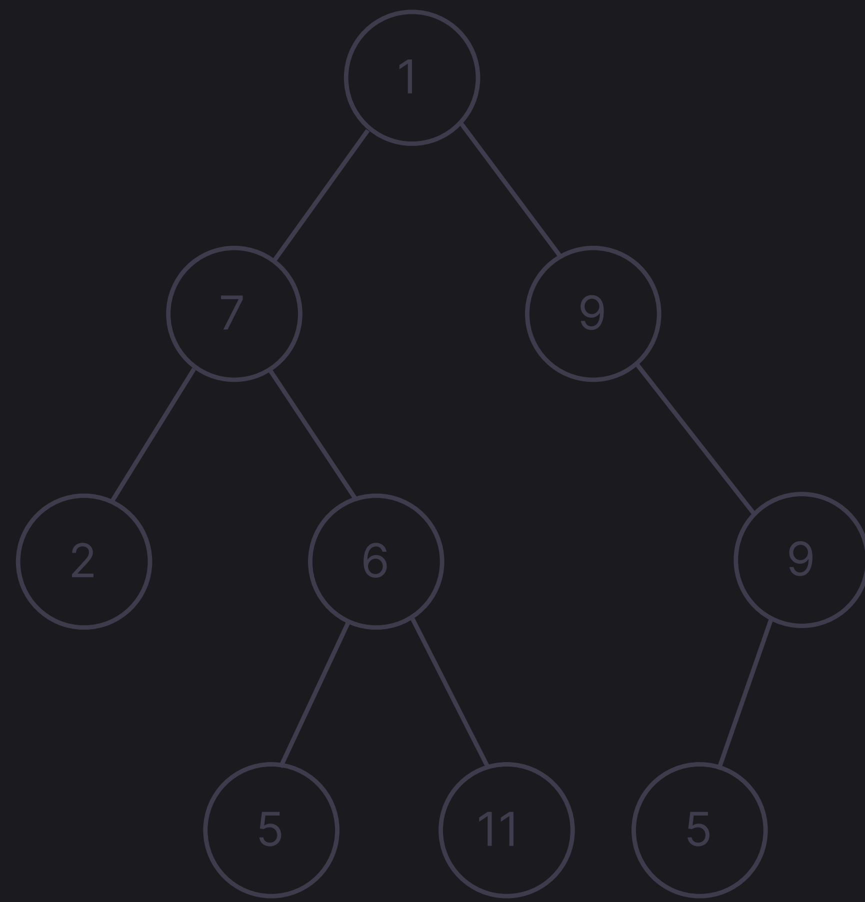
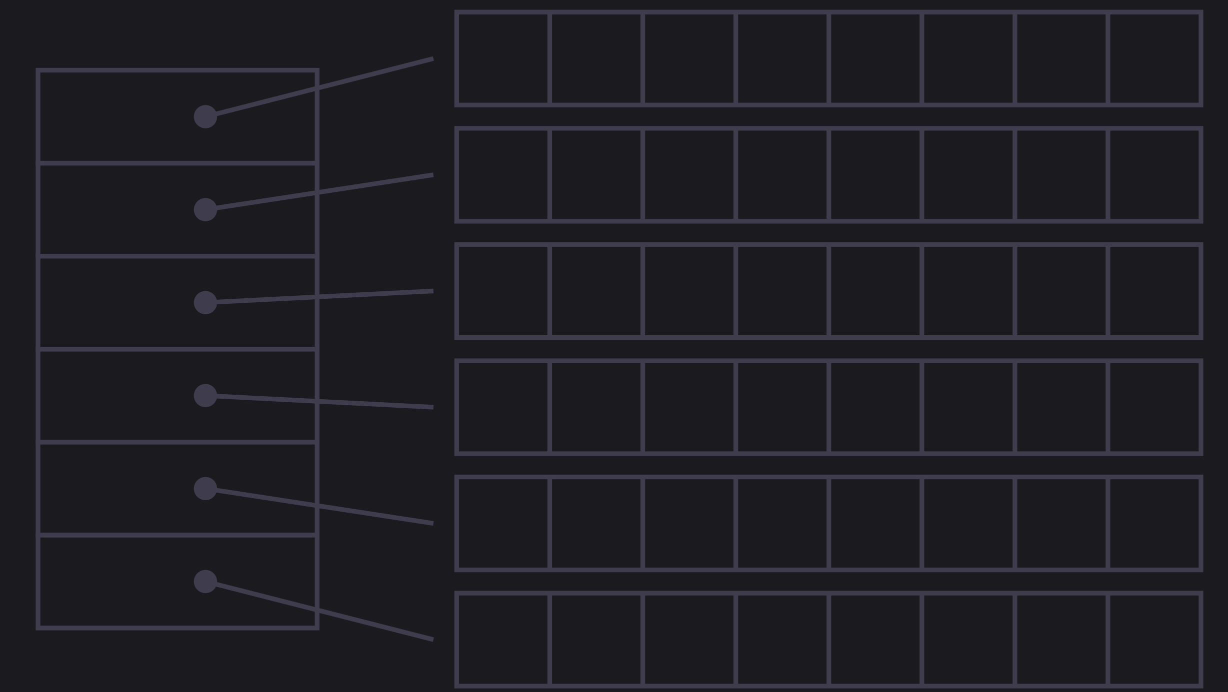
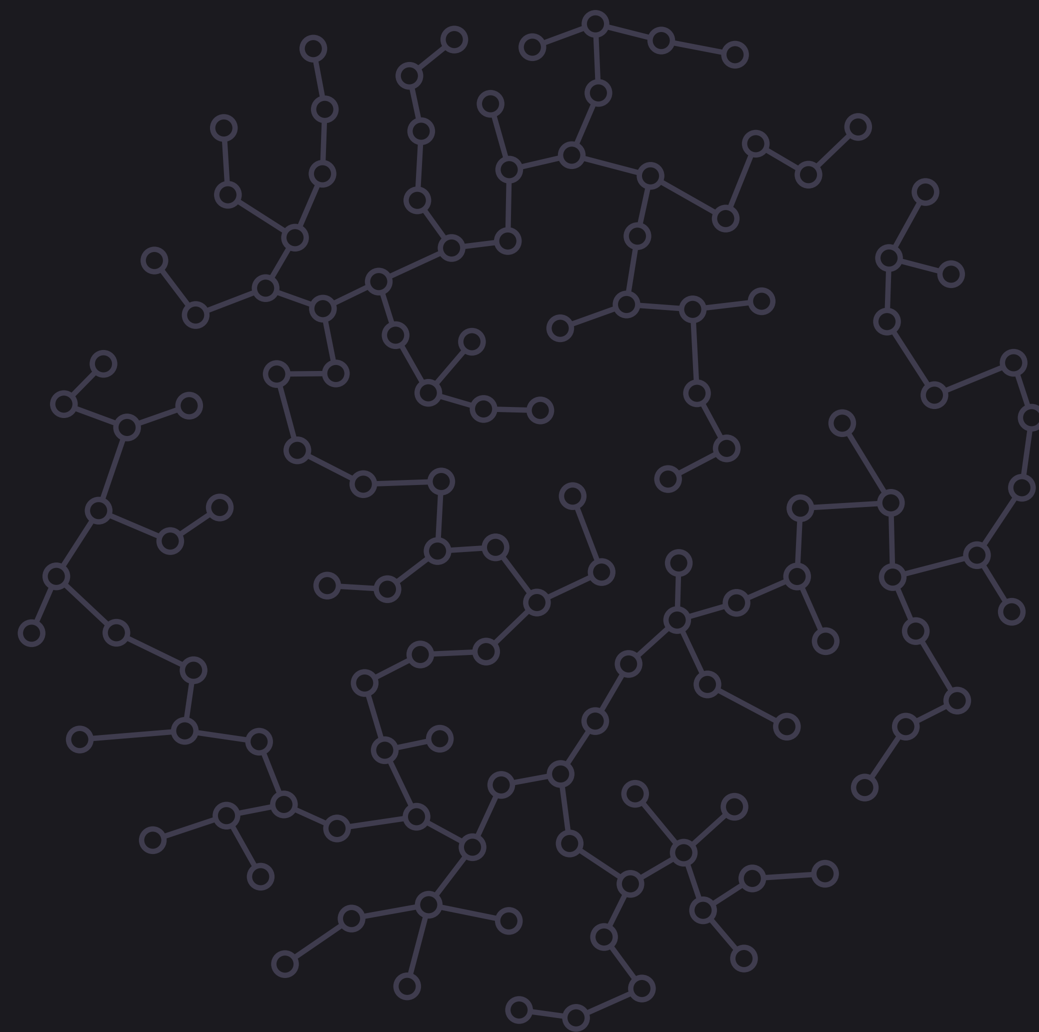


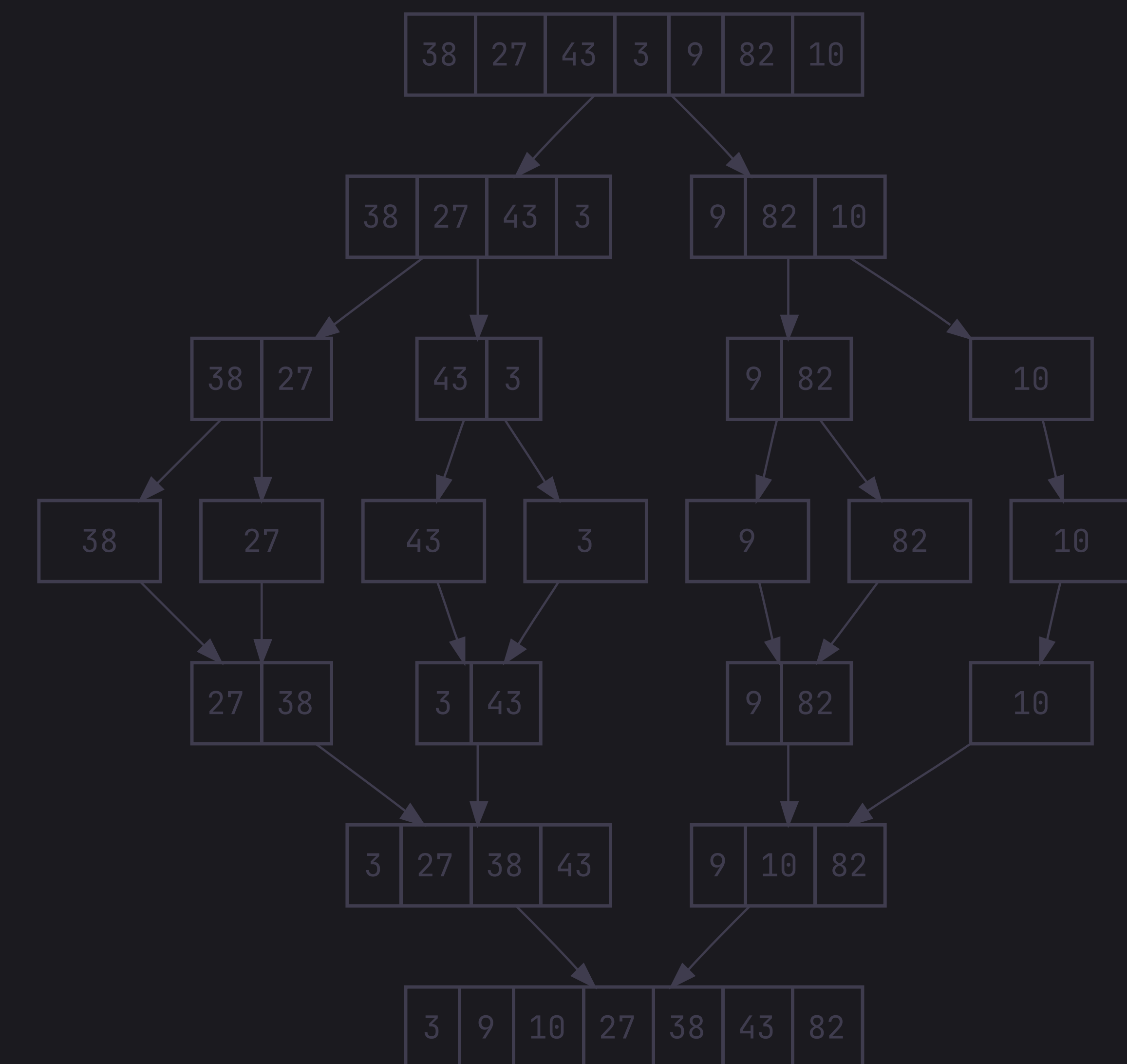



# data structures & algorithms

## Tutorial 6

(Week 7)





