# Introduction To Haskell Programming

Prof. S. P. Suresh

Chennai Mathematical Institute

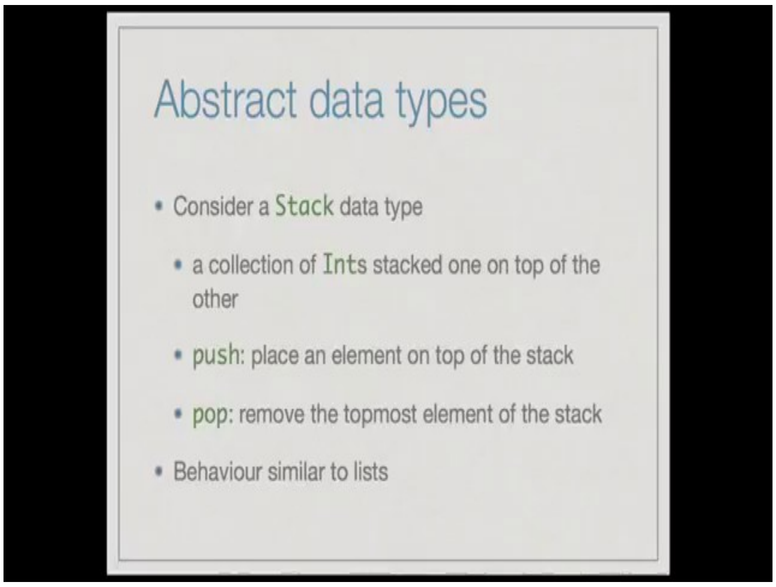
**Module # 05**

**Lecture - 02**

**Abstract Data Types**

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

(Refer Slide Time: 00:07)

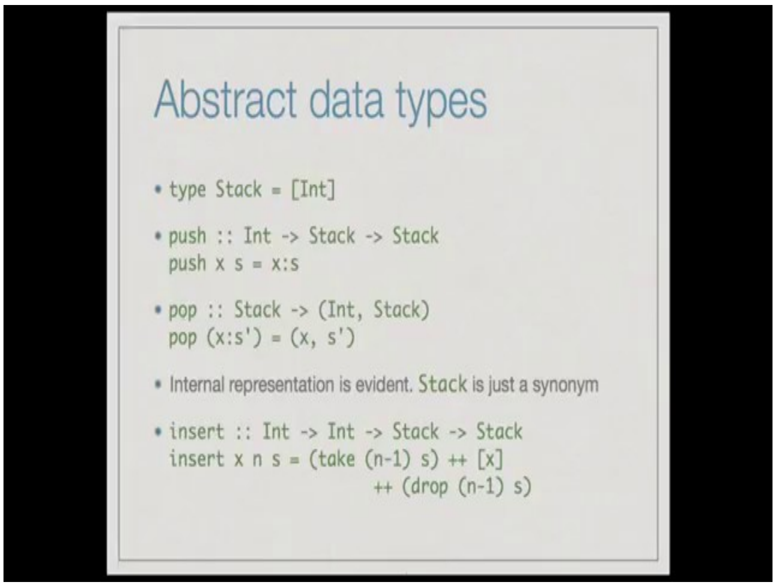


## Abstract data types

- Consider a Stack data type
  - a collection of Ints stacked one on top of the other
  - push: place an element on top of the stack
  - pop: remove the topmost element of the stack
- Behaviour similar to lists

Consider the example of a stack data type, a stack is a collection of integers, stacked 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)



## Abstract data types

- `type Stack = [Int]`

-
  ```
  push :: Int -> Stack -> Stack
  push x s = x:s
  ```

-
  ```
  pop :: Stack -> (Int, Stack)
  pop (x:s') = (x, s')
  ```
- Internal representation is evident. `Stack` is just a synonym

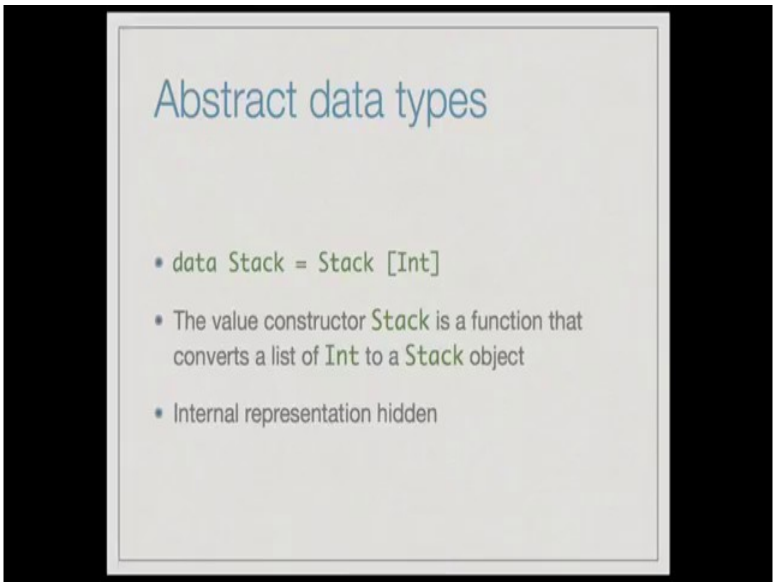
-
  ```
  insert :: Int -> Int -> Stack -> Stack
  insert x n s = (take (n-1) s) ++ [x]
                      ++ (drop (n-1) s)
  ```

