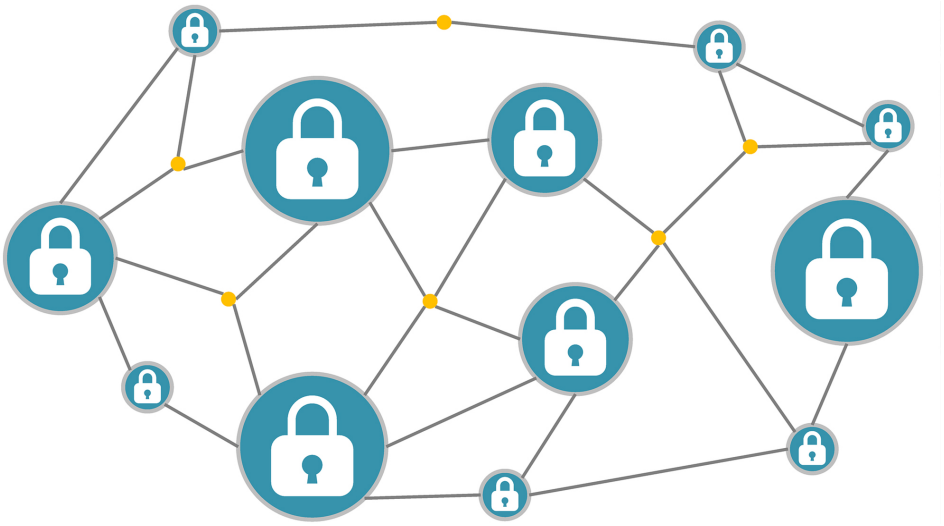
SORBONNE UNIVERSITÉ	Structures de données ✦ LU2IN006 ✦	Groupe 9 2021 / 2022

Projet :
Blockchain appliquée à un processus électoral



LIN Clément	XIAO Carine	XU Audic

Introduction :

Dans ce projet, nous considérons l'organisation d'un processus électoral par scrutin uninominal majoritaire à deux tours. La problématique étant la désignation du vainqueur.
Or depuis toujours, la tenue d'un processus électoral pose des questions de confiance et de transparence épineuses. L'objectif est donc de proposer une piste de réflexion sur les protocoles et 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.

Pour cela, nous allons procéder progressivement en cinq étapes :

- implémentation d'outils de cryptographie
- création d'un système de déclaration sécurisés par chiffrement asymétrique
- manipulation d'une base centralisée de déclarations
- implémentation d'un mécanisme de consensus
- manipulation d'une base décentralisée de déclarations

Partie I : Implémentation d'outils de cryptographie

Dans cette première partie, nous allons développer des fonctions permettant de chiffrer un message de façon asymétrique, à l'aide d'une clé publique et d'une clé privée. L'algorithme de cryptographie asymétrique que nous allons implémenter est le protocole RSA, qui s'appuie sur des nombres premiers pour la génération des clés publiques et secrètes.

