# Технический отчет: Архитектура высокопроизводительных систем в Modern.NET (.NET 8)

## 1. Введение: Смена парадигмы управления памятью и структурами данных

Переход от разработки на Unity к построению высоконагруженных серверных систем на платформе Enterprise.NET (ASP.NET Core) требует фундаментальной перестройки инженерного мышления. В среде GameDev центральной абстракцией является "кадр" (Frame), и управление памятью подчиняется жестким временным ограничениям игрового цикла (Game Loop). Основная цель — избежать пауз Garbage Collector (GC Spikes), которые приводят к визуальным фризам. Это порождает специфические паттерны: отказ от LINQ в Update(), использование структур вместо классов для снижения давления на кучу, и ручное управление пулами объектов.1

В серверной разработке (Backend Engineering), особенно в контексте High-Load систем, парадигма смещается от минимизации времени кадра к максимизации пропускной способности (Throughput) и обеспечению низкой латентности (Latency) при конкурентном доступе тысяч потоков. Здесь коллекции перестают быть просто контейнерами для данных; они становятся критическими узлами инфраструктуры, определяющими масштабируемость системы. Ошибка в выборе структуры данных — например, использование Dictionary вместо FrozenDictionary для статических данных или наивная сортировка списка вместо PriorityQueue — может привести к деградации производительности всей системы под нагрузкой.

Данный отчет представляет собой исчерпывающий теоретический анализ тем "Пятого дня" четвертой недели обучения, посвященный новейшим оптимизациям, внедренным в.NET 6 и.NET 8. Мы детально рассмотрим три ключевых столпа современной производительности: неизменяемые "замороженные" коллекции (System.Collections.Frozen), алгоритмические очереди с приоритетом (PriorityQueue) и управление памятью через пулинг (ArrayPool<T>). Анализ проводится на уровне внутреннего устройства CLR (Common Language Runtime), работы JIT-компилятора и архитектуры процессорного кэша.

## 2. System.Collections.Frozen: Эволюция неизменяемости и оптимизация чтения

В экосистеме.NET долгое время существовала дихотомия между изменяемыми коллекциями (Dictionary<TKey, TValue>) и неизменяемыми (ImmutableDictionary<TKey, TValue>). С выходом.NET 8 в пространство имен System.Collections.Frozen были добавлены новые типы — FrozenDictionary и FrozenSet, которые решают специфическую задачу: обеспечение максимальной скорости чтения для данных, которые загружаются один раз и живут в течение всего жизненного цикла приложения.2

### 2.1. Проблема производительности ImmutableDictionary

Чтобы понять ценность FrozenDictionary, необходимо проанализировать архитектуру его предшественника — ImmutableDictionary. Классические неизменяемые коллекции в.NET реализованы на основе сбалансированных бинарных деревьев (обычно AVL-деревья). Это решение продиктовано необходимостью эффективной "модификации": при добавлении элемента в неизменяемый словарь создается новая версия дерева, которая переиспользует (structural sharing) большую часть узлов старого дерева.

Однако эта структура имеет критические недостатки для сценариев Read-Heavy:

1. **Алгоритмическая сложность:** Поиск в AVL-дереве имеет сложность $O(\log N)$, что значительно медленнее $O(1)$ в хеш-таблице.4
2. **Промахи кэша (Cache Misses):** Дерево состоит из множества узлов-объектов, разбросанных по куче (Managed Heap). Переход по ссылкам (node.Left, node.Right) вызывает частые промахи кэша процессора (Pointer Chasing), так как данные не лежат в памяти последовательно.
3. **Накладные расходы памяти:** Каждый элемент требует создания объекта-обертки (Node), что увеличивает нагрузку на GC и потребление памяти.5

### 2.2. Философия FrozenDictionary: Анализ при построении

FrozenDictionary<TKey, TValue> отказывается от идеи эффективной модификации в пользу абсолютной скорости чтения. Ключевая особенность этих коллекций — чрезвычайно дорогая фаза создания (ToFrozenDictionary()), в ходе которой происходит глубокий статистический анализ входных данных для выбора оптимальной внутренней структуры хранения.6

