PROGRAMMING AND DATA STRUCTURES

USING DATA STRUCTURES
LIST, STACK, QUEUE, PRIORITY QUEUE

HOURIA OUDGHIRI

FALL 2023

OUTLINE

- The Java Collection Framework
- Java Collection Components: Containers, Iterators, and Algorithms
- Java Collection Containers (Data Structures): ArrayList, LinkedList, Stack, Queue, and PriorityQueue

STUDENT LEARNING OUTCOMES

At the end of this chapter, you should be able to:

- Describe the Java Collection Framework hierarchy
- Use the common methods in the interface Collection
- Use the iterators to traverse elements of a collection
- Use the static methods (algorithms) in the class **`Collections`**
- Use ArrayList, LinkedList, Stack, and PriorityQueue classes to store and manipulate data

DATA STRUCTURES

Data Structure: Collection of data organized in a specific way

- Arrays are the most commonly used data structure
- You can write any program without using any data structure other than arrays
- The program efficiency can be increased if you choose the appropriate data structures
- Choosing efficient data structures and algorithms - key issues in developing high-performance software

DATA STRUCTURES

- Data Structure is a generic class with
  - Data collection storage
  - Methods to manipulate the data (find, insert, remove, display, …)

- **`ArrayList`** is a data structure - an array and methods to access it (**`contains()`**, **`add()`**, **`remove()`**, **`get()`**, **`set()`**, **`toString()`**, **`...`**)

Java Collection Framework

Containers

Data Structures

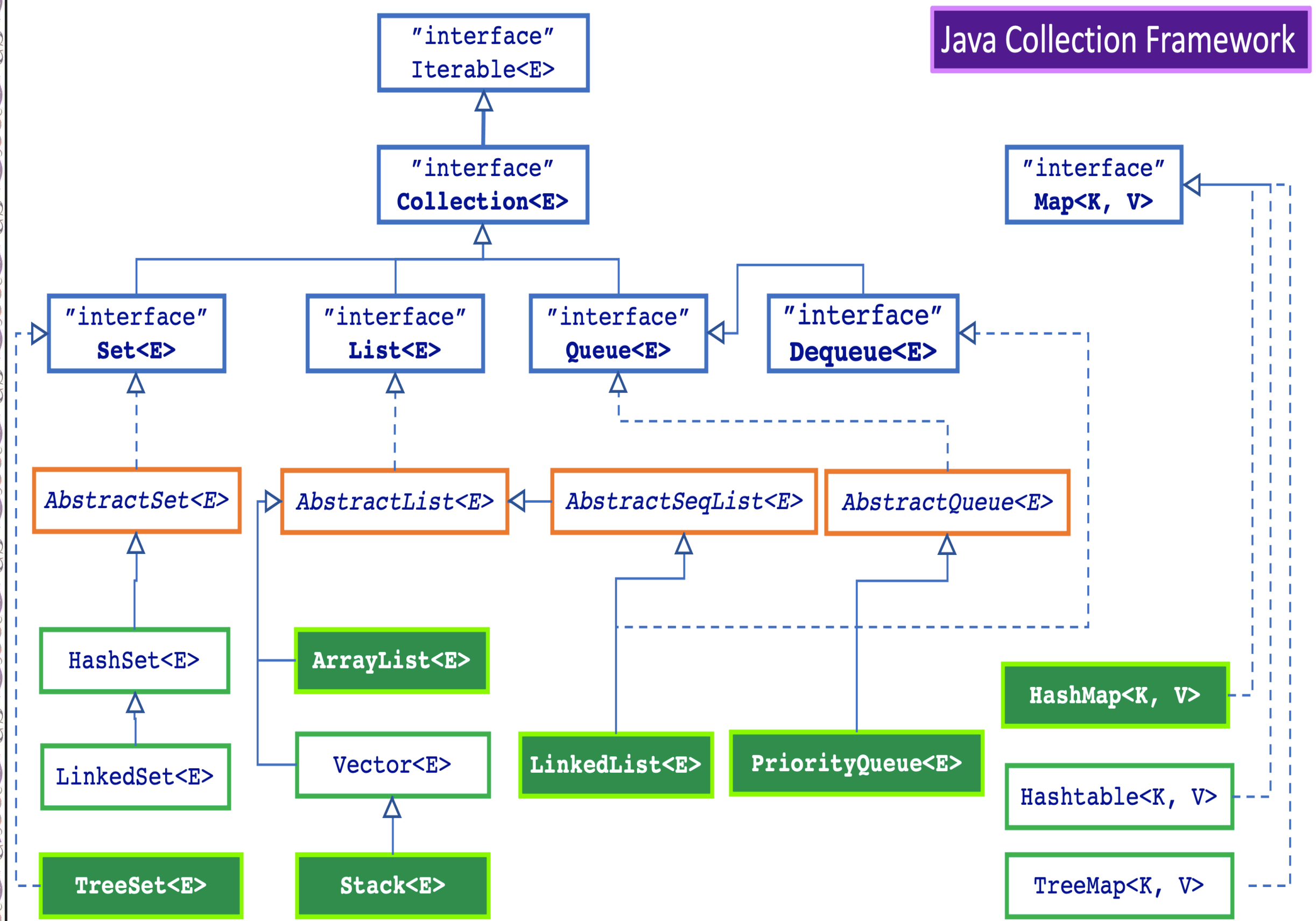
Iterators

Access to the containers

Algorithms

Utility methods to manipulate the containers

DATA STRUCTURES



Java Collection Framework

- **Containers (`java.util`)**
  - List (`ArrayList<E>`, `LinkedList<E>`)
  - Stack (`Stack<E>`)
  - Queue (`LinkedList<E>`)
  - Priority Queue (`PriorityQueue<E>`)
  - Binary Tree (`TreeSet<E>`)
  - Hash Table (`HashMap<K, V>`)
- Different ways to organize and manipulate data

"interface" Java.util.Collection<E>

```
+add(E): boolean
+addAll(Collection<? Extends E>):boolean    (Set Union)
+clear(): void
+contains(Object): boolean
+containsAll(Collection<?>): boolean
+equals(Object): boolean
+remove(Object): boolean
+removeAll(Collection<?>): boolean    (Set difference)
+retainAll(Collection<?>): boolean    (Set intersection)
+size(): int
+toArray(): Object[]
+toArray(T[]): T[]
+iterator():Iterator<E>
```

