1. Теоретические основы

В среде .NET стандартный массив (`Array`) является структурой фиксированной длины. Он аллоцируется (память для него выделяется) в непрерывной области памяти, и его размер не может быть изменен после создания.

Класс `List<T>` (динамический массив) является абстракцией *(принцип абстракции предполагает скрытие сложной логики за простыми интерфейсами-рычагами)* над обычным массивом. Он инкапсулирует *(то есть скрывает внутри себя и не дает доступ пользователю класса - вам)* логику управления памятью, создавая иллюзию "бесконечного" контейнера, который автоматически расширяется по мере добавления элементов.

2. Структура данных (Internal State)

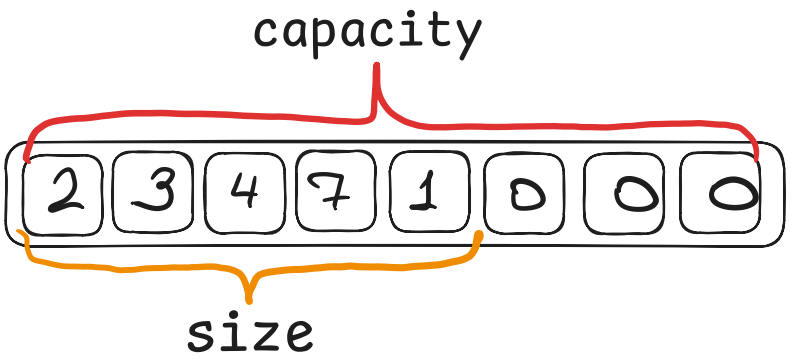
Для реализации логики списка необходимы два поля класса:

1. **Буфер данных (`_items`)**: массив типа `T[]`. Это физическое хранилище элементов. Длина этого массива (`_items.Length`) называется **Capacity** (Емкость).
2. **Счетчик элементов (`_size`)**: целое число (`int`). Хранит количество фактически добавленных пользователем элементов. Это значение возвращается свойством **Count**.

Получается:

- `Capacity` - сколько памяти выделено системой (размер буфера).
- `Count` - сколько памяти реально используется (значимые данные).

Инвариант: `Count` всегда должен быть меньше или равен `Capacity`.
Инвариант - выражение, определяющее непротиворечивое внутреннее состояние объекта. *Вернитесь на строку выше и прочтите еще раз.* Для списка нарушение этого инварианта невозможно, ввиду логики его работы.



На рисунке выше показана разница между `_size` и он же `Count` и `_items.Length` он же `Capacity`. На примере `Count` равен **5**, а `Capacity` равен **8**. Индекс последнего элемента (значение **1**) - 4. Изначально список был `Capacity=4`. После перед добавлением элемента **"1"** внутренний массив "вырос" в два раза. **1** встал на новое место `_size`, остальные элементы "свободны". Когда будем добавлять 9ый элемент список (`Capacity`) снова "вырастет" в 2 раза и будет уже 16.

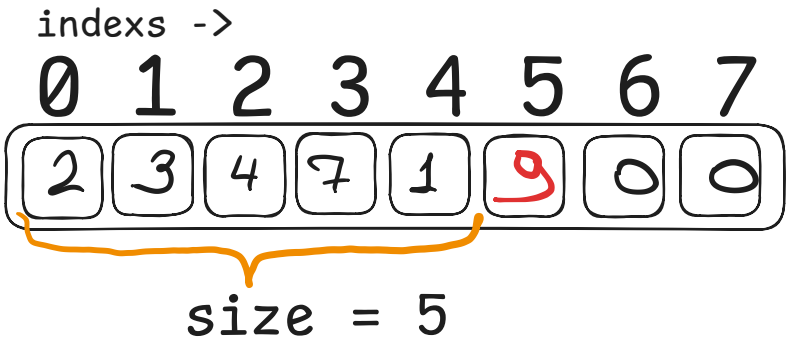
3. Алгоритмическая декомпозиция основных операций

3.1. Операция добавления (Method Add)

Операция имеет амортизированную временную сложность **O(1)**. А амортизированная она, потому что в большинстве случаев вызовов этого метода метод `Add` просто записывает в ячейку массива значение. Но в редких случаях, когда наш `Capacity == Count` (то есть место кончилось), эта операция будет **O(n)** - где `n` это количество элементов `Count`.

Алгоритм:

1. Сравниваем текущий `_size` с длиной массива `_items`.
2. Опционально: Если `_size == _items.Length` (буфер полон), инициируется процедура расширения (`Resize`) (как раз в этом случае будет **O(n)**).
3. Значение записывается в ячейку массива по индексу `_size`.
4. Значение `_size` увеличивается на 1.



На примере выше показано добавление нового элемента **"9"**. Элемент "9" добавляется в индекс `_size`, который равен **5**.

3.2. Процедура расширения (Method Resize/EnsureCapacity)

Стратегией расширения будет геометрическая прогрессия. *Обычно емкость удваивается*. Это минимизирует количество дорогих операций копирования памяти.

