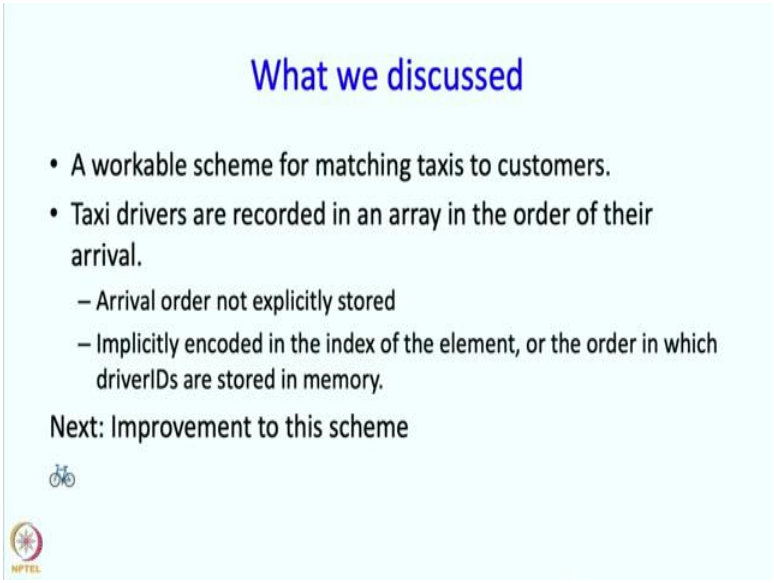
An Introduction to Programming through C ++

Professor. Abhiram G. Ranade
Department of Computer Science and Engineering,
Indian Institute of Technology Bombay.

Lecture 15

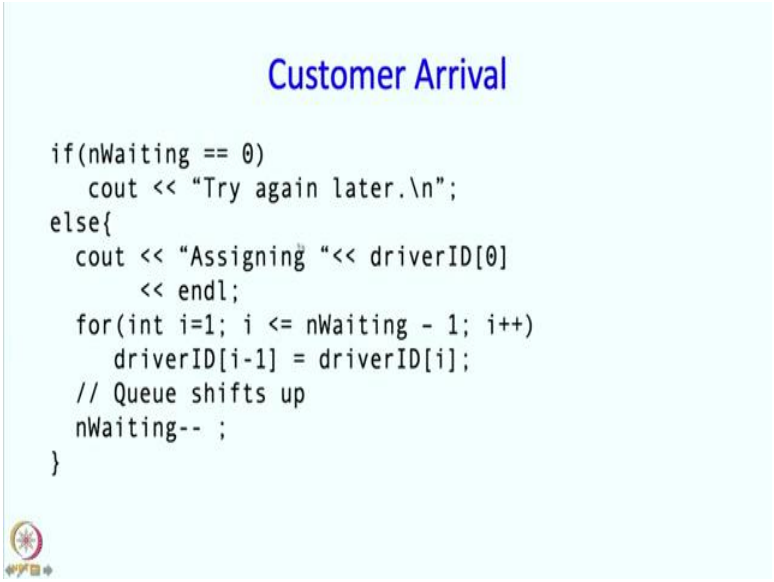
Array part-1 More efficient Queues in dispatching taxis

(Refer Slide Time: 00: 22)



Welcome back in the last segment we discussed a scheme for dispatching taxis that is matching taxis to customers. Now in that scheme, first of all that scheme was perfectly fine in the sense that it did the job but it's seems that the scheme can be improved so let us see why.