# Burning questions from last week?

# This week's lab

Stacks

This week we are exploring two very useful data structures and their applications

- Hash Maps
- Stacks
- The Two Sum Problem
- Stack Calculator

# Hash Maps

Where “it’s an ***organised*** mess. I know ***exactly*** where everything is” is actually true

```
std::unordered_map<std::string, int> map {};
```

# Hash Maps

```
std::unordered_map<std::string, int> favFoods {};
```

| | |
|---|---|
| “taco” | 7 |
| “pizza” | 12 |
| “salad” | 1 |
| “noodles” | 8 |

# Hash Maps

```
std::unordered_map<std::string, int> favFoods {};
```

| “taco” | 7 |
| --- | --- |
| “pizza” | 12 |
| “salad” | 1 |
| “noodles” | 8 |

```
favFoods["pizza"] = 15;
std::cout << favFoods["pizza"] << "\n";
```

# Hash Maps

You can also insert new elements like so

| | |
|---|---|
| “taco” | 7 |
| “pizza” | 12 |
| “salad” | 1 |
| “burger” | 4 |
| “noodles” | 8 |

```
favFoods["burger"] = 4;
```

std::unordered_map doesn't give you any guarantee on element order!

# Hash Maps

Or delete elements like so

| “taco” | 7 |
| --- | --- |
| “salad” | 1 |
| “burger” | 4 |
| “noodles” | 8 |

```
favFoods.erase("pizza");
```

# Hash Maps

Or delete elements like so

| “taco” | 7 |
| --- | --- |
| “salad” | 1 |
| “burger” | 4 |
| “noodles” | 8 |

```
favFoods.contains("taco") == true;
```

# Hash Maps

Or delete elements like so

| “taco” | 7 |
| --- | --- |
| “salad” | 1 |
| “burger” | 4 |
| “noodles” | 8 |

```
favFoods.contains("apples") == false;
```

# Two Sum

You are given a **vector** of **integers** called **vec** and another **integer** called **target**.

You are promised that there are two indices **i** and **j** such that **i ≠ j** and **target == vec[i] + vec[j]**

# Two Sum

You are given a vector of integers called vec and another integer called target. You are promised that there are two indices i and j such that i ≠ j and target == vec[i] + vec[j]

target = 13

vec =

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

# Two Sum

You are given a vector of integers called vec and another integer called target. You are promised that there are two indices $i$ and $j$ such that $i \neq j$ and $target == vec[i] + vec[j]$

target = 13

vec =

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

# Two Sum

You are given a vector of integers called vec and another integer called target. You are promised that there are two indices i and j such that $i \neq j$ and target == vec[i] + vec[j]

target = 13

vec =

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

output = { 2, 5 }

# Two Sum

You are given a vector of integers called vec and another integer called target. You are promised that there are two indices $i$ and $j$ such that $i \neq j$ and target == vec[i] + vec[j]

target = 13

vec =

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

output = { 2, 5 }

# Two Sum

## Any suggestions for a brute force algorithm? 🤔

You are given a vector of integers called vec and another integer called target. You are promised that there are two indices i and j such that i ≠ j and target == vec[i] + vec[j]

# Two Sum: Key Idea

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Every **value** in the vector has a **“matching partner”** which may or may not also be in the vector

**matchingPartner = target - value**

# Two Sum: Key Idea

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

```
target = 13
matchingPartner = target - value
```

# Two Sum: Key Idea

10

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

```
target = 13
matchingPartner = target - value
```

# Two Sum: Key Idea

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

7

```
target = 13
matchingPartner = target - value
```

# Two Sum: Key Idea

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

(Above index 2: 9; above index 5: 4)

```
target = 13
matchingPartner = target - value
```

# Two Sum: Key Idea

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

So if we could quickly check if the matchingPartner is in the vector, we could solve Two Sum in **linear time**

# Two Sum: Key Idea

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

With a **vector** if i know the **index**
I can quickly get the **value**

# Two Sum: Key Idea

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

With a **vector** if i know the **index**
I can quickly get the **value**

# Two Sum: Key Idea

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

But if i have a **value** i need to look through the array to find the **index**

(this is slow)

# Two Sum: Key Idea

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

How could we maybe speed up this lookup with a hash map?

# Two Sum: Key Idea

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

| value | indices |
|---|---|
| 3 | { 0 } |
| 1 | { 1, 3 } |
| 4 | { 2 } |
| 5 | { 4, 8 } |
| 9 | { 5 } |
| 2 | { 6 } |
| 6 | { 7 } |

We can create a reverse lookup table

# Two Sum: Key Idea

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

| value | indices |
|---|---|
| 3 | { 0 } |
| 1 | { 1, 3 } |
| 4 | { 2 } |
| 5 | { 4, 8 } |
| 9 | { 5 } |
| 2 | { 6 } |
| 6 | { 7 } |

We can create a reverse lookup table

# Two Sum: Key Idea

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

| value | indices |
|---|---|
| 3 | { 0 } |
| 1 | { 1, 3 } |
| 4 | { 2 } |
| 5 | { 4, 8 } |
| 9 | { 5 } |
| 2 | { 6 } |
| 6 | { 7 } |

We can create a reverse lookup table

# Two Sum: Key Idea

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

| value | indices |
|---|---|
| 3 | 0 |
| 1 | 1 |
| 4 | 2 |
| 5 | 4 |
| 9 | 5 |
| 2 | 6 |
| 6 | 7 |

But for this problem we only need one index, so we can keep track of the first index we see a particular value

# Two Sum: Key Idea

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

| value | indices |
|---|---|
| 3 | 0 |
| 1 | 3 |
| 4 | 2 |
| 5 | 8 |
| 9 | 5 |
| 2 | 6 |
| 6 | 7 |

Or alternatively we can keep track of the last time we see a particular value

# Two Sum

↓

vec = | 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

target = 13

matchingPartner = target - 3 = 10

Hashmap

| value | index |
|---|---|
| 3 | 0 |

does not contain 10

If the matchingPartner is **not** in the hashmap, then we store the **current number** we have just seen and the **index** we saw it at in the hashmap

# Two Sum

↓

vec = | 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|

0 1 2 3 4 5 6 7 8

target = 13

matchingPartner = target - 3 = 10

Hashmap

| value | index |
|---|---|
| 3 | 0 |

does not contain 10

If the matchingPartner is **not** in the hashmap, then we store the **current number** we have just seen and the **index** we saw it at in the hashmap

# Two Sum

Hashmap

↓

vec = 

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

| value | index |
|---|---|
| 3 | 0 |
| 1 | 1 |

does not contain 12

target = 13

matchingPartner = target - 1 = 12

If the matchingPartner is **not** in the hashmap, then we store the **current number** we have just seen and the **index** we saw it at in the hashmap

# Two Sum

Hashmap

↓

vec = 

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

target = 13

matchingPartner = target - 4 = 9

| value | index |
|---|---|
| 3 | 0 |
| 1 | 1 |
| 4 | 2 |

does not contain 9

If the matchingPartner is **not** in the hashmap, then we store the **current number** we have just seen and the **index** we saw it at in the hashmap

# Two Sum

vec = 

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

target = 13

matchingPartner = target - 1 = 12

Hashmap

| value | index |
|---|---|
| 3 | 0 |
| 1 | 3 |
| 4 | 2 |

does not contain 12

If the matchingPartner is **not** in the hashmap, then we store the **current number** we have just seen and the **index** we saw it at in the hashmap

# Two Sum

vec =

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

target = 13

matchingPartner = target - 5 = 8

Hashmap

| value | index |
|---|---|
| 3 | 0 |
| 1 | 3 |
| 4 | 2 |
| 5 | 4 |

does not contain 8

If the matchingPartner is **not** in the hashmap, then we store the **current number** we have just seen and the **index** we saw it at in the hashmap

# Two Sum

vec = 

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

target = 13

matchingPartner = target - 9 = 4

Hashmap

| value | index |
|---|---|
| 3 | 0 |
| 1 | 3 |
| 4 | 2 |
| 5 | 4 |
| 9 | 5 |

does contain 4

If the matchingPartner is in the hashmap, then we have found a two sum and we return the index of both numbers

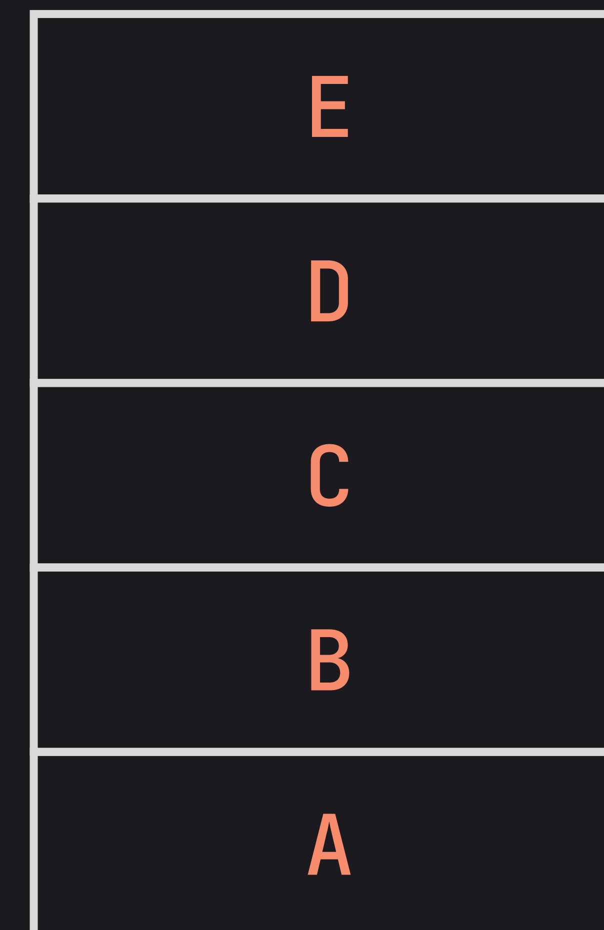
# Two Sum

```
function twoSum(vec, target):
    partners = {}
    for i = 0 to vec.size()-1:
        matchingPartner = target - vec[i]
        if matchingPartner in partners:
            return { partners[matchingPartner], i }
        else:
            partners[vec[i]] = i
    return { 0, 0 }
```

# Stacks

# Stacks

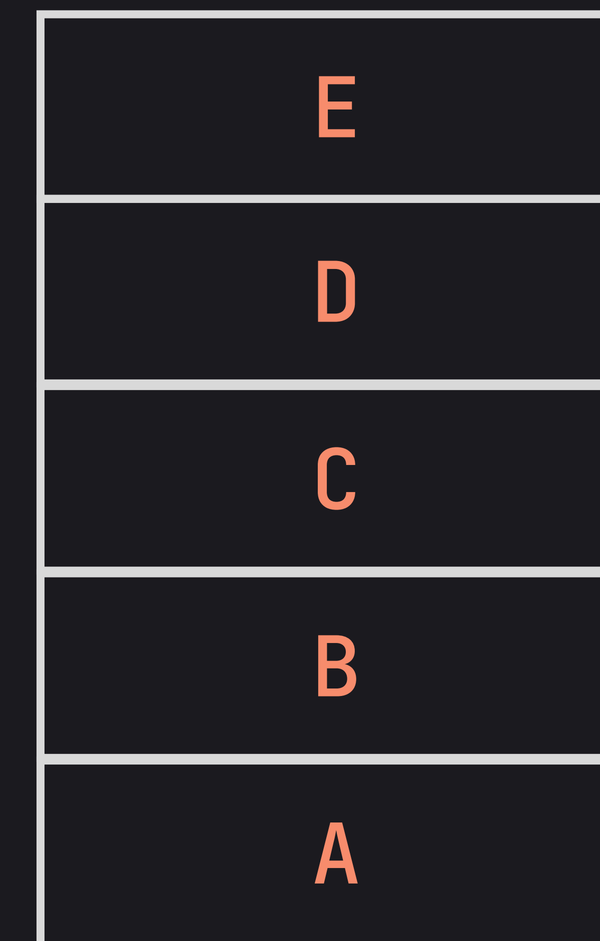
Stacks are a collection of elements that behave like a stack of coins or plates



