# Chapter 19. Data Structures and Algorithm Complexity

## In This Chapter

In this chapter we will **compare the data structures** we have learned so far by the **performance** (execution speed) of the basic operations (addition, search, deletion, etc.). We will give specific tips in what situations **what data structures to use**. We will explain how to choose between data structures like hash-tables, arrays, dynamic arrays and sets implemented by hash-tables or balanced trees. Almost all of these structures are implemented as part of NET Framework, so to be able to **write efficient and reliable code** we have to learn to apply the most appropriate structures for each situation.

## Why Are Data Structures So Important?

You may wonder why we pay so **much attention to data structures** and why we review them in such a great details. The reason is we aim to make out of you **thinking** software engineers. Without knowing the basic data structures and computer algorithms in programming well, you cannot be good developers and risk to stay an amateur. Whoever knows **data structures and algorithms** well and starts thinking about their correct use has big chance to become a professional – one that analyzes the problems in depth and proposes efficient solutions.

There are hundreds of books written on this subject. In the four volumes, named **"The Art of Computer Programming", Donald Knuth** explains data structures and algorithms in more than 2500 pages. Another author, **Niklaus Wirth**, has named his book after the answer to the question "why are data structures so important", which is "**Algorithms + Data Structures = Programs**". The main theme of the book is again the fundamental algorithms and data structures in programming.

**Data structures and algorithms are the fundamentals of programming. In order to become a good developer it is essential to master the basic data structures and algorithms and learn to apply them in the right way.**

To a large degree our book is focused on learning data structures and algorithms along with the programming concepts, language syntax and problem solving. We also try to illustrate them in the context of modern software engineering with C# and .NET Framework.

# Algorithm Complexity

We cannot talk about **efficiency of algorithms and data structures** without explaining the term "algorithm complexity", which we have already mentioned several times in one form or another. We will avoid the mathematical definitions and we are going to give a simple explanation of what the term means.

**Algorithm complexity** is a **measure** which evaluates the order of the **count of operations**, performed by a given or algorithm as a function of the size of the input data. To put this simpler, complexity is a rough **approximation of the number of steps** necessary to execute an algorithm. When we evaluate complexity we speak of order of operation count, not of their exact count. For example if we have an order of $N^2$ operations to process N elements, then $N^2/2$ and $3*N^2$ are of one and the same quadratic order.

Algorithm complexity is commonly represented with the **O(f) notation**, also known as **asymptotic notation** or "**Big O notation**", where **f** is the function of the size of the input data. The asymptotic computational complexity **O(f)** measures the order of the consumed resources (CPU time, memory, etc.) by certain algorithm expressed as function of the input data size.

Complexity can be **constant**, **logarithmic**, **linear**, **n*log(n)**, **quadratic**, **cubic**, **exponential**, etc. This is respectively the order of constant, logarithmic, linear and so on, number of steps, are executed to solve a given problem. For simplicity, sometime instead of "**algorithms complexity**" or just "**complexity**" we use the term "**running time**".

**Algorithm complexity is a rough approximation of the number of steps, which will be executed depending on the size of the input data. Complexity gives the order of steps count, not their exact count.**

## Typical Algorithm Complexities

This table will explain what every type of complexity (running time) means:

| Complexity | Running Time | Description |
|---|---|---|
| constant | O(1) | It takes a **constant number of steps** for performing a given operation (for example 1, 5, 10 or other number) and this count does not depend on the size of the input data. |
| logarithmic | O(log(N)) | It takes the order of **log(N) steps**, where the base of the logarithm is most often 2, for performing a given operation on N elements. For example, if N = 1,000,000, an algorithm with a complexity O(log(N)) |

| | | |
|---|---|---|
| | | would do about 20 steps (with a constant precision). Since the base of the logarithm is not of a vital importance for the order of the operation count, it is usually omitted. |
| linear | O(N) | It takes nearly the **same amount of steps as the number of elements** for performing an operation on N elements. For example, if we have 1,000 elements, it takes about 1,000 steps. Linear complexity means that the number of elements and the number of steps are linearly dependent, for example the number of steps for N elements can be N/2 or 3*N. |
| | O(n*log(n)) | It takes **N*log(N) steps** for performing a given operation on N elements. For example, if you have 1,000 elements, it will take about 10,000 steps. |
| quadratic | $O(n^2)$ | It takes the order of **$N^2$ number** of steps, where the N is the size of the input data, for performing a given operation. For example if N = 100, it takes about 10,000 steps. Actually we have a quadratic complexity when the number of steps is in quadratic relation with the size of the input data. For example for N elements the steps can be of the order of $3*N^2/2$. |
| cubic | $O(n^3)$ | It takes the order of **$N^3$ steps**, where N is the size of the input data, for performing an operation on N elements. For example, if we have 100 elements, it takes about 1,000,000 steps. |
| exponential | $O(2^n)$, O(N!), $O(n^k)$, … | It takes a number of steps, which is with an **exponential** dependability with the size of the input data, to perform an operation on N elements. For example, if N = 10, the exponential function $2^N$ has a value of 1024, if N = 20, it has a value of 1 048 576, and if N = 100, it has a value of a number with about 30 digits. The exponential function N! grows even faster: for N = 5 it has a value of 120, for N = 10 it has a value of 3,628,800 and for N = 20 – 2,432,90,008,176,640,000. |

When evaluating complexity, **constants are not taken into account**, because they do not significantly affect the count of operations. Therefore an algorithm which does N steps and algorithms which do N/2 or 3*N respectively are considered linear and approximately equally efficient, because they perform a number of operations which is of the same order.

## Complexity and Execution Time

The **execution speed** of a program depends on the complexity of the algorithm, which is executed. If this complexity is low, the program will execute fast even for a big number of elements. If the complexity is high, the program will execute slowly or will not even work (it will hang) for a big number of elements.

If we take an average computer from 2008, we can assume that it can perform about **50,000,000 elementary operations per second**. This number is a rough approximation, of course. The different processors work with a different speed and the different elementary operations are performed with a different speed, and also the computer technology constantly evolves. Still, if we accept we use an average home computer from 2008, we can make the following conclusions about the **speed of execution** of a given program depending on the algorithm complexity and size of the input data.

| Algorithm | 10 | 20 | 50 | 100 | 1,000 | 10,000 | 100,000 |
|---|---|---|---|---|---|---|---|
| O(1) | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. |
| O(log(n)) | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. |
| O(n) | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. |
| O(n*log(n)) | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. |
| $O(n^2)$ | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | 2 sec. | 3-4 min. |
| $O(n^3)$ | < 1 sec. | < 1 sec. | < 1 sec. | < 1 sec. | 20 sec. | 5.55 hours | 231.5 days |
| $O(2^n)$ | < 1 sec. | < 1 sec. | 260 days | hangs | hangs | hangs | hangs |
| O(n!) | < 1 sec. | hangs | hangs | hangs | hangs | hangs | hangs |
| $O(n^n)$ | 3-4 min. | hangs | hangs | hangs | hangs | hangs | hangs |

We can draw many **conclusions** from the above table:

- Algorithms with a **constant**, **logarithmic** or **linear complexity** are so **fast** that we cannot feel any delay, even with a relatively big size of the input data.
- Complexity **O(n*log(n))** is **similar to the linear** and works nearly as fast as linear, so it will be very difficult to feel any delay.
- **Quadratic** algorithms work very well up to several thousand elements.
- **Cubic** algorithms work well if the elements are not more than 1,000.
- Generally these so called **polynomial algorithms** (any, which are not exponential) are considered to be fast and working well for thousands of elements.
- Generally the **exponential algorithms do not work well** and we should avoid them (when possible). If we have an exponential solution to a task, maybe we actually do not have a solution, because it will work only if the number of the elements is below 10-20. Modern cryptography is based exactly on this – there are not any fast (non-exponential) algorithms for finding the secret keys used for data encryption.

