CS 201c: Data Structures
Programming Evaluation 2 Solutions

First, we store a single float *curr*, which denotes the current time.

Key idea:

Let (r_1, x_1, t_1, v_1), (r_2, x_2, t_2, v_2), …, (r_n, x_n, t_n, v_n) be the configuration of cars on the highway at current time. [Here x_1 < x_2 < … < x_n are x-coordinates of the cars with registration numbers r_1, r_2, …, r_n at (last insert) times t_1, t_2, …, t_n respectively. Further, the velocities of these cars are v_1, v_2, …, v_n.]

Let u_1, u_2, …, u_{n-1} be a sequence of future times with the following property:

> u_i is the unique future time at which cars r_{i} and r_{i+1} have the same x-coordinate. To be specific,
>
> (i) u_i = *curr*+(x_{i+1} - x_{i})/(v_{i+1} - v_{i}) if v_{i} > v_{i+1}, and
> (ii) u_i is equal to infinity, if v_{i} <= v_{i+1}.

Further, if u_i is finite, then the x-coordinate e_i of the meeting point of cars r_{i-1} and r_i at future time u_i is equal to:

> e_i = x_i + (u_i - t_i)*v_i

Let t be the *earliest* future time such that there are at least two cars on the highway, which currently have different x-coordinates, but will have the same x-coordinate at time t. Then,

> t = min { u_1, u_2, …, u_{n-1} }

Implementation:

Recall the solution of Programming Evaluation 1. Out of the three balanced BSTs there, we maintain only two trees T1 and T3 (since all cars now are traveling in the same direction - from left to right):

- The first BST T1 stores the cars (currently on the highway) with their registration id as the key. Further, for each registration id r in T1, a pointer to the unique node corresponding to car r in tree T3 is also stored.
- The second BST T3 also stores the cars currently on the highway. Further, these cars are now essentially ordered according to their current x-coordinates on the highway. Note that, in addition to registration number r and the pair (x,t), we now have to also store a velocity field v with every car entry in both these trees.

The comparison function for the cars stored in BST T3 is defined as follows (to also incorporate velocity field). Consider two entries: (r1, x1, t1, v1) and (r2, x2, t2, v2) in tree T3.

Then,

(r1, x1, t1, v1) < (r2, x2, t2, v2)

if and only if

at least one of these two conditions are true:

(i) [x1+v1*(curr-t1)] < [x2+v2*(curr-t2)],
(ii) [x1+v1*(curr-t1)] == [x2+v2*(curr-t2)] AND v1 < v2.

[Thus, cars are ordered by increasing x-coordinates of their current locations. If two cars have the same current location, they are ordered among themselves in increasing order of velocities. Note that we have assumed here that if any two cars have the same x-coordinate (at any point of time), they are guaranteed to have different velocities.]

In addition to the 4-tuple (r,x,t,v), we also store an extra field h (heap pointer) for each car in tree T3, which points to the car pair in heap H, where car r is the left car (see definition of H below). [If no such car pair exists in heap H, this pointer is equal to NULL.]

(You can reuse your code for Programming Evaluation 1 for the above.)

In addition to the above two data structures, we also maintain a min-heap H. At any point of time, the heap H has at most n-1 entries (we do not keep entries for which u_i is infinity):

(r_1, r_2, u_1, e_1), (r_2, r_3, u_2, e_2), …, (r_{n-1}, r_n, u_{n-1}, e_{n-1})

(Here r_i's, u_i's, and e_i's have the meaning as defined in the Key Idea above.)

H is a min-heap ordered according to the following comparison function (for i not equal to j):

(r_{i}, r_{i+1}, u_i, e_i) < (r_{j}, r_{j+1}, u_j, e_j)

if and only if

exactly one of these three conditions is true:

(i) u_i < u_j,
(ii) [u_i == u_j] AND [e_i < e_j]
(iii) [u_i == u_j] AND [e_i == e_j] AND velocity of car r_{i} is greater than the

velocity of car r_{j}

[The above comparison function ensures that car pairs are first ordered in increasing order of their (future) meeting times. If two car pairs have the same (future) meeting time, they are ordered in increasing order of the x-coordinates of their (future) meeting point. Finally, if two car pairs have the same meeting time as well as the same meeting location, they are ordered in decreasing order of the velocities of the left cars in the respective pairs.]

