PROGRAMMING AND DATA STRUCTURES

USING DATA STRUCTURES
LIST, STACK, QUEUE, PRIORITY QUEUE

HOURIA OUDGHIRI

FALL 2022

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
- Choosing efficient data structures and algorithms - key issues in developing high-performance software

DATA STRUCTURES

- 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

DATA STRUCTURES

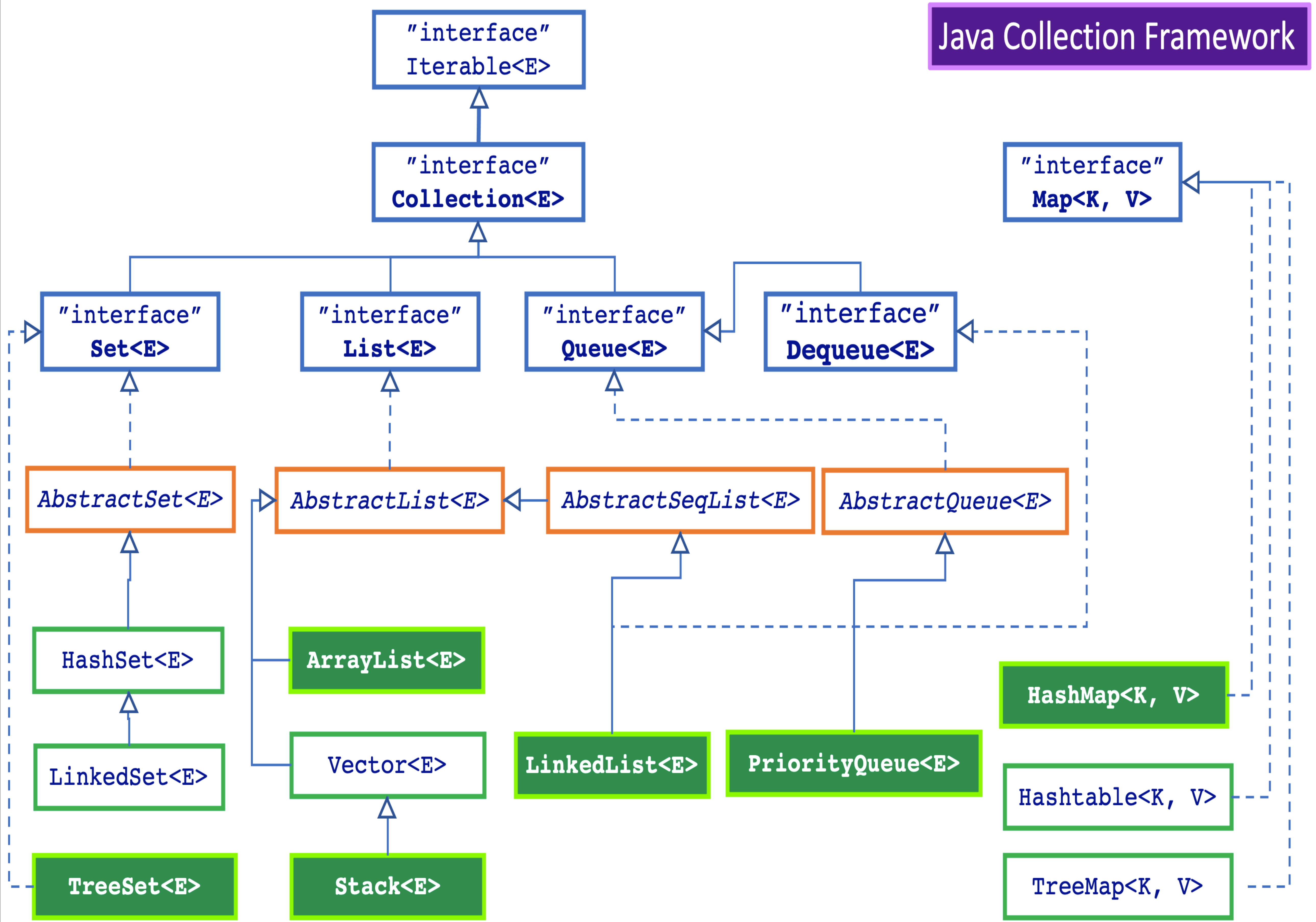
- Data Structure is a generic class with
  - Data collection storage
  - Methods to manipulate the data (find, insert, remove, display, …)
- **`ArrayList`** is a data structure - an array and methods to access it (**`contains()`**, **`add()`**, **`remove()`**, **`get()`**, **`set()`**, **`toString()`**, …)

Java Collection Framework

- **Containers -** Data structures
- **Iterators -** objects to iterate through the containers' data items
- **Algorithms -** Utility methods to manipulate containers (sort, search, shuffle, etc.)

DATA STRUCTURES

Java Collection Framework



Java Collection Framework

- **Containers (`java.util`)**
  - List (**`ArrayList<E>`**, **`LinkedList<E>`**)
  - Stack (**`Stack<E>`**)
  - Queue (**`LinkedList<E>`**)
  - Priority Queue (**`PriorityQueue<E>`**)
  - Binary Tree (**`HashSet<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>
```

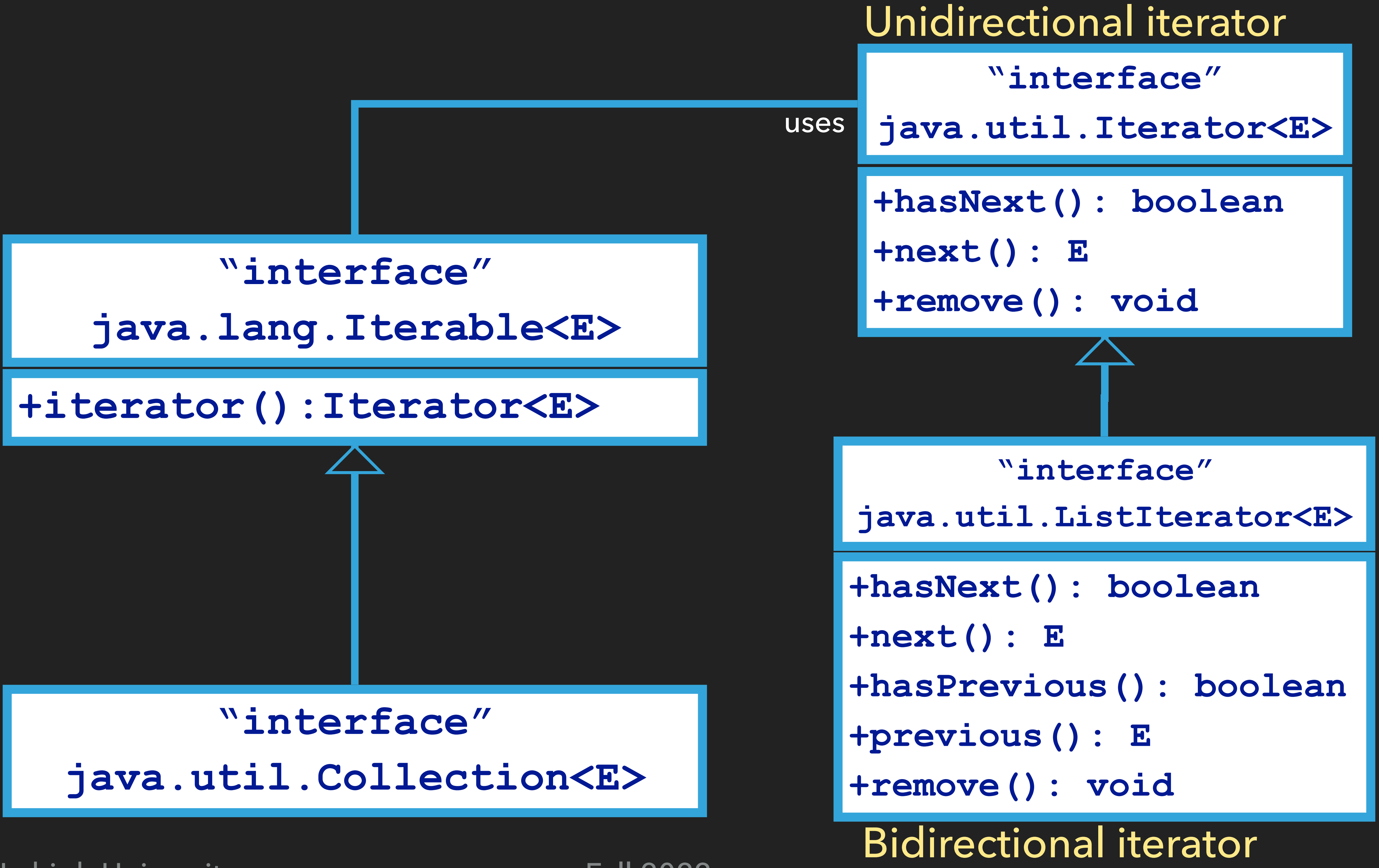
Java Collection Framework (Containers)

```
import java.util.ArrayList;
import java.util.Collection;

public class Test {
  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("Cities 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);
```

Java Collection Framework (Containers)

