Java Data Structures

Tips and Tricks - Part 1

Micah Stairs

Outline

- Time Complexity
- Arrays
- Lists
- Sets
- Queues
- Stacks
- Deques

Time Complexity

Time Complexity

Constant Time: **$O(1)$**

Logarithmic Time: **$O(\log n)$**

Linear Time: **$O(n)$**

Linearithmic Time: **$O(n \log n)$**

Quadratic Time: **$O(n^2)$**

Cubic Time: **$O(n^3)$**

Exponential Time: **$O(b^n)$** where $b > 1$

Factorial Time: **$O(n!)$**

Time Complexity

You can perform roughly 1 billion basic operations per second (depends on machine, language, etc.).

Some operations are more expensive than others. For example, method calls are slower than accessing an array.

Time Complexity

Since most operations we’ll end up doing aren’t basic, and to account for the any constant factors, the number of operations we aim for is going to be lower, let’s say 1 million.

This will help give us an rough idea of what the time complexity of our solution needs to be based on the input size.

Time Complexity

Input Size	Expected Complexity
≤ 10	$O(n!)$ or $O(b^n)$
≤ 20	$O(2^n)$
≤ 100	$O(n^3)$
≤ 1,000	$O(n^2)$
≤ 100,000	$O(n \log n)$
≤ 1,000,000	$O(n)$
> 1,000,000	$O(1)$

Arrays

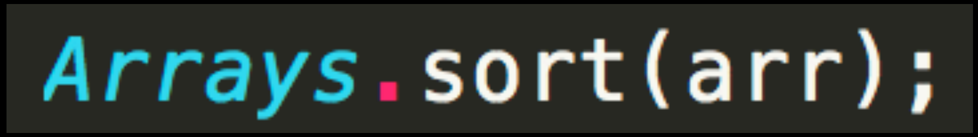
Arrays

An array is a fixed-sized container used to hold elements.

3	5	2	0	-5	3	2	1	97

Arrays

java.util.Arrays contains many useful methods for dealing with arrays.

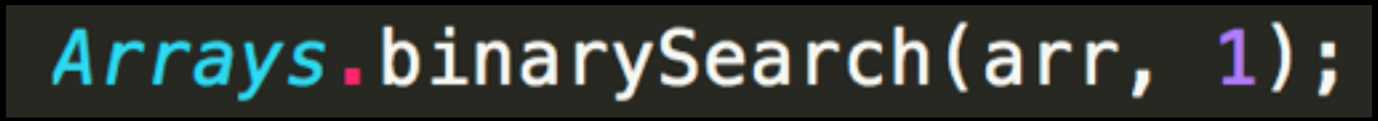


```
Arrays.sort(arr);
```

0	1	2	3	4	5	6	7	8
-5	0	1	2	2	3	3	5	97

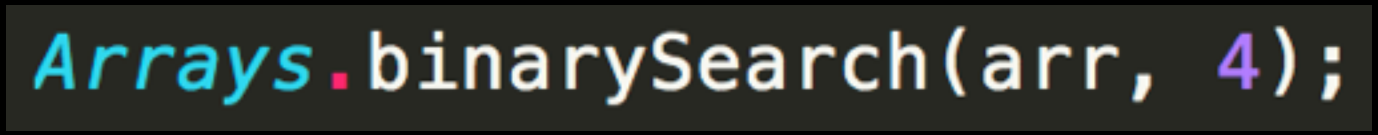
Arrays

If your array is sorted, you can do binary searches on it to find elements faster.



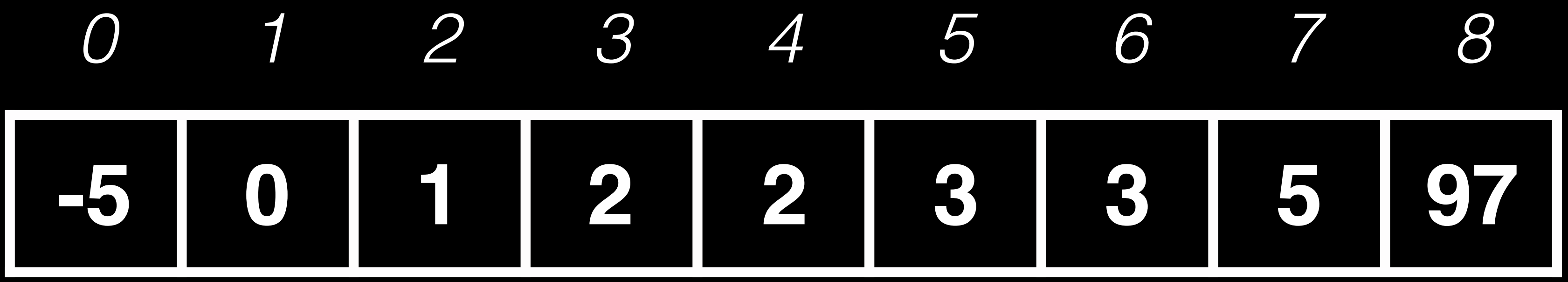
```
Arrays.binarySearch(arr, 1);
```

Outputs “2”



```
Arrays.binarySearch(arr, 4);
```

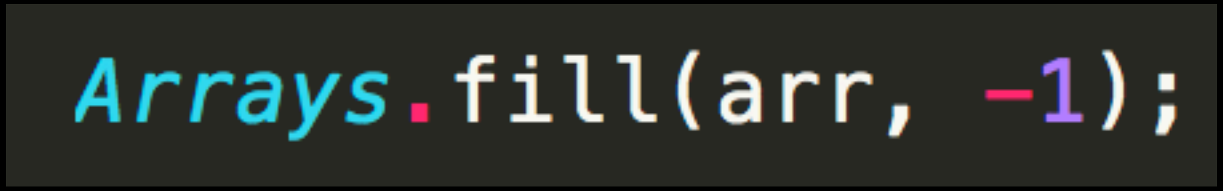
Outputs “-8”



0	1	2	3	4	5	6	7	8
-5	0	1	2	2	3	3	5	97

Arrays

Sometimes you just need to fill your array with something and you're too lazy to use a "for loop".