> В задании указано выбрать самостоятельно на\во сколько увеличивать. Если вы дочитали до сюда - поздравляю, но рекомендую попробовать увеличивать на +1, +10 или умножить в X раз на свое усмотрение для понимания разницы.

Алгоритм:

1. Определяется новая емкость. Если текущая равна 0, устанавливается дефолтная (обычно 4). В противном случае: `NewCapacity = CurrentCapacity * 2`.
2. Создается новый массив `T[]` с размером `NewCapacity`.
3. Данные из старого массива копируются в новый (обычно используется `Array.Copy`).
4. Внутреннее поле `_items` переназначается на новый массив. Старый массив попадает под сборку мусора (GC).

3.3. Доступ по индексу (`Indexer this[int index]`)

Обеспечивает доступ к элементам за время **O(1)**.

Логика:

1. Проверяется условие `index ⩾ 0 && index < _size`.

- **Важно:** Проверка идет именно по `_size` (количество добавленных элементов), а не по `Capacity` (емкость). Обращение к "мусорным" ячейкам буфера (между `_size` и `Capacity`) запрещено. Технически эти ячейки в нашей реализации списка будут заняты стандартными значениями типа данных (для `int` это 0 кст).

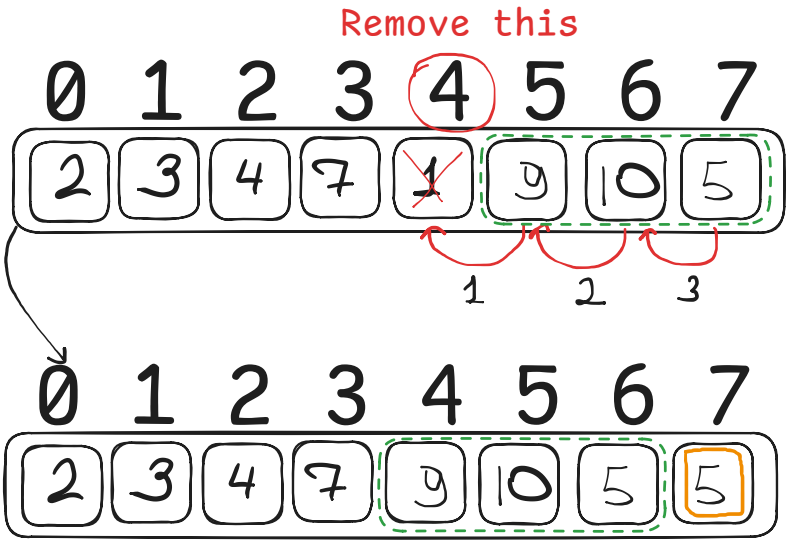
2. При нарушении границ выбрасывается исключение `ArgumentOutOfRangeException`.
3. Возвращается или изменяется значение `_items[index]`.

3.4. Операция удаления (Method RemoveAt)

Является самой ресурсоемкой операцией, сложность **O(n)**, так как требует сдвига массива для сохранения непрерывности данных.

Алгоритм:

1. Проверка корректности индекса (см. выше, попадание в диапазон)
2. Определяется количество элементов, находящихся справа от удаляемого индекса.
3. Блок элементов справа от индекса копируется на одну позицию влево, перезаписывая удаляемый элемент.
4. Значение `_size` уменьшается на 1.
5. Последний элемент массива (теперь ставший дубликатом) принудительно зануляется (`default(T)`). Это необходимо для корректной работы Garbage Collector, чтобы предотвратить утечки памяти (если `T` — ссылочный тип).



На примере выше показано удаление элемента с индексом 4. По сути, мы должны используя `Array.Copy` скопировать значения из `index+1` в `index`, повторить столько раз, сколько элементов справа. Технически, в нашей реализации значение 5 на индексе 7 останется в массиве. Но из-за `_size-1` мы не сможем получить к нему доступ из-за наших проверок на валидность индекса.

3.5. Операция вставки по индексу (Method Insert)

Операция позволяет поместить элемент в произвольную позицию списка. В отличие от метода `Add`, который всегда добавляет элемент в конец, `Insert` требует подготовки свободного слота внутри массива. Временная сложность операции составляет **O(n)**, так как требуется перемещение части элементов.

Алгоритм:

1. Проверяется условие index < 0 || index > _size.
   - **Важно**: Допустимым является значение `index == _size`. В этом случае поведение метода эквивалентно вызову `Add` (вставка в конец).
2. Перед вставкой необходимо убедиться, что буфер имеет свободное место. Если `_size == _items.Length`, инициируется процедура `Resize`.
3. Если вставка производится не в конец списка, необходимо освободить ячейку с индексом `index`.
   - Блок элементов, начиная с позиции `index` и до конца массива (`_size - 1`), копируется на одну позицию **вправо**.

    - Порядок копирования критичен во избежание перезаписи данных (в ручных циклах перебор идет с конца).
4. В освобожденную ячейку `_items[index]` записывается новое значение.
5. Значение `_size` увеличивается на 1.