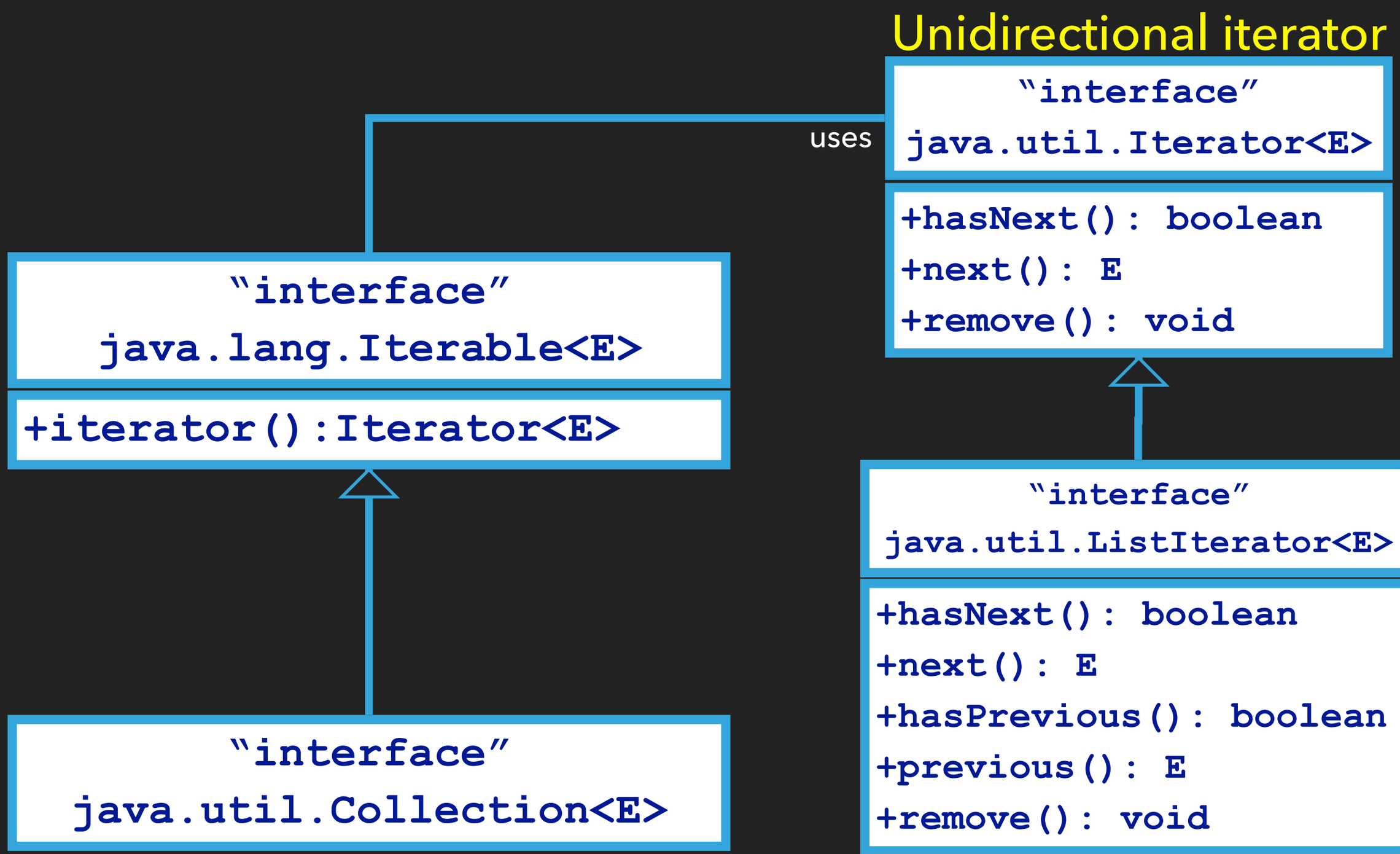
Java Collection Framework (Collection)

```
import java.util.Collection;
import java.util.ArrayList;
public class CollectionTest{
    public static void main(String[] args) {
        Collection<String> c1 = new ArrayList<String>();
        c1.add("New York");
        c1.add("Tokyo");
        c1.add("Paris");
        c1.add("Rome");
        c1.add("Brasilia");
        System.out.println("\nCities in collection 1: " + c1);
        System.out.println("\nIs Paris in the collection? " +
                           c1.contains("Paris"));
        c1.remove("Paris");
        System.out.println("\nThere are " + c1.size() +
                         " cities in collection 1");
        Collection<String> c2 = new ArrayList<String>();
        c2.add("Madrid");
        c2.add("Bangkok");
        c2.add("Moscow");
        c2.add("Beirut");
        c2.add("Rome");
        System.out.println("\nCities in collection 1: " + c1);
        System.out.println("\nCities in collection 2: " + c2);
        Collection<String> c3;
        c3 = (ArrayList<String>) ((ArrayList<String>)c1).clone();
        c3.addAll(c2);
        System.out.println("\n\nCities in collection 1 or collection 2: " + c3);

        c3 = (ArrayList<String>) ((ArrayList<String>)c1).clone();
        c3.retainAll(c2);
        System.out.println("\nCities in collection 1 and collection 2: " + c3);

        c3 = (ArrayList<String>) ((ArrayList<String>)c1).clone();
        c3.removeAll(c2);
        System.out.println("\nCities in collection 1, but not in collection 2:" +  c3);
    }
}
```

Java Collection Framework (Iterators)



Java Collection Framework (Iterators)

```
import java.util.Collection;
import java.util.ArrayList;
import java.util.Iterator;
public class IteratorsTest{
    public static void main(String[] args) {
        ArrayList<String> al = new ArrayList<>();
        al.add("New York");
        al.add("Tokyo");
        al.add("Paris");
        al.add("Rome");
        al.add("Brasilia");
        System.out.println(al);
        Iterator<String> iter = al.iterator();
        while(iter.hasNext()){
            System.out.println(iter.next().toUpperCase());
        }
    }
}
```

Java Collection Framework (**Algorithms**)

Java.util.Collections

+sort(List): void

+binarySearch(List, Object): int

+reverse(List): void

+shuffle(List): void

+copy(List, List): void

+fill(List, Object): List

+swap(List, int, int):void

Java Collection Framework (Algorithms)