# Stacks

```
void pop();
```

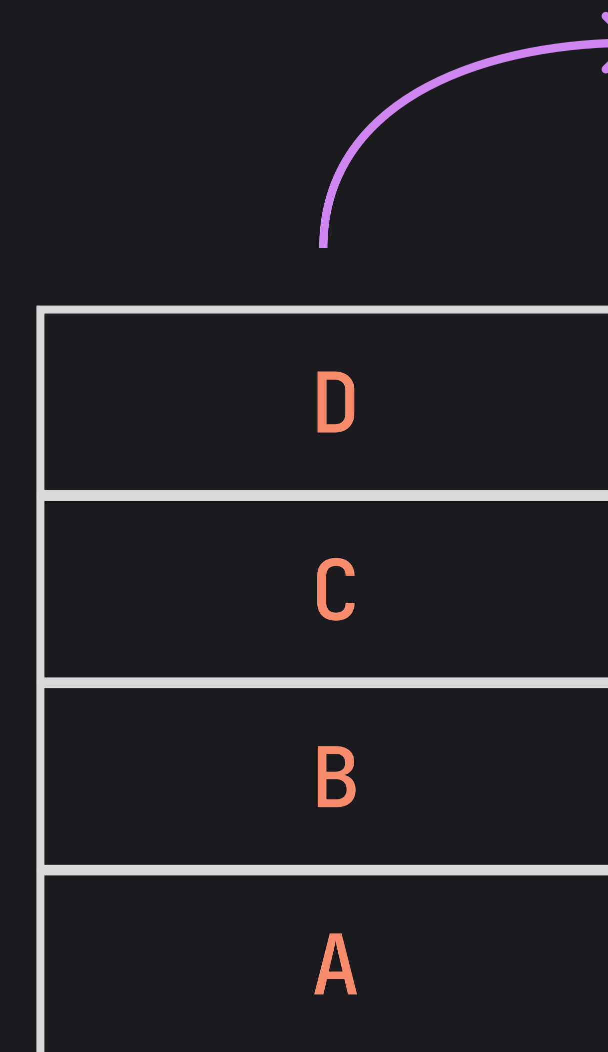
remove an element from the top of stack




```
stack.pop()
```

```
void pop();
```

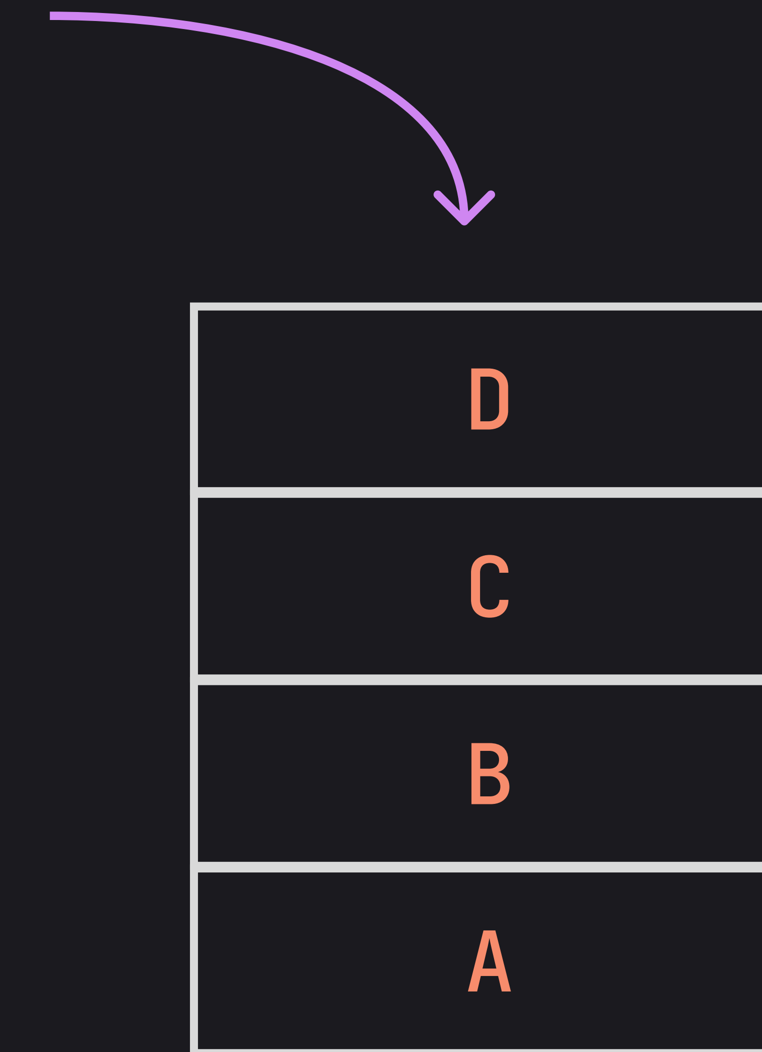
remove an element from the top of stack




```
stack.pop()
```

```
void push(E);
```

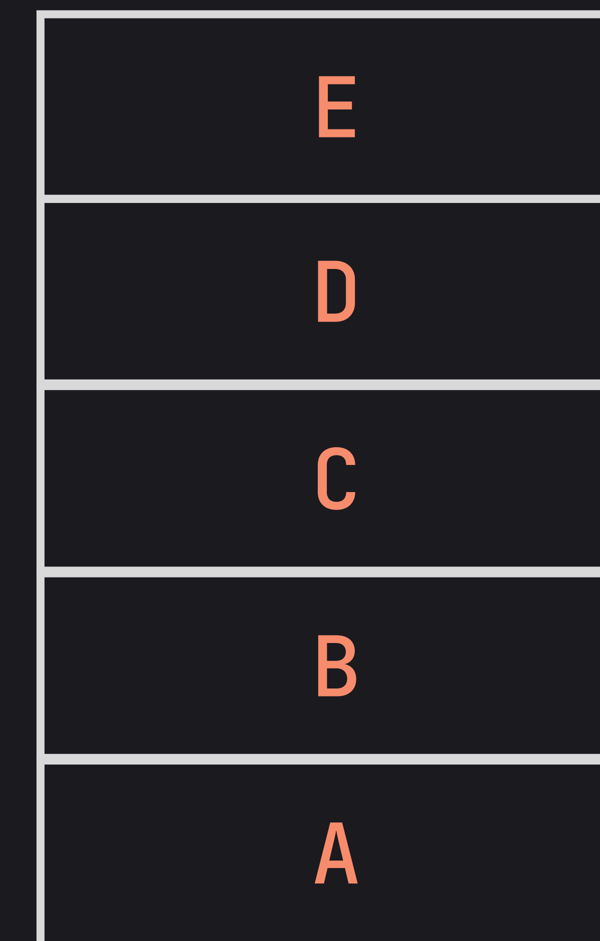
Adds an element to the top of stack




```
stack.pop()
```

```
void push(E);
```

Adds an element to the top of stack

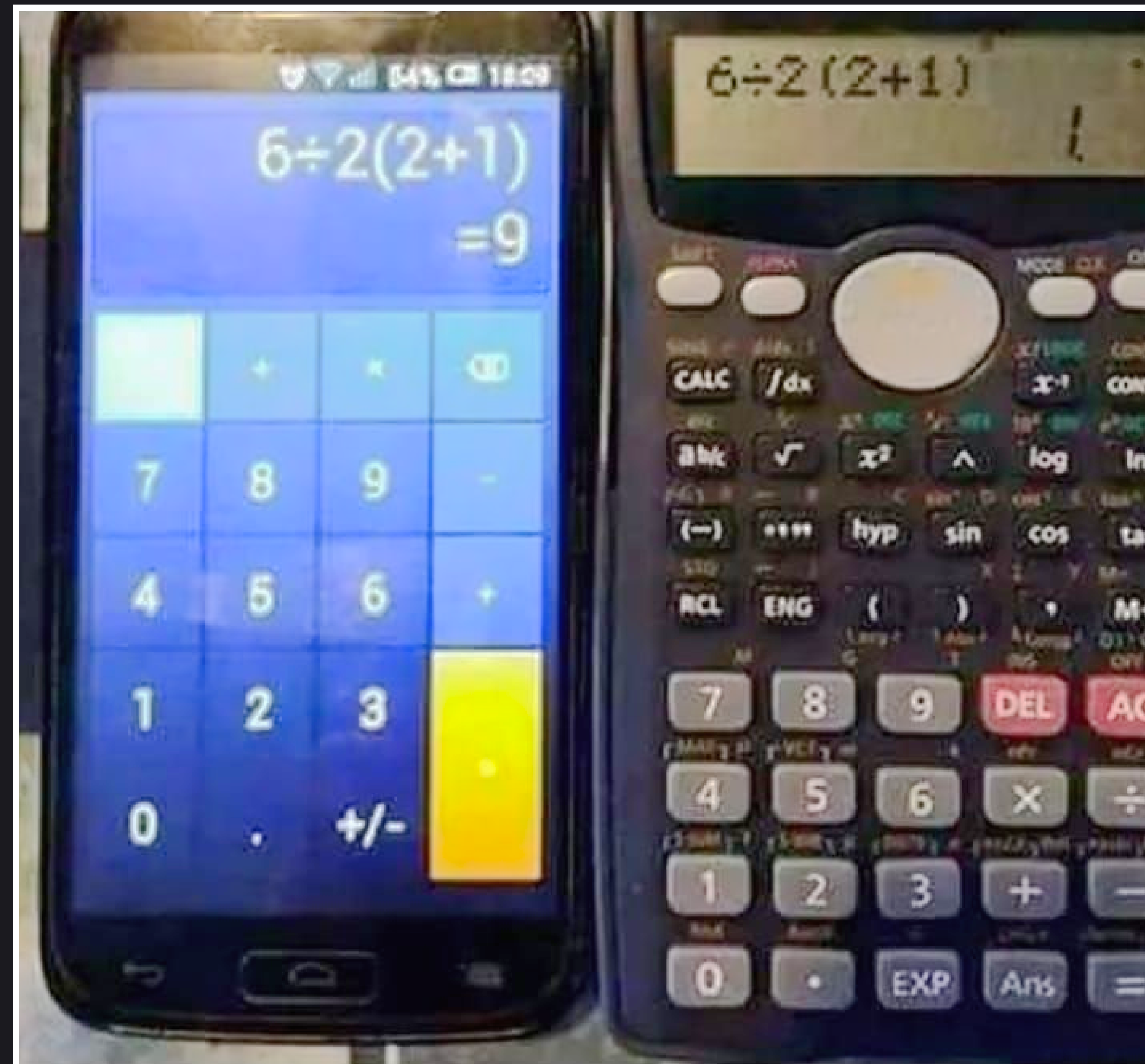


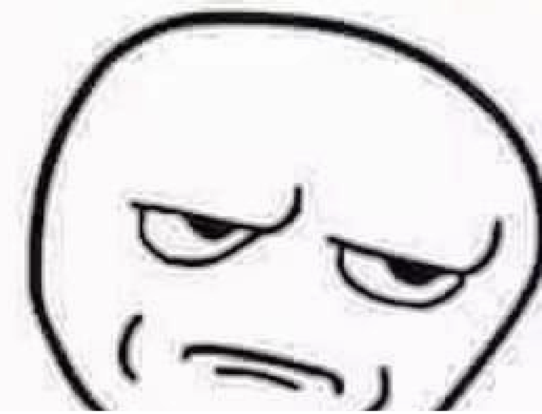

stack.pop()

