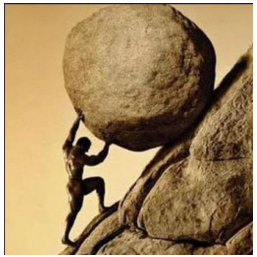
# PSO 4

## Tree Queue Heap

justin-zhang.com/teaching/CS251

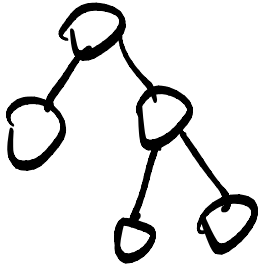
Project 1 due next week

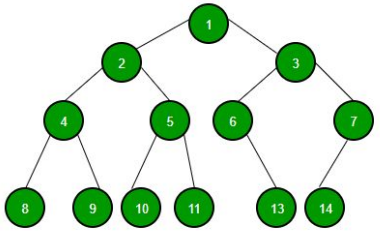
- See ed for updates



Hw 2 grades out by Friday

# Binary Trees






What is a full tree?

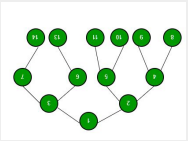
- each node is
  - a leaf
  - (internal node) with 2 children

What is a complete tree?

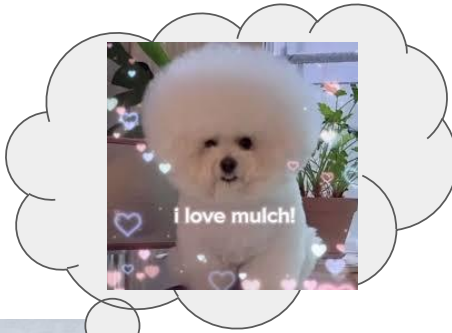
- every level except last is full
- last level - 'left leaning'




Actually that “tree” is upside down. It should look like



I know this because I touch grass




**(Binary Tree)**

(1) A full binary tree cannot have which of the following number of nodes?

A. 3
B. 7
C. 11
D. 12
E. 15

**(Binary Tree)**

(1) A full binary tree cannot have which of the following number of nodes?

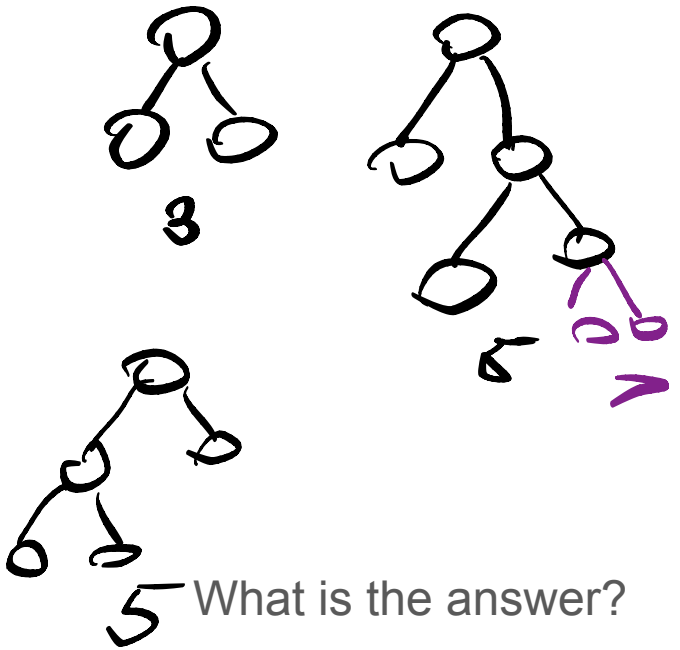
A. 3
B. 7
C. 11
D. 12 – even #
E. 15

Definition of a full binary tree

Every node is either a

- leaf or,
- inner node with two children

examples




What is the answer?

(2) Given the number of nodes $n = 7$, how many distinct shapes can a full binary tree have?

A. 3
B. 4
C. 5
D. 6
E. 7

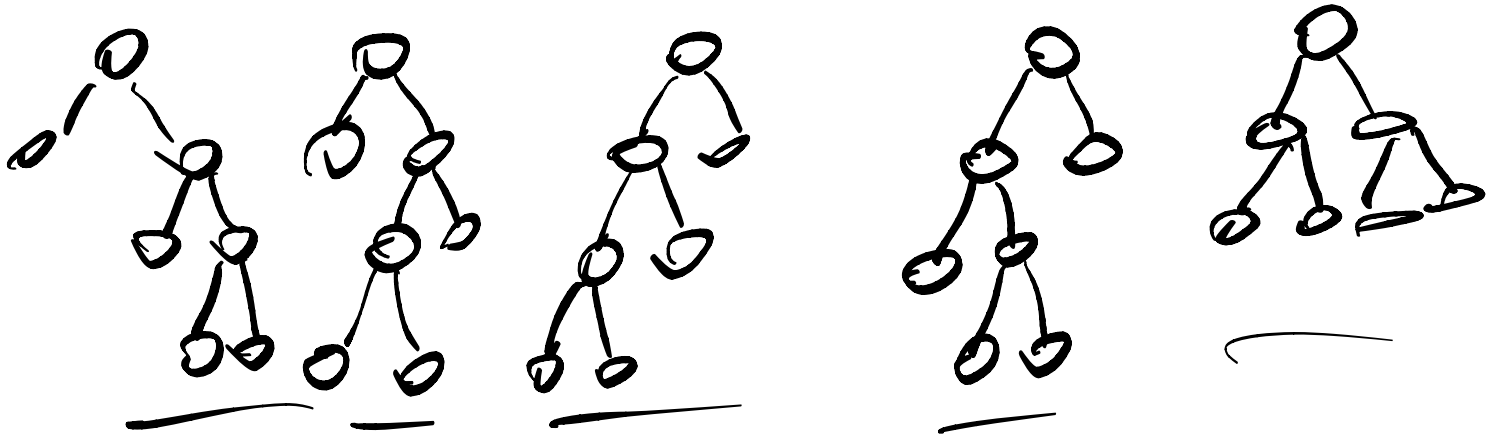
How to proceed?

(2) Given the number of nodes $n = 7$, how many distinct shapes can a full binary tree have?

A. 3
B. 4
C. 5
D. 6
E. 7

How to proceed?

Every answer is at most 7.. Just draw them all out!



(2) Given the number of nodes $n = 7$, how many distinct shapes can a full binary tree have?

A. 3
B. 4
C. 5
D. 6
E. 7

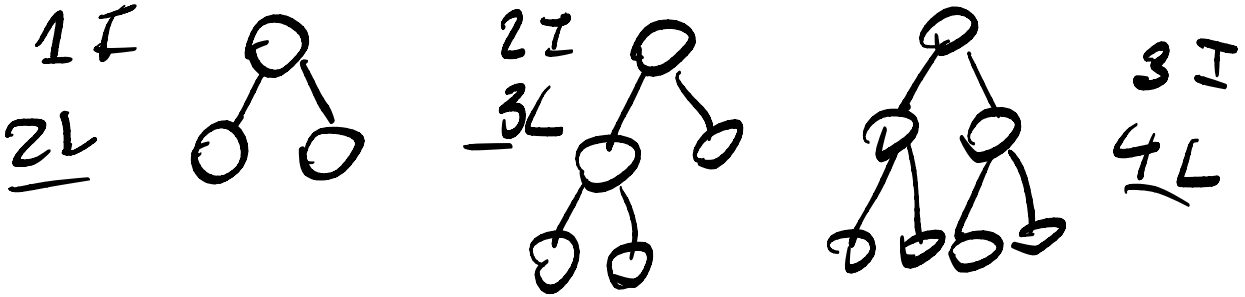
How to proceed?

Every answer is at most 7.. Just draw them all out!

(3) The number of leaf nodes is always greater than the number of internal nodes in a full binary tree.

A. True
B. False

Thoughts?




Exercise: Prove # leaves = # Internal + 1
in full binary trees.

(3) The number of leaf nodes is always greater than the number of internal nodes in a full binary tree.

A. True
B. False

If the thought isn't a strong 'yes' then draw examples

(4) The number of leaf nodes is always greater than the number of internal nodes in a complete binary tree.

A. True

B. False

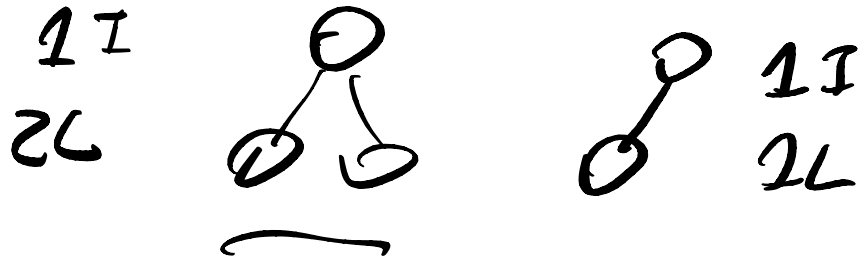
Definition of a *complete* binary tree?

(4) The number of leaf nodes is always greater than the number of internal nodes in a complete binary tree.

A. True
B. False

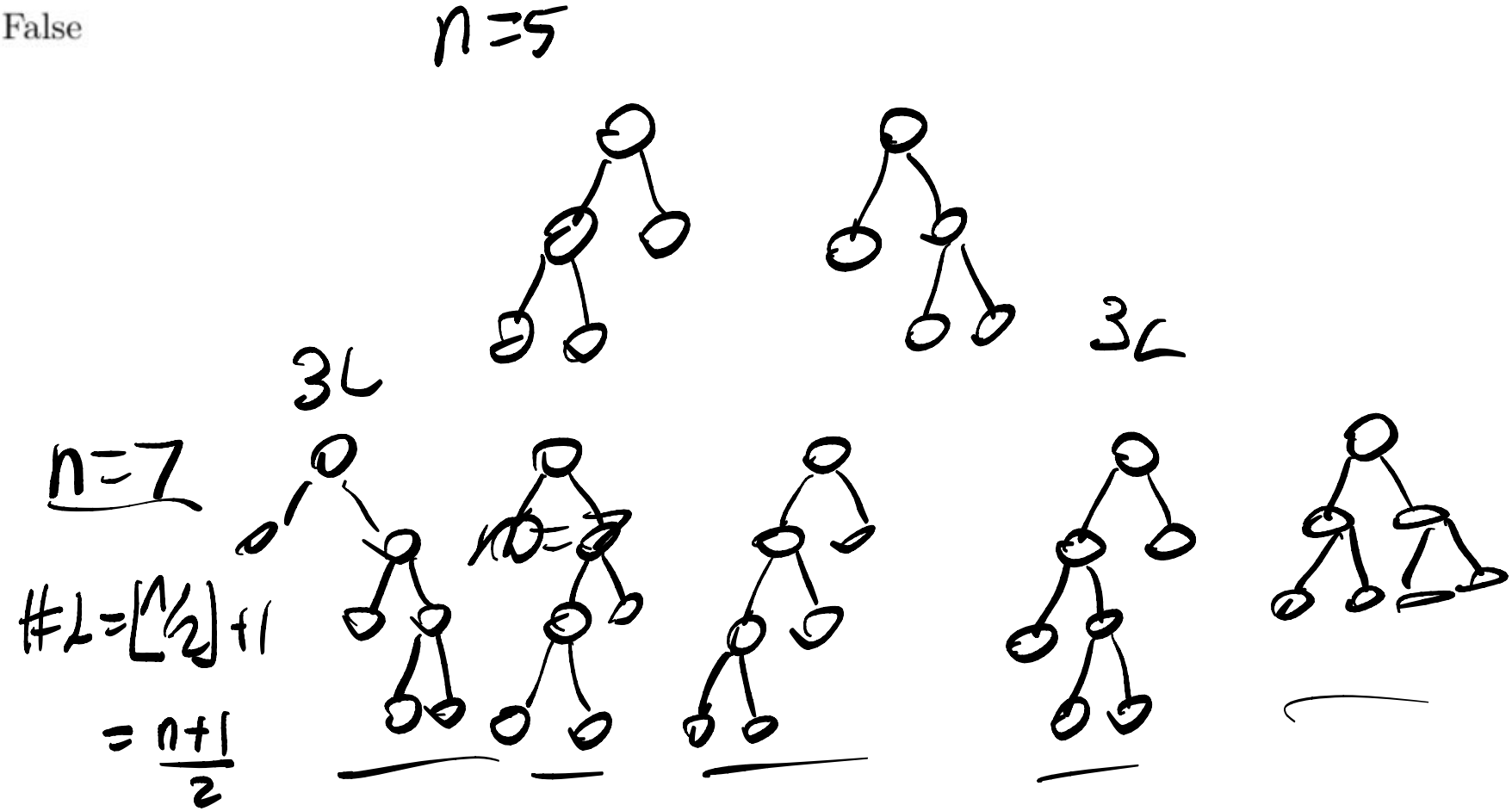
Definition of a *complete* binary tree?

- Every level of the tree except the last is complete
- Last level is *left-leaning*



(5) Given the number of nodes in a full binary tree, the number of its leaf nodes is determined.

