ET0736

Lesson 6

Java Collections Framework

Topics

- Java Collection Framework
- Collection Interface – List, Queue and Set
- List – ArrayList, Vector, LinkedList and Stack
- Queue – ArrayQueue and PriorityQueue
- Set – HashSet, LinkedHashSet and TreeSet
- Map Interface – HashMap, LinkedHashMap and TreeMap

Java Collections Framework

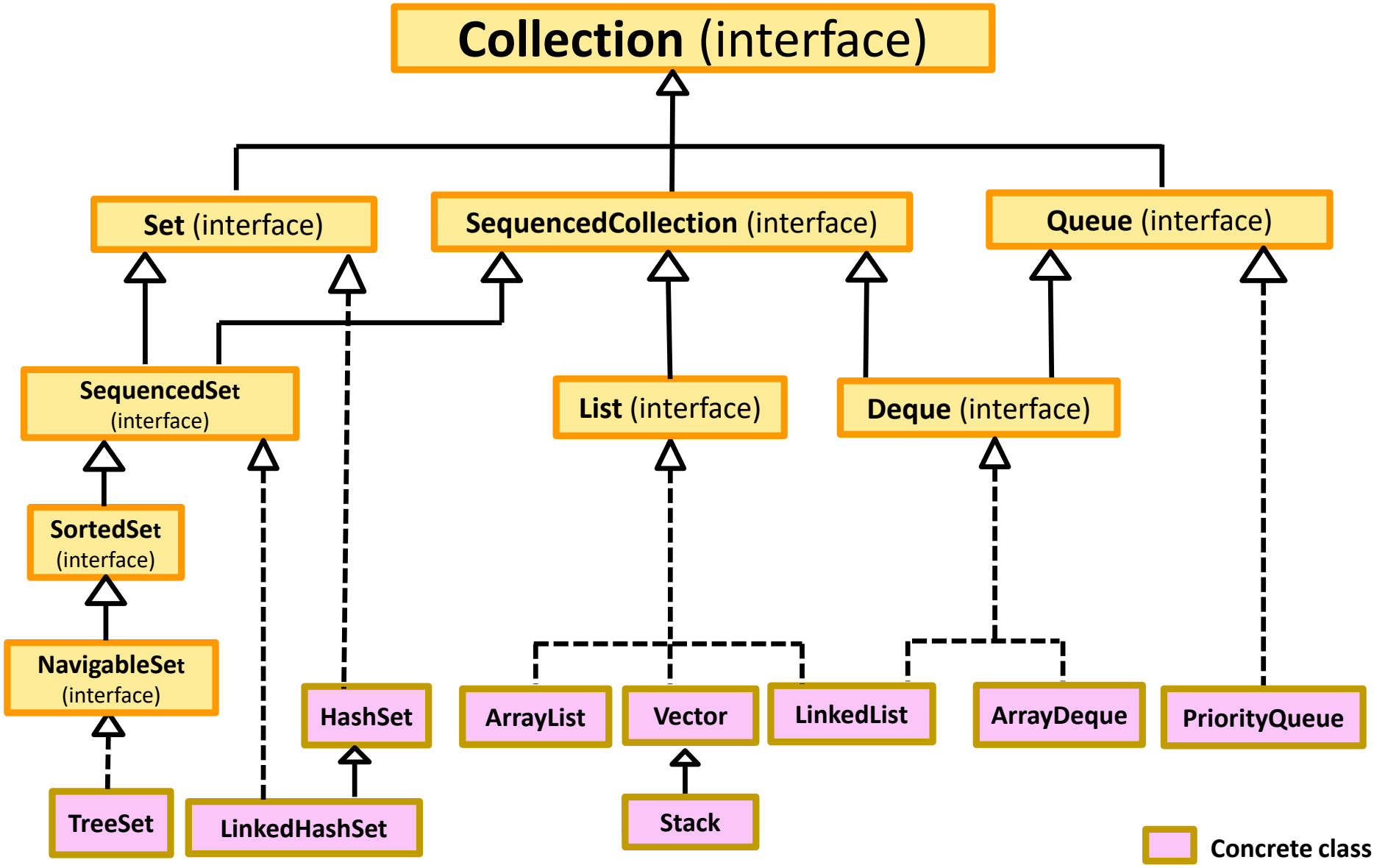
Supports 2 types of containers

- Collection – stores elements
- Map – stores key-value pairs

Collection Interface

- It is one of the root interface in Java Collections Framework
- It is not directly implemented by any classes
- It is indirectly implemented via its sub-interfaces
    - List – store an ordered collection of elements
    - Queue – store elements in first-in-first-out manner
    - Set – store a group of non-duplicate elements

