CSCI 136
Data Structures &
Advanced Programming

Lecture 14

Fall 2016

Instructor: Bill Lenhart

Announcements

- Problem Set 2 due Thursday@11:00pm
  - Leave in instructor cubby outside of TCL 303
- Mid-Term Review Session
  - 10/16 and 10/17, 8:00-9:00 pm in a location TBD
  - No prepared remarks, so bring questions!
- Mid-term exam is Wednesday, October18
  - During your normal lab session
  - You'll have approximately 1 hour & 45 minutes (if you come on time!)
  - Closed-book: Covers Chapters 1-7 & 9, handouts, and all topics up through Sorting
  - A "sample" mid-term and study sheet will be available online

Last Time

- Basic Sorting Summary
- Comparator interfaces for flexible sorting
- More Efficient Sorting Algorithms
  - MergeSort
  - QuickSort

Today

- QuickSort and Sorting Wrap-Up
- Linear Structures
  - The Linear Interface (LIFO & FIFO)
  - The AbstractLinear and AbstractStack classes
- Stack Implementations
  - StackArray, StackVector, StackList,
- Stack applications
  - Expression Evaluation
  - PostScript: Page Description & Programming
  - Mazerunning (Depth-First-Search)

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

Recall Merge Sort

```
private static void mergeSortRecursive(Comparable data[],
                    Comparable temp[], int low, int high) {
   int n = high-low+1;
   int middle = low + n/2;
   int i;

   if (n < 2) return;
   // move lower half of data into temporary storage
   for (i = low; i < middle; i++) {
       temp[i] = data[i];
   }
   // sort lower half of array
   mergeSortRecursive(temp,data,low,middle-1);
   // sort upper half of array
   mergeSortRecursive(data,temp,middle,high);
   // merge halves together
   merge(data,temp,low,middle,high);
}
```

Quick Sort

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

        /* 1 – split with 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 by Hungarian Folk Dance

