DSA PROJECT REPORT

Project : Bitcoin

Team Number : 1

Team members :

- Jewel Benny (2020102057)
- Pallav Koppisetti (2020102070)
- Anmoldeep Dhillon (2020101085)
- Atharv Sujlegaonkar (2020102025)

Index

Explanation of adding user part:

The Functions used are :

1) AddUser:

I. Data structure Used: hash table, linked lists.
II. Time complexity:

O(1) when load factor is less than 0.5, after that it deteriorates to O(n) , which is handled by resizing the hash table and rehashing, rehashing being amortized time of O(1) on an average. Updating the linked list of transaction pointers is also in O(1) time.

III. Algorithm:

AddUser function first calls a function called randomID which generates a random string of 10 alphanumeric characters, which is assigned to our new user.

Then it takes this string and hashes it using a rolling hash function called “hash”, which outputs a key. Now, in our case since we are using quadratic probing , we take this key and pass it to “quadprob” function which gives a valid index in our hash table.

Then we insert our user into our user hash table or create one if one does not exist that stores user ID, join date and time and creates a pointer that is basically part of a linked list which increases in size as that user does any transactions and this pointer called UTH, that points to the transaction that the new user is a part of in the block. The most recent transaction being at the top.

IV. Error handling:

Due to the efficiency degradation of hash table as buckets keep getting filled , we resize our table and rehash all the buckets as soon as load factor hits 0.5 to keep our efficiency of the hash table considerably intact.

2) randomID:

I. Data structure used: character array.
II. Time complexity: O(1) due to random access feature of arrays.
III. Algorithm:

This function first initializes a static character array and assigns it a string that has all the alphabets (both uppercase and lowercase) and numbers from 0-9 and then we call a function that ensures that we have seeded our pseudo random number generator. We generate random numbers that are less than the string size and then use that as an index to copy that character into our RandomID array. We then return this array which basically becomes our UserID.

3) Hashing:

I. Data structure Used: character array.
II. Time complexity: O(string length)
III. Algorithm :

This function takes our UserID as input (which is basically a char array) and performs rolling hashing on it .We pick our modulus constant to be a huge number (1e9+9) so as give different keys and the hashing constant is chosen as 31 based off of statistical data. After the rolling hashing, a key is generated that is returned.

4) Quadratic probing:

I. Data structures: Array
II. Time complexity: O(n) worst time if not rehashed.

III. Algorithm:

This function basically helps us in open addressing which is used to tackle collisions while hashing. It tackles the issues of linear probing such as primary clusters by incrementing probing check index by i^2 instead of i (i belongs to whole numbers). But this might face issues as buckets keep getting filled without rehashing.

IV. Error handling:

Even though quadratic probing helps in avoiding primary clusters, we still have the problem of secondary clusters., which we can avoid by double hashing i.e. hashing the key again to give a new key and then finding an index. We rehash and resize table as soon as we hit load factor = 0.5 to maintain O(1) runtime operations of hash tables.

User interface:

A basic UI has been created to test out functions easily using the terminal. Kindly refer to the readme file to know more about the functions/command that provided in the user interface.

The user interface asks the user a yes/no question to know whether the user wants to continue running the program or would like to exit out of it.

Have provided proper error messages to be printed so as to take care and warn the user of providing invalid input to the function.

Link:
https://github.com/pallavkoppisetti/DSA-project/blob/main/README.md

Explanation of Transact Part :

1) TransactionValidity(User *Sender, User *Receiver, long double AmountToBeTransferred):

Its time compexity is O(1) respectively. This function checks whether a transaction is valid or not. The conditions covered in this function to check the validity of the transaction are:

1. Sender with given UID does not exist:
   Returns 2 in this case.

2. Receiver with given UID does not exist:
Returns 3 in this case.

3. Sender and Receiver UIDs are same.
   Returns 6.

4. Wallet balance of sender is lesser than amount to be transferred.
   Returns 4.

5. Amount to be transferred non-positive.
   Returns 5

   In other cases, transaction is considered to be valid, and the function returns 1.

2) UpdateUserHistory (User *Sender, User *Receiver, char *SenderUID, char *ReceiverUID, long double AmountToBeTransferred) :

Its time complexity is - O(1). It Updates the user (sender and receiver) transaction history (calls push function) and the wallet balance.

3) CreateUserTransactHistory() :

Its Time complexity - O(1). It Creates user transaction history after adding the user, it allocates memory to struct pointer containing head and tail nodes and calls InitCreateTransactHistory and returns pointer to struct pointer

4) InitCreateTransactHistory(pointer* Q):

Time complexity - O(1)

It allocates memory to head and tail nodes in pointer* Q.

5) PrintUserTransactionHistory(char *UID):

-Its Time complexity-O(n)

-Where n is the no of transactions in user history.

-It prints user transaction history in order from latest to oldest by taking user id as parameter.

6) UpdateBlockTransactionHistory(char *SenderUID,

char *ReceiverUID, long double AmountToBeTransferred) :

Time complexity - O(1)

It Updates temporary transaction array that stores transactions. When the array is full the address is passed to the block and starts from 0 transactions again. So, the memory for this array is allocated again.

7) Transact() :

Its Time Complexity - Sum of time complexities of all the functions it calls to perform the transactions.

It is the main transaction function which takes input and calls the other functions to perform the transactions.