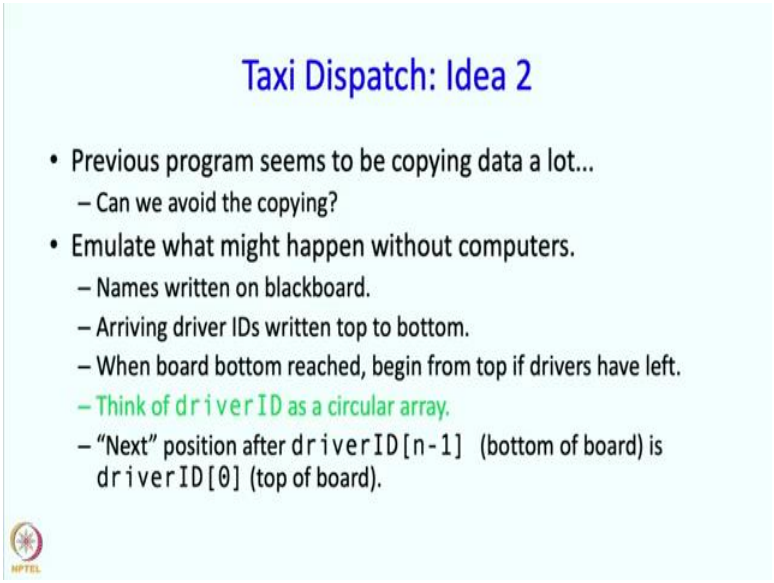
(Refer Slide Time: 00: 44)



So, here, what we did was after a customer came in we assigned the waiting drivers but then after that we sort of moved all waiting drivers one step forward this seems like a little bit of

unnecessary thing. So, you will see that can essentially eliminate this movement. So, there might be a maybe say 400 drivers waiting. So, when we are doing these four hundred moves so why should we do that. So, you will see now that we can do without actually moving so many drivers forward.

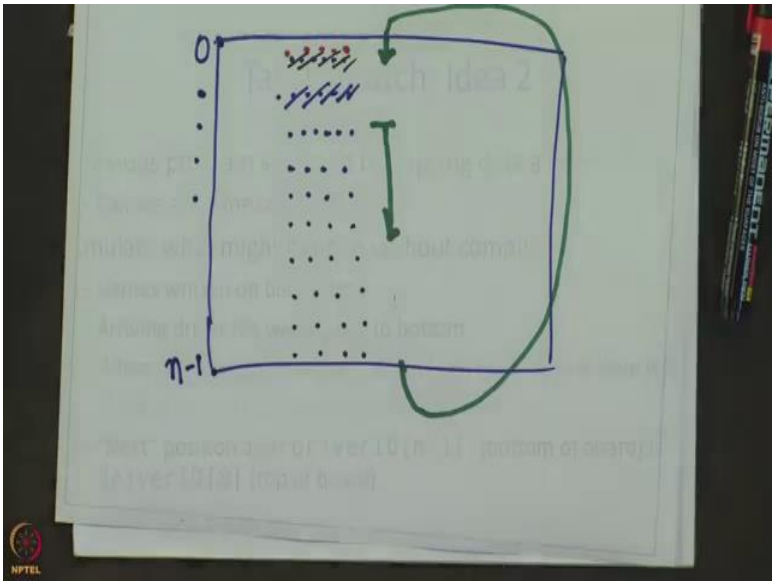
(Refer Slide Time: 01:30)



So, here is the new idea. So, the previous program seems to be copying data a lot. So, we are going to avoid copying. So, for this we are going to look at what might happen without computers. So, this is actually something that you should think about any way. So, if you are trying to solve a problem and practically anything that you do in this course you are going to do something which you probably would do manually. You are not going to do something which is terribly different from what you do manually.

So, therefore figuring out look what would I do manually. So, in this case would I do the equivalent of that copying manually that is something that you should ask. What would you do manually. So, maybe you might write the names on a black board. So, what will it look like.

(Refer Slide Time: 02: 27)



