CSCI 136
Data Structures &
Advanced Programming

Lecture 13

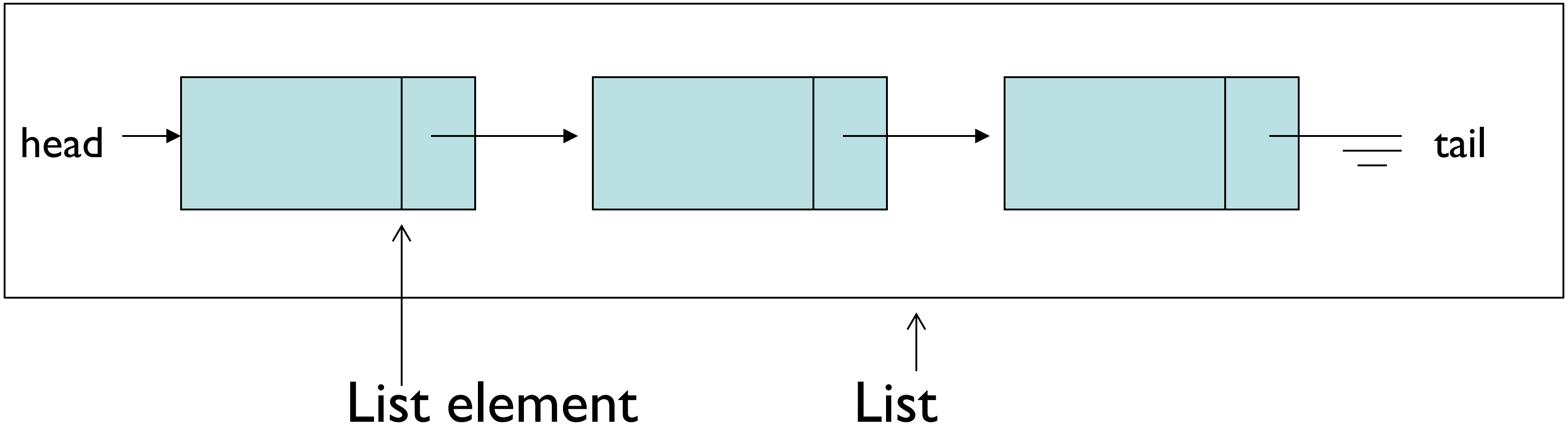
Fall 2019

Instructors: B&S

Linked List Basics

- There are two key aspects of Lists
  - Elements of the list
  - The list itself

- Visualizing lists

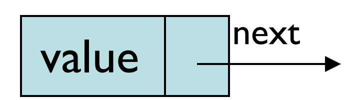


Linked List Basics

- List nodes are recursive data structures
  - (in a way)
- Each “node” has:
  - A data value
  - A “next” value that identifies the next element in the list
  - Can also have “previous” that identifies the previous element (“doubly-linked” lists)
- What methods does Node class need?

SinglyLinkedLists

- Terminology alert!
  - SinglyLinkedListNode = SLLN in these notes
  - SLLN = Node in structure5 (and in Ch 9)
  - Let's look at SLLN.java
  - How about SinglyLinkedList?
  - SinglyLinkedList = SLL in my notes



- What would addFirst(E d) look like?
- getFirst()?
- addLast(E d)? (more interesting)
- getLast()?

More SLL Methods

- How would we implement:
  - get(int index), set(E d, int index)
  - add(E d, int index), remove(int index)
- Left as an exercise:
  - contains(E d)
  - clear()
- Note: E is value type

Get and Set

```
public E get(int index) {
   Assert.pre(index < size() - 1, “Index out of range”);
   // or should we return null in above case?
   SLLN finger = head;
   for (int i=0; i<index; i++){
        finger = finger.next();
   }
   return finger.value();
}

public E set(E d, int index) {
   Assert.pre(index < size() - 1, “Index out of range”);
   // Same question!
   SLLN finger = head;
   for (int i=0; i<index; i++){
        finger = finger.next();
   }
   E old = finger.value();
   finger.setValue(d);
   return old;
}
```

Remove

```
public E remove(int index) {
   if(index >= size()) return null;

   E old;

   if (index == 0) return removeFirst();
   else if (index == size()-1) return removeLast();

   else {
        SLLN finger = head;
        for (int i=0; i<index - 1; i++) { //stop one before index
               finger = finger.next();
        }
        old = finger.next.value();
        finger.setNext(finger.next().next());
        count--;
        return old;
   }
}
```

