1. Create a design **before** you begin to code that describes or shows how we can store data in a hash table and what kind of problem we could solve with a hash table.
   Simple Hash Table Design: (Note: This design was basically the same as what you used in the videos)
   A basic hash table comprises these components that work together to efficiently store and retrieve data:
   1. **Storage Array**:
      - It's a data structure used to store the values associated with the keys in the hash table.
      - Its size is determined by the capacity of the hash table.
   2. **Hash Function**:
      - Converts a string key into an integer index.
      - The index is used to place the key in the storage array.
   3. **Collision Handling**:
      - When two keys hash to the same index, it is called a collision.
      - This program uses simple overwrite, which is not ideal for practical use but demonstrates the concept.
   4. **Resize Function**:
      - For resizing the hash table when it becomes too full.

   SimpleHashTable can store and find unique IDs, like usernames or email addresses, in systems like databases or user management tools. However, its basic collision handling and resize function limitations may impact performance and memory usage, especially in large-scale applications.

   These are the functions that I will be using:

   Constructor: Initializes the hash table and allocates memory for the storage array.
   Hash Function: Converts a string key to an integer using ASCII values and takes the modulus with the capacity.
   Add: Inserts a key into the hash table, incrementing the collision count if necessary.
   Remove: Deletes a key from the hash table.
   Search: Checks if a key exists in the hash table.
   Resize: Placeholder function for increasing the hash table's capacity.
   ToString: Generates a string representation of the hash table.
   GetSize: Retrieves the current number of elements in the hash table.
   GetCollisionCount: Returns the count of collisions that have occurred.

   Smart Hash Table Design:
   A very similar design to the simple one but will use a vector of vectors to handle collisions. I'll also change the hash function to hash by division as opposed to addition.
   Sources:
   - https://www.geeksforgeeks.org/the-c-standard-template-library-stl/
   - https://www.geeksforgeeks.org/program-to-implement-separate-chaining-in-c-stl-without-the-use-of-pointers/?ref=ml_lbp
   - Wikipedia: Hash table

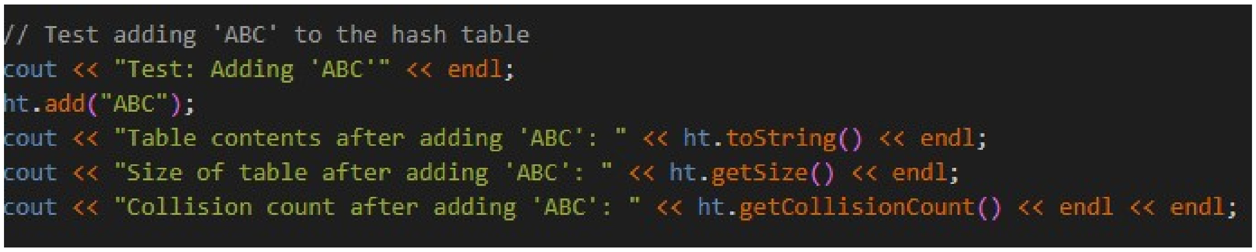
Components

1. **Storage Array**:
   - A vector of vectors (chaining to handle collisions).
   - Size determined by the initial capacity and resized when needed.
2. **Hash Function**:
   - Converts a string key into an integer index using the division method.
3. **Collision Handling**:
   - Uses chaining (each index in the storage array contains a vector of keys).
4. **Resize Function**:
   - Resizes the hash table when it gets full.

Functions used:
All the same as the simple design

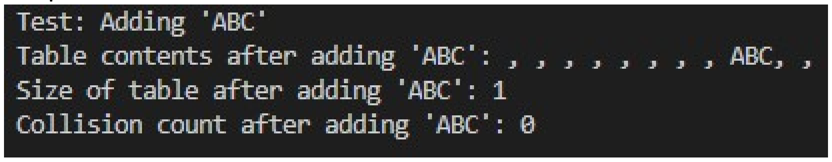
2. Create some tests (at least one per piece of functionality) **before** you begin coding that you want your hashtable to pass before you start coding.

Source: I used the same tests that you used in the video, except that also added a resize test.



