Practical Task 6.2

(Credit Task)

Submission deadline: Monday, May 8
Discussion deadline: Friday, May 26

General Instructions

The objective of this task is to implement the Binary Heap, a data structure which has the structure of the complete binary tree. Like a binary tree, a heap consists of a collection of nodes that can be considered as building blocks of the data structure. The tree structure that a binary heap represents is complete; that is, every level, except possibly the last one, is entirely filled, and all nodes are as far left as possible. This makes a binary heap with n nodes be always of a $O(\log n)$ height. In addition to the standard properties of a binary tree, a binary heap must also adhere to the mandatory **Heap Ordering** property. The ordering can be one of the two types:

- The *Min-Heap Property*: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.
- The *Max-Heap Property*: the value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.

The binary heap is not a sorted structure; that is, there is no relationship among nodes on any given level, even among siblings. It is a very useful data structure when one needs to remove the object with the lowest or highest (in case of the max-heap ordering) priority.

The binary heap can be uniquely represented by storing its level order traversal in an array or an array-based collection, e.g., a list (also known as a vector). Note that the links between nodes are not required. To simplify the implementation, the first entry of the array with index 0 is skipped; it contains a dummy (default) element. Therefore, the root of a heap is the second item in the array at index 1, and the length of the array is $n+1$ for a heap with n data elements. This implies that for the k^{th} element of the array the following statements are valid:

- the left child is located at index $2 \cdot k$;
- the right child is located at index $2 \cdot k + 1$;
- the parent is located uniquely at index $\lfloor k/2 \rfloor$.

Insertion of a new element initially appends it to the end of a heap as the last element of the array at index $n+1$. The *Heap Ordering* property is then restored by comparing the added element with its parent and moving the added element up a level (swapping positions with the parent). This process is commonly known as “UpHeap”, or “Heapify-Up”, or “Sift-Up”. The comparison is repeated until the parent is larger (or smaller, in case of the max-heap ordering) than or equal to the percolating element. The worst-case runtime of this operation is $O(\log n)$ since we need at most one swap on each level of a heap on the path from the inserted node to the root.

The *minimum* (or *maximum*, in case of the max-heap ordering) element can be found at the root, which is the element of the array located at index 1. *Deletion* of the *minimum* element first replaces it with the last element of the array at index n, and then restores the *Heap Ordering* property by following the process known as “DownHeap”, or “Heapify-Down”, or “Sift-Down”. Like insertion, the worst-case runtime of this operation is $O(\log n)$.

The following steps indicate what you need to do.

1. Download the three C# source code files attached to this task. These files serve as a template for the program that you need to develop. Create a new C# project and import the files. Your newly built project should compile and work without errors, although it will initially fail most of provided test cases.

- **Heap.cs** file contains the partially completed Heap<K,D> class that you will need to finish by adding remaining methods. Therefore, explore this code thoroughly as the missing methods may have implementation similar to some existing methods.
- **Tester.cs** file is where you will find the Main method to be used as the starting point of your program. It also contains a series of tests that will verify if your Vector class works correctly. When first running your program, these tests will fail. Once you have completed the task, all these tests will report success. See Appendix A for an example of what the program will output when working correctly.

2. Find the nested Node class inside the Heap<K,D> class and explore its structure. This is a generic class that represents a node of a binary heap. Consider it as a building block of the Heap<K,D>. It consists of

   - a generic type raw **data** (a *payload*),
   - a generic type **key** necessary to position the node in regard to the order of nodes present in the Heap<K,D>, and
   - an integer-valued **position** (index) that locates the node in the array-based collection of nodes of the Heap<K,D>.

   Because both K and D types are generic, the **key** and the **data** may be of an arbitrary type: a string, an integer, or a user-defined class. Finally, note that the Node class is ready for you to use. It provides the following functionality:

   - **Node(K key, D value, int position)**

     Constructor. Initializes a new instance of the Node class associated with the specified generic key. The node records the given generic data as well as its own index-based position within the array-based collection of data nodes privately owned by the related Heap<K,D>.

   - **D Data**

     Property. Gets or sets the data of generic type D associated with the Node.

   - **K Key**

     Property. Gets or sets the key of generic type K assigned to the Node.

   - **int Position**

     Property. Gets or sets the index-based position of the Node in the array-based collection of nodes constituting the Heap<K,D> .

   - **string ToString( )**

     Returns a string that represents the current Node.

