Rapport de projet : Blockchain appliquée à un processus électoral

Plusieurs facteurs affectent la transparence du processus électoral. L'implication de l'organisme exécutif dans la mise en place des élections, alors que celui-ci même est candidat à ces derniers, baisse de l'honnêteté du processus. Le comptage des votes, une démarche fastidieuse réalisé par des personnes physiques, ainsi que l'anonymat des votes rendent faible la fiabilité du comptage.

Pour répondre à la problématique de la désignation d'un vainqueur dans un processus électoral fiable et permettant le vote à distance pour lutter contre l'abstention, nous allons, dans ce projet, nous pencher sur la conception d'un prototype de protocoles et de structures de données, dans le but d'implémenter efficacement un processus de vote uninominal sur un système décentralisé, en garantissant la fiabilité, sécurité et intégrité de l'élection.

Le projet est divisé en 4 parties, dans le dossier code :

- un fichier.c par partie de la forme partie#.c.
- un header par partie de la forme partie#.h.
- un fichier test par partie de la forme main_test_p#.c.
- 2 fichier de test de performances.
- un Makefile englobant tout le projet (Cependant, le all au début du Makefile ne concerne que les fichiers test des parties).

Partie 1 : Implémentation d'outils de cryptographie

La première partie de ce projet tourne autour de l'introduction d'un système de cryptage de messages utilisant un couple de clefs, une clé publique et une clé secrète, dont les valeurs ainsi que le cryptage sont basé sur le calcule du modulo de nombres premiers.

```c
//
//  partie1.h
//  partie1
//

#ifndef partie1_h
#define partie1_h

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

int is_prime_naive(long p);

long modpow_naive(long a, long m, long n);

int modpow(long a, long m, long n);

int doc(long a, long m, long n);

void perf_modpow(long a, long n);

int witness(long a, long b, long d, long p);

long rand_long(long low, long up);

int is_prime_miller(long p, int k);

long random_prime_number(int low_size, int up_size, int k);

long extended_gcd(long s, long t, long* u, long* v);

void generate_keys_values(long p, long q, long* n, long* s, long*u);

long* encrypt(char* chaine, long s, long n);

char* decrypt(long* crypted, int size, long u, long n);

#endif /* partie1_h */

```

Exercice 1: Résolution du problème de primalité

Q.1

int is_prime_naive(long p) :

Teste la primalité de **p** en testant en boucle si **p** est divisible par **a**, pour **a** compris entre 3 et **p**-1. Renvoie **1** si **a** ne divise jamais **p**, **0** sinon.
Complexité : O(p-3).

Q.2

Le plus grand nombre premier testé en moins de 2 millièmes de secondes est **15000413**. La vitesse augmente considérablement entre 10000000 et 15000000.

Q.3

long modpow_naive(long a, long m, long n) :

Calcule et retourne **a^m** mod **n** à l'aide d'une méthode naïve, en multipliant à chaque itération la valeur courante par **a** puis en appliquant le mod **n**, le tout **m** fois.
Complexité : O(m).

Q.4

int modpow(long a, long m, long n) :

Calcule et retourne **a^m** mod **n** à l'aide d'une méthode récursive.
Complexité : O(log(m)).

Q.5

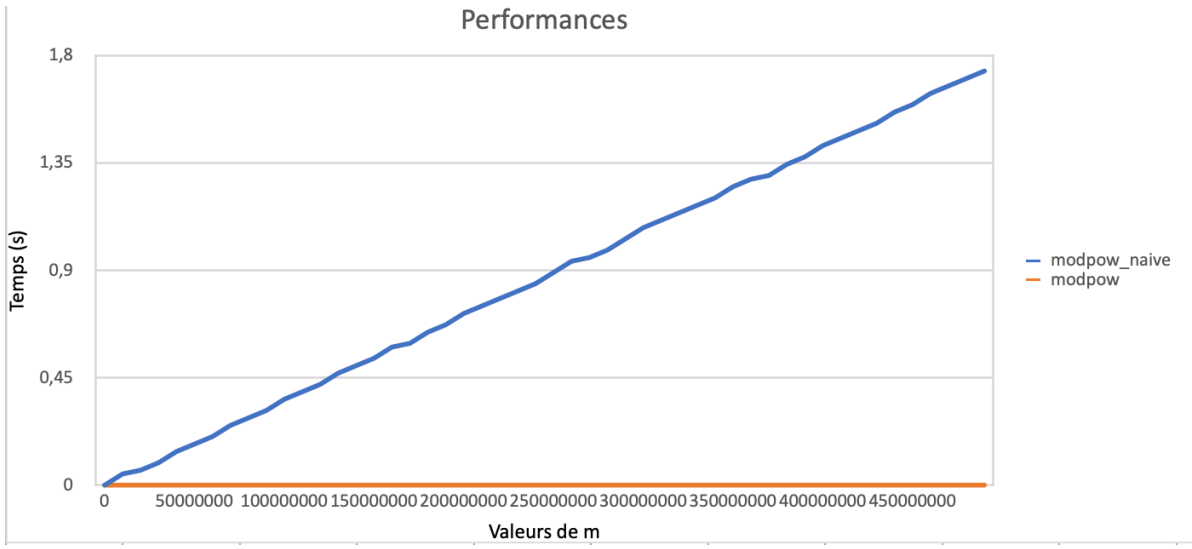
int doc(long a, long m, long n) :

Ouvre un fichier de nom « Perf_modpow » pour y noter les performances de **modpow_naive** et **modpow** sur les valeurs **a**,**m**, et **n**.

void perf_modpow(long a, long n) :

Invoque en boucle la fonction **doc** sur les valeurs **a**,**n** et pour un **m** variant de 0 à 490 000 000 par pas de 10 000 000.

À l'aide des fonctions précédentes, on construit le fichier de test de performances **main_perf_modpow** qui nous donne la courbe suivante :



Le temps de traitement de **modpow_naive** est visiblement supérieur à celui de **modpow**.

Q.7

Le test du témoin de Miller a une probabilité d'erreur d'au maximum 1/4 par valeur testé. Or, comme nous allons tester **k** valeurs, la probabilité d'erreur de l'algorithme est de 4^-**k**.

Q.8

long random_prime_number(int low_size, int up_size, int k) :

Utilise **is_prime_miller** et **rand_long** pour effectuer **k** testes de Miller et renvoie un nombre premier généré aléatoirement dont la taille en bits est compris entre **low_size** et **up_size**.

