Лекция №13. Линейные структуры данных: Стек, Очередь, Дек. Управление памятью: Куча (Heap) vs Стек (Stack).

ОГЛАВЛЕНИЕ

§13.1 Введение в линейные структуры данных

Линейные структуры данных представляют собой упорядоченные последовательности элементов, в которых каждый элемент имеет единственного предшественника и единственного преемника, за исключением крайних — первого и последнего. Такая организация обеспечивает простоту доступа, вставки и удаления элементов, при этом поведение структуры определяется правилами, по которым осуществляются эти операции.

Линейные структуры лежат в основе многих алгоритмов и системных решений: от обработки выражений и управления вызовами функций до реализации алгоритмов обхода графов и буферизации данных. Их ключевое преимущество — предсказуемость и эффективность базовых операций, таких

как добавление, извлечение и доступ к элементам, при условии, что поведение структуры соответствует решаемой задаче.

В **.NET Framework** и **.NET Core** для работы с линейными структурами предусмотрены специализированные обобщённые классы, такие как **Stack<T>**, **Queue<T>** и **LinkedList<T>**, а также возможность построения более сложных структур на их основе. Эти типы реализуют характерные дисциплины доступа к данным — например, «**последним пришёл — первым ушёл**» (*LIFO*) или «**первым пришёл — первым ушёл**» (*FIFO*), — что делает их незаменимыми в различных сценариях разработки.

Хотя все линейные структуры хранят данные последовательно, их различие заключается не в форме хранения, а в правилах доступа к элементам. Именно эти правила определяют, где и как та или иная структура применяется. Далее мы подробно рассмотрим три фундаментальных представителя линейных структур — **стек**, **очередь** и **дек**, — изучим их поведение, алгоритмы работы и типичные области применения.

§13.2 Линейная структура данных. Стек (Stack)

Стек — это линейная структура данных, организованная по принципу **LIFO** (*Last In, First Out*), то есть последним пришёл — первым вышел. Это означает, что элемент, добавленный последним, будет извлечён первым. Такое поведение аналогично реальной жизни: представьте стопку тарелок — вы можете положить новую тарелку сверху и снять только верхнюю.

Работа стека показана ниже:

В **.NET** стек представлен обобщённым классом **Stack<T>**, находящимся в пространстве имён **System.Collections.Generic**. Он реализует интерфейс **IEnumerable<T>**, что позволяет использовать его в циклах **foreach**, а также поддерживает типобезопасность за счёт использования обобщений. Создание стека осуществляется через конструктор:

```
var stack = new Stack<string>();
```

— это создаёт пустой стек строк. Также можно инициализировать стек коллекцией:

```
new Stack<int>(new[] {1, 2, 3})
```

при этом элементы будут добавлены в обратном порядке относительно исходной последовательности, так как стек строится путём последовательного добавления (*push*).

Основные методы стека:

- **Push(T item)** — добавляет элемент на вершину стека.
- **Pop()** — удаляет и возвращает верхний элемент стека. Если стек пуст, выбрасывается исключение **InvalidOperationException**.
- **Peek()** — возвращает верхний элемент без его удаления; также требует наличия элементов.
- **Count** — свойство, возвращающее количество элементов в стеке.
- **Clear()** — удаляет все элементы.
- **Contains(T item)** — проверяет наличие элемента в стеке (с временной сложностью *O(n)*).
- **ToArray()** — возвращает массив элементов в порядке от вершины к основанию стека.
- **TrimExcess()** — уменьшает вместимость стека, если количество элементов значительно меньше ёмкости внутреннего массива.

Работа стека основана на динамическом массиве, который автоматически расширяется при нехватке места. Это обеспечивает амортизированную временную сложность *O(1)* для операций **Push** и **Pop**.

Стек находит широкое применение в алгоритмах и системном

программировании. Одно из ключевых применений — вычисление выражений в **обратной польской записи** (*ОПЗ*). В **ОПЗ** операции записываются после операндов, и для вычисления выражения используется стек: числа помещаются в стек, а при встрече оператора из стека извлекаются два верхних элемента, к ним применяется операция, и результат возвращается обратно в стек.

Например, выражение **3 4 + 5 *** обрабатывается следующим образом: **3** и **4** кладутся в **стек**, при **+** они заменяются на **7**; затем **5** кладётся в **стек**, при ***** — **7** и **5** перемножаются, результат **35** остаётся в стеке.

§13.2.1 Обратная Польская Запись

Обратная Польская Запись (англ. *Reverse Polish notation, RPN*) — форма записи математических и логических выражений, в которой операнды расположены перед знаками операций.

§13.2.1.1 Преобразование выражения в Обратную Польскую Запись с использованием стека

Как правило арифметические выражения удобно преобразовывать в **обратную польскую запись** (*ОПЗ*), чтобы избавиться от скобок, содержащихся в выражении. Выражения, преобразованные в **ОПЗ**, можно вычислять последовательно, слева направо.

Существует два наиболее известных способа преобразования в **ОПЗ**. Рассмотрим коротко каждый из них.

Нам понадобится стек для переменных типа **char**, т.к. исходное выражение мы получаем в виде строки.

Рассматриваем поочередно каждый символ:

1. Если этот символ – число (или переменная), то просто помещаем его в выходную строку.
2. Если символ – знак операции *(+, -, *, /*), то проверяем приоритет данной операции. Операции умножения и деления имеют наивысший приоритет (допустим он равен 3). Операции сложения и вычитания имеют меньший приоритет (равен 2). Наименьший приоритет (равен 1) имеет

открывающая скобка.

