WELCOME TO ADS

GUESS MY WORD

not independent

→ We need efficient ways of solving a given problem → Algorithms

&

→ We need efficient ways of representing data needed to solve the problem → Data Structures

BUT WHY?

To ensure the quality of software

ADS = tools to analyze

efficiency do the thing fast

correctness do the right thing

THE COURSE

LECTURES

THEORETICAL EXERCISES

IMPLEMENTATION EXERCISES

TODAY

BIG-OH NOTATION

a tool to analyze algorithms

LISTS, STACKS AND QUEUES

examples of data structures

BIG-OH NOTATION

ADS1, S2023

HOW FAST IS AN ALGORITHM?

```
public static int mySum( int n )
{
    int partialSum;
    partialSum = 0;

    for (int i = 0; i <= n; i++){
        partialSum +=i;
    }
    return partialSum;
}
```

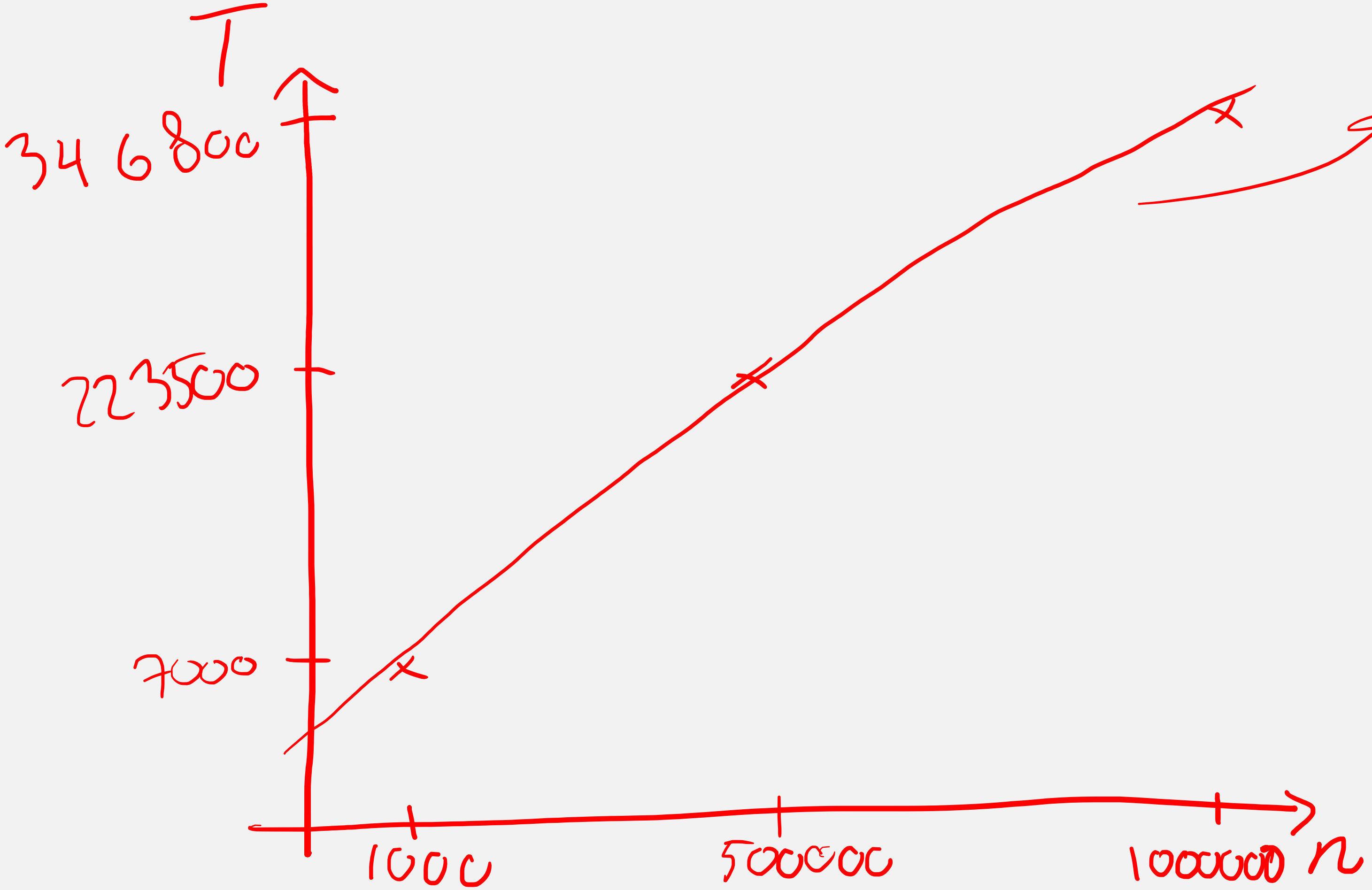
computes the sum

$1+2+3+4+\cdots+n$

HOW FAST IS AN ALGORITHM?

```
int [] numbers = {1000, 500000, 1000000};
for(int number: numbers) {
    long startTime = System.nanoTime();
    mySum(number);
    long endTime = System.nanoTime();
    System.out.println(endTime - startTime);
}
___________________________________________________
7000
223500
346800
```

HOW FAST IS AN ALGORITHM?



this function is

$T(n) = 0{,}34n + 22270$

time taken for input of size n

WHY WAS THIS A BAD PROCEDURE?

→ Depends on computer

→ Depends on programming language

→ Only checked a few input sizes

SO, INSTEAD ...

use PSEUDO-CODE

and BIG-OH

PSEUDO-CODE

```
public static int mySum( int n )
{
    int partialSum;
    partialSum = 0;

    for (int i = 0; i <= n; i++){
        partialSum +=i;
    }
    return partialSum;
}
```

```
function mySum(n):
  p=0
  for i from 0 to n:
     p=p+i
  return p
```

ANALYZING PSEUDO-CODE

```
function mySum(n):
    p = 0
    for i from 0 to n:
        p = p + i
    return p
```

1 assignment = 1 time unit

1 initialization
n+1 increments
n+2 tests
} 2n+4 time units

n+1 additions
n+1 assignments
} 2n+2 time units

1 return = 1 time units

assume addition, multiplication etc takes 1 time unit

$T(n) = 1 + 2n + 4 + 2n + 2 + 1 = 4n + 8$ time units

the constants may vary depending on how you count …

BIG-OH

Get rid of all non-essential info

$T(n) = \cancel{0.34}\, 4n + 2\cancel{2270}$

$T(n) = \cancel{4}n + \cancel{8}$

$T(n) = O(n)$ ← big-Oh

= linear time complexity

$T(n)$ is a linear function of n

BIG-OH

identify fastest growing term

$T(n)$	$O(n)$
$\underline{4n^3} + 100n^2 + 3n$	$O(n^3)$
$6n\log n + \sqrt{n}$	$O(n \log n)$
$7n! + 1000n$	$O(n!)$