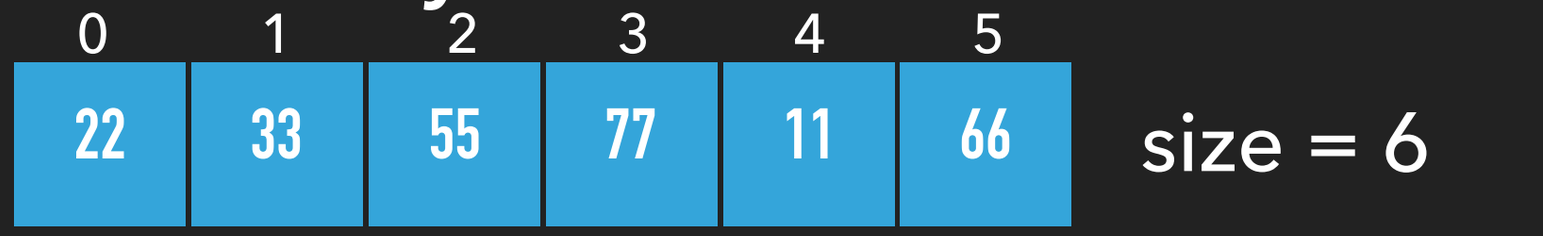
```
import java.util.Collection;
import java.util.Iterator;
import java.util.Collections;
import java.util.ArrayList;
public class AlgorithmsTest{
    public static void main(String[] args) {
        ArrayList<String> al = new ArrayList<>();
        al.add("New York");
        al.add("Tokyo");
        al.add("Paris");
        al.add("Rome");
        al.add("Brasilia");
        System.out.println("\nOriginal list: " + al);
        Collections.sort(al);
        System.out.println("\nSorted list: " + al);
        Collections.shuffle(al);
        System.out.println("\nShuffled list: "+ al);
    }
}
```

Java Collection Framework (**Containers**)

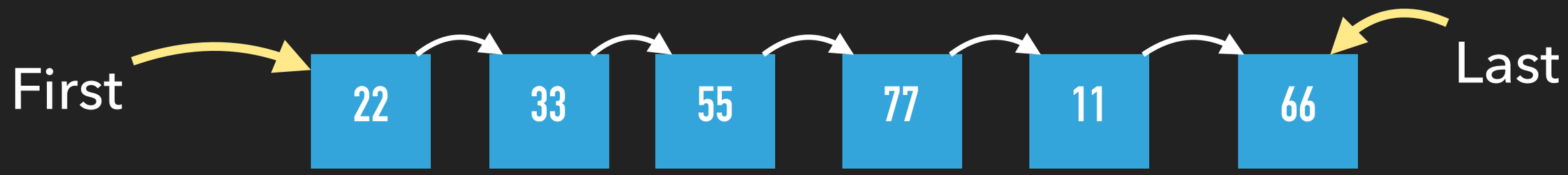
- **List**: store ordered collection of elements
- **Stack**: stores elements that are processed in **LIFO** fashion (Last-In First-Out)
- **Queue**: stores elements that are processed in **FIFO** fashion (First-In First-Out)
- **PriorityQueue**: stores elements that are processed in their natural ordering or using a specific priority

List

- Array based list
  - **`ArrayList`** - Random Access to the elements - index to any element

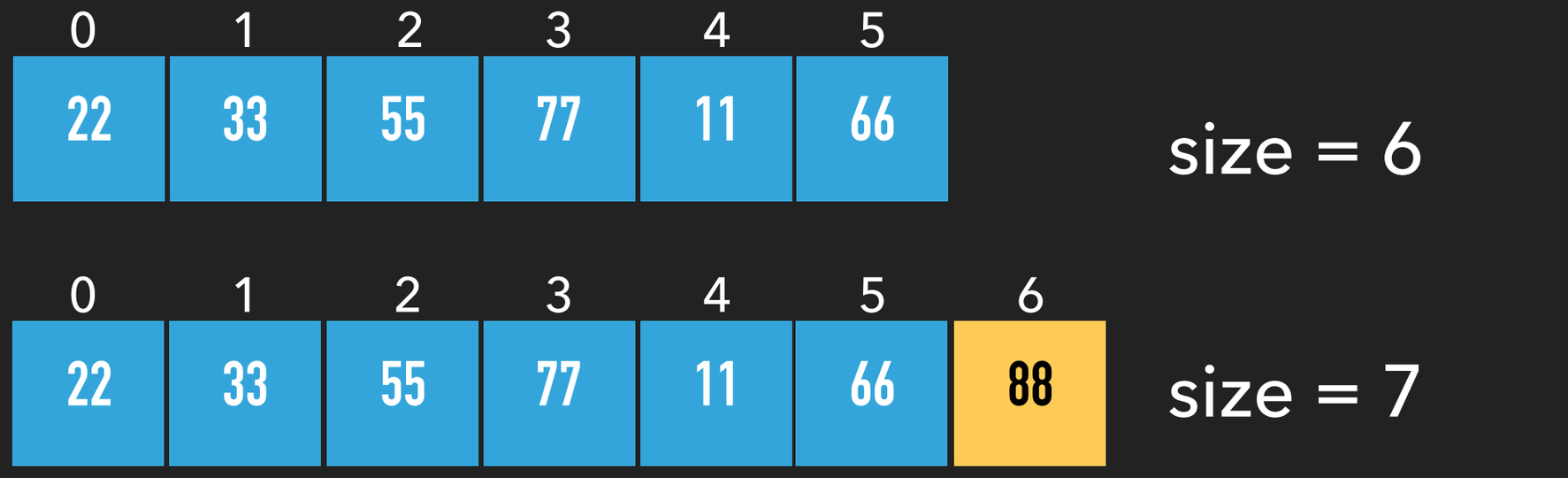


- Linked List
  - **`LinkedList`** - Sequential access only (first, last, next)

