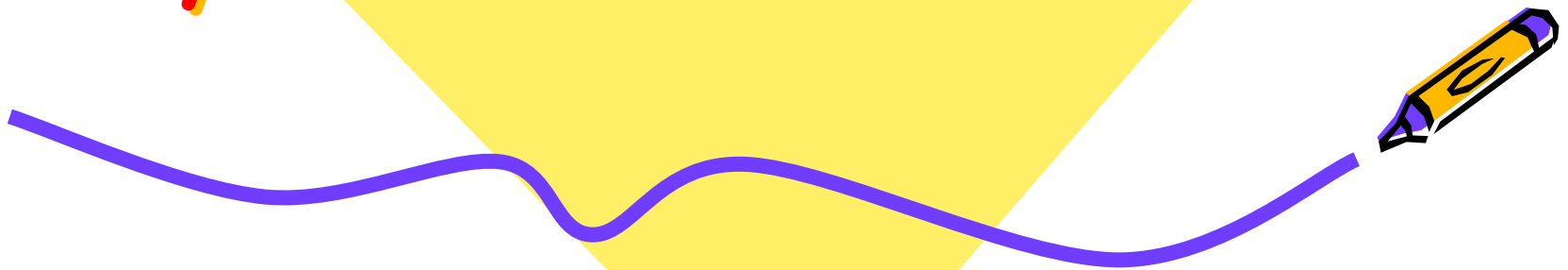


Лекция 9. Силно типизирани колекции в `System.Collections.Generic`



Силно типизирани колекции в `System.Collections.Generic`

- Както подчертахме в предната лекция, в .NET Framework 2.0 беше въведено ново именовано пространство за работа с колекции - `System.Collections.Generic`.
- Това именовано пространство съдържа интерфейси и класове, които дефинират **генерични (основни, родови)** колекции, позволяващи на потребителя да създава силно типизирани колекции, осигуряващи:
 - по-висока производителност от колекциите с общо предназначение (разположени в `System.Collections`)
 - по-висока сигурност от не-генеричните, също строго типизирани колекции (на `System.Collections.Specialized`).



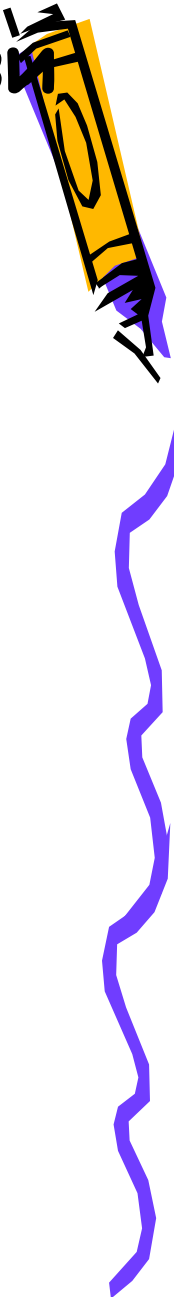
Колекции, на които ще се спрем в тази лекция:

Речникови колекции

- `SortedList<TKey, TValue>`
- `Dictionary<TKey, TValue>`
- `SortedDictionary<TKey, TValue>`

Списъчни колекции

- `List<T>`
- `Queue<T>`
- `Stack<T>`



Речници

- Всички класове, имплементиращи речникови колекции в `System.Collections.Generic` наследяват интерфейс `IDictionary<TKey, TValue>`, имплементирайки всички операции, дефинирани в него.
- Интерфейс `IDictionary<TKey, TValue>` е дефиниран в `System.Collections.Generic`, където:
 - **TKey**, дефинира типа на ключа (key)
 - **TValue** типа на стойността (value).
- ```
public interface IDictionary<TKey, TValue>
: ICollection<KeyValuePair<TKey, TValue>>,
IEnumerable<KeyValuePair<TKey, TValue>>,
IEnumerable
```



Този интерфейс е описание на абстрактната структура от данни "речник" и дефинира задължителните операции, които речниците трябва да имплементират, именно:

public void Add (TKey key, TValue value)

- Добавя специфицираните key и value в речника, където: key – ключа на добавения елемент, value – стойността на добавения елемент. При повечето имплементации на класа в .NET, при добавяне на ключ, който вече съществува в речника, се хвърля изключение (ArgumentException).

public TValue this [TKey key] { get; set; }

- Извлича или установява (gets or sets) стойността, асоциирана със специфициран ключ (key).

public bool TryGetValue (TKey key,out TValue value)

- Извлича стойността, асоциирана със специфициран ключ key.



public bool ContainsKey (TKey key)

- Определя дали речника съдържа специфицирания ключ key.

public KeyCollection Keys { get; }

- Извлича колекция съдържаща ключовете на речника.

public ValueCollection Values { get; }

- Извлича колекция съдържаща стойностите на речника.

public bool Remove (TKey key) - премахва стойност със специфициран ключ (key) от речника.

