Concurrent Queue

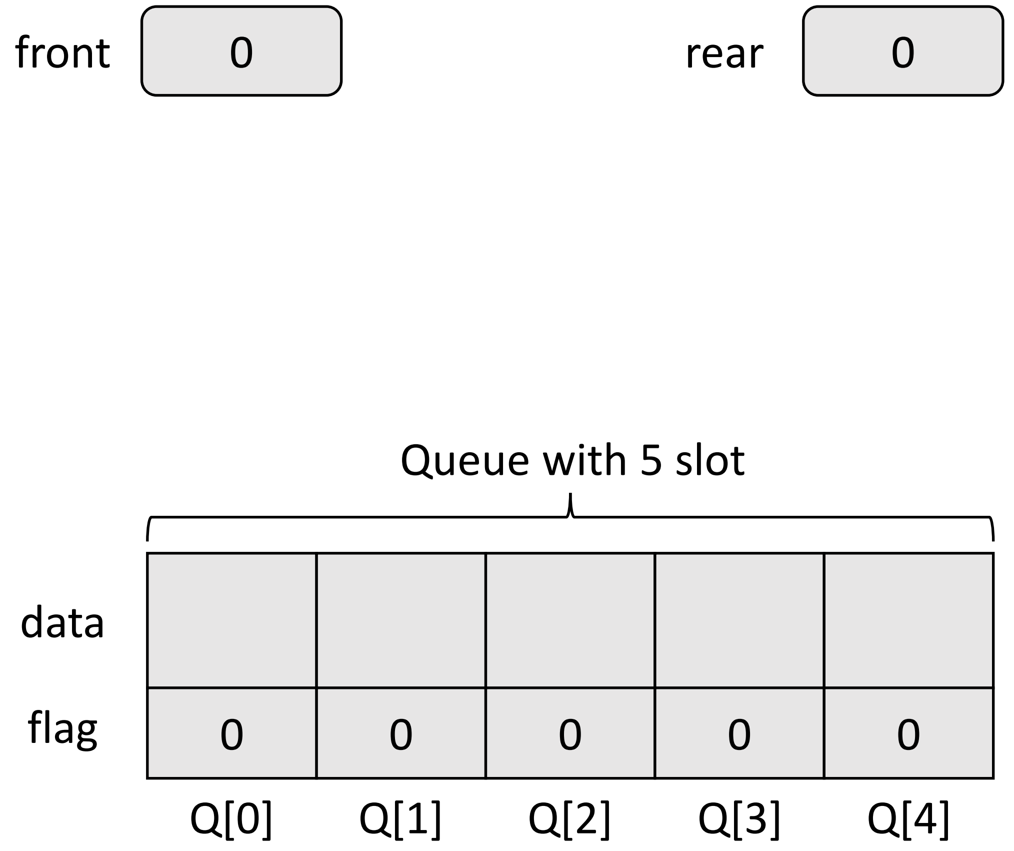
Concurrent Programming

Introduction

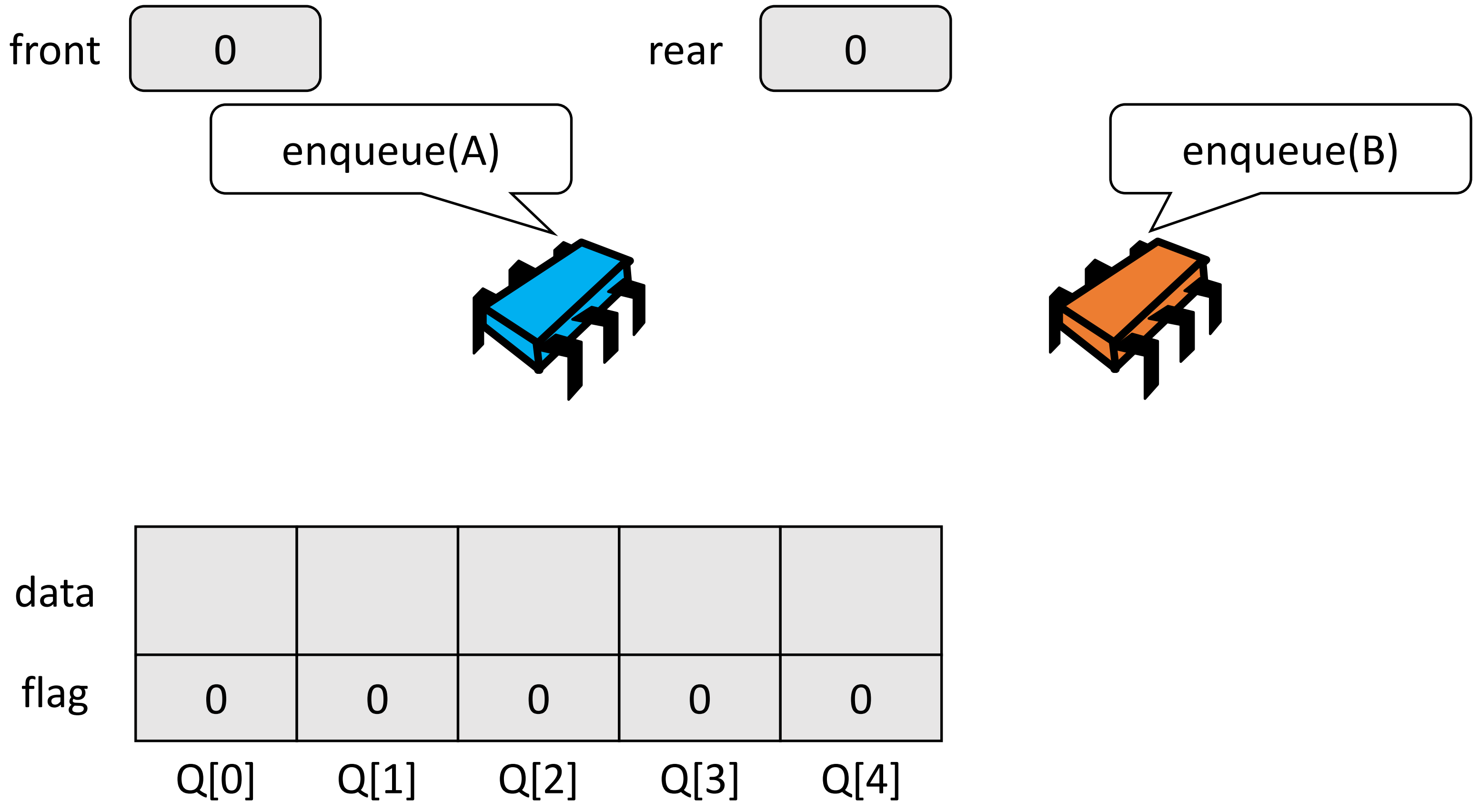
- Bounded Lock-free Queue

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

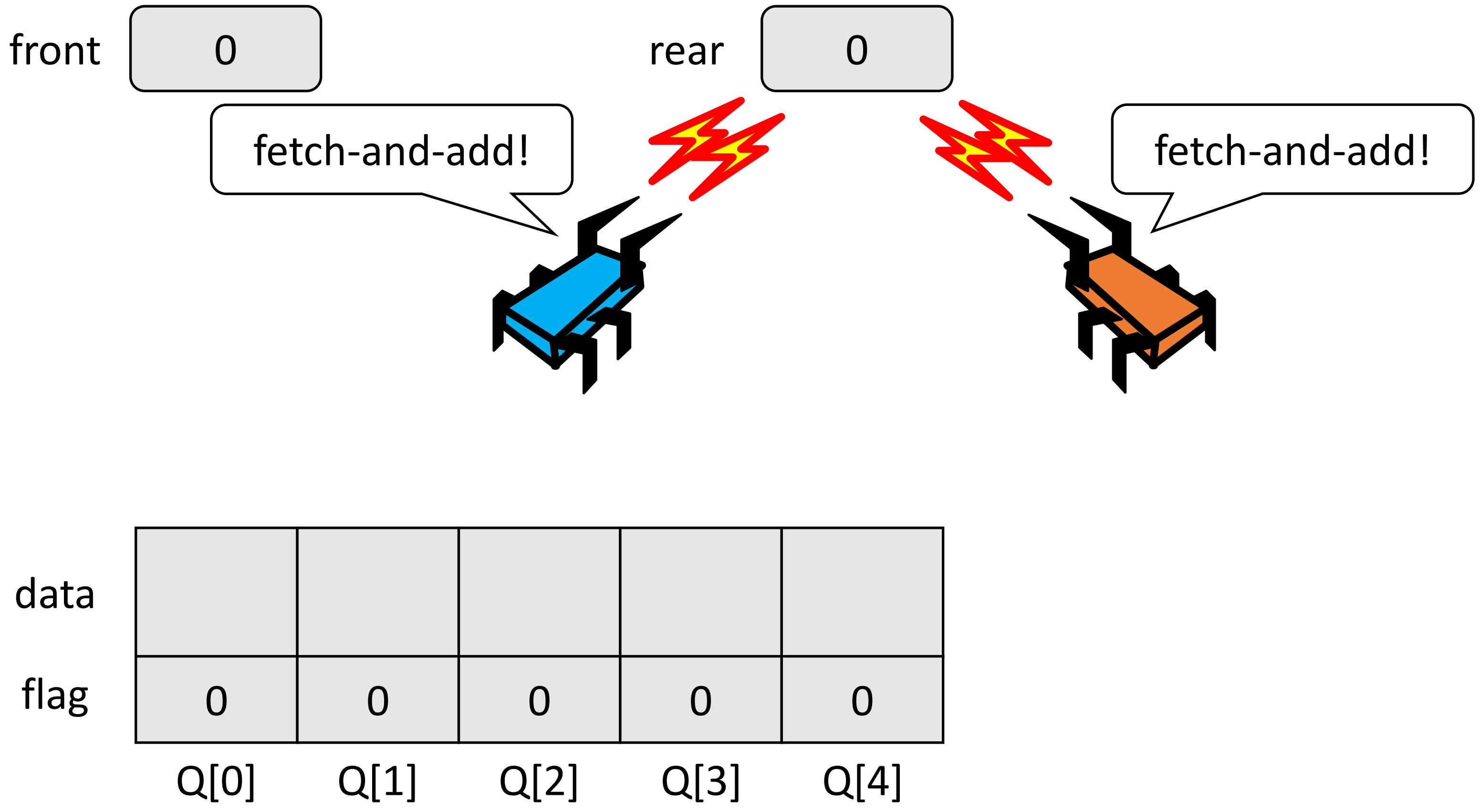
Bounded lock-free queue



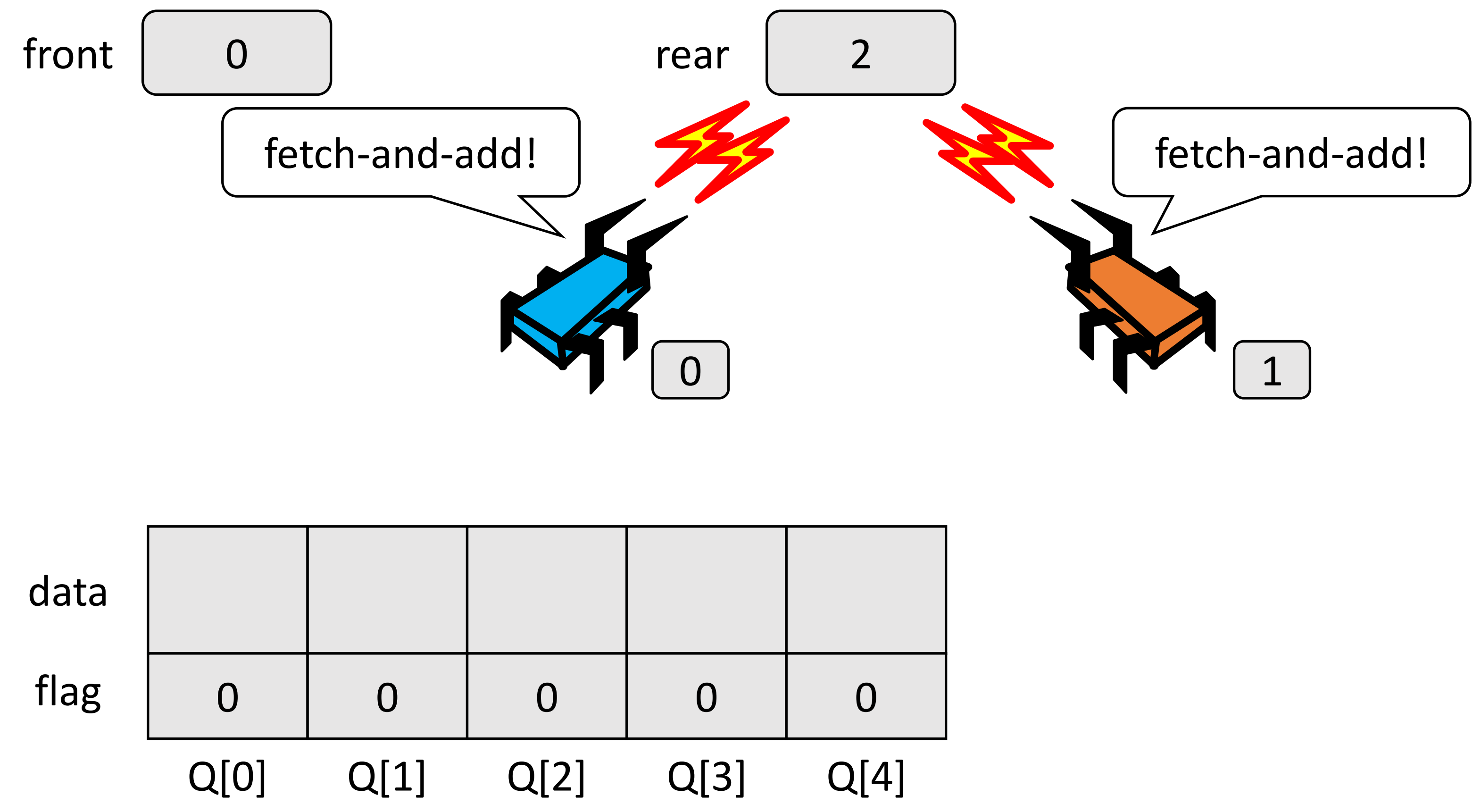
Enqueue


front 0
rear 0
enqueue(A)
enqueue(B)
data
flag
0 0 0 0 0
Q[0] Q[1] Q[2] Q[3] Q[4]

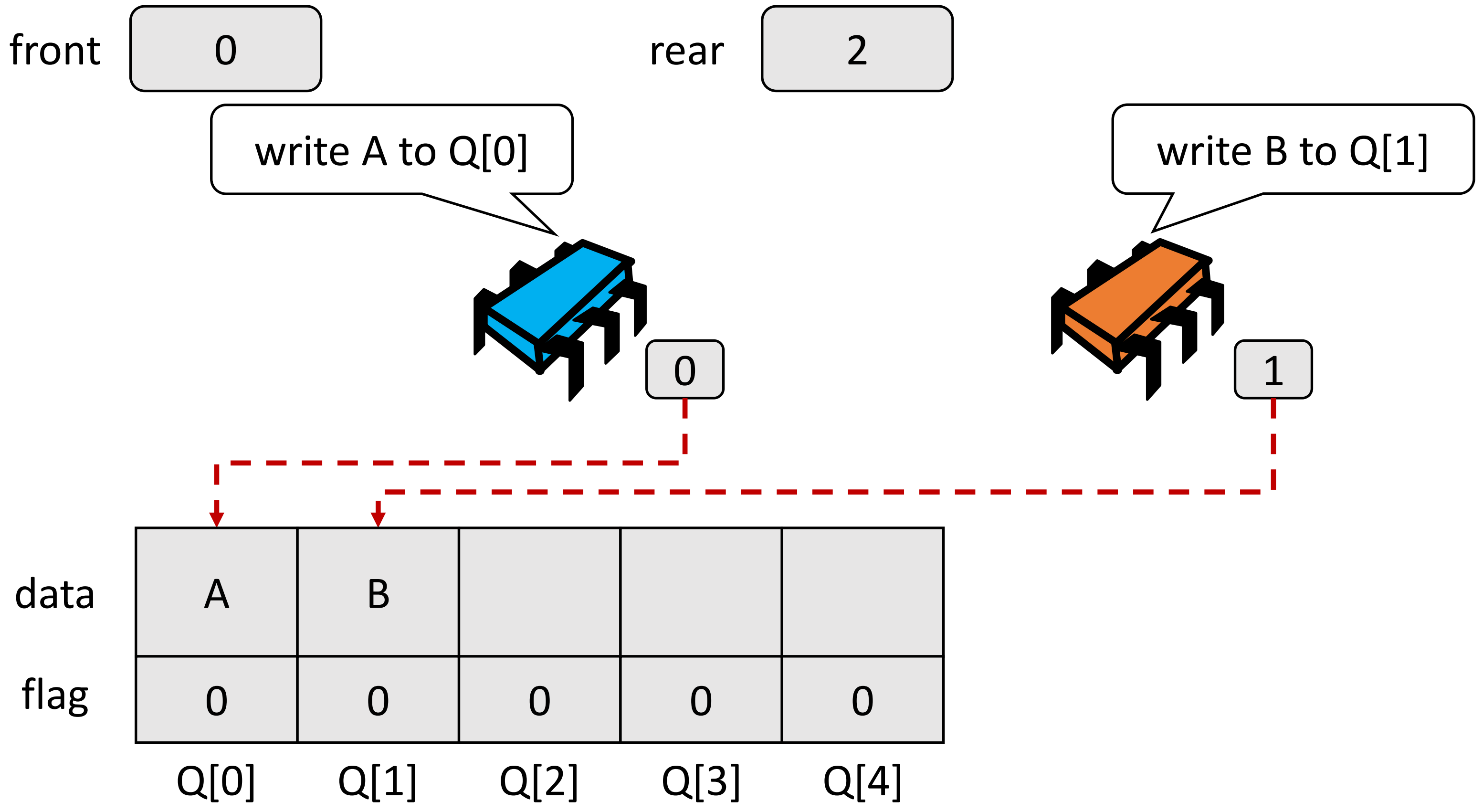
Enqueue


front
0
rear
0
fetch-and-add!
fetch-and-add!
data
flag
0
0
0
0
0
Q[0]
Q[1]
Q[2]
Q[3]
Q[4]

