# Exercises: Implement a Collection of Persons

This document defines the **in-class exercises** assignments for the ["Data Structures" course @ Software University](https://softuni.bg/trainings/1147/Data-Structures-June-2015). You have to implement a **collection of persons** that performs fast enough the following operations:

* **Add-Person(*email*, *name*, *age*, *town*)**
  + The *email* is unique (it uniquely identities the person)
  + If the *email* already exists returns **false** (without adding the person), otherwise return **true**
* **Find-Person(*email*)**
  + Returns the **Person** object or **null** (if it does not exits)
* **Delete-Person(*email*)**
  + Returns **true** (successfully deleted) or **false** (not found)
* **Find-People(*email\_domain*)**
  + Returns a sequence of matched persons sorted by *email*
* **Find-People(*name*, *town*)**
  + Returns a sequence of matched persons sorted by *email*
* **Find-People(*start\_age*, *end\_age*)**
  + Returns a sequence of matched persons sorted by *age*, then by *email* (as second criteria)
* **Find-People(*start\_age*, *end\_age*, *town*)**
  + Returns a sequence of matched persons sorted by *age*, then by *email* (as second criteria)

## Collection of Persons – Project Skeleton

You are given a **Visual Studio project skeleton** (unfinished project) holding the unfinished classes PersonCollectionSlow and PersonCollection and **unit tests** covering the functionality of the “persons collection” data structure. The project holds the following assets:

TODO

The project skeleton opens correctly in **Visual Studio 2013** but can be open in other Visual Studio versions as well and also can run in **SharpDevelop** and **Xamarin Studio**. Your goal is to implement the missing functionality in order to finish the project.

First, let's take a look at the BinaryHeap<T> class. It holds a **binary heap** of parameterized type T. You need to finish it:

TODO

The project comes also with **unit tests** covering the functionality of the **binary heap** (see the class PersonCollectionSlow):

TODO

## Run the Unit Tests to Ensure All of Them Initially Fail

**Run the unit tests** from the UnitTestPersonCollection class. All of them should fail:

TODO

This is quite normal. We have unit tests, but the code covered by these tests is missing. Let's write it.

## Implement a Straightforward (and Slow) Solution

First, let’s start with a simple solution: implement the “person collection” data structure as List<Person>. The finder methods could be implemented by **LINQ queries** with filtering and sorting.

TODO

Sample code:

|  |
| --- |
| public class PersonCollectionSlow : IPersonCollection  {  private List<Person> persons = new List<Person>();  public bool AddPerson(string email, string name, int age, string town)  {  if (this.FindPerson(email) != null)  {  // Person already exists  return false;  }  var person = new Person() {  Email = email, Name = name, Age = age, Town = town };  this.persons.Add(person);  return true;  }  public int Count  {  get { return this.persons.Count; }  }  public Person FindPerson(string email)  {  return this.persons.FirstOrDefault(p => p.Email == email);  }  public bool DeletePerson(string email)  {  var person = this.FindPerson(email);  return this.persons.Remove(person);  }  public IEnumerable<Person> FindPersons(string emailDomain)  {  return this.persons  .Where(p => p.Email.EndsWith("@" + emailDomain))  .OrderBy(p => p.Email);  }  public IEnumerable<Person> FindPersons(string name, string town)  {  return this.persons  .Where(p => p.Name == name && p.Town == town)  .OrderBy(p => p.Email);  }  public IEnumerable<Person> FindPersons(int startAge, int endAge)  {  return this.persons  .Where(p => p.Age >= startAge && p.Age <= endAge)  .OrderBy(p => p.Age).ThenBy(p => p.Email);  }  public IEnumerable<Person> FindPersons(  int startAge, int endAge, string town)  {  return this.persons  .Where(p => p.Town == town)  .Where(p => p.Age >= startAge && p.Age <= endAge)  .OrderBy(p => p.Age).ThenBy(p => p.Email);  }  } |

## Run the Unit Tests

TODO

## Implement More Efficient Underlying Data Structures

Now let’s implement an improved solution, which uses **more efficient underlying data structures**.