```
Collection<String> 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.ArrayList;
import java.util.Iterator;
public class Test {
 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");

  Iterator<String> iter = al.iterator();
  System.out.print("[ ");
  while(iter.hasNext()){
  System.out.print(iter.next().toUpperCase() +
                  " ");
  }
  System.out.print(“]");
```

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**)

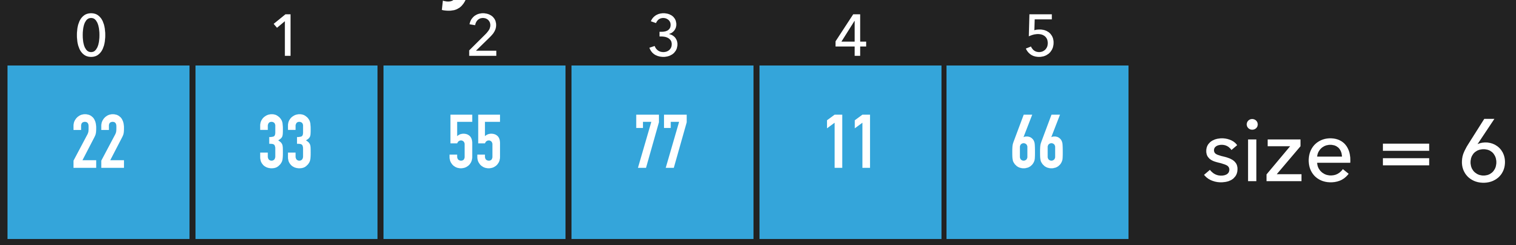
```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
public class Test {
  public static void main(String[] args) {
    ArrayList<String> al = new ArrayList<>();
    al.add("New York"); al.add("Tokyo");
    al.add("Paris"); al.add(“Rome");
    al.add("Brasilia");

    Collections.sort(al);
    System.out.println("\nSorted list: " + al);
    Collections.shuffle(al);
    System.out.println("\nShuffled list: "+ al);
  }
}
```

Java Collection Framework (**Containers**)

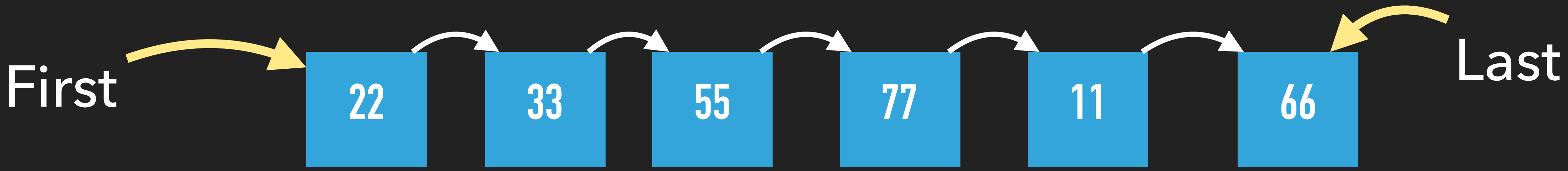
- ✦ **List**: store ordered collection of elements
- ✦ **Stack**: stores elements that are processed in LIFO fashion (Last-In First-Out)
- ✦ **Queue**: stores elements that are processed in FIFO fashion (First-In First-Out)
- ✦ **PriorityQueue**: stores elements that are processed in the order defined by a 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)***

0	1	2	3	4	5
22	33	55	77	11	66

size = 6

0	1	2	3	4	5	6
22	33	55	77	11	66	88

size = 7

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

0	1	2	3	4	5
22	33	55	77	11	66

size = 6

0	1	2	3	4	5	6
22	33	55	99	77	11	66

size = 7

ArrayList

- `remove(77)`

0	1	2	3	4	5
22	33	55	77	11	66

size = 6

0	1	2	3	4	5
22	33	55	11	66	66

size = 5

- *`remove(2)`*

0	1	2	3	4	5
22	33	55	77	11	66

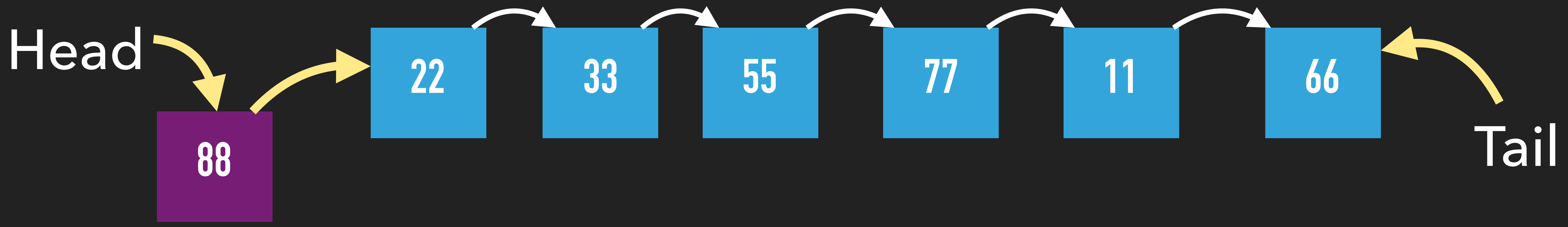
size = 6

0	1	2	3	4	5
22	33	77	11	66	66

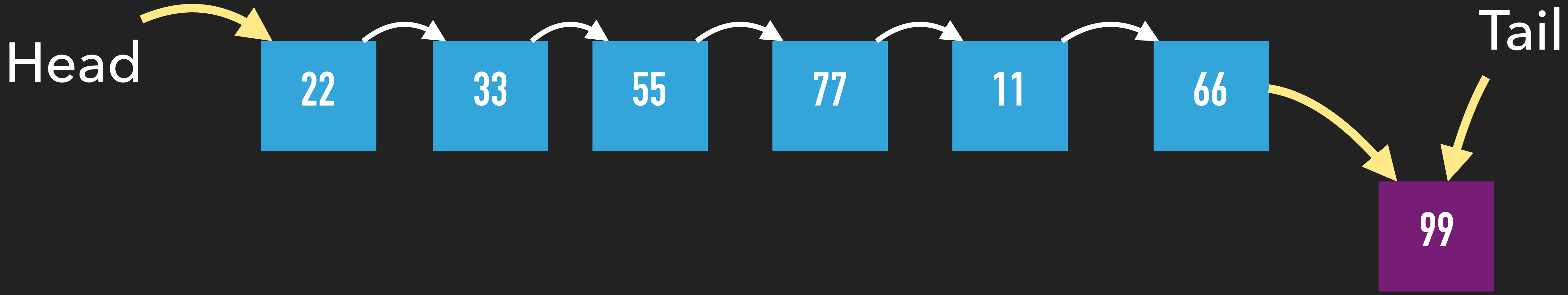
size = 5

Linked List

- *addFirst(88)*

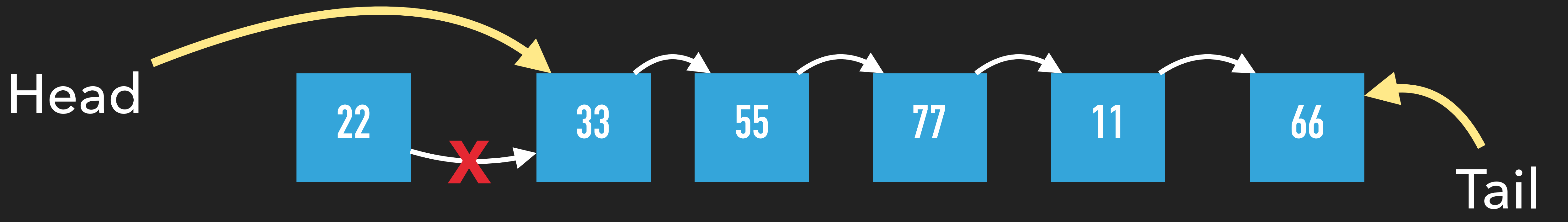


- *addLast(99)*

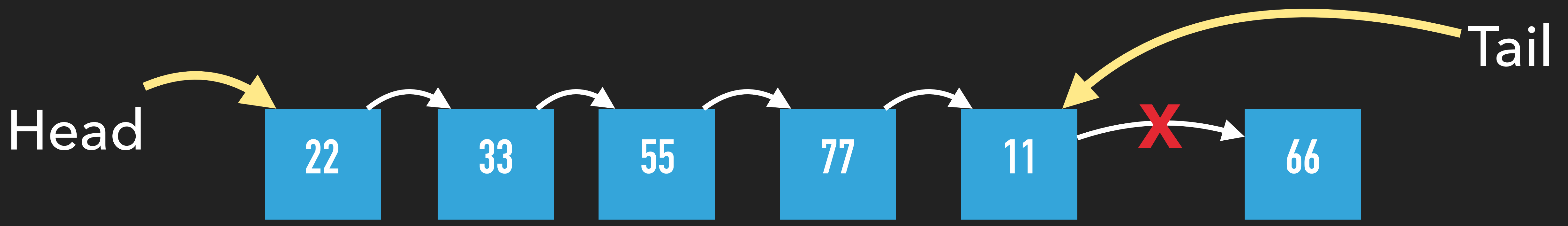


Linked List

- *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
+listIterator(): ListIterator<E>
+listIterator(int): ListIterator<E>
```

```
import java.util.LinkedList;
import java.util.ListIterator;
public class TestList {
  public static void main(String[] args) {
    LinkedList<String> linkedList = new LinkedList<>();
    linkedList.addFirst("red");
    linkedList.addFirst("green");
    linkedList.addLast("blue");

    System.out.println("Linked list forward:”);
    ListIterator<String> forward = linkedList.listIterator();
    while (forward.hasNext()) {
       System.out.println(forward.next());
    }
    System.out.println("Linked list backward:");
    ListIterator<String> backward;
    backward = linkedList.listIterator(linkedList.size());
    while (backward.hasPrevious()) {
       System.out.println(backward.previous() + " ");
     }
  }
}
```

