

CircularBuffer

It looks like you're interested in a **CircularBuffer** (also called a ring buffer). Below is a concise overview and practical implementations in a few languages.

What a CircularBuffer is

- **A fixed-size, FIFO data structure that reuses a single array in a circular manner.**
- **Two indexes: head (read position) and tail (write position).**
- **When tail reaches the end, it wraps to index 0; same for head.**
- **Common policies when full:**
 - **Overwrite oldest element (advance head when writing to a full buffer).**
 - **Reject new writes (return error/boolean).**

Key operations

- **push/enqueue: add an element at tail.**
- **pop/dequeue: remove and return element at head.**
- **peek: view element at head without removing.**
- **size/capacity: number of stored elements and maximum size.**
- **isEmpty/isFull: state checks.**
- **clear/reset.**

Example: Python implementation (overwrite-on-full)

class CircularBuffer:

```
def __init__(self, capacity):
    if capacity <= 0:
        raise ValueError("Capacity must be positive")
    self._buf = [None] * capacity
    self._cap = capacity
    self._head = 0 # next position to read
    self._tail = 0 # next position to write
    self._size = 0

def __len__(self):
    return self._size

def capacity(self):
    return self._cap

def is_empty(self):
```

```

    return self._size == 0

def is_full(self):
    return self._size == self._cap

def push(self, item):
    overwrote = self.is_full()
    self._buf[self._tail] = item
    self._tail = (self._tail + 1) % self._cap
    if overwrote:
        # move head forward to drop oldest
        self._head = (self._head + 1) % self._cap
    else:
        self._size += 1
    return overwrote # True if an old element was overwritten

def pop(self):
    if self.is_empty():
        raise IndexError("pop from empty CircularBuffer")
    item = self._buf[self._head]
    self._buf[self._head] = None # optional, helps GC
    self._head = (self._head + 1) % self._cap
    self._size -= 1
    return item

def peek(self):
    if self.is_empty():
        raise IndexError("peek from empty CircularBuffer")
    return self._buf[self._head]

def clear(self):
    self._buf = [None] * self._cap
    self._head = 0
    self._tail = 0
    self._size = 0

def to_list(self):
    # returns items from head to tail in logical order
    out = []
    for i in range(self._size):
        idx = (self._head + i) % self._cap
        out.append(self._buf[idx])
    return out

```

Example: C++17 templated ring buffer (no overwrite; push fails if full)

```

#include <vector>
#include <optional>
#include <stdexcept>

template <typename T>
class CircularBuffer {
public:
    explicit CircularBuffer(size_t capacity)
        : buf_(capacity), cap_(capacity) {
        if (capacity == 0) throw std::invalid_argument("capacity must be > 0");
    }

    bool empty() const { return size_ == 0; }
    bool full() const { return size_ == cap_; }
    size_t size() const { return size_; }
    size_t capacity() const { return cap_; }

    // Returns true if pushed; false if full.
    bool push(const T& value) {
        if (full()) return false;
        buf_[tail_] = value;
        tail_ = (tail_ + 1) % cap_;
        ++size_;
        return true;
    }

    bool push(T&& value) {
        if (full()) return false;
        buf_[tail_] = std::move(value);
        tail_ = (tail_ + 1) % cap_;
        ++size_;
        return true;
    }

    // Returns std::nullopt if empty.
    std::optional<T> pop() {
        if (empty()) return std::nullopt;
        T value = std::move(buf_[head_]);
        head_ = (head_ + 1) % cap_;
        --size_;
        return value;
    }

    // Peek without removing; std::nullopt if empty
    std::optional<std::reference_wrapper<const T>> peek() const {

```

```

        if (empty()) return std::nullopt;
        return std::cref(buf_[head_]);
    }

    void clear() {
        head_ = tail_ = size_ = 0;
    }

private:
    std::vector<T> buf_;
    size_t cap_;
    size_t head_ = 0;
    size_t tail_ = 0;
    size_t size_ = 0;
};

```

Example: Java implementation with overwrite option

```

public class CircularBuffer<E> {
    private final Object[] buf;
    private final int cap;
    private int head = 0;
    private int tail = 0;
    private int size = 0;
    private final boolean overwriteOnFull;

    public CircularBuffer(int capacity, boolean overwriteOnFull) {
        if (capacity <= 0) throw new IllegalArgumentException("capacity must be
> 0");
        this.buf = new Object[capacity];
        this.cap = capacity;
        this.overwriteOnFull = overwriteOnFull;
    }

    public int size() { return size; }
    public int capacity() { return cap; }
    public boolean isEmpty() { return size == 0; }
    public boolean isFull() { return size == cap; }

    public boolean push(E item) {
        if (isFull() && !overwriteOnFull) return false;
        buf[tail] = item;
        tail = (tail + 1) % cap;
        if (isFull()) {
            head = (head + 1) % cap; // overwrite oldest

```

```

    } else {
        size++;
    }
    return true;
}

public E pop() {
    if (isEmpty()) return null;
    @SuppressWarnings("unchecked")
    E item = (E) buf[head];
    buf[head] = null;
    head = (head + 1) % cap;
    size--;
    return item;
}

public E peek() {
    if (isEmpty()) return null;
    @SuppressWarnings("unchecked")
    E item = (E) buf[head];
    return item;
}

public void clear() {
    for (int i = 0; i < cap; i++) buf[i] = null;
    head = tail = size = 0;
}
}

```

Thread-safety notes

- **For single-producer single-consumer (SPSC), you can avoid locks using atomic head/tail with memory barriers.**
- **For multiple producers/consumers (MPSC/MPMC), consider using locks or specialized lock-free algorithms; ring buffers are popular in high-throughput systems (e.g., Disruptor pattern).**

Common use cases

- **Streaming data (audio, sensors)**
- **Log buffers**
- **Producer-consumer queues with bounded memory**
- **Rate limiting and sliding windows**

If you tell me your language, capacity policy (overwrite vs reject), and whether you need thread-safety, I can tailor an implementation and tests for your needs.

GPT-5

План работы. Мы хотим сделать лучший в мире CircularBuffer

- 1. Написали код CircularBuffer - код компилируется**
- 2. Надо будет небольшой код-ревью для каждой функции.**
- 3. Возможно Предложить альтернативы.**
- 4. Написать код Пример использования.**
- 5. Написать небольшой тест.**

В начале смотрим сам класс Буфера.

Сделай небольшой код-ревью.

```
public class ThreadSafeCircularBuffer<T>  
{  
//private readonly T[] buffer;  
private T[] buffer;  
private int start = 0;  
private int count = 0;  
// private readonly int capacity;  
private int capacity;  
private readonly ReaderWriterLockSlim _lock = new ReaderWriterLockSlim();  
  
// Конструктор  
public ThreadSafeCircularBuffer(int size)  
{  
if (size <= 0)  
throw new ArgumentException("Размер буфера должен быть  
положительным");  
  
capacity = size;  
buffer = new T[size];  
}
```

Ниже краткое код-ревью представленного фрагмента класса ThreadSafeCircularBuffer<t>. Сфокусируюсь на корректности, потокобезопасности, инвариантах и API.

Общее

- Название: ThreadSafeCircularBuffer намекает на полную потокобезопасность. Это требует аккуратного контракта на все публичные методы. Пока виден только конструктор; важно, чтобы все методы чтения/записи, перечисления и метаданные (Count, Capacity,**

IsEmpty/IsFull) использовали один и тот же примитив синхронизации и не допускали гонок.

- **Политика:** не ясно, буфер **overwrite** или **reject** при переполнении. Это влияет на инварианты и на API (**TryEnqueue** vs **EnqueueOverwrite**).
- **Семантика очереди:** обычно у кольцевого буфера есть **head/tail** (или **start/end**). У вас есть **start** и **count** — это достаточно, но потребуются аккуратные вычисления индексов для чтения/записи.
- **Исключения/валидация:** конструктор валидирует **size > 0** — хорошо. Сообщение об ошибке можно сделать более информативным, в том числе указав значение.

Поля

- **buffer:** поле не **readonly**. Если не планируете делать динамическое расширение, имеет смысл сделать его **readonly**, чтобы подчеркнуть неизменность ссылки и упростить reasoning. Сейчас вы также сделали **capacity** не **readonly**, что тоже намекает на потенциальное изменение размера. Если ресайз не планируется — сделайте оба **readonly**.
- **start** и **count:** поля **int** без **volatile**. Так как используется **ReaderWriterLockSlim**, доступ к ним должен происходить только под замком. Тогда **volatile** не нужен. Важно придерживаться этого везде.
- **capacity:** хранится отдельно, хотя **buffer.Length** уже хранит емкость. Дублирование может привести к рассинхронизации. Совет: использовать **buffer.Length** как источник истины и удалить отдельное поле **capacity**.
- **_lock:** **ReaderWriterLockSlim** — уместно, если будут операции только чтения (например, **Peek/Enumerate/Count**) параллельно с редкими записями. Если операций записи много, проще и дешевле может быть обычный **lock**. Также стоит указать **LockRecursionPolicy.NoRecursion** явно, если вы не планируете рекурсию.

Конструктор

- Проверка **size <= 0** — корректно. Можно бросать **ArgumentOutOfRangeException** с именем параметра.
- Инициализация: **capacity = size; buffer = new T[size];** — см. замечание про дублирование. Лучше **buffer = new T[size];** и затем обращаться к **buffer.Length**.

