RAPPORT: Logiciel de gestion de version

Yanis BAOUCHE – Xinyu CHEN

Résumé

Ce projet consiste en la création d'un outil de suivi et de version de code similaire à Git, qui est développé en plusieurs parties et exercices. Ce projet vise à simuler certaines fonctionnalités de Git, en mettant en œuvre des structures de données et des algorithmes pour simuler des commandes Git telles que "git add", "git commit" et "git checkout".

L'outil permet de stocker, suivre et gérer plusieurs versions d'un projet. Les différentes fonctionnalités du logiciel incluent la création d'enregistrements instantanés de projet, la navigation entre les différentes versions, la construction et la maintenance d'une arborescence des différentes versions, la vérification de l'identité des utilisateurs et la sauvegarde des changements qui ne sont pas dans un instantané.

Sous Git, tous les objets, y compris les fichiers et les métadonnées, sont enregistrés sous forme de fichiers avec un chemin dérivé de leur contenu. La première partie du projet consiste à écrire un programme permettant d'enregistrer un instantané, où le chemin du fichier est modifié en fonction de son contenu pour sauvegarder différentes versions du fichier. Les autres parties du projet incluent la gestion de plusieurs instantanés, la gestion des commits, la gestion d'une timeline arborescente et la gestion des fusions de branches.

RAPPORT 3

Le projet est organisé en plusieurs dossiers pour une meilleure compréhension et une gestion simplifiée. Le code est réparti en différentes sections, une pour chaque exercice. Les dossiers principaux sont les suivants :

- **src** : Ce dossier contient tous les fichiers source (.c) pour chaque exercice ainsi leur fonctions main pour les tester. Chaque fichier implémente les fonctions et les éléments spécifiques à cet exercice.

- **obj** : Ici se trouvent les fichiers objet (.o) créés lors de la compilation des fichiers source. Ces fichiers sont utilisés pour construire le programme final exécutable.

- **headers** : Ce dossier contient tous les fichiers d'en-tête (.h) pour chaque exercice. Ces fichiers définissent l'organisation des fonctions et des structures de données dans les fichiers de code. Chaque fichier d'en-tête inclut celui de l'exercice précédent, garantissant ainsi une cohérence et une connexion entre les exercices.

- **exe** : Ce dossier contient le programme final (.exe) obtenu à partir des fichiers objet. Il est utilisé pour tester et exécuter les différentes fonctionnalités du code.

- **TestFiles** : Dans ce dossier, on retrouve les fichiers de test au format texte (.txt) qui servent à vérifier le bon fonctionnement des différentes fonctionnalités implémentées dans le code.

Pour chaque exercice, il y a un fichier de code (.c), un fichier d'en-tête (.h), un fichier objet (.o) et un fichier main associé. Le fichier main permet de tester les fonctionnalités développées dans le fichier de code pour cet exercice.

Un fichier caché ".add" représente la zone de préparation (staging area), où les fichiers modifiés sont ajoutés avant d'être commités. Le fichier ".add" correspond à un WorkTree, initialement vide, rempli progressivement par l'utilisateur.

La fonction myGitCommit(char* branch_name, char* message) permet de simuler la commande "git commit", en créant un point de sauvegarde des modifications. Elle effectue plusieurs vérifications, charge le WorkTree du fichier ".add", crée un commit avec les modifications et met à jour les références HEAD et branch_name.

Le projet gère une timeline arborescente avec plusieurs branches pour suivre l'évolution d'un proje. Le fichier caché ".current_branch" stocke le nom de la branche courante et facilite la navigation entre les différentes branches.

La fonction myGitCheckoutBranch(char* branch) simule la commande "git checkout" pour naviguer entre les branches. Elle modifie le fichier ".current_branch", met à jour la référence HEAD et restaure le WorkTree correspondant au dernier commit de la branche.

La fonction myGitCheckoutCommit(char* pattern) simule également la commande "git checkout", mais permet de retourner à n'importe quelle version d'un projet en utilisant le hash d'un commit.

Chaque exercice possède une description brève, de son objectif et de ses arguments.

En résumé, nous avons essayé d’organiser le code de manière claire, avec une séparation bien définie entre les différentes parties et exercices. Cela nous a permis d’avoir une meilleure compréhension du code et une modification ou l'ajout de fonctionnalités au projet plus simple.

RAPPORT 4

Description des structures et description globale du code

RAPPORT 5

Description des fonctions principales et de nos choix d'implémentation.

Les fonctions les plus importantes de ce projet sont les suivantes :

- initRefs() - Initialise le répertoire de références et crée les fichiers master et HEAD (vides).
- createUpdateRef(char* ref_name, char* hash) - Met à jour une référence en remplaçant son contenu par le hash fourni. Si la référence n'existe pas, la fonction crée d'abord le fichier.
- getRef(char* ref_name) - Récupère le hash vers lequel pointe une référence.
- myGitAdd(char* file_or_folder) - Ajoute un fichier ou un répertoire au WorkTree correspondant à la zone de préparation (staging area).
- myGitCommit(char* branch_name, char* message) - Réalise un commit sur une branche avec ou sans message descriptif.

Dans ce projet, les structures WorkTree et WorkFile sont utilisées pour représenter l'état actuel du projet (les fichiers et répertoires modifiés) ainsi que les modifications apportées depuis le dernier commit. Les structures de données choisies sont importantes pour simuler efficacement les commandes Git et gérer les différentes versions du projet.

La structure WorkFile représente un fichier individuel dans l'arborescence du projet. Elle contient le nom du fichier (name), le hash du contenu du fichier (hash) et le mode d'accès au fichier (mode). Cette structure permet de stocker les informations essentielles sur chaque fichier et de les utiliser pour simuler les commandes Git.

La structure WorkTree est une structure qui contient un tableau de pointeurs vers des objets WorkFile (tab), la taille actuelle du tableau (size) et le nombre de fichiers actuellement stockés dans le tableau (n). Cette structure est utilisée pour gérer l'ensemble des fichiers modifiés dans le projet et pour simuler les opérations sur l'ensemble du projet, telles que les commits.

La structure kvp (key-value pair) et la structure HashTable sont utilisées pour créer une table de hachage. Dans ce projet, la table de hachage peut être utilisée pour stocker des informations sur les commits, les branches, ou d'autres éléments qui nécessitent une recherche rapide en fonction d'une clé (par exemple, le hash d'un commit).

En résumé, les choix d'implémentation des structures WorkTree, WorkFile, Cell, List, kvp et HashTable sont faits pour faciliter la simulation des commandes Git, la gestion de l'arborescence du projet, et la manipulation des différentes versions du projet.