DSA Project Description

Welcome to your Data Structures and Algorithms **Project Assignment**! This is your opportunity to apply the concepts you've learned this semester to a hands-on, real-world-inspired project. Working in **groups of 5-7**, you will select *one* project from the list below, implement it, and present your work.

This assignment will test your technical skills, creativity, and ability to collaborate

Submission Guidelines

- **Deadline**: Submit your project at least **1 day before your presentation date** (any day before that is accepted).
- **Format**: Upload all files to your group's designated repository or file upload via email (using GitHub earns you extra point).

Bonus Opportunity

Groups that complete **more than one project** will earn **bonus marks**! To qualify:

- All projects must be fully functional and meet all requirements.
- Clearly document how you prioritized tasks and divided work.

Final Presentation

- **When**: After your final exam (**specific dates/times will be announced**).
- **What to Prepare**:
  - A **10-minute presentation** covering:
    - Your project's purpose and design.
    - Key challenges and how you overcame them.
    - A live demo or video walkthrough.
    - Lessons learned and potential improvements.

- **Q&A**: Be ready to answer technical questions from instructor/lab assistants.

Grading Criteria

- **Functionality**: Does the project work as intended?
- **Code Quality**: Is the code clean, efficient, and well-documented?
- **Creativity**: Did you add unique features or optimizations?
- **Presentation**: Clear explanation and demo quality.

1. Sorting Algorithm Visualizer

A **Sorting Algorithm Visualizer** is a tool that demonstrates how sorting algorithms reorganize data (e.g., numbers, bars) step-by-step using animations. It helps users visually compare how different algorithms (e.g., Bubble Sort, Quick Sort) work and understand their efficiency.

How Does It Work?

1. **Data Representation**:
   - Display data as bars (height = value) or numbers.
   - Highlight elements being compared/swapped during sorting.
2. **Algorithms**:
   - Implement common algorithms (e.g., Bubble, Quick, Insertion).
   - Animate each step (e.g., swaps).

Implementation Guide

1. **Choose a Framework**
2. **Generate Random Data**
3. **Implement Sorting Algorithms**
4. **Visualization**

Example Workflow

1. Generate random data (e.g., `[5, 2, 9, 1]`).
2. Select "Bubble Sort" and click "Play".
3. Watch adjacent elements swap until the data is sorted.

Deliverables

1. Source code with at least 3 algorithms.
2. Documentation: How to run, algorithm explanations

2. Mini Blockchain

Disclaimer: This projects does not include the full implementation of the blockchain system, it lacks core functionalities (like mining, POW and etc...) but its enough to get you started.

This project is a minimal blockchain implementation where blocks are cryptographically linked to their predecessors. Each block contains data, a timestamp and a hash of the previous block,

Why Build This?

1. **Understand Blockchains**: Learn core concepts like hashing, and immutability.
2. **Hands-On Cryptography**: Use SHA-256 to link blocks and ensure integrity.
3. **Tamper-Evident Design**: See how changing a block breaks the chain.

How Does It Work?

1. **Block Structure**:

   Each block contains:

   - **Index**: Position in the chain (eg: Block 0, Block 1).
   - **Timestamp**: When the block was created.

- **Data**: Arbitrary data (e.g., strings).
- **Previous Hash**: SHA-256 hash of the previous block.
- Hash: The SHA-256 hash of the block's contents (index, timestamp, data, previous hash)

2. Linking Blocks
   - The **previous hash** field connects each block to its predecessor.
   - Example:
     - Block 1's `previous_hash` = Block 0's hash.
     - Block 2's `previous_hash` = Block 1's hash
3. **Validation**
   - Verify that every block's hash matches its calculated hash.
   - Verify that each block's `previous_hash` matches the previous block's hash.

Implementation Guidelines

1. **Define Block Components**
   - Create a `Block` class/structure with fields for index, timestamp, data, previous hash, and hash.
2. **Create the Genesis Block (The first Block)**
   - Manually construct the first block (index = 0) with arbitrary data and a `previous_hash` of "0"
3. Build the Chain
   - Add newly mined blocks to the chain, ensuring each references the previous block's hash
4. Validate the Chain
   - Loop through the chain and check:
     1. Each block's hash is valid.
     2. Each block's `previous_hash` matches the prior block's hash

Example Workflow

1. **Initialize the Blockchain**:
    - Create the genesis block:
        - `Index`: 0, `Data`: "Genesis Block", `Previous Hash`: "0", `Hash`: `d3f6...`.
2. **Mine Block 1**:
    - `Data`: "Alice paid Bob $10".
    - Capture timestaamp
    - Add to chain: Previous Hash = `d3f6...`, Hash = `00a1...`.
3. **Mine Block 2**:
    - `Data`: "Bob paid Charlie $5".
    - Capture timestamp
    - Add to chain: Previous Hash = `00a1...`, Hash = `005c...`.
4. **Tamper with Block 1**:
    - Change data to "Alice paid Bob $1000".
    - Recalculate Block 1's hash.
    - Block 2's `previous_hash` now points to an invalid hash.
5. **Validation Fails**:
    - The chain is invalid because Block 1's hash changed and Block 2's `previous_hash` no longer matches.