1. Résolution du problème de primalité

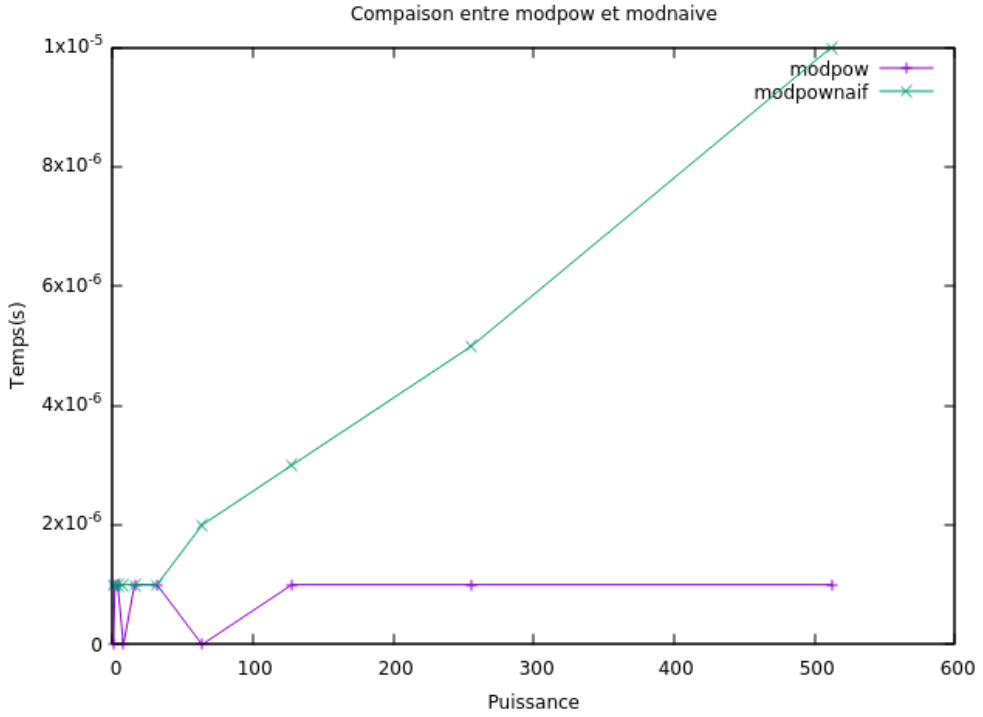
Pour générer efficacement des nombres premiers, il faut avoir un moyen d'effectuer rapidement des tests de primalité.

Nous avons dans un premier temps implémenté la fonction **int is_prime_naive(long p)** qui renvoie 1 si et seulement si aucun des entiers entre 3 et p-1 ne divise p. Sa complexité est en O(p). Avec cette fonction, le plus grand nombre premier que nous arrivons à tester en moins de 2 secondes est voisin de 296 902 367.

Mais pour le bon fonctionnement du protocole RSA, nous avons besoin d'un test de primalité plus efficace : le test probabiliste de Miller-Rabin. Pour cela, nous avons besoin de calculer efficacement une exponentiation modulaire : $a^m \bmod n$ où $a, m, n \in \mathbb{N}$.

Ainsi, nous avons implémenté la fonction **long modpow_naive(long a, long m, long n)** qui retourne la valeur de (a^m mod n) de façon naïve. Sa complexité est en O(m).
Mais nous remarquons qu'il serait plus efficace de réaliser des élévations au carré, nous avons donc implémenté la fonction **int modpow(long a, long m, long n)** qui réalise cela. Sa complexité est en $O(\log_2(m))$.

Voici un graphe qui compare les performances des fonctions (en seconde) en fonction de m.
On a fixé a = 15329813, et n = 17211.



Nous avons ensuite les fonctions : **int witness(long a, long b, long d, long p)** qui teste si a est un témoin de Millier pour p, **long rand_long(long low, long up)** qui retourne un entier **long** généré aléatoirement entre low et up inclus et **int is_prime_miller(long p, int k)** qui réalise le test de Miller-Rabin.

Néanmoins, le test de Miller-Rabin n'est pas fiable à 100% et la borne supérieure sur sa probabilité d'erreur est de 0,25. Heureusement, la probabilité d'erreur devient rapidement très faible quand k augmente. De plus, au vu de sa complexité, nous continuerons à utiliser cet algorithme plutôt que la méthode naïve dans la suite du projet.

Nous pouvons maintenant générer des nombres premiers de taille (= son nombre de bits) compris entre low_size et up_size. La fonction implémentée s'appelle **long random_prime_number(int low_size, int up_size, int k)** avec k le nombre de tests de Miller à réaliser.

2. Implémentation du protocole RSA

Comme précisé précédemment, pour utiliser le protocole RSA, nous devons générer une clé publique de la forme (s, n) qui permet de chiffrer des messages et une clé secrète de la forme (u, n) qui permet les déchiffrements.
Avec p et q deux (grands) nombres premiers distincts, la démarche à suivre est la suivante :

- calculer $n = p \times q$, et $t = (p - 1) \times (q - 1)$
- générer aléatoirement des entiers s inférieur à t jusqu'à en trouver un tel que PGCD(s, t) = 1
- déterminer u tel que $s \times u = 1$

La fonction **void generate_key_values(long p, long q, long* n, long* s)** permet de générer les clés publiques et secrètes à partir des nombres premiers p et q, en suivant le protocole RSA. Cette fonctions fait aussi appel à la fonction **long extended_gcd(long s, long t, long* u, long* v)** qui est une version récursive de l'algorithme d'Euclide étendu.

