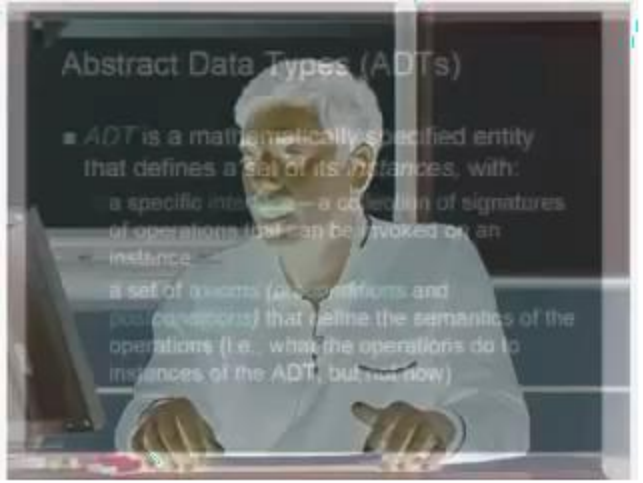
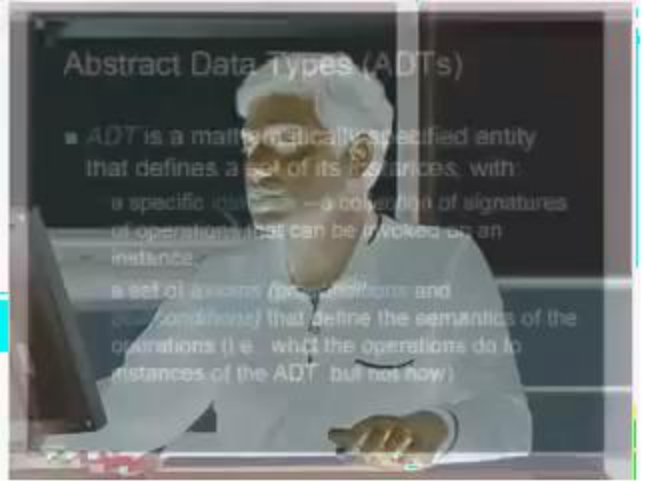
Stacks

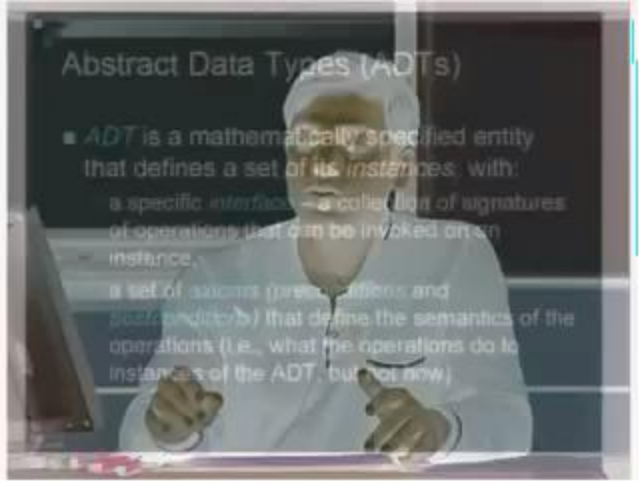
- Abstract Data Types (ADTs)
- Stacks
- Interfaces and exceptions
- Java implementation of a stack
- Application to the analysis of a time series
- Growable stacks
- Stacks in the Java virtual machine


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)

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)


Abstract Data Types (ADTs)
an 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 (pre-conditions and
post-conditions) that define the semantics of the
operations (i.e. what the operations do to
instances of the ADT, but not how)


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)

Abstract Data Types (ADTs)

Types of operations:

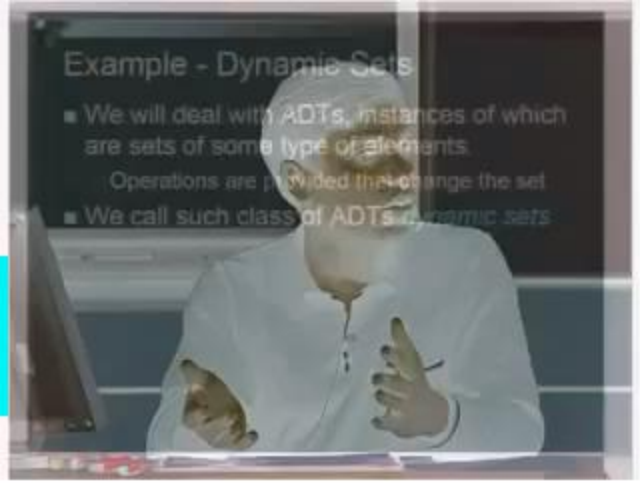
- Constructors
- Access functions
- Manipulation procedures

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*

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*


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

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

Dynamic Sets (2)

- An example dynamic set ADT
  - Values: finite subsets of some type of elements
  - Operations are provided that change the set
    - 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
- We call such class of ADTs *dynamic sets*

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 ≠ u*
  - **IsIn(Delete(S, v), v)** = *false*
  - **IsIn(Delete(S, u), v)** = **IsIn(S, v)**, *if v ≠ u*