This is the retake assignment, it is similar, but it is not the same!
Delivery form added.
(minor grammar corrections)

FILE_VAULT.SH (retake):
Management and Automation!

Assignment Description

Intro and background info.

Bash, short for "Bourne Again Shell", is a challenging scripting language and one of the widely-used command-line shell and scripting languages in the world of DevOps (Development and Operations) and automated environment setup. DevOps is a set of practices that combine software development (Dev) and IT operations (Ops) to automate and streamline the software delivery and infrastructure management processes. Some examples of using bash in DevOps and Automatic Environment Setup:

1. **Integration**: Bash can be easily integrated into various DevOps tools and platforms. It can call APIs, execute commands, and parse data from tools like version control systems, continuous integration servers, and container orchestration platforms, enabling seamless integration into the DevOps toolchain.
2. **Infrastructure as Code (IaC)**: Bash is often used to write scripts that define infrastructure as code. IaC enables the creation and management of infrastructure resources (e.g., servers, databases, networking) using code. Bash scripts can interact with cloud providers' APIs or configuration management tools, making it easier to

Bash often plays a significant role in operating systems in several ways. Here are some examples:

1. **Scripting and Automation:** In the context of operating systems, Bash is a powerful scripting language. It helps to write Bash scripts to automate various tasks, such as file management, process control, and system administration. Bash scripts can be used to automate repetitive tasks and perform system-wide operations.
2. **Debugging and Troubleshooting:** Bash can be used for debugging and troubleshooting purposes. Bash scripts can assist in diagnosing issues, checking logs, and performing system health checks, which are vital skills for system administrators and developers working on operating systems.
3. **File System Manipulation:** Understanding how the file system works is a central component of operating system studies. Bash provides commands for navigating, creating, modifying, and deleting files and directories, making it a valuable tool for teaching file system concepts.

Your goal.

It is very common to work with protected archives and to make a personalised backup system connected to your available resources. The goal of this assignment is

for you to implement a *Bash* script that creates a simplified “file vault”. The file vault is a “safe (enough)” compressed and encrypted storage system, in this case, based on a queue data structure, where the files are *enqueued* and *dequeued*.

This utils grants both the functionality of a secure backup and easy archive.

The archive must never be empty, so if there is nothing inside, there will be generated a text file named “nothing.txt”, containing “almost nothing” as a sequence of characters.

The archive can work with entire directories, single files and/or a combination of both. Ex.: directories containing directories that contain multiple files.

The implementation

Validation checks must be everywhere inside the Bash script, even where it is not explicitly required. Here are some functionality to review:

- **User Input and Parameter Values Validation**: validate user and non-user-provided arguments everywhere to ensure they are valid commands and options. Provide feedback if the arguments are not valid or recognized.
    1. **Providing meaningful feedback is essential**.
    2. **Impeding the propagation of an unwanted character is essential**.
    3. **Intercepting errors on commands is one of the most common problems**. Take care of isolating every command that is possibly failing.
- **NO FILE MANIPULATION OUTSIDE THE INDICATED FILES!**

install() function is responsible for the following tasks:

- **Generic Installation**: The script will generate a setup file (“*vault.conf*”) that contains whatever configuration information is needed for the script to run. Namely, the file should contain the location of the directory where the *vault* is installed. If you need other temporary configurations, this is where to find them. The file will always be placed in the “*home*” directory of the current user, and it will be named “*vault.conf*”.
    1. **It should install the structure of the vault**: the script should address the default installation location, in the home, in a directory named “vault/”, it will generate the nothing file (see section dedicated) and proceed with the compression/encryption to generate the first empty archive.
    2. **Generate accessible functions**: the script should generate functions to be kept in memory. All the exposed functions will allow you to run them from any directory where they are called. If the shell is closed the functions should disappear. The functions generated are: “assignment”, “aqueue”, “apqueue”,”adequeue”, and “apdequeue” (see usage)
- **Specific installation**: if a valid non-existing directory name is passed it should do the same as before but in the location that has been passed, be it an absolute path or relative to the current working dir.