Maintenant que l'on à les clés, **long* encrypt(char* chaîne, long s, long n)** chiffre la chaîne de caractères *chaîne* à l'aide de la clé publique, et **char* decrypt(long* crypted, int size, long u, long n)** déchiffre le message crypted de taille size à l'aide de la clé secrète.

Un fichier main.c est fourni afin de réaliser les tests.

Partie II : Déclaration sécurisés par chiffrement asymétrique

Un citoyen interagit pendant les élections en effectuant des déclarations. Nous supposons ici que l'ensemble des candidats est déjà connu et que les citoyens ont juste à soumettre des déclarations de vote.

1. Manipulation de structures sécurisées

Dans notre modèle, chaque citoyen possède une carte électorale qui est définie par un couple de clé publique et secrète, qu'il utilise pour signer sa déclaration et attester de son authenticité.

Nous avons dans un premier temps déclarer les structures à utiliser :

- **Key**, qui contient deux long, représentant une clé
- **Signature**, qui contient un tableau de long et sa taille, représentant une signature
- **Protected**, qui contient une clé, un message (déclaration de vote) et une signature, représentant une déclaration signée.

Sur la structure **Key**, nous avons implémenté les fonctions suivantes :

- **void init_key(Key* key, long val, long n)** qui initialise une clé déjà allouée
- **void init_pair_keys(Key* pKey, Key* sKey, long low_size, long up_size)** qui utilise le protocole RSA pour initialiser des clés publiques et secrètes
- **char* key_to_str(Key* key)** qui permet de passer d'une clé à sa représentation sous forme de chaîne de caractères
- **Key* str_to_key(char* str)** qui permet de passer d'une chaîne de caractères de la forme (x, y) à une variable de type Key
- **int sameKey(Key* k1,Key* k2)**, qui teste l'égalité entre deux clés.

Une 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 sa clé publique. Sur la structure **Signature**, nous avons implémenté les fonctions suivantes :

- **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
- **Signature* sign(char *mess, Key* sKey)** qui crée une signature à partir du message mess et de la clé secrète
- **char* signature_to_str(Signature* sgn)** qui permet de passer d'une signature à sa représentation sous forme de chaîne de caractères
- **Signature* str_to_signature(char* str)** qui permet de passer d'une chaîne de caractères de la forme #a#b#...#z à une variable de type Signature.

Sur la structure **Protected**, nous avons implémenté les fonctions suivantes :

- **Protected* init_protected(Key* pKey, char* mess, Signature* sgn)** qui alloue et initialise la structure
- **int verify(Protected* pr)** qui vérifie que la signature contenue dans pr correspond bien au message et à la personne contenus dans pr
- **char* protected_to_str(Protected* p)** qui permet de passer d'un Protected à sa représentation sous forme de chaîne de caractères
- **Protected *str_to_protected(char* str)** qui permet de passer d'une chaîne de caractères à une variable de type Signature, la chaîne doit contenir dans l'ordre (séparé d'un espace): la clé publique de l'émetteur, son message et sa signature.

2. Création de données pour simuler le processus de vote

Afin de mettre en place le scrutin, nous avons généré 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.

Cette partie du projet est composée d'une unique fonction **void generate_random_data(int nv, int nc)** qui :

- génère nv couples de clés 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 (un clé publique par ligne)
- génère une déclaration de vote signée pour chaque citoyens (candidat choisi aléatoirement)
- crée un fichier declarations.txt contenant toutes les déclarations signées (une déclaration par ligne)

Un fichier main2.c est fourni afin de réaliser les tests.

Partie III : Base de déclarations centralisées

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.

1. Lecture et stockage des données dans des listes chaînées

On s'intéresse ici à la lecture et au stockage des données sous forme de liste (simplement) chaînées. On va donc s'intéresser aux fichiers keys.txt, candidates.txt et declarations.txt que nous pourrons générer avec la partie précédente.

Définissons d'abord les structures à utiliser :

- **CellKey**, qui contient une clé et un pointeur sur le CellKey suivant, représentant une liste chaînée de clés
- **CellProtected**, qui contient une déclaration signée et un pointeur sur le CellProtected suivant, représentant une liste chaînée de déclarations signées.

