Class Design

FileBlock stores the actual data: a ID, a 500-byte char array for the payload, a checksum, and a boolean to track if it's occupied. I made the payload a fixed array instead of using dynamic memory because the spec says all blocks are exactly 500 bytes, so there's no point in allocating dynamically. The occupied flag is needed for open addressing to tell the difference between an empty slot and one that was never used.

HashTable is the base class that has the two hash functions h1 and h2 from the spec. I made it abstract with pure virtual functions (gpt suggested) so both types of hash tables can inherit from it. This way the main program can just use one pointer type for both implementations.

OpenAddressingHash uses an array of FileBlocks and a separate boolean array called deleted. I made a separate deleted array instead of putting the deletion flag inside FileBlock because when you're searching with double hashing, you need to keep probing past deleted spots but stop at empty spots. If you don't track this separately, deletions break the probe chain.

SeparateChainingHash uses an array of Node pointers where each Node has a FileBlock and a next pointer. I kept all chains sorted by ID because the PRINT command needs to output IDs in order. I could have sorted them every time PRINT is called but that seemed wasteful if you call PRINT multiple times. Keeping them sorted more work per insert but way faster.

I used singly-linked lists instead of doubly-linked because you only need to go forward through the chain and the extra pointers for going backwards just waste memory.

UML Diagram

HashTable	OpenAddressingHash	SeparateChainingHash
# m: unsigned int	- table: FileBlock[] - deleted: bool[]	- table: Node**
+ HashTable(size: unsigned int) # h1(k: unsigned int): unsigned int # h2(k: unsigned int): unsigned int + store(id: unsigned int, data: string): bool + search(id: unsigned int): int + deleteBlock(id: unsigned int): bool + corrupt(id: unsigned int, data: string): bool + validate(id: unsigned int): string	+ OpenAddressingHash(size: unsigned int) + store(id: unsigned int, data: string): bool + search(id: unsigned int): int + deleteBlock(id: unsigned int): bool + corrupt(id: unsigned int, data: string): bool + validate(id: unsigned int): string	+ SeparateChainingHash(size: unsigned int) + store(id: unsigned int, data: string): bool + search(id: unsigned int): int + deleteBlock(id: unsigned int): bool + corrupt(id: unsigned int, data: string): bool + validate(id: unsigned int): string + printChain(index: unsigned int): void

Open Addressing and Separate Chaining inherit from HashTable

FileBlock	Node
- id: unsigned int - payload: char[500] - checksum: int - occupied: bool	+ block: FileBlock + next: Node*
+ FileBlock() + setData(id: unsigned int, data: char*): void + corruptData(data: char*): void + computeChecksum(): int + validateChecksum(): bool + getId(): unsigned int + isOccupied(): bool + clear(): void	+ Node(id: unsigned int, data: string)

OpenAddressingHash contains FileBlocks, SeparateChainingHash contains Nodes, and each Node contains a FileBlock

Function Design

OpenAddressingHash::store(): Uses double hashing to find a spot. Calculates h1(k) for the starting position and h2(k) for the step size. Then it keeps probing with (pos + step) mod m until it finds an empty or deleted slot. It also checks if the ID already exists while probing. Returns true if it worked, false if the table is full or the ID is already there.

SeparateChainingHash::store(): First it checks if the ID already exists by going through the chain. Then it inserts the new node in sorted order. There's three cases: empty chain (just set it as the head), new ID is smaller than the head (insert at front), or it goes somewhere in the middle or end (traverse until you find the right spot).

search(): Both versions return the position in the table (or chain index for separate chaining) instead of returning the actual FileBlock. This way you can't accidentally modify the data from outside. Returns -1 if not found which is a standard way to show failure.

deleteBlock(): For open addressing, it finds the block using search() then marks it as deleted. For separate chaining, it finds the node and removes it from the linked list.

computeChecksum(): Adds up all 500 bytes in the payload and takes mod 500. I had to cast to unsigned char when adding because regular chars can be negative and that messes up the sum.

Runtime Analysis

Assumptions: For any hash value, there's at most m collisions, m << T (total number of things inserted), and m is O(1).

Open Addressing:

STORE: Calculating h1(k) and h2(k) is just basic math so it's O(1). With our assumption that there's at most m collisions and m = O(1), we only probe O(1) times. Each probe just checks if the spot is empty and compares IDs which is O(1). So overall STORE is O(1) average.

SEARCH: Same as STORE, compute the hash functions in O(1), probe O(1) times, do O(1) work each time. Total is O(1) average.

DELETE: Calls search() which is O(1), then just clears the block and sets the deleted flag which are both O(1). So DELETE is O(1).

Separate Chaining:

STORE: Computing h1(k) is O(1). Then you go through the chain to check for duplicates and find where to insert. The chain has at most m elements and m = O(1) by assumption, so going through it is O(m) = O(1). Making the new node and linking it is O(1). Total is O(1) average.

SEARCH: Hash is O(1). Going through a chain of length m = O(1) is O(m) = O(1). Each comparison is O(1). So SEARCH is O(1) average.

DELETE: Hash is O(1), traverse chain of length O(1), and unlinking the node is just changing a pointer which is O(1). Total is O(1) average.

[1] OpenAI, “ChatGPT (GPT-5),” chat.openai.com, accessed Oct. 5 2025.
Spelling, grammar, & formatting/organization were corrected with assistance from ChatGPT. Suggestions on improvements were also taken from GPT and implemented by myself.