Enqueue

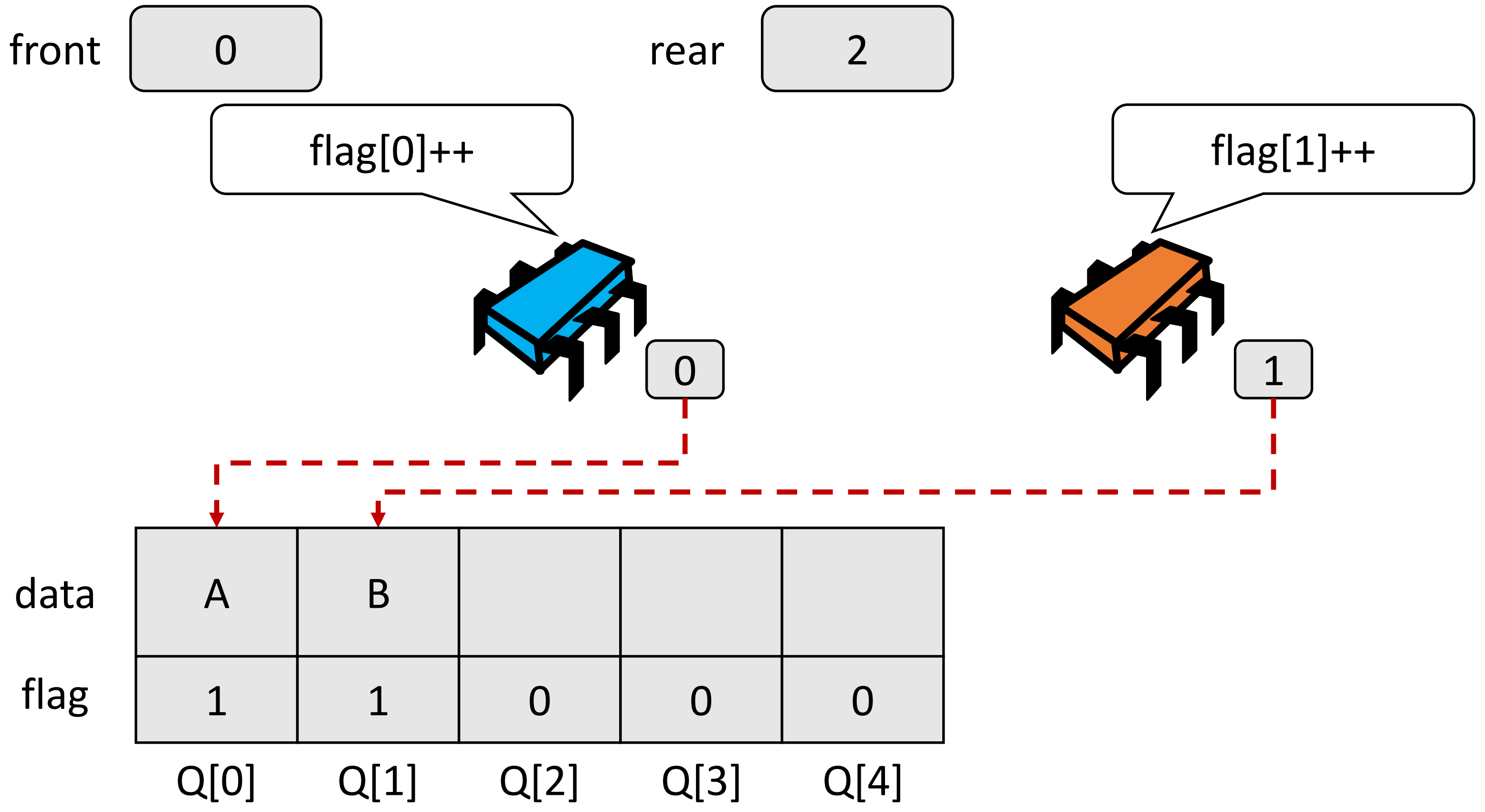

front
0
rear
2
fetch-and-add!
fetch-and-add!
0
1
data
flag
0
0
0
0
0
Q[0]
Q[1]
Q[2]
Q[3]
Q[4]

Enqueue

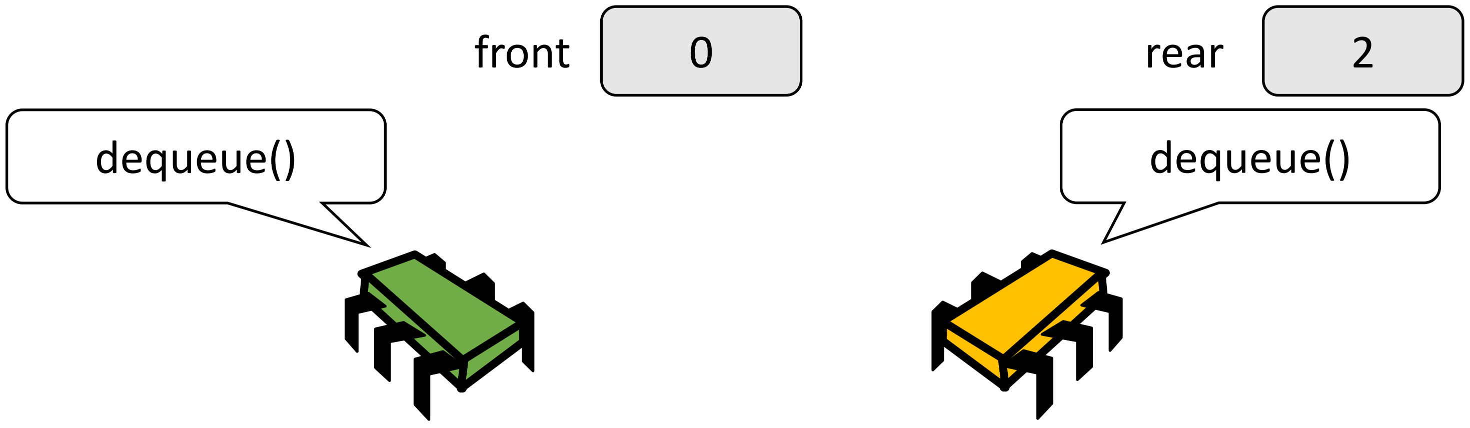


front
0
rear
2
write A to Q[0]
write B to Q[1]
0
1
data
A
B
flag
0
0
0
0
0
Q[0]
Q[1]
Q[2]
Q[3]
Q[4]

Enqueue

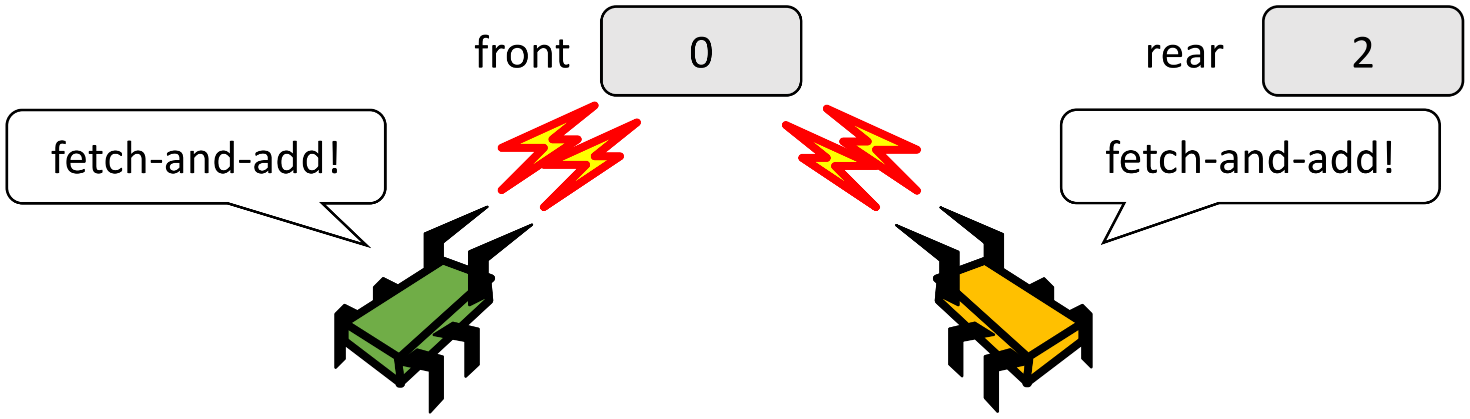


Dequeue



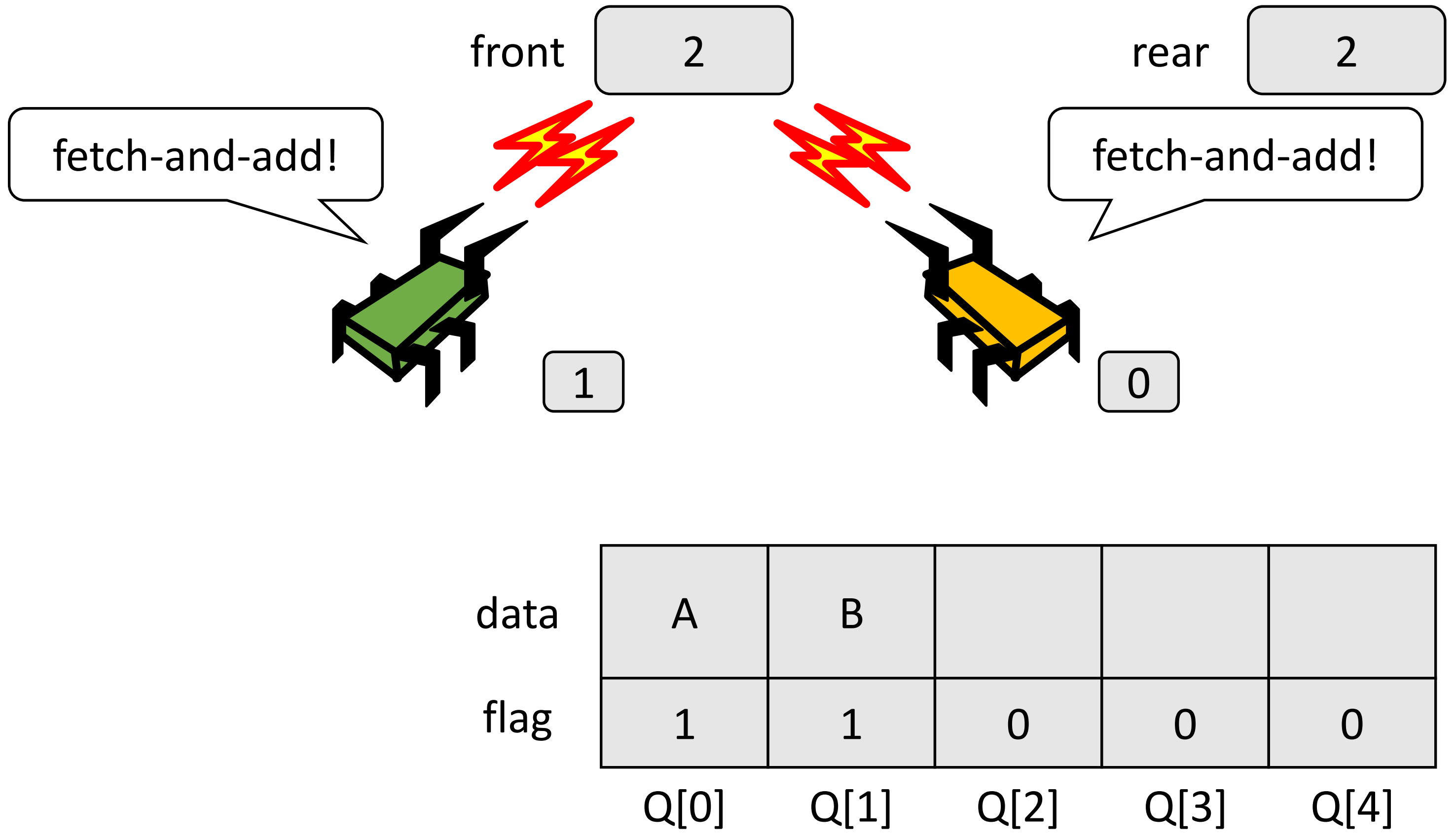
	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B			
flag	1	1	0	0	0

Dequeue

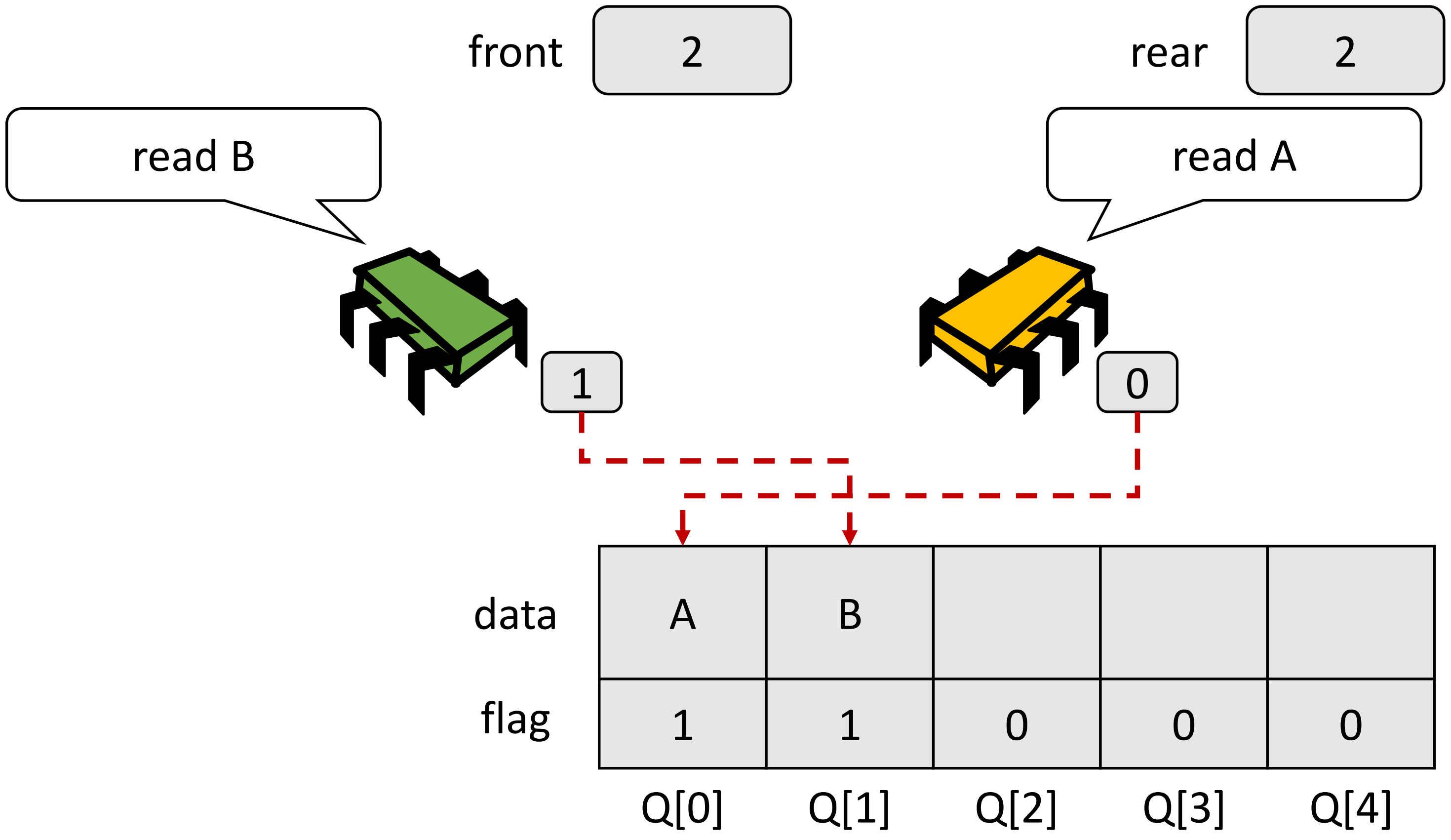


	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B			
flag	1	1	0	0	0

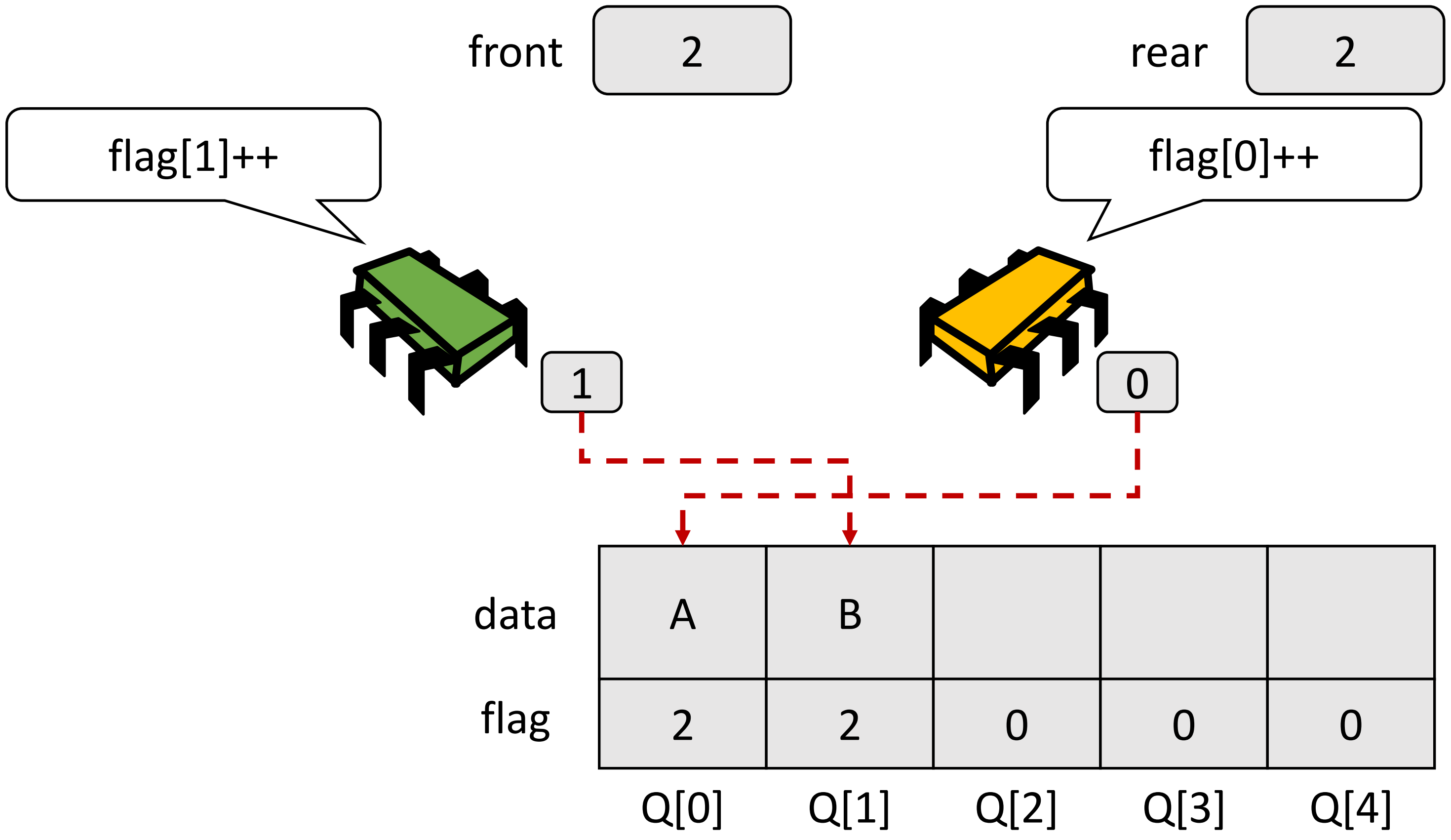
Dequeue


front
2
rear
2
fetch-and-add!
fetch-and-add!
1
0
data
A
B
flag
1
1
0
0
0
Q[0]
Q[1]
Q[2]
Q[3]
Q[4]