   The Node class implements the IHeapifyable<K,D> interface, which is defined in the attached IHeapifyable.cs file. Note that this interface is parametrized by the same two generic data types as the Heap<K,D>. The reason for the use of the interface is that the Node class is a data structure internal to the Heap<K,D>, therefore an instance of the Node must not be exposed to the user. It must remain hidden for the user in order to protect the integrity of the whole data structure. Otherwise, manipulating the nodes directly, the user may easily corrupt the structure of the binary heap and violate the important *Heap-Ordering* rule. Nevertheless, because the user needs access to the data that the user owns and stores inside the binary heap, the Node implements the interface that permits reading and modifying the data. Therefore, the primal purpose of the IHeapifyable<K,D> is to record and retrieve the data associated with a particular node and track the position of the node in the array-based collection of nodes of the Heap<K,D>.

   Check the IHeapifyable<K,D> interface to see that the only property it allows to change is the **Data**. The other two properties, i.e., the **Key** and the **Position**, are read-only. Note that the value of a key is set at the time the node is added to a heap. It then can be changed only via dedicated operations, like the *DecreaseKey*. The Heap<K,D> is entirely responsible for the **Position**, thus modification of this property by the user is impossible.

3. Proceed with the given template of the Heap<K,D> class and explore the methods that make an example for you, in particular:

- **Heap(IComparer<K> comparer)**

  Constructor. Initializes a new instance of the Heap<K,D> class and records the specified **comparer** that enables comparison of two keys of type K. The given **comparer** defines whether it is a min or max binary heap.

- **int Count**

  Property. Gets the number of elements stored in the Heap<K,D>.

- **IHeapifyable<K, D> Min( )**

  Returns the element with the minimum (or maximum) key positioned at the top of the Heap<K,D>, without removing it. The element is cast into the IHeapifyable<K,D> interface. The method throws InvalidOperationException if the Heap<K,D> is empty.

- **IHeapifyable<K, D> Insert(K key, D value)**

  Inserts a new node containing the specified key-value pair into the Heap<K,D>. The position of the new element in the binary heap is determined according to the Heap-Order policy resulting from the encapsulated **comparer** object. It returns the newly created node cast into the IHeapifyable<K,D> interface.

- **void Clear( )**

  Removes all nodes from the Heap<K,D> and sets the Count to zero.

- **string ToString( )**

  Returns a string representation of the Heap<K,D>.

Rather than an array, the Heap<K,D> utilizes the native .NET Framework List<T> generic collection as the internal data structure. This collection is dynamic as opposed to the array, which is static. This fact should simplify your work. Furthermore, note that the internal structure of the Heap<K,D> can be explored only implicitly through the positions of the nodes constituting it.

Keep in mind that the comparison of nodes is performed by the **comparator** originally set within the constructor of the Heap<K,D>. Providing different comparator to the constructor will change the behaviour of the Heap<K,D>. When keys are ordered in ascending order, the Heap<K,D> acts as a min-heap. When the **comparator** orders keys in descending order, the Heap<K,D> behaves as a max-heap.

4. You must complete the Heap<K,D> class and provide the following functionality to the user:

   - **IHeapifyable<K, D> Delete()**

     Deletes and returns the node cast to the IHeapifyable<K,D> positioned at the top of the Heap<K,D>. This method throws InvalidOperationException if the Heap<K,D> is empty.

   - **IHeapifyable<K, D>[] BuildHeap(K[] keys, D[] data)**

     Builds a binary heap following the bottom-up approach[1]. Each new element of the heap is derived by the key-value pair (keys[i],data[i]) specified by the method's parameters. It returns an array of nodes casted to the IHeapifyable<K,D>. Each node at index i must match its key-value pair at index i of the two input arrays. This method throws InvalidOperationException if the Heap<K,D> is not empty.

   - **void DecreaseKey(IHeapifyable<K, D> element, K new_key)**

     Decreases the key of the specified element presented in the Heap<K,D>. The method throws InvalidOperationException when the node stored in the Heap<K,D> at the position specified by the element is different to the element. This signals that the given element is inconsistent to the current state of the Heap<K,D>.

   Note that you are free in writing your code that is private to the Heap<K,D> unless you respect all the requirements in terms of functionality and signatures of the specified methods.

5. As you progress with the implementation of the Heap<K,D> class, you should start using the Tester class to thoroughly test the Heap<K,D> aiming on the coverage of all potential logical issues and runtime errors. This (testing) part of the task is as important as writing the Heap<K,D> class. The given version of the testing class covers only some basic cases. Therefore, you should extend it with extra cases to make sure that your data structure is checked against other potential mistakes.

[1] See the explanation of the algorithm in Lecture 6.

