Projet : Blockchain appliqué à un processus électoral

Ce projet porte sur le sujet de l'organisation d'un processus électoral par scrutin numérique implémentant la technologie de la blockchain. Le projet est divisé en plusieurs parties ci-dessous (chaque exercice est codé dans un seul fichier C si possible) :

- Partie I : Implémentation des outils de cryptographie
- Partie II : Création d'un système de déclarations de votes sécurisés par un chiffrement asymétrique
- Partie III : Manipulation d'une base centralisée de déclarations
- Partie IV : Implémentation d'un mécanisme de consensus
- Partie V : Manipulation d'une base décentralisée de déclarations

Ce projet a vu le jour pour répondre à de nombreux problèmes comme le taux d'abstention, la confiance et la transparence des processus électoral centralisé. Puisque la fiabilité des élections centralisé est garantie par le principe de confiance avec autrui, cela peut poser problème si une ou plusieurs personnes trahit cette confiance. L'autre problème est le caractère anonyme d'un vote qui fait que personne ne peut vérifier si le vote a bien été comptabilisée chez le bon candidat.

L'objectif de ce projet consiste à résoudre une grande partie de ses problématiques liée à un processus électoral centralisé en garantissant l'intégrité, la sécurité et la transparence de l'élection tout en désignant le vainqueur final de l'élection.

Partie I : Implémentation des outils de cryptographie

Dans cette partie, Nous allons coder les différents outils cryptographiques nécessaire pour l'obtention des paires de clé publique et secrète unique pour chaque citoyen inscrit sur la liste électoral. Obtention des clés se passent via un protocole de cryptographie asymétrique appelé RSA (ce protocole s'appuie sur la génération de nombres premiers). Il y a 2 types de clés :

- Clé publique : cette clé est envoyée à l'envoyeur permettant de chiffrer son message
- Clé secrète : Permet de déchiffrer les messages à la réception

Exemple connu pour expliquer ce fonctionnement est celui de Alice et Bob souhaitant communiquer entre eux sans que le message soit intercepté par quelqu'un d'autre :

"Bob souhaite envoyer un message à Alice. Pour cela, il utilise la clé publique d'Alice pour chiffrer le message avant de lui envoyer. Une fois reçu, Alice peut utiliser sa clé secrète pour d´déchiffrer le message. Ces clés peuvent aussi servir à signer des messages (pour vérifier la provenance). Plus précisément, Alice peut utiliser sa clé secrète pour signer des messages qu'elle envoie, et sa clé publique permet aux destinataires de vérifier la signature."

Exercice 1 - Résolution du problème de primalité

Cet exercice consiste à développer des outils permettant de générer des nombres premiers efficacement et de vérifier que ces nombres sont bien des nombres premiers.

Fichier : ***primalite.h***

```
#ifndef PRIMALITE_H
#define PRIMALITE_H

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

int is_prime_naive(long p);//verifie si le nombre est premier ou non

long modpow_naive(long a, long m, long n);//modulation exponentiel naive (a^b%n)

int modpow(long a,long m,long n);//modulation exponentiel améliorer par succession elevation au carré

int witness(long a, long b, long d, long p);//test si a est un temoin de Miller pour p

long rand_long(long low, long up);//retourne un long compris entre low et up inclus

int is_prime_miller(long p, int k);//realise le test de Miller-Rabin

long random_prime_number(int low_size, int up_size, int k);//retourne un nombre premier de taille comprise entre low_size et up_size

#endif
```

Fichier : ***primalite.c***

```
#include "primalite.h"
#include <math.h>

//complexite en O(p)
int is_prime_naive(long p){

  for(int i=3;i<p-1;i++){

                if(p%i==0){
                        return 0;
                }
}
return 1;
```

```
}

//complexite en O(a^m)
long modpow_naive(long a, long m, long n){

long res = a;

for(int i=0;i<m-1;i++){

res = res*a;
res = res%n;
}

return res;
}
//complexite en O(log2(m))
int modpow(long a,long m,long n){

   long res = 1;

   while(m > 0){

      if(m%2==1){
         res = (a*res)%n;
      }

      m = m/2;
      a = (a*a)%n;
   }

   return res;
}

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

long x = modpow(a,d,p);

if(x == 1){

return 0;
}
for(long i = 0;i<b;i++){

if(x == p-1){

return 0;
}

x = modpow(x,2,p);
}

return 1;
}
```

```
long rand_long(long low, long up){
return rand() % (up-low+1)+low;
}

int is_prime_miller(long p, int k){

if(p == 2){

return 1;
}
if (!(p & 1) || p <= 1){

return 0;
}

long b = 0;
long d = p-1;

while(!(d & 1)){

d = d/2;
b = b+1;
}

long a;
int i;
for(i =0;i<k;i++){

a = rand_long(2,p-1);
if(witness(a,b,d,p)){

return 0;
}
}

return 1;
}

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

   long entier = 0;

   do{
         entier = rand_long(pow(2,(low_size-1)),pow(2,up_size)-1);

   }while(is_prime_miller(entier,k) == 0);

   return entier;
}
```

Un programme est principalement accompagné d'un fichier .h nommé fichier header (en-tête) et d'un fichier .c liée à celui-ci. Au-dessus, nous avons pris comme exemple les fichiers primalite.h et primalite.c . Le fichier header (.h) consiste à garder les différentes signatures des fonctions et le

code des structures utilisé. Ici dans le fichier primalite.h on inscrit les signatures des différentes fonctions permettant d'interagir sur les structures et les fonctions dans d'autres fichiers. Quand au fichier (.c), on inscrit tous le fonctionnement des fonctions préalablement signer dans le fichier .h liée. Cela permet une meilleure gestion et une aisance pour la réutilisation des fonctions dans d'autres fichiers permettant éviter la répétition de code et le risque de problème.

Description des fonctions principales de primalite.c :

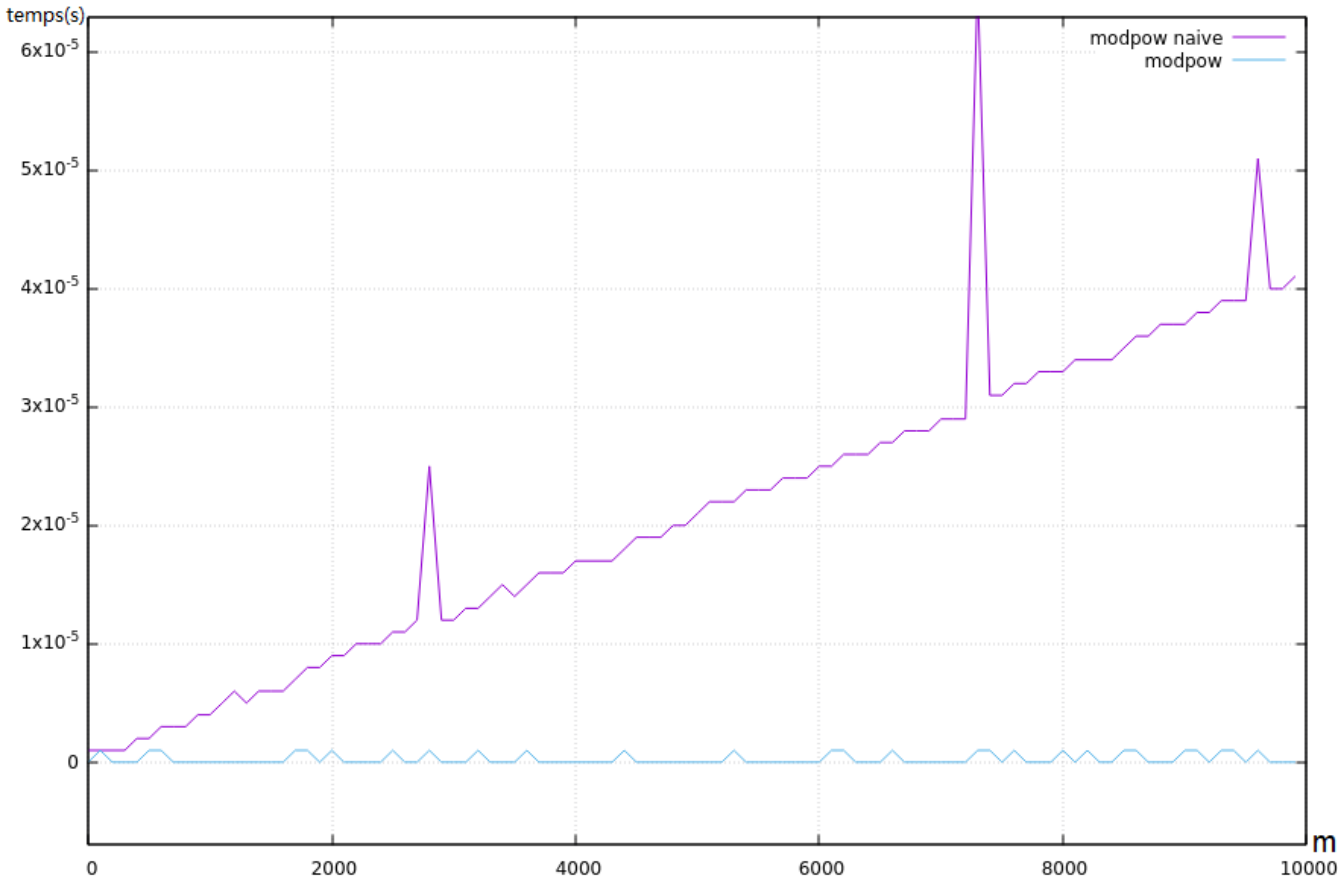
- ***int is_prime_naive(long p);*** / cette fonction consiste à vérifier si le nombre p est un nombre premier (enumère tous les entiers entre 3 et p-1)
- ***long modpow_naive(long a, long m, long n);*** / cette fonction calcule la modulation exponentielle naïvement (a ^ m%n)
- ***int modpow(long a,long m,long n);*** /cette fonction calcule la modulation exponentielle efficacement (grâce à la méthode élévation au carré suivie de modulo)
- ***int witness(long a, long b, long d, long p);*** /cette fonction test si a est un témoin de Miller pour p
- ***long rand_long(long low, long up);*** / cette fonction retourne un long compris entre low et up inclus
- ***int is_prime_miller(long p, int k);*** /cette fonction teste si p est un nombre premier grâce à la méthode Miller-rabbin
- ***long random_prime_number(int low_size, int up_size, int k);*** /cette fonction retourne un nombre premier entre low_size et up_size

Q1.1 : La complexité de la fonction en fonction de p est en O(p)

Q1.2: Le nombre premier qu'on peut tester en moins de 2 millièmes de seconde avec la fonction est 5431.

Q1.3: la complexité est en O(a^m).

Q1.5:



On peut apercevoir que la fonction modpow_naive augmente progressivement en temps de calcule quand m augmentent alors que la fonction modpow stagne un peu près à la même valeur de temps de calcule sans augmentation lorsque m augmente. On constate donc l'efficacité de la fonction modpow comparer à la fonction modpow_naive.

