Programming Project 2

Due date: 09:00pm Tuesday, October 2, 2018

General Instructions

1. Make sure that Microsoft Visual Studio 2015 is installed on your machine. Community version can be found on the SIT221 course web-page in CloudDeakin in Resources → Course materials → Visual Studio Community 2015 Installation Package. Your submission will be tested in this environment later by the markers.

   If you use Mac, you may install Visual Studio for Mac from https://e5.onthehub.com/WebStore/ProductsByMajorVersionList.aspx?ws=a8de52ee-a68b-e011-969d-0030487d8897&vsro=8, or use Xamarin Studio.

   If you are an on-campus student, this is your responsibility to check that your program is compilable and runs perfectly not just on your personal machine. You may always test your program in a classroom prior to submission to make sure that everything works properly in the MS Visual Studio 2015 environment. Markers generally have no obligations to work extra to make your program runnable; hence verify your code carefully before submission.

2. Download the zip file named as "Programming Project 2 (template)" attached to this task. It contains a template for the project that you need to work on. Open the solution file in Microsoft Visual Studio. There are two connected projects inside: DataStructures_Algorithms and Runner. At this moment, there exists only one file in Project02 subfolder of DataStructures_Algorithms that contains an interface to be used in task 3.1. You will populate this directory with C# source code files introducing new classes and interfaces. There are no constraints on how to design and implement the new modules internally, but their exterior must meet all the requirements of specific tasks in terms of functionality accessible by a user.

   There are a number of files in the Runner project containing testing routines for different tasks that you must tackle within the scope of this project. All the classes presented in the Runner project implement the IRunner interface and act as various main methods by means of their Run(string[] args) function. Make sure that you set the right runner class in Program.cs. You may modify each of the classes as you need as their main purpose is to test your code. However, they all have already a good structure to check the methods and classes required by the instruction. Therefore, you should first explore the content of the classes and see whether they satisfy your goals in terms of testing. Some lines are commented out to make the current version compilable, so assure that you make them active as you progress with the coding part for different tasks.

3. Through these tasks, you will have to read some chapters of the SIT221 course book "Data structures and algorithms in Java" (2014) by M. Goodrich, R. Tamassia, and M. Goldwasser. If you do not have your own book, you may access the book on-line for free from the reading list application in CloudDeakin available in Resources → Course materials → Course Book: Data structures and algorithms in Java.

4. You may also need to read Chapter 11 of SIT232 Workbook available in Resources → Course materials → SIT232 Workbook to remind / study Generics and their application in object-oriented programming. You may also need to read Chapter 6 of SIT232 Workbook about Polymorphism and Interfaces as you need excellent understanding of these topics to progress well through the practical tasks of the course. Make sure you are proficient with them as they form a basis to design and develop programming modules in this and all the subsequent tasks. You may find other important topics required to complete the task, like exceptions handling, in other chapters of the workbook.

Objectives

1. Study implementation and application of the Binary Heap data structure
2. Prepare library documentation for the reusable data structure.

Specification

This assignment is structured in such way that all the subsequent parts rely on the outcome of its first task. Thus, you first must succeed with implementation and testing of the Minimum Binary Heap, one of the core data structures realizing the principles of a priority queue. Before proceeding with the rest of the tasks, make sure that all the necessary operations of the binary heap produce correct results and throw specified exceptions. The later tasks will ask you either to comment the developed public functions of the corresponding class, or to apply the data structure to solve a number of important real-life problems. The first application of the data structure deals with sorting of generic objects and leads to the sorting algorithm known as Heap-Sort. In the second application, the binary heap is adopted as part of Dijkstra's algorithm to solve the Single-Source Shortest Path Problem. As the data structure enables retrieval of the minimum key element from an array-based collection in logarithmic time, it significantly improves the algorithm's runtime complexity comparing to the application of an ordinary list structure.

