Transit – 100 course points

The purpose of this assignment is to practice your understanding of linked structures.

Start your assignment early! This assignment is substantially harder than the previous assignment. You need time to understand the assignment and to answer the many questions that will arise as you read the description and the code provided.

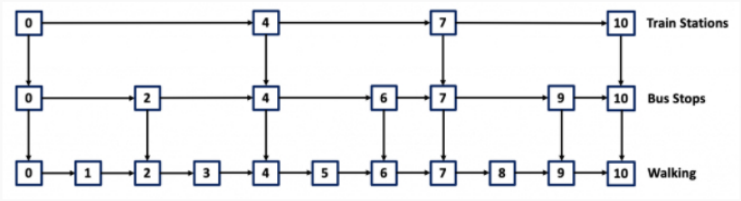
Refer to our Programming Assignments FAQ for instructions on how to install VSCode, how to use the command line and how to submit your assignments.

Overview

You've just moved into a new city, and you're planning your daily commute. You have 3 modes of public transportation available to you: the subway, the bus, and of course you can just walk. You can imagine locations in the city as a number line which increments from your starting location of 0 (zero). You can only move forward along this number line in your journey. The train is much faster than the bus, and the bus is much faster than walking. Unfortunately, the train can only stop at points where a train station exists, and the bus can only stop at points where a bus stop exists. You may assume that a bus stop exists at every train station, and that you may begin walking from any bus stop. You may walk forward and end your walking journey at any point.

The Linked Structure

You plan and visualize your journey using a special 3 layered linked list. The top layer represents the train, the middle layer represents the bus, and the bottom layer represents walking. Each node in your layered linked list contains an **int** representing the location, a reference **next** representing the next location you can visit in your current mode of transport, and a reference **down** which takes you down to the SAME location in a slower mode of transport. For example in the diagram below, the node 4 in the train layer points to the node 4 in the bus layer, which points to the node 4 in the walking layer. **next** will be set to null if there are no more locations to visit on the current mode of transport, and **down** will be set to null if you are in the walking layer.



The image above helps us visualize our journey using a special 3 layer linked list. The top layer represents the train, the middle layer represents the bus, and the bottom layer represents walking.

Each node in the 3 layer linked list contains an int representing the location, a reference **next** representing the next train station, bus stop, or city location based on whether you're on the train, bus, or walking level, and a reference **down** representing the next slower mode of transport. For the walking layer, **down** will be set to **null** since walking is the slowest mode of transport available. For the train layer, **down** will point to the node in the bus layer which contains the same point in the city, and similarly for the bus layer to the walking layer. When there are no more train stations, bus stops, or city locations, **next** will be set to **null**.

Implementation

Overview of files

- **TNode** class, which houses an **int** for location, and **next** and **down** pointers. **Do not edit this class**. It is not submitted, but it is used exactly as written to grade your code.
- **Driver** class, which you can run to test any of your methods interactively. Feel free to edit this class, as it is provided only to help you test. It is not submitted and it is not used to grade your code.
- **StdIn** and **StdOut**, which are used by the driver. **Do not edit these classes.**.
- **Transit** class, which contains annotated method signatures for all the methods you are expected to fill in. You should write your solutions in this file, and it is the file which will be submitted for grading.
- Multiple text files which contain input data, and can be read by the driver as test cases. Feel free to edit them or even make new ones to help test your code. They are not submitted.

Transit.java

NOTE: You are allowed (encouraged, even) to make helper methods in your Transit.java file to be used in your graded methods. Just make sure that they are created with the private keyword. Do not add new imports.

Methods to be implemented by you:

1. makeList

Implement this method to build your layered linked list structure. It takes as arguments 3 sorted integer arrays representing train stations, bus stops, and walking locations. It is guaranteed that:

- the bus array contains all values in the train array.
- the walking array contains all values in the bus array.
- the walking array contains every integer from 1 to its largest value.

Every layer of your list has to start with a node at location 0 (zero), and by following the next pointers, it visits each value in the corresponding array. **