Rapport du projet final – LU2IN006

Youssef El Atia - David Lin

17 avril 2022

Réalisation d'un logiciel de gestion de versions

Table des matières :

1. Introduction du Projet 3
1.1 Git 3
1.2 Reformulation synthétique de notre projet 3
2 Description global de notre code 3
3 Description des structures utilisés 4
3.1 List 4
3.2 WorkFile 4
3.3 WorkTree 4
3.4 kvp 5
3.5 Commit 5
3.6 Descriptions des bibliothèques importés 6
4 Conception de notre projet 6
5 Description du code 8
5.1 Description des fonctions importantes utilisés le projet 8
5.2 Commande git 10
6 Conclusion 11

1. Introduction du Projet

1.1 Git

Git est un système de contrôle de version décentralisé qui permet de suivre l'évolution d'un projet informatique. Il permet à plusieurs développeurs de travailler simultanément sur un même projet en gérant les modifications apportées aux fichiers. Chaque modification est enregistrée sous forme de "commit" qui contient un message explicatif et une référence à la version précédente. Les branches permettent de travailler sur des fonctionnalités indépendantes et de les fusionner par la suite. Git facilite également la collaboration entre les développeurs en permettant le partage et la synchronisation du code source entre les différentes personnes travaillant sur le projet

1.2 Reformulation synthétique de notre projet

Ce projet a pour objectif d'étudier le fonctionnement d'un logiciel de gestion de versions, en se concentrant sur la création d'enregistrements instantanés, la navigation entre les différentes versions, la construction et la maintenance d'une arborescence de versions, ainsi que la sauvegarde de changements non inclus dans une version instantanée. Notre objectif final est d'avoir un exécutable intitulé "myGit" qui nous permettra de simuler les différentes commandes disponibles sur Git, telles que "git add", "git commit", "git branch", etc. Ce projet a été conçu en langage C.

2 Description global de notre code

Notre projet se trouve dans le dossier "projet_el-atia_lin". Ce projet a été réalisé dans un esprit de modularité, car un projet est destiné à évoluer, et un code plus léger favorise la facilité de développement. À l'intérieur de ce dossier, nous avons plusieurs fichiers .c ainsi qu'un répertoire headers. A chaque fichier .c on a un fichier .h associé :

- **hash.c** et **hash.h** : qui contiennent les fonctions relatives à la gestion d'un hash d'un fichier
- **list.c** et **list.h:** qui contiennent les fonctions de gestions d'une structure de type liste simplement chainée.
- **work.c** et **work.h** : qui contiennenttient les fonctions de gestions des WorkFiles et des WorkTrees (sauvegarde et restauration d'un worktreee...)
- **commit.c** et **commit.h** : qui contiennent les fonctions de gestions des commits (créations,affichage..)
- **ref.c** et **ref.h** : qui contiennent les fonctions de gestions des références et les fonctions qui font le liens entre les références et les commits.
- **branch.c** et **branch.h** : qui contiennentient les fonctions de gestions des branches
- **timeline.c** et **timeline.h** qui contiennent les fonctions de gestions d'une timeline arborescente.
- **merge.c** et **merge.h** : qui contiennent les fonctions de gestions des fusions des branches.

Un fichier principal "**main.c**" est également mis à la disposition de l'utilisateur afin qu'il puisse tester les fonctions utilisées. Un fichier "**myGit.c**" est également fourni afin de permettre la simulation des commandes Git.

Une documentation complète des commandes disponibles dans notre Git est disponible en dernière page.

3 Description des structures utilisés

Lors de ce projet, nous avons été amenés à manipuler plusieurs types de structures de données. Voici une description pour chacune d'entre elles.

3.1 List

Cette structure de type liste chaînée contient un champ 'data' qui va contenir un pointeur de type 'char*' pointant sur un tableau de chaînes de caractères, et un pointeur 'next' qui pointe vers l'élément suivant de la liste, initialisé à NULL. Nous avons choisi d'implémenter une liste chaînée car nous souhaitons par exemple connaître les noms des fichiers et répertoires qui s'y trouvent, et étant donné que nous ne connaissons pas la taille à l'avance, une liste chaînée semble être la structure la plus appropriée. Cette structure se trouve dans le fichier 'list.h'.

```
1  typedef struct cell {
2      char* data;
3      struct cell* next;
4  } Cell;
5
6  typedef Cell* List;
```

