Programming, Data Structures and Algorithms

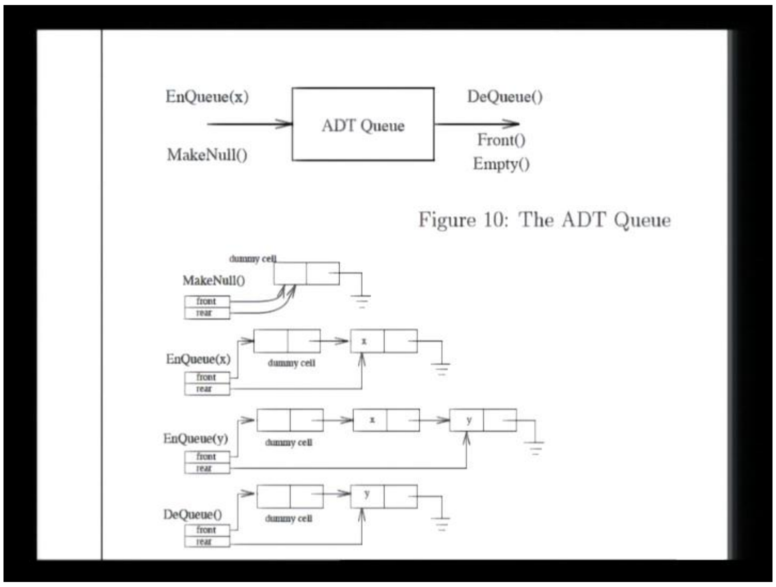
Prof: Hema Murthy
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Module – 07
Lecture - 47
Queues: First in first out
Operations: enqueque, dequeque, empty, front
Implementation of a queue using linked lists
Implementation of a queue using arrays
Implementation of a queue using circular arrays
modulo addition operations

In the last class, we just studied about the stack ADT. Today, I will talk about another kind of list ADT, just called queue ADT. Queue is what, now queue is does not require too much of an introduction. Stack for example, you already list you already know, you always use this list all the time in your life. And stack for example, once I give you the example a stacking plates on a cupboard or stacking books on a desk, you know that immediately that the stack ADT is to be a last in first out kind of a data structure.

Now, a queue is something that you all know you, you stack in India we always stand in queues for almost everything whether it is a railway station, the bus stand or we know getting your registration done in your college, getting your id card, wherever you have whole a lot of queues. And grocery store for that matter, if you go to big bazaar or something that you are standing in the queue for a long time.

(Refer Slide Time: 01:15)

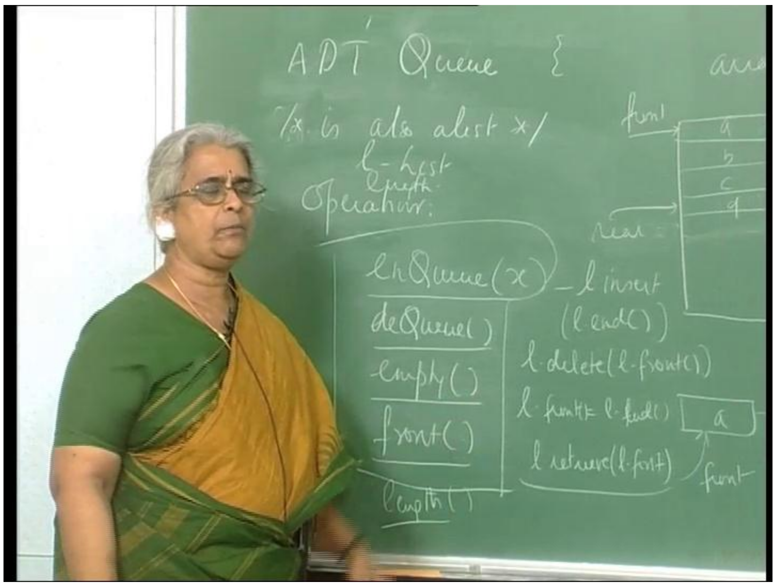


And that is what a queue ADT is all about, what is the property of the queue ADT is just that the person who is at the beginning of the queue is served first and then the next person and so on. So, this data structures somewhat different, unlike the original list ADT where we said, you can insert anywhere and delete anywhere and the stack ADT where we said, the last person gets priority, gets deleted first. This queue ADT is one which is of the type what is called first in, first out.

