# Property-based Testing

Venkatesh-Prasad Ranganath
Kansas State University

Slides with * in title capture whiteboard content

# Testing a sort function*

Given X[1:n], sort returns X ascending order as R[1:m].

1. **Ordering**: for all 1 <= i < n, R[i] <= R[i+1]
2. **Same Length**: n == m [Redundant cos' 3 implies 2]
3. **Same Elements with Same Frequency**:
    1. If x in X, then x is in R and frequency(x, X) == frequency(x, R)
    2. If x in R, then x is in X and frequency(x, R) == frequency(x, X)

# What is a Property?*

- A statement that is true of any (valid) implementation of the specification
- A full set of rules that embodies the specification
- A singular expected behavior of the specification
- an attribute, quality, or characteristic of something (from a Dictionary)

From Spring'16

# What is a Property?*

- Type of values
- Some attributes of an UUT
  - Behavior, Structure, Shape
  - Holds for “all” instances
- Something that describes something
- Attribute of a function that is consistent independent of input
- Concept that should be realized for something to function as intended (*Isn’t this specification?*)

From Spring’17

# Properties for Identifying Type of Triangle

You are given a Python function type_of_triangle(x, y, z) that detects the type of triangle based on the lengths of its sides. The function accepts the lengths of the sides of a triangle as three  integers x, y, and z and returns either "equilateral", "isosceles", "scalene", or "not a triangle" depending on the type of triangle formed by the input.  The function *assumes* the input parameters will be positive integers (>0).  Further, the function *guarantees* to return only one of the above four strings.

Identify the properties to test the correctness of this function.  In stating the properties, you may use the names x, y, and z to denote the first, second, and third parameters of type_of_triangle.

# Properties for Identifying Type of Triangle

A. x == y == z

B. x == y != z or x == z != y or y == z != x

C. x != y != z

D. x+y > z and y+z > x and z+x > y

E. x > 0 and y > 0 and z > 0  (Assumed)

1. if A and D, then type_of_triangle(x,y,z) == “equilateral”
2. if B and D, then type_of_triangle(x,y,z) == “isosceles”
3. if C and D, then type_of_triangle(x,y,z) == “scalene”
4. if !D, then type_of_triangle(x,y,z) == “not a triangle”

# Properties of a Stack*

1. If len() > 0 and x = pop(), then x was the last item pushed and remove x from stack
2. If nothing has be placed on the stack, then len() = 0
3. Calls to successful push should increment length by 1
4. Calls to successful (non-None) pop should decrement length by 1
5. Popping an empty stack returns None
6. push(None) results in ValueError exception

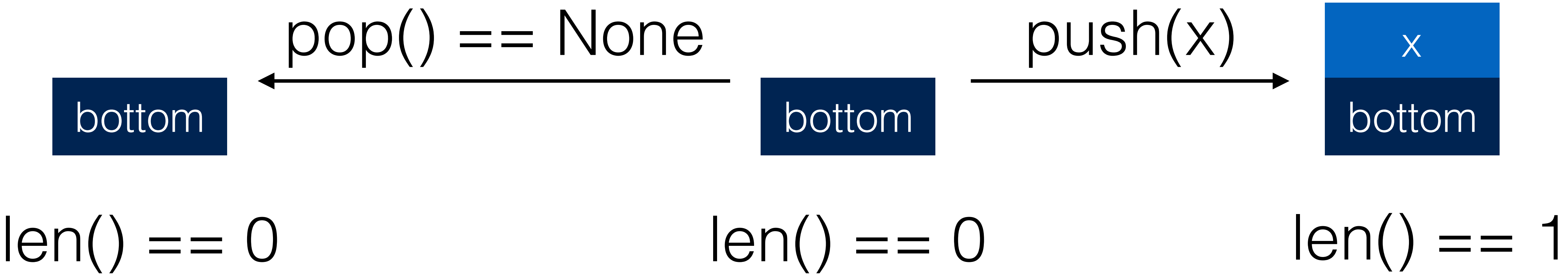
Unamended properties from class possibly with redundancy.