• и др.



- Както вече знаем възможността за обхождането на елементите на един списък с `foreach` се дължи именно на имплементирането на интерфейс `IEnumerable`, от класа на обхождания обект.
- Тъй като `IDictionary<TKey, TValue>` имплементира интерфейса

`IEnumerable<KeyValuePair<TKey, TValue>>`,

това означава, че `foreach` итерира върху списък с обекти от

тип `KeyValuePair<TKey, TValue>`.



## Клас Dictionary<TKey, TValue>

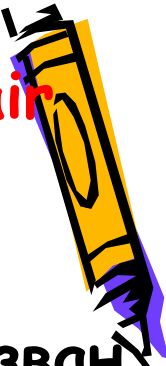
- Клас Dictionary<TKey, TValue> от пространство System.Collections.Generic осигурява съответствието между ключове и стойности. Всяко вкарване на елемент в речника означава вкарване на нова стойност и нейния асоцииран ключ. Всеки ключ в един Dictionary трябва да бъде уникален.
- Извличането на стойност от речника, чрез използване на ключа е много бързо, почти  $O(1)$ , защото class Dictionary е имплементиран като hash таблица.
- Този клас (Dictionary<TKey, TValue>) и класа SortedList<TKey, TValue>, който ще разгледаме по-надолу са алтернатива на класовете, съответно: Hashtable и SortedList от именовано пространство System.Collections.





- Всяка двойка key-value на клас Dictionary<TKey, TValue> може да бъде извлечена като **KeyValuePair** структура. За сравнение, при Hashtable - като **DictionaryEntry** чрез не-генеричния IDictionary интерфейс.
- А както подчертахме, оператор foreach, използван за обхождане на един речник (например, с име myDictionary) от именовано пространство System.Collections.Generic изисква именно това - типа на всеки елемент да е структура KeyValuePair<Tkey, TValue>, където Tkey и TValue са съответно, типа на ключа (например, int) и типа на стойността (например string) на елемента:

```
foreach (KeyValuePair<int, string> kvp in myDictionary)
{
 Console.WriteLine("Key = {0}, Value = {1}",
 kvp.Key, kvp.Value);
}
```



```
using System;
using System.Collections.Generic;
public class Example //Един ПРИМЕР
{ public static void Main()
```

```
{
 //Създава нов речник от string, с ключове - също тип
 string.
 Dictionary<string, string> re4nik = new Dictionary<string,
 string>();
 // Добавяне на елементи. Няма дублиране на ключове.
 re4nik.Add("window", "прозорец");
 re4nik.Add("door", "врата");
 re4nik.Add("gate", "врата");
 re4nik.Add("room", "стая");
 //Метод Add хвърля изключение, ако ключът вече
 съществува.
```



//Метод Add хвърля изключение, ако ключът вече съществува.  
Затова:

```
try
{ re4nik.Add("window", "витрина"); }
catch (ArgumentException)
{
 Console.WriteLine("An element with Key = \"window\" already exists.");
}
```

//Свойство Item е другото име на indexer, така че може да пропуснете неговото име, когато достъпвате елементи.

```
Console.WriteLine("For key = \"room\", value = {0}.",
 re4nik["room"]);
```

//indexer може да се използва за промяна и извличане на стойност, асоциирана с ключ. При промяна:

```
re4nik["room"] = "помещение";
Console.WriteLine("For key = \"room\", value = {0}.",
 re4nik["room"]);
```

//ако един ключ не съществува, то се добавя нова двойка key/value.

```
re4nik["glass-case"] = "витрина";
```

//При извличане, indexer генерира изключение

KeyNotFoundException, ако търсеният ключ не е в Dictionary.