(Refer Slide Time: 01:45)

You also knows as FIFO first in the queue and therefore, I must be served first. In India of course, in some places when you we have a lot of population, it behaves like the normal list. But, in a civilized society you will have that you follow the queue principles first in first out.

(Refer Slide Time: 02:19)



So, what are the various operations that we perform, you want to perform enqueue, dequeue, empty and front, you want to find who is there in the front of the queue, this are the four operations which you would like to perform. Basically, let why would I want to look at enqueue of course, a particular job as to be enqueued and dequeue when you want to remove an element from the queue, empty queue we want to find whether the queue is empty or not.

For example, if I want to close the counter and I decide that if I have multiple counters let us say, waiting at the passport office or whatever a multiple counters, I would like to check whether the queue is become empty, so that I can close a particular counter. And front of the queue is to check who is there in the front of the queue. For example, if you go to the passport office, the facilities tatkal verses ordinary or whatever.

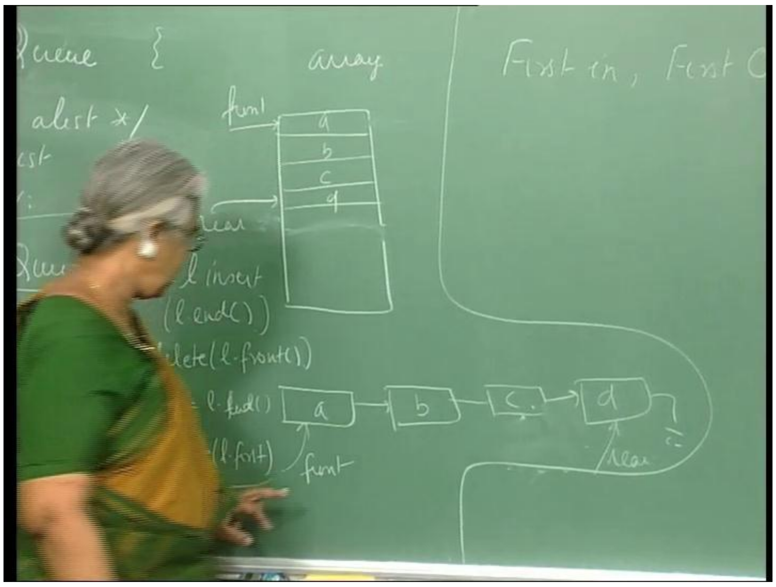
So, I would like to move this person to different queue and so do not be in this queue, if you are asking for some other priority base thing, because this is for non-priority people, everybody has a same preference and if you come in first, you get served first. So, let us see how we can operate on this and clearly since the queue is also a special type of a list, we can again use the list ADT to perform the operations.

So, what will enqueue become now, if I have a list ADT it will say l dot insert at l dot end. We have always inserting at the end of the queue and dequeue would be l dot delete l dot front and empty will be l dot front, just like in the stack equal to l dot end would be an empty queue. Front would be l dot retrieve of l dot first. So, what I have shown here, I have shown you at the all the operations that you want on the list can be on the queue can be implemented using the list.

So, what I could do is in my ADT I can say l is of type list and define operations on list to perform this operations on the queue. So, this is the first thing that I always advice any student to do. If you have one ADT which we implemented which is completely debugged, then my advice is you just use the reuse that ADT again, that is what I am doing again here, just like we did for the stack. I define the ADT queue in terms of the list and do operations on list to mimic the operations of the queue that is all. Then, what happens you very quickly have a queue ADT.

Let us see on the other hand I want to implement my own ADT queue, let that is what I am going to talk about now. So, what is that we have, the operations which remove elements are dequeue, front and empty which gets from the queue and things that put elements on to the queue. First of course, you always have to create a empty queue, this is make null operation and then enqueue which puts elements on to the queue. In my implementation, this we were use in C plus plus what I do is to make deletion easy, I have done a link list implementation here, we will talk about array implementation in a little later.