Further Notes

– The lecture notes of week 6 should be good to understand the logic behind the binary heap and its array-based implementation.
– Explore the material of chapter 9.4 of the SIT221 course book "Data structures and algorithms in Java" (2014) by Michael T. Goodrich, Irvine Roberto Tamassia, and Michael H. Goldwasser (2014). You may access the book on-line from the reading list application in CloudDeakin available in Content → Reading List → Course Book: Data structures and algorithms in Java.
– As a supplementary material, to learn more about the complexity and implementation issues of binary heaps, you may refer to the Section 9.2.3 of Chapter 9 of SIT221 Workbook available in CloudDeakin available in Content → Learning Resources → SIT221 Workbook.

Submission Instructions and Marking Process

To get your task completed, you must finish the following steps strictly on time.

– Make sure your programs implement the required functionality. They must compile, have no runtime errors, and pass all test cases of the task. Programs causing compilation or runtime errors will not be accepted as a solution. You need to test your programs thoroughly before submission. Think about potential errors where your programs might fail.
– **Submit** the expected code files as a solution to the task via OnTrack submission system.
– Once your solution is accepted by your tutor, you will be invited to **continue its discussion and answer relevant theoretical questions through a face-to-face interview**. Specifically, you will need to meet with the tutor to demonstrate and discuss the solution in one of the dedicated practical sessions (run online via MS Teams for online students and on-campus for students who selected to join classes at Burwood\Geelong). Please, come prepared so that the class time is used efficiently and fairly for all students in it. Be on time with respect to the specified discussion deadline.
  You will also need to **answer all additional questions** that your tutor may ask you. Questions will cover the lecture notes; so, attending (or watching) the lectures should help you with this **compulsory** discussion part. You should start the discussion as soon as possible as if your answers are wrong, you may have to pass another round, still before the deadline. Use available attempts properly.

Note that we will not accept your solution after **the submission deadline** and will not discuss it after **the discussion deadline**. If you fail one of the deadlines, you fail the task, and this reduces the chance to pass the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Remember that this is your responsibility to keep track of your progress in the unit that includes checking which tasks have been marked as completed in the OnTrack system by your marking tutor, and which are still to be finalised. When grading your achievements at the end of the unit, we will solely rely on the records of the OnTrack system and feedback provided by your tutor about your overall progress and the quality of your solutions.

