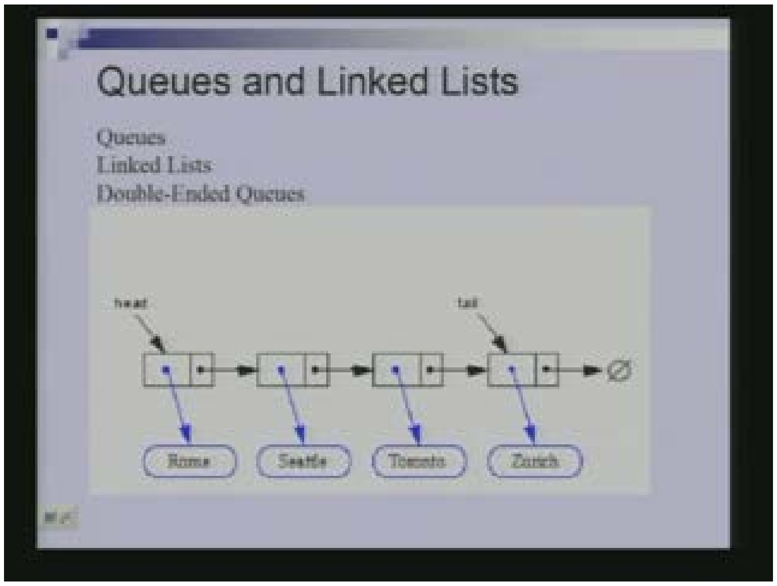
Data Structures and Algorithms
Dr. Naveen Garg
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 3
Queues and Linked Lists

In the last lecture we looked at stacks as a data type. We saw how to implement stacks using an array. Today we are going to look at queues and linked list and in the later part of the class, we are going to do sequences. In particular the first part of the class I am going to do queues, linked list and double ended queues.

(Refer Slide Time: 01:22)



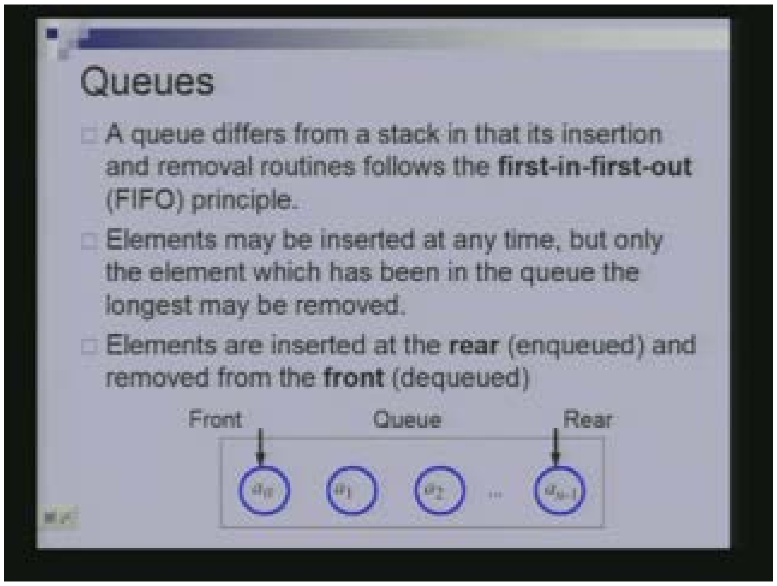
Queues and Linked Lists

Queues
Linked Lists
Double-Ended Queues

head
tail
Rome
Seattle
Toronto
Zurich

What is the queue and how does it differ from the stack? The stack followed the last-in first-out principle, the element that was inserted last in to stack was the one that was removed first.

(Refer Slide Time: 01:34)



Queues

- A queue differs from a stack in that its insertion and removal routines follows the **first-in-first-out** (FIFO) principle.
- Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed.
- Elements are inserted at the **rear** (enqueued) and removed from the **front** (dequeued)

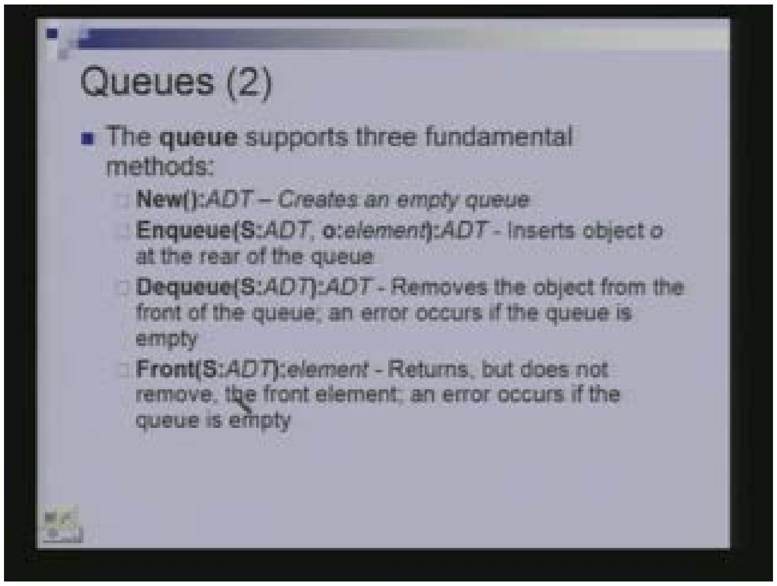
Front
Queue
Rear
a_0
a_1
a_2
...
a_{n-1}

Queue on the other hand follows the first-in-first-out principle. Whoever joins the queue earlier is the first to be removed from the queues that is first to be processed. You are all familiar with the queues. In a queue for instance there is a notion of a first element and the notion of the rear element.

When an element is inserted in to the queue, it comes at the rear. If I remove an element from queue it is the element which is sitting at the front end would be removed. We always insert an element at the end and when we remove an element it is always the element at the front is removed.

The queue is also an abstract data type and we can define a few methods on the queue. The methods given in the slide are the standard operations. The method new would create a queue and enqueue is the method to add an element to the queue and dequeue is to remove an element from the queue.
When you dequeue a queue or when you remove an element from the queue, you get another queue.

(Refer Slide Time: 02:48)

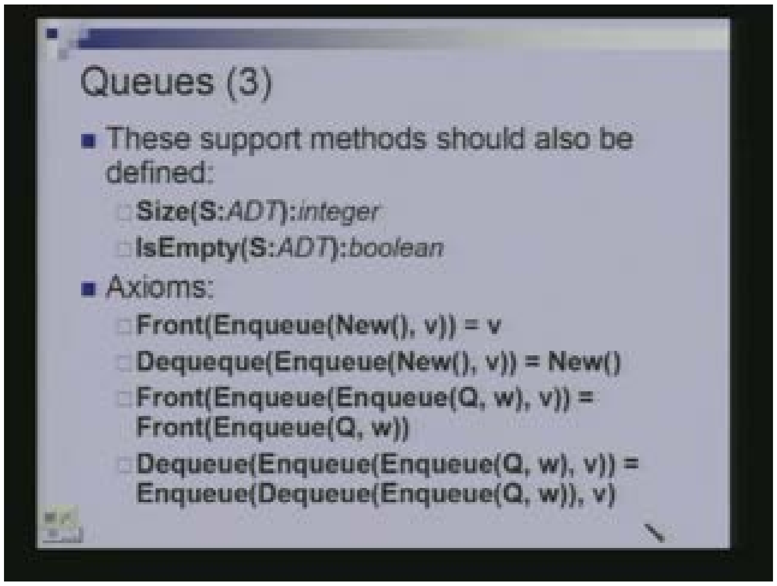


Queues (2)

- The **queue** supports three fundamental methods:
  - **New():***ADT – Creates an empty queue*
  - **Enqueue(S:***ADT*, **o:***element***):***ADT* - Inserts object *o* at the rear of the queue
  - **Dequeue(S:***ADT***):***ADT* - Removes the object from the front of the queue; an error occurs if the queue is empty
  - **Front(S:***ADT***):***element* - Returns, but does not remove, the front element; an error occurs if the queue is empty

The front is the method which gives the first element of the queue.
How it does differ from dequeue?
It does not remove the front element, it only tell us which is the front element of the queue.

(Refer Slide Time: 03:46)



Queues (3)

- These support methods should also be defined:
  - **Size(S:***ADT***):***integer*
  - **IsEmpty(S:***ADT***):***boolean*
- Axioms:
  - **Front(Enqueue(New(), v)) = v**
  - **Dequeque(Enqueue(New(), v)) = New()**
  - **Front(Enqueue(Enqueue(Q, w), v)) = Front(Enqueue(Q, w))**
  - **Dequeue(Enqueue(Enqueue(Q, w), v)) = Enqueue(Dequeue(Enqueue(Q, w)), v)**

We also have some other support methods, to implement the queue. One could be size and the other is IsEmpty. Size would tell us how many elements are there in the queue and IsEmpty would tell us whether the queue is empty or not. It would return true if the queue is empty else it would return false.

Just as we defined axioms for the stacks, you can define similar axioms for queues. If I create a new queue and I insert an element or enqueue an element v and then when I say what is the element at the front of the queue. It should be v and suppose I create a new queue and enqueue an element and then I dequeue an element, then I should get the empty queue which is the same as whatever obtained if I just called new.

Similarly if I had a queue and I enqueued an element w, which means I added an element to the queue. Then I added another element v to the queue, thus w is ahead of v in the queue. If I call front, first I will get all the elements of the queue, followed by w and then followed by v.

The element in the front of (Q, w) is the element at the front of the queue.
Why I have written front of (Q, w) and not front of queue?
If I have just written front of queue, then it would not have been defined. If a queue is empty then there is no notion of front of the queue that is why I have written Front (Enqueue (Q, w)).

Same thing as before in which I had a queue Q, if I insert w in to the queue then I insert a v and then I removed an element. The element which was at the front of queue would be removed, if the queue was empty then it would have been w. The following operation is the same as in which I had a Q, I added w to the queue then I removed an element from the queue, then I added v again. The queue that I have obtained as the result of the below mentioned 2 procedures should be the same.

Dequeue(Enqueue(Enqueue(Q,w),v))=Enqueue(Dequeue(Enqueue(Q,w)),v)

Let us check out whether the result is true.

Let us assume that the queue was initially empty.
What does this statement Dequeue (Enqueue (Enqueue (Q, w), v)) gives?
First I added w then added v and then I removed an element. If I removed an element from the queue I would get w, that is w is removed. Then I get v as the remaining queue.

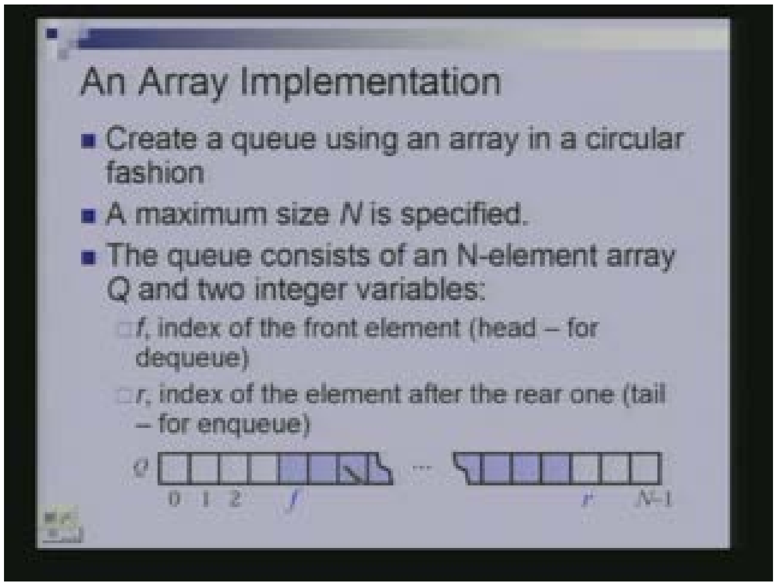
Let us look at this Enqueue (Dequeue (Enqueue (Q, w)), v).
Queue is empty, I added w to the queue then I removed an element and once again I have left with an empty queue. Then I enqueue v, thus the queue has v in it. If queue is empty then in both the cases the queue will have only v in it at the end of the procedure.

If queue is not empty then I enqueued w, then I enqueued v again. Hence I have a queue in which first I have all the elements of Q, followed by w, followed by v. When I dequeued, I will be left with the original Q without the front element, followed by w and then followed by v.

Let us see if we get the same thing in Enqueue (Dequeue (Enqueue (Q, w)), v).
I started with Q and I added w to it. Now I have queue which has Q and w. Then I dequeued which means I removed the front element of queue. The queue contains all the elements except the front element, then I have w in that queue and I added v at the end. Thus I get the same result.

How do we implement a queue?
We are going to use an array in a circular fashion to implement the queue.
What does it mean?
Suppose someone tells that the queue is never going to be larger than n elements. I am going to allocate an array of size N. I am going to have 2 variables f and r, f for front and r for rear. f is the index of the front element that is f will be referring to the front element of the queue. r is an index which is the element following the rear element. The blue part is the one which is occupied by the queue.

(Refer Slide Time: 08:11)


An Array Implementation

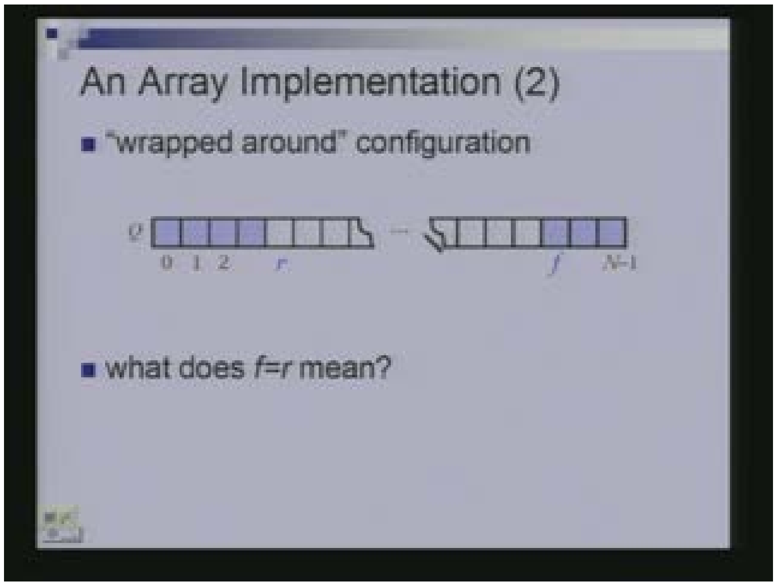
- Create a queue using an array in a circular fashion
- A maximum size *N* is specified.
- The queue consists of an N-element array Q and two integer variables:
  - *f*, index of the front element (head – for dequeue)
  - *r*, index of the element after the rear one (tail – for enqueue)

How did the queue reach the blue colored part?
I would have started with the front that is the first element I inserted must have come to the 0th location. Then the next element I inserted must have come to the 1st location and the 3rd element must have come to the 2nd location and so on. Then I also delete the elements. When I delete, an element goes away. In effect the elements in the queue drift right and hence the front and the rear element has moved to right.
This implies that we have deleted f-1 elements. It is not completely accurate. I had said something like in a circular fashion.