(Refer Slide Time: 05:49)



So, in a link list implementation is shown here, you have pointers which points to the front of the queue and another pointer will points to the end of the queue. In this example here, that I have done, what I do is in addition to that these are the things by the way. Because, you are hiding the implementation from the user of your ADT, you can do some clever things in your implementation to make some of the operations easy.

What I have done in this implementation over here is if you notice here, my make null operation create an empty queue or create a dummy cell, I did this for the list also if you remember. I create a dummy cell then, my enqueue becomes insertion at the end of the dummy cell. What is the advantage of this, I do not know I have to check whether the queue is empty and then insert. I know that as soon as I have created an empty queue, there is one dummy cell and therefore, insertion becomes simply inserting at the end of this dummy cell.

So, then I let us say enqueued one more element. So, what do I do, I go to the end of the queue and put another element over here, that is I have a front pointer and I have a rear pointer. Please look at this slide, there are two pointers pointing to the queue, front points to the same location and rear points to the same location, then it is empty. But, what is nice is, because I created a dummy cell, what happens is when I created a dummy cell, the advantage is that there is always one element there both at the front and rear end pointing. It is not pointing to somewhere, it is nowhere.

So, whenever I do this when I am doing the enqueue I am inserting x over here. So, what is the interesting is now front is pointing to the dummy cell, rear is pointing to the element. Now, I am enqueue one more element then what happens, rear is updated and y is enqueued after x and rear is updated earlier. So, now what I am going to do when I do dequeue, I just when I dequeue I remove this element. So, I keep the dummy cell as it is and dequeue the element next to the dummy cell, this is the fundamental advantage of using a dummy cell and rear is pointing to y.

(Refer Slide Time: 08:26)

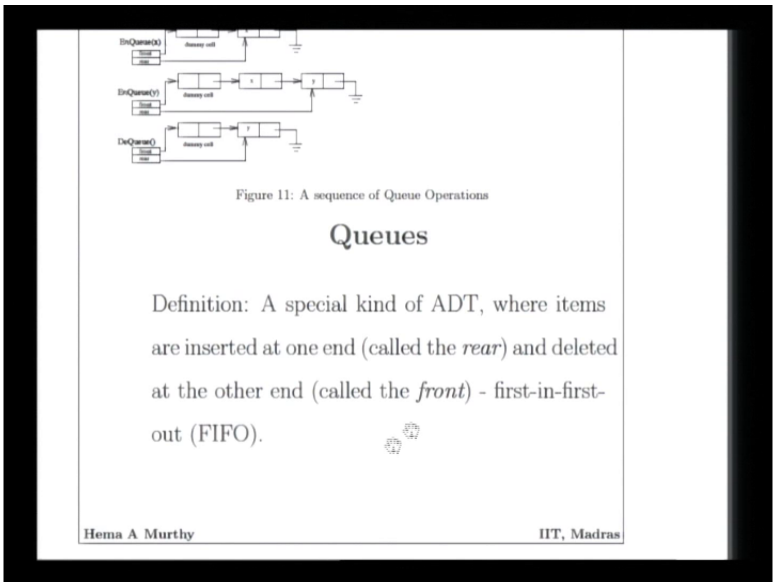

EnQueue(x)
dummy cell
front
rear
EnQueue(y)
x
y
DeQueue()