Add

```
public void add(E d, int index) {
   if(index > size()) return null;
   E old;

   if (index==0) { addFirst(d); }

   else if (index==size()) { addLast(d); }

   else {
        SLLN finger = head;
        SLLN previous = null;
        for (int i=0; i<index; i++) {
             previous = finger;
             finger = finger.next();
        }
        SLLN elem = new SLLN(d, finger);
        previous.setNext(elem); // new "ith" item added after i-1
        count++;
   }
}
```

Linked Lists Summary

- Recursive data structures used for storing data
- More control over space use than Vectors
- Easy to add objects to front of list
- Components of SLL (SinglyLinkedList)
  - head, elementCount
- Components of SLLN (Node):
  - next, value

Vectors vs. SLL

- Compare performance of
  - size
  - addLast, removeLast, getLast
  - addFirst, removeFirst, getFirst
  - get(int index), set(E d, int index)
  - remove(int index)
  - contains(E d)
  - remove(E d)

SLL Summary

- SLLs provide methods for efficiently modifying front of list
  - Modifying tail/middle of list is not quite as efficient
- SLL runtimes are consistent
  - No hidden costs like Vector.ensureCapacity()
  - Avg and worst case are always the same
- Space usage
  - No empty slots like vectors
  - But keep extra reference for each value
    - overhead proportional to list length
      - (but this is constant and predictable)

Food for Thought: SLL Improvements to Tail Ops

- In addition to Node head and int elementCount, add Node tail reference to SLL
- Result
  - addLast and getLast are fast
  - removeLast is not improved
    - We need to know element before tail so we can reset tail pointer
- Side effects
  - We now have three cases to consider in method implementations: empty list, head == tail, head != tail
  - Think about addFirst(E d) and addLast(E d)

CircularlyLinkedLists

- Use *next* reference of last element to reference head of list
- Replace ***head*** reference with ***tail*** reference
- Access head of list via *tail.next*
- <u>ALL</u> operations on head are fast!
- addLast() is still fast
- Only modest additional complexity in implementation
- Can “cyclically reorder” list by changing *tail* node
- Question: What’s a circularly linked list of size 1?

DoublyLinkedLists

- Keep reference/links in **both** directions
  - previous and next
- DoublyLinkedListNode instance variables
  - DLLN next, DLLN prev, E value
- Space overhead is proportional to number of elements
- ALL operations on tail (including removeLast) are fast!
- Additional work in each list operation
  - Example: add(E d, int index)
  - Four cases to consider now: empty list, add to front, add to tail, add in middle

```java
public class DoublyLinkedNode<E>
{
	protected E data;
	protected DoublyLinkedNode<E> nextElement;
	protected DoublyLinkedNode<E> previousElement;

// Constructor inserts new node between existing nodes
public DoublyLinkedNode(E v,
		DoublyLinkedNode<E> next,
		DoublyLinkedNode<E> previous)
{
	data = v;
	nextElement = next;
	if (nextElement != null)  // point next back to me
		nextElement.previousElement = this;
	previousElement = previous;
	if (previousElement != null) // point previous to me
		previousElement.nextElement = this;
}
```

DoublyLinkedList Add Method

```
public void add(int i, E o) {
        Assert.pre((0 <= i) && (i <= size()),
                "Index in range.");
        if (i == 0) addFirst(o);
        else if (i == size()) addLast(o);
        else {
                // Find items before and after insert point
                DoublyLinkedNode<E> before = null;
                DoublyLinkedNode<E> after = head;
                // search for ith position
                while (i > 0) {
                   before = after;
                   after = after.next();
                   i--;
                }
        // before, after refer to items in slots i-1 and i
        // continued on next slide
```

DoublyLinkedList Add Method

```
        // Note: Still in "else" block!
        // before, after refer to items in slots i-1 and i

        // create new value to insert in correct position
        // Use DLN constructor that takes parameters
        // to set its next and previous instance variables
        DoublyLinkedNode<E> current =
              new DoublyLinkedNode<E>(o,after,before);

        count++; // adjust size
    }
}
```

```java
public E remove(E value) {
      DoublyLinkedNode<E> finger = head;
      while ( finger != null &&
               !finger.value().equals(value) )
            finger = finger.next();
      if (finger == null) return null;

      // fix next field of previous element
      if (finger.previous() != null)
            finger.previous().setNext(finger.next());
      else head = finger.next();

      // fix previous field of next element
      if (finger.next() != null)
            finger.next().setPrevious(finger.previous());
      else tail = finger.previous();
      count--;
      return finger.value();
}
```