# Stack Implementation

```
void push(T val); // add an element to the top of stack

void pop(); // remove an element from the top of stack

T& top(); // get the element at the top of stack

int size(); // return number of elements in the stack

bool empty(); // check if stack is empty
```

# Stack Based Calculator





# Simple Calculator

2 + 3

Lets look at how a simple calculator would work

# Simple Calculator

'2' '+' '3'

The sum is given to us as a collection of chars

# Simple Calculator

```cpp
std::vector<char> tokens = { '2', '+', '3' }
```

Which we store in a vector

# Simple Calculator

```
std::vector<char> tokens = { '2', '+', '3' }

eval(tokens); // returns: 5
```

And then we can pass this vector to a function that evaluates the expression

# More Complex Operations

5 * (2 + 3)

If we try and do a ***slightly*** more complex

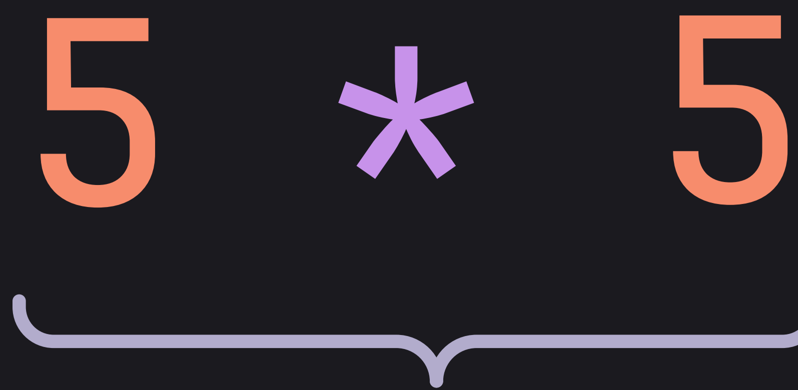
we see that writing a calculator is
actually quite hard

# More Complex Operations

5 * (2 + 3)

we first need to calculate this

# More Complex Operations




Then we can
tackle this

# More Complex Operations

25

Because we don’t alway calculate ***left to right***
writing an algorithm for this can be quite hard

# Reverse Polish Notation

# Reverse Polish Notation

| infix notation (normal way) | postfix notation (Reverse Polish) |
| --- | --- |
| 1 + 3 | 1, 3 + |
| 6 - 5 | 6, 5 - |
| 4 * 7 | 4, 7 * |
| 9 / 3 | 9, 3 / |

# Reverse Polish Notation

| infix notation (normal way) | postfix notation (Reverse Polish) |
| --- | --- |
| 1 + 3 | 1 3 + |
| 6 - 5 | 6 5 - |
| 4 * 7 | 4 7 * |
| 9 / 3 | 9 3 / |

# Reverse Polish Notation

| infix notation (normal way) | postfix notation (Reverse Polish) |
| --- | --- |
| (1 + 5) + 3 | (1, 5 +), 3 + |
| (2 * 3) - 5 | (2, 3 *), 5 - |
| (4 + 1) * (7 - 3) | (4, 1 +), (7, 3 -) * |

# Reverse Polish Notation

| infix notation (normal way) | postfix notation (Reverse Polish) |
| --- | --- |
| 1 + 5 + 3 | 1, 5 +, 3 + |
| 2 * 3 - 5 | 2, 3 *, 5 - |
| (4 + 1) * (7 - 3) | 4, 1 +, 7, 3 - * |

One neat thing about RPN is that you don’t need brackets

# Reverse Polish Notation

| infix notation (normal way) | postfix notation (Reverse Polish) |
| --- | --- |
| 1 + 5 + 3 | 1 5 + 3 + |
| 2 * 3 - 5 | 2 3 * 5 - |
| (4 + 1) * (7 - 3) | 4 1 + 7 3 - * |

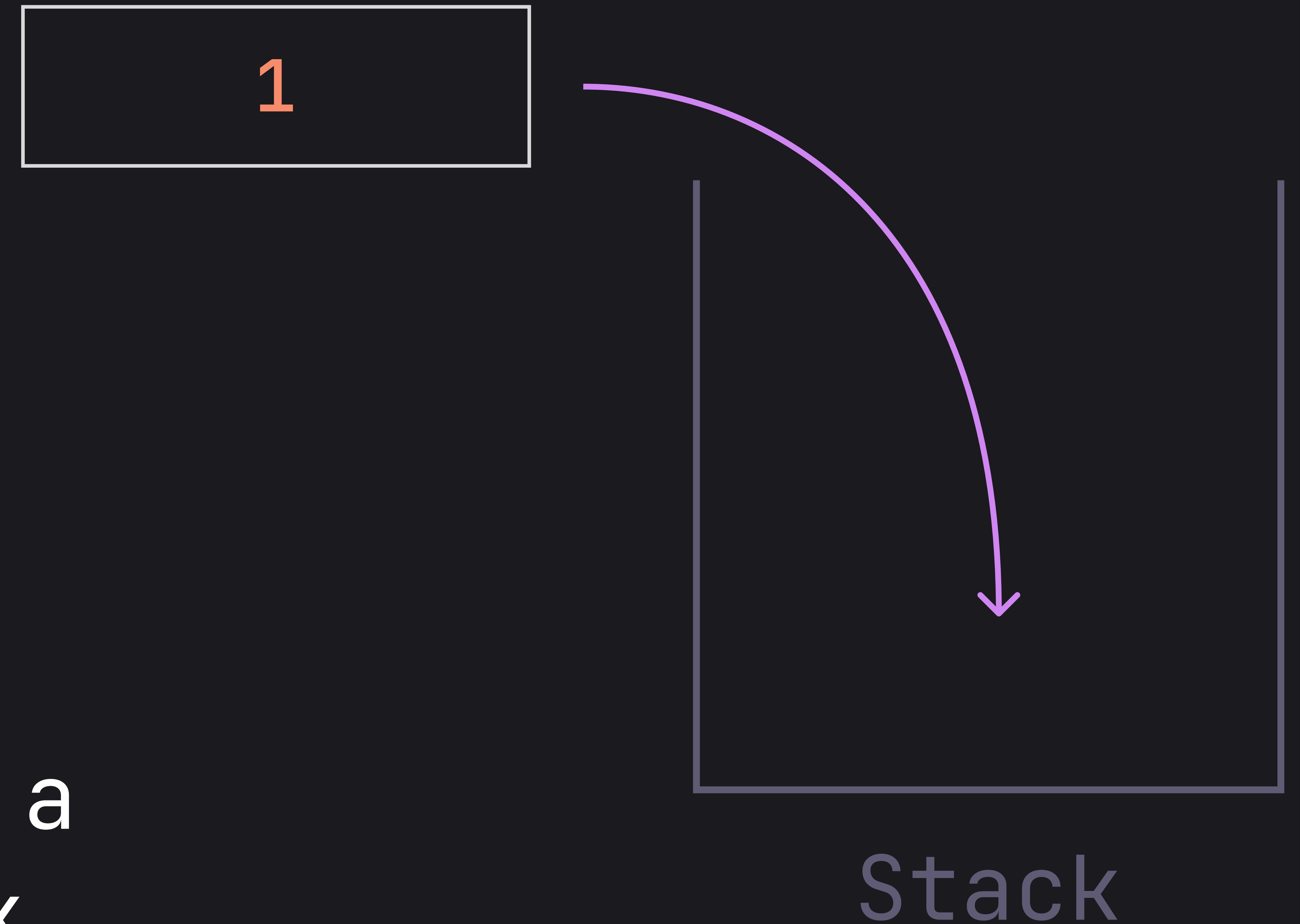
One neat thing about RPN is that you don't need brackets