# Properties of a Stack*

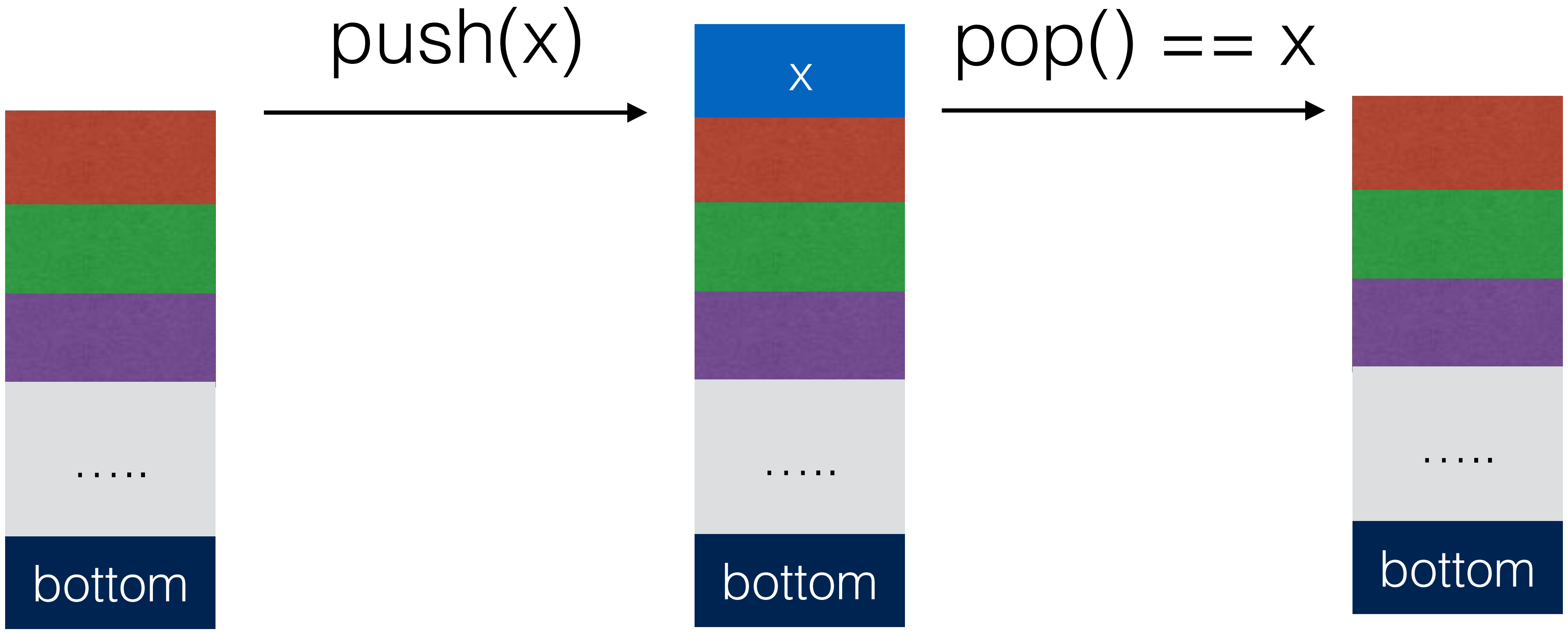
7. n number of successful pushes followed by n number of successful pops, values pushed should be observed in reverse order when popped
8. len should always return non-negative integer
9. Popping and empty stack doesn't change length
10. If len() > 0, then pop() return None
11. Length of stack equals to # of successful pushes - # of successful pops

Unamended properties from class possibly with redundancy.

