COMPTE RENDU Projet

Projet : Blockchain appliquée à un processus électoral

- LU2IN006 -

XUE Lorie - HUYNH Yok Yann

28705252 - 28707630

Groupe 10

Table des matières

— Gestion des fichiers

Pour tester les fonctions, veuillez d'abord exécuter dans le terminal dans le dossier `Projet_Xue_Huynh/projet` :

```
make all
```

Le dossier `data` servira de base centralisé, contenant les fichiers `keys.txt, declarations.txt, candidates.txt`
Le dossier `fichiers` contient les fichiers .c et .h
Le dossier `fichiers_o` contient les fichiers .o lié à la compilation
Le dossier `exec` contient les fichiers exécutables de chaque main.
Le dossier `courbetemps` contient les images des courbes du temps de calcul des algorithmes
Le dossier `test` contient des fichiers qu'on a utilisé pour certains test
Le dossier `Blockchain` contient des fichiers de `block` qu'on verra dans la Partie 5

Les fonctions de tests se feront directement depuis le fichier exécutable `main` qui affichera la liste des main et vous choisirez l'exercice dont vous voulez les tests.
Pour l'exécuter : `./main`

Lorsque vous n'aurez plus besoin des fichiers veuillez écrire dans le Terminal :

```
make clean
```

Cela videra toutes les données des dossiers `data, Blockchain, exec, fichiers_o .`

Dans ce projet, nous voulons mettre en place une simulation d'un processus électoral sécurisé.
Chaque citoyen peut déclarer sa candidature ou voter pour un des candidats.
Pour la confidentialité des citoyens, nous allons mettre en place des outils de cryptographie.

— Partie 1 : Implémentation d'outils de cryptographie.

Cette première partie est consacrée aux fonctions de chiffrement de message de façon asymétrique à l'aide du protocole RSA.
La cryptographie asymétrique 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.

1. Résolution du problème de primalité

Comme le protocole RSA que nous utilisons s'appuie sur des nombres premiers, il est nécessaire de pouvoir gérer facilement des nombres premiers.
Q 1.1. La fonction int `is_prime_naive(long p)` permet de déterminer si p est premier en sachant qu'il est impair et qu' elle a une complexité en $\Theta(p-1)$ et quand p<3 on a le meilleur cas qui est de complexité de $\Omega(1)$.

Q1.2. Le plus grand nombre que nous avons réussi à tester avec cette fonction est 131071.

Pour générer plus rapidement de très grand nombre premier, nous allons implémenter un test de primalité plus efficace : le test de primalité Miller-Rabin.