With this in mind, here is a definition of stack, we define stack as a type synonym for list of integers, type Stack = [Int]. Now, the push function can be defined as follows, push is a function which takes two arguments an integer and a stack as inputs and outputs a stack. push x s = x: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 (x:s') = (x, s'). So, with this definition the internal representation of stack is very evident, stack is just a synonym for list of Ints. 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 a stack, insert x n s = ( take (n-1) s ) ++ [x] ++ ( drop (n-1) 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.

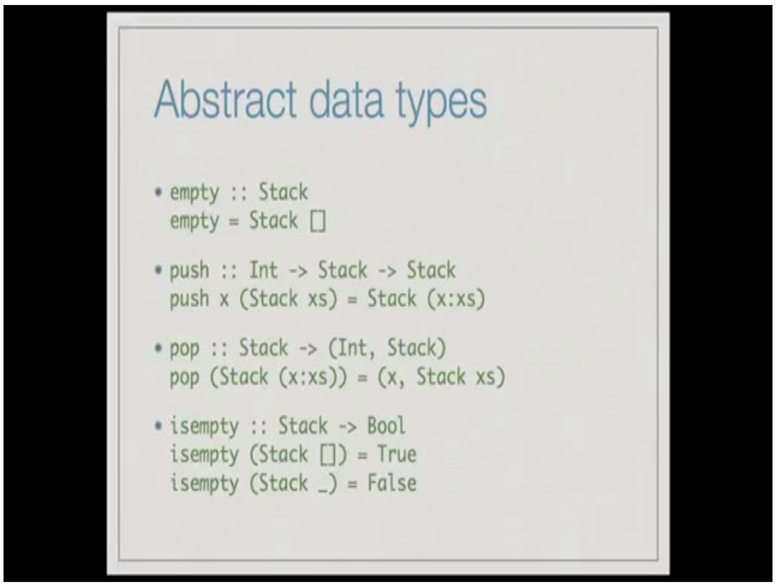
(Refer Slide Time: 02:51)


## Abstract data types

- `data Stack = Stack [Int]`
- The value constructor `Stack` is a function that converts a list of `Int` to a `Stack` object
- Internal representation hidden

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 = Stack [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 the so called 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.

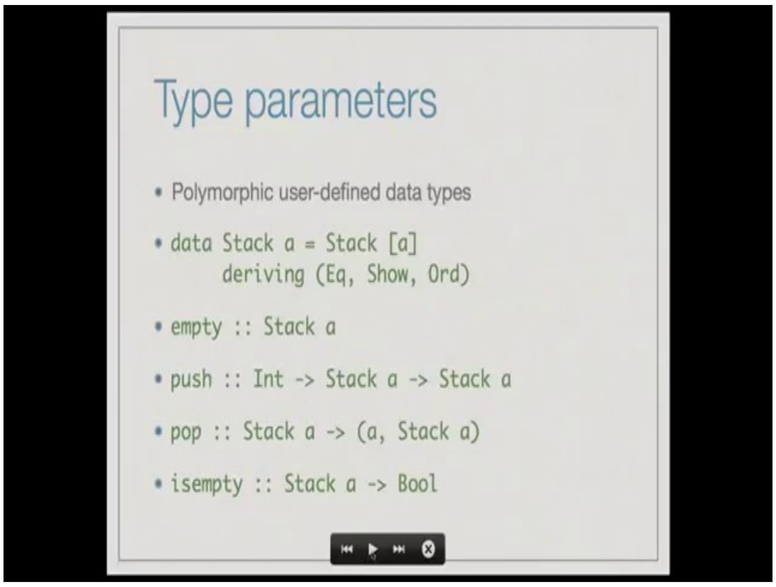
(Refer Slide Time: 03:44)


## Abstract data types

```
• empty :: Stack
  empty = Stack []

• push :: Int -> Stack -> Stack
  push x (Stack xs) = Stack (x:xs)

• pop :: Stack -> (Int, Stack)
  pop (Stack (x:xs)) = (x, Stack xs)

• isempty :: Stack -> Bool
  isempty (Stack []) = True
  isempty (Stack _) = False
```

Now, you can define the following functions on this new data type, empty which returns an empty stack and the definition is empty = Stack [], push which takes an int and a stack and returns the stack, push x (Stack xs) = Stack (x:xs). 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 an int and a stack, pop (Stack (x:xs)) = (x, Stack xs), isempty is the function that checks if the Stack is empty, it is a function from Stack -> Bool, isempty (Stack []) = True, isempty (Stack _) = False.