В отличие от обычного Dictionary, который использует универсальную стратегию разрешения коллизий, FrozenDictionary адаптирует свою внутреннюю архитектуру под конкретный набор ключей. В.NET 8 существует около 18 различных внутренних реализаций (стратегий), выбираемых динамически.6

#### Таблица 1. Сравнительный анализ характеристик словарей

| **Характеристика** | **Dictionary<T>** | **ImmutableDictionary<T>** | **FrozenDictionary<T>** |
| --- | --- | --- | --- |
| **Структура данных** | Хеш-таблица (Chaining) | AVL-дерево | Специализированная хеш-таблица |
| **Сложность чтения** | $O(1)$ (в среднем) | $O(\log N)$ | $O(1)$ (оптимизировано) |
| **Сложность записи** | $O(1)$ | $O(\log N)$ | Не поддерживается (Immutable) |
| **Thread Safety** | Нет (требует lock) | Да (неявная) | Да (полная, без локов) |
| **Стоимость создания** | Низкая | Высокая | Очень высокая (Анализ ключей) |
| **Overhead памяти** | Средний | Высокий (Узлы) | Минимальный (Массивы) |
| **Сценарий использования** | General Purpose | Функциональный стиль | Configs, Static Lookups |

### 2.3. Внутренние стратегии оптимизации (Deep Dive)

Процесс создания FrozenDictionary включает в себя перебор стратегий для минимизации коллизий и оптимизации вычисления хеша. Рассмотрим наиболее интересные реализации, скрытые внутри CLR.

#### 2.3.1. LengthBucketsFrozenDictionary: Сортировка по длине

Если анализ показывает, что строковые ключи имеют разнообразную длину (например, набор команд: "GET", "POST", "DELETE", "OPTIONS"), рантайм выбирает стратегию LengthBucketsFrozenDictionary.6

Механизм работы:

Вместо вычисления хеш-кода всей строки сразу, алгоритм сначала использует длину строки как дискриминатор.

1. Входящий запрос TryGetValue("POST") проверяет длину: 4 символа.
2. Словарь обращается к специальному "бакету" для строк длины 4.
3. Только внутри этого малого подмножества происходит сравнение.

Это фактически устраняет коллизии между ключами разной длины без затрат процессорного времени на чтение содержимого строки, что является существенной оптимизацией по сравнению с обычным GetHashCode, который обязан прочитать символы строки.

#### 2.3.2. OrdinalStringFrozenDictionary и оптимизация подстрок

Для наборов ключей, имеющих общие префиксы (например, URL-пути: /api/v1/users, /api/v1/orders, /api/v1/products), стандартное хеширование неэффективно, так как начало строк идентично, и энтропия находится в конце.

FrozenDictionary анализирует ключи и находит уникальные подстроки, обеспечивающие максимальную вариативность. Внутренний компаратор настраивается на хеширование *только* этих отличительных участков (например, символов с 8 по 15). Это позволяет избежать обработки избыточных данных и значительно ускоряет lookup.6

Более того, если набор ключей состоит только из ASCII-символов, выбирается оптимизированная реализация, игнорирующая проверки на Unicode-нормализацию, что дает дополнительный прирост скорости.8

#### 2.3.3. Int32FrozenDictionary: Идеальное хеширование

Для целочисленных ключей (int) ситуация уникальна. Значение int само по себе является своим хеш-кодом. Int32FrozenDictionary использует это свойство, исключая вызов GetHashCode() и виртуальные вызовы компараторов. Ключ используется как прямой индекс (или через простую модульную арифметику) для доступа к массиву значений. Это приводит к приросту производительности чтения на 30-40% по сравнению с обычным Dictionary<int, V>.6

### 2.4. Архитектурные рекомендации и антипаттерны

Использование FrozenDictionary требует дисциплины. Это не замена Dictionary для любых задач.

Антипаттерн: Частое создание.