Dequeue

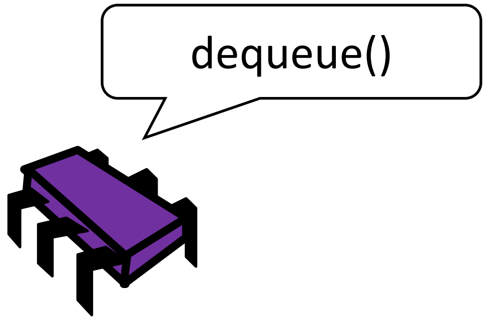


Dequeue



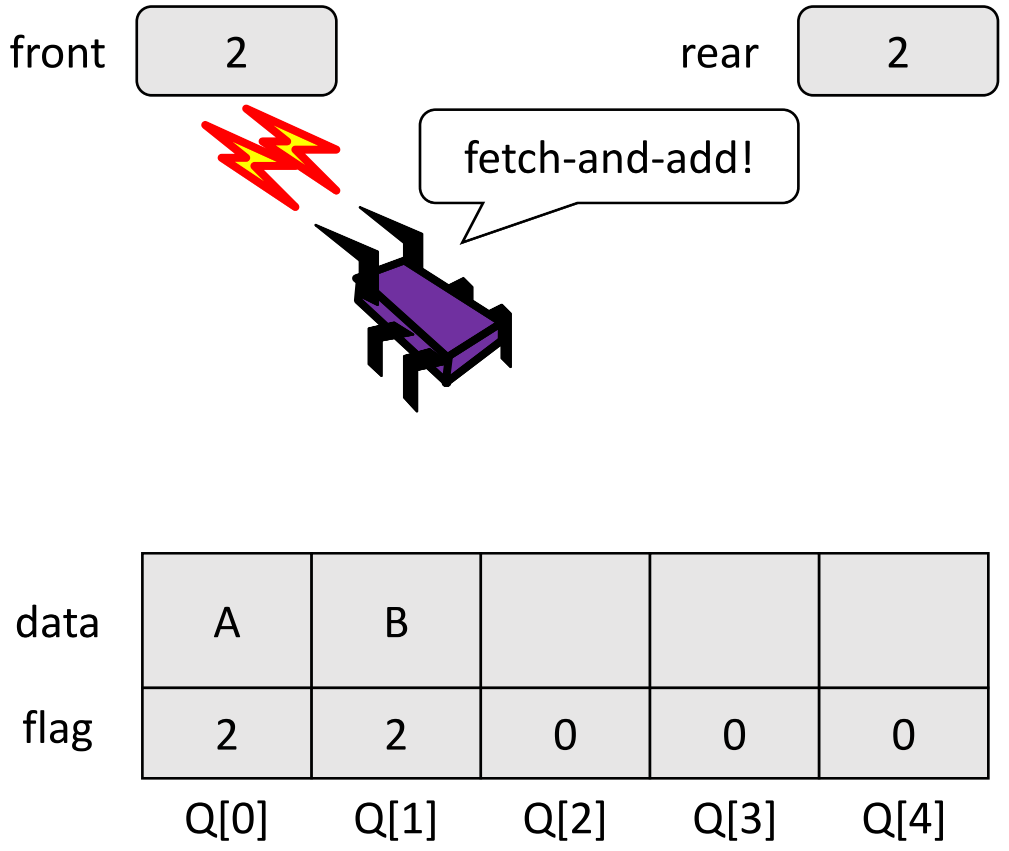
Dequeue

front 2   rear 2

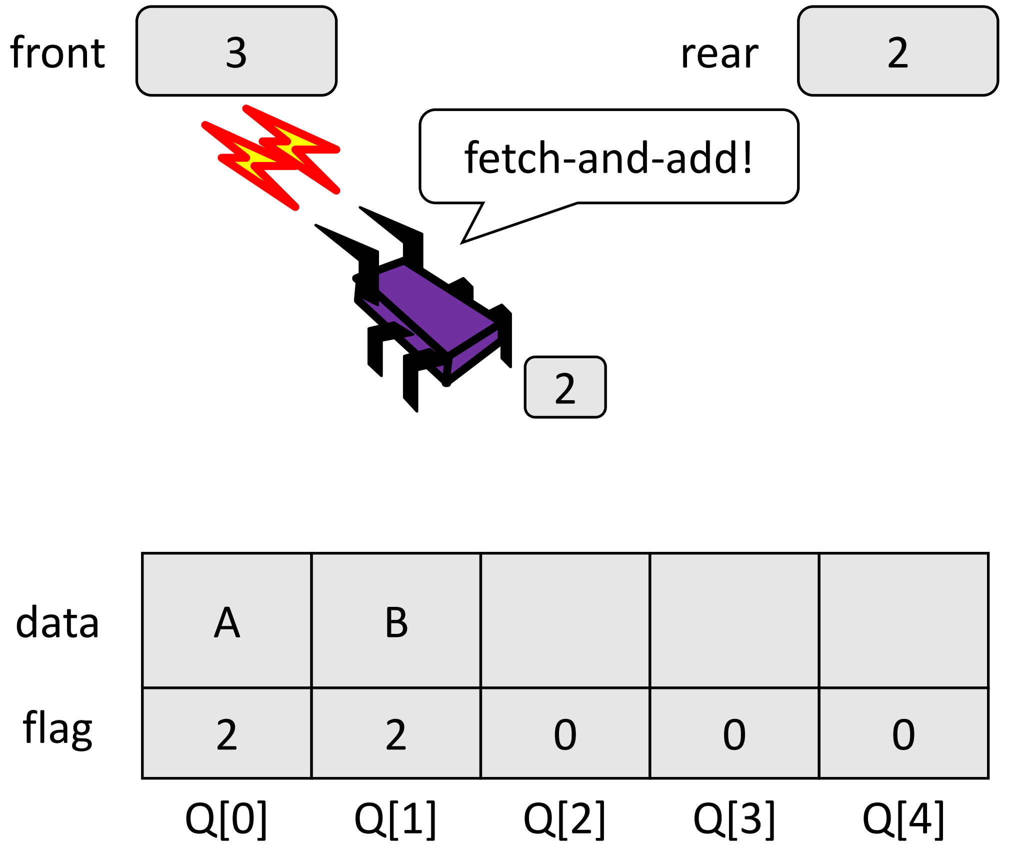


	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B			
flag	2	2	0	0	0

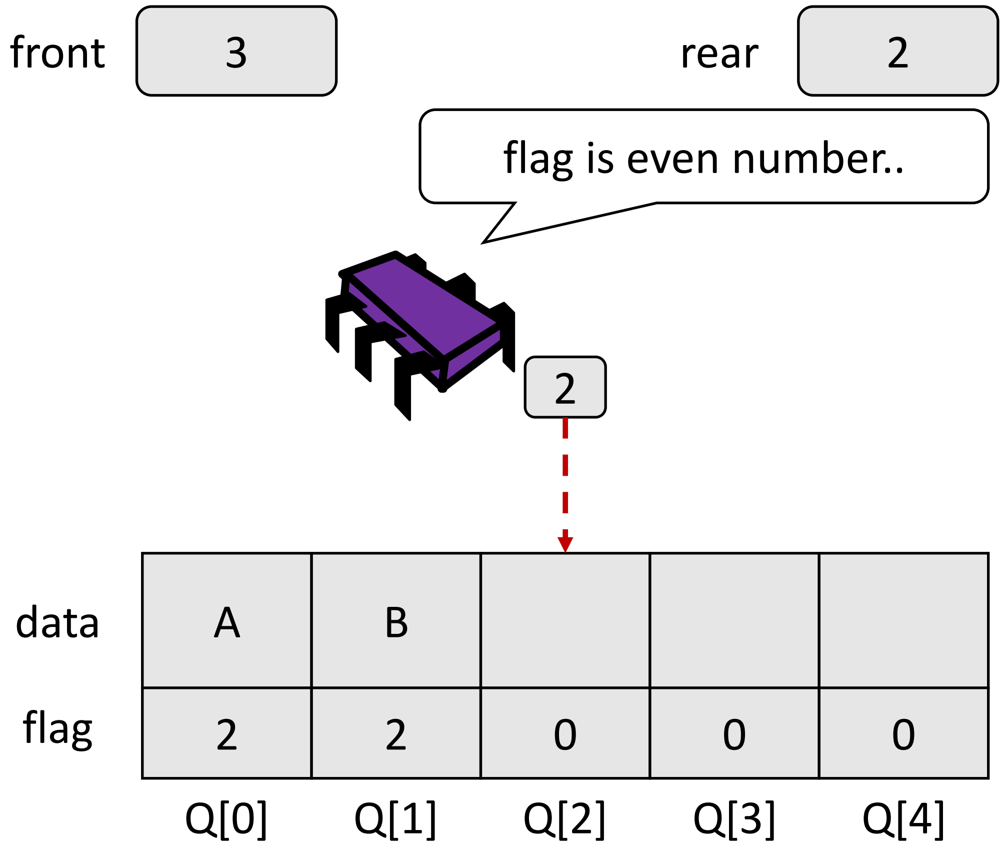
Dequeue



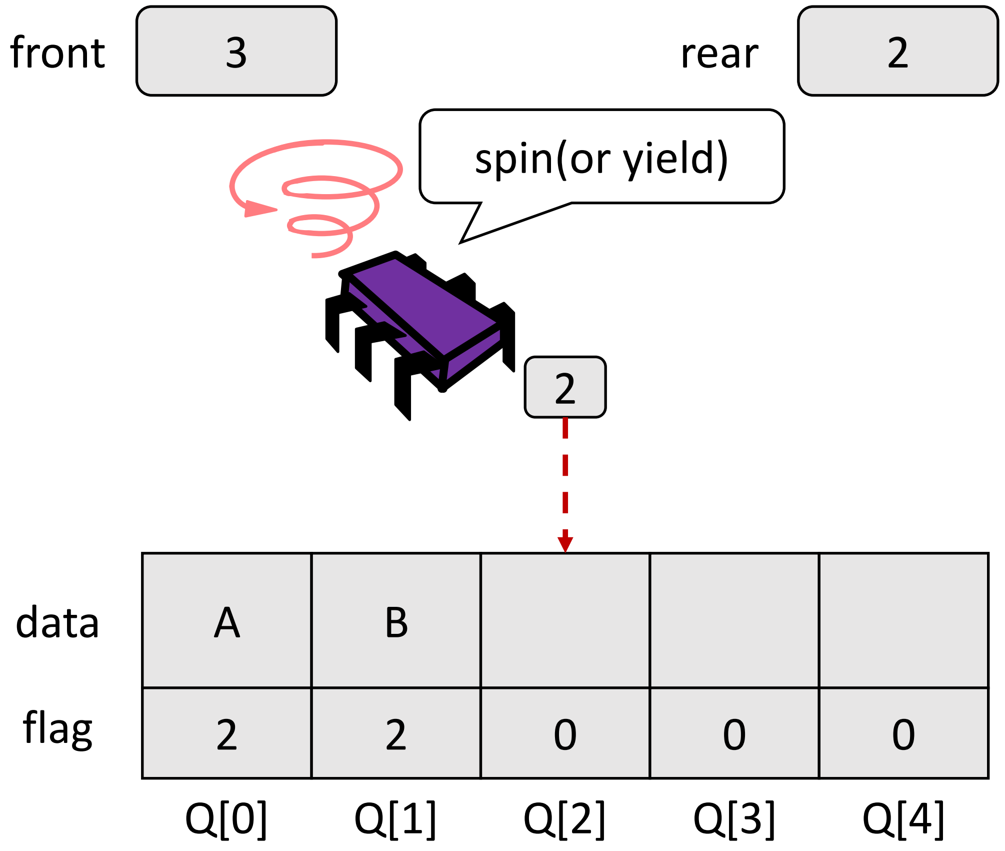
Dequeue



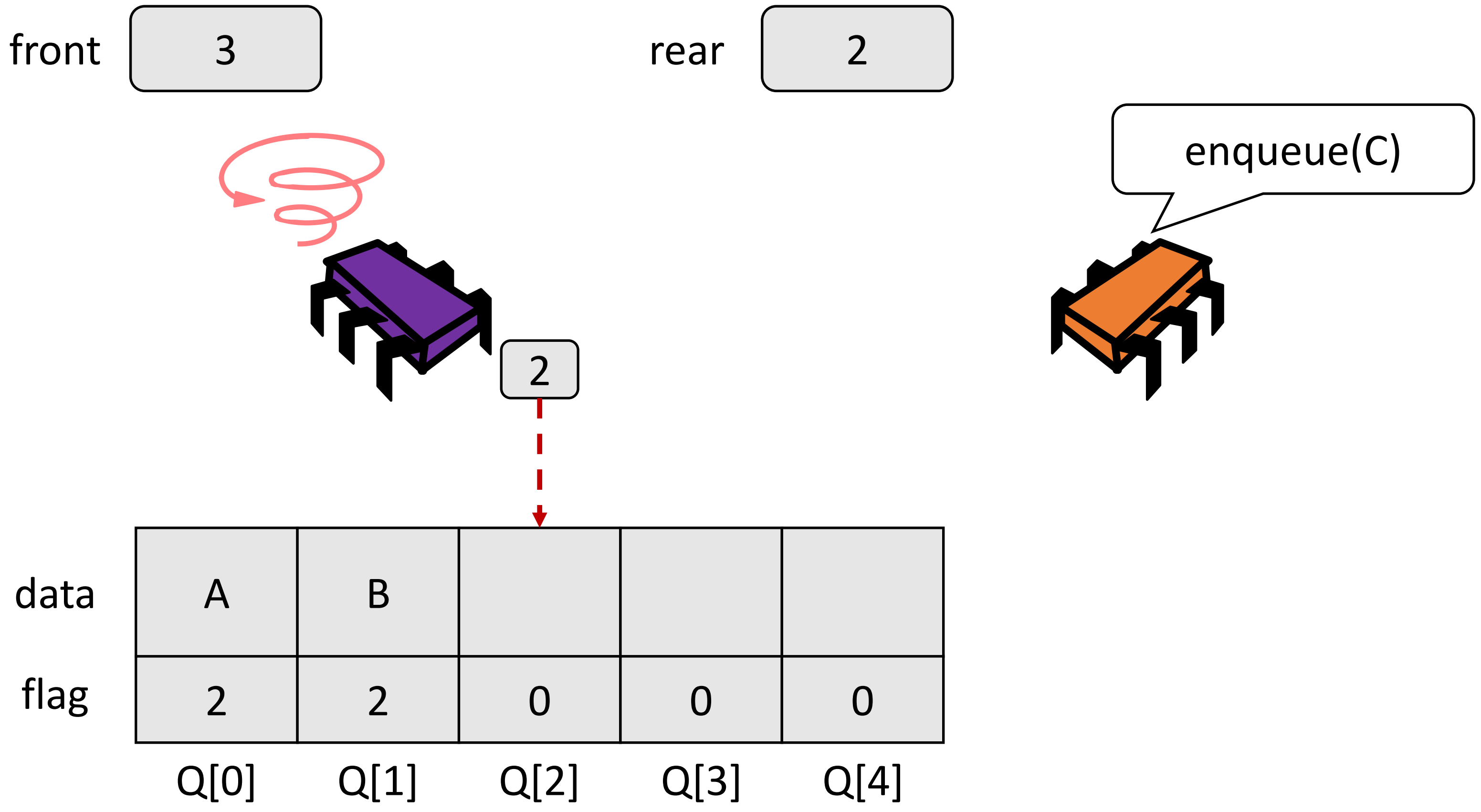
Dequeue


front
3
rear
2
flag is even number..
2
data
A
B
flag
2
2
0
0
0
Q[0]
Q[1]
Q[2]
Q[3]
Q[4]

