LU2IN006 - Structures de données

Projet : Blockchain appliquée à un processus électoral

Vincent Fiszbin 28711002 et Davy Tse 3672079

Table des matières :

Reformulation du sujet

Le projet a pour objectif l'implémentation de protocoles et structures de données permettant la simulation d'un processus de vote transparent et garantissant l'intégrité des votes de chaque électeur.
Il est possible de chiffrer, signer et déchiffrer la déclaration de vote d'un électeur, de vérifier sa validité grâce aux outils cryptographiques implémentés dans la partie 1.
Les fonctions de la partie 2 permettent la manipulation de structures représentant les clés, intentions de vote, signatures, déclarations. Il est également possible de générer des citoyens, parmi lesquels sont choisis aléatoirement des candidats qui reçoivent des intentions de vote elles-mêmes aléatoires. Les citoyens, candidats et déclarations de votes peuvent être écrits dans des fichiers pour en conserver la trace.
Les informations contenues dans ces fichiers peuvent en être extraites grâce aux fonctions de la partie 3. La validité des déclarations signées ainsi extraites peut être vérifiée, ce qui est une première étape pour garantir l'intégrité de l'élection. Une fois les déclarations validées, il est possible de les "dépouiller" et de déterminer le gagnant de l'élection.
La partie 4 nous amène à la création d'un système d'élection décentralisée basé sur le principe de blockchain. Les fonctions de cette partie concernent la création de blocks, la validation de ceux-ci par un système de preuve de travail, la mise en place d'arbres de blocks pour y déterminer la branche la plus longue, c'est-à-dire la blockchain qui doit être la plus digne de confiance.
C'est aussi à l'issue de cette partie que la simulation complète d'une élection est mise en place, en mettant bout à bout les fonctions créées dans les parties précédentes.

Organisation et utilisation des fichiers

Tous les fichiers du projet sont dénommés selon le même schéma indiquant le numéro de la partie et le numéro de l'exercice correspondant dans le sujet.
Par exemple, le fichier p1exo1.c contient le code source des questions posées dans l'exercice 1 de la partie 1 du sujet.

Les headers sont dénommés de la même manière et contiennent les importations de bibliothèques nécessaires, les déclarations de structures ainsi que les prototypes de fonctions de l'exercice concerné.

Il existe un fichier main pour chaque partie qui regroupe les tests de toutes les fonctions de la partie en question. Le seul fichier qui ne respecte pas le schéma p.x.exo.x, est le fichier mainfinal.c qui contient le main de simulation d'élection.

Le projet peut être compilé à l'aide du Makefile qui, par défaut, génère 5 exécutables : p1main, p2main, p3main et p4main correspondants au main de chaque partie, et mainfinal qui exécute la simulation complète d'élection conformément à ce qui est demandé dans la question 9.6.
Il est possible de ne compiler, par exemple, que les fichiers correspondants à la partie 1 en utilisant la règle p1main.

Pour tester les fonctions du projet, on peut lancer chacun de ces exécutables. A la fermeture des programmes, ou en cas d'erreur quelconque lors de leur exécution, toute mémoire allouée est libérée pour éviter les fuites de mémoire.

Important : avant d'exécuter mainfinal, il faut s'assurer que le répertoire Blockchain existe bien, et qu'il ne contient pas déjà des fichiers pouvant interférer avec le fonctionnement normal du programme. Des fichiers résiduels dans ce répertoire peuvent être **à l'origine de fuites mémoires s'il ne sont pas supprimés avant de lancer mainfinal**. Pour supprimer tous les fichiers temporaires, dont ceux du répertoire Blockchain, vous pouvez utiliser la commande **make clean**. Nous conseillons d'utiliser make clean avant de lancer mainfinal.

Description des fonctions principales et structures manipulées

Partie 1 : Implémentation d'outils de cryptographie

Exercice 1 : Résolution du problème de primalité

Les fonctions de cet exercice sont des fonctions utilitaires pour le cryptage des messages dans les parties suivantes. Le but de l'exercice est la génération aléatoire de nombre premiers, la fonction **random_prime_number** assure cette tâche en générant aléatoirement une entier premier d'une certaine taille (en nombre de bits). Pour cela, la fonction génère continuellement des entiers de la bonne taille jusqu'à ce que l'un d'eux passe le test de primalité de Miller qui permet d'affirmer, avec une probabilité relativement importante d'avoir raison, que cet entier est bien premier.

```
random_prime_number(low_size, up_size):
  while (not is_prime_miller(p)) {
      p = random long between 2^(low_size - 1) and 2^(up_size) - 1
  }
  return p
```

Pseudocode résumant l'algorithme de la fonction random_prime_number

La plupart des autres fonctions de l'exercice sont des fonctions supports ou alternatives pour établir la primalité d'un entier. De plus amples détails sur ces fonctions sont donnés, notamment sur leur complexité, dans la partie Réponses aux questions de ce rapport.

Exercice 2 : Implémentation du protocole RSA

La fonction **generate_key_values** génère une clé publique et une clé privée selon le protocole RSA tel que décrit dans le sujet, à partir de nombres premiers eux mêmes

générés en utilisant les fonctions vues dans l'exercice 1. Le protocole requiert un calcul de PGCD réalisé dans la fonction **extended_gcd** selon l'algorithme d'Euclide étendu.