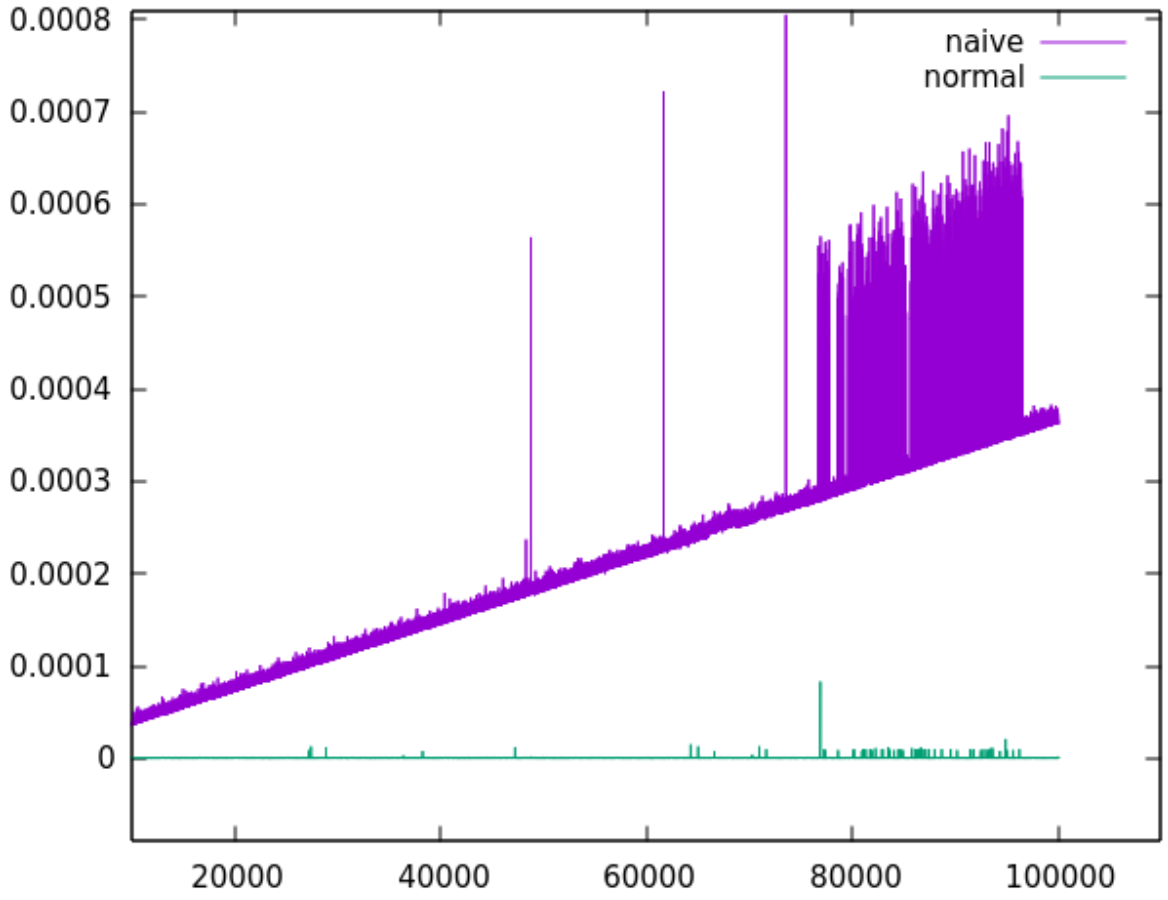
EXPONENTIATION MODULAIRE RAPIDE

Calcul de $a^m \ mod \ n$
Q1.3. `long modpow_naive(long a, long m, long n);` multiplie la valeur courante par a puis lui applique le modulo n, ceci est répété m fois. On a donc une complexité en $\Theta(m)$.

Q1.4. Pour être plus efficace, nous allons implémenter un algorithme récursif de meilleur complexité $O(log_2(m))$: `int modpow(long a, long m, long n);` qui renvoie :

- 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.

Q1.5. Observons désormais les performances des deux fonctions précédentes. Les données des performances sont récupérées dans un fichier `"mod_pow_naive.txt"`. On observe que notre deuxième algorithme qui est `modpow` est effectivement beaucoup plus rapide et efficace que `modpow_naive` .



TEST DE MILLER-RABIN

Le test de Miller-Rabin permet de déterminer la primalité d'un nombre impair p quelconque.

Soient b et d deux entiers tels que $p = 2^b d + 1$. Etant 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 p n'est pas premier mais on ne peut pas affirmer dans le cas contraire que p est premier. Mais si on répète ce test suffisamment de fois pour des valeurs aléatoires entre 1 et p-1 alors on peut dire que p est premier avec un taux de probabilité élevé.

L'énoncé nous donne les fonctions suivantes :
— `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, c'est-à-dire si p est premier pour a.
— `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).

Q1.7. Supposons que p ne soit pas premier, dans ce cas on a $\frac{1}{4}$ de chance de tomber sur un nombre premier dont p n'est pas divisible par ce nombre pour un nombre k de tests. Donc, la borne supérieure sur la probabilité d'erreur de l'algorithme : $(\frac{1}{4})^k$

GENERATION DE NOMBRES PREMIERS

`long random_prime_number(int low_size, int up_size, int k)` retourne un nombre premier de taille d'octet comprise entre low size et up size.

FONCTION DE TESTS

Parmi la liste des main dans le fichier exécutable `main` :
Choisissez l'exercice 1 en tapant 1

On vous demande un nombre que l'on va tester s'il est premier, dans les fonctions :
`is_prime_naive`
`is_prime_miller` pour 1 test et 120 tests
`is_prime_naive_temps` qui renvoie le temps d'exécution de `is_prime_naive`

Plus le nombre premier entré est à la fois grand et à la fois premier, plus le temps d'exécution sera grand, la borne sera de $O(p-1)$ qui est le pire cas.

2. Implémentation du protocole RSA

Les citoyens auront chacun 2 clés, une qui sera publique représentant son identité et une qui sera secrète représentant la confidentialité.

