# CSCI 2270 – Data Structures Spring 2020

*Instructors: Zagrodzki, Ashraf, Trivedi*

Assignment 4

Due Sunday, February 16, 11:59PM

# Assignment 4 - Linked Lists

**Communication Between Towers**

OBJECTIVES

**1. Delete, readjust, and detect loop in a linked list**
**2. Get practice implementing classes**

**This assignment is an extension of assignment 3.**

## Background

In the Lord of the Rings trilogy, there is a scene where a warning beacon is lit in the towers of Minas Tirith, which is seen by a second beacon, prompting them to light their own fire which a third beacon sees, and so forth. This was a rapid means of communication in the days before telegraphs were invented. In this assignment, you're going to simulate a communications network using a linked list. Each node in the list will represent a country and you need to be able to send a message between nodes from one side of the world to the other.

## Building your own communications network

You will be implementing a class to simulate a linear communication network between countries. There are three files in Moodle containing a code skeleton to get you started. *Do not modify the header file or your code won't work in Moodle!* You will have to complete both the class implementation in CountryNetwork.cpp and the driver file main.cpp.

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

```
struct Country
{
    string name;          // name of the country
    string message;       // message this country has received
    int numberMessages;   // no. of messages passed through this country
    Country *next;        // pointer to the next country
};
```

## Class Specifications

The **CountryNetwork** class definition is provided in the file *CountryNetwork.hpp* in Moodle. *Do not modify this file or your code won't work on Moodle!* Fill in the file *CountryNetwork.cpp* according to the following specifications.

**Country* head;**

➔ Points to the first node in the linked list

**CountryNetwork();**

➔ Class constructor; set the head pointer to NULL

**bool isEmpty();**

➔ Return true if the head is NULL, false otherwise

**void insertCountry(Country* previous, string countryName);** *// Beware of edge cases*

➔ Insert a new country with name **countryName** in the linked list after the country pointed to by **previous**. If **previous** is NULL, then add the new country to the beginning of the list. Print the name of the country you added according to the following format:

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

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

**void deleteCountry(string countryName);** *// Beware of edge cases*

➔ Traverse the list to find the node with name **countryName**, then delete it. If there is no node with name **countryName**, print *"Country does not exist."*

**void loadDefaultSetup();**

➔ First, delete whatever is in the linked list using the member function **deleteEntireNetwork**. Then add the following six countries, in order, to the network with **insertCountry**: "United States", "Canada", "Brazil", "India", "China", "Australia"

**Country* searchNetwork(string countryName);**

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

**void deleteEntireNetwork();**

➔ If the list is empty, do nothing and return. Otherwise, delete every node in the linked list and set **head** to NULL. Print the name of each node as you are deleting it according to the following format:

```
cout << "deleting: " << node->name << endl;
```

After the entire linked list is deleted, print:

```
cout << "Deleted network" << endl;
```

**void readjustNetwork(int startIndex, int endIndex);**

➔ Manipulate **next** pointers to readjust the linked list. Here, ***startIndex*** is index of a node from starting. Similarly ***endIndex*** is index of a node from beginning. The function will send the chunk of the link list between start index and end index at 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 ***startIndex=1 and endIndex=3***, then the linked list after readjustNetwork should be "**`A -> E -> B -> C -> D->  NULL`**".

If you have linked list like this: "**`A -> B -> C -> D -> NULL`**", and ***startIndex=0 and endIndex=2***, then the linked list after readjustNetwork 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 ***endIndex*** is bigger than the number of nodes in the linked list or smaller than ***0***, then print "*Invalid end index*".

➔ ***endIndex*** should be lesser than the index of the last element in the linked list. Otherwise print "*Invalid end index*".

➔ If ***startIndex*** is bigger than the number of nodes in the linked list or smaller than ***0***, then print "*Invalid start index*".

➔ If ***startIndex > endIndex*** print "Invalid indices".
[NOTE: Change the order of the "node" (by manipulating  the next pointers of each node), not the "value of the node"]

**bool detectLoop();**

➔ 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

**Country * createLoop(string countryName);**

- As a way to test the detectLoop() function, develop a createLoop() function that adds a loop to the linked list pointed to by **head**.
- You'll achieve this by creating a link from the last node in the linked list to an intermediate node. The function takes as argument the country name of that intermediate node to loop back into.
- The function should return 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 --

  **`createLoop("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.
- If the country is not present in the linked list, the function should return without creating a loop. A pointer to the last node should still be returned.

