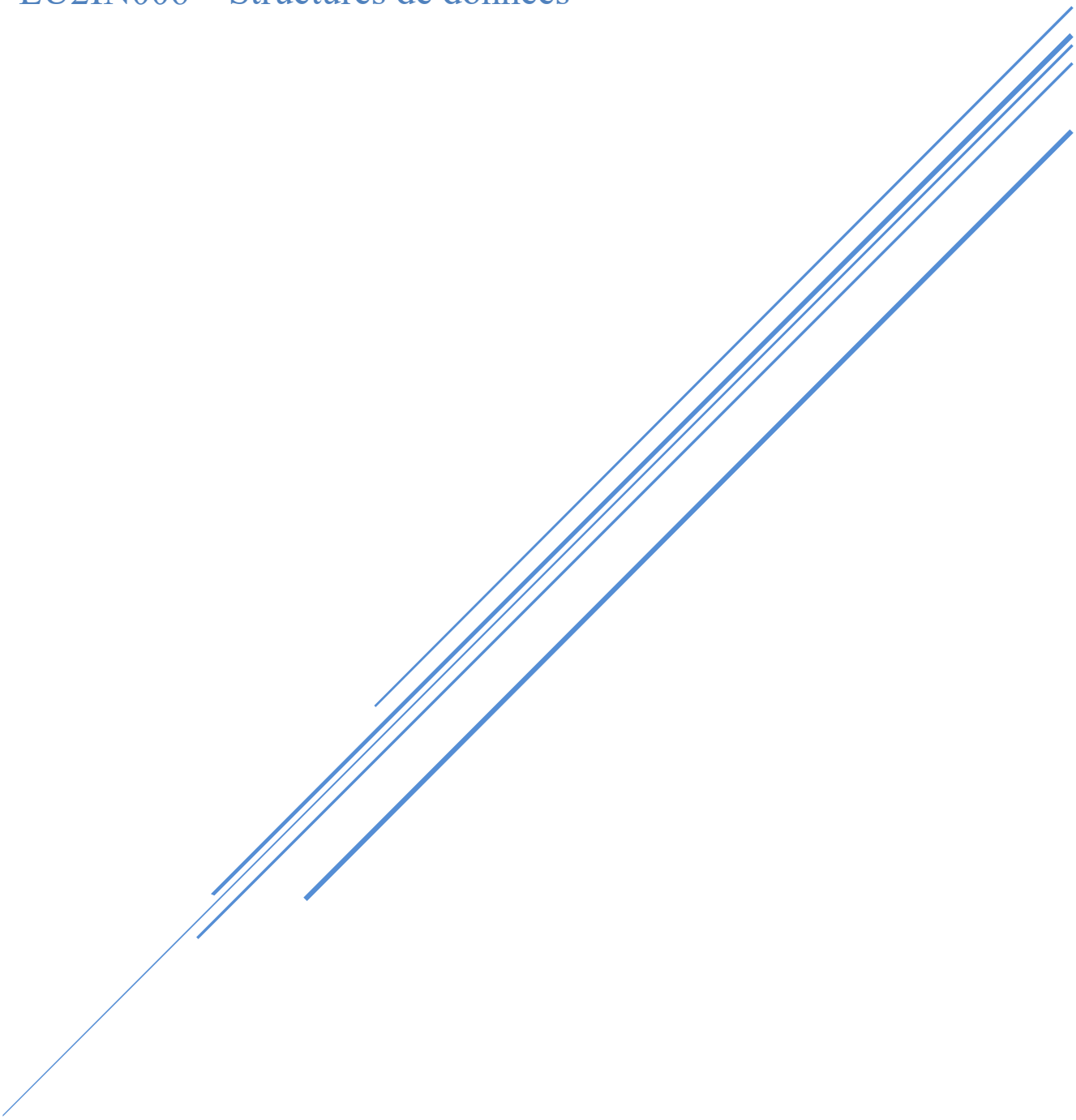
LOGICIEL DE GESTION DE VERSIONS

LU2IN006 – Structures de données



ACHOURI Sira Lina 21102770
ANTHONY QUEEN Bryan 21104114
Sorbonne Université – Sciences et Ingénierie
Année universitaire : 2022-2023

