[SWE2015-42] Introduction to Data Structures (자료구조개론)

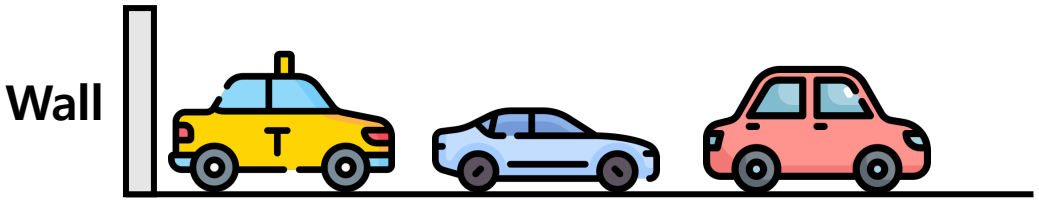
Queues

Department of Computer Science and Engineering

Instructor: Hankook Lee (이한국)

(Recap) What is Stack?

- Stack is a collection of elements that are inserted and removed according to **the last-in first-out (LIFO) principle**
  - Insertion, deletion, and information access are only possible at the top on the stack
  - **LIFO:** the last added element will be the first element to be removed



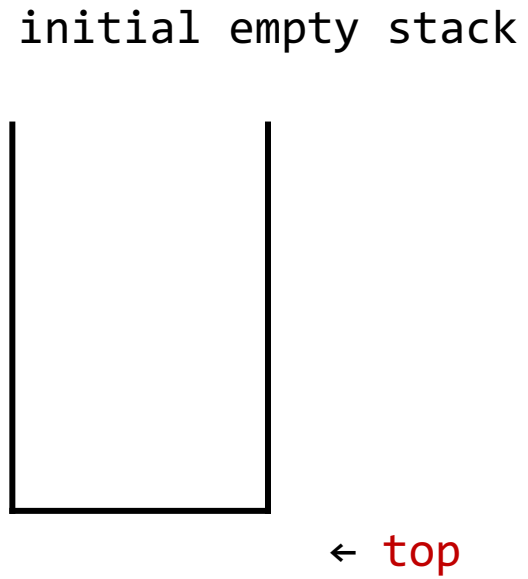
← This blue book was stacked last

← This yellow book was stacked first

(Recap) What is Stack?

- Stack is a collection of elements that are inserted and removed according to **the last-in first-out (LIFO) principle**
  - Insertion, deletion, and information access are only possible at the top on the stack
  - **LIFO:** the last added element will be the first element to be removed

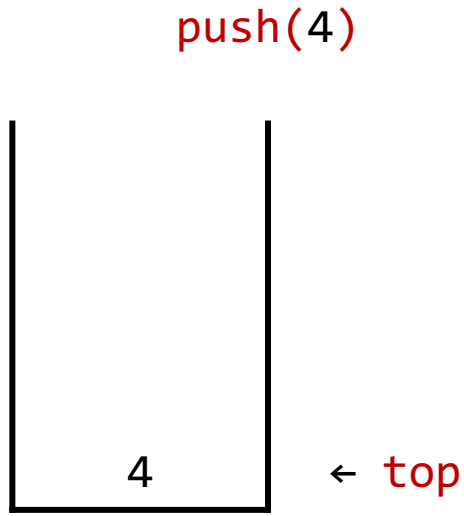
- Main components
  - `top` - represent the top of the stack
    - This can be represented by index or pointer
  - `push()` - insert an element to the top of the stack
  - `pop()` - delete the topmost element from the stack
  - `peek()` - return the value of topmost element of the stack



(Recap) What is Stack?

- Stack is a collection of elements that are inserted and removed according to **the last-in first-out (LIFO) principle**
  - Insertion, deletion, and information access are only possible at the top on the stack
  - **LIFO:** the last added element will be the first element to be removed

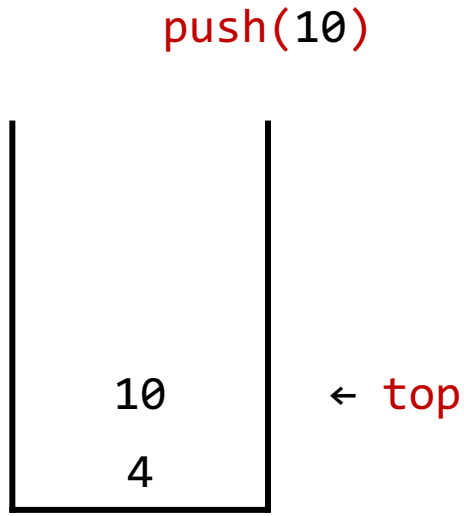
- Main components
  - `top` - represent the top of the stack
    - This can be represented by index or pointer
  - `push()` - insert an element to the top of the stack
  - `pop()` - delete the topmost element from the stack
  - `peek()` - return the value of topmost element of the stack



(Recap) What is Stack?

- Stack is a collection of elements that are inserted and removed according to **the last-in first-out (LIFO) principle**
  - Insertion, deletion, and information access are only possible at the top on the stack
  - **LIFO:** the last added element will be the first element to be removed

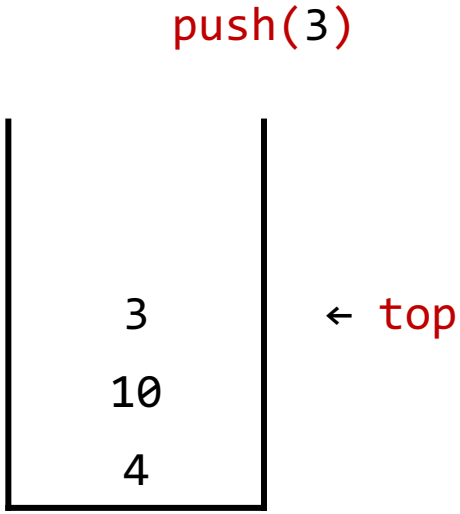
- Main components
  - `top` - represent the top of the stack
    - This can be represented by index or pointer
  - `push()` - insert an element to the top of the stack
  - `pop()` - delete the topmost element from the stack
  - `peek()` - return the value of topmost element of the stack



(Recap) What is Stack?

- Stack is a collection of elements that are inserted and removed according to **the last-in first-out (LIFO) principle**
  - Insertion, deletion, and information access are only possible at the top on the stack
  - **LIFO:** the last added element will be the first element to be removed

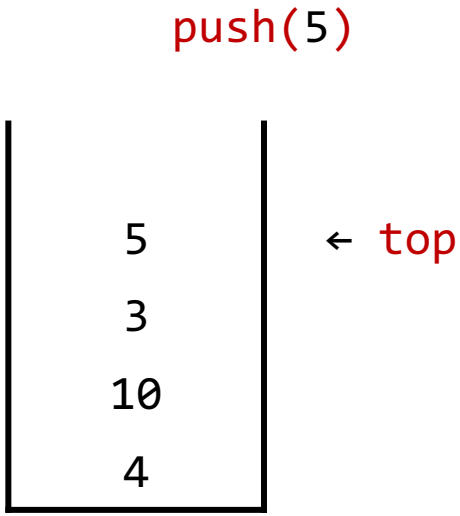
- Main components
  - `top` - represent the top of the stack
    - This can be represented by index or pointer
  - `push()` - insert an element to the top of the stack
  - `pop()` - delete the topmost element from the stack
  - `peek()` - return the value of topmost element of the stack



(Recap) What is Stack?

- Stack is a collection of elements that are inserted and removed according to **the last-in first-out (LIFO) principle**
  - Insertion, deletion, and information access are only possible at the top on the stack
  - **LIFO:** the last added element will be the first element to be removed

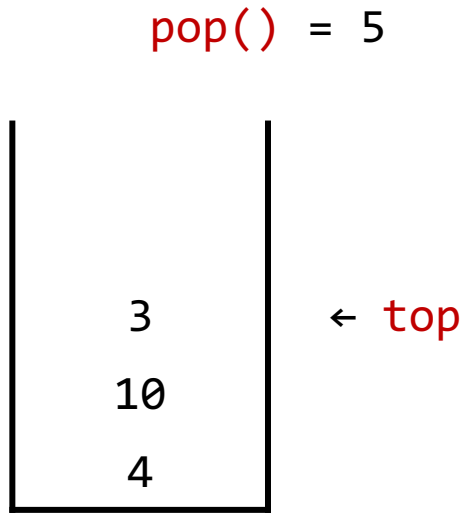
- Main components
  - `top` - represent the top of the stack
    - This can be represented by index or pointer
  - `push()` - insert an element to the top of the stack
  - `pop()` - delete the topmost element from the stack
  - `peek()` - return the value of topmost element of the stack



(Recap) What is Stack?

- Stack is a collection of elements that are inserted and removed according to **the last-in first-out (LIFO) principle**
  - Insertion, deletion, and information access are only possible at the top on the stack
  - **LIFO:** the last added element will be the first element to be removed

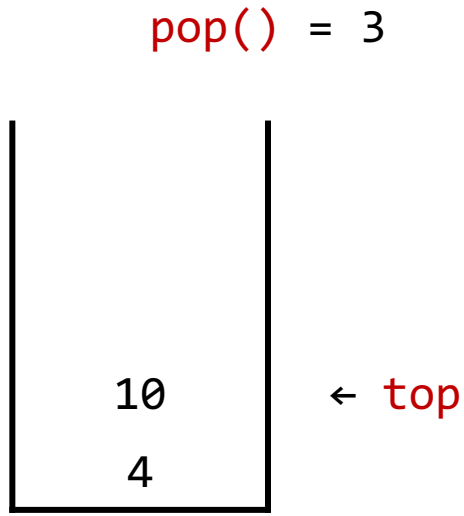
- Main components
  - `top` - represent the top of the stack
    - This can be represented by index or pointer
  - `push()` - insert an element to the top of the stack
  - `pop()` - delete the topmost element from the stack
  - `peek()` - return the value of topmost element of the stack



(Recap) What is Stack?

- Stack is a collection of elements that are inserted and removed according to **the last-in first-out (LIFO) principle**
  - Insertion, deletion, and information access are only possible at the top on the stack
  - **LIFO:** the last added element will be the first element to be removed

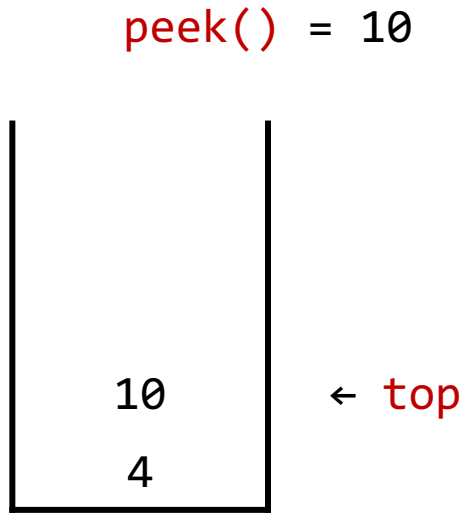
- Main components
  - `top` - represent the top of the stack
    - This can be represented by index or pointer
  - `push()` - insert an element to the top of the stack
  - `pop()` - delete the topmost element from the stack
  - `peek()` - return the value of topmost element of the stack



(Recap) What is Stack?

- Stack is a collection of elements that are inserted and removed according to **the last-in first-out (LIFO) principle**
  - Insertion, deletion, and information access are only possible at the top on the stack
  - **LIFO:** the last added element will be the first element to be removed

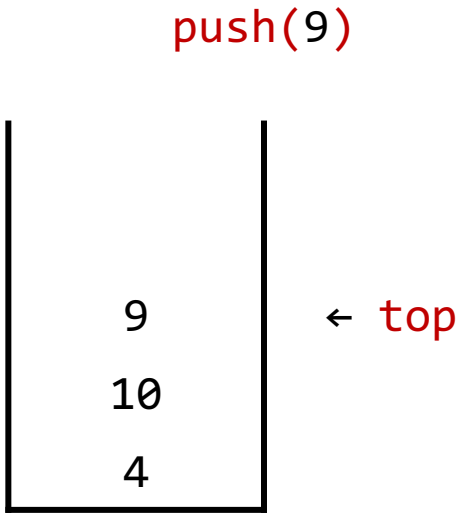
- Main components
  - `top` - represent the top of the stack
    - This can be represented by index or pointer
  - `push()` - insert an element to the top of the stack
  - `pop()` - delete the topmost element from the stack
  - `peek()` - return the value of topmost element of the stack



(Recap) What is Stack?

- Stack is a collection of elements that are inserted and removed according to **the last-in first-out (LIFO) principle**
  - Insertion, deletion, and information access are only possible at the top on the stack
  - **LIFO:** the last added element will be the first element to be removed

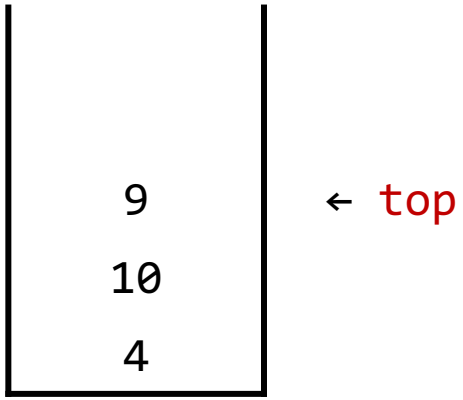
- Main components
  - `top` - represent the top of the stack
    - This can be represented by index or pointer
  - `push()` - insert an element to the top of the stack
  - `pop()` - delete the topmost element from the stack
  - `peek()` - return the value of topmost element of the stack



(Recap) What is Stack?

- Stack is a collection of elements that are inserted and removed according to **the last-in first-out (LIFO) principle**
  - Insertion, deletion, and information access are only possible at the top on the stack
  - **LIFO:** the last added element will be the first element to be removed

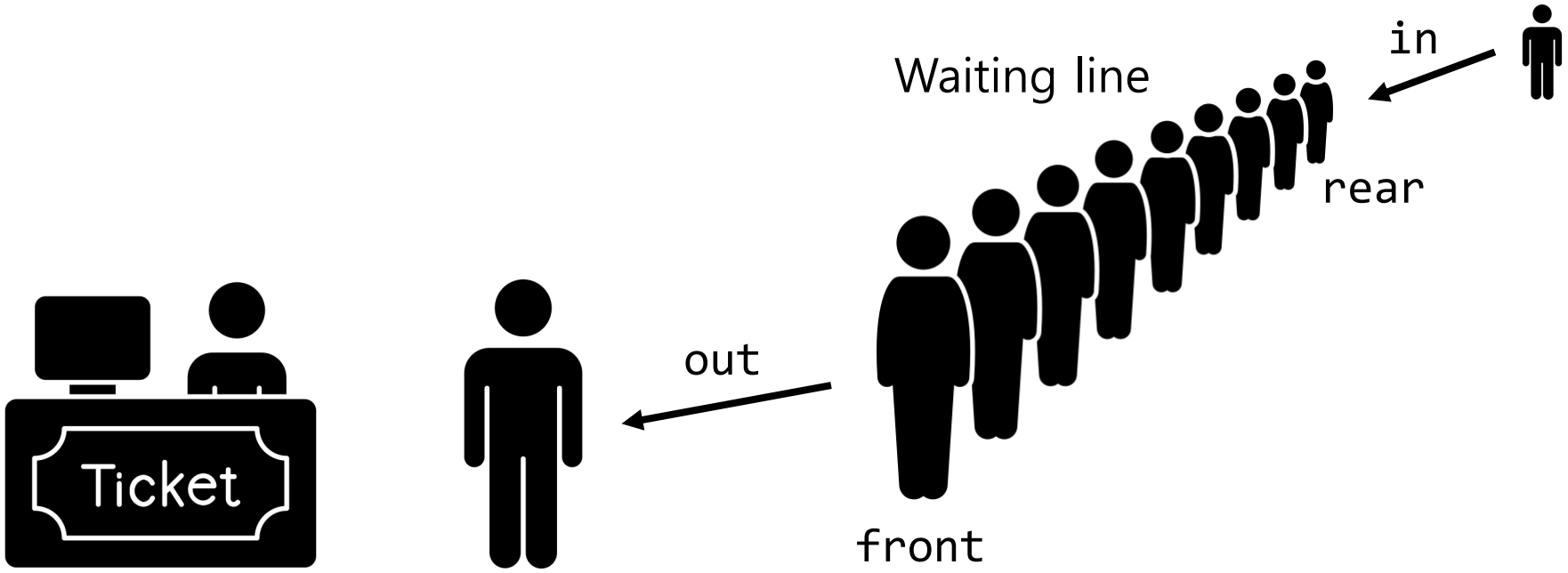
- Main components
  - `top` - represent the top of the stack
    - This can be represented by index or pointer
  - `push()` - insert an element to the top of the stack
  - `pop()` - delete the topmost element from the stack
  - `peek()` - return the value of topmost element of the stack



  - **Note.** We cannot access items other than at the top (by definition)

What is Queue?

