STACKS

Mahsa Ziraksima
ACIBADEM UNIVERSITY
Fall 2025

SUMMARY

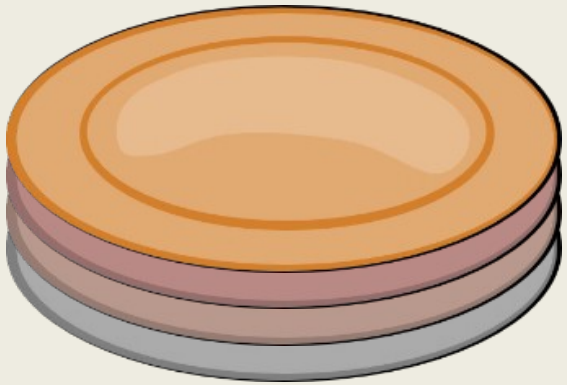
- Stacks’ Performance
- Creating a Stack
- Using Stacks to Reverse Strings
- Min Stack
- Stacked Parentheses

STACK

- A stack is an abstract data type and a linear data structure that allows you to remove only the most recent element you added.

  example of a **last- in, first- out** (**LIFO**) data structure

- where the last item you put into the data structure is the first item to come out of it.
- **limited-access** data structure: a type of data structure that forces you to access its information in a particular order.