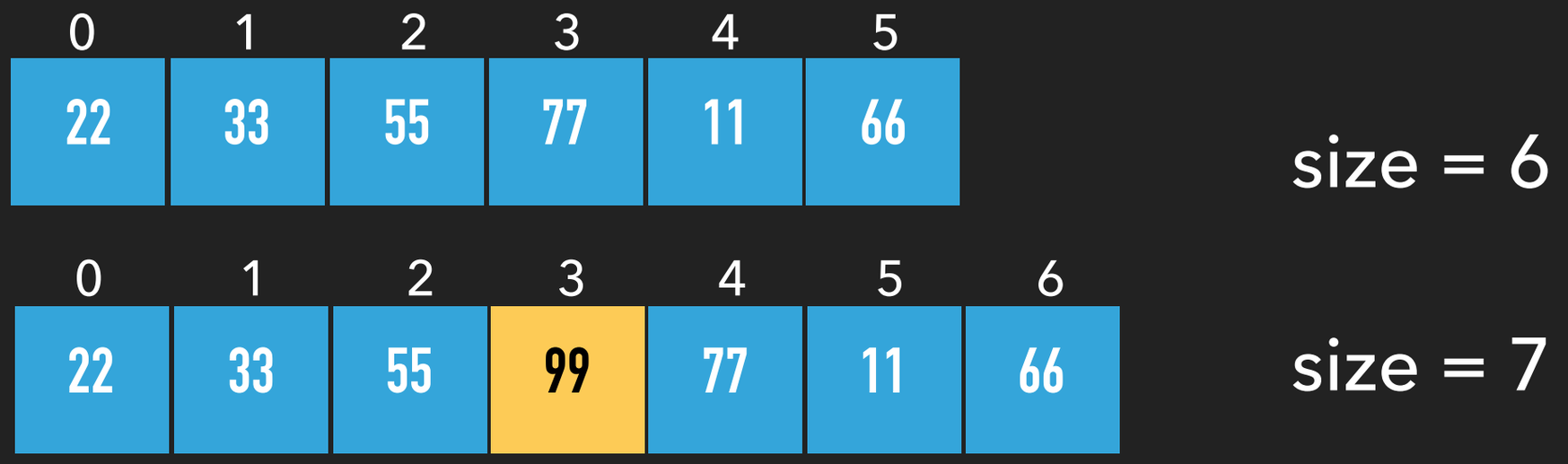


ArrayList

- **`add(88)`**

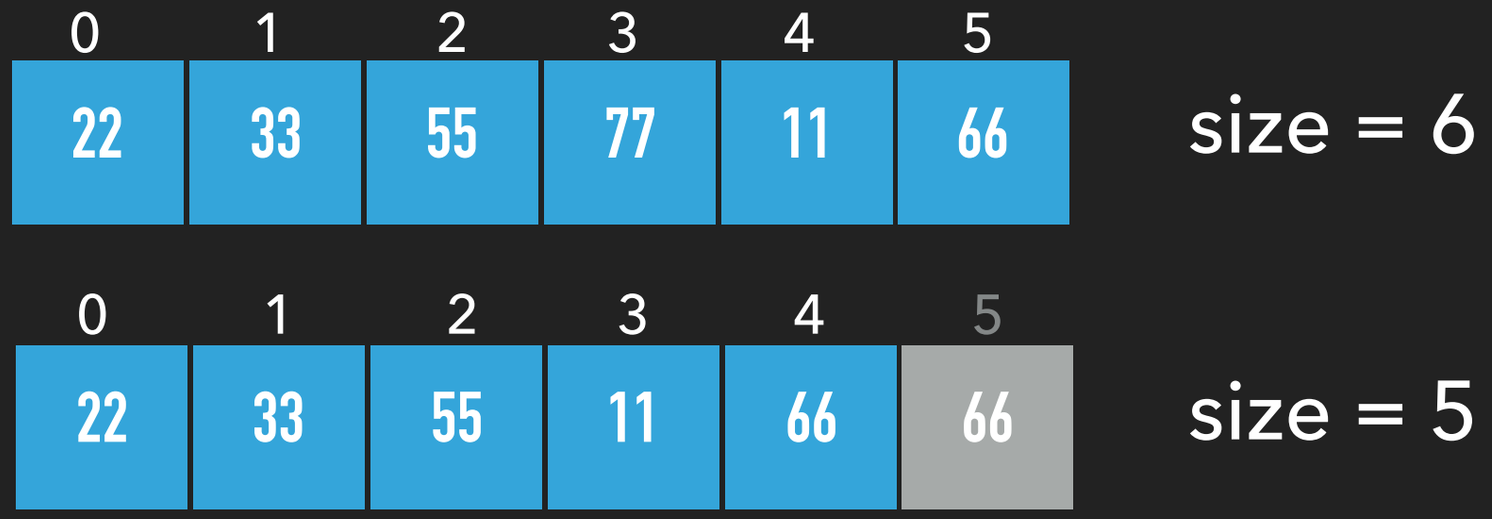


- **`add(3, 99)`**

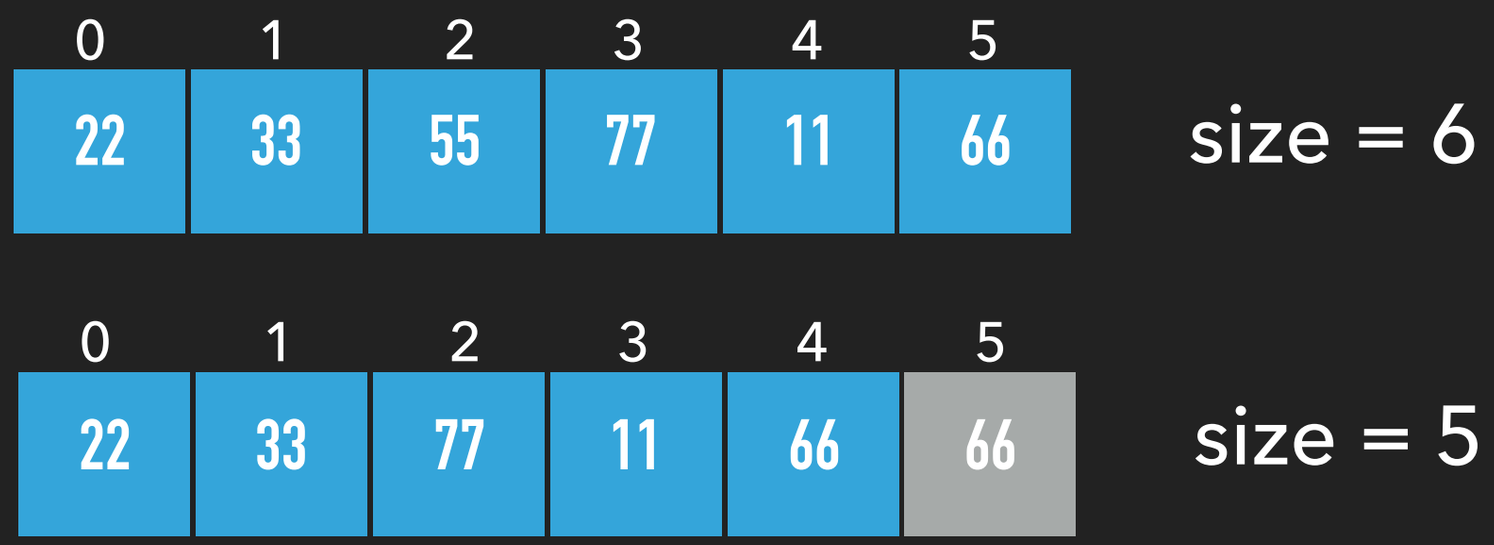


ArrayList

- **`remove(77)`**

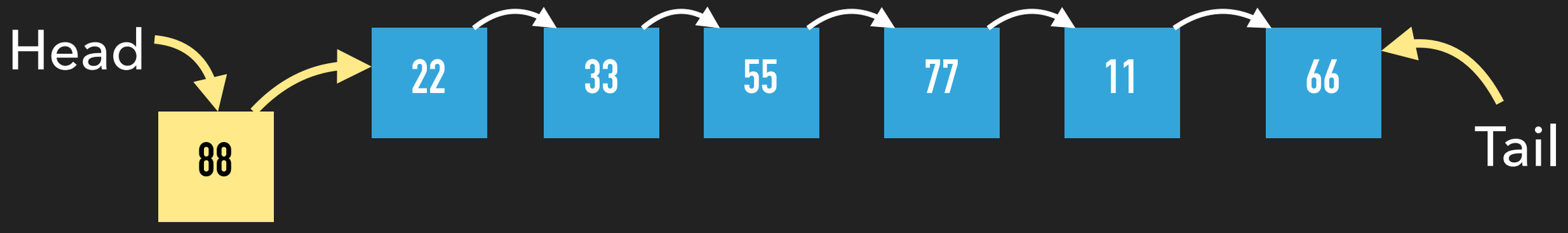


- ***`remove(2)`***

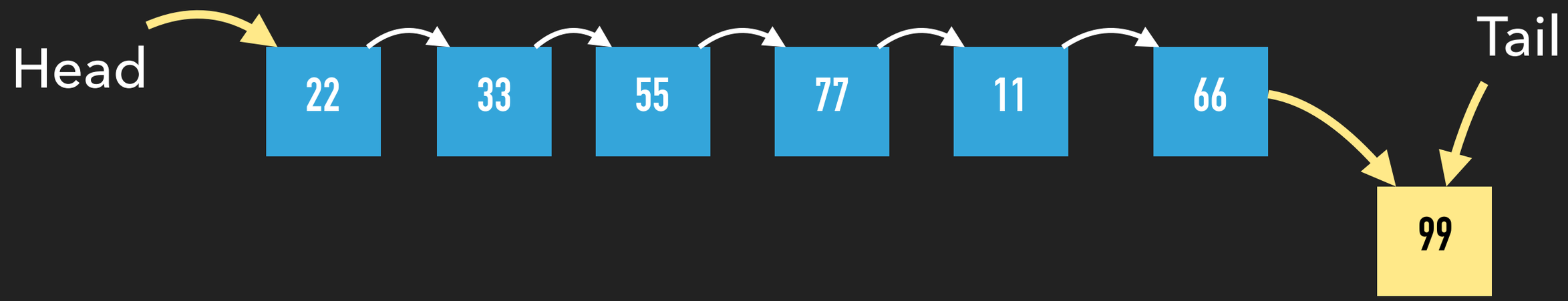


Linked List

- ***addFirst(88)***

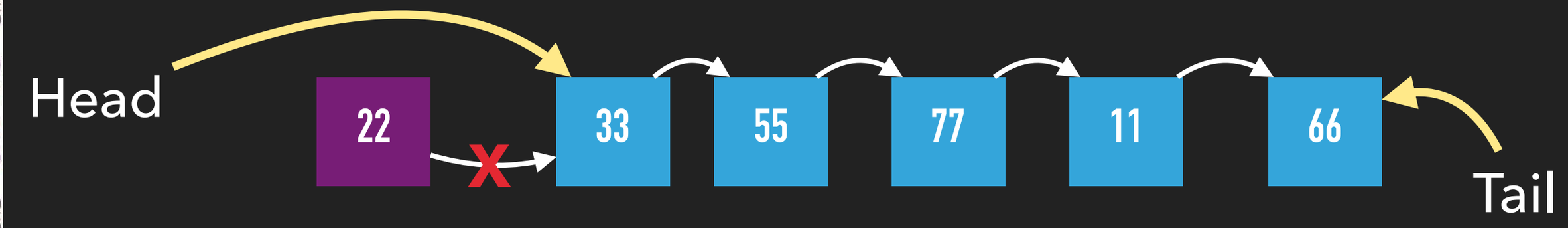


- ***addLast(99)***