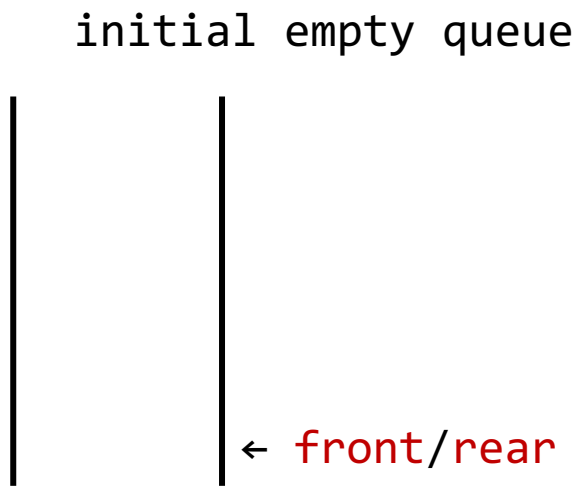
- Queue is a collection of elements that are inserted and removed according to **the first-in first-out (FIFO) principle**
  - **FIFO:** the first added element will be the first element to be removed
  - The elements in a queue are added at one end (called `rear`)
  - The elements in a queue are removed from the other end (called `front`)



What is Queue?

- Queue is a collection of elements that are inserted and removed according to **the first-in first-out (FIFO) principle**
  - **FIFO:** the first added element will be the first element to be removed
  - The elements in a queue are added at one end (called `rear`)
  - The elements in a queue are removed from the other end (called `front`)

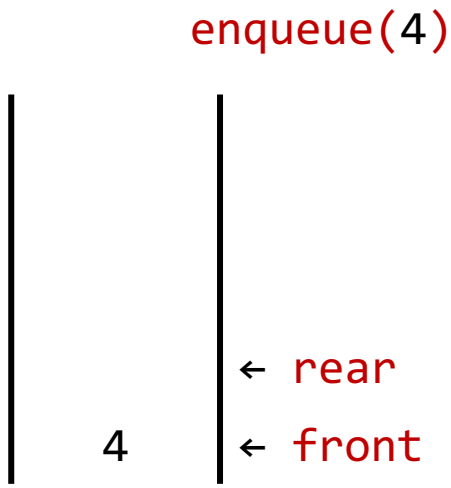
- Main components
  - `front` - the position where deletions can be done
  - `rear` - the position where insertions can be done
  - `enqueue()` - insert an element at the rear of the queue
  - `dequeue()` - delete the most front element of the queue
  - `peek()` - return the value of the most front element



What is Queue?

- Queue is a collection of elements that are inserted and removed according to **the first-in first-out (FIFO) principle**
  - **FIFO:** the first added element will be the first element to be removed
  - The elements in a queue are added at one end (called `rear`)
  - The elements in a queue are removed from the other end (called `front`)

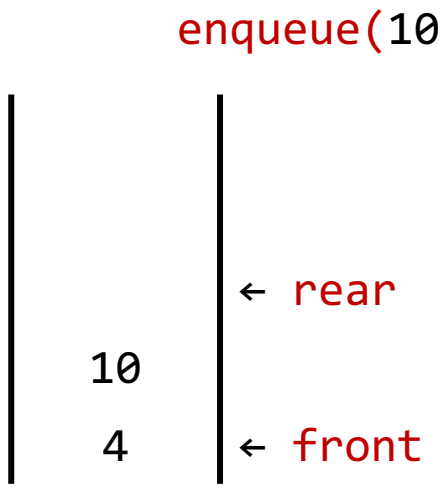
- Main components
  - `front` - the position where deletions can be done
  - `rear` - the position where insertions can be done
  - `enqueue()` - insert an element at the rear of the queue
  - `dequeue()` - delete the most front element of the queue
  - `peek()` - return the value of the most front element



What is Queue?

- Queue is a collection of elements that are inserted and removed according to **the first-in first-out (FIFO) principle**
  - **FIFO:** the first added element will be the first element to be removed
  - The elements in a queue are added at one end (called `rear`)
  - The elements in a queue are removed from the other end (called `front`)

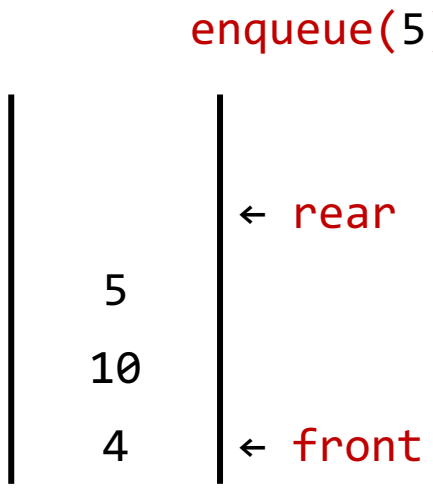
- Main components
  - `front` - the position where deletions can be done
  - `rear` - the position where insertions can be done
  - `enqueue()` - insert an element at the rear of the queue
  - `dequeue()` - delete the most front element of the queue
  - `peek()` - return the value of the most front element



What is Queue?

- Queue is a collection of elements that are inserted and removed according to **the first-in first-out (FIFO) principle**
  - **FIFO:** the first added element will be the first element to be removed
  - The elements in a queue are added at one end (called `rear`)
  - The elements in a queue are removed from the other end (called `front`)

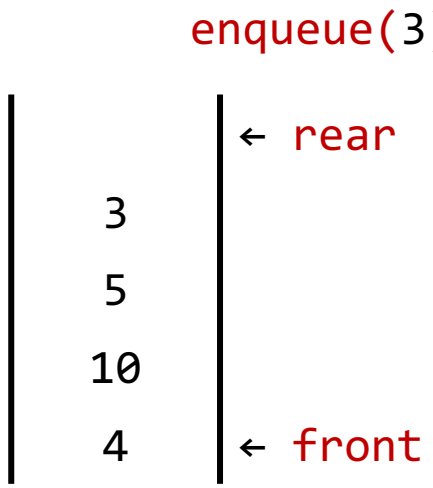
- Main components
  - `front` - the position where deletions can be done
  - `rear` - the position where insertions can be done
  - `enqueue()` - insert an element at the rear of the queue
  - `dequeue()` - delete the most front element of the queue
  - `peek()` - return the value of the most front element



What is Queue?

- Queue is a collection of elements that are inserted and removed according to **the first-in first-out (FIFO) principle**
  - **FIFO:** the first added element will be the first element to be removed
  - The elements in a queue are added at one end (called `rear`)
  - The elements in a queue are removed from the other end (called `front`)

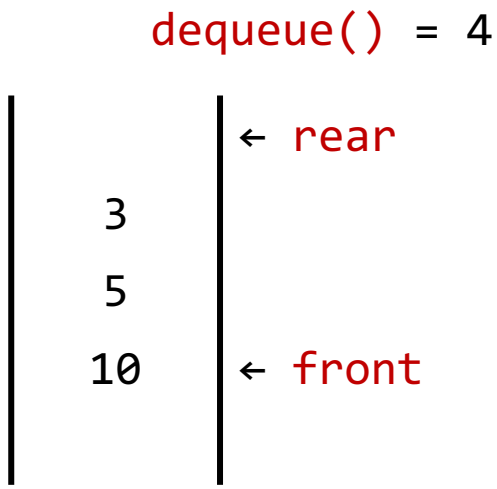
- Main components
  - `front` - the position where deletions can be done
  - `rear` - the position where insertions can be done
  - `enqueue()` - insert an element at the rear of the queue
  - `dequeue()` - delete the most front element of the queue
  - `peek()` - return the value of the most front element



What is Queue?

- Queue is a collection of elements that are inserted and removed according to **the first-in first-out (FIFO) principle**
  - **FIFO:** the first added element will be the first element to be removed
  - The elements in a queue are added at one end (called `rear`)
  - The elements in a queue are removed from the other end (called `front`)

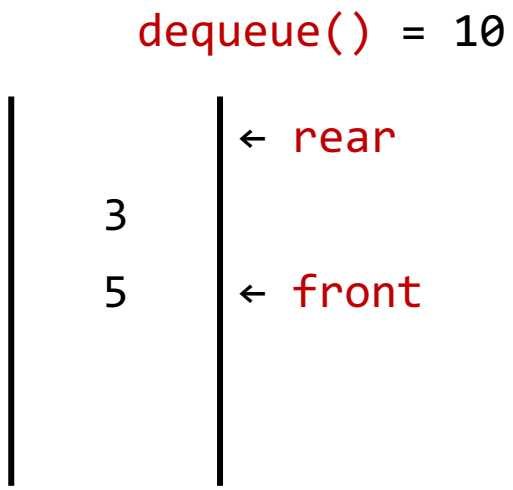
- Main components
  - `front` - the position where deletions can be done
  - `rear` - the position where insertions can be done
  - `enqueue()` - insert an element at the rear of the queue
  - `dequeue()` - delete the most front element of the queue
  - `peek()` - return the value of the most front element



What is Queue?

- Queue is a collection of elements that are inserted and removed according to **the first-in first-out (FIFO) principle**
  - **FIFO:** the first added element will be the first element to be removed
  - The elements in a queue are added at one end (called `rear`)
  - The elements in a queue are removed from the other end (called `front`)

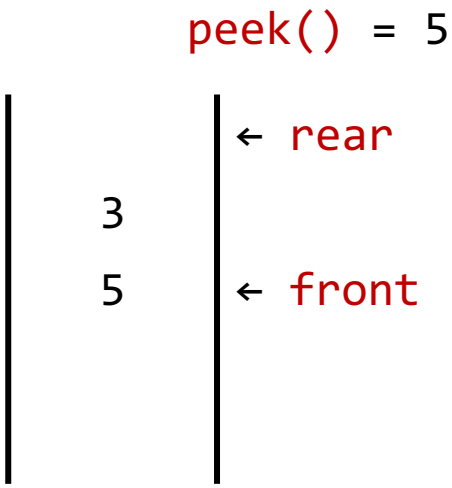
- Main components
  - `front` - the position where deletions can be done
  - `rear` - the position where insertions can be done
  - `enqueue()` - insert an element at the rear of the queue
  - `dequeue()` - delete the most front element of the queue
  - `peek()` - return the value of the most front element



What is Queue?

- Queue is a collection of elements that are inserted and removed according to **the first-in first-out (FIFO) principle**
  - **FIFO:** the first added element will be the first element to be removed
  - The elements in a queue are added at one end (called `rear`)
  - The elements in a queue are removed from the other end (called `front`)

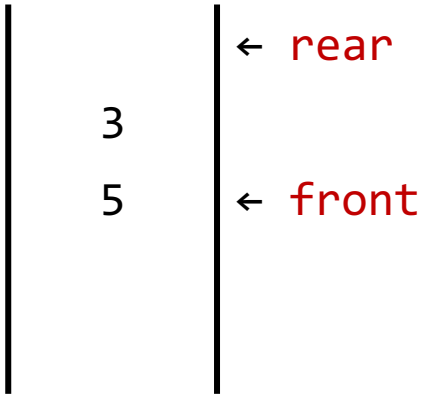
- Main components
  - `front` - the position where deletions can be done
  - `rear` - the position where insertions can be done
  - `enqueue()` - insert an element at the rear of the queue
  - `dequeue()` - delete the most front element of the queue
  - `peek()` - return the value of the most front element



What is Queue?

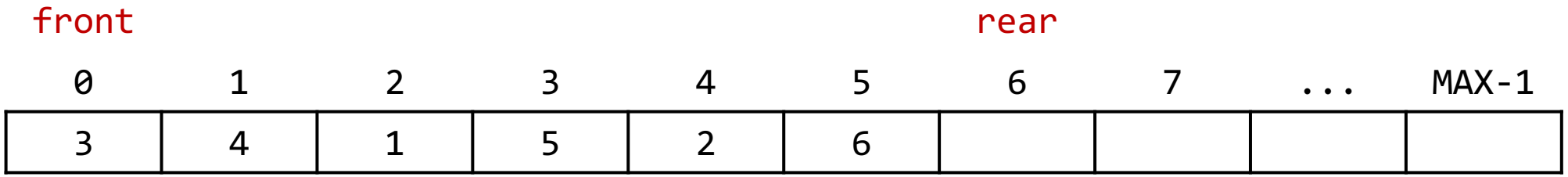
- Queue is a collection of elements that are inserted and removed according to **the first-in first-out (FIFO) principle**
  - **FIFO:** the first added element will be the first element to be removed
  - The elements in a queue are added at one end (called `rear`)
  - The elements in a queue are removed from the other end (called `front`)

- Main components
  - `front` - the position where deletions can be done
  - `rear` - the position where insertions can be done
  - `enqueue()` - insert an element at the rear of the queue
  - `dequeue()` - delete the most front element of the queue
  - `peek()` - return the value of the most front element
  - **Note.** We cannot access items other than at the front (by definition)

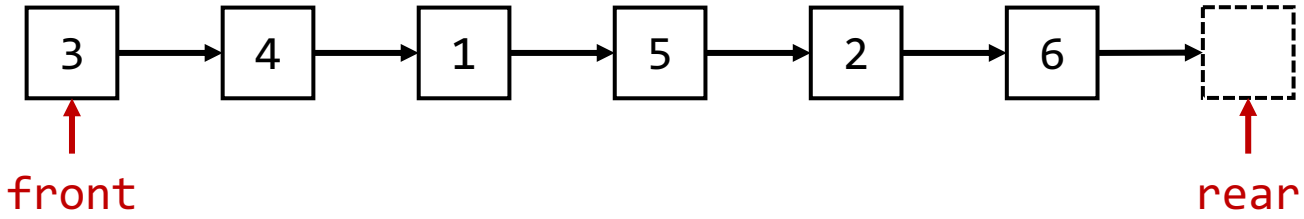


Queues - Implementation

- You have **two options** for queue implementation
  1. **Use Array**

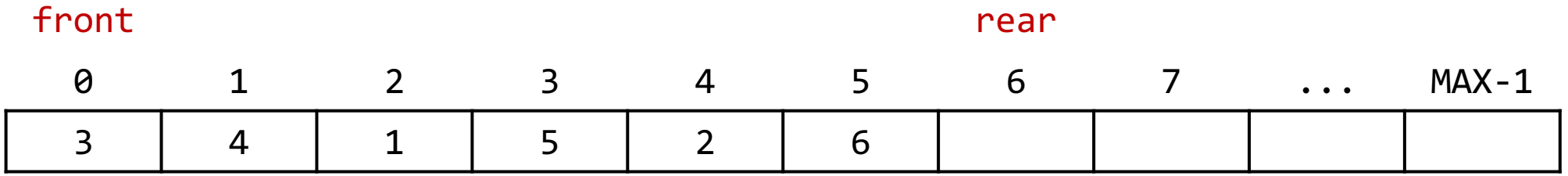


  2. **Use Linked list**



Queues - Array-based Implementation

- How to implement a queue using **the array structure**?