Q1.7: la probabilité d'erreur de l'algorithme est de borne supérieure de (¼)^i avec i valeurs renvoyés

Durant cet exercice, nous avons utilisé différente équations et algorithmes mathématique pour obtenir des valeurs de clés unique de nombre premiers. L'exponentiation modulaire et élévation au carré suivi d'un modulo permet d'obtenir un grand nombre premier sans la nécessité d'avoir une puissance de calcul colossale (se référer à la fonction modpow pour constater le calcul réalisé). Le test de Miller-Rabin est un algorithme randomisé très utile puisqu'il consiste à vérifier si le nombre est premier ou non (très probablement premier). Ce calcule est effectué par équation p = 2^b*d+1 avec p un nombre impair quelconque, b et d deux entiers et a un entier strictement inférieur à p. a est un témoin de Miller pour p si :

- a ^d%p != 1
- a ^(2 ^r)*d%p != -1 pour r ∈ {0, 1, . . . , b − 1}

Exercice 2 - Implémentation du protocole RSA

Durant cet exercice, nous implémenterons le protocole RSA. Pour pouvoir utiliser le protocole RSA et permettre l'envoi de donnée chiffrer nous devons générez des couples de clés publique et secrète. Pour que les échanges soient sécurisés le couple de clé est calculé de tels sorte qu'il soit quasiment impossible de retrouver une clé secrète à partir d'une clé publique même avec la méthode du reverse engineering. Le protocole RSA est fondé sur la difficulté de factorisation de grands entiers. Le protocole nécessite de grands nombres premiers p et q distincts générés aléatoirement :

- Calcule n = p*q et t = (p-1) *(q-1)
- Génère aléatoirement des entiers s inférieur à t jusqu'à trouver PGCD(s,t) = 1
- Détermine u avec s*u%t = 1

Les clés sont constituées de la sorte :

- Public Key : PKey = (s,n)
- Secret Key : SKey = (u,n)

Pour vérifier la validité de PGCD(s,t) = 1 on passe par l'algorithme Euclide étendu. L'algorithme consiste a calculer la valeur de PGCD(s,t) avec s et t deux nombres entiers et u et v des entiers vérifiant l'équation de Bézout :

- s*u+t*v = PGCD(s,t)

Quand PGCD(s,t) = 1, on obtient bien s*u%t = 1

Algorithme d'Euclide étendu donnée dans l'énoncé du projet :

```
long extended_gcd(long s, long t, long *u, long *v){

   if(s == 0){

      *u = 0;
      *v = 1;
      return t;
   }

   long uPrim, vPrim;
   long gcd = extended_gcd(t%s, s, &uPrim, &vPrim);
   *u = vPrim -(t/s)*uPrim;
   *v = uPrim;
   return gcd;

}
```

Fichier : **generate.h**

```
#ifndef GENERATE_H
#define GENERATE_H

#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "primalite.h"

long extended_gcd(long s, long t, long *u, long *v);//fonction representant l'algorithme
d'euclide(PGCD)

void generate_key_values(long p, long q, long *n, long *s, long *u);//genere les valeurs d'une clé

long* encrypt(char* chaine, long s, long n);//chiffre une chaine de caractere avec la clé secrete et
retourne un tableau de long

char* decrypt(long* crypted, int size, long u, long n);//decrypte un tableau de long avec la clé
publique

#endif
```

Fichier : **generate.c**

```
#include "generate.h"

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

   if(s == 0){

      *u = 0;
      *v = 1;
      return t;
   }

   long uPrim, vPrim;
   long gcd = extended_gcd(t%s, s, &uPrim, &vPrim);
   *u = vPrim -(t/s)*uPrim;
   *v = uPrim;
   return gcd;

}

void generate_key_values(long p, long q, long *n, long *s, long *u){

*n = p*q;
long t = (p-1)*(q-1);
long v = 0;
long sp = 0;
```

```
do{
sp = rand_long(1,t);

}while(!(extended_gcd(sp,t,u,&v) == 1 && (sp<t) && (*u > 0) && (sp*(*u))%t == 1));

*s = sp;
}


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

if(chaine == NULL){
printf("Erreur chaine NULL(encrypt)\n");
return NULL;
}

int len = strlen(chaine);
long* encrypt_tab = malloc(sizeof(long)*len);

if(encrypt_tab == NULL){
printf("Erreur d'allocation du tableau encrypt_tab(encrypt)\n");
return NULL;
}
//initialise le tableau a 0
for(int j=0;j<len;j++){
encrypt_tab[j] = 0;
}

//chiffre le message
for(int i=0;i<len;i++){

encrypt_tab[i] = modpow((int)chaine[i],s,n);
}

return encrypt_tab;
}

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

if(crypted == NULL){
printf("Tableau crypted en entree NULL(decrypt)\n");
return NULL;
}

char* message = (char*)malloc(sizeof(char)*(size+1));

if(message == NULL){
printf("Probleme allocation de message(decrypt)\n");
return NULL;
}

int i;
for(i=0;i<size;i++){
message[i] = modpow(crypted[i],u,n);
```

```
}

message[i] = '\0';

return message;
}
```

Description des fonctions principales de generate.c:

- ***long extended_gcd(long s, long t, long *u, long *v) ;*** / fonction calculant les valeurs des clés via l'algorithme d'Euclide étendu
- ***void generate_key_values(long p, long q, long *n, long *s, long *u);*** / génère les valeurs des clés grâce à la fonction extended_gcd
- ***long* encrypt(char* chaine, long s, long n);*** / chiffre le message en chiffrant chaque lettre du message par la clé publique en calculant **[c = m ^ s% n]** avec c le caractère chiffré de m représentant une lettre du message
- ***char* decrypt(long* crypted, int size, long u, long n);*** / déchiffre le tableau crypted représentant le message chiffrer en calculant **[m = c ^ u% n]** avec c représentant une valeur du tableau crypted et m une lettre du message (sans oublier le caractère spécial '\0')

Ce second exercice permettra par la suite de chiffrer des messages et déchiffrer celui-ci sans avoir la possibilité de modifié celui-ci grâce aux protocole RSA et de cryptographie asymétrique.

Ce schéma représente le fonctionnement du protocole RSA, le message au centre est Imperceptible si la personne n'a pas la clé de déchiffrement. Ceci permet de sécuriser un message et cela est actuellement utilisé dans de nombreuse application de messagerie telle que Signal.

1er fichier test :

Fichier : **main.c**

```
#include "generate.h"
#include "primalite.h"

void print_long_vector(long *result, int size){

printf("Vector: [");

for(int i=0;i<size;i++){
```

```
printf("%lx \t",result[i]);
}

printf("]\n");
}

int main(){

srand(time(NULL));

//Generation de cle :
long p = random_prime_number(3,7,5000);
long q = random_prime_number(3,7,5000);

while(p==q){

q = random_prime_number(3,7,5000);
}

long n,s,u;

generate_key_values(p,q,&n,&s,&u);
//Pour avoir des cles positives :

if(u<0){

long t = (p-1)*(q-1);

u = u*t; //on aura toujours s*u mod t = 1
}

//Affichage des cles en hexadecimal
printf("cle_publique = (%lx, %lx) \n",s,n);
printf("cle privee = (%lx, %lx) \n",u,n);

//Chiffrement:
char mess[10] = "Hello";
int len = strlen(mess);
long* crypted = encrypt(mess,s,n);

printf("Initial message: %s \n",mess);
printf("Encoded_representation : \n");
print_long_vector(crypted,len);

//Dechiffrement
char* decoded = decrypt(crypted,len,u,n);
printf("Decoded: %s\n",decoded);

free(crypted);
free(decoded);
```

```
return 0;

}
```

Voici le fichier permettant de vérifier le bon fonctionnement des fonctions précédemment citées.

Partie II : Création d'un système de déclarations de votes sécurisés par un chiffrement asymétrique

Cette partie consiste à mettre en œuvre le système de déclaration et l'interaction du citoyen pendant les élections via des déclarations de vote. Ces déclarations seront effectuées par les clés secrètes des citoyens.

Exercice 3 – Manipulations de structures sécurisées

Dans notre projet, les cartes électorales sont représentées par les couples de clés :

- La clé secrète permet de signer une déclaration de vote, cette clé ne doit être connue que par lui
- La clé publique permet aux autres citoyens d'attester l'authenticité de la déclaration de vote donc de validité de la signature. Cette clé est aussi utilisée pour identifier l'auteur du vote mais aussi si une personne vote en sa faveur.

La déclaration de vote se fait par le chiffrement du message contenant la clé publique du candidat et vérifie la validité de la signature par déchiffrement.

MANIPULATION DE CLES

Le protocole RSA permet la création de couple de clé unique [clé publique et clé secrète].

Fichier : **manipulation_cle.h**

```
#ifndef VOTE_H
#define VOTE_H

typedef struct Key{//structure clé
long val;
long n;

}Key;
```

```
void init_key(Key* key,long val, long n);//initialise la clé avec les valeurs val et n (clé deja alloué)

void init_pair_keys(Key* pKey, Key* sKey, long low_size, long up_size);//initialise les paires de clés avec le protocole RSA (paires clés deja alloué)

char* key_to_str(Key* key);//retourne une chaine de caractere representant la clé (format string : "(%lx,%lx)")

Key* str_to_key(char* str);//retourne la clé representé dans la chaine de caractere


#endif
```

Fichier : **manipulation_cle.c**

```
#include "manipulation_cle.h"
#include "primalite.h"
#include "generate.h"

#define MILLER_NB 5000

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

if(key == NULL){
printf("Erreur key NULL(init_key)\n");
return;
}

key->val = val;
key->n = n;
}

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

if(pKey == NULL || sKey == NULL){
printf("Erreur pKey ou sKey NULL(init_pair_keys)\n");
return;
}

long p = random_prime_number(low_size,up_size,MILLER_NB);
long q = random_prime_number(low_size,up_size,MILLER_NB);

while(p==q){
q = random_prime_number(low_size,up_size,MILLER_NB);
}

long n,s,u;

generate_key_values(p,q,&n,&s,&u);

if(u<0){
```

```
long t = (p-1)*(q-1);
u = u*t;
}

init_key(pKey,s,n);
init_key(sKey,u,n);
}

char* key_to_str(Key* key){

if(key == NULL){
printf("Erreur key NULL(key_to_str)\n");
return NULL;
}

char* key_str = (char*)malloc(sizeof(char)*256);

if(key_str == NULL){
printf("Probleme allocation memoire de key_str(key_to_str)\n");
return NULL;
}

sprintf(key_str,"(%lx,%lx)",key->val,key->n);

return key_str;
}

Key* str_to_key(char* str){

if(str == NULL){
printf("Erreur chaine de caracteres str en entree NULL(str_to_key)\n");
return NULL;
}

Key* k = (Key*)malloc(sizeof(Key));

if(k == NULL){
printf("Probleme allocation memoire de k(str_to_key)\n");
return NULL;
}

long val;
long n;

sscanf(str,"(%lx,%lx)",&val,&n);

k->val = val;
k->n = n;

return k;
}
```

Description des fonctions principales de manipulation_cle.c :

- ***void init_key(Key* key,long val, long n) ;*** / initialise la clé donnée en argument avec val et n (la clé doit être déjà allouer en mémoire)
- ***void init_pair_keys(Key* pKey, Key* sKey, long low_size, long up_size);*** / initialise le couple de clé donnée en argument avec des valeurs aléatoires de nombre premiers compris entre low_size et up_size (la clé publique et secrète doit être déjà allouer en mémoire)
- ***char* key_to_str(Key* key);*** / converti la clé en chaine de caractère contenant la clé sous le format (%lx,%lx) (en hexadécimal)
- ***Key* str_to_key(char* str);*** / récupère la clé contenue dans la chaine de caractère

Les fichiers précédents permettent la manipulation de clé et leur initialisation. Des fonctions sont aussi implémentées pour permettre la conversion des clés en chaine de caractère pour un affichage ou l'écriture de ces clés dans un fichier plus tard.

SIGNATURE & DECLARATIONS SIGNEES

Dans ce projet, une déclaration de vote consiste a une transmission de la clé publique du candidat sur qui porte le vote. Dans un processus de scrutin, chaque citoyens peut réalisé des déclarations de voté signée et validé son authenticité de son vote par le chiffrement de la clé publique du candidat choisis.Cette signature est représenté par un tableau de long qui peut être émis que par le citoyen émettant son vote.le chiffrement de la signature s'effectue a partir de la clé secrète. Le protocole de déclaration de vote est le suivant:

- Le message mess est obtenue en transformant la clé publique du candidat sur lequel l'électeur porte le vote en représentant la clé en chaine de caractère
- L'électeur chiffre le message mess avec sa clé secrète pour générer la signature associée à la déclaration de vote (sous forme de tableau de long)
- L'électeur publie la déclaration sécurisé composé du message mess, la signature de déclaration de vote sur le candidat et sa clé publique. La clé publique de l'auteur permet de déchiffrer la signature et de vérifier l'authenticité du vote en comparant le message mess et la signature déchiffrer. Toute personne souhaitant vérifier l'authenticité de la déclaration peut le réalisé en déchiffrant la signature avec la clé publique.

