**Lab: Heaps and BST**

This document defines the lab for ["Data Structures – Fundamentals (C#)" course @ Software University](https://softuni.bg/trainings/3419/data-structures-fundamentals-with%20csharp-june-2021). Please submit your solutions (source code) of all below described problems in [Judge](https://judge.softuni.bg/Contests/2493/05-Heaps-and-Binary-Trees-Lab).

**Do not change the names of the provided projects, interfaces, classes and methods. You are free to create new ones as long as you follow the previously described rule.**

You have to implement all the methods in order to solve the problems below. Each problem is a single task, however you are free to add more methods with any access modifier you want.

## Binary Tree

Inside the given skeleton. You should implement the **IAbstractBinaryTree<T>** class with the following operations:

* **T Value –** gets the **value** of a node
* **IAbstractBinaryTree<T> LeftChild –** returns the **left sub tree** of a node
* **IAbstractBinaryTree<T> RightChild –** returns the **right sub tree** of a node
* **string AsIndentedPreOrder(int indent) –** returns the tree as a **string** - each inner level is **idented with the requested number of** **spaces** as **padding**
* **List<IAbstractBinaryTree<T>> PreOrder() –** returns the **tree** in **pre-order** – first we **add** the **visiting** node then we **continue** with the **left** and **right** child
* **List<IAbstractBinaryTree<T>> InOrder() –** returns the **tree** in **in-order** – first we move **left** as **much** as we **can** then **add** the **visiting** node and then we continue the **right** child
* **List<IAbstractBinaryTree<E>> PostOrder()** – returns the **tree** in **post-order** – first we move **left**, then **right** and at the end as we **have no path**, we **add** the **visiting** node
* **void ForEachInOrder(Action<T> action) – applies** an **Action** on **each** node traversed **in-order**

### Examples

This problem is really a lot like **DFS** or **BFS** we already **know** **how** to **solve**. With a little **twist** we can reuse **recurrence** and **solve** all of them. Think about the **definition** which **action** is **before** the **next** one and the **other** **way** **around**.

**Hints:**

There are of course **hints** inside the **presentation** if you are **stuck** **somewhere**. Do not forget to validate the input parameters wherever this is possible (do not throw exceptions).

## MaxHeap

## Inside the given skeleton. You should implement the MaxHeap<T> class with the following operations:

* int Size – returns the **number** of **elements** in the structure
* void Add(T item) – **adds** an **element**
* T Peek() – returns the **maximum** **element** **without** **removing** it. If there are no elements, throw an **InvalidOperationException**.

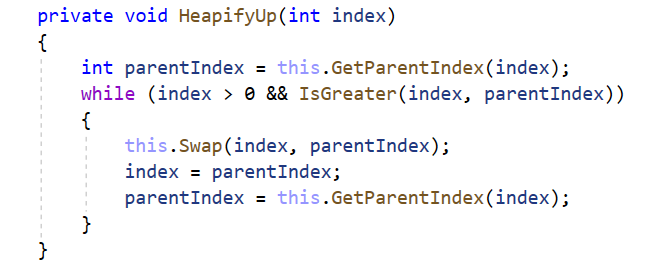
### Peek

In a **max heap**, the max element should always stay at **index 0**. Peek should return that element, without removing it. Verify that the structure is not empty, otherwise throw **InvalidOperationException** with some message.

### Add

Adding 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.

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 by yourself.



## PriorityQueue

Inside the given skeleton. You should implement the **PriorityQueue<E>** class with the following operations:

* int Size – returns the **number** of **elements** in the structure
* void Add(T item) – **adds** an **element**
* T Peek() – returns the **maximum** **element** **without** **removing** it. If there are no elements, throw an **InvalidOperationException**.
* **T Dequeue() –** returns the **maximum** **element** **and** **removes** it**.** If there are no elements, throw an **InvalidOperationException**.

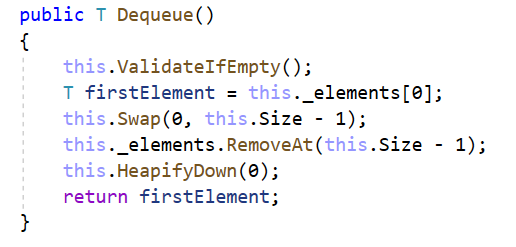
**Add, Peek and Size**

**How different are those methods to the once implemented for the MaxHeap problem? Can you reuse those methods?**

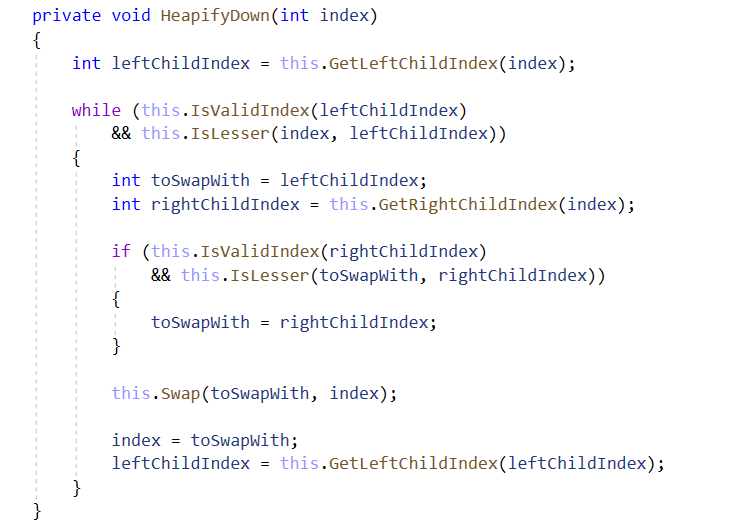
**Dequeue**

In a **PriorityQueue**, the max element should always stay at index 0. **Peek** should return that element, and remove it. Verify that the structure **is not empty**, otherwise throw **InvalidOperationException** (the message is optional).

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**



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



## Binary Search Tree (BST)

Inside the given skeleton. You should implement the **BinarySearchTree<T>** class with the following operations:

* **void Insert(T item)** – **adds** an **element** only if it does not already exists.
* **bool Contains(T item)** – returns a value indicating whether the passed in element exists in the tree.
* **IAbstractBinarySearchTree<T> Search(T item) –** returns a BST **with given** **element value** if such exists. If not, returns null.
* **Node<T> Root** – gets the **root** of the tree
* **Node<T> LeftChild –** gets **the left** subtree of the **node**
* **Node<T> RightChild** – gets **the right** subtree of the **node**
* **T Value –** gets the value of **the root**

### Examples

This time you have to solve the problem on **your own**. Think about it we **know** all we **need** to so **far**. It is **pretty** **simple**.

**Hints:**

There are of course **hints** inside the **presentation** if you are **stuck** **somewhere**.