# 题目

实现一个拥有如下方法的线程安全有限阻塞队列：

BoundedBlockingQueue(int capacity) 构造方法初始化队列，其中capacity代表队列长度上限。

void enqueue(int element) 在队首增加一个element. 如果队列满，调用线程被阻塞直到队列非满。

int dequeue() 返回队尾元素并从队列中将其删除. 如果队列为空，调用线程被阻塞直到队列非空。

int size() 返回当前队列元素个数。

你的实现将会被多线程同时访问进行测试。每一个线程要么是一个只调用enqueue方法的生产者线程，要么是一个只调用dequeue方法的消费者线程。size方法将会在每一个测试用例之后进行调用。

请不要使用内置的有限阻塞队列实现，否则面试将不会通过。

示例 1:

输入:

1

1

["BoundedBlockingQueue","enqueue","dequeue","dequeue","enqueue","enqueue","enqueue","enqueue","dequeue"]

[[2],[1],[],[],[0],[2],[3],[4],[]]

输出:

[1,0,2,2]

解释:

生产者线程数目 = 1

消费者线程数目 = 1

BoundedBlockingQueue queue = new BoundedBlockingQueue(2); // 使用capacity = 2初始化队列。

queue.enqueue(1); // 生产者线程将1插入队列。

queue.dequeue(); // 消费者线程调用dequeue并返回1。

queue.dequeue(); // 由于队列为空，消费者线程被阻塞。

queue.enqueue(0); // 生产者线程将0插入队列。消费者线程被解除阻塞同时将0弹出队列并返回。

queue.enqueue(2); // 生产者线程将2插入队列。

queue.enqueue(3); // 生产者线程将3插入队列。

queue.enqueue(4); // 生产者线程由于队列长度已达到上限2而被阻塞。

queue.dequeue(); // 消费者线程将2从队列弹出并返回。生产者线程解除阻塞同时将4插入队列。

queue.size(); // 队列中还有2个元素。size()方法在每组测试用例最后调用。

示例 2:

输入:

3

4

["BoundedBlockingQueue","enqueue","enqueue","enqueue","dequeue","dequeue","dequeue","enqueue"]

[[3],[1],[0],[2],[],[],[],[3]]

输出:

[1,0,2,1]

解释:

生产者线程数目 = 3

消费者线程数目 = 4

BoundedBlockingQueue queue = new BoundedBlockingQueue(3); // 使用capacity = 3初始化队列。

queue.enqueue(1); // 生产者线程P1将1插入队列。

queue.enqueue(0); // 生产者线程P2将0插入队列。

queue.enqueue(2); // 生产者线程P3将2插入队列。

queue.dequeue(); // 消费者线程C1调用dequeue。

queue.dequeue(); // 消费者线程C2调用dequeue。

queue.dequeue(); // 消费者线程C3调用dequeue。

queue.enqueue(3); // 其中一个生产者线程将3插入队列。

queue.size(); // 队列中还有1个元素。

由于生产者/消费者线程的数目可能大于1，我们并不知道线程如何被操作系统调度，即使输入看上去隐含了顺序。因此任意一种输出[1,0,2]或[1,2,0]或[0,1,2]或[0,2,1]或[2,0,1]或[2,1,0]都可被接受。

# 分析

## 方法一：互斥锁+条件变量

要实现一个线程安全的有限阻塞队列，核心需求是保证多线程环境下的操作互斥性，并在队列满/空时阻塞相应的生产者/消费者线程。可以通过互斥锁（mutex）保证临界区安全，结合条件变量（condition\_variable）实现线程的阻塞与唤醒。

思路：

1、数据结构选择：使用std::deque作为底层容器存储队列元素（支持高效的队首插入和队尾删除）。

2、互斥锁：用std::mutex确保所有对队列的操作（入队、出队、size）在临界区内执行，避免多线程竞争。

3、条件变量：

- not\_full：当队列满时，阻塞生产者线程；当队列从满变为非满时，唤醒等待的生产者。

- not\_empty：当队列空时，阻塞消费者线程；当队列从空变为非空时，唤醒等待的消费者。

4、阻塞逻辑：

- enqueue：若队列满，生产者线程等待not\_full；否则入队，并唤醒可能等待的消费者（队列从空变为非空时）。

- dequeue：若队列空，消费者线程等待not\_empty；否则出队，并唤醒可能等待的生产者（队列从满变为非满时）。

- size：返回当前队列元素数量，需在互斥锁保护下读取。

代码：

class BoundedBlockingQueue {

private:

deque<int> q; // 底层容器存储队列元素

const int capacity; // 队列容量上限

mutex mtx; // 互斥锁，保护所有对队列的操作

condition\_variable not\_full; // 条件变量：队列非满时唤醒生产者

condition\_variable not\_empty; // 条件变量：队列非空时唤醒消费者

public:

// 构造函数：初始化容量

BoundedBlockingQueue(int capacity) : capacity(capacity) {}

// 入队：在队首增加元素，队列满时阻塞

void enqueue(int element) {

unique\_lock<mutex> lock(mtx); // 加锁（自动释放）

// 等待队列非满（循环避免虚假唤醒）

not\_full.wait(lock, [this]() { return q.size() < capacity; });

q.push\_front(element); // 入队（队首）

not\_empty.notify\_one(); // 唤醒一个等待的消费者（队列从空→非空）

}

// 出队：返回队尾元素并删除，队列空时阻塞

int dequeue() {

unique\_lock<mutex> lock(mtx); // 加锁（自动释放）

// 等待队列非空（循环避免虚假唤醒）

not\_empty.wait(lock, [this]() { return !q.empty(); });

int val = q.back(); // 取队尾元素

q.pop\_back(); // 出队

not\_full.notify\_one(); // 唤醒一个等待的生产者（队列从满→非满）

return val;