Получив один из этих символов, мы должны проверить стек:

a) Если стек все еще пуст, или находящиеся в нем символы (а находится в нем могут только знаки операций и открывающая скобка) имеют меньший приоритет, чем приоритет текущего символа, то помещаем текущий символ в стек.

b) Если символ, находящийся на вершине стека имеет приоритет, больший или равный приоритету текущего символа, то извлекаем символы из стека в выходную строку до тех пор, пока выполняется это условие; затем переходим к пункту а).

3. Если текущий символ – открывающая скобка, то помещаем ее в стек.

4. Если текущий символ – закрывающая скобка, то извлекаем символы из стека в выходную строку до тех пор, пока не встретим в стеке открывающую скобку (т.е. символ с приоритетом, равным 1), которую следует просто уничтожить. Закрывающая скобка также уничтожается.

Если вся входная строка разобрана, а в стеке еще остаются знаки операций, извлекаем их из стека в выходную строку.

Рассмотрим алгоритм на примере простейшего выражения: Дано выражение:

```
a + ( b - c ) * d
```

Рассмотрим поочередно все символы:

Символ	Действие	Состояние выходной строки после совершенного действия	Состояние стека после совершенного действия
a	'a' - переменная. Помещаем ее в выходную строку	a	пуст
+	'+' - знак операции. Помещаем его в стек (поскольку стек пуст, приоритеты можно не проверять)	a	+
(	'(' - открывающая скобка. Помещаем в стек	a	+ (

b	'b' - переменная. Помещаем ее в выходную строку	a b	+ (
-	'-' - знак операции, который имеет приоритет 2. Проверяем стек: на вершине находится символ '(', приоритет которого равен 1. Следовательно мы должны просто поместить текущий символ '-' в стек	a b	+ (-
c	'c' - переменная. Помещаем ее в выходную строку	a b c	+ (-
)	')' - закрывающая скобка. Извлекаем из стека в выходную строку все символы, пока не встретим открывающую скобку. Затем уничтожаем обе скобки	a b c -	+
*	'*' - знак операции, который имеет приоритет 3. Проверяем стек: на вершине находится символ '+', приоритет которого равен 2, т.е. меньший, чем приоритет текущего символа '*'. Следовательно мы должны просто поместить текущий символ '*' в стек	a b c -	+ *
d	'd' - переменная. Помещаем ее в выходную строку	a b c – d	+ *

Теперь вся входная строка разобрана, но в стеке еще остаются знаки операций, которые мы должны просто извлечь в выходную строку. Поскольку **стек** — это структура, организованная по принципу LIFO, сначала извлекается символ '*****', затем символ '**+**'. Итак, мы получили конечный результат:

```
a b c - d * +
```

§13.2.1.2 Алгоритм вычисления выражения, записанного в Обратной Польской Записи

Для реализации этого алгоритма используется стек для чисел (или для переменных, если они встречаются в исходном выражении). Алгоритм очень прост. В качестве входной строки мы теперь рассматриваем выражение, записанное в **ОПЗ**:

1. Если очередной символ входной строки – число, то кладем его в

стек.

2. Если очередной символ – знак операции, то извлекаем из стека два верхних числа, используем их в качестве операндов для этой операции, затем кладем результат обратно в стек.

Когда вся входная строка будет разобрана в стеке должно остаться одно число, которое и будет результатом данного выражения.

Рассмотрим этот алгоритм на примере выражения:

```
7 5 2 - 4 * +
```

Рассмотрим поочередно все символы:

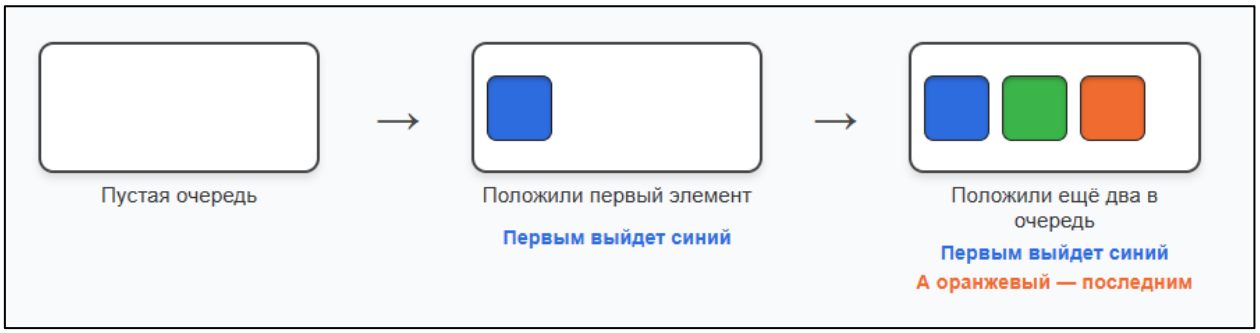
Символ	Действие	Состояние стека после совершенного действия
7	'7' – число. Помещаем его в стек	7
5	'5' – число. Помещаем его в стек	7 5
2	2' – число. Помещаем его в стек	7 5 2
-	'-' – знак операции. Извлекаем из стека 2 верхних числа (5 и 2) и совершаем операцию 5 - 2 = 3, результат которой помещаем в стек	7 3
4	'4' – число. Помещаем его в стек	7 3 4
*	'*' – знак операции. Извлекаем из стека 2 верхних числа (3 и 4) и совершаем операцию 3 * 4 = 12, результат которой помещаем в стек	7 12
+	'+' – знак операции. Извлекаем из стека 2 верхних числа (7 и 12) и совершаем операцию 7 + 12 = 19, результат которой помещаем в стек	19

Теперь строка разобрана и в стеке находится одно число 19, которое является результатом исходного выражения.