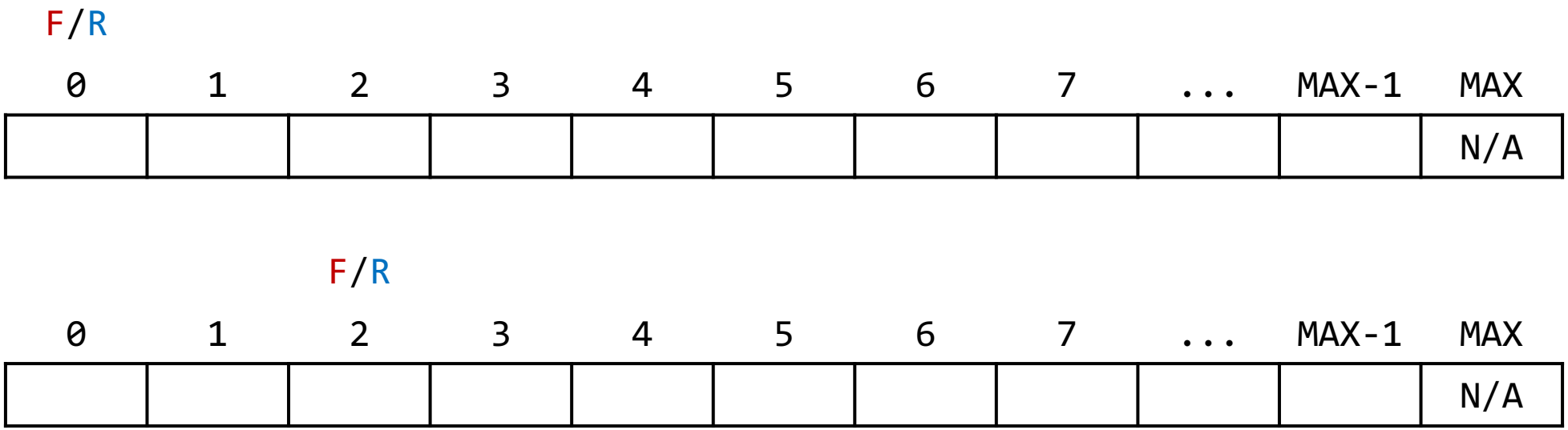



```c
#define MAX_SIZE 100
typedef struct _Queue {
    int front, rear;     // front, rear indices
    int items[MAX_SIZE]; // array for queue elements
} Queue;

Queue createQueue();
void removeQueue(Queue *queue); // nothing to do here
bool isEmpty(Queue *queue);
bool isFull(Queue *queue);
void enqueue(Queue *queue, int item);
int dequeue(Queue *queue);
int peek(Queue *queue);
```

Queues - Array-based Implementation

- How to implement a queue using **the array structure**?
  - `front(F)` - the position where deletions can be done
  - `rear(R)` - the position where insertions can be done

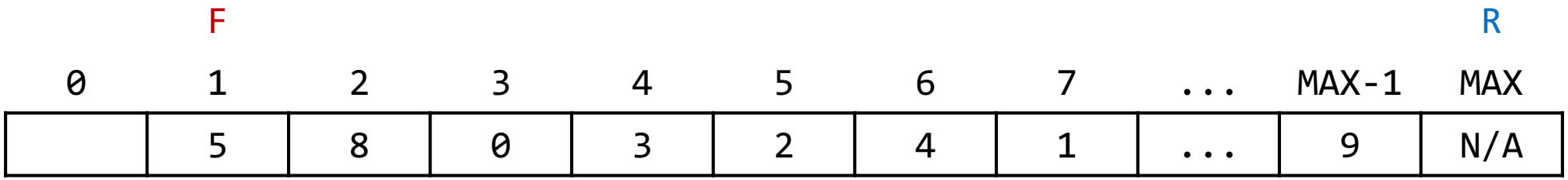
  `(Q)` What is the empty state?
    - When `front == rear`

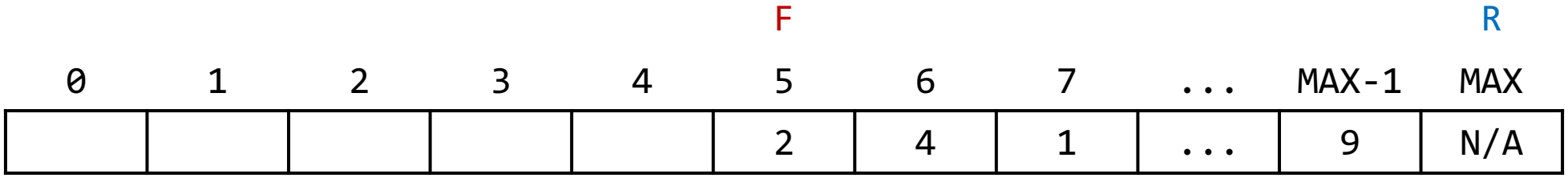


Queues - Array-based Implementation

- How to implement a queue using **the array structure**?
  - `front(F)` - the position where deletions can be done
  - `rear(R)` - the position where insertions can be done

  `(Q)` What is the full state?
    - When `rear == MAX_SIZE`





Queues - Array-based Implementation

- How to implement a queue using **the array structure**?

```
Queue createQueue() {
    // Declare a new queue
    // Set the initial value for the front index
    // Set the initial value for the rear index
    // Return the new queue
}

bool isEmpty(Queue *queue) {
    // Check whether queue is empty or not
}

bool isFull(Queue *queue) {
    // Check whether queue is full or not
}
```

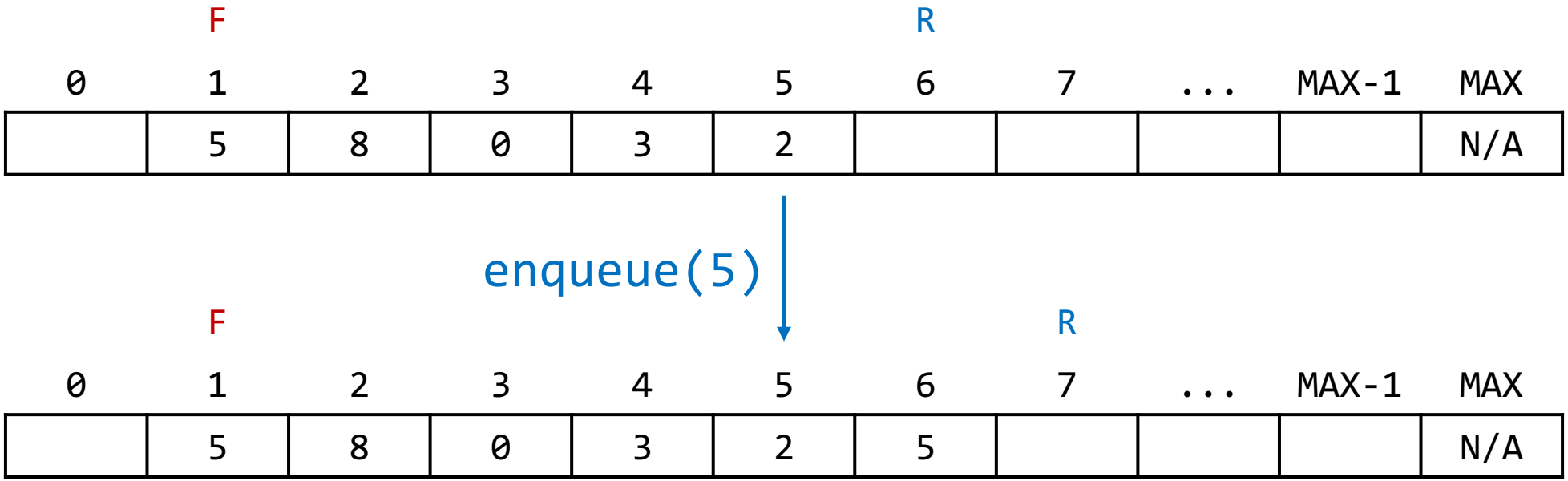
Queues - Array-based Implementation

- How to implement a queue using **the array structure**?

```
Queue createQueue() {
    Queue newQueue;      // Declare a new queue
    newQueue.front = 0; // Set the initial value for the front index
    newQueue.rear = 0;  // Set the initial value for the rear index
    return newQueue;    // Return the new queue
}

bool isEmpty(Queue *queue) {
    return newQueue->front == newQueue->rear; // Check whether queue is empty or not
}

bool isFull(Queue *queue) {
    return newQueue->rear == MAX_SIZE; // Check whether queue is full or not
}
```

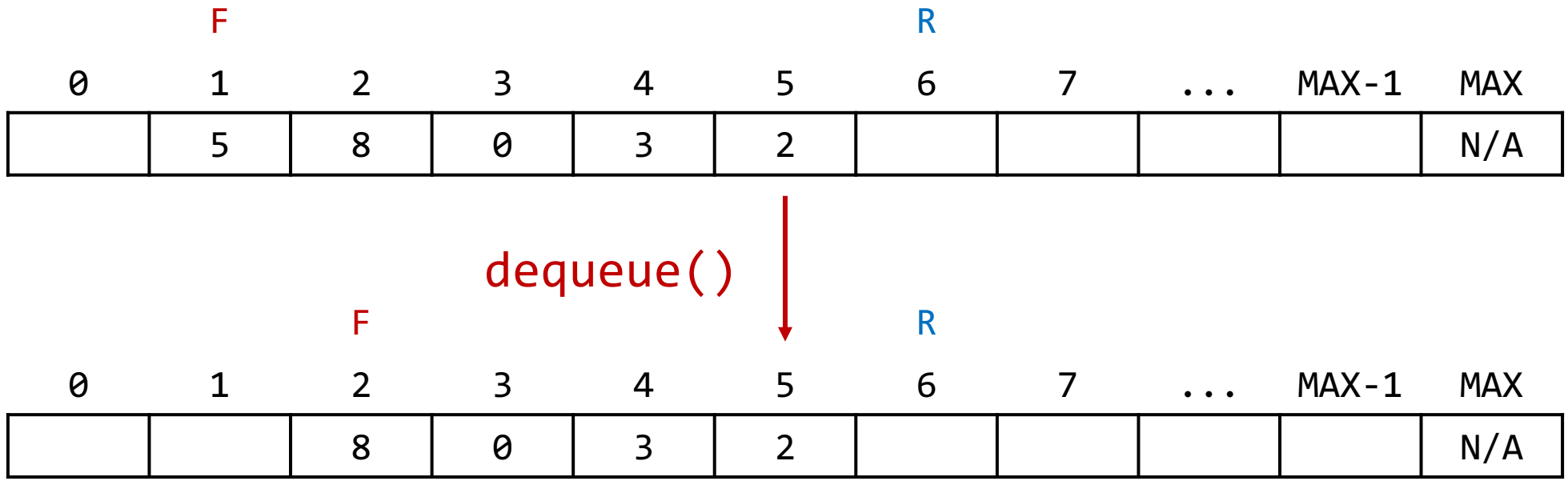
Queues - Array-based Implementation

- How to implement a queue using **the array structure**?
  - `enqueue()` puts an item, and then increases the `rear` index



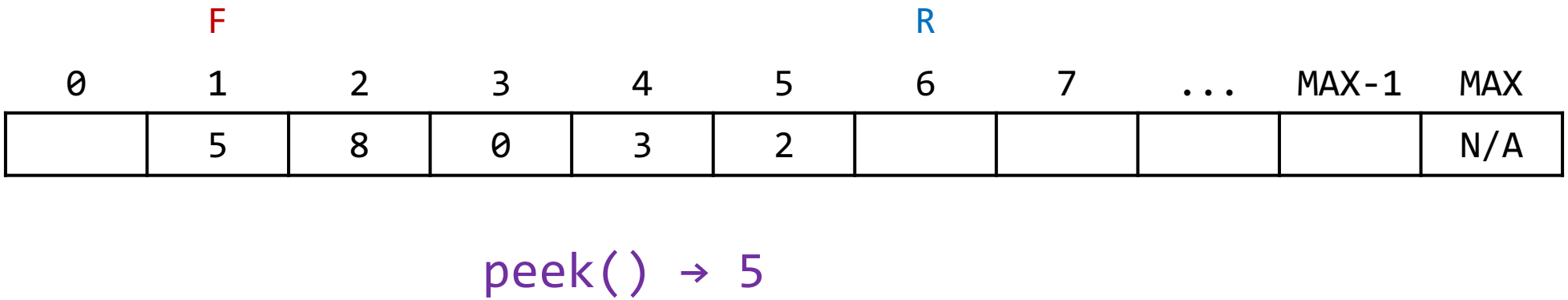
Queues - Array-based Implementation

- How to implement a queue using **the array structure**?
  - `enqueue()` puts an item, and then increases the `rear` index
  - `dequeue()` reads the front item, and then increases the `front` index



Queues - Array-based Implementation

- How to implement a queue using **the array structure**?
  - `enqueue()` puts an item, and then increases the `rear` index
  - `dequeue()` reads the front item, and then increases the `front` index
  - `peek()` simply reads and returns the front item



`peek()` → 5

Queues - Array-based Implementation

- How to implement a queue using **the array structure**?

```
void enqueue(Queue *queue, int item) {
    // Put item into queue at rear
    // Increase rear index
}

int dequeue(Queue *queue) {
    // Read front element
    // Increase front index
    // Return previous front element
}

int peek(Queue *queue) {
    // Return front element
}
```

Queues - Array-based Implementation

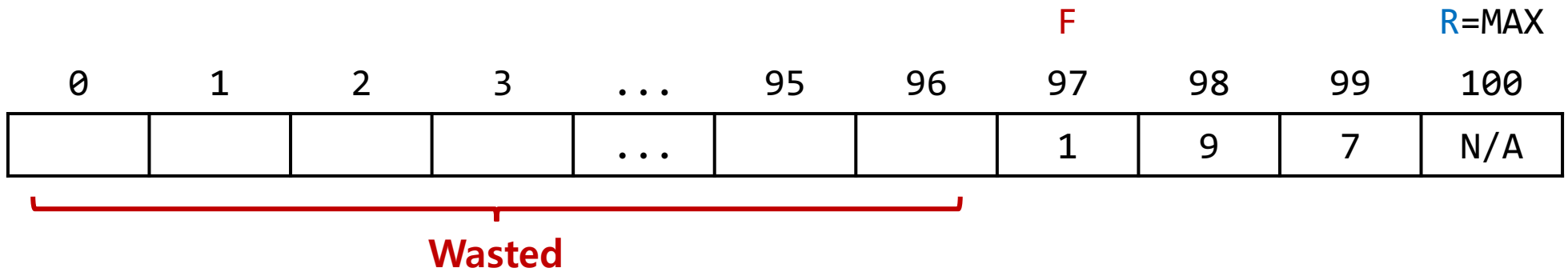
- How to implement a queue using **the array structure**?

```
void enqueue(Queue *queue, int item) {
    queue->items[queue->rear] = item; // Put item into queue at rear
    queue->rear ++;                   // Increase rear index
}

int dequeue(Queue *queue) {
    int item = queue[queue->front]; // Read front element
    queue->front ++;                // Increase front index
    return item;                    // Return previous front element
}

int peek(Queue *queue) {
    return queue[queue->front]; // Return front element
}
```

- **Corner cases:** You must check a structure is empty or full when insert or delete an element from a structure

Queues – Array-based Implementation

- How to implement a queue using **the array structure**?
  - **An issue:** This implementation cannot fully utilize memory



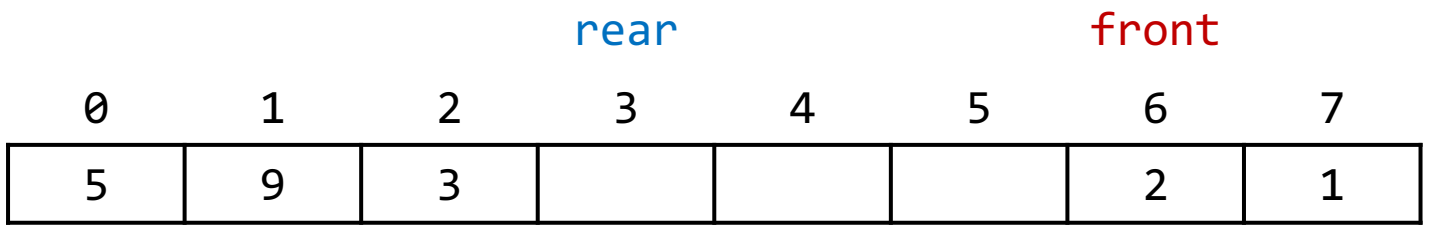
  - How to solve the issue?