Task 1. Implementation of the Binary Heap data structure

At this stage, you are to be already familiar with the concept of the Minimum Binary Heap which has been discussed during the lectures. A binary heap can be considered as a special case of a binary tree, which nodes are seen as building blocks of the data structure. The important fact is that the binary heap is a complete binary tree, which satisfies the heap ordering property. Due to this fact, a binary heap with n nodes always has an $O(\log n)$ height. In case of the "min-heap" property, the key of each node must be greater than or equal to the key of its parent, with the minimum key element at the root.

Because the nodes of a binary heap entirely determine its internal structure, a user is not allowed to manipulate them, and especially their order, directly. Therefore, there should be an interface representing the exterior of the Node class while hiding the details of its implementation. This is to be your starting point, where the first step is to develop the **public**

IHeapifyable<D,K> interface limiting the methods accessible by the user to those that are stated in the following contract.

Data	Property. Gets or sets the data of generic type D associated with a particular node presented in a binary heap.
Key	Property. Gets the key of generic type K assigned to a particular node presented in a binary heap.
Position	Property. Gets the position of a particular node presented in a binary heap. The position is equal to the index of the corresponding element placed into the array-based collection of elements internal to the binary heap. The position is an integer number greater than or equal to 1.

Complete the interface, which primal purpose is to record and retrieve the data affiliated with a particular node. It further allows to read the key value and track the position of the node within the internal binary heap array. Note that this interface is parametrized by two generic data types: Type D stands for the data stored by the node and K determines the type of the key. Thus, the key may have arbitrary representation, for example, be a string or long integer number.

You can now develop the internal (**private**) Node class as part of the MinHeap<D,K> generic **public** class representing the minimum binary heap. This means that the Node class must be incorporated into the MinHeap<D,K> yet must implement the IHeapifyable<D,K> interface. With regard to other implementation aspects of the Node class, you may introduce any variables, properties and methods that you find necessary to complete the task and fulfil the requirements of the subsequent MinHeap<D,K>.

Finally, complete the MinHeap<D,K> class meeting the following requirements. Both the IHeapifyable<D,K> and the MinHeap<D,K> must be placed into Project02 subfolder of the project.

MinHeap(IComparer<IHeapifyable<D,K>> comparer)	Initializes a new instance of the MinHeap<D,K> class and records the specified reference to the object that enables comparison of two nodes, both implementing the <IHeapifyable<D,K> interface. The newly created instance of the MinHeap<D,K> class must be empty. If the MinHeap<D,K> is empty, the Count property is set to 0. This constructor is an $O(1)$ operation.
Count	Property. Gets the number of elements contained in the MinHeap<D,K>. Retrieving the value of this property is an $O(1)$ operation.
IHeapifyable<D,K> Insert(K key, D value)	Adds a new node containing the specified key-value pair to the MinHeap<D,K>. The position of the new element in the binary heap is determined according the minimum binary heap policy. Comparison of existing (and the newly added one) elements is performed by the comparator originally set within the constructor of the MinHeap<D,K>. Returns the reference of type IHeapifyable<D,K> pointing to the newly added node. This method is an $O(\log n)$ operation, where n is Count.

