# WEEK NINE

JAVA COLLECTION INTERFACE
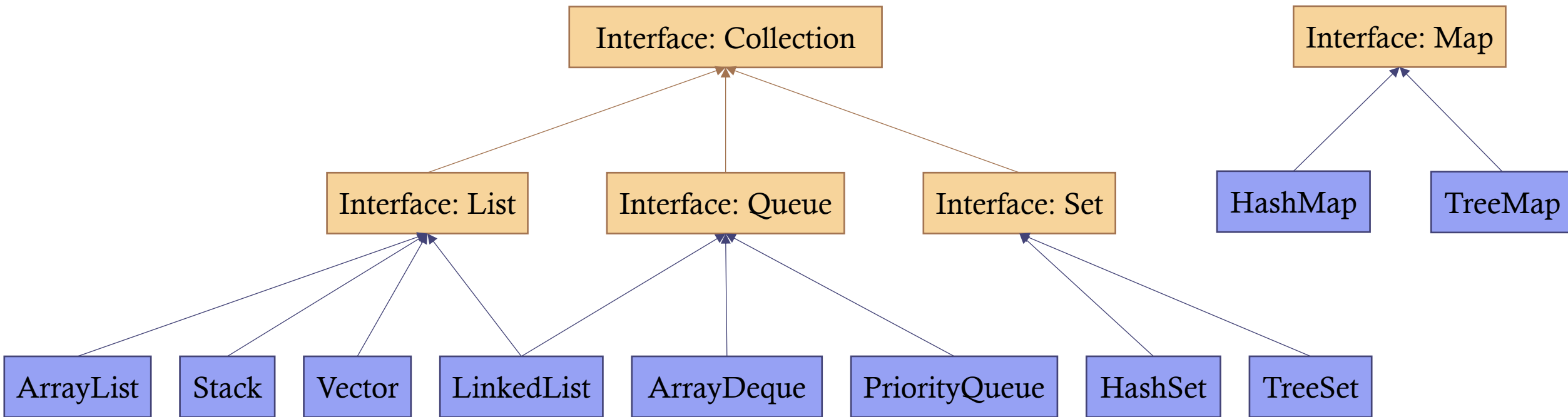
ABSTRACT DATA TYPES (ADTS)

- 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
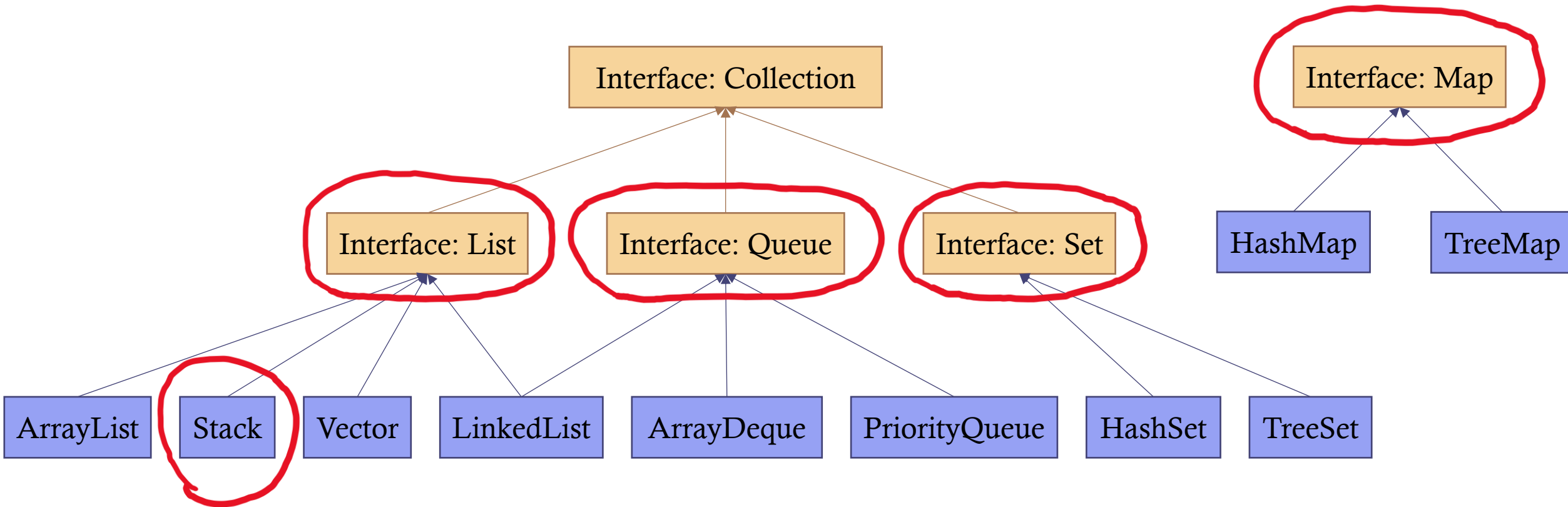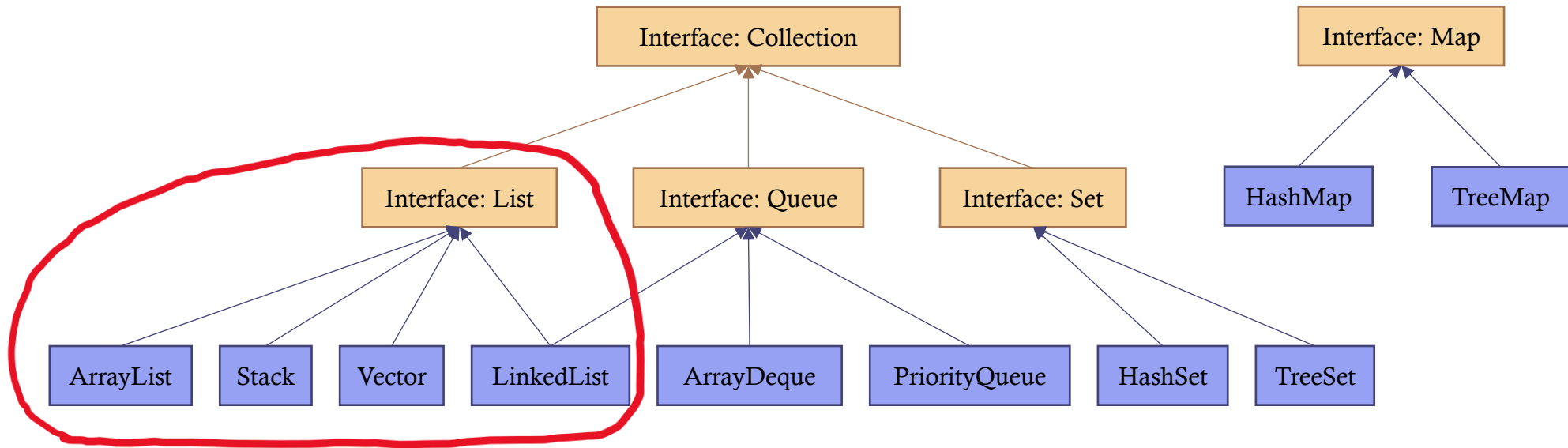
LISTS