A. True
B. False



**(Stack and Queue)**

Design a stack using two queues satisfying the following requirements

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.
3. Popping from the stack takes no more than $O(n)$ operations if performed after another pop, where $n$ is the number of elements in the data structure.

Push
Pop // O(1)

Pop
Pop // O(n)

Assume `Queue` interface

- `q = Queue.init()`
- `q.enq(x)`
- `x = q.deq()`
- `q.size()`

**(Stack and Queue)**

Design a stack using two queues satisfying the following requirements

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.
3. Popping from the stack takes no more than $O(n)$ operations if performed after another pop, where $n$ is the number of elements in the data structure.

Assume `Queue` interface

- `q = Queue.init()`
- `q.enq(x)`
- `x = q.deq()`
- `q.size()`

Implement `Stack` interface

- `s = Stack.init()`
- `s.push(x)`
- `x = s.pop()`

**(Stack and Queue)**

Design a stack using two queues satisfying the following requirements

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.
3. Popping from the stack takes no more than $O(n)$ operations if performed after another pop, where $n$ is the number of elements in the data structure.

Assume `Queue` interface

- `q = Queue.init()`
- `q.enq(x)`
- `x = q.deq()`
- `q.size()`

```
def Stack.init():
```

**(Stack and Queue)**

Design a stack using two queues satisfying the following requirements

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.
3. Popping from the stack takes no more than $O(n)$ operations if performed after another pop, where $n$ is the number of elements in the data structure.

Assume `Queue` interface

- `q = Queue.init()`
- `q.enq(x)`
- `x = q.deq()`
- `q.size()`

```
def Stack.init():
    q1 = Queue.init()
    q2 = Queue.init()
```

**(Stack and Queue)**

Design a stack using two queues satisfying the following requirements

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.
3. Popping from the stack takes no more than $O(n)$ operations if performed after another pop, where $n$ is the number of elements in the data structure.

Assume `Queue` interface

- `q = Queue.init()`
- `q.enq(x)`
- `x = q.deq()`
- `q.size()`

```
def Stack.init():
    q1 = Queue.init()
    q2 = Queue.init()
```

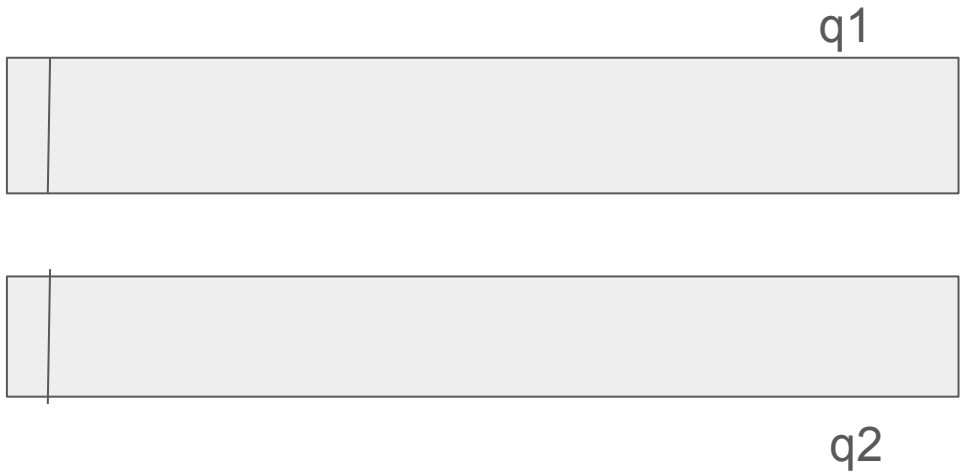
General Strat for these types of problems

- Fulfill conditions incrementally,
- When things break, fix them.
- *Occam's razor*

# Example: Starting with the Simplest Push Impl.

1. Pushing an element to the stack takes no more than $O(1)$ operations.

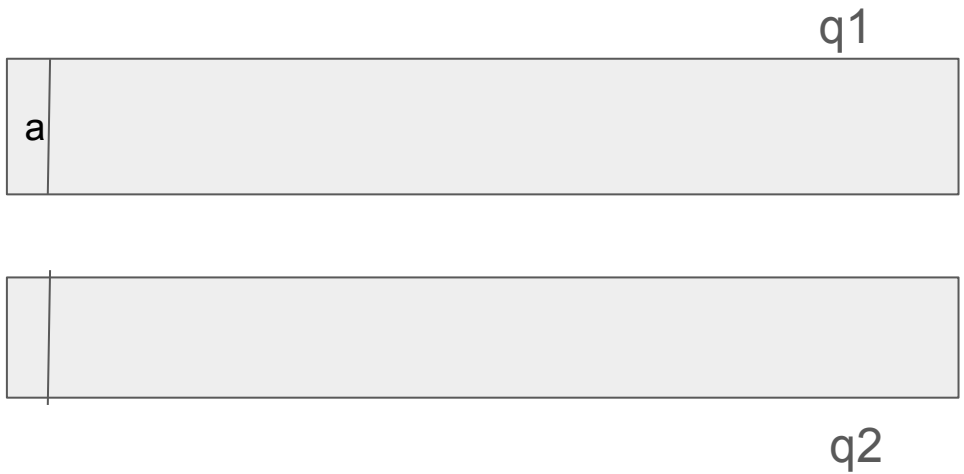
Push(a)
Push(b)
Push(c)
Push(d)



# Example

1. Pushing an element to the stack takes no more than $O(1)$ operations.

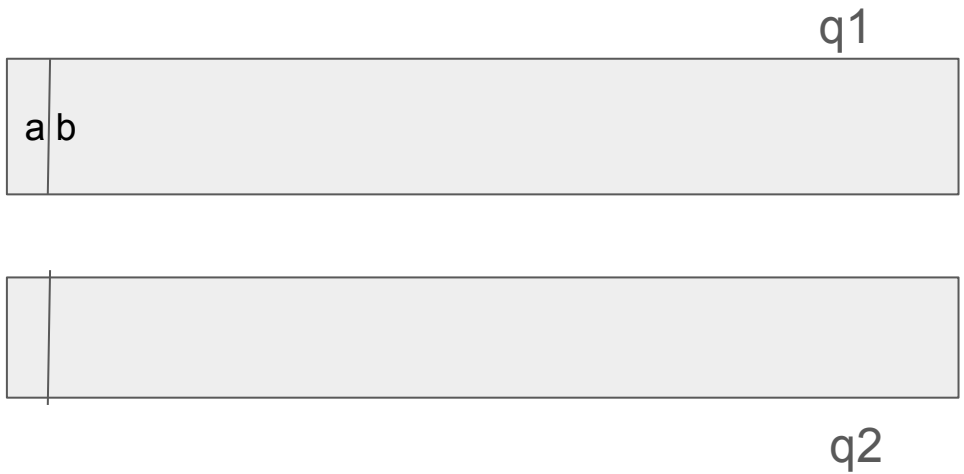
**Push(a)**
Push(b)
Push(c)
Push(d)



# Example

1. Pushing an element to the stack takes no more than $O(1)$ operations.

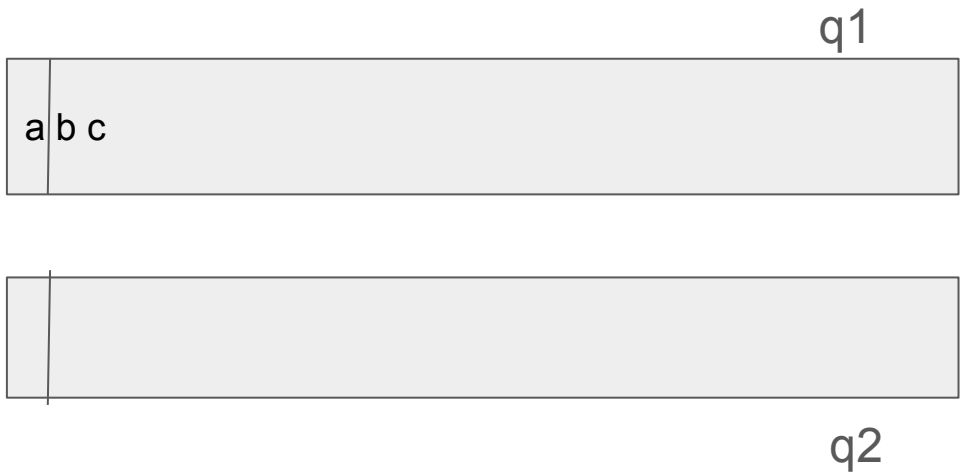
Push(a)
**Push(b)**
Push(c)
Push(d)



# Example

1. Pushing an element to the stack takes no more than $O(1)$ operations.

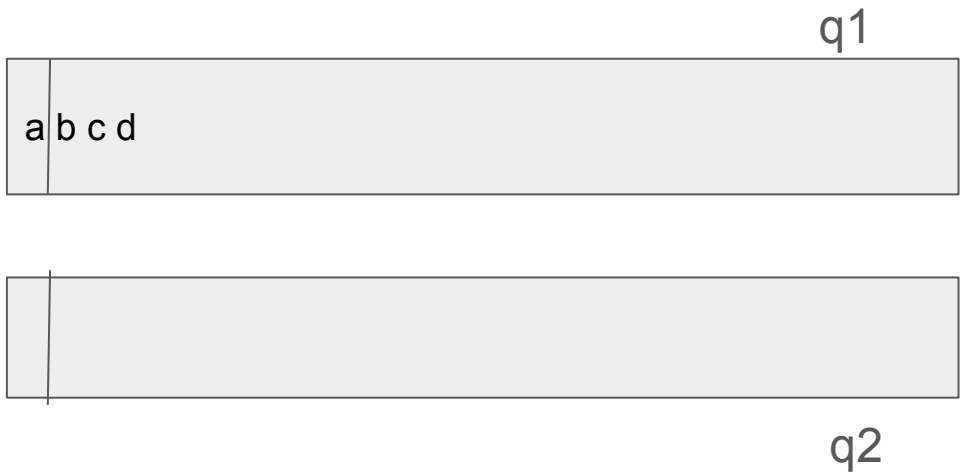
Push(a)
Push(b)
**Push(c)**
Push(d)



# Example

1. Pushing an element to the stack takes no more than $O(1)$ operations.

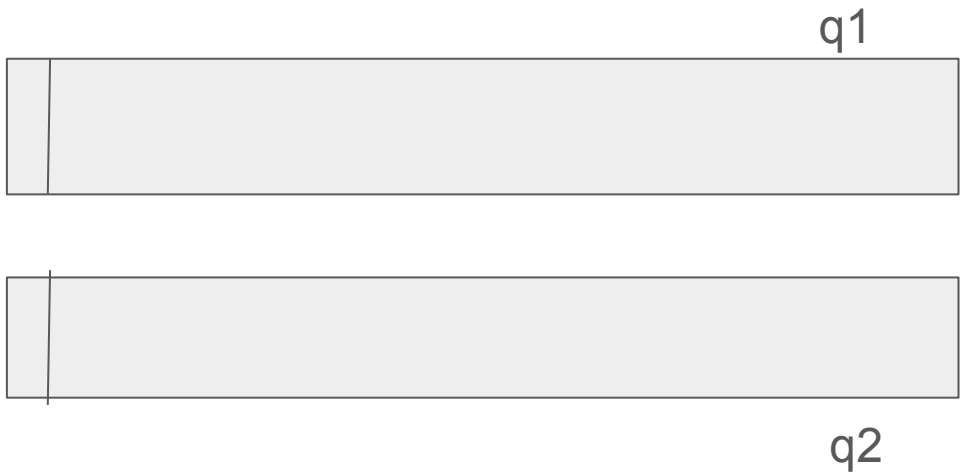
Push(a)
Push(b)
Push(c)
**Push(d)**



# Adding a Pop: Push, Pop?

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.

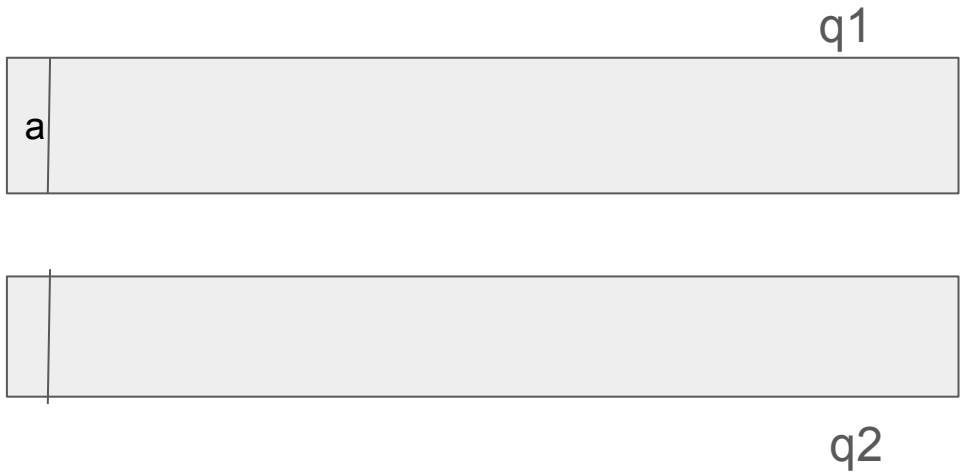
Push(a)
Push(b)
Pop() #should pop b
Push(c)
Pop() # should pop c



# Push, Pop?

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.