Duane's Structure Hierarchy

The structure5 package has a hierarchical structure

- A collection of *interfaces* that describe---but do not implement---the functionality of one or more data structures
- A collection of *abstract classes* provide partial implementations of one or more data structures
    - To factor out common code or instance variables
- A collection of concrete (fully implemented) classes to provide full functionality of a data structure

AbstractList Superclass

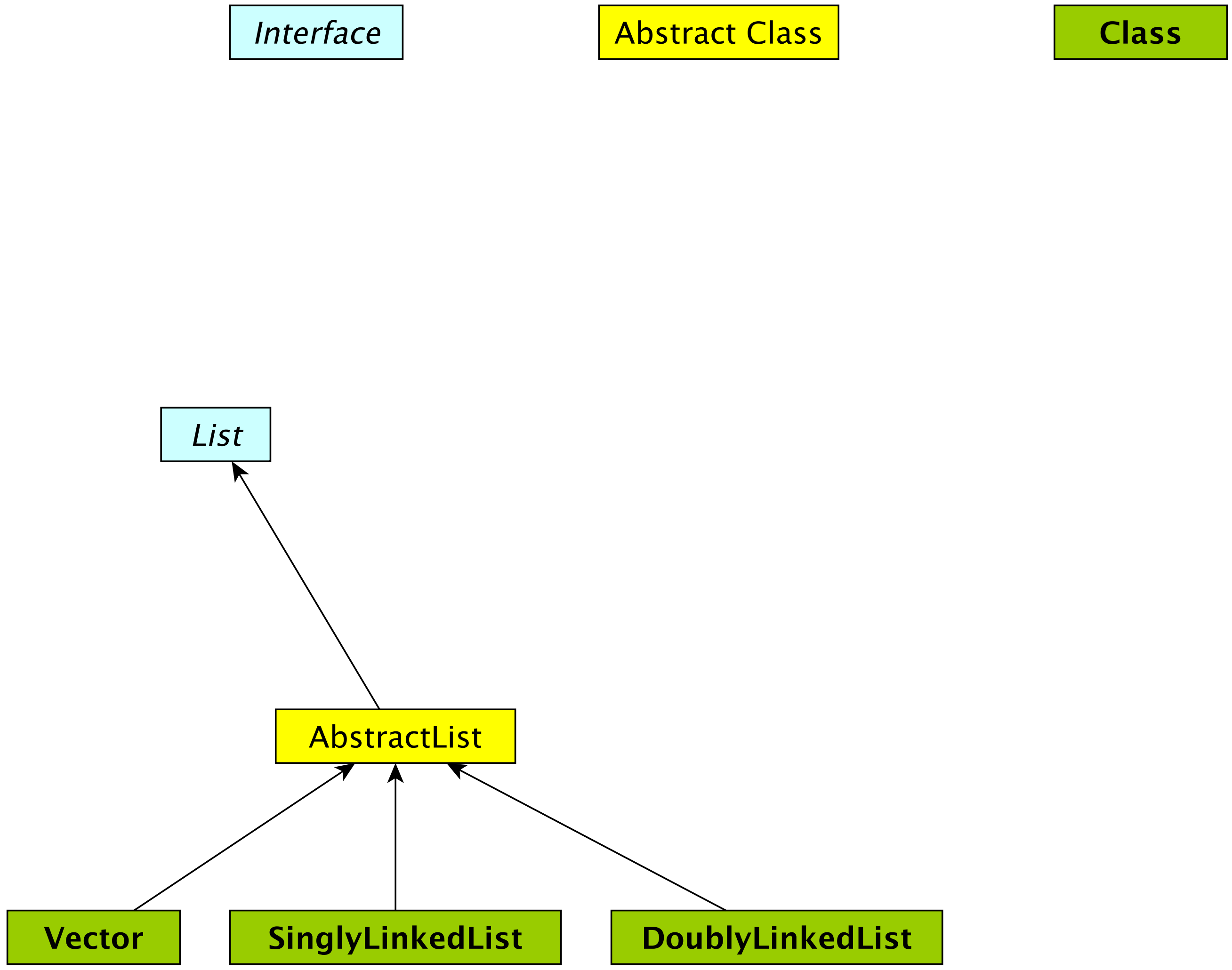
```
abstract class AbstractList<E> implements List<E> {
    public void addFirst(E element) { add(0, element); }
    public E getLast() { return get(size()-1);}
    public E removeLast() { return remove(size()-1); }
}
```

- AbstractList provides *some* of the list functionality
  - Code is shared among all sub-classes (see Ch. 7 for more info)

