[A-11] Nov. 7

Recitation - Week 11

This assignment is due Monday, November 10, 2025 at 05:00 CST. See the syllabus for our late policy.

Launch Autograder

Agenda

- (5 minutes) Announcements
- (20 minutes) Presentation: MP DNA Splicing
- (10 minutes) TA Introduction to Recitation Activities
- (10 minutes) Work on Linked List PrairieLearn Activity
- (30 minutes) Work on `Student Queue` Activity
- (If time remains, or at home) Work on Binary Tree PrairieLearn Activity

Part 1:

For the first part of recitation this week, you will work in groups to complete the Linked List `PushBack` activity. Once you have completed 1 variant with 100%, move on to Part 2.

If you finish Part 2 before the end of recitation, your group should complete the Binary Tree `InsertChild` activity.

We highly recommend practicing more than 1 variant of both activities, either after you complete Part 2 or later at home. On Saturday at 5am, we will release the "Independent Practice" version of both activities on PrairieLearn for your independent study.

You will be expected to be familiar with the linked list and binary tree editors in PrairieLearn. If you experience issues, please let us know on Discuss.

Part 2:

During the remainder of this recitation secition, you will be working on the `Student Queue` activity below. This activity is intended to give you more practice with traversing and manipulation Linked Lists, both of which will be relevant to MP6: DNA Splicing.

To create your personal repository for this assignment, click on the button below to access the template repository. Select **"Use this template"** from the top-right corner, followed by **"Create a new repository"**. In the dialog that pops up, set yourself as the owner, enter in "recitation-student-queue" as the repository name, and set the visibility to **Private**. Once you have created your repository, you are ready to clone and start working!

Template Repository

Activity: Student Queue

Ungraded Autograder

This is due on Monday, November 10, 2025 at 05:00:00 CST

This is not a graded activity.

A Linked List is a dynamic structure that stores data in the form of a sequence of nodes. While Linked Lists may have much of the same base functionality that Arrays do, they offer much more efficient insertions and deletions.

In this activity, you will implement a priority queue using a Linked List that manages students as they join office hours. Our system will maintain an order for students on the queue, such that students with higher class attendance are helped first. Ties between students are broken by prioritizing the student with a lower GPA.

You'll work with the StudentQueue class, which uses Nodes containing Student objects to represent students on the queue.

What is a priority queue?

A priority queue is an abstract data type that stores elements in a specific order based on their assigned priority. Each new element is placed at the appropriate position in the queue to reflect its priority, ensuring that elements with higher priority appear before those with lower priority. When multiple elements share the same priority, they are sorted by their arrival time within that priority "group". This ensures that elements that arrived first (within that priority "group") are dequeued before those that arrived later.

In our Office Hours Queue System:

Students with higher attendance percentages have higher priority. If attendance percentages are equal, the student who has a lower GPA has higher priority.

Here's a brief overview of these structures:

```
    // class header in node.hpp
```

```
struct Node {
    Node(const Student& student): student_(student),
next_(nullptr) {}
    Student student_;
    Node* next_ = nullptr;
};

// class header in student.hpp
class Student {
public:
    Student(const std::string& name, double gpa, double
attendancePercentage);
    std::string GetName() const;
    double GetGPA() const;
    double GetAttendancePercentage() const;
    bool operator<(const Student& other) const; // TODO
private:
    std::string name_;
    double gpa_ = 0.0;
    double attendancePercentage_ = 0.0;
};

// class header in student_queue.hpp
class StudentQueue {
public:
    StudentQueue() = default;
    ~StudentQueue();
    StudentQueue(const StudentQueue&); // TODO
    StudentQueue& operator=(const StudentQueue&); // TODO
    void Enqueue(const Student& student); // TODO
    void Dequeue(); // TODO
    Student Front() const;
    void Display() const;

private:
    Node* head_ = nullptr;
    Node* tail_ = nullptr;
    void Clear(); // TODO
```

```
};
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
```

```
29
30
31
32
33
34
35
36
37
38
```

Functions to Implement:

1. Overloaded Comparison Operator

```
bool Student::operator<(const Student& other) const
```

- Compares the current `Student` object with another `Student` object.
- Returns `true` if the current student has a lower priority than `other` based on specified criteria, such as attendance and GPA.
- Used to maintain ordering in data structures requiring comparisons, such as sets or priority queues.
- Handles cases where primary comparison (attendancePercentage_) criteria are equal by applying secondary criteria (gpa_).
- Students with higher attendance percentages have higher priority. If attendance percentages are equal, the student who has a lower GPA has higher priority.