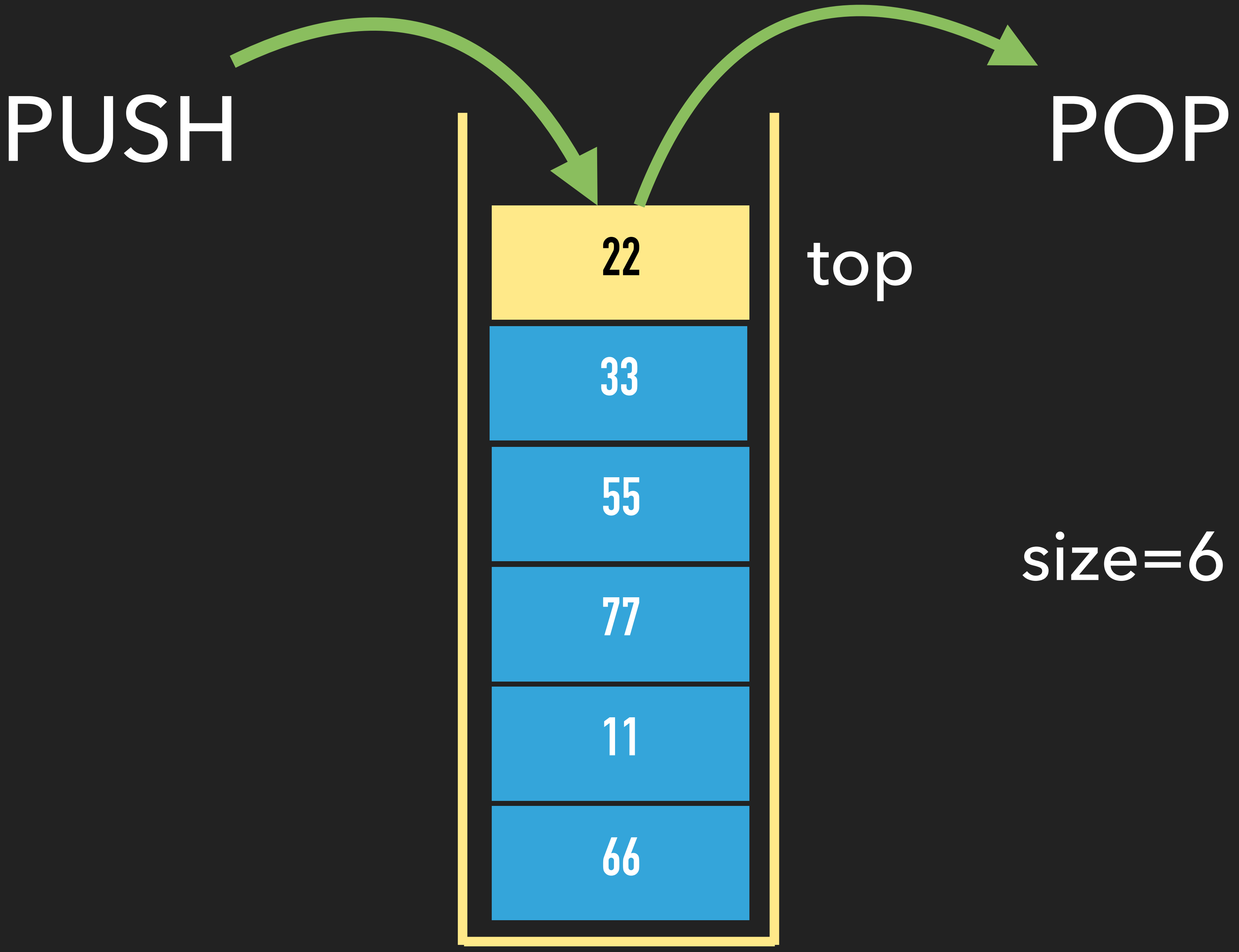
List

- 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

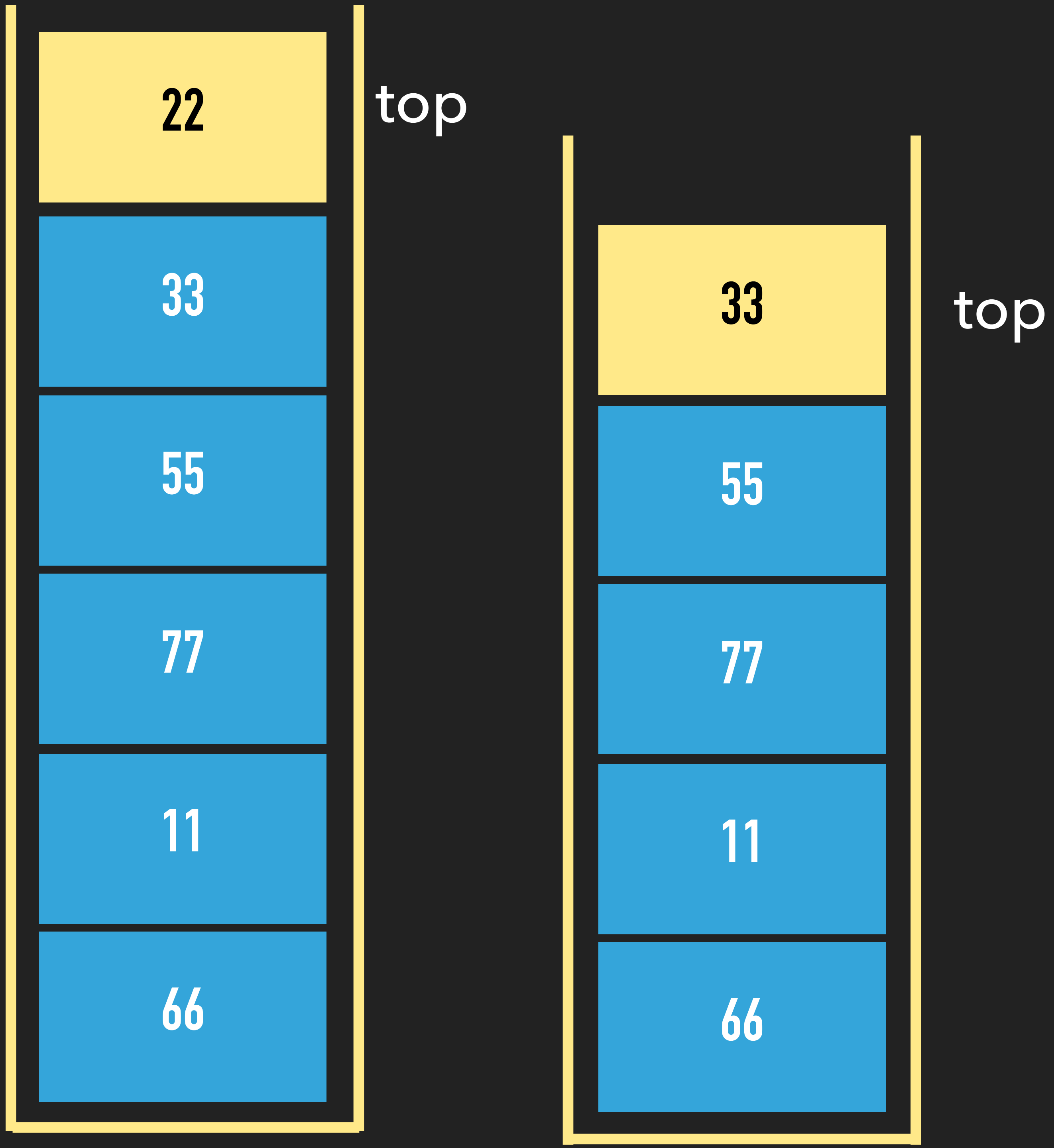
- LIFO structure - (Last In First Out)
- Access to the top of the stack only
- Operations: **`push()`**, **`pop()`**, and **`peek()`**
- Used for tracking method calls and arithmetic expression evaluation

