PSO 3

Exam this **Friday**. Topics are

1. Run time Expressions/Asymptotic Analysis
2. Array
3. Linked List
4. Stack n’ Queue
5. Trees
6. Heaps & Heap Sort

Question 1

(Linked List) Consider a sorted circular doubly linked list of N numbers where the head element points to the smallest element in the list. Provide the asymptotic complexity in big-Θ with a brief explanation (including assumptions and analysis for each case, if there is more than one) for the following operations.

1. Inserting an element in its sorted position.

Question 1

(Linked List) Consider a sorted circular doubly linked list of N numbers where the head element points to the smallest element in the list. Provide the asymptotic complexity in big-Θ with a brief explanation (including assumptions and analysis for each case, if there is more than one) for the following operations.

2. Finding the smallest element in the list.

Question 1

(Linked List) Consider a sorted circular doubly linked list of N numbers where the head element points to the smallest element in the list. Provide the asymptotic complexity in big-Θ with a brief explanation (including assumptions and analysis for each case, if there is more than one) for the following operations.

3. Finding the 3^{rd} - largest element in the list.

Question 1

(Linked List) Consider a sorted circular doubly linked list of N numbers where the head element points to the smallest element in the list. Provide the asymptotic complexity in big-Θ with a brief explanation (including assumptions and analysis for each case, if there is more than one) for the following operations.

4. Finding the median in the list.

Question 1

(Linked List) Consider a sorted circular doubly linked list of N numbers where the head element points to the smallest element in the list. Provide the asymptotic complexity in big-Θ with a brief explanation (including assumptions and analysis for each case, if there is more than one) for the following operations.

1. Inserting an element in its sorted position.
2. Finding the smallest element in the list.
3. Finding the 3^{rd} - largest element in the list.
4. Finding the median in the list.

(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

E. 15

Definition of a full binary tree?

(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

examples

Definition of a full binary tree?

Every node is either a

- leaf or,
- inner node with two children

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?

(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

(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.