Pour envoyer une message chiffré, on peut l'encrypter avec la clé publique du destinataire à l'aide de la fonction encrypt qui itère sur le message et encrypte chaque caractère en faisant appel à modpow. Tous les caractères encryptés sous forme d'entiers sont stockés dans un tableau retourné par la fonction encrypt.

```
encrypt(message, cle_publique(s,n)):
   message_crypte = []
   i = 0
   while (message[i]) {
       m = message[i]
       message_crypte[i] = m^s mod n
       i++;
   }
   return message_crypte
```

Pseudocode résumant l'algorithme de la fonction encrypt

Le message chiffré peut ensuite être déchiffré à l'aide de la clé privé du destinataire dans la fonction decrypt qui, en connaissant la taille du tableau contenant le message chiffré, décrypte chaque caractère pour retrouver le message d'origine.

```
decrypt (message_crypte, taille_message, cle_secrete(u,n)){
   message_decrypte = ""
   for (i = 0 to taille_message){
       c = message_crypte[i]
       message_decrypte[i] = c^u mod n
   }
   return message_decrypte;
}
```

Pseudocode résumant l'algorithme de la fonction decrypt

Partie 2 : Déclarations sécurisées

Exercice 3 : Manipulation de structures sécurisées

Il s'agit, dans cet exercice, de créer des fonctions de manipulation des structures utilisées lors du processus de vote.
Parmi ces structures il y a **Key**, qui ne fait que regrouper en une seule entité (une clé) les deux valeurs composant une clé, privé ou publique. Cette structure peut s'apparenter à un couple de données.

```c
typedef struct key{
   long val;
   long n;
} Key;
```
Code C de déclaration de la structure Key

La structure **Signature** permet quant à elle de stocker à la fois le message chiffré, c'est à dire un tableau de long, retourné par la fonction encrypt et la taille de ce tableau qui est nécessaire lorsque l'on souhaite déchiffrer ce message.

```c
typedef struct signature{
   long* content;
   int size;
} Signature;
```
Code C de déclaration de la structure Signature

La structure **Protected** regroupe les deux structures précédentes en une seule, elle encapsule toutes les données qui constituent une déclaration de vote de la part d'un citoyen, à savoir sa clé publique qui l'identifie aux yeux des autres citoyens, son message (déclaration de vote) et la signature associée, chiffrée à l'aide de sa clé privée.

```c
typedef struct protected{
   Key *pKey;
   char *mess;
   Signature *sgn;
} Protected;
```
Code C de déclaration de la structure Protected

Nous avons créé des fonctions pour manipuler ces structures comme **init_key** qui permet d'initialiser une structure Key avec une clé publique et privée.
init_pair_keys initialise une clé public pKey et une clé secrète sKey avec des valeurs générées aléatoirement selon le protocole RSA puisqu'elle fait appel à la fonction **generate_key_values** de l'exercice précédent.
La fonction **sign** crée une signature à partir d'un message qu'elle encrypte et dont elle passe le tableau contenant les caractères encryptés ainsi que sa taille à la fonction **init_signature**. Cette dernière alloue et remplit une structure de type Signature avec les arguments donnés en paramètre.
init_protected ne fait qu'allouer et remplir les champs d'une structure Protected.
Enfin, une fonction verify permet de vérifier que la signature contenue dans une structure Protected correspond bien au message et à la personne contenus dans celle-ci. C'est cette fonction qui permet de détecter qu'une déclaration de vote est invalide car son intégrité a pu être altérée.

Le reste des fonctions de cet exercice permet de passer chacune des structures décrites ci-dessus sous forme de chaîne de caractère, et inversement d'initialiser chaque structure à partir de sa représentation en chaîne de caractère.

Exercice 4 : Implémentation d'un mécanisme de consensus

Pour les besoins de cet exercice nous avons défini un structure **Citoyen** qui regroupe la clé publique et privé d'un citoyen dans le processus de vote, c'est en quelque sorte sa carte électorale.

```
typedef struct citoyen{
   Key *clepublic
   Key *cleprive;
}Citoyen;
```

Code C de déclaration de la structure Citoyen

La fonction **generate_random_data** est une mise en pratique des fonctions créées dans l'exercice 3.
Tout d'abord un certain nombre de "citoyens" sont créés, ou plutôt leur carte électorale représentée ici par la structure Citoyen, chacun se voit attribuer une clé publique et privé générées aléatoirement qui sont ensuite écrites dans un fichier keys.txt. Chaque structure citoyen est aussi enregistrée dans un tableau à une dimension, nous avons fait le choix de dupliquer cette information qui se trouve donc à la fois sur le disque dans le fichier keys.txt et en mémoire vive dans le tableau contenant tous les citoyens car ne pas avoir à relire sans arrêt le fichier keys.txt facilite grandement la suite des opération.
En effet, nous sommes fréquemment amenés à choisir aléatoirement un citoyen parmi ceux existants, ce qui revient à générer un entier aléatoire ne dépassant pas le nombre de citoyens qui peut être utilisé comme indice pour accéder au citoyen en question dans le tableau.
Cette opération d'une complexité O(1) nous amène à penser qu'un tableau est la structure de données la plus indiquée dans cette situation, alors qu'avec une liste chaînée par exemple, l'accès à un élément dont on connaît l'indice serait en Θ(i), où i est l'indice.

Une fois les citoyens générés, on choisit aléatoirement parmi eux un certain nombre de candidats, leurs clés publiques sont écrites dans le fichier candidates.txt et là encore nous avons choisi de garder l'ensemble des candidats dans un tableau pour les raisons déjà mentionnées.

Il est à noter que lors de la génération des clés publiques et privées des citoyens, la fonction vérifie systématiquement que ces clés ne sont pas déjà présentent dans le tableau, ce qui implique un certain coût en temps de calcul puisque le tableau est parcouru à chaque fois, mais cela assure qu'il n'existe jamais deux fois le même citoyen. De même lorsque les candidats sont tirés aléatoirement, on s'assure que le candidat n'a pas déjà été tiré pour ne pas dupliquer les clés publiques d'un candidat. De cette façon, aucun candidat ne peut prendre l'avantage en récoltant plus de voix à l'aide de candidatures multiples.

Pour finir, on itère une dernière fois sur le tableau des citoyens pour que chacun vote pour un candidat choisi de manière aléatoire. Là encore l'usage de tableaux s'illustre par son efficacité tant l'opération d'accès à un élément dont on connaît l'indice est répétée.
Les déclarations de vote des citoyens sont écrites dans le fichier declarations.txt qui contient donc la clé publique, le message et la signature de chaque déclaration.

