Abstract Data Type

Abstract Data Types

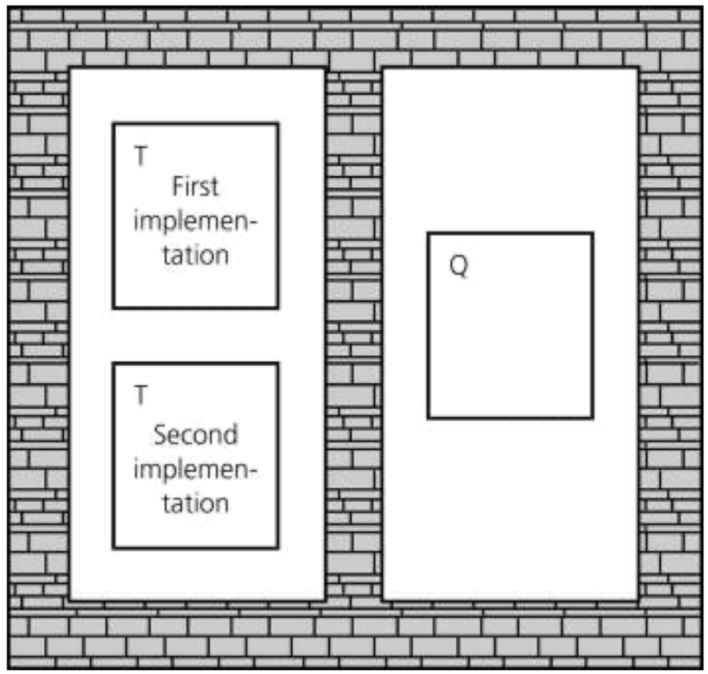
- An ADT is composed of
  - A collection of data
  - A set of operation on that data
- Specification of an ADT
  - Describes precisely what effect the operations have on the data but
  - does not describe **how** operations are implemented
- An ADT is not only a data structure!
  - Implementation of an ADT includes choosing a particular data structure

Concrete Data Types

- The term ***concrete data type*** is usually used in contrast with ADT
  - An ADT is a collection of data and a set of operations on the data
  - A concrete data type is an implementation of an ADT using a specific data structure
- In Java, it is called *Collection* (or *Container*)

Abstraction

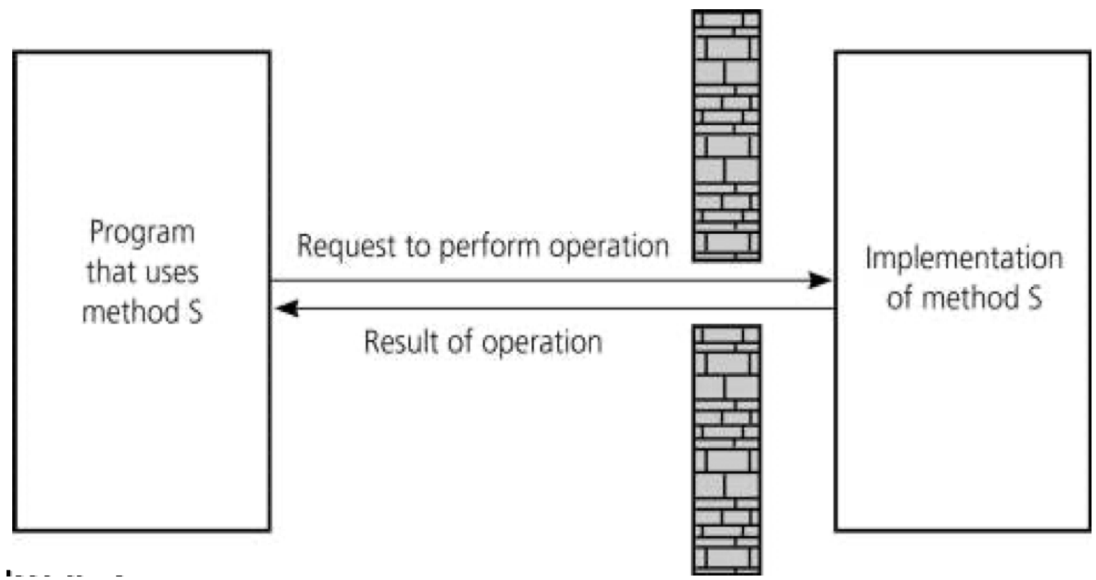
- Assume Q is a task and T is an ADT which is used by task Q