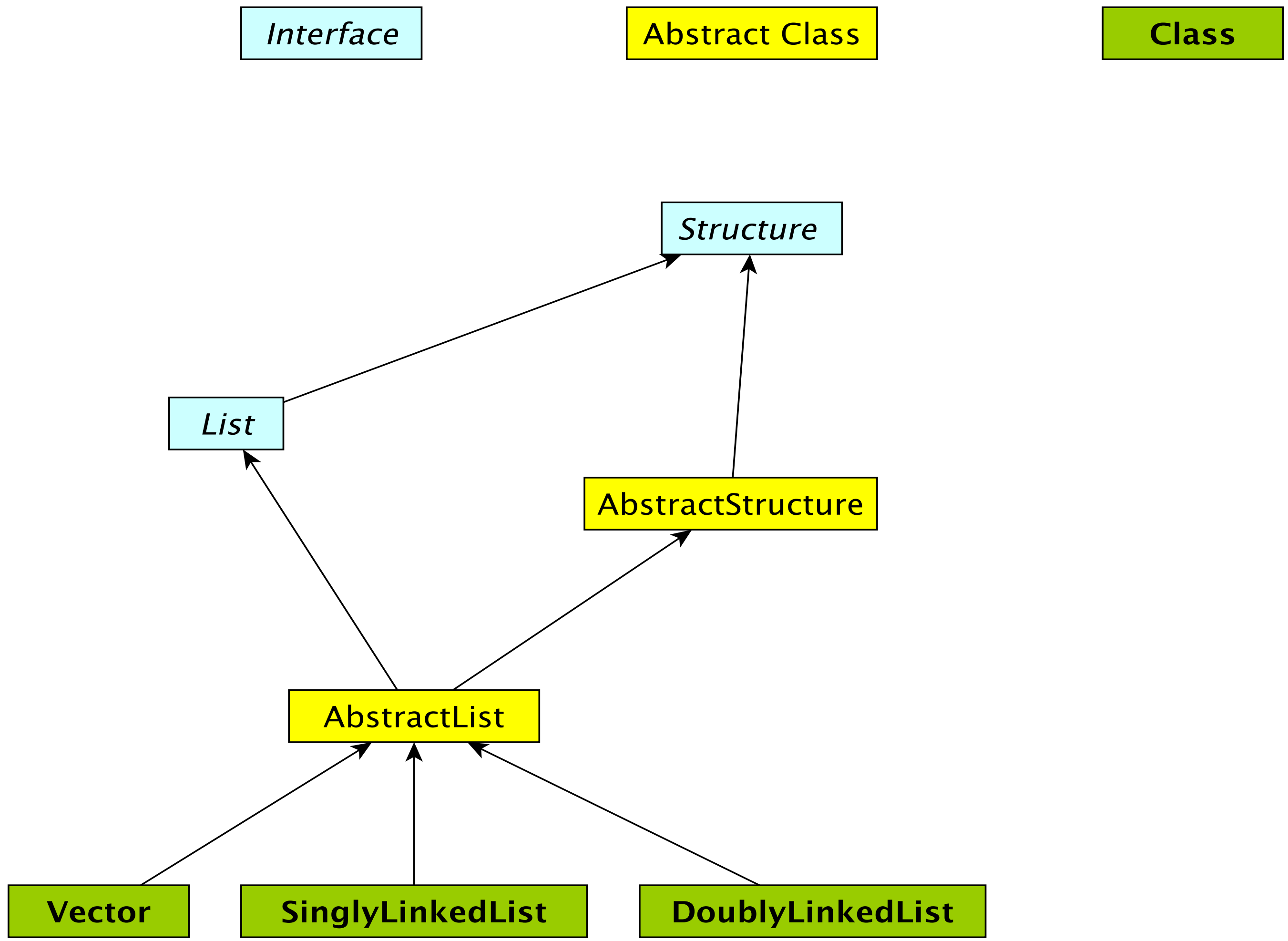
    ```
    public boolean isEmpty() { return size() == 0; }
    ```
  - Concrete classes (SLL, DLL) can override the code implemented in AbstractList
- Abstract classes in general do not implement every method
  - For example, size() is not defined although it is in the List interface
- Can't create an "AbstractList" directly
- Other lists extend AbstractList and implement missing functionality as needed

```
class Vector extends AbstractList {
    public int size() { return elementCount; }
}
```

