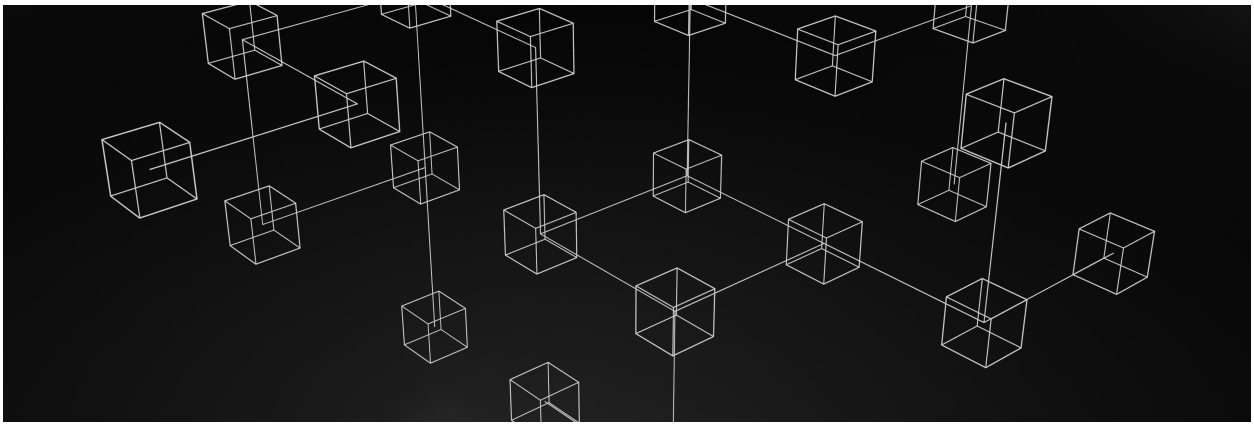


Structures de données: Blockchain appliquée à un processus éléctoral

Jules de Richemont et Léa Cohen-Solal

Dirigé par Nawal Benabbou

LU2IN006 Sorbonne Université

Introduction

En juin 2021 le taux d'abstention aux élections régionales a atteint les 66,7%. Une première historique. Face à cette hausse de l'abstention d'autres façons de voter méritent d'être étudiées. Le but de ce projet est d'étudier la manipulation des structures de données en vue d'une élection au suffrage universel direct par scrutin uninominal majoritaire, caractéristique notamment des élections présidentielles françaises. Ces manipulations devront permettre l'intégrité, la sécurité et la transparence de l'élection pour représenter une piste d'alternative au type de suffrage acutellement en vigueur.

Les fonctions et fonctions de test codées sont réparties dans des dossiers par numéro d'exercice. Pour rendre le code des exercices plus lisibles, nous avons crée pour chaque partie des fichiers *.h* et des fichiers *.c* . La compilation est facilitée par l'utilisation d'un *Makefile*.

Table des matières

1. Développement d’outils cryptographiques

1.1. Résolution des problèmes de primalité

1.1.1. Méthode naïve et exponentiation rapide

Dans la première partie nous allons déterminer de différentes manières si un nombre est premier ou pas. Étudions ces différentes fonctions :

```
int is_prime_naive(long p)
```

→ Cette fonction teste si tous les nombres entre 3 et p-1 divisent p. Elle est donc en complexité $O(p)$. S’il existe i ∈ { 3,...,p-1} tel que i divise p alors p n’est pas premier. On augmente i de 2 à chaque itération car nous avons déjà préalablement vérifié que p n’est pas pair, et donc pas divisible par un nombre pair.

Nous remarquons que dès qu’un nombre dépasse les 11 chiffres, le temps d’exécution de la fonction dépasse les 10 secondes alors qu’il n’était qu’à 2 secondes pour un nombre à 9 chiffres. A partir de 11 chiffres le temps d’exécution atteint 55 secondes et on peut donc conjecturer qu’il croit de manière exponentielle. Il est donc nécessaire d’implémenter une fonction plus efficace.

```
long modpow_naive(long a, long m, long n)
long  modpow(long a, long  m, long  n)
```

→ Ces deux fonctions renvoient toutes les deux $a^m \mod (n)$, l’une de manière itérative et l’autre de manière recursive. En terme de compléxités, `modpow_naive` est en $O(m)$ et `modpow` est en $O(log2(n))$.

