CSCI 136
Data Structures & Advanced Programming

Lecture 12

Fall 2019

Instructors: B&S

Last Time

- Class extension
  - Abstract base classes
  - Concrete extension classes
- List: A general-purpose structure

Today

- Implementing Lists with linked structures
  - Singly Linked Lists
    - See Lecture 11 slides
  - Circularly & Doubly Linked Lists
- The structure5 hierarchy so far

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*
- ALL operations on head are still fast : O(1) time
- addLast() is now fast – O(1) time
- Only modest additional complexity in implementation
- Can “cyclically reorder” list by changing *tail* node
- Question: What’s a circularly linked list of size 1?
- Warning: `add(x)` adds `x` to *head of list* (not tail)….
  - Demand a refund!

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
- Warning: `add(x)` adds `x` to *head of list* (not tail)….
  - Demand a refund!

```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
       }
}
```

```
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); }
}
```

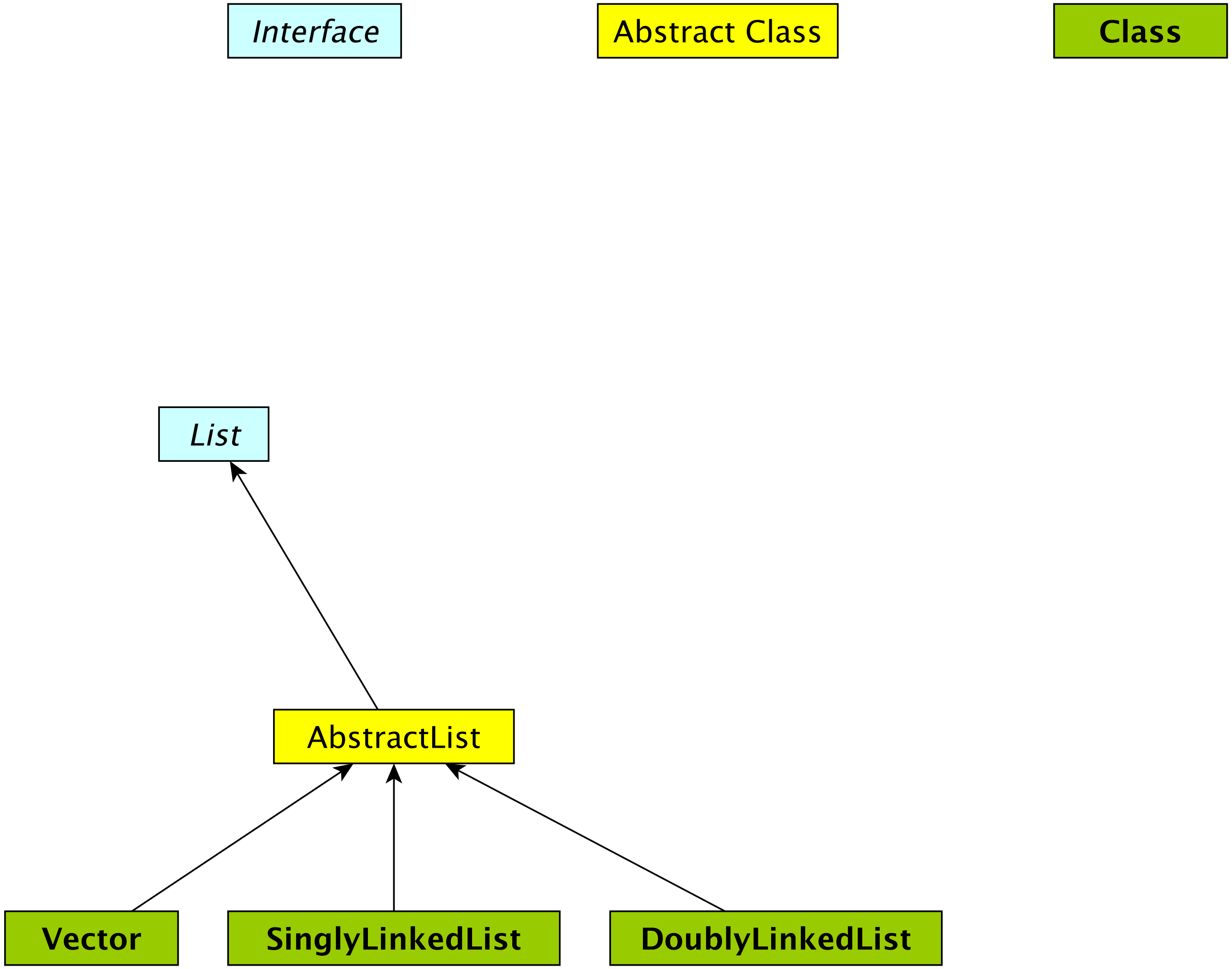
- AbstractList provides *some* of the list functionality
  - Code is shared among all sub-classes (see Ch. 7 for more info)

