# Projet : Blockchain appliquée à un processus éléctoral

Nous considérons dans ce projet l'organisation d'un processus électoral par scrutin uninominal majoritaire à deux tours (comme ici en France).

- Partie 1 : Implémentation d'outils de cryptographie.
- Partie 2 : Création d'un système de déclarations sécurisés par chiffrement asymétrique.
- Partie 3 : Manipulation d'une base centralisée de déclarations.
- Partie 4 : Implémentation d'un mécanisme de consensus.
- Partie 5 : Manipulation d'une base décentralisée de déclarations.

Chaque partie est divisée en exercices, qui vont vous permettre de concevoir progressivement le programme final. Il est impératif de travailler régulièrement afin de ne pas prendre de retard et de pouvoir profiter des séances correspondantes pour chaque partie du projet.

## Cadre du projet

Dans ce projet, nous considérons la problématique de désignation du vainqueur d'un processus électoral. Dans un processus électoral, chaque participant peut déclarer sa candidature au scrutin et/ou donner sa voix à un candidat déclaré. La tenue d'un processus électoral a, depuis toujours, posé des questions de confiance et de transparence épineuses, dans la mesure où les élections sont généralement organisées par le système exécutif en place, qui est souvent candidat à sa réélection et donc soupçonné d'interférences. De plus, le compte des voix fait appel à des assesseurs, ce qui en fait un travail long et avec peu de garanties de fiabilité, dans la mesure où tout le monde ne peut pas vérifier que le compte a eu lieu dans des conditions honnêtes. Enfin, le caractère anonyme de la désignation par bulletin fait que personne ne peut vérifier a posteriori que sa voix a été comptabilisée chez le bon candidat.

Un autre aspect à considérer est celui de l'ergonomie pour le votant. Plus précisément, un système décentralisé permettrait un vote à distance, et le vote à distance (tout comme le vote par correspondance) a longtemps été envisagé comme un outil pour combattre l'abstention, qui a atteint un record historique lors des des élections régionales et départementales de juin 2021 (66,7 % et allant jusqu'à 87 % chez les jeunes de moins de 25 ans). Un projet de loi a par ailleurs été déposé à l'assemblée nationale le 21 septembre 2021, qui permettrait le vote par correspondance. Ce projet de loi serait un premier pas vers le vote électronique, avec comme argument principal le fait que le vote postal a été instauré en Outre-Rhin en 1957, justifiant possiblement l'écart entre le taux d'abstention en France et celui de l'Allemagne (qui ne dépasse pas les 33%).

L'objectif de ce projet est donc de proposer une piste de réflexion sur les protocoles et sur les structures de données à mettre en place pour permettre d'implémenter efficacement le processus de désignation du vainqueur de l'élection, tout en garantissant l'intégrité, la sécurité et la transparence de l'élection.

# Développement d'outils cryptographiques

Dans cette partie, nous allons développer des fonctions permettant de chiffrer un message de façon asymétrique. La cryptographie asymétrique est une cryptographie qui fait intervenir deux clés :

- Une clé publique que l'on transmet à l'envoyeur et qui lui permet de chiffrer son message.
- Une clé secrète (ou privée) qui permet de déchiffrer les messages à la réception.

Par exemple, 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 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.

L'algorithme de cryptographie asymétrique que nous allons implémenté est le protocole RSA, très utilisé actuellement sur internet pour transmettre des données confidentielles (notamment dans le cadre du e-commerce). Ce protocole s'appuie sur des nombres premiers pour la génération des clés publiques et secrètes. Nous allons donc commencer par traiter le problème de la génération de nombres premiers.

## Exercice 1 – Résolution du problème de primalité

Pour générer efficacement des nombres premiers, il faut avoir un moyen d'effectuer rapidement des tests de primalité. Le problème de primalité se définit comme suit : étant donné un entier $p$ impair, $p$ est-il un nombre premier ?

### IMPLÉMENTATION PAR UNE MÉTHODE NAÏVE

Un méthode naïve consiste à énumérer tous les entiers entre 3 et $p-1$, et conclure que $p$ est premier si et seulement si aucun de ces entiers ne divise $p$.

**Q 1.1** Implémentez la fonction `int is_prime_naive(long p)` qui, étant donné un entier impair $p$, renvoie 1 si $p$ est premier et 0 sinon. Quelle est sa complexité en fonction de $p$ ?

**Q 1.2** Quel est le plus grand nombre premier que vous arrivez à tester en moins de 2 millièmes de seconde avec cette fonction ?

Pour parvenir à générer de très grands nombres premiers, nécessaires au bon fonctionnement du protocole RSA en pratique, nous allons implémenter un test de primalité plus efficace : le test de primalité de Miller-Rabin. Ce test probabiliste nécessite de pouvoir calculer efficacement une exponentiation modulaire, c'est-à-dire la valeur $a^m \bmod n$, étant donnés trois entiers $a$, $m$ et $n$. C'est l'objet des questions suivantes.

### EXPONENTIATION MODULAIRE RAPIDE

Pour calculer $a^m \bmod n$, sans passer par le calcul de la valeur $a^m$ (qui peut être très grande), une méthode naïve consiste à itérer les étapes suivantes : on multiplie la valeur courante par $a$ puis on applique le modulo $n$ sur le résultat, avant de passer à l'itération suivante. Il s'agit de répéter ces opérations $m$ fois.

**Q 1.3** Implémenter la fonction `long modpow_naive(long a, long m, long n)` qui prend en entrée trois entiers $a$, $m$ et $n$, et qui retourne la valeur $a^b$ mod $n$ par la méthode naïve. Quelle est sa complexité ?

**Q 1.4** Au lieu de multiplier par $a$ à chaque itération, on peut réaliser des élévations au carré (directement suivies de modulo) pour obtenir un algorithme de complexité logarithmique (c-à-d en $O(log_2(m))$). Donnez le code d'une fonction `int modpow(long a, long m, long n)` réalisant cette succession d'élévation au carré.

**Indication :** pour une version récursive, il suffit de remarquer que $a^b$ mod $n$ est égal à :
- 1 quand $m = 0$ (cas de base).
- $b * b$ mod $n$ avec $b = a^{m/2}$ mod $n$, quand $m$ est pair.
- $a * b * b$ mod $n$ avec $b = a^{\lfloor m/2 \rfloor}$ mod $n$, quand $m$ est impair.

Attention, à chaque appel, au plus un appel récursif est à effectuer.

