Functional Programming in Haskell

Madhavan Mukund and S. P. Suresh

Chennai Mathematical Institute

Module # 05

Lecture - 02

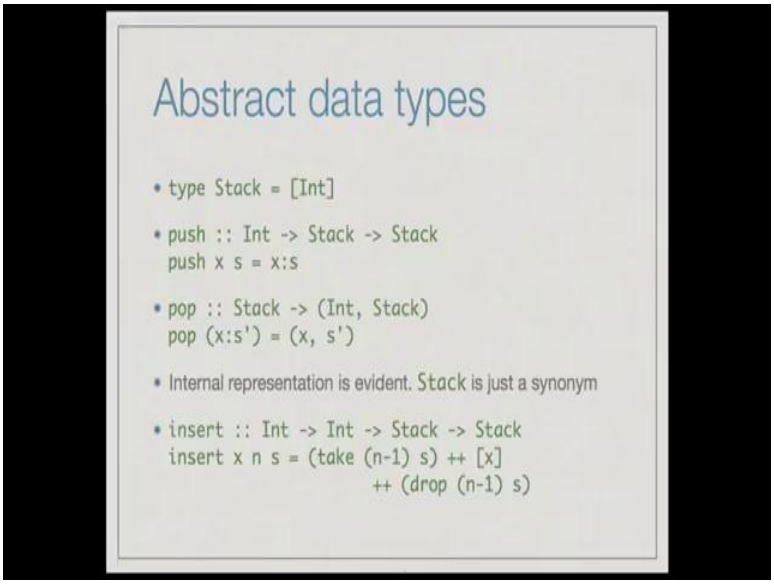
Abstract Data Types

In this lecture, we shall be introducing Abstract Data Types in Haskell.

(Refer Slide Time: 00:07)

Consider the example of a stack data type, a stack is a collection of integers, stack one on top of the other. It supports two operations push and pop, push places an element on top of the stack and pop removes the top most element of the stack. We can see that this behavior is similar to lists, the push corresponds to adding a new element at the head of a list by using the colon operator, pop corresponds to retrieving both the head and tail of the list.

(Refer Slide Time: 00:47)

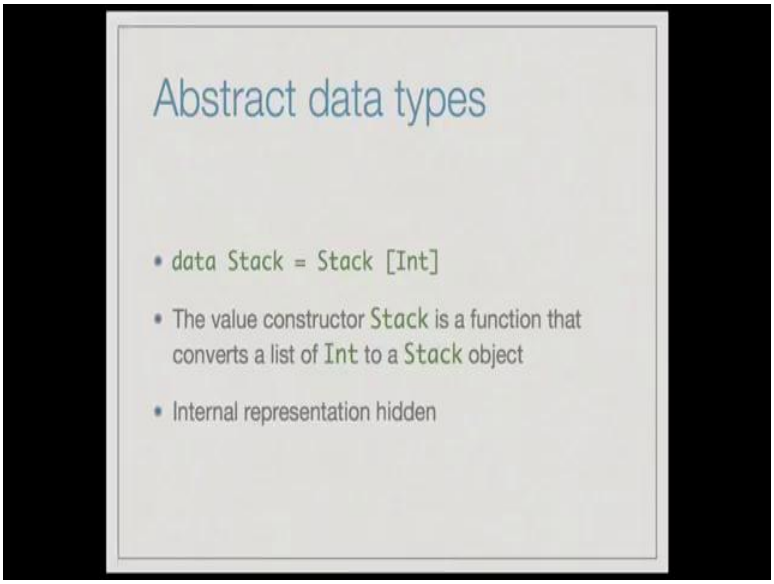


With this in mind, here is a definition of stack, we define stack as a type synonym for list of integers, type stack equals list int. Now, the push function can be defined as follows, push a function which takes two arguments and integer and a stack as inputs and outputs a stack push x is nothing but, x colon s. The head of the list is the top of the stack, pop is a function that takes a stack as input and produces an int and a stack as output. The int is the top element of the stack and the stack that is output is the stack after the top element has been removed.

The definition of pop is as follows, pop of x colon s prime equals the pair x comma s prime. So, with this definition the internal representation of stack is very evident, stack is just a synonym for list of int's. The drawback is that this allows one to write functions that uses the internal representation. For instance, here is a function insert that inserts an element at the nth position from the top of the stack, insert x n s, the x is an element to be inserted, n is the position and s is the stack and the output is also the stack, insert x n s equals take n minus 1 of s plus plus the singleton list x plus plus drop n minus 1 of s.

