THE UNIVERSITY OF SYDNEY

COMP2017/COMP9017 Assignment 2

Assignment due: 3 May 2020, 11:59 PM AEST (Week 9 Sunday)
This assignment is worth 10 % of your final assessment

1 Introduction

Version control systems are crucial in any software project. In this assignment, you will design and implement the storage method, as well as some functions for Simple Version Control (SVC), a (very) simplified system derived from the Git version control system.

2 Description of SVC

Projects that are to be placed under SVC must be initialised with `svc_init`. This allows the system to create necessary data structures containing information about the state of the project.

Projects are comprised of files, and SVC computes a hash of each file to figure out if a change has occurred. Only files which are explicitly added to the version control system are tracked.

Commits are like a snapshot in time of the state of the project. In SVC, they contain details about which files have been added, removed, or modified, and sufficient information to restore a file to this state. The currently active commit is often referred to as the `HEAD`. Each commit has a "commit id" which uniquely identifies a given commit. Commit ids in SVC are represented in hexadecimal numbers and are exactly 6 characters long.

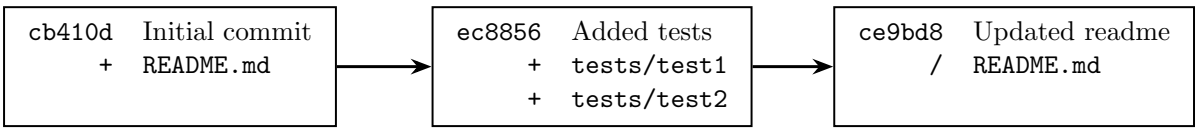
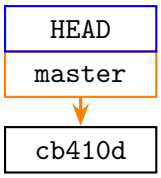


Figure 2.1: Examples of commits

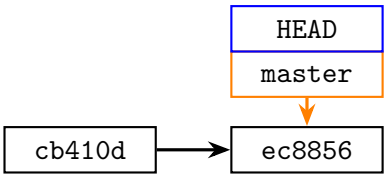
Branches are useful to keep working copies of the project while working on new features or fixing bugs. When a project is created, a `master` branch is created by default. For example, this branch may be used to keep release versions of projects, and a branch `features/something_special` might contain additional code with a new feature that is not yet complete and would not be suitable for releasing.

A branch can be "checked out" to indicate that we want to switch our project to a specific branch and work from there. For example, after we create the `features/something_special` branch, it should be checked out before working on that feature.

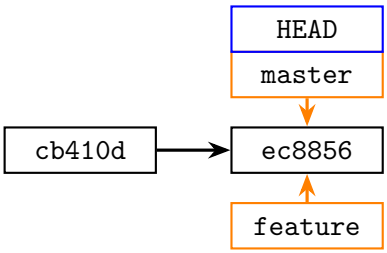
Merging is the process of integrating changes from one branch into another. Once a new feature is complete, or a bug is fixed and testing is complete, we may wish to merge these changes back into the `master` branch to release. Note that merges can happen between any two branches, not necessarily with the `master` branch.



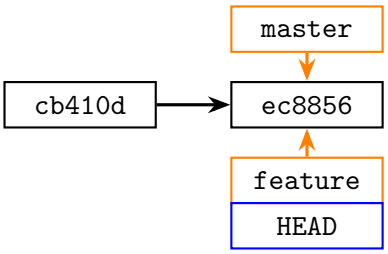
(a) After the initial commit



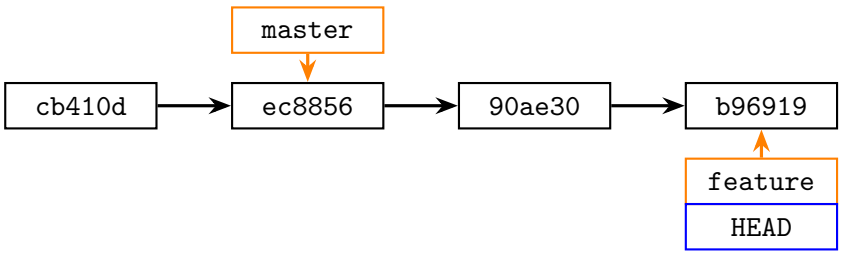
(b) After another commit to the `master` branch