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()
```

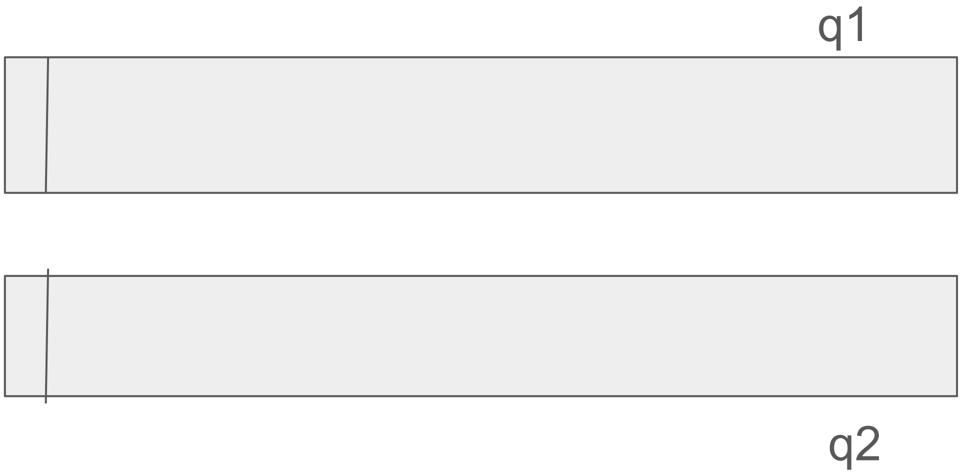
General Strat for these types of problems

- 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.

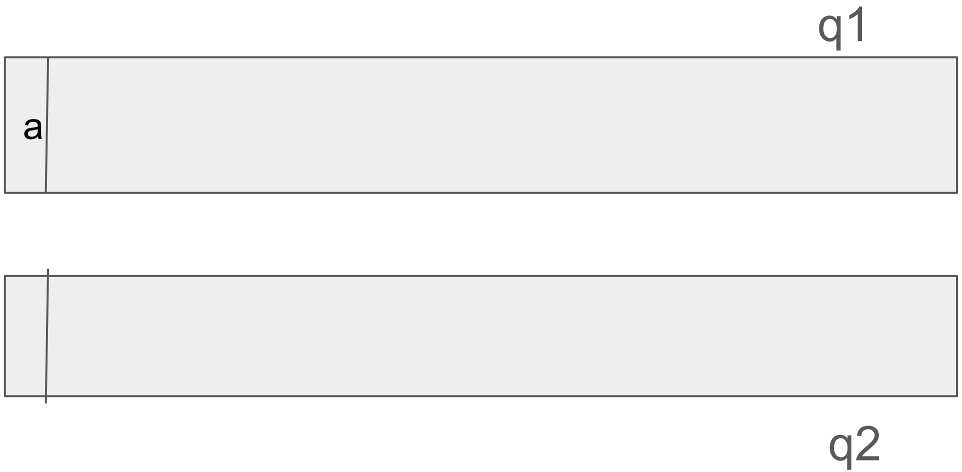
Push(a)
Push(b)
Push(c)
Push(d)



Example

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

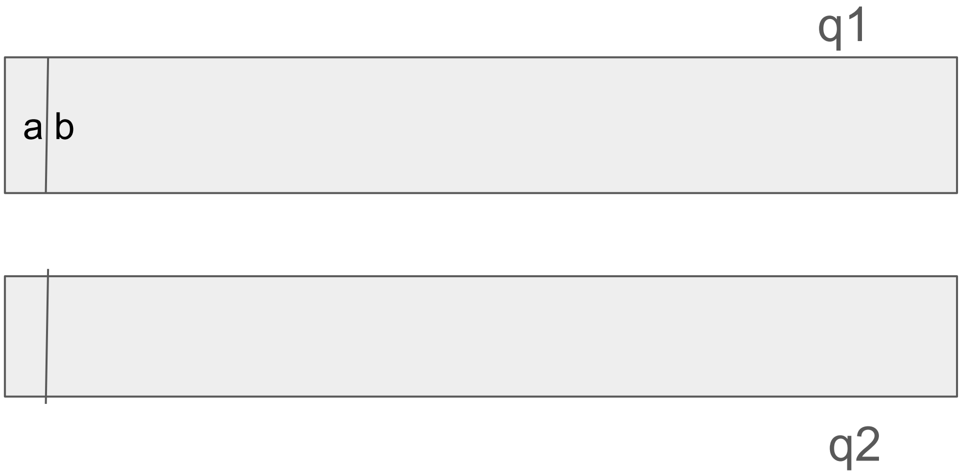
Push(a)
Push(b)
Push(c)
Push(d)



Example

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

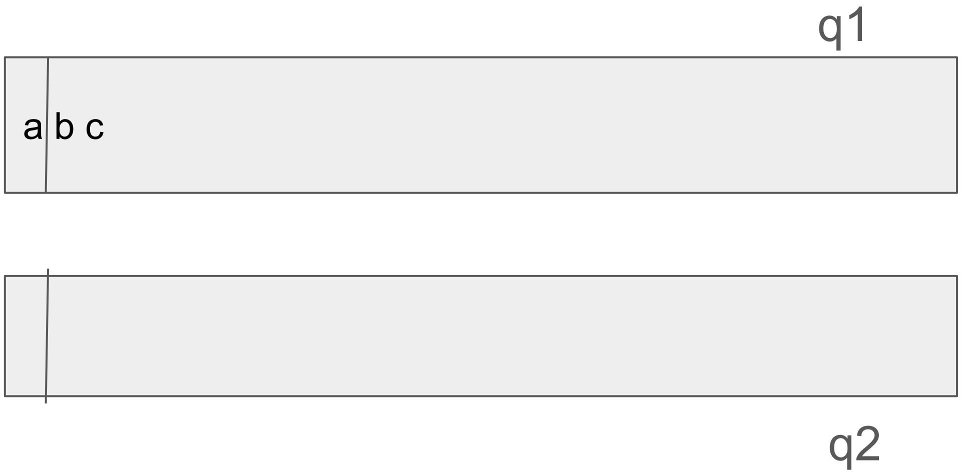
Push(a)
Push(b)
Push(c)
Push(d)



Example

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

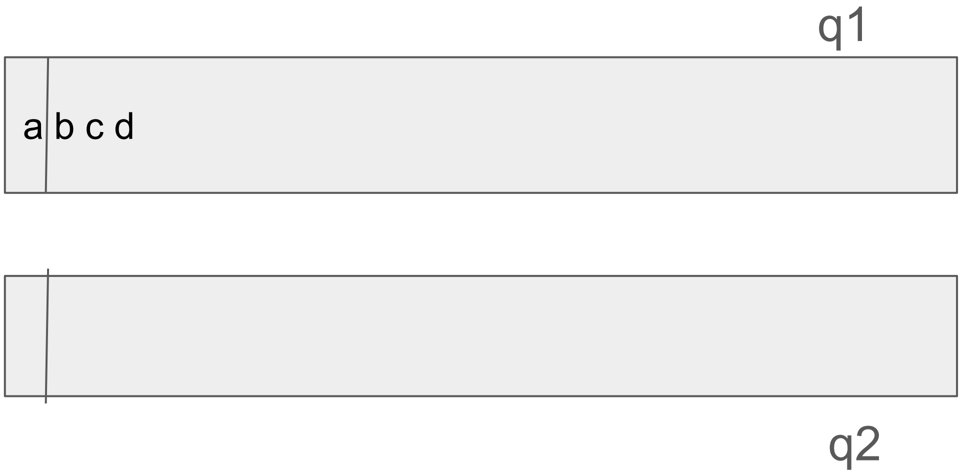
Push(a)
Push(b)
Push(c)
Push(d)



Example

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

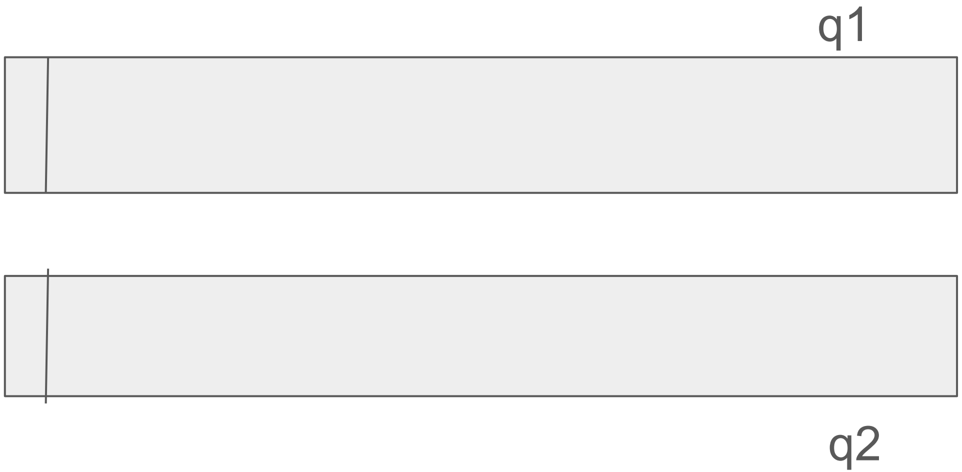
Push(a)
Push(b)
Push(c)
Push(d)



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.

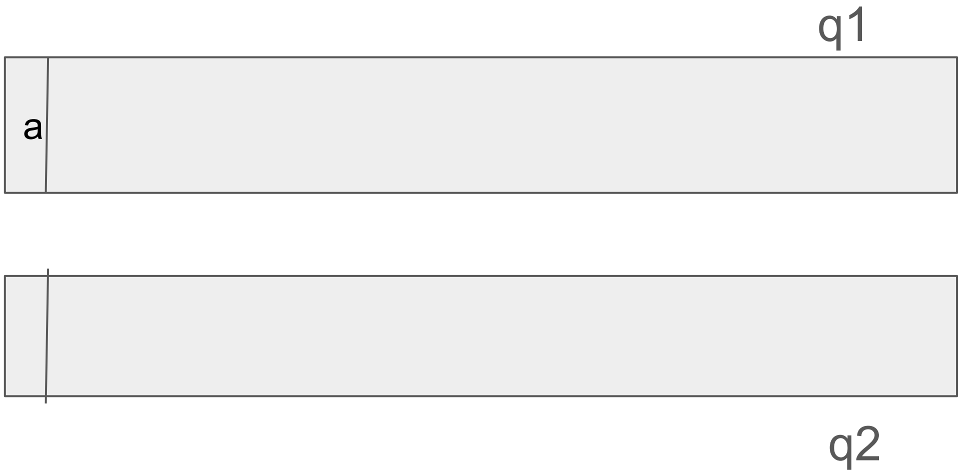
Push(a)
Push(b)
Pop() #should pop b
Push(c)
Pop() # should pop c



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.

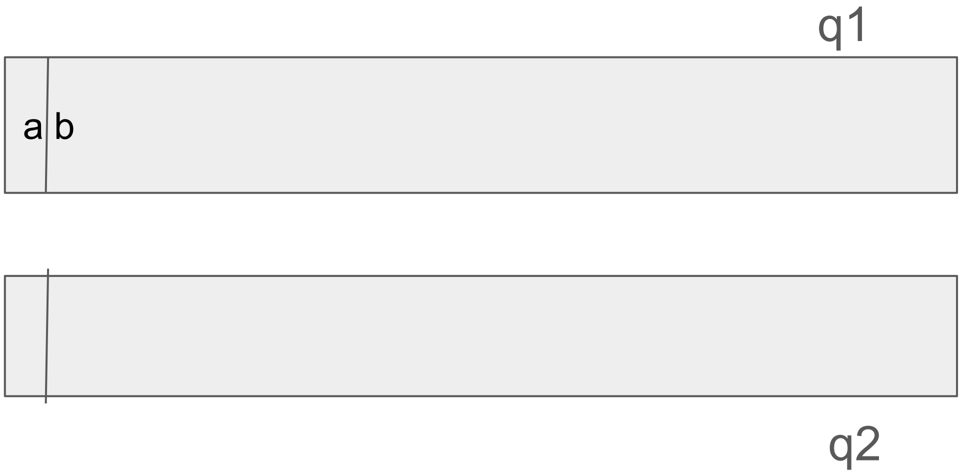
Push(a)
Push(b)
Pop() #should pop b
Push(c)
Pop() # should pop c



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.

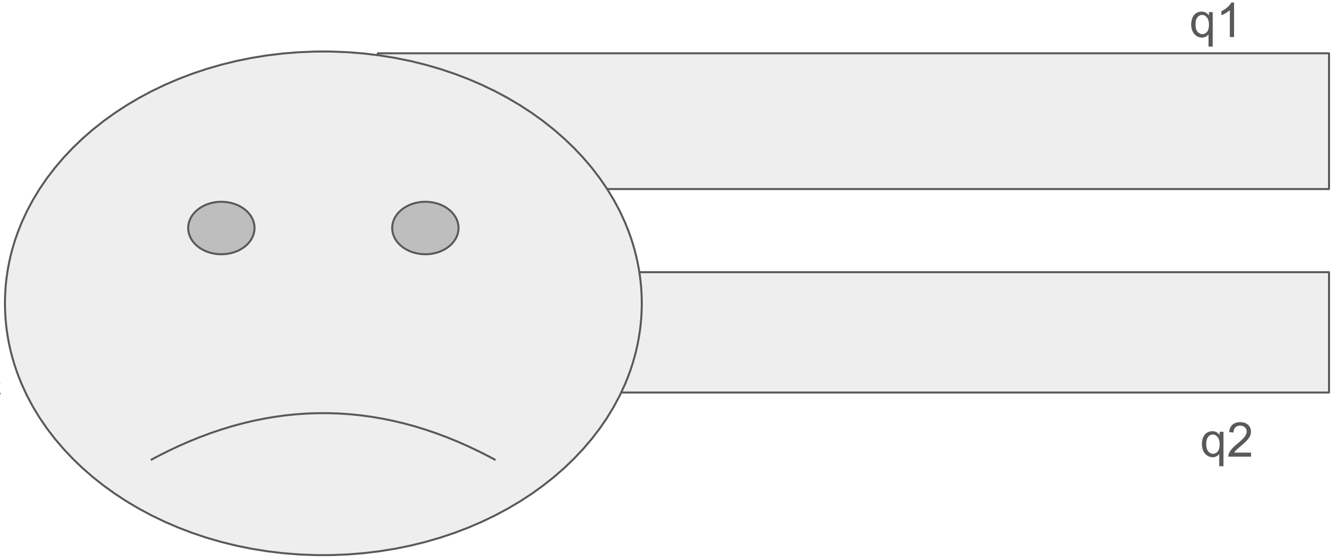
Push(a)
Push(b)
Pop() #should pop b
Push(c)
Pop() # should pop c



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.

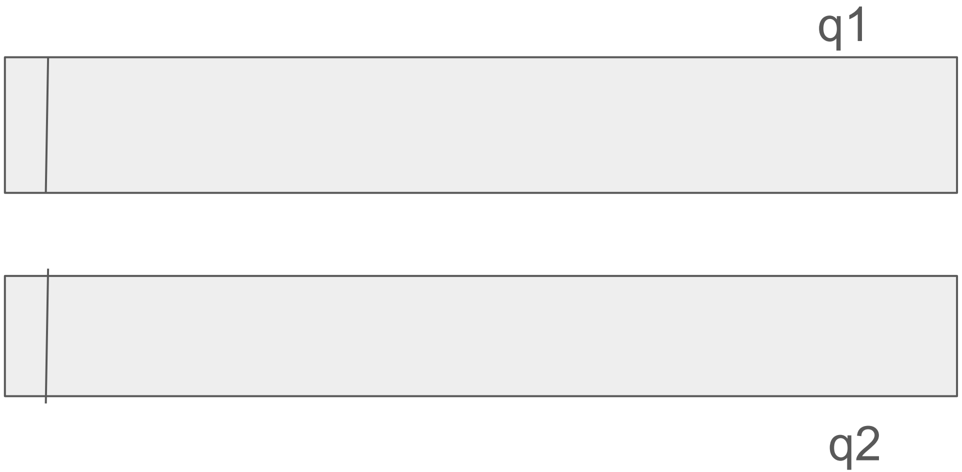
Push(a)
Push(b)
Pop() #should pop b
Push(c)
Pop() # should pop c



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.

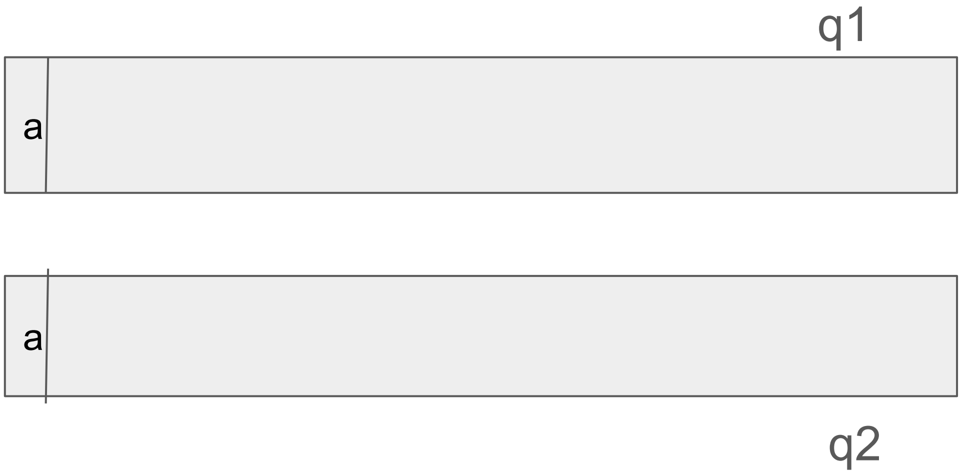
Push(a)
Push(b)
Pop() #should pop b
Push(c)
Pop() # should pop c



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.

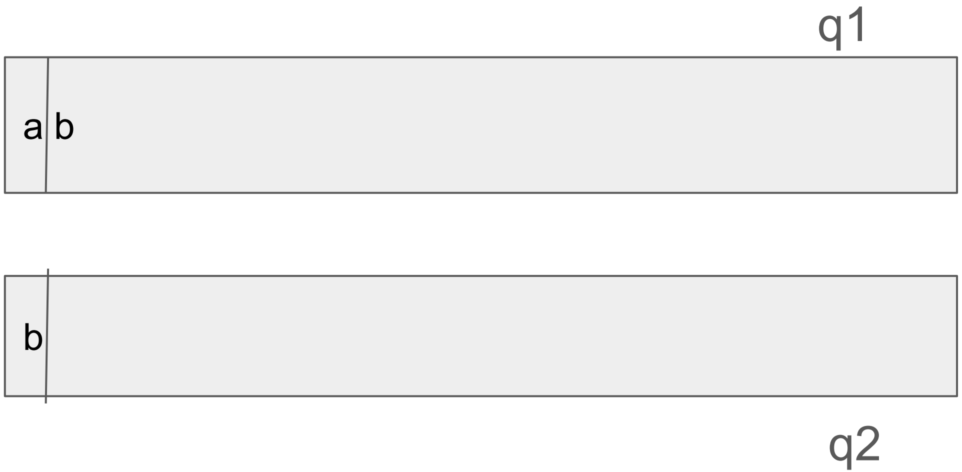
Push(a)
Push(b)
Pop() #should pop b
Push(c)
Pop() # should pop c



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

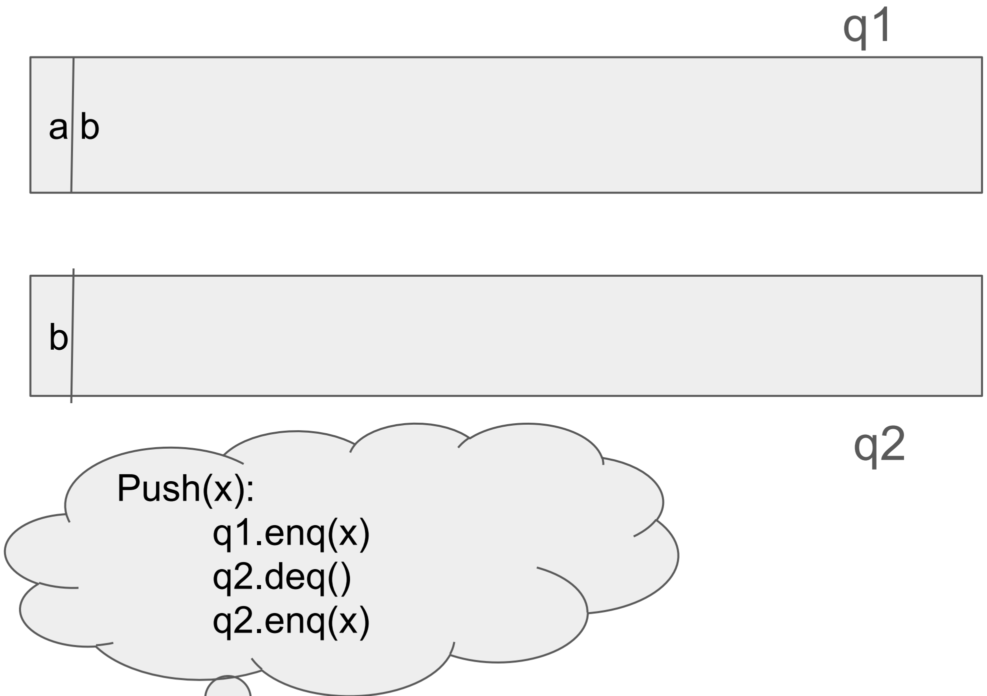


How to implement this?

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.

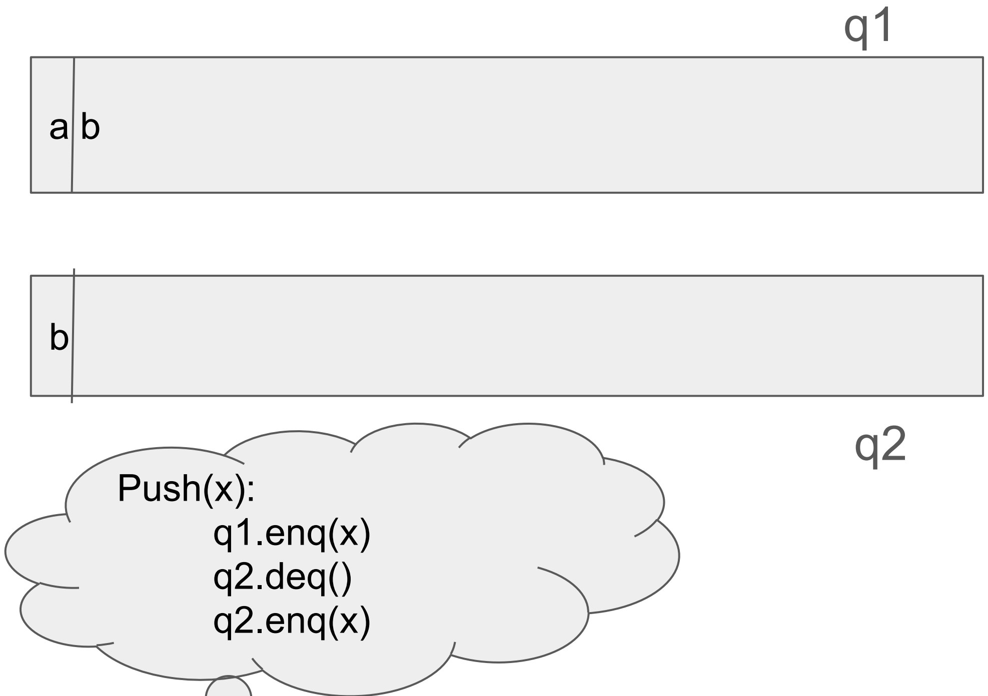
Push(a)
Push(b)
Pop() #should pop b
Push(c)
Pop() # should pop c



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.pop()
```