§13.3 Линейная структура данных. Очередь (Queue)

Очередь — это линейная структура данных, организованная по принципу **FIFO** (*First In, First Out*), то есть первым пришёл — первым вышел. Такой порядок напоминает реальную очередь в магазине: тот, кто встал первым, обслуживается первым. Элементы добавляются в конец очереди, а извлекаются — из её начала.

Работа очереди показана ниже:



В **.NET** очередь представлена обобщённым классом **Queue<T>**, доступным в пространстве имён **System.Collections.Generic**. Как и **Stack<T>**, он типобезопасен и реализует интерфейс **IEnumerable<T>**, что позволяет выполнять итерацию по элементам в порядке их поступления. Создаётся очередь с помощью конструктора:

```
var queue = new Queue<int>();
```

— это инициализирует пустую очередь целых чисел. Также можно передать начальную коллекцию:

```
new Queue<string>(new[] {"a", "b", "c"})
```

при этом элементы будут добавлены в том же порядке, и первый из них станет «головой» очереди.

Очередь в **.NET** реализована на основе кольцевого (циклического) массива, что позволяет эффективно управлять памятью и поддерживать высокую производительность операций вставки и извлечения. При необходимости массив автоматически расширяется, обеспечивая амортизированную временную сложность *O(1)* для основных операций.

Основные методы и свойства:

- **Enqueue(T item)** — добавляет элемент в конец очереди.
- **Dequeue()** — удаляет и возвращает элемент из начала очереди. Если очередь пуста, выбрасывается исключение **InvalidOperationException**.
- **Peek()** — возвращает элемент из начала очереди без его удаления.
- **Count** — свойство, возвращающее текущее количество элементов.
- **Clear()** — удаляет все элементы из очереди.
- **Contains(T item)** — проверяет наличие элемента (*O(n)*).

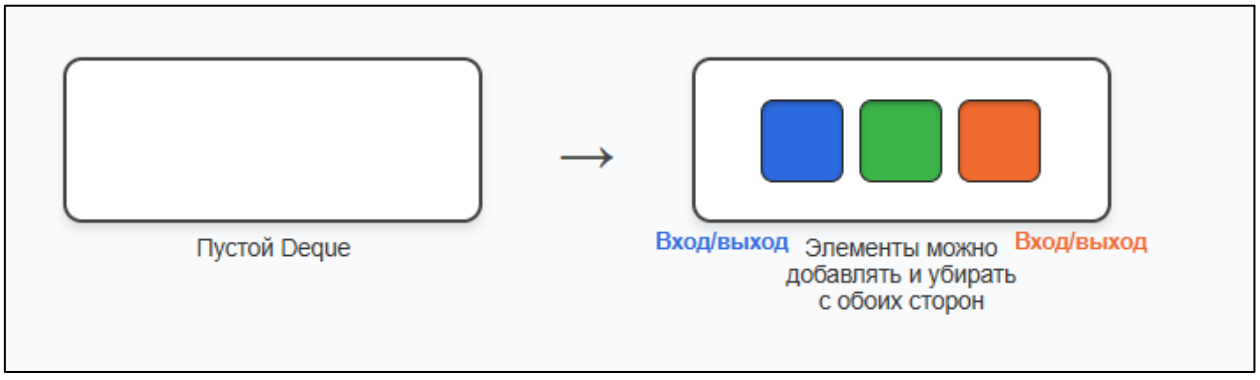
- **ToArray()** — создаёт массив, содержащий элементы очереди в порядке их следования.
- **TrimExcess()** — при значительном превышении ёмкости над количеством элементов может быть вызван для оптимизации памяти.

Очередь находит широкое применение в задачах, где важен порядок обработки. Например, она используется в алгоритмах обхода графов в ширину (*BFS — breadth-first search*), где вершины посещаются по уровням. Также очередь применяется в системах обработки задач: веб-серверы используют очереди для управления входящими запросами, операционные системы — для планирования процессов (например, в алгоритмах *FCFS — First Come, First Served*).

В многопоточном программировании для безопасного обмена данными между потоками часто используются потокобезопасные реализации очередей, такие как **ConcurrentQueue<T>** из пространства имён **System.Collections.Concurrent**, которая позволяет нескольким потокам одновременно добавлять и извлекать элементы без необходимости явной синхронизации.

§13.4 Линейная структура данных. Дек (Deque)

Дек, или **двусторонняя очередь** (от англ. *deque — double-ended queue*), представляет собой линейную структуру данных, которая расширяет возможности обычной очереди, позволяя добавлять и удалять элементы с обоих концов — как с начала, так и с конца. Название «**дек**» подчёркивает эту *двунаправленность*: он сочетает в себе черты как стека, так и очереди, обеспечивая гибкость при организации доступа к данным.



В **.NET** стандартная библиотека не предоставляет отдельного типа с названием **Deque**, однако функциональность дека в значительной степени покрывается классом **LinkedList<T>** — двусвязным списком из пространства имён **System.Collections.Generic**. Хотя **LinkedList<T>** не называется деком напрямую, он позволяет эффективно выполнять вставку и удаление в начало и конец за время *O(1)*, что делает его удобной основой для реализации дека.