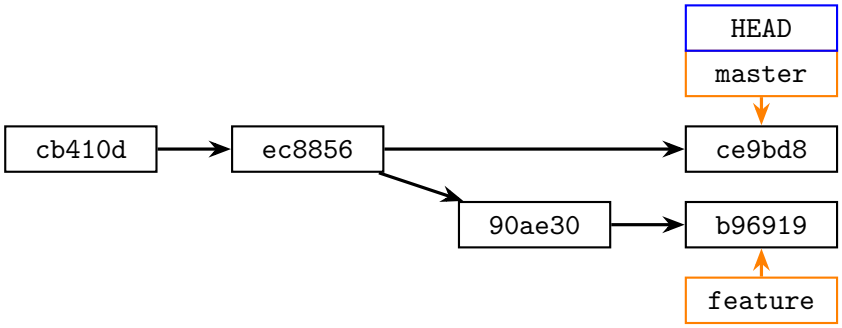
(c) New branch created



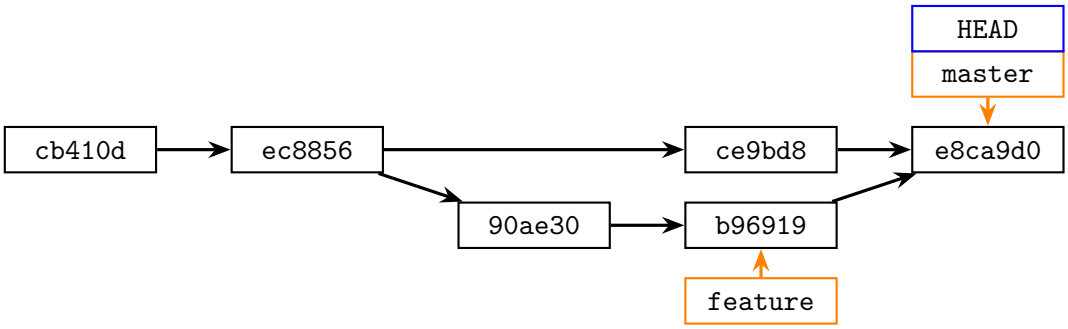
(d) After checking out the `feature` branch



(e) Two commits to the `feature` branch



(f) `master` branch checked out and one commit made



(g) `feature` branch merged into `master`

Figure 2.2: Examples of branches and merging

Sometimes, we may want to reset to a specific commit. In this case, the files are reverted to the state they were in at that commit, and any new commits continue from that commit. This may result in some commits being detached from the rest of the SVC system. For example, if the `master` branch is reset to `ec8856` and a new commit is made, `ce9bd8` would not be reachable.

3 Functions to Implement

```
void *svc_init();
```

This function will only be called once, before any other functions are called. In this function, you may choose to perform preparation steps. If you require some data structure to help with other operations, return a pointer to the area of memory containing this structure. In all further function calls, this will be passed in as `void *helper`.

```
void cleanup(void *helper);
```

This function will only be called once, after all other functions have been called. Ensure all dynamically allocated memory has been freed when this function returns.

```
int hash_file(void *helper, char *file_path);
```

Given the path to a file, compute and return the hash value using the algorithm described in Section 3.1. If `file_path` is `NULL`, return `-1`. If no file exists at the given path, return `-2`. This function should work even for files that are not being tracked.

```
char *svc_commit(void *helper, char *message);
```

Create a commit with the message given, and all changes to the files being tracked. This should return the commit id, which can be calculated by implementing the algorithm described in Section 3.2. If there are no changes since the last commit, or `message` is `NULL`, return `NULL`.

```
void *get_commit(void *helper, char *commit_id);
```

