Workplace Technology & Skills (CS3306)
Assignment 3

Version Control With Git

Assignment Specifications Subject to Change

*(Version V 1.0; Due: March 1; Marks: 50)**

1 Introduction

This assignment is about local version control with Git. After this assignment you should know how to (1) create your local Git repository, (2) make changes and commit your changes to the repository, (3) define branches and carry out local develoment in these branches, (4) merge the branches, (5) use `git bisect` to locate the source of bugs in your branches, ...

2 Assignment Details

2.1 Introduction

This project in the assignment is a `C` project with one library and three branches. The functions which are defined in the library's API are: `create()`, `push()`, and `pop()`. There are 3 virtual files: `list.h`, `list.c`, and `main.c`. The project has the following development branches.

`api` Defines the API/header file, `list.h`, of the list library.

`dev` Implements the API, `list.c`, of the list library.

`master` Uses the functions in list library in `main.c`.

However, for simplicity you won't not do actual code development but you will simulate the development by writing details about the development to a file, `recorder`, which is tracked by all branches. Besides `recorder`, the repository doesn't track any other file. You should name your project repository directory `repo`.

2.2 Development Scenario

Your project will *simulate* the following development scenario. Exact detail about the scenario and the order of the development tasks is provided in Section 2.4. The creation of the branches and the merging of branches is not explained here.

*This assignment is for CS3306 students. You may only submit work which is your own. You may not share this assignment or solutions with others. This means, for example, that you may not post this assignment and/or post solutions on `GitHub`, on social media, or other public of private media.

- `main` branch: compile `main.c` with empty `main()`;
- `api` branch: create header file `list.h` which defines list type and functions `create()` and `push()`;
- `main` branch: compile `main.c` with calls to `create()` and `push()`;
- `dev` branch: implement `create()` and `push()` in `list.c`;
- `main` branch: compile, link, and execute main with calls to `create()` and `push()`;
- `api` branch: define `pop()` in `list.h`;
- `dev` branch: implement `pop()` in `list.c`;
- `main` branch: compile, link, and execute main with calls to `create()`, `push()`, and `pop()`.
- `dev` branch: introduces a bug;
- `dev` branch: more development;
- `dev` branch: fixes bug;
- `main` branch: compile, link, and execute main with calls to `create()`, `push()`, `pop()`.

2.3 Driver Script

For ease of development and project submission, you must implement a Bourne shell script called `creator.sh` (`creator` for short), which contains the commands which are needed for the (simulated) project development and the Git version control commands. The `creator` script should be defined in the top-level of your assignment's submission directory (in the parent of your repository directory). As shown during one of the lectures, using a script simplifies paracticing, removing the entire project, and creating the project from scratch (by running the script). I suggest you start the script as follows.

```
#!/bin/sh

# ====================================================
# YOUR NAME: YOUR ID
# ====================================================

REPO=repo        # top-level repository directory

# ====================================================
rm -rf ${REPO}   # remove repository if it exists
mkdir ${REPO}    # create reposity directory
cd ${REPO}       # move into the repository's directory
git init --quiet # initialise the repository
# ====================================================
```

2.4 Detailed Development Scenario

The following are the remaining tasks in the project's development. For each task you should add a line/a few lines to the `creator` script.

1. Create the `recorder` file, start tracking `recorder`, and create the first commit node.
   Please make sure that the commit message is descriptive and identifies the branch which is responsible for creating the commit. You should do the same for further commits.
2. For further parts of the assignment, please remember the SHA1 hash of the first commit and assign it to the `SHA1` variable:

```
SHA1=‘git log --oneline | sed -n -e 'ls/ .*//p'‘
```

3. Create the `api` branch, **introduce `create()` and `push()` in `api` branch**, and create a new commit node. You should simulate the function's introduction with `echo`: `echo "introduce create() and push() in api branch" > recorder`, just before you add `recorder` to the stage and create a new commit.
4. In the `master` branch **compile empty `main()`**: `echo "compile empty main() in main branch" > recorder`, stage `recorder` and commit.

