# Архитектура коллекций и управление памятью в Enterprise.NET: Глубокий анализ теории второго дня

## 1. Введение: Смена парадигмы от алгоритмической абстракции к архитектурной реальности

Переход квалифицированного Unity-разработчика в сферу Enterprise-бэкенда на платформе.NET требует не просто изучения новых библиотек, а фундаментальной перестройки ментальной модели взаимодействия кода с аппаратным обеспечением. В среде GameDev (Unity) основной паттерн управления памятью диктуется жестким временным бюджетом кадра (16.6 мс для 60 FPS). Разработчики привыкают к стратегии избегания аллокаций любой ценой ("Zero Allocation in Update") и часто используют структуры данных, такие как пулы массивов с ручной индексацией, чтобы предотвратить скачки времени работы Garbage Collector (GC Spikes), которые приводят к визуальным "фризам".

Однако в среде высоконагруженного.NET Backend (ASP.NET Core) парадигма кардинально меняется. Здесь мы оперируем понятиями пропускной способности (Throughput), масштабируемости (Scalability) и конкурентного доступа (Concurrency). Ошибочный выбор между List<T> и LinkedList<T>, основанный на устаревших представлениях о сложности $O(1)$ vs $O(n)$, может привести к катастрофической деградации производительности всей системы под нагрузкой. Теория второго дня учебного плана, посвященная иерархии памяти, локальности и спискам, является поворотным моментом, где абстрактная теория алгоритмов сталкивается с физической реальностью современных суперскалярных процессоров.

Данный аналитический отчет представляет собой исчерпывающее техническое исследование теории второго дня. Мы отойдем от поверхностного использования API и спустимся на уровень управления кэш-линиями процессора, конвейеризации инструкций и внутренней реализации CLR (Common Language Runtime). Мы докажем, почему классическое утверждение "LinkedList лучше для вставок" является мифом в контексте современного железа, и детально разберем механизмы Span<T> и ref struct, которые позволяют достичь производительности уровня C++ в безопасном управляемом коде.

## 2. Физика вычислений: Иерархия памяти и "Стена Памяти"

Чтобы понять, почему List<T> часто превосходит LinkedList<T> вопреки теоретической алгоритмической сложности, необходимо детально рассмотреть субстрат, на котором выполняется код. Центральный процессор (CPU) не взаимодействует с оперативной памятью (RAM) напрямую побайтово, как это может представляться в высокоуровневой абстракции "кучи". Он работает через сложную иерархию кэшей, созданную для маскировки колоссальной латентности доступа к данным.

### 2.1 Пропасть латентности: Такты CPU против Доступа к RAM

Современные процессоры серверного класса (Intel Xeon, AMD EPYC) работают на частотах в диапазоне 3.0–5.0 ГГц. Это означает, что один тактовый цикл (cycle) занимает приблизительно 0.2–0.3 наносекунды. В рамках одного такта современное ядро, использующее суперскалярную архитектуру, способно декодировать и отправить на исполнение несколько инструкций (обычно 4–6). Однако скорость света и электрическая емкость накладывают жесткие физические ограничения на скорость передачи данных от модулей DIMM к кристаллу процессора.

Разрыв между скоростью выполнения инструкций процессором и скоростью доставки данных из основной памяти называется "Стеной Памяти" (Memory Wall). Анализ латентности различных уровней иерархии памяти в современных системах (данные актуальны для архитектур 2024-2025 годов) демонстрирует масштаб проблемы:

| **Уровень памяти** | **Типичный размер** | **Латентность (Такты CPU)** | **Латентность (Время)** | **Аналогия (Масштабированная)** |
| --- | --- | --- | --- | --- |
| **Регистры (Registers)** | < 1 КБ | 0 циклов | Мгновенно | Данные в ваших руках. |
| **L1 Cache (Data)** | 32 КБ – 48 КБ | 4 – 5 циклов | ~1 нс | Взять книгу со стола перед вами. |
| **L2 Cache** | 256 КБ – 2 МБ | 12 – 14 циклов | ~3 – 5 нс | Взять книгу с полки в той же комнате. |
| **L3 Cache (LLC)** | 8 МБ – 128 МБ+ | 40 – 60 циклов | ~10 – 20 нс | Пройти в коридор за книгой. |
| **Main Memory (RAM)** | 16 ГБ – 1 ТБ+ | **200 – 350+ циклов** | **60 – 100 нс** | Поездка в библиотеку на другой конец города. |

**Аналитический инсайт:** Критически важно осознать масштаб этой диспропорции. Одиночный промах кэша (Cache Miss), требующий обращения к RAM, заставляет процессор простаивать в течение 200–400 тактов.2 Если процессор способен выполнять 4 инструкции за такт, то за время ожидания данных из памяти теряется возможность выполнить от 800 до 1600 полезных инструкций.