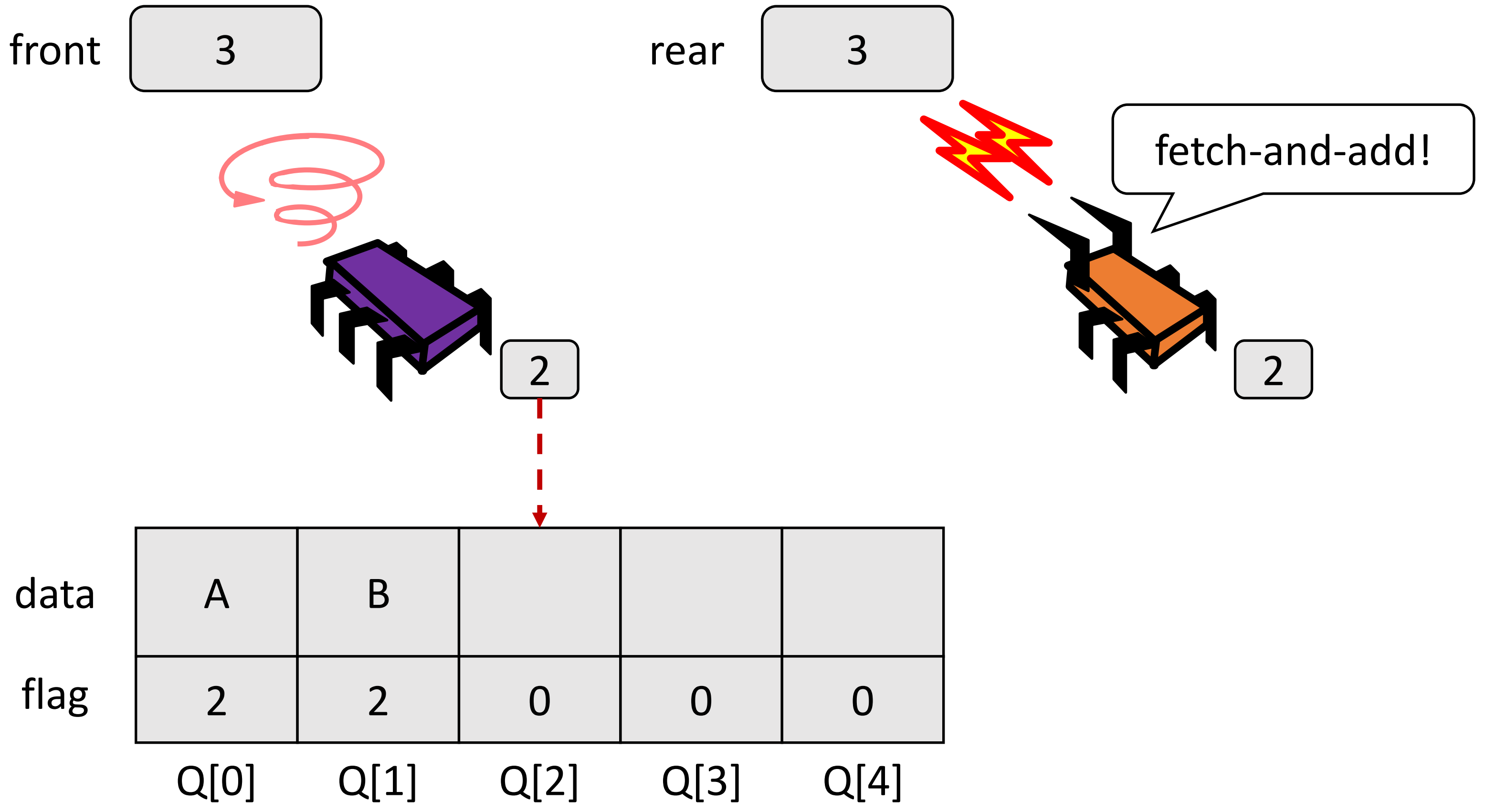
Dequeue



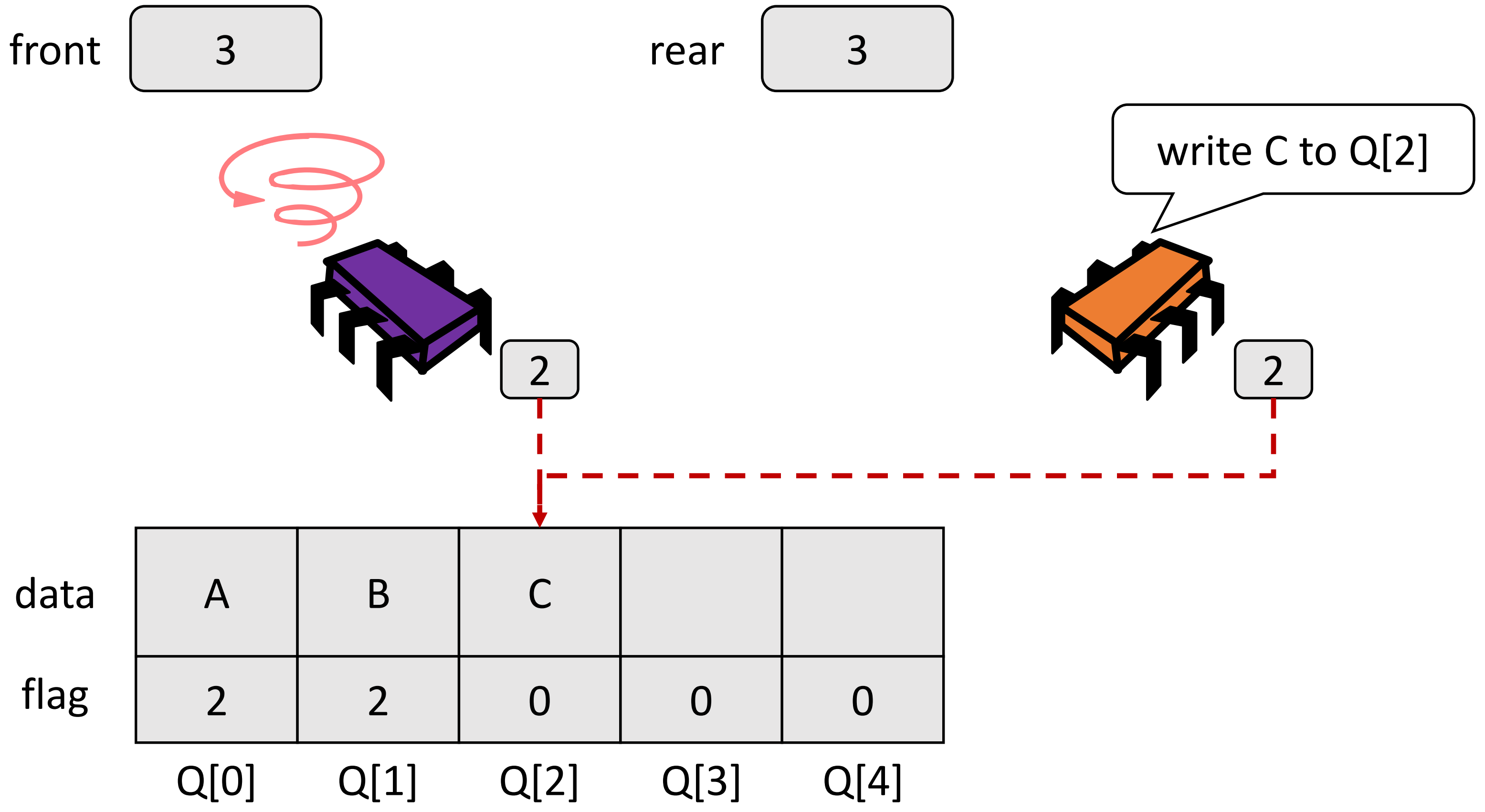
Dequeue



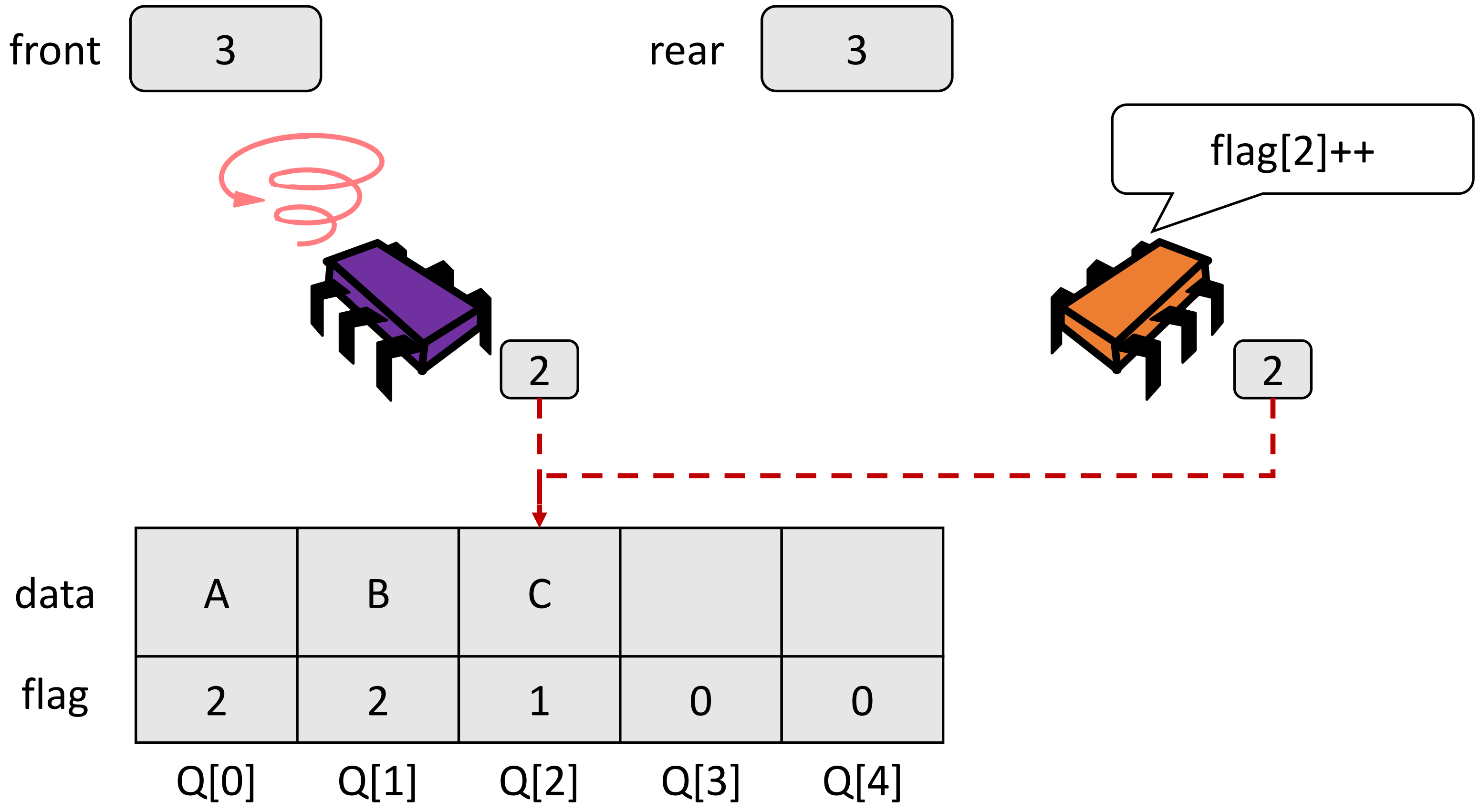
Dequeue



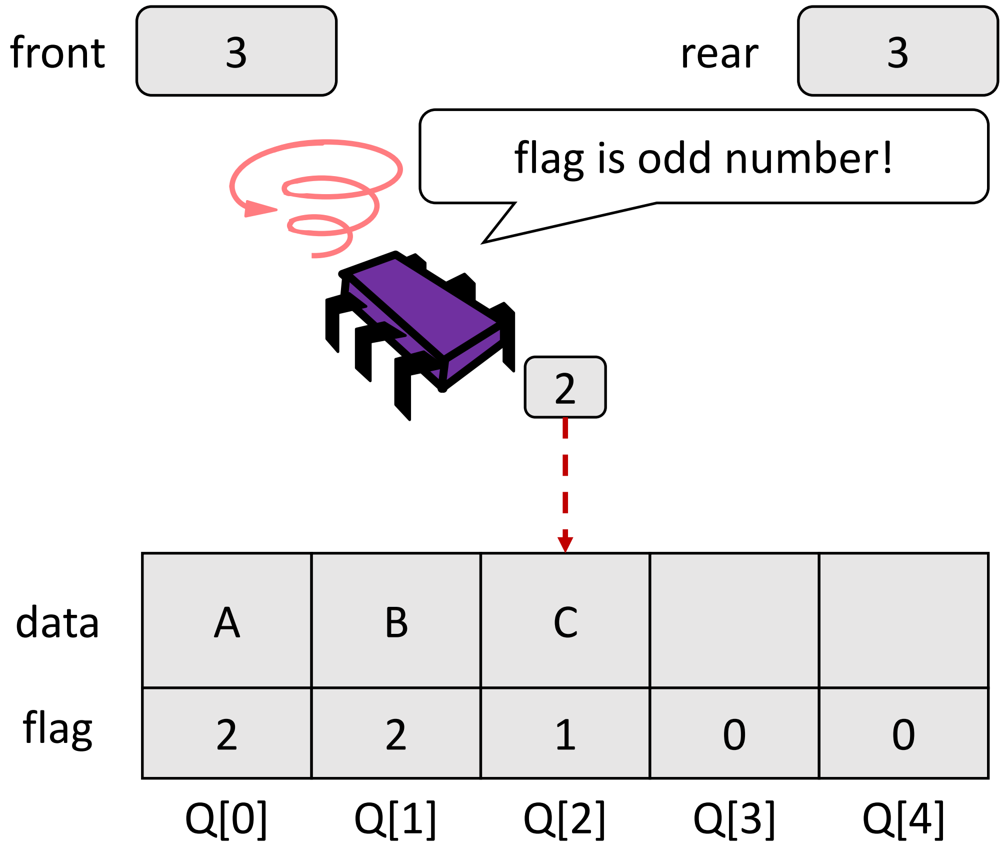
Dequeue



Dequeue

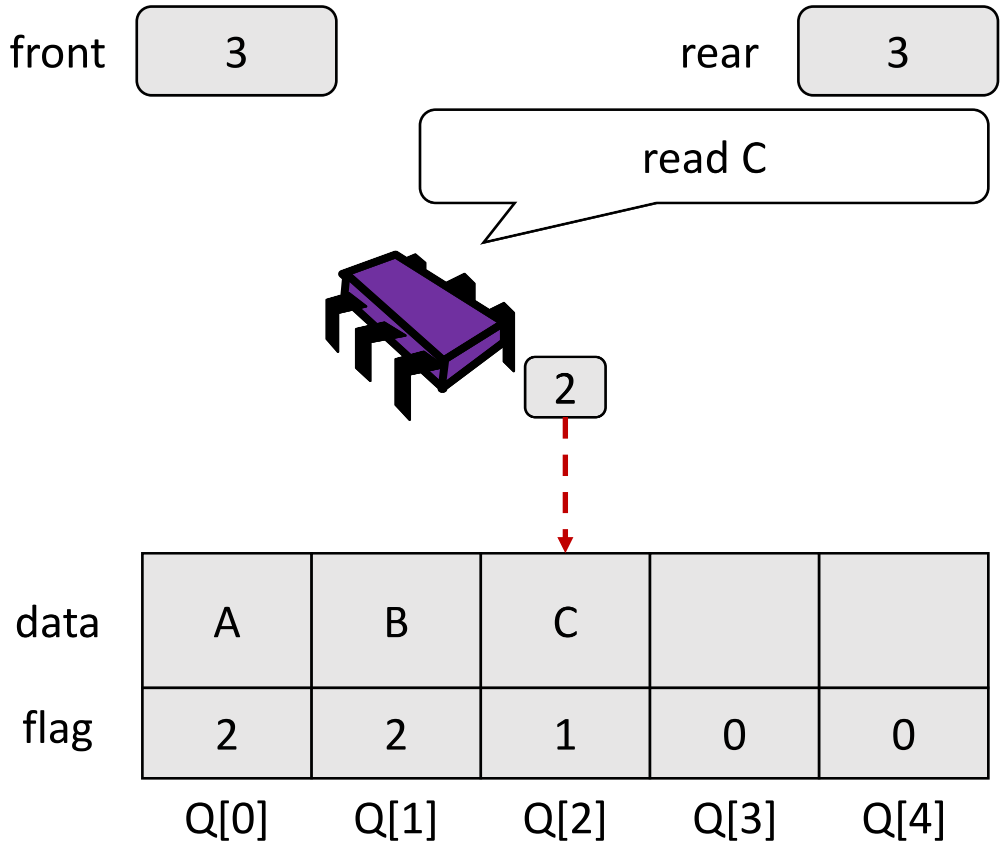


Dequeue

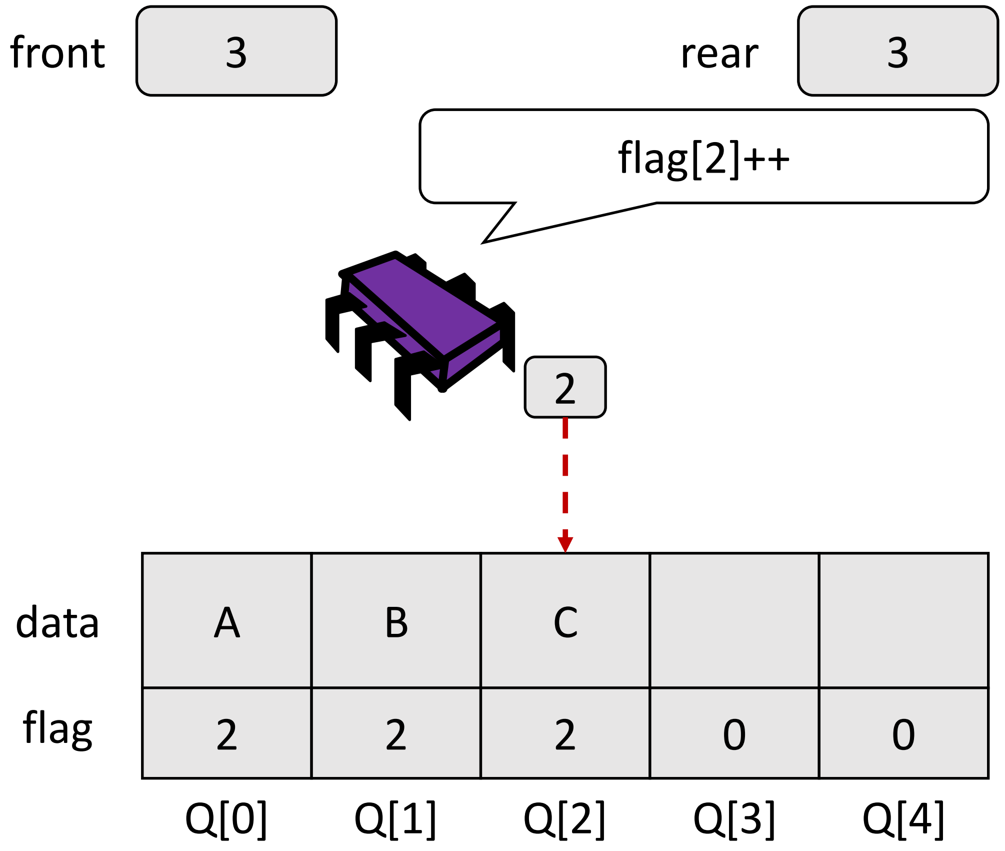


	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B	C		
flag	2	2	1	0	0

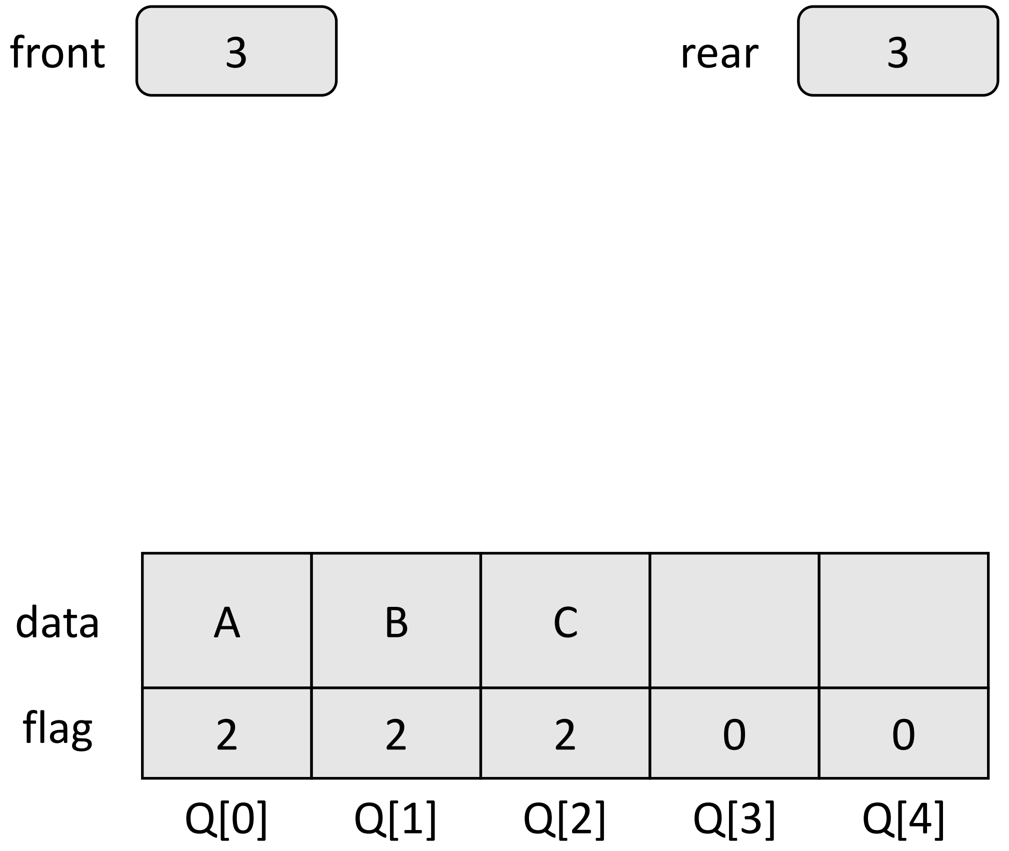
Dequeue



