# Упражнения: Други структури от данни

## Прочетете повече за хеш таблиците

Преди да започнете се запознайте с концепцията за хеш таблица: [https://bg.wikipedia.org/wiki/Хеш-таблица](https://bg.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%82%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0). Забележете, че съществуват много стратегии за управление на колизиите като свързване на елементи или отворено адресиране. Тук ще използваме една от най-простите стратегии - свързване на елементите в колизия чрез свързани списъци.

Типичните операции в хеш таблица са:

* Добавяне на елемент
* Премахване на елемент
* Проверка дали даден ключ е в таблицата
* Извличане на елемент
* Промяна на елемент
* Обхождане на всички елементи
* Обхождане на всички ключове
* Обхождане на всички стойности
* Извличане на броя на елементите

## Дефиниране на класове за хеш таблица

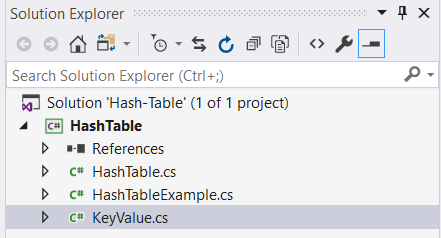
Създайте класове, реализиращи хеш таблица със следната функционалност:

* *Добавяне на елемент*
* *Премахване на елемент*
* *Проверка дали даден ключ е в таблицата*
* *Извличане на елемент*
* *Промяна на елемент*

### Реализация

*Създайте нов проект (конзолно приложение) във Visual Studio и добавете следните класове:*

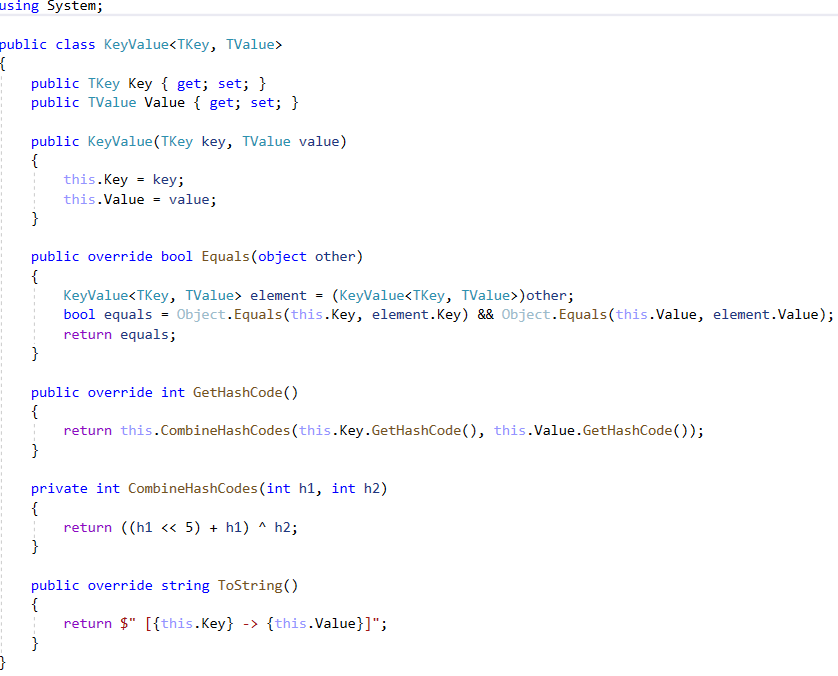
* *HashTable<TKey, TValue>*
* *KeyValue<TKey, TValue>*

**

*Класът KeyValue<TKey, TValue> ще съдържа елемент от хеш таблицата, като ключа на елемента ще бъде от тип TKey, а стойността от тип TValue.*

*Два елемента от тип KeyValue<TKey, TValue> ще бъдат считани за еднакви ако ключовете и стойностите им са еднакви.*

*Хешът на обект от тип KeyValue<TKey, TValue> ще бъде получаван от комбинацията между хешовете на ключа и на стойността.*

**

*Класът HashTable<TKey, TValue> ще съдържа масив от свързани списъци. Този масив ще пази елементите на хеш таблицата. При липса на колизии запълнените клетки на този масив ще съдържат свързан списък с точно един елемент. При идентифициране на колизия съответния свързан списък ще съдържа всички елементи, които са в колизия.*