Это фундаментально меняет подход к оптимизации. Алгоритм, который выполняет в 10 раз больше инструкций (например, сдвиг массива при вставке), но работает исключительно в кэше L1, будет выполняться значительно быстрее, чем алгоритм с минимальным числом инструкций (переброс ссылок в связном списке), который постоянно провоцирует промахи кэша (Cache Misses). В высоконагруженных системах "подсчет кэш-промахов" становится важнее "подсчета инструкций" (Big O).

### 2.2 Кэш-линии и Пространственная Локальность (Spatial Locality)

Контроллер памяти не извлекает данные побайтово. Обмен данными между RAM и кэшами процессора происходит блоками фиксированного размера, называемыми **Кэш-линиями (Cache Lines)**. В подавляющем большинстве современных архитектур (x86-64, ARM64) размер кэш-линии составляет **64 байта**.3

Этот архитектурный факт лежит в основе преимущества массивов:

* **Пространственная локальность:** Это принцип, согласно которому, если программа обращается к определенному адресу памяти, высока вероятность, что в ближайшее время она обратится к соседним адресам.
* **Эффект массива:** Когда код запрашивает первый элемент массива int (array), процессор загружает в кэш L1 не только 4 байта этого числа, но и всю 64-байтовую линию. Это означает, что следующие 15 элементов массива (array...array) уже находятся в самом быстром кэше. Доступ к ним происходит за 4 такта (L1 Hit).
* **Фрагментация списка:** В противоположность этому, узлы LinkedList<T> являются ссылочными типами (классами), размещаемыми в куче (Heap). В долгоживущем приложении куча фрагментируется. Узел A может находиться по адресу 0x1000, а узел B (A.Next) — по адресу 0x50000. Эти адреса попадают в разные кэш-линии. При переходе от A к B процессор вынужден загружать новую кэш-линию. Более того, из 64 загруженных байт полезными оказываются только данные узла (например, одно число), а остальные байты содержат служебные заголовки или соседние, не связанные объекты. Это называется низкой эффективностью использования кэша.5

### 2.3 Аппаратный префетчинг (Hardware Prefetching)

Современные процессоры не просто пассивно ждут запросов данных. Они оснащены сложными блоками **Аппаратного Префетчинга** (Hardware Prefetchers). Эти блоки анализируют паттерны доступа к памяти в реальном времени.

Если префетчер обнаруживает линейный паттерн доступа (Stride Pattern), например, обращение к адресам $X$, $X+64$, $X+128$, он начинает спекулятивно загружать следующие кэш-линии из RAM в L2 и L1 еще до того, как исполняемый код явно запросит эти данные.8

* **Массивы и List<T>:** Являются идеальной структурой данных для префетчера. Паттерн доступа к памяти при итерации строго последователен. Процессор может "стримить" данные из памяти с максимальной пропускной способностью шины.
* **Связные списки:** Паттерн доступа зависит от данных. Адрес следующего узла хранится внутри текущего узла. Процессор не может узнать адрес Next, пока не загрузит текущий узел. Это создает зависимость по данным, которая ломает логику стандартных страйдовых префетчеров, приводя к проблеме "Pointer Chasing" (гонка за указателями).10

## 3. Глубокое погружение: Архитектурный крах LinkedList

Учебный план второго дня ставит задачу сравнить List и LinkedList. В контексте Enterprise-разработки это сравнение часто оказывается разгромным для связных списков. Рассмотрим микроархитектурные причины этого явления.

### 3.1 Pointer Chasing и Цепочки Зависимостей

Термин "Pointer Chasing" описывает процесс обхода структуры данных, где адрес следующего элемента можно узнать только после прочтения текущего.