**If you solve a given problem with an exponential complexity this means that you have solved it for a small amount of input data and generally your solution does not work.**

The data in the table is just for orientation, of course. Sometimes a **linear algorithm could work slower than a quadratic** one or a cubic algorithm could work faster than O(n*log(n)). The reasons for this could be many:

- It is possible the **constants** in an algorithm with a low complexity to be big and this could eventually make the algorithm slow. For example, if we have an algorithm, which makes **50*n steps** and another one, which makes **1/100*n*n** steps, for elements up to 5000 the quadratic algorithm will be faster than the linear.
- Since the complexity evaluation is made in the **worst case scenario**, it is possible a quadratic algorithm to work better than O(n*log(n)) in 99% of the cases. We can give an example with the algorithm **`QuickSort`** (the standard sorting algorithm in .NET Framework), which in the **average case** works a bit better than **`MergeSort`**, but in the worst case **`QuickSort`** can make the order of $\mathbf{n^2}$ **steps**, while **`MergeSort`** does always **O(n*log(n)) steps**.
- It is possible an algorithm, which is evaluated to execute with a linear complexity, to not work so fast, because of an **inaccurate complexity evaluation**. For example if we search for a given word in an array of words, the complexity is **linear**, but at every step **string comparison** is performed, which is not an elementary operation and can take much more time than performing simple elementary operation (for example comparison of two integers).

## Complexity by Several Variables

Complexity can depend on several input variables at once. For example, if we look for an element in a rectangular **matrix with sizes M and N**, the searching speed depends on M and N. Since in the worst case we have to traverse the entire matrix, we will do M*N number of steps at most. Therefore the complexity is **O(M*N)**.

## Best, Worst and Average Case

Complexity of algorithms is usually evaluated in the **worst case** (most unfavorable scenario). This means in the average case they can work faster, but in the worst case they work with the evaluated complexity and not slower.

Let's take an example: **searching in array**. To find the searched key in the **worst case**, we have to check all the elements in the array. In the **best case** we will have luck and we will find the element at first position. In the average case we can expect to check half the elements in the array until we find the one we are looking for. Hence in the **worst case** the complexity is O(N) – **linear**. In the average case the complexity is O(N/2) = O(N) – linear, because when evaluating complexity one does not take into account the constants. In the best case we have a constant complexity O(1), because we make only one step and directly find the element.

## Roughly Estimated Complexity

Sometimes it is **hard to evaluate the exact complexity** of a given algorithm, because it performs operations and it is not known exactly how much time they will take and how many operations will be done internally. Let's take the example of **searching a given word in an array of strings** (texts). The task is easy: we have to traverse the array and search in every text with **`Substring()`** or with a regular expression for the given word. We can ask ourselves the question: if we had 10,000 texts, would this work fast? What if the texts were 100,000? If we carefully think about it, we will implement that in order to evaluate adequately, we have to **know how big are the texts**, because there is a difference between searching in people's names (which are up to 50-100 characters) and searching in scientific articles (which are roughly composed by 20,000 – 30,000 characters). However, we can evaluate the complexity using the length of the texts, through which we are searching: it is at least **O(L)**, where L is the sum of the lengths of all texts. This is a pretty rough evaluation, but it is much more accurate than complexity O(N), where N is the number of the texts, right? We should think whether we take into account all situations, which could occur. Does it matter **how long the searched word is**? Probably searching of long words is slower than searching of short words. In fact things are slightly different. If we search for "**aaaaaaa**" in the text "**aaaaaabaaaaacaaaaaabaaaaacaaaaab**", this will be slower than if we search for "xxx" in the same text, because in the first case we will get more sequential matches than in the second case. Therefore,

in some special situations, searching seriously depends on the length of the word we search and the complexity O(L) could be underestimated.

## Complexity by Memory

Besides the number of steps using a function of the input data, one can **measure other resources**, which an algorithm uses, for example **memory**, count of disk operations, etc. For some algorithms the execution speed is not as important as the **memory they use**. For example if a given algorithm is linear but it uses RAM in the order of $N^2$, it will be probably shortage of memory if N = 100,000 (then it will need memory in order of 9 GB RAM), despite the fact that it should work very fast.

## Estimating Complexity – Examples

We are going to give **several examples**, which show how you can estimate the complexity of your algorithms, and decide whether the code written by you will work fast:

If we have a single loop from 1 to N, its complexity is **linear** – **O(N)**:

```
int FindMaxElement(int[] array)
{
   int max = int.MinValue;
   for (int i = 1; i < array.Length; i++)
   {
      if (array[i] > max)
      {
         max = array[i];
      }
   }
   return max;
}
```

This code will work well even if the number of elements is huge.

If we have **two of nested loops from 1 to N**, their complexity is **quadratic – O($N^2$)**. Example:

```
int FindInversions(int[] array)
{
   int inversions = 0;
   for (int i = 0; i < array.Length - 1; i++)
   {
      for (int j = i + 1; j < array.Length; j++)
      {
         if (array[i] > array[j])
         {
```

```
            inversions++;
          }
        }
      }
      return inversions;
}
```

If the elements are no more than several thousand or tens of thousands, this code will work well.

If we have **tree nested loops from 1 to N**, their complexity is **cubic – O(N$^3$)**. Example:

```
long Sum3(int n)
{
  long sum = 0;
  for (int a = 1; a < n; a++)
  {
    for (int b = 1; b < n; b++)
    {
      for (int c = 1; c < n; c++)
      {
        sum += a * b * c;
      }
    }
  }
  return sum;
}
```

This code will work well if the number of elements is below 1,000.

If we have **two nested loops from 1 to N and from 1 to M** respectively, their complexity will be **quadratic – O(N*M)**. Example:

```
long SumMN(int n, int m)
{
  long sum = 0;
  for (int x = 1; x <= n; x++)
  {
    for (int y = 1; y <= m; y++)
    {
      sum += x * y;
    }
  }
  return sum;
}
```

The speed of this code **depends on two variables**. The code will work well if M, N < 10,000 or if at least the one variable has a value small enough.

We should pay attention to the fact that **not always** tree nested loops mean cubic complexity. Here is an example in which the complexity is **O(N*M)**:

```
long SumMN(int n, int m)
{
  long sum = 0;
  for (int x = 1; x <= n; x++)
  {
    for (int y = 1; y <= m; y++)
    {
      if (x == y)
      {
        for (int i = 1; i <= n; i++)
        {
          sum += i * x * y;
        }
      }
    }
  }
  return sum;
}
```

In this example the most inner loop executes exactly min(M, N) times and does not significantly affect the algorithm speed. The outer code executes approximately N*M + min(M,N)*N steps, i.e. its complexity is **quadratic**.

When using a **recursion**, the **complexity is more difficult to be estimated**. Here is an example:

```
long Factorial(int n)
{
  if (n == 0)
  {
    return 1;
  }
  else
  {
    return n * Factorial(n - 1);
  }
}
```

In this example the complexity is obviously linear – O(N), because the function **`Factorial()`** executes exactly once for each of the numbers 1 … n.

Here is a recursive function for which it is very **difficult to estimate** the complexity:

```
long Fibonacci(int n)
{
   if (n == 0)
   {
      return 1;
   }
   else if (n == 1)
   {
      return 1;
   }
   else
   {
      return Fibonacci(n - 1) + Fibonacci(n - 2);
   }
}
```

If we write down what really happens when the upper code executes, we will see that the function calls itself as many times as the Fibonacci's n+1 number is. We can roughly evaluate the complexity by another way too: since on every step of the function execution 2 recursive calls are done in average, the count of the recursive calls must be **in order of $2^n$**, i.e. we have an **exponential complexity**. This automatically means that for values greater than 20-30 the function will "hang". You may check this yourself.

The same function for calculating the **$n^{th}$ number of Fibonacci** can be written with a **linear complexity** in the following way:

```
long Fibonacci(int n)
{
   long fn = 1;
   long fn1 = 1;
   long fn2 = 1;
   for (int i = 2; i < n; i++)
   {
      fn = fn1 + fn2;
      fn2 = fn1;
      fn1 = fn;
   }
   return fn;
}
```

You see that the complexity estimation helps us to **predict whether a given code will work slowly** before we have run it and it implies we should look for a more efficient solution.