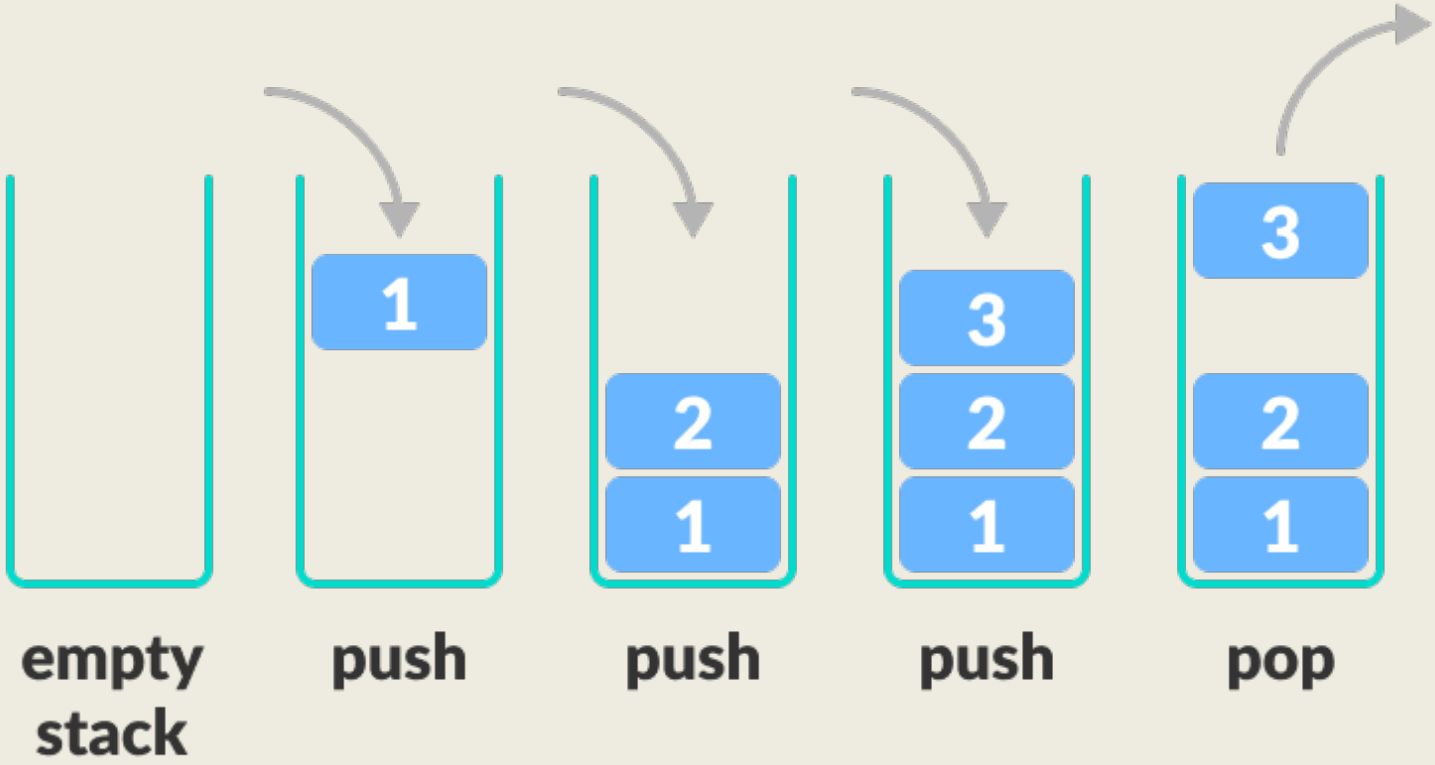


STACK

- Primary operations:
  - ***Pushing*** *an item onto a stack means putting a new item in it.*
  - ***Popping*** *an item from a stack means removing the last item from it.*

- A **bounded** stack is a stack that limits how many items you can add to it.
- An **unbounded** stack is a stack that does not limit how many elements you can add to it.

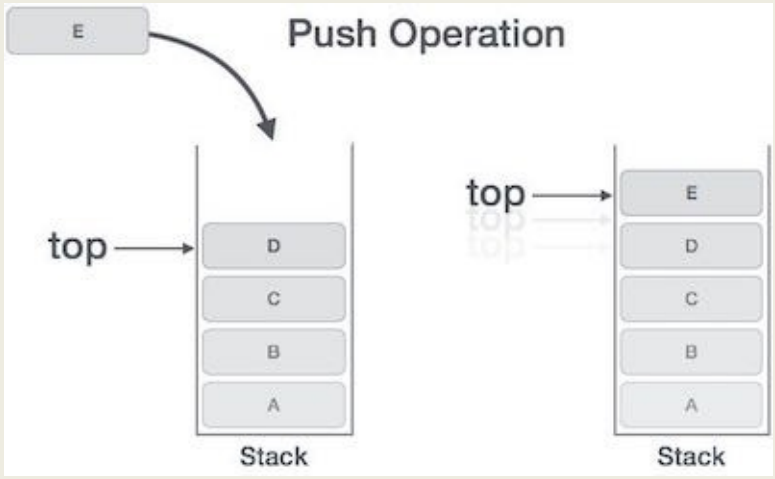
STACK



STACK

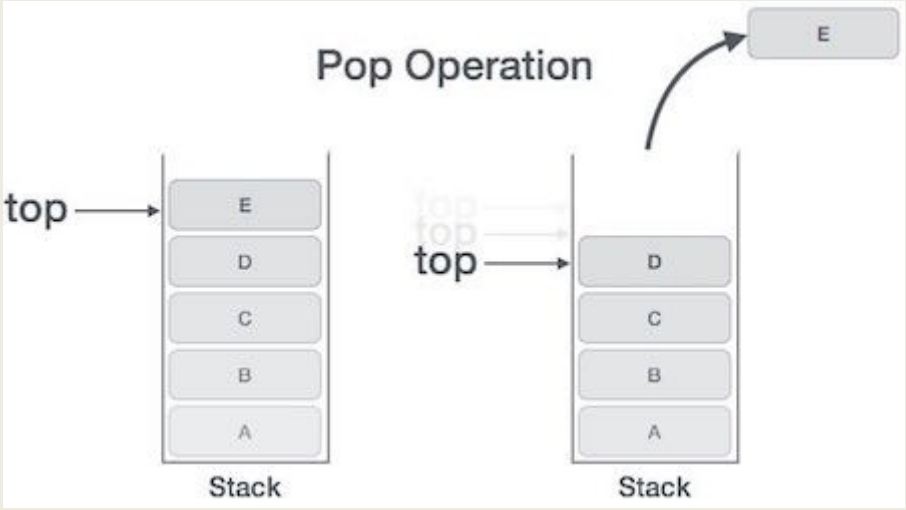
- Push

  ✓ top = stack size → stack is full



- pop

  ✓ top = 0 → stack is empty



STACK

- Challenge: what would be the results of A, B, and C?

