Matteo Chianale

Eliott Rakotondramanitra

LU2IN006

Sujet : projet d'organisation d'un processus électoral par scrutin uninominal majoritaire « à deux tours »

Le projet se découpe en plusieurs parties :

L'objectif est d'implémenter des protocoles et des structures de données qui vont permettre de garantir l'intégrité, la sécurité et la transparence de l'élection.

Nous avons pour chaque exercice un fichier.c et un fichier.h ainsi qu'un fichier test qui lui correspond.

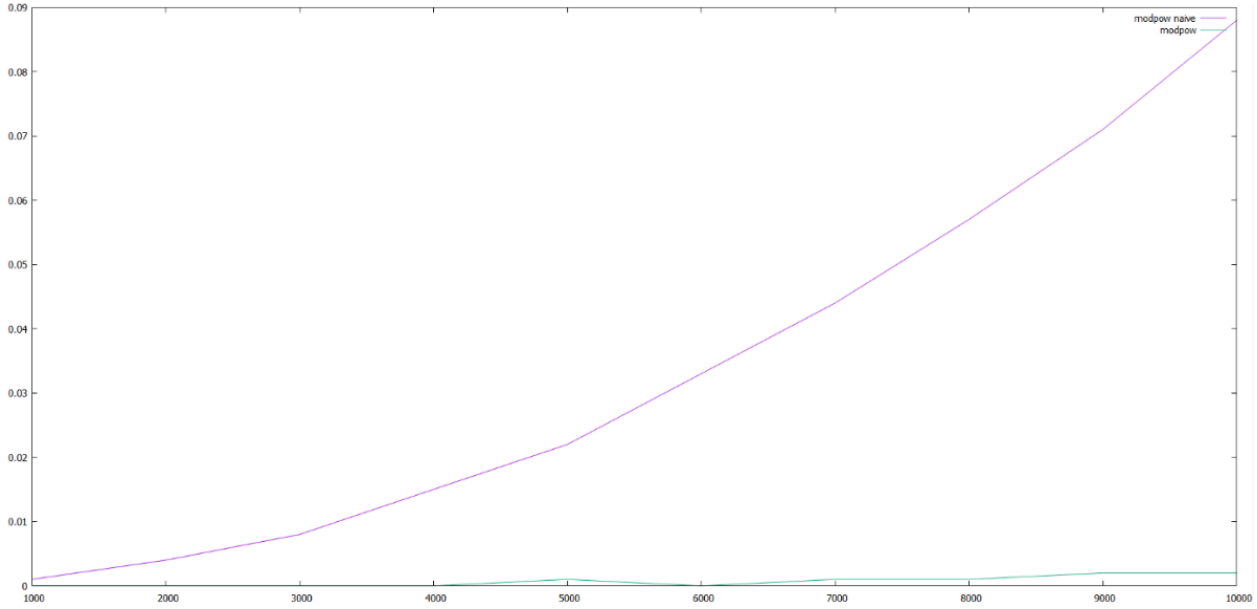
N.B : les réponses aux questions sont signalées en rouge.

Réponses aux questions de l'exercice 1 :

Question.1.2 : Quel est le plus grand nombre premier que vous arrivez à tester en moins de 2 millièmes de seconde avec cette fonction ? Environ : 99377, 99623, 99527,

Question.1.3 : Il y a m boucles avec une multiplication par itération donc la complexité de modpow_naive = 0(m).

Question.1.5 :



Graphe crée par la fonction *AnalyseModPow(132589), voir analyse_modpow.pdf*

On constate que modpow(134589, j, k) est stable par rapport à l'axe des abscisses où j croît vers 10000 et k est un entier entre 1 et 10.

À l'inverse modpow_naive(134589, j, k) est croissante et tend vers plus infini quand j prend des grandes valeurs.

Donc en termes de performances, modpow est nettement plus efficace que sa version naïve.

Question.1.7 :

Pour un nombre p non premier quelconque il y a le 3/4 des valeurs entre 2 et p-1 qui sont des témoins de Miller pour p. Alors nous avons une probabilité de 1/4 que l'algorithme soit faux et renvoie que p est premier pour un seul test. Soit k le nombre de tests, alors comme les tests sont indépendants entre eux, la borne sup de la probabilité d'erreur de l'algorithme est (1/4)^k

PARTIE 1 Implémentation d'outils de cryptographie

Exercice 1 : problème de primalité

L'objectif final de l'exercice est de pouvoir générer de grands nombres premiers.

Exercice1.h :

```
#ifndef EXERCICE1
#define EXERCICE1
int is_prime_naive(long);
long modpow_naive(long, long, long);
long modpow(long, long, long);
long mod_test(long, long, long);
void AnalyseModPow(int);
int witness(long, long, long, long);
long rand_long(long, long);
int is_prime_miller(long, int);
long ipow(long , long);
long random_prime_number(int,int,int);
#endif
```

- La fonction principale du fichier est **random_prime_number** qui à partir d'une taille minimum, d'une taille maximum et de k répétitions de l'expérience de Miller va retourner un nombre premier compris entre les tailles min et max. (on fait une boucle while tant que le nombre n'est pas candidat au test de Miller).
- La fonction **modpow** est la version optimale pour calculer une exponentiation modulaire
- **is_prime_miller** nous permet à l'aide de modpow de vérifier si un nombre impair est premier.

Explication random_prime_number :

```
long random_prime_number(int low_size, int up_size, int k) {
    if ((low_size == 0)&&(up_size == 0)) { return -1; } /* up = 1 et low = 1 => rand_long(1, 1) tjr egal à 1 => boucle infinie */
    assert(low_size <= up_size);
    assert(low_size <= 31);
    assert(up_size <= 31);
    long low =  ipow(2, low_size-1); // plus petit entier a low_size bits, 2^(low_size-1)

    long up = ipow(2, up_size)-1; // plus grand entier a up_size bits : 2^(up_size)-1*/
    int b = 0;
    long res;
    while(b == 0) {
        res = rand_long(low, up);
        b = is_prime_miller(res, k);
    }
    return res;
}
```

- Une fois les assert passés on commence par calculer les deux entiers de tailles respectivement low_size bits et up_size bits qui seront les bornes de notre intervalle. Pour cela on utilise une fonction intermédiaire **ipow** qui calcule 2 à la puissance low_size ou up_size.