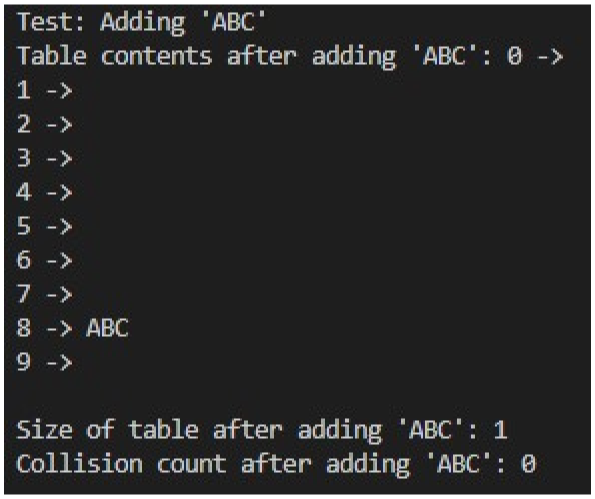
```
// Test adding 'ABC' to the hash table
cout << "Test: Adding 'ABC'" << endl;
ht.add("ABC");
cout << "Table contents after adding 'ABC': " << ht.toString() << endl;
cout << "Size of table after adding 'ABC': " << ht.getSize() << endl;
cout << "Collision count after adding 'ABC': " << ht.getCollisionCount() << endl << endl;
```

Simple:

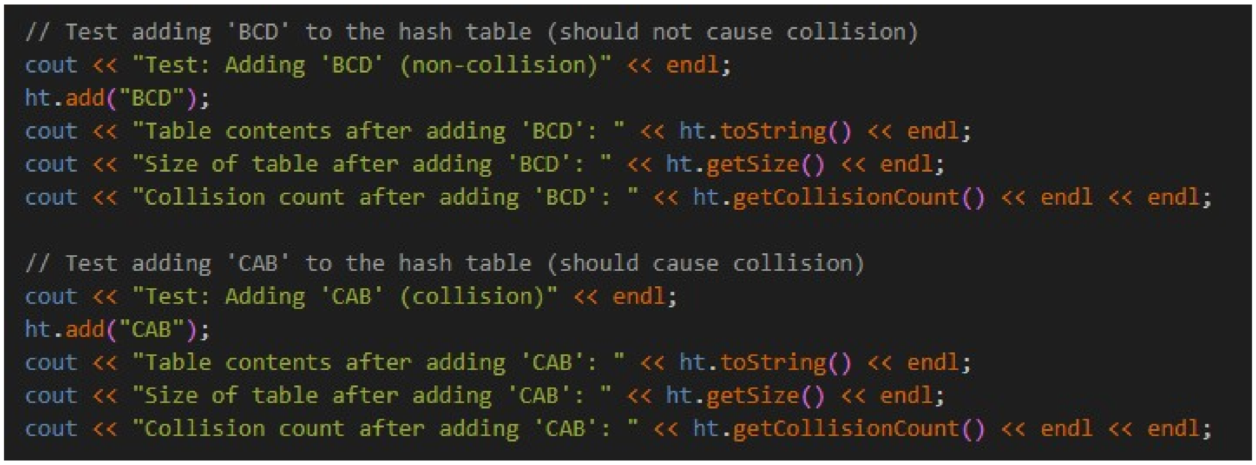


```
Test: Adding 'ABC'
Table contents after adding 'ABC': , , , , , , , , ABC, ,
Size of table after adding 'ABC': 1
Collision count after adding 'ABC': 0
```

Smart:

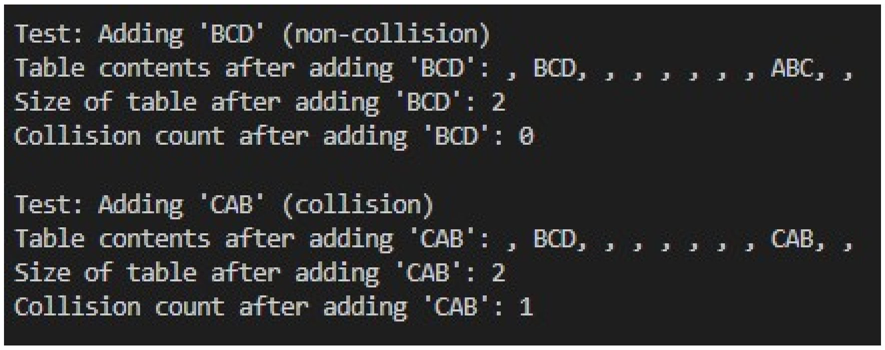


```
Test: Adding 'ABC'
Table contents after adding 'ABC': 0 ->
1 ->
2 ->
3 ->
4 ->
5 ->
6 ->
7 ->
8 -> ABC
9 ->

Size of table after adding 'ABC': 1
Collision count after adding 'ABC': 0
```