Потенциальные риски и рекомендации по дизайну

- **Инварианты:** для корректности нужно поддерживать: **0 <= start < capacity** и **0 <= count <= capacity**. Индекс записи обычно вычисляется как **(start + count) % capacity**. При **overwrite**-режиме логика иная (при

записи в полный буфер увеличиваем `start` и оставляем `count == capacity`).

- **Исключения vs Try*:** стоит определиться: при попытке чтения из пустого буфера бросаете `InvalidOperationException` или используете `TryDequeue(out T item)`. Аналогично для записи в полный буфер в режиме `reject`.
- **Перечисление:** если будет реализован `IEnumerable<t>`, нужно решить, фиксируете ли срез (копия на момент начала) или допускаете изменение во время итерации (это сложно и требует версионирования и `InvalidOperationException` при модификации).
- **Dispose:** `ReaderWriterLockSlim` реализует `IDisposable` — стоит реализовать `IDisposable` у буфера и освободить `_lock`.
- **Производительность:** `ReaderWriterLockSlim` дороже, чем простой `lock`, особенно при низком содержании параллелизма. Если ваша модель — 1 producer/1 consumer — можно без замков, на `Interlocked`, но это уже другой класс. Для общего случая начните с `lock`.
- **Nullability:** для ссылочных типов полезно продумать поведение с `null` (разрешено ли? обычно да).

Предложения к улучшению прямо сейчас

- **Если `resize` не требуется:**
 - Сделать `buffer` `readonly`.
 - Удалить поле `capacity` и использовать `buffer.Length`.
- **Выбрать политику переполнения и отразить это в API.**
- **Добавить свойства:**
 - `public int Capacity => buffer.Length;`
 - `public int Count { get { using read lock; return count; } }`
 - `public bool IsEmpty => Count == 0; public bool IsFull => Count == Capacity;`
- **Реализовать базовые методы под замком:**
 - `Enqueue(T item)`: либо бросает при `IsFull`, либо `overwrite`.
 - `bool TryEnqueue(T item)`
 - `T Dequeue()` / `bool TryDequeue(out T item)`
 - `T Peek()` / `bool TryPeek(out T item)`
- **Реализовать `Dispose` и финализацию не нужна.**
- **Добавить проверку переполнения арифметики индексов? Не требуется при `int` и корректной модульной арифметике, но аккуратно с отрицательными значениями.**

Небольшой эскиз исправленного каркаса (без реализации методов — далее на следующих шагах можем дописать):

```
public sealed class ThreadSafeCircularBuffer<T> : IDisposable
{
    private readonly T[] _buffer;
```



```

private int _start;
private int _count;
private readonly ReaderWriterLockSlim _lock = new
ReaderWriterLockSlim(LockRecursionPolicy.NoRecursion);

public ThreadSafeCircularBuffer(int capacity)
{
    if (capacity <= 0)
        throw new ArgumentOutOfRangeException(nameof(capacity), capacity,
"Capacity must be positive.");

    _buffer = new T[capacity];
    _start = 0;
    _count = 0;
}

public int Capacity => _buffer.Length;

public int Count
{
    get
    {
        _lock.EnterReadLock();
        try { return _count; }
        finally { _lock.ExitReadLock(); }
    }
}

public bool IsEmpty => Count == 0;
public bool IsFull => Count == Capacity;

// TODO: Enqueue/TryEnqueue, Dequeue/TryDequeue, Peek, Clear, ToArray,
etc.

public void Dispose()
{
    _lock.Dispose();
}
}

```

Если планируется поддержка **overwrite**-режима и **reject**-режима, можно сделать это параметром конструктора или двумя отдельными методами: **EnqueueOverwrite(T item)** и **TryEnqueue(T item)**. Также возможно вынести тип блокировки на интерфейс **ILock** для тестируемости и профилирования.

GPT-5

Я думаю. Может сразу весь код выложить. Или лучше по одной функции выкладывать. ?

Коротко:

- **Если у тебя уже есть рабочий вариант и хочется быстрой обратной связи по архитектуре и стилю — выкладывай целиком. Так видно контекст, инварианты и взаимосвязи методов.**
- **Если код сырой или есть конкретные проблемные места — лучше по частям: сначала интерфейс/публичные методы, затем критичные участки (Enqueue/Dequeue, индексы, обработка переполнения), потом остальное.**
- **Для больших кусков (>300–500 строк) — разбей на модули: интерфейс, конструктор и поля, основные операции, вспомогательные методы, тесты/бенчмарки.**

Практичный план:

- 1. Покажи публичный контракт: какие есть режимы (overwrite/reject), сигнатуры Enqueue/TryEnqueue/Dequeue/TryDequeue/Peek/Count/Capacity/Clear/ToArray, и требования по потокобезопасности.**
- 2. Затем конструктор и поля (индексы head/tail, счетчик, маска для степеней двойки, блокировка/безлок).**
- 3. Потом реализация Enqueue/Dequeue с инвариантами.**
- 4. Добавим остальное и тесты.**

Если удобно — скинь весь код в одном сообщении, я дам обзор и отмечу, что лучше проверить глубже. Если хочешь итеративно — начнем с публичного API и конструктора.

GPT-5

```
public class ThreadSafeCircularBuffer<T>
{
//private readonly T[] buffer;
private T[] buffer;
private int start = 0;
private int count = 0;
// private readonly int capacity;
private int capacity;
private readonly ReaderWriterLockSlim _lock = new ReaderWriterLockSlim();

// Конструктор
public ThreadSafeCircularBuffer(int size)
{
```

```
if (size <= 0)
throw new ArgumentException("Размер буфера должен быть
положительным");
```

```
capacity = size;
buffer = new T[size];
}
```

```
// Запись слева с вытеснением справа
public void EnqueueLeft(T item)
{
_lock.EnterWriteLock();
try
{
int index = (start - 1 + capacity) % capacity;
```

```
if (count == capacity)
start = (start + 1) % capacity;
else
count++;
```

```
buffer[index] = item;
start = index;
}
finally
{
_lock.ExitWriteLock();
}
}
```

```
// Запись справа с вытеснением слева
public void EnqueueRight(T item)
{
_lock.EnterWriteLock();
try
{
int end = (start + count) % capacity;
```

```
if (count == capacity)
start = (start + 1) % capacity;
else
count++;
```

```
buffer[end] = item;
}
```

```
finally
{
    _lock.ExitWriteLock();
}
}

// Чтение слева
public T DequeueLeft()
{
    _lock.EnterWriteLock();
    try
    {
        if (count == 0)
            throw new InvalidOperationException("Буфер пуст");

        T item = buffer[start];
        start = (start + 1) % capacity;
        count--;
        return item;
    }
    finally
    {
        _lock.ExitWriteLock();
    }
}

// Чтение справа
public T DequeueRight()
{
    _lock.EnterWriteLock();
    try
    {
        if (count == 0)
            throw new InvalidOperationException("Буфер пуст");

        int index = (start + count - 1) % capacity;
        T item = buffer[index];
        count--;
        return item;
    }
    finally
    {
        _lock.ExitWriteLock();
    }
}
```

```
// Итератор слева направо  
public IEnumerator<T> IteratorFromLeft()  
{  
    _lock.EnterReadLock();  
    try  
    {  
        int current = start;  
        int elementsRead = 0;  
  
        while (elementsRead < count)  
        {  
            yield return buffer[current];  
            current = (current + 1) % capacity;  
            elementsRead++;  
        }  
    }  
    finally  
    {  
        _lock.ExitReadLock();  
    }  
}  
  
// Итератор справа налево  
public IEnumerator<T> IteratorFromRight()  
{  
    _lock.EnterReadLock();  
    try  
    {  
        int current = (start + count - 1) % capacity;  
        int elementsRead = 0;  
  
        while (elementsRead < count)  
        {  
            yield return buffer[current];  
            current = (current - 1 + capacity) % capacity;  
            elementsRead++;  
        }  
    }  
    finally  
    {  
        _lock.ExitReadLock();  
    }  
}
```

// Текущая длина буфера

public int Count

```
{  
get  
{  
_lock.EnterReadLock();  
try  
{  
return count;  
}  
finally  
{  
_lock.ExitReadLock();  
}  
}  
}
```

// Максимальная вместимость

public int Capacity => capacity;

// Проверка на пустоту

public bool IsEmpty

```
{  
get  
{  
_lock.EnterReadLock();  
try  
{  
return count == 0;  
}  
}
```

finally

```
{  
_lock.ExitReadLock();  
}  
}  
}
```

// Проверка на заполненность

public bool IsFull

```
{  
get  
{  
_lock.EnterReadLock();  
try
```

```
{  
return count == capacity;  
}  
finally  
{  
_lock.ExitReadLock();  
}  
}  
}
```

// Очистка буфера

```
public void Clear()  
{  
_lock.EnterWriteLock();  
try  
{  
start = 0;  
count = 0;  
}  
finally  
{  
_lock.ExitWriteLock();  
}  
}
```

// Получение элемента по индексу

```
public T Peek(int index)  
{  
_lock.EnterReadLock();  
try  
{  
if (index < 0 || index >= count)  
throw new IndexOutOfRangeException();  
  
return buffer[(start + index) % capacity];  
}  
finally  
{  
_lock.ExitReadLock();  
}  
}  
  
// Phind  
/// <summary>  
/// Попытка извлечь элемент без выбрасывания исключения  
/// </summary>
```

```
public bool TryDequeueLeft(out T item)
{
    _lock.EnterWriteLock();
    try
    {
        if (count == 0)
        {
            item = default(T);
            return false;
        }
```

```
        item = buffer[start];
        start = (start + 1) % capacity;
        count--;
        return true;
    }
    finally
    {
        _lock.ExitWriteLock();
    }
}
```