- Puis on effectue une boucle dans laquelle on appelle rand_long qui retourne un entier long généré aléatoirement entre low et up inclus on utilise un témoin b qui vérifie si le nombre est bien premier.

Nous avons créé un fichier test1.c pour pouvoir vérifier la fonctionnalité de notre programme :

```c
    int low_size;
    int up_size;
    int k;
    for(int i = 0; i < 100; i++) {
        up_size = rand()%32; /* doit etre en 0 et 30, si up_size = 31 => up = 2^32 - 1, trop grand pour long (ne marche pas avce 32 et 31)*
        low_size = rand()%(up_size+1); /* low_size <= up_size */
        k = rand()%10 + 1;
        printf("random_prime_number(%d,%d,%d) = %ld\n", low_size, up_size, k, random_prime_number(low_size, up_size, k));
    }
    return 0;
}
```

```
random_prime_number(9,13,3) = 4363
low_size : 4, up_size : 26
random_prime_number(4,26,6) = 13709
low_size : 4, up_size : 6
random_prime_number(4,6,7) = 41
low_size : 16, up_size : 27
random_prime_number(16,27,9) = 38651
low_size : 5, up_size : 11
random_prime_number(5,11,9) = 347
low_size : 2, up_size : 5
random_prime_number(2,5,4) = 17
low_size : 10, up_size : 22
random_prime_number(10,22,5) = 5647
low_size : 10, up_size : 22
random_prime_number(10,22,7) = 28753
low_size : 7, up_size : 25
random_prime_number(7,25,8) = 10993
low_size : 2, up_size : 24
random_prime_number(2,24,9) = 23269
low_size : 16, up_size : 17
random_prime_number(16,17,3) = 40949
low_size : 8, up_size : 17
random_prime_number(8,17,4) = 28283
low_size : 8, up_size : 22
random_prime_number(8,22,10) = 6451
low_size : 11, up_size : 13
random_prime_number(11,13,1) = 5479
low_size : 4, up_size : 4
random_prime_number(4,4,10) = 13
low_size : 3, up_size : 7
random_prime_number(3,7,9) = 71
low_size : 19, up_size : 24
```

Terminal pour test1.c

Exercice 2 : Implémentation du protocole RSA

Pour pouvoir envoyer des données confidentielles avec le protocole RSA, il faut tout d'abord générer deux clés : une clé publique permettant de chiffrer des messages et une clé secrète pour pouvoir les déchiffrer. Nous allons nous aider de la fonction **random_prime_number** de l'exercice 1.

Exercice2.h :

```
#ifndef EXERCICE2
#define EXERCICE2
long extended_gcd(long, long, long *, long *);
void assert_generate_key_values(long, long, long , long , long );
void generate_key_values(long , long , long*, long *, long *);
long* encrypt(char * , long , long );
char * decrypt(long * , int , long , long);
void print_long_vector(long *, int);
#endif
```

La fonction principale est generate_key_values qui permet de générer la clé publique pkey = (s, n) et la clé secrète skey = (u, n), à partir des nombres premiers p et q, en suivant le protocole RSA.

Protocole :

1. Calculer $n = p \times q$ et $t = (p-1) \times (q-1)$.
2. Générer aléatoirement des entiers s inférieur à t jusqu'à en trouver un tel que $\text{PGCD}(s,t) = 1$.
3. Déterminer u tel que $s \times u \bmod t = 1$.

Le couple $pKey = (s,n)$ constitue alors la clé publique, tandis que le couple $sKey = (u,n)$ forme la clé secrète. Par définition, u est l'inverse de s modulo t (c'est ce qui permettra le déchiffrement).

Pour déterminer rapidement la valeur $\text{PGCD}(s,t)$ et l'entier u vérifiant $s \times u \bmod t = 1$, on peut utiliser l'algorithme d'Euclide étendu. En effet, étant deux nombres entiers s et t, cet algorithme calcule la valeur $\text{PGCD}(s,t)$ et détermine les entiers u et v vérifiant l'équation de Bezout :

$$s \times u + t \times v = \text{PGCD}(s,t) \tag{1}$$

```
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;
    long gcd = 0;
    while(gcd != 1) {
        *s = rand_long(1, t);
        gcd = extended_gcd(*s, t, u, &v);
    }
    if ( *u < 0) {
        *u = *u + t;
    }
    return;
}
```

Nous pouvons désormais avec la clé publique ainsi que la clé privée chiffrer et déchiffrer des messages.

- **Chiffrement :** on chiffre le message m en calculant $c = m^s \bmod n$ (c est la représentation chiffrée de m).
- **Déchiffrement :** on déchiffre c pour retrouver m en calculant $m = c^u \bmod n$.