    (Approach 1) Use a circular queue

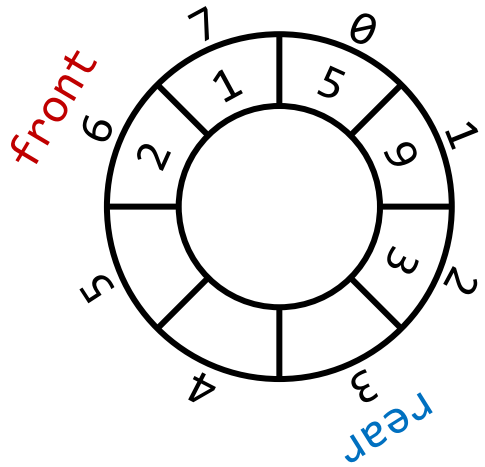
    (Approach 2) Use linked list for the implementation

Queues - Array-based Implementation

- How to implement a queue using **the array structure**?
  - **Circular queue:** conceptually, the first index comes right after the last index



Equivalent



(Q) How to compute the next index?

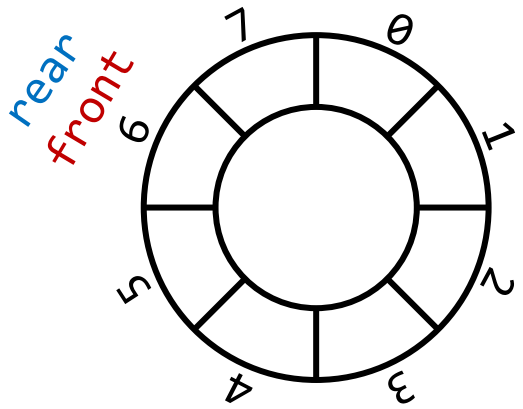
(A) `next_index = (curr_index + 1) % SIZE`

- `7 = (6 + 1) % 8`
- `0 = (7 + 1) % 8`
- `1 = (0 + 1) % 8`

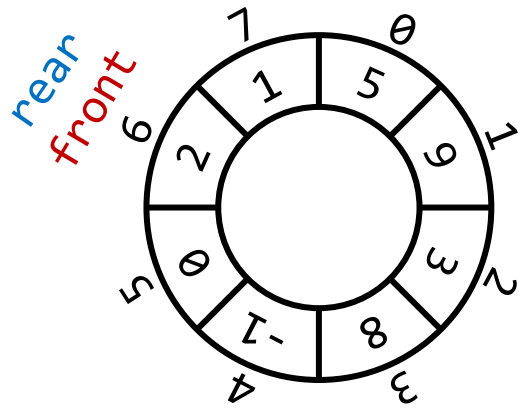
Queues – Array-based Implementation

- How to implement a queue using **the array structure**?
  - **Circular queue:** conceptually, the first index comes right after the last index

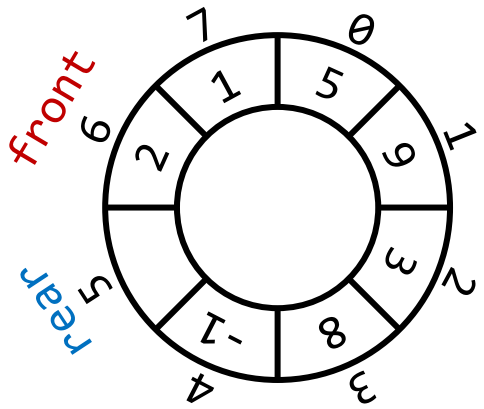
  (Q) How to distinguish empty and full states?



`rear == front`



`rear == front`



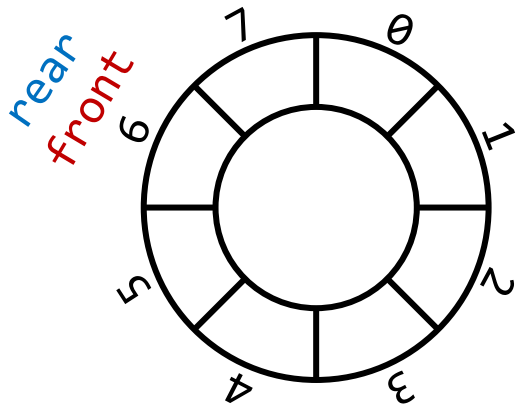
`(rear+1)%SIZE == front`

Queues - Array-based Implementation

- How to implement a queue using **the array structure**?
  - **Circular queue:** conceptually, the first index comes right after the last index

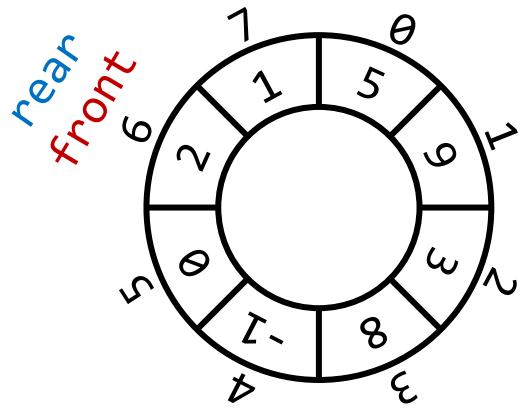
(Q) How to distinguish empty and full states?

(A) Simply, consider `SIZE-1` as the full size



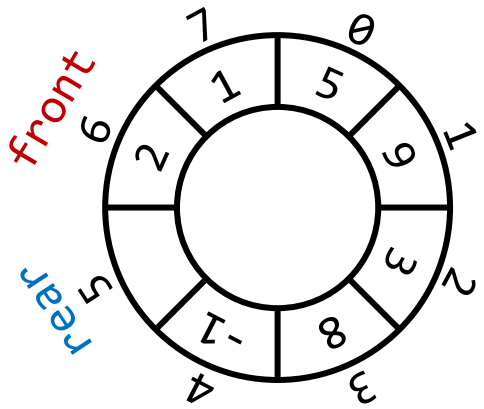
`rear == front`

EMPTY



`rear == front`

NOT REACHABLE STATE

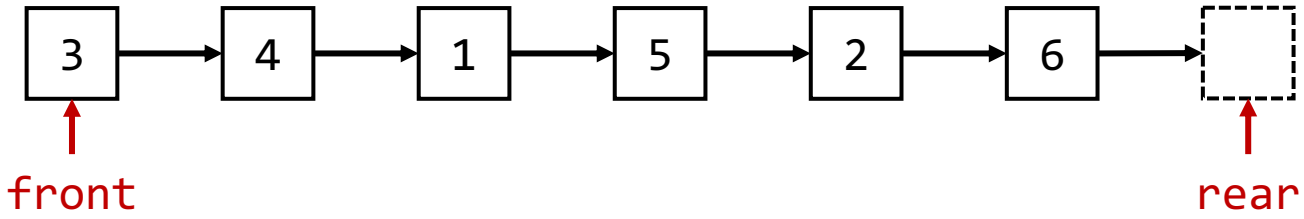


`(rear+1)%SIZE == front`

FULL

Queues - List-based Implementation

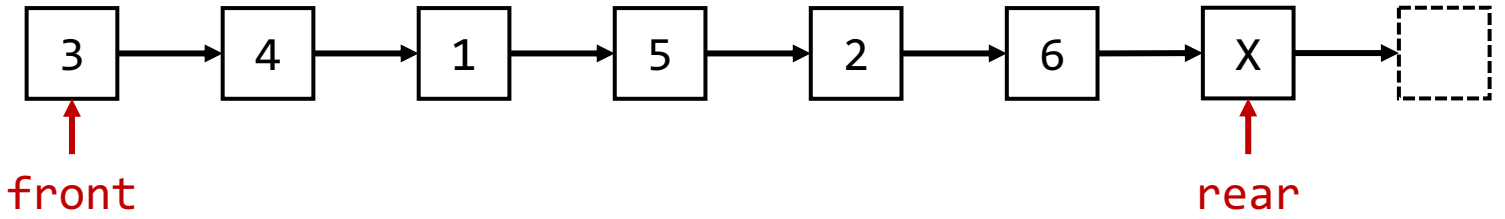
- How to implement a queue using **the linked list structure**?



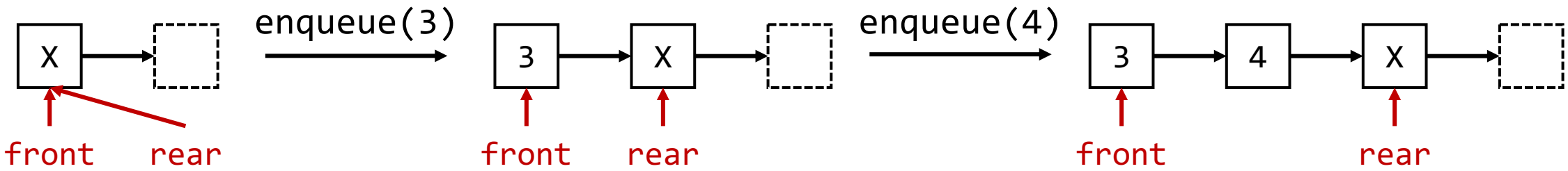

```
typedef struct _Node { int item; struct _Node *next; } Node;
typedef struct _Queue {
    Node *front, *rear; // front & rear pointers
} Queue;

Stack createQueue();
void removeQueue(Queue *queue); // must remove dynamically allocated variables
bool isEmpty(Queue *queue);
bool isFull(Queue *queue);
void enqueue(Queue *queue, int item);
int dequeue(Queue *queue);
int peek(Queue *queue);
```

Queues - List-based Implementation

- How to implement a queue using **the linked list structure**?



  - For easy implementation, recommend to use a dummy node X at the `rear`

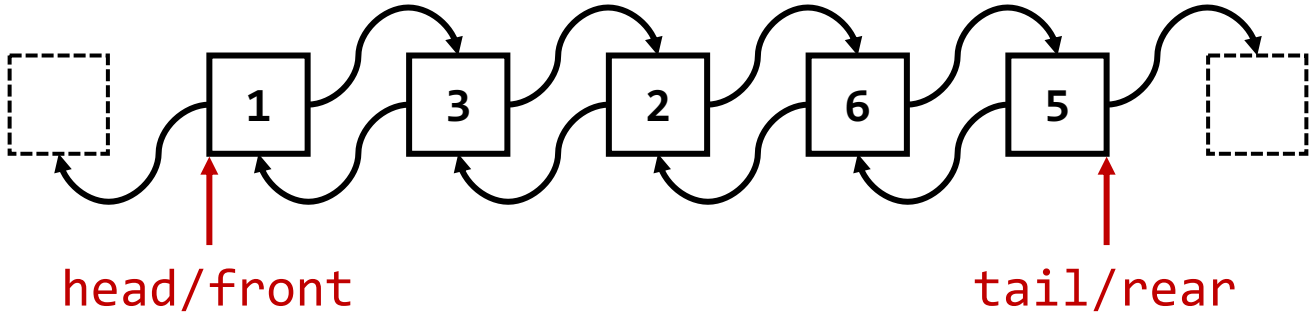


EMPTY

Queues - Variants

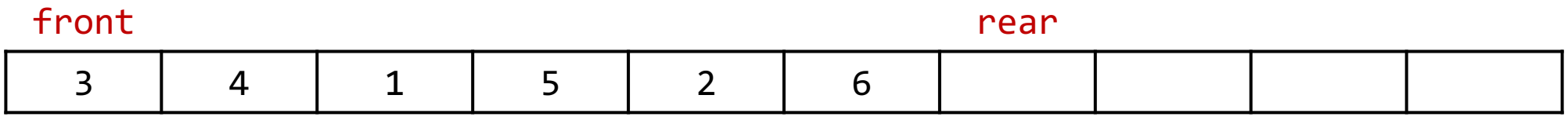
(Variant 1) Deque

- The elements in a deque can be added and removed at both the front and the rear
- This is equivalent to a head-tail doubly-linked list



(Variant 2) Priority Queue

- An element with higher priority is processed before an element with a lower priority

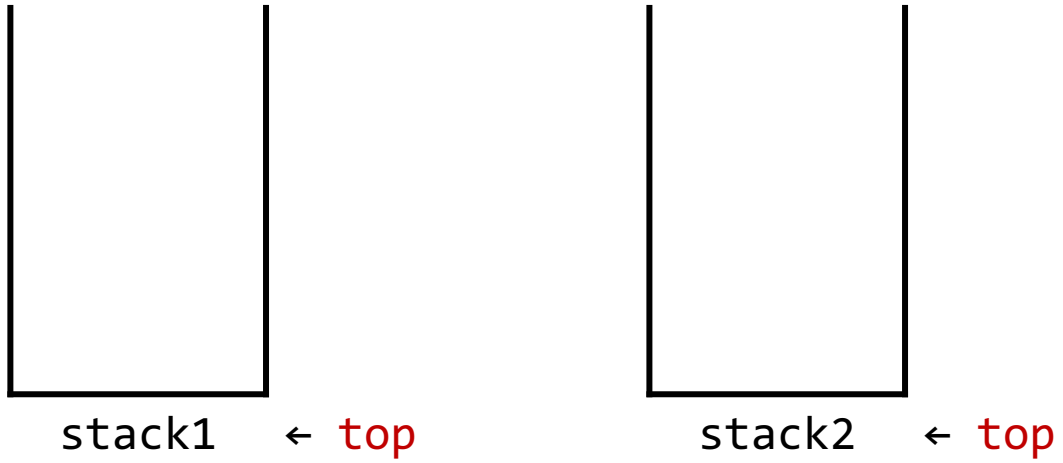


If a smaller value has a higher priority, `dequeue()` removes `1` first

Queues - Problem Solving Practice

- Problem: **Queue Implementation Using Two Stacks**
  - You have two stacks: `stack1`, `stack2`
  - You can use only stack operations: `push()`, `pop()`, `peek()`, `isEmpty()`

  (Q) How to implement the queue operations: `enqueue()`, `dequeue()`?

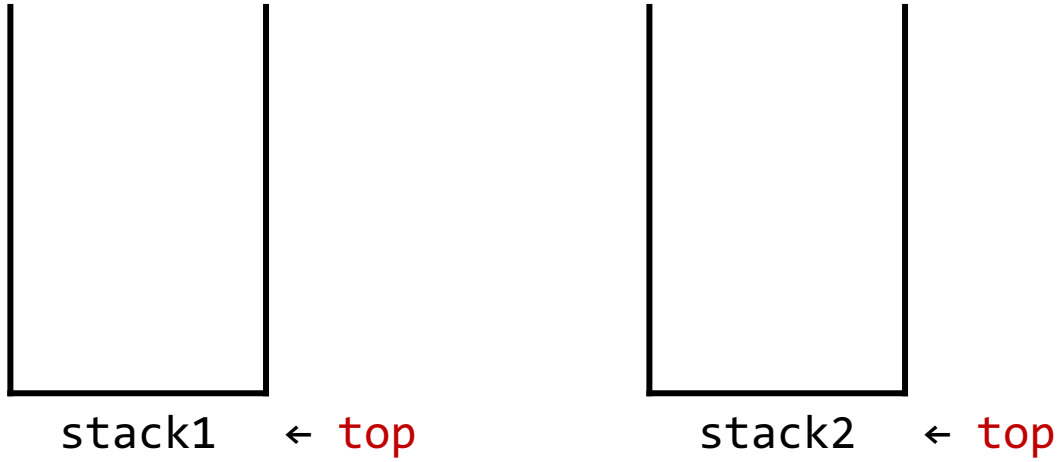


Queues - Problem Solving Practice

- Problem: **Queue Implementation Using Two Stacks**
  - You have two stacks: `stack1`, `stack2`
  - You can use only stack operations: `push()`, `pop()`, `peek()`, `isEmpty()`

(Q) How to implement the queue operations: `enqueue()`, `dequeue()`?