Les possibilités d'erreur d'allocation sont nombreuses dans cette fonction, et nous avons protégé, à notre connaissance, l'ensemble de ces possibilités pour que le programme se termine proprement en libérant toute mémoire allouée. La fonction **free_generate_random_data** permet d'alléger quelque peu le nombre de lignes occupées pour la protection des mallocs dans la fonction.

Exemple de clé publique et de clé secrète (keys.txt):
pKey: (a7,1bd), sKey: (d7,1bd)
pKey: (67,d7), sKey: (1f,d7)
pKey: (611,ca7), sKey: (b51,ca7)
pKey: (899,1295), sKey: (b2d,1295)
pKey: (3b,103), sKey: (b,103)

Exemple de structure Protected : clé public, message et sa signature (declarations.txt)
(a7,1bd) (611,ca7) #1b8#15d#5e#5e#144#135#80#186#155#
(67,d7) (899,1295) #a#65#d#d#2c#31#32#d#98#79#
(611,ca7) (685,b33) #9ed#2ad#8bc#824#738#716#2cb#2cb#abb#
(899,1295) (17,25e5) #bef#da0#f64#7ee#1238#91b#ed8#91b#13e#
(3b,103) (33,1c3) #d5#c1#c1#7b#46#4e#c1#84#

Partie 3 : Base de déclarations centralisé

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

Dans cette partie, les fonctions créées servent à lire les fichiers (keys.txt, candidates.txt et declarations.txt) générées dans l'exercice précédent, et à stocker les informations qu'ils contiennent, à savoir les déclaration de vote de type Protected et les clés des citoyens ou candidats, dans des listes chaînées.
Nous avons donc rédigé des fonctions de lecture de fichier, de création et suppression de listes chaînées, d'ajout d'élément en tête de liste, et d'affichage de ces listes.

Lors de l'élaboration des fonctions de calcul de la valeur hachée d'un bloc de déclarations de vote, nous avons rencontré un problème : lorsqu'on écrit une liste chaînée dans un fichier et qu'on la lit, son ordre est inversé étant donné que l'insertion se fait en tête. Or, lors du calcul de la valeur hachée d'un bloc, l'ordre des déclarations a son importance : le modifier revient à modifier la représentation en chaîne de caractères utilisée pour calculer la valeur hachée du bloc. Pour cette raison, nous avons conçu une deuxième fonction d'insertion : en fin de liste chaînée, afin de préserver l'ordre des listes et donc l'intégrité des valeurs hachées. Ceci se fait au prix d'une complexité en O(n) où n est la taille de la liste chaînée, là où l'insertion en tête se faisait en O(1).
Il est cependant possible de retrouver une complexité en O(1) même avec l'insertion en fin de liste à condition de conserver un pointeur sur la fin de la liste.

Les deux structures définissant les éléments des listes chaînées de l'exercice :

```c
typedef struct cellKey{
   Key* data;
   struct cellKey* next;
} CellKey;
```
Code C de déclaration de la structure CellKey

```c
typedef struct cellProtected{
   Protected* data;
   struct cellProtected* next;
} CellProtected;
```
Code C de déclaration de la structure CellProtected

Exercice 6 : Détermination du gagnant de l'élection

Pour cet exercice où il est question de déterminer le candidat qui rempotre l'élection, nous avons écrit la fonction **supprime_declarations_non_valides** qui parcourt une liste chaînée de déclarations et en teste la validité avec la fonction **verify** qui assure que la signature contenue dans la déclaration correspond bien au message et à la clé aussi contenus dans la déclaration (de type Protected). Les déclarations non valides sont supprimées de la liste chaînée. Dans un processus électoral, cela permettrait de filtrer les tentatives de fraude puisque les votes dont l'intégrité n'est plus assurée (ce serait le cas si l'on tentait de remplacer la clé du candidat pour lequel un électeur a voté) sont détectés et non comptabilisés.
Dans la suite de l'exercice nous avons conçu des fonctions de manipulation (création, suppression, recherche) de tables de hachage avec résolution des collisions par probing linéaire. Cela signifie que pour insérer un élément dans la table de hachage, on tente d'abord de l'insérer à la position retournée par la fonction de hachage, et si cette dernière est déjà occupée, on tente l'insertion dans la case suivante, et ainsi de suite jusqu'à trouver une case libre. Ce processus est circulaire puisque si l'on atteint la dernière case de la table, on repart de la première, et on est certain de ne pas pouvoir insérer/trouver l'élément que lorsqu'on est revenu à la position initiale donnée par la fonction de hachage. Cela signifie que la complexité pire cas des opérations d'insertion/recherche est en O(n) où n est la taille de la table de hachage.
La résolution par probing suppose que la table de hachage est au moins aussi grande que l'ensemble des éléments que l'on souhaite y stocker. Idéalement, la table doit être considérablement plus grande que l'ensemble à contenir puisque cela limite le nombre de tentatives infructueuses d'insertion/recherche des éléments.
Pour la fonction de hachage, nous avons choisi d'additionner les deux valeurs composant une clé (val et n) afin d'assurer une répartition suffisamment uniforme, et le résultat de cette opération est modulo la taille de la table de hachage.

```c
typedef struct hashcell{
   Key* key;
   int val;
}HashCell;
```
Code C de déclaration de la structure HashCell

```c
typedef struct hashtable{
   HashCell** tab;
   int size;
}HashTable;
```
Code C de déclaration de la structure HashTable

La fonction **compute_winner** fait appel à deux tables de hachage : l'une contenant les votants, et l'autre les candidats à l'élection. Ces tables sont alors utilisées pour déterminer le vainqueur de l'élection de la façon suivante :
Pour chaque déclaration (que l'on suppose valide puisque la fonction supprime_declarations_non_valides doit être utilisée sur la liste de déclaration en amont) on recherche la clé du votant dans la table de hachage des votants pour s'assurer qu'il a bien le droit de voter et on vérifie que la valeur associée à sa clé dans la table correspond bien à 0, ce qui confirme qu'il n'a pas déjà voté.
Ensuite, on vérifie de la même manière dans la table des candidats que le candidat indiqué dans la déclaration est valide, puis on comptabilise ce vote en incrémentant de 1 la valeur associée à la clé du candidat dans cette table.

