Augmented Data Structure

Jin Long Cao

November 2022

GutSend (aka GS) contracts drivers to pickup and deliver fast food orders. They want you to produce software to keep track of the details. They need you to design:

GS-add(new-driver): Add a new driver with an initially empty list of deliveries to GS's contractors. Return a unique identifier `driver-ID`

GS-schedule(order-ID, current-time): Add `order-ID` and the `current-time` (the time `GS-schedule` is executed) to the list of deliveries for the driver with a shortest list. If there are two or more drivers with a shortest list, arbitrarily add the order to one of them. Return the `driver-ID` of the driver contracted to deliver this order.

GS-next(driver-ID): Remove the order with the earliest `current-time` recorded from `driver-ID`'s list. Again, for ties arbitrarily choose one. Return the removed order's `order-ID`.

GS-order(order-ID): Return what to pick up, from where, and where to deliver it.

GS-quit(driver-ID): Apparently driver `driver-ID` has gotten disgusted by having no employee rights under Ontario labour law, and has quit. Merge `driver-ID`'s list with those of another driver with a shortest list of deliveries, other than `driver-ID`. Return the merged driver's `driver-ID` or "fail" if there is no such driver.

GS would like the following worst-case run-time characteristics:

GS-add: If there are m drivers, $\mathcal{O}(\lg m)$.

GS-schedule: If there are m drivers, and a shortest list has n orders, $\mathcal{O}(\lg mn)$.

GS-next: If driver `driver-ID` has n orders, and there are m drivers, $\mathcal{O}(\lg nm)$.

GS-order: $\mathcal{O}(1)$.

GS-quit: If `driver-ID` has n_1 orders, there are m drivers on the list after `driver-ID` quits, the shortest list after `driver-ID` quits has n_2 orders, and $n_3 = \max(n_1, n_2)$, then $\mathcal{O}(\lg mn_3)$.

Solutions

GS-add(new-driver):

We'll implement this using a Priority Queue ADT (P_1) with heap size, heap shape, and min-heap order. We'll also have a counter variable (starting at 1) such that it every time we add a new driver, this number increases by 1. When we get a new driver, we'll increase heap size by 1, the counter value will be the new-driver's driver-ID, and increment the counter. Since the counter is increasing, min-heap order will be maintained (cost $O(1)$). Also we'll set a satellite value (driver-ID.list) which will be the initial empty list of deliveries (which cost $O(1)$). For later conveniences, we'll also add another satellite value (driver-ID.size) which will tell us the size of the list (initially set to 0). We'll insert this driver-ID.size into another Priority Queue ADT (P_2) with heap size, heap shape, and max-heap order. The node has a variable that keeps track of the driver-ID. Finally, return the unique identifier driver-ID.

GS-schedule(order-ID, current-time):

recall: $\lg(mn) = \lg(m) + \lg(n) \Rightarrow O(\lg m) + O(\lg n) = O(\lg m + \lg n) = O(\lg mn)$
For worst-case run-time, it cost $O(\lg n)$ at most to extract minimum from P_2 (which gives us driver-ID of the driver with the shortest list of n orders).
$O(\lg m)$ to find driver with a shortest list out of whole driver tree (P_1). Current-time will be a satellite value for order-ID and order-ID is in a list of other order-ID in driver-ID.list.
Then, we'll add order-ID and order-ID.current-time to driver-ID.list to the driver with the shortest list, increment driver-ID.size. For later conveniences, we'll insert order-ID.current-time into another Priority Queue ADT (P_3) in ascending order where the latest order-ID.current-time has the highest priority. The node has a variable that keeps track of the order-ID. Which takes no more than $O(\lg n)$ (for n orders).
Also every time we schedule an order, we'll insert the order's information into a dynamic array (called order) which only cost a constant (e.g. order-ID 3 picks up clothes, from Walmart, and delivers to 123 random street. Then order[3] = "picks up clothes, from Walmart, and delivers to 123 random street."). In general, order[order-ID] = Description of order-ID, what to pick up, from where, and where to deliver it. Two orders should not have the same order-ID, if it does, the newer one replaces the later one.
Hence, if there are m drivers, and a shortest list of n orders then it would take $O(\lg n)$ to find the driver with the shortest list and it will take $O(\lg m)$ to find such driver.

$$\begin{aligned} O(\lg m) + O(\lg n) &= O(\lg m + \lg n) \\ &= O(\lg mn) \end{aligned}$$

It will take $O(\lg mn)$ to satisfies such task. Finally, return the driver-ID of the driver contacted to deliver such order.

GS-next(driver-ID):

It costs $O(\lg m)$ to find driver-ID out of all the drivers from P_1 (where there are m drivers).
Now, we want to find the order-ID with the earliest current-time recorded (out of n orders) in $O(\lg n)$. This can be done my extracting minimum element from P_3 which takes at most $O(\lg n)$.
Hence, if driver driver-ID has n orders, and there are m drivers, then it would take $\lg m$ to find the specific driver and $\lg n$ to find the order-ID of the specific driver's earliest order-ID. Since the order-ID is found, removing and returning the removed order's order-ID will only take a constant.

$$\begin{aligned} O(\lg n) + O(\lg m) &= O(\lg n + \lg m) \\ &= O(\lg mn) \end{aligned}$$

It will take $O(\lg mn)$ to satisfies such task. Finally, return the removed order's order-ID.

GS-order(order-ID):

In GS-schedule, every time we scheduled something, the order's information has been added to a dynamic array (called order). Just like an normal java or python list/array, to insertion and searching in an array cost $O(1)$. It's like setting a variable to something. No matter how big the array is or how many drivers there is, it will always cost $O(1)$ to search for the information of an order-ID. Return order[order-ID] to return what to pick up, from where, and where to deliver it of order's order-ID.

GS-quit(driver-ID):

There's two cases that could happen,

1. If shortest list is the driver that quits (such that $n_1 \leq n_2$) then $n_3 = n_2$ and the worst-case run-time cost $O(\lg n_2) = O(\lg n_3)$ at most to extract minimum from P_2 (which gives us driver-ID of the driver with the shortest list of n_2 orders)

2. If the shortest list is not the driver that quits (such that $n_1 \geq n_2$) then $n_3 = n_1$ and the worst-case run-time cost $O(\lg n_1) = O(\lg n_3)$ at most to extract minimum from P_2 (which gives us driver-ID of the driver with the shortest list of n_2 orders)

Notice that in any case it takes $O(\lg n_3)$ to find the driver with the shortest list and $O(\lg m)$ to find driver with a shortest list out of whole driver tree (P_1).
Hence, if driver-ID has n_1 orders, there are m drivers on the list after driver-ID quits, the shortest list after driver-ID quits has n_2 orders, and $n_3 = \max(n_1, n_2)$, then

$$\begin{aligned} O(\lg \max(n_1, n_2)) + O(\lg m) &= O(\lg n_3) + O(\lg m) \\ &= O(\lg mn_3) \end{aligned}$$

It will take $O(\lg mn_3)$ to satisfies such task. Finally, return the merged driver's driver-ID.