3.2 WorkFile

Un WorkFile est une structure qui nous permet de représenter un fichier ou un répertoire pour lequel nous souhaitons enregistrer un instantané. Cette structure est composée de 3 champs : 'name' de type 'char*' correspondant au nom du fichier ou du répertoire, 'hash' de type 'char*' correspondant au hash associé à son contenu, initialisé à NULL par défaut, et 'mode' de type 'int' nous permettant d'indiquer les autorisations associées au fichier (modification, lecture et exécution), initialisé à NULL par défaut. Cette structure se trouve dans le fichier 'work.h'.

```
1  typedef struct {
2      char* name;
3      char* hash;
4      int mode;
5  } WorkFile;
```

3.3 WorkTree

L'enregistrement d'un seul instantané n'est pas suffisant car le plus souvent il faut gérer des ensembles de fichiers, c'est la raison pour laquelle nous avons implémenté les structures de worktrees et workfiles. Un WorkTree est une structure qui nous permet de stocker plusieurs WorkFile. C'est un tableau de WorkFile qui a un champ 'tab' de type 'WorkFile*' pointant vers un tableau de WorkFile, un champ 'size' pour indiquer la taille qui est fixe et définie par une constante du programme, N, qui vaut 1000, et un champ 'n' de type 'int' qui est initialisé par défaut à 0.

```
7  typedef struct {
8      WorkFile* tab;
9      int size;
10     int n;
11 } WorkTree;
```

3.4 kvp

```c
typedef struct key_value_pair{
    char* key;
    char* value;
} kvp;
```

La structure de données kvp (key-value pair) est utilisée dans notre projet pour représenter les informations nécessaires à un Commit. Un Commit doit contenir une paire de clé-valeur de la forme ('tree', 'hash'), où 'hash' est le hash du fichier correspondant à l'enregistrement d'un WorkTree. Nous utilisons cette structure pour garantir qu'un Commit contient bien cette paire de clé-valeur. Elle est composée d'un champ 'key' de type 'char*' et d'un champ 'value' de type 'char*'.

3.5 Commit

```c
typedef struct hash_table{
    kvp** T;
    int n;
    int size;
} HashTable;

typedef HashTable Commit;
```

Pour pouvoir suivre l'évolution d'un projet, il est nécessaire d'organiser chronologiquement différents enregistrements instantanés. Pour cela, nous utilisons des commits (ou points de sauvegarde), qui sont des enregistrements instantanés associés à des étapes jugées importantes dans la chronologie du projet. Dans notre implémentation, un commit est associé à l'enregistrement d'un instantané d'un WorkTree (voir section 2.2). Par conséquent, un commit est implémenté comme une table de hachage (Note 1.2), où les éléments de la table correspondent aux informations associées au point de sauvegarde. Un commit doit contenir une paire de clé-valeur de la forme mentionnée dans la section 2.2.4.

Note 1.2 : Une table de hachage est une structure de données qui permet de stocker et de rechercher des données en associant une clé à une valeur à l'aide d'une fonction de hachage, permettant ainsi un accès rapide aux données en temps constant en moyenne, indépendamment de la taille de la table. Lorsque les cellules de la table de hachage contiennent elles-mêmes les clés, alors on dit que la table de hachage est gérée en adressage ouvert.

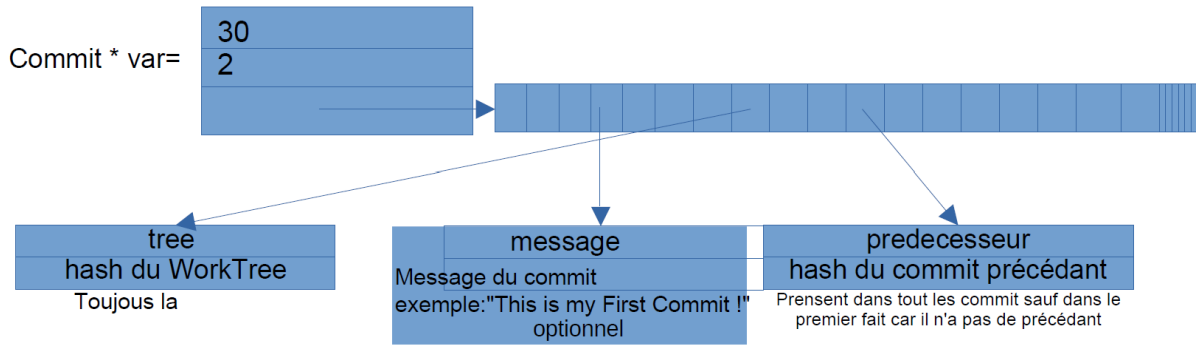


Figure 1.1 : Schéma de la structure d'un commit

3.6 Descriptions des bibliothèques importés

Les bibliothèques utilisées dans ce projet sont les suivantes :

stdio.h : pour les fonctions d'E/S (entrée/sortie) standard.