Fonction **encrypt** et **decrypt :**

```c
long* encrypt(char * chaine, long s, long n) {
    long * tab = (long*)malloc(strlen(chaine)*sizeof(long));
    assert(tab);
    int i = 0;
    int m;
    printf("conversion des carateres de la chaine en int : \n");
    while(chaine[i] != '\0') {
        m = (int)chaine[i];
        printf("%c : %d\n" , chaine[i], m);
        tab[i]= modpow(m, s, n);
        i++;
    }
    return tab;
}

/* Q.2.3 */
char * decrypt(long * crypted, int size, long u, long n) {
    char * chaine  = (char*)malloc(size+1);
    assert(chaine);
    long m;
    for(int i = 0; i < size; i++) {
        m = modpow(crypted[i], u, n);
        chaine[i] = (char)((int)m);
        printf("%c %ld\n" , chaine[i], m);
    }
    chaine[size] = '\0';
    return chaine;
}
```

Pour encrypter on cast chaque caractère de la chaine en int et on lui applique la fonction modpow avec la clé publique pour chiffrer et on se retrouve avec un tableau de int correspondant au message

Pour décrypter on applique cette fois modpow mais avec la clé secrète sur les différents éléments du tableau puis on cast en char.

```
low_size : 3, up_size : 7
low_size : 3, up_size : 7
p : 79, q : 37
cle publique = (1517, 2923)
cle privee = (1901, 2923)
conversion des carateres de la chaine en int :
H : 72
e : 101
l : 108
l : 108
o : 111
Initial message Hello
Encoded representation :
Vector: [1078   1565    2199    2199    518     ]
H 72
e 101
l 108
l 108
o 111
Decoded Hello
```

Terminal pour main2.c

PARTIE 2 : Création d'un système de déclarations sécurises par chiffrement asymétrique

Exercice 3 Manipulations de structures sécurisées

un citoyen interagit pendant les élections en effectuant des déclarations, l'objectif de l'exercice est d'implémenter une structure qui puisse permettre la sécurité de la déclaration.

Exercice3.h :

```c
#ifndef EXERCICE3
#define EXERCICE3
//structure

typedef struct key{
    long x;
    long n;
} Key;
typedef struct protected{
    Key * pKey;
    char * mess;
    Signature * sgn;
}Protected;

typedef struct signature{
    long * tab;
    int taille;
}Signature;
```

Nous avons 3 structures :

- La structure Key correspond à la structure d'une clé publique ou secrète.
- La structure de signature permet d'attester de l'authenticité d'une déclaration, le tableau de long est généré à l'aide de la clé sécrète de l'émetteur et peut être déchiffré à l'aide de la clé publique de l'émetteur.
- la structure de protected est la version sécurisée de la déclaration, on y retrouve la clé publique du candidat, le message de la déclaration de vote, ainsi que la signature associée.

```c
void init_key(Key*, long, long);
//Q 3.3
void init_pair_keys(Key*, Key*, long, long);
//Q 3.4
char* key_to_str(Key*);
Key* str_to_key(char*);
//Q 3.6
Signature * init_signature(long *,int);
//Q 3.7
Signature* sign(char*, Key*);
//Q 3.8
char* signature_to_str(Signature *);
//Q3.10
Protected* init_protected(Key*, char*, Signature*);
//Q3.11
int verify(Protected*);
//Q3.12
char * protected_to_str(Protected*);
Protected* str_to_protected(char*);
#endif
```

Analysons les principales fonctions :

```
void init_pair_keys(Key* pKey, Key* sKey, long low_size, long up_size){
    long p = random_prime_number(low_size ,up_size,5000);
    long q = random_prime_number(low_size ,up_size,5000);
    while ( p == q ) {
        q = random_prime_number(low_size,up_size,5000) ;
    }
    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);
}
```

La fonction **init_pair_keys** permet d'initialiser la paire de clés d'une personne (publique et secrète).

On commence par générer deux grands nombres premiers, puis on utilise la fonction **generate_key_values** de l'exercice 2 et on initialise les structures de clés.

```
Signature * sign(char* mess, Key* sKey){
    long * tab = encrypt(mess,sKey->x,sKey->n);
    Signature * res = init_signature(tab,strlen(mess));
    return res;
}
```

La fonction **sign** crée la signature associée à la déclaration de vote.

Pour cela on encrypte le message à l'aide de la clé sécrète de l'émetteur puis on initialise la structure de signature.

```
int verify(Protected*pr){
    char * mess_sgn = decrypt(pr->sgn->tab,pr->sgn->taille,pr->pKey->x,pr->pKey->n);
    if(strcmp(mess_sgn,pr->mess) == 0){
        return 1;
    }
    return 0;
}
```

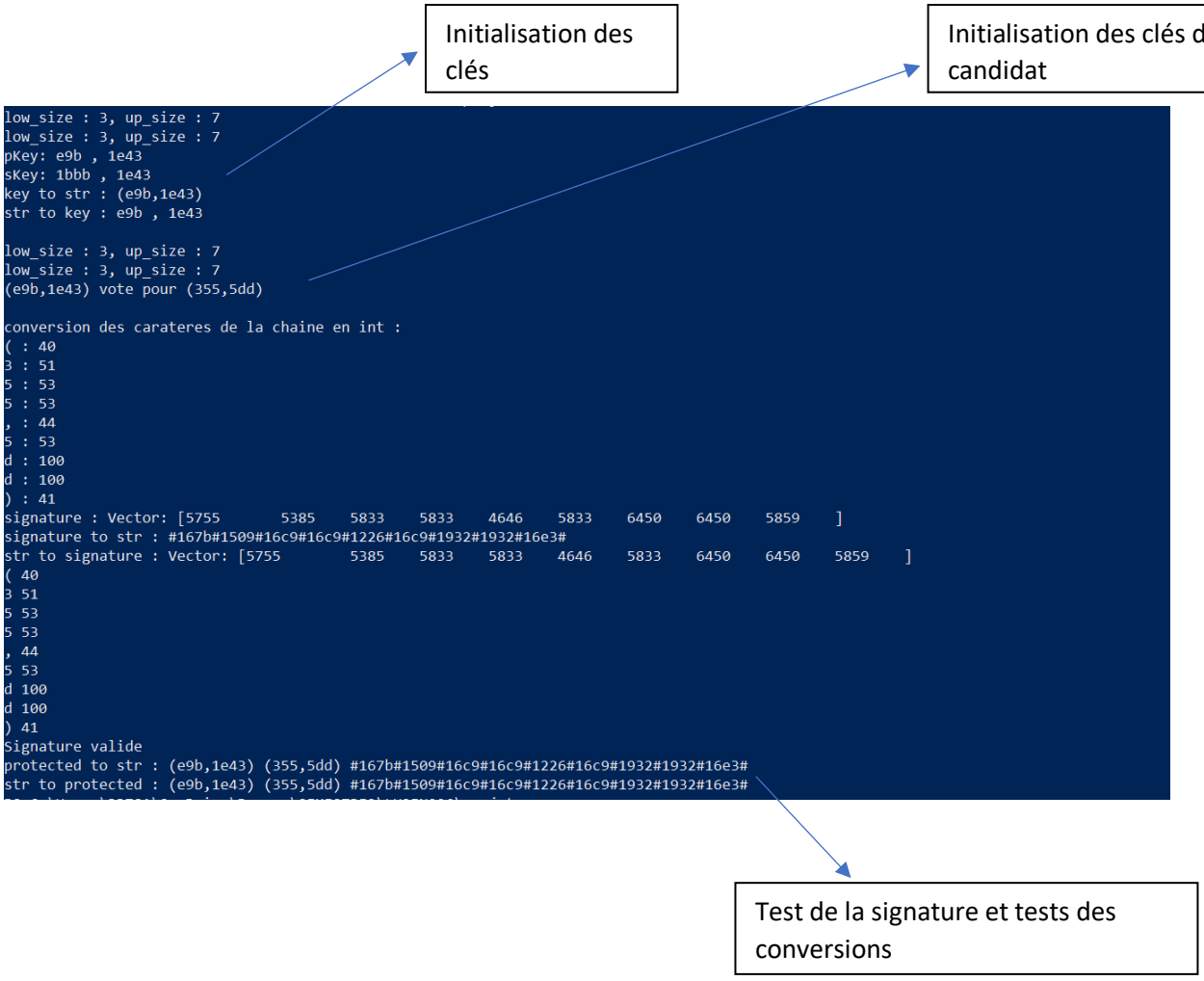
La fonction **verify** vérifie si la signature correspond bien à la déclaration de vote.