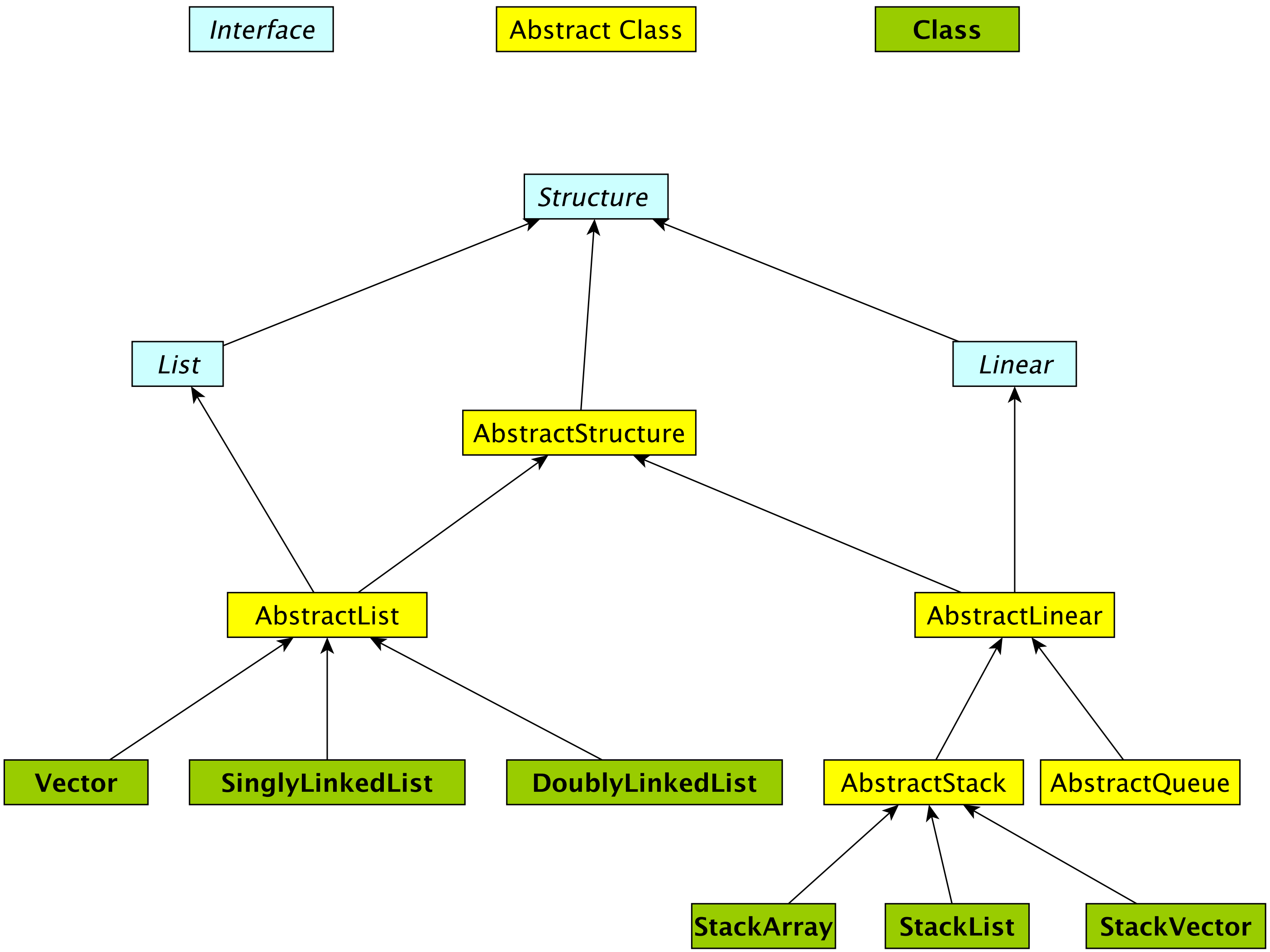
The Structure5 Universe (almost)



The Structure5 Universe (so far)



The Structure5 Universe (soon)