Les clés contiennent chacune 2 valeurs dans chaque clé, sous la forme d'un `struct` comme ci-dessous.

```
typedef struct key{
      long val;
      long n;
} Key;
```

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

`void generate_keys_values(long p, long q, long *n, long *s, long *u)` permet de générer la clé publique pkey = (s, n) et la clé secrète skey = (u, n) à l'aide de la fonction fournit : `long extended_gcd(long s, long t, long *u, long *v)` qui est une version récursive de l'algorithme d'Euclide.

Pour que les clés secrètes soient difficiles à retrouver, les clés devront respecter ses conditions:

- $n = p \times q$ et $t = (p - 1) \times (q - 1)$, où p et q sont des nombres premiers
- $s < t$ et $PGCD(s, t) = 1$
- $s \times u \, mod \, t = 1$, on trouve une valeur u

Pour la confidentialité des votes ou des messages, nous aurons besoin de chiffrer et déchiffrer leur contenu. Et pour cela, nous utiliserons le protocole RSA qui est un algorithme de cryptographie qui permet de chiffrer et déchiffrer à l'aide de clés publiques et secrètes.

CHIFFREMENT ET DÉCHIFFREMENT DE MESSAGES

Voici un schéma explicatif d'un envoi de message par le protocole RSA.

`long* encrypt(char* chaine, long s, long n);` chiffre le message m en calculant $c = m^s \, mod \, n$.

`char* decrypt(long* crypted, int size, long u, long n);` qui permet de déchiffrer c pour retrouver m en calculant $m = c^u \, mod \, n$.

Grâce à ce protocole, nous pouvons envoyer des messages.
Et l'individu pourra le décrypter uniquement avec la clé secrète de l'envoyé.

FONCTION DE TESTS

Une fonction de test nous est fournie par l'énoncé, il suffit de vérifier son bon fonctionnement et qu'il n'y ait aucune perte de mémoire lors de l'exécution du main.
Parmi la liste des main dans le fichier exécutable `main` :
Choisissez l'exercice 2 en tapant 2

Ce test nous permet de vérifier que le cryptage et décryptage des clés générés sont corrects.

— Partie 2 : Création d'un système de déclarations sécurisées par chiffrement asymétrique.

1. Manipulation de structures sécurisées

Dans cette partie, nous allons implémenter des déclarations de citoyen (vote ou candidature). 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.

Chaque citoyen possède une carte électorale, qui est définie par un couple de clés.

Dans un processus de scrutin, il faut que chaque personne puisse produire des déclarations de vote signées.

```
typedef struct signature{
     int size;
     long *content;
}Signature;
```

Les signatures sont obtenues avec la fonction `sign` , c'est le message du vote qui est la clé publique du candidat qui sera crypté avec la clé secrète du citoyen.
On peut maintenant créer des déclarations signées (données protégées).

```
typedef struct protected{
     Key *pKey; //clé publique de l'émetteur
     char *mess; //son message
     Signature *sgn; //sa signature
}
```

Dans notre contexte, une déclaration de vote consiste simplement à transmettre la clé publique du candidat sur qui porte le vote.
Pour attester de l'authenticité de la déclaration, on a une fonction `verify` qui vérifie si le décryptage de la signature correspond au message envoyé par le votant, avec la clé publique `pKey`. Donc, n'importe qui peut vérifier l'authenticité de la déclaration.
Et comme on a utilisé la clé privée pour le cryptage, il est difficile de se faire passer pour un autre citoyen.

FONCTION DE TESTS

Une fonction de test nous est fournie par l'énoncé, il suffit de vérifier son bon fonctionnement .
Parmi la liste des main dans le fichier exécutable `main` :
Choisissez l'exercice 3 en tapant 3

Ce test nous permet de vérifier le bon fonctionnement des conversions entre chaîne de caractères et des `struct(Key, Signature, Protected)` et de vérifier les déclarations.

Nous avons écrit des fonctions (`Free_Key, Free_Protected, Free_Signature`) afin de libérer l'espace mémoire des structures après leur utilisation.

Nous avons à présent tout pour représenter citoyens, candidats, et déclarations de votes. Il nous manque les données que l'on va générer.