Pour cela on fait le processus inverse en décryptant le message à l'aide de la clé publique.

Dans le main on test :

- l'initialisation des clés
- les différentes conversions des clés
- les tests de signature
- tests de la structure protected

Terminal pour main3.c :



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

L'objectif de l'exercice est de simuler un processus de vote à l'aide de trois fichiers : un fichier contenant les clés de tous les citoyens, un fichier indiquant les candidats et un fichier contenant des déclarations signées.

Exercice4.h :

```
/* 4.1 */
int recherche_char(char **, char *, int);
void generate_random_data(int , int);
#endif
```

La fonction principale generate_random_data commence par générer nv couples de clés représentant les nv citoyens. Nous créons deux tableaux de char qui vont stocker les clés publiques et privés des citoyens et on écrit ces clés dans un fichier « keys.txt ».

```c
// citoyen
FILE * couple = fopen("keys.txt", "w");
assert(couple);

char * tabV[nv]; //cle prive des citoyens
char * tabVP[nv];//cle public des citoyens

for(int i = 0; i < nv; i++) {
    init_pair_keys(pKey, sKey, 3, 7);
    tabV[i] = strdup(key_to_str(sKey));
    tabVP[i] = strdup(key_to_str(pKey));
    fprintf(couple, "%s %s\n", key_to_str(pKey), key_to_str(sKey));
}
fclose(couple);
```

Puis on sélectionne nc clés publiques aléatoirement pour définir les nc candidats, pour cela on commence par créer un tableau de char dans lequel on mettra les clés publiques des candidats. Pour cela on fait une boucle qui à chaque tour va prendre une clé au hasard dans le tableau des clés publiques des citoyens.

Ensuite nous avons rajouté une fonction recherche_car qui va rechercher si dans le tableau des clés publiques candidates la clé est déjà existante afin d'éviter les doublons. Enfin on ajoute la clé dans le tableau candidat et on l'écrit dans un fichier **« candidats.txt ».**

```c
// candidat
FILE * candidat= fopen("candidates.txt", "w");
assert(candidat);

char * tabC[nc];//cle public des candidats
char * tmp;
int n = 0; // nbr d'element dans tabC
int r; // valeur aleatoire

for(int i = 0; i < nc; i++) {
    r = rand()%nv;
    tmp = strdup(tabVP[r]);
    while(recherche_char(tabC, tmp, n)==0) { // empeche les doublons de candidats
        r = rand()%nv;
        tmp = strdup(tabVP[r]);
    }
    tabC[i] = strdup(tmp);
    n++;
    fprintf(candidat, "%s\n",  tmp);
}
fclose(candidat);
```

```c
int recherche_char(char ** tab, char * elt, int n) {
    for(int i = 0; i < n; i++) {
    if(strcmp(tab[i], elt)==0) {
        return 0; // retourne 0 si la chaine existe deja
    }
    }
    return 1;   // retourne 1 si la chaine n'existe pas
}
```

Et enfin pour créer les déclarations on parcourt notre tableau des clés privées des citoyens et on crée une signature à partir d'une clé publique candidate prise au hasard dans le tableau candidat et la clé privée du citoyen. Les déclarations sont écrites dans le fichier **« déclarations.txt**

PARTIE 3 : Manipulation d'une base centralisée de déclarations

Dans cette partie, on considère un système de vote centralisée, dans lequel toutes les déclarations de vote sont envoyées au système de vote, qui a pour rôle de collecter tous les votes et d'annoncer le vainqueur de l'élection à tous les citoyens. les déclarations de vote sont enregistrées au fur et à mesure dans un fichier appelé declarations.txt, et une fois que le scrutin est clos, ces données sont chargées dans une liste chainée.

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

L'objectif de cet exercice est la lecture et le stockage de données sous forme de listes simplement chainées de clés (dans les fichiers « keys.txt » et « candidates.txt ») et de déclarations signées (dans « declaration.txt »).