Sur la structure **CellKey**, nous avons implémenté les fonctions suivantes :

- **CellKey* create_cell_key(Key* key)**, qui alloue et initialise une cellule de clé
- **void cell_en_tete(CellKey** cKey, Key* key)**, qui ajoute une clé en tête de liste
- **CellKey* read_public_keys(FILE* f)**, qui retourne une liste chaînée de clés publiques construites à partir de keys.txt ou candidates.txt
- **void print_list_keys(CellKey* LCK)**, qui permet d'afficher une liste chaînée de clés
- **void delete_cell_key(CellKey* c)**, qui supprime une cellule de liste chaînée de clés
- **void delete_list_keys(CellKey* LCK)**, qui supprime une liste chaînée de clés
- **void delete_list_keys_vide(CellKey *LCK)**, qui supprime une liste de chaînée de clés vides.

Sur la structure **CellProtected**, nous avons implémenté les fonctions suivantes :

- **CellProtected* create_cell_protected(Protected* pr)**, qui alloue et initialise une cellule de liste chaînée
- **void ajout_protected(CellProtected** LCP, Protected* p)**, qui ajoute une déclaration signée en tête de liste
- **CellProtected* read_protected (FILE* f)**, qui retourne une liste chaînée de déclarations signées construites à partir du fichier declarations.txt
- **void print_cellProtected(CellProtected* LCP)**, qui permet d'afficher une liste chaînée de déclarations signées
- **void delete_cell_protected(CellProtected* c)**, qui supprime une cellule de liste chaînée déclarations signées
- **void delete_list_cell_protected(CellProtected *LCP)**, qui supprime entièrement une liste chaînée
- **delete_list_cell_protected_contenu(CellProtected *LCP)**, qui supprime le contenu de la liste mais pas les cellules de celle ci
- **delete_list_cell_protected_elem(CellProtected *LCP)**, qui supprime les cellules de la liste mais pas leur contenu.

2. Détermination du gagnant de l'élection

Une fois toutes les données collectées, le système commence par retirer toutes les déclarations contenant une fausse signature, c'est-à-dire les tentatives de fraudes.

Ici, nous travaillons avec les structures suivantes :

- **HashCell**, qui contient une clé et un entier
- **HashTable**, qui contient un tableau de hashCell et sa taille.

Ces deux structures nous permettent de construire des tables de hachage. Une première table qui permet de compter le nombre de votes en faveur des candidats : ses clés sont les clés publiques des candidats et les valeurs sont égales au nombre de votes comptabilisés. Une seconde table qui permet de vérifier que chaque personne votante en a le droit et qu'elle ne vote qu'une seule fois au plus : ses clés sont les clés publiques des citoyens inscrits sur la liste électorale et les valeurs sont égales à zéro (pas encore voté) ou à un (déjà voté).
Les cas de collisions sont gérées par probing linéaire.

Dans un premier temps, la fonction **void filter(CellProtected** LCP)** supprime toutes les déclarations dont la signature n'est pas valide. Ensuite, nous avons une liste de fonctions qui manipulent la table de hachage :

- **HashCell* create_hashcell(Key* key)**, qui alloue une cellule de la table de hachage et qui initialise ses champs à zéro
- **int hash_function(Key* key, int size)**, qui retourne la position d'un élément dans la table de hachage, nous avons fait le choix d'additionner les deux champs de la clé, puis de faire modulo size
- **int find_position(HashTable* t, Key* key)**, qui cherche dans la table t s'il existe un élément dont la clé publique est key
- **HashTable* create_hashtable(CellKey* keys, int size)**, qui crée et initialise une table de hachage avec keys, les collisions sont gérées par probing linéaire
- **void delete_hashtable(HashTable* t)**, qui supprime une table de hachage
- **void delete_hashtable_vide(HashTable* t)**, qui supprime une table de hachage vide.