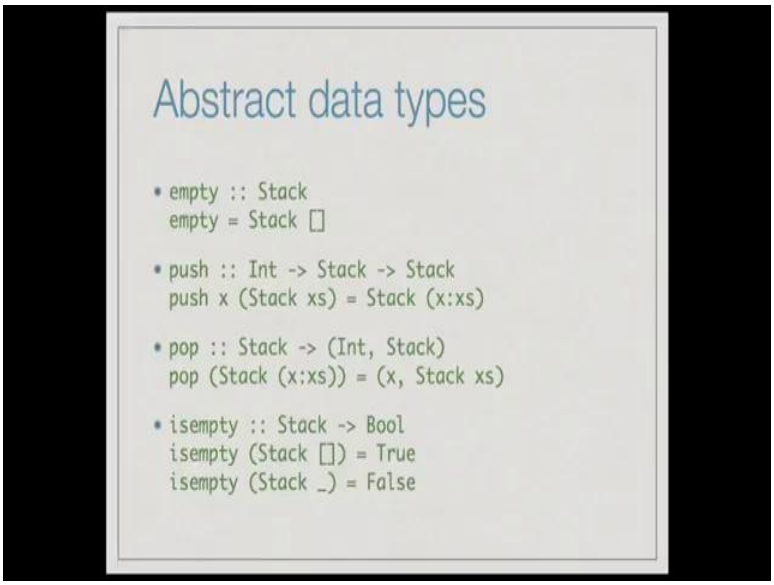
You see that here the internal representation of the stack, namely that it is just a list of integers is used. One does not always want to allow such a use, for a stack data type you would want to allow only the push and pop operations and perhaps to check if the stack is empty.

(Refer Slide Time: 02:51)



This motivates the definition of stack as an abstract data type hiding the internal representation. We can define stack as a user defined data type, data stack equals stack list int. Recall that the stack on the left is the name of the type, the new type that we are defining and the stack on the right is so called the value constructor. The stack on the right is just a function from the list of int to stack, the value constructor stack is a function that converts a list of int to a stack object. The internal representation is hidden, you can access a stack only through it is constructor.

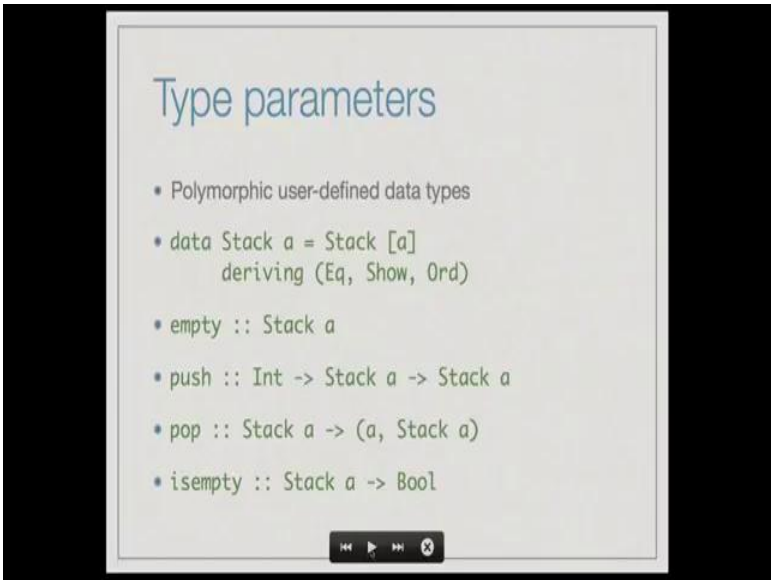
(Refer Slide Time: 03:44)



Now, you can define the following functions on this new data type, empty which returns an empty stack and the definition is empty equals stack empty list, push which takes an

internal stack and returns the stack, push of x and stack x is stack of x colon x s. Here we use pattern matching on the user defined data type, which is something familiar to us. Pop is a function that takes a stack and returns the int under stack, pop of stack x colon x s equals the pair x comma stack x s, isempty is the function that checks if the stack is empty, it is a function from stack to Bool, isempty of stack empty list equals true, isempty of stack anything else is false.

(Refer Slide Time: 04:40)



If the stack data type that we defined earlier was very specific in the sense that it stored only integer values. We might want to implement a stack that stores data of any type whatever. This is achieved by using polymorphic user defined data types, which use type parameters. The definition is as follows, data stack a equals stack list of a, recall that the stack on the left is a type name and the stack on the right is a constructor.

