

# Data Structures and Algorithms

—

Lab 2  
Elementary Data Structures

# Agenda

- Lecture recap
- Implementing `Stack` using `ArrayList` (live coding)
- Java's `ArrayDeque` (code review)
- Implementing `Queue` using `LinkedList` (live coding)
- Exercise: parse and evaluate arithmetic expression
- CodeForces!

# Elementary Data Structures

- Which **Elementary Data Structures** do you know?

# Elementary Data Structures

- List
- Stack
- Queue
- Deque (Stack + Queue)

How do we store data?

# How do we store data?

- Array-like structure
- Linked structure
- Question: What are the pros and cons of each?

# How do we store data?

	<b>Array-like structure</b>	<b>Linked structure</b>
<b>Random access (by index)</b>		
<b>Add/grow</b>		

# How do we store data?

	Array-like structure	Linked structure
Random access (by index)	$O(1)$	$O(n)$
Add/grow	$O(n)$	$O(1)$



ADT vs DS?

# ADT vs DS?

- **Abstract Data Type**  
List, Stack, Queue, PriorityQueue
- **Data Structure**  
ArrayList, LinkedList, ...

# Stack Data Structure

- A stack is a linear data structure that follows the principle of Last In First Out (LIFO). This means the last element inserted inside the stack is removed first.
- You can think of the stack data structure as the pile of plates on top of another.
- The stack data structure follow the LIFO (Last In First Out) Principle.

**TOP = -1**



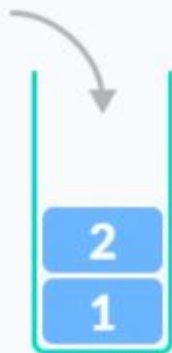
**empty  
stack**

**TOP = 0**  
**stack[0] = 1**



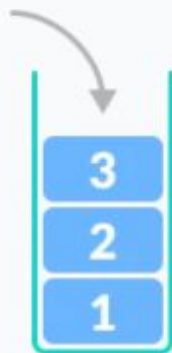
**push**

**TOP = 1**  
**stack[1] = 2**



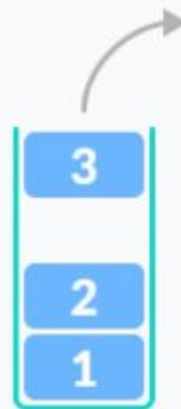
**push**

**TOP = 2**  
**stack[2] = 3**



**push**

**TOP = 1**  
**return stack[2]**



**pop**

Working of Stack Data Structure

# Applications of Stack Data Structure

- To reverse a word - Put all the letters in a stack and pop them out. Because of the LIFO order of stack, you will get the letters in reverse order.
- In compilers - Compilers use the stack to calculate the value of expressions like  $2 + 4 / 5 * (7 - 9)$  by converting the expression to prefix or postfix form.
- In browsers - The back button in a browser saves all the URLs you have visited previously in a stack. Each time you visit a new page, it is added on top of the stack. When you press the back button, the current URL is removed from the stack, and the previous URL is accessed.

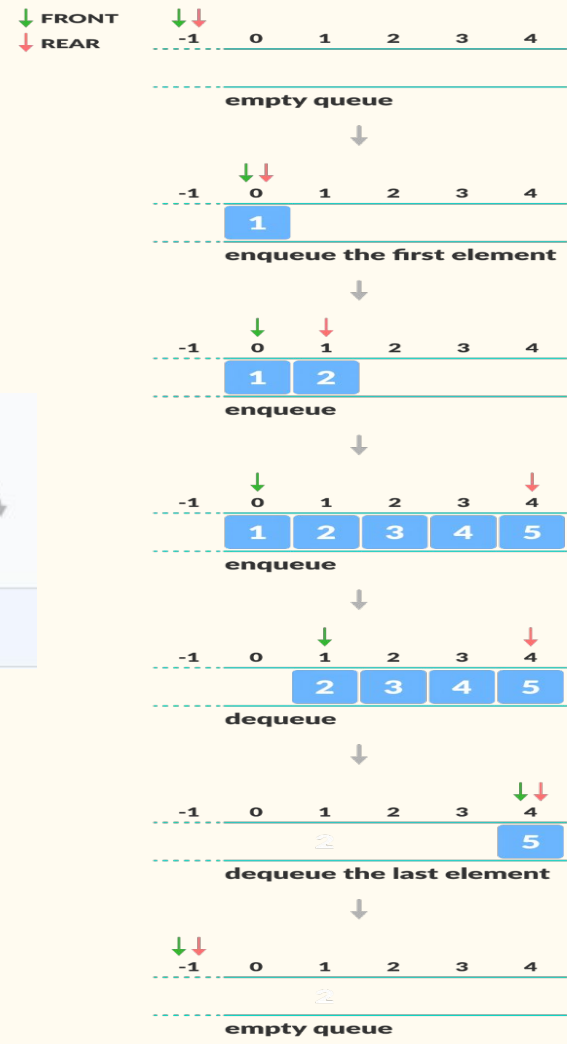
**Live coding:**    ArrayList (dynamic array) => Stack

