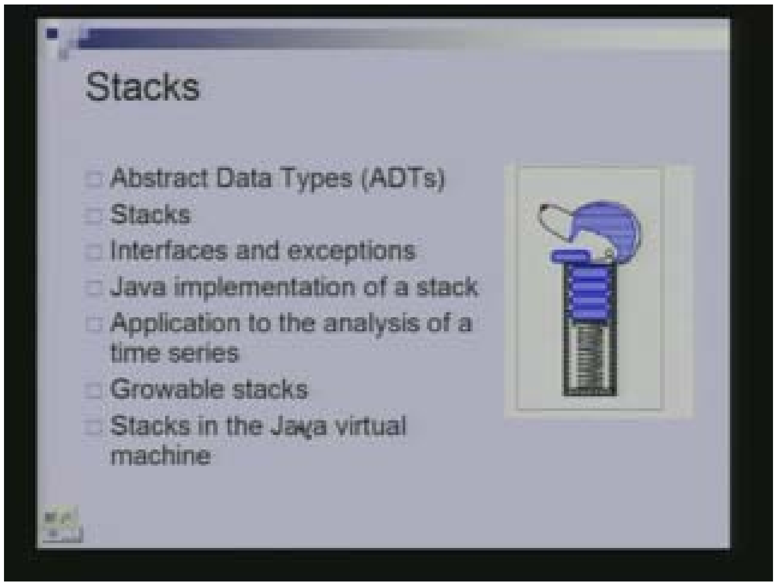
Data Structures and Algorithms
Dr. Naveen Garg
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 2
Stacks

Let us see about stacks. We will mainly see about stacks, besides we will talk about abstract data types, interfaces, exceptions, how stacks are implemented in java and application to the analysis of time series. We will also talk about growable stacks, which do a little bit of amortized analysis and then we will talk about stacks in java virtual machine.

(Refer Slide Time: 1:16)

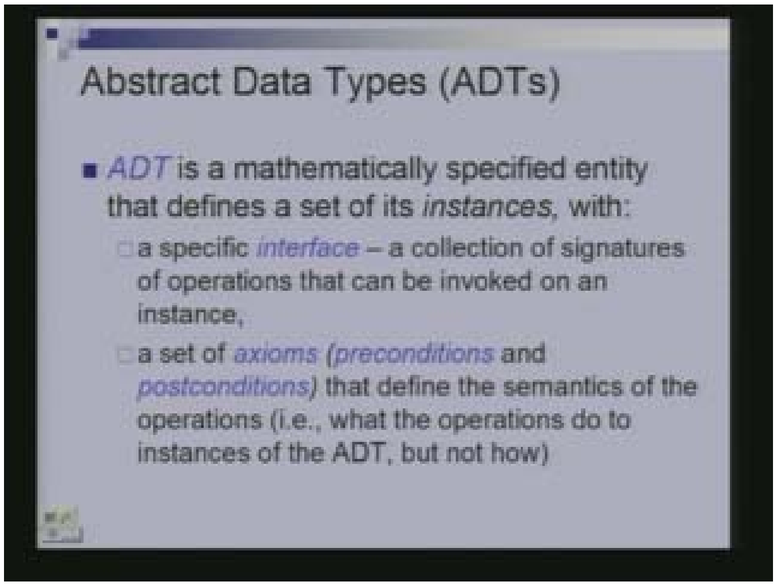


Stacks

- Abstract Data Types (ADTs)
- Stacks
- Interfaces and exceptions
- Java implementation of a stack
- Application to the analysis of a time series
- Growable stacks
- Stacks in the Java virtual machine

What is an abstract data type? It is basically a specification of the instances and the set of axioms that define the semantics of the operations on those instances. What does it all mean? You know the data types like integer, real numbers and so on. You can understand the notion of addition and that is the same way as you add 2 integers in mathematics.

(Refer Slide Time: 1:51)



Abstract Data Types (ADTs)

- *ADT* is a mathematically specified entity that defines a set of its *instances*, with:
  - a specific *interface* – a collection of signatures of operations that can be invoked on an instance,
  - a set of *axioms* (*preconditions* and *postconditions*) that define the semantics of the operations (i.e., what the operations do to instances of the ADT, but not how)

Similarly we will define data types and certain operations on those data types. Those operations would be defined through an interface which basically gives us the signature of the operation that is the parameters that operation requires and so on. We will also specify the results of those operations through a set of axioms.

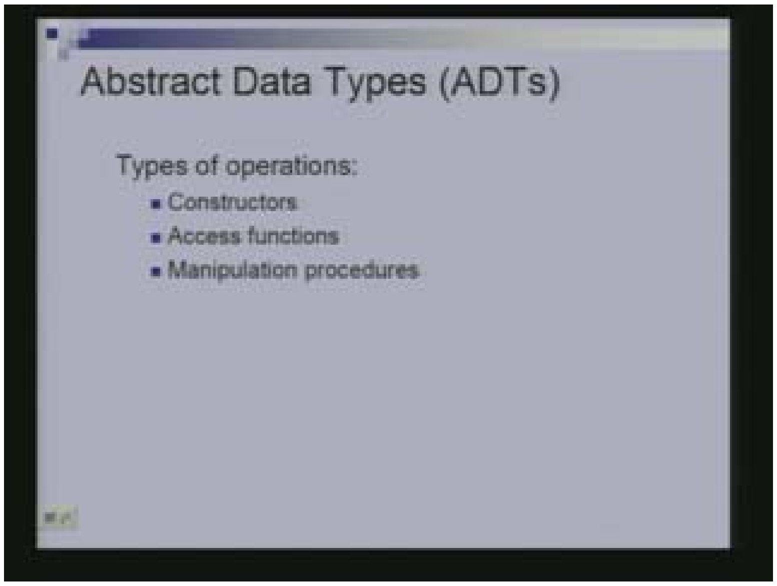
Just as in the case of integers, you know the sum of 2 integers as defined in mathematics. For example if you add a variable of type A and another variable of type B. If you sum them up, then the answer will be of type variable as you would know it from your mathematics class. We will be clear if we see an example.

The operations that you have been talking about are essentially of three kinds. One would be just a constructor operation which is as same as the constructor method in java.

Using this method you can create an instance of that particular data type. When you are talking about sophisticated data types, this method has to do a lot of work Access functions are the functions which let us to access elements of the data type and manipulation procedure would let us to manipulate or modify the data type.

Why are we talking about data types? Data types help us to identify the requirements for the building blocks of our algorithmic procedure. It provides a language which will help us to talk at a higher level of abstraction. As just as we are talking in terms of adding up of integers or in terms of stacks or queues or any of the advanced data type.

(Refer Slide time: 3:17)

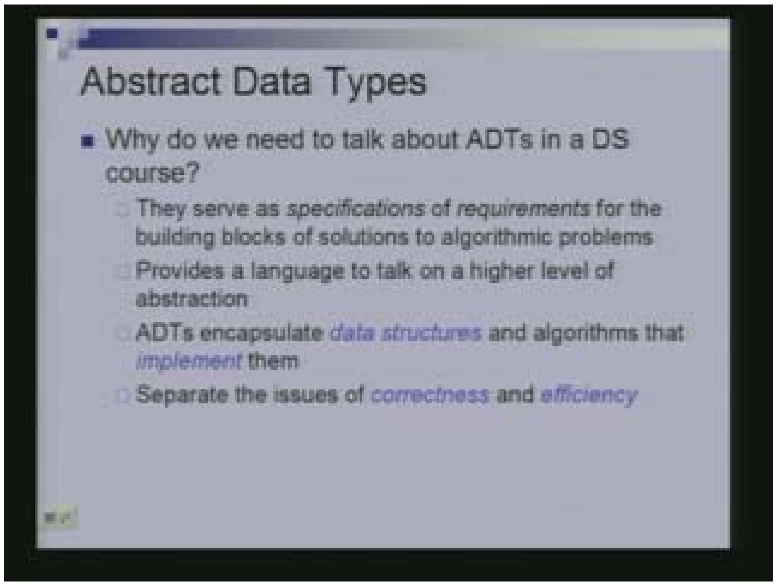


Abstract Data Types (ADTs)

Types of operations:

- Constructors
- Access functions
- Manipulation procedures

(Refer Slide Time: 3:57)

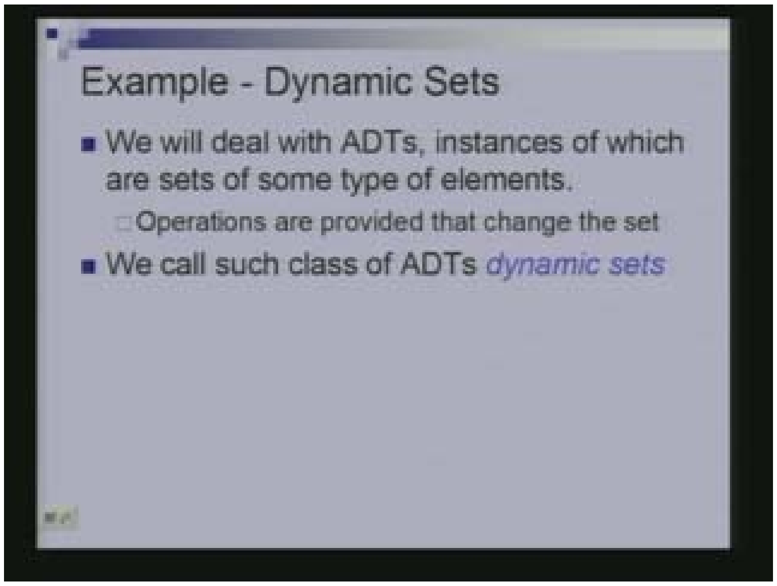


Abstract Data Types

- Why do we need to talk about ADTs in a DS course?
  - They serve as *specifications of requirements* for the building blocks of solutions to algorithmic problems
  - Provides a language to talk on a higher level of abstraction
  - ADTs encapsulate *data structures* and algorithms that *implement* them
  - Separate the issues of *correctness* and *efficiency*

They encapsulate the data structure like how the data is organized and the algorithms that work on that data structures. Also they help us to separate the issues of correctness and efficiency. We will see more of this as we see the example of data types. Let me start by giving a simple example of the data type and that is a dynamic set. A set is defined as a collection of objects. Suppose we also had operations, which would let us modify that collection of objects, which means add or remove an object of that collection. Such a set we would call it as a dynamic set.

(Refer Slide Time: 4:47)



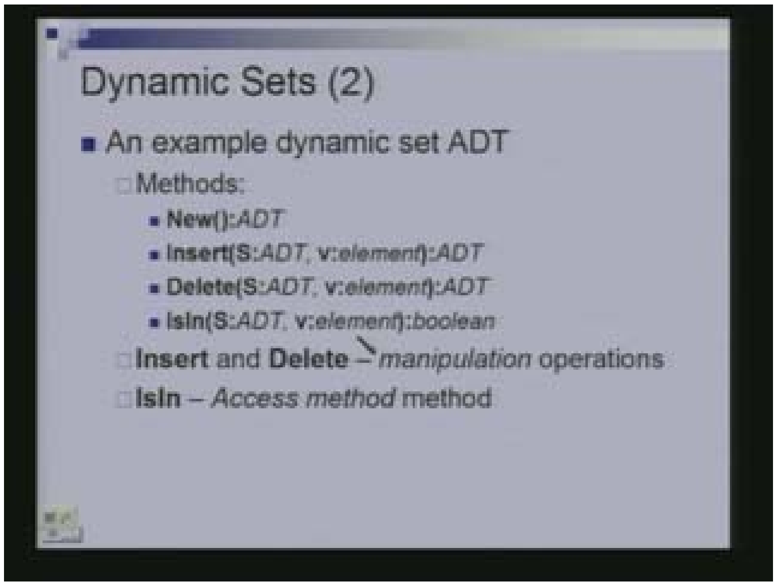
Example - Dynamic Sets

- We will deal with ADTs, instances of which are sets of some type of elements.
  - Operations are provided that change the set
- We call such class of ADTs *dynamic sets*

We call it as dynamic, because we are changing the set which is the collection of objects. We will create data types for such dynamic sets. What are the kinds of methods that you have in a dynamic set? You would have a method to create a dynamic set, which would be a method new. There would be a method insert, to insert an element in to a dynamic set. S is the dynamic set and this method has two parameters let us say, the set s and the element. The result is an instance of the set itself which gives a new set, another set and also includes the element v in it. Similarly the delete method removes the element v from the set S. These are the two methods for updating the set.

New method is for creating or constructing the set and IsIn is one of the access methods. All of it is telling us whether the element is in the set or not. The return value is of type Boolean. If v is in the set then it is true otherwise false. Axioms are the one which define how the operations should behave. We can write axioms in the following form. When I create a new set and if the set is empty then the answer should be always false, no matter what v is. If I have a dynamic set S and I insert an element v in it. Then the resulting set which has v in it should be true.

