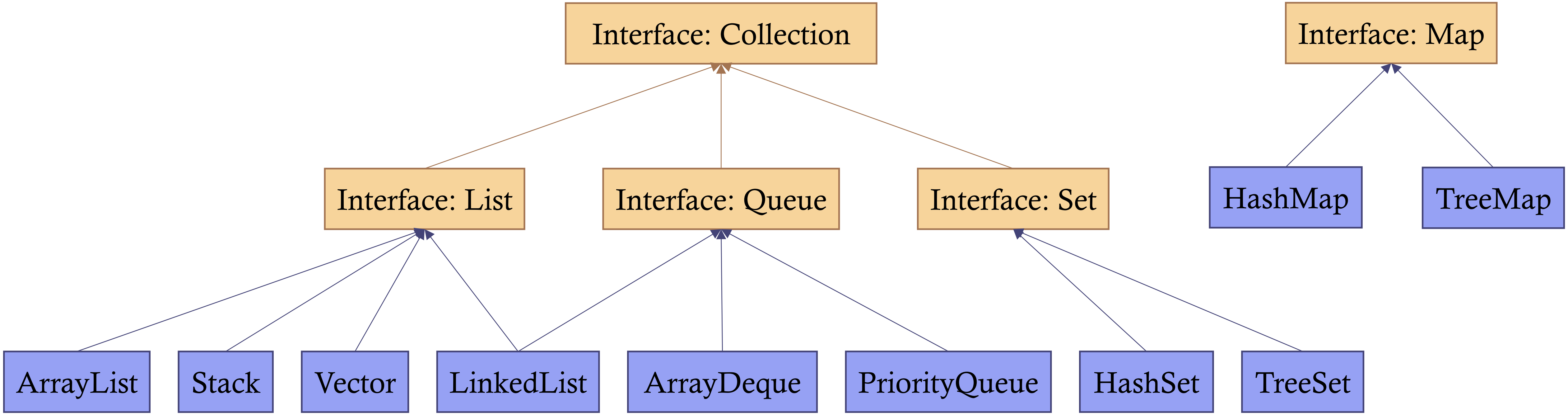
WEEK NINE

Acknowledgements: Slides created based off material provided by Dr. Michael Raymer and Dr. Travis Doom