Simplified Collection Interface

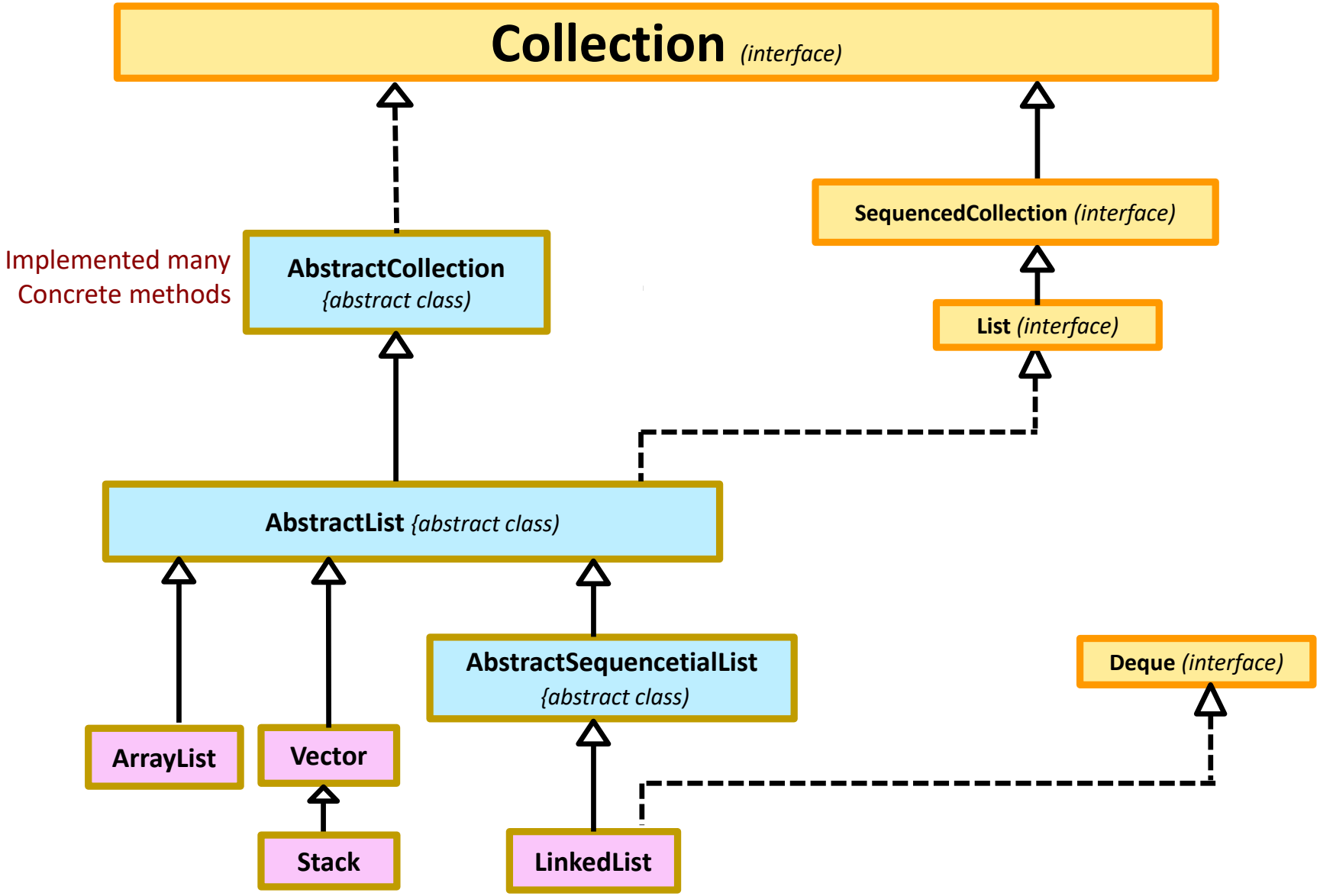


Concrete Classes in Collection

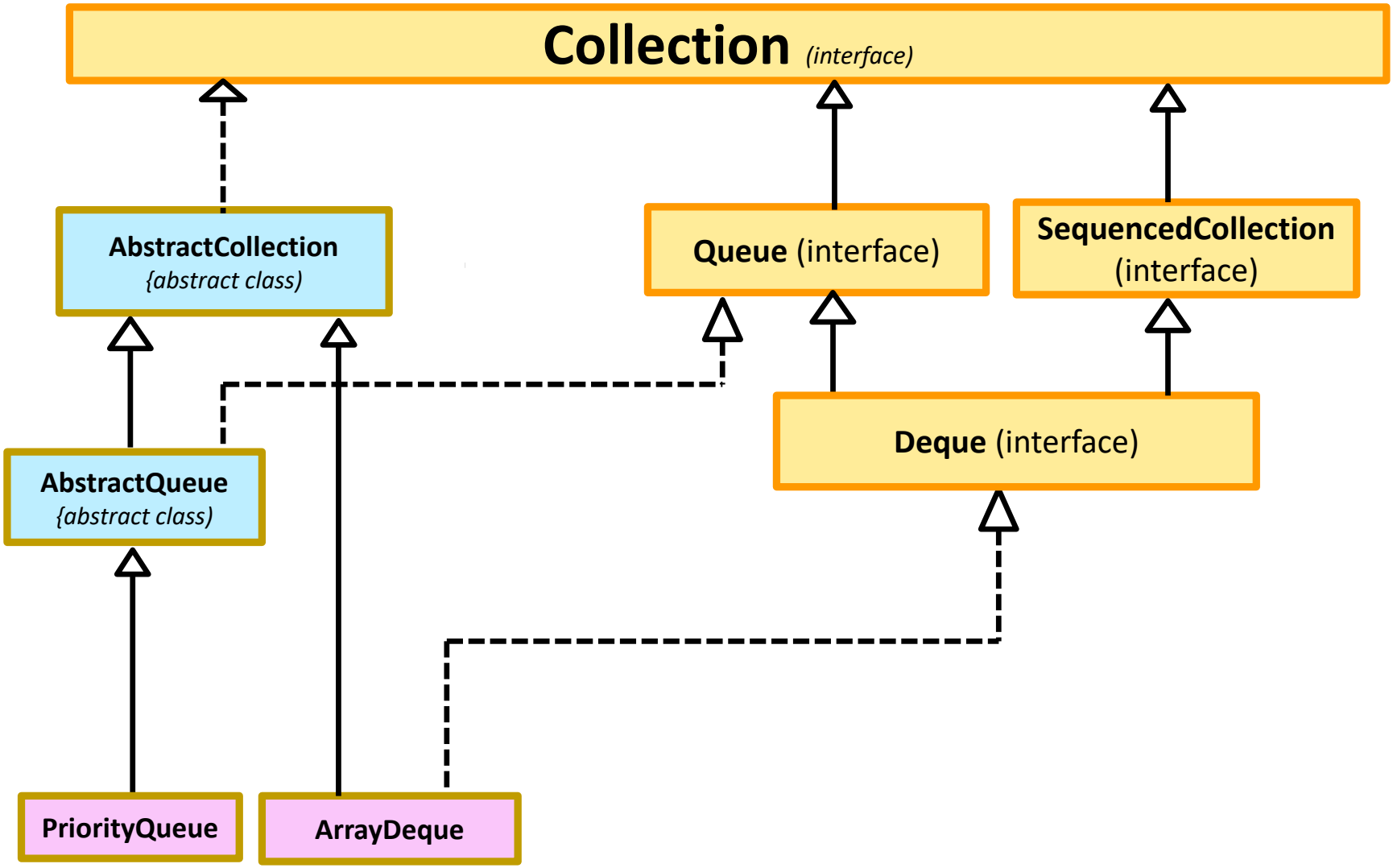
In fact, the following concreate classes of data structures in Java are direct subclasses of some other intermediate abstract classes:

- ArrayList, Vector, LinkedList and Stack
- ArrayQueue and PrioityQueue
- HashSet, LinkedHashSet and TreeSet

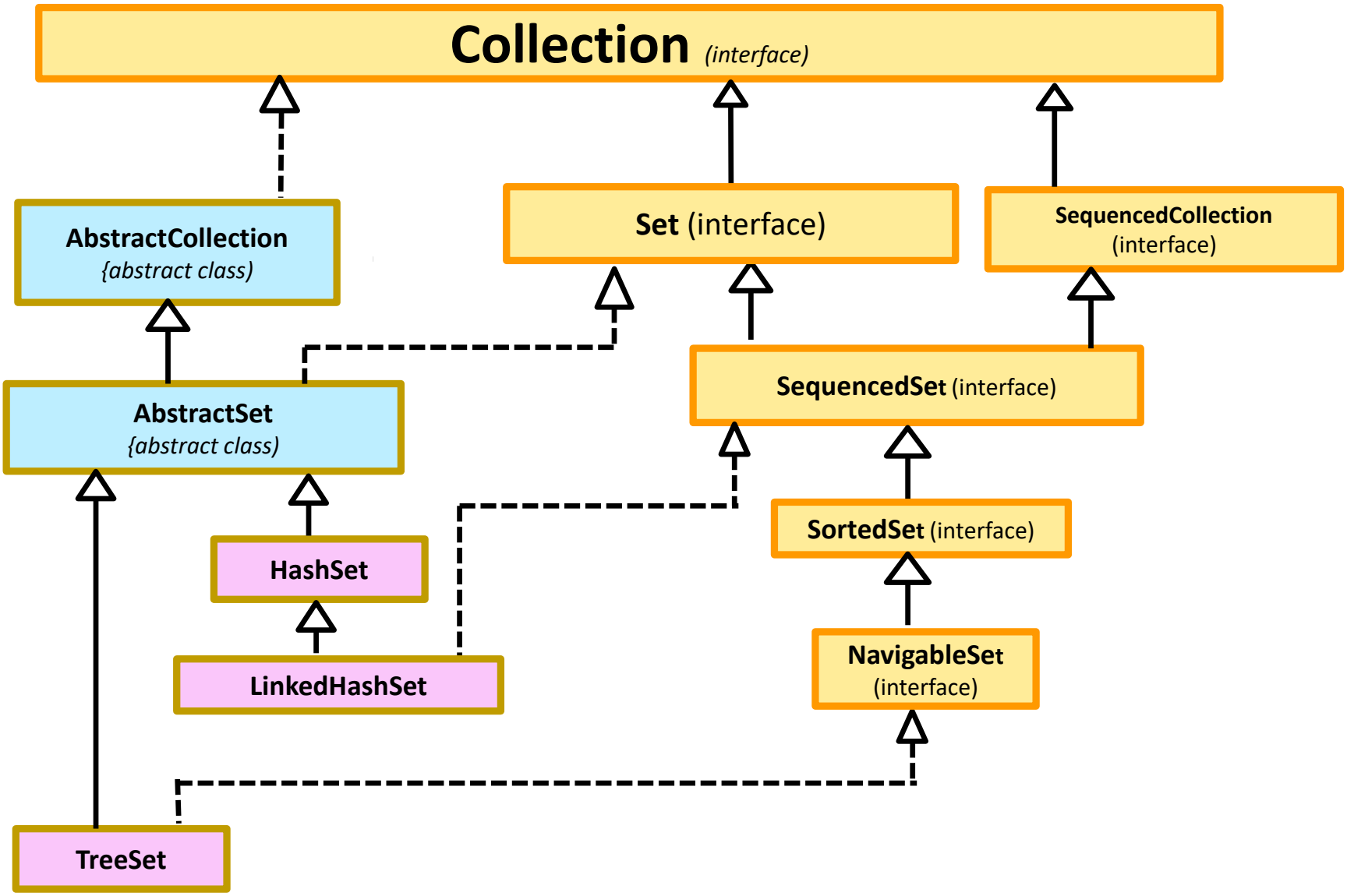
Details of classes – ArrayList, Vector, LinkedList



Details of classes – ArrayQueue, priorityQueue



Details of classes – HashSet, LinkedHashSet, TreeSet

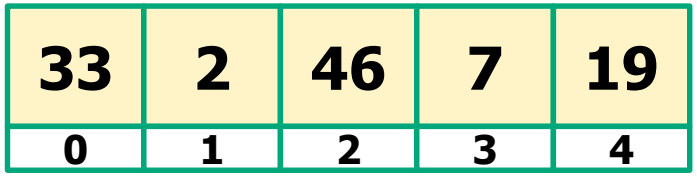


Classes implement List interface

- ***ArrayList*** – dynamic array (non-synchronised)
- ***Vector*** – legacy container, dynamic array (synchronised)
- ***Stack*** (sub class of Vector) – supports pop/peek/push operations
- ***LinkedList*** - dynamic and stores elements in non-consecutive manner in memory

ArrayList - concept

- It is basically resizable array.
- Elements are stored according to the natural order, the order in which the elements are added
- Elements are stored in consecutive memory locations

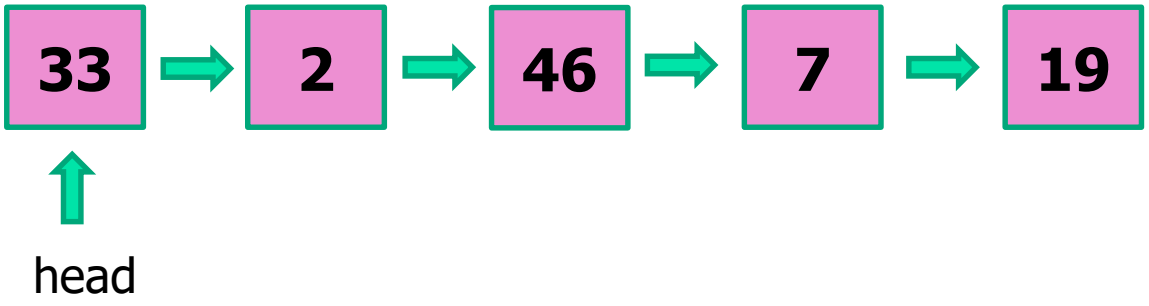


Linked List - concept

- An array has its elements stored in consecutive memory space, with index running from 0 and pointing to each element.