stdlib.h : pour les fonctions générales du langage C (malloc(), free()...).

string.h : pour les fonctions de manipulation de chaînes de caractères.

dirent.h : pour les fonctions de gestion des répertoires, notamment opendir et

readdir. sys/stat.h : pour les fonctions de récupération d'informations sur les fichiers à partir de leurs descripteurs de fichiers.

unistd.h : pour la gestion efficace des fichiers temporaires, notamment pour stocker le hash d'un fichier en attente de lecture.

sys/types.h : pour les fonctions de gestion des types de fichiers.

errno.h : pour la gestion des erreurs en cas d'échec d'une fonction.

4 Conception de notre projet

La conception de notre version de git en C a été réalisée en suivant une approche modulaire pour faciliter la maintenance du code et la réutilisation des différents modules. Nous avons commencé par définir les objectifs de notre projet et les exigences spécifiques liées à cette dernière.

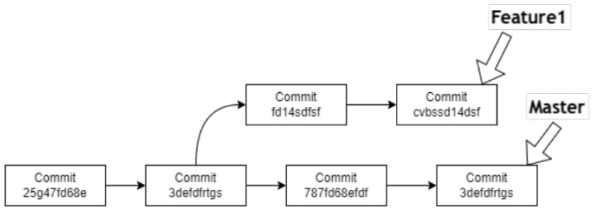
Nous avons ensuite suivi les étapes suivantes pour concevoir notre projet :

- **Création d'enregistrements instantanés** : le but de cette première est de poser les bases essentielles au bon fonctionnement de notre projet. En effet nous nous intéresserons à la gestion de fichiers sous git, notre but étant de réussir à la fin de cette première partie d'être en mesure d'enregistrer un instantané d'un fichier donné. Pour ce faire nous avons donc utiliser la structure une structure de type List nous offrant plusieurs fonctionnalités utiles pour la suite du projet (comme obtenir les noms des fichiers et répertoires qui s'y trouvent). Enfin, nous avons trouvé une façon de créer des fichiers temporaires, c'est-à-dire des copies de fichiers à des instants. Cela nous permettra plus tard de récupérer différentes versions d'un projet.

- **Enregistrements de plusieurs instantanés** : Dans cette seconde partie notre but est de pouvoir enregistrer un instantané de plusieurs fichiers et ou répertoire, de la même manière que sous git. Nous utiliserons alors un WorkTree qui va nous permettre de stocker en mémoire plusieurs dossier et fichiers représenter eux même par un WorkfFile, l'objectif final de cette étape de pouvoir avoir cette fois non plus un enregistrement d'un seul fichier mais d'un WorkTree, c'est-à-dire d'un ou plusieurs fichiers/répertoires.

-**Gestion des commit** : Nous savons comment faire un enregistrement instantané d'un WorkTree, nous souhaitons désormais suivre l'évolution du projet, c'est-à-dire créer des enregistrements instantanés associés à chaque étape que l'on juge importante dans la chronologie du projet. Pour cela nous avons fait le choix d'utiliser des commits. Avec le choix de notre implémentation un commit est donc associé à l'enregistrement instantanée d'un WorkTree, accompagné d'informations relatives aux points de sauvegarde (auteur, message...). Le but de cette partie étant de pouvoir simuler les premières commandes de git tel que git init, add commit (la documentation complète de notre exécutable myGit se trouve en) et réaliser nos premiers commit, tout cela pour l'instant organisé de manière linéaire (même branche). Pour cela nous avons fait une fonction qui créer un répertoire. refs qui contient les références, et notamment un fichier HEAD et master pour l'instant. Nous avons fait le choix d'organiser les commit en listes simplement chaînée, ainsi il suffit de connaître le hash du dernier commit pour avoir accès à tous les autres à l'aide de la clé predecessor.

-**Gestion d'une timeline arborescente** : Nous avons précédemment vu comment réaliser des commits sur une seule et même branche. Néanmoins lors d'un projet certains utilisateurs sont amenée à réaliser des modifications qu'ils ne veulent pas push sur la branche principale, et donc de considérer plusieurs versions du même code. Par conséquent le but de 4ème partie est donc de considérer plusieurs branches. et de savoir comment naviguer en entre les différentes branches de l'arborescence. Nous avons par conséquent notre fichier myGit afin de prendre en comptes les commandes relatives à la gestions des branches (git checkout, git branch...).