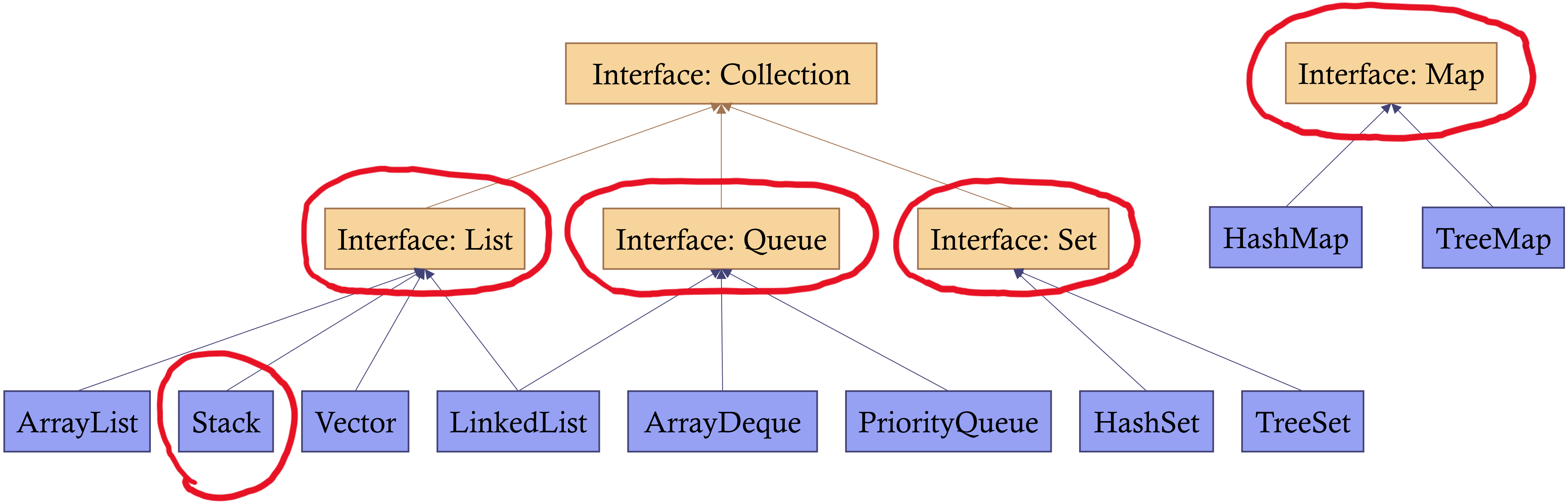
JAVA COLLECTION INTERFACE

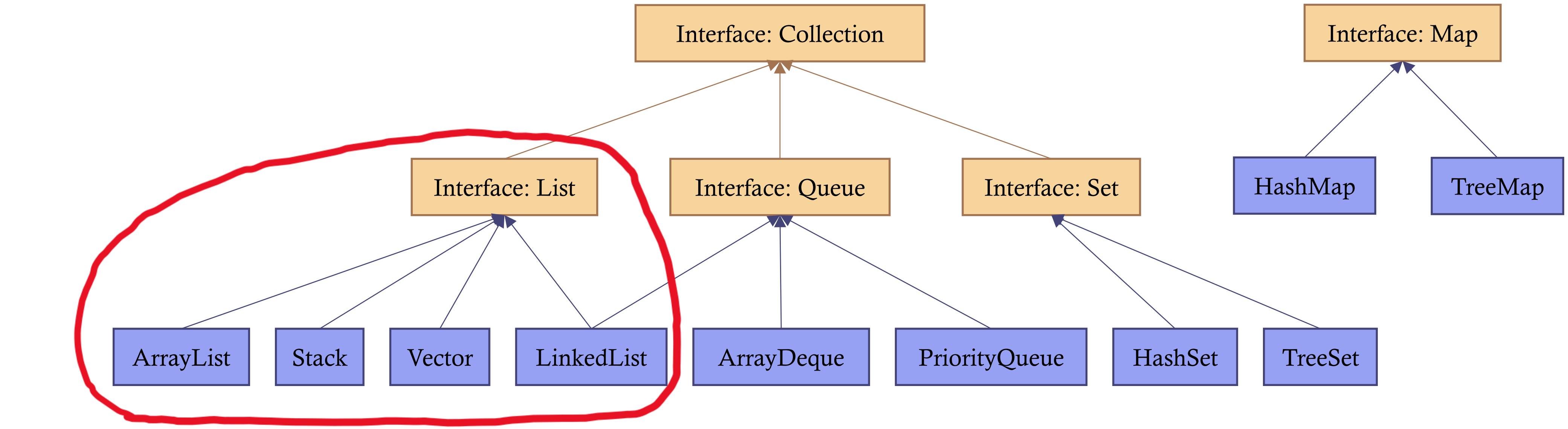


ABSTRACT DATA TYPES (ADTS)

- Abstract container for data with some loose operations
- An ADT is a general idea for data collection, details are hidden
- Describes how we want to store/interact data at a high-level
- An ADT **does not:**
    - Specify/restrict the type of data to be stored (generics)
    - Specify the implementation of operations (method bodies don't need to be specified)
    - Dictate how the data is actually stored/accessed from memory
- In Java, ADTs are often (but not always) implemented via interfaces