**Q 1.5** Comparez les performances des deux méthodes d'exponentiation modulaire en traçant des courbes de temps en fonction de $m$. Qu'observez-vous ?

Pour implémenter le test de Miller-Rabin, nous utiliserons la fonction `modpow` ci-après.

## TEST DE MILLER-RABIN

Le test de primalité de Miller-Rabin est un algorithme randomisé qui utilise la propriété suivante. Soit $p$ un nombre impair quelconque. Soient $b$ et $d$ deux entiers tels que $p = 2^b d + 1$. Étant donné un entier $a$ strictement inférieur à $p$, on dit que $a$ est un témoin de Miller pour $p$ si :
- $a^d$ mod $p \neq 1$,
- et $a^{2^r d}$ mod $p \neq -1$ pour tout $r \in \{0, 1, \ldots, b-1\}$.

Si $a$ est un témoin de Miller pour $p$, alors il est possible de prouver que $p$ n'est pas premier. Malheureusement, dans le cas contraire, on ne peut pas dire que $p$ est premier. Par contre, si on répète ce test suffisamment de fois, pour des valeurs de $a$ tirées au hasard entre 1 et $p-1$, et qu'aucune de ces valeurs générées ne correspond à un témoin de Miller pour $p$, alors on peut dire que $p$ est très probablement premier.

Ci-dessous, vous trouverez le code des fonctions :
- `int witness(long a, long b, long d, long p)` qui teste si $a$ est un témoin de Miller pour $p$, pour un entier $a$ donné.
- `long rand_long(long low, long up)` qui retourne un entier `long` généré aléatoirement entre `low` et `up` inclus.
- `int is_prime_miller(long p, int k)` qui réalise le test de Miller-Rabin en générant $k$ valeurs de $a$ au hasard, et en testant si chaque valeur de $a$ est un témoin de Miller pour $p$. La fonction retourne 0 dès qu'un témoin de Miller est trouvé ($p$ n'est pas premier), et retourne 1 si aucun témoin de Miller n'a été trouvé ($p$ est très probablement premier).

```
1
2  int witness(long a, long b, long d, long p) {
3      long x = modpow(a,d,p);
4      if(x == 1){
5          return 0;
6      }
7      for(long i = 0; i< b; i++){
8          if(x == p-1){
```

```
9               return 0;
10          }
11          x = modpow(x,2,p);
12      }
13      return 1;
14  }
15
16  long rand_long(long low, long up){
17      return rand() % (up - low +1)+low;
18  }
19
20  int is_prime_miller(long p, int k) {
21      if (p == 2) {
22          return 1;
23      }
24      if (!(p & 1) || p <= 1) { //on verifie que p est impair et different de 1
25          return 0;
26      }
27      //on determine b et d :
28      long b = 0;
29      long d = p - 1;
30      while (!(d & 1)){ //tant que d n'est pas impair
31          d = d/2;
32          b=b+1;
33      }
34      // On genere k valeurs pour a, et on teste si c'est un temoin :
35      long a;
36      int i;
37      for(i = 0; i < k; i++){
38          a = rand_long(2, p-1);
39          if(witness(a,b,d,p)){
40              return 0;
41          }
42      }
43      return 1;
44  }
```

**Q 1.6** Intégrez ces fonctions dans votre programme.

On s'intéresse maintenant à la fiabilité du test de Miller-Rabin, autrement dit à sa probabilité d'erreur. L'algorithme fait une erreur quand il déclare qu'un entier $p$ est premier alors qu'il ne l'est pas. Cela se produit quand, pour un entier $p$ non premier, l'algorithme ne trouve pas de témoin de Miller pour $p$ parmi les $k$ valeurs de $a$ générées.

**Q 1.7** En utilisant le fait que, pour tout entier $p$ non premier quelconque, au moins $\frac{3}{4}$ des valeurs entre 2 et $p-1$ sont des témoins de Miller pour $p$, donner une borne supérieure sur la probabilité d'erreur de l'algorithme.

Comme la probabilité d'erreur de cet algorithme devient rapidement très faible quand $k$ augmente, et que sa complexité pire-cas est en $O(k(log_2(p))^3)$, alors il est plus intéressant d'utiliser cet algorithme pour effectuer des tests de primalité que la méthode naïve implémentée au tout début de ce projet. On utilisera donc le test de primalité de Miller-Rabin dans la suite du projet.

## GÉNÉRATION DE NOMBRES PREMIERS

On souhaite à présent utiliser le test de Miller-Rabin pour générer des nombres premiers de grande taille, où la taille d'un entier est donnée par son nombre de bits. L'idée est de générer aléatoirement

des entiers de la bonne taille (avec la fonction `rand_long` définie précédemment), jusqu'à en trouver un qui réussit le test de Miller-Rabin. Le théorème des nombres premiers assure que l'on trouve par cette méthode un nombre premier au bout d'un nombre d'essais raisonnable.

**Q 1.8** Écrire une fonction `long random_prime_number(int low_size, int up_size, int k)` qui étant donnés :

- deux entiers `low_size` et `up_size` représentant respectivement la taille minimale et maximale du nombre premier à générer,
- et un entier $k$ représentant le nombre de tests de Miller à réaliser,

retourne un nombre premier de taille comprise entre `low_size` et `up_size`.

**Rappel utile :** $2^{t-1}$ est le plus petit entier à $t$ bits, tandis que $2^t - 1$ est le plus grand.

**Remarque :** Dans le cadre de ce projet, on considérera uniquement des entiers avec au plus 7 bits. Pour pouvoir travailler avec des nombres plus grands, il faudrait arrêter d'utiliser des `long` et passer plutôt sur des nombres de 128 bits qu'on implémenterait avec une librairie de modélisation mathématique.

## Exercice 2 – Implémentation du protocole RSA

Le chiffrement RSA, nommé ainsi par les initiales de ses trois inventeurs (Rivest, Shamir et Adleman), est un algorithme de cryptographie asymétrique qui a été décrit en 1977 et breveté en 1983.