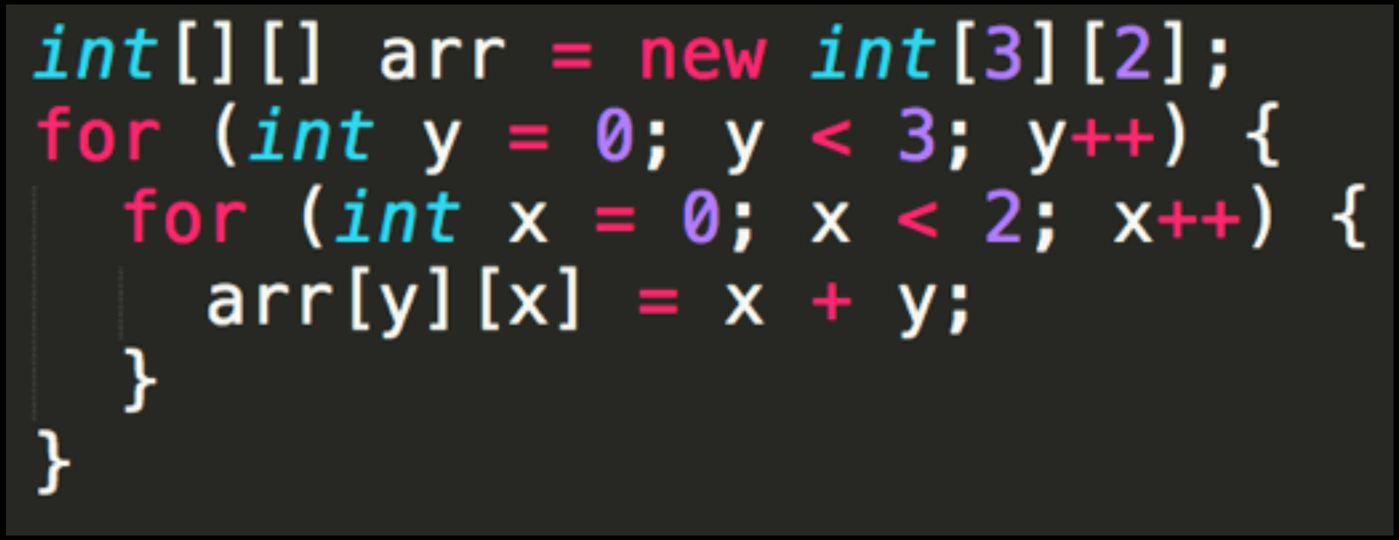


```
Arrays.fill(arr, -1);
```

0	1	2	3	4	5	6	7	8
-1	-1	-1	-1	-1	-1	-1	-1	-1

Arrays

Multi-dimensional arrays are useful to represent things such as grids.

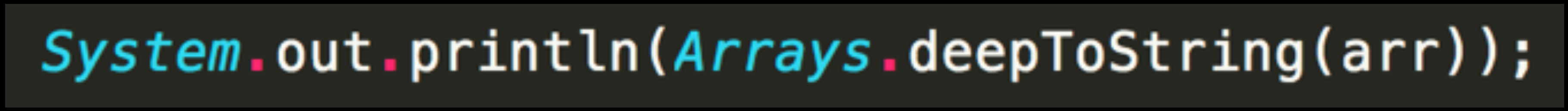


```java
int[][] arr = new int[3][2];
for (int y = 0; y < 3; y++) {
  for (int x = 0; x < 2; x++) {
    arr[y][x] = x + y;
  }
}
```

	0	*1*
0	**0**	**1**
1	**1**	**2**
2	**2**	**3**

Arrays

Arrays.deepToString() is more useful than *Arrays.toString()* for multi-dimensional strings.



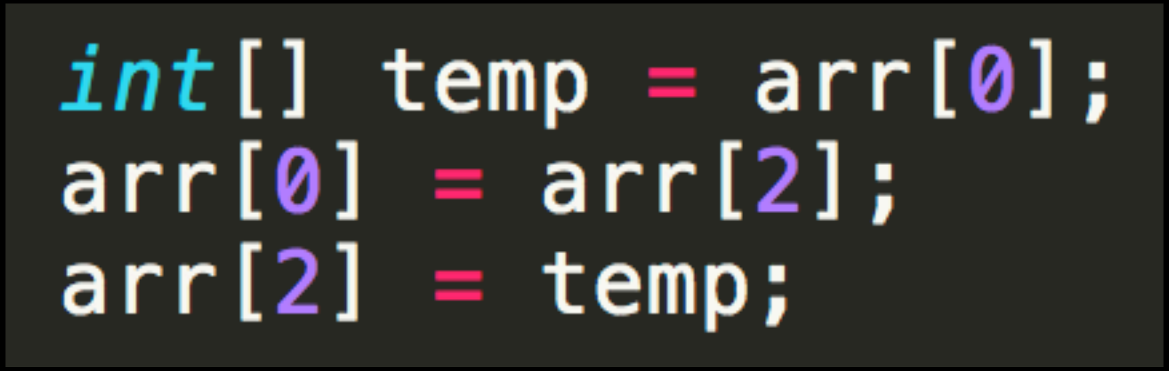
```
System.out.println(Arrays.deepToString(arr));
```

Outputs “[[0, 1], [1, 2], [2, 3]]