FORMAL DEFINITION

there exists

$T(n) = O(f(n))$ means that

$\exists\, n_0, c$ such that

if $n > n_0$ then $c f(n) \geq T(n)$

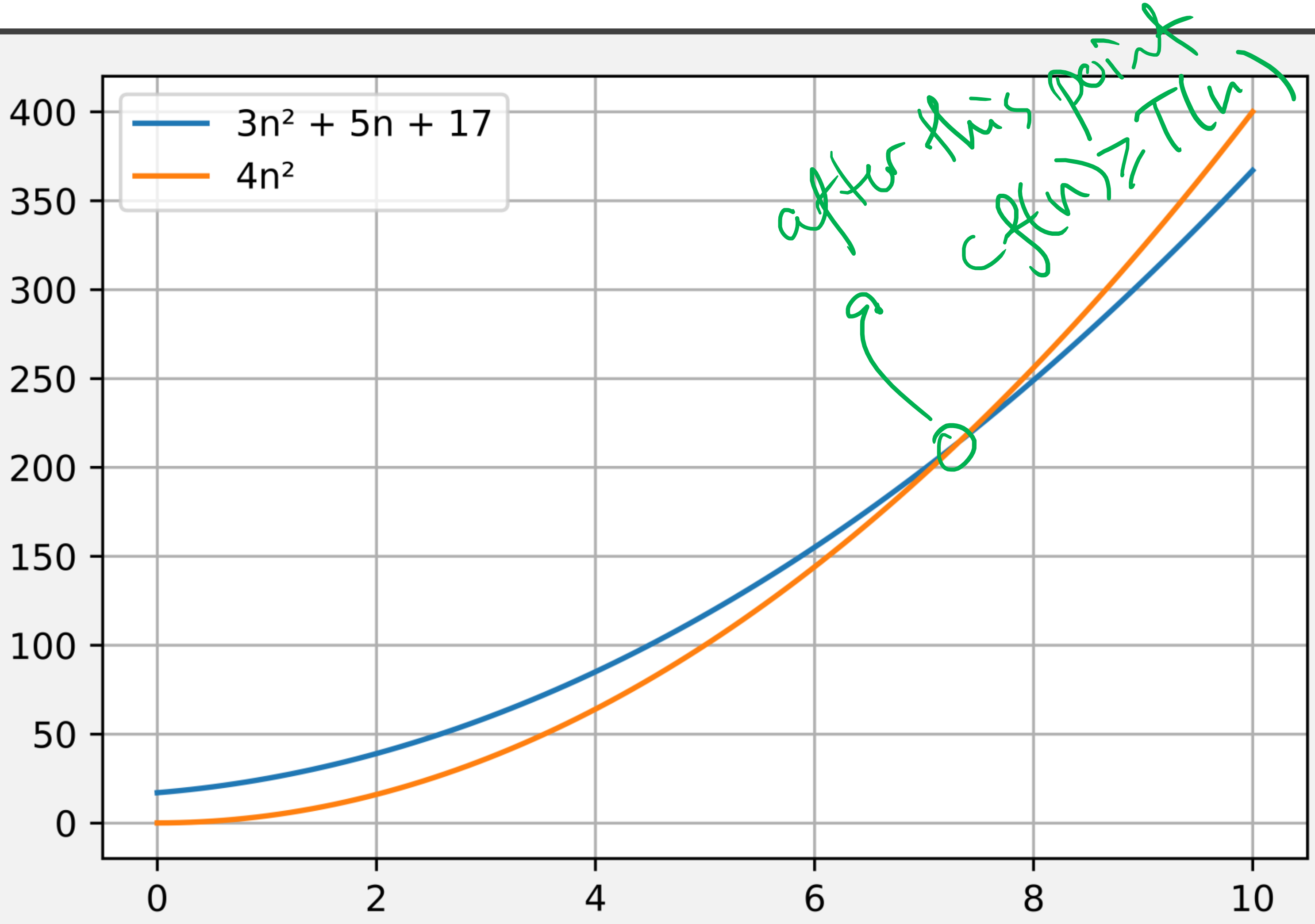
Example: $T(n) = 3n^2 + 5n + 17 = O(n^2)$

pick $c = 4$ then $4n^2 \geq 3n^2 + 5n + 17$

$\Rightarrow n^2 \geq 5n + 17$

$n_0 = 8$ will do: true for all $n > 8$

FORMAL DEFINITION



A CURIOUS RESULT OF THE FORMAL DEFINITION

$n^3 = O(n^2)$ false!

$n^2 = O(n^2)$ true!

$n = O(n^2)$ true!

$T(n) = O(f(n))$ means that $f(n)$ is asymptotically larger than $T(n)$

$O \Rightarrow$ worst-case.

A WHOLE FAMILY OF OH'S

$\rightarrow$ asymptotic notation

Really important

Big-Oh $\quad T(n) = O(f(n)) \quad \approx$ "$T(n) \le f(n)$"

Big-Theta $\quad T(n) = \Theta(f(n)) \quad \approx$ "$T(n) = f(n)$"

