CSCI 2270 – Data Structures

Instructors: Zagrodzki, Ashraf

Assignment 4

Due Tuesday, September 29 2020, 11:59PM

Linked Lists - Part 2

Communication Between Buildings

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 this problem you're going to model a communication network among buildings in CU Boulder using a linked list. Each node in the list will represent a building and you need to be able to send a message between nodes from one side of the campus to the other.

Building your own communications network

You will be implementing a class to simulate a linear communication network between buildings. There are three files in Canvas containing a code skeleton to get you started. *Do not modify the header file or your code won't work in Canvas!* You will have to complete both the class implementation in CUBuildingNetwork.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 CUBuilding
{
    string name;          // name of the building
    string message;       // message this building has received
    int numberMessages;   // number of messages passed through this building
    CUBuilding *next;     // pointer to the next building
    int totalRoom;        // number of rooms in this building
};
```

Class Specifications

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

CUBuilding* head;

➔ Points to the first node in the linked list

CUBuildingNetwork();

➔ Class constructor; set the head pointer to NULL

bool isEmpty();

➔ Return true if the head is NULL, false otherwise

void addBuildingInfo(CUBuilding* previous, string buildingName, int numOfRooms);
// Beware of edge cases

➔ Insert a new building with name **buildingName** and **totalRoom** in the linked list after the building pointed to by **previous**.

➔ If **previous** is NULL, then add the new building to the beginning of the list.

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

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

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

void deleteCUBuilding(string buildingName); *// Beware of edge cases*

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

void loadDefaultSetup();

➔ First, delete whatever is in the linked list using the member function **deleteEntireNetwork**. Then add the following six buildings, in order, to the network with **addBuildingInfo**: "FLMG", "DLC", "ECOT", "CASE", "AERO", "RGNT". Room numbers are 2, 10, 6, 5, 4, 9 respectively.

CUBuilding* searchForBuilding(string buildingName);

➔ Return a pointer to the node with name **buildingName**. If **buildingName** 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

CUBuilding * createLoop(string buildingName);
- ➔ 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 building 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 building 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 printNetwork();
- ➔ 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 ==
FLMG(2) -> DLC(10) -> ECOT(6) -> CASE(5) -> AERO(4) -> RGNT(9) -> NULL
===
```

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

To Dos

You need to implement-
1. **void deleteCUBuilding(string buildingName)**
2. **void deleteEntireNetwork()**
3. **CUBuilding * createLoop(string buildingName)**
4. **bool detectLoop()**
5. **void readjustNetwork(int startIndex, int endIndex)**

Other functions are implemented in the starter code.

Main driver file

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: FLMG (HEAD)
adding: DLC (prev: FLMG)
adding: ECOT (prev: DLC)
adding: CASE (prev: ECOT)
adding: AERO (prev: CASE)
adding: RGNT (prev: AERO)
== CURRENT PATH ==
FLMG(2) -> DLC(10) -> ECOT(6) -> CASE(5) -> AERO(4) -> RGNT(9) -> NULL
===
```

Option 2: Print Network Path

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

```
// Output for the default setup
== CURRENT PATH ==
FLMG(2) -> DLC(10) -> ECOT(6) -> CASE(5) -> AERO(4) -> RGNT(9) -> NULL
===

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

Option 3: Add Building

Prompt the user for three inputs: the name of a new building(**newName**) to add to the network, number of rooms in the new building(**newNumRooms**), and the name of a building already in the network, which will precede the new building.(**prevName**) Use the member functions **searchForBuilding** and **addBuildingInfo** to insert the new building into the linked-list right after the node with the building name prevName.

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

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

- Once a valid previous building name is passed and the new building 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 ADEN after CASE:

```
Enter a new building name:
ADEN
Enter total room number:
5
Enter the previous building name (or First):
CASE
adding: ADEN (prev: CASE)
== CURRENT PATH ==
FLMG(2) -> DLC(10) -> ECOT(6) -> CASE(5) -> ADEN(5) -> AERO(4) -> RGNT(9) -> NULL
===
```

Option 4: Delete Building

Prompt the user for a building name, then pass that name to the **deleteCUBuilding** 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 DLC:

```
Enter a building name:
DLC
== CURRENT PATH ==
FLMG(2) -> DLC(10) -> ECOT(6) -> CASE(5) -> AERO(4) -> RGNT(9) -> 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 building 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 building name to loop back:
ECOT
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 ==
FLMG(2) -> CASE(5) -> AERO(4) -> RGNT(9) -> DLC(10) -> ECOT(6) -> NULL

===
```

Option 7: Clear network

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

```
Network before deletion
== CURRENT PATH ==
FLMG(2) -> DLC(10) -> ECOT(6) -> CASE(5) -> AERO(4) -> RGNT(9) -> NULL
===
deleting: FLMG
deleting: DLC
deleting: ECOT
deleting: CASE
deleting: AERO
deleting: RGNT
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;
```