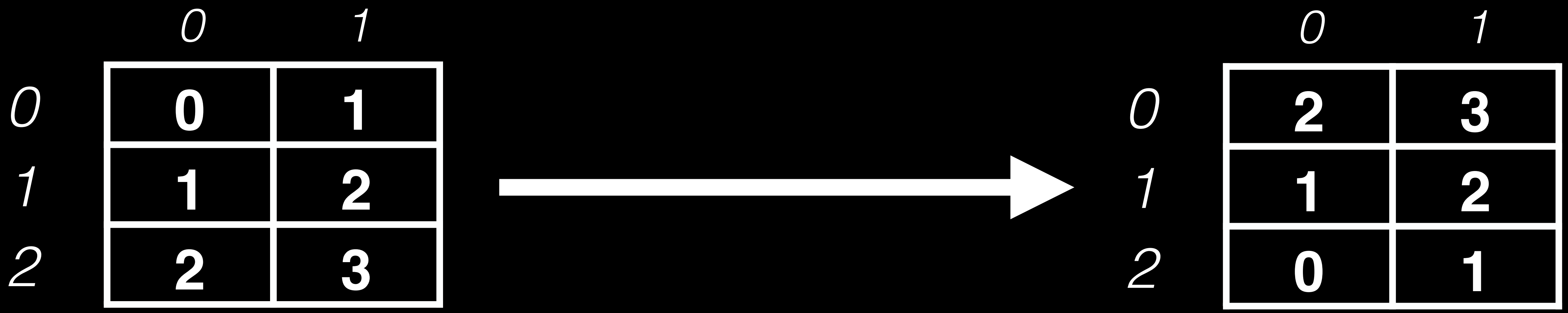
	0	*1*
0	**0**	**1**
1	**1**	**2**
2	**2**	**3**

Arrays

You can efficiently swap two rows of a 2D array in constant time.



```
int[] temp = arr[0];
arr[0] = arr[2];
arr[2] = temp;
```



	0	1
0	**0**	**1**
1	**1**	**2**
2	**2**	**3**

	0	1
0	**2**	**3**
1	**1**	**2**
2	**0**	**1**

Arrays

Pros:

- O(1) updates
- O(1) lookups

Cons:

- Not easily resizable

Lists

Lists

Dynamic container to hold elements.

3	5	2	0	-5	3	2	1	...

Lists

The most useful implementations of the *List* interface are *ArrayList* and *LinkedList.*

The time complexity of each operation is dependent on the implementation.

3	5	2	0	-5	3	2	1	...

Lists

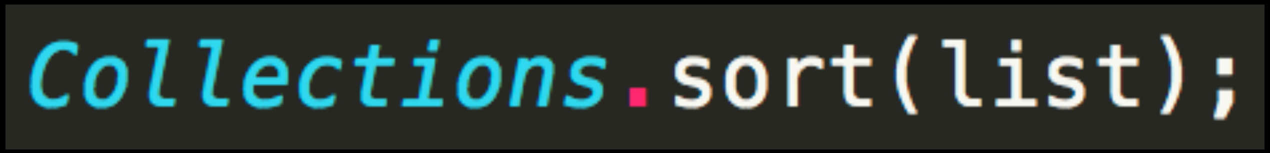
The *ArrayList* class is best at operations such as *get()* and *set()*. Adding an element at a specific position is O(n).

The *LinkedList* class is best at operations such as *add()* and *remove()*. Iterators can be used to efficiently add or remove elements.

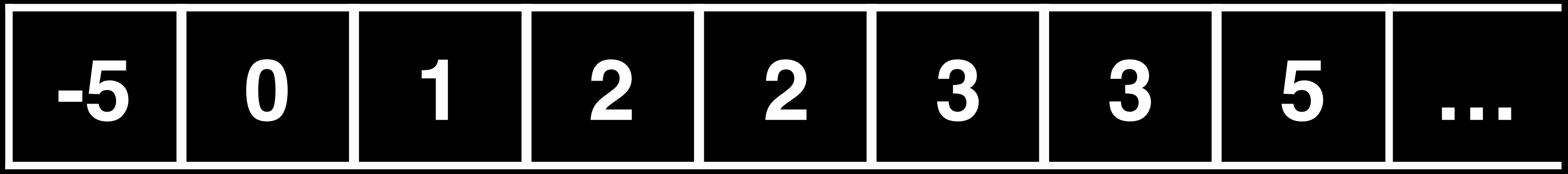
3	5	2	0	-5	3	2	1	...

Lists

java.util.Collections contains many useful methods for dealing with collections such as lists.



```
Collections.sort(list);
```



-5	0	1	2	2	3	3	5	…

Lists

Other methods in *java.util.Collections* include *reverse()*, *swap()*, *rotate()*, and *binarySearch()*.

-5	0	1	2	2	3	3	5	...

Lists

Pros:

- Resizable

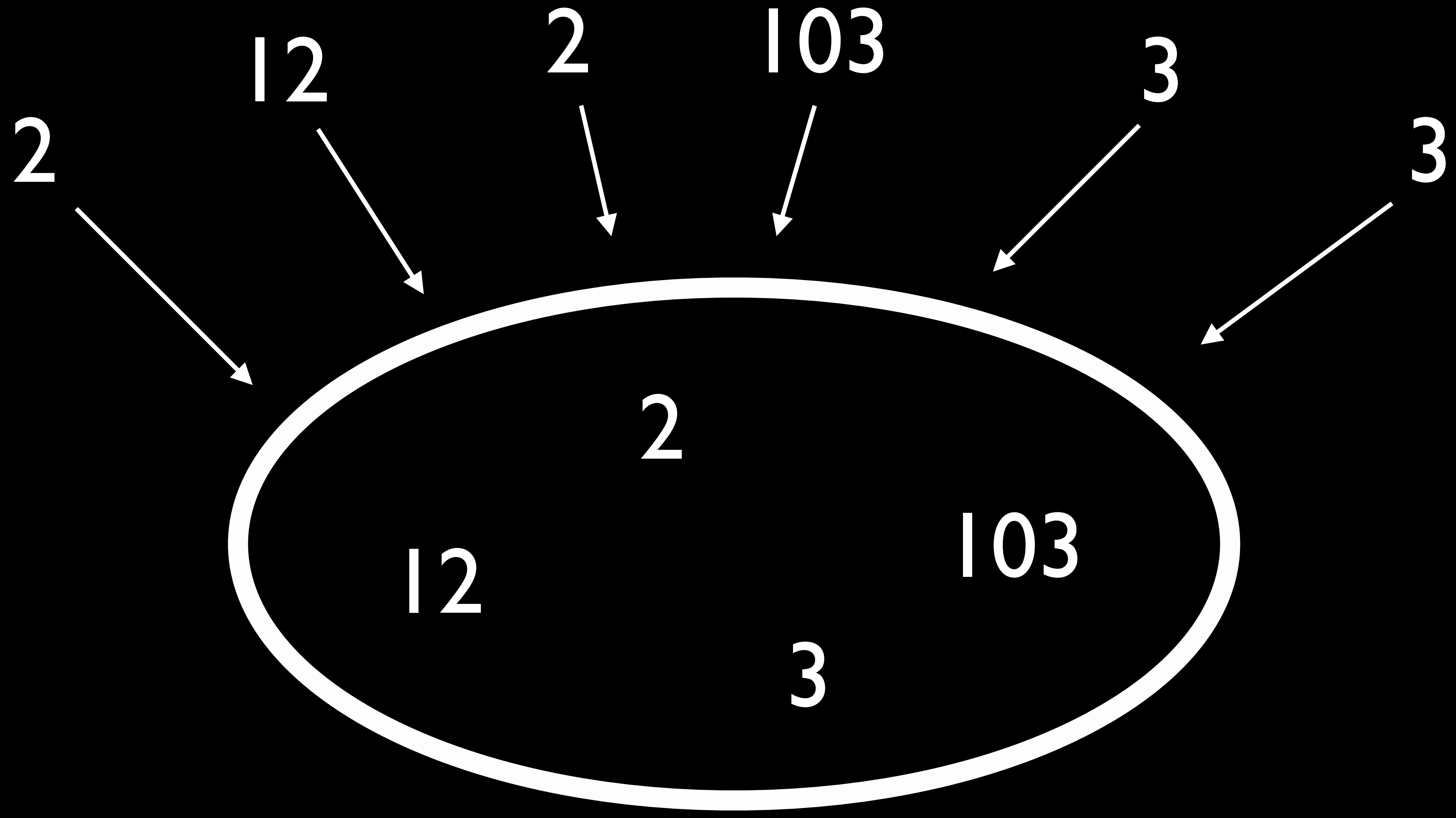
Cons:

- Not easily extendable to multiple dimensions.
- Can’t use with primitive types.

Sets

Sets

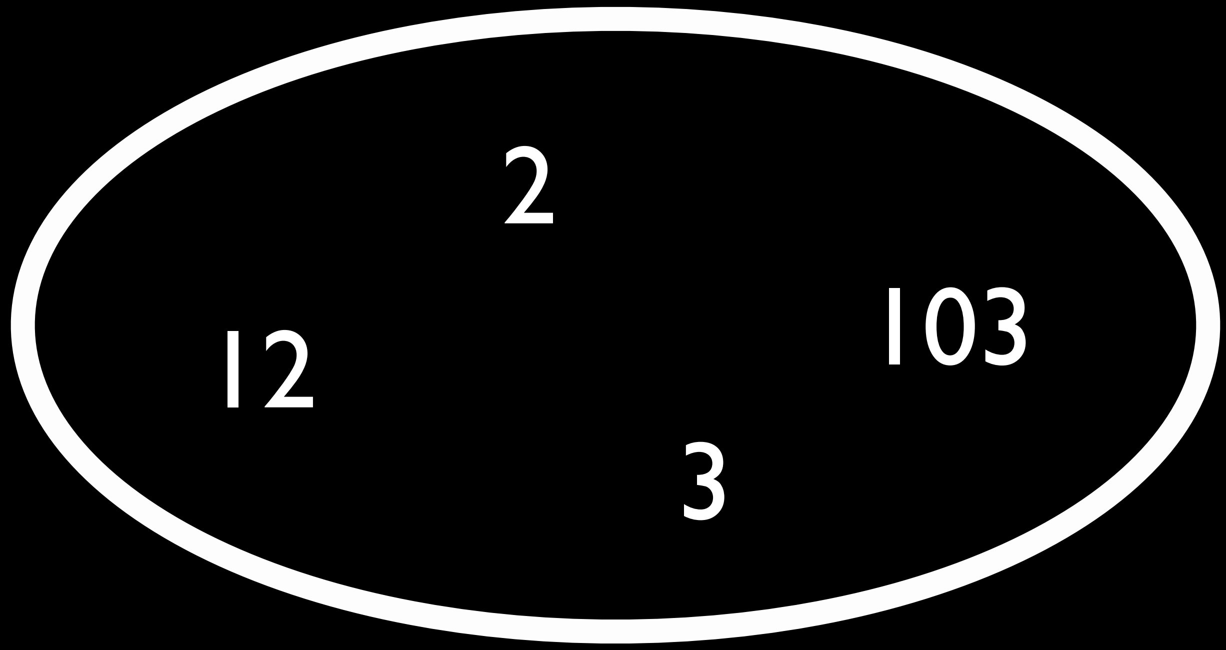
A container used to hold or count unique elements.



Sets

The most useful implementations of the *Set* interface are *HashSet* and *TreeSet*.