Deliverables

Source code + executable (if applicable).
Documentation: How to run, algorithm explanations

3. LRU Cache

An **LRU (Least Recently Used) Cache** is a data structure that stores a limited number of items (like key-value pairs) and automatically removes the *least*

recently used item when the cache is full.

Why Use an LRU Cache?

- **Speed**: Retrieves data in constant time (`O(1)`) for `get` and `put` operations.
- **Efficiency**: Ensures the cache doesn't exceed its capacity by evicting unused items.
- **Real-World Uses**:
  - Web browsers caching recently visited pages.
  - Databases caching frequently queried results.
  - Operating systems managing memory pages.

How Does It Work?

Structure

- **Hash Map**: Stores keys and maps them to nodes in a linked list for `O(1)` lookups.
- **Doubly Linked List**: Tracks the *order of usage*.
  - **Most Recently Used (MRU)**: Items at the **head** of the list.
  - **Least Recently Used (LRU)**: Items at the **tail** of the list.

Operations

- `get(key)`:
  - If the key exists:
    1. Move its node to the **head** (mark it as "recently used").
    2. Return the value.
  - If the key doesn't exist: Return `-1` or a default value.
- `put(key, value)`:
  - If the key exists:

1. Update its value.
2. Move the node to the **head**.

- If the key doesn't exist:
  1. Create a new node and add it to the **head**.
  2. If the cache is **full**:
     - Remove the node at the **tail** (LRU item).
     - Remove its key from the hash map.

Example Workflow

1. **Initial State** (Capacity = 3):
   - Cache: Empty.
2. `put(1, "A")`:
   - Add `1 → "A"` to the head.
3. `put(2, "B")`:
   - Add `2 → "B"` to the head.
4. `get(1)`:
   - Move `1 → "A"` to the head (now MRU).
5. `put(3, "C")`:
   - Add `3 → "C"` to the head.
6. `put(4, "D")` (Cache is full):
   - Remove LRU item (`2 → "B"` at the tail).
   - Add `4 → "D"` to the head.

Deliverables

1. Working Code
2. Documentation explaining the metadata format.

4. Chunk-Link

Chunk-Link is a file storage and reconstruction system that splits a file into chunks, stores them in a linked list structure (each chunk points to the next), and reconstructs the file by traversing the list.

Why Build This?

- **Conceptual Learning**: Reinforces linked list mechanics (pointers, traversal)
- **Practical Use**: Simulates peer-to-peer file sharing (e.g., BitTorrent's sequential logic).
- **Error Handling**: Teaches data integrity checks (e.g., checksums).

How Does It Work?

1. **Node Creation**
    - Each node stores:
        - `data`: The file chunk (bytes).
        - `next_checksum`: SHA-256 hash of the *next* node's data.
        - `next_node`: Pointer to the next node.
2. **File Splitting**:
    - Split a file into fixed-size chunks (e.g., 1MB each).
    - Each node will contain a single chunk of the data
3. **Linked List Structure**:
    - Each chunk includes metadata pointing to the next chunk.
    - eg `metadata -> chunk_1 -> chunk_2 -> chunk_3 -> ...`
4. **Reconstruction**:
    - Start with the first chunk, download sequentially, and merge chunks.
5. **Traversal & Verification:**
    - Start at the head node.

- Verify the `next_checksum` matches the actual hash of the `next_node.data`
- If mismatch: Raise error (data corruption detected).

Key Components to Implement

- **Chunking Logic**: Split files and generate metadata.
- **Linked List Traversal**: Follow pointers to fetch chunks in order.
- **Validation**: Add checksums (e.g., SHA-256) to detect corruption.

Example Workflow with Byte Data

1. **Original File Data**: (`"Hello World"` in bytes)

```
file_data = b"\x48\x65\x6c\x6c\x6f\x20\x57\x6f\x72\x6c\x64"
```

2. **Split into Chunks** (example chunk size = `5 bytes`)

```
chunks = [
    b"\x48\x65\x6c\x6c\x6f",   # "Hello"
    b"\x20\x57\x6f\x72\x6c",   # " Worl"
    b"\x64"                     # "d"
]
```

3. **Create Linked Nodes**:
   - Node-0
     - `data` = **`b"\x48\x65\x6c\x6c\x6f"`**
     - `next_checksum` = sha256 checksum of next data (`b"\x20\x57\x6f\x72\x6c"`)
     - `next_node` = `None` (since we havent created the next node)
   - Node-1
     - `data` = **`b"\x20\x57\x6f\x72\x6c"`**
     - `next_checksum` = sha256 checksum of next data (`b"\x64"`)
     - `Node-0.next_node` = `Node-1`

    - `next_node` = `None`
- Node-2
    - `data` = `b"\x64"`
    - `next_checksum` = `None` (since there is no more data)
    - `Node-1.next_node` = `Node-2`
    - `next_node` = `None`

Deliverables

1. Working Code
2. Documentation explaining the metadata format.

Bonus Points for All Projects

1. **GitHub Repository**: Clean code, commit history, and tags.
2. Team work
3. Early submission