//затова при извличане, чрез indexer :

```
try
{
 Console.WriteLine("For key = \"bla-bla\", value = {0}.",
 re4nik["bla-bla"]);
}
catch (KeyNotFoundException)
{
 Console.WriteLine("Key = \"bla-bla\" is not found.");
}
```

// TryGetValue() - ефективен метод за извличане на стойност от Dictionary.

```
string value = "";
if (re4nik.TryGetValue("bla-bla", out value))
{
 Console.WriteLine("For key = \"bla-bla\", value = {0}.", value);
}
else
{
 Console.WriteLine("Key = \"bla-bla\" is not found.");
}
```



// ContainsKey - може да покаже дали ключът вече съществува.

```
if (!re4nik.ContainsKey("house"))
{
 re4nik.Add("house", "къща");
 Console.WriteLine("Value added for key = \"house\": {0}",
 re4nik["house"]);
}
```

// Когато използвате foreach - елементите са KeyValuePair objects.

```
Console.WriteLine();
foreach (KeyValuePair<string, string> kvp in re4nik)
{
 Console.WriteLine("Key = {0}, Value = {1}",
 kvp.Key, kvp.Value);
}
```

// За да се получат само стойностите, използвайте свойство Values на Dictionary (с име re4nik в примера):

```
Dictionary<string, string> ValueCollection valueColl = re4nik.Values;
```

*/\* Dictionary<TKey, TValue>.ValueCollection съдържа стойностите на Dictionary<TKey, TValue> \*/*

```
Console.WriteLine();
foreach (string s in valueColl)
{
 Console.WriteLine("Value = {0}", s);
}
```



//За извличане само на на ключовете, използвайте свойство Keys.

```
Dictionary<string, string>.KeyCollection keyColl = re4nik.Keys;
/* Dictionary<string, string>.KeyCollection съдържа ключовете на
Dictionary<TKey, TValue> */
```



```
Console.WriteLine();
foreach (string s in keyColl)
{
 Console.WriteLine("Key = {0}", s);
}
```

//използвайте Remove method - за изтриване на двойка key/value.

```
Console.WriteLine("\nRemove(\"glass-case\")");
re4nik.Remove("glass-case");
if (!re4nik.ContainsKey("glass-case"))
{
 Console.WriteLine("Key \"glass-case\" is not found.");
} Console.ReadLine();
}
```



/\* Примерът извежда следното:

An element with Key = "window" already exists.

For key = "room", value = стая.

For key = "room", value = помещение.

Key = "bla-bla" is not found.

Key = "bla-bla" is not found.

Value added for key = "house": къща

Key = window, Value = прозорец

Key = door, Value = врата

Key = gate, Value = врата

Key = room, Value = помещение

Key = glass-case, Value = витрина

Key = house, Value = къща

Value = прозорец

Value = врата

Value = врата

Value = помещение

Value = витрина

Value = къща

Key = window

Key = door

Key = gate

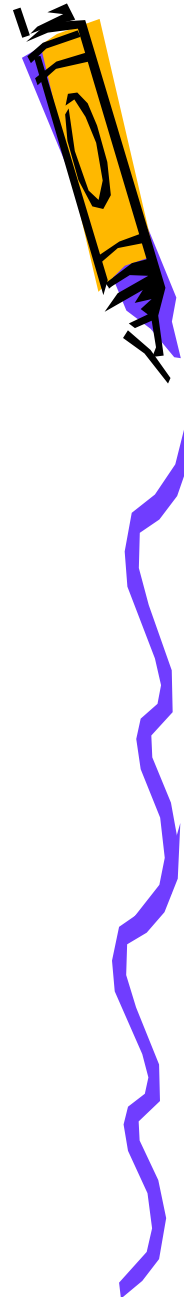
Key = room

Key = glass-case

Key = house

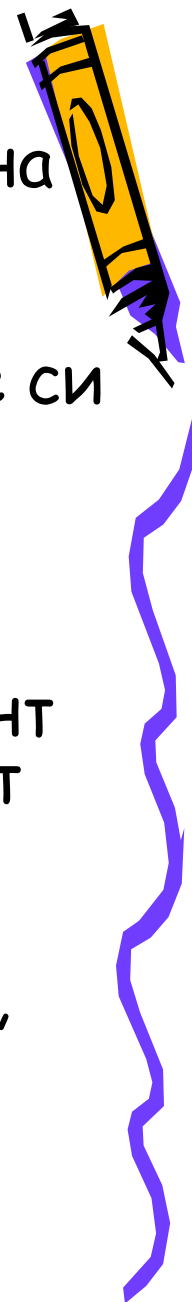
Remove("glass-case")

Key "glass-case" is not found.



# Клас SortedList <TKey, TValue>

- Клас SortedList е генеричен клас, реализация на подредено **двоично дърво за търсене** (binary search tree).
- Двоичните дървета за търсене пазят ключовете си в подреден ред, за да може търсенето да е по-бързо. Сложността на търсенето по ключ в SortedList е  **$O(\log n)$** , където  $n$  е броя на елементите му. Това е много по-добре от линейното време нужно за да се намери елемент по ключ в (неподреден) масив, но е по-бавно от еквивалентната операция върху хеш-таблица.
- По своята сложност на търсене  **$O(\log n)$**  и обектния си модел този клас (SortedList <TKey, TValue>) е подобен на генеричния клас **SortedDictionary<TKey, TValue>**, който ще разгледаме по-надолу.





