Avril 2023

RAPPORT DE PROJET

RÉALISATION D'UN LOGICIEL DE GESTION DE VERSIONS

Maïssa Chemali
Sarah Daher

LU2IN006 - Structures de données

SOMMAIRE

PRESENTATION DU SUJET 1

STRUCTURES MANIPULEES 2

Liste chaînée 2
WorkFile et WorkTree 2
Commit 3
Référence et branche 3

DESCRIPTION DU CODE 4

Organisation 4
Description des fonctions principales 4

REMARQUES 10

Fonctions rajoutées 10
Problèmes rencontrés 11

CONCLUSION 12

Présentation du sujet

Un logiciel de gestion de versions est un outil informatique permettant de stocker et de gérer un ensemble de fichiers et répertoires et de suivre leur évolution. Il consiste en la maintenance et la construction d'une arborescence claire entre les différentes versions d'un même projet. En particulier, un tel outil est très utile pour deux raisons :

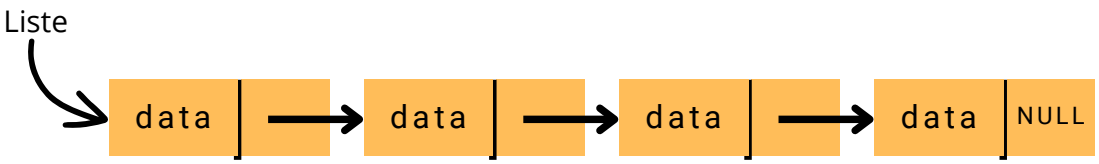
- Il permet de restaurer une version précédente d'un fichier par la tenue d'un historique des sauvegardes, ce qui peut s'avérer nécessaire lors d'une modification apportant des erreurs.

- Il facilite également la collaboration de plusieurs personnes sur un même projet en créant une nouvelle version du fichier à chaque sauvegarde effectuée. Il est ensuite possible de choisir quelle version conserver en fonction des besoins.

Ce projet consiste à étudier plusieurs structures de données mises en œuvre dans un logiciel de gestion de versions, et à implémenter plusieurs commandes caractéristiques de ce type d'outil.

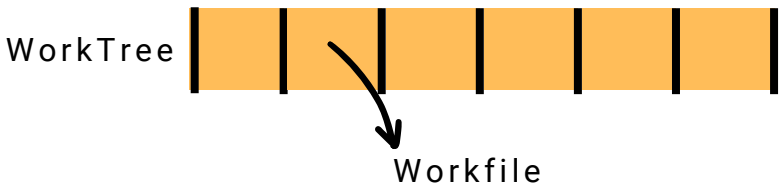
Structures manipulées

1 - Liste chaînée



Une liste chaînée est une structure de données définie par un ensemble de cellules, possédant chacune un élément qui pointe sur un autre élément de la liste, de sorte à former une chaîne linéaire.
Ainsi une liste chaînée est un pointeur vers une cellule : sa tête de liste.
Dans le contexte de ce projet, chaque cellule est caractérisée par une chaîne de caractères ("data") et un pointeur vers la cellule suivante ("next").

2 - WorkFile et WorkTree

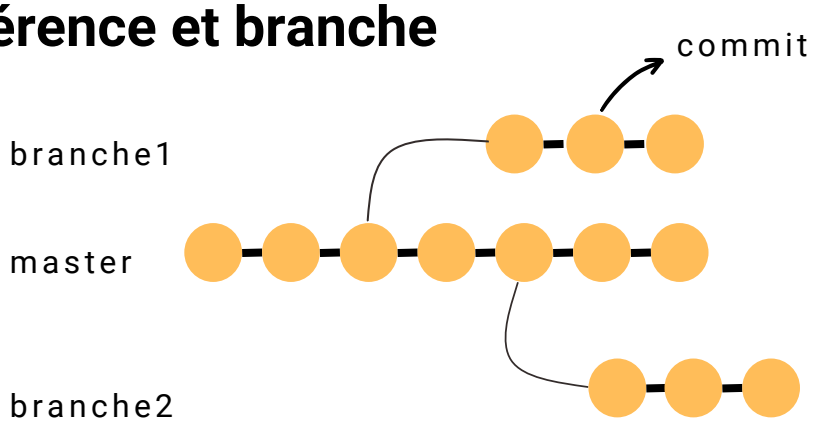


Un WorkFile est une structure représentant un fichier ou un répertoire dont on souhaite créer un enregistrement instantané. Il contient les informations suivantes : le nom du fichier ou du répertoire ("name"), le hash associé à son contenu obtenu grâce à la fonction sha256 ("hash"), ainsi que le mode représentant les droits d'accès du fichier ("mode").
Un WorkTree est une structure composée d'un tableau de Workfiles ("tab"), de sa taille("size"), et de son nombre de cases occupées("n").

