Lab: 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. Binary Tree

Inside the given skeleton. You should implement the **BinaryTree<E>** class with the following operations:

- **E getKey()** – returns the **key** of a node
- **AbstractBinaryTree<E> getLeft()** – returns the **left sub tree** of a node
- **AbstractBinaryTree<E> getRight()** – returns the **left right tree** of a node
- **void setKey(E key) – sets** the **key** of a node
- **String asIndentedPreOrder(int indent)** – returns the tree as **String** each inner level is **idented with 2 spaces** as **padding**

- **List<AbstractBinaryTree<E>> preOrder() –** returns the **tree** in **preOrder** – first we **add** the **visiting** node then we **continue** with the **left** and **right** child
- **List<AbstractBinaryTree<E>> inOrder() –** returns the **tree** in **inOrder** – first we move **left** as **much** as we **can** then **add** the **visiting** node and then we continue the **right** child
- **List<AbstractBinaryTree<E>> postOrder()** – returns the **tree** in **postOrder** – first we move **left**, then **right** and at the end as we **have no path**, we **add** the **visiting** node
- **void forEachInOrder(Consumer<E> consumer) – applies** a **Consumer<E>** on **each** node traversed **inOrder**

Examples

Look at the provided tests inside the skeleton.

This problem is really a lot like **DFS** or **BFS** we already **know how** to **solve**. With a little **twist** we can reuse **recurrence** and **solve** all of them, think about the **definition** which **action** is **before** the **next** one and the **other way around** etc…

Hints:

There are of course **hints** inside the **presentation** if you are **stuck somewhere**.

2. MaxHeap

Inside the given skeleton. You should implement the **MaxHeap<E>** class with the following operations:

- **int size()** – returns the **number** of **elements** in the structure
- **void add(E element)** – **adds** an **element**
- **E peek()** – returns the **maximum element without removing** it

```
public class MaxHeap<E extends Comparable<E>> implements Heap<E> {
    private List<E> elements;

    public MaxHeap() {
        this.elements = new ArrayList<>();
    }
```

Examples

Look at the provided tests inside the skeleton.

Peek

In a **max heap**, the max element should always stay at index 0. Peek should return that element, without removing it. Verify that the structure is not empty, otherwise throw **IllegalStateException** with some message.

```java
@Override
public E peek() {
    if (this.size() == 0) {
        throw new IllegalStateException("Heap is empty upon peek attempt");
    }
    return this.elements.get(0);
}
```

Add

Adding an element should put it at the end and then bubble it up to its correct position. **HeapifyUp** receives as a parameter the index of the element that will bubble up towards the top of the pile.

```java
@Override
public void add(E element) {
    this.elements.add(element);
    this.heapifyUp( index: this.size() - 1);
}
```

Time to implement **HeapifyUp**. While the index is greater than 0 (the element has a parent) and is greater than its parent, swap child with parent. Implement the helper methods (**parent()** and **less()**) by yourself.

```java
private void heapifyUp(int index) {
    while (hasParent(index) && less(parent(index), elements.get(index))) {
        int parentAt = getParentAt(index);
        Collections.swap(this.elements, parentAt, index);
        index = parentAt;
    }
}
```

3. PriorityQueue

Inside the given skeleton. You should implement the **MaxHeap<E>** class with the following operations:

- **int size()** – returns the **number** of **elements** in the structure
- **void add(E element)** – **adds** an **element**
- **E peek()** – returns the **maximum element without removing** it
- **E poll() –** returns the **maximum element and removes** it

Examples

Look at the provided tests inside the skeleton.

Add, Peek and Size

How **different** are those methods to the once **implemented** for the **MaxHeap problem**? Can you **reuse** those methods?

Poll

In a **PriorityQueue**, the max element should always stay at index 0. **Peek** should return that element, and remove it. Verify that the structure **is not empty**, otherwise throw **IllegalStateException** with some message.

```java
private void ensureNonEmpty() {
    if (this.size() == 0) {
        throw new IllegalStateException("Heap is empty upon peek/poll attempt");
    }
}
```

Next, we need to save the element on the top of the heap (index 0), **swap** the **first** and **last elements**, **exclude** the **last element** and **demote** the one **at the top until it has correct position**

```java
@Override
public E poll() {
    ensureNonEmpty();
    E element = this.elements.get(0);
    Collections.swap(this.elements, 0, this.elements.size() - 1);
    this.elements.remove(this.elements.size() - 1);
    this.heapifyDown(0);
    return element;
}
```

The **`HeapifyDown()`** function will demote the element at a given index until it has no children or it is greater than its both children. The first check will be our loop condition

```java
private void heapifyDown(int index) {
    while (index < this.elements.size() / 2) {
        int child = 2 * index + 1;

        if (child + 1 < this.elements.size() && less(this.elements.get(child), this.elements.get(child + 1))) {
            child = child + 1;
        }

        if (less(this.elements.get(child), this.elements.get(index))) {
            break;
        }

        Collections.swap(this.elements, index, child);
        index = child;
    }
}
```

4. Binary Search Tree (BST)

Inside the given skeleton. You should implement the **BinarySearchTree<E>** class with the following operations:

- **`void insert(E element)`** – **adds** an **element**
- **`boolean contains(E element)`** – returns the **maximum element without removing** it
- **`AbstractBinarySearchTree<E> search(E element)` –** returns the **tree with given element value as root** if exists if not return **empty** tree
- **Node<E> getRoot()** – returns the **root** of a tree
- **Node<E> getLeft()** – returns the **leftChildren** of a tree node

- **Node<E> getRight()** – returns the **rightChildren** of a tree node
- **E getValue()** – returns the **value** of a tree node

Examples

Look at the provided tests inside the skeleton.

This time you have to solve the problem on **your own**. Think about it we **know** all we **need** to so **far**. It is **pretty simple**. Use the **tests** provided and create **new test** cases for **debugging** and code **correctness** validation.

Hints:

There are of course **hints** inside the **presentation** if you are **stuck somewhere**.

"Somewhere, something incredible is waiting to be known."

— Carl Sagan