(Step 1) Think properties of stack and queue

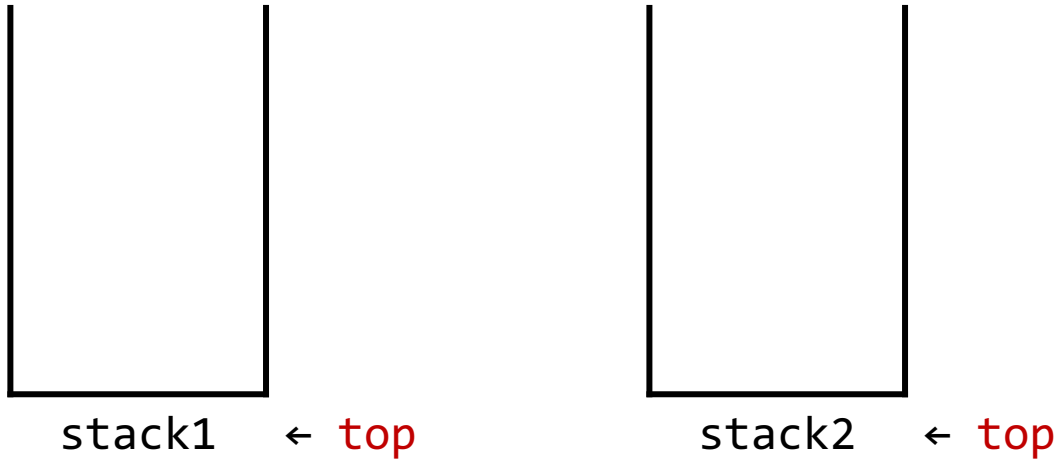


Queues - Problem Solving Practice

- Problem: **Queue Implementation Using Two Stacks**
  - You have two stacks: `stack1`, `stack2`
  - You can use only stack operations: `push()`, `pop()`, `peek()`, `isEmpty()`

  (Q) How to implement the queue operations: `enqueue()`, `dequeue()`?

  (Step 1) Think properties of stack and queue
    - Stack and queue follow the LIFO and FIFO principles, respectively



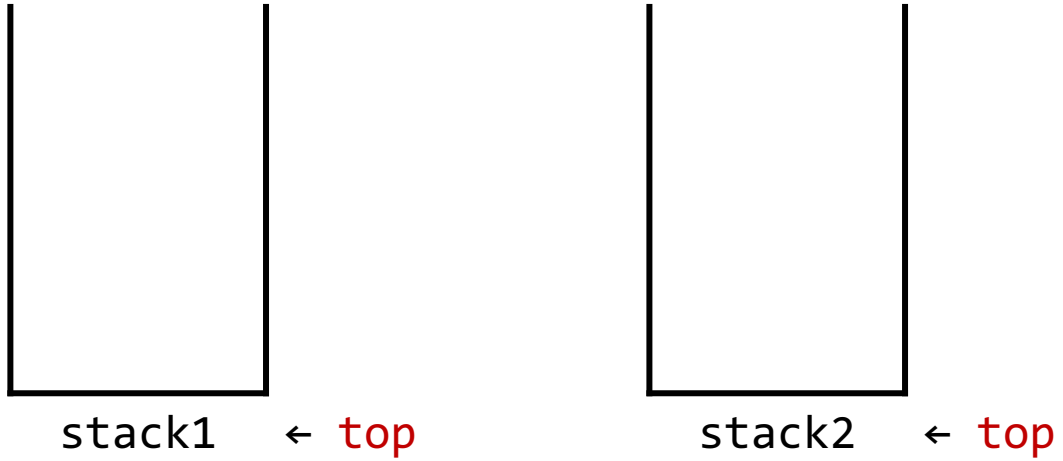
Queues - Problem Solving Practice

- Problem: **Queue Implementation Using Two Stacks**
  - You have two stacks: `stack1`, `stack2`
  - You can use only stack operations: `push()`, `pop()`, `peek()`, `isEmpty()`

(Q) How to implement the queue operations: `enqueue()`, `dequeue()`?

(Step 1) Think properties of stack and queue

(Step 2) Find a relation between the properties



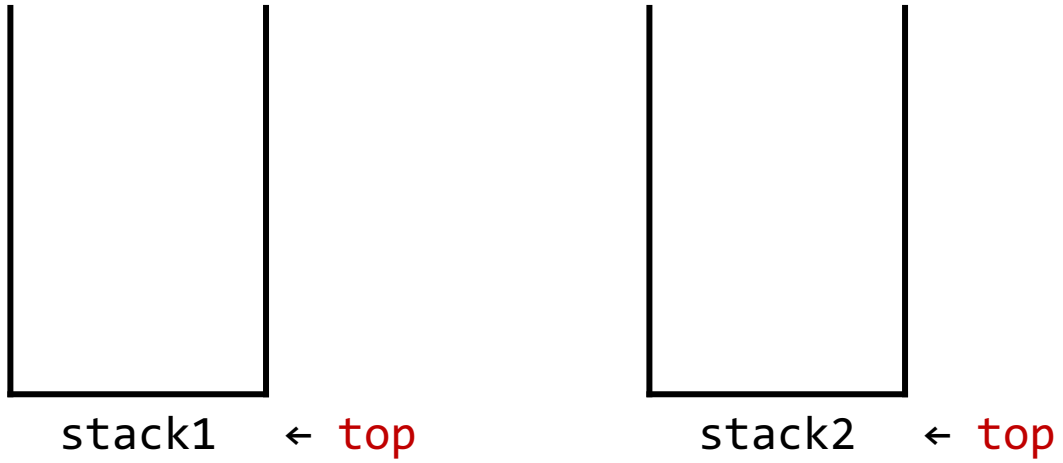
Queues - Problem Solving Practice

- Problem: **Queue Implementation Using Two Stacks**
  - You have two stacks: `stack1`, `stack2`
  - You can use only stack operations: `push()`, `pop()`, `peek()`, `isEmpty()`

  (Q) How to implement the queue operations: `enqueue()`, `dequeue()`?

  (Step 1) Think properties of stack and queue

  (Step 2) Find a relation between the properties
    - Conceptually, LIFO (reversion) + LIFO (reversion) = FIFO (same order)

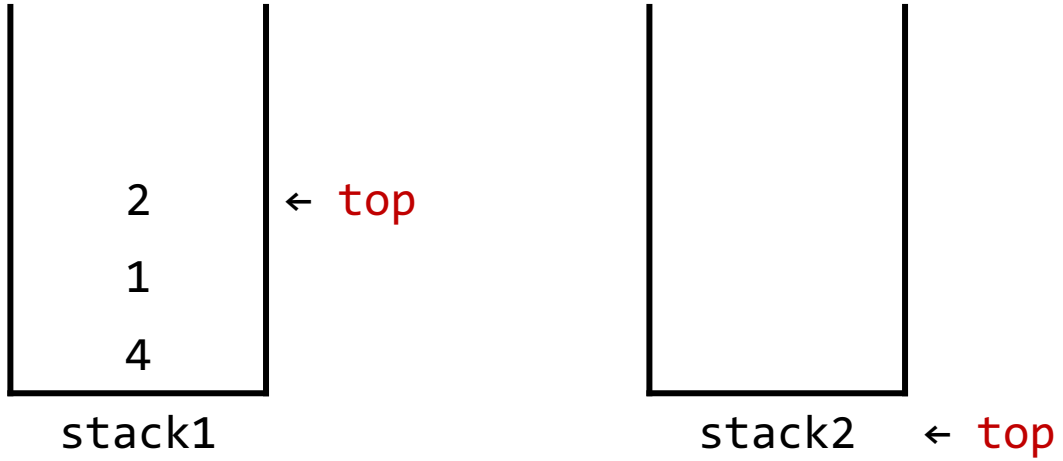


Queues - Problem Solving Practice

- Problem: **Queue Implementation Using Two Stacks**
  - You have two stacks: `stack1`, `stack2`
  - You can use only stack operations: `push()`, `pop()`, `peek()`, `isEmpty()`

  (Q) How to implement the queue operations: `enqueue()`, `dequeue()`?
  - `enqueue(x)` - Simply put an item `x` into `stack1`

```
enqueue(4)
enqueue(1)
enqueue(2)
```

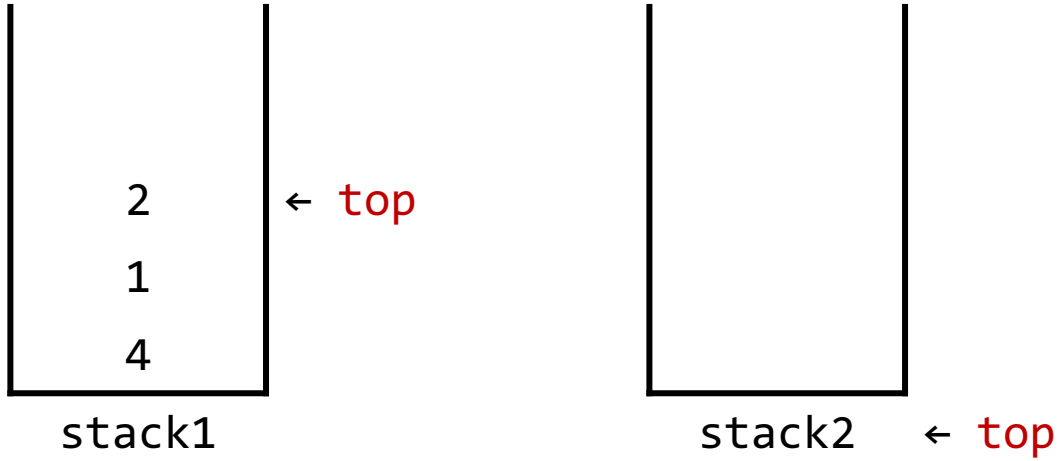


Queues - Problem Solving Practice

- Problem: **Queue Implementation Using Two Stacks**
  - You have two stacks: `stack1`, `stack2`
  - You can use only stack operations: `push()`, `pop()`, `peek()`, `isEmpty()`

  (Q) How to implement the queue operations: `enqueue()`, `dequeue()`?
  - `enqueue(x)` - Simply put an item `x` into `stack1`
  - `dequeue()` - (a) Move all items from `stack1` to `stack2`, then (b) `pop(stack2)`

```
enqueue(4)
enqueue(1)
enqueue(2)
dequeue() ...
```

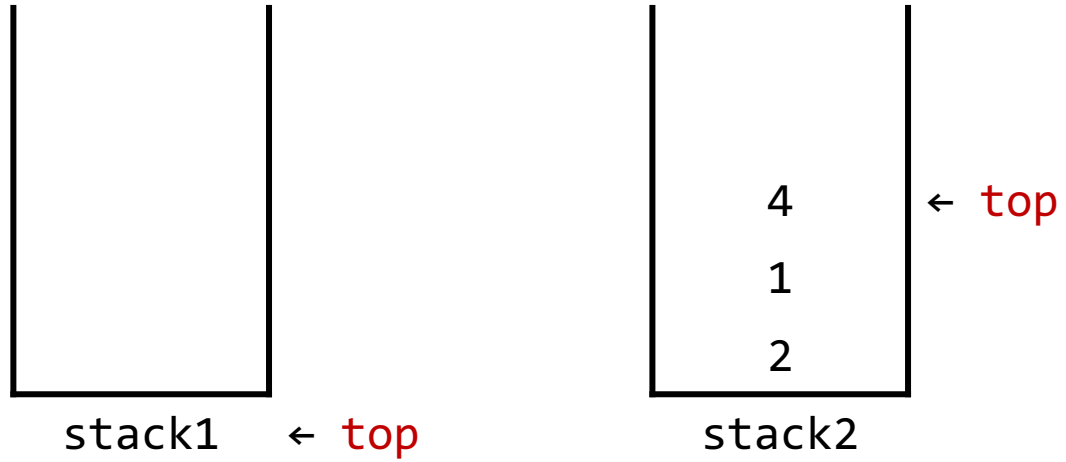


Queues - Problem Solving Practice

- Problem: **Queue Implementation Using Two Stacks**
  - You have two stacks: `stack1`, `stack2`
  - You can use only stack operations: `push()`, `pop()`, `peek()`, `isEmpty()`

  (Q) How to implement the queue operations: `enqueue()`, `dequeue()`?
  - `enqueue(x)` - Simply put an item `x` into `stack1`
  - `dequeue()` - (a) Move all items from `stack1` to `stack2`, then (b) `pop(stack2)`

```
enqueue(4)
enqueue(1)
enqueue(2)
dequeue() ...
```

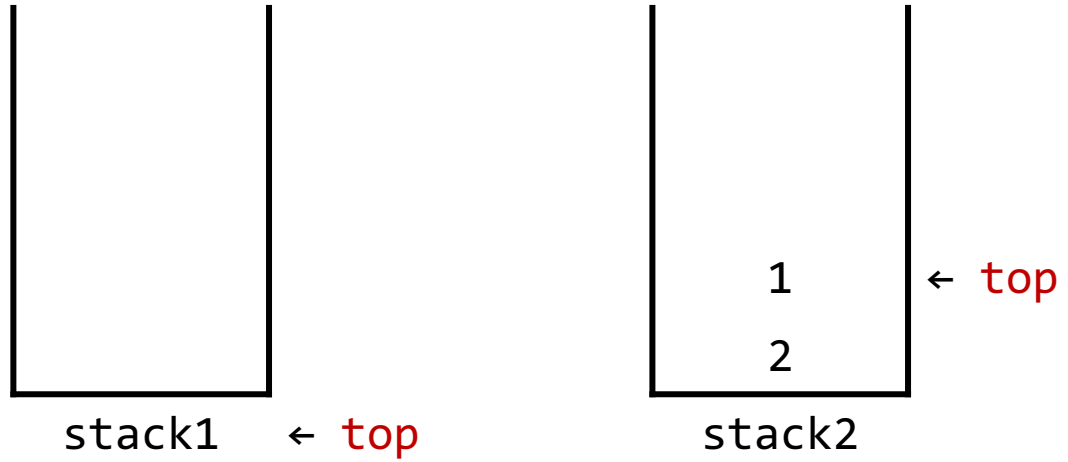


Queues - Problem Solving Practice

- Problem: **Queue Implementation Using Two Stacks**
  - You have two stacks: `stack1`, `stack2`
  - You can use only stack operations: `push()`, `pop()`, `peek()`, `isEmpty()`

  (Q) How to implement the queue operations: `enqueue()`, `dequeue()`?
  - `enqueue(x)` - Simply put an item `x` into `stack1`
  - `dequeue()` - (a) Move all items from `stack1` to `stack2`, then (b) `pop(stack2)`

```
enqueue(4)
enqueue(1)
enqueue(2)
dequeue() → 4
```

