Problem 1: In class we saw how to use AVL trees to maintain a dynamic set on which you need to perform the following operations: Insert, Delete, Search, Successor. All these operations run in $O(\log n)$ on a tree with $n$ elements.

We now want to add the procedure Rank. When we call Rank($x, i$) it returns the $i^{th}$ element in the tree rooted at $x$.

Show how to implement this operation in $O(\log n)$ time on a modified version of AVL trees.

Solution: We are going to add two fields to each node $x$: the field $NL(x)$ stores the numbers of nodes in the left subtree of $x$, and similarly $NR(x)$ the number of nodes in the right subtree of $x$. With this information it is easy to answer Rank queries. When you call Rank($x, i$) you check the following

- if $i = NL(x) + 1$ then the answer is $x$ (since there are exactly $i - 1$ elements smaller than $x$ in the tree)
- if $i < NL(x) + 1$ then we recursively call Rank($L(x), i$) since the $i^{th}$ element in the tree rooted at $x$ is also the $i^{th}$ element in the tree rooted at $L(x)$
- if $i > NL(x) + 1$ then we recursively call Rank($R(x), j$) where $j = i - NL(x) - 1$ since the $i^{th}$ element in the tree rooted at $x$ is now the $j^{th}$ element in the tree rooted at $R(x)$

Since we call Rank once at each level of the tree, the procedure will take $O(\log n)$ on a balanced AVL tree, assuming we can maintain the additional fields during insertion and deletions.

For insertion this is done by adding 1 to the appropriate fields on the path of insertion. Similarly for deletion, you have to subtract 1 from the appropriate fields on the path from the root to the node that was removed (remember that in the case of deletion the node that is actually removed from the tree is always a leaf or a node with one child). This will take $O(\log n)$ as well.

The last thing to do is to make sure that we can maintain these fields during rotations. But that’s also easy since obviously the number of nodes at the subtree rooted at $x$ is $NL(x) + NR(x) + 1$ so it is easy in $O(1)$ time to fix the value of the fields at the nodes whose left and right child links are modified during a rotation.

Problem 2: The Brocard point of a triangle ABC is the point P in the triangle chosen so that $\angle PAB = \angle PBC = \angle PCA$ as described in the figure below.

The common angle is called the Brocard angle. The largest possible Brocard angle is $\pi/6$ for the case of an equilateral triangle.

Given the coordinates of the vertices as input, how would you compute the coordinates of the Brocard point?

Solution: This problem was assigned last year at the ACM programming contest. One simple way to do this would be to use binary search on the angle space. Start with $\pi/12$ for the $PAB$ and $PBC$ angles (this uniquely determines the point $P$) and check what $PCA$ is and then go higher or lower.

Problem 3: City College is holding a fundraiser at the XYZ corporation. Each employee $x$ at XYZ as pledged to contribute $c_x$ to City College if invited to the fundraiser, otherwise he/she will contribute nothing.

The employees at XYZ are organized according to a hierarchical structure which we can represent as a tree. The XYZ CEO is at the top and the children of each node are its direct subordinates.

To avoid fights at the party and make sure that people can talk and drink freely, City College does not want to invite an employee and his/her direct supervisor to the party.

Your job is to come up with a guest list that satisfies the above constraints and maximizes the amount of donations that City College will receive at the party.
Solution: Consider the tree that stores the employees at XYZ. Each node $x$ also contains the field $c(x)$ which is the amount pledged by $x$. We also add another field which is $opt(x)$, the optimal amount raised by the employees in the subtree rooted at $x$. Obviously for the leaves in the tree $opt(x) = c(x)$. Now let’s focus on the nodes at level 1 (i.e. right above the leaves). If $x$ is such a node, the optimal solution will be the max between $c(x)$ and $\sum_y opt(y)$ for $y$ running over all the possible children of $x$. We now have the optimal solution computed at level 0 and level 1. Note that you can also store a bit $inv(x)$ on node $x$ denoting if that node has been invited or not in the optimal solution (which you set to the correct value, depending on the choice you make above).

To continue filling the tree we denote with $C(x)$ be the set of children of $x$, and $G(x)$ be the set of grandchildren of $x$. Then in general we have that $opt(x)$ is the maximum between

$$c(x) + \sum_{y \in G(x)} opt(y) \text{ if we invite } x$$

$$\sum_{y \in C(x)} opt(y) \text{ if we do not invite } x$$

This will allow us to fill the tree in a bottom up fashion with the optimal solution up to the root. To keep track of whom to invite we need to maintain at each time two copies of the currently filled tree. One with the ”invited bits” set for the optimal solution up to the grandchildren of $x$ and one with the ”invited bits” set for the optimal solution up to the children of $x$. This will allow us to continue with the correct invitation list as we fill the tree using the choice above.