(Refer Slide Time: 04:40)



## Type parameters

- Polymorphic user-defined data types
- data Stack a = Stack [a]
      deriving (Eq, Show, Ord)
- empty :: Stack a
- push :: Int -> Stack a -> Stack a
- pop :: Stack a -> (a, Stack a)
- isempty :: Stack a -> Bool

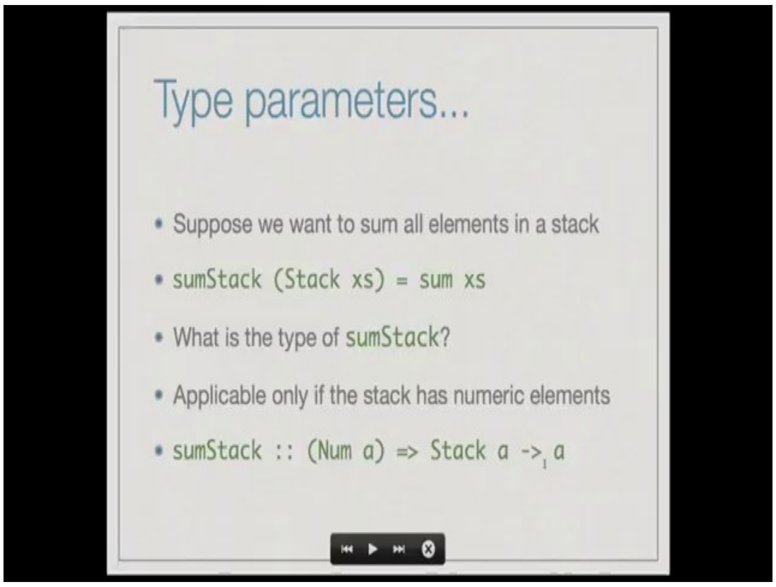
The Stack data type that we defined earlier was very specific in the sense that it stored only integer values. We might want to implement a Stack that stores data of any type whatsoever. This is achieved by using polymorphic user defined data types, which use type parameters. The definition is as follows, data Stack a = Stack [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 etc. 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 input and produces an object of type Stack a as output. In this particular case we are also deriving Eq, Show, Ord, etc 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 input and produces a Stack a as output.

Pop which is a function that takes a Stack a as input and produces the pair (a , Stack a), one value of type a and a stack which is of type Stack a as output, isempty is a 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)

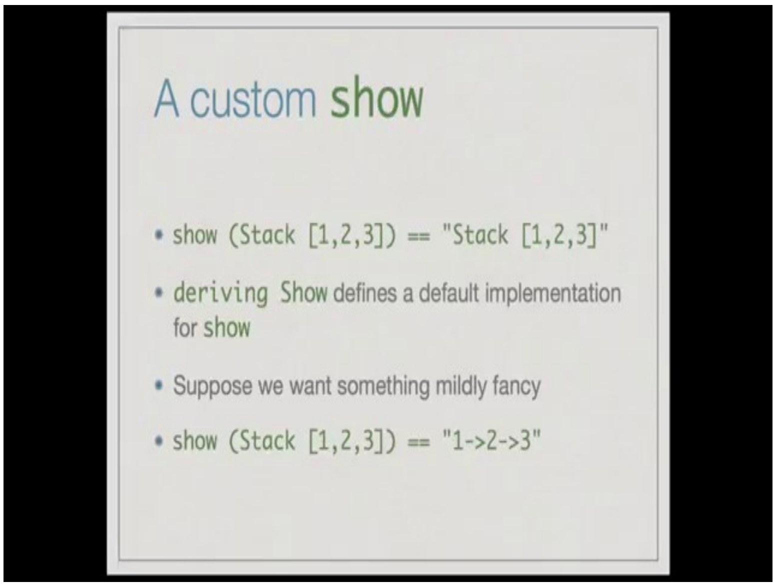


## Type parameters...

- Suppose we want to sum all elements in a stack
- sumStack (Stack xs) = sum xs
- What is the type of sumStack?
- Applicable only if the stack has numeric elements
- sumStack :: (Num a) => Stack a -> a

Sometimes functions on these polymorphic data types will not be completely polymorphic, but only conditionally polymorphic. Here is an example, suppose we want to sum all elements in a stack you would achieve it as follows, sumStack (Stack xs) = sum xs, the function sum applied to xs, 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, its type is (Num a) => Stack a -> a. Therefore, the sumStack function is applicable only if the stack has numeric elements, in other words the sumStack function has type (Num a) implies Stack a -> a.