/// <summary>

/// Попытка извлечь элемент с правого конца буфера без выбрасывания исключения

/// </summary>

/// <param name="item">Извлеченный элемент или значение по умолчанию, если буфер пуст</param>

/// <returns>true, если элемент успешно извлечен; false, если буфер пуст</returns>

```
public bool TryDequeueRight(out T item)
{
    _lock.EnterWriteLock();
    try
    {
        if (count == 0)
        {
            item = default(T);
            return false;
        }
```

```
        int index = (start + count - 1) % capacity;
        item = buffer[index];
        count--;
        return true;
    }
```



```

}
finally
{
    _lock.ExitWriteLock();
}
}

/// <summary>
/// Получить текущее состояние заполненности буфера
/// </summary>
public int AvailableSpace => capacity - count;
// -----

// Получение всех элементов слева
public void GetAllFromLeft(List<T> result)
{
    _lock.EnterReadLock();
    try
    {
        result.Clear();
        int current = start;
        for (int i = 0; i < count; i++)
        {
            result.Add(buffer[current]);
            current = (current + 1) % capacity;
        }
    }
    finally
    {
        _lock.ExitReadLock();
    }
}

// Получение всех элементов справа
public void GetAllFromRight(List<T> result)
{
    _lock.EnterReadLock();
    try
    {
        result.Clear();
        int current = (start + count - 1) % capacity;
        for (int i = 0; i < count; i++)
        {
            result.Add(buffer[current]);

```

```
current = (current - 1 + capacity) % capacity;
}
}
finally
{
    _lock.ExitReadLock();
}
}
```

// Попытка получения всех элементов слева (без исключения)

```
public bool TryGetAllFromLeft(List<T> result)
```

```
{  
    _lock.EnterReadLock();
```

```
    try
```

```
    {
```

```
        if (count == 0)
```

```
            return false;
```

```
        result.Clear();
```

```
        int current = start;
```

```
        for (int i = 0; i < count; i++)
```

```
        {
```

```
            result.Add(buffer[current]);
```

```
            current = (current + 1) % capacity;
```

```
        }
```

```
        return true;
```

```
    }
```

```
    finally
```

```
    {
```

```
        _lock.ExitReadLock();
```

```
    }
```

```
}
```

// Попытка получения всех элементов справа (без исключения)

```
public bool TryGetAllFromRight(List<T> result)
```

```
{
```

```
    _lock.EnterReadLock();
```

```
    try
```

```
    {
```

```
        if (count == 0)
```

```
            return false;
```

```
        result.Clear();
```

```
        int current = (start + count - 1) % capacity;
```

```
        for (int i = 0; i < count; i++)
```

```
{  
result.Add(buffer[current]);  
current = (current - 1 + capacity) % capacity;  
}  
return true;  
}  
finally  
{  
_lock.ExitReadLock();  
}  
}
```

// Изменение размера буфера с сохранением элементов слева

```
public void ReSizeFromLeft(int newSize)  
{  
if (newSize <= 0)  
throw new ArgumentException("Размер должен быть положительным");  
  
_lock.EnterWriteLock();  
try  
{  
T[] newBuffer = new T[newSize];  
int elementsToCopy = Math.Min(count, newSize);  
  
// Копируем элементы слева  
int srcIndex = start;  
for (int i = 0; i < elementsToCopy; i++)  
{  
newBuffer[i] = buffer[srcIndex];  
srcIndex = (srcIndex + 1) % capacity;  
}  
  
buffer = newBuffer;  
start = 0;  
count = elementsToCopy;  
capacity = newSize;  
}  
finally  
{  
_lock.ExitWriteLock();  
}  
}
```

// Изменение размера буфера с сохранением элементов справа
public void ReSizeFromRight(int newSize)

```

{
if (newSize <= 0)
throw new ArgumentException("Размер должен быть положительным");

_lock.EnterWriteLock();
try
{
T[] newBuffer = new T[newSize];
int elementsToCopy = Math.Min(count, newSize);

// Вычисляем новый start для нового буфера
int newStart = newSize - elementsToCopy;
int srcIndex = (start + count - 1) % capacity;

// Копируем элементы справа
for (int i = newStart; i < newStart + elementsToCopy; i++)
{
newBuffer[i] = buffer[srcIndex];
srcIndex = (srcIndex - 1 + capacity) % capacity;
}

buffer = newBuffer;
start = newStart;
count = elementsToCopy;
capacity = newSize;
}
finally
{
_lock.ExitWriteLock();
}
}

```

// *****

/*

Сдвиги с вытеснением

При таком сдвиге элементы смещаются на заданное количество позиций, а вытесненные элементы удаляются.

Сдвиги с за цикливанием

При таком сдвиге элементы перемещаются циклически, без потери данных.

*/

/*

ShiftLeft/ShiftRight (с вытеснением):

Сдвигают элементы на заданное количество позиций

Вытесненные элементы удаляются из буфера

Размер буфера уменьшается на количество сдвигов

RotateLeft/RotateRight (с заикливанием):

Перемещают элементы циклически

Ни один элемент не теряется

Просто изменяется точка начала буфера

***/**

// Сдвиг влево с вытеснением

public void ShiftLeft(int positions)

```
{  
_lock.EnterWriteLock();  
try  
{  
if (positions <= 0) return;  
positions %= capacity;
```

```
start = (start + positions) % capacity;  
count = Math.Min(count - positions, capacity);  
}  
finally  
{  
_lock.ExitWriteLock();  
}  
}
```

// Сдвиг вправо с вытеснением

public void ShiftRight(int positions)

```
{  
_lock.EnterWriteLock();  
try  
{  
if (positions <= 0) return;  
positions %= capacity;
```

```
start = (start - positions + capacity) % capacity;  
count = Math.Min(count - positions, capacity);  
}  
finally  
{  
_lock.ExitWriteLock();  
}  
}
```

```
// Циклический сдвиг влево  
public void RotateLeft(int positions)  
{  
_lock.EnterWriteLock();  
try  
{  
if (positions <= 0 || count == 0) return;  
positions %= count;
```

```
start = (start + positions) % capacity;  
}  
finally  
{  
_lock.ExitWriteLock();  
}  
}
```

```
// Циклический сдвиг вправо  
public void RotateRight(int positions)  
{  
_lock.EnterWriteLock();  
try  
{  
if (positions <= 0 || count == 0) return;  
positions %= count;
```

```
start = (start - positions + capacity) % capacity;  
}  
finally  
{  
_lock.ExitWriteLock();  
}  
}
```

```
//public IEnumerable<object> TryGetAllFromLeft()  
//{
```

```

// throw new NotImplementedException();
//}
// Получение всех элементов слева через IEnumerable
public IEnumerable<T> GetAllFromLeft()
{
    _lock.EnterReadLock();
    try
    {
        int current = start;
        for (int i = 0; i < count; i++)
        {
            yield return buffer[current];
            current = (current + 1) % capacity;
        }
    }
    finally
    {
        _lock.ExitReadLock();
    }
}

// Получение всех элементов справа через IEnumerable
public IEnumerable<T> GetAllFromRight()
{
    _lock.EnterReadLock();
    try
    {
        int current = (start + count - 1) % capacity;
        for (int i = 0; i < count; i++)
        {
            yield return buffer[current];
            current = (current - 1 + capacity) % capacity;
        }
    }
    finally
    {
        _lock.ExitReadLock();
    }
}

// Итератор с петлей слева направо
public IEnumerable<T> LoopIteratorFromLeft1()
{
    _lock.EnterReadLock();
    try

```

```

{
if (count == 0) yield break;

// Определяем длину петли
int loopLength = 2 * (capacity - 1);

for (int i = 0; i < loopLength; i++)
{
// Вычисляем индекс в буфере
int offset = i < capacity ? i : loopLength - i;
int index = (start + offset) % capacity;

yield return buffer[index];
}
}
finally
{
_lock.ExitReadLock();
}
}

// Итератор с петлей справа налево
public IEnumerable<T> LoopIteratorFromRight1()
{
_lock.EnterReadLock();
try
{
if (count == 0) yield break;

// Определяем длину петли
int loopLength = 2 * (capacity - 1);

for (int i = 0; i < loopLength; i++)
{
// Вычисляем индекс в буфере
int offset = i < capacity ? i : loopLength - i;
int index = (start + count - 1 - offset + capacity) % capacity;

yield return buffer[index];
}
}
finally
{
_lock.ExitReadLock();
}
}

```



```

}
// -----
// Итератор с петлей слева направо для создания симметричных форм
public IEnumerable<T> LoopIteratorFromLeft2()
{
    _lock.EnterReadLock();
    try
    {
        if (count == 0) yield break;

        // Длина петли всегда 2 * (capacity - 1)
        int loopLength = 2 * (capacity - 1);

        for (int i = 0; i < loopLength; i++)
        {
            // Вычисляем смещение: сначала растет, потом уменьшается
            int offset = i < capacity ? i : loopLength - i;

            // Вычисляем индекс в буфере с учетом начала
            int index = (start + offset) % capacity;

            yield return buffer[index];
        }
    }
    finally
    {
        _lock.ExitReadLock();
    }
}