1.1.2. Test de Miller-Rabin

```
int is_prime_miller(long p, int k)
```

Comme expliqué dans le sujet, le test de Miller-Rabin va déterminer si un nombre n’est pas premier. Cet algorithme génère plusieurs valeurs au hasard en testant si celles ci sont des “témoins de Miller”. Si dans ces valeurs aléatoires une est témoin de Miller alors le nombre est premier, sinon le nombre n’est ***probablement*** pas premier.

L’avantage de ce test est que sa complexité est intéressante: en $O(log(n)^3)$ pour le test d’un témoin en exponentiation rapide, et en $O(k \times log(n)^3)$ pour la fonction. Son inconvénient est qu’il a un certain pourcentage d’erreur. En effet il peut déclarer qu’un nombre est premier alors qu’il ne l’est pas (pas de témoin de Miller trouvé). La probabilité de cette erreur est de $\frac{1}{4}$ pour chaque valeur testée. Si p nest pas premier et

que k=1, la probabilité que la valeur tirée ne soit pas un témoin de Miller est de $\frac{1}{4}$. Si l'on répète cette expérience k fois, la probabilité de se tromper devient de plus en plus faible et sur k experiences la probabilité d'erreur est de $\frac{1}{4^k}$.

Comparaison des deux méthodes

Le temps d'exécution d'une fonction est un paramètre très important pour déterminer son efficacité. Pour un nombre x pour lequel il faut determiner s'il est premier ou non, nous avons un temps d'execution assez different entre les deux fonctions. Les courbes violettes correspondent aux temps de la fonction `is_prime_naive` et les courbes vertes à ceux de `is_prime_miller`. Ces courbes sont plutôt cohérentes car la fonction `is_prime_naive(n)` teste si tous les nombres entre *3* et *n-1* divisent *n*, alors que la fonction de miller en teste au maximum *k* < *n*. On observe en outre que `is_prime_naive` est plus efficace en deçà de 70000 itérations de boucle, et que `is_prime_miller` est plus efficace au delà de 85000 itérations.

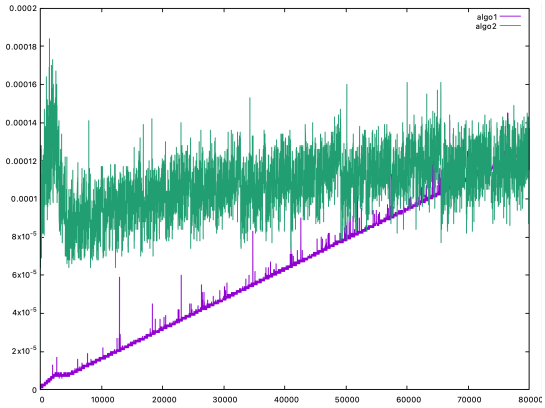


Figure : Point d'intersection des courbes de temps des fonctions prime_miller et prime_naive

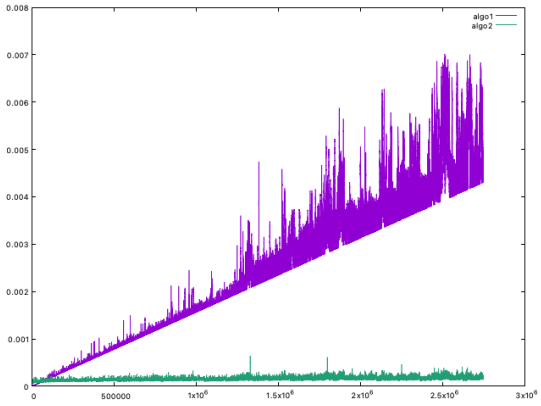


Figure : courbes des temps des fonctions prime_miller (vert) et prime_naive (violet)

1.1.3. Génération de nombres premiers

```
long random_prime_number(int low_size, int up_size, int k)
```

Principe de la fonction :

1. Générer un nombre aléatoire *x* de taille *t* telle que *low_size* < *t* < *up_size*
2. Executer la fonction `is_prime_miller(long x, int k)`
3. Répéter ce processus tant que l'on a pas trouvé un nombre premier.

1.2. Implémentation du processus RSA

1.2.1. Génération d'une paire (clé publique, clé privée)

```
long extended_gcd ( long s , long t , long *u , long * v );
void generate_key_values(long p, long q, long* n, long* s, long* u)
```

→ Cette fonction va générer la clé publique pKey et la clé secrète skey en suivant le protocole RSA expliqué dans le sujet, qui est fondé sur la difficulté de factoriser de grands entiers.