Типичный цикл обхода списка:

C#

while (current!= null) {  
 DoWork(current.Value);  
 current = current.Next; // <-- Критическая зависимость  
}

Это создает жесткую зависимость **Read-After-Write (RAW)**, или истинную зависимость по данным (True Data Dependency).10

#### Reorder Buffer (ROB) и остановка конвейера

Современные процессоры используют внеочередное исполнение (**Out-of-Order Execution, OoO**). Они имеют буфер инструкций, называемый **Reorder Buffer (ROB)**, который позволяет процессору "заглядывать вперед" и выполнять инструкции, не зависящие от текущей долгой операции (например, загрузки из памяти).

Однако при обходе связного списка возникает фундаментальная проблема:

1. **Сериализация доступа:** Загрузка узла $N+1$ не может начаться, пока не завершится загрузка узла $N$. Если узел $N$ отсутствует в кэше, процессор ждет ~300 тактов. В это время он не может инициировать загрузку $N+1$, так как не знает адреса. Запросы к памяти выстраиваются в строгую очередь, делая параллелизм уровня памяти (Memory Level Parallelism, MLP) невозможным.8
2. **Насыщение ROB:** Пока процессор ждет данные, он пытается найти другую полезную работу в окне инструкций. В узком цикле обхода списка (foreach) независимых инструкций практически нет. Все инструкции зависят от загружаемого узла. В результате ROB заполняется зависимыми инструкциями, и диспетчер перестает выдавать новые задачи. Конвейер останавливается (Pipeline Stall). Инструкционный параллелизм (ILP) падает до нуля.13

Исследования показывают, что при Pointer Chasing процессор может тратить до 90% времени в состоянии простоя (Stall), ожидая данных из памяти.16

### 3.2 Накладные расходы на память и давление на GC

В.NET LinkedList<T> реализован как двусвязный список. Каждый элемент обернут в класс LinkedListNode<T>. Рассмотрим раскладку памяти (Memory Layout) для 64-битной архитектуры.

**Структура LinkedListNode<T> в памяти:**

* **Object Header:** 8 байт (SyncBlockIndex и др.).
* **MethodTable Pointer:** 8 байт (указатель на тип).
* **Next Pointer:** 8 байт (ссылка на следующий узел).
* **Previous Pointer:** 8 байт (ссылка на предыдущий узел).
* **List Reference:** 8 байт (ссылка на контейнер LinkedList).
* **Value (T):** Размер типа T (минимум 4 байта для int + выравнивание).
* **Padding:** Выравнивание до граница 8 байт.

Для хранения одного значения int (4 байта) узел списка потребляет 48 байт в куче (или больше, в зависимости от версии CLR и выравнивания).5

Для сравнения, List<int> хранит данные в массиве int, где на каждый элемент приходится ровно 4 байта (плюс константные накладные расходы на сам объект массива).

**Влияние на сборку мусора (Garbage Collection):**

1. **Плотность данных:** List хранит данные в 12 раз плотнее. Это означает, что в одну кэш-линию (64 байта) помещается 16 элементов массива, но едва помещается один узел списка.
2. **Обход графа объектов:** Во время фазы маркировки (Mark Phase) GC должен обойти все живые объекты.
   * List<int> на 1 миллион элементов — это **один** объект для GC (массив). Сборщик мусора просто помечает массив как живой. Внутренности (числа) не являются ссылками, их сканировать не нужно.
   * LinkedList<int> на 1 миллион элементов — это **1 миллион и один** объект. Сборщик мусора должен посетить каждый узел, прочитать ссылки Next/Previous и перейти по ним. Это создает колоссальную нагрузку на процессор во время сборки мусора, увеличивая время пауз (Stop-the-World).19

## 4. Миф об O(1) вставке: Почему List побеждает

Ключевой аспект второго дня — развенчание мифа о том, что LinkedList всегда лучше для вставок в середину. Теоретически, вставка в список — это $O(1)$ (при наличии ссылки на узел), а в массив — $O(n)$. Однако на практике List<T>.Insert часто выигрывает.