Dequeue



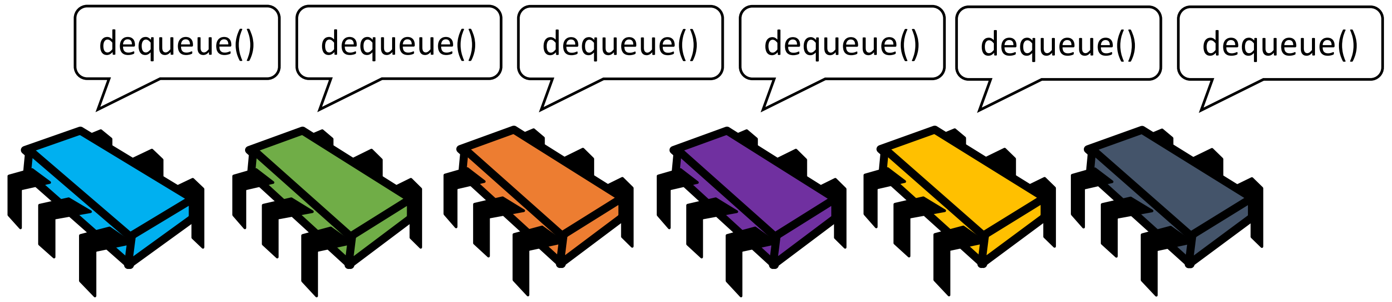
Dequeue



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

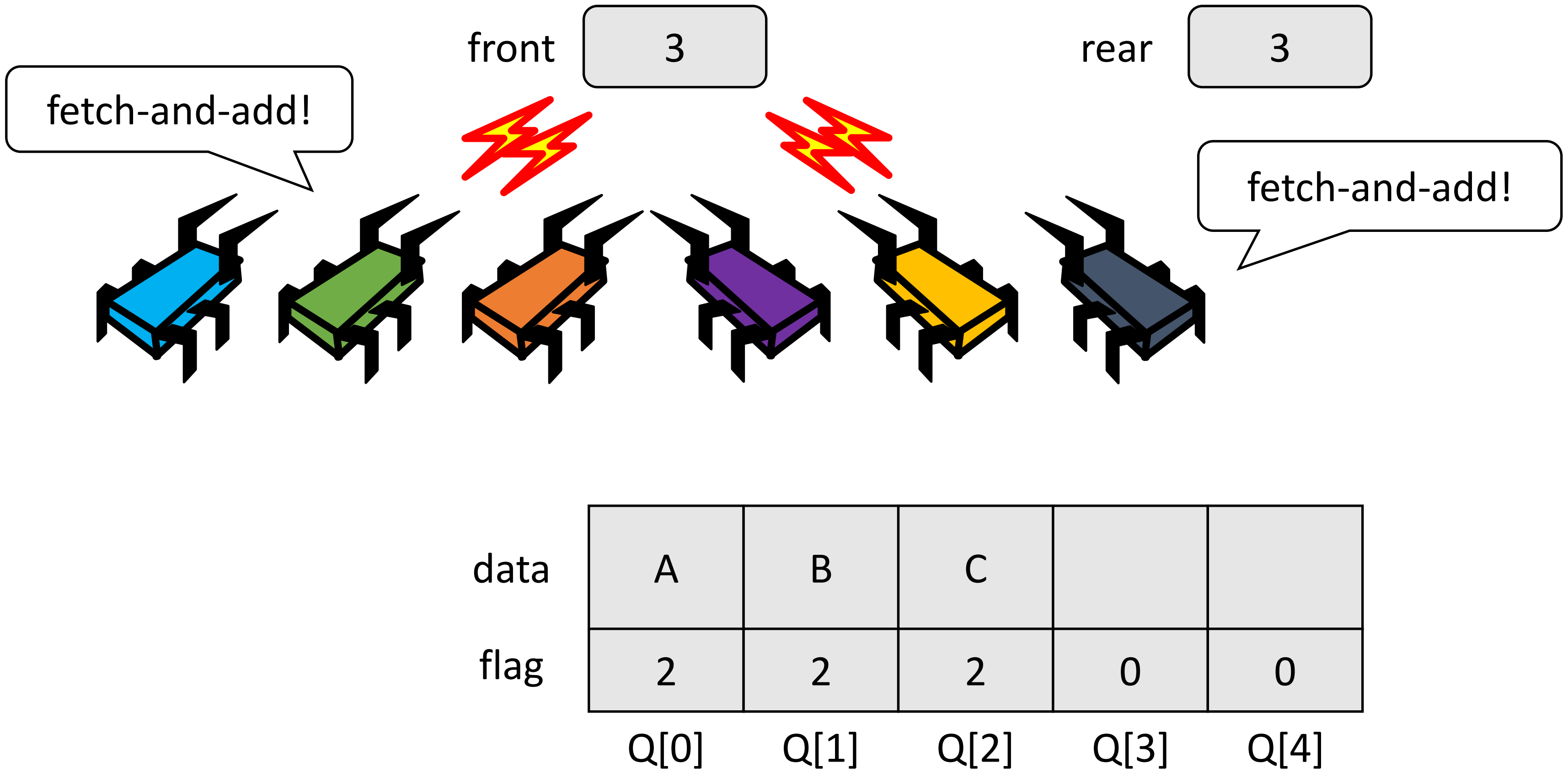
Dequeue

front 3 rear 3

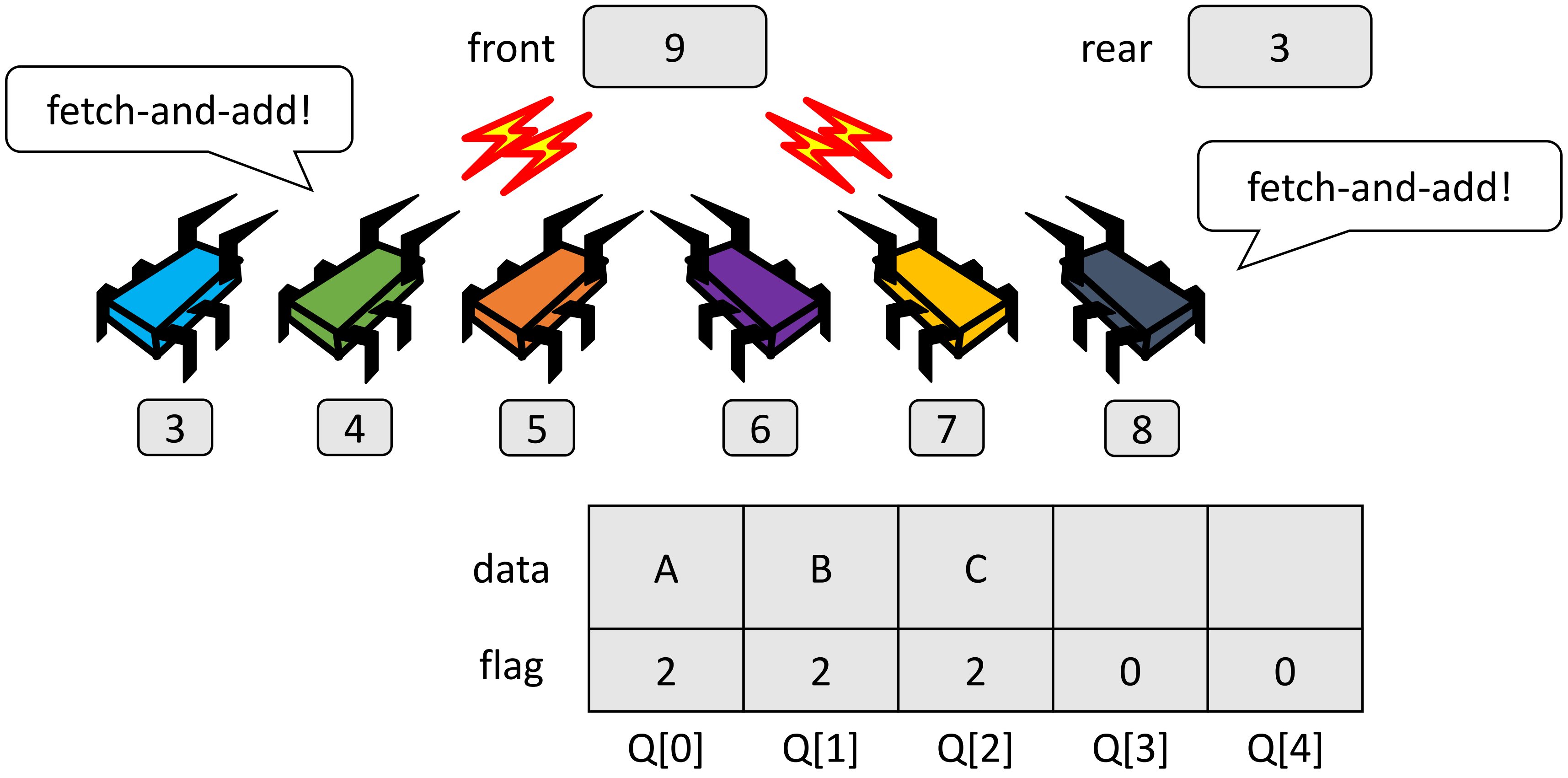


	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B	C		
flag	2	2	2	0	0

Dequeue

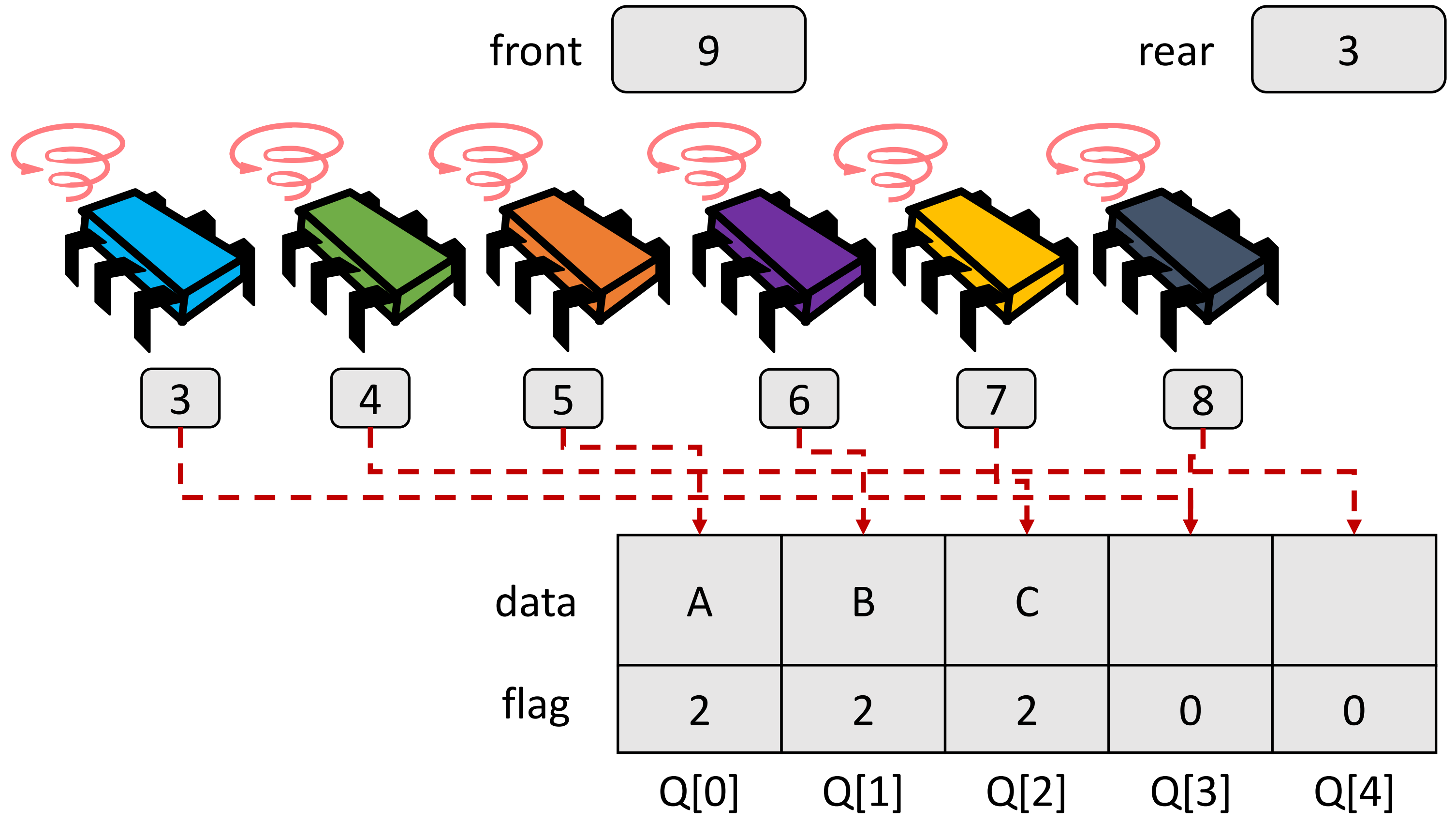

front
3
rear
3
fetch-and-add!
fetch-and-add!
data
flag
A
B
C
2
2
2
0
0
Q[0]
Q[1]
Q[2]
Q[3]
Q[4]

