CSCI 136
Data Structures & Advanced Programming

Lecture 16

Spring 2018

Profs Bill & Jon

Announcements

- Mid-Term Review Session
  - Tonight (3/12), 7:00-8:00 pm in TPL 203
  - No prepared remarks, so bring questions!
- Modified (extra) office hours (see calendar)
- Mid-term exam is Wednesday, March 14
  - During your normal lab session
  - You'll have 1 hour & 45 minutes (if you come on time!)
  - Closed-book
  - Covers Chapters 1-7 & 9 and all topics up through sorting
  - A "sample" mid-term and study sheet are available online
    - See Handouts & Problem Sets

Last Time

- Sorting Wrap-Up (Merge and Quick)
- Problem Solving Day

Today

- Linear Structures
  - The Linear Interface (LIFO & FIFO)
  - The AbstractLinear and AbstractStack classes
- Stack Implementations
  - StackArray, StackVector, StackList,
- Stack applications
  - Expression Evaluation
  - PostScript: Page Description & Programming

Linear Structures

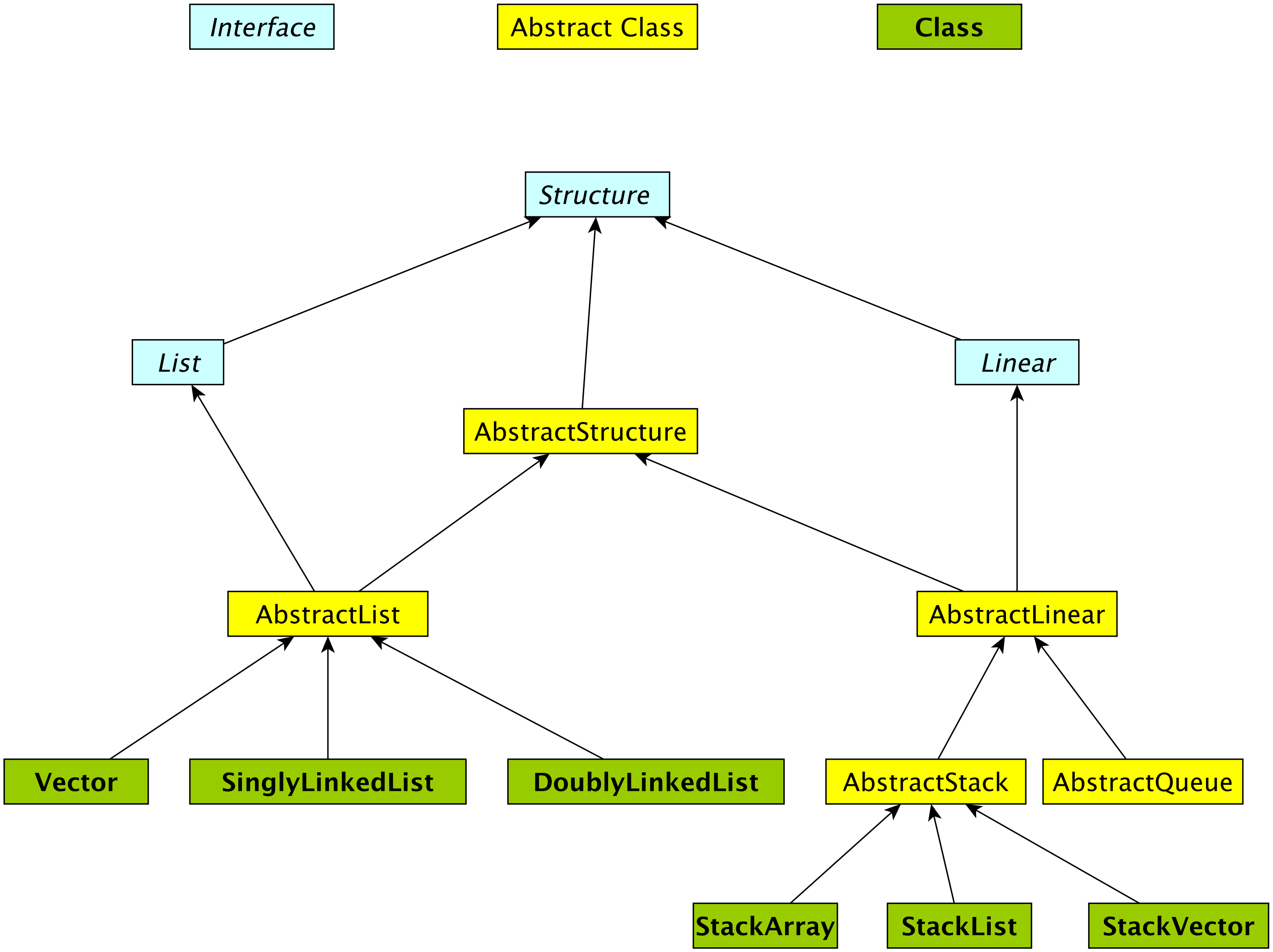
- What if we want to impose *access restrictions* on our lists?
  - I.e., we only provide one way to add and remove elements from list
    - No longer provide access to middle list elements
- Key Examples: removal order depends on the order that elements were added
  - LIFO: Last In First Out
  - FIFO: First In First Out