Fichier : **signature.h**

```
#ifndef SIGNATURE_H
#define SIGNATURE_H

#include "manipulation_cle.h"
#include "generate.h"

typedef struct signature{//signature du vote (permet de verifier la validité du vote)

long* content;//tableau de long contenant le message chiffre
int size;//taille du tableau content
```

```
}Signature;

typedef struct protected{//structure contenant la clé publique de dechiffrement de la signature et le message permettant de verifier que le message dechiffrer est bien valide

Key* pKey;//clé publique de l'auteur de la signature
char* mess;//message a comparer avec le message dechiffrer pour la validité
Signature* sgn;//signature déposer par l'auteur en chiffrant le message avec sa clé privé

}Protected;

Signature* init_signature(long* content, int size);//créer et initialise une signature avec le tableau contenant le message deja chiffre et la taille du tableau

Signature* sign(char* mess, Key* sKey);//permet de chiffrer le message avec la clé secrete et de retourne une signature initialisée

char* signature_to_str(Signature* sgn);//retourne une chaine de caractere representant la signature (format string : "#x0#x1#...#xn#" avec xi le ieme entier du tableau)

Signature* str_to_signature(char* str);//retourne la signature representé par la chaine de caractere

Protected* init_protected(Key* pKey,char* mess,Signature* sgn);//creer et initialise une declaration signé a partir d'une clé publique,d'un message et d'une signature

int verify(Protected* pr);//verifie que la declaration est bien valide

char* protected_to_str(Protected* p);//retourne une chaine de caractere representant la declaration (format string : "(%lx,%lx) (%lx,%lx) #x0#x1#...#xn#" avec la clé publique,le message et la signature)

Protected* str_to_protected(char* str);//retourne une declaration representé par la chaine de caractere

void libere_signature(Signature* sgn);//libere la signature en memoire

void libere_protected(Protected* pr);//libere la declaration signées en memoire

#endif
```

Fichier : **signature.c**

```
#include "signature.h"

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

if(content == NULL){
printf("tableau content NULL en entree (init_signature)\n");
return NULL;
}
```

```
   Signature* sgn = (Signature*)malloc(sizeof(Signature));

   if(sgn == NULL){
         printf("Probleme allocation de sgn (init_signature)\n");
         return NULL;
   }

   sgn->content = (long*)malloc(sizeof(long)*size);

   if(sgn->content == NULL){
         printf("Probleme allocation memoire de content(init_signature)\n");
         free(sgn);
         return NULL;
   }

   sgn->size = size;

   memcpy(sgn->content,content,sizeof(long)*size);//on recopie la memoire du tableau
content
   free(content);//on libere le tableau content (eviter les fuites memoires venant de la
fonction str_to_signature donnée dans l'énoncé

   return sgn;
}

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

   if(mess == NULL || sKey == NULL){
         printf("Erreur mess ou sKey NULL en entree(sign)\n");
         return NULL;
   }

   long* tab_encrypt = encrypt(mess,sKey->val,sKey->n);

   Signature* sgn = init_signature(tab_encrypt,strlen(mess));

   return sgn;
}

char* signature_to_str(Signature* sgn){

if(sgn == NULL){
printf("Erreur sgn NULL en entree(signature_to_str)\n");
return NULL;
}

char* result = malloc(10*sgn->size*sizeof(char));

if(result == NULL){
printf("Erreur allocation result (signature_to_str)\n");
return NULL;
}

result[0] = '#';
```

```
int pos = 1;
char buffer[156];

for(int i=0;i<sgn->size;i++){

sprintf(buffer,"%lx",sgn->content[i]);

for(int j=0;j < strlen(buffer);j++){

result[pos] = buffer[j];
pos = pos+1;
}

result[pos] = '#';
pos = pos+1;

}

result[pos] = '\0';
result = realloc(result, (pos+1)*sizeof(char));

return result;
}

Signature* str_to_signature(char* str){

if(str == NULL){
printf("Erreur str NULL en entree(str_to_signature)\n");
return NULL;
}

int len = strlen(str);
long* content = (long*)malloc(sizeof(long)*len);

if(content == NULL){
printf("Probleme allocation content(str_to_signature)\n");
return NULL;
}

int num=0;
char buffer[256];
int pos = 0;

for(int i=0;i < len;i++){

if(str[i] != '#'){

buffer[pos] = str[i];
pos= pos+1;
}else{

if(pos != 0){

buffer[pos] = '\0';
```

```
sscanf(buffer, "%lx", &(content[num]));
num = num+1;
pos = 0;
}
}
}

content = realloc(content,num*sizeof(long));

return init_signature(content,num);
}


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

if(pKey == NULL || mess == NULL || sgn == NULL){
printf("Erreur pKey, mess ou sgn NULL en entree(init_protected)\n");
return NULL;
}

Protected* p = (Protected*)malloc(sizeof(Protected));

if(p == NULL){
printf("Probleme allocation de p(init_protected)\n");
return NULL;
}

p->pKey = (Key*)malloc(sizeof(Key));

if(p->pKey == NULL){
printf("Probleme allocation de pKey(init_protected)\n");
free(p);
return NULL;
}
memcpy(p->pKey,pKey,sizeof(Key));//on copie la memoire de la clé

p->mess = strdup(mess);

if(p->mess == NULL){
printf("Probleme allocation de mess(init_protected)\n");
free(p->pKey);
free(p);
return NULL;
}

long* content = (long*)malloc((sgn->size)*sizeof(long));
memcpy(content,sgn->content,(sgn->size)*sizeof(long));//on copie la memoire du tableau
content

//on realise des copies et non des affections pour eviter lors d'un free en externe d'affecter
les structures et vider les contenues sans le vouloir

p->sgn = init_signature(content,sgn->size);
```

```
if(p->sgn == NULL){
printf("Probleme allocation de sgn(init_protected)\n");
free(p->mess);
free(p->pKey);
free(p);
return NULL;
}

return p;
}

int verify(Protected* pr){

if(pr == NULL || pr->pKey == NULL || pr->sgn == NULL){
printf("Erreur pr (ou sous pointeur) NULL en entrée(verify)\n");
return 0;
}

Signature* sgn = pr->sgn;
Key* pKey = pr->pKey;
char* message_decrypt;

message_decrypt = decrypt(sgn->content,sgn->size,pKey->val,pKey->n);

//si la signature est bien correct (message dechiffrer correspond au message) on retourne 1
if(strcmp(message_decrypt,pr->mess) == 0){
free(message_decrypt);
return 1;
}

//sinon on retourne 0
free(message_decrypt);
return 0;
}

char* protected_to_str(Protected* p){

if(p == NULL ||p->pKey == NULL || p->sgn == NULL){
printf("Erreur p (ou sous pointeur) NULL en entree(protected_to_str)\n");
return NULL;
}

char* str = (char*)malloc(sizeof(char)*256);

if(str == NULL){
printf("Probleme allocation de str(protected_to_str)\n");
return NULL;
}

char* key_str = key_to_str(p->pKey);
char* signature_str = signature_to_str(p->sgn);

sprintf(str,"%s %s %s",key_str,p->mess,signature_str);
```

```
free(key_str);
free(signature_str);
return str;
}

Protected* str_to_protected(char* str){

if(str == NULL){
printf("Erreur str NULL en entree(str_to_protected)\n");
return NULL;
}

Protected* p = (Protected*)malloc(sizeof(Protected));

if(p == NULL){
printf("Probleme allocation de p(str_to_protected)\n");
return NULL;
}

char* str_pKey = (char*)malloc(sizeof(char)*256);

if(str_pKey == NULL){
printf("Probleme allocation de str_pKey(str_to_protected)\n");
free(p);
return NULL;
}

char* str_mess = (char*)malloc(sizeof(char)*256);

if(str_mess == NULL){
printf("Probleme allocation de str_mess(str_to_protected)\n");
free(str_pKey);
free(p);
return NULL;
}

char* str_sgn = (char*)malloc(sizeof(char)*256);

if(str_sgn == NULL){
printf("Probleme allocation de str_sgn(str_to_protected)\n");
free(str_mess);
free(str_pKey);
free(p);
return NULL;
}

sscanf(str,"%s %s %s ",str_pKey,str_mess,str_sgn);

p->pKey = str_to_key(str_pKey);
p->sgn = str_to_signature(str_sgn);
p->mess = (char*)malloc(sizeof(char)*((p->sgn->size)+1));

if(p->mess == NULL){
```

```
printf("Probleme allocation de mess(str_to_protected)\n");

free(str_mess);

if(p->pKey != NULL){
free(p->pKey);
}
if(p->sgn != NULL){
free(p->sgn->content);
free(p->sgn);
}
free(p);
}

strncpy(p->mess,str_mess,(p->sgn->size)+1);//on copie le nombre de caractère exacte de la chaine de caractere de la signature dans la structure

free(str_mess);
free(str_pKey);
free(str_sgn);

return p;
}

void libere_signature(Signature* sgn){

free(sgn->content);
free(sgn);
}

void libere_protected(Protected* pr){

free(pr->pKey);
free(pr->mess);
libere_signature(pr->sgn);
free(pr);
}
```

Description des fonctions principales de manipulation signature.c :

- ***Signature* init_signature(long* content, int size)*** **; /** initialisation de la signature avec le tableau contenant le message chiffrer et la taille du tableau
- ***Signature* sign(char* mess, Key* sKey);*** **/** chiffre le message avec la clé secrète et initialise la signature avec le message chiffrée
- ***char* signature_to_str(Signature* sgn);*** / retourne une chaine caractère contenant les informations de la signature donnée en argument au format #x0#x1#...#xn#" avec xi le i-ème entier du tableau
- ***Signature* str_to_signature(char* str);*** / récupère la signature depuis la chaine de caractère

- ***Protected* init_protected(Key* pKey,char* mess, Signature* sgn)*;** / initialise une déclaration de vote avec la clé publique, le message et la signature
- ***int verify(Protected* pr)*;** / vérifie la déclaration signée si elle est valide ou non (retourne 1 si elle est valide sinon 0)
- ***char* protected_to_str(Protected* p) ;*** / retourne une chaine caractère contenant les informations de la déclaration signée sous le format "(%lx,%lx) (%lx,%lx) #x0#x1#...#xn#"
- ***Protected* str_to_protected(char* str);*** / récupère la déclaration depuis la chaine de caractère
- ***void libere_signature(Signature* sgn);*** / libère la signature en mémoire
- ***void libere_protected(Protected* pr);*** / libère la déclaration signée en mémoire

Les diffèrentes fonctions de ce fichier permet de représenter les déclarations de vote de chaques citoyens et permet de verifier l'authenticité de la déclaration de vote et leur unicité.

2ème fichier test :

Fichier : **main2.c**

```
#include "primalite.h"
#include "generate.h"
#include "manipulation_cle.h"
#include "signature.h"

void print_long_vector(long *result, int size){

printf("Vector: [");

for(int i=0;i<size;i++){

printf("%lx \t",result[i]);
}

printf("]\n");
}

int main(void){

srand(time(NULL));

//Testing Init Keys
Key* pKey = malloc(sizeof(Key));
Key* sKey = malloc(sizeof(Key));

init_pair_keys(pKey,sKey,3,7);

printf("pKey: %lx, %lx\n",pKey->val,pKey->n);
printf("sKey: %lx, %lx\n",sKey->val,sKey->n);
```

```
//Testing Key Serialisation
char* chaine = key_to_str(pKey);

printf("key_to_str: %s \n",chaine);

Key* k = str_to_key(chaine);

printf("str_to_key: %lx, %lx \n",k->val,k->n);

        free(chaine);
free(k);

//Testing signature

//Candidate keys:
Key* pKeyC = malloc(sizeof(Key));
Key* sKeyC = malloc(sizeof(Key));

init_pair_keys(pKeyC,sKeyC,3,7);

//Declaration:
char* mess = key_to_str(pKeyC);
char* pKey_str = key_to_str(pKey);

printf("%s vote pour %s\n",pKey_str,mess);

Signature* sgn = sign(mess, sKey);

printf("signature: ");
print_long_vector(sgn->content,sgn->size);

chaine = signature_to_str(sgn);
printf("signature_to_str: %s \n", chaine);

libere_signature(sgn);

sgn = str_to_signature(chaine);
printf("str_to_signature: ");
print_long_vector(sgn->content,sgn->size);

free(pKey_str);
free(chaine);

//Testing protected:
Protected* pr = init_protected(pKey,mess,sgn);

free(mess);
libere_signature(sgn);


//Verification:
```

```
if(verify(pr)){

printf("Signature valide \n");
}else{

printf("Signature non valide\n");
}

chaine = protected_to_str(pr);
printf("protected_to_str: %s\n", chaine);

libere_protected(pr);

pr = str_to_protected(chaine);

free(chaine);

pKey_str = key_to_str(pr->pKey);
char* sgn_str = signature_to_str(pr->sgn);

printf("str_to_protected: %s %s %s\n",pKey_str,pr->mess,sgn_str);

libere_protected(pr);
free(pKey_str);
free(sgn_str);

free(pKey);
free(sKey);
free(pKeyC);
free(sKeyC);

return 0;
}
```

Voici le 2ème fichier main permettant de tester les fonctions précédentes.