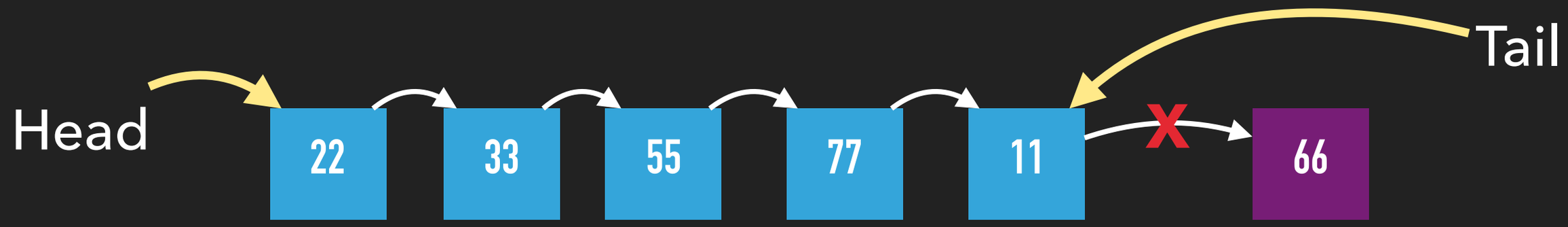


Linked List

- ***removeFirst()***



- ***removeLast()***



Linked List

Java.util.LinkedList<E>

```
+LinkedList()
+LinkedList(Collection<? Extends E>)
+addFirst(E): void
+addLast(E): void
+getFirst(): E
+getLast(): E
+removeFirst(): E
+removeLast(): E
+iterator(): Iterator<E>
+listIterator(): ListIterator<E>
+listIterator(int): ListIterator<E>
```