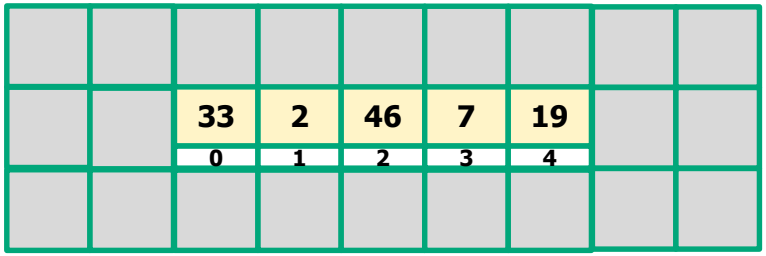
33	2	46	7	19
0	1	2	3	4

- A linked list has its elements spread out in memory in non-consecutive locations, with a head pointing to the 1st node and in turns the 1st node points to the 2nd node, and so on. Hence, index is irrelevant.

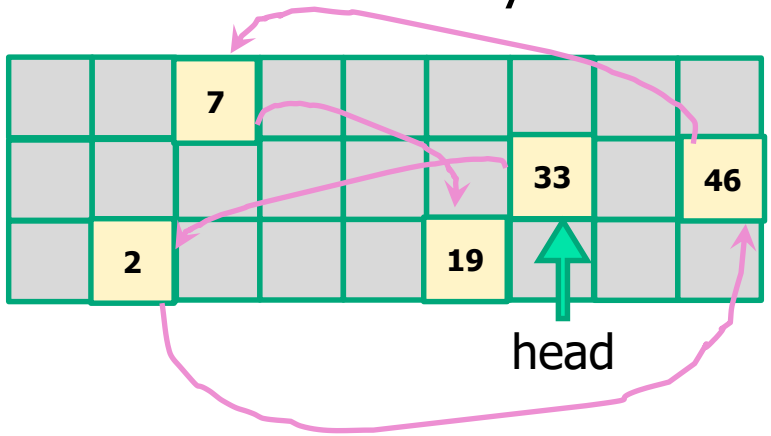


Main differences (from Array)

- So, array in memory

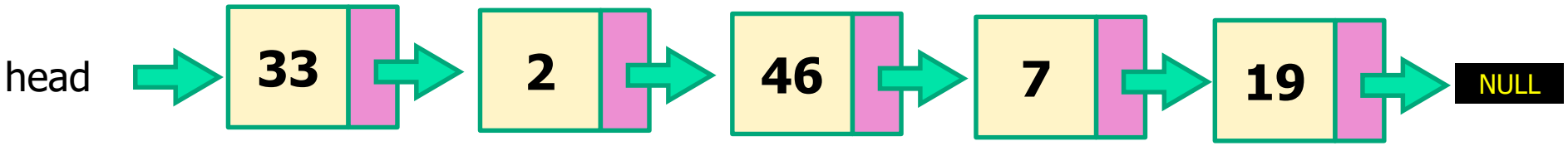


- So, nodes in linked list are everywhere in memory:



Chain of Nodes

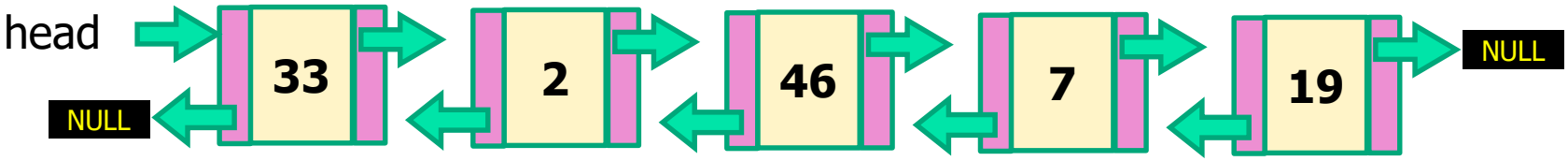
- Every node in a linked list contains:
  - Data
  - Reference to next node



- Last node in a linked list contains:
  - next = null

Doubly Linked List - concept

- Every node in a linked list contains:
  - Data
  - Reference to next node as well as *previous* node



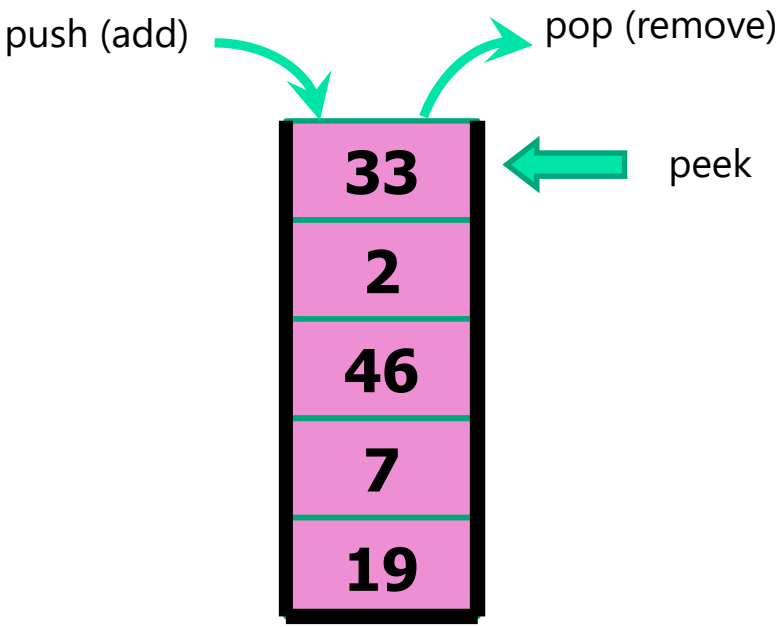
- Last node in a doubly linked list contains:
  - next = null
- First node in a doubly linked list contains:
  - previous = null

LinkedList class in Java

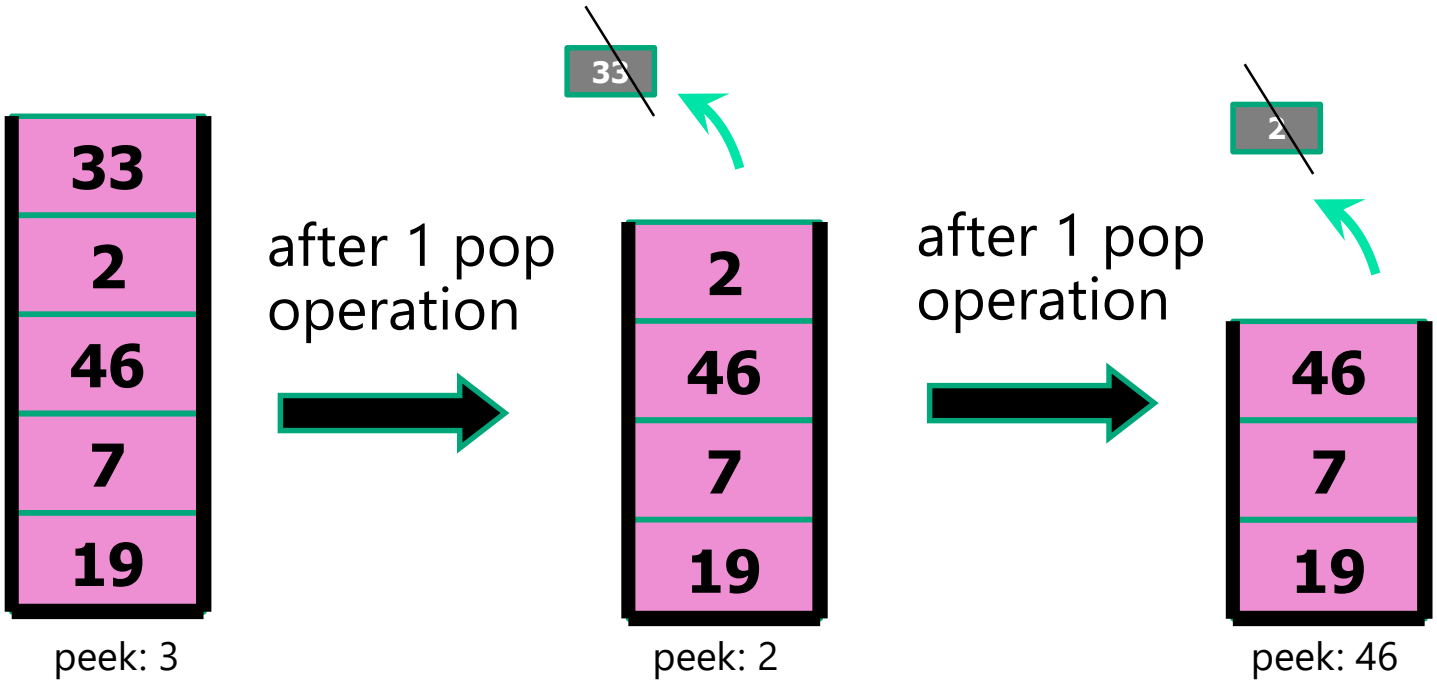
- ***LinkedList*** class is a part of the Collection framework presents in java.util package.
- It is also a linear data structure
- Internally, the ***LinkedList*** is implemented using the doubly linked list data structure.
- A ***LinkedList*** can contain the same or different object types.
- A ***LinkedList*** is efficient in removing/adding elements at both ends.
- A ***LinkedList*** is not efficient in removing/adding elements in between due to tracing (aka *pointer-chasing*) overhead
- A ***LinkedList*** does not need to move existing elements when removing/adding elements (advantage over ArrayList)
- Overall efficiency***, ArrayList*** usually out performed.

