CSCI 136
Data Structures &
Advanced Programming

Williams College

Linear Structures

- General idea: we impose *access restrictions* on our data structure, disallowing add/remove/access at arbitrary indices
  - **No** `get(int i)`, `set(int i, E value)`
  - **No** `add(int i)`, `remove(int i)`

- Insight: By limiting access, we can actually gain some utility—linear structures are useful building blocks with important use cases!

Examples: Dining Hall

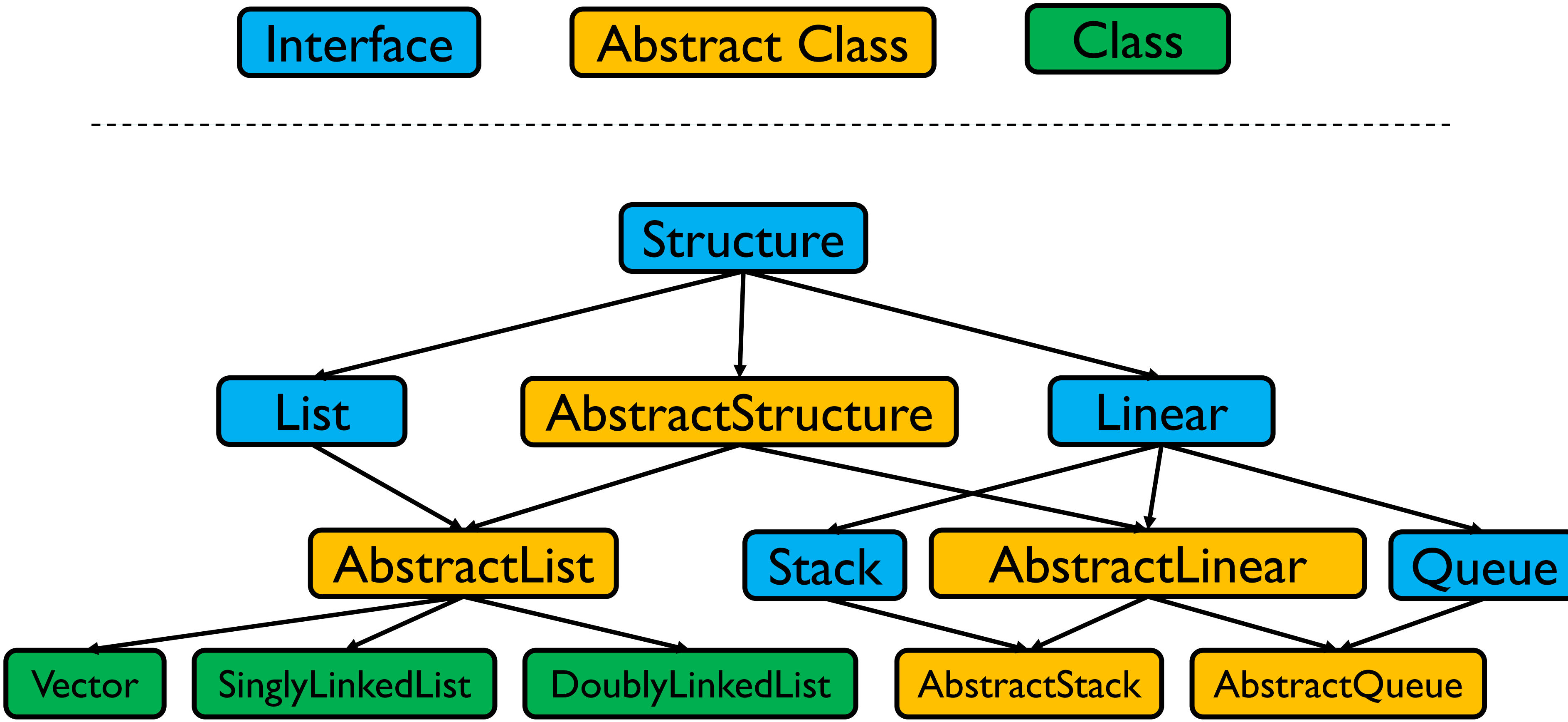
- FIFO: First In – First Out (Queue)
  - Line at dining hall

- LIFO: Last In – First Out (Stack)
  - Pile of plates or cups at dining hall

Examples: Computer Science

- FIFO: First In – First Out (Queue)
  - Data packets arriving at a router
- LIFO: Last In – First Out (Stack)
  - Java Virtual Machine stack

The Structure5 Universe (+ Linear!)



