Concurrent Queue

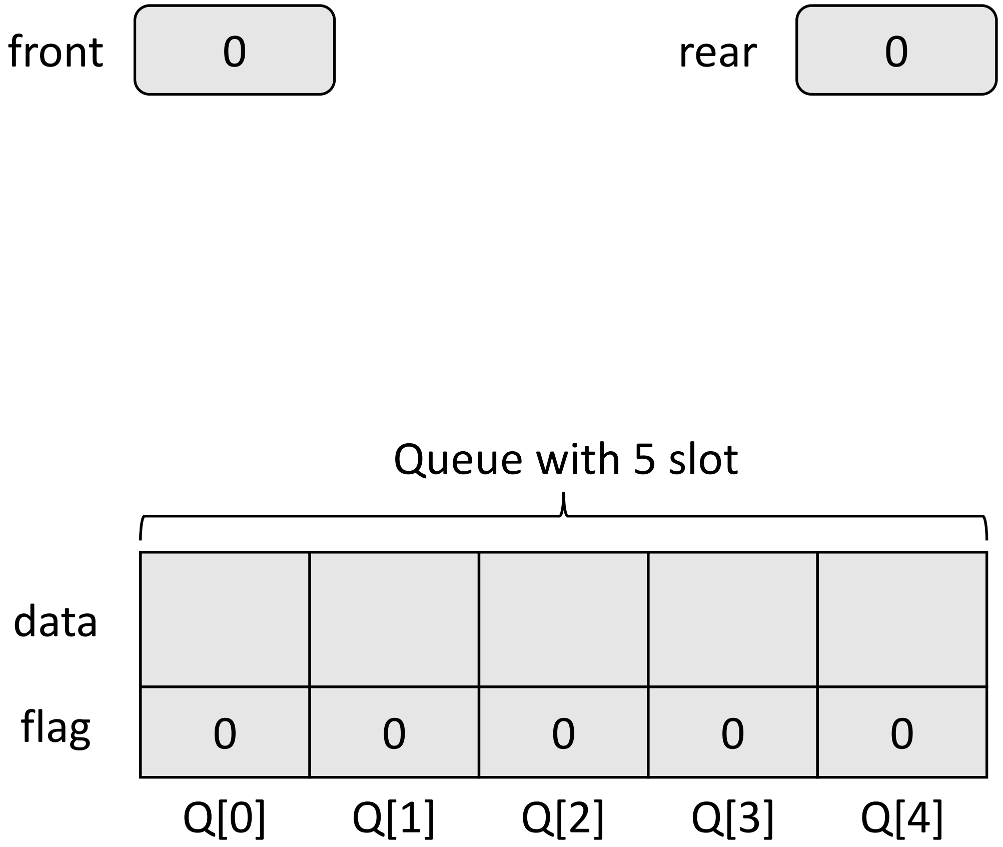
Concurrent Programming

Introduction

- Bounded Lock-free Queue

- Evaluation
  - Queue with coarse-grained locking
  - Unbounded Lock-free Queue (covered in lecture note)
  - Bounded Lock-free Queue

Bounded lock-free queue



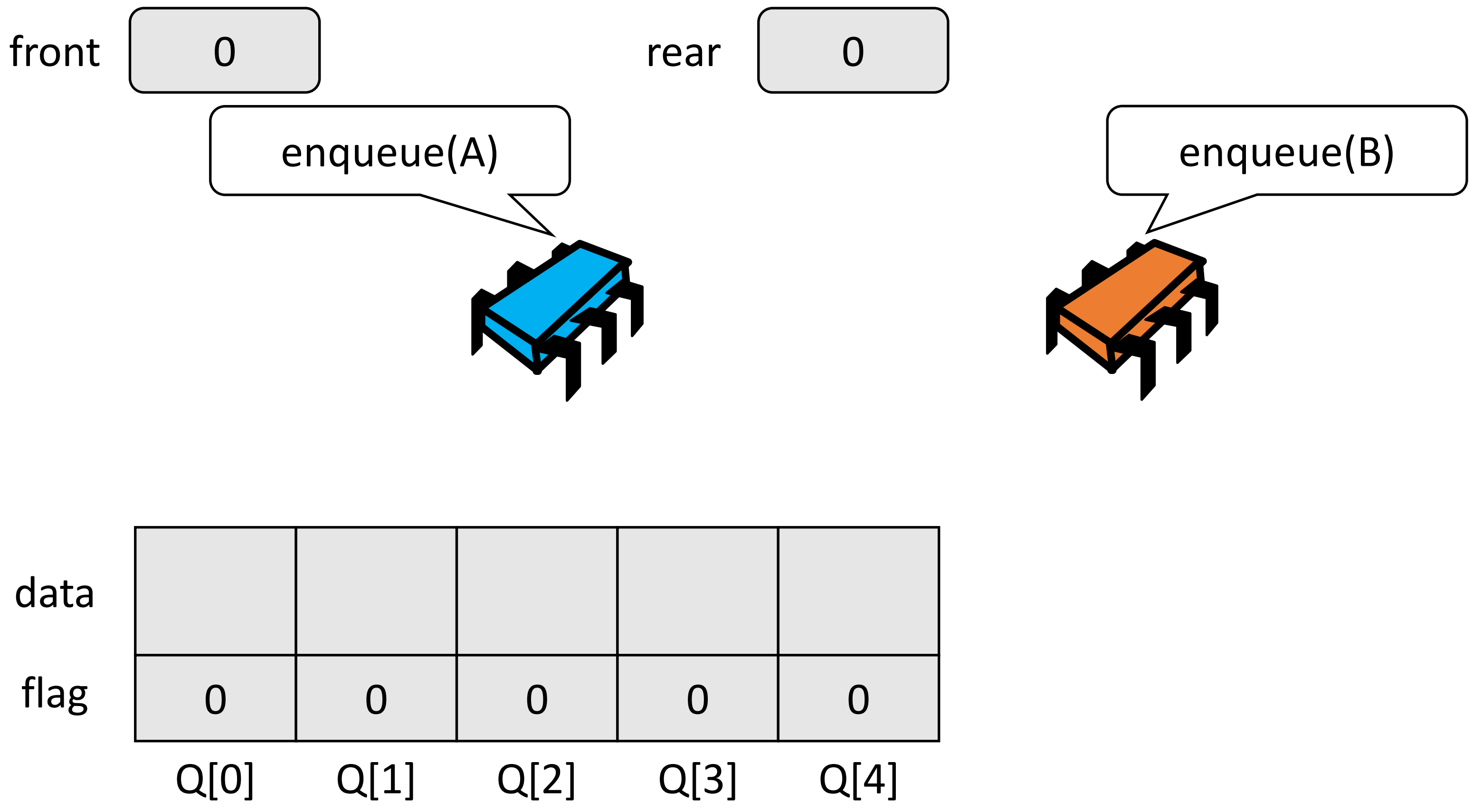
front 0

rear 0

Queue with 5 slot

	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data					
flag	0	0	0	0	0

Enqueue



front 0

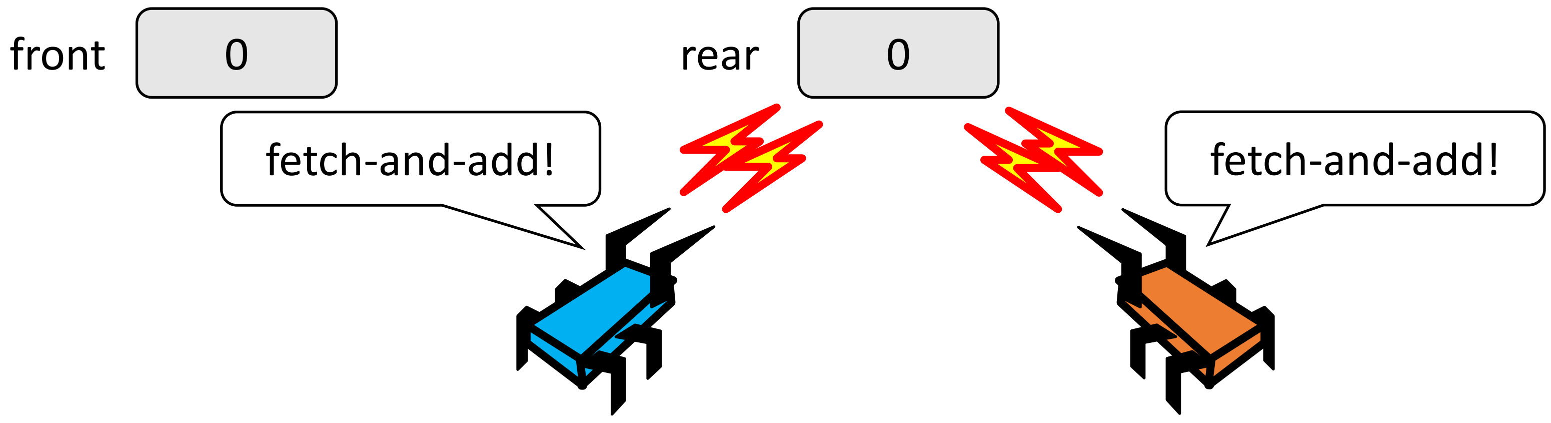
rear 0

enqueue(A)

enqueue(B)

	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data					
flag	0	0	0	0	0

Enqueue



front 0

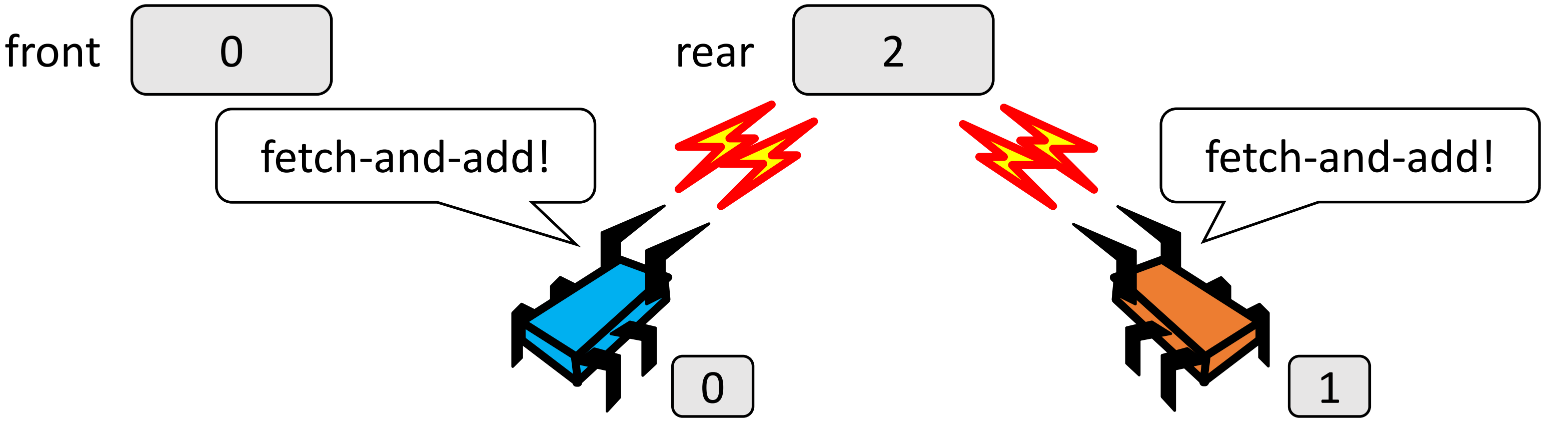
rear 0

fetch-and-add!

fetch-and-add!

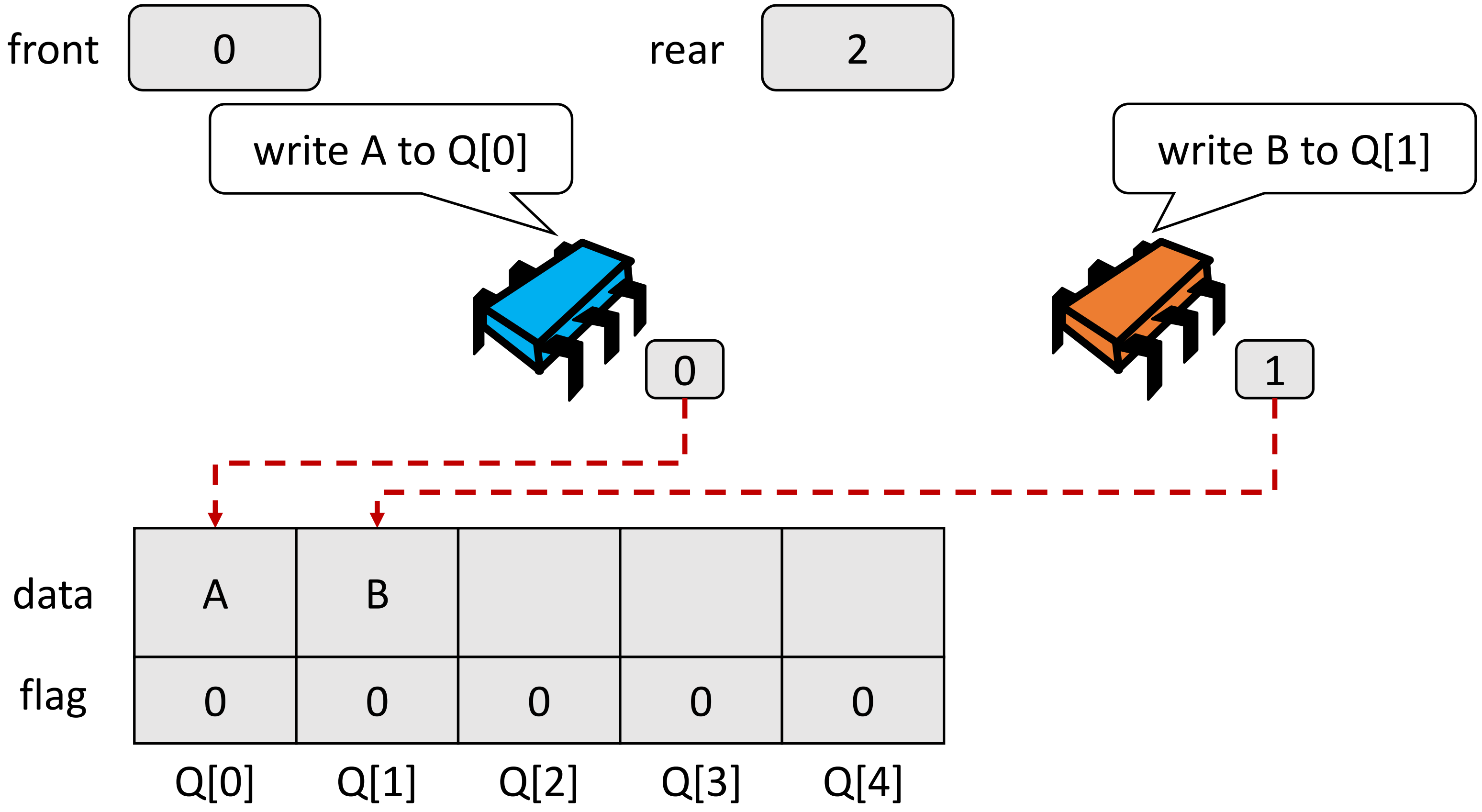
	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data					
flag	0	0	0	0	0