SOMMAIRE

Introduction :

Dans le cadre de notre projet, nous sommes amenés à simuler un logiciel de gestion de versions ou VCS en anglais (version control system). C'est un outil permettant de sauvegarder et gérer des codes source d'un projet. Ce type de logiciel permet de créer un historique de toutes les modifications effectuées sur un fichier et permet d'avoir la possibilité de restaurer des versions antérieures du code, ce qui est par exemple très pratique lors de corruption de fichier. L'un des principaux avantages de ce type d'outil est l'amélioration des travaux collaboratifs avec notamment la possibilité de créer des versions alternatives afin de développer différentes fonctionnalités d'un projet sans affecter l'ensemble de ce dernier. Le logiciel Git, affilié à la plateforme GitHub, fait partie des logiciels de gestion de versions les plus utilisés notamment par les développeurs et les éditeurs logiciel, c'est d'ailleurs de ce logiciel que ce projet est inspiré afin d'étudier les diverses propriétés des VCS. L'objectif de ce projet est la mise en œuvre de fonctionnalités de certains aspects des logiciels de versionnage de code tout en manipulant différentes structures de données.

I- Création d'un enregistrement instantané

Tout d'abord, nous allons commencer par implémenter un programme permettant de créer un enregistrement instantané. Enregistrer un instantané consiste à créer une copie du contenu d'un fichier dans un autre fichier ayant pour chemin un hachage obtenu à partir du contenu du fichier à sauvegarder. Pour ce faire, nous allons utiliser la fonction SHA256, une fonction de hachage utilisée en cryptographie, puis on obtient le chemin vers l'instantané en insérant un '/' entre le deuxième et troisième caractère du hachage. Nous allons aussi manipuler des fichiers temporaires, très utiles et efficaces pour stocker le hachage d'un fichier en attente de lecture par notre programme, ces fichiers temporaires sont créés avec la fonction mkstemp() de la bibliothèque standard unistd.h.

Dans cette partie, nous manipulons une structure de données linéaire : les listes simplement chaînées, présentes dans le fichier liste.c *(figure 1)*. Ces dernières sont représentées par des cellules contenant une chaîne de caractères. Les opérations courantes effectuées avec cette structure sont la recherche d'un élément dans la liste, l'insertion d'un élément, l'affichage, la libération de la mémoire occupée par la liste, la taille d'une liste, etc. Il existe aussi d'autres de fonctions de manipulation des listes chainées, comme les fonctions de conversion vers des chaînes de caractères et des fichiers et leurs transformations inverses.

La fonction principale de cette première partie est la fonction blobFile() du fichier dir.c, qui crée un enregistrement instantané du fichier donné en paramètre. La fonction commence par construire le chemin vers l'instantané grâce à la fonction de hachage SHA256, puis crée le dossier et le fichier associés à l'enregistrement instantané. Enfin, elle termine par copier le contenu du fichier en paramètre vers l'instantané.

```c
typedef struct cell {
char* data;
struct cell* next;
} Cell;

typedef Cell* List;
```

figure 1 : Structures d'une cellule et d'une liste

II- Enregistrement de plusieurs instantanés

Jusqu'ici, nous avons implémenté une fonction permettant de créer un enregistrement instantané d'un fichier, mais dans la plupart du temps, nous exploitons des ensembles de fichiers et de répertoires formant une structure hiérarchique. Ainsi, nous devrons pouvoir créer des sauvegardes de ces structures arborescentes afin de aussi de pouvoir récupérer une version antérieure de code source.