(Refer Slide Time: 08:01)

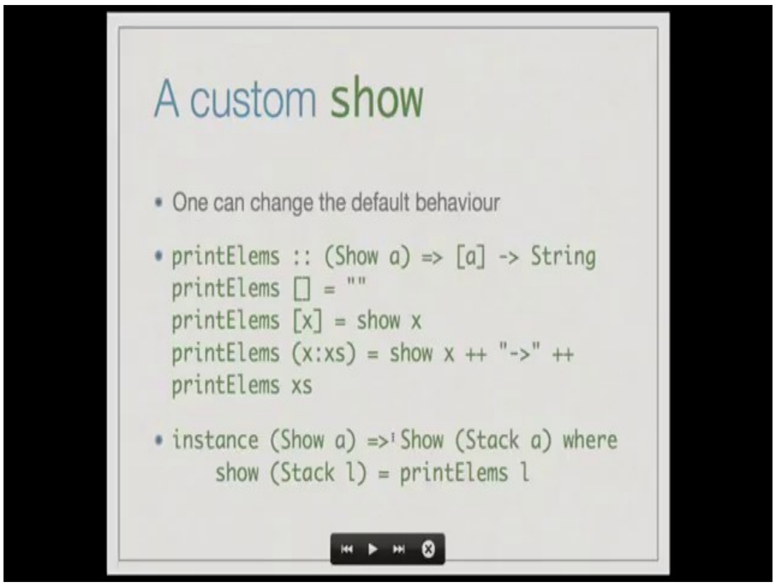


## A custom show

- `show (Stack [1,2,3]) == "Stack [1,2,3]"`
- `deriving Show` defines a default implementation for show
- Suppose we want something mildly fancy
- `show (Stack [1,2,3]) == "1->2->3"`

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 [1,2,3]) is just this string which says "Stack [1,2,3]". Suppose you want something fancier, let say we want to say show (Stack [1,2,3]) == "1 -> 2 -> 3". The string "1 -> 2 -> 3" means that you would have to define our own custom show function.

(Refer Slide Time: 08:51)



## A custom show

- One can change the default behaviour

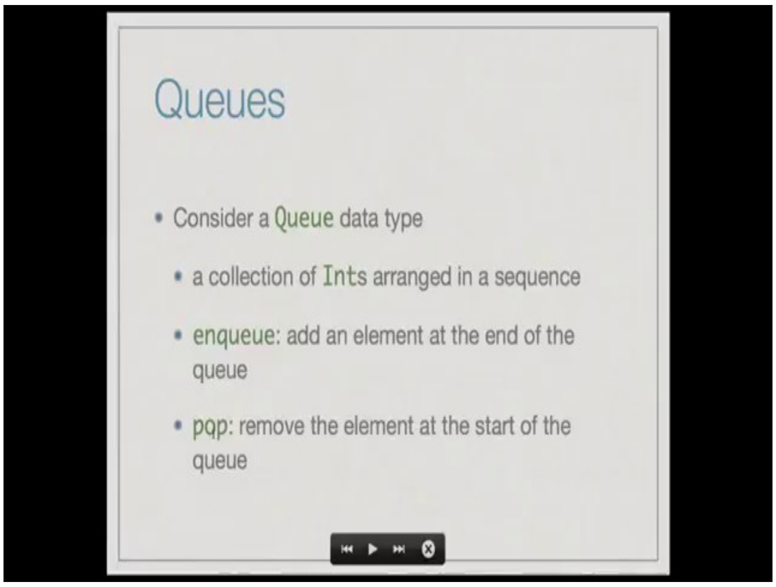
-
  ```
  printElems :: (Show a) => [a] -> String
  printElems [] = ""
  printElems [x] = show x
  printElems (x:xs) = show x ++ "->" ++
  printElems xs
  ```

-
  ```
  instance (Show a) => Show (Stack a) where
      show (Stack l) = printElems l
  ```

One can change the default behavior of show as follows, instance (Show a) => Show (Stack a) where show (Stack l) = printElems l. 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 (Stack l) = printElems l. The printElems function is given here, printElems is the function from list a to String provided a belongs to the type class show, printElems of the empty list is just the empty string, printElems of the singleton list containing x is just show of x, printElems of x:xs, is show x ++ "->" ++ printElems xs.

