Légende

Structure

.h

Le fichier params.h contient de nombreuses options pour modifier plus simplement le programme (ex: nb de declaration de vote, nb de candidat, nb de tour de test etc) ainsi que des path servant a structurer le projet

# PARTIE 1

### miller_rabin.h

```
int is_prime_naive(long p)
void print_is_prime(long p)
long modpow_naive(long a, long n, long m)
long modpow(long a, long n, long m)

int witness(long a, long b, long d, long p);
long rand_long(long low, long up);
int is_prime_miller(long p, int k);
long random_prime_number(int low_size, int
up_size, int k);

Fonctions de tests de primalite naive et
d'exponentiation modulaire

Fonctions de test de primalite probabiliste
(Miller Rabin) et de generation aleatoire de
nombre premier
```

### keys.h

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

void print_long_tab(long *tab, int size)

Fonctions de generation de pairs de cles pour le chiffrement RSA
ainsi que des fonctions de chiffrement/dechiffrement a l'aide de ces
cles.
```

# PARTIE 2

### Key

```
long val;
long n;

n: produit des deux nombres
premier
val: u ou s
```

### Signature

```
long *content;
int size;

content: message que la personne qui
signe a chiffre avec sa cle secrete
size: taille du tableau content
```

### Protected

```
Key *pKey;
char *declaration_vote
Signature *sgn;

pKey: cle publique de celui
qui vote qui permet de valider
la signature
declaration_vote: cle publique
de pour qui il vote
sgn: signature du votant
```

### keys_struct.h

```
void free_signature(Signature *s);
void free_protected(Protected *p);

void init_key(Key* key, long val, long n);
void init_pair_keys(Key* pKey, Key* sKey, long low_size, long up_size);

char* key_to_str(Key* key);
Key* str_to_key(char* str);

Signature* init_signature(long* content, int size);
Signature* sign(char* mess, Key* sKey);
char *signature_to_str(Signature *sgn);
Signature *str_to_signature(char *str);

Protected* init_protected(Key* pKey, char* mess, Signature* sgn);
int verify(Protected* pr);
char* protected_to_str(Protected *pr);
Protected* str_to_protected(char* str);

void print_long_vector(long *result, int size);

Fonctions permettant la creation de structure necessaire au processus electoral:
- Key -> cle publique ou secrete necessaire au chiffrement
- Signature -> Permet de signer une declaration a l'aide de sa cle secrete
- Protected -> contient la declaration de vote, la signature ainsi que la cle publique du
 votant qui permet de verifier sa signature
Les fonctions to_str et str_to permettrons la serialisation/deserialisation dans des
fichiers texte
Les fonctions sign et verify permettent de signer une declaration puis de la valider à
l'aide de la cle publique du votant
```

### generate_data.h

```
int is_in(int *tab, int val, int size);
void generate_random_data(int nv, int nc);
void generate_random_data_test(int nv, int nc);

is_in: renvoie 1 si val est dans tab 0 sinon
generate_random_data: Generation de .txt contenant la liste des cles de tout le monde, la
liste des cles des candidats, la liste des declaration de vote
generate_random_data_test: même chose mais dans un autre repertoire
```

# PARTIE 3

### CellKey

```
Key *data;
struct cellKey* next;

data: pointeur vers une cle
next: pointeur vers l'element suivant de
la liste chainee
```

### HashCell

```
Key *key;
int val;

key: cle d'un electeur
val: entier representant si la personne a
vote (0: non, 1: oui)
```

### CellProtected

```
Protected* data;
struct cellProtected* next;

data: pointeur vers une protected
next: pointeur vers l'element suivant de
la liste chainee
```

### HashTable

```
HashCell **tab;
int size;

tab: tableau de hashcell (table de
hachage)
size: taille de la table de hachage
```

### linked_keys.h

```
CellKey* create_cell_key(Key* key);
CellKey* insert_cell_tete(Key* key, CellKey *next);
CellKey** read_public_keys(char* nomfic);

void print_list_keys(CellKey *LCK);
void delete_cell_key(CellKey *ck);
void free_list_keys(CellKey **lk);

CellProtected* inserer_list_protected(Protected *pr,
CellProtected *next);
CellProtected* create_cell_protected(Protected *pr);
CellProtected* read_declarations(char* nomfic);
void print_list_protected(CellProtected *pr);
void delete_cell_protected(CellProtected *pr);
void free_cell_protected(CellProtected *pr);

Fonctions permettant de deserialiser les
cles/declarations contenu dans les .txt et de mettre dans
une liste chainee
```

### hash.h

```
void verify_list_protected(CellProtected **cp);

HashCell *create_hashcell(Key* key);
int hash_function(Key* key, int size);
int find_position(HashTable *t, Key* key);
HashTable* create_hashtable(CellKey* key, int size);
void delete_hashtable(HashTable *t);

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

verify_list_protected: verifier les signatures contenus
dans les protected à l'aide de la clé publique du votant.
Supprime les signatures invalides de la liste chaînée.

