CSCI 136
Data Structures &
Advanced Programming

Williams College

Linear Structures

- What if we want to impose *access restrictions* on our lists?

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

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

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

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

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

Examples

- FIFO: First In – First Out (Queue)

Examples

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

Examples

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

Examples

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

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

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

Linear Structures

- General idea: we impose *access restrictions* on our data structure, disallowing add/remove/access at arbitrary indices

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)`

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)

- LIFO: Last In – First Out (Stack)

Examples: Dining Hall

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

- LIFO: Last In – First Out (Stack)

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)

- LIFO: Last In – First Out (Stack)

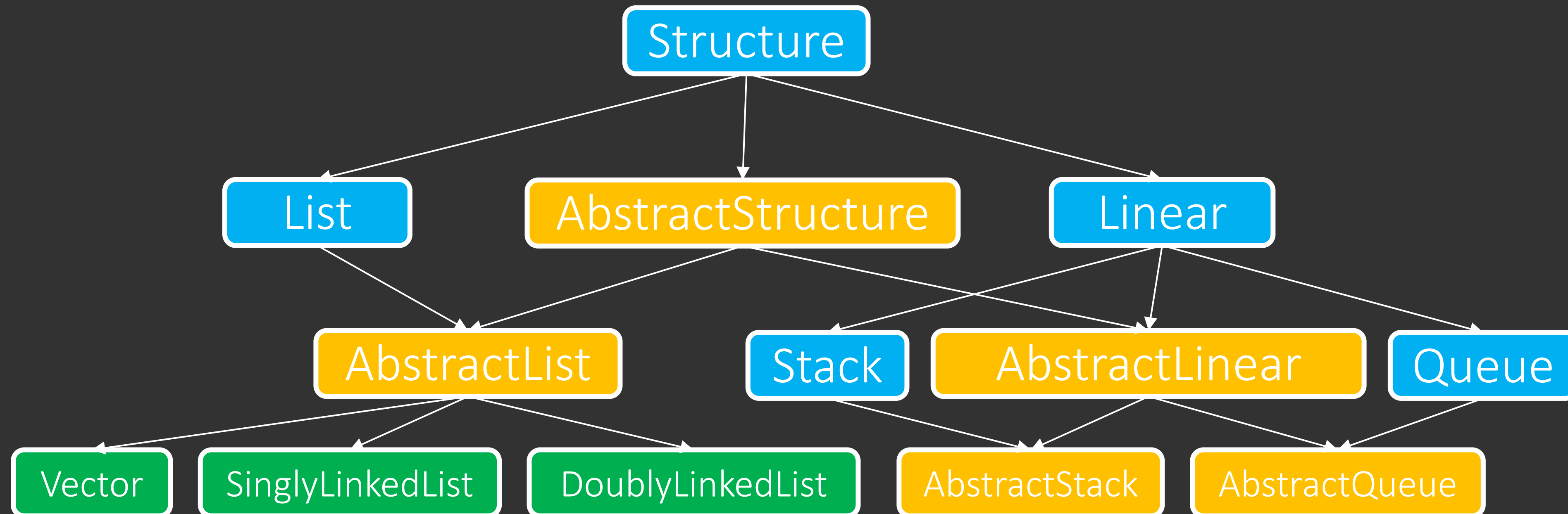
Examples: Computer Science

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

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`?

Linear Interface

- How should `Linear` interface differ from `List`?
  - Should have fewer methods than `List` interface since we are limiting access …

Linear Interface

- How should `Linear` interface differ from `List`?
  - Should have fewer methods than `List` interface since we are limiting access …
- Methods:

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)`, …

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?

AbstractStack

- What methods do we *need* to define?
  - `Stack` interface methods

AbstractStack

- What methods do we *need* to define?
  - `Stack` interface methods
- `Stack` introduces new terms: `push`, `pop`, `peek`

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)

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

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

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)

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

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?