Given a `commit_id`, return a pointer to the area of memory you stored this commit. If a commit with the given id does not exist, or `commit_id` is `NULL`, this function should return `NULL`. Note only the `NULL` return values for this function will be tested. The commit you return here will be passed to some of the other functions to implement.

```
char **get_prev_commits(void *helper, void *commit, int *n_prev);
```

Given a pointer to a commit, return a dynamically allocated array. Each element in the array should point to the id of a direct parent commit. The number of direct parent commits should be stored in the area of memory pointed to by `n_prev`. If `n_prev` is `NULL`, return `NULL`. If `commit` is `NULL`, or it is the very first commit, this function should set the contents of `n_prev` to `0` and return `NULL`. Note: only the allocated array will be freed for you by the tester.

```
void print_commit(void *helper, char *commit_id);
```

Given a `commit_id`, print the details of the commit as detailed below. If no commit with this id exists, or `commit_id` is `NULL`, you should print `Invalid commit id`

For a valid commit, you should print the commit id, the branch it was committed to, a list of the added, removed and changed files in any order, and a list of the tracked files at this point in commit history on that branch along with their hash values, also in any order. Hashes are left padded with spaces to be exactly 10 characters wide.

```
commit id [branch name]: commit message
    + added file(s)
    - removed file(s)
    / changed file(s) [previous hash --> new hash]

    Tracked files (number of tracked files):
    [hash] file name
```

```
int svc_branch(void *helper, char *branch_name);
```

Create a new branch with the given name. In this SVC, valid branch names are restricted to those that contain only alphanumeric characters, underscores, slashes and dashes: `a-z, A-Z, 0-9, _, /, -`. If the given branch name is invalid or `NULL`, return `-1`. If the branch name already exists, return `-2`. If there are uncommitted changes, return `-3`. If the branching is successful, return `0`. Note: creating a branch does not check it out.

```
int svc_checkout(void *helper, char *branch_name);
```

Make this branch the active one. If `branch_name` is `NULL` or no such branch exists, return `-1`. If there are uncommitted changes, return `-2` and do not make this the active branch. Otherwise, return `0` and make it the active branch. Note in SVC, the branch is not created if it does not exist.

```
char **list_branches(void *helper, int *n_branches);
```

Print all the branches in the order they were created. In addition, return a dynamically allocated array of the branch names in the same order, and store the number of branches in the memory area pointed to by `n_branches`. If `n_branches` is `NULL`, return `NULL` and do not print anything. Note: only the allocated array will be freed for you by the tester.

```
int svc_add(void *helper, char *file_name);
```

This is a notification that a file at the path `file_name` should be added to version control. If `file_name` is `NULL`, return `-1` and do not add it to version control. If a file with this name is already being tracked in the current branch, return `-2`. If this file does not exist, return `-3`. Otherwise, add the file to the SVC system and return the file's hash value.

```
int svc_rm(void *helper, char *file_name);
```

This is a notification that a file at the path `file_name` should be removed from the version control system. If `file_name` is `NULL`, return `-1`. If the file with the given name is not being tracked in the current branch, return `-2`. Otherwise, remove the file from SVC and return its last known hash value (from adding or committing).

```
int svc_reset(void *helper, char *commit_id);
```

Reset the current branch to the commit with the id given, discarding any uncommitted changes. If `commit_id` is `NULL`, return `-1`. If no commit with the given id exists, return `-2`. It is guaranteed that if a commit with this id exists, there will be one simple path from the HEAD of the current branch. That is, all commits from HEAD to the commit will have exactly one previous commit. Reset the branch to this commit and return 0. Note that this function means that some commits may be detached from the rest of the SVC system.

```
char *svc_merge(void *helper, char *branch_name, resolution *
    resolutions, int n_resolutions);
```

This function will be called to merge the branch with the name `branch_name` into the current branch. If `branch_name` is `NULL`, print `Invalid branch name` and return `NULL`. If no such branch exists, print `Branch not found` and return `NULL`. If the given name is the currently checked out branch, print `Cannot merge a branch with itself` and return `NULL`. If there are uncommitted changes, print `Changes must be committed` and return `NULL`. In all other cases, the merge procedure begins. Note that the way branches are merged in SVC is different to Git.