(Refer Slide Time: 5:37)



Dynamic Sets (2)

- An example dynamic set ADT
  - Methods:
    - **New()**:*ADT*
    - **Insert(S:***ADT*, **v:***element***)**:*ADT*
    - **Delete(S:***ADT*, **v:***element***)**:*ADT*
    - **IsIn(S:***ADT*, **v:***element***)**:*boolean*
  - **Insert** and **Delete** – *manipulation* operations
  - **IsIn** – *Access method* method

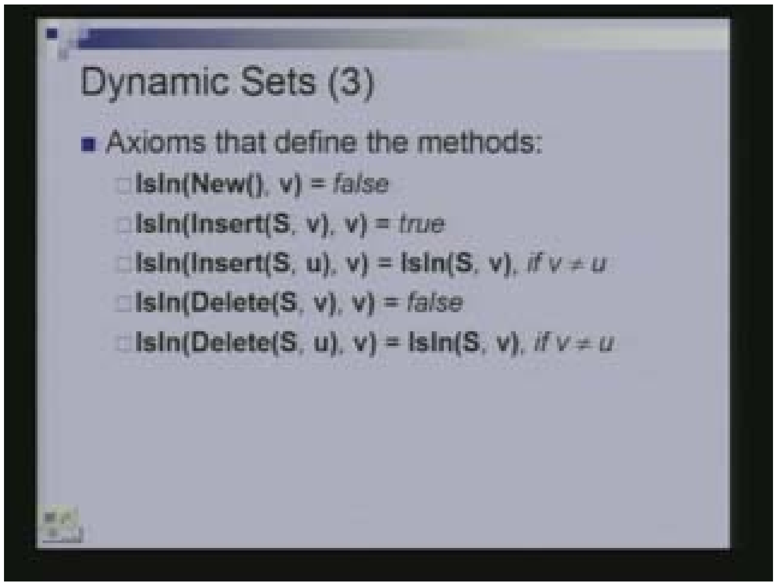
If I have a set S and when I insert u in it, then the resulting set has u in it. Then if I where to ask whether v is in the resulting set, I will know that only if v was in the previous set S.

Thus the answer to this operation IsIn (Delete(S, u), v), should be the same as the answer to this operation IsIn(S, v), provided v is different from u. IsIn (Delete(S, u), v) = IsIn(S, v), if $v \neq u$ Suppose I have a set S and I delete v from it. If I ask whether v is in the resulting set, then the answer should be false.

These are some basic axioms that define the nature of these operations and also the functionality of these operations. Still we did not specify how to do these operations or we did not talk about an algorithm or any procedure.

At the least we have talked about the code for implementing the dynamic set. When you are talking about abstract data types, we are interested in more of the specification. That is what the instances would be like and what are the operations permitted on those instances, and the axioms that govern those operations.

(Refer Slide Time: 6:46)

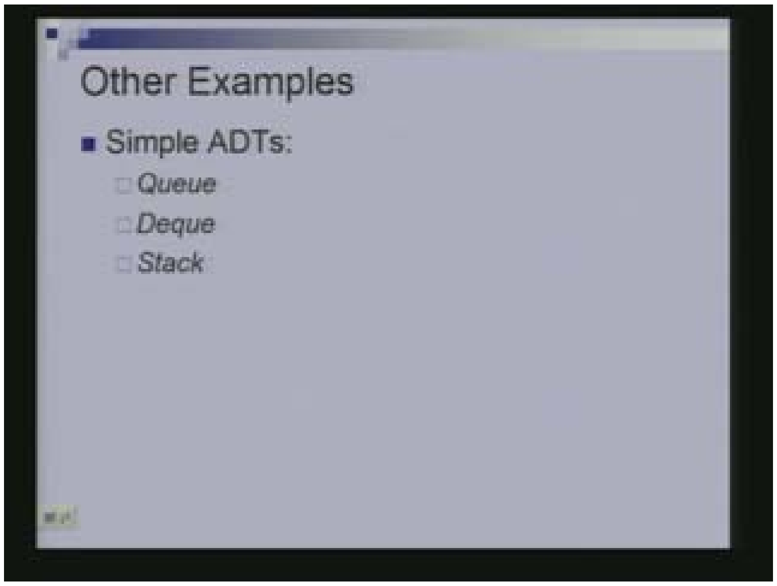


Dynamic Sets (3)

- Axioms that define the methods:
  - **IsIn(New(), v)** = *false*
  - **IsIn(Insert(S, v), v)** = *true*
  - **IsIn(Insert(S, u), v) = IsIn(S, v)**, *if* $v \neq u$
  - **IsIn(Delete(S, v), v)** = *false*
  - **IsIn(Delete(S, u), v) = IsIn(S, v)**, *if* $v \neq u$

Some simple abstract data type that you may be familiar with is queue, but we will be doing it later. Let us see about stacks.

(Refer Slide Time: 8:52)

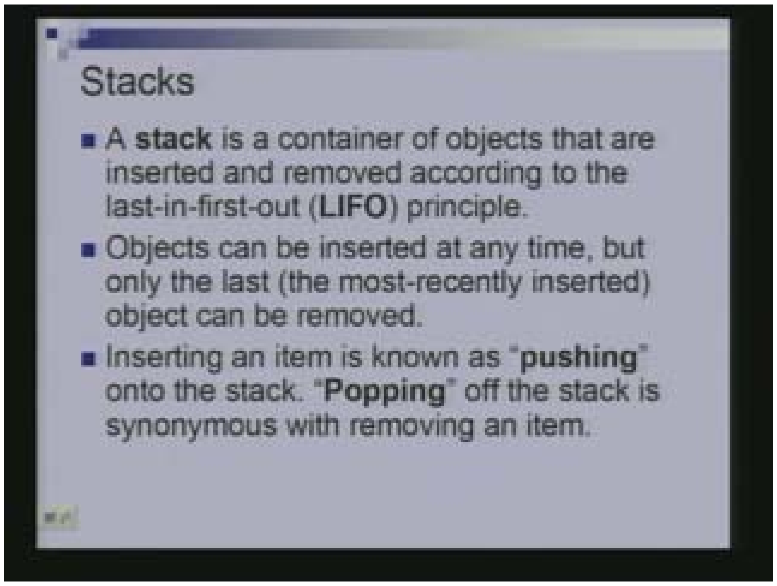


Other Examples

- Simple ADTs:
  - *Queue*
  - *Deque*
  - *Stack*

What is the stack? It is the collection of elements but this collection follows the last-in-first-out principle. What does it mean? It means that the element which is inserted last would be removed first. If I insert an element and then I remove an element from this collection. Then the element that would be removed was the one which was inserted at the last. The operation of inserting an element is called pushing onto the stack and the operation of removing an element is called popping off the stack.

(Refer Slide Time: 9:10)



Stacks

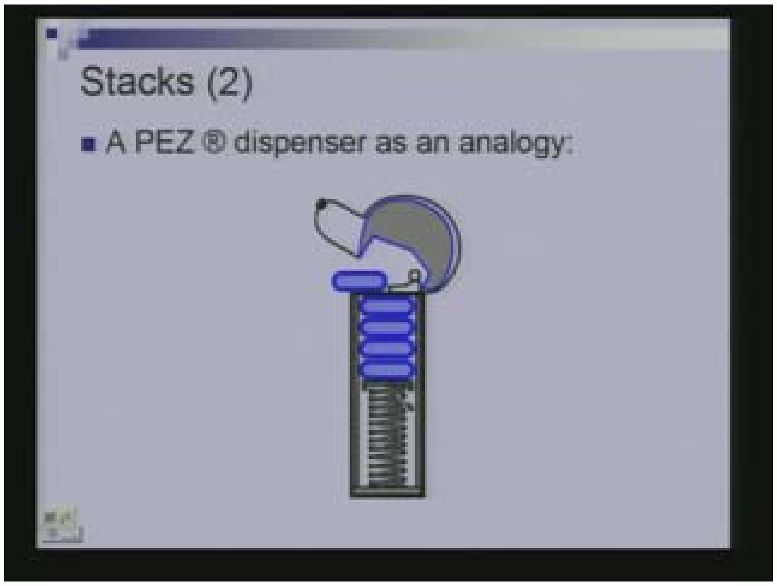
- A **stack** is a container of objects that are inserted and removed according to the last-in-first-out (**LIFO**) principle.
- Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
- Inserting an item is known as "**pushing**" onto the stack. "**Popping**" off the stack is synonymous with removing an item.

Some of you might have seen this kind of toys. It has a collection of elements for instance may be stack of trays in your mess. What you do is when you put a tray, you put it on the top and when you remove it you would always remove the one which is at the top.

When you remove or pop of an element, it is always the one which you inserted at the last. We are going to define the abstract data type that is supported by four methods which are the key methods.

The "new" is a method to create a stack. In the push method when I specify an element o it adds this element to the abstract data type. It inserts an object o on to the top of the stack.

(Refer Slide Time: 9:52)

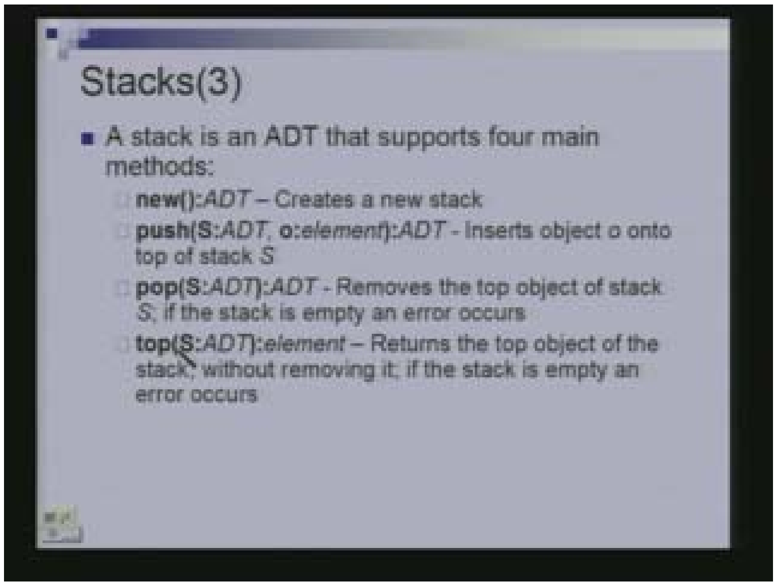


Stacks (2)

- A PEZ ® dispenser as an analogy:

Pop takes stack as the parameter and it does not take any parameter other than abstract data type. When I say pop the stack, it just removes the top element from the stack. If the stack is empty, they should flag an error stating that the stack is empty. The top operation returns the top element, it does not remove it and that is how it differs from the pop. Pop operation removes that element but the top tell us only about the top element. Again if the stack is empty then top does not make any sense, it should flag an error.

(Refer Slide Time: 10:22)

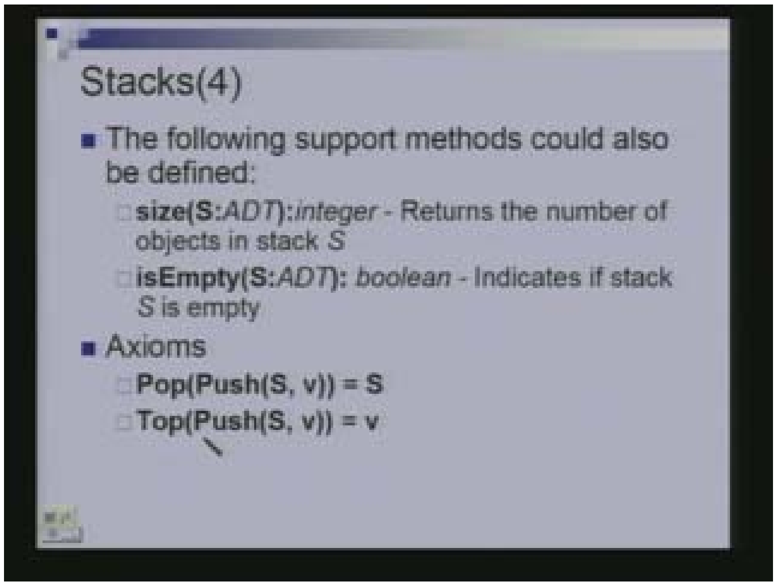


Stacks(3)

- A stack is an ADT that supports four main methods:
  - **new():***ADT* – Creates a new stack
  - **push(S:***ADT*, **o:***element***):***ADT* - Inserts object *o* onto top of stack *S*
  - **pop(S:***ADT***):***ADT* - Removes the top object of stack *S*; if the stack is empty an error occurs
  - **top(S:***ADT***):***element* – Returns the top object of the stack, without removing it; if the stack is empty an error occurs

We can also have some support methods which will help us do these operations. Size is one such method. Size tells us about how many elements are there in the stack and isEmpty tell us whether the stack is empty or not. The 6 methods that we saw are push, pop, new, top, size and isEmpty. These are all the methods and hope you all understood about what these methods are doing.

