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

This document defines the lab for ["Data Structures – Fundamentals with C#" course @ Software University](https://softuni.bg/trainings/3112/data-structures-fundamentals-with-csharp-september-2020). Please submit your solutions (source code) of all below described problems in [Judge](https://judge.softuni.bg/Contests/2447/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.**

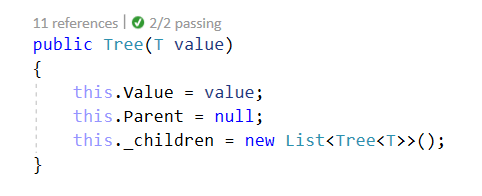
You have to implement all the methods in order to solve the problems below. Each problem is a single task, however you are free to add more methods with any access modifier you want.

## Create Tree

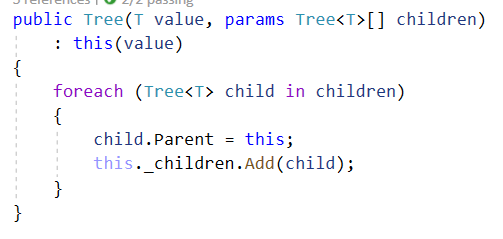
### **Implement the **Tree** class's constructor to set the **correct key** and to be able to build full tree by accepting **all the children** for each node and set **their parent**.**

### Solution

#### First Constructor



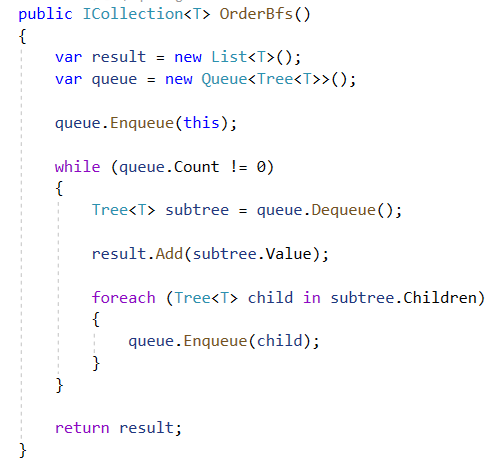
#### Second Constructor



## BFS Traversal

### **Implement the **Tree** class's **public ICollection<T> OrderBFS()** method which should **traverse** the elements in order of each level sequentially. Be careful - the order of output matters.**

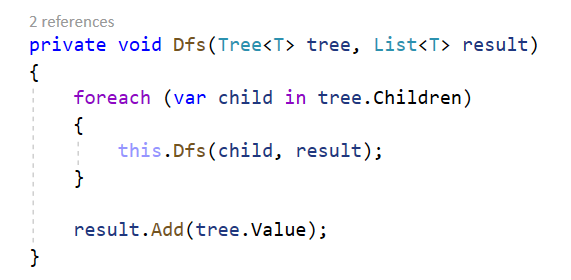
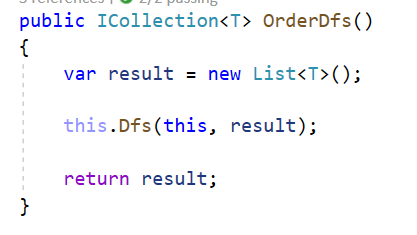
### Solution



## DFS Traversal

### **Implement the **Tree** class's **public ICollection<T> OrderDFS()** method which should traverse the elements in terms of **descendant before sibling** order of each level sequentially. Be careful - the order of output matters. Try to implement it using** [recursion](https://en.wikipedia.org/wiki/Recursion_(computer_science))**.**

### Solution



### Bonus Solution

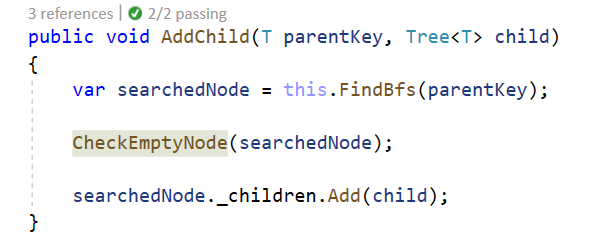
Try to think of a way to implement the **DFS algorithm** using a **Stack**.