Enqueue



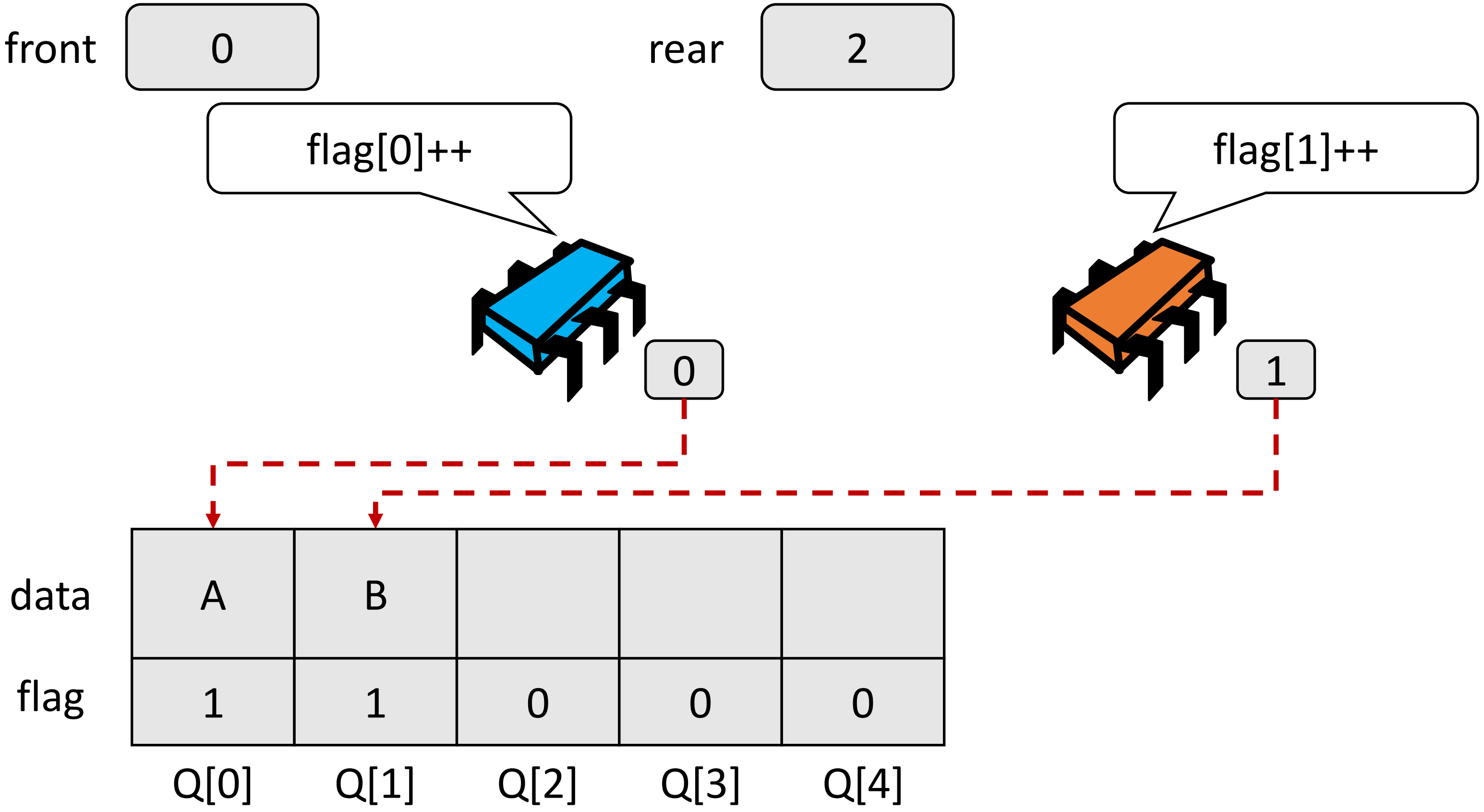
	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data					
flag	0	0	0	0	0

Enqueue



	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B			
flag	0	0	0	0	0

Enqueue



data	A	B			
flag	1	1	0	0	0
	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]

Dequeue

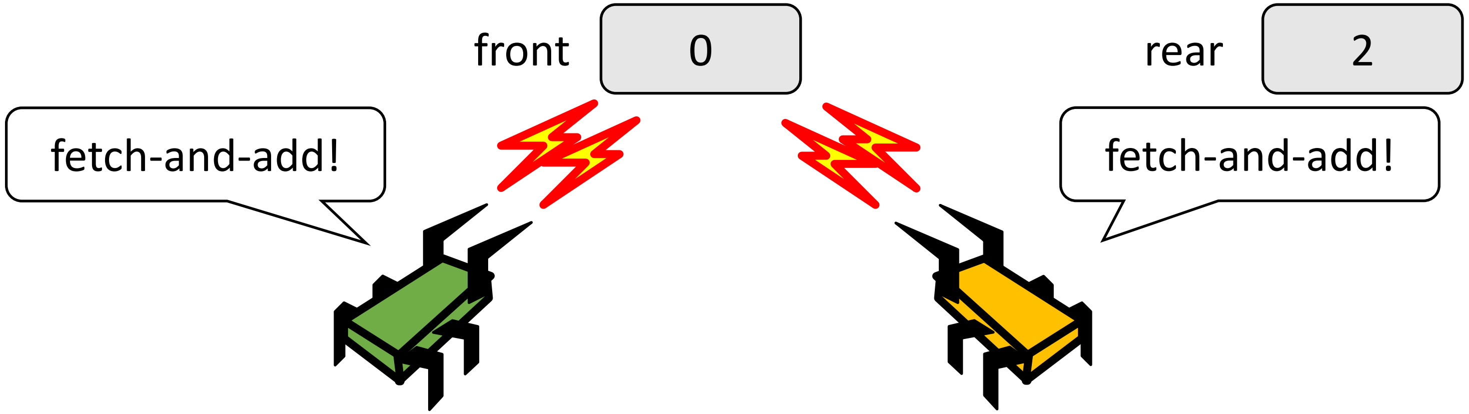
front 0

rear 2

dequeue()
dequeue()

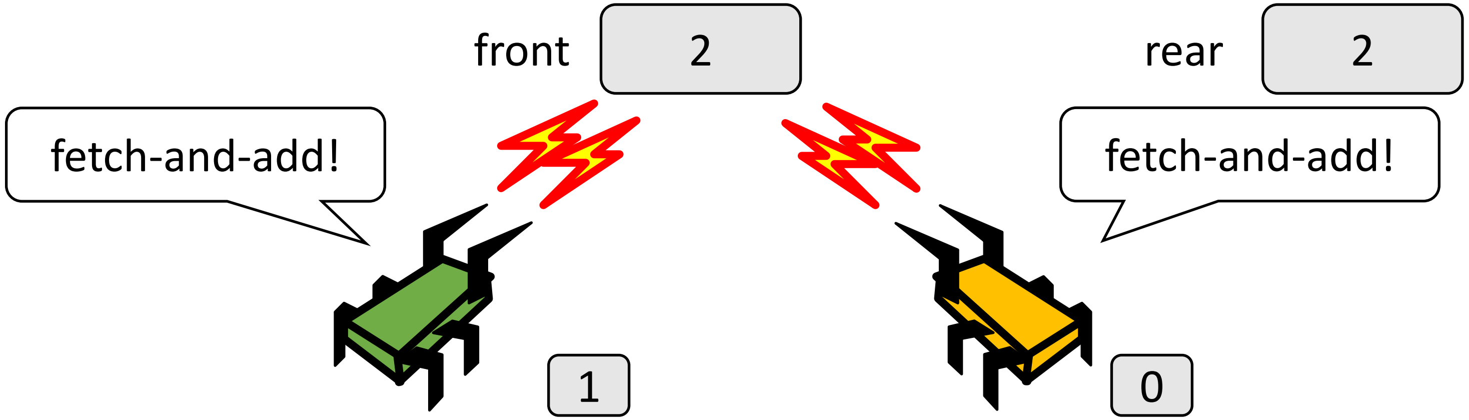
	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B			
flag	1	1	0	0	0

Dequeue



	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B			
flag	1	1	0	0	0

Dequeue



front 2

rear 2

fetch-and-add!

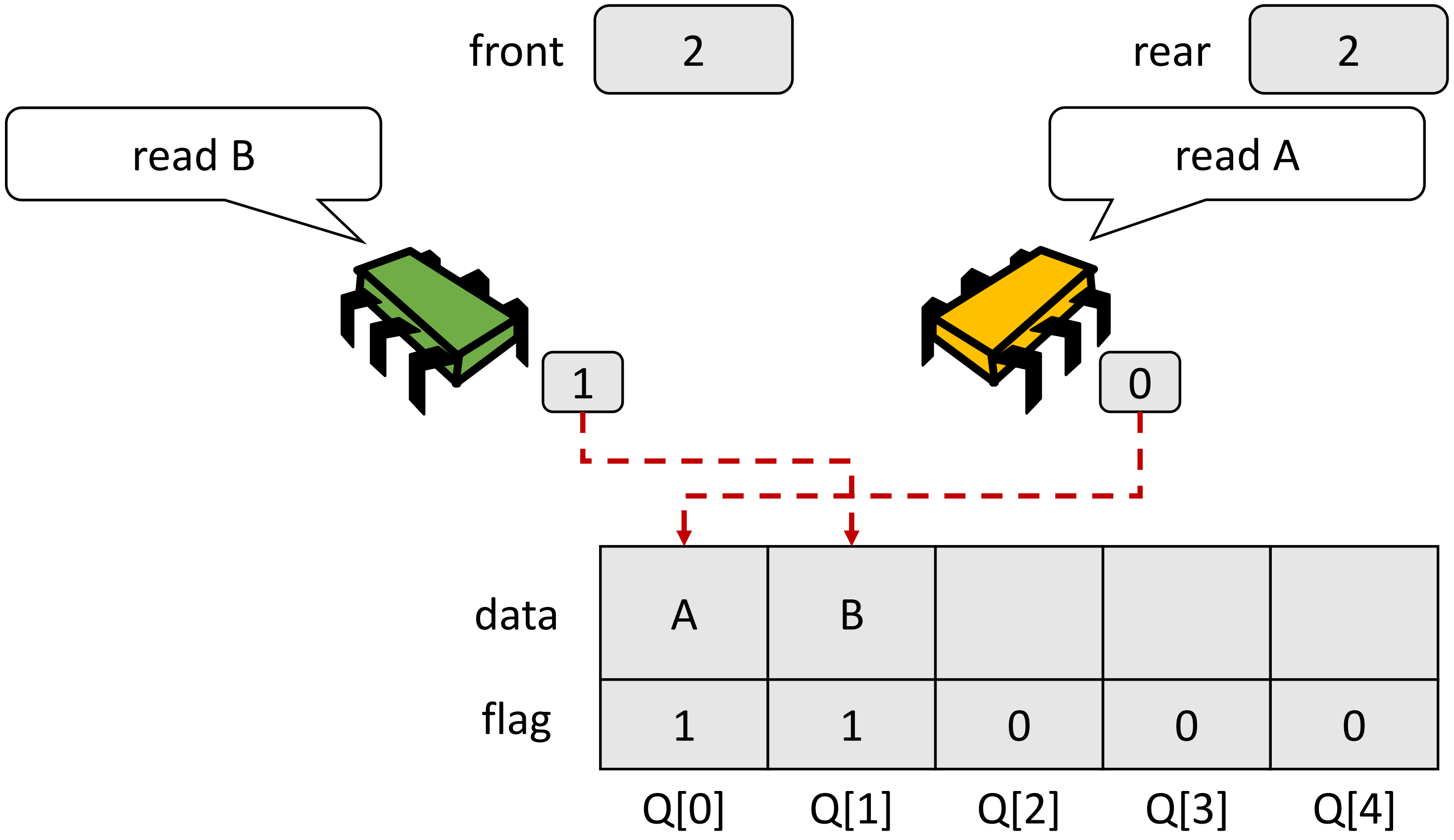
fetch-and-add!

1

0

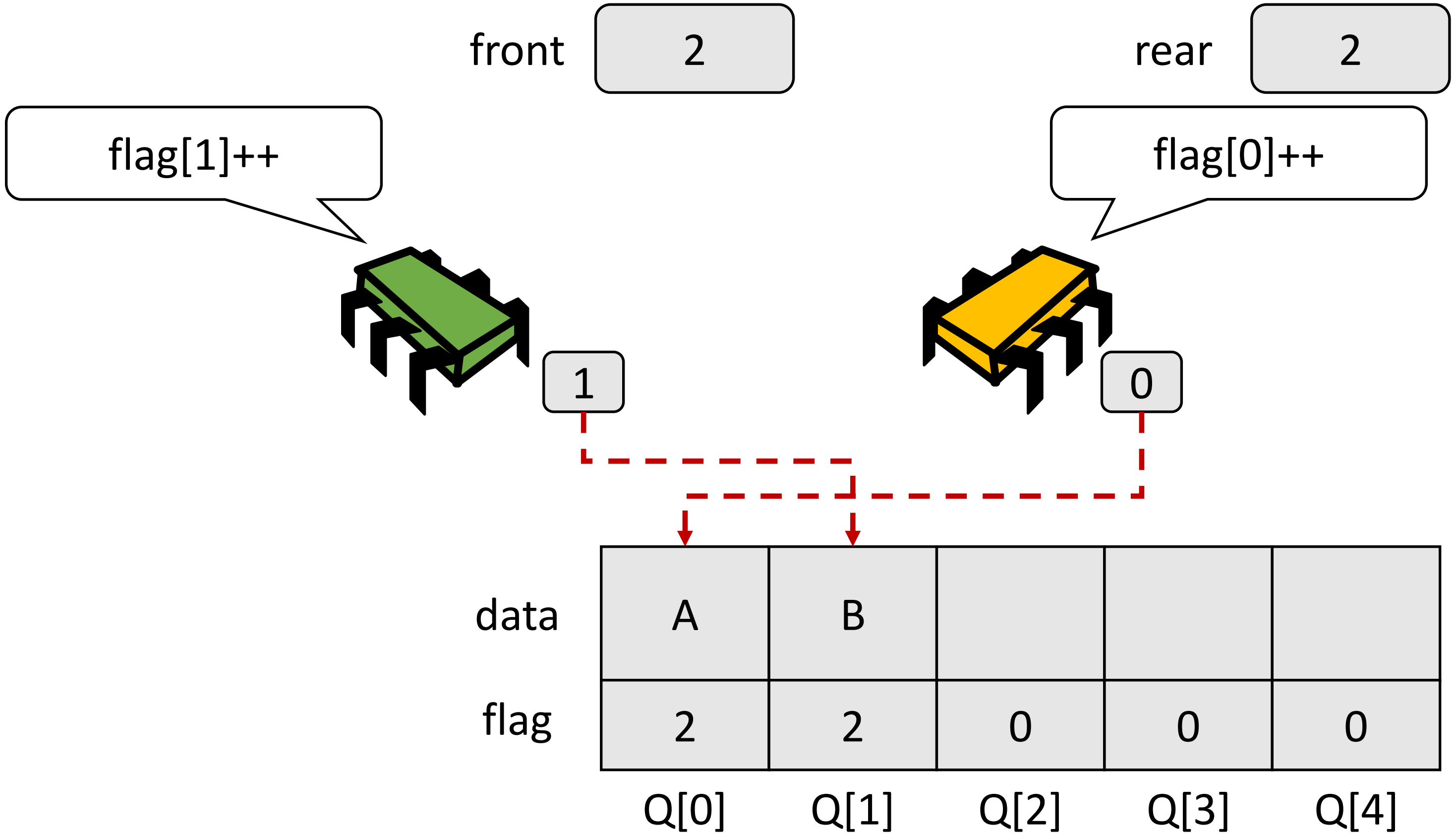
	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B			
flag	1	1	0	0	0

