CSCI205 Computer Science III
Lab Assignment
Abstract Lists and LinkedLists

AI Statement: Any use of AI must be cited (trust me, it is obvious). All analysis should be written in your own words (again, it is obvious). Instructions must be followed as written.

Why Linked Lists Are Great To Study
Linked lists hold a special place in the hearts of many programmers. Linked lists are great to study because...

- *Nice Domain* The linked list structure itself is simple. Many linked list operations such as "reverse a list" or "delete a list" are easy to describe and understand since they build on the simple purpose and structure of the linked list itself.
- *Complex Algorithms* Even though linked lists are simple, the algorithms that operate on them can be as complex and beautiful as you want. It's easy to find linked list algorithms that are complex, and pointer intensive.
- *Pointer Intensive* Linked list problems in C/C++ are really about pointers. The linked list structure itself is obviously pointer intensive. Furthermore, linked list algorithms often break and re-weave the pointers in a linked list as they go. Linked lists really test your understanding of pointers.
- *Visualization* Visualization is an important skill in programming and design. Ideally, a programmer can visualize the state of memory to help think through the solution. Even the most abstract languages such as Java and Python have layered, reference based data structures that require visualization. Linked lists have a natural visual structure for practicing this sort of thinking. It's easy to draw the state of a linked list and use that drawing to think through the code.

Not to appeal to your mercenary side, but for all of the above reasons, linked list problems are often used as interview and exam questions. They are short to state, and have complex, pointer intensive solutions. No one really cares if you can build linked lists, but they do want to see if you have programming agility for complex algorithms and pointer manipulation. Linked lists are the perfect source of such problems.

Linked List Ground Rules
The linked list you design should model the "classic" singly linked list structure: Inside the list class, a single head pointer points to the first node in the list. If you like you can also add a ***tail pointer***. This can make operations at the end of the list not require iteration.

Each node contains a single ***.next*** pointer to the next node. The ***.next*** pointer of the last node is NULL. The empty list is represented by a NULL head pointer. All of the nodes are allocated on the heap using ***new***

```
// The structure describing the node
template <class T>
struct Node{
  T item;         // data object
  Node<T>* next; // pointer to "next" node
};
```

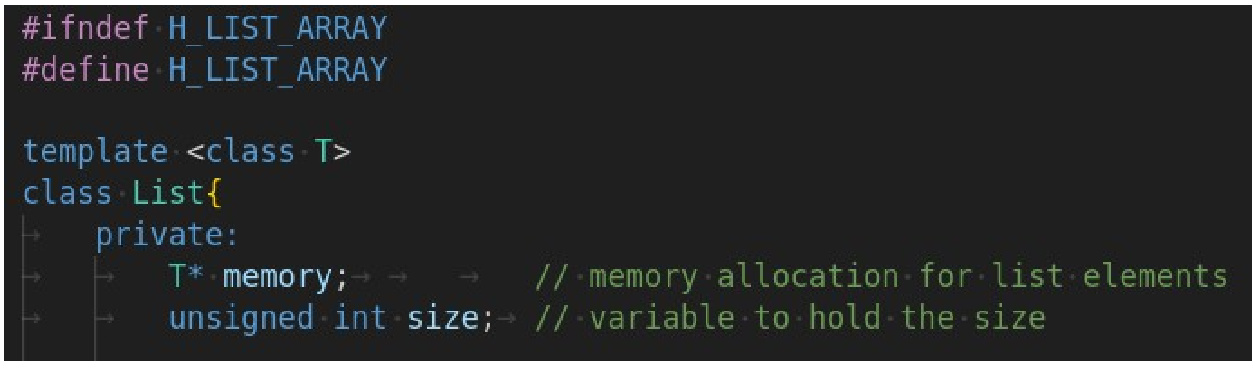
Feel free to use the provided lecture code as a base for the following exercises. You may be asked to rename certain behaviors to align with expected list behavior abstraction. Please follow the instructions carefully. **You will be building two list implementations as you move through the exercises.** These implementations are to provide a framework for comparison and contrast. These implementations are to have identical abstract interfaces (method names). So, for the programmer ***using*** these objects there would be no external differences.

Abstract List Behaviors: Each of your list implementations (described below) should have the following behaviors. You will be comparing and contrasting the 2 implementations. In both implementations include ***detailed analysis*** of the efficiency and memory requirements. Also include Big O notation and some details on how you arrived at your conclusions. Please be thorough here.

• void insert(item, position)	// insert item at position
• T get(position)	// get item at position
• int find(item)	// find item and return its position
• T remove(item)	// remove and return item specified by the parameter
• void print()	// print list in some attractive format
• int length()	// returns the number of elements in the list

List Implementations: What follows are descriptions of the two list implementations you will be building. You may want to use sub-directories for each one to avoid confusion.