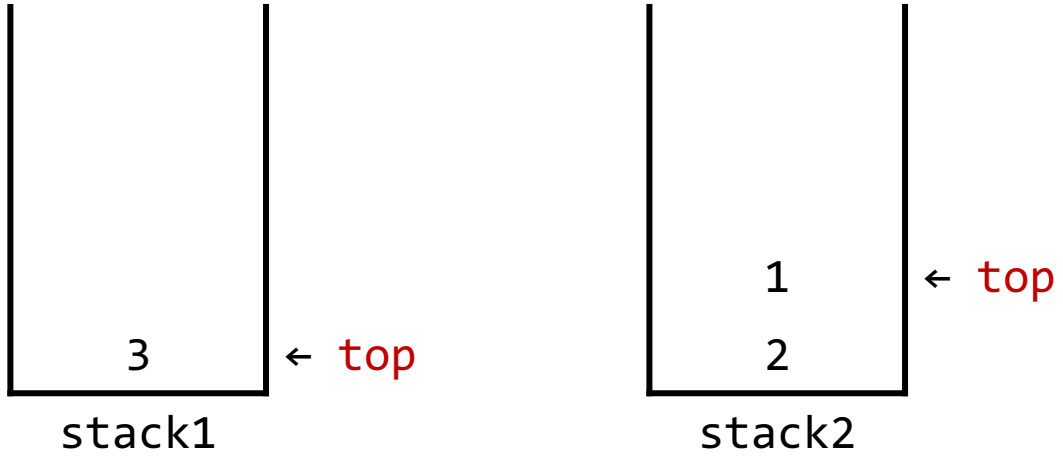


Queues - Problem Solving Practice

- Problem: **Queue Implementation Using Two Stacks**
  - You have two stacks: `stack1`, `stack2`
  - You can use only stack operations: `push()`, `pop()`, `peek()`, `isEmpty()`

  (Q) How to implement the queue operations: `enqueue()`, `dequeue()`?
  - `enqueue(x)` - Simply put an item `x` into `stack1`
  - `dequeue()` - (a) Move all items from `stack1` to `stack2`, then (b) `pop(stack2)`

```
enqueue(4)
enqueue(1)
enqueue(2)
dequeue() → 4
enqueue(3)
```

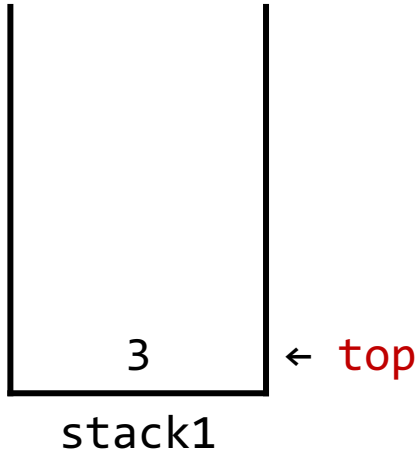


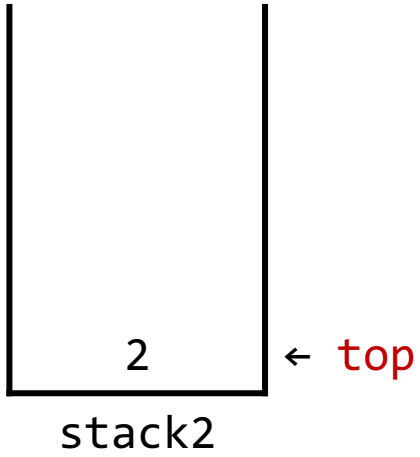
Queues - Problem Solving Practice

- Problem: **Queue Implementation Using Two Stacks**
  - You have two stacks: `stack1`, `stack2`
  - You can use only stack operations: `push()`, `pop()`, `peek()`, `isEmpty()`

  (Q) How to implement the queue operations: `enqueue()`, `dequeue()`?
  - `enqueue(x)` - Simply put an item `x` into `stack1`
  - `dequeue()` - (a) Move all items from `stack1` to `stack2`, then (b) `pop(stack2)`
    - Execute (a) only if `stack2` is empty

```
enqueue(4)
enqueue(1)
enqueue(2)
dequeue() → 4
enqueue(3)
dequeue() → 1
```



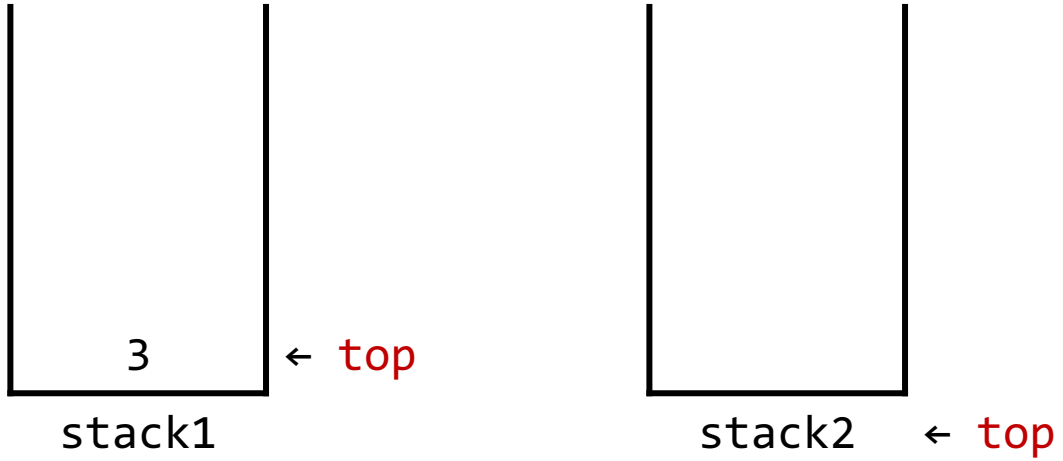


Queues - Problem Solving Practice

- Problem: **Queue Implementation Using Two Stacks**
  - You have two stacks: `stack1`, `stack2`
  - You can use only stack operations: `push()`, `pop()`, `peek()`, `isEmpty()`

  (Q) How to implement the queue operations: `enqueue()`, `dequeue()`?
  - `enqueue(x)` - Simply put an item `x` into `stack1`
  - `dequeue()` - (a) Move all items from `stack1` to `stack2`, then (b) `pop(stack2)`
    - Execute (a) only if `stack2` is empty

```
enqueue(4)
enqueue(1)
enqueue(2)
dequeue() → 4
enqueue(3)
dequeue() → 1
dequeue() → 2
```

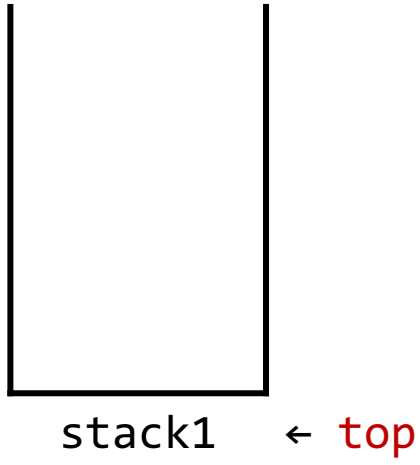


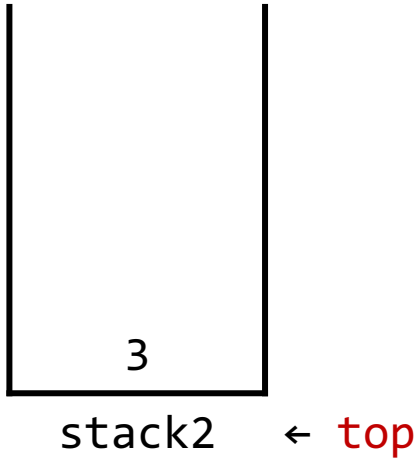
Queues - Problem Solving Practice

- Problem: **Queue Implementation Using Two Stacks**
  - You have two stacks: `stack1`, `stack2`
  - You can use only stack operations: `push()`, `pop()`, `peek()`, `isEmpty()`

  (Q) How to implement the queue operations: `enqueue()`, `dequeue()`?
  - `enqueue(x)` - Simply put an item `x` into `stack1`
  - `dequeue()` - (a) Move all items from `stack1` to `stack2`, then (b) `pop(stack2)`
    - Execute (a) only if `stack2` is empty

```
enqueue(4)
enqueue(1)
enqueue(2)
dequeue() → 4
enqueue(3)
dequeue() → 1
dequeue() → 2
dequeue() ...
```



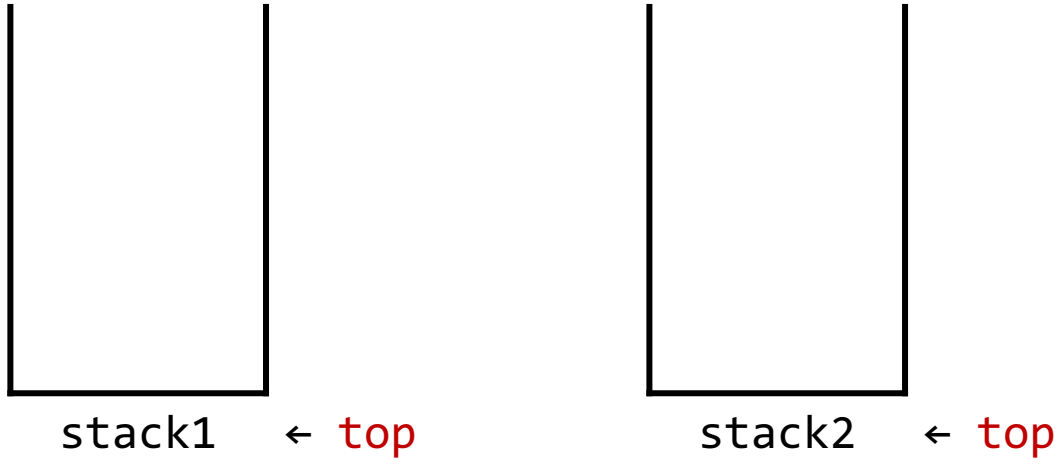


Queues - Problem Solving Practice

- Problem: **Queue Implementation Using Two Stacks**
  - You have two stacks: `stack1`, `stack2`
  - You can use only stack operations: `push()`, `pop()`, `peek()`, `isEmpty()`

  (Q) How to implement the queue operations: `enqueue()`, `dequeue()`?
  - `enqueue(x)` - Simply put an item `x` into `stack1`
  - `dequeue()` - (a) Move all items from `stack1` to `stack2`, then (b) `pop(stack2)`
    - Execute (a) only if `stack2` is empty

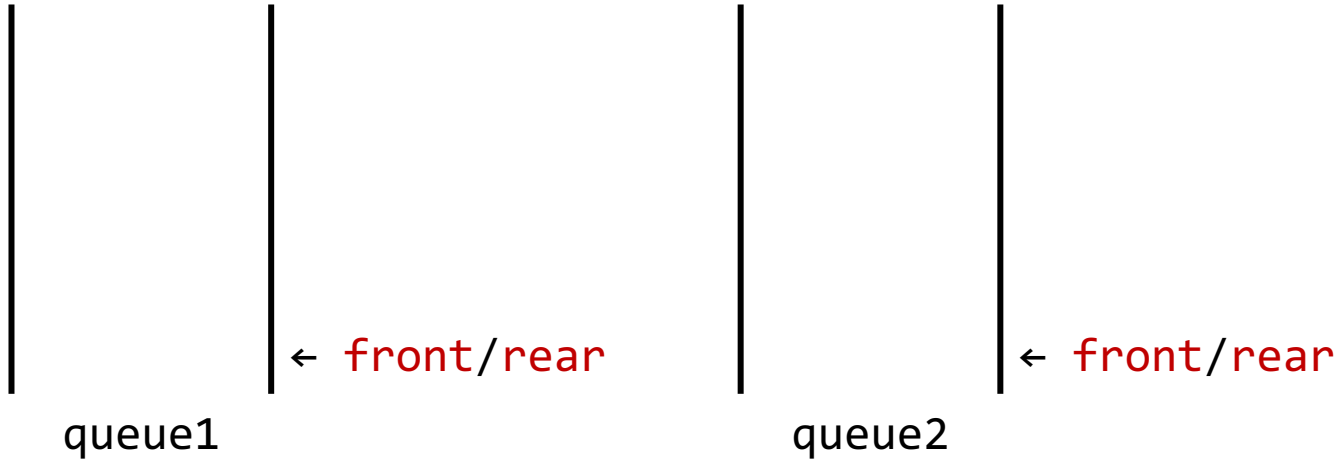
```
enqueue(4)
enqueue(1)
enqueue(2)
dequeue() → 4
enqueue(3)
dequeue() → 1
dequeue() → 2
dequeue() → 3
```



Queues - Problem Solving Practice

- Problem: **Stack Implementation Using Two Queues**
  - You have two stacks: `queue1`, `queue2`
  - You can use only queue operations: `enqueue()`, `dequeue()`, `peek()`, `isEmpty()`

(Q) How to implement the stack operations: `push()`, `pop()`?

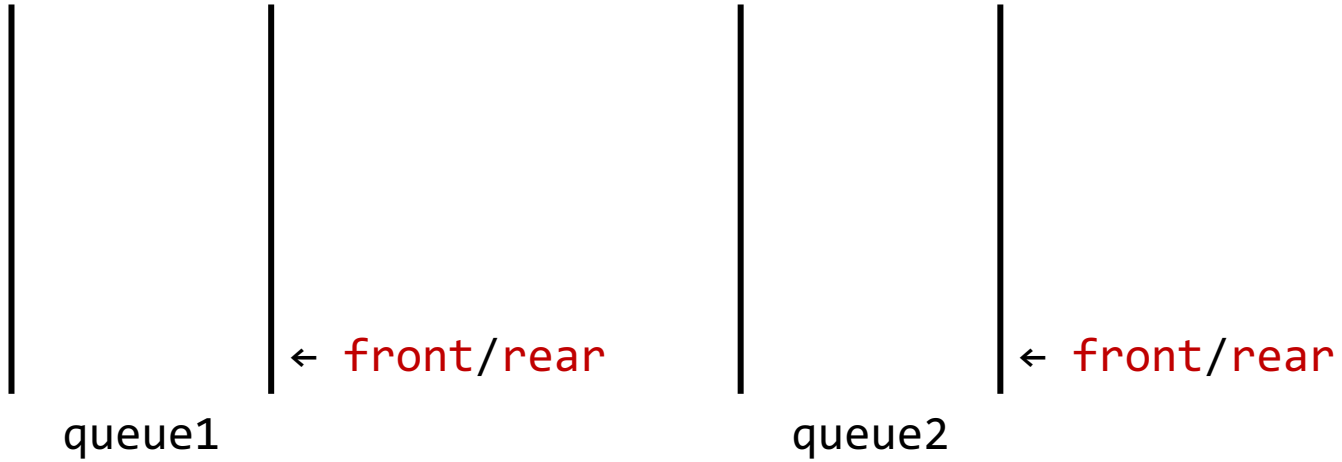


Queues - Problem Solving Practice

- Problem: **Stack Implementation Using Two Queues**
  - You have two stacks: `queue1`, `queue2`
  - You can use only queue operations: `enqueue()`, `dequeue()`, `peek()`, `isEmpty()`

(Q) How to implement the stack operations: `push()`, `pop()`?