Stack - concept

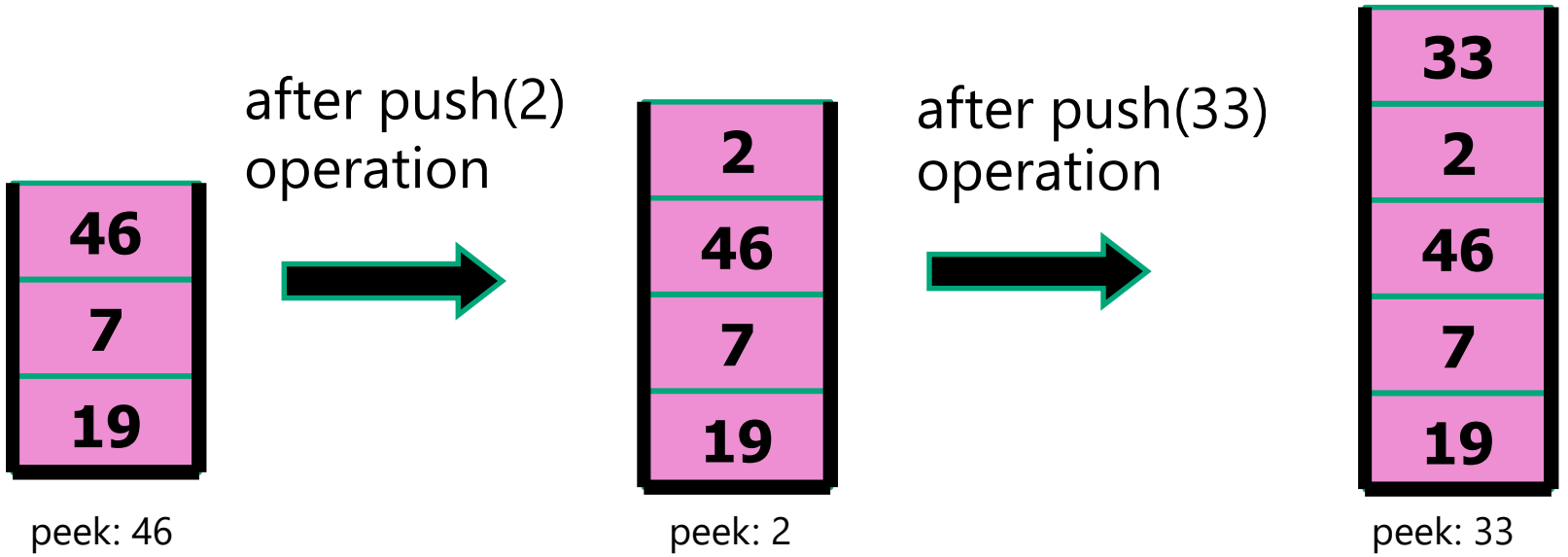
- The stack is a linear data structure that is used to store the collection of objects in a Last-In-First-Out (LIFO) manner.
- Data can only be added/removed from the ‘top’ of the stack



Stack example (logical)

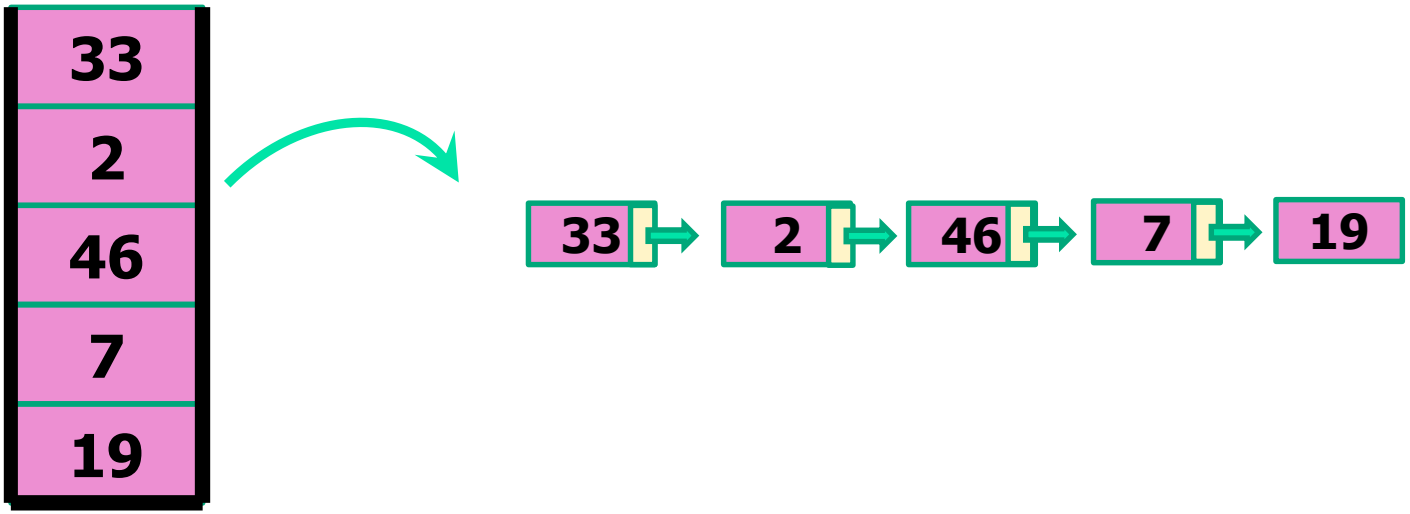


Stack example (logical)



Stack from LinkedList

- A simple stack can be implemented using concept of linked list:



Stack class in Java

- Extends from Vector class
- Hence, internally, it is implemented using dynamic array (in contiguous memory locations) and not linked list
- Many methods implemented in the Java Stack class are very similar to that of ArrayList and LinkedList.
- However, in general, the reason for choosing Stack is more for the FILO nature.

Methods in List Classes

Some of the common methods among

ArrayList, Vector, LinkedList, Stack

```
add(E): boolean
add(int index, E): void
addFirst(E): void
addLast(E): void
```

```
remove(E): boolean
remove(int index): void
removeFirst(E): void
removeLast(E): void
```

```
get(int index): E
getFirst(): E
getLast(): E
```

```
size(): int
isEmpty(): boolean
contains(E): Boolean
clone(): object
```

More methods just for

LinkedList, Stack

```
peek(): E
pop(): E
push(E): void
poll(): E
```

Added Sep 2023

addFirst(), addLast(), removeFirst(), removeLast(), getFirst(), getLast()

More methods just for

LinkedList

```
peekFirst(): E
peekLast(): E
pollFirst(): E
pollLast(): E
```

** LinkedList inherited these from 2nd interface, the Deque interface*

Creating List Objects

Examples:

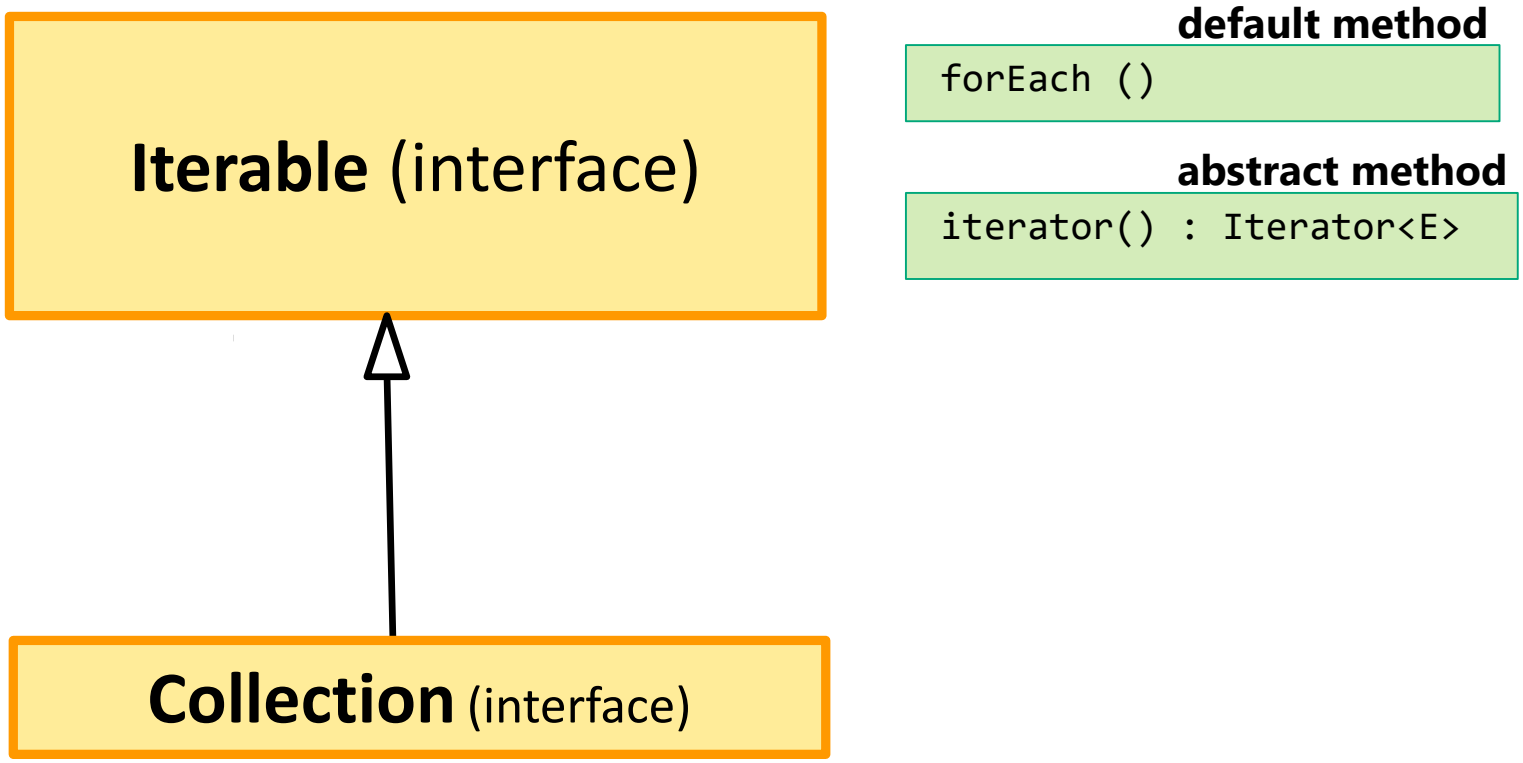
```
ArrayList<Integer> a  = new ArrayList<>();
Vector<Double> b  = new Vector<>();
LinkedList<String> c = new LinkedList();
Stack<Circle> d = new Stack();
```

```
Collection<String> e = new ArrayList<>();
Collection<Integer> f  = new LinkedList<>();
Collection<Member> g = new Vector<>();
Collection<Circle> h = new Stack<>();
```

```
List<String> i = new ArrayList<>();
List<Integer> j  = new LinkedList<>();
List<Member> k = new Vector<>();
List<Circle> m = new Stack<>();
```

Iterable Interface

Collection is a sub interface of Iterable interface.



Iterator

- Is a classic design pattern for navigating through a data structure.
- Useful methods: ***hasNext()*** and ***next()***

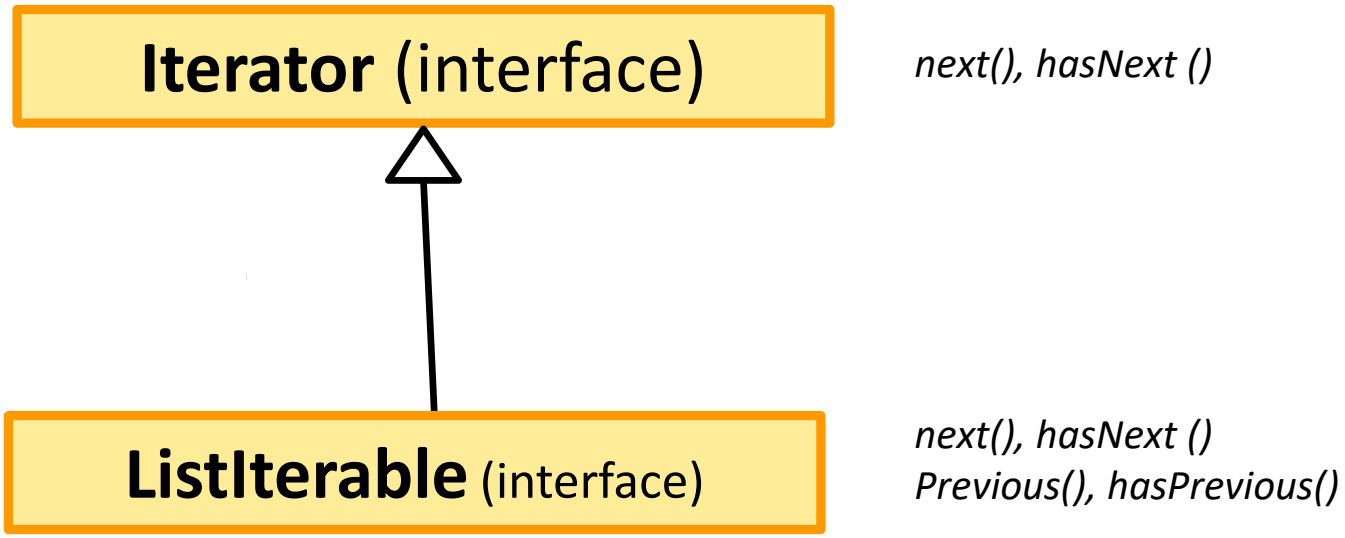
```
ArrayList<String> a  = new ArrayList<>();
a.add("Ang Mo Kio");
a.add("Dover");
a.add("Changi");

Iterator<String> it = a.iterator();
while (it.hasNext()) {
    System.out.println(it.next().toUpperCase());
}
```

Output:

ANG MO KIO
DOVER
CHANGI

Iterator and ListIterator Interface

- ***ListIterator*** extends from ***Iterator*** interface
- Allows bidirectional traversal of the list
- ***List*** interface has methods that returns Iterator and also ListIterator objects



ListIterator

- Extends from Iterator
- Provides bidirectional traversal of the list
- Useful methods: ***hasNext(), next(), hasPrevious and Previous***

```
ArrayList<String> a  = new ArrayList<>();
a.add("Ang Mo Kio");
a.add("Dover");
a.add("Changi");

ListIterator<String> it = a.listIterator();
while (it.hasNext()) {
     System.out.println(it.next().toUpperCase());
}
System.out.println("---------");
while (it.hasPrevious()) {
     System.out.println(it.previous().toLowerCase());
}
```

Output:

```
ANG MO KIO
DOVER
CHANGI
---------
changi
dover
ang mo kio
```

forEach

- Inherited from interface Iterable

```
ArrayList<String> a  = new ArrayList<>();
a.add("Ang Mo Kio");
a.add("Dover");
a.add("Changi");

a.forEach(e->System.out.println(e.toUpperCase()));
```

Lambda expression

Output:
ANG MO KIO
DOVER
CHANGI

Using Stack Class

Example: Create an ***Stack*** to store Student objects from the top.
Method used: ***push() or add()***

```
import java.util.*;
public static void main(String[] args) {
    Stack<Student> a = new Stack<Student>();
    a.push(new Student("KK"));
    a.push(new Student("bobo"));
    a.add(new Student("tutu"));
    System.out.println(a);
}
```

```
class Student {
    String name;
    Student (String s) { name=s; }
    @Override
    public String toString() { return(name); }
}
```

Output:

[KK, bobo, tutu]

Using Stack Class

Example: Check the top element (without removing).

Method used: ***peek()***

```
import java.util.*;
public static void main(String[] args) {
    Stack<Student> a = new Stack<Student>();
    a.push(new Student("KK"));
    a.push(new Student("bobo"));
    a.add(0,new Student("tutu"));
    System.out.println(a.peek());
}
```

```
class Student {
    String name;
    Student (String s) { name=s; }
    @Override
    public String toString() { return(name); }
}
```

Output:

bobo

Using Stack Class

Example: Remove the top element

Method used: ***pop()***

```
import java.util.*;
public static void main(String[] args) {
    Stack<Student> a = new Stack<Student>();
    a.push(new Student("KK"));
    a.push(new Student("bobo"));
    a.add(0,new Student("tutu"));
    System.out.println(a.pop());
    System.out.println(a);
}
```

```
class Student {
    String name;
    Student (String s) { name=s; }
    @Override
    public String toString() { return(name); }
}
```

Output:

Bobo
[tutu,kk]

Using Stack Class

Example: insert an item.