Попытка использовать ToFrozenDictionary() внутри горячего пути (например, при обработке каждого HTTP-запроса) приведет к катастрофическому падению производительности. Время, затрачиваемое на статистический анализ ключей и построение оптимизированных таблиц, на порядки превышает выигрыш от быстрого чтения.4

Рекомендация: Singleton конфигурации.

Идеальный сценарий — статические справочники, конфигурации, таблицы маршрутизации, карты локализации, загружаемые при старте приложения (Startup). В этом случае высокая стоимость инициализации амортизируется за время жизни процесса, а выигрыш в скорости доступа и потокобезопасности (отсутствие блокировок при чтении) дает ощутимый эффект масштабирования.3

## 3. System.Collections.Generic.PriorityQueue: Алгоритмическая эффективность

В задачах серверной разработки часто возникает потребность обрабатывать элементы не в порядке их поступления (FIFO), а в зависимости от их важности. Примеры включают планировщики задач (Task Scheduler), системы лимитирования запросов (Rate Limiting) и алгоритмы поиска путей (для игровых серверов). До.NET 6 разработчики были вынуждены использовать неэффективные решения вроде сортировки List<T> ($O(N \log N)$) или SortedDictionary, который имеет большие накладные расходы по памяти.12

Введение PriorityQueue<TElement, TPriority> в.NET 6 дало разработчикам высокопроизводительную реализацию структуры данных "Куча" (Heap), оптимизированную под современные процессоры.12

### 3.1. Теория кучи (Heap Theory) и выбор арности (Arity)

Очередь с приоритетом реализуется на базе структуры данных Min-Heap (или Max-Heap). Концептуально это полное дерево, где каждый родительский узел имеет приоритет выше (значение меньше), чем его дети. Однако физически эта структура укладывается в массив, что обеспечивает отсутствие накладных расходов на объекты-узлы и ссылочную целостность.14

Ключевым архитектурным решением в.NET реализации является использование **четвертичной кучи (Quaternary Min-Heap, d=4)**, в отличие от классической бинарной кучи (d=2), изучаемой в академических курсах.12

#### 3.1.1. Влияние кэш-линий процессора (Cache Locality)

Выбор $d=4$ обусловлен физикой работы кэш-памяти CPU. Современные процессоры загружают данные из RAM блоками по 64 байта (Cache Line).

* **Бинарная куча (d=2):** Индексы потомков узла $i$ находятся по адресам $2i+1$ и $2i+2$. С ростом глубины дерева, расстояния между родителями и детьми в массиве растут экспоненциально, что приводит к тому, что при переходе от родителя к ребенку происходит промах кэша (Cache Miss).16
* **Четвертичная куча (d=4):** Узел имеет 4 потомка. Это уменьшает высоту дерева в $ \log\_4 N $ раз по сравнению с $ \log\_2 N $. Более плоская структура дерева означает меньше переходов по памяти. Кроме того, 4 потомка (особенно если это int или ссылки) с высокой вероятностью попадают в одну или соседние кэш-линии, что позволяет процессору загружать их одной операцией (Spatial Locality).14

### 3.2. Алгоритмическая сложность и компромиссы

Переход на $d=4$ меняет константы в формулах сложности, но сохраняет логарифмический класс:

* **Enqueue (Вставка):** $O(\log\_4 N)$. Вставка требует "всплытия" (bubble-up) элемента. Поскольку дерево ниже, операций сравнения с родителями требуется меньше. Это делает вставку в.NET PriorityQueue чрезвычайно быстрой.17
* **Dequeue (Извлечение):** $O(4 \cdot \log\_4 N)$. Удаление минимума требует перемещения последнего элемента в корень и его "погружения" (sift-down). На каждом уровне необходимо сравнить элемент со всеми $d$ детьми, чтобы найти наименьшего. При $d=4$ количество сравнений на уровне растет, но количество уровней падает. Бенчмарки показывают, что выигрыш от локальности кэша перевешивает затраты на дополнительные сравнения для объемов данных, типичных для Enterprise-систем.14
* **Peek (Просмотр):** $O(1)$. Минимальный элемент всегда лежит по индексу 0.

