# Exercises: Heaps and BST

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 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 **"...\src\main\java"** folder this should work for all tasks regardless of current DS implementation.

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

Some of the problem will have simple **Benchmark tests** inside the skeleton. You can try to run those with **different values** and **different implementations** in order to **observe** behaviour. 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 that is 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 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 (depends on some configurations) need to specify the SDK, you can download JDK 13** from **HERE.**

## 1. BST Operations

You are given a skeleton, in which you will have to implement the following operations:

- `void insert(E)` – Recursive implementation
- `void eachInOrder(Consumer<E>)` – In-Order traversal
- `bool contains(E)` – Iterative implementation
- `BST<E> search(E)` – Returns copy of the BST
- `List<E> range(E, E)` – Returns collection with the elements found in the BST. Both borders are **inclusive**.
- `deleteMin()` – Deletes the smallest element in the tree. Throws exception if the tree is empty.

You can reuse the solution from the Lab we had and only add implementations for the new methods. However it is indeed a good idea to try implementing those methods again, repetition will give you better understanding and you can choose some different approach.

# Implement additional methods described below:

## Delete Max

Implement a **method** which **deletes** the **max element** in a BST (Binary Search Tree). If the tree is empty it should throw exception **IllegalArgumentException**. The logic is similar to the **`deleteMin()`** method, but you need to traverse the tree to the right.

## Count

Implement a **method** which returns the count of elements in the BST.

### Hints

If our current node is **`null`**, we will return 0. Otherwise, we will return the count of our current node:

We also have to modify our **`insert()`** method. It will set the count of elements of our new node to the count of its children nodes plus itself:

Next, we need to find a way to update the recalculate the count for each node when **`deleteMin()`** is invoked. One way would be to change the **`deleteMin()`** implementation to be recursive:

What will happen if our tree is empty and we call **`deleteMin()`**? **Fix** it. Our count is ready.

## Rank

Implement a **method** which **returns** the **count** of elements **smaller than** a given **value**.

### Hints

Create a new recursive method that will **`return 0`** if the node is null:

Then, we need to **compare the element** with the value of the node we are currently looking at. If the element is **smaller**, we can **go to the left**. If it is **larger**, we need to **get the count of the left** elements and **go to the right**. If we **find the element**, we will return the **count of elements**, **smaller** than it.

You can try it out, it should work as expected.

## Floor

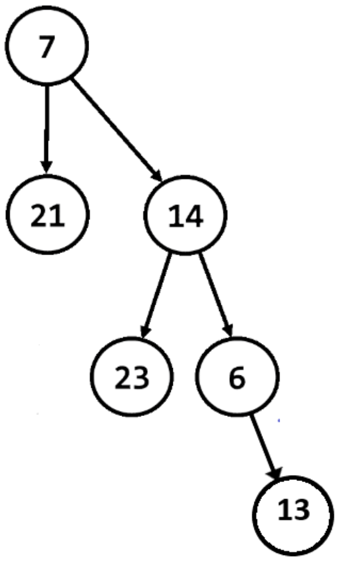
Implement a **method** which **finds** (returns) the **nearest smaller value** than given in the BST. This operation is similar to **`deleteMin()`**.

## Ceiling

Implement a **method** which **finds** (returns) the **nearest larger value** than given in the BST. This operation is similar to **`floor()`** and **`deleteMax()`**.

## 2. Lowest Common Ancestor

You are given the binary tree and two values **$V_1$** and **$V_2$**. You need to return the lowest common ancestor (**LCA**) of **$V_1$** and **$V_2$** in the binary tree. Note that **lowest** here **means** in terms of **level distance**. The close the node is to both values the **lower** we say it is. In other words you can **ignore** the **value** you **should only care for the distance**.



In the given tree above the lowest common ancestor of the nodes 23 and 13 is the **node 14**. Node 14 is the **lowest** node which has nodes 23 and 13 **as descendants**.

## 3. Min Heap

Based on the implementation of the **MaxHeap<E>** and ADS **Heap<E>** from the lab implement data structure that acts and operates the same way, however this time the **heap property** you want to **preserve** is the **Min heap property**.

You will be given **no hints** here since the solution is really **similar** to the one from the lab.

Remember you have to throw **IllegalStateException** if you attempt to **peek () or poll () on empty heap**.

In addition **poll ()** should simply **remove** the element at the **top** of the **heap** (remember how **PriorityQueue worked**).

If you get **stuck** remember **you have** all the code inside the **lab resources**.

## 4. Decrease Key

Extend your Min Heap to support the **decrease** (E element) operation that changes the priority of a given key. In a Min Heap this should increase the priority of a given key, moving it higher in the tree structure, e.g. decreasing the price of a given product, increases its priority for the customers. Study the code provided you have to focus on the implementation of that method everything else is provided.

## 5. Cookies

We want the sweetness of all his cookies to be greater than value **K**. To do this, we need to repeatedly mix two cookies with the least sweetness. We creates a special combined cookie with sweetness calculated by:

- (**least sweet** cookie) + (**2** * **2nd least sweet cookie)**.

We repeats this procedure until all the cookies in the collection have a sweetness **not less than K**.

You are given the cookies. Return the number of operations required to give the cookies a sweetness **not less than K**. Return **-1** if this isn't possible.

Implement the **Integer** solve (int **k**, int[] **cookies**) **method** inside the provided **CookiesProblem** class.

## Input

Study the tests provided. Basically the first parameter is the **K** value and the second one is the **array** representing the **cookies**.

The first line consists of integers, the number of cookies and, the minimum required sweetness, separated by a space.

## Output

The method should **return a single integer -1** if there is **no solution**. **Otherwise** return the **number of operations** required to complete the task.

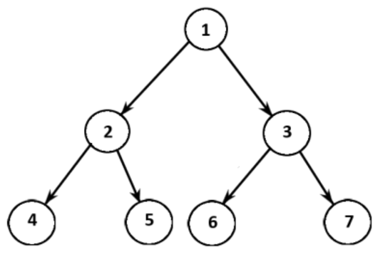
| Input | Output |
|---|---|
| 7<br>1 2 3 9 10 12 | 2 |

## 6. Top View

You are given the binary tree. Print the **top view** of the binary tree. More info can be fund here.

Top view means when you look the tree from the top the nodes, what you will see will be called the top view of the tree. See the example below.

### Examples



Given the above tree the result should be: **1, 2, 4, 3, 7,** where order of **output does not matter.**

"In the beginning there was nothing, which exploded."

— Terry Pratchett, Lords and Ladies