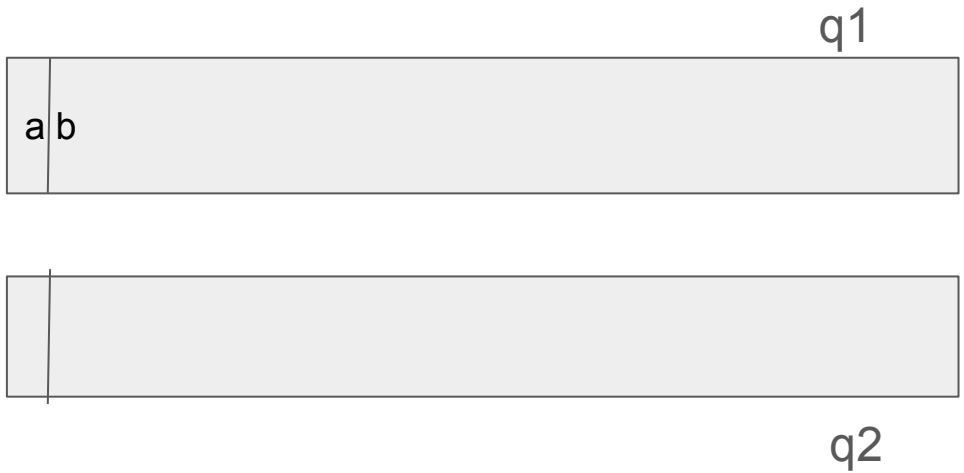
**Push(a)**
Push(b)
Pop() #should pop b
Push(c)
Pop() # should pop c



# Push, Pop? (use deq?)

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.

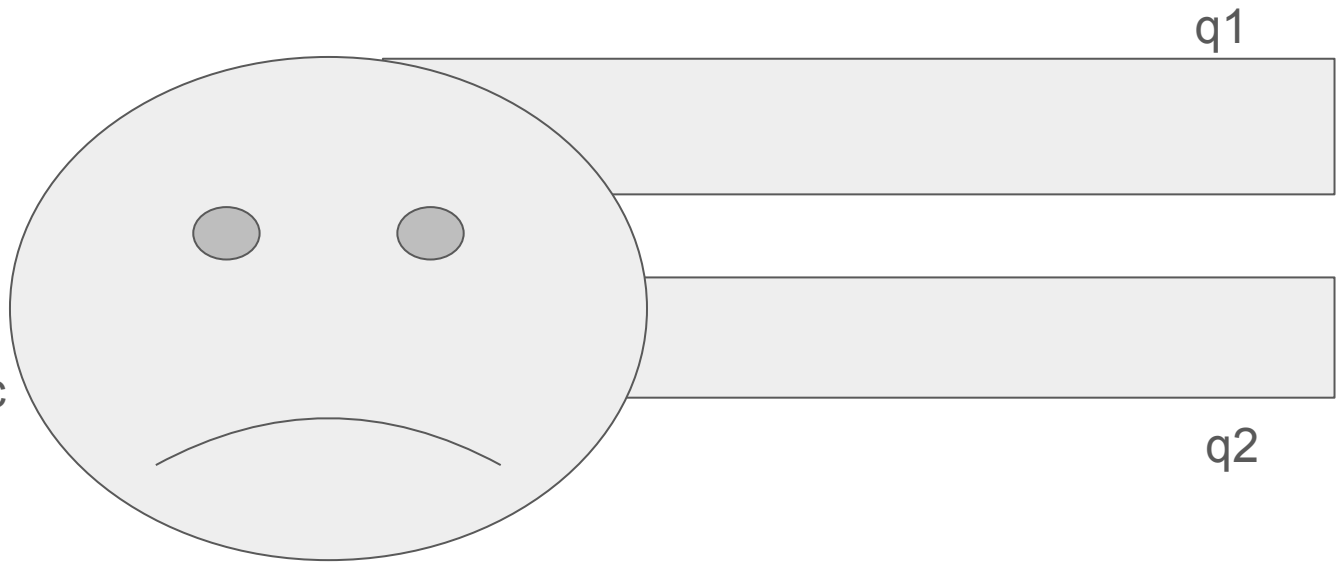
Push(a)
**Push(b)**
Pop() #should pop b
Push(c)
Pop() # should pop c



# Push, Pop?

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.

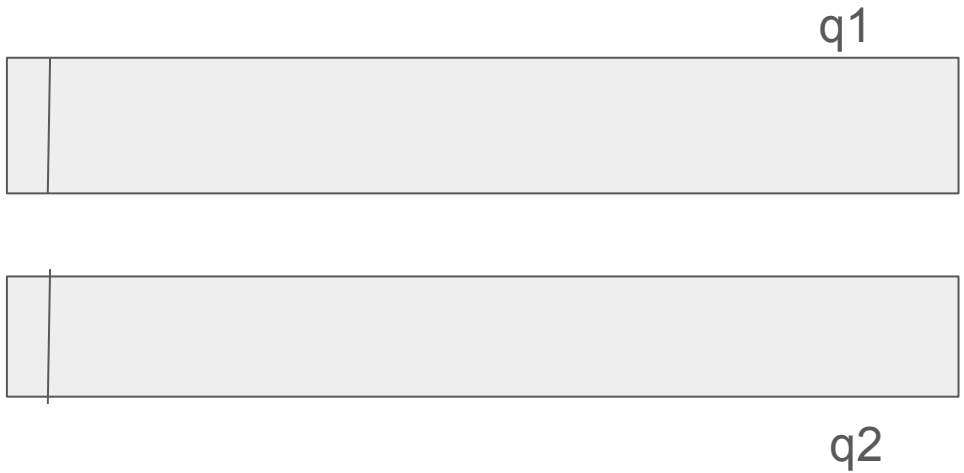
Push(a)
Push(b)
**Pop()** #should pop b
Push(c)
Pop() # should pop c



# Idea: use q2 to store “last element”

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.

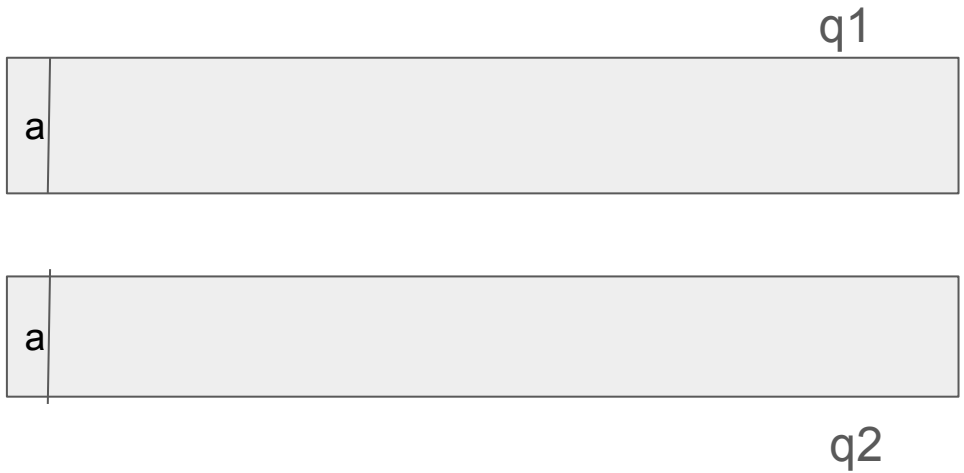
Push(a)
Push(b)
Pop() #should pop b
Push(c)
Pop() # should pop c



# Idea: use q2 to store “last element”

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.

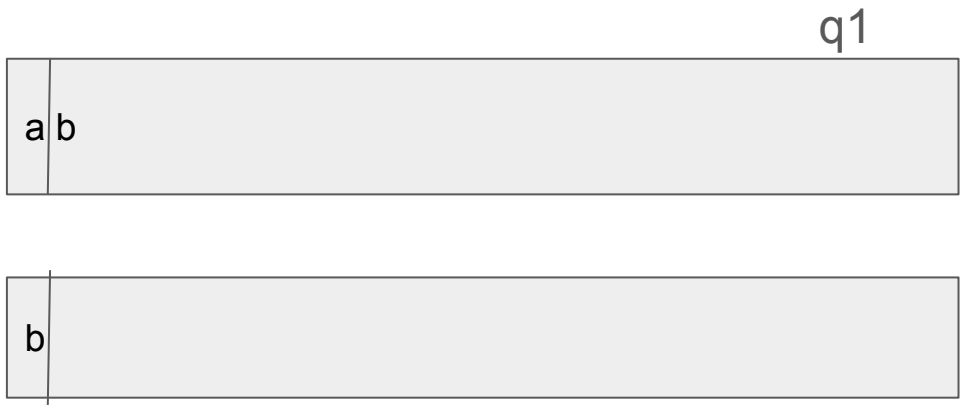
**Push(a)**
Push(b)
Pop() #should pop b
Push(c)
Pop() # should pop c



# Idea: use q2 to store “last element”

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.

Push(a)
**Push(b)**
Pop() #should pop b
Push(c)
Pop() # should pop c



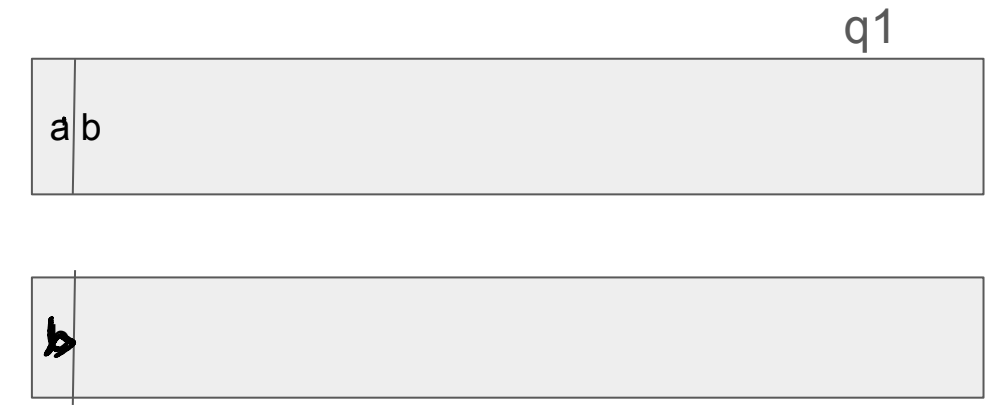

q2.deq() // discards a
q2.enq(b)

How to implement this?

# Idea: use q2 to store “last element”

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.

Push(a)
**Push(b)**
Pop() #should pop b
Push(c)
Pop() # should pop c

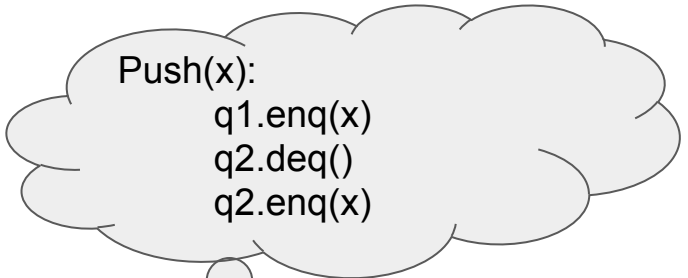


# Idea: use q2 to store “last element”

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.

Push(a)
Push(b)
**Pop()** #should pop b
Push(c)
Pop() # should pop c





# Idea: use q2 to store “last element”

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.

Push(a)
Push(b)
**Pop()** #should pop b
Push(c)
Pop() # should pop c


Push(x):
    q1.enq(x)
    q2.deq()
    q2.enq(x)

Pop():
    If q2.size() > 0:
        Return q2.deq()

# Pushing after a pop?

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.

Push(a)
Push(b)
**Pop()** #should pop b
Push(c)
Pop() # should pop c

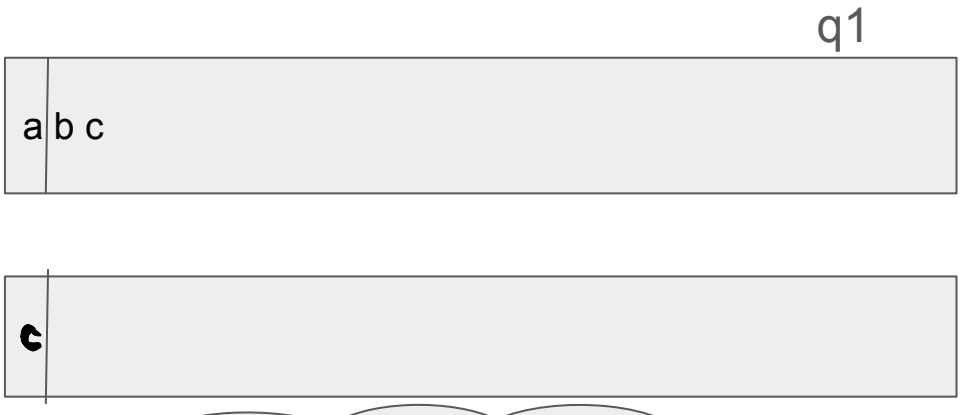

Push(x):
    q1.enq(x)
    q2.deq()
    q2.enq(x)

Pop():
    If q2.size() > 0:
        Return q2.deq()

# Pushing after a pop? Only pop if non-empty

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.