L'utilisation de tables de hachages est tout indiquée pour réaliser ces opérations, puisque l'accès à la valeur associée aux clés publiques des déclarations se fait en O(1), tandis qu'avec une liste ou un tableau par exemple, la complexité d'une opération équivalente serait en O(n) (n la taille du tableau/liste), puisqu'on devrait parcourir le tableau/liste jusqu'à trouver la clé recherchée.

Après avoir dépouillé toutes les déclarations, il faut encore parcourir la table des candidats (une dernière opération en Θ(n), n la taille de la table) pour déterminer le candidat dont le nombre de votes est maximum, il s'agit du gagnant de l'élection.

Il est à noter que nous n'avons pas pris de dispositions particulière en cas d'ex aequo de plusieurs candidats, il pourrait être envisageable de tirer au sort qui l'emportera parmi les candidat à égalité, ou d'invalider l'élection pour qu'elle recommence. Nous n'avons pas choisi de méthode de résolution particulière, et le premier candidat dans la table de hachage à avoir le nombre maximum de voix sera le gagnant.

Partie 4 : Blocs et persistance des données

Exercice 7 : Structure d'un block et persistance

Dans cet exercice nous manipulons des bloques de déclarations de vote dont voici la structure en C :

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

Code C de déclaration de la structure Block

Nous avons pris la liberté de rajouter un champ à la structure initialement proposée dans le sujet : le nombre de votes (nb_votes) contenu dans la liste votes, car cela facilite grandement certaines opérations, notamment la lecture d'un Block depuis un fichier.

En effet, nous avons conçu une fonction d'écriture (**ecrire_block**) qui écrit la représentation d'un Block dans le fichier dont le nom est donné en argument, et une fonction de lecture (**lire_block**) qui alloue et initialise les champs d'un Block en lisant la représentation du Block depuis un fichier. Etant donné que nous conservons le nombre de votes dans la structure, cela nous permet de savoir exactement le nombre de lignes que la fonction doit lire pour reconstituer la liste chaînée de votes du Block.

Dans ce projet, la structure Block est l'élément constitutif de la Blockchain utilisée lors du processus d'élection. Comme pour d'autres types de Blockchain, le système de validation que nous utilisons est celui de la preuve de travail, cette preuve se fait avec la fonction **compute_proof_of_work** qui incrémente l'entier nonce d'un block jusqu'à ce que sa valeur hachée commence bien par un nombre d (donné en argument) de zéros.

La valeur hachée est calculée par la fonction **crypt_to_sha256** qui prend en argument la représentation sous forme de chaîne de caractère du Block. Cette représentation contient la clé de l'auteur du Block, les votes, la preuve de travail nonce et la valeur hachée du bloc précédent, de telle sorte que si le moindre de ces éléments est modifié, la valeur hachée du Block change complètement et ne commence plus par d zéros successifs, ce qui rend le Block invalide.
La fonction **verify_block** vérifie la validité d'un Block en s'appuyant sur ce principe, elle recalcule la valeur hachée du Block et s'assure que cette dernière commence bien par d zéros successifs.

Exercice 8 : Structure Arborescente

Les Blocks d'une Blockchain sont chaînées les uns aux autres par le champ previous_hash qui indique la valeur hachée du bloc précédent, et qui, si elle est modifiée, change du tout au tout la valeur hachée du Block, le rendant par la même invalide, cela assure l'inviolabilité de l'ordre dans la chaîne de Blocks.
Mais en cas de tentative de fraude ou d'incohérence, plusieurs Blocks peuvent avoir le même previous_hash, il n'existe alors plus une unique chaîne mais plusieurs chaînes possibles qui peuvent être représentées par une structure arborescente telle que celle définie ci-dessous :

```
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;
```

Code C de déclaration de la structure CellTree

Afin de manipuler ces structures arborescentes, nous avons écrit divers fonctions de gestion d'un arbre dont les nœuds sont des Blocks (insertion, suppression, affichage des nœuds).
Dans un tel arbre, la fonction **highest_child** retourne le nœud fils du nœud passé en argument ayant la plus grande hauteur (par rapport à la racine).
La fonction **fusion_liste_protected** permet la fusion de deux listes chaînées de Protected, c'est-à-dire la fusion de listes de déclarations de votes. Pour ce faire, la fonction duplique (cette étape pourrait être évitée, mais elle nous permet de libérer séparément la liste contenue dans l'arbre et celle retournée par la fonction) et insère en tête de la première liste les éléments de la seconde. Cela implique donc de parcourir l'entièreté de la seconde liste à chaque appel de fonction, une opération qui pourrait se résumer à un changement de pointeur comme expliqué dans la partie réponse aux questions.

Enfin, **fusion_votes_arbre** fait appel aux deux fonctions précédentes pour fusionner toutes les déclarations de vote de la plus longue branche de l'arbre, c'est-à-dire la branche pour laquelle le plus de travail a été fourni. La règle veut que c'est cette branche (cette Blockchain parmi celles possibles) qui doit être retenue comme étant la plus digne de confiance par tous les participants au système.

