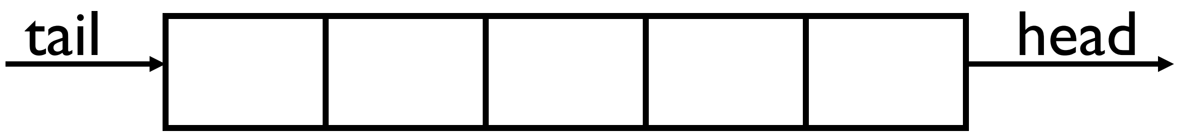
CSCI 136
Data Structures & Advanced Programming

Williams College

Video Goals

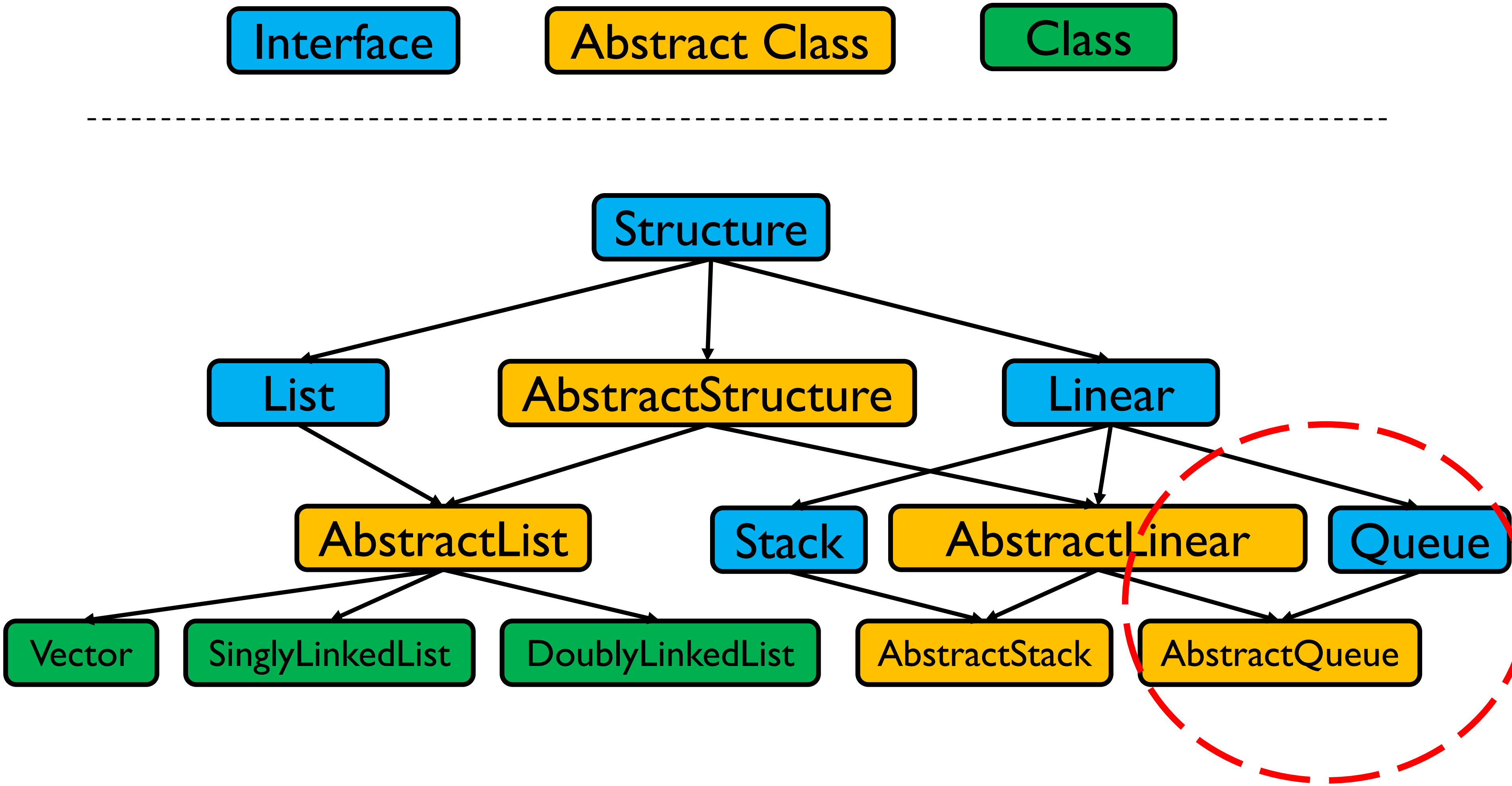
- Describe the queue abstract data type
- Contextualize queues within the Structure5 class hierarchy
- Discuss three queue implementations, focusing on their tradeoffs