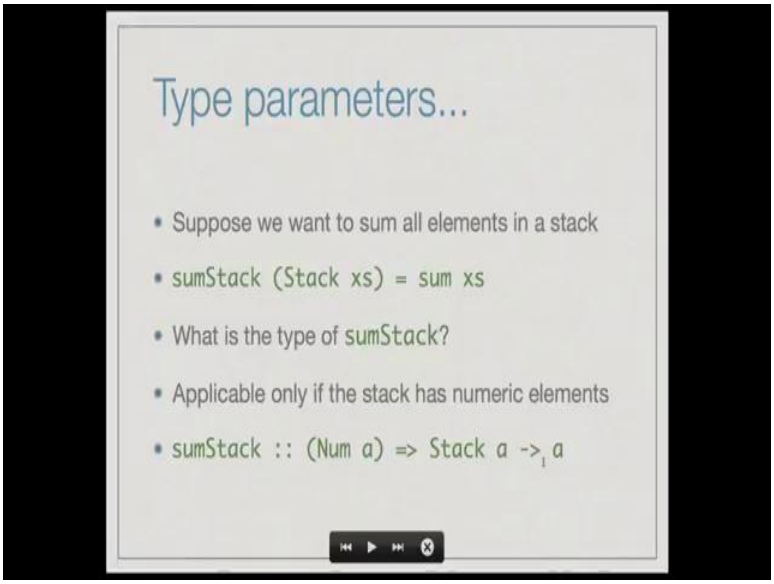
When we use type parameters, the functions on the right are called value constructors and the name of the type itself is called a type constructor. Because, for every instantiation of the type a, you get a new type stack a. For instance, if a is int you will get a stack of integers, if a is float you will get a stack of floats etcetera. So, the stack on the right is called a value constructor and the stack on the left is called a type constructor.

In a polymorphic type, the value constructor is nothing but, a polymorphic function which takes a list of a as an input and produces an object of types stack a as an output. In this particular case we are also deriving Eq, Show, Ord, etcetera for this data type. We can define the following functions again, empty which is a function that just returns a

stack of type a, push which is a function that takes an int and a stack a as an input and produces a stack a as output.

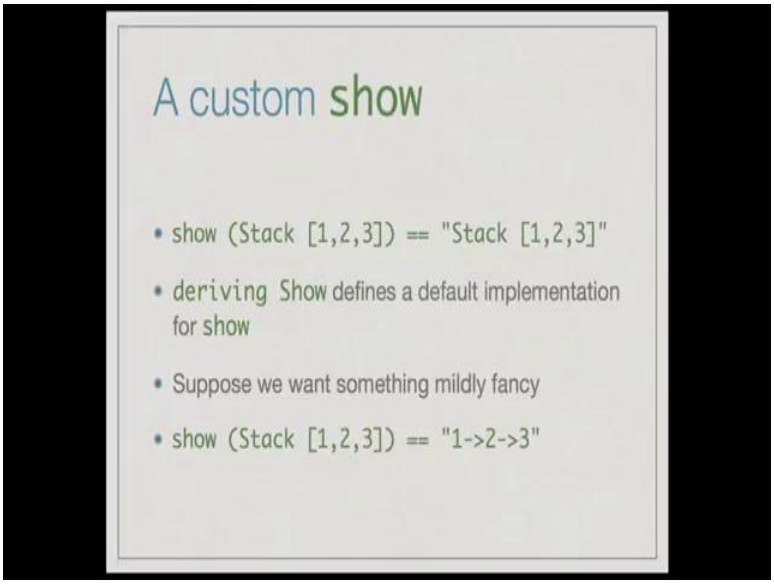
Pop which is a function that takes stack a as input and produces the pair a comma stack a, one value of type a and a stack which is of type stack a as output, isempty is the function that takes a stack as input and produces a Bool as output. The function definitions are exactly the same as given in the previous slide, except now that the types are more general.

(Refer Slide Time: 07:00)



