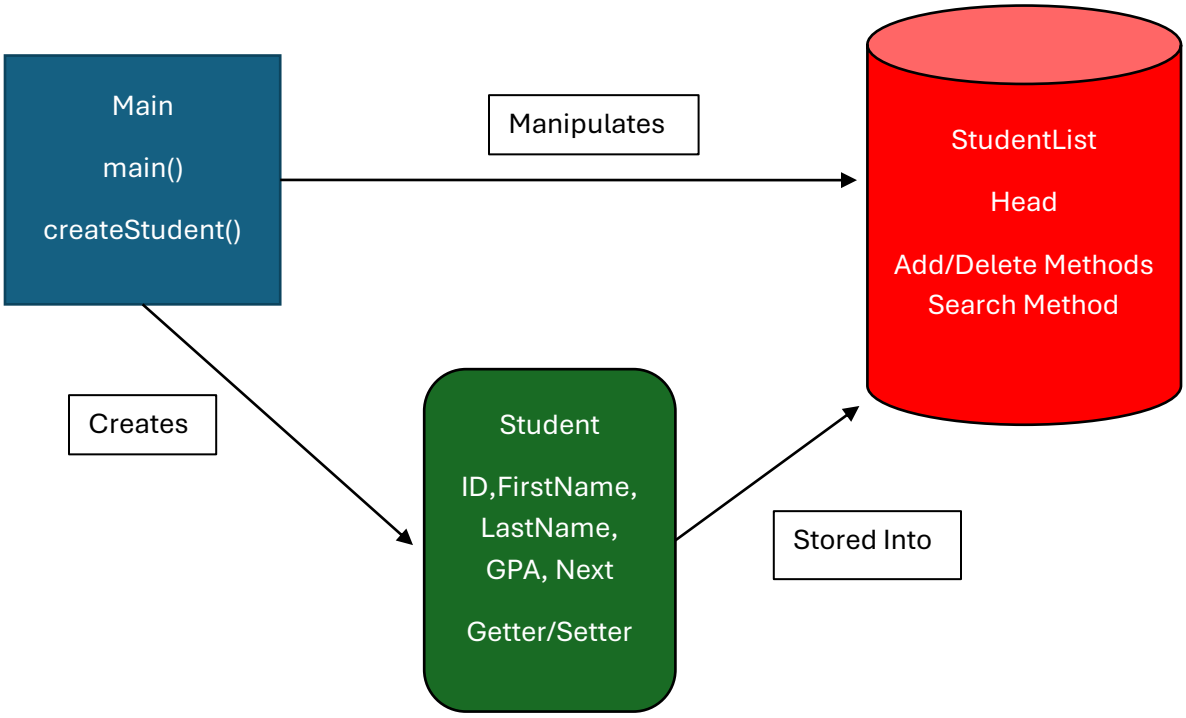
Unsorted List

1. Plan

For the bulk of the assignment, I will need two classes. Student, and StudentList. Student is the node class to contain information about the individual students entered into the database. StudentList will be the list object which will hold the nodes.

The diagram below illustrates the creation process of a new Student object



Implementation

First is to create a student class. The student is only responsible for holding the values needed for its exact node. This includes ID, FirstName, LastName, GPA, Status, and the pointer to the next node. The screenshot below shows the values held by each node.

```
private:
    int id;
    std::string firstName;
    std::string lastName;
    double gpa;
    status studentStatus;

    Student* next; // Pointer to the next student in the list
```

The printed output of Student toString method

```
ID: 1
Name: Test One
GPA: 3.000000
Year: Active
```

Only having these minimal values helps make the program simpler to develop and allows the Student class to work on other implementations without needing to be part of a list. Next is the Student List object. This only needs to hold one value, being the first element of the list.

```
private:
    Student* head;
```