Taxi Dispatch: Idea 2

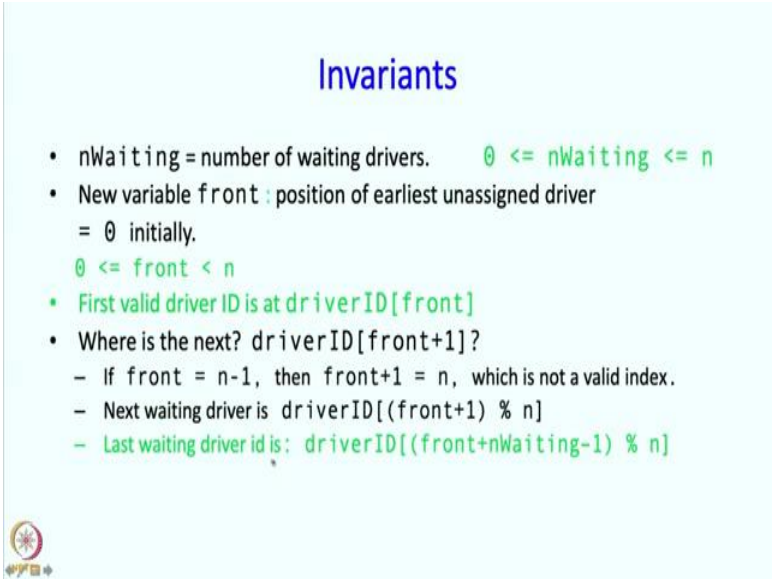
- Previous program seems to be copying data a lot...
  - Can we avoid the copying?
- Emulate what might happen without computers.
  - Names written on blackboard.
  - Arriving driver IDs written top to bottom.
  - When board bottom reached, begin from top if drivers have left.
  - Think of `driverID` as a circular array.
  - "Next" position after `driverID[n-1]` (bottom of board) is `driverID[0]` (top of board).

So, this is my black board and then I start righting the names of the arriving drivers from the top. So, I wrote that and maybe at some point a customer comes. So, what will I do? I will take the first driver over here and give it to the customers and maybe I will erase this name on the black board. So, I cannot erase when I write it in pen and paper but if it is actually a black board I can wipe it off. So, I am not going to erase this and push everyone up I am just going to erase it and say that look now really the front of the queue is over here. If one more customer comes I will erase a little bit more and I will keep going in this manner but suppose a new driver comes. So, then I will put down driver information over here all the way till this point. What happens if I come down all the way and a new driver comes well in that case I may start from the top - the new driver information would be put at the top. So, now this is an

interesting way in which the information is given over here. The information starts from over here goes down and then continues from the top. So, you can think of this as some kind of a circular queue so it's sort of starts over here and comes back over here. So, think of that black board as sort of a rolling circular board. So, because it is circular board we need to worry a little about where is the first person. So, we know that the first person is over here if that person if the driver is given to a customer then the first person becomes the next but we are really not shifting up the driver names. So, that is exactly what we are going to do in our solution.

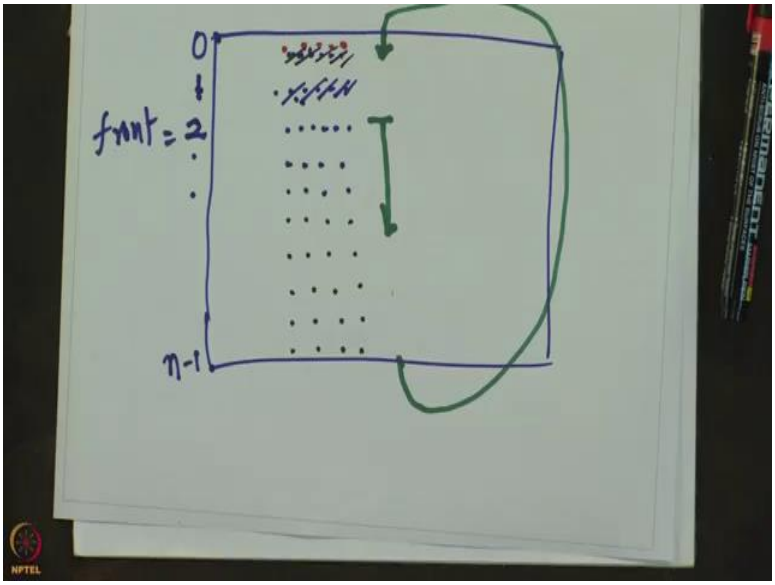
So, arriving driver ID is written top to bottom when board bottom reached we began from top if the drivers have left of course if the drivers have not left then we cannot do anything we have to tell the new driver that look my queue is full I do not have space to write down your number and so please wait to please try a little bit later. But usually this will not happen because we will allocate reasonably large amount of space anyway. So, we are going to think of this driver ID as a circular array what that means it that the next position after driverID[n-1] so, this is index 0 this is index n minus 1 so after this position then the next position is this after this the next position is this and so on. But after this normally you might tempted to say that there is no next position but no in this case we are going to say the next position is this so it's kind of a circular array. So, that is basically the idea that we are going to implement and again the idea it is not completely simple, it's more complicated from the previous idea and therefore we should make decisions sensibly at the beginning and then stick to those decisions. So, that's we are going to do next. So, we are going to say what our invariants are.