Array: Using the provided **List.hpp** class implement the List ADT with a ***compact, templated, dynamically sized array*** as a private member. The array will be the memory organization for the list elements. All expected list behaviors (listed above) will be performed on the array. Be sure to maintain compactness across all behaviors.

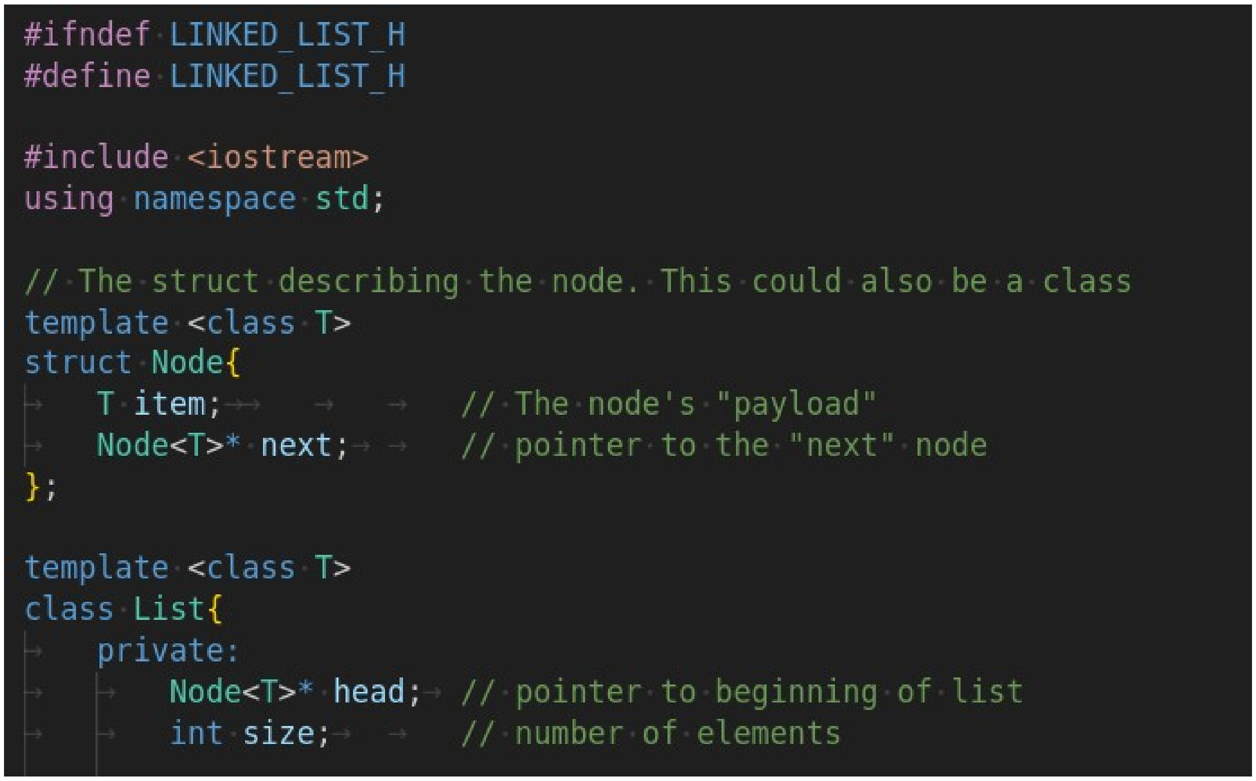


```
#ifndef H_LIST_ARRAY
#define H_LIST_ARRAY

template <class T>
class List{
    private:
        T* memory;              // memory allocation for list elements
        unsigned int size;      // variable to hold the size
```

Sizing: When the underlying array becomes full the following should occur

- The array should be resized to double current capacity
- All existing elements should be copied over to the new array with all positions and order maintained.
- This should happen via a private helper method. Include ***detailed*** comments about the efficiency of the algorithm. Include an analysis on any additional memory that is necessary.

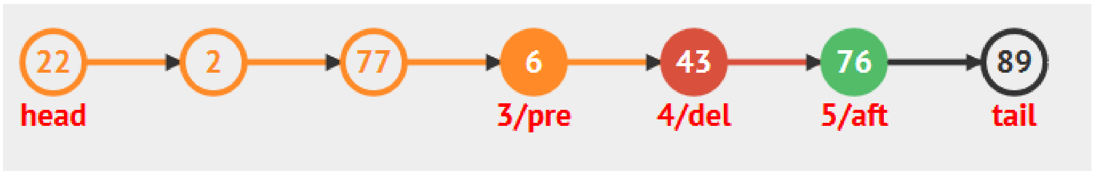
Linked List: Using the provided **List.hpp** class implement the List ADT with a Linked List as the underlying memory organization



