Projet Structure de données

(BAMHAOUD Yassine - NECHBA Mohamed)

https://www.github.com/BigYass/projet_scv

I. Sujet

Durant ce projet nous avons possédé à la réalisation d'un outil de suivi et de visionnage de code type git. Il permet aux développeurs de travailler en équipe sur un même projet en suivant l'historique des modifications apportées aux fichiers de code source.

Un système de Git offre plusieurs fonctionnalités essentielles pour la gestion du code source. Tout d'abord, il permet de créer des branches, qui sont des copies indépendantes du code source permettant de travailler sur des fonctionnalités, des corrections de bugs ou d'autres modifications de manière isolée. Les branches peuvent être fusionnées pour combiner les modifications de différentes branches en une seule.

Git offre des fonctionnalités de sauvegarde, car chaque copie du référentiel contient l'historique complet des modifications, ce qui permet de récupérer des versions antérieures du code source en cas de perte de données.
L'objectif de ce projet est d'étudier le fonctionnement d'un logiciel de gestion de versions, en détaillant différentes structures de données impliquées dans sa mise en œuvre.

II. Structures abordées

1. Cell - Cellule d'une liste

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

Description : Cell est une structure de données de type liste chaînée qui traite des chaînes de caractères. Cette structure va servir à la gestion des instantanés de fichiers et entre autres pouvoir retrouver le chemin d'un fichier à partir de son hash.

2. WorkTree - Représentation d'un fichier

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

Description : Un Workfile représente un fichier ou répertoire dont on souhaite enregistrer un instantané. C'est une structure simple qui possède trois champs : le nom du fichier, son hash et les autorisations.

3. WorkTree - Représente un répertoire de travail

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

Description : Un WorkTree est simplement un tableau dynamique de Workfile.

Les deux prochaines structures vont servir à la gestion des points de sauvegarde de fichier (commits) qui sont associ´e `a l'enregistrement instantanee d'un WorkTree, accompagnée d'autres informations relatives au point de sauvegarde comme par exemple l'auteur du commit

4. kvp - Une paire de deux chaîne de caractères

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

Description : kvp permet de gérer les clés et valeur (qui sont des chaînes de caractère) d'une table de hachage (voir ci dessous) que représente le commit.

5. HashTable - Table de hachage (dictionnaire)

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

Description : Tout simplement une table de hachage qui va donc permettre de creer et gérer les commits.

III. Structure du projets

Structure générale (Arborescence)

. Racine du projet
├── **bin** Dossier contenant le binaire
├── **obj** Dossier contenant les .o du projets
│ ├── **org** Objet du package 'org'
│ ├── **test** Objet du package 'test'
│ └── **util** Objet du package 'util'
└── **src** Code source du projet (contient les .c)
├── **include** Contient les headers du projets
│ ├── **org** Header du package 'util'
│ ├── **test** Header du package 'util'
│ └── **util** Header du package 'util'
├── **org** Code du package 'util'
├── **test** Code du package 'util'
└── **util** Code du package 'util'

Structure du code (explication)

src/*.c

Dans le dossier src/ il y a les code source "généraux". Ici il y a seulement le fichier myGit.c contenant le main du projet qui prend en charge les différentes commandes de l'exécutable final.

src/util/*.c

Dans le dossier src/util, on a les codes appelés d'utilité générale. Le fichier file.c prend en charge les fonctions de manipulation de fichier tels cp. Le fichier hash.c contient les différentes fonctions de hachage. Le fichier liste.c contient les fonctions de manipulation des listes. Tool.c contient des fonctions d'utilités générales tels qu'un générateur de chaîne de caractère. Ces fonctions sont utiles notamment pour le test des autres fonctionnalités.

src/org/*.c

Dans le dossier src/org, on a les codes appelés d'organisation. Le code permet notamment l'organisation des branches, commit ou workfile/worktree. Le fichier branch.c contient les fonctionnalités de manipulation des branches. Le fichier commit.c contient les fonctionnalités de manipulation des commits (qui sont des Table de hachage déguisé). Le fichier my_git.c contient les fonctions de manipulation du projet. Le fichier refs.c contient les fonctionnalités de manipulation des références (Ce qui est contenu dans le dossier .refs). Enfin le fichier workfile.c permets la manipulation des WorkTree qui sont une représentation du répertoire de travail du projet. Les WorkTree sont très utilisés pour la manipulation des différends commits.

src/test/*.c

Ce dossier contient les fonctionnalités pour test et debug l'exécutable. Aucune de ces fonctionnalités est utilisée dans le "release" (la version donnée au client). Le fichier debug.c contient une seule fonction err_logf(params) qui un printf qui se comporte différemment selon la valeur de la macro DEBUG. Test.c quant à lui contient des fonctions utiles pour tester plusieurs fonctions rapidement. Elle a été très utile pour détecter les fuites de mémoire et les fautes de segmentation.

src/include

Ce dossier contient les headers du fichier ce qui signifie que dans notre projet les .c et .h sont complètement séparés. Le fichier const.h contient des macros utilisés comme constantes. La plus importante est DEBUG qui définit la verbose du programme. Chaque macro est commenté pour plus d'information..

IV. Fonctions

Intéressons nous à notre fonction char* saveWorkTree(WorkTree* wt, char* path) qui permet de créer un enregistrement instantané de l'entièreté du contenu d'un worktree passé en paramètre.
On commence par une allocation dynamique de la mémoire pour la variable full_path et on choisit de l'initialiser à 0 avec la fonction memset().
Dans full_path on copiera le chemin d'accès et le nom du fichier.
On vérifie ensuite que le fichier existe dans le worktree puis on y écrit le hachage du fichier et le mode.
On a choisi de stocker les fichiers contenus dans le répertoire du fichier en cours de traitement à l'aide de la fonction appendWorkTree() dans un worktree précédemment créé pui on rappelle récursivement saveWorkTree() pour obtenir le hachage du sous-arbre.
Après avoir fini de traiter chaque fichier, la mémoire est libérée en utilisant les fonctions free() et freeList().

Passons sur la functions List* merge(const char* remote_branch, const char* message) qui fusionne la branche courante avec la branche passée en paramètre si aucun conflit n'existe.
La fonction commence par vérifier si les paramètres remote_branch et message sont nuls. Ensuite on a récupéré les informations de la branche courante en utilisant différentes fonctions auxiliaires et on vérifie que ces infos sont bien récupérées. On fusionne les work trees de la branche courante et distante et on libère tous les mémoires alloués.
Les conflits détectés lors de la fusion sont stockés dans une liste et sont renvoyés directement s'il en existe la fonction ne procède pas aux autres opérations.
La fonction crée alors un nouveau commit avec le contenu du worktree fusionné et met à jour la branche courante en l'ajoutant dedans. On supprime la branche distante et on restaure le projet du worktree fusionné, on libère toutes les mémoires allouées puis la fonction return NULL si la fusion a été effectuée avec succès sans conflit.

Enfin nous avons WorkTree *stwt(char *s).