

# Data Structure Project: Blockchain applied to an electoral process

04.25.2022

---



Chadi Tawbi - Said Bougjdj  
2nd year Computer Science  
2021-2022




## Introduction

The project consists of the implementation and the manipulation of cryptographic tools for the organization of an electoral process by first-past-the-post voting. The goal of this project is to make a secure election but also to see the difference between a centralized and decentralized database and to be able to deduce if a blockchain implementation is preferable in the framework of a voting process.

**Here are the different steps we took to achieve this objective:**

### **Centralized database:**

- **step 1:** the implementation of codes at the base of some encryption tools. More specifically, we have addressed the problem of generating prime numbers.
- **step 2:** we used the previous codes to implement functions that are at the origin of the encryption and decryption of the RSA protocol, where pairs of public and private keys of the recipient receiving the encrypted message are needed.
- **step 3:** after studying a singular case, we extend this cryptographic mechanism to a voting situation where voters use their private keys to sign during the vote, whose authenticity can be verified by anyone using the public key. Thus, key and signature functions are manipulated.
- **step 4:** we simulated a vote with a set of public and private keys and votable candidates.
- **step 5:** we proceed to the accounting of the voting declarations using the public keys, grouped in linked lists.



**step 6:** then we use the hash tables to count the number of votes for each candidate to finally find the winning candidate.

### **Decentralized database:**

**To set up a decentralized database, we started by creating a blockchain**

**step 1:** thanks to the SHA-256 cryptographic hash function we have set up a consensus mechanism called 'proof of work'.

**step 2:** to limit fraud we have created a tree structure of blocks. With this, we can deduce which is the longest blockchain starting from the root and therefore conclude that this blockchain is probably the least fraudulent.


**Step 3:** Finally, we simulated the vote submissions by citizens and the valid block creations by assessors (volunteer citizens). Then, we created the corresponding block tree, and computed the winner of the election by trusting the longest chain in the tree.

## Global descriptions of code and structures:

### Structures used:

**Many structures have been manipulated:**

- **A Key structure**, identifying the public and secret keys, whose values are generated from the initial algorithms (key->value, key->size)
- **A Signature structure** corresponding to the encrypted message in the voting situation, vote (sgn->content (array of long) , sgn->size (int) )
- **A Protected structure** which contains the public key of the voter, his declaration and the associated signature. (p->pkey , p->mess (decla) ,p->sgn (associated signature))
- **A CellKey structure** gathering the public keys in the form of a linked list
- **A CellProtected structure**, linked lists of signed declarations
- **A HashCell structure** that contains a key and an integer val. Val for a citizen represents whether he voted or not (1 if he voted and 0 otherwise). For a candidate, val represents the number of votes won.
- **A HashTable structure** corresponding to a Hash Table which contains HashCell placed in the table thanks to a hash function. It also contains an integer size which corresponds to the size of the table.



-**A Block structure** which represents a block of the blockchain which is chained using the hash value of the previous block.

This one contains:

The public key of its creator.

A list of voting statements.

The hash value of the block.

The hash value of the previous block.

A proof of work.

-**A CellTree structure** which corresponds to a tree of blocks. This one allows us to represent nodes with an arbitrary number of children without having to use a chained list. So we can browse this tree with the help of firstChild which represents the first child and then nextBro to access the children until we get the NULL pointer. Finally we have the height field which represents the height of the node

.