Partition

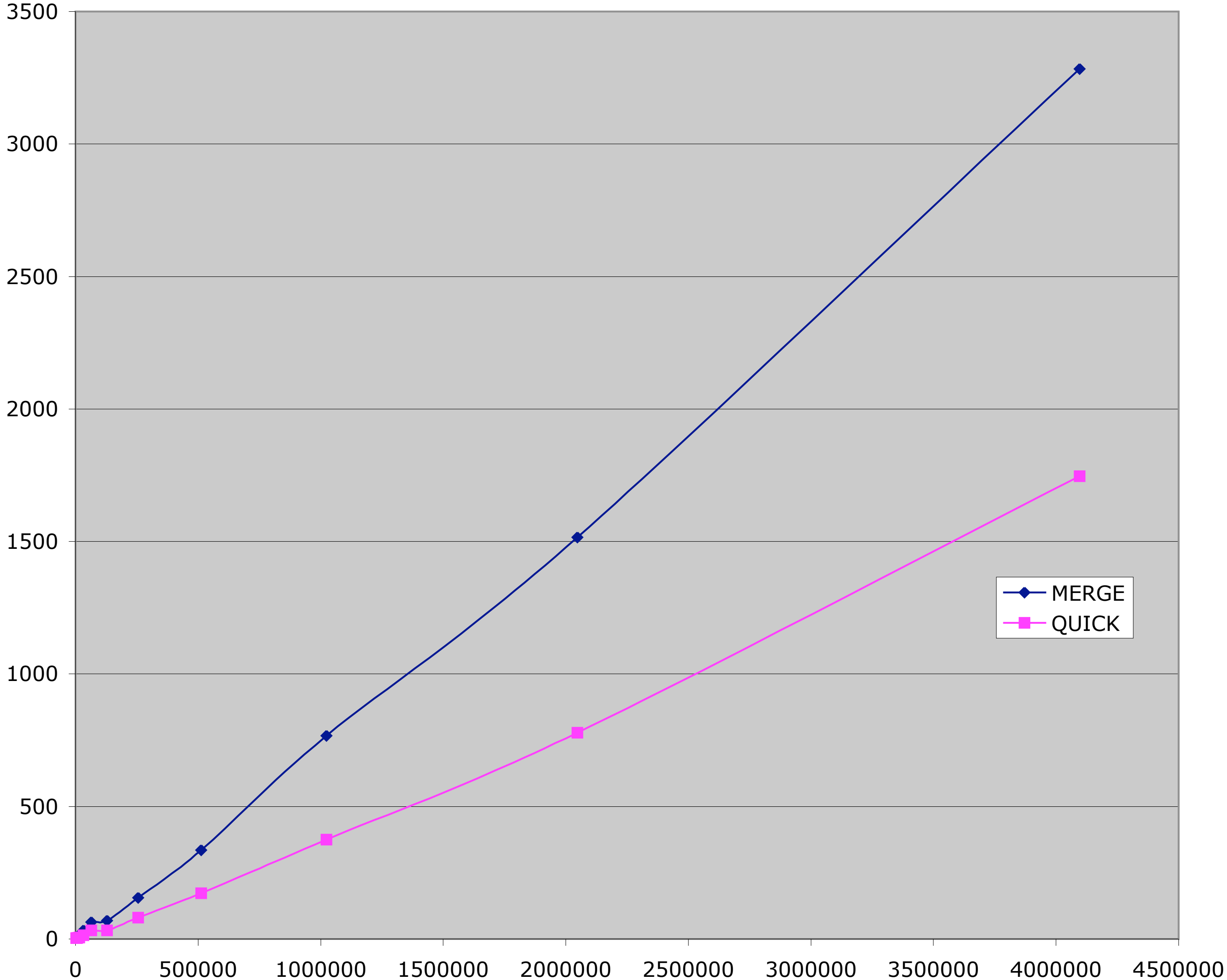
```
int partition(int data[], int left, int right) {
  while (true) {
    while (left < right && data[left] < data[right])
      right--;
    if (left < right) {
      swap(data,left++,right);
    } else {
      return left;
    }

    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 MergSort)
    - In fact, it's n + c compared to 2n + c for MergeSort

Merge vs. Quick



Food for Thought…

- How to avoid picking a bad pivot value?
  - Pick median of 3 elements for pivot (heuristic!)
- Combine selection sort with quick sort
  - For small n, selection sort is faster
  - Switch to selection sort when n is small
  - 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

More Skill-Testing (Try these at home)