What does the circular fashion mean and why am I saying such a thing?
Let us say that we kept inserting the elements in queue. I insert another element then I insert another element and at one stage I cannot insert anymore elements because I have already reached the end of this array. But I have a space available in the front then I will wrap around and start inserting the elements.

Your queue in some point will look like the one which is given in the slide below. The front was at the left of the rear but now front is at the right of the rear because we have queue which is now starting from the right side and going to the left side. When I insert an element it will still come to the r^{th} location and then to the next location and so on.

(Refer Slide Time: 10:15)



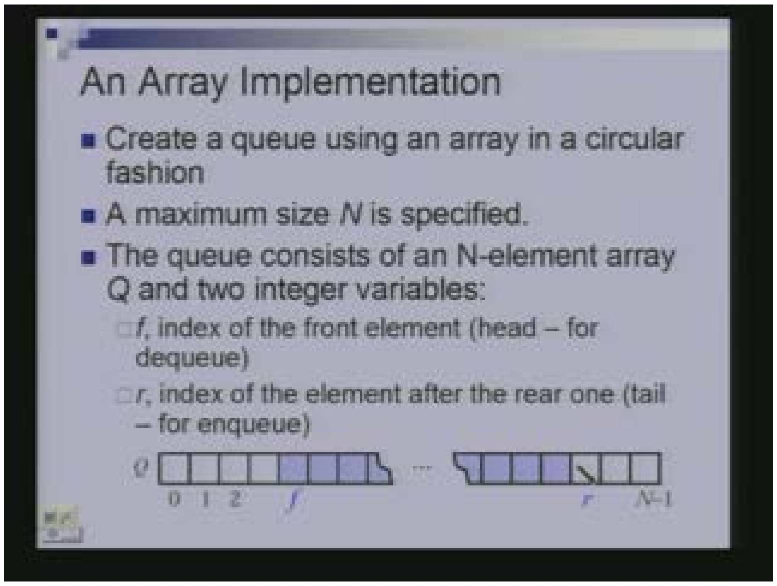
An Array Implementation (2)

- "wrapped around" configuration

- what does $f=r$ mean?

When we started the front was referring to 0th location that is f should have been at minus one, because the front refers to the first element of the queue. If there is nothing in the queue then f should be minus one and rear refers to 0, because rear refers to an empty location.

(Refer Slide Time: 10:59)

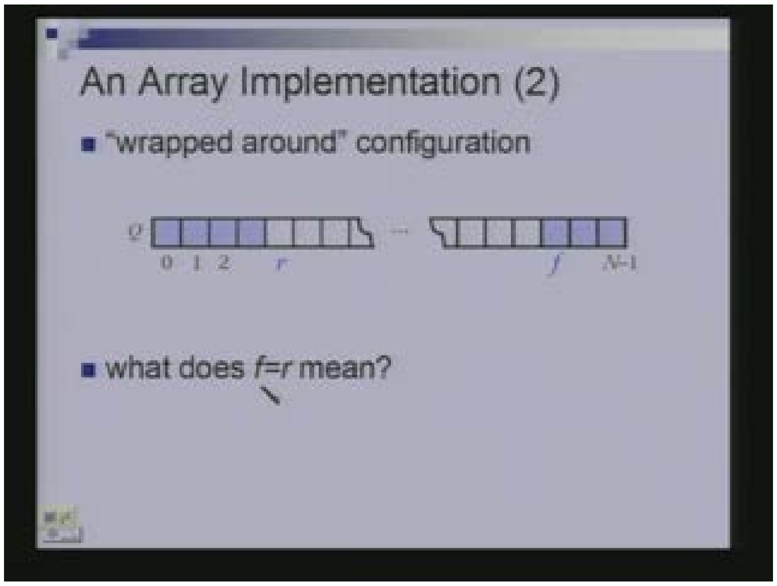


An Array Implementation

- Create a queue using an array in a circular fashion
- A maximum size N is specified.
- The queue consists of an N-element array Q and two integer variables:
  - f, index of the front element (head – for dequeue)
  - r, index of the element after the rear one (tail – for enqueue)

What does if at some point I reach a situation when f = r?
Is that empty or full?

(Refer Slide Time: 11:41)


An Array Implementation (2)

- "wrapped around" configuration

- what does $f=r$ mean?

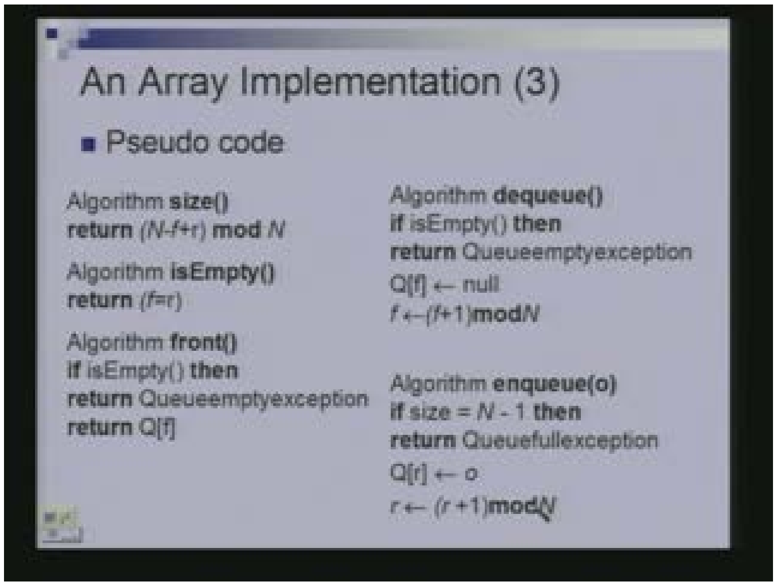
What will happen when it becomes empty?
Suppose if I kept removing the elements starting from the f^{th} location, I did not add any other element and then I removed all the elements before r^{th} location.

Where f would be located?
f would be increment to r, so f becomes r. When f is r, queue is empty. Suppose I kept adding the elements to the queue. When I add, r will move one step, another step and so on. When I add an element close to f^{th} location, then r would be referring to f. Again f equals r. We will add the n^{th} element. There is an ambiguity and we have to resolve it in some manner.

f = r means, both empty and full. Since we will have a problem, if you do not know whether the queue is empty or full. We will try and ensure that we never had n elements to the queue. When the queue has only n-1 one elements, we will declare it before. That is what we are going to do.

Let us look at the code for enqueue.
This is just pseudo code. If the size of the queue or the number of elements in the queue is n-1, then we are going to stop and say that the queue is full and we will return the queue full exception. Otherwise if it is not the case then add the rear location, put the element that you are trying to insert and increment r.

(Refer Slide Time: 13:30)



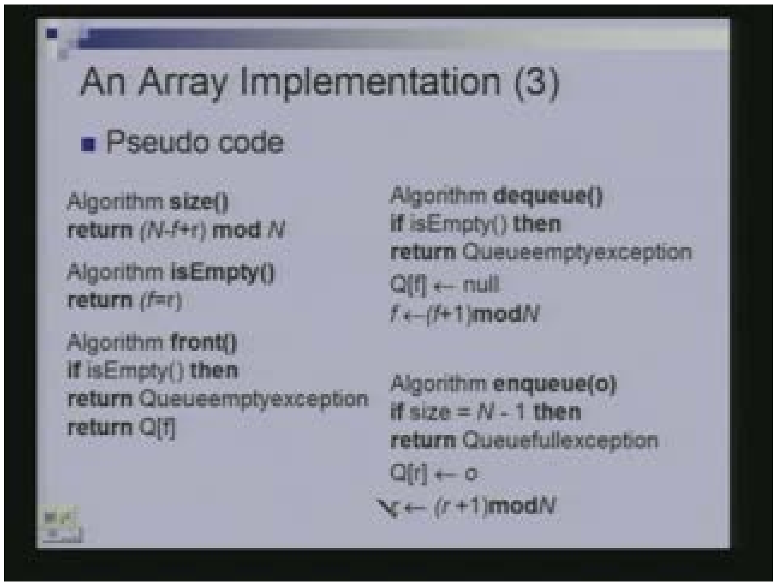
An Array Implementation (3)

- Pseudo code

```
Algorithm size()
return (N-f+r) mod N

Algorithm isEmpty()
return (f=r)

Algorithm front()
if isEmpty() then
return Queueemptyexception
return Q[f]

Algorithm dequeue()
if isEmpty() then
return Queueemptyexception
Q[f] ← null
f ←(f+1)modN

Algorithm enqueue(o)
if size = N - 1 then
return Queuefullexception
Q[r] ← o
r ← (r +1)modN
```

The modN is required, because we need to do the wrap around since it is circular. Indices goes from 0 through n-1 only, r is already n-1 and I increment r at this r ← (r+1) modN point. Then I do not want it to become n, but I want it to become 0 and hence modN is 0 and I will have it in r. In the pseudo-code, size is the method and it should have been enclosed in brackets.

(Refer Slide Time: 14:13)



An Array Implementation (3)

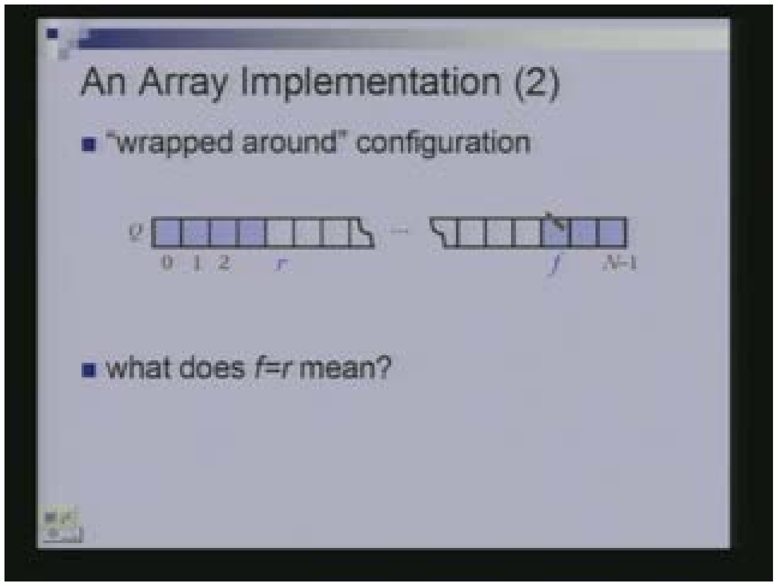
- Pseudo code

```
Algorithm size()
return (N-f+r) mod N

Algorithm isEmpty()
return (f=r)

Algorithm front()
if isEmpty() then
return Queueemptyexception
return Q[f]

Algorithm dequeue()
if isEmpty() then
return Queueemptyexception
Q[f] ← null
f ←(f+1)modN

Algorithm enqueue(o)
if size = N - 1 then
return Queuefullexception
Q[r] ← o
r ← (r +1)modN
```

What does the method size do?
It returns (N-f+r) value.
Why it should not return r-f.?

r-f is negative in the setting which is given in the slide (Refer Slide Time: 15:04), but in the setting which is given in the slide (Refer Slide Time:: 10:59), r-f tells me exactly the number of elements in the queue. r-f is the correct thing except it might be negative.

(Refer Slide Time: 15:04)



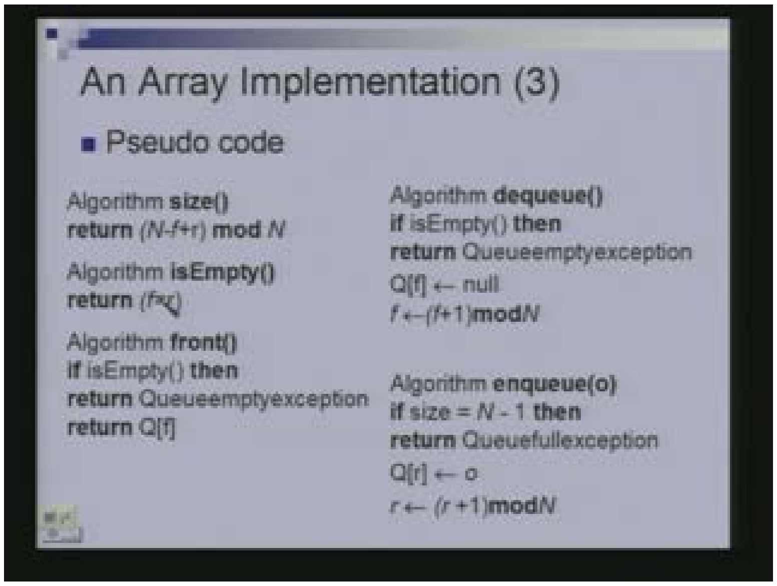
An Array Implementation (2)

- "wrapped around" configuration
- what does $f=r$ mean?

How many elements are there in the queue which is given in the above slide?
It should be n- r+f or n-f+r, which is anyone of these.
The quantity (N-f+r) would be the number of elements that you would get and this quantity is always positive, because r-f can at worst be minus n. Thus N+r-f would always be a positive quantity. You can return this (N-f+r) as the size, as this will tell you the right number of elements. Check this out if you are confused.

isEmpty () is a method and we said queue is empty if f=r. There was an ambiguity and we never had more than n-1 elements into the queue. If f =r, that means the queue is empty and it is not full. Thus f=r returns empty also it returns true for this (Algorithm isEmpty ()) method.