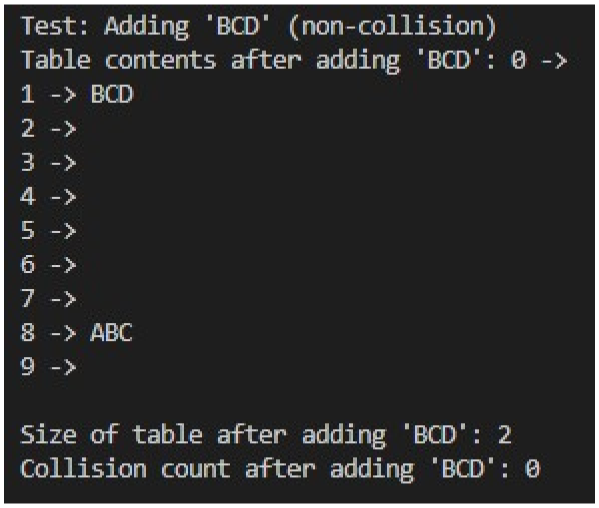
```
// Test adding 'BCD' to the hash table (should not cause collision)
cout << "Test: Adding 'BCD' (non-collision)" << endl;
ht.add("BCD");
cout << "Table contents after adding 'BCD': " << ht.toString() << endl;
cout << "Size of table after adding 'BCD': " << ht.getSize() << endl;
cout << "Collision count after adding 'BCD': " << ht.getCollisionCount() << endl << endl;

// Test adding 'CAB' to the hash table (should cause collision)
cout << "Test: Adding 'CAB' (collision)" << endl;
ht.add("CAB");
cout << "Table contents after adding 'CAB': " << ht.toString() << endl;
cout << "Size of table after adding 'CAB': " << ht.getSize() << endl;
cout << "Collision count after adding 'CAB': " << ht.getCollisionCount() << endl << endl;
```

Simple:

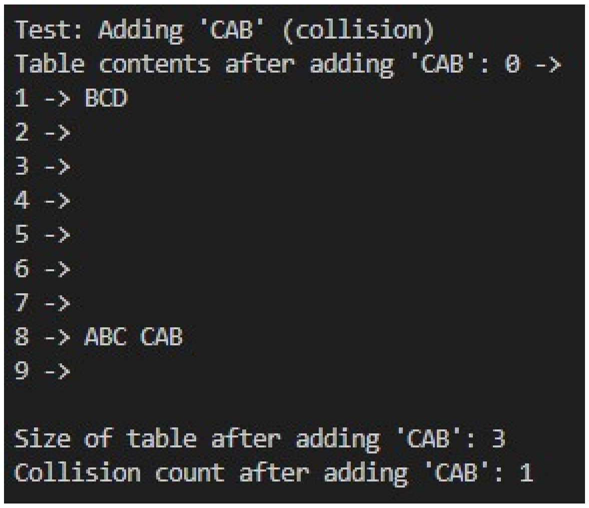


```
Test: Adding 'BCD' (non-collision)
Table contents after adding 'BCD': , BCD, , , , , , , ABC, ,
Size of table after adding 'BCD': 2
Collision count after adding 'BCD': 0

Test: Adding 'CAB' (collision)
Table contents after adding 'CAB': , BCD, , , , , , , CAB, ,
Size of table after adding 'CAB': 2
Collision count after adding 'CAB': 1
```

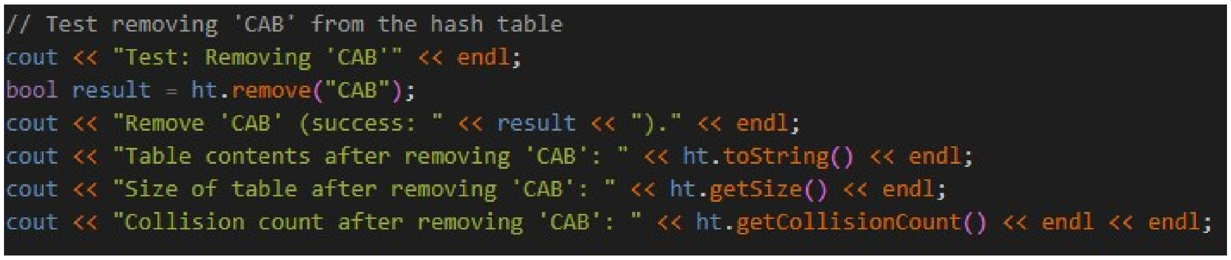
Smart:



```
Test: Adding 'BCD' (non-collision)
Table contents after adding 'BCD': 0 ->
1 -> BCD
2 ->
3 ->
4 ->
5 ->
6 ->
7 ->
8 -> ABC
9 ->

Size of table after adding 'BCD': 2
Collision count after adding 'BCD': 0
```