n = 5 A = 10 B = 2 C = 5

push (B)
push (A + B)
pop (C)
push (A – B)
push (C)
push (B)
pop (A)
pop (B)
push (A × B)
push (C)
push (A)
pop (B)
pop (C)
pop (A)

		2	
	2	12	
	12	24	
12	8	8	8
2	2	2	2

A	B	C
~~10~~	~~2~~	~~5~~
~~2~~	~~12~~	~~12~~
24	2	12

STACK

- Challenge: which of the following sequences could be produced?
  - *2 1 3 4* ✔
  - *3 1 4 2* ✖
  - *3 2 4 1* ✔
  - *4 2 3 1* ✖
  - *4 3 1 2* ✖

push 1
push 2
pop 2
pop 1
push 3
pop 3
push 4
pop 4

push 1
push 2
push 3
pop 3
pop 2
push 4
pop 4
pop 1

STACK

```
3
3
2
1
```

```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, data):
        self.items.append(data)

    def pop(self):
        return self.items.pop()

    def size(self):
        return len(self.items)

    def is_empty(self):
        return len(self.items) == 0

    def peek(self):
        return self.items[-1]


stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)


print(stack.peek())

for i in range(3):
    print(stack.pop()
```

STACK

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None


class Stack:
    def __init__(self):
        self.head = None

    def push(self, data):
        node = Node(data)
        if self.head is None:
            self.head = node
        else:
            node.next = self.head
            self.head = node

    def pop(self):
        if self.head is None:
            raise IndexError('pop from empty stack')
        poppednode = self.head
        self.head = self.head.next
        return poppednode.data
```

```
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)

for i in range(3):
    print(stack.pop())

stack = []
print(stack)
stack.append('Kanye West')
print(stack)
stack.append('Jay-Z')
print(stack)
stack.append('Chance the Rapper')
print(stack)
stack.pop()
print(stack)
```

```
3
2
1
[]
['Kanye West']
['Kanye West', 'Jay-Z']
['Kanye West', 'Jay-Z', 'Chance the Rapper']
['Kanye West', 'Jay-Z']
```

STACK

- Performance:
  - *Pushing and popping elements from a stack are all O(1)*
  - *Printing:*
    - printing each element as it comes off your stack, O(n)
    - printing by using a temporary stack, O(2*n)
- Accessing an element is O(n)

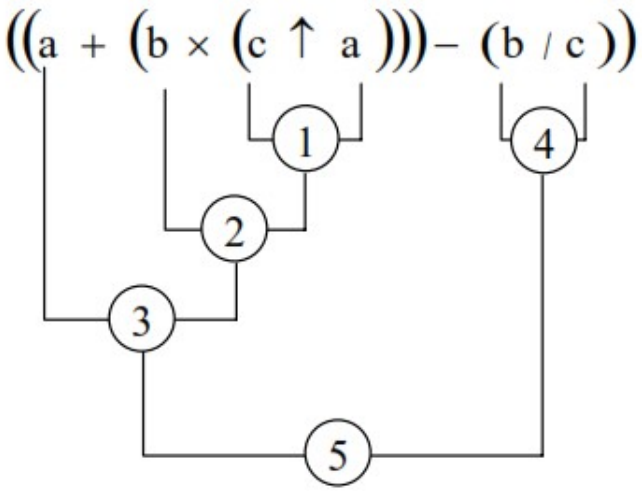
Data Structure	Time Complexity							
	Average				Worst			
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)

STACK

- Challenge: given a computational expressions, convert its infix form to postfix.

a+b →

infix	a + b
postfix	ab +
prefix	+ ab



$\text{postfix} = (a(b(ca))\uparrow \times + (bc)/) = abca\uparrow \times + bc/-$

$\text{prefix} = -+a \times b \uparrow ca/bc$

STACK

- Challenge: given a computational expressions, convert its infix form to postfix.

Algorithm

1. Scan the infix expression from left to right.

2. If the scanned character is an operand, output it.

3. Else,

.....**3.1** If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '('), push it.

.....**3.2** Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

4. If the scanned character is an '(', push it to the stack.

5. If the scanned character is an ')', pop the stack and and output it until a '(' is encountered, and discard both the parenthesis.

6. Repeat steps 2-6 until infix expression is scanned.

7. Pop and output from the stack until it is not empty.

8. Print the output

STACK

- Challenge: given a computational expressions, convert its infix form to postfix.

example	stack	output
(3 + 5) * 4	(	
(3 + 5) * 4		3
(3 + 5) * 4	(+	3
(3 + 5) * 4	(+	3 5
(3 + 5) * 4		3 5 +
(3 + 5) * 4	*	3 5 +
(3 + 5) * 4	*	3 5 + 4
		3 5 + 4 *
3 * (5 + 4)		3
3 * (5 + 4)	*	3
3 * (5 + 4)	* (	3
3 * (5 + 4)	* (	3 5
3 * (5 + 4)	* (+	3 5
3 * (5 + 4)	* (+	3 5 4
3 * (5 + 4)	*	3 5 4 +
		3 5 4 + *

STACK

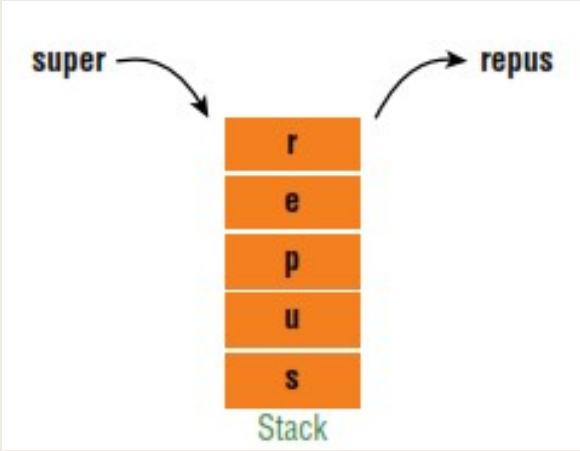
- Challenge: reverse a string three different ways.

