Data structures: motivation

- The choice of a suitable data structure can make all the difference between an efficient and a failing program
- The input and output of any algorithm is stored inside a data structure
- Data structures organize data for quick and efficient access

Abstract Data Types and Data structures

[Review 02.01]

by Marina Barsky

Abstraction

Definition

- *Abstraction* - the process of extracting only **essential property** from a real-life entity
- In CS: Problem → storage + operations

↓

Abstract Data Type (**ADT**):

result of the process of abstraction

- A specification of ***data to be stored*** together with a set of ***operations*** on that data
- ADT = Data + Operations

ADT is a mathematical concept (from *theory of concepts*)

ADT is a language-agnostic concept

- Different languages support ADT in different ways
- In C++ or Java, use *class* construct to create a new ADT

ADT includes:

- **Specification:**
  - What needs to be stored
  - What operations are supported
- **Implementation:**
  - Data structures and algorithms used to meet the specification

ADT: Specification vs. implementation

Specification and **implementation** have to be disjoint:

- **One** specification
- **One or more** implementations
  - **Using different data structures**
  - **Using different algorithms**

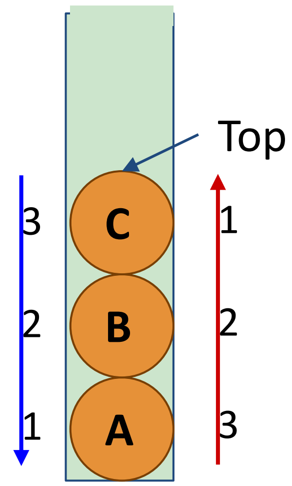
[Example 1: Abstraction for HR roster]

We want to model a list of company employees

- When the company grows - we should be able to add a new employee
- When the company downsizes we should be able to remove the last-added employee (seniority principle)

[Abstraction of HR roster: *Stack*]

- If these are the only important requirements to the HR roster, then we can solve this problem using ***Stack*** Abstract Data Type
- Stack stores a list of elements and allows only 2 operations: **adding a new element on top** of the stack and **removing the element from the top** of the stack
- Thus, the elements are sorted by the time stamp - from recent to older
- Stack is also called a **LIFO** queue (**L**ast **I**n - **F**irst **O**ut)



Specification of Stack ADT

Stack: Abstract Data Type which supports following operations:

- *Push(e)*: adds element to collection
- *Top()*: returns most recently-added element
- *Pop()*: removes and returns most recently-added element
- Boolean *IsEmpty()*: are there any elements?
- Boolean *IsFull()*: is there any space left?

[Stack ADT: possible Data Structures]

	Array Impl.	Link. List Impl.
Push(e)	O(1)	O(1)
Top()	O(1)	O(1)
Pop()	O(1)	O(1)
IsEmpty()	O(1)	O(1)
IsFull()	O(1)	O(1)

Considerations: Linked Lists have storage overhead
Arrays need to be resized when full

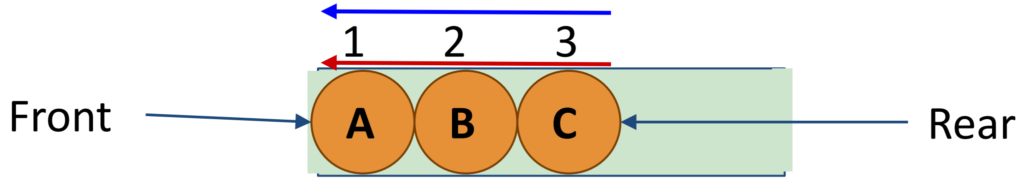
[Example 2: Abstraction of ER Queue]

We want to model a list of patients waiting in the Hospital ER

- When a new patient arrives - we should be able to add him to the queue
- When the doctor calls for the next patient, we should be able to remove the patient from the front of the queue

[Abstraction of ER Queue: *Queue*]

- If these are the only two required operations, then we can model the ER queue using a ***Queue*** **ADT**
- As in the Stack ADT, the elements in the Queue are also sorted by timestamp, but in a different order: from the earlier to the later
- This ADT is called a *FIFO Queue* (First In First Out)