- **If there is a mistake**: such as no access, the directory to be created is already present, or other errors are present, the script should always communicate with the user the kind of problem generated, it is ok to keep it generic if there are too many options to deal with.

setup() function is responsible for the following tasks:

- **Relocating the existing vault to a new location**: The *setup()* migrates the existing vault to the passed new directory, exactly as the install but with an explicit parameter, which will be the location of the new vault. The configuration file will remain the same.

handle_error() *function is responsible for the following tasks:*

- **Error Handling**: The code provides error handling using the `handle_error()` function. If errors occur during any step of the process, it prints error messages and, if specified, executes commands to handle the error (e.g., installing missing dependencies).
- **Reverse incomplete steps**: there are some sets of steps that need to be executed together. If a set cannot be completed for some reason or throws an error, the `rollback` function should clean and reverse to the starting point of the set.
- **Returning adequate return value**.

dequeue() *function is responsible for the following tasks:*

- **Extraction**: a dequeue extracts the first enqueued-in element of the queue. The extracted element should be placed in the current directory, if that is not possible it should provide errors.
- **If the vault is encrypted**: it should pass the password, or fail.

enqueue() *function is responsible for the following tasks:*

- **Archive**: The enqueue item should be put in the archive from the current directory (or absolute path), if that is not possible it should provide errors. The enqueue will put the item, at the bottom of the queue.
- **If the vault is encrypted or a password is provided**: it should use the password to encrypt the whole archive or fail in case it is encrypted and the password is not the correct one.

uninstall() function is responsible for the following tasks:

- **Dependency, file, and directory structure:** Every dependency, folder, and file that was installed or created in the installation needs to be removed. There should be no trace that the script was ever run once. This should not delete the script itself.

Usage of *vault.conf*:

- **Configuration File Usage**: The code reads configuration settings from a file named " *vault.conf*", it is always located in the home directory of the current user.

Usage of *"vault/"*:

- **The vault directory and files**: the directory of the vault is by default located in the home and named "vault/", it can be set up in a different place via the setup command. The configuration will tell the script where to locate it.
    1. **Content of the vault (the archive)**: the directory must contain an archive "*archive.vault*" and the record of what is inside.
        1. **If the vault archive is empty:** you should always have the "nothing.txt" in.
        2. **If the archive is not empty:** you should have a collection of archived elements.
    2. **You must keep a queue of elements inserted into the queue to be able to *enqueue/dequeue* items.** The file is named **"queue.vault".**

It is **mandatory** to keep the existing template code as given. However, you can add additional functions and code to complete the application (all in the same single file). For example, the template assumes that **error handling is always done** using the generic function `handle_error()`. If something goes wrong halfway through a set of steps the function should roll back to reverse partial steps.

The empty file (*nothing.txt*):

-----> Begin the content of *nothing.txt* <-----

```
almost nothing
```

-----> End of the content of *nothing.txt* <-----

Usage.

Your implementation needs to be tested on Linux Debian 11 (same distribution as last year).

To run and test your Bash script implementation, change the path in the command line to the folder where the script is located. The following commands and options are expected to be used ("# " represents the bash prompt):

To load the functions and install the script we will execute:

```
# source ./assignment.sh
```

Or

```
# . ./assignment.sh
```

There is no need for parameters, the default behaviour on load should be “install”

To uninstall the script

```
# ./assignment.sh --uninstall
```

To uninstall the script with the bash function

```
# assignment --uninstall
```

To change the setup of the script without uninstalling it and move the current one to a new location

```
# ./assignment.sh --setup newlocationdir/
```

To set the script with the bash function

```
# assignment --setup newlocationdir/
```

To enqueue an item into the vault

```
# ./assignment.sh enqueue existingdir/

# ./assignment.sh enqueue existingdir

# ./assignment.sh enqueue existingfile.whocares
```

To queue an item with the bash function

```
# aqueue item
```

To queue an item into the vault with a password

```
# ./assignment.sh enqueue -p existingdir/
```

To queue an item with the bash function and a password

```
# apqueue item
```

To dequeue an item from the vault

```
# ./assignment.sh dequeue
```

To dequeue an item from the vault with the bash function

```
# adequeue
```

To dequeue an item from the vault with a password

```
# ./assignment.sh dequeue -p
```

THIS IS THE LAST RETAKE!

To dequeue an item from the vault with a password with the bash function

```
# apdequeue
```

Note: In the exposed functions, we use conventions created for this assignment, the initial “a” stands for assignment, the first “p”, if there are two, refers to the presence of a password, while the rest of the function name is the name of the command.

Limitations (let’s make it easier).