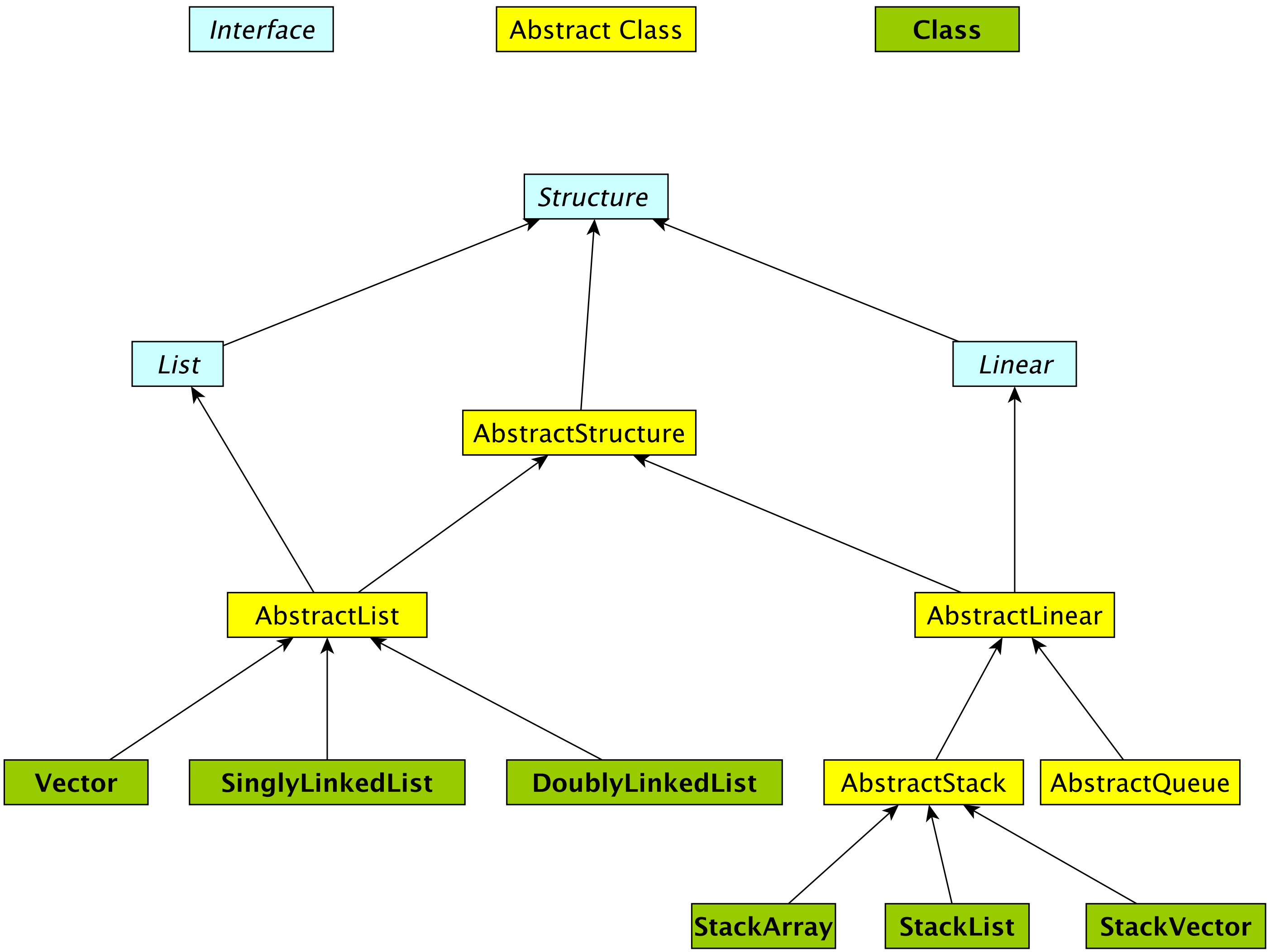
Linear Structures

- What if we want to impose *access restrictions* on our lists?
  - I.e., provide only one way to add and remove elements from list
  - No longer provide access to middle
- Key Examples: Order of removal depends on order 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)
  - Stack of trays at dining hall
  - Java Virtual Machine stack

The Structure5 Universe (next)