```

```

// Итератор с петлей справа налево для создания симметричных форм
public IEnumerable<T> LoopIteratorFromRight2()
{
    _lock.EnterReadLock();
    try
    {
        if (count == 0) yield break;

        // Длина петли всегда 2 * (capacity - 1)
        int loopLength = 2 * (capacity - 1);

        for (int i = 0; i < loopLength; i++)
        {
            // Вычисляем смещение: сначала уменьшается, потом растет
            int offset = i < capacity ? i : loopLength - i;

```

```
// Вычисляем индекс с конца буфера  
int index = (start + count - 1 - offset + capacity) % capacity;
```

```
yield return buffer[index];  
}  
}  
finally  
{  
    _lock.ExitReadLock();  
}  
}
```

```
// 25.09.19
```

```
// -----  
public void GetAllFromLeft(ICollection<T> collection)  
{  
    _lock.EnterReadLock();  
    try  
    {  
        int current = start;  
        for (int i = 0; i < count; i++)  
        {  
            collection.Add(buffer[current]);  
            current = (current + 1) % capacity;  
        }  
    }  
    finally  
    {  
        _lock.ExitReadLock();  
    }  
}
```

```
public void GetAllFromRight(ICollection<T> collection)  
{  
    _lock.EnterReadLock();  
    try  
    {  
        int current = (start + count - 1) % capacity;  
        for (int i = 0; i < count; i++)  
        {  
            collection.Add(buffer[current]);  
            current = (current - 1 + capacity) % capacity;  
        }  
    }  
}
```

finally

```
{  
_lock.ExitReadLock();  
}  
}
```

//2. Через массивы

//csharp

public void GetAllFromLeft(T[] array)

```
{  
if (array.Length < count)  
throw new ArgumentException("Массив слишком мал для хранения всех  
элементов");
```

```
_lock.EnterReadLock();
```

```
try
```

```
{  
int current = start;  
for (int i = 0; i < count; i++)  
{  
array[i] = buffer[current];  
current = (current + 1) % capacity;  
}  
}
```

finally

```
{  
_lock.ExitReadLock();  
}  
}
```

public void GetAllFromRight(T[] array)

```
{  
if (array.Length < count)  
throw new ArgumentException("Массив слишком мал для хранения всех  
элементов");
```

```
_lock.EnterReadLock();
```

```
try
```

```
{  
int current = (start + count - 1) % capacity;  
for (int i = 0; i < count; i++)  
{  
array[i] = buffer[current];  
current = (current - 1 + capacity) % capacity;  
}  
}
```

```
}  
finally  
{  
    _lock.ExitReadLock();  
}  
}
```

//3. Комбинированный подход с использованием обобщений

//Можно создать универсальный метод, который будет работать с любыми коллекциями, реализующими ICollection<T>:

```
//csharp  
public void PopulateCollection<TCollection>(TCollection collection)  
where TCollection : ICollection<T>  
{  
    _lock.EnterReadLock();  
    try  
    {  
        int current = start;  
        for (int i = 0; i < count; i++)  
        {  
            collection.Add(buffer[current]);  
            current = (current + 1) % capacity;  
        }  
    }  
    finally  
    {  
        _lock.ExitReadLock();  
    }  
}
```

// Заполнение коллекции элементами слева направо

```
public void PopulateCollectionFromLeft<TCollection>(TCollection collection)  
where TCollection : ICollection<T>  
{  
    _lock.EnterReadLock();  
    try  
    {  
        int current = start; // начинаем с начальной позиции  
  
        for (int i = 0; i < count; i++)  
        {  
            collection.Add(buffer[current]); // добавляем элемент в коллекцию  
            current = (current + 1) % capacity; // переходим к следующему элементу с  
            учётом цикличности  
        }  
    }  
}
```

- **Функциональность богатая: двусторонние enqueue/dequeue, try-варианты, peek, перечисления, resize, сдвиги/ротации, выгрузка в коллекции/массивы. Это круто.**

- **Используется ReaderWriterLockSlim: разумно, но есть важные нюансы с yield/итераторами и копиями snapshot.**
- **Основные проблемы:**
 - **Итераторы с yield держат ReadLock на время потенциально долгой внешней итерации — это может блокировать запись надолго.**
 - **Несогласованность логики сдвигов ShiftLeft/ShiftRight: count может стать отрицательным.**
 - **Некоторые методы проигнорируют частично заполненный буфер в специальных итераторах (LoopIterator*), где используются capacity вместо count.**
 - **При Clear вы не очищаете массив (возможно хорошо для производительности, но стоит явно документировать семантику).**
 - **В ResizeFromRight логика копирования корректна, но нет проверки count > capacity перед расчетом (вы делаете Min, ок), но стоит защититься от странных значений positions в сдвигах и делениях на 0.**
 - **Возможное UB при size == 0 в модульных операциях предотвращено конструкцией в конструкторе — хорошо.**
 - **Enumerator-ы IteratorFromLeft/Right возвращают IEnumerator<t>, но реализованы как методы-итераторы с yield внутри ReadLock — см. основную проблему.**
 - **TryGetAll*/GetAll* с List<t> очищают входящий список — хорошо, но стоит документировать, что список будет очищен. Либо переименовать в CopyAll*.**
 - **Потенциальная гонка состояний в AvailableSpace => capacity - count без lock при чтении полей? У вас ReadLock не используется в этом свойстве — вы используете автосвойство без lock. У вас AvailableSpace — expression-bodied property без блокировки. Это data race.**
 - **Peek возвращает ссылку из массива — это ок, но при T класс ссылки не копируются; норм.**
 - **В TryDequeueRight вы уменьшаете count, но не обновляете start — это верно, так как вы удаляете справа.**
 - **EnqueueLeft/Right при переполнении смещают start, тем самым вытесняют противоположный край — корректно.**

Корректность и инварианты

- **Инварианты:**
 - **$0 \leq \text{count} \leq \text{capacity}$**
 - **$0 \leq \text{start} < \text{capacity}$**
 - **При любой операции индекс вычисляется с модулем capacity**
- **Проверки и проблемы:**
 - **ShiftLeft(int positions):**

- **positions %= capacity;** если **capacity > 0**, ок.
- **count = Math.Min(count - positions, capacity);** если **positions > count**, то **count - positions < 0**, вы получите отрицательный **count**. Это нарушает инвариант. Нужно сделать **count = Math.Max(0, count - positions)**.
- **ShiftRight(int positions):** та же проблема — используйте **Math.Max(0, count - positions)**.
- **RotateLeft/RotateRight:** **positions %= count**, но если **count == 0**, вы раньше выходите — ок. Если **count > 0**, модуль по **count** корректен. Но менять **start** через **capacity** в модуле правильно.
- **IteratorFromLeft/Right:** логически корректны, но смотрите раздел про блокировки.
- **LoopIteratorFromLeft1/2 и Right1/2:**
 - Используют **capacity** при расчете **loopLength** и **offset**, игнорируя **count**. Это, вероятно, неверно: если буфер частично заполнен, вы будете возвращать элементы не только из диапазона заполненных, а фактически весь массив (включая **default(T)** для неиспользуемых ячеек). Судя по комментарию "для создания симметричных форм", возможно, задумывался паттерн по всей **capacity**. Тогда это надо явно документировать. Чаше ожидается симметрия по фактической длине (**count**). Рекомендую использовать **count** для построения петли: **loopLength = 2 * (count - 1)**, и индексы через **count**. И обязательно **handle count == 1 (loopLength = 0)**.
- **Resize:**
 - **ReSizeFromLeft:**
 - Копируете слева **count** или **newSize** — корректно.
 - После копии **start=0, count=elementsToCopy, capacity=newSize** — ок.
 - **ReSizeFromRight:**
 - Вычисляете **newStart = newSize - elementsToCopy;** копируете справа налево, заполняя [**newStart, newStart+elementsToCopy**).
 - Это корректно и сохраняет правую часть. Хорошо.
 - Рекомендация: добавить проверки на новую **capacity** при **newSize < 0** уже есть; также стоит очищать вне диапазона новые ячейки необязательно, но норм.
- **Clear:**
 - Только сброс индексов. Если **T** — **reference type**, старые элементы остаются в массиве и будут удерживаться GC, что может быть нежелательно. Можно предусмотреть опциональное очищение элементов, например метод **Clear(bool wipe)**, или по умолчанию для **reference types** проходиться и занулять если есть требования по освобождению памяти.

- **EnqueueLeft/Right при переполнении:**
 - **Логика корректна. EnqueueLeft:**
 - $index = (start - 1 + capacity) \% capacity$
 - если full, вы продвигаете start вправо, затем пишете по index и присваиваете start=index — net effect: start сдвинут влево и при full вытеснили правый конец. Это правильно.
 - **EnqueueRight:**
 - $end = (start + count) \% capacity$
 - если full, сдвигаете start вправо и фактически вытесняете левый. Правильно.

