CSCI 136
Data Structures &
Advanced Programming

Lecture 11

Fall 2019

Instructors: Bill & Sam

Last Time

- Comparables and Comparators

Today: Better Sorting

- Comparator example
- Merge Sort
- Quick Sort
- Class extension
  - Abstract base classes
  - Concrete extension classes

Faster Sorting: Merge Sort

- A *divide and conquer* algorithm
- Typically used on arrays
- Merge sort works as follows:
  - If the array is of length 0 or 1, then it is already sorted.
  - Divide the unsorted array into two arrays of about half the size of original.
  - Sort smaller arrays recursively by re-applying merge sort.
  - Merge the two smaller arrays back into one sorted array.
- Time Complexity?
  - Spoiler Alert! We'll see that it's O(n log n)
- Space Complexity?
  - O(n)

Merge Sort

- [8 14 29 1 17 39 16 9]
- [8 14 29 1] [17 39 16 9] split
- [8 14] [29 1] [17 39] [16 9] split
- [8] [14] [29] [1] [17] [39] [16] [9] split
- [8 14] [1 29] [17 39] [9 16] merge
- [1 8 14 29] [9 16 17 39] merge
- [1 8 9 14 16 17 29 39] merge

Merge Sort : Pseudo-code

- How would we *design* it?
- First pass…

// recursively mergesorts A[from .. To] "in place"
void recMergeSortHelper(A[], int from, int to)
    if (from ≤ to)
        mid = (from + to)/2
        recMergeSortHelper(A, from, mid)
        recMergeSortHelper(A, mid+1, to)
        merge(A, from, to)

But *merge* hides a number of important details….

Merge Sort : Java Implementation

- How would we *implement* it?
  - Review MergeSort.java
  - Note carefully how temp array is used to reduce copying
  - Make sure the data is in the correct array!
- Time Complexity?
  - Takes at most k comparisons to merge two lists of size k
  - Number of splits/merges for list of size n is log n
  - Claim: At most time O(n log n)…We'll see soon...
- Space Complexity?
  - O(n)?
  - Need an extra array, so really O(2n)! But O(2n) = O(n)

Merge Sort = O(n log n)

- [8 14 29 1 17 39 16 9]
- [8 14 29 1] [17 39 16 9] split
- [8 14] [29 1] [17 39] [16 9] split — log n
- [8] [14] [29] [1] [17] [39] [16] [9] split
- [8 14] [1 29] [17 39] [9 16] merge
- [1 8 14 29] [9 16 17 39] merge — log n
- [1 8 9 14 16 17 29 39] merge

merge takes at most n comparisons per line

Time Complexity Proof

- Prove for $n = 2^k$ (true for other n)
- That is, MergeSort for performs at most
  - $n * \log(n) = 2^k * k$ comparisions of elements
- Base cases $k \leq 1$: 0 comparisons: $0 < 1 * 2^1$ ✓
- Induction Step: Suppose true for all integers smaller than k. Let T(k) be # of comparisons for 2^k elements. Then
- $\underline{T(k) \leq} \, 2^k + 2 * T(k-1) \leq 2^k + 2(k-1)2^{k-1} \leq \underline{k * 2^k}$ ✓

Merge Sort

- Unlike Bubble, Insertion, and Selection sort, Merge sort is a divide and conquer algorithm
  - Bubble, Insertion, Selection sort complexity: $O(n^2)$
  - Merge sort complexity: $O(n \log n)$
- Are there any limitations with Merge sort?
- Why would we ever use any other algorithm for sorting?

Drawbacks to Merge Sort

- Need extra temporary array
  - If data set is large, this could be a problem
- Waste time copying values back and forth between original array and temporary array
- Can we avoid this?