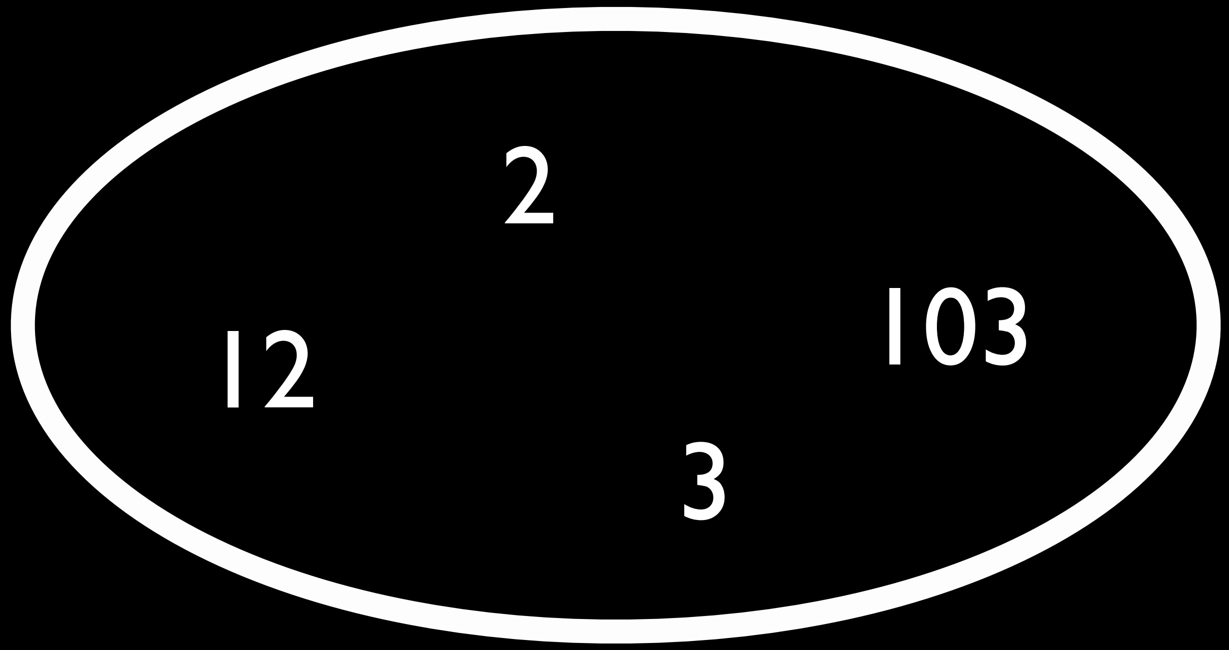
The time complexity of each operation is dependent on the implementation.



Sets

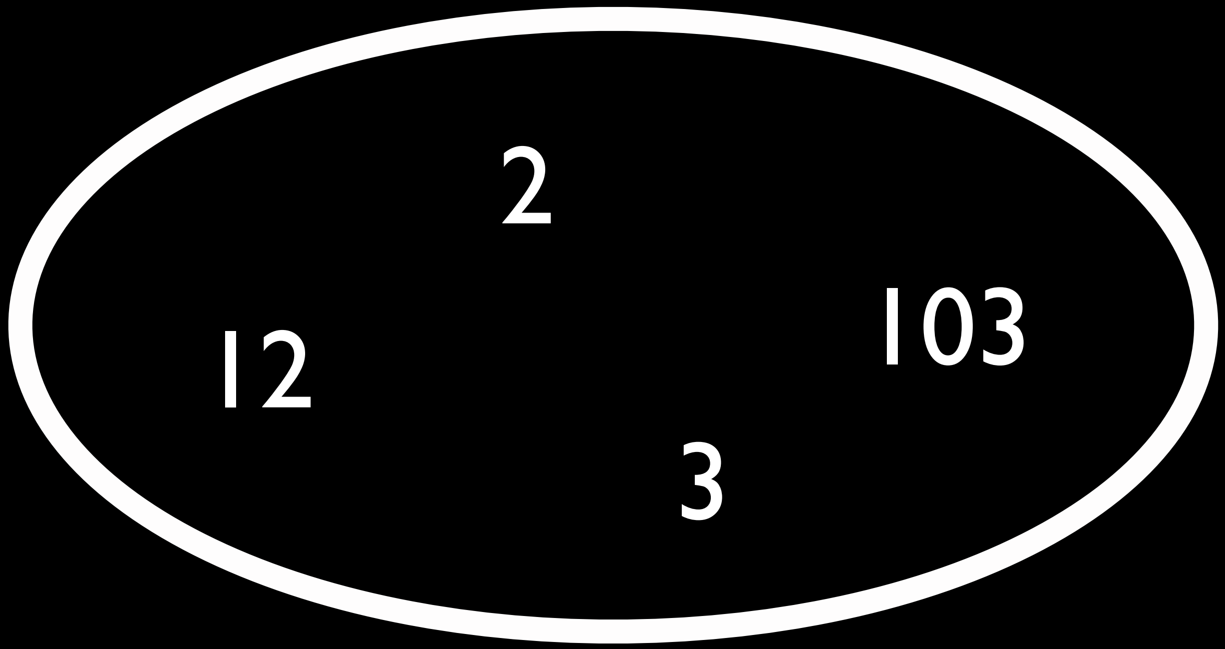
The *HashSet* class have expected constant time *add()*, contains*()*, and *remove()* operations.

The *TreeSet* has logarithmic time *add()*, contains*()*, and *remove()* operations, but the elements are sorted.



Sets

The *addAll()* method allows you to compute the union of two sets, and the *retainAll()* method allows you to compute the intersection of two sets.



Sets

Pros:

- Efficiently manages unique elements.
- The *contains()* method is more efficient than what an array or list could achieve.

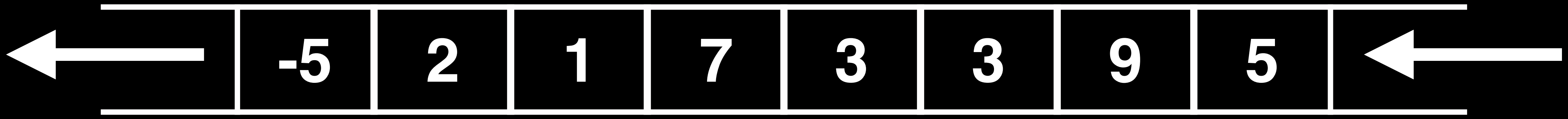
Cons:

- Can't index into.