Dans ce protocole on se donne deux nombres premiers générés aléatoirement, puis on détermine $s = p \times q$ et $t = (p-1) \times (s-1)$ tels que pour $s \in \mathbb{N}$ et $u \in \mathbb{N}$, ont ait $PGCD(s,t) = 1$ et $(s \times u) \mod (t) = 1$.

Cela nous permet alors de générer une clé secrète (u, n) et une clé publique(s, n).

1.2.2. Chiffrement et déchiffrement de messages

```
long* encrypt(char* chaine, long s, long n)
char* decrypt(long* crypted, int size, long u, long n)
```

→ La fonction encrypt va chiffrer la chaîne de caractère *chaine* grâce à la clé publique *(s,n)*. Nous créons un `long*res` et nous procédons de la façon suivante :

- On sélectionne une lettre de la chaine *chaine*.
- On ajoute `modpow(lettre, s, n)` à la case de *res* correspondante ($i^{ème}$ case pour la $i^{ème}$ itération).
- On répète ces deux instructions en boucle jusqu'à arriver à la fin de la chaîne *chaine*.

Nous allons donc obtenir un tableau de *long* correspondant à notre chaîne chiffrée.

→ Inversement la fonction decrypt va dechiffrer le message par l'intermédiaire de la clé secrète (u, n). Nous allons créer une chaîne de caractères *chaine* et nous procédons de la façon suivante :

- On sélectionne une case du tableau *crypted*.
- On ajoute `modpow(case, u, n)` à la chaîne.
- On répète ces deux instructions jusqu'à arriver à la fin du tableau.

C'est ici que nous voyons l'utilité du système de la clés privée (secrète) et publique. En effet un message ne peut pas être déchiffré sans la clé secrète qui est normalement détenu par une seule personne, personne qui est la principale concernée par le message en question.

2. Système de déclarations sécurisées

Pour le modèle d'élection que nous avons choisis, nous avons donc pour chaque électeur, une clé publique et une clé secrète. Il s'agirait maintenant de définir la "déclaration" et la signature de celle ci grâce aux clés publique et privée.

2.1. Manipulation de structures sécurisées

Nous avons donc defini la structure de la clé *Key*, de la signature *Signature,* et d'une déclaration signée *Protected.*

Ces structures s'accompagnent de plusieurs fonctions classiques d'initialisation, d'allocation, d'ajout... Toutes sont amplement détaillées dans le sujet et ne seront donc pas ré-explicitées ici.

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

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

typedef struct protected {
    Key * pKey;
    char * mess;
    Signature* sign;
}Protected;
```

Figure : structures Key, Signature et Protected

2.2. Création de données pour simuler le processus de vote

Pour pouvoir mettre en place ce processus de vote par l'intermédiaire des clés publiques et privées, nous allons tout d'abord générer pour chaque citoyen une carte électorale unique contenant sa clé publique et sa clé secrète.