Nous allons travailler avec de nouvelles structures de données dans le fichier work.c. Tout d'abord, nous avons les WorkFiles *(figure 2)*, qui représentent les fichiers ou les répertoires dont nous voulons faire des enregistrements instantanés. Cette structure est composée du nom d'un fichier ou d'un répertoire, d'un hachage qui représente l'instantané associé et enfin d'un mode. Le mode d'un fichier ou d'un répertoire, représenté par 3 entiers en octal, définit les autorisations sur celui-ci, que ce soit la lecture, l'écriture ou l'exécution et ce pour nous-même, pour notre groupe et pour tous les utilisateurs. Nous avons à notre disposition deux fonctions qui permettent de manipuler les modes, getChmod() et setMode(). Ensuite, nous travaillons aussi avec des WorkTrees *(figure 3)* : ce sont des tableaux à une dimension de WorkFiles. Elles définissent le contenu d'un répertoire, composé de fichiers et/ou d'autres répertoires. Cette structure contient la taille du tableau, que nous avons, au préalable, fixée avec la macro SIZE_WT dans le fichier work.h, et aussi le nombre d'éléments présents dans le tableau. De la même manière que pour les listes, nous avons des fonctions permettant d'initier un WorkTree, de rechercher un élément et d'insérer un WorkFile dans un WorkTree, ainsi que des fonctions de conversion en chaîne de caractères et en fichiers pour ces deux structures.

Pour pouvoir produire un instantané d'un WorkTree, nous utilisons la fonction blobWorkTree(), qui crée un instantané et renvoie le hachage affilié, et qui suit la même structure que la fonction blolbFile(). Pour différencier l'enregistrement instantané d'un fichier de celui d'un WorkTree, nous ajoutons l'extension ".t" à la fin du hachage du WorkTree.

La fonction récursive saveWorkTree() quand-à-elle permet de sauvegarder un WorkTree en produisant un enregistrement instantané de son contenu. Nous commençons par parcourir le WorkTree. On différencie deux cas : si l'élément courant du WorkTree est un fichier, nous appelons la fonction blobFile() pour créer un instantané, puis nous mettons à jour le hachage et le mode du fichier associé au WorkTree. Sinon, nous avons un WorkFile qui présente un répertoire. Dans ce cas on construit un WorkTree avec ce répertoire en parcourant la liste des fichiers et des répertoires qui le composent, pour ensuite lancer un appel récursif sur ce nouveau WorkTree. On finit par mettre à jour le hachage et le mode du WorkTree.

Enfin, nous avons la fonction restoreWorkTree() qui restaure une version des fichiers et des répertoires associés au WorkTree en paramètre. Si l'élément du WorkTree est affilié au hachage d'un fichier, on copie la version de l'enregistrement instantané dans le fichier. Sinon, si l'élément est associé au hachage d'un autre WorkTree (présence de ".t" en fin de chaîne), nous récupérerons le WorkTree associé à ce hachage et restaurons ce dernier de manière récursive.

```
typedef struct {
  char *name; // nom du fichier
  char *hash; // hash associé à son contenu
  int mode;   // autorisations associés au fichie
} WorkFile;
```

figure 2 : Structures d'un WorkFile

```
typedef struct {
  WorkFile *tab;
  int size; // taille du taleau
  int n;    //  nombre d'élément présent dans le tableau
} WorkTree;
```

figure 3 : Structures d'un WorkTree

III- Gestion des Commits

Dans cette section, on se consacre à la gestion des commits et plus particulièrement à la simulation des commandes *git add* et *git commit* dans le fichier commit.c. Comme nous pouvons à présent créer des enregistrements instantanés de nos fichiers et répertoires, il faut avoir la possibilité de naviguer entre les différentes versions du code source. Pour cela, nous allons organiser de manière chronologique les différents instantanés à l'aide de points de sauvegarde appelés commit. Un commit est associé à l'enregistrement instantané d'un WorkTree. Il peut aussi contenir des informations supplémentaires comme par exemple l'auteur et la date du commit ou encore un message descriptif du commit.

Nous allons implanter un commit par une table de hachage dont les couples *clé : valeur* sont deux chaînes de caractères, représentées par la structure de donnée kvp *(figure 3)*. La structure de commit *(figure 4)* possède un tableau de pointeur de kvp, qui est de taille fixée par la macro SIZE_C, elle contient aussi le nombre d'éléments du tableau. Les couples *clé : valeur* sont insérés à partir de la fonction de hachage « djb2 » que l'on renomme « hash » qui gère les collisions (clé déjà associée à une valeur) par adressage ouvert et probing linéaire. Comme pour les structures précédentes, nous pouvons convertir les commits en chaînes de caractères et en fichiers et inversement. De plus, nous pouvons aussi créer un enregistrement instantané du contenu d'un commit en reprenant le même schéma que les WorkTree mais en ajoutant cette fois-ci l'extension ".c" en fin du chemin de l'instantané.

