PROJECT 3:

I chose separate chaining as my collision resolution strategy, due to the fact that I already have a templated LinkedList class. My bucket size is the size of the list integer divided by 5. This means that on average each bucket will have 5 items in its corresponding LinkedList. The variable number of buckets means that the load factor will always be 5, or truncated to 5. My hash function is simply the total items % bucket size.

PROJECT 2:

Linked lists were surprisingly easy in my opinion. The merge and purge wasn't bad either. The toughest part for me was figuring out how to delete the existing array and create a new one after a merge/purge,

PROJECT 1:

During the error checking of each eclipse, I create an array that stores each column, so after creation, and before addition to the array, I can check for duplicate catalog numbers. To do this, I have to do a linear search through the existing array, which is time consuming, and needs to be improved.

I modified Binary search to find the first instance of a possibly duplicated key, then used a small linear search to cout all eclipses with the same key, until they are no longer exactly the same.

LABS 1-4:

In this program, I decided to keep each eclipse as a string for most of the code. It was a simpler way to deal with the commas and to make each line a csv. To throw an error when specific columns are not int or double was a challenge. I had to use two temporary variables, tempIndex and tempLength to make substrings that corresponded to the data between each column. I then used a try/catch method with the function “stoi” and used cerr when there was a caught error.

I made a class called “Eclipse”, and did all of the calculations for errors, and partial eclipse within it, along with the print method, and the overloaded ostream display method.

The RemoveableArray class was a lot simpler than I thought it would be. It was easy to write generic code within the class.