Push(a)
Push(b)
Pop() #should pop b
**Push(c)**
Pop() # should pop c



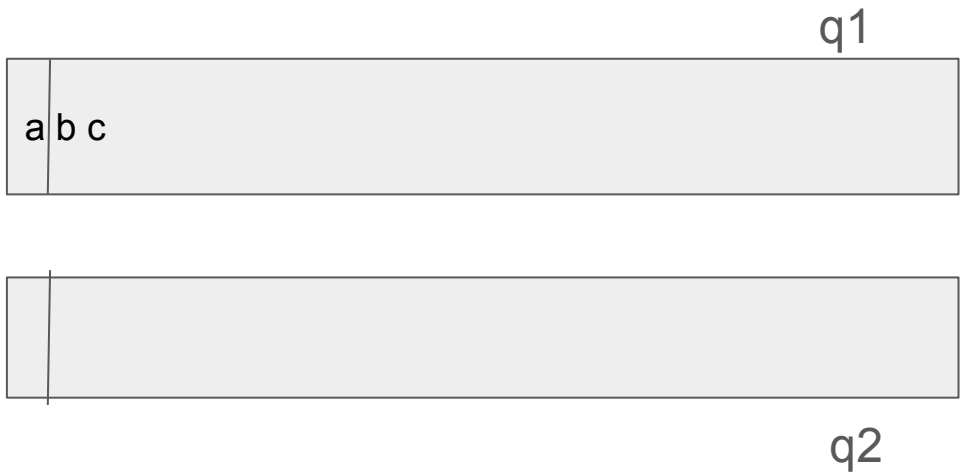

```
Push(x):
        q1.enq(x)
        if q2.size > 0: q2.deq()
        q2.enq(x)
```

```
():
    If q2.size() > 0:
        Return q2.deq()
```

# Idea: use q2 to store “last element”

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.

Push(a)
Push(b)
Pop() #should pop b
Push(c)
**Pop()** # should pop c

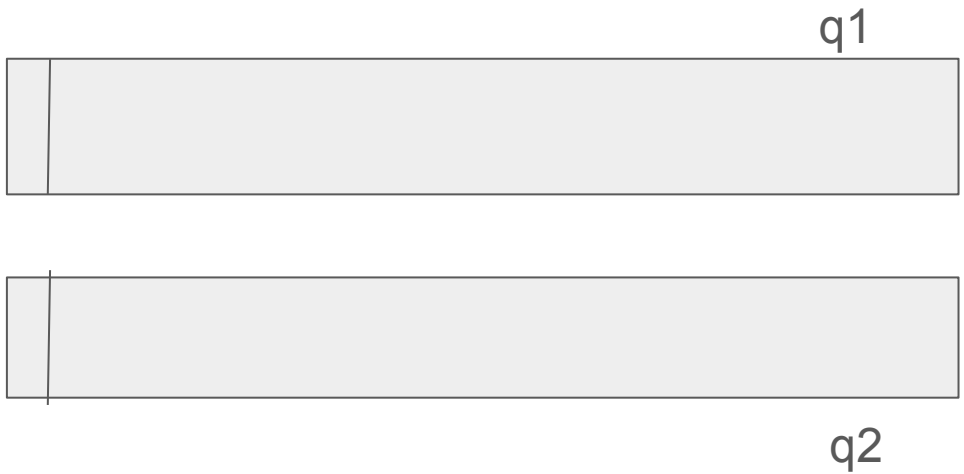



Not exactly a stack, but…
this stack impl is “correct” for the **first two** rules!

# Last requirement

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.
3. Popping from the stack takes no more than $O(n)$ operations if performed after another pop, where $n$ is the number of elements in the data structure.

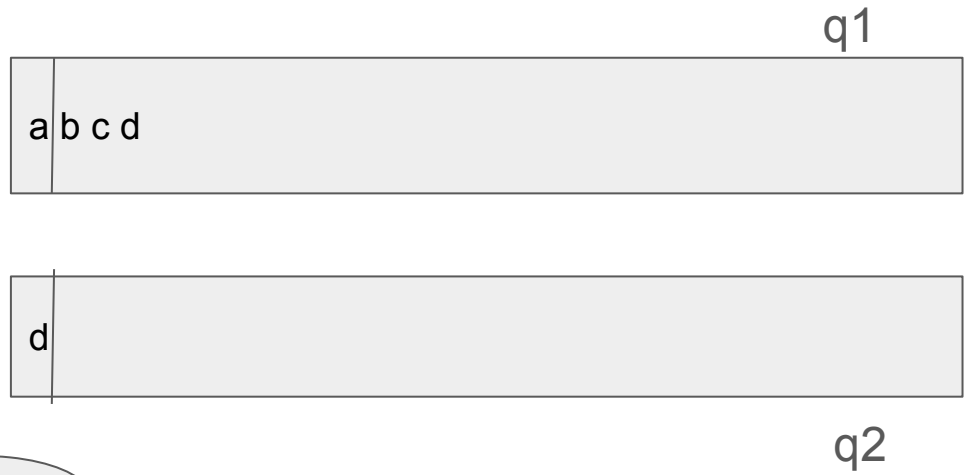
Push(a)
Push(b)
Push(c)
Push(d)
Pop() #should pop d
Pop() # should pop c




Try our implementation as-is

# Last requirement

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.
3. Popping from the stack takes no more than $O(n)$ operations if performed after another pop, where $n$ is the number of elements in the data structure.



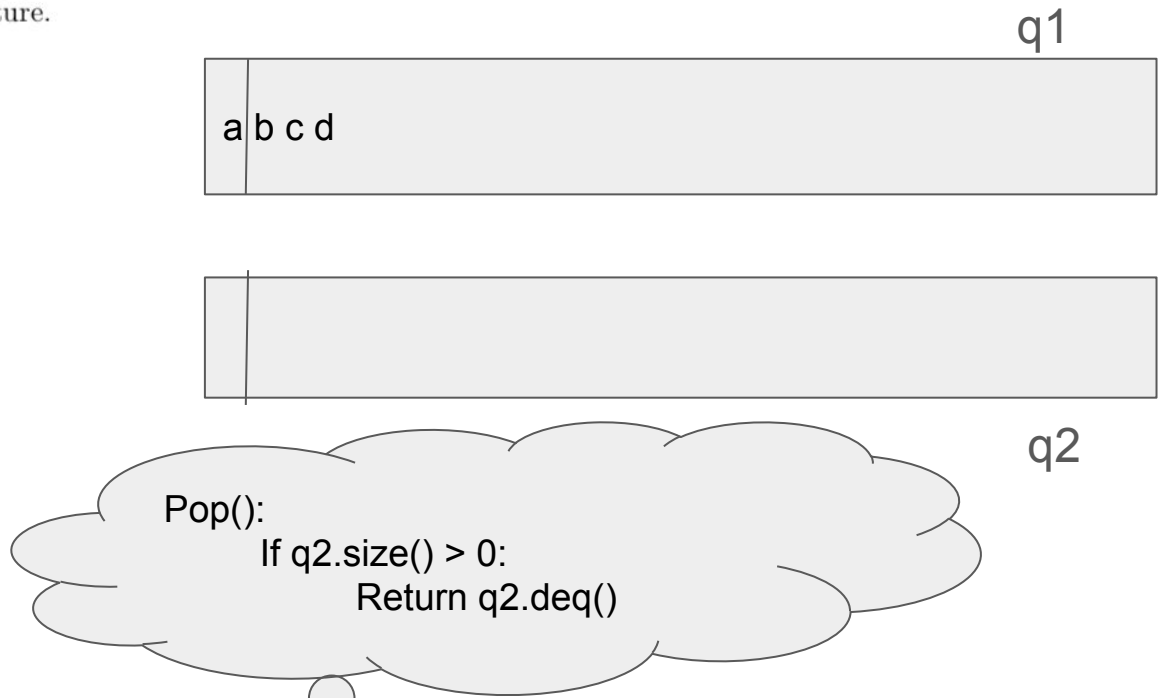

**Push(a)**
**Push(b)**
**Push(c)**
**Push(d)**
Pop() #should pop d
Pop() # should pop c

```
Push(x):
        q1.enq(x)
        if q2.size > 0: q2.deq()
        q2.enq(x)
```

# Last requirement

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.
3. Popping from the stack takes no more than $O(n)$ operations if performed after another pop, where $n$ is the number of elements in the data structure.

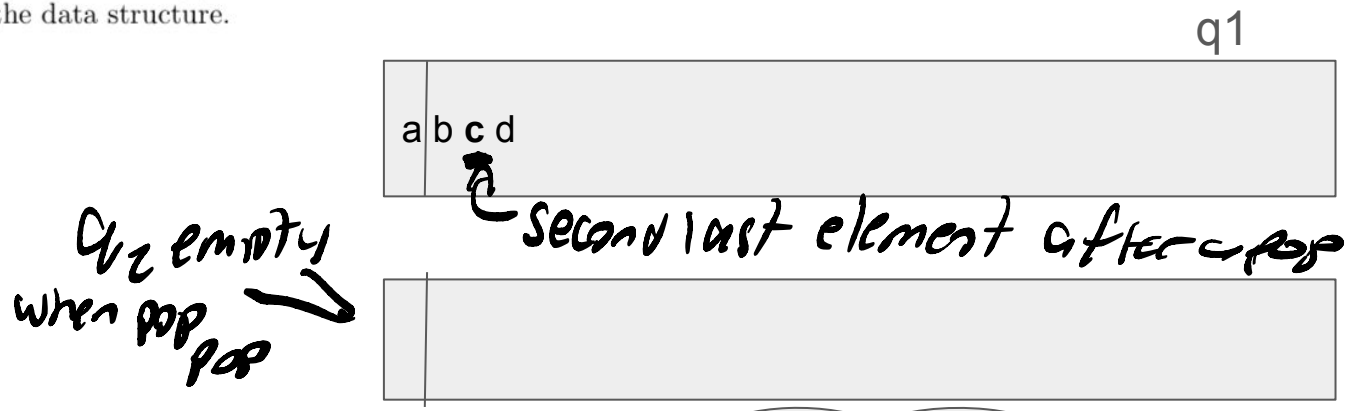
Push(a)
Push(b)
Push(c)
Push(d)
**Pop()** #should pop d
Pop() # should pop c



# Last requirement: How do we get **c**?

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.
3. Popping from the stack takes no more than $O(n)$ operations if performed after another pop, where $n$ is the number of elements in the data structure.

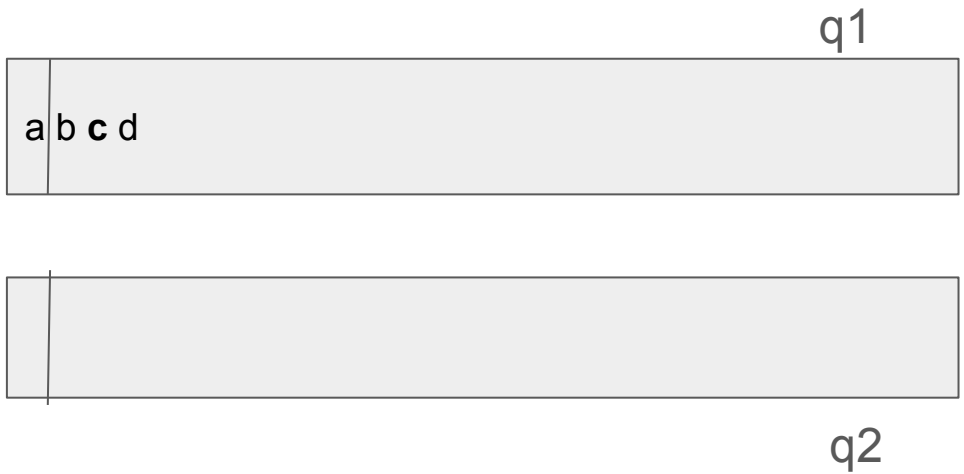
Push(a)
Push(b)
Push(c)
Push(d)
Pop() #should pop d
**Pop()** # should pop c



# Last requirement: How do we get **c**?

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.
3. Popping from the stack takes no more than $O(n)$ operations if performed after another pop, where $n$ is the number of elements in the data structure.