Exercice 9 : Simulation du processus de vote

Pour cet ultime exercice nous avons réalisé les dernières fonctions nécessaires à la simulation du processus électoral.
submit_vote tout d'abord, qui permet d'ajouter une déclaration de vote à la fin du "Pending Vote.txt".
Ce fichier est ensuite lu par la fonction **create_block** qui regroupe les déclarations lues dans un Block (la structure décrite dans l'exercice 7) et initialise le champ previous_hash en

récupérant le last_node (retourne la valeur hachée du dernier Block de la branche la plus longue d'un arbre) de l'arbre passé en argument. C'est donc create_block qui vient "chaîner" les Blocks successifs entre eux et donc créer la Block**chain**. Ensuite la valeur hachée et la preuve de travail nonce sont calculées par un appel de compute_proof_of_work, puis la représentation du Block est écrite dans un fichier "Pending_Block".

Le fichier Pending_Block est à son tour lu par la fonction **add_block** qui va vérifier la validité du Block lu en appelant verify_block, après quoi, s'il est valide, le Block, sera écrit dans le répertoire "Blockchain"

L'intégralité de ce répertoire est par la suite lue dans la fonction **read_tree** qui stocke les Blocks dans un tableau de nœuds.
Pour chaque élément du tableau, le tableau est parcouru dans son ensemble afin de déterminer les fils de l'élément. Ces parcours successifs permettent de reconstituer l'arbre utilisé lors de la création des Blocks avec create_block, et la complexité de cette opération en en Θ(n^2) où n est la taille du tableau de noeuds.
Le tableau est ensuite parcouru une dernière fois pour trouver le seul nœud sans père, à savoir la racine de l'arbre. C'est un pointeur sur cette racine qui est retourné par la fonction.

Une fois l'arbre de Blocks reconstitué, il est passé en argument de la fonction **compute_winner_BT** qui va appeler fusion_votes_arbre pour obtenir la liste des déclarations de vote de la plus longue branche de l'arbre, cette liste est ensuite débarrassée de ses déclarations invalides par supprime_declarations_non_valides avant d'être utilisée par compute_winner pour déterminer le gagnant de l'élection.
La suite de l'exercice consiste à mettre bout à bout toutes ces fonctions pour créer la simulation d'élection.

Ce processus est illustré par notre schéma récapitulatif qui explicite le lien entre les différentes fonctions appelées (et décrites précédemment dans le rapport) ainsi que les principales structures de données utilisées à chaque étape. Le schéma se trouve dans les pages ci-dessous et en PDF parmi les fichiers du rendu.

Pour sélectionner des assesseurs, qui sont dans ce processus les auteurs des Blocks, nous avons créé des fonctions dans le fichier mainfinal :
La première, **tab_assesseurs**, remplit et retourne un tableau de clés en sélectionnant les nb premiers citoyens pour officier en tant qu'assesseur.
La seconde, **random_assesseur**, choisit aléatoirement un assesseur dans le tableau d'assesseurs disponibles. Nous avons fait le choix d'un tableau, plutôt qu'une liste par exemple, puisque l'accès à un élément dont on connaît l'indice (tiré aléatoirement) se fait en Θ(1).