Stack



Stack

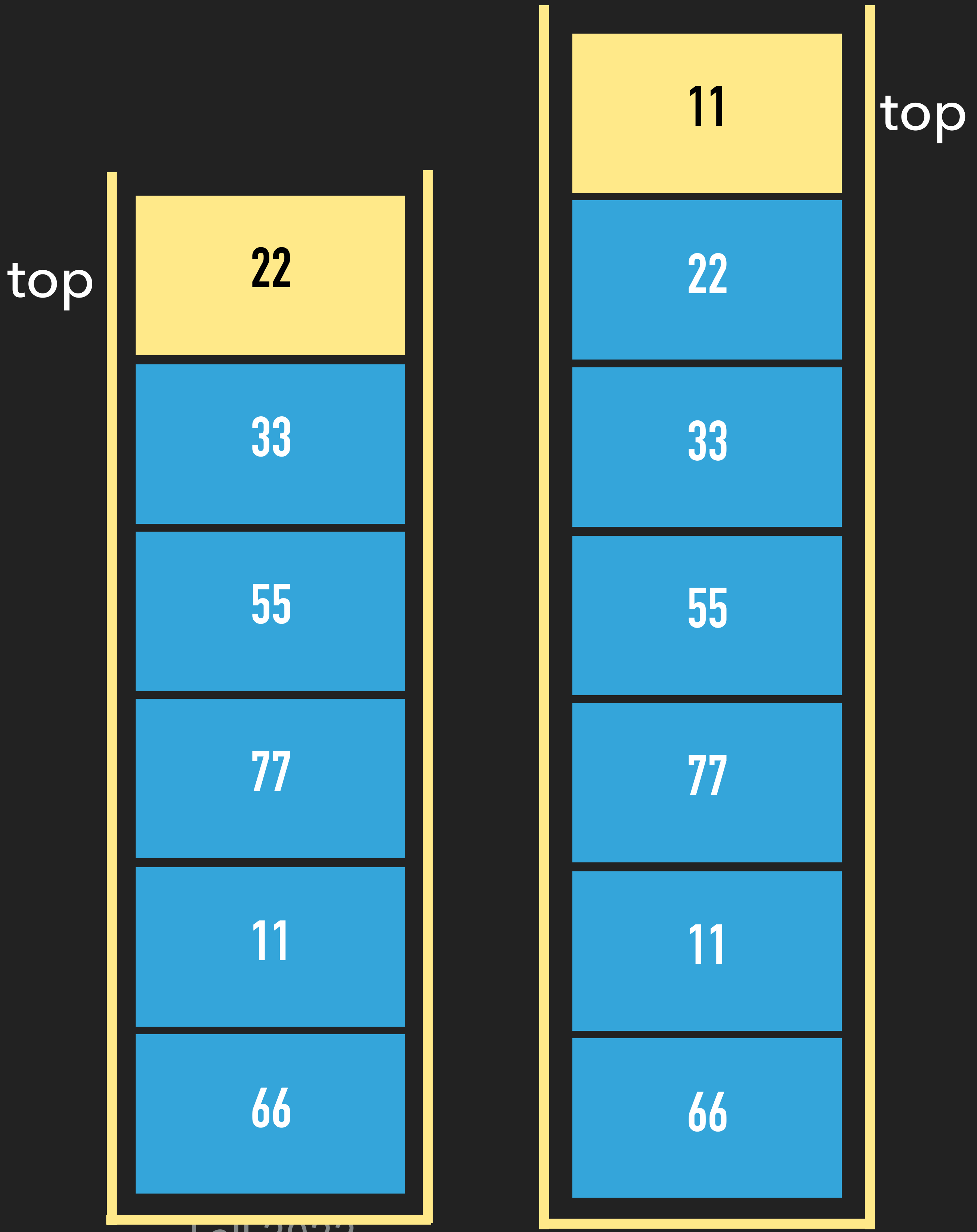
pop()



size=5

Stack

```
push(11)
```



size=7

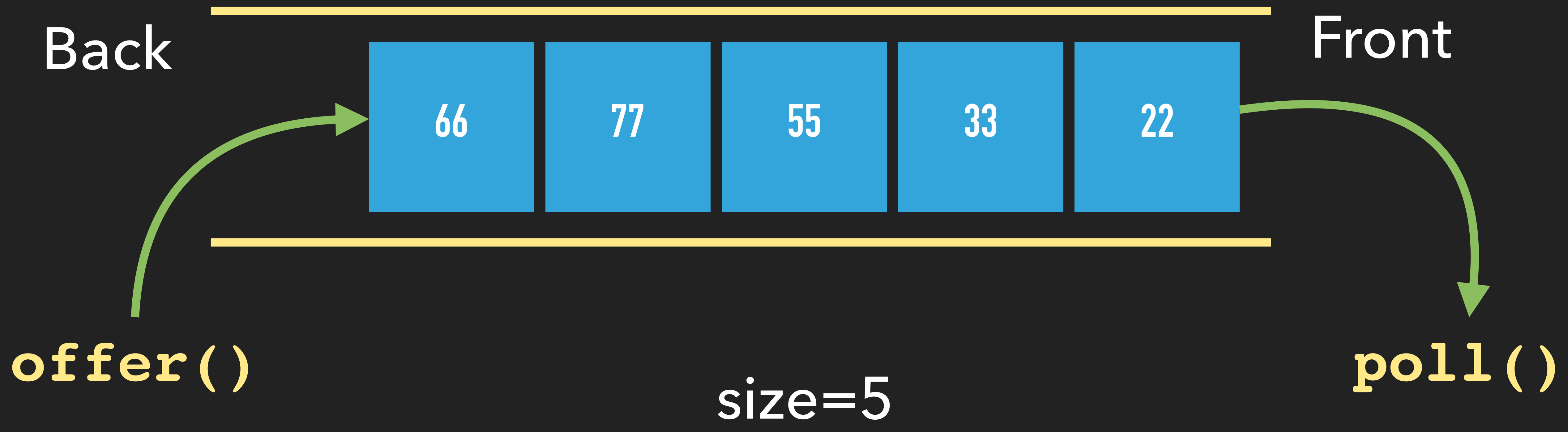
Stack

```
Java.util.Stack<E>
-------------------------
+Stack(): void
+isEmpty(): boolean
+peek(): E
+pop(): E
+push(E): void
+search(Object): int
```

Queue

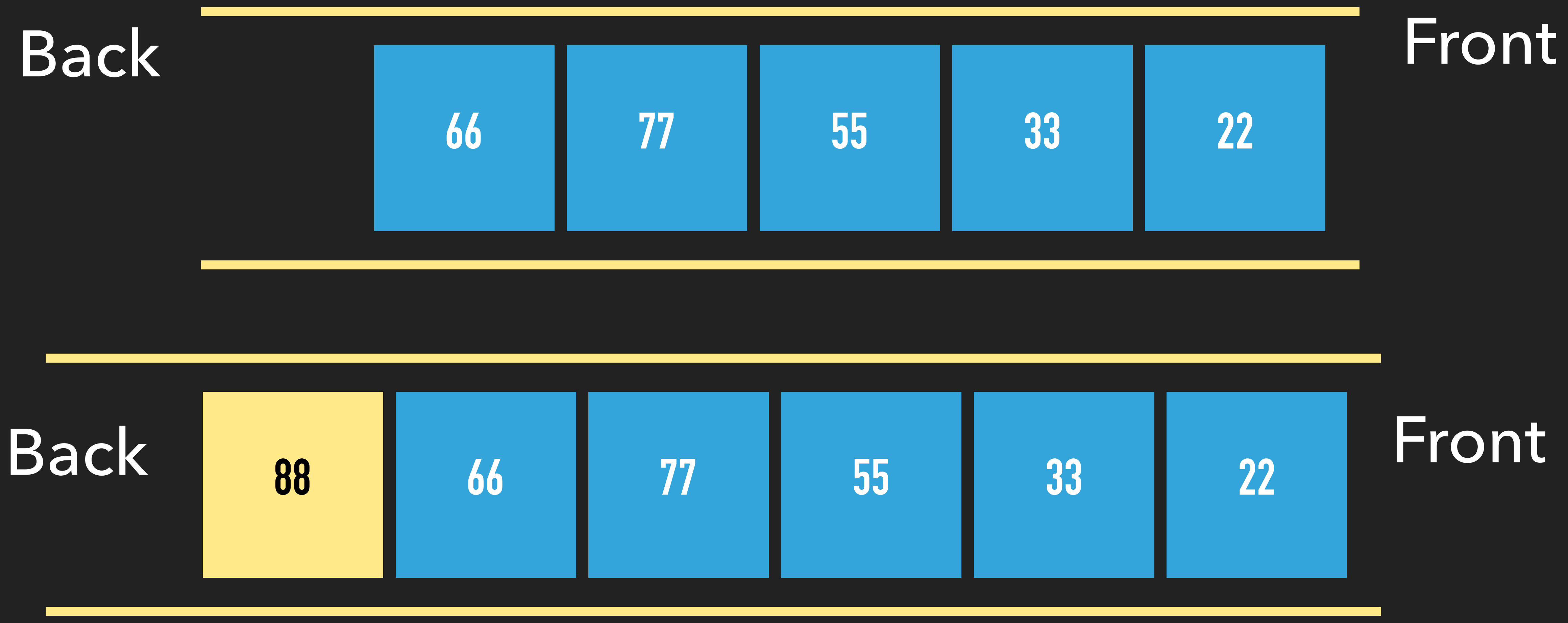
- **FIFO** structure - (First In First Out)
- Access at the front (or back) only
- Operations: `offer()`, `poll()`, and `peek()`
- Used for task scheduling and many real-life problem modeling
- Implemented as a linkedList in the Java API