Figure 11: A sequence of Queue Operations

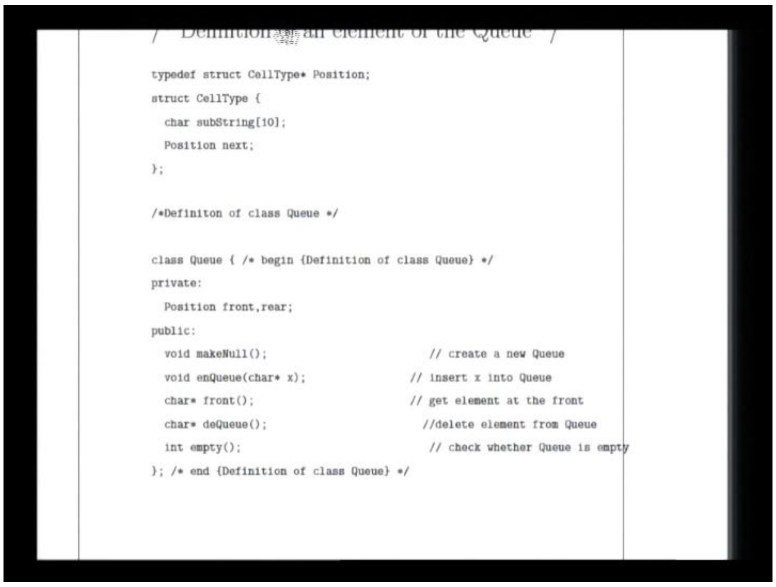
Queues

Definition: A special kind of ADT, where items are inserted at one end (called the *rear*) and deleted at the other end (called the *front*) - first-in-first-out (FIFO).

Hema A Murthy IIT, Madras

So, this is the advantage of implementing, when I am doing a link list expression study. When I do a link list implementation of queues, this is the big advantage of, you know, using a dummy cell to implement when you create an empty queue. This is the big advantage of it.

(Refer Slide Time: 08:39)

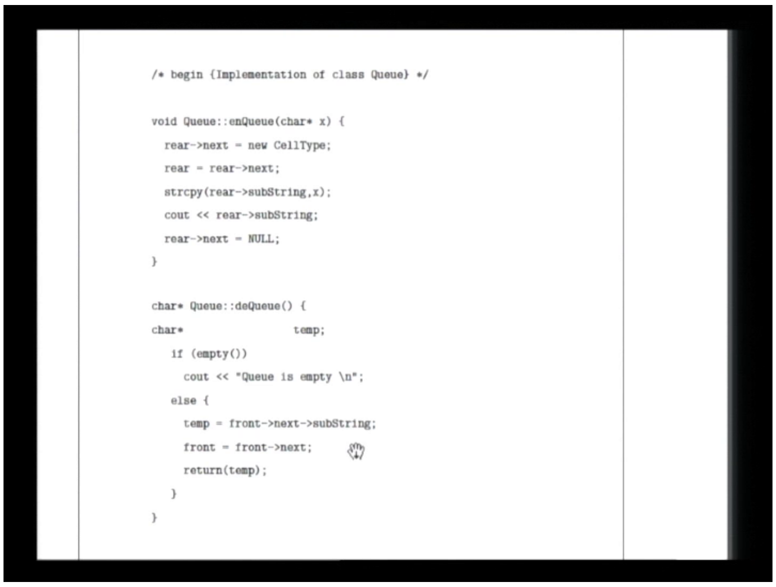


```
/* Definition of an element of the Queue */

typedef struct CellType* Position;
struct CellType {
  char subString[10];
  Position next;
};

/*Definiton of class Queue */

class Queue { /* begin {Definition of class Queue} */
private:
  Position front,rear;
public:
  void makeNull();                    // create a new Queue
  void enQueue(char* x);            // insert x into Queue
  char* front();                    // get element at the front
  char* deQueue();                    //delete element from Queue
  int empty();                        // check whether Queue is empty
}; /* end {Definition of class Queue} */
```

And here is an implementation over here and I have just given you the C plus plus implementation here. So, you have a recursive data structure just as before and notice that again, I have what you called typedef position. So, for example I have said typedef struct CellType position, let us see how you do this in the case of the array. In the case of the array, when I do a typedef into the integer which will be typedef to this.

Basically, the user is simply going to use this typedef position, just like you use a type integer, char, float, for that matter we create a new type called position. And how the implementation of the position is, in this case it is a pointer to a recursive data structure, in a case of array it is said to be an index to an index in the element in the array. And of course, you need two positions front and rear, because we have two pointers for this. And we have a creation of the empty queue, then you have the enqueue, then you have a dequeue and then you have a sums an operation it tells you what is there in the front of the queue and other which tells you whether the queue is empty or not.