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

Pour pouvoir mettre en place la simulation du scrutin, nous devons génerer les couples de clés publique et secrète unique représentant les cartes éléctorales de chaques citoyens. Après que les déclarations de votes soit signée par les citoyens avec leur clé secrète (qui la garde impèrativement secrète). Le système collectent et vérifie que les déclarations sont bien correcte au fur et a mesure des votes puis ensuite comptabilise chaque vote a la fin du scrutin. Le système vérifie aussi que chaque citoyen n'a voté qu'une fois lors du scrutin. La simulation du processus de vote s'effectue a l'aide de trois fichier qui sont un fichier contenant les clés de tous les citoyens, un fichier contenant les clés publiques des candidats et un fichier de déclarations signées.

Fichier : **generate_data.h**

```
#ifndef GENERATE_DATA_H
#define GENERATE_DATA_H

#include <stdio.h>
#include "primalite.h"
#include "generate.h"
#include "manipulation_cle.h"
#include "signature.h"

void generate_random_data(int nv, int nc);

/*
genere nv citoyens et nv candidates parmi les citoyens et les fichiers "keys.txt","candidates.txt" et
"declarations.txt"

- "keys.txt" : contient les nv couples de clés différents des citoyens (un couple par ligne)

- "candidates.txt" : contient les clés publiques de tous les candidats choisis aléatoirement parmi
les citoyens (une clé publique par ligne)

- "declarations.txt" : contient les declarations signées de chaques citoyens (une declaration par
ligne)

*/

#endif
```

Fichier : **generate_data.c**

```
#include "generate_data.h"

#define LOW_SIZE 3
#define UP_SIZE 7

void generate_random_data(int nv, int nc){

FILE* fkey = fopen("keys.txt","w");
FILE* fcandidates = fopen("candidates.txt","w");
FILE* fdeclarations = fopen("declarations.txt","w");

Key** tab_pKeys = (Key**)malloc(sizeof(Key*)*nv);

if(tab_pKeys == NULL){
printf("Erreur allocation tab_pKeys(generate_random_data)\n");
return;
}

Key** tab_sKeys = (Key**)malloc(sizeof(Key*)*nv);
```

```
if(tab_sKeys == NULL){
printf("Erreur allocation tab_sKeys(generate_random_data)\n");
free(tab_pKeys);
return;
}

Key** tab_pKeys_candidates = (Key**)malloc(sizeof(Key*)*nc);

if(tab_pKeys_candidates == NULL){
printf("Erreur allocation tab_pKeys_candidates(generate_random_data)\n");
free(tab_pKeys);
free(tab_sKeys);
return;
}

int tab_b[nv];//tableau de booléen utilisé pour eviter les doublons lors des choix des candidats
int r_pKey = 0;
char* pKey_str = NULL;
char* sKey_str = NULL;
char* pr_str = NULL;
Signature* sgn = NULL;
Protected* pr = NULL;

//initialisation du tableau booléen a 0
for(int l=0;l<nv;l++){
tab_b[l] = 0;
}

//allocation des cases des tableaux
for(int i=0;i<nv;i++){
tab_pKeys[i] = (Key*)malloc(sizeof(Key));
tab_sKeys[i] = (Key*)malloc(sizeof(Key));
}
//creation et ecriture des clés dans le fichiers "keys.txt"
for(int j=0;j<nv;j++){

init_pair_keys(tab_pKeys[j],tab_sKeys[j],LOW_SIZE,UP_SIZE);
pKey_str = key_to_str(tab_pKeys[j]);
sKey_str = key_to_str(tab_sKeys[j]);
fprintf(fkey,"%s %s\n",pKey_str,sKey_str);

free(pKey_str);
free(sKey_str);
}

fclose(fkey);

//recupere et ecrit les clés publiques des candidates dans le fichier "candidates.txt"
r_pKey = rand()%nv;//retourne un nombre aléatoire entre 0 et nv (nv representant le nombre de
citoyen créer precedement)
tab_b[r_pKey] = 1;
```

```
tab_pKeys_candidates[0] = tab_pKeys[r_pKey];
pKey_str = key_to_str(tab_pKeys_candidates[0]);
fprintf(fcandidates,"%s\n",pKey_str);//ecrit le 1er candidat dans le fichier

free(pKey_str);

//inscription des restes des candidates dans le fichier (nc represente le nombre de candidates
selectionné);
for(int k=1;k<nc;k++){

while(tab_b[r_pKey]){
r_pKey = rand()%nv;
}

tab_b[r_pKey] = 1;
tab_pKeys_candidates[k] = tab_pKeys[r_pKey];
pKey_str = key_to_str(tab_pKeys_candidates[k]);
fprintf(fcandidates,"%s\n",pKey_str);

free(pKey_str);
}

fclose(fcandidates);

//ecriture des declarations signées dans le fichier "declarations.txt"
for(int v=0;v<nv;v++){

pKey_str = key_to_str(tab_pKeys_candidates[rand()%nc]);
sgn = sign(pKey_str,tab_sKeys[v]);
pr = init_protected(tab_pKeys[v],pKey_str,sgn);

free(pKey_str);
libere_signature(sgn);

//verifie si la declaration est bien valide avant de l'ecrire dans le fichier
if(verify(pr)){

pr_str = protected_to_str(pr);
fprintf(fdeclarations,"%s\n",pr_str);

free(pr_str);
}

libere_protected(pr);
}

fclose(fdeclarations);

//libere les tableaux et leur cases
for(int x=0;x<nv;x++){
```

```
free(tab_pKeys[x]);
free(tab_sKeys[x]);
}

free(tab_pKeys);
free(tab_sKeys);
free(tab_pKeys_candidates);
}
```

Description des fonctions principales de manipulation_generate_data.c :

- ***void generate_random_data(int nv, int nc);***

/ Génère des fichiers “keys.txt”,”candidates.txt” et “declarations.txt” :

Keys.txt : contient les nv clés de citoyens

Candidates.txt : contient les nc clés publiques de candidates

Declarations.txt : contient les déclarations signées des citoyens

Partie III : Manipulation d’une base centralisée de déclarations

Dans cette partie, on représente un système de vote centralisé. Le système de vote récupère les déclarations de vote signée et annonce le vainqueur de l’élection a tous les citoyens. Les déclarations de votes sont inscrit dans le fichier declarations.txt au fur et mesure que les votes sont récupérés. Une fois que scrutin est clos, les données sont chargées dans une liste chaînée.

La vérification des données s’effectue par le système en récuperant l’ensemble des clés publiques des citoyens et des candidats qui sont stockés dans les fichiers keys.txt et candidates.txt

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

Dans cette exercice, nous allons développé des fonctions permettant de lire des fichiers et récuperer les clés des citoyens et des candidates en les stockants dans une listes chainées de clés publiques. Le fichier declarations.txt permet de recupérer les déclarations signées et de les stocker dans une liste chaînée de déclarations signées.

Fichier : **import_data.h**

```
#ifndef IMPORT_DATA_H
#define IMPORT_DATA_H

#include <stdio.h>
#include <string.h>
#include "primalite.h"
#include "generate.h"
#include "manipulation_cle.h"
#include "signature.h"
```

```
typedef struct cellKey{//structure representant une liste de clés (cellule clé)

Key* data;// data contient les données de la clé publique
struct cellKey* next;//clé suivante

}CellKey;

typedef struct cellProtected{//structure representant une liste de declarations signées
(cellule declaration)

Protected* data;//data contient les données de la déclarations signées
struct cellProtected* next;//déclarations suivante

}CellProtected;

CellKey* create_cell_key(Key* key) ;//creer et initialise une cellule clé avec la clé donnée en
argument

CellKey* inserer_tete_key(CellKey* list_key, Key* key) ;//insere une clé en tete de liste

CellKey* read_public_keys(char* pKeys_file);//lis le fichier "pKeys_file" contenant des clés et
retourne la liste des clés

void print_list_keys(CellKey* LCK) ;//affiche la liste des clés

void delete_cell_key(CellKey* c);//supprime une cellule clé

void delete_list_keys(CellKey* l);//supprime la liste de clés

CellProtected* create_cell_protected(Protected* pr);//creer et initialise une cellule
declaration avec la declaration signée donnée en argument

CellProtected* inserer_tete_protected(CellProtected* cl, Protected* pr);//insere une
declaration en tete de liste

CellProtected* read_protected(char* declarations_file);//lis le fichier "declarations_file"
contenant les declarations et retourne la liste des declarations

void print_list_protected(CellProtected* c);//affiche la liste des declarations

void delete_cell_protected(CellProtected* c);//supprime une cellule declarations

void delete_list_protected(CellProtected* cl);//supprime la liste de declarations

CellProtected* verify_declarations(CellProtected* cp);//verifie les declarations et supprime
les declarations fausses de la liste

void libere_contenue_cellProtected_list(CellProtected* cp);//libere les contenues de
cellProtected

#endif
```

Fichier : **import_data.c**

```
#include "import_data.h"

CellKey* create_cell_key(Key* key){

if(key == NULL){

printf("clé valeur NULL(create_cell_key)\n");
return NULL;
}

CellKey* ck = (CellKey*)malloc(sizeof(CellKey));

if(ck == NULL){
printf("Erreur d'allocation CellKey(create_cell_key)\n");
return NULL;
}

ck->data = key;
ck->next = NULL;

return ck;
}

CellKey* inserer_tete_key(CellKey* list_key, Key* key){

if(key == NULL){
printf("clé valeur NULL(inserer_tete_key)\n");
}

      if(list_key == NULL){

         list_key = create_cell_key(key);
         return list_key;
      }

CellKey* k = create_cell_key(key);

k->next = list_key;
return k;
}

CellKey* read_public_keys(char* pKeys_file){

char* buffer = (char*)malloc(sizeof(char)*256);

if(buffer == NULL){
printf("Erreur allocation buffer(read_public_keys)\n");
return NULL;
}

char* str_key = (char*)malloc(sizeof(char)*256);
```

```
if(str_key == NULL){
printf("Erreur allocation str_key(read_public_keys)\n");
free(buffer);
return NULL;
}

CellKey* ck = NULL;

FILE* f = fopen(pKeys_file,"r");

if(f == NULL){

printf("Erreur ouverture du fichier\n");
free(buffer);
free(str_key);
return NULL;
}
//recupere les clés du fichier
while(fgets(buffer,256,f) != NULL){

sscanf(buffer,"%s",str_key);
ck = inserer_tete_key(ck,str_to_key(str_key));
}

fclose(f);
free(buffer);
free(str_key);

return ck;
}

void print_list_keys(CellKey* LCK){

if(LCK == NULL){
return;
}

CellKey* tmp = LCK;
char* key_str = NULL;

while(tmp != NULL){

key_str = key_to_str(tmp->data);

printf("%s\n",key_str);
free(key_str);

tmp = tmp->next;

}
}

void delete_cell_key(CellKey* c){
```

```
if(c != NULL){
free(c->data);
free(c);
}
}

void delete_list_keys(CellKey* l){

if(l==NULL){
return;
}

CellKey* tmp_l = NULL;

while(l != NULL){

tmp_l = l->next;
delete_cell_key(l);
l = tmp_l;

}
}

CellProtected* create_cell_protected(Protected* pr){

CellProtected* cp = (CellProtected*)malloc(sizeof(CellProtected));

if(cp == NULL){
printf("Erreur allocation cellprotected(create_cell_protected)\n");
return NULL;
}

cp->data = pr;
cp->next = NULL;

return cp;
}

CellProtected* inserer_tete_protected(CellProtected* cl, Protected* pr){

if(cl == NULL){
cl = create_cell_protected(pr);
return cl;
}

CellProtected* c = create_cell_protected(pr);

c->next = cl;
return c;
}

CellProtected* read_protected(char* declarations_file){

char* buffer = (char*)malloc(sizeof(char)*256);
```

```
if(buffer == NULL){
printf("Erreur allocation buffer(read_public_keys)\n");
return NULL;
}

CellProtected* cp = NULL;

FILE* f = fopen(declarations_file,"r");

if(f == NULL){

printf("Erreur ouverture du fichier\n");
free(buffer);
return NULL;
}

while(fgets(buffer,256,f) != NULL){
cp = inserer_tete_protected(cp,str_to_protected(buffer));
}

fclose(f);
free(buffer);

return cp;
}

void print_list_protected(CellProtected* c){

if(c == NULL){
return;
}

CellProtected* tmp = c;
char* pr_str = NULL;

while(tmp != NULL){

pr_str = protected_to_str(tmp->data);

printf("%s\n",pr_str);
free(pr_str);

tmp = tmp->next;
}
}

void delete_cell_protected(CellProtected* c){

if(c != NULL){
libere_protected(c->data);
free(c);
}
}
```

```
void delete_list_protected(CellProtected* cl){

if(cl==NULL){
return;
}

CellProtected* tmp_cl = NULL;

while(cl != NULL){

tmp_cl = cl->next;
delete_cell_protected(cl);
cl = NULL;
cl = tmp_cl;

}
}

CellProtected* verify_declarations(CellProtected* cp){

CellProtected* tmp = cp;
CellProtected* pred = NULL;

while(tmp!=NULL){

//si declaration fausse on supprime de la liste
if(verify(tmp->data) == 0){
if(pred == NULL){

cp = cp->next;
delete_cell_protected(tmp);
tmp = NULL;
pred = tmp;
tmp = cp;

}else{
pred->next = tmp->next;
delete_cell_protected(tmp);
tmp = pred->next;
}
}
//sinon on passe au suivant
else{
   pred = tmp;
   tmp = tmp->next;
}
}
return cp;
}

void libere_contenue_cellProtected_list(CellProtected* cp){

CellProtected* tmp = cp;
```

```
while(tmp!=NULL){
libere_protected(tmp->data);
tmp = tmp->next;
}
}
```