  - We might have different options to implement T but, the implementation of T does not affect task Q and it is hidden from Q

Abstraction

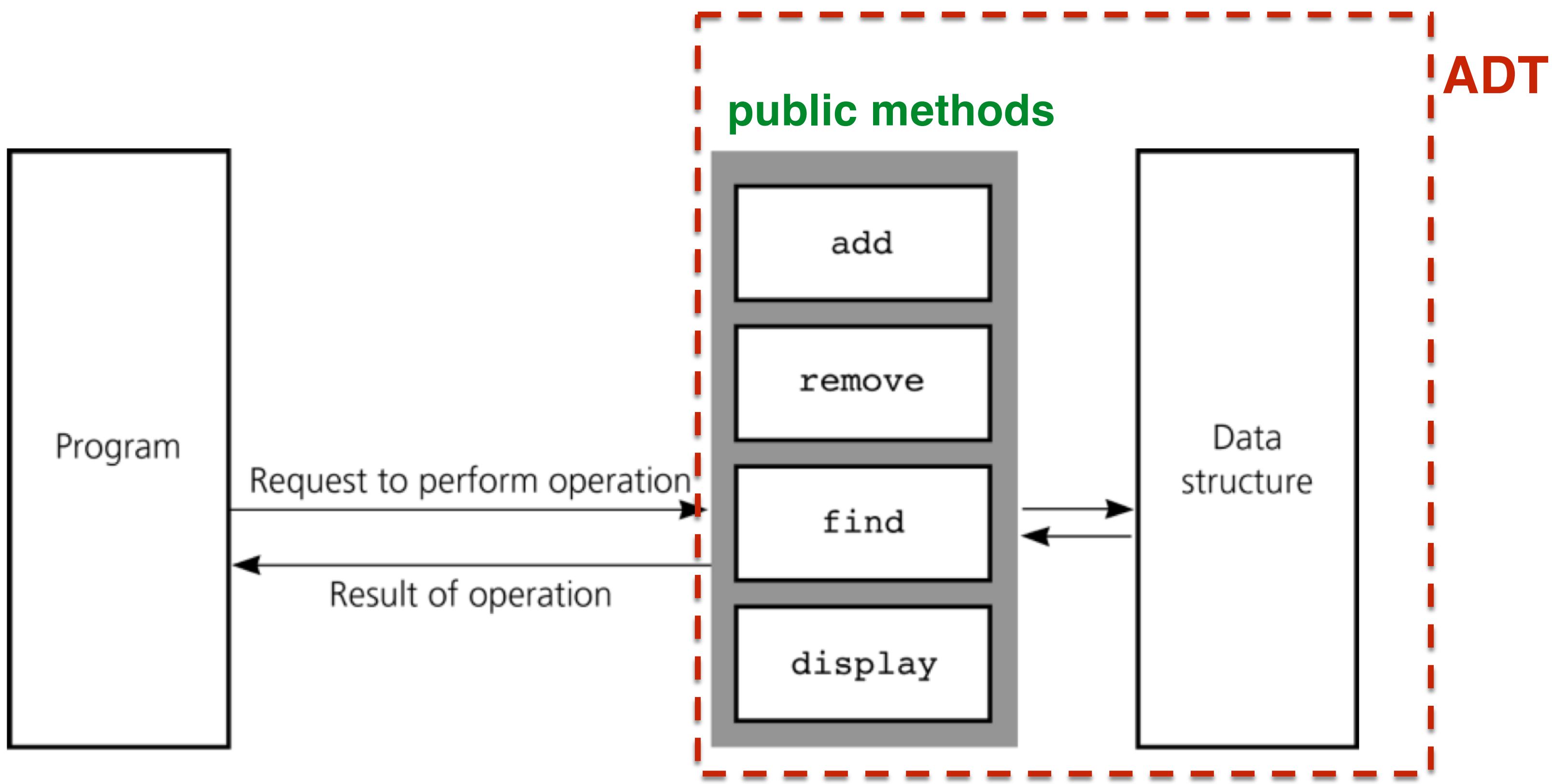
- But abstraction (isolation) is not total
- We need to know how to interact with the ADT