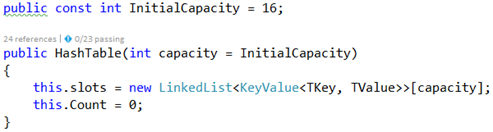
**

## Реализация на конструктора на хеш таблицата

Реализирайте конструктора на хеш таблицата. Неговото предназначение е да задели памет за клетките, които ще съдържат елементите на хеш таблицата. Ще ни бъдат нужни два конструктора:

* Конструктор без параметри
* Конструктор, приемащ 1 целочислен параметър - капацитета на хеш таблицата

### Реализация



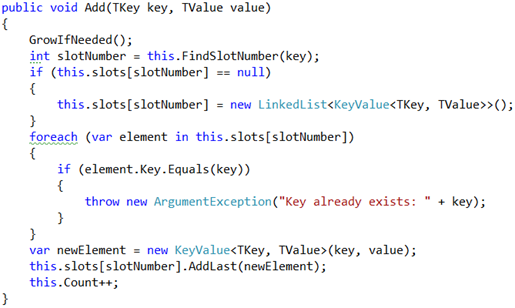
Тук константата **InitialCapacity** дефинира началния размер на хеш таблицата - 16 елемента. За съхранение на елементите се използва масив от свързани списъци, за да може да бъде приложена стратегията за разрешаване на конфликти - свързани елементи.

## Реализация на добавяне на елемент

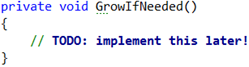
Реализирайте метода **Add(key, value)**, който добавя елемент в хеш таблицата. Този мето трябва да вземе предвид следните ситуации:

* Добавяне на елемент, който няма колизия със вече съществуващ
* Установяване на колизия при добавяне на елемент
* Установяване на дублиращ се ключ
* Увеличаване на размера на хеш таблицата при нужда (удвояване на размера й, когато сме близо до запълване)

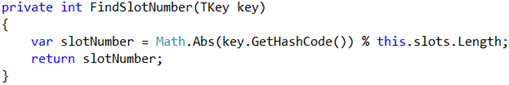
### Реализация



Стартираме с проверка дали хеш таблицата е пълна. При идентифицирано запълване на хеш таблицата трябва да удвоим размера й. Това се прави от метода **GrowIfNeeded()**. Нека разгледаме поведението на този метод по-късно и за момента го оставим празен:



Следващата стъпка е да намерим клетката в хеш таблицата, която ще държи нашия елемент. Индекса на клетката се изчислява от хеша на ключа. Обикновено за тази цел се използва **GetHashCode()**, наследен от класа **System.Object**, който предоставя изчисление на хеш за вградени и дефинирани от потребител типове в работната рамка .НЕТ. Този метод връща 32 битово число. Тъй като числото ще бъде използвано за индекс в масив, то трябва да е в интервала [0, размера на масива - 1]. Затова делим по модул абсолютната стойност на хеша на размера на таблицата. По този начин винаги получаваме индекс в рамките на размера на хеш таблицата.



След като вече имаме номера на клетката там се съдържа или инстанция на свързан списък или **null**. И в двата случая в тази клетка трябва да имаме свързан списък, съдържащ всички елементи на хеш таблицата с хеш равен на този на елемента, който добавяме.

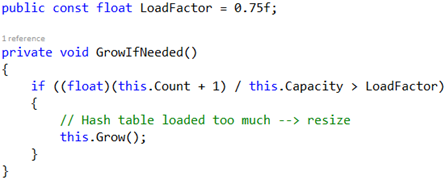
Проверяваме свързания списък за елемент с ключ равен на ключа на елемента, който добавяме. Ако намерим такъв - операцията спира и излизаме от метода с изключение. Ако ключа не се дублира, елемента се добавя към свързания списък и увеличаваме броя на елементите в хеш таблицата с 1.

## Реализация на методите GrowIfNeeded() и Grow()

Предназначението на метода **GrowIfNeeded()** е да разпознава дали прагът на запълване на хеш таблицата е достигнат и в този случай да удвои размера й.

### Реализация

В нашия случай капацитета на хеш таблицата ни ще бъде удвоен ако тя е запълнена на 75% или повече и ние се опитаме да добавим нов елемент. В този случай ще бъде извикан методът **Grow()**, който извършва оразмеряването на таблицата.