-**Gestions des fusions des branches** : Dans cette dernière partie, le but est de pouvoir intégrer la fusion de branches dans Git, en effet, c'est une fonctionnalité importante lorsque plusieurs personnes travaillent en parallèle sur un même projet et doivent réunir leur travail. La fusion de branches permet de créer un nouveau commit en fusionnant les WorkTrees des derniers commits de deux branches. Cependant, cela peut entraîner des conflits lors de la fusion des WorkTrees, par exemple si deux fichiers/répertoires portent le même nom mais ont des contenus différents. Nous avons donc proposé des méthodes de gestion de conflits pour permettre une fusion de branches pertinente.

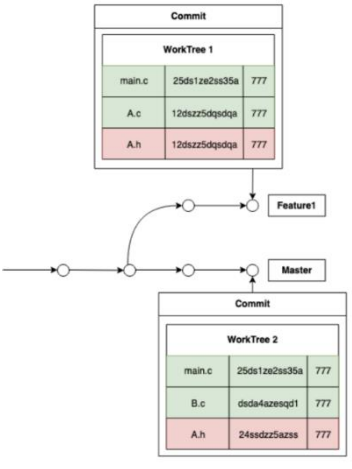


Schéma 1.2 : cas de conflit en voulant fusionner la branche master avec la branche Feature1

5 Description du code

5.1 Description des fonctions importantes utilisés le projet

char* sha256file(char *file) : prend en paramètre un pointeur sur le path du fichier dont on souhaite obtenir le hash et nous renvoie une chaîne d e caractères contenant le hash du fichier à calculer . Le hash est calculé à partir de la commande bash sha256sum qui nous permet de hacher le contenu d'un fichier en utilisant la fonction de hachage SHA256. Cette fonction est primordiale et est réutilisé dans de nombreuses fonctions telle que blobfile, blobWorktree, blobCommit, car elle est utile pour enregistre un instantané d'un fichier (ou également nous le verrons un WorkFile,Worktree ou encore un Commit) donné en entrée.

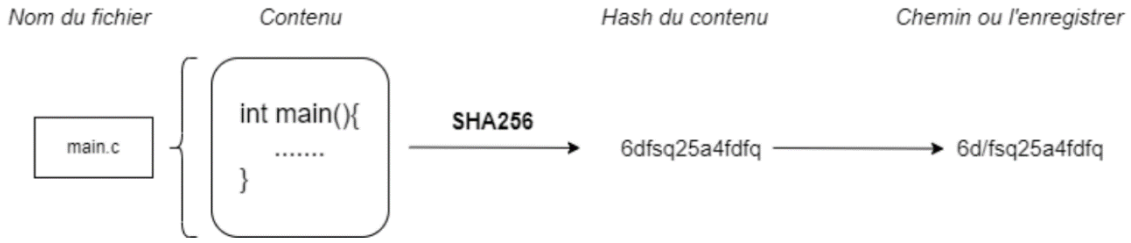


Figure 1.3 : Représentation de la fonction SHA256

void blobFile(char* file) : nous permet d'enregistrer un instantanée du fichier que l'on donne en paramètre. La fonction prend en entrée un nom de fichier "file" et calcule le hash SHA-256 du fichier en appelant la fonction sha256file(file). Elle crée une copie du hash pour récupérer les deux premiers caractères. Elle vérifie par la suite si le dossier correspondant aux deux premiers caractères du hash existe déjà, et, si ce n'est pas le cas, elle crée un dossier avec ces deux caractères en utilisant la commande shell mkdir, elle calcule par la suite le chemin du fichier à partir de son hash en appelant la fonction hashToPath(hash). Enfin elle copie le fichier dans le dossier correspondant à son hash en utilisant la fonction void cp(). Elle finit par libérer la mémoire allouée pour le hash, la copie du hash et le chemin du fichier en appelant free() pour éviter les fuites de mémoire.

