Web-разработка на C# и платформе Microsoft .NET

Коллекции

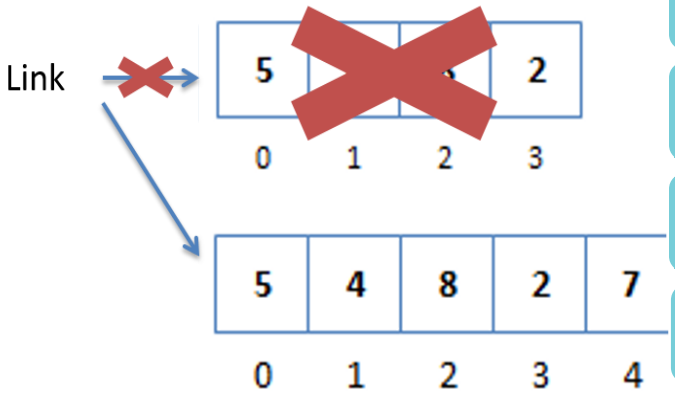
План занятия

- Способы хранения информации
- Основные коллекции
- Базовые интерфейсы коллекций

Критерии оценки

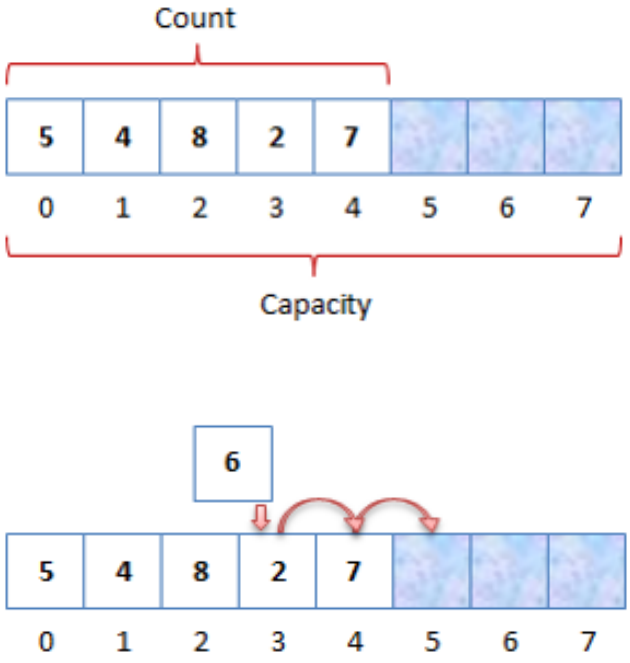
1. Затраты памяти
2. Время произвольного доступа
3. Время последовательного доступа
4. Скорость поиска
5. Скорость упорядочивания
6. Скорость добавления/удаления в начало/конец
7. Скорость добавления/удаления в произвольной позиции

Массив



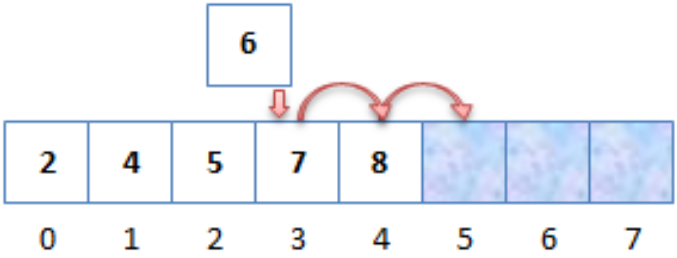
- Память
- Произвольный доступ
- Последовательный доступ
- Поиск
- Упорядочивание
- Добавление/удаление в начало/конец
- Добавление/удаление в произвольной позиции