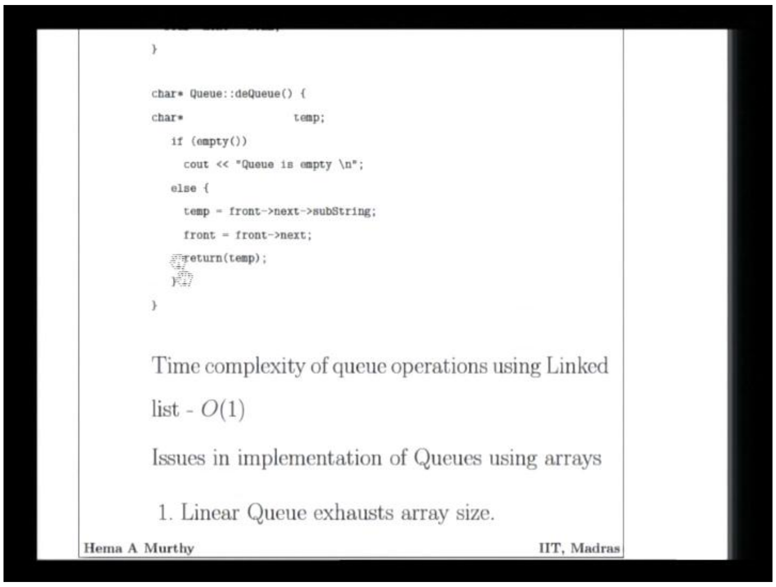
(Refer Slide Time: 09:45)



```
/* begin {Implementation of class Queue} */

void Queue::enQueue(char* x) {
  rear->next = new CellType;
  rear = rear->next;
  strcpy(rear->subString,x);
  cout << rear->subString;
  rear->next = NULL;
}

char* Queue::deQueue() {
char*                  temp;
   if (empty())
     cout << "Queue is empty \n";
   else {
     temp = front->next->subString;
     front = front->next;
     return(temp);
   }
}
```

And here is a simple implementation of how the rear is done, rear point and next it and then basically wherever the rear is you create a new element at the end of, create a CellType with new element and put this element over here. Dequeue what are we doing, this is the fundamental advantage, dequeue is empty then you say queue is empty; otherwise, we want to be do it just simply update a front to point to the next element.

(Refer Slide Time: 10:17)



```
}

char* Queue::deQueue() {
char*                  temp;
   if (empty())
     cout << "Queue is empty \n";
   else {
     temp = front->next->subString;
     front = front->next;
     return(temp);
   }
}
```

Time complexity of queue operations using Linked list - $O(1)$

Issues in implementation of Queues using arrays

1. Linear Queue exhausts array size.

Hema A Murthy IIT, Madras

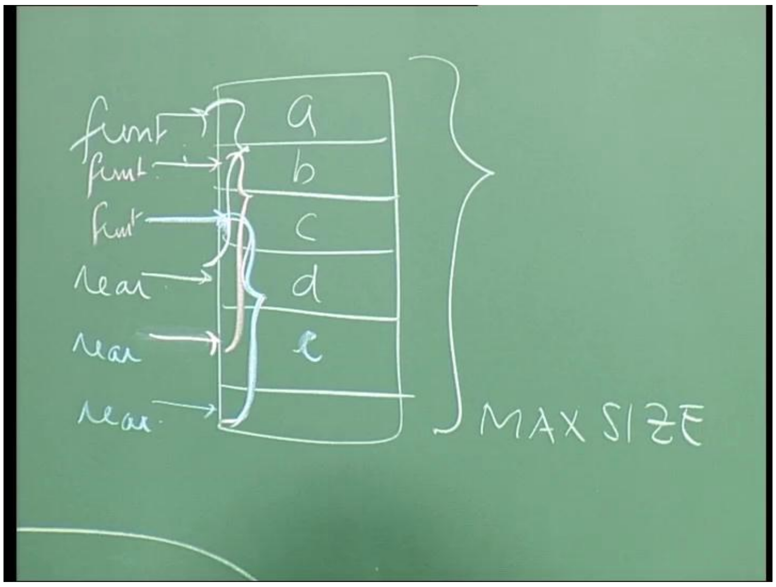
What is the interesting, the time complexity of all these operations notice that there is no loop in any one of them, there is no going from the beginning of the list to the end of the list. Therefore, all these operations this is, if you let us take enqueue for that matter, this is order one, this is order one, this is order one, this is order one. So, this is the form of p 1 plus p 2 plus p 3 plus whatever, if you take each one of these as a subprograms over here.