Description des fonctions principales de manipulation import_data.c :

- **CellKey* create_cell_key(Key* key) ;** / créer une cellule clé et initialise avec la clé
- **CellKey* inserer_tete_key(CellKey* list_key, Key* key) ;** / insère en tête la clé dans la liste des clés
- **CellKey* read_public_keys(char* pKeys_file);** / lis le fichier et récupère les clés publiques
- **void print_list_keys(CellKey* LCK) ;** / affiche la liste des clés
- **void delete_cell_protected(CellProtected* c) ;** / supprime une cellule de déclaration et les données
- **void delete_list_protected(CellProtected* cl) ;** / supprime la liste des cellules de déclarations et ces données
- **CellProtected* verify_declarations(CellProtected* cp);** / vérifie les déclarations signées
- **void libere_contenue_cellProtected_list(CellProtected* cp);** / libère les contenues des cellules de déclarations

Exercice 6 – Détermination du gagnant de l'élection

Une fois toutes les donneés collectées, le système retire les déclarations falsifiés et détermine le gagnant de l'élection.

Fichier : **hash.h**

```
#ifndef HASH_H
#define HASH_H

#include <stdio.h>
#include "primalite.h"
#include "generate.h"
#include "manipulation_cle.h"
#include "signature.h"
#include "import_data.h"

typedef struct hashcell{

   Key* key;//clé publique du citoyen
   int val; //valeur permettant de savoir si le citoyen a voter ou non
```

```
}HashCell;

typedef struct hashtable{

   HashCell** tab;//tableau d'hachage contenant les cellules de clés (probing lineaire)
   int size;//taille du tableau d'hachage

}HashTable;


HashCell* create_hashcell(Key* key);//creer et initialise une HashCell

int hash_function(Key* key, int size);//fonction hachage retournant la position de la clé dans la table de hachage

int find_position(HashTable* t, Key* key);//retrouve la position de la clé dans la table de hachage si existe sinon la position a laquelle elle devrait etre (probing lineaire)

HashTable* create_hashtable(CellKey* keys, int size);//creer et initiale la table de hachage avec sa taille "size" et la liste des clés donnée en entrée

void delete_hashtable(HashTable* t);//supprime la table de hachage

Key* compute_winner(CellProtected* decl, CellKey* candidates, CellKey* voters, int sizeC, int sizeV);//retourne la clé publique du gagnant de l'election a partir des déclaration signées ,des clés publiques des candidats et des voteurs donnée en entrée (sizeC et SizeV taille des tables de hachages pour candidates et citoyens(=voters))

#endif
```

Fichier : **hash.c**

```
#include "hash.h"

HashCell* create_hashcell(Key* key){

   HashCell* hc = (HashCell*)malloc(sizeof(HashCell));

   if(hc == NULL){
         printf("Erreur allocation hashcell(create_hashcell)\n");
         return NULL;
   }

   hc->key = (Key*)malloc(sizeof(Key));

   if(hc->key == NULL){
         printf("Erreur allocation key(create_hashcell)\n");
         return NULL;
   }

   init_key(hc->key,key->val,key->n);
```

```
   hc->val = 0;

   return hc;
}

int hash_function(Key* key, int size){

   return (key->val)%size;
}

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

   int i = hash_function(key,t->size);
   int j = i;
   int cpt = 0;

   HashCell* cell = t->tab[i];//recupere emplacement de la clé donnée par hash_function

   //s'il n'y aucune clé ou si la clé est bien a cette position on retourne la position(table de hachage
fonctionne sous probing lineaire)
   if( cell == NULL || ((cell->key->val == key->val) && (cell->key->n == key->n))){
      return i;
   }
   //on cherche sur les cases suivantes si la clé est bien la
   else{

      while(cpt < t->size){

          if(cell == NULL || ((cell->key->val == key->val) && (cell->key->n = key->n))){
          return j;
          }
        j = (j+1)%t->size; //si j>size on repart au debut table de hachage avec "%t->size"
        cpt++;
        cell = t->tab[j];
      }

      //si la clé n'est pas trouvé on retourne la position ou elle devait etre
      return i;
   }
}

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

   HashTable* t = (HashTable*)malloc(sizeof(HashTable));

   if(t == NULL){
          printf("Erreur d'allocation structure hashtable(create_hashtable)\n");
          return NULL;
   }

   t->tab = (HashCell**)malloc(sizeof(HashCell*)*size);
```

```
  if(t->tab == NULL){
        printf("Erreur d'allocation hashtable(create_hashtable)\n");
        free(t);
        return NULL;
  }

  t->size = size;

  CellKey* tmp = keys;
  HashCell* hc = NULL;

  //initialise le tableau a NULL
  for(int i=0;i<size;i++){
    t->tab[i] = NULL;
  }

  int k = 0;
  int j = 0;
  int cpt = 0;

  //on place les clés dans la table de hachage selon la methode du probing lineaire
  while(tmp!=NULL){

    k = hash_function(tmp->data,size);

    if(t->tab[k] == NULL){

      hc = create_hashcell(tmp->data);
      t->tab[k] = hc;
    }
    else{

      j=k;

      while(t->tab[j] != NULL && cpt<t->size){

cpt++;
        j = (j+1)%t->size;
      }

      hc = create_hashcell(tmp->data);
      t->tab[j] = hc;
    }

    tmp = tmp->next;
  }

  return t;
}
```

```
void delete_hashtable(HashTable* t){

  HashCell** tab = t->tab;

  for(int i=0;i<t->size;i++){

        if(tab[i] != NULL){

              free(tab[i]->key);
              free(tab[i]);
        }
  }
  free(t->tab);
  free(t);
}

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

HashTable* hc = create_hashtable(candidates,sizeC);
HashTable* hv = create_hashtable(voters,sizeV);

CellProtected* d = decl;
int kv = 0;
int kc = 0;
Key* pKeyc = NULL;
char* pKeyc_str = NULL;
HashCell* winner = NULL;

//verification et comptabilisation des votes
while(d != NULL){

kv = find_position(hv,d->data->pKey);
pKeyc = str_to_key(d->data->mess);
kc = find_position(hc,pKeyc);

if((hc->tab)[kc] != NULL){

pKeyc_str = key_to_str((hc->tab)[kc]->key);

if((!(hv->tab)[kv]->val) && (strcmp(pKeyc_str,d->data->mess) == 0)){
(hv->tab)[kv]->val = 1;
(hc->tab)[kc]->val += 1;
}

free(pKeyc_str);
}

free(pKeyc);
pKeyc = NULL;
d = d->next;
```

```
}

//Compare le nombre de vote des candidates et recupere la clé publique du gagnant
for(int i=0;i<sizeC;i++){

if(winner == NULL && ((hc->tab)[i]!= NULL)){

winner = (HashCell*)malloc(sizeof(HashCell));
winner->key = (Key*)malloc(sizeof(Key));
winner->key->val = (hc->tab)[i]->key->val;
winner->key->n = (hc->tab)[i]->key->n;
winner->val = (hc->tab)[i]->val;

}else if(((hc->tab)[i] != NULL) && (winner->val<(hc->tab[i]->val))){

winner->key->val = (hc->tab)[i]->key->val;
winner->key->n = (hc->tab)[i]->key->n;
winner->val = (hc->tab)[i]->val;

}
}

pKeyc = winner->key;
free(winner);

//suppression des hashtables
delete_hashtable(hc);
delete_hashtable(hv);

return pKeyc;
}
```

Description des fonctions principales de manipulation hash.c :

- **HashCell* create_hashcell(Key* key) ;** / créer une cellule pour table hachage et l'initialise avec la clé en argument
- **int hash_function(Key* key, int size);** / fonction de hachage retournant la position de la clé dans la table de hachage
- **int find_position(HashTable* t, Key* key);** / si la clé est trouvé dans la table retourne sa position s'il elle existe sinon elle retourne l'emplacemeent ou elle devait etre (probing lineaire)
- **HashTable* create_hashtable(CellKey* keys, int size);** / création d'une table de hachage de taille size et initialise la table hachage avec la liste des clés
- **void delete_hashtable(HashTable* t);** / supprime la table de hachage
- **Key* compute_winner(CellProtected* decl, CellKey* candidates, CellKey* voters, int sizeC, int sizeV);** / calcule le gagnant a partir des declarations,clé publique du candidats et les clés des citoyens et retourne la clé publique du gagnant (sizeC et sizeV tailles des tables de hachages)

Partie IV : Implémentation d'un mécanisme de consensus

Dans la partie prècedente le système de vote était centralisé. Nous allons maintenant décentralisé le système de vote grace a la techologie de la blockchain qui est une base de données décentralisée et sécurisée. Chaque personne du réseau contient une copie de la base. Dans notre projet les données sont les déclarations de vote signées et les personnes du réseau les citoyens.

Voici le protocole de vote :

- Vote : un citoyen transmet sa déclaration de vote signée sur le réseau entre les horraires du scrutin
- Création du bloc : les citoyens volontaires (assesseurs) récuperent et vérifient les déclarations de vote récuperer en derniers sur le réseau. Représente un bloc contenant les déclarations valide et envoie celui-ci sur le reseau
- Mise a jour des données : lorsqu'un nouveau bloc est mis sur le réseau, le citoyen ajoute se bloc a sa base de données (chaine de blocs)

Le mécanisme de proof of work est un mécanisme permettant une fraude très difficile puisque la valeur est haché par une fonction cryptographique asymetrique très difficile a inverser. La mise en place d'une structure en blockchain ou chaine de bloc contenant des emplacements dans les blocs de valeurs hachés de bloc précédent et du bloc actuelle depuis les données du bloc permet éviter des fraudes et créer des blocs valides. La règle a suivre sur la techologique de la blockchain est la confiance de validité accordé a la plus longue chaine de blocs possible. Cela permet de supprimer les possibles blocs modifiés pour une tentative de fraude.

Exercice 7 – Structure d'un block et persistance

Dans cette exercice, nous allons développer le système de gestion des blocs de la blockchain.