INTERESTING ADTS IN JAVA





LISTS

- ArrayLists
- LinkedLists

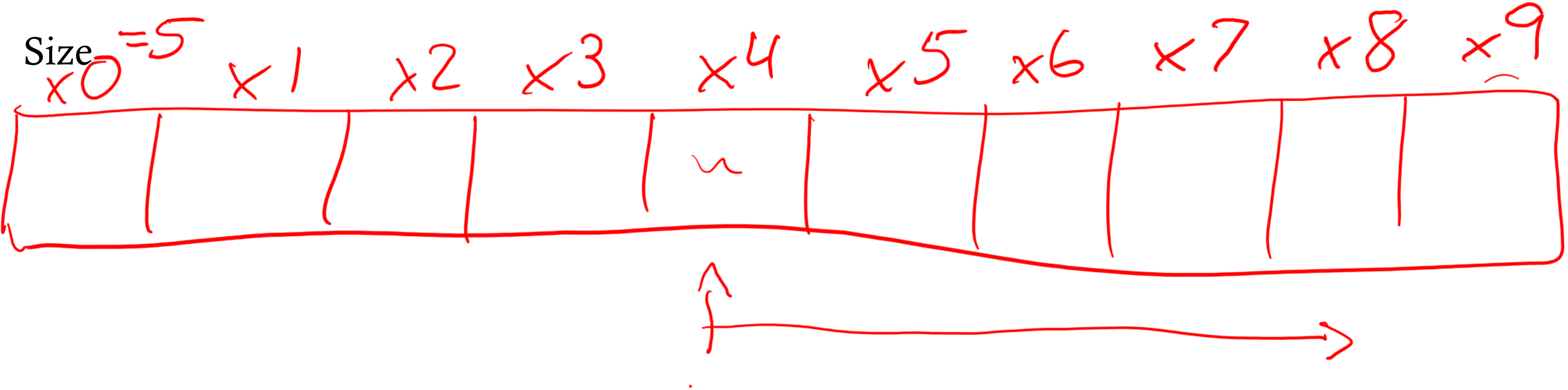
LIST INTERFACE

- An ordered, indexed collection
- Users can precisely insert/access elements via an index
- Duplicates typically allowed
- Key methods:
  - add()
  - get()
  - isEmpty()
  - remove()

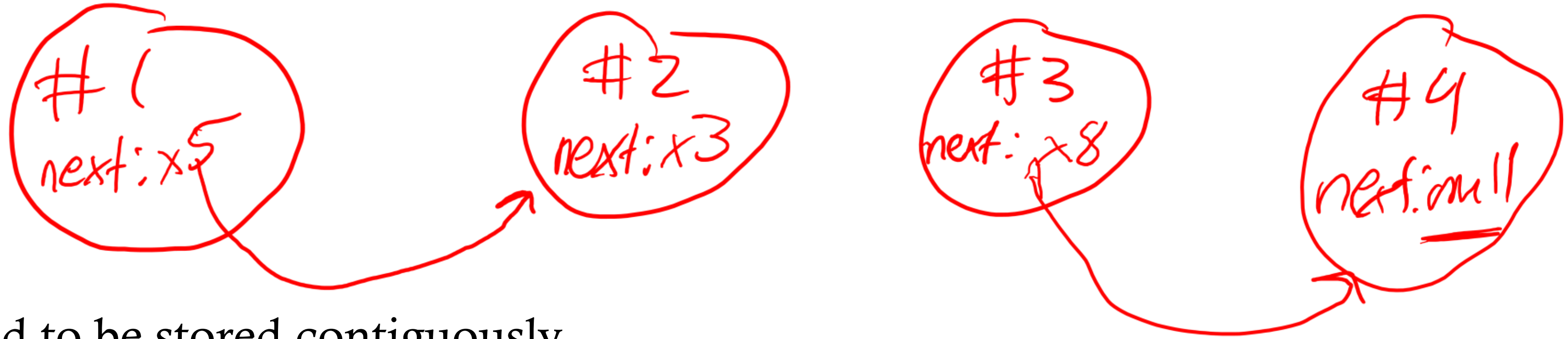
ARRAYLIST

get(2)
add(3, "hi")

