Структуры данных

Когда мы пишем программы, нам постоянно нужно хранить и обрабатывать данные. Для этого используются **структуры данных** — способы организации информации в памяти.

Разные структуры подходят для разных задач. На этом занятии мы рассмотрим:

- **Массив**
- **Стек**
- **Очередь**
- **Словарь**
- **Связный список**
- **Кортеж**

Важно не только уметь пользоваться этими структурами, но и понимать, **как они устроены внутри**, ведь это напрямую влияет на эффективность программы. Хотя все структуры данных, которые мы рассматриваем в этой лекции, уже реализованы в C# и предоставляются в стандартной библиотеке (`Stack<T>` , `Queue<T>` , `LinkedList<T>` , `Dictionary<K,V>`), понимание их внутреннего устройства помогает:

- выбирать наиболее подходящую структуру для конкретной задачи;
- оценивать производительность операций добавления, удаления и поиска;
- при необходимости реализовывать собственные структуры с оптимальными характеристиками.

1. Стек (Stack)

Стек (Stack) — это структура данных, работающая по принципу **LIFO (Last In — First Out)**: последним пришёл — первым вышел.

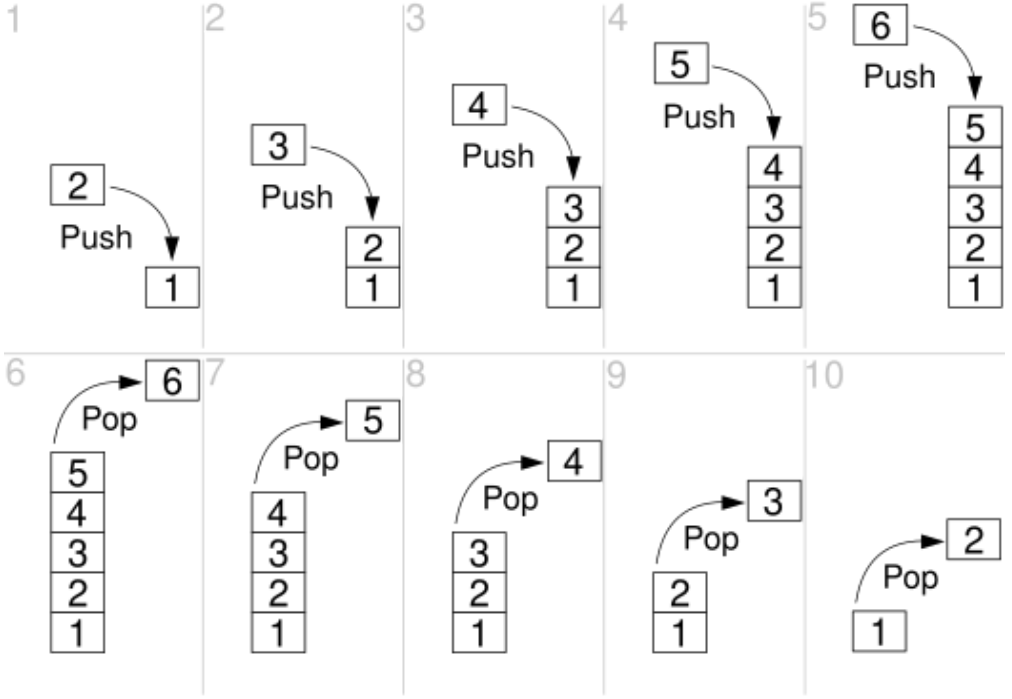
Представьте стопку тарелок:

- Вы кладёте новые тарелки сверху.
- Достаёте тоже только сверху.
- Нижняя тарелка будет доступна только после того, как уберёте все верхние.

Основные операции

У стека есть три ключевые операции:

- **Push** — положить элемент в стек.
- **Pop** — достать верхний элемент (и удалить его).
- **Peek** — посмотреть на верхний элемент (не удаляя его).



Реализация через список

```
public class MyStack<T>
{
    private List<T> _items = new List<T>();;
    public int Count => _items.Count;

    // Добавить элемент в стек
    public void Push(T item)
    {
        _items.Add(item);
    }

    // Удалить верхний элемент и вернуть его
    public T Pop()
    {
        if (Count == 0)
            throw new InvalidOperationException("Стек пуст");
        T item = _items[Count-1];
        _items.RemoveAt(Count-1);
        return item;
    }

    // Посмотреть верхний элемент
    public T Peek()
    {
        if (_items.Count == 0)
            throw new InvalidOperationException("Стек пуст");
        return _items[Count - 1];
    }
}
```

2. Очередь (Queue)

