Name : Venkata Satya Sai Subhash Vadlamani

UFID : 4326-8265

Email : v.vadlamani@ufl.edu

Introduction :

I have developed a code for a ride sharing software that can be used to satisfy multiple ride requests at a time. A ride is uniquely identified by three attributes:

1) rideNumber : Unique Integer identifier for each ride.
2) rideCost: The estimated cost(in integer dollars) for the ride.
3) tripDuration : the total time(in integer minutes) needed to get from pickup to destination.

I used a Min Heap to store the triplets of each and every ride (rideNumber, rideCost, tripDuration) based on the rideCost and we use a Red Black Tree to store the triplets of each and every ride (rideNumber, rideCost, tripDuration) based on the rideNumber. I also maintain the pointers between the nodes of the RBT and the Min Heap.

I have implemented the following functions and will explain the time complexities of the following functions.

1) Print(rideNumber): prints the triplet (rideNumber, rideCost, tripDuration)
    a. To implement this function, we just access the Red Black Tree which is a form of a binary search tree.
    b. We already know that the Red Black Tree is ordered based on the rideNumber. Hence, to get the triplet, we just perform an operation similar to the search on a binary search tree.
    c. If we find the triplet in the Red Black Tree, we are going to return the triplet and if it is not found, we are going to return the following triplet (0, 0, 0).
    d. Since the time complexity of the search on a binary search tree(Red black Tree in this case) is O(log n), 'n' is the number of nodes in the Red Black Tree(Number of active rides), the time com plexity of the print(rideNumber) function is O(log n). The space complexity is O(1) because we are not creating any new nodes.

2) Print(rideNumber1, rideNumber2) : prints all triplets (r_x , rideCost, tripDuration) for which rideNumber1 <= r_x <= rideNumber2.
    a. The actual searching of the nodes in the Red Black Tree is being performed by the function 'find_rides_in_range' function that is present in the red_black_tree.py file. Here, we notice that the recursive function is being called in the part of the tree which

might contain the rides. Hence, we can see that the RBT tree search is entering the part of the RBT tree that might have the ride. Hence, we are avoiding unnecessary searches.

b. We continuously append the rides that we found to the result list and we then return the result list.
c. Thus, the time complexity of this function is O(log n + S) where 'n' is the number of actie trips and 'S' is the number of rides that lie in the range 'rideNumber1' to 'rideNumber2'. The space complexity is O(1) because we are not creating any new nodes.

3) Insert(rideNumber, rideCost, tripDuration) : Here, the 'rideNumber' differs from any rideNumbers that exist in the system already.
    a. There is a method called 'insert_ride' in the gator_taxi.py file which is responsible for inserting the ride into both the Red Black Tree and the Min Heap. First, we check if there already exists a ride in the Red Black Tree with the 'rideNumber'. If it already exists, we do not insert the new ride and print the line 'Duplicate RideNumber' . If there does not already exist a ride in the Red Black tree with the 'rideNumber' we proceed forward with the insertion.
    b. Now, We create a Min Heap Node and the Red Black Tree node and maintain the pointers from the node of one data strucuture and another data structure.
    c. Then, we proceed forward with the insertion into the min heap and the red black tree.
    d. The insertion functions of the min heap and the red black tree are located in the functions named 'insert' in 'min_heap.py' and 'red_black_tree.py'.
    e. We know that the insertion of the nodes in the min heap and the red_black_tree have the time complexity of O(log n) ('n' -> Number of active trips in the system).
    f. Thus, the time complexity of the Insert(rideNumber, rideCost, tripDuration) is O(log n). The space complexity is O(1) since we are inserting just a single node in both the min heap and the red black tree.

4) GetNextRide(): This function outputs the ride with the lowest cost. If two rides have the same lowest cost, the ride with the lowest trip duration among those rides is outputted. Then, we delete this ride from both the min Heap and the Red Black Tree.
    a. This function is implemented by the function with the name 'get_next_ride' in the gator_taxi.py file. We first check if there exist any elements in the min heap. If there are not elements in the min heap, we output that the min heap is currently empty.
    b. We already know that the root of the min heap represents the next ride. Thus, we pop the element from the min heap, adjust the min heap elements such that the min heap properties are still maintained(every node is smaller than it's children). The time complexity of this operation is O(log n)
    c. After this, we take the node that is returned when we perform the delete on the min heap to delete the corresponding node from the Red Black Tree.
    d. After the deletion of the node from the Red Black Tree, we have to perform balancing of the Red Black tree to make sure that the Red Black Tree properties remain. The time complexity of this operation is O(log n).

e. Since both the operations, deletion of the node from the min heap and the deletion of the node from the Red Black Tree take O(log n) time, the time complexity of GetNextRide() method is also O(log n). The space complexity is O(1) since we are just deleting one one from the red black tree and the min heap.

5) CancelRide(rideNumber): This method is used to delete the trip whch is identified by the 'rideNumber' from both the min heap and the red black tree. If there does not exist a trip with the corresponding 'rideNumber' we just ignore the method call CancelRide(rideNumber).
    a. The functionality of this method is present in the cancel_ride() method in the gator_taxi.py file.
    b. Here, we first delete the node from the Red Black Tree(since the nodes in the red black tree are ordered based on the ride number, it is easy to find that ride in the Red Black Tree first). After this, we use the node that is returned when we perform the delete operation on the red black tree to delete the corresponding node in the Min Heap.
    c. In Red Black Tree, after the deletion of a particular node, we also perform the balancing of the tree to make sure that the properties of the Red Black Trees still hold true. The deletion of the node from the Red Black Tree has the time complexity of O(log n)
    d. To delete the the element from the min heap, we basically swap the last element of the min heap with the element that we are to be deleting, then decrease the size of the heap by one and then balance the heap. This operation takes O(log n) time.
    e. Since both the of the sub operations within the CanceRide(rideNumber) take place one after another and the time complexities of these operations is O(log n), the time complexity of the CancelRide(rideNumber) operation is also O(log n) where 'n' is the number of active trips in the system at that time.. The space complexity is O(1) because we are just deleting one node from both the min heap and the red black tree.

6) UpdateTrip(rideNumber, new_tripDuration) -> This happens when the rider wishes to change the destination.
    a. Here, if the new trip duration is less than or equal to the old trip duration, we just update the trip duration without changing the cost
    b. If the new trip duration is less than twice the old trip duration, we delete the old trip from the system and we create a new trip with the cost of new $10 more than the old cost and we also use the new trip duration.
    c. If the new trip duration is more than twice the old trip duration, then we just cancel the the trip with the current ride number.
    d. In the first case, we just change the triplet in the min heap based on the new trip duration. This just involves looking for the node and updating it and rearringing the min heap so that the min heap properties still remain true. The time complexity of this operation is O(log n)
    e. The other operations involve the operations that we have already discussed, 'Insert(rideNumber, rideCost, tripDuration)' and 'CancelRide(rideNumber)'. Both these operations take O(log n) time complexity.

f. Thus, the overall time complexity of the UpdateTrip(rideNumber, new_tripDuration) is also O(log n). The space complexity is O(1) since we are just updating at max one node in both the min heap and the red black tree.

I have clearly listed down the time complexity and space complexity of each and every function that we were asked to implement. I have also explained the reasoning behind the time complexities.