Exercice5.h :

```c
#ifndef EXERCICE5
#define EXERCICE5
typedef struct cellKey {
    Key * data ;
    struct cellKey * next ;
} CellKey ;

typedef struct cellProtected {
    Protected * data ;
    struct cellProtected * next ;
    } CellProtected ;
//5.1
CellKey * create_cell_key(Key*);
//5.2
CellKey * insere_entete(CellKey*, Key*);
//5.3
CellKey * read_public_keys(char *);
//5.4
void print_list_keys(CellKey*);
//5.5
void delete_cell_key(CellKey*);
void delete_list_keys(CellKey*);
//5.6
CellProtected* create_cell_protected(Protected*);
//5.7
CellProtected* inserer_en_tetekp(CellProtected*, Protected*);
```

```c
//5.8
CellProtected* read_protected(char*);
//5.9
void print_list_protected(CellProtected *);
//5.10
void delete_cell_protected(CellProtected*);
void delete_list_protected(CellProtected*);
#endif
```

Les deux structures utilisées sont des listes chainées. La première est une liste chainée de clé et la deuxième de protected (structure définie dans l'exercice3 contenant la clé publique de l'émetteur, sa déclaration de vote ainsi que la signature associée).

Nous avons implémenté des fonctions de manipulation pour les deux listes permettant de créer une liste, insérer un élément, afficher la liste, supprimer la liste ainsi que créer une liste à partir d'un fichier.

Les fonctions principales de cet exercice sont **read_public_keys** et **read_protected.**

Regardons comment fonctionne read_protected :

```
CellProtected* read_protected(char* nomfile) {
    char ligne[256];
    char messL[256];
    char signL[256];
    char pKeyL[256];
    FILE * f = fopen(nomfile, "r");
    assert(f);
    //1)
    fgets(ligne, 256, f);
    sscanf(ligne, "%s %s %s", pKeyL, messL, signL); // Initialisation de la liste chainee
    //2)
    Key * pKey = malloc(sizeof(Key));
    pKey = str_to_key(pKeyL);
    Signature* sign = str_to_signature(signL);
    Protected* pr = init_protected(pKey, messL, sign);
    CellProtected* cp = create_cell_protected(pr);
    //3)
    while(fgets(ligne, 256, f) != NULL) {
        sscanf(ligne, "%s %s %s",  pKeyL, messL, signL);
        pKey = str_to_key(pKeyL);
        Signature* sign = str_to_signature(signL);
        Protected* pr = init_protected(pKey, messL, sign);
        cp = inserer_en_tetekp(cp, pr);
    }
    fclose(f);
    return cp;
}
```

La fonction lit des structures protected (clé publique de l'émetteur, clé du candidat, signature) qui sont écrites dans un fichier sous cette forme :

```
(55f,9bb) (1b5,30b) #8ed#697#686#4f8#58c#742#14a#686#6f7#
(6fd,17bd) (6fd,17bd) #d7c#dfc#66f#11c2#10bd#a8f#88f#35c#11c2#a62#
(feb,12e9) (23,5b) #a6f#1216#d9d#fe1#13a#c3#3bb#
(2b9,7a9) (2b9,7a9) #4e#617#554#4d4#5c5#3da#526#4d4#d7#
(257,1553) (e9,6af) #1472#ca6#d0f#3e6#12d8#14e0#66#677#
(2953,2a37) (2953,2a37) #1a7a#b2e#f1#1848#b4f#a80#b2e#26a1#b4f#58e#113c#
(e9,6af) (e9,6af) #200#202#39#53e#6aa#44c#578#29#
(231,289) (23,5b) #27a#d7#1eb#a5#14#8e#1e1#
(1b5,30b) (2b9,7a9) #2e1#ff#1bf#23a#212#e8#81#23a#290#
(d57,126d) (2b9,7a9) #57f#1114#47#ffc#1210#500#11db#ffc#11e#
```

1) On récupère une ligne du fichier correspondant à une protected et on scan la clé publique, le message ainsi que la signature
2) Puis à l'aide des fonctions de l'exercice 3 on initialise la structure protected puis on crée la liste chainée

3) Puis on réitère les étapes jusqu'à la fin du fichier en insérant les éléments en tête de liste

Exercice 6 : Détermination du gagnant de l´élection

Nous allons dans cet exercice à l'aide des données que nous avons déterminer le gagnant de l'élection.

Exercice6.h

```h
#ifndef EXERCICE6
#define EXERCICE6

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

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

//1
CellProtected *  delete_protected_novalide(CellProtected*);
//2
HashCell* create_hashcell(Key*);
//3
int hash_function(Key*, int);
//4
int find_position(HashTable*, Key*);
//5
HashTable* create_hashtable(CellKey*, int);
//6
void delete_hashtable(HashTable*);
//7
int compare(Key*, Key*);
int is_in_Hastable(HashTable*,Key *, int *);
void vote(HashTable* H, Key * key);
Key * compute_winner(CellProtected*, CellKey*,CellKey*, int, int);


#endif
```

Nous allons utiliser dans cet exercice 2 nouvelles structures. Une structure **Hashcell** qui contient une clé et une valeur ainsi qu'une table de hachage **Hashtable** de **Hashcell**. La table de hachage est une structure qui nous permettra de rechercher un élément de manière efficace (en O(1)).

delete_protected_novalide est une fonction qui vérifie dans une liste de **protected** si tous les éléments sont valides et les supprime sinon.

hash_function est notre fonction de hachage qui renvoie la position d'un élément dans notre table de hachage. Nous avons choisi :

```c
int hash_function(Key* key, int size){
    return (key->x + key->n)%size;
}
```

find_position recherche dans la table de hachage s'il existe un élément de clé key et retourne la position supposée sinon.