(Refer Slide Time: 06:27)



So, as before we are going to have nWaiting denote the number of waiting driver. So, that variable we are still going to have we have to keep track of how many drivers are waiting and we are also going to have this invariant that nWaiting is going to lie between 0 and n but now we are going to have a new variable called front. The front is the position of the earliest unassigned driver.

(Refer Slide Time: 06:58)



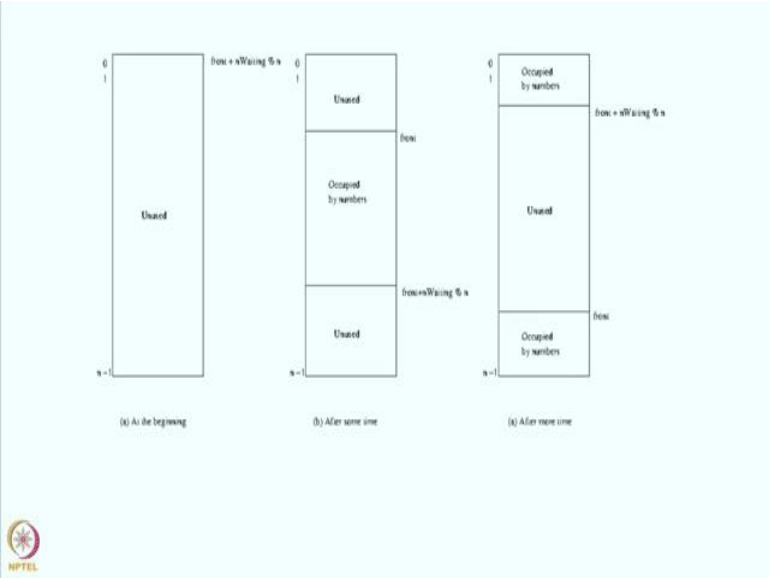
Invariants

- nWaiting = number of waiting drivers. `0 <= nWaiting <= n`
- New variable `front` : position of earliest unassigned driver
  = 0 initially.
  `0 <= front < n`
- First valid driver ID is at `driverID[front]`
- Where is the next? `driverID[front+1]`?
  - If `front = n-1`, then `front+1 = n`, which is not a valid index.
  - Next waiting driver is `driverID[(front+1) % n]`
  - Last waiting driver id is: `driverID[(front+nWaiting-1) % n]`

So, if you look at this picture. So, let us say these two drivers which we have assigned the position 0 and position 1 these drivers have gone away because they have been given customers. So, then this position two is the first position where there is an unassigned driver. So, really front will be equal to 2 at this time in our entire execution. So, front is index of the earliest unassigned driver so it's 0 initially. But as driver leaves this front increases but we want front also to be between 0 and n. Well this time we want front not to equal n because front is not the number it is an index. So, front cannot be n so it can be most n minus 1. So, front is an index so it can start at it can be 0 all the way till n minus 1. Then the first valid driver ID is at driverID[front]. So, that is what we said now this raises an interesting question where is the next driver ID should we say that it is at front plus 1 the next driver ID shall we

say that it is the element driver ID of front plus 1 well there is a bit of problem if front is n minus 1 as it can will be because after all we have said that front can be n minus 1. So, if it is indeed n minus 1 then front plus 1 of n which is what we are asking over here is n and that is not a valid index. So, if front is n minus 1 what we would want this to be we would want this to be 0 as we said if this is front then the next one from here is 0. So, therefore we are going to say that the next waiting driver is at (front plus 1) mod n. So, this plus 1 mod N is like going forward on a circular array. So, if we keep doing that may be I should explain this. So if, front is n minus 1 then front plus 1 is n but n mod n is 0 which is exactly what we wanted. So, in this by the same logic last waiting driver ID is not at front plus nWainting minus 1, which is what would be the case if this was not a circular array because this is a circular array we have to take a mod n. So, these are the things in green are the proposition that we have to care about, these are the invariants. So, we have to make sure that these statements are always going to hold.