Linked List

```
import java.util.LinkedList;
import java.util.ListIterator;

public class LLTest{
    public static void main(String[] args) {
        LinkedList<String> ll = new LinkedList<>();
        ll.addFirst("New York");
        ll.addLast("Tokyo");
        ll.addFirst("Paris");
        ll.addLast("Rome");
        ll.addLast("Brasilia");

        System.out.print("\nLinked list forward: [ ");
        ListIterator<String> forward = ll.listIterator();
        while (forward.hasNext()) {
            System.out.print(forward.next() + " ");
        }
        System.out.println("]");
        System.out.print("\nLinked list backward: [ ");
        ListIterator<String> backward = ll.listIterator(ll.size());
        while (backward.hasPrevious()) {
            System.out.print(backward.previous() + " ");
        }
        System.out.println("]\n");
    }
}
```

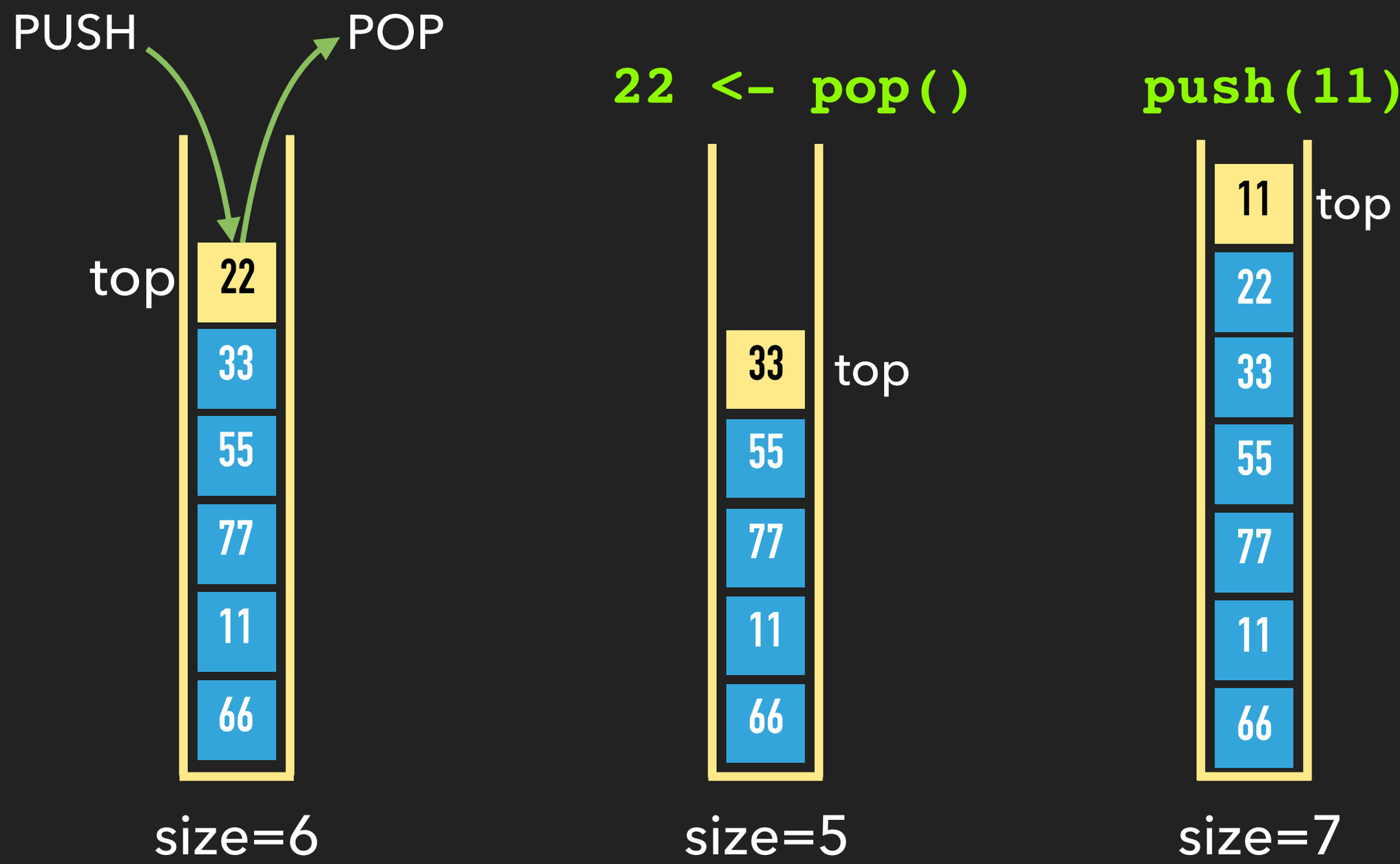
List

- ArrayList
  - Random access to any element
  - Uses an array (contiguous memory space)
  - Size of the array can be adjusted at runtime
- LinkedList
  - Sequential access to the list elements
  - Uses as much memory as the number of elements in the list (more efficient in memory usage)

Stack

- LIFO structure - (Last In First Out)
- Access to the top of the stack only
- Operations: `push()`, `pop()`, and `peek()`
- Used for tracking method calls and arithmetic expression evaluation

Stack


PUSH
POP
top
22
33
55
77
11
66
size=6
22 <- pop()
33
top
55
77
11
66
size=5
push(11)
11
top
22
33
55
77
11
66
size=7