Queues



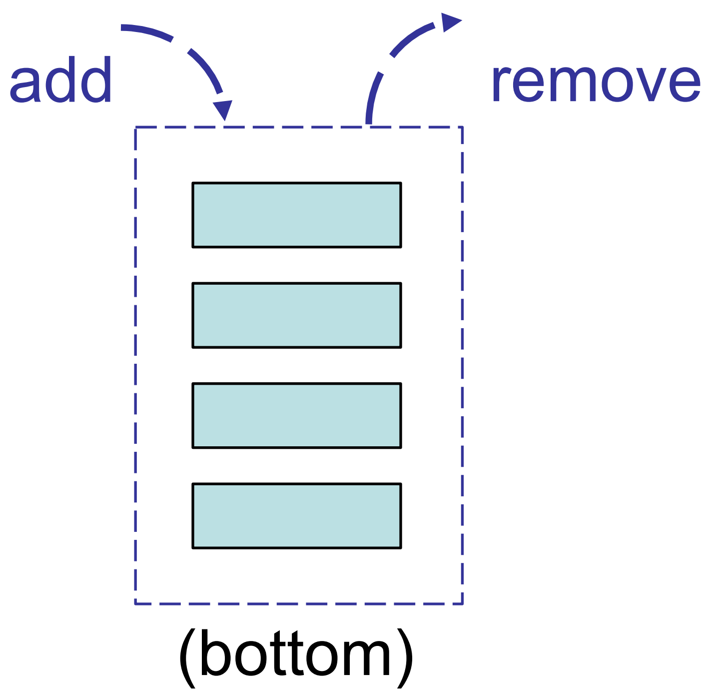
- A Queue is a collection of elements, but access is restricted to the "head" and "tail"
- Many "real-world" examples, including:
  - Lines at movie theater, grocery store, etc.
  - OS event queue (keeps keystrokes, mouse clicks, etc., in order)
  - Printers
  - Routing network traffic

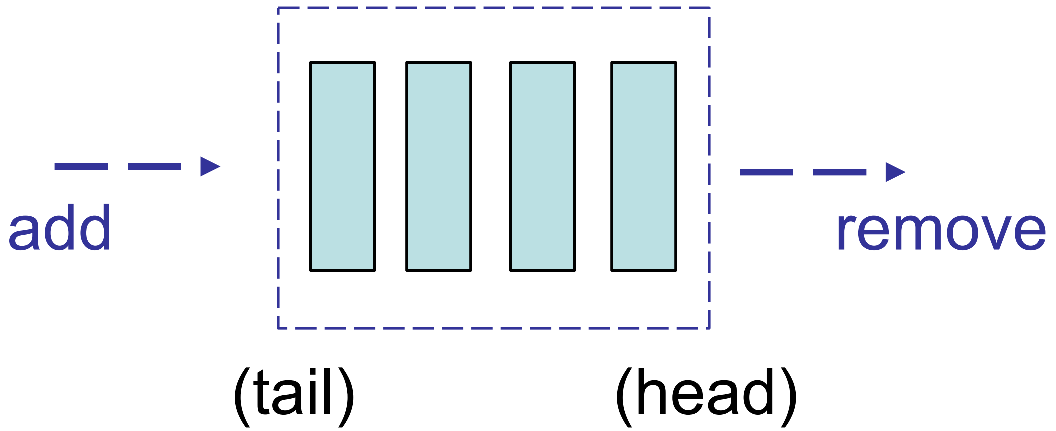
The Structure5 Universe (+ Linear!)



