Actor Binary Tree: Instructions

Help

When you're ready to submit your solution, go to the assignments list.

Attention: You are allowed to submit **an unlimited number of times**! for grade purposes. Once you have submitted your solution, you should see your grade and a feedback about your code on the Coursera website within 20 minutes. If you want to improve your grade, just submit an improved solution.

To get started, download the actorbintree.zip handout archive file and extract it somewhere on your machine.

Binary Trees

Binary trees are tree based data structures where every node has at most two children (left and right). In this exercise, every node stores an integer element. From this we can build a binary search tree by requiring for every node that

- values of elements in the left subtree are strictly smaller than the node's element
- values of elements in the right subtree are strictly bigger than the node's element

In addition, there should be no duplicates, hence we obtain a binary tree set.

Your task in this assignment is to implement an actor-based binary tree set where each node is represented by one actor. The advantage of such an actor-based solution is that it can execute fully asynchronously and in parallel.

The API

You can find the message-based API for the actor-based binary tree to be implemented in the supplied `BinaryTreeSet` object in the file `BinaryTreeSet.scala`.

The operations, represented by actor messages, that the implementation should support are the following:

- `Insert`
- `Remove`
- `Contains`

All three of the operations expect an `ActorRef` representing the requester of the operation, a numerical identifier of the operation and the element itself. `Insert` and `Remove` operations should result in an `OperationFinished` message sent to the provided requester `ActorRef` reference including the id of the operation. `Remove` and `Insert` should return an `OperationFinished` message even if the element was not in the tree or already present, respectively. `Contains` should result in a `ContainsResult` message containing the result of the lookup (a Boolean which is true if and only if the element is in the tree when the query arrives) and the identifier of the `Contains` query.

Handling of Removal

You should observe that both the `Insert` and `Contains` operations share an important property, namely, they only traverse a linear path from the root of the tree to the appropriate inner node or leaf. Since the tree nodes are actors which process messages one-by-one, no additional synchronization is needed between these operations. Removal in a binary tree unfortunately results in tree restructuring, which means that nodes would need to communicate and coordinate between each other (while additional operations arrive from the external world!).

Therefore, instead of implementing the usual binary tree removal, in your solution you should use a flag that is stored in every tree node (`removed`) indicating whether the element in the node has been removed or not. This will result in a very simple implementation that is concurrent and correct with minimal effort. Unfortunately this decision results in the side effect that the tree set accumulates "garbage" (elements that have been removed) over time.

Garbage Collection

As we have seen, removal of entries can be implemented simply by using a removal flag with the added cost of growing garbage over time. To overcome this limitation you will need to implement a "garbage collection" feature. Whenever your binary tree set receives a `GC` message, it should clean up all the removed elements, while additional operations might arrive from the external world.

The garbage collection task can be implemented in two steps. The first subtask is to implement an internal `CopyTo` operation on the binary tree that copies all its non-removed contents from the binary tree to a provided new one. This implementation can assume that no operations arrive while the copying happens (i.e. the tree is protected from modifications while copying takes places).

The second part of the implementation is to implement garbage collection in the manager (`BinaryTreeSet`) by using the copy operation. The newly constructed tree should replace the old one and all actors from the old one should be stopped. Since copying assumes no other concurrent operations, the manager should handle the case when operations arrive while still performing the copy in the background. It is your responsibility to implement the manager in such a way that the fact that garbage collection happens is invisible from the outside (of course additional delay is allowed). For the sake of simplicity, your implementation should ignore GC requests that arrive while garbage collection is taking place.

Ordering Guarantees

Replies to operations may be sent in any order but the contents of `ContainsResult` replies must obey the order of the operations. To illustrate what this means observe the following example:

Client sends:

```
Insert(testActor, id=100, elem=1)
Contains(testActor, id=50, elem=2)
Remove(testActor, id=10, elem=1)
Insert(testActor, id=20, elem=2)
Contains(testActor, id=80, elem=1)
Contains(testActor, id=70, elem=2)
```

Client receives:

```
ContainsResult(id=70, true)
OperationFinished(id=20)
```

```
OperationFinished(id=100)
ContainsResult(id=80, false)
OperationFinished(id=10)
ContainsResult(id=50, false)
```

While the results seem "garbled", they actually strictly correspond to the order of the original operations. On closer examination you can observe that the order of original operations was [100, 50, 10, 20, 80, 70]. Now if you order the responses according to this sequence the result would be:

```
Insert(testActor, id=100, elem=1) -> OperationFinished(id=100)
Contains(testActor, id=50, elem=2) -> ContainsResult(id=50, false)
Remove(testActor, id=10, elem=1) -> OperationFinished(id=10)
Insert(testActor, id=20, elem=2) -> OperationFinished(id=20)
Contains(testActor, id=80, elem=1) -> ContainsResult(id=80, false)
Contains(testActor, id=70, elem=2) -> ContainsResult(id=70, true)
```

As you can see, the responses the client received are the same, hence they must have been executed sequentially, and only the responses have arrived out of order. Thus, the responses obey the semantics of sequential operations -- it is simply their arrival order is not defined. You might find it easier for testing to use sequential identifiers for the operations, since that makes it easier to follow the sequence of responses.

You might also note that out-of-order responses can only happen if the client does not wait for each individual answer before continuing with sending operations.

While this loose ordering guarantee on responses might look strange at first, it will significantly simplify the implementation of the binary tree and you are encouraged to make full use of it.

Your task

You can find code stubs in the file `BinaryTreeSet.scala` which provides you with the API as described above, the `BinaryTreeSet` and `BinaryTreeNode` classes. The `BinaryTreeSet` represents the whole binary tree. This is also the only actor that is explicitly created by the user and the only actor the user sends messages to.

You can implement as many or as few message handlers as you like and you can add additional variables or helper functions. We provide suggestions in your code stub, marked with the comment `optional`, but you are free to use it fully or partially; the optional elements are not part of the tested API.

To see a binary tree in operation check our provided tests in `BinaryTreeSuite.scala`. Note in particular that it is the user who triggers garbage collection by sending a `GC` message (for the sake of simplicity of this exercise).

Don't forget to make sure that no `Operation` messages interfere during garbage collection and that the user does not receive any messages that may result from the copying process.

The following may be useful for your implementation:

- Another way to stop an actor, besides the `stop` method you have seen, is to send it a `PoisonPill` message.
- `context.parent` returns the ActorRef of the actor which created the current actor (i.e. its parent).

- If you see a log message like the following

  [INFO] [11/21/2013 14:04:13.237] [PostponeSpec-akka.actor.default-dispatcher-2] [akka://PostponeSpec/deadLetters] Message [actorbintree.BinaryTreeSet$OperationFinished] from Actor[akka://PostponeSpec/user/$e/my-actor#-1012560631] to Actor[akka://PostponeSpec/deadLetters] was not delivered. [1] dead letters encountered.

it means that one of your messages (here the OperationFinished) message was not delivered from actor `my-actor` to actor `deadLetters`—the latter is where actors forward their messages after they terminate. You should check that you do not stop actors prematurely.