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

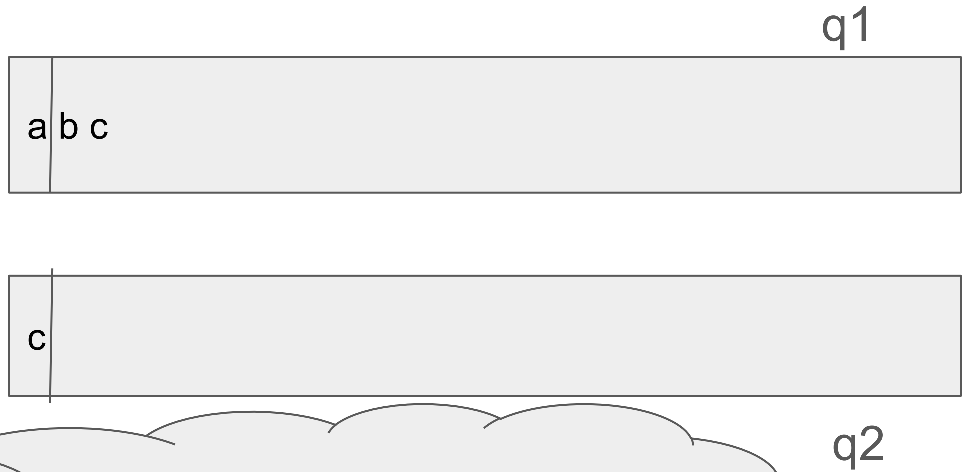
```
Push(x):
    q1.enq(x)
    q2.deq()
    q2.enq(x)
```