Then, what are the time complexity of the segment it is basically the math's of all these therefore, the time complexity of enqueue operation is order 1. Similarly, we can argue that the time complexity of the dequeue operation is also order 1 and creating an empty queue, every one of these is order 1. So, this particular list implementation it is not difficult, it only takes order 1 time.

Now, additionally sometimes for example, there is also another operation that you would like to perform have which is called the queue length which will return a value which tells you how long the queue is some ADTs also at the implementation of the queue also have queue length. What is the advantage of this? For example, if I am again grocery store like you know food world or something like that I may decide looking at the length of the queue whether I want to increase the number of counters.

So, it is a big advantage to do something like this to. It can also have an extra function called length. Now, if I am looking at the extra function called length, how expensive will this operation be, if you keep… So, basically you need another variable called length which maintains the length of the queue and then what will happen is you just keep on incrementing it when it enqueues and decrementing it when it dequeues. So, again that operation is not going to cost you more than order one in terms of time complexity. So, we talked about a link list implementation of the queue and I like to talk about an array implementation.

(Refer Slide Time: 12:23)

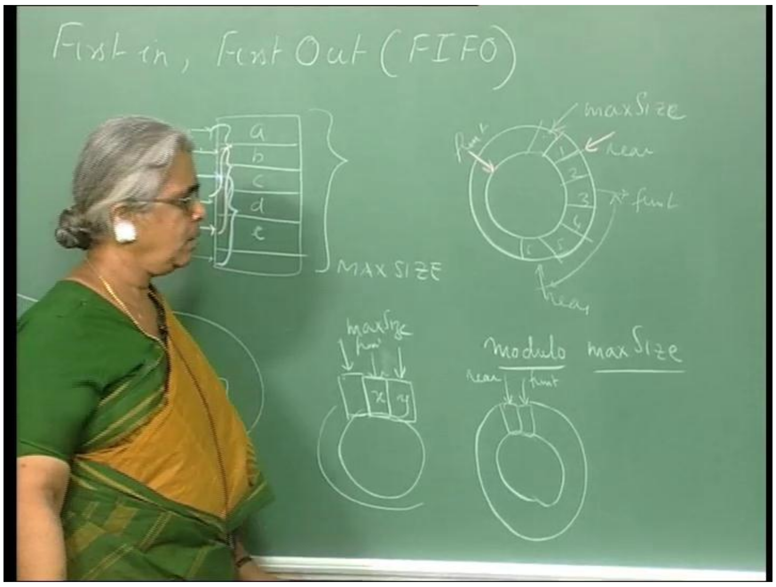


So, what we will do with an array implementation of queues, so let us say this is my front this is the rear, let us say I have an array of some particular size, let me call it some max size. I am going to leave you this I am not going to give you the complete implementation of this. So, this is my front the rear, now I dequeue then what happens front moves over here, this becomes front.

Again I dequeue this becomes front, then let us say enqueue, then may be rear moves here, let us say I have e and then now I reach the end, again I rear. So, what happens is the problem is that the queue is found somewhere between front and rear in the array, let me say this is the... So; that means if I am looking at the pink color over here, the queues found between front and rear, from looking at blue the queues is found between front and rear and similarly the whites give me the queue between the front and rear.

But, the basic problem with this is this kind of an array implementation of queue what will happen very quickly it will exhaust the size of the array. What will happen is, suppose I keep on dequeuing an element, then the front comes over here and then I cannot add any more elements, this not a good idea.