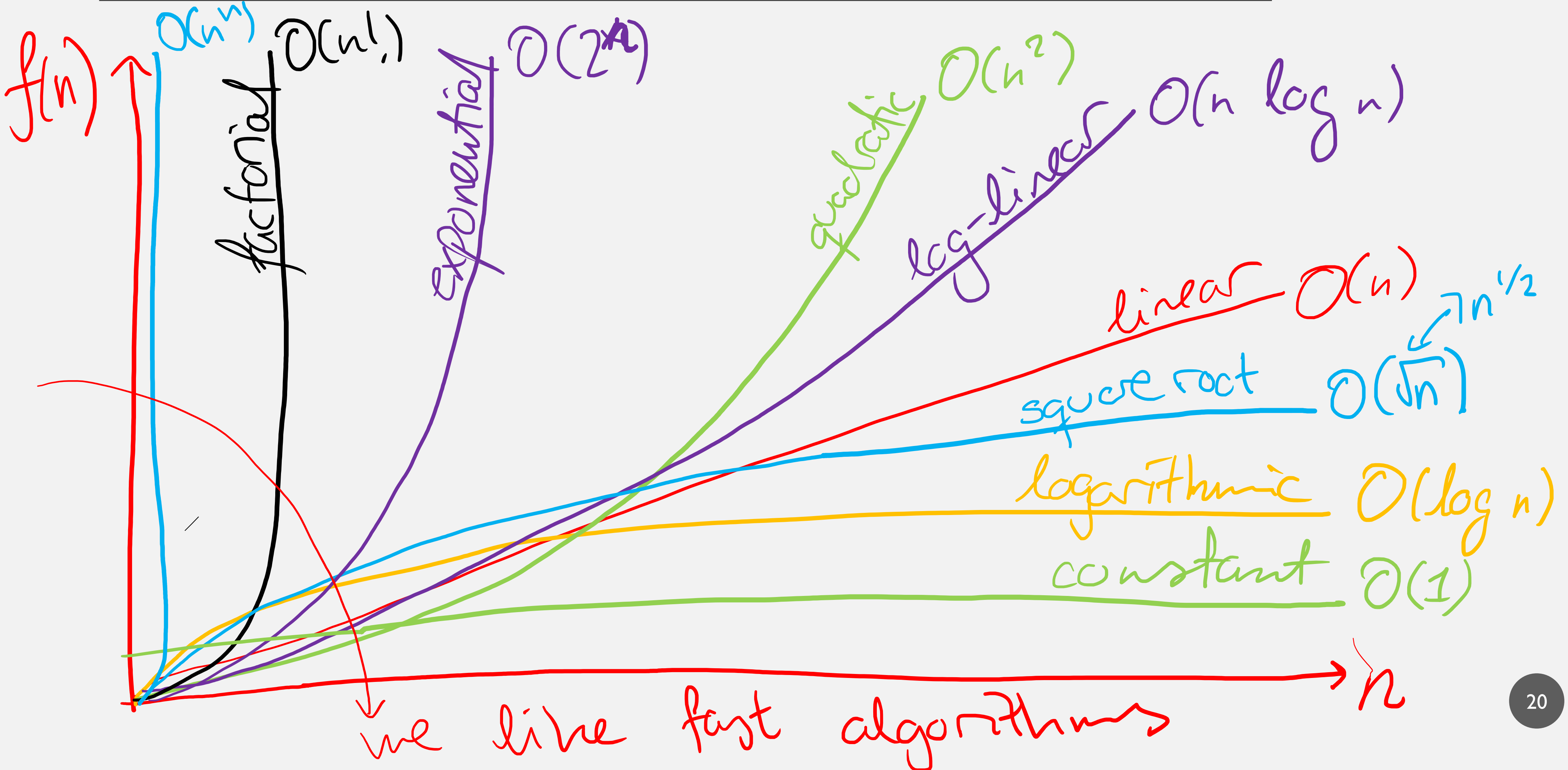
Rare

Big-Omega $\quad T(n) = \Omega(f(n)) \quad \approx$ "$T(n) \geq f(n)$"

Little-Oh $\quad T(n) = o(f(n)) \quad \approx$ "$T(n) < f(n)$"

Little-Omega $\quad T(n) = \omega(f(n)) \quad \approx$ "$T(n) > f(n)$"

COMMON GROWTH RATES



EXAMPLE 1: A SINGLE FOR LOOP

```
function find(array, value):
    for i from 0 to length(array):
        if array[i] == value:
            return i    ← found
    return -1    ← not found
```

case: not found

loop through n times before return

$n = O(n)$

case: found

on average, $n/2$ iterations

$\frac{n}{2} = O(n)$

linear search

$O(n)$

EXAMPLE II: NESTED LOOPS

```
function unique(array):
    for i from 0 to length(array) – 1:
        for j from 0 to length(array) – 1:
            if i ≠ j and array[i] == array[j]:
                return false
    return true
```

worst-case: unique

outer loop runs n times

inner loop runs n times per outer loop run

$\Rightarrow n \times n = n^2 = \underline{O(n^2)}$

EXAMPLE II: NESTED LOOPS

```
function unique(array):
    for i from 0 to length(array) – 1:
        for j from i+1 to length(array) – 1:
            if i ≠ j and array[i] == array[j]:
                return false
    return true
```

time if unique:

$$(n-1)+(n-2)+(n-3)+\dots$$
$$\dots+3+2+1$$
$$=\frac{n(n-1)}{2}=\frac{n^2}{2}-\frac{n}{2}=O(n^2)$$

still quadratic, but twice as fast

EXAMPLE III: LOGARITHMS

```
for (int i = 1; i < n; i *= 2) {
        do something
}
```

i doubled each iteration

terminates when $i \geq n$

$i = 1, 2, 4, 8, 16 \cdots\cdots, 2^k, \cdots\cdots n$

no. of iterations

terminates $2^k = n$

$k = \log_2 n$

$\Rightarrow$ $O(\log n)$

EXAMPLE IV: SQUARE ROOTS

```
for (int i = 1; i*i < n; i++) {
        do something
}
```