3 - Commit

Un commit est une structure associée à l'enregistrement instantané d'un WorkTree à une étape considérée importante dans l'implémentation du projet. Il est représenté par une table de hachage contenant des paires (clé, valeur), notamment la paire ("tree", hash) où hash est le hash du fichier associé à l'enregistrement instantané du WorkTree, ainsi que d'autres paires sélectionnées en fonction des besoins, telles que celles représentant l'auteur, le message ou le prédécesseur.
La clé "predecessor" permet une organisation des commits en liste chainée : à partir d'un commit donné on peut accéder à son prédécesseur s'il en possède un.

4 - Référence et branche



Les références sont des fichiers contenant le hash d'un commit ; elles sont stockées dans le répertoire caché .refs. En pratique, une référence est donc un pointeur vers un commit.
Le répertoire .refs contient au minimum les deux références suivantes :

- HEADER : contient le hash du dernier commit de la branche courante. Il pointe donc vers la version actuelle du code dans une branche donnée.
- master : contient le hash du dernier commit de la branche principale. Il pointe donc vers la version fiable la plus récente du projet.

Une branche est une référence qui évolue au fur et à mesure que de nouveaux commits sont ajoutés à cette branche. Elle permet de garder une trace de l'évolution d'une ligne de développement spécifique. Cette structure en arborescence permet à plusieurs utilisateurs de travailler simultanément sur des branches différentes.

Description du code

1 - Organisation

Notre code est divisé en plusieurs fichiers .c, chacun regroupe un ensemble de fonctions conçues pour manipuler une certaine structure de données. Ainsi, un fichier .c est dédié à chacune des structures présentées plus haut et son nom lui correspond (du type : nomStructure.c). Les fichiers du type test_nomStructure.c représentent des tests écrits pour vérifier le bon fonctionnement des fonctions relatives à chaque structure.
Le fichier "merge.c" contient les fonctions impliquées dans la fusion des branches. Enfin, le fichier exos.h regroupe les signatures de toutes les fonctions du projet, et on dispose d'un Makefile qui permet de faciliter la compilation.
Par ailleurs, le fichier myGit.c contient le code principal qui permet à un utilisateur de manipuler son projet. L'exécution de ce fichier offre à l'utilisateur un contrôle sur la gestion des commits et des branches en lui permettant d'exécuter les commandes de son choix et assure également une meilleure collaboration entre les membres d'une équipe. Pour avoir la documentation complète de ces commandes tapez ./myGit help dans le Terminal.

2 - Description des fonctions principales

- **hashToPath** : char* hashToPath(char* hash);

  Prend une chaîne de caractères représentant un hash en paramètre et renvoie cette chaîne après y avoir inséré le caractère '\' entre le deuxième et le troisième caractère.

- **blobFile :** void blobFile(char* file);

  Enregistre un instantané du fichier "file" donné en paramètre. Pour réaliser cette tâche, elle suit le protocole suivant :
  - hache le contenu du fichier (sha256).
  - transforme ce hash en un chemin en utilisant la fonction hashToPath.
  - crée un répertoire portant les deux premières lettres du chemin.
  - copie le contenu du fichier "file" dans un nouveau fichier ayant comme nom le reste du chemin.
  - retourne le hash pour permettre de localiser cet enregistrement.

- **blobWorkTree :** char* blobWorkTree(WorkTree* wt)

  Crée un enregistrement instantané du WorkTree "wt" donné en paramètre. Pour ce faire, elle procède ainsi :
  - crée un fichier temporaire et y stocke le contenu du WorkTree (fonction wttf), puis hache son contenu.
  - récupère le chemin et crée le répertoire (fonction hashToFile), puis ajoute l'extension ".t" au chemin afin d'identifier le WorkTree.
  - copie le contenu du fichier temporaire dans le fichier correspondant au chemin.
  - retourner le hash.