# Comparison between Basic Data Structures

After you have been introduced to the term algorithm complexity, we are now ready to make a **comparison between the basic data structures**, which we know from the last few chapters, and to estimate with what complexity each of them performs the basic operations like **addition**, **searching**, **deletion** and **access by index** (when applicable). In that way we could easily judge according to the operations we expect to need, **which structure would be the most appropriate**. The complexities of the basic operations on the basic data structures, which we have reviewed in the previous chapters, are given in the table below:

| Data structure | Addition | Search | Deletion | Access by index |
|---|---|---|---|---|
| Array (**T[]**) | O(N) | O(N) | O(N) | O(1) |
| Linked list (**LinkedList<T>**) | O(1) | O(N) | O(N) | O(N) |
| Dynamic array (**List<T>**) | O(1) | O(N) | O(N) | O(1) |
| Stack (**Stack<T>**) | O(1) | - | O(1) | - |
| Queue (**Queue<T>**) | O(1) | - | O(1) | - |
| Dictionary, implemented with a hash-table (**Dictionary<K, T>**) | O(1) | O(1) | O(1) | - |
| Dictionary, implemented with a balanced search tree (**SortedDictionary<K, T>**) | O(log(N)) | O(log(N)) | O(log(N)) | - |
| Set, implemented with a hash-table (**HashSet<T>**) | O(1) | O(1) | O(1) | - |
| set, implemented with a balanced search tree (**SortedSet<T>**) | O(log(N)) | O(log(N)) | O(log(N)) | - |

We let the reader to think about how these complexities were estimated.

# When to Use a Particular Data Structure?

Let's skim through all the structures in the table above and explain **in what situations we should use them** as well as how their complexities are evaluated.

## Array (T[])

The arrays are **collections of fixed number of elements** from a given type (for example numbers) where the **elements preserved their order**. Each

element can be accessed through its index. The arrays are memory areas, which have a **predefined size**.

**Adding a new element** in an array is a **slow operation**. To do this we have to allocate a memory with the same size plus one and copy all the data from the original array to the new one.

**Searching** in an array **takes time** because we have to compare every element to the searched value. It takes N/2 comparisons in the average case.

**Removing an element** from an array is a **slow operation**. We have to allocate a memory with the same size minus one and copy all the old elements except the removed one.

**Accessing by index** is direct, and thus, a **fast operation**.

The arrays should be used only when we have to p**rocess a fixed number of elements** to which we need a **quick access by index**. For example, if we have to sort some numbers, we can keep them in an array and then apply some of the well-known sorting algorithms. If we have to change the elements' count, the array is not the correct data structure we should use.

**Use arrays when you have to process a fixed number of elements to which you need an access through index.**

## Singly / Doubly Linked List (LinkedList<T>)

Singly and doubly **linked lists** hold **collection of elements**, which preserve their order. Their representation in the memory is dynamic, pointer-based. They are linked sequences of element.

**Adding is a fast operation** but it is a bit slower than adding to a `List<T>` because every time when we add an element to a linked list we allocate a new memory area. The memory allocation works at speed, which cannot be easily predicted.

**Searching** in a linked list is a **slow operation** because we have to traverse through all of its elements.

**Accessing an element by index** is a slow operation because there is **no indexing** in singly and doubly linked lists. You have to go through all the elements from the start one by one instead.

**Removing** an element at a specified index is a **slow operation** because reaching the element through its index is a slow operation. Removing an element with a specified value is a slow operation too, because it involves searching.

Linked list can **quickly add and remove elements** (with a constant complexity) **at its two ends** (head and tail). Hence, it is very handy for an implementation of stacks, queues and similar data structures.

Linked lists are **rarely used** in practice because the dynamic arrays (**List<T>**) can do almost exact same operations **LinkedList** does, plus for the most of them it works faster and more comfortable.

When you need a linked list, use **List<T>** instead of **LinkedList<T>**, because it doesn't work slower and it gives you better speed and flexibility. Use **LinkedList** when you have to **add and remove elements at both ends** of the data structure.

**When you need to add and remove elements at both ends of the list, use LinkedList<T>. Otherwise use List<T>.**

## Dynamic Array (List<T>)

Dynamic array (**List<T>**) is one of the **most popular data structures** used in programming. It does not have fixed size like arrays, and allows direct access through index, unlike linked lists (**LinkedList<T>**). The dynamic array is also known as "array list", "resizable array" and "dynamic array".

**List<T>** holds its elements in an array, which has a bigger size than the count of the stored elements. Usually when we **add an element**, there is an empty cell in the list's inner array. Therefore this operation takes a **constant time**. Occasionally the array has been filled and it has to expand. This takes linear time, but it rarely happens. If we have a large amount of additions, the average-case complexity of adding an element to **List<T>** will a constant – O(1). If we sum the steps needed for adding 100,000 elements (for both cases – "fast add" and "add with expand") and divide by 100,000, we will obtain a constant which will be nearly the same like for adding 1,000,000 elements.

This statistically-averaged complexity calculated for large enough amount of operations is called **amortized complexity**. Amortized linear complexity means that if we add 10,000 elements consecutively, the overall count of steps will be of the order of 10,000. In most cases add it will execute in a constant time, while very rarely adding will execute in linear time.

Searching in **List<T>** is a **slow operation** because you have to traverse through all the elements.

**Removing by index** or value executes in a linear time. It is a **slow operation** because we have to move all the elements after the deleted one with one position to the left.

The **indexed access** in **List<T>** is instant, in a **constant time**, since the elements are internally stored in an array.

Practically **List<T> combines the best of arrays and lists**, for which it is a preferred data structure in many situations. For example if we have to process a text file and to extract from it all words (with duplicates), which

match a regular expression, the most suitable data structure in which we can accumulate them is **List<T>**, because we need a list, the length of which is unknown in advance and can grow dynamically.

The dynamic array (**List<T>**) is appropriate, when we have to add elements frequently as well as keeping their order of addition and access them through index. If we often we have to search or delete elements, **List<T>** is not the right data structure.

**Use List<T>, when you have to add elements quickly and access them through index.**

## Stack

**Stack** is a linear data structure in which there are 3 operations defined: adding an element at the top of the stack (**push**), removing an element from the top of the stack (**pop**) and inspect the element from the top without removing it (**peek**). All these operations are **very fast** – it takes a **constant time** to execute them. The stack does not support the operations search and access through index.

The stack is a data structure, which has a **LIFO behavior** (last in, first out). It is used when we have to model such a behavior – for example, if we have to keep the path to the current position in a recursive search.

**Use a stack when you have to implement the behavior "last in, first out" (LIFO).**

## Queue

**Queue** is a linear data structure in which there are two operations defined: adding an element to the tail (**enqueue**) and extract the front-positioned element from the head (**dequeue**). These two operations take a **constant time** to execute, because the queue is usually implemented with a linked list. We remind that the linked list can quickly add and remove elements from its both ends.

The queue's behavior is **FIFO (first in, first out)**. The operations searching and accessing through index are not supported. Queue can naturally model a list of waiting people, tasks or other objects, which have to be processed in the same order as they were added (enqueued).

As an example of **using a queue** we can point out the implementation of the **BFS (breadth-first search) algorithm**, in which we start from an initial element and all its neighbors are added to a queue. After that they are processed in the order they were added and their neighbors are added to the queue too. This operation is repeated until we reach the element we are looking for or we process all elements.

**Use a queue when you have to implement the behavior "first in, first out" (FIFO).**

## Dictionary, Implemented with a Hash-Table (Dictionary<K, T>)

The data structure "dictionary" suggests **storing key-value pairs** and provides a **quick search by key**. The implementation with a hash table (the class **`Dictionary<K,T>`** in .NET Framework) has a **very fast add, search and remove** of elements – **constant complexity** at the average case. The operation access through index is not available, because the elements in the hash-table have no order, i.e. an almost **random order**.