Това, по което се различават **SortedList** и **SortedDictionary** е използването на паметта, скоростта на вмъкване и отстраняване на елемент.

Предимства на SortedList пред SortedDictionary:

- SortedList използва по-малко памет от SortedDictionary.
- Но, SortedDictionary има по-бързо вмъкване и изтриване на двойка -  $O(\log n)$ , при  $O(n)$  - за SortedList, където  $n$  е броя на елементите в речника.
- Веднъж попълнен, търсенето в SortedList е по-бързо от това в Dictionary.
- Нека да разгледаме следния, обилно снабден с коментари пример, демонстриращ работата с речник SortedList:



```
using System;
using System.Collections.Generic;
public class Example //Един пример
{
 public static void Main()
 {
 //Създава се нов сортиран списък от strings, с ключ -
 //string.
 SortedList<string, string> re4nik =
 new SortedList<string, string>();
 // Добавяне на няколко елемента в списъка. Няма
 // дублирани ключове.
 re4nik.Add("path", "път");
 re4nik.Add("folder", "директория");
 re4nik.Add("directory", "директория");
 re4nik.Add("file", "Файл");
 // Add метод хвърля изключение ArgumentException,
 // ако новият ключ е вече в списъка.
 }
}
```



// Add метод хвърля изключение, ако новият ключ е вече в списъка.

```
try
{ re4nik.Add("path", "икона"); }
catch (ArgumentException)
{
 Console.WriteLine("An element with Key = \"path\" already exists.");
}
```

// Свойство Item (другото име на indexer), може да се използва за извличане на стойност по ключ:

```
Console.WriteLine("For key = \"file\", value = {0}.",
 re4nik["file"]);
```

//...и за смяна на стойността, по задаван ключ (key).

```
re4nik["file"] = "файл";
Console.WriteLine("For key = \"file\", value = {0}.",
 re4nik["file"]);
```

// ако key не съществува, то чрез indexer се добавя двойка (Add key/value)

```
re4nik["icon"] = "икона";
```

// indexer хвърля изключение, ако търсения key не е в списъка.



// indexer хвърля изключение, ако търсения key не е в списъка.

```
try
{ Console.WriteLine("For key = \"tif\", value = {0}.",
 re4nik["tif"]);
}
catch (KeyNotFoundException)
{ Console.WriteLine("Key = \"tif\" is not found.");
}
```

// TryGetValue() - ефективен начин за извличане на стойности.

```
string value = "";
if (re4nik.TryGetValue("tif", out value))
{
 Console.WriteLine("For key = \"tif\", value = {0}.", value);
}
else
{
 Console.WriteLine("Key = \"tif\" is not found.");
}
```

// ContainsKey - тества за наличие на ключ.



// ContainsKey може да се използва преди Add...

```
if (!re4nik.ContainsKey("drive"))
{
 re4nik.Add("drive", "устройство");
 Console.WriteLine("Value added for key = \"drive\": {0}",
 re4nik["drive"]);
}
```

// foreach за обхождане - KeyValuePair обекти.

```
Console.WriteLine();
foreach (KeyValuePair<string, string> kvp in re4nik)
{
 Console.WriteLine("Key = {0}, Value = {1}",
 kvp.Key, kvp.Value);
}
```

// За да извлечен стойностите само - свойство Values.

```
ICollection<string> ilistValues = re4nik.Values;
```

// ICollection<string> - съдържа стойностите на списъка

//Свойство Values - ефективен начин за извличане на стойности  
чрез индекс.

```
Console.WriteLine();
foreach (string s in ilistValues)
{
 Console.WriteLine("Value = {0}", s);
}
```



//Свойство Values - ефективен начин за извличане на стойности чрез индекс.

```
Console.WriteLine("\nIndexed retrieval using the Values " +
 "property: Values[2] = {0}", re4nik.Values[2]);
```

// За да извлечен ключовете само - свойство Keys.

```
ICollection<string> ilistKeys = re4nik.Keys;
```

// ICollection<string> - съдържа ключовете на списъка

```
Console.WriteLine();
```

```
foreach (string s in ilistKeys)
```

```
{ Console.WriteLine("Key = {0}", s); }
```

//Свойство Keys - ефективен начин за извличане на ключове по индекс.

```
Console.WriteLine("\nIndexed retrieval using the Keys " +
 "property: Keys[2] = {0}", re4nik.Keys[2]);
```

// Използване на Remove метод за изтриване на двойка key/value.

```
Console.WriteLine("\nRemove(\"icon\")");
```

```
re4nik.Remove("icon");
```

```
if (!re4nik.ContainsKey("icon"))
```

```
{ Console.WriteLine("Key \"icon\" is not found."); }
```

```
}
```



file:///C:/Documents and Settings/PS/My Documents/Visual Studio 2005/Projec...