Push(a)
Push(b)
Push(c)
Push(d)
Pop() #should pop d
**Pop()** # should pop c



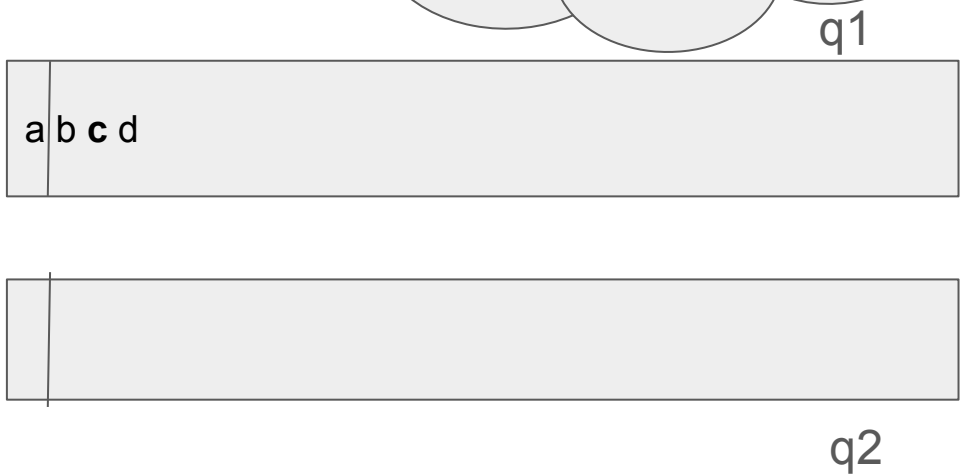

Idea: Deque everything from q1 into q2
Keep track of elements seen to get c

# Last requirement: How do we get **c**?

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.
3. Popping from the stack takes no more than $O(n)$ operations if performed after another pop, $n$ is the number of elements in the data structure.

```
while q1.size > 0:
        seen = q1.pop()
        q2.enq(seen)
        #how to get c?
```

Push(a)
Push(b)
Push(c)
Push(d)
Pop() #should pop d
**Pop()** # should pop c




Idea: Deque everything from q1 into q2
Keep track of elements seen to get c

# Last requirement: How do we get **c**?

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.
3. Popping from the stack takes no more than $O(n)$ operations if performed after another pop, $n$ is the number of elements in the data structure.

```
while q1.size > 0:
        seen = q1.pop()
        If q1.size() == 1:
            res = seen
            //…
        else:
            q2.enq(seen)
```

q1

a b **c** d

when removing c,
q1.size = 1

q2

Push(a)
Push(b)
Push(c)
Push(d)
Pop() #should pop d
**Pop()** # should pop c

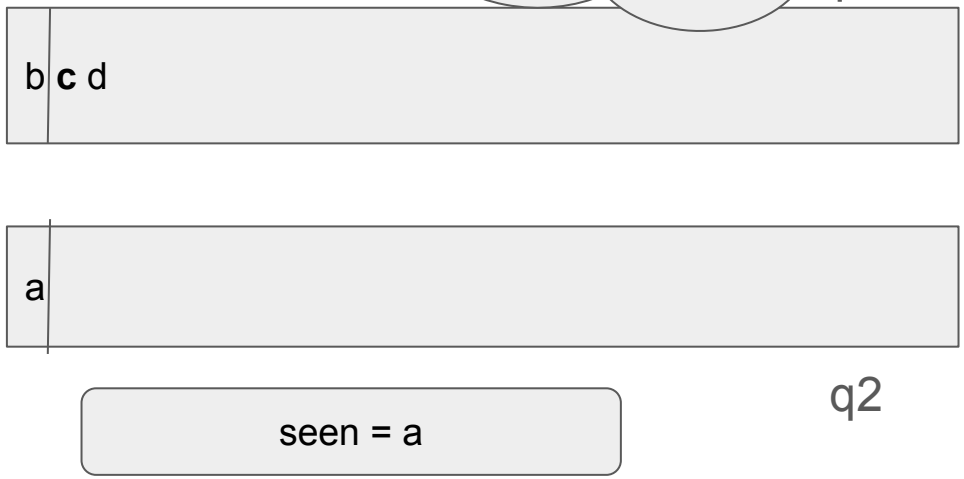
Idea: Deque everything from q1 into q2
Keep track of elements seen to get c

# Last requirement: How do we get **c**?

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.
3. Popping from the stack takes no more than $O(n)$ operations if performed after another po
   $n$ is the number of elements in the data structure.

```
while q1.size > 0:
        seen = q1.pop()
        If q1.size() == 1:
            res = seen
            //…
        else:
            q2.enq(seen)
```

Push(a)
Push(b)
Push(c)
Push(d)
Pop() #should pop d
**Pop()** # should pop c



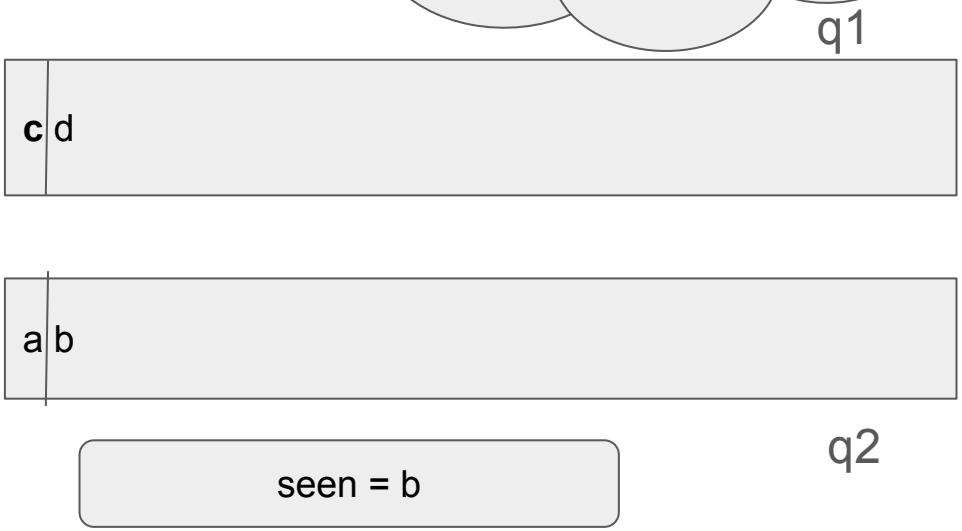

Idea: Deque everything from q1 into q2
Keep track of elements seen to get c

# Last requirement: How do we get **c**?

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.
3. Popping from the stack takes no more than $O(n)$ operations if performed after another pop, $n$ is the number of elements in the data structure.

```
while q1.size > 0:
        seen = q1.pop()
        If q1.size() == 1:
            res = seen
            //…
        else:
            q2.enq(seen)
```

Push(a)
Push(b)
Push(c)
Push(d)
Pop() #should pop d
**Pop()** # should pop c



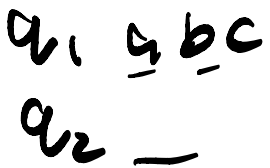

Idea: Deque everything from q1 into q2
Keep track of elements seen to get c

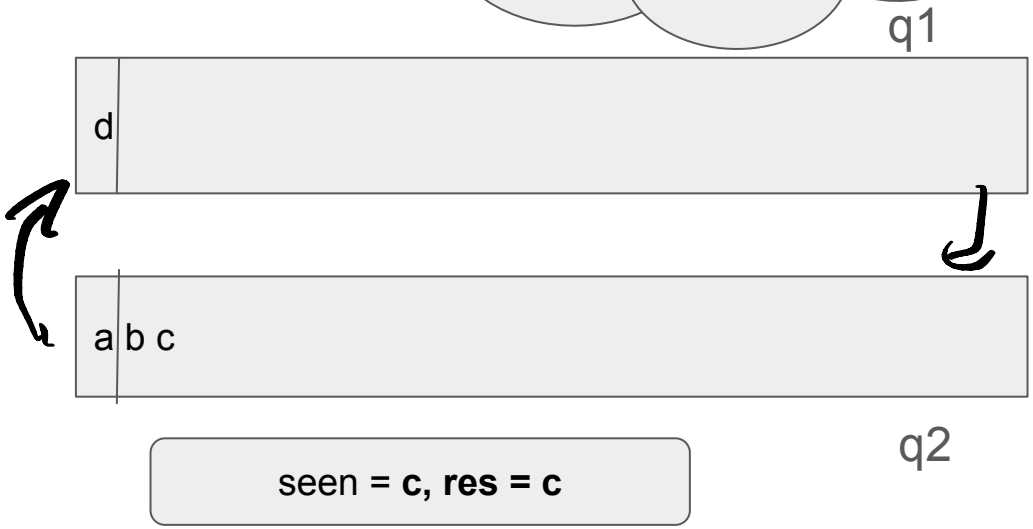
# Last requirement: How do we get **c**?

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.
3. Popping from the stack takes no more than $O(n)$ operations if performed after another pop, $n$ is the number of elements in the data structure.

```
while q1.size > 0:
        seen = q1.pop()
        If q1.size() == 1:
            res = seen
        else:
            q2.enq(seen)
```

Push(a)
Push(b)
Push(c)
Push(d)
Pop() #should pop d
**Pop()** # should pop c




Ok we got **c**, but the queues are messy!
How can we bring it back to prior state?

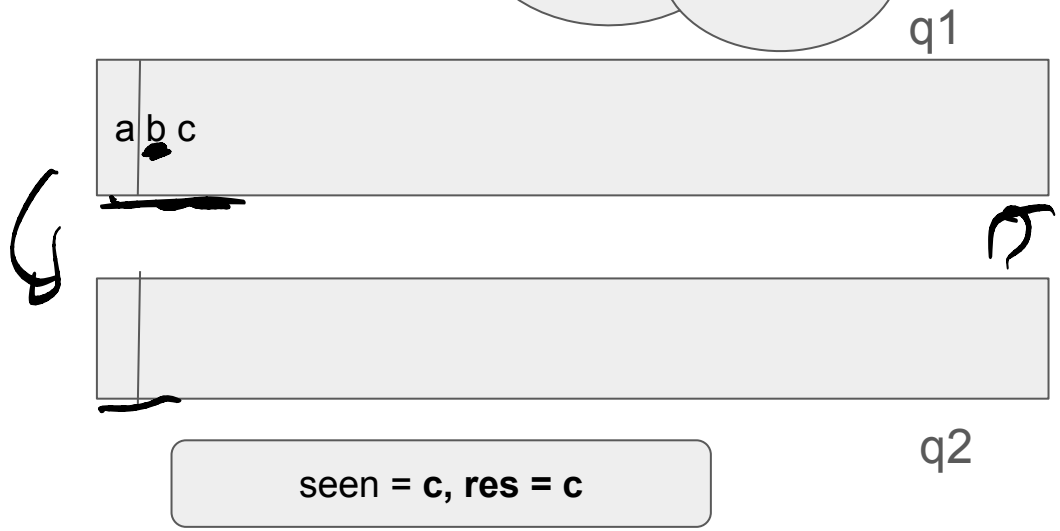
# Last requirement: How do we get **c**?

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.
3. Popping from the stack takes no more than $O(n)$ operations if performed after another pop, w
   $n$ is the number of elements in the data structure.

```
while q1.size > 0:
        seen = q1.pop()
        If q1.size() == 1:
            res = seen
            q2.enq(seen)
            q1 = q2, reinit q2
        else:
            q2.enq(seen)
```

Push(a)
Push(b)
Push(c)
Push(d)
Pop() #should pop d
**Pop()** # should pop c

Pop() #should get b ✓



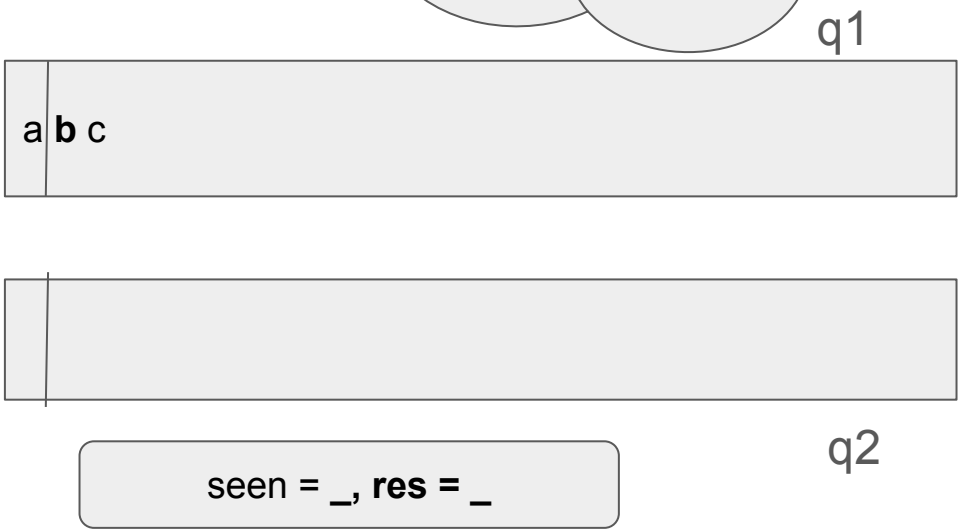

Ok we got **c**, but the queues are messy!
How can we bring it back to prior state?