Fichier : **block.h**

```
#ifndef BLOCK_H
#define BLOCK_H

#include <openssl/sha.h>
#include "manipulation_cle.h"
#include "signature.h"
#include "import_data.h"
#include "primalite.h"
#include "generate.h"
#include "generate_data.h"
#include "hash.h"

typedef struct block{

Key* author; //clé de l'auteur du bloc
CellProtected* votes; //contient les votes des citoyens
unsigned char* hash; //valeur haché du bloc (identifiant unique d'un bloc)
unsigned char* previous_hash; //valeur haché du bloc précédent
```

```
int nonce; // valeur unique (de communication) indiquant la preuve de calcul pour trouver la valeur haché du bloc
}Block;

void write_block(Block* b,char *filename);//ecrit un bloc dans un fichier *filename*

Block* read_block(char* filename);//lis un fichier *filename* contenant un bloc et retourne ce bloc

int length_block(Block* b);//retourne la taille totale des données des blocs sous la forme d'une chaine de caractere (pour block_to_str)

char* block_to_str(Block* block);//retourne une chaine de caractere contenant tous les informations du bloc (utilisé pour creer la valeur haché du bloc)

unsigned char* crypt_SHA256(char* str);//retourne une valeur haché de la chaine de caractere donnée en entrée grace a la technologie SHA256

void compute_proof_of_work(Block *B, int d);//realise le calcul de la preuve du travail (cela consiste a obtenir une valeur haché avec d zeros sucessives au debut)

int verify_block(Block* b,int d);//permet de verifier que le bloc est bien valide (verifie que le bloc a une valeur haché correct avec d zeros successives)

void delete_block(Block* b);//supprime la structure bloc en gardant la clé de l'auteur et les données des declarations signées(supprime seulement les cellprotected et garde les protected)

void delete_full_block(Block* b);//supprimer un bloc en entier (y compris la clé de l'auteur et les données des declarations)

#endif
```

Fichier: **block.c**

```
#include "block.h"

void write_block(Block* b,char *filename){

if(b == NULL){
printf("Erreur bloc null (write_block)\n");
return;
}

FILE* f = fopen(filename,"w");

if(f == NULL){
printf("Erreur ouverture fichier (write_block)\n");
return;
}

char* author_str = key_to_str(b->author);
```

```
CellProtected* votes = NULL;
CellProtected* tete_votes = NULL;//permet de garder l'en-tete de la liste de declarations
Protected* pr = NULL;
CellProtected* tmp = b->votes;
char* pr_str = NULL;
char hash_value[256];
char previous_hash_value[256];

//boucle evitant l'inversion de l'ecriture des declarations dans le fichier (evite d'avoir un bloc avec des declarations inverser en ordre lors de la lecture)
while(tmp != NULL){

pr = init_protected(tmp->data->pKey,tmp->data->mess,tmp->data->sgn);

votes = inserer_tete_protected(votes,pr);

tmp = tmp->next;
}

//recupere le 1er element de la liste de protected(pour la suppression a la fin)
tete_votes = votes;

//valeur haché par default si aucune valeur
if(b->hash == NULL){
b->hash = crypt_SHA256("0"); //cette valeur peut etre changé plus tard mais necessaire pour eviter des erreurs
}

//hv et phv pointeur char* temporaire necessaire pour sprintf() (se referer a la documentation sprintf())
char* hv = hash_value;
char* phv = previous_hash_value;

// ecriture des valeurs hachés en hexadecimal dans des chaines de caracteres (pour l'ecriture dans un fichier)
for(int i = 0;i < SHA256_DIGEST_LENGTH;i++){
hv += sprintf(hv,"%02x",b->hash[i]);
phv += sprintf(phv,"%02x",b->previous_hash[i]);
}

//ecriture du bloc dans le fichier
fprintf(f,"%s %s %s %d\n",author_str,hash_value,previous_hash_value,b->nonce);

while(votes != NULL){

pr_str = protected_to_str(votes->data);

fprintf(f,"%s\n",pr_str);
free(pr_str);

votes = votes->next;
```

```
}

delete_list_protected(tete_votes);
free(author_str);
fclose(f);
}

Block* read_block(char* filename){

FILE* f = fopen(filename,"r");

if(f == NULL){
printf("Erreur ouverture fichier (write_block)\n");
return NULL;
}

char* buffer = (char*)malloc(256*sizeof(char));//buffer pour recuperer les lignes du fichier via fgets (eviter les problemes de type buffer overflow avec les fscanf)

if(buffer == NULL){
printf("Erreur allocation buffer (read_block)\n");
return NULL;
}

Block* b = (Block*)malloc(sizeof(Block));

if(b == NULL){
printf("Erreur allocation block (read_block)\n");
free(buffer);
return NULL;
}

//recupere la 1er ligne du fichier contenant la clé de l'auteur les valeurs haché et le nonce et reconstruit le bloc (les declarations sont recuperer plus tard via un autre fgets)
if(fgets(buffer,256,f) != NULL){

char* author_str = (char*)malloc(256*sizeof(char));

if(author_str == NULL){
printf("Erreur allocation author_str (read_block)\n");
free(buffer);
free(b);
return NULL;
}

unsigned char hash[SHA256_DIGEST_LENGTH];
unsigned char previous_hash[SHA256_DIGEST_LENGTH];
char* hash_value = (char*)malloc(256*sizeof(char));//cette chaine recuperer les valeurs haché en hexadecimal

if(hash_value == NULL){
```

```
printf("Erreur allocation author_str (read_block)\n");
free(buffer);
free(b);
free(author_str);
return NULL;
}

char* previous_hash_value = (char*)malloc(256*sizeof(char));//cette chaine recuperer les valeurs haché en hexadecimal

if(previous_hash_value == NULL){
printf("Erreur allocation author_str (read_block)\n");
free(buffer);
free(b);
free(author_str);
free(hash_value);
return NULL;
}

int nonce = 0;
CellProtected* votes = NULL;

//on stock chaque elements dans leur chaine respectif pour les traiter si necessaire (comme les clé publique etc)
if(sscanf(buffer,"%s %s %s %d",author_str,hash_value,previous_hash_value,&nonce) == 4 ){

char* hv = hash_value;
char* phv = previous_hash_value;

//passage de hexadecimal a des caracteres non signés des valeurs haché (hexa -> unsigned char)
for(int i = 0;i < SHA256_DIGEST_LENGTH;i++){
sscanf(hv,"%02hhx",&hash[i]);
sscanf(phv,"%02hhx",&previous_hash[i]);
hv += 2; //+2 car chaque valeur en hexa sous 2 caracteres : %02x dans le fichier
phv += 2;
}

b->author = str_to_key(author_str);
b->hash = hash;
b->previous_hash = previous_hash;
b->nonce = nonce;

//on recupere les declarations de votes du fichier
while(fgets(buffer,256,f) != NULL){
votes = inserer_tete_protected(votes,str_to_protected(buffer));
}

b->votes = votes;

//on libere les chaines de caracteres temporaires
free(author_str);
```

```
free(hash_value);
free(previous_hash_value);
free(buffer);
fclose(f);

return b;

//si le format du fichier est incorrect, manque d'un element on non conforme
}else{

printf("Erreur probleme de format fichier(read_block_file)\n");
free(author_str);
free(hash_value);
free(previous_hash_value);
free(buffer);
fclose(f);
return NULL;
}

//probleme fichier vide
}else{
printf("Erreur fichier vide(read_block_file)\n");
free(buffer);
fclose(f);
return NULL;
}
}

int length_block(Block* b){

int size = 0;
char* author_str = key_to_str(b->author);
char* pr_str = NULL;
CellProtected* tmp = b->votes;

size = strlen(author_str)+strlen(b->previous_hash)+(b->nonce)%10+10;

while(tmp!=NULL){

pr_str = protected_to_str(tmp->data);
size += strlen(pr_str)+5;
free(pr_str);
tmp = tmp->next;
}

free(author_str);
return size;
}

char* block_to_str(Block* b){
```

```
int size_str = length_block(b);//recupere la taille du bloc sous forme de chaine de caractere (pour allocation de la chaine de caractere)
char* author_str = key_to_str(b->author);
char* pr_str = NULL;
char* block_str = (char*)malloc(size_str*sizeof(char));

if(block_str == NULL){
printf("Erreur d'allocation block_str(block_to_str)\n");
return NULL;
}

char* previous_hash_value = (char*)malloc(256*sizeof(char));

if(previous_hash_value == NULL){
printf("Erreur d'allocation previous_hash_value(block_to_str)\n");
free(block_str);
return NULL;
}

char* phv = previous_hash_value;

//recupere la valeur haché sous forme hexadecimal %02x
for(int i = 0;i < SHA256_DIGEST_LENGTH;i++){
phv += sprintf(phv,"%02x",b->previous_hash[i]);
}

sprintf(block_str,"%s%s%d",author_str,previous_hash_value,b->nonce);

CellProtected* tmp = b->votes;

while(tmp!=NULL){

pr_str = protected_to_str(tmp->data);
strncat(block_str,pr_str,strlen(pr_str)+5);//concatenation des declarations a la chaine de caractere finale
free(pr_str);

tmp = tmp->next;
}

free(author_str);
free(previous_hash_value);

return block_str;
}

unsigned char* crypt_SHA256(char* str){

unsigned char* d = SHA256(str,strlen(str),0);

return d;
```

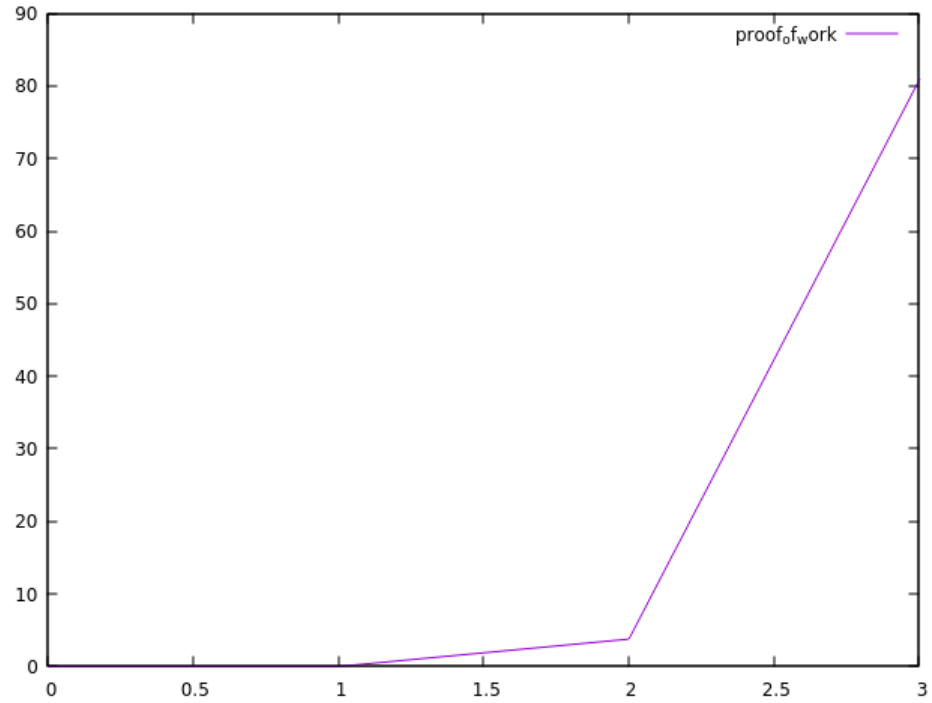
```
}

void compute_proof_of_work(Block *B, int d){

B->nonce = 0;
unsigned char* hash_value = NULL;
char* block_str = NULL;
int proof = 0;//variable temporaire permettant de verfier la validité du bloc

while(!proof){

B->nonce++;
block_str = block_to_str(B);//recupere la chaine de caractere representant le bloc
hash_value = crypt_SHA256(block_str);
free(block_str);

proof = 1;

for(int i=0;i<d;i++){

//verification de la validité du bloc (d zeros sucessives)
if(hash_value[i] != 0){

proof = 0;
break;
}
}

//affichage de la valeur haché
/*
for(int i = 0;i < SHA256_DIGEST_LENGTH;i++)
printf("%02x",hash_value[i]);
putchar('\n');
printf("Nonce: %d\n",B->nonce);
*/
}

B->hash = hash_value;
}

int verify_block(Block* b,int d){

for(int i=0;i<d;i++)
if(b->hash[i] != 0)
return 0;
return 1;
}

void delete_block(Block* b){

if(b->votes == NULL){
```

```
free(b);
return;
}

CellProtected* tmp = b->votes;
CellProtected* tmp2 = NULL;

//supprime seulement les cellules de cellProtected(les données protected ne sont pas supprimer des cellules!)
while(tmp != NULL){

tmp2 = tmp->next;
free(tmp);
tmp = tmp2;
}
free(b);
}

void delete_full_block(Block* b){

free(b->author);
delete_list_protected(b->votes);
free(b);
}
```

Description des fonctions principales de manipulation block.c :

- **void write_block(Block* b,char *filename);** / écrit le bloc dans un fichier
- **Block* read_block(char* filename);** / lecture du bloc dans un fichier
- **int length_block(Block* b); /** retourne la taille en chaine de caractere du bloc
- **char* block_to_str(Block* block); /** retourne la chaine de caractere contenant les données du bloc (clé de l'auteur, déclarations de votes, hash précedent et le nonce)
- **unsigned char* crypt_SHA256(char* str); /**chiffre la chaine de caractere avec la technologie SHA-256
- **void compute_proof_of_work(Block *B, int d); /** calcule la valeur haché avec d zeros successives des informations du bloc et change la valeur de nonce (calcule de la preuve de travail)
- **int verify_block(Block* b,int d); /** verifie si le bloc est bien valide avec la valeur haché contenant d zeros successives
- **void delete_block(Block* b); /** supprime le bloc et les cellules de déclaration sans supprimer les données (supprime pas la clé de l'auteur ni les protected)
- **void delete_full_block(Block* b); /** supprime le bloc complétement (y compris la clé de l'auteur et les données protected)

Q7.8 :



On constate que la fonction a un temps moyens de moins de 10s pour une valeur de d inférieur à 2 et lorsque d est supérieur a 2 , on a une augmentation en exponentielle du temps de calcule de la valeur haché (malheureusement la temps d'attente est beaucoup trop long pour les valeurs de d supérieurs a 3 on peut essayer de tracer graphiquement la courbe et obtenir le temps d'attente des valeurs d égale à 4 voir plus)

Partie V : Manipulation d'une base décentralisée de déclarations

Exercice 8 – Structure arborescente

Dans cette exercice, nous allons developpés les fonctions permettant de gérer une structure arborescente de nœuds contenant des blocs (BlockChain). La règle a suivre est de faire confiance a la plus longue chaine de nœud de l'arbre en partant de la racine.