(Refer Slide Time: 10:12)

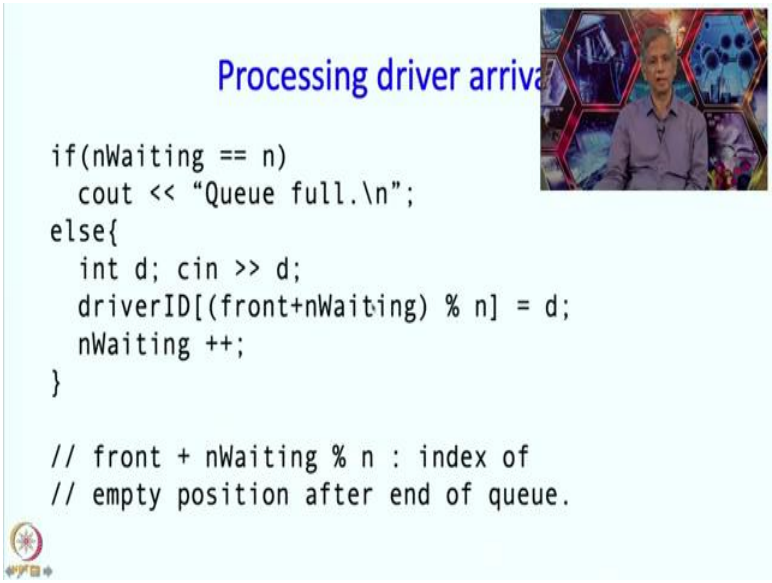


So, here is the pictorial view of the same thing what we have said over here is that this is the initial position so this part is the entire Array and this entire Array is right now unused. So, what I have shown over here are antacids 0 all the way till n minus 1. What happens after some drivers have come in and some customers have also come in well if so many drivers came in and if so many customers left then this initial position becomes unused. These last parts are unused and this middle part is occupied by waiting drivers where the ID's of the waiting drivers.

But now if we continue in this manner what might happen is more and more drivers come over here they may also occupy the top position and maybe more and more drivers will leave. So, at some point in execution the situation might resemble this, that there are some occupied positions by drivers and the middle portion is unused but that is ok this is the front.

So, this is the first driver position or the position of the earliest driver waiting in the system and this is the next one and so on and this is the one after that and this is the last driver that came in and then after that there is this empty space. This is just to clarify what these variables represent and just give you picture of that. So, how do we process a driver arrival ?

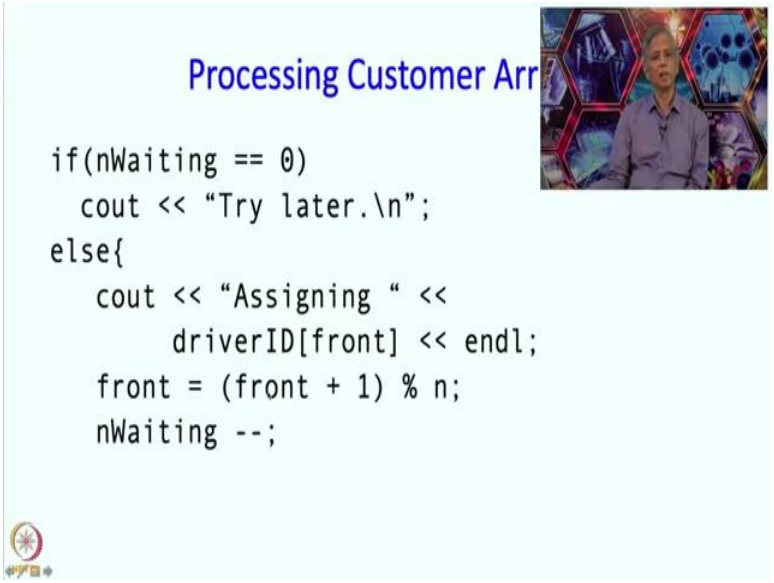
(Refer Slide Time: 11: 58)