### 4.1 Сила memmove и SIMD-оптимизаций

Когда мы вставляем элемент в середину List<T>, все последующие элементы должны быть сдвинуты вправо. В ранних версиях языков это делалось поэлементным копированием в цикле. В современном.NET (Core 3.1, 5, 6, 8) метод List<T>.Insert использует оптимизированный метод Array.Copy, который внутри вызывает нативную функцию memmove.21

Роль SIMD (Single Instruction, Multiple Data):

Современные реализации memmove в стандартной библиотеке C и в CLR используют векторные инструкции процессора (SSE2, AVX, AVX-512).

* **AVX2:** Позволяет копировать 32 байта (8 целых чисел) за одну инструкцию.
* **AVX-512:** Позволяет копировать 64 байта (16 целых чисел) за одну инструкцию.23

Это означает, что "дорогостоящий" сдвиг массива $O(n)$ выполняется с использованием всей ширины шины памяти. Процессор может копировать гигабайты данных в секунду. Сдвиг 10 000 элементов (40 КБ данных) полностью укладывается в кэш L1 и занимает микросекунды.

### 4.2 Сравнительные бенчмарки и точка пересечения

Бенчмарки показывают, что List<T>.Insert работает быстрее, чем LinkedList<T>.AddAfter, для коллекций размером до **10 000 – 50 000 элементов** (в зависимости от размера типа T).25

**Почему List побеждает на малых и средних N:**

1. **Скорость блочного копирования:** Аппаратное ускорение memmove компенсирует алгоритмическую сложность.
2. **Стоимость аллокации:** Вставка в LinkedList требует new LinkedListNode<T>(). Аллокация памяти в куче — это не бесплатная операция. Она требует поиска свободного блока, обновления указателей free-list, зануления памяти. Стоимость одной аллокации (в тактах CPU) может превышать стоимость копирования тысяч байт в массиве.20
3. **Локальность:** Данные массива, скорее всего, уже "горячие" (в кэше). Новый узел списка может быть выделен в "холодной" области памяти, вызывая немедленный промах кэша.

Точка пересечения (Crossover Point):

Существует пороговое значение $N$, после которого стоимость сдвига массива ($O(n)$) наконец превышает константную стоимость аллокации узла ($O(1)$). Для простых типов (int) это значение может достигать 50 000 - 100 000 элементов. Однако в типичных сценариях обработки запросов коллекции редко достигают таких размеров, что делает List<T> предпочтительным выбором в 99% случаев.19

## 5. Современная архитектура.NET: Span<T> и ref struct

Второй день обучения вводит концепции, появившиеся в новейших версиях.NET (.NET Core 2.1+ /.NET 5+), которые революционизировали работу с памятью: Span<T> и ref struct. Это переход от модели "владения" памятью к модели "безопасного просмотра" (viewing).

### 5.1 Проблема аллокаций при работе с подстроками

В классическом C# работа с частью массива или строки требовала создания копии.

Пример: string sub = text.Substring(0, 10);

* Аллоцирует новый объект string в куче.
* Копирует символы из оригинала.
* Создает нагрузку на GC (Garbage Pressure).  
  В высоконагруженных парсерах (JSON, HTTP-заголовки) эти временные объекты создают миллионы ненужных аллокаций, приводя к частым сборкам мусора.29

### 5.2 Внутреннее устройство Span<T>: Революция ref struct

Span<T> — это структура, обеспечивающая типобезопасное представление непрерывной области памяти. Она может указывать на:

1. Управляемые массивы (T).
2. Память на стеке (stackalloc T).
3. Неуправляемую память (void\*).

Внутренняя реализация:

Концептуально Span<T> содержит два поля:

C#

public readonly ref struct Span<T> {  
 internal readonly ByReference<T> \_pointer; // Управляемый внутренний указатель  
 private readonly int \_length;  
}

* **\_pointer (Interior Pointer):** Это не обычная ссылка на объект. Это **Управляемый Внутренний Указатель** (ByReference<T>). В отличие от обычной ссылки, которая обязана указывать на *начало* объекта (MethodTable), внутренний указатель может указывать в *середину* объекта (например, на 50-й элемент массива).
* **Взаимодействие с GC:** Это было сложнейшей инженерной задачей для команды CLR. Сборщик мусора должен уметь отслеживать эти внутренние указатели. Если во время компактизации (Compact Phase) GC перемещает массив в памяти, он обязан атомарно обновить поле \_pointer внутри всех активных Span на стеке, чтобы они указывали на новый адрес элемента.32 ByReference<T> является "JIT Intrinsic" — типом, который известен компилятору и рантайму особым образом и не может быть выражен в обычном C# до версии 11 (где появились ref поля).33