### GÉNÉRATION D'UNE PAIRE (CLÉ PUBLIQUE, CLÉ SECRÈTE)

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. Pour que les échanges soient sécurisés, le couple (clé secrète, clé publique) doit être engendré de manière à ce qu'il soit calculatoirement impossible de retrouver la clé secrète à partir de la clé publique. Le fonctionnement du protocole RSA est fondé sur la difficulté de factoriser de grands entiers. Plus précisément, pour générer un couple (clé secrète, clé publique), le protocole RSA a besoin de deux (grands) nombres premiers $p$ et $q$ distincts (générés aléatoirement), et réalise les opérations suivantes :

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 :

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

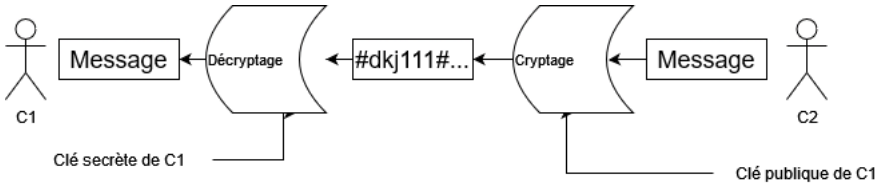
En particulier, quand $\text{PGCD}(s,t) = 1$, on a bien $s \times u \bmod t = 1$. Un version récursive de l'algorithme d'Euclide étendu vous est donné ci-dessous :

```c
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;
}
```

**Q 2.1** Implémentez la fonction `void generate_key_values(long p, long q, long* n, long *s, long *u)` 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.

## CHIFFREMENT ET DÉCHIFFREMENT DE MESSAGES

On s'intéresse maintenant à l'envoi de message. Supposons que la personne C2 souhaite envoyer un message à la personne C1 en utilisant le protocole RSA. Dans ce cas, la personne C2 utilise la clé publique de la personne C1 pour chiffrer le message avant son envoi. À sa réception, la personne C1 déchiffre le message à l'aide de sa clé secrète.




Soient $pKey = (s, n)$ et $sKey = (u, n)$ la clé publique et la clé secrète du destinataire, et $m$ le message à lui envoyer représenté par un entier (inférieur à $n$).

- **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 $m = c^u \bmod n$.

**Q 2.2** Implémentez une fonction `long* encrypt(char* chaine, long s, long n)` qui chiffre la chaîne de caractères `chaine` avec la clé publique $pKey = (s, n)$. Pour cela, la fonction convertit chaque caractère en un entier de type `int` (sauf le caractère spécial '\0'), et retourne le tableau de `long` obtenu en chiffrant ces entiers.

**Rappel :** il faut utiliser la fonction `modpow` pour réaliser une exponentiation modulaire efficace.

**Q 2.3** Implémentez une fonction `char* decrypt(long* crypted, int size, long u, long n)` qui déchiffre un message à l'aide de la clé secrète $skey = (u, n)$, en connaissant la taille du tableau d'entiers. Cette fonction renvoie la chaîne de caractères obtenue, sans oublier le caractère spécial '\0' à la fin.

## FONCTION DE TESTS

Pour vérifier le bon fonctionnement de votre programme, reproduisez le programme principal suivant et compilez le en utilisant les fonctions que vous avez implémentées dans les questions précédentes.

```c
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_keys_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
    }

    //Afichage 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);

    return 0;
}
```

# Déclarations sécurisées

Revenons à notre problème de vote. Dans ce problème, un citoyen interagit pendant les élections en effectuant des *déclarations*. En pratique, ces déclarations peuvent soit être des déclarations de candidature soit des déclarations de vote. Pour simplifier le projet, on va supposer que l'ensemble des candidats est déjà connu, et que les citoyens ont juste à soumettre des déclarations de vote.

## Exercice 3 – Manipulations de structures sécurisées

Dans notre modèle, chaque citoyen possède une carte électorale, qui est définie par un couple de clés :

- Une clé *secrète* (ou privée) qu'il utilise pour signer sa déclaration de vote. Cette clé ne doit être connue que par lui.
- Une clé *publique* permettant aux autres citoyens d'attester de l'authenticité de sa déclaration (vérification de la signature). Cette clé est aussi utilisée pour l'identifier dans une déclaration de vote, non seulement quand il vote, mais aussi quand quelqu'un souhaite voter en sa faveur.

Dans cet exercice, nous verrons que signer une déclaration se fait par chiffrement du contenu (avec la clé secrète), puis vérifier une signature se fait simplement par déchiffrement (avec la clé publique).

### MANIPULATION DE CLÉS

On rappelle que, dans le protocole RSA, la clé publique et la clé secrète d'un individu sont des couples d'entiers, notés respectivement $pKey = (s, n)$ et $sKey = (u, n)$ dans l'exercice précédent.

**Q 3.1** Définissez une structure `Key` qui contient deux `long` représentant une clé (publique ou secrète).

**Q 3.2** Écrivez une fonction `void init_key(Key* key, long val, long n)` permettant d'initialiser une clé déjà allouée.

**Q 3.3** Écrivez une fonction `void init_pair_keys(Key* pKey, Key* sKey, long low_size, long up_size)` qui utilise le protocole RSA pour initialiser une clé publique et une clé secrète (déjà allouées).

**Indication :** Cette fonction fait appel aux fonctions `random_prime_number`, `generate_keys_values` et `init_key`. Inspirez-vous des lignes 14 à 25 de la fonction `main` de l'exercice précédent.

**Q 3.4** Écrire deux fonctions `char* key_to_str(Key* key)` et `Key* str_to_key(char* str)` qui permettent de passer d'une variable de type `Key` à sa représentation sous forme de chaîne de caractères et inversement. La chaîne de caractères doit être de la forme `"(x,y)"`, où `x` et `y` sont les deux entiers de la clé exprimés en hexadécimal.

**Indication :** les fonctions `sprintf` et `sscanf`, et le spécificateur `%lx` peuvent vous être utiles.

**Remarque :** l'hexadécimal est un système très souvent utilisé en informatique, notamment parce qu'il permet une conversion sans aucun calcul avec le système binaire (système employé par nos ordinateurs), et qu'il possède l'avantage de rendre des entiers très grands plus compactes et plus lisibles.