Queue



Queue

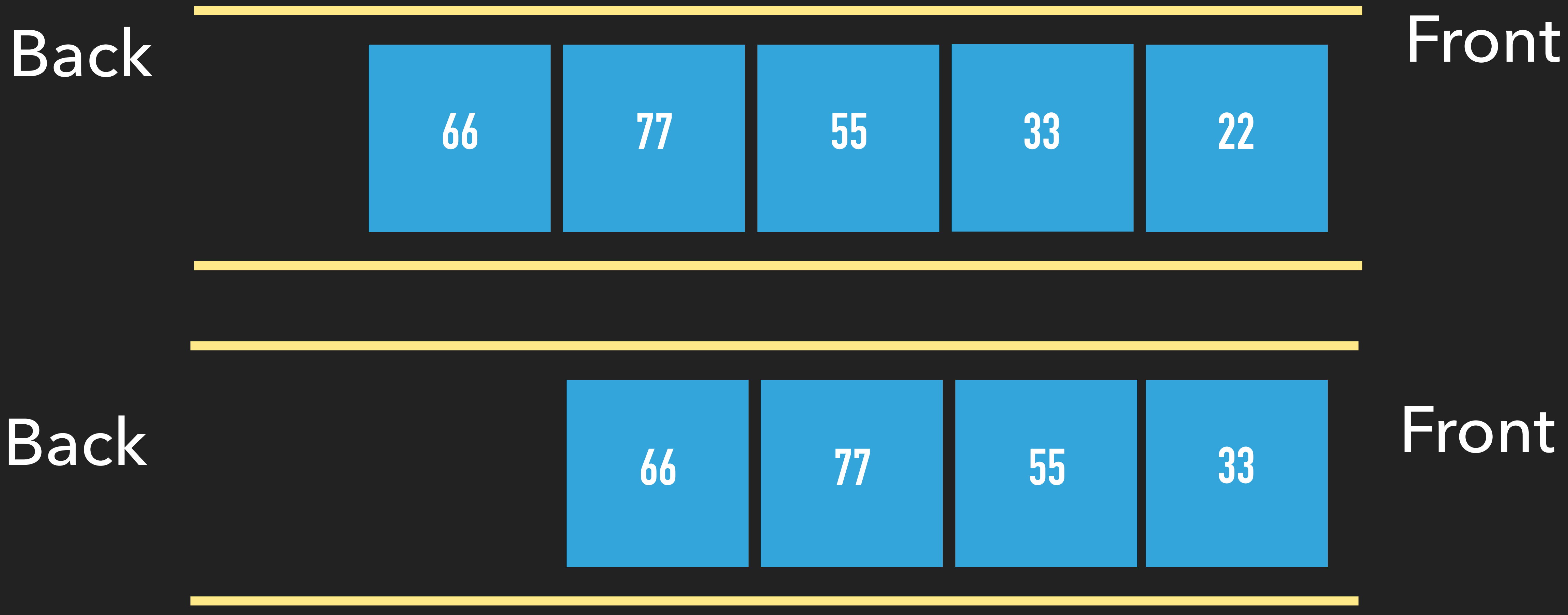
```
offer(88)
```



size=6

Queue

`poll()`

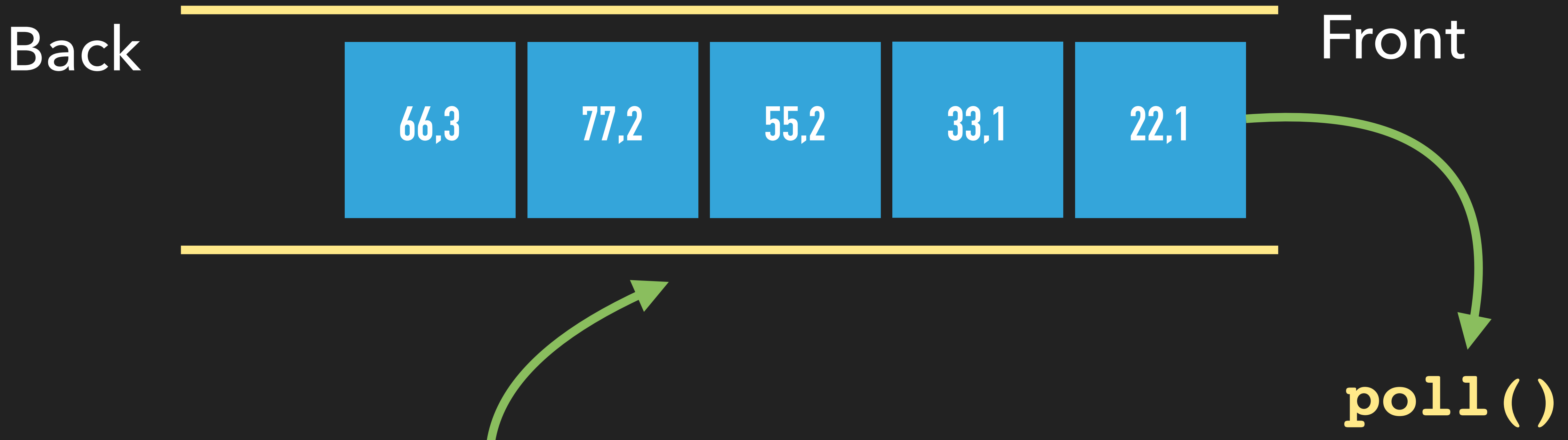


size=4

Priority Queue

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

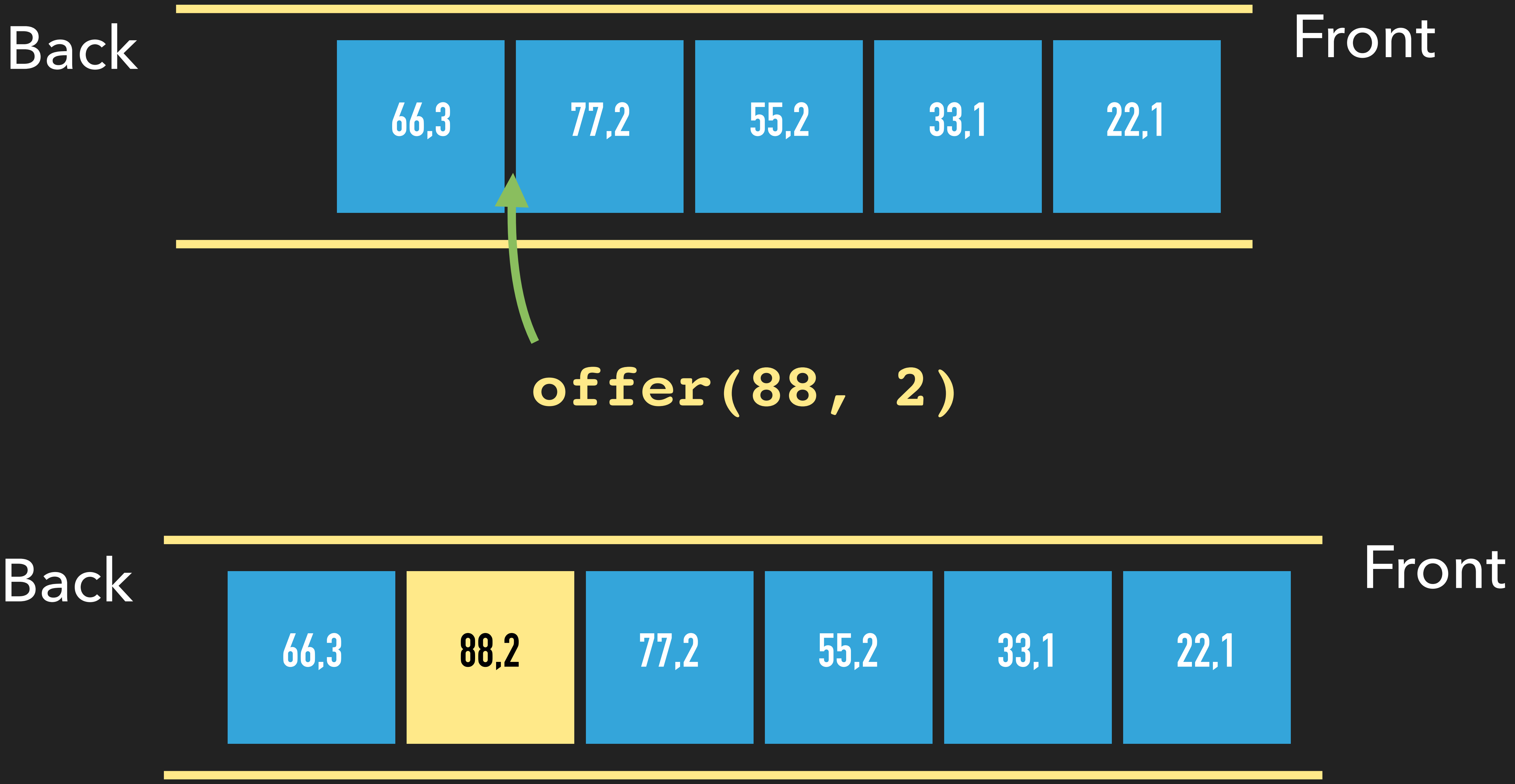
Priority Queue



`offer()` - dependent on the priority

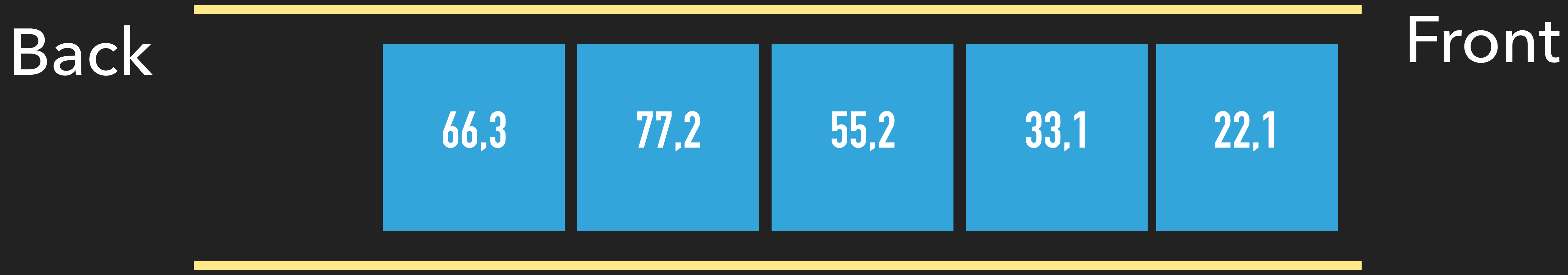
Priority Queue

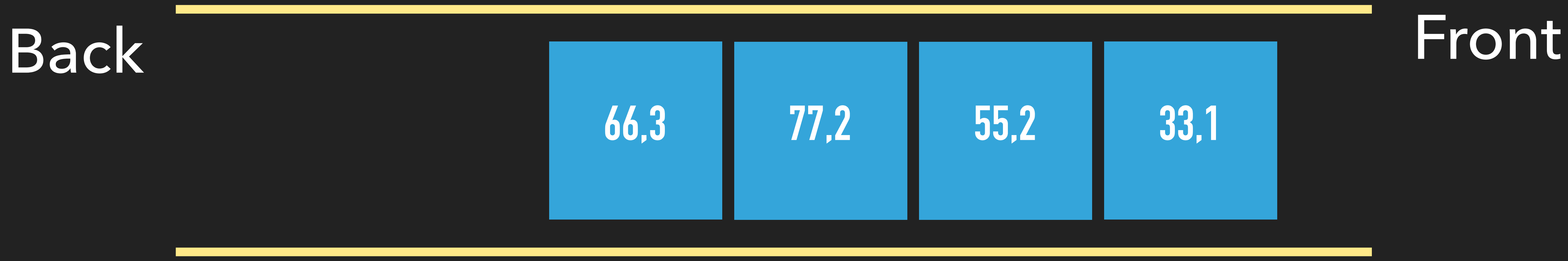
```
offer(88, 2)
```



Priority Queue

`poll()`





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
```

Java API data structures

```
public static void main(String[] args) {
  ArrayList<String> AL = new ArrayList<>();
  LinkedList<String> LL = new LinkedList<>();
  LinkedList<String> Q = new LinkedList<>();
  Stack<String> S = new Stack<>();
  PriorityQueue<String> PQ = new PriorityQueue<>();
  String[] fruits = {"Orange", “Kiwi",
                     "Pomegranate","Melon", “Apple",
                     "Banana", "Strawberry" };
  for(int i=0; i<fruits.length; i++) {
    AL.add(fruits[i]);
    LL.addFirst(fruits[i]);
    S.push(fruits[i]);