Потокобезопасность

- **Использование ReaderWriterLockSlim:**
 - Все модифицирующие операции используют EnterWriteLock/ExitWriteLock — ок.
 - Все чтения Count, IsEmpty, IsFull, Peek, итераторы — EnterReadLock/ExitReadLock — ок.
 - Ошибка: свойство AvailableSpace не использует lock. Сейчас это: `public int AvailableSpace => capacity - count;` Это может наблюдать несогласованные значения. Исправьте, как Count/IsEmpty, с ReadLock или Interlocked чтением обеих переменных, но проще ReadLock.
- **Итераторы с yield и lock:**
 - **Pattern yield-with-lock опасен:** когда метод встречает yield, он возвращает управление вызывающему коду, а finally выполнится только после завершения перечисления или его разрушения GC. То есть ReadLock удерживается всю длительность foreach, что может блокировать все писатели и вызвать дедлок при повторной попытке тех же потоков получить write-lock.
 - **Рекомендации:**
 - Делать snapshot копию элементов под ReadLock в локальный массив или List<t>, затем отпустить lock и yield из уже скопированной коллекции. Это уменьшает время удержания блокировки и предотвращает блокировку писателей на длительное время.
 - Или предоставить метод ToArray()/CopyTo и вообще избегать IEnumerable, которые держат lock.
- **Возврат IEnumerable<t> с живыми данными:**
 - IteratorFromLeft/Right возвращают IEnumerable, но реализация на IEnumerator<t> была бы та же проблема. Лучше вернуть snapshot.

API и удобство

- **Дублирование методов:**

- У вас много вариантов `GetAllFromLeft/Right`: возвращающие `IEnumerable`, заполняющие `List<t>`, `ICollection<t>`, массивы, универсальные `PopulateCollection<tcollection>`. Это полезно, но есть большой объем кода. Можно централизовать: один приватный метод `CopyToSpan/CopyToArray` с параметром направления, на его основе построить все публичные. Это сократит риск несогласованности.
- Имена:
 - `IteratorFromLeft/Right` возвращают `IEnumerator<t>`, но возвращаемый тип объявлен как `IEnumerator<t>`. Это ок. Но методы `GetAllFromLeft/Right` возвращают `IEnumerable<t>`. Консистентнее назвать `EnumerateFromLeft/Right` для тех, что возвращают `IEnumerable<t>`. И `IteratorFromLeft/Right` — лучше `IEnumerator<t>` тоже.
- Try-методы:
 - Есть `TryDequeueLeft/Right` — хорошо. Аналогично можно добавить `TryEnqueueLeft/Right`, если вы хотите без исключений сообщать о вытеснении или невозможности (у вас всегда есть вытеснение, так что Try не обязателен).
- Peek:
 - Возвращает элемент по относительному индексу слева. Может пригодиться также `PeekLeft/PeekRight`, `PeekLeft/Right(out T)`, `TryPeek*`.
- Capacity и изменение размера:
 - `Capacity` — `get-only` свойство правильно. При `resize` меняется. Может быть желательно публично доступный метод `EnsureCapacity` или `Grow` до степени 2.

Производительность

- Modulo (%) для индексов:
 - При `capacity` степени двойки можно оптимизировать с маской: `index & (capacity - 1)` вместо `%`. Можно добавить внутренний флаг `powerOfTwo` и маску `int mask = capacity - 1`. Тогда все вычисления индексов быстрее.
- `ReaderWriterLockSlim`:
 - При высоких нагрузках и коротких критических секциях можно рассмотреть `SpinLock` или простую `lock(object)` — но `RWLSlim` даёт преимущество при большом количестве одновременных читателей. Однако из-за `yield` проблема нивелирует это преимущество. После снапшота `RW` можно быть полезным.
- Избегать лишних модулей:
 - В местах где часто вычисляется `(start + count) % capacity` — доп. локальные переменные.

Edge cases

- В конструкторе `size <= 0` — исключение. Хорошо.
- В `RotateLeft/Right` при `positions` может быть `> int.MaxValue`?
Практически нет, но при отрицательных значениях вы фильтруете `if positions <= 0 return` — ок.
- В `ShiftLeft/Right` `positions % capacity` — если `capacity == 0` никогда, т.к. запрещено. Ок.
- В `LoopIterator*` при `capacity == 1 loopLength = 0` — цикл не выполнится. Если смысл в «петле» — возможно ожидалось вернуть один элемент дважды? Нужно определить контракт. Сейчас безопасно.
- Многопоточность и перечисления: если пользователь комбинирует методы перечисления без копии с параллельной записью, вы избежите `torn reads` из-за `ReadLock`. Но будет блокировка писателей — см. выше.

Конкретные замечания по коду

1. `AvailableSpace` без `lock` — исправить:

```
public int AvailableSpace
{
    get
    {
        _lock.EnterReadLock();
        try { return capacity - count; }
        finally { _lock.ExitReadLock(); }
    }
}
```

2. `ShiftLeft/Right`: не допускать отрицательного `count`:

```
count = Math.Max(0, count - positions);
```

3. Итераторы и методы, возвращающие `IEnumerable<t>`:

- Реализовать через `snapshot`. Пример — ниже.

4. `IteratorFromLeft/Right` сигнатуры:

- Сейчас `IteratorFromLeft` возвращает `IEnumerator<t>`. Обычно публичный API отдает `IEnumerable<t>`, чтобы его можно было использовать напрямую в `foreach`. Если хотите именно `IEnumerator<t>`, лучше добавить `overload IEnumerable`.

5. `GetAllFromLeft/Right(List<t>)` очищают вход: задокументировать.

6. `Clear`:

- Добавить опцию очистки массива, или второй метод `Wipe` для освобождения ссылок. Для `value types` это не нужно.

7. `Resize` методы:

- Добавить проверки `newSize == capacity`: быстрый `return`.
- В `ResizeFromRight`: после копии `start = newStart`. Это ок.

8. `TryDequeueRight`: хорошо.

9. `Rotate*` по `count`:

- При `count == 0` вы уже `return`; при `count == 1` `positions % 1 == 1 ==> 0`, ок.
- 10. Дублирующий код `LoopIteratorFromLeft1/2` и `Right1/2` одинаков:
 - Поддерживайте одну реализацию или обобщите. И определитесь, должны ли эти методы работать по `count` или по `capacity`.
- 11. `IteratorFromLeft/Right` как методы, возвращающие `IEnumerator<t>`:
 - Сейчас они написаны с `yield return` в методах, тип возвращаемого — `IEnumerator<t>`. Это легально, компилятор создаст объект-итератор, реализующий `IEnumerator<t>`. Но из-за `lock` это опасно. Смените на `IEnumerable<t> + snapshot`.
- 12. В `EnqueueLeft` при `full`:
 - Вы делаете `if (count == capacity) start = (start + 1) % capacity;` затем `buffer[index] = item;` затем `start = index;`
 - Это эквивалентно одному `start = index` после корректного вычисления `index`. Но промежуточный инкремент `start` сдвигает окно и может не требоваться. Однако поведение корректное. Можно упростить:
 - Если `full`: `start = (start - 1 + capacity) % capacity; buffer[start] = item;` (`count` не меняется).
 - Если не `full`: `start = (start - 1 + capacity) % capacity; buffer[start] = item; count++;`
 - Это уменьшит одну модульную операцию и сделает код проще.
- 13. `EnqueueRight` можно упростить аналогично:
 - `var end = (start + count) % capacity;`
 - `if (count == capacity) { buffer[end] = item; start = (start + 1) % capacity; }`
`else { buffer[end] = item; count++; }`
- 14. `Peek(int index)` — ок, но традиционно бросают `ArgumentOutOfRangeException`.
- 15. Потокобезопасность свойства `Capacity` — только `get` без `lock`. Меняется только под `write-lock`. Читать без `lock` формально `data race`. Добавьте `ReadLock` как для `Count` или задокументируйте, что `Capacity` может быть изменена только вызывающим потоком через `Resize*` и после этого он знает значение. Лучше добавить `ReadLock`.

Рекомендованные правки (ключевые куски)

1. Fix `ShiftLeft/ShiftRight`:

```
public void ShiftLeft(int positions)
{
    _lock.EnterWriteLock();
    try
    {
```

```

    if (positions <= 0 || count == 0) return;
    positions %= capacity;
    // смещаем начало влево относительно логической левой границы
    start = (start + positions) % capacity;
    count = Math.Max(0, count - positions);
}
finally
{
    _lock.ExitWriteLock();
}
}

```

```

public void ShiftRight(int positions)
{
    _lock.EnterWriteLock();
    try
    {
        if (positions <= 0 || count == 0) return;
        positions %= capacity;
        start = (start - positions + capacity) % capacity;
        count = Math.Max(0, count - positions);
    }
    finally
    {
        _lock.ExitWriteLock();
    }
}

```

1. Protected properties:

```

public int AvailableSpace
{
    get
    {
        _lock.EnterReadLock();
        try { return capacity - count; }
        finally { _lock.ExitReadLock(); }
    }
}

```

```

public int Capacity
{
    get
    {
        _lock.EnterReadLock();
    }
}

```

```

        try { return capacity; }
        finally { _lock.ExitReadLock(); }
    }
}

```

1. Snapshot-перечисление:

```

private T[] SnapshotLeft()
{
    _lock.EnterReadLock();
    try
    {
        var arr = new T[count];
        int current = start;
        for (int i = 0; i < count; i++)
        {
            arr[i] = buffer[current];
            current = (current + 1) % capacity;
        }
        return arr;
    }
    finally
    {
        _lock.ExitReadLock();
    }
}

private T[] SnapshotRight()
{
    _lock.EnterReadLock();
    try
    {
        var arr = new T[count];
        int current = (start + count - 1 + capacity) % capacity;
        for (int i = 0; i < count; i++)
        {
            arr[i] = buffer[current];
            current = (current - 1 + capacity) % capacity;
        }
        return arr;
    }
    finally
    {
        _lock.ExitReadLock();
    }
}

