ECE 250 -Project2

Hashing

Design Document

Abdullah Al Amaan, UW User ID= aaamaan

November 4th, 2024

Class Design:

In this project, I implemented two primary classes:

FileBlock:

Purpose:

This class represents a single file block in the hash table. Each file block stores an ID, a payload and a checksum. The checksum verifies data integrity when a block is stored or retrieved. It also supports corruption operations by updating the payload.

Data Structures:

- **unsigned int id:** Stores the unique identifier for each file block.
- **char payload[500]:** An array of characters that has been statically allocated to hold up to 500 bytes of payload data.
- **int checksum:** An integer value used for data integrity verification, calculated as a modulo sum of the payload.
- **FileBlock* next:** A pointer for separate chaining in the hash table, allowing the formation of a linked list in case of hash collisions.

Why I Used This:

- **Static Array for Payload:** Using a static array of size 500 keeps memory allocation fixed for each file block, ensuring memory efficiency.
- **Checksum:** The formula that I have used allows data integrity checks to verify payload corruption.
- **Separate Chaining:** This technique is used to handle hash collisions in the hash table when multiple file blocks map to the same index.

Design Decisions:

- **Data Protection:** The member variables are private to prevent direct manipulation, and access is provided through member functions.

- **Efficient Memory Management:** The **next** pointer allows chaining in order to facilitate effective collision solution without expanding the size of the primary table.

HashTable:

Purpose:

This class manages a hash table that stores **FileBlock** objects. It uses either open addressing with double hashing or separate chaining according to need. The hash table allows insertion, deletion, search, corruption, and validation of file blocks.

Data Structures:

- **FileBlock** table:** A pointer array representing the hash table, where each slot may point to a **FileBlock** or **nullptr**.
- **unsigned int size:** The total capacity of the hash table, set during initialization.
- **bool useOpenAddressing:** A Boolean flag indicating whether open addressing or separate chaining is used.

Why I Used This:

- **Array of Pointers:** This table allows storage of **FileBlock** pointers, providing space efficiency for both open addressing and separate chaining.
- **Open Addressing and Separate Chaining:** Open addressing minimizes memory overhead, while separate chaining allows for linked list chains, reducing clustering issues.

Design Decisions:

- **Collision Resolution:** The class provides two options for handling collisions, allowing flexibility.
- **Data Integrity:** Each file block has a checksum to ensure data consistency during store and retrieve operations.
- **Memory Management:** The **clearTable** function deallocates memory for all **FileBlock** objects and linked lists, preventing memory leaks on reinitialization and destruction.

Function Design:

FileBlock Functions:

- **calculateChecksum():** Computes the checksum as a modulo sum of payload bytes. The runtime is O(1) since the payload size is fixed at 500 characters**.**
- **corrupt(const std::string& new_data):** Clears the current payload and replaces it with **new_data** without recomputing the checksum. The runtime is O(n), where n is the length of **new_data.**
- **validateChecksum():** Compares the current checksum with a newly computed checksum to verify data integrity. The runtime is O(1).

HashTable Functions:

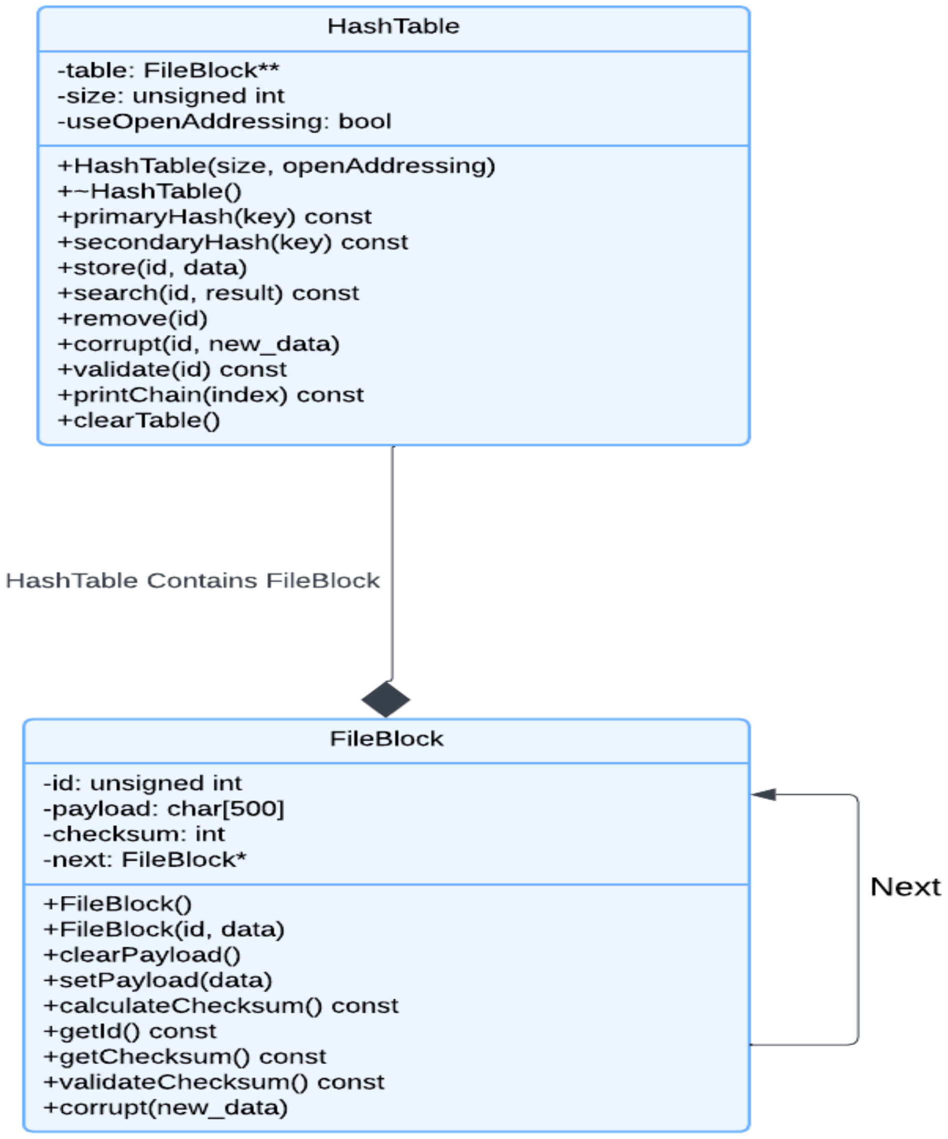
- **primaryHash(unsigned int key):** Computes the primary hash by taking key % size. This is an O(1) operation.
- **secondaryHash(unsigned int key):** Computes a secondary hash for open addressing by using key / size % size. The result is adjusted to ensure an odd value, maintaining consistent spacing in double hashing. Runtime is O(1).
- **store(unsigned int id, const std::string& data):** Stores a file block with the given ID and data. The runtime is O(1) for open addressing and O(n) for separate chaining, where n is the number of collisions at the target index.
- **search(unsigned int id, FileBlock*& result):** Searches for a file block by ID. Returns the index if found and sets result to the **FileBlock** pointer. The runtime is O(1) for open addressing and O(n) for separate chaining, where n is the number of elements in the chain.
- **remove(unsigned int id):** Removes a file block by ID. In open addressing, it sets the slot to **nullptr**. In separate chaining, it removes the specific node in the chain. The runtime is O(1) for open addressing and O(n) for separate chaining.
- **corrupt(unsigned int id, const std::string& new_data):** Locates the block and replaces its payload with new_data without recomputing the checksum. Runtime is O(1) for open addressing and O(n) for separate chaining.
- **validate(unsigned int id):** Validates the checksum of a block. Runtime is O(1) for open addressing and O(n) for separate chaining.
- **clearTable():** Deletes all **FileBlock** instances in the hash table, ensuring memory is deallocated. The runtime is O(m + n), where m is the table size, and n is the total number of nodes in all chains.

Constructors and Destructors:

- **FileBlock Constructor:** Initializes the **FileBlock** with either a default or specific ID and data, clearing the payload and setting the checksum.
- **HashTable Constructor:** Allocates memory for the table and sets each entry to **nullptr.**
- **clearTable() in Destructor:** Ensures that all dynamically allocated **FileBlock** objects are deleted upon hash table reinitialization or destruction, preventing memory leaks.

Core Cleanup: In the main function, dynamically allocated memory for the hash table is deleted when the **NEW** command is used to create a new hash table and during the **EXIT** command to ensure proper cleanup.

UML Diagram:



Conclusion:

The design of the **FileBlock** and **HashTable** classes provides a structured approach to storing and managing file blocks using hashing techniques. This system can handle hash collisions using either open addressing or separate chaining, providing flexibility for different use cases. Efficient memory management and data protection ensure that data integrity is maintained throughout operations while preventing memory leaks.