    Q.offer(fruits[i]);
    PQ.offer(fruits[i]);
  }
}
```

```
System.out.print("Array List: [");
 for(int i=0; i<fruits.length; i++) {
    System.out.print(AL.get(i) + " ");
 }
 System.out.println(“]");

 System.out.print("Linked List: [");
 for(Iterator<String> i=LL.iterator();i.hasNext();)
    System.out.print(i.next() + " ");
 System.out.println(“]");

 System.out.print("Queue: [");
 while(!Q.isEmpty())
    System.out.print(Q.poll() + " ");
 System.out.println(“]");

 System.out.print("Stack: [");
 while(!S.isEmpty())
    System.out.print(S.pop() + " ");
 System.out.println(“]");

 System.out.print("Priority Queue: [");
 while(!PQ.isEmpty())
    System.out.print(PQ.poll()+ " ");
 System.out.println("]");
```

DATA STRUCTURES

```
Array List: [Orange Kiwi Pomegranate Melon
             Apple Banana Strawberry ]

Linked List: [Strawberry Banana Apple Melon
              Pomegranate Kiwi Orange ]

Queue: [Orange Kiwi Pomegranate Melon Apple
        Banana Strawberry ]

Stack: [Strawberry Banana Apple Melon
        Pomegranate Kiwi Orange ]