Linear Interface

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

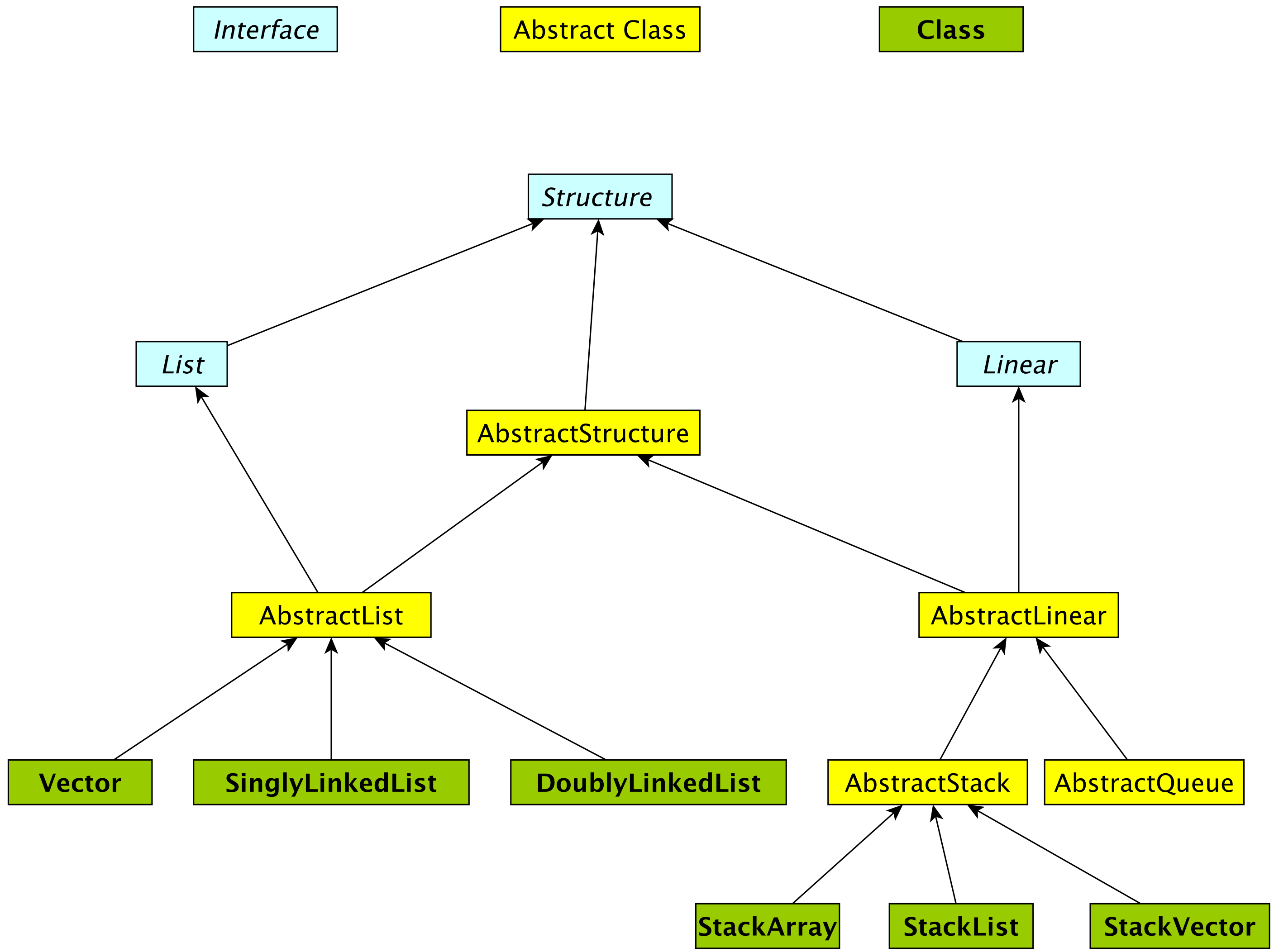
Linear Structures

- What if we want to impose *access restrictions* on our lists?
  - I.e., provide only one way to add and remove elements from list
  - No longer provide access to middle
- Key Examples: Order of removal depends on order 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)
  - Stack of trays at dining hall
  - Java Virtual Machine stack

The Structure5 Universe (next)



Linear Interface

- How should it differ from List interface?
  - Should have fewer methods than List interface since we are limiting access …
- Methods:
  - Inherits all of the Structure interface methods
    - add(E value) – Add a value to the structure.
    - E remove(E o) – Remove value o from the structure.
      - But this is awkward---why?
    - int size(), isEmpty(), clear(), contains(E value), …
  - 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
  - Less functionality can result in
    - Simpler implementation
    - Greater efficiency
- Approaches
  - Use existing structures (Vector, LL), or
  - Use underlying organization, but simplified

Stacks

- Examples: stack of trays or cups
  - Can only take tray/cup from top of stack
- 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:
  - pop = remove
  - push = add
  - peek = get
- In Stack interface, pop/push/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