Notice the recursive call 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)

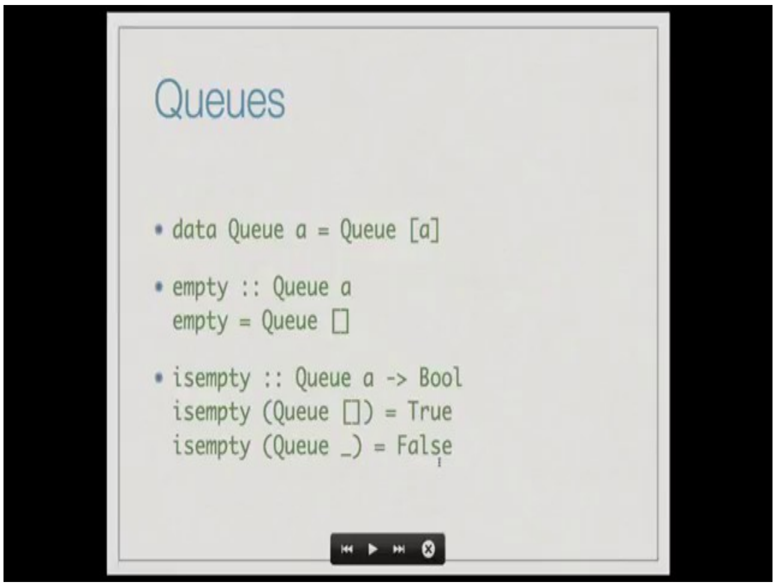


## Queues

- Consider a Queue data type
  - a collection of Ints arranged in a sequence
  - enqueue: add an element at the end of the queue
  - pqp: remove the element at the start of the queue

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 at the start of the queue.

(Refer Slide Time: 10:48)



## Queues

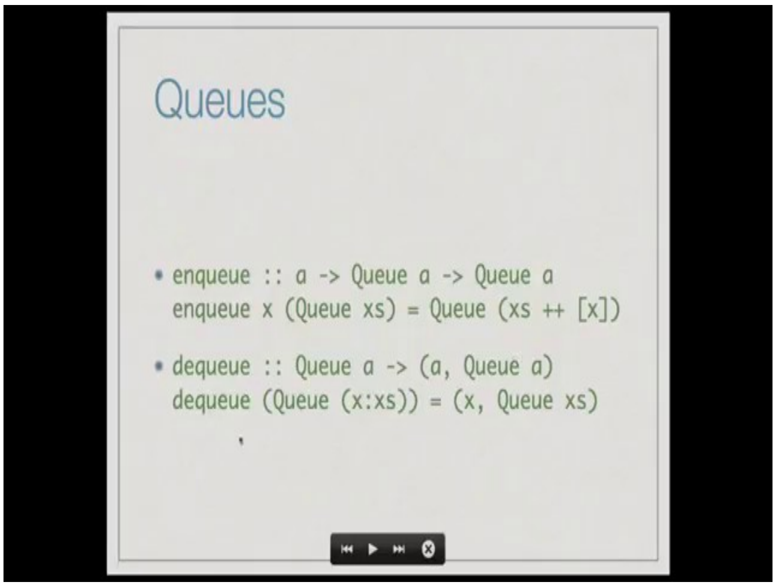
- `data Queue a = Queue [a]`

-
  ```
  empty :: Queue a
  empty = Queue []
  ```

-
  ```
  isempty :: Queue a -> Bool
  isempty (Queue []) = True
  isempty (Queue _) = False
  ```

Here is the definition, data Queue a = Queue [a], recall again that Queue is the type constructor and the Queue on the right is a value constructor, which is a function from [a] to the type Queue a. The empty queue is just given by Queue [], isempty is a function from Queue a -> Bool, isempty (Queue [] ) = True, isempty (Queue _ ) = False.

(Refer Slide Time: 11:24)



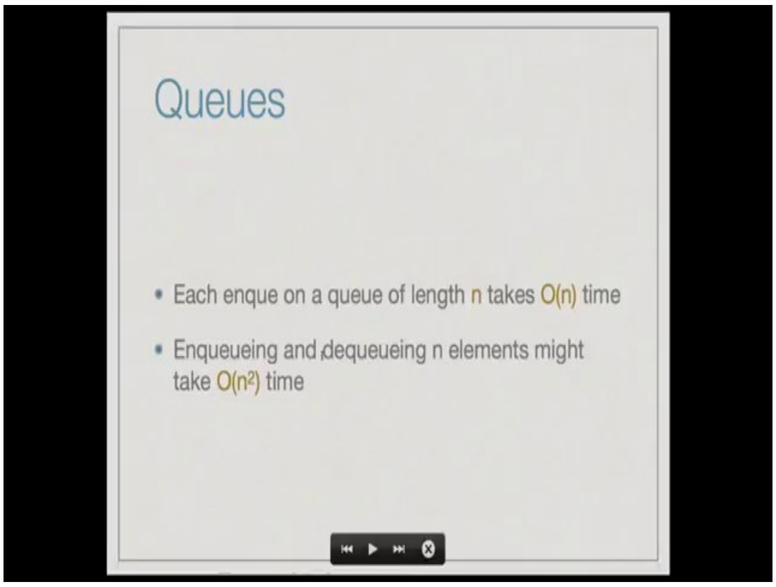
## Queues

-
  ```
  enqueue :: a -> Queue a -> Queue a
  enqueue x (Queue xs) = Queue (xs ++ [x])
  ```

-
  ```
  dequeue :: Queue a -> (a, Queue a)
  dequeue (Queue (x:xs)) = (x, Queue xs)
  ```