Stack

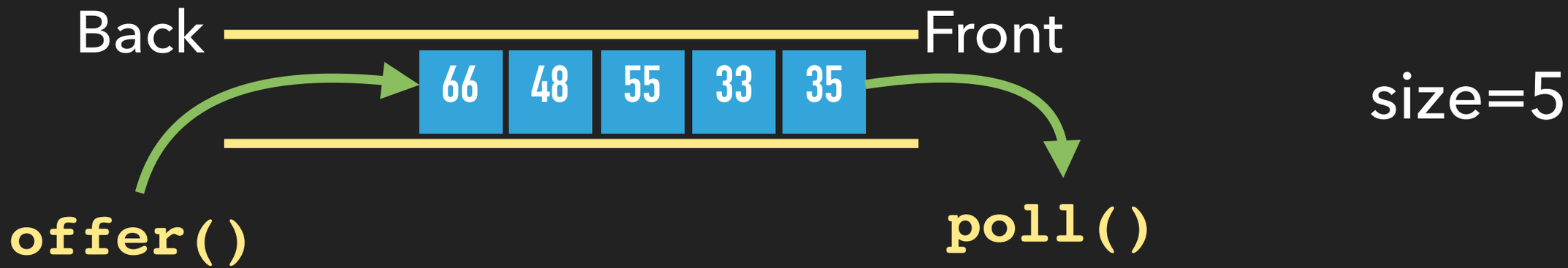
Java.util.Stack<E>

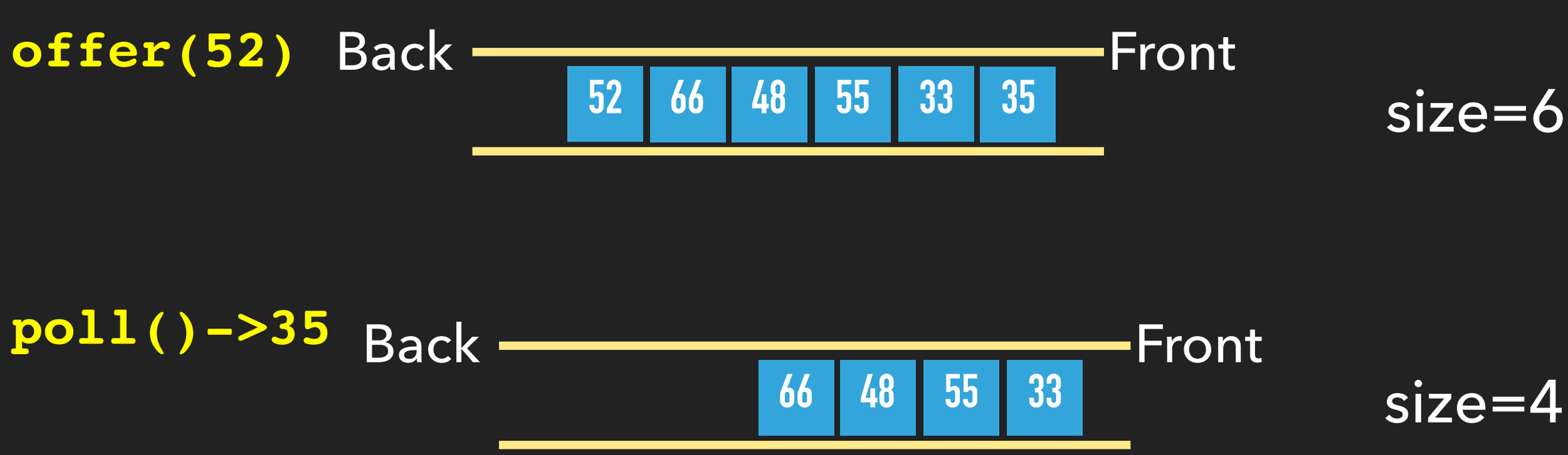
```
+Stack(): void
+isEmpty(): boolean
+peek(): E
+pop(): E
+push(E): void
```

Queue

- **FIFO** structure - (First In First Out)
- Access at the front (or back) only
- Operations: **`offer()`**, **`poll()`**, and **`peek()`**
- Used for task scheduling and many real-life problem modeling
- Implemented as a linked list in the Java API

Queue


Back
Front
66
48
55
33
35
offer()
poll()
size=5


offer(52)
Back
Front
52
66
48
55
33
35
size=6
poll()->35
Back
Front
66
48
55
33
size=4

Queue

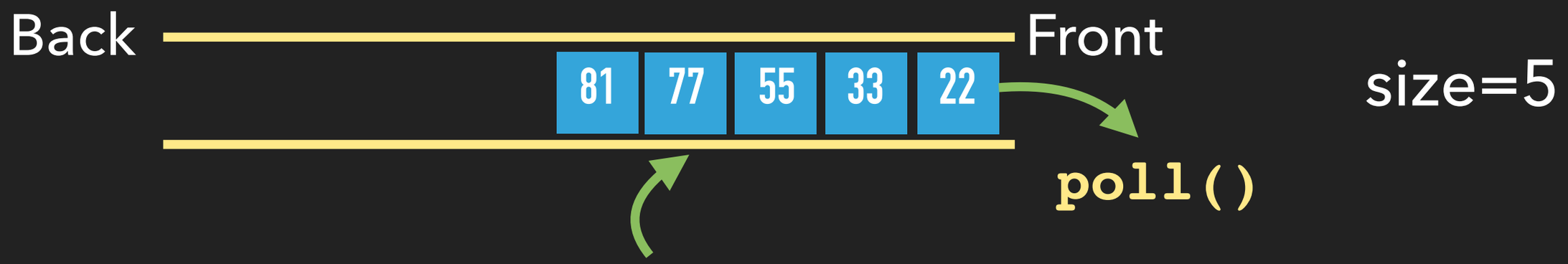
Java.util.LinkedList<E>

```
+LinkedList()
+LinkedList(Collection<? Extends E>)
+addFirst(E): void
...
+getLast(): E
+removeFirst(): E
+poll(): E
+offer(E): void
+peek(): E
```

Priority Queue

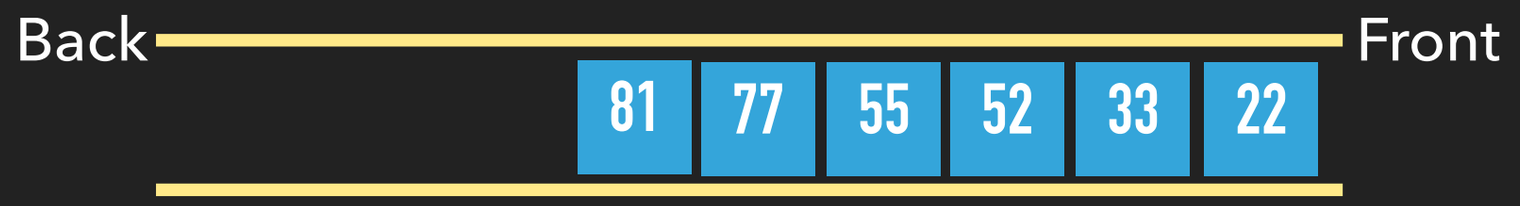
- FIFO structure with priority
- Access at the front or back only
- Elements are inserted according to a priority (natural ordering)
- Operations: **`offer()`**, **`poll()`**, and **`peek()`**
- Used for task scheduling and many real-life problem modeling too