create_hashtable est la fonction qui va allouer une table de hachage à partir d'une liste de clés en sachant que les collisions sont résolues par probing linéaire (si la position supposée de notre élément est déjà prise on parcourt le tableau à partir de cette position jusqu'à trouver une position libre, ici NULL) :

résolution par probing linéaire si la case est déjà occupée

```c
while(keys != NULL){
    pos = hash_function(keys->data,t->size);
    if (t->tab[pos] != NULL) {
        for(int i = pos; i < t->size; i++) { //probing lineaire
            if ( t->tab[i % (t->size)] == NULL) {
                t->tab[i % (t->size)] = create_hashcell(keys->data);
                break;
            }
        }
    }
    else{
    t->tab[pos] = create_hashcell(keys->data);
    }
    keys = keys->next;
}
```

Puis nous avons la fonction principale **compute_winner** qui va faire appel à **vote** et **is_in_Hastable.** La fonction prend en argument une liste de déclaration de vote, la liste des candidats, la liste des personnes votantes ainsi que la taille des deux tables de hachage que l'on va créer, une pour les candidats et mettre à jour le nombre de votes et une pour les votants pour vérifier qu'une personne est bien sur la liste électorale et qu'il n'a pas déjà voté.

- On commence par vérifier les déclarations à l'aide de la fonction vue précédemment.
- Puis on utilise la fonction **is_in_Hastable** qui vérifie qu'une clé est bien dans la table renvoie 1 si c'est vérifié et 0 sinon (elle utilise une fonction **compare** qui vérifie si deux clés sont égales). Elle renvoie également par référence la valeur de la structure **HashCell** correspondante (pour la table des votants 1 si la personne a déjà voté et 0 sinon, et pour la table des candidats correspond au nombre de votes). La fonction nous permet de vérifier pour chaque déclaration si le votant est bien sur la liste électorale, qu'il n'a pas déjà voté et que la personne sur qui porte le vote est bien un candidat de l'élection.
- Si toutes les conditions sont satisfaites on appel la fonction **vote** qui incrémente la valeur de clé dans la table de hachage.
- Puis on détermine le gagnant dans la table des candidats ayant le plus de votes.

```c
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);
    decl = delete_protected_novalide(decl); // on supprime les mauvaises declarations

    int temp_val = 0;
    int temp_vote = 0;
    int max = 0;
    Key * vainqueur = NULL;
    while( decl != NULL){
        Key * voter = decl->data->pKey;
        Key * candidat = str_to_key(decl->data->mess);

        //On verfife que la personne est bien sur la liste electorale et On verifie que la personne vote bien pour ur
        if( is_in_Hastable(Hc, candidat, &temp_vote) == 1 && is_in_Hastable(Hv, voter, &temp_val) == 1 ) {
        //On verifie que la personne n'a pas deja vote
            if (temp_val == 0) {
                vote(Hc, candidat);
                vote(Hv, voter);

            if( temp_vote + 1 > max) {
                max = temp_vote + 1;
                vainqueur = candidat;
            }
        }
    }
        decl = decl->next;
```

A noté que la table de hachage prend tout son sens dans la fonction **is_in_Hastable** car complexité O(1) dans le meilleur des cas.

Nous testons le bon fonctionnement de la simulation dans le fichier test56.c

Aperçu :

```
liste citoyens :
(709,797)
(1e35,21a3)
(329,863)
(2dd,403)
(1e0b,2039)
liste candidats:
(2dd,403)
(1e0b,2039)
(329,863)
liste declarations (apres supression) :
(709,797) (2dd,403) #50f#3fa#574#574#181#584#76#780#c#
(1e35,21a3) (1e0b,2039) #b4d#17dd#170#1258#10ce#6c3#1b74#1258#9e9#ae8#5ee#
(329,863) (2dd,403) #2ef#80c#536#536#425#45d#2e7#16c#519#
(2dd,403) (329,863) #298#19#1df#386#29c#38#13a#19#301#
(1e0b,2039) (2dd,403) #a51#1efc#74f#74f#af7#a4b#75e#29b#d9#
vainqueur : (2dd,403)
```

PARTIE 4 : Implémentation d'un mécanisme de consensus et Manipulation d'une base décentralisée de déclarations.

Dans cette partie nous allons introduire la structure principale du projet, la Blockchain.

Cette structure permettra à chaque citoyen de vérifier par lui-même le résultat du vote et d'éviter les tentatives de fraudes.

Exercice 7 : Structure d'un block et persistance

Exercice7.h :

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

Voici la structure de bloc représentant la Blockchain :

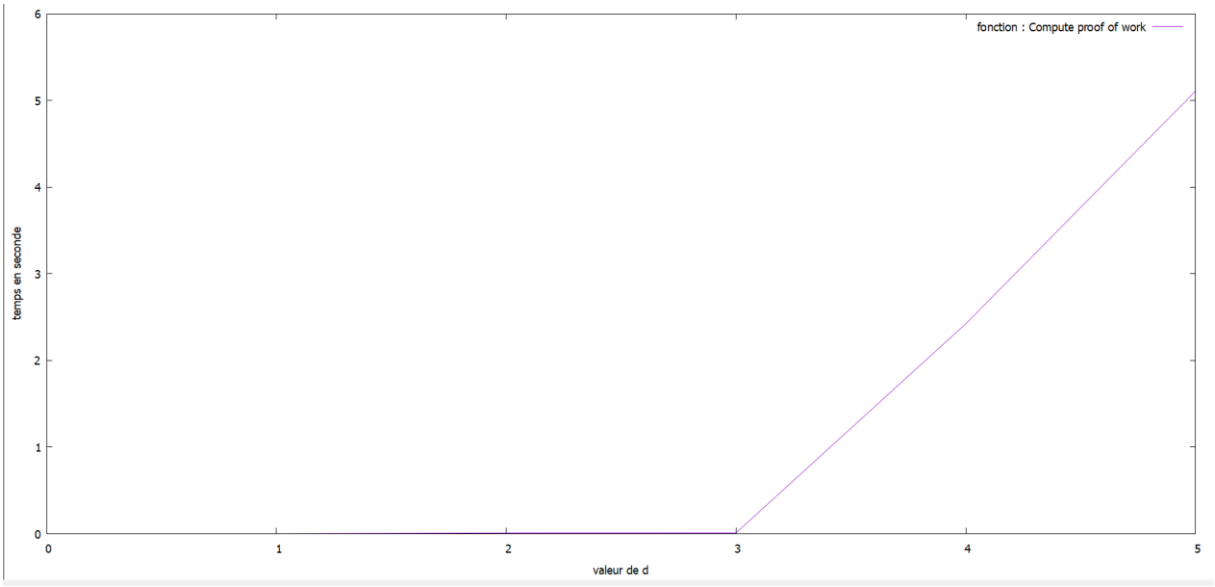
- **author** est la clé de l'auteur du bloc
- **votes** est la liste des déclarations de vote
- **hash** la valeur hachée du bloc
- **previous_hash** la valeur hachée du bloc précédent
- **nonce** est la preuve de travail

```c
//preliminaire
Block * new_block(int, Key *, char *, char *, CellProtected *);
//1
void write_block(char *, Block *);
//2
Block * read_block(char*);
//3
char* block_to_str(Block* block);
//5
unsigned char * str_to_hash(const char *);
//6
int nb_zero(unsigned char *);
void compute_proof_of_work(Block *, int);
//7
int verify_block(Block *, int);
//8
void AnalyseCompute(Block *, int);
//9
void delete_block(Block*);

#endif
```