Альтернативно, при необходимости, дек можно реализовать самостоятельно на основе динамического массива с циклическим использованием памяти, но стандартные средства **.NET** в этом плане ограничены.

Работа с **LinkedList<T>** как с деком предполагает использование методов: *AddFirst(T), AddLast(T), RemoveFirst(), RemoveLast()*, а также свойств **First** и **Last** для просмотра крайних элементов без извлечения. Например:

```
list.AddLast(5)
```

добавляет элемент в хвост, а

```
list.RemoveFirst()
```

удаляет из головы — это соответствует поведению очереди. В то же время **AddLast** и **RemoveLast** имитируют стек. Таким образом, дек может использоваться как в режиме **FIFO**, так и **LIFO**, в зависимости от выбранных операций.

Основные операции, ассоциируемые с деком:

- **PushFront(T item)** — условное добавление в начало (аналог *AddFirst*).
- **PushBack(T item)** — добавление в конец (аналог *AddLast*).
- **PopFront()** — извлечение из начала.
- **PopBack()** — извлечение из конца.
- **PeekFront()** — просмотр первого элемента.
- **PeekBack()** — просмотр последнего элемента.
- **Count** — количество элементов.
- **Clear()** — полная очистка.

Дек находит применение в задачах, где требуется гибкий доступ к обоим концам последовательности. Одно из классических применений — алгоритм скользящего окна, например, при поиске максимума в каждом подмассиве фиксированной длины: в деке поддерживается упорядоченная структура индексов, позволяющая за *O(1)* получать текущий максимум.

Также дек используется в реализации буферов с ограниченным размером, в алгоритмах обхода дерева по уровням с чередованием направлений (*zigzag traversal*), а также в парсерах и кэширующих системах, где важна возможность как быстрой вставки, так и извлечения с обеих сторон.

§13.5 Реализация структур в C#: классы Stack<T>, Queue<T>, LinkedList<T>

Как уже было сказано в предыдущих параграфах, в **.NET** для работы с линейными структурами данных предусмотрены типобезопасные обобщённые классы — **Stack<T>**, **Queue<T>** и **LinkedList<T>**. Каждый из них реализует определённый порядок доступа к элементам и предоставляет набор методов, соответствующих логике своей структуры: стек (*LIFO*), очередь (*FIFO*) и двусвязный список, который может служить основой для дека.

Стек — это коллекция, в которой элементы добавляются и извлекаются с одного конца, называемого «вершиной». Основные операции — **Push** и **Pop**.

```
 1. var stack = new Stack<string>();
 2.
 3. // Push — добавляет элемент на вершину стека
 4. stack.Push("Первый");
 5. stack.Push("Второй");
 6. stack.Push("Третий");
 7.
 8. // Count — возвращает количество элементов
 9. // Вывод: 3
10. Console.WriteLine($"Количество: {stack.Count}");
11. // Peek — возвращает верхний элемент, не удаляя его
12. // Вывод: "Третий"
13. Console.WriteLine($"Peek: {stack.Peek()}");
14. // Вывод: 3 (не изменился)
15. Console.WriteLine($"Count - Peek: {stack.Count}");
```

```
// Pop — удаляет и возвращает верхний элемент
// Вывод: "Третий"
Console.WriteLine($"Pop: {stack.Pop()}");
// Вывод: 2
Console.WriteLine($"Count - Pop: {stack.Count}");

// Contains — проверяет наличие элемента (O(n))
// Вывод: True
Console.WriteLine($"Содержит 'Первый'?  +
                   {stack.Contains("Первый")}");

// ToArray — возвращает массив от вершины к основанию
var a = stack.ToArray();
// Вывод: [Второй, Первый]
Console.WriteLine($"ToArray: [{string.Join(", ", a)}]");

// Clear — удаляет все элементы
stack.Clear();
// Вывод: 0
Console.WriteLine($"Count - Clear: {stack.Count}");

// Попытка Pop или Peek на пустом стеке вызывает исключение
// InvalidOperationException: Stack empty
// stack.Pop();
```

Метод *TrimExcess()* может быть полезен, если после удаления большого количества элементов внутренний массив остаётся избыточно большим. Он пересчитывает порог и, при необходимости, уменьшает ёмкость:

```
var s = new Stack<int>(Enumerable.Range(1, 1000));
for (int i = 0; i < 995; i++)
    // Осталось 5 элементов
    s.Pop();
// Ёмкость теперь соответствует текущему размеру
s.TrimExcess();
```

Очередь добавляет элементы в конец и извлекает из начала. Это поведение идеально подходит для сценариев обработки в порядке поступления.

```
var queue = new Queue<int>();

// Enqueue — добавляет элемент в конец очереди
queue.Enqueue(10);
queue.Enqueue(20);
queue.Enqueue(30);
```

```
// Count — текущее количество элементов
// Вывод: 3
Console.WriteLine($"Count: {queue.Count}");

// Peek — возвращает первый элемент без удаления
// Вывод: 10
Console.WriteLine($"Peek: {queue.Peek()}")
// Вывод: 3
Console.WriteLine($"Count - Peek: {queue.Count}");

// Dequeue — удаляет и возвращает первый элемент
// Вывод: 10
Console.WriteLine($"Dequeue: {queue.Dequeue()}");
// Вывод: 20
Console.WriteLine($"Dequeue: {queue.Dequeue()}");

// Contains — проверка наличия (O(n))
// Вывод: True
Console.WriteLine($"30? {queue.Contains(30)}");

// ToArray — возвращает элементы в порядке очереди
var a = queue.ToArray();
// Вывод: [30]
Console.WriteLine($"ToArray: [{string.Join(", ", a)}]");

// Clear — полная очистка
queue.Clear();
// Вывод: 0
Console.WriteLine($"Count - Clear: {queue.Count}");

// TrimExcess — оптимизация памяти
var l = new Queue<int>(Enumerable.Range(1, 1000));
for (int i = 0; i < 990; i++)
    l.Dequeue();
l.TrimExcess(); // Ёмкость уменьшена
```