enqueue adds an element to the end of the list, so enqueue x (Queue xs) = Queue (xs ++ [x]), dequeue is a function that takes a Queue as input and produces an element of type a and a Queue a object as output. dequeue (Queue (x:xs)) = (x, Queue xs).

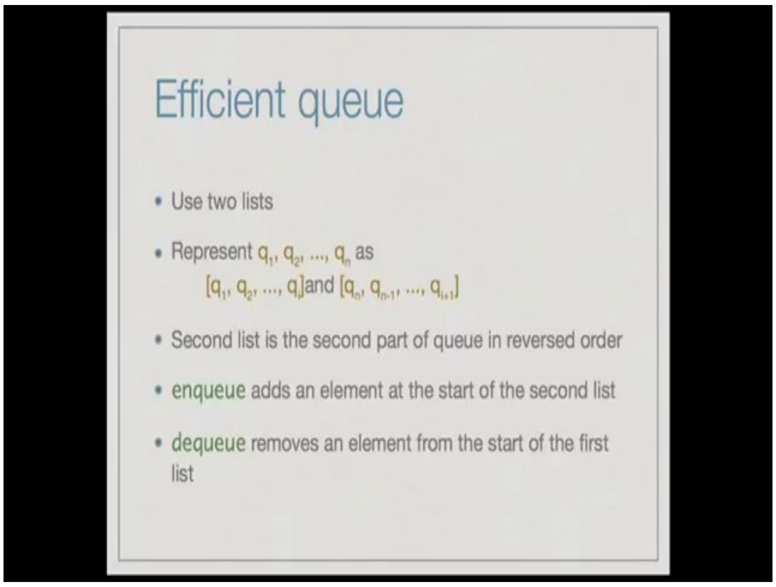
(Refer Slide Time: 12:01)


## Queues

- Each enque on a queue of length n takes O(n) time
- Enqueueing and dequeueing n elements might take $O(n^2)$ time

In this implementation each enqueue on a queue of length n takes O(n) time, because we are adding the element at the end of the list. So, enqueueing and dequeueing  n elements might take O(n2) 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.

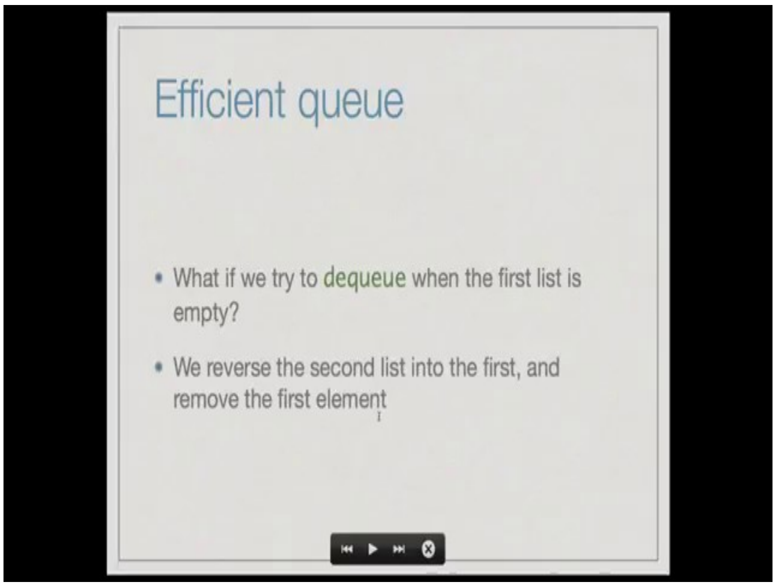
(Refer Slide Time: 12:31)


## Efficient queue

- Use two lists
- Represent $q_1, q_2, ..., q_n$ as $[q_1, q_2, ..., q_i]$ and $[q_n, q_{n-1}, ..., q_{i+1}]$
- Second list is the second part of queue in reversed order
- enqueue adds an element at the start of the second list
- dequeue removes an element from the start of the first list

Here is a more efficient implementation of a queue. we use two lists and represent the queue consisting of elements [q1, q2 ,…., qn] in this order as two lists, the first list consisting of some elements in this order [q1, q2 ,…., qi] . And the second list consisting of the remaining elements in the reverse order [qn, qn-1 ,…., qi+1] . The second list is the second part of the queue after q1 to qi in reversed order, Now enqueue can be made to add an element at the start 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 than adding an element at the end of the list.