Exercice 2: Implémentation du protocole RSA

Q.1

void generate_key_values(p,q,n,s,u) :

Utilise les nombres premiers **p** et **q** ainsi que **extended_gcd(s,t,u,v)** pour appliquer le protocole RSA pour calculer et initialiser les variables **n**, **s** et **u**. On obtient ainsi un couple de clefs publique et secrète.

Q.2

long* encrypt(char* chaine, long s, long n) :

Renvoie le cryptage de chaque caractère de **chaine** par la **clé secrète (s,n)** en suivant le protocole RSA (pour chaque char c, calculer c^**s** mod **n**), sous la forme d'un tableau de long.

Q.3

char* decrypt(long* crypted, long u, long n) :

Renvoie le décryptage des long de **crypted** par la **clé publique (u,n)** en suivant le protocole RSA (pour chaque long m, calculer m^**u** mod **n**), sous la forme d'une chaine de caractère.

Test : partie 1

Le fichier **main_test_p1.c** est un fichier test pour tester les fonctions principales de la partie 1, en voici un exemple de résultat :

```
cle publique = (b5, 179)
cle privee =  (d, 179)
Initial message:  Hello
Encoded representation :
Vector: [b0   cd    45    45    173   ]
Decoded: Hello
```

Grâce à la fonction **generate_keys_values**, on initialise un couple de clefs, crypte le message « Hello » à l'aide de la fonction **crypted** et de la clé secrète, puis décrypte le tableau de long résultant avec la fonction **decrypt**.

Partie 2 : Déclarations sécurisées

La deuxième partie de ce projet se consacre à la modélisation des votes par l'introduction de nouvelles structures, celle des key, celle des signatures ainsi que des déclarations. Dans une déclaration se retrouve la clé publique du votant, celle du candidat et enfin la signature du vote. Et enfin, dans une signature se trouve le cryptage de la clé publique du candidat par la clé publique du votant. On apprendra à gérer ces structures pour ensuite les implémenter dans un processus électoral.

```c
#ifndef partie2_h
#define partie2_h

#include <stdio.h>
#include "partie1.h"

typedef struct {
    long val;
    long n;
} Key;

typedef struct {
    Key* pKey;
    char* mess;
    Signature* sgn;
} Protected;

typedef struct {
    long* content;
    int size;
} Signature;

void init_key(Key* key, long val, long n);

void init_pair_keys(Key* pKey, Key* sKey, long low_size, long up_size);

char* key_to_str(Key* key);

Key* str_to_key(char* str);
```

```c
Signature* init_signature(long* content, int size);

Signature* sign(char* mess, Key* sKey);

char* signature_to_str(Signature* sgn);

Signature* str_to_signature(char* str);

Protected* init_protected(Key* pKey, char* mess, Signature* sgn);

int verify(Protected* pr);

char* protected_to_str(Protected* pr);

Protected*  str_to_protected(char* str);

void generate_random_data(int nv, int nc);

#endif /* partie2_h */
```

Exercice 3: Manipulations de structures sécurisées

Q.1

La structure **Key** contient deux long représente les clefs (publique ou secrète).

Q.2

void init_key(Key* key, long val, long n) :

Permet d'initialiser les variables dans la structure **key** avec **val** et **n** pour un **key** a la mémoire déjà allouée.

Q.3

void init_pair_keys(Key* pKey, Key* sKey, long low_size, long up_size) :

Permet d'initialiser un couple de clé publique et clé secrète en les stockant dans **pKey** (publique) et **sKey** (secrète) pour des Key a la mémoire déjà allouées et en utilisant la méthode RSA pour une taille en bits compris entre **low_size** et **up_size**.

Q.4

char* key_to_str(Key* key) :

Renvoie une chaine de caractères représentant la clé **key**.

Key* str_to_key(char* str) :

Déclare, initialise et renvoie la clé représenté par **str**.

Q.5

La structure **Signature** a un tableau de long **content** et un entier **size** représentant la taille du tableau. Son tableau permettra de stocké le cryptage de la clé publique du candidat choisi par la clé secrète de l'électeur.

Q.6

Signature* init_signature(long* content, int size) :

Permet d'allouer la mémoire et d'initialiser une signature à partir d'un tableau de long **content** de taille **size** à la mémoire déjà allouée et déjà initialisé.

Q.7

Signature* Sign(char* mess, Key* sKey) :

permet d'allouer et d'initialiser une signature, mais cette fois, en cryptant le message **mess** par la clé secrète **sKey** à l'aide de la fonction **crypted**.

Q.9

La structure **Protected,** représentant une déclaration, contient la clé publique de l'électeur **pKey**, une chaine de caractère **mess** correspondant à la clé publique du candidat sous forme de char*, ainsi qu'une signature **sgn** formé du cryptage de **mess** par la clé secrète de l'électeur.

Q.10

Protected* init_protected(Key* pKey, char* mess, Signature* sgn) :

Permet d'allouer la mémoire et d'initialiser une déclaration avec la clé publique de l'électeur **pKey**, la clé publique du candidat choisi **mess**, et la signature **sgn** leur correspondant sans vérifier sa validité.

Q.11

int verify(Protected* pr) :

Permet de vérifier la validité d'une déclaration **pr**, elle est valide si le décryptage de la signature par la clé publique de l'électeur donne bien la clé publique du candidat, la fonction renvoie 1 si la signature est valide, 0 sinon.

Q.12

char* protected_to_str(Protected* pr) :

Renvoie une chaine de caractères représentant la déclaration **pr**.

Protected* str_to_protected(char* str) :

Déclare, initialise et renvoie la déclaration représenté par **str**.

Test : partie 2

Le fichier **main_test_p2.c** contient des tests pour les fonctions principales de cette partie, un exemple de résultats :

```
pKey: bf , fd
sKey: 5b , fd
key to str : (bf,fd)
str to key : bf , fd
(bf,fd) vote pour (8a7,f1d)
signature : Vector: [6a  22  f0  9a  b0  50  1b  90 97 ]
 signature to str : #6a#22#f0#9a#b0#50#1b#90#97#
str to signature : Vector: [6a    22  f0  9a  b0  50 1b 90 97 ]
Signature valide
 protected to str : (bf,fd) (8a7,f1d) #6a#22#f0#9a#b0#50#1b#90#97#

str to protected : (bf,fd) (8a7,f1d) #6a#22#f0#9a#b0#50#1b#90#97#
```

En premier lieu, on utilise la fonction **init_pair_keys** pour initialiser un couple de clefs électeur, ensuite on affiche la clé publique à l'aide la fonction **key_to_str** puis transforme la chaine de caractère résultante en clé.