### SIGNATURE

Dans notre contexte, une déclaration de vote consiste simplement à transmettre la clé publique du candidat sur qui porte le vote. Dans un processus de scrutin, il faut que chaque personne puisse produire

des déclarations de vote *signée* pour attester de l'authenticité de la déclaration. Cette signature consiste en un tableau de `long` qui ne peut être généré que par l'émetteur de la déclaration (avec sa clé secrète), mais qui peut être vérifié par n'importe qui (avec la clé publique de l'émetteur). Plus précisément, nous allons utilisé le protocole de déclaration de vote suivant :

1. Un électeur $E$ souhaite voter pour le candidat $C$. Dans ce cas, la déclaration de vote de $E$ est le message `mess` obtenu en transformant la clé publique du candidat $C$ en sa représentation sous forme de chaîne de caractères (obtenu par la fonction `key_to_str`).
2. Avant de publier sa déclaration, l'électeur $E$ utilise une fonction de signature (appelée `sign` ci-après) qui permet de générer la signature associée à sa déclaration de vote. Cette signature prendra la forme d'un tableau de `long` obtenu par chiffrement du message `mess` avec la clé secrète de l'électeur $E$ (à l'aide de la fonction `encrypt`).
3. L'électeur peut ensuite publier une déclaration sécurisée, composée de sa déclaration `mess`, de la signature associée, et de sa clé publique. De cette manière, toute personne souhaitant vérifier l'authenticité de la déclaration peut le faire en déchiffrant la signature avec la clé publique de $E$ : le résultat obtenu doit correspondre exactement au message `mess`.

**Q 3.5** Une signature est donc simplement un tableau de `long` dont on connaît la longueur. Définissez la structure `Signature`.

**Q 3.6** Écrivez une fonction `Signature* init_signature(long* content, int size)` qui permet d'allouer et de remplir une signature avec un tableau de `long` déjà alloué et initialisé.

**Q 3.7** Écrivez une fonction `Signature* sign(char* mess, Key* sKey)` qui crée une signature à partir du message `mess` (déclaration de vote) et de la clé secrète de l'émetteur.

**Q 3.8** On vous donne ci-dessous les fonctions `signature_to_str` et `str_to_signature`, qui permettent de passer d'une `Signature` à sa représentation sous forme de chaîne de caractères et inversement. La chaîne de caractères est de la forme "$\#x_0\#x_1\#\ldots\#x_n\#$" où $x_i$ est le $i^{\text{ème}}$ entier du tableau de la signature donné en hexadécimal. Intégrez ces fonctions dans votre programme.

```
1  char* signature_to_str(Signature* sgn){
2      char* result = malloc(10*sgn->size*sizeof(char));
3      result[0]='#';
4      int pos = 1;
5      char buffer[156];
6      for (int i=0; i<sgn->size; i++){
7          sprintf(buffer, "%lx", sgn->content[i]);
8          for (int j=0; j< strlen(buffer); j++){
9              result[pos] = buffer[j];
10             pos = pos+1;
11         }
12         result[pos] = '#';
13         pos = pos+1;
14     }
15     result[pos] = '\0';
16     result = realloc(result, (pos+1)*sizeof(char));
17     return result;
18 }
19
20 Signature* str_to_signature(char* str){
21     int len = strlen(str);
22     long* content = (long*)malloc(sizeof(long)*len);
23     int num = 0;
24     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);
}
```

## DÉCLARATIONS SIGNÉES

On peut maintenant créer des déclarations signées (données protégées).

**Q 3.9** Définissez la structure `Protected` qui contient la clé publique de l'émetteur (l'électeur), son message (sa déclaration de vote), et la signature associée.

**Q 3.10** Écrivez une fonction `Protected* init_protected(Key* pKey, char* mess, Signature* sgn)` qui alloue et initialise cette structure (cette fonction ne vérifie pas si la signature est valide).

**Q 3.11** Écrivez une fonction `int verify(Protected* pr)` qui vérifie que la signature contenue dans `pr` correspond bien au message et à la personne contenus dans `pr`.

**Q 3.12** Écrivez les fonctions `protected_to_str` et `str_to_protected` qui permettent de passer d'un `Protected` à sa représentation sous forme de chaîne de caractères et inversement. La chaîne doit contenir dans l'ordre :

- la clé publique de l'émetteur,
- son message,
- sa signature,

séparés par un espace.

**Indication :** utilisez les fonctions `key_to_str` et `signature_to_str`.

## FONCTION DE TESTS

Pour tester vos fonctions, vous pouvez vous inspirer de ce main.

```
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 Serialization
```

```
13      char* chaine = key_to_str(pKey);
14      printf("key_to_str: %s \n", chaine);
15      Key* k = str_to_key(chaine);
16      printf("str_to_key: %lx , %lx \n", k->val, k->n);
17
18      //Testing signature
19      //Candidate keys:
20      Key* pKeyC = malloc(sizeof(Key));
21      Key* sKeyC = malloc(sizeof(Key));
22      init_pair_keys(pKeyC, sKeyC,3,7);
23      //Declaration:
24      char* mess = key_to_str(pKeyC);
25      printf("%s vote pour %s\n",key_to_str(pKey), mess);
26      Signature* sgn = sign(mess, sKey);
27      printf("signature: ");
28      print_long_vector(sgn->content,sgn->size);
29      chaine = signature_to_str(sgn);
30      printf("signature_to_str: %s \n", chaine);
31      sgn = str_to_signature(chaine);
32      printf("str_to_signature: ");
33      print_long_vector(sgn->content,sgn->size);
34
35
36      //Testing protected:
37      Protected* pr = init_protected(pKey,mess, sgn);
38      //Verification:
39      if (verify(pr)){
40          printf("Signature valide\n");
41      }else{
42          printf("Signature non valide\n");
43      }
44      chaine = protected_to_str(pr);
45      printf("protected_to_str: %s\n", chaine);
46      pr = str_to_protected(chaine);
47      printf("str_to_protected: %s %s %s\n",key_to_str(pr->pKey),pr->mess,
            signature_to_str(pr->sgn));
48
49      free(pKey);
50      free(sKey);
51      free(pKeyC);
52      free(sKeyC);
53      return 0;
54  }
```

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

Pour pouvoir mettre en place le scrutin, il faut d'abord générer pour chaque citoyen une carte électorale unique, comprenant sa clé publique et sa clé secrète, et recenser toutes les clés publiques de ces cartes électorales. Pour que le vote soit anonyme, le système ne doit pas savoir à qui correspondent ces clés publiques. Par ailleurs, les citoyens ont la responsabilité de garder leur clé secrète, et doivent l'utiliser au moment de voter, pour transmettre une déclaration de vote signée. Le système de vote collecte les déclarations signées au fur et à mesure qu'elles arrivent, et vérifie l'authenticité de chaque vote avant de le comptabiliser. Le système devra aussi vérifier que chaque citoyen a voté au plus une fois.

Dans le cadre de ce projet, nous allons simuler ce 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.