# Discovering Properties involving Multiple Operations



# Discovering Properties involving Multiple Operations



# Discovering Properties involving Multiple Operations

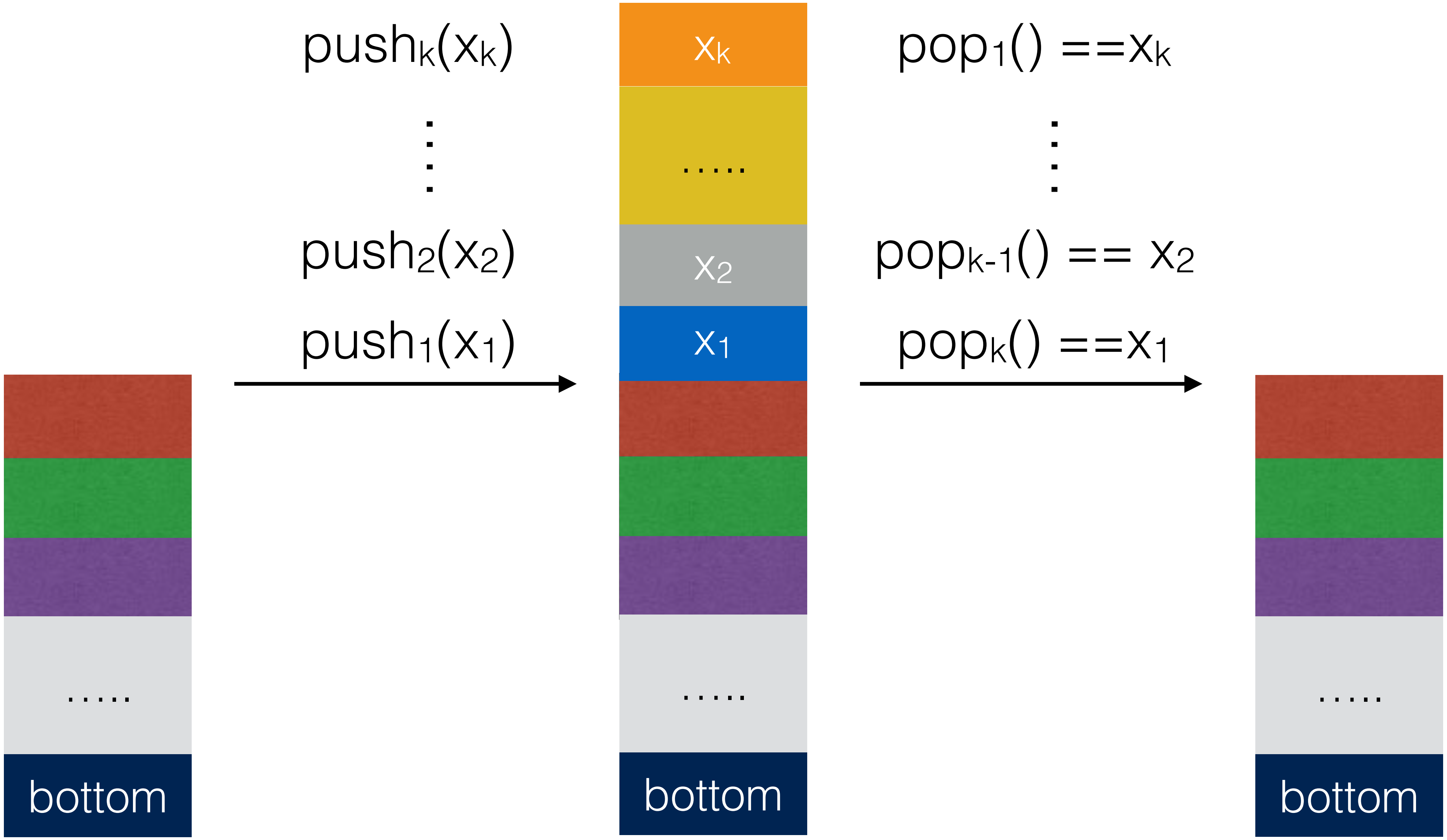
```
@given(st.integers())
def test_p1(x):
  s = Stack()
  s.push(x)
  assert s.pop() == x
```

This test does not completely capture the property — it only considers newly created Stacks.

```
@given(st.lists(st.integers()), st.integers())
def test_p1(y, x):
  s = Stack()
  for i in y:
    s.push(i)
  s.push(x)
  assert s.pop() == x
```

This test completely captures the property — it considers any stack.

# Discovering Properties involving Multiple Operations



$\text{push}_k(x_k)$

⋮

$\text{push}_2(x_2)$

$\text{push}_1(x_1)$

$x_k$

.....

$x_2$

$x_1$

.....

