**Data Structures used in Project and Comparison with other Data Structures**

Data structures used in Project:

* Merkel Tree
* Hash Pointer

1. Merkle Tree:

* Merkle tree is used ṭo generate cryptographic hash of the data stored in block.
* Merkle trees allow for efficient verification of the integrity of large datasets. By storing the hashes of data blocks in a hierarchical structure, a single hash (the root) can represent the entire dataset, making it easy to verify if any part of the data has been altered.
* They enable compact proofs of inclusion or exclusion. To verify that a specific piece of data is part of the dataset, you only need to provide the hashes along the path from the leaf node to the root, rather than the entire dataset.
* Merkle trees are a foundational component in many blockchain systems, such as Bitcoin and Ethereum, where they are used to securely manage transactions and ensure data consistency across distributed nodes.
* Why Merkle tree?? Why not Verkle Tree?
* Merkle trees enable the easy analysis of whether data is accurate or not.
* Merkle trees are simpler and easier to implement.
* A Merkle tree compresses the required data for operation into much smaller sizes than storing every transaction individually.
* Merkle trees generally require less memory overhead compared to Verkle trees, which can be beneficial in resource-constrained environments or applications that require fast execution.
* Verkle trees have unfortunately not been implemented in blockchains yet. However, their fundamentals are sound and the future is promising.

1. Hash Pointers:

* A Hash pointer is a pointer where data is stored and with the pointer, the cryptographic hash of the data is also stored.
* So, it points to the data and also allows us to verify the data.
* Structure:

Typedef struct hashpointer {

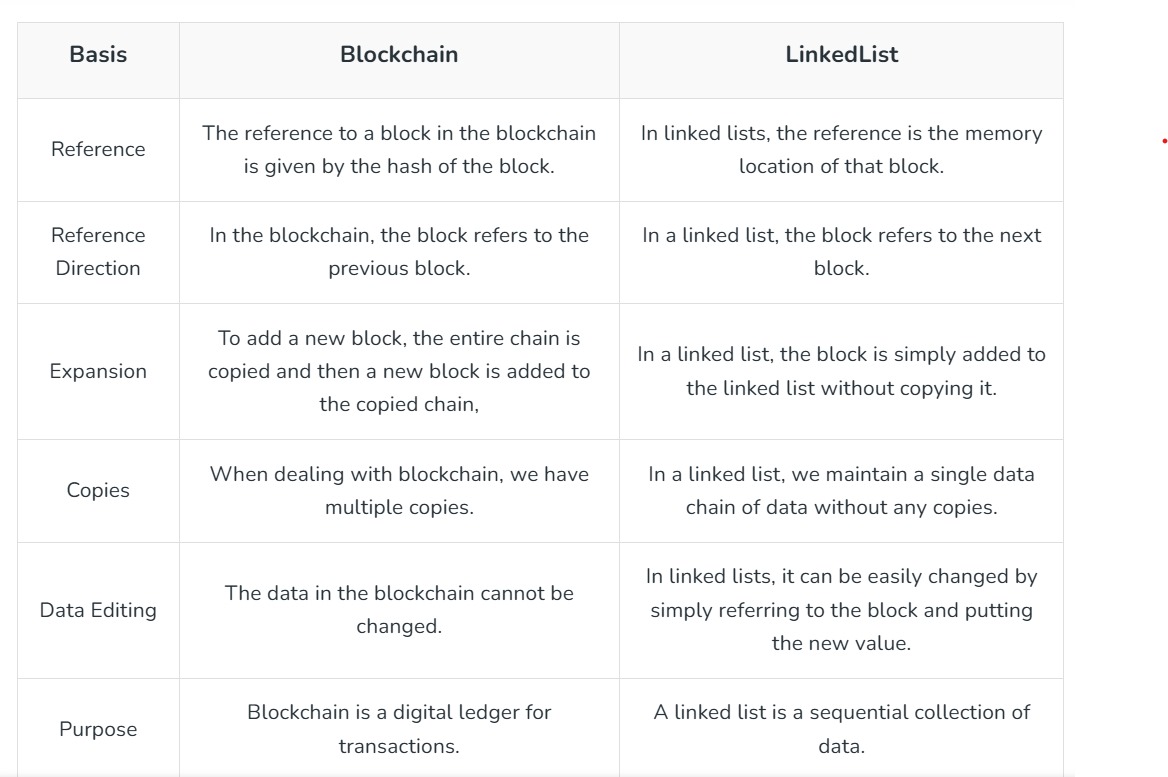
Node \*pointer;

