Projet: Réalisation d'un logiciel de gestion de versions

Nous considérons dans ce projet la réalisation d'un outil de suivi et de versionnage de code (type git). Ce projet est découpée en plusieurs parties, qui seront ajoutées progressivement au sujet pendant le semestre. Il faudra donc télécharger le sujet du projet à chaque début de séance de TME (pour avoir accès à la suite du projet). Chaque partie est divisée en exercices, qui vont vous permettre de concevoir progressivement le programme final. Il est impératif de travailler régulièrement pendant le semestre, afin de ne pas prendre de retard et de pouvoir profiter des séances de TME associées à chacune des parties du projet.

Cadre du projet

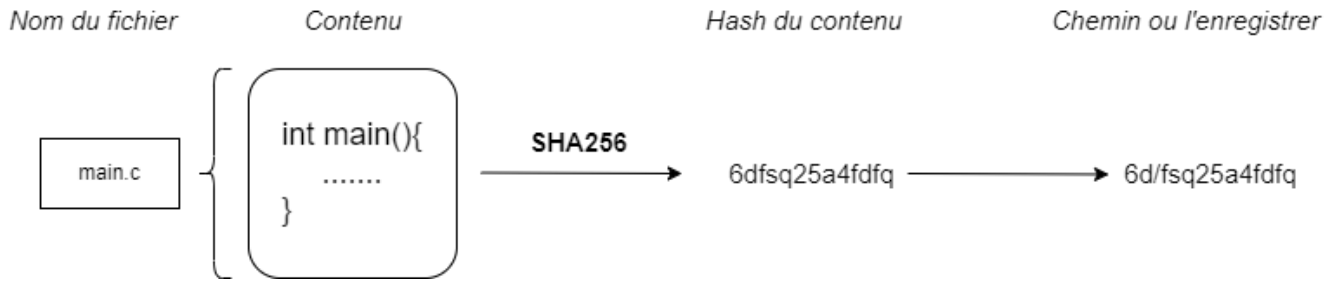
Un logiciel de gestion de versions est un outil permettant le stockage, le suivi et la gestion de plusieurs versions d'un projet (ou d'un ensemble de fichiers). En particulier, ces outils offrent un accès aisé à l'historique de toutes les modifications effectuées sur les fichiers, permettant notamment de récupérer une version antérieure en cas de problème. Par ailleurs, ces outils sont très utiles dans le cadre de travail collabaratif, permettant de fusionner de manière intelligente différentes versions d'un même projet. Par exemple, ces outils sont couramment utilisés en développement logiciel, pour faciliter le travail en équipe et conserver le code source relatifs à différentes versions d'un même logiciel.

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 oeuvre. En particulier, nous allons nous intéresser aux fonctionnalités suivantes :

- Comment permettre à un utilisateur de créer des enregistrements instantanés de son projet ?
- Comment lui permettre de naviguer librement à travers les différents instantanés ?
- Comment construire et maintenir une arborescence des différentes versions de son projet ?
- Comment vérifier l'identité des utilisateurs ?
- Comment sauvegarder des changements qui ne sont pas dans un instantané ?

Vers la création d'enregistrements instantanés

Sous git, tout les objets, qu'ils soient relatifs aux fichiers versionnés ou à leurs méta-données, sont enregistrés sous forme de fichiers. Ces fichiers ont pour particularité de pouvoir dériver le chemin où ils sont stockés à partir de leur contenu, par exemple comme décrit dans la figure suivante :



Ici la fonction de hachage SHA256 est appliquée sur le contenu du fichier, puis le chemin où doit être stocké le fichier est obtenu en insérant un "/" entre le deuxième et le troisième caractères du hash. Faire dépendre le chemin du contenu permet notamment de sauvegarder toutes les différentes versions du fichier. En effet, modifier le contenu du fichier va modifier le chemin vers lequel le sauvegarder et ainsi créer différentes sauvegardes correspondant à différents états du fichier. Quand on réalise une telle sauvegarde, on dit communément qu'on "enregistre un instantané" ou encore qu'on crée un "enregistrement instantané". L'objectif de cette première partie du projet est d'écrire un programme permettant d'enregistrer un instantané, comme décrit dans l'exemple.

Exercice 1 – Prise en main du langage Bash

Les données sur le disque sont stockées dans des fichiers (file en anglais), qui sont des structures de données qui apparaissent aux programmes comme des suites finies d'octets. La structure de données qui organise les fichiers sur le disque s'appelle le système de fichiers (file system). Le système de fichiers est l'une des fonctionnalités principales d'un système d'exploitation. Si généralement on y a recours en utilisant une interface graphique, nous apprendrons ici à le contrôler par du code. Pour cela, nous allons commencer par quelques exercices de prise en main du langage Bash permettant d'utiliser une interface en ligne de commande.

Q 1.1 Pour vous familiariser avec les commandes usuelles, commencez par suivre la séquence de d'instructions suivantes, et utiliser à chaque fois la commande `man` pour afficher la documentation de la commande concernée :

- Ouvrez un terminal (sous linux, cela positionne par défaut sur le chemin principal "∼/").
- Utilisez la commande `pwd` pour afficher le répertoire courant.
- Utilisez la commande `cd` pour vous positionner sur votre répertoire de travail pour cette UE.
- Utilisez la commande `mkdir` pour créer un répertoire nommé "projet_scv".
- Positionnez vous dans le répertoire nouvellement créé, puis utilisez la commande `touch` pour créer les fichiers "main.c", "Makefile" et "test.txt".

— Utilisez `ls` pour lister les fichiers de votre répertoire.
— Après avoir modifié le fichier "main.c" via votre éditeur de texte préféré, utilisez la commande `cat` pour en afficher le contenu.
— Utilisez la commande `rm` pour supprimer le fichier "test.txt".

Linux met en place des flux globaux permettant à des commandes appelées, de dialoguer avec le programme appelant, en écrivant ou en lisant depuis ces flux. Ces flux globaux sont :
— `stdin` (entrée standard) : flux d'entrée du programme. Par défaut, il s'agit des données saisies au clavier. Ce flux permet notamment de définir des programmes interactifs, récupérant des données saisies sur le terminal par l'utilisateur, par le biais de la fonction `scanf` (par exemple).
— `stdout` (sortie standard) : ce flux correspond à la sortie du programme. Par défaut, il s'agit du terminal ayant lancé le programme. Il permet notamment d'afficher des données sur le terminal.
— `stderr` (sortie standard d'erreur) : ce flux sert à récupérer les messages d'erreurs, qui par défaut sont affichés sur le terminal qui a lancé le programme.