Par la suite, un nouveau couple de clefs candidats est initialisé par la fonction **init_pair_keys**, ainsi, une signature du vote est initialisé par la fonction **sign**. Après son affichage à l'aide des fonctions **signature_to_str** et **print_long_vector**, et on transforme la chaine de caractère résultante avec **str_to_signature**.

On initialise la déclaration correspondante aux données précédentes en invoquant la fonction **init_protected**, puis vérifie si la signature de la déclaration est valide avec la fonction **verify**.

Finalement, on affiche la déclaration à l'aide de **protected_to_str**, et enfin, on transforme la chaine de caractère résultante à l'aide de **str_to_protected**.

Exercice 4: Création de données pour simuler le processus de vote

La fonction **generate_random_data(nv,nc)** génère **nv** électeurs, puis sélectionne aléatoirement **nc** candidats parmi les électeurs, et enfin, génère **nv** déclarations chacune propre à chaque électeur, en leurs attribuant aléatoirement à chacun un candidat. 3 documents sont crée pour stocker ces données.

Clés publiques/privées de chaque électeur

```
(1823,22b1) (1067,22b1)
(fa1,171d) (1361,171d)
(71,2c3) (179,2c3)
(14ed,1687) (515,1687)
(137,719) (4df,719)
(2e7,47b) (197,47b)
(37,d7) (37,d7)
(1e59,2731) (2495,2731)
(10c9,1295) (4f9,1295)
(bf,173) (107,173)
(4d,eb) (8d,eb)
(17,30b) (197,30b)
(7b5,213d) (1abd,213d)
(ce7,102d) (b97,102d)
(76d,1cdb) (e75,1cdb)
(12d,229) (f1,229)
(2ef,4f7) (2ef,4f7)
(10e5,1415) (76d,1415)
(ef5,191f) (11ad,191f)
(881,ddf) (671,ddf)
```

Test : generate_random_data

Pour **nv** = 20 ; **nc** = 5.

Candidats

```
(1e59,2731)
(37,d7)
(137,719)
(ef5,191f)
(7b5,213d)
```

Les déclarations

```
(1823,22b1) (137,719) #205e#cbe#16f3#1d36#dcf#1d36#cbe#2192#186e#
(fa1,171d) (37,d7) #1424#150a#1692#e46#420#1692#eb2#
(71,2c3) (7b5,213d) #192#172#3f#33#3c#166#46#2e#193#3e#
(14ed,1687) (37,d7) #fa1#15c2#bd1#fa0#64#bd1#1b4#
(137,719) (7b5,213d) #4a3#643#106#368#5e6#434#32#471#2b8#55a#
(2e7,47b) (ef5,191f) #3fc#e9#459#3e3#2a3#7#82#7#459#391#
(37,d7) (1e59,2731) #a0#31#b5#a7#44#2c#32#af#9c#31#15#
(1e59,2731) (137,719) #1c7d#25b1#1658#15b1#f2#15b1#25b1#16a8#97c#
(10c9,1295) (137,719) #595#1b9#525#4af#399#4af#1b9#104c#86c#
(bf,173) (137,719) #73#103#58#e6#82#e6#103#da#117#
(4d,eb) (ef5,191f) #7d#10#2#76#13#d1#43#d1#2#d3#
(17,30b) (1e59,2731) #1c2#66#1e0#16b#23a#29e#1e3#2c#1fc#66#cd#
(7b5,213d) (7b5,213d) #1bfe#a48#20fc#1f4f#8e2#4d5#edf#15d9#c96#16e1#
(ce7,102d) (37,d7) #75d#8b#14b#743#7e3#14b#a69#
(76d,1cdb) (137,719) #175#1bbf#188c#108e#58c#108e#1bbf#ac5#fbc#
(12d,229) (ef5,191f) #98#a#66#1a8#33#1dc#94#1dc#66#213#
(2ef,4f7) (1e59,2731) #28#25f#483#1a9#39#89#168#4d1#33#25f#29#
(10e5,1415) (7b5,213d) #250#1152#11d0#350#25f#bee#6cf#106d#101b#d32#
(ef5,191f) (7b5,213d) #d4b#422#625#bf6#b3c#843#17ee#11f#875#29#
(881,ddf) (1e59,2731) #42b#e3#b86#848#65c#1b2#89f#a19#b6f#e3#a04#
```

Partie 3 : Base de déclarations centralisée

La troisième partie de ce projet tournera autour de l'implémentation de fonctions de gestion de données sous la forme des structures implémentés précédemment dans le but de simuler un processus électoral.

```c
//
//  partie3.h
//  partie3
//

#ifndef partie3_h
#define partie3_h

#include "partie2.h"

typedef struct cellKey{
    Key* data;
    struct cellKey* next;
} CellKey;

typedef struct cellProtected{
    Protected* data;
    struct cellProtected* next;
} CellProtected;

typedef struct hashcell{
    Key* key;
    int val;
} HashCell;

typedef struct hashtable{
    HashCell** tab;
    int size;
} HashTable;

CellKey* create_cell_key(Key* key);

void add_key(CellKey** cell, Key* clef);

CellKey* read_public_keys(char* file);

void print_list_keys(CellKey* LCK);

void delete_cell_key(CellKey* c);

void delete_list_keys(CellKey** C);

CellProtected* create_cell_protected(Protected* pr);

void add_pr(CellProtected** cellp, Protected* pr);

CellProtected* read_protected(char* file);

void print_list_protected(CellProtected* LCK);

void delete_cell_protected(CellProtected* c);

void delete_list_protected(CellProtected** c);

void* validation(CellProtected** c);

HashCell* create_hashcell(Key* key);

int hash_function(Key* key, int size);

int find_position(HashTable* t, Key* key);

HashTable* create_hashtable(CellKey* keys, int size);

void delete_hashtable(HashTable* t);

Key* compute_winner(CellProtected* decl, CellKey* candidates, CellKey* voters, int sizeC, int sizeV);

#endif /* partie3_h */
```

Sous-titre

Sous-titre

Exercice 5 : Lecture et stockage des données dans des listes chaînées

Q.2

void add_key(CellKey cell, Key* clef) :**

Déclare, alloue la mémoire et initialise une cellule pour la Key **clef** et l'ajoute à la tête de la liste chainée **cell**.