Dequeue



	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B			
flag	1	1	0	0	0

Dequeue

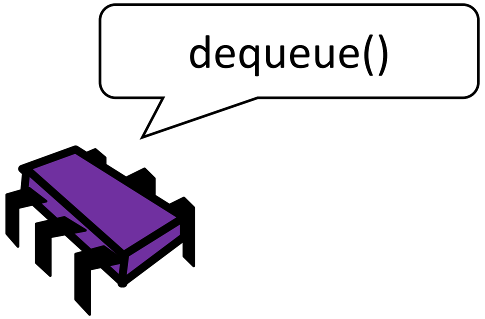


	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B			
flag	2	2	0	0	0

Dequeue

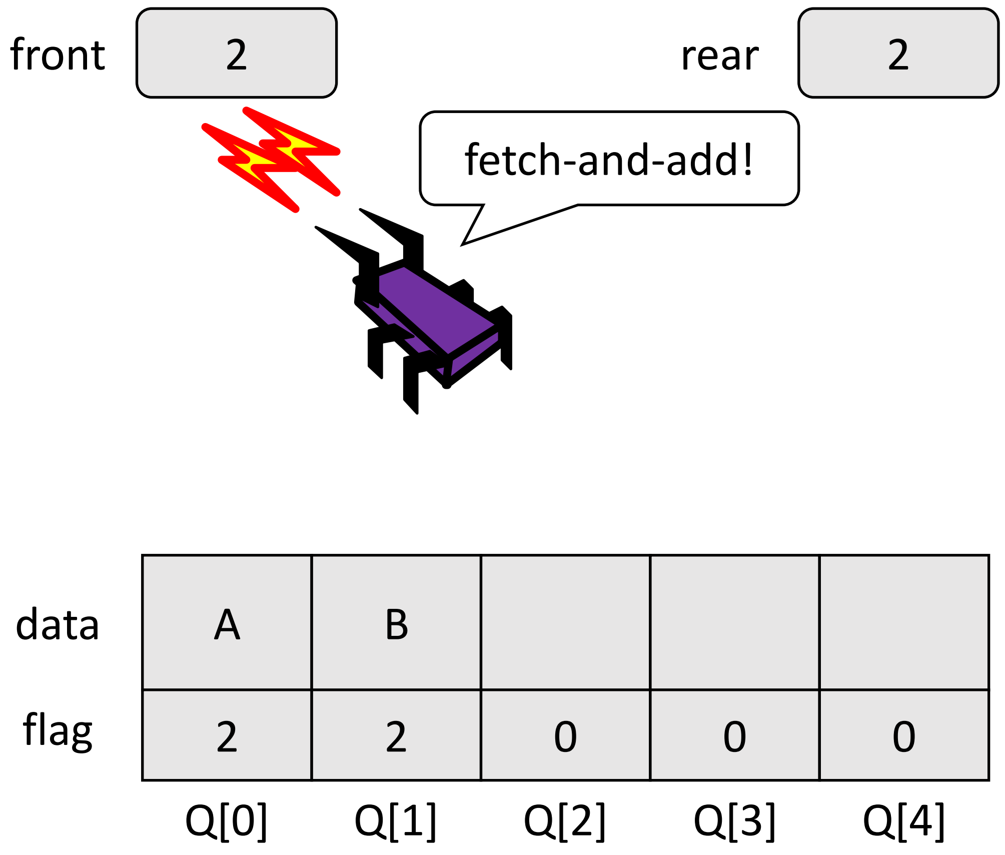
front 2

rear 2



	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B			
flag	2	2	0	0	0

Dequeue



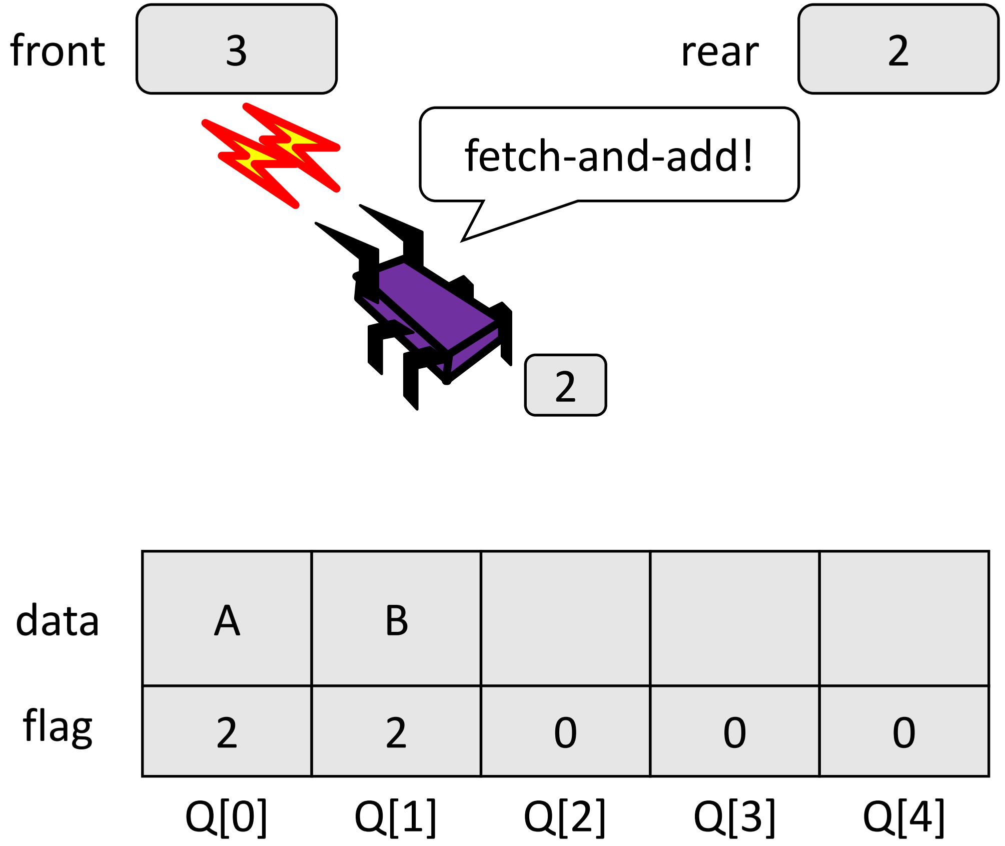
front 2

rear 2

fetch-and-add!

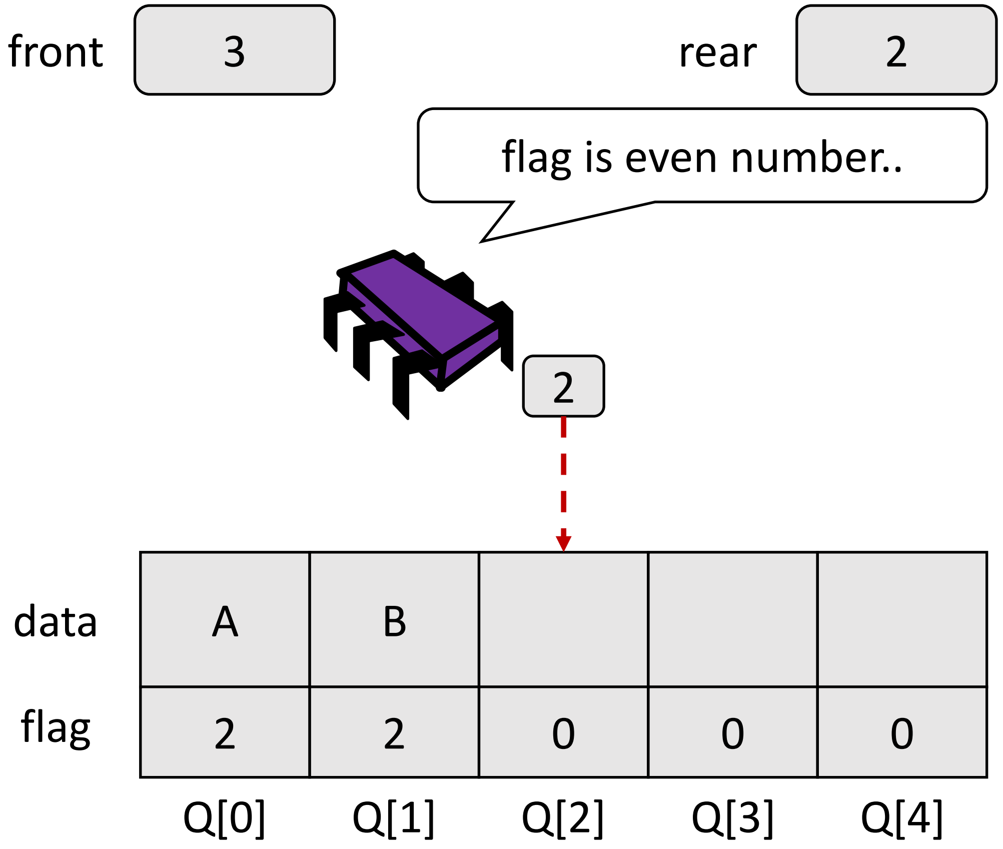
	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B			
flag	2	2	0	0	0

Dequeue



	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B			
flag	2	2	0	0	0

Dequeue



front 3

rear 2

flag is even number..

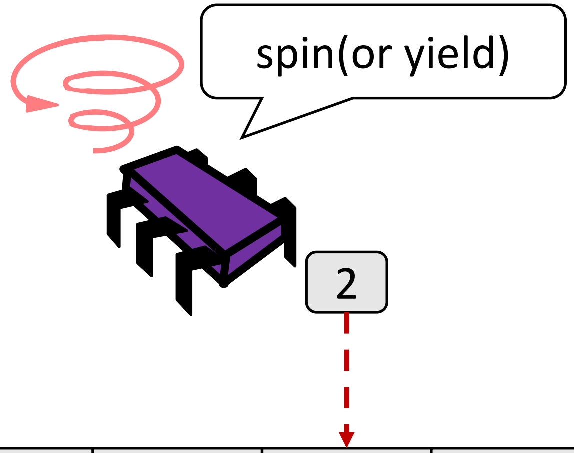
2

	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B			
flag	2	2	0	0	0

Dequeue

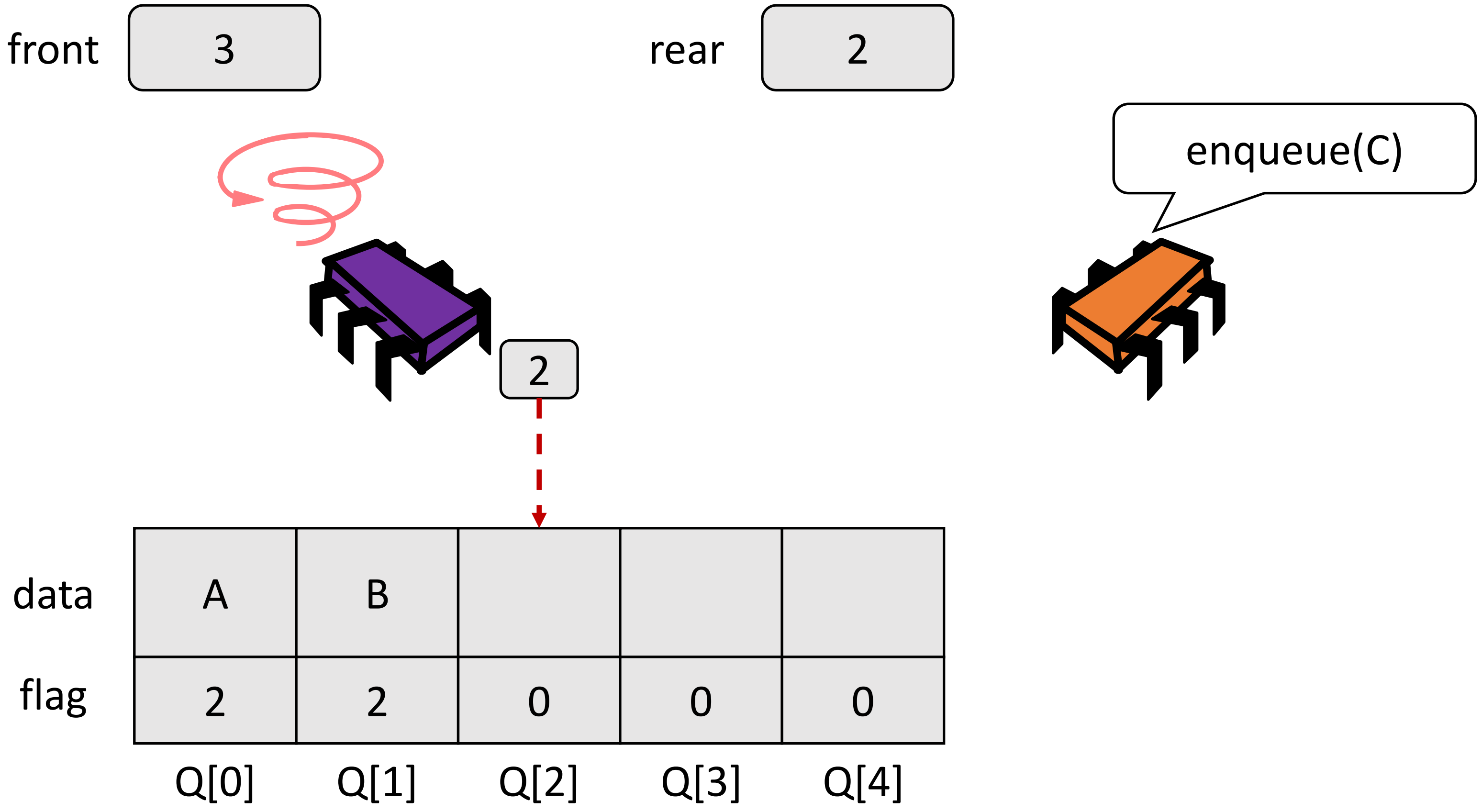
front 3

rear 2



	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B			
flag	2	2	0	0	0

Dequeue



front 3

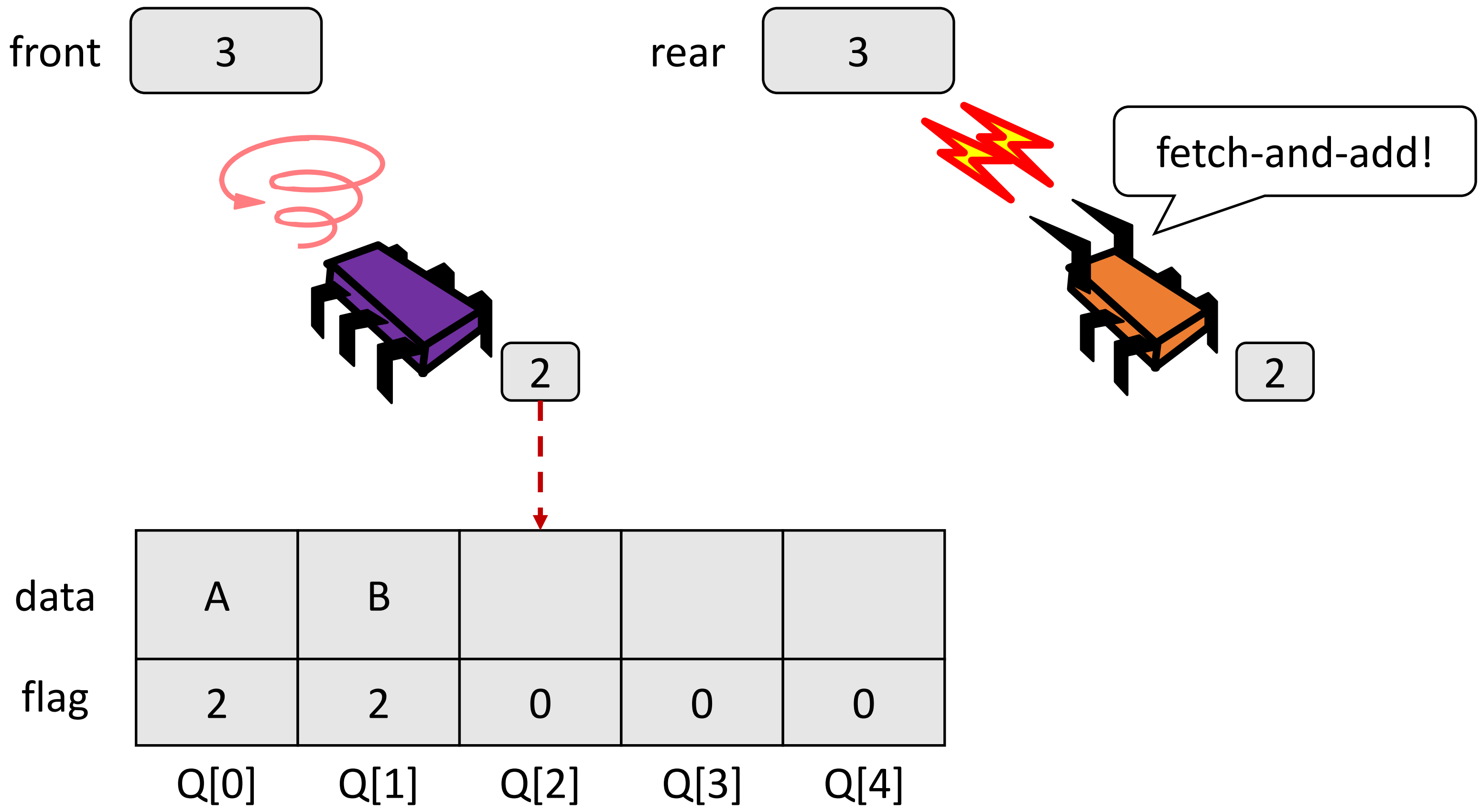
rear 2

enqueue(C)

