LW: Linked List Operations

For this labwork, you will be completing two required functions and two optional functions for a representative example of a linked list data structure. The linked list consists of two classes for representing the linked list and its nodes, respectively.

- Refer to **Background** for a brief overview of the linked list data structure that you will be coding.
- Refer to **Grading** for details of satisfying labwork completion. Work with others in your group on the labwork, but submit individually on Gradescope.
- Refer to **Instructions** for information on completing the different tasks in the labwork.

Objectives

- Correctly code common functions of a linked list implementation.
- Correctly allocate and deallocate memory on the heap with a linked list data structure.
- Use pointers to correctly create and destroy a linked list.
- Use pointers to correctly insert and remove nodes from a linked list.

Grading

As stated in the syllabus, you must complete both items below to receive credit for this week's lab. The Teaching Assistants will strictly enforce attendance that is described in this section at the start and end of your lab.

Submission Grade

- **Attend your assigned lab.** TAs will be checking that you are actively working on labwork throughout the lab.
- **Complete the labwork.** The minimum score on Gradescope that **you individually** need to achieve to receive labwork credit before the deadline is **40 points**.

Attendance Grade

- Your TA will mark your attendance at the start of class, and will confirm your attendance after your group shows the minimum required completed work to be checked by your TA.
- If you do not attend your lab at the start of class or if you do not receive confirmation from your TA when your group submits, then your attendance will not be recorded.

Makeup Work

Before you can do any make-up work, you must provide your instructor with any documentation for your excused absence.

Submission

You must download the starter code. Once you add the required methods, you can execute the application locally and submit it to Gradescope for unit tests. The Gradescope submission requires the following files:

- `linked_list.cpp:` the linked list implementation that you will complete
- `linked_list.h:` the linked list header file with the required function prototypes

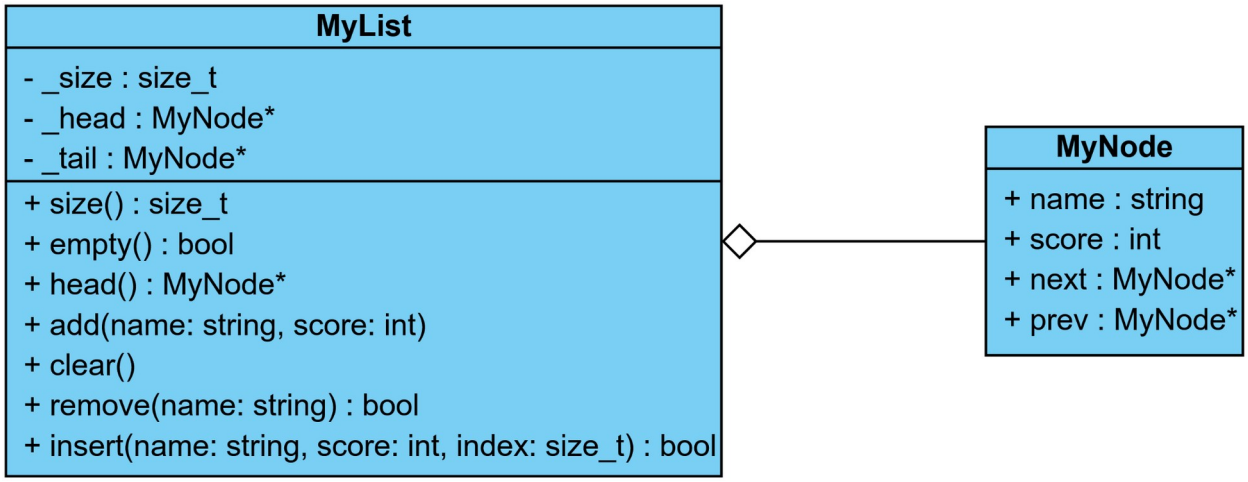
You do not need to modify `linked_list.h` in the starter code.

Instructions

Below are the full set of instructions for this labwork. The Teaching Assistants will provide assistance in this section while your group is working on the labwork.

Task 1: Familiarize yourself with the starter code

The starter code defines the classes `MyList` and `MyNode`. The following figure illustrates the relationship between these classes. You can also find additional information of the `MyList` and `MyNode` classes in `linked_list.h`.



Task 2: Implement add(const std::string&, int)

About

For the second task, you will implement a function that adds a single node to the end of the linked list. The parameters will be used to create a node object that will be added to the linked list.

Parameters

The function consists of two parameters:

- `std::string name:` The name value of the node object.
- `int score:` The score value of the node object.

Additional Information

Documentation for `add(const std::string&, int)` can be found in `linked_list.h`.

Task 3: Implement clear()

About

For the third task, you will implement a function that removes all the nodes from the linked list.

Additional Information

Documentation for `clear()` can be found in `linked_list.h`.

[OPTIONAL] Task 4: Implement remove(const std::string&)

About

For the fourth task, you will implement a function that removes a single node from the linked list.

Parameter

The function consists of one parameter:

- `std::string name:` The name value of the node object to remove.

Additional Information

Documentation for remove(const std::string&) can be found in linked_list.h.