(Refer Slide Time: 11:34)



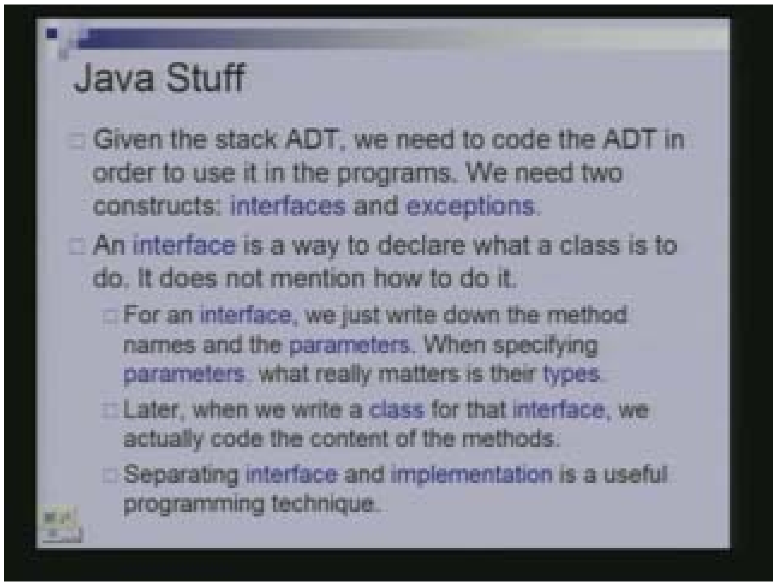
Stacks(4)

- The following support methods could also be defined:
  - **size(S:***ADT***):***integer* - Returns the number of objects in stack *S*
  - **isEmpty(S:***ADT***):** *boolean* - Indicates if stack *S* is empty
- Axioms
  - **Pop(Push(S, v)) = S**
  - **Top(Push(S, v)) = v**

Axiom governs the behavior of these methods. If S is the stack, when I push an element on to S and then when I pop it, I should get back S. While doing a top operation, when I push an element on to a stack and then when I do a top operation I should get v, because v would be the top element of the stack. So far we have defined about the stack abstract data type, the methods and 2 axioms.

Axioms may not be complete but this is what the axioms would look like. How do we translate abstract data type into code? We need 2 constructs for that and they are the interfaces and exceptions. What is an interface? An interface is a way to declare about what a class has to do and what are the various methods associated with the class. It does not tell us about how those methods are done. That would be a part of the implementation of that interface or a class.

(Refer Slide Time: 12:46)

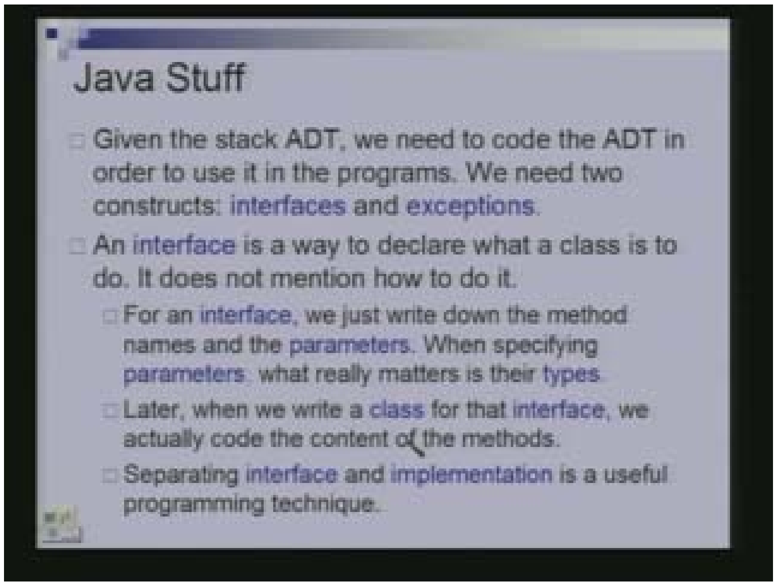


Java Stuff

- Given the stack ADT, we need to code the ADT in order to use it in the programs. We need two constructs: interfaces and exceptions.
- An interface is a way to declare what a class is to do. It does not mention how to do it.
  - For an interface, we just write down the method names and the parameters. When specifying parameters, what really matters is their types.
  - Later, when we write a class for that interface, we actually code the content of the methods.
  - Separating interface and implementation is a useful programming technique.

For an interface we just right down the various names of the methods and the parameters it is going to take. In fact we do not even specify the names of parameter, we just have to specify the types of the parameter. When we write a class for an interface, we will actually provide the code for those various methods.

(Refer Slide Time: 13:16)

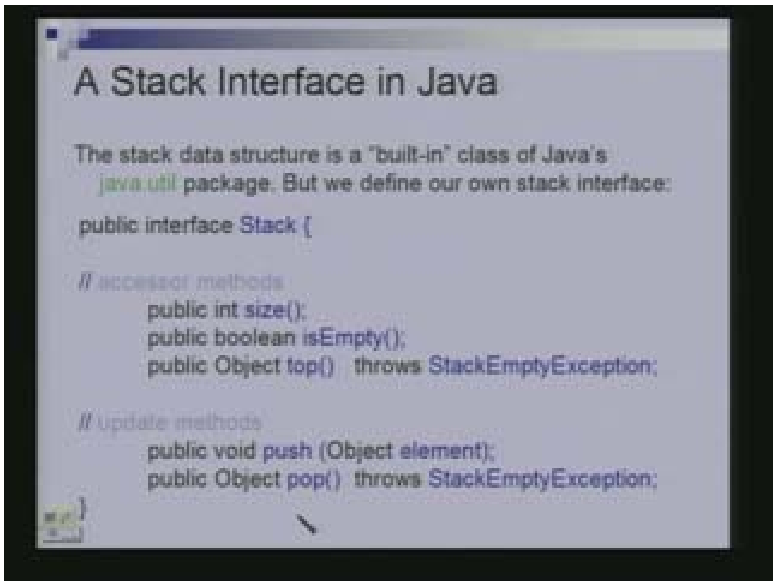


Java Stuff

- Given the stack ADT, we need to code the ADT in order to use it in the programs. We need two constructs: interfaces and exceptions.
- An interface is a way to declare what a class is to do. It does not mention how to do it.
  - For an interface, we just write down the method names and the parameters. When specifying parameters, what really matters is their types.
  - Later, when we write a class for that interface, we actually code the content of the methods.
  - Separating interface and implementation is a useful programming technique.

I might specify an interface for a stack and I am going to ask you to write the classes for that interface. Different people will write different classes to implement the interface in a completely different ways. I can still use your classes or any implementation of the interface, in a program that I have written, provided that must meet the interface specification which I have given to you. All I need to know is that the implementation meets the specification so that I can use that in the coding of my own program. It helps us to separate the implementation from the specification and that is why it is a very useful programming technique.

(Refer Slide Time: 15:15)



A Stack Interface in Java

The stack data structure is a "built-in" class of Java's java.util package. But we define our own stack interface:

```
public interface Stack {

// accessor methods
        public int size();
        public boolean isEmpty();
        public Object top()   throws StackEmptyException;

// update methods
        public void push (Object element);
        public Object pop()  throws StackEmptyException;
}
```

Let us see about how a stack implementation looks like in java. Java has a built-in stack data structure but nevertheless we will define a stack interface. We just define the various methods that are going to be a part of this interface.

There is one method called size, in which I need to specify the types of the parameters and the return type of the method. I have not specified how these methods are implemented. This is just an interface.

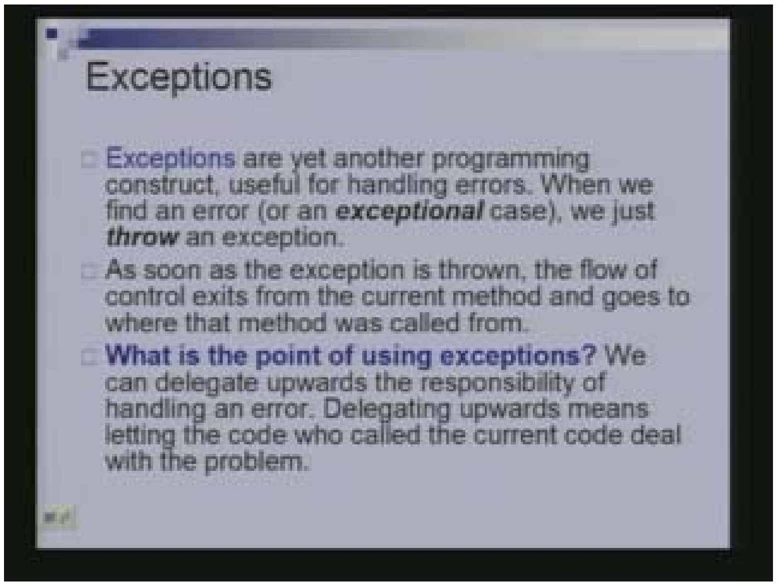
In an interface we need to know the types of the parameter. When I am pushing, it takes a parameter of type object. Object is the generic type in java and all objects are derived from this type.

The method isEmpty returns boolean. It just tells us whether the stack is empty or not. The top gives you the top element in the stack and it returns an object. It throws StackEmptyException, if this stack is empty then top () method should somehow signal that the stack is empty. We are going to do that using the notion of exceptions.

Void means it does not return any object or any value. It does not return a stack but it is a method which is executed on this stack and it modifies the stack. Thus stack cannot be considered as a parameter. What is an exception?

Exceptions are the mechanisms to handle errors. When we have an error or when we reach some exceptional condition or an exceptional case in the execution of program, we throw an exception. The term used in java is throw.

(Refer Slide Time: 18:07)



Exceptions

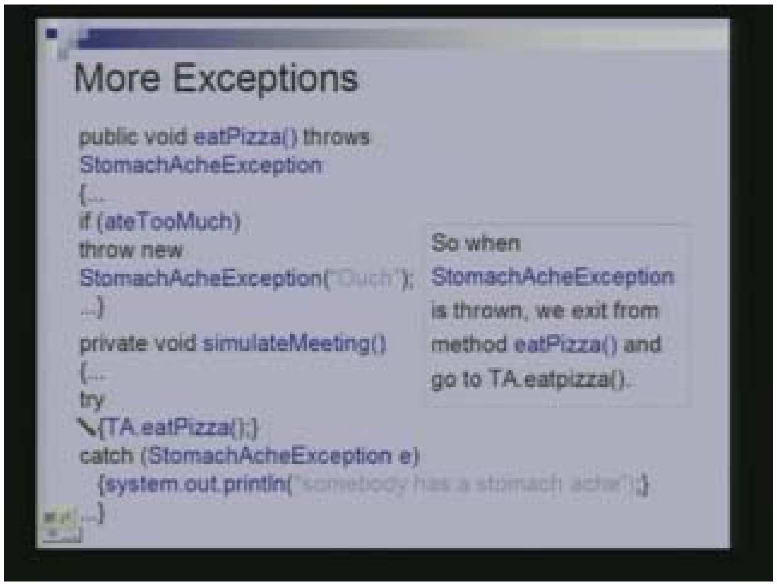
- Exceptions are yet another programming construct, useful for handling errors. When we find an error (or an ***exceptional*** case), we just ***throw*** an exception.
- As soon as the exception is thrown, the flow of control exits from the current method and goes to where that method was called from.
- **What is the point of using exceptions?** We can delegate upwards the responsibility of handling an error. Delegating upwards means letting the code who called the current code deal with the problem.

As soon as an exception is thrown, the flow of control moves from the current method to the point where the method was called. The idea essentially is that, when an exception occurs you delegate the responsibility of handling that exceptional case, to the procedure which called that particular method.

You will be clear, if you see an example. I have two methods, one is an eat pizza method which throws a stomachache exception, also there is some dotted code. If you eat too much of pizza, then there is a problem and you throw StomachAcheException.

The procedure public void eatPizza () throws was called in the method eatpizza (), which is inside the stimulate meeting procedure. When this StomachAcheException is thrown, the flow of control will come to TA.eatPizza (). Thus when this StomachAcheException is thrown, we will exist this method eatpizza () and go to TA.eatpizza ().

(Refer slide time 19.07)



More Exceptions

```
public void eatPizza() throws
StomachAcheException
{...
if (ateTooMuch)
throw new
StomachAcheException("Ouch");
...}
private void simulateMeeting()
{...
try
{TA.eatPizza();}
catch (StomachAcheException e)
  {system.out.println("somebody has a stomach ache");}
...}
```

So when StomachAcheException is thrown, we exit from method eatPizza() and go to TA.eatpizza().

In the coding after {…} there are bunches of other statements that would not be executed. The flow of control would interrupt the dotted point and would reach TA.eatpizza (). There is also a notion of try and catch blocks.

When the exception is thrown what happens to the variable that we have modified?
It depends upon the procedure call, think as if we are returning from this procedure StomachAcheException or a method. If those are local variables then you do not want to see them. If they are global variables and if it is modified in the if-loop, then those modifications are carried over to the TA.eatpizza () method.

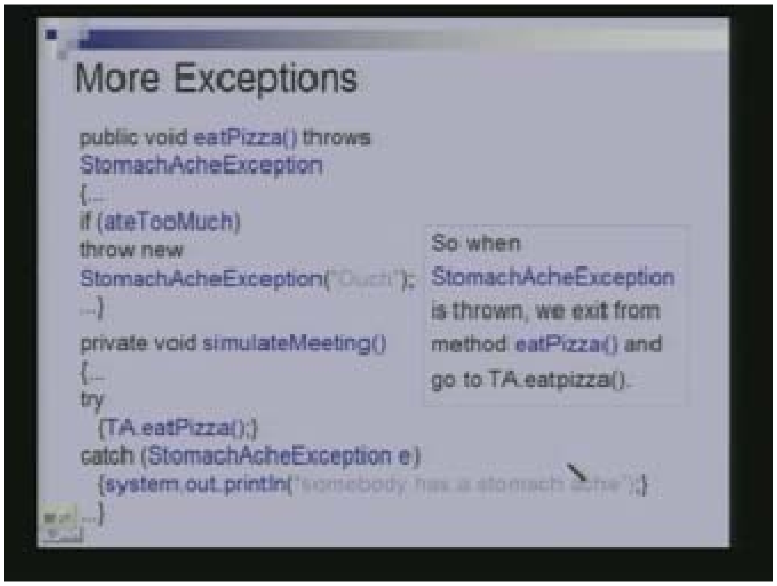
There is something called as a try and a catch block. If you think that there could be possible exception in this (TA.eatpizza ()) method, then you enclose the method within a try block. Start it with a try, open a bracket, and then include the method which you are calling and close it with a bracket.