- Stack live coding using ArrayList in Java

```
interface Stack<T> {  
    void push(T value);  
    T pop();  
    T peek();  
    int size();  
    boolean isEmpty();  
}
```

# Queue Data Structure

- A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.
- Queue follows the First In First Out (FIFO) rule - the item that goes in first is the item that comes out first.




FIFO Representation of Queue



## Live coding: Linked list $\Rightarrow$ Queue

- Queue live coding using Linked list
- Which type of linking will we choose? (SLL or DLL)

```
interface Queue<T> {  
    void offer(T value);  
    T poll();  
    T peek();  
     int size();  
    boolean isEmpty();  
}
```

# Applications of Queue Data Structure

- CPU scheduling, Disk Scheduling
- When data is transferred asynchronously between two processes. The queue is used for synchronization. For example: IO Buffers, pipes, file IO, etc
- Handling of interrupts in real-time systems.
- Call Center phone systems use Queues to hold people calling them in order.

# Deque Data Structure

- Deque or Double Ended Queue is a type of queue in which insertion and removal of elements can either be performed from the front or the rear.
- Thus, it does not follow FIFO rule (First In First Out).



Representation of Deque

# ArrayDeque.java

The full source code for `java.util.ArrayDeque`:

<http://fuseyism.com/classpath/doc/java/util/ArrayDeque-source.html>

```
Object[] elements;  
int head;  
int tail;
```

# ArrayDeque.java

```
/**
 * Inserts the specified element at the front of this deque.
 *
 * * @param e the element to add
 * * @throws NullPointerException if the specified element is null
 */
public void addFirst( @NotNull() E e) {
    if (e == null)
        throw new NullPointerException();
    elements[head = (head - 1) & (elements.length - 1)] = e;
    if (head == tail)
        doubleCapacity();
}
```

# ArrayDeque.java

```
/**
 * Inserts the specified element at the end of this deque.
 *
 * <p>This method is equivalent to {@link #add}.
 *
 * @param e the element to add
 * @throws NullPointerException if the specified element is null
 */
public void addLast( @NotNull() E e) {
    if (e == null)
        throw new NullPointerException();
    elements[tail] = e;
    if ( (tail = (tail + 1) & (elements.length - 1)) == head)
        doubleCapacity();
}
```

# ArrayDeque.java

```
public E pollFirst() {  
    int h = head;  
    /unchecked/  
    E result = (E) elements[h];  
    // Element is null if deque empty  
    if (result == null)  
        return null;  
    elements[h] = null;    // Must null out slot  
    head = (h + 1) & (elements.length - 1);  
    return result;  
}
```

# ArrayDeque.java

```
public E pollLast() {  
    int t = (tail - 1) & (elements.length - 1);  
    /unchecked/  
    E result = (E) elements[t];  
    if (result == null)  
        return null;  
    elements[t] = null;  
    tail = t;  
    return result;  
}
```



# Exercise: Shunting-yard algorithm

Shunting-yard algorithm is an algorithm that parses an expression in infix notation (e.g. arithmetic expression) into an internal representation. Here we will consider converting to Reverse Polish Notation (RPN), also known as postfix notation.

$$1 + (2 - 3) \times 4$$

# Exercise: Shunting-yard algorithm

Shunting-yard algorithm is an algorithm that parses an expression in infix notation (e.g. arithmetic expression) into an internal representation. Here we will consider converting to Reverse Polish Notation (RPN), also known as postfix notation.