(Refer Slide Time: 14:35)



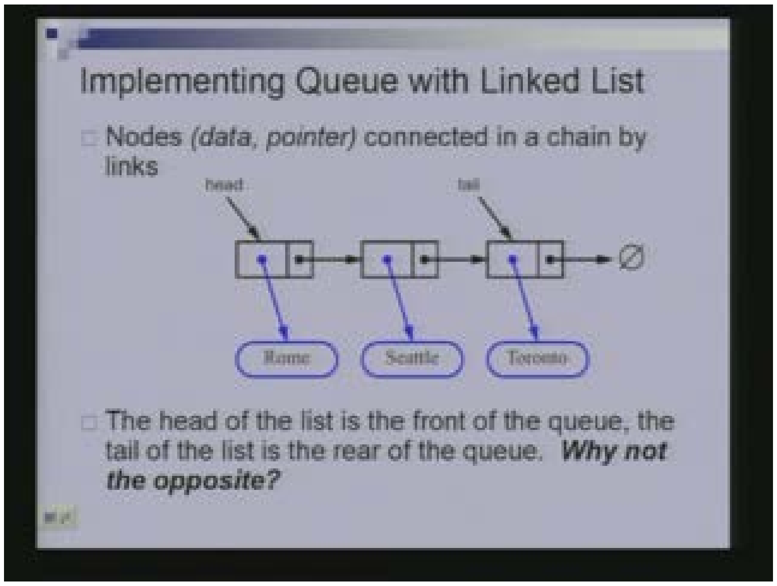
An Array Implementation (3)

- Pseudo code

```
Algorithm size()
return (N-f+r) mod N

Algorithm isEmpty()
return (f=r)

Algorithm front()
if isEmpty() then
return Queueemptyexception
return Q[f]

Algorithm dequeue()
if isEmpty() then
return Queueemptyexception
Q[f] ← null
f ← (f+1)modN

Algorithm enqueue(o)
if size = N - 1 then
return Queuefullexception
Q[r] ← o
r ← (r +1)modN
```

For front if the queue is empty then it raises an exception, otherwise just return a front element. We are not removing the front element as we are doing it in the dequeue method. In the case of dequeue method, we will increment the front index and remove the front element by setting Q (f) ← null.

(Refer Slide Time: 16:39)



Implementing Queue with Linked List

- Nodes *(data, pointer)* connected in a chain by links

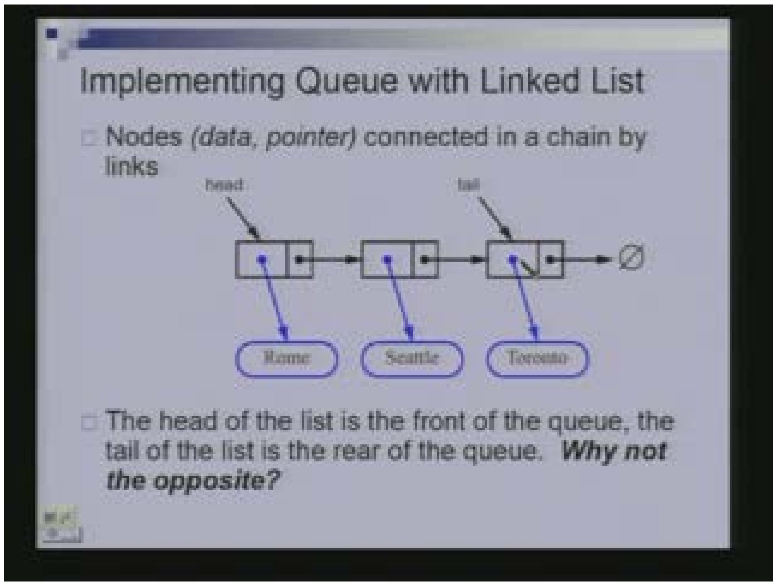
- The head of the list is the front of the queue, the tail of the list is the rear of the queue. ***Why not the opposite?***

You can also implement the queue using a linked list. We saw an array to implement our queue. The disadvantage of using an array is fixed size. If you know the maximum size that the queue can take then it is ok, but if you have no idea about the maximum size, then you could either use the method which we did in the last class were in when the size increases beyond what we have allotted, then we double the size of the queue. You could either do that or you could use an implementation which uses a linked list.

What is essentially a linked list?
It has nodes and it has pointers which are basically referring to the next nodes in the list. The first node is referred to as head of the list and the last node is referred to as the tail of list. Each of the nodes has some element or some data in it.

If I am going to use a linked list to implement the queue, then the question is which should be the front of the queue, whether the head node should be the front of queue or the tail node should be the front of the queue. The head of the list should be the front of the queue, the tail of the list cannot be the front of the queue.

(Refer Slide Time: 17:09)



Implementing Queue with Linked List

- Nodes *(data, pointer)* connected in a chain by links

head
tail
Rome
Seattle
Toronto
∅

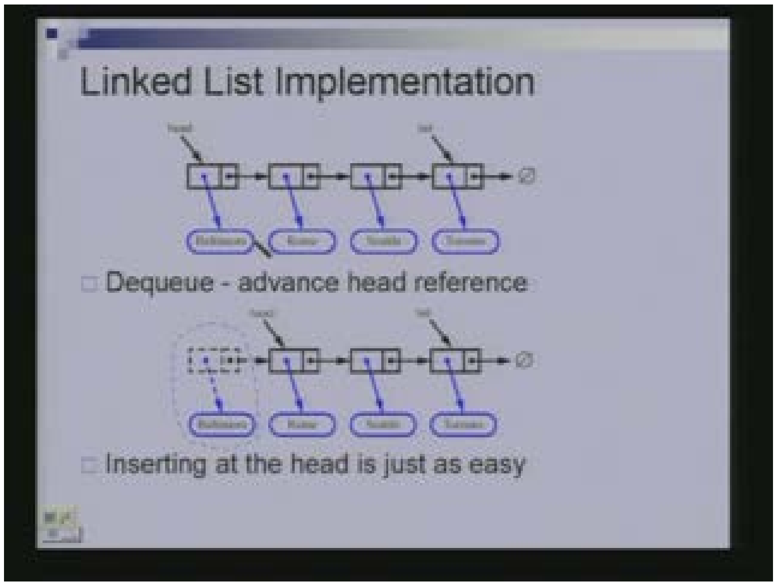
- The head of the list is the front of the queue, the tail of the list is the rear of the queue. ***Why not the opposite?***

Why the tail of the list cannot be the front of the queue? Why cannot I have my queue in which the 1st element is this, the 2nd second element is this and the third element is this? The problem is with removing, note that I cannot remove the torcezo element. The linked list does not permit me to do this.

Can I remove the torcezo element from linked list?
Not directly, because to remove that element I have to change the 2nd pointer. But there is no way of accessing that pointer and hence I cannot remove that element. I can remove the rome element, there is no problem in it, but I cannot remove the torcezo element.

In a queue the removal is being done at the front that is we remove the element at the front of the queue. Since I cannot remove the element which is sitting at the last place and I cannot call this as the front of the queue. I would like to have rome as the front of my queue.

(Refer Slide Time: 18:38)



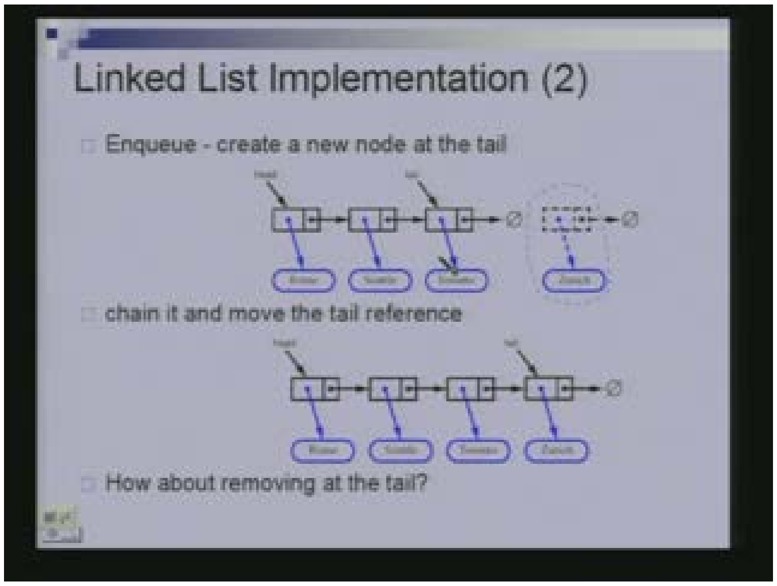
Linked List Implementation

- Dequeue - advance head reference
- Inserting at the head is just as easy

Let us see how we are going to implement our methods.
Suppose I have to dequeue which means that the front of the queue is the one which is at your left. Head part is the front of the queue and the tail part is going to be the rear of the queue. If I have to remove the element at the front of the queue that is to dequeue, I should point the head to the 2nd node.

Thus the front element will get removed and I just increment or just making the head point to 2nd node. In this manner I can delete the head element very easily and also I can insert a new element to the head easily. I just create a new node, connect the new node and make the head point to the new node. Thus inserting at the head is very easy. The head is the front of the queue, I can just move the head to one step right and in that manner, remove the front element of the queue.

(Refer Slide Time: 20:44)


Linked List Implementation (2)

- Enqueue - create a new node at the tail
- chain it and move the tail reference
- How about removing at the tail?

head
tail
Zurich
head
tail
Zurich

If I have to add an element, enqueue has to be done at the rear of the queue. In the above slide, first diagram is my queue and the last element is the rear of the queue. I need to add a new element at the rear end of the queue. The pointer should now get modified to point to the newly added element and the tail should be update to the next node because that will become tail and the pointer after the rear element should be null.

I can always add an element at the tail but it is difficult to remove an element in constant time, because to remove the tail node, I need to access the previous node. The only way you can to do in this kind of list is to start from the beginning and move all the way to the right till you get to the tail node. Then you will be able to access the previous node.

What is problem in removing in the tail node?
The problem is that after I remove the tail node, what is the new tail of the list. It is the last before node, I have to make the tail point to that last before node.

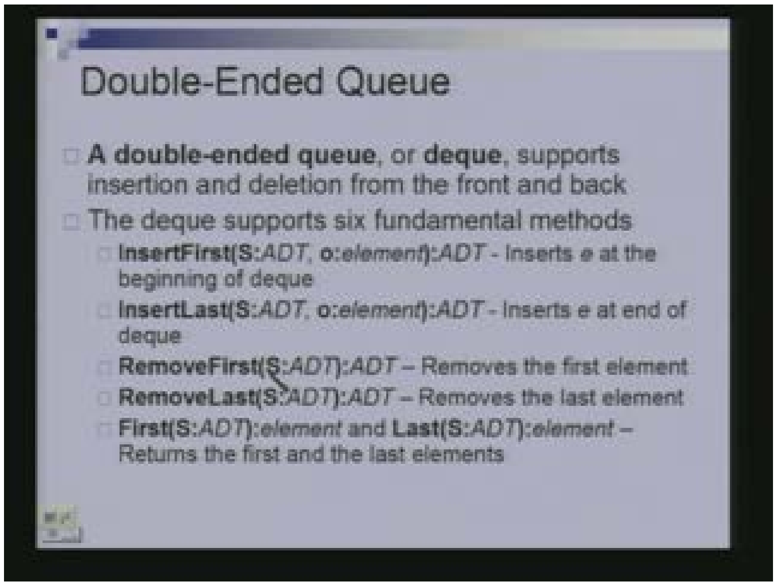
How do I get to that last before node?
I need to go through the entire list, to get to this node.
I am not saying that it is not possible, but it is a very expensive operation. It is not worth while to remove at the tail and so we will remove at the head and add at the tail, which means the front of our queue will be at the head and the rear of the queue would be the tail.

So far we have seen the queue data type. Now I am going to introduce another data type called double-ended queue.

What is the double-ended queue?
It is a queue in which we support, insert and delete operations at both the ends. We have Insert First which is to insert at the front of the queue, Insert Last is to insert at the end of

the queue, Remove First is to remove at the front of the queue and Remove Last is to remove an element at the end of the queue. Also we have the first and the last operations. Such a thing is called double-ended queue, at both the ends we can do both the operations of insert and delete.

(Refer Slide Time: 22:36)



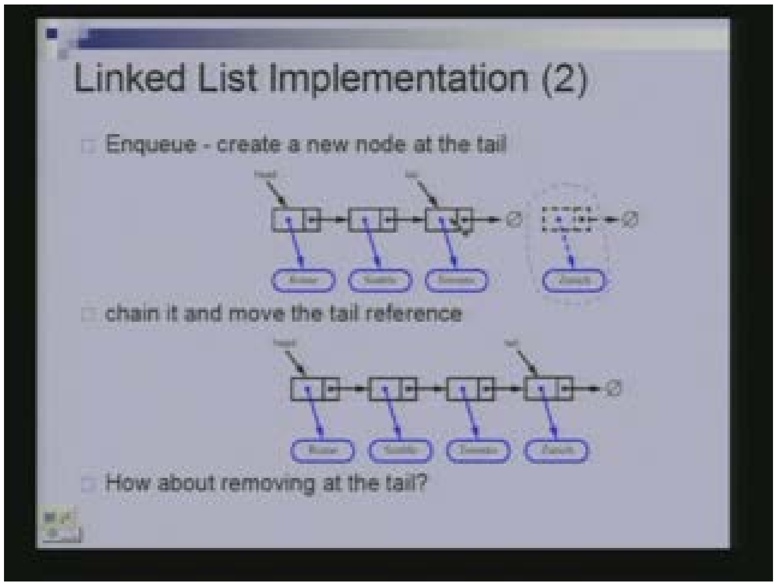
Double-Ended Queue

- **A double-ended queue**, or **deque**, supports insertion and deletion from the front and back
- The deque supports six fundamental methods
  - **InsertFirst(S:***ADT*, **o:***element***):***ADT* - Inserts *e* at the beginning of deque
  - **InsertLast(S:***ADT*, **o:***element***):***ADT* - Inserts *e* at end of deque
  - **RemoveFirst(S:***ADT***):***ADT* – Removes the first element
  - **RemoveLast(S:***ADT***):***ADT* – Removes the last element
  - **First(S:***ADT***):***element* and **Last(S:***ADT***):***element* – Returns the first and the last elements

A singly linked list is not a good idea to implement such a double-ended queue. Why because as I have said repeatedly, we cannot remove the element at the tail or it is very expensive.

