CSCI 2270 – Data Structures Fall 2022

Assignment 4 - Linked Lists (Part-II)

Show time scheduler

OBJECTIVES

1. **Delete, re-adjust, and detect loop in a linked list**
2. **Practice implementing classes in C++**

NOTE: This assignment is an extension of Assignment 3.

Continuation of Previous Assignment: Building your own show schedule

This assignment builds on the previous one. You will need to reuse the ShowList class implementation you already completed, and then add to it as described in this readme.

There are three files in your repository containing a code skeleton to get you started. *Do not modify the header file or your code won't work!* You will have to complete both the class implementation in ShowsList.cpp and the driver file main_1.cpp.

NOTE: When experimenting with your program, make sure to run your program ./run_app_1 WITHOUT command line arguments.

The linked-list will be implemented using the following struct (already included in the header file):

```
//Show: node struct that will be stored in the ShowsList linked-list
    struct Show
    {
      std::string name;         // name of the show
      double rating;         // average rating this show has received
      int numberRatings;    // total count of ratings received by this show
      Show *next;            // pointer to the next show
    };
```

Class Specifications

The **ShowsList** class definition is provided in the file *ShowsList.hpp*. *Do not modify this file or your code won't work!* Fill in the file *ShowsList.cpp* according to the following specifications. **Make sure you complete the functions in the order they are given here.**

You should be able to use your solutions from Assignment-3 for the first part of this assignment.

Show* head;

➔ Points to the first node in the linked list

ShowsList();

➔ Class constructor; set the head pointer to NULL

bool isEmpty();

➔ Return true if the head is NULL, false otherwise

void displayShows();

➔ Print the names of each node in the linked list. Below is an example of correct output using the default setup. (Note that you will **cout << "NULL"** at the end of the list)

```
== CURRENT SHOWS LIST ==
Friends-> Ozark-> Stranger Things-> The Boys-> Better Call Saul-> NULL
===
```

➔ If the show list is empty then print *"nothing in path"* with an endline.

void addShow (Show* previousShow, string showName); *// Beware of edge cases*

➔ Insert a new show with name **showName** in the linked list after the show pointed to by **previousShow**.

➔ If **previousShow** is NULL, then add the new show to the beginning of the list.

➔ Print the name of the show you added according to the following format:

```
// If you are adding at the beginning use this: cout
<< "adding: " << showName << " (HEAD)" << endl;

// Otherwise use this:
cout << "adding: " << showName << " (prev: " <<
previousShow->name << ")" << endl;
```

void buildShowsList();

➔ Add the following five shows, in order, to the list with **addShow**: "Friends", "Ozark", "Stranger Things", "The Boys", "Better Call Saul". This is the default setup of shows in the linked list, therefore, you can keep 0 as the **rating** value and the **numberRatings** for each of them.

Show* searchShow(string showName);

➔ Return a pointer to the node with name **showName**. If **showName** cannot be found, return NULL

void addRating(string receiver, double rating);

➔ Find the node with the name **receiver**. For the receiver node, update the rating. You will update the rating following the running average rating formula of the previous assignment. You will also update the numberRatings.

- `node->numberRatings++;`
- `node->rating = (node->rating*(node->numberRatings-1) + rating)/node->numberRatings`

If the list is empty, print "Empty list" and exit the function. If the node is not present, print “Show not found”. For both cases, change the line after printing.

Otherwise print the updated rating using following cout assuming temp is pointing to the concerned node

```
cout << "The rating has been updated: " << node->rating << endl;
```

TODO Implementations in that order (New for Assignment-4)

ShowList* produceGlitch(string showName);

/* Do not make any changes to this function. This function is already implemented for you. */

- As a way to test the **`findGlitch()`** function, we have developed a **`produceGlitch()`** function that adds a loop to the linked list pointed to by head.
- This implementation is achieved by creating a link from the last node in the linked list to an intermediate node. The function takes as argument the **showName** of that intermediate node to loop back into.
- The function returns the last node of the linked list before creation of the loop. This will be needed by the driver function to break the loop.
- For example, consider the linked list: `A -> B -> C -> D -> E -> NULL`. Suppose the function is called as -- **`produceGlitch`**(“C”);

After execution of the function the linked list should be `A -> B -> C -> D -> E -> C` and it will return a pointer to the node `E`.

