Section B

Problem Description: a complicated chemical supply chain

Overview

In this section you will create a simulated chemical industry exchange market where different players sell and buy basic elements or composite chemicals.

Every player repeatedly performs an action depending on its role. There are four roles:

- *ElementSupplier*: sells batches of basic chemical elements
- *Mixer*: buys the batches of chemical elements it needs, produces a new composite chemical, and sells it
- *Chemist*: buys composite components, uses them, and then disposes of them

The *exchange* is the place where all these operations happen. The exchange keeps *stocks* of each type of chemicals. Suppliers sell basic chemical elements via the exchange, mixers buy them to produce composites that are then sold again via the exchange. You will have to implement the behaviors of the exchange and its players for Q3.

To unnecessarily complicate all the process, each stock implements a peculiar data structure. Items are *pushed* in the stock labeled also with the identifier of the player that sold them. Each player has a unique integer id (`Player.id`). A player with a larger id is considered a more valuable stakeholder and its items are to be sold first. In the real world, this would be a priority queue, where the player's id is the priority of a stored item. Such a priority queue may be implemented with a heap, for example, but not here :)

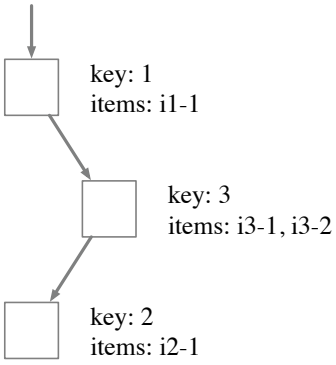
Here you need to implement the stocks as a variation of binary search trees (BSTs). Each node of the tree has a player id as its key and contains a collection of all the items pushed by that player. Items are *popped* out from the collection of the node with the largest id in the tree, using a *last-in first-out* order. When such collection becomes empty, the node is removed from the BST.

For example, consider the case of three agents A1, A2, and A3 (with ids 1, 2, and 3, respectively) performing the following push operations, in the given order:

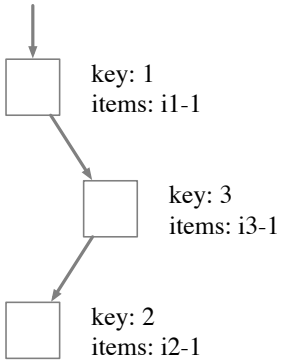
```
push(i1-1, A1)
push(i3-1, A3)
push(i2-1, A2)
push(i3-2, A3)
```

The resulting BST structure is shown in Figure 1a. Consider now a pop operation is performed. The highest priority node is 3 and it contains two items. The last of such items added to the collection is `i3-2`, which is removed from the collection and returned. The resulting BST is shown in Figure 1b.

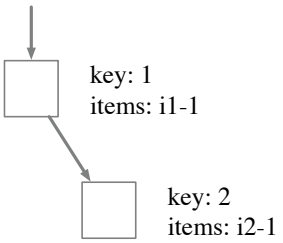
Now another pop is performed. The highest priority node is still 3 and it contains only one item (`i3-1` in the example), which will be returned by the pop operation. Since Node 3 does not contain items anymore, it has to be removed, leaving the BST as shown in Figure 1c.



(a) Stock after the example push sequence.



(b) Stock after the first pop.



(c) Stock after the second pop.

Getting Started

If you have cloned your repo under your Home directory, then you can find these files at:

- `~/javainterimtestpartb_rhj20/src/`

During the test, you will modify **only** the files listed below among those provided in the skeleton source. You are free to add as many files as you see fit, but not to modify provided source files not listed below. You can add tests or otherwise modify the test suite.

- `datastructures/LIFOImpl.java`: where you will implement a last-in first-out (Stack) data structure
- `datastructures/StockImpl.java`: where you will implement the BST-based data structure, whose nodes contain a LIFO collection of items
- `market/ExchangeImpl.java`: where you will implement the functionalities of the exchange
- `players/*`: where you will implement the agents specified below

Do not change the names or the contents of any other provided source class (in the folder `src`). Do not change the public interfaces of the provided sources (apart from possibly synchronizing methods as necessary for Question 4). You can add as many private methods, private fields, or additional classes as you see fit. Look also for comments in the code and at the tests for additional hints.

Testing

You are provided with a set of test cases in the `test` directory. The tests aim at exercising a variety of behaviors of your implementation and to further explain what your code is expected to do. The test suite is not exhaustive: even if your solution passes all the tests, your work will be assessed by the examiners, who may also use a different test suite to check your code.

What to do

1. **LIFO item collections.** In the package `datastructures` you will find `LIFOImpl`, which implements the interface `LIFO`. This is essentially a stack, whose *pop* operation returns the last element that was added via a *push* operation. The tests in `datastructures.LIFOTest` will help you check your implementation.