terminate when $i^2 \geq n$

$\Downarrow$

$i \geq \sqrt{n}$

$O(\sqrt{n})$

LISTS, STACKS AND QUEUES

ADS1, S2023

LISTS

head

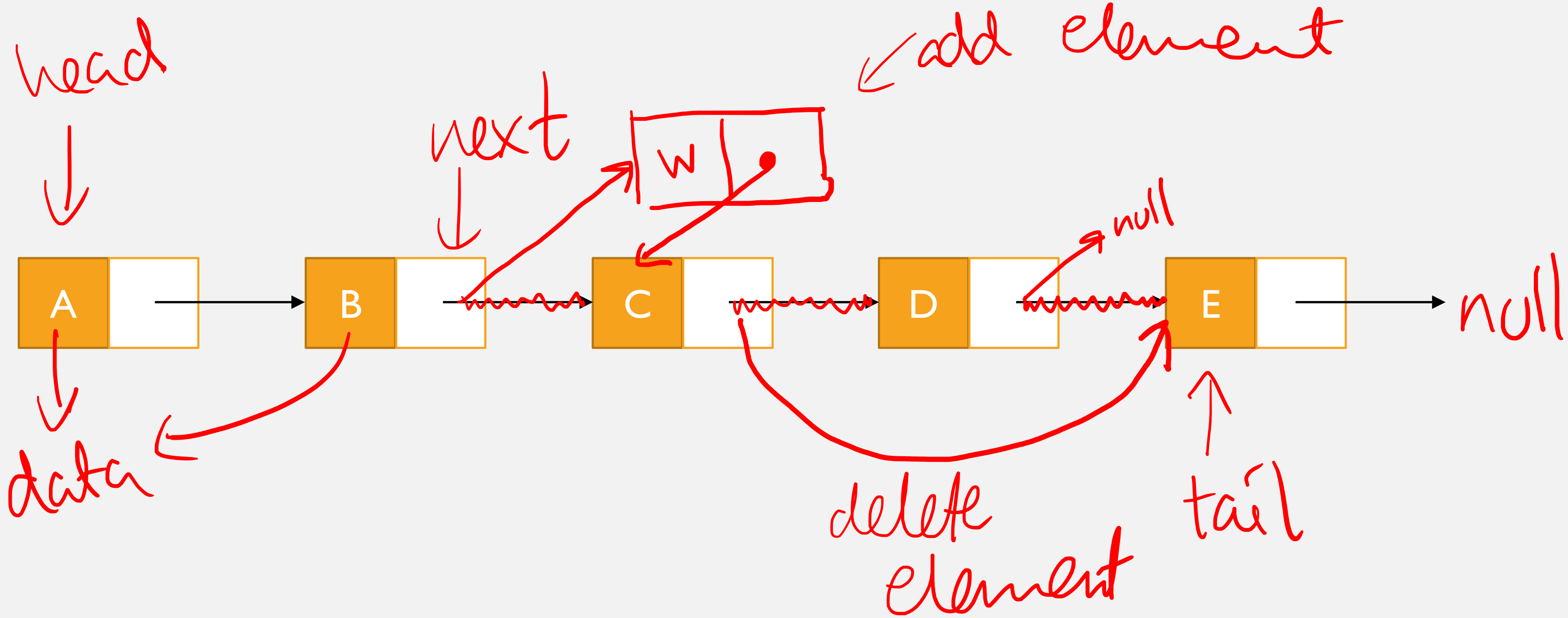
tail

a linear ordering of elements

OPERATIONS ON A LIST

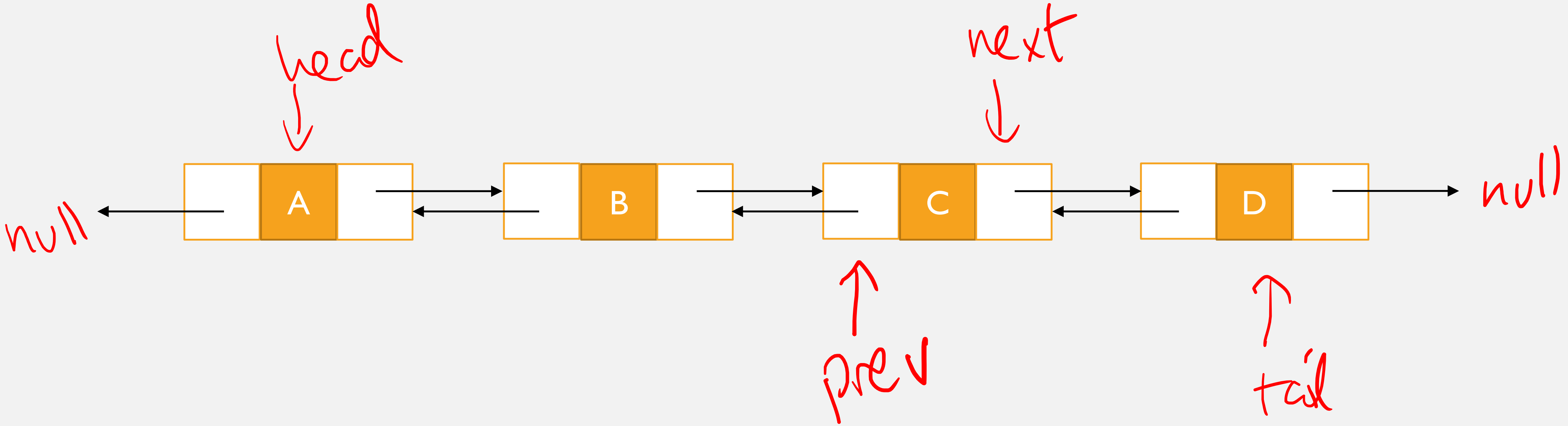
- a constructor for creating an empty list
- an operation for testing whether or not a list is empty
- an operation for getting the size of the list
- an operation for inserting an entity into a list
  - at a specific position
  - at the tail or the head
- an operation for deleting an entity from a list
  - at a specific position
  - at the tail or the head
- an operation for finding an entity in a list

LINKED LIST



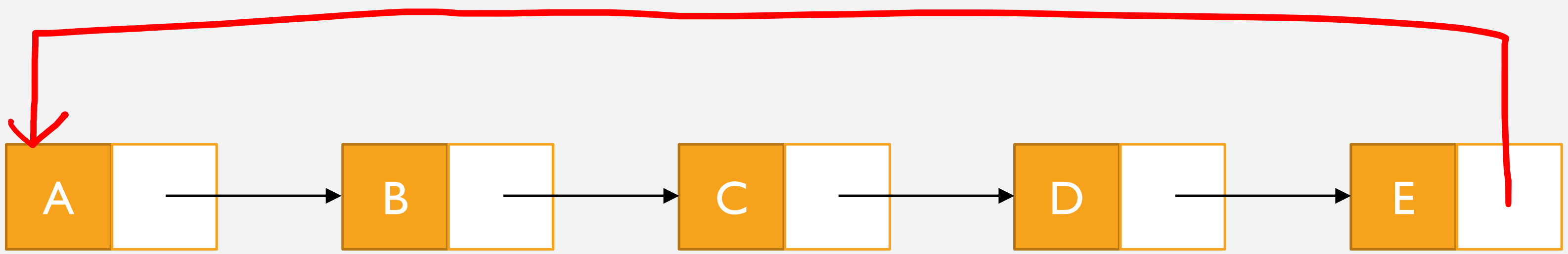
head
add element
next
w
null
A
B
C
D
E
null
data
delete element
tail

DOUBLY LINKED LIST

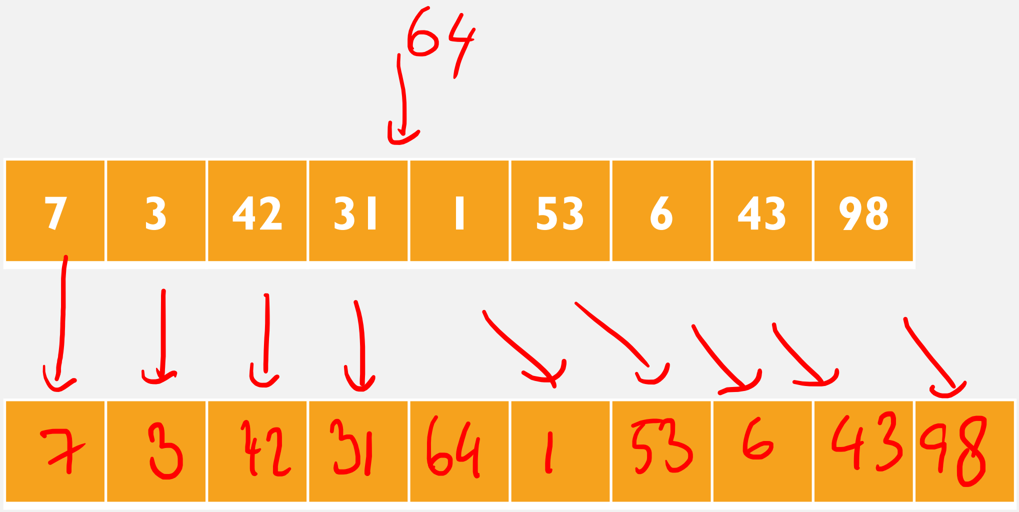


makes some algorithms more efficient

CIRCULAR LIST



ARRAYLIST



LINKED LISTS VS ARRAYLISTS

	Linked Lists	ArrayLists
Access	$O(n)$	$O(1)$
Insert/Remove	$O(1)$	$O(n)$
Determine size	$O(n)$	$O(1)$

STACKS

top

last in
first out

OPERATIONS ON A STACK

- a constructor for creating an empty stack
- an operation for testing whether or not a stack is empty
- an operation for inspecting the top element (peek)
- an operation for retrieving the top element (pop)
- an operation for adding a new element (push)