Specification of Queue ADT

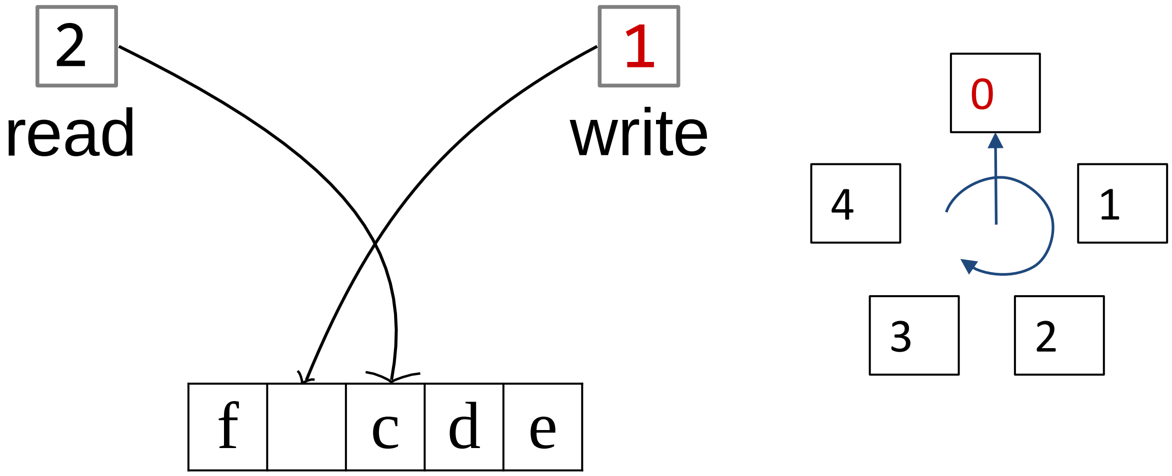
Queue: Abstract Data Type which supports the following operations:

- ***Enqueue****(e)*: adds element *e* to collection
- ***Dequeue****()*: removes and returns least recently-added key
- Boolean ***IsEmpty****( )*: are there any elements?
- Boolean ***IsFull****()*: is there any space left?

[Queue Implementation with Linked List]

- Augment Linked List with the *tail* pointer
- For `Enqueue(e)` use `list.add(e)` - which adds an element at the end
- For `Dequeue()` use `list.remove(list.head)`
- For `IsEmpty()` use (`list.head` = NULL?)

[Queue implementation with Circular Array]



Enqueue(g)

[Queue ADT: possible Data Structures]

	Link. List Impl. with tail	Array Impl. circular
Enqueue (e)	O(1)	O(1)
Dequeue()	O(1)	O(1)
IsEmpty()	O(1)	O(1)

Considerations:

Linked Lists have unlimited storage

Arrays need to be resized when full

Linked Lists have simpler maintenance

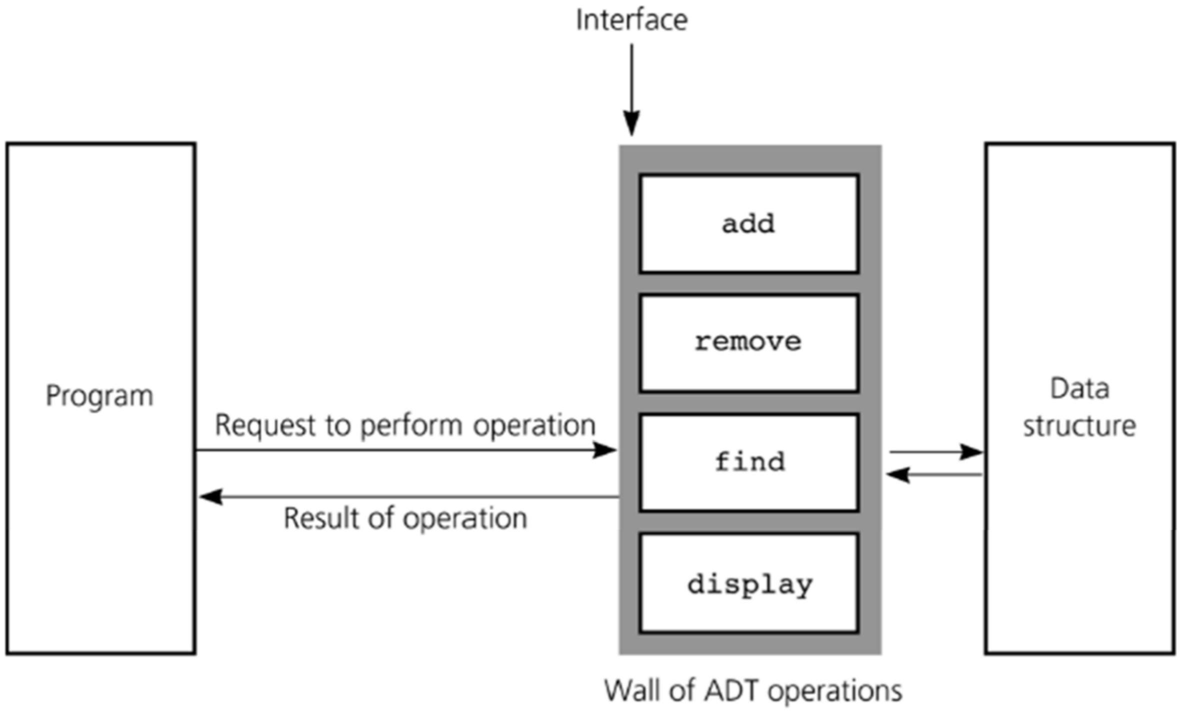
Hide implementation details from users of ADT

Users of ADT:

- Aware of the **specification** **only**
  - Usage only based on the specified operations
- Do not care / need not know about the actual ~~**implementation**~~
  - i.e. Different implementations do **not** affect the users of ADT