Sous linux, il est possible, après avoir appelée une commande, de rediriger une des sorties vers un fichier. Par exemple, avec la commande `ls > list.txt`, vous obtenez un fichier "list.txt" contenant la liste des fichiers et répertoires présents dans le répertoire courant. Il est également possible de rediriger la sortie d'une commande vers une autre, en utilisant ce que l'on appelle une "pipeline". Par exemple, la commande `cat noms.txt | sort` permet de lire la liste de noms présente dans le fichier noms.txt, et de la transmettre à la fonction sort qui va la trier (par ordre alphabétique).

Q 1.2 Sous linux, la commande `sha256sum` permet de hacher le contenu d'un fichier en utilisant la fonction de hachage SHA256. En utilisant une redirection et une pipeline, écrivez une commande qui transmets le contenu du fichier "main.c" à la commande `sha256sum` puis écrit le hash correspondant dans un fichier (temporaire) appelé "file.tmp".
Ces commandes peuvent aussi être utilisées à travers un code C, par le biais de la fonction `system` qui, comme son nom l'indique, permet de faire des appels système. Pour bien comprendre le fonctionnement, commencez par exécuter le programme C suivant :

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
    system("ls");
}
```

Q 1.3 Écrivez une fonction `int hashFile(char* source, char* dest)` qui, étant donné le chemin de deux fichiers, calcule le hash du contenu du premier fichier et l'écrit dans le deuxième fichier.
Bien que la fonction `system` permette d'exécuter des commande Bash, ce n'est pas le seul moyen de manipuler le système de fichiers. Un autre moyen, qui est préférable quand il est disponible, est d'utiliser des librairies C prévues à cet effet. Par exemple, on préférera ce code :

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
    FILE* f = fopen("test.txt", "w");
    fprintf(f, "Test");
}
```

à son équivalent suivant :

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
    system("echo Test $>$ test.txt");
}
```

En particulier, la librairie `<unistd.h>` permet une gestion efficace des fichiers temporaires, ce qui nous sera très utile pour stocker le hash d'un fichier en attente de lecture par notre programme. Voici un exemple de code illustrant comment créer un fichier temporaire avec la fonction `mkstemp` de cette librairie :

```c
static char template[] = "/tmp/myfileXXXXXX";
char fname[1000];
strcpy(fname, template);
int fd = mkstemp(fname);
```

Dans cet exemple, l'appel à `mkstemp(fname)` crée un fichier temporaire dont le nom sera stocké dans `fname`. Ce nom sera généré de manière unique à partir du motif `template`. Plus précisément, il faut que les six derniers caractères de `template` soient "XXXXXX" pour que la fonction `mkstemp` les remplacent de sorte à créer un nom de fichier unique. La fonction `mkstemp` ouvre ensuite le fichier temporaire nouvellement créé, et renvoie un descripteur de fichier ouvert (en lecture et écriture).

Q 1.4 En utilisant la commande `sha256sum`, un pipe, une redirection, et un fichier temporaire (qu'il faudra supprimé après usage), écrivez une fonction `char* sha256file(char* file)` qui renvoie une chaîne de caractères contenant le hash du fichier donné en paramètre.

Exercice 2 – Implémentation d'une liste de chaînes de caractères

Dans cet exercice, il s'agira d'implémenter les fonctions permettant de gérer une structure de données de type liste chaînée dont la définition est la suivante :

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

typedef Cell* List;
```

Q 2.1 Écrivez une fonction `List* initList()` qui initialise une liste vide. Veillez par la suite à ne plus initialiser de liste autrement que par cette fonction.

Q 2.2 Écrivez une fonction `Cell* buildCell(char* ch)` permettant d'allouer et de retourner une cellule de la liste.

Q 2.3 Écrivez une fonction `void insertFirst(List *L, Cell* C)` permettant d'ajouter un élément en tête d'une liste.

Q 2.4 Écrivez une fonction `char* ctos(Cell* c)` qui retourne la chaîne de caractères qu'elle représente, puis utilisez cette fonction pour écrire la fonction `char* ltos(List* L)` qui transforme une liste en une chaîne de caractères avec le format suivant : chaîne1|chaîne2|chaîne3|...

Q 2.5 Écrivez une fonction `Cell* listGet(List* L, int i)` qui renvoie le $i^{\text{ème}}$ élément d'une liste.

Q 2.6 Écrivez une fonction `Cell* searchList(List* L, char* str)` qui recherche un élément dans une liste à partir de son contenu et renvoie une référence vers lui ou NULL s'il n'est pas dans la liste.

Q 2.7 Écrivez une fonction `List* stol(char* s)` qui permet de transformer une chaîne de caractères représentant une liste en une liste chaînée.

Q 2.8 Écrivez la fonction `void ltof(List* L, char* path)` permettant d'écrire une liste dans un fichier, et la fonction `List* ftol(char* path)` permettant de lire une liste enregistrée dans un fichier.

Exercice 3 – Gestion de fichiers sous git

L'objectif final de cet exercice est de produire une fonction qui enregistre un instantané d'un fichier dont le nom est donné en paramètre.

Q 3.1 Les fonctions `opendir` et `readdir` de la librairie `<dirent.h>` permettent d'explorer un répertoire. Par exemple, on peut afficher le noms des fichiers et des répertoires qui le composent en procédant de la manière suivante :

```
1 DIR *  dp = opendir (root_dir);
2 struct dirent *ep;
3 if (dp != NULL)
4 {
5     while ((ep = readdir (dp)) != NULL)
6     {
7         printf("\%s␣\n", ep->d_name);
8     }
9 }
```

Écrivez une fonction `List* listdir(char* root_dir)` qui prend en paramètre une adresse et renvoie une liste contenant le noms des fichiers et répertoires qui s'y trouvent.

Q 3.2 Utilisez la question précédente pour écrire une fonction `int file_exists(char *file)` qui retourne 1 si le fichier existe dans le répertoire courant et 0 sinon.

Q 3.3 Écrivez une fonction `void cp(char *to, char *from)` qui copie le contenu d'un fichier vers un autre, en faisant une lecture ligne par ligne du fichier source.
Indication : pensez à vérifier que le fichier source existe avant.

Q 3.4 Écrivez une fonction `char* hashToPath(char* hash)` qui retourne le chemin d'un fichier à partir de son hash (on rappelle que le chemin s'obtient en insérant un "/" entre le deuxième et le troisième caractères du hash).

Q 3.5 En utilisant la commande Bash `mkdir` (qui permet de créer un répertoire), écrivez une fonction `void blobFile(char* file)` qui enregistre un instantané du fichier donné en entrée.