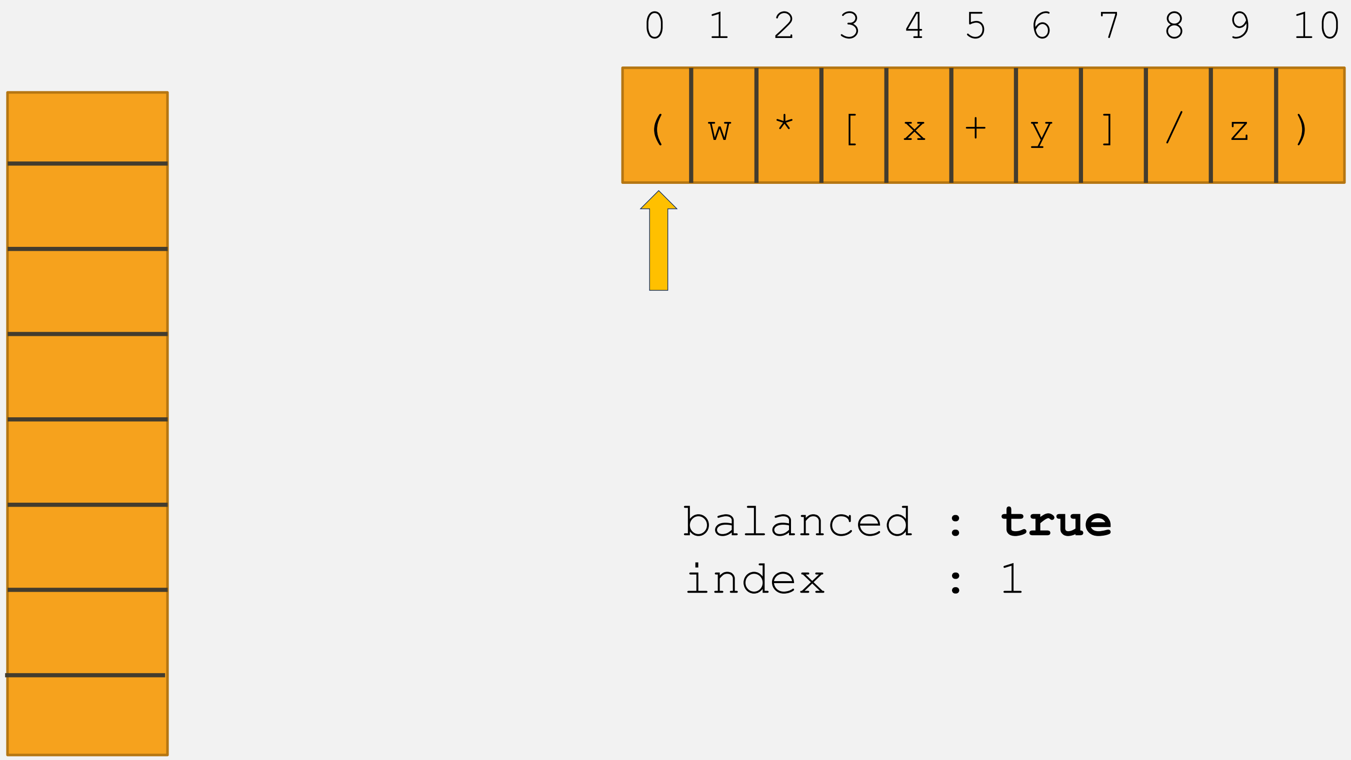
https://yongdanielliang.github.io/animation/web/Stack.html

STACK I: BALANCED PARENTHESES

```
( a + b * ( c / ( d – e ) ) ) + ( d / e )
```

STACK I: BALANCED PARENTHESES

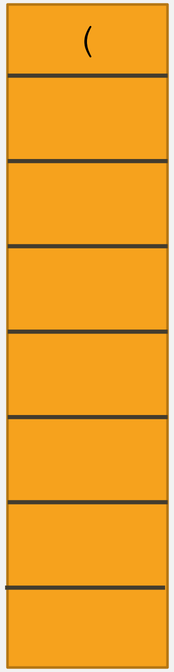
Expression: `(w * [x + y] / z)`




```
balanced : true
index    : 1
```

STACK I: BALANCED PARENTHESES

Expression: `(w * [x + y] / z)`

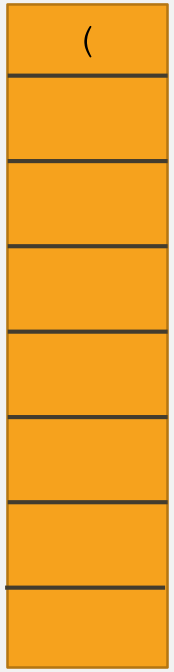


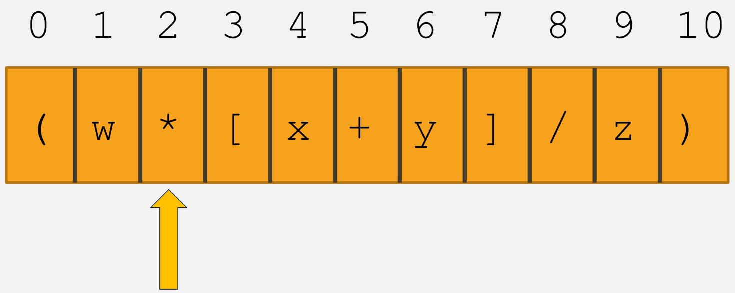
0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

```
balanced : true
index    : 1
```

STACK I: BALANCED PARENTHESES

Expression: (w * [x + y] / z)

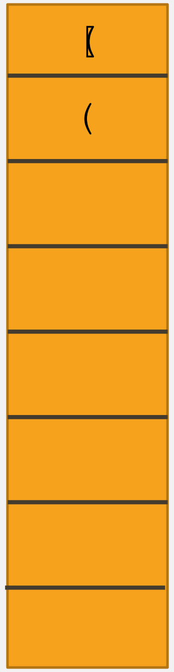


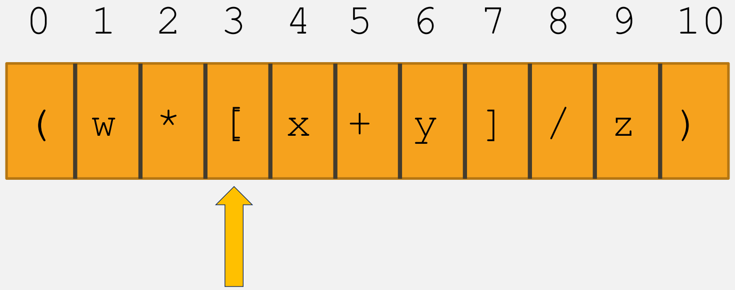


balanced : **true**
index : 2

STACK I: BALANCED PARENTHESES

Expression: (w * [x + y] / z)