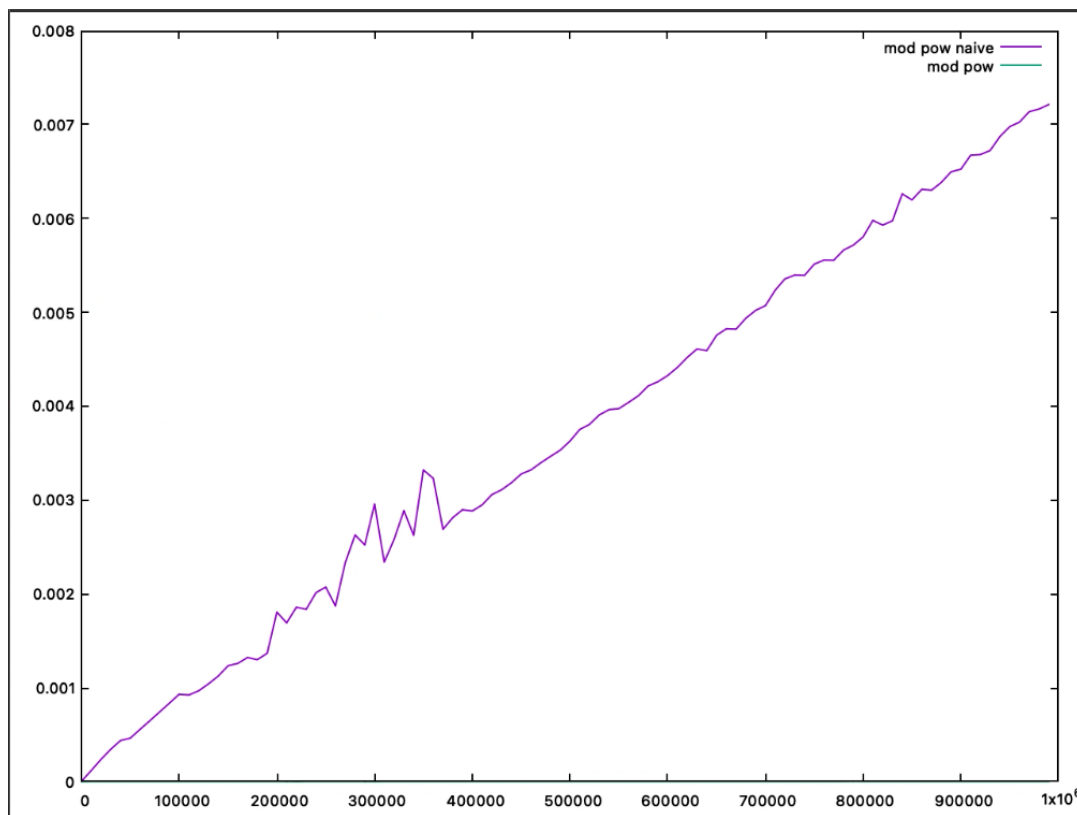
## Description of the code and the different functions:

### Fast modular exponentiation:

**int is\_prime\_naive(long p):** test if p is a prime or not.

**long modpow\_naive(long a, long m, long n):** returns the value  $a^b \bmod n$  by the naive method.

**long modpow(long a, long m, long n):** performs a squaring.



Looking at this graph, (the modpow curve is well present but difficult to see) we can see that modpow\_naive has a complexity of  $O(n)$  while modpow has a complexity of  $O(\log_2 n)$ . So we can conclude that modpow is much faster and it is this function that we will use for the rest of our project.

### **Test of Miller-Rabin:**

**int witness(long a, long b, long d, long p):** this one returns 1 if a is a Miller witness for p, for a given integer a.

**long rand\_long(long low, long up):** returns a randomly generated long between low and up inclusive.

**int is\_prime\_miller(long p, int k):** this one checks k times if we find a Miller witness for p. It returns 0: if we find a witness for p, we can say that p is not prime. Otherwise it returns 1, if we don't find a witness, then it is likely that p is prime.

**long random\_prime\_number(int low\_size, int up\_size, int k):** returns a prime number of size between low\_size and up\_size.

### **RSA Protocol:**

**long extended\_gcd(long s, long t, long \*u, long \*v):** Returns the largest common divisor between s and t.

**void generate\_key\_values(long p, long q, long \*n, long \*s, long \*u):** The variable n being the multiplication of p and q. This function generates a public key (s, n) and a secret key (u, n).