- ArrayLists

- LinkedLists

LIST INTERFACE
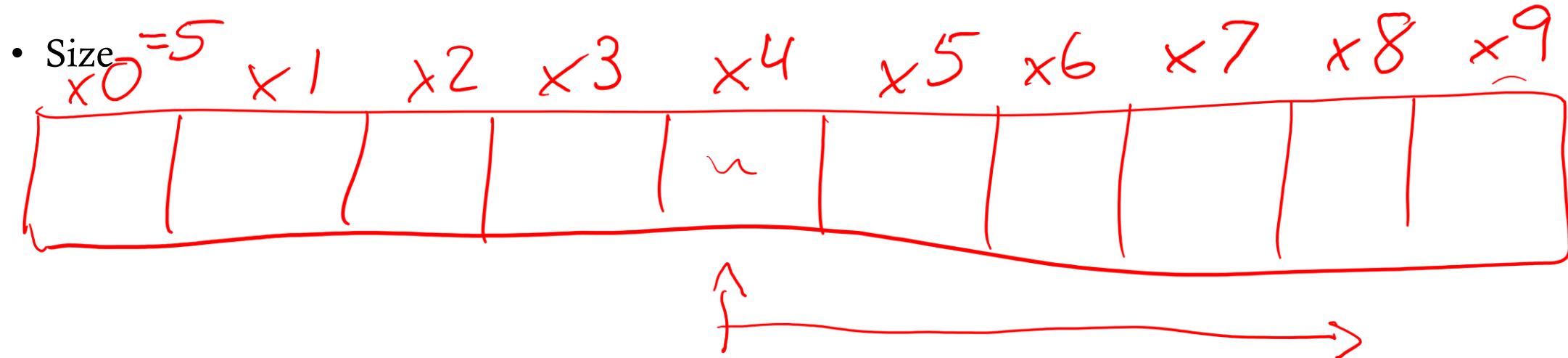
- An ordered, indexed collection

- Users can precisely insert/access elements via an index

- Duplicates typically allowed

- Key methods:
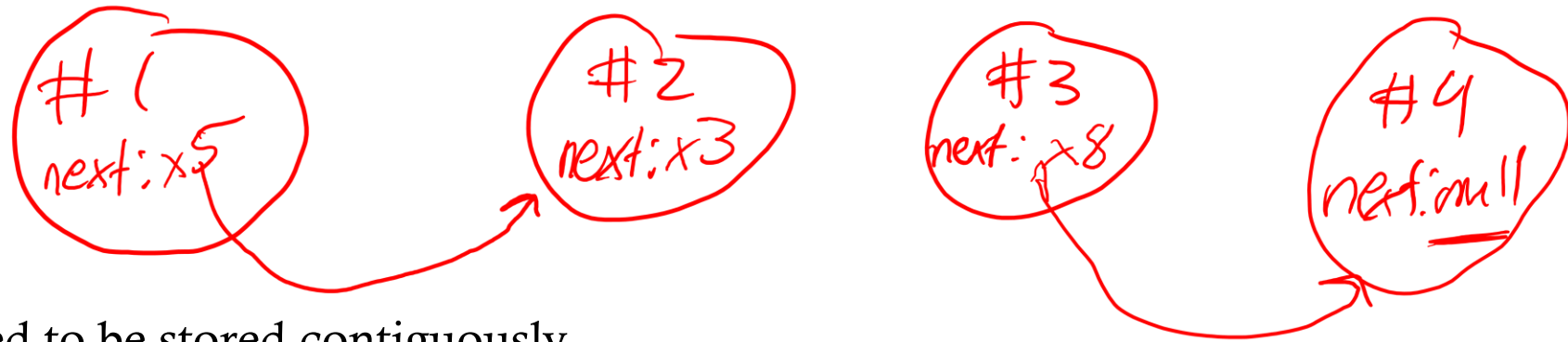 - add()
  - get()
  - remove()
  - isEmpty()

