CS302

Assignment 9: New City

Description

You just got off the phone with Galvin Belson and you've been hired to work for Hoolie. Which is a great opportunity, but the issue is you have to move to a new city and you've lived your whole life in the city in which you were born, so you might feel lost in this new city. Luckily you have all the street names listed out but some of the streets are only one way, which can make commuting an issue, so you want to use 2 way streets.

For the program, you will be given a set of streets (along with their connections), and you want to find all paths from a start street to a destination street that contain only 2 way streets, if there are no such paths, output no paths or some equivalent message. Each node can be though of as an intersection (which we will use a street name to denote the intersection) and an edge is the actually street that connects to another intersection (another node that we use a street name to denote this intersection). You will need to implement the following classes to perform this task.

Adjacency List

```
template <class Type>
class vertex
{
  struct node
  {
    Type item;
    node * link;
  };

public:
  class edgeIterator
  {
  public:
    friend class vertex;
    edgeIterator();
    edgeIterator(node*);
    edgeIterator operator++(int);
    Type operator*();
    bool operator==(const edgeIterator&);
```

```
    bool operator!=(const edgeIterator&);
private:
    node * current;
};

  vertex();
  vertex(const vertex<Type>&);
  const vertex& operator=(const vertex<Type>&);
  ~vertex();
  edgeIterator begin();
  edgeIterator end();
  void addEdge(const Type&);
private:
  node * neighbors;
};
```

Each member of edgeIterator class will contain/perform the following

- node * current - stores the address of a node object where the edgeIterator object points to
- vertex<Type>::edgeIterator::edgeIterator() - default constructor that sets current to NULL
- vertex<Type>::edgeIterator::edgeIterator(vertex<Type>::node * edge) - a constructor that takes in a node object which gets assigned to current
- typename vertex<Type>::edgeIterator vertex<Type>::edgeIterator::operator++(int) - an operator function that sets the iterator to point to the next node object, you will need to set current to point to the next node
- Type vertex<Type>::edgeIterator::operator*() - an operator that dereferences the iterator, returns the item field of the node that current points to
- bool vertex<Type>::edgeIterator::operator==(const vertex<Type>::edgeIterator& rhs) - compares the address of the iterator on the left side with the iterator on the right side, returns true if they both point to the same node, and returns false otherwise
- bool vertex<Type>::edgeIterator::operator!=(const vertex<Type>::edgeIterator& rhs) - compares the address of the iterator on the left side with the iterator on the right side, returns false if they both point to the same node, and returns true otherwise

Each member of vertex class will contain/perform the following

- struct node - needed for the adjacency list
- node * neighbors - the head of the linked list, the linked list stores all the neighbors of the vertex
- vertex<Type>::vertex() - default constructor that sets neighbors to NULL
- vertex<Type>::vertex(const vertex<Type>& copy) - a copy constructor that deep copies the neighbor list of the object passed into the constructor to the object that calls the constructor
- const vertex<Type>& vertex<Type>::operator=(const vertex<Type>& rhs) - assignment operator, that performs a deep copy of the right side object with the left side object (the object that calls the operator function)
- vertex<Type>::~vertex() - destructor, deallocates all the nodes in its neighbor list
- typename vertex<Type>::edgeIterator vertex<Type>::begin() - returns a edgeIterator object whose current will be the head of the neighbor list for the vertex object
- typename vertex<Type>::edgeIterator vertex<Type>::end() - returns a edgeIterator object whose current will be assigned to NULL

- `void vertex<Type>::addEdge(const Type& edge)` - adds a new node into the neighbor list (a head insert would be the best way to implement this)

Stack Class

You have the option to use `myStack` from an earlier assignment, or you can use STL stack as well, but you will have a slight reduction to your assignment grade for using STL stack.

Hash Map Class

You will use a hash to constructor your adjacency list since the nodes are not labeled with indices but rather names (in a string form), thus a hash map is the perfect structure to use here. Once again, you can use STL `unordered_map` here, but if your program works with `hashMap`, you will be awarded extra credit points.

Contents of main

The input file given to you consist of the following: the first line contains a source street name and a destination street name (in which you want to find all 2 way paths between these two streets/nodes), then each line will contain a single edge (a from and to street name separated by a space). Once you have a filstream ready to read, you will need to populate an adjacency list, you will declare one of the following data structures

- `hashMap<string, vertex<string> > adjList;`
- `unordered_map<string, vertex<string> > adjList;`

So each street name maps to its list of neighbors. If I want to traverse all of `"Main"` street neighbors, I would write the following

```
vertex<string>::edgeIterator it;
vertex<string>::edgeIterator nil;

nil = adjList["Main"].end();

for (it = adjList["Main"].begin(); it != nil; it++)
  cout << *it << endl;
```

If I want to add an edge into the adjacency list that goes from `"Main"` street to `"Sahara"`, I would write `adjList["Main"].addEdge("Sahara");` In order to output each path, you would need a DFS type traversal, here is the function prototype that I used to construct the paths

```
bool getPaths(string current, string finalDestination,
      hashMap< string, vertex<string> > graph,
      hashMap<string, bool>& nodesInPath,
      myStack<string>& recStack, int& pathNo);
```

Content of the function:

- `string current` - the node name (street name) we are currently processing
- `string finalDestination` - the node node (street name) that were are trying to find a path consisting of two way streets
- `hashMap< string, vertex<string> > graph` - the adjacency list
- `hashMap<string, bool>& nodesInPath` - maps a node to a boolean value, denotes if which nodes are in our current path

- `myStack<string>& recStack` - stores the order of nodes in our current path
- `int& pathNo` - denotes the amount of paths from the start node to the end node that has been found so far
- Here is a description of the function
  - You perform a DFS-like traversal, but a node could be revisited multiple times
  - As you traverse nodes (go to a node that is not in our current path), we want to update nodesInPath and recStack accordingly (as we enter and leave any particular node)
  - Once a 2 way path to finalDestination is found, you output the nodes in the path (using the recStack), may need to update pathNo as well
  - Returns `true` if at least one path was found, so in main you know whether to output if no paths were found or not, but it seems like pathNo could also tell you if no paths were found as well
  - Returns `false` if no paths were found

Specifications

- Comment your code and your functions
- Do not add extra class members or remove class members and do not modify the member functions of the class
- No global variables (global constants are ok)
- Make sure your program is memory leak free
- If you use STL stack, you will receive a deduction on this assignment
- Extra credit will be awarded if you use `hashMap` instead of `unordered_map` for the adjacency list

Sample Run

```
$ g++ main.cpp
$ ./a.out

Last one ): graph01.txt

Path 1 : First <=> North <=> Rainbow <=> Last
Path 2 : First <=> Main <=> North <=> Rainbow <=> Last
Path 3 : First <=> Elm <=> Rainbow <=> Last

$ ./a.out

Last one ): A9_1.txt

Path 1 : First <=> Main <=> North <=> Rainbow <=> Last
Path 2 : First <=> North <=> Rainbow <=> Last
Path 3 : First <=> Elm <=> Rainbow <=> Last
```

Submission

Compress your source files and upload to class website by deadline

References

- Link to the top image can be found at *https* : *//www.pngkit.com/bigpic/u2t4i1a9u2i1t4r5/*