Stacks vs. Queues

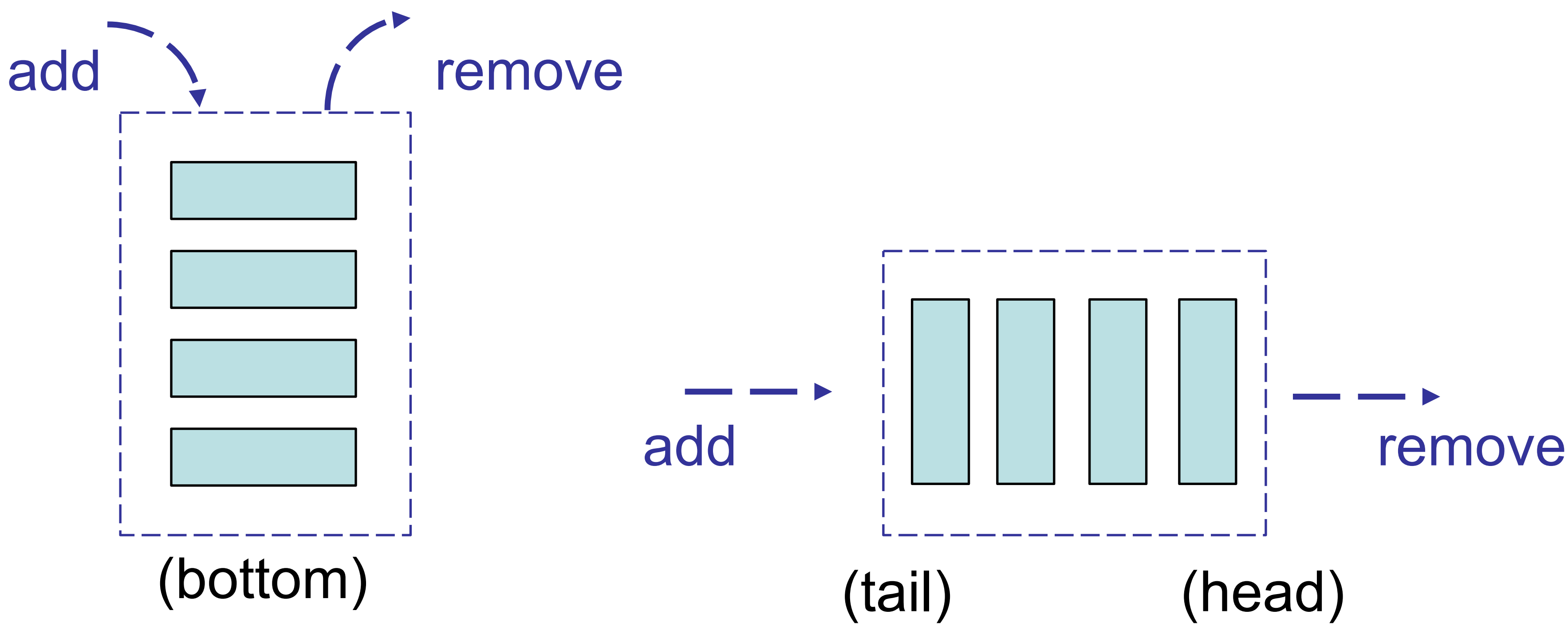
- Stacks are LIFO (Last In First Out)
- Queues are FIFO (First In First Out)