ARRAYLIST

get (2)

add (3, "hi")

- Element references stored contiguously in memory

- Start = $x4$

- Size = 5

x0   x1   x2   x3   x4   x5   x6   x7   x8   x9
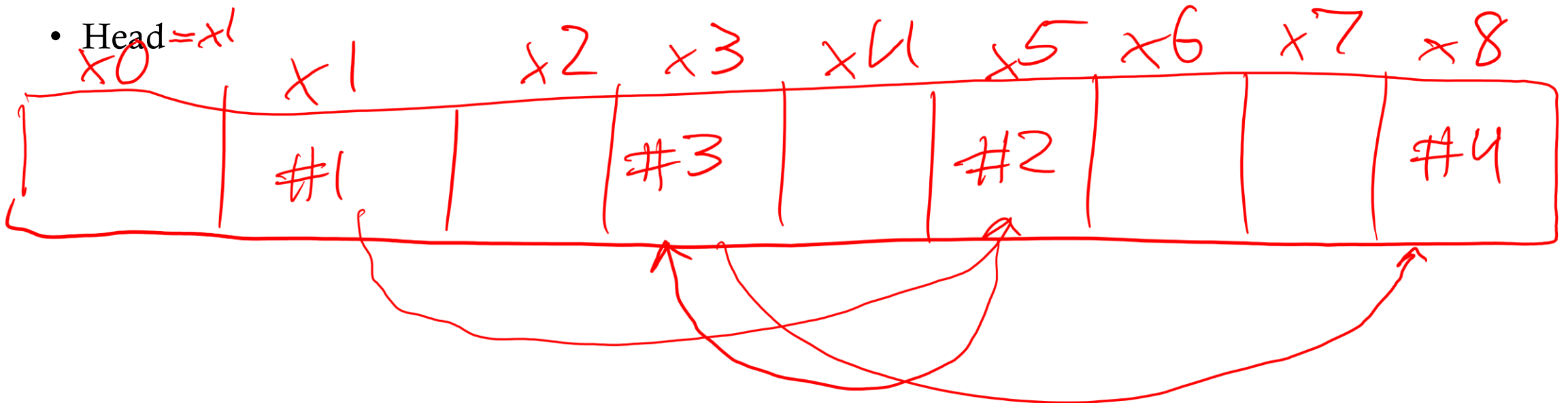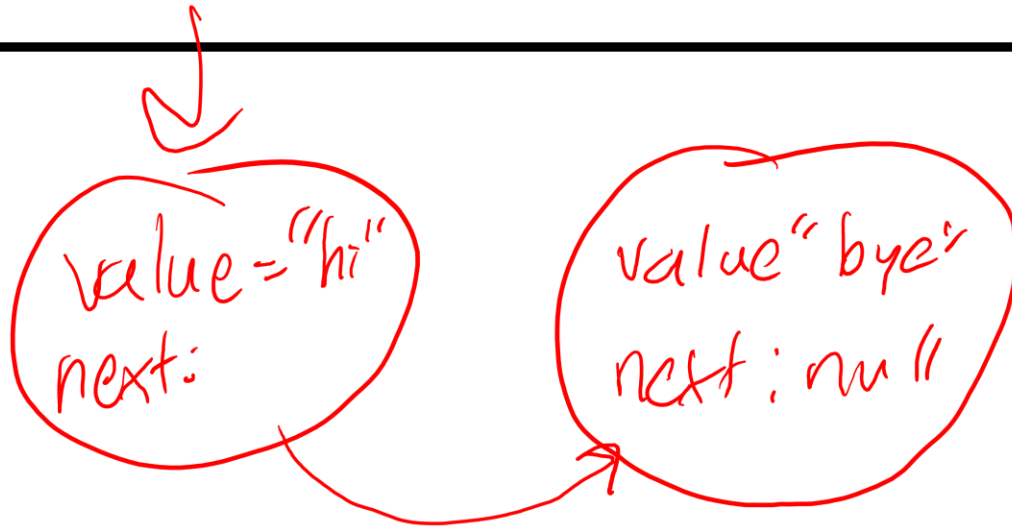
LINKEDLIST

- Element references don't need to be stored contiguously

- Each element holds a *link* or reference to the next item in the list

- Head

LINKEDLIST

How do we implement our own LinkedList?