### 3.3. Практическое применение: Rate Limiter и планировщики

Одной из практических задач, упомянутых в плане обучения, является реализация **Distributed Rate Limiter** с использованием алгоритма Sliding Window Log.

Проблема: Необходимо хранить временные метки (Timestamps) запросов за последнюю минуту и удалять устаревшие.

Наивное решение: Использовать List<DateTime>. Удаление устаревших элементов из начала списка (RemoveAt(0)) вызывает сдвиг всего массива — операция $O(N)$.

Решение на PriorityQueue:

1. Храним запросы в PriorityQueue<RequestData, DateTime>, где приоритет — время запроса.
2. Для очистки окна проверяем Peek(). Если время Peek() < (Now - WindowSize), выполняем Dequeue().
3. Эта операция выполняется за логарифмическое время, что позволяет обрабатывать миллионы запросов в секунду без блокировки CPU на перекладывании массивов памяти.19

Пример реализации "Активной экспирации":

В высоконагруженных системах (как Azure Job Scheduler) использование PriorityQueue вместо сканирования списков позволило сократить задержки планирования с секунд до миллисекунд и снизить потребление CPU на порядок.14

## 4. Advanced Memory Management: ArrayPool<T> и борьба с LOH

При переходе к бэкенд-разработке одной из самых сложных концепций для Unity-разработчика становится управление памятью в контексте Server GC. В Unity основной враг — это аллокации в кадре. В ASP.NET Core основной враг — фрагментация **Large Object Heap (LOH)** и паузы второго поколения (Gen 2 GC).1

### 4.1. Физика LOH и фрагментация

В CLR любой объект размером >= 85,000 байт (для массивов double этот порог может быть иным) попадает в LOH. В отличие от SOH (Small Object Heap), LOH не подвергается компактификации (дефрагментации) при обычной сборке мусора, так как копирование больших объемов памяти слишком дорого.

Постоянное создание и уничтожение буферов (например, для чтения тела HTTP-запроса или обработки изображений) приводит к тому, что LOH наполняется "дырами" — свободной памятью, разбитой на мелкие куски, в которые невозможно поместить новый большой объект. Это неизбежно ведет к OutOfMemoryException даже при наличии свободной RAM и провоцирует частые и долгие сборки Gen 2.22

### 4.2. Архитектура ArrayPool<T>.Shared

Класс System.Buffers.ArrayPool<T> предоставляет механизм переиспользования массивов. Статическое свойство Shared дает доступ к глобальному пулу, который имеет сложную внутреннюю структуру, оптимизированную для конкурентного доступа (lock-free на горячих путях).24

#### 4.2.1. Иерархия хранения: TLS и Глобальные партиции

ArrayPool<T>.Shared использует двухуровневую систему кэширования для минимизации синхронизации потоков:

1. **Thread-Local Storage (TLS):** Каждый поток имеет свой персональный "карман" для массива.
   * Когда поток вызывает Rent(), пул сначала проверяет TLS. Если там есть массив подходящего размера, он возвращается мгновенно без использования каких-либо блокировок (lock или Interlocked). Это обеспечивает фантастическую производительность для сценария "взял-вернул в том же потоке".26
   * Однако TLS обычно хранит только один массив каждого размера.
2. **Глобальные партиции (Global Partitions):** Если в TLS нет массива, запрос уходит в глобальное хранилище.
   * Чтобы избежать единого глобального лока (Bottleneck), глобальное хранилище разделено на партиции (Buckets), количество которых зависит от количества ядер CPU.
   * Доступ к партициям защищен легковесными спин-локами (SpinLock), что быстрее, чем полноценные мониторы ОС, но все же требует синхронизации.26

#### 4.2.2. Стратегия "Bucketing" (Корзины)

Пул не хранит массивы произвольного размера. Он использует систему корзин, основанную на степенях двойки.