**long\* encrypt(char\* string, long s, long n):** This one encrypts the string with the public key pKey = (s, n). To do this, the function performs a modular exponentiation (using modpow) and converts each character into an integer and returns the array of long obtained by encrypting these integers.

**char\* decrypt(long\* crypted, int size, long u, long n):** Using the secret key sKey=(u, n), this one decrypts a message (using modpow) and returns the resulting string.

**void print\_long\_vector(long \*result, int size):** displays an array of long of length size.

### **Manipulation of keys:**

**void init\_key(Key\* key, long val, long n):** initializes an already allocated key.

**void init\_pair\_keys(Key\* pKey, Key\* sKey, long low\_size, long up\_size);** This calls the functions random\_prime\_number, generate\_keys\_values and init\_key to initialize the public and private keys via the RSA protocol.

**char\* key\_to\_str(Key\* key):** converts a key to string.

**Key\* str\_to\_key(char\* str):** converts string to key.

**Key \*copie\_key(Key \*key):** Make a copy of a key.

### **Manipulation of signatures:**

**Signature\* init\_signature(long\* content, int size):** allocates and initializes a signature with an array of long.

**Signature\* sign(char\* mess, Key\* sKey):** returns a pointer to a signature that contains an encrypted mess (so a long array, cf encrypted) using the secret key to encrypt.

**char\* signature\_to\_str(Signature \*sgn):** converts a signature to a string with a '#' between each encrypted char.

**Signature\* str\_to\_signature (char\* str):** converts a string to signature.



### **Manipulation of declarations:**

**Protected\* init\_protected(Key\* pKey, char\* mess, Signature\* sgn):** initializes a signed vote declaration.

**int verify(Protected\* pr):** returns 1 if the signature contained in pr matches the message to the person contained in pr, 0 otherwise.

**char\* protected\_to\_str(Protected \*pr):** converts protected to string.

**Protected\* str\_to\_protected(char\* str):** converts string to protected.

### **Simulation of an election:**

**void generate\_random\_data(int nv, int nc):** This generates public and secret key pairs nv times, which are saved in the file keys.txt. Then we randomly choose different nc (candidates) public keys from keys.txt and we save them in the file candidates.txt. Finally, we create nv declarations and we save them in the file déclaration.txt.

### **Linked list of keys:**

**CellKey\* create\_cell\_key(Key\* key):** creates an empty CellKey structure

**void insert(CellKey\*\* LCK, Key\* key):** inserts a CellKey at the head of the list

**CellKey\* read\_public\_keys(char\* filename):** creates a list containing all the keys of the file.

**void print\_list\_keys(CellKey\* LCK):** displays a list of keys.

**void delete\_cell\_key(CellKey\* LCK):** deletes a cell from the list of keys.

**void delete\_list\_keys(CellKey\* LCK):** delete a list of keys.

**int list\_keys\_length(CellKey \*c):** returns the size of the list of keys.

### **Linked list of declarations:**

**CellProtected\* create\_cell\_protected(Protected\* pr):** creates an empty Protected list structure

**void inserter\_CellProtected(CellProtected\* cpr, Protected\* data):** adds a signed declaration at the head of the list.

**CellProtected\* read\_protected():** creates a list containing all the signed declarations of the file

**void print\_list\_protected(CellProtected\* c):** display the list of declarations

**void delete\_cell\_protected(CellProtected\* c):** deletes a cell from a linked list of signed declarations

**void delete\_list\_protected(CellProtected\* c):** deletes the whole list of declarations

**CellProtected \*merge\_list\_protected(CellProtected \*votes1, CellProtected \*votes2):** merges two lists of declarations.

### **Manipulation of Hash table:**

**void fraud(CellProtected\*\* LCP):** removes all invalid declarations from a list of declarations.

**HashTable\* create\_hashtable(CellKey\* keys,int size):** allocate and initialize a Hash table by placing each cell (cf create\_hashcell) in their respective position thanks to the hash function (see hash\_function).