value = "hi"
next:

value "bye"
next : null

LinkedList<E>

- head : Node<E>

Node<E>
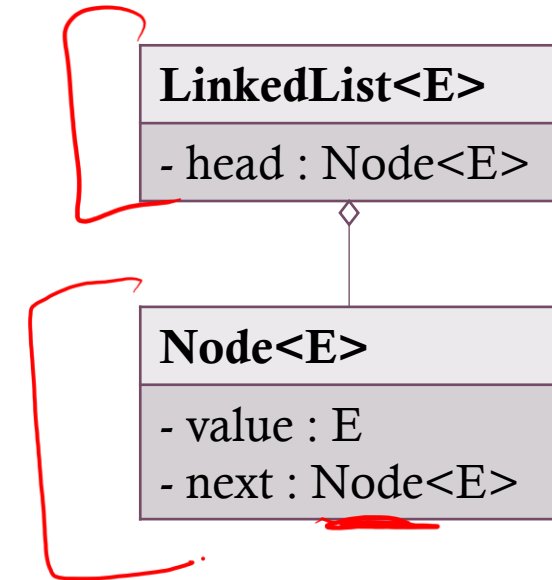
- value : E
- next : Node<E>

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
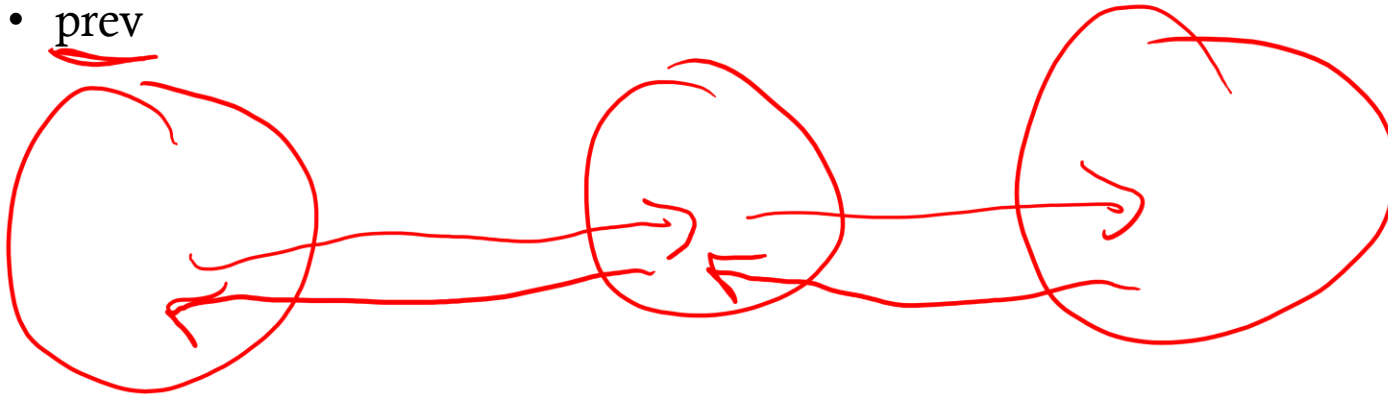
PROS AND CONS OF LIST IMPLEMENTATIONS

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**
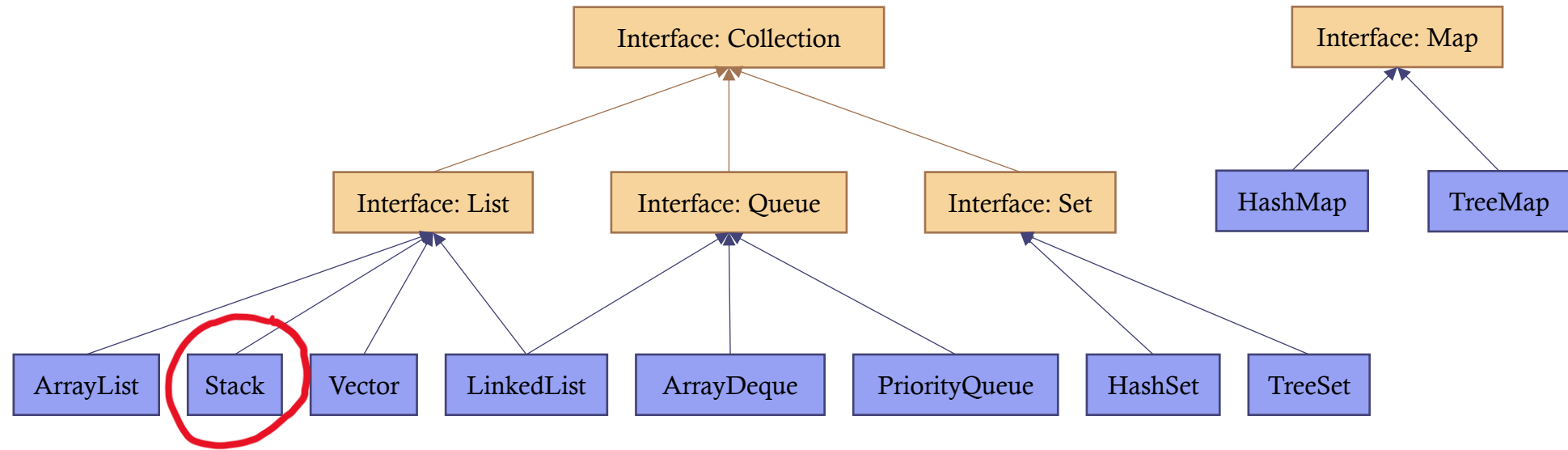
STACKS