Stacks vs. Queues

- Stacks are LIFO (Last In First Out)
- Queues are FIFO (First In First Out)



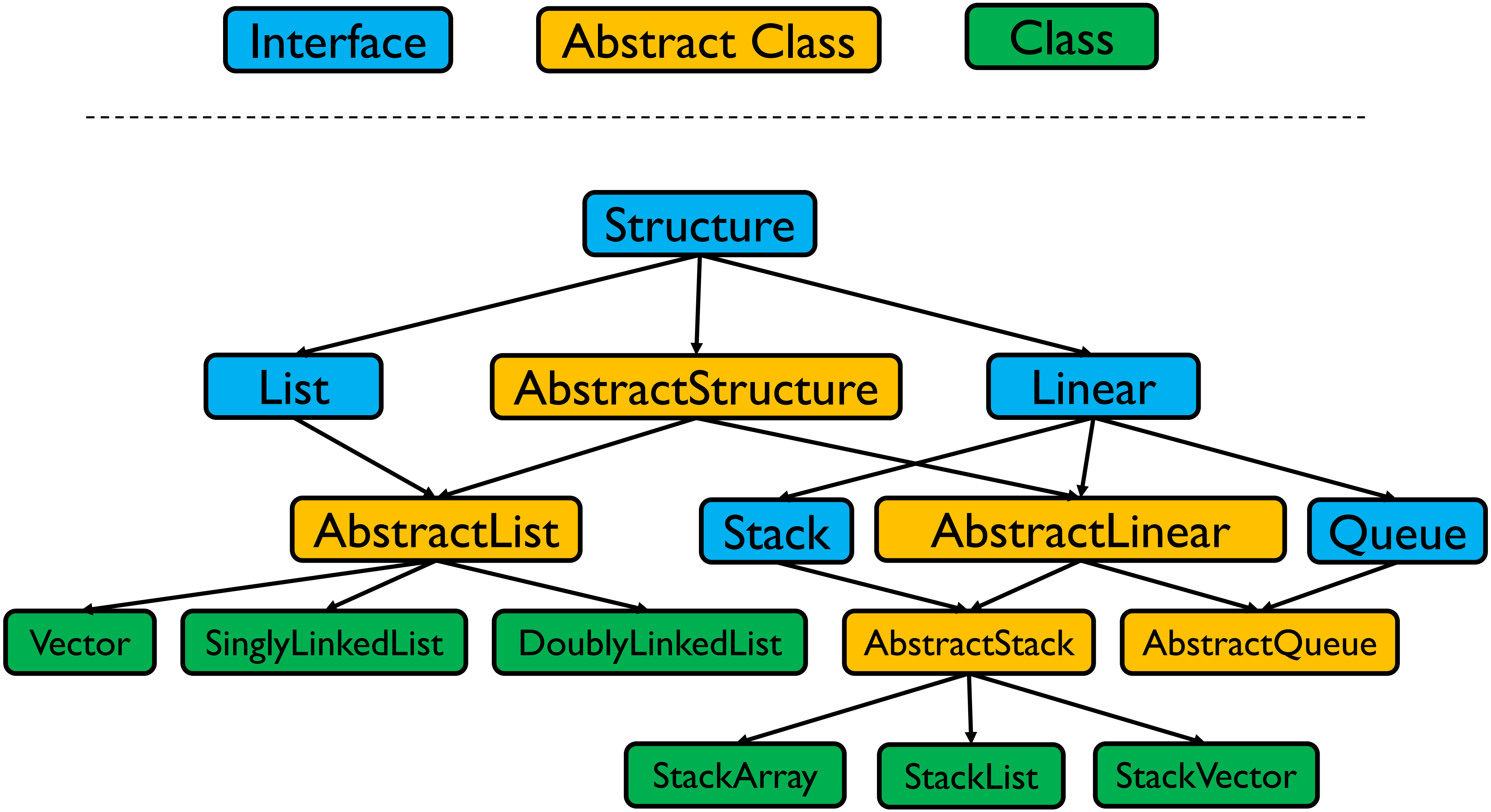
Stacks vs. Queues

- Both Stacks and Queues linear data structures (implement `Linear`, extend abstract classes that extend `AbstractLinear`),
- Like Stacks, Queues have their own terminology, which can be mapped to `Linear` interface methods:
  - `enqueue`: *insert* value at back of queue
  - `dequeue`: *remove* value from front of queue,
  - (`peek`: *access* value at front of queue)