2. Création de données pour simuler le processus de vote

Nous allons stocker nos données dans des fichiers, qui seront directement dans le dossier `data`.
`void generate_random_data(int nv, int nc);` génère des données dans les fichiers :

- keys.txt contenant tous ces couples de clés
  (un couple par ligne, ligne pair (clé publique) , ligne impair (clé secrète))
- candidates.txt contenant la clé publique de tous les candidats (une clé publique par ligne)
  Les candidats sont choisis au hasard parmi les citoyens.
- declarations.txt contenant toutes les déclarations signées
  (une déclaration par ligne sous la forme : *pKey mess sgn*).
  On supposera que tout le monde peut voter dont les candidats.

Nous avons écrit des fonctions (`ecrire_citoyen_fichier, ecrire_candidat_fichier, ecrire_declaration_fichier`) permettant d'écrire dans chacun des fichiers pour alléger la taille du code de `generate_random_data`.

FONCTION DE TESTS

Parmi la liste des main dans le fichier exécutable `main` :
Choisissez l'exercice 4 en tapant 4

Par défaut, `generate_random_data` générera 10 citoyens et 2 candidats, on vous laissera le choix de voir l'affichage du contenu des fichiers :
1- keys.txt
2- candidates.txt
3- declarations.txt

Ce test permet de vérifier que les données ont bien été générées.

— Partie 3 : Manipulation d'une base centralisée de déclarations.

Dans cette partie, nous allons utiliser les fichiers créés précédemment pour vérifier l'intégrité des données et comptabiliser les votes et récupérer l'ensemble des clés publiques des citoyens et des candidats.

1. Lecture et stockage des données dans des listes chaînées

Pour stocker dans notre programme les données des fichiers, nous allons créer des listes chaînées des clés et des déclarations.

Liste chaînée de clés

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

Liste chaînée de déclarations

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

La création d'une cellule sont créés à partir de `create_cell_key`, `create_cell_protected` et ajouté dans la liste avec `ajout_tete_cell_key`, `ajout_tete_cell_protected`.

`read_public_keys` et `read_protected` lisent les fichiers "keys.txt" et "declarations.txt" . Ils reprennent les fonctions des listes chaînées, et les conversions de la partie précédente.

FONCTION DE TESTS

Parmi la liste des main dans le fichier exécutable `main` :
Choisissez l'exercice 5 en tapant 5

Pour vérifier que la lecture de fichier c'est bien passé, nous avons les fonctions `print_list_keys, print_list_protected` qui affiche les valeurs de la liste chaînée.

Ce test permet de vérifier le bon fonctionnement de la lecture de fichier, et de la création des listes chaînées

2. Détermination du gagnant de l'élection

Avant de déterminer le gagnant de l'élection, il faut retirer toutes les tentatives de fraude. Toutes les déclarations dont la signature n'est pas valide seront supprimées grâce à la fonction : `verify_cell_protected`.

Nous allons avoir besoin de tables de hachage pour les candidats et les électeurs pour retrouver rapidement les différents citoyens ou candidats.

```
Typedef struct hashtable{
      HashCell** tab;
      Int size;
} HashTable;
```

Le tableau est composé de cellules de clé : la clé publique (candidat ou votant) et une valeur. Cette valeur correspond au nombre de voix d'un candidat ou si un votant a déjà voté (1 : voté, 0 : pas voté).
Par défaut, lors de la création de la cellule, `val` vaut 0.

```
Typedef struct hashcell{
      Key* key;
      int val;
} Hashcell;
```

La table de hachage est géré de sorte que la clé, ou l'indice de la table est $val \times n \,\%\, size$ (donnée par la fonction `hash_function`) , où val et n sont la valeur de la clé, et $size$ est la taille de la table.
Les collisions sont gérées par probing linéaire à l'aide de `find_position` qui renvoie la position où devrait être la clé publique entré en paramètre, elle sera utilisée par la fonction `ajout_hashtable` qui ajoute une clé publique dans la table de hachage.

On peut à présent créer des tables de hachages directement avec la taille et une liste chaînée de clés qui sera ajoutée au fur et à mesure dans la fonction `create_hashtable`.