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()
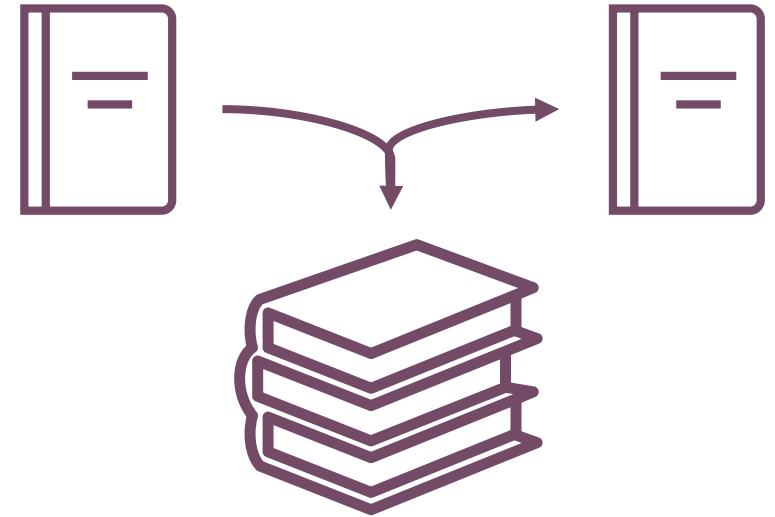
STACK USES

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

- Reversing a set of commands (undo/redo)

- Returning to start (reversing a set of directions)

- Reversing a list

- Deck of cards (draw/discard piles)

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()
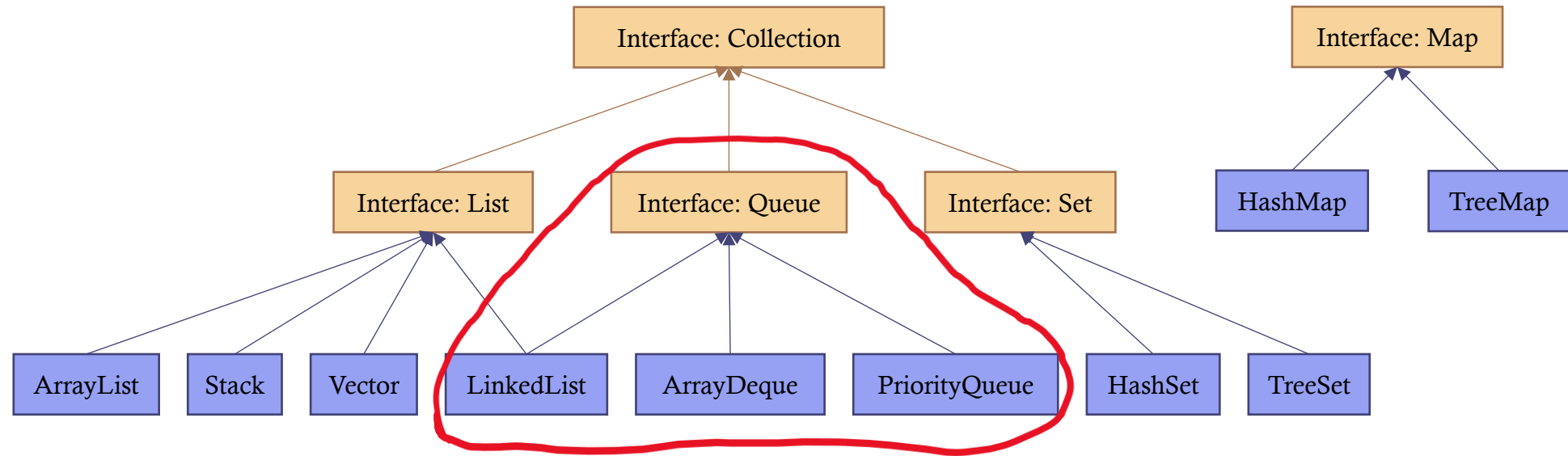
QUEUES

- 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()

PRIORITY QUEUES

- Polls items based on their "priority"
  - Essentially, we define the order that items are removed
  - Higher priority means the item will come first when sorted
  - **NOTE:** priority queues are not internally sorted
  - The only guarantee is that the next item that is polled is the next item according to the set priority
- Default ordering
  - Numbers sorted lowest first
  - Strings sorted alphabetically
  - Other types sorted via compareTo() or a comparator
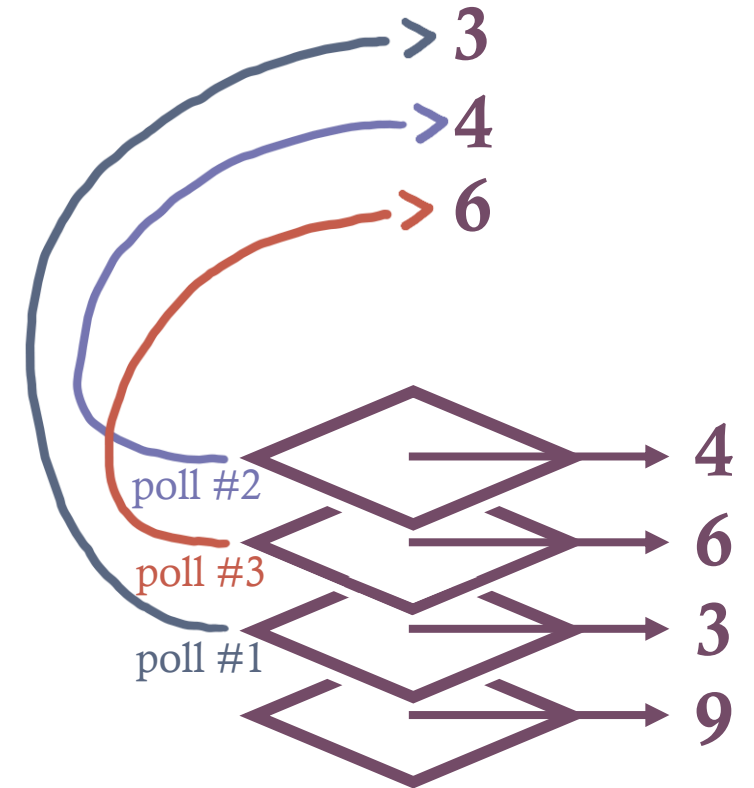
PRIORITY QUEUE USES