    ```
    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
- Concrete list classes extend AbstractList, implementing missing functionality

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

The Structure5 Universe (almost)



Interface

Abstract Class

Class

List

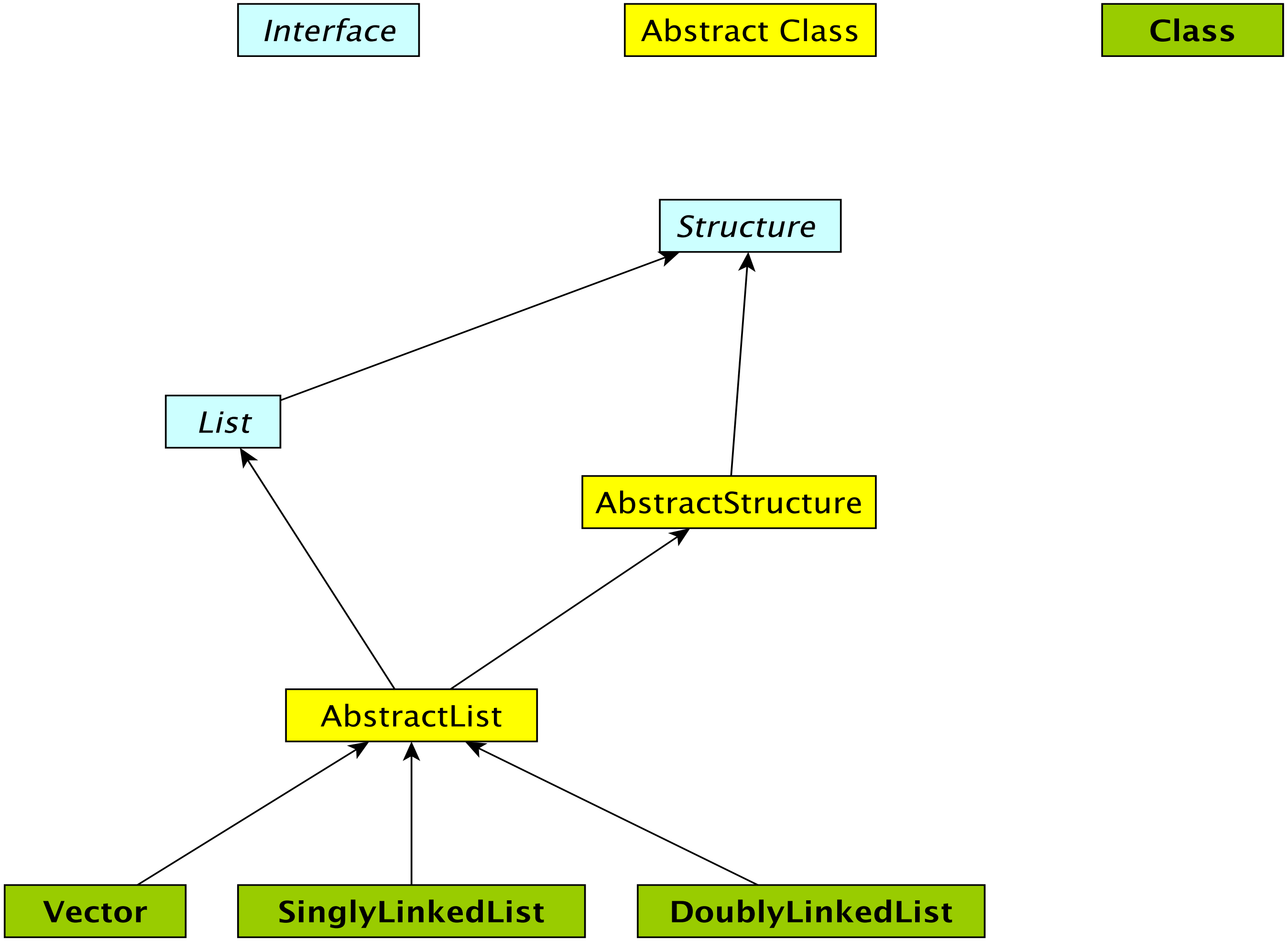
AbstractList

Vector

SinglyLinkedList

DoublyLinkedList

The Structure5 Universe (so far)