## Add Child

Implement the Tree class's **public void AddChild method**. 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**.

### Solution



Implement the **FindBfs** and **CheckEmptyNode** methods on your own. You can also use a DFS algorithm to find the wanted node.

## Remove Node

Implement the Tree class's **public void RemoveNode** method. Find a **Tree** **node** with a **specified** **key** and **remove it from the initial tree.**

Removing **also** **removes** **any** **descendants** of that node. Simple example would be if we remove the **root** of the **tree,** we get **empty** **tree** afterwards. If the searched node doesn't exist throw an **ArgumentNullException**.

If we remove a **leaf,** we only affect that **specific** **node**. If we remove node that is **parent** to **leaf/s** we remove **all** **leafs** that have the node as parent **etc…** Think of all the possible cases of the remove operation.

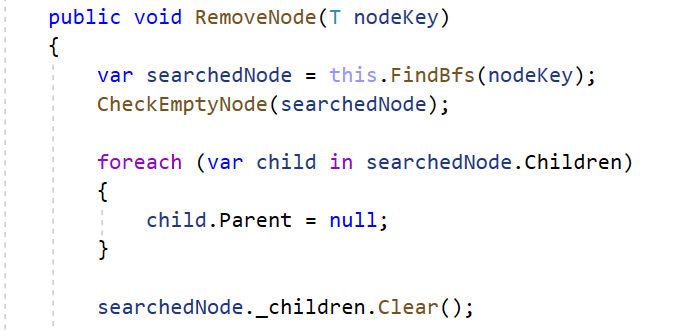
For this task the tests will traverse the nodes in **BFS order**.

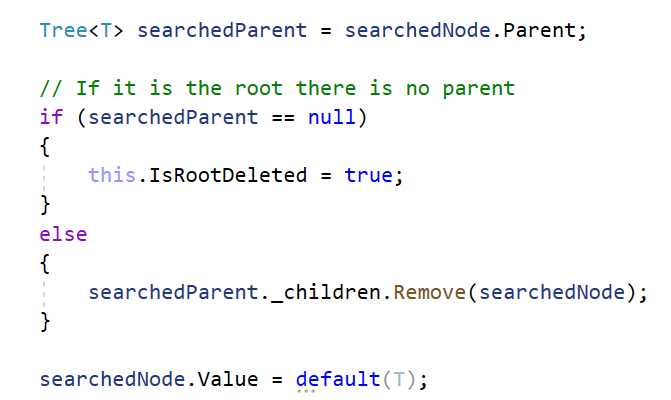
**Note**: if there are **no nodes** simply **return** **an empty List<E>**.

**Example:**

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

### Solution





This **IsRootDeleted** flag marks if we have removed the root node. Make sure that if the root node is removed the BFS and DFS algorithms return an **empty collection**.

## Swap Nodes

Implement the Tree class's **public void Swap** method. 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**. Think about the edge cases what will happen if we attempt to **swap** the **root** with a **leaf**. Or in this order one of the **middle** nodes with a **leaf** etc.…

For this task the tests will traverse the nodes in **BFS** **order**.

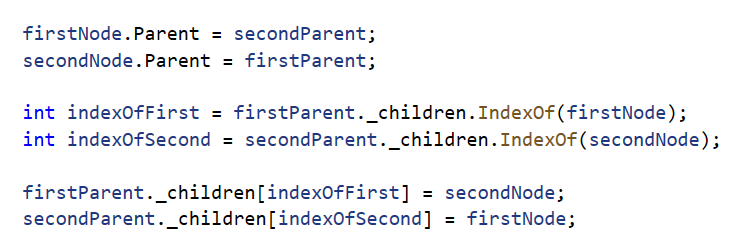
Note: if there are **no nodes** simply **return an empty List<E>**.

**Example:**

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

### Solution

Check if the **nodes exist**, after that switch their **parent references** and switch them in **both children collections**.



If one of the nodes is the **root node**, change the **root node's value** and **replace the old children** with the new ones inside the second node. Try to do that by yourself.