Given the following list of integers:

9 5 6 1 10 15 2 4

1) Sort the list using Bubble sort. Show your work!
2) Sort the list using Insertion sort. . Show your work!
3) Sort the list using Merge sort. . Show your work!
4) Verify the best and worst case time and space complexity for each of these sorting algorithms as well as for selection sort.

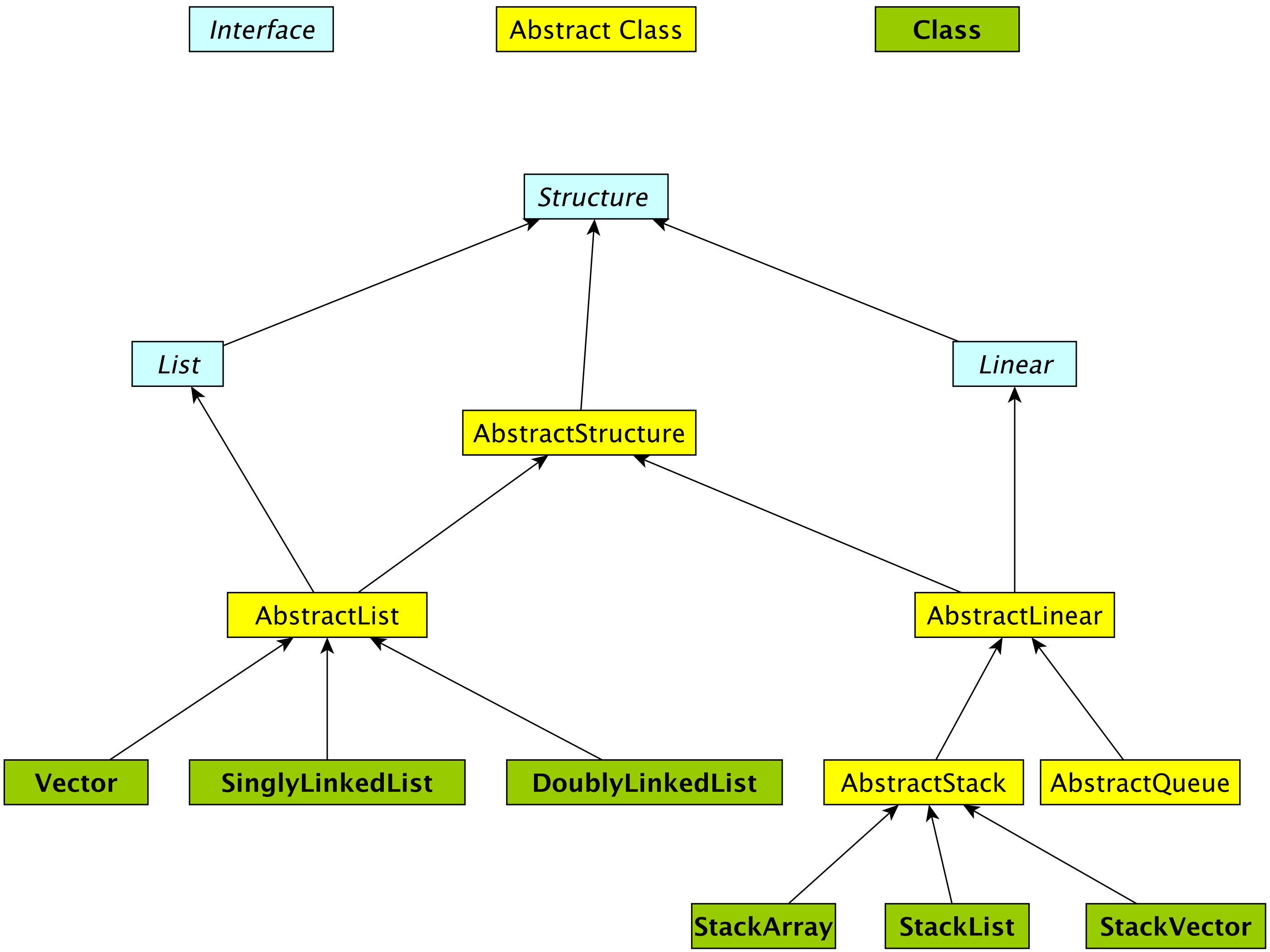
Linear Structures

- 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.
    - 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

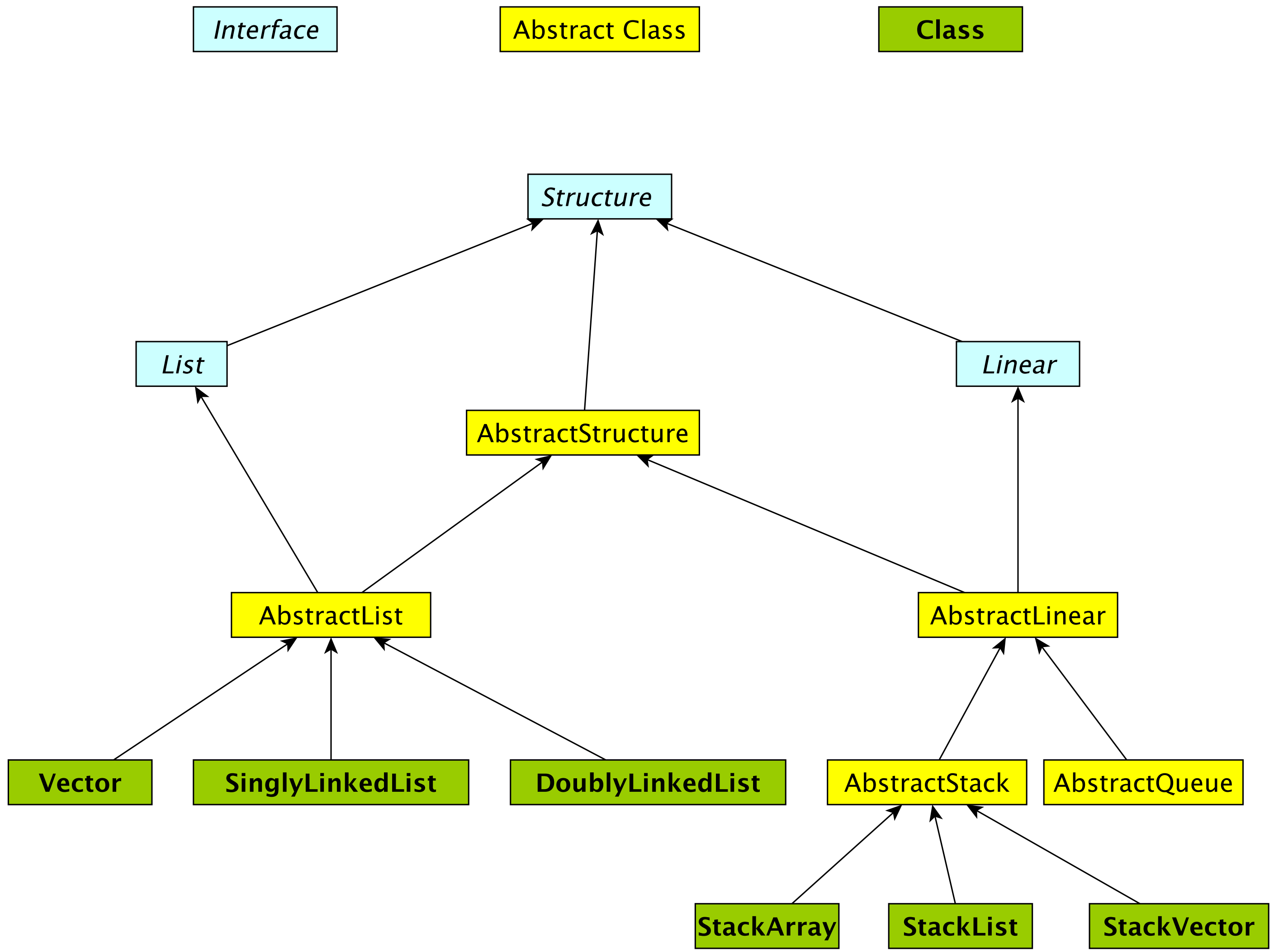
- structure5.StackList
  - SLL data
  - Add/remove from head

+ 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 → 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) → 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