(Refer Slide Time: 14:05)



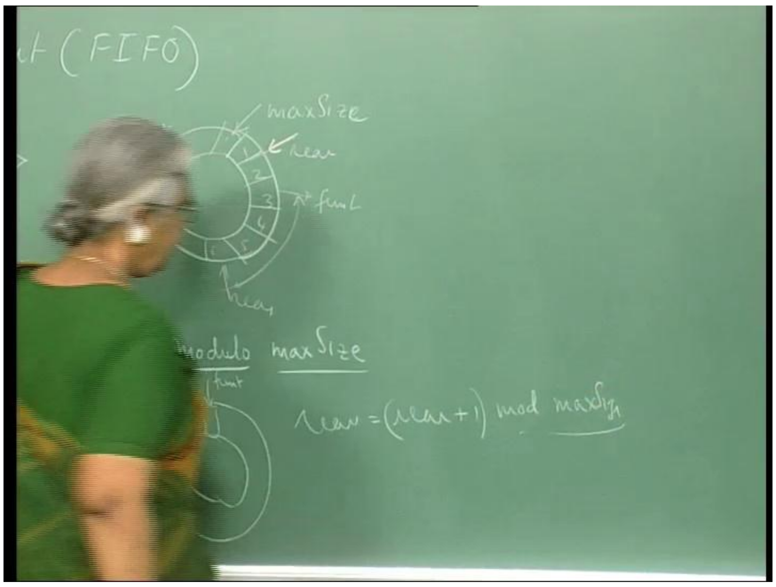
So, what we normally do is that we convert this array implementation to a circular queue, let us say this is 1, 2, 3, 4, 5, 6 and so on. And this is the max size, it is still an array implementation, but we do what you call a logical tying of the front and the rear of the queue. So, what is the meaning now, if this is front here, this could be rear here, and it could also happen that this is front and this is rear, how do you perform this kind of arithmetic now. That means, our circular array and somewhere going clock wise, I will find the front and the rear of the queue.

So, how do you perform these operations, you guessed it right, you have do a modulo operations, modulo based on what max size, to find the location of the element. So, what happens if all the time we will taking whenever we keep incrementing, the array for example, initially let us say both front and rear are pointing to the first element or let us say front is pointing to the first element and rear is pointing to max size over that to show that it is symmetrically, I will tell you this is become a problem, but we will talk about in a little later.

Then, what are we doing when I am going to enqueue, then what I am going to do now, I am going to move, put the rear over here and I will put the element over here, this is front let us say keep enqueue, this is front this is rear. Then, what happens I am always doing modulo size arithmetic. So, when it exceeds it over here, then what will happen the rear will start over flowing onto the beginning of the array is this problem with this implementation I want to you think about it. I said, I start with front pointing here and rear pointing here to indicate that it is an empty queue.

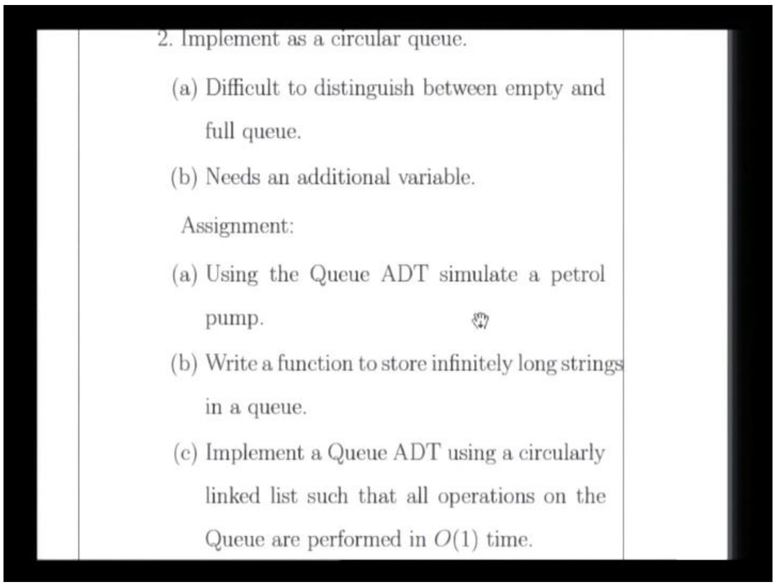
Now, how do we distinguish between an empty and a full queue, this becomes a problem. So, we will need another variable or alternatively you can think of something more clever, and I would like you to think about this particular problem. So, circular queue the advantages is that the queue is found anywhere between two indices front and rear. How are these indices computed, the indices are computed using modulo arithmetic and a modulo is based on...