Queues

Queues

An container which is First-In-First-Out (FIFO).

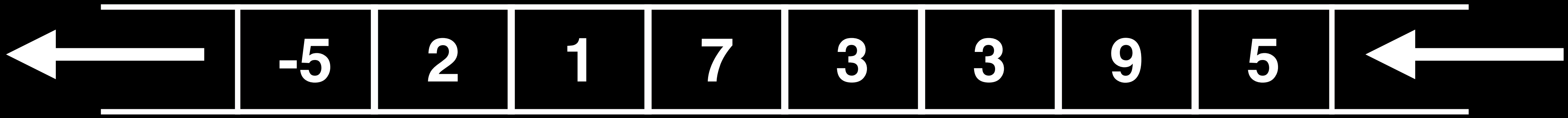


-5 2 1 7 3 3 9 5

Queues

The most useful implementations of the *Queue* interface are *LinkedList* and *PriorityQueue*.

We will talk about Priority Queues next week.

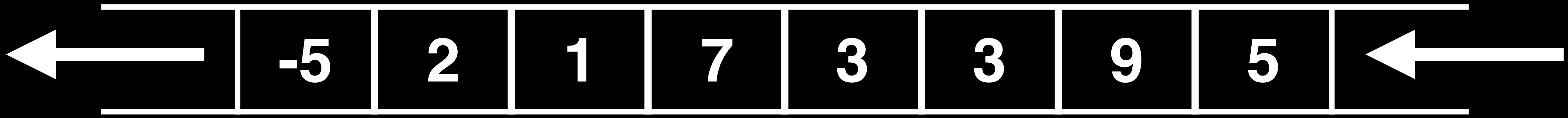


-5 2 1 7 3 3 9 5

Queues

The most common use of a Queue is for doing a Breadth-First Search (BFS).

More on this next week.



Queues

Pros:

- Can easily process elements in the order that they were inserted in.
- Can be used to efficiently keep track of the x most recently added elements.

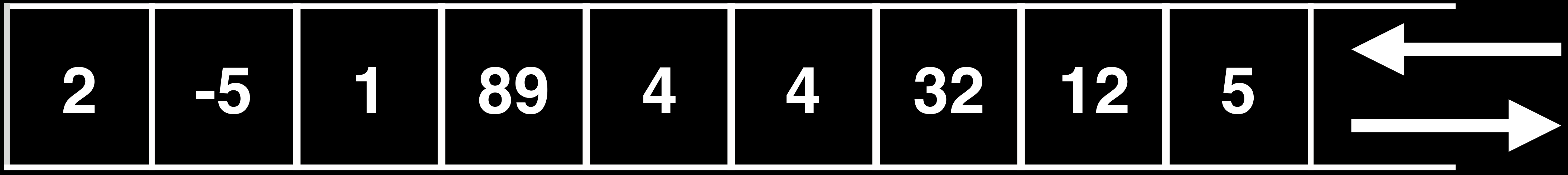
Cons:

- Can’t index into or add an element at a specific position.

Stacks

Stacks

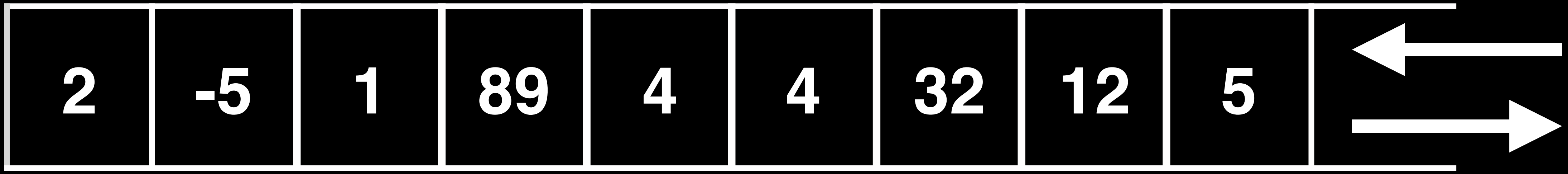
An container which is Last-In-First-Out (LIFO).



2	-5	1	89	4	4	32	12	5	

Stacks

Stacks are useful for tasks such as matching brackets, simulating recursion, and syntax parsing.



2	-5	1	89	4	4	32	12	5	

Stacks

Problem: Given a string made up of these bracket symbols: ( ) [ ] { }, determine whether the brackets properly match.