Enregistrement de plusieurs instantanés

Dans la partie précédente, nous avons vu comment enregistrer un instantané d'un fichier. Dans un projet, on est souvent amené à manipuler un ensemble de fichiers, structurés en arborescence (avec des répertoires). Dans ce cas, on peut vouloir enregistrer un instantané de plusieurs fichiers et/ou répertoires du projet. Pour ce faire, nous allons travailler avec la structure suivante :

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

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

Un `Workfile` représente un fichier ou répertoire dont on souhaite enregistrer un instantané. Il possède trois champs :

- `name` correspond au nom du fichier ou du répertoire.
- `hash` correspond au hash associé à son contenu, et est initialisé à NULL.
- `mode` donne les autorisations associés au fichier (modification, lecture et exécution) et est initialisé à zéro.

Un `WorkTree` est simplement un tableau de `Workfile`. Les autorisations associés à un fichier décrivent qui peut le modifier, le lire et l'exécuter, et sont représentées par un nombre de 3 digits en octal (par exemple 777 donne à tout le monde le droit de lire, écrire et exécuter le fichier). Ces autorisations peuvent être récupérées en utilisant la fonction suivante qui nous sera utile dans le deuxième exercice de cette partie (c-à-d dans l'exercice 5) :

```
int getChmod(const char *path){
    struct stat ret;

    if (stat(path, &ret) == -1) {
        return -1;
    }

    return (ret.st_mode & S_IRUSR)|(ret.st_mode & S_IWUSR)|(ret.st_mode & S_IXUSR)|/*owner*/
        (ret.st_mode & S_IRGRP)|(ret.st_mode & S_IWGRP)|(ret.st_mode & S_IXGRP)|/*group*/
        (ret.st_mode & S_IROTH)|(ret.st_mode & S_IWOTH)|(ret.st_mode & S_IXOTH);/*other*/
}
```

Pour modifier les autorisations, on pourra utiliser la fonction suivante :

```
void setMode(int mode, char* path){
    char buff[100];
    sprintf(buff, "chmod %d %s", mode, path);
    system(buff);
}
```

Le but de cette partie est de pouvoir créer un enregistrement instantané d'un WorkTree et de son contenu, puis de permettre la restauration de cet ensemble de fichiers comme décrit dans ces enregistrements instantanés.

Exercice 4 – Fonctions de manipulation de base

Dans cet exercice, il s'agit d'écrire les fonctions de manipulation de base.

MANIPULATION DE WORKFILE

Q 4.1 Écrivez une fonction `WorkFile* createWorkFile(char* name)` qui permet de créer un `WorkFile` et de l'initialiser.

Q 4.2 Écrivez une fonction `char* wfts(WorkFile* wf)` qui permet de convertir un `WorkFile` en chaîne de caractères contenant les différents champs séparés par des tabulations (caractère '\t').

Q 4.3 Écrivez une fonction `WorkFile* stwf(char* ch)` qui permet de convertir une chaîne de caractères représentant un `WorkFile` en un `WorkFile`.

MANIPULATION DE WORKTREE

Q 4.4 Écrivez une fonction `WorkTree* initWorkTree()` permettant d'allouer un `WorkTree` de taille fixée (donnée par une constante du programme) et de l'initialiser.

Q 4.5 Écrivez une fonction `int inWorkTree(WorkTree* wt, char* name)` qui vérifie la présence d'un fichier ou répertoire dans un `WorkTree`. Cette fonction doit retourner la position du fichier dans le tableau s'il est présent, et -1 sinon.

Q 4.6 Écrivez une fonction `int appendWorkTree(WorkTree* wt, char* name, char* hash, int mode)` qui ajoute un fichier ou répertoire au `WorkTree` (s'il n'existe pas déjà).

Q 4.7 Écrivez une fonction `char* wtts(WorkTree* wt)` qui convertit un `WorkTree` en une chaîne de caractères composée des représentations des `WorkFile` séparées par un saut de ligne (caractère '\n').

Q 4.8 Écrivez une fonction qui convertit une chaîne de caractères représentant un `WorkTree` en un `WorkTree`.

Q 4.9 Écrivez une fonction `int wttf(WorkTree* wt, char* file)` qui écrit dans le fichier `file` la chaîne de caractères représentant un `WorkTree`.

Q 4.10 Écrivez une fonction `WorkTree* ftwt(char* file)` qui construit un `WorkTree` à partir d'un fichier qui contient sa représentation en chaîne de caractères.

Exercice 5 – Enregistrement instantané et restauration d'un WorkTree

Réaliser un enregistrement instantané d'un `WorkTree` revient à créer un enregistrement instantané du fichier qui le représente. Néanmoins, pour pouvoir ensuite le distinguer d'un fichier classique, on ajoutera l'extension ".t" au nom de son enregistrement instantané. Par exemple, si le hash du fichier représentant le `WorkTree` est "dsfsd245azd", alors l'enregistrement instantané est dans "ds/fsd245azd.t".

Q 5.1 Écrivez une fonction `char* blobWorkTree(WorkTree* wt)` qui crée un fichier temporaire représentant le `WorkTree` pour pouvoir ensuite créer l'enregistrement instantané du `WorkTree` (avec l'extension ".t"). Cette fonction devra retourner le hash du fichier temporaire.

On s'intéresse maintenant à une fonction `char* saveWorkTree(WorkTree* wt, char* path)` qui, étant donné un `WorkTree` dont le chemin est donné en paramètre, crée un enregistrement instantané de tout son contenu (de manière récursive), puis de lui même. Plus précisément, la fonction parcourt le tableau de `WorkFile` de `wt`, et pour chaque `WorkFile WF`, elle réalise le traitement suivant :

- Si `WF` correspond à un fichier, alors la fonction `blobFile` est utilisée pour créer un enregistrement instantané de ce fichier, puis on récupère le hash du fichier et son mode (avec la fonction `getChmod`) pour le sauvegarder dans `WF`.
- Si `WF` correspond à répertoire, la fonction doit réaliser un appel récursif sur ce répertoire. Pour cela, il faudra créer un nouveau `WorkTree newWT` représentant tout le contenu de ce répertoire (fonction `listdir`), réaliser un appel récursif sur `newWT`, puis récupérer son hash et son mode pour le sauvegarder dans `WF`.

Après avoir traité tout le tableau de `wt`, la fonction `saveWorkTree` se termine en appelant `blobWorkTree` sur `wt` pour créer son enregistrement instantané et retourner son hash.

Q 5.2 Écrivez la fonction `char* saveWorkTree(WorkTree* wt, char* path)`.