2

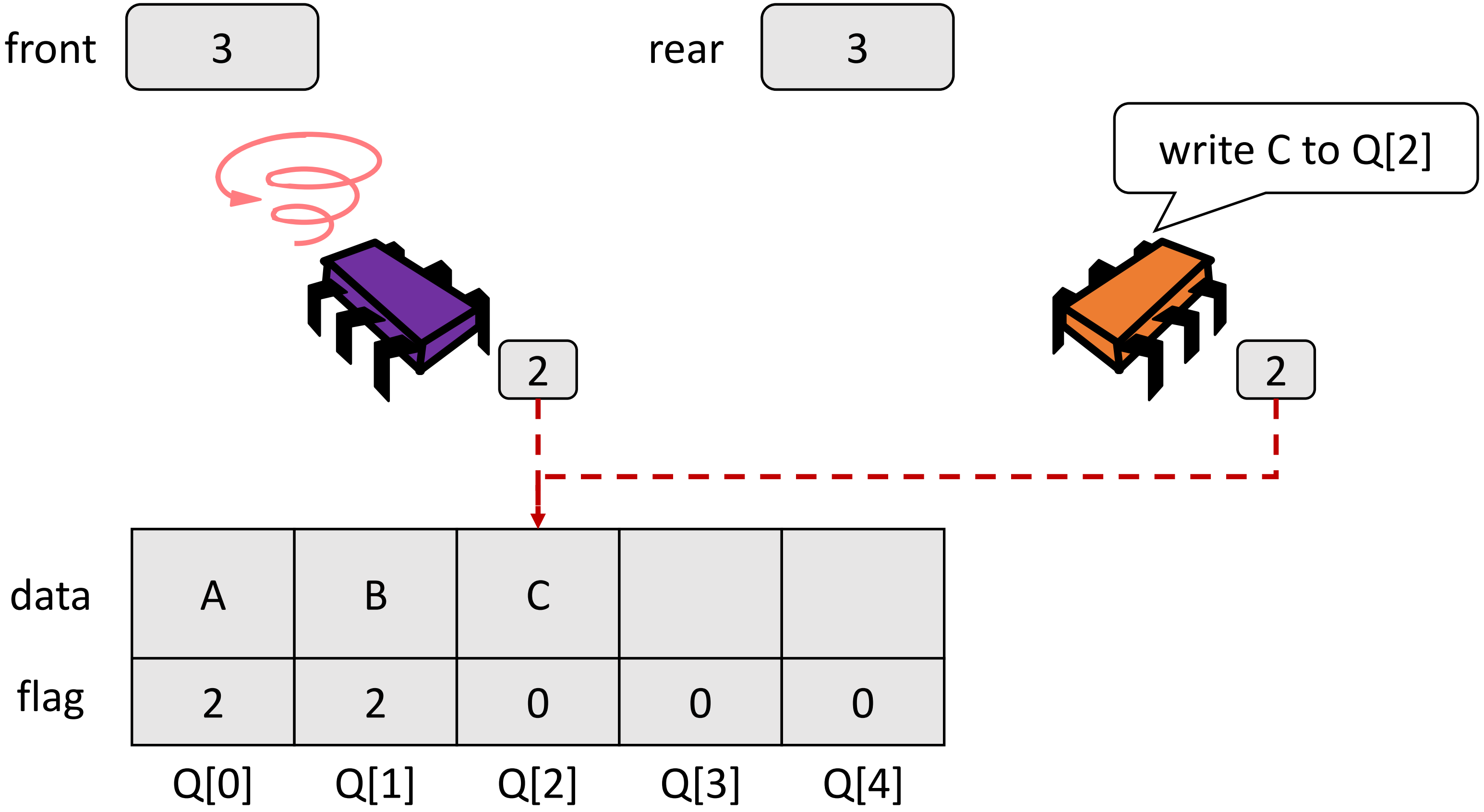
	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B			
flag	2	2	0	0	0

Dequeue



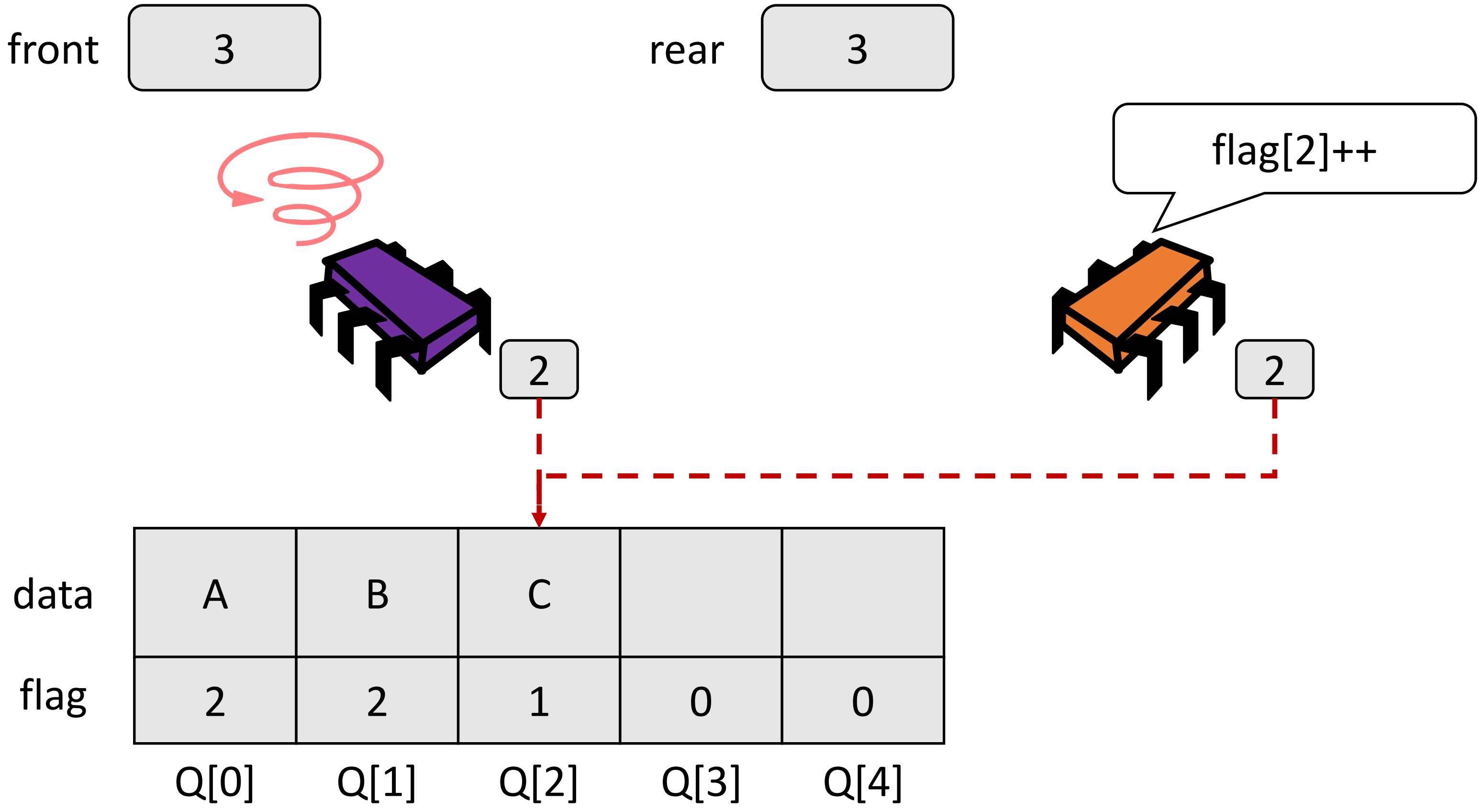
	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B			
flag	2	2	0	0	0

Dequeue



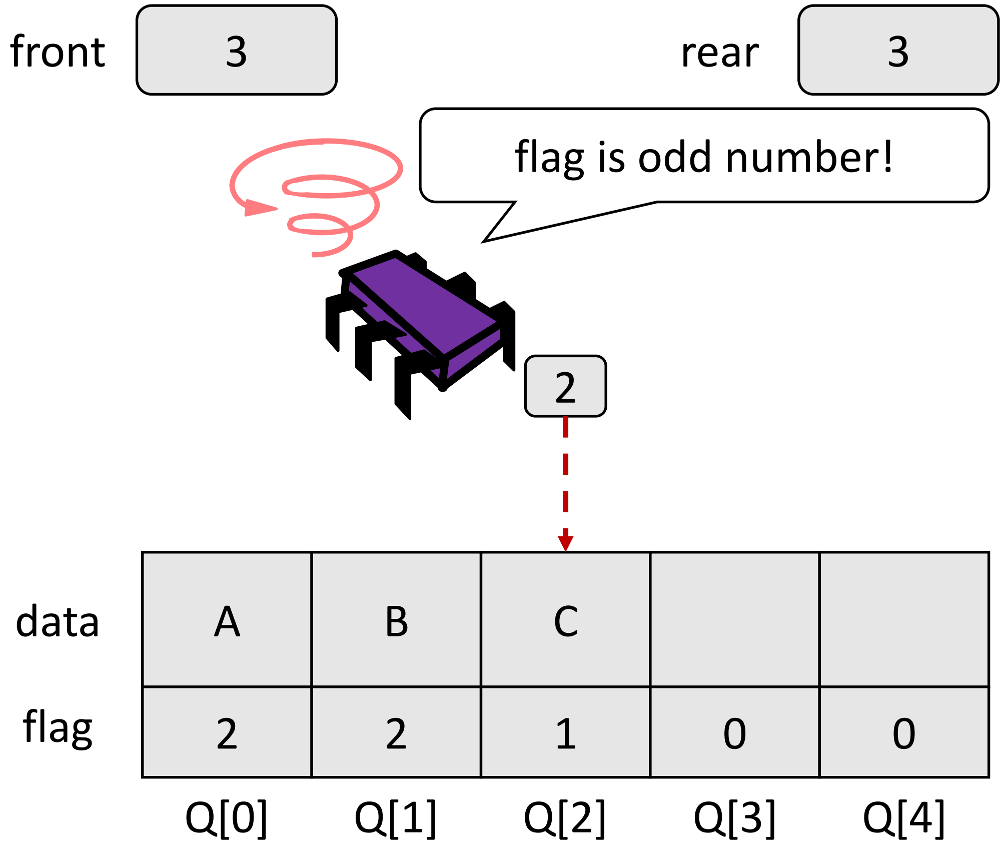
	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B	C		
flag	2	2	0	0	0

Dequeue



	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B	C		
flag	2	2	1	0	0

Dequeue



front 3

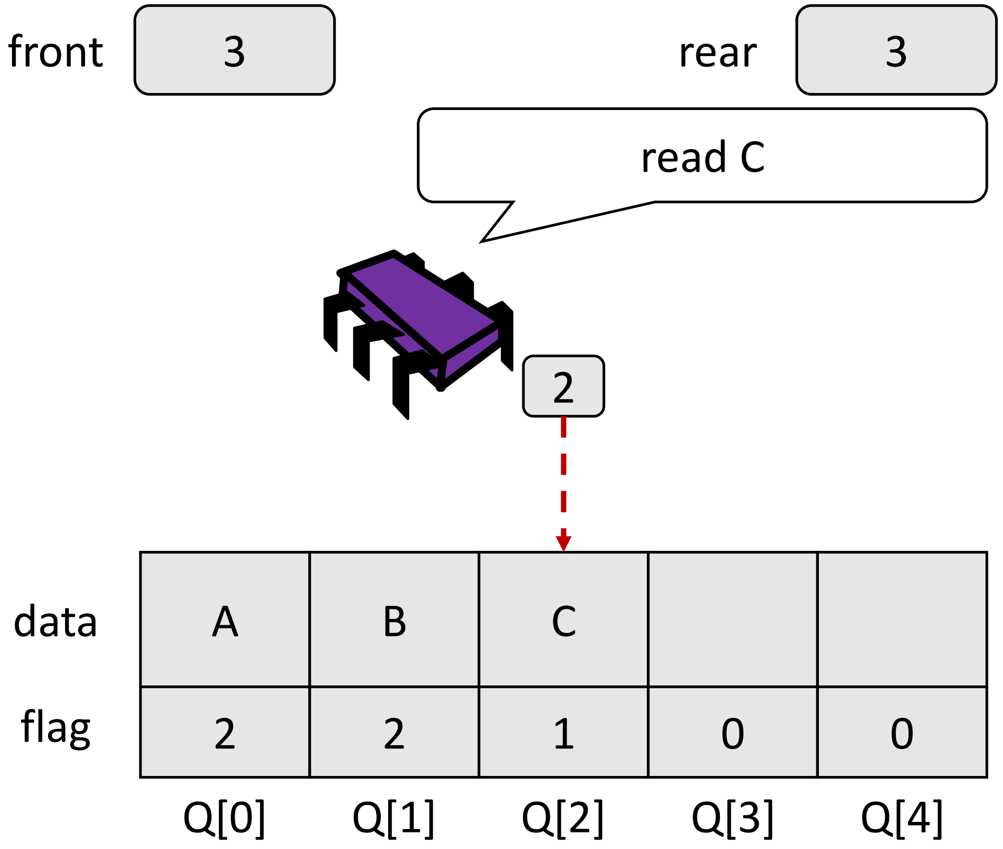
rear 3

flag is odd number!