Enfin, le travaille de la fonction **Key* compute_winner(CellProtected* decl, CellKey* candidates, CellKey* voters, int sizeC, int sizeV)** est de créer une tâche de hachage pour la liste des candidats, créer une table de hachage pour la liste des votants, et pour chaque déclaration :

- vérifier que la personne qui vote en a le droit et qu'elle n'a pas déjà voté
- vérifier que la personne sur qui porte le vote est bien un candidat
- si toutes les conditions sont vérifiées, comptabiliser le vote
- mettre à jour les deux tables de hachage.

A la fin, elle déclare le candidat gagnant !

PS : si plusieurs candidats sont ex aequo, le premier dans la liste d'hachage est élu.

Un fichier main3.c est fourni afin de réaliser les tests.

Partie IV : Implémentation d'un mécanisme de consensus

Dans la partie précédente, nous avons implémenté un système de vote centralisé. Mais 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. Pour cela, nous allons utiliser une blockchain. Le protocole se fait en trois temps :

- le vote par les citoyens
- la création de blocs par des assesseurs
- la mise à jour des données par les citoyens.

Concernant les fraudes, nous allons utiliser un mécanisme de consensus dit par proof of work, fondé sur une fonction de hachage facile à calculer mais difficile à inverser.

1. Structure d'un bloc et persistance

Intéressons nous à la gestion de block. Pour modéliser un bloc, on utilisera la structure **Block**, qui contient la clé publique de son créateur, une liste de déclaration de vote, la valeur hachée du bloc, la valeur hachée du bloc précédent, et la preuve de travail.

La fonction **void ecrireBlock(Block* b, FILE* f)** nous permet d'écrire un bloc dans un fichier. Ce fichier contient dans l'ordre :

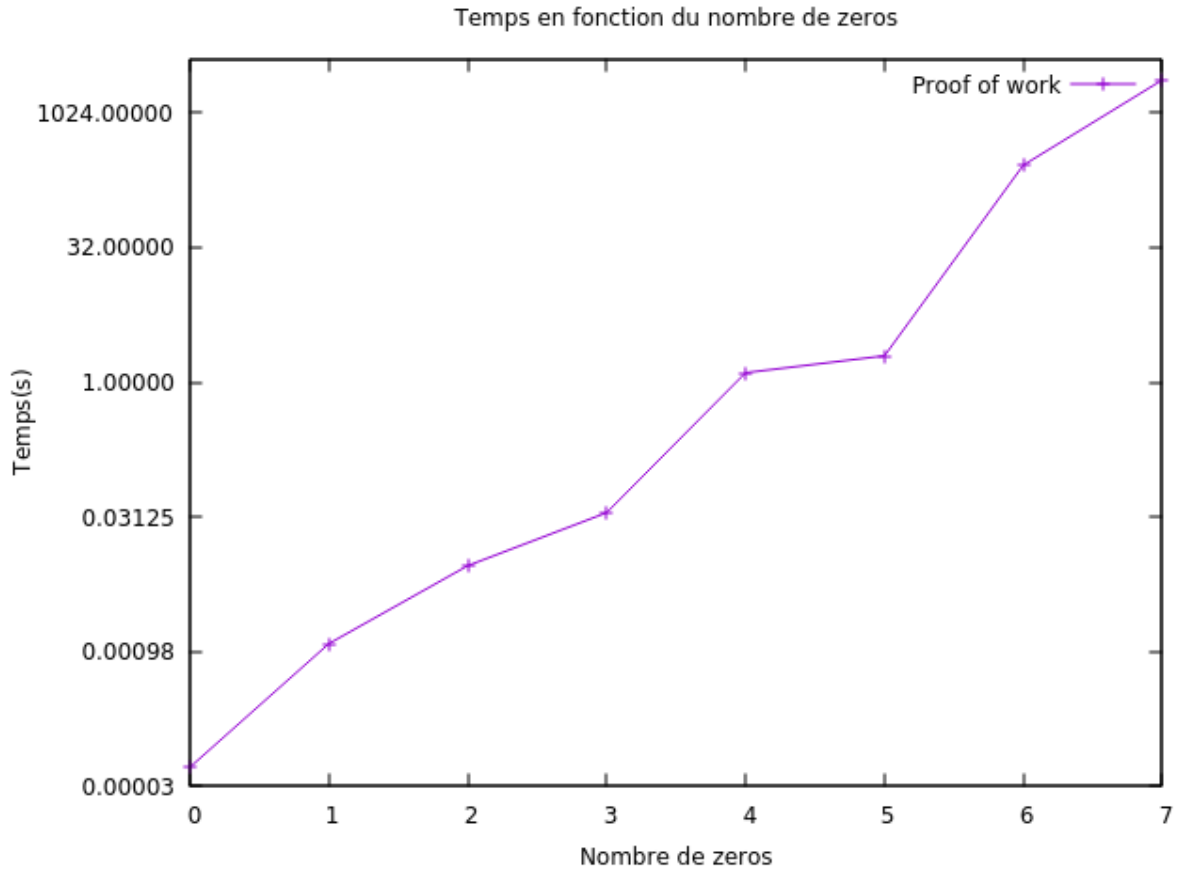
- l'auteur du bloc
- sa valeur hachée
- la valeur hachée du bloc précédent
- sa preuve de travail
- les déclarations