[{}()] → true

(()()) → true

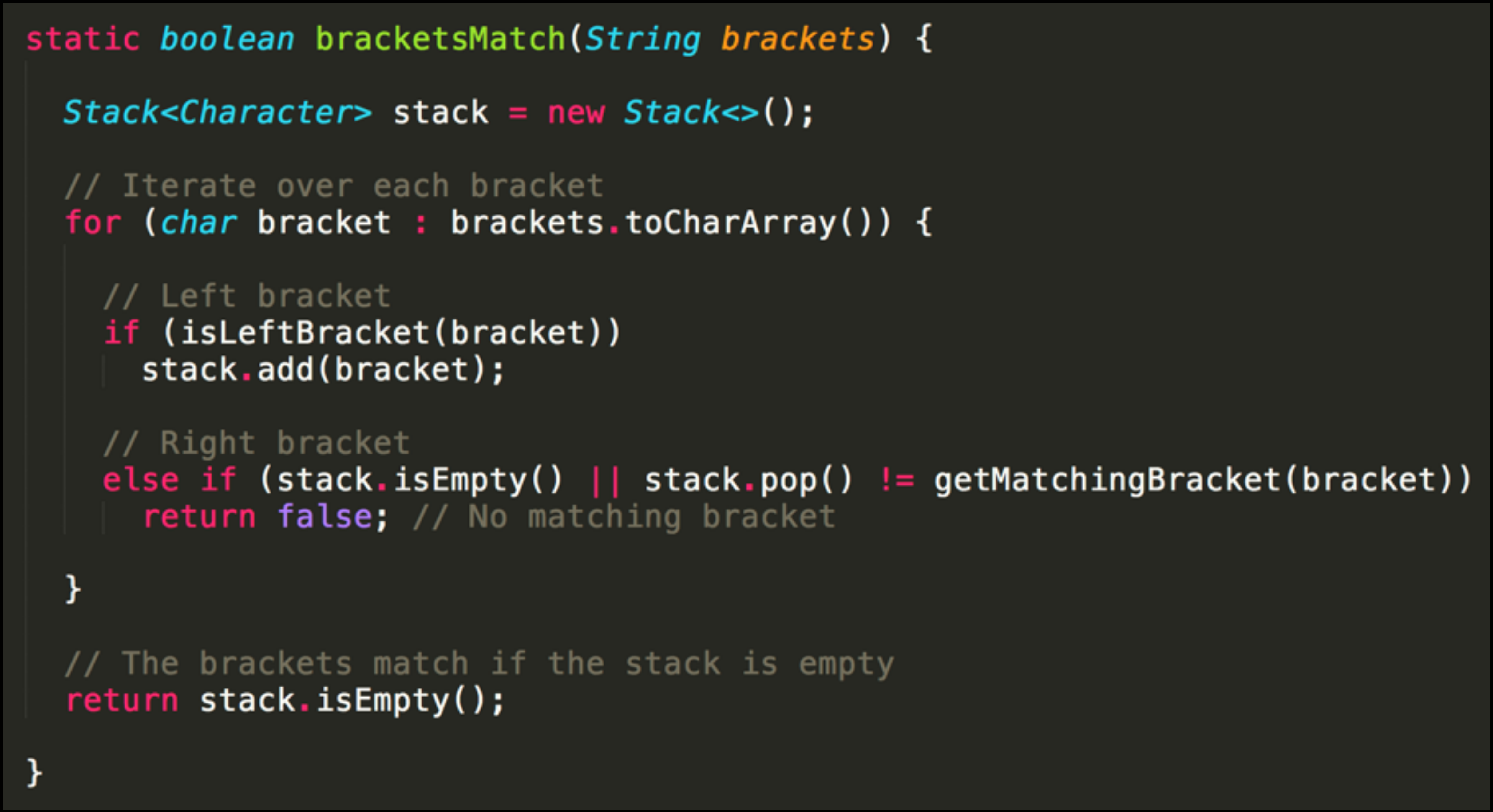
Stacks

Problem: Given a string made up of these bracket symbols: ( ) [ ] { }, determine whether the brackets properly match.