2

	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B	C		
flag	2	2	1	0	0

Dequeue



front 3

rear 3

read C

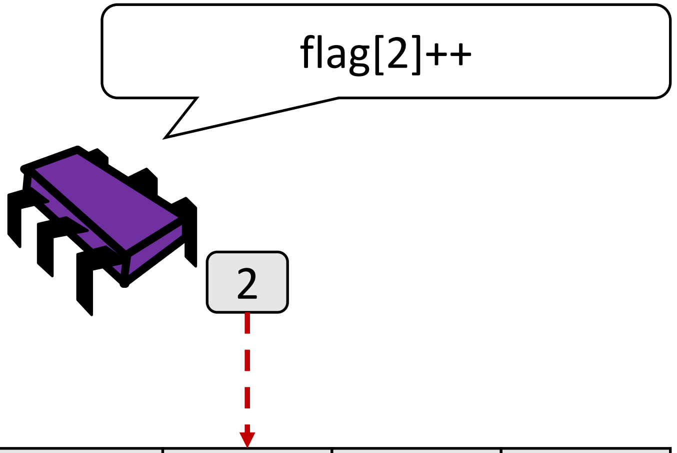
2

	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B	C		
flag	2	2	1	0	0

Dequeue

front 3

rear 3



data	A	B	C		
flag	2	2	2	0	0
	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]

Dequeue

front 3

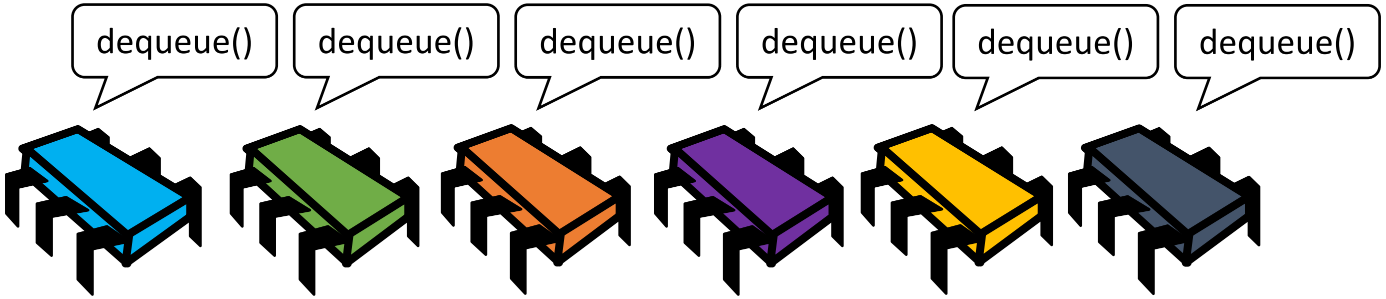
rear 3

	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B	C		
flag	2	2	2	0	0

Dequeue

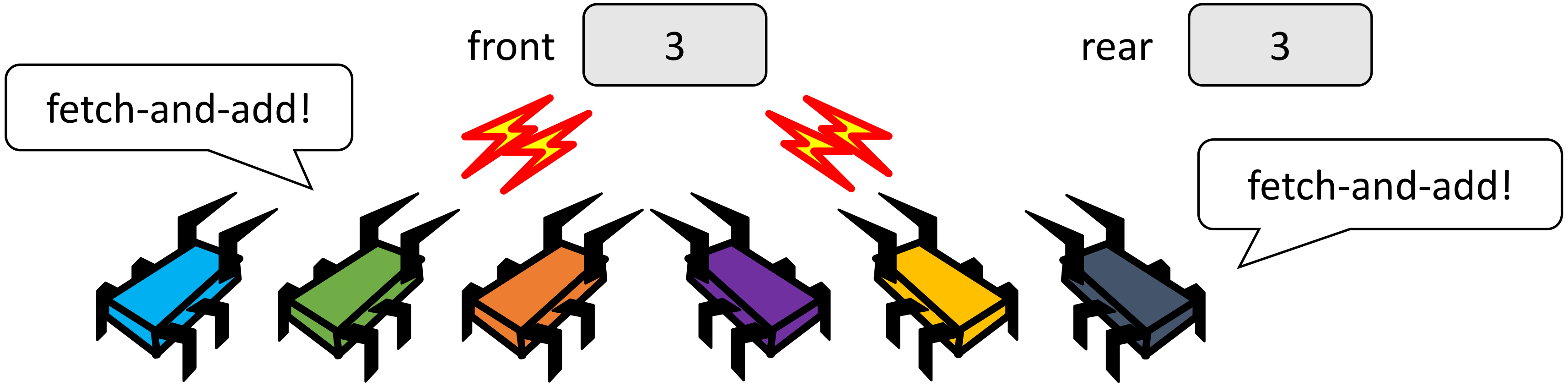
front 3

rear 3



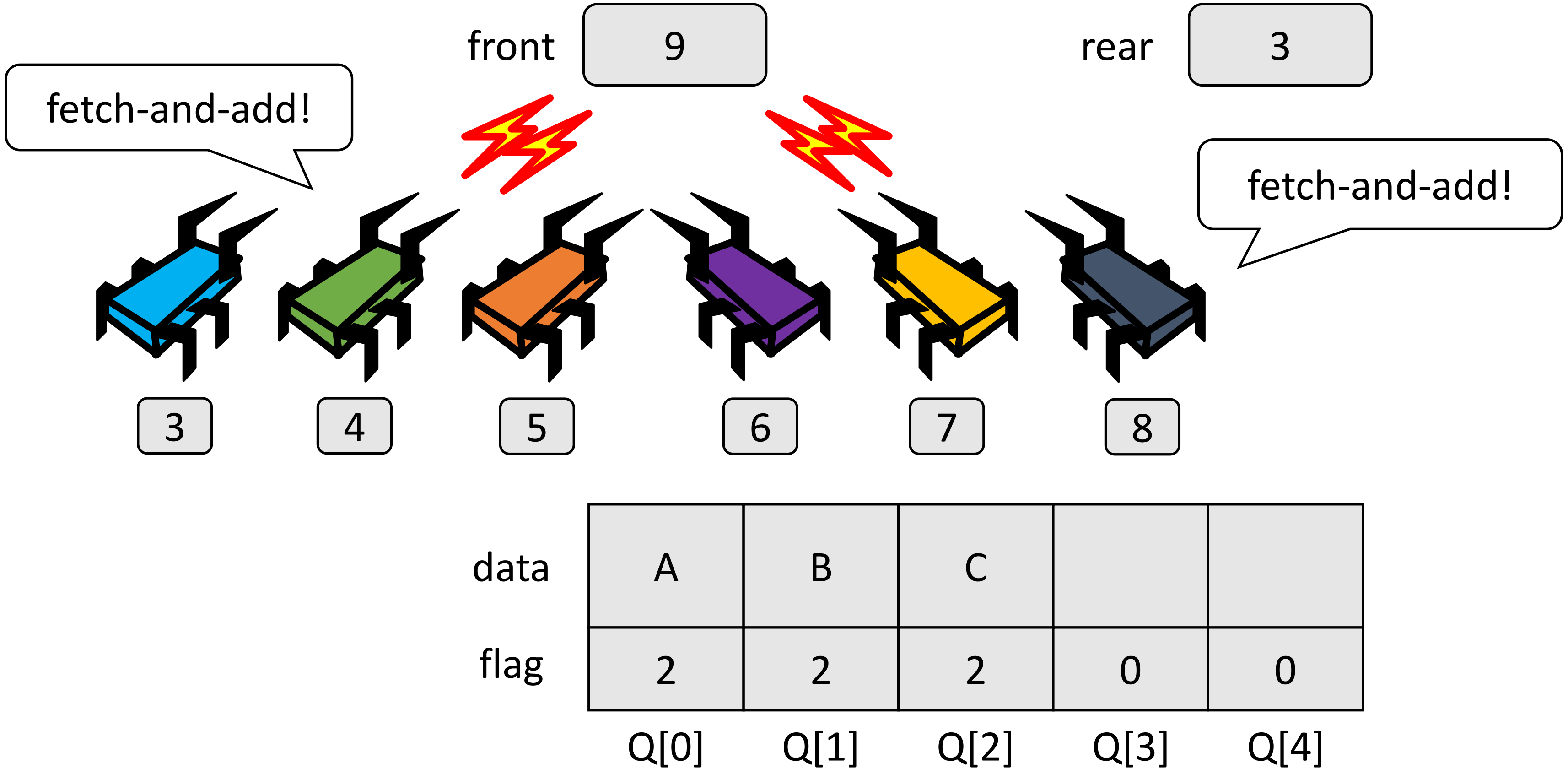
	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B	C		
flag	2	2	2	0	0

Dequeue



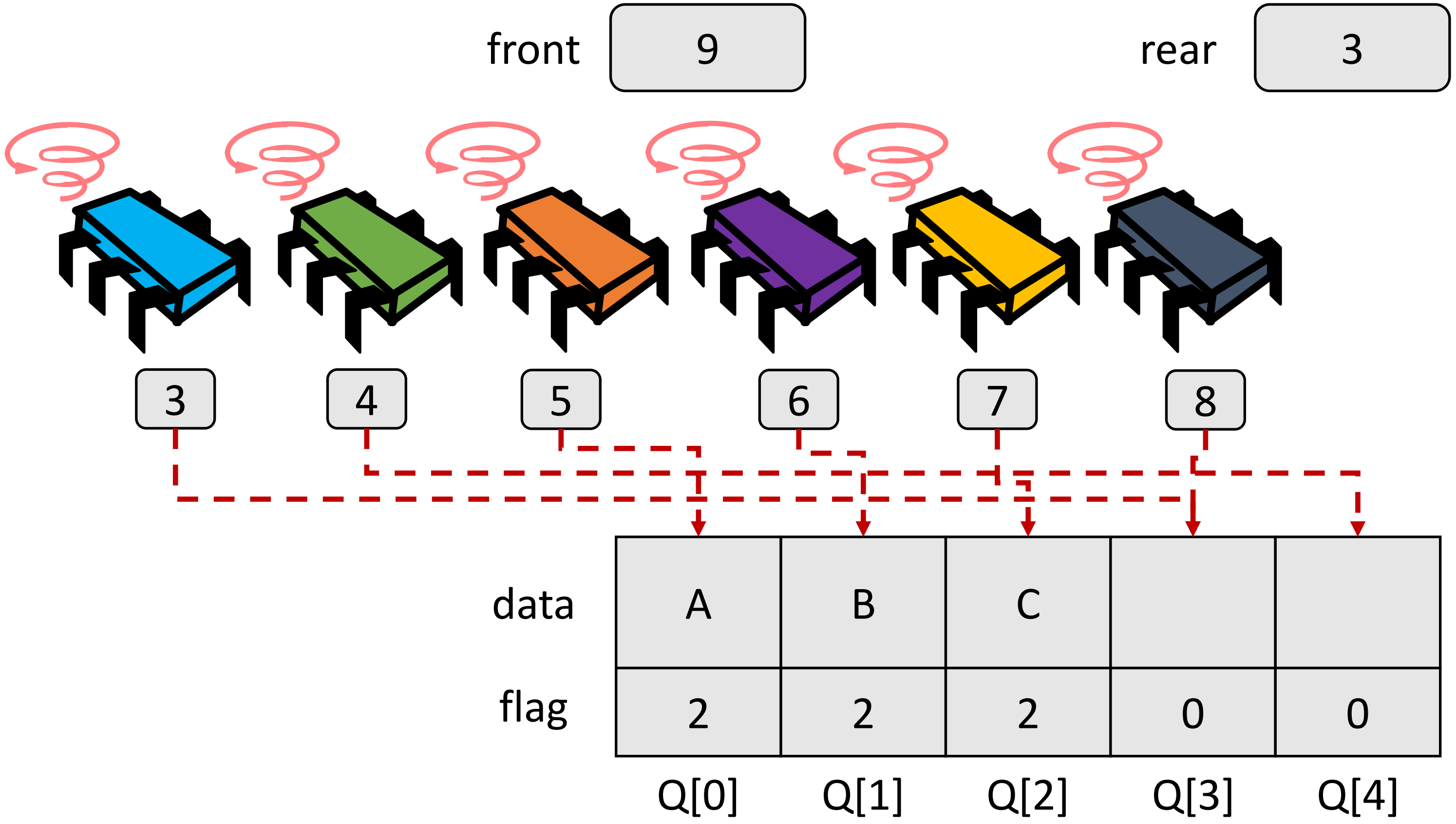
	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B	C		
flag	2	2	2	0	0

