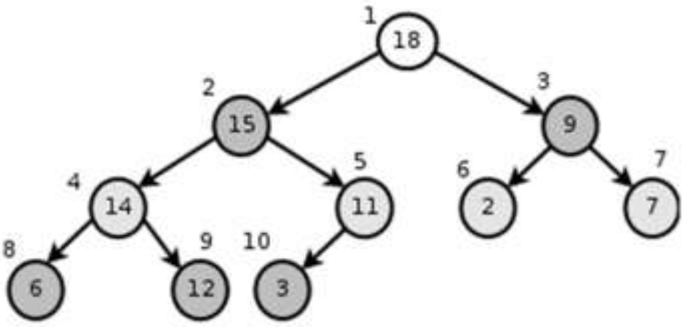
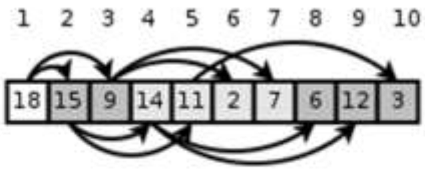
Exercises: Implement a Binary Heap

This document defines the **in-class exercises** assignments for the ["Data Structures" course @ Software University](https://softuni.bg/courses/data-structures).
You have to implement a **binary heap**:





The binary heap holds its element in an **array**. Elements are numbered with **indexes** 0 … length-1. The array represents a perfectly **balanced binary tree**. Each node **i** may have children (left and right) and parent:

- **`parent(i) = (i - 1) / 2`**
- **`leftChild(i) = 2 * i + 1`**
- **`rightChild(i) = 2 * i + 2`**

Binary heaps always hold the "***heap property***":

- Each **node** is **smaller** or equal than its **parent** node.

We should **maintain the "*heap property*"** all the time during our work, so "**heapify up**" or "**heapify down**" should apply each time after we modify the heap. See the steps below to learn how to maintain it.

Problem 1. Learn about Binary Heap in Wikipedia

Before starting, get familiar with the concept of **binary heap**: https://en.wikipedia.org/wiki/Binary_heap.

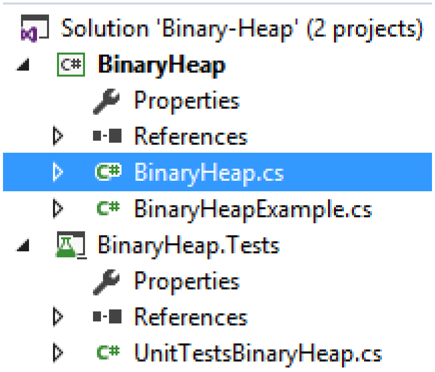
The typical **operations** over a binary heap are:

- **`Build-Max-Heap(arr)`** – builds a binary heap from array of unordered elements
- **`Heapify-Down(index)`** – apply the "*heap property*" down from given node
- **`Extract-Max()`** – extract (and remove) the max element from the heap.
- **`Insert(element)`** – inserts a new element in the heap (and maintains the "*heap property*")
- **`Heapify-Up(index)`** – apply the "*heap property*" up from given node
- **`Peek-Max()`** – finds the max element from the heap (without remove).

Let's start coding!

Problem 2. BinaryHeap<T > – Project Skeleton

You are given a **Visual Studio project skeleton** (unfinished project) holding the unfinished class **`BinaryHeap<T>`** and **unit tests** covering its functionality. The project holds the following assets:

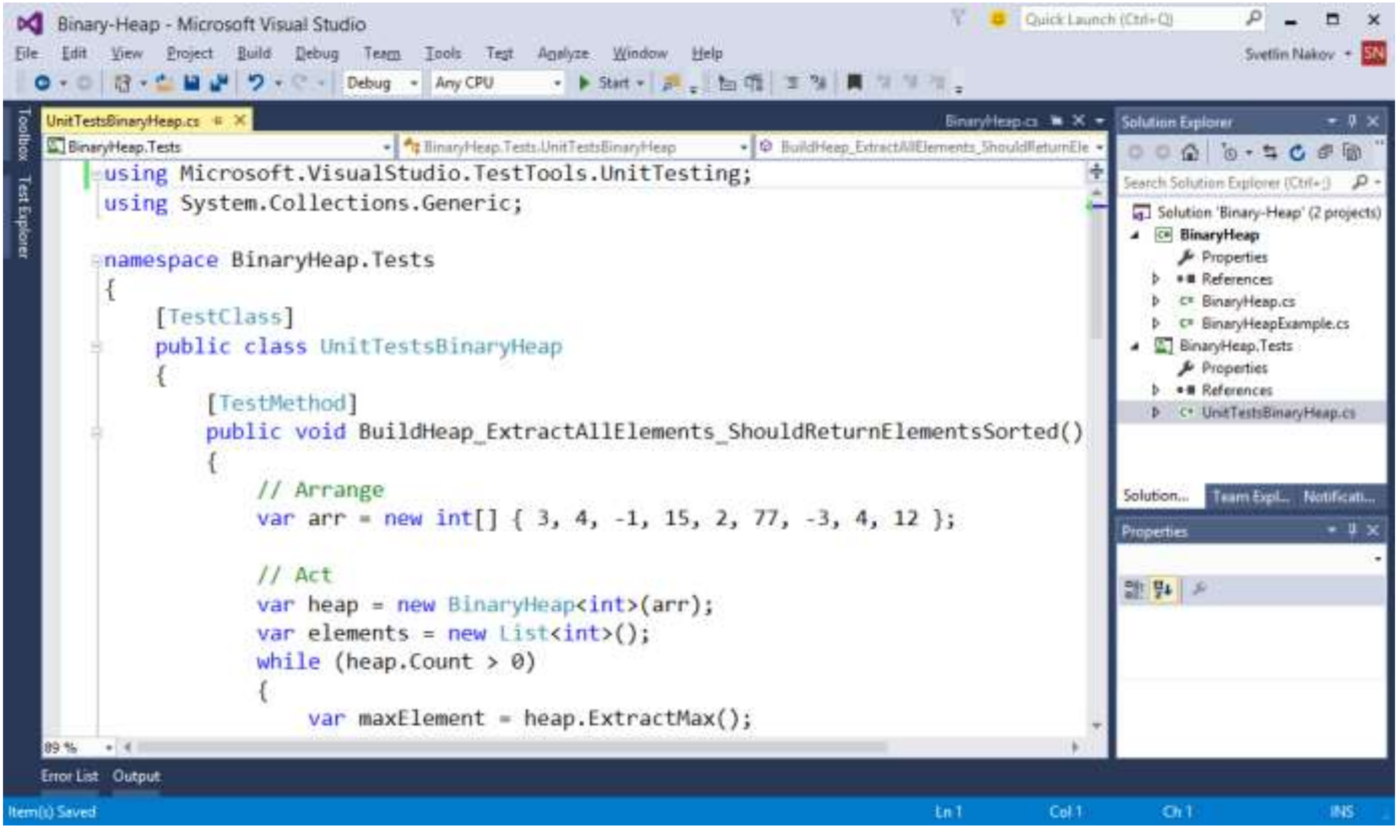


The project skeleton opens correctly in **Visual Studio 2013** but can be open in other Visual Studio versions as well and also can run in **SharpDevelop** and **Xamarin Studio**. Your goal is to implement the missing functionality in order to finish the project.

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:

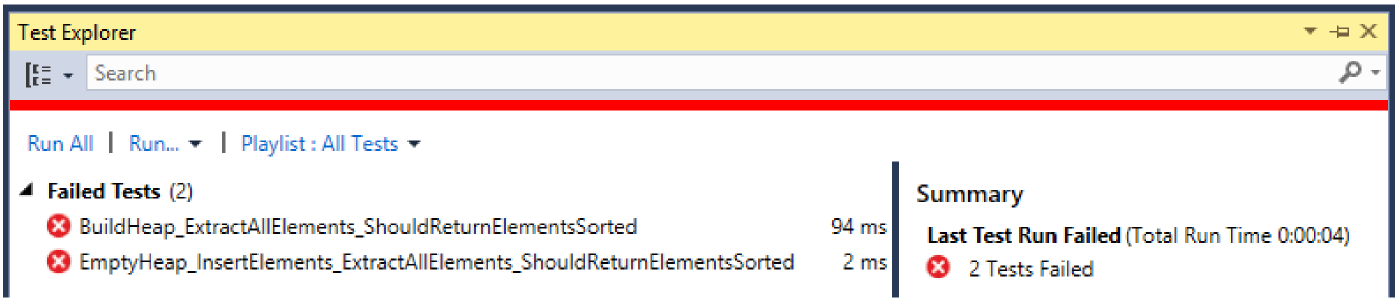
```
public class BinaryHeap<T> where T : IComparable<T>
{
    public BinaryHeap()...

    public BinaryHeap(T[] elements)...

    public int Count...

    public T ExtractMax()...

    public T PeekMax()...

    public void Insert(T node)...

    private void HeapifyDown(int i)...

    private void HeapifyUp(int i)...
}
```

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



Problem 3. Run the Unit Tests to Ensure All of Them Initially Fail

Run the unit tests from the **`BinaryHeap.Tests`** project. All of them should fail:



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

Problem 4. Define the Binary Heap Internal Data

The **internal data** holding the binary heap elements is quite simple, it is just a **list** of elements (array that can grow):

```
private List<T> heap;
```

Problem 5. Implement the Binary Heap Constructor

Now, let's implement the binary heap **constructor**. Its purpose is to allocate the internal array that will hold the binary heap elements (balanced binary tree). The binary heap constructor has two forms:

- Parameterless constructor – should allocate and empty binary heap
- Constructor with parameter **`array`** – converts existing array of elements to binary heap

The first **parameterless constructor** is quite simple:

```
public BinaryHeap()
{
    this.heap = new List<T>();
}
```

The **second constructor** is more complex:

```
public BinaryHeap(T[] elements)
{
    this.heap = new List<T>(elements);
    for (int i = this.heap.Count / 2; i >= 0; i--)
    {
        HeapifyDown(i);
    }
}
```

The above code first **allocates the internal list** to hold the binary heap elements, then **fills** the passed as argument elements in the internal list and then "***heapifies***" the elements. This means that each **element** becomes **less or equal to its parent**. This happens by moving up each element, which is bigger than its parent. See the implementation of **`HeapifyDown(index)`** method.

We implement also the **`Count`** property which it trivial and returns the number of elements in the heap:

```
public int Count
{
    get
    {
        return this.heap.Count;
    }
}
```

Problem 6. Implement HeapifyDown(index) Method

The **`HeapifyDown(index)`** method starts from given **index** and **reorders the element** from this index **down to its correct place**. The element is swapped with its biggest child element (if the "*heap property*" is not hold). This happens recursively again, and again until we reach a leaf node or the heap property is already hold. See the code below:

```
private void HeapifyDown(int i)
{
    var left = 2 * i + 1;
    var right = 2 * i + 2;
    var largest = i;
    if (left < this.heap.Count &&
        this.heap[left].CompareTo(this.heap[largest]) > 0)
    {
        largest = left;
    }
    if (right < this.heap.Count &&
        this.heap[right].CompareTo(this.heap[largest]) > 0)
    {
        largest = right;
    }
    if (largest != i)
    {
        T old = this.heap[i];
        this.heap[i] = this.heap[largest];
        this.heap[largest] = old;
        HeapifyDown(largest);
    }
}
```

Problem 7. Implement the ExtractMax() Method

Now, we are ready to implement the most important method **`ExtractMax()`** which returns and removes the maximal element:

```
public T ExtractMax()
{
    var max = this.heap[0];
    this.heap[0] = this.heap[heap.Count - 1];
    this.heap.RemoveAt(this.heap.Count - 1);
    if (this.heap.Count > 0)
    {
        HeapifyDown(0);
    }
    return max;
}
```

How it works? It works in thee steps:

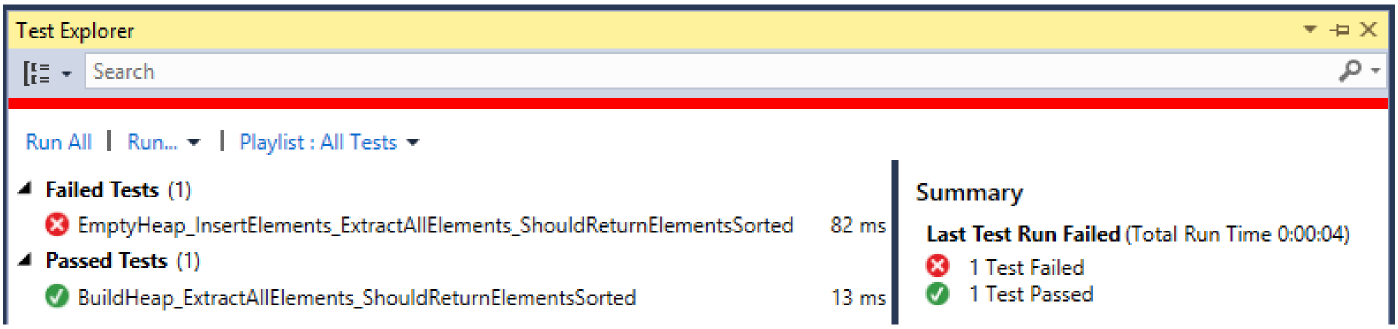
1. Takes as result the **maximal element** – the elements at **index 0** (the root node in the tree).
2. **Deletes the last element** from the internal list holding the heap elements and **moves it at position 0** (as root node).
3. Moves down the root node to **apply the "*heap property*"**, i.e. call **`Heapify-Down()`**.

We also implement the **Peek-Max** operation. It is trivial: just **return the root element** (from index 0):

```
public T PeekMax()
{
    var max = this.heap[0];
    return max;
}
```

Problem 8. Run the Unit Tests

We have **partially implemented** the binary heap. It supports **`Build-Heap`** and **`Extract-Max`** operations. Let's run the tests. We can expect some of them to pass:



To have more tests passed, we need to implement the rest of the functionality. Let's continue.

Problem 9. Implement Insert(node) Method

Let's implement **inserting a new node**. It should append the new node at the **end of the internal list** holding the binary heap elements and **pull it up** until it finds its correct place in the heap:

```
public void Insert(T node)
{
    this.heap.Add(node);
    HeapifyUp(this.heap.Count - 1);
}
```

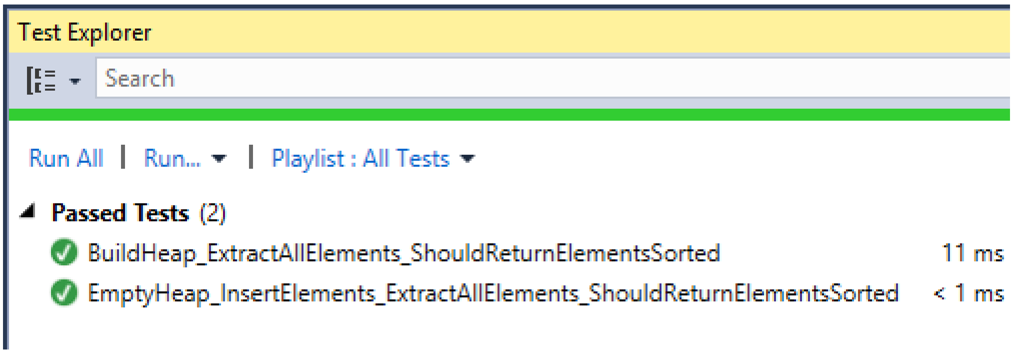
This method relies on the **`Heapify-Up`** operation. It starts from given index and **interchanges** the element at this **index** with its **parent** until the "*heap property*" becomes valid:

```
private void HeapifyUp(int i)
{
    var parent = (i - 1) / 2;
    while (i > 0 && this.heap[i].CompareTo(this.heap[parent]) > 0)
    {
        // Swap heap[i] with heap[parent]

        // Move to the parent node

    }
}
```

The above code is **intentionally blurred**. Write it yourself!

Problem 10. Run the Unit Tests Again

Run the unit tests again to check whether the methods testing the "insert" functionality work as expected:



Congratulations! You have implemented your binary heap.