- Methods' specifications govern how they interact with each other

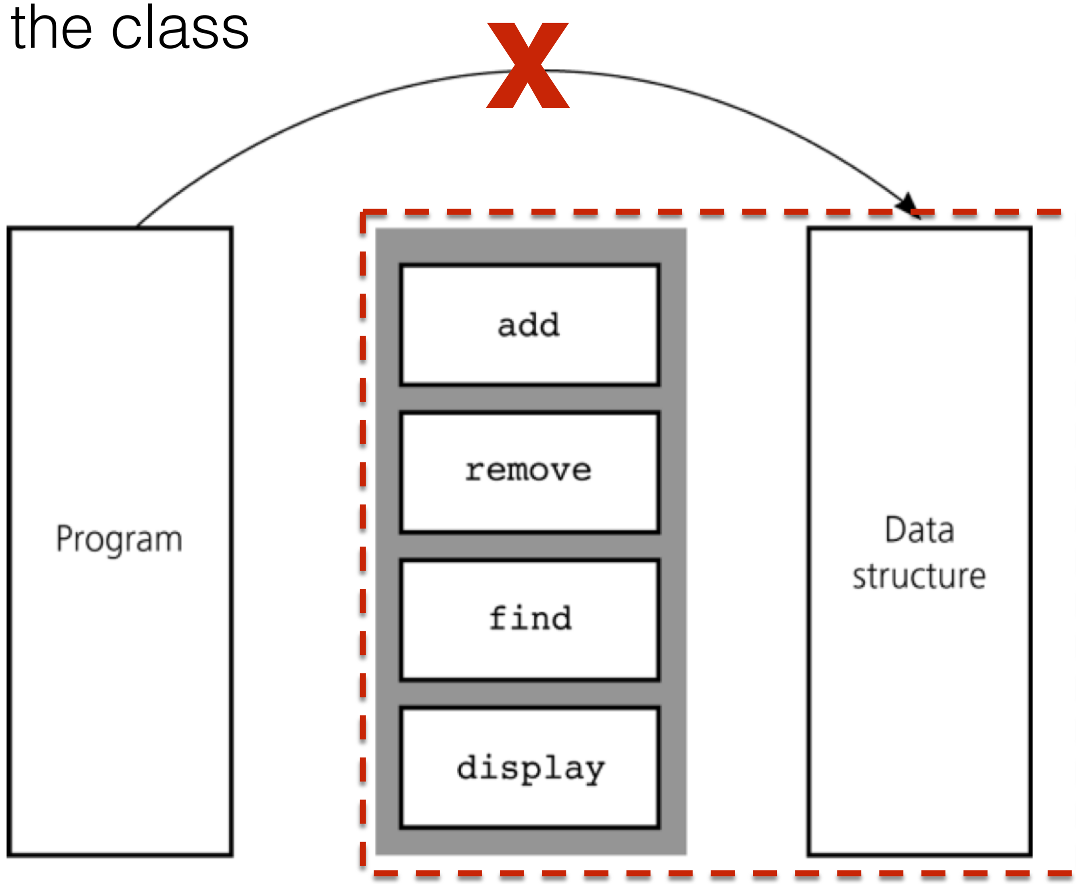
Implementing ADT

- Methods' specifications govern how they interact with each other
- ADT operations provides access to the data