Method used: ***add(index, Object)***

```
import java.util.*;
public static void main(String[] args) {
    Stack<Student> a = new Stack<Student>();
    a.push(new Student("KK"));
    a.push(new Student("bobo"));
    a.add(0, new Student("tutu"));
    System.out.println(a);
}
```

```
class Student {
    String name;
    Student (String s) { name=s; }
    @Override
    public String toString() { return(name); }
}
```

Output:

[tutu, KK, bobo]

Using ListIterator for Stack

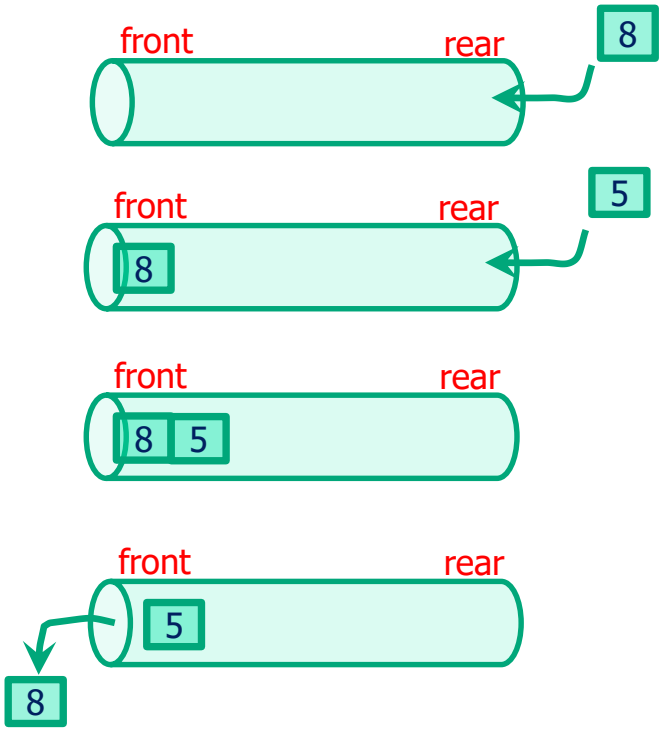
Similarly, navigation in a Stack can be done with a ListIterator

```
import java.util.*;
public static void main(String[] args) {
    Stack<Student> a = new Stack<Student>();
    a.push(new Student("KK"));
    a.push(new Student("bobo"));
    a.add(0,new Student("tutu"));
    ListIterator it = a.listIterator();
    while (it.hasNext()) { System.out.println(it.next()); }
}
```

```
class Student {
    String name;
    Student (String s) { name=s; }
    @Override
    public String toString() { return(name); }
}
```

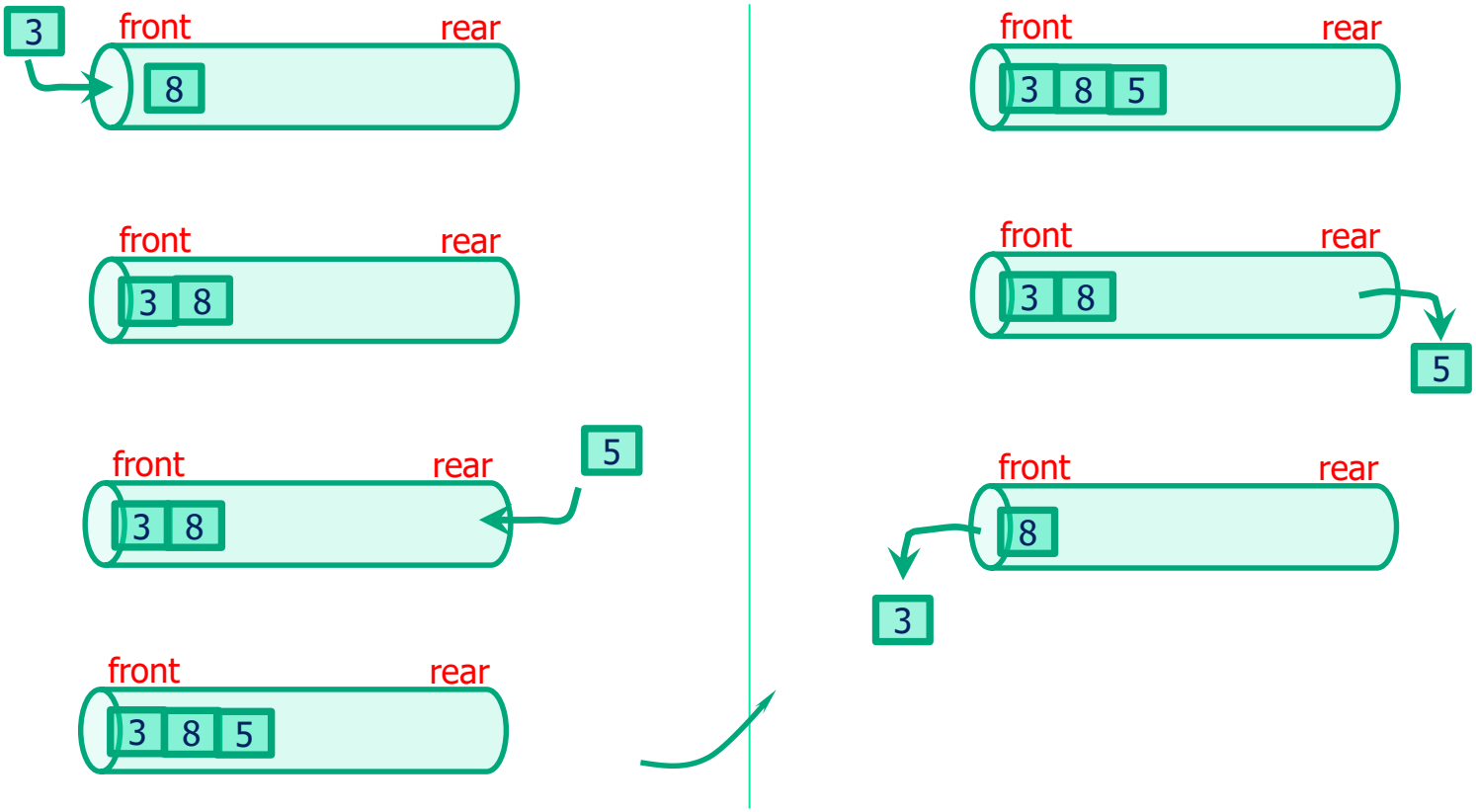
Queue - concept

- First-In-First-Out protocol – just like a line in front of a hawker stall
- Elements are inserted at the end of the queue
- Elements are removed from the front of the queue