[OPTIONAL] Task 5: Implement insert(const std::string&, int, size_t)

About

For the optional fifth task, you will implement a function that inserts a single node into the linked list.

Parameters

The function consists of three parameters:

- `std::string name:` The name value of the node object to insert.
- `int score:` The score value of the node object to insert.
- `size_t index:` The index to insert the node into.

Additional Information

Documentation for `insert(const std::string&, int)` can be found in `linked_list.h`.

Examples

Dark Red Bold text is program output.

Main

```
int main() {
   MyList list;
   cout << list << endl;
   return 0;
}
```

Expected Output

```
<empty>
```

Main

```
int main() {
   MyList list;
   list.add("Juan", 95);
   list.add("Jill", 98);
   list.add("Joon", 90);
   cout << list << endl;
   return 0;
}
```

Expected Output

```
[ Juan, 95 ] --> [ Jill, 98 ] -->
[ Joon, 90 ]
```

Main

```
int main() {
   MyList list;
   list.add("Juan", 95);
   list.add("Jill", 98);
   cout << list << endl;
   cout << endl;
   list.remove("Jill");
   cout << list << endl;
   return 0;
}
```

Expected Output

```
[ Juan, 95 ] --> [ Jill, 98 ]

[ Juan, 95 ]
```

Main

```
int main() {
   MyList list;
   list.add("Juan", 95);
   list.add("Jill", 98);
   cout << list << endl;
   cout << endl;
   list.clear();
   cout << list << endl;
   return 0;
}
```

Expected Output

```
[ Juan, 95 ] --> [ Jill, 98 ]

<empty>
```

Main

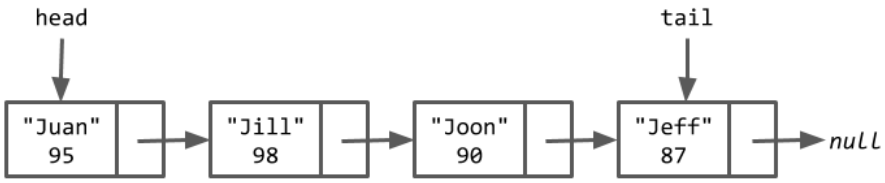
```
int main() {
   MyList list;
   list.add("Juan", 95);
   list.add("Jill", 98);
   list.insert("Joon", 90, 1);
   cout << list << endl;
   return 0;
}
```

Expected Output

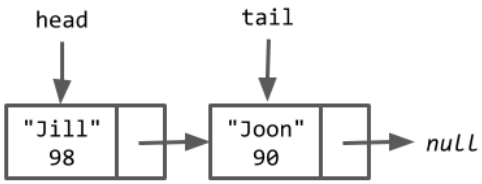
```
[ Juan, 95 ] --> [ Joon, 90 ] -->
[ Jill, 98 ]
```

Background

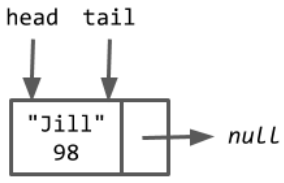
The following examples show different variations of a singly linked list in this labwork, as well as example ways of adding and removing nodes. *Note that the UML diagram includes a previous pointer on the node class, which is only necessary for a doubly linked list. You do not need to create a doubly linked list, but you may do so if you wish. None of the test cases expect a doubly linked list, but a doubly linked list will work correctly (assuming it is implemented correctly) for all tests.*



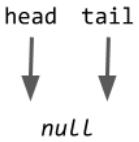
Example 1. *A singly linked list with four elements and distinct head and tail pointers.*



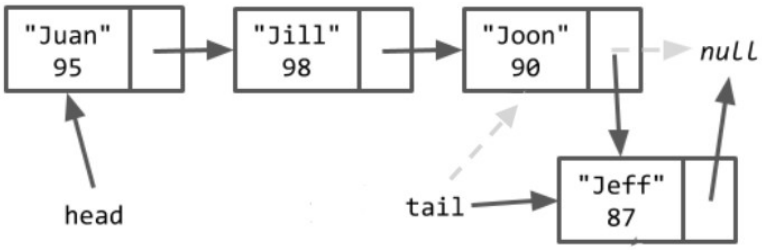
Example 2. *A singly linked list with two elements and neighboring head and tail pointers.*



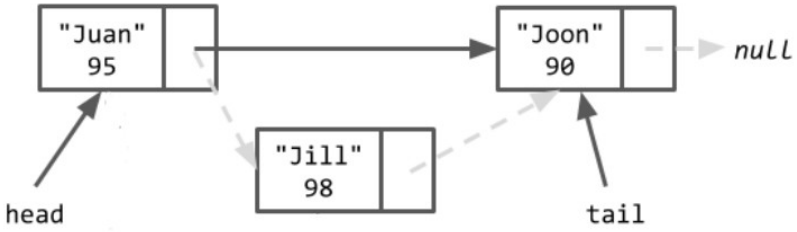
Example 3. *A singly linked list with one elements and shared head and tail pointers.*



Example 4. *An empty singly linked list with head and tail pointers pointing to null.*



Example 5. *An example of adding a node.*



Example 6. *An example of removing a node.*