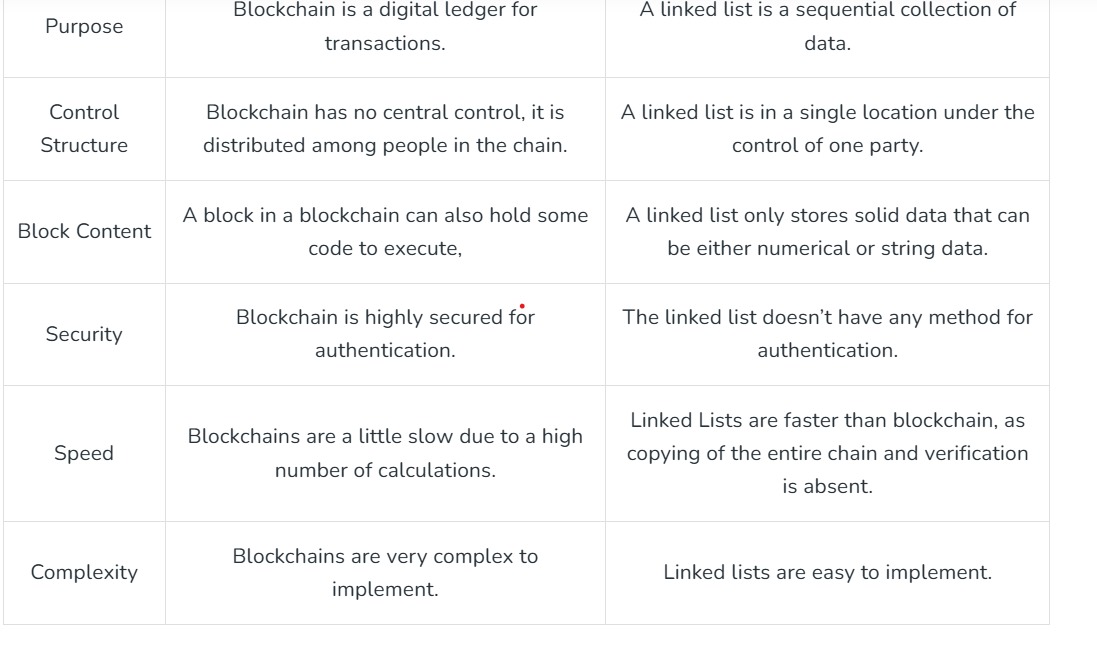
String hash;

}HP;

* Hash Pointers over simple pointers:
* Simple pointers don’t verify data integrity, so changes to data go unnoticed.
* Altered blocks wouldn’t invalidate the chain, leading to inconsistencies, by using simple pointers.
* Hash pointers invalidate affected blocks.
* Manipulation risks increase without the security of hash pointers, threatening decentralization.
* Simple pointers complicate reaching agreement among nodes, undermining blockchain reliability.

Difference between Blockchain and Linked List





1. Why we are using SHA256 ?? why not other hashing algorithm?

* SHA-256 is part of the SHA-2 family and offers strong security against attacks. Its design minimizes the risk of collisions

1. SHA-1:

- Security: Considered weak due to vulnerabilities that allow for collision attacks. It's no longer recommended for security-sensitive applications.

- Performance: Faster than SHA-256, but the speed comes at the cost of security.

- Use Cases: Previously used in digital signatures and certificates, but largely replaced by SHA-2 and SHA-3.

2. SHA-512:

- Security: Provides a similar level of security as SHA-256 but has a larger hash size (512 bits), making it more resistant to certain types of attacks.

- Performance: Generally faster on 64-bit processors, but can be slower on 32-bit systems compared to SHA-256.

- Use Cases: Used in applications requiring high security with larger hashes, but not as common in blockchain as SHA-256.

3. SHA-3:

- Security: Offers strong security with a different underlying structure (Keccak), making it resistant to many known attacks.

- Performance: Generally slower than SHA-256 for hashing, but it can be more efficient for some specific use cases due to its versatility.

- Use Cases: Suitable for applications requiring future-proofing against potential vulnerabilities in SHA-2.

4. RIPEMD-160:

- Security: Offers good security but is less tested than SHA-2. It’s generally considered safe but not as robust as SHA-256.

- Performance: Comparable to SHA-256, but not as widely adopted.

- Use Cases: Used in Bitcoin addresses, often in combination with SHA-256.

5. BLAKE2:

- Security: Offers a high level of security and is designed to be faster than MD5 and SHA-2 while maintaining security.

- Performance: Generally faster than SHA-256, making it attractive for high-performance applications.

- Use Cases: Used in various applications, but not as common in blockchain compared to SHA-256.

##Summary :

- SHA-256: Strong balance of security and performance, widely adopted in blockchain.

- SHA-1: Outdated and insecure, not recommended.

- SHA-512: Higher security, larger output; less common in blockchain.

- SHA-3: Future-proof, but generally slower than SHA-256.