Quick Note about Terminology

- Note: `Stack` interface *extends* `Linear` interface
  - Interfaces *extend* other interfaces
  - Classes *implement* interfaces
- If you look at the structure5 documentation for Linear, you will see:
  - A list of superinterfaces
  - A list of subinterfaces
  - A list of implementing classes

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 new methods
    - `E get()` – Preview the *next* object to be removed.
    - `E remove()` – Remove the *next* value from the structure.
    - `boolean empty()` – same as `isEmpty()`

AbstractStack

- What methods do we *need* to define?
  - `Stack` interface methods
- `Stack` introduces new terms: `push`, `pop`, `peek`
  - Only use `push`, `pop`, `peek` when talking about stacks (not queues)
  - `push` = add to top of stack
  - `pop` = remove from top of stack
  - `peek` = look at top of stack (do not remove)

Linear Structure Philosophy

- Why no "random access"? (i.e., no access to middle of list)
  - Supporting/Providing less functionality can yield:
    - Simpler implementations of our algorithms
    - Greater algorithmic efficiency
- What should be our Data structure implementation approach?
  - Use existing structures (`Vector`, `LinkedList`), or
  - Reimplement "stripped down" versions of those structures (same underlying organization) simplified

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

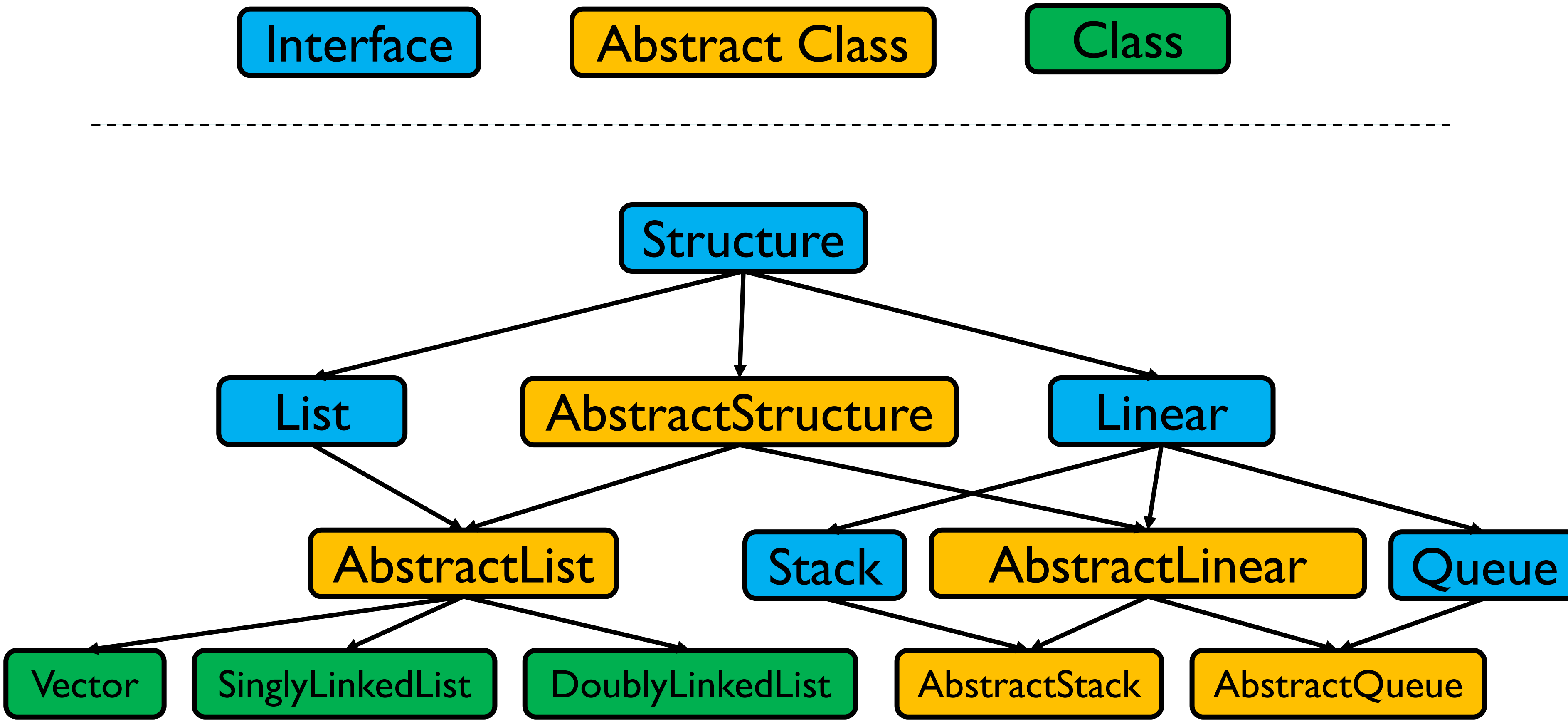
- `structure5.StackVector`
  - Vector data
  - Add/remove from tail