Examples

- FIFO: First In – First Out (Queue)
  - Line at dining hall
  - Data packets arriving at a router

- LIFO: Last In – First Out (Stack)
  - Pile of trays at dining hall
  - Java Virtual Machine stack

The Structure5 Universe (next)



Interface

Abstract Class

Class

Structure

List

AbstractStructure

Linear

AbstractList

AbstractLinear

Vector

SinglyLinkedList

DoublyLinkedList

AbstractStack

AbstractQueue

StackArray

StackList

StackVector

Linear Interface

- How should `Linear` interface differ from `List`?
  - Should have fewer methods than `List` interface since we are limiting access …
- Methods:
  - Inherits all of the `Structure` interface methods
    - `add(E value)` – Add `value` to the structure.
    - `E remove(E o)` – Remove value o from the structure.
    - `size()`, `isEmpty()`, `clear()`, `contains(E val)`, …
  - Adds
    - `E get()` – Preview the *next* object to be removed.
    - `E remove()` – Remove the *next* value from the structure.
    - `boolean empty()` – same as `isEmpty()`

Linear Structures

- Why no “random access”?
  - I.e., no access to middle of list
- More restrictive than general List structures
  - But less functionality can result in:
    - Simpler implementation
    - Greater efficiency
- Approaches
  - Use existing structures (`Vector`, `LinkedList`), or
  - Use same underlying organization, but simplified

Stacks

- Examples: pile of trays or cups
  - Can only take tray/cup from top of pile
- What methods do we need to define?
  - Stack interface methods
- New terms: `push`, `pop`, `peek`
  - Only use `push`, `pop`, `peek` when talking about stacks
  - `push` = add to top of stack
  - `pop` = remove from top of stack
  - `peek` = look at top of stack (do not remove)

Notes about Terminology

- When using stacks:
  - push = add
  - pop = remove
  - peek = get
- In Stack interface, push/pop/peek methods call add/remove/get methods that are defined in Linear interface
- But "add" is not mentioned in Stack interface (it is inherited from Linear)
- Stack interface *extends* Linear interface
  - Interfaces *extend* other interfaces
  - Classes *implement* interfaces

Stack Implementations

- Array-based stack
  - int top, Object data[ ]
  - Add/remove from index top

+ all operations are O(1)

– wasted/run out of space

- Vector-based stack
  - Vector data
  - Add/remove from tail

+/– most ops are O(1) (add is O(n) in worst case)

– potentially wasted space

- List-based stack
  - SLL data
  - Add/remove from *head*

+ all operations are O(1)

+/– O(n) space overhead (no "wasted" space)

Stack Implementations

- structure5.StackArray
  - int top, Object data[ ]
  - Add/remove from index top

+ all operations are O(1)

– wasted/run out of space

- structure5.StackVector
  - Vector data
  - Add/remove from tail

+/– most ops are O(1) (add is O(n) in worst case)

– potentially wasted space

- structure5.StackList
  - SLL data
  - Add/remove from head

+ all operations are O(1)

+/– O(n) space overhead (no "wasted" space)