An element with Key = "path" already exists.

For key = "file", value = Файл.

For key = "file", value = файл.

Key = "tif" is not found.

Key = "tif" is not found.

Value added for key = "drive": устройство

Key = directory, Value = директория

Key = drive, Value = устройство

Key = file, Value = файл

Key = folder, Value = директория

Key = icon, Value = икона

Key = path, Value = път

Value = директория

Value = устройство

Value = файл

Value = директория

Value = икона

Value = път

Indexed retrieval using the Values property: Values[2] = файл

Key = directory

Key = drive

Key = file

Key = folder

Key = icon

Key = path

Indexed retrieval using the Keys property: Keys[2] = file

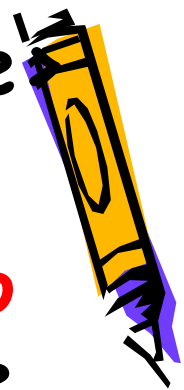
Remove("icon")

Key "icon" is not found.

-

# Клас SortedDictionary <TKey, TValue>

- Класът SortedDictionary <TKey, TValue> (имплементация на структура **подредено двоично балансирано дърво за търсене**) е една речникова колекция - алтернатива на Dictionary<TKey, TValue> (имплементация на структура хеш-таблица), която непрекъснато поддържа наредба на множеството си от ключове.
- Елементите на този речник са винаги сортирани по ключ - това свойство е единственото предимство на този речник пред реализацията на речник с хеш-таблица (клас Dictionary<TKey, TValue>).





- Пазенето на ключовете, нерекъснато сортирани, има своята цена.
- Сложността на търсене по ключ в този речник е по-висока (сложност  $O(\log n)$ , където  $n$  е броя на елементите в речника) от тази в `Dictionary<TKey, TValue>` (сложност  $O(1)$ ).
- По тази причина, ако няма специални изисквания за наредба на ключовете, за предпочитане е да се използва `Dictionary<TKey, TValue>`.



- Както казахме вече, `SortedDictionary<TKey, TValue>` е generic class, който прилича на `SortedList<TKey, TValue>`.
- Предимство на `SortedDictionary` пред `SortedList` е, че `SortedDictionary` има по-бързо вмъкване и изтриване на двойка -  $O(\log n)$ , при  $O(n)$  - за `SortedList`, където  $n$  е броя на елементите в речника.



# Клас List<T>

- Клас List<T> представлява строго типизирана списъчна колекция от обекти (може да бъде видяна като алтернатива на ArrayList от System.Collections), които могат да бъдат достъпвани по индекс.
- Осигурява методи за търсене, сортиране и манипулиране на списъка.



Пример:

```
using System;
```

```
using System.Collections.Generic;
```

```
public class List_Collections
```

```
{ public static void Main()
```

```
{ List<string> sandwich = new List<string>();
```

```
 sandwich.Add("bacon"); //индекс 0
```

```
 sandwich.Add("tomato"); //индекс 1
```

```
 sandwich.Insert(1, "cheese");
```

```
 //вмъкваме на елемент на място с индекс 1
```

```
 foreach (string element in sandwich)
```

```
 {
```

```
 Console.WriteLine(element);
```

```
 }
```

```
 Console.WriteLine(sandwich.Capacity); //4
```

```
 Console.ReadKey(); //bacon cheese tomato
```

```
}
```

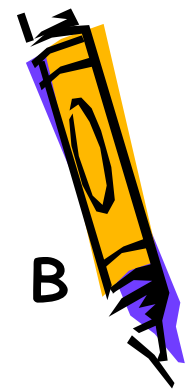
```
}
```



Други силно типизирани колекции в `System.Collections.Generic`, алтернатива на разгледани колекции в предната лекция са `Queue<T>` и `Stack<T>` :

- Клас `Queue<T>` - FIFO типизирана колекция от обекти, аналогична на `Queue` в `System.Collections`.