Динамический массив



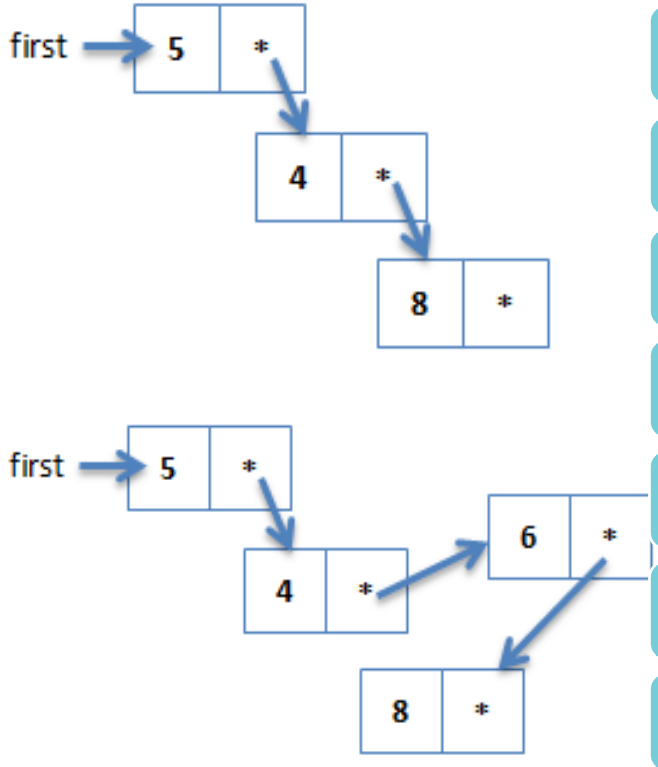
- Память
- Произвольный доступ
- Последовательный доступ
- Поиск
- Упорядочивание
- Добавление/удаление в начало/конец
- Добавление/удаление в произвольной позиции

Сортированный массив



- Память
- Произвольный доступ
- Последовательный доступ
- Поиск
- Упорядочивание
- Добавление/удаление в начало/конец
- Добавление/удаление в произвольной позиции

Связный список



- Память
- Произвольный доступ
- Последовательный доступ
- Поиск
- Упорядочивание
- Добавление/удаление в начало/конец
- Добавление/удаление в произвольной позиции

Хеш таблицы

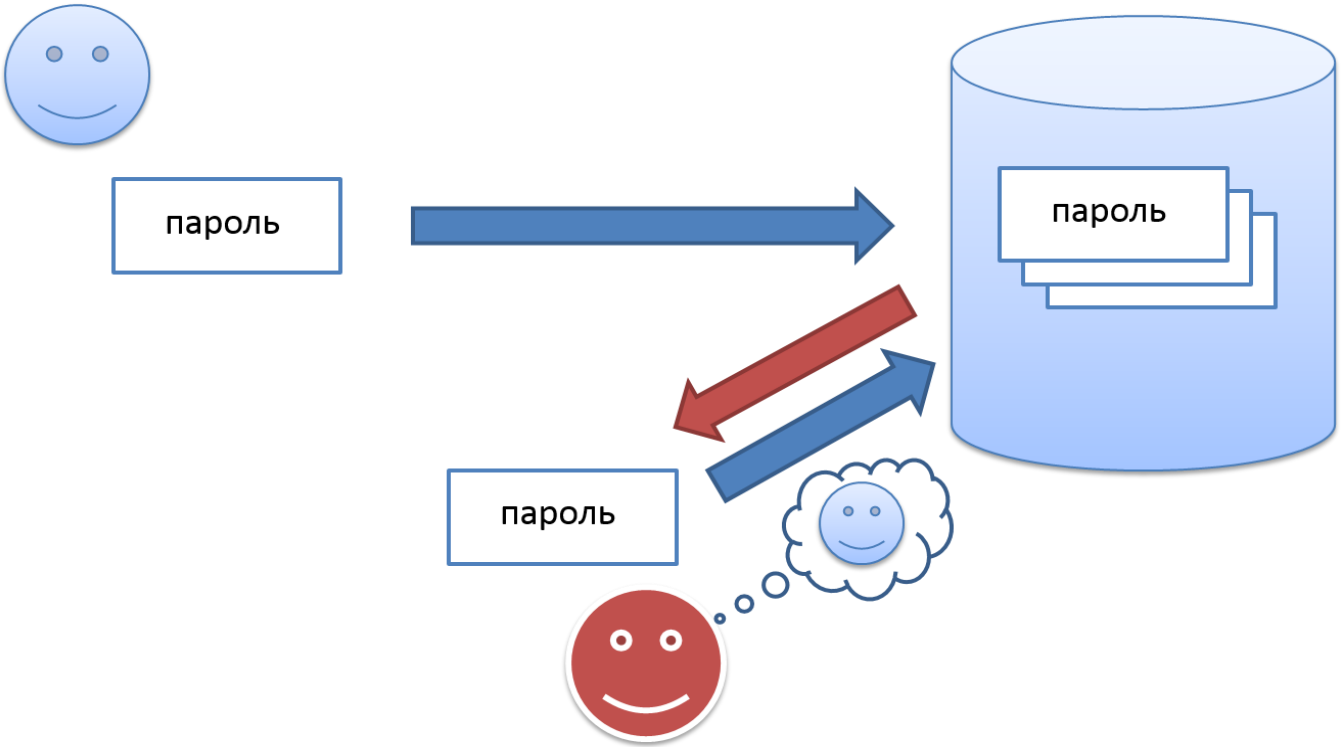
Иванов	данные
Петров	данные
Сидоров	данные