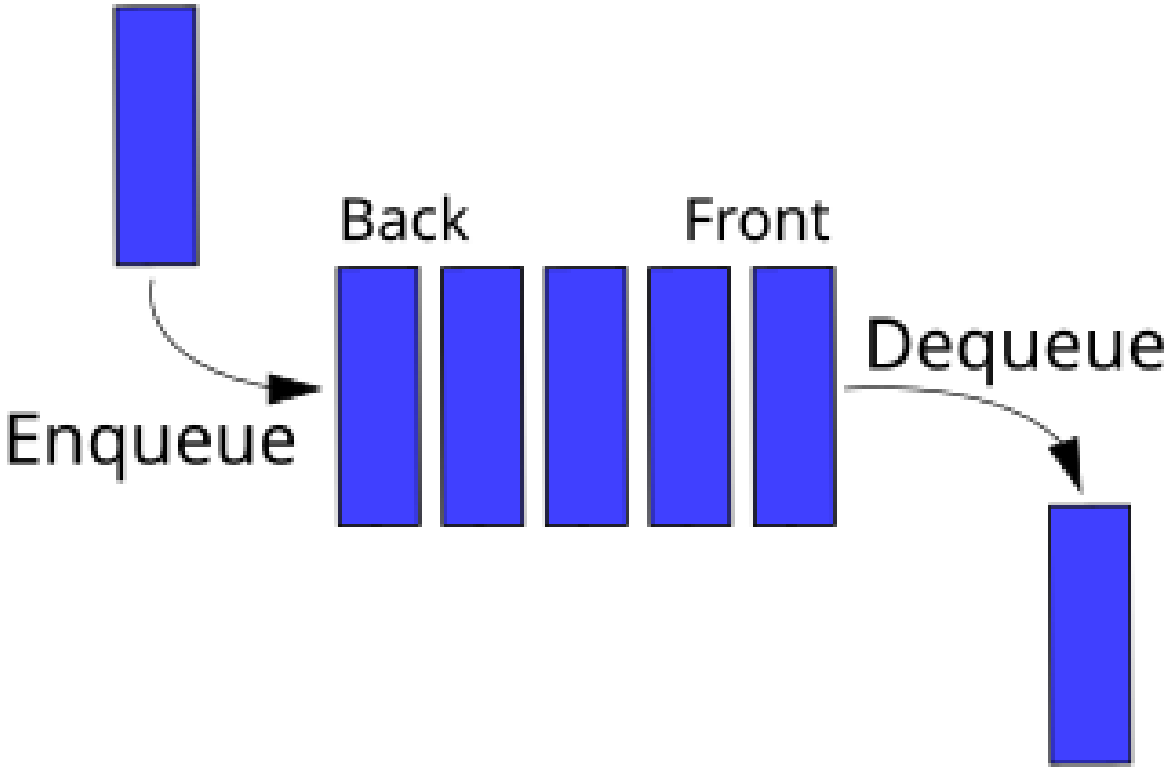
Очередь (Queue) — это структура данных, работающая по принципу **FIFO (First In — First Out)**: первым пришёл — первым вышел.

Примеры из жизни:

- Очередь в магазине: первый покупатель встал в очередь — первый обслужен.
- Очередь запросов в сервере: первый запрос — первый обработан.

Основные операции очереди

- **Enqueue** — поставить элемент в конец очереди.
- **Dequeue** — достать элемент из начала очереди.
- **Peek** — посмотреть на первый элемент, не удаляя его.



Реализация через список

```
public class MyQueue<T>
{
    private List<T> _items = new List<T>();
    public int Count => _items.Count;

    // Добавить в конец
    public void Enqueue(T item)
    {
        _items.Add(item);
    }

    // Удалить из начала
    public T Dequeue()
    {
        if (Count == 0)
            throw new InvalidOperationException("Очередь пуста");
        T value = _items[0];
        _items.RemoveAt(0);
        return value;
    }

    // Посмотреть первый элемент
    public T Peek()
    {
        if (Count == 0)
            throw new InvalidOperationException("Очередь пуста");
        return _items[0];
    }
}
```

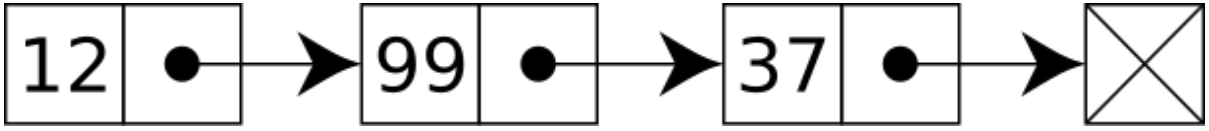
Проблема

При выполнении операции **Dequeue** элемент удаляется с начала списка. Это неэффективно, потому что после удаления первый элемент освобождает место, и все остальные элементы нужно «подвинуть» на одну позицию вперёд. То есть чем больше элементов в списке, тем дольше выполняется такая операция. В отличие от удаления из конца списка, где ничего двигать не нужно, удаление из начала всегда требует полного сдвига содержимого.