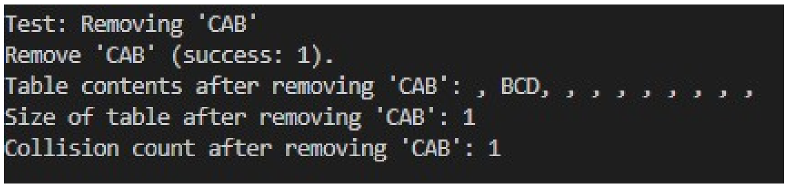


```
Test: Adding 'CAB' (collision)
Table contents after adding 'CAB': 0 ->
1 -> BCD
2 ->
3 ->
4 ->
5 ->
6 ->
7 ->
8 -> ABC CAB
9 ->

Size of table after adding 'CAB': 3
Collision count after adding 'CAB': 1
```



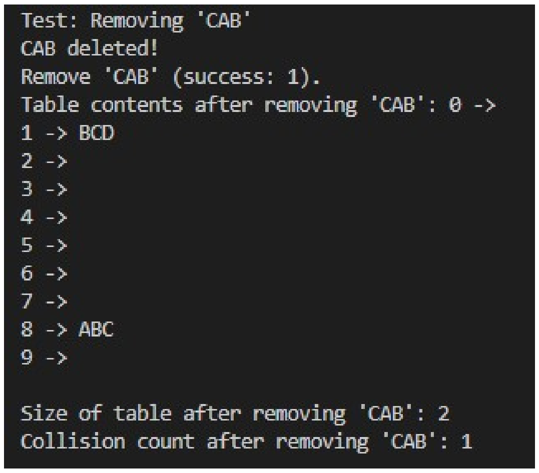
```cpp
// Test removing 'CAB' from the hash table
cout << "Test: Removing 'CAB'" << endl;
bool result = ht.remove("CAB");
cout << "Remove 'CAB' (success: " << result << ")." << endl;
cout << "Table contents after removing 'CAB': " << ht.toString() << endl;
cout << "Size of table after removing 'CAB': " << ht.getSize() << endl;
cout << "Collision count after removing 'CAB': " << ht.getCollisionCount() << endl << endl;
```

Simple:

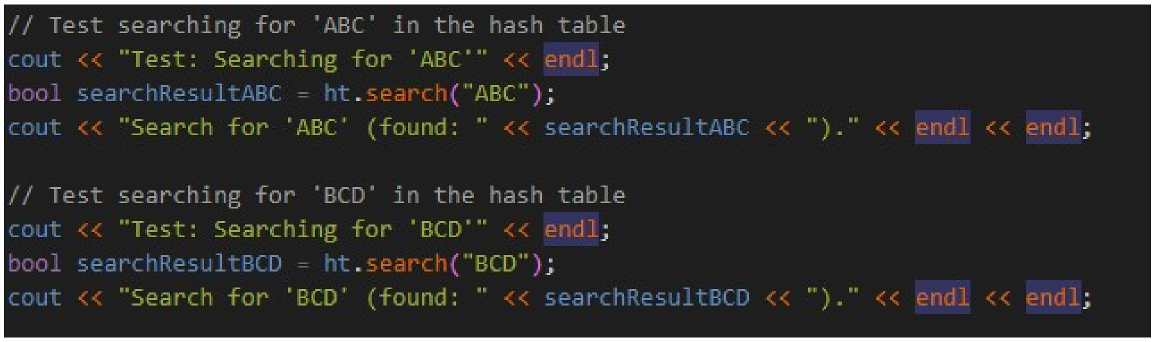


```
Test: Removing 'CAB'
Remove 'CAB' (success: 1).
Table contents after removing 'CAB': , BCD, , , , , , , , ,
Size of table after removing 'CAB': 1
Collision count after removing 'CAB': 1
```

Smart:

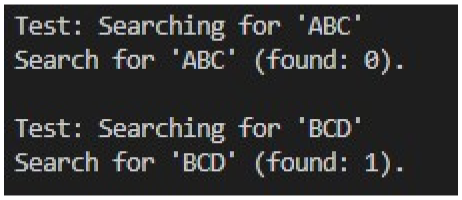


```
Test: Removing 'CAB'
CAB deleted!
Remove 'CAB' (success: 1).
Table contents after removing 'CAB': 0 ->
1 -> BCD
2 ->
3 ->
4 ->
5 ->
6 ->
7 ->
8 -> ABC
9 ->

Size of table after removing 'CAB': 2
Collision count after removing 'CAB': 1
```