IHeapifyable<D,K> Min()	Returns the element yielding the minimum key, i.e. one that is positioned at the top of the current MinHeap<D,K>, without removing it. This method is similar to the DeleteMin() method, but Min() does not modify the MinHeap<D,K>. The method throws InvalidOperationException if the MinHeap<D,K> is empty. The resulting element is casted to the IHeapifyable<D,K> interface type. This method is an $O(1)$ operation.
IHeapifyable<D,K> DeleteMin()	Removes and returns the element yielding the minimum key, i.e. one that is positioned at the top of the current MinHeap<D,K>. This method throws InvalidOperationException if the MinHeap<D,K> is empty. The method is similar to the Min() method, but Min() does not modify the MinHeap<D,K>. The resulting node is casted to the IHeapifyable<D,K> interface type. This method is an $O(\log n)$ operation.
void Clear()	Removes all nodes from the current MinHeap<D,K>. Count is set to zero. This method is an $O(n)$ operation.
bool IsEmpty()	Determines whether the current MinHeap<D,K> is empty. It returns *true* if the minimum binary heap does not contain any element; otherwise, *false*. This method is an $O(1)$ operation.
IHeapifyable<D,K>[] BuildHeap(K[] keys, D[] data)	Builds a minimum binary heap using the specified keys and data according to the bottom-up approach. Every new element to be attached to the MinHeap<D,K> is defined by the key-value pair (keys[i],data[i]). To build a heap, the related MinHeap<D,K> must have no elements, so this method throws Invalid-OperationException if the MinHeap<D,K> is not empty. The method returns a collection of inserted elements casted to the IHeapifyable<D,K> interface type. This method is an $O(n)$ operation, where n is the number of input data elements.
void DecreaseKey(IHeapifyable<D,K> element, K new_key)	Decreases the key of the specified element presented in the current MinHeap<D,K>. The method throws InvalidOperation-Exception if the element stored in the MinHeap<D,K> in the position specified by the element given as an argument is different to that node. This method is an $O(\log n)$ operation.
string ToString()	Returns a string that represents the current MinHeap<D,K>. ToString() is the major formatting method in the .NET Framework. It converts an object to its string representation so that it is suitable for display.

Note that any operations on the binary heap are only available via the prescribed public methods and properties of the MinHeap<D,K>, which is responsible for their correctness as well as communication with the underlying array-based structure. The internal structure of the heap can be explored only implicitly through the positions of the nodes constituting it. Indeed, one may still need to know the position of a node to carry out such necessary operations as DecreaseKey(). Because of this, the Node class (and the IHeapifyable<D,K> interface) provides Position as a read-only property. In this assignment, you should utilize the List<T>, which is a built-in .NET Framework class, as the internal data structure of the heap rather than the raw array. In fact, you will have to change the structure of the array dynamically, so List<T> should simplify your task.

The MinHeap<D,K> relies on the internal comparator, which must implement the

IComparer< IHeapifyable<D,K> >

interface to make comparison of two nodes (remember that the Node class implements the IHeapifyable<D,K>) possible. The reference to a suitable comparator is actually passed to the constructor and should be kept internally by the MinHeap<D,K>. When the specified reference is null, the constructor should link the corresponding reference to the default comparator, which can be defined as an internal class of the following form.

```
private class DefaultComparer : IComparer<IHeapifyable<D, K>>
{
	public int Compare(IHeapifyable<D, K> x, IHeapifyable<D, K> y)
	{
		return Comparer<K>.Default.Compare(x.Key, y.Key);
	}
}
```

In fact, this default comparator delegates the comparison process to the default comparator for the generic type K that is affiliated with the key values. Because of this, the declaration of MinHeap<D,K> must impose the following constraint on type K:

```
public class MinHeap<D, K> where K : IComparable<K>
```

The DefaultComparer class is given to you as an example and you may immediately include it into the MinHeap<D,K> class. Indeed, you likely have to write a similar code for the subsequent Heap-Sort algorithm in order to adopt the minimum binary heap as a baseline structure to sort generic objects. In this sense, you must find out a way to adopt a comparator capable to compare objects of type K to build a new comparator to compare the nodes implementing the IHeapifyable<D,K>. Therefore, the provided example serves as a hint.

In conclusion, check the MinHeap_Test class in the Runner project. This class has already an appropriate structure to test the required methods of the MinHeap<D,K> class. Furthermore, Chapter 9.4 of the SIT221 course book “Data structures and algorithms in Java” and the Section 9.2.3 of Chapter 9 of the SIT221 Workbook along with the material of the lecture notes will assist you with the theory part and implementation issues of binary heaps.

Task 2. Preparing library documentation for the binary heap class.

