Rapport du projet
“Réalisation d’un logiciel de gestion de versions”

LU2IN006

OULFID Hamza

LAHBIB Yassin

[Introduction]

Le but de ce projet était d'étudier le fonctionnement d'un logiciel de gestion versions (git) pour avoir nos propres commandes pour gérer nos fichiers locaux.

[Présentation du projet]

[Structuration des dossiers]

Nous avons structuré nos fichiers de la manière suivante :

- **include** contient tous les fichiers .h
- **src** contient le code source du projet
- **bin** contient tous les fichiers objet nécessaire à notre myGit

[Gestion du code]

- ### instantane

La première structure que nous avons utilisée est cell :

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

typedef Cell* List;
```

Nous avons codé des fonctions de gestion de cette structure comme : insertFirst,ctos,listGet,searchList,stol,ltof,ftol.

La fonction principale de cette partie est **void blobFile(char* file)** qui permet de faire un enregistrement instantané du fichier en paramètre elle utilise **char* hashToPath(char* hash)** qui retourne le chemin d'un fichier à partir de son hash. Un enregistrement instantané correspond à une sauvegarde à un instant donné du fichier, pour se faire nous avons créé une fonction **blobfile** qui prend un fichier en entrée, hash son contenue et copie l'intégralité du fichier dans le chemin de son hash.

- ### Worktree

Les structures utilisé pour cette partie son Workfile et WorkTree:

```
typedef struct {
    char* name;
    char* hash;
    int mode;
} WorkFile;

typedef struct {
    WorkFile* tab;
    int size;
    int n;
} WorkTree;
```

Un WorkTree représente un tableau de WorkFile ,un Workfile étant une structure représentant le nom, le hash du contenue et le mode du fichier. Comme pour les Listes nous avons définie plusieurs fonctions de gestion de ces deux structures qui vont nous permettre de les manipuler.

Les fonctions principales sont :

char* blobworkTree(WorkTree* wt) qui fonctionne comme blobFile mais rajoute l'extension .t .

char* saveWorkTree(WorkTree* wt , char* path) cette fonction permet de sauvegarder un espace de travail associé au workTree, puis calcul son hash. Si elle trouve un répertoire dans wt, elle appelle à nouveau cette fonction et va parcourir le contenu de ce nouveau répertoire.

void restoreWorkTree(WorkTree* wt ,char* path) cette fonction parcoure le tableau de wt et pour chaque workFile si n'est pas un .t on créer une copie à l'endroit du PATH sinon (c'est un répertoire) on créer un nouveau workTree on change le PATH puis on appel de nouveau la fonction pour parcourir le nouveau workTree.

- ## commit

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

Cette structure nous permet de manipuler des paires de (key,value), afin de pouvoir s'en servir dans la gestion de nos commit.

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

typedef  HashTable  Commit;
```

Un commit est représenté par la structure **HashTable**, qui contient un tableau de (key,value) représenté par la structure kvp précédente. Cette structure permet de sauvegarder un espace de travail associé à un certain WorkTree en ajoutant comme valeurs à la clef "tree" de chaque commit, le chemin d'accès au WorkTree associé.On y garde également une trace du commit précédent, ainsi qu'une description pour pouvoir les distinguer.

myGitAdd(char* file_or_folder) met à jours la zone de préparation représenter par le fichier caché .add, étant lui meme la représentation fichier d'un WorkTree

myGitCommit(char* branch_name, char* message) pour pouvoir effectuer un commit, nous sauvegardons le WorkTree associé à la zone de préparation, en récupérant son hash que nous stockons en tant que valeurs à la clé "tree".

Si la branche_name pointe déjà sur un commit, nous le récupérons et le stockons en tant que prédécesseur du nouveau commit, sur lequel la branche pointera.

- ## branch

Dans cette partie nous allons introduire les branches, pour les manipuler on utilise 2 répertoires importants qui sont .current_branch qui contient le nom de la branch courante et .refs qui pour chaque branch va contenir le hash du dernier commit ainsi que HEAD qui contient le hash d'un commit.

void restoreCommit(char* hash_commit) permet de restaurer un worktree a un commit donnée en allant chercher la version correspondant au commit grâce à son hash.

Void myGitCheckoutBranch(char* branch) permet de basculer d'une branch a une autre en modifiant le nom du fichier dans .current_branch avec le nom de la branch souhaité et HEAD va contenir le hash du dernier commit de branch puis on va restore le commit avec le reference contenu dans HEAD

List* filterList(List* L, char* pattern) va prendre un pattern puis va comparer avec tous les éléments de la L si leur hash contient le pattern puis va prendre tous les éléments qui contiennent le pattern pour les mettre dans une liste.

void myGitCheckoutCommit(char* pattern) fonctionne comme myGitCheckoutBranch sauf qu'ici c'est nous qui choisissons à l'aide du pattern à quel commit le worktree se restaure, puis on met à jour la ref de HEAD.

- ## merge

Nous possédons désormais plusieurs branches sur notre projet. Afin de pouvoir les fusionner, nous avons besoin de certaine fonction manipulant des commits, ainsi que des branches

WorkTree* mergeWorkTrees(WorkTree* wt1, WorkTree* wt2, Listconflicts) p**ermet de parcourir les deux WorkTree, créant ainsi le WorkTree de retour possédant les WorkFiles non conflictuels, tels que leurs contenue hashé est identique ou étant présent seulement dans une branche.

Pour les WorkFiles conflictuels on les ajoute dans la list des conflits passé en paramètre pouvant être analysé par la suite.

List* merge(char* remote branch, char* message) permet de fusionner la branche courante à la branche passée en paramètre, en récupérant les derniers WorkTree des deux branches, fais un appele a mergeWorkTrees qui retourne le WorkTree possédant tous les éléments non conflictuels.

Si des conflits sont existant nous allons effectuer un commit de suppression sur la branche à laquelle nous voulons supprimer les fichiers conflictuels.

- Si des conflits sont constatés la fonction retourne la liste des conflits permettant ensuite à createDeletionCommit d'opérer

- Sinon On restaure le WorkTree de fusions, puis on commit dessus en lui associant les deux commit des deux branches comme prédécesseurs

void createDeletionCommit(char* branch, List* conflicts, char* message) permet de faire le commit de suppression sur la branch passé en paramètre.

-Si la branch en question est la branche courante, alors on vide la zone de préparation, pour pouvoir la remplir des WorFiles n'étant pas conflictuels, pour ensuite procéder à un commit de cette zone, permettant de ne plus avoir les workFiles conflictuels.

-Sinon on se déplace sur la branch pour récupérer son dernier workTree puis faire un commit des workFiles n'étant pas conflictuels.

- ## myGit

Ce fichier est notre main qui nous permet de mettre en relation toutes les fonctions vues au cours du projet, il nous permet de créer notre exécutable grâce auquel on lance les différentes commandes liées aux fonctions que nous avons implémentées.

./myGit --help pour plus d'informations