- Takes input.
- Calls SearchUserByID(SenderUID) and SearchUserByID(ReceiverUID)
- Calls TransactionValidity

If transaction is valid, then it calls UpdateBlockTransactionHistory() and UpdateUserHistory()

8) RepTransact(char *SenderUID, char *ReceiverUID, long double AmountToBeTransferred):

For performing transaction n no. of times.

Data structures used:

1. Doubly linked list for user transaction history.
2. Array of transactions of 50 elements that stores.

Explanation of block and hash functions, and other print functions :

1) CreateBlock() :

- Its Time Complexity – O(1)

The CreateBlock function as the name suggests, is a function that creates a new block and adds it to the blockchain. It is called automatically when the number of transactions performed is a multiple of 50. A block contains the details – block number, block nonce, hash of the previous block, hash of itself and history of 50 transactions.

50 transactions at a time are stored in a temporary array. When the count of temporary transactions reaches 50, a pointer in the block points to the location of the temporary array. Then the temporary transaction count is reset, and the array is reallocated to a new address in memory.

2) GenerateNonce() :

Time Complexity – O(1)

A simple function to generate a random number between 1 and 500.

3) ValidateBlockChain function() :

This function is used to validate if the blockchain obtained is correct or not.

Time Complexity – O(N) where N is the number of blocks in the blockchain.

This function checks whether a block has been attacked (i.e., had its nonce changed. It iterates through the blocks. If it finds out that the nonce of a block has been modified (i.e., Hash value changes) then it corrects its nonce to the original value.

4) PrintBlock(int BlockNumber) :

Time Complexity – O(1)

Prints the details of the block with block number BlockNumber. Also prompts the user whether to display information regarding transactions stored in the block. If the input is an invalid block number, it throws an error message.

5) PrintUserDetails(char *UID) :

Time Complexity – O(M*(N + M)) where N is the number of valid transactions the user has performed, and M is the size of user hash table.

If an invalid UID is inputted, it throws an error message. Else it prints the user details including User ID, Wallet Balance, Join Date Time a prompt whether to display the transaction history of the user.

6) PrintUserTransactionHistory(char *UID) :

Time Complexity – O(N + M) where N is the number of valid transactions the user has performed, and M is the size of user hash table.

If an invalid UID is inputted, it throws an error message. Else it prints the transaction history of the user.

7) UpdateBlockTransactionHistory(char *SenderUID, char *ReceiverUID, long long AmountToBeTransferred) :

TimeComplexity – O(1)

Stores the transaction details in a temporary array, when the array has 50 transactions, it adds the transactions to a new block. Then the temporary array is reallocated to a new address in memory.

Data Structures used :

1. A simple hash table to store the blocks where the i^{th} element in the hash table is a pointer pointing to the block with block number i + 1. The hash table size is doubled when the number of occupied slots is half the table size.

2. An array of 50 elements to store transactions temporarily until they get added to a block.

Explanation of hash function

For getting Hashvalue of each block we used the openssl library for getting access to cryptographic hash function SHA256 which is predefined in it.

We created the following function to get the hash value :

unsigned char hash value* Generate hashvalue(Blockptr Cblock) :

This function returns the hash value which is of type unsigned char which takes the block as an argument to use all the blocks variable to get hashvalue.

First in this function we convert the array of transaction which contains 50 transactions to a string.

We do this by running a loop from 0 to 50 and concatenating every element of transaction like senderUID, to a Buffer string which is declared firstly in the function.

Then we run the special function of snprintf to combine all the block variable and string which is made earlier from transaction array to a single string.

It literally combines everything into a single string which we input into the SHA256 cryptographic function.

SHA256 is a special cryptographic function which is used to get hash value of a block. It takes the buffer string, size of the string and destination string as the arguments. It returns the unsigned char array which we copy into our block hash array which is also of type unsigned char by using strcpy function.

And then we finally return the hash value.

Data structures used:

1] Array of type unsigned char to combine all the block variables.

Time complexity is O(1).

Explanation of attack function and validate blockchain function :

attack() function:

In this function we first generate random value by using rand() function. This function will generate the random value every time we run it and passing it to a variable called RandomBlockNumber.

If the generated number is less than or equal to the number of blocks then we proceed towards next part of the function.

In the next part we just find the block which has block number == randomly generated number.

Then we modify its nonce by calling the generateNonce() function which generates the nonce.

And hence we successfully modify the Nonce of the block and print attack is successful.

If the randomly generated number is greater than the number of blocks then we print attack unsuccessful because there is no block which has block number same as randomly generated number.

Time complexity of the function is O(1).

Division of work :

Jewel Benny :

- block.c
    - CreateBlock
    - PrintBlockDetails
    - GenerateNonce
    - ValidateBlockChain
- transact.c
    - PrintUserDetails
    - UpdateBlockTransactionHistory
    - PrintUserTransactionHistory
- Explanations for some block functions
- Testing and debugging code

Pallav Koppisetti :

- user.c
    - AddUser
    - SearchUserByID
    - randomID
    - Functions for hashing (user part)
    - Explanations for user part
- user interface
- README.md
- Testing and debugging code

Anmoldeep Dhillon :

- transact.c
    - Transact
    - TransactionValidity
    - Storing user transaction history
    - Other supporting functions
    - Explanations for transact part
- testing and debugging code

Atharv Sujlegaonkar :

- creating hash function which generates hash value of block
- creating attack function
    - Explanations for attack function
- Explanations for hash functions
- Editing and compiling the report
- testing and debugging code

 # Link for the Github repository of the project :
https://github.com/pallavkoppisetti/DSA-project