NOTE: node E was the last node before creation of the loop.

bool findGlitch();

➔ Someone has messed up your show schedule and now there’s a glitch. The glitch is the presence of a loop in your show list. You need to implement findGlitch() function with the following specifications to detect a loop.

- Traverse through the linked list (pointed to by ***head***) to detect the presence of a loop. A loop is present in the list when the tail node points to some intermediate node (including itself) in the linked list, instead of pointing to null value.

  For example, the following list with `A` at the head has a loop: `A -> B -> C -> D -> E -> B`. Notice that all the nodes are unique, except for node `B` which is repeated twice. This means that the last unique node `E` is connected back to the node `B` that appears before it in the linked list.

- Return **`true`** if the list contains a loop, else return **`false`**.
- Refer to the following links for the algorithm of loop detection:
  - https://www.youtube.com/watch?v=apIw0Opq5nk
  - https://www.youtube.com/watch?v=MFOAbpfrJ8g

void removeShows(string showName); *// Beware of edge cases*

➔ Traverse the list to find the node with name **showName**, then delete it from the list. If there is no node with name **showName**, print *"Show does not exist."* with an endline

void purgeCompleteShowList();

➔ If the list is empty, do nothing and return.

➔ Otherwise, delete every node in the linked list, starting from the head, and set head to NULL. Print the following statement for every node being deleted

```
cout << "deleting: " << showName << endl;
```

Print *"Deleted show list"* with an endline after purging the complete list.

void rearrangeShowList(int start, int end);

➔ Manipulate next pointers to re-adjust the linked list. Here, **start** is the index of a node from starting. Similarly **end** is the index of a node at the end of the chunk. The function will send the chunk of the linked list from **start** index to **end** index to the end of the linked list. Consider the node at head as index 0.

For example, if you have linked list like this: `A -> B -> C -> D -> E-> NULL`, and `start=1` and `end=3`, then the linked list after **`rearrangeShowList`** should be `A -> E -> B -> C -> D-> NULL`.

If you have linked list like this: `A -> B -> C -> D -> NULL`, and `start=0` and `end=2`, then the linked list after **`rearrangeShowList`** should be `D-> A -> B -> C -> NULL`. Here, `D` is the new head.

- If the linked list is empty, `print "Linked List is Empty"`.
- If `end` is bigger than the number of nodes in the linked list or smaller than 0, then `print "Invalid end index"`.
- `end` should be lesser than the index of the last element in the linked list. Otherwise `print "Invalid end index"`.
- If `start` is bigger than the number of nodes in the linked list or smaller than 0, then `print "Invalid start index"`.
- If `start > end`, `print "Invalid indices"`.

NOTE: Change the order of the "node" (by manipulating the next pointers of each node), not the "value of the node".

Main driver file

NOTE: Main driver file is provided in starter code. You do not have to code it. You can use that to test your functions. We will walk through a brief introduction of the driver here

Your program will start by displaying a menu by calling the **displayMenu** function included in main.cpp. The user will select an option from the menu to decide what the program will do, after which, the menu will be displayed again. The specifics of each menu option are described below.

Option 1: Build schedule

This option calls the **buildShowsList()** function, then calls the **displayShows()** function. You should get the following output:

```
adding: Friends (HEAD)
adding: Ozark (prev: Friends)
adding: Stranger Things (prev: Ozark)
adding: The Boys (prev: Stranger Things)
adding: Better Call Saul (prev: The Boys)
== CURRENT SHOWS LIST ==
Friends-> Ozark-> Stranger Things-> The Boys-> Better Call Saul-> NULL
===
```

Option 2: Display Shows

Call the **displayShows** function. Output should be in the format below:

```
// Output for the default setup
== CURRENT SHOWS LIST ==
Friends-> Ozark-> Stranger Things-> The Boys-> Better Call Saul-> NULL
===
// Output when the linked list is empty
nothing in path
```

Option 3: Add Show

Prompt the user for two inputs: the name of a new show to add to the list, **showName**, and the name of a show already in the list, **previousShow**, which will precede the new show. Use the member functions **searchShows** and **addShows** to insert **showName** into the linked-list right after the node with the show name **previousShow**.

