**Lab: Trees Representation and Traversal (BFS and DFS)**

This document defines the lab for the ["Data Structures – Fundamentals with C#" course @ Software University](https://softuni.bg/trainings/4266/data-structures-fundamentals-with-csharp-september-2023). Please submit your solutions (source code) of all below-described problems in [Judge](https://judge.softuni.org/Contests/2447/03-Trees-Representation-and-Traversal-BFS-and-DFS-Lab).

Your task is to implement the **ADS** **IAbstractTree<T>** inside the **Tree<T>** class provided.

**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 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.

## Create Tree

### **Implement the **Tree** class's constructor to set the **correct key** and to be able to build a full tree by accepting **all the children** for each node and set **their parent**.**

### Solution

#### First Constructor

Text

Description automatically generated with medium confidence

#### Second Constructor

Graphical user interface, text, application

Description automatically generated

## BFS Traversal

### **Implement the **Tree** class's **OrderBFS()** method which should **traverse** the elements in order of each level sequentially.**

## DFS Traversal

### **Implement the **Tree** class's **OrderDFS()** method which should traverse the elements in terms of **descendant before sibling** order of each level sequentially. Try to implement it using** [recursion](https://en.wikipedia.org/wiki/Recursion_(computer_science))**.**

## Add Child

Find the **Tree** **node** with a **specified** **key** and **add** a **child** tree to its children.

You must traverse all the Nodes and find the one with the **same key**. After that simply attach the child **Tree**. If the searched node doesn't exist throw an **ArgumentNullException**.

## Remove Node

Find a **Tree** **node** with a **specified** **key** and **remove it from the tree.**

If we remove a **leaf,** we only affect that **specific** **node**. If we remove an internal node, we remove **the entire subtree** that have that node as parent. Simply put, removing a node **also** **removes** **any** **descendants** of that node. If the searched node doesn't exist throw an **ArgumentNullException**. If we try to remove the root node, we should throw **ArgumentException**.

**Example:**

|  |  |  |
| --- | --- | --- |
| **Initial Tree** | **Operation** | **Result Tree** |
|  | Remove(19) |  |

## Swap Nodes

This time you must find the **Tree** **nodes** with the **specified** **keys** and **swap them.** Swap the **two** **nodes** alongside their **descendants**. If **one** of the searched nodes doesn't exist throw an **ArgumentNullException**. Keep in mind that **swapping** should also **arrange** **the** **references** inside the **nodes** in a **proper** **way**. When swapping a node with it’s own child, remove the original node and all other descendants and leave only the child that is swapped. If you need to swap the root, throw **ArgumentException**

**Example:**

|  |  |  |
| --- | --- | --- |
| **Initial Tree** | **Operation** | **Result Tree** |
|  | Swap(19, 14) |  |
|  | Swap(19, 31) |  |