   Your tasks are:

   - implement the method `push(item)` of `LIFOImpl`, which adds an item to the collection
   - implement the method `pop()` which returns the last item added to the collection
   - implement the method `size()` which returns the number of elements in the collection

   [**7 marks**]

2. **BST-based stocks.**

   In the package `datastructures` you will find `StockImpl`, which implements the interface `Stock`. This is the BST-based priority "queue" described in the overview. The tests in `datastructures.StockTest` will help you check your implementation.

   Heads up: for Q4, you will have to make this structure thread-safe. You may want to think whether to directly design a thread-safe solution or leave this extension for later.

   Your tasks are:

   - design a class for the BST nodes. Each node has a key used to keep the nodes in order. The key will be the `Player.id` of the player adding items to the BST via a push operation. Player ids are unique by construction, so there should be no duplicates in the BST. Besides its key, each node has a LIFO collection of items (for which you developed an implementation for the previous question).
   - implement the method `push(item, agent)`. This method adds an item to the tree. It traverses the tree to find the node with key equal to `player.id` and adds the item to its collection. If such node does not exist, it adds it to the tree preserving the BST invariants, and stores the item in its collection of items.
   - implement the method `Optional<E> pop()`. This method traverses the tree to find the node with the largest key and returns an optional containing one of the items from the node's collection (the returned item is removed from the collection). If the node's collection becomes empty, the node has to be removed from the tree (not only logically flagged as removed), preserving its BST invariants. If the tree is empty, the method returns `Optional.empty()`.
   - implement the method `size()`, which returns the total number of items stored in the BST.

   Hint: the highest priority node in a BST is always the rightmost one. Therefore, it can only be either... Keep it in mind for both this question and Q4.

   [**20 marks**]

3. **Exchange and players.**

   You are required to implement the class `exchange.ExchangeImpl` and the players in the package `players`. All the players inherit from `domain.Player`, which takes care of

assigning them a unique `id` upon creation. `Player` also extends `Thread` and defines its run method: until interrupted, a player repeatedly `doAction()` and then sleeps for a random time. For each player, you only need to implement `doAction()` (possibly adding fields and private methods as you see fit). The tests in `exchange.ExchangeTest` will help you check your implementation and provide additional hints.

Your tasks are:

- implement the methods of `ExchangeImpl`. The method names are self-explanatory. The exchange keeps a *stock* of each product (element or composite chemical) and pushes and pops from those stocks as needed.
- implement the method `doAction()` of `players.CarbonSupplier` and `players.HydrogenSupplier`. Element suppliers just sell to the market a new batch of their basic chemical element every time `doAction` is called. The classes for the chemical materials are specified in `chemicals`. Each supplier sells elementary chemicals of the type corresponding to its name, e.g., `XSupplier` sells `X`.
- implement the method `doAction()` of `players.MethaneMixer`. At each invocation of `doAction`, this mixer repeatedly invokes *buy* operations on the exchange until it secures one unit of carbon and two units of hydrogen. Then, it sells a new unit of `Methane`, constructed with the set of basic elements it secured.
- implement the method `doAction()` of `Chemist`. At each invocation, the consumer tries to buy a unit of `Methane` from the market. If it gets one (i.e., the `Optional`) is not empty, it `think()` (method inherited from `Player`), otherwise `doAction()` returns immediately.

[8 marks]

4. **Thread-safe stocks.**

   For this question, you are required to make the `StockImpl` implementation (all three methods) you wrote for Q2 thread-safe. A coarse-grained access control is worth up to 5 marks. Any correct fine-grained access control is worth up to 10 marks.

   Depending on your strategy to make `StockImpl` thread-safe, you may or may not need to make also `LIFOImpl` thread-safe, i.e., if a node of the BST-based `StockImpl` can only be accessed by a single thread there is probably no need to make also `LIFOImpl` thread-safe. However, if for your design you need to make also `LIFOImpl` thread-safe, use a coarse-grained strategy for `LIFOImpl`.

   Before moving to the fine-grained implementation, make sure to save a copy of your coarse-grained one (e.g., in a class `StockImplCoarseGrained`), or to write clearly in a comment at the beginning of the class how you would implement such coarse-grained strategy.

   The tests in `datastructures.StockStressMonkey` may help you find errors in your solution (unfortunately, passing the tests does not imply the solution is correct).

   Hint1: the size of the stock is just an integer that gets incremented or decremented by push and pop operations...

   Hint2: as noted in Q2, items are removed always from the BST node with the largest id...

   [15 marks]

Total for Section B: 50 marks