EXAM EXAMPLE

This exam example for PG4200 is composed of 10 questions/exercises. Each question is worth 10 points, for a total of 100 points. You have 3 hours to answer as many of them as possible.

All the questions are written in English. To answer these questions, it is preferred that you do it in English. However, any other language officially recognized by Kristiania (e.g., Norwegian) is obviously acceptable.

Each question/exercise might actually be composed of more than 1 question/part (e.g., several sentences ending with a "?"). You need to address all of them.

When you see required to explain "*in details*", just a couple of sentences will likely not be enough for answering in details. You might expect to need to write at least half-page, but likely no more than a full one. However, these are just high level indications, and of course the amount needed to answer in details depends on the question.

When writing code on a piece of paper or in a text editor (not an IDE), it is obviously expected that there will be syntactic errors. Those will not reduce your grade. Still, the more you can be close to the actual Java syntax, the better. You are **NOT** allowed to use pseudo-code or different programming languages. If you do not remember the exact name for a specific class/method, use a meaningful name that somehow reflects the needed functionality.

When you are asked to implement a class extending a given interface, you will also need to implement any *private/protected* method required to be called from the specified *public* methods.

As discussed in class, you are **NOT** allowed to use "*delegate*" implementations (unless explicitly requested). For example, if you need to implement a *Map*, you cannot rely on an existing implementation, and then call methods on it, e.g. writing something like "*V get(K k){return delegate.get(k);}*" is not allowed.

1) Explain *in details* what are the main differences (if any) between the *Queue* and the *Stack* data structures.

2) On the same problem, consider two algorithms *A* and *B* with worst case complexity for their runtime being bound by *O(n)* for *A*, and by *O(log n)* for *B*. Is *B* always going to be better and preferable over *A*? If yes, then explain why. If not, explain in which cases *A* could be actually better than *B*.

3) If you need to find an element *X* in an array of size *n*, what is the complexity of the worst case, i.e. *O(f(n))*, of number of comparisons you need to do before finding *X*? What can you do to improve it (and by how much?) if you know that the array is sorted?

4) Consider the following code in an implementation of a Hash Map, where *M* is the size of the internal array, and the method *index* returns a position in the array based on the hash code of the key:

```
 private int index(K key){
    int hash = key.hashCode() & 0x7f_ff_ff_ff;
    return hash % M;
 }
```

Why there is the need for "`& 0x7f_ff_ff_ff`"? What does it do? What could happen if that instruction is removed? Explain *in details*.

5) On graphs, what is the difference between a Depth-First Search (DFS) and a Breadth-First Search (BFS)? Which one should you use? What are the tradeoffs? Explain *in details*.

6) Consider a telephone number as an 8 digit number. It might be preceded by a country code, which is either a + or 00 followed by 2 digits. Write a regular expression to match strings representing valid telephone numbers with such constraints.

7) In the context of decision problems, what are the set *P* and the set *NP*? Is *P*==*NP*? Or is *P*!=*NP*? Explain *in details*.

8) Given the following method signature, implement a Merge Sort algorithm.

```
public <T extends Comparable<T>> void sort(T[] array)
```

9) Consider the following implementation of a *flatMap()* method from the *MyStreamSupport* class seen in the course. Such implementation is incomplete. Add the missing code in the *accept()* method.

```
@Override
public <R> MyStream<R> flatMap(Function<OUT, MyStream<R>> mapper) {
    Objects.requireNonNull(mapper);

    return new Pipeline<OUT, R, T>(this) {

        @Override public
       ChainedReference<OUT, R> chainConsumerToCurrentPipe(
                                                 Consumer<R> consumer) {

            return new ChainedReference<OUT, R>(consumer) {
                @Override
                public void accept(OUT u) {
                    /*
                       Your code here
                     */
                }
            };
        }
    };
}
```

10) Given the following interface, implement it in a concrete class using a Binary Search Tree.

```
public interface MyMap<K extends Comparable<K>, V> {

    /**
     *   Create a mapping from the given Key to the given Value.
     *   If a mapping for Key already exists, replace the old
     *   value with this new one
     */
    void put(K key, V value);

    /**
     * Remove the given key from the container.
     */
    void delete(K key);

    /**
     *  Return the value in the container mapped by the given key
     */
    V get(K key);

    /**
     *  The number of elements in the container
     */
    int size();
    /**
     *  Check if there is no element in the container
     */
    default boolean isEmpty(){
        return size() == 0;
    }
}
```

THIS MARKS THE END OF THE EXAM TEXT