**Log Aggregator Solution**

This is the solution of the Log Aggregator problem

While solving such problems, it is a good idea to first identify all the information that needs to be stored or cached and the best data structure to store it. To finalize the data structure, identify what operations you will be performing on this data and the most efficient data structures for these operations.

In our case, we can see get methods to fetch logIds, by machineId and serviceId each. While we can calculate this mapping when the respective functions are called, it would be much more efficient to pre-calculate these mappings so that the implementations of getLogsFromMachine() and getLogsOfService() can be optimized.

This means we need a mapping for machineId -> logId (which is stored in logsForMachine in the solution below) and serviceId -> logId (which is stored in logsForService in the solution below) which can be returned directly without any further pre-computation.

When we talk about mappings, hashmaps or dictionaries are our best bet due to efficient key-based lookups. Each language has a data structure providing similar functionality. For example, in Java, we can use a HashMap, in Python, we can use a dictionary, etc.

Now, we already have mappings for logs against machineId and serviceId. Similarly, we will also have a mapping for logId -> message (stored in logs variable in the solution below) to store the message strings for each logId.

So far, our init method or constructor looks something like this -

Now lets start with the function definitions.

For the function pushLog(), we have logId, machineId, serviceId and message as the parameters. This makes our pushLog() function the best place to create a mapping for logId -> message, machineId -> logId and serviceId -> logId.

At this point, our pushLog() function looks like this:

The getLogsFromMachine() and getLogsFromService() functions simply return the logIds saved against machineId and serviceId in our dictionaries or HashMaps.

For the search() method, as we can see from the requirements, we need to return all logs for a service that is a perfect match for the substring. To do this we can simply fetch all logIds against a serviceId, then run a substring search for all the messages against these logIds. Return the messages which are a perfect match.

Now our search() function would look something like this