2. Enqueue Function

```
void StudentQueue::Enqueue(const Student& student)
```

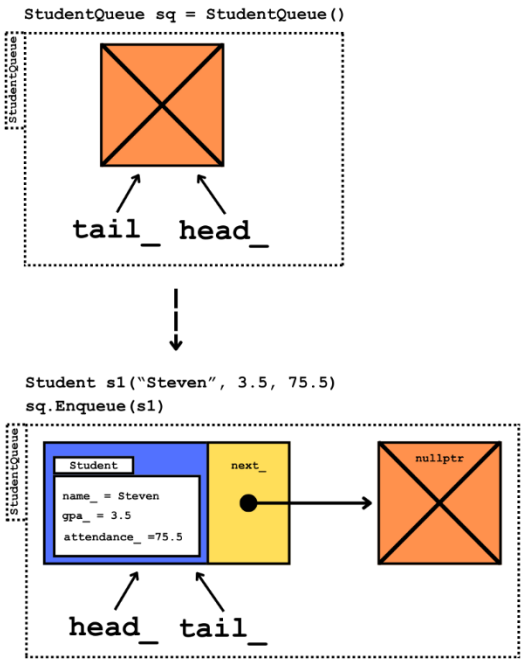
Inserts a new student onto the queue in the appropriate location, based on our protocol for determining priority.

In order to implement this behavior, you will use the previously overloaded comparison operator in order to determine the correct position a given student should be inserted into.

Below are example cases to consider when implementing the Enqueue function:

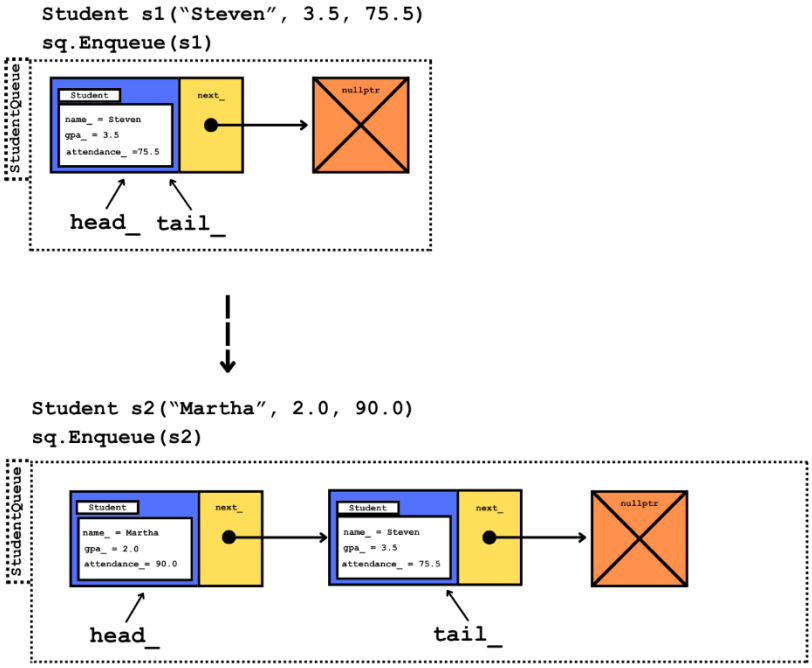
Case 1: Empty Queue (0 → 1 node)

- If the queue is empty, create a new node and point both head and tail to it.