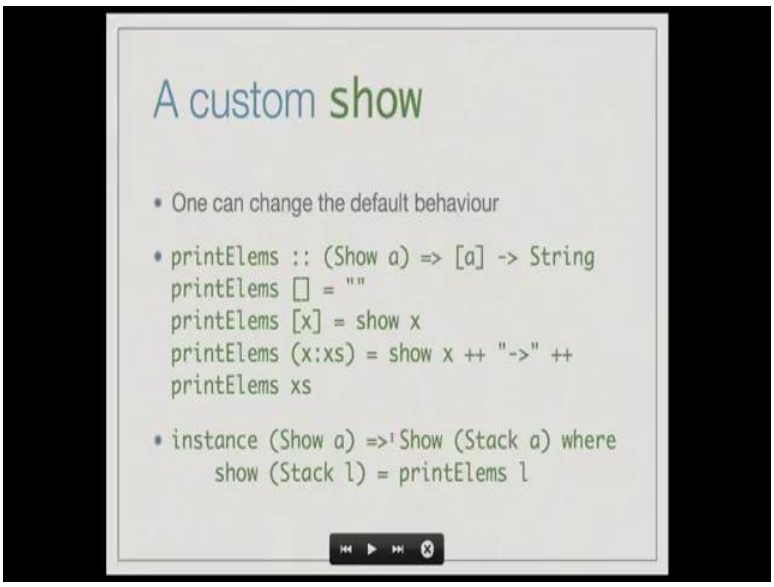
Sometimes functions on these polymorphic data types will not be completely polymorphic, but only conditionally polymorphic. Here is an example, suppose we want to some all elements in a stack you would achieved it as follows, sumstack of stack access is just sum the functions sum applied to access, the function sum adds all the elements in the input list. What is the type of sumStack? Notice that, the type of sum, sum is a polymorphic function which is conditionally polymorphic, it is type is num a implies list a arrow a. Therefore, the sumStack function is applicable only if the stack has numerical elements, in other words the sumStack function has type num a implies stack a arrow a.

(Refer Slide Time: 08:01)



Earlier we said we can derive show, we can derive stack as a type which belongs to the class show. But, this defines the default implementation for show, show stack list 1, 2, 3 is just this string which says stack followed by a representation of the list 1, 2, 3 within square brackets. Suppose you want something fancier, let say we want to say show of stack 1, 2, 3 is 1 arrow 2 arrow 3 within goods. The string 1 arrow 2 arrow 3 this means that you would have to define our own custom show function.

(Refer Slide Time: 08:51)

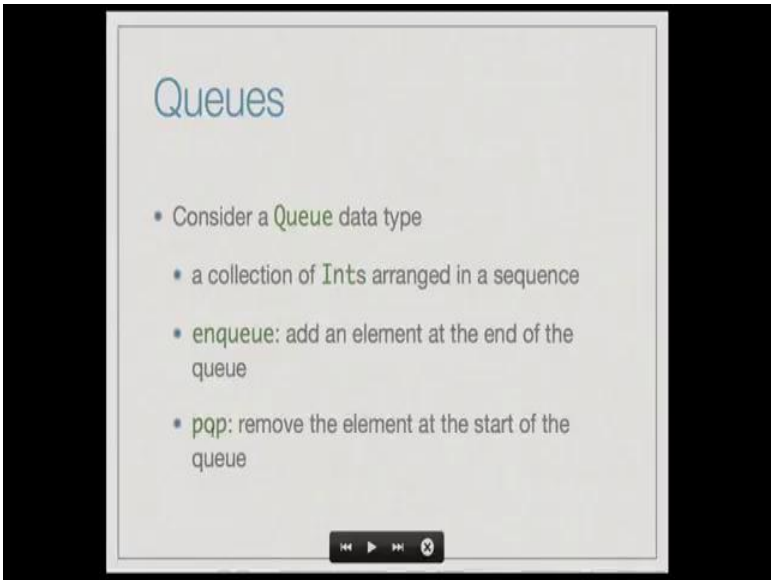


One can change the default behavior of show as follows, we say instance show a implies show stack a, this declaration means that stack a is an instance of the type class show provided a is an instance of the type class show, where the new definition of show the

small s show that we are defining is show of stack l equals print elements l. The print elements function is given here, print elements is the function from list a to string provided a belongs to the type class show, print elements of the empty list is just the empty string, print elements of the singleton list containing x is just show of x, print elements of x colon x is, is show of x plus plus the arrow, the string consisting only of the arrow symbol plus plus print elements of the x s.