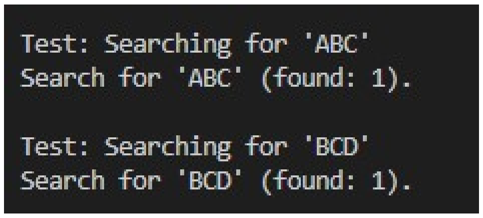
```cpp
// Test searching for 'ABC' in the hash table
cout << "Test: Searching for 'ABC'" << endl;
bool searchResultABC = ht.search("ABC");
cout << "Search for 'ABC' (found: " << searchResultABC << ")." << endl << endl;

// Test searching for 'BCD' in the hash table
cout << "Test: Searching for 'BCD'" << endl;
bool searchResultBCD = ht.search("BCD");
cout << "Search for 'BCD' (found: " << searchResultBCD << ")." << endl << endl;
```

Simple:

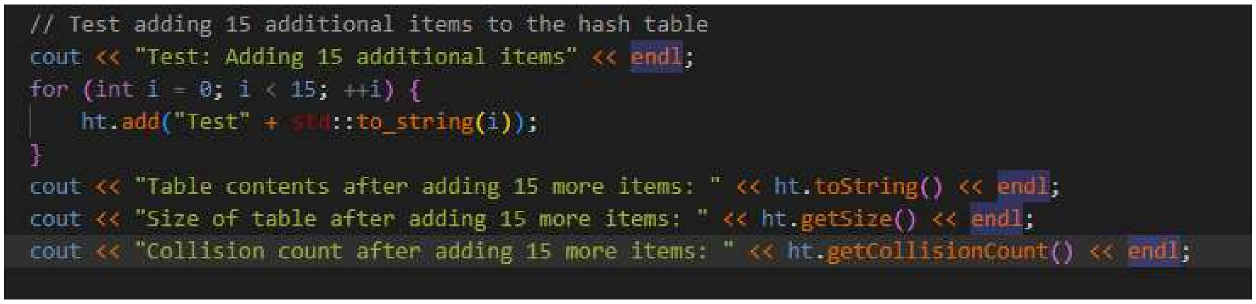


```
Test: Searching for 'ABC'
Search for 'ABC' (found: 0).

Test: Searching for 'BCD'
Search for 'BCD' (found: 1).
```

Smart:

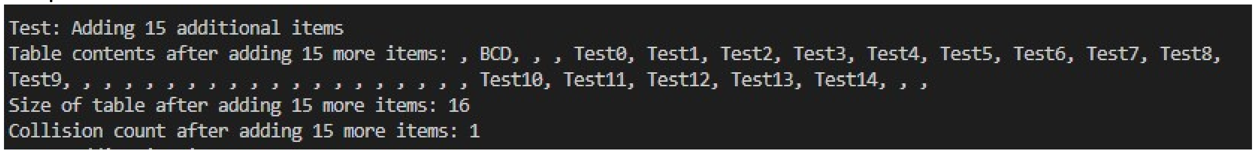


```
Test: Searching for 'ABC'
Search for 'ABC' (found: 1).

Test: Searching for 'BCD'
Search for 'BCD' (found: 1).
```



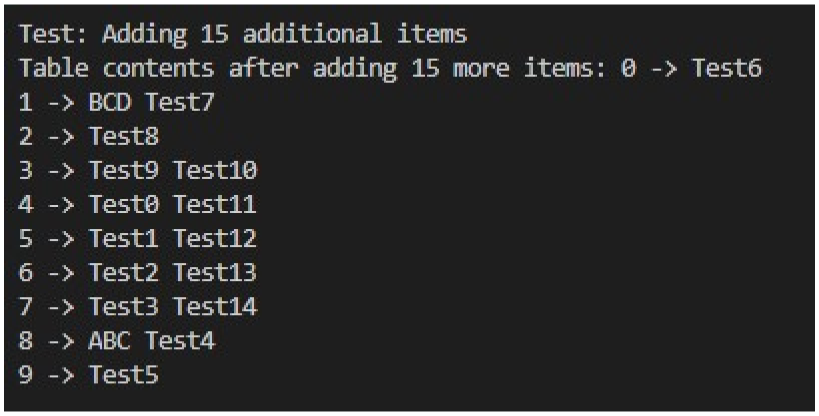
```
// Test adding 15 additional items to the hash table
cout << "Test: Adding 15 additional items" << endl;
for (int i = 0; i < 15; ++i) {
    ht.add("Test" + std::to_string(i));
}
cout << "Table contents after adding 15 more items: " << ht.toString() << endl;
cout << "Size of table after adding 15 more items: " << ht.getSize() << endl;
cout << "Collision count after adding 15 more items: " << ht.getCollisionCount() << endl;
```

Simple:



```
Test: Adding 15 additional items
Table contents after adding 15 more items: , BCD, , , Test0, Test1, Test2, Test3, Test4, Test5, Test6, Test7, Test8,
Test9, , , , , , , , , , , , , , , , , , , , Test10, Test11, Test12, Test13, Test14, , ,
Size of table after adding 15 more items: 16
Collision count after adding 15 more items: 1
```

