Lab: Heaps and Priority Queues

This document defines the **in-class exercises** assignments the "Data Structures" course @ Software University. You can submit your code in the SoftUni Judge System - https://judge.softuni.bg/Contests/590/Heaps-and-Priority-Queues-CSharp-Lab.

Problem 1. Max Binary Heap

You are given a skeleton. You should implement the following operations:

- **`int Count`** → returns the number of elements in the structure
- **`void Insert(T item)`** → adds an element
- **`T Peek()`** → returns the maximum element without removing it
- **`T Pull()`** → removes and returns the maximum element

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

    public int Count { … }
    public void Insert(T item) { … }
    public T Peek() { … }
    public T Pull() { … }
}
```

Examples

```
public static void Main(string[] args)
{
    Console.WriteLine("Created an empty heap.");
    var heap = new BinaryHeap<int>();
    heap.Insert(5);
    heap.Insert(8);
    heap.Insert(1);
    heap.Insert(3);
    heap.Insert(12);
    heap.Insert(-4);

    Console.WriteLine("Heap elements (max to min):");
    while (heap.Count > 0)
    {
        var max = heap.Pull();
        Console.WriteLine(max);
    }
}
```

Insert and Peek

First, you will need a container for all the elements. You can implement a **resizing array** yourself or even better, use the default implementation of your language:

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

The **count** property should return the **size** of the underlying data structure

```
public int Count
{
    get { [redacted] }
}
```

In a **max heap**, the max element should always stay at index 0. Peek should return that element, without removing it

```
public T Peek()
{
    [redacted]
}
```

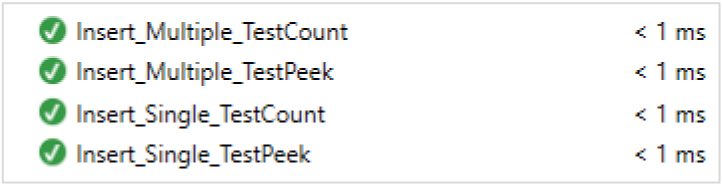
Inserting an element should put it at the end and then bubble it up to its correct position. **`HeapifyUp`** receives as a parameter the index of the element that will bubble up towards the top of the pile.

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

Time to implement **`HeapifyUp`**. While the index is greater than 0 (the element has a parent) and is greater than its parent, swap child with parent. Implement the helper methods (**`Parent()`**, **`IsLess()`** and **`Swap()`**) by yourself.

```
private void HeapifyUp(int index)
{
    while (index > 0 && IsLess(Parent(index), index))
    {
        this.Swap(index, Parent(index));
        index = Parent(index);
    }
}
```

Check if the tests for **`Insert()`** and **`peek()`** are passing


```
✓ Insert_Multiple_TestCount      < 1 ms
✓ Insert_Multiple_TestPeek       < 1 ms
✓ Insert_Single_TestCount        < 1 ms
✓ Insert_Single_TestPeek         < 1 ms
```

Pull

First, check if there are elements in the heap

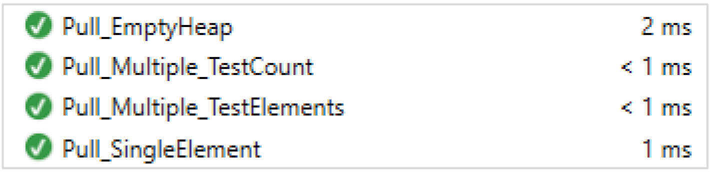
```
public T Pull()
{
    if (this.Count <= 0)
    {
        throw new InvalidOperationException();
    }
}
```

Next, we need to save the element on the top of the heap (index 0), swap the first and last elements, exclude the last element and demote the one at the top until it has correct position

```
T item = this.heap[0];

this.Swap([redacted]);
this.heap.RemoveAt([redacted]);
this.HeapifyDown([redacted]);

return item;
```

The **`HeapifyDown()`** function will demote the element at a given index until it has no children or it is greater than its both children. The first check will be our loop condition

```
private void HeapifyDown(int index)
{
    while (index < [redacted])
    {
        int child = Left(index);
        if (HasChild(child + 1) && IsLess(child, child + 1))
        {
            child = child + 1;
        }

        if (IsLess(child, index))
        {
            break;
        }

        this.Swap(index, child);
        index = child;
    }
}
```



```
✓ Pull_EmptyHeap                  2 ms
✓ Pull_Multiple_TestCount       < 1 ms
✓ Pull_Multiple_TestElements    < 1 ms
✓ Pull_SingleElement              1 ms
```

Problem 2. Heap Sort

You are given a skeleton. Find the static class Heap and implement the following operation:

- **`void Sort(T[] array)`** → performs an in-place sort of the given array in **O(NlogN)** time complexity

Elements should be sorted from smallest to largest.

Example

```
var arr = new int[] { 5, 2, 0, -4, 3, 12 };
Console.WriteLine("Unsorted: " + string.Join(" ", arr));

Heap<int>.Sort(arr);
Console.WriteLine("Sorted: " + string.Join(" ", arr));
```

Solution

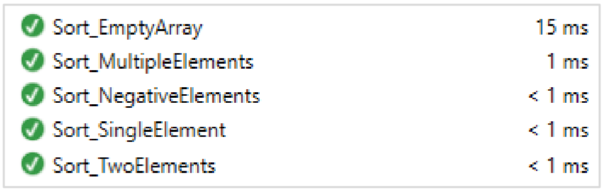
The **`Sort()`** function first builds the heap (first loop) and then finds correct position for each element (second loop)

```
public static void Sort(T[] arr)
{
    int n = arr.Length;
    for (int i = n / 2; i >= 0; i--)
    {
        // TODO: heapify down element at i
    }

    for (int i = n - 1; i > 0; i--)
    {
        // TODO: swap 0 with i
        // TODO: heapify down element at 0
    }
}
```

You will need a modified version of the **`HeapifyDown()`** operation that receives the array and the border of unsorted/sorted elements

```
private static void Down(T[] arr, int current, int border)
{
    while (current < border / 2)
    {
        // TODO: get greater child

        // TODO: if current is greater than greater child -> break

        // TODO: swap elements
        // TODO: update index
    }
}
```

Check if all tests pass



```
✓ Sort_EmptyArray          15 ms
✓ Sort_MultipleElements     1 ms
✓ Sort_NegativeElements   < 1 ms
✓ Sort_SingleElement      < 1 ms
✓ Sort_TwoElements        < 1 ms
```

Congratulations, you have completed the lab for Heaps and Priority Queues!