding to the point (x_i, y^*). By repeating this procedure for each topping sorted by x, we can compute the maximum tastiness of any slice at x_i for every x_i in $O(n \log n)$ time (along with its associated point). Since some slice of maximum tastiness exists with an x coordinate at some x_i for $i \in \{0, \ldots, n-1\}$, as argued in part (a), then taking the maximum of all slices found in $O(n)$ time will correctly return a tastiest slice possible.

Rubric:

- 3 points for description of a correct augmentations
- 1 point for correct argument of correctness
- 1 point for correct argument of running time
- Partial credit may be awarded

(ci) [25 points] Write a Python function `tastiest_slice(toppings)` that implements your algorithm from part (c), including an implementation of your data structure from part (b). You can download a code template containing some test cases from the website.

Solution:

```python
class Part_B_Node(BST_Node):
    def subtree_update(A):
        super().subtree_update()
        A.sum = A.item.val                          # sum
        if A.left:  A.sum += A.left.sum
        if A.right: A.sum += A.right.sum
        left = -float('inf')                        # max prefix
        right = -float('inf')
        middle = A.item.val
        if A.left:
            left = A.left.max_prefix
            middle += A.left.sum
        if A.right:
            right = middle + A.right.max_prefix
        A.max_prefix = max(left, middle, right)
        if left == A.max_prefix:                    # max prefix key
            A.max_prefix_key = A.left.max_prefix_key
        elif middle == A.max_prefix:
            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):
        k = self.root.max_prefix_key
        s = self.root.max_prefix
        return (k, s)

def tastiest_slice(toppings):
    B = Part_B_Tree()    # use data structure from part (b)
    X, Y, T = 0, 0, 0
    n = len(toppings)
    toppings.sort(key = lambda topping: topping[0])
    for (x, y, t) in toppings:
        B.insert(Key_Val_Item(y, t))
        (Y_, T_) = B.max_prefix()
        if T < T_:
            X, Y, T = x, Y_, T_
    return (X, Y, T)
```