Dequeue



	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B	C		
flag	2	2	2	0	0

Dequeue

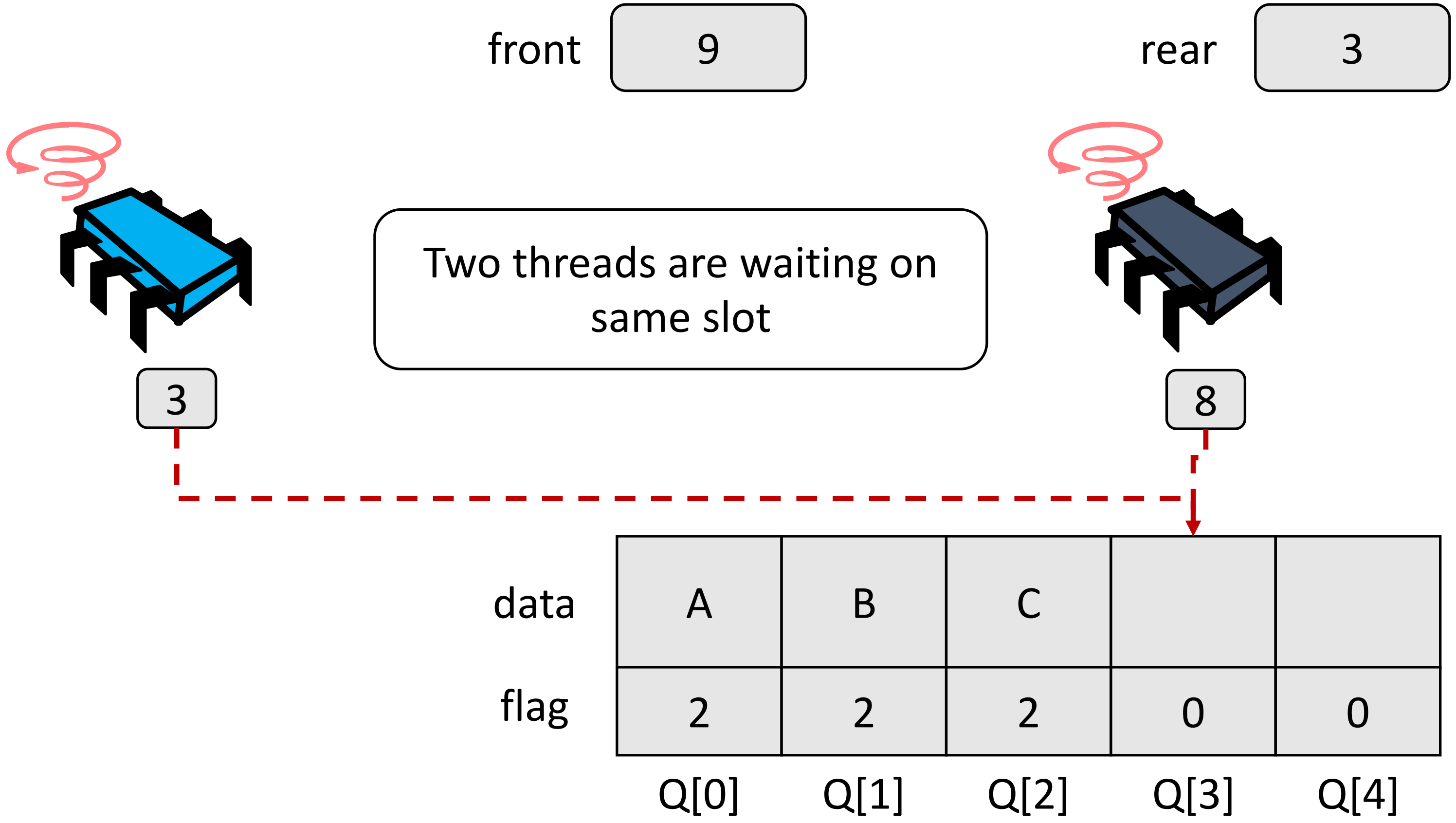


front 9

rear 3

	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B	C		
flag	2	2	2	0	0

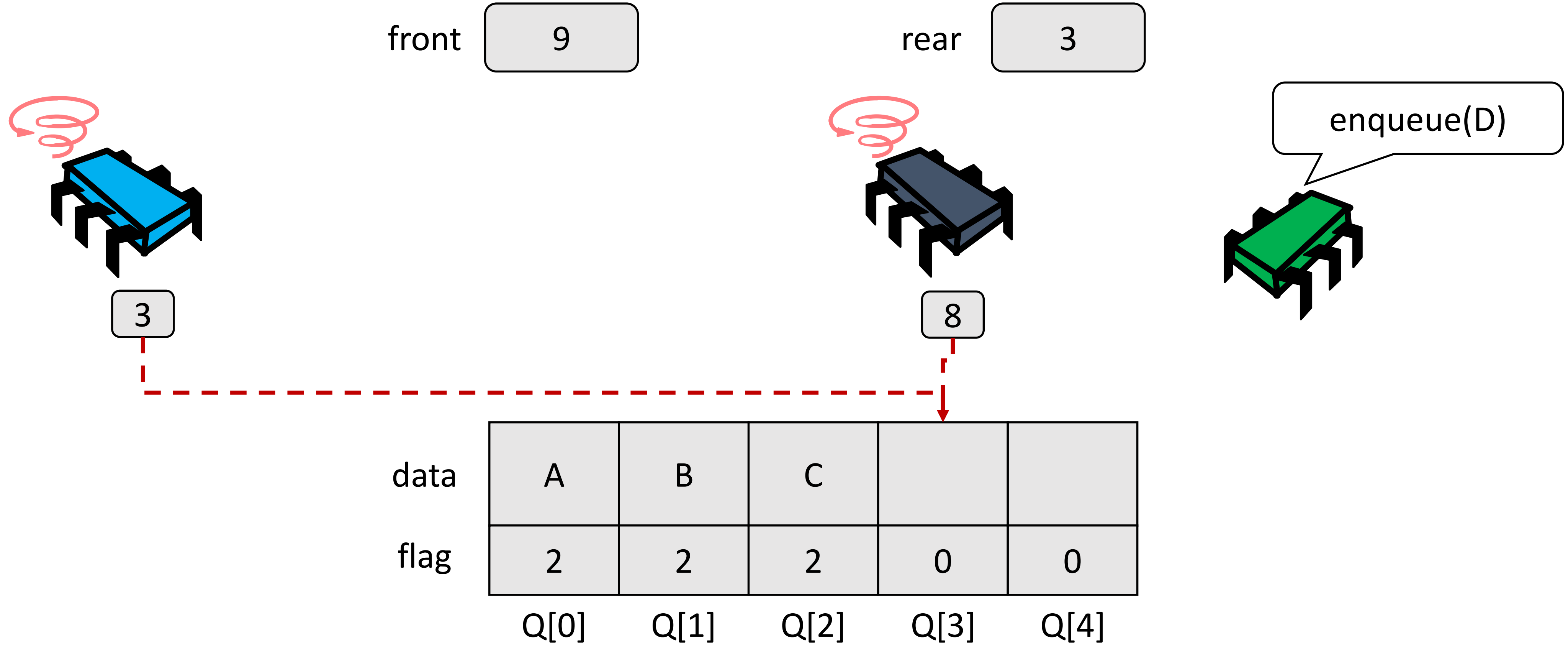
Dequeue



Two threads are waiting on same slot

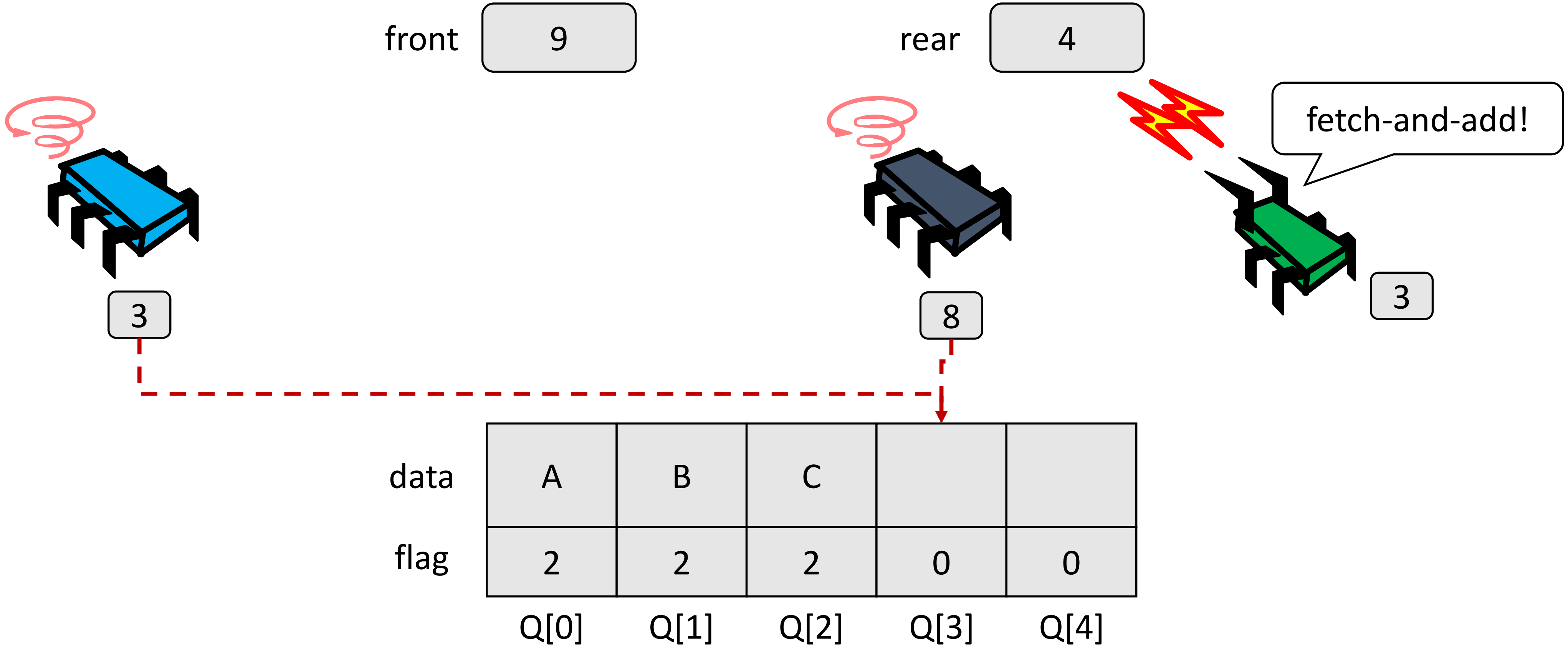
	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B	C		
flag	2	2	2	0	0

Dequeue



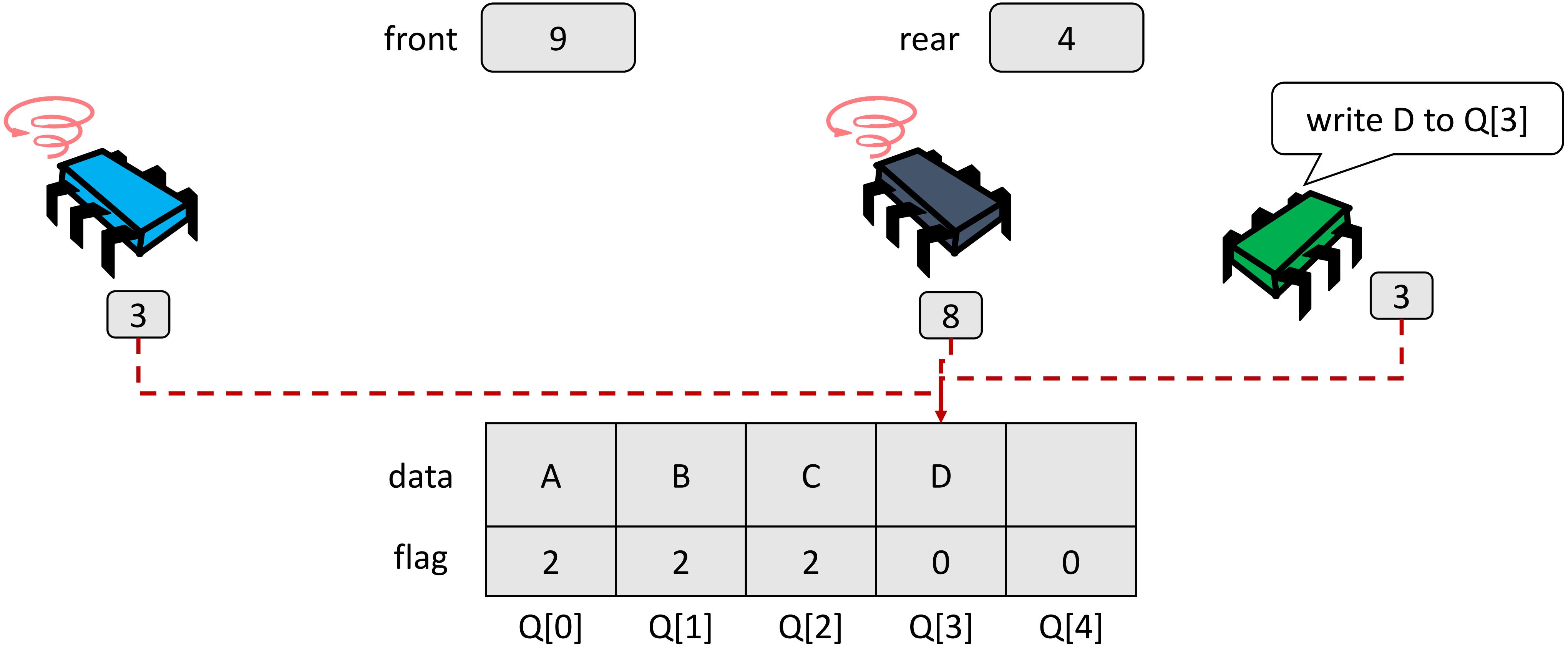
	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B	C		
flag	2	2	2	0	0

Dequeue



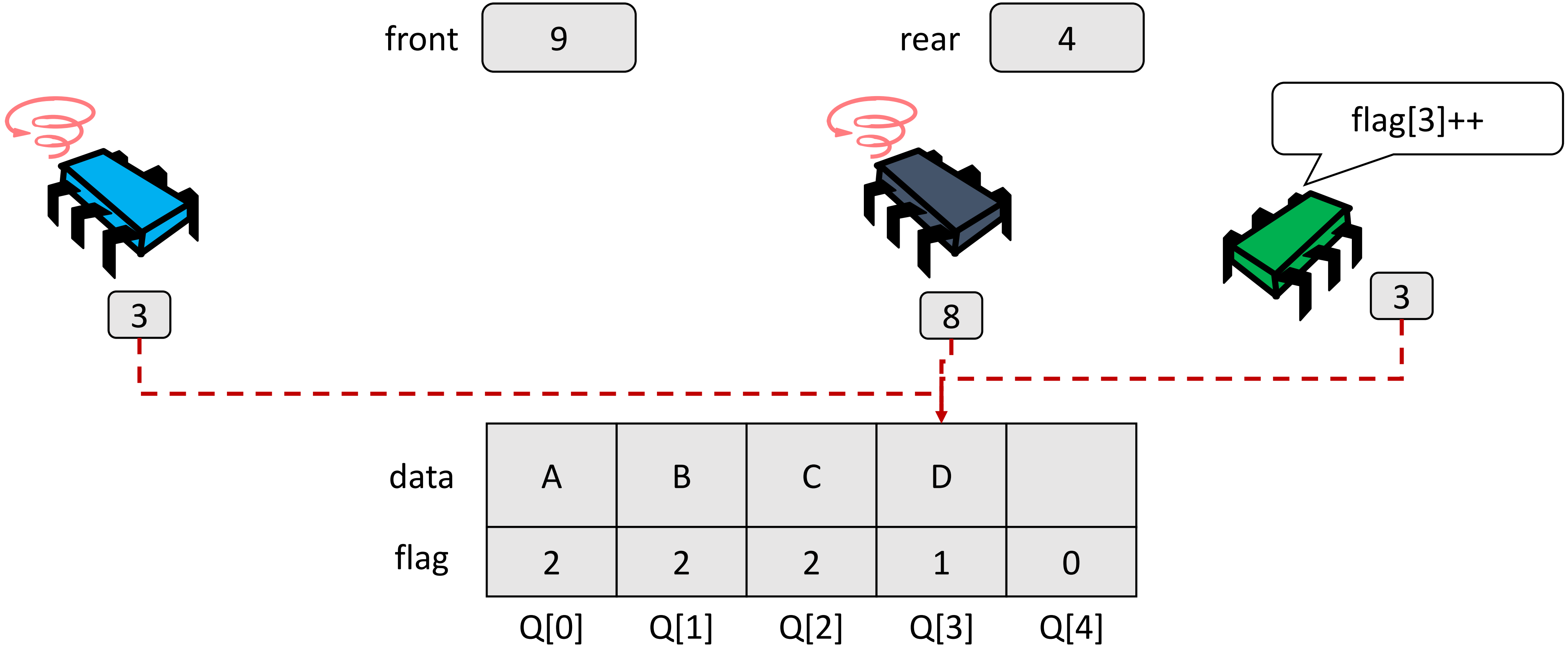
	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B	C		
flag	2	2	2	0	0

Dequeue



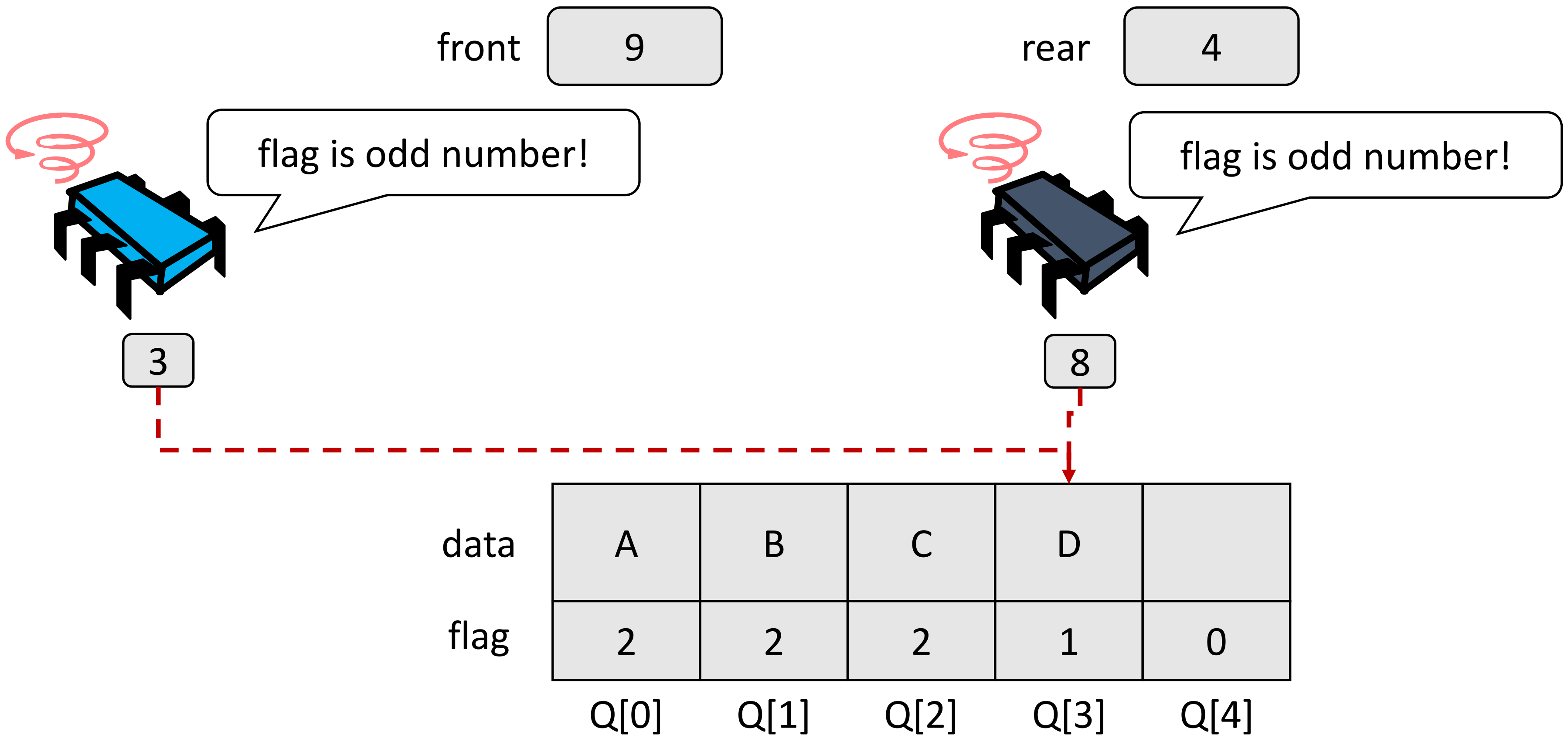
	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B	C	D	
flag	2	2	2	0	0

Dequeue



	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B	C	D	
flag	2	2	2	1	0

Dequeue



front 9

rear 4

flag is odd number!

