1. Class Design

1.1 HashTable Class Design

The `HashTable` class is simply a base class from which both hashing implementations are derived.

1.1.1 Data Structure Choices

- **Hash Table (`std::vector`)**: Chosen over dynamic array for automatic memory management.

1.1.2 Access Control

- **Private**: Internal state (`size` , `map`) and implementation details (hash functions) to prevent external modification.
- **Public**: Interface functions (constructor/destructor, core operations) required for usage from `main` .

1.1.3 Virtual Members

- Other than `newTable` , all interface functions (`store` , `search` , `deleteKey` , `corrupt` , `validate` , `print`) were made `virtual` . These functions are purely virtual; they must be implemented by the derived classes.

1.1.4 Polymorphic Design

Hash implementation uses inheritance with base `HashTable` and derived classes (`OpenAddressingHashTable` , `SeparateChainingHashTable`) instead of separate classes or conditional logic to make the code more reusable and to enforce consistent interface across implementations.

1.2 HashNode Class Design

The `HashNode` class is used to represent individual fileblocks in the hash table. This implementation uses a linked list for separate chaining. The `next` pointer is unused for open addressing.

1.2.1 Payload Datatype Choice

- `std::string` was chosen for automatic memory management and richer manipulation methods compared to an array or a vector of `char` .
- Using `std::string` simplifies the `corrupt()` function since we can just set the `string` variable to the new payload with a simple reassignment, without having to replace all characters with `\0` before corruption occurs or reading characters in a `for` loop.
- `std::string` was sufficient for this design since the upper size limit is 500 characters.

1.2.2 Access Control

- **Private**: Data fields (`id` , `checksum` , `data` , `next`) are all private to prevent external modification and maintain linked list structure. The `calc_checksum` function is an internal helper function for checksum computation and is private as well.
- **Public**: The destructor is required to delete the linked list since each node has a pointer. Setters and getters are available as needed for accessing and modifying member variables. `update_checksum` , `corrupt` , and `validate` are required to be usable by the hash table classes for interface functions.

1.2.3 Important Members

- `~HashNode()` : The destructor recursively deletes the `next` node and then itself.
- `corrupt()` : Takes advantage of the `std::string` datatype and simply reassigns the `data` member variable to the new payload.

1.3 OpenAddressingHashTable Class Design

The `OpenAddressingHashTable` class is derived from `HashTable` and uses double hashing and open addressing.

1.3.1 Collision Resolution

- Uses double hashing with two hash functions; `hash2` ensures offset is always odd to probe all possible locations.
- Probing sequence: `(hash1(k) + i * hash2(k)) % size` where `i` is the probe number
- For `search` ,

1.3.2 Key Operations

- **Store**: Probes until empty slot found or returns to original position (in which case all possible slots have been probed and table is full).
- **Search/Delete**: Uses `findKey` helper to locate element position.
- `findKey` centralizes probing logic for all operations that need to locate a key

1.4 SeparateChainingHashTable Class Design

The SeparateChainingHashTable class is derived from `HashTable` and uses chaining for collision resolution.

1.4.1 Chain Data Structure Choice

- A linked list implementation was selected over vector for better memory efficiency as linked list only allocates what is needed
- The nodes in each linked list are inserted in ascending order by `id` so that `print` can run in linear time.

1.4.2 Collision Resolution

- Uses a single hashing function. Upon collision, the new key is inserted into its appropriate spot in the linked list.

1.4.2 Key Operations

- **Store**: Traverses chain to maintain sorted order, allocates new node. Performs necessary pointer rearrangement to make room for new node.
- **Search/Delete**: Uses `findNode` helper to locate node in chain
- **Delete**: Special cases for head node, updates pointers for chain integrity
- `findNode` centralizes chain traversal logic for reuse

2. Runtime Analysis

Both implementations achieve $O(1)$ average runtime assuming at most m collisions where $m << T$ and m is $O(1)$:

2.1 Open Addressing

`store`, `search`, and `delete` all require the operation of searching, either for a target key or an empty slot. Both cases require at most m offsets to the hash value, where m is the maximum number of collisions. This is done in $O(m)$. Other function specific operations:

- `store`: creating a new node and inserting it into the vector are both $O(1)$.
- `search`: no additional work required.
- `delete`: creating a default `HashNode` and replacing the existing node with it are both $O(1)$.
  All of these operations are therefore done in $O(m) = O(1)$.

2.2 Separate Chaining

- `store`, `search`, and `delete` all require the operation of searching, either for a target key or an empty slot. Both cases require checking at most m nodes in the linked list, where m is the maximum number of collisions. This is done in $O(m)$. Other function specific operations:
- `store`: creating a new node, inserting it into the linked list and reassigning pointers are all $O(1)$.
- `search`: no additional work required.
- `delete`: reassigning pointers around the deleted node is $O(1)$.
  All of these operations are therefore done in $O(m) = O(1)$.

UML Diagram