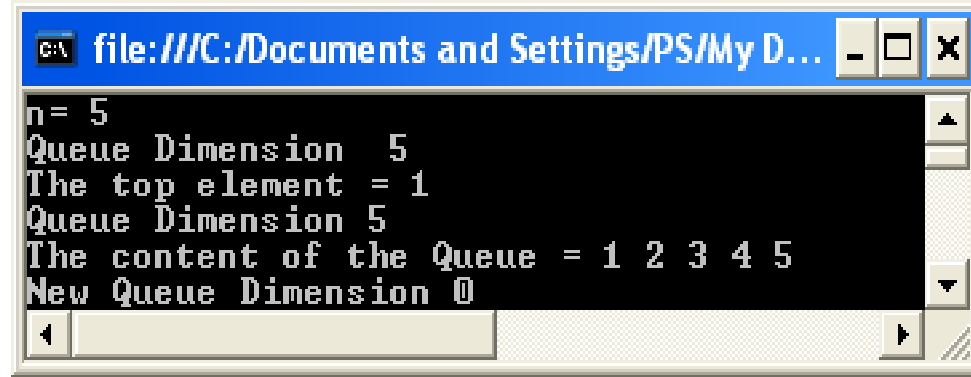
**Пример:** Да запишем в опашка всички числа от 1 до  $n$  (потребителя задава стойност на  $n$ ), след което - да ги извлечем от опашката.



```

using System;
using System.Collections.Generic;
namespace MyProgram
{
 class Program
 {
 public static void Main()
 {
 Console.WriteLine("n= ");
 int n = int.Parse(Console.ReadLine());
 Queue<int>intQ = new Queue<int>();
 for (int i = 1; i <= n; i++)
 intQ.Enqueue(i);
 Console.WriteLine("Queue Dimension " + intQ.Count);
 Console.WriteLine("The top element = " + intQ.Peek());
 Console.WriteLine("Queue Dimension " + intQ.Count);
 Console.WriteLine("The content of the Queue = ");
 while (intQ.Count != 0)
 Console.WriteLine("{0} ", intQ.Dequeue());
 Console.WriteLine("\nNew Queue Dimension " + intQ.Count);
 Console.ReadKey();
 }
 }
}

```



```

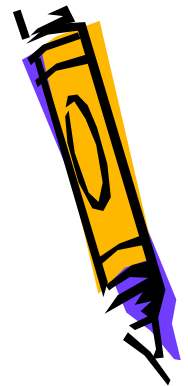
file:///C:/Documents and Settings/PS/My D...
n= 5
Queue Dimension 5
The top element = 1
Queue Dimension 5
The content of the Queue = 1 2 3 4 5
New Queue Dimension 0

```



# Клас `Stack<T>`

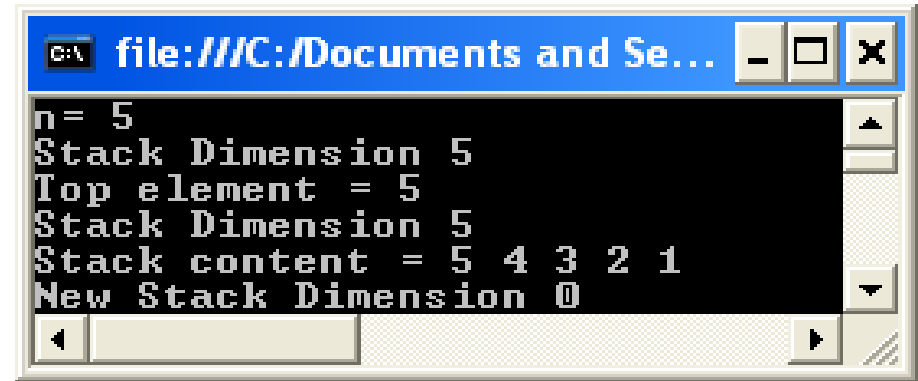
- Клас `Stack<T>` - LIFO типизирана колекция от обекти, аналогична на `Stack` в `System.Collections`.
- **Пример:** Да запишем в стек всички числа от 1 до  $n$  (потребителя задава стойност на  $n$ ), след което - да ги извлечем от стека.



```

using System;
using System.Collections.Generic;
namespace ConsoleApplication
{
 class Program
 {
 public static void Main()
 {
 Console.WriteLine("n= ");
 int n = int.Parse(Console.ReadLine());
 Stack<int> intStack = new Stack<int>();
 for (int i = 1; i <= n; i++)
 intStack.Push(i);
 Console.WriteLine("Stack Dimension " + intStack.Count);
 Console.WriteLine("Top element = " + intStack.Peek());
 Console.WriteLine("Stack Dimension " + intStack.Count);
 Console.Write("Stack content = ");
 while (intStack.Count != 0)
 Console.Write("{0} ", intStack.Pop());
 Console.WriteLine("\nNew Stack Dimension " + intStack.Count);
 Console.ReadKey();
 }
 }
}