### 5.3 Ограничения безопасности ref struct

Поскольку Span<T> может указывать на память стека (которая живет только пока выполняется метод), его нельзя хранить в куче. Если бы Span на стековый массив сохранился в поле класса, и метод завершился, ссылка указывала бы на "мусор" в стеке (Use-After-Free).

Для предотвращения этого введен модификатор ref struct.

**Правила безопасности ref struct:**

1. **Только на стеке:** ref struct не может быть полем класса или обычной структуры (не ref). Он не может быть упакован (boxed) в object или интерфейс.
2. **Запрет на интерфейсы:** До.NET 8 ref struct не мог реализовывать интерфейсы, так как приведение к интерфейсу вызывает боксинг. (В.NET 8+ это разрешено с ограничениями).
3. **Запрет в async/await:** Span<T> нельзя использовать в асинхронных методах, если переменная должна пережить await. Асинхронные методы компилируются в машину состояний (класс/структуру в куче), куда копируются локальные переменные. Поскольку Span нельзя хранить в куче, компилятор запрещает такой код.36

Эти ограничения гарантируют, что Span всегда живет короче или столько же, сколько память, на которую он указывает. Это позволяет писать код с производительностью C++ (прямой доступ к памяти), но с гарантиями безопасности памяти C# (Memory Safety).

### 5.4 Zero-Allocation Slicing и прирост производительности

Использование метода Span.Slice(start, length) создает новый экземпляр структуры Span (просто копируя указатель и меняя длину), не аллоцируя память и не копируя данные. Это операция $O(1)$.

Бенчмарки:

Замена String.Substring на ReadOnlySpan<char>.Slice в задачах парсинга строк обычно дает:

* **Аллокации:** Падают до **0 байт**.
* **Скорость:** Увеличивается в **2-10 раз** за счет устранения копирования памяти и работы менеджера кучи.30

В.NET 8 внедрены дополнительные оптимизации для Span, использующие AVX-512 для операций типа IndexOf, SequenceEqual и Contains, что делает работу со спанами еще быстрее, чем с традиционными методами строк.23

## 6. Практическое применение: Анализ заданий второго дня

Практическое задание "Cache Misses Analyzer" направлено на эмпирическое подтверждение изложенной теории.

### 6.1 Задание: Бенчмарк итерации

* **Гипотеза:** Суммирование элементов List<int> будет значительно быстрее, чем LinkedList<int>, даже при одинаковом количестве операций сложения.
* **Механизм:** Итерация List компилируется в цикл с инкрементом указателя на 4 байта. Аппаратный префетчер обнаруживает страйд и подгружает кэш-линии. Итерация LinkedList компилируется в current = current.Next, вызывая зависимость по данным и промахи кэша.
* **Ожидаемый результат:** List должен быть быстрее в 5–20 раз, в зависимости от степени фрагментации кучи во время теста.41 Важно использовать атрибут [HardwareCounters(HardwareCounter.CacheMisses)] в BenchmarkDotNet для доказательства корреляции между промахами кэша и временем выполнения.

### 6.2 Задание: Zero-Allocation Parser

* **Задача:** Распарсить строку "User:John;Age:30;Role:Admin".
* **Наивный подход:** str.Split(';') -> создает массив строк string и новые объекты строк для каждой подстроки. Высокие аллокации.
* **Оптимизированный подход (Span):**
  1. Создать ReadOnlySpan<char> из строки.
  2. Найти индекс разделителя ; через span.IndexOf(';').
  3. Взять "срез" через Slice.
  4. Обработать срез (например, int.Parse(spanSegment) — современные перегрузки int.Parse принимают Span).
  5. Сдвинуть начало спана на следующий сегмент.
* **Результат:** Парсер работает без единой аллокации в куче, не создавая нагрузки на GC. Это критически важно для Middleware компонентов, обрабатывающих тысячи запросов в секунду.