3. Связный список (LinkedList)

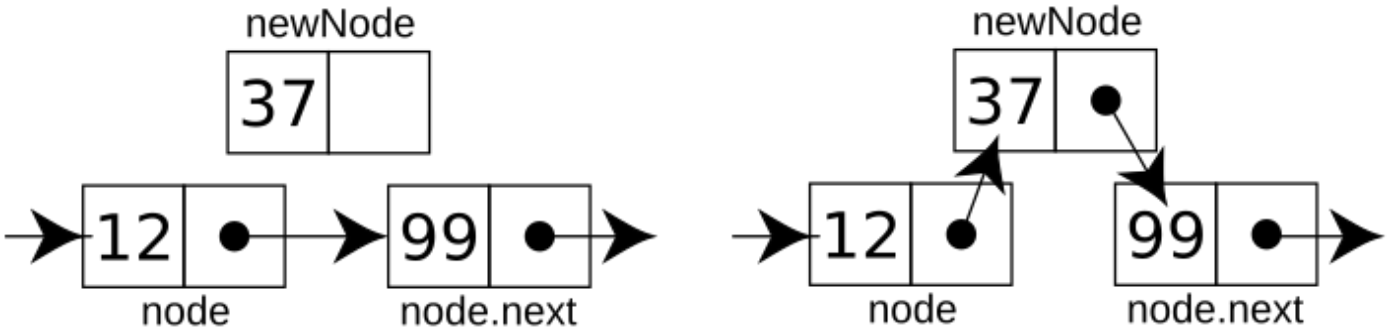
Чтобы решить проблему сдвига, используют **связный список**.
Связный список (LinkedList) — это набор узлов, где каждый узел хранит:

- **значение** (Value),
- **ссылку на следующий элемент** (Next).
  В случае двусвязного списка добавляется ещё ссылка **Prev** на предыдущий узел.

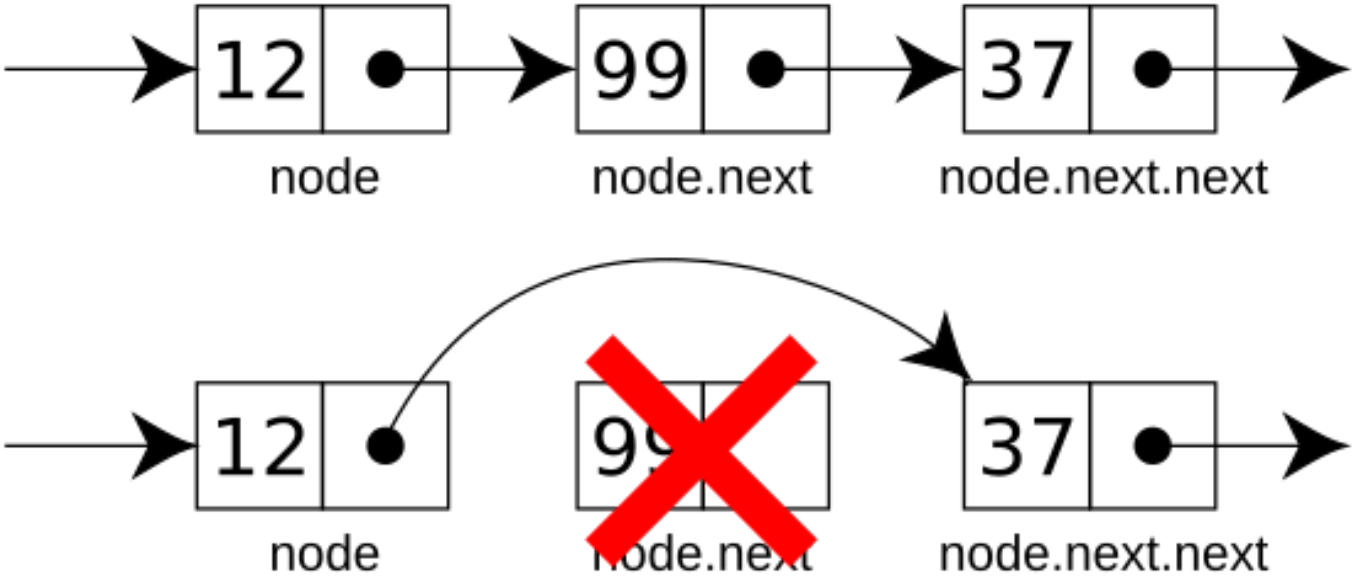


Преимуществом связного списка по сравнению с обычным списком является эффективное добавление и удаление элементов:

- Добавление элемента:



- Удаление элемента:



Очередь на связном списке

Теперь реализуем очередь на основе связного списка:

```
public class Node
{
    // Значение, которое хранит узел
    public T Value { get; set; }
    // Ссылка на следующий элемент списка
    public Node Next { get; set; }
    // Конструктор: создаём узел с указанным значением
    public Node(T value)
    {
        Value = value;
        Next = null;
        // по умолчанию ни на кого не указывает
    }
}

public class MyQueue<T>
{
    // Первый элемент списка (голова)
    private Node<T> _head;
    // Последний элемент списка (хвост)
    private Node<T> _tail;
    private int _count;

    public int Count => _count;

    public void Enqueue(T item)
    {
        var node = new Node<T>(item);
        if (_tail == null)
```

```
            _head = _tail = node;
        else
        {
            _tail.Next = node;
            _tail = node;
        }
        _count++;
    }

    public T Dequeue()
    {
        if (_head == null)
            throw new InvalidOperationException("Очередь пуста");
        T value = _head.Value;
        _head = _head.Next;
        if (_head == null)
            _tail = null;
        _count--;
        return value;
    }

    public T Peek()
    {
        if (_head == null)
            throw new InvalidOperationException("Очередь пуста");
        return _head.Value;
    }
}
```

4. Кортежи (Tuples)

Кортеж — это упорядоченный набор нескольких значений, объединённых в один объект без необходимости создавать отдельный класс или структуру.