Tandis que la fonction **Block* lireBlock(FILE* f)** permet de faire l'opération inverse.

Ensuite, il nous faut créer des blocs valides. Un bloc ne sera considéré comme valide que si la valeur hachée de ses données commence par un certain nombre d de zéros successifs. Pour faire cela, on a d'abord la fonction **char* bloc_to_str(Block* b)** qui génère une chaîne de caractère représentant le bloc à partir de ses données. Ensuite, pour hachée une chaîne de caractère, on utilise la fonction **unsigned char* hash_SHA256(char * c)** qui fait appel à la fonction de hachage cryptographique SHA256. Et enfin, la fonction **void compute_proof_of_work(Block* b, int d)** incrémente la preuve de travail nonce, jusqu'à ce que la valeur haché de la chaîne de caractère représentant le bloc concaténé à nonce, commence par d zéros successifs.

Par mesure de précaution et pour limiter les fraudes, la fonction **int verify_block(Block* b, int d)** vérifie qu'un bloc est valide grâce à la preuve de travail.

Voici un graphe qui trace le temps moyen pour trouver nonce en fonction de la valeur d :



On voit que pour d ≥ 4 , le temps mis est supérieur à une seconde.

Un fichier main4.c est fourni afin de réaliser les tests.

Partie V : Manipulation d'une base décentralisée de déclarations

Les blockchains forment une chaîne, mais en cas de fraude, on se retrouve avec une structure d'arborescence. Dans ce cas, la règle à suivre est de faire confiance à la chaîne la plus longue en partant de la racine.

1. Structure arborescente

Pour représenter un arbre de blocs, la structure utilisée est **CellTree**, qui contient un bloc, un pointeur sur son père, un pointeur sur son premier fils, un pointeur sur son frère, et sa hauteur. La hauteur d'un arbre est comptée en nombre d'arcs.

Dans cette partie, nous avons une liste de fonction qui manipule les noeuds de l'arbre :

- **CellTree* create_node(Block* b)**, qui crée et initialise un noeud avec une hauteur égale à zéro
- **int update_height(CellTree* father, CellTree* child)**, qui met à jour la hauteur du noeud father quand l'un de ses fils a été modifié
- **void add_child(CellTree* father, CellTree* child)**, qui ajoute un fils à un noeud en mettant à jour la hauteur de tous les ascendants
- **void print_tree(CellTree* tree)**, qui affiche un arbre
- **void delete_tree(CellTree* tree)**, qui supprime l'arbre.

Comme nous l'avons dit plus haut, en cas de fraude, le protocole à suivre est de faire confiance à la chaîne la plus longue en partant de la racine de l'arbre. Et comme chaque bloc contient la valeur hachée du bloc précédent, il faut identifier efficacement le dernier bloc de cette chaîne pour pouvoir en créer de nouveaux. Pour implémenter cela, la fonction **CellTree* highest_child(CellTree* cell)** renvoie le noeud fils de cell qui a la plus grande hauteur. Et grâce à ça, **CellTree* last_node(CellTree* cell)** retourne le dernier bloc de la plus longue chaîne. Ensuite, pour rassembler les votes, on a la fonction **CellProtected* fusion_declaration(CellProtected* cell1, CellProtected* cell2)** qui fusionne deux listes.