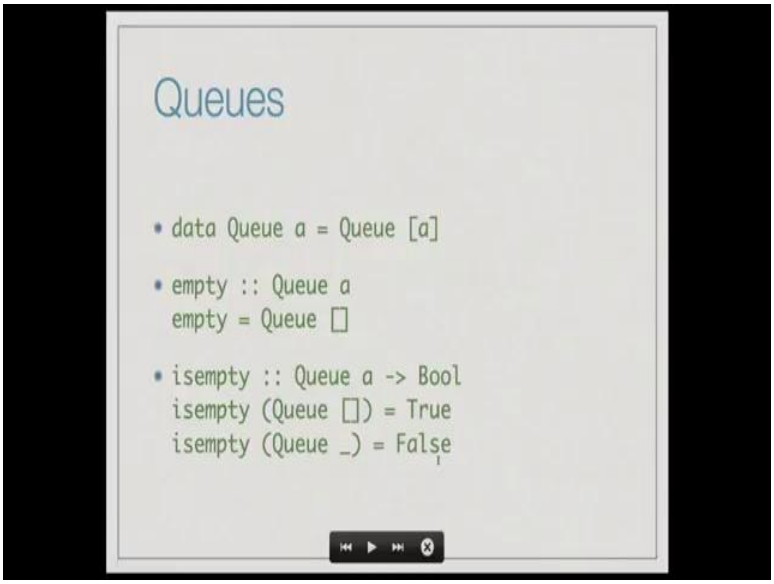
Notice the recursive called here, in this manner we can define custom implementations of the functions that are provided by type classes. If we just say deriving capital show you would get a default implementation of the show function. But, we are always allowed to redefine the show function in which case we allow to declare our data type to be an instance of the type class show.

(Refer Slide Time: 10:30)



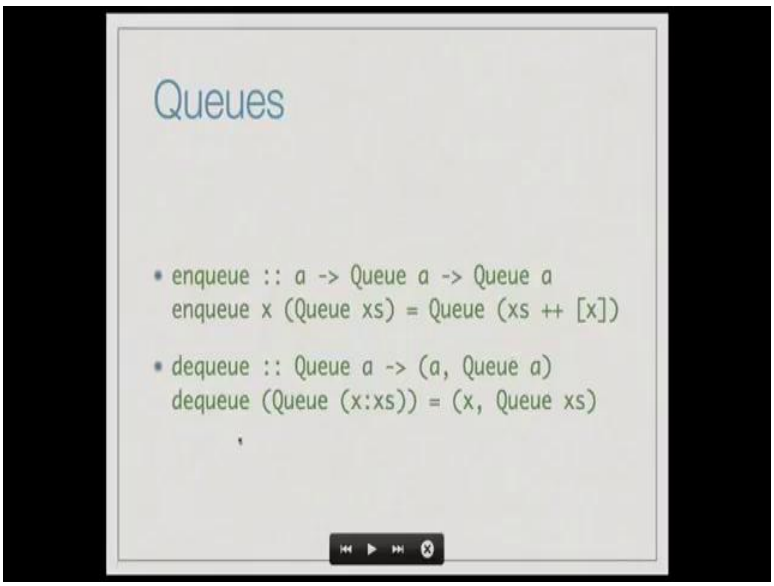
Let us consider another data type queue, a queue is a collection of integers arranged in a sequence, enqueue adds an element at the end of the queue, dequeue removes the element of the start of the queue.

(Refer Slide Time: 10:48)



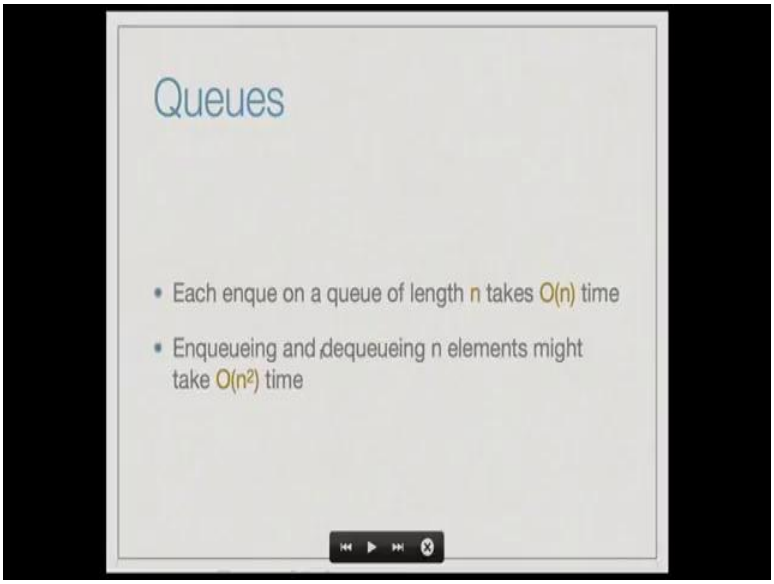
Here is the definition, data queue a equals q list a, recall again that queue is the type constructor and the queue on the right is a value constructor, which is a function from list a to the type q a. The empty q is just given by q empty list, isempty is a function from q a to Bool, isempty of q empty list equals true, isempty of q anything else is false.