Старый синтаксис: `Tuple<T1, T2>`

```
Tuple<int, string> data = new Tuple<int, string>(1, "Hello");

int number = data.Item1;
string text = data.Item2;
```

Изначально в C# были только `Tuple<>` (например, `Tuple<int, string>`), но они имели недостатки: неудобный доступ к элементам (`Item1` , `Item2` , ...) и менее читаемый код.

Позже появились `ValueTuple` и новый синтаксис в скобках (`(int, string)`), который позволяет называть элементы (`(int Id, string Name)`) и делает работу с кортежами более удобной и наглядной.

Новый синтаксис: ValueTuple (с C# 7.0)

```
(int id, string name) user = (1, "Alice");

Console.WriteLine(user.id);   // 1
Console.WriteLine(user.name); // Alice
```

С помощью кортежа можно вернуть несколько значений из метода:

```
public static (int sum, int product) Calculate(int a, int b)
{
    return (a + b, a * b);
}

// Использование:
var result = Calculate(3, 4);
Console.WriteLine(result.sum);     // 7
Console.WriteLine(result.product); // 12
```

5. Словарь

Словарь (Dictionary<K,V>) — это коллекция, которая хранит данные в виде **пар ключ–значение**, где каждому ключу соответствует одно значение. Ключи должны быть уникальными, а доступ к данным осуществляется очень быстро.

Словари применяются там, где важен быстрый поиск по ключу:

- хранение и поиск настроек по имени;
- подсчёт количества встречающихся элементов (например, слов в тексте);
- отображение идентификаторов на объекты (id → пользователь);
- кэширование данных для ускорения работы программы.

Самый простой способ реализовать словарь — использовать список кортежей, в которых одно значение будет ключом, а другое значением:

```
//K - тип ключа, V - тип значения
class MyDictionary<K, V>
{
    //список пар ключ-значение
    List<(K, V)> collection = new List<(K, V)>();

    public V this[K key]
    {
        get
        {
            foreach (var item in collection)
            {
                if (item.Item1.Equals(key))
                    return item.Item2;
            }
            throw new KeyNotFoundException();
        }
```

```
        set
        {
            bool valueFound = false;
            //ищем существующую запись
            for (int i = 0; i < collection.Count; i++)
            {
                if (collection[i].Item1.Equals(key))
                {
                    collection[i] = (key, value);
                    valueFound = true;
                    break;
                }
            }
            //если не находим - создаем новую
            if (!valueFound)
            {
                collection.Add((key, value));
            }
        }
    }
}
```

Проблема

Если реализовать словарь как список кортежей `(ключ, значение)`, то при поиске нужного элемента придётся проходить весь список и проверять каждый ключ по очереди.

Это означает, что чем больше данных хранится, тем дольше будет выполняться поиск. То же самое касается удаления и проверки наличия элемента — приходится «перебирать» список.

Поэтому такой подход работает, но становится неудобным и медленным при большом количестве элементов.

Чтобы ускорить поиск, используется **хэш-функции**.

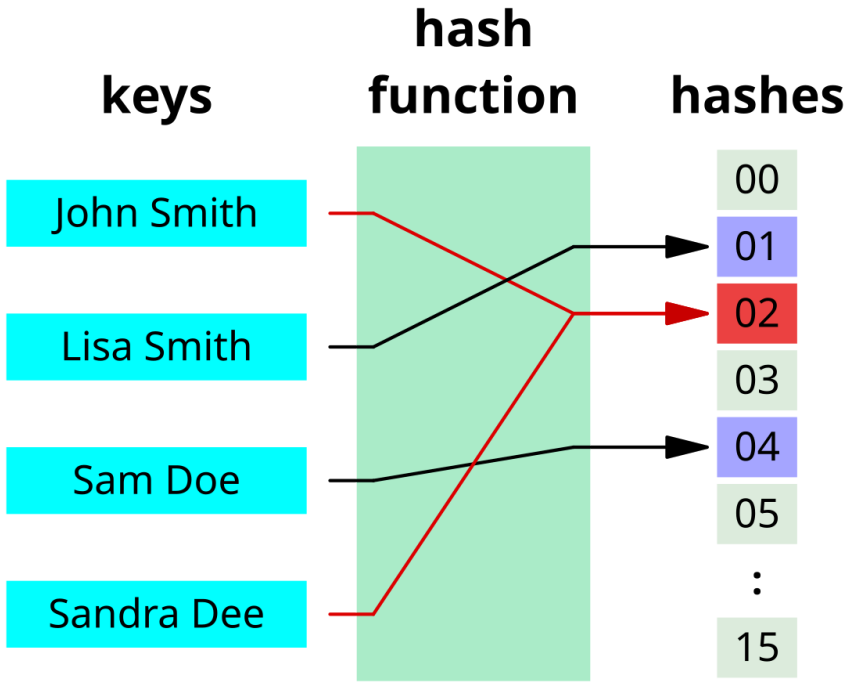
Что такое хэш-функция?