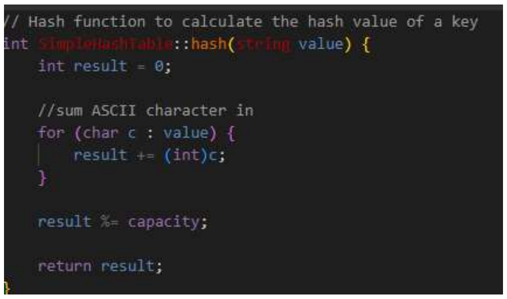
Smart:



```
Test: Adding 15 additional items
Table contents after adding 15 more items: 0 -> Test6
1 -> BCD Test7
2 -> Test8
3 -> Test9 Test10
4 -> Test0 Test11
5 -> Test1 Test12
6 -> Test2 Test13
7 -> Test3 Test14
8 -> ABC Test4
9 -> Test5
```

3. **Create a hashtable that resolves collisions by simply overwriting the old value with the new value, including at least:**

    1. Describe the way that you decide on hashing a value
       (this can be simple or complex based on how interesting you find the topic)

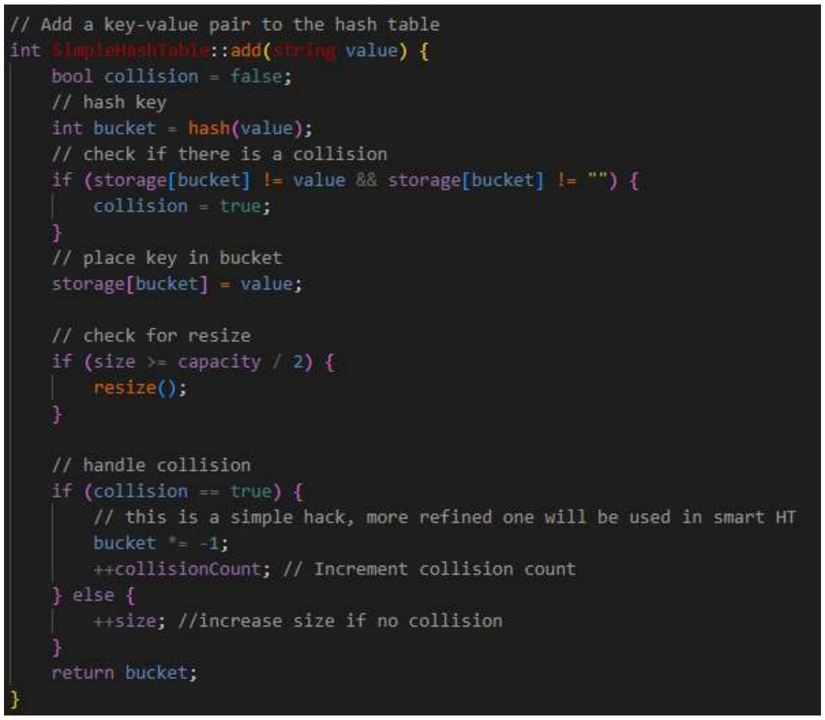


```
// Hash function to calculate the hash value of a key
int SimpleHashTable::hash(string value) {
    int result = 0;

    //sum ASCII character in
    for (char c : value) {
        result += (int)c;
    }

    result %= capacity;

    return result;
}
```

I chose to use the same one shown in the in-class video. It sums the ASCII characters, so ABC would have the same value as BAC and CAB, but BCD would have different values.

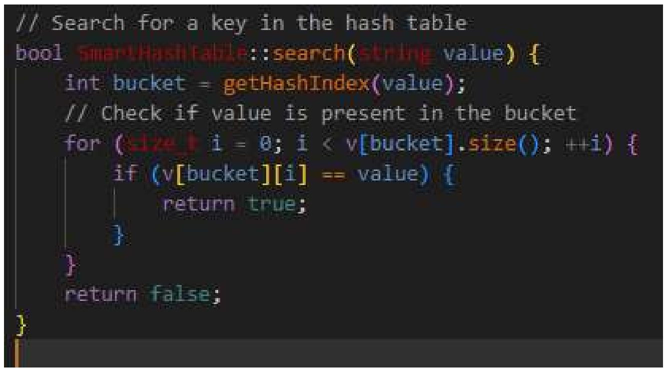
2. An insert function that places the value at the appropriate location based on its hash value

