# YSC4231: Parallel, Concurrent and Distributed Programming

Programming Assignment 7

In this assignment you'll be working with actors, their communication and behavioural patterns. The goal of this assignment is to implement a distributed set, based on the binary tree construction, so that the tree itself and each of its nodes are represented by individual actors.

The template repository for this assignment is available on Canvas. Once you clone the template repository, you'll be ready to go! The files you need to modify are:

- `ActorTreeSet.scala`
- `ActorTreeNode.scala`

To hand in your code for this assignment, submit the link to the tagged release in your repository on GitHub. Feel free to provide additional comments about your implementation and discoveries to the `README.md` file in the root of the project. A project is supplied with a number of tests, which you may use to validate the correctness of your implementation. Some of the tests we have implemented are not provided and are reserved for grading purposes. As previously, you are encouraged to add your own tests to the suite.

### Working with the Actor Tree

A client may insert elements (represented by numbers integers) to the set or remove them by means of sending individual operations as messages to the main set actor. Those messages are then appropriately relayed through the tree structure. Different kinds of operations (`Insert`/`Remove`/`Contains`) are implemented as sub-types of the trait `Operation`. Each operation, besides the payload `elem`, contains a reference `requester` to the sender and a custom identifier `id`, which can be used by the client to distinguish between the results of multiple requests. The two kinds of responses are implemented as sub-types of the trait `OperationReply`.

### Implementing Individual Nodes

Each individual node of a tree is implemented by an actor, an instance of the class `ActorTreeNode`. It contains an immutable payload `elem`, as well as a mutable boolean flag `initiallyRemoved`. In order to support multiple concurrent requests to the tree, the nodes corresponding to the removed elements are not detached (thus, the tree may become unbalanced, which is not our concern in this case). Instead, they are marked as *removed*, so they can be "un-removed" in the future if the corresponding element is re-inserted.

### Garbage Collection

In practice, many insertions and removals might lead to the tree being littered with "dead" nodes. To solve this problem, you will need to implement a mechanism of *garbage collection*.

A garbage collection can be initiated by any client of the set by sending a message `GC` to the set actor. Once having received `GC`, the set should rebuild the tree, so it would only contain "alive" nodes. Once the tree is rebuilt, it should replace the one accessible by the old root from the main set object. The actors corresponding to the retired nodes should be put to rest by terminating them.

The garbage collection can be implemented in two steps. First, you may want to implement an internal operation that copies all alive contents of a tree to a new tree (using the `CopyTo` and `CopyFinished` messages). This implementation can assume that no operations arrive while the copying happens (*i.e.*, the tree must be protected from modifications while copying is in progress). The second part of the implementation is replace the old tree by a new one, and then stopping all actors from the old one. Since no concurrent operations are allowed while copying, the implementation should handle the case when operations arrive while the copying is happening in the background. You must ensure that garbage collection happens "invisible" to the clients (some delay delay is allowed). Feel free to ignore GC requests that arrive while garbage collection is in progress. Your GC implementation will be testsed using "hidden" tests reserved for grading.

### Ordering of the Operations

From the perspective of the clients, all operations with the set should be *linearisable* (with regard to the arrangement of the operations and the corresponding replies in a combined history), and the garbage collection should be transparent, *i.e.*, not noticeable by the clients. The linearisability means that your implementation can answer requests from the same client in any order — from their perspective they would correspond to "overlapping" calls. However, if the client waits until each operation's response comes back, the results should be consistent with the sequential behaviour of a set.

### Hints

- Make sure to implement a state-transition logic of your actor tree via `context.become(...)` method.
- Multiple behaviours of an actor can be arranged into an implicit stack by calling `become` with an additional argument: `context.become(..., discardOld = false)`, which is similar to "pushing" a behaviour to a stack. Use `context.unbecome()` to restore the previous behaviour of an actor, in a pop-like manner.
- Check the tests to get an idea on how the set might be used.
- The method `context.parent` returns the references of the actor which created the current actor.
- Ensure that no `Operation` messages interfere during GC and that the user does not receive any messages resulting from the tree copying process.
- Recall that you can stop an actor either by means of `stop` method or by sending it a `PoisonPill`.
- You may want to split the logic of handling the messages by a tree node itself or delegating it to its children in the tree. If you decide to follow this pattern, feel free to use the templates of the functions `handleSelfAndRespond` and `delegateToChild`.
- If you see a log message like

  ```
  [INFO] [...] Message [...] from Actor[...] to Actor[...]
         was not delivered. [1] dead letters encountered.
  ```

  it means that one of your messages hasn't been delivered from an actor the client. You should check that you do not stop actors prematurely.