Quick Sort

- Quick sort is designed to behave much like Merge sort, without requiring extra storage space

Merge Sort	Quick Sort
Divide list in half	Partition* list into 2 parts
Sort halves	Sort parts
Merge halves	Join* sorted parts

Quick Sort

```
public void quickSortRecursive(Comparable data[],
                               int low, int high) {
    // pre: low <= high
    // post: data[low..high] in ascending order
        int pivot;
        if (low >= high) return;

       /* 1 - place pivot */
        pivot = partition(data, low, high);
       /* 2 - sort small */
       quickSortRecursive(data, low, pivot-1);
       /* 3 - sort large */
       quickSortRecursive(data, pivot+1, high);
}
```

Partition

1. Put first element (pivot) into sorted position
2. All to the left of "pivot" are smaller and all to the right are larger
3. Return index of "pivot"

Partition

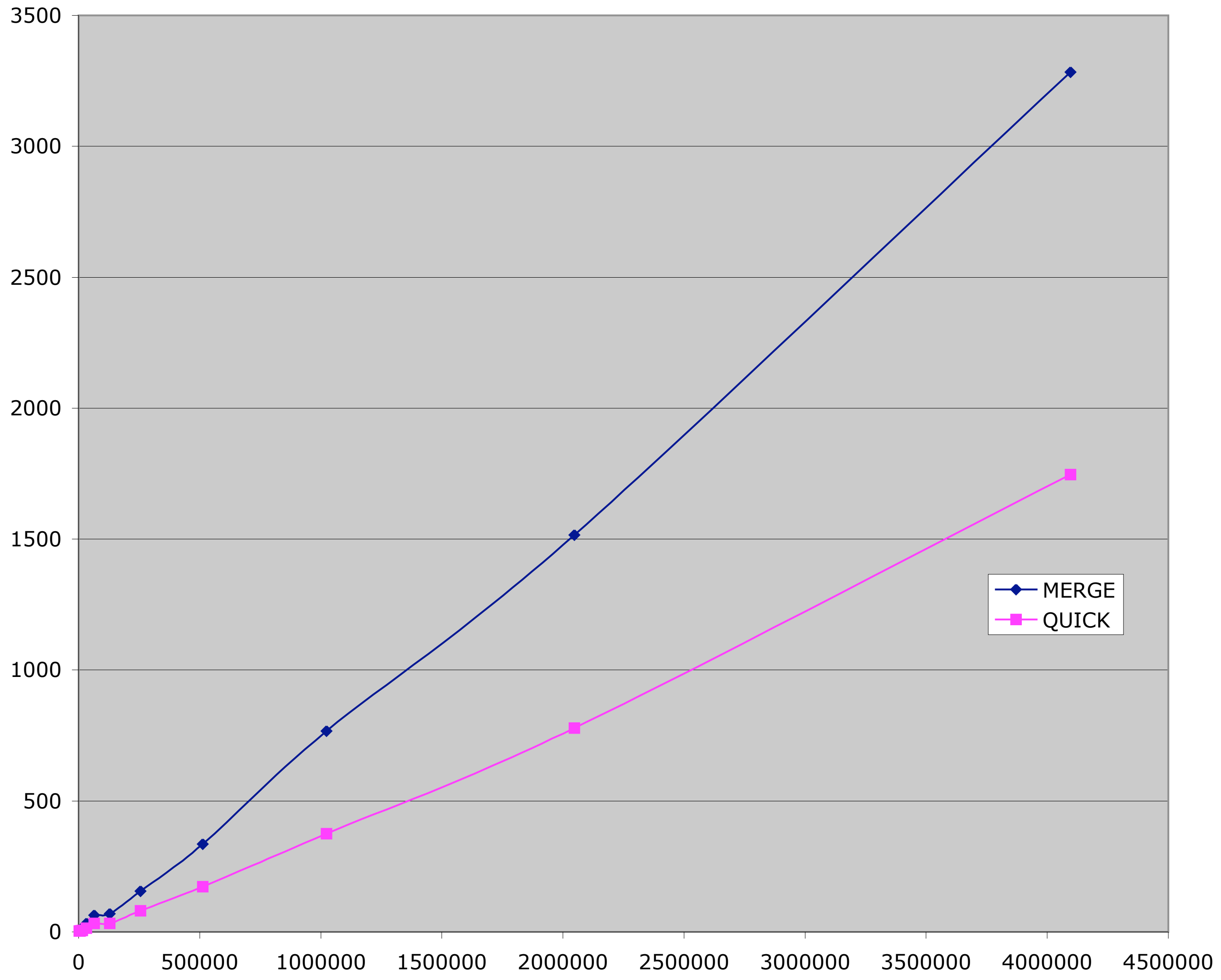
```
int partition(int data[], int left, int right) {
  while (true) {
    while (left < right && data[left] < data[right])
      right--;
    if (left < right) {
      swap(data,left++,right);
    } else {
      return left;
    }

    while (left < right && data[left] < data[right])
      left++;
    if (left < right) {
      swap(data,left,right--);
    } else {
      return right;
    }
  }
}
```