Implementing ADT

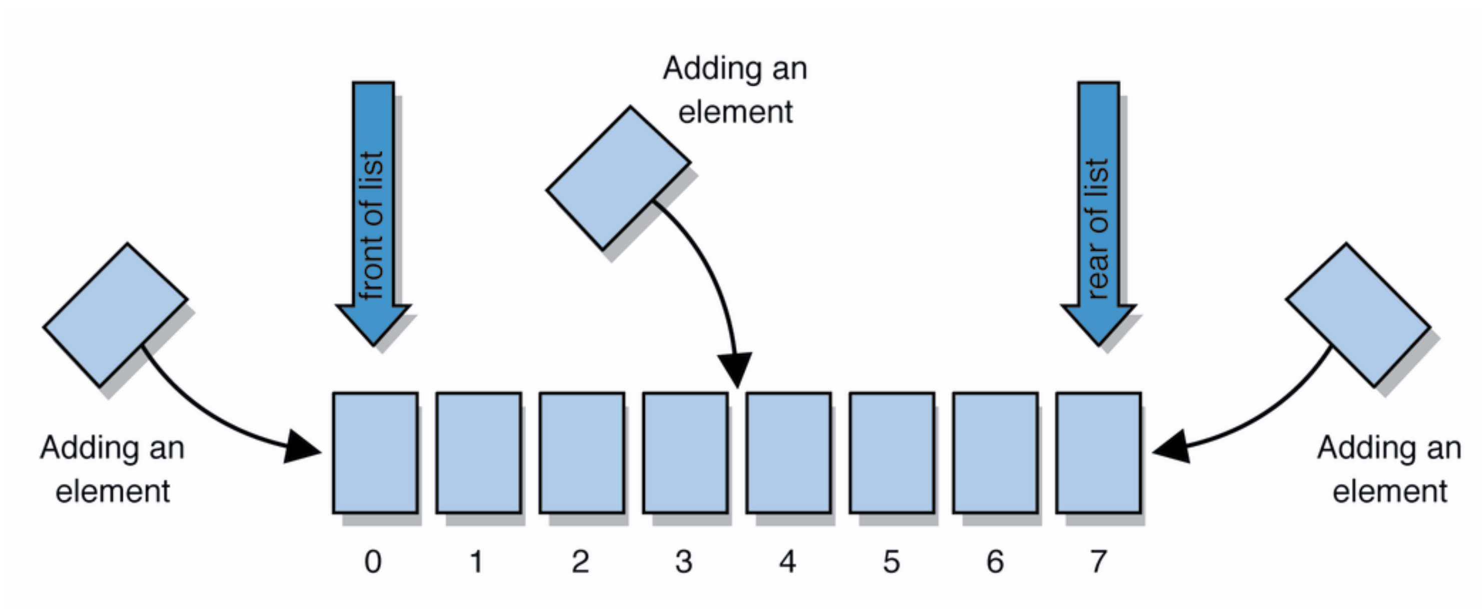
- Abstraction prevent direct access to data
- Implementation details should be firmly hidden from outside of the class



Examples

List

- A collection storing an ordered sequence of elements
  - Each element is accessible by a **0-based index**
  - A list has a **size** (number of elements that have been added)
  - Elements can be added to the front, rear, or elsewhere



Idea of a List

- Rather than creating an array of boxes, create an object that represents a "list" of items. (initially an empty list.)

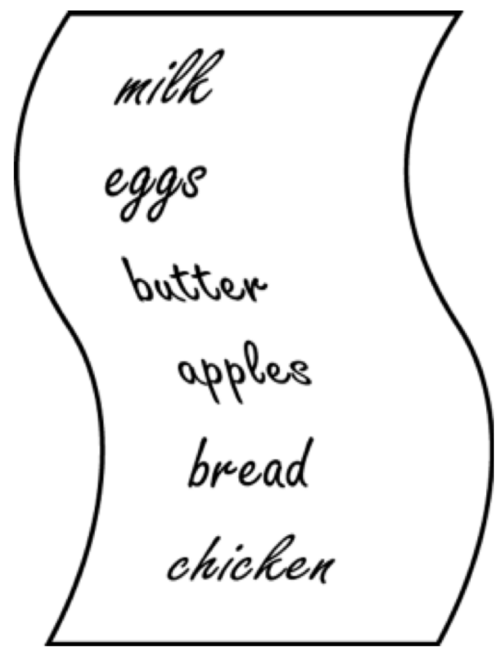
[]

- You can add items to the list.
    - The default behavior is to add to the end of the list.

    [milk, eggs, butter, apples, bread, chicken, meat]

- The list object keeps track of the element values that have been added to it, their order, indexes, and its total size.

- For example a shopping list!



ADT List operations

`add(`**value**`)`	appends value at the end of list
`add(`**index**`,` **value**`)`	inserts given value just before the given index, shifting subsequent values to the right
`clear()`	removes all elements of the list
`indexOf(`**value**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**index**`)`	returns the value at given index
`remove(`**index**`)`	removes/returns value at given index, shifting subsequent values to the left
`set(`**index**`,` **value**`)`	replaces value at given index with given value
`size()`	returns the number of elements in list
`contains(`**value**`)`	returns true if given value is found somewhere in this list