- **blobCommit :** char* blobCommit(Commit *c)**;**

  Crée un enregistrement instantané du commit passé un paramètre. Cette fonction procède de la même manière que blobWorkTree, à l'exception de l'extension rajoutée : ce n'est pas ".t" mais ".c".

- **saveWorkTree :** char *saveWorkTree(WorkTree *wt, char *path);

  Crée récursivement un enregistrement instantané de tout le contenu du WorkTree "wt" donné en paramètre. Pour ce faire, elle parcourt un à un tous les éléments du WorkTree et pour chacun procède ainsi :
  - construit son chemin d'accès final en concaténant son chemin d'accès et son nom.
  - si ce WorkFile est un fichier :
    - stocke son mode d'accès et son hash à la place correspondante dans le WorkTree.
    - crée un instantané de ce fichier à l'adresse indiquée par le chemin total (fonction blobFile).
  - sinon c'est un répertoire :
    - crée un nouveau WorkTree et y ajoute les fichiers et les sous-répertoires (non cachés) présents dans le WorkFile.
    - stocke son mode d'accès et son hash à la place correspondante dans le WorkTree, et appelle pour obtenir le hash la fonction saveWorkTree sur le nouveau WorkTree créé, avec en paramètre le chemin d'accès final obtenu en première étape.
  - dans tous les cas, la fonction retourne le hash.

- **restoreWorkTree :** void restoreWorkTree(WorkTree *wt, char *path);

  Restaure récursivement l'arborescence des fichiers et répertoires du WorkTree "wt" donné en paramètre, sauvegardé précédemment à l'aide de la fonction "saveWorkTree". Pour cela, elle parcourt un à un tous les éléments du WorkTree et pour chacun opère ainsi :
  - récupère son chemin d'accès absolu (fonction hashToPath) où se trouve sa dernière sauvegarde, et construit le chemin destination en concaténant le chemin d'accès passé en paramètre avec son nom.
  - si ce WorkFile est un fichier, donc son chemin ne contient pas ".t" :
    - copie le contenu de sa dernière sauvegarde dans le chemin de destination.
  - sinon c'est un répertoire, donc son chemin est celui d'un WorkTree :
    - ajoute l'extension ".t" au chemin d'accès absolu.
    - restaure le contenu du répertoire en appelant la fonction restoreWorkTree sur le WorkTree représentant son contenu (fonction ftwt).
  - dans tous les cas, la fonction ne retourne rien.

- **myGitAdd :** void myGitAdd(char *file_or_folder);

  Ajoute l'élément passé en paramètre dans le WorkTree correspondant à la zone de préparation (fichier ".add"). Pour ce faire, elle procède ainsi :
  - si le fichier ".add" n'existe pas, on le crée et on initialise un nouveau WorkTree ; sinon, on transforme ".add" en WorkTree (fonction ftwt).
  - si l'élément passé en paramètre est un fichier ou un répertoire existant, on l'ajoute dans le WorkTree (fonction appendWorkTree) puis on écrit dans la zone de préparation la chaîne de caractères qui correspond au WorkTree modifiée (fonction wttf) ; sinon, on affiche un message d'erreur.
  - dans tous les cas, on ne retourne rien et on termine.

- **myGitCommit :** void myGitCommit(char *branch_name, char *message);

  Crée un point de sauvegarde dans une branche passée en paramètre. Elle met en œuvre la méthode suivante pour y parvenir :
  - si le fichier caché ".add" représentant la zone de préparation n'existe pas, on le crée.
  - récupère le WorkTree correspondant au fichier ".add" (fonction ftwt), et l'enregistre (fonction saveWorkTree).
  - crée un commit dont la valeur associée à la clef "tree" est le hash du WorkTree récupéré (fonction createCommit).
  - La branche passée en paramètre contient le hash du dernier commit de la branche, donc du prédécesseur du nouveau commit créé. Ainsi si le hash n'est pas NULL, on remplit le champ prédécesseur du nouveau commit. On remplit également le champ message avec le message donné en paramètre si il n'est pas NULL (fonction commitSet).
  - crée un enregistrement du nouveau commit (fonction blobCommit) et met à jour les références (fonction createUpdateRef).
  - vide la zone de préparation.
  - ne retourne rien et termine.