- `PriorityQueue<Integer> q = new PriorityQueue<>();`

- Customer assistance order (call-ahead/scheduled customers skip the line)

- Grabbing word bank words in alphabetical order

- Banking system set to handle larger transactions first

ACTIVITY

- Create a system to manage tasks

- You will need to create a very simple Task class
  - Field for a description of the Task
  - Field for the Task priority level
  - Method to allow sorting via priority

- Main class should have the following functionality
  - Ability to grab the highest priority Task currently in the list
  - Ability to "complete" Tasks and move them to a structure for completed items
  - Ability to retrieve the most recently completed Task
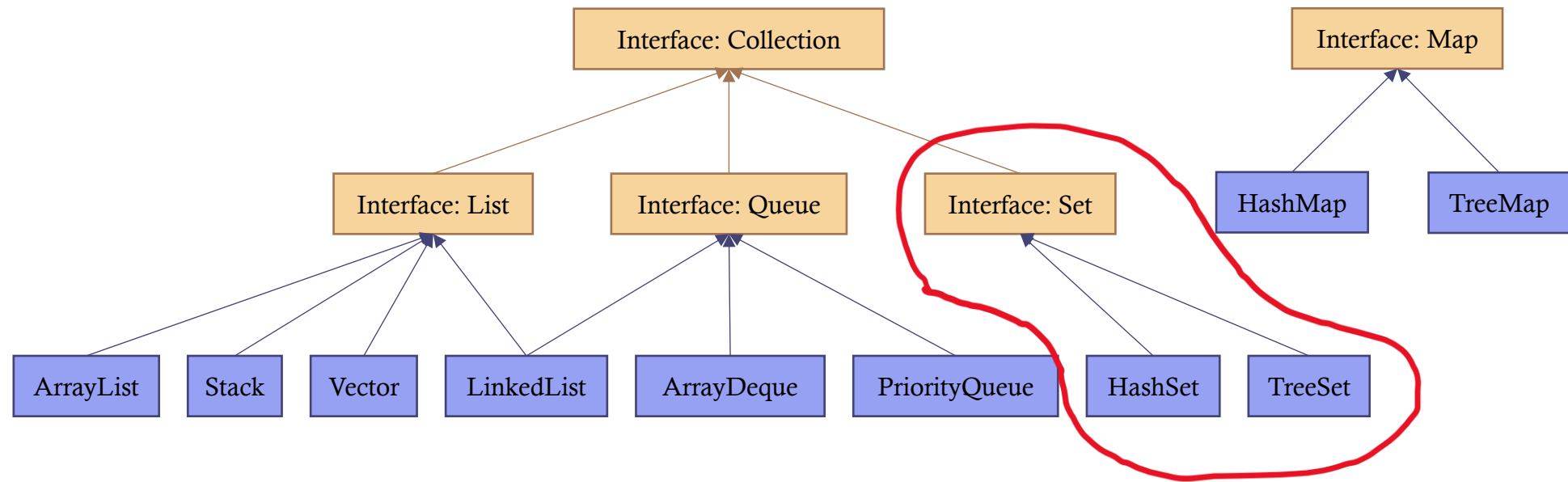
SETS

- HashSet

- TreeSet

SET INTERFACE

- Modeled after mathematical sets

- No order

- No duplicates allowed

- Primary purpose is to determine whether an element is in a specific set

- Key methods:
  - add()
  - remove()
  - isEmpty()
  - size()