```
void generate_random_data(int nv, int nc);
```

→ Cette fonction va générer des citoyens aléatoirement en stockant dans différents fichiers leurs clés et déclarations.

La complexité de cette fonction croit de manière exponentielle et son temps d'exécution devient très vite non négligeable lorsque le nombre de citoyens augmente.

En effet en augmentant progressivement le nombre de citoyens de la fonction, et en notant le temps d'exécution de la fonction à chaque fois on obtient la courbe suivante :

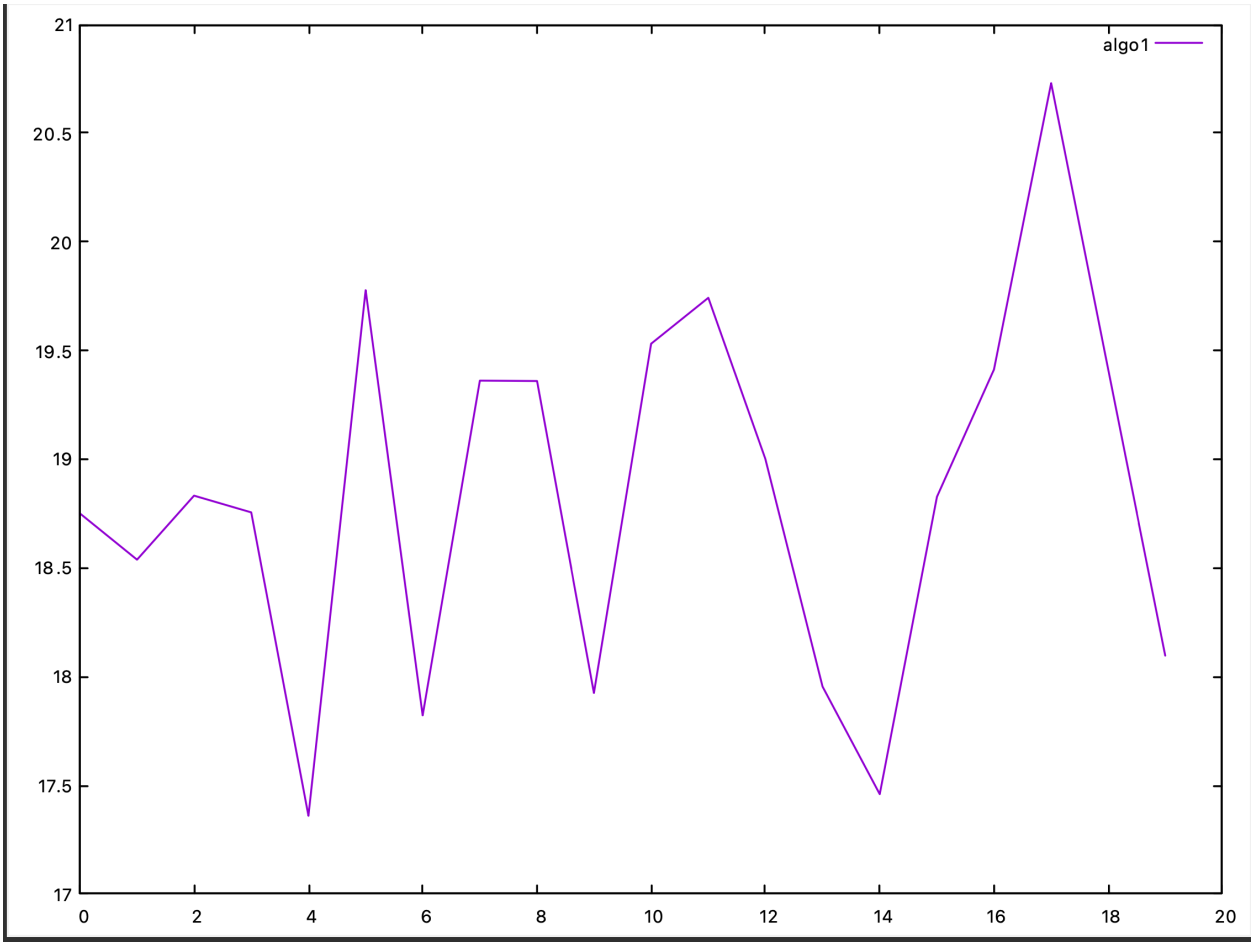


Figure : courbe du temps d'exécution de la fonction generate_random_data Exercice 4

3. Base de déclarations centralisée

3.1. Lecture et stockage des données dans des listes chaînées

3.1.1. Listes chaînées de clés et de déclarations signées

Dans cette partie nous avons crée une structure *CellKey* (liste chaînée de clés) et *Cellprotected* (liste chaînée de déclarations) et implémenté des fonctions permettant de lire des clés/déclarations dans un fichier et de les stocker dans une liste chaînée et inversement.

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


typedef struct cellProtected{
    Protected *data;
    struct cellProtected *next;

}CellProtected;
```

Figure : structures CellKey, CellProtected

3.2. Détermination du gagnant de l’élection

Lors d’une élection de manière informatisée certaines fraude peuvent se produire :

- Une personne qui veut voter plusieurs fois
- Une personne qui vote avec une fausse signature

Nous avons donc mis en place des structures permettant d’éviter ces fraudes.