LinkedList<T> не реализует интерфейс дека напрямую, но позволяет эффективно добавлять и удалять элементы с обоих концов, что делает его идеальной базой для реализации дека. Ниже — примеры всех ключевых операций, соответствующих деку.

```
var deque = new LinkedList<int>();

// PushFront (AddFirst) — добавление в начало
deque.AddFirst(1);
deque.AddFirst(0); // Теперь: 0 -> 1
```

```
// PushBack (AddLast) — добавление в конец
deque.AddLast(2); // Теперь: 0 -> 1 -> 2
deque.AddLast(3); // Теперь: 0 -> 1 -> 2 -> 3

// Count — количество элементов
// Вывод: 4
Console.WriteLine($"Count: {deque.Count}");

// PeekFront — просмотр первого элемента
if (deque.First != null)
    // Вывод: 0
    Console.WriteLine($"PeekFront: {deque.First.Value}");

// PeekBack — просмотр последнего элемента
if (deque.Last != null)
    // Вывод: 3
    Console.WriteLine($"PeekBack: {deque.Last.Value}");

// PopFront — извлечение из начала
int front = deque.First.Value;
deque.RemoveFirst();
// Вывод: 0
Console.WriteLine($"PopFront: {front}");
// Вывод: 3
Console.WriteLine($"Count - PopFront: {deque.Count}");

// PopBack — извлечение с конца
int back = deque.Last.Value;
deque.RemoveLast();
// Вывод: 3
Console.WriteLine($"PopBack: {back}");
// Вывод: 2
Console.WriteLine($"Count - PopBack: {deque.Count}");

// Содержимое после операций
Console.WriteLine("Оставшиеся элементы:");
foreach (var item in deque)
    Console.Write(item + " "); // Вывод: 1 2
Console.WriteLine();

// Contains — проверка наличия элемента (O(n))
// Вывод: True
Console.WriteLine($"Содержит 1? {deque.Contains(1)}");

// Clear — удаление всех элементов
deque.Clear();
// Вывод: 0
Console.WriteLine($"Count - Clear: {deque.Count}");
```

Классы **Stack<T>**, **Queue<T>** и **LinkedList<T>** предоставляют

эффективные и надёжные реализации фундаментальных линейных структур данных в C#.

- **Stack<T>** идеален для LIFO-логики: ОПЗ, отмена действий, рекурсия.
- **Queue<T>** — для FIFO-обработки: задачи, события, алгоритмы в ширину.
- **LinkedList<T>** — гибкая структура, позволяющая реализовать дек, LRU-кэш или произвольные списки с частыми вставками/удалениями.

§13.6 Сравнение стека, очереди и дека

Ниже представлена сравнительная таблица, обобщающая ключевые характеристики стека, очереди и дека в контексте:

Характеристика	Стек	Очередь	Дек
Принцип доступа	LIFO	FIFO	Двусторонний доступ
Основные операции	Push, Pop, Peek	Enqueue, Dequeue, Peek	AddFirst / AddLast, RemoveFirst / RemoveLast
Добавление	Только на вершину	Только в конец	В начало или в конец
Удаление	Только с вершины	Только из начала	Из начала или из конца
Сложность вставки/удаления	O(1)	O(1)	O(1)
Сложность поиска	O(n)	O(n)	O(n)
Доступ по индексу	Нет	Нет	Нет (только через обход)
Внутренняя реализация	Динамический массив	Кольцевой буфер	Двусвязный список
Итерация	От вершины к основанию	От начала к концу	В порядке следования узлов
Типичные применения	ОПЗ, DFS, отмена действий	BFS, планирование, буферы	Скользящее окно, LRU-кэш, деки