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

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



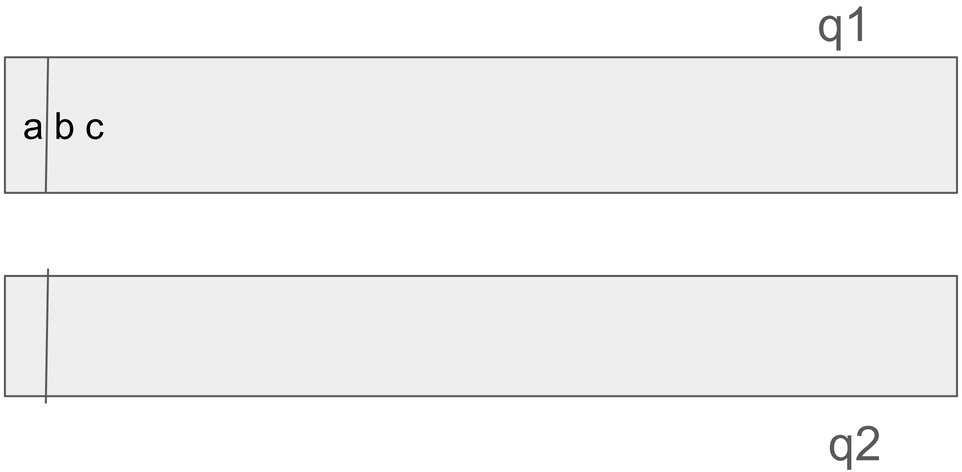

```
Push(x):
        q1.enq(x)
        if q2.size > 0: q2.deq()
        q2.enq(x)
```

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

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.

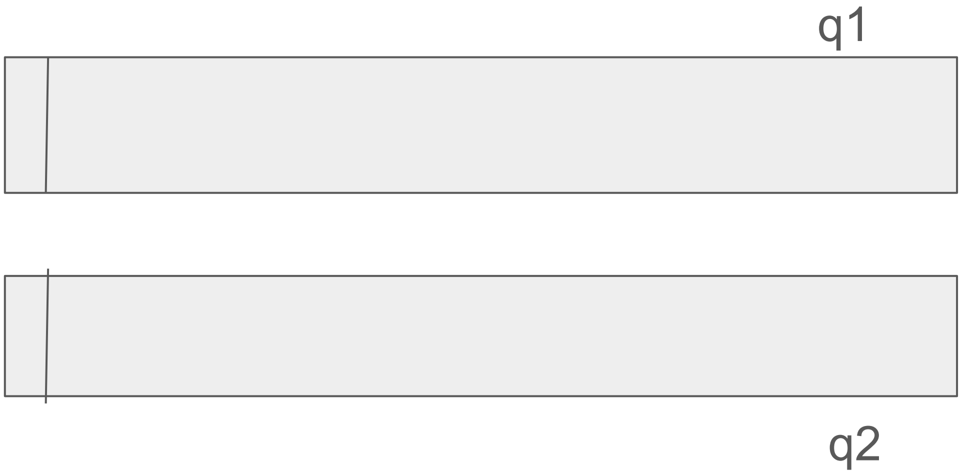
Push(a)
Push(b)
Pop() #should pop b
Push(c)
Pop() # should pop c



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.

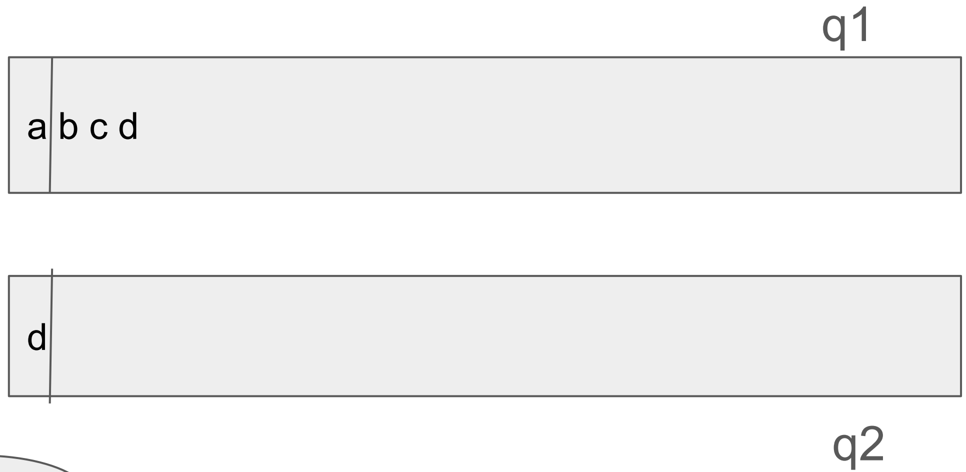


Push(a)
Push(b)
Push(c)
Push(d)
Pop() #should pop d
Pop() # should pop c

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.