(Refer Slide Time: 11:24)



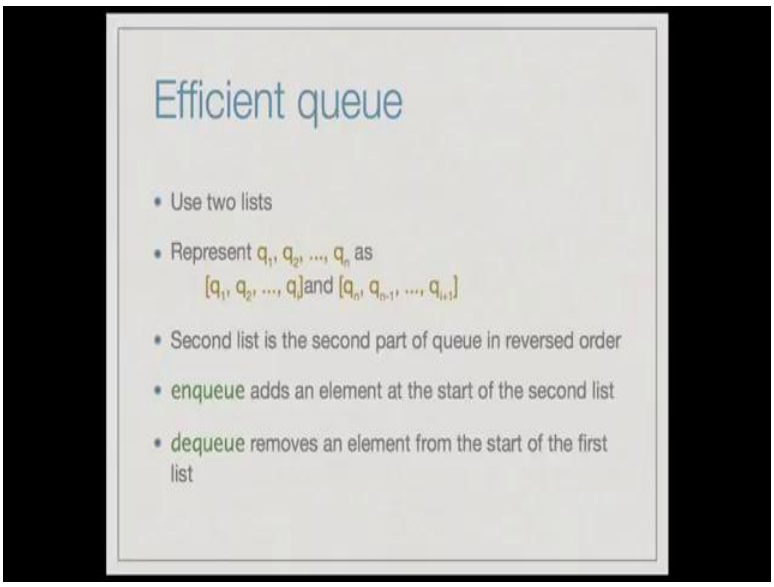
Enqueue adds an element to the end of the list, so enqueue x q x s equals q of x s plus plus the singleton list x, dequeue is a function that takes a q as input and produces an element of type a and q a object as output dequeue of q x colon x s is the pair consisting of x and q x s.

(Refer Slide Time: 12:01)



In this implementation each enqueue on a queue of length n takes order n time, because we are adding the element at the end of the list. So, enqueueing and dequeueing and n elements might take order n squared time depending on the operations that we use, this is the worst case time that could be consumed by a sequence of n enqueue and dequeue operations.

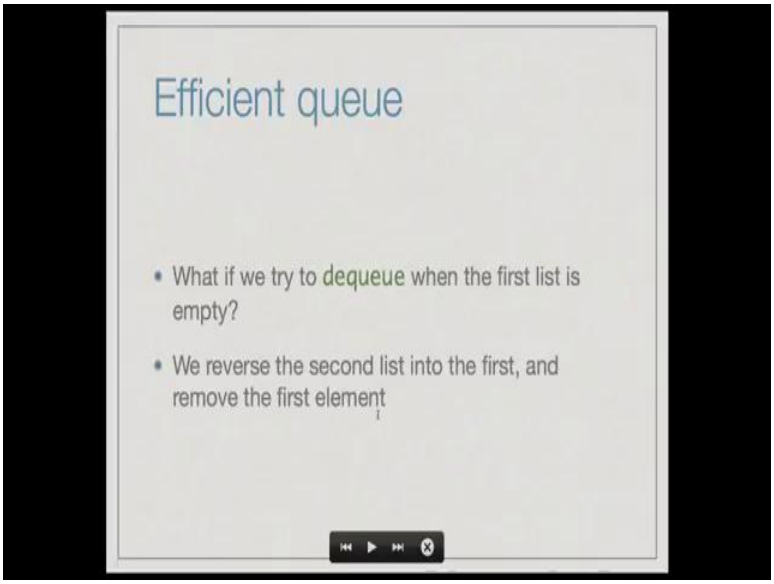
(Refer Slide Time: 12:31)



Here is a more efficient implementation of a q we use two lists and represent q 1, q 2 to q n the q consisting elements q 1, q 2 to q n this order as two lists, the first list consisting of some elements in this order q 1, q 2 till q i. And the second list consisting of the remaining elements in the reverse order q n, q n minus 1 etcetera till q i plus 1. The