Хэш-функция — это алгоритм, который преобразует входные данные любого размера (например, строку или файл) в число фиксированной длины — **хэш**.

Пример: пусть хэш-функция выдаёт значения от 0 до 15 (4 бита).

- Строка `"John Smith"` может получить хэш `02`.
- Строка `"Sandra Dee"` тоже может получить хэш `02`.

Такая ситуация называется **коллизией**, и она неизбежна, потому что разных строк бесконечно много, а возможных хэшей — ограниченное количество.



Простейшая хэш-функция

Для наглядности напишем простую хэш-функцию на C#:

```
int Hash(string text)
{
    int hash = 0;
    foreach (var c in text)
        hash += (int)c;    // суммируем коды символов
    return hash % 256;     // берём остаток от деления на 256
}
```

Эта функция всегда возвращает число от `0` до `255` (1 байт).
Но у неё есть проблема: разные строки могут давать одинаковый хэш. Например:

- `"abc"` и `"cab"` → одинаковый хэш (порядок символов не учитывается).

Поэтому такие простые функции непрактичны, и в реальности применяются более надёжные алгоритмы, например **SHA-256**.

Где применяются хэши

1. **Файловые хранилища**
   Вместо сравнения файлов целиком можно сравнивать их хэши. Это быстрее и позволяет находить дубликаты.
   - Если хэши разные → файлы точно разные.
   - Если хэши одинаковые → нужно сравнить файлы полностью (на случай коллизии).
2. **Отслеживание изменений в файлах**
   Если изменить даже один байт в файле, его хэш изменится радикально. Это удобно для контроля целостности данных.
3. **Хранение паролей**
   Пароли в базе данных не хранят в открытом виде, а записывают только их хэши.
   - Пользователь вводит пароль.
   - Система хэширует его и сравнивает с хэшем в базе.
   - Если хэши совпадают → пароль верный.

Чтобы повысить безопасность, используется **соль** — случайное число, добавляемое к паролю перед хэшированием. Это усложняет подбор паролей при взломе базы данных.

Хэш-таблица

Ещё одно важное применение хэшей — это структура данных **хэш-таблица**, которая позволяет ускорить работу словаря.
Идея в том, что для каждого ключа вычисляется его **хэш**, и именно по хэшу мы находим место для хранения значения.

Можно думать об этом так:

- ключ → хэш-функция → индекс в массиве → значение.

Таким образом, доступ к элементам становится таким же быстрым, как доступ к элементу массива по индексу.