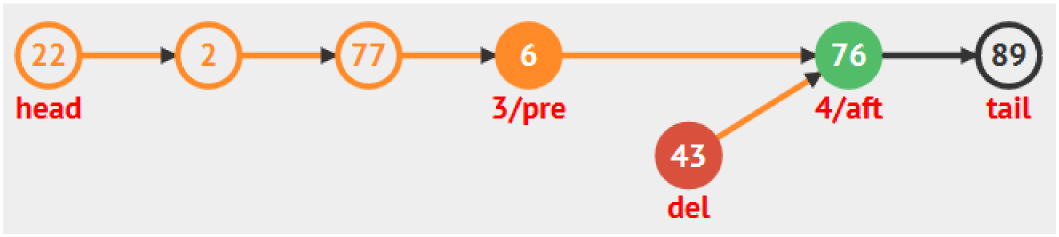
```cpp
#ifndef LINKED_LIST_H
#define LINKED_LIST_H

#include <iostream>
using namespace std;

// The struct describing the node. This could also be a class
template <class T>
struct Node{
    T item;             // The node's "payload"
    Node<T>* next;      // pointer to the "next" node
};

template <class T>
class List{
    private:
        Node<T>* head;  // pointer to beginning of list
        int size;       // number of elements
```

Deleting a node from a linked list

Example: Delete node 43. Starting at "head" iterate to the Node to be deleted. You will need references to the "node to delete" and the "previous node" in order to have a handle on all the references you'll need to accomplish this task. We need to "prune" the "node to delete" out of the structure by removing all valid references.



You'll need to re-route Node 6's ***next reference*** to point to Node 76
The address of Node 76 is stored as ***next*** in Node 43



When that has been finished, you will need to deallocate the memory for Node 43 or valgrind (and your professor) will not be happy.

Exercises: Complete the following exercises by implementing the behaviors in both list classes. For each method include detailed analysis of the efficiency and the ultimate O notation. The ***compact array*** needs to maintain its "gap free" organization with the first item always at index 0

- int count(items)   // how many instances of "item" are there in the list?
- void remove_duplicates()   // remove all duplicates from the list
- void reverse()   // reverse the list
- void append(&list)   // appends parameter "list" to "this" list

Analysis: When both implementations have been finished, provide a written analysis **(in your own words. Not AI)** of the two approaches. Compare and contrast the two implementations across all behaviors. Include the pros and cons of each. Include Big O notation for each and details on how you came up with your notation.

You are not required to design an experiment and graph the efficiency of each algorithm but it would be a great way to illustrate the differences.

- Where does the array implementation perform well?
- What are the characteristics of arrays that contribute to this performance?
- Where does the linked list perform well?
- What are the characteristics of linked lists that contribute to this performance? Are there any tradeoffs?
- Are there any tradeoffs to achieve the good performance? Can you quantify?

Application: Put your linked list to work in the following scenario. You have been asked to create an application that assigns students to dorms.

You are required to use your own structures. Failure to do so will result in an egregious penalty.

- There are four dorms: Gryffindor, Slytherin, Ravenclaw, and Hufflepuff, but new dorms are currently being built so the application needs to be flexible and be able to handle the addition of new dorms in the future. There is a master list of dorms that the program will need to read from, and the program should be able to handle new additions without modifications. ***It is possible that your work will be graded with more (or fewer) dorms than are currently on this list, so you may want to test this.***

- You have also been provided with a listing of all the students along with their ID. The goal of the application is to assign each student to a dorm and build a student roster. The order of the students on the roster does not matter.

- After consultation with your team, you have decided to design a Student class to use with the linked lists to organize the assignments. Each dorm will be assigned its own list of students that will store the assignments as they happen. With the current list of 4 dorms, you will need 4 lists, but you will need to handle ***N*** dorms in order to deal with future expansion. You have been provided with this file.

Process: Read the master dorm list into a structure. A ***hash map*** (think Python Dictionary) would be a great choice to store the dorms. The key could be a string to store the dorm name and the value could be the list that will hold the assigned students. If you want to create a dorm class that would also be a good idea. It is not required though. Building a dorm class with the roster as a member would obviate the need for the hash map, as you could just use a regular list. This design is 100% up to you.

Once the dorms have been read in you will need to process the student file. The only criteria for dorm assignment is that the current student should be assigned to the dorm with the smallest number of students.

If there are multiple dorms with the same number of students, randomly choose between them.

Reassignment: Students can be re-assigned to a different dorm once the rosters have been finalized. Design a system that accomplishes this task. Include comments that describe the efficiency of the system and include the big O notation. Also include details on you how you derived your notation.

Finally, output each roster to its own text file. The name of the file should the ***the_dorm_name.txt***

For example: Gryffindor.txt

Submission: Push all files to your repo. Please use logically named subdirectories for organization.