**Q 4.1** Écrivez une fonction `void generate_random_data(int` $nv$`, int` $nc$`)` qui :
- génère $nv$ couples de clés (publique, secrète) différents représentant les $nv$ citoyens,
- crée un fichier `keys.txt` contenant tous ces couples de clés (un couple par ligne),
- sélectionne $nc$ clés publiques aléatoirement pour définir les $nc$ candidats,
- crée un fichier `candidates.txt` contenant la clé publique de tous les candidats (une clé publique par ligne),
- génère une déclaration de vote signée pour chaque citoyen (candidat choisi aléatoirement),
- crée un fichier `declarations.txt` contenant toutes les déclarations signées (une déclaration par ligne).

# Base de déclarations centralisée

Dans cette partie, on considère un système de vote centralisé, 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. En pratique, 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 chaînée. Pour pouvoir vérifier l'intégrité des données et comptabiliser les votes, le système doit aussi récupérer l'ensemble des clés publiques des citoyens et des candidats, qui sont stockées respectivement dans les fichiers appelés `keys.txt` et `candidates.txt` (cf exercice précédent).

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

Dans cet exercice, on s'intéresse à la lecture et au stockage des données sous forme de listes (simplement) chaînées. On va d'abord s'intéresser aux fichiers `keys.txt` et `candidates.txt`, qui conduisent à des listes chaînées de clés (publiques), puis on va s'occuper du fichier `declarations.txt`, qui correspond à une liste chaînée de déclarations signées.

### LISTE CHAÎNÉE DE CLÉS

Pour les listes chaînées de clés, nous allons utiliser la structure suivante :

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

**Q 5.1** Écrivez une fonction `CellKey* create_cell_key(Key* key)` qui alloue et initialise une cellule de liste chaînée.

**Q 5.2** Écrivez une fonction qui ajoute une clé en tête de liste.

**Q 5.3** Écrivez une fonction `read_public_keys` qui prend en entrée le fichier `keys.txt` ou le fichier `candidates.txt`, et qui retourne une liste chaînée contenant toutes les clés publiques du fichier.

**Q 5.4** Écrivez une fonction `void print_list_keys(CellKey* LCK)` permettant d'afficher une liste chaînée de clés, puis utilisez cette fonction pour vérifier votre fonction de lecture.

**Q 5.5** Écrivez une fonction `void delete_cell_key(CellKey* c)` qui supprime une cellule de liste chaînée de clés. Écrivez ensuite une fonction `delete_list_keys` qui supprime une liste chaînée de clés.

### LISTE CHAÎNÉE DE DÉCLARATIONS SIGNÉES

Pour les listes chaînées de déclarations signées, nous allons utiliser la structure suivante :

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

**Q 5.6** Écrivez une fonction `CellProtected* create_cell_protected(Protected* pr)` qui alloue et initialise une cellule de liste chaînée.

**Q 5.7** Écrivez une fonction qui ajoute une déclaration signée en tête de liste.

**Q 5.8** Écrivez une fonction `read_protected` qui lit le fichier `declarations.txt`, et qui crée une liste contenant toutes les déclarations signées du fichier.

**Q 5.9** Écrivez une fonction d'affichage pour vérifier votre fonction de lecture.

**Q 5.10** Écrivez une fonction `void delete_cell_protected(CellProtected* c)` qui supprime une cellule de liste chaînée de déclarations signées. Écrivez ensuite une fonction qui supprime entièrement une liste chaînée.

## Exercice 6 – Détermination du gagnant de l'élection

Une fois toutes les données collectées, le système commence par retirer toutes les déclarations contenant une fausse signature (tentative de fraude).

**Q 6.1** Écrivez une fonction qui, étant donnée une liste chaînée de déclarations signées, supprime toutes les déclarations dont la signature n'est pas valide.

Pour déterminer le gagnant de l'élection de manière efficace, nous allons utiliser la structure suivante :

```
1 typedef struct hashcell{
2     Key* key;
3     int val;
4 } HashCell;
5
6 typedef struct hashtable{
7     HashCell** tab;
8     int size;
9 } HashTable;
```

Cette structure de données va nous permettre de construire deux tables de hachage :

- Une table de hachage qui permettra de compter le nombre de votes en faveur des candidats, et de vérifier en $O(1)$ que le candidat indiqué dans un vote est bien un candidat de l'élection. Pour cette table de hachage, les clés de la table sont les clés publiques des candidats, et les valeurs sont égales au nombre de votes comptabilisés.
- Une table de hachage qui permettra de vérifier en $O(1)$ que chaque personne votante est bien inscrite sur la liste électorale et qu'elle ne vote pas plus d'une fois. Pour cette table de hachage, les clés de la table correspondent aux clés publiques des citoyens inscrits sur la liste électorale, et les valeurs sont égales à zéro pour les citoyens n'ayant pas (encore) voté, et sont égales à un pour ceux qui ont déjà voté.

**Q 6.2** Écrivez une fonction `HashCell* create_hashcell(Key* key)` qui alloue une cellule de la table de hachage, et qui initialise ses champs en mettant la valeur à zéro.

**Q 6.3** Proposez une fonction `int hash_function(Key* key, int size)` qui retourne la position d'un élément dans la table de hachage.

**Q 6.4** Écrivez une fonction `int find_position(HashTable* t, Key* key)` qui cherche dans la table `t` s'il existe un élément dont la clé publique est `key`, en sachant que les collisions sont gérées par

probing linéaire. Si l'élément a été trouvé, la fonction retourne sa position dans la table, sinon la fonction retourne la position où il aurait dû être.

**Q 6.5** Écrivez une fonction `HashTable* create_hashtable(CellKey* keys, int size)` qui crée et initialise une table de hachage de taille `size` contenant une cellule pour chaque clé de la liste chaînée `keys`.

**Q 6.6** Écrivez une fonction `void delete_hashtable(HashTable* t)` qui supprime une table de hachage.

**Q 6.7** Écrivez une fonction `Key* compute_winner(CellProtected* decl, CellKey* candidates, CellKey* voters, int sizeC, int sizeV)` qui calcule le vainqueur de l'élection, étant donnés une liste de déclarations avec signatures valides (`decl`), une liste de candidats (`candidates`), et une liste de personnes autorisées à voter (`voters`). La fonction commence par créer deux tables de hachage :

— $H_c$ : une table de hachage (de taille `sizeC`) pour la liste des candidats.