Priority Queue

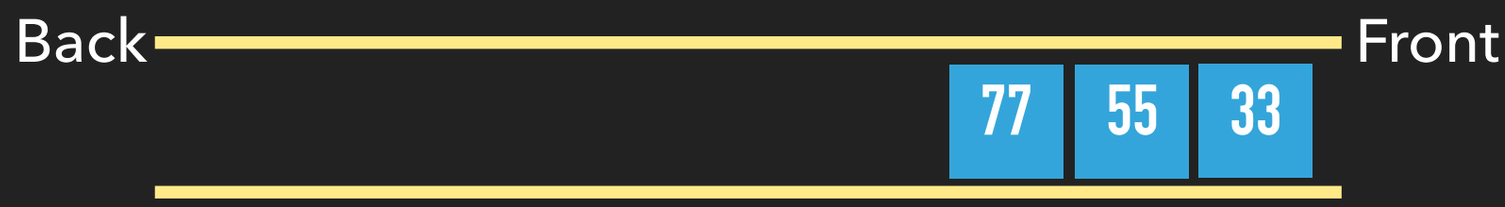


`offer()` - dependent on the priority (natural ordering)

`offer(52)`



`poll()->22`



Priority Queue

- Priority Queue uses the natural ordering (`compareTo()` from **`Comparable`**) or a comparator (`compare()`)

```
java.util.PriorityQueue<E>
```

```
+PriorityQueue()
+PriorityQueue(Comparator<? super E> c)
+offer(E): boolean
+poll(): E
+remove(): E
+peek(): E
```

Using Java API data structures (Version 1)

```
import java.util.Collection;
import java.util.Iterator;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Stack;
import java.util.PriorityQueue;
public class DSTest1{
    public static void main(String[] args) {
        Collection<String> [] ds = new Collection[5];
        ds[0] = new ArrayList<>();
        ds[1] = new LinkedList<>();
        ds[2] = new Stack<>();
        ds[3] = new LinkedList<>();
        ds[4] = new PriorityQueue<>();

        String[] fruits = {"Orange","Kiwi","Pomegranate","Melon", "Apple","Banana","Strawberry"};
        String[] names = {"ArrayList","LinkedList","Stack","Queue","PriorityQueue"};

        // Using add() from Interface Collection
        for(int i=0; i<fruits.length; i++) {
            for(int j=0; j<5; j++)
                ds[j].add(fruits[i]);
        }
        // Using iterators
        System.out.println("Using Iterators");
        for(int i=0; i<5; i++){
            Iterator<String> iterator = ds[i].iterator();
            System.out.print(names[i] + ": ");
            print(iterator);
        }
        System.out.println();
    }
    public static <E> void print(Iterator<E> iterator){
        System.out.print("[ ");
        while(iterator.hasNext()){
            System.out.print(iterator.next() + " ");
        }
        System.out.println("]");
    }
}
```

Using Java API data structures (Version 2)

```
import java.util.Collection;
import java.util.Iterator;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Stack;
import java.util.PriorityQueue;
public class DSTest2{
    public static void main(String[] args) {
        Collection<String> [] ds = new Collection[5];
        ds[0] = new ArrayList<>();
        ds[1] = new LinkedList<>();
        ds[2] = new Stack<>();
        ds[3] = new LinkedList<>();
        ds[4] = new PriorityQueue<>();

        String[] fruits = {"Orange","Kiwi","Pomegranate","Melon",
                           "Apple","Banana","Strawberry"};
        String[] names = {"ArrayList","LinkedList","Stack","Queue","PriorityQueue"};

        // Using add() from Interface Collection
        for(int i=0; i<fruits.length; i++) {
            for(int j=0; j<5; j++)
                ds[j].add(fruits[i]);
        }
        // Using toString()
        System.out.println("Using toString()");
        for(int i=0; i<5; i++){
            System.out.println(names[i] + ": " + ds[i]);
        }
    }
}
```

Using Java data structures (Version 3)

```
import java.util.Collection;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Stack;
import java.util.PriorityQueue;
public class DSTest3{
    public static void main(String[] args) {
        Collection<String> [] ds = new Collection[5];
        ds[0] = new ArrayList<>();
        ds[1] = new LinkedList<>();
        ds[2] = new Stack<>();
        ds[3] = new LinkedList<>();
        ds[4] = new PriorityQueue<>();
        String[] fruits = {"Orange","Kiwi","Pomegranate","Melon","Apple","Banana","Strawberry" };
        String[] names = {"ArrayList","LinkedList","Stack","Queue","PriorityQueue"};
        // Using data structure specific methods
        for(int i=0; i<fruits.length; i++) {
            ds[0].add(fruits[i]);
            ((LinkedList)ds[1]).addFirst(fruits[i]);
            ((Stack)ds[2]).push(fruits[i]);
            ((LinkedList)ds[3]).offer(fruits[i]);
            ((PriorityQueue)ds[4]).offer(fruits[i]);
        }
        System.out.println("\nUsing DS specific interface");
        System.out.print("\nArray List: [");
        for(int i=0; i<fruits.length; i++) {
            System.out.print(((ArrayList)ds[0]).get(i) + " ");
        }
        System.out.println("]");
        System.out.print("\nLinked List: [");
        while(((LinkedList)ds[1]).size() != 0) {
            System.out.print(((LinkedList)ds[1]).getLast() + " ");
            ((LinkedList)ds[1]).removeLast();
        }
        System.out.println("]");
        System.out.print("\nStack: [");
        while(!ds[2].isEmpty())
            System.out.print(((Stack)ds[2]).pop() + " ");
        System.out.println("]");

        System.out.print("\nQueue: [");
        while(!ds[3].isEmpty())
            System.out.print(((LinkedList)ds[3]).poll() + " ");
        System.out.println("]");

        System.out.print("\nPriority Queue: [");
        while(!ds[4].isEmpty())
            System.out.print(((PriorityQueue)ds[4]).poll()+ " ");
        System.out.println("]\n");
    }
}
```

Summary

- Java Collection Framework Hierarchy
- Data structures: **`ArrayList`**, **`LinkedList`**, **`Stack`**, **`Queue`**, **`PriorityQueue`**
- Iterators (**`Iterator<E>`** and **`ListIterator<E>`**)
- Algorithms (**`search`**, **`sort`**, **`shuffle`**, **`inverse`**, **`swap`**, ...)