(Refer Slide Time: 13:28)

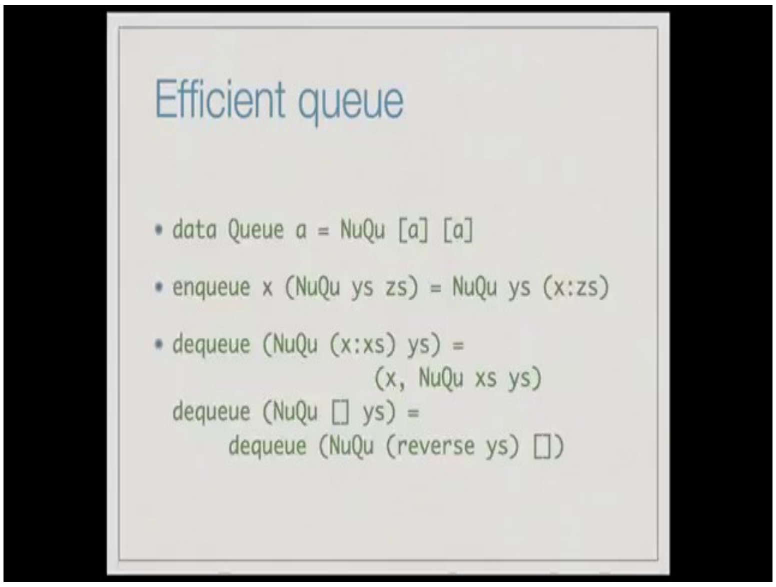


## Efficient queue

- What if we try to dequeue when the first list is empty?
- We reverse the second list into the first, and remove the first element

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

(Refer Slide Time: 13:55)



## Efficient queue

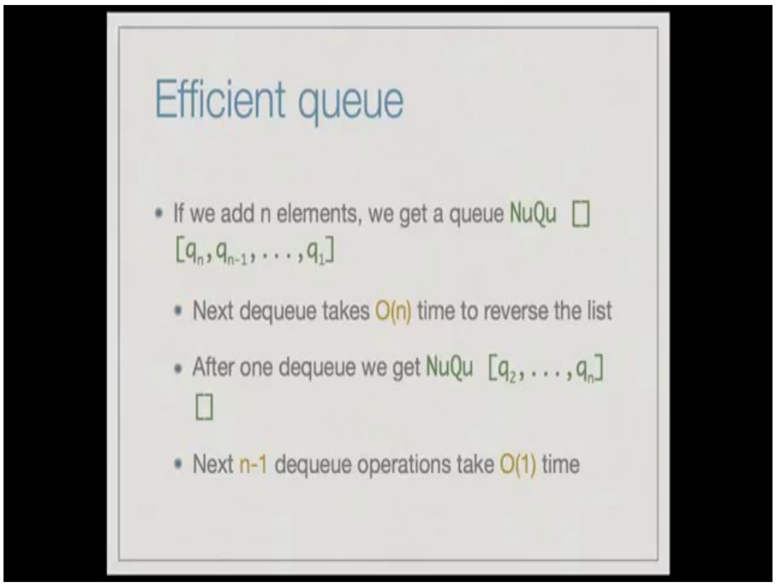
```
• data Queue a = NuQu [a] [a]

• enqueue x (NuQu ys zs) = NuQu ys (x:zs)

• dequeue (NuQu (x:xs) ys) =
                   (x, NuQu xs ys)
  dequeue (NuQu [] ys) =
       dequeue (NuQu (reverse ys) [])
```

This leads us to the following implementation, data Queue a = NuQu [a] [a]. NuQu 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 to distinguish this from the earlier implementation we call it NuQu, and it is a constructor which takes two lists as arguments. So, data Queue a = NuQu [a] [a], now the enqueue function as we described earlier adds an element to the start of the second list. enqueue x (NuQu ys zs) = NuQu ys (x:zs), the first list is retained as it is and x is added to the front of the second list x:zs.

dequeue (NuQu (x:xs) ys) this is the case, where the first list is non empty is nothing but, (x, NuQu xs ys) . We have removed the first element from the first list and retained the second list as it is. dequeue (NuQu [] ys ) 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 (NuQu [] ys ) is nothing but, dequeue (NuQu (reverse ys) []) .