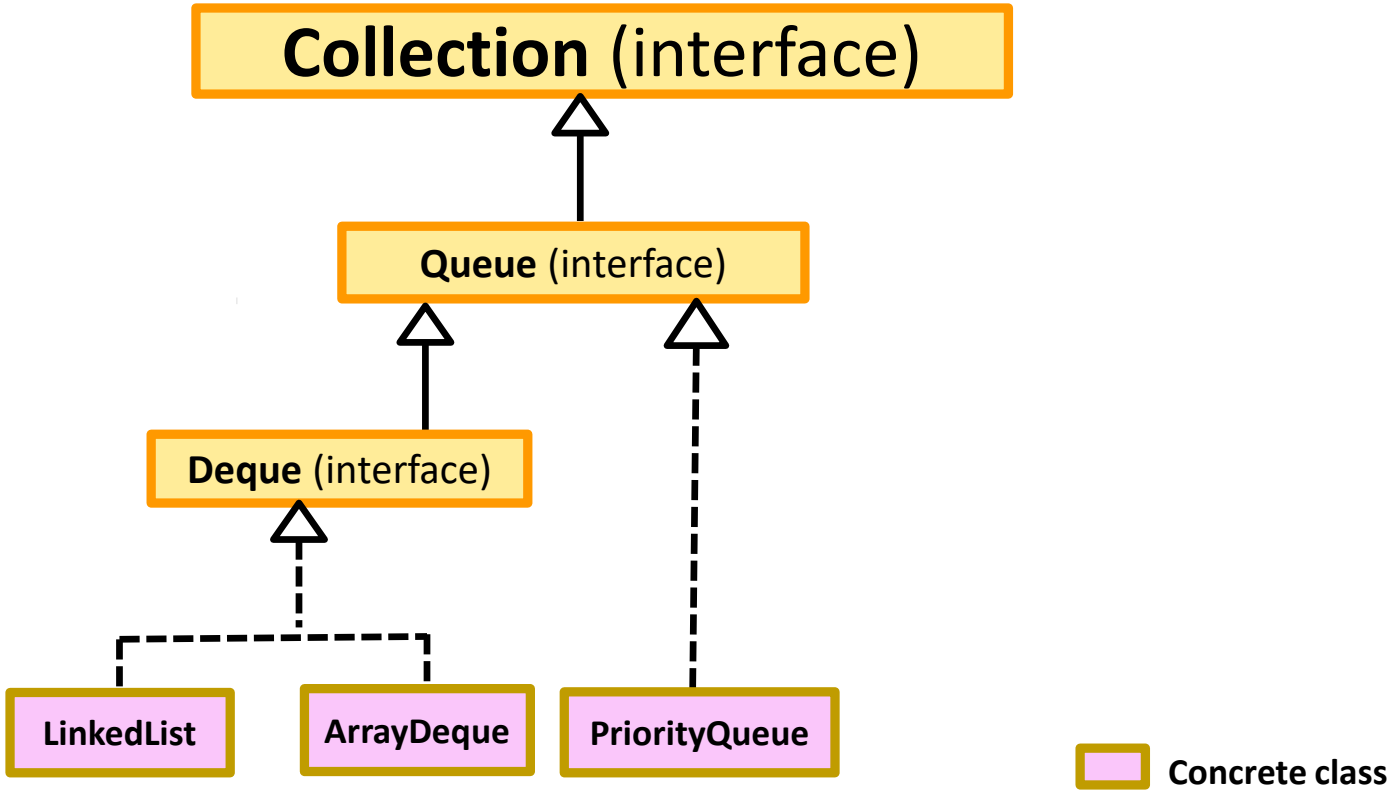
DeQueue – “Double-ended” Queue

- Data can be inserted and removed from either end



Queue in Java

- Queue in Java is an interface
- Deque is a child interface of Queue



ArrayDeque class

- implements ***Deque*** interface
- resizable-array that allows elements to be added/removed from both ends
- aka ***Array Deck***
- has ***iterator***
- no ***listIterator***
- has supporting methods to be used as a ***stack***

Methods in ArrayDeque Class

Many methods are common between ArrayDeque and LinkedList especially after Sep 2023.

ArrayList, Vector, LinkedList, Stack, ArrayDeque

```
add(E): boolean
~~add(int index, E): void~~
addFirst(E): void
addLast(E): void
```

```
remove(E): boolean
~~remove(int index): void~~
removeFirst(E): void
removeLast(E): void
```

```
~~get(int index): E~~
getFirst(): E
getLast(): E
```

```
size(): int
isEmpty(): boolean
contains(E): Boolean
clone(): object
```

LinkedList, Stack, ArrayDeque

```
peek(): E
pop(): E
push(E): void
poll(): E
```

LinkedList, ArrayDeque

```
peekFirst(): E
peekLast(): E
pollFirst(): E
pollLast(): E
```

Using ArrayDeque Class

Example: Create a deque and add an item. Use ***iterator*** to retrieve items.

Method used: ***add, addFirst, addLast, size()***

```
    ArrayDeque<Student> a = new ArrayDeque<Student>();
    a.add(new Student("KK"));
    a.addFirst(new Student("bobo"));
    a.addLast(new Student("tutu"));
    System.out.println(a.size());
    Iterator<Student> it = a.iterator();
    while (it.hasNext())
       System.out.println(it.next().name);
```

```
class Student {
    String name;
    Student (String s) { name=s; }
    @Override
    public String toString() { return(name); }
}
```

Output:

3
Bobo
KK
tutu

PriorityQueue Class

- implements ***Queue*** interface
- elements in the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time
- Object elements stored must be comparable
- used when elements of the queue are needed to be processed according to the priority
- Using Iterator or forEach will to retrieve elements will not guarantee the correct order
- Use poll() instead to retrieve the elements (but this will also remove the element)
- Make a copy by ➔ addAll()

Using PriorityQueue Class

Example: Create a PriorityQueue and add an item.

Method used: ***add*** (no ***addFirst***, no ***addLast***)

Print out by: ***iterator*** (no guarantee of correct order)

```
PriorityQueue a = new PriorityQueue();
a.add(82);
a.add(11);
a.add(35);

Iterator it = a.iterator();
while (it.hasNext())
   System.out.println(it.next());
```

Output:

11
82
35

Using PriorityQueue Class

Example: Create a PriorityQueue and add an item.

Print out by: ***forEach*** (no guarantee of correct order)

```
PriorityQueue a = new PriorityQueue();
a.add(82);
a.add(11);
a.add(35);

a.forEach(e -> System.out.println(e);
```

Output:

11
82
35

Using PriorityQueue Class

Example: Create a PriorityQueue and add an item.

Print out by: ***poll (read and remove)*** (correct order)

```
PriorityQueue a = new PriorityQueue();
a.add(82);
a.add(11);
a.add(35);

While (!a.isEmpty())

  System.out.println(a.poll());
```

Output:

11
35
82

Using PriorityQueue Class

Example: Create a PriorityQueue with comparator (order ***Student*** objects by name, ascending)

Print out by: ***poll (read and remove)*** (correct order)

```
PriorityQueue<Student> a = new PriorityQueue<Student>(new SortByName());
a.add(new Student("KK“, 3.8));   // name, GPA
a.add(new Student("Bobo“, 3.5));
a.add(new Student("Tutu“, 2.9));

PriorityQueue<Student> b = new PriorityQueue<Student>(new SortByName());
b.addAll(a);
while (!b.isEmpty())
   System.out.println(b.poll().name);
```

Output:

Bobo
KK
Tutu

```
class SortByName implements Comparator<Student> {
   @Override
   public int compare(Student s1, Student s2) {
     return ((s1.name.compareTo(s2.name)));
   }
}
```

Using PriorityQueue Class

Example: Create a PriorityQueue with comparator (order ***Student*** objects by GPA, descending)

Print out by: ***poll (read and remove)*** (correct order)

```
PriorityQueue<Student> a = new PriorityQueue<Student>(new SortByGPA());
a.add(new Student("KK“, 3.8));   // name, GPA
a.add(new Student("Bobo“, 3.5));
a.add(new Student("Tutu“, 2.9));

PriorityQueue<Student> b = new PriorityQueue<Student>(new SortByGPA());
b.addAll(a);
while (!b.isEmpty())
   System.out.println(b.poll().name);
```

Output:

KK
Bobo
Tutu

```
class SortByGPA implements Comparator<Student> {
   @Override
   public int compare(Student s1, Student s2) {
      return (Double.compare(s2.GPA, s1.GPA));
   }
}
```

Using PriorityQueue Class

Example: Create a PriorityQueue and add 3 integers.
Get and remove the head of the queue.

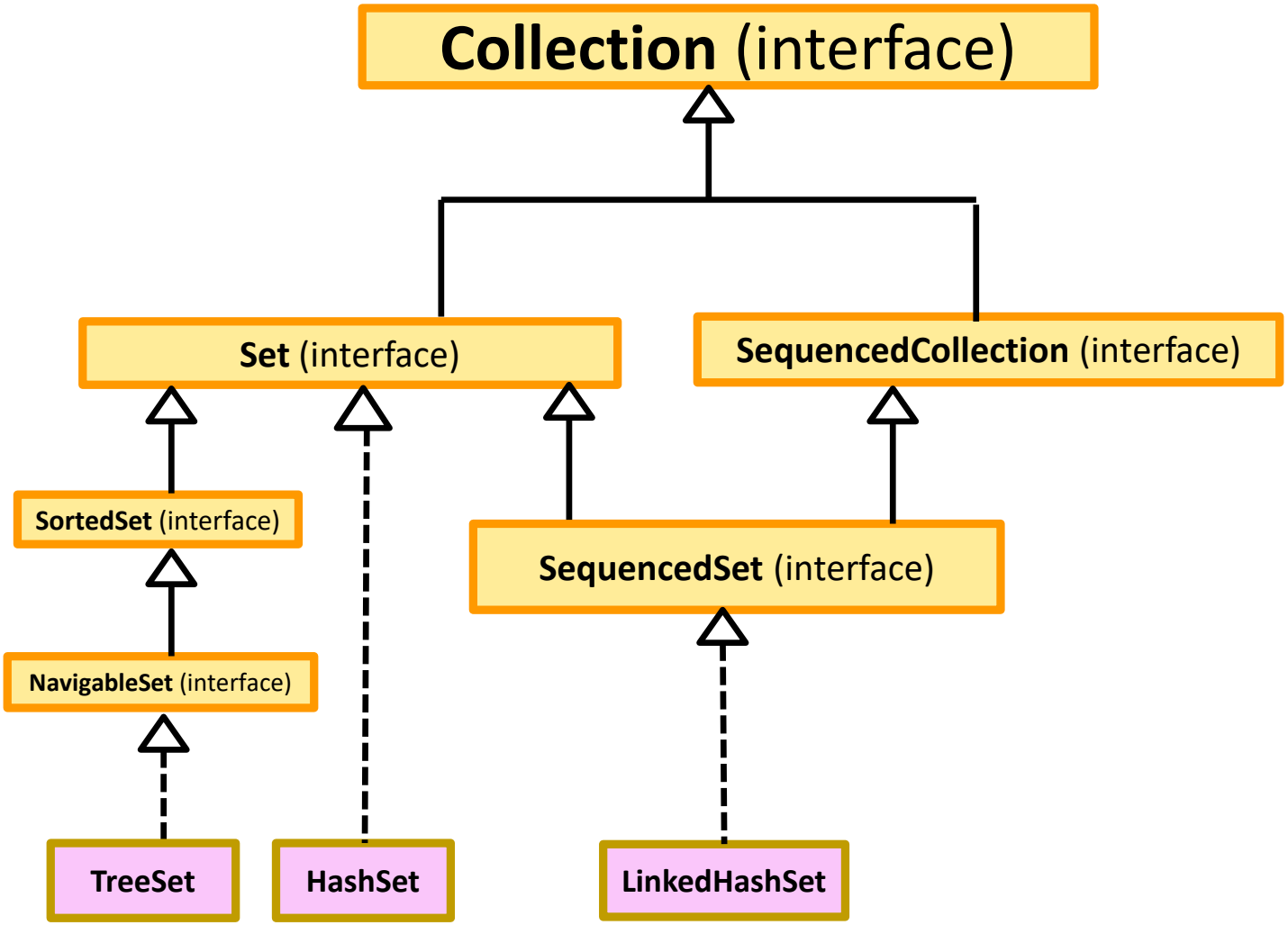
Method used: ***poll***

```
PriorityQueue a = new PriorityQueue();
a.add(82);
a.add(11);
a.add(35);
System.out.println(a.poll());
```