Dequeue

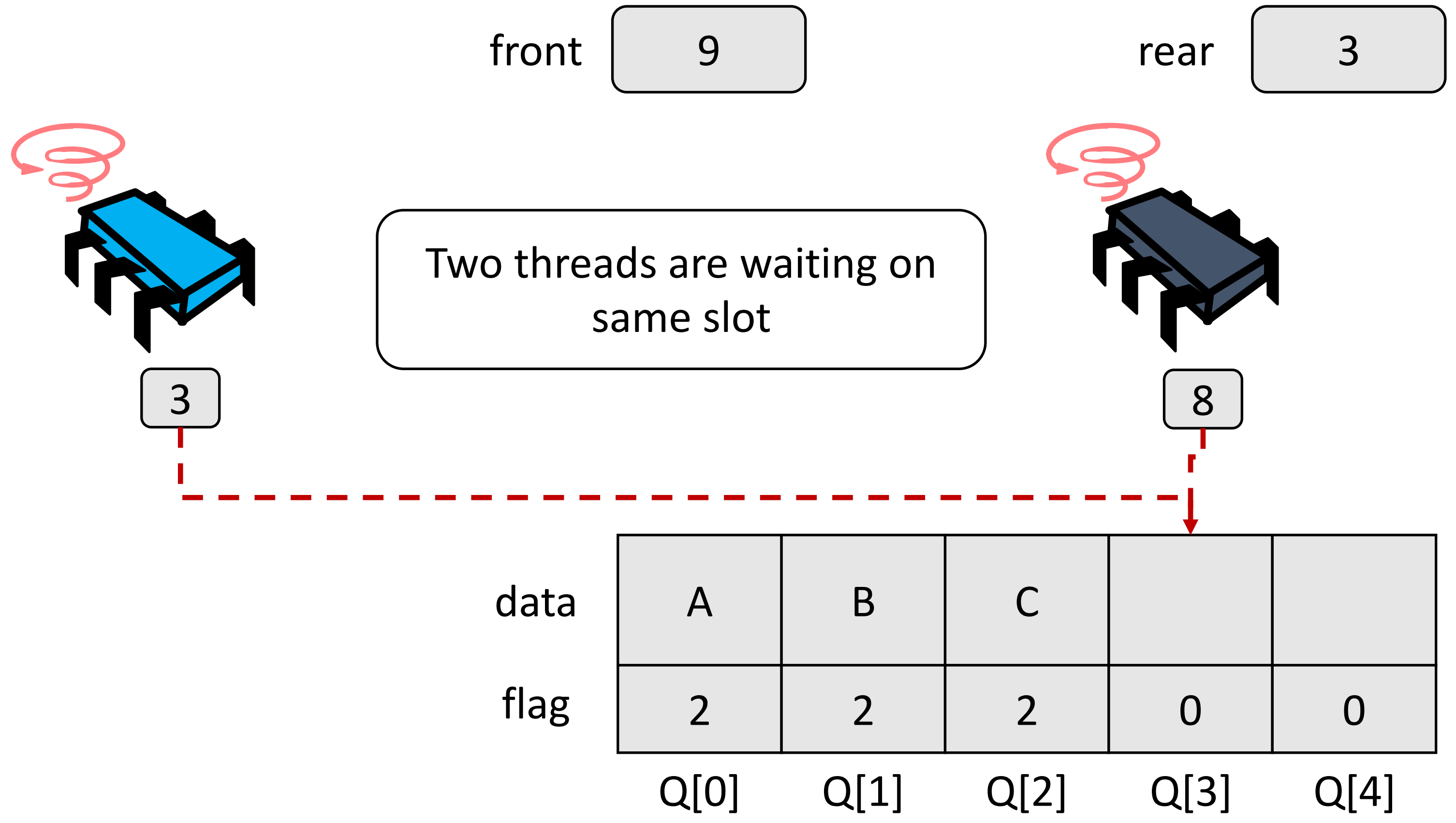


	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
data	A	B	C		
flag	2	2	2	0	0

Dequeue

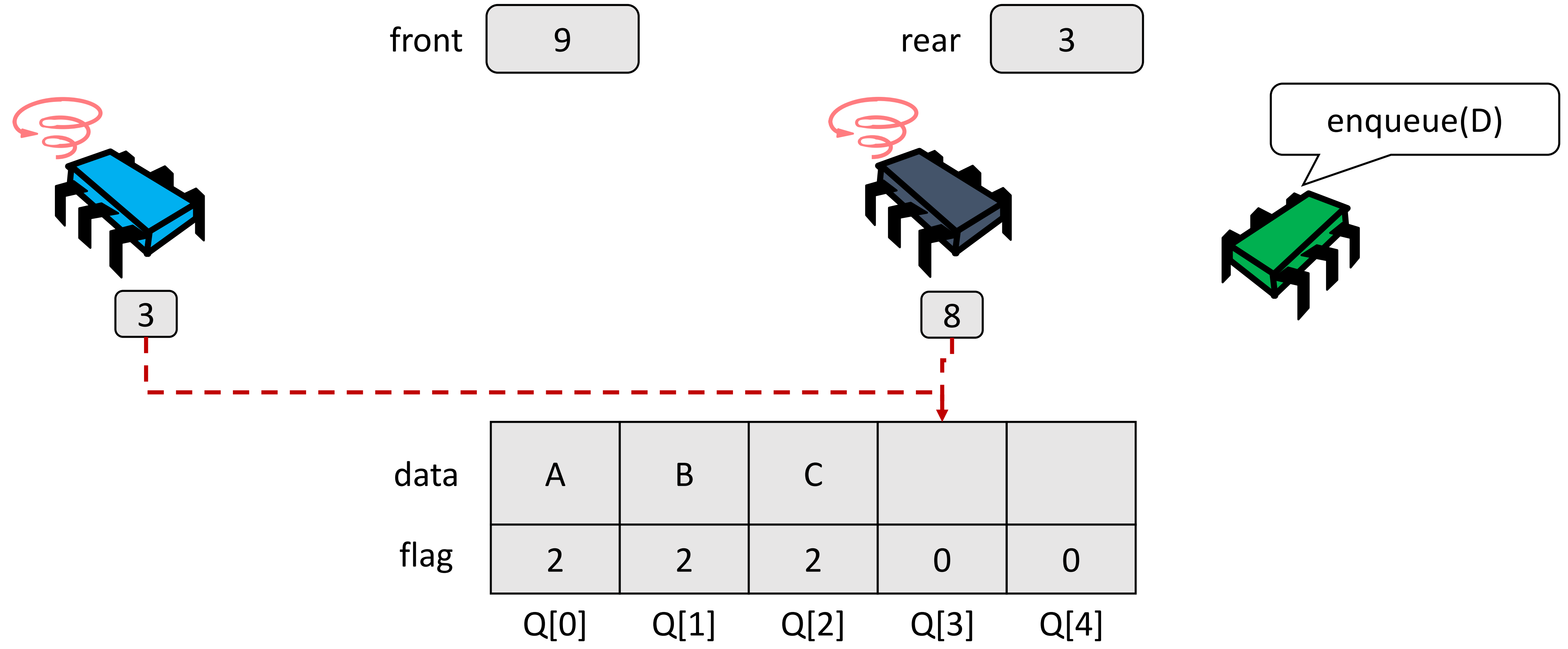


Dequeue

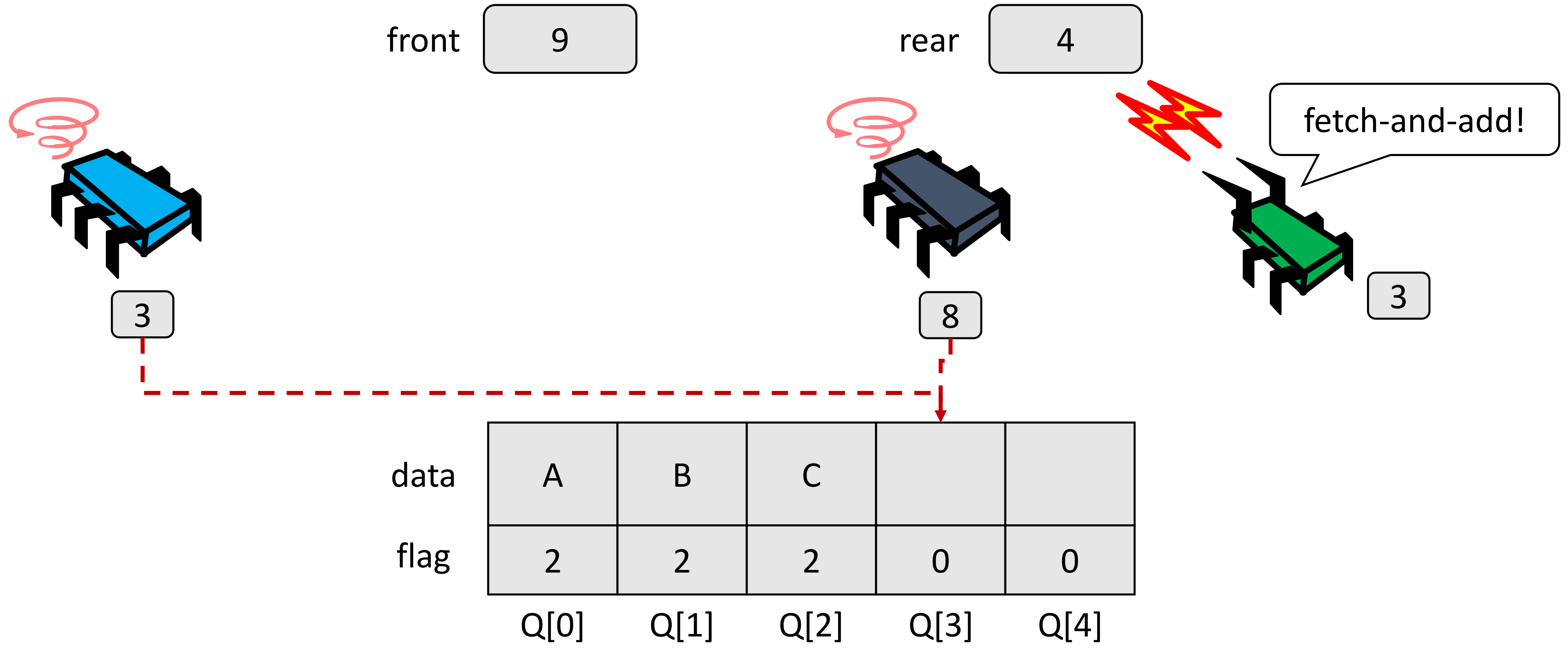


front
9
rear
3
Two threads are waiting on same slot
3
8
data
A
B
C
flag
2
2
2
0
0
Q[0]
Q[1]
Q[2]
Q[3]
Q[4]

Dequeue

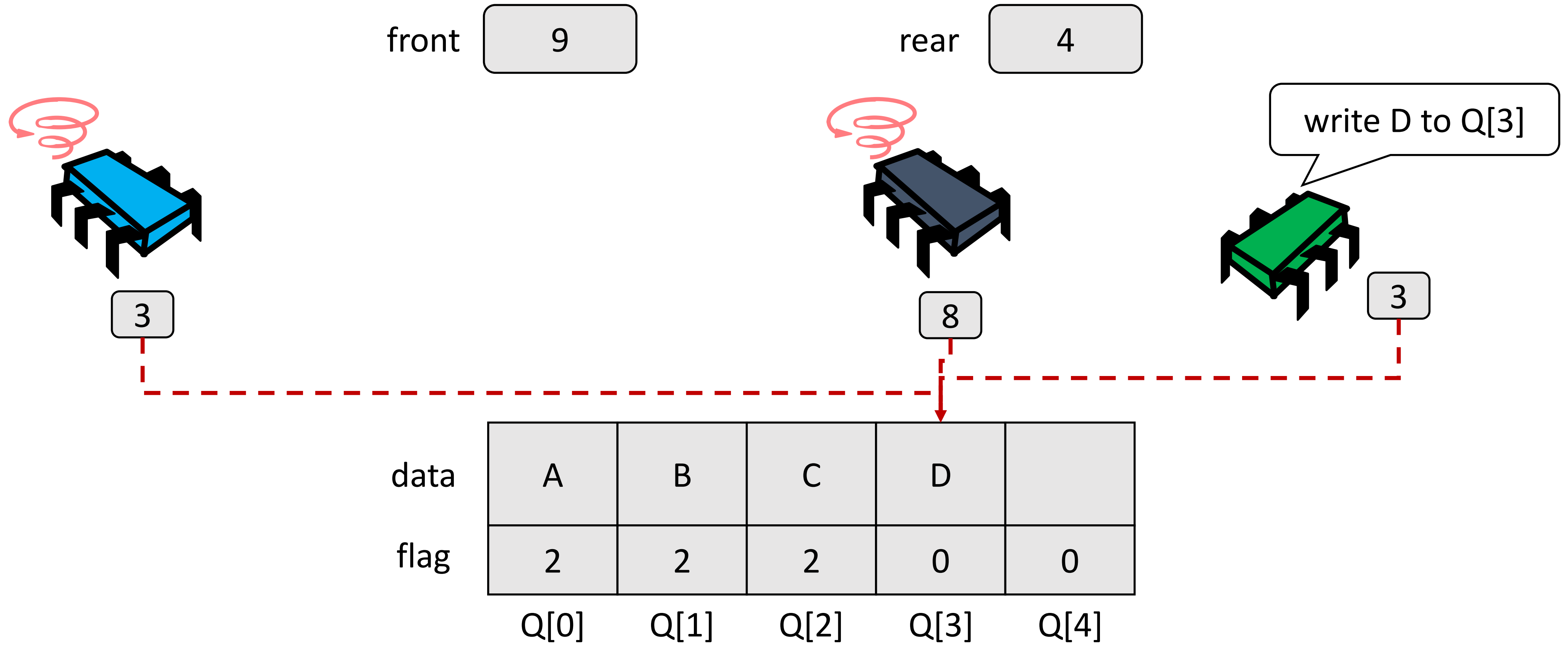


Dequeue



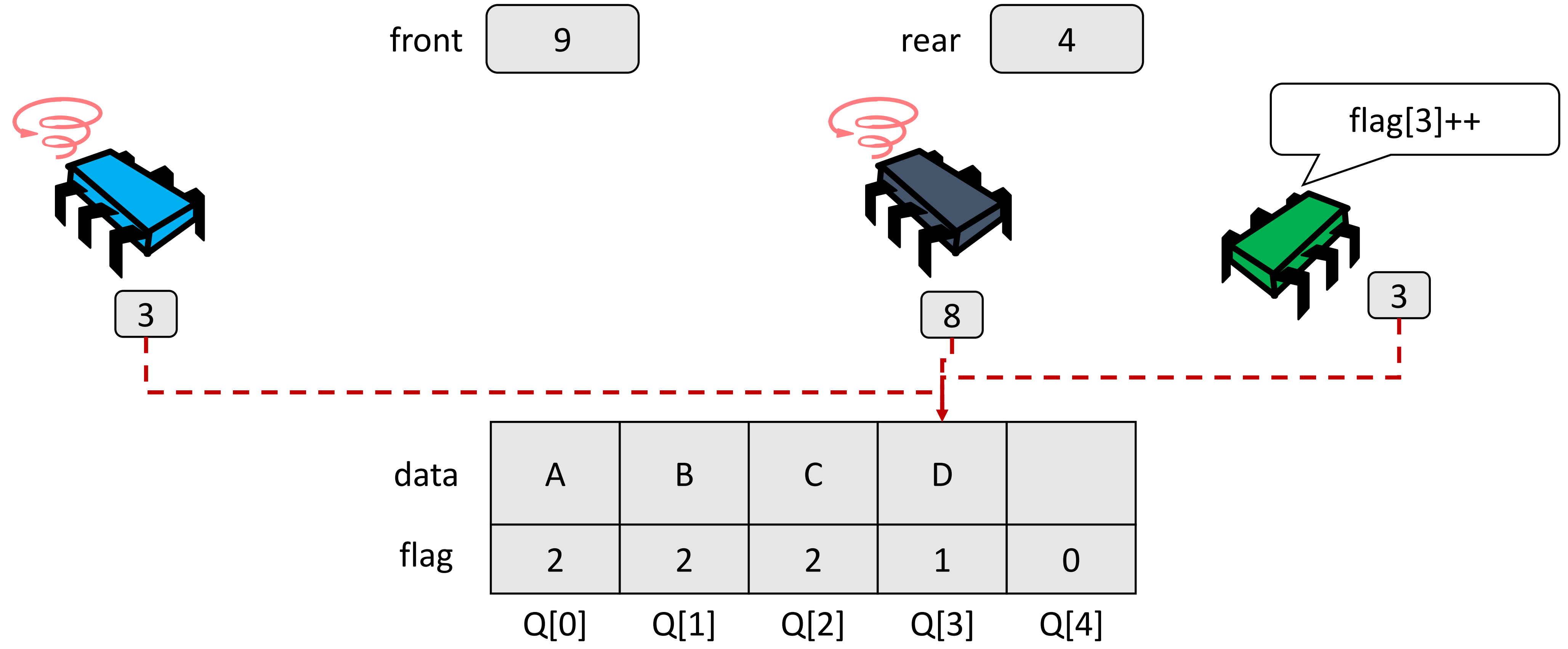
front
9
rear
4
fetch-and-add!
3
8
3
data
flag
A
B
C
2
2
2
0
0
Q[0]
Q[1]
Q[2]
Q[3]
Q[4]