What is the good solution to this problem?
We are going to use doubly linked list to implement double-ended queues.

(Refer Slide Time: 23:28)

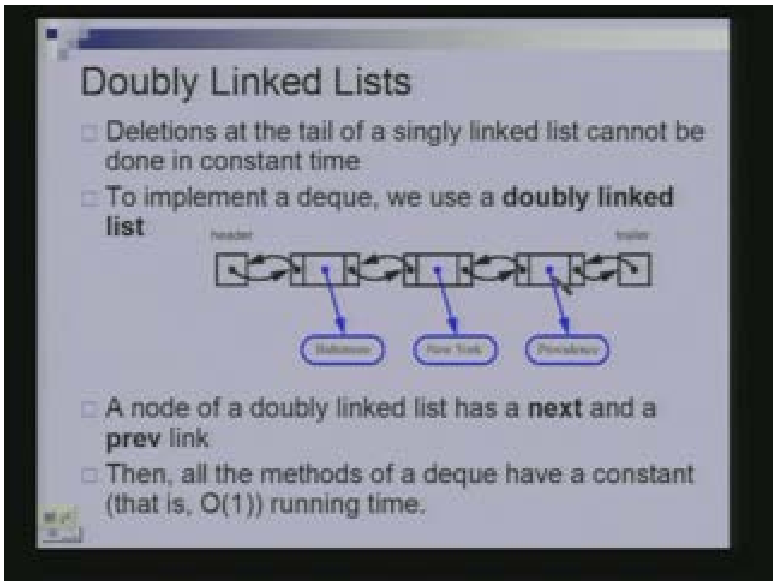


Linked List Implementation (2)

- Enqueue - create a new node at the tail
- chain it and move the tail reference
- How about removing at the tail?

What is the doubly linked list?
A doubly linked list has nodes with two pointers, one is next pointer and the other is the previous pointer. We are also going to have two sentinel nodes. Each node has two pointers, one pointing to the next and one pointing to the previous.

(Refer Slide Time: 24:55)

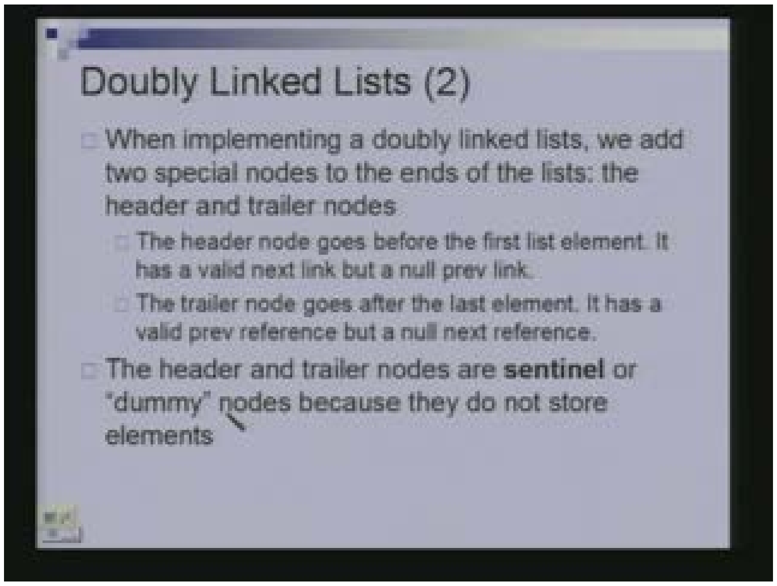


Doubly Linked Lists

- Deletions at the tail of a singly linked list cannot be done in constant time
- To implement a deque, we use a **doubly linked list**
- A node of a doubly linked list has a **next** and a **prev** link
- Then, all the methods of a deque have a constant (that is, O(1)) running time.

Using such a list we can implement all the operations of double-ended queue in constant time. The problem earlier was how to delete the node which is at the end. The head and the trailer nodes are the 2 sentinel nodes. I have a pointer to 2 sentinel nodes and to get to the last element, I just follow the pointer once and get to that element. To delete that node, move to the previous port and set its next pointer to trailer and send the previous pointer of trailer to that node.

(Refer Slide Time: 25:15)

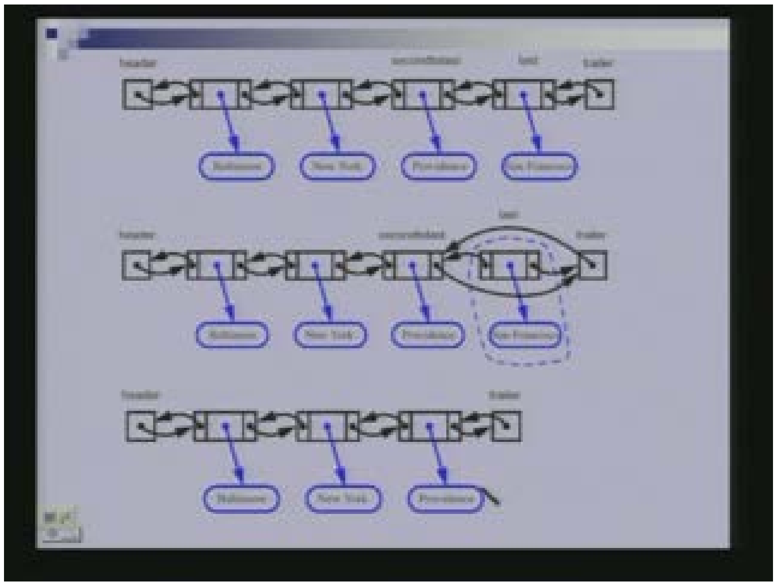


Doubly Linked Lists (2)

- When implementing a doubly linked lists, we add two special nodes to the ends of the lists: the header and trailer nodes
  - The header node goes before the first list element. It has a valid next link but a null prev link.
  - The trailer node goes after the last element. It has a valid prev reference but a null next reference.
- The header and trailer nodes are **sentinel** or "dummy" nodes because they do not store elements

We need header and trailer nodes in a doubly linked list. These nodes are called sentinel nodes or dummy nodes because they do not contain any data inside them and they are just there to mark the start and the end. This is useful.
How do you delete at the end?
I have to delete San Francisco out of this list. All I have to do is make the sentinel node point to the previous node and make that previous node to point to the sentinel node. Then the last node is deleted and in the slide (Refer Slide Time: 25:52) the last one becomes my new list. That was the only thing I could not do in a singly linked list and I have shown it here. Hence all the other operations can be done in constant time.

(Refer Slide Time: 25:52)



Thus using a doubly linked list, we can implement all the operations of double-ended queue in constant time. We can insert at the front, insert at the end, delete at the front or delete at the end all in constant time.

What is meant by constant time?

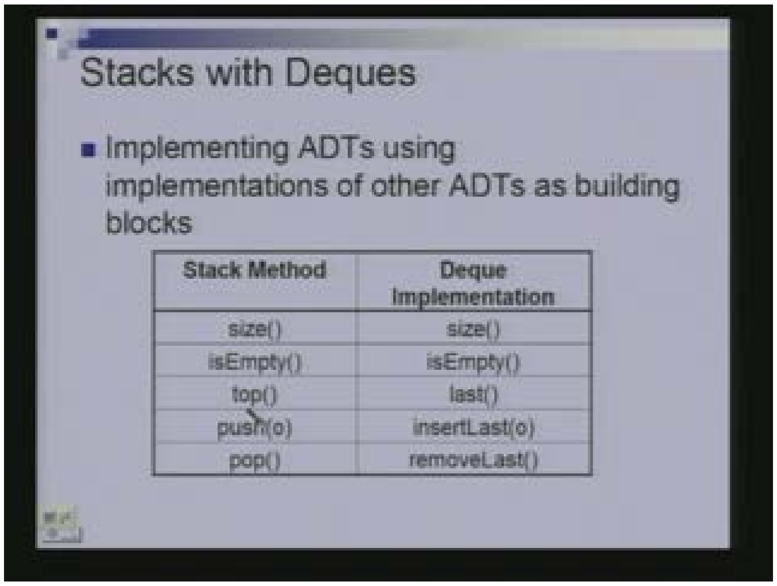
It is the time which is independent of number of elements in the list and your running time will not be depended upon the time.

Double-ended queue is a fairly generic data type, it can used to implement other data types also. Suppose you had an implementation of double-ended queue and you can use that to make a stack or a queue.

Let us see the implementation of a double-ended queue.

I can use the methods of this implementation to implement a stack. For instance in the method top (), the top element of the stack would correspond to the last element of our double-ended queue.

(Refer Slide Time: 27:17)



Stacks with Deques

- Implementing ADTs using implementations of other ADTs as building blocks

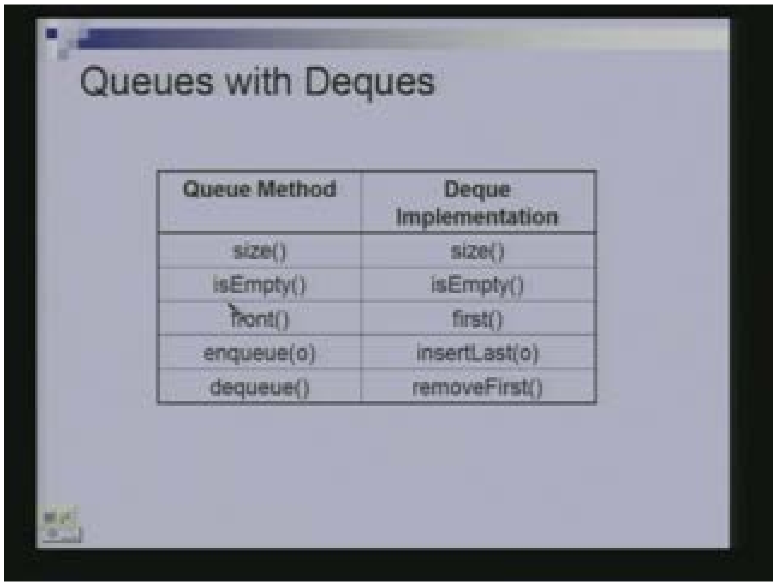
Stack Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
top()	last()
push(o)	insertLast(o)
pop()	removeLast()

Thus the method top () would return the last element of the double-ended queue. The method push () would correspond to inserting at the end of my double ended queue and the method pop () would correspond to deleting at the end of my double ended queue. I could also make the last () to correspond to the front element of my double ended queue. In that case the last () would have been my front and insert Last (0) would have been insert Front () and remove Last () would have been my remove Front (). You can use it either way you like it.

Size () just corresponds to the size of my double ended queue and isEmpty () corresponds to isEmpty of my double-ended queue. Because these are only dependent upon the number of elements in the queue.

Similarly I can use a double-ended queue to implement the queue. Front () gives the first element of the double-ended queue, enqueue () corresponds to last that is it inserts at the rear. When I say dequeue, it removes the first element of the double-ended queue. If I have a dequeue implementation, I can use the methods to implement a stack or a queue or one of these data types.

(Refer Slide Time: 28:59)

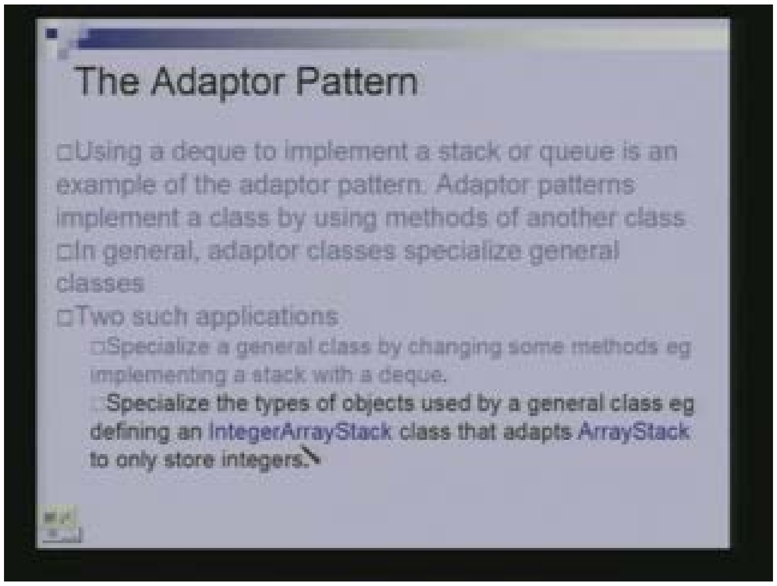


Queues with Deques

Queue Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
front()	first()
enqueue(o)	insertLast(o)
dequeue()	removeFirst()

We have used a double-ended queue to implement a stack or queue and this is an example of an adapter pattern. Thus adapter patterns implements a class using methods of another class. In general, adapter classes specialize general classes and we can have certain applications.