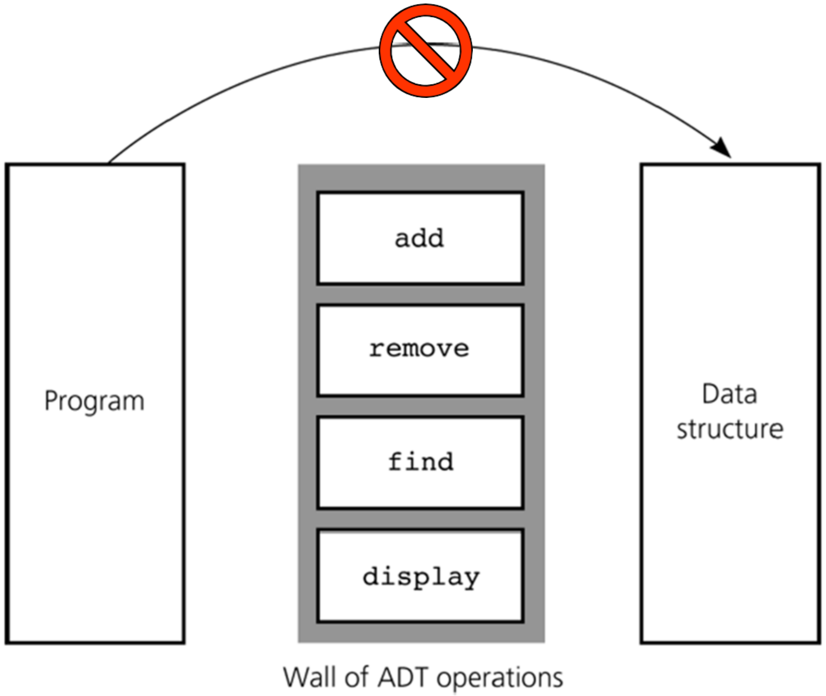
A Wall of ADT

- ADT operations provide:
  - Interface to data structures
  - Secure access

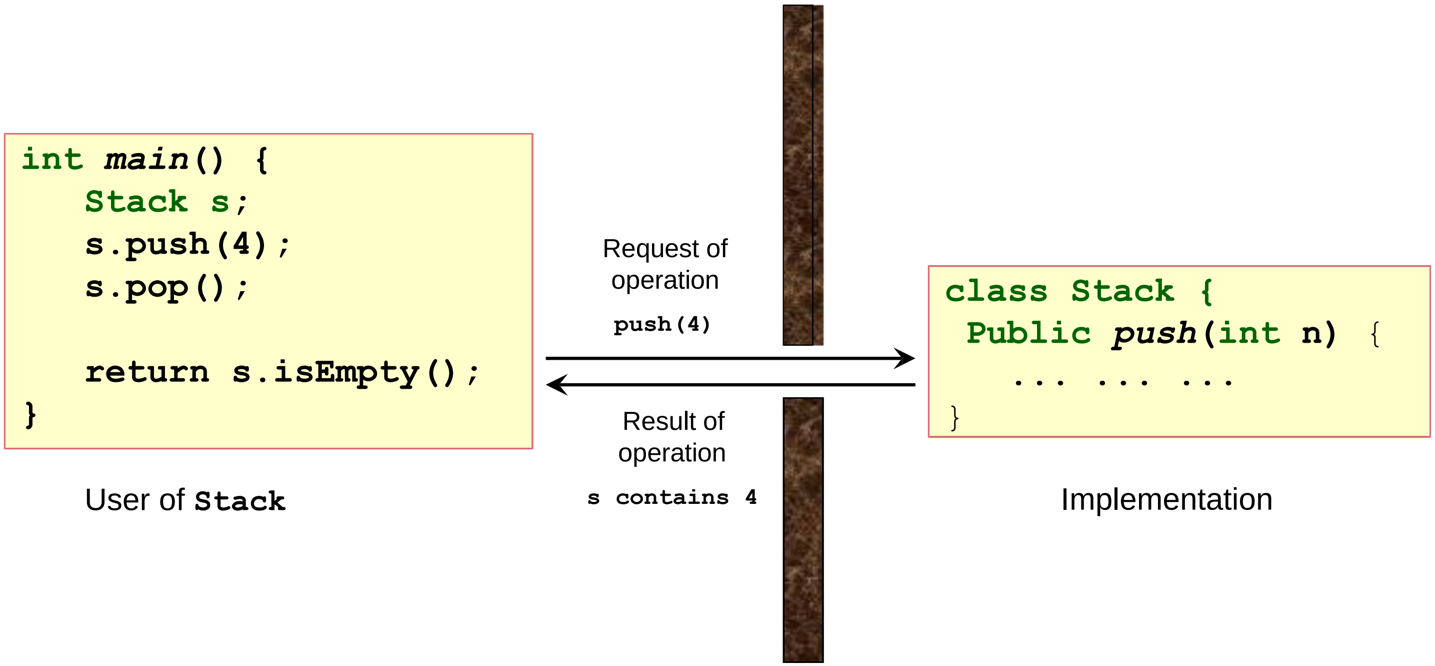


Impenetrable wall

- User programs **should not**:
  - Use the underlying data structure directly
  - Depend on implementation details



Specification as slit in the wall



- User only depends on specifications:
  - Function name, parameter types, and return type

Advantages of ADT

- Hide the implementation details by building walls around the data and operations
  - So that changes in either will not affect other program components that use them
- Functionalities are less likely to change
- Localize rather than globalize changes
- Help manage software complexity

Activity 5. Algorithm design ideas

In preparation for Assignment 2

Problem 1

Balanced brackets

Input: A string str consisting of '(', ')', '[', ']','{', '}' characters.

Output: Return whether or not the string's parentheses and brackets are balanced.

Examples

Balanced:

“([])[]()”,

“((([([])]))())”

Unbalanced:

“(]()”

“][”

“([)]”

Problem 2

Maintaining max

Input: A list of numbers stored in the Stack which supports the usual push and pop operations in time O(1)

Output: Max value currently in the Stack in time O(1)