- RIPEMD-160: Good security, but not as widely used as SHA-256.

- BLAKE2: Fast and secure, but less prevalent in blockchain.

##SHA-256 remains the standard in many blockchain applications due to its proven security, efficiency, and community trust.

When considering data structures for efficiently searching through a blockchain, several options stand out. Here's a comparative study of the most relevant data structures:

### 1. \*\*Linked List\*\*

- \*\*Description\*\*: Each block points to the next block, forming a sequential structure.

- \*\*Time Complexity\*\*: O(n) for search (linear search).

- \*\*Space Complexity\*\*: O(n), where n is the number of blocks.

- \*\*Pros\*\*: Simple to implement; easy to append new blocks.

- \*\*Cons\*\*: Inefficient for searching; requires traversal of all blocks.

### 2. \*\*Array/List\*\*

- \*\*Description\*\*: Blocks are stored in an array, allowing indexed access.

- \*\*Time Complexity\*\*: O(1) for access, O(n) for search (if unsorted).

- \*\*Space Complexity\*\*: O(n).

- \*\*Pros\*\*: Fast access by index; straightforward implementation.

- \*\*Cons\*\*: Inefficient for searching unless sorted; resizing is costly.

### 3. \*\*Hash Table\*\*

- \*\*Description\*\*: Blocks are stored in a hash table using a unique key (e.g., block index).

- \*\*Time Complexity\*\*: O(1) average for search.

- \*\*Space Complexity\*\*: O(n) plus overhead for the table.

- \*\*Pros\*\*: Fast searches; efficient retrieval.

- \*\*Cons\*\*: Collisions can affect performance; requires a good hash function.

### 4. \*\*Binary Search Tree (BST)\*\*

- \*\*Description\*\*: Blocks are stored in a binary tree based on keys (e.g., block index).

- \*\*Time Complexity\*\*: O(log n) for search (in a balanced tree).

- \*\*Space Complexity\*\*: O(n).

- \*\*Pros\*\*: Allows ordered traversal; can be balanced (e.g., AVL tree) for better performance.

- \*\*Cons\*\*: Can degrade to O(n) in the worst case if unbalanced; more complex to implement.

### 5. \*\*Trie\*\*

- \*\*Description\*\*: A tree-like data structure that stores blocks based on prefixes of their keys.

- \*\*Time Complexity\*\*: O(k), where k is the length of the key.

- \*\*Space Complexity\*\*: O(n \* m), where m is the maximum key length.

- \*\*Pros\*\*: Fast prefix searches; useful for search operations based on keys.

- \*\*Cons\*\*: More complex implementation; can consume more memory.

### 6. \*\*B-tree / B+ tree\*\*

- \*\*Description\*\*: A self-balancing tree data structure that maintains sorted data and allows for efficient insertion, deletion, and search.

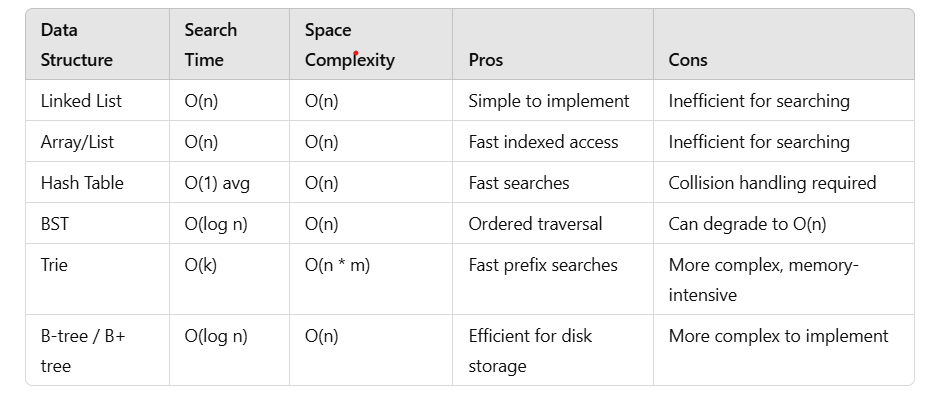
- \*\*Time Complexity\*\*: O(log n) for search.

- \*\*Space Complexity\*\*: O(n).

- \*\*Pros\*\*: Efficient for disk storage; can handle large amounts of data; keeps data sorted.

- \*\*Cons\*\*: More complex to implement than other structures.

Summary table



For searching algorithm we can use following data structures :

1. Linked List: Simple Traversal through each block and searching for each hash (merkle tree)

Time complexity = > O(n)

1. Hash Table : Fast search. Time complexity => O(1)…but collision may occur
2. Binary Search Tree : Time complexity => O(log n)
3. Can also use Trie or B-/B+ Tree but they are more complex to implement