Complexity

- Time:
  - Partition is O(n)
  - If partition breaks list exactly in half, same as merge sort, so O(n log n)
  - If data is already sorted, partition splits list into groups of 1 and n-1, so $O(n^2)$
- Space:
  - O(n) (so is MergeSort)
    - In fact, it's n + c compared to 2n + c for MergeSort

Merge vs. Quick (Average Time)



Food for Thought…

- How to avoid picking a bad pivot value?
  - Pick median of 3 elements for pivot (heuristic!)
  - i.e. first, middle, last
- Combine selection sort with quick sort
  - For small n, selection sort is faster
  - Switch to selection sort when elements is <= 7
  - Switch to selection/insertion sort when the list is almost sorted (partitions are very unbalanced)
    - Heuristic!

Sorting Wrapup

	Time	Space
Bubble	Worst: $O(n^2)$ Best: $O(n)$ - if “optimiazed”	$O(n)$: $n + c$
Insertion	Worst: $O(n^2)$ Best: $O(n)$	$O(n)$: $n + c$
Selection	Worst = Best: $O(n^2)$	$O(n)$: $n + c$
Merge	Worst = Best:: $O(n \log n)$	$O(n)$: $2n + c$
Quick	Average = Best: $O(n \log n)$ Worst: $O(n^2)$	$O(n)$: $n + c$

Class Specialization

- Classes can *extend* other classes
  - Inherit fields and **method bodies**
- By extending other classes, we can create specialized sub-classes
- Java supports class extension/specialization
- Java enforces *type-safety*: Objects behave according to their type
  - Some checks are made at compile-time
  - Some checks are made at run-time
- We'll first use this feature to factor out code

Abstract Classes

- Note: All of our Card implementations code `toString()` in identical fashion.
- It’s good to be able to “factor out” common code so that it only has to be maintained in one place
- *Abstract classes* to the rescue….
- An abstract class allows for a *partial* implementation
- We can then *extend* it to a complete implementation
- Let’s do this with our cards.
  - Examine CardAbstract.java....
- As with interfaces, can‘t use “new“ with abstract types!

Abstract Classes

Notes from `CardAbstract` class example

- `CardAbstract` *implements* `Card` (partially)
- `CardAbstract` is declared to be *abstract*
  - It contains the implementation of `toString(), equals(), and compareTo()` [Note: We made our cards comparable!]

How do the full implementations (`CardRankSuit`, etc) change?

- They are declared to *extend* `CardAbstract`
- They don't need to say "`implements Card`"
- They don't contain the `toString()` method
  - They *inherit* that method from `CardAbstract`
  - But could *override* that method if desired

Extending Concrete Classes

Let's call a class *concrete* if it is not abstract

We can extend concrete classes

Example: Adding a point count to a `Card`

- Suppose we wanted to add a point value to each of the playing cards in `CardRankSuit`
- We *extend* that class

```
class CardRankSuitPoints extends CardRankSuit {… }
```

- This new class can now contain additional instance variables and methods
- Let's look at the code for CardRankSuitPoints.java….