L'appel à `saveWorkTree` permet de conserver une sauvegarde de l'état de plusieurs fichiers à un instant donné. Si entre temps, les fichiers on été modifiés et que l'on souhaite revenir en arrière, il convient d'avoir une fonction récursive qui restaure un `WorkTree`, c'est-à-dire qui recrée l'arborescence des fichiers comme décrit par ses enregistrements instantanés. Pour restaurer un `WorkTree`, on va définir une fonction `void restoreWorkTree(WorkTree* wt, char* path)`, qui parcourt le tableau de `wt`, en réalisant pour chacun de ses `WorkFile WF` le traitement suivant :

- Trouver l'enregistrement instantané correspondant au hash de `WF`.
- Si l'enregistrement ne possède pas l'extenstion ".t", il s'agit d'un fichier. Dans ce cas, on doit créer une copie de l'enregistrement à l'endroit indiqué par la variable `path`, et lui donner le nom et les autoristations correspondans aux champs `name` et `mode`.
- Si l'enregistrement possède l'extension ".t", alors il s'agit d'un répertoire. Dans ce cas, il faut créer le `WorkTree` associé, modifier la variable `path` en y ajoutant ce répertoire à la fin, puis faire un appel récursif sur ce nouveau `WorkTree`.

Q 5.3 Écrivez la fonction `void restoreWorkTree(WorkTree* wt, char* path)`.
Conseil : pensez à sauvegarder votre code dans un autre répertoire avant de tester cette fonction, au risque de supprimer tout votre projet !

Gestion des commits

Dans les parties précédentes, nous avons vu comment enregistrer un instantané de plusieurs fichiers et/ou répertoires, ce qui permet de sauvegarder l'état d'un projet à un instant donné. Pour pouvoir suivre l'évolution d'un projet, il faut pouvoir organiser chronologiquement différents enregistrements instantanés. Pour cela, on utilise ce qu'on appelle des *commits* (ou points de sauvegarde), qui sont des enregistrements instantanés associés à des étapes jugées importantes dans la chronologie du projet (ceux dont on souhaite garder une trace quand on s'intéresse à l'évolution du projet).

Avec notre implémentation, un commit est donc associé à l'enregistrement instantané d'un `WorkTree`, accompagné d'autres informations relatives au point de sauvegarde comme par exemple l'auteur du commit, un message décrivant le commit, des informations permettant d'ordonner chronologiquement ce commit par rapport aux autres, etc. Comme certaines de ces données sont facultatives, nous allons travailler avec la structure suivante :

