Lab Exercise 5
PathFinder

You want to develop a software to be used by city bus/metro rail commuters. Consider the metro rail network of Delhi. You may have a look at the latest Delhi metro route map 2023.
The purpose of the software is to help a commuter to work out a path from his/her current stop (metro station) to a destination stop.

The Delhi Metro Rail Corporation (DMRC) is willing to provide you with all the data related to metro routes and stations on each route. However, what is made available to you from DMRC is a set of MetroLine objects (each such object is a doubly linked list), one for each metro line/track (an object of class ***MetroLine***). The linked list object carries a field which gives the name of the line and carries pointers to the sentinel nodes. Each node (object of class ***MetroStop***) in the doubly linked list carries the name of the metro station, and a pointer to its ***MetroLine***, along with pointers to the next and the previous nodes. There is no information given to you regarding which node (stop) is a junction of more than one metro line/track.

Given this collection of linked lists, you need to create a class called PathFinder.
This class will have methods for several tasks.

Task 1: Create an AVL tree of all the stations.

The first task is to find what all nodes are junctions. You instantiate an ***AVL tree*** object. You start traversing each linked list (the ***MetroLine*** object) and keep inserting the metro station (stop) names on that linked list into the AVL tree. The key to be inserted into the AVL tree is the name of the metro stop.

The AVL tree nodes are objects of class ***AVLNode***. The keys for the AVL tree nodes are the metro stop names, hence we use the lexicographic ordering of the stop name to order the keys. If a stop is a junction, its name will get inserted more than once into the AVL Tree. When a junction name is inserted for the first time, a new AVLNode is instantiated and inserted into the tree. But when the name of the junction is inserted for the the second time (or more times) the ***AVLNode*** object (with that key, i.e., junction name) will be already there in the AVL tree. A search leading to an existing ***AVLNode*** with that key would imply that the ***AVLNode*** is a junction.

The ***AVLNode*** refers to a metro stop and needs to keep a list of pointers to the corresponding node (object of class ***MetroStop***) in the linked list of ***MetroLine***. In case an ***AVLNode*** is a junction, there will be multiple metro lines passing through it. So, the ***AVLNode*** will keep a list of pointers to the distinct nodes (objects of ***MetroStop***) referring to to the same stop (junction). The distinct nodes are from the linked lists (of class ***MetroLine***) of the lines that pass through the junction.

- Note that even though a junction will be a common stop in multiple metro lines, each linked list object (of class ***MetroLine***) will have distinct node objects (***MetroStops***) all bearing the name of the same metro stop which is a junction.
- So the AVL tree node, if it is a junction, needs to keep a list of pointers to the separate (line/track linked-list) node objects. After the AVL tree has been constructed, it is possible to examine a node and identify if it is a junction, and if yes, which are the lines passing through the junction.

Task 2: Search for the path:

The second task is to work out a path between a given origin stop to a destination stop. We make a simplifying assumption that the metro track does not have cycles, that means, it is not possible to plan a trip which reaches back to the origin and visits every intervening stop exactly once.

To work out the path, we shall work with objects of class ***Trip***. As we begin searching from an origin stop, we shall instantiate ***Trip*** objects. For a given line/track, there will be two ***trip*** objects that can be instantiated, one for the forward direction towards the head station of the linked list (***MetroLine***) and one for the backward direction towards the tail station of the linked list (***MetroLine***). A trip object marks the initiation of search along the line/track in a particular direction. A trip object also carries a pointer to a previous trip object that instantiated it. If the search finds the destination, we are done and we report the trip objects involved in the search. If a search visits a node which is a junction (the AVL tree would tell us if it a stop is junction), then we shall instantiate two new ***trip*** objects for each track/line that crosses the junction. So, if a junction has 3 tracks, there will be 6 new trip objects instantiated (one for each direction). As a new trip object is instantiated, it is placed in an object called ***Exploration***. The ***Exploration*** object is a queue which stores the unexplored trips.

Mechanism of Search:

Search along a ***trip*** requires traversing the linked list (***MetroLine***) starting from the junction in the specified direction. If the search along a trip reaches the destination stop, we back-traverse along the trips and print the sequence of intermediate stations and the line/track names. Your program should also print the total fare to be paid for the route. If a trip is unsuccessful in finding the destination, then the PathFinder will dequeue and explore the next ***trip*** object in the ***Exploration*** queue. Make check to avoid the same trip getting explored again and again. If the ***Exploration*** queue becomes empty, it means that the destination stop is not reachable from the origin stop. To summarise, exploring a ***trip*** object will dynamically create more trip objects when junctions are encountered. While exploring along a trip, you need to query the ***AVLTree*** for every station to check if it is a junction or not. Note that the information about whether a particular station is a junction or not is available only with the ***AVLNode*** object.

Return search result as a ***Path*** object:

The PathFinder class will have a member function called ***findPath*** ("origin stop", "destination stop") which returns a pointer to an object of class ***Path***, if a path is found, or returns NULL in case there is no path found. This object will have 3 data members — a vector of strings for the names of the intermediate stations, another vector of strings with the corresponding metro lines and an integer field to store the total fare.

Computing the Fare

The input file will give you the fare applicable for a single metro line. The fare against each station is the fare from the origin stop of the metro line to a given station. Fare for a segment (one intermediate stop A to another intermediate stop B, assuming that B comes after A) can be computed as absolute value of the difference of the fare for stop B and stop A.
If a trip involves multiple metro lines, you can assume the total fare is computed as the sum of the fare of the intermediate segments. You are free to design what data fields you would like to store in the classes to compute the fare.

Caution: Note that this assignment requires you to use linked lists, queue, and an AVLTree. You must use these data structures the way they are supposed to be used. You can use the implements of linked lists and queue that you might have done in previous exercises. However, the AVLTree needs to be implemented and it should involve restructuring steps to restore the height balance property after every insertion. The test cases will check the height balance property of your AVLTree, which is the primary focus of this lab exercise.

Advisory: Like the previous lab exercises, you will be getting the class templates and the test cases. If you find any inconsistencies in the templates or the test cases, please write back to the instructor. Kindly have patience and be *agile* to incorporate changes made to the class templates in response to your concerns. Start the exercise as soon as possible so that the last minute hassle is not there. Do not expect the deadline to get extended.

Sample Input:
There are going to be multiple input files, each corresponding to a different track. More such input files and class templates will be provided. The last integer in every line is the price of the ticket (from origin). Every character which precedes the last integer is the name of the stop.

File Name: Yellow.txt

```
Samaypur Badli		0
Rohini Sector 18, 19	4
Haiderpur Badli Mor	8
Jahangirpuri	12
Adarsh Nagar	16
Azadpur		20
Model Town	24
GTB Nagar	28
Vishwa Vidyalaya	32
Vidhan Sabha	36
Civil Lines	40
Kashmere Gate		44
Chandni Chowk		48
Chawri Bazar	52
New Delhi	56
Rajiv Chowk	60
Patel Chowk	64
Central Secretariat	68
Udyog Bhawan		72
Lok Kalyan Marg		76
Jor Bagh	80
Dilli Haat - INA	84
AIIMS	88
Green Park	92
Hauz Khas	96
Malviya Nagar	100
Saket	104
Qutab Minar	108
Chhatarpur	112
Sultanpur	116
Ghitorni	120
Arjan Garh	124
Guru Dronacharya	128
Sikanderpur	132
MG Road		136
IFFCO Chowk	140
HUDA City Centre	144
Sector 45	148
Cyber Park	152
Sector 46	156
Sector 47	160
```