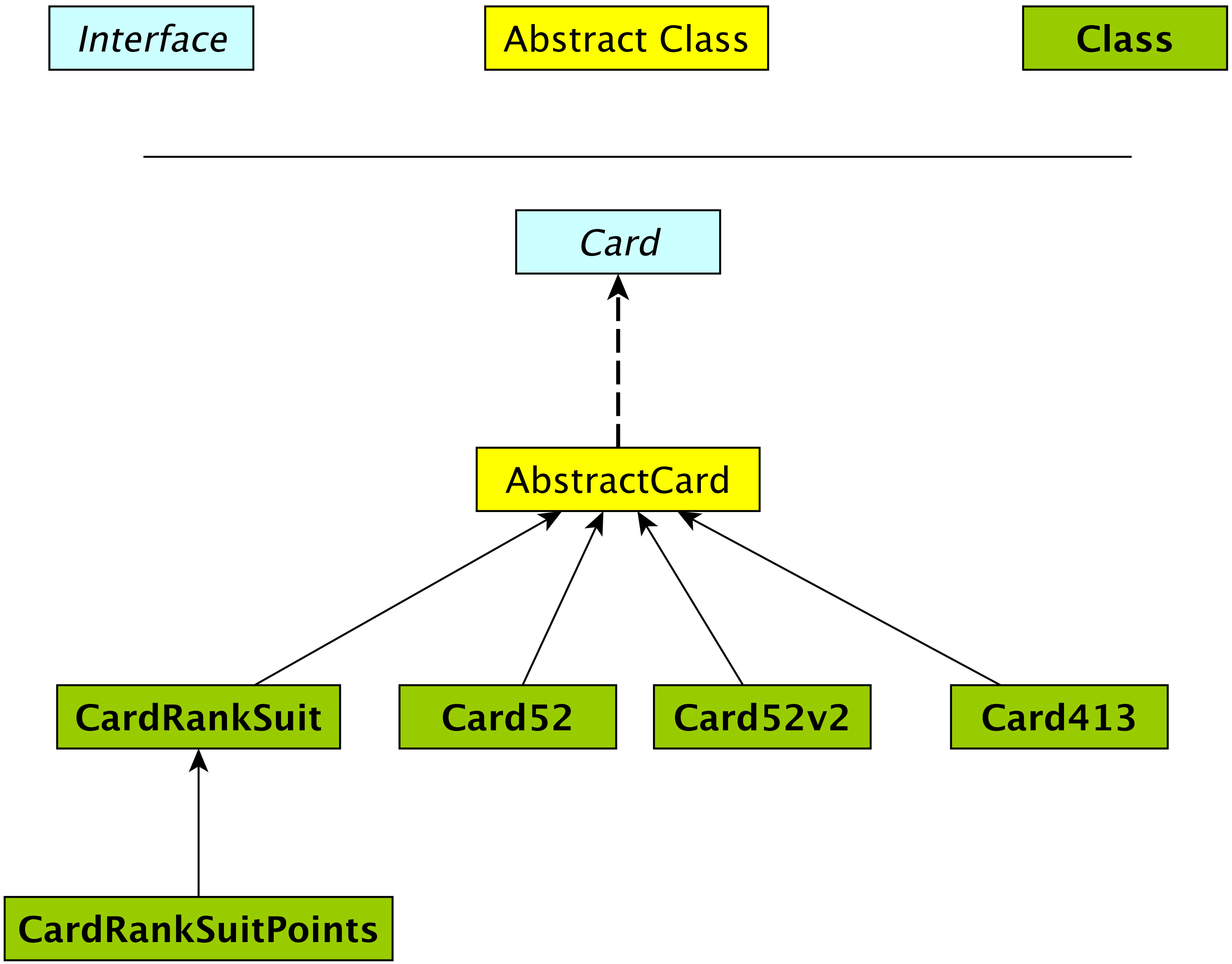
CardRankSuitPoints Notes

- Constructor calls `CardRankSuit` constructor using *super*
- We can override methods---e.g., `toString()`
- Can use a `CardRankSuitPoints` object wherever we use a `Card`
  - But! Can only use new features (`getPoints()`) if the object is declared to be of type `CardRankSuitPoints`

```
CardRankSuitPoints c1 = new CardRankSuitPoints(
    Rank.ACE, Suit.CLUBS, 4);
int p1 = c1.getPoints(); // Legal
Card c2 = new CardRankSuitPoints(Rank.ACE,
    Suit.CLUBS, 4);
int p2 = c2.getPoints(); // Bad! c2 is of type Card
int p3 = ((CardRankSuitPoints) c2).getPoints(); // Legal
```

- Java enforces *type-safety*: An variable of type X can only be assigned a value of type X or of a type that extends X

The Card Classes Hierarchy



Pros and Cons of Vectors

Pros

- Good general purpose list