— $H_v$ : une table de hachage (de taille `sizeV`) pour la liste des votants.

La fonction parcourt ensuite la liste des déclarations, et pour chaque déclaration, elle vérifie grâce aux deux tables de hachage que :

— la personne qui vote a le droit de voter et qu'elle n'a pas déjà voté,

— et que la personne sur qui porte le vote est bien un candidat de l'élection.

Si toutes ces conditions sont vérifiées, le vote est comptabilisé dans la table $H_c$ et la table $H_v$ est mise à jour pour indiquer que ce votant vient de voter. Une fois que tous les votes ont été comptabilisés, la fonction détermine le gagnant en utilisant la table $H_c$.

**Remarque :** en pratique, il faudra que `sizeC` et `sizeV` soient plus grands que le nombre de candidats et le nombre de votants respectivement, puisque les collisions sont gérées par probing.

# Blocs et persistance des données

Dans la section précédente, nous avons étudié et implémenté un système de vote centralisé, dans lequel les déclarations de vote ont été collectées et gérées par une autorité régulatrice. Pour des questions de confiance, il serait préférable d'utiliser un système de vote décentralisé, permettant à chaque citoyen de vérifier par lui-même le résultat du vote. De ce fait, nous allons à présent nous tourner vers une *blockchain* (chaîne de blocs), qui est une base de données décentralisée et sécurisée dans laquelle toutes les personnes sur le réseau possèdent une copie de la base ; dans notre contexte, les données sont les déclarations de vote signées, et les personnes sur le réseau sont les citoyens.

On va s'intéresser au protocole de vote suivant :

- **Vote :** pour voter, un citoyen doit envoyer sa déclaration de vote signée sur le réseau entre l'ouverture et la clôture du scrutin.
- **Création des blocs :** pendant tout le déroulement du scrutin, à intervalles de temps réguliers, des citoyens volontaires (assesseurs) collectent et vérifient les dernières déclarations qui ont été envoyées sur le réseau, regroupent les déclarations valides dans un bloc, et envoient ce bloc sur le réseau.
- **Mise à jour des données :** quand un citoyen reçoit un nouveau bloc par le réseau, il ajoute ce bloc à sa base de données (qui est sous forme de chaîne de blocs).

Pour rendre la fraude difficile, on va utiliser un mécanisme de consensus dit par *proof of work* (preuve de travail) fondée sur une fonction de hachage cryptographique, autrement dit une fonction de hachage facile à calculer mais difficile à inverser. Le principe général est décrit ci-dessous (on travaillera bien sûr étape par étape).

**Création de blocs valides.** Pour créer un bloc contenant les données `data` (comme des déclarations de vote), on va demander au créateur d'inverser partiellement une fonction de hachage cryptographique : le créateur doit en fait trouver un entier `nonce` tel que la valeur hachée de `data` suivi de `nonce` commence par un certain nombre de bits à zéro. Avec une fonction de hachage cryptographique, cette inversion partielle ne peut être réalisée que par brute force, et on verra que le temps nécessaire grandit exponentiellement avec le nombre de zéros requis. Cependant, quand on connaît l'entier `nonce`, on peut vérifier rapidement que le résultat commence bien par le bon nombre de zéros. De ce fait, la valeur `nonce` constitue une sorte de "preuve de travail" au sens où elle permet de vérifier facilement que le créateur du bloc a fait beaucoup de calculs pour réaliser cette inversion partielle. Un bloc ne sera considéré comme valide que s'il est accompagné d'une preuve de travail (c'est-à-dire d'un entier `nonce`). Dans ce contexte, le choix du nombre de bits à zéro permet de limiter le nombre de blocs créés, et définit implicitement l'intervalle de temps moyen entre deux blocs successifs de la *blockchain* (on verra cela plus tard).

**Gestion des fraudes.** Pour limiter les possibilités de fraude, on va demander au créateur d'un bloc d'inclure dans les données `data` la valeur hachée du bloc précédent, ce qui permet non seulement d'ordonner les déclarations de vote chronologiquement, mais aussi de détecter les fraudes. Par exemple, supposons que le citoyen malveillant $M$ souhaite envoyer des fausses données au citoyen $C$, et notons $B_i$ le dernier bloc reçu par tous les citoyens. Pour que $C$ ne se doute de rien, $M$ doit inclure dans son bloc frauduleux la valeur hachée du bloc $B_i$. Cependant, le citoyen $C$ va recevoir deux blocs avec le même

prédécesseur : le bloc frauduleux et le bloc $B_{i+1}$ (contenant la suite des vraies données). À ce moment là, le citoyen $C$ ne sait pas encore lequel est frauduleux, mais il lui suffit d'attendre un peu pour avoir la réponse. En effet, pour essayer de tromper $C$, le citoyen $M$ doit continuer de lui envoyer des blocs frauduleux, chacun contenant la valeur hachée du bloc frauduleux précédent. Comme créer un bloc est une opération coûteuse (inversion d'une fonction de hachage cryptographique) et que les autres citoyens continuent de créer des blocs collaborativement, la chaîne composée des blocs frauduleux ne peut pas avancer aussi vite que celle composée des vraies données. De ce fait, pour contourner ces fraudes, il suffit que chaque citoyen suive la règle suivante : à chaque instant, il faut faire confiance à la plus longue chaîne de blocs reçue.

## Exercice 7 – Structure d'un block et persistance

Dans cet exercice, nous allons nous intéresser à la gestion de blocs. On utilisera la structure suivante :

```
1 typedef struct block{
2     Key* author;
3     CellProtected* votes;
4     unsigned char* hash;
5     unsigned char* previous_hash;
6     int nonce;
7 }Block;
```

Ainsi, un bloc contiendra :
- La clé publique de son créateur.
- Une liste de déclarations de vote.
- La valeur hachée du bloc.
- La valeur hachée du bloc précédent.
- Une preuve de travail.

### LECTURE ET ÉCRITURE DE BLOCS

**Q 7.1** Écrivez une fonction permettant d'écrire dans un fichier un bloc. Le fichier devra contenir l'auteur du bloc, sa valeur hachée, la valeur hachée du bloc précédent, sa preuve de travail, et toutes les déclarations de vote, les unes à la suite des autres.

**Q 7.2** Écrire une fonction permettant de faire l'opération inverse, c'est-à-dire lire un bloc depuis un fichier.

### CRÉATION DE BLOCS VALIDES

Pour créer des blocs, on utilisera un mécanisme par *proof of work* : un bloc ne sera considéré comme valide que si la valeur hachée de ses données commence par $d$ zéro successifs.