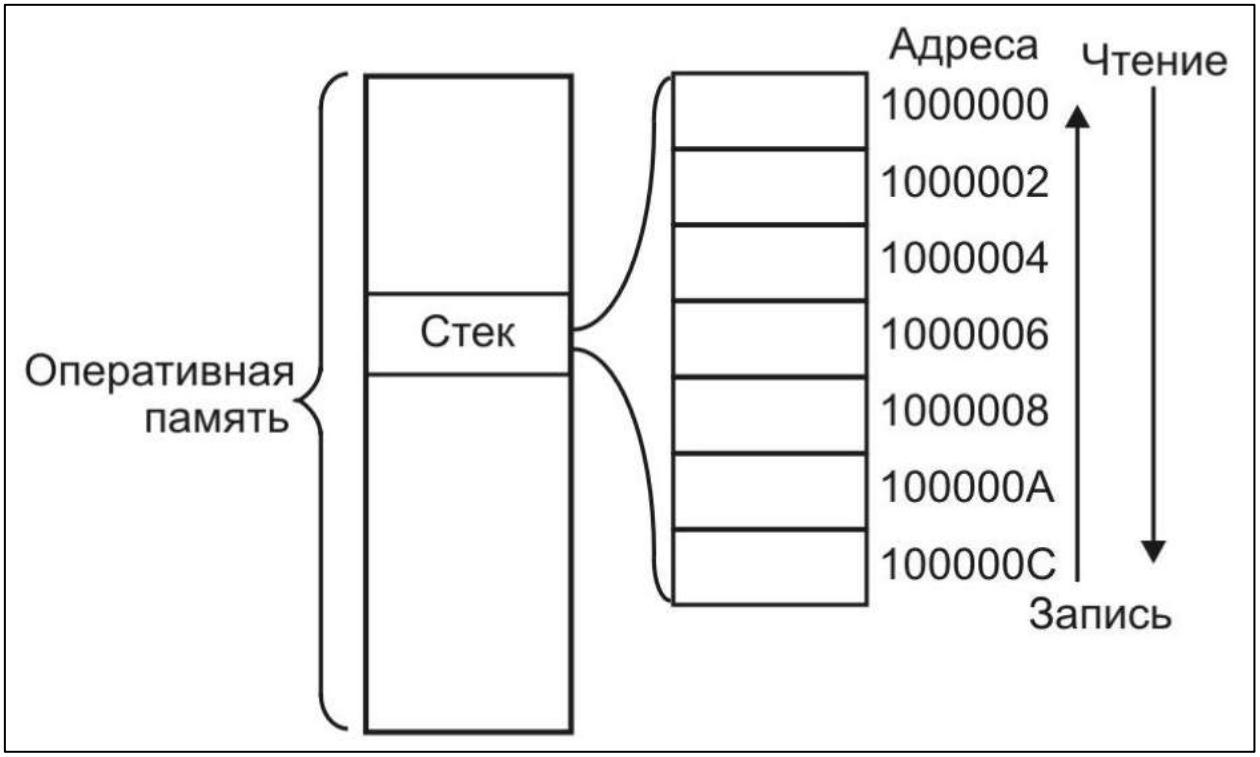
Из таблицы видно, что выбор структуры зависит от логики доступа, а не только от производительности. **Стек** и **очередь** — узкоспециализированные структуры: они просты, эффективны и идеальны в своих сценариях. Дек,

напротив, предлагает максимальную гибкость, но требует более сложной реализации и не всегда оправдан, если доступ нужен только с одной стороны

§13.7 Управление памятью в .NET: Куча (Heap) vs Стек (Stack)

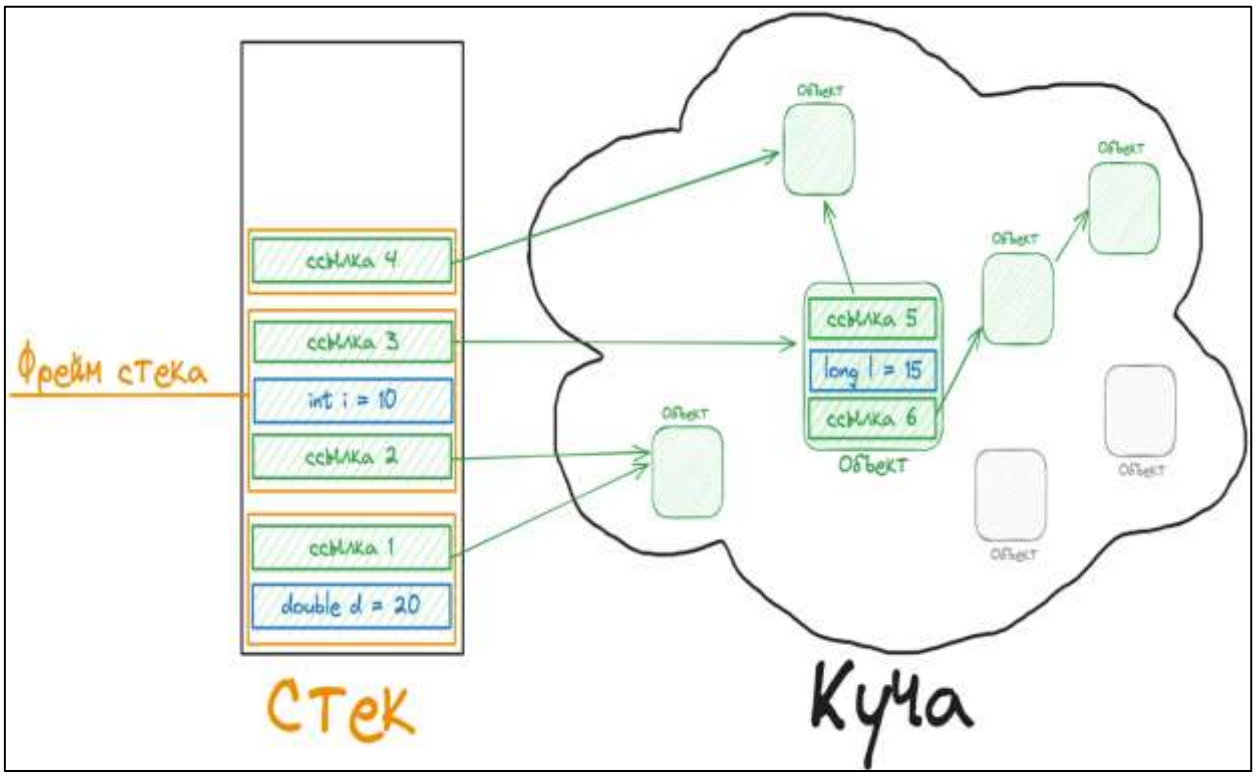
В платформе **.NET** управление памятью — это фундаментальный механизм, обеспечивающий эффективное выполнение приложений и предотвращение утечек ресурсов. Два ключевых компонента памяти, с которыми работает среда выполнения (**CLR** — *Common Language Runtime*), — это стек и куча. Они различаются по назначению, способу выделения памяти и времени жизни данных.