Fichier: **block_tree.h**

```
#ifndef BLOCK_TREE_H
#define BLOCK_TREE_H

#include "block.h"
#include "import_data.h"
#include <stdio.h>
```

```
typedef struct block_tree_cell{ //noeud d'un arbre de blocs

Block* block;//bloc du noeud
struct block_tree_cell* father; //noeud père (null si racine)
struct block_tree_cell* firstChild;//noeud 1er fils
struct block_tree_cell* nextBro; //noeud frere
int height;//hauteur du noeud

}CellTree;

CellTree* create_node(Block* b);//creer un noeud et initialise le noeud

int update_height(CellTree* father, CellTree* child);//met a jour la hauteur de father si child a changé de hauteur

void add_child(CellTree* father, CellTree* child);//ajoute un noeud fils au père et met a jour les hauteurs ascendants

void print_tree(CellTree* tree);//affiche les noeuds de l'arbre

void delete_node(CellTree* node);//supprime un noeud

void delete_tree(CellTree* tree);//supprime l'arbre

CellTree* highest_child(CellTree* cell);//retourne le fils avec la hauteur la plus grande du noeud cell

CellTree* last_node(CellTree* tree);//retourne le dernier noeud de la plus grande chaine de noeuds de l'arbre

CellProtected* fusion_declarations(CellProtected* l1, CellProtected* l2);//fusionne deux listes de declarations signées

CellProtected* fusion_highest_chain(CellTree* tree);//fusionne tous les declarations signées de la plus longue chaine de noeuds de l'arbre

#endif
```

Fichier: **block_tree.c**

```
#include "block_tree.h"

CellTree* create_node(Block* b){

CellTree* t = (CellTree*)malloc(sizeof(CellTree));

t->block = b;
t->father = NULL;
t->firstChild = NULL;
t->nextBro = NULL;
t->height = 0;
```

```
return t;
}

int max(int n1,int n2){

if(n1>n2)
return n1;
else
return n2;
}

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

int h = max(father->height,child->height+1);

//si la hauteur a changé
if(h != father->height){

father->height = h;
return 1;
}

return 0;
}

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


if(father->firstChild == NULL){

father->firstChild = child;
child->father = father;

}else{

CellTree* t = father->firstChild;

while(t->nextBro != NULL){

t = t->nextBro;
}

t->nextBro = child;
child->father = father;
}

//actualisation des hauteurs des noeuds
while(father != NULL){
```

```
update_height(father,child);
child = father;
father = father->father;
}
}

void print_tree(CellTree* tree){

if(tree==NULL){return;}

printf("hauteur : %d id: ",tree->height);

for(int i = 0;i < SHA256_DIGEST_LENGTH;i++)
printf("%02x",tree->block->hash[i]);

putchar('\n');

print_tree(tree->firstChild);
print_tree(tree->nextBro);
}

void delete_node(CellTree* node){

if(node == NULL){return;}

delete_block(node->block);
free(node);
node = NULL;
}

void delete_tree(CellTree* tree){

if(tree == NULL){return;}

delete_tree(tree->nextBro);
delete_tree(tree->firstChild);

delete_node(tree);
}

CellTree* highest_child(CellTree* cell){

CellTree* t = cell->firstChild->nextBro;
CellTree* hc = cell->firstChild;

while(t != NULL){

//comparaison des hauteurs parmi tous les freres
if(t->height > hc->height){
```

```
hc = t;
}

t = t->nextBro;
}
return hc;
}

CellTree* last_node(CellTree* tree){

CellTree* t = tree;

while(t->firstChild != NULL){

t = highest_child(t);
}

return t;
}

CellProtected* fusion_declarations(CellProtected* l1, CellProtected* l2){

CellProtected* tete = l1;//recupere l'en-tete de la 1er liste

while(l1->next != NULL){
l1 = l1->next;
}
l1->next = l2; //ajout de la seconde liste a la premiere

return tete;
}

CellProtected* fusion_highest_chain(CellTree* tree){

CellProtected* cp = tree->block->votes;
CellTree* t = highest_child(tree);

//recupere le noeud le plus haut et fusionne sa liste avec les autres noeuds hauts
while(t->firstChild != NULL){

cp = fusion_declarations(cp,t->block->votes);
t = highest_child(t);
}

cp = fusion_declarations(cp,t->block-> votes);//fusionne la derniere liste du dernier
noeud

return cp;
}
```

Description des fonctions principales de manipulation block_tree.c :

- **CellTree* create_node(Block* b);** / écrit le bloc dans un fichier
- **int update_height(CellTree* father, CellTree* child);** / met a jour la hauteur du père en prenant le max (père,child+1)
- **void add_child(CellTree* father, CellTree* child);** /ajoute un fils au père
- **void print_tree(CellTree* tree);** / afficher tous les nœuds de l'arbre
- **void delete_node(CellTree* node);** / supprime un nœud de l'arbre
- **void delete_tree(CellTree* tree);** / supprime tous les nœuds de l'arbre
- **CellTree* highest_child(CellTree* cell);** / retourne le fils ayant la plus grande hauteur
- **CellTree* last_node(CellTree* tree);** / retourne le dernier nœud de l'arbre
- **CellProtected* fusion_declarations(CellProtected* l1, CellProtected* l2);** / fusionne deux listes de déclarations signées
- **CellProtected* fusion_highest_chain(CellTree* tree);** / fusionne la chaine la plus longue de nœuds de l'arbre

Q8.8: la complexité de la fonction est en O(n) avec n la taille de la 1er liste de déclarations. Pour avoir une fusion en O(1) il faut implementer dans la structure un pointeur permettant acceder aux dernières élements de la liste directement

Exercice 9 – Simulation du processus de vote

Durant cet exercice, on simulera un processus de vote numérique lié a la technologie blockchain. Un répertoire Blockchain qui contiendra la chaine de blocs qu'un citoyen a construite en local à partir des blocs venant du réseau. Un fichier Pending_block contenant les blocs en attente d'ajout sur la blockchain final et un fichier Pending_votes.txt contenant les votes a ajouté dans un bloc.

Les fonctions qu'on implémentera dans cette partie permettrons de gérer une chaine de blocs et l'ajout de nouveau bloc contenant les nouvelles déclarations de votes.

Fichier: **processus_vote.h**

```
#ifndef PROCESSUS_VOTE_H
#define PROCESSUS_VOTE_H

#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include "manipulation_cle.h"
#include "signature.h"
#include "import_data.h"
```

```
#include "hash.h"
#include "block.h"
#include "block_tree.h"

void submit_vote(Protected* p);//permet a un citoyen de soumettre un vote, ajoute son vote a la fin du fichier "Pending_votes.txt"

void create_block(CellTree* tree, Key* author, int d);//creer un bloc valide contenant les votes en attentes depuis le fichier "Pending_votes.txt" et ecrit se bloc dans le fichier "Pending_block" ("Pending_votes.txt supprimé après la creation du bloc)

void add_block(int d, char* name);//verifie que le bloc du fichier Pending_block est valide et ajoute le bloc dans le repertoire Blockchain sous le nom *name* (Pending_block" supprimé après l'appel de la fonction)

CellTree* read_tree();//lis les fichiers dans le repertoire Blockchain et construit l'arbre correspondant

Key* compute_winner_BT(CellTree* tree, CellKey* candidates,CellKey* votes,int sizeC,int sizeV);//retourne la clé publique du vainqueur de l'election en realisant le calcule a partir de la long chaine de noeuds de l'arbre et des listes de clés des candidats et voteurs (sizeC et sizeV les tailles des tables de hachage)

#endif
```

Fichier: **processus_vote.c**

```
#include "processus_vote.h"

void submit_vote(Protected* p){

FILE* f = fopen("Pending_votes.txt","a");
char* pr_str = protected_to_str(p);

fprintf(f,"%s\n",pr_str);

free(pr_str);
fclose(f);
}

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

Block* b = (Block*)malloc(sizeof(Block));

if(b == NULL){
return;
}

b->author = author;
b->votes = read_protected("Pending_votes.txt");
b->hash = NULL;
```

```
if(remove("Pending_votes.txt") != 0){

printf("Erreur suppression du fichier Pending_votes.txt(create_block)\n");
delete_list_protected(b->votes);
free(b);
return;
}

CellTree* lnode = last_node(tree);

if(lnode == NULL){

b->previous_hash = crypt_SHA256("0"); //valeur haché du bloc de la racine (probleme Seg Fault
avec previous_hash NULL)
}
else{
b->previous_hash = lnode->block->hash;
}

compute_proof_of_work(b,d);

write_block(b,"Pending_block");

libere_contenue_cellProtected_list(b->votes);

delete_block(b);

}

void add_block(int d, char* name){

Block* b = read_block("Pending_block");

if(verify_block(b,d) == 1){

char directory[256] = "./Blockchain/";

if(strlen(directory)+strlen(name)>256){

printf("Erreur nom de fichier en entree trop long(add_block)");
free(b->author);
libere_contenue_cellProtected_list(b->votes);
delete_block(b);
return;
}

strcat(directory,name);
strcat(directory,".txt");

write_block(b,directory);
```

```
}

remove("Pending_block");

libere_contenue_cellProtected_list(b->votes);
free(b->author);
delete_block(b);
}

CellTree* read_tree(){

CellTree** T = (CellTree**)malloc(256*sizeof(CellTree*));
CellTree* noeud = NULL;
Block* b = NULL;
int size = 0;

DIR* rep = opendir("./Blockchain/");

if(rep != NULL){

struct dirent* dir;

while(dir = readdir(rep)){

if(strcmp(dir->d_name,".")!=0 && strcmp(dir->d_name,"..")!=0){

printf("Chemin du fichier : ./Blockchain/%s \n",dir->d_name);

char directory_file[256] = "./Blockchain/";
strcat(directory_file,dir->d_name);

b = read_block(directory_file);
noeud = create_node(b);
size++;
T[size-1] = noeud;
}
}

closedir(rep);

for(int i=0;i<size;i++)
for(int j=0;j<size;j++)
if(strcmp(T[i]->block->hash,T[j]->block->previous_hash) == 0)
add_child(T[i],T[j]);


for(int k=0;k<size;k++)
if(T[k]->father == NULL)
noeud = T[k];

return noeud;
```

```
}
return NULL;
}

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

CellProtected* declarations = NULL;
   CellTree* child = NULL;
   Key* winner = NULL;

   if(tree != NULL){
      declarations = tree->block->votes;
      child = highest_child(tree);
   }

   while(tree != NULL && child != NULL){

      declarations = fusion_declarations(declarations, child->block->votes);
      tree = child;
      child = highest_child(tree);
   }

   verify_declarations(declarations);

winner = compute_winner(declarations, candidates, voters, sizeC, sizeV);

delete_list_protected(declarations);

   return winner;
}
```

Description des fonctions principales de manipulation simulation_vote.c :

- **void submit_vote(Protected* p);** / permet de soumettre un vote (ajoute son vote a la fin du fichier Pending_votes.txt ou créer le fichier Pending_votes.txt pour insertion du vote)
- **void create_block(CellTree* tree, Key* author, int d);** / création d'un bloc valide avec les déclarations de votes en attente dans Pending_votes.txt et ajoute ce bloc dans le fichier de bloc en attente Pending_block
- **void add_block(int d, char* name);** / ajoute le bloc contenu dans le fichier Pending_block dans le répertoire Blockchain avec comme nom de fichier name si le bloc est valide
- **CellTree* read_tree();** /récupere les blocs contenus dans le répertoire Blockchain et construit l'arbre correspondant
- **Key* compute_winner_BT(CellTree* tree, CellKey* candidates,CellKey* votes,int sizeC,int sizeV);** / Calcule et retourne la clé publique du gagnant de l'élection depuis

un arbre de bloc, les clés des citoyens et des candidates (sizeC et size V taille des tables de hachages)

Voici le fichier de test utilisé pour les fonctions précédentes. (Le fichier comporte quelques problèmes qui n'a pas pu être réglé malheureusement).