* Вызов Rent(1000) вернет массив размером 1024.
* Вызов Rent(1025) вернет массив размером 2048.
* **Важно:** Свойство .Length арендованного массива всегда больше или равно запрошенному. Разработчик обязан использовать передаваемый размер (или Span), а не длину массива, чтобы не прочитать "мусорные" данные из хвоста буфера.28

### 4.3. Безопасность и "Use-After-Return"

Главный риск использования ArrayPool — это ручное управление памятью в управляемой среде. Самая опасная уязвимость — **Use-After-Return**.30

**Сценарий атаки/ошибки:**

1. Сервис А арендует буфер, пишет в него конфиденциальные данные пользователя X.
2. Сервис А возвращает буфер в пул (Return), но *осталяет ссылку* на него в каком-то поле или замыкании.
3. Сервис Б арендует этот же буфер для пользователя Y.
4. Сервис А (ошибочно) продолжает писать по старой ссылке, перезаписывая данные пользователя Y или читая их.

Это классическая уязвимость повреждения памяти, которая в C# может привести к утечке данных между сессиями. Поскольку CLR не обнуляет массивы при возврате (если не передан флаг clearArray=true, который убивает производительность), ответственность лежит полностью на разработчике.31

Паттерн защиты (Defensive Coding):

Всегда сбрасывайте ссылку на массив в null сразу после возврата в пул. Использование оберток IDisposable (struct wrappers) помогает автоматизировать возврат, но не защищает от копирования ссылки.32

#### 4.3.3. Политика очистки (Trimming)

ArrayPool обладает интеллектом: он следит за давлением на память (Memory Pressure). Если система сообщает о нехватке памяти, пул перестает принимать массивы обратно (просто отбрасывает их, отдавая GC) и очищает свои внутренние кэши. Это предотвращает ситуацию, когда пул удерживает гигабайты памяти, которые нужны другим частям приложения.33 Однако, "утечка" арендованных массивов (Rent без Return) приведет к тому, что пул будет постоянно аллоцировать новые массивы, сводя на нет всю оптимизацию.35

## 5. Синтез: Применение в реальной архитектуре

Объединяя изученные концепции, мы формируем архитектурный профиль современного высоконагруженного.NET сервиса.

### 5.1. Сценарий: In-Memory Data Grid (Mini-Redis)

Рассмотрим задачу создания кэширующего микросервиса (из практического задания), обрабатывающего TCP-запросы.

1. **Обработка команд (Command Parsing):**
   * Для чтения байтов из сокета используется ArrayPool<byte>.Shared. Это предотвращает LOH-фрагментацию при обработке больших пейлоадов (например, SET key value с 1MB данных).
   * Полученный буфер оборачивается в ReadOnlySpan<byte> для парсинга без аллокаций строк (Zero-Allocation Parsing).
2. **Маршрутизация команд (Routing):**
   * Словарь доступных команд ("SET", "GET", "DEL") и их обработчиков хранится в FrozenDictionary<string, ICommand>.
   * Инициализация происходит в Startup.cs.
   * При обработке запроса поиск обработчика происходит за $O(1)$ без блокировок, используя оптимизацию по длине строк или хеширование подстрок.
3. **Управление TTL (Time-To-Live):**
   * Для реализации активной экспирации ключей используется фоновый поток с PriorityQueue<string, long>, где приоритет — время истечения срока действия (Unix Timestamp).
   * Цикл очистки спит до времени Peek(). Как только время наступает, он пробуждается, удаляет ключи из основного хранилища и извлекает их из очереди. Это эффективнее, чем таймер на каждый ключ или полный перебор (Scan) словаря.