# Reverse Polish Notation

Another neat thing about RPN is its easy to compute

1 3 +

If the current element is a **number** add it to the stack

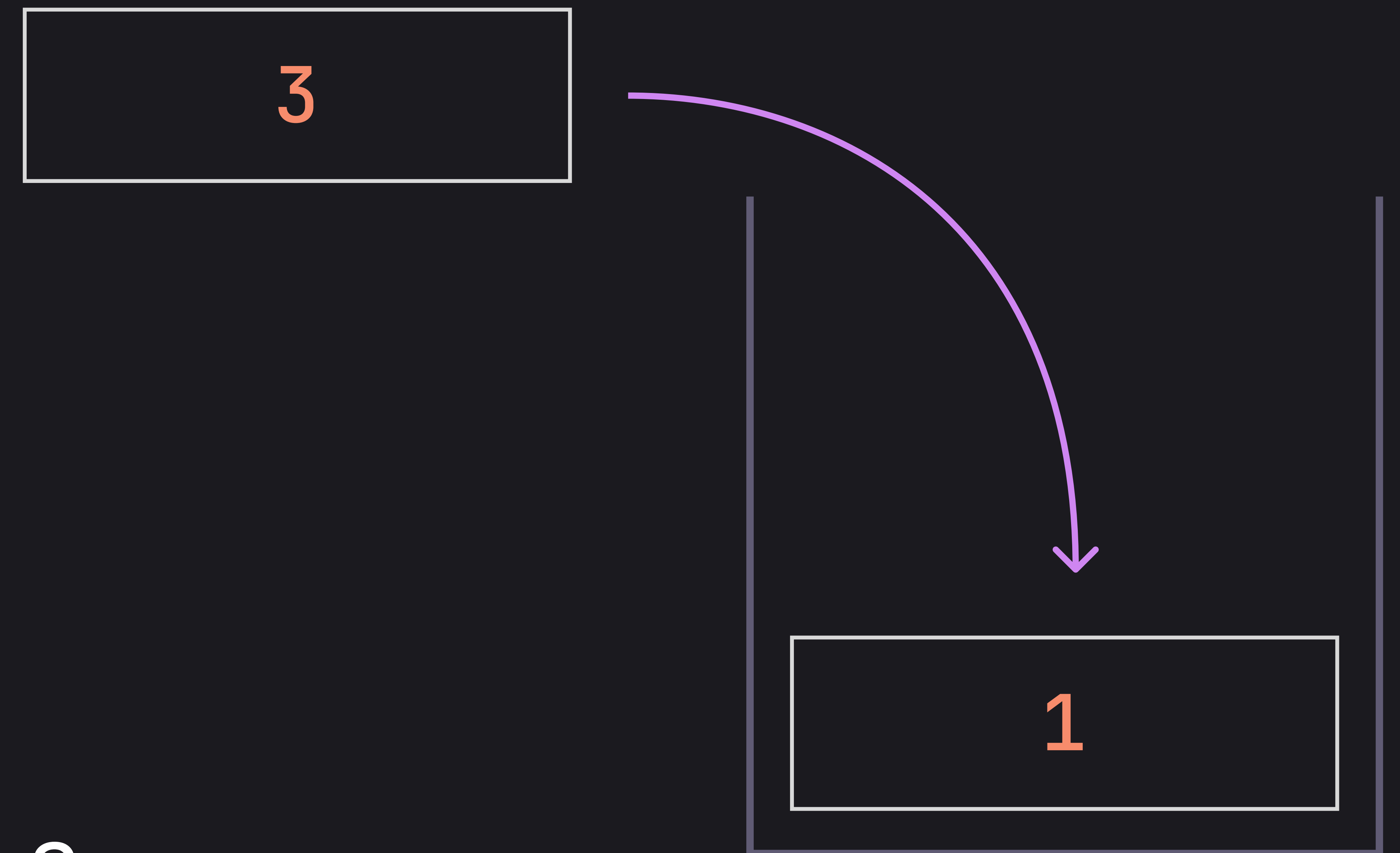


# Reverse Polish Notation

Another neat thing about RPN is its easy to compute

1 3 +

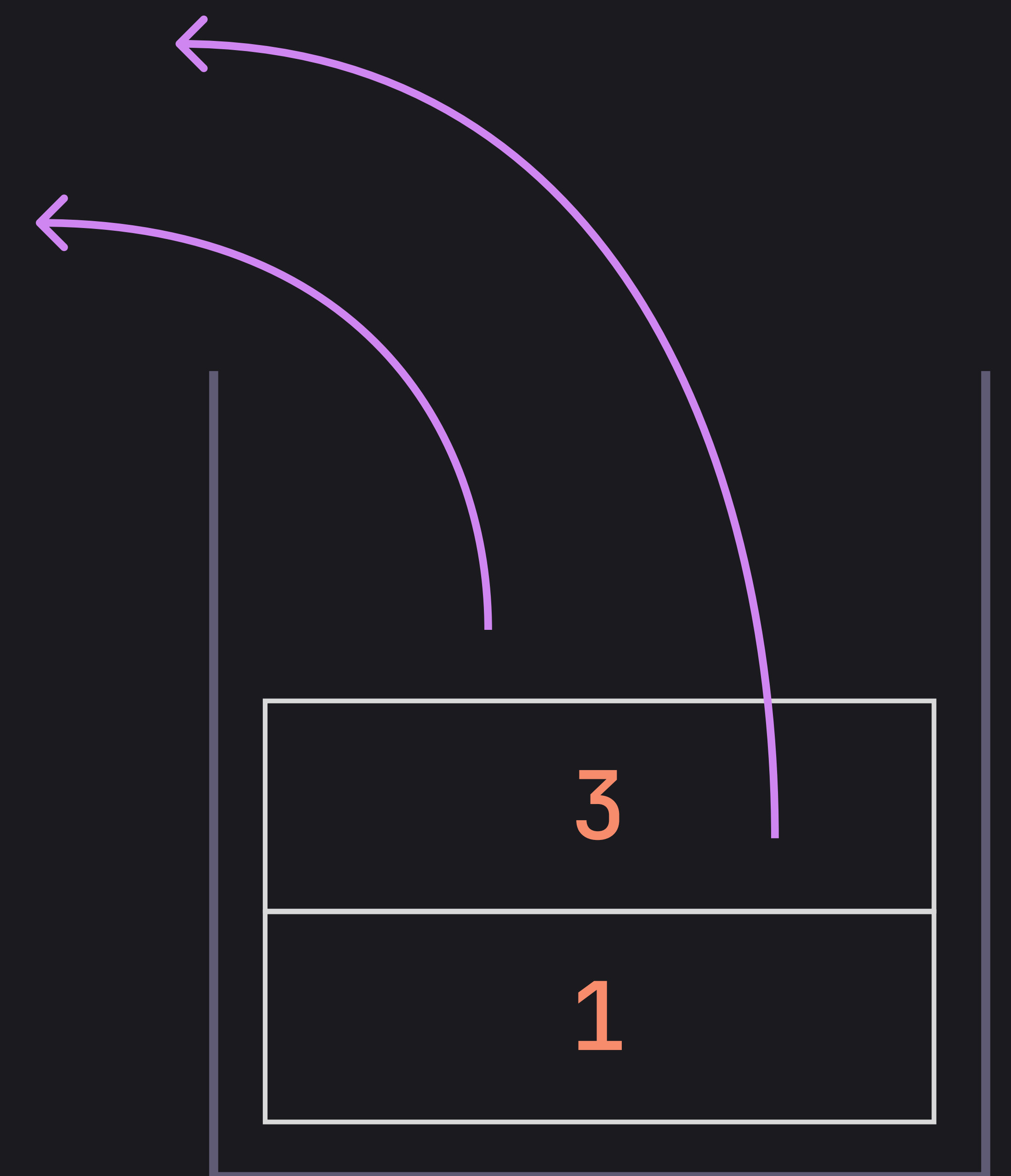
If the current element is a **number** add it to the stack



# Reverse Polish Notation

Another neat thing about RPN is its easy to compute

1 3 +




If the current element is an **operator**
pop two elements from the stack

# Reverse Polish Notation

Another neat thing about RPN is its easy to compute

1 + 3 = 4

1 3 +

↑

Then apply the operator to those two elements

# Reverse Polish Notation

Another neat thing about RPN is its easy to compute

4

1 3 +

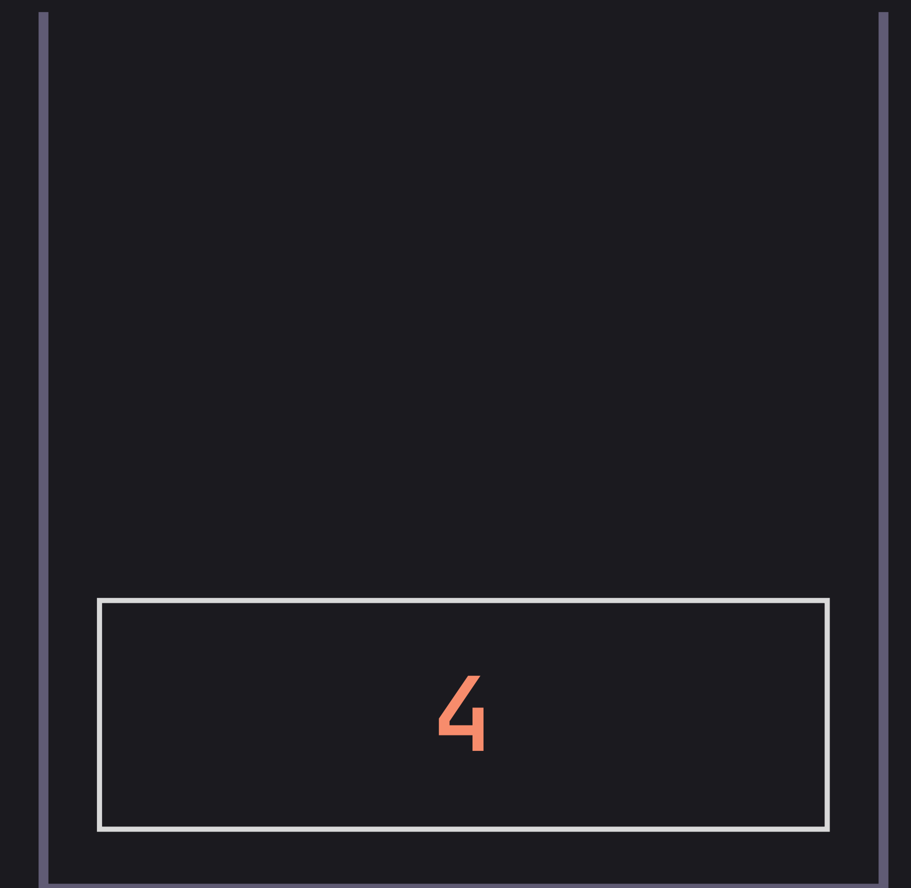
Now finally add that result back to the stack

# Reverse Polish Notation

Another neat thing about RPN is its easy to compute

1 3 +

Once we have processed the equation, the answer should be the only element in the stack