Quelques fonctions ont été ajoutés en plus de celles demandées :

`int est_dans_table(HashTable *t, Key *k);` // renvoie 1 si k(Key) existe dans la table de hachage sinon 0
`Key *le_plus_vote(HashTable *h);` // renvoie le candidat qui a le plus de vote parmi ceux de la table
`void delete_hashtable(HashTable *t);`

Avec toutes ces fonctions, nous pouvons compter les votes dans `compute_winner` qui nous renverra la clé du candidat gagnant, en plusieurs étapes : *(les déclarations sont supposés vérifier avant d'entrer dans la fonction)*

1. Création de table hachage des votants, et des candidats
2. Comptage des votes
   Si le votant n'a pas voté, alors si son message de vote fait bien parti des candidats, on ajoute +1 dans le nombre de vote du candidat.
3. Vérifie qui est le candidat qui a le plus grand nombre de votes
4. Supprime les tables de hachage et renvoie le candidat gagnant

FONCTION DE TESTS

Parmi la liste des main dans le fichier exécutable `main` :
Choisissez l'exercice 6 en tapant 6

Premièrement on teste la fonction qui vérifie les déclarations en les affichant avant et après vérification.
Puis on passe à la simulation d'une élection avec 500 citoyens et 5 candidats (générés avec `generate_random_data`) en s'aidant de `compute_winner` et on affiche le gagnant.

— Partie 4 : Implantation d'un mécanisme de consensus.

Dans le reste du projet, nous allons implémenter un système de vote décentralisé en utilisant la blockchain. Pour rendre la fraude difficile, on va utiliser un mécanisme de consensus dit par proof of work (preuve de travail). Nous allons utiliser la fonction de hachage cryptographique SHA-256.

1. Structure d'un block et persistance

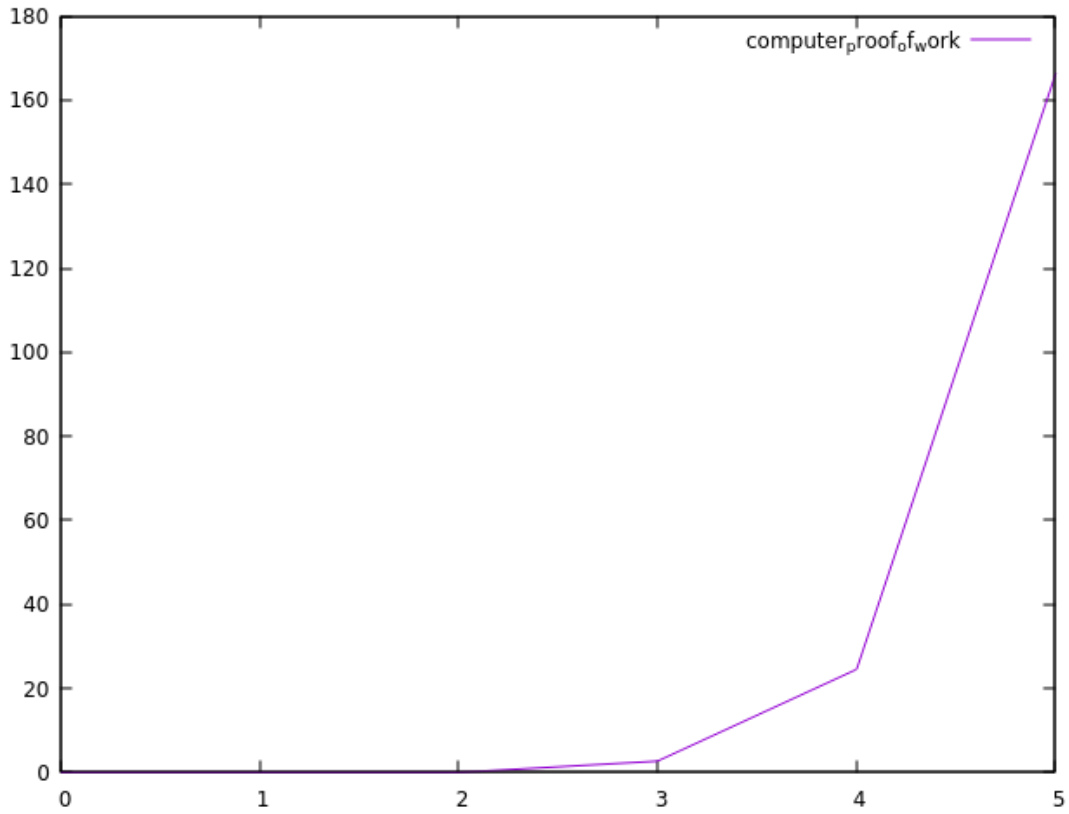
```
Typedef struct block {
     Key* author;
     CellProtected* votes;
     unsigned  char* hash;
     unsigned  char* previous_hash;
     int nonce;
} Block;
```