- Une table de hachage *HashCell* qui va permettre de s’assurer que la personne est réellement un candidat et compter le nombre de votes par candidat.
- Une table de hachage *HashTable* qui verifie que l’électeur est inscrit sur les listes électorales et qu’il ne vote qu’une seule fois.

```
int hash_function (Key *key, int size);
```

→ Cette fonction va déterminer la position d'un élément dans la table de hachage. Cette fonction informatique admet de nombreuses variantes associées chacune à une fonction numérique de hachage h distincte. Les meilleures fonctions de hachage sont celles pour lesquelles le calcul de $h(k)$ est rapide et qui minimisent au maximum les collisions, i.e. la probabilité qu'on ait $h(k_1) = h(k_2)\ pour\ k_1 \neq k_2$.

```
int find_position(HashTable* t, Key* key);
```

→ Cette fonction va placer la clé *key* dans la tableau de hachage *t* et sa position sera trouvée par l'intermédiaire de la fonction `hash_function`. Comme nous l'avons énoncé plus tôt il est essentiel de minimiser les risques de collision. Cependant il est impossible d'éviter les collisions totalement, et les deux méthodes de résolution les plus classiques sont par chaînage ou adressage ouvert. Nous avons opté pour une table de hachage avec adressage ouvert et probing linéaire:

$h(k,i) = (h'(k) + i)\%m$ où $h' : U \to \{0, ..., m-1\}$ avec U l'univers des clés et m la taille du tableau

```
HashTable* create_hashtable(CellKey* keys, int size);
```

→ Cette fonction va utiliser les 2 autres fonctions définies précédemment pour créer une table de hachage et y placer la clé passée en paramètre.

```
Key *compute_winner(CellProtected *decl,CellKey *candidates,CellKey *voters,int sizeC, int sizeV)  ;
```

→ Enfin cette fonction est l'aboutissement du mécanisme de consensus, elle permet de calculer le candidat qui a obtenu un nombre de vote majoritaire. Elle s'appuie pour cela sur le champ *val* de la structure *HashCell* qui compte permet aussi bien de savoir si un citoyen a voté que de compter le nombre de votes pour un candidat.

4. Blocs et persistance des données

4.1. Structure d'un bloc et persistance

Un certain nombre de techniques sont mises en place pour detecter les fraudes. Parmi elles, la ***proof of work***.

Nous allons créer la structure *Block* qui va permettre de stocker une liste de déclarations de vote, ainsi que la clé publique de son créateur, sa valeur hachée et la valeur hachée du bloc qui le précède. Pour éviter les fraudes on va demander au créateur du block d'inverser partiellement une fonction de hachage

cryptographique de sorte qu'il trouve un entier *nonce* commençant par un certain nombre de zéros successifs.

En testant cette fonction implémentée plus tard nous voyons bien que le temps nécessaire croit de maniere exponentielle si on augmente le nombre de 0 requis. Un bloc ne sera considéré comme valide que s'il a une *preuve de travail*, c'est-à-dire si la valeur hachée de ses données commence par *d* zéro successifs.

4.1.1. Lecture et écriture de blocs

Nous avons tout d'abord créé deux fonctions permettant d'écrire dans un fichier un bloc et inversement:

```
void write_block (char* filename, Block* b);
Block* read_block (char* filename);
```

4.1.2. Création de blocs valides

Pour utiliser la technique de ***proof of work*** nous avons besoin de convertir notre bloc en chaîne de caractères. Pour cela nous avons implémenté la fonction `block_to_str`. Elle nous servira pour trouver la valeur hachée des données d'un bloc.

```
char* block_to_str (Block* block);
```

→ Pour transformer une chaîne de caractères en sa valeur hachée nous allons utiliser la bibliothèque *openssl*.

```
unsigned char* hash(char* s);
```

→ Cette fonction fait appel à la fonction `SHA256` de la bibliothèque *<openssl/sha.h>* qui fournit la valeur hachée du bloc associé à *s* par `block_to_str` en hexadécimal.