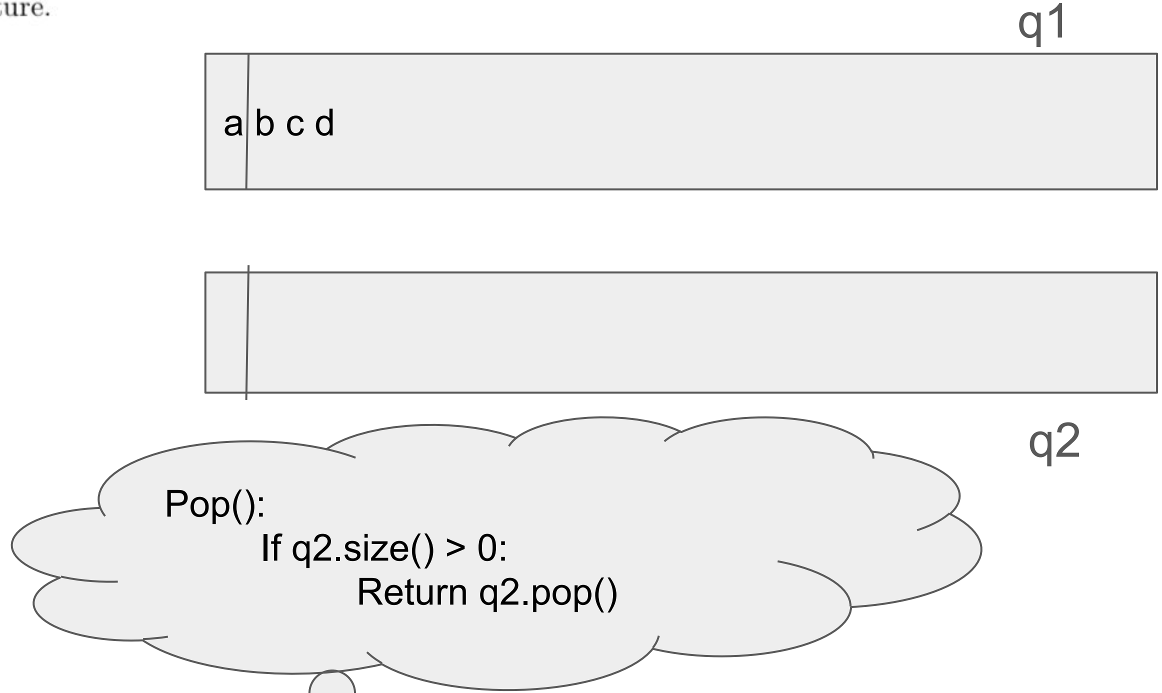
Push(a)
Push(b)
Push(c)
Push(d)
Pop() #should pop d
Pop() # should pop c

```
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.

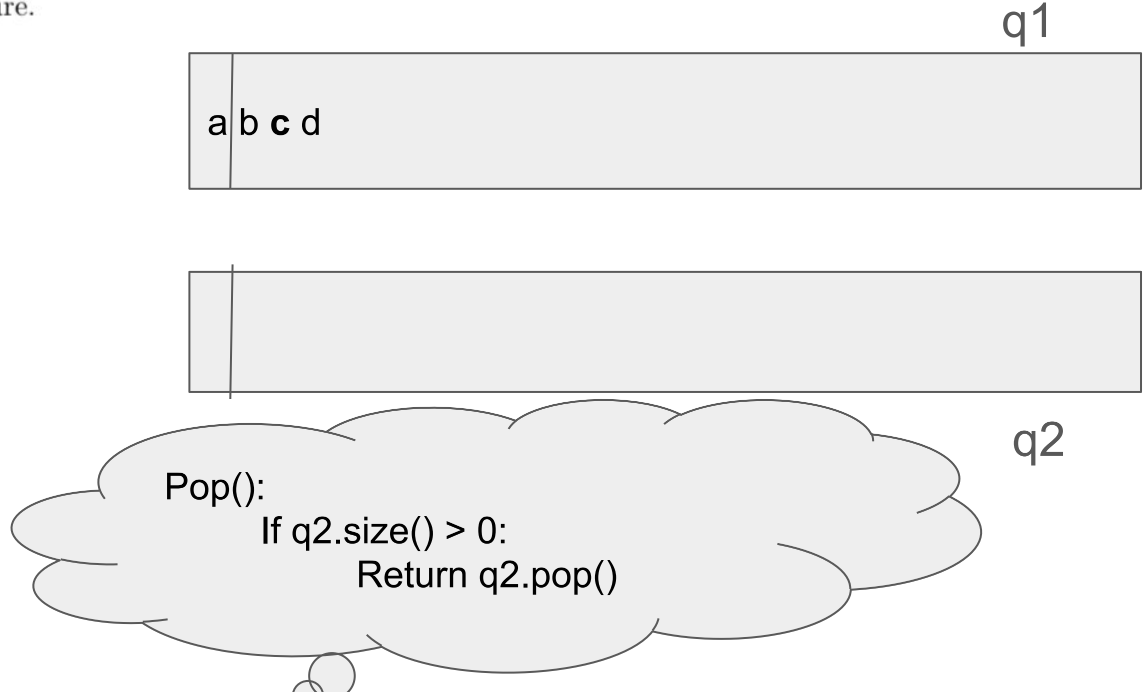
Push(a)
Push(b)
Push(c)
Push(d)
Pop() #should pop d
Pop() # should pop c



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

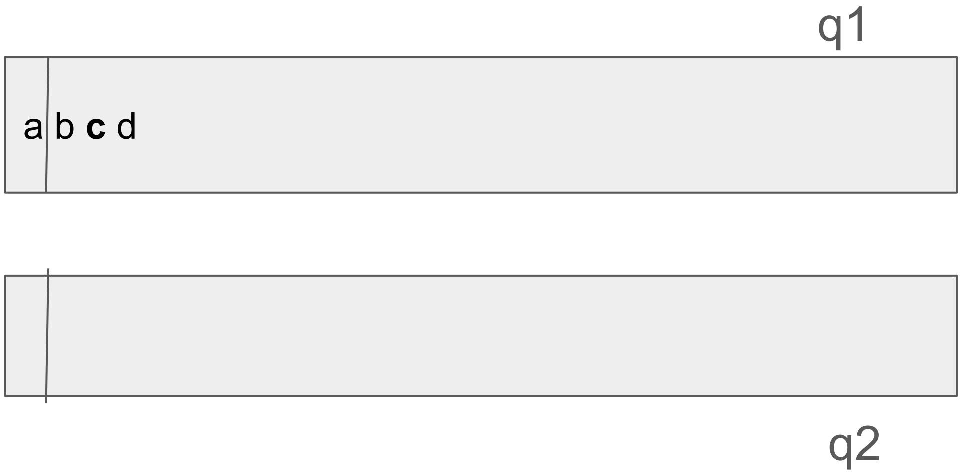
1. Pushing an element to the stack takes no more than $O(1)$ operations.
2. Popping from the stack takes no more than $O(1)$ operations if performed after a push.
3. Popping from the stack takes no more than $O(n)$ operations if performed after another pop, where n is the number of elements in the data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

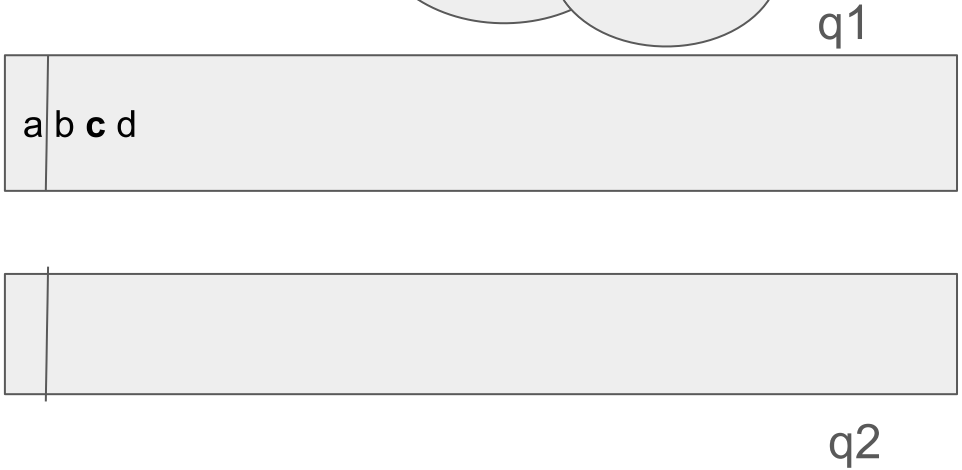
Idea: Deque everything from q1 into q2
Keep track of elements seen to get c

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



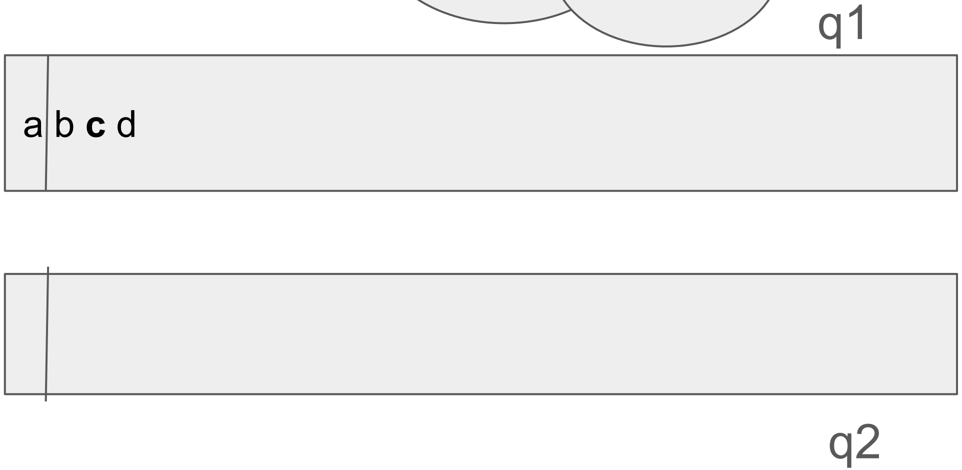
Idea: Deque everything from q1 into q2
Keep track of elements seen to get c

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

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

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.

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



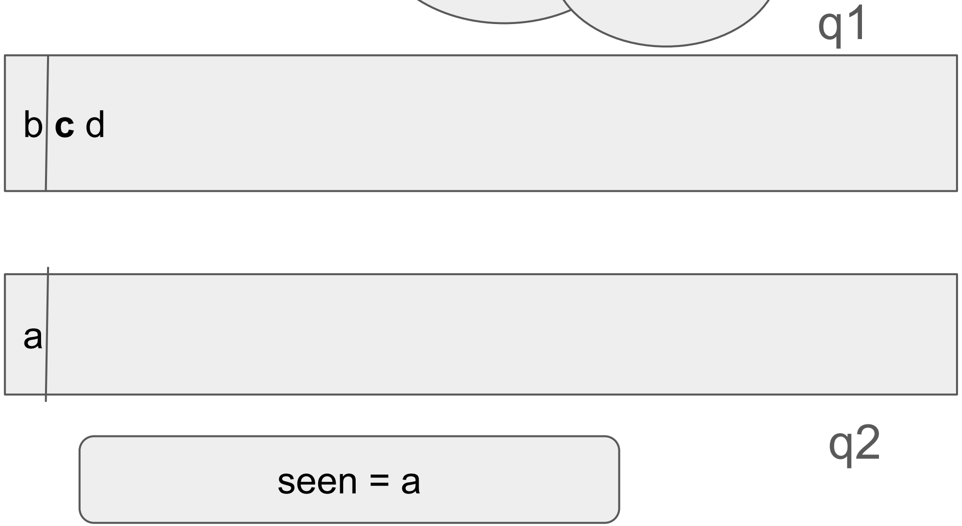
Idea: Deque everything from q1 into q2
Keep track of elements seen to get c

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)
        If q1.size() == 1:
            res = seen
```

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



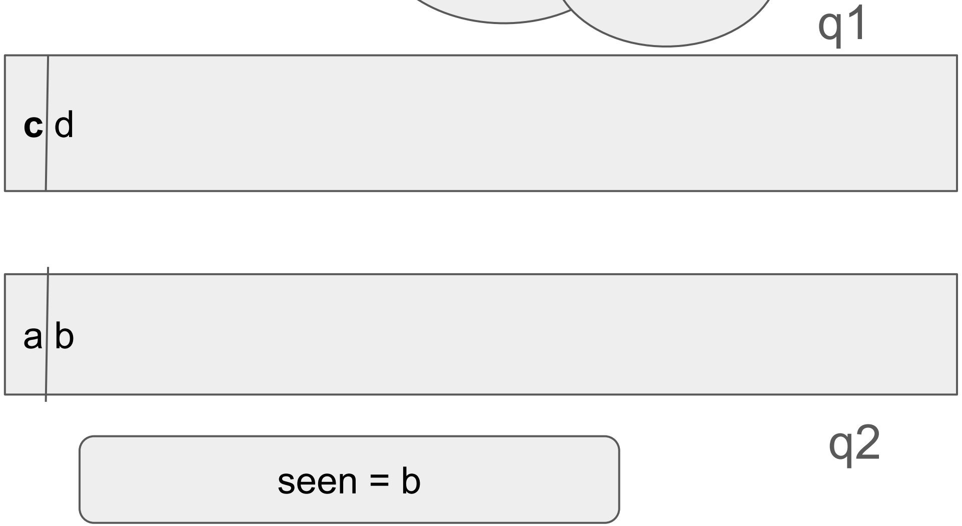
Idea: Deque everything from q1 into q2
Keep track of elements seen to get c