Dequeue

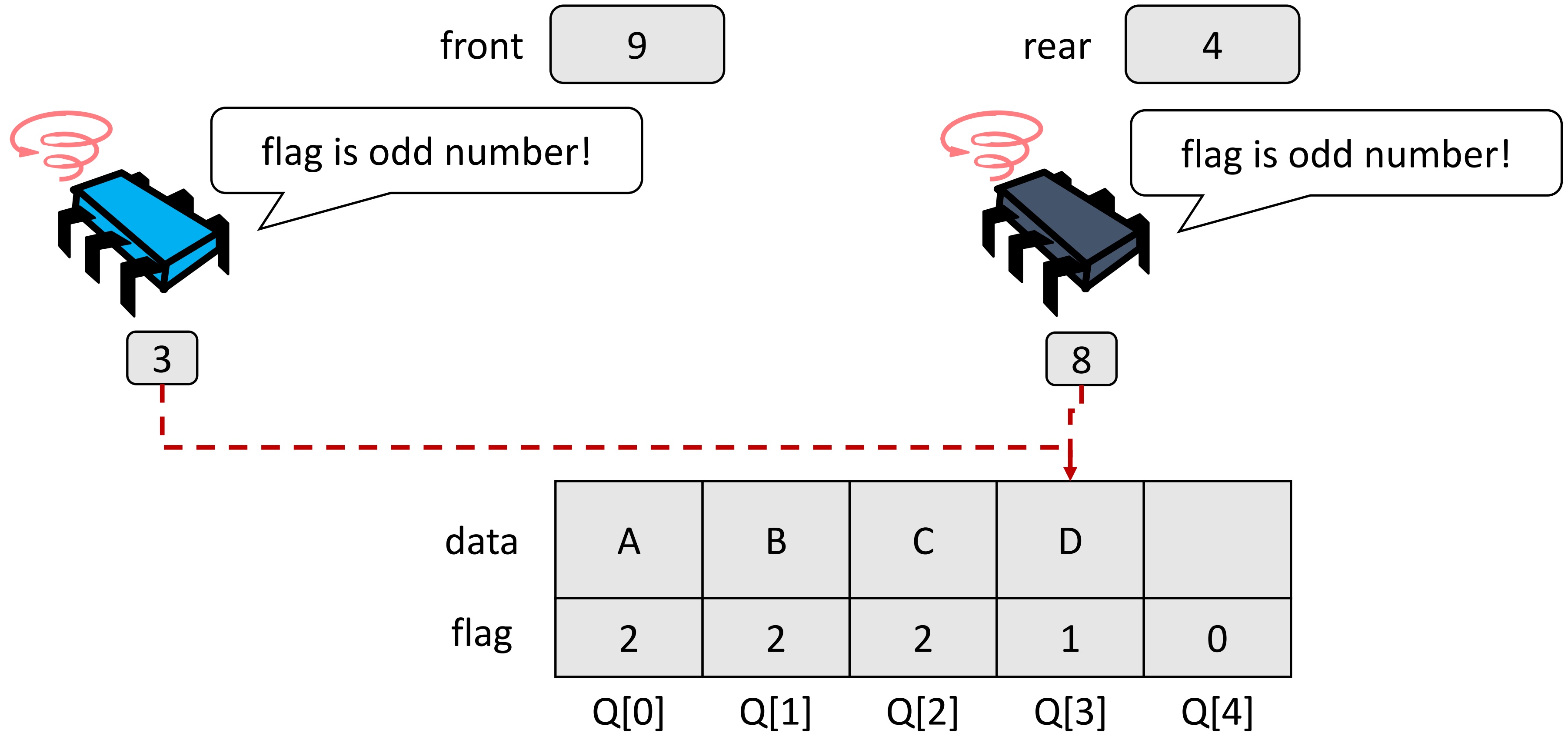


front
9
rear
4
write D to Q[3]
3
8
3
data
flag
A
B
C
D
2
2
2
0
0
Q[0]
Q[1]
Q[2]
Q[3]
Q[4]

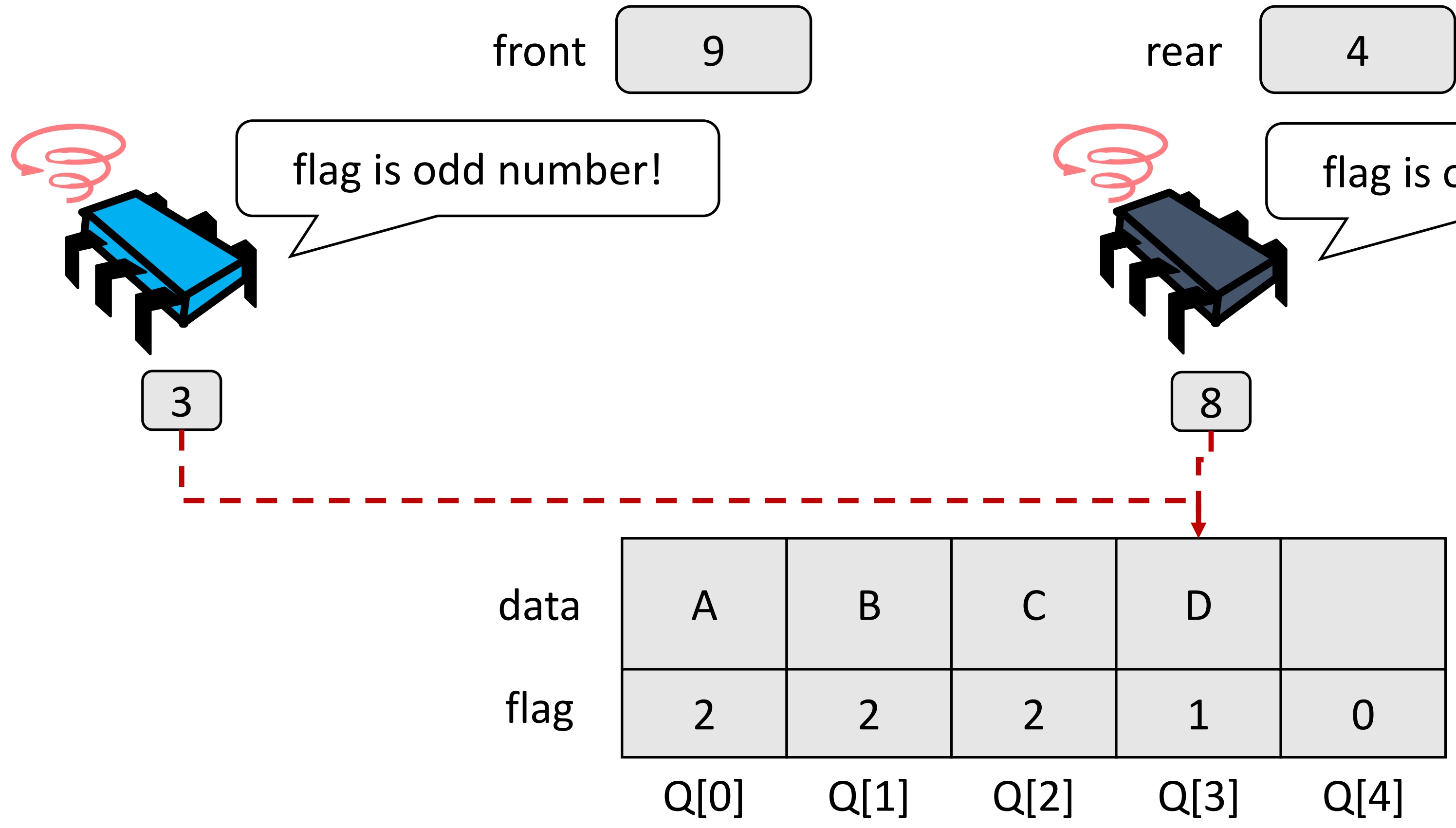
Dequeue



Dequeue


front
9
rear
4
flag is odd number!
flag is odd number!
3
8
data
flag
A
B
C
D
2
2
2
1
0
Q[0]
Q[1]
Q[2]
Q[3]
Q[4]

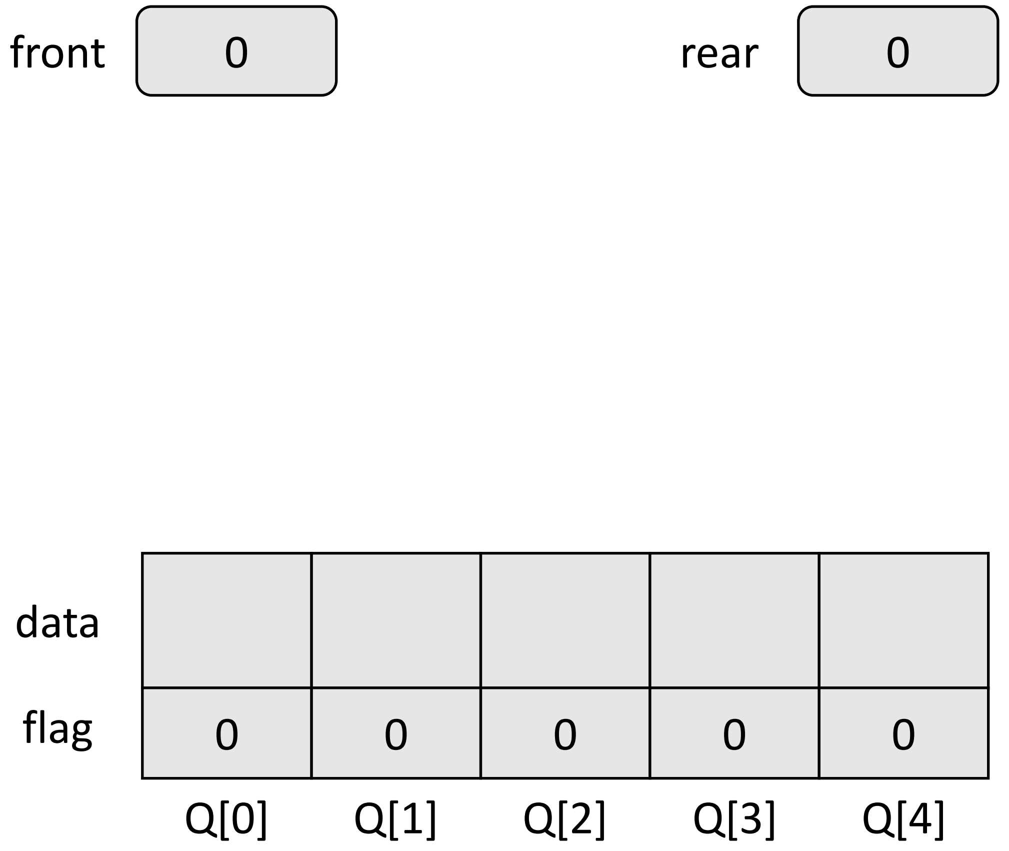
Dequeue



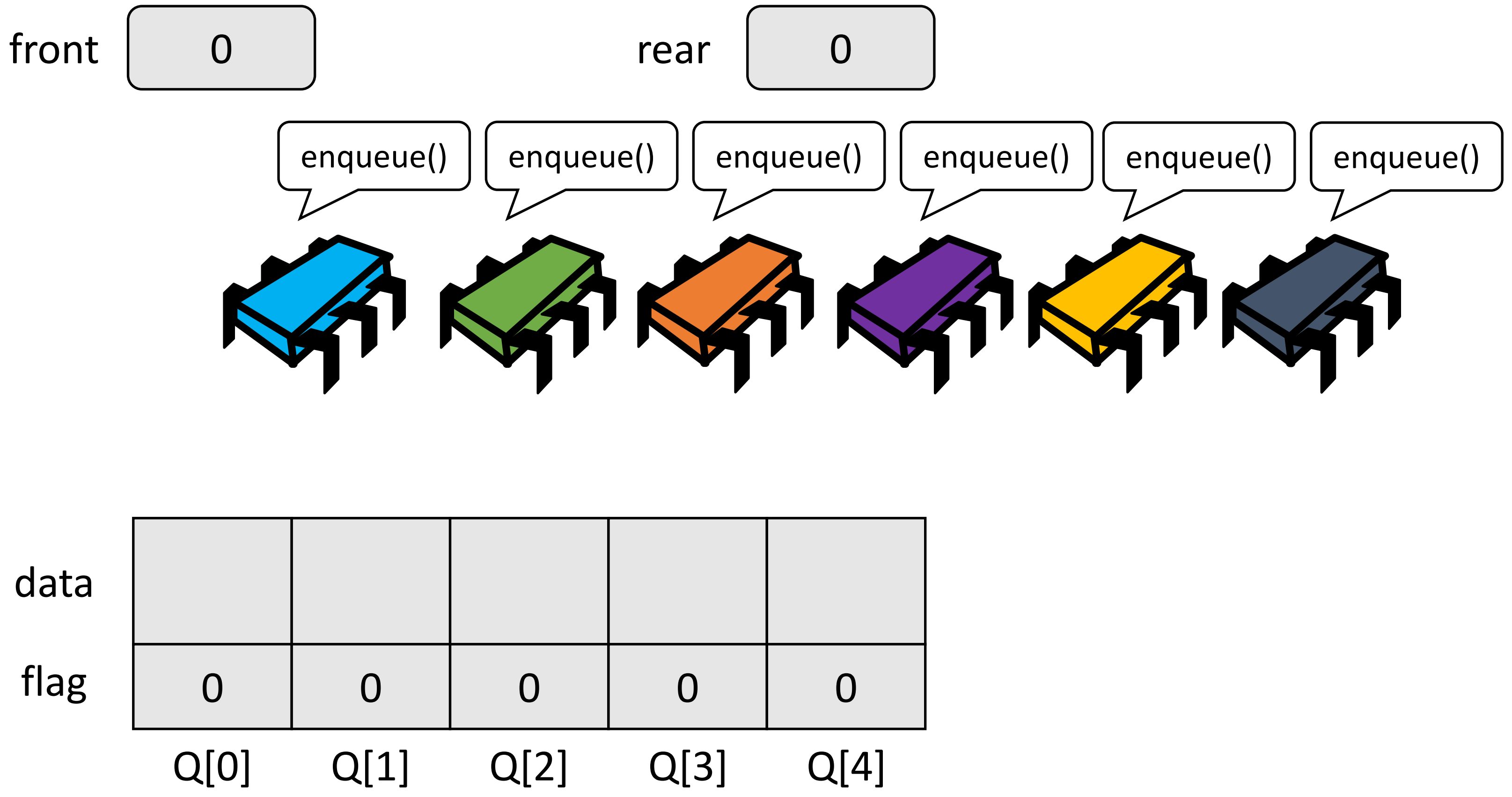
For the fairness, which thread should read the data D?

How can you implement it?