Voici les autres fonctions de l'exercice :

- **new_block** initialise une structure Block
- **write_block** écrit un block dans un fichier
- **read_block** lit un block depuis un fichier
- **block_to_str** transforme un bloc en une chaine de caractères (c'est cette chaine que l'on hachera avec la fonction de hachage pour avoir la valeur hachée du block **hash**)
- **str_to_hash** est la fonction de hachage qui hache une chaine de caractères avec la fonction de hachage cryptographique SHA-256
- **compute_proof_of_work** est la fonction qui va permettre de rendre un bloc valide. Un bloc est valide si la valeur hachée du bloc commence par **d** zéros successifs. Pour cela on va initialiser la preuve de travail **nonce** à 0 puis nous incrémentons **nonce** jusqu'à trouver une valeur Hash du bloc commençant par **d** zéros, **nonce** permet de vérifier facilement que le créateur du bloc a fait beaucoup de calculs pour réaliser cette inversion partielle. La fonction fait appel à une fonction **nb_zero** qui compte le nombre de zéros dans une chaine de caractères.
- **verify_block** vérifie si un bloc est valide
- **AnalyseCompute** est une fonction qui étudie le temps moyen de la fonction **compute proof_of_work** selon la valeur de d. Nous exécutons cette fonction dans le fichier analyse_7.c



Graphe crée par la fonction *AnalyseCompute , voir analyse_compute.pdf*

Question 7.8 : On constate que lorsque d = 4 le temps mis par la fonction **compute_proof_work** dépasse 1 sec (2.5 sec environ), le temps nécessaire grandit bien exponentiellement avec le nombre de zéros requis (à noter que le programme crash pour une valeur de **d** supérieur à 7).

- **delete_block** supprime un bloc

Ces fonctions sont testées dans le fichier main7.c

Aperçu du terminal :

```
test de base : Rosetta code
764faf5c61ac315f1497f9dfa542713965b785e5cc2f707d6468d7d1124cdfcf
str_to_hash(Rosetta code) => 764faf5c61ac315f1497f9dfa542713965b785e5cc2f707d6468d7d1124cdfcf
str_to_hash(Previous code) => 9073b55e1abe06b288173a4cdd429004ebaeab7f8fbeebb685afbacf0f23beaa
block b : author | previous_hash | votes (declarations.txt) | nonce
(527,725) 9073b55e1abe06b288173a4cdd429004ebaeab7f8fbeebb685afbacf0f23beaa (527,725) (4ff,104b) #53c#41#19a#19a#2b#3d4#c5#41#6e1#4d5# (197,
7e5) (4ff,104b) #625#725#3ed#3ed#388#503#34f#725#2f8#156# (109,149) (d,23) #147#b1#e2#c5#10#125# (4ff,104b) (197,7e5) #bcc#3e3#6c5#b1b#d46#
b1b#29e#6cf#b39# 0
block Bdup (block lu pour Block.txt :
(527,725) 9073b55e1abe06b288173a4cdd429004ebaeab7f8fbeebb685afbacf0f23beaa (4ff,104b) (197,7e5) #bcc#3e3#6c5#b1b#d46#b1b#29e#6cf#b39# (109,
149) (d,23) #147#b1#e2#c5#10#125# (197,7e5) (4ff,104b) #625#725#3ed#3ed#388#503#34f#725#2f8#156# (527,725) (4ff,104b) #53c#41#19a#19a#2b#3d
4#c5#41#6e1#4d5# 0
compute_proof_of_work(b, 4):
nouvelle valeur de b->hash : 0000efa575bfa806ac419d3cda4dc334ed94f4f7a37268490ff93dcb173d05f4
nouvelle valeur de b->nonce : 186298
block valide pour d = 4
```

Exercice 8 : Structure arborescente

Chaque bloc possède la valeur hachée du bloc qui le précède ce qui forme une chaine. en cas de triche, on peut se retrouver avec plusieurs blocs indiquant le même bloc précédent, ce qui conduit à une structure arborescente.

Nous allons donc implanter dans cet exercice la dernière structure du projet un arbre de blocs.