# Last requirement: How do we get **c**?

1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.
3. Popping from the stack takes no more than $O(n)$ operations if performed after another pop, w
   $n$ is the number of elements in the data structure.

```
while q1.size > 0:
        seen = q1.pop()
        If q1.size() == 1:
            res = seen
            q2.enq(seen)
            q1 = q2, reinit q2
        else:
            q2.enq(seen)
```

Push(a)
Push(b)
Push(c)
Push(d)
Pop() #should pop d
Pop() # should pop c
**Pop() # should be b**




**Now if we pop again, the queues are in the correct state!**
**How do we show this always works?**

# Philosophy of Data Structures: Culling Chaos

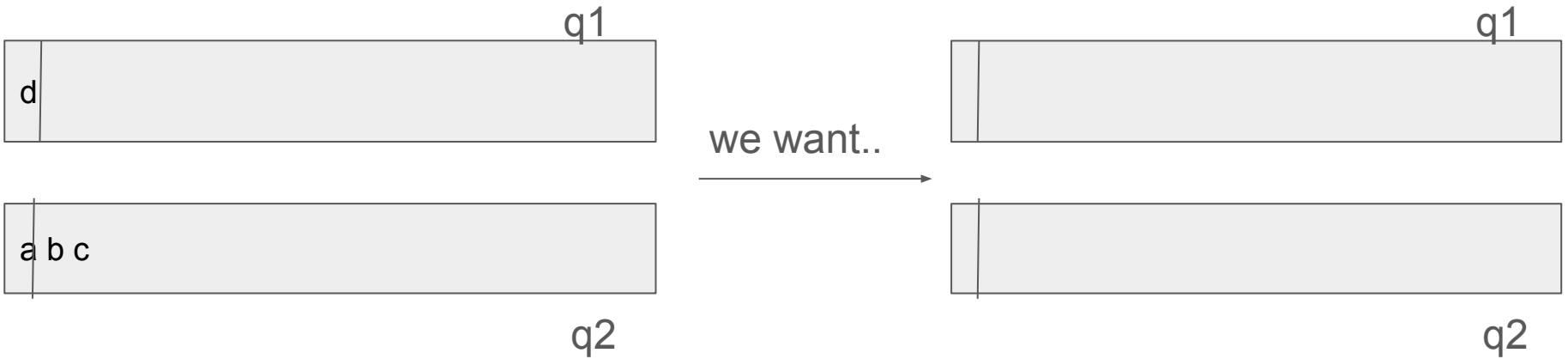
Sure fire design philosophy of data structures is **maintaining Invariants**

If I can make sure my data structures always look the same then easy to…

- Satisfy time efficiencies
- Write elegant pseudocode
- Prove/guarantee your impl. is efficient/correct

Example?

# Invariant for our stack? After pop pop

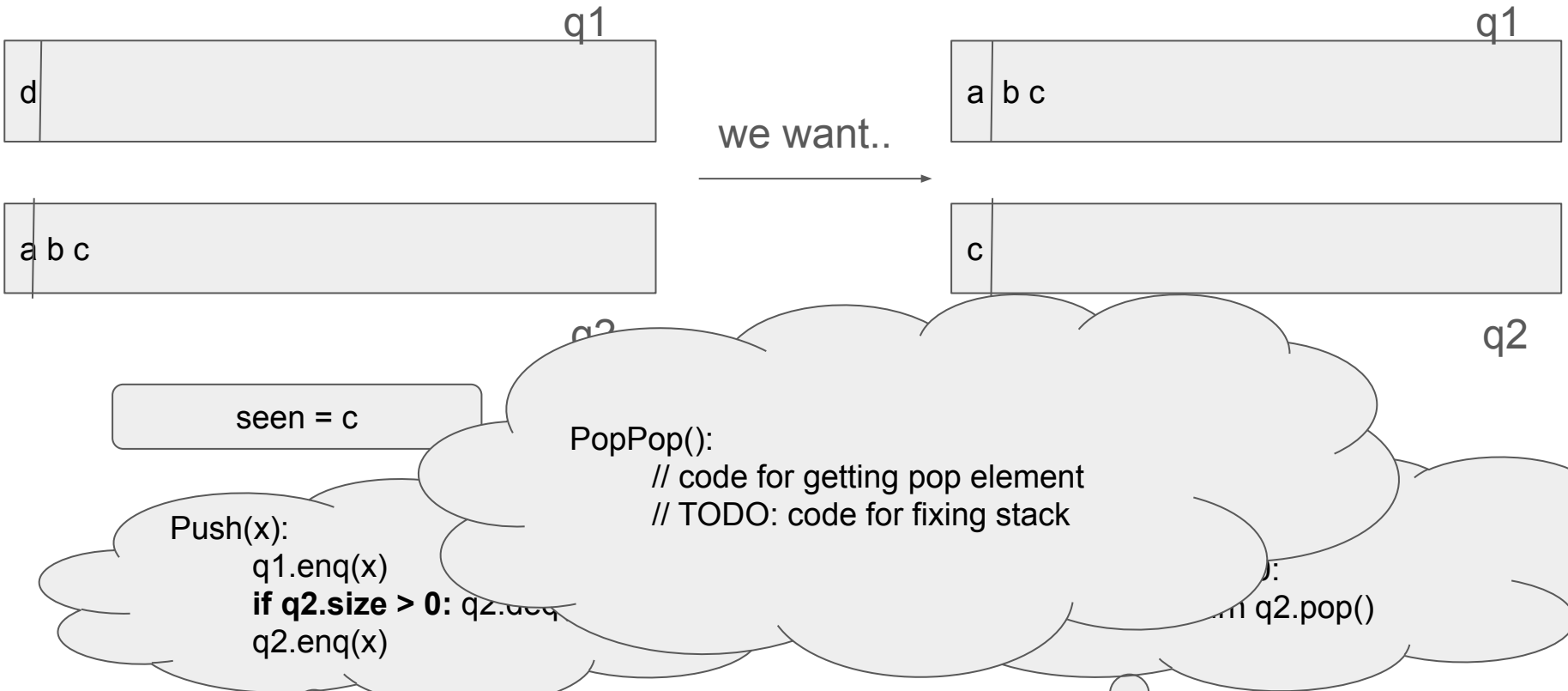



seen = c

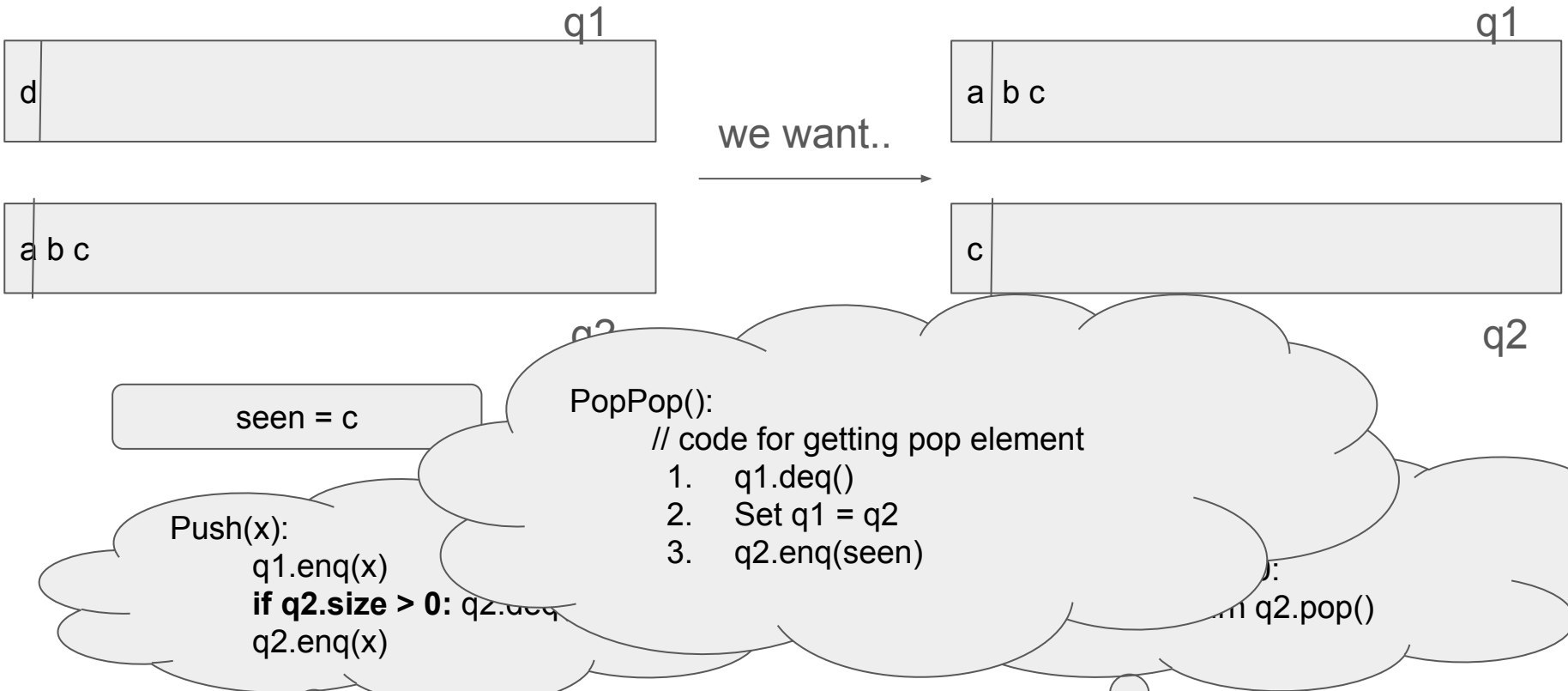
```
Push(x):
        q1.enq(x)
        if q2.size > 0: q2.deq()
        q2.enq(x)
```

```
PushPop():
        If q2.size() > 0:
                Return q2.pop()
```

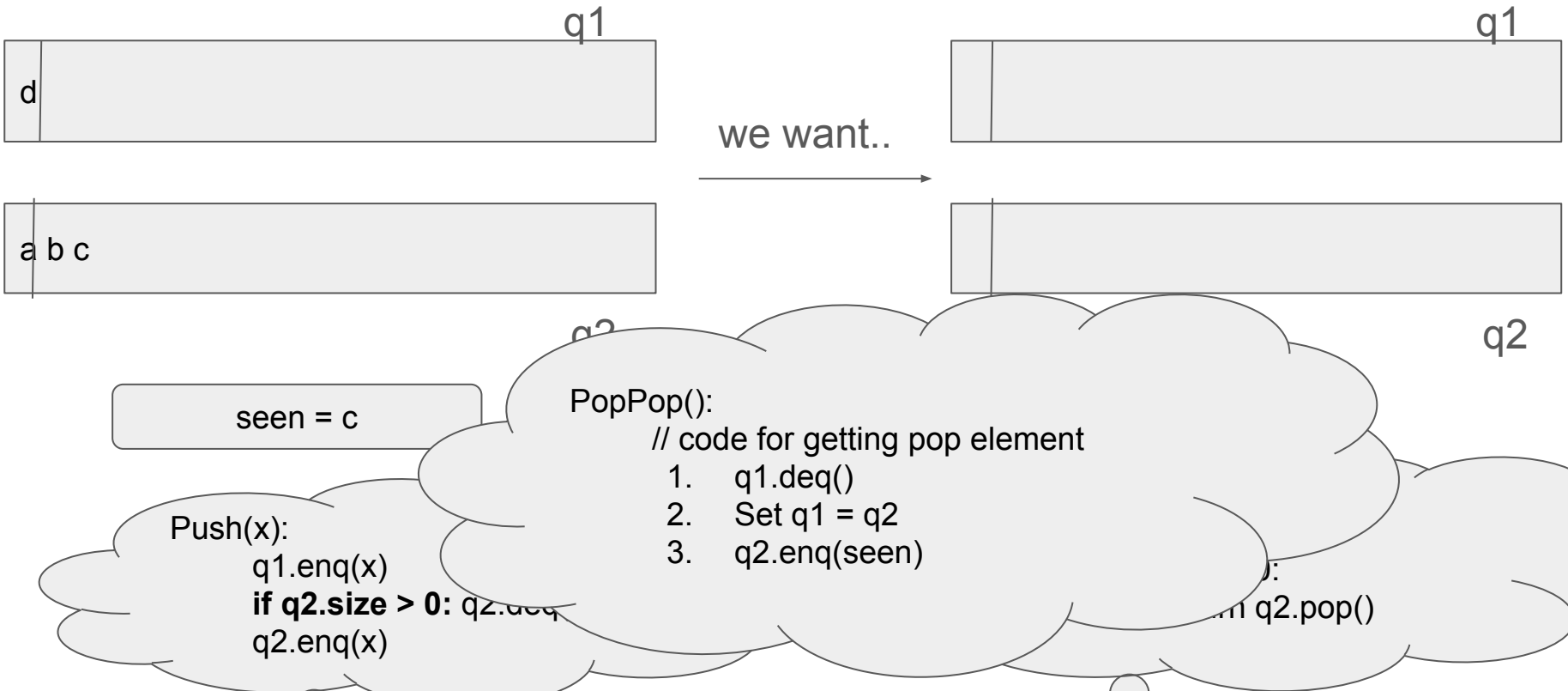
# Invariant for our stack? After pop pop