   From in-class video example. I did add a check for resize portion



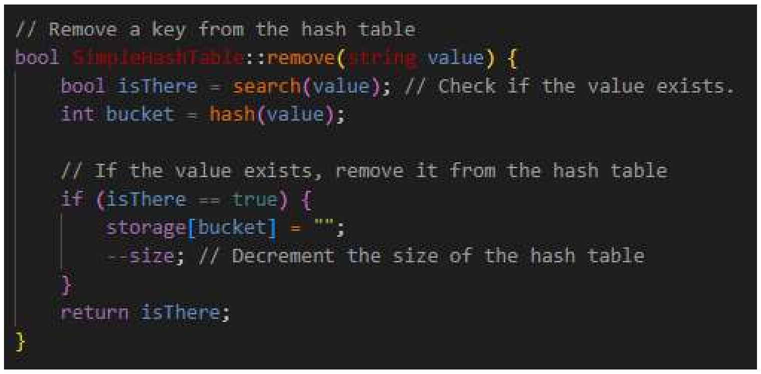
```cpp
// Add a key-value pair to the hash table
int SimpleHashTable::add(string value) {
    bool collision = false;
    // hash key
    int bucket = hash(value);
    // check if there is a collision
    if (storage[bucket] != value && storage[bucket] != "") {
        collision = true;
    }
    // place key in bucket
    storage[bucket] = value;

    // check for resize
    if (size >= capacity / 2) {
        resize();
    }

    // handle collision
    if (collision == true) {
        // this is a simple hack, more refined one will be used in smart HT
        bucket *= -1;
        ++collisionCount; // Increment collision count
    } else {
        ++size; //increase size if no collision
    }
    return bucket;
}
```

3. A contains function that returns whether the value is already in the hashtable



```cpp
// Search for a key in the hash table
bool SmartHashTable::search(string value) {
    int bucket = getHashIndex(value);
    // Check if value is present in the bucket
    for (size_t i = 0; i < v[bucket].size(); ++i) {
        if (v[bucket][i] == value) {
            return true;
        }
    }
    return false;
}
```

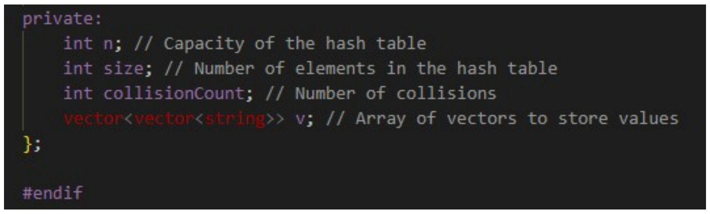
4. (optional) A delete function that removes a value based on its hash and then returns that value…



```cpp
// Remove a key from the hash table
bool SimpleHashTable::remove(string value) {
    bool isThere = search(value); // Check if the value exists.
    int bucket = hash(value);

    // If the value exists, remove it from the hash table
    if (isThere == true) {
        storage[bucket] = "";
        --size; // Decrement the size of the hash table
    }
    return isThere;
}
```

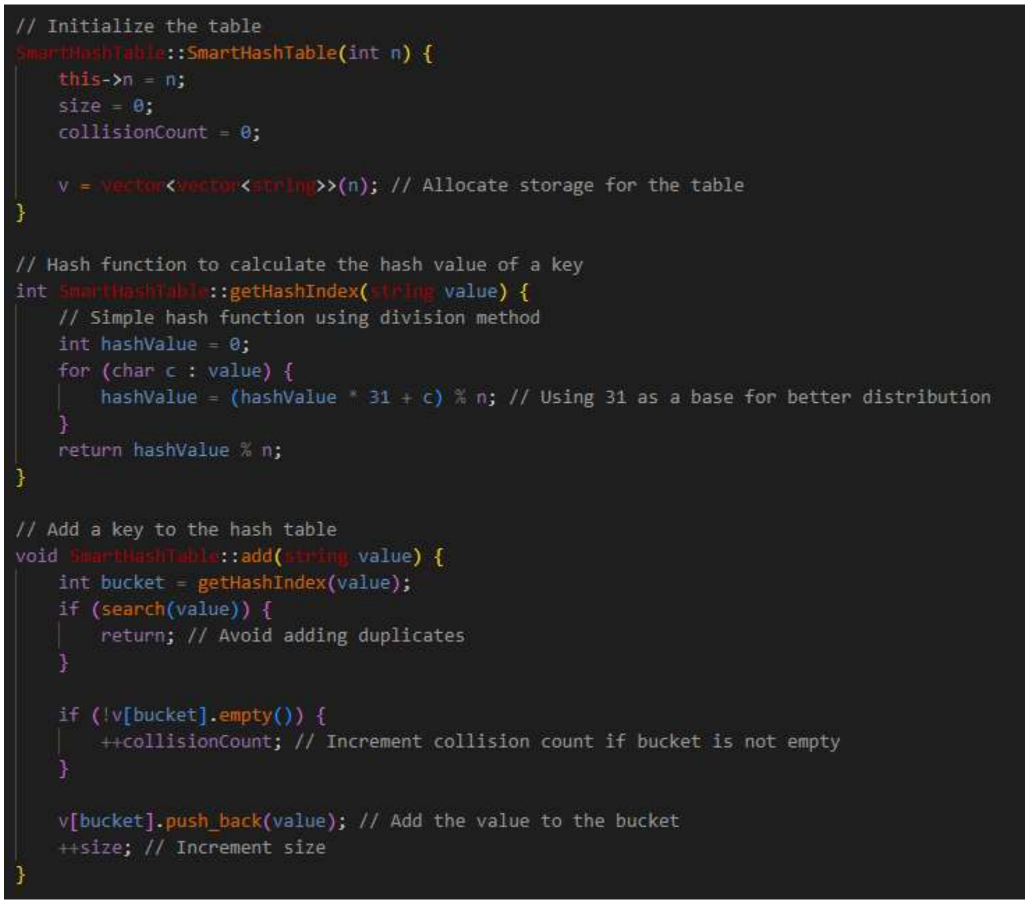
4. Then create a smarter hashtable (double hashing or chaining) including at least the same functions as the simple hashtable