Ainsi, un bloc contiendra :

- La clé publique de son créateur (l'auteur est un citoyen volontaire)
- Une liste de déclarations de vote.
- La valeur hachée du bloc.
- La valeur hachée du bloc précédent.
- Une preuve de travail (vérifie que le créateur du block a fait beaucoup de calcul)

`compute_proof_of_work` permet d'affecter une valeur `nonce` et `hash` dans le block, `hash` est donnée par conversion du block en chaîne de caractères (`block_to_str`), puis ensuite donnée par notre fonction de hachage (`str_to_SHA256`), et `nonce` est incrémenté à chaque nouvelle valeur de `hash`, tant que `hash` n'a pas le nombre de zéro `d` demandé en paramètre.

7.8 Pour étudier le temps moyen de la fonction `compute_proof_of_work`, nous utilisons la fonction `ecrire_temps_fichier` qui écrira les données dans le fichier `proof_algo.txt`.



On remarque que l'on a une courbe exponentielle, donc plus le nombre de zéro demandé est élevé, plus le temps sera long. Cela permet de créer des intervalles de temps entre chaque block.

Pour vérifier que le block créée est bien valide, on a la fonction `verify_block` qui vérifie que `hash` commence bien par `d` zéros.

On utilisera prochainement ces block comme données, donc on aura les fonction `ecrire_fichier_block` et `lecture_fichier_block`.

FONCTION DE TESTS

Parmi la liste des main dans le fichier exécutable `main` :
Choisissez l'exercice 7 en tapant 7

Ce test permet de vérifier le bon fonctionnement de la fonction de hachage.
Puis on initialise un bloc et on obtient sa preuve de travail avec d=1 et on vérifie la validité du bloc.

On écrit le bloc créé précédemment dans le fichier “block_test.txt” et on le récupère avec la fonction `lecture_fichier_block` pour vérifier si les données sont cohérentes, il affiche toutes les erreurs, s’il y a une incohérence.
Ensuite nous affichons le contenu du fichier et vérifions qu’il a le format :

```
author hash previous_hash nonce
Déclaration (1 par ligne)
```

2. Structure arborescente

Pour plus de sécurité, nous allons utiliser une structure arborescente où leur niveau est déterminé par leur valeur hachée. C’est-à-dire, le nœud aura dans `previous_hash` de son block la valeur hachée du père.

```
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;
```

Pour la création d’un noeud de l’arbre :

- `create_node`

Pour la mise à jour de l’arbre, nous avons :

- `update_height   //met à jour la taille de l’arbre`
- `add_child       // ajoute un noeud fils à l’arbre`

Pour retrouver un noeud d’un arbre :

- `highest_child   //renvoie le plus grand des fils du noeud`
- `last_node       //renvoie le dernier noeud de l’arbre`

8.8 Pour la fusion des 2 listes de déclarations, on a nommé la fonction `fusion_LDP` qui prend en paramètre 2 listes chaînées de déclarations.

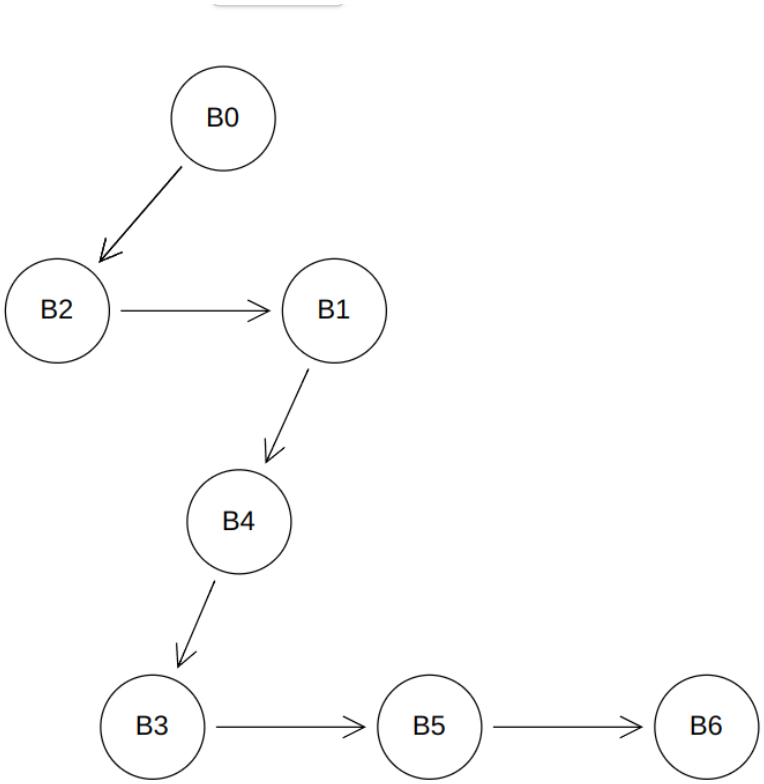
Soit n2 la taille de la 2ème liste chaînée. Cette fonction boucle au maximum n2 fois.
A chaque tour de boucle, on fait un appel à `ajout_tete_cell_protected` qui a une complexité en O(1). Donc cette algorithme a une complexité en O(n2).
Pour avoir une fusion en O(1), on aurait eu besoin d’un pointeur vers le dernier élément de la liste 1.