**Q 7.3** Écrivez une fonction `char* block_to_str(Block* block)` qui génère une chaîne de caractères représentant un bloc. Cette chaîne de caractères doit contenir :
- La clé de l'auteur.
- La valeur hachée du bloc précédent.
- Une représentation de ses votes.
- La preuve de travail.

**Remarque :** Cette chaîne de caractères est une représentation des données contenu dans le bloc, et sera utilisée pour calculer la valeur hachée du bloc.

Dans ce projet, nous allons utiliser la fonction de hachage cryptographique `SHA-256` qui est un standard, utilisé par exemple dans le cadre des crypto-monnaies. Cette fonction prend en entrée des messages de taille variable (au plus $2^{64}$ bits), et produit des valeurs hachées de taille fixe (256 bits).

**Q 7.4** Sur votre machine, installez la bibliothèque openssl en utilisant la commande suivante : `sudo apt-get install libssl-dev` (normalement pas besoin sur les machines de la PPTI). Pour vérifier que votre installation a fonctionné, compilez et exécutez le code suivant (option compilation : `-lssl -lcrypto`).

```
#include <openssl/sha.h>

const char *s = "Rosetta code";
unsigned char *d = SHA256(s, strlen(s), 0);
int i;
for (i = 0; i < SHA256_DIGEST_LENGTH; i++)
    printf("%02x", d[i]);
putchar('\n');
```

**Remarque :** Le spécificateur %02x convertit un entier en hexadécimal, et si besoin rajoute un zéro devant pour avoir au moins deux chiffres.

**Q 7.5** En vous inspirant des lignes de code précédentes, écrivez une fonction qui prend une chaîne de caractères, et retourne sa valeur hachée obtenue par l'algorithme `SHA256`. Appliquez votre fonction sur la chaîne "Rosetta code" pour vérifier votre fonction.

**Q 7.6** Pour rendre un bloc valide, on procède par brute force : on commence avec l'attribut `nonce` égal à zéro, puis on incrémente l'attribut **nonce** jusqu'à ce que la valeur hachée du bloc commence par $d$ zéros successifs. Écrivez une fonction `void compute_proof_of_work(Block *B, int d)` qui réalise ces opérations.

**Remarque :** Comme la valeur hachée est donnée en hexadécimal, exiger $d$ zéros successifs revient à demander $4d$ zéros successifs sur la représentation binaire.

**Q 7.7** Écrivez une fonction `int verify_block(Block*, int d)` qui vérifie qu'un bloc est valide.

**Q 7.8** Étudiez le temps moyen de la fonction `compute_proof_of_work` selon la valeur de $d$. Tracez une courbe du temps moyen en fonction de $d$, et en déduire la valeur de $d$ à partir de laquelle ce temps dépasse une seconde.

**Remarque :** Le choix de la valeur $d$ permet de limiter le nombre de blocs créés, en définissant implicitement l'intervalle de temps moyen entre deux blocs successifs de la *blockchain*.

**Q 7.9** Écrivez une fonction `void delete_block(Block* b)` qui supprime un bloc. Cette fonction ne libère pas la mémoire associée au champs `author`. Pour la liste chaînée `votes`, on libère les éléments de la liste chaînée (`CellProtected`), mais pas leur contenu (`Protected`).

## Exercice 8 – Structure arborescente

Dans une *blockchain*, chaque bloc contient la valeur hachée du bloc qui le précède, ce qui forme une chaîne. Cependant, en cas de triche, on peut se retrouver avec plusieurs blocs indiquant le même bloc

précédent, ce qui conduit à une structure arborescente. Dans ce cas, la règle à suivre est de faire confiance à la chaîne la plus longue (en partant de la racine de l'arbre), ce qui permet de retomber sur une chaîne de blocs.

### MANIPULATION D'UN ARBRE DE BLOCS

Pour représenter un arbre de blocs, nous allons utiliser la structure suivante :

```
1 typedef struct block_tree_cell{
2    Block* block;
3    struct block_tree_cell* father;
4    struct block_tree_cell* firstChild;
5    struct block_tree_cell* nextBro;
6    int height;
7 }CellTree;
```

Cette structure permet de représenter des noeuds avec un nombre arbitraire de fils, sans avoir à créer de structure de liste chaînée. En effet, pour parcourir tous les fils d'un noeud, il suffit de récupérer le premier fils à l'aide du champs `firstChild`, puis d'utiliser le champs `nextBro` pour obtenir les fils suivants, jusqu'à tomber sur le pointeur `NULL`. Le champs `height` correspond à la hauteur du noeud, exprimée en nombre d'arcs ; en particulier, une feuille de l'arbre possède une hauteur égale à zéro.

**Q 8.1** Écrivez une fonction `CellTree* create_node(Block* b)` permettant de créer et initialiser un noeud avec une hauteur égale à zéro.

**Q 8.2** Écrivez une fonction `int update_height(CellTree* father, CellTree* child)` permettant de mettre à jour la hauteur du noeud `father` quand l'un de ses fils a été modifié : la hauteur du noeud `father` doit être égale au max entre sa hauteur courante et la hauteur du noeud `son` + 1. Cette fonction doit retourner 1 si la hauteur du noeud `father` a changé, et 0 sinon.

**Q 8.3** Écrivez une fonction `void add_child(CellTree* father, CellTree* child)` qui ajoute un fils à un noeud, en mettant à jour la hauteur de tous les ascendants.

**Q 8.4** Écrivez une fonction `print_tree` qui affiche un arbre : pour chaque noeud, il faut afficher la hauteur du noeud et la valeur hachée du bloc correspondant (son identifiant en quelque sorte).

**Q 8.5** Écrivez une fonction `void delete_node(CellTree* node)` qui supprime un noeud de l'arbre, en faisant appel à la fonction `delete_block` (question 7.9). Écrivez ensuite une fonction qui supprime l'arbre.

### DÉTERMINATION DU DERNIER BLOC

À partir d'un arbre de blocs, le consensus est de faire confiance à la chaîne de blocs la plus longue en partant de la racine de l'arbre. Comme chaque bloc contient la valeur hachée du bloc précédent, il faut identifier efficacement le dernier bloc de cette chaîne pour pouvoir créer des nouveaux blocs.