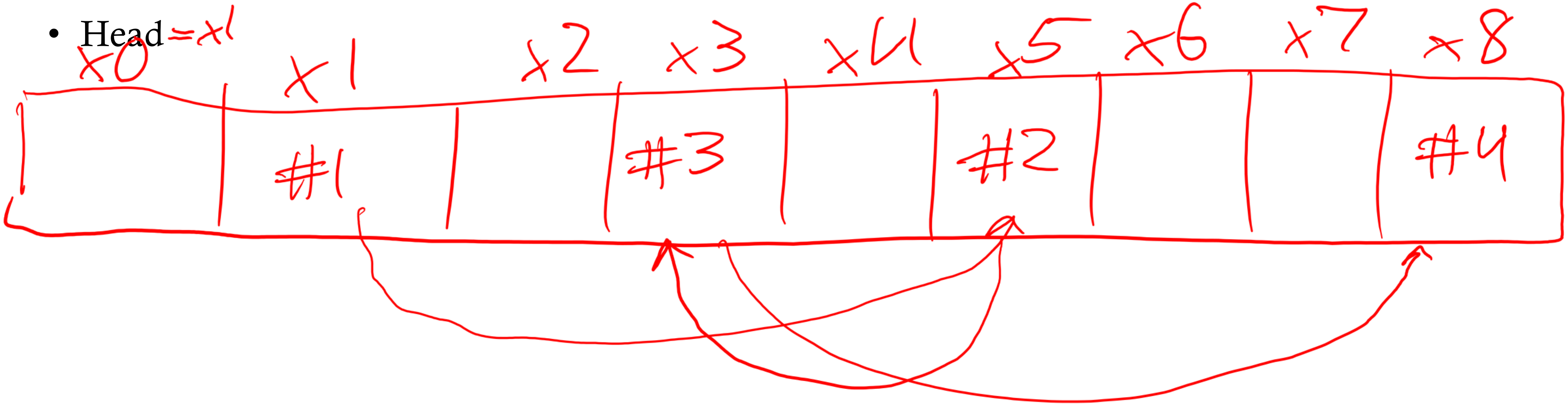
- Element references stored contiguously in memory
- Start = x4
- Size = 5

