Description
Problem 5.5 asks to implement separate chaining hash tables using singly linked lists.

There is an inner node class to represent the nodes in a singly linked list and several common hash table methods in the hash table class

The insert method adds the specified key into the hash table. It calculates the hash code of the key and then finds the corresponding list in the hash table. If the list is empty, it creates a new node with the key and sets it as the head of the list. Otherwise, it traverses the list until it reaches the end and then adds the node to the end. Finally, it increments the size of the hash table by 1.

The contains method returns true if the hash table contains the specified key, and false otherwise. It calculates the hash code of the key and then traverses the corresponding list, checking each node to see if its key matches the specified key. If a matching node is found, it returns true. Otherwise, it returns false after checking all the nodes in the list.

The remove method removes the specified key from the hash table, if it exists. It calculates the hash code of the key and then traverses the corresponding list, searching for a node with a key that matches the specified key. If such a node is found, it is removed from the list by linking the previous node to the next node. If the head node is the one that matches the key, it is removed by setting the second node as the new head of the list. Finally, the size of the hash table is decremented by 1.

The size method returns the number of elements currently stored in the hash table.

The hash method computes the hash code of the specified key. It takes the remainder of the masked key when divided by the length of the hash table.

The print method prints out the elements in the hash table, along with their indices. It traverses each list in the hash table and prints out the key of each node in the list, along with the index of the list. If a list is empty, it does not print anything after specifying the index.

The main class tests all these methods.

Complexity Analysis
The time complexity of the insert method is O(1) on average, as it simply needs to calculate the hash code, access the appropriate list, and add the key to the end. However, in the case that all keys map to the same hash table index, the time complexity of put is O(N), as it needs to traverse the entire list to find the end.

The time complexity of the contains method is also O(1) on average, as it needs to calculate the hash code, access the appropriate list, and traverse the list until it finds a matching key or reaches the end of the list. However, it can be O(N) when all N keys in the hash table map to the same hash table index, resulting in a long linked list. In this case, the method would need to traverse the entire linked list.

The time complexity of the remove method is very similar to the contains method. It is O(1) on average as it needs to calculate the hash code, access the appropriate list, and traverse the list until it finds the node to remove or reaches the end of the list. However, when all keys in the hash table map to the same hash table index, the method would need to traverse the entire linked list, which would take O(N) time.

The O(N) case for these three methods can be avoided with a good hash function and a good load factor, which I implement in my test cases. However, if a small size is chosen, it may result in a worse time complexity.

The size method simply returns the size of the hash table, which is stored as an instance variable, so its time complexity is O(1).

The time complexity of the hash method is O(1), as it simply performs some basic arithmetic operations and a single modulo operation.

The time complexity of print is O(N),. This is because it needs to traverse each list in the hash table and print out each key in the list.