(Refer Slide Time: 29:39)



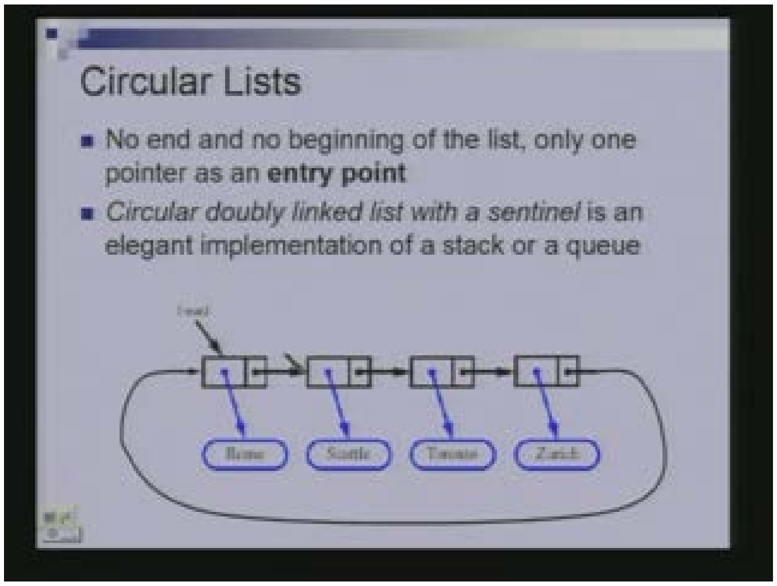
The Adaptor Pattern

- Using a deque to implement a stack or queue is an example of the adaptor pattern. Adaptor patterns implement a class by using methods of another class
- In general, adaptor classes specialize general classes
- Two such applications
  - Specialize a general class by changing some methods eg implementing a stack with a deque.
  - Specialize the types of objects used by a general class eg defining an IntegerArrayStack class that adapts ArrayStack to only store integers.

One application is that we can just implement by changing some methods. For example we can implement a stack by using a double-ended queue. Another application would be an implementation of a stack. We define an interface called stack and implemented it using an array. That implementation is called an array stack.

What are the contents of array stack?
They are any arbitrary objects and I can adapt ArrayStack implementation to an implementation called IntegerArrayStack which only uses integer objects in it. All I have to do is suitably cast the type of the objects that I am pushing in to the stack or removing out of the stack.
There is another data structures called circularly linked list and it is very simple. In that the last element is pointing to the first element of the list.

(Refer Slide Time: 31:38)



Circular Lists

- No end and no beginning of the list, only one pointer as an **entry point**
- *Circular doubly linked list with a sentinel* is an elegant implementation of a stack or a queue

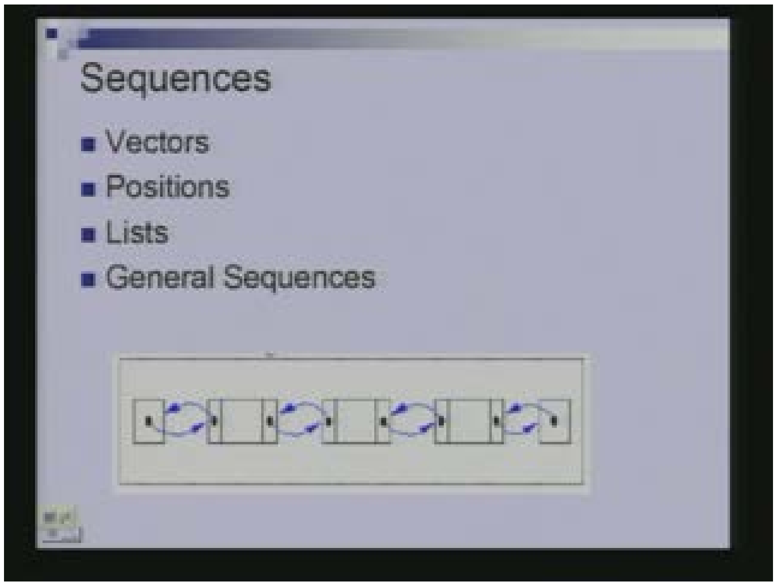
head
Rome
Seattle
Toronto
Zurich

There are no 2 pointers head and tail that is there is only one pointer which is pointing to the start of the circular list and you can use the data structure which is given in the above slide to implement both queue and the stack.
How will you use this data structure to implement a queue?
In a queue we will make the first node as the front of the queue and the last node as the rear of the queue.
How will I add an element at the rear?
To add an element before the first node, make the pointer point to the first node and make the head point to, it is not straight forward because if you mean the big pointer then how you will make this to point to the new node you have just created. We want to create a new node at the end. Make the element which you are inserting to go into the new node and create a new node and copy the element Rome into the new node. Make the head point to that new node and copying is not costly because here you are copying only the reference.

Think about the circular list and it is a very straight forward. In this manner you can insert an element in the queue, if you are using this circular list to implement the queue. Removing an element corresponds to removing the first one.
How do you remove the first one?

If I have to just remove the first element in the list, then how do I make the pointer from the last node to point to the 2nd node. There is a problem in doing this.
What do you do again?
Let us remove the 2nd node and copy the contents of that node to the 1st node. We have to remove the Rome.
How do I remove the Rome?
I copy Seattle to Rome. Thus Rome has Seattle in it and I remove the 2nd node. Copying just means changing the reference. Hence we discusses about queues and double-ended queues.
We are going to the second part where we will quickly look at some sequences. We are going to talk about vectors, positions, list and general sequences. We will be using the data structures like arrays and linked lists to implement these data types.

(Refer Slide Time: 35:06)

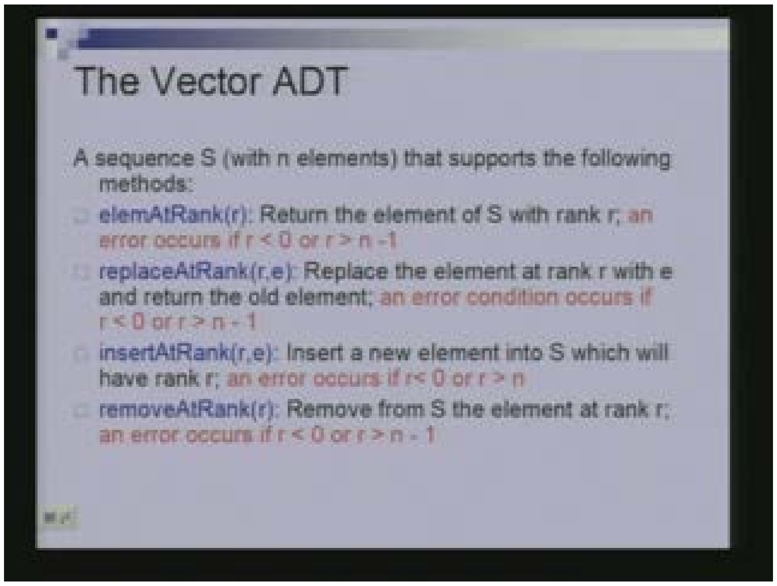


Sequences

- Vectors
- Positions
- Lists
- General Sequences

What is the vector data type?
Vector data type is a sequence of n elements that supports the following methods which are given in the slide below. These are indicative methods and not all the methods.
Essentially in a vector it is a sequence where there is a notion of rank with every element of the sequence. Think of sequence of elements right 7,11,13,19. We know that 7 was the 1st element, 1l was the 2nd element, 13 was the 3rd element and 5 was the 4th element.

(Refer Slide Time: 35:34)



The Vector ADT

A sequence S (with n elements) that supports the following methods:

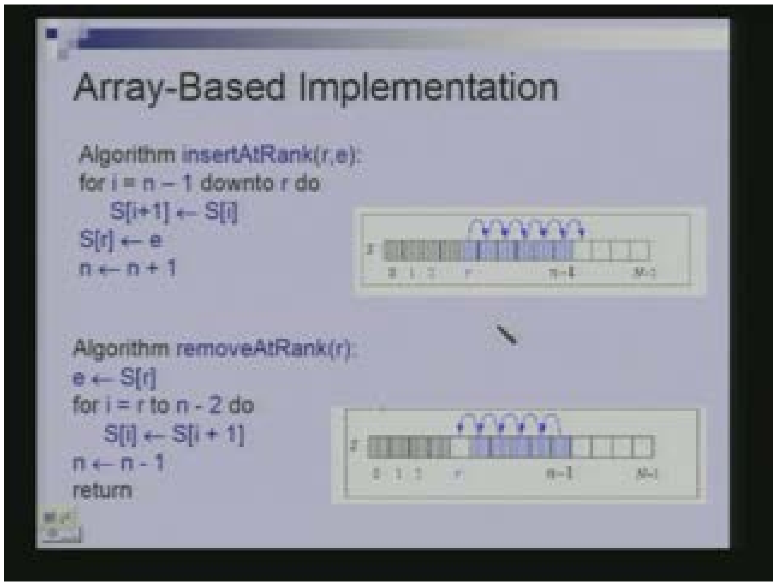
- elemAtRank(r): Return the element of S with rank r; an error occurs if r < 0 or r > n -1
- replaceAtRank(r,e): Replace the element at rank r with e and return the old element; an error condition occurs if r < 0 or r > n - 1
- insertAtRank(r,e): Insert a new element into S which will have rank r; an error occurs if r< 0 or r > n
- removeAtRank(r): Remove from S the element at rank r; an error occurs if r < 0 or r > n - 1

With each element there is a notion of rank, and then I can have methods like elemAtRank r. Rank here corresponds to let us say rank(r) integers. First element was the element at rank 1 and 2nd element was the element at rank 2 and so on.
Suppose if I ask to give the element at rank r or replace the element at rank r by the element e, insert an element e at rank r or delete the element at rank r. I could have such methods.

When I remove the element at rank r, for instance let us say the rank of the students in a particular class. There is a departmental rank 1, the departmental rank 2 and departmental rank 3 and so on. Suppose the departmental rank 4 changes and goes to some other department. The department rank 4 is the rank of the one who had the rank 5 before. The same notion follows and everyone would move up by 1 rank.

Let us see how to implement the data type using arrays.
I am going to have an array, in which I will have the element with rank 1, rank 2 and rank 3 and so on. If I have to insert an element at rank r, I have to put an element in the r^{th} location, which means I have to shift all these elements to one step right. That is what I am doing and I put an element in that location.

(Refer Slide Time: 37:53)



Array-Based Implementation

```
Algorithm insertAtRank(r,e):
for i = n – 1 downto r do
    S[i+1] ← S[i]
S[r] ← e
n ← n + 1
```

```
Algorithm removeAtRank(r):
e ← S[r]
for i = r to n - 2 do
    S[i] ← S[i + 1]
n ← n - 1
return
```

S
0 1 2 r n-1 N-1
S
0 1 2 r n-1 N-1

In a for loop, first we are moving n-1 one step to the right by this statement S [i+1] ← S[i]. First we are doing this for n-1, then n-2 where n-2 is moved one step to the right till r is moved to the one step right. Finally element e is put at position r and the size is increased by 1 where n sores the size of the vector.

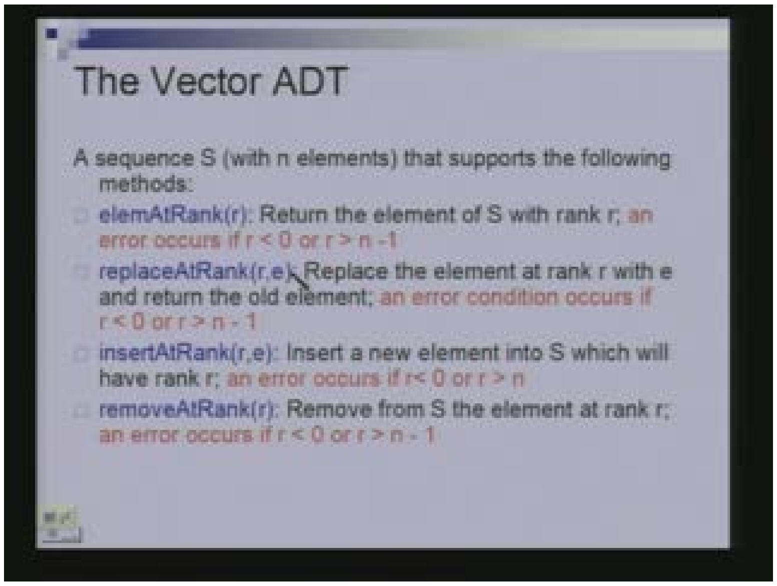
S[r] ← e
n ← n+1

Similarly when I am removing an element at rank r, I am essentially shifting the entire elements one step to the left. All elements starting from r to n-2 and then S[i] gets S [i+1]. At the location r, I will get the element which was sitting at location r+1.

How expensive are these operations in the worst case?
Order n in the worst case because we might have to shift up to n elements to the right or to the left. This implementation is expensive from this point of view, if I have to do these two operations insert at a certain rank or remove at a certain rank. I have 2 in the worst case spent order n time.

The other operations are faster. How much time does the elemAtRank(r) takes, because I just go to the r^{th} location in that array and retrieve the elements sitting there. replaceAtRank (r, e) again order one, because I just go to the r^{th} location and replace that element with element e.

(Refer Slide Time: 40:06)



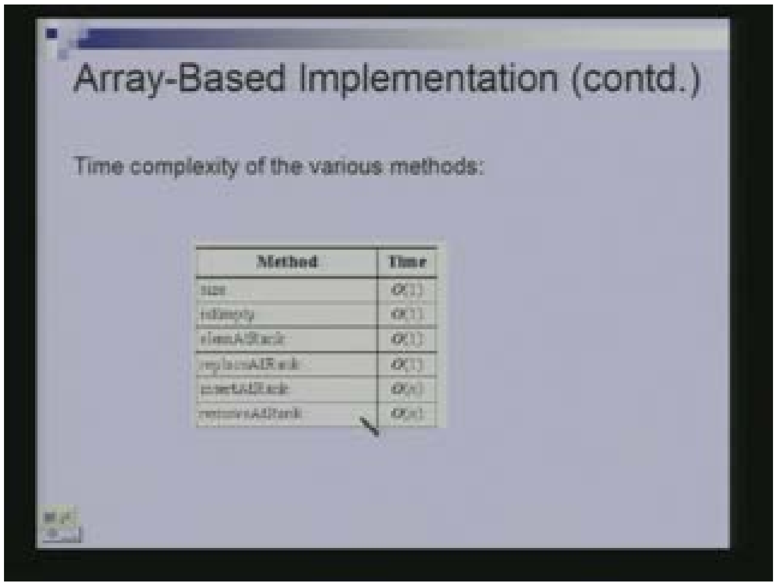
The Vector ADT

A sequence S (with n elements) that supports the following methods:

- elemAtRank(r): Return the element of S with rank r; an error occurs if r < 0 or r > n -1
- replaceAtRank(r,e): Replace the element at rank r with e and return the old element; an error condition occurs if r < 0 or r > n - 1
- insertAtRank(r,e): Insert a new element into S which will have rank r; an error occurs if r< 0 or r > n
- removeAtRank(r): Remove from S the element at rank r; an error occurs if r < 0 or r > n - 1

The chart given below shows the time complexity of various methods. All methods except inserted at rank and remove at rank take constant time but these two methods could take order n time in the worst case.