   I'm using separate chaining without the use of pointers. I do this using a vector of vectors.



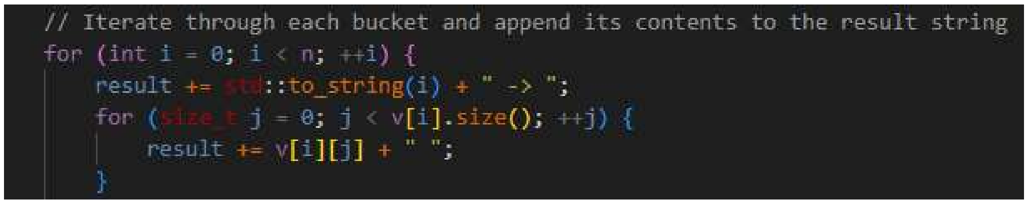
```
private:
    int n; // Capacity of the hash table
    int size; // Number of elements in the hash table
    int collisionCount; // Number of collisions
    vector<vector<string>> v; // Array of vectors to store values
};

#endif
```

- I'm using this https://www.geeksforgeeks.org/program-to-implement-separate-chaining-in-c-stl-without-the-use-of-pointers/?ref=ml_lbp as a source so that I can implement this change. I've also changed my hashing function to be one of division to reduce the chances of "bumps" or busy points. I've simplified my add function to work well with the vector of vectors.



```
// Initialize the table
SmartHashTable::SmartHashTable(int n) {
    this->n = n;
    size = 0;
    collisionCount = 0;

    v = vector<vector<string>>(n); // Allocate storage for the table
}

// Hash function to calculate the hash value of a key
int SmartHashTable::getHashIndex(string value) {
    // Simple hash function using division method
    int hashValue = 0;
    for (char c : value) {
        hashValue = (hashValue * 31 + c) % n; // Using 31 as a base for better distribution
    }
    return hashValue % n;
}

// Add a key to the hash table
void SmartHashTable::add(string value) {
    int bucket = getHashIndex(value);
    if (search(value)) {
        return; // Avoid adding duplicates
    }

    if (!v[bucket].empty()) {
        ++collisionCount; // Increment collision count if bucket is not empty
    }

    v[bucket].push_back(value); // Add the value to the bucket
    ++size; // Increment size
}
```

Most of my functions are now using some sort for loop to go through each bucket:



```
    // Iterate through each bucket and append its contents to the result string
    for (int i = 0; i < n; ++i) {
        result += std::to_string(i) + " -> ";
        for (size_t j = 0; j < v[i].size(); ++j) {
            result += v[i][j] + " ";
        }
```

5. Compare some information relating to collisions (frequency) and their effect on complexity (of insert and contains methods)

   Simple: Collisions resolved by writing over existing values. This can lead to frequent collisions if the hash function distributes the keys evenly.

   O(1) insertion and search (contains) operations on average
   O(n) in the worst-case scenario where all keys hash to the same index.

   Double: Applies a secondary hash function to calculate alternative indices for colliding keys, reducing the frequency of collisions

   Chaining: Addresses collisions by creating linked lists or other data structures (vertices) at each index to store multiple values associated with the same hash.

   O(1) for average insertion and search complexity
   O(n) in worst case scenario where all the keys hash to the same index.

6. Once you have implemented and tested your code, add to the README file what line(s) of code or inputs and outputs show your work meeting each of the above requirements (or better, include a small screen snip of where it meets the requirement!).

   DONE!