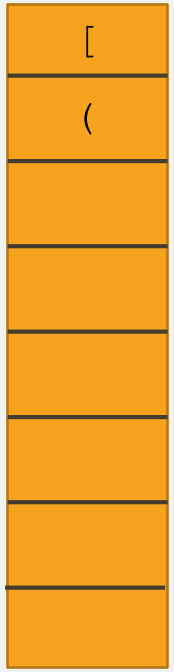


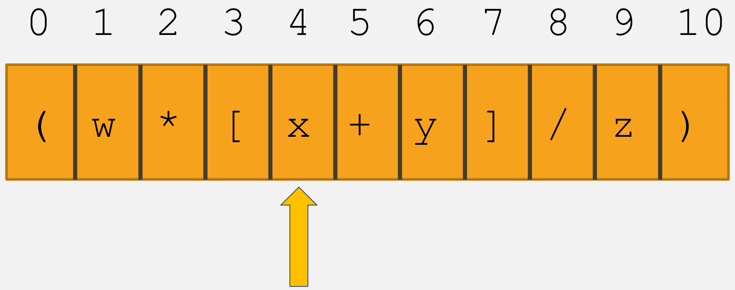



```
balanced : true
index    : 3
```

STACK I: BALANCED PARENTHESES

Expression: `(w * [x + y] / z)`

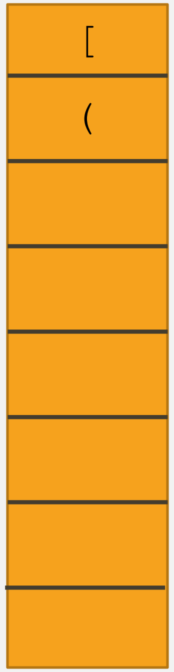


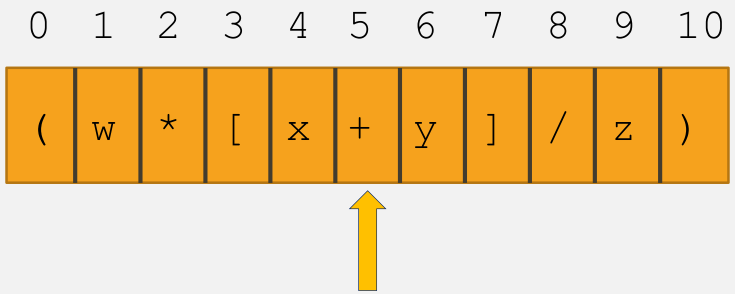



```
balanced : true
index    : 4
```

STACK I: BALANCED PARENTHESES

Expression: `(w * [x + y] / z)`

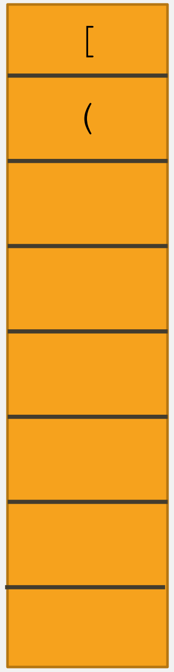


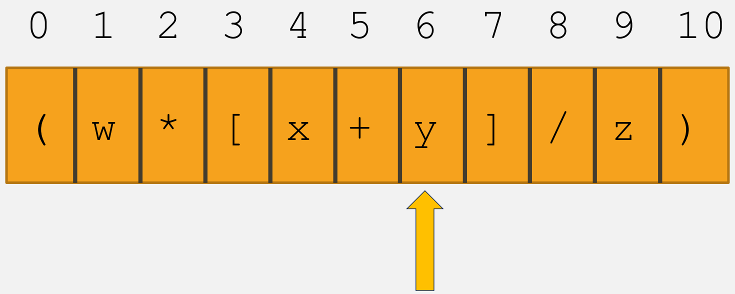



```
balanced : true
index    : 5
```

STACK I: BALANCED PARENTHESES

Expression: `(w * [x + y] / z)`

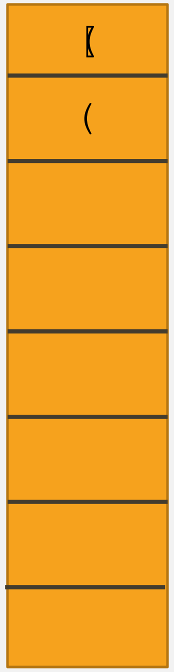





```
balanced : true
index    : 6
```

STACK I: BALANCED PARENTHESES

Expression: `(w * [x + y] / z)`



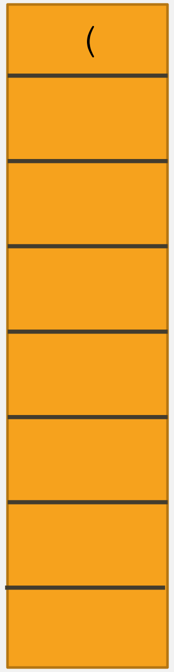
0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

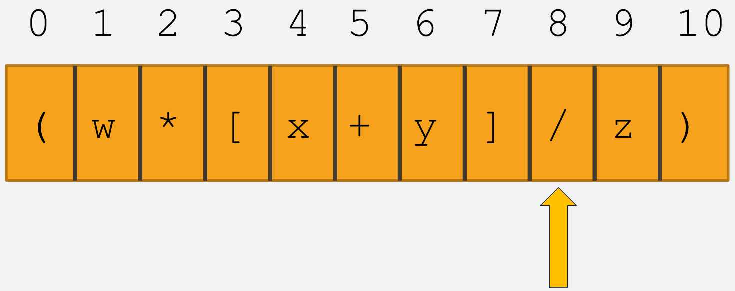
Matches!
`Balanced` still `true`

```
balanced : true
index    : 7
```

STACK I: BALANCED PARENTHESES

Expression: `(w * [x + y] / z)`

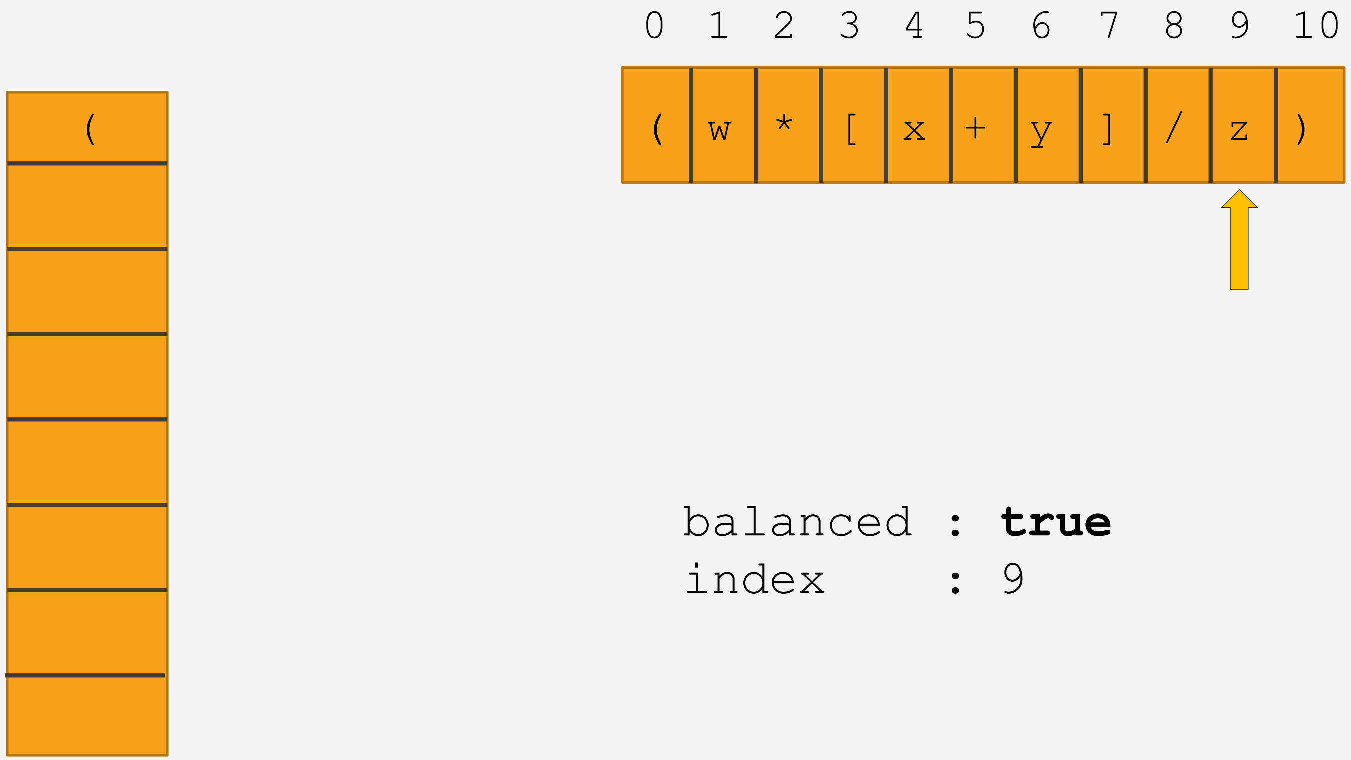





```
balanced : true
index    : 8
```