Access and study the material available at

https://docs.microsoft.com/en-us/dotnet/csharp/codedoc

Provide annotation for the following properties and methods of the MinHeap<D,K> class:

- MinHeap (the constructor)
- Count
- Insert
- Min
- DeleteMin
- BuildHeap
- DecreaseKey

You must follow the general instructions explained in the article and cover all relevant aspects of the code using the system of XML tags. Your annotation should be concise and adhere to the style guidelines that you have seen previously for the built-in libraries of the .NET Framework. Make sure that where required you do apply multiple tags, for example <summary>, <returns>, <exception>, <param>, <typeparam> and etc. The use of particular tags is dictated by the specificity of a method.

Task 3. Applications of the minimum binary heap

Part 3.1 Implement the Heap-Sort algorithm as a sorting approach for a collection of generic type objects. Use an instance of the MinHeap<D,K> class as a core mechanism producing the next smallest element during the solution construction process. Remember that the algorithm consists of two parts: the binary heap construction and the iterative extraction of the top most elements. The corresponding sorting function must be encapsulated within a new class named "HeapSort" and implement the already provided ISorter interface. This will require you to give specific implementation for the expected

```
public void Sort<T>(T[] sequence, IComparer<T> comparer) where T : IComparable<T>
```

method of the interface. The class must have only a default constructor. You are allowed to add any extra private methods and attributes if necessary. The new class should be placed into the Project02 subfolder of the attached .NET framework project.

The coding part for this task should rely on the theoretical insights that have been previously discussed during the lectures and are presented in the corresponding lecture notes. You may further read Chapter 9.4 of the SIT221 course book "Data structures and algorithms in Java" to strengthen your understanding, but the version discussed there assumes in-place sorting which is likely impossible in case of this task as you are to address the binary heap externally. Therefore, your algorithm will probably use an auxiliary array to store intermediate results of sorting, thus require extra $O(n)$ space.

HINT: You will likely need to create a private class within the HeapSort class with the goal to convert the **IComparer<T> comparer** provided as an argument to the Sort method into another (new) one that deals with comparison of two IHeapifyable<D,K> elements. This is required since the MinHeap<D,K> class accepts only an **IComparer<IHeapifyable<D,K>>** based comparator. You should adopt the example provided in part 1 of the assignment to cope with this problem. The Runner project contains the HeapSort_Test class designed to help you with testing the Heap-Sort algorithm.

Part 3.2 People of Horsham are seeking your help. The issue is that long time ago their town and its surrounding roads were designed based on a two-dimensional grid of square cells. This forces the citizens to walk in the town strictly in one of four directions: west, east, south, or north. You may imagine that for each pair of integers x, y there is a grid point in the town with coordinates (x, y) and you are currently stand at the grid point (x_s, y_s) as a starting point. You want to get home to the grid point (x_h, y_h). Generally, if you are at (x, y), you can do a step to one of the four neighbouring grid points: $(x + 1, y)$, $(x, y + 1)$, $(x - 1, y)$, or $(x, y - 1)$. Each step takes roughly one second. Clearly, the way to home may take a while and be quite tedious.

Fortunately, Elon Musk gifted to the citizens three of his experimental teleporting systems that are to be placed in different parts of the town. Each of them connects two different grid points. If you are at one of the endpoints, you may activate the teleport and travel to its other endpoint. Traveling by teleport takes 10 seconds. Therefore, there are two ways in which you can travel. Your first option is walking and your second option is using a teleport. The teleports are not fixed yet with regard to their positions in the town, because the citizens still try to find the best places for them analysing different scenarios.