TODO

Sample code:

|  |
| --- |
| using System.Collections.Generic;  using Wintellect.PowerCollections;  public class PersonCollection : IPersonCollection  {  private Dictionary<string, Person> personsByEmail = new Dictionary<string, Person>();  private Dictionary<string, SortedSet<Person>> personsByEmailDomain = new Dictionary<string, SortedSet<Person>>();  private Dictionary<string, SortedSet<Person>> personsByNameAndTown = new Dictionary<string, SortedSet<Person>>();  private OrderedDictionary<int, SortedSet<Person>> personsByAge = new OrderedDictionary<int, SortedSet<Person>>();  private Dictionary<string, OrderedDictionary<int, SortedSet<Person>>> personsByAgeAndTown = new Dictionary<string, OrderedDictionary<int, SortedSet<Person>>>();  public bool AddPerson(string email, string name, int age, string town)  {  if (this.FindPerson(email) != null)  {  // Person already exists  return false;  }  var person = new Person()  {  Email = email,  Name = name,  Age = age,  Town = town  };  // Add by email  this.personsByEmail.Add(email, person);  // Add by email domain  var emailDomain = this.ExtractEmailDomain(email);  if (! this.personsByEmailDomain.ContainsKey(emailDomain))  {  this.personsByEmailDomain.Add(emailDomain, new SortedSet<Person>());  }  this.personsByEmailDomain[emailDomain].Add(person);  // Add by {name + town}  var nameAndTown = this.CombineNameAndTown(name, town);  if (!this.personsByNameAndTown.ContainsKey(nameAndTown))  {  this.personsByNameAndTown.Add(nameAndTown, new SortedSet<Person>());  }  this.personsByNameAndTown[nameAndTown].Add(person);  // Add by age  if (!this.personsByAge.ContainsKey(age))  {  this.personsByAge.Add(age, new SortedSet<Person>());  }  this.personsByAge[age].Add(person);  // Add by {age + town}  OrderedDictionary<int, SortedSet<Person>> personsByTown;  if (! this.personsByAgeAndTown.TryGetValue(town, out personsByTown))  {  personsByTown = new OrderedDictionary<int, SortedSet<Person>>();  personsByAgeAndTown.Add(town, personsByTown);  }  if (!personsByTown.ContainsKey(age))  {  personsByTown.Add(age, new SortedSet<Person>());  }  personsByTown[age].Add(person);  return true;  }  private string ExtractEmailDomain(string email)  {  var domain = email.Split('@')[1];  return domain;  }  private string CombineNameAndTown(string name, string town)  {  const string separator = "|!|";  return name + separator + town;  }  public int Count  {  get { return this.personsByEmail.Count; }  }  public Person FindPerson(string email)  {  Person person;  var personExists = this.personsByEmail.TryGetValue(email, out person);  return person;  }  public bool DeletePerson(string email)  {  var person = this.FindPerson(email);  if (person == null)  {  // Person does not exist  return false;  }  // Delete person from personsByEmail  var personDeleted = this.personsByEmail.Remove(email);  // Delete person from personsByEmailDomain  var emailDomain = this.ExtractEmailDomain(email);  this.personsByEmailDomain[emailDomain].Remove(person);  // Delete person by personsByNameAndTown  var nameAndTown = this.CombineNameAndTown(person.Name, person.Town);  this.personsByNameAndTown[nameAndTown].Remove(person);  // Delete person by personsByAge  this.personsByAge[person.Age].Remove(person);  // Add person from personsByAgeAndTown  personsByAgeAndTown[person.Town][person.Age].Remove(person);  return true;  }  public IEnumerable<Person> FindPersons(string emailDomain)  {  if (this.personsByEmailDomain.ContainsKey(emailDomain))  {  return this.personsByEmailDomain[emailDomain];  }  return new Person[0];  }  public IEnumerable<Person> FindPersons(string name, string town)  {  var nameAndTown = this.CombineNameAndTown(name, town);  if (this.personsByNameAndTown.ContainsKey(nameAndTown))  {  return this.personsByNameAndTown[nameAndTown];  }  return new Person[0];  }  public IEnumerable<Person> FindPersons(int startAge, int endAge)  {  var personsInRange =  this.personsByAge.Range(startAge, true, endAge, true);  foreach (var persons in personsInRange)  {  foreach (var person in persons.Value)  {  yield return person;  }  }  }  public IEnumerable<Person> FindPersons(  int startAge, int endAge, string town)  {  if (! this.personsByAgeAndTown.ContainsKey(town))  {  yield break;  }  var personsInRange =  this.personsByAgeAndTown[town]  .Range(startAge, true, endAge, true);  foreach (var persons in personsInRange)  {  foreach (var person in persons.Value)  {  yield return person;  }  }  }  } |

## Run the Unit Tests Again

Run the unit tests again …

TODO

**Congratulations!** You have implemented efficiently the “persons collection” data structure.