1.0 Class Design

1.1 FileBlock

FileBlock is the class that holds the data in the program. The payload is implemented with a vector since it has built in memory management and resizing and is O(1) for most operations. The key member variables (ID, payload and checksum) are private to ensure data integrity.

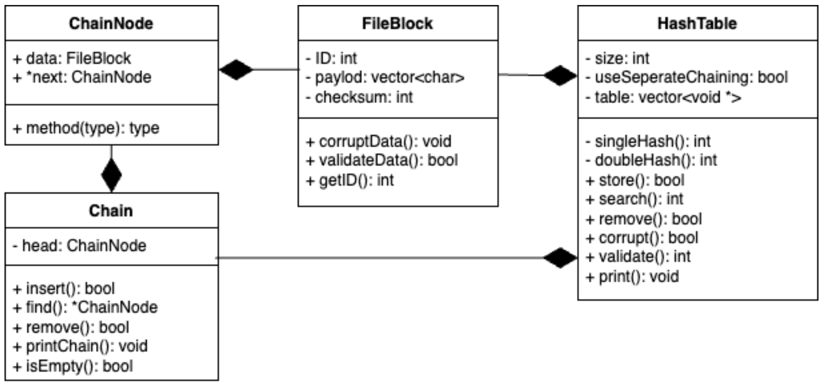
1.2 ChainNode and Chain

Chain is composed of ChainNodes. A ChainNode is a simple class which contains a FileBlock and a pointer to the next node. Chain only stores the pointer to the first node in the Chain since the ChainNode for a singly linked list. A singly linked list was chosen over a doubly linked list due to the reduced memory overhead and the nature of hash table operations, where traversal is minimal and generally unidirectional. Chain provides an API for traversing and managing the ChainNodes by updating the head and next pointers when needed.

1.3 HashTable

HashTable can be instantiated with separate chaining or open addressing collision resolution. The table itself is a vector. An array would have been a valid option as well since we don't need to resize the array but the vector class has an easy to use API for managing elements of the vector and has built in memory management and good time complexity. The HashTable manages the locations of the FileBlocks throughout the data structures and provide the primary API for this program will keeping critical member variables private. In the open addressing case, to handle deleted entries, the design uses a tombstone marker, which ensures that previously occupied slots are distinguished from truly empty slots.

2.0 UML Diagram



3.0Runtime

Assumption: for a given hash value, there are at most m collisions and m << T, the total number of elements inserted, and m is O(1). For open addressing, this assumption means that finding an valid index or element in the HashTable is a O(1) operation. For chaining, this means finding the index of the chain and then any chain traversing are also O(1).

STORE

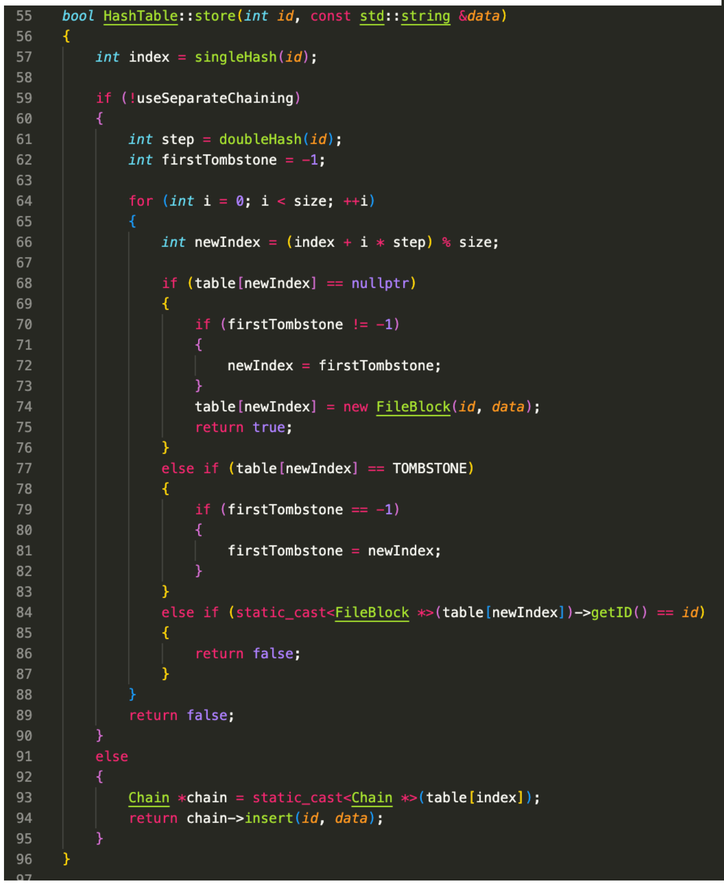
STORE calls HashTable.store() which start by trying to find a valid index for the FileBlock which is O(1) based on the above assumption. If a valid index is found a FileBlock is created at that index which is O(1). In the chaining case, Chain.insert() which traverses the chain before adding the FileBlock which from above is O(1).