In this problem, you are given four long type integers (i.e. of type ***long*** in C#) encoding your current coordinates and those of your destination in the following order: x_s, y_s, x_h, and y_h. Furthermore, you know the potential coordinates of the three teleporting system, each system is encoded via an array of size four of long integers. In the array, the first two numbers represent the (x_1, y_1) coordinates of the first end-point and the next two numbers determine the (x_2, y_2) coordinates of its counterpart. Remember that travelling using a teleporting system is possible in both directions, but mixing-up end-points of two different systems is not safe (and therefore impossible) as it may lead to brain duality. Your goal is to compute the shortest time in which you can reach the destination in each of the test scenarios. Note that the coordinates of the teleporting systems can differ from scenario to scenario.

The general constraints imposed on the input data are as follows:

- Traveling by a teleport is not mandatory. You may pass through its endpoint and decide not to use it.
- The values of x_s, y_s, x_h, y_h as well as the coordinates of six teleports are integers in the range between 0 and 1,000,000,000, inclusive.
- There are exactly three teleporting systems.
- All eight points (your location, the location of your home, and the six teleport endpoints) will be pairwise distinct.

Watch out for integer overflows as some paths may be very long. Thus, employ the long integer type rather than the type **int** that you have used to.

Consider the following examples:

1) For input data
   3, 3, 4, 5, [1000, 1001, 1000, 1002], [1000, 1003, 1000, 1004], [1000, 1005, 1000, 1006]
   the solution is 3. You must do at least 3 steps. For example, from (3,3) to (3,4), then to (3,5), and finally to (4,5). The teleports are all too far away to be useful.
2) For input data
   0, 0, 20, 20, [1, 1, 18, 20], [1000, 1003, 1000, 1004], [1000, 1005, 1000, 1006]
   the solution is 14. The journey can be done in 40 steps (thus 40 seconds), but there is a better solution: Make 2 steps to get from (0,0) to (1,1), use the teleport to get to (18,20), and make 2 steps to get to (20,20). This solution takes (2+10+2) = 14 seconds.
3) For input data
   0, 0, 20, 20, [1000, 1003, 1000, 1004], [18, 20, 1, 1], [1000, 1005, 1000, 1006]
   the solution is 14. The teleports may be used in either direction, and in any order.
4) For input data

10, 10, 10000, 20000, [1000, 1003, 1000, 1004], [3, 3, 10004, 20002], [1000, 1005, 1000 1006]
the solution is 30.

5) For input data
3, 7, 10000, 30000, [3, 10, 5200, 4900], [12212, 8699, 9999, 30011], [12200, 8701, 5203, 4845]
the solution is 117. Sometimes the best solution requires us to use more than one teleport. In this case, the optimal solution looks as follows:
   - Walk to (3,10).
   - Use the first teleport.
   - Walk from (5200,4900) to (5203,4845).
   - Use the third teleport.
   - Walk from (12200,8701) to (12212,8699).
   - Use the second teleport.
   - Walk from (9999,30011) to (10000,30000).
6) For input data
0, 0, 1000000000, 1000000000, [0, 1, 0, 999999999], [1, 1000000000, 999999999, 0], [1000000000, 1, 1000000000, 999999999]
the solution is 36.

To prepare your code-based solution, create a new **public** class named "Teleportation" in Project02 subfolder of the project. You must write a **public** function

```
long Solve( long x_me, long y_me, long x_home, long y_home,
            long[] teleport1, long[] teleport2, long[] teleport3 )
```

which, when given your current coordinates (xMe, yMe) and those of your destination (xHome, yHome) along with the coordinates of the three teleporting systems (each presented as an array of four numbers), returns a long integer that tells the shortest time to achieve the destination starting at the current position.

Second, develop an algorithm to solve the problem and make sure you pass all the subsequent test instances in reasonable time. At this moment, you may already know Dijkstra's algorithm as one of the shortest path algorithms. This algorithm can adopt a minimum binary heap as a data structure used to select the next node to visit within its solution construction process. This task has a particular guideline on algorithmic technique to be applied. You must implement Dijkstra's algorithm that involves the minimum binary heap data structure that you have already developed in task 1. However, you may implement your algorithm as you wish and introduce any private methods and variables unless the prescribed ***Solve(…)*** method returns correct answer to the problem.