## 7. Заключение: Взгляд Senior Architect

Теория второго дня — это не просто сравнение API коллекций; это урок **Механической Симпатии** (Mechanical Sympathy). Как архитектор Enterprise.NET систем, вы должны понимать, что выбор между List и LinkedList — это выбор между согласованностью с железом и борьбой с ним.

Современный бэкенд-инженер должен ставить **Локальность Данных** во главу угла при проектировании in-memory структур. Процессор — это гоночный болид, а подсистема памяти — это пробка в час пик. List<T>, массивы и Span<T> позволяют болиду ехать по свободной трассе (L1 Cache). LinkedList<T> заставляет болид останавливаться на каждом светофоре (RAM Latency).

**Рекомендации:**

1. **Default Choice:** List<T> (или массивы) должны быть выбором по умолчанию для 99% сценариев хранения последовательных данных.
2. **Избегайте LinkedList:** Используйте LinkedList<T> только в специфических нишах:
   * Когда требуется **O(1) удаление** узла, на который *уже есть ссылка*, без инвалидации других итераторов (например, в реализации LRU Cache).
   * Когда необходимо объединять (splice) списки без копирования данных.
3. **Используйте Span:** Для парсинга, обработки строк и работы с буферами ввода-вывода всегда предпочитайте Span<T> и Memory<T> вместо создания временных массивов или строк.

Понимание этих низкоуровневых деталей отличает разработчика, просто пишущего код на C#, от инженера, создающего высокопроизводительные системы на платформе.NET.