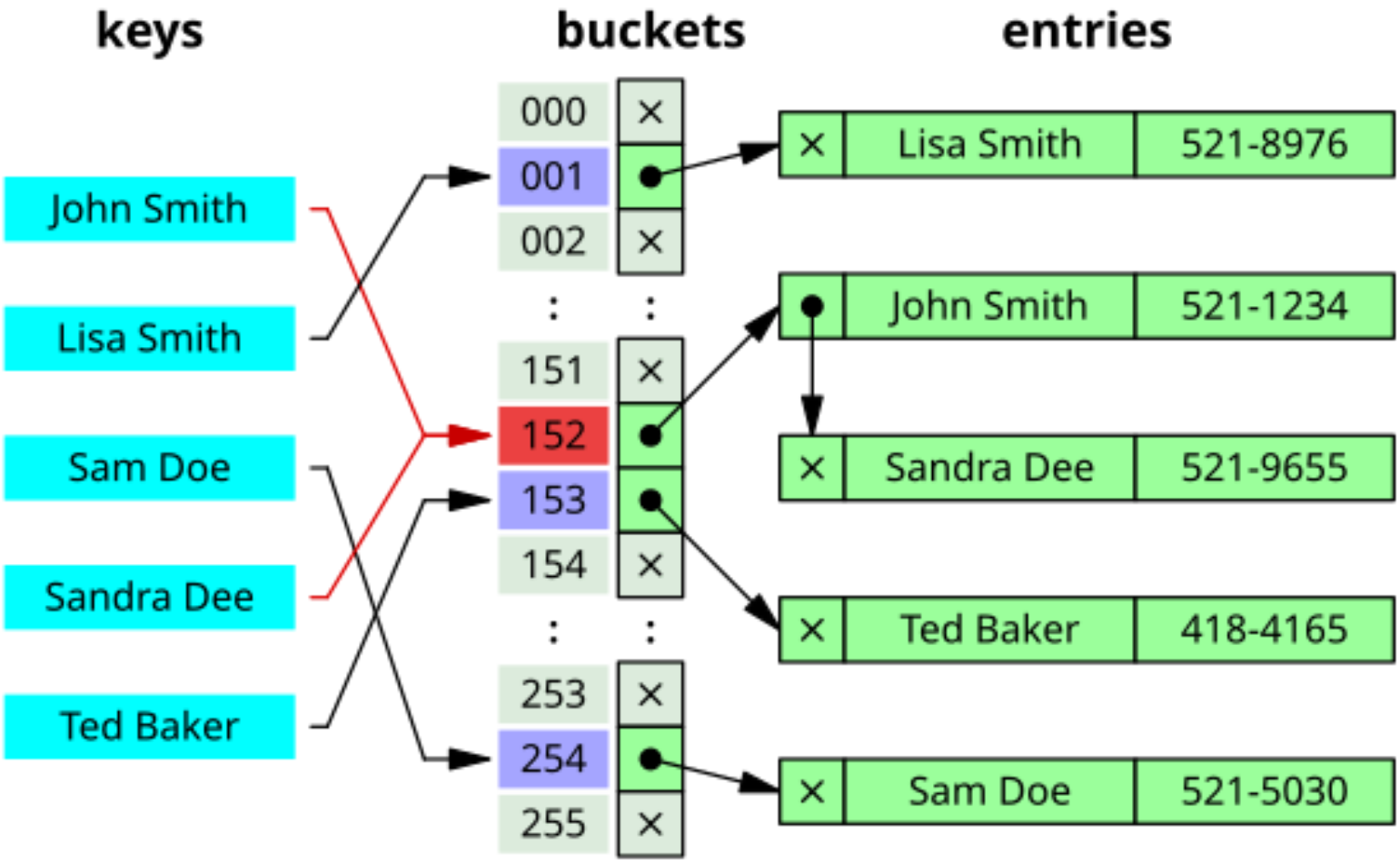
Проблема коллизий

В том случае, если разные ключи дают одинаковый хэш, в ячейке массива хранится не одно значение, а, **связанный список** элементов с одинаковым хэшем.

Тогда поиск работает в два шага:

1. Находим список по хэшу.
2. Перебираем элементы внутри этого списка.

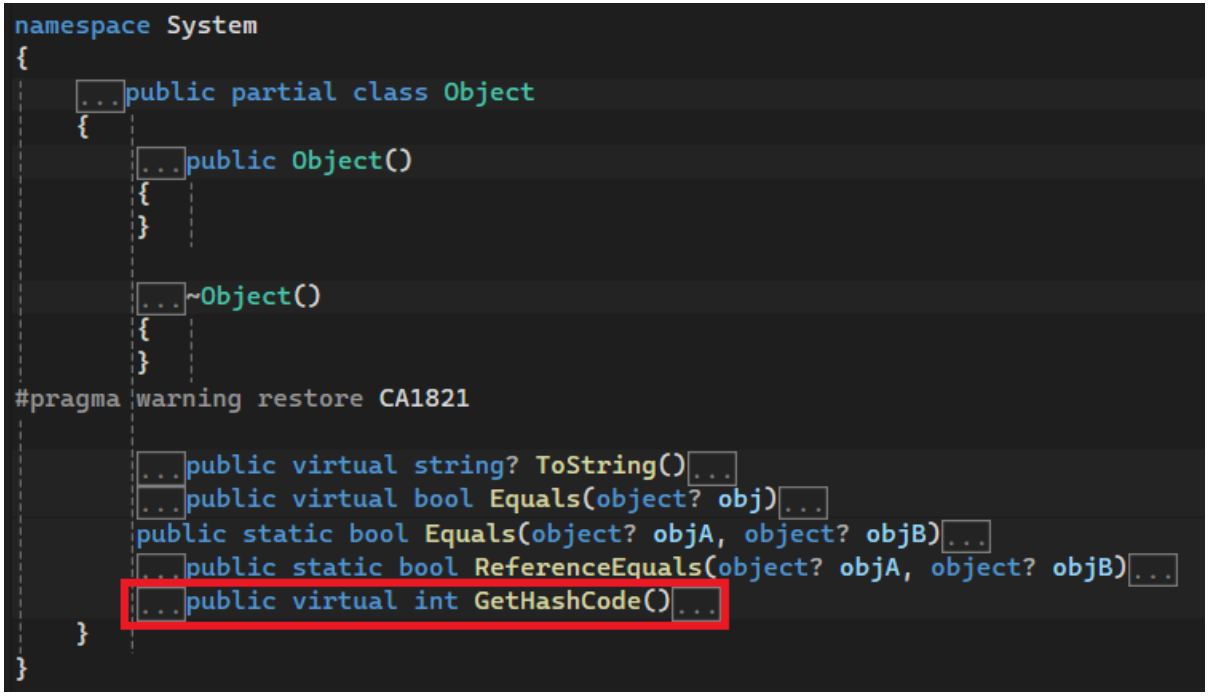
Хотя в худшем случае (если у всех ключей одинаковый хэш) словарь будет работать так же медленно, как и на списке, на практике хорошие хэш-функции делают такие ситуации очень редкими.


keys
buckets
entries
John Smith
Lisa Smith
Sam Doe
Sandra Dee
Ted Baker
000
001
002
151
152
153
154
253
254
255
Lisa Smith
521-8976
John Smith
521-1234
Sandra Dee
521-9655
Ted Baker
418-4165
Sam Doe
521-5030

GetHashCode

Ключами словаря могут быть не только строки, но и любые другие типы данных. Возникает вопрос: **как для них вычисляется хэш?**

В .NET за это отвечает метод `GetHashCode`, который определён в базовом классе `object`.