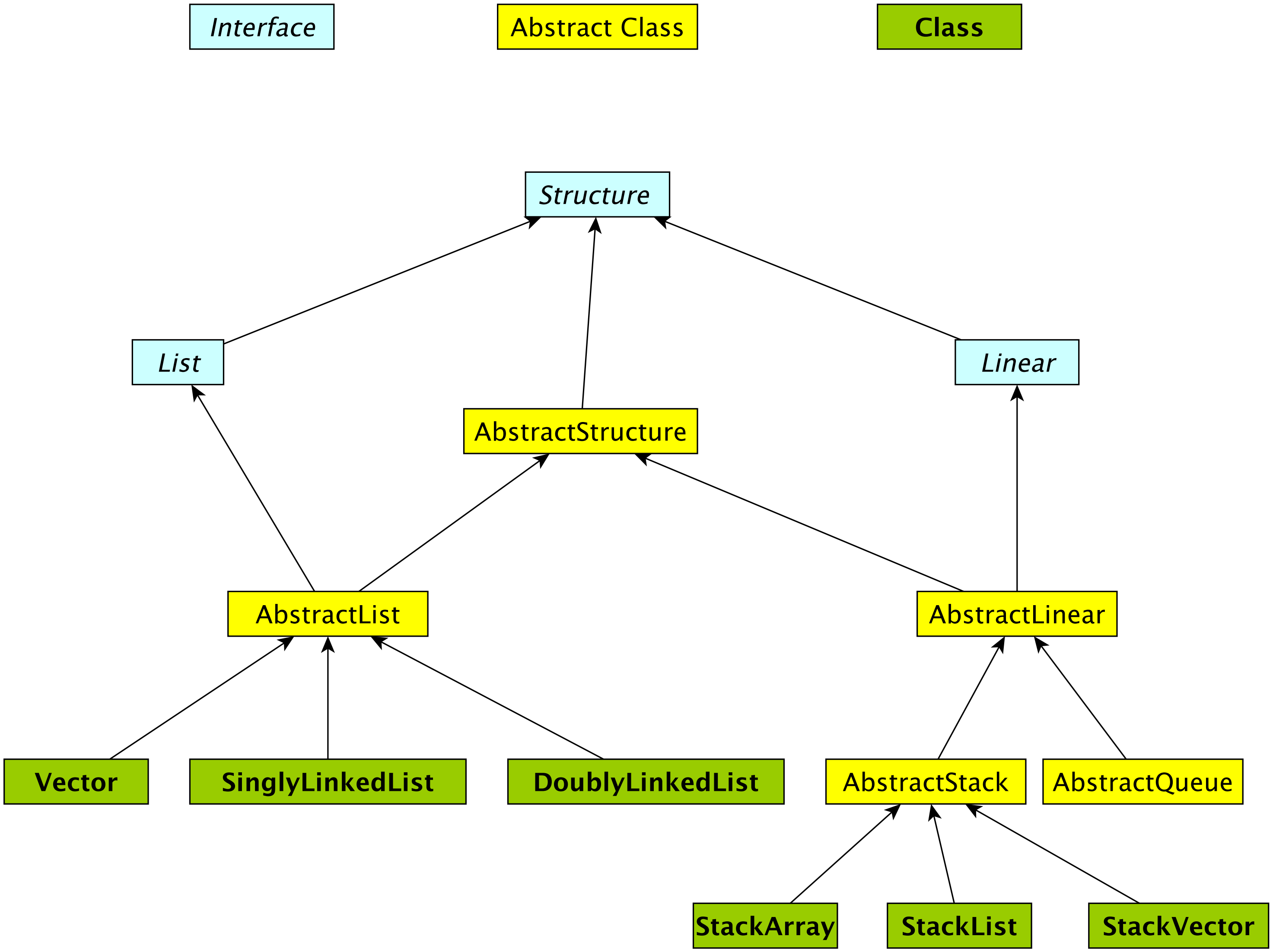
Summary Notes on The Hierarchy

- `Linear` interface *extends* `Structure`
  - `add(E val)`
  - `empty()`
  - `get()`
  - `remove(),`
  - `size()`
- `AbstractLinear` (partially) *implements* `Linear`
- `AbstractStack` class (partially) *extends* `AbstractLinear`
  - Essentially introduces "stack-ish" names for methods
    - `push(E val)` is `add(E val)`
    - `pop()` is `remove()`
    - `peek()` is `get()`

Building The Hierarchy

- Now we can extend `AbstractStack` to make "concrete" Stack types
  - `StackArray<E>`
    - holds an array of type E
    - add/remove at high end
  - `StackVector<E>`
    - Similar to StackArray<E>, but with a vector for dynamic growth
  - `StackList<E>`
    - A singly-linked list with add/remove at `head`
  - For each, we implement `add`, `empty`, `get`, `remove`, `size` directly
    - `push`, `pop`, `peek` are indirectly implemented by abstract class

The Structure5 Universe (so far)



Interface

Abstract Class

Class

Structure

List

Linear

AbstractStructure

AbstractList

AbstractLinear

Vector

SinglyLinkedList

DoublyLinkedList

AbstractStack

AbstractQueue

StackArray

StackList

StackVector

Stack Applications

- The `Stack` implementation is simple, but there are *many* applictaions
  - Evaluating mathematical expressions
  - Searching (Depth-first search)
  - Removing recursion for optimization
  - Simulations
  - …

Evaluating Arithmetic Expressions

- Computer programs regularly use stacks to evaluate arithmetic expressions
- Example: x*y+z
  - First rewrite as xy*z+
    - *we'll look at this rewriting process in more detail soon*
  - Then:
    - push x
    - push y
    - * (*pop twice, multiply popped items, push result*)
    - push z
    - + (*pop twice, add popped items, push result*)

Converting Expressions

- We (humans) primarily use infix notation to evaluate expressions
  - (x+y)*z
- Computers traditionally used postfix (also called Reverse Polish) notation
  - xy+z*
  - Operators appear after operands, parentheses are not necessary
- How do we convert between the two?
  - Compilers do this for us