```



```

file:///C:/Documents and Se...
n= 5
Stack Dimension 5
Top element = 5
Stack Dimension 5
Stack content = 5 4 3 2 1
New Stack Dimension 0

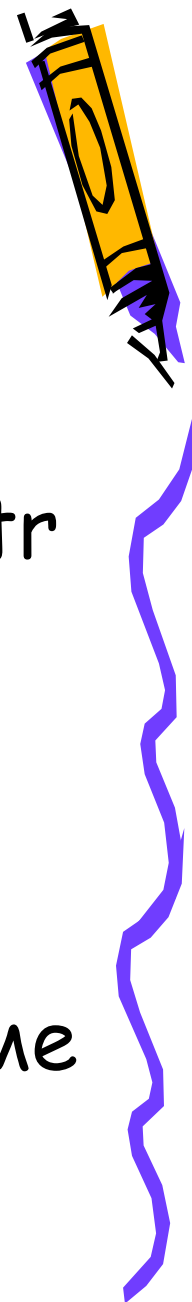
```





# Практически насоки за избор на колекция

- Представената във "Въведение в програмирането със C#" (<http://www.introprogramming.info/intro-csharp-book/read-online/glava19-strukturi-ot-danni-supostavka-i-preporuki/>) сравнителна таблица е подходящ начин за оценка на основните класове, които разгледахме в тази лекция:



| структура                                                                          | добавяне     | търсене      | изтриване    | достъп по индекс |
|------------------------------------------------------------------------------------|--------------|--------------|--------------|------------------|
| масив (T[])                                                                        | $O(N)$       | $O(N)$       | $O(N)$       | $O(1)$           |
| динамичен масив (List<T>)                                                          | $O(1)$       | $O(N)$       | $O(N)$       | $O(1)$           |
| стек (Stack<T>)                                                                    | $O(1)$       | -            | $O(1)$       | -                |
| опашка (Queue<T>)                                                                  | $O(1)$       | -            | $O(1)$       | -                |
| Речник, реализиран с хеш-таблица (Dictionary<K, T>)                                | $O(1)$       | $O(1)$       | $O(1)$       | -                |
| Речник, реализиран с балансирано дърво (SortedDictionary<K, T>, SortedList <K, T>) | $O(\log(N))$ | $O(\log(N))$ | $O(\log(N))$ | -                |

**Въпрос: Трябва ли колекцията, която ни е нужна да обработва фиксиран брой елементи до които ви е необходим достъп по индекс?**

**Ако да – то изберете масив!**

- Масивите трябва да се ползват само когато трябва да обработим фиксиран брой елементи, до които е необходим достъп по индекс.
- Те представляват област от паметта с определен, предварително зададен размер. Добавянето на нов елемент в масив е много бавна операция, защото реално трябва да се задели нов масив с размерност по-голяма с 1 от текущата и да се прехвърлят старите елементи в новия масив.
- Търсенето в масив изисква сравнение на всеки елемент с търсената стойност. В средния случай са необходими  $N/2$  сравнения.
- Изтриването от масив е много бавна операция, защото е свързана със заделяне на масив с размер с 1 по-малък от текущия и преместване на всички елементи без изтрития в новия масив.
- Достъпът по индекс става директно и затова е много бърза операция.



Въпрос: Трябва ли ни колекция, в която лесно да добавяме нови елементи (която да има динамичен брой елементи), до които е необходим достъп по индекс?

Ако да - то  $\text{List}\langle T \rangle$  е правилен избор! Иначе - друга колекция!

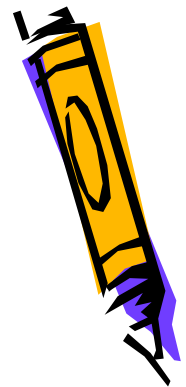
- Търсенето и изтриването на в  $\text{List}\langle T \rangle$  е бавна операци.
- Добавянето на елемент  $\text{List}\langle T \rangle$  е бърза операция, тъй  $\text{List}\langle T \rangle$  вътрешно съхранява елементите си в масив, който има размер по-голям от броя съхранени елементи. При добавяне на елемент обикновено във вътрешния масив има свободно място и затова тази операция отнема константно време. Понякога масивът се препълва и се налага да се разшири. Това отнема линейно време, но се случва много рядко. В крайна сметка при голям брой добавяния усреднената сложност на добавянето на елемент към  $\text{List}\langle T \rangle$  е константна -  $O(1)$ . Тази усреднена сложност се нарича амортизирана сложност. Амортизирана линейна сложност означава, че ако добавим последователно 10 000 елемента, ще извършим сумарно брой стъпки от порядъка на 10 000 и болшинството от тях ще се изпълнят за константно време, а останалите (една много малка част) ще се изпълнят за линейно време.