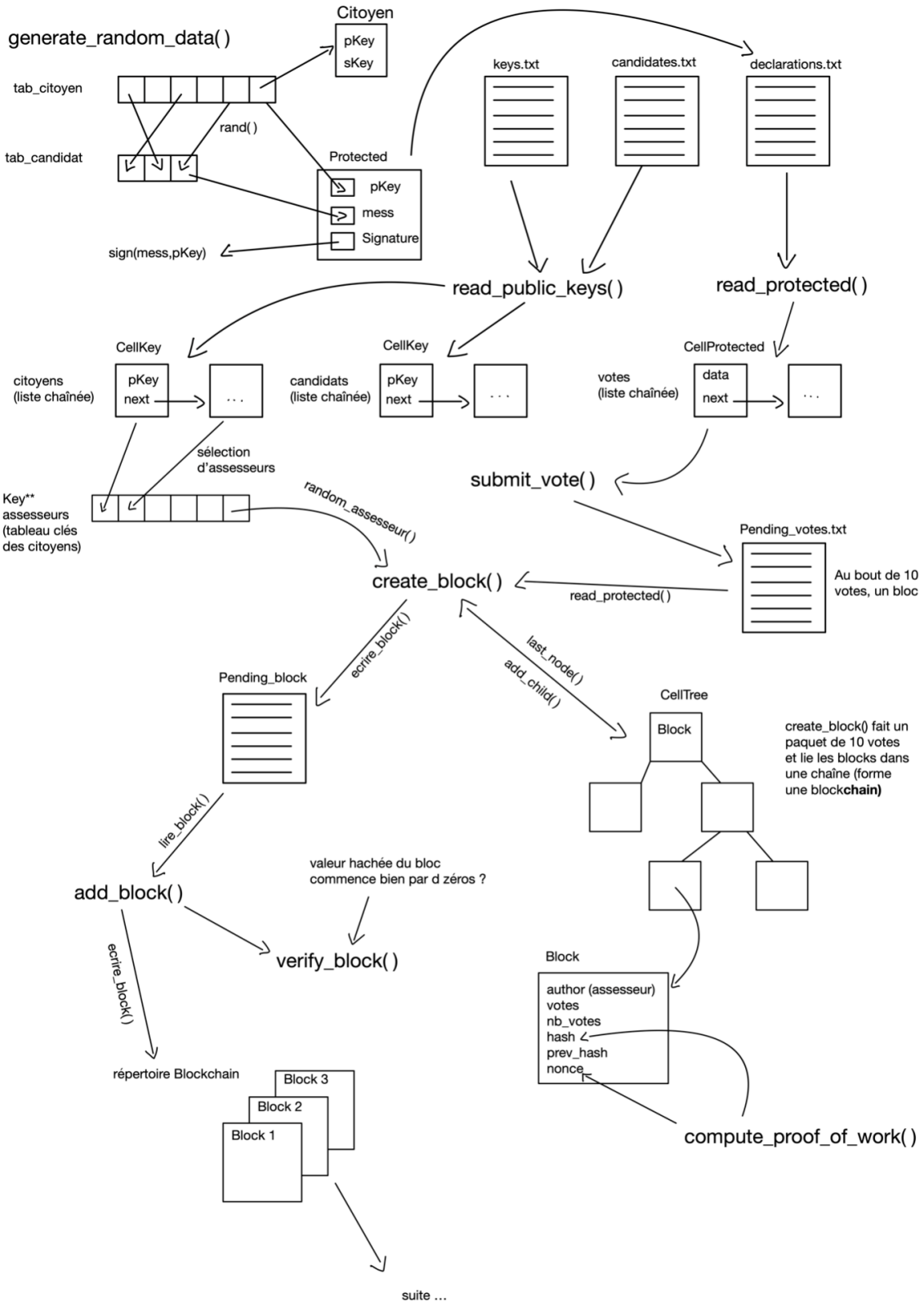


Schéma illustrant le processus de simulation d'élection (1 sur 2)

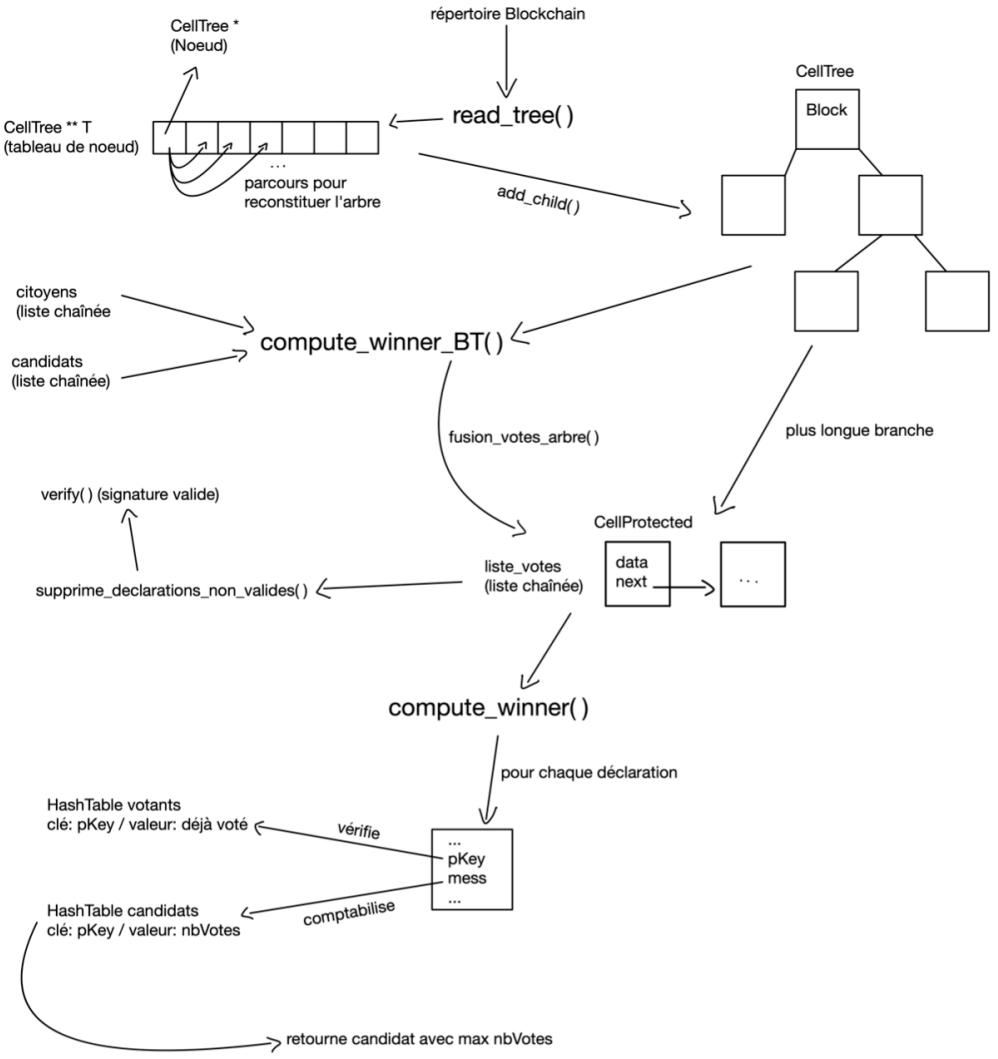


Schéma illustrant le processus de simulation d'élection (2 sur 2)

Réponses aux questions

Partie 1 : Implémentation d'outils de cryptographie

Exercice 1

Question 1.1

Pour cet algorithme, la complexité pire cas attendue est en O(p-3), on atteint cette complexité dans le cas où le nombre p passé en paramètre est impair et est premier, la boucle for va alors itérer exactement p-3 fois avant que l'algorithme ne se termine.
Si le nombre est pair ou inférieur à 2, l'algorithme se termine immédiatement, et si le nombre est impair et n'est pas premier, la boucle for se termine de manière anticipée dès qu'un entier compris entre 3 et p-1 est diviseur de p.