STACK I: BALANCED PARENTHESES

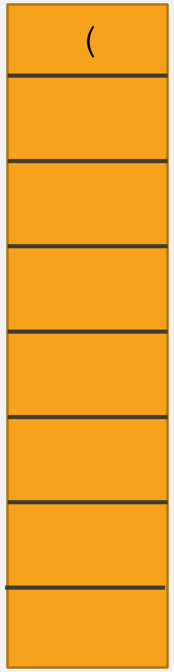
Expression: `(w * [x + y] / z)`




```
balanced : true
index    : 9
```

STACK I: BALANCED PARENTHESES

Expression: `(w * [x + y] / z)`



0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

Matches!
`Balanced` still `true`

```
balanced : true
index    : 10
```

STACK II: RPN CALCULATOR

reverse polish notation = postfix notation

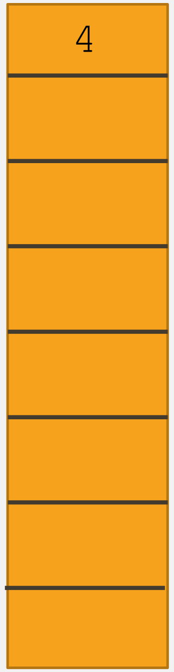
4, 7, −, 3, ×

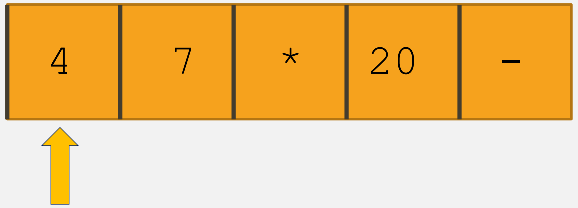
$(4-7)\times 3$

3, 4, 7, ×, 2, /, +

$(4\cdot 7)/2+3$

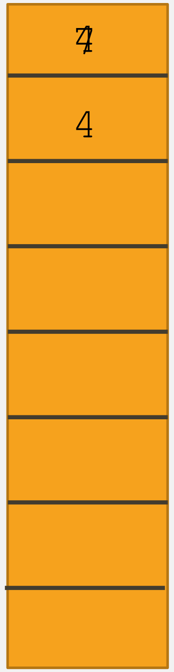
STACK II: RPN CALCULATOR

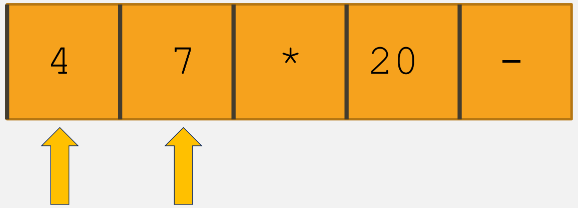





```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

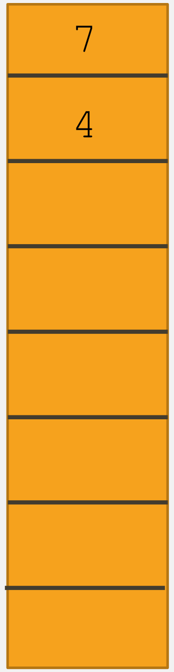
STACK II: RPN CALCULATOR



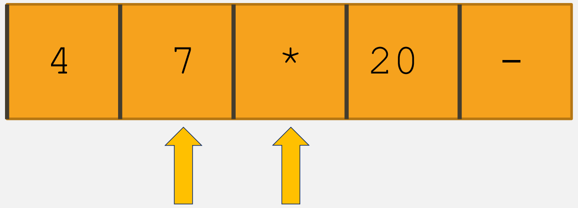



```
1.  create an empty stack of integers
2.  while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the token on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand off the stack
9.       evaluate the operation
10.      push the result onto the stack
11. pop the stack and return the result
```

STACK II: RPN CALCULATOR

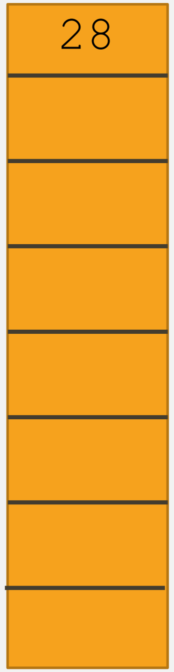


4 * 7

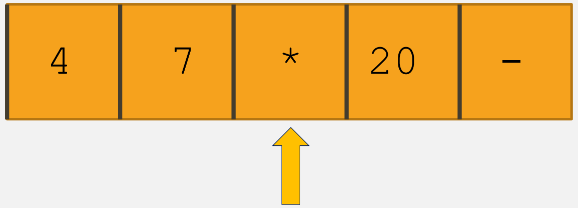



```
1.  create an empty stack of integers
2.  while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the token on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand off the stack
9.       evaluate the operation
10.      push the result onto the stack
11. pop the stack and return the result
```

STACK II: RPN CALCULATOR

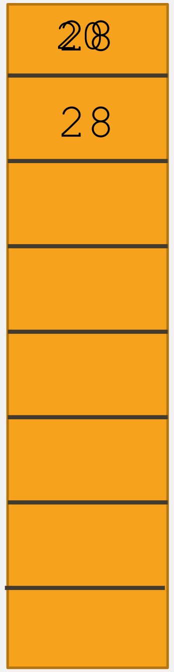


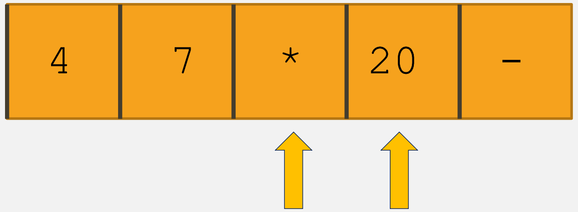
28




```
1.  create an empty stack of integers
2.  while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the token on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand off the stack
9.       evaluate the operation
10.      push the result onto the stack
11. pop the stack and return the result
```

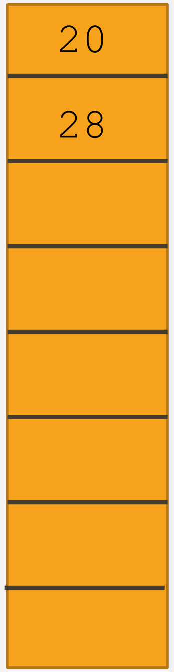
STACK II: RPN CALCULATOR



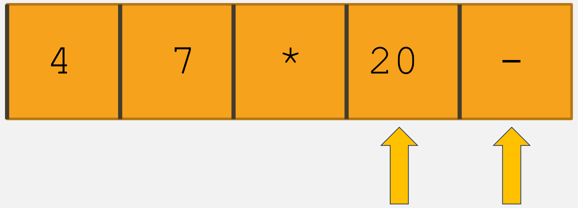



```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

STACK II: RPN CALCULATOR

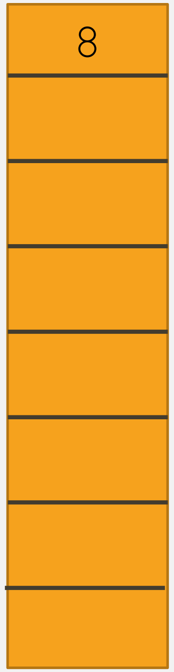


28 - 20

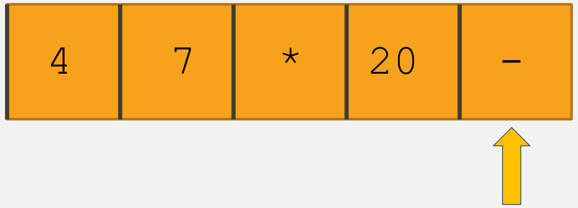



```
1.  create an empty stack of integers
2.  while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the token on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand off the stack
9.       evaluate the operation
10.      push the result onto the stack
11. pop the stack and return the result
```