Q.3

CellKey* read_public_keys(char* file) :

Permet de lire le fichier de nom **file** contenant des clefs sous forme de chaine de caractères, et leur initialise une liste chainée de cellules à l'aide de la fonction **add_key**. Renvoie un pointeur sur le début de la liste chainée.

Q.4

void print_list_keys(CellKey* LCK) :

Permet d'afficher les clefs dans la liste chainée **LCK** sous la forme d'une liste.

Q.5

void delete_cell_key(CellKey* LCK) :

Permet de free la mémoire allouée à une cellule LCK et sa clé.

void delete_list_keys(CellKey c) :**

Permet de free la mémoire allouée à une liste chainée **c** de CellKey, utilise la fonction **delete_cell_key**.

Q.6

CellProtected* create_cell_protected(Protected* pr):

Permet de déclarer, d'allouer la mémoire et d'initialiser une cellule avec la déclaration **pr.**

Q.7

void add_pr(CellKey cellp, Protected* pr) :**

Déclare, alloue la mémoire et initialise une cellule pour la déclaration **pr** et l'ajoute à la tête de la liste chainée **cellp**.

Q.8

CellProtected* read_protected(char* file) :

Permet de lire le fichier de nom **file** contenant des déclarations sous forme de chaine de caractères, et leur initialise une liste chainée de cellules à l'aide de la fonction **add_pr**. Renvoie un pointeur sur le début de la liste chainée.

Q.9

void print_list_protected(CellProtected* LCK) :

Permet d'afficher la liste chainée LCK sous la forme d'une liste.

Q.10

void delete_cell_protected(CellProtected* c) :

Permet de free la mémoire allouée à une cellule et sa déclaration.

void delete_list_protected(CellProtected c) :**

Permet de free la mémoire allouée à une liste chainée de cellules en utilisant la fonction delete_cell_protected.

Exercice 6 : Détermination du gagnant de l'élection

Q.1

void validation(CellProtected c) :**

Utilise la fonction **verify** précédemment définie pour vérifier la validité de chaque déclaration de la liste chainée passé en paramètre, si la déclaration n'est pas valide, la fonction la supprime de la liste chainée à l'aide de la fonction **delete_cell_protected**.

Q.2

HashCell* create_hashcell(Key* key) :

Déclare et alloue la mémoire d'un pointeur sur une cellule HashCell, puis initialise sa donnée **key** avec la clé passé en paramètre et initialise sa donnée **val** à 0.

Q.3

int hash_function(Key* key, int size) :

Retourne le reste de la division euclidienne de la donnée **val** de la clé **key** passé en paramètre par l'entier **size**. C'est une fonction de hachage.

Q.4

int find_position(HashTable* t, Key* key) :

Fonction de probing linéaire sur le tableau de cellules de clés **tab** de la table de hachage **t** passé en paramètre. À l'aide la fonction de hachage **hash_function**, la fonction retourne la position de key dans **tab**, ou si **key** n'est pas dans **tab**, une position où elle pourrait être stocké.

Q.5

HashTable* create_hashtable(CellKey* Keys, int size) :

Permet de déclarer, d'allouer la mémoire et d'initialiser une table de hachage avec un **tab** de taille **size**, ainsi que parcourir la liste chainée **Keys** passé en paramètre pour placé les HashCell* correspondant dans le tab à l'aide des fonctions **find_position** et **create_hashcell**.

Q.6

void delete_hashtable(HashTable* t) :

Permet de libérer la mémoire allouée à la table de hachage **t** ainsi que son **tab** sans libérer la mémoire des cellules CellKey* stockés dans les cellules HashCell*.

Q.7

Key* compute_winner(CellProtected* decl, CellKey* candidates, CellKey* voters, int sises, int sizeV) :

À travers l'utilisation de **create_hashtable**, **find_position et str_to_key,** la fonction recense les votes dans **decl** pour les différents candidats et marque si chaque **voters** a déjà voté ou non. Après la recherche du **candidates** avec le plus gros nombre de votes, la fonction retourne sa clé publique. C'est une fonction qui empêche les fraudes suivantes : le même votant vote plusieurs fois, le vote est comptabilisé pour le mauvais candidats, le votant n'est pas un vrai électeur.

Test : partie 3

Le fichier **main_test_p3.c** contient des tests pour les fonctions principales de cette partie, un exemple de résultats pour 20 électeurs et 5 candidats :

```
(14f,5bd)
(1bcb,292d)
(2b3,b5f)
(1d3,2c3)
(8d,31f)
(7f,cb)
(8d7,9e9)
(16f,661)
(1091,1295)
(549,1747)
(12b,a43)
(fbb,1751)
(d27,15b9)
(1cc7,25e5)
(97f,2d8f)
(ba7,1691)
(41,637)
(73,10d3)
(4c7,f77)
(2419,301d)

(2419,301d)
(16f,661)
(7f,cb)
(14f,5bd)
(1bcb,292d)

(14f,5bd) (2419,301d) #194#269#2d8#1e#237#40b#52d#524#1e#413#d7#
(1bcb,292d) (7f,cb) #217#9d2#15fc#22b4#1c5d#27a4#2277#
(2b3,b5f) (7f,cb) #548#383#27c#846#86#626#7b0#
(1d3,2c3) (7f,cb) #10d#e6#1fa#1bd#127#af#156#
(8d,31f) (16f,661) #139#1bc#159#1a9#303#159#159#1bc#a4#
(7f,cb) (1bcb,292d) #2f#31#69#a2#69#48#8#39#8#64#68#
(8d7,9e9) (14f,5bd) #3d8#31#6f#2b4#2c#420#14e#4c5#69d#
(16f,661) (7f,cb) #5b7#f6#f1#41a#2f5#59b#69#
(1091,1295) (7f,cb) #e1a#47d#99#c97#73#4e2#1077#
(549,1747) (16f,661) #ef7#561#69e#4bd#126b#69e#69e#561#753#
(12b,a43) (16f,661) #9c4#37f#796#88a#890#796#796#37f#606#
(fbb,1751) (2419,301d) #832#93a#65b#1205#123c#db9#bc2#1696#1205#839#14f8#
(d27,15b9) (14f,5bd) #aa0#95a#102b#9d2#1be#465#1271#1273#e4b#
(1cc7,25e5) (1bcb,292d) #228f#56c#2069#2086#2069#1a5c#1515#2051#1515#21b7#1552#
(97f,2d8f) (16f,661) #114d#b43#1a30#2350#2c1c#1a30#1a30#b43#935#
(ba7,1691) (7f,cb) #14a4#10b2#db4#164b#1669#ec9#560#
(41,637) (2419,301d) #46b#229#3da#482#468#45f#326#f9#482#a#36c#
(73,10d3) (1bcb,292d) #d0#b80#e0c#394#e0c#75c#ae2#17f#ae2#997#cd1#
(4c7,f77) (14f,5bd) #f7#cff#514#582#544#45d#bc6#b75#f49#
(2419,301d) (16f,661) #1ac5#2be4#2156#d7#237c#2156#2156#2be4#29#

vainqueur : (7f,cb)
```