Question 1.2

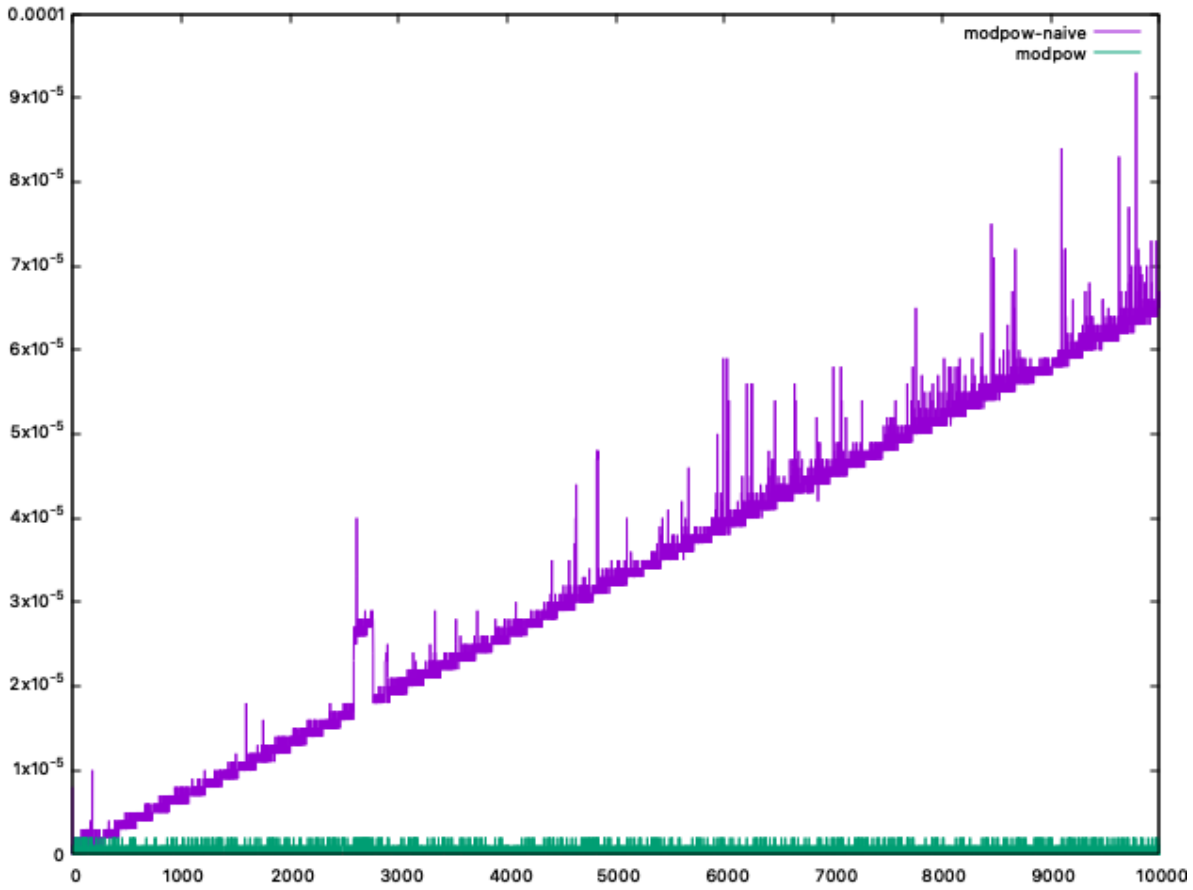
Le plus grand nombre premier calculable avec la fonction is_prime_naive en moins de 2 millièmes de secondes est 535499 sur notre machine.

Question 1.3

La complexité de modpow_naive est Θ(m), la boucle for de l'algorithme exécutant exactement m tours de boucles et le corps de boucle ne comprenant qu'une opération élémentaire.

Question 1.5

Ci-dessous, les courbes représentant de le temps d'exécution de modpow_naive (en violet) et de modpow(en vert) en fonction du paramètre m.



Les courbes reflètent parfaitement la complexité attendue pour ces deux algorithmes, en effet la courbe de modpow_naive est très clairement linéaire ce qui coïncide avec une complexité Θ(m). Tandis que la courbe de modpow, dont la complexité est en $O(\log_2(m))$, présente une croissance bien plus faible et ne décolle pas de l'axe des abscisses.

Question 1.7

Pour déterminer si un nombre p est probablement premier, on doit vérifier que k nombres aléatoires choisis entre 2 et p-1 (inclus) ne sont pas des témoins de Miller.
Si on suppose que pour tout nombre p, ¾ des nombres entre 2 et p-1 sont des témoins de Miller, la probabilité de tirer une nombre entre 2 et p-1 qui ne soit pas un témoin de Miller est donc de ¼ .
Sachant qu'on effectue k tirages, déterminer la probabilité que le test de Miller soit erroné (le test conclut que p est premier alors qu'il ne l'est pas) revient à calculer la probabilité de ne jamais tirer de témoin de Miller, ou plutôt de ne tirer que des nombres qui ne sont pas des témoins de Miller. Cette probabilité n'excédera donc pas $(\frac{1}{4})^k$.

Ce qui explique que la probabilité d'erreur du test devient rapidement faible lorsque k augmente.

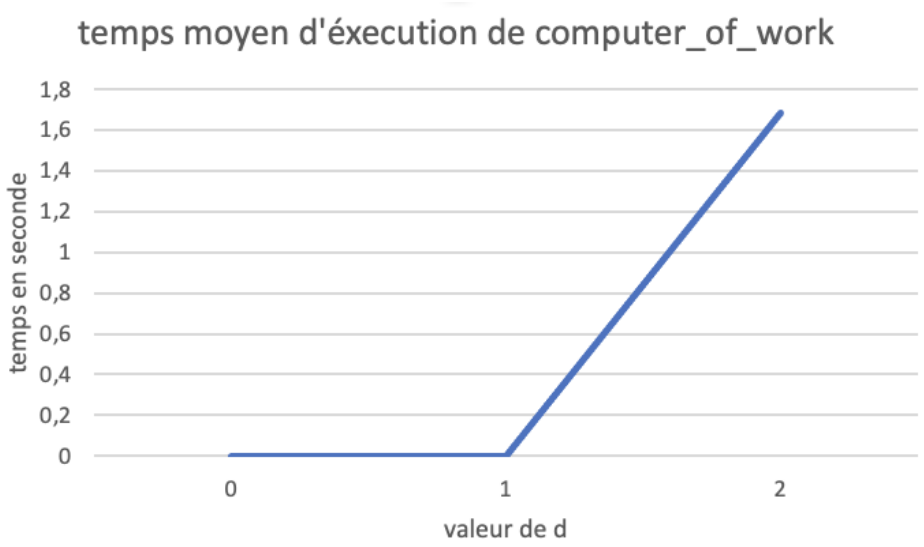
Partie 4 : Blocs et persistance des données

