A circular emblem with a shield and text

AI-generated content may be incorrect.

CS-221: Data Structures and Algorithms

Semester Project

**Deliverable #2:** Initial Implementation (19/10/25)

**Course Instructor:** Dr. Zubair Ahmad

**Group Members:**

1. Abdul Raffay Gill (2024019)
2. Syeda Maheen Kazmi (2024627)
3. Youraj Ali Karim (2024666)

**Repository Link:** [**https://github.com/syedamaheenkazmi172/Blockchain\_implementation\_using\_DSA.git**](https://github.com/syedamaheenkazmi172/Blockchain_implementation_using_DSA.git)

Mini Implementation of Blockchain using Data Structures and Algorithms

In this initial implementation phase, we focused on integrating the fundamental data structures taught in our course, such as linked lists and queues, into the blockchain simulation. This report explains how these structures have been implemented and what specific roles they play within the system. It also highlights the basic framework we have established for user interaction, including user login, wallet access, and data handling through file input, which will serve as the foundation for later, more advanced blockchain operations.

1. **Linked List Integration:**

The linked list serves as the foundation of our blockchain structure. Each block in the chain is represented using a *struct Block*, which contains data fields such as *index*, *prevHash*, *nonce*, *hash*, and *fee*, along with pointers *next* and *prev* to connect adjacent blocks. This interconnection allows sequence-based traversal which ensures that every block maintains a direct link to both its predecessor and successor, like how a real blockchain maintains continuity between blocks. The *BlockChain* class, which will be completely implemented when the mining process is being implemented, manages these nodes through the *head* and *previous* pointers, which respectively represent the starting and last added blocks in the chain. This structure allows easy insertion of new blocks while maintaining the linked nature of the blockchain.

Within the linked list implementation, three core functions have been made; the *noncemaker()*, *hashmaker()* and *transaction()* functions. The *noncemaker()* function generates a random 6-digit number for each transaction. The nonce is then used as a parameter in the *hashmaker()* function which hashes the nonce using a simple hashing function. Finally, the *transaction()* function, which takes the transaction fee as a parameter, initializes the block and calls the *nonce()* function to create a nonce, calls the *hashmaker()* function to create the hash and initializes the data fields of the block with all the required data. The returned block is used in the queue implementation.

1. **Queue Implementation:**

The queue is used to temporarily store blocks that are waiting to be added to the blockchain, effectively simulating the transaction pool in real blockchain systems. Implemented using a *struct queue*, it maintains two pointers, *front* and *rear*, for efficient enqueue and dequeue operations. When a new block is created through the *transaction()*, it is enqueued at the rear of the queue. Additionally, the queue is sorted in descending order based on the transaction fee, giving higher-priority transactions preference for inclusion in the chain, thus forming a simple *priority queue*. The queue structure includes essential operations such as *enqueue()*, *dequeue()*, *isEmpty()*, and *getFront()*, making it a dynamic and organized way to manage pending transactions before mining.

The queue and linked list work together to represent the full lifecycle of blockchain transactions. When a user performs a transaction, a new *Block* is created and added to the queue through the *enqueue()* function. This queue acts as a waiting area for unconfirmed transactions, temporarily storing blocks based on their transaction fees in descending order. During the mining phase, these queued blocks will be sequentially transferred from the queue into the linked list structure, where each block will be permanently linked to the previous one using the *prevHash* value. This connection ensures that once a transaction is mined, it becomes an immutable part of the blockchain, while the queue continues to handle incoming transactions efficiently.

1. **User Interaction and Input Handling:**

The input handling in this phase focuses on enabling user interaction through file-based login and wallet access. The *User* class manages all input operations related to authentication and user data. When the program starts, it prompts the user to log in either as a regular user or as a miner. For user login, it takes the username and private key as input, then verifies them by reading from two separate files, *login.txt* (for usernames) and *Login\_p.txt* (for private keys). If the credentials match, the user is granted access to their wallet, where multiple options such as creating transactions, viewing the blockchain, and checking balances are displayed. This implementation provides a basic yet functional method of handling input without a database, forming the foundation for more secure and interactive user operations in later stages.