**HashCell\* create\_hashcell(Key\* key):** allocates and initializes to 0 a HashCell that will be placed in the hash table.

**int hash\_function(Key\* key,int size):** compute the position of the cell with the values of the key using the formula  $(v+1)*(n+1)\%size$

**int find\_position(HashTable \*t ,Key\* key):** searches in the hash table if there is an element whose public key is key and if it is found, the function returns its position in the table otherwise, it returns the position where it should have been.

**void delete\_hashtable(HashTable\* t):** delete a hash table

**Key\* compute\_winner(CellProtected\* decl, CellKey\* candidates, CellKey\* voters, int sizeC, int sizeV):** create two hash tables (one table of sizeC for the list of candidates and one table of sizeV for the list of voters). Verify that all the declarations are valid (see fraud), that the person who votes has the right to vote and has not already voted and that the person who is voting is a candidate. When the conditions are met, the vote is counted in the candidates table and the voters table is updated to indicate that the voter has just voted. Finally the function runs through the hash table of candidates to determine the winner (the one with the most votes).

### **Manipulation of blocks:**

**Block \*creerBlock(Key \*author, CellProtected \*votes, unsigned char \*hash, unsigned char \*previous\_hash, int nonce):** allocate and initialize a block.

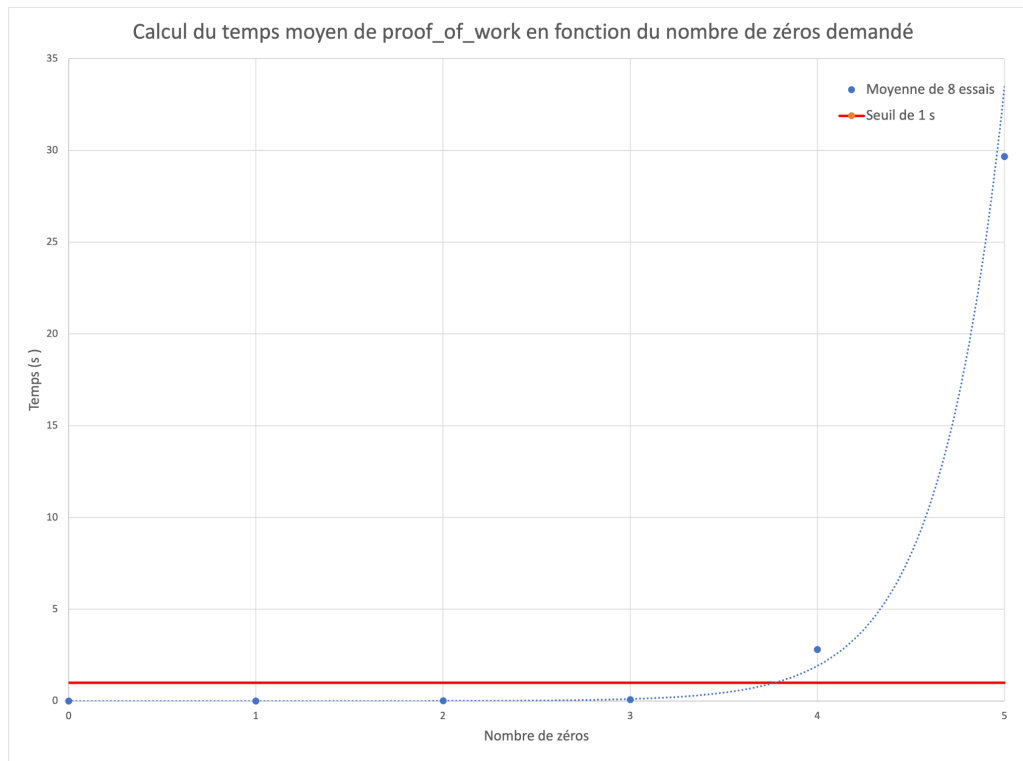
**void enregistrarBlock(char \*filename, Block \*block):** write a block to a file.