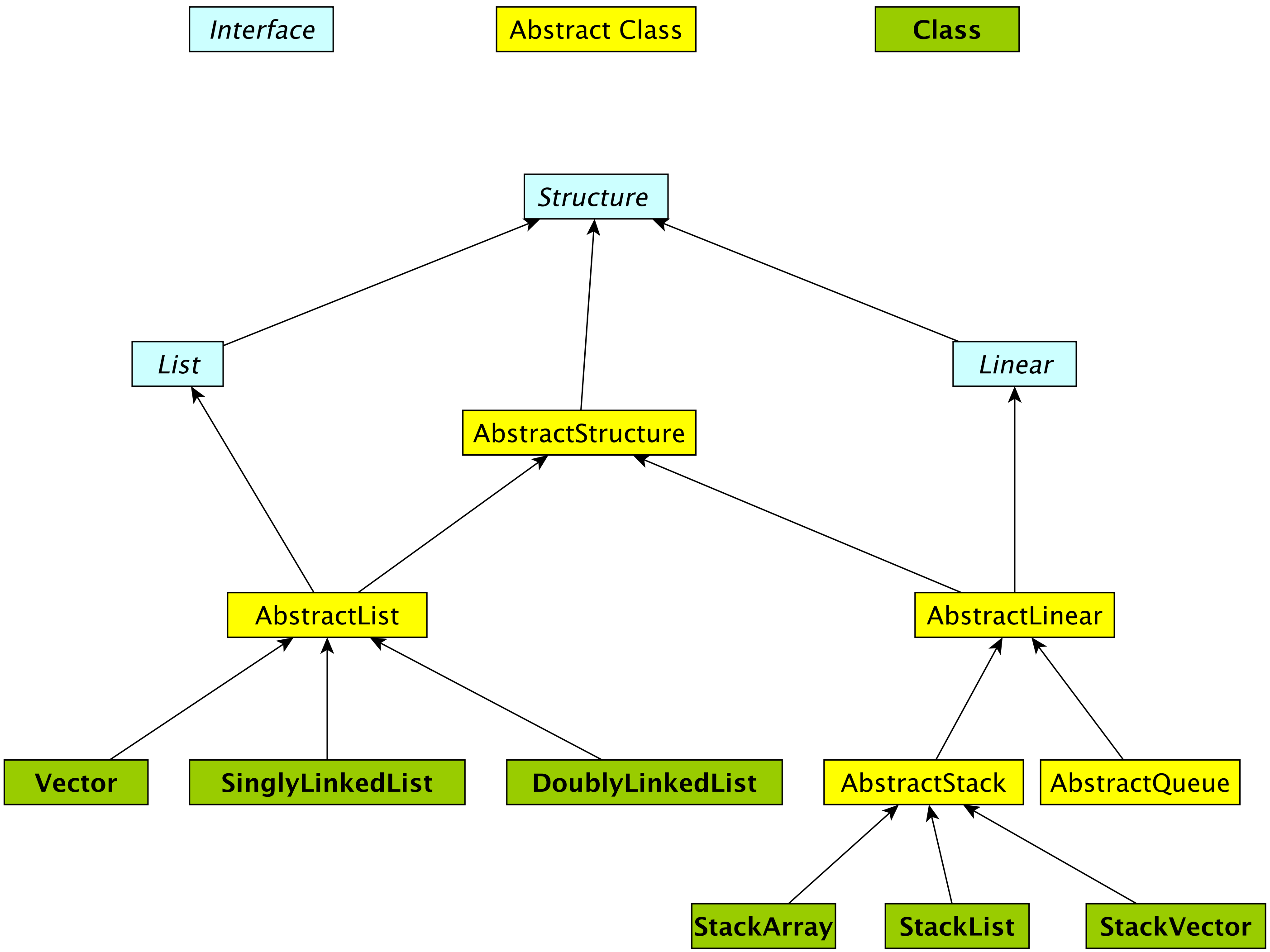
- structure5.StackList
  - SLL data
  - Add/remove from head

  + 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()
- 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, but with a vector for dynamic growth
  - StackList<E>: A singly-linked list with add/remove at head
  - We implement add, empty, get, remove, size directly
    - push, pop, peek are then indirectly implemented

The Structure5 Universe (so far)



Stack Applications

- Stack Implementation is simple, applications are many
  - 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 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 a priori that the operator takes n arguments.)
    - If there are fewer than n values on the stack $\rightarrow$ error.
    - Else, pop the top n values from the stack.
      - 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

Example

- $(x*y)+(z*w) \rightarrow xy*zw*+$
- Evaluate:
  - 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

Preview: 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….
- You'll be writing a (tiny part of) gs in lab soon....

Preview: PostScript

- Types: numeric, boolean, string, array, dictionary
- Operators: arithmetic, logical, graphic, …
- Procedures
- Variables: for objects and procedures
- PostScript is just as powerful as Java, Python, ...
  - Not as intuitive
  - Easy to automatically generate