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

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

typedef HashTable Commit;
```

En d'autres termes, un commit est implémenté par une table de hachage dont les clés et les valeurs sont des chaînes de caractères. Plus précisément, les éléments de la table doivent correspondre aux informations associées au point de sauvegarde. En particulier, un `Commit` devra au moins contenir une paire de la forme `("tree", hash)` où `hash` est le hash du fichier correspondant à l'enregistrement instantané d'un `WorkTree`.

Autre exemple, pour ajouter le message de description "version v1 du projet", il faudra ajouter à la table la paire `("message", "version v1 du projet")`. Pour récupérer la description plus tard, il suffira de faire une recherche dans la table avec la clé `"message"`. Si cette recherche ne donne rien pour un `Commit` donné, on en déduira que cette sauvegarde ne possède pas de message de description.

Remarque : pour ceux qui connaissent Git, cette partie a pour but de simuler les commandes `git add` et `git commit` de manière simplifiée.

Exercice 6 – Fonctions de base pour les commits

Dans cet exercice, nous allons implémenter des fonctions de base pour manipuler un `Commit`.

Q 6.1 Écrivez une fonction `kvp* createKeyVal(char* key, char* val)` permettant d'allouer et d'initialiser un élément. Écrivez une fonction `void freeKeyVal(kvp* kv)` permettant de libérer la mémoire associé à un élément.

Q 6.2 Écrivez une fonction `char* kvts(kvp* k)` qui permet de convertir un élément en une chaîne de caractères de la forme "clé :valeur". Écrivez une fonction `kvp* stkv(char* str)` qui permet de faire la conversion inverse.

Q 6.3 Écrivez une fonction `Commit* initCommit()` qui permet d'allouer et d'initialiser un `Commit` de taille fixée (donnée par une constante du programme).

Q 6.4 Choisissez une fonction de hachage sur ce site : `http://www.cse.yorku.ca/~oz/hash.html`.

Q 6.5 Écrivez une fonction `void commitSet(Commit* c, char* key, char* value)` qui insère la paire `(key, value)` dans la table, en gérant les collisions par adressage ouvert et probing linéaire.

Q 6.6 À l'aide des fonctions précédentes, écrivez une fonction `Commit* createCommit(char* hash)` qui alloue et initialise un `Commit`, puis ajoute l'élément obligatoire correspondant à la clé `"tree"`.

Q 6.7 Écrivez une fonction `char* commitGet(Commit* c, char* key)` qui cherche dans la table s'il existe un élément dont la clé est `key` (en sachant que les conflits sont résolus par adressage ouvert et probing linéaire). La fonction retourne la valeur de l'élément s'il existe, et NULL sinon.

Q 6.8 Écrivez une fonction `char* cts(Commit* c)` qui convertit un commit en une chaîne de caractères. Cette chaîne doit être composée des chaînes de caractères représentant chacun de ses couples (clé, valeur), séparées par un saut de ligne. Écrivez ensuite une fonction `Commit* stc(char* ch)` qui réalise la conversion inverse.

Q 6.9 Écrivez une fonction `void ctf(Commit* c, char* file)` qui écrit dans le fichier `file` la chaîne de caractères représentant le commit `c`. Écrivez la fonction `Commit* ftc(char* file)` qui permet de charger un `Commit` depuis un fichier le représentant.

Q 6.10 Comme pour un `WorkTree`, réaliser un enregistrement instantané d'un `Commit` revient à créer un enregistrement instantané du fichier qui le représente, et pour le distinguer des autres types de fichiers, on ajoute l'extension ".c" au nom de son enregistrement instantané (au lieu de ".t" pour un `WorkTree`). En procédant comme pour la fonction `blobWorkTree` (question 5.1), écrivez une fonction `char* blobCommit(Commit* c)` qui crée un enregistrement instantané d'un `Commit` en passant par un fichier temporaire. Cette fonction devra retourner le hash du fichier temporaire.

Exercice 7 – Gestion temporelle des commits de manière linéaire

Dans cet exercice, nous allons permettre à l'utilisateur de créer des points de sauvegarde, qui seront organisés de manière linéaire, ordonnés par ordre chronologique, du plus récent au plus ancien. Pour ce faire, chaque `Commit` doit maintenant posséder au minimum les clés suivantes :

- `author` : l'identifiant de l'utilisateur qui a créé le point de sauvegarde.
- `tree` : le hash du `WorkTree` correspondant au point de sauvegarde.
- `message` : une description de ce point de sauvegarde.
- `predecessor` : le hash du `Commit` correspondant au point de sauvegarde précédent.

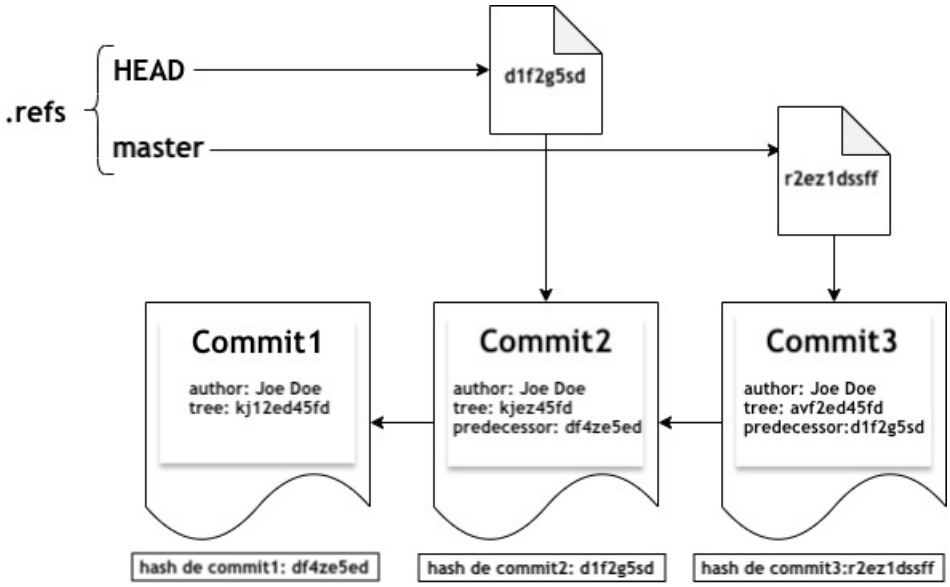
Les commits sont donc ici organisés en liste simplement chainée : il suffit de connaître le hash du dernier `Commit` pour avoir accès à tous les autres à l'aide de la clé "predecessor". Une telle liste sera appelée *branche* dans la suite.

Pour avoir accès à cette branche et la manipuler facilement, nous allons utiliser ce qu'on appelle des *références*. Sous Git, les références sont des pointeurs vers des commits. Ils se présentent sous la forme

de fichiers contenant le hash du `Commit` concerné. Ces fichiers sont stockés dans un répertoire caché appelé `.refs`. Dans ce répertoire, il nous faudra au moins les deux références suivantes :

- `master` : un fichier contenant le hash du dernier `Commit` (le plus récent). Ce fichier pourrait s'appeler autrement, mais une fois que ce nom est choisi, il donnera son nom à la branche.
- `HEAD` : un fichier contenant le hash d'un commit quelconque. Ce fichier est utilisé pour simuler des déplacements dans la timeline, par exemple quand on souhaite consulter des anciennes versions du projet. Par défaut, ce fichier contient le hash du dernier `Commit` de la branche.

