Exercises: B-Trees-2-3-Trees and AVL Trees

This document defines the lab for "Data Structures – Advanced (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 a **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 the 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 test 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 for uploading the solutions for each task. Submit as **.zip archive** the files contained inside **"...\src\main\java"** folder this should work for all tasks regardless of the current DS implementation.

In order for the solution to compile the tests **successfully** the project **must** have a **single Main.java** file containing a 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** in order 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.

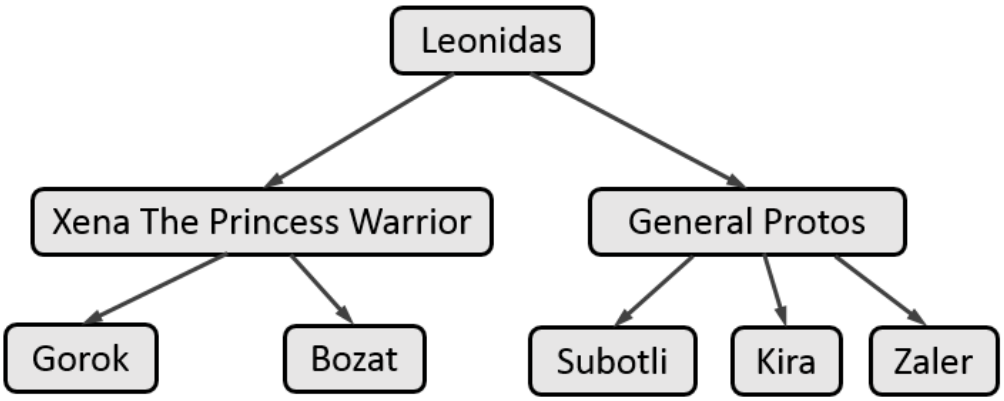
The Benchmark tool we are using is **JMH** (Java Microbenchmark Harness) and which is a Java harness for building, running, and analyzing, **nano/micro/milli/macro** benchmarks written in Java and other languages targeting the JVM.

Additional information can be found here: JMH and 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**.

1. Hierarchy

A **Hierarchy** is a data structure that stores elements in a hierarchical order. See the example:



It supports the following operations:

- **`add(element, child)`** - adds **`child`** to the hierarchy as a child of **`element`**.
  - Throws an exception if **`element`** does not exist in the hierarchy.
  - Throws an exception if **`child`** already exists (duplicates are not allowed).
- **`remove(element)`** - removes the element from the hierarchy.
  - If it has children, they become children of the element's parent.
  - If an element is the root node, throws an exception.
- **`getCount()`** - returns the count of all elements in the hierarchy
- **`contains(element)`** - determines whether the element is present in the hierarchy.
- **`getParent(element)`** - returns the parent of the element.
  - Throws an exception if an **`element`** does not exist in the hierarchy.
  - Returns the **default value for the type** (e.g. **`int`** → **`0`**, **`string`** → **`null`**, etc.) if an element has no parent.
- **`getChildren(element)`** - returns a collection of all direct children of the element in order of their addition.
  - Throws an exception if an **`element`** does not exist in the hierarchy.
- **`getCommonElements(Hierarchy other)`** - returns a collection of all elements that are present in both hierarchies (order does not matter).
- **`forEach()`** - enumerates all elements in the hierarchy by levels.
  - In the image above, the elements would be enumerated as such - **Leonidas** -> **Xena the Princess Warrior** -> **General Protos** -> **Gorok** -> **Bozat** -> **Subotli** -> **Kira** -> **Zaler**.

Input and Output

You are given an **IntelliJ Java project** holding the interface **`IHierarchy`**, the unfinished class **`Hierarchy`** and **tests** covering its **functionality** and its **performance**.

Your task is to **finish this class** to make the tests run correctly.

- You are **not allowed to change the tests**.
- You are **not allowed to change the interface**.

Interface IHierarchy

```
public interface IHierarchy<T> extends Iterable<T> {
    int getCount();
    void add(T element, T child);
    void remove(T element);
    Iterable<T> getChildren(T element);
    T getParent(T element);
    boolean contains(T element);
    Iterable<T> getCommonElements(IHierarchy<T> other);
}
```

2. Implement 2-3 Tree Insertion

You are given an **IntelliJ Java project** holding the unfinished class **TwoThreeTree** and **tests** covering its **functionality**.

Your task is to implement the insertion method. You are free to create additional methods of your own choice, however, only the insertion will be tested. Do **not change the getAsString()** method it is used in testing.

Hint: If you have any problem with the implementation you can check the presentation from the lab, here are some graphs describing the steps while adding items.

3. Implement AVL Tree Deletion

Extend your AVL Tree to support:

- **`void deleteMin()`** → deletes the minimum element (balances the tree if necessary)
- **`void delete(T item)`** → deletes the given element (balances the tree if necessary)
- **`void deleteMax()`** → deletes the maximum element (balances the tree if necessary)

You are given a skeleton with additional tests that cover delete operations.

"It's still magic even if you know how it's done." (A Hat Full of Sky)