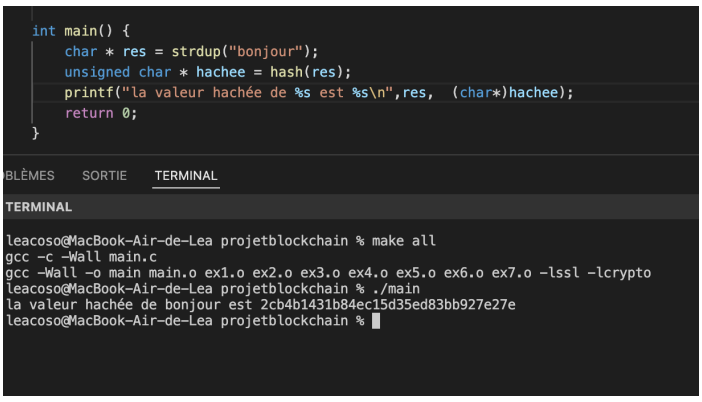


Figure : Jeu de tests sur la fonction *hash* exercice 7

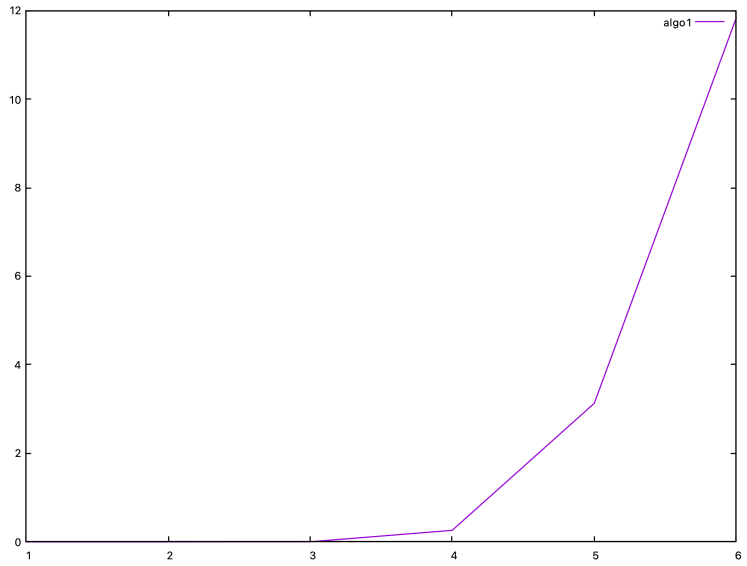
Par exemple la valeur hachée de “Bonjour” est “2cb4b1431b84ec15d35ed83bb927e27e”

```
void compute_proof_of_work (Block* b,int d );
```

→ Cette fonction va initialiser *nonce* du bloc b à 0 et va réaliser les instructions suivantes :

- hacher la chaîne de caractère `block_to_str(b)`
- vérifier si cette valeur hachée commence par *d* zéros successifs :
  - Si oui, la fonction se termine
  - Sinon, on implémente *nonce* de 1
- Répéter ces instructions jusqu’à sortir de la fonction

Nous voyons bien que plus la valeur de d augmente plus le temps d’exécution de cette fonction augmente, il augmente de manière exponentielle. Nous pouvons le constater avec cette courbe :



Courbe de temps de la fonction *compute_proof_of_work exercice 7*

A partir de *d=3* , le temps d’exécution augmente drastiquement car la complexité de `compute_proof_of_work` est exponentielle. Ce temps est signe de *preuve de travail.* En effet il qui permet d’eviter les fraudes puisqu’il devient impossible sur la durée pour un individu mal intentionné de falsifier des blocs, car il aura besoin d’une puissance de calcul trop importante pour rivaliser avec les vérifications du reste du reseau. D’où l’adage “*À chaque instant , il faut faire confiance à la plus longue chaîne de blocs reçue”*

```
int verify_block(Block *b, int d);
```

→ Cette fonction vérifie simplement si le bloc est valide ou pas en testant si on a bien *d* zéros successifs au début de la valeur hachée du bloc.