Priority Queue: [Apple Banana Kiwi Melon
                 Orange Pomegranate Strawberry ]
```

Application

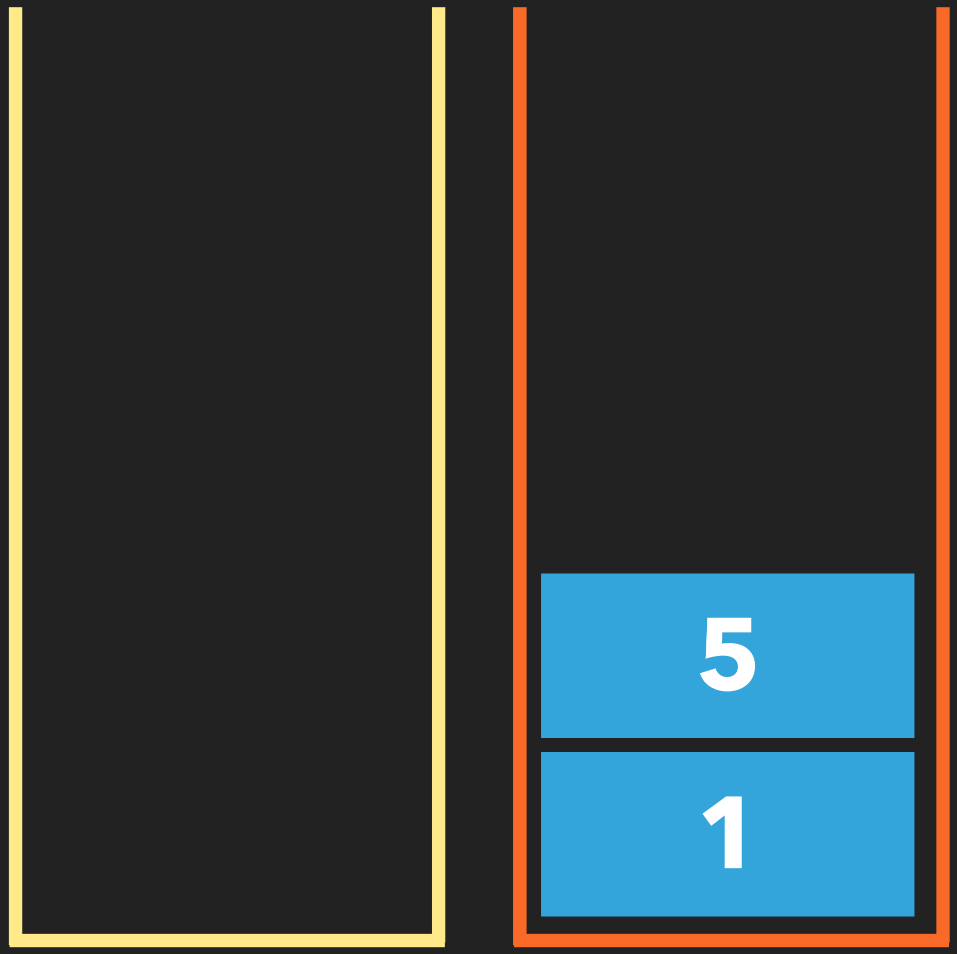
Evaluate arithmetic expressions using a stack

Infix expression: (1 + 5) * (8 - (4-1))

Postfix expression: 1 5 + 8 4 1 - - *

Application

✦ Postfix expression: 1 5 + 8 4 1 - - *

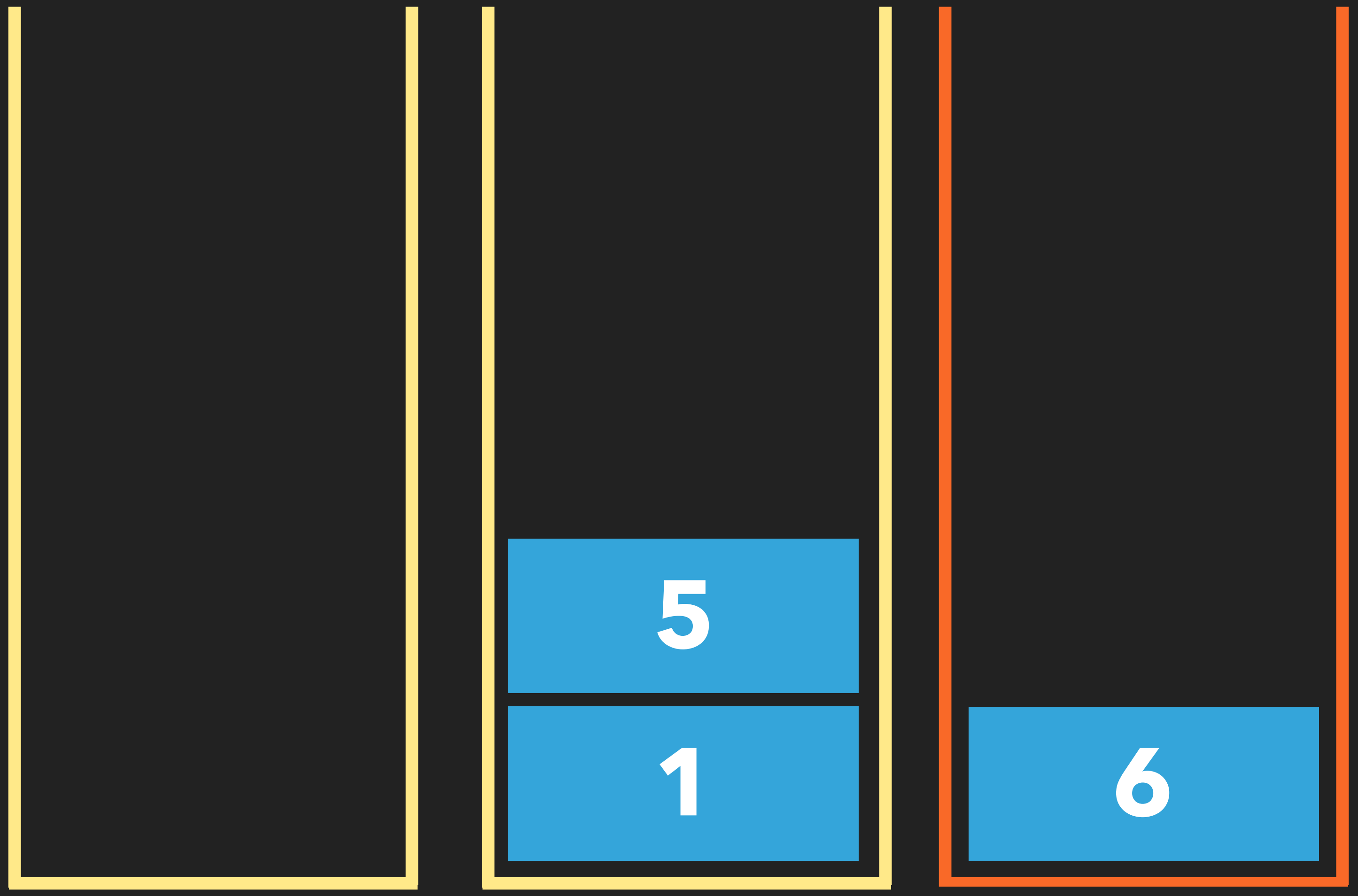


Stack
Empty

After
`push(1)`
`push(5)`

Application

- Postfix expression: 1 5 + 8 4 1 - - *

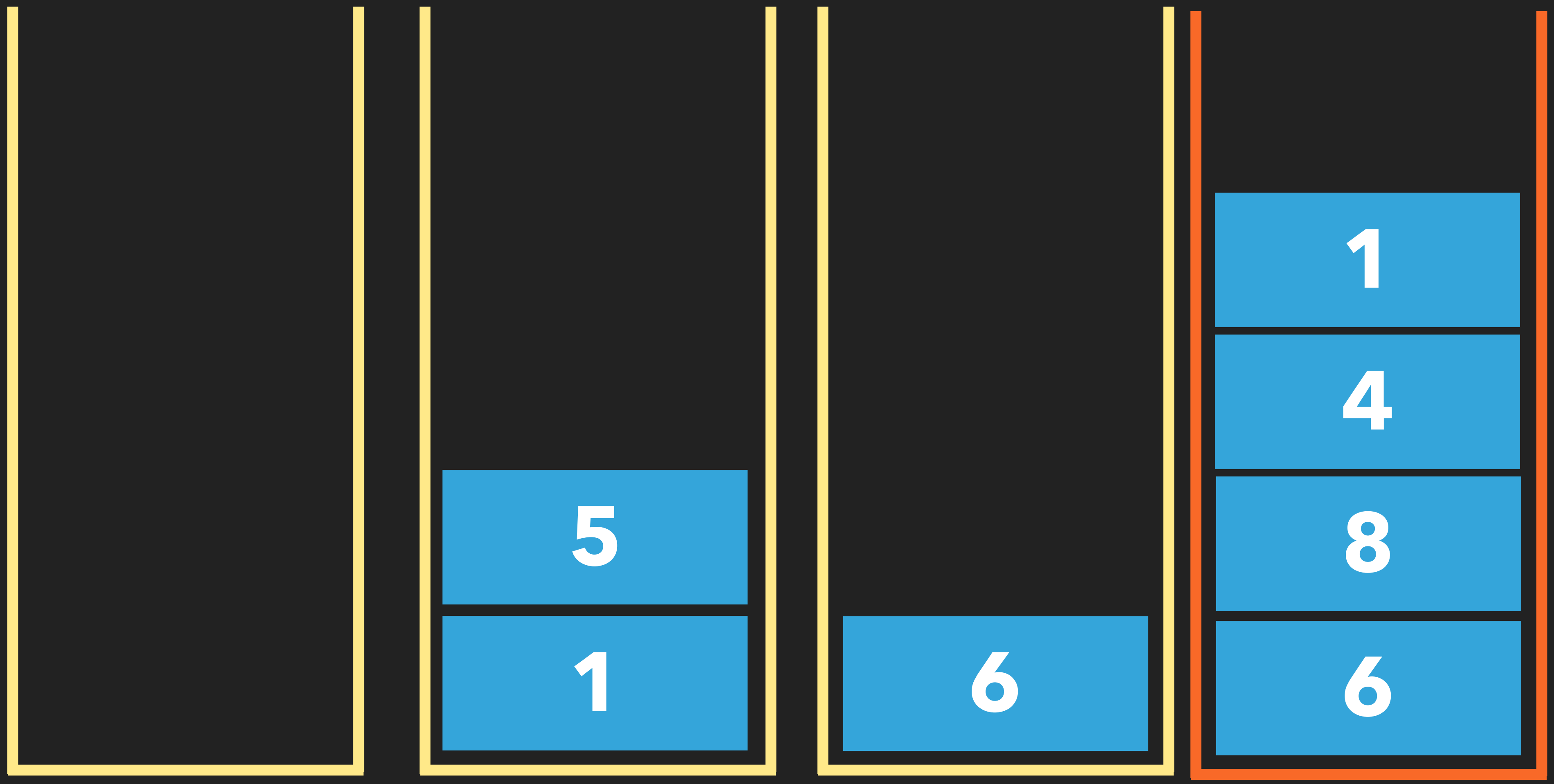


After