```

```

}

public IEnumerable<T> EnumerateFromLeft()
{
    var snap = SnapshotLeft();
    for (int i = 0; i < snap.Length; i++)
        yield return snap[i];
}

public IEnumerable<T> EnumerateFromRight()
{
    var snap = SnapshotRight();
    for (int i = 0; i < snap.Length; i++)
        yield return snap[i];
}

```

1. Упрощение EnqueueLeft/Right:

```

public void EnqueueLeft(T item)
{
    _lock.EnterWriteLock();
    try
    {
        int newStart = (start - 1 + capacity) % capacity;
        buffer[newStart] = item;
        if (count == capacity)
        {
            // вытеснили справа, start сдвигается на newStart, count не меняется
            start = newStart;
        }
        else
        {
            start = newStart;
            count++;
        }
    }
    finally
    {
        _lock.ExitWriteLock();
    }
}

public void EnqueueRight(T item)
{
    _lock.EnterWriteLock();

```

```

try
{
    int end = (start + count) % capacity;
    buffer[end] = item;
    if (count == capacity)
    {
        // вытесняем слева
        start = (start + 1) % capacity;
    }
    else
    {
        count++;
    }
}
finally
{
    _lock.ExitWriteLock();
}
}

```

1. Очистка с опцией wipe:

```

public void Clear(bool wipe = false)
{
    _lock.EnterWriteLock();
    try
    {
        if (wipe && RuntimeHelpers.IsReferenceOrContainsReferences<T>())
        {
            int current = start;
            for (int i = 0; i < count; i++)
            {
                buffer[current] = default!;
                current = (current + 1) % capacity;
            }
        }
        start = 0;
        count = 0;
    }
    finally
    {
        _lock.ExitWriteLock();
    }
}

```

1. LoopIterator* по фактической длине:

- Если хотите строить «петлю» по count:

```
public IEnumerable<T> LoopEnumerateFromLeft()
{
    var snap = SnapshotLeft();
    if (snap.Length == 0) yield break;
    int loopLength = 2 * (snap.Length - 1);
    for (int i = 0; i < loopLength; i++)
    {
        int offset = i < snap.Length ? i : loopLength - i;
        yield return snap[offset];
    }
}
```

- Аналогично для Right, используя снапшот справа.

1. Документация

- Я бы добавил XML-комментарии к каждому публичному методу с четкой семантикой вытеснения, сложности $O(1)/O(n)$, behavior при многопоточности (snapshot vs live), и гарантии исключений.

Тесты, на что обратить внимание

- Многопоточные:
 - Несколько читателей + писатели: убедиться, что писатели не залипают на итераторах (после правки с snapshot).
 - Сценарии переполнения: EnqueueLeft/Right при полном буфере, проверка вытеснения правильного края.
- Сдвиги/ротации:
 - ShiftLeft/Right в диапазоне $[0..count+2]$, особенно positions $\geq capacity$ и positions $> count$.
 - Rotate при count 0/1/N, positions большие и кратные count.
- Resize:
 - ResizeFromLeft/Right при уменьшении/увеличении; сравнить последовательности до/после.
- Границы индексов:
 - Peek с -1, $\geq count$ должно кидать правильное исключение.
- Очистка:
 - Clear(wipe) освобождает ссылки; GC/WeakReference можно проверить.

Итого

- Исправьте: `AvailableSpace/Ccapacity` с `ReadLock`, `ShiftLeft/Right` с `Max(0,..)`, устраняйте `yield` под удерживаемым `lock` используя `snapshot`, при желании упростите `EnqueueLeft/Right`.
- Определитесь с семантикой `LoopIterator*`: по `capacity` или `count`. Если по `capacity` — четко документируйте, что элементы вне `count` могут быть `default(T)` и это «формообразующие» методы, а не чтение содержимого.
- Подумайте об освобождении ссылок в `Clear/Resize`.

Если хотите, могу прислать полностью переписанный класс с указанными изменениями и унификацией методов перечисления/копирования, а также базовыми тестами `xUnit/NUnit`.

GPT-5

В целом нормальная работа ?

Я многопоточность уже потом добавлял.

Вообще я его проектировал для работы в однопоточной среде, потому что как-то не очень мне нравится многопоточный доступ в такой сложной реализации (`count`, `capacity`, вытеснения).

В целом — да, базовая идея и реализация для однопоточного сценария выглядят здраво, а «добавленная потом» многопоточность чаще всего и приводит к шероховатостям именно в структурах с кольцевыми буферами/вытеснением. Пара тезисов с акцентом на практику:

- Если изначально целевой сценарий — однопоточность, лучше так и зафиксировать контрактом:
 - Явно указать в XML-документации/комментариях, что потокобезопасность не гарантируется.
 - Не тянуть `ReaderWriterLockSlim/lock` внутрь — это усложняет и замедляет «обычный» путь. Вынесите «потокобезопасную» версию в отдельный адаптер/обертку.
- Если многопоточность все же нужна:
 - Предпочтительнее выделить два варианта: `NonThreadSafe` и `ThreadSafe` (обертка). Это снимет часть когнитивной нагрузки и позволит оптимизировать однопоточный путь.
 - `ReaderWriterLockSlim` подходит для «много чтений — мало записей», но для кольцевых буферов с частыми `push/pop` `RWLock` часто уступает одному монолитному `lock` по латентности. Начните с простого `lock` и измерьте.
 - Избегайте `lock escalation` при вложенных вызовах (например, перечисление, которое внутри читает `Count` и индексирует). Лучше спроектировать «атомарные снапшоты» для операций перечисления/копирования.
 - Не смешивайте логику корректности и синхронизации: каждая публичная операция должна сама полностью обеспечивать свою

безопасность, а не полагаться на «внешний» уже-взятый ReadLock.

Замечания по типичным узким местам таких реализаций:

- **Count vs Capacity:**
 - Все операции, итерирующиеся по данным, должны идти по **Count**, а не по **Capacity**. Если есть методы низкоуровневого доступа к «сырому» буферу (по **Capacity**), их четко документировать как «формообразующие», не читающие содержимое.
- **Очистка ссылок:**
 - В **Clear**, **RemoveRange**, **Dequeue/Pop** и при перезаписи вытеснением — зануляйте ссылки под управляемыми типами, чтобы не удерживать память дольше нужного (особенно при долгоживущем буфере).
- **Исключения и контракт:**
 - Индексация: четкая проверка границ $0 \leq \text{index} < \text{Count}$, иначе **ArgumentOutOfRangeException**.
 - Операции, меняющие размеры, должны гарантировать непротиворечивость даже при исключении (**no-throw** **move/копирование** участков массива желательно).
- **Перечисление:**
 - Итератор должен быть устойчив к модификациям? Если нет — бросайте **InvalidOperationException** при обнаружении версии (**version stamp**) изменения коллекции.
 - Нужен быстрый путь **CopyTo/ToArray** без аллокации лишнего (две **Array.Copy** при «разрыве» кольца).
- **Резайз/рост:**
 - Рост в 2х (или 1.5х) с переносом по логическому порядку. При переносе — единый линейный сегмент (**start..start+Count**) в новый массив с нуля.
 - После **resize** помнить обнулить «хвост» старого массива, если в нем были активные ссылки, либо перед переносом, либо после.
- **API четкость:**
 - Если коллекция «**Queue-подобная**» — дайте семантики **TryEnqueue/Enqueue/Dequeue/TryDequeue**, **Peek/TryPeek**.
 - Если «**Deque**» — добавьте операции к обоим концам и держите их в равной асимптотике.
- **Инварианты:**
 - Всегда держите: $0 \leq \text{Count} \leq \text{Capacity}$; **head** в $[0..\text{Capacity})$, **tail** = $(\text{head} + \text{Count}) \% \text{Capacity}$.
 - Все мутации сводите к двум-трем базовым примитивам, чтобы не дублировать индексную арифметику и не плодить ошибки.

Про потокобезопасность с вытеснением:

- Если допускаете «вытеснение» при Enqueue, это усложняет lock-стратегию: читатели могут увидеть непоследовательный срез, если не защищать голову/хвост общей критической секцией. Проще: один lock на все операции. Если нужна высокая пропускная — подумайте о lock-free варианте, но это отдельный проект с сильной сложностью (Interlocked на head/tail + Single-Writer/Single-Reader или MPSC/SPSC паттерны).
- ReaderWriterLockSlim с длинными Read секциями на перечислении может блокировать записывающих, что ощущается как «фризы». Для этого лучше:
 - Выдать снимок (ToArray) без удержания ReadLock на весь foreach.
 - Или хранить version и не блокировать чтение, а при несоответствии версии — прерывать enumerator.

Что можно улучшить «в коде прямо сейчас» даже для однопоточного пути:

- Убрать внутренние RWLock'и; оставить чистую логику, добавить поле version и корректные исключения в Enumerator.
- Проверить, что все публичные методы используют Count, а не Capacity, при проходах.
- Проставить обнуление ссылок в Clear/Resize/Dequeue/RemoveRange и там, где происходит вытеснение.
- Упростить индексную арифметику через один helper: `PhysicalIndex(int logical) => (head + logical) % capacity`.
- Добавить тесты:
 - Базовые: Enqueue/Dequeue, wrap-around, рост, вытеснение, Clear, ToArray, CopyTo (при разрыве).
 - Property-based: после любой последовательности операций логический порядок равен эталону (например, List<t> как оракул).
 - Enumerators: модификация во время foreach вызывает InvalidOperationException.
 - Если оставляете ThreadSafe-обертку — стресс-тест на контеншн.