Въпрос: Трябва ли ни колекция, която реализира поведение "последен влязъл, пръв излязъл" (LIFO).

Ако да - то  $\text{Stack}\langle T \rangle$  е правилен избор!  
Иначе - друга колекция!

- Стекът е структура от данни, в която са дефинирани 3 операции, които се изпълняват бързо, с константна сложност.
- добавяне на елемент на върха на стека,
- изтриване на елемент от върха на стека
- и извличане на елемент от върха на стека без премахването му.
- Операциите търсене и достъп по индекс не се поддържат.



Въпрос: Трябва ли ни колекция, която реализира поведение "пръв влязъл, пръв излязъл" (FIFO). Ако да – то Queue<T> е правилен избор! Иначе – друга колекция!

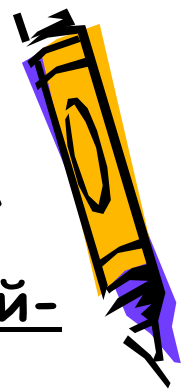
- Опашката по естествен начин моделира списък от чакащи хора, задачи или други обекти, които трябва да бъдат обработени последователно, в реда на постъпването им.
- Опашката е структура от данни, в която са дефинирани две операции, които се изпълняват бързо, с константна сложност:
- добавяне на елемент и
- извличане на елемента, който е наред.
- Операциите търсене и достъп по индекс не се поддържат.



Въпрос: Трябва ли ни колекция от двойки ключ-стойност (речникова колекция), в която искате бързо да добавяте елементи и възможно най-бързо да търсите по ключ.

Ако да - то Dictionary<K,T> е правилен избор! Иначе - друга колекция!

- Счита се, че хеш-таблицата (класа Dictionary<K,T>) е най-бързата структура от данни, която осигурява добавяне, търсене и изтриването на елементи по ключ - със константна сложност в средния случай. Препоръчва се да се използва винаги, когато ни трябва бързо търсене по ключ. Например, ако трябва да преброим колко пъти се среща в текстов файл всяка дума измежду дадено множество думи, можем да ползваме Dictionary<string, int> като ползваме за ключ търсените думи, а за стойност - колко пъти се срещат във файла.
- Операцията достъп по индекс не е достъпна, защото елементите в хеш-таблицата се нареждат по почти случаен начин и редът им на постъпване не се запазва.
- Когато се налага в един ключ да съхраняваме няколко стойности, можем да ползваме List<T> като стойност за този ключ и в него да натрупваме поредица от елементи. Например ако ни трябва хеш-таблица Dictionary<int, string>, в която да натрупваме двойки {цяло число, символен низ} с повторения, можем да ползваме Dictionary<int, List<string>>.



Въпрос: Трябва ли ни колекция от двойки ключ-стойност (речникова колекция), в която искате бързо да добавяте елементи и да търсите по ключ, и имате нужда да **извличане на елементите, сортирани в нарастващ ред на ключа?**

Ако да - то `SortedDictionary<K,T>` е правилен избор!  
Иначе - друга колекция!

- Реализацията на структурата от данни "речник" чрез подредено двоично дърво за претърсване (binary search tree) (класът `SortedDictionary<K,T>`) позволява съхранение на двойки ключ-стойност, при което те са винаги подредени (сортирани) по големина на ключа (в нарастващ ред). В някои задачи, които можем да решим също успешно и с `Dictionary<K,T>`, последното е предимство.
- Структурата осигурява логаритмична сложност на изпълнение -  $O(\log(N))$  на основните операции: добавяне на елемент, търсене по ключ и изтриване на елемент). Това означава 10 стъпки при 1000 елемента и 20 стъпки при 1 000 000 елемента.





- За разлика от хеш-таблиците, където при лоша хеш-функция може да се достигне до линейна сложност на търсенето и добавянето, при структурата `SortedDictionary<K,T>` броят стъпки за изпълнение на основните операции в средния и в най-лошия случай е един и същ -  $\log_2(N)$ . При балансираните дървета няма хеширане, няма колизии и няма риск от използване на лоша хеш-функция.
- Отново, както при хеш-таблиците, един ключ може да се среща в структурата най-много веднъж. Ако искаме да поставяме няколко стойности под един и същ ключ, трябва да ползваме за стойност на елементите някакъв списък, например `List<T>`.