### 5.2. Итоговые рекомендации для архитектора

1. **Статика — в Frozen:** Любые справочники, конфигурации и таблицы, которые не меняются после старта, *должны* быть FrozenDictionary. Это дешевый способ снизить CPU Usage на lookup-операциях.
2. **Сортировка — в PriorityQueue:** Никогда не используйте List.Sort() или OrderBy() для динамических задач выбора приоритетного элемента. Используйте PriorityQueue для обеспечения предсказуемой задержки (Latency).
3. **Буферы — в ArrayPool:** Любой массив byte, char или int размером более 1-2 КБ, живущий короткое время (scope запроса), должен браться из пула.
4. **Профилирование прежде всего:** Не применяйте FrozenDictionary для короткоживущих данных (слишком дорогое создание). Не применяйте ArrayPool для мелких массивов (накладные расходы на TLS/SpinLock могут превысить выигрыш от GC). Всегда верифицируйте гипотезы через BenchmarkDotNet.

Этот набор инструментов и знаний переводит разработчика из категории "Consumer of Framework" в категорию "Performance Engineer", способного проектировать системы, выдерживающие нагрузки уровня Enterprise.

### Таблица 2. Сводная таблица сложности операций

| **Структура данных** | **Операция** | **Сложность** | **Комментарий** |
| --- | --- | --- | --- |
| **Dictionary<K,V>** | Lookup | $O(1)$ | Зависит от качества хеш-функции. Возможна деградация до $O(N)$. |
| **FrozenDictionary<K,V>** | Lookup | $O(1)$ | Оптимизированные константы, защита от коллизий. |
| **PriorityQueue<E,P>** | Enqueue | $O(\log\_4 N)$ | Очень быстрая вставка за счет низкой высоты дерева. |
| **PriorityQueue<E,P>** | Dequeue | $O(4 \log\_4 N)$ | Чуть больше сравнений, но высокая локальность кэша. |
| **ArrayPool<T>** | Rent | $O(1)$ | В случае попадания в TLS (fast path). |
| **ArrayPool<T>** | Return | $O(1)$ | Возврат в TLS или глобальный бакет. |

## 6. Библиография и источники

В отчете использованы данные из официальной документации Microsoft, исходного кода runtime.NET (GitHub), технических блогов инженеров производительности и результатов бенчмарков.

* **Frozen Collections:** 2
* **PriorityQueue:** 12
* **ArrayPool:** 21
* **General Context:** 1