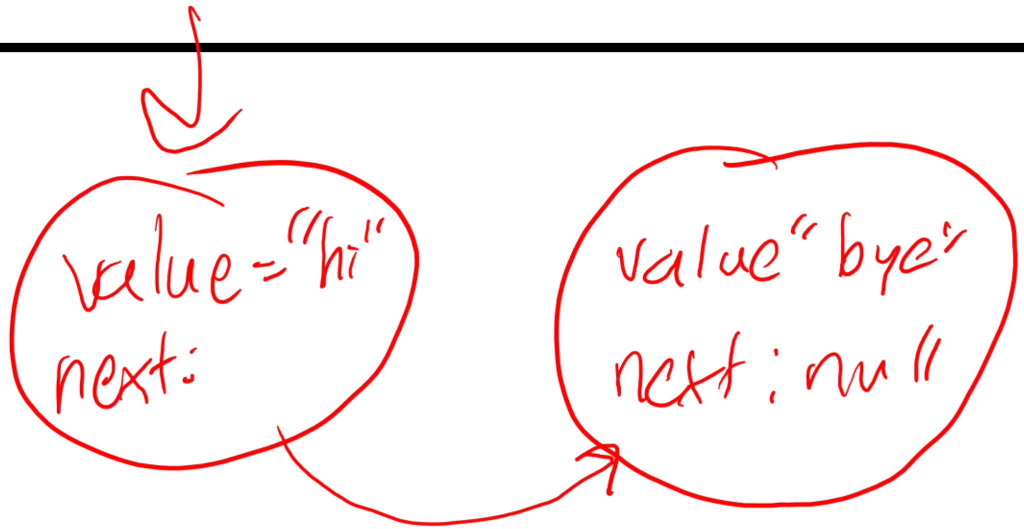


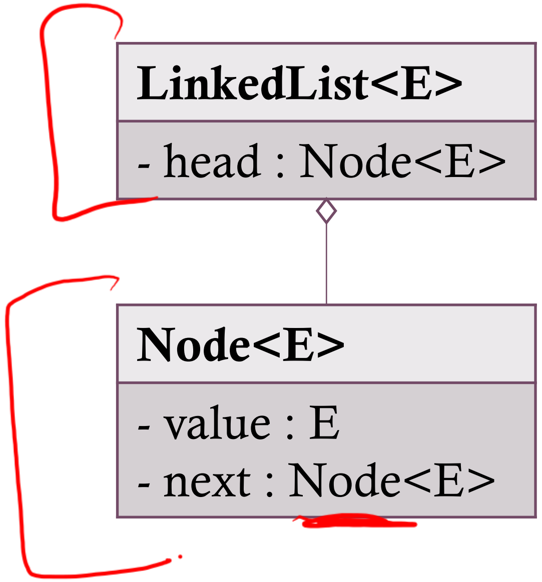
LINKEDLIST



- Element references don't need to be stored contiguously
- Each element holds a *link* or reference to the next item in the list
- Head = x1






LinkedList<E>

- head : Node<E>

Node<E>

- value : E
- next : Node<E>

LINKEDLIST

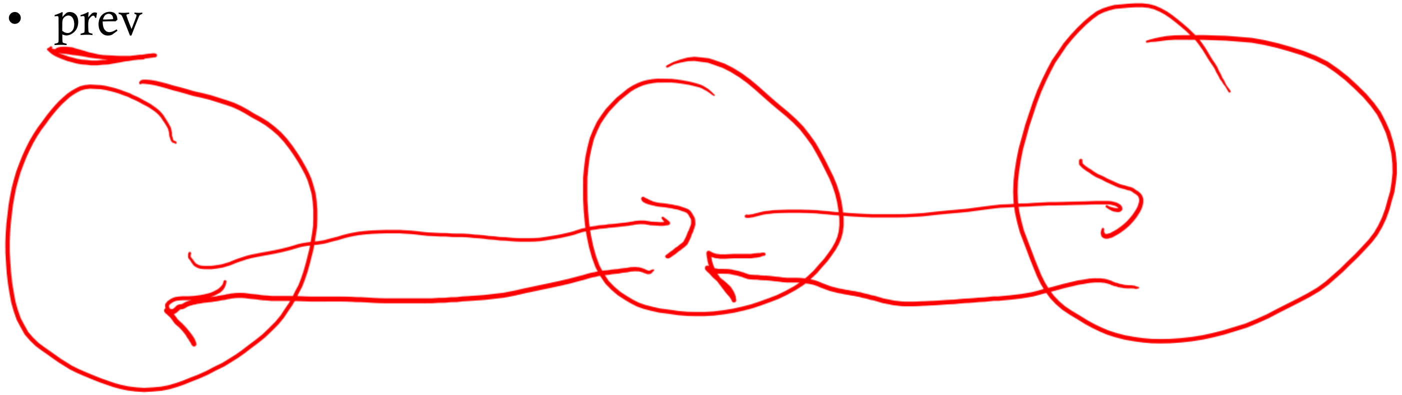
How do we implement our own LinkedList?

ACTIVITY

- Write one of the following methods for a LinkedList class
  - toString(): returns the contents of the list in String format
  - get(): takes in an index and returns the value of the node at that position
  - replace(): takes in an index and new value; replaces the value at the index position with the new value
- Make sure to consider if your method relies on any methods in the Node class