(A) To implement stack's `pop()`, you need to put the last item at the front

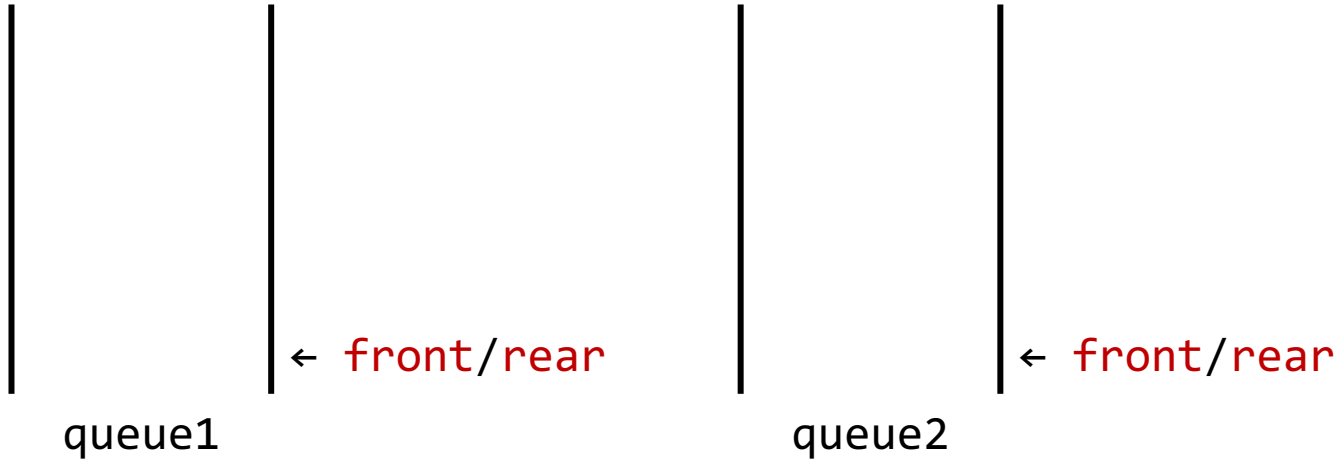


Queues - Problem Solving Practice

- Problem: **Stack Implementation Using Two Queues**
  - You have two stacks: `queue1`, `queue2`
  - You can use only queue operations: `enqueue()`, `dequeue()`, `peek()`, `isEmpty()`

  (Q) How to implement the stack operations: `push()`, `pop()`?

  (A) To implement stack's `pop()`, you need to put the last item at the front
  - `push(x)` - Temporarily move `queue1`'s items using `queue2` then put `x` at the front
  - `pop()` - Remove the front element in queue

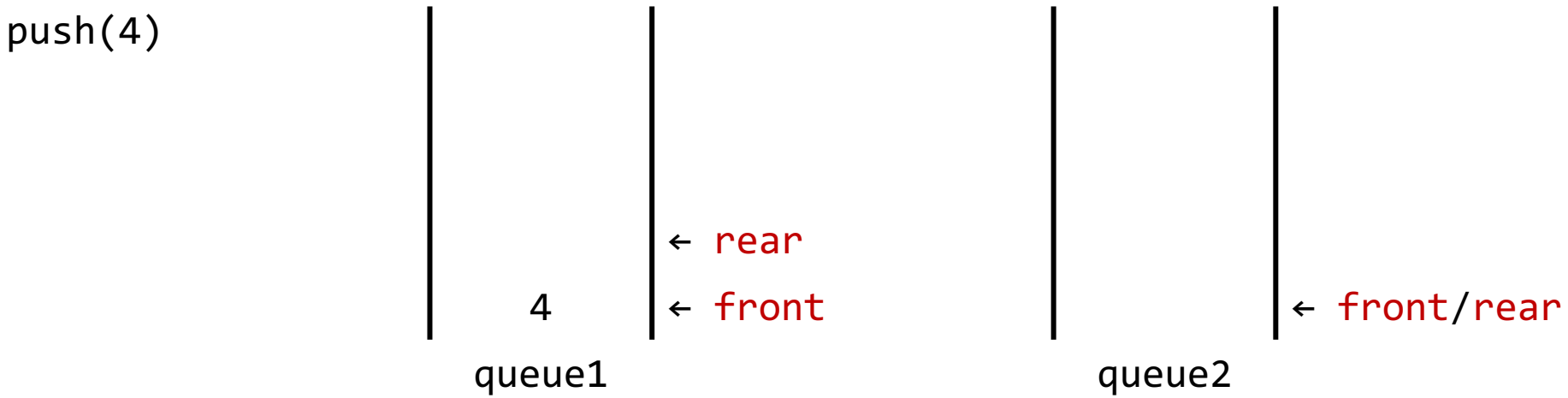


Queues - Problem Solving Practice

- Problem: **Stack Implementation Using Two Queues**
  - You have two stacks: `queue1`, `queue2`
  - You can use only queue operations: `enqueue()`, `dequeue()`, `peek()`, `isEmpty()`

  (Q) How to implement the stack operations: `push()`, `pop()`?

  (A) To implement stack's `pop()`, you need to put the last item at the front
  - `push(x)` - Temporarily move `queue1`'s items using `queue2` then put `x` at the front
  - `pop()` - Remove the front element in queue

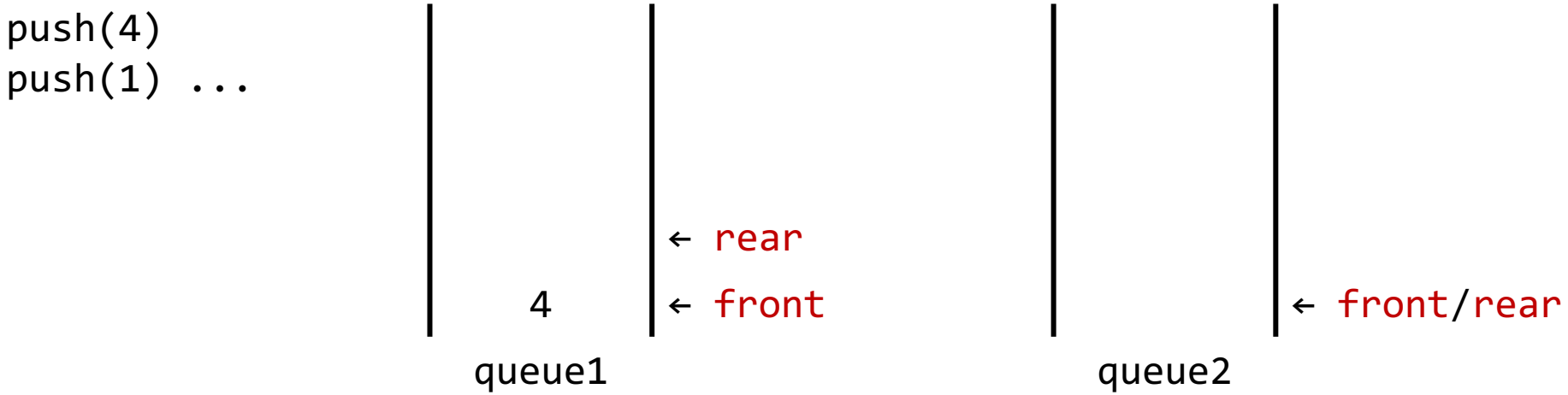


Queues - Problem Solving Practice

- Problem: **Stack Implementation Using Two Queues**
  - You have two stacks: `queue1`, `queue2`
  - You can use only queue operations: `enqueue()`, `dequeue()`, `peek()`, `isEmpty()`

  (Q) How to implement the stack operations: `push()`, `pop()`?

  (A) To implement stack's `pop()`, you need to put the last item at the front
  - `push(x)` - Temporarily move `queue1`'s items using `queue2` then put `x` at the front
  - `pop()` - Remove the front element in queue

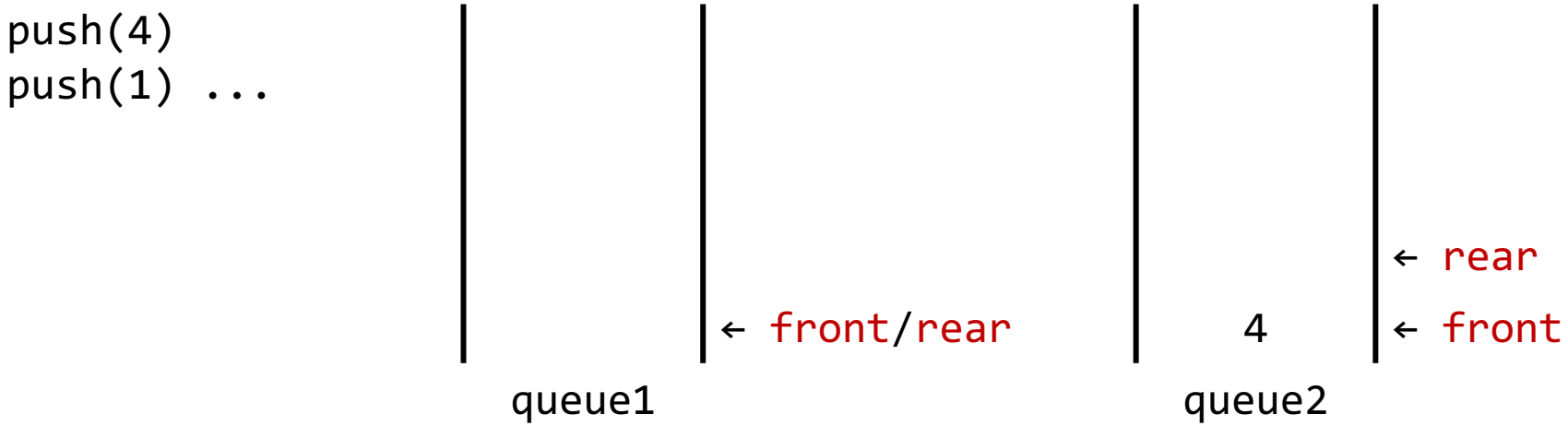


Queues - Problem Solving Practice

- Problem: **Stack Implementation Using Two Queues**
  - You have two stacks: `queue1`, `queue2`
  - You can use only queue operations: `enqueue()`, `dequeue()`, `peek()`, `isEmpty()`

  (Q) How to implement the stack operations: `push()`, `pop()`?

  (A) To implement stack's `pop()`, you need to put the last item at the front
  - `push(x)` - Temporarily move `queue1`'s items using `queue2` then put `x` at the front
  - `pop()` - Remove the front element in queue

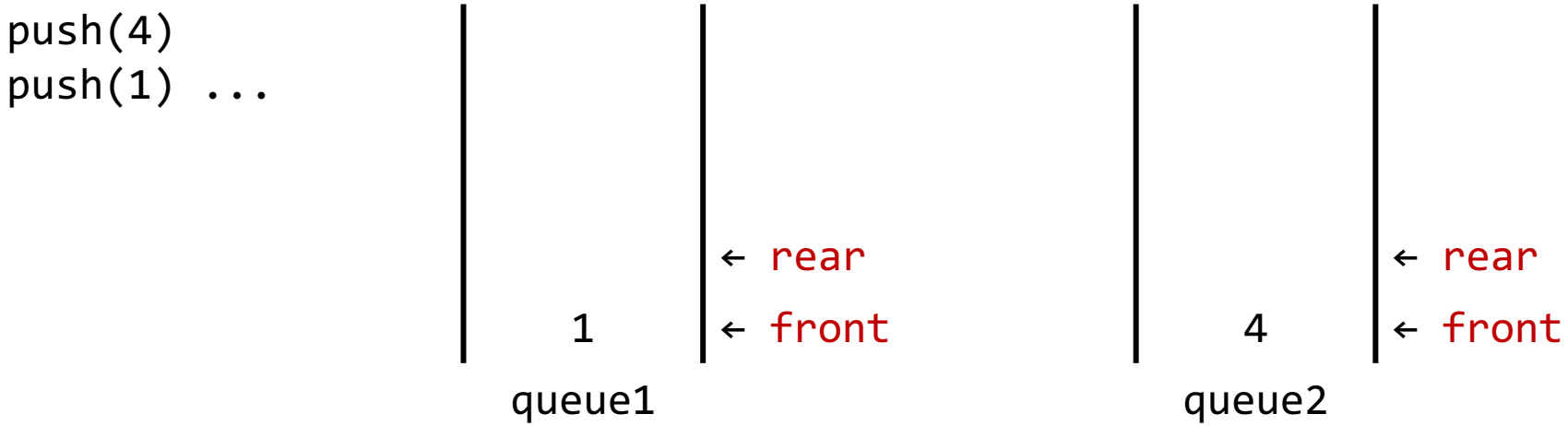


Queues - Problem Solving Practice

- Problem: **Stack Implementation Using Two Queues**
  - You have two stacks: `queue1`, `queue2`
  - You can use only queue operations: `enqueue()`, `dequeue()`, `peek()`, `isEmpty()`

  (Q) How to implement the stack operations: `push()`, `pop()`?

  (A) To implement stack's `pop()`, you need to put the last item at the front
  - `push(x)` - Temporarily move `queue1`'s items using `queue2` then put `x` at the front
  - `pop()` - Remove the front element in queue

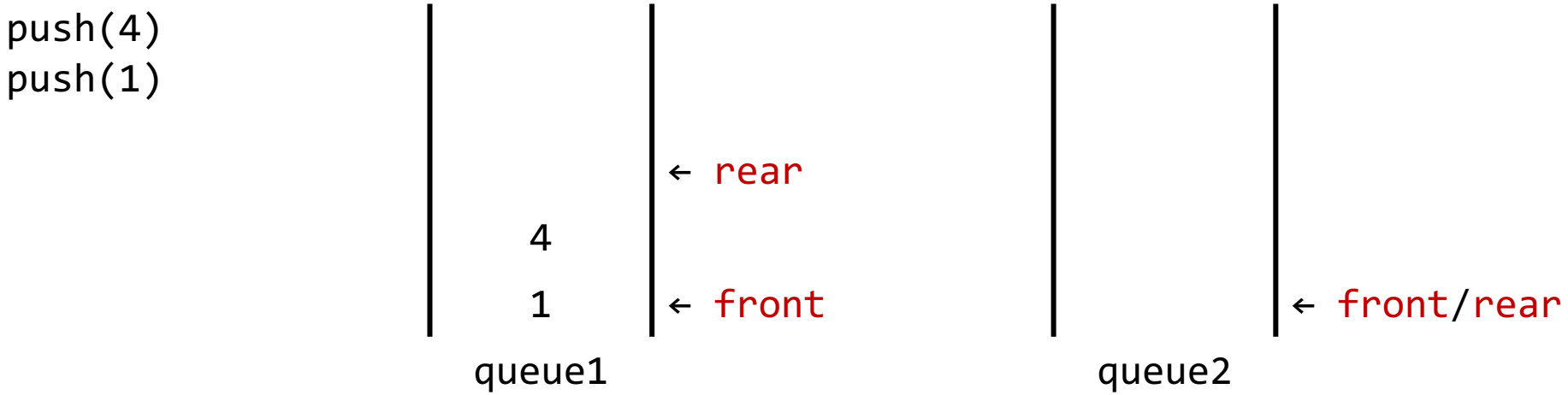


Queues - Problem Solving Practice

- Problem: **Stack Implementation Using Two Queues**
  - You have two stacks: `queue1`, `queue2`
  - You can use only queue operations: `enqueue()`, `dequeue()`, `peek()`, `isEmpty()`

  (Q) How to implement the stack operations: `push()`, `pop()`?