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()
        q2.enq(seen)
        If q1.size() == 1:
            res = seen
```

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



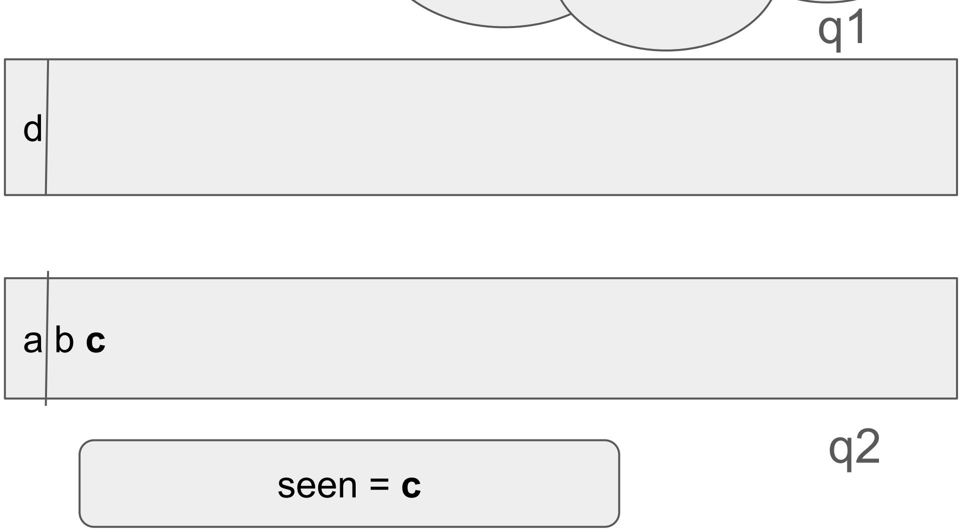
Idea: Deque everything from q1 into q2
Keep track of elements seen to get c

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

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

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.

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



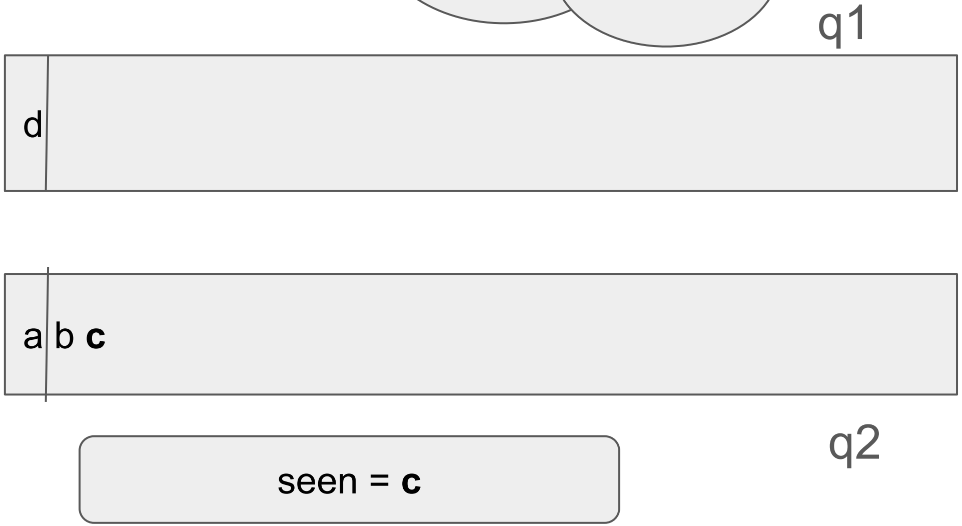
Idea: Deque everything from q1 into q2
Keep track of elements seen to get c

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)
        If q1.size() == 1:
            res = seen
```

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



Cool we have our result!
But our “stack” is ugly now.. How do we push/pop again?

Philosophy of Data Structures: Culling Chaos

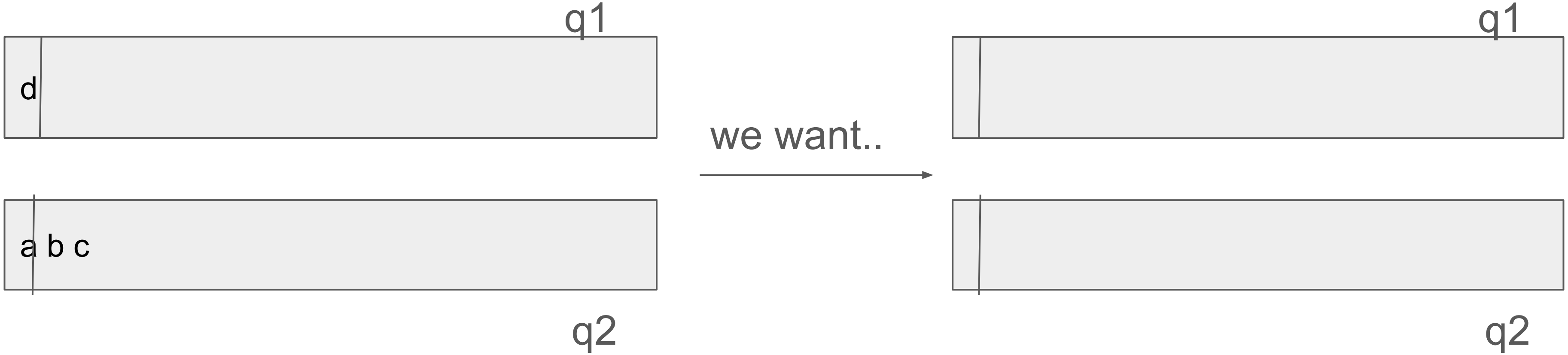
Sure fire design philosophy of data structures is **maintaining Invariants**