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)

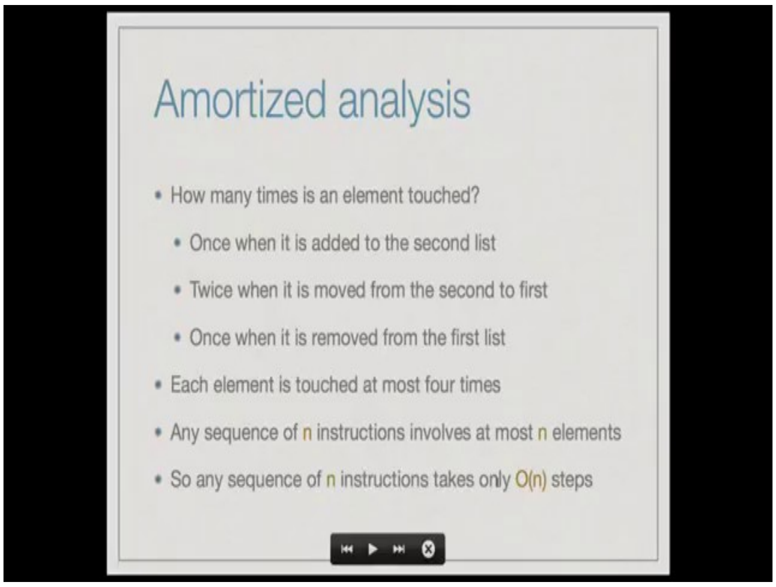


## Efficient queue

- If we add n elements, we get a queue NuQu [] $[q_n, q_{n-1}, \ldots, q_1]$
  - Next dequeue takes O(n) time to reverse the list
  - After one dequeue we get NuQu $[q_2, \ldots, q_n]$ []
  - Next n-1 dequeue operations take O(1) time

If we add n elements to the queue we get NuQu [] [qn, q n - 1, …, q1]. The next dequeue takes O(n) time to reverse the list qn to q1 into the first list and if we dequeue once now we get NuQu [q2, .., qn] [], the beauty is that the next n-1 dequeue operations take only O(1) time, we paid O(n) time when we did this dequeue operation. But, we were repaid by having to spend only constant time on the next n-1 dequeue operations, because all we have to do is just remove the element from the front of the first list therefore, on average we will still be consuming O(n) time.

(Refer Slide Time: 16:55)



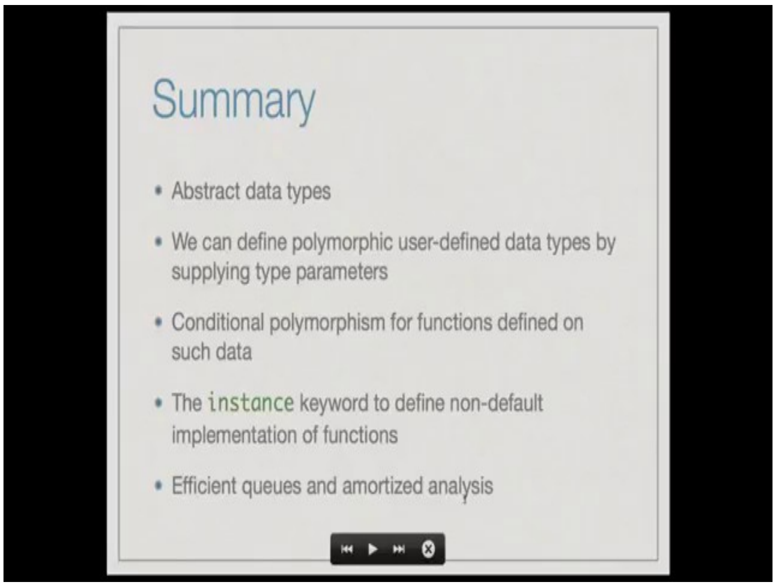
## Amortized analysis

- How many times is an element touched?
  - Once when it is added to the second list
  - Twice when it is moved from the second to first
  - Once when it is removed from the first list
- Each element is touched at most four times
- Any sequence of n instructions involves at most n elements
- So any sequence of n instructions takes only O(n) steps

This is made precise by something called amortized analysis, here we precisely count how much time it takes to complete a sequence of operations, rather than a single operation. Even though a single dequeue may take as much as O(n) time, when we consider a sequence of n instructions this story is different, it is not that we take O(n2) time for completing a 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 is as follows, look at how many times an element has touched once it 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 at 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 O(n)  steps.

(Refer Slide Time: 18:24)



## Summary

- Abstract data types
- We can define polymorphic user-defined data types by supplying type parameters
- Conditional polymorphism for functions defined on such data
- The `instance` keyword to define non-default implementation of functions
- Efficient queues and amortized analysis

To summarize, we defined abstract data types in this lecture through the example of the stack and a queue. We have also defined 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, whatsoever by the use of supplying type parameters. The functions on these data types are polymorphic, but sometimes they are conditionally polymorphic, not freely polymorphic.

For instance, for the function sumStack etcetera, because the function makes sense only if the type parameter satisfies certain properties, like being a numeric data type 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.