List* listdir(char *root_dir) :prend en paramètre une adresse et renvoie une liste (définie au 3.1) qui liste les fichiers et dossiers dans un répertoire donné. Elle crée une liste chaînée vide et parcourt les entrées du répertoire en créant une cellule pour chaque fichier ou dossier rencontré, qu'elle insère en tête de liste. Elle ferme ensuite le répertoire et retourne la liste contenant les noms des fichiers et dossiers. Si une erreur d'ouverture du répertoire se produit, elle affiche un message d'erreur et retourne NULL.

char* blobWorkTree(WorkTree* wt) : cette fonction prend en paramètre le WorkTree à représenter, et elle va créer un fichier temporaire le représentant pour pouvoir par la suite créer l'enregistrement instantané de ce dernier, avec une extension .t. Cette fonction nous retourne le hash du fichier temporaire. Elle génère un fichier temporaire dans /tmp, puis écrit le contenu de la structure WorkTree dans ce fichier, calcule son hachage SHA-256,

génère un chemin de fichier basé sur ce hachage, crée un répertoire correspondant aux deux premiers caractères du chemin, et, s'il n'existe pas, copie le contenu du fichier temporaire dans le fichier correspondant au chemin, et retourne le hachage du fichier temporaire.

char* saveWorkTree(Worktree* wt, char* path): nous permet de sauvegarder la structure de données WorkTree pointée par wt, dans un dossier spécifié par le chemin path. Elle itère à travers les éléments de wt, récupère le chemin du dossier courant, crée un nouveau WorkTree pour les sous-dossiers, ajoute les fichiers et dossiers à ce nouveau WorkTree, sauvegarde récursivement le nouveau WorkTree dans le dossier courant, récupère les permissions des fichiers et dossiers, et retourne le hachage du WorkTree initial.

void restoreWorkTree(Worktree* wt, char* path) : permet de restaurer les fichiers et dossiers sauvegardés dans un dépôt en utilisant la structure de données WorkTree. Elle commence par construire le chemin absolu du fichier/dossier à restaurer en concaténant path avec le nom de fichier/dossier dans wt. De mê me elle construit le chemin du fichier/dossier dans le dépôt avec le hachage stocké dans wt. Elle vérifie ensuite, si l'élément dans wt est un fichier ou un dossier en utilisant la fonction isWorkTree(hash). Si l'enregistrement possède l'extension ".t", alors il s'agit d'un répertoire. Dans ce cas, il faut créer le WorkTree associé, modifier la variable path en y ajoutant ce répertoire à la fin, puis faire un appel récursif sur ce nouveau WorkTree. Par contre, si c'est un fichier on crée une copie de l'enregistrement à l'endroit indiqué par la variable path, et on lui donne le nom et les autorisations correspondants aux champs name et mode.

char* blobCommit(Commit* c) :crée un enregistrement instantané d'un commit en passant par un fichier temporaire, cette fonction nous retourne le hash du fichier temporaire. Elle crée un fichier temporaire, avec un nom généré de manière aléatoire, dans le répertoire "/tmp". Elle écrit ensuite les données du Commit, dans ce fichier, et calcule le hash SHA256 du contenu du fichier, à l'aide de la fonction sha256file. Elle génère un nom de fichier, à partir du hash en utilisant la fonction hashToFile, et y ajoute l'extension ".c". Enfin, elle copie le contenu du fichier généré dans le fichier temporaire.

WorkTree* mergeWorkTrees(WorkTree* wt1, WorkTree* wt2, List **conflicts): cette fonction prend deux pointeurs vers des structures de type WorkTree, wt1 et wt2, ainsi qu'un pointeur vers un pointeur de liste 'conflicts'. Elle initialise une nouvelle structure WorkTree wt et une liste conflicts. Elle parcourt le premier WorkTree wt1, et compare chaque fichier avec le WorkTree wt2. Si un fichier a le même nom, mais un hash différent entre les deux WorkTrees, il est ajouté à la liste des conflits. Sinon, le fichier est ajouté à wt. Par la suite, elle parcourt wt2 de la même manière et ajoute les fichiers qui ne sont pas déjà présents dans wt. Finalement, elle retourne wt, le WorkTree résultant de la fusion.