- The archive will never contain files with the same name at the same level in the testing.
- The archive might contain files with the same name but in different items (like in two different directories that contain the same files).
- The archive might contain directories with the same name.
- The archive will never get names of files/directories that are not valid.
- If a password is already set up, it is for ALL the archive and not just one entry.
- You can write in your config file if there is a password or not, or you can mark your queue file with the fact that it is or is not encrypted. The important is that you keep it coherent.
- You should never handle passwords directly. For your information, there will not be passwords with non-printable characters.
- For simplification purposes, an archive that is encrypted can’t become unencrypted. An unencrypted archive can become encrypted.
    - To unencrypt an archive, it must be deleted and reinstalled.
- There will not be test cases of only empty directories enqueue into the archive.
- There will not be used files “nothing.txt” in the testing.
- There is no need to install dependency in this assignment (but if one is missing, it should notified to the user).
- User input should be limited to the command line interface, there will be no need to intercept input after the script has been invoked.
- The name of the script is not relevant and will be modified by the system during the delivery, in this document it is a placeholder to stay generic on purpose as any valid filename should be ok.
- If you do not remember what a queue is, please refer to the reference methods in the course slides!

Useful commands to use:

tar (see man tar): to compress directories and single files. Common parameters for tar are “*czvpf*” (you can look up all the man options). Typically, tars can preserve permissions, and file types, and compress portions of disks (even images) with **gzip**. From here the usual extension “filename.tar.**gz**”.

gpg (see man gpg): to encrypt any content. A common set of parameters to create an encrypted file is *“--symmetric --cipher-algo aes256 -o specialarchiveofmine.tar.gz”*. The option “*-d*” is often used to decrypt a file.

Note that directories might or might not end with “/”.

The commands can be combined easily in one or used in steps. We encourage you to use these and not look up others. They are both present in the reference distribution, but you should always check if they are present or not and in case provide for the installation request.

Tip: test your application extensively before submission to handle all types of errors. For example, run every command mentioned above but do not provide all arguments and handle each error you encounter. Remove required files during testing and handle printed errors accordingly. This will make your implementation robust.

Grading.

To pass the requirements below need to be satisfied (VLD):

- The implementation follows the provided structure in the template.
- Each mentioned command above works as expected.
- No errors during the execution of the bash script.
- Comments are added to explain the provided logic.
- Use proper access level to commands, files, and folders that are strictly needed e.g. u+x

If any of the following problems are present, the script will be regarded as insufficient with no further checks (NVL):

- The assignment does not reflect the updates of the current version.
- The file is corrupted/incomplete/gives errors on the first run (e.g.: end-line Windows corruption→”\g”, “^M”).
- The implementation does not follow the provided structure in the template.
- You are manipulating, or altering in any way, any file that does not concern the assignment (ex.: user files that are not part of the command executed, configuration of the shell, unless explicitly requested, etc…).
- One of the commands in the description is not working as expected.
- The added code is not organized in functions. Or added logic is not readable and/or not commented on.
- Any fatal error (e.g.: wrongly implemented requirement, missed implementation, etc.). The sum of less significant errors might also cause failing results if the requirements are not met in that case.
- Presence of unhandled errors/checks due to poor testing or unexpected environment issues (missing access authority, missing a file, input errors generated by users or by data read).
- Plagiarism (AI-generated code is plagiarism too, any usage of **automated generation of code is forbidden**).
- Using the root (e.g.: *su* or *sudo*) permission in combination with a command that does not require it. Similarly, the use of higher access levels than strictly needed for files and folders for users and groups when created.

- The delivery does not contain the students' names and numbers **of all the participants**.
- Double delivery (2 students submit the same assignment).

If no delivery is provided or the delivery is late, the grade will be ND.

The script will be tested in the given environment and might also be tested in the delivery system for plagiarism/nonsense.

The first run of the script will likely be tested directly with the interpreter as follows:
`# bash yourscript.whatever "parameters"`

Submission

It is allowed to deliver in groups of *up to 2* students. Each student will be responsible for the whole assignment.
The reviewers will always have the possibility of an **oral check to verify the integrity and/or originality** of the delivery (also indicated in the course description).

The instructions for the submission will be added here when they are ready.

https://forms.office.com/e/Enj3LVB1h5

Deadline

The deadline for this upload is the **17th of January at 11:30 P.M**.