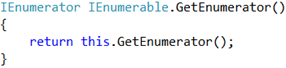
Реализирайте интерфейса **IEnumerable<T>**, така че да е възможно итерирането през елементите.

## Реализация на обхождане на всички елементи

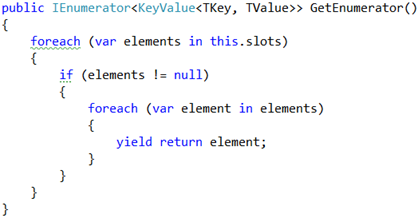
Имплементирайте интерфейса **IEnumerable<T>**, така че да е възможно итерирането през елементите на хеш таблицата посредством конструкцията **foreach**.

### Реализация

За да можем да използваме конструкцията foreach с потребителски дефинирани колекции в C#, те трябва да имплементират интерфейса **IEnumerable<T>**. Тъй като хеш таблицата съдържа елементи от тип **KeyValue<TKey, TValue>**, трябва да се имплементира интерфейса **IEnumerable<KeyValue<TKey, TValue>**. Този интерфейс задължава реализацията на следните два метода:



и



Първия метод извиква втория, а втория върши реалната работа по предоставяне на елементите. В него се обхождат всички елементи в хеш таблицата и за всеки от свързаните списъци, съхранени там, се обхождат всички елементи. Този метод използва конструкцията **yield return** (генерираща функция), която връща елементите “при поискване”. За да научите повече за генериращите функции прочетете <https://en.wikipedia.org/wiki/Generator_(computer_programming)>.

## Реализация на метода Find(key)

Реализирайте метод, който търси елемент в хеш таблицата по ключ и връща като резултат стойността му, а в случай, че ключа не е намерен - връща **null**. Алгоритъмът за търсене на елемент в хеш таблицата трябва да бъде с константна сложност.

### Подсказка

Вариант на реализацията с линейна сложност би била обхождане на всички елементи в свързаните списъци от хеш таблицата.

За да може да сведем сложността до константна трябва да намерим позицията на свързания списък, отговарящ за този ключ, да обходим елементите му и да намерим този с ключ равен на ключа, с който търсим.

## Реализация на методите Get(key), TryGetValue(key, Out Value) и ContainsKey(key)

Реализирайте следните методи, като алгоритмите за търсене на елемент в хеш таблицата трябва да бъдат с константна сложност:

* **Get(key)** - намира елемент по ключ и връща като резултат стойността му. Ако ключа не бъде намерен, метода хвърля изключение.
* **TryGetValue(key, out value)** - намира елемент по ключ:
  + ако елементът е намерен метода връща като резултат **True**, а стойността на намерения елемент се записва в изходния параметър **value**
  + ако елементът не бъде намерен, метода връща **False**
* **Contains(key)** - търси конкретен ключ в хеш таблицата и връща булев резултат - ако ключа е намерен метода връща **True** иначе **False**.

### Подсказка

Виж подсказката на Зад. 7.

## Реализация на метода AddOrReplace(key, Value)

Реализирайте метод, който търси елемент в хеш таблицата по ключ и:

* ако ключа съществува в таблицата подменя стойността му със стойността на параметъра **value**
* ако ключа не съществува в таблицата - създава нов елемент и го добавя

### Подсказка

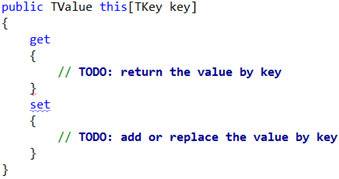
Решението е комбинация на Зад. 7 и Зад. 4.

## Реализация на Indexer This[key]

Реализирайте Indexer в класа HashTable, като **get** секцията му търси елемент в хеш таблицата по зададен ключ, а **set** секцията му подменя стойността на елемент, съхранен на даден ключ. Ако ключа не бъде намерен в таблицата се хвърля изключение.

### Подсказка

Синтаксиса на Indexer е:



Решението е комбинация на Зад. 7 и Зад. 9.

## Реализация на Remove(key)

Реализирайте метод за премахване на елемент от хеш таблицата, който приема като параметър ключа на елемента, който трябва да бъде премахнат и връща boolean стойност - **True** ако елемента е премахнат успешно и **False** ако не е намерен.