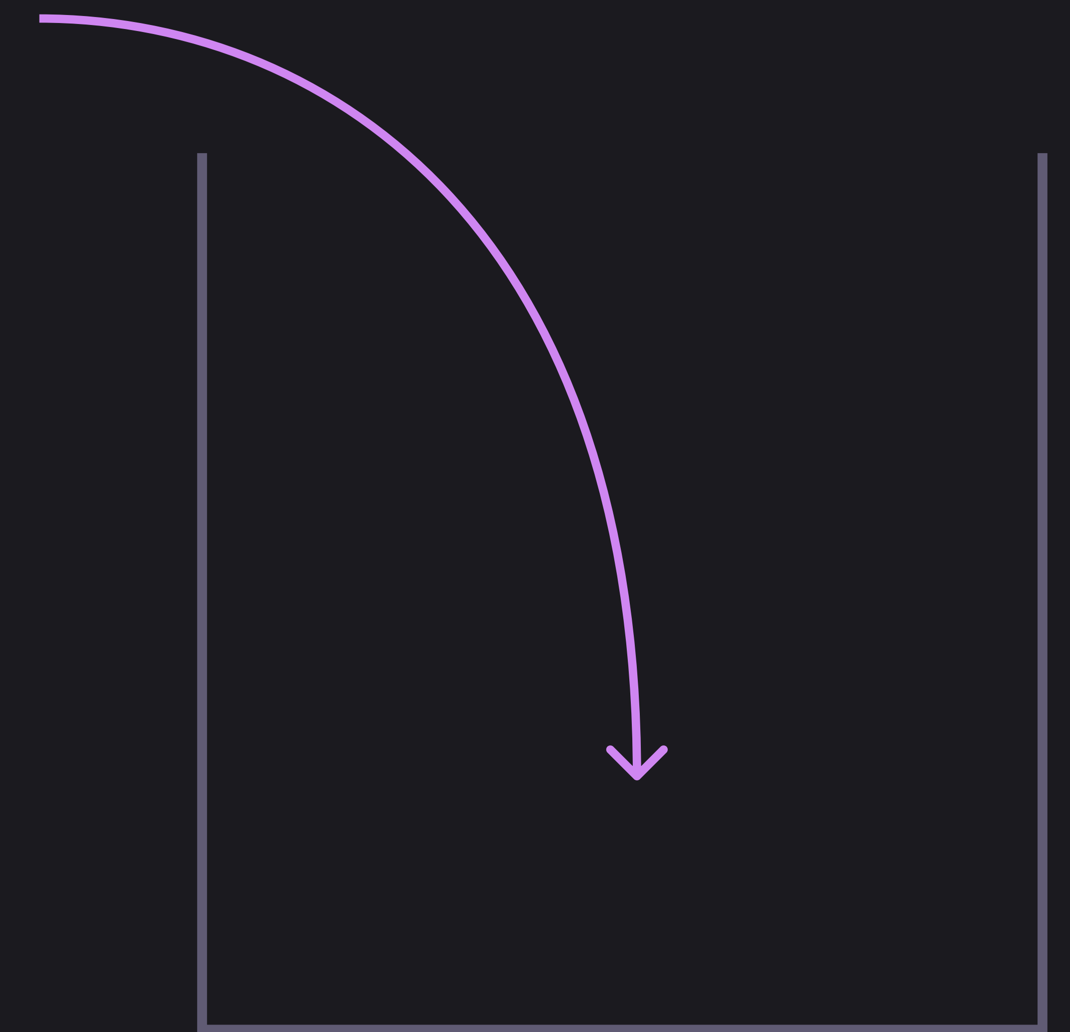


# Reverse Polish Notation

Lets try a harder example

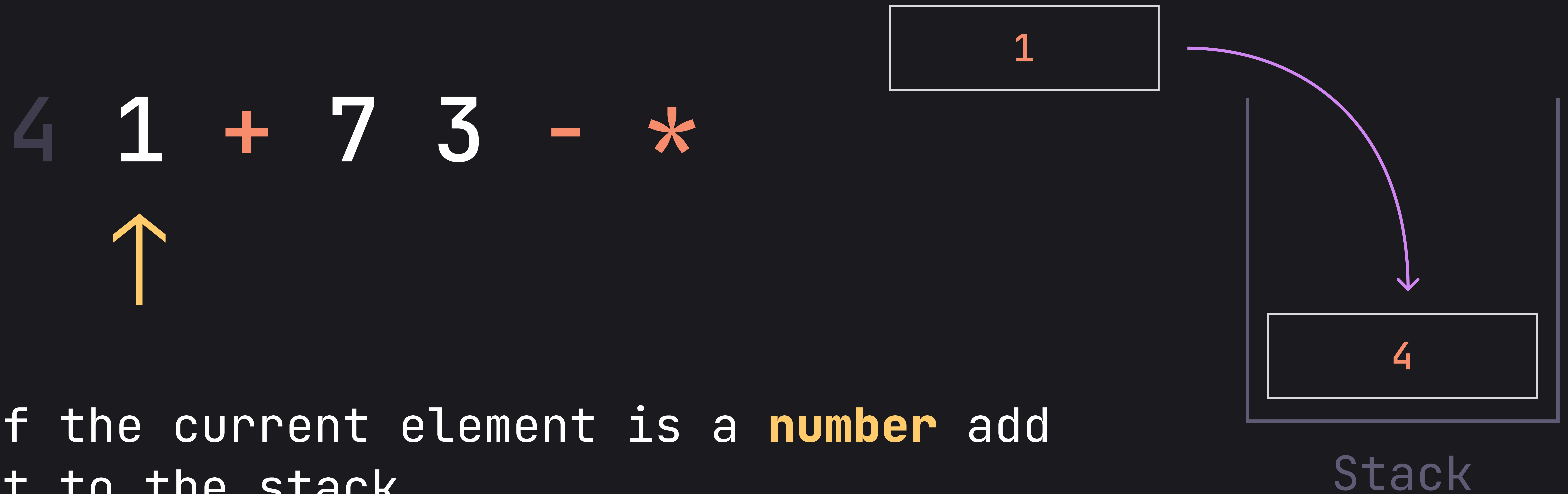
4 1 + 7 3 - *

If the current element is a **number** add it to the stack



# Reverse Polish Notation

Lets try a harder example




If the current element is a **number** add it to the stack

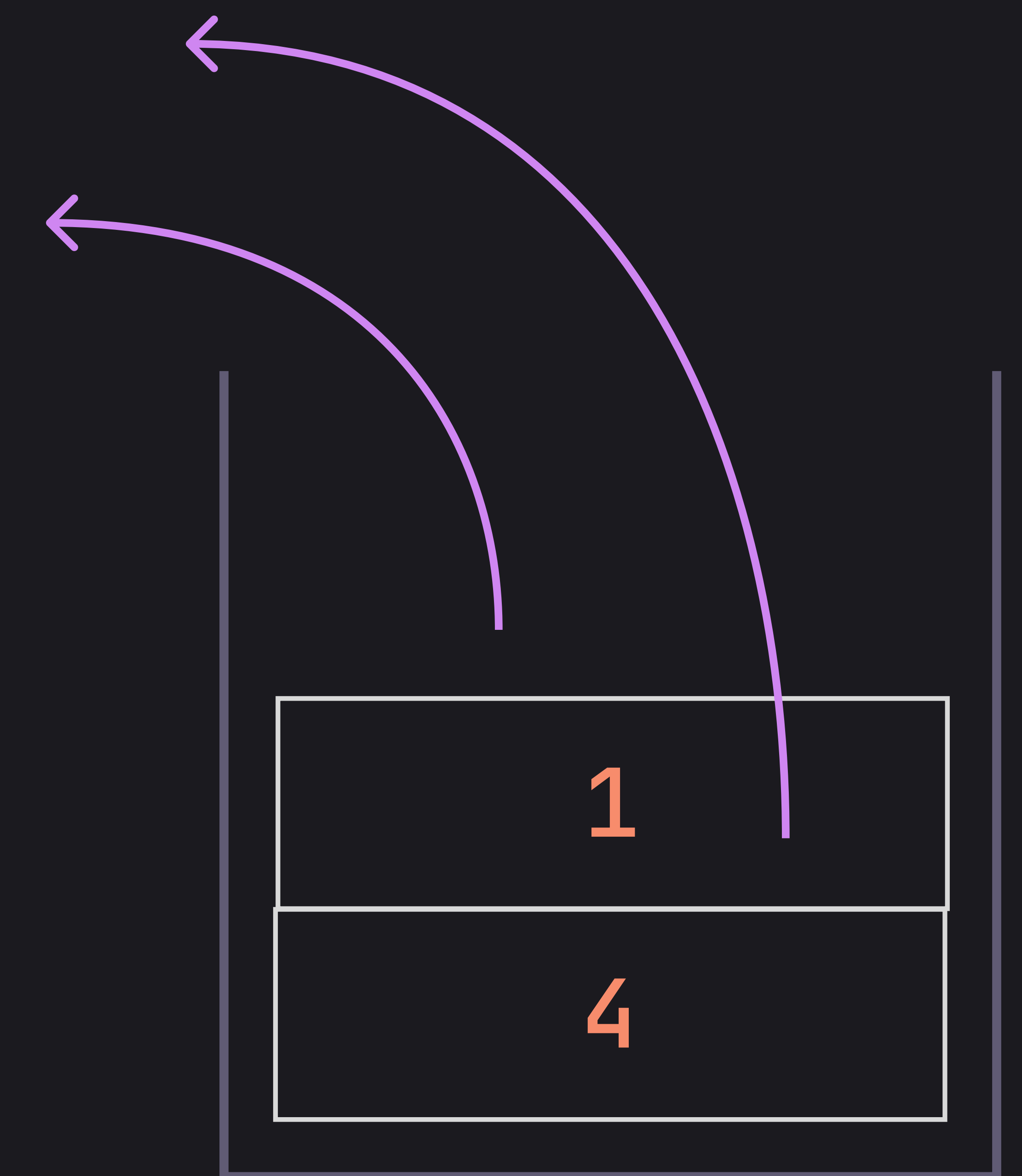
# Reverse Polish Notation

Lets try a harder example

4 1 + 7 3 - *

↑

If the current element is an **operator**
pop two elements from the stack



# Reverse Polish Notation

Lets try a harder example

4 + 1 = 5

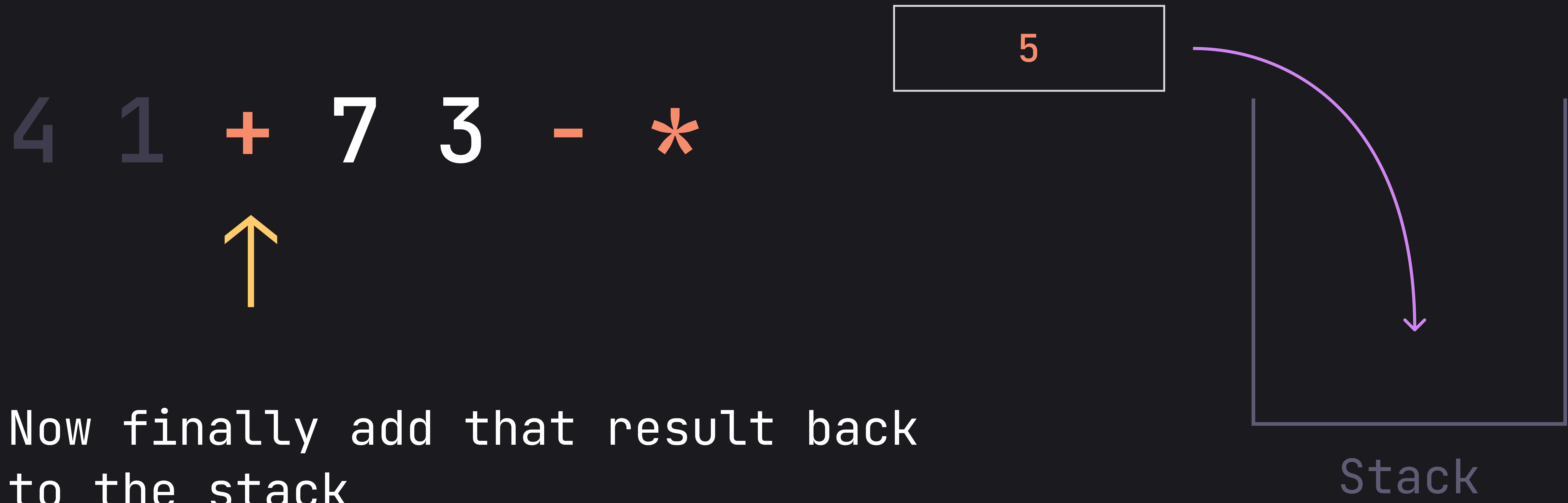
4 1 + 7 3 - *

↑