- Dynamically Resizeable
- Fast access to elements
  - vec.get(387425) finds item 387425 in the same number of operations regardless of vec's size

Cons

- Slow updates to front of list (why?)
- Hard to predict time for add (depends on internal array size)
- Potentially wasted space

Today we look at another way to store data: Linked Lists

But First : List Interface

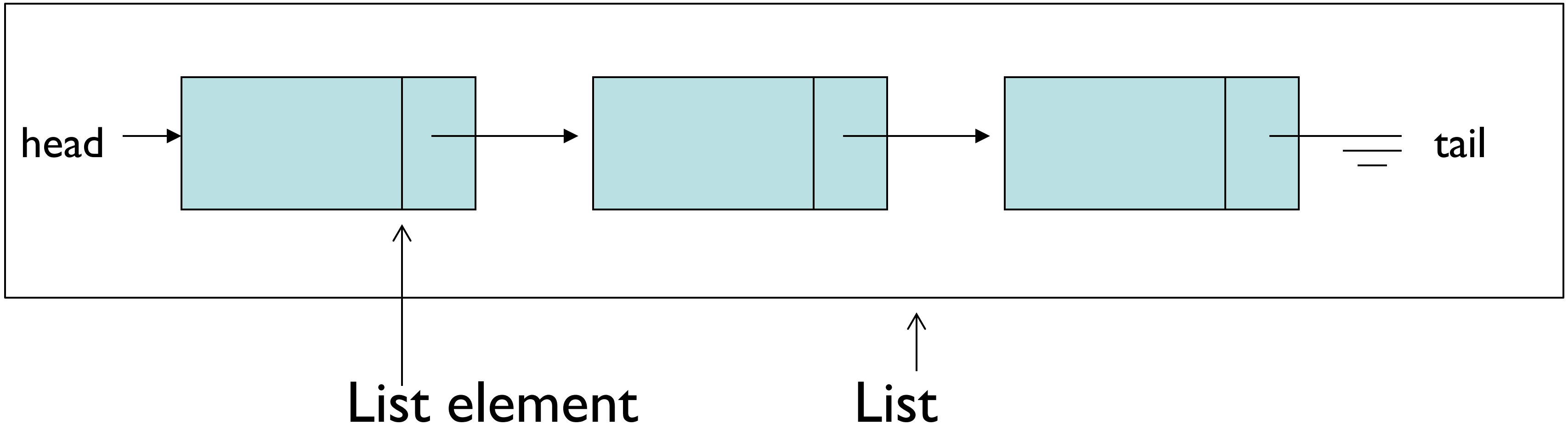
```
interface List {
    size()
    isEmpty()
    contains(e)
    get(i)
    set(i, e)
    add(i, e)
    remove(i)
    addFirst(e)
    getLast()
    .
    .
    .
}
```

- Flexible interface
- Can be used to describe many different types of lists
- It's an interface…therefore it provides no implementation
- Vector implements List
- Other implementations are possible
  - SinglyLinkedList
  - CircularlyLinkedList
  - DoublyLinkedList