If there was no exception raised in TA.eatpizza () method or this particular exception StomachAcheException did not get raised in this method, then we will just skip the catch block, then go on to the statement, after the catch block.

If an exception was raised in this (TA.eatpizza ()) method, because this method might raise many exceptions. If this (StomachAcheException) exception was raised in the method, then we would come in to the catch block and execute the statements.

If the method raises an exception, then if that exception is caught through a catch block, then we would execute the statements which are written inside the catch block. Any kind of statements can be written inside the catch block, not necessarily System.out.exception

(Refer Slide Time: 20:50)



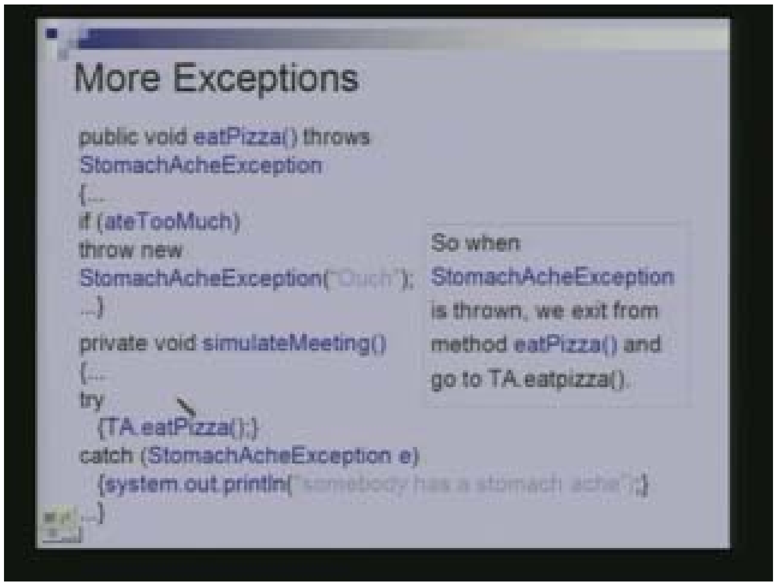
More Exceptions

```
public void eatPizza() throws
StomachAcheException
{...
if (ateTooMuch)
throw new
StomachAcheException("Ouch");
...}
private void simulateMeeting()
{...
try
  {TA.eatPizza();}
catch (StomachAcheException e)
  {system.out.println("somebody has a stomach ache");}
...}
```

So when StomachAcheException is thrown, we exit from method eatPizza() and go to TA.eatpizza().

What would happen, if I did not write the catch block? This procedure simulate meeting, would throw the exception to the point from where its parent procedure was called. When StomachAcheException throws an exception, the TA.eatpizza () method would also throw an exception, then the control will go to procedure from where simulate meeting is called. It is fine if it catches the exception at that point, if not it will throw an exception to the high level procedure and finally your procedure will stop with your exception appearing at your console.

(Refer Slide Time: 22:25)



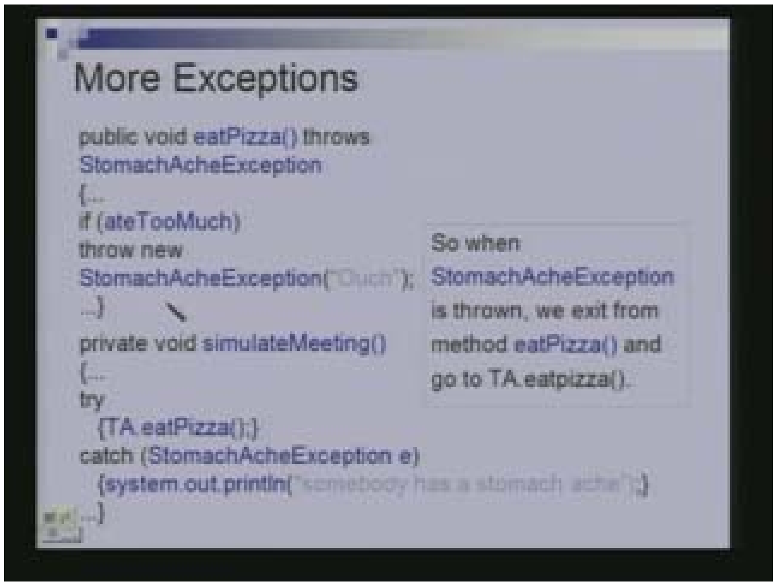
More Exceptions

```
public void eatPizza() throws
StomachAcheException
{...
if (ateTooMuch)
throw new
StomachAcheException("Ouch");
...}
private void simulateMeeting()
{...
try
  {TA.eatPizza();}
catch (StomachAcheException e)
  {system.out.println("somebody has a stomach ache");}
...}
```

So when StomachAcheException is thrown, we exit from method eatPizza() and go to TA.eatpizza().

In this manner it is getting propagated all the way up to, where your procedure stops and the exception is shown to the user. System.out.println is just the method to print the statement.

(Refer Slide Time: 23:41)



More Exceptions

```
public void eatPizza() throws
StomachAcheException
{...
if (ateTooMuch)
throw new
StomachAcheException("Ouch");
...}
private void simulateMeeting()
{...
try
  {TA.eatPizza();}
catch (StomachAcheException e)
  {system.out.println("somebody has a stomach ache");}
...}
```

So when StomachAcheException is thrown, we exit from method eatPizza() and go to TA.eatpizza().

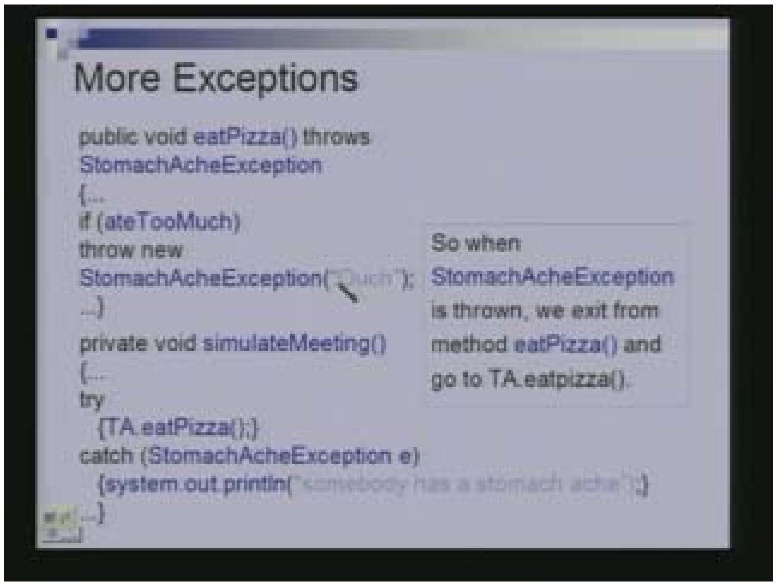
An exception is really a java class in which I am creating an object or an instance for this class. Then I am initializing that instance with any parameter and I can specify some set of parameters in the statement given below.

StomachAcheException (“Ouch”); StomachAcheException itself is a class and for this class, I am creating an object by making a call to the statement. StomachAcheException (“Ouch”); When the catch statement is caught, e in that statement would get assigned to the object that is created by StomachAcheException (“Ouch”) statement.

The try and catch block would come together. If the method were not enclosed between try and catch, then the exception would just get propagate upwards in the procedural hierarchy. StomachAcheException would throw an exception and the calling procedure of simulate Meeting would throw an exception, till it is caught at some point. If not it reaches the console.

What does the name of the class followed by brackets and some parameters written would signify in java? For example: StomachAcheException (“Ouch”); In java it signifies, that you are creating an object for this class and you are invoking the constructor method with “Ouch” as the parameters.

(Refer Slide Time: 24:07)



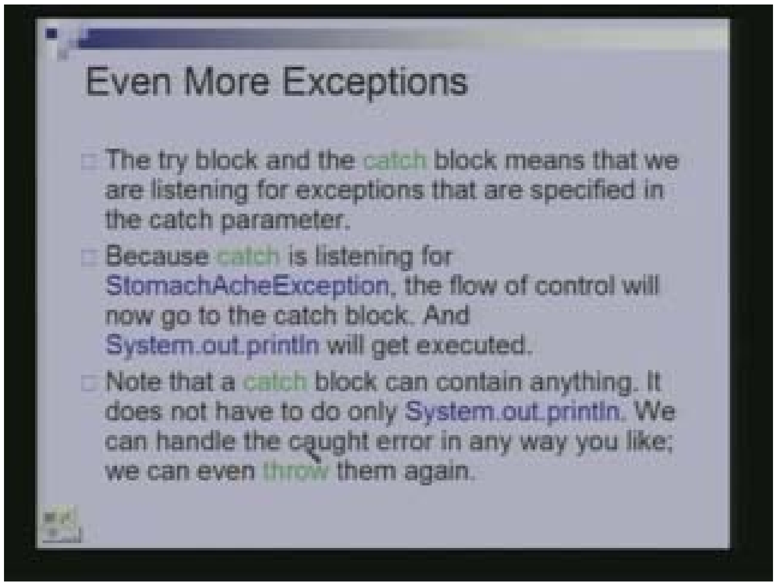
More Exceptions

```
public void eatPizza() throws
StomachAcheException
{...
if (ateTooMuch)
throw new
StomachAcheException("Ouch");
...}
private void simulateMeeting()
{...
try
  {TA.eatPizza();}
catch (StomachAcheException e)
  {system.out.println("somebody has a stomach ache");}
...}
```

So when StomachAcheException is thrown, we exit from method eatPizza() and go to TA.eatpizza().

The try and catch block are a method for listening exceptions and catching them. As I mentioned before, a catch block can contain anything. It does not mean that it should have only system.out.println, it can also throw an exception in turn.

(Refer Slide Time: 26:16)



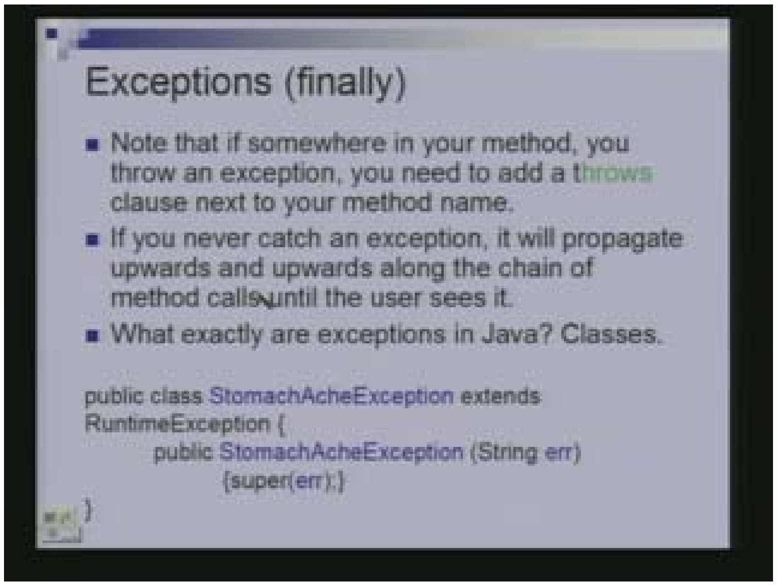
Even More Exceptions

- The try block and the catch block means that we are listening for exceptions that are specified in the catch parameter.
- Because catch is listening for StomachAcheException, the flow of control will now go to the catch block. And System.out.println will get executed.
- Note that a catch block can contain anything. It does not have to do only System.out.println. We can handle the caught error in any way you like; we can even throw them again.

It also helps us to exit from the program when an exception occurs. If you throw an exception in any method, then you need to add a throws class next to the method name. When we wrote the method eatPizza () we had, throws StomachAcheException. A method can throw more than one exception.

In java everything is really an object. StomachAcheException is the name of the class. Public class **StomachAcheException** extends And the statement given below is the constructor method for the class. Thus the name of the constructor method is the same as the class name. Public StomachAcheException (string err) The constructor method takes a single parameter, which is a string. Super means that it is calling the super class with the same parameter.

(Refer Slide Time: 28:00)



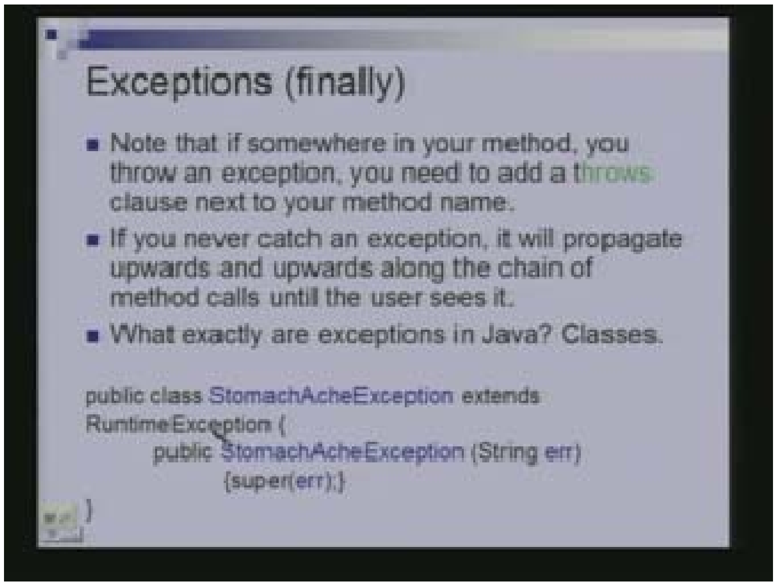
Exceptions (finally)

- Note that if somewhere in your method, you throw an exception, you need to add a throws clause next to your method name.
- If you never catch an exception, it will propagate upwards and upwards along the chain of method calls until the user sees it.
- What exactly are exceptions in Java? Classes.

```
public class StomachAcheException extends
RuntimeException {
        public StomachAcheException (String err)
                {super(err);}
}
```

Again as I mentioned before, if you never catch an exception it will propagate upwards, along with the chain of method calls, till it reaches the console. Since the stomach ache exception is extending a run time exception, it will call the constructor method for the run time exception.