- Память
- Произвольный доступ
- Последовательный доступ
- Поиск
- Упорядочивание
- Добавление/удаление в начало/конец
- Добавление/удаление в произвольной позиции

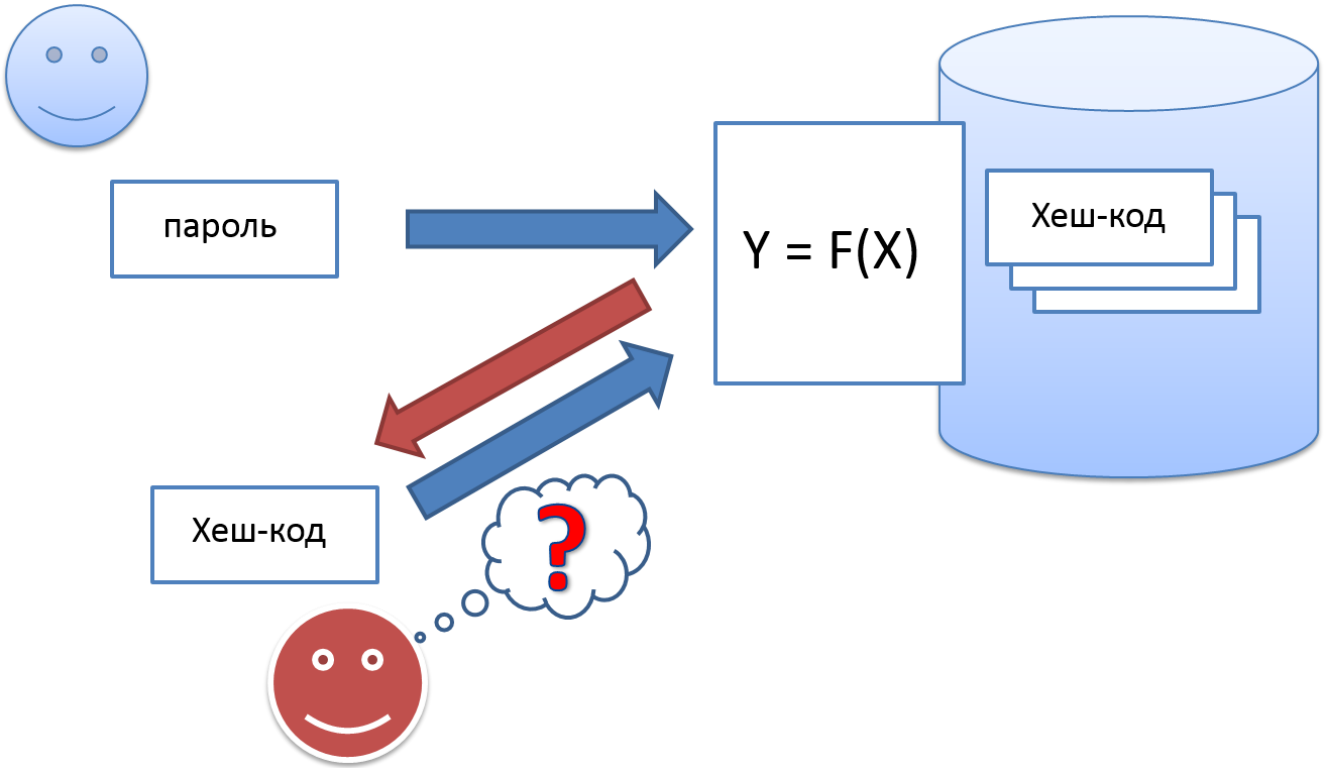
Использование хеширования

1. • Проверка целостности данных (контрольные суммы)
2. • Проверка паролей
3. • Быстрый поиск данных по ключу
   - Базы данных
   - Ассоциативные массивы