```
a_string = "Hello world"

#1st method
print(a_string[::-1])

#2nd method
print(''.join(reversed(a_string)))

#3rd method
def reverse_string(a_string):
    stack = []
    string = ""
    for c in a_string:
        stack.append(c)
    for c in a_string:
        string += stack.pop()
    return string

print(reverse_string(a_string))
```

```
dlrow olleH
```

```
dlrow olleH
```

```
dlrow olleH
```



STACK

- Challenge: design a data structure that supports stack operations such as push and pop and includes a method to return the smallest element.

1. `self.main`
   - Stores all pushed values normally.
2. `self.min`
   - Stores the **minimum value so far** at each position.

O(1)

```
class MinStack():
    def __init__(self):
        self.main = []
        self.min = []

    def push(self, n):
        if len(self.main) == 0:
            self.min.append(n)
        elif n <= self.min[-1]:
            self.min.append(n)
        else:
            self.min.append(self.min[-1])
        self.main.append(n)

    def pop(self):
        self.min.pop()
        return self.main.pop()

    def get_min(self):
        return self.min[-1]
```

```
min_stack = MinStack()
min_stack.push(10)
min_stack.push(6)
min_stack.push(59)

print(min_stack.main)
print(min_stack.min)
print(min_stack.get_min())
print(min_stack.main)
print(min_stack.min)
min_stack.pop()
min_stack.pop()
min_stack.pop()
print(min_stack.main)
print(min_stack.min)
```

```
[10, 6, 59]
[10, 6, 6]
6
[10, 6, 59]
[10, 6, 6]
[]
[]
```

STACK

- Challenge: Given a string, use a stack to check whether it has balanced parentheses, meaning every time there is an open parenthesis, there is a subsequently closed parenthesis.

```
def check_parentheses(a_string):
    stack = []
    for c in a_string:
        if c == "(":
            stack.append(c)
        if c == ")":
            if len(stack) == 0:
                return False
            else:
                stack.pop()
    return len(stack) == 0


print(check_parentheses("int(12)3)"))
```

```
 (str(1)) # Balanced
print(Hi!)) # Not balanced

)(  )(
```

```
False
```