(Refer Slide Time: 28:17)



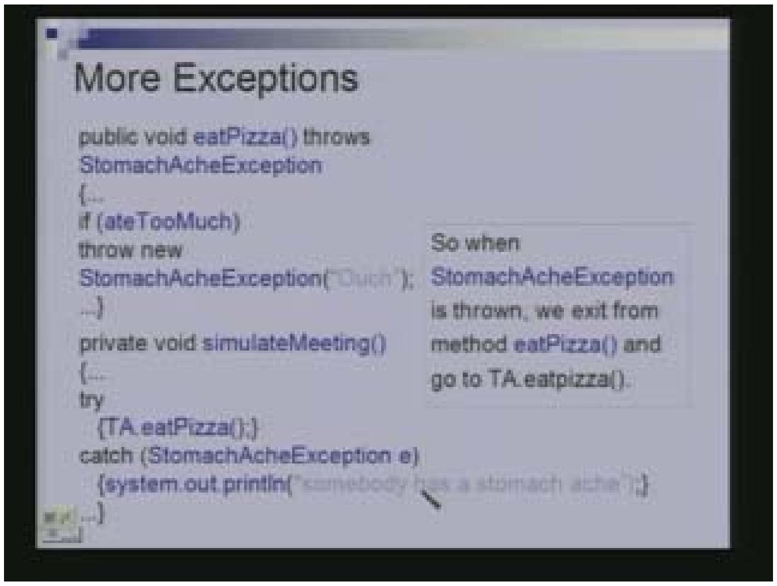
Exceptions (finally)

- Note that if somewhere in your method, you throw an exception, you need to add a throws clause next to your method name.
- If you never catch an exception, it will propagate upwards and upwards along the chain of method calls until the user sees it.
- What exactly are exceptions in Java? Classes.

```
public class StomachAcheException extends
RuntimeException {
        public StomachAcheException (String err)
                {super(err);}
}
```

If a particular method throws more than one exception, then you will have to specify all those exceptions which it throws, next to the method name. Even in the try block you can have many catch statements. First we can catch one particular exception followed by some other exception and so on. Look at your java book for more details.

(Refer Slide Time: 29:10)



More Exceptions

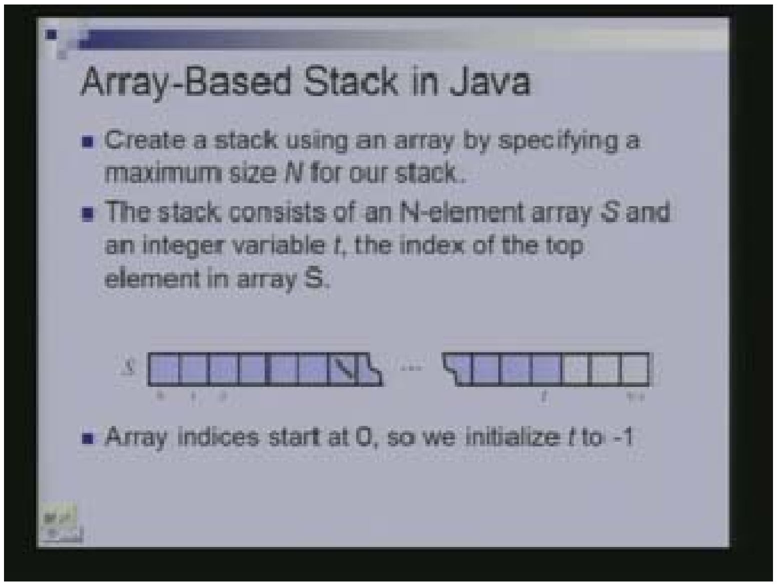
```
public void eatPizza() throws
StomachAcheException
{...
if (ateTooMuch)
throw new
StomachAcheException("Ouch");
...}
private void simulateMeeting()
{...
try
  {TA.eatPizza();}
catch (StomachAcheException e)
  {system.out.println("somebody has a stomach ache");}
...}
```

So when StomachAcheException is thrown, we exit from method eatPizza() and go to TA.eatpizza().

Let us look at the stacks. We had created the interface for our stack. We are going to implement the methods and there are many ways of implementing a stack. First we are going to implement using an array.

Let us say the maximum size of our stack is N and I am going to have an array of n elements of the stack. I am going to have a variable t, which will tell about the location of the top element of the stack. The variable t gives the index of the top element in the array S. The first element will be at location 0 and then when I push another element it will move to the next location and so on.

(Refer Slide Time: 29:43)



Array-Based Stack in Java

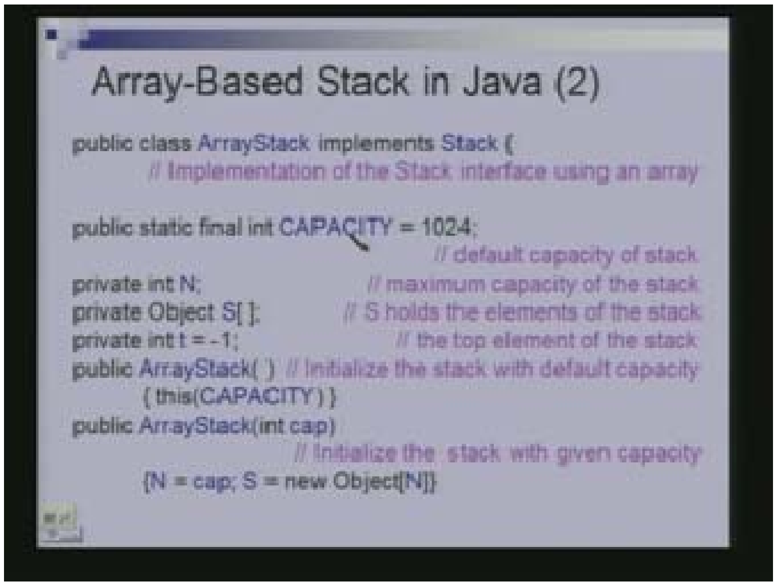
- Create a stack using an array by specifying a maximum size *N* for our stack.
- The stack consists of an N-element array *S* and an integer variable *t*, the index of the top element in array S.

- Array indices start at 0, so we initialize *t* to -1

I have actually listed out an entire implementation for our stack interface. My implementation is called array stack because I am using an array to implement the stack.

The statement mentioned below says that I am implementing the stack interface. Public class ArrayStack implements Stack Implement stack means, it is implementing the stack interface that we provided. I have set with a default capacity for the stack which is 1024, otherwise the capacity of the stack would be in the variable N. Final is just specifying that the value of capacity is always a constant and it can never be changed.

(Refer Slide Time: 30:48)



Array-Based Stack in Java (2)

```
public class ArrayStack implements Stack {
        // Implementation of the Stack interface using an array

public static final int CAPACITY = 1024;
                                        // default capacity of stack
private int N;                          // maximum capacity of the stack
private Object S[ ];                    // S holds the elements of the stack
private int t = -1;                     // the top element of the stack
public ArrayStack( )  // Initialize the stack with default capacity
        { this(CAPACITY) }
public ArrayStack(int cap)
                                // Initialize the  stack with given capacity
        {N = cap; S = new Object[N]}
```

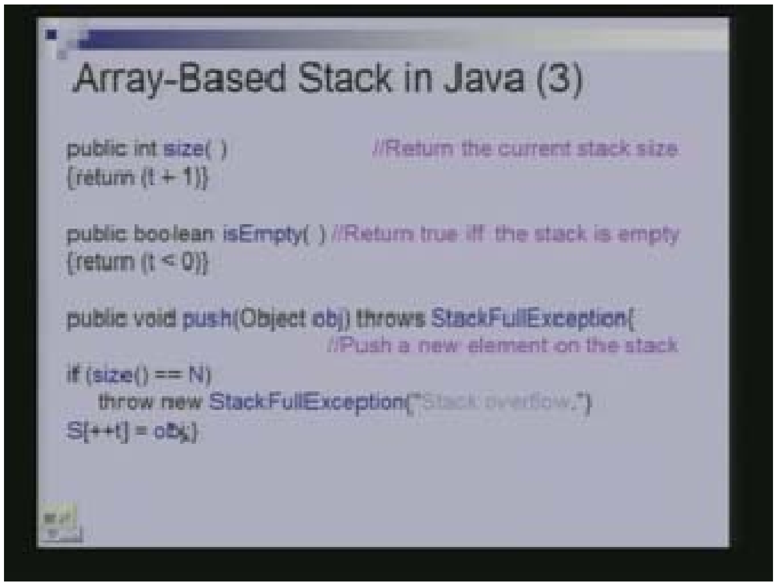
S is an array which is going to hold the elements of the stack. Thus S is an array of object and t is the index of the top element. Initially t =-1, because there is nothing inside the t. t=0 means the top element is in the location 0 and when the stack is empty t =-1.

Public ArrayStack ()
Public ArrayStack (int cap)

The above two statements are the constructor methods. If you do not specify anything or if you just call the array stack without any parameters, then I am going to create a stack whose capacity is 1024. If you call array stack with some number let us say 37, then I am going to create a stack of size 37.

What should size do? Size should just return how many elements are there in my stack. If t is the index of the top element, then t+1 elements are there because we just started from zero. The stack is empty if t =-1 that is t <0. If t <0 then isEmpty () method would return true, otherwise it returns false.

(Refer Slide Time: 32:42)



Array-Based Stack in Java (3)

```
public int size( )                //Return the current stack size
{return (t + 1)}

public boolean isEmpty( ) //Return true iff the stack is empty
{return (t < 0)}

public void push(Object obj) throws StackFullException{
                                  //Push a new element on the stack
if (size() == N)
    throw new StackFullException("Stack overflow.")
S[++t] = obj;}
```

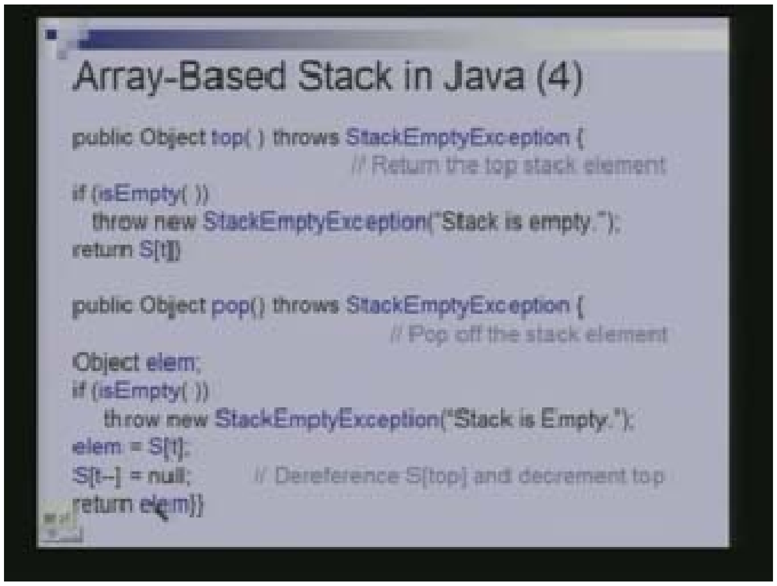
If I want to push an object ob in to the stack and if the size of the stack already equals n, then I should throw a stack full exception. Else I should first increment t and then put the object at the new incremented location. S [++t] is the first increment, and then put the object at that location.

I have to give you the top element of the stack for that, I should check whether the stack is empty or not. If the stack is empty then I throw a stack empty exception. If the stack is empty then the flow of control would exit from throw new Stack Empty Exception ("Stack is empty"). And when the stack is not empty it just returns the top element of the stack S[t].