bottom

$\text{pop}_1() == x_k$

⋮

$\text{pop}_{k-1}() == x_2$

$\text{pop}_k() == x_1$

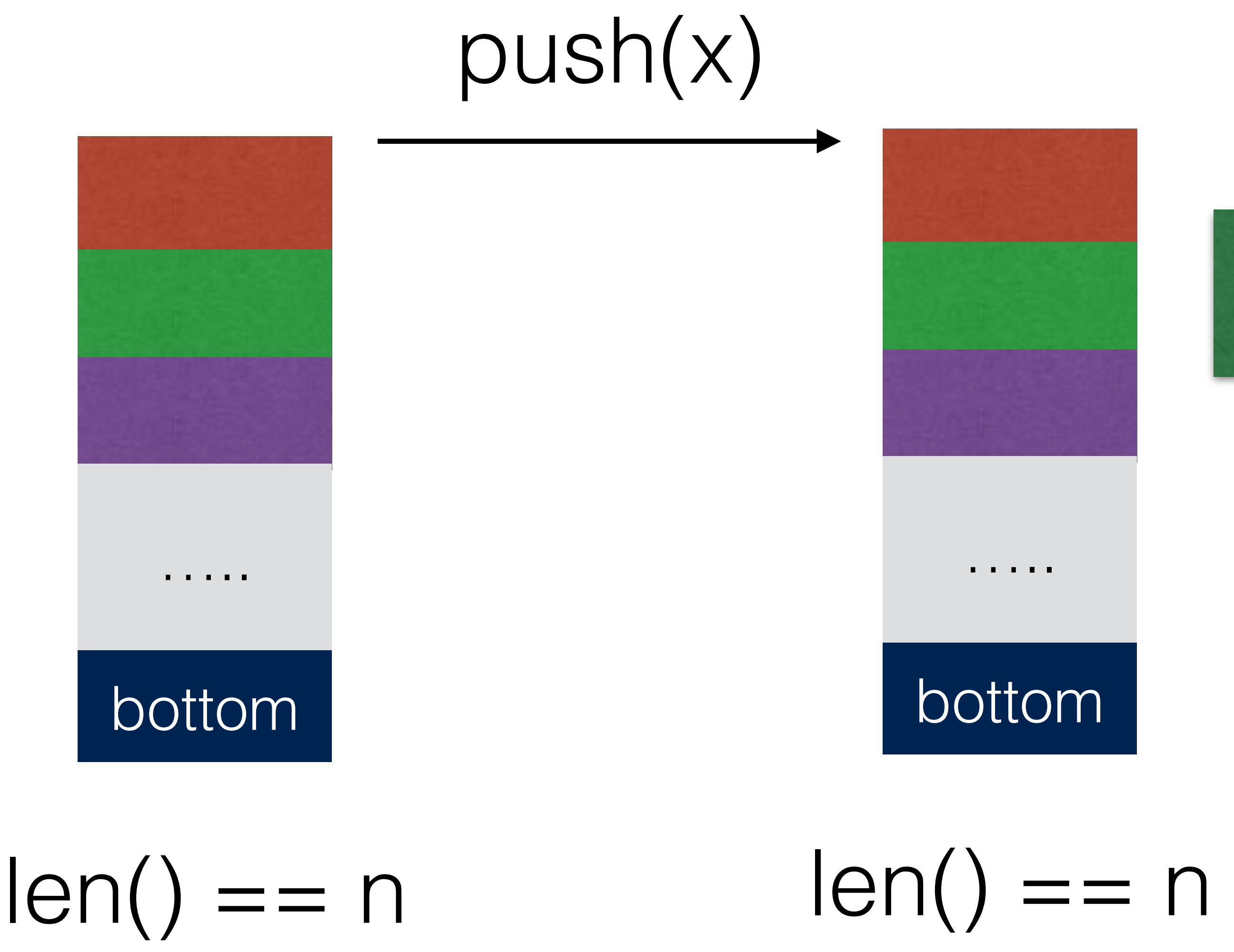
.....

bottom

.....

bottom

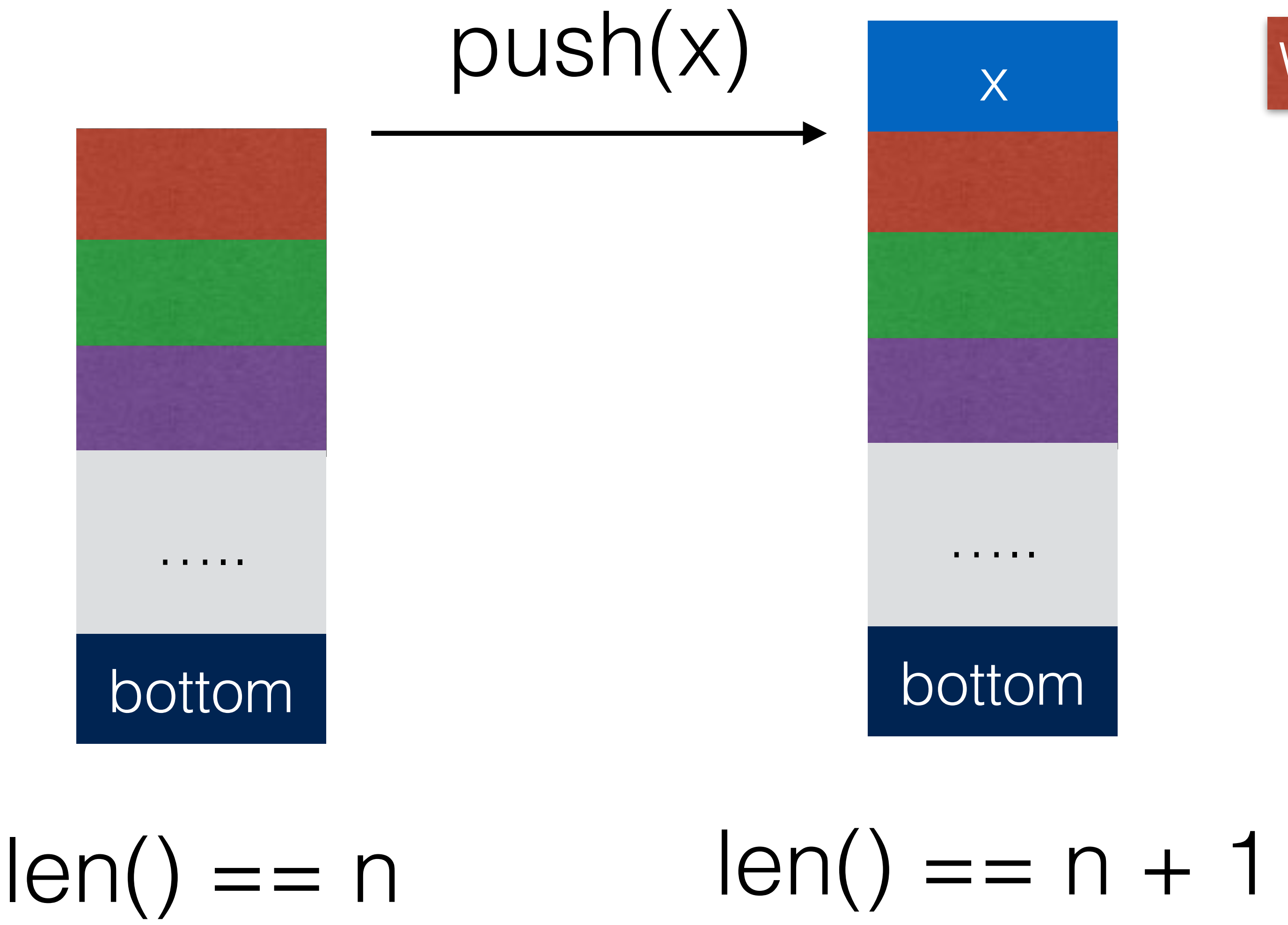
# Discovering Properties involving Multiple Operations