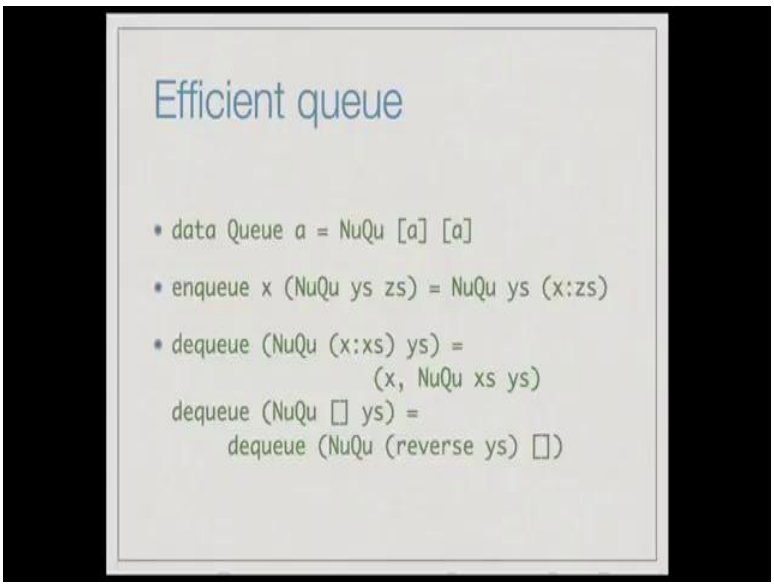
second list is the second part of the queue after q 1 to q i in reversed order no enqueue it can be made to add a element at the stack of the second list and dequeue removes an element from the start of the first list. Recall that adding an element at the start of a list is a much simpler operation then adding an element at the end of the list.

(Refer Slide Time: 13:28)



Now, that is a problem if we try to dequeue from a q, where the first list is empty, if the q itself is the empty then we cannot dequeue from it, but if the q is non empty, but the first list is empty in our representation then we need to reverse the second list in to the first and remove the first element.

(Refer Slide Time: 13:55)

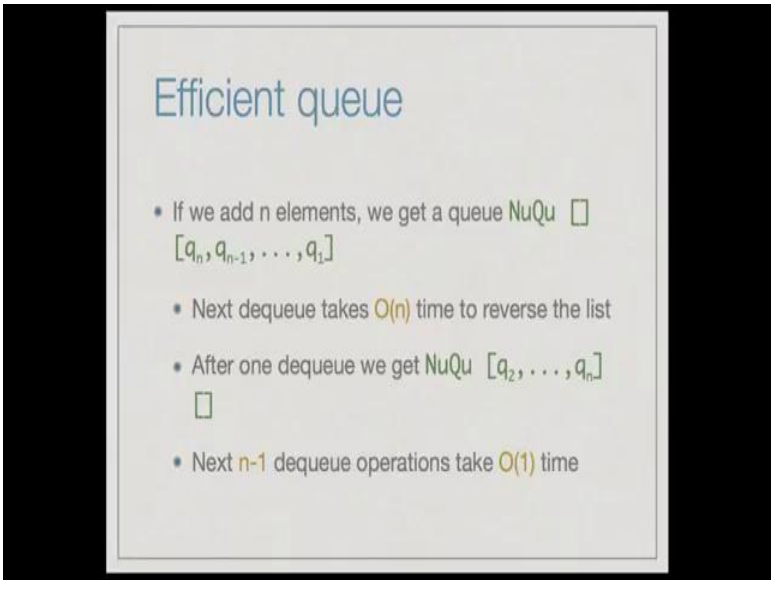


This leads us to the following implementation, data queue a equals NuQu this is the new constructor, typically the convention is that the value constructor has the same name as the type constructor or type name. But, here just distinguish this from the earlier implementation we call it NuQu, NuQu and it is the constructor which takes two lists arguments. So, data queue a equals NuQu list a list a, now the enqueue function as we described earlier adds an element to the start of the second list enqueue of x and NuQu y is an z it is nothing but, NuQu of y is the first list is retained as it is and x is added to the front of the second list x colon z s.

Dequeue NuQu of x colon x s y s this is the case, where the first list is non empty is nothing but, x comma new queue of x s and y s. We have removed the first element from the first list and retain the second list as it is. Dequeue of NuQu empty list y s this is the case where the first list is empty, what we do is reverse the second list in to the first position and take the second list to be the empty list and dequeue from here. So, dequeue of NuQu empty list why is is nothing but, dequeue of NuQu reverse y is empty list.

This is seemingly a better implementation, because every time we enqueue we add to the beginning of the second list, but there are times when you have to reverse the second list in to the first list, namely when the first list is empty, how much does this cost.