If I want to pop the stack, then once again I check if the stack is empty. If the stack is not empty then I save the top element in location elem. Then I decrement t, because I am removing the top element and to the location which I set earlier was set to null that is I dereference it. Because earlier at the top location t, it was initially 37 and I had an object in that location. I need to remove that object and decrement t to36. Then I return the top element. Pop also returns the top element.

```
        S [t--] =null;
        Return elem
```

(Refer Slide Time: 34:02)



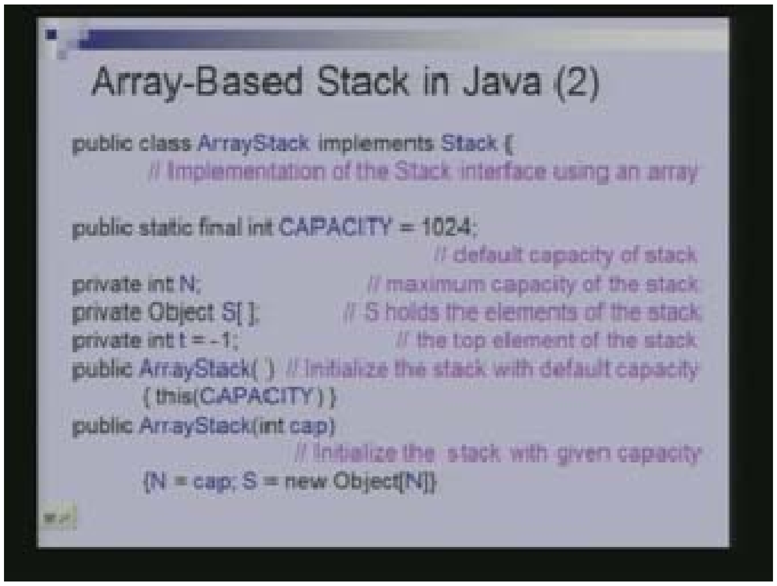
Array-Based Stack in Java (4)

```
public Object top( ) throws StackEmptyException {
                                        // Return the top stack element
if (isEmpty( ))
   throw new StackEmptyException("Stack is empty.");
return S[t]}

public Object pop() throws StackEmptyException {
                                         // Pop off the stack element
Object elem;
if (isEmpty( ))
    throw new StackEmptyException("Stack is Empty.");
elem = S[t];
S[t--] = null;         // Dereference S[top] and decrement top
return elem}}
```

Stack Empty Exception is a class and I use new to create an object or an instance for this class. Why should we necessarily dereference the objects? It is best to deference, because you can remove those objects or you can get rid of those objects otherwise they will lie in your memory. Thus t is just an integer and it is a private member of this class, private because no one else knows about t and S is an array of objects and S can therefore access the t^{th} element of this array.

(Refer Slide Time: 36:54)



Array-Based Stack in Java (2)

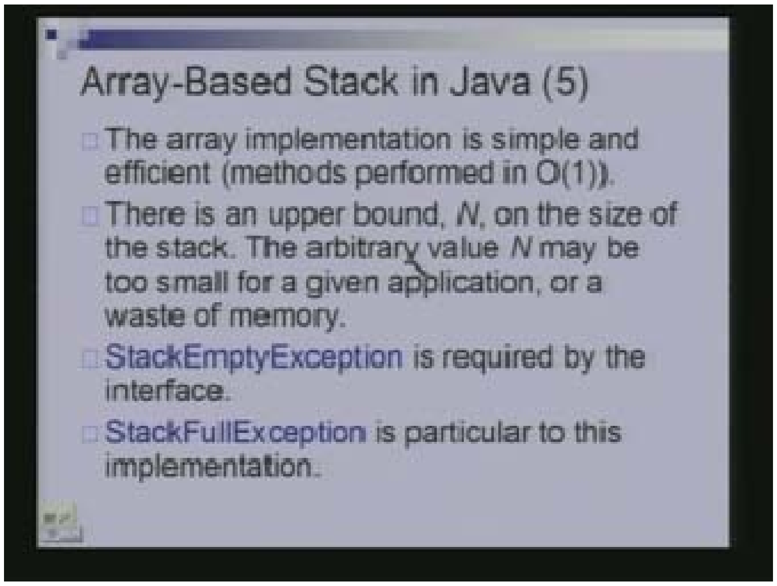
```
public class ArrayStack implements Stack {
          // Implementation of the Stack interface using an array

public static final int CAPACITY = 1024;
                                      // default capacity of stack
private int N;                     // maximum capacity of the stack
private Object S[ ];            // S holds the elements of the stack
private int t = -1;                   // the top element of the stack
public ArrayStack( ) // Initialize the stack with default capacity
          { this(CAPACITY) }
public ArrayStack(int cap)
                          // Initialize the  stack with given capacity
          {N = cap; S = new Object[N]}
```

The array implementation is very simple as all the operations were taking constant time. None of the operations required time propositional to the virtual dependent upon the number of elements in the array in this stack at that point. Each of those methods take O (1) time. The problem is that we are working with an upper bound on the size of the stack. This upper bound may have the default value1024, which we took in our example or it may be specified at the time of creation of stack. The problem is because you do not know the size of the stack.

We might allocate a very large size for the stack, but it might be a waste of memory or we might allocate a very small stack in which we could not be able to run our procedure to complete, because we would soon have a stack full exception. Stack Empty Exception is the requirement of the interface, because the top and the pop methods are not defined, if the stack is empty.

(Refer Slide Time: 37:48)



Array-Based Stack in Java (5)

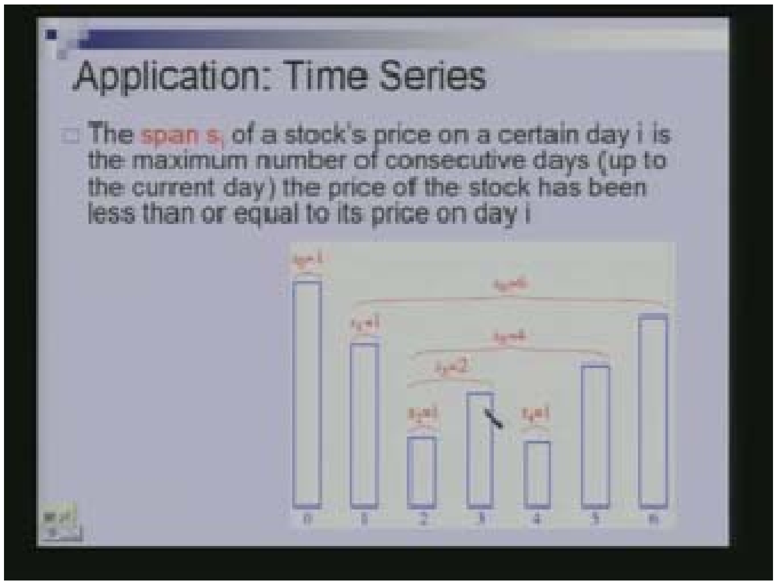
- The array implementation is simple and efficient (methods performed in O(1)).
- There is an upper bound, *N*, on the size of the stack. The arbitrary value *N* may be too small for a given application, or a waste of memory.
- StackEmptyException is required by the interface.
- StackFullException is particular to this implementation.

It is the requirement of the interface but a stack full exception is an artifact of this implementation. If I had some other way of implementation, then I would never have to raise a stack full exception. Let us look at an example. We will see how to implement the stack and never to have a stack full exception, so that we can always grow the stack when needed.

Let us look at an application of stacks. We have the daily stock prices of a particular stock. If I give you the price on day 0, the price on day 1 and so on, then the span s_i of a stock price on a certain day i is defined as the maximum number of consecutive days that the price of the stock has been less than or equal to its price on day i.

The following example would make it clear. S_5 is the span of the stock price on day five and it is equal to the maximum number of days that the price of this stock has been less than or equal to 5^{th} day price. For four days the price of this stock was less than or equal to 5^{th} day price and so the span of the stock price on day five equals 4, inclusive of the current day. Following picture would make it clear hence we are counting 1,2,3,4 and for day six, it is 1, 2, 3,4,5,6 that is 6 days.

(Refer Slide Time: 39:25)



Application: Time Series

- The span s_i of a stock's price on a certain day i is the maximum number of consecutive days (up to the current day) the price of the stock has been less than or equal to its price on day i

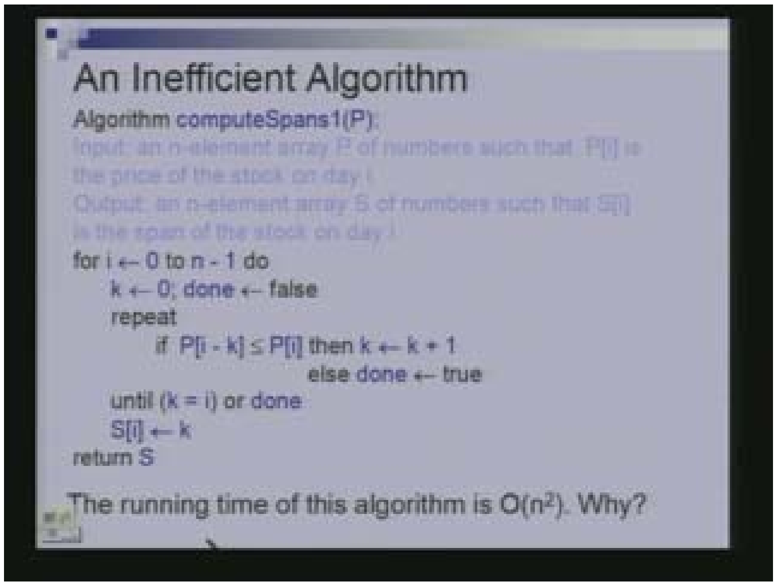
How can you compute the span in an array S, if I give you the stock prices in an array p. P is an array of numbers. To compute S[i], you are going to look at the price of the stock on day i, i-1, i-2 and so on.

The index k will start from zero and it will keep going down till the price of the stock on day i- k is less than the price of the stock on day i. The moment you find a case such that the price of stock on day i- k is actually more than the price on day i, you stop the loop which is given below, otherwise you keep incrementing k.

If P [i-k] $\leq$ P[i] then k🡐 k+1

If this quantity P [i-k] $\leq$ P[i] is less, then you increment k else you say done is true. Done will help us to exist the repeat-until loop. It will exit the repeat-until loop if done is true or if k equals i, which means that you have reached day 0. Then the span will be determined by the value of k, because k tell us the span of stock price on day i. Thus S[i] gets the value k, (S[i] 🡐 k). This is one way of computing this span.

(Refer Slide Time: 41:07)



An Inefficient Algorithm

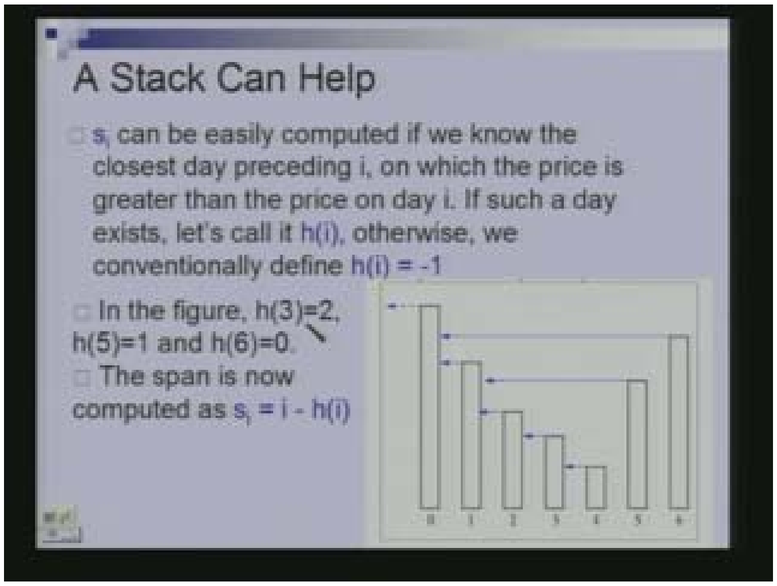
```
Algorithm computeSpans1(P):
Input: an n-element array P of numbers such that P[i] is
the price of the stock on day i
Output: an n-element array S of numbers such that S[i]
is the span of the stock on day i
for i ← 0 to n - 1 do
    k ← 0; done ← false
    repeat
        if P[i - k] ≤ P[i] then k ← k + 1
                        else done ← true
    until (k = i) or done
    S[i] ← k
return S
```

The running time of this algorithm is O(n^2). Why?

How much time does it take? It takes n^2 time as we can see on the slide. Why should it take n^2 time? Because we are repeatedly comparing it How many times the repeat-until loop can be executed in the worst case? It is i times, where i varying itself from 0 through n-1. The total number of times one of this statement get executed might be n^2 or $\frac{n^2}{2}$. Thus the running time of this algorithm is O (n^2) in the worst case.

Question is can we do something better? Yes, as we are talking of stacks, we can use a stack to do something better. To compute the span, we need to know the closest day preceding i, on which the stock price is greater than the price on day i.