Appendix A: Expected Printout

The following provides an example of the output generated from the testing module (Tester.cs) once you have correctly implemented all methods of the Heap<K,D> class.

```
Test A: Create a min-heap by calling 'minHeap = new Heap<int, string>(new IntAscendingComparer());'
 :: SUCCESS: min-heap's state []


Test B: Run a sequence of operations:
Insert a node with name Kelly (data) and ID 1 (key).
 :: SUCCESS: min-heap's state [(1,Kelly,1)]


Insert a node with name Cindy (data) and ID 6 (key).
```

```
 :: SUCCESS: min-heap's state [(1,Kelly,1),(6,Cindy,2)]


Insert a node with name John (data) and ID 5 (key).
 :: SUCCESS: min-heap's state [(1,Kelly,1),(6,Cindy,2),(5,John,3)]


Insert a node with name Andrew (data) and ID 7 (key).
 :: SUCCESS: min-heap's state [(1,Kelly,1),(6,Cindy,2),(5,John,3),(7,Andrew,4)]


Insert a node with name Richard (data) and ID 8 (key).
 :: SUCCESS: min-heap's state [(1,Kelly,1),(6,Cindy,2),(5,John,3),(7,Andrew,4),(8,Richard,5)]


Insert a node with name Michael (data) and ID 3 (key).
 :: SUCCESS: min-heap's state [(1,Kelly,1),(6,Cindy,2),(3,Michael,3),(7,Andrew,4),(8,Richard,5),(5,John,6)]


Insert a node with name Guy (data) and ID 10 (key).
 :: SUCCESS: min-heap's state [(1,Kelly,1),(6,Cindy,2),(3,Michael,3),(7,Andrew,4),(8,Richard,5),(5,John,6),(10,Guy,7)]


Insert a node with name Elicia (data) and ID 4 (key).
 :: SUCCESS: min-heap's state [(1,Kelly,1),(4,Elicia,2),(3,Michael,3),(6,Cindy,4),(8,Richard,5),(5,John,6),(10,Guy,7),(7,Andrew,8)]


Insert a node with name Tom (data) and ID 2 (key).
 :: SUCCESS: min-heap's state
[(1,Kelly,1),(2,Tom,2),(3,Michael,3),(4,Elicia,4),(8,Richard,5),(5,John,6),(10,Guy,7),(7,Andrew,8),(6,Cindy,9)]


Insert a node with name Iman (data) and ID 9 (key).
 :: SUCCESS: min-heap's state
[(1,Kelly,1),(2,Tom,2),(3,Michael,3),(4,Elicia,4),(8,Richard,5),(5,John,6),(10,Guy,7),(7,Andrew,8),(6,Cindy,9),(9,Iman,10)]


Insert a node with name Simon (data) and ID 14 (key).
 :: SUCCESS: min-heap's state
[(1,Kelly,1),(2,Tom,2),(3,Michael,3),(4,Elicia,4),(8,Richard,5),(5,John,6),(10,Guy,7),(7,Andrew,8),(6,Cindy,9),(9,Iman,10),(14,Simon,1
1)]


Insert a node with name Vicky (data) and ID 12 (key).
 :: SUCCESS: min-heap's state
[(1,Kelly,1),(2,Tom,2),(3,Michael,3),(4,Elicia,4),(8,Richard,5),(5,John,6),(10,Guy,7),(7,Andrew,8),(6,Cindy,9),(9,Iman,10),(14,Simon,1
1),(12,Vicky,12)]


Insert a node with name Kevin (data) and ID 11 (key).
 :: SUCCESS: min-heap's state
[(1,Kelly,1),(2,Tom,2),(3,Michael,3),(4,Elicia,4),(8,Richard,5),(5,John,6),(10,Guy,7),(7,Andrew,8),(6,Cindy,9),(9,Iman,10),(14,Simon,1
1),(12,Vicky,12),(11,Kevin,13)]


Insert a node with name David (data) and ID 13 (key).
 :: SUCCESS: min-heap's state
[(1,Kelly,1),(2,Tom,2),(3,Michael,3),(4,Elicia,4),(8,Richard,5),(5,John,6),(10,Guy,7),(7,Andrew,8),(6,Cindy,9),(9,Iman,10),(14,Simon,1
1),(12,Vicky,12),(11,Kevin,13),(13,David,14)]


Test C: Run a sequence of operations:
Delete the minimum element from the min-heap.
```

```
 :: SUCCESS: min-heap's state
[(2,Tom,1),(4,Elicia,2),(3,Michael,3),(6,Cindy,4),(8,Richard,5),(5,John,6),(10,Guy,7),(7,Andrew,8),(13,David,9),(9,Iman,10),(14,Simon
,11),(12,Vicky,12),(11,Kevin,13)]


Delete the minimum element from the min-heap.
 :: SUCCESS: min-heap's state
[(3,Michael,1),(4,Elicia,2),(5,John,3),(6,Cindy,4),(8,Richard,5),(11,Kevin,6),(10,Guy,7),(7,Andrew,8),(13,David,9),(9,Iman,10),(14,Sim
on,11),(12,Vicky,12)]


Delete the minimum element from the min-heap.
 :: SUCCESS: min-heap's state
[(4,Elicia,1),(6,Cindy,2),(5,John,3),(7,Andrew,4),(8,Richard,5),(11,Kevin,6),(10,Guy,7),(12,Vicky,8),(13,David,9),(9,Iman,10),(14,Simo
n,11)]


Delete the minimum element from the min-heap.
 :: SUCCESS: min-heap's state
[(5,John,1),(6,Cindy,2),(10,Guy,3),(7,Andrew,4),(8,Richard,5),(11,Kevin,6),(14,Simon,7),(12,Vicky,8),(13,David,9),(9,Iman,10)]


Delete the minimum element from the min-heap.
 :: SUCCESS: min-heap's state