Если хочешь, пришлю:

- 2 варианта: NonThreadSafe кольцевой буфер (Deque) + ThreadSafe обертку с простым lock.
- Переписанный Enumerator с version.
- Набор тестов и бенчмарков BenchmarkDotNet для сравнения RWLock vs lock vs без синхронизации.

GPT-5

Вы

"Если хочешь, пришлю:

2 варианта: NonThreadSafe кольцевой буфер (Deque) + ThreadSafe обертку с простым lock.

Переписанный Enumerator с version.

Набор тестов и бенчмарков BenchmarkDotNet для сравнения RWLock vs lock vs без синхронизации"

Высылайте. Посмотрим, что из этого выйдет.

Отправляю обещанные варианты. Я ориентировался на .NET 6+/C# 10. Код самодостаточный, можно копипастить в один solution и прогонять.

Структура:

- **Core: небезопасный для потоков кольцевой буфер (Deque/RingBuffer).**
- **ThreadSafe-обертка сверху с простым lock.**
- **Enumerator с версионностью (fail-fast).**
- **Набор минимальных тестов xUnit.**
- **Бенчмарки BenchmarkDotNet: без синхронизации vs lock vs ReaderWriterLockSlim (для интереса).**

1. Непотокобезопасный кольцевой буфер (Deque/RingBuffer)

```
using System;  
using System.Collections;  
using System.Collections.Generic;
```

```
namespace RingBufferLib
```

```
{  
    // Двухсторонняя очередь на кольцевом массиве:  
    // - O(1) push/pop с обеих сторон  
    // - Автоматическое расширение (двукратное)  
    // - Fail-fast Enumerator по version  
    public class Deque<T> : IReadOnlyCollection<T>, IEnumerable<T>  
    {  
        private T[] _buffer;  
        private int _head; // указывает на индекс первого элемента  
        private int _tail; // указывает на позицию СЛЕДУЮЩУЮ за последним  
элементом  
        private int _count;  
        private int _version;  
  
        private const int DefaultCapacity = 4;  
  
        public Deque(int capacity = DefaultCapacity)  
        {  
            if (capacity < 0) throw new  
ArgumentOutOfRangeException(nameof(capacity));
```

```

    _buffer = capacity == 0 ? Array.Empty<T>() : new T[capacity];
    _head = 0;
    _tail = 0;
    _count = 0;
    _version = 0;
}

public int Count => _count;
public int Capacity => _buffer.Length;
public bool IsEmpty => _count == 0;

public T this[int index]
{
    get
    {
        if ((uint)index >= (uint)_count) throw new
ArgumentOutOfRangeException(nameof(index));
        return _buffer[PhysicalIndex(index)];
    }
    set
    {
        if ((uint)index >= (uint)_count) throw new
ArgumentOutOfRangeException(nameof(index));
        _buffer[PhysicalIndex(index)] = value;
        _version++;
    }
}

// Добавить в конец (правую сторону)
public void PushBack(T item)
{
    EnsureCapacityForOneMore();
    _buffer[_tail] = item;
    _tail = NextIndex(_tail);
    _count++;
    _version++;
}

// Добавить в начало (левую сторону)
public void PushFront(T item)
{
    EnsureCapacityForOneMore();
    _head = PrevIndex(_head);
    _buffer[_head] = item;
    _count++;
}

```

```

        _version++;
    }

    // Удалить с конца
    public T PopBack()
    {
        if (_count == 0) throw new InvalidOperationException("Deque is empty.");
        _tail = PrevIndex(_tail);
        var item = _buffer[_tail];
        _buffer[_tail] = default!;
        _count--;
        _version++;
        return item;
    }

    // Удалить с начала
    public T PopFront()
    {
        if (_count == 0) throw new InvalidOperationException("Deque is empty.");
        var item = _buffer[_head];
        _buffer[_head] = default!;
        _head = NextIndex(_head);
        _count--;
        _version++;
        return item;
    }

    public T PeekFront()
    {
        if (_count == 0) throw new InvalidOperationException("Deque is empty.");
        return _buffer[_head];
    }

    public T PeekBack()
    {
        if (_count == 0) throw new InvalidOperationException("Deque is empty.");
        return _buffer[PrevIndex(_tail)];
    }

    public void Clear()
    {

```

```

    if (_count > 0)
    {
        if (RuntimeHelpersIsReferenceOrContainsReferences<T>())
        {
            // очистка слотов для освобождения ссылок
            for (int i = 0; i < _count; i++)
            {
                _buffer[PhysicalIndex(i)] = default!;
            }
        }
        _head = 0;
        _tail = 0;
        _count = 0;
        _version++;
    }
}

```

```

public bool TryPopFront(out T value)
{
    if (_count == 0)
    {
        value = default!;
        return false;
    }
    value = PopFront();
    return true;
}

```

```

public bool TryPopBack(out T value)
{
    if (_count == 0)
    {
        value = default!;
        return false;
    }
    value = PopBack();
    return true;
}

```

```

public bool Contains(T item, IEqualityComparer<T>? comparer = null)
{
    comparer ??= EqualityComparer<T>.Default;
    for (int i = 0; i < _count; i++)
    {
        if (comparer.Equals(_buffer[PhysicalIndex(i)], item))

```

```

        return true;
    }
    return false;
}

public void CopyTo(T[] array, int arrayIndex)
{
    if (array == null) throw new ArgumentNullException(nameof(array));
    if (arrayIndex < 0 || arrayIndex > array.Length) throw new
ArgumentOutOfRangeException(nameof(arrayIndex));
    if (array.Length - arrayIndex < _count) throw new
ArgumentException("Destination array is too small.");

    int rightSegment = Math.Min(_count, Capacity - _head);
    Array.Copy(_buffer, _head, array, arrayIndex, rightSegment);
    Array.Copy(_buffer, 0, array, arrayIndex + rightSegment, _count -
rightSegment);
}

public Enumerator GetEnumerator() => new Enumerator(this);
IEnumerator<T> IEnumerable<T>.GetEnumerator() => new
Enumerator(this);
IEnumerator IEnumerable.GetEnumerator() => new Enumerator(this);

public struct Enumerator : IEnumerator<T>
{
    private readonly Deque<T> _deque;
    private readonly int _version;
    private int _index; // логический индекс от 0..Count-1
    private T _current;

    internal Enumerator(Deque<T> deque)
    {
        _deque = deque;
        _version = deque._version;
        _index = -1;
        _current = default!;
    }

    public T Current => _current;
    object IEnumerator.Current => _current!;

    public bool MoveNext()
    {
        if (_version != _deque._version)

```



```
        throw new InvalidOperationException("Collection was modified  
during enumeration.");
```

```
        int next = _index + 1;  
        if ((uint)next < (uint)_deque._count)  
        {  
            _index = next;  
            _current = _deque._buffer[_deque.PhysicalIndex(_index)];  
            return true;  
        }  
        _index = _deque._count;  
        _current = default!;  
        return false;  
    }
```

```
    public void Reset()  
    {  
        if (_version != _deque._version)  
            throw new InvalidOperationException("Collection was modified  
during enumeration.");  
        _index = -1;  
        _current = default!;  
    }
```

```
    public void Dispose() { }
```

```
    private void EnsureCapacityForOneMore()  
    {  
        if (_count == Capacity)  
        {  
            int newCapacity = Capacity == 0 ? DefaultCapacity : Capacity * 2;  
            Resize(newCapacity);  
        }  
    }
```

```
    private void Resize(int newCapacity)  
    {  
        var newBuffer = new T[newCapacity];  
        if (_count > 0)  
        {  
            int rightSegment = Math.Min(_count, Capacity - _head);  
            Array.Copy(_buffer, _head, newBuffer, 0, rightSegment);  
            Array.Copy(_buffer, 0, newBuffer, rightSegment, _count -  
rightSegment);
```

```

    }
    _buffer = newBuffer;
    _head = 0;
    _tail = _count;
    // version не увеличиваем: логически коллекция не менялась для
клиентов
}

```

```

private int PhysicalIndex(int logicalIndex)
{
    int idx = _head + logicalIndex;
    if (idx >= Capacity) idx -= Capacity;
    return idx;
}

```

```

private int NextIndex(int idx) => (++idx == Capacity) ? 0 : idx;
private int PrevIndex(int idx) => (--idx < 0) ? Capacity - 1 : idx;

```

// Быстрый аналог

```

RuntimeHelpers.IsReferenceOrContainsReferences<T>()
// чтобы не тянуть System.Runtime.CompilerServices в примере
private static bool
RuntimeHelpersIsReferenceOrContainsReferences<TT>()
{
    return !typeof(TT).IsValueType ||

```

```

System.Runtime.CompilerServices.RuntimeHelpers.IsReferenceOrContainsReferences<TT>();
}
}
}

```

1. Потокбезопасная обертка с простым lock

```

using System;
using System.Collections;
using System.Collections.Generic;

namespace RingBufferLib
{
    // Обертка, делегирующая операции внутрь Deque<T> с синхронизацией
через lock.
    // Enumerator делает snapshot (копию) на момент начала итерации, чтобы
не лочить длительно.