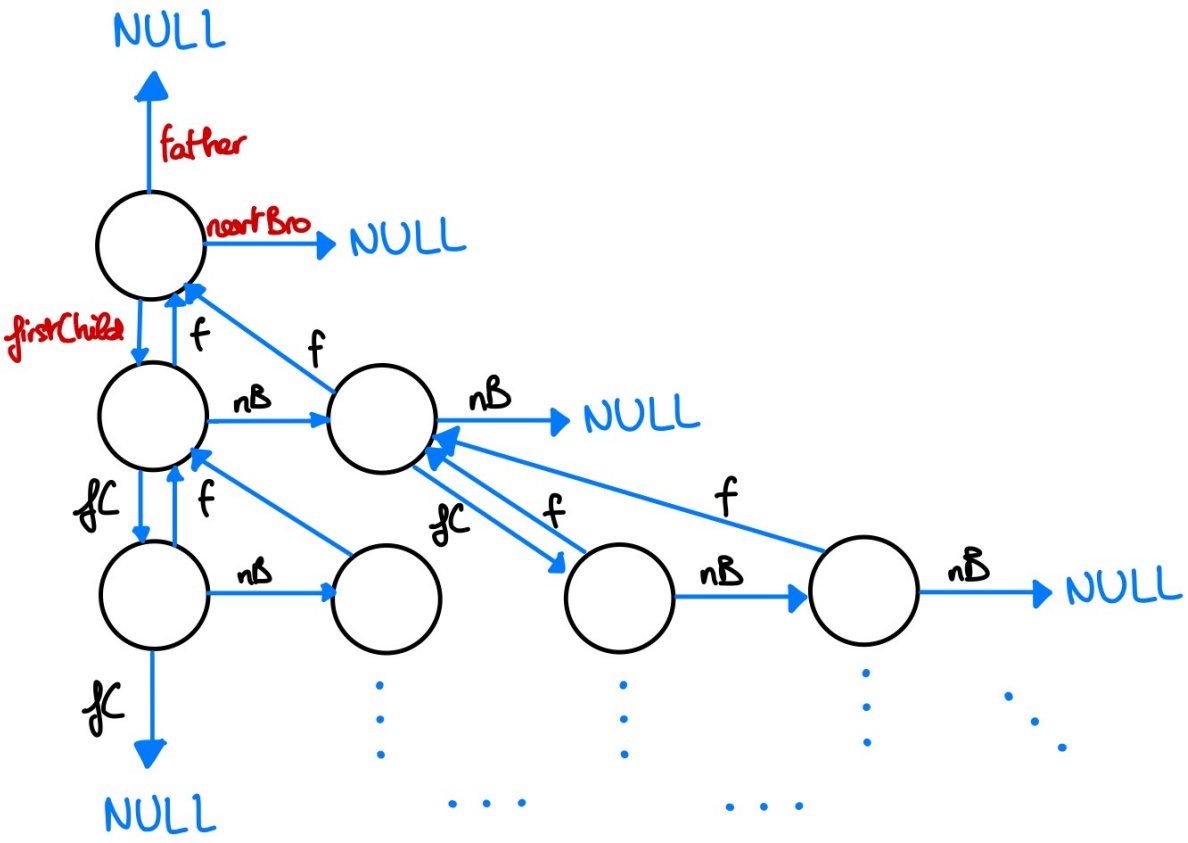
```
void delete_block(Block *b);
void free_block(Block* b);
```

→ Précisons que nous avons choisi d'implémenter en plus de la fonction préconisée dans le sujet une fonction `free_block` qui libère tous les champs mémoires du bloc, *author* inclus, de sorte à éviter les fuites mémoire.

4.2. Structure arborescente

4.2.1. Manipulation d'un arbre de blocs

Pour représenter une blockchain, nous avons utilisé un arbre de blocs dont chaque noeud représente un bloc. Chaque bloc a une hauteur, un père (sauf pour la racine) et peut avoir un nombre arbitraire de fils et fères. Cette structure nous permet de modéliser la blockchain et les tentatives de fraude.



Lorsque l'on ajoute ou supprime un noeud la fonctions `update_height` permet de mettre à jour les hauteurs respectives des différents noeuds de l'arbre.

```
void add_child(CellTree *father,CellTree *child);
```

→ Dans cette fonction nous ajoutons le noeud *child* en tête c'est à dire que le noeud *child* devient le premier fils du noeud *father*. De plus grâce à la fonction `update_height` nous pouvons mettre à jour les hauteur des noeuds pères de *child*.

Nous implémentons par ailleurs les fonctions classiques d'affichage et de suppression des noeuds ou de l'arbre.

4.2.2. Détermination du dernier bloc

Dans le cadre de ce projet, le mécanisme de consensus que nous avons choisi d'implémenter est celui du *Proof of Work*, qui requiert pour chaque ajout de bloc une quantité de ressources (puissance de calcul et énergie) très importante. De ce fait il est compliqué de falsifier un bloc et quasi-impossible, du fait de la quantité de ressources nécessaires, d'inscire ce bloc dans la blockchain à long terme. D'où l'adage déjà évoqué: *toujours faire confiance à la plus longue chaîne de blocs reçue à partir de la racine de l'arbre.*

```
CellTree* highest_child(CellTree* cell);
CellTree* last_node(CellTree* tree);
```

→ Dans cette partie nous avons donc implémenté deux fonctions nous permettant de retrouver la plus longue chaîne, et de retrouver la valeur hachée de son dernier bloc. Pour trouver cette chaîne il nous suffit de remonter de père en père (*T→father*) depuis le *CellTree** renvoyé par `last_node`