3.2 SEARCH

SEARCH calls HastTable.search() which does essentially the same thing as HashTable.store() but looks for a specific FileBlock ID instead of a valid index or Chain to create a FileBlock. In the chaining case, Chain.find() is called. As above, these operations are O(1).

3.3 DELETE

DELETE calls HastTable.remove() which does essentially the same thing as HashTable.store() but looks for a specific FileBlock ID instead of a valid index then deletes the FileBlock if it finds it. In the chaining case, Chain.remove() is called. As above, these operations are O(1).

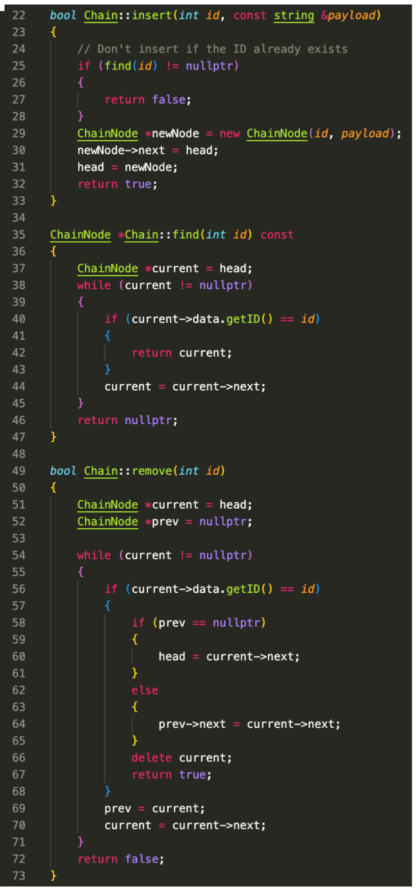


```cpp
bool HashTable::store(int id, const std::string &data)
{
    int index = singleHash(id);

    if (!useSeparateChaining)
    {
        int step = doubleHash(id);
        int firstTombstone = -1;

        for (int i = 0; i < size; ++i)
        {
            int newIndex = (index + i * step) % size;

            if (table[newIndex] == nullptr)
            {
                if (firstTombstone != -1)
                {
                    newIndex = firstTombstone;
                }
                table[newIndex] = new FileBlock(id, data);
                return true;
            }
            else if (table[newIndex] == TOMBSTONE)
            {
                if (firstTombstone == -1)
                {
                    firstTombstone = newIndex;
                }
            }
            else if (static_cast<FileBlock *>(table[newIndex])->getID() == id)
            {
                return false;
            }
        }
        return false;
    }
    else
    {
        Chain *chain = static_cast<Chain *>(table[index]);
        return chain->insert(id, data);
    }
}
```



```cpp
bool Chain::insert(int id, const string &payload)
{
    // Don't insert if the ID already exists
    if (find(id) != nullptr)
    {
        return false;
    }
    ChainNode *newNode = new ChainNode(id, payload);
    newNode->next = head;
    head = newNode;
    return true;
}

ChainNode *Chain::find(int id) const
{
    ChainNode *current = head;
    while (current != nullptr)
    {
        if (current->data.getID() == id)
        {
            return current;
        }
        current = current->next;
    }
    return nullptr;
}

bool Chain::remove(int id)
{
    ChainNode *current = head;
    ChainNode *prev = nullptr;

    while (current != nullptr)
    {
        if (current->data.getID() == id)
        {
            if (prev == nullptr)
            {
                head = current->next;
            }
            else
            {
                prev->next = current->next;
            }
            delete current;
            return true;
        }
        prev = current;
        current = current->next;
    }
    return false;
}
```

HashTable.search() and HastTable.remove() were omitted due to similarity.