STACK II: RPN CALCULATOR

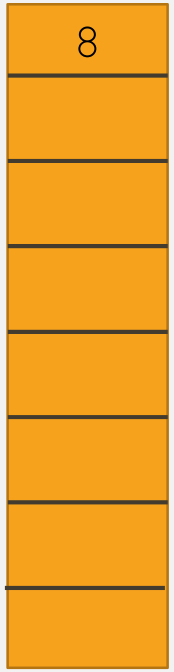


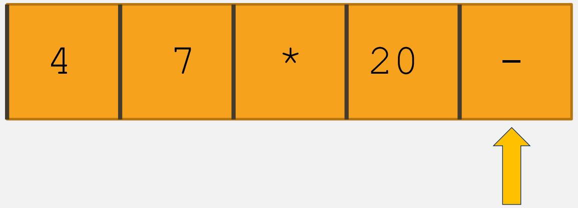
8




```
1.  create an empty stack of integers
2.  while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the token on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand off the stack
9.       evaluate the operation
10.      push the result onto the stack
11. pop the stack and return the result
```

STACK II: RPN CALCULATOR






```
1. create an empty stack of integers
2. while there are more tokens
3.   get the next token
4.   if the first character of the token is a digit
5.      push the token on the stack
6.   else if the token is an operator
7.      pop the right operand off the stack
8.      pop the left operand off the stack
9.      evaluate the operation
10.     push the result onto the stack
11. pop the stack and return the result
```

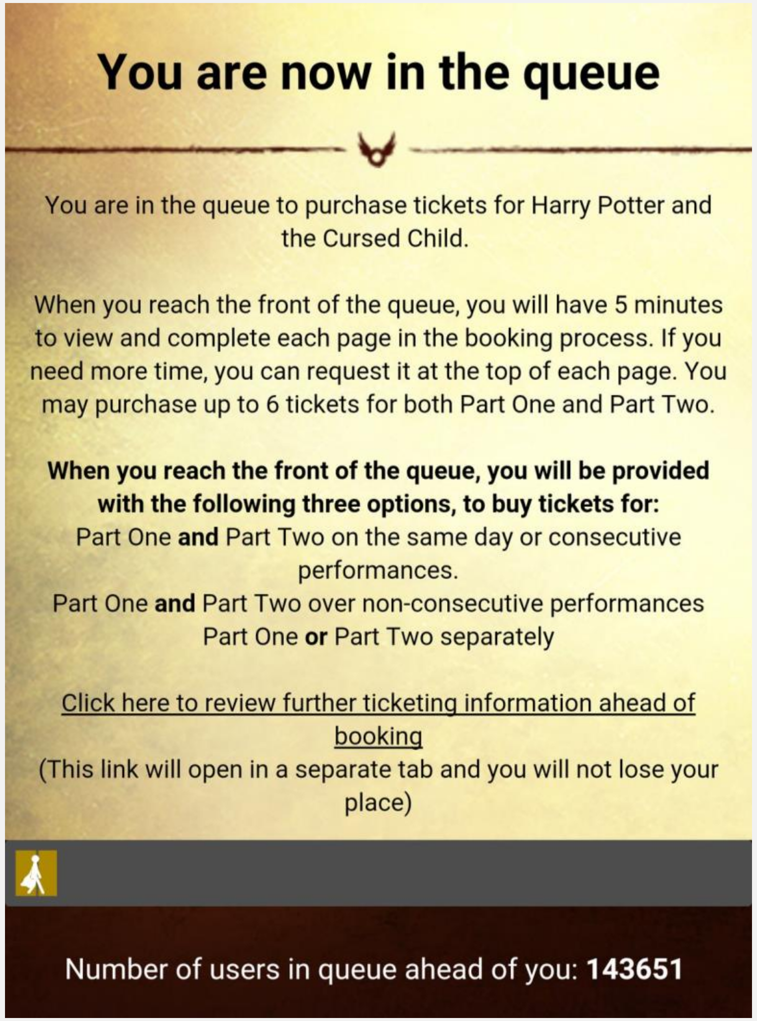
QUEUES

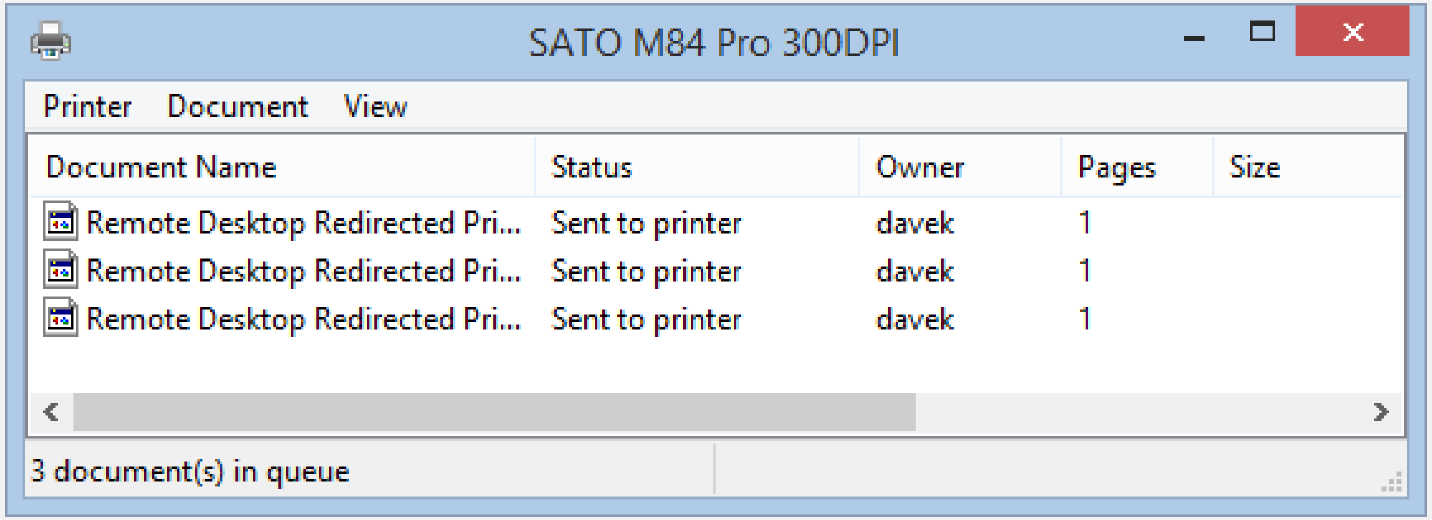
~~ta~~ first in, first out

OPERATIONS ON A QUEUE

- a constructor for creating an empty queue
- an operation for testing whether or not a queue is empty
- an operation for inspecting the front of the queue (peek)
- remove an entity from (the front of) the queue (dequeue)
- add an entity to (the back of) the queue (enqueue)

EXAMPLES OF QUEUES





At the beginning of the lecture, I placed all my PowerPoint slides in a queue, Q.

Now watch what happens when I perform the operation Q.dequeue().