Fonctions permettant de creer les tables de hachage
associées aux candidats et aux electeurs (on utilise le
probing lineaire pour les collisions)
compute_winner: génère les tables de hachages et permet
de comptabiliser les votes en vérifiant les signatures,
l'existence d'un candidat et le fait que l'électeur n'ai
pas encore voter. (renvoie la clé du gagnant de
l'élection)
```

# PARTIE 4

### Block

```
Key *author;
CellProtected *votes;
unsigned char *hash;
unsigned char *previous_hash;
int nonce;


author: clé publique du createur du block (assesseur)
votes: liste chainée de déclarations de votes
hash: valeur hachee du block
previous_hash: valeur hachée de bloc precedent dans la
blockchain
nonce: preuve de travail
```

### CellTree

```
Block *block;
CellTree *father;
CellTree *firstChild;
CellTree *nextBro;
int height;

block: block associé au noeud
father: pointeur vers le pere
firstChild: pointeur vers le premier fils
nextBro: pointeur vers le prochain frere
height: hauteur du noeud
```

### block.h

```
void delete_block(Block *b);
void delete_block_v2(Block *b);
void write_block(char* file_name, Block *b);
Block* read_block(char *file_name);
char* block_to_str(Block *block);

char* hash_sha256(char* str);
void compute_proof_of_work(Block *B, int d);
int verify_block(Block *b, int d);

Fonctions de serialisation/deserialisation des blocks

hash_sha256: Génère une chaîne de 256 charactères via
une chaîne issue de block_to_str (SHA-256)

compute_proof_of_work: modifie la valeur du champ hash
et nonce tant qu'il n'y a pas d zeros successifs au
debut de hash

verify_block: verifie que la valeur hash du block a
bien d zeros successifs au début (rend 1 si le block
est valide 0 sinon)
```

### tree.h

```
CellTree *create_node(Block *b);

int update_height(CellTree *father, CellTree *child);
void add_child(CellTree *father, CellTree *child);
void print_node(CellTree *node);
void print_tree(CellTree *ct);
void print_clean_tree(CellTree *ct, int max_height);
void delete_tree_cell(CellTree *node);
void delete_tree(CellTree *ct);
void delete_tree_cell_v2(CellTree *node);
void delete_tree_v2(CellTree *ct);

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

CellProtected **merge_list_decla(CellProtected **l1,
CellProtected **l2);

Fonctions permettant de générer un arbre de block
(représentant notre blockchain)

highest_child: fonction permettant de récupérer le
fils le plus loin, cela sert à récupérer la chaîne de
block la plus longue de notre blockchain (on
considère la chaîne la plus longue comme étant celle
qu'on doit croire car celle qui a nécessité le plus
de puissance de calcul)

merge_list_decla: Renvoie une liste chaînée issue de
la fusion de l1 et l2, permet la récupération de la
liste chaînées associés à plusieurs blocks

Certaines fonctions ont du être recoder suite à un
changement d'énoncé, elle porte la mention "v2"
```

# PARTIE 5

### vote.h

```
void submit_vote(Protected *p);
void create_block(CellTree *tree, Key *author, int d);
CellTree *init_tree(Key *author, int d);
void add_block(int d, char *name);
char* generate_uuid();
CellTree* read_tree();
Key* compute_winner_BT(CellTree *tree, CellKey
*candidates, CellKey *voters, int sizeC, int sizeV);

generate_uuid: génère des chaine aleatoire (pas des
vraies uuid) qui permette de nomme les fichiers
contenant les blocks de notre blockchain

init_tree: Fonction créant un block qui va être la
racine de l'abre

Globalement on soumet des votes via submit_vote dans un
fichier d'attente, au bout d'un certain nombre de vote
on génère un block associés aux votes et on l'ajoute à
notre blockchain. Après la fin de la periode de vote on
recupere tous les blocks de notre blockchain et on
genere l'abre associes. On suit ensuite le chemin le
plus long dans l'arbre et créer la liste chaînée de
declaration de vote associés a chaque blocks. Ensuite on
fait le décompte pour annonce le vainqueur. (On vérifie
bienour la validite des signatures et des blocks)
```