}

// 返回当前队列元素个数（线程安全）

int size() {

lock\_guard<mutex> lock(mtx); // 加锁（自动释放）

return q.size();

}

};

代码解释：

1、成员变量：

- deque<int> q：作为队列的底层容器，push\_front实现队首入队，pop\_back实现队尾出队，符合题目对enqueue（队首增加）和dequeue（队尾删除）的定义。

- capacity：队列容量上限，构造函数初始化后不可修改。

- mutex mtx：所有操作必须先获取该锁，确保同一时间只有一个线程访问队列。

- not\_full和not\_empty：条件变量用于线程阻塞与唤醒，避免忙等（busy waiting），提高效率。

2、enqueue方法：

- 用unique\_lock加锁（支持条件变量的wait操作）。

- not\_full.wait(lock, ...)：若队列已满（q.size() == capacity），则释放锁并阻塞当前线程，直到队列非满（被其他线程唤醒且条件满足）。循环判断条件是为了避免虚假唤醒（操作系统可能无理由唤醒线程）。

- 入队后，通过not\_empty.notify\_one()唤醒一个等待的消费者线程（若存在），因为队列可能从空变为非空。

3、dequeue方法：

- 类似地，用unique\_lock加锁，通过not\_empty.wait在队列空时阻塞。

- 出队后，通过not\_full.notify\_one()唤醒一个等待的生产者线程（若存在），因为队列可能从满变为非满。

4、size方法：

- 用lock\_guard加锁（简单的自动释放锁机制），确保读取队列大小时的线程安全。

线程安全与阻塞保证

- 互斥性：所有对q的操作（入队、出队、size）均在锁保护下执行，避免多线程并发修改导致的数据不一致。

- 阻塞机制：通过条件变量实现“队列满时生产者阻塞”“队列空时消费者阻塞”，符合有限阻塞队列的核心需求。

- 唤醒机制：每次入队/出队后针对性唤醒可能等待的线程，减少不必要的阻塞，提高效率。

该实现完全满足题目要求，不依赖内置阻塞队列，且能正确处理多线程环境下的并发访问。

## 方法二：互斥锁+条件变量

两个条件变量，一个互斥量。

互斥量控制临界资源访问的唯一性， 条件变量控制临界资源访问的时序性。

条件变量一个判空，一个判满。

注意，这里要考虑条件变量的惊群问题，必须在while循环中wait

记住口诀，口诀是现编的。

临界资源必加锁，两个条件互等待

读时循环等判空，写时循环等判满

读完解一满等待，写完解一读等待

下面解释一下条件变量的惊群现象， 当多个线程等待在一个条件变量时，比如当前队列为空，多个读线程在rcond的阻塞队列中。这时如果一个写线程写了数据到队列中，并且发出了一个pthread\_cond\_signal(&rcond); 直觉上认为是一个读线程被唤醒，但实际上并非如此，pthread\_cond\_signal也可能激活多个读线程去竞争资源。这就是惊群现象。

那么为什么放在while循环里可以解决惊群问题么？我们要从头捋一下这个加锁和等待条件变量的过程。

以写操作为例：

对互斥量加锁，如果当前已经有线程正持有mutex的锁，那么当前读线程会添加到mutex的阻塞队列中，直到对mutex加锁成功。

加锁成功，进入临界区，如果当前队列已满，则等待wcond条件变量，线程添加到wcond的阻塞队列中。

如果有一个读线程释放了signal信号，一批写线程被唤醒，会从wcond的阻塞队列移到mutex的阻塞队列中，继续尝试对mutex加锁来获取访问临界资源的权限。

对mutex加锁成功后，才有权利向队列中写数据。所以这里要循环判断队列是否仍为满，因为有可能其他被唤醒的线程先获取到了锁，对队列做了写操作，此时队列又一次变满，当前写线程再一次等待wcond信号，被放到wcond的阻塞队列中。

试想一下，如果代码中的while是if的话，会出什么情况？

在step3时会有多个写线程被唤醒，从wcond阻塞队列移动到mutex的阻塞队列中，这些在mutex的阻塞队列中的写线程终将会获取到锁，如果不再次判断条件的话，队列的长度会大于max\_size;

读操作同理。

**代码：**

class BoundedBlockingQueue {

public:

BoundedBlockingQueue(int capacity) {

max\_size = capacity;

pthread\_mutex\_init(&mutex, 0);

pthread\_cond\_init(&rcond, 0);

pthread\_cond\_init(&wcond, 0);

}

void enqueue(int element) {

pthread\_mutex\_lock(&mutex);

while (data.size() == max\_size) { // 处理条件变量wcond惊群问题

pthread\_cond\_wait(&wcond, &mutex);

}

data.push(element);

pthread\_cond\_signal(&rcond);

pthread\_mutex\_unlock(&mutex);

}

int dequeue() {

pthread\_mutex\_lock(&mutex);

while (!data.size()) { // 处理条件变量rcond惊群问题

pthread\_cond\_wait(&rcond, &mutex);

}

int val = data.front(); data.pop();

pthread\_cond\_signal(&wcond);

pthread\_mutex\_unlock(&mutex);

return val;

}

int size() {

return data.size();

}

private:

pthread\_mutex\_t mutex;

pthread\_cond\_t rcond;

pthread\_cond\_t wcond;

int max\_size;

queue<int> data;

};