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

figure 4 : Structures d'un kvp

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

figure 5 : Structures d'un Commit

Afin de se déplacer entre les commits, ces derniers sont organisés en liste chainée qu'on appelle « branche » *(figure 5)*. Pour accéder à une branche et parcourir les commits de cette dernière, nous allons utiliser des pointeurs sur le dernier commit d'une branche, nous nommons ce pointeur référence. Une référence est représentée par un fichier contenant le hachage d'un commit. Il existe deux références indispensables qui se trouvent dans un dossier caché nommé "./refs" : la référence master, qui contient le hachage du dernier commit sur ce qu'on appelle la branche principale et la référence HEAD, qui peut pointer vers n'importe quel commit et permet de se déplacer entre les différents commits. Pour pouvoir naviguer sur les commits de la branche master, on ajoute à chaque commit, exceptés le premier, un couple *(predecessor, hash)* avec le hachage du commit qui le précède dans la branche. Les fonctions relatives aux références sont codées dans le fichier ref.c.

Commençons par présenter la fonction myGitAdd(), qui prend en paramètre le nom d'un fichier ou d'un répertoire. Avant d'effectuer un commit, il faut savoir quels fichier et répertoire ont été modifiés, nous utilisons un fichier caché ".add", nommé zone de préparation, qui représente un WorkTree dans lequel nous allons ajouter nos fichiers et répertoires à sauvegarder. La fonction myGitAdd() débute par créer le fichier .add, s'il n'existe pas, puis elle récupère le WorkTree associé pour lui ajouter un nouvel élément, celui présent dans le fichier .add.

Nous passons maintenant à myGitCommit() qui prend en paramètre la branche sur laquelle nous voulons ajouter un commit. Tout d'abord, il faut que le dossier des références soit initialisé et que la branche existe. De plus, la référence HEAD doit pointer vers le dernier commit de la branche sur laquelle on sauvegarde le commit. Si ces conditions sont vérifiées, nous pouvons récupérer le WorkTree à enregistrer dans le fichier .add, puis exécuter saveWorkTree() pour sauvegarder ce dernier. Enfin, on crée

le commit associer à ce WorkTree et on met à jour les références en écrivant le hachage du commit dans HEAD et dans la branche appelante.

IV- Gestion d'une timeline arborescente

Approfondissons maintenant la structure de notre projet en créant plusieurs branches afin d'obtenir une arborescence de commit, ce qui permettrait à des utilisateurs de pouvoir modifier du code sans affecter tous les projets en créant des versions alternatives. Avec cette nouvelle structure, il faut pouvoir facilement se déplacer entre chaque branche mais aussi entre tous les commits, cette action est simulée par la commande git checkout que nous allons implémenter dans le fichier checkout.c.

Lorsqu'on veut créer une nouvelle branche, on crée une référence qui porte son nom. La référence master représente la branche principale d'un projet. Pour mettre en place de nouvelles branches et créer de nouveaux commits, nous devons savoir sur quelle branche nous nous situons afin de pouvoir faire les commits sur la bonne branche. Pour cela, nous allons utiliser un fichier caché .current_branch qui contient la branche courante. Pour revenir à une version antérieure du code, il faut pouvoir restaurer le point de sauvegarde associé, pour cela, nous utilisons la fonction restoreCommit() permettant d'aller à n'importe quel commit grâce à son hachage pour pouvoir restaurer le WorkTree associé. Nous commençons donc à recréer le commit au moyen de son hash donnée en paramètre, puis on récupère le WorkTree affilié en cherchant la valeur associée à la clé "tree" qui contient le hachage du WorkTree, et enfin on restaure ce dernier.

Comme nous pouvons restaurer n'importe quel commit, nous aurons la possibilité de naviguer facilement entre les branches avec la fonction myGitCheckoutBranch(). Il suffit simplement de modifier la branche courante dans le fichier « .current_branch », puis de récupérer le hachage du dernier commit de la branche sur laquelle nous voulons nous diriger, grâce à la référence associée à la branche, et de mettre à jour la référence HEAD pour avoir la même référence que la branche sur laquelle on se déplace avant de restaurer le commit.