(Refer Slide Time: 43:26)

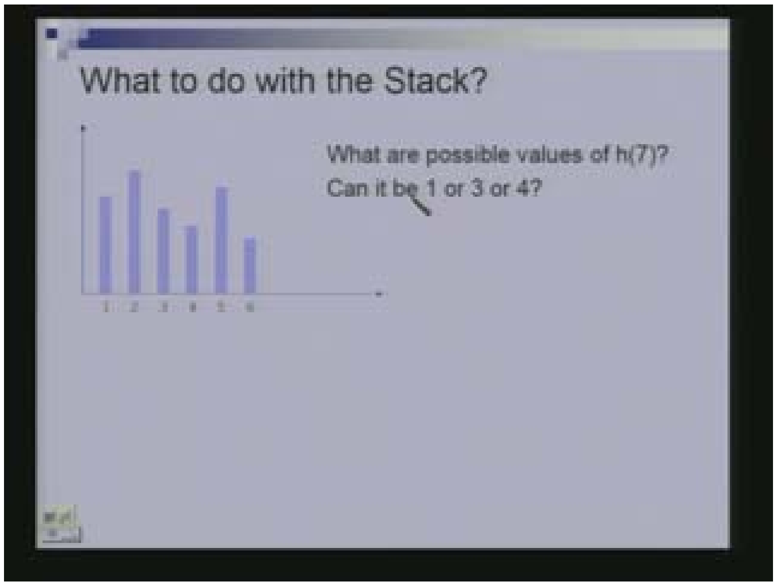


A Stack Can Help

- s_i can be easily computed if we know the closest day preceding i, on which the price is greater than the price on day i. If such a day exists, let's call it h(i), otherwise, we conventionally define h(i) = -1
- In the figure, h(3)=2, h(5)=1 and h(6)=0.
- The span is now computed as s_i = i - h(i)

In the example, for day 5 I need to know the closest day preceding day 5, on which the stock price is greater than the price on day 5. On day 1, it was greater. For day 5, the span would be 1,2,3,4. I am going to call the quantity h (i). h (i) is the closest day preceding i, on which the price is greater than the price on day i. Thus h (3) =2, h(2)=1, h(1)= 0, h(0)= -1 and h(4)=3, h(5)=1, h(6)=0. Once you computed h, how can you determine the span S_i? The span for the price on day 5 is 5-1 that is 4. If we can compute these h quantities, we can easily compute the span.

(Refer Slide Time: 45:31)



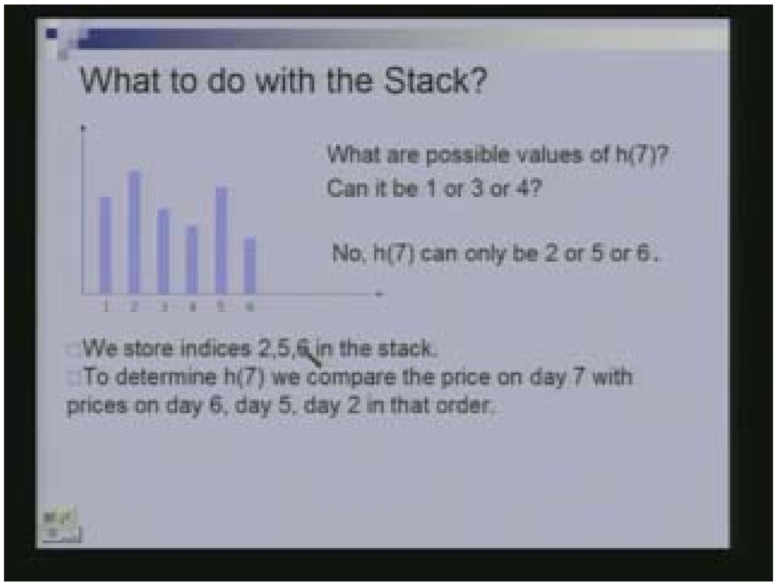
What to do with the Stack?

What are possible values of h(7)?
Can it be 1 or 3 or 4?

How do we compute the h quantities? Suppose those 6 quantities which is in the slide, were the prices that I have given from day 1 through 6. Can h (7) be 1? I have not told the price on day 7 but through the definition of h (7), it is the closest day preceding 7, on which the stock's price is larger than price on day 7. But that day cannot be 1 at all, because the price on day 2 is larger than the price on day 1. Similarly that day cannot be 3 or 4. What are the possible values that h (7) can take? 2, 5 and 6 are the only possible values that h (7) can take.

We will store the indices 2, 5, 6 in a stack. 2 will be at the bottom of the stack, 5 will be above that and 6 on the top. To determine h (7), first we need to compare the price on day 7 with price on day 6. Suppose the price on day 7 is less than the price on day 6 then what is the h (7). It is 6, but if the price on day 7 was greater than price on day 6, then I will compare with price on day 5. If it is greater than the price on day 5, then I will compare it with 2. If it is greater than 2, then it is minus one that is h (7) =-1.

(Refer Slide Time: 46:41)



What to do with the Stack?

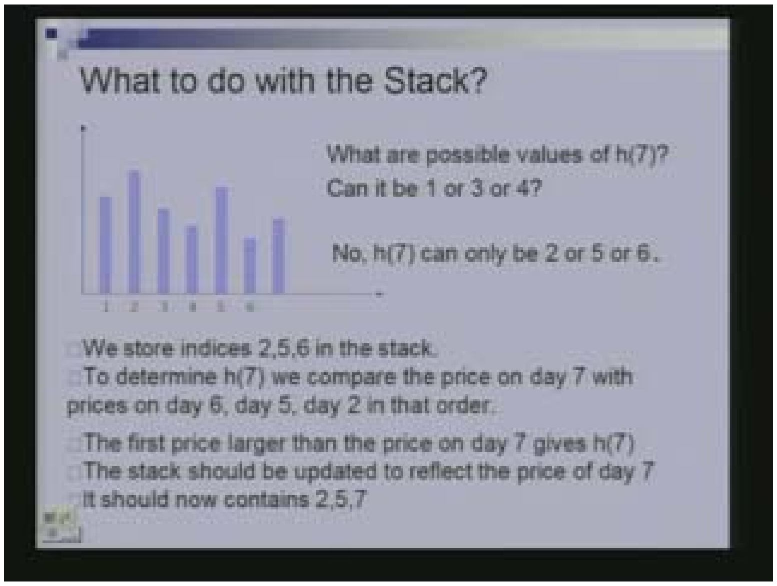
What are possible values of h(7)?
Can it be 1 or 3 or 4?

No, h(7) can only be 2 or 5 or 6.

- We store indices 2,5,6 in the stack.
- To determine h(7) we compare the price on day 7 with prices on day 6, day 5, day 2 in that order.

Suppose the new bar I have drawn is the price on day 7 and it is greater than the price on day 6 then h (7) is 5. The first price greater than the price on day 7 in the comparison gives me h (7). But once I know h (7), I should update my stack. Now what should my stack contain? Earlier it contains the indices 2, 5, 6 and now it contains 2, 5 and 7. It is clear because h (8) can never be 6, I should get rid of 6 by replacing it with 7. You can see that the stack would be the right way to do these things.

(Refer Slide Time: 47:40)



What to do with the Stack?

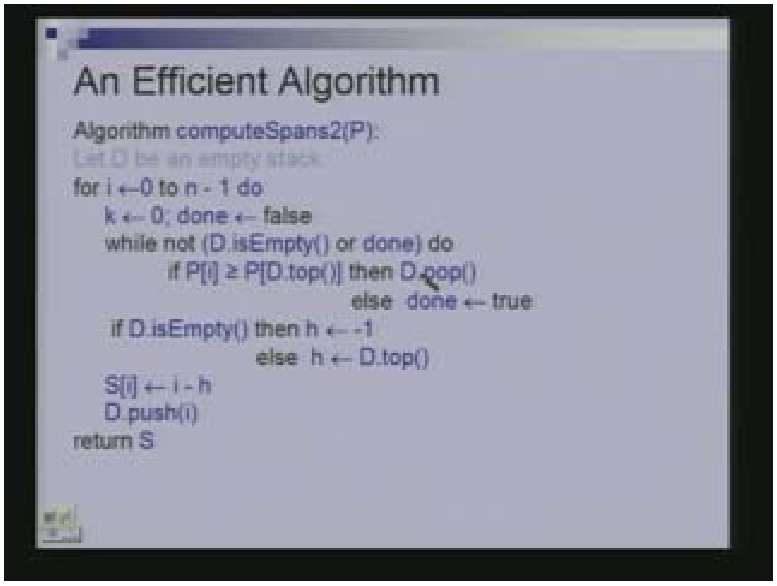
What are possible values of h(7)?
Can it be 1 or 3 or 4?

No, h(7) can only be 2 or 5 or 6.

- We store indices 2,5,6 in the stack.
- To determine h(7) we compare the price on day 7 with prices on day 6, day 5, day 2 in that order.
- The first price larger than the price on day 7 gives h(7)
- The stack should be updated to reflect the price of day 7
- It should now contains 2,5,7

Let us look at the procedure. Let d be the stack and initially it is empty. When I get a certain price, I am going to compare that price with the price on the top of the stack. If it is less than the price on the top of the stack that is on the top of the stack we had 2, 5, 6 and if the 7th bar is less than the 6 then it is just what is there on the top. Then we are done, the index on the top of the stack will give the h value. If it is more than the price on the top of the stack, then I will pop of or remove the top of the stack, because I need to compare with the next one. I pop of because I will not need that quantity any more.

(Refer Slide Time: 48:36)



An Efficient Algorithm

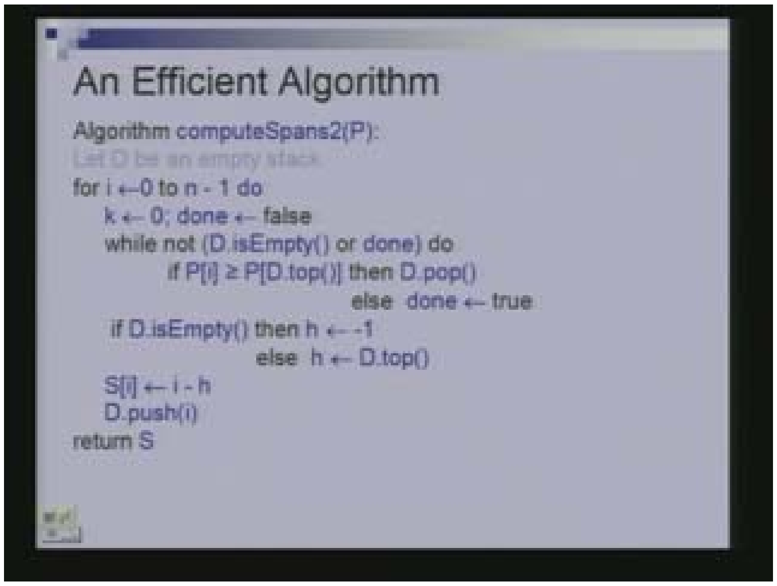
```
Algorithm computeSpans2(P):
Let D be an empty stack.
for i ← 0 to n - 1 do
    k ← 0; done ← false
    while not (D.isEmpty() or done) do
            if P[i] ≥ P[D.top()] then D.pop()
                            else done ← true
    if D.isEmpty() then h ← -1
                   else h ← D.top()
    S[i] ← i - h
    D.push(i)
return S
```

As you recall from the previous slide, since 7th bar was more than 6th bar, I can actually get rid of 6th bar because in the stack, I do not need that next time. We are going to go around the loop till either done becomes true. When it becomes true it says that I have found a price which is greater than the current day price. But if done never becomes true and the stack becomes empty then h=-1, exactly that is being set here in the below statement.

```
If D.isEmpty () then h ← -1
                  Else h ← D.top ()
```

When I exit this (P[i] $\leq$ P [D.top ()]) loop, the stack is empty then h=-1 else h is the top value of the stack. Once I know h, I can compute S[i] and keep it. I will push i back in because in my previous slide when I got this 7, I now push 7 in for my next computation. How much time does this take? May be n times. While loop might execute a lot of times and why this should take only n time. Is the worst case n^2 or n? It may be n^2.

(Refer Slide Time: 49:52)



An Efficient Algorithm

```
Algorithm computeSpans2(P):
Let D be an empty stack
for i ← 0 to n - 1 do
    k ← 0; done ← false
    while not (D.isEmpty() or done) do
            if P[i] ≥ P[D.top()] then D.pop()
                              else done ← true
     if D.isEmpty() then h ← -1
                    else h ← D.top()
    S[i] ← i - h
    D.push(i)
return S
```

How many elements do we pushed on the stack? Each element is pushed once and we have pushed at most n elements on to the stack, if there are n elements to begin with. Every time when the while loop is executed, we pop of one element from the stack. How many times the while loop gets executed? It is n times. Every time when the loop executes, we are removing an element from the stack. If the total number of elements ever we pushed on the stack was n, then how can we pop of more than n elements from the stack. It means that the total number of times the loop executed is not more than n.

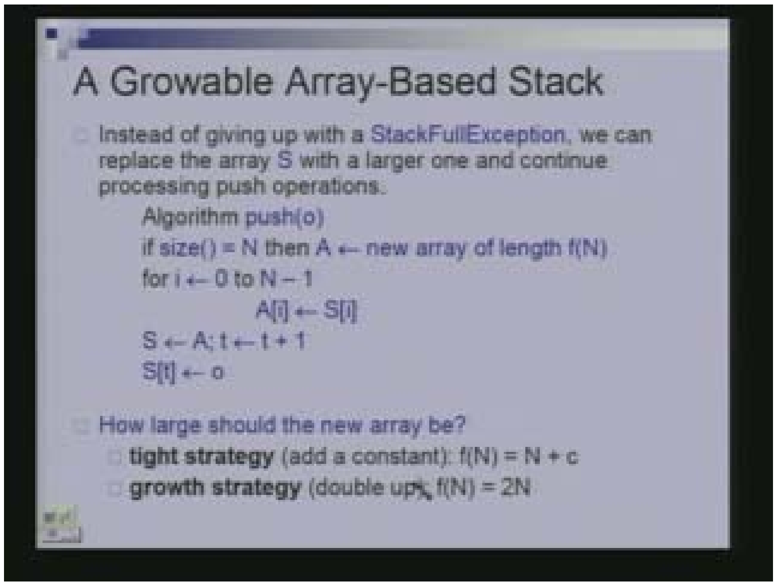
How many times do the statements inside the for-loop execute? All most n times, because these are all a part of the for-loop. If statements execute exactly n times. What is the total time it will take? It is order n and not n^2. I am saying the total number of times the loop executes when all iteration put together is no more than n. Because every time the loop executes and when we go through the loop, we remove one element stack from the stack.