  (A) To implement stack's `pop()`, you need to put the last item at the front
  - `push(x)` - Temporarily move `queue1`'s items using `queue2` then put `x` at the front
  - `pop()` - Remove the front element in queue

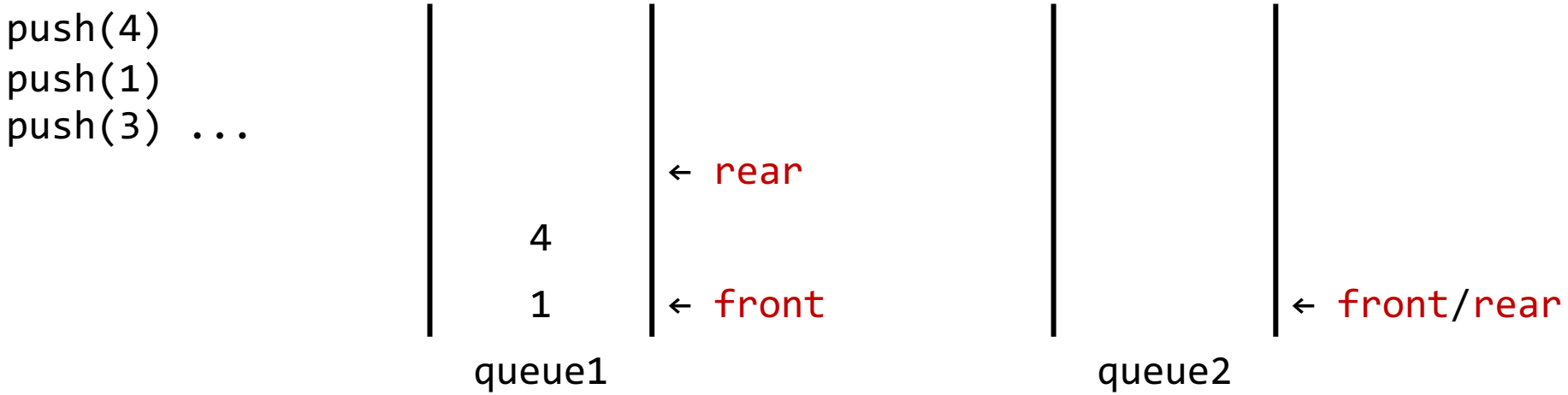


Queues - Problem Solving Practice

- Problem: **Stack Implementation Using Two Queues**
  - You have two stacks: `queue1`, `queue2`
  - You can use only queue operations: `enqueue()`, `dequeue()`, `peek()`, `isEmpty()`

  (Q) How to implement the stack operations: `push()`, `pop()`?

  (A) To implement stack's `pop()`, you need to put the last item at the front
  - `push(x)` - Temporarily move `queue1`'s items using `queue2` then put `x` at the front
  - `pop()` - Remove the front element in queue

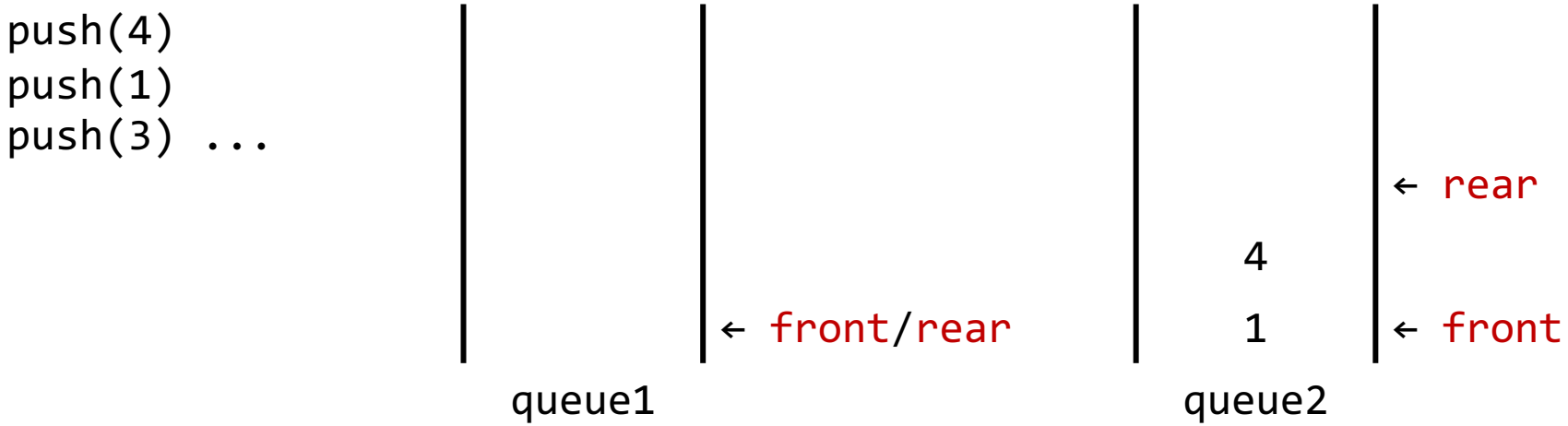


Queues - Problem Solving Practice

- Problem: **Stack Implementation Using Two Queues**
  - You have two stacks: `queue1`, `queue2`
  - You can use only queue operations: `enqueue()`, `dequeue()`, `peek()`, `isEmpty()`

  (Q) How to implement the stack operations: `push()`, `pop()`?

  (A) To implement stack's `pop()`, you need to put the last item at the front
  - `push(x)` - Temporarily move `queue1`'s items using `queue2` then put `x` at the front
  - `pop()` - Remove the front element in queue



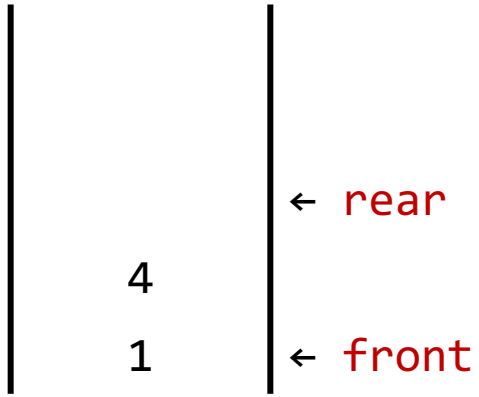
Queues - Problem Solving Practice

- Problem: **Stack Implementation Using Two Queues**
  - You have two stacks: `queue1`, `queue2`
  - You can use only queue operations: `enqueue()`, `dequeue()`, `peek()`, `isEmpty()`

  (Q) How to implement the stack operations: `push()`, `pop()`?

  (A) To implement stack's `pop()`, you need to put the last item at the front
  - `push(x)` - Temporarily move `queue1`'s items using `queue2` then put `x` at the front
  - `pop()` - Remove the front element in queue

```
push(4)
push(1)
push(3) ...
```



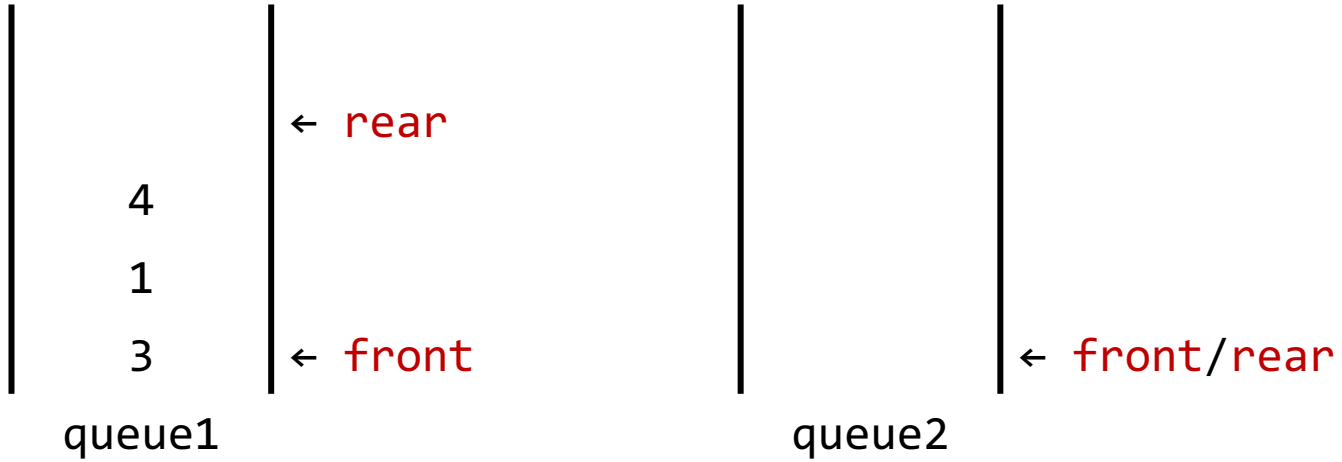
Queues - Problem Solving Practice

- Problem: **Stack Implementation Using Two Queues**
  - You have two stacks: `queue1`, `queue2`
  - You can use only queue operations: `enqueue()`, `dequeue()`, `peek()`, `isEmpty()`

  (Q) How to implement the stack operations: `push()`, `pop()`?

  (A) To implement stack's `pop()`, you need to put the last item at the front
  - `push(x)` - Temporarily move `queue1`'s items using `queue2` then put `x` at the front
  - `pop()` - Remove the front element in queue

```
push(4)
push(1)
push(3)
```



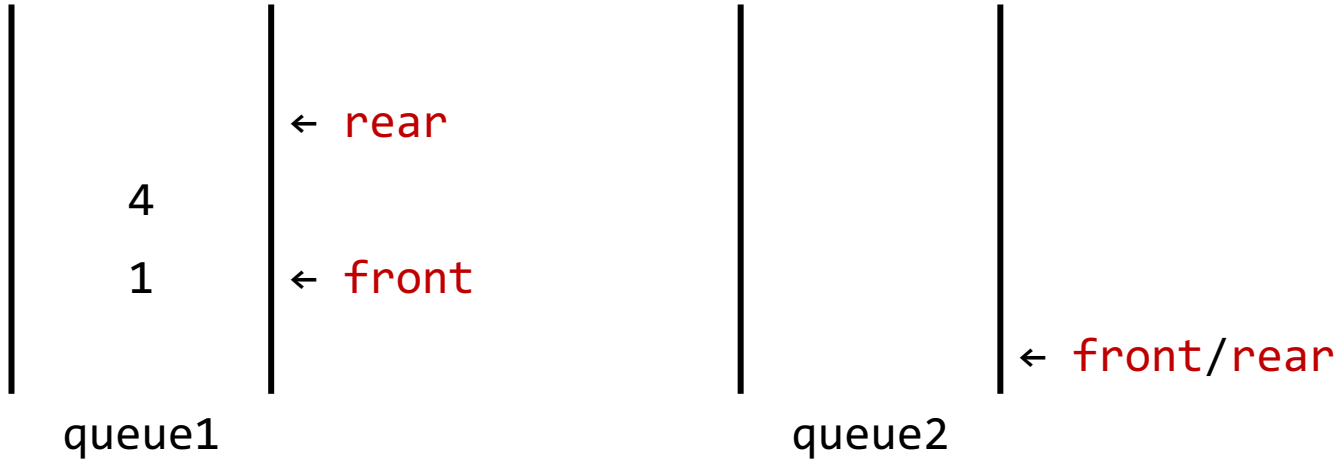
Queues - Problem Solving Practice

- Problem: **Stack Implementation Using Two Queues**
  - You have two stacks: `queue1`, `queue2`
  - You can use only queue operations: `enqueue()`, `dequeue()`, `peek()`, `isEmpty()`

  (Q) How to implement the stack operations: `push()`, `pop()`?

  (A) To implement stack's `pop()`, you need to put the last item at the front
  - `push(x)` - Temporarily move `queue1`'s items using `queue2` then put `x` at the front
  - `pop()` - Remove the front element in queue

```
push(4)
push(1)
push(3)
pop() → 3
```



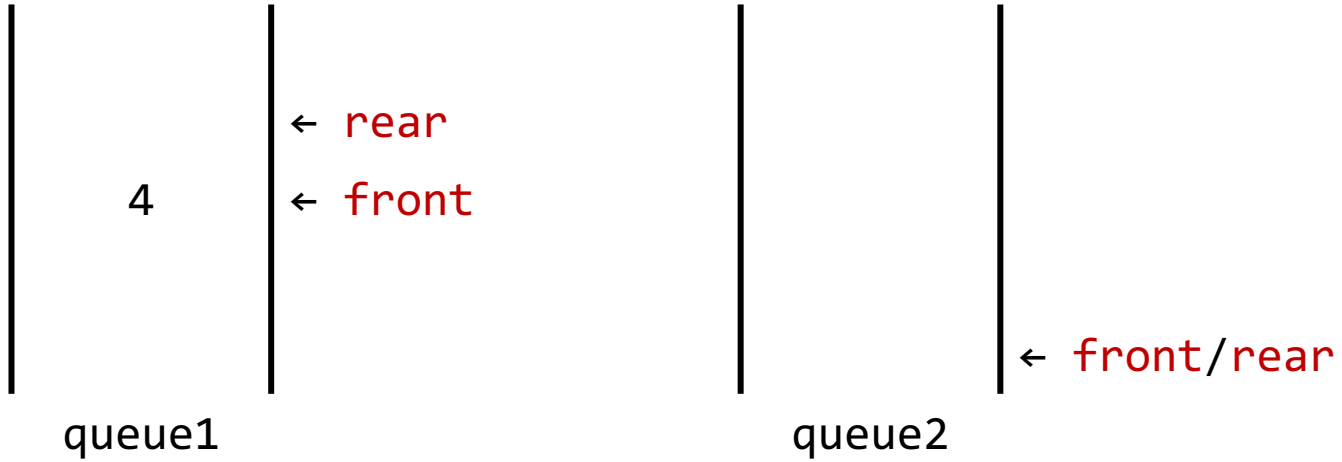
Queues - Problem Solving Practice

- Problem: **Stack Implementation Using Two Queues**
  - You have two stacks: `queue1`, `queue2`
  - You can use only queue operations: `enqueue()`, `dequeue()`, `peek()`, `isEmpty()`

  (Q) How to implement the stack operations: `push()`, `pop()`?

  (A) To implement stack's `pop()`, you need to put the last item at the front
  - `push(x)` - Temporarily move `queue1`'s items using `queue2` then put `x` at the front
  - `pop()` - Remove the front element in queue

```
push(4)
push(1)
push(3)
pop() → 3
pop() → 1
```



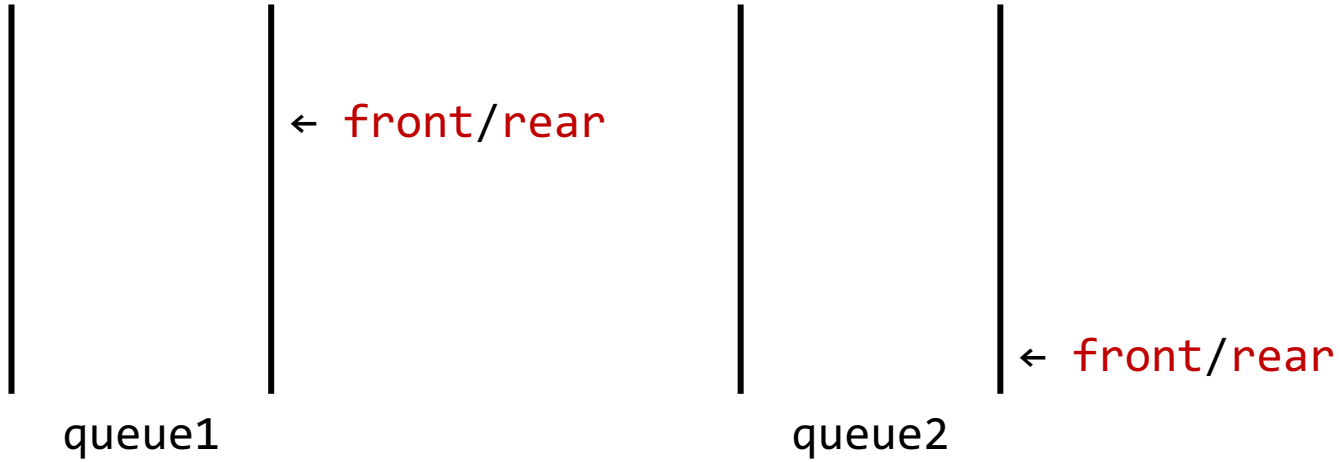
Queues - Problem Solving Practice

- Problem: **Stack Implementation Using Two Queues**
  - You have two stacks: `queue1`, `queue2`
  - You can use only queue operations: `enqueue()`, `dequeue()`, `peek()`, `isEmpty()`

  (Q) How to implement the stack operations: `push()`, `pop()`?

  (A) To implement stack's `pop()`, you need to put the last item at the front
  - `push(x)` - Temporarily move `queue1`'s items using `queue2` then put `x` at the front
  - `pop()` - Remove the front element in queue

```
push(4)
push(1)
push(3)
pop() → 3
pop() → 1
pop() → 4
```



Any Questions?