- **myGitCheckoutBranch :** void myGitCheckoutBranch(char *branch);

  Définit comme branche courante la branche passée en paramètre. Pour ce faire, elle procède ainsi :
  - écrit dans le fichier représentant la branche courante (".current_branch") le nom de la branche passée en paramètre.
  - récupère le hash du commit vers lequel cette branche pointe et le met dans la référence "HEAD" (fonction createUpdateRef).
  - restaure ce commit .

- **myGitCheckoutCommit :** void myGitCheckoutCommit(char *pattern);

  Permet d'accéder à un commit en passant un préfixe de son hash en paramètre. Pour réaliser cela, elle suit la procédure suivante :
  - construit la liste contenant tous les commits dont le hash commence par la chaîne de caractères pattern donnée en argument (fonction filterList).
  - s'il ne reste aucun hash : on a une erreur et on termine.
  - sinon s'il ne reste qu'un seul hash : on met à jour la référence "HEAD" (fonction createUpdateRef), on restaure le commit correspondant (fonction restoreCommit) et on termine.
  - sinon s'il reste plusieurs hashs : on les affiche (fonction ltos), on demande à l'utilisateur de préciser sa requête, et on termine.

- **restoreCommit :** void restoreCommit(char* hash_commit);

  Restaure le WorkTree associé au commit dont le hash est donné en paramètre. Pour ce faire, elle procède ainsi :
  - récupère le chemin du commit à partir de son hash (fonction hashToPathCommit).
  - crée le commit à partir de son chemin (fonction ftc).
  - récupère le hash du WorkTree associé par la clé "tree", puis construit son chemin.
  - crée le WorkTree associé à partir de son path (fonction ftwt).
  - restaure ce WorkTree (fonction restoreWorkTree).

- **mergeWorkTrees :** WorkTree* mergeWorkTrees(WorkTree* wt1, WorkTree* wt2, List** conflicts);

  Construit et renvoie un WorkTree composé des éléments des deux WorkTrees passés en entrée qui ne sont pas en conflit Les éléments en conflit sont ajoutés à une liste donnée en paramètre. Pour effectuer cette opération, elle parcourt tous les éléments du premier WorkTree et teste pour chacun son éventuelle présence dans le second. S'il est présent dans les deux WorkTrees, elle teste alors si les hashs sont identiques ; si c'est le cas, cet élément est en conflit et on l'ajoute à la liste. Sinon, on l'ajoute au nouveau WorkTree. On parcourt ensuite le second WorkTree : pour chacun de ses éléments, si il n'est pas présent dans le premier WorkTree, on l'ajoute au nouveau WorkTree. Enfin, on retourne le nouveau WorkTree.

- **merge :** List *merge(char *remote_branch, char *message);

  Fusionne la branche courante avec la branche passée en paramètre à condition qu'il n'y ait aucun conflit. Pour ce faire, elle procède ainsi :
  - construit deux WorkTrees, l'un associé à la branche courante et l'autre à celle passée en paramètre (fonction branchToWorkTree).
  - crée un WorkTree de fusion temporaire avec mergeWorkTrees.
  - si aucun conflit n'est détecté, alors le WorkTree créé est bien le WorkTree de fusion :
    - on l'enregistre (fonction blobWorkTree), on crée le commit associé (fonction createCommit) et on indique ses prédécesseurs et son message.
    - on enregistre ce commit de fusion (fonction blobCommit).
    - mise à jour des références.
    - on restaure le WorkTree de fusion (restoreWorkTree) et on retourne NULL.
  - sinon, on retourne la liste des conflits

- **createDeletionCommit :** void createDeletionCommit(char *branch, List *conflicts, char *message);

  Crée et ajoute un commit de suppression sur la branche passée en paramètre. Ce commit correspond à la suppression des éléments présents dans la liste des conflits. Elle adopte la démarche suivante pour y parvenir :
  - se déplace sur la branche donnée en paramètre (fonction myGitCheckoutBranch).
  - récupère le worktree associé au dernier commit de cette branche (fonction branchToWorkTree).
  - vide la zone de préparation si elle existe.
  - parcourt les éléments du WorkTree : chaque élément qui n'est pas un conflit est ajouté à la zone de préparation (fonction myGitAdd).
  - crée le commit de suppression à partir de tous les éléments présents dans la zone de préparation (fonction myGitCommit).
  - se replace sur la branche d'origine.
  - ne retourne rien et termine.