La figure suivante permet d'illustrer cette structure sur un exemple avec trois points de sauvegarde.



MANIPULATION DES RÉFÉRENCES

Q 7.1 Écrivez une fonction `void initRefs()` qui crée le répertoire caché `.refs` (s'il n'existe pas déjà), puis crée les fichiers `master` et `HEAD` (vides).

Q 7.2 Écrivez une fonction `void createUpdateRef(char* ref_name, char* hash)` qui met à jour une référence en remplaçant son contenu par `hash`. Si la référence n'existe pas, la fonction commence par créer le fichier.

Q 7.3 Écrivez une fonction `void deleteRef(char* ref_name)` qui permet de supprimer une référence.

Q 7.4 Écrivez une fonction `char* getRef(char* ref_name)` qui récupère vers quoi pointe une référence (c-à-d le hash contenu dans le fichier). Si le fichier est vide, la fonction retourne une chaîne de caractère vide. Si le fichier n'existe pas, la fonciton retourne NULL.

SIMULATION DE LA COMMANDE GIT ADD

Avant de réaliser un commit, il faut indiquer au programme quels sont les fichiers et/ou répertoires pour lesquels on souhaite créer un point de sauvegarde. Ce sont des fichiers/répertoires qui ont été modifiés

depuis le dernier point de sauvegarde, et dont la version courante est suffisamment satisfaisante pour vouloir les sauvegarder. Sous Git, cela se fait via la commande `git add` qui place ces fichiers/répertoires dans une zone appelée *zone de préparation* (*staging area*). Dans ce projet, nous allons simuler la zone de préparation à l'aide d'un fichier caché ".add". Ce fichier est la représentation d'un `WorkTree`, initialement vide, et dans lequel l'utilisateur ajoutera progressivement les fichiers et/ou répertoires qui doivent faire partie du prochain commit.

Q 7.5 Écrivez une fonction `void myGitAdd(char* file_or_folder)` qui permet à un utilisateur d'ajouter un fichier ou un répertoire dans le `WorkTree` correspondant à la zone de préparation. Si le fichier ".add" n'existe pas, il faudra d'abord le créer.
Indication : il faudra utiliser la fonction `appendWorkTree`.

SIMULATION DE LA COMMANDE GIT COMMIT

On souhaite maintenant pouvoir réaliser un commit, c'est-à-dire créer un point de sauvegarde à la manière de la commande `git commit`. Pour cela, on souhaite avoir une fonction `void myGitCommit(char* branch_name, char* message)` qui réalise les étapes suivantes :

- Si le répertoire ".refs" n'existe pas, la fonction affiche le message "Initialiser d'abord les références du projet", puis se termine. Si le fichier `branch_name` n'existe pas, la fonction affiche le message "La branche n'existe pas", puis se termine. Si les fichiers `HEAD` et `branch_name` ne pointent pas vers la même chose, la fonction affiche "HEAD doit pointer sur le dernier commit de la branche", puis se termine.
- Sinon, charger le `WorkTree` correspondant au fichier ".add", puis supprimer ce fichier.
- Enregistrer un instantané de ce `WorkTree` avec la fonction `saveWorkTree` et récupérer son hash.
- Créer un `Commit c` qui contiendra une paire `("tree", hash)` avec le hash du `WorkTree`.
- Lire le fichier `branch_name` pour récupérer le hash du dernier commit de la branche, puis ajouter ce hash comme `predecessor` de `c`. Si le fichier `branch_name` est vide, on ne crée pas de `predecessor` pour ce commit.
- Si l'argument `message` n'est pas NULL, ajouter le au commit `c` comme message descriptif.
- Créer un enregistrement instantané de `c` avec la fonction `blobCommit`, et récupérer son hash.
- Mettre à jour le fichier `branch_name` en remplaçant son contenu par le hash de `c`.
- Mettre à jour le fichier `"HEAD"` en remplaçant son contenu par le hash de `c`.

Q 7.6 Écrivez la fonction `void myGitCommit(char* branch_name, char* message)`.

MAIN PRINCIPAL

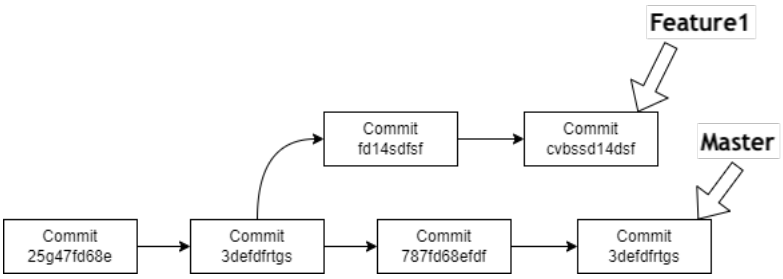
Il s'agit maintenant de créer un exécutable permettant à un utilisateur de suivre l'évolution d'un projet. Pour cela, il faudra créer un fichier appelé "myGit.c" contenant un main, et l'exécutable s'appellera "myGit". Ce main sera enrichi tout au long du projet, mais pour l'instant, nous souhaitons que l'utilisateur puisse taper les commandes suivantes dans le terminal :

- `./myGit init` : initialise le répertoire de références.
- `./myGit list-refs` : affiche toutes les références existantes.
- `./myGit create-ref <name> <hash>` : crée la référence `<name>` qui pointe vers le commit correspondant au hash donné.
- `./myGit delete-ref <name>` : supprime la référence name.
- `./myGit add <elem> [<elem2> <elem3> ...]` : ajoute un ou plusieurs fichiers/répertoires à la zone de préparation (pour faire partie du prochain commit).

— `./myGit list-add` : affiche le contenu de la zone de préparation.
— `./myGit clear-add` : vide la zone de préparation.
— `.myGit commit <branch_name> [-m <message>]` : effectue un commit sur une branche, avec ou sans message descriptif.

Gestion d'une timeline arborescente

Dans la partie précédente, nous avons travaillé avec une seule branche, permettant de suivre l'évolution d'un projet de manière linéaire. Le grand avantage de travailler avec plusieurs branches est de pouvoir considérer plusieurs versions d'un même code. En particulier, cela permet à plusieurs personnes de travailler simultanément sur différentes parties/fonctionnalités d'un projet, de pouvoir considérer des versions alternatives d'un code, ou encore de réaliser des modifications sans affecter la branche principale. L'illustration ci-dessous représente un projet avec deux branches : `master` qui est la branche principale et `Feature1` qui correspond à l'implémentation de la première fonctionnalité d'un projet. Dans le répertoire `.refs`, il y aura donc une référence appelée `master` contenant le hash du dernier commit de la branche principale, ainsi qu'une référence appelée `Feature1` contenant le hash du dernier commit de cette branche parallèle.