Finalement, pour pouvoir se déplacer librement entre les différentes versions du projet, il faut pouvoir restaurer n'importe quel commit simplement. Cependant, avec la fonction restoreCommit(), il nous faut le hachage entier d'un commit pour pouvoir le restaurer, ce qui est très laborieux. Nous allons donc implémenter la fonction myGitCheckoutCommit() pour simplifier cette action. Tout d'abord, nous devons récupérer tous les commits effectués sur chaque branche : il suffit de parcourir chaque branche avec sa référence présente dans le dossier.refs, puis récupérer tous les hachage de ses commits en se déplaçant via la clé *predecessor*. Comme le hachage est une suite complexe de caractères, il est facile de pourvoir retrouver un commit grâce au début de son hachage en filtrant ceux qui ne correspondent pas. S'il y a plusieurs commits, nous pouvons afficher leur hachage pour pouvoir rappeler la fonction. Enfin on termine par restaurer le commit recherché.

V- Gestion des fusions de branches

Dans cette dernière section, nous allons nous intéresser à la fusion de branche dans le fichier merge.c. Lorsque plusieurs développeurs veulent finaliser leur travail, ils doivent pouvoir réunir les différentes versions de leurs codes en un même endroit. Pour ce faire nous devons pouvoir fusionner et supprimer des branches en créant un commit de fusion. La gestion de la fusion est délicate. En effet, si nous avons deux fichiers avec des contenus différents sur des branches opposées, il faut pouvoir décider quelle version conserver. On dit que ces fichiers sont en conflits : ils possèdent le même nom mais un hachage (contenu) différent. Pour cela, nous allons implémenter une solution simple de résolution de conflits avec l'utilisation d'un commit de suppression. Si nous avons deux versions différentes d'un fichier sur deux branches, on peut effectuer un commit avant la fusion sur l'une des deux branches, avec les fichiers qui ne sont pas en conflit pour pouvoir conserver la version du fichier en conflits de l'autre branche dans le commit de fusion. À noter que ce commit de fusion contiendra la référence vers deux commits, celle du dernier commit de la branche courante et celle de la branche à supprimer qu'on appellera *merged_predecessor*.

Pour implémenter la commande git merge, commençons par le cas sans conflits. Tout d'abord, nous devons récupérer les WorkTree associés au derniers commits de chacune des branches, grâce aux références des branches, puis leurs WorkTree respectifs avec leur hachage à l'aide d'une fonction auxiliaire de récupération de WorkTree à partir de commit ctwt(). Ensuite, nous devons construire le WorkTree de fusion en combinant les WorkFiles de chaque WorkTree, puis créer le commit de fusion en mettant à jour les prédécesseurs, avant de le restaurer. Enfin, il faut supprimer la branche qui a fusionné.

Dans le cas de conflits, la fonction merge peut vérifier si des éléments sont en conflits et en créer une liste. Pour le commit de suppressions, nous laissons à l'utilisateur le choix de la version des fichiers qu'on gardera dans le commit de fusion. Si l'on veut garder les versions des fichiers d'une branche, il suffit de faire le commit de suppression sur l'autre branche. Sinon, l'utilisateur doit pouvoir choisir la version de chaque fichier ou répertoire qu'on conservera. Il faudra donc effectuer un commit de suppressions sur chaque branche. Pour ce faire, nous pouvons passer par une fonction auxiliaire qui, à partir de la liste des conflits, divise cette liste en deux, en demandant à l'utilisateur pour chaque élément en conflits, la branche de la version du fichier conflits à conserver pour pouvoir faire les commits de suppression.

Conclusion

L'ensemble des commandes qui permettent de mettre en œuvre ces fonction est codé dans le fichier myGit.c. L'utilisateur devra compiler le programme en saisissant **make myGit** sur le terminal. Il pourra ensuite saisir une commande de la forme **./myGit [commande]** pour pouvoir créer, sauvegarder et gérer les différentes version de ses fichiers et répertoire. Des répertoires (« test » et « auteurs ») contenant des fichiers et d'autres répertoires sont fournis pour mener les tests. Des exemples de tests sont également fournis dans le répertoire « test_main » avec des exemples de commits, de WorkTrees et de fichiers instantanés.