DOUBLELINKEDLIST

- Each node now has two pointers:
  - next
  - prev



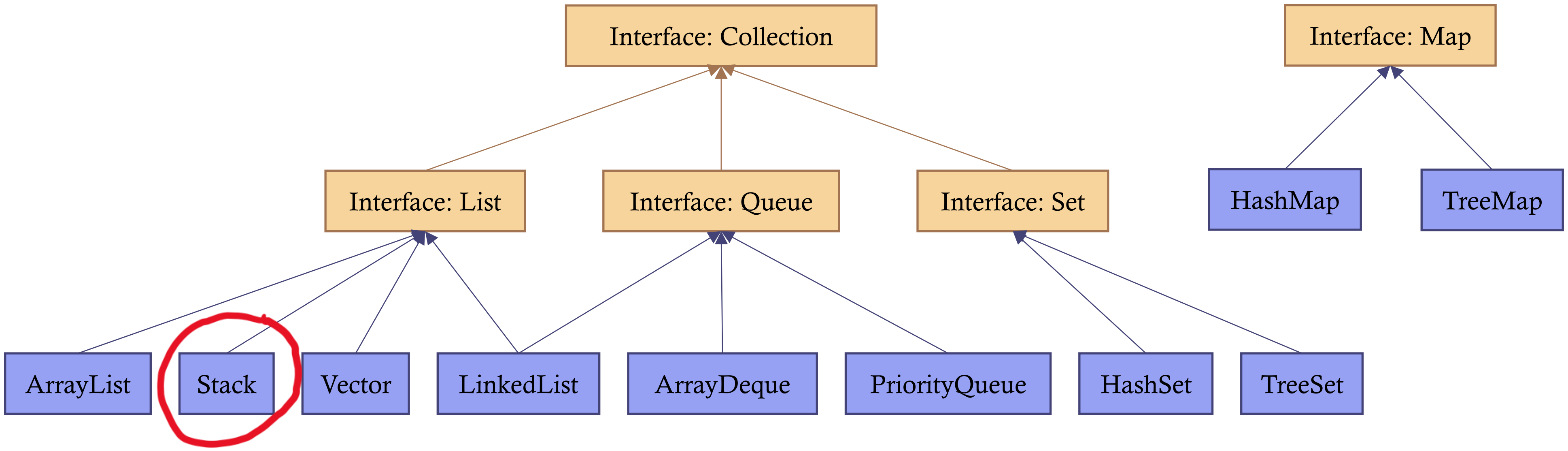
PROS AND CONS OF LIST IMPLEMENTATIONS

ArrayList

- Slow insert/remove
- **Efficient data access/storage**
- **Uses less memory**
- Less efficient memory usage

LinkedList

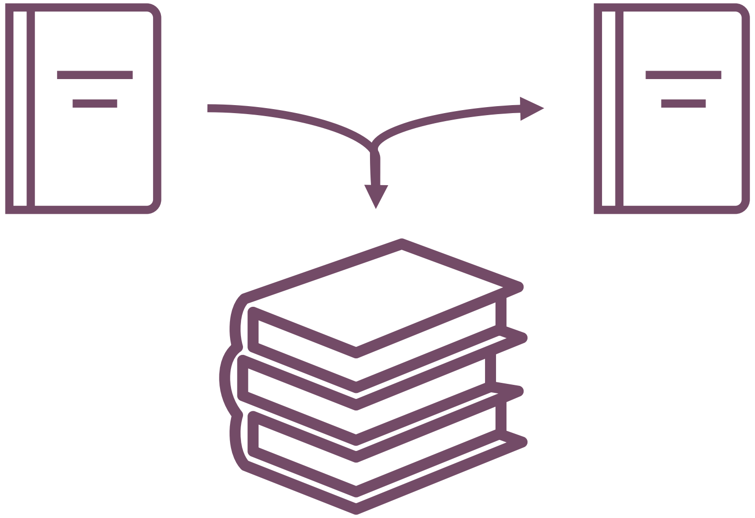
- **Fast insert/remove**
- Slow data access/storage
- Uses more memory
- **More efficient memory usage**


Interface: Collection
Interface: Map
Interface: List
Interface: Queue
Interface: Set
HashMap
TreeMap
ArrayList
Stack
Vector
LinkedList
ArrayDeque
PriorityQueue
HashSet
TreeSet