```

```
public class ThreadSafeDeque<T> : IReadOnlyCollection<T>,
IEnumerable<T>
{
    private readonly Deque<T> _inner;
    private readonly object _gate = new();

    public ThreadSafeDeque(int capacity = 4)
    {
        _inner = new Deque<T>(capacity);
    }

    public int Count
    {
        get { lock (_gate) return _inner.Count; }
    }

    public int Capacity
    {
        get { lock (_gate) return _inner.Capacity; }
    }

    public void PushBack(T item)
    {
        lock (_gate) _inner.PushBack(item);
    }

    public void PushFront(T item)
    {
        lock (_gate) _inner.PushFront(item);
    }

    public T PopBack()
    {
        lock (_gate) return _inner.PopBack();
    }

    public T PopFront()
    {
        lock (_gate) return _inner.PopFront();
    }

    public bool TryPopFront(out T value)
    {
        lock (_gate) return _inner.TryPopFront(out value);
    }
}
```

```

public bool TryPopBack(out T value)
{
    lock (_gate) return _inner.TryPopBack(out value);
}

public T PeekFront()
{
    lock (_gate) return _inner.PeekFront();
}

public T PeekBack()
{
    lock (_gate) return _inner.PeekBack();
}

public void Clear()
{
    lock (_gate) _inner.Clear();
}

public bool Contains(T item, IEqualityComparer<T>? comparer = null)
{
    lock (_gate) return _inner.Contains(item, comparer);
}

public void CopyTo(T[] array, int arrayIndex)
{
    lock (_gate) _inner.CopyTo(array, arrayIndex);
}

public IEnumerator<T> GetEnumerator()
{
    // Snapshot во избежание длительных блокировок в foreach
    T[] snapshot;
    lock (_gate)
    {
        snapshot = new T[_inner.Count];
        _inner.CopyTo(snapshot, 0);
    }
    return ((IEnumerable<T>)snapshot).GetEnumerator();
}

```

System.Collections.IEnumerator

System.Collections.IEnumerable.GetEnumerator() => GetEnumerator();

```
}  
}
```

1. Тесты xUnit

```
using System;  
using System.Linq;  
using RingBufferLib;  
using Xunit;
```

```
namespace RingBufferTests
```

```
{
```

```
    public class DequeTests
```

```
    {
```

```
        [Fact]
```

```
        public void PushPopFrontBack_Basic()
```

```
        {
```

```
            var dq = new Deque<int>(2);
```

```
            dq.PushBack(1);
```

```
            dq.PushBack(2);
```

```
            dq.PushFront(0);
```

```
            Assert.Equal(3, dq.Count);
```

```
            Assert.Equal(0, dq.PeekFront());
```

```
            Assert.Equal(2, dq.PeekBack());
```

```
            Assert.Equal(0, dq.PopFront());
```

```
            Assert.Equal(2, dq.PopBack());
```

```
            Assert.Equal(1, dq.PopFront());
```

```
            Assert.True(dq.IsEmpty);
```

```
        }
```

```
        [Fact]
```

```
        public void AutoGrow_And_Indexer()
```

```
        {
```

```
            var dq = new Deque<int>(1);
```

```
            for (int i = 0; i < 100; i++) dq.PushBack(i);
```

```
            Assert.Equal(100, dq.Count);
```

```
            for (int i = 0; i < 100; i++) Assert.Equal(i, dq[i]);
```

```
            dq[50] = 999;
```

```
            Assert.Equal(999, dq[50]);
```

```
        }
```

```
        [Fact]
```

```
        public void WrapAround_Correctness()
```

```

{
    var dq = new Deque<int>(4);
    dq.PushBack(1);
    dq.PushBack(2);
    dq.PushBack(3);
    Assert.Equal(1, dq.PopFront());
    Assert.Equal(2, dq.PopFront());
    dq.PushBack(4);
    dq.PushBack(5);
    dq.PushFront(0);
    // Теперь кольцевание активно, проверим порядок:
    var arr = dq.ToArray();
    Assert.Equal(new[] { 0, 3, 4, 5 }, arr);
}

```

[Fact]

public void Enumerator_FailFast_OnModification()

```

{
    var dq = new Deque<int>();
    dq.PushBack(1);
    dq.PushBack(2);
    var e = dq.GetEnumerator();
    Assert.True(e.MoveNext());
    dq.PushBack(3);
    Assert.Throws<InvalidOperationException>(() => e.MoveNext());
}

```

[Fact]

public void ThreadSafeWrapper_Basic()

```

{
    var dq = new ThreadSafeDeque<int>(2);
    dq.PushBack(1);
    dq.PushFront(0);
    Assert.Equal(2, dq.Count);
    Assert.Equal(0, dq.PeekFront());
    Assert.Equal(1, dq.PeekBack());

    var list = dq.ToList();
    Assert.Equal(new[] { 0, 1 }, list);
}
}

```

internal static class DequeExtensions

```

{
    public static T[] ToArray<T>(this Deque<T> dq)

```

```

    {
        var arr = new T[dq.Count];
        dq.CopyTo(arr, 0);
        return arr;
    }
}
}

```

1. Бенчмарки BenchmarkDotNet

Сравним три сценария:

- **NonThreadSafe (Deque)** с одним продюсером и одним консюмером последовательно (без конкуренции) — базовая производительность.
- **ThreadSafeDeque** с lock.
- Обертка с **ReaderWriterLockSlim** для интереса.

```

using System;
using System.Threading;
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;
using RingBufferLib;

```

```

namespace RingBufferBench

```

```

{
    [MemoryDiagnoser]
    public class DequeBench
    {
        [Params(1024)]
        public int N;

        private Deque<int> _plain = null!;
        private ThreadSafeDeque<int> _locked = null!;
        private RWLockDeque<int> _rw = null!;

        [GlobalSetup]
        public void Setup()
        {
            _plain = new Deque<int>(N);
            _locked = new ThreadSafeDeque<int>(N);
            _rw = new RWLockDeque<int>(N);
        }

        [Benchmark(Baseline = true)]
        public int Plain_PushPop()
        {

```

```

    int sum = 0;
    for (int i = 0; i < N; i++) _plain.PushBack(i);
    for (int i = 0; i < N; i++) sum += _plain.PopFront();
    return sum;
}

```

[Benchmark]

```

public int Locked_PushPop()

```

```

{
    int sum = 0;
    for (int i = 0; i < N; i++) _locked.PushBack(i);
    for (int i = 0; i < N; i++) sum += _locked.PopFront();
    return sum;
}

```

[Benchmark]

```

public int RWLock_PushPop()

```

```

{
    int sum = 0;
    for (int i = 0; i < N; i++) _rw.PushBack(i);
    for (int i = 0; i < N; i++) sum += _rw.PopFront();
    return sum;
}
}

```

```

public class Program

```

```

{
    public static void Main(string[] args)
    {
        BenchmarkRunner.Run<DequeBench>();
    }
}

```

// Пример обертки на ReaderWriterLockSlim для сравнения.

// В данном паттерне преимущество RWLockSlim не очевидно, но полезно для эксперимента.

```

public class RWLockDeque<T>

```

```

{
    private readonly Deque<T> _inner;
    private readonly ReaderWriterLockSlim _lock =
new(LockRecursionPolicy.NoRecursion);

```

```

    public RWLockDeque(int capacity = 4)

```

```

    {
        _inner = new Deque<T>(capacity);
    }
}

```



```

    }

    public void PushBack(T item)
    {
        _lock.EnterWriteLock();
        try { _inner.PushBack(item); }
        finally { _lock.ExitWriteLock(); }
    }

    public void PushFront(T item)
    {
        _lock.EnterWriteLock();
        try { _inner.PushFront(item); }
        finally { _lock.ExitWriteLock(); }
    }

    public T PopFront()
    {
        _lock.EnterWriteLock();
        try { return _inner.PopFront(); }
        finally { _lock.ExitWriteLock(); }
    }

    public T PopBack()
    {
        _lock.EnterWriteLock();
        try { return _inner.PopBack(); }
        finally { _lock.ExitWriteLock(); }
    }

    public T PeekFront()
    {
        _lock.EnterReadLock();
        try { return _inner.PeekFront(); }
        finally { _lock.ExitReadLock(); }
    }

    public T PeekBack()
    {
        _lock.EnterReadLock();
        try { return _inner.PeekBack(); }
        finally { _lock.ExitReadLock(); }
    }
}
}

```

Как запускать

- **Создайте solution с тремя проектами:**
 - **RingBufferLib (Class Library)** — содержит **Deque**, **ThreadSafeDeque**.
 - **RingBufferTests (xUnit Test Project)**.
 - **RingBufferBench (Console App)** с **BenchmarkDotNet**.
- **Установите пакеты:**
 - **xUnit:** **xunit**, **xunit.runner.visualstudio**
 - **BenchmarkDotNet:** **BenchmarkDotNet**
- **Запустите тесты через Test Explorer.**
- **Для бенчмарков — конфигурация Release, x64, без отладчика.**

Заметки по дизайну

- **Версионность для Enumerator повышается при любых mutating-операциях. Resize не считает логической модификацией — стандартный подход, чтобы foreach не падал при простом расширении изнутри, но в нашем случае Resize происходит внутри Push/Pop, так что внешне версия и так изменится.**
- **CopyTo копирует логический порядок.**
- **ThreadSafe обертка делает snapshot для foreach, чтобы не держать lock в пользовательском коде.**
- **Для высоких нагрузок и contention lock чаще выигрывает у RWLockSlim, но бенчи покажут.**

Если нужно, добавлю:

- **TryPeekFront/Back, Remove(predicate), RemoveAt(logicalIndex).**
- **Пулинг массивов через ArrayPool<t>.**
- **Специализацию под value-type без IsReferenceOrContainsReferences вызова.**