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

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

Stack Implementations

- Array-based stack

Stack Implementations

- Array-based stack

- Vector-based stack

Stack Implementations

- Array-based stack
- Vector-based stack
- List-based stack

Stack Implementations

- Array-based stack
  - int top, Object data[ ]

- Vector-based stack

- List-based stack

Stack Implementations

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

- Vector-based stack

- List-based stack

Stack Implementations

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

Stack Implementations

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

Stack Implementations

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

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

- List-based stack
  - SLL data

Stack Implementations

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

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

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

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

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

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)
+/– O(n) space overhead

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

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)
+/– O(n) space overhead

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

+ all operations are O(1)
+/– O(n) space overhead

Stack Implementations

- `structure5.StackArray`
  - int top, Object data[ ]
  - Add/remove from index top
- `structure5.StackVector`
  - Vector data
  - Add/remove from tail
- `structure5.StackList`
  - SLL data
  - Add/remove from head

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

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)
+/– O(n) space overhead

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

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)
+/– O(n) space overhead

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

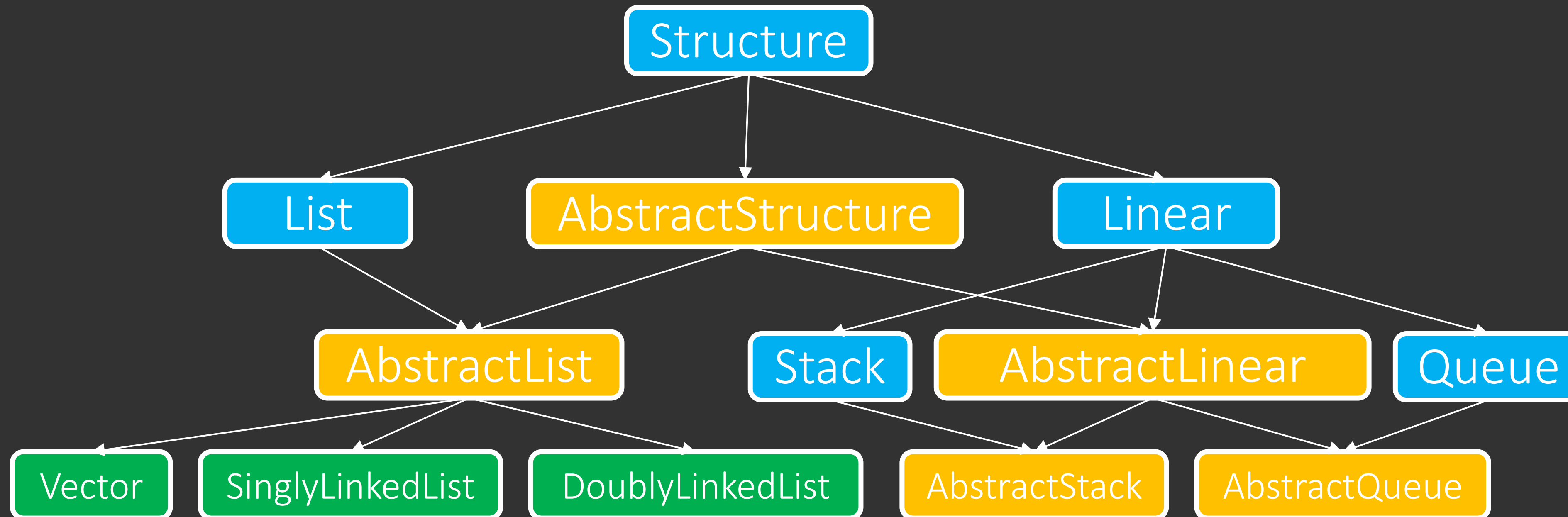
+ all operations are O(1)
+/– O(n) space overhead

The Structure5 Universe (+ Linear!)

Interface

Abstract Class

Class



Summary Notes on The Hierarchy

Summary Notes on The Hierarchy

- `Linear` interface *extends* `Structure`

Summary Notes on The Hierarchy

- `Linear` interface *extends* `Structure`
  - `add(E val)`
  - `empty()`
  - `get()`
  - `remove()`
  - `size()`

Summary Notes on The Hierarchy

- `Linear` interface *extends* `Structure`
  - `add(E val)`
  - `empty()`
  - `get()`
  - `remove()`
  - `size()`
- `AbstractLinear` (partially) *implements* `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`

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

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`:

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

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

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`

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!)

Interface

Abstract Class

Class