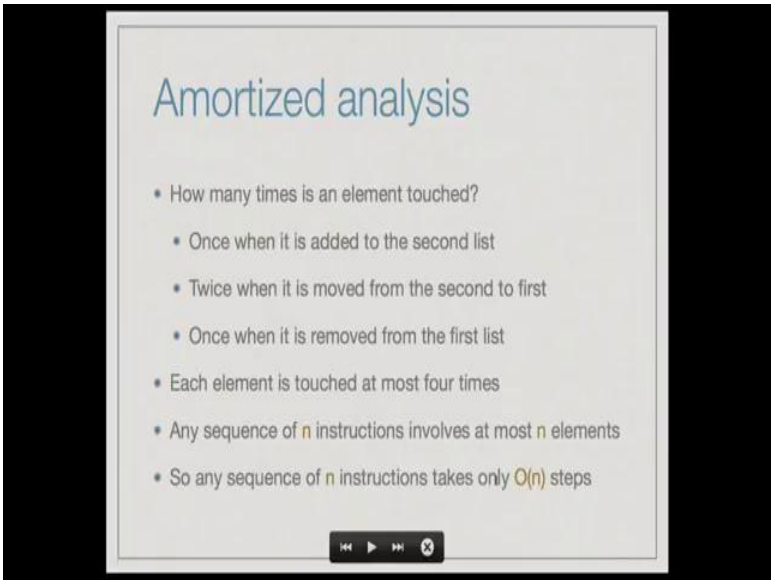
(Refer Slide Time: 15:56)



If we add n elements to the queue we get NuQu of empty list q n, q n minus 1 to q n. The next dequeue takes order n time to reverse the list queue n to queue 1 into the first list and if we dequeue once now we get NuQu of queue 2 to queue n and empty list, if the

beauty is that the next n minus 1 dequeue operations take only order one time, we paid order n item when we did this dequeue operation. But, we were repaid by hang to spend only constant time on the next n minus 1 dequeue operations, because already have to do is just remove the element from the front of the first list therefore, an average we will still the consuming order n time.

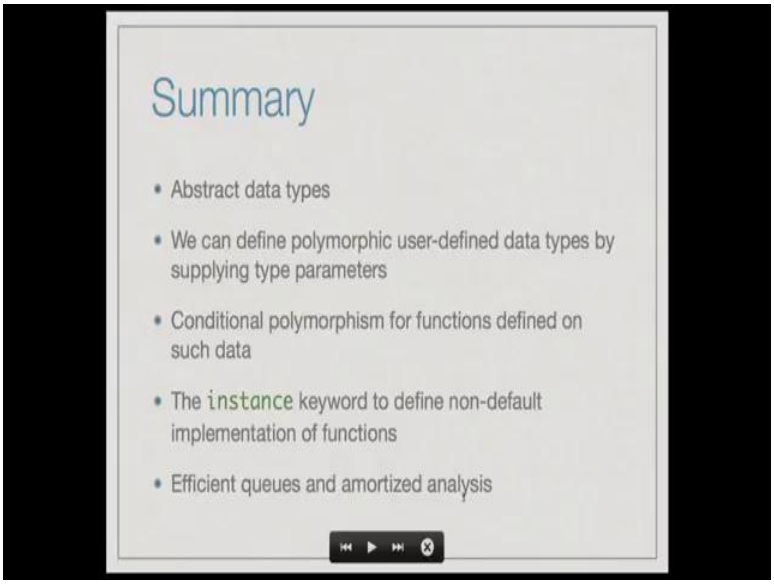
(Refer Slide Time: 16:55)



This is made presides by something called amortized analysis, here we precisely count how much time it takes to complete the sequences of operations, rather than a single operation. Even though a single dequeue may take as much as order n time, when we consider a sequence of n instructions this story is different, it is not that we take order n squared time for completing the sequence of n instructions. In fact, we only take order n steps to complete a sequence of n instructions.

And the way to show it to as follows as, look at how many times an elements touched on it is enters the queue, it is touched once when it is added to the second list as part of the enqueue and it is touched twice when it is moved from the second queue to the first queue. This is when we do a dequeue operation when the first list is empty and this element is touched once when it is removed from the first list, this is when we dequeue from the first list when this element was the head. So, therefore, each element is touched at most four times any sequence of n instructions involves at most n elements and therefore, any sequence of n instructions takes only order n steps.

(Refer Slide Time: 18:24)



To summarize, you would define abstract data types in this lecture through the example of the stack and a queue. You also define polymorphic user defined data types by showing how you can generalize this stack and queue data types to store not just integers, but any data type a, whatever by the use of supplying type parameters. The functions on these data types are polymorphic, but sometimes their conditionally polymorphic, not freely polymorphic.

For instance, for the function some stack etcetera, because the function make sense only if the type parameter satisfies certain properties, like being a numeric data types for instance. We have shown that we can use the instance key word to define non default implementations of functions like show. Finally, we looked at an example of how we can implement queues more efficiently and analyzed the efficiency of this new implementation by using the technique of amortized analysis.