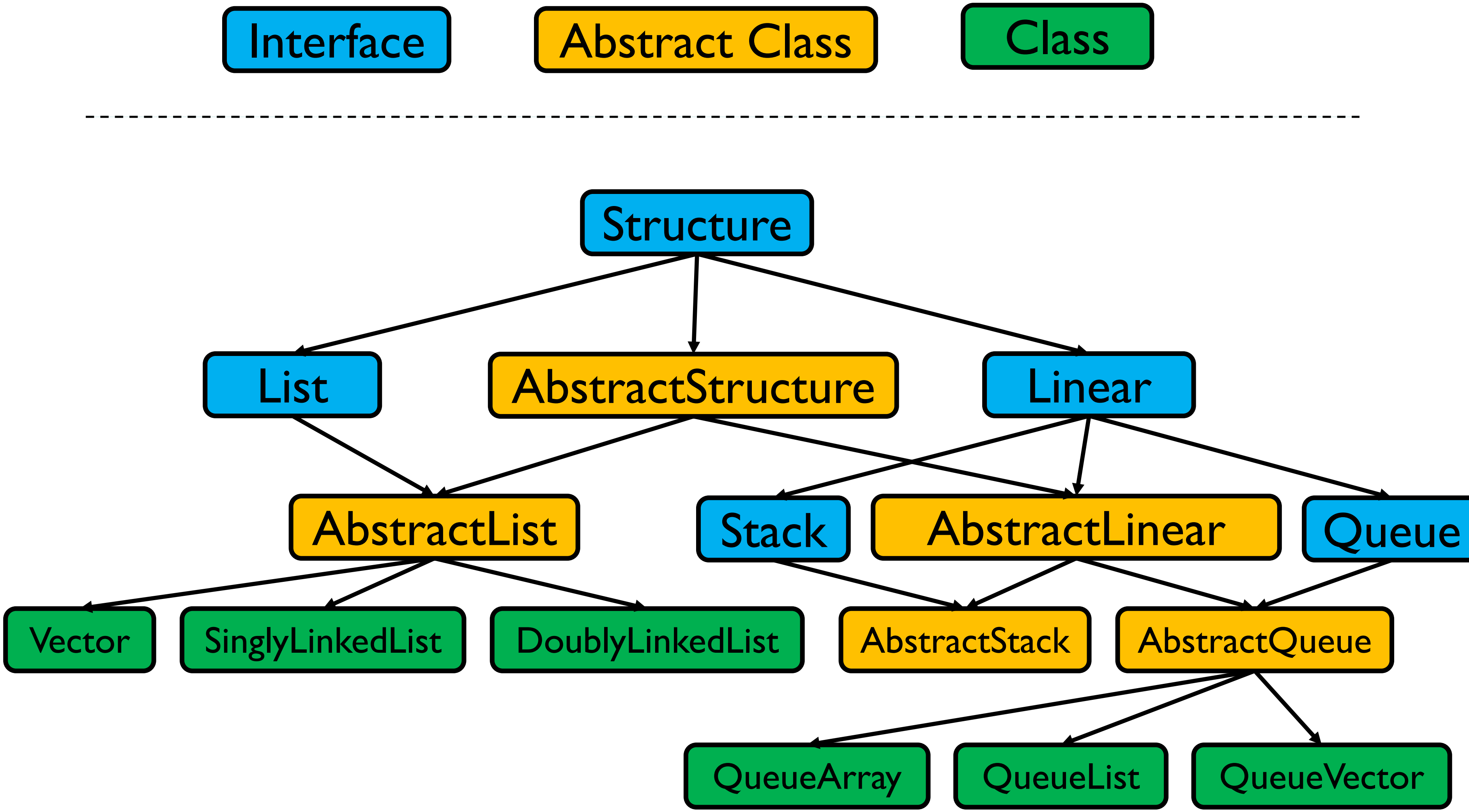
Stacks vs. Queues

- Also like Stacks, Queues can be implemented:
  - By using existing structures (e.g., `Vector`, `LinkedList`), or
  - As "stripped down" versions of those structures
    - We can implement a stacks/queues using the same underlying organization as those structures, but with reduced/simplified/optimized implementations

The Structure5 Universe (+ Stacks!)



The Structure5 Universe (+ Queues!)



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

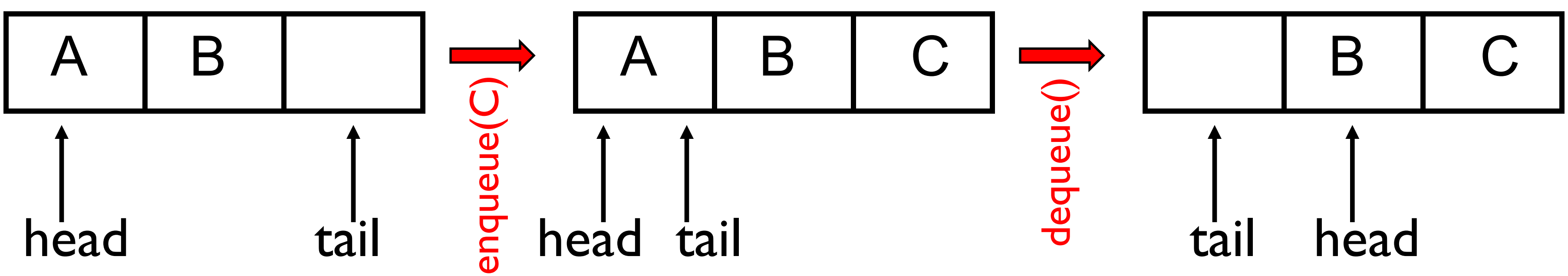
```
class QueueList<E> implements Queue<E> {
    protected List<E> data; //uses a CircularList
}
```

Tradeoffs:

- QueueArray:
  - `enqueue` is O(1): (rough idea) data[tail] = item;
  - `dequeue` is O(1): (rough idea) data[head] = null; head++;
  - Faster operations, but limited size
- QueueVector:
  - `enqueue` is O(1): uses `vec.addLast`
  - `dequeue` is O(n): uses `vec.removeFirst`
- QueueList:
  - `enqueue` is O(1): uses `lst.addLast`
  - `dequeue` is O(1): uses `lst.removeFirst`
    - Note: uses a Circularly Linked List so we have fast head and tail operations, but we only store one reference per node (next)

QueueArray

- Perhaps the most interesting implementation, so 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;
     }
```

QueueArray-style QueueVector?

- Why not use this same design with a Vector as our building block? Several decisions to make:
  - How do we interpret the respective meanings of `vec.elementCount`, `q.head`, and `q.count`?
  - How do we "grow" our Vector when our start/end are not at index 0 and `vec.size()-1`?
- These are all things that we can overcome, but we can't simply use a Vector as a "black box"
  - Note: structure5 takes the "black box" approach; intentionally demonstrates tradeoff of specialization

Takeaways

- Queues, like stacks, limit our access to specific locations of our data structure
  - However, this mimics common access patterns
- We can design a data structure that takes advantage of these limitations to optimize perf
- By utilizing these data structures, we can simplify/influence our algorithm design
- Enqueue/dequeue and push/pop are common terms, so be comfortable using them