Стек — это участок памяти, используемый для хранения локальных переменных и информации о вызовах методов. Он работает по принципу **LIFO** (*Last In, First Out*): каждый раз при вызове метода в стек помещается новый фрейм (кадр), содержащий параметры, локальные переменные и адрес возврата. После завершения метода фрейм удаляется. Память в стеке выделяется и освобождается очень быстро, поскольку управление осуществляется процессором напрямую. Стек имеет ограниченный размер (обычно несколько мегабайт), и его переполнение приводит к исключению **StackOverflowException**.



На представленной схеме показана упрощённая модель организации стека в оперативной памяти процесса, выполняющегося в **.NET** (или в любой другой среде выполнения).

Куча (*heap*) — это динамическая область памяти, общая для всех потоков, где размещаются объекты ссылочного типа: экземпляры классов, массивы, делегаты и другие сложные структуры. В отличие от стека, выделение памяти в куче происходит не по строгому порядку, а по мере необходимости, с помощью оператора **new**. Освобождение памяти в **.NET** управляется автоматически сборщиком мусора (*Garbage Collector, GC*), который периодически определяет, какие объекты больше не используются, и освобождает их. Это избавляет разработчика от ручного управления памятью, но влечёт за собой непредсказуемость времени освобождения и возможные задержки при сборке.



Различия между стеком и кучей проявляются на уровне типов. Значимые типы (*struct, int, bool, DateTime и т.д.*) по умолчанию размещаются на стеке, если они являются локальными переменными или полями других значимых типов. Ссылочные типы (*class, string, object, массивы*) всегда создаются в куче, а на стеке хранится лишь ссылка на них. Например:

```
 1. void Example()
 2. {
 3.     // x — значение, хранится на стеке
 4.     int x = 42;
 5.     // s — ссылка на объект в куче
 6.     string s = "Hello";
 7.     // объект Person — в куче, ссылка — на стеке
 8.     var person = new Person();
 9. } // При выходе из метода:
10. // x, s, person удаляются со стека,
11. // но объект в куче может остаться,
12. // если на него есть другие ссылки
```

После завершения метода фрейм удаляется со стека, и все локальные переменные исчезают. Однако объекты в куче продолжают существовать, пока на них есть активные ссылки. Только когда **GC** обнаруживает, что объект недостижим, он помечает его как мусор и освобождает память, обычно во время следующей сборки.

С точки зрения производительности, работа со стеком значительно быстрее: выделение и освобождение памяти сводится к сдвигу указателя. Куча же требует более сложного управления — выделение памяти медленнее, а сборка мусора может вызывать паузы в выполнении программы. Однако куча позволяет хранить объекты произвольного размера и длительности жизни, что делает её незаменимой для большинства прикладных задач.

§13.8 Как работает стек вызовов (Call Stack)

Стек вызовов — это специализированная область памяти, управляющая последовательностью выполнения методов в программе. Он реализован как стек (*LIFO — Last In, First Out*) и играет ключевую роль в поддержке вызовов функций, передачи параметров, хранения локальных переменных и возврата управления после завершения метода. Каждый раз, когда вызывается метод, в стеке вызовов создаётся новый кадр (*stack frame*), содержащий всю необходимую информацию для его выполнения: аргументы, локальные переменные, адрес возврата и состояние выполнения. При завершении метода его кадр удаляется, и управление возвращается к вызывающему методу, расположенному в предыдущем кадре.

Работа стека вызовов особенно наглядна при рекурсии — когда метод вызывает сам себя. Каждый новый уровень рекурсии добавляет новый кадр на вершину стека, и только после достижения базового случая начинается пошаговое «*раскручивание*» — возврат из каждого вызова в обратном порядке.

Рассмотрим пример:

```
 1. static int Factorial(int n)
 2. {
 3.     if (n <= 1) return 1;
 4.     return n * Factorial(n - 1);
 5. }
 6.
 7. static void Main()
 8. {
 9.     int result = Factorial(3);
10.     Console.WriteLine(result);
11. }
```

При вызове *Factorial(3)* формируется следующая цепочка кадров в стеке вызовов:

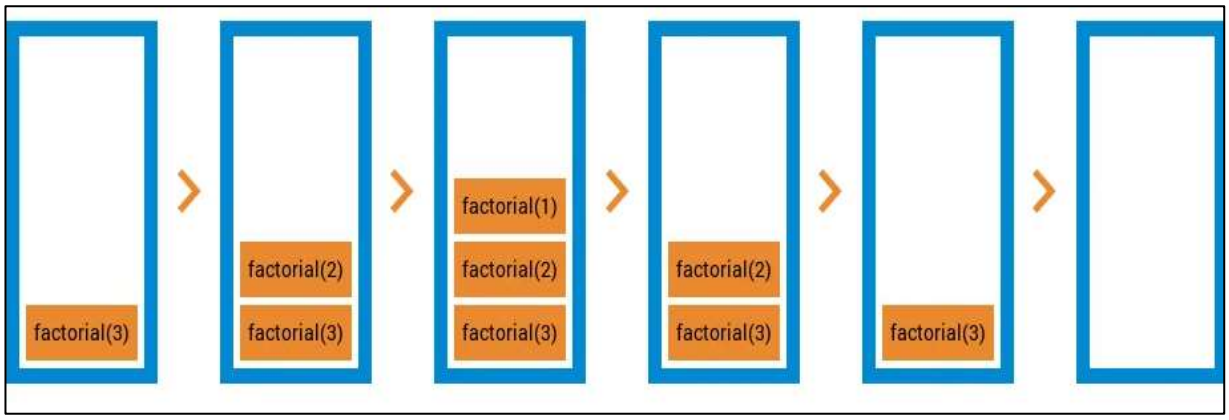
- **Main()** — точка входа, вызывает *Factorial(3)*.
- **Factorial(3)** — помещает **n = 3**, вычисляет **3 * Factorial(2)**, но не может завершиться, пока не получит результат **Factorial(2)**.
- **Factorial(2)** — **n = 2**, требует результат **Factorial(1)**.
- **Factorial(1)** — базовый случай, возвращает **1**.

Теперь начинается возврат:

- **Factorial(1)** → возвращает **1** в **Factorial(2)**.
- **Factorial(2)** → вычисляет **2 * 1 = 2**, возвращает **2** в **Factorial(3)**.
- **Factorial(3)** → вычисляет **3 * 2 = 6**, возвращает **6** в **Main()**.

Каждый кадр удаляется сразу после возврата значения. Таким образом, стек вызовов обеспечивает корректную последовательность выполнения и возврата, гарантируя, что каждый метод завершится в правильном контексте.

Схематичная демонстрация работы стека вызовов представлена ниже:



Однако ограниченный размер стека (обычно 1 МБ на поток) делает его уязвимым к переполнению — **StackOverflowException**. Это часто происходит при бесконечной или слишком глубокой рекурсии, когда кадры накапливаются быстрее, чем освобождаются. Например, вызов **Factorial(-1)** без корректной проверки приведёт к бесконечной цепочке вызовов и аварийному завершению программы.

§13.9 Утечки памяти и управление ресурсами

Напомним, что **утечка памяти** (*memory leak*) в программировании — это ситуация, когда программа использует память, но не освобождает её после завершения использования. В результате память больше не используется программой, но не может быть возвращена операционной системой или другими процессами.

Несмотря на наличие автоматического управления памятью в **.NET** через сборщика мусора (*Garbage Collector*), приложения всё ещё могут страдать от утечек памяти — ситуации, когда объекты в куче остаются недоступными для освобождения, хотя фактически больше не используются. Это происходит не из-за отсутствия free, как в C++, а из-за сохранения достижимых ссылок на ненужные объекты, что препятствует их сборке.

Сборщик мусора освобождает память только для объектов, на которые нет активных ссылок, достижимых от корней (локальные переменные, статические поля, параметры и т.д.). Если ссылка остаётся, даже случайно, объект будет удерживаться в памяти неограниченно долго.

Один из типичных сценариев утечки — подписка на события без

отписки:

```
 1. class EventPublisher
 2. {
 3.     public event Action OnEvent = delegate { };
 4.     public void Raise() => OnEvent();
 5. }
 6.
 7. class EventSubscriber
 8. {
 9.     // Тяжёлый объект
10.     private string _data = new string('x', 10000);
11.
12.     public EventSubscriber(EventPublisher p)
13.     {
14.         // Подписались...
15.         p.OnEvent += HandleEvent;
16.     }
17.
18.     private void HandleEvent() { /* обработка */ }
19. }
```

Если экземпляр **EventSubscriber** больше не нужен, но не отписался от события, **EventPublisher** по-прежнему хранит ссылку на него (через делегат). Следовательно, сборщик мусора не сможет удалить **EventSubscriber**, даже если на него нет других ссылок — он остаётся достижимым через цепочку: *publisher → event → subscriber*. Это классическая утечка памяти.

Другой пример — кэши без ограничений:

```
1. private static Dictionary<string, object> _cache
2.     = new();
3.
4. public void AddToCache(string key, object data)
5. {
6.     // Элементы накапливаются вечно
7.     _cache[key] = data;
8. }
```

Без механизма очистки (например, по времени жизни или размеру) кэш будет расти неограниченно, потребляя всё больше памяти.

Также утечки возникают при неправильном использовании финализируемых объектов, особенно тех, что управляют неуправляемыми ресурсами — файлами, сокетами, дескрипторами **GDI**. Хотя **.NET**

предоставляет **IDisposable** и оператор **using**, их игнорирование приводит к задержке освобождения ресурсов:

```
 1. void BadExample()
 2. {
 3.     var fileStream =
 4.         new FileStream("data.txt", FileMode.Open);
 5.     // ... работа
 6.     // Нет Dispose() и нет using
 7.     // — файл остаётся заблокированным
 8.     // Объект будет освобождён только
 9.     // при финализации, что непредсказуемо
10. }
```

Правильное управление ресурсами требует явного вызова Dispose() через блок **using**:

```
1. void GoodExample()
2. {
3.     using (var s = new FileStream("data.txt", FileMode.Open))
4.     {
5.         // ... работа
6.     } // Dispose() вызывается автоматически
7. }
```

Или использование **using** в объявлении (*C# 8+*):

```
1. using var ф = new FileStream("f.txt",FileMode.Open);
2. // ... работа
3. // Dispose вызывается в конце области видимости
```