4.2.3. Extraction des déclarations de vote

Dans cette partie, nous avons implémenté la fonction qui nous permet d'obtenir la liste de déclarations contenues dans les blocs de la plus longue chaîne, en fusionnant avec la fonction `fusion` des listes chaînées deux par deux, et en choisissant à chaque fois le bloc le plus fiable avec `highest_child`.

```
void fusion(CellProtected **s1, CellProtected ** s2);
CellProtected *fusion_highest(CellTree *racine);
```

La fonction `fusion` actuellement en $O(n)$ avec n la longueur de la première liste, pourrait avoir une complexité plus intéressante en $O(1)$, si nous avions implémenté la structure *CellProtected* par une liste doublement chaînée car nous aurions alors eu un accès direct au premier et au dernier élément de chaque liste.

4.3. Simulation du processus de vote

Dans cette partie nous simuleront le fonctionnement de la blockchain à l'aide d'un répertoire *Blockchain* et des fichiers *Blockchain/Pending_block* et *Blockchain/Pending_votes.txt.*

Les votes des citoyens récoltés nous permettront de créer des blocs à intervalles réguliers que nous ajouterons ensuite à la base de données décentralisée.

4.3.1. Vote et création de blocs valides

On implémente la soumission de votes et l'ajout de blocs à l'aide des fichiers *Blockchain/Pending_block* et *Blockchain/Pending_votes.txt.*

```
Block *init_block(Key *author, CellProtected *votes, unsigned char *hash, unsigned char *previous_hash, int nonce);
void write_block(char *filename, Block *block);
Block* read_block(char *filename);
```

→ Nous avons choisi de créer des fonctions supplémentaires qui nous seront utiles pour la création des blocs et les manipulation fichier-bloc.

```
void submit_vote(Protected* p) ;
void create_block(CellTree** tree, Key* author, int d);
void add_block(int d, char* name);
```

→ `create_block` permet de créer un bloc temporaire qu'on ajoute à la base de données s'il est bien vérifié par `add_block`.

4.3.2. Lecture de l'arbre et calcul du gagnant

```
CellTree* read_tree();
```

→ Cette fonction s'appuie sur la bibliothèque <*dirent.h*> pour lire un arbre depuis un fichier. Elle lit l'intégralité des blocs présents dans le dossier *Blockchain.*

```
Key* compute_winner_BT(CellTree *tree, CellKey *candidates, CellKey *voters, int sizeC, int sizeV);
```

→ Cette fonction s'appuie sur compute_winner pour calculer le gagnant de l'élection à partir d'un arbre. Attention on suppose que la racine elle-même n'est pas une fraude. On pourrait suggérer comme amélioration que la racine soit un "bloc blanc" identique pour tout le monde.

Enfin le `main` permet de générer une base de données décentralisée en ajoutant des blocs crées tous les 10 votes grâce à `create_block`, `submit_vote` et `add_vote` à un arbre construit par `read_tree` et affiché par `print_tree`, puis en calculant le vainqueur de l'élection avec `compute_winner_BT`.

4.3.3 Conclusion

Au cours de ce projet et au fil des exercices nous avons traité de nombreuses structures de données qui bout-à-bout nous ont permit de construire une blockchain dont le mécanisme de consensus est le Proof of Work.

Cette implémentation décentralisée comporte un certain nombre d'avantages: l'anonymat, la sécurité (gestion efficace des fraudes et falsifications), la traçabilité et la transparence des votes et la possibilité pour les votants de faire une procuration. Néanmoins la blockchain ici présentée mérite d'être encore perfectionnée car elle présente encore deux inconvénients majeurs: d'un côté elle est difficile à appréhender ce qui pourrait susciter la méfiance des citoyens, et d'un autre elle doit encore être sécurisée. En effet la structure construite présente plusieurs failles: des possibles attaques ou pannes des serveurs, le cas où on trouve (au moins) 2 chaînes de blocs de longueurs maximale, la vulnérabilité du réseau si un pirate possède plus 50% de la puissance de calcul du réseau…

Enfin nous pouvons dire que le concept de blockchain est une piste très prometteuse d'alternative à notre système de vote par sa transparence, sa sécurité et sa capacité à faciliter l'accès au plus grand nombre au vote ou à sa procuration. Le « vote en ligne » pourrait donc bien permettre de réduire le taux d'abstention mais sa complexité et son impact environnemental jouent en sa défaveur.