On utilise la fonction **generate_random_data** pour générer 20 électeurs, 5 candidats et 20 déclarations, puis respectivement les fonctions **read_public_keys**, **read_protected**, **print_list_keys** et **print_list_protected** pour lire les données sous forme de liste chainée de cellules, puis on les affiche.
Et enfin on utilise la fonction **compute_winner** sur les liste chainées pour vérifier et comptabiliser les votes pour désigner un vainqueur et le retourner. On affiche la clé publique du vainqueur grâce à la fonction **key_to_str**.

Partie 4 : Blocs et persistance des données

Dans cette dernière partie nous allons nous concentrer sur l'implémentation d'une structure de block, des fonctions pour les manipuler, ainsi que l'utilisation de la blockchain pour stocker les déclarations de votes et simuler un processus électoral.

```c
//
//  partie4.h
//  partie4
//

#ifndef partie4_h
#define partie4_h

#include "partie3.h"
#include <openssl/sha.h>
#include <dirent.h>

typedef struct block{
    Key* author;
    CellProtected* votes;
    unsigned char* hash;
    unsigned char* previous_hash;
    int nonce;
}Block;

typedef struct block_tree_cell{
    Block* block;
    struct block_tree_cell* father;
    struct block_tree_cell* firstChild;
    struct block_tree_cell* nextBro;
    int height;
}CellTree;

void write_block(Block* b, char* file);

Block* read_block(char* file);

char* block_to_str(Block* block);

unsigned char* hash_SHA256(unsigned char* str);

int nb_zero_succ(unsigned char* str);

void compute_proof_of_work(Block *B, int d);

int verify_block(Block* B, int d);

int docprime(Block* b, int d);

void perf_compute(Block* b);

```

```c
void delete_block(Block* b);

CellTree* create_node(Block* b);

int update_height(CellTree* father, CellTree* child);

void add_child(CellTree* father, CellTree* child);

int print_tree(CellTree* tree);

void delete_node(CellTree* node);

void delete_tree(CellTree* tree);

CellTree* highest_child(CellTree* cell);

CellTree* last_node(CellTree* tree);

CellProtected* fusion(CellProtected* cell1, CellProtected* cell2);

CellProtected* fusion_ultime(CellTree* tree);

void submit_vote(Protected* p);

void create_block(CellTree** tree, Key* author, int d);

void add_block(int d, char* name);

CellTree* read_tree();

Key* compute_winner_BT(CellTree* tree, CellKey* candidates, CellKey* voters, int sizeC, int sizeV);

#endif /* partie4_h */
```

Exercice 7 : Structure d'un block et persistance

Q.1

void write_block(Block* b, char* file) :

Utilise **key_to_str** et **protected_to_str** pour écrire les données d'un block **b** dans un nouveau fichier de nom **file**.

Q.2

Block* read_block(char* file) :

Utilise **str_to_key**, **create_cell_protected**, et **str_to_protected** pour déclarer, allouer la mémoire et initialiser le block avec les données dans le fichier de nom **file**. Retourne un pointeur sur le block en question.

Q.3

char* block_to_str(Block* block) :

Retourne une chaine de caractère représentant **block**.

Q.5

unsigned char* hash_SHA256(unsigned char* str) :

Fonction de hachage de message, utilise la fonction SHA256 pour hacher chaque caractère de **str**, libère la mémoire allouée à **str** puis forme la chaine de caractère en mettant chaque caractère résultat au format hexadécimal. Retourne un pointeur sur la nouvelle chaine.

Q.6

int nb_zero_succ(unsigned char* str) :

Retourne le nombre de 0 successifs au début de la chaine de caractère **str**.

void compute_proof_of_work(Block* B, int d) :

Met la variable **nonce** de **B** à 0, déclare et initialise une chaine de caractère avec **block_to_str** puis hash en boucle la chaine à l'aide de **hash_SHA256** jusqu'à avoir d 0 successifs au début de la chaine et en augmentant **nonce** de 1 par itération.

Q.7

int verify_block(Block* B, int d) :

Sauvegarde le **nonce** de **B** puis utilise **compute_proof_of_work** sur B et enfin retourne 0 si le **nonce** originel de **B** n'est pas égal à son nouveau **nonce**, 1 sinon. Ça permet de vérifier si le block n'est pas frauduleux puisque bien haché.

Q.8

int docprime(Block* b, int d) :

Mesure les performances de l'utilisation de la fonction **compute_proof_of_work** sur les paramètres **b** et **d** puis les imprime sur un fichier de nom « Perf_compute ».

void perf_compute(Block* b) :
Invoque en boucle **doc_prime** sur **b** en faisant varier **d** de 0 à 5.

Le fichier **main_perf_compute.c** permet de déclarer, d'allouer la mémoire et d'initaliser un block quelconque puis d'utiliser la fonction **perf_compute**. Ainsi, on obtient les données suivantes :

Valeur de d – Perf
0 – 0.000000
1 – 0.000000
2 – 0.000000
3 – 0.015625
4 – 0.062500
5 – 5.203125

On en conclu qu'à partir de 5 le temps de calcule dépasse 1 seconde.

Q.9
void delete_block(Block* b) :
Libère la mémoire allouées aux hash dans le block et au block passé en paramètre.

Exercice 8 : Structure arborescente

Q.1
CellTree* create_node(Block* b) :
Déclare, alloue la mémoire et initialise une cellule arborescente avec le block **b** passé en paramètre et en mettant la variable **height** à 0. Retourne un pointeur sur la cellule.

Q.2
int update_height(CellTree* father, CellTree* child) :
Met à jour la variable **height** du **father** par rapport à celle du **child**. Retourne 1 si la hauteur a été modifié, 0 sinon.