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

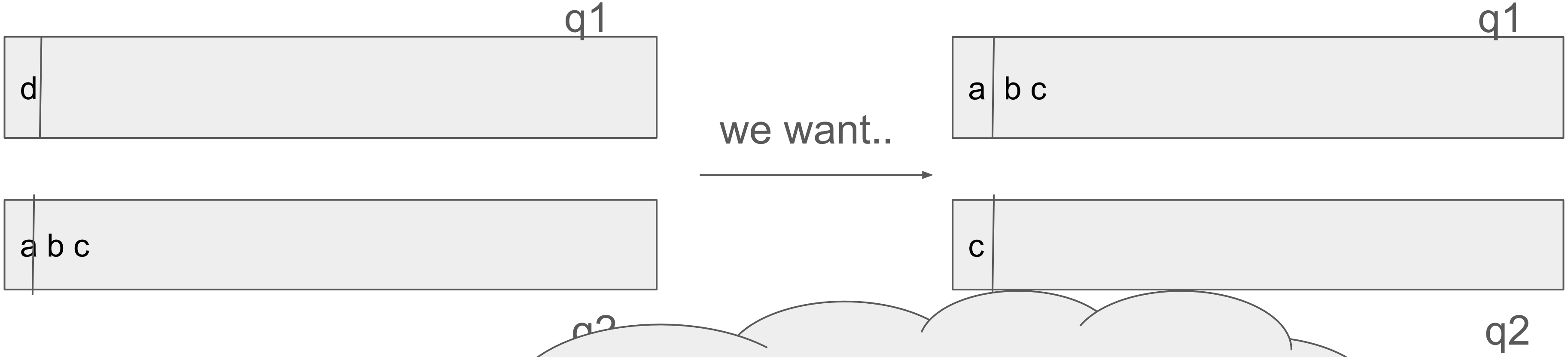


seen = c

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



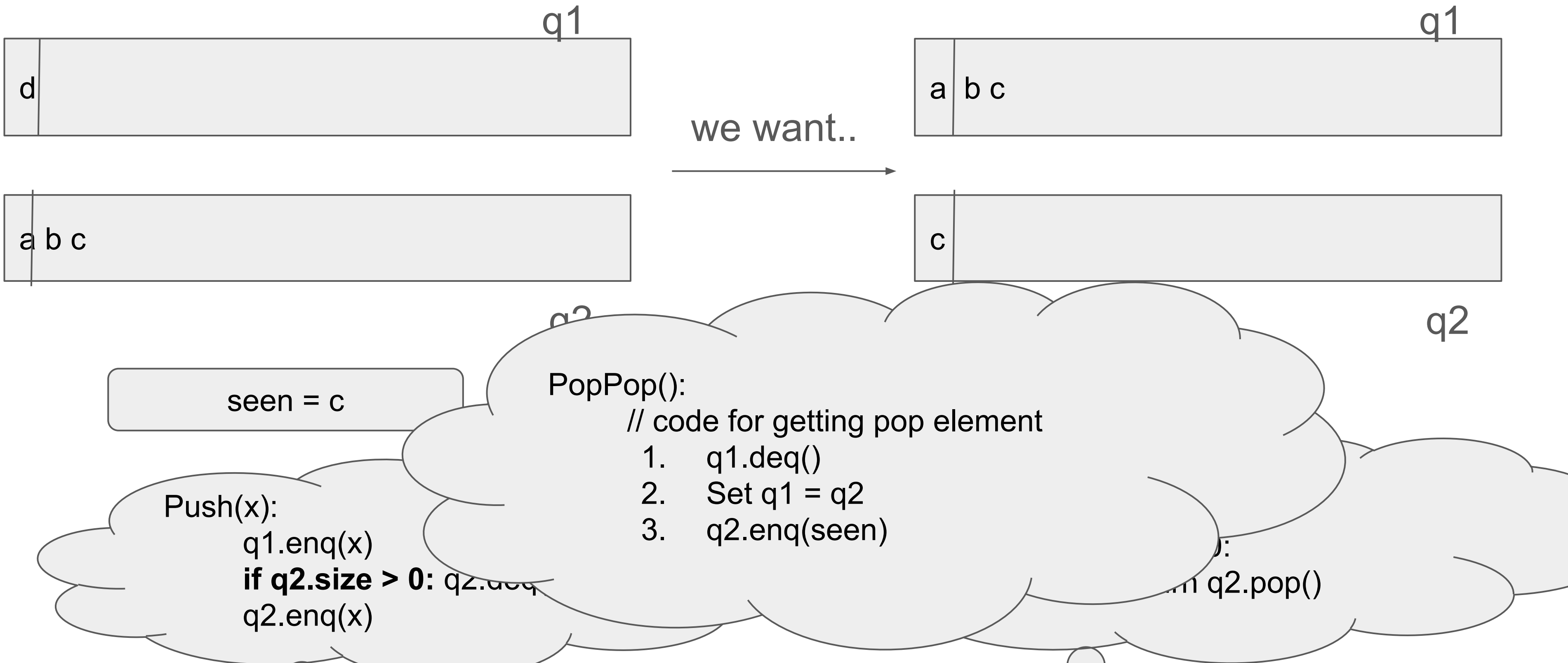
seen = c

```
PopPop():
        // code for getting pop element
        // TODO: code for fixing stack
```

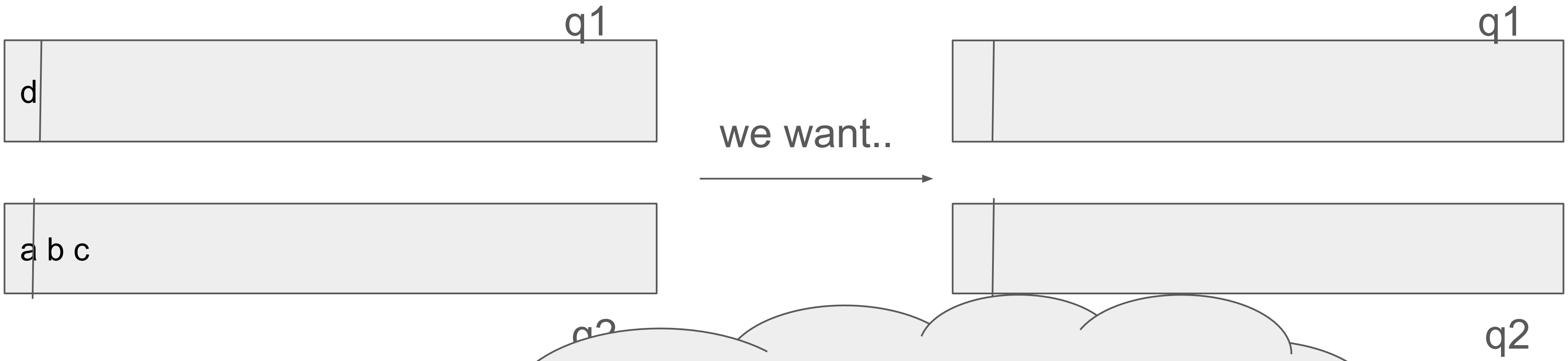
Push(x):
        q1.enq(x)
        if q2.size > 0: q2.deq
        q2.enq(x)

):
n q2.pop()

Invariant for our stack? After pop pop



Invariant for our stack? After pop pop



seen = c

PopPop():

// code for getting pop element

1. q1.deq()
2. Set q1 = q2
3. q2.enq(seen)

Push(x):

q1.enq(x)

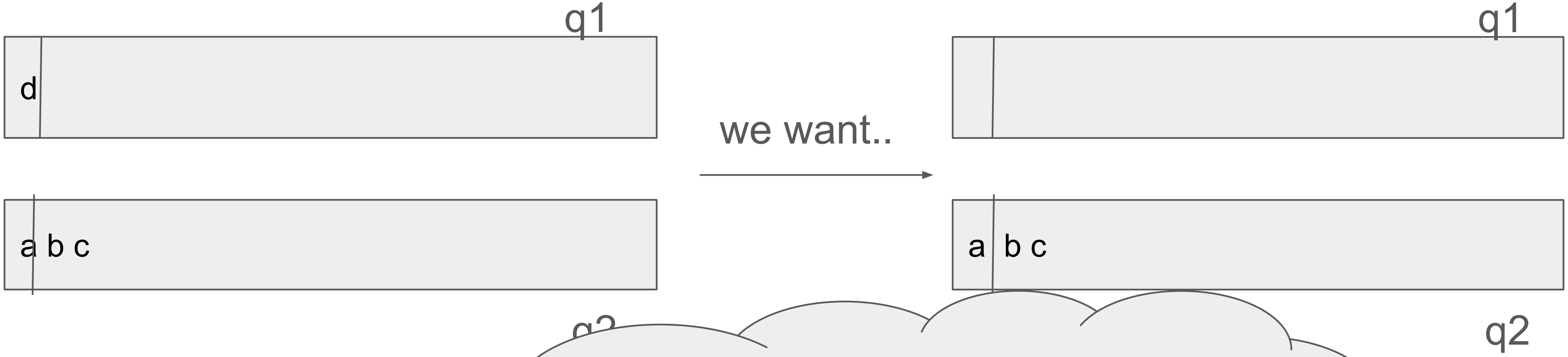
if q2.size > 0: q2.deq

q2.enq(x)

):

n q2.pop()

Invariant for our stack? After pop pop



seen = c

PopPop():

// code for getting pop element

1. **q1.deq()**
2. Set q1 = q2
3. q2.enq(seen)

Push(x):

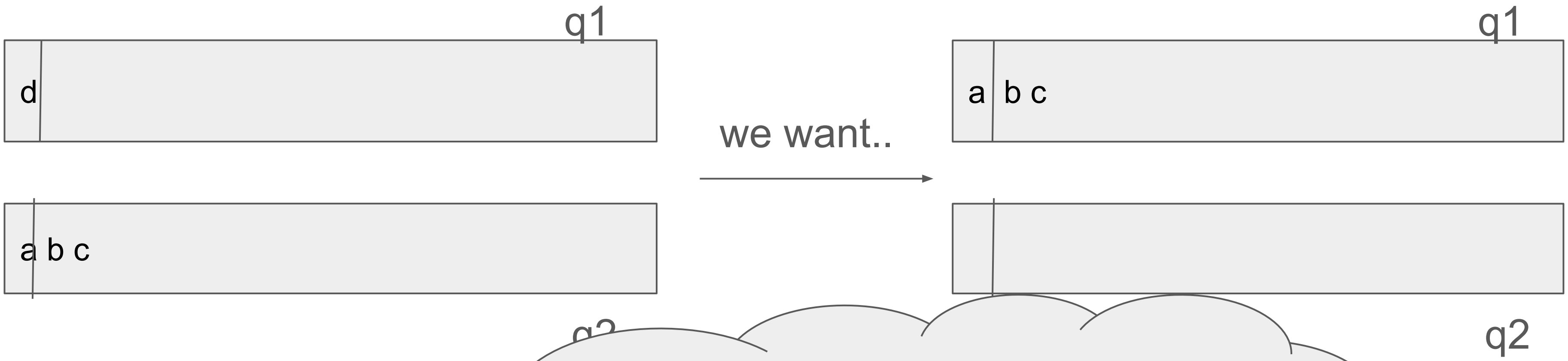
q1.enq(x)

if q2.size > 0: q2.deq

q2.enq(x)

q2.pop()

Invariant for our stack? After pop pop



seen = c

PopPop():

// code for getting pop element

1. q1.deq()
2. **Set q1 = q2**
3. q2.enq(seen)

Push(x):

q1.enq(x)