Output:

11

Simplified - Set in Java



HashSet in Java

- Most commonly used
- Implements in the Set interface
- Underlying data structure is key-value ***hash table***
- Does not allowed duplicate
- Does not guarantee as to what order the elements would be (List always maintain the order in which element is added)
- Hence, meaningless to access/remove element by index

HashSet

```
Set<String> schools = new HashSet<String>();

schools.add(“EEE”);
schools.add(“MAE”);
schools.add(“SMA”);
schools.add(“CLS”);
schools.add(“EEE”);
```

```
System.out.println (schools.size());
System.out.println (schools);
System.out.println (schools.isEmpty());
System.out.println (schools.contains(“MAD”));
```

Try – ***remove(), addAll()***

```
schools.remove(0);       // will this work?
```

HashSet – navigation methods

```
Set<String> schools = new HashSet<String>();

schools.add(“EEE”);
schools.add(“MAE”);
schools.add(“SMA”);
schools.add(“CLS”);
```

Using for-loop

```
for (String school : schools) { System.out.println(school); }
```

Using iterator

```
Iterator<String> schoolIterator = schools.iterator();
while (schoolIterator.hasNext()) {
  System.out.println(schoolIterator.next());
}
```

LinkedHashSet in Java

- Similar to HashSet class except that it ***maintains the order*** in which elements are being added
- It uses a hash table and a doubly-linked list to store & maintain the elements.

LinkedHashSet

```
LinkedHashSet<Integer> a = new LinkedHashSet<>();

a.add(123);
a.add(45);
a.add(9);
a.add(3210);
a.add(9);
Iterator it = a.iterator();
while (it.hasNext()){
    System.out.println(it.next());
}
```

Output:

123
45
9
3210

TreeSet in Java

- Similar to HashSet class and LinkedHashSet except that it automatically ***sorts*** the elements
- Implements self-balancing binary search tree
- The elements must implement the *Comparable Interface* in order to be sorted
- To implement the interface, the object class must implement the *compareTo()* method, which returns 0, negative or positive integer value (for equal, smaller or greater than the object respectively)
- The wrapper classes Integer, Double, Byte, Short, Long, Float, Character and String class all implement the Comparable interface

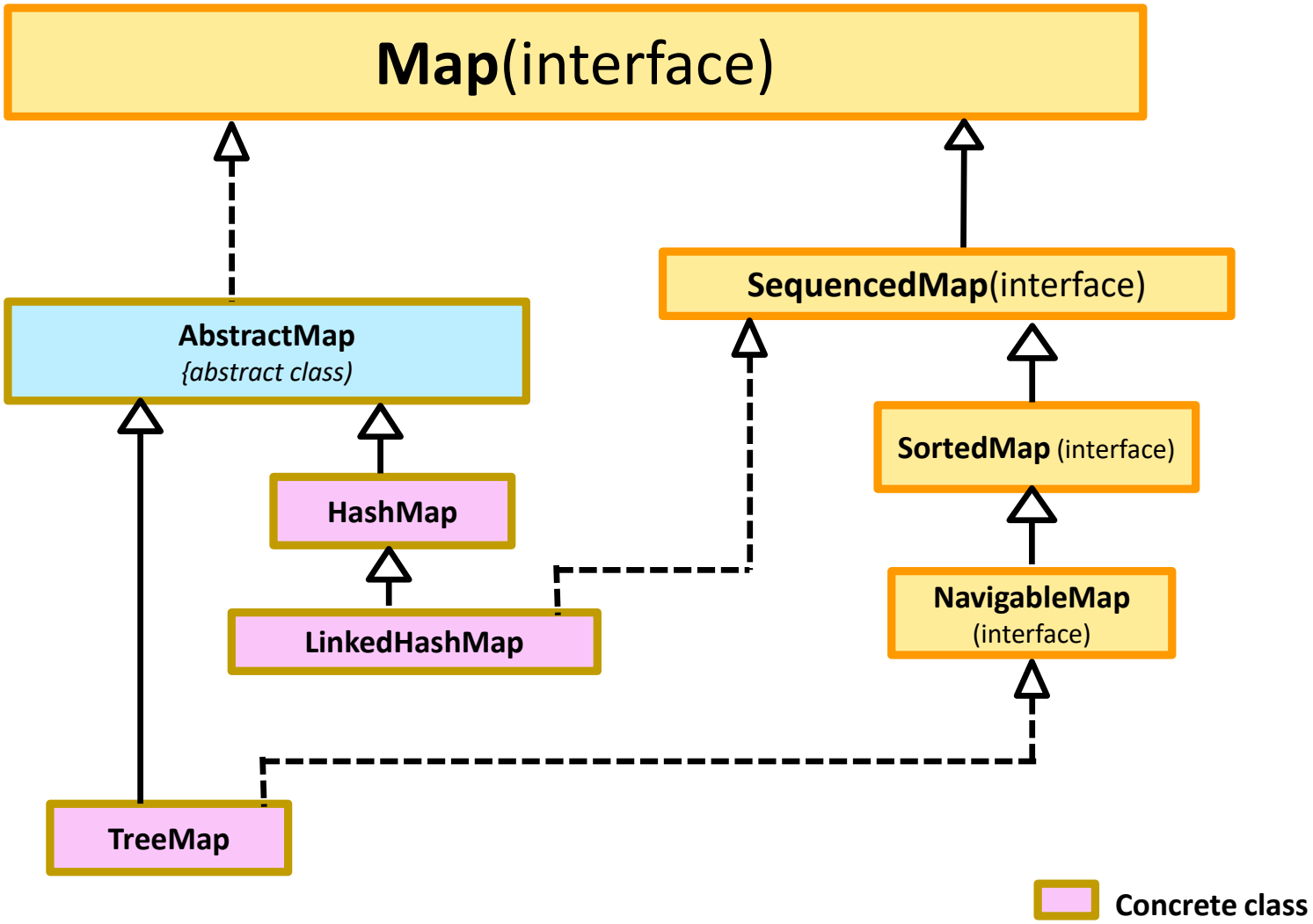
TreeSet

```
  TreeSet<Integer> a = new TreeSet<>();

  a.add(123);
  a.add(45);
  a.add(9);
  a.add(3210);
  a.add(9);

  TreeSet<Integer> b = new TreeSet<>(a);

  b.forEach((x)->System.out.println(x));
}
```

Output:

9
45
123
3210

Map Interface



Concrete Classes in Map

- A Map is a container that stores a collection of key-value pairs.
- Similar to set, the concrete classes of map are:
  - HashMap – store values based on keys
    – no duplicate
    – allows 1 or more null
    - does not keep the order in which values are added
  - LinkedHashMap – linked list implementation of HashMap
    – keeps the order in which values are added
  - TreeMap - unique values
    – sorted according to the key (in which the class must implement Comparable interface)
- Unlike Collection, the interfaces in Map family do not provide iterator for traversal through the container

HashMap

```
HashMap<String,String> a = new HashMap<>();
 a.put("EEE","Electrical Electronics Engineering");
 a.put("DE", "Digital Electronics");
 a.put("new module", null);

a.forEach((abbr,fullname)->System.out.println(abbr +"==>" +fullname));

System.out.println(a.get("DE"));
```

Output:

```
DE==>Digital Electronics
new module==>null
EEE==>Electrical Electronics Engineering
Digital Electronics
```

- ***forEach*** – default method inherited from Map interface

TreeMap

```
TreeMap<Integer,String> a = new TreeMap<>();
 a.put(7,"Electrical Electronics Engineering");
 a.put(2, "Engineering Maths");
 a.put(1, "Sports for Life");
 a.put(1, "Engineering Maths");

a.forEach((id,fullname)->System.out.println(id +"==>" +fullname));
```

Output:

1==>Engineering Maths
2==>Engineering Math
7==>Electrical Electronics Engineering

- ***No duplication of key allowed***