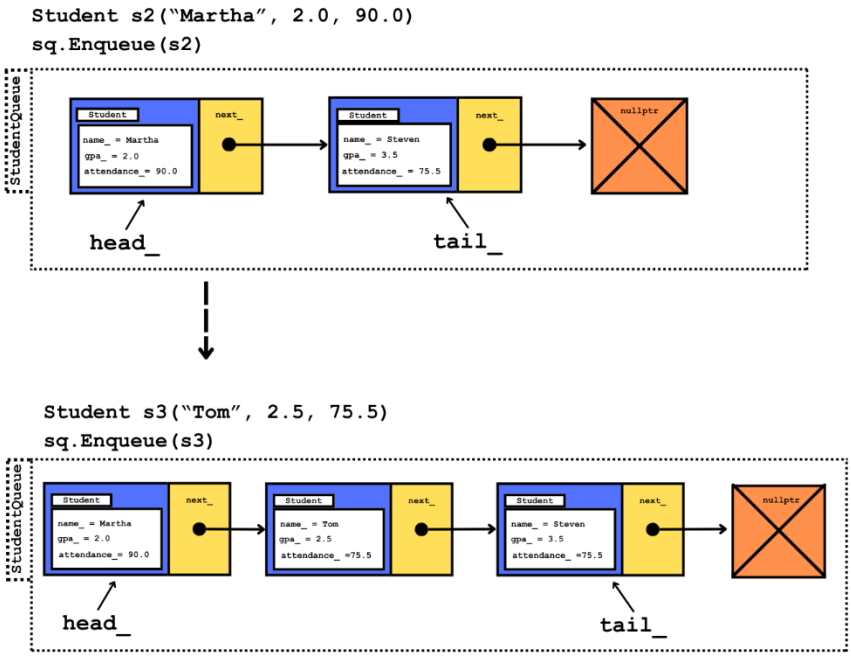
Case 2: Insert at Front (1 → 2 nodes)

- If the new student has higher priority than the current head, they should be inserted at the front.
- Update the head pointer appropriately.



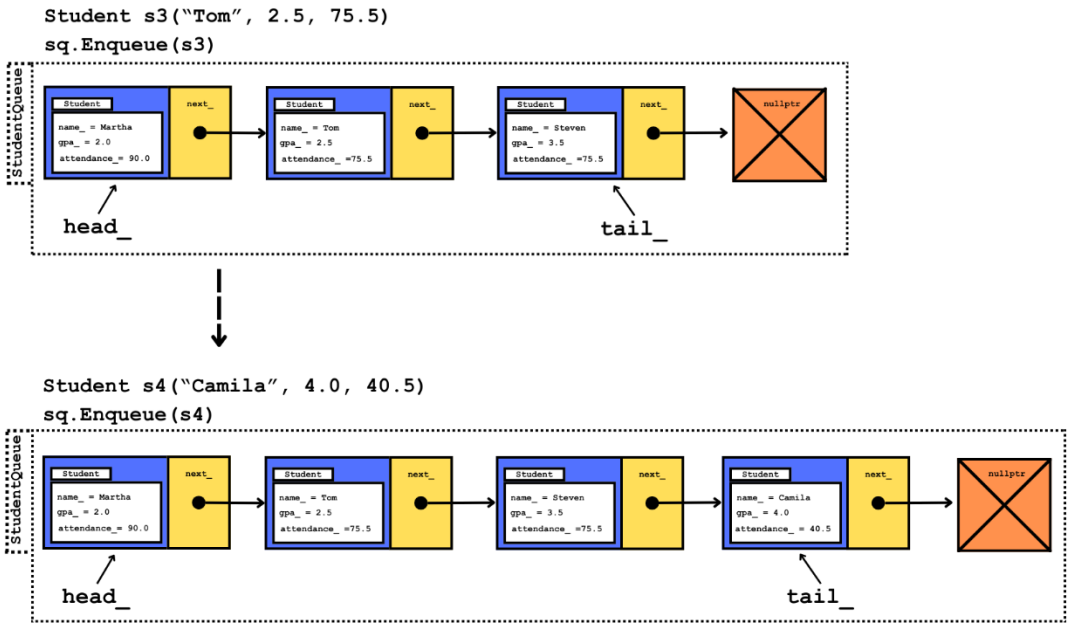
Case 3: Insert in Middle (2 → 3 nodes)

- Insert between existing nodes based on relative priority.
- Handle ties using GPA, where a lower GPA denotes a higher priority.
- In this case, although Tom and Steven have the same attendance rate, Tom has a lower GPA, so he is placed before Steven.