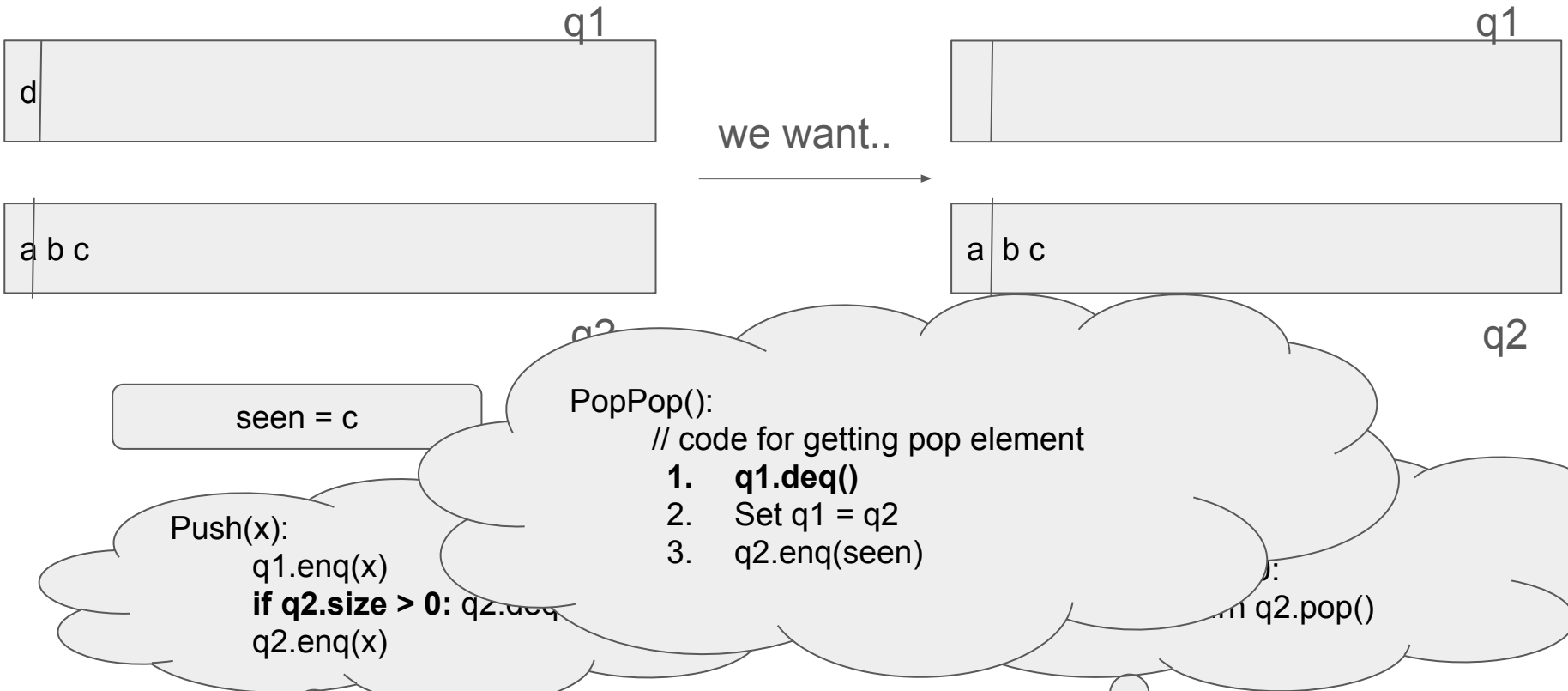
# Invariant for our stack? After pop pop



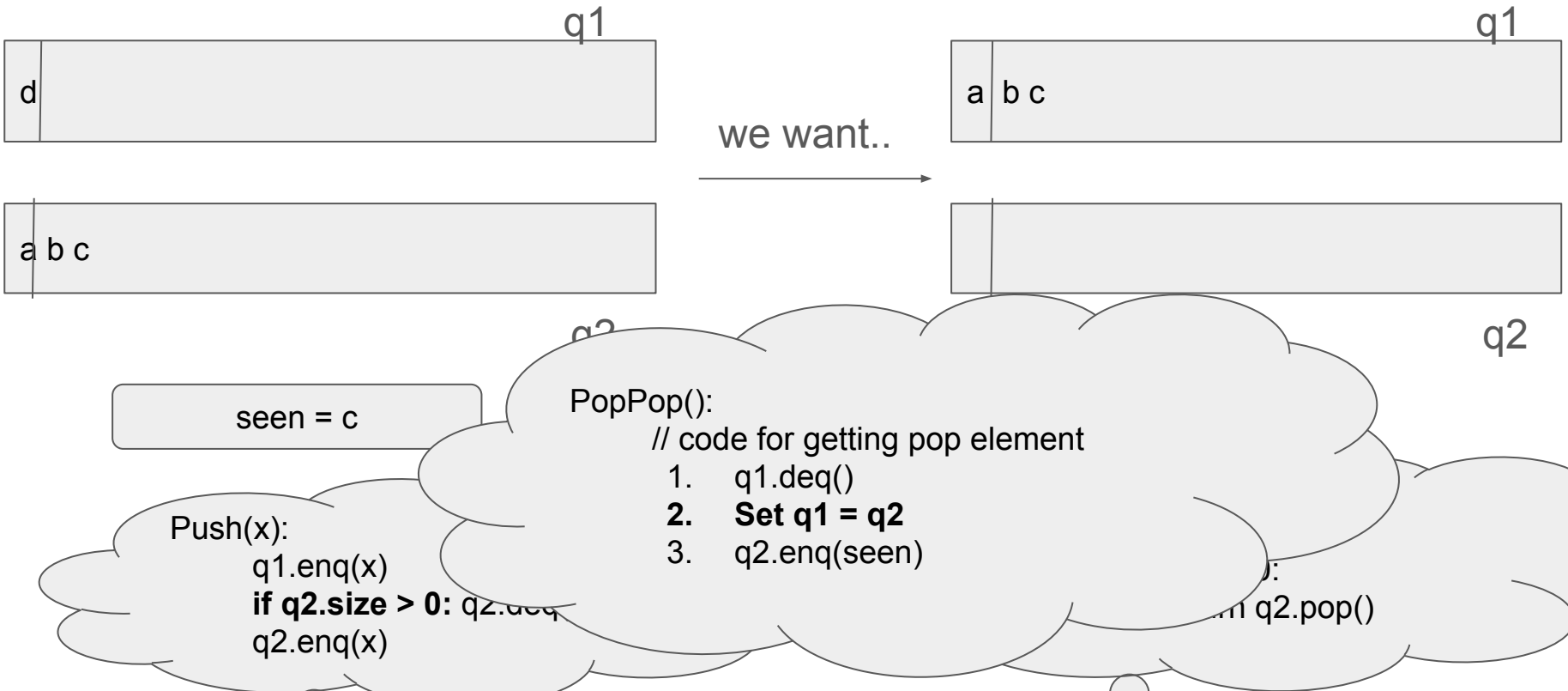
# Invariant for our stack? After pop pop



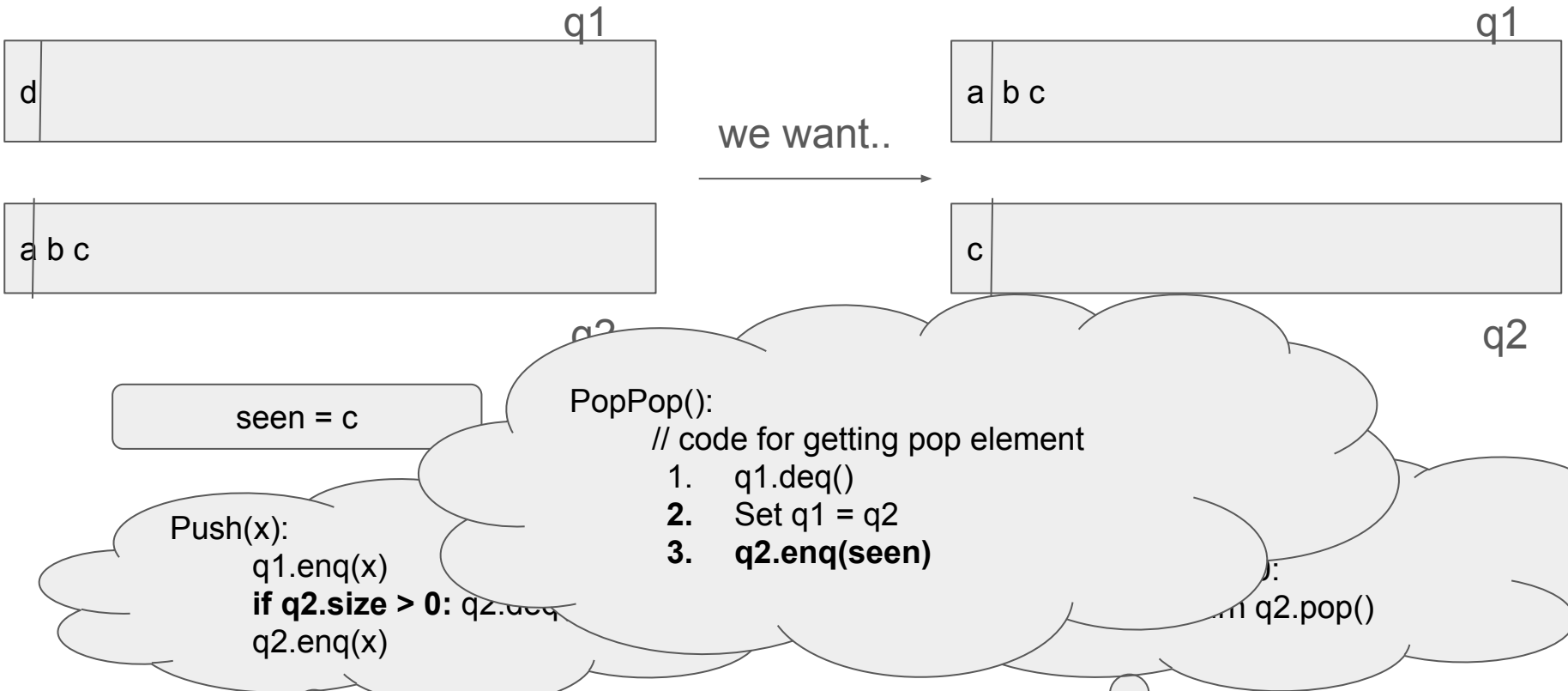
# Invariant for our stack? After pop pop



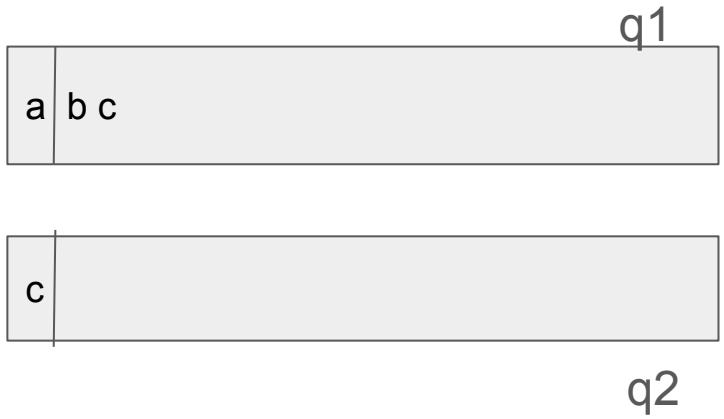
# Invariant for our stack? After pop pop



# Invariant for our stack? After pop pop



# We don’t have to change our previous push/pop impl.!

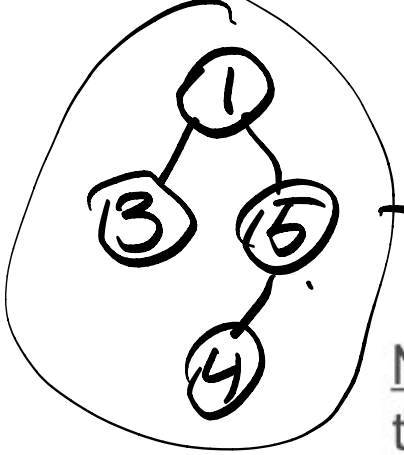



seen = c

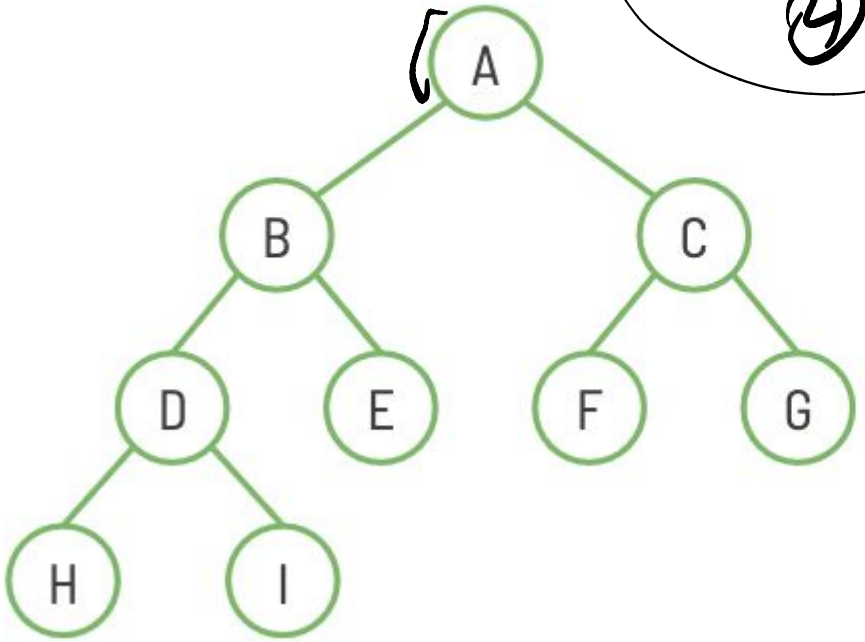
```
Push(x):
        q1.enq(x)
        if q2.size > 0: q2.deq()
        q2.enq(x)
```

```
PushPop():
        If q2.size() > 0:
                Return q2.pop()
```