To merge two branches together, all tracked files in both branches are used. If there are conflicting files, it will appear in the `resolutions` array. Each `resolution` struct contains the name of the conflicting file, and a path to a resolution file. This file contains the contents that the file should contain after the merge. However, if the path given is `NULL`, the file should be deleted.

A commit with the message `Merged branch [branch_name]` replacing `[branch_name]` with `branch_name` is created with the necessary changes for the current branch to reflect changes made in the other branch. The previous commits order should be the current branch's `HEAD` and then the other branch's `HEAD`. The function should then print the message `Merge successful` and return the new commit id.

3.1 File Hash Algorithm

Below is the pseudocode to determine the hash value of a file. Note: this is **not** the same algorithm used in real world version control systems.

```
function file_hash(file_path):
    file_contents = read(file_path)
    file_length = num_bytes(file_contents)
    hash = 0
    for unsigned byte in file_path:
        hash = (hash + byte) % 1000
    for unsigned byte in file_contents:
        hash = (hash + byte) % 2000000000
    return hash
```

3.2 Commit ID Algorithm

Below is the pseudocode to determine the commit id of a file. Note: this is **not** the same algorithm used in real world version control systems.

```
function get_commit_id(commit):
    id = 0
    for unsigned byte in commit.message:
        id = (id + byte) % 1000
    for change in commit.changes in increasing alphabetical order of file_name:
        if change is addition, id = id + 376591
        if change is deletion, id = id + 85973
        if change is modification, id = id + 9573681
        for unsigned byte in change.file_name:
            id = (id * (byte % 37)) % 15485863 + 1
    return id as hexadecimal string
```

4 Examples

4.1 File Hashing

File Name	File Contents	Hash Value
`Tests/diff.txt`	Empty file (0 bytes)	385
`Tests/diff.txt`	`This is some text in a file\n`	2832
`sample.txt`	`Hello, world\n`	1178
`sample.txt`	`Hello, world!\n`	1211
`Tests/test1.in`	`5 3 2\n`	564
`hello.py`	`print("Hello")\n`	2027

Taking the last example, `hello.py` corresponds to ASCII values [104, 101, 108, 108, 111, 46, 112, 121]. Following the algorithm gives a hash of 811 % 1000 = 811. The contents of the file correspond to [112, 114, 105, 110, 116, 40, 34, 72, 101, 108, 108, 111, 34, 41, 10] giving (811 + 1216) % 2000000000 = 2027.

4.2 SVC Example 1

```
void *helper = svc_init();
```
Return value: helper

```
hash_file(helper, "hello.py");
```
Return value: `2027` (from example above)

```
hash_file(helper, "fake.c");
```
Return value: `-2` (non-existent file)

```
svc_commit(helper, "No changes");
```
Return value: `NULL`

```
svc_add(helper, "hello.py");
```
Return value: `2027`

```
svc_add(helper, "Tests/test1.in");
```
Return value: `564` (from example above)

```
svc_add(helper, "Tests/test1.in");
```
Return value: `-2`

```
svc_commit(helper, "Initial commit");
```
Return value: `"74cde7"`