Interface

Abstract Class

Class

Structure

List

AbstractStructure

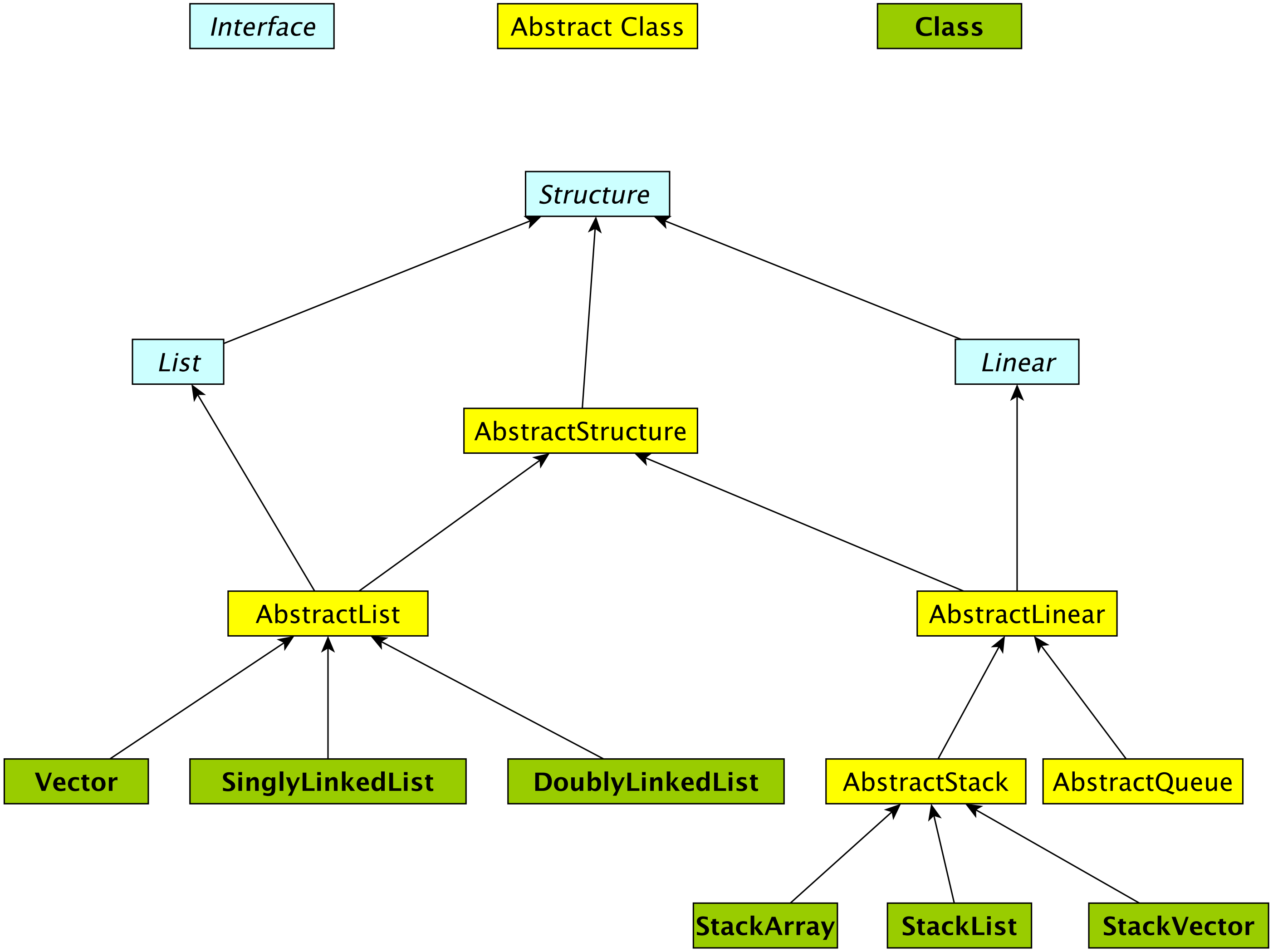
AbstractList

Vector

SinglyLinkedList

DoublyLinkedList

The Structure5 Universe (soon)


Interface
Abstract Class
Class
Structure
List
Linear
AbstractStructure
AbstractList
AbstractLinear
Vector
SinglyLinkedList
DoublyLinkedList
AbstractStack
AbstractQueue
StackArray
StackList
StackVector