Soit $n = |cell1|$ (= le nombre de déclarations de cell1), alors cette dernière fonction est de complexité $\Theta(n)$. Pour améliorer sa complexité en $O(1)$, il faudrait ajouter un nouveau champ dans la structure, qui serait un pointeur vers le dernier élément de la liste.

Une fois la plus longue chaîne trouvée, on rassemble les votes de celle ci avec la fonction **CellProtected* regroup_cellP(CellTree* cell)**. Manque plus que le dépouillement des votes avec la fonction **compute_winner** de la partie III.

2. Simulation du processus de vote

Pour simuler le fonctionnement d'une blockchain, on utilisera le répertoire et fichiers suivants :

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

Dans un premier temps, nous avons créé le répertoire “Blockchain”.

Nous avons la fonction **void submit_vote(Protected* p)** qui permet aux citoyens de voter, leurs votes sont ajoutés à la fin du fichier Pending_votes.txt . Ensuite, pour rassembler les votes du fichier dans un bloc, nous utilisons **void create_block(CellTree* tree, Key* author, int d)**. Après avoir créé le bloc, le fichier “Pending_votes.txt” est supprimé et le bloc est écrit dans “Pending_block”. Pour traiter ce fichier, la fonction **void add_block(int d, char* name)** vérifie d’abord que le bloc écrit dedans est valide. Si c’est le cas, elle crée un fichier appelé name qui représente le bloc et l’ajoute dans “Blockchain”. Que le bloc soit valide ou pas, “Pending_block” est supprimé.

On souhaite maintenant construire l’arbre correspondant aux blocs contenus dans le répertoire “Blockchain”. La première étape est de créer un nœud pour chaque bloc contenu dans le répertoire. Les noeuds sont stockés dans un tableau T de type CellTree**. La seconde étape est de parcourir le tableau T afin de trouver et faire les liens de parentés. La dernière étape est de parcourir une dernière fois T pour trouver le noeud dont le champ father est égal à NULL. La fonction **CellTree* read_tree()** réalise ces étapes et retourne la racine de l’arbre construit.

Il faut maintenant comptabiliser les voix de la plus longue chaîne de l’arbre et annoncer le gagnant. Comme on sait déjà obtenir la plus longue chaîne, **Key* compute_winner_BT(CellTree* tree, CellKey* candidates, CellKey* voters, int sizeC, int sizeV)** va extraire la liste des déclarations de la chaîne en question, supprimer les déclarations de vote non valides, et calculer le vainqueur.

Un fichier main5.c est fourni afin de réaliser les tests.

Pour conclure ce projet, nous dirons que l'utilisation d’une blockchain dans le cadre d’un processus de vote est un bon moyen de lutter contre l’abstention des citoyens. Néanmoins, elle présente des défauts contre les fraudes. En effet, le consensus consistant à faire confiance à la plus longue chaîne lors de la comptabilisation des votes peut présenter des failles, notamment lors de la clôture du vote : lorsque celle-ci est proche, nous avons plus de mal à distinguer les votes légitimes des frauduleux.

Partie Bonus : Manuel d'utilisation

Cette partie vous permettra d'utiliser, tester et modifier notre code sans aucun problème, à l'aide de ce manuel d'utilisation.

1. Compiler

Pour compiler les fichiers sources (.c et .h) ainsi que les exécutables, Il suffit de taper:
`make`

2. Exécuter

Les quatre premiers fichiers main n'utilisent pas d'argument. Tandis que le main5 en utilise, le premier argument correspond au nombre de votant, le 2ème est le nombre de candidats. (100 et 10 sont choisis ici à titre indicatif)

main: `./main`
main2: `./main2`
main3: `./main3`
main4: `./main4`
main5: `./main5 100 10`

3. Tracer les graphes

Nous avons deux scripts gnuplot pour tracer les graphes : Gnuplot_modpow.txt et Gnuplot_compute_proof_of_work.txt.
Voici leur méthode d'utilisation :

```
gnuplot -p <Gnuplot_modpow.txt
gnuplot -p <Gnuplot_compute_proof_of_work.txt
```

4. Nettoyage

Pour nettoyer les fichiers créés, il suffit de taper :
`make clean`