**`Dictionary<K,T>`** keeps internally the elements in an **array** and puts every element at the position calculated by the hash-function. Thus the array is partially filled – in some cells there is a value, others are empty. If more than one element should be placed in a single cell, elements are stored in a linked list. It is called **chaining**. This is one of the few ways to resolve the collision problem. When the load factor exceeds 75%, the size is doubled and all the elements occupy new positions. This operation has a linear complexity, but it is executed so rarely, that the amortized complexity remains a constant.

Hash-table has one peculiarity: if we choose a **bad hash-function** causing many collisions, the basic operations can become **very inefficient** and reach linear complexity. In practice, however, this hardly happens. Hash-table is considered to be the fastest data structure, which provides adding and searching by key.

Hash-table in .NET Framework permits **each key to be put only once**. If we add two elements with the same key consecutively, the last will replace the first and we will eventually lose an element. This important feature should be considered.

From time to time one key will have to keep multiple values. This is not standardly supported but we can store the values matching this key in a **`List<T>`** as a sequence of elements. For example if we need a hash-table **`Dictionary<int, string>`**, in which to accumulate pairs {integer, string} with duplicates, we can use **`Dictionary<int, List<string>>`**. Some external libraries have ready to use data structure called **`MultiDictionary<K,V>`**.

Hash-table is **recommended** to be used every time we need **fast addition** and **fast search by key**. For example if we have to count how many times each word is encountered in a set of words in a text file, we can use **`Dictionary<string, int>`** – the key will be a particular word, the value – how many times we have seen it.

**Use a hash-table, when you want to add and search by key very fast.**

A lot of programmers (mostly beginners) live with the delusion the main advantage of using a hash-table is the comfort of **searching a value by its key**. Actually this is wrong. We can implement searching a key with an array, a list or even a stack. There is no problem, everyone can build it. We can define a class **Entry**, which holds a key-value pair and after that we will work with an array or a list with **Entry** elements. We can implement the search but by any circumstances it will work slowly. This is the big problem with lists and arrays – they do not offer a fast search. Unlike them the hash-table can **search and add new elements very fast**.

**The main advantage of the hash-table over the other data structures is a very quick searching and addition. The comfort for the developers is a secondary factor.**

## Dictionary, Implemented with a Balanced Tree (SortedDictionary<K,T>)

The implementation of the data structure **"dictionary" as a red-black tree** (the class **SortedDictionary<K,T>**) is a structure storing key-value pairs where keys are ordered increasingly (sorted). The structure provides a **fast execution** of basic operations (add an element, search by key and remove an element). The complexity of these operations is **logarithmic** – O(log(N)). Thus, it will take 10 steps for add / search / remove when the dictionary holds 1,000 elements and 20 steps in case of 1,000,000 elements.

Unlike hash-tables, where we can reach linear complexity if we pick a bad hash-function, in **SortedDictionary<K,T>** the count of the steps of the basic operations in the average and worst case are the same – $\mathbf{log_2(N)}$. When we work with balanced trees, there is no hashing, no collisions and no risk of using a bad hash-function.

Again, as in the hash-tables, one key can be stored at most once in the structure. If we want to associate several values with one key, we should use some kind of a list for the values, for example **List<T>**.

**SortedDictionary<K,T>** holds internally its values in a **red-black balanced tree** ordered by key. This means if we traverse the structure (using its iterator or **foreach** loop in C#) we will get the elements sorted in ascending order by key. Sometimes this can be very useful property.

Use **SortedDictionary<K,T>** when you need a structure which can add, search and remove an element fast and you also need to extract the elements sorted in ascending order. In general **Dictionary<K,T>** works a bit faster than **SortedDictionary<K,T>** and is preferable.

As an example of using a **SortedDictionary<K,T>**, we can give the following task: **find all the words in a text file, which occur exactly 10 times, and print them alphabetically**. This is a task that we can solve as successful with **Dictionary<K,T>** too, but we will have to do an additional sorting at the

end. For the solution of this task we can use **SortedDictionary<string, int>** and to traverse through all the words in the text file. For each word we will keep in the sorted dictionary how many times we have encountered it. After that we can go through all the elements in the dictionary and print those words, which have been encountered exactly 10 times. They will be alphabetically ordered, since this is the natural internal order of the sorted dictionary data structure.

**Use SortedDictionary<K,T> when you want fast addition of elements and searching by key as well as the elements to be sorted by key.**

## Set, Implemented with a Hash-Table (HashSet<T>)

The data structure **"set" is a collection of elements with no duplicates**. The basic operations are adding an element to the set, checking if an element belongs to the set (searching) and removing an element from the set. The operation searching through index is not supported, i.e. we do not have a direct access to the elements via ordering number, because in this structure there is not any order.

Set, implemented with a **hash-table** (the class **HashSet<T>**) is a special case of a hash-table, in which we have only keys. The values associated with these keys do not matter.

As in the hash-table, the basic operations in the data structure **HashSet<T>** are implemented with a **constant complexity O(1)**. Another similarity to hash-table is if we choose a bad hash-function, we can reach a linear complexity executing the basic operations. Fortunately in practice this almost never happens.

As an example of using a **HashSet<T>**, we can point out the task of finding all the different words in a text file.

**Use HashSet<T>, when you have to quickly add elements to a set and check whether a given element belongs to a set.**

## Set, Implemented with a Balanced Tree (SortedSet<T>)

The data structure set, implemented with a red-black tree, is a special case of **SortedDictionary<K,T>** in which keys and values coincide.

Similar to **SortedDictionary<K,T>**, the basic operations in **SortedSet<T>** are executed with logarithmic complexity **O(log(N))**, which is the same in the average and worst case.

As an example of using a **SortedSet<T>** we can point out the task of finding all the different words in a given text file and printing them alphabetically ordered.

**Use SortedSet<T>, when you have to quickly add an element to a set and check whether given element belongs to the set as well as need all the elements sorted in ascending order.**

# Choosing a Data Structure – Examples

We are going to **show several problems**, where the choice of an appropriate data structure is crucial to the efficiency of their solution. The purpose of this is to show you typical situations, in which the reviewed data structures are used and to teach you in what scenarios what data structures you should use.

## Generating Subsets

It is given a set of strings **S**. For example **S** = {ocean, beer, money, happiness}. The task is to write a program, which prints **all subsets of S**.

The problem has many and different solutions, but we are going to focus on the following one: We **start from the empty set** (with 0 elements):

```
{}
```

We **add to it every element of S** and we get a collection of subsets with one element:

```
{ocean}, {beer}, {money}, {happiness}
```

To each of the one-elemental subsets we **add every element from S**, which has not been added yet to the corresponding subset and now we have all two-elemental subsets:

```
{ocean, beer}, {ocean, money}, {ocen, happiness}, {beer, money},
{beer, happiness}, {money, happiness}
```

If we keep on the same way, we will get all 3-elemental subsets and after that all 4-elemental etc. to the N-elemental subsets.

How to implement this algorithm? We have to choose **appropriate data structures**, right?

We can start with the data structure keeping the initial set of elements **S**. It can be an **array, linked list, dynamic array** (**List<string>**) or **set**, implemented as **SortedSet<string>** or **HashSet<string>**. To answer the question which structure is the most appropriate, let's think of which are the operations we are going to do on this structure. We can think of only one operation –

traversing through all the elements of **S**. This operation can be implemented efficiently with any of these structures. We **choose an array** because it is the simplest data structure of all and it is easy to work with.

The next step is to **pick a structure** in which we will store **one of the subsets** we generate, for example {ocean, happiness}. Again we ask ourselves the question what are the operations we execute on this subset of words. The operations are a check whether an element exists and an addition of an element, right? Which data structure quickly implements both operations? The arrays and lists do not search quickly, dictionaries store key-value pairs, which is not our case. Almost no options are left, so we are going to see what the data structure set offers. It supports a quick searching and addition. Which implementation to choose – **SortedSet<string>** or **HashSet<string>**? We do not have a requirement to sort the words in alphabetical order, so we **choose the faster implementation – HashSet<string>**.