We never pushed more than n elements or the total number of elements we pushed on the stack is at most n.

One problem with our stack implementation is that we had to give maximum size for the stack. We are going to look at an implementation in which the stack can grow, if it ever gets filled. When we are pushing an element, if the size of the stack is n then I create a new array of length f (N) and I copy all the elements of my original stack S in to A and then I rename it as S. It becomes my new stack with f (N) locations in it whose capacity is f (N). Thus f (N) will be larger than n for us and so we had to increase the size of the stack. I increment the top counter and I am trying to push the new object into my top location.

How should we should f (N)? There are two strategies in which one could adopt. One could either have a tight strategy or a growth strategy. In a tight strategy, we always increment the size of the array by some constant c. We just increment that is additive increment. In the growth strategy, we double the size of the stack.

(Refer Slide Time: 54:02)



A Growable Array-Based Stack

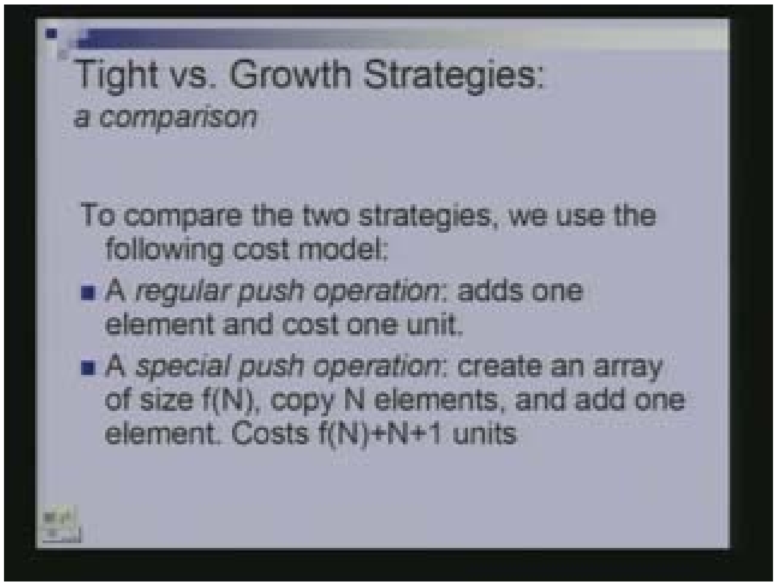
- Instead of giving up with a StackFullException, we can replace the array S with a larger one and continue processing push operations.

```
Algorithm push(o)
if size() = N then A ← new array of length f(N)
for i ← 0 to N – 1
        A[i] ← S[i]
S ← A; t ← t + 1
S[t] ← o
```

- How large should the new array be?
  - **tight strategy** (add a constant): f(N) = N + c
  - **growth strategy** (double up): f(N) = 2N

We want to compare and see which of these is a better strategy. We are going to think push as of two kinds. One is regular push. In a regular push, there was just space in the stack and you just push the element and it takes one unit of time. A special push is one in which the stack is already full and you have to create a larger stack and copy the elements form the earlier stack to this larger stack and then push the element.

(Refer Slide Time: 55:40)



Tight vs. Growth Strategies:

a comparison

To compare the two strategies, we use the following cost model:

- A *regular push operation*: adds one element and cost one unit.
- A *special push operation*: create an array of size f(N), copy N elements, and add one element. Costs f(N)+N+1 units

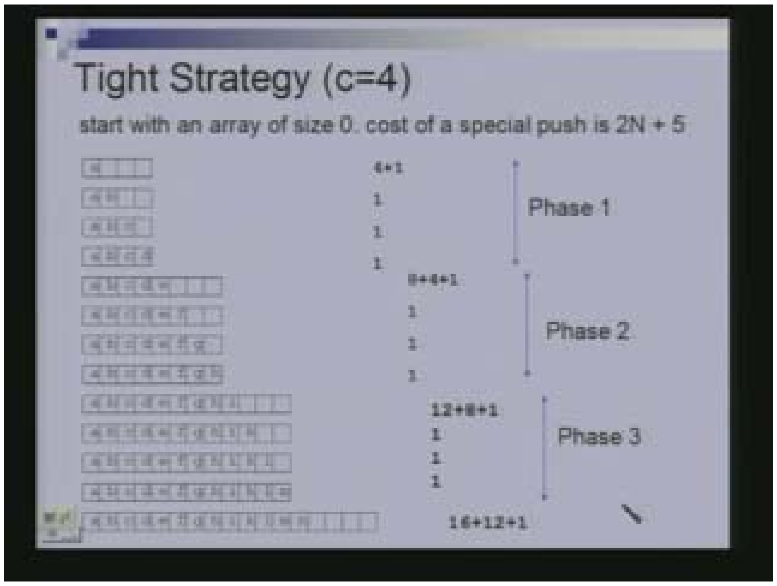
You created a stack of size f (N) that costs f (N) units and you copied the n elements that cost n units. And then you pushed one more element that cost one more unit. The total cost of this special push operation would be f (N) +N+1.

Let us see how the tight strategy behaves when we increment the size of the stack by c units. For example c is taken as 4. Initially I started with the array of size 0. When the first element came to push that was a, then I created a stack of size 4 and I push this first element in and the total cost was 4+1.

When the second element came, I do not need to enlarge my stack, because I have space. It just cost one unit which is a regular push. The 3rd and 4th operation is also a regular push which costs 1 unit each.

The next one is a special push, I am not trying to push e because the stack is already full. I need to create an array of size 8 and I need to copy the 4 elements, then I need to push 1. The total cost becomes 8+4+1 and these are all the 3 regular pushes. Then once again when I fill it, I will create an array of size 12, because c is 4. I am incrementing the size of the array by 4 units every time.

(Refer Slide Time: 56:07)



Tight Strategy (c=4)

start with an array of size 0. cost of a special push is 2N + 5

4+1

1

1

1

8+4+1

1

1

1

12+8+1

1

1

1

16+12+1

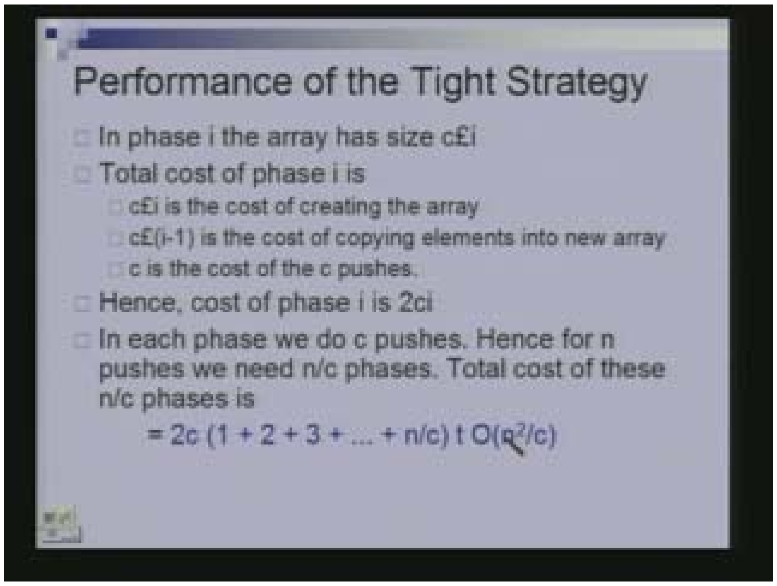
Phase 1

Phase 2

Phase 3

I create an array of size 12 and I copy the eight elements then I push one more element, so I get 12+8+1 and so on and finally I have 16+12+1. When the size of the array was 4, I am going to call it as phase 1 and when the size of the array is 8, I will call it as phase 2 and so on. Let us see the total cost of the procedure.

(Refer Slide Time: 58:18)



Performance of the Tight Strategy

- In phase i the array has size c£i
- Total cost of phase i is
  - c£i is the cost of creating the array
  - c£(i-1) is the cost of copying elements into new array
  - c is the cost of the c pushes.
- Hence, cost of phase i is 2ci
- In each phase we do c pushes. Hence for n pushes we need n/c phases. Total cost of these n/c phases is

  = 2c (1 + 2 + 3 + ... + n/c) t O(n^2/c)

The pound symbol in the above slide is a multiplication operator. In phase I, the size of the array is c * i. In phase 1 it is c, in phase 2 it is 2 c, in phase 3 it is 3c and so on. The size of the array is c*i.

What is the total cost of phase i? t At the beginning of phase i, I first create an array of size c* i then I copy the elements of the previous array. Let us look at an example. I first create an array of size 8 then I copy the previous 4 elements. So I copy c* i-1 elements and it is the cost of copying the elements in to the new array. Then I will do c more pushes in this phase before the array gets filled. C is the total cost of the 4 regular pushes that I have been doing. The total cost is c* i + c*i-1+c which is 2 times ci. Thus the cost of phase i is 2ci.

In each phase I am doing c pushes, then if I have to do a total of n pushes then I need $\frac{n}{c}$ phases. Total cost of $\frac{n}{c}$ phases would be 2c (1+2+3+…+$\frac{n}{c}$) because the cost of i^{th} phase is 2ci and this sum is roughly $\frac{n^2}{c^2}$ times 2c and that is not t, it is approximately O ($\frac{n^2}{c}$).

So far we have seen the tight strategy. We could also take creation as order 1, in that case the analysis would change slightly. For the purposes of analysis I am just taking it as, if you are creating an array of size something, you take that much as the cost.

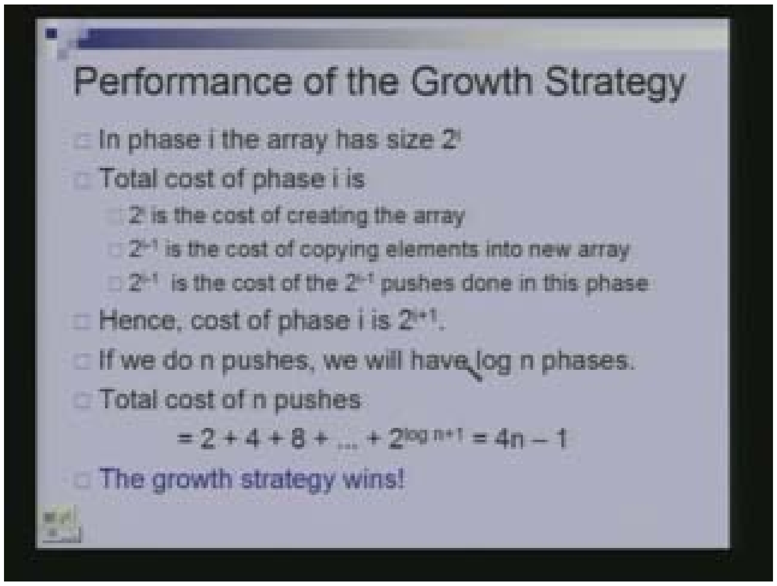
Let us see the growth strategy. In the growth strategy, I start with an array of size 0, when I get the first element I create an array of size 1. When I am trying to push an element I would double this array, so I create an array size 2 and I push this element. When I try to push the element, I double this array and create an array of size 4 and push the element. I have space for one more element and this is the regular push while pushing d. When I try to push the 5th element, I will double the size of array again, so I create an array of size 8 and copy these elements and then push the 5th element and so on.

Once again we can analyze the cost. 1 is the cost of creating the array and you do not have to copy anything and 1 is the cost for pushing them. Here we created an array of cost 2, we copied 1 element and 1 was the cost of pushing. We created an array of size 4, 2 was the cost of copying, 1 was the cost of pushing the element c and here was the regular push so it was 1 and so on.

We define a phase as when the size of the array was 1, we call it as Phase 0. When it is 2 we call it as phase 1. When it was 4, we call it phase 2 and when it was 8, we call it as phase 3 and so on. In phase i, the array has size 2^i. Phase 3 it has size 8 and phase 4 it has size 16 and so on. In the phase i, since the size is 2^i I spent 2^i units of time in creating the array. Then I have to copy elements of the previous array that is 2^{i-1} elements.

How many elements are left after copying 2^{i-1} elements? 2^{i-1} elements are still left and they have to be pushed in, in this phase. 2^{i-1} is the cost of pushing in those 2^{i-1} elements. The total cost of phase i is $2^i + 2^{i-1} + 2^{i-1}$ which is 2^{i+1}. If we do n push, we would have log n phases, because the way the array is growing.

(Refer Slide Time: 1:00:55)



Performance of the Growth Strategy

- In phase i the array has size 2^i
- Total cost of phase i is
  - 2^i is the cost of creating the array
  - 2^{i-1} is the cost of copying elements into new array
  - 2^{i-1} is the cost of the 2^{i-1} pushes done in this phase
- Hence, cost of phase i is 2^{i+1}.
- If we do n pushes, we will have log n phases.
- Total cost of n pushes

  $= 2 + 4 + 8 + \ldots + 2^{\log n+1} = 4n - 1$
- The growth strategy wins!

The total cost of n pushes is going to be 2+4+8+…+ $2^{\log n+1}$, which is 4n. The total cost of n pushes is 4n growth strategy and in the tight strategy it is $\frac{n^2}{c}$. Hence this is clearly a better strategy.