Q.3

void add_child(CellTree* father, CellTree* child) :

Lie **child** à **father**, si **father** a déjà un enfant alors ajoute **child** à la suite des **nextBro** du **firstChild** de **father,** sinon met **child** en tant que **firstChild** du **father**. Utilise **update_height** pour mettre à jour la hauteur des prédécesseurs à la suite de l'ajout.

Q.4

int print_tree(CellTree* tree) :

Affiche le hachage de chaque block de chaque noeud de l'arbre **tree** passé en paramètre.

Q.5

void delete_node(CellTree* node) :

Libère la mémoire du **block** du noeud **node** passé en paramètre à l'aide de la fonction **delete_block**, puis libère la mémoire du noeud.

void delete_tree(CellTree* tree) :

Libère la mémoire de chaque noeud de l'arbre **tree** à l'aide de la fonction **delete_node**.

Q.6

CellTree* highest_child(CellTree* cell) :

Parcours les fils de **Cell** et retourne un pointeur sur celui avec la plus grosse valeur **height**.

Q.7

CellTree* last_node(CellTree* tree) :

Parcours l'arbre obtenu avec **highest_child** du **tree** pour retourner un pointeur sur sa feuille.

Q.8

CellProtected* fusion(CellProtected* cell1, CellProtected* cell2) :

Lie les deux liste chainée passée en paramètres et retourne un pointeur le début de la liste résultante.

Q.9

CellProtected* fusion_ultime(CellTree* tree) :

Utilise les fonctions **highest_child** et **fusion** pour construire une liste chainée formée des liste chainée **votes** de chaque **block** de chaque **noeud** de la plus longue chaine de **noeuds** de l'arbre **tree**. Retourne un pointeur sur le début de la liste chainée résultante.

Exercice 9 : Simulation du processus de vote

Q.1

void submit_vote(Protected* p) :

Permet d'imprimer la déclaration **p** passé en paramètre à la suite d'un fichier texte de nom « **Pending_votes.txt** ».

Q.2

void create_block(CellTree tree, Key* author, int d) :**

Lit les déclarations en attente dans le fichier « **Pending_votes.txt** » à l'aide de la fonction **read_protected**, puis déclare, alloue la mémoire et initialise un **block** avec le hachage du dernier **block** dans **tree** à l'aide de la fonction **last_node**. Ensuite, avec **compute_proof_of_work** appliqué avec l'entier d, elle initialise la preuve de travail **nonce** pour enfin écrire ce **block** sur un fichier « **Pending_block** » et l'ajouter à la suite de **tree** avec **add_child** ce qui nous permet par la suite de créer les block suivants. Enfin cette fonction supprime le fichier « **Pending_votes.txt** ».

Q.3

void add_block(int d, char* name) :

La fonction lit le block dans « **Pending_block** » avec **read_block**, puis vérifie le **block** à l'aide de **verify_block** appliqué avec l'entier **d**, et enfin si le **block** est valide, la fonction utilise **write_block** pour l'imprimer sur un fichier de nom **name** et stocké dans un répertoire « **Blockchain** » et supprime le fichier « **Pending_block** ».

Q.4

CellTree* read_tree( ) :

Cette fonction lit tout les **block** dans le répertoire **Blockchain** et les stock dans un tableau de **noeuds**. Ensuite la fonction lie, à l'aide la fonction **add_child**, les éléments du tableau en fonction des correspondances des variables **hash** et **previous_hash** de leurs **blocks** pour former un arbre avec tout les **noeuds**. Enfin, la fonction cherche la racine de l'arbre (le noeud avec un père NULL) et la retourne.

Q.5

Key* compute_winner_BT(CellTree* tree, CellKey* candidates, CellKey* voters, int sizeC, int sizeV) :

Utilise **fusion_ultime** sur **tree**, nettoie la liste chainée de déclarations résultante avec la fonction **validation**, et enfin retourne la clé retourné par la fonction **compute_winner** avec en entré la liste chainée de déclarations précédemment déclarée et les paramètres **candidates**, **voters**, **sizeC** et **sizeV**.

Q.6

Le fichier main_test_p4 regroupe toutes les instructions.

Q.7

L'utilisation du blockchain pour stocker les déclarations de votes est une méthode très sécurisés pour divers raisons :

- elle permet la décentralisation du processus électoral, donc aucun n'organisme possiblement malveillant a la main sur les votes.
- Le hachage des blocks étant très très fastidieux à faire seul, il est très difficile voire impossible d'intégrer un block frauduleux dans la blockchain, puisque son auteur devra continuer à hacher des blocks puisqu'on est fait confiance à la plus longue chaine.

Cependant certaines fraudes sont toujours possibles, comme mettre un block frauduleux à la toute fin de la chaine.

Test : partie 4

Exemple de résultat de main_test_p4 pour 1000 électeurs et 5 candidats :

764faf5c61ac315f1497f9dfa542713965b785e5cc2f707d6468d7d1124cdfcf

hauteur noeud : 99
block :
5b33f11bbbf0292616083b696e0852025d61f3ecd38d241f9d6624a9e729c1fc

hauteur noeud : 98
block :
d8957e96899973ab9742c2ce06b8b4033690ede58fc3f7ae3fd82c1212192f40

hauteur noeud : 97
block :
032e02574a1dd331cb3237b1bd03a6362ec34ccbb507d8d59c4826a7a403ddaa

hauteur noeud : 96
block :
c791f43b9df9d62ac5bc025a677e836671e5c20ae112dd98a48ff4b7a0631b21

hauteur noeud : 95
block :
0e7e443668e1350a2a28cca1bed9742c77d0e595b726b002d1482df0a89a594d

hauteur noeud : 94
block :
b3d071819e8cfc6dfc3417059fcb0d7301cf86082eaec8c324597bee1e64cbca

hauteur noeud : 93
block :
e0c610a7c1e9bf4e859f3cd7c6bb276cd30603099c925220ec4bc005f99339d0

hauteur noeud : 92
block :
97acec1c91974484a9088abd338c2232790140004 77d228a6a9978ebca64368e

hauteur noeud : 91
block :
3f002e5a1d0379f3822158e172ebfc5818de4e7c2ba0178fe3f12a220578673d

hauteur noeud : 90
block :
80a95274ee12eecc2977142982c05e4c825a1c178c83dc684f73e3c6b8987546

hauteur noeud : 89
block :
3a55e1a767a838e614181fce88913b1c4f6b8fd966f57e7ee5cd1fee2f36b333