Методы `GetHashCode` и `Equals` тесно связаны между собой:

- если два объекта равны с точки зрения `Equals`, то их `GetHashCode` тоже должен возвращать одинаковое значение;
- обратное правило не обязательно: два разных объекта могут иметь одинаковый хэш (это и есть коллизия).

По умолчанию у **классов** хэш вычисляется по ссылке на объект;

Пример

Рассмотрим класс точки с координатами `X` и `Y`.
Если мы хотим использовать точки как ключи в словаре, нужно переопределить методы `Equals` и `GetHashCode`, чтобы точки с одинаковыми координатами считались равными.

```
class Point
{
    public int X { get; set; }
    public int Y { get; set; }

    // Сравнение объектов по координатам
    public override bool Equals(object obj)
    {
        var point = obj as Point;
        return point != null && X == point.X && Y == point.Y;
    }

    // Хэш вычисляется из координат
    public override int GetHashCode()
    {
        return HashCode.Combine(X, Y);
    }
}
```

Теперь словарь будет корректно работать даже с разными объектами, но одинаковыми по содержимому:

```
var point1 = new Point { X = 1, Y = 1 };
var dictionary = new Dictionary<Point, string>();

dictionary[point1] = "Test";
Console.WriteLine(dictionary[point1]); // Test

var point2 = new Point { X = 1, Y = 1 };
Console.WriteLine(dictionary[point2]); // Test
```

Хотя `point1` и `point2` — это **разные объекты в памяти**, словарь воспринимает их как один и тот же ключ, потому что их `Equals` возвращает `true`, а `GetHashCode` выдаёт одинаковое значение.

Пример словаря на основе хэш-таблицы

```
public class MyDictionary<K, V>
{
    private readonly int _capacity;
    private readonly LinkedList<(K Key, V Value)>[] _buckets;

    public MyDictionary(int capacity = 16)
    {
        _capacity = capacity;
        _buckets = new LinkedList<(K, V)>[_capacity];
        for (int i = 0; i < _capacity; i++)
            _buckets[i] = new LinkedList<(K, V)>();
    }

    // Вычисляем индекс бакета через GetHashCode
    private int GetBucketIndex(K key)
    {
        return Math.Abs(key.GetHashCode()) % _capacity;
    }

    // Индексатор для чтения и записи по ключу
    public V this[K key]
    {
        get
        {
            int index = GetBucketIndex(key);
            foreach (var pair in _buckets[index])
            {
                if (pair.Key.Equals(key))
                    return pair.Value;
            }
```

```
            throw new KeyNotFoundException();
        }
        set
        {
            int index = GetBucketIndex(key);
            var list = _buckets[index];
            var node = list.First;

            while (node != null)
            {
                if (node.Value.Key.Equals(key))
                {
                    node.Value = (key, value); // обновляем значение
                    return;
                }
                node = node.Next;
            }

            // Если ключ не найден — добавляем новую пару
            list.AddLast((key, value));
        }
    }
}
```

6. Интерфейс IComparable и метод CompareTo

На прошлой лекции мы изучали метод `Equals`, который позволяет определить, **равны ли два объекта**.
Однако иногда нужно не просто узнать равенство, а определить **порядок объектов**: какой меньше, какой больше. Для этого используется интерфейс `IComparable<T>` и его метод `CompareTo`.

Метод `CompareTo(T other)` возвращает:

- **отрицательное число**, если текущий объект меньше `other`;
- **0**, если объекты равны;
- **положительное число**, если текущий объект больше `other`.

Пример без CompareTo

Допустим, у нас есть класс `Point` с координатами, и мы хотим вручную определить, какая точка "меньше" другой по X:

```
class Point
{
    public int X { get; set; }
    public int Y { get; set; }
}

var point1 = new Point { X = 3, Y = 4 };
var point2 = new Point { X = 5, Y = 2 };

// Попытка сравнить напрямую
// if (point1 < point2) // ❌ Ошибка: оператор < не определен для Point
```

Мы не можем использовать стандартные операторы `<` или `>`, потому что класс `Point` не знает, как себя сравнивать.

