Problem 1188: Design Bounded Blocking Queue

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

Implement a thread-safe bounded blocking queue that has the following methods:

BoundedBlockingQueue(int capacity)

The constructor initializes the queue with a maximum

capacity

.

void enqueue(int element)

Adds an

element

to the front of the queue. If the queue is full, the calling thread is blocked until the queue is no longer full.

int dequeue()

Returns the element at the rear of the queue and removes it. If the queue is empty, the calling thread is blocked until the queue is no longer empty.

int size()

Returns the number of elements currently in the queue.

Your implementation will be tested using multiple threads at the same time. Each thread will either be a producer thread that only makes calls to the

enqueue

method or a consumer thread that only makes calls to the

dequeue

method. The

size

method will be called after every test case.

Please do not use built-in implementations of bounded blocking queue as this will not be accepted in an interview.

Example 1:

Input:

1 1 ["BoundedBlockingQueue","enqueue","dequeue","dequeue","enqueue","enqueue","enqueue","enqueue","dequeue"] [[2],[1],[],[],[0],[2],[3],[4],[]]

Output:

[1,0,2,2]

Explanation:

Number of producer threads = 1 Number of consumer threads = 1

BoundedBlockingQueue queue = new BoundedBlockingQueue(2); // initialize the queue with capacity = 2.

queue.enqueue(1); // The producer thread enqueues 1 to the queue. queue.dequeue(); // The consumer thread calls dequeue and returns 1 from the queue. queue.dequeue(); // Since the queue is empty, the consumer thread is blocked. queue.enqueue(0); // The producer thread enqueues 0 to the queue. The consumer thread is unblocked and returns 0 from the queue. queue.enqueue(2); // The producer thread enqueues 2 to the queue. queue.enqueue(3); // The producer thread enqueues 3 to the queue. queue.enqueue(4); // The producer thread is blocked because the queue's capacity (2) is reached. queue.dequeue(); // The consumer thread returns 2 from the queue. The producer thread is unblocked and enqueues 4 to the queue. queue.size(); // 2 elements remaining in the queue. size() is always called at the end of each test case.

Example 2:

Input:

3 4 ["BoundedBlockingQueue","enqueue","enqueue","enqueue","dequeue","dequeue","dequeue","enqueue"] [[3],[1],[0],[2],[],[],[],[3]]

Output:

[1,0,2,1]

Explanation:

Number of producer threads = 3 Number of consumer threads = 4

BoundedBlockingQueue queue = new BoundedBlockingQueue(3); // initialize the queue with capacity = 3.

queue.enqueue(1); // Producer thread P1 enqueues 1 to the queue. queue.enqueue(0); // Producer thread P2 enqueues 0 to the queue. queue.enqueue(2); // Producer thread P3 enqueues 2 to the queue. queue.dequeue(); // Consumer thread C1 calls dequeue. queue.dequeue(); // Consumer thread C2 calls dequeue. queue.dequeue(); // Consumer thread C3 calls dequeue. queue.enqueue(3); // One of the producer threads enqueues 3 to the queue. queue.size(); // 1 element remaining in the queue.

Since the number of threads for producer/consumer is greater than 1, we do not know how the threads will be scheduled in the operating system, even though the input seems to imply the ordering. Therefore, any of the output [1,0,2] or [1,2,0] or [0,1,2] or [0,2,1] or [2,0,1] or [2,1,0] will be accepted.

Constraints:

1 <= Number of Prdoucers <= 8

1 <= Number of Consumers <= 8

1 <= size <= 30

0 <= element <= 20

The number of calls to

enqueue

is

greater than or equal to

the number of calls to

dequeue

.

At most

40

calls will be made to

enque

,

deque

, and

size

.

Code Snippets

C++:

```
class BoundedBlockingQueue {
public:
BoundedBlockingQueue(int capacity) {

}

void enqueue(int element) {

}

int dequeue() {

}

int size() {

}
};
```

Java:

```
class BoundedBlockingQueue {

public BoundedBlockingQueue(int capacity) {

}

public void enqueue(int element) throws InterruptedException {

}

public int dequeue() throws InterruptedException {
}

public int size() {

}
}
```

Python3:

```
class BoundedBlockingQueue(object):

def __init__(self, capacity: int):



def enqueue(self, element: int) -> None:



def dequeue(self) -> int:



def size(self) -> int:
```

Python:

```
class BoundedBlockingQueue(object):
def __init__(self, capacity):
"""
:type capacity: int
"""



def enqueue(self, element):
"""
:type element: int
:rtype: void
"""



def dequeue(self):
"""
:rtype: int
"""
```

```
def size(self):
"""
:rtype: int
"""
```

Solutions

C++ Solution:

```
/*
* Problem: Design Bounded Blocking Queue
* Difficulty: Medium
* Tags: queue
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

class BoundedBlockingQueue {
public:
BoundedBlockingQueue(int capacity) {

}

void enqueue(int element) {

}

int dequeue() {

}

int size() {

}
};
```

Java Solution:

```
/**
* Problem: Design Bounded Blocking Queue
* Difficulty: Medium
* Tags: queue
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

class BoundedBlockingQueue {

public BoundedBlockingQueue(int capacity) {

}

public void enqueue(int element) throws InterruptedException {

}

public int dequeue() throws InterruptedException {

}

public int size() {

}
}
```

Python3 Solution:

```
"""
Problem: Design Bounded Blocking Queue
Difficulty: Medium
Tags: queue

Approach: Optimized algorithm based on problem constraints
Time Complexity: O(n) to O(n^2) depending on approach
Space Complexity: O(1) to O(n) depending on approach
"""
```

```
class BoundedBlockingQueue(object):

def __init__(self, capacity: int):


def enqueue(self, element: int) -> None:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class BoundedBlockingQueue(object):
def __init__(self, capacity):
"""
:type capacity: int
"""


def enqueue(self, element):
"""
:type element: int
:rtype: void
"""


def dequeue(self):
"""
:rtype: int
"""


def size(self):
"""
:rtype: int
"""
```