hauteur noeud : 88
block :
4fee1d593ec226e5b1d599fd840dc5f01958d955ecd178bfd0b77d3e68c80291

hauteur noeud : 87
block :
358d231a726b8facd170f4a5a442b0bc48290cf0c4faafa2c6b2a7d6f87de256

hauteur noeud : 86
block :
bb098c8ebbc3605f05a40c2fa745198578bb384f46e4ed92e4ba7efba8cc1527

hauteur noeud : 85
block :
e1bddd960a60cf317c412814b390e3bd20ce8261d0c580f97ac3e53cc3e03f0f

hauteur noeud : 84
block :
87f45719c4b1649a61adc813e7ca8b22c523bd731b581fd61253def22ae2671d

hauteur noeud : 83
block :
2ceed71f5949a42d958d1c5a2ffd4cc315ba2a8bf47c288a075a33bf150ee022

hauteur noeud : 82
block :
51872005ec021d54866dc2f24c1af9cd42bf2c51b0de4cf257905cc67f9e5b59

hauteur noeud : 81
block :
074bf9f965ffb54152df59fff09ea88bf628fcb727528c3bbcb0d9e43e6d379c

hauteur noeud : 80
block :
87167f8d1ef06870aacab98929edde3f33413835f82605fc5518d37ad98bf881

hauteur noeud : 79
block :
72b723cc8a0e553d6e8da00793c203e198f14bf1cae0ffe2ba5081f35ac1adc6

hauteur noeud : 78
block :
2411b888af57d846c472f59db3aeac84e2ce0cb0f06468f06fea8614477629bd

hauteur noeud : 77
block :
f32259fc6bf0d49a4b6a2f1ca5e4d3f8e65e457c9a123cdfe5e86b7a8bcb87c9

hauteur noeud : 76
block :
d3a32682b5e0e0aa0d1d0226378c30b6dbda59f1f4218dcf75f9ec3122a1bef
e

hauteur noeud : 75
block :
adb8c09b086ab55bfa7117728a7f77b71f14fe8cb3f9cdd54a10b73d0951a5f8

hauteur noeud : 74
block :
e85defbdbfb782bd8654d5463f3affd578dead4c9ffb0f3fe7a8b4321fce3cc3

hauteur noeud : 73
block :
94fee67e2075417c8ed979720ec6788bbc311c3b9100780311eeaaa8fa4365e
a

hauteur noeud : 72
block :
cd4b45f244b00ef94a7a7e3ec00f749774fde240056bb99d4af4670814d17bb
1

hauteur noeud : 71
block :
dd943729784620e0bdfe9b9836db8617e48f1258c849a6640c818e59fc29374
7

hauteur noeud : 70
block :
5a06b77b83d7851b97b46c788884ca169b70cc8bd1587e5e45ac737008ec0
d5d

hauteur noeud : 69
block :
e169adb65dea68e1dde52e67fcf55e973cca2dc198211c4abcd941f183b370d
f

hauteur noeud : 68
block :
a12b5de5d83b0e71cf94ac6682cc75c1ceaf84cef787e5e1b580e999c6642147

hauteur noeud : 67
block :
c1b62330149080274dad749cbdce1c66873997605298c5fa04a59837676e6fb0

hauteur noeud : 66
block :
efb72e1776be46e5fbdbf0201d4bf379f94323471632ccd12e5b19f6fd3473b5

hauteur noeud : 65
block :
da17de3d025134aaff712b0864c2807e8505c56660b4c936686961f5b09807bc

hauteur noeud : 64
block :
44a80d0eaac6951210d740794771f26a78f5a758dbfdc5ceb3965ad969f403b0

hauteur noeud : 63
block :
bbf4ffaefcb3d468fa9f857605cd43b985394d863a8b690ba2b43672a014dce6

hauteur noeud : 62
block :
00403c1c9e246a9735fd7e963f2e10d9bdb5b2cb6d0f6634077980eef64d629f

hauteur noeud : 61
block :
bb5822376c8330eb6305d219b8fa00ee10b99ee29cef1a1e67d4d96e15107a99

hauteur noeud : 60
block :
8662ca1c4620aca1fe2fde0bb999b83a711b3a183c42c0ba85ea042fe333aa1
0

hauteur noeud : 59
block :
c8c6565dc68b81f4e3e0b67f38686fc801eb5569c2cca9f4ed681484a198709
d

hauteur noeud : 58
block :
1ef82ded4d9e389fb4c05ca1f37cd08d1e6df544d2624e4321ad0f72c8ea12b
b

hauteur noeud : 57
block :
5e9b3ce51f52dac4062bda5da356f9599d92f6cd7a03f50548fb5d888c8bbdf0

hauteur noeud : 56
block :
3ef5b46718b0e5fb3c313bdc3bfe53561352c22394cfe85bdbcc045e6eac6b7
b

hauteur noeud : 55
block :
c4914a235de403018366ac754c287b8e3b9ed21135ea28b740acb0c045e9ea
2f

hauteur noeud : 54
block :
db37458c48331c6b5c273d3d8b92a01a5e94b3379ca51e378116963db235f0
da

hauteur noeud : 53
block :
0fb835a56bb250926de6398870c4a1fb7606631a1bcc8b9ff9f459f0919a4f3b

hauteur noeud : 52
block :
5942122bc3e4f302a6853b39b6bc21ade1950aafcd295f196bfb1ba010709666

hauteur noeud : 51
block :
93ef945841bc48e752fdc8c6e975d8d53c8162efd52835e67be93102eac7d849

hauteur noeud : 50
block :
abaa30e32f5cb3ff000475cc01e76da4b49398866033780f19ad3c365c4b9fa8

hauteur noeud : 49
block :
fd37d9df57adeccea5a8d6e550921ce97a9b4b31a43d70cb2501e70e34bd841a

hauteur noeud : 48
block :
88e44ed288a5f058fe8068aaad77caf9ee7451e8b40e0de076308a691152ff2b

hauteur noeud : 47
block :
9313fb8a8579d7d6ee67bd972e6b8a1b4188dc08c7f077ff8e67fefeb7b4de97

hauteur noeud : 46
block :
e87f549aa7fcbfb3987d1f5e2e86ab632af7428852a6869b6388d4266d631d23

hauteur noeud : 45
block :
1aeddc037f02f781846fdff66f6024837a3f09251936c7621879d62b4bb650ad