**Q 8.6** Écrivez une fonction `CellTree* highest_child(CellTree* cell)` qui renvoie le noeud fils avec la plus grande hauteur.

**Q 8.7** Utilisez la fonction précédente pour écrire une fonction `CellTree* last_node(CellTree* tree)` permettant de retourner la valeur hachée du dernier bloc de cette plus longue chaîne.

### EXTRACTION DES DÉCLARATIONS DE VOTE

La règle est de faire confiance à la chaîne de blocs la plus longue en partant de la racine de l'arbre. Chaque bloc de cette chaîne contient une liste chaînée de déclarations signées (champs `votes`). Pour déterminer le vainqueur de l'élection, il suffit de fusionner ces listes chaînées de déclarations, puis de faire appel à la fonction `compute_winner`.

**Q 8.8** Écrivez une fonction permettant de fusionner deux listes chaînées de déclarations signées. Quelle est la complexité de votre fonction ? Quelle modification dans la structure faudrait-il faire pour avoir une fusion en $O(1)$ ? On ne vous demande pas d'implémenter cette modification.

**Q 8.9** Utilisez la fonction précédente et la fonction `highest_child` pour retourner la liste obtenue par fusion des listes chaînées de déclarations contenues dans les blocs de la plus longue chaîne.

## Exercice 9 – Simulation du processus de vote

Dans le cadre de ce projet, on simulera le fonctionnement d'une *blockchain* en utilisant le répertoire et le fichiers suivants :

- **Blockchain :** répertoire représentant la *blockchain* qu'un citoyen a construit en local, à partir des blocs qu'il a reçu du réseau. Ce répertoire contient un fichier par bloc de la *blockchain*.
- **Pending_block :** fichier contenant le dernier bloc créé et envoyé par un des assesseurs. Ce bloc est en attente d'ajout dans la *blockchain*.
- **Pending_votes.txt :** fichier texte contenant les votes en attente d'ajout dans un bloc.

Commencez par créer le répertoire "Blockchain". Dans cet exercice, nous allons simuler les soumissions de vote par les citoyens et les créations de blocs valides par des assesseurs (citoyens volontaires). Puis, nous allons créer l'arbre de blocs correspondant, et calculer le gagnant de l'élection en faisant confiance à la plus longue chaîne de l'arbre.

### VOTE ET CRÉATION DE BLOCS VALIDES

**Q 9.1** Écrivez une fonction `void submit_vote(Protected* p)` permettant à un citoyen de soumettre un vote, autrement d'ajouter son vote à la fin du fichier "Pending_votes.txt".

**Remarque :** si le fichier n'existe pas, cette fonction le crée.

**Q 9.2** Écrivez une fonction `void create_block(CellTree* tree, Key* author, int d)` qui :

- crée un bloc valide contenant les votes en attente dans le fichier "Pending_votes.txt",
- supprime le fichier "Pending_votes.txt" après avoir créé le bloc,
- et écrit le bloc obtenu dans un fichier appelé "Pending_block".

**Indication :** cette fonction doit utiliser `last_node`, `read_protected`, `compute_proof_of_work` et `write_block` (question 7.1).

**Q 9.3** Écrire une fonction `void add_block(int d, char* name)` qui vérifie que le bloc représenté par le fichier "Pending_block" est valide. Si c'est le cas, la fonction crée un fichier appelé `name` représentant le bloc, puis l'ajoute dans le répertoire "Blockchain". Dans tous les cas, le fichier "Pending_block" est ensuite supprimé.

## LECTURE DE L'ARBRE ET CALCUL DU GAGNANT

On souhaite maintenant construire l'arbre correspondant aux blocs contenus dans le répertoire "Blockchain". Pour cela, on va procéder en plusieurs étapes :

1. On crée un noeud de l'arbre pour chaque bloc contenu dans le répertoire, et on stocke tous les noeuds dans un tableau $T$ de type `CellTree**`.
2. On parcourt le tableau $T$ de noeuds, et pour chaque noeud $T[i]$, on recherche tous ses fils $T[j]$ ; ce sont les noeuds contenant un bloc dont le champs `previous_hash` est égal au champs `hash` du bloc du noeud $T[i]$. Chaque fils $T[j]$ est ajouté à la liste des fils du noeud $T[i]$ (avec la fonction `add_child`).
3. On parcourt une dernière fois le tableau $T$ pour trouver la racine de l'arbre (noeud dont le champs `father` est égal à NULL), et la retourner.

**Q 9.4** Écrivez une fonction `CellTree* read_tree()` qui réalise ces opérations. Pour l'étape 1, inspirez-vous du code suivant :

```
#include <dirent.h>
#include <stdio.h>

int main(){
    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);
            }
        }
        closedir(rep);
    }
}
```

**Q 9.5** Écrivez une fonction `Key* compute_winner_BT(CellTree* tree, CellKey* candidates, CellKey* voters, int sizeC, int sizeV)` qui détermine le gagnant de l'élection en se basant sur la plus longue chaîne de l'arbre. Plus précisément, cette fonction réalise les étapes suivantes :

1. Extraction de la liste des déclarations de vote (avec la fonction de la question 8.8).
2. Suppression des déclarations de vote non valides (avec la fonction de la question 6.1).
3. Calcule le vainqueur de l'élection (avec la fonction de la question 6.7)

**Rappel :** les entiers `sizeC` et `sizeV` correspondent à la taille des tables de hachage utilisées pour le calcul du vainqueur de l'élection.

**Q 9.6** Écrivez une fonction `main` qui réalise les opérations suivantes :

1. Génération d'un problème de vote avec 1000 citoyens et 5 candidats (fonction `generate_random_data`).
2. Lecture des déclarations de vote (fonction `read_protected`), des candidats et des citoyens (fonction `read_keys`).
3. Soumission de tous les votes (fonction `submit_vote`), avec la création d'un bloc valide tous les 10 votes soumis (fonction `create_block`), suivi directement par l'ajout du bloc dans la *blockchain* (fonction `add_block`).

4. Lecture (fonction `read_tree`) et affichage (fonction `print_tree`) de l'arbre final.
5. Calcul et affichage du gagnant (fonction `compute_winner_BT`).

**Q 9.7** Conclusion du projet : que pensez-vous de l'utilisation d'une *blockchain* dans le cadre d'un processus de vote ? Est-ce que le consensus consistant à faire confiance à la plus longue chaîne permet d'éviter toutes les fraudes ?