CSCI 136
Data Structures & Advanced Programming

Lecture 17

Spring 2018

Profs Bill & Jon

Administrative Details

- Congratulations
- Lab 7: PostScript
  - Will be posted over Spring Break
  - Can't wait!?
    - Read about it in Java Structures: Section 10.5
  - No partners this time
  - Review before lab & come to lab with design doc

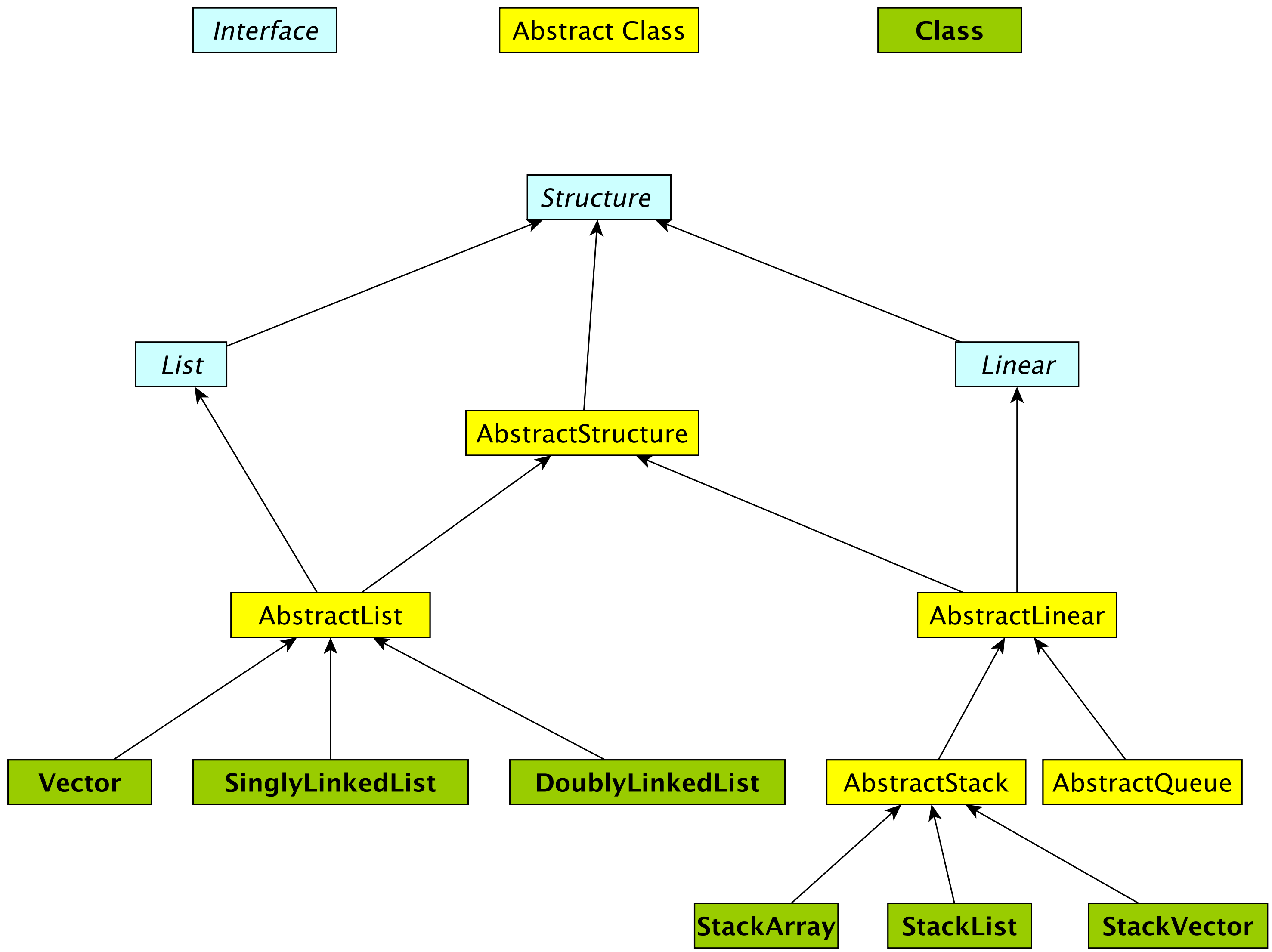
Last Time : Linear Structures

- Linear Interface
- AbstractLinear
- Stacks
  - StackArray
  - StackList
  - StackVector

Today: Linear Structures

- Stack Applications
  - Postfix expressions
  - Postscript
  - Program Stack
- Queues
  - Implementation Details
  - Applications
- Iterators

The Structure5 Universe (next)



The Linear 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()`

Building The Hierarchy

- We extend `AbstractStack` to make “concrete” Stack types
  - `StackArray<E>`
    - holds an array of type E
    - add/remove at high end
  - `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

Stack Applications

- The `Stack` implementation is simple, but there are *many* applictaions
  - Evaluating mathematical expressions
  - Searching (Depth-first search)
  - Removing recursion for optimization
  - …

See book for details because this is VERY useful!