Case 4: Insert at End (3 → 4 nodes)

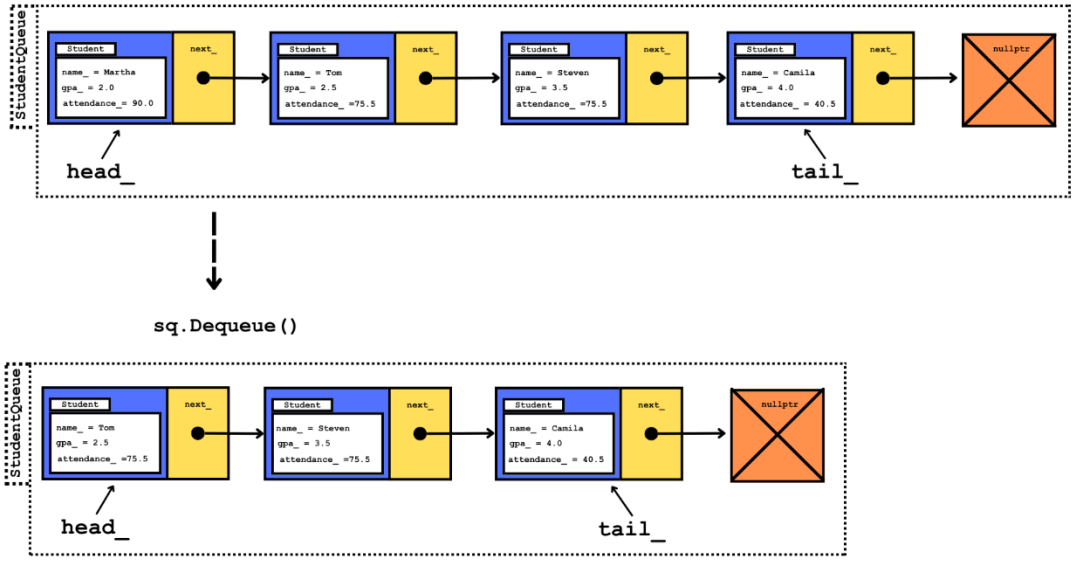
- If the new student has a lower priority than all nodes currently in the queue, they should be placed at the end.
- Update tail pointer appropriately.



3. Dequeue Function

```
void StudentQueue::Dequeue()
```

- Removes the first node from the queue.
- Updates head pointer appropriately.
- Properly deallocates removed node.
- If the queue is empty, simply return.



4. Clear Function

```
void StudentQueue::Clear()
```

- Deallocates all nodes in the queue.
- Resets head and tail pointers to nullptr.
- Must be used by destructor and assignment operator.
- Should handle empty queues correctly.

5. Copy Constructor

```
StudentQueue::StudentQueue(const StudentQueue& other)
```

- Creates a deep copy of the provided StudentQueue.
- Must properly handle empty queues.
- Must maintain the same order as the original queue.
- All nodes must be newly allocated.
- Ensure that head_ and tail_ pointers are properly updated.

The video below depicts usage of the copy constructor to instantiate a new StudentQueue object.

Note that when completed, each node in new_queue has a different memory address from those on old_queue.

6. Copy Assignment Operator

```
StudentQueue& StudentQueue::operator=(const StudentQueue& other)
```

- Performs deep copy of the provided StudentQueue.
- Must handle self-assignment correctly.
- Must clear existing nodes before copying.
- Must maintain the same order as the original queue.

The video below depicts usage of the copy assigment operator to update the values of a previously constructed StudentQueue object.

Note that the nodes previously stored in old_queue must first be deleted before the process of deep copying can continue as in the previous video.

Once again, we can observe that the nodes of old_queue and new_queue contain different memory addresses, resulting in a successful deep copy.

Important Notes:

- Always check for nullptr when traversing the list.
- Properly maintain both head_ and tail_ pointers.
- Remember to properly deallocate memory used to prevent leaks.
- Test each case thoroughly before submission.

Drag files here or click to upload
All file names must match one of the accepted files

student.cc

student.hpp

student_queue.cc

student_queue.hpp

Grade
Save
History

```
> Click Grade to submit and grade your code.
```