Further, as noted above, there is cross-referencing between the BST T3 and the heap H:

> *A pointer h is maintained from the entry for car r in tree T3, to the car pair entry in heap H (if it exists) which has r as the left car.*

Outline of individual functions:

insert(int r, float x, float v):

Check for registration number r in tree T1.

If car r is found, return 0.

```
If car r is not found:
    Insert details of car r in tree T1.
    Insert details of car r  in tree T3.
    Let r_prev be the inorder predecessor of car r in tree T3.
    Let r_next be the inorder successor of car r in tree T3.
    (r_prev and r_next can be NULL.)

    If both r_prev and r_next are not NULL, remove entry for the consecutive car pair
    (r_prev, r_next) from heap H. [You can find this entry in heap H, by following the pointer
    to H from the entry for car r_prev in tree T3.]

    If r_prev is not NULL, insert suitable entry for the new consecutive car pair (r_prev, r)
    into heap H. Further, set the pointer from entry for car r_prev in T3 to this new entry in
    heap H.

    If r_next is not NULL, insert suitable entry for the new consecutive car pair (r, r_next) into
    heap H. Further, set the pointer from entry for car r_next in T3 to this new entry in heap
    H.

    Return 1.
```

remove(int r):

```
Check for registration number r in tree T1.

If car r is not found, return 0.

If car is found:
        Find details of car r in tree T3.

        Let r_prev be the inorder predecessor of car r in tree T3.
        Let r_next be the inorder successor of car r in tree T3.
        (r_prev and r_next can be NULL.)

        If r_prev is not NULL, remove entry for the pair (r_prev, r) from heap H.
        If r_next is not NULL, remove entry for the pair (r, r_next) from heap H.
        [You can find these entries in heap H, by following the pointers to H (stored in field h)
        from the entries for car r_prev and r in tree T3 respectively.]

        If both r_prev and r_next are not NULL, insert suitable entry for the new consecutive car
        pair (r_prev, r_next) into heap H. Further, set the pointer from entry for car r_prev in T3
        to this new entry in heap H.

        Remove entry for car r from both trees T1 and T3.

        Return 1.
```

int next_crossing():

```
        If H is empty, return 0.

        Delete the smallest element in heap H. Let it be (r1, r2, u, e).

        Let v1 and v2 be the velocities of cars r1 and r2 respectively.

        Call remove(r1).
        Call remove(r2).

        Make a first empty balanced BST U, ordered by registration numbers.
        For each registration number, U also stores location (at future time u) and
        velocity of the car.

        Insert two entries (r1, e, v1) and (r2, e, v2) in BST U.
```

Make a *second* empty balanced BST V, ordered by registration numbers. V stores only registration numbers, and no other auxiliary information.

Insert two entries r1 and r2 in BST V.

[While loop is in blue font.]

```
While ( the current minimum entry in heap H has meeting time equal to u ):

    Delete the minimum entry (r_new1, r_new2, u, e_new) from heap H.

    Let v_new1 and v_new2 be the velocities for cars r_new1 and r_new2 respectively.


    Call remove(r_new1).
    Call remove(r_new2).
    [Again if a registration number does not exist in trees T1 and T3, no action is taken.]

    Insert two entries (r_new1, e_new, v_new1) and (r_new2, e_new, v_new2) into BST U. (If a registration number already exists in tree U, the entry is not added to the tree.)

    if (e_new == e)
        Insert r_new1 and r_new2 in BST V. [If any registration number already exists in the tree, no action is taken.]

    if (e_new > e)

        Print e followed by an inorder traversal of tree V.
        [The above line prints all cars which meet at location e at future time u, in increasing order of registration numbers.]

        Set r1=r_new1, r2=r_new2, and e=e_new.

        Set V to empty tree.
        Insert r1 and r2 in tree V.
```

Print e followed by an inorder traversal of tree V.
[Again, the above line prints all cars which meet at location e at future time u, in increasing order of registration numbers.]

```
    Set *curr*=u.

    For each entry (r, e, v) in BST U:
        Call *insert(r, e, v)*.
    Return 1.
```

Note. You have to write a template class for a balanced BST only once. The four trees T1, T3, U, and V can all be implemented by instantiating this BST template. In fact, you can reuse the balanced BST template written for Programming Evaluation 1.