So, first of all if nWaiting equals n that means that our entire array is full of drivers. So, we do not have space to put in this new driver so in this case we are going to say well our queue is full please try later. But if it is not n if we do have space then we get the driver details. So, the driver might type in some ID or maybe some phone number whatever conventions we have and we are going to place it in the position where we said the next driver ID should be put in and that is front plus nWaiting mod n and whatever we read in is going to be put in that position. And at this point you will increment nWaiting. Because we got a new driver and there is one additional driver waiting that is it. So, front plus nWaiting mod n is the index of the empty position after the end of the queue. So, this is just a reminder as to why we use this

over here we have discussed this earlier and this is the position where the next driver should go. How do we process a customer?

(Refer Slide Time: 13: 19)

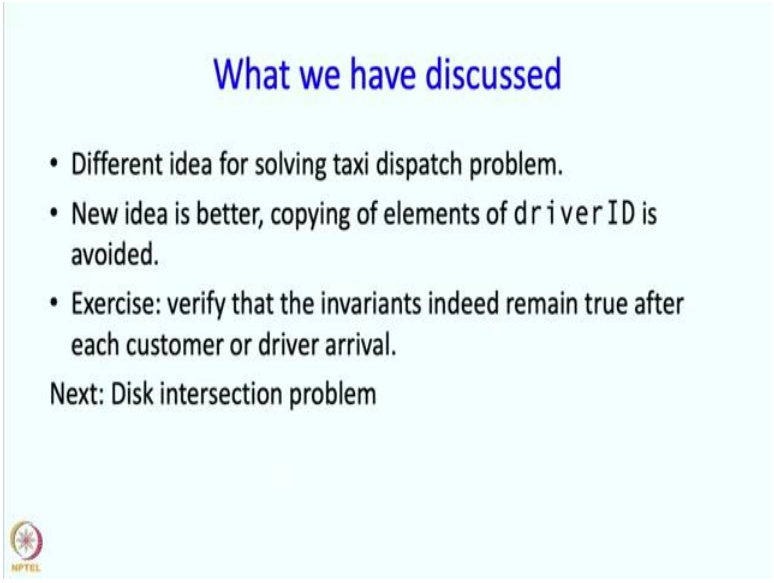


Well this is kind of a compliment of it, compliment of driver processing. Now this time we are going to check if there are any waiting drivers at all. If there are no waiting drivers then the customer cannot be served. So, we have to send a message tell the customer look try later well of course this program this will be read the output the program will be read by the controller who is typing the C's and D's and details of the drivers. So, the controller will read this message try later and presumably the controller will tell the customers.

Otherwise, what happens? Well otherwise we are going to print a message saying look here is a driver for you and which drivers ID will be take? Well according to our invariants the driver ID in index front is the earliest waiting driver. So, that driver ID is going to be sent or assigned to that customers. And we have to say that this position is no longer containing useful data because we assigned that driver.

So, you have to increment front but remember now this is a circular queue. So, we increment it, but if that increment has gone that n we wanted to become 0, so, therefore we take mod n. And we decrement nWaiting that is it. We are not going to shift up the waiting driver's data in our queue. Instead we are going to shift down what is the index of the first unassigned driver.

(Refer Slide Time: 15:08)



This is another solution a different idea and it is a better idea because it does not do really any copying but we have had to have extra variable called front. So, we have to manipulate one extra variable but we saved a lot of copying. So, now, as an exercise I would like you to go through the programs and check that the invariants are indeed true after each customer or driver arrival and of course the code is there in the book and I would like you to run the code and test for yourself whether the code works correctly. So, that concludes this segment in the next segment I am going to talk about something called the disc intersection problem but before that we will take short break.