Пример: реализация IComparable

Теперь реализуем интерфейс и метод `CompareTo`, чтобы точку можно было сравнивать по X:

```
class Point : IComparable<Point>
{
    public int X { get; set; }
    public int Y { get; set; }

    public int CompareTo(Point other)
    {
        if (other == null) return 1; // текущий объект "больше" null
        return X.CompareTo(other.X); // сравниваем по X
    }

    public override bool Equals(object obj)
    {
        if (obj is not Point other) return false;
        return X == other.X && Y == other.Y;
    }

    public override int GetHashCode() => HashCode.Combine(X, Y);

    public override string ToString() => $"({X},{Y})";
}
```

Пример использования CompareTo

```
var point1 = new Point { X = 3, Y = 4 };
var point2 = new Point { X = 5, Y = 2 };

if (point1.CompareTo(point2) < 0)
    Console.WriteLine($"{point1} меньше {point2}");
else if (point1.CompareTo(point2) > 0)
    Console.WriteLine($"{point1} больше {point2}");
else
    Console.WriteLine($"{point1} равен {point2}");
```

Вывод:

```
(3,4) меньше (5,2)
```

7. Перегрузка операторов

мы можем использовать метод `Equals` для проверки **равенства объектов**. Так же можно реализовать интерфейс `IComparable<T>` и метод `CompareTo` для сравнения. Однако для удобства и более привычного синтаксиса можно перегружать стандартные операторы (`==` , `!=` , `<` , `>` , `<=` , `>=`) и определять арифметические операции для своих типов.

Перегрузка `==` и `!=`

```
class Point
{
    public double X { get; set; }
    public double Y { get; set; }

    public override bool Equals(object obj)
    {
        if (obj is not Point p) return false;
        return X == p.X && Y == p.Y;
    }

    public override int GetHashCode() => HashCode.Combine(X, Y);

    public static bool operator ==(Point p1, Point p2) => p1.Equals(p2);
    public static bool operator !=(Point p1, Point p2) => !(p1 == p2);
}

var point1 = new Point { X = 1, Y = 1 };
var point2 = new Point { X = 1, Y = 1 };

Console.WriteLine(point1 == point2); // true
Console.WriteLine(point1 != point2); // false
```

Перегрузка `<`, `>`, `<=`, `>=` через CompareTo

Чтобы перегрузить операторы сравнения, удобно использовать метод `CompareTo` из `IComparable<Point>`:

```
class Point : IComparable<Point>
{
    public double X { get; set; }
    public double Y { get; set; }

    public int CompareTo(Point other)
    {
        if (other == null) return 1;
        return X.CompareTo(other.X); // сравнение по X
    }

    public override bool Equals(object obj)
    {
        if (obj is not Point p) return false;
        return X == p.X && Y == p.Y;
    }

    public override int GetHashCode() => HashCode.Combine(X, Y);

    public static bool operator <(Point p1, Point p2) => p1.CompareTo(p2) < 0;
    public static bool operator >(Point p1, Point p2) => p1.CompareTo(p2) > 0;
    public static bool operator <=(Point p1, Point p2) => p1.CompareTo(p2) <= 0;
    public static bool operator >=(Point p1, Point p2) => p1.CompareTo(p2) >= 0;
}
```

Использование:

```
var point1 = new Point { X = 2, Y = 3 };
var point2 = new Point { X = 5, Y = 1 };

if (point1 < point2)
    Console.WriteLine($"{point1.X},{point1.Y} меньше {point2.X},{point2.Y}");

if (point2 >= point1)
    Console.WriteLine($"{point2.X},{point2.Y} больше или равно {point1.X},{point1.Y}");
```

Вывод:

```
2,3 меньше 5,1
5,1 больше или равно 2,3
```

Перегрузка арифметических операторов

Аналогично можно перегружать арифметические операторы (`+` , `-` , `*`) и делать их совместимыми с другими типами:

```
public static Point operator +(Point p1, Point p2) => new Point { X = p1.X + p2.X, Y = p1.Y + p2.Y };
public static Point operator -(Point p1, Point p2) => new Point { X = p1.X - p2.X, Y = p1.Y - p2.Y };
public static Point operator *(Point p, double value) => new Point { X = p.X * value, Y = p.Y * value };
public static Point operator *(double value, Point p) => p * value;
```

Рекомендации

- Перегружайте операторы только тогда, когда это имеет **понятный смысл**.
- Для сравнения объектов используйте `Equals` или `CompareTo` , а перегруженные операторы — для удобного синтаксиса.
- Не перегружайте операторы так, чтобы поведение было **неинтуитивным**, иначе код станет запутанным.

Практическое задание

Что нужно сделать:

1. **Создайте два стека:**
   - `undoStack` для хранения выполненных команд,
   - `redoStack` для хранения отменённых команд.
2. В интерфейс `ICommand` добавьте метод:

   ```
   void Unexecute();
   ```

   чтобы каждая команда умела отменять свои действия.
3. **При выполнении любой команды** (например `add`, `delete`, `update`, `status`) сохраняйте её в `undoStack`.
4. **Реализуйте команду `undo`:**
   - взять последнюю команду из `undoStack`,
   - отменить её действие,
   - положить её в `redoStack`.
5. **Реализуйте команду `redo`:**
   - взять команду из `redoStack`,
   - снова выполнить её действие,
   - вернуть обратно в `undoStack`.
6. Делайте коммиты после **каждого изменения**. Один большой коммит будет оцениваться в два раза ниже.
7. Обновите **README.md** — добавьте описание новых возможностей программы.
8. Сделайте push изменений в GitHub.