  +/– most ops are O(1) (add is O(n) in worst case but amortized O(1))
  – potentially wasted space

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

  + all operations are O(1)
  +/– O(n) space overhead (no "wasted" space)

The Structure5 Universe (+ Linear!)



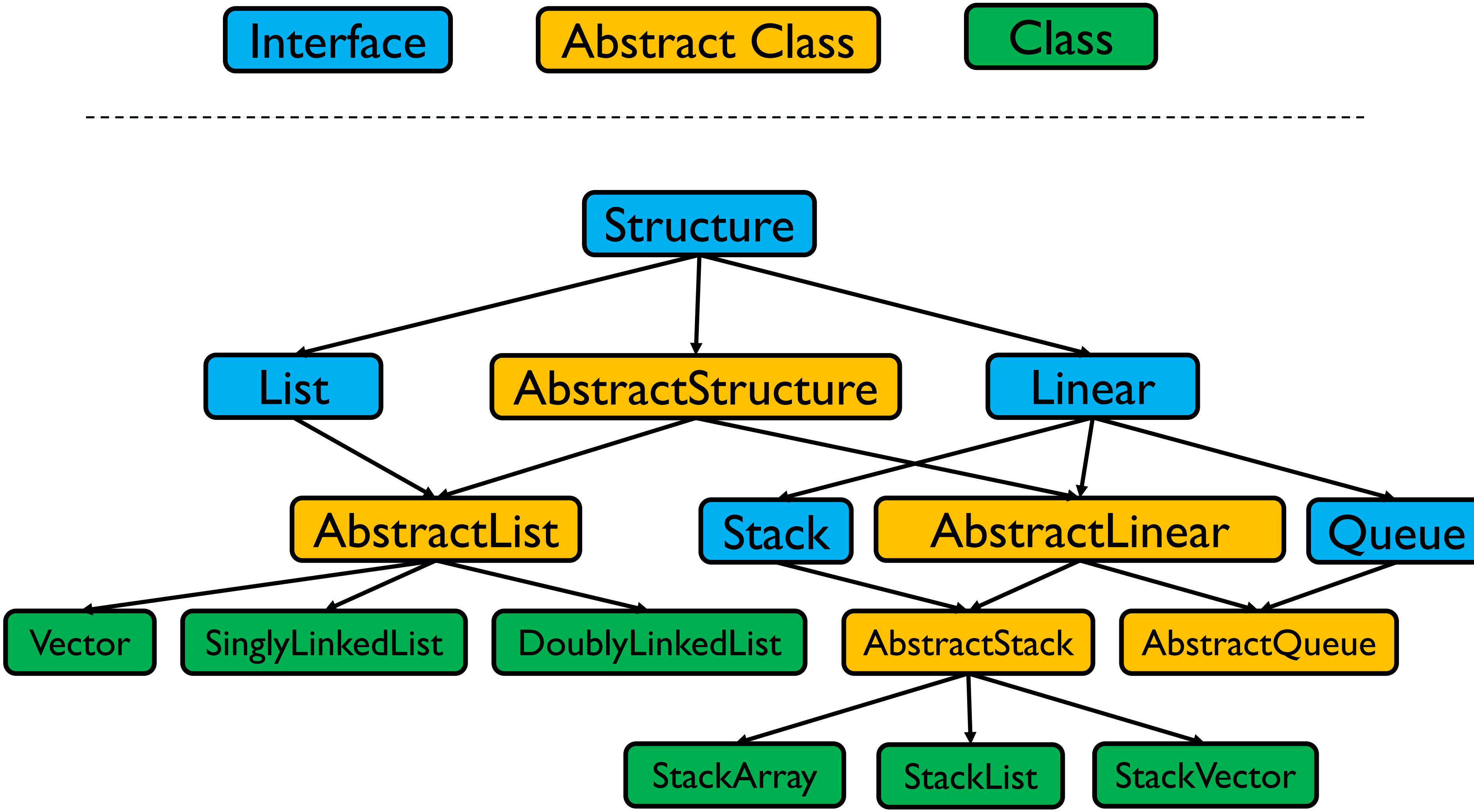
Summary Notes on The Hierarchy

- `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 linear methods
    - `push(E val)` is `add(E val)`
    - `pop()` is `remove()`
    - `peek()` is `get()`

Rounding Out The Hierarchy

- Rundown of classes that extend `AbstractStack`:
  - `StackArray<E>`
    - holds an array of type E
    - add/remove at high end
    - Can't add once the array fills
  - `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 (+ Stacks!)



CSCI 136
Data Structures &
Advanced Programming

Williams College

Video Goals

- Describe a few real-world problems
- Describe how to map one of those problems to the stack abstract data type
- Work through some examples to give us experience with (and appreciated of) stacks

Stack Applications

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

See textbook for details because this is VERY useful!

Evaluating Arithmetic Expressions

- Computer programs regularly use stacks to evaluate arithmetic expressions (as does the HP-12C calculator if you want to be a CFA…)
- 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)

Converting Expressions

- Example: x*y+z*w
- Conversion

1) Add full parentheses to preserve order of operations

   ((x*y)+(z*w))

2) Move all operators (+-*/) after operands

   ((xy*)(zw*)+)

3) Remove parentheses

   xy*zw*+

Use Stack to Evaluate Postfix Exp

- While there are input "tokens" (i.e., symbols) left:
  - Read the next token from input.
  - If the token is a value, push it onto the stack.
  - Else, the token is an operator that takes n arguments.
    (It is known that an operator takes n arguments by its definition.)
    - If there are fewer than n values on the stack → error.
    - Else, pop the top n values from the stack and:
      - Evaluate the operator, with the values as arguments.
      - Push the returned result, if any, back onto the stack.
  - The top value on the stack is the result of the calculation.
  - Note that results can be left on stack to be used in future computations:
    - Eg: 3 2 * 4 + followed by 5 / yields 2 on top of stack

Symbolic Example: Converting then Evaluating

- (x*y)+(z*w) → xy*zw*+
- Evaluate `xy*zw*+` :
  - Push x
  - Push y
  - Mult: Pop y, Pop x, Push x*y
  - Push z
  - Push w
  - Mult: Pop w, Pop z, Push z*w
  - Add: Pop x*y, Pop z*w, Push (x*y)+(z*w)
  - Result is now on top of stack

Concrete Example: Converting then Evaluating

- (x*y)+(z*w) → xy*zw*+
- Evaluate `xy*zw*+` :
  - Push x
  - Push y
  - Mult: Pop y, Pop x, Push x*y
  - Push z
  - Push w
  - Mult: Pop w, Pop z, Push z*w
  - Add: Pop x*y, Pop z*w, Push (x*y)+(z*w)
  - Result is now on top of stack
- Try with: w=3, x=4, y=5, z=6

PostScript

- PostScript is a programming language used for generating vector graphics
  - Best-known application: describing pages to printers
- It is a stack-based language
  - Values are put on stack
  - Operators pop values from stack, put result back on
  - There are numeric, logic, string values
  - Many operators
- Let's try it: The 'gs' command runs a PostScript interpreter….
- Implementing a tiny part of gs is something we will do in lab... it's a lot of fun!