63
6   75
     94
  37
28   22
  19
     46   85

"black"
"yellow"
"purple"
"orange"
"red"
"blue"
"green"
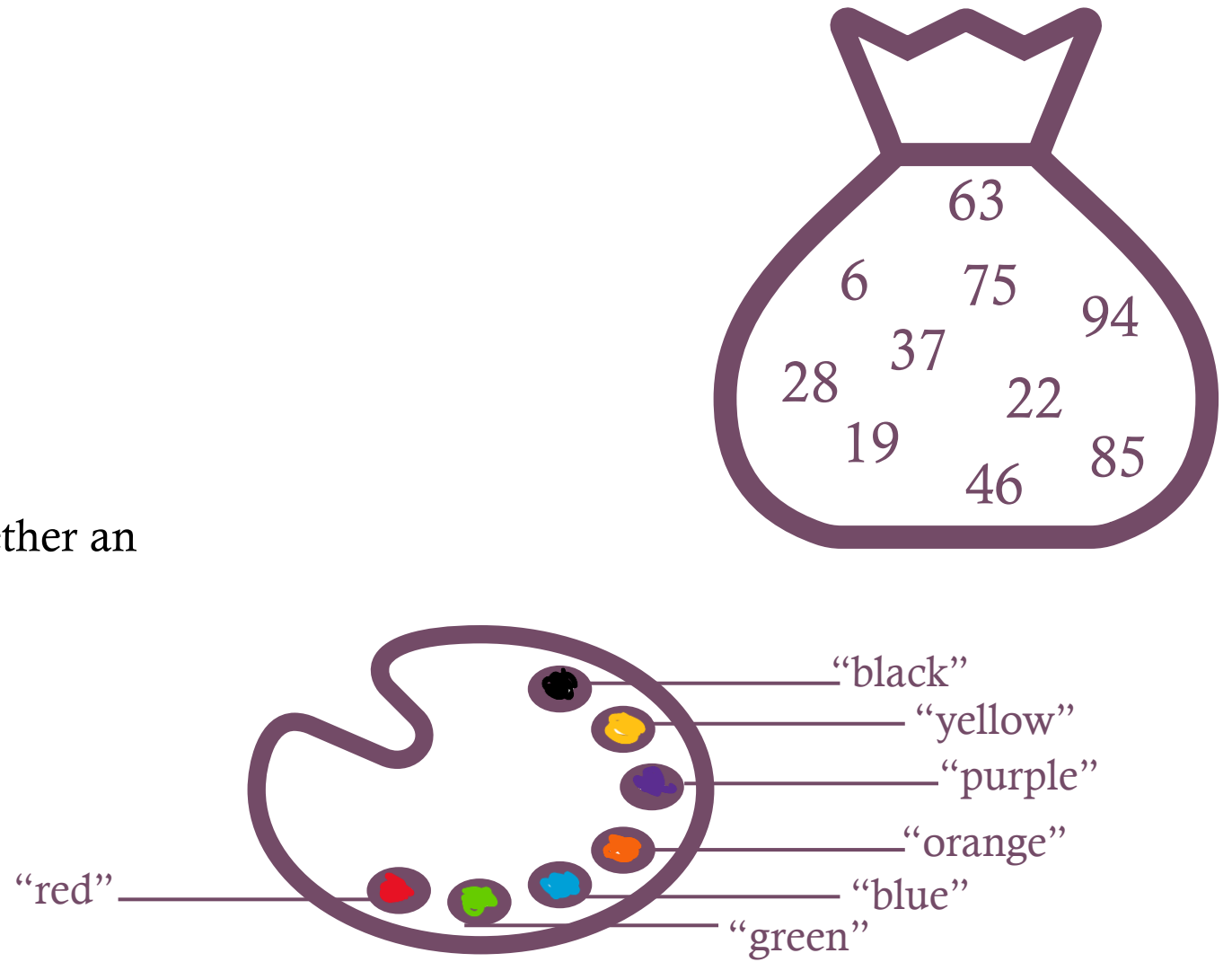
SET USES

- Powerball

- Check for duplicate letters in a word

- Roster of UIDs

- Color palette

HASHSET

- `Set<Integer> s1 = new HashSet<>();`

- Based on a high-performance data structure called a hash table
  - Hash is calculated for the element to be stored using a key
  - *Hash % value* is performed to determine which "bucket" to place the element in
  - Behind the scenes, structure is optimized to have few empty buckets and to minimize number of elements per bucket
  - Since key is known, hash and corresponding bucket can easily be determined when searching for an element
  - Then only that bucket needs to be searched

TREESET

- `Set<Integer> s2 = new TreeSet<>();`

- Based on a tree data structure
  - Elements are naturally ordered
  - Elements are linked to each other in a tree format
  - For this tree implementation, each element may only have two child nodes
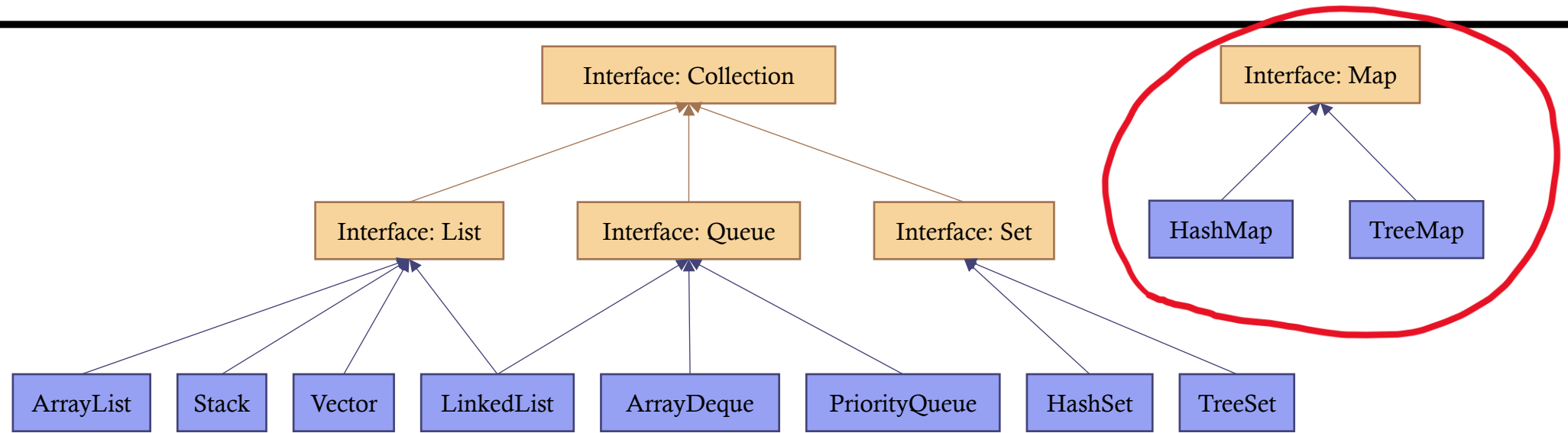
MAPS