Then apply the operator to those two elements

# Reverse Polish Notation

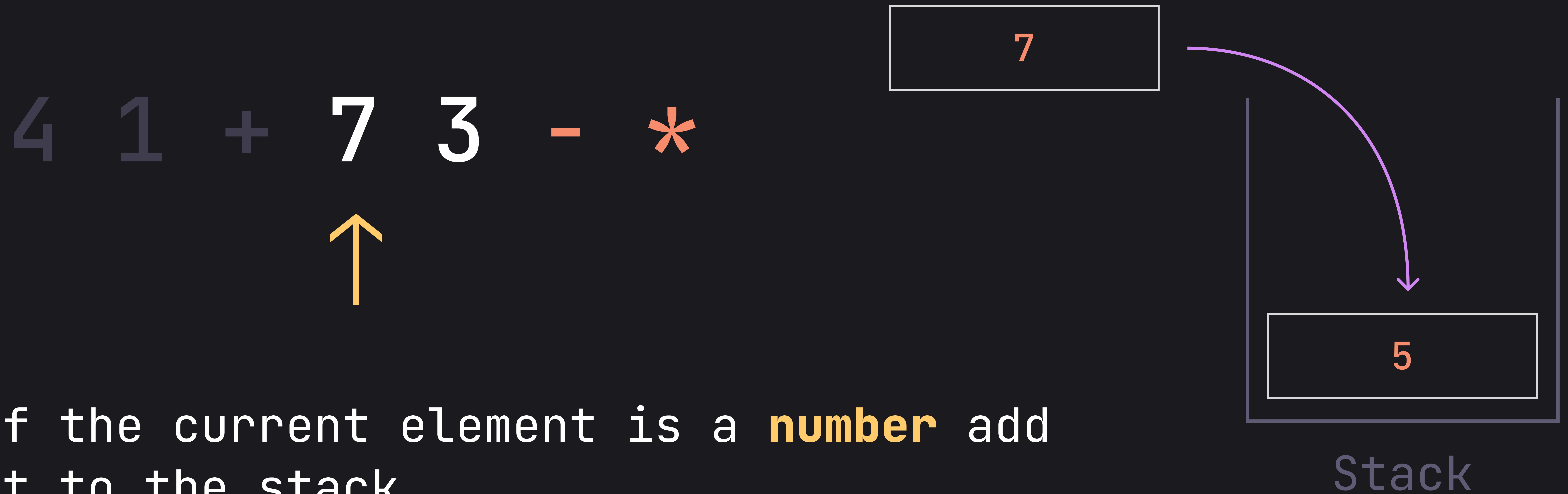
Lets try a harder example




Now finally add that result back to the stack

# Reverse Polish Notation

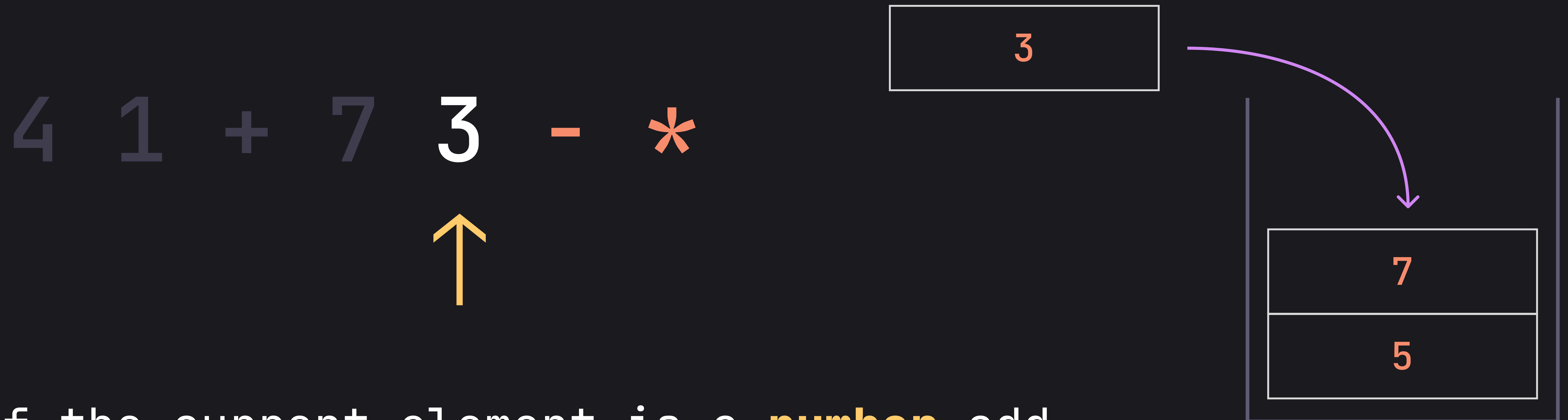
Lets try a harder example




If the current element is a **number** add it to the stack

# Reverse Polish Notation

Lets try a harder example



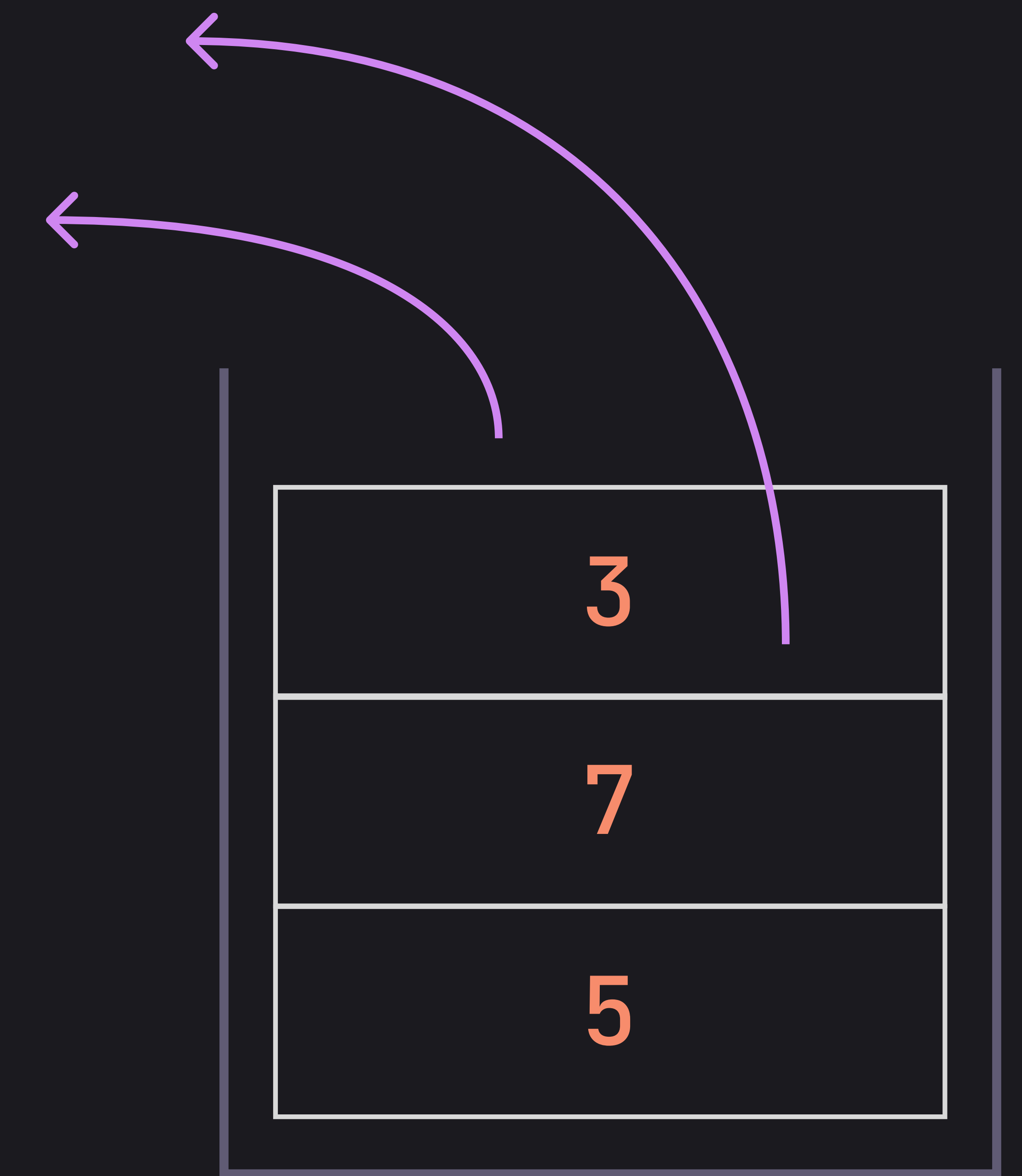

If the current element is a **number** add it to the stack

# Reverse Polish Notation

Lets try a harder example

4 1 + 7 3 - *

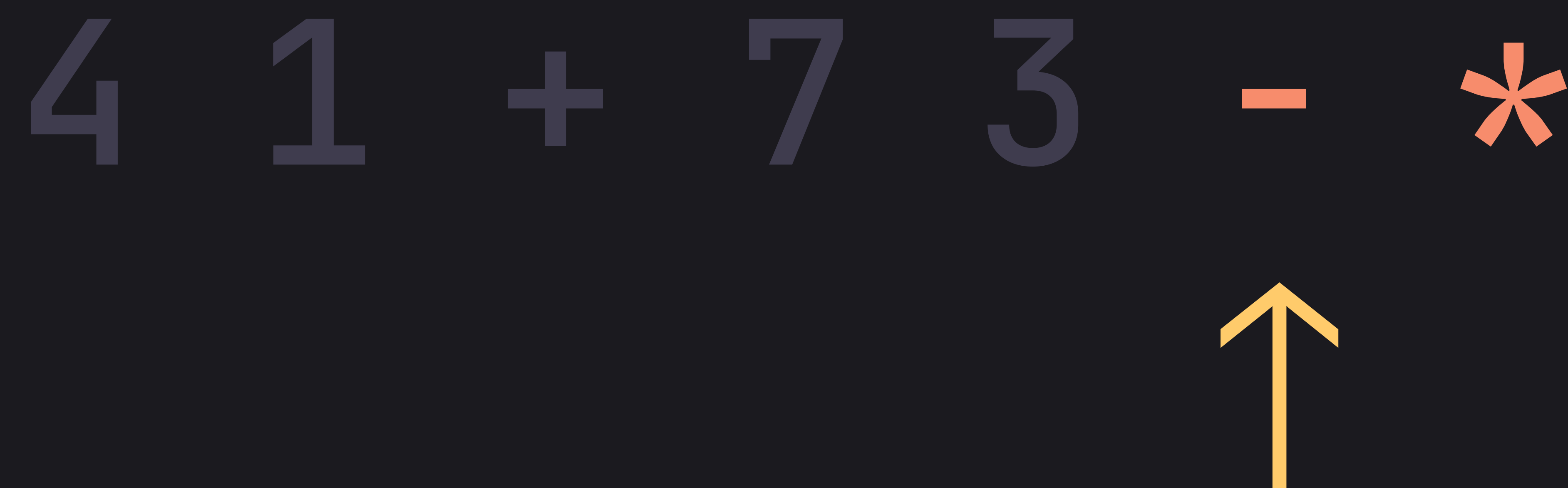
If the current element is an **operator**
pop two elements from the stack