Comme plusieurs branches peuvent maintenant exister simultanément, nous allons utiliser un fichier caché appelé `.current_branch`, permettant à tout moment de savoir dans quelle branche nous nous situons. Ce fichier contiendra le nom de la branche courante. Avec plusieurs branches, il est primordial de pouvoir manipuler facilement une branche, et de savoir comment naviguer entre les différentes branches de l'arborescence. C'est l'objectif de cette partie.

Exercice 8 – Fonctions de base de manipulation des branches

Q 8.1 Écrivez une fonction `void initBranch()` qui crée le fichier caché `.current_branch` contenant la chaîne de caractères "master".

Q 8.2 Écrivez une fonction `int branchExists(char* branch)` qui vérifie l'existence d'une branche.

Q 8.3 Écrivez une fonction `void createBranch(char* branch)` qui crée une référence appelée `branch`, qui pointe vers le même commit que la référence `HEAD`. **Indication :** utiliser la fonction `getRef`.

Q 8.4 Écrivez une fonction `char* getCurrentBranch()` qui lit le fichier caché `.current_branch` pour retourner le nom de la branche courante.

Q 8.5 Écrivez une fonction `void printBranch(char* branch)` qui parcourt la branche appelée `branch`, et pour chacun de ses commits, affiche son hash et son message descriptif (s'il en existe un).

Q 8.6 Écrivez une fonction `List* branchList(char* branch)` qui contruit et retourne une liste chaînée contenant le hash de tous les commits de la branche. **Rappel :** on parcourt une branche à l'aide de la clé "predecessor".

Q 8.7 Écrivez une fonction `List* getAllCommits()` qui renvoie la liste des hash des commits de toutes les branches (sans répétition).

Exercice 9 – Simulation de la commande git checkout

La commande `git checkout` permet de naviguer entre les branches. Quand la commande `checkout` est appelée avec un nom de branche, elle permet de changer de branche. Plus précisément, pour simuler cette commande, nous allons écrire une fonction `void myGitCheckoutBranch(char* branch)` qui procède comme suit :

- On modifie le fichier `.current_branch` pour contenir le nom de la branche donné en paramètre.
- On modifie la référence `HEAD` pour contenir le hash du dernier commit de `branch` (on rappelle que ce hash est contenu dans le fichier `branch`).
- On restaure le worktree correspondant au dernier commit de `branch`.

La commande `git checkout` peut aussi être appelée avec le hash de n'importe quel commit, permettant à l'utilisateur de retourner sur n'importe quelle version de son projet. Cette commande permet aussi à l'utilisateur de ne donner que les premiers caractères du hash d'un commit, pour ne pas avoir à le taper entièrement au clavier. Pour simuler cela, nous allons écrire une fonction `void myGitCheckoutCommit(char* pattern)` qui procède comme suit :

- On récupère la liste de tous les commits existants.
- On filtre cette liste pour ne garder que ceux qui commencent par `pattern`.
- S'il ne reste plus qu'un hash après ce filtre, alors on met à jour la référence `HEAD` pour pointer sur ce hash, et on restaure le worktree correspondant.
- S'il ne reste plus aucun hash après le filtre, la fonction affiche un message d'erreur à l'utilisateur.
- S'il reste plusieurs hash après le filtre, la fonction les affiche tous et demande à l'utilisateur de préciser sa requête.

Le but de cet exercice est d'écrire ces deux fonctions.

Q 9.1 Écrivez une fonction `void restoreCommit(char* hash_commit)` qui permet de restaurer le worktree associé à un commit dont le hash est donné en paramètre. **Indication :** Il faut utiliser la fonction `restoreWorkTree`.

Q 9.2 Écrivez la fonction `void myGitCheckoutBranch(char* branch)` qui procède comme décrit au début de l'exercice.

Q 9.3 Écrivez une fonction `List* filterList(List* L, char* pattern)` qui retourne une nouvelle liste contenant uniquement les éléments de `L` qui commencent par la chaîne de caractères `pattern`.

Q 9.4 Écrivez une fonction `void myGitCheckoutCommit(char* pattern)` qui procède comme décrit au début de l'exercice.

Exercice 10 – Mise à jour du main principal

Q 10.1 Modifier votre main principal pour que la commande `./myGit init` réalise aussi l'initialisation de la branche courante (fonction `initBranch`).

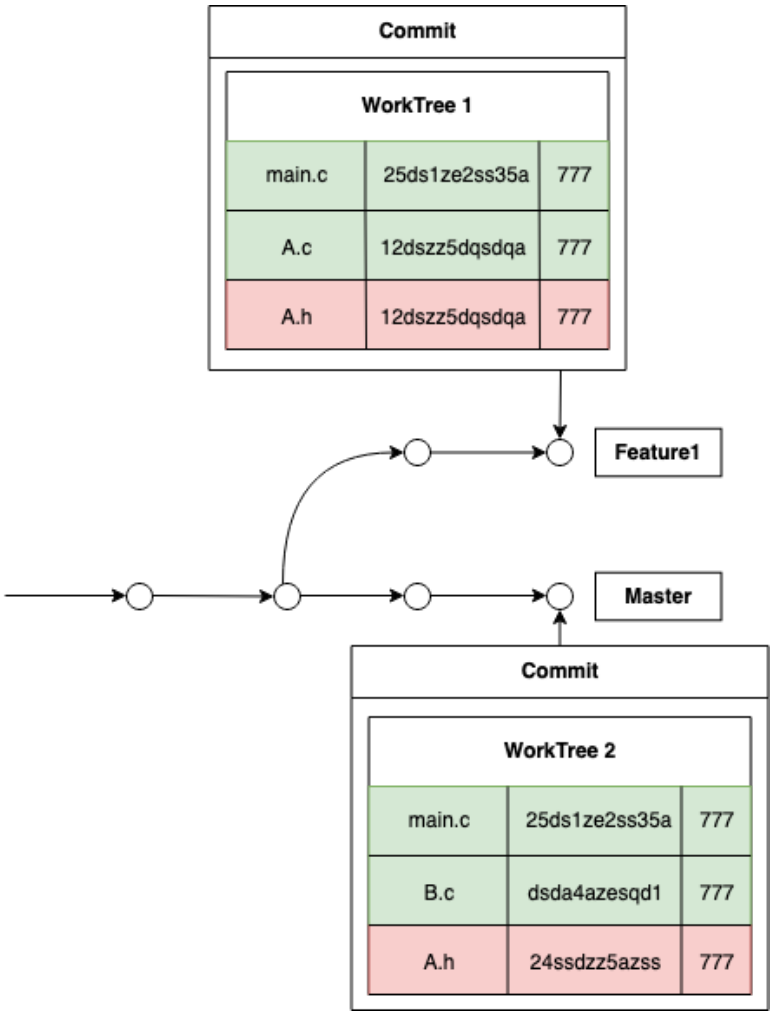
Q 10.2 Compléter le main principal pour que l'utilisateur puisse maintenant taper les commandes suivantes dans le terminal :

- `./myGit get-current-branch` : affiche le nom de la branche courante.
- `./myGit branch <branch-name>` : crée une branche qui s'appelle `<branch-name>` si elle n'existe pas déjà. Si la branche existe déjà, la commande doit afficher un message d'erreur.
- `./myGit branch-print <branch-name>` : affiche le hash de tous les commits de la branche, accompagné de leur message descriptif éventuel. Si la branche n'existe pas, un message d'erreur est affiché.
- `./myGit checkout-branch <branch-name>` : réalise un déplacement sur la branche `<branch-name>`. Si cette branche n'existe pas, un message d'erreur est affiché.
- `./myGit checkout-commit <pattern>` : réalise un déplacement sur le commit qui commence par `<pattern>`. Des messages d'erreur sont affichés quand `<pattern>` ne correspond pas à un seul commit.

Gestion des fusions de branches