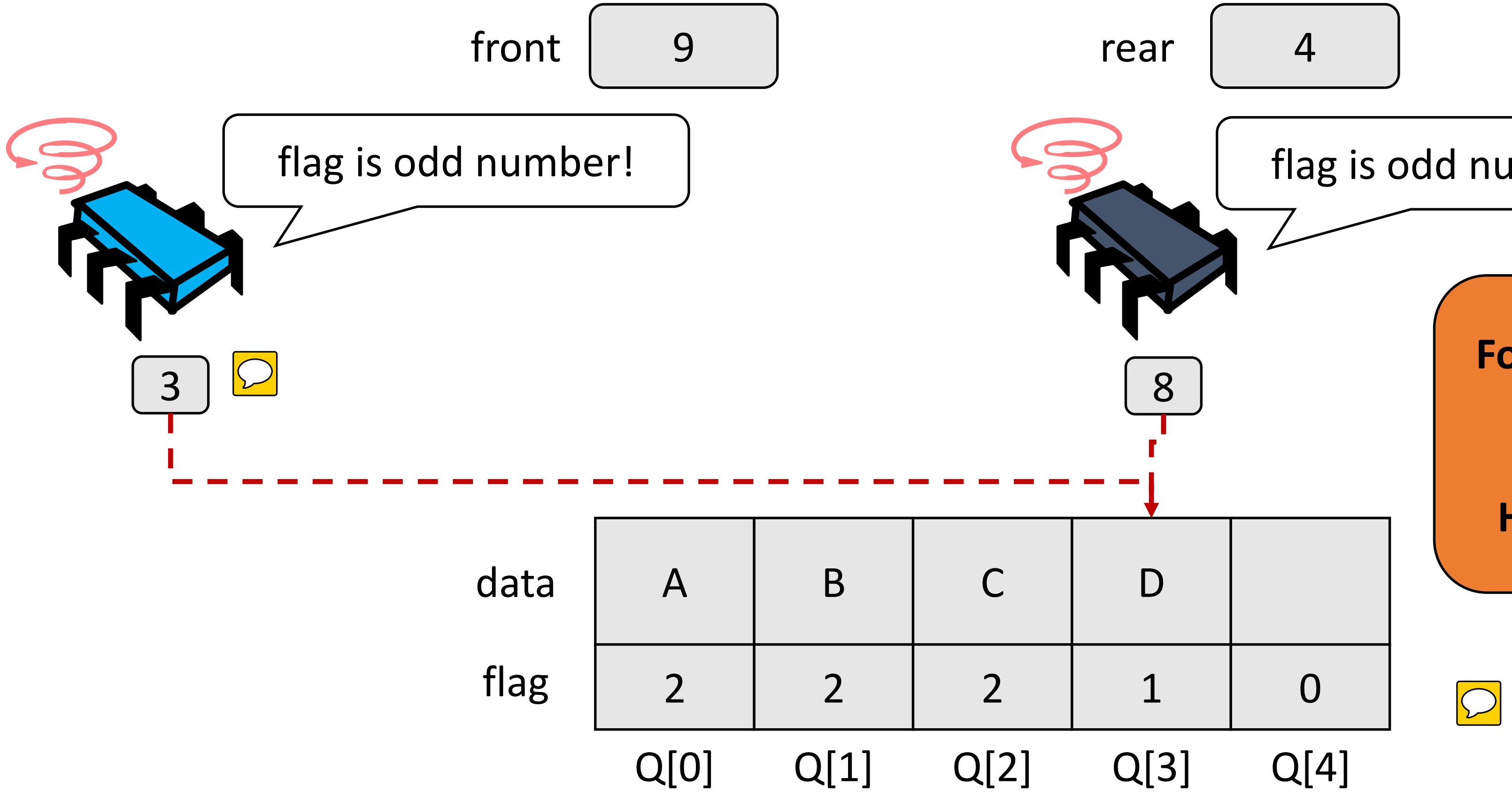
flag is odd number!

3

8

	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B	C	D	
flag	2	2	2	1	0

Dequeue

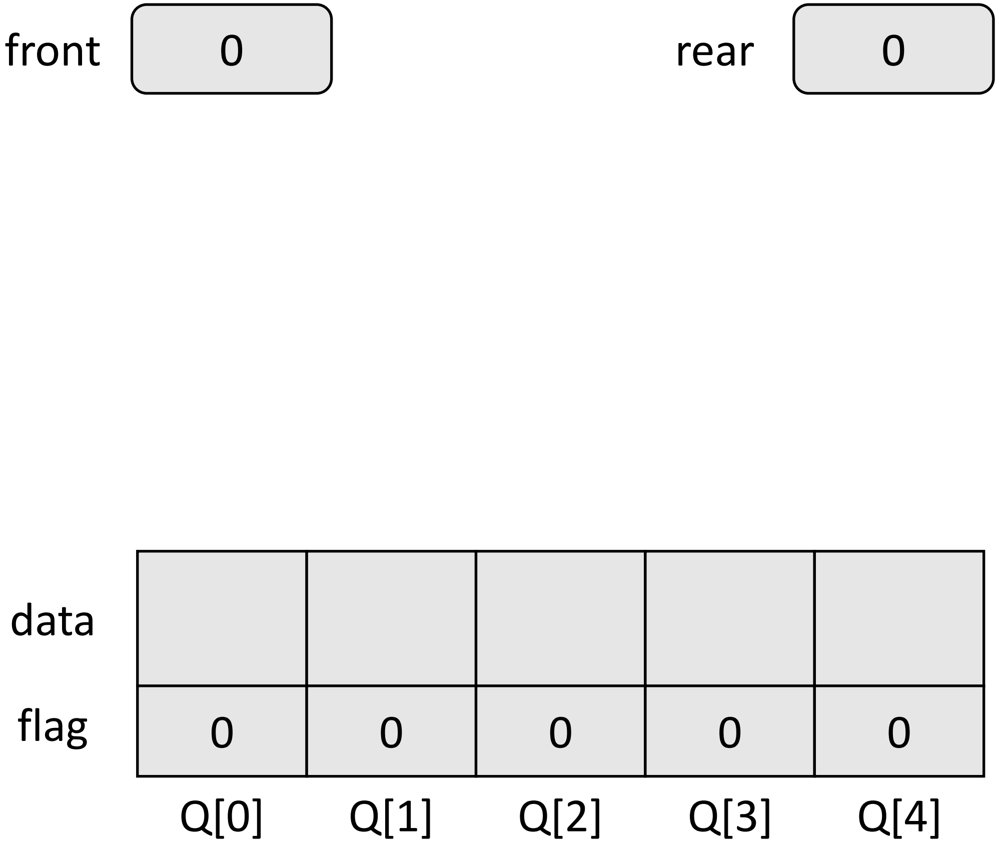


	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B	C	D	
flag	2	2	2	1	0

For the fairness, which thread should read the data D?

How can you implement it?

Enqueue



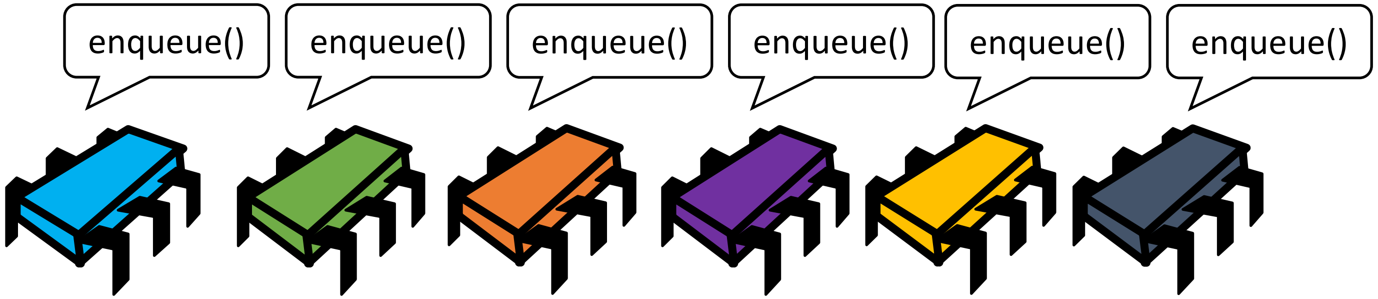
front 0

rear 0

	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data					
flag	0	0	0	0	0

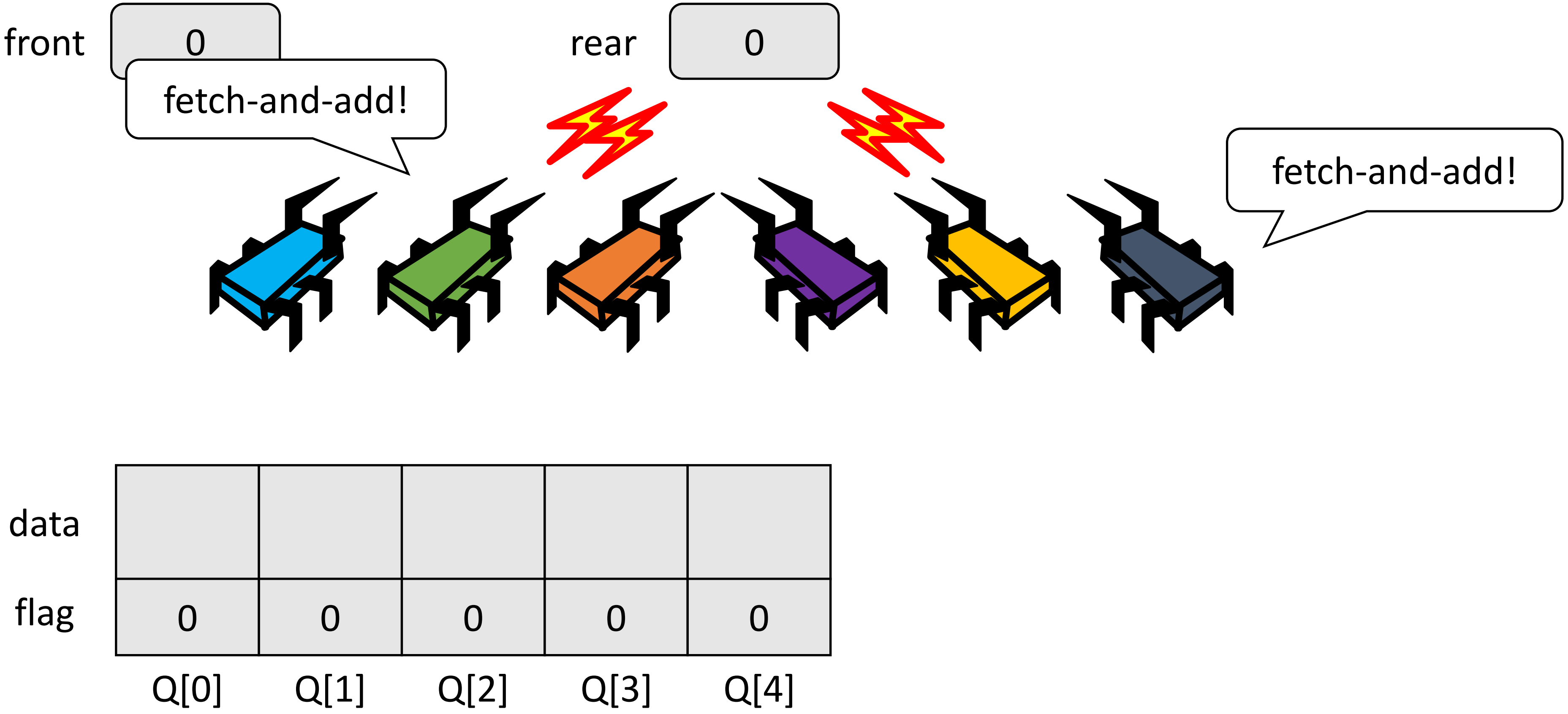
Enqueue

front 0 rear 0



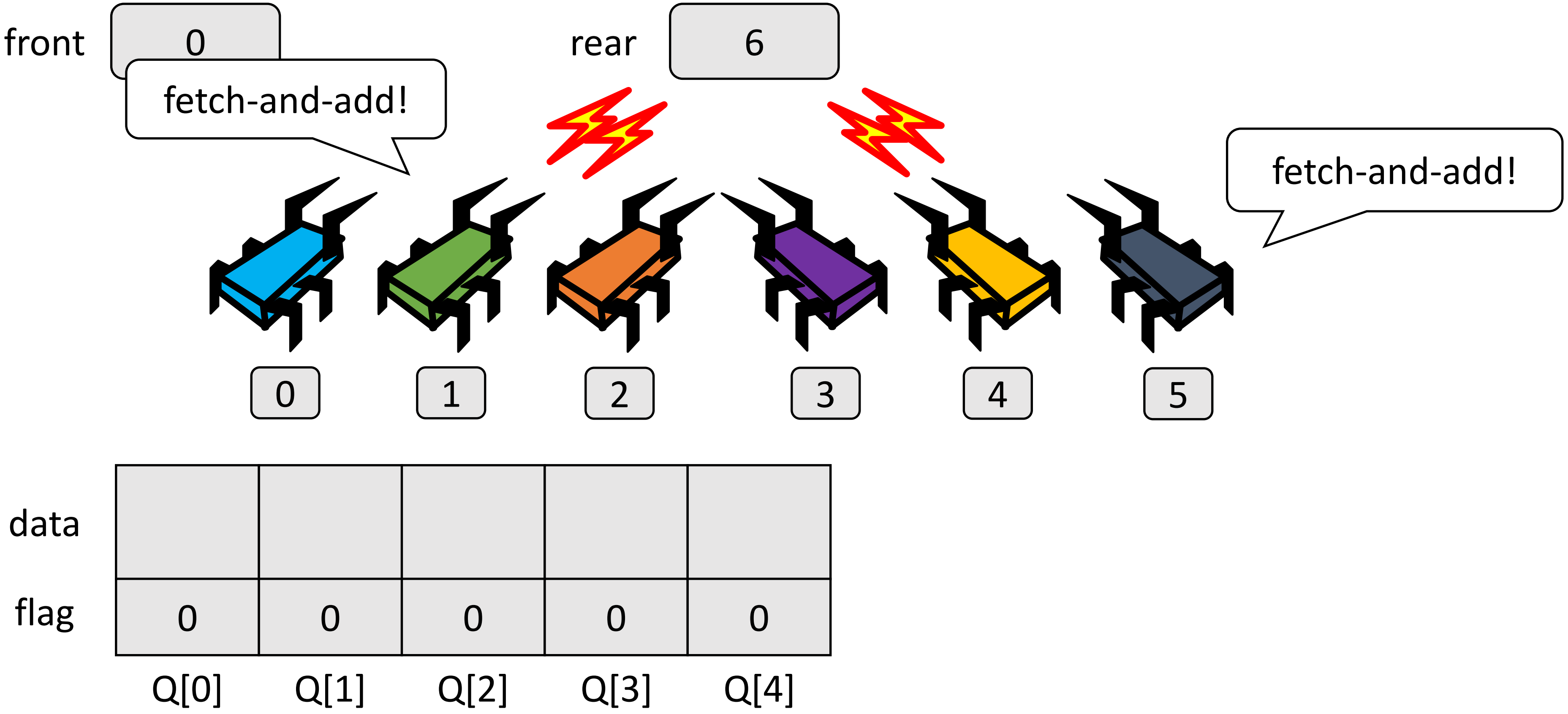
	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data					
flag	0	0	0	0	0