- HashMap

- TreeMap

MAP INTERFACE

- Maps keys and values

- Essentially works like a look-up table

- Keys must be unique and only connect to one value

- Values may be duplicated

- Key methods:
  - put() – provide with key and value
  - get() – provide with a key
  - remove() – provide with a key
  - keySet() – returns a Set!
  - values() – returns a Collection

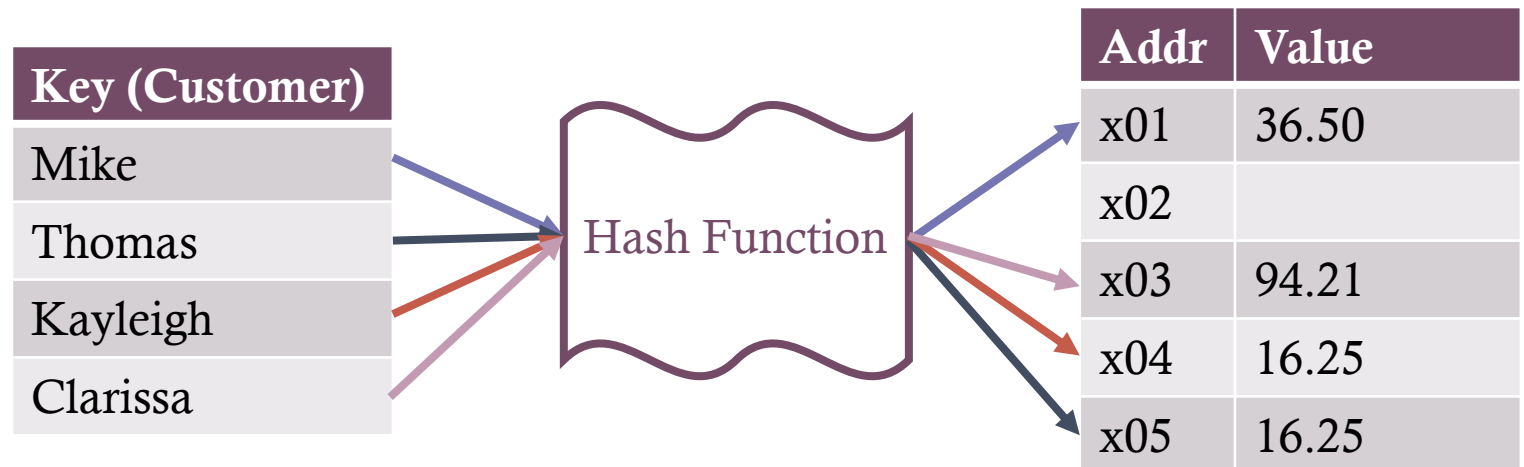| Key (Word)	Value (Count)
"the"	7
"quick"	2
"brown"	2
"fox"	1
"jumps"	5

MAP USES

- Read a text file and count the number of uses of each word

- Dictionary: pairing words with their definitions

- Student name and number of credit hours

- Customer and total money spent

HASHMAP & TREEMAP

- `Map<Integer, String> checkOuts1 = new HashMap<>();`

- `Map<Integer, String> checkOuts2 = new TreeMap<>();`

- Underlying implementations are similar to Sets
  - Hash table
  - Naturally ordered tree

Key (Customer)
Mike
Thomas
Kayleigh
Clarissa

Hash Function

Addr	Value
x01	36.50
x02	
x03	94.21
x04	16.25
x05	16.25

ITERABLE & ITERATOR INTERFACES

Allows us to step through a data structure implementation

WHY DO WE NEED BOTH?

- `Iterable:`
  - Requires implementation of: `Iterator<E> iterator()`
  - Forces you to create an `Iterator` as well
  - Allows a way to pass our class's information to the `Iterator`
  - Benefits of polymorphism as the class itself is now `Iterable`

- `Iterator:`
  - Requires implementation of: `boolean hasNext(), E next()`
  - Allows implementations of the iteration functionality
  - Needs to be implemented via a separate class so multiple `Iterator` instances can exist at once (think nested loops)

ITERABLE IMPLEMENTATION

```java
public class BingoMachine <T extends Number> implements Iterable<T>{
    ...
    public Iterator<T> iterator(){
        return new BingoMachineIterator();
    }
}
```

ITERATOR CLASS AND CONSTRUCTOR

```
public class BingoMachineIterator implements Iterator<T>{
    private ArrayList<Integer> order;
    private int nextItem;

    public BingoMachineIterator(){
        nextItem = 0;
        order = new ArrayList<Integer>();
        for (int i = 0; i < contents.size(); i++){
            order.add(i);
        }
        Collections.shuffle(order);
    }
}
```

ITERATOR METHODS

```java
public class BingoMachineIterator implements Iterator<T>{
    private ArrayList<Integer> order;
    private int nextItem;

    public boolean hasNext(){
        return (nextItem < (contents.size()));
    }

    public T next(){
        return contents.get(order.get(nextItem++));
    }
}
```