# Binary Heaps



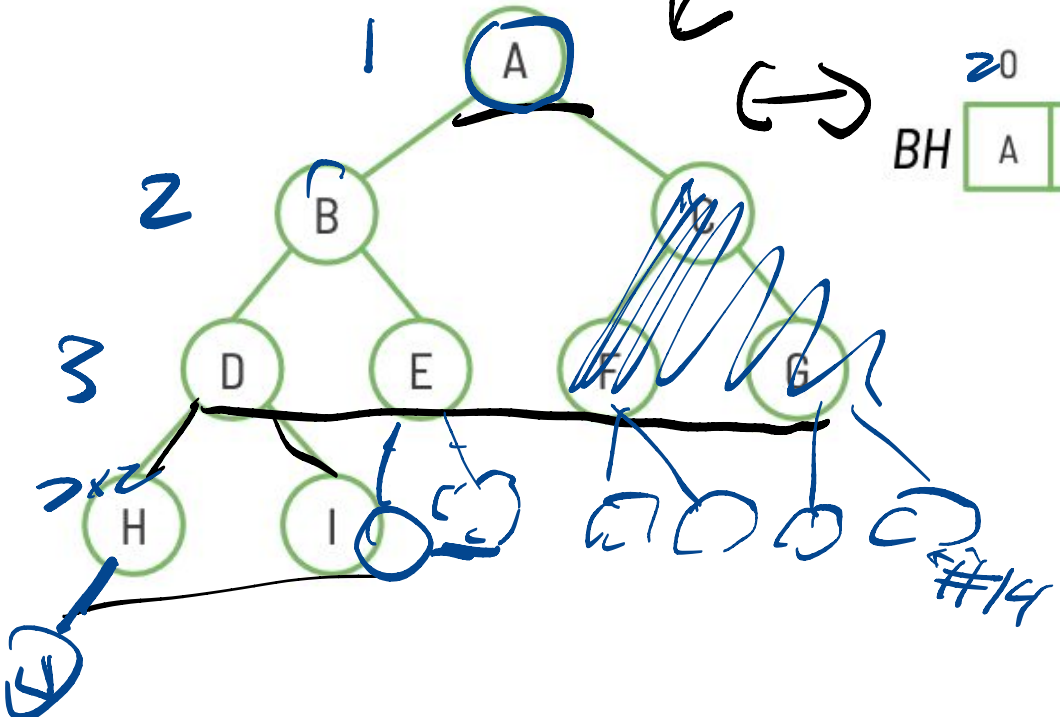

→ [5, 1, 4, 3]




Max-heap (aka **Max Priority Queue**) if the key in each node is **larger than** or equal to the keys in that node's two children (if any).

Min-heap (aka **Min Priority Queue)** if the key in each node is **less than** or equal to the keys in that node's two children (if any).

# Binary Heaps as Arrays





$$\text{leftchild}(i \in \mathbb{Z}_{\geq 0}) := 2i + 1$$

$$\text{rightchild}(i \in \mathbb{Z}_{\geq 0}) := 2i + 2$$

$$\text{parent}(i \in \mathbb{Z}^{+}) := \left\lfloor \frac{i-1}{2} \right\rfloor$$

## Question 3

**(Binary heap)**

(1) If the binary heap is represented as an array, and the root is stored at index 0, where is the left child of the node at index $i = 23$ stored?

A. 45

B. 46

C. 47

D. 48

E. 49

General formula for this?

## Question 3

**(Binary heap)**

(1) If the binary heap is represented as an array, and the root is stored at index 0, where is the left child of the node at index $i = 23$ stored?

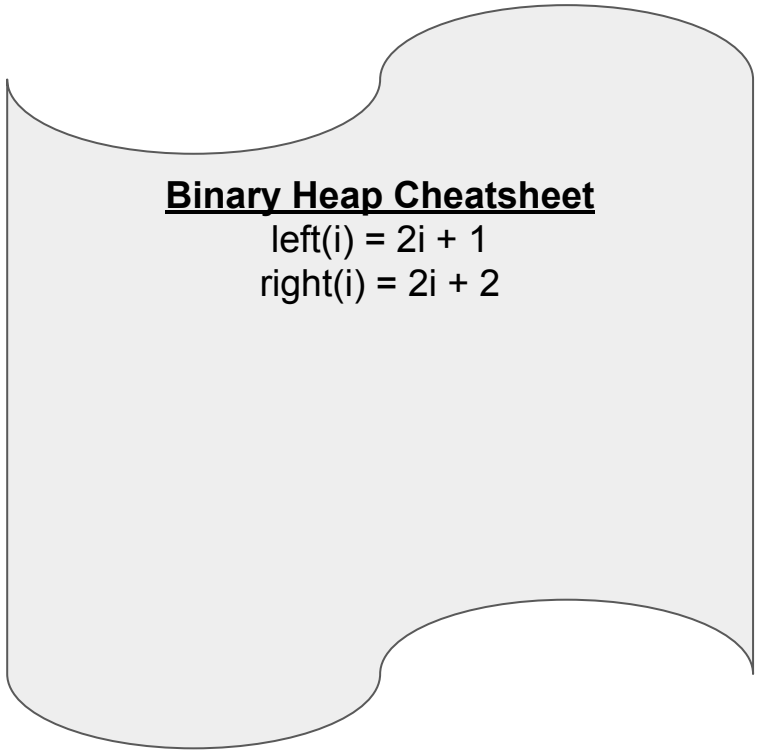
A. 45

B. 46

C. 47

D. 48

E. 49



(2) If the binary heap is represented as an array, and the root is stored at index 0, where is the parent of the node at index $i = 99$ stored?

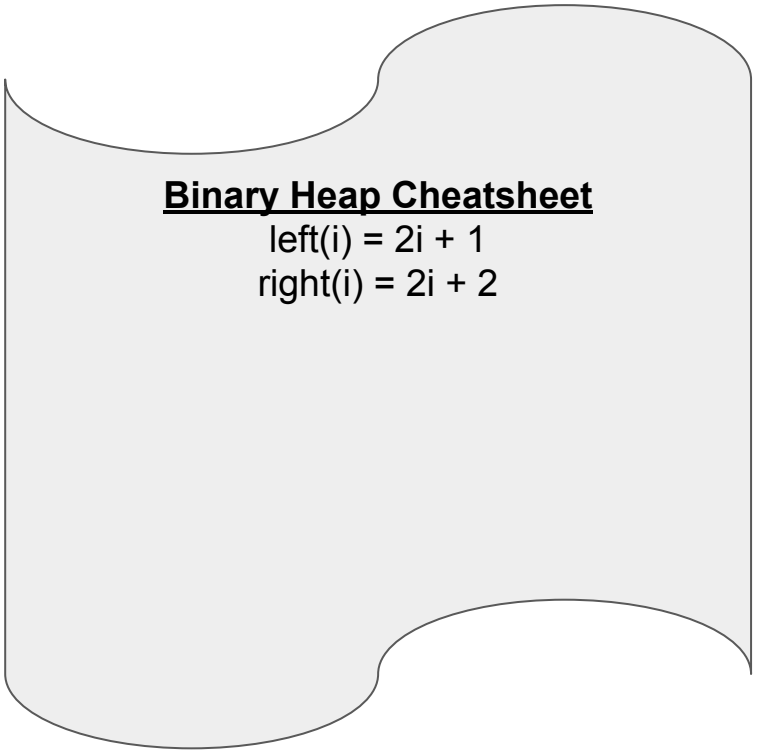
A. 45

B. 46

C. 47

D. 48

E. 49



(2) If the binary heap is represented as an array, and the root is stored at index 0, where is the parent of the node at index $i = 99$ stored?

A. 45

B. 46

C. 47

D. 48

E. 49

**Binary Heap Cheatsheet**

left(i) = 2i + 1

right(i) = 2i + 2

parent(i) = ⌊(i - 1) / 2⌋

$P(99) = \left\lfloor \frac{99-1}{2} \right\rfloor$

(3) If the binary heap is represented as an array of length $n = 99$, and the root is stored at index 0, where is the last non-leaf node stored?

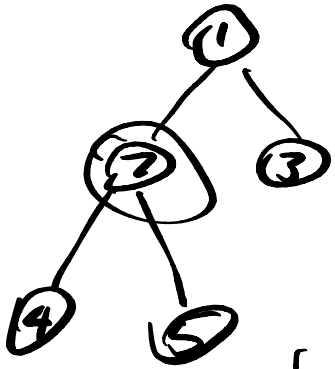
A. 45

B. 46

C. 47

D. 48

E. 49




$\lfloor n/2 - 1 \rfloor$

**Binary Heap Cheatsheet**

left(i) = 2i + 1

right(i) = 2i + 2

parent(i) = ⌊(i - 1) / 2⌋

(3) If the binary heap is represented as an array of length $n = 99$, and the root is stored at index 0, where is the last non-leaf node stored?

A. 45

B. 46

C. 47

D. 48

E. 49

**Binary Heap Cheatsheet**

left(i) = 2i + 1

right(i) = 2i + 2

parent(i) = ⌊(i - 1) / 2⌋

lastNonLeaf() = ⌊(n / 2) - 1⌋

48.5

General intuition: There are ~n/2 leaves since these are **complete trees**

(4) If the binary heap is represented as an array of length $n = 99$, and you want to insert an element, how many different locations of the element are possible after insertion?

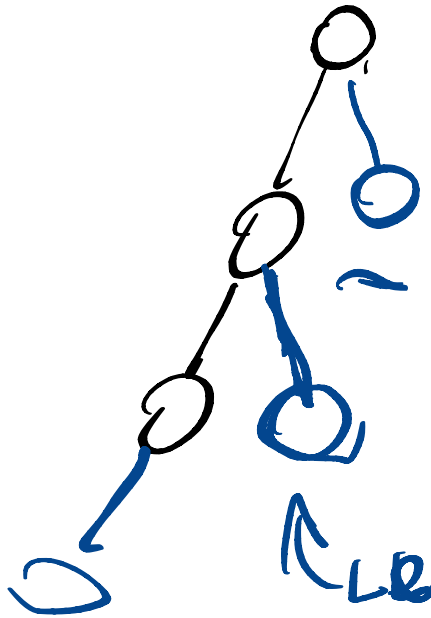
A. 5

B. 6

C. 7

D. 8

E. 9



I can insert ≤ 1 location per level.

(# levels + 1) in my heap

↑ Left leaning

**Binary Heap Cheatsheet**

left(i) = 2i + 1

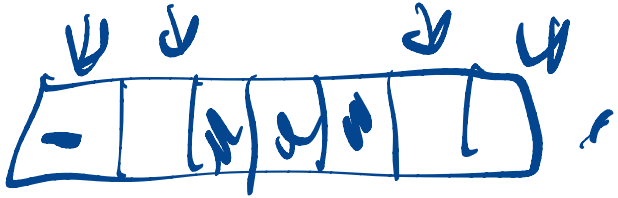
right(i) = 2i + 2

parent(i) = ⌊(i - 1) / 2⌋

lastNonLeaf() = ⌊(n / 2) - 1⌋

(4) If the binary heap is represented as an array of length $n = 99$, and you want to insert an element, how many different locations of the element are possible after insertion?

A. 5
B. 6
C. 7
D. 8
E. 9




| nodes per level | sum |
|---|---|
| 1 | 1 |
| 2 | 3 |
| 4 | 7 |
| 8 | 15 |
| 16 | 31 |
| 32 | 63 |
| 64 | 127 |

6 + 1

how the tree looks

7 positions

**Binary Heap Cheatsheet**

left(i) = 2i + 1
right(i) = 2i + 2
parent(i) = ⌊(i - 1) / 2⌋
lastNonLeaf() = ⌊(n / 2) - 1⌋
height() = ⌊lg n⌋