Lastly, we will choose one more data structure in which we are going to keep the **collection of the subsets** of words, for example:

```
{ocean, beer}, {sea, money}, {sea, happiness}, {beer, money},
{beer, happiness}, {money, happiness}
```

Using this structure we have to be able to add as well as traverse through all its elements consecutively. The following structures meet the requirements: **list, stack, queue and set**. In each of them we can add quickly and go through its elements. If we examine the algorithm for generating subsets, we will notice each is processed in style: "**first generated, first processed**". The subset, which had been firstly generated, has been firstly processed and subsets with one more element have been generated from it, right? Therefore our algorithm will most accurately fit the data structure **queue**. We can describe the algorithm as follows:

1. We start with a queue, containing the empty set **{}**.
2. We dequeue an element called **subset** and try to add each element from S which **subset** does not contain. The result is a set, which we enqueue.
3. We repeat step 2 until the queue becomes empty.

You can see how a few thoughts brought us to the classical algorithm "**breadth-first search**" (**BFS**). Once we know what data structures we should use, implementation is quick and easy. Here is how it might look:

```
string[] words = {"ocean", "beer", "money", "happiness"};
Queue<HashSet<string>> subsetsQueue =
  new Queue<HashSet<string>>();
HashSet<string> emptySet = new HashSet<string>();
subsetsQueue.Enqueue(emptySet);
while (subsetsQueue.Count > 0)
```

```
{
   HashSet<String> subset = subsetsQueue.Dequeue();

   // Print current subset
   Console.Write("{ ");
   foreach (string word in subset)
   {
      Console.Write("{0} ", word);
   }
   Console.WriteLine("}");

   // Generate and enqueue all possible child subsets
   foreach (string element in words)
   {
      if (! subset.Contains(element))
      {
         HashSet<string> newSubset = new HashSet<string>();
         newSubset.UnionWith(subset);
         newSubset.Add(element);
         subsetsQueue.Enqueue(newSubset);
      }
   }
}
```

If we execute the code above, we will see that it successfully **generates all subsets of S**, but some of them are **generated twice**.

```
{ }
{ ocean }
{ beer }
{ money }
{ happiness }
{ ocean beer }
{ ocean money }
{ ocean happiness }
{ beer ocean }
…
```

In the example the subsets **{ ocean beer }** and **{ beer ocean }** are actually one and the same subset. It seems we have not thought of duplicates, which occur when we mix the order of elements in the same subset. How can we avoid duplicates?

Let's associate the words by their indices.

**ocean → 0**

**beer → 1**
**money → 2**
**happiness → 3**

Since the subsets {1, 2, 3} and {2, 1, 3} are actually one and a same subset, in order to avoid duplicates, we are going to impose a requirement to generate only subsets, in which the **indices are in ascending order**. Instead of subsets of words we can keep subsets of indices, right? In these subsets of indices we need two operations: adding an index and getting the biggest index so we can add only indices bigger than it. Obviously we do not need **HashSet<T>** anymore, but we can successfully use **List<T>**, in which the elements are ordered in ascending order by index and the biggest element is naturally placed last.

Finally, our algorithm looks something like this:

1. Let **N** be the number of elements in **S**. We start with a **queue**, containing the **empty set {}**.
2. We dequeue an element called **subset**. Let **start** be the biggest index in **subset**. We add to **subset** all indices, which are bigger than **start** and smaller than N. As a result we get several new subsets, which we enqueue.
3. Repeat step 2 until the queue is empty.

Here is how the **implementation of the new algorithm** looks like:

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

public class Subsets
{
  static string[] words = { "ocean", "beer", "money",
    "happiness" };

  static void Main()
  {
    Queue<List<int>> subsetsQueue = new Queue<List<int>>();
    List<int> emptySet = new List<int>();
    subsetsQueue.Enqueue(emptySet);
    while (subsetsQueue.Count > 0)
    {
      List<int> subset = subsetsQueue.Dequeue();
      Print(subset);
      int start = -1;
      if (subset.Count > 0)
      {
        start = subset[subset.Count - 1];
```

```
            }
            for (int i = start + 1; i < words.Length; i++)
            {
                List<int> newSubset = new List<int>();
                newSubset.AddRange(subset);
                newSubset.Add(i);
                subsetsQueue.Enqueue(newSubset);
            }
        }
    }

    static void Print(List<int> subset) {
        Console.Write("[ ");
        for (int i=0; i<subset.Count; i++) {
            int index = subset[i];
            Console.Write("{0} ", words[index]);
        }
        Console.WriteLine("]");
    }
}
```

If we run the program we will get the following **correct result**:

```
[ ]
[ ocean ]
[ beer ]
[ money ]
[ happiness ]
[ ocean beer ]
[ ocean money ]
[ ocean happiness ]
[ beer money ]
[ beer happiness ]
[ money happiness ]
[ ocean beer money ]
[ ocean beer happiness ]
[ ocean money happiness ]
[ beer money happiness ]
[ ocean beer money happiness ]
```

## Sorting Students

It is given a **text file**, containing the data of a group of **students and courses** which they attend, separated by |. The file looks like this:

```
Chris | Jones | C#
Mia | Smith | PHP
Chris | Jones | Java
Peter | Jones | C#
Sophia | Wilson | Java
Mia | Wilson | C#
Mia | Smith | C#
```

Write a program **printing all courses and the students**, who have joined them, **ordered by last name**, and then by first name (if the last names match).

We can implement the problem using a **hash-table**, which will hold a list of students by a course name. We are choosing a hash-table, because we can **quickly search by course name** in it.

In order to meet the requirements for an order by name and surname, we are going to **sort the particular list of students** from each course, before we print it. Another option is to use **SortedSet<T>** for the students attending each course (because it is internally sorted), but since one can have students with the same name, we have to use **SortedSet<List<String>>**. It becomes too complicated. We choose the easier way – using **List<Student>** and sorting it before we print it.

In any case we will have to implement the **IComparable** interface so we can define **the order of the elements** of type **Student** according to the task requirements. It is important to firstly compare the family names and if they are the same to compare the first names. We remind that in order to sort the elements of a given class it is explicitly necessary to define the logic of their order. In .NET Framework this is done by the **IComparable<T>** interface (or through lambda functions like we shall see in the chapter "Lambda Expressions and LINQ"). Let's define the class **Student** and implement **IComparable<Student>**. We get something like this:

```
public class Student : IComparable<Student>
{
   private string firstName;
   private string lastName;

   public Student(string firstName, string lastName)
   {
      this.firstName = firstName;
      this.lastName = lastName;
   }

   public int CompareTo(Student student)
   {
```

```
      int result = lastName.CompareTo(student.lastName);
      if (result == 0)
      {
         result = firstName.CompareTo(student.firstName);
      }
      return result;
   }

   public override String ToString()
   {
      return firstName + " " + lastName;
   }
}
```

Now we are able to write the code, which **reads the students and their courses and stores them in a hash-table**, which keeps a list of students by a course name (**Dictionary<string, List<Student>>**). And then it is easy – we iterate over the courses, sort the students and print them:

```
// Read the file and build the hash-table of courses
Dictionary<string, List<Student>> courses =
   new Dictionary<string, List<Student>>();
StreamReader reader = new StreamReader("Students.txt");
using (reader)
{
   while (true)
   {
      string line = reader.ReadLine();
      if (line == null)
      {
         break;
      }
      string[] entry = line.Split(new char[] { '|' });
      string firstName = entry[0].Trim();
      string lastName = entry[1].Trim();
      string course = entry[2].Trim();
      List<Student> students;
      if (! courses.TryGetValue(course, out students))
      {
         // New course -> create a list of students for it
         students = new List<Student>();
         courses.Add(course, students);
      }
      Student student = new Student(firstName, lastName);
      students.Add(student);
```