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 are 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 `xy*zw*+` :
  - 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
- Try with: w=3, x=4, y=5, z=6

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....

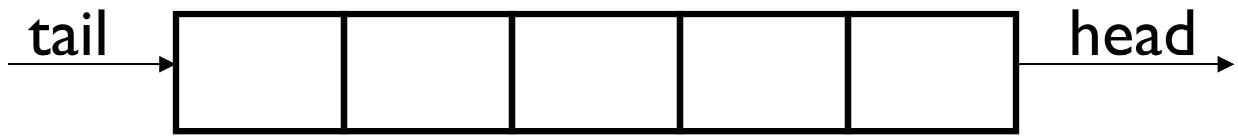
Preview: PostScript

- 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
    - RNAbows

Stacks vs. Queues

- Stacks are LIFO (Last In First Out)
- Queues are FIFO (First In First Out)
  - Another linear data structure (implements `Linear` interface)
  - Queue interface methods: enqueue (add), dequeue (remove), getFirst (get), peek (get)

Queues



- Examples:
  - Lines at movie theater, grocery store, etc.
  - OS event queue (keeps keystrokes, mouse clicks, etc, in order)
  - Printers
  - Routing network traffic (more on this later)

Queue Interface

```
public interface Queue<E> extends Linear<E> {
  public void enqueue(E item);
  public E dequeue();
  public E getFirst(); //value not removed
  public E peek();  //same as get()
}
```

Implementing Queues

As with Stacks, we have three options:

QueueArray

```
class QueueArray<E> implements Queue<E> {
   protected Object[] data; //can't instantiate E[]
   int head;
   int count; // can be used to determine tail...
}
```

QueueVector

```
class QueueVector<E> implements Queue<E> {
   protected Vector<E> data;
}
```

QueueList

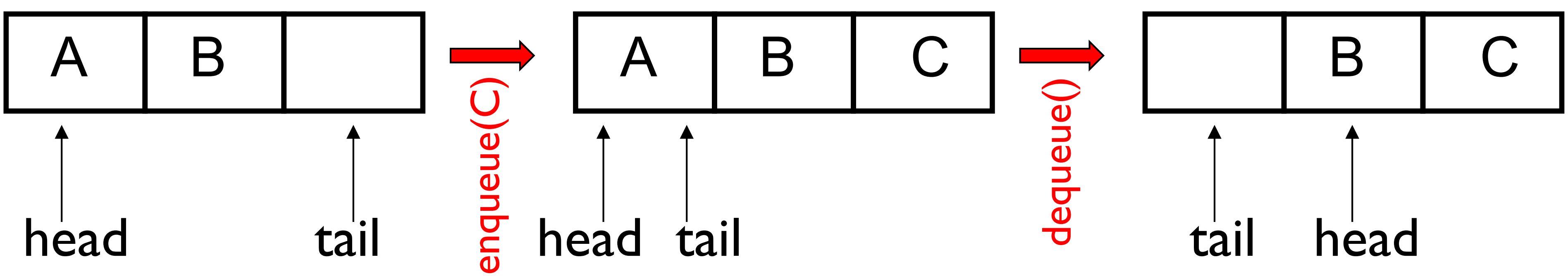
```
class QueueList<E> implements Queue<E> {
   protected List<E> data; //uses a CircularList
}
```

Tradeoffs:

- QueueArray:
  - enqueue is O(1)
  - dequeue is O(1)
  - Faster operations, but limited size
- QueueVector:
  - enqueue is O(1) (but O(n) in worst case - ensureCapacity)
  - dequeue is O(n)
- QueueList:
  - enqueue is O(1) (addLast)
  - dequeue is O(1) (CLL removeFirst)

QueueArray

- Let's look at an example…
- How to implement?
  - enqueue(item), dequeue(), size()



head points to front of queue; tail points to next empty space (where next item will be added)

head and tail "wrap around" array; when queue is full, head == tail

After wrap around, head > tail in some cases!

```
public class QueueArray<E> {

     protected Object[] data;       // Must use object because...
     protected int head;
     protected int count;

    public QueueArray(int size) {
          data = new Object[size];  // ... can’t say “new E[size]”
    }

    public void enqueue(E item) {
        assert (count < data.length) : ”The queue is full.";
        int tail = (head + count) % data.length;
        data[tail] = item;
        count++;
    }

    public E dequeue() {
          assert (count > 0) :"The queue is empty.";
          E value = (E)data[head];
          data[head] = null;
          head = (head + 1) % data.length;
          count--;
          return value;
    }

     public boolean empty() {
          return count>0;
     }
```