DSA MINI PROJECT

TEAM NUMBER 50

BITCOIN

TEAM MEMBERS:

1) Karmanjyot Singh (2020101062)
2) Pranathi (2020101083)
3) Shruti (2020102053)
4) Ruchitha (2020101093)
5) Aryan (2020102032)

DATA STRUCTURES USED

1) Arrays

- Scanning user input for operations.\
- Storing the hash function(strings for storing the 32-bit block hash)

2) Linked Lists

- Block Chain:
  Header:
  - Pointers to the Head and Tail of the Block Chain
  - Number of Blocks
- BlockNodes:
  - Block Number

- Nonce
- Pointer to next block in the chain
- Previous Block Hash
- Transaction list head for 50 transactions

- Transaction List: (User Transaction History and Block Transactions)
  Header:
    - Pointers to the Head and Tail of the Transaction List
    - Number of Transactions
  ListNodes:
  - Sender's UID
  - Receiver's UID
  - Transaction Amount
  - Time stamp

3) AVL Trees (self-balancing data structure)

- Stores User Nodes based on their UID
- Node:
  - UID
  - Transaction History List specific to that User
  - Wallet Balance
  - Height of the User Node in the tree (for balancing the tree)
  - Time stamp
  - Pointers to its Left Child and Right Child

- Balancing the tree: LeftChild UID < CurrentUser UID < RightChild UID

ALGORITHMS and FUNCTIONS USED AND THEIR COMPLEXITY

1) AddUser

1. Generates a random ID
2. Checks if the ID has already been used
3. If used, it generates another ID and this process continues until a UID is generated
4. Recursively looks for the correct position of the new User
5. Performs rotations if the balancing property is disturbed after insertion

- Utility functions used:

1. Search-to search avl tree, returns pointer the user node if found in the avl tree else null.
2. CreateUserNode-For creating a user node in the AVL tree
3. Height-Returns height of a particular user node in the AVL tree
4. SingleRotateWithLeft
5. SingleRotateWithRight
6. DoubleRotateWithLeft
7. DoubleRotateWithRight

• Average Time Complexity: O(logn) (for validating and inserting the new Node)

• Worst-case Time Complexity: O(logn)

2) Transact

1. Checks if Amount is greater than zero
2. Checks if Sender's UID and Receiver's UID are valid
3. Checks if Sender has enough Wallet Balance
4. Prints User Data before transfer
5. Updates Wallet Balance according to the amount to be transferred
6. Updates Transaction History List of both the Users
7. Prints User Data after transfer
8. If the transaction is valid, it updates the Block Transaction List

9. Creates a new Block if the Current Transaction list has 50 transactions.
10. Utility functions used:
    - Search
    - PrintUserInfo-utility function for printing user information before and after transaction.< userid,wallet balance,joining date/time>
    - UpdateHistrory-To update the transaction history of the user nodes in the tree
    - CreateNode- utility function for creating transaction node
    - AddTransactionToList-utility function for adding a transaction node to the given list header
    - CreateHead- utility function for creating transaction list head
    - PrintTransactionList-utility function for printing details of all transactions stored in transaction list head.

• Average Time Complexity: O(logn)
• Worst-case Time Complexity: O(logn)

3) CreateBlock

1. Takes the Current Block Chain and adds a new block to its rear end, when current list header has had stored 50 transactions.
2. Randomly generates a number for the Nonce
3. This function is called only when the CurrentBlock has 50 transactions
4. To add this block to the BlockChain, we use the function AddToChain
5. Time Complexity: O(1)

4) Attack

1. Generates a random number between 1 and 50

2. If a Block currently in the Block Chain has a Block Number that is equal to the randomly generated number, it modifies its nonce and returns 1 (ATTACK SUCCESSFUL)
3. Else, it returns 0 (ATTACK FAILS)
4. Average Time Complexity: O(n) (n: number of Blocks in the chain)
5. Worst-case Time Complexity: O(n) (n: number of Blocks in the chain)

5) ValidateBlockChain

1. Checks if the Block Chain contains more than one Block
2. Recursively checks for invalid PrevBlockHash
3. If PrevBlockHash is invalid, it looks for the correct value of Nonce
4. Fixes the value of PrevBlockHash
5. Prints the number of Attacks detected
6. Utility Functions Used: CreateHash
7. Average Time Complexity: O(n) (n: number of Blocks in the chain)
8. Worst-case Time Complexity: O(n) (n: number of Blocks in the chain)

6) PrintUserInfo

1. Checks if UID is valid
2. If valid, prints the User's information
3. Utility Functions Used: Search
4. Average Time Complexity: O(logn)
5. Worst-case Time Complexity: O(logn)

7) PrintTransactionList

1. Checks if UID is valid
2. If valid, prints the User's Transaction History List
3. Utility Functions Used: Search

4. Average Time Complexity: O(m) (m = No. of transactions in the list)
5. Worst-case Time Complexity: O(m) (m = No. of transactions in the list)

NOTE:WHILE PRINTING DETAILS OF A USER O(logn) time factor also occurs to account for search operation in the user tree [IN FUNCTION PRINTUSERTRANSACTIONS] (n is the number of user nodes in the tree)

8) CreateHash

1. Creates a Hash Value for the Block
2. Hash value is a string of 32 bits
3. First 4 bits are integers,generated by HashTransaction Function
4. In HashTransaction, we add ((Receiver's Id + Sender's Id)/amount transferred from Sender to Receiver) of each transaction present in that block and finally return this sum
5. This sum is stored as characters in the first 4 bits of our hash string
6. The next 12 bits of the hash string are formed from the the previousblockhash value of the current block,performing bitwise XOR & shift operation on the prevblock hash.
7. The next 10 bits of the Hash string are formed by HornerRule which generates a hash value from previousblockhash and nonce of the block. Here, we also check for overflow.
8. The leftover 6 bits are also formed by HornerRule where a Hash value is generated from previousblockhash and sum of blocknumber and nonce of the current block.
9. Time complexity : o(1)

9) Utility Functions

1. Height: Returns height of UserNode in the AVLTree
2. CreateUserNode: Creates empty User Node, with given UID

3. Search: Recursively searches for a User in the LeftTree and the RightTree
4. CreateHash: Generates PrevBlockHash
5. CreateChain: Creates empty BlockChain
6. CreateBlock: Creates empty Block
7. AddToChain: Updates Block information and adds it to the BlockChain.
8. CreateNode: Creates an empty Node for a Transaction List.
9. CreateHead: Creates an empty Transaction List.
10. AddTransactionToList: Creates a Node and adds it to the Transaction List
11. Hornersrule():Horner rule implmentation for calculating string hash→a subroutine for create hash functions
12. HashTransaction()-a hash function implmentation as a sub-routine for create hash for block, involving transaction history and nonce of the block.

DIVISION OF WORK

1) Karmanjyot Singh (2020101062)
• AddUser
• User Interface (and data structures used in the project)
• Utility Functions
• Hash Function
2) Pranathi (2020101083)
• CreateBlock
• Hash Function
3) Aryan (2020102032)
• ValidateBlockChain
• Hash Function
4) Ruchitha (2020101093)
• Attack
• Hash Function
5) Shruti (2020102053)
• Transact
• Hash Function