   The remaining tasks won't explanation the simulation mechanism (redirecting to `recorder`).
5. Create a sub-branch of the `api` branch which is named `dev`.
6. In the `dev` branch **implement `create()` and `push()`**.
7. Using `git merge`, merge the `api` and the `master` branch. We shall ignore merge conflicts. You can ignore merge conflicts by adding the `-s ours` flag to the `git merge` command.
8. In the `master` branch: **compile `main()` with `create()` and `push()`**.
9. Merge the `dev` branch and the `master` branch.
10. In the `master` branch: **run `main()` with `create()` and `push()`**.
11. In the `api` branch **introduce `pop()`**.
12. Merge the `api` branch and the `dev` branch.
13. In the `dev` branch: **implement `pop()`**.
14. Merge the `dev` branch and the `master` branch.
15. In the `master` branch: **run `main()` with `push()` and `pop()`**.
16. In the `dev` branch: simulate a bug in the development. As a result of the bug the function `push()` returns an empty list. Simply add the line **BUG: `push()` returns empty list** to the end of the `recorder` file and create a new commit node.
17. In the `dev` branch: **more development** (first time).
18. In the `dev` branch: **more development** (second time).
19. In the `dev` branch: **more development** (third time).
20. In the `dev` branch: use `git bisect` to locate the commit which introduced the bug.[1] The variable `SHA1` contains the `SHA1` hash of the commit node. Please add the `git bisect` commands to your `creator` script.
21. In the `dev` branch: eliminate the bug. Simply delete the line with `BUG` from the `recorder` file using `sed` or `grep -v`. Add the line **FIXED BUG in dev branch** to the `recorder` file and commit.
22. Merge the `dev` branch and the `master` branch.
23. In the `master` branch: **run main with `create()`, `push()`, and `pop()`**.
24. Add the command `git log --oneline --graph` at the end of your `recorder` file.

When you are done, your commit graph should be similar to the one which is depicted in Figure 1.

2.5 Remaining Comments

Comment. Your `creator` script must carry out the tasks in the same order as in Section 2.4. You will lose marks for tasks which are omitted and for tasks which are carried out in the wrong order.

Comment. You should use `git merge` for the merging, not `git rebase`.

Comment. Git is very chatty. Please add the `--quiet` flag to the commands which allow the flag. (The only commands which don't allow the flag are the `git bisect` commands.)

Comment. The lines in your `creator` script should not exceed 74 characters.

[1] You can recognise the bug by grepping for `BUG` in the `recorder` file. For real-life scenarios, you may need a unit test or some other technique.

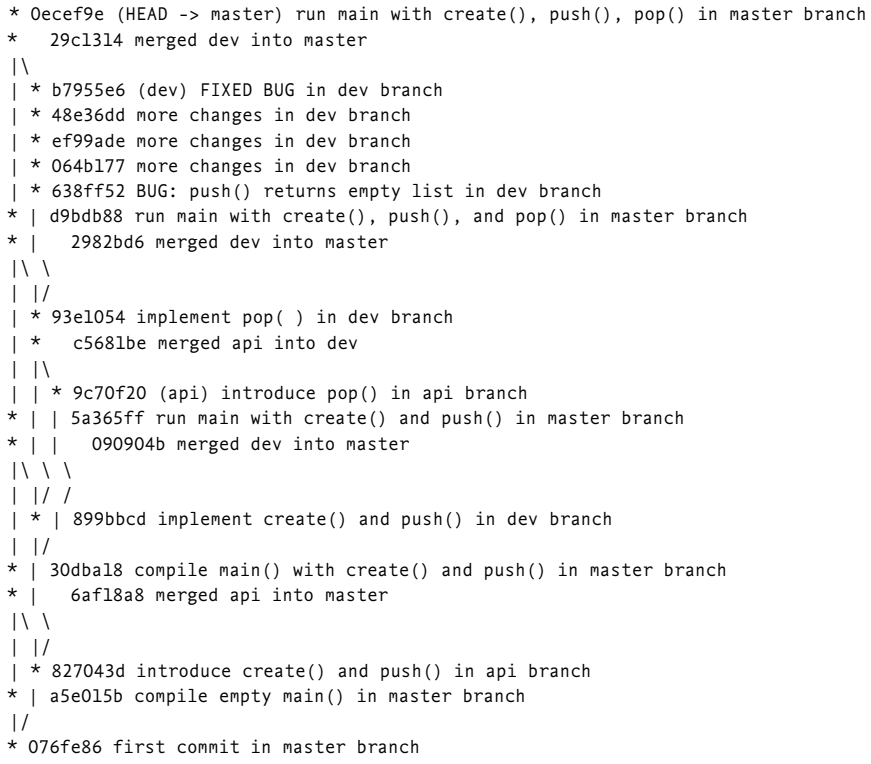

Figure 1: commit graph of project.

Comment. Your commit messages should be short and should end in the words "in *X* branch," where *X* is the branch which created the commit node.

Comment. Please add a short comment in your `creator` script which explains what you are doing. The comments should be of the form "`# (X)) comment`," where `X` is the number of the corresponding task in the list of tasks which are listed in Section 2.4. Additional comments should not be numbered.

Comment. You should not use functions or (shell or `git`) macros in your script.

Comment. You must submit your `creator` script.

Submission Details

- Upload `creator.sh` to the CS3306 `Canvas` area as a single file. before 23.55pm, March 1, 2022.
- Marks shall be deducted for (1) poor choice of variable names, (2) poor layout, and (3) lack of comments and comments which are too generic.
- No marks shall be awarded for scripts which have syntax errors.