```
    }
}

// Print the courses and their students
foreach (string course in courses.Keys)
{
    Console.WriteLine("Course " + course + ":");
    List<Student> students = courses[course];
    students.Sort();
    foreach (Student student in students)
    {
        Console.WriteLine("\t{0}", student);
    }
}
```

The above code first **parses all the lines** consecutively by splitting them by a vertical bar "|". Secondly it cleans the spaces from the beginning and the end. After storing every student's information, it is **checked in the hash-table** whether its course exists. If the course has been found, the student is added to the list of the students of this course. Otherwise, a new list is created and the student is added to it. Then the list is added in the hash-table using the course name as a key.

**Printing the courses** and students is not difficult. All keys are extracted from the hash-table. These are the names of the courses. For each course its students' list is **extracted, sorted and printed**. The sorting is made by the built-in method **Sort()** using the comparison method **CompareTo(…)** from the interface **IComparable<T>** as defined in the class **Student** (comparison firstly by family name, and if they are the same, comparison by first name). At the end the sorted students are printed by the overridden virtual method **ToString()**. Here is how the output of the upper program looks:

```
Course C#:
        Chris Jones
        Peter Jones
        Mia Smith
        Mia Wilson
Course PHP:
        Mia Smith
Course Java:
        Chris Jones
        Sophia Wilson
```

## Sorting a Phone Book

It is given a **text file, containing people's names, their city names and phone numbers**. The file looks like this:

```
Kenneth | Virginia Beach | 1-541-754-3010
Paul    | San Antonio    | 1-535-675-6745
Mary    | Portland       | 1-234-765-1983
Laura   | San Antonio    | 1-454-345-2345
Donna   | Virginia Beach | 1-387-387-2389
```

Write a program which prints all the **city names in an alphabetical order** and for each one of them prints all **people's names in alphabetical order** and their corresponding **phone number**.

The problem can be solved in many ways, for example we sort by two criteria: firstly by city name and secondly by person name and then we print the phone book.

However, let's solve the problem without sorting, but by using the standard data structures in .NET Framework. We want the city names to be sorted. This means that it is best to use a data structure, which internally keeps the elements **sorted**. Such as, for example, a balanced search tree – **SortedSet<T>** or **SortedDictionary<K,T>**. Since every record from the phone book contains beside a city name – other data, it is more convenient to have a **SortedDictionary<K,T>**, which keeps a list of people's names and their phone numbers. We want the list of the people's names from every city to be sorted in alphabetical order by name. Hence, we can use the data structure **SortedDictionary<K,T>** again. The key will be the name of the person and its value will be his phone number.

At the end we get the nested structure **SortedDictionary<string, SortedDictionary<string, string>>**. Here is a sample implementation, which shows how we can solve the problem using this structure:

```
// Read the file and build the phone book
SortedDictionary<string, SortedDictionary<string, string>>
   phonesByTown = new SortedDictionary<string,
      SortedDictionary<string, string>>();
StreamReader reader = new StreamReader("PhoneBook.txt");
using (reader)
{
   while (true)
   {
      string line = reader.ReadLine();
      if (line == null)
      {
         break;
```

```
        }
        string[] entry = line.Split(new char[]{'|'});
        string name = entry[0].Trim();
        string town = entry[1].Trim();
        string phone = entry[2].Trim();

        SortedDictionary<string, string> phoneBook;
        if (! phonesByTown.TryGetValue(town, out phoneBook))
        {
            // This town is new. Create a phone book for it
            phoneBook = new SortedDictionary<string, string>();
            phonesByTown.Add(town, phoneBook);
        }
        phoneBook.Add(name, phone);
    }
}

// Print the phone book by towns
foreach (string town in phonesByTown.Keys)
{
    Console.WriteLine("Town " + town + ":");
    SortedDictionary<string, string> phoneBook =
        phonesByTown[town];
    foreach (var entry in phoneBook)
    {
        string name = entry.Key;
        string phone = entry.Value;
        Console.WriteLine("\t{0} - {1}", name, phone);
    }
}
```

If we execute this sample code with input – the sample phone book, we will get the expected result:

```
Town Portland:
        Mary - 1-234-765-1983
Town San Antonio:
        Laura - 1-454-345-2345
        Paul - 1-535-675-6745
Town Virginia Beach:
        Donna - 1-387-387-2389
        Kenneth - 1-541-754-3010
```

## Searching in a Phone Book

Here is another problem, so we can strengthen the way, in which we think in order to **choose appropriate data structures**. A **phone book** is stored in a text file, containing **names of people, their city names and phone numbers**. People's names can be in the format first name or nickname or first name + last name or first name + surname + last name. The file could have the following look:

```
Kevin Clark       | Virginia beach | 1-454-345-2345
Skiller           | San Antonio    | 1-566-533-2789
Kevin Clark Jones | Portland       | 1-432-556-6533
Linda Johnson     | San Antonio    | 1-123-345-2456
Kevin             | Phoenix        | 1-564-254-4352
Kevin Garcia      | Virginia Beach | 1-445-456-6732
Kevin             | Phoenix        | 1-134-654-7424
```

It is possible several people to be given under the same name or even the same city. It is possible someone to have **several phone numbers**. In this case he is given several times in the input file. **Phone book could be huge** (up to 1,000,000 records).

A file holding a sequence of queries is given. The **queries** are two types:

- **Search by name / nickname / surname / last name**. The query looks like this: **`list(name)`**.
- **Search by name / nickname / surname / last name + city name**. The query looks like this: **`find(name, town)`**.

Here is a sample query file:

```
list(Kevin)
find(Angel, San Antonio)
list(Linda)
list(Clark)
find(Jones, Phoenix)
list(Grandma)
```

Write a program, which by given phone book and query file **executes and respond to all the queries**. For each query a list of records in the phone book has to be printed or the message "**Not found**", if the query cannot find anything. **Queries could be a large number**, for example 50,000.

This problem is not as easy as the previous ones. One easy to implement solution could be to **scan the entire phone book** for every query and extract all records in which there is a match with the searched information. But this **will work slowly**, because the records and queries could be a lot. It is necessary to find a way for a quick search without scanning the entire phone book every time.

In paper phone books the numbers are given by the **people's name, sorted in alphabetical order**. Sorting will not help us, because someone can search by first name, other – by last name, third – by nickname and city name. We have to search by any of the above at the same time. The question is **how do we do it**?

If we think a bit it will occur to us that the problem requires **searching by any of the words**, which can be seen at the first column of the phone book and eventually by the combination of a word from the first column and a town from the second one. We know that the fastest search is implemented with a **hash-table**. So far so good, but what are we going to keep as a key and value in the hash-table?

What if we use **several hash-tables**: one for searching a word from the first column, another for searching in the second column, third for searching by city and so on? If we think a bit more, we will ask ourselves the question – why do we need several hash-tables? Can't we **search in one common hash-table**? If we have a name "Peter Jones", we will store his phone number under the keys "Peter" and "Jones". If someone searches by any of these keys, he will find Peter's phone number.

So far so good, but how do we **search by both first name and city name**, for example "Peter from Virginia Beach"? It is possible firstly to find all with a name "Peter" and then to print those who are from Virginia Beach. This will work, but if there are a lot of people named Peter, searching will work slowly. Then why don't we make a **hash-table with a key name of a person and value another hash-table**, which by city name will return a list of phone numbers? This could work. We have done something similar in the previous task, haven't we?

Can we come up with **something smarter**? Can't we just put the phone numbers of all the people named Peter from Virginia Beach under a key "**Peter from Virginia Beach**" in the main hash-table in the phone book? It seems this could solve our problem and we will use only one hash-table for all the queries.

Using the last idea, we could invent the following **algorithm**: we read line by line from the phone book and **for each word from the name of a person** $d_1$, $d_2$, …, $d_k$ and **for each city name** `t` we **make new records in the phonebook hash-table** by the following keys: $d_1$, $d_2$, …, $d_k$, "$d_1$ `from t`", "$d_2$ `from t`", …, "$d_k$ `from t`". Now it is guaranteed we could search by any of a person's names as well as name and town. In order to search without bothering about letter case we can transform the words to lowercase in advance. After that the searching is trivial – we just search in the hash-table by a given word `d` or if a town `t` is given "`d from t`". Since we could have many phone numbers under the same key, for a value in the hash-table we should use a list of strings (`List<string>`).

Let's skim through an implementation of the described algorithm:

```
class PhoneBookFinder
{
   const string PhoneBookFileName = "PhoneBook.txt";
   const string QueriesFileName = "Queries.txt";

   static Dictionary<string, List<string>> phoneBook =
      new Dictionary<string, List<string>>();

   static void Main()
   {
      ReadPhoneBook();
      ProcessQueries();
   }

   static void ReadPhoneBook()
   {
      StreamReader reader = new StreamReader(PhoneBookFileName);
      using (reader)
      {
         while (true)
         {
            string line = reader.ReadLine();
            if (line == null)
            {
               break;
            }
            string[] entry = line.Split(new char[]{'|'});
            string names = entry[0].Trim();
            string town = entry[1].Trim();

            string[] nameTokens =
               names.Split(new char[] {' ', '\t'} );
            foreach (string name in nameTokens)
            {
               AddToPhoneBook(name, line);
               string nameAndTown = CombineNameAndTown(town, name);
               AddToPhoneBook(nameAndTown, line);
            }
         }
      }
   }

   static string CombineNameAndTown(   string town, string name)
   {
```

```
    return name + " from " + town;
  }

  static void AddToPhoneBook(string name, string entry)
  {
    name = name.ToLower();
    List<string> entries;
    if (! phoneBook.TryGetValue(name, out entries))
    {
      entries = new List<string>();
      phoneBook.Add(name, entries);
    }
    entries.Add(entry);
  }

  static void ProcessQueries()
  {
    StreamReader reader = new StreamReader(QueriesFileName);
    using (reader)
    {
      while (true)
      {
        string query = reader.ReadLine();
        if (query == null)
        {
          break;
        }
        ProcessQuery(query);
      }
    }
  }

  static void ProcessQuery(string query)
  {
    if (query.StartsWith("list("))
    {
      int listLen = "list(".Length;
      string name = query.Substring(
        listLen, query.Length-listLen-1);
      name = name.Trim().ToLower();
      PrintAllMatches(name);
    }
    else if (query.StartsWith("find("))
    {
```

```
            string[] queryParams = query.Split(
              new char[] { '(', ' ', ',', ')' },
              StringSplitOptions.RemoveEmptyEntries);
            string name = queryParams[1];
            name = name.Trim().ToLower();
            string town = queryParams[2];
            town = town.Trim().ToLower();
            string nameAndTown =
              CombineNameAndTown(town, name);
            PrintAllMatches(nameAndTown);
        }
        else
        {
            Console.WriteLine(
              query + " is invalid command!");
        }
    }

    static void PrintAllMatches(string key)
    {
        List<string> allMatches;
        if (phoneBook.TryGetValue(key, out allMatches))
        {
            foreach (string entry in allMatches)
            {
                Console.WriteLine(entry);
            }
        }
        else
        {
            Console.WriteLine("Not found!");
        }
        Console.WriteLine();
    }
}
```

While reading the phone book line by line and splitting by the vertical bar "|", we **extract the three columns** (names, city name and phone number). After that the names are split and each word is **added in the hash-table**. Additionally we add each word, combined with the city name (so that we can search by name + city name).

The second part of the algorithm is the **command execution**. In this part each line from the query file is read and processed. The process includes parsing the command, extracting the name or name and city name and

searching. The search is directly done by using the **hash-table**, which is created after reading the phone book file.

To be able to ignore the difference between lowercase and uppercase, all keys in the hash-table are added as lowercase. When we search, we do it lowercase too.

## Choosing a Data Structure – Conclusions

By the many examples it is clear that the choice of an appropriate data structure is **highly dependable on the specific task**. Sometimes **data structures have to be combined** or we have to use several of them simultaneously.

What data structure should we pick mostly **depends on the operations we will perform**, so always ask yourselves "what operations should the structure, I need, perform efficiently". If you are familiar with the operations, you can easily conform which structure does them most efficiently and at the same time is easy and handy.

In order to efficiently choose an appropriate data structure, you should firstly **invent the algorithm**, which you are going to implement, and then **look for an appropriate data structures** for it.

**Always go from the algorithm to the data structures, never backwards.**

# External Libraries with .NET Collections

It is a well-known fact that the standard data structures in .NET Framework **`System.Collections.Generic`** have pretty poor functionality. It lacks implementations of basic concepts in data structures such as multi-sets, priority queues, for which there should be standard classes as well as basic system interfaces.

When we have to use a **special data structure**, which is not standardly implemented in .NET Framework, we have two options:

- First option: we **implement the data structure ourselves**. This gives us flexibility, because the implementation will completely meet our needs, but it takes a lot of time and it has a great chance of making mistakes. For example, if one has to qualitatively implement a balanced tree, this may take an experienced software developer several days (along with the tests). If the same is implemented by inexperienced software developer it will take a lot more time and most probably there will be errors in the implementation.
- Second option (generally preferable): **find an external library**, which has a full implementation of the needed functionality. This approach has an advantage of saving us time and troubles, because in most cases the external libraries of data structures are well-tested. They have been

used for years by thousands of software developers and this makes them mature and reliable.

## Power Collections for .NET

One of the most popular and richest libraries with efficient implementations of the fundamental data structures for C# and .NET software developers is the open-source project "**Wintellect's Power Collections for .NET**" – http://powercollections.codeplex.com. It provides free, reliable, efficient, fast and handy implementations of the following commonly used **data structures**, which are missing or partly-implemented in .NET framework:

- **`Set<T>`** – **set** of elements, **implemented with a hash-table**. It efficiently implements the basic operations over sets: adding, deleting and searching an element as well as union, intersection, difference between sets and many more. By functionality and way of work the class looks like the standard class **`HashSet<T>`** in .NET Framework.
- **`Bag<T>`** – **multi-set of elements** (set with duplicates), implemented with a **hash-table**. It efficiently implements all basic operations over multi-sets.
- **`OrderedSet<T>`** – **ordered set of elements** (without duplicates), implemented with a **balanced search tree**. It efficiently implements all basic operations over sets and when traversing through its elements it returns them in ascending order (according to the used comparer). It allows a fast extraction of subsets of values in a given interval.
- **`OrderedBag<T>`** – **ordered multi-set** of elements, implemented with a **balanced search tree**. It efficiently implements all basic operations over multi-sets and when going through all its elements it returns them in ascending order (according to the used comparer). It allows a quick extraction of subsets of values in a given interval.
- **`MultiDictionary<K,T>`** – it is a **hash-table allowing key duplicates**. For every key there is a collection of values stored, not one single value.
- **`OrderedDictionary<K,T>`** – it represents a **dictionary, implemented with a balanced search tree**. It allows a fast search by key and when going through its elements it returns them in ascending order. It enables us to quickly extract the values from a given key range. By functionality and way of work the class looks like the standard class **`SortedDictionary<K,T>`** in .NET Framework.
- **`Deque<T>`** – represents efficient implementation of a queue with two ends (**double ended queue**), which practically combines the data structures stack and queue. It allows efficient addition, extraction and deletion of elements in both ends.
- **`BagList<T>`** – **list of elements, accessed through index**, which allows a **quick insertion and deletion** of an element from a particular position. The operations index accessing, adding, inserting at position

and removing an element from position have a complexity O(log N). The implementtation is with a balanced tree. The structure is a good alternative of **List<T>**, in which the insertion and removal of element at a particular position takes linear time because of the need of the replacement of linear number of elements to the left or right.

We let the reader the opportunity to download the library "**Power Collections for .NET**" from its site and to **experiment with it**. It can be very useful when you solve some of the problems from the exercises.

## C5 Collections for .NET

Another very powerful library of data structures and collection classes is "**The C5 Generic Collection Library for C# and CLI**" (www.itu.dk/research/c5/). It provides standard interfaces and collection classes like **lists**, **sets**, **bags**, **multi-sets**, **balanced trees** and **hash tables**, as well as **non-traditional data structures** like "hashed linked list", "wrapped arrays" and "interval heaps". It also describes a set of collection-related **algorithms** and **patterns**, such as "read-only access", "random selection", "removing duplicates", etc. The library comes with solid documentation (a book of 250 pages). The C5 collections and the book about them are the ultimate resource for data structure developers.

# Exercises

1. Hash-tables do not allow storing **more than one value in a key**. How can we get around this restriction? Define a class to hold multiple values in a hash-table.
2. Implement a data structure, which can quickly do the following two operations: **add an element** and **extract the smallest element**. The structure should accept adding duplicated elements.
3. It is given a text file **students.txt** containing information about students and their specialty in the following format:

```
Steven Davis | Computer Science
Joseph Johnson | Software Engeneering
Helen Mitchell | Public Relations
Nicolas Carter | Computer Science
Susan Green | Public Relations
William Johnson | Software Engeneering
```

Using **SortedDictionary<K,T>** print on the console the specialties in an alphabetical order and for each of them print the names of the students, firstly sorted by family name and secondly – by first name, as shown:

```
Computer Sciences: Nicolas Carter, Steven Davis
Public Relations: Susan Green, Helen Mitchell
```

```
Software Engeneering: Joseph Johnson, William Johnson
```

4. Implement a class **BiDictionary<K1,K2,T>,** which allows adding triplets **{key1, key2, value}** and quickly search by either of the keys **key1**, **key2** as well as searching by combination of the both keys. Note: Adding many elements with the same keys is allowed.
5. A big chain of supermarkets sell **millions of products**. Each of them has a unique number (barcode), producer, name and price. What data structure could we use in order to quickly **find all products, which cost between 5 and 10 dollars**?
6. A **timetable** of a conference hall is a list of events in a format **[starting date and time; ending date and time; event's name]**. What data structure could we to be able to quickly **add events** and **quickly check whether the hall is available in a given interval** [starting date and time; ending date and time]?
7. Implement the data structure **PriorityQueue<T>**, which offers quick execution of the following operations: **adding an element**, **extracting the smallest element**.
8. Imagine you **develop a search engine**, which gathers all the advertisements for used cars in ten websites for the last few years. After that the search engine allows a quick search by one or several criteria: a brand, model, color, year of production and price. You are not allowed to use database management system (like SQL Server, MySQL or MongoDB) and you must implement your own indexing in the memory, without storing it to the hard disk and without using LINQ. When one searches by price minimal and maximal price is given. When one searches by year of production a starting and ending years are given. What data structures would you use in order to ensure fast searching by one or several criteria?

## Solutions and Guidelines

1. You can use **Dictionary<key, List<value>>** or create your own class **ValuesCollection**, which can take care of the values with the same key and use **Dictionary<key, ValuesCollection>**.
2. You can use **SortedSet<List<int>>** and its operations **Add()** and **First()**. **SortedSet<T>** keeps the elements in it sorted and can accept external **IComparer<T>**.

   The problem has a more efficient solution though – the data structure called "**binary heap**". You can read about it on Wikipedia: http://en.wikipedia.org/wiki/Binary_heap.
3. The task is similar to the one from the section "Sorting Students".
4. One of the solutions to this task is to use two instances of the class **Dictionary<K,T>** for each of the two keys and when you add or remove

an element from **BiDictionary<K1,K2,T>**, you add or remove the element from the **two hash-tables** correspondingly. When you search by first or second key, you should check the elements in the first or the second hash-table respectively. When you search by two keys, you could search in the two hash-tables separately and intersect the matching subsets.

Another, simpler approach is to hold 3 hash tables: **Dictionary<K1,T>**, **Dictionary<K2,T>** and **Dictionary<Tuple<K1,K2>,T>**. The system generic class **Tuple<K1,K2>** can be used to combine two keys and use it as a **composite key**.

5. If we keep the products sorted by price in an **array** (for example in **List<Product>**, which we firstly fill and then **sort**), in order to find all the products which cost between 5 and 10 bucks we can use a **binary search** twice. Firstly we can find the smallest index **start**, in which lies a product costing at least 5 bucks. After that we can find the biggest index **end**, in which lies a product costing at most 10 bucks. All the products at positions in the interval [**start** … **end**] will cost between 5 and 10 dollars. If you are interested in the algorithm binary search in a sorted array you could inform yourself reading Wikipedia: http://en.wikipedia.org/wiki/Binary_search.

   Generally the approach using a sorted array and binary search in it works excellent, but there is a disadvantage: the addition in a sorted array is a very slow operation, because it requires moving a linear number of elements with one position ahead of the inserted new element.

   To overcome this we can use the class **SortedSet<T>**. It supports **fast insertion** keeping the elements in a **sorted order**. It has an operation **SortedSet<T>.GetViewBetween(lowerBound, upperBound)** that returns a subset of the elements in certain **range** (interval).

   You may also use the class **OrderedSet<T>** from "Wintellect's Power Collections for .NET" library (http://www.codeplex.com/PowerCollections) which is more powerful and more flexible. It has a method for extracting a sub-range of values: **OrderedSet<T>.Range(from, fromInclusive, to, toInclusive)**.

6. We can create **two sorted arrays** (**List<Event>**): the first will keep the events sorted in ascending order by **starting date and time**; the second will keep the same events sorted by **ending date and time**. By using binary search we can find all the events which can be partly or fully found between the two moments of time [**start, end**] by doing the following:

   - Find the **set S** of all events starting after the moment **start** (using binary search).
   - We can find all the **set E** of all events ending before the moment **end** (using binary search).
   - **Intersect** these two sets: **C = S ∩ E**. If the intersection **S** of the two sets of events have common elements (**S** in non-empty set), then in

the searched interval [**start** … **end**] the hall is occupied. Otherwise it is available.

This solution has a **disadvantage**: adding elements in the sorted arrays will be slow. We should either add all elements initially and then sort the two arrays and never change them afterwards or try to keep the arrays sorted when adding new elements (which will be slow).

Another solution, which is **easier** to implement and **more efficient**, is to use two instances of the class **`OrderedBag<T>`** from the "Power Collections for .NET" library (the first with event's **start** date and time as a key and the second with event's **end** date and time as a key). The class has methods to extract the subsets **S** and **E**: **`RangeFrom(from, fromInclusive)`** and **`RangeTo(to, toInclusive)`**. We still will need to intersect these sets and check whether their intersection is empty or not.

The most efficient solution is to use a data structure called "**interval tree**". Read more in Wikipedia: http://en.wikipedia.org/wiki/Interval_tree. You may find an **open source C# interval tree** implementation in CodePlex: http://intervaltree.codeplex.com.

7. Since there is no internal implementation of the data structure "**priority queue**" in .NET, you can use the data structure **`OrderedBag<T>`** from Wintellect's Power Collections. It had **`Add(…)`** and **`GetFirst()`** and **`RemoveFirst()`** methods. You can read more about priority queues on Wikipedia: http://en.wikipedia.org/wiki/Priority_Queue.

   The classic, **simplest efficient priority queue** implementation the data structure "**binary heap**": http://en.wikipedia.org/wiki/Binary_heap.

   An efficient ready-to-use C# implementation of priority queue is the class **`IntervalHeap<T>`** in the C5 Collections: http://www.itu.dk/research/c5/.

8. For **searching by brand, model and color** we can use one hash-table per each, which will search by a given criteria and return a list of cars (**`Dictionary<string, List<Car>>`**).

   For searching by **year of production** and **price range** we can use lists **`List<Car>`**, sorted in ascending order (and binary search).

   To search by several criteria at once we can extract the cars' **subsets by the first criteria**, after that the cars' **subsets by the second criteria** and so on. At the end we can find the **intersection** of the sets. Intersection of two sets can be found by looking for every element in the smaller set in the bigger set. The easiest way is **`Car`** to implement **`Equals()`** and **`GetHashCode()`** and after that to use the class **`HashSet<Car>`** for set intersections.