Iterating through the list is rather simple. Assuming the list has content, we first access the head node, which stores a reference to the next element in the list. We then access the next element in the list and so on until either a certain node is found or the end of the list is reached. Shown is a search method which does exactly that.

```cpp
// Search method - returns index of found Student, -1 if not found
int StudentList::searchByID(int idToSearch)
{
    Student* temp = head;
    int index = 0;
    while (temp != nullptr)
    {
        if (temp->getId() == idToSearch)
        {
            return index;
        }
        temp = temp->getNext();
        index++;
    }
    // Will only get here if not found
    return -1;
}
```

The list has three requirements:

1) Add a new Student object
2) Delete a Student from the list
3) Display all students

One big requirement is to not tightly couple the objects with the main function. For the purposes of this assignment, only the main function itself should be printing to terminal.

```
// Add methods
void addFront(Student newStudent);
void append(Student newStudent);

// Search method - returns index of found Student, -1 if not found
int searchByID(int idToSearch);

// Delete Methods
void deleteByID(int idToDelete);
void deleteByIndex(int indexToDelete);

// toString
std::string toString();

// Destructor
~StudentList();
```

Next is to finalize the main function itself. First is to create a main menu where the user can choose their action. (I decided to do a little bit more than what is required for the assignment). To simplify the main function itself, I made the main menu a separate function which main calls to get the users intended function.

```
int mainMenu()
{
    cout << "Student List Menu" << endl;
    cout << "1. Add new student to front" << endl;
    cout << "2. Append new student to end" << endl;
    cout << "3. Search for student by ID" << endl;
    cout << "4. Delete student by ID" << endl;
    cout << "5. Delete student by index" << endl;
    cout << "6. Print student list" << endl;
    cout << "7. Exit" << endl;
    cout << "Enter your choice: ";
    int choice;
    do
    {
        cin >> choice;
        if (cin.fail() || choice < 1 || choice > 7)
        {
            cin.clear(); // clear the fail state
            cin.ignore(numeric_limits<streamsize>::max(), '\n'); // discard invalid input
            choice = -1; // set choice to an invalid number
            cout << "Invalid input. Please enter a number between 1 and 7: ";
        }
    } while (choice < 1 || choice > 7);
    return choice;
}
```

I then take the users function choices and run the code as needed. Shown is part of the switch statement which carries out the users preferred function.

```
choice = mainMenu();
switch (choice)
{
case 1:
    // Add new student to front
    list.addFront(createStudent());
    break;
case 2:
    // Append new student to end
    list.append(createStudent());
    break;
```

Next is the running of the code.

```
Student List Menu
1. Add new student to front
2. Append new student to end
3. Search for student by ID
4. Delete student by ID
5. Delete student by index
6. Print student list
7. Exit
Enter your choice: 1
Enter student ID: 1
Enter first name: One
Enter last name: One
Enter GPA: 1.0
Enter status (1: Active, 2: Graduated, 3: Suspended, 4: Unknown): 1
Student List Menu
1. Add new student to front
2. Append new student to end
3. Search for student by ID
4. Delete student by ID
5. Delete student by index
6. Print student list
7. Exit
Enter your choice: 2
Enter student ID: 2
Enter first name: Two
Enter last name: Two
Enter GPA: 2.0
Enter status (1: Active, 2: Graduated, 3: Suspended, 4: Unknown): 2
Student List Menu
1. Add new student to front
2. Append new student to end
3. Search for student by ID
4. Delete student by ID
5. Delete student by index
6. Print student list
7. Exit
Enter your choice: 6
ID: 1
Name: One One
GPA: 1.000000
Year: Active

ID: 2
Name: Two Two
GPA: 2.000000
Year: Graduated
```

Challenges

I honestly didn't have too many challenges with this assignment. However, I have learned how important modularity is for coding. Upon review, I realize that how easy it would be to modify the existing classes to fit other purposes. Making a double linked list for example would only need a reference to a previous node in the Student class, and since the existing single linked list doesn't use a previous node, enabling the Student class for it (theoretically at least) would not interfere with the functionality of the existing single linked

list. So I hope to explore code modularity more and enable my code to be portable between projects. Especially if it's going to help with future school assignments.