STACKS

STACK ADT

- Follows the Last In First Out philosophy (LIFO)
- Restricts modification to the “top” of the collection
- Could imagine as a stack of books
- Key methods:
  - push(): add
  - pop(): remove
  - peek(): view but don’t remove
  - contains()
  - clear()
  - isEmpty()



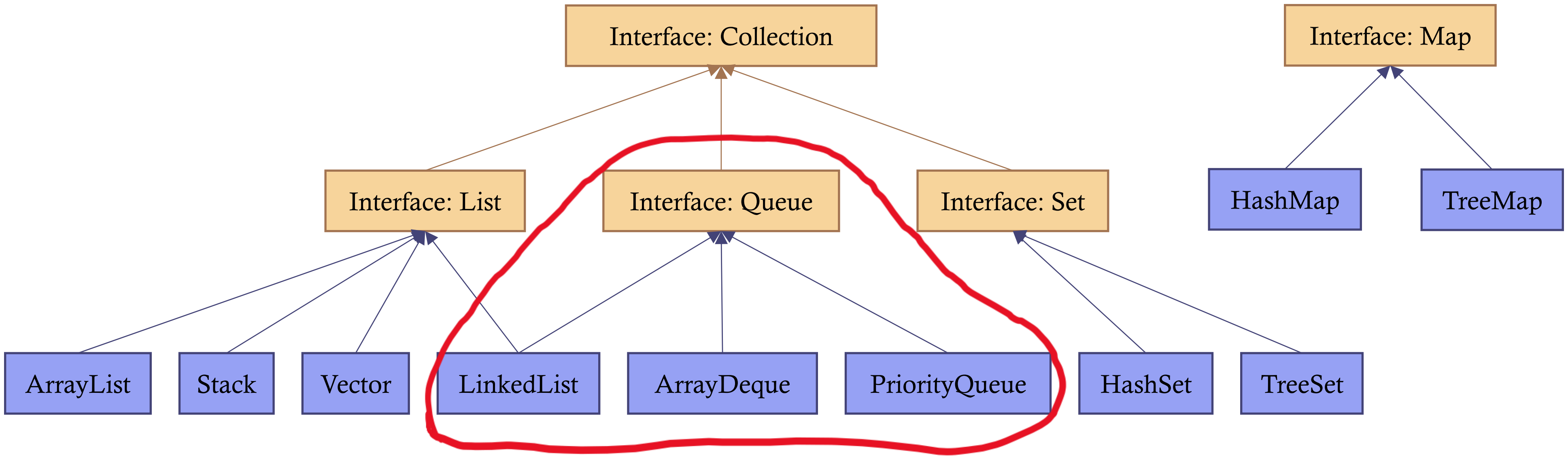
STACK USES

- `Stack<Integer> s = new Stack<>();`

- Reversing a set of commands (undo/redo)
- Returning to start (reversing a set of directions)
- Reversing a list

STACK ACTIVITY

- What are the contents of the stack after the following methods are run?
  - push(5)
  - push(3)
  - push(8)
  - pop()
  - peek()
  - pop()
  - push(1)
  - push(4)
  - pop()



QUEUES

- LinkedList
- ArrayDeque
- PriorityQueue

QUEUE ADT

- Follows the First In First Out philosophy (FIFO)
- Only adds to the front and removes from the back
- Restricts internal modification
- Could imagine as a line at a checkout
- Key methods:
  - offer(): add
  - poll(): remove
  - peek(): view but don’t remove
  - contains()
  - clear()
  - isEmpty()

QUEUE USES

- `Queue<Integer> q = new ArrayDeque<>();`
- `Queue<Integer> q = new LinkedList<>();`

- Sequence of customers to handle one by one
- Process tasks in the order they are received
- Schedule of items

QUEUE ACTIVITY

- What are the contents of the stack after the following methods are run?
  - offer(4)
  - offer(7)
  - offer(1)
  - poll()
  - poll()
  - offer(3)
  - offer(1)
  - peek()
  - offer(3)
  - poll()