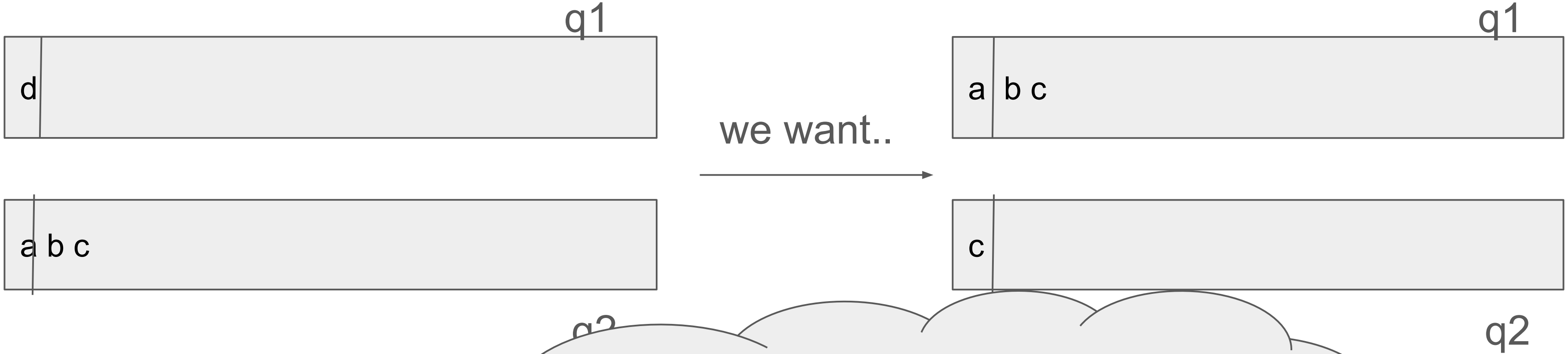
if q2.size > 0: q2.deq

q2.enq(x)

):

q2.pop()

Invariant for our stack? After pop pop



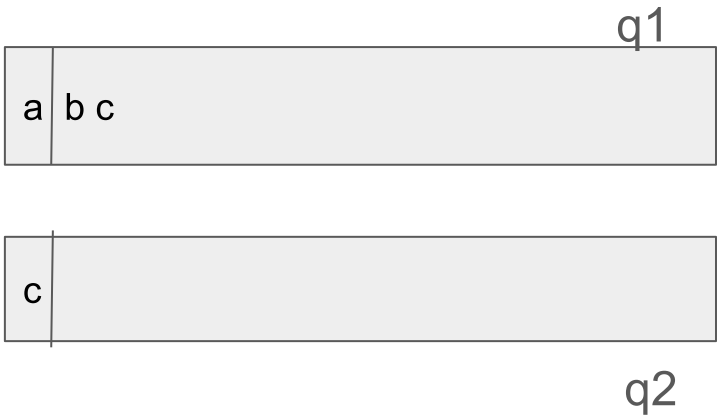
seen = c

PopPop():
  // code for getting pop element
  1. q1.deq()
  2. Set q1 = q2
  3. **q2.enq(seen)**

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

):
n q2.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()
```

Question 4

(Review)

(1) The big-O closed-form runtime expression $T(n)$ for the recurrence $T(n) = 3T(n/3) + n$ is (assume n is a power of 3 and $T(1) = 1$)

A. $O(n)$

B. $O(n \log n)$

C. $O(n^3 \log n)$

D. $O(\sqrt[3]{n} \log n)$

E. $O(n \sqrt[3]{\log n})$

(2) Two algorithms are developed based on the following template

```
function A(n : Z≥1 power of 2)
    if n = 1 then
        return 1
    end if
    ______________
    return A(n/2) + A(n/2)
end function
```

The missing part requires $F(n)$ time in Algorithm $\mathcal{A}_1$, and requires $G(n)$ time in Algorithm $\mathcal{A}_2$, where $F(n)$ and $G(n)$ are two functions of n.

If $F(n) = \Theta(G(n))$**, then** $\mathcal{A}_1(n) = \Theta(\mathcal{A}_2(n))$**.**

The above statement is

A. True

B. False

C. Possibly true/ Possible false

(3) Consider a sorted circular doubly-linked list where the head element points to the smallest element in the list. What is the time complexity to find the largest element in the list?

A. $O(1)$

B. $O(\log n)$

C. $O(n)$

D. $O(n \log n)$

(1) The big-O closed-form runtime expression $T(n)$ for the recurrence $T(n) = 3T(n/3) + n$ is (assume n is a power of 3 and $T(1) = 1$)

A. $O(n)$

B. $O(n \log n)$

C. $O(n^3 \log n)$

D. $O(\sqrt[3]{n} \log n)$

E. $O(n \sqrt[3]{\log n})$

(1) The big-O closed-form runtime expression $T(n)$ for the recurrence $T(n) = 3T(n/3) + n$ is (assume n is a power of 3 and $T(1) = 1$)

A. $O(n)$

B. $O(n \log n)$

C. $O(n^3 \log n)$

D. $O(\sqrt[3]{n} \log n)$

E. $O(n \sqrt[3]{\log n})$

Same as T(n) = 2T(n / 2) + n,
Solve using tree method.

Exercise

For constant k, show T(n) = kT(n/ k) + n is O(nlgn)

(2) Two algorithms are developed based on the following template

```
function A(n : Z≥1 power of 2)
    if n = 1 then
        return 1
    end if
    ____________
    return A(n/2) + A(n/2)
end function
```

The missing part requires $F(n)$ time in Algorithm $\mathcal{A}_1$, and requires $G(n)$ time in Algorithm $\mathcal{A}_2$, where $F(n)$ and $G(n)$ are two functions of n.

If $F(n) = \Theta(G(n))$, then $\mathcal{A}_1(n) = \Theta(\mathcal{A}_2(n))$.

The above statement is

A. True

B. False

C. Possibly true/ Possible false

```
function A(n : Z≥1 power of 2)
    if n = 1 then
        return 1
    end if
    ______F or G(n)
    return A(n/2) + A(n/2)
end function
```

If $F(n) = \Theta(G(n))$**, then** $\mathcal{A}_1(n) = \Theta(\mathcal{A}_2(n))$**.**

Tree method:

Recurrence:

Tree:

Cost per level i:

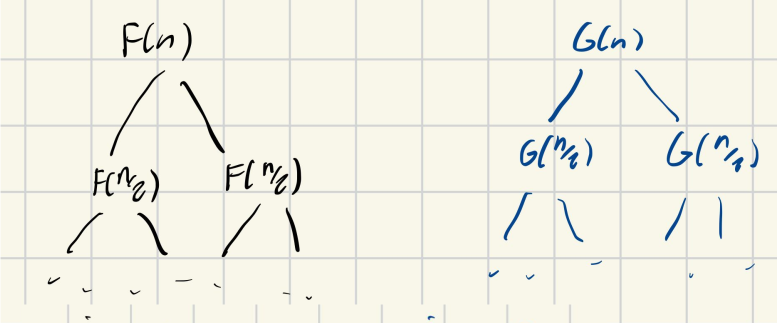
Levels:

Sum

```
function A(n : Z≥1 power of 2)
    if n = 1 then
        return 1
    end if
    ________ F or G(n)
    return A(n/2) + A(n/2)
end function
```

If $F(n) = \Theta(G(n))$, then $\mathcal{A}_1(n) = \Theta(\mathcal{A}_2(n))$.

Tree method:



wts:

Cost per level i: $2^i F(n/2^i)$ $\quad$ $2^i G(n/2^i)$

Levels: lg n

Sum:

$$\sum_{i=1}^{\lg n} 2^i F(n/2^i) \qquad \sum_{i=1}^{\lg n} 2^i G(n/2^i)$$

If $F(n) = \Theta(G(n))$, then $\mathcal{A}_1(n) = \Theta(\mathcal{A}_2(n))$.

$$\sum_{i=1}^{\lg n} 2^i F(n/2^i) \qquad \sum_{i=1}^{\lg n} 2^i G(n/2^i)$$

wts:

$$\sum_{i=1}^{\lg n} 2^i F(n/2^i) \in \Theta\left(\sum_{i=1}^{\lg n} 2^i G(n/2^i)\right)$$

This is true if ..

WTS:

$$\sum_{i=1}^{\lg n} 2^i F(n/2^i) \in \Theta\left(\sum_{i=1}^{\lg n} 2^i G(n/2^i)\right)$$

This is true if for each $i = 1, 2, \ldots, \lg n$

$$2^i F(n/2^i) \in \Theta\left(2^i G(n/2^i)\right)$$

And this is true if..

wts:

$$\sum_{i=1}^{\lg n} 2^i F(n/2^i) \in \Theta\left(\sum_{i=1}^{\lg n} 2^i G(n/2^i)\right)$$

This is true if for each $i = 1, 2, \dots, \lg n$

$$2^i F(n/2^i) \in \Theta(2^i G(n/2^i))$$

And this is true if

$$F(n/2^i) \in \Theta(G(n/2^i))$$

Which is true by our if condition!

If $F(n) = \Theta(G(n))$, **then** $\mathcal{A}_1(n) = \Theta(\mathcal{A}_2(n))$.

(3) Consider a sorted circular doubly-linked list where the head element points to the smallest element in the list. What is the time complexity to find the largest element in the list?

A. $O(1)$

B. $O(\log n)$

C. $O(n)$

D. $O(n \log n)$