The ASCII values for the commit message `Initial commit` are [73, 110, 105, 116, 105, 97, 108, 32, 99, 111, 109, 109, 105, 116]. After the first stage, this gives an id of 1395 % 1000 = 395. The two changes are additions of `hello.py` [104, 101, 108, 108, 111, 46, 112, 121] and `Tests/test1.in` [84, 101, 115, 116, 115, 47, 116, 101, 115, 116, 49, 46, 105, 110]. Following the algorithm, the id is then 7654887. Converting this to a 6 character hexadecimal string gives `74cde7`.

```c
void *commit = get_commit(helper, "74cde7");
```
Return value: Pointer to area of memory containing the commit created above

```c
int n_prev;
char **prev_commits = get_prev_commits(helper, commit, &n_prev);
```
Return value: `NULL`
Afterwards, `n_prev = 0`

```c
print_commit(helper, "74cde7");
```
Output:

```
74cde7 [master]: Initial commit
    + hello.py
    + Tests/test1.in

    Tracked files (2):
    [      2027] hello.py
    [       564] Tests/test1.in
```

```c
int n;
char **branches = list_branches(helper, &n);
```
Output: `master`
Return value: Array with pointer to area of memory containing the string `master`
Afterwards, `n = 1`

4.3 Commit ID

In the above example, the commit ID for the valid commit "Initial commit" is calculated by:

1. The commit message has ascii values [73, 110, 105, 116, 105, 97, 108, 32, 99, 111, 109, 109, 105, 116] so after the first step, the ID is 1395 % 1000 = 395
2. There are two changes, both of which are additions
3. The first change adds 376591 to the ID, and then for each byte, the calculation in Section 3.2 is followed, giving 111
4. The second change also adds 376591 to the ID, and the above is repeated giving 7654887
5. Converting this to a 6 digit hexadecimal number gives `74cde7`

4.4 SVC Example 2

Starting from a blank project (no files), we create two files:

`COMP2017/svc.h` has the contents

```
#ifndef svc_h\n
#define svc_h\n
void *svc_init(void);\n
#endif\n
```

`COMP2017/svc.c` has the contents

```
#include "svc.h"\n
void *svc_init(void) {\n
    // TODO: implement\n
}\n
```

The hashes of the two files are 5007 and 5217 (all 'tabs' are four spaces and new line characters have been explicitly shown above).

```
svc_add(helper, "COMP2017/svc.h");
```
Return value: `5007`

```
svc_add(helper, "COMP2017/svc.c");
```
Return value: `5217`

```
svc_commit(helper, "Initial commit");
```
Return value: `"7b3e30"`

```
svc_branch(helper, "random_branch");
```
Return value: `0`

```
svc_checkout(helper, "random_branch");
```
Return value: `0`

Next, the file `COMP2017/svc.c` is changed to have the following contents

```
#include "svc.h"\n
void *svc_init(void) {\n
    return NULL;\n
}\n
```

which has the hash 4798.

```
svc_rm(helper, "COMP2017/svc.h");
```
Return value: `5007`

```
svc_commit(helper, "Implemented svc_init");
```
Return value: `"73eacd"`

You realise you accidentally deleted `COMP2017/svc.h` and want to revert to the initial commit

```
svc_reset(helper, "7b3e30");
```
Return value: `0`

Then, the file `COMP2017/svc.c` is changed again to have the contents shown above.

```
svc_commit(helper, "Implemented svc_init");
```
Return value: `"24829b"`

```
void *commit = get_commit(helper, "24829b");
```
Return value: Pointer to area of memory containing the commit created above

```
int n_prev;
char **prev_commits = get_prev_commits(helper, commit, &n_prev);
```
Return value: Pointer to an array of length one, containing `"7b3e30"`
Afterwards, `n_prev = 1`

```
svc_checkout(helper, "master");
```
Return value: `0`

The test framework creates a file `resolutions/svc.c` with the contents

```
#include "svc.h"\n
void *svc_init(void) {\n
    return NULL;\n
}\n
```

The following code is then executed to perform a merge:

```
// Resolution(s) are created by the test framework
resolution *resolutions = malloc(sizeof(resolution));
resolutions[0].file_name = "COMP2017/svc.c";
resolutions[0].resolved_file = "resolutions/svc.c";

// Call to merge function
svc_merge(helper, "random_branch", resolutions, 1);

// The test framework will free the memory
free(resolutions);
```