**void printPath();**

- 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 path)

```
== CURRENT PATH ==
United States -> Canada -> Brazil -> India -> China -> Australia -> NULL
===
```

- If the network is empty then print *"nothing in path"*

## TO DOs

You need to implement-

1. **void deleteCountry(string countryName)**
2. **void deleteEntireNetwork()**
3. **Country * createLoop(string countryName)**

4. **bool detectLoop()**
5. **void readjustNetwork(int startIndex, int endIndex)**

**Other functions are implemented in the starter code.**

## Main driver file [ Main driver is provided in starter code]

*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 Network

This option calls the **loadDefaultSetup** function, then calls the **printPath** function. You should get the following output:

```
adding: United States (HEAD)
adding: Canada (prev: United States)
adding: Brazil (prev: Canada)
adding: India (prev: Brazil)
adding: China (prev: India)
adding: Australia (prev: China)
== CURRENT PATH ==
United States -> Canada -> Brazil -> India -> China -> Australia -> NULL
===
```

### Option 2: Print Network Path

Calls the **printPath** function. Output should be in the format below:

```
// Output for the default setup
== CURRENT PATH ==
United States -> Canada -> Brazil -> India -> China -> Australia -> NULL
===

// Output when the linked list is empty
== CURRENT PATH ==
nothing in path
```

```
===
```

### Option 3: Add Country

Prompt the user for two inputs: the name of a new country to add to the network, **newCountry**, and the name of a country already in the network, **previousCountry**, which will precede the new country. Use the member functions **searchNetwork** and **insertCountry** to insert **newCountry** into the linked-list right after the node with the country name **previousCountry**.

- If the user wants to add the new country to the head of the network then they should enter “First” instead of a previous country name.
- If the user enters an invalid previous city (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 country name or “First”:

  ```
  cout << "INVALID country...Please enter a VALID previous country
  name:" << endl;
  ```

- Once a valid previous country name is passed and the new country is added, call the function **printPath** 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 Colombia after Brazil:

```
Enter a new country name:
Colombia
Enter the previous country name (or First):
Brazil

adding: Colombia (prev: Brazil)
== CURRENT PATH ==
United States -> Canada -> Brazil -> Colombia -> India -> China -> Australia ->
NULL
===
```

### Option 4: Delete Country

Prompt the user for a country name, then pass that name to the **deleteCountry** function and call **printPath** 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 Canada:

```
Enter a country name:
```

```
Canada
== CURRENT PATH ==
United States -> Brazil -> India -> China -> Australia -> NULL
===
```

### Option 5: Create and Detect loop in network

Call the **createLoop** and **detectLoop** functions.

User will be prompted to enter the name of the country to loop back. **createLoop** function will be called to create the loop accordingly. After that **detectLoop** will be called. Depending on the status of loop creation **detectLoop** will return either true (if the loop is created) or false (if the loop could not be created). After calling **createLoop** 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.

```
#> 5
Enter the country name to loop back:
India
Network contains a loop
Breaking the loop
```

### Option 6: Re-adjust Network

Call the **readjustNetwork** function, then the **printPath** 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):

```
#> 6
Enter the start index:
1
Enter the end index:
2
== CURRENT PATH ==
United States -> India -> China -> Australia -> Canada -> Brazil -> NULL
===
```

### Option 7: Clear network

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

```
Network before deletion
== CURRENT PATH ==
United States -> Canada -> Brazil -> India -> China -> Australia -> NULL
===
deleting: United States
deleting: Canada
deleting: Brazil
deleting: India
deleting: China
deleting: Australia
Deleted network
Network after deletion
== CURRENT PATH ==
nothing in path
===
```

### Option 8: Quit

Print the following message:

```
cout << "Quitting... cleaning up path: " << endl;
```

Then call **printPath**, followed by **deleteEntireNetwork**. Now, check if the network is empty using **isEmpty**. If it is, print:

```
cout << "Path cleaned" << endl;
```

Otherwise, print:

```
cout << "Error: Path NOT cleaned" << endl;
```

Finally, print the following before exiting the program:

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