Nous arrivons à la dernière partie du projet (à réaliser sur deux séances). Dans cette partie, nous nous intéressons à la fusion de branches qui est une fonctionnalité très utile, par exemple quand plusieurs personnes travaillent en parallèle sur le même projet, et qu'il s'agit de réunir leur travail. La commande `git merge` de Git permet de fusionner deux branches, c'est-à-dire de créer un nouveau commit dont le worktree correspond à la fusion des worktrees des derniers commits de ces deux branches. Notez que ce nouveau commit est un peu spécial car il possède deux prédécesseurs au lieu d'un seul.

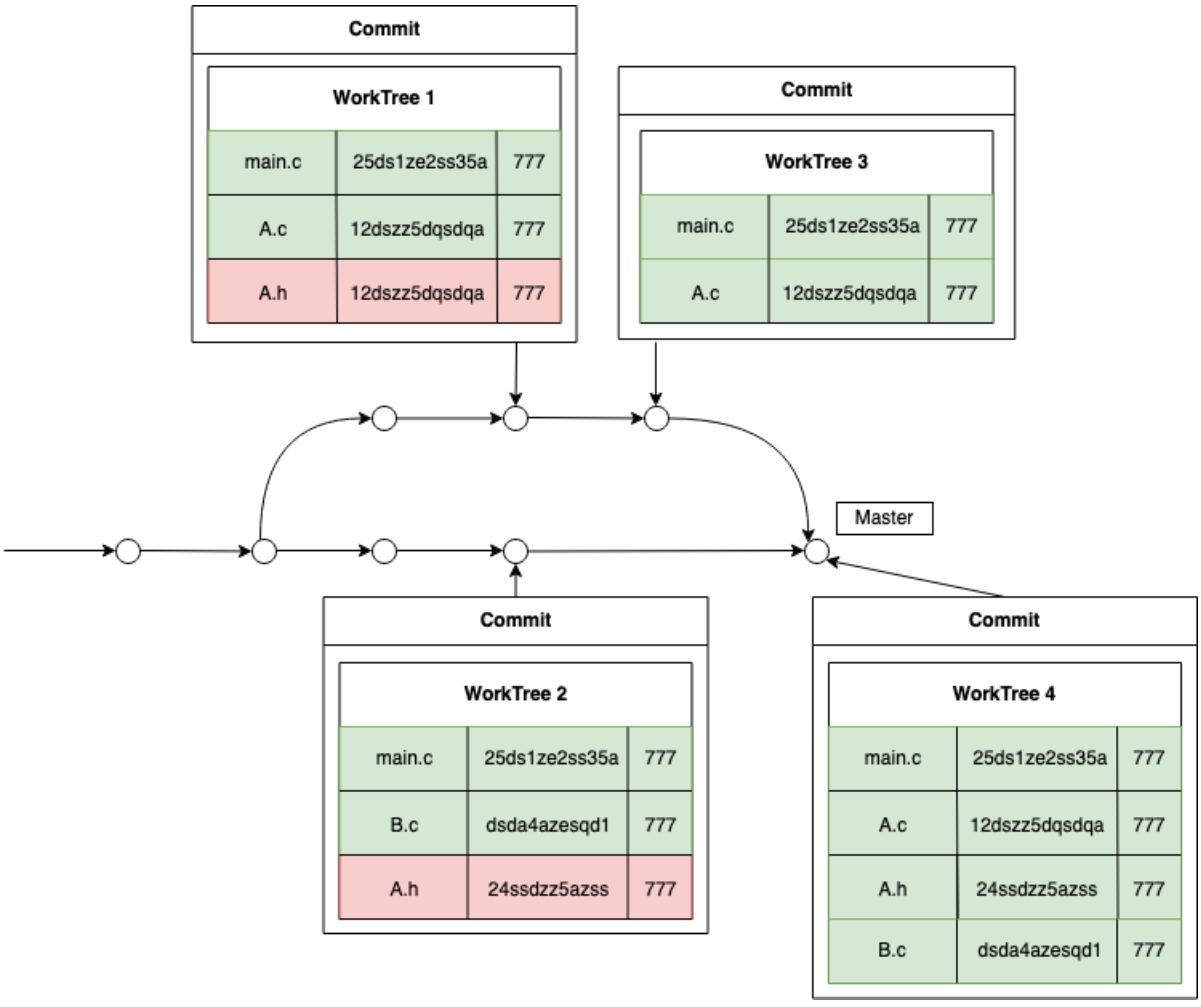
Remarquez que fusionner deux worktrees peut créer des conflits : un conflit survient lorsque les worktrees à fusionner contiennent un fichier/répertoire de même nom, mais avec des hash (et donc des contenus) différents. La figure ci-dessous illustre un cas de conflit en voulant fusionner la branche `master` avec la branche `Feature1`. Nous devrons donc proposer des méthodes de gestion de conflits pour pouvoir fusionner des branches de manière pertinente.



Exercice 11 – Une première méthode de fusion

Dans cet exercice, nous allons nous intéresser à une méthode de gestion de conflits très simple. Pour résoudre un conflit, on va créer un commit dit "de suppression" qui consiste à supprimer les éléments conflictuels d'une des deux branches avant de faire une fusion sans conflit. Par exemple, dans l'illustration précédente, si l'utilisateur préfère conserver le fichier "A.h" de la branche `master` plutôt que celui de la branche `Feature1`, alors nous allons créer un commit de suppression sur la branche `Feature1` qui consiste à supprimer le fichier "A.h" de cette branche (cf. WorkTree 3 sur la figure ci-dessous). Ensuite, on peut réaliser la fusion des deux branches qui se fera donc sans conflit (cf. WorkTree 4). Notez qu'après avoir fusionné la branche `Feature1` avec la branche `master`, nous avons supprimé la référence `Feature1` car cette branche n'existe plus. Néanmoins, le commit de fusion doit bien avoir deux prédécesseurs :

— la clé "predecessor" dont la valeur est le hash du dernier commit de branche courante,
— la clé "merged_predecessor" correspondant au hash du dernier commit de la branche supprimée.



Q 11.1 Écrivez une fonction `WorkTree* mergeWorkTrees(WorkTree* wt1, WorkTree* wt2, List**`

`conflicts)` qui, étant donné deux worktrees, crée :
- une liste de chaînes de caractères composée des noms de fichiers/répertoires qui sont en conflit.
- un nouveau `WorkTree` composé des fichiers et/ou répertoires qui ne sont pas en conflit.

Q 11.2 Écrivez une fonction `List* merge(char* remote_branch, char* message)` qui fusionne la branche courante avec la branche passée en paramètre si aucun conflit n'existe. Dans ce cas, la fonction doit procéder comme suit :
- créer le worktree de fusion (avec la fonction `mergeWorkTrees`),
- créer le commit associé à ce nouveau worktree, en indiquant qui sont ses prédécesseurs, et en lui ajoutant le message descriptif passé en paramètre,
- réaliser un enregistrement instantané du worktree de fusion et de ce nouveau commit,
- ajouter le nouveau commit à la branche courante,
- mettre à jour la référence de la branche courante et la référence `HEAD` pour pointer vers ce nouveau commit,
- supprimer la référence de la branche passée en paramètre,
- restaurer le projet correspondant au worktree de fusion.

La fonction retourne NULL dans ce cas. Si par contre il existe au moins un conflit, aucune de ces opérations n'est effectuée. À la place, la fonction retourne la liste des conflits (obtenue avec la fonction `mergeWorkTrees`).

Q 11.3 On souhaite maintenant pouvoir créer des commits de suppression en cas de conflits. Écrivez une fonction `void createDeletionCommit(char* branch, List* conflicts, char* message)` qui crée et ajoute un commit de suppression sur la branche `branch`, correspondant à la suppression des éléments de la liste `conflicts`. Pour ce faire, on peut procéder ainsi :
- On commence par se déplacer sur la branche en question (à l'aide de `myGitCheckoutBranch`).
- On récupère le dernier commit de cette branche, et le worktree associé,
- On vide la zone de préparation (c-à-d le fichier ".add"), puis on utilise la fonction `myGitAdd` pour ajouter les fichiers/répertoires du worktree qui ne font pas partie de la liste des conflits.
- On appelle la fonction `myGitCommit` pour créer le commit de suppression.
- On revient sur la branche de départ (à l'aide de `myGitCheckoutBranch`).

Q 11.4 Compléter le `main` pour que l'utilisateur puisse taper la commande `./myGit merge <branch> <message>`, et que selon les cas, le traitement suivant soit appliqué :
- S'il n'y a pas de collision entre la branche courante et la branche `<branch>`, on réalise le merge (avec la fonction `merge`), et on affiche un message à l'utilisateur pour lui dire que la fustion s'est bien passée.
- S'il y a des collisions, on doit proposer à l'utilisateur de choisir une des options suivantes :
  1. Garder les fichiers de la branche courante, et donc créer un commit de suppression pour la branche `<branch>`, avant de faire appel à la fonction `merge`.
  2. Garder les fichiers de la branche `<branch>`, et donc créer un commit de suppression pour la branche courante, avant de faire appel à la fonction `merge`.
  3. Choisir manuellement, conflit par conflit, la branche sur laquelle il souhaite garder le fichier/répertoire qui est en conflit. Dans ce cas, il faudra que l'on divise la liste de conflits en deux listes, selon ce que l'utilisateur nous dira, puis créer un commit de suppression sur chaque branche avec ces deux listes, avant de faire la fusion avec la fonction `merge`.

Exercice 12 – Une méthode de fusion basée sur l'algorithme de Needleman-Wunsch

Suite la semaine prochaine.