Pour la fusion des 2 listes de déclarations, on a nommé la fonction `decl_longue_chaine` qui prend en paramètre un arbre.
Elle fusionne les listes de la plus longue chaîne de l'arbre c'est-à-dire des fils ayant la plus grande hauteur et fait donc appel à `highest_child` et `fusion_LDP`.

FONCTION DE TESTS

Parmi la liste des main dans le fichier exécutable `main` :
Choisissez l'exercice 8 en tapant 8

On teste les fonctions `highest_child`, `last_node` et `decl_longue_chaine` avec cet arbre.



— Partie 5 : Manipulation d'une base décentralisée de déclarations.

Dans cette partie, nous allons simuler les soumissions de votes par les citoyens et les créations de blocs valides par des assesseurs. 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.

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.

La soumission des votes se fait par le biais de `submit_vote` qui ajoute la déclaration à la suite du fichier `Pending_votes.txt`.
Ensuite, la création du block se fera par la fonction `create_block` qui lira le fichier `Pending_votes.txt` et le supprimera, et on ajoutera le block dans le fichier `Pending_block`.
Le block sera ajouté dans le dossier `Blockchain` sous le nom de BlockNum.txt, où Num est le numéro du block. On l'ajoutera avec la fonction `add_block`.

Pour ensuite récupérer toutes ces données nous avons la fonction `read_tree` qui lit tous les fichiers de Blockchain, et renvoie un arbre.

Pour déterminer le gagnant de l'élection, nous avons la fonction `compute_winner_BT` qui prend en paramètre un arbre, la liste des candidats et des votants, et leurs tailles respectives.
Il fait exactement la même chose que `compute_winner` sauf qu'on passe par un arbre.
Il prend la plus longue chaîne de l'arbre avec `decl_longue_chaine`, car on suppose que c'est le plus fiable parmi toutes les branches.
Ensuite on vérifie les déclarations à l'aide de `verify_cell_protected`.
Nous réutilisons `compute_winner` et on renvoie le gagnant.

FONCTION DE TESTS

Parmi la liste des main dans le fichier exécutable `main` :
Choisissez l'exercice 9 en tapant 9

Ce test permet d'exécuter notre simulation finale du processus électoral avec une blockchain.

1. Génération de 1000 citoyens et de 5 avec `generate_random_data`
2. Lecture des déclarations de vote, des candidats, et des citoyens (`read_protected`, `read_keys`)
3. Soumission des votes (`submit_vote`) avec la création de block valide tous les 10 votes (`create_block`), suivi de son ajout dans la blockchain (`add_block`)
4. Lecture (`read_tree`) et de son affichage (`print_tree`)
5. Calcul et affichage du gagnant à l'aide de `compute_winner_BT`

Conclusion :

Si on veut mettre en place un processus de vote basé sur une blockchain, il faut s'assurer que son nonce(nombre de bits à zéro) soit grand. Cependant, le temps nécessaire à l'exécution du programme grandit exponentiellement avec le nombre de zéros choisit. De plus, plus il y aura de votes, plus le temps sera important aussi.
On a supposé que l'on fait confiance à la plus grande chaîne, cependant, il est possible qu'un individu puisse envoyer plusieurs block qui se suivent, alors la sécurité est percée. Donc dans une vraie élection, il est préférable de ne pas se baser sur la plus grande chaîne.