(Refer Slide Time: 40:33)

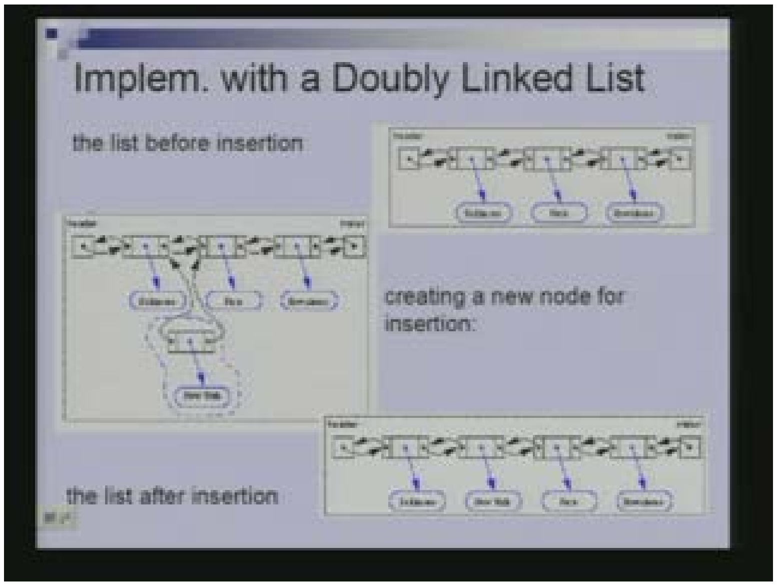


Array-Based Implementation (contd.)

Time complexity of the various methods:

Method	Time
size	$O(1)$
isEmpty	$O(1)$
elemAtRank	$O(1)$
replaceAtRank	$O(1)$
insertAtRank	$O(n)$
removeAtRank	$O(n)$

(Refer Slide Time: 40:25)


Implem. with a Doubly Linked List

the list before insertion

creating a new node for insertion:

the list after insertion

Can you think of some other way of implementing this list?
We can implement through doubly linked list.

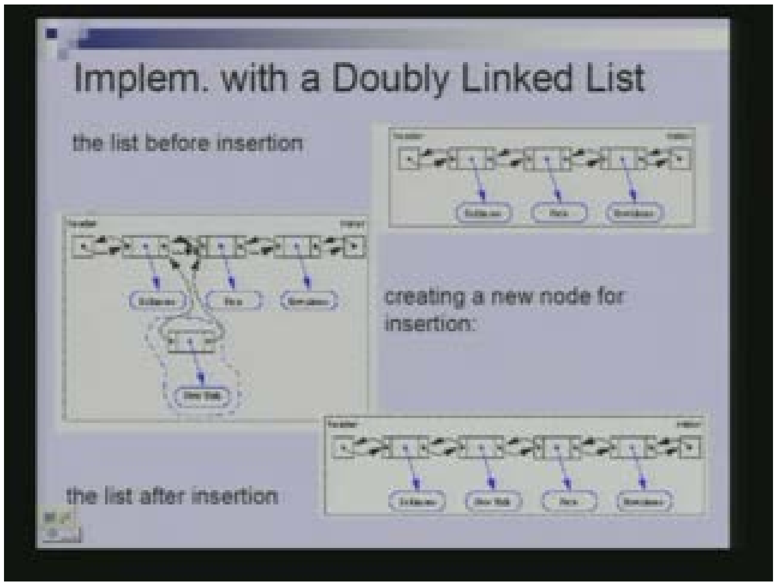
Can you use a doubly linked list to implement a vector?
I am showing you here the operation of inserting at a certain rank.

There are 3 diagrams in the above slide. In the 1st diagram, the 1st node is the header and the next one is the element at rank 1. Following one is the element at rank 2 and the next one is the element at rank 3. Suppose I want to insert an element at rank 2, I have to make a new node and put it between 1 and 3.

How much time does it take?
Create the node and to insert it, I make a pointer point to the next node and make the previous pointer point to the previous node. This is how I insert newyork and the 3rd diagram is the one which I get after insertion.

There are 2 issues. First if I know where I have to insert, then I take constant time but to find out where I have to insert takes order n times. Because if I have to insert at rank 17 then I have to step through that linked list till 17th position and then I would know to insert at that location.

(Refer Slide Time: 41:03)



Implem. with a Doubly Linked List

the list before insertion

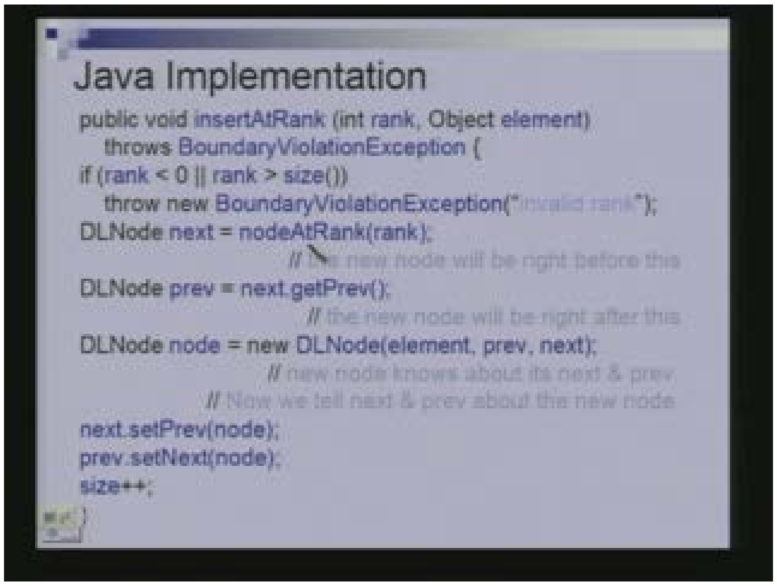
creating a new node for insertion:

the list after insertion

Once I know to insert at this location then it is easy. I will insert the element in 3 or 4 pointer changes.

The following would be a java code for inserting at a rank.
I am assuming the existence of the procedure nodeAtRank (rank). This is the method that I am going to be defining shortly.
What does this method do?
Given a rank, it tells me which is the node at that rank.

(Refer Slide Time: 42:39)

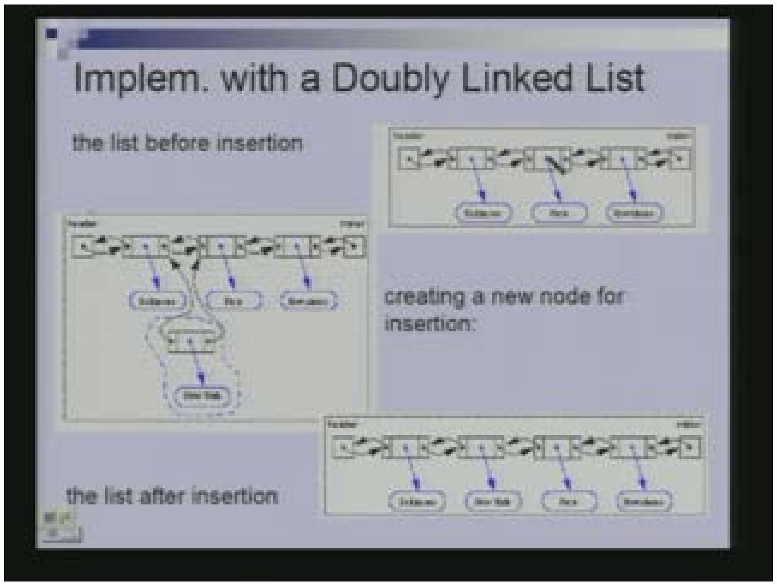


Java Implementation

```
public void insertAtRank (int rank, Object element)
    throws BoundaryViolationException {
if (rank < 0 || rank > size())
    throw new BoundaryViolationException("invalid rank");
DLNode next = nodeAtRank(rank);
                    // the new node will be right before this
DLNode prev = next.getPrev();
                    // the new node will be right after this
DLNode node = new DLNode(element, prev, next);
                    // new node knows about its next & prev
              // Now we tell next & prev about the new node
next.setPrev(node);
prev.setNext(node);
size++;
}
```

For instance, to insert the node at rank 2, first I will call the procedure with rank 2 it will give me the 2nd node of the 1st diagram because that is the node at rank 2.

(Refer Slide Time: 43:05)



Implem. with a Doubly Linked List

the list before insertion

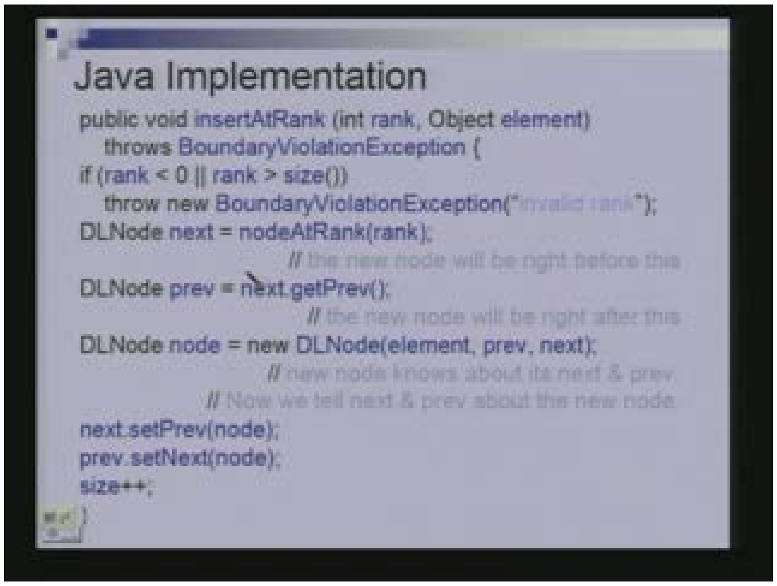
creating a new node for insertion:

the list after insertion

I have to get to the previous node of that node. If I get to this node (next) at rank 2, then I get to the previous node (next.getPrev ()) and this is the node previous to rank 2 which is at rank 1. The new node that I have to insert has to be between next and prev. I create the new node and I set its previous field to refer to the previous node and I set its next field to refer to the next node.

```
   DLNode next=nodeAtRank (rank);
   DLNode prev=next.getPrev ();
   DLNode node=new DLNode (element, prev, next);
```

(Refer Slide Time: 43:25)



Java Implementation

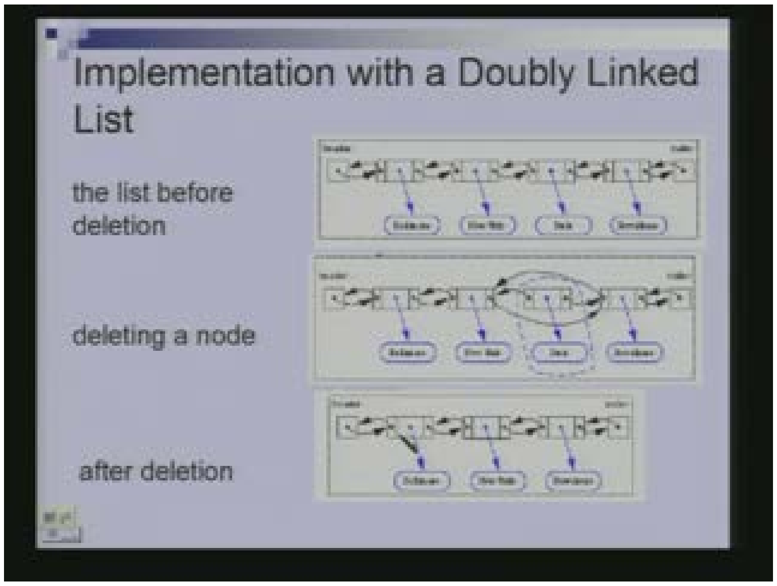
```
public void insertAtRank (int rank, Object element)
   throws BoundaryViolationException {
if (rank < 0 || rank > size())
   throw new BoundaryViolationException("invalid rank");
DLNode next = nodeAtRank(rank);
                    // the new node will be right before this
DLNode prev = next.getPrev();
                     // the new node will be right after this
DLNode node = new DLNode(element, prev, next);
                  // new node knows about its next & prev
            // Now we tell next & prev about the new node
next.setPrev(node);
prev.setNext(node);
size++;
}
```

DLNode prev=next.getPrev (); → this was the node at rank 1 and DLNode next=nodeAtRank (rank); was the node earlier at rank 2. In this manner I create the new node at the appropriate place and then I also need to check the previous and next field of the prev and next node. That is what I am doing here.

```
   next.setPrev (node);
   Prev.setNext (node);
   Size++;
```

Do not get intimated by this code, it is just doing what is shown in the picture. I am assuming the existence of this procedure DLNode next=nodeAtRank (rank) in which, the given rank will tell me which is the node at that rank in the original list.

(Refer Slide Time: 45:07)



Implementation with a Doubly Linked List

the list before deletion

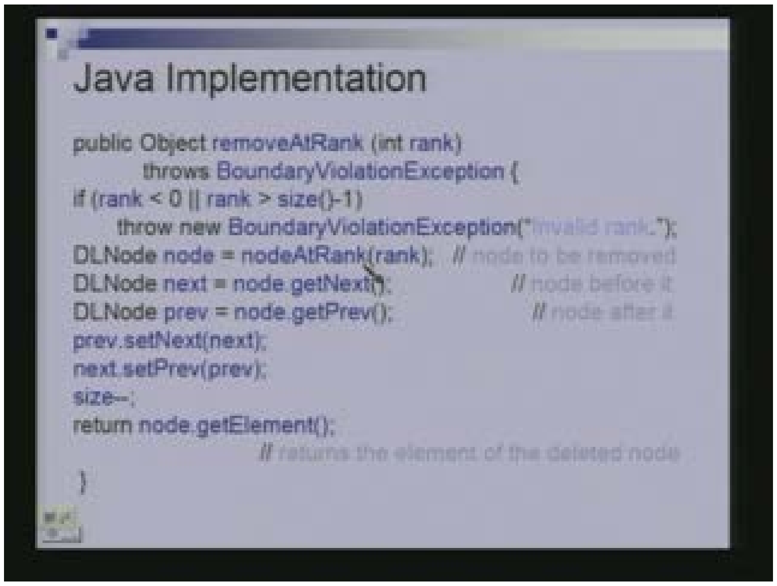
deleting a node

after deletion

I will show you the process of deletion. If I have to remove the element at rank 3, I will first find out the node which is at this rank so I get to the node which is selected in the 2nd diagram and then I have to go to the next node, go to the previous node and update their next and previous pointers. Thus the pointer will point to the next node and previous node and in this manner I will get rid of that node and at the end I will get the 3rd diagram as the final node.