When x == None

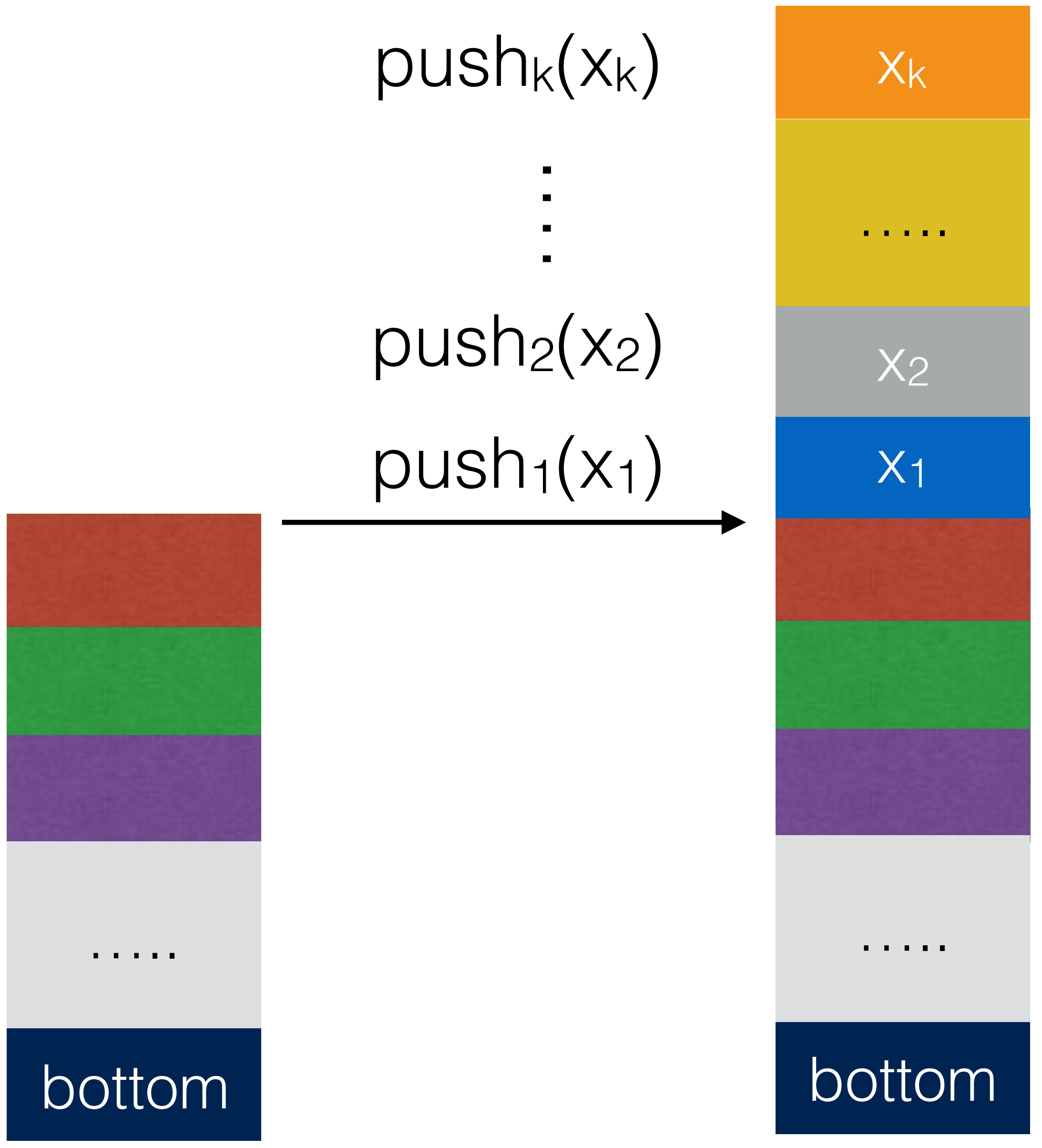
Results in ValueError Exception

# Discovering Properties involving Multiple Operations



When x != None

# Discovering Properties involving Multiple Operations

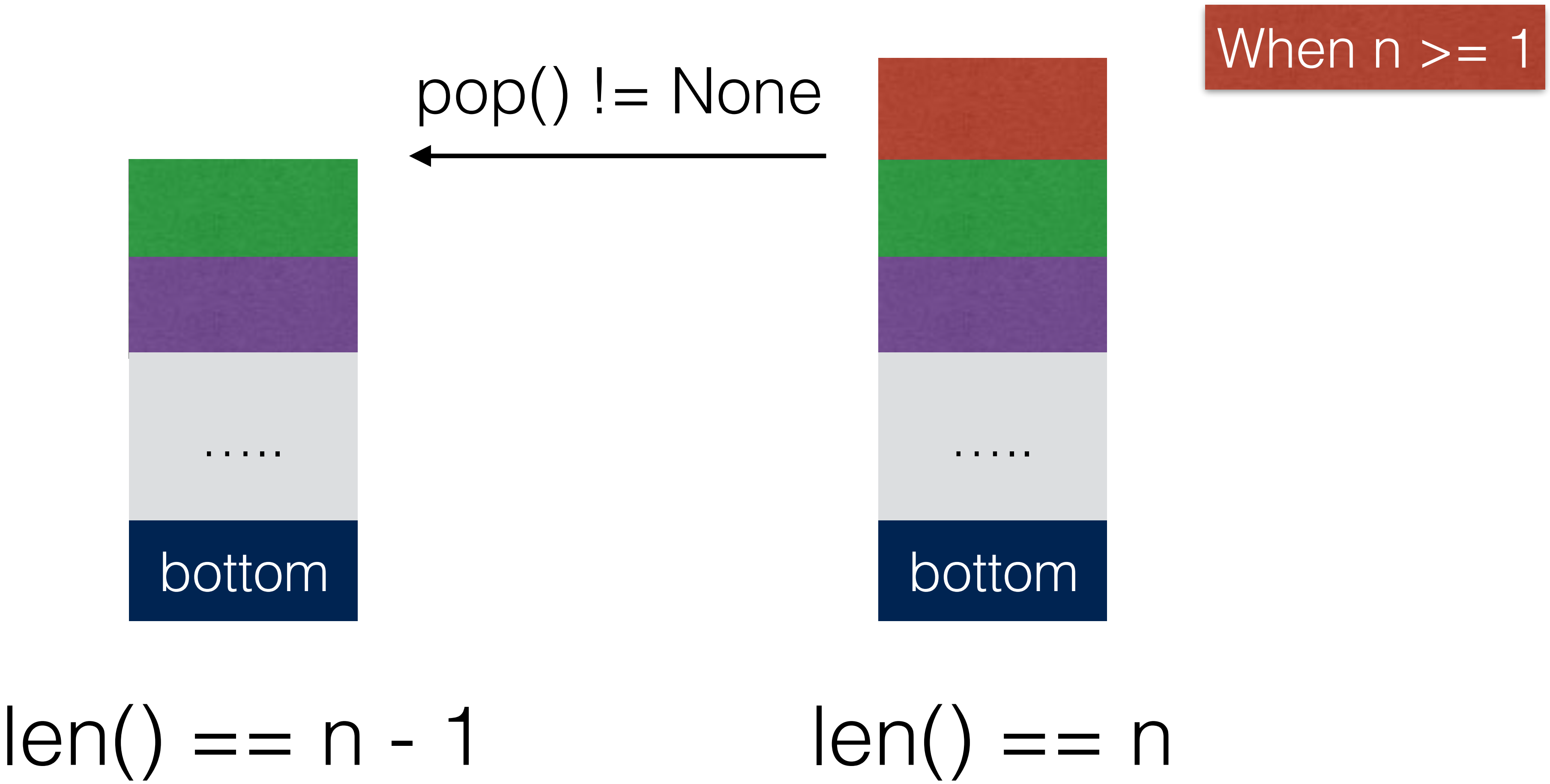


When $x_i$ != None