Enqueue



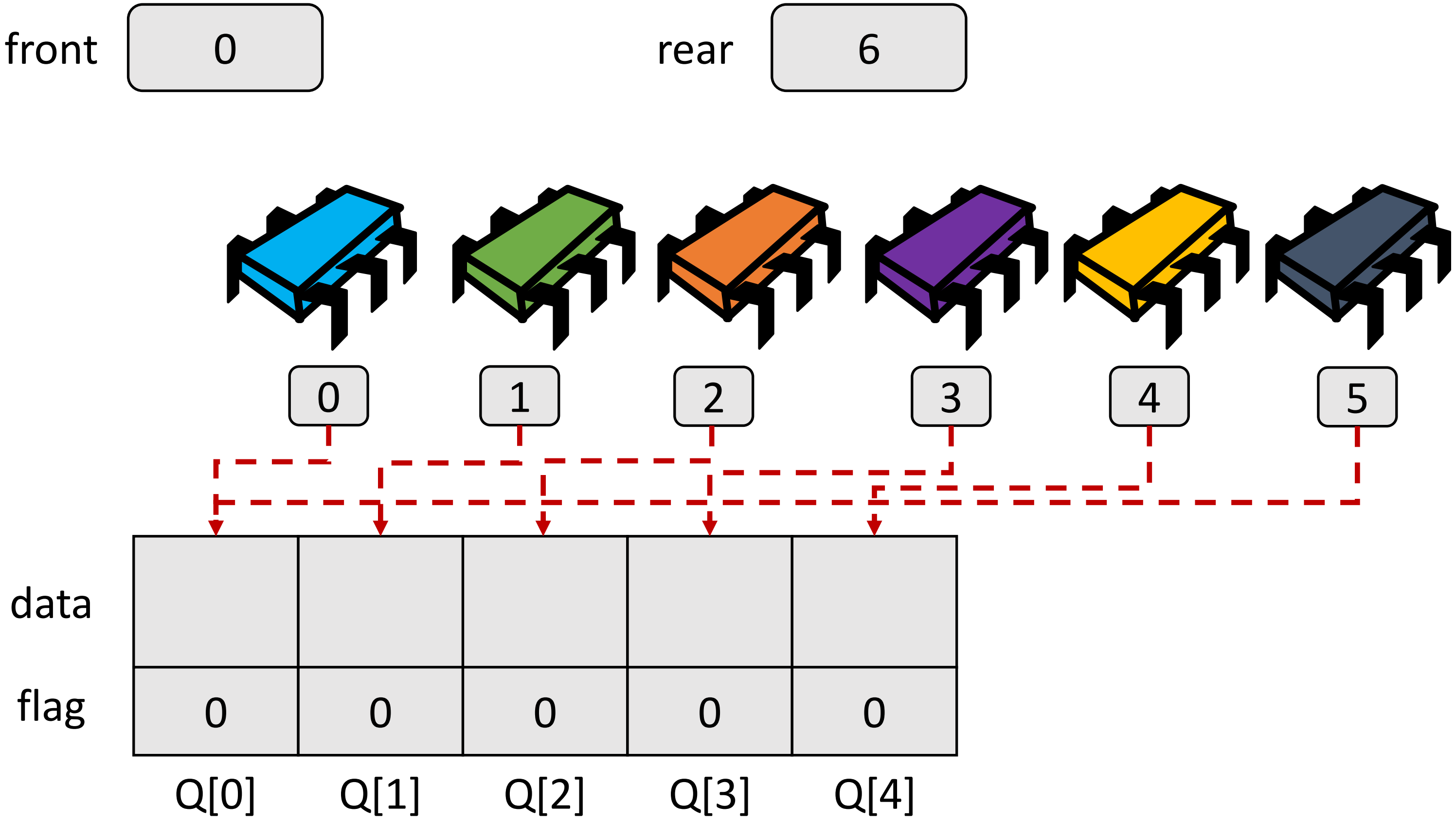
	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data					
flag	0	0	0	0	0

Enqueue



	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data					
flag	0	0	0	0	0

Enqueue

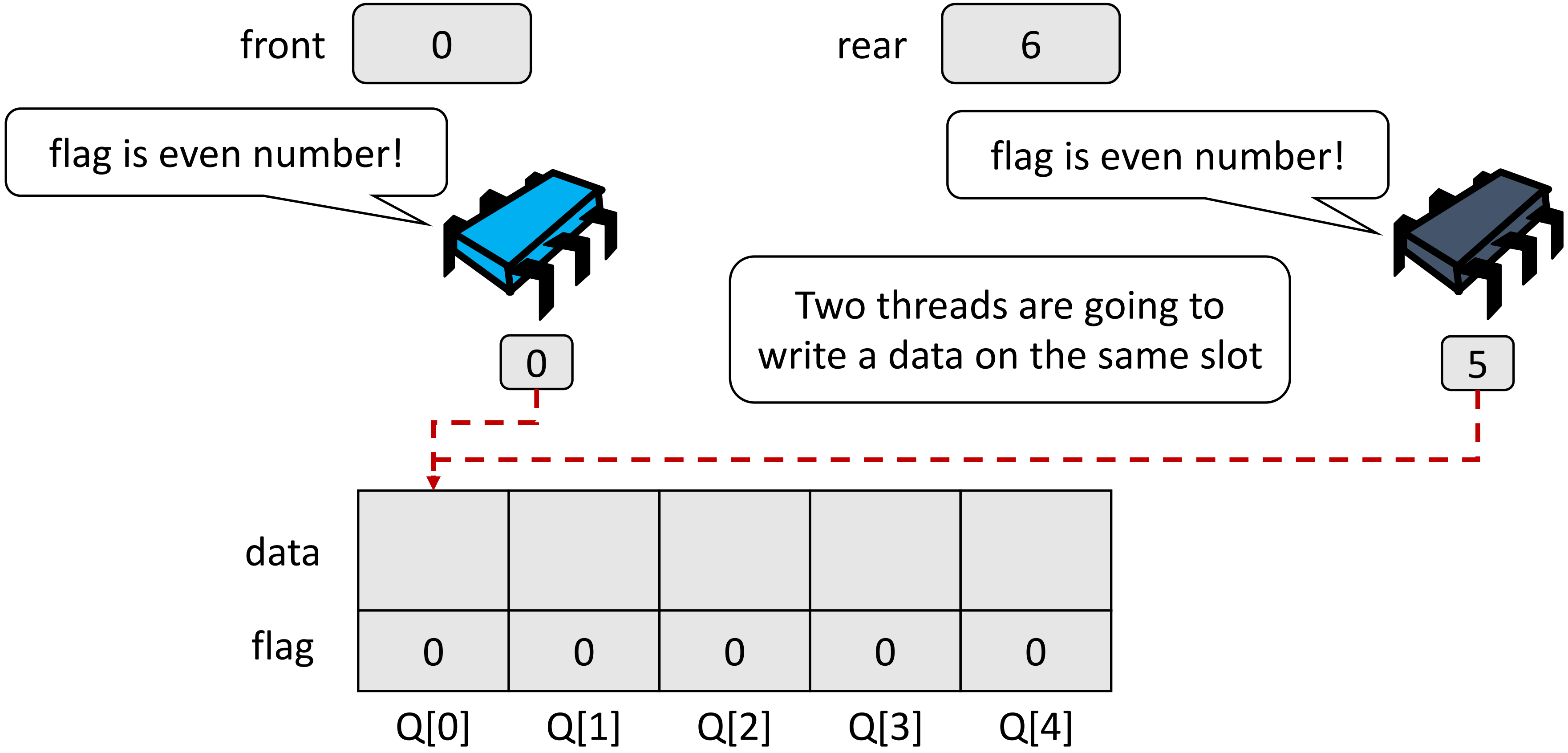


front 0

rear 6

	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data					
flag	0	0	0	0	0

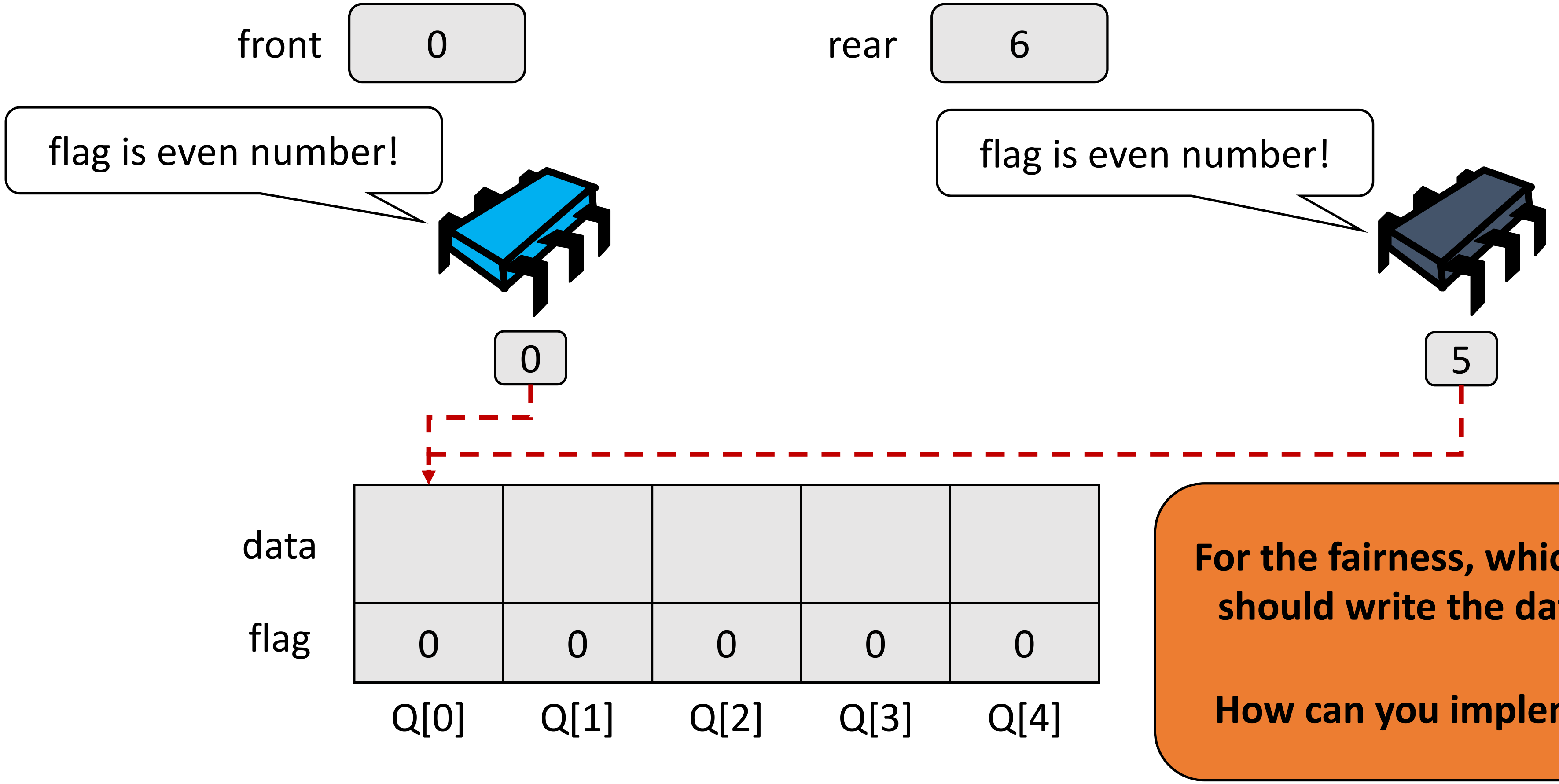
Enqueue



Two threads are going to write a data on the same slot

	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data					
flag	0	0	0	0	0

Enqueue



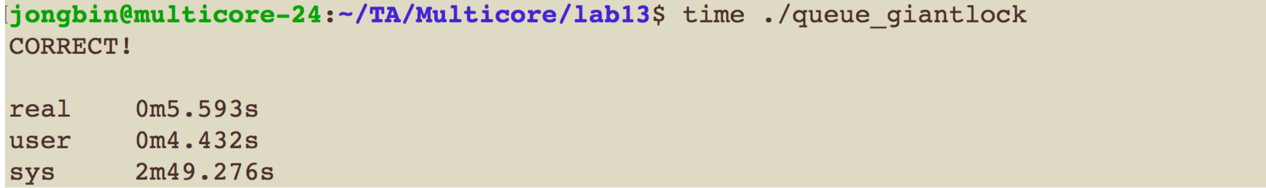
	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data					
flag	0	0	0	0	0

For the fairness, which thread should write the data first?

How can you implement it?

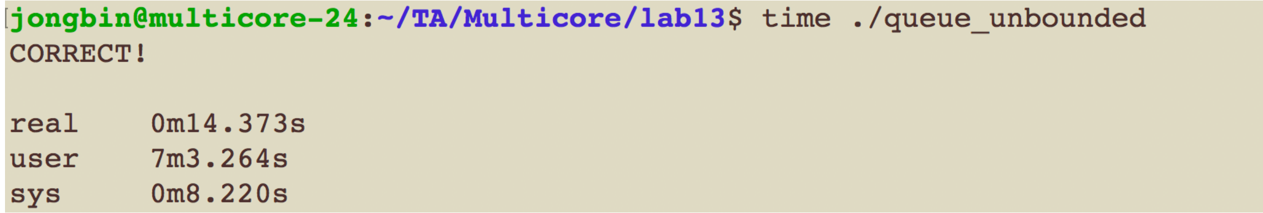
Evaluation

CPU: 24 x 2 (Hyperthreading enabled)
Number of producer / consumer threads: 16
Number of enqueue / dequeue operations per thread: 1,000,000



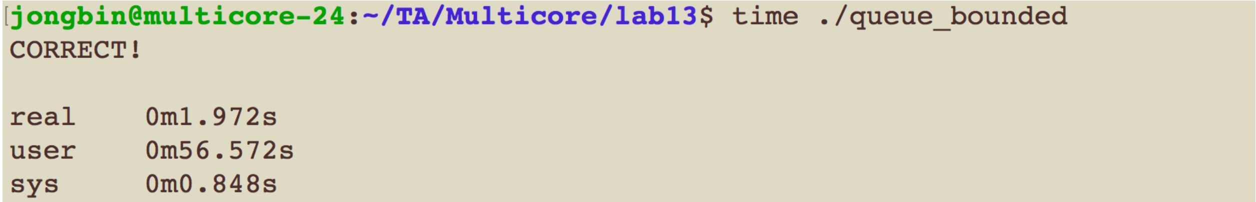
```
jongbin@multicore-24:~/TA/Multicore/lab13$ time ./queue_giantlock
CORRECT!

real	0m5.593s
user	0m4.432s
sys	2m49.276s
```



```
jongbin@multicore-24:~/TA/Multicore/lab13$ time ./queue_unbounded
CORRECT!

real	0m14.373s
user	7m3.264s
sys	0m8.220s
```



```
jongbin@multicore-24:~/TA/Multicore/lab13$ time ./queue_bounded
CORRECT!

real	0m1.972s
user	0m56.572s
sys	0m0.848s
```

Thank You