{] → false

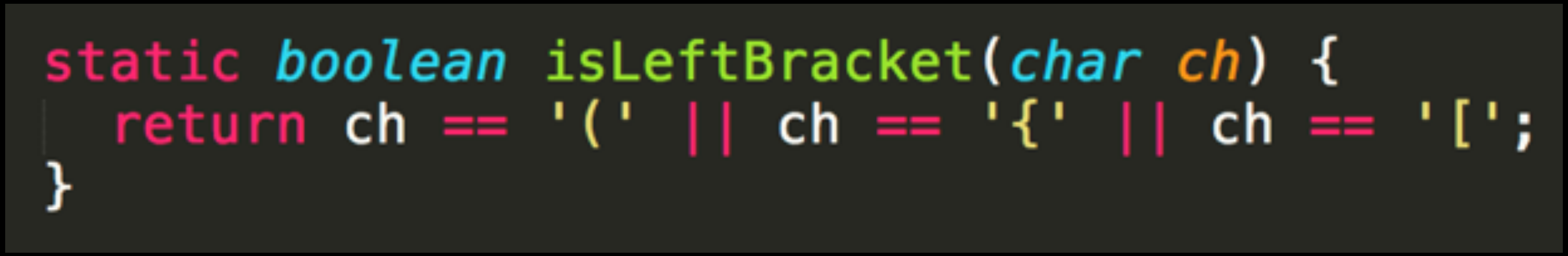
[()]))() → false

Stacks

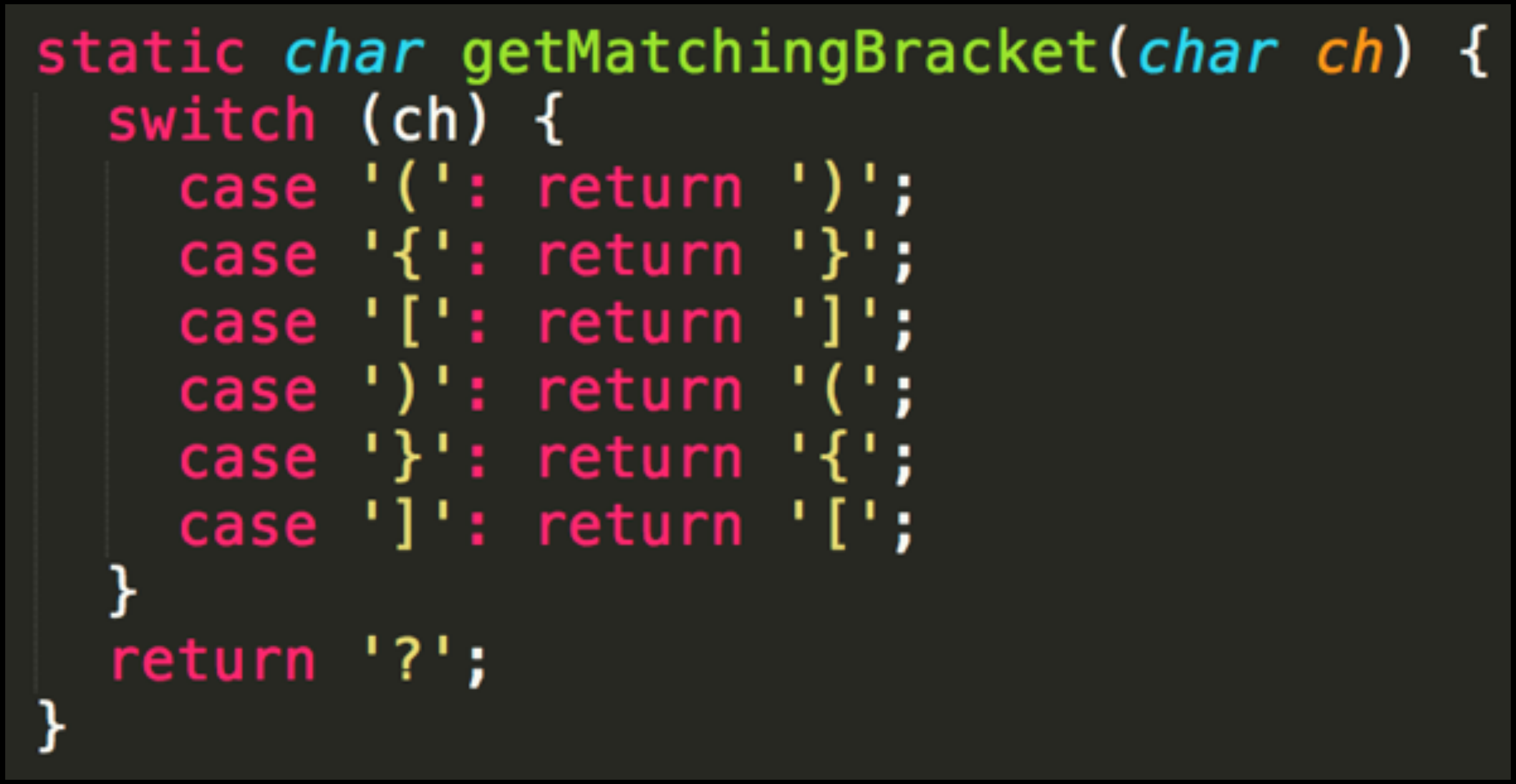


```
static boolean bracketsMatch(String brackets) {

  Stack<Character> stack = new Stack<>();

  // Iterate over each bracket
  for (char bracket : brackets.toCharArray()) {

    // Left bracket
    if (isLeftBracket(bracket))
      stack.add(bracket);

    // Right bracket
    else if (stack.isEmpty() || stack.pop() != getMatchingBracket(bracket))
      return false; // No matching bracket

  }

  // The brackets match if the stack is empty
  return stack.isEmpty();

}
```

Stacks



```java
static boolean isLeftBracket(char ch) {
  return ch == '(' || ch == '{' || ch == '[';
}
```



```java
static char getMatchingBracket(char ch) {
  switch (ch) {
    case '(': return ')';
    case '{': return '}';
    case '[': return ']';
    case ')': return '(';
    case '}': return '{';
    case ']': return '[';
  }
  return '?';
}
```

Stacks

Pros:

- Can easily process the most recently added elements first.

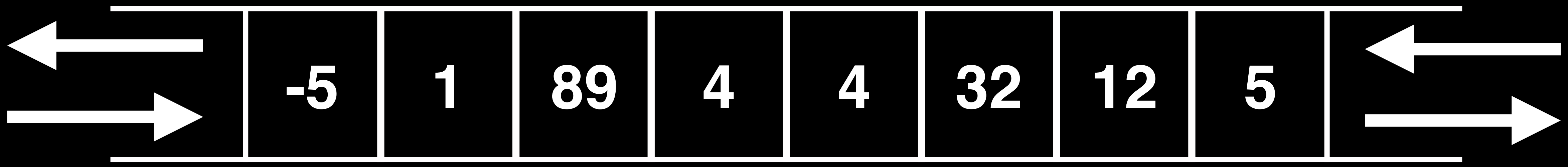
Cons:

- Can't index into or add an element at a specific position.

Deques

Deques

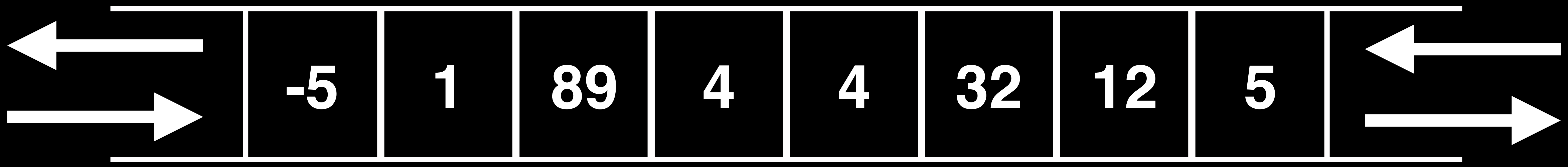
An container which can be both First-In-First-Out (FIFO) and Last-In-First-Out (LIFO).



-5	1	89	4	4	32	12	5

Deques

The most useful implementations of the *Deque* interface are *LinkedList* and *ArrayDeque*.



-5 1 89 4 4 32 12 5

Deques

Under the hood, *ArrayDeque* works much the same as an ArrayList. So this class is useful if you know many elements you might need at one given time (so that you can specify its initial capacity).

ArrayDeque also does not allow you to insert *null* elements.

Deques

LinkedList allows *null* elements, but is slower at adding and removing a lot of elements (since the space needs to be allocated and deallocated each time).

Deques

Pros:

- You get all of the benefits of both a Queue and Stack.

Cons:

- Still can’t index into or add an element at a specific position.

Sources

- https://docs.oracle.com