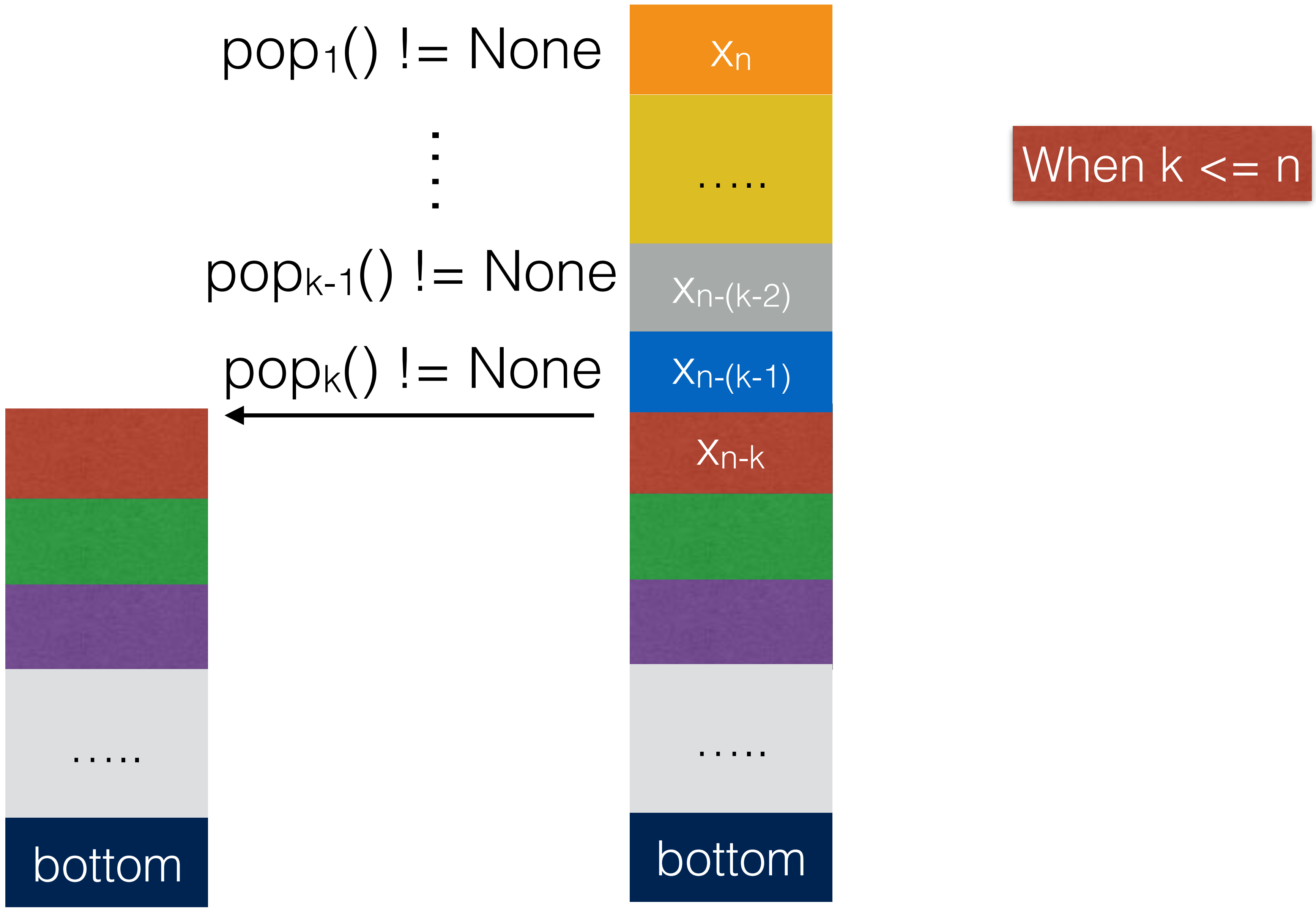
len() == n

len() == n + k

# Discovering Properties involving Multiple Operations



# Discovering Properties involving Multiple Operations



pop$_1$() != None

⋮

pop$_{k-1}$() != None

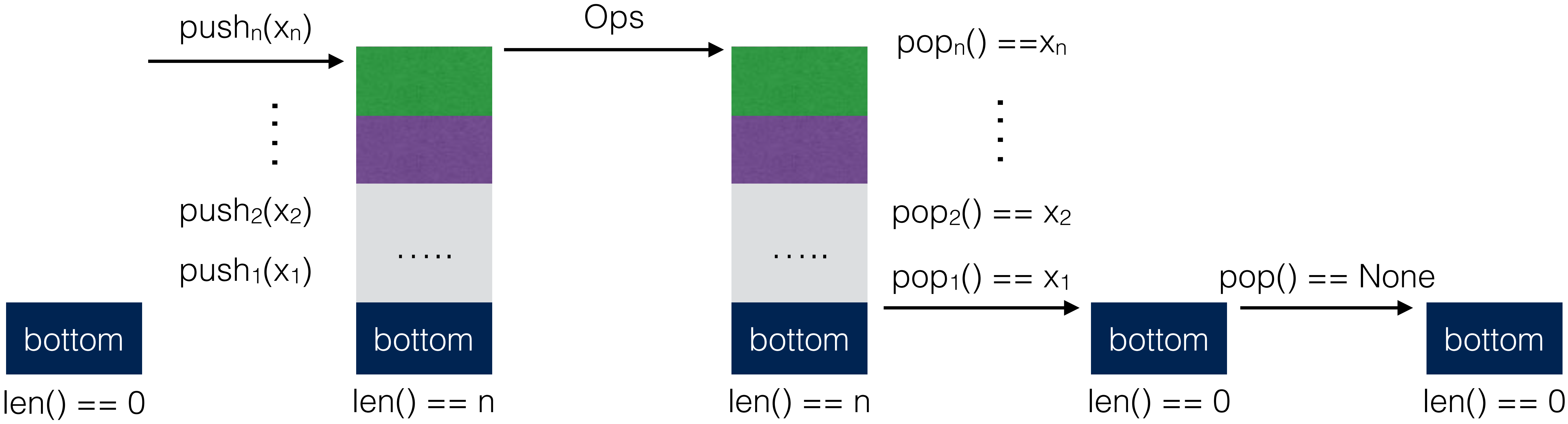
pop$_k$() != None

When k <= n

len() == n - k