- If the user wants to add the new show to the head of the list then they should enter "First" instead of a previous show name.
- If the user enters an invalid previous show name (not present in the linked list), then you need to prompt the user with the following error message and collect input again until they enter a valid previous show name or "First":

```
cout << "INVALID(previous show name)... Please enter a VALID previous show name!" << endl;
```

- Once a valid previous show name is passed and the new show is added, call the function **displayShows** to demonstrate the new linked-list.

For example, the following should be the output if the linked-list contains the default setup from option (1) and the user wants to add Vikings after Ozark:

```
Enter a new show name:
Vikings
Enter the previous show name (or First): Ozark

adding: Vikings (prev: Ozark)
== CURRENT SHOWS LIST ==
Friends-> Ozark-> Vikings-> Stranger Things-> The Boys-> Better Call Saul-> NULL
===
```

Option 4: Add Rating

Prompt the user for two inputs: Show Name, and the Rating *(Hint: use **getline** in case there are spaces in the user input)*. Pass the Show name and rating to the **addRating**

For example, the following should be the output if the linked-list contains the default setup from option (1):

```
Example 1:
Enter name of the show to add the rating:
Ozark
Enter the rating:
4.5
The rating has been updated: 4.5
```

If the user then decides to add rating to show “Peaky Blinders” which does not exists in the show list

```
Example 2:
Enter name of the show to add the rating:
Peaky Blinders
Enter the rating:
4.5
Show not found
```

Option 5: Delete Show

Prompt the user for a show name, then pass that name to the **removeShows** function and call **displayShows** to demonstrate the new linked-list.

For example, the following should be the output if the linked-list contains the default setup from option (1) and the user wants to delete Ozark:

```
Enter a show name:
Ozark
== CURRENT SHOWS LIST ==
Friends-> Stranger Things-> The Boys-> Better Call Saul-> NULL
===
```

Option 6: Produce and Find Glitch in Show List

Call the **`produceGlitch`** and **`findGlitch`** functions.

User will be prompted to enter the name of the show to loop back. **`produceGlitch`** function will be called to create the loop accordingly. After that **`findGlitch`** will be called. Depending on the status of loop creation **`findGlitch`** will return either **`true`** (if the loop is created) or **`false`** (if the loop could not be created). After calling **`produceGlitch`** function, If there is a loop it will be broken by the driver (*refer to the starter code for more detail*). So, in this operation, a loop is created in the linked list (if appropriate input is given) and it is removed immediately.

```
Enter the Show name to loop back to:
Friends
Show list contains a loop.
Breaking the loop.
```

Option 7: Rearrange Show List

Call the **rearrangeShowList** function, then the **displayShows** function. User should be prompted to input the `start` index and `end` index.

For example, the following should be the output if the linked list contains the default setup from option (1):

```
Enter the start index:
1
Enter the end index:
2
== CURRENT SHOWS LIST ==
Friends-> The Boys-> Better Call Saul-> Ozark-> Stranger Things-> NULL
===
```

Option 8: Clear Schedule

Call the **purgeCompleteShowList** function. For example, deleting the default network should print:

```
Show list before deletion
== CURRENT SHOWS LIST ==
Friends-> Ozark-> Stranger Things-> The Boys-> Better Call Saul-> NULL
===
deleting: Friends
deleting: Ozark
deleting: Stranger Things
deleting: The Boys
deleting: Better Call Saul
Deleted show list
Show list after deletion
nothing in path
```

Option 9: Quit

Print the following message:

```
cout << "Quitting..." << endl;
```

Finally, print the following before exiting the program:

```
cout << "Goodbye!" << endl;
```

Important Note regarding Test Cases

In order to test the functions specific to Assignment-4, you will need to implement functions from Assignment-3 as well (you can use your solutions from the previous assignment).