```
pop() - 5
pop() - 1
push(1+5)
```

Application

- Postfix expression: 1 5 + 8 4 1 - - *

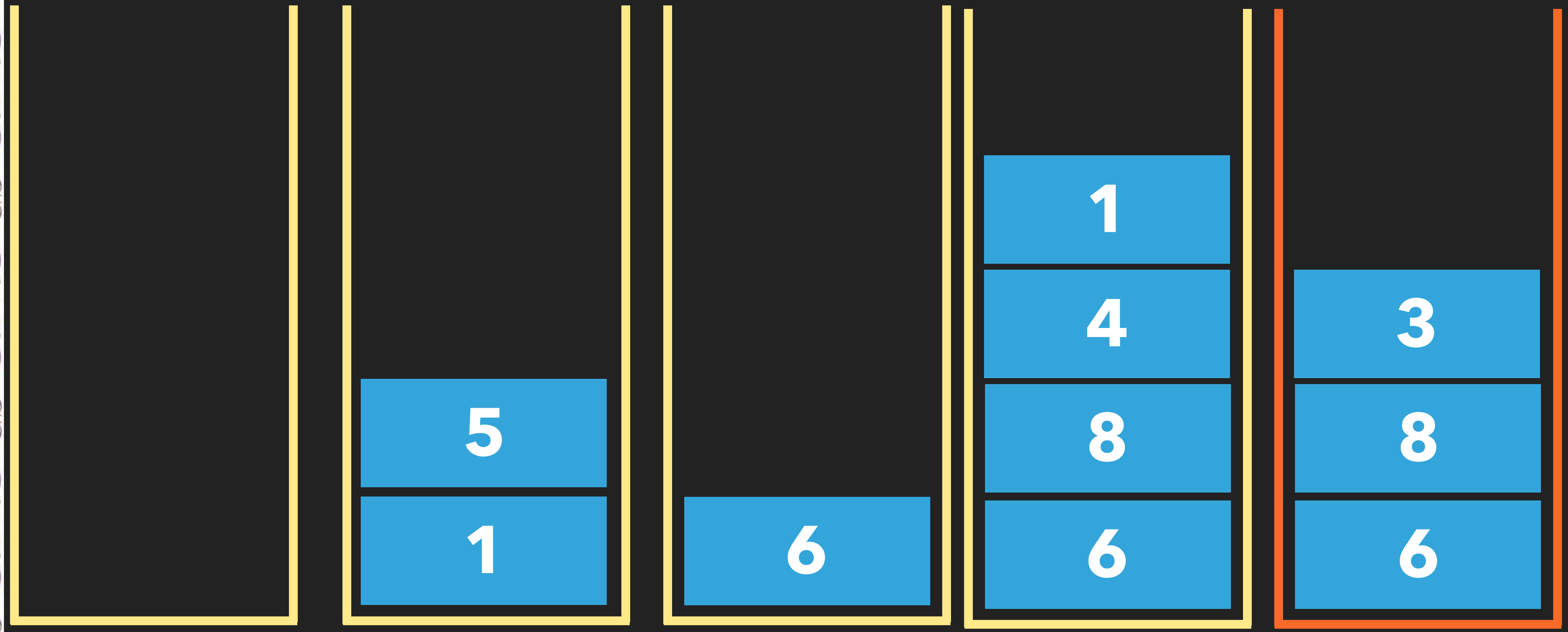


After

```
push(8)
push(4)
push(1)
```

Application

- Postfix expression: 1 5 + 8 4 1 - - *

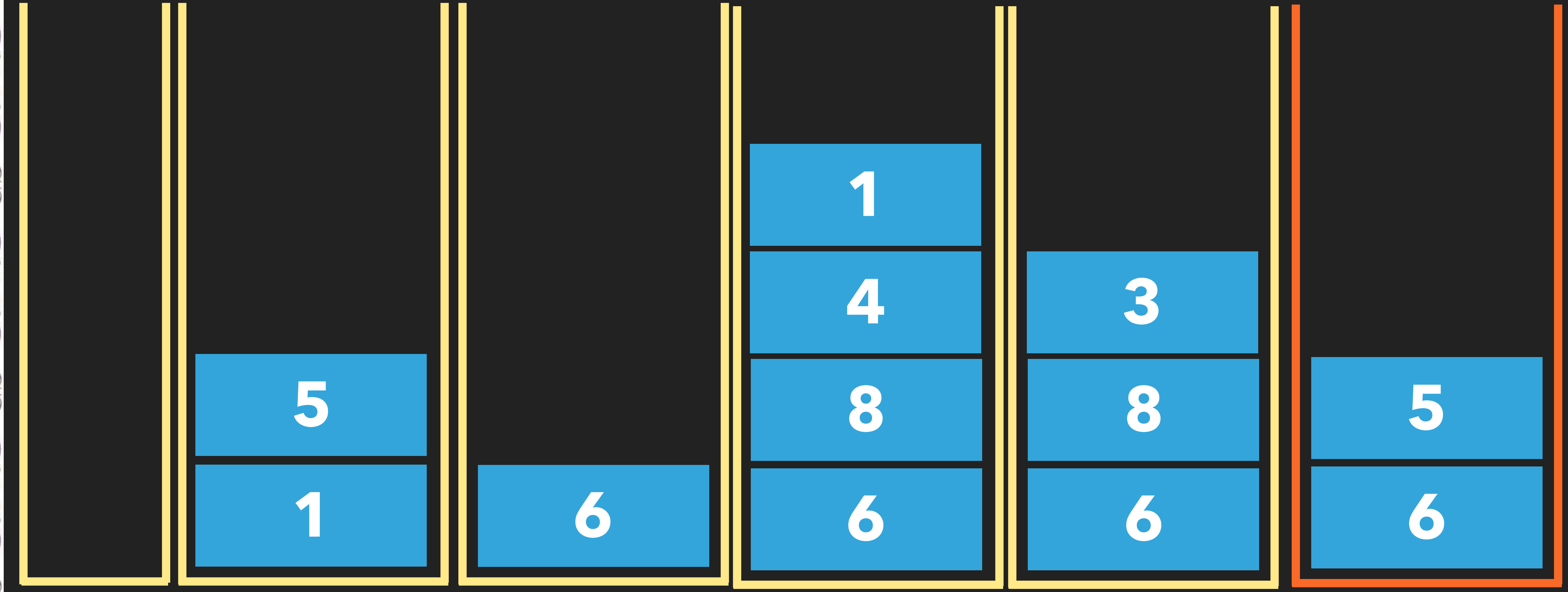


After

```
pop() - 1
pop() - 4
push(4-1)
```

Application

- Postfix expression: 1 5 + 8 4 1 - - *

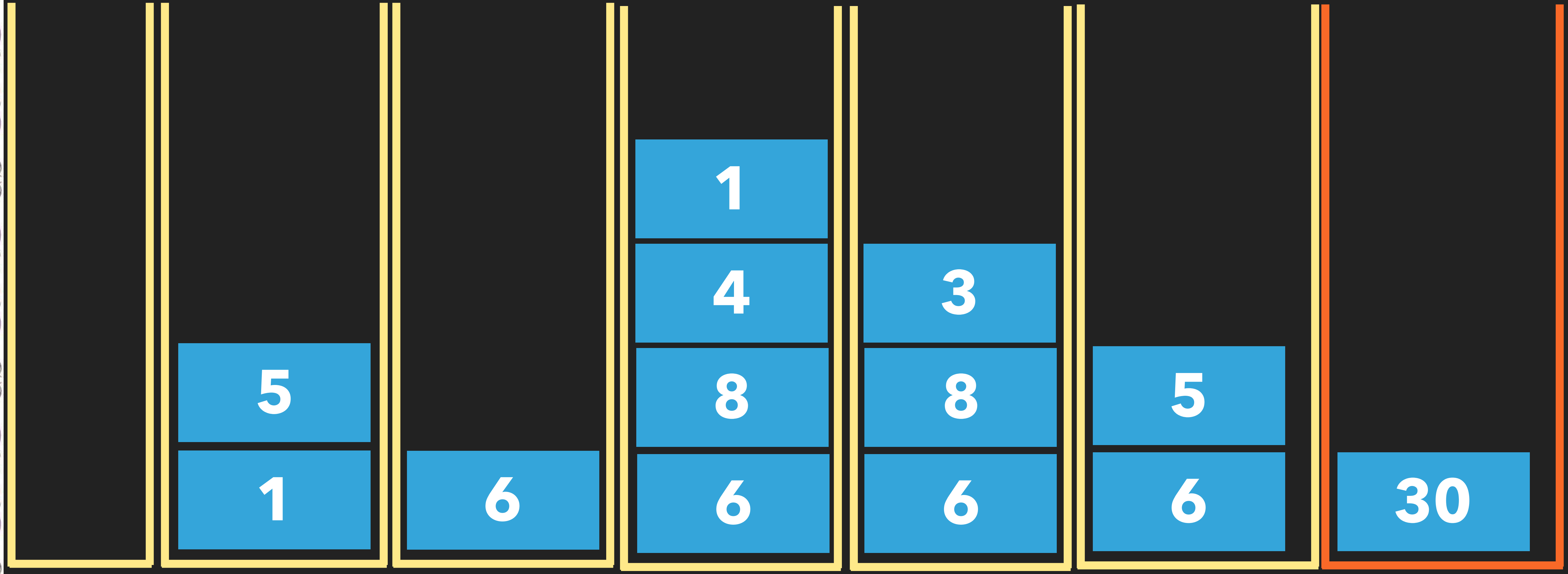


After

```
pop() - 3
pop() - 8
push(8-3)
```

Application

- Postfix expression: 1 5 + 8 4 1 - - *



After

```
pop() - 5
pop() - 6
push(6*5)
```

Practice

- Evaluate the postfix expression

```
12 25 5 1 / / * 8 7 + -
```

- Using a stack - show all the steps

Application

- Algorithm to process a postfix expression

```
1.Create an empty stack
2.While (!end of postfix expression)
    1.Read the next token (operand or operator)
    2.If the token is an operand, push(token)
      in the stack
    3.If the token is an operator, pop two
      values, perform the operation, and push
      the result in the stack
3.Pop the result from the stack
4.If the stack is not empty, “postfix
 expression malformed, else display result
```

Summary

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