Проверка паролей



Проверка паролей



Сравнение

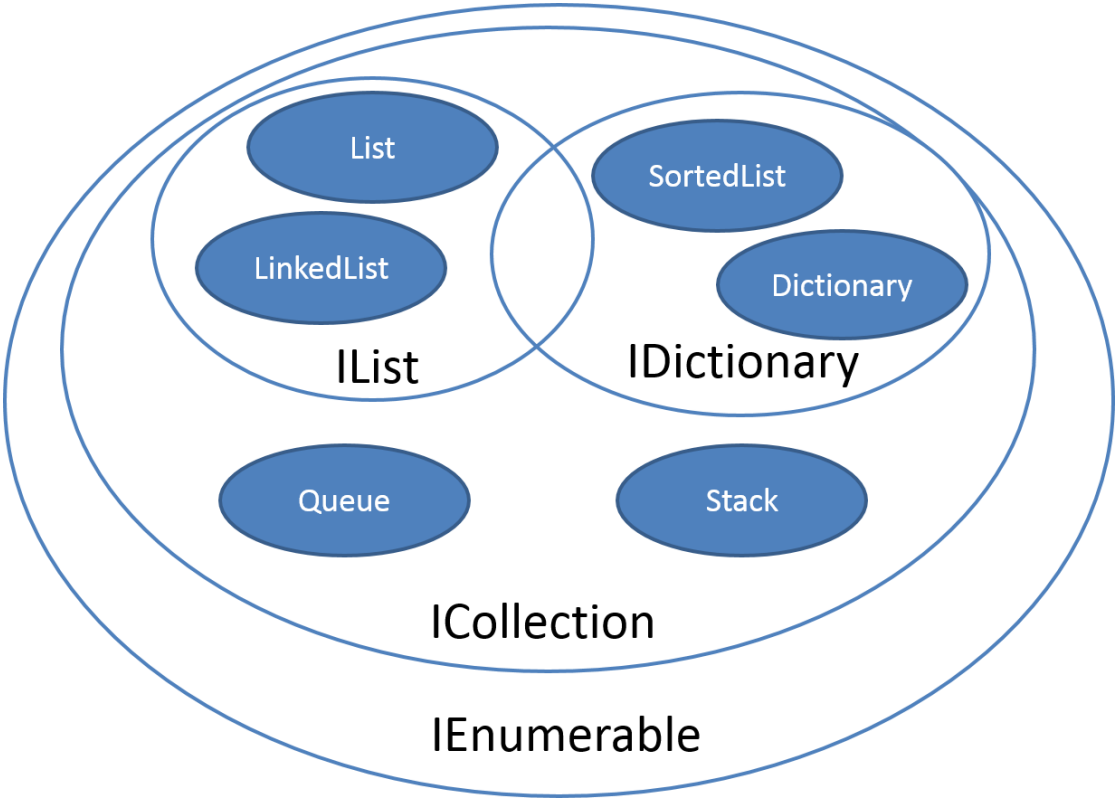
Способ хранения	Время доступа к элементу	Время поиска	Добавление в конец	Добавление в произвольную позицию	Затраты памяти
Массив	const	~N	N/A	N/A	N
Массив с запасом	const	~N	const	~N	Ближайшая степень 2, не меньшая, чем N
Упорядоченный массив	const	~$\log_2(N)$	N/A	N/A	N
Список	~N	~N	const	const	N + N * размер ссылки
Хеш-таблица (dictionary)	const	const	N/A	N/A	Ближайшее простое число, не меньшее, чем степень 2, не меньшая, чем 1.39 * N

Основные интерфейсы коллекций

- Перечисление: IEnumerable и IEnumerable<T>
- Коллекция: ICollection и ICollection<T>
- Список: IList и IList<T>
- Множество: ISet<T>
- Словарь: IDictionary и IDictionary<TKey, TValue>

Обобщённые интерфейсы располагаются в пространстве имён System.Collections.Generic, а необобщённые —System.Collections.

