Hashing Checksum (Project 2)

Overview

This project implements a hash table with two collision resolution strategies: open addressing and separate chaining. The design emphasizes flexibility, memory efficiency, and optimized collision handling. This document outlines the critical design decisions, class structures, and algorithms used in the implementation.

Class Design

- Reason for using two classes

  The decision to use two classes, Block and HashStructure, based on the principles of separation of concerns. Each class encapsulates its own data and operations, promoting modularity and east of testing. For instance, the Block class handles the core data and integrity functions of each hash table entry, while HashStructure manages the overall hash table structure and collision resolution.

  Block class: This class is designed to represent individual data entries in the hash table. It includes an id, a payload (a vector of characters), and a checksum to ensure data integrity. This encapsulation allows each block to manage its own integrity checks and data storage independently, facilitating modular testing and data handling within HashStructure.

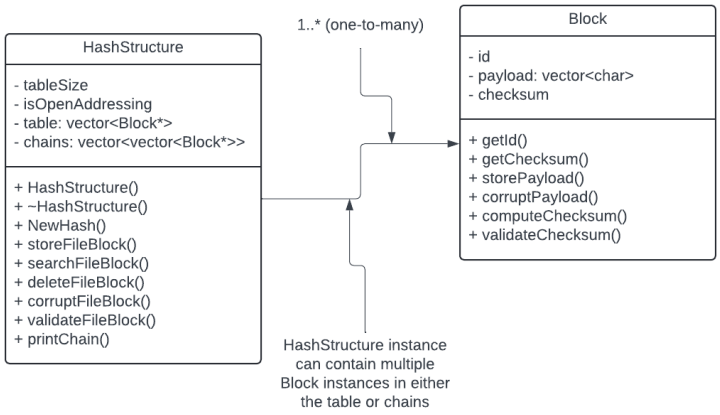
  HashStructure class: This class represents the hash table and supports both open addressing and separate chaining as collision resolution methods. By maintaining separate containers (table and chains) for the two strategies, HashStructure provides a flexible and efficient data storage structure that can switch between methods based on requirements. This separation simplifies the logic and enables clear distinctions between the two collision strategies.

- Private Attributes of the two vector arrays

  - table vector holds Block* pointers for open addressing, where each slot can either contain a Block or be nullptr if empty. Making it private ensures that Block objects are only accessed and modified through HashStructure methods, preserving data integrity.
  - chains vector is a vector of vectors of Block* pointers, representing chains for separate chaining. Keeping this attribute private allows HashStructure to handle its memory management and prevent unauthorized modifications.

UML Design

Each HastStructure can manage an arbitrary number of Block objects, as indicated by the 1-to-* (one to many) association.



Function Design

- Key Functions:
    - Void Block::storePayload: Updates the payload and recalculates the checksum to ensure data integrity, maintaining the Block's state.
    - Std::string HashStructure::storeFileBlock: Inserts a Block into the hash table. In open addressing, secondaryHash is used to find empty slots, while in separate chaining, the Block is appended to the chain.
    - std::string HashStructure::deleteFileBlock: Finds and deletes a Block by id, freeing allocated memory as it gets deleted. In open addressing, it nullifies the slot, while in separate chaining, it removes the Block from the chain.

Runtime

- It is generally O(1) runtime. For primaryHash and secondaryHash, both has O(1) runtime since they preform a single modulo operation and basic arithmetic to compute the index and determine a step size for probing. For open addressing in the storeFileBlock function, the best case would be O(1) if the slot at the primary hash index is empty and O(n), where n is the size of table vector, which is worst case, if the table is full and all slots must be probed. For separate chaining in the same function, the best cast would be the same as the open addressing, but the worst case would be O(k), where k is the number of elements in the chain at the hashed index. The time complexity of separate chaining depends on the chain length, not size of table.