Return value: `"48eac3"`
The commit message is `Merged branch random_branch`. The conflicts array indicates that only the change is to be kept, following the merge rules described above. This means that from the perspective of the `master` branch, the only change to be committed is this modification.

```
int n_prev;
void *commit = get_commit(helper, "48eac3");
char **prev_commits = get_prev_commits(helper, commit, &n_prev);
```
Return value: Pointer to an array of length two, containing `"7b3e30"` and `"24829b"` in that order. Afterwards, `n_prev = 2`

5 Notes and Hints

- In SVC, branches are not deleted. After merging a branch, it remains in the list of branches.
- When a reset to a commit occurs, don't forget to undo all types of changes (addition, deletion, modification) to the files too!
- Useful tools and functions: address sanitizer, `valgrind`, `strdup`
- The hashing algorithm may produce the same hash, even if the contents of the file have changed. Test cases are designed with this in mind and it is guaranteed this will not occur.
- Error codes should be checked in order described in the function descriptions.
- A file may be deleted from the file system without `svc_rm` being called Your code should detect that this has happened and remove the file from SVC accordingly. Note, however, that the function `svc_rm` should **not** delete the file from the file system!
- You should write your own test cases and draw out diagrams to help visualise the problem.
- The maximum file path is 260 characters long (excluding NULL terminator)
- Branch names are at most 50 characters long (excluding NULL terminator)
- All other strings can be of any length
- The two examples given above correspond to the example test cases on Ed
- `svc_add` is a command to start tracking the file in the current branch (different to Git!). From this point, any changes to these files should then be detected and put in the 'staged' state by your code
- `svc_rm` is a command to stop tracking the file in the current branch (different to Git!)
- Changes to files and manual deletion should only be detected when `svc_commit` is called

5.1 Staging

In version control systems, there is a concept of "staging". Adding/removing/changing files are kept in the "staging" phase until a call to commit actually occurs. This means that multiple changes can occur in the staging phase that should only be represented as a single change when a commit occurs. For example, modifying a file twice should appear only as a single change in the commit. Adding a file and then removing it before committing will mean neither of these appear in the commit's changes. Think about all the possible combinations of file operations that could occur and what the effective change actually is (hint: there are 5 pairs you should consider).

5.2 Resolution Struct

The `resolution` struct is provided in the scaffold code and is also provided below for reference. Do **not** modify this struct

```
typedef struct resolution {
    char *file_name;
    char *resolved_file;
} resolution;
```

6 Submission

You must submit your code via git to Ed for automatic marking. It must not produce any compilation errors, and it must free any dynamically allocated memory unless otherwise specified in Section 3.

Your code will be compiled with the following options:
`gcc -O0 -std=gnu11 -lm -Wextra -Wall -Werror -g -fsanitize=address`

Warning: Any attempts to deceive or disrupt the marking system will result in an immediate zero for the entire assignment. Negative marks can be assigned if you do not follow the assignment problem description or if your code is unnecessarily or deliberately obfuscated.

7 Mark Breakdown

This assignment is worth 10 % of your final mark. There are 10 points available (1 % each):

Component	Points	Description
Correctness	6	Proportion of test cases passed on Ed
Performance	2	How much memory and time does your program take?
Code structure and style	2	Manual marking by your tutor. Marks depend on style, layout and readability of your code

Note on performance: memory and time usage are each worth 1 point and will be proportionately allocated based on other students' submissions. The memory usage is based on the maximum memory allocated at any given time in the execution of your code. You should consider different data structures and algorithms before implementing your solution.

8 Academic Declaration

By submitting this assignment you declare the following:

I declare that I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure, and except where specifically acknowledged, the work contained in this assignment/project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgment from other sources, including published works, the Internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

I acknowledge that the School of Computer Science, in assessing this assignment, may reproduce it entirely, may provide a copy to another member of faculty, and/or communicate a copy of this assignment to a plagiarism checking service or in-house computer program, and that a copy of the assignment may be maintained by the service or the School of Computer Science for the purpose of future plagiarism checking.