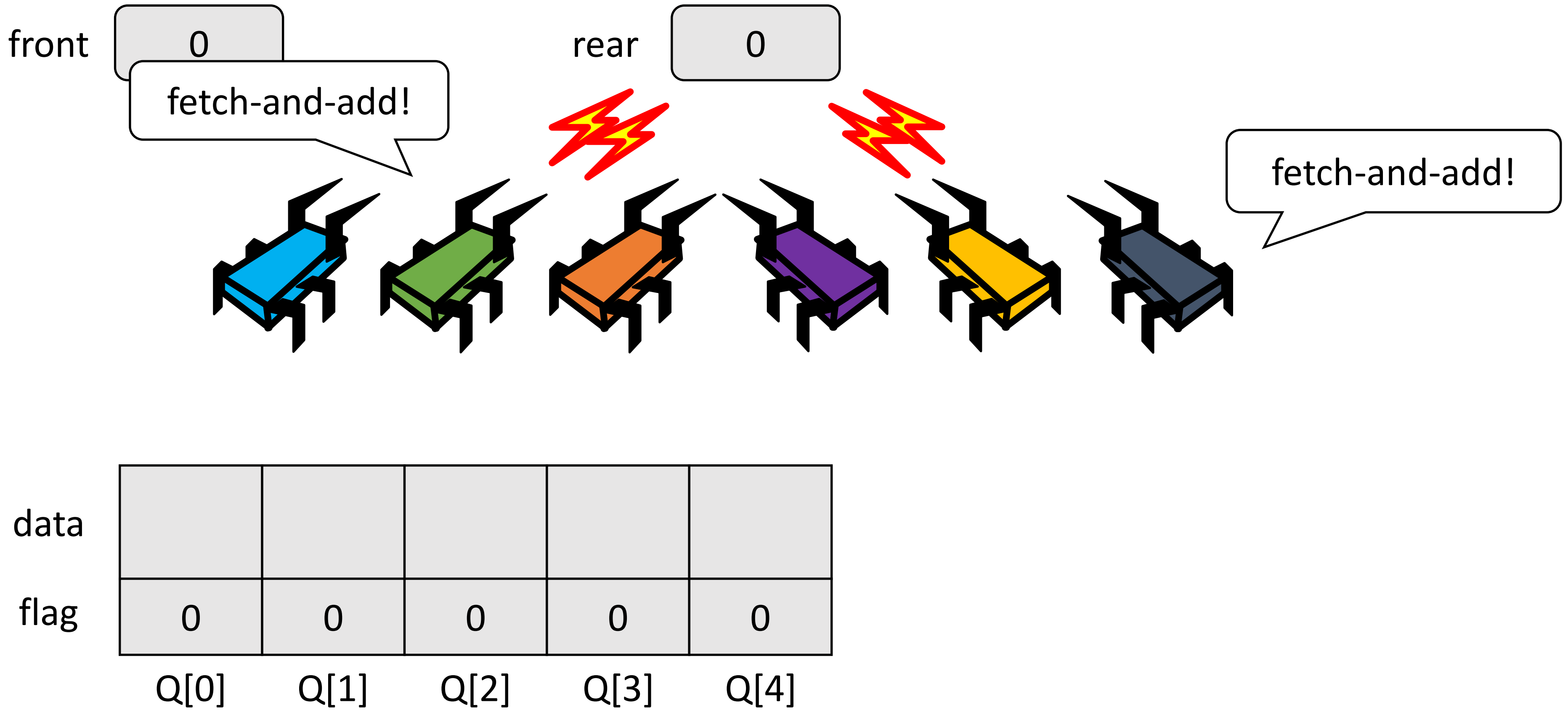
Enqueue



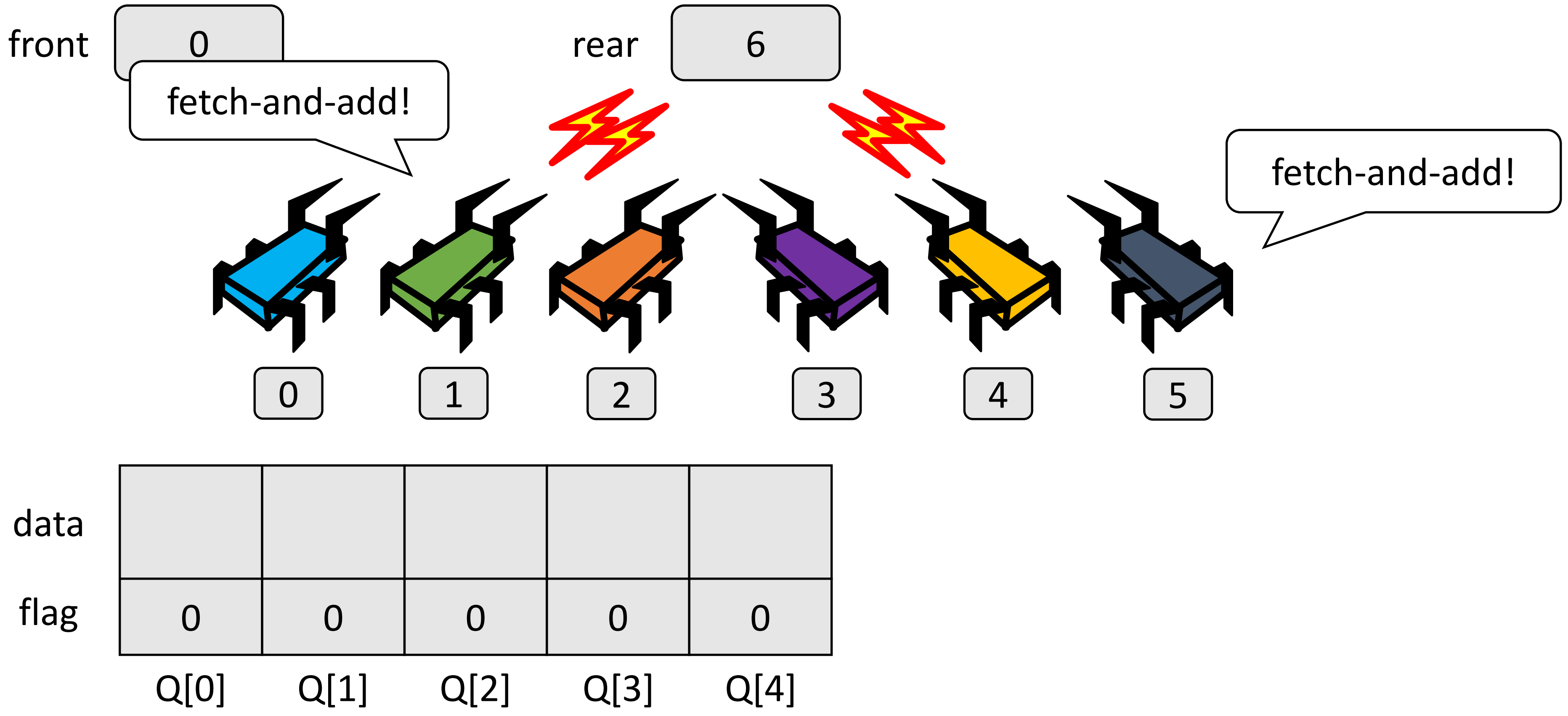
Enqueue


front
0
rear
0
enqueue()
enqueue()
enqueue()
enqueue()
enqueue()
enqueue()
data
flag
0
0
0
0
0
Q[0]
Q[1]
Q[2]
Q[3]
Q[4]

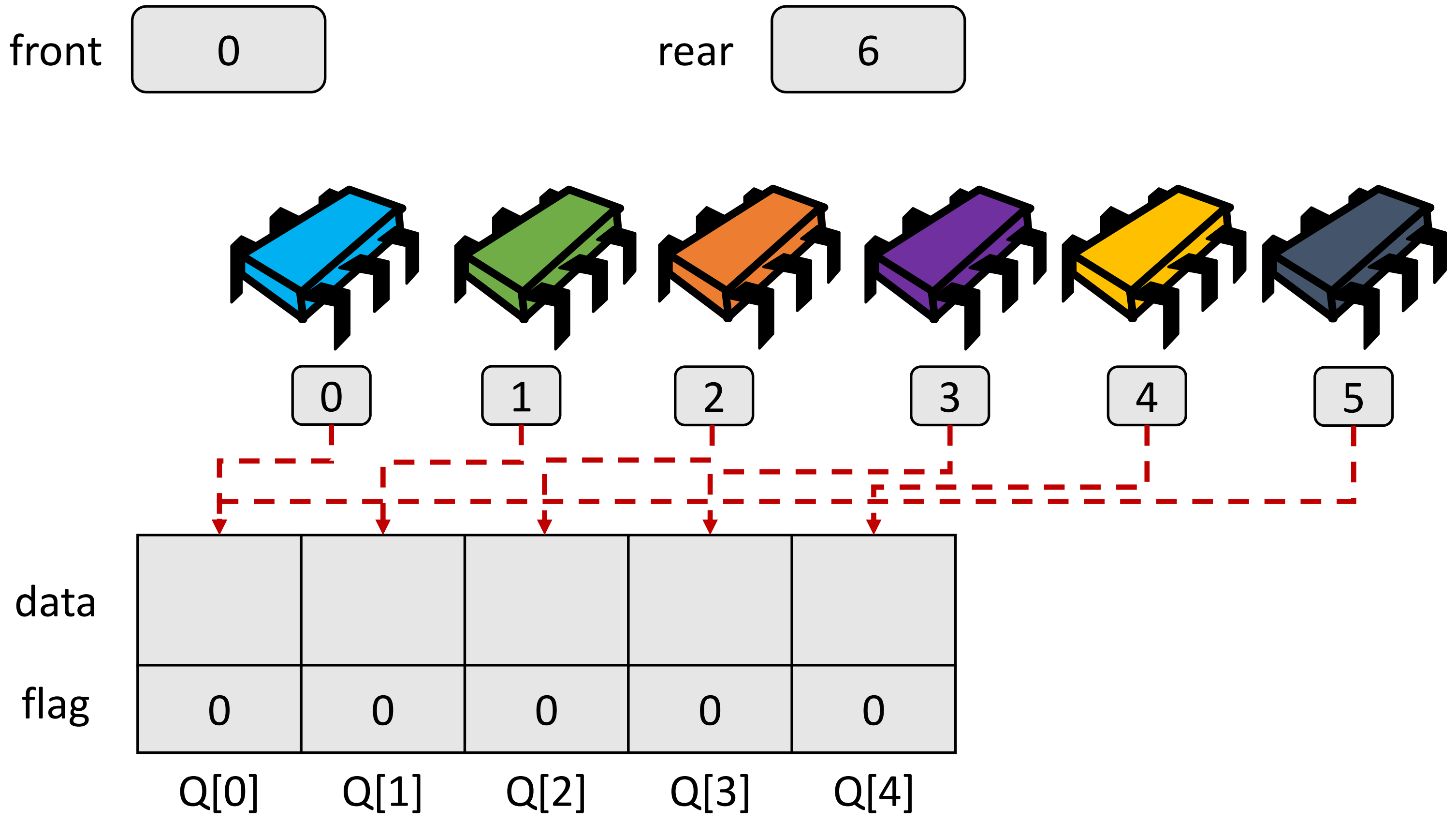
Enqueue


front
0
fetch-and-add!
rear
0
fetch-and-add!
data
flag
0
0
0
0
0
Q[0]
Q[1]
Q[2]
Q[3]
Q[4]

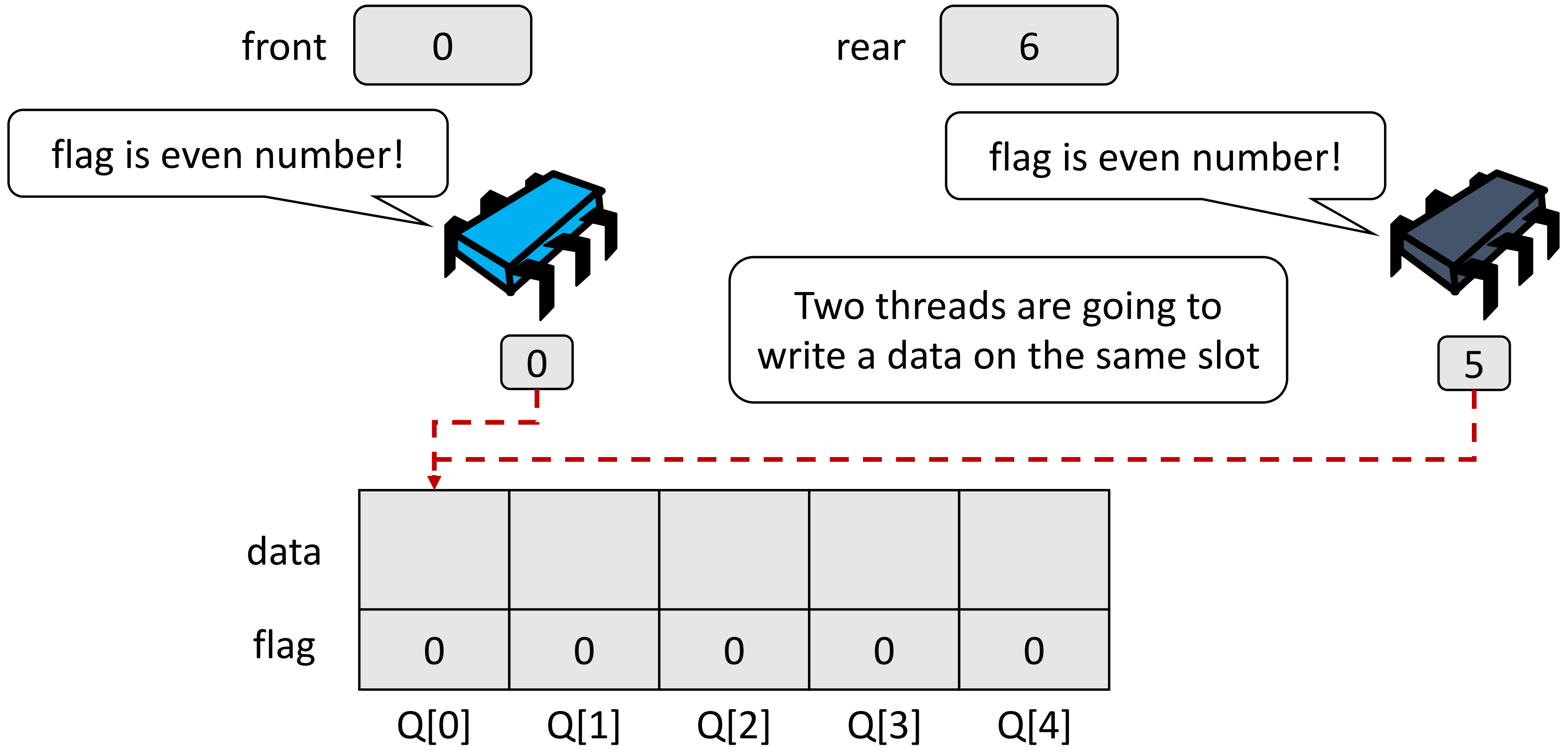
Enqueue


front
0
fetch-and-add!
rear
6
fetch-and-add!
0
1
2
3
4
5
data
flag
0
0
0
0
0
Q[0]
Q[1]
Q[2]
Q[3]
Q[4]

Enqueue

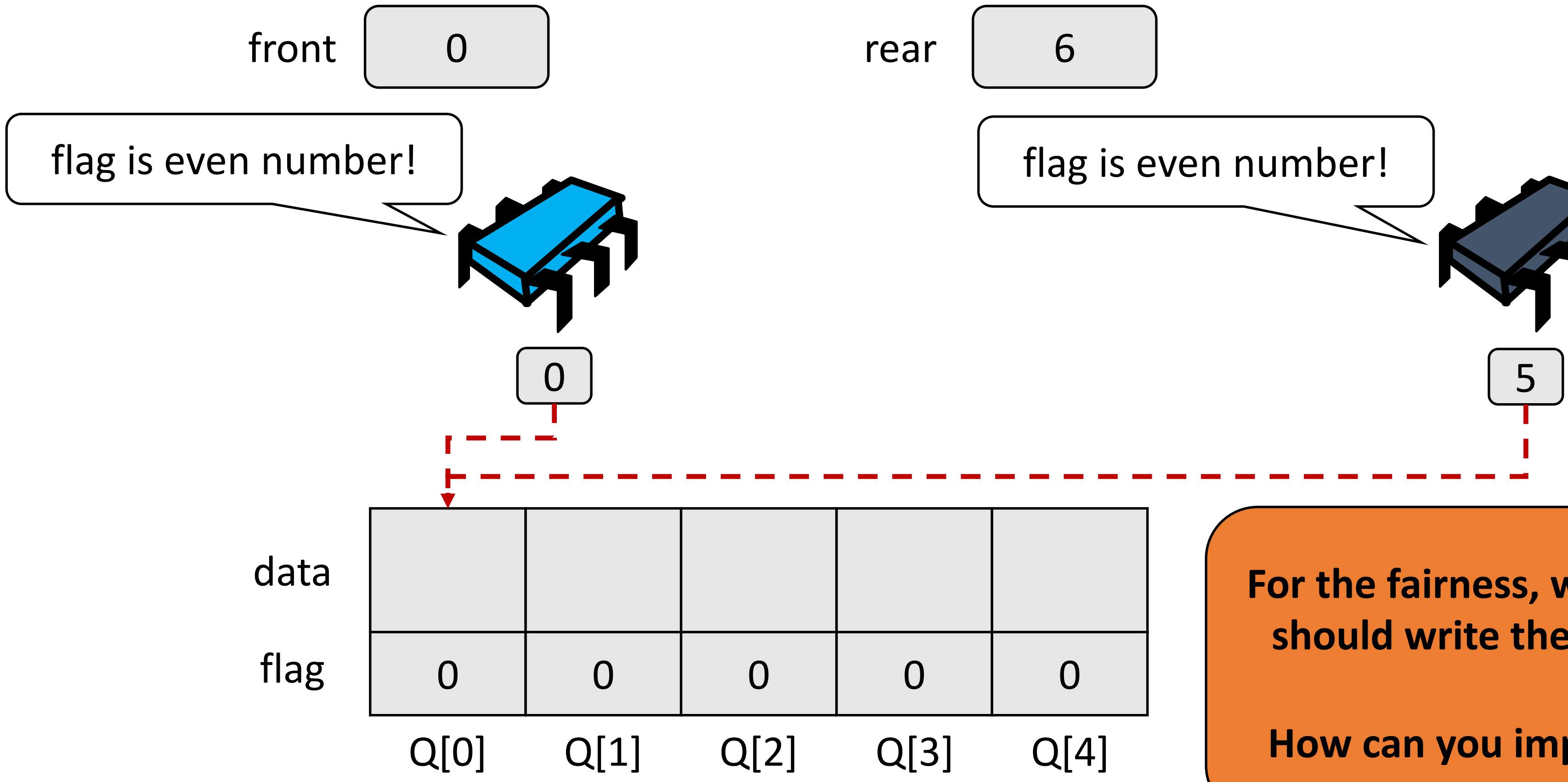


Enqueue



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

Enqueue

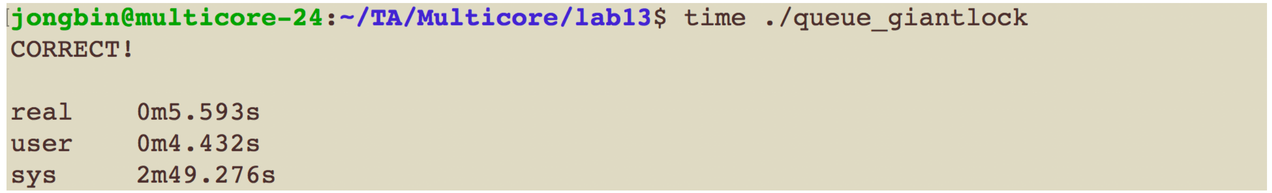


For the fairness, which thread should write the data first?

How can you implement it?

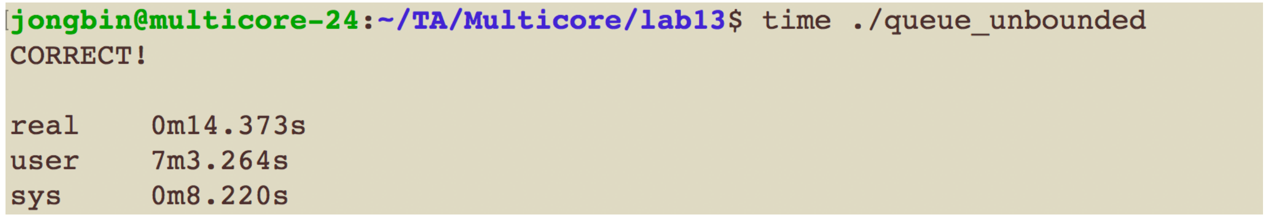
Evaluation

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



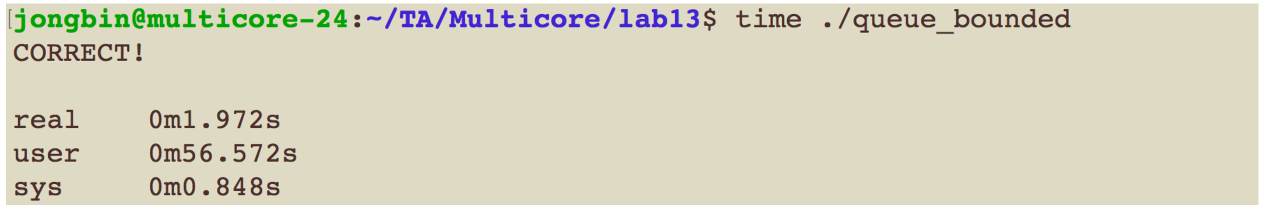
```
jongbin@multicore-24:~/TA/Multicore/lab13$ time ./queue_giantlock
CORRECT!

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