Fichier: **simulation.c**

```
#include "processus_vote.h"
#include "primalite.h"
#include "generate.h"
#include "manipulation_cle.h"
#include "signature.h"
#include "generate_data.h"
#include "import_data.h"
#include "hash.h"
#include "block.h"
#include "block_tree.h"
#include "processus_vote.h"

#define NB_CITOYENS 1000
#define NB_CANDIDATS 5

int main(){

generate_random_data(NB_CITOYENS,NB_CANDIDATS);

CellProtected* decl = read_protected("declarations.txt");
CellKey* candidates = read_public_keys("candidates.txt");
CellKey* voters = read_public_keys("keys.txt");

CellTree* tree = NULL;
int nb_votes = 0;

CellProtected* tmp = decl;

while(tmp != NULL){

submit_vote(tmp->data);
nb_votes++;

if(nb_votes == 10){

create_block(tree,tmp->data->pKey,1);
char* key_str = key_to_str(tmp->data->pKey);
add_block(1,key_str);
free(key_str);
nb_votes = 0;
}
```

```
tmp = tmp->next;
}

tree = read_tree();
print_tree(tree);

Key* k = compute_winner_BT(tree,candidates,voters,NB_CANDIDATS,NB_CITOYENS);

char* winner_str = key_to_str(k);

printf("Winner: %s\n",winner_str);

delete_list_keys(voters);
delete_list_keys(candidates);
delete_list_protected(decl);
delete_tree(tree);

return 0;
}
```

Q9.7 : L'utilisation d'une blockchain dans le cadre d'un processus de vote comporte de nombreuses avantage et permet d'éviter les problèmes de fraude puisque le système de consensus est basé sur un système de non-confiance et tout le monde participe pour vérifier que les blocs et les déclarations sont bien valides ceux qui empeche une tentative de fraude d'un groupe malveillant. Malgrè tout cela comporte un risque, on ne sait pas si le vote était un vote engagé ou non et que la personne n'a pas été forcée à le faire. La faille principale de ce système d'élection basé sur la blockchain est l'attaque par l'ingénierie sociale qui est une attaque directement liée à l'humain et non au système. Le consensus consistant à faire confiance a la plus longue chaine à une limite car un groupe comportant un grand nombre de personnes collaborant entre eux peuvent essayer de faire passer un bloc frauduleux. Le consensus se base sur la puissance de calcul et si un groupe malveillant avait l'avantage de la puissance de calcule alors le système serait détruit. Mais des solutions sont developpées pour palier a ces problèmes.

Fichier : **main_test.c**

```
#include "primalite.h"
#include "generate.h"
#include "manipulation_cle.h"
#include "signature.h"
#include "generate_data.h"
#include "import_data.h"
#include "hash.h"
#include "block.h"
#include "block_tree.h"
```

```
#include "processus_vote.h"

int main(void){

srand(time(NULL));

printf("is_prime_native : %d\n",is_prime_naive(5504));
printf("is_prime_native : %d\n",is_prime_naive(5431));

   printf("modpow_naive : %ld\n",modpow_naive(2,7,5));

   printf("modpow : %d\n",modpow(2,7,5));

   printf("rand_long : %ld\n",rand_long(3,10000));

   long p = random_prime_number(2,7,5000);
   long q = random_prime_number(2,7,5000);
   long n;
   long s;
   long u;

   printf("p:%ld q:%ld\n",p,q);

   generate_key_values(p,q,&n,&s,&u);

   printf("p:%ld q:%ld n:%ld s:%ld u:%ld\n",p,q,n,s,u);

   char* test = "test";

   long* tab = encrypt(test,s,n);


   for(int z=0;z<strlen(test);z++){
         printf("tab[%d] = %ld\n",z,tab[z]);
   }


char* final = decrypt(tab,strlen(test),u,n);

   printf("decrypt: %s\n",final);

   free(tab);
   free(final);

   Key* pKey = (Key*)malloc(sizeof(Key));
   Key* sKey = (Key*)malloc(sizeof(Key));
```

```
    init_pair_keys(pKey,sKey,1,7);

    printf("pKey: (%ld,%ld) sKey: (%ld,%ld)\n",pKey->val,pKey->n,sKey->val,sKey->n);

    char* key_pKey = key_to_str(pKey);
    char* key_sKey = key_to_str(sKey);

    printf("pKey: %s sKey: %s\n",key_pKey,key_sKey);

    Key* pKey1 = str_to_key(key_pKey);
    Key* sKey1 = str_to_key(key_sKey);

    printf("pKey: (%ld,%ld) sKey: (%ld,%ld)\n",pKey1->val,pKey1->n,sKey1->val,sKey1->n);

    free(pKey);
    free(sKey);
    free(key_pKey);
    free(key_sKey);
    free(pKey1);
    free(sKey1);

    generate_random_data(12,6);

    CellKey* pKeys = read_public_keys("keys.txt");
    CellKey* pKeys_c = read_public_keys("candidates.txt");

    printf("clé public citoyen:\n");
    print_list_keys(pKeys);
    printf("clé public candidates:\n");
print_list_keys(pKeys_c);

delete_list_keys(pKeys);
delete_list_keys(pKeys_c);

printf("declarations:\n");
CellProtected* cp = read_protected("declarations.txt");
print_list_protected(cp);

printf("verification:\n");
cp = verify_declarations(cp);
print_list_protected(cp);
delete_list_protected(cp);

pKeys = read_public_keys("keys.txt");

HashTable* t = create_hashtable(pKeys,20);
delete_hashtable(t);
```

```
delete_list_keys(pKeys);

CellProtected* decl = read_protected("declarations.txt");
CellKey* candidates = read_public_keys("candidates.txt");
CellKey* voters = read_public_keys("keys.txt");

Key* k = compute_winner(decl,candidates,voters,25,25);

char* winner_key = key_to_str(k);

printf("Winner: %s\n",winner_key);

free(k);
free(winner_key);

delete_list_keys(voters);
delete_list_keys(candidates);
delete_list_protected(decl);

decl = read_protected("declarations.txt");

Block* b = (Block*)malloc(sizeof(Block));
b->author = (Key*)malloc(sizeof(Key));
b->author->val = 545486;
b->author->n = 5806;
b->votes = decl;
b->hash = crypt_SHA256("previous");
b->previous_hash = crypt_SHA256("test");
b->nonce = 0;

write_block(b,"block#1_test.txt");

delete_full_block(b);

Block* b2 = read_block("block#1_test.txt");
write_block(b2,"block#2_test.txt");

char* test_block = block_to_str(b2);

printf("test_block:\n %s\n",test_block);

free(test_block);

delete_full_block(b2);

test = "Rosetta code";
```

```
unsigned char* str_test = crypt_SHA256(test);

printf("crypt:\n");

for(int i = 0;i < SHA256_DIGEST_LENGTH;i++)
printf("%02x",str_test[i]);
putchar('\n');

decl = read_protected("declarations.txt");

Block* b3 = (Block*)malloc(sizeof(Block));
b3->author = (Key*)malloc(sizeof(Key));
b3->author->val = 545486;
b3->author->n = 5806;
b3->votes = decl;
b3->hash = NULL;
b3->previous_hash = crypt_SHA256("958948646");
b3->nonce = 0;

compute_proof_of_work(b3,1);

printf("final hash value exit: \n");

for(int i = 0;i < SHA256_DIGEST_LENGTH;i++)
printf("%02x",b3->hash[i]);
putchar('\n');

printf("Nonce exit: %d \n",b3->nonce);

printf("verify block: %d\n",verify_block(b3,2));

printf("verify block: %d\n",verify_block(b3,7));

delete_full_block(b3);

CellProtected* votes_blocks1 = read_protected("declarations_test.txt");
CellProtected* votes_blocks2 = read_protected("declarations_test.txt");
CellProtected* votes_blocks3 = read_protected("declarations_test.txt");
CellProtected* votes_blocks4 = read_protected("declarations_test.txt");
CellProtected* votes_blocks5 = read_protected("declarations_test.txt");

Block* b4 = (Block*)malloc(sizeof(Block));
b4->author = (Key*)malloc(sizeof(Key));
b4->author->val = 545486;
b4->author->n = 5806;
b4->votes = votes_blocks1;
b4->hash = NULL;
```

```
b4->previous_hash = crypt_SHA256("0");
b4->nonce = 0;

compute_proof_of_work(b4,1);

Block* b5 = (Block*)malloc(sizeof(Block));
b5->author = (Key*)malloc(sizeof(Key));
b5->author->val = 8786476;
b5->author->n = 97476;
b5->votes = votes_blocks2;
b5->hash = NULL;
b5->previous_hash = b4->hash;
b5->nonce = 0;

compute_proof_of_work(b5,1);

Block* b6 = (Block*)malloc(sizeof(Block));
b6->author = (Key*)malloc(sizeof(Key));
b6->author->val = 9631845;
b6->author->n = 863322457;
b6->votes = votes_blocks3;
b6->hash = NULL;
b6->previous_hash = b5->hash;
b6->nonce = 0;

compute_proof_of_work(b6,1);

Block* b7 = (Block*)malloc(sizeof(Block));
b7->author = (Key*)malloc(sizeof(Key));
b7->author->val = 864256;
b7->author->n = 8652145;
b7->votes = votes_blocks4;
b7->hash = NULL;
b7->previous_hash = b5->hash;
b7->nonce = 0;

compute_proof_of_work(b7,1);

Block* b8 = (Block*)malloc(sizeof(Block));
b8->author = (Key*)malloc(sizeof(Key));
b8->author->val = 456329;
b8->author->n = 19428;
b8->votes = votes_blocks5;
b8->hash = NULL;
b8->previous_hash = b7->hash;
b8->nonce = 0;
```

```
compute_proof_of_work(b8,1);

CellTree* racine = create_node(b4);
CellTree* fils1 = create_node(b5);
CellTree* fils2 = create_node(b6);
CellTree* fils3 = create_node(b7);
CellTree* fils4 = create_node(b8);

add_child(racine,fils1);
add_child(fils1,fils2);
add_child(fils1,fils3);
add_child(fils3,fils4);

printf("racine hauteur: %d\n",racine->height);
printf("fils1 hauteur: %d\n",fils1->height);
printf("fils2 hauteur: %d\n",fils2->height);
printf("fils3 hauteur: %d\n",fils3->height);
printf("fils4 hauteur: %d\n",fils4->height);

print_tree(racine);

CellTree* ln = last_node(racine);

printf("ln : %p\n",ln);
printf("%p\n",fils4);

//printf("declaration fusion print: \n");
//CellProtected* fdecl = fusion_highest_chain(racine);
//print_list_protected(fdecl);

free(b4->author);
free(b5->author);
free(b6->author);
free(b7->author);
free(b8->author);

libere_contenue_cellProtected_list(b4->votes);
libere_contenue_cellProtected_list(b5->votes);
libere_contenue_cellProtected_list(b6->votes);
libere_contenue_cellProtected_list(b7->votes);
libere_contenue_cellProtected_list(b8->votes);

delete_tree(racine);

CellProtected* sub = read_protected("declarations_test.txt");

CellProtected* tmp = sub;
```

```
while(tmp != NULL){
submit_vote(tmp->data);
tmp = tmp->next;
}

delete_list_protected(sub);

CellTree* tree = NULL;
Key* author = (Key*)malloc(sizeof(Key));
author->val = 28511;
author-> n = 6541546;

create_block(tree,author,1);

add_block(1,"test_block");

//CellTree* tr = read_tree();

//print_tree(tr);

candidates = read_public_keys("candidates.txt");
voters = read_public_keys("keys.txt");

Key* winner = compute_winner_BT(tree,candidates,voters,20,20);

char* key_winner = key_to_str(winner);

printf("gagnant election test : %s\n",key_winner);

delete_tree(tree);
free(author);
//delete_tree(tr);
delete_list_keys(voters);
delete_list_keys(candidates);
free(key_winner);

return 0;
}
```

Fichier test personnel permettant de tester toutes les fonctions de tous les fichiers de l'exercice 1 à l'exercice 9.