[(6,Cindy,1),(7,Andrew,2),(10,Guy,3),(9,Iman,4),(8,Richard,5),(11,Kevin,6),(14,Simon,7),(12,Vicky,8),(13,David,9)]


Delete the minimum element from the min-heap.
 :: SUCCESS: min-heap's state
[(7,Andrew,1),(8,Richard,2),(10,Guy,3),(9,Iman,4),(13,David,5),(11,Kevin,6),(14,Simon,7),(12,Vicky,8)]


Delete the minimum element from the min-heap.
 :: SUCCESS: min-heap's state [(8,Richard,1),(9,Iman,2),(10,Guy,3),(12,Vicky,4),(13,David,5),(11,Kevin,6),(14,Simon,7)]


Delete the minimum element from the min-heap.
 :: SUCCESS: min-heap's state [(9,Iman,1),(12,Vicky,2),(10,Guy,3),(14,Simon,4),(13,David,5),(11,Kevin,6)]


Delete the minimum element from the min-heap.
 :: SUCCESS: min-heap's state [(10,Guy,1),(12,Vicky,2),(11,Kevin,3),(14,Simon,4),(13,David,5)]


Delete the minimum element from the min-heap.
 :: SUCCESS: min-heap's state [(11,Kevin,1),(12,Vicky,2),(13,David,3),(14,Simon,4)]


Delete the minimum element from the min-heap.
 :: SUCCESS: min-heap's state [(12,Vicky,1),(14,Simon,2),(13,David,3)]


Delete the minimum element from the min-heap.
 :: SUCCESS: min-heap's state [(13,David,1),(14,Simon,2)]


Delete the minimum element from the min-heap.
 :: SUCCESS: min-heap's state [(14,Simon,1)]


Delete the minimum element from the min-heap.
 :: SUCCESS: min-heap's state []
```

```
Test D: Delete the minimum element from the min-heap.
 :: SUCCESS: InvalidOperationException is thrown because the min-heap is empty


Test E: Run a sequence of operations:


Insert a node with name Kelly (data) and ID 1 (key).
 :: SUCCESS: min-heap's state [(1,Kelly,1)]


Build the min-heap for the pair of key-value arrays with
[1, 6, 5, 7, 8, 3, 10, 4, 2, 9, 14, 12, 11, 13] as keys and
[Kelly, Cindy, John, Andrew, Richard, Michael, Guy, Elicia, Tom, Iman, Simon, Vicky, Kevin, David] as data elements
 :: SUCCESS: InvalidOperationException is thrown because the min-heap is not empty


Test F: Run a sequence of operations:
Clear the min-heap.
 :: SUCCESS: min-heap's state []


Build the min-heap for the pair of key-value arrays with
[1, 6, 5, 7, 8, 3, 10, 4, 2, 9, 14, 12, 11, 13] as keys and
[Kelly, Cindy, John, Andrew, Richard, Michael, Guy, Elicia, Tom, Iman, Simon, Vicky, Kevin, David] as data elements
 :: SUCCESS: min-heap's state
[(1,Kelly,1),(2,Tom,2),(3,Michael,3),(4,Elicia,4),(8,Richard,5),(5,John,6),(10,Guy,7),(6,Cindy,8),(7,Andrew,9),(9,Iman,10),(14,Simon,1
1),(12,Vicky,12),(11,Kevin,13),(13,David,14)]



Test G: Run a sequence of operations:


Delete the minimum element from the min-heap.
 :: SUCCESS: min-heap's state
[(2,Tom,1),(4,Elicia,2),(3,Michael,3),(6,Cindy,4),(8,Richard,5),(5,John,6),(10,Guy,7),(13,David,8),(7,Andrew,9),(9,Iman,10),(14,Simon
,11),(12,Vicky,12),(11,Kevin,13)]


Delete the minimum element from the min-heap.
 :: SUCCESS: min-heap's state
[(3,Michael,1),(4,Elicia,2),(5,John,3),(6,Cindy,4),(8,Richard,5),(11,Kevin,6),(10,Guy,7),(13,David,8),(7,Andrew,9),(9,Iman,10),(14,Sim
on,11),(12,Vicky,12)]


Run DecreaseKey(node,0) for node (13,David,8) by setting the new value of its key to 0
 :: SUCCESS: min-heap's state
[(0,David,1),(3,Michael,2),(5,John,3),(4,Elicia,4),(8,Richard,5),(11,Kevin,6),(10,Guy,7),(6,Cindy,8),(7,Andrew,9),(9,Iman,10),(14,Simo
n,11),(12,Vicky,12)]



Test H: Run a sequence of operations:


Create a max-heap by calling 'maxHeap = new Heap<int, string>(new IntDescendingComparer());'
 :: SUCCESS: max-heap's state []


Build the max-heap for the pair of key-value arrays with
[1, 6, 5, 7, 8, 3, 10, 4, 2, 9, 14, 12, 11, 13] as keys and
```

```
[Kelly, Cindy, John, Andrew, Richard, Michael, Guy, Elicia, Tom, Iman, Simon, Vicky, Kevin, David] as data elements

 :: SUCCESS: max-heap's state
[(14,Simon,1),(9,Iman,2),(13,David,3),(7,Andrew,4),(8,Richard,5),(12,Vicky,6),(10,Guy,7),(4,Elicia,8),(2,Tom,9),(6,Cindy,10),(1,Kelly,
11),(3,Michael,12),(11,Kevin,13),(5,John,14)]




 ------------------- SUMMARY -------------------

Tests passed: ABCDEFGH
```