### Подсказка

За да се премахне елемент по ключ, трябва да бъде намерен свързания списък, в който се намира елемента, след което да се премахне елемента от свързания списък.

## Реализация на Clear()

Реализирайте метод, който премахва всички елементи на хеш таблицата.

### Подсказка

Можете да заделите нова памет за хеш таблицата и да върнете брояча на елементите на нула по същия начин, по който това е направено в конструктора.

## Реализация на свойствата за достъп до всички ключове и всички стойности – Keys и Values

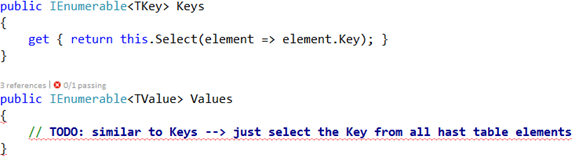
Реализирайте свойства **Keys** и **Values** в класа HashTable. Двете свойства трябва да имат единствено **get** секции, в които да се връща колекция от съответно всички ключове или всички стойности на елементите в хеш таблицата.

### Подсказка

Има различни начини да построите колекцията от ключове и колекцията от стойности.

Един начин да върнете всички ключове е да направите колекция **List<TKey>** и в нея последователно да добавяте ключовете на елементите от хеш таблицата докато ги обхождате. Аналогично за стойностите.

Друг начин е да използвате Linq extension методи по следния начин:



## Преброяване на символи

Напишете програма, която прочита текст от клавиатурата и преброява употребата на всеки символ в него. Уникалните символи се отпечатват в азбучен ред, а срещу всеки от тях се отпечатва цяло число - колко пъти той се среща в текста.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input** | **Output** |  | **Input** | **Output** |
| Coding rocks | : 1 time/s  C: 1 time/s  c: 1 time/s  d: 1 time/s  g: 1 time/s  i: 1 time/s  k: 1 time/s  n: 1 time/s  o: 2 time/s  r: 1 time/s  s: 1 time/s |  | Did you know Math.Round rounds to the nearest even integer? | : 9 time/s  .: 1 time/s  ?: 1 time/s  D: 1 time/s  M: 1 time/s  R: 1 time/s  a: 2 time/s  d: 3 time/s  e: 7 time/s  g: 1 time/s  h: 2 time/s  i: 2 time/s  k: 1 time/s  n: 6 time/s  o: 5 time/s  r: 3 time/s  s: 2 time/s  t: 5 time/s  u: 3 time/s  v: 1 time/s  w: 1 time/s  y: 1 time/s |

## Телефонен указател

Напишете програма, която прочита от клавиатурата имена и телефони на контакти от телефонен указател.

Формата на името и телефона са както следва: **{име}-{телефон}**

Попълването на телефонния указател приключва при въвеждане на командата **search**

След въвеждане на командата **search**, програмата ви трябва да може да търси контакт по име. При въвеждане на име, програмата извежда на екрана името на контакта и телефонния му номер, а ако не намери контакта - връща **“Contact {name} does not exist!”**.

Програмата приключва изпълнение при въвеждане на команда **end**.

|  |  |
| --- | --- |
| **Input** | **Output** |
| Peter-0888080808  **search**  Mariika  Peter  end | Contact Mariika does not exist.  Peter -> 0888080808 |
| Peter-+359888001122  Stamat(Gosho)-666  Gero-5559393  Simo-02/987665544  **search**  Simo  simo  Stamat  Stamat(Gosho)  end | Simo -> 02/987665544  Contact simo does not exist.  Contact Stamat does not exist.  Stamat(Gosho) -> 666 |

## Двоично дърво за търсене

*Реализирайте двоично дърво за търсене със следната функционалност:*

* *Добавяне на елемент*
* *Търсене на елемент*
* *Премахване на елемент*
* *Проверка дали даден елемент съществува в дървото*

*Създайте програма, която прочита от конзолата цяло число N, след което N на брой елементи и ги добавя в двоично дърво за търсене.*

*Изтрийте от дървото следните елементи:*

* *Най-големия*
* *Най-малкия*
* *най-близкия (със закръгляне нагоре) до средното аритметично на всички елементи.*

*Отпечатайте дървото на конзолата - всеки елемент на нов ред с отместване по 2 интервала повече за всяко по-дълбоко ниво.*