The Chained Hashing Assignment:

Implement the Table template class from Chapter 12, using a Chained hash table to store the items. This will be using link lists. You will need to write the lined list code yourself (No Linked List Tool kit software allowed.

Purposes:

Ensure that you understand and can implement the important data structure of a hash table.

Before Starting:

Read Sections 12.2 and 12.3.

Files that you must write:

1. **table2.h**
2. **Download table2.h**
3. **: Header file for this version of the Table class. You don't have to write much of this file. Just copy our version from table2.h and add your name and other information at the top. NOTE: This is a template class. The template parameter, called RecordType, may be instantiated as any structure / class that contains an integer member variable called key.**
4. **table2.tpp.h**
5. **Download table2.tpp.h**
6. **: The implementation file for the new Table class. There is some code here to help you get starts. You must write most of the functions here yourself.**

Other files that you may find helpful / necessary:

1. **testtab2.cxx**
2. **Download testtab2.cxx**
3. **: A simple interactive test program.**
4. **tabexam.cxx**
5. **Download tabexam.cxx**
6. **: A non-interactive test program that will be used to grade the correctness of your Table class.**

The Table Class Using a Chained Hash Table

Discussion of the Assignment

This version of the Table is a template class with a template parameter called RecordType, as described in Sections 12.2 and 12.3. In an actual program (such as tabtest.cxx), the RecordType may be instantiated as any structure / class type with an integer member variable called key.

Start by understanding the private member variables that have been proposed in table2.h. For example, where are the head pointers stored for the separate linked lists? Write an invariant that describes the use of these private member variables, and put the invariant at the top of your implementation file.

I suggest that you also write some private member functions that can help in the implementations of the public member functions. For example, one of the private member functions that I found useful was:

```
Node<RecordType>* find_node(int key);

// Precondition: none.

// Postcondition: If there is a node in the Table with the
specified

// key, then the return value is a pointer to this node.
Otherwise the

// return value is nullptr.
```

I made use of find_node( )in my implementations of find and is_present( ). If you decide to use something like find_node( ), then list the prototype in the private section of the Table class. Put the implementation in the implementation file along with a precondition/postcondition contract.

Hints for the remove function:

1. **Hash the key. If there is no linked list for than value, you are done!**
2. **Traverse the linked list looking for the key. Track the precursor and cursor as you go.**
3. **If the item is at the Head of the linked list, remove the head (special head function)**
4. **If it is in the middle, remove that item**
5. **If it is not in the list, do nothing.**

NOTE: Since your Table class is using dynamic memory (a bunch of linked lists), the Table must have a copy constructor, assignment operator, and a destructor! But keep in mind that much of your work can be carried out by the functions of the linked list toolkit.

Use the interactive test program and the debugger to track down errors in your implementation. If you have an error, *do not start making changes until you have identified the cause of the error.* If you come to me for help, I will always ask you to do the following:

1. **Show us the invariant that describes how your private member variables implement the Table class.**
2. **Use the debugger to show us the problem!**