Similarly I can write down the java code for doing this. Once again I am assuming the procedure nodeAtRank, which tells me about the node which is sitting at that rank.

(Refer Slide Time: 45:27)



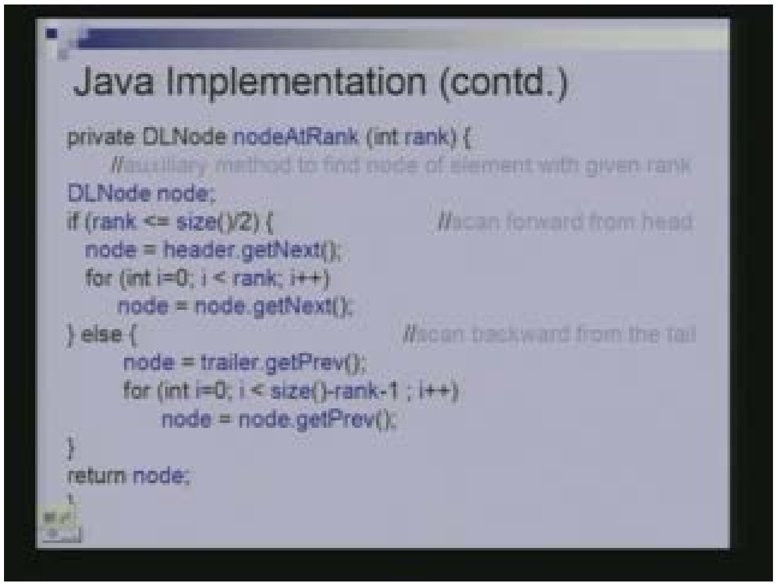
Java Implementation

```
public Object removeAtRank (int rank)
        throws BoundaryViolationException {
if (rank < 0 || rank > size()-1)
      throw new BoundaryViolationException("Invalid rank.");
DLNode node = nodeAtRank(rank);   // node to be removed
DLNode next = node.getNext();          // node before it
DLNode prev = node.getPrev();           // node after it
prev.setNext(next);
next.setPrev(prev);
size--;
return node.getElement();
                    // returns the element of the deleted node
 }
```

How do I implement this procedure nodeAtRank?
There is nothing else I can do except that I march to the list and keep incrementing my counter till I reach that rank. I have done essentially that except a small improvement, that if the rank is less than the number of the size of the list by 2, then I start from the header and if it is more than size by 2 I start from the tail.

Just to small improvement nothing more you do such a thing, because if your list has hundred elements and you are looking for the element at rank 98, then there is no point to start from the header it is better to start from the tail.

(Refer Slide Time: 45:34)



Java Implementation (contd.)

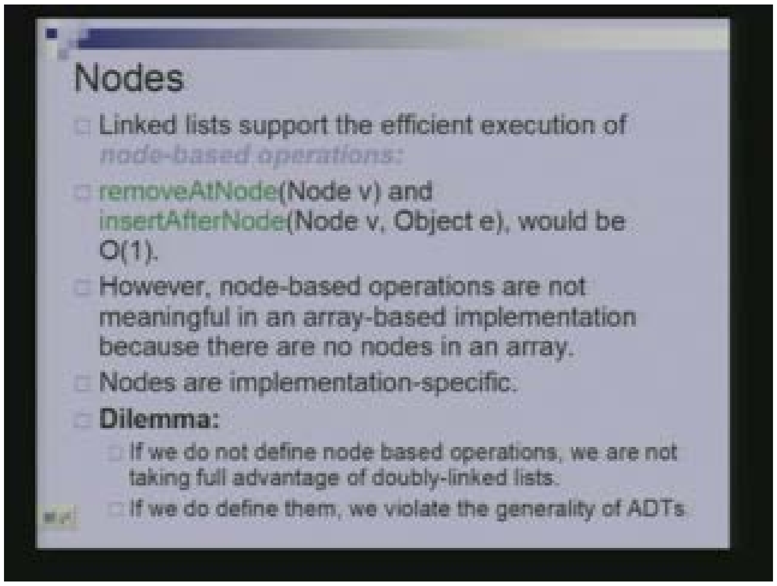
```
private DLNode nodeAtRank (int rank) {
      //auxiliary method to find node of element with given rank
DLNode node;
if (rank <= size()/2) {                    //scan forward from head
   node = header.getNext();
   for (int i=0; i < rank; i++)
        node = node.getNext();
} else {                                   //scan backward from the tail
        node = trailer.getPrev();
        for (int i=0; i < size()-rank-1 ; i++)
             node = node.getPrev();
}
return node;
}
```

That is as far as the vector abstract data type is concerned except that when I say remove the element at a particular rank or insert the element at a particular rank. As you have seen both the implementations we have a problem.

Whether we use an array or a list to do that implementation, we seem to require order n time in the worst case, just to be able to find out where the element correspond to that rank is. In an array, we know the element corresponding to that rank is and we have to move the elements when we insert or delete.

Linked lists are better in supporting node based operations. I have a linked list and I tell you delete this node, if it is a doubly linked list you can delete that node in constant time. If I say this is a node and insert a new node after this node I could insert a new node after that node in constant time or if I say delete the inserted node before this node, again I can insert a node in constant time.

(Refer Slide Time: 47:11)



Nodes

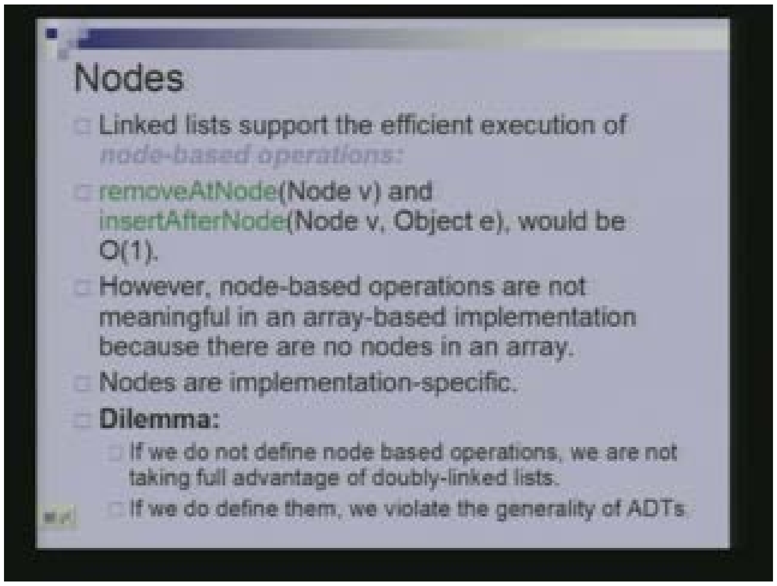
- Linked lists support the efficient execution of *node-based operations:*
- removeAtNode(Node v) and insertAfterNode(Node v, Object e), would be O(1).
- However, node-based operations are not meaningful in an array-based implementation because there are no nodes in an array.
- Nodes are implementation-specific.
- **Dilemma:**
  - If we do not define node based operations, we are not taking full advantage of doubly-linked lists.
  - If we do define them, we violate the generality of ADTs

We have the data structure which is very efficient, which can do constant time operations provided that we give access to the node. Some how I access the particular node at which we want to insert or delete. That is what mentioned below.

removeAtNode (Node v) and
insertAfterNode (Node v, Object e)

You can remove at a node or you can insert after a node and you can insert before a node all in constant time. However when I give you access to a particular node then in some sense, I am also telling you how I have implemented my list. Whether it is a doubly linked list or a singly linked list and what are the pointers and stuff like that.

(Refer Slide Time: 47:56)

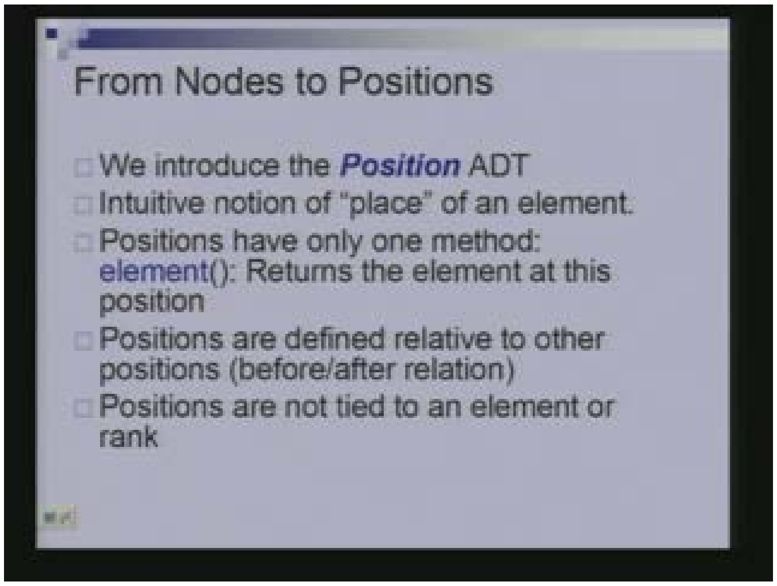


Nodes

- Linked lists support the efficient execution of *node-based operations:*
- removeAtNode(Node v) and insertAfterNode(Node v, Object e), would be O(1).
- However, node-based operations are not meaningful in an array-based implementation because there are no nodes in an array.
- Nodes are implementation-specific.
- **Dilemma:**
  - If we do not define node based operations, we are not taking full advantage of doubly-linked lists.
  - If we do define them, we violate the generality of ADTs.

Suppose I want to hide all those information, so that you can still use node based operation without knowing the actual implementation of how the thing was done. So one can have different implementations. We are going to do this using a notion of positions.
Position is an abstract data type which intuitively captures the place where a certain element is stored. In your data structure, there is only one method which is associated with the position and is the method element.

(Refer Slide Time: 48:57)



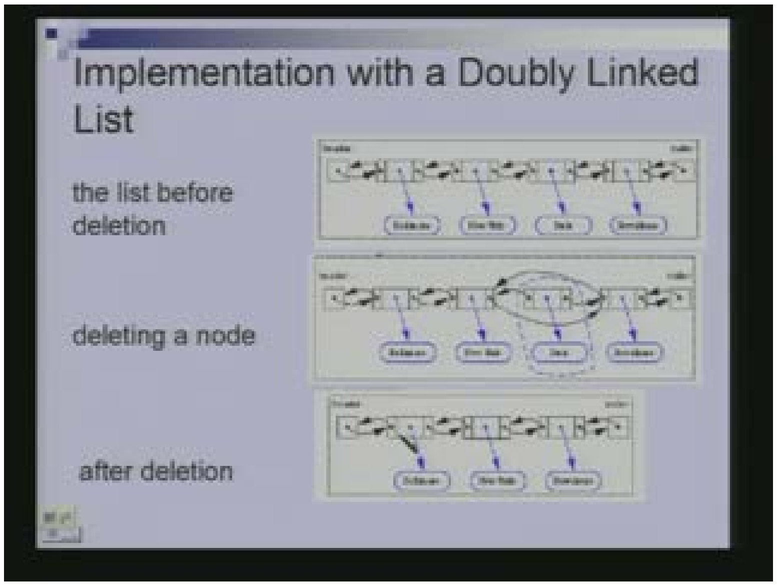
From Nodes to Positions

- We introduce the ***Position*** ADT
- Intuitive notion of "place" of an element.
- Positions have only one method: element(): Returns the element at this position
- Positions are defined relative to other positions (before/after relation)
- Positions are not tied to an element or rank

Given an object of this data type position, I can only call this method element on that object and that will tell me about the element which is sitting at that particular position. If this is not making much sense, then think of position as reference to a particular node. Think of it as a pointer, because using that pointer you can access the element which is situated in the node and nothing else.

You cannot use that pointer to update the next or the previous fields, or you do not even know how the node is implemented. You do not need to know whether the implementer has used a doubly linked list or singly linked list or a circular list. It is an abstract data type which hides all the details and you can only use the method element (), on the abstract data type position.

(Refer Slide Time: 49:41)



Implementation with a Doubly Linked List

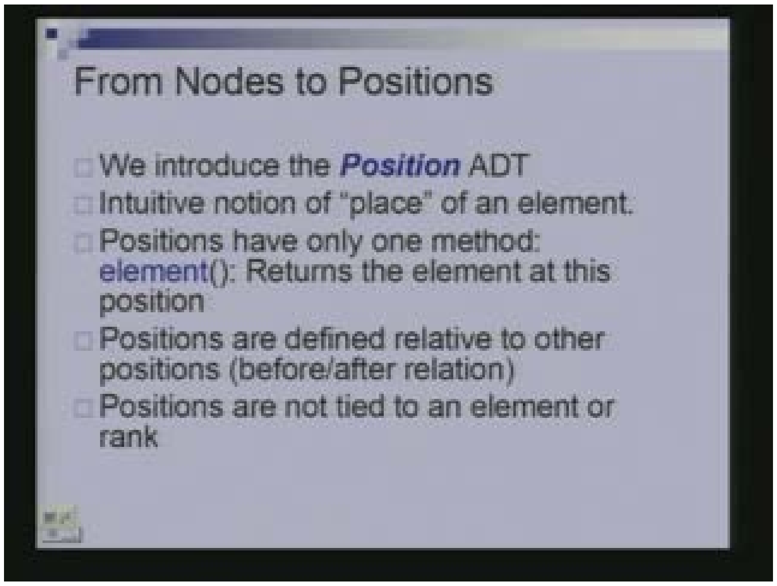
the list before deletion

deleting a node

after deletion

With the notion of position, there will be a relative order of positions jus as in the case of a linked list. There is the 1st element in your linked list, 2nd element and the position is referring to the 1st element or the 1st node or the 2nd node or the 3rd node of the list.