**Block \*lireBlock(char \*filename):** read a block from a file.

**char \*block\_to\_str(Block \*block):** generates a string representing a block that contains the author's key, the hash value of the previous block, all its votes and a proof of work.

**unsigned char\* hash\_function\_block(const char\* str):** this is the SHA-256 hash function that takes as input messages of variable size and produces hashed values of fixed size(256 bits).

**void compute\_proof\_of\_work(Block \*B, int d):** To make a block valid, we proceed by brute force. This means that we start with the nonce attribute at zero, then we increment the nonce attribute until the hashed value of the block starts with d successive zeros.



We can observe an exponential growth of the time when the number of 0 d increases. Its complexity is therefore  $O(2^d)$ .

We can see that from 3 zeros the execution time of the function exceeds 1 second (It all depends on the processor on which the execution is done).

**int verify\_block(Block \*B,int d):** verifies that a block is valid (if the hashed value of its data starts with d successive zeros).

**void delete\_block(Block \*b):** deletes the block but does not free the memory associated with the author field and does not free the content of the chained list of votes.

### **Manipulation of Tree of blocks:**

**CellTree\* create\_node(Block\* b):** create and initialize a node with a height of 0.

**int update\_height(CellTree \*father, CellTree \*child):** updates the height of the father node when one of its children has been modified.

**void add\_child(CellTree \*father, CellTree \*child):** adds a child to a father and updates the height of all ascendants (with the help of update\_height).

**void print\_tree(CellTree \*tree):** displays the tree and more precisely the height and the hash value of the block of each node.

**void delete\_node(CellTree \*node):** Deletes a node

**void delete\_tree(CellTree \*tree):** Deletes a tree

**CellTree\* highest\_child(CellTree\* cell):** Retrieves the child node with the biggest height.

**CellTree \*last\_node(CellTree \*tree):** Returns the leaf of the largest "branch" of the tree.

**CellProtected\* get\_trusted\_list\_protected(CellTree\* tree):** Retrieves the list of declarations of the longest branch (with the help of highest\_child) starting from the root of the tree. All the lists of declarations are merged thanks to the merge\_list\_protected function seen before.

### **Simulation of the election:**

**void submit\_vote(Protected\* p):** Allows a citizen to submit a vote, thus adding his vote at the end of the "Pending\_votes.txt" file

**void create\_block(CellTree\*\* tree, Key\* author, int d):** This creates a valid block that contains the pending votes in the file "Pending\_votes.txt" and writes the resulting block to a file called "Pending\_block".The file "Pending\_votes.txt" is deleted after the block is created.

**void add\_block(int d, char\* name):** The block in "Pending\_block" goes through a verification to see if it is valid (cf verify\_block).if it is the case then a name file representing the block is created and is added to the "Blockchain" directory. The file "Pending\_block" is deleted in any case.

**CellTree \*read\_tree():** browse the "Blockchain" directory and for each block, create a node and thus create a block tree.

**Key\* compute\_winner\_BT(CellTree\* tree, CellKey\* candidates, CellKey\* voters, int sizeC, int sizeV):** This one first extracts the list of voting declarations of the highest height of the root (cf get\_trusted\_list\_protected). Then it deletes all the invalid vote declarations with the fraud function.finally with the compute\_winner function, it finds the winner and returns his public key.



## **Conclusion:**

The use of a blockchain is a very interesting model in this context for the realization of a voting process.

However, it is not possible to completely prevent fraud attempts. Lets say the longest branch has multiple brothers of the same length, it is impossible to distinguish the branch of verified statements and the one that is fraudulent.

In conclusion, we would still recommend the use of blockchain for an election over a centralized version since it is very hard to manipulate blocks and the election as a whole.