1. Testing a Struct interface

Given the following interface contract for a stack data structure, you are tasked with writing a comprehensive set of unit tests to test all the behaviors defined by the interface specification.

You may use a standard testing framework for the language you prefer.

Note: depending on the chosen language, the specification may have slightly different types/keywords than the ones used by the language (null vs None, raise vs throw etc).

A solution will be considered complete if it tests all the defined implicit and explicit behaviors. Please provide a readme explaining the solution, the setup and any used dependencies.

Interface Struct

Methods:

size() -> integer

Returns an integer representing the total number of items in the stack.

push(element)

Pushes the element onto the top of the stack.
Throws a custom NullElementException if the supplied element is null.

pop() -> element

Removes the top element from the stack and returns its value.
Throws a custom EmptyStackException if the stack is empty when this method is called.

peek() -> element

Retrieves the top element from the stack without removing it, and returns its value.
Throws a custom EmptyStackException if the stack is empty when this method is called.

empty() -> boolean

Tests whether the stack is empty.

2. Data Store Library

For this task, you are tasked with designing and implementing a library that can be used to store and retrieve arbitrary data in multiple formats & destinations.

This library will be used by 3rd party developers, and it should expose a simple and structured API.

A record will be defined as a simple data structure where every key maps to a primitive value (fields values cannot be objects, arrays etc).

Given this definition of a record, the library must support the following operations:

- Record inserts & batch inserts
- Record query/retrieval
- Query filters (equality operations only), limit & offset
- Update and delete operations

You may use the storage format and destination of your choice, but the library must be expandable.
Support for other storage formats & destinations should not require significant code changes in order to be added (Example formats: json, xml, bytes etc. Example destinations: local drive, ftp, cloud storage etc).
Every supported storage format may be combined with any of the supported storage destinations.

The solution should support at least one format and one destination, and should also contain mock implementations of an alternative format and an alternative destination. (Example: data is stored in json format on the local drive. Mock: data stored as xml and uploaded to S3 - the mock methods do not have to be implemented)

The end user should be able to configure the library according to their needs.

The exposed API should be thread-safe, and tests must be present alongside the submitted solution.

The task will be scored based on the following criteria:

- Overall design and usability
- Simplicity of use
- Correctness
- Supported features
- Quality of code
- Quality of tests

- Documentation

This is primarily a design task - the performance of the chosen storage type does not matter.

Sample usage instructions should also be provided for the exposed interfaces.