(Refer Slide Time: 50:35)

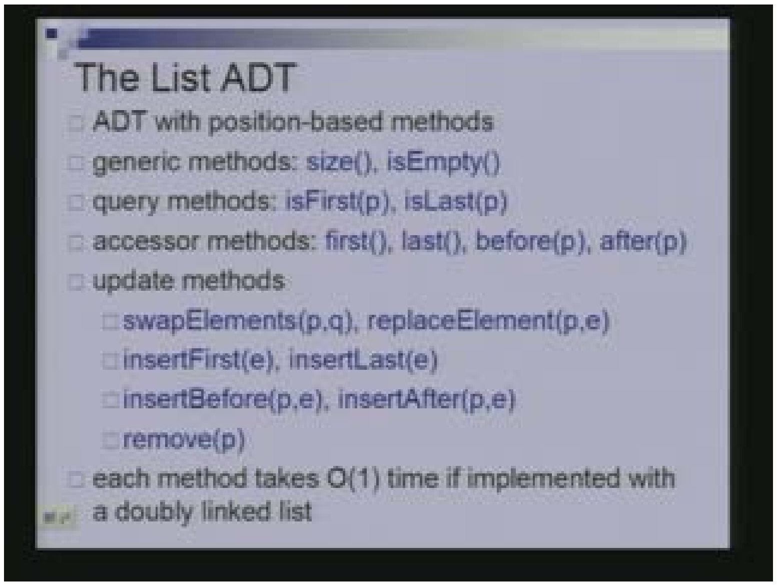


From Nodes to Positions

- We introduce the ***Position*** ADT
- Intuitive notion of "place" of an element.
- Positions have only one method: element(): Returns the element at this position
- Positions are defined relative to other positions (before/after relation)
- Positions are not tied to an element or rank

Similarly 1st position, the 2nd position, the 3rd position and so on. Given a position that, there is the notion of the position before which refers to the node before that position and a position after that position.

(Refer Slide Time: 51:23)



The List ADT

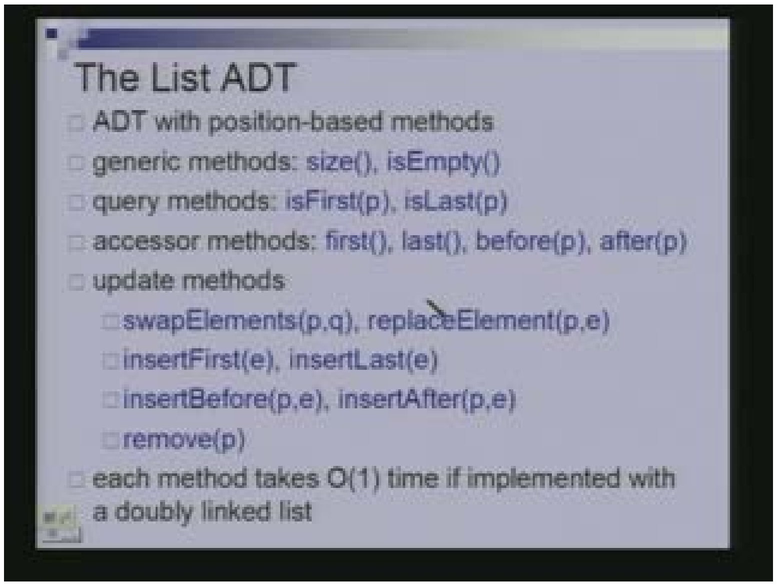
- ADT with position-based methods
- generic methods: size(), isEmpty()
- query methods: isFirst(p), isLast(p)
- accessor methods: first(), last(), before(p), after(p)
- update methods
  - swapElements(p,q), replaceElement(p,e)
  - insertFirst(e), insertLast(e)
  - insertBefore(p,e), insertAfter(p,e)
  - remove(p)
- each method takes O(1) time if implemented with a doubly linked list

We can now define a list abstract datatype which uses the positions.

What would this abstract datatype have?
It would have generic methods like size () and isEmpty () and it could have query method, given a particular position I can have a method which asks is this the first

position of my list. If it is this will say yes and otherwise say no and whether it is the last position of the list. I can have excessive methods like first (), last (), before (p) and after (p).

(Refer Slide Time: 53:11)



The List ADT

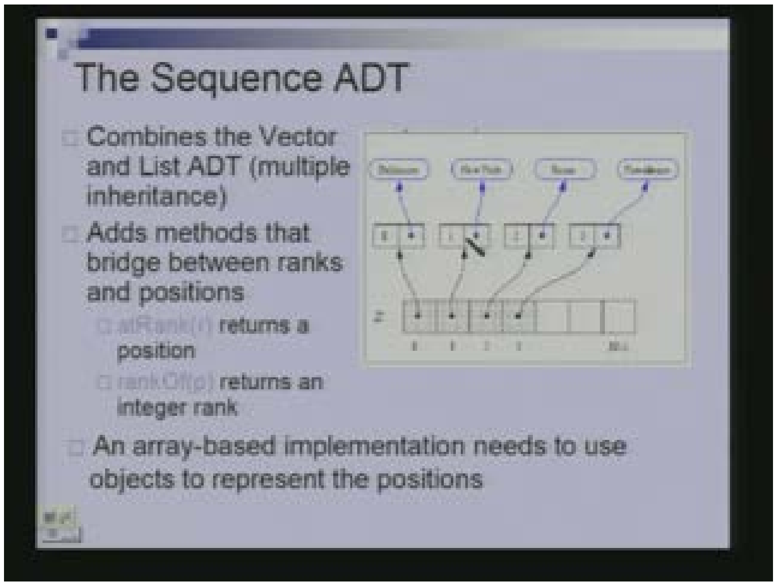
- ADT with position-based methods
- generic methods: size(), isEmpty()
- query methods: isFirst(p), isLast(p)
- accessor methods: first(), last(), before(p), after(p)
- update methods
  - swapElements(p,q), replaceElement(p,e)
  - insertFirst(e), insertLast(e)
  - insertBefore(p,e), insertAfter(p,e)
  - remove(p)
- each method takes O(1) time if implemented with a doubly linked list

First will give me the first position, last would give me the last position, before (p) will give me the position before this position p and after will give me after this position p. I can have update methods like swapElements (p, q).
What does this do?
Given a positions P and Q, it swaps the contents of these positions. Whatever may be the elements sitting at these 2 positions it swaps the contents. I can replace the element at position p with e (replaceElement (p, e)) and similarly I can insert the element e (insertFirst (e)) at the very first position. I can insert the element e (insertLast (e)) at the last position and so on. Using a doubly linked list you can actually implement all of these methods in constant time.

The list abstract datatype is just as the same as your linked list data structure except that we are getting an abstract datatype implementation of it. We are trying to capture all of those methods that you can do on a linked list as an abstract datatype. This datatype can be implemented using a double linked list and it can be implemented using a singly linked list except that it is more efficient if you implement it using a doubly linked list. In the doubly linked list all of these methods can be done at a constant time. Using a singly linked list some of these methods might take linear time in the worst case.

Finally we have the notion of a sequence abstract data type. We talked of the vector abstract data type where there is a notion of rank associated with each element. Then there is a list data type where there is a notion of positions and the sequence abstract data type has both of these. It combines the vector and the list abstract data type and it inherits both of these interfaces and that is multiple inheritance.

(Refer Slide Time: 53:19)



The Sequence ADT

- Combines the Vector and List ADT (multiple inheritance)
- Adds methods that bridge between ranks and positions
  - atRank(r) returns a position
  - rankOf(p) returns an integer rank
- An array-based implementation needs to use objects to represent the positions

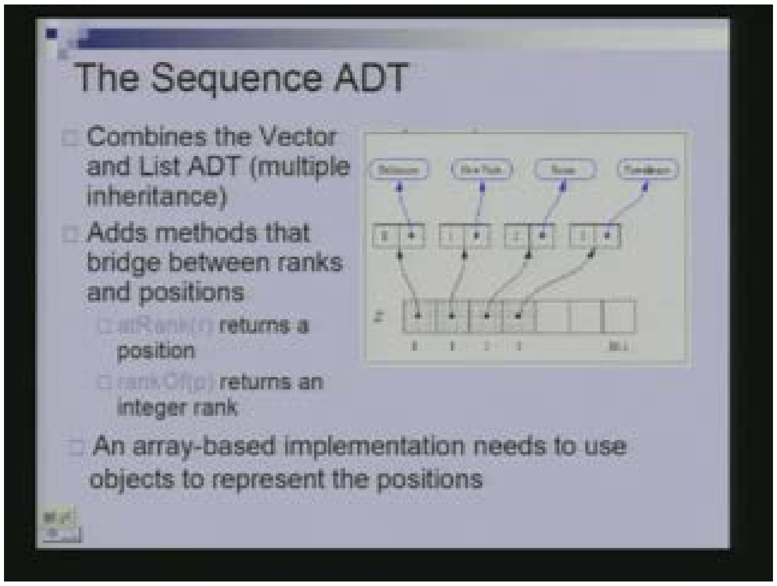
Besides the methods that are listed for vector and list abstract data type, it has two additional methods which helps you to go from one to other. Given a particular rank r, the method atRank(r) will return me the position corresponding to this rank. Given a position p the method rankOf (p) will tell me the rank corresponding to this position.

You could have an implementation of the kind which was given in the slide for a sequence.
In the above slide, given an array in which each element of the array refers to the position and the point 2 is same in both the cases. With the given particular location, I can identify the rank which it corresponds to by looking at the element.

How is the method rankOf (p) implemented?
P corresponds to a position, a position here is the thing which is given in the middle of the diagram. Given a particular position and how do I know the rank corresponding to that position. I just look in to the 3rd element that gives me the rank corresponding to that position. Given a particular rank how do I determine the position corresponding to that rank. Suppose you gave me rank 1, when I follow 1st reference, 1 is the position corresponding to this rank. At that position there is an element stored which is newyork. At the position besides the element, there is something else stored which is kind of provides cross reference.

At each of these positions I have an element stored and a rank of that element in my sequence. Suppose I had to insert an element at rank 2, I am going to create new position and the element would sit in that position and 2 would refer to that position and all of these will have to move to one step right. Not only have to move to the right, we have to change the ranks and update the position. Again inserting at the particular rank will take order n time of the worst case and similarly deleting an element. If I had given particular position and if I wanted to delete the element at that position.

(Refer Slide Time: 56:07)



The Sequence ADT

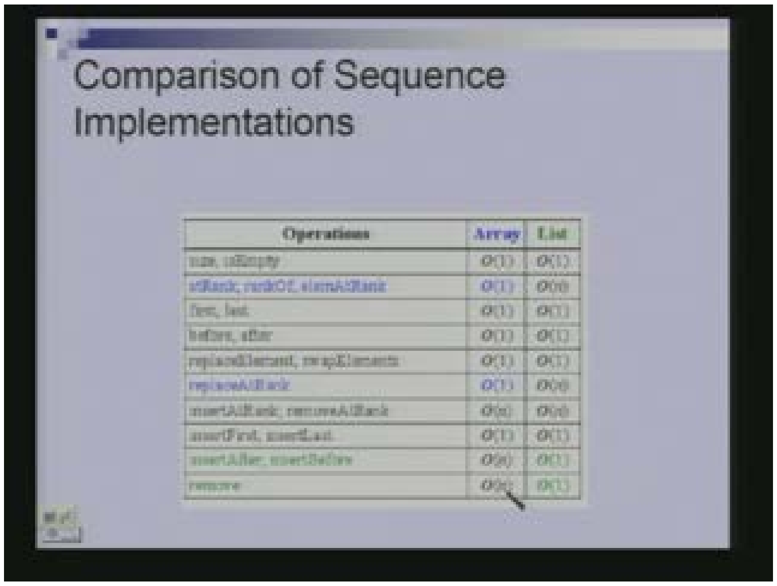
- Combines the Vector and List ADT (multiple inheritance)
- Adds methods that bridge between ranks and positions
  - atRank(r) returns a position
  - rankOf(p) returns an integer rank
- An array-based implementation needs to use objects to represent the positions

How do we delete an element at a certain position in the case of a doubly linked list? You need to think about this. So leave it as an exercise.

This is a comparison of sequence operations, you can implement a sequence using an array in the picture I have shown you previously and you can also implement a sequence using a doubly linked list. This would be the worst case of running time.

You can see in the case of an array implementation, if you want to insert an element at a certain rank or you want to remove an element at a certain rank. It will take order n time. If you want to insert after or insert before a certain position, this will also take order n time and if you need to remove an element at a certain position, this will also take order n time.

(Refer Slide Time: 57:51)



Comparison of Sequence Implementations

Operations	Array	List
size, isEmpty	$O(1)$	$O(1)$
atRank, rankOf, elemAtRank	$O(1)$	$O(n)$
first, last	$O(1)$	$O(1)$
before, after	$O(1)$	$O(1)$
replaceElement, swapElements	$O(1)$	$O(1)$
replaceAtRank	$O(1)$	$O(n)$
insertAtRank, removeAtRank	$O(n)$	$O(n)$
insertFirst, insertLast	$O(1)$	$O(1)$
insertAfter, insertBefore	$O(n)$	$O(1)$
remove	$O(n)$	$O(1)$

Not so in the case of a doubly linked list because then you can just zap out the element from there. You can just update the pointers before and after and do these in constant time. But then what becomes more expensive is, because in a doubly linked list you cannot figure out the rank of an element. I have to go to the entire list to figure out the rank. Any rank based operation will take order n time, whether you want to find the rank of n element or you want to find out the element at a particular rank, find out the position corresponding to certain rank, all of these would take order n time.

We learnt about queues, double ended queues and also how to use linked list and doubly linked list to implement the these data types. Then we also looked at the vector abstract data type, the list abstract data type which is essentially a concretization of the linked list data structure and we also looked at sequence data types which is basically inheriting all the methods of your list data type and your vector data type.