Интерфейсы



Перечислитель

Предоставляет возможность перебирать элементы какой-либо последовательности.

```
public interface IEnumerator
{
    // Получение текущего элемента
    object Current { get; }

    // Переход к следующему элементу
    bool MoveNext();

    // Сброс к начальному положению
    void Reset();
}
public interface IEnumerator<T> : IEnumerator
{
    // Получение текущего элемента
    T Current { get; }
}
```

Перечисление

Базовый интерфейс для последовательности элементов.
Гарантирует возможность их перебора.

```
public interface IEnumerable
{
    // Получение (создание) перечислителя
    IEnumerator GetEnumerator();
}

public interface IEnumerable<out T> : IEnumerable
{
    // Получение (создание) перечислителя
    IEnumerator<T> GetEnumerator();
}
```

Обобщенная коллекция

```
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
    // Количество элементов в коллекции
    int Count { get; }

    // Добавление элемента
    void Add(T item);

    // Очистка коллекции
    void Clear();

    // Проверка наличия элемента
    bool Contains(T item);

    // Копирование всех элементов коллекции в массив
    void CopyTo(T[] array, int arrayIndex);

    // Удаление элемента
    bool Remove(T item);
}
```

Обобщенный список

Обеспечивает доступ по индексу

```
public interface IList<T> : ICollection<T>, IEnumerable<T>,
                            IEnumerable
{
    // Произвольный доступ по индексу
    T this[int index] { get; set; }

    // Поиск элемента
    int IndexOf(T item);

    // Вставка элемента
    void Insert(int index, T item);

    // Удаление элемента по индексу
    void RemoveAt(int index);
}
```

Обобщенный словарь

```
public interface IDictionary<TKey, TValue> : ICollection<KeyValuePair<TKey, TValue>>,
                                IEnumerable<KeyValuePair<TKey, TValue>>, IEnumerable
{
    // Коллекция ключей
    ICollection<TKey> Keys { get; }

    // Коллекция значений
    ICollection<TValue> Values { get; }

    // Произвольный доступ по ключу
    TValue this[TKey key] { get; set; }

    // Добавление пары ключ-значение
    void Add(TKey key, TValue value);

    // Проверка наличия ключа
    bool ContainsKey(TKey key);

    // Удаление ключа
    bool Remove(TKey key);

    // Безопасное получение значения
    bool TryGetValue(TKey key, out TValue value);
}
```

Зачем использовать обобщенные коллекции?

- Чтобы не выполнять постоянное приведение типа при чтении элементов.
- Чтобы не упаковывать значимые типы.
- Для дополнительной типобезопасности.

- Иначе говоря, в 99% случаев **следует использовать обобщённые коллекции**.

Основные обобщенные коллекции в C#

List<T> — динамический массив
LinkedList<T> — двунаправленный связный список
Queue<T> — очередь
Stack<T> — стек
HashSet<T>, SortedSet<T>—множества
Dictionary<T>, SortedList<T>, SortedDictionary<T> — словари пар ключ—значение

List<T>

Динамический массив

```
List<string> strings = new List<string>();

strings.Add("Hello");

string[] source = { "A", "B", "CCC", "example" };
strings.AddRange(source);

strings[0] = "dfsdg";
strings.Insert(0, "sadaf");
```

LinkedList<T>

```
public class LinkedList<T> : ICollection<T>, IEnumerable<T>, ICollection, IEnumerable
{
    public LinkedListNode<T> First { get; }
    public LinkedListNode<T> Last { get; }

    public void AddAfter(LinkedListNode<T> node, LinkedListNode<T> newNode);
    public LinkedListNode<T> AddAfter(LinkedListNode<T> node, T value);

    public void AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode);
    public LinkedListNode<T> AddBefore(LinkedListNode<T> node, T value);

    public void AddFirst(LinkedListNode<T> node);
    public LinkedListNode<T> AddFirst(T value);

    public void AddLast(LinkedListNode<T> node);
    public LinkedListNode<T> AddLast(T value);

    public LinkedListNode<T> Find(T value);
    public LinkedListNode<T> FindLast(T value);

    public void Remove(LinkedListNode<T> node);
    public bool Remove(T value);
    public void RemoveFirst();
    public void RemoveLast();
}
```

Queue<T>

Очередь. Реализует принцип FIFO

```
public class Queue<T> : IEnumerable<T>, ICollection, IEnumerable
{
    // Извлечение с начала очереди
    public T Dequeue();

    // Помещение в конец очереди
    public void Enqueue(T item);

    // Просмотр начала очереди
    public T Peek();
}
```

Stack<T>

Стек. Реализует принцип LIFO

```
public class Stack<T> : IEnumerable<T>, ICollection, IEnumerable
{
    // Просмотр вершины стека
    public T Peek();

    // Извлечение с вершины стека
    public T Pop();

    // Помещение на вершину стека
    public void Push(T item);
}
```

СПАСИБО ЗА ВНИМАНИЕ