#### Источники

1. What Is CPU Cache? Guide to L1, L2 & L3 Performance - acemagic, дата последнего обращения: декабря 3, 2025, <https://acemagic.com/blogs/tips-tricks/what-is-cpu-cache>
2. What is the Cost of an L1 Cache Miss? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/1126529/what-is-the-cost-of-an-l1-cache-miss>
3. Caching and Performance of CPUs - Jyotiprakash's Blog, дата последнего обращения: декабря 3, 2025, <https://blog.jyotiprakash.org/caching-and-performance-of-cpus>
4. Arrays vs Linked Lists [2025 Guide] - HCL GUVI, дата последнего обращения: декабря 3, 2025, <https://www.guvi.in/blog/arrays-vs-linked-lists/>
5. How to Write Faster Code Than 90% of Programmers : r/Unity3D - Reddit, дата последнего обращения: декабря 3, 2025, <https://www.reddit.com/r/Unity3D/comments/69ybpt/how_to_write_faster_code_than_90_of_programmers/>
6. Does spatial locality matter for cache performance if object size > cache line?, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/50179938/does-spatial-locality-matter-for-cache-performance-if-object-size-cache-line>
7. Pointer Cache Assisted Prefetching - Computer Science - University of California San Diego, дата последнего обращения: декабря 3, 2025, <https://cseweb.ucsd.edu/~calder/papers/MICRO-02-PCache.pdf>
8. Tolerating Latency Through Prefetching, дата последнего обращения: декабря 3, 2025, <https://www.cs.cmu.edu/afs/cs/academic/class/15418-f19/www/lectures/20_prefetching.pdf>
9. Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation - Parallel Data Lab, дата последнего обращения: декабря 3, 2025, <http://pdl.cmu.edu/PDL-FTP/associated/16iccd_impica.pdf>
10. Building More Powerful Less Expensive Supercomputers Using Processing-In-Memory (PIM) LDRD Final Report - OSTI.gov, дата последнего обращения: декабря 3, 2025, <https://www.osti.gov/servlets/purl/993898>
11. Detecting and Eliminating Pipeline Stalls Through Intelligent Code Analysis, дата последнего обращения: декабря 3, 2025, <https://www.in-com.com/blog/detecting-and-eliminating-pipeline-stalls-through-intelligent-code-analysis/>
12. Code Transformations to Improve Memory Parallelism \*, дата последнего обращения: декабря 3, 2025, <https://jilp.org/vol2/v2paper11.pdf>
13. Measuring Reorder Buffer Capacity - Blog - Henry Wong, дата последнего обращения: декабря 3, 2025, <https://blog.stuffedcow.net/2013/05/measuring-rob-capacity/>
14. Techniques for Efficient Processing in Runahead Execution Engines - cs.wisc.edu, дата последнего обращения: декабря 3, 2025, <https://pages.cs.wisc.edu/~isca2005/papers/06B-01.PDF>
15. Instruction-level parallelism in practice: speeding up memory-bound programs with low ILP : r/cpp - Reddit, дата последнего обращения: декабря 3, 2025, <https://www.reddit.com/r/cpp/comments/vg2kwa/instructionlevel_parallelism_in_practice_speeding/>
16. What is a Linked list? Types of Linked List with Code Examples - freeCodeCamp, дата последнего обращения: декабря 3, 2025, <https://www.freecodecamp.org/news/what-is-a-linked-list-types-and-examples/>
17. Per-node memory overhead - linked list - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/29389854/per-node-memory-overhead>
18. c# - Why is a LinkedList Generally Slower than a List? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/5983059/why-is-a-linkedlist-generally-slower-than-a-list>
19. .NET — LinkedList vs ToArray : r/csharp - Reddit, дата последнего обращения: декабря 3, 2025, <https://www.reddit.com/r/csharp/comments/1ajya7n/net_linkedlist_vs_toarray/>
20. Performance Improvements in .NET 10 - Microsoft Developer Blogs, дата последнего обращения: декабря 3, 2025, <https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-10/>
21. Why is memmove faster than memcpy? - c++ - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/28623895/why-is-memmove-faster-than-memcpy>
22. Performance Improvements in .NET 8 - Microsoft Developer Blogs, дата последнего обращения: декабря 3, 2025, <https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-8/>
23. Announcing .NET 8 Preview 4 - Microsoft Developer Blogs, дата последнего обращения: декабря 3, 2025, <https://devblogs.microsoft.com/dotnet/announcing-dotnet-8-preview-4/>
24. What is the performace difference in using List
25. ArrayList has faster insert speed than LinkedList : r/javahelp - Reddit, дата последнего обращения: декабря 3, 2025, <https://www.reddit.com/r/javahelp/comments/whnih5/arraylist_has_faster_insert_speed_than_linkedlist/>
26. LinkedList
27. c# - When should I use a List vs a LinkedList - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/169973/when-should-i-use-a-list-vs-a-linkedlist>
28. An Ultimate Guide to Maximizing Performance with Span in C# | Programming Pulse, дата последнего обращения: декабря 3, 2025, <https://programmingpulse.vercel.app/blog/an-ultimate-guide-to-maximizing-performance-with-span-in-csharp>
29. Getting incredible performance improvement in C# by using Span | by Atakan Korez, дата последнего обращения: декабря 3, 2025, <https://medium.com/@atakankorez/getting-incredible-performance-improvement-in-c-by-using-span-1e993374b856>
30. Insane better performance with Span in C# | by John klaumann - Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@johnklaumann/insane-better-performance-with-span-in-c-e485ef176bb>
31. C# - All About Span: Exploring a New .NET Mainstay | Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/archive/msdn-magazine/2018/january/csharp-all-about-span-exploring-a-new-net-mainstay>
32. Managed pointers, Span, ref struct, C#11 ref fields and the scoped keyword - NDepend Blog, дата последнего обращения: декабря 3, 2025, <https://blog.ndepend.com/managed-pointers-span-ref-struct-c11-ref-fields-and-the-scoped-keyword/>
33. How does a Span survive garbage collection? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/74157694/how-does-a-span-survive-garbage-collection>
34. Low level struct improvements - C# feature specifications - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-11.0/low-level-struct-improvements>
35. C# 11.0 new features: ref fields and the scoped keyword - Endjin, дата последнего обращения: декабря 3, 2025, <https://endjin.com/blog/2024/09/dotnet-csharp-11-ref-fields-scoped-keyword>
36. ref struct types - C# reference - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/ref-struct>
37. What is the difference between Span
38. Performance comparison, AsSpan() vs Substring() - DotnetBenchmarks.com, дата последнего обращения: декабря 3, 2025, <https://dotnetbenchmarks.com/benchmark/1088>
39. Span – Adam Sitnik – .NET Performance and Reliability, дата последнего обращения: декабря 3, 2025, <https://adamsitnik.com/Span/>
40. Cache locality performance - c++ - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/14977845/cache-locality-performance>