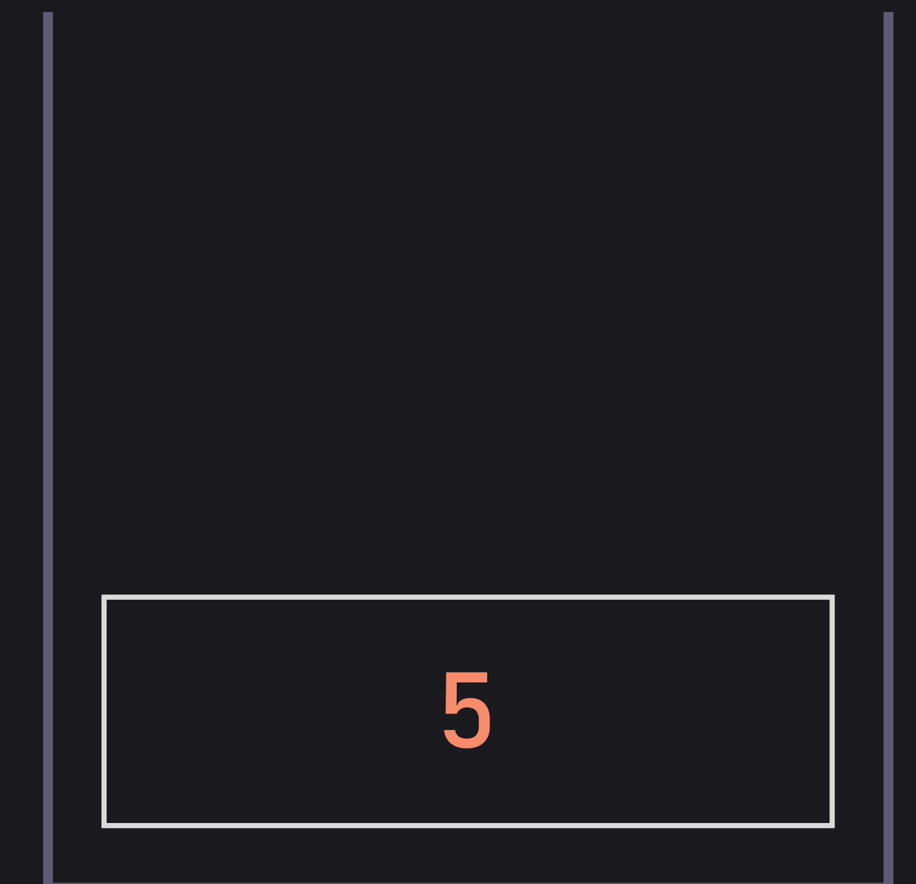


# Reverse Polish Notation

Lets try a harder example

7 - 3 = 4

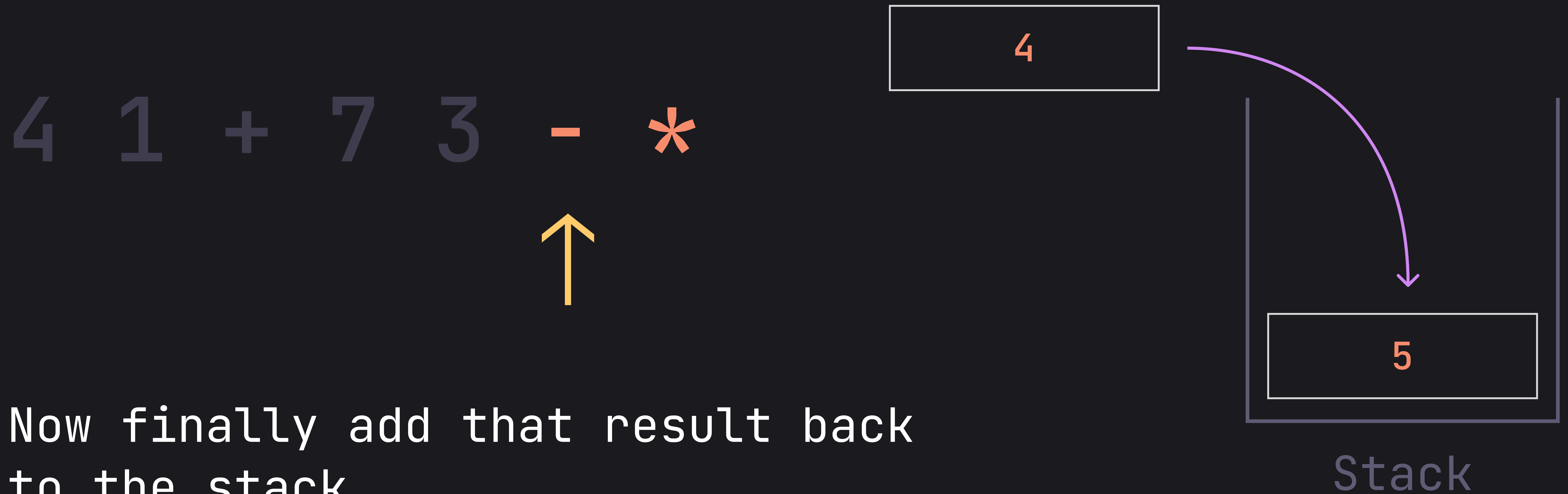



Then apply the operator to those two elements

# Reverse Polish Notation

Lets try a harder example

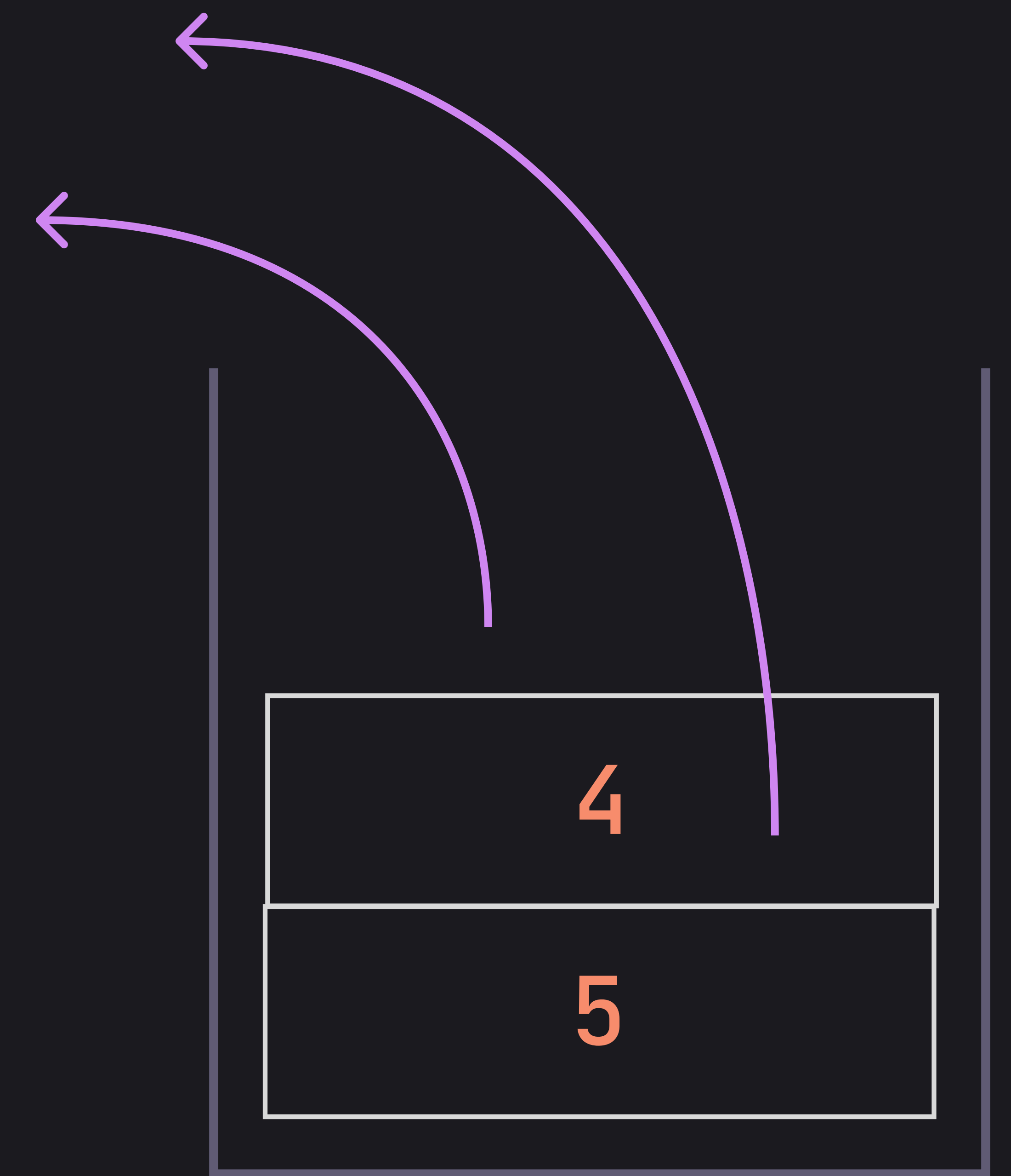



Now finally add that result back to the stack

# Reverse Polish Notation

Lets try a harder example

4 1 + 7 3 - *




If the current element is an **operator**
pop two elements from the stack

# Reverse Polish Notation

Lets try a harder example

5 * 4 = 20

4 1 + 7 3 - *

Then apply the operator to those two elements

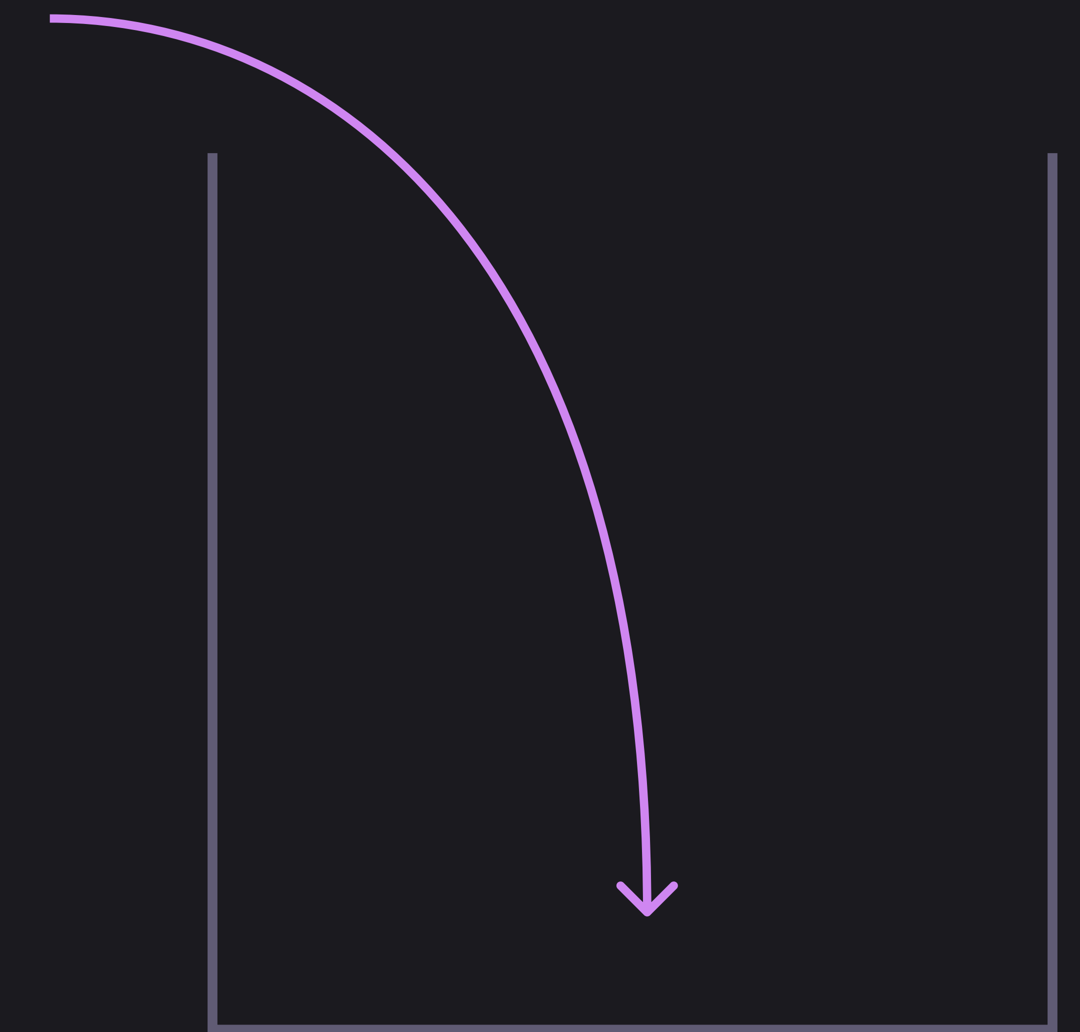
# Reverse Polish Notation

Lets try a harder example

20

4 1 + 7 3 - *

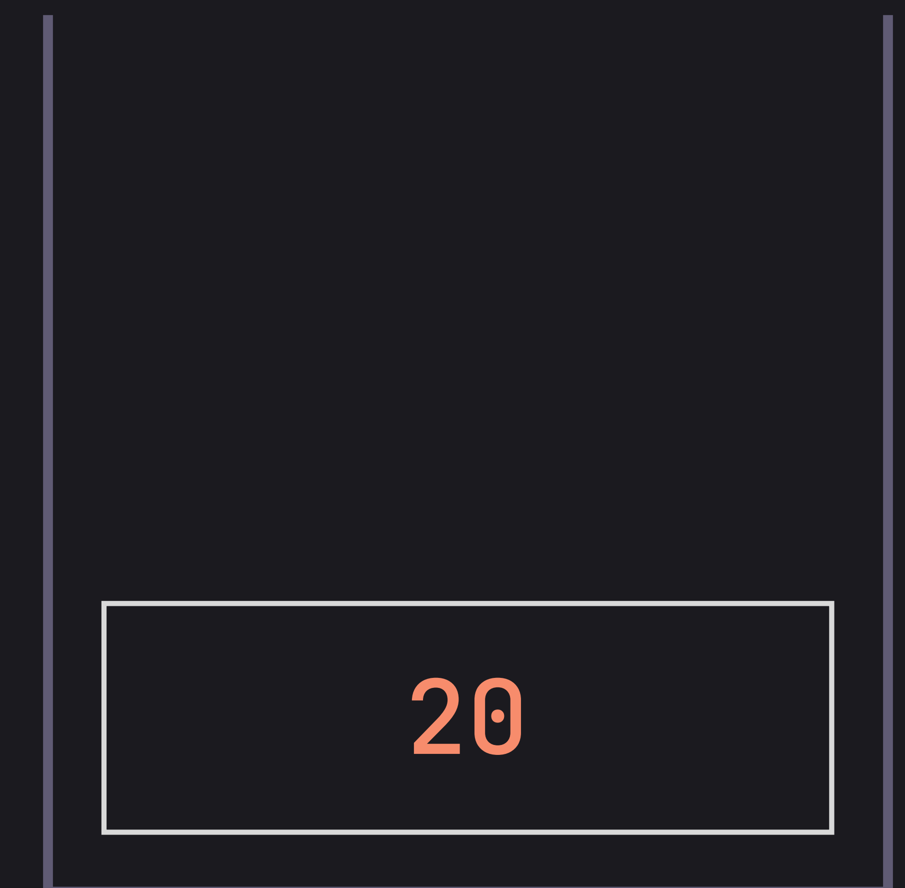
Now finally add that result back to the stack



# Reverse Polish Notation

Lets try a harder example

4 1 + 7 3 - *




Once we have processed the equation, the answer should be the only element in the stack

# Reverse Polish Notation

*psuedo code*

```
function evalRPN(tokens):
    if tokens is empty:
        return 0
    stack = []
    for token in tokens:
        if isDigit(token):
            stack.push(int(val))
        else:
            second = stack.pop()
            first = stack.pop()
            if token is '+':
                val = first + second
            else if token is '*':
                val = first * second
            else if token is '-':
                val = first - second
            else if token is '/':
                val = first / second
            else:
                throw Error("invalid token")
            stack.push(val)
    return stack.top()
```