Exercice 7 : Structure d’un block et persistance

Question 7.8

test 1
d=0 time : 0.000036
d=1 time : 0.000022
d=2 time : 1.711841
test 2
d=0 time : 0.000037
d=1 time : 0.000026
d=2 time : 1.684302
test 3
d=0 time : 0.000027
d=1 time : 0.000033
d=2 time : 1.695638
test 4
d=0 time : 0.000042
d=1 time : 0.000024
d=2 time : 1.683484
test 5
d=0 time : 0.000025
d=1 time : 0.000073
d=2 time : 1.675269
test 6
d=0 time : 0.000053
d=1 time : 0.000021
d=2 time : 1.696974
test 7
d=0 time : 0.000025
d=1 time : 0.000026
d=2 time : 1.678683
test 8
d=0 time : 0.000030
d=1 time : 0.000022
d=2 time : 1.672596
test 9
d=0 time : 0.000026
d=1 time : 0.000031
d=2 time : 1.683565
test 10
d=0 time : 0.000033
d=1 time : 0.000034
d=2 time : 1.684901

temps moyen
d=0 : 0.0000334
d=1 : 0.0000312
d=2 : 1.6867255



A mesure que d augmente, le temps moyen d'exécution de compute_proof_of_work croît très rapidement. Sur notre machine, la barre des 1 secondes est dépassée dès qu'on fixe d à 2, c'est à dire qu'on incrémente le champ nonce du Block jusqu'à ce que la valeur hachée du Block commence par d zéros.
Si le nombre d de zéros devient conséquent, le temps de calcul moyen est considérable.
A chaque incrémentation de nonce, il est nécessaire de recréer la représentation sous forme de chaîne de caractère du Block puis de calculer la valeur hachée résultante, c'est un processus coûteux en temps de calcul.
Cela montre bien qu'un réel "travail" est nécessaire pour faire accepter un nouveau Block dans la Blockchain, ce qui limite le risque de fraude, car si suffisamment d'assesseurs (auteurs des blocks) travaillent à la création de Blocks, il est très difficile pour un assesseur isolé de continuellement être le premier à terminer la preuve de travail pour proposer une version frauduleuse de la Blockchain.

Il serait toutefois possible (si cela était souhaitable) d'optimiser le temps de calcul de notre fonction compute_proof_of_work. La fonction block_to_str qui appelée à chaque itération pourrait notamment être plus performante. Par exemple, les opérations de concaténation de chaînes de caractères sont très coûteuses, il est possible de simplifier toutes ces opérations en conservant en mémoire la taille de la chaîne en construction et d'utiliser realloc pour progressivement faire grandir la chaîne sans avoir à la reparcourir à chaque concaténation.

Exercice 8 : Structure arborescente

Question 8.8

La fonction fusion_liste_protected permet de fusionner deux listes chaînées (list1 et list2) de déclarations signées. Pour ce faire on parcourt l'ensemble de la list2, partant du principe que la list2 sera généralement plus petite que la list1 qui est la liste qui grandit à mesure qu'on fait appel à la fonction, cela limite le nombre d'éléments à parcourir.
Lors du parcours de la list2, on duplique chacun de ses éléments pour l'ajouter en tête de la list, la complexité de cette fonction est donc en O(n) où n est la taille de la list2.
Les éléments de list2 sont dupliqués afin de pouvoir libérer séparément les éléments de la liste fusionnée et ceux de l'arbre qu'on parcourt en appelant fusion_liste_protected.

La complexité de la fonction pourrait être en O(1) si l'on disposait d'un pointeur sur le premier et le dernier élément des listes chaînées que l'on cherche à fusionner. De cette façon la fusion pourrait se résumer à changer le pointeur vers l'élément suivant du dernier élément de list1 vers le premier élément de list2.
On pourrait changer la structure utilisée pour des listes circulaires doublement chaînées, ce qui permettrait une fusion en O(1).

Exercice 9 : Votes et création de bloc valide

Question 9.7

L'utilisation d'une blockchain dans le processus d'élection présente certains avantages : elle permet un niveau de traçabilité des votes inexistant dans le processus actuel. La signature quasi "inviolable" des déclarations de vote limite les possibilités de fraude sur les déclarations de vote même. On peut attribuer un vote à un citoyen, et on ne peut en changer le contenu sans en changer la signature. Tandis que le citoyen qui place son vote dans une enveloppe en papier en perd la trace à jamais, après quoi il est impossible de recompter les votes en s'assurant que chaque vote provient bien d'un citoyen autorisé à voter.
Ensuite, la décentralisation de l'information empêche, en théorie, qu'un seul acteur ait un contrôle absolu sur le processus électoral. Si l'information de l'élection est construite et partagée par l'ensemble des acteurs, la manipulation du processus électoral s'en trouve bien plus difficile.

Toutefois, la règle consistant à faire confiance à la plus longue branche de l'arbre des Blockchain, si elle limite grandement le risque de fraude, n'est pas nécessairement infaillible. On pourrait imaginer qu'un collectif suffisamment large d'assesseurs (ou disposant d'une puissance de calcul plus importante) soit capable de proposer de manière ininterrompue des Blocks frauduleux.

D'autre part, dans cette implémentation les votes ne sont pas réellement anonymes puisque dans toute déclaration la clé publique du candidat se trouve dans la même structure, et accessible à tous, que la clé publique du citoyen ayant voté pour ce candidat.
Cela constitue un risque pour le secret du vote, et donc une remise en cause de l'aspect dénocratique de l'élection.