(Refer Slide Time: 17:06)



For example, I will say rear is instead of saying rear plus 1 we will do rear plus 1 mod max size. And we have do a little more big book keeping as I already said when we… to distinguish between a full and an empty queue, there are two ways of doing it, you can introduce an extra variable. So, the point is if I am using an array implementation what is happening, the linear array the array size gets exhausted very quickly. But, the circular array whatever is away allowable in this, it the queue can be basically the queue moves like this, that is the fundamental property of the circular force. nice data structure and I want you to implement all of this. Interestingly, again all the operations on the queue can be implemented in order 1 time. Let me leave you with this.

(Refer Slide Time: 18:07)



2. Implement as a circular queue.
   (a) Difficult to distinguish between empty and full queue.
   (b) Needs an additional variable.

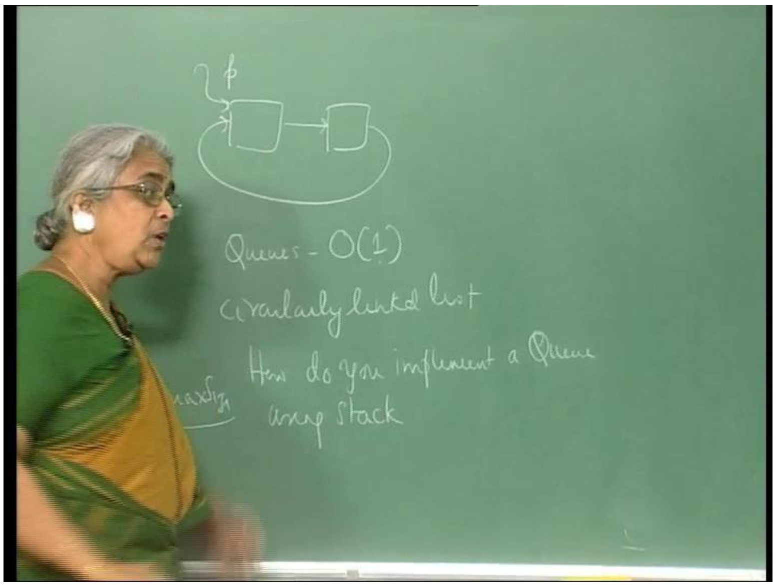
Assignment:

(a) Using the Queue ADT simulate a petrol pump.
(b) Write a function to store infinitely long strings in a queue.
(c) Implement a Queue ADT using a circularly linked list such that all operations on the Queue are performed in $O(1)$ time.

As I already said implementation is a circular queue in an array, there are issues you need to find out how to fix this, difficult to distinguish between empty and a full queue. Then, I wanted you to do this as an assignment, it is nice to simulation is very good way of learning about ADTs. So, I would like to simulate a petrol pump, where I have a petrol pump, it serves petrol for various people. Let us say you have two different petrol pumps, pumps one for the 4 wheeler and one for the 2 wheeler and for each one of them you can have arrivals that happen to the particular system, give you an example.

Let us say the two wheelers arrive very frequently may be I have every, I almost have a 2 wheeler every in every minute, whereas I have one 4 wheeler every 5 minutes or something like that. Simulate this and put it in a queue and then enqueue and dequeue and just give statistics of how long the queue is at different intervals of time. In other very interesting implementation of queues is using the circularly link list.

(Refer Slide Time: 19:20)



That is, I am having a link list over here and I tie the back to the front and I am giving you only one pointer p, no two pointers we have front and rear. I want you to implement the queue, all operations on queues must be order 1, I want you to do this. I am giving only one pointer, no front and rear this is only pointer p. So, now where should p point to, should point to the front, should point to the rear. I want you to think about it, it is a circularly linked list.

Now, this one more problem which is not listed there, how do you implement a queue we saw in the beginning of the class, how to implement a queue using a list ADT. The question that I want to ask you is how do you implement a queue using stacks? I want to use a stack ADT. Remember we you reuse the list ADT to build a queue ADT, can I do the same thing after all queues also a list, type of list says stack is also a type of list. Now, can I use the stack ADT to implement the queue ADT? What would I require, I want you to ponder about this particular problem.