List* merge(char* remote branch, char* message) : Cette fonction prend en entrée le nom d'une branche distante 'remote_branch' et un message de fusion 'message'. Elle effectue la fusion de la branche courante avec la branche distante. Elle commence par récupérer le hash du commit et du WorkTree de la branche courante, ainsi que le hash du commit et du WorkTree de la branche distante. Ensuite, elle fusionne les deux WorkTrees

en appelant la fonction mergeWorkTrees en passant les pointeurs vers les WorkTrees courant et distant, ainsi qu'un pointeur vers une liste conflicts. Si des conflits sont détectés, la fonction retourne la liste des conflits. Et, si aucune liste de conflits n'est générée, la fonction crée un nouveau commit en utilisant le WorkTree fusionné. Elle met à jour les métadonnées du commit, telles que le message, les prédécesseurs (commit courant et commit distant), et crée une référence vers le nouveau commit. Ensuite, elle supprime la référence de la branche distante, restaure le WorkTree fusionné dans le répertoire de travail actuel. Enfin, la fonction retourne NULL pour indiquer que la fusion s'est déroulée sans conflits.

void createDeletionCommit(char* branch, List* conflicts, char* message): cette fonction crée et ajoute un commit de suppression sur la branche 'branch', correspondant à la suppression des éléments de la liste 'conflicts'. Pour ce faire cette fonction effectue un checkout sur une branche spécifiée, récupère le dernier commit et le WorkTree associé, vide la zone de préparation, ajoute les fichiers du WorkTree qui ne sont pas en conflit, crée un commit de suppression avec un message spécifié, et enfin revient sur la branche de départ.

5.2 Commande git

Notre programme myGit est utilisé en ligne de commande, avec des arguments pour sélectionner les différentes fonctionnaliste tel que ./myGit <argument1> <argument2>... Voici la liste des commandes disponibles sur notre version de myGit accompagnées de leur documentation :

. **myGit init** : initialise le répertoire de références et la brache courante .

. **myGit list-refs** : affiche toutes les références existantes.

. **myGit create-ref <name> <hash>** : crée la référence name qui pointe vers le commit correspondant au hash donée.

. **myGit delete-ref <name>:** supprime la référence name.

. **myGit add <elem> [<elem2><elem3>...]** : ajoute un ou plusieurs fichiers/répertoires à la zone de préparation (pour faire partie du prochain commit).

. **myGit list-add** : affiche le contenu de la zone de préparation.

. **myGit clear-add** : vide la zone de préparation.

. **myGit commit <branch_name> [-m <message>]** : effectue un commit sur une branche, avec ou sans message descriptif.

. **myGit get-current-branch** : affiche le nom de la branche courante

. **myGit brach <branch-name>:** crée une branche qui s'appelle <branch_name>, si elle n'existe pas, la commande doit afficher un message d'erreur.

. **myGit branch-print** : affiche le hash de tous les commits de la branche, accompagné de leur message descriptif éventuel. Si la branche n'existe pas, un message d'erreur est affiché

. **myGit checkout-branch <branch-name> :** réalise un déplacement sur la branche <branch-name> . Si cette branche n'existe pas, un message d'erreur est affiché

. **myGit checkout-commit <pattern>** : réalise un déplacement sur le commit qui commence par <pattern>. Des messages d'erreur sont affichés quand <pattern> ne correspond pas à exactement un commit.

. **myGit merge :** nous permet de fusionner les modifications de deux branches différentes en une seule.

- S'il n'y a pas de collision entre la branche courante et la branche <branch>, on réalise le merge et on affiche un message à l'utilisateur pour lui dire que la fusion s'est bien passée

- S'il y a des collisions, on propose à l'utilisateur de choisir une des options suivantes :

  1. Garder les fichiers de la branche courante, et donc créer un commit de suppression pour la branche, avant de faire appel à la fonction merge.

  2. Garder les fichiers de la branche, et donc créer un commit de suppression pour la branche courante, avant de faire appel à la fonction merge.

  3. Choisir manuellement, conflit par conflit, la branche sur laquelle il souhaite garder le fichier/répertoire qui est en conflit. Dans ce cas, il faudra que l'on divise la liste de conflits en deux listes, selon ce que l'utilisateur nous dira, puis créer un commit de suppression sur chaque branche avec ces deux listes, avant de faire la fusion avec la fonction merge.

6 Conclusion

En somme, ce projet nous a permis d'approfondir notre compréhension du fonctionnement des logiciels de gestion de versions ainsi que des différentes structures de données qui y sont impliquées. Nous avons pu étudié les fonctionnalités clés d'un tel système, telles que la création d'enregistrements instantanés, la navigation entre les versions, ainsi que la construction et la maintenance d'un arbre de versions. Ainsi nous avons pris conscience de la complexité de la mise en œuvre de ces fonctionnalités, et de l'importance de comprendre les interactions entre les différentes structures de données. Par ailleurs, grâce à notre travail sur ce projet, nous avons acquis une compréhension approfondie de ces concepts fondamentaux ainsi que des compétences pratiques pour mettre en place un système de contrôle de version.