#### Источники

1. План обучения: Архитектура коллекций и кэширование
2. Frozen Collections in .NET 8 - Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/c-sharp-programming/frozen-collections-in-net-8-055b007587d0>
3. .NET 8 — FrozenDictionary performance | code-corner.dev, дата последнего обращения: декабря 3, 2025, <https://code-corner.dev/2023/11/08/NET-8-%E2%80%94-FrozenDictionary-performance/>
4. Cloned Dictionary vs. Immutable Dictionary vs. Frozen Dictionary in High Traffic Systems - RavenDB NoSQL Database, дата последнего обращения: декабря 3, 2025, <https://ravendb.net/articles/cloned-dictionary-vs-immutable-dictionary-vs-frozen-dictionary-in-high-traffic-systems>
5. c# - Immutable Dictionary Vs Dictionary Vs C5 - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/16612936/immutable-dictionary-vs-dictionary-vs-c5>
6. FrozenDictionary under the hood - alexeyfv, дата последнего обращения: декабря 3, 2025, <https://www.alexeyfv.xyz/en/post/2024-08-22-frozen-dictionary/>
7. Hash collision counting in FrozenDictionary - alexeyfv, дата последнего обращения: декабря 3, 2025, <https://www.alexeyfv.xyz/en/post/2024-09-09-frozen-dictionary-collisions-count/>
8. What are some specific optimizations of frozen collections vs immutable / readonly? - Reddit, дата последнего обращения: декабря 3, 2025, <https://www.reddit.com/r/dotnet/comments/17xoyj6/what_are_some_specific_optimizations_of_frozen/>
9. What is the usecase for .net8's FrozenDictionary and FrozenSet? : r/csharp - Reddit, дата последнего обращения: декабря 3, 2025, <https://www.reddit.com/r/csharp/comments/1813gwm/what_is_the_usecase_for_net8s_frozendictionary/>
10. FrozenDictionary
11. Frozen collections in .NET 8 : r/dotnet - Reddit, дата последнего обращения: декабря 3, 2025, <https://www.reddit.com/r/dotnet/comments/z7kpd4/frozen_collections_in_net_8/>
12. PriorityQueue
13. Priority Queue implementation using a Heap, should I do it through inheritance?, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/74435076/priority-queue-implementation-using-a-heap-should-i-do-it-through-inheritance>
14. Priority Queues in .NET: Our Jump from Naive Implementation to Binary Heap | by Abhishek Joshi | CodeToDeploy | Nov, 2025 | Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/codetodeploy/priority-queues-in-net-our-jump-from-naive-implementation-to-binary-heap-35ec198c1be6>
15. Introduction to Priority Queue - GeeksforGeeks, дата последнего обращения: декабря 3, 2025, <https://www.geeksforgeeks.org/dsa/priority-queue-set-1-introduction/>
16. Why is Binary Heap is better than BST for Priority Queue? : r/algorithms - Reddit, дата последнего обращения: декабря 3, 2025, <https://www.reddit.com/r/algorithms/comments/fmvnke/why_is_binary_heap_is_better_than_bst_for/>
17. c# - What is the time complexity for PriorityQueue added in .NET 6 - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/74180472/what-is-the-time-complexity-for-priorityqueue-added-in-net-6>
18. Priority Queues - Algorithms, 4th Edition, дата последнего обращения: декабря 3, 2025, <https://algs4.cs.princeton.edu/24pq/>
19. Unlocking the Power of Heaps and Priority Queues in C#: From Basics to LeetCode Mastery, дата последнего обращения: декабря 3, 2025, <https://medium.com/@hanxuyang0826/unlocking-the-power-of-heaps-and-priority-queues-in-c-from-basics-to-leetcode-mastery-df00264ff3e5>
20. System Design: Sliding window based Rate Limiter - Codementor, дата последнего обращения: декабря 3, 2025, <https://www.codementor.io/@arpitbhayani/system-design-sliding-window-based-rate-limiter-157x7sburi>
21. Deep Dive into .NET Large Object Heap (LOH) and Practical Use of ArrayPool
22. Memory Optimization With ArrayPool in C# - Code Maze, дата последнего обращения: декабря 3, 2025, <https://code-maze.com/csharp-arraypool-memory-optimization/>
23. Large Arrays, and LOH Fragmentation. What is the accepted convention? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/2402282/large-arrays-and-loh-fragmentation-what-is-the-accepted-convention>
24. NET35\_ArrayPool.cs - GitHub Gist, дата последнего обращения: декабря 3, 2025, <https://gist.github.com/neuecc/aa7154c5bed38a21b8e9a66dc82482b7>
25. ArrayPool
26. The BIG performance difference between ArrayPools in .NET | by Eugene Peshkov | Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@epeshk/the-big-performance-difference-between-arraypools-in-net-b25c9fc5e31d>
27. Investigate locked-free implementations of ArrayPool · Issue #7013 · dotnet/runtime - GitHub, дата последнего обращения: декабря 3, 2025, <https://github.com/dotnet/runtime/issues/7013>
28. ArrayPool create method in C# - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/62584548/arraypool-create-method-in-c-sharp>
29. ArrayPool
30. [API Proposal]: Allow replace ArrayPool
31. ArrayPool
32. How to return array to ArrayPool when it was rented by inner function? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/78250438/how-to-return-array-to-arraypool-when-it-was-rented-by-inner-function>
33. ArrayPool
34. ArrayPool trimming does not respect GCHardLimit · Issue #853 · dotnet/runtime - GitHub, дата последнего обращения: декабря 3, 2025, <https://github.com/dotnet/runtime/issues/853>
35. Mastering ArrayPool
36. Priority Queue | Baeldung on Computer Science, дата последнего обращения: декабря 3, 2025, <https://www.baeldung.com/cs/priority-queue>
37. ArrayPool: The most underused memory optimization in .NET | by Vladamisici - Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@vladamisici1/arraypool-the-most-underused-memory-optimization-in-net-8c47f5dffbbd>
38. Is ArrayPool