Lab: Trees Representation and Traversal (BFS, DFS)

This document defines the lab for "Data Structures – Fundamentals (Java)" course @ Software University.

Please submit your solutions (source code) of all below-described problems in Judge.

Write Java code for solving the tasks on the following pages. Code should compile under the Java 8 and above standards, you can write and locally test your solution with the Java 13 standard, however, **Judge will run the submission with Java 10 JRE**. Avoid submissions with **features included after Java 10** release doing **otherwise** will result in **compile time error**.

Any code files that are part of the task are provided as **Skeleton**. In the beginning import the project skeleton, do not change any of the interfaces or classes provided. You are free to add additional logic in form of methods in both interfaces and implementations you are not allowed to delete or remove any of the code provided. Do not change the names of the files as they are part of the tests logic. **Do not change the packages** or move any of the files provided inside the skeleton if you have to add a new file add it in the same package of usage.

Some **tests may be provided** within the skeleton – use those for local **testing and debugging**, however, there **is no guarantee that there are no hidden tests added inside Judge**.

Please follow the exact instructions on uploading the solutions for each task. Submit as **.zip archive** the files contained inside the **"...\src\main\java"** folder this should work for all tasks regardless of current DS implementation.

For the solution to compile the tests **successfully** the project **must** have a **single Main.java** file containing single **public static void main(String[] args)** method even an empty one within the **Main class**.

Some of the problems will have simple **Benchmark tests** inside the skeleton. You can try to run those with **different values** and **different implementations** to **observe** behavior. However, **keep** in mind that the result comes **only as numbers** and this data may be **misleading** in some situations. Also, the tests are not started from the command prompt which may **influence** the **accuracy** of the results. Those tests are only added as an **example** of **different data structures performance** on their **common** operations.

The Benchmark tool we are using is **JMH** (Java Microbenchmark Harness) and which is a Java harness for building, running, and analyzing, **nano/micro/milli/macro** benchmarks written in Java and other languages targeting the JVM.

Additional information can be found here: JMH and also there are other examples over the **internet**.

Important: when importing the skeleton **select import project** and then **select from** the **maven module**, this way any following **dependencies** will be **automatically resolved**. The project has **NO default version** of **JDK so after the import you may (depending on some configurations) need to specify the SDK, you can download JDK 13** from **HERE**.

Your task is to implement the **ADS AbstractTree<E>** inside the **Tree<E>** class provided. 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.

© SoftUni – about.softuni.bg. Copyrighted document. Unauthorized copy, reproduction or use is not permitted.

1. 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. Also, make sure to create the proper fields.

Solution:

```
public Tree(E key, Tree<E>... children) {
    this.key = key;
    this.children = new ArrayList<>();
    for (Tree<E> child : children) {
        this.children.add(child);
        child.parent = this;
    }
}
```

2. BFS Traversal

Implement the Tree class's **public List<E> orderBfs ()** method which should traverse the elements in order of each level sequentially be careful the order of output matters.

Solution:

```
@Override
public List<E> orderBfs() {
    List<E> result = new ArrayList<>();
    Deque<Tree<E>> queue = new ArrayDeque<>();
    queue.offer( e: this);
    while (queue.size() > 0) {
        Tree<E> current = queue.poll();
        result.add(current.key);
        for (Tree<E> child : current.children)
            queue.offer(child);
    }
    return result;
}
```


© SoftUni – about.softuni.bg. Copyrighted document. Unauthorized copy, reproduction or use is not permitted.

3. DFS Traversal

Implement the Tree class's **public List<E> 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.

Solution:

```
@Override
public List<E> orderDfs() {
    List<E> order = new ArrayList<>();
    this.dfs( tree: this, order);
    return order;
}

private void dfs(Tree<E> tree, List<E> order) {
    for (Tree<E> child : tree.children) {
        this.dfs(child, order);
    }
    order.add(tree.key);
}
```

4. Add Child

This time you have to find a **Tree node** with a **specified key** and **add** a **child** tree to its children. You have to traverse all the Nodes and find the one with the same key. After that simply attach the child Tree. Make some tests of your own to see how you can make a single tree from multiple trees regardless of the sizes of both.

5. Remove Node

This time you have to find a **Tree node** with a **specified key** and **remove it from the initial tree.** You have to be sure that removing **also removes any descendants** of that node. A simple example would be if we remove the **root** of the **tree** we get an **empty tree** afterward. If we remove a **leaf** we only affect that **specific node**. If we remove a 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.

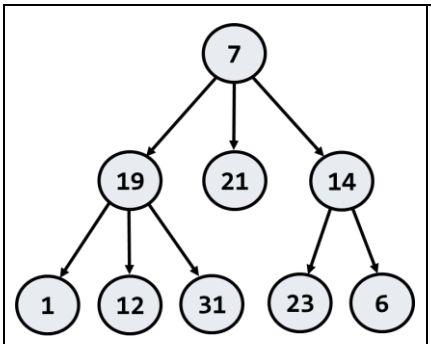
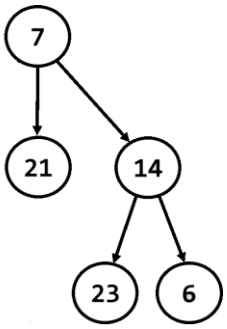
For this task, the tests will traverse the nodes in **BFS order**.

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

Example:

Initial Tree	Operation	Result Tree

© SoftUni – about.softuni.bg. Copyrighted document. Unauthorized copy, reproduction or use is not permitted.

	`Remove(19)`	

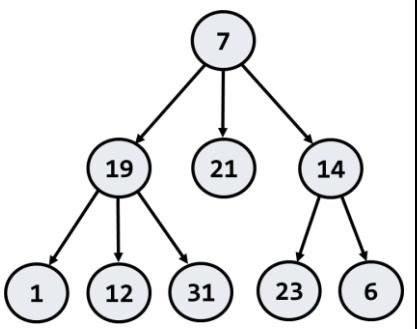
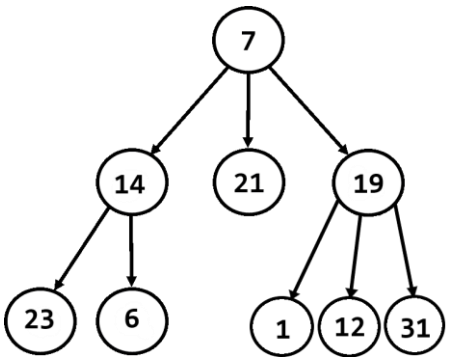
6. Swap Nodes

This time you have to find a **Tree node** with **specified keys** and **swap them.** Now this problem is a **bit harder**. But you **have all the knowledge** required. So what you have to do is **swap two nodes** alongside their descendants. 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)`	

"The first principle is that you must not fool yourself and you are the easiest person to fool." — Richard Feynman

© SoftUni – about.softuni.bg. Copyrighted document. Unauthorized copy, reproduction or use is not permitted.