Exercice8.h :

```c
#ifndef EXERCICE8
#define EXERCICE8
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;
```

Il s’agit d’un arbre de block avec :

- **block** les nœuds de l’arbre sont des blocs
- **father** le parent du nœud
- **firstChild** le premier fils du nœud
- **nextBro** le premier frère du nœud
- **height** la hauteur du nœud

```
//1
CellTree * create_node(Block *);
//2
int update_height(CellTree *, CellTree *);
//3
void add_child(CellTree *, CellTree *);
//4
void print_tree(CellTree *);
//5
void delete_node(CellTree *);
void delete_tree(CellTree *);
```

Nous retrouvons également des fonctions de manipulations de l'arbre :

- **create_node** qui crée et initialise un nœud de hauteur 0
- **update_height** qui met à jour la hauteur du père lorsque l'un des fils à été modifié
- **add_child** qui ajoute un fils à un nœud
- **print_tree** qui affiche l'arbre
- **delete_node** qui supprime un nœud
- **delete_tree** qui supprime un arbre

```
//6
CellTree * highest_child(CellTree *);
//7
CellTree * last_node(CellTree *);
```

Ces fonctions vont nous permettre de déterminer le dernier boc de l'arbre :

A partir d'un arbre de blocs, en cas de triche le consensus est de faire confiance à la chaîne de blocs la plus longue en partant de la racine de l'arbre.

- **highest_child** est une fonction qui renvoie le nœud fils avec la plus grande hauteur (comme la hauteur est directement renseignée dans la structure il suffit de parcourir tous les fils et de renvoyer celui avec la plus grande hauteur)
- **last_node** retourne le dernier bloc de cette plus longue chaine (on parcourt l'arbre récursivement à l'aide de highest_child)

Pour finir ces fonctions vont permettre d'extraire les déclarations de vote de la plus longue chaine de l'arbre pour déterminer le gagnant de l'élection :

```
CellProtected * fusion_liste(CellProtected *, CellProtected *);
//9
CellProtected * get_votes(CellTree *);
#endif
```

- **fusion_liste** permet de fusionner deux listes chainées de déclarations signées, il faut pour cela parcourir la première liste et faire pointer le pointeur **next** du dernier élément vers le premier élément de la deuxième liste.

Question 8.8 : Soit n la taille de la première liste, la complexité de la fonction est en O(n). En implémentant une liste doublement chainée nous aurions eu directement accès au dernier élément de la première liste, la complexité aurait été en O(1)).

- **get_votes** retourne la liste de déclaration obtenue en fusionnant les listes de la chaine la plus longue de l'arbre

Exercice 9 : Simulation du processus de vote

C'est la dernière partie du projet, dans cet exercice nous mettons en œuvre les fonctions et les structures vu précédemment.

Exercice9.h :

```
#ifndef EXERCICE9
#define EXERCICE9
//1
void submit_vote(Protected *);
//2
void create_block(CellTree *, Key *, int);
//3
void add_block(int , char *);
//4
CellTree * read_tree();
//5
Key * compute_winner_BT(CellTree * , CellKey *, CellKey *, int, int);
#endif
```

Pour cette partie finale on utilisera le répertoire **Blockchain** contenant les fichiers suivants :

- Pending_votes.txt
- Pending_block.txt

Cela permettra d'assurer un processus de vote en trois temps : Votes des citoyens, Créations de blocs à intervalles réguliers, Mise à jour de la base de données.

Vote et création de blocs valides :

- **submite_vote** ajoute le vote d'un citoyen dans Pending_votes.
- **create_block** crée le bloc obtenu à partir des votes dans Pending_votes et l'ajoute dans Pending_block.
- **add_block** qui vérifie si le bloc dans Pending_block est valide puis crée un fichier contenant le bloc et l'ajoute au répertoire **Blockchain.**

Lecture de l'arbre et calcul du gagnant :

- **read_tree** construit l'arbre correspondant aux blocs contenus dans le répertoire **Blockchain**.
- **compute_winner_BT** retourne le gagnant de l'élection à partir de l'arbre construit avec **read_tree.**

Nous testons la simulation finale du processus électoral dans le fichier **election.c** dans lequel nous simulons l'élection avec 1000 citoyens, 3 candidats et un bloc valide pour **d** = 3 (pour réduire le temps d'exécution).

Aperçu du terminal :

```
Noeud de hauteur : 2 et id : 0001fb23897959915220391d41f26f54a9b40269850042c1b5985e158a1df3d6
Noeud de hauteur : 1 et id : 0001ffaeab10840170db44d9013e67fc861fdb7495dd5483a48b753b2fd69fe1
Noeud de hauteur : 0 et id : 00081e434720ba262bbc1a19ff82d264e8de23b52bbcbcaefc452a0a7d9592ac
vainqueur : (56f,17bd)
```

Question 9.7 : Conclusion sur l'utilisation d'une Blockchain dans le cadre d'un processus de vote :

Les avantages :

- transparence des élections, chacun peut vérifier le décompte des voix et suivre l'avancement de l'élection
- l'anonymat
- décentralisation du système
- difficulté pour inverser la clé SHA256 (trouver une valeur avec **d** zéros successifs, la probabilité de trouver un nombre aléatoire qui vérifie la condition est 1/(2^d) pratiquement impossible pour **d** = 30 par exemple)

Les inconvénients :

- Le processus est très volumineux en calcul, par exemple dans notre cas pour une valeur **d** supérieur à 7 le programme plante (complexité exponentielle de la preuve de travail)
- Le système peut n'être plus en mesure de répondre en un temps raisonnable et finir par se mettre en indisponibilité si nous lui envoyons un trop grand nombre de requêtes
- ajouter des blocs avec des transactions frauduleuses et soumettre la chaîne ainsi construite dès qu'elle est plus longue que la chaîne valide construite selon le protocole, permet de contourner le consensus consistant à faire confiance à la plus longue chaîne.

Conclusion du Projet :

Le projet est intéressant car il nous a permis d'utiliser toutes les structures de données étudiées en cours et de l'appliquer à un problème concret d'actualité. Nous venons à la conclusion que l'approche d'une Blockchain pour le déroulement d'une élection est attirante par sa transparence et sa sécurité mais qu'il est loin d'être optimal (dans notre projet) par les problèmes évoqués précédemment.