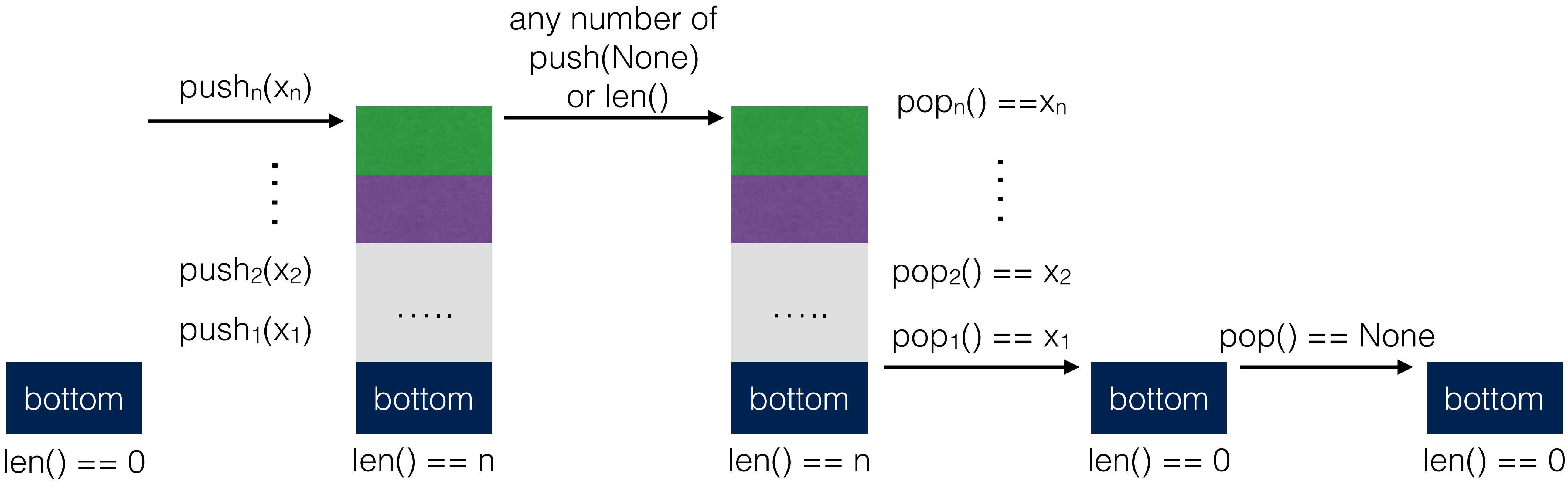
len() == n

# Discovering Properties involving Multiple Operations



- Ops is a sequence of push, pop, and len operations such that #push(y) in Ops == #pop(z) in Ops
- y != None
- z != None
- $x_i$ != None

# Discovering Properties involving Multiple Operations



$push_n(x_n)$

⋮

$push_2(x_2)$

$push_1(x_1)$

bottom

len() == 0

.....

bottom

len() == n

any number of push(None) or len()

.....

bottom

len() == n

$pop_n() == x_n$

⋮

$pop_2() == x_2$

$pop_1() == x_1$

bottom

len() == 0

pop() == None

bottom

len() == 0

When $x_i$ != None