Linked List Basics

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

- Visualizing lists

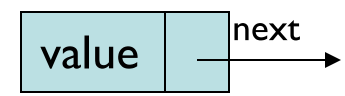


Linked List Basics

- List nodes are recursive data structures
- 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 = SLLE in these notes
  - SLLE = Node in structure5 (and in Ch 9)
  - Let's look at SLLE.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 proportial 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

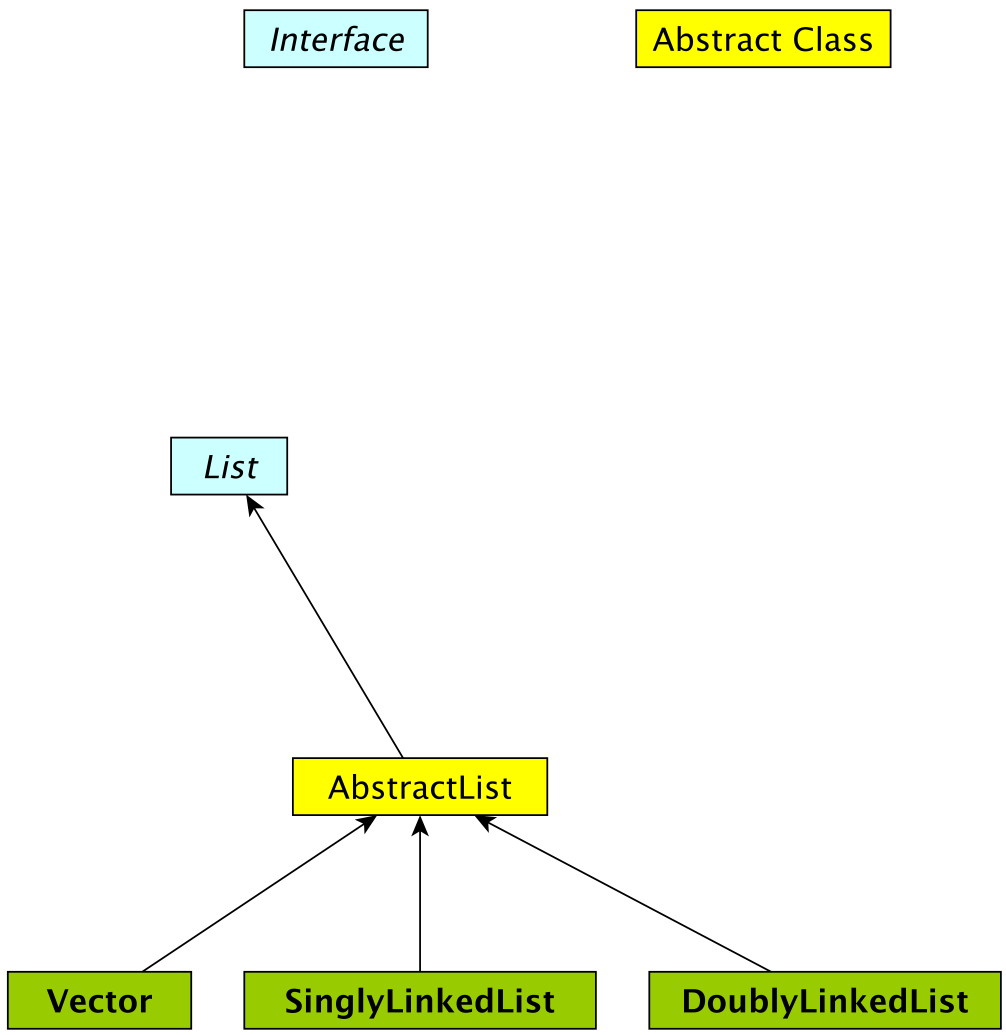
```
abstract class AbstractList<E> implements List<E> {
    public void addFirst(E element) { add(0, element); }
    public E getLast() { return get(size()-1);}
    public E removeLast() { return remove(size()-1); }
}
```

- 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)