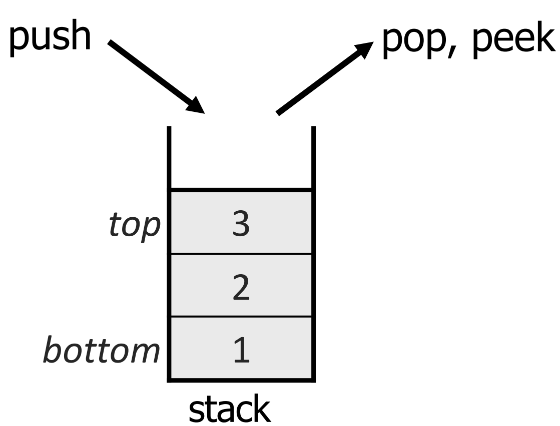
ADT Sorted List

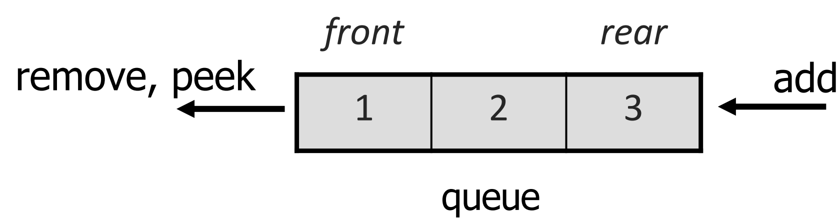
- A List which
  - Maintains items in sorted order
  - Inserts and deletes items by their values, not their positions

`add(`**value**`)`	insert value to the correct position
`clear()`	removes all elements of the list
`indexOf(`**value**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**index**`)`	returns the value at given index
`remove(`**value**`)`	removes/returns value (if exists), shifting subsequent values to the left
`size()`	returns the number of elements in list
`contains(`**value**`)`	returns true if given value is found somewhere in this list

Stack and Queue

- Sometimes it is good to have a collection that is less powerful, but is optimized to perform certain operations very quickly.
- Two specialty collections:
  - **stack:** Retrieves elements in the reverse of the order they were added.
  - **queue:** Retrieves elements in the same order they were added.

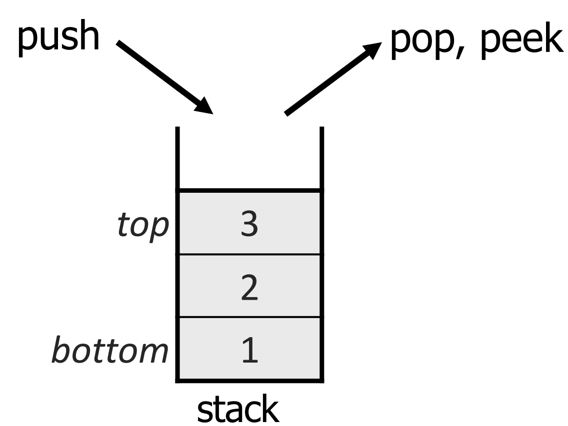




ADT Stack

- The elements are stored in order of insertion, but we do not think of them as having indexes.
- Insertion and deletion only examine the last element added

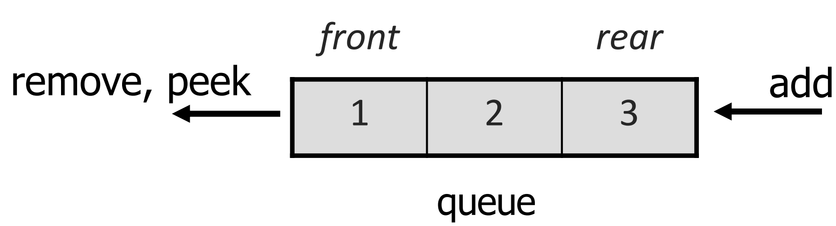
`push(`**value**`)`	places given value on top of stack
`pop()`	removes top value from stack and returns it
`peek()`	returns top value from stack without removing it
`size()`	returns number of elements in stack
`isEmpty()`	returns `true` if stack has no elements



ADT Queue

- The elements are stored in order of insertion, but we do not think of them as having indexes.
- Insertion only add to the end of queue and deletion only examine at the front of queue

`add(`**value**`)`	places given value at back of queue
`remove()`	removes value from front of queue and returns it
`peek()`	returns front value from queue without removing it
`size()`	returns number of elements in queue
`isEmpty()`	returns `true` if queue has no elements



Why use ADTs?

- We just need to understand the idea of the collection and what operations it can perform.
- The details of implementation is hidden
- We can have different kind of implementations
- Why would we want more than one kind (different implementations) of List, Queue, etc.?
  - Each implementation is more efficient at certain tasks.
  - You choose the optimal implementation for your task, and if the rest of your code is written to use the ADT interfaces, it will work.