Remarques

1 - Fonctions rajoutées

Nous avons implémentées quelques fonctions qui n'étaient pas exigées mais qui nous ont paru nécessaires pour une bonne lisibilité du code :

- **hashToFile:** char *hashToFile(char *hash)

  Crée, s'il n'existe pas déjà, un répertoire ayant pour nom les deux premiers caractères du hash passé en argument. Renvoie le chemin pour y accéder.

- **isDirectory :** int isDirectory(const char *path);

  Vérifie si le chemin indiqué en argument est celui d'un répertoire ou non. Retourne 1 si c'est le cas.

- **concat_paths :** char *concat_paths(char *path1, char *path2);

  Concatène les deux chemins passés en paramètre en formant une chaîne de caractères de la forme : path1/path2. Renvoie cette chaîne ainsi formée.

- **hashToPathCommit :** char *hashToPathCommit(char *hash);

  Prend le hash d'un commit en paramètre et retourne le path associé.

- **branchToWorkTree :** WorkTree *branchToWorkTree(char *branch);

  Prend le nom d'une branche en paramètre et retourne le WorkTree associé au commit sur lequel elle pointe.

Nous avons également rajouter des fonctions permettant de libérer la mémoire :

- **freeList :** void freeList(List *l);
- **freeWorkTree :** void freeWorkTree(WorkTree *wt);
- **freeCommit :** void freeCommit(Commit *c);

Enfin, des fonctions permettant l'affichage de certaines structures, qui nous ont été utiles pour vérifier notre code:

- **printList :** void printList(List *L);
- **afficherWorkFile :** void afficherWorkFile(WorkFile *wf);
- **afficherWorktree :** void afficherWorkTree(WorkTree *wt);

2 - Problèmes rencontrés

Nous avons relevé une incohérence dans les fonctions fournies getChmod et setMode de l'exercice 5. En effet, la fonction getChmod retourne un nombre octal tandis que la fonction setMode fournie dans le sujet du projet exécute une commande qui manipule un nombre décimal. Nous avons rencontré des erreurs car les fichiers n'avaient pas les autorisations nécessaires pour être ouverts et modifiés. Il nous a donc paru nécessaire pour poursuivre le projet de convertir dans un sens où dans l'autre afin de ne manipuler qu'un même type d'entier. Ainsi, après modification notamment de la fonction setMode (où l'on a remplacé les %d par des %o, afin de manipuler un octal), nous n'avons plus rencontré ces erreurs d'ouverture. Nous avons subi une perte de temps considérable en raison de cette incohérence, car nous avons dû consacrer du temps à comprendre son origine, étant donné que nous n'avions pas implémenté ces fonctions nous-mêmes.

Nous avons également consacré beaucoup de temps pour déboguer le programme et en supprimer les nombreuses fuites mémoires. Nous sommes satisfaites du résultat car à présent notre code fonctionne correctement et sans aucune fuite mémoire. Ce travail a été essentiel pour livrer un projet qui répond à nos attentes.

Par ailleurs, la référence "master" n’a jamais été exploitée dans le cadre du projet ce qui nous paraît regrettable car c'est une composante essentielle de git.

Conclusion

Pour finir, ce projet a été une expérience enrichissante pour nous. Tout au long, nous avons mis en pratique nos connaissances en matière de programmation et de structure de données dans le but de créer notre propre système de gestion de versions en utilisant les concepts fondamentaux de Git, tels que les commits, les branches, les fusions et les références.
Malgré quelques difficultés, nous avons réussi à relever au maximum ces défis grâce à notre travail d'équipe. Nous avons également réalisé l'importance de la qualité du code et de la documentation, et avons mis en place des tests et des commentaires pour garantir sa fiabilité et sa lisibilité.
Nous sommes fières du résultat final, qui témoigne des efforts que nous avons déployé tout au long du semestre pour comprendre en profondeur le fonctionnement interne de Git et pour développer notre propre implémentation.