Finally, check the Project02 subfolder in Data directory of the Runner project. The class ***Teleportation_Generator*** there was written for you to provide a benchmark suite of test instances. Its static ***Count()*** method returns the total number of instances that it is able to create. The public **static** method

```
long Generate( int k,
               out long x_me, out long y_me,
```

```
                    out long x_home, out long y_home,
                    out long[] teleport1, out long[] teleport2, out long[] teleport3 )
```

of this class produces instance k (this value is taken as an input argument) and returns the coordinates of the current position and destination as well as the arrays containing the coordinates of the teleporting systems. The names of the variables are consistent to those given as input arguments to the Solve() method of the Teleportation class. Its return value provides a long integer giving the correct answer for instance k. Therefore, you may extract data calling the **Generate()** method on the class **Teleportation _Generator** as it is already done in **Teleportation_Test** class in the Runner project.

As a reference material, explore Section 14.6.2 of the SIT221 course book "Data structures and algorithms in Java".

HINT: You may implement Dijkstra's algorithm as a private static method of the Teleportation class. Its signature can be as follows:

```
static private long ShortestPath(long[,] distances, int start_ind, int end_ind)
```

Here, the specified distances in the form of a two-dimensional array represent the weighted adjacency matrix. The algorithm should then involve the minimum binary heap and you need to think about what the keys and values should represent with regard to the shortest path problem in general, and in terms of Dijkstra's algorithm in particular.

Submission instructions

Submit your program code via the CloudDeakin System by the deadline. The normal Deakin University policy is applied for late submissions. You **must zip** your IHeapifyable.cs, MinHeap.cs, HeapSort.cs, and Teleportation.cs files only and name the resulting zip file as Project2.zip. Please, make sure that the file names are correct. You must not submit the whole MS Visual Studio project / solution files.

This is your responsibility to keep file names correct as some of them will be processed automatically by scripts. If they are in a wrong format, the scripts will fail to mark them and you may not get marks.

Marking scheme

You may get the following scores for this project:

- In part 1, in order to get at least partial marks, the constructor along with Insert() and DeleteMin() methods of the MinHeap class must produce correct results and return expected values. If they fail, we are not able to test the class and give you any marks. Your MinHeap class **must be compilable to deserve any marks**. The Insert(), DeleteMin(), BuildHeap(), and DecreaseKey() methods, when are performing as prescribed, are worth of 5 scores each. The correct implementation and work of the Count property and Min(), Clear(), IsEmpty() methods yield 1 score each. In total, part 1 can give you 24 cores.

- In part 2, you can get 10 mark for outstanding work covering all the aspects of the specified methods, 5 scores for reasonable effort when the documentation lacks of some desired tags or details. Otherwise, zero scores will be granted for unsatisfactory work.
- In part 3.1, you can get either 6 scores if your HeapSort class **runs properly and sorts generic data correctly**, or 0 scores if it **does not work**, **causes compilation or runtime errors**, **has no code commenting** (markers will use this to verify your algorithm), or **does not sort generic data as prescribed**.
- In parts 3.2, you will get s**cores proportional to the number of passed test instances**. You are given only 50% of instances we have, so the rest 50% are kept as a secret and will be used (and released after the deadline) to verify your algorithms automatically. Thus, part 3.2 has 108 tests in total and your result will be a fraction scaled to 18 scores (the maximum). Note that we will have a look at your code to avoid cheating; that is, the case when your algorithm just simply returns the expected answers because you know them. We do not treat this as a real solution and you will get zero for it.

In the best case, you may get up to 58 scores. Your scores will be capped by 50. So, you may potentially skip some tasks. However, we strongly encourage you to attempt all the exercises as (i) this content is examinable and you may get similar questions on your final exam, and (ii) you never know where you may lose some marks on other questions. Your marks will be finally scaled to 15% of the final grade for this unit.

Good luck!