hauteur noeud : 44

block :
68bdac0ab0024edd769a0fd06b3975b77b7a4734b107e46927f9c497ee541091

hauteur noeud : 43
block :
c23b069c07d0ae60a7560d1a4fb909baf73d67d43baa750ffc1dc19518679847

hauteur noeud : 42
block :
94d47bef2f29454090a485e89867e838803016d6d50b3aec369f1f81af54a8da

hauteur noeud : 41
block :
0db8906d6aca9f8d64de191c4da3e5bea0637f4e5131374bb76197f9dfa3e887

hauteur noeud : 40
block :
1c975ff1c4ece746a871d787223562921f7024c0cc6c677c304cfff80e0c8282

hauteur noeud : 39
block :
61acf83d53b52413d3484be6258f3ed0caeebb29587f7918c4d9abbdf94682c2

hauteur noeud : 38
block :
96ac00829a3a93ef087a45241153a8212517af2401abad8b6cfc1567de75fb77

hauteur noeud : 37
block :
751b7a7c51810c43b1d0650dee40f71d9a9d3dbd1c58f39c9c4f3147a2511d31

hauteur noeud : 36

block :
92aa86ffeb90ae4420e098c536f8ef6007e71722d15be5be11a12e26540dc5a4

hauteur noeud : 35
block :
46c8dbd4742fafb5ee63918ede7b59880c55468a02083ff122e1829864cbf3b7

hauteur noeud : 34
block :
335f1f24d29f09493a9f770f1e78f3c86c24db5850b8f9bccc967a1e13821eb6

hauteur noeud : 33
block :
cf00cf5bc71fe826aa3cc726311e63bee43f7568124e15f8deb71166b909caab

hauteur noeud : 32
block :
1d1bd24bf7e26699e3fc2887260b59526e686713ebf16ca0cba99f1876d9481a

hauteur noeud : 31
block :
4d86ecb53a5dd260b279838c17969575a7551b45d6ddf9740ee7c013bb24aa49

hauteur noeud : 30
block :
8a3591f6c832b2d60e780a65213fcbc47a3863be516d32b6995b0209ba6d33fa

hauteur noeud : 29
block :
e4dce700a35a5da7e264ac953effdf21df86642158b6e6ab23c44460a91cc509

hauteur noeud : 28

block :
e5dc7e3d0f6fedf98897f2af0e37aa2764e69273c30c193ed8a7357188403a4f

hauteur noeud : 27
block :
1b088d4beb4b70bf66f59294e790c2b649bb73a8a923ea4757649b44ac6f1f41

hauteur noeud : 26
block :
ba70c8606b74e358e1d62b7127358c0b8813a485122b601c7b68e72bc4289ce7

hauteur noeud : 25
block :
c6209f6392a25b36df74ce40ccc2027f609065dfdfd41c4f67d6c46346024a8a

hauteur noeud : 24
block :
efaad87d858d879c1443599e71142ef687b8604510e74d5a7006dd14e42b94df

hauteur noeud : 23
block :
f96fffd61d1fdaa41eec0b9cbcc95456313c89aec5eaa88f07c2e580a4674a9a

hauteur noeud : 22
block :
86fc2061c0c765db5dda4e300e88d386c69d0decab24f8d46ed91d278ab80f68

hauteur noeud : 21
block :
f68174da5d12f62f7e1c6fb2be8a5ee3022b44613698809e4c67c4e24bd2082f

hauteur noeud : 20

block :
48c46735c704ead1d6f02b2e426e24e9e05100bd406ae5ba1b716ee8629e3e89

hauteur noeud : 19
block :
a9243487c66ee51fcb7c227b4965547b042bc1125891c566e4fb6d92b4438bf1

hauteur noeud : 18
block :
c979a73c5c1a7c1e1ff446ee6164c7446a7fdee42f4ddfbd7fb91f0ebf7a131a

hauteur noeud : 17
block :
ebda967794bbc16b434bc85e251e7b826ef4cf04e076c341888bb0d510dc8dcd

hauteur noeud : 16
block :
da5ab910596fc6f0a60edb84c116a47b5c3509c7c1c974bc0eef9c6d9aa0ecb6

hauteur noeud : 15
block :
4026259498e3403fb65f9dbb952f6e61c087f867c0be1f1cb414755e7e5c23bf

hauteur noeud : 14
block :
e90db8a5eca0d82105fa8e9ba16cd27a86c4c70bd9543d832e35c1a124e488f2

hauteur noeud : 13
block :
7ed7037bd920386b20dcc3e9fd6ff3fb1ada57d6a75e5d942bb4643e0cb12297

hauteur noeud : 12

block :
ead2daf1be5657b9e261a943b1dc34947684f653c14435994437f5d14612b39b

hauteur noeud : 11
block :
03a3720540b92a8e617ac4680b63151d6d570193afc2bb3030a5b99db90611be

hauteur noeud : 10
block :
8265ab89968b607edef2373cca9d9aeebbe7ffa99b35a2854b5f6174d4e1bbab

hauteur noeud : 9
block :
9bf103bfd3f9d69e145be6781befda208df500c0f58381d073be722d84f2b4d8

hauteur noeud : 8
block :
c360b8a3f9788db615ef1c86de41f0abea567136b2d4b7bc6990a99b9cc138b9

hauteur noeud : 7
block :
6bf91cf14843685749660cd135063751df38e0cd36f3f37f2e6880002f568d3e

hauteur noeud : 6
block :
54e905c5168d6601bd14e318cea495d2a3c743e43571f6317123c139a3342fa2

hauteur noeud : 5
block :
a1941a5dec3c9e48435e4e30afec29e4483601a48d17eae8db1124ae2909aa1f

hauteur noeud : 4

block :
f9dde576873656584c50e4e240d2ac6e26bdc01675f8eeae7e4de0f2dbf82f25

hauteur noeud : 3
block :
d33ed48c00a52951036992b32e380876ffbc4a02969d1cc2561c4e3e1c45b9a8

hauteur noeud : 2
block :
88986b0fa3169079a7c4fc3056c283a383a1eb6eacc51506616acf92435633fe

hauteur noeud : 1
block :
ad5431abc8be9c2936c769ce6773f23c948c2ea599bbc2bf4408e86e9146b4d8

hauteur noeud : 0
block :
432581e22508b44d4be937a765a19dc3b50ae19e782c6101c78df57a2c95ae79

vainqueur de l'election : (30f,ccd)