**1 + (2 - 3) × 4**



**1 2 3 - 4 × +**

# Exercise: Shunting-yard algorithm (pseudocode)

1. While there are tokens to be read:
  2. Read a token
  3. If it's a number add it to queue
  4. If it's an operator
    5. While there's an operator on the top of the stack with greater precedence:
      6. Pop operators from the stack onto the output queue
    7. Push the current operator onto the stack
  8. If it's a left bracket push it onto the stack
  9. If it's a right bracket
    10. While there's not a left bracket at the top of the stack:
      11. Pop operators from the stack onto the output queue.
    12. Pop the left bracket from the stack and discard it
13. While there are operators on the stack, pop them to the queue

# Exercise: Shunting-yard algorithm

Shunting-yard algorithm is an algorithm that parses an expression in infix notation (e.g. arithmetic expression) into an internal representation. Here we will consider converting to Reverse Polish Notation (RPN), also known as postfix notation.

**1 + (2 - 3) × 4**



**1 2 3 - 4 × +**

Challenges:

Discuss the algorithm. What are the corner cases?

What data structures would you use? Why?

To build the algorithm, we will need:

1. 1 stack for operations
2. 1 queue of the output
3. 1 array (or other list) of tokens.

Token list

)
3
-
9
(
/
18
+
4

Operator stack



Output Queue



Input buffer	Stack	Output Queue
1 + (2 - 3) × 4		
+ (2 - 3) × 4		1
(2 - 3) × 4	+	1
2 - 3) × 4	+ (	1
- 3) × 4	+ (	1 2
3) × 4	+ ( -	1 2
) × 4	+ ( -	1 2 3
× 4	+	1 2 3 -
4	+ ×	1 2 3 -
	+ ×	1 2 3 - 4
		1 2 3 - 4 × +

## Exercise: RPN evaluation

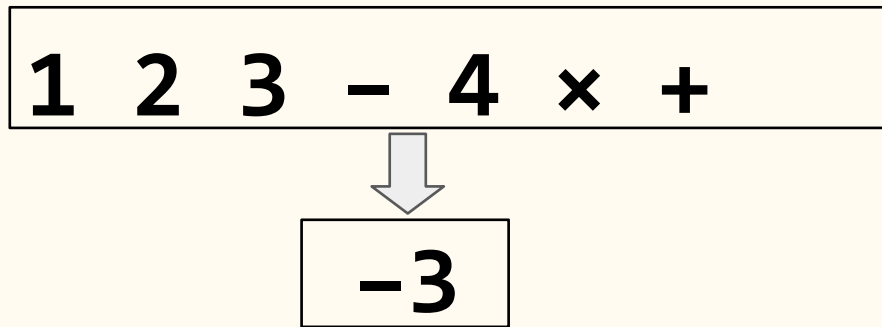
Given an expression in Reverse Polish Notation it is fairly straightforward to evaluate it.

<b>1</b>	<b>2</b>	<b>3</b>	<b>-</b>	<b>4</b>	<b>×</b>	<b>+</b>
----------	----------	----------	----------	----------	----------	----------

# -3

## Exercise: RPN evaluation

Given an expression in Reverse Polish Notation it is fairly straightforward to evaluate it.



Challenges:

Discuss the algorithm. What are the corner cases?

What data structures would you use? Why?



## Exercise: RPN evaluation (pseudocode)

```
S = new empty stack
while not eof
    t = read token
    if t is a binary operator
        y = pop(S)
        x = pop(S)
        push(S, t(x, y))
    else
        push(S, t)
print the contents of the stack S
```

Input Queue	Evaluation Stack
1 2 3 - 4 x +	
2 3 - 4 x +	1
3 - 4 x +	1 2
- 4 x +	1 2 3
4 x +	1 -1
x +	1 -1 4
+	1 -4
	-3

# Homework practice

- Implement both **Stack** and **Queue**
- Make sure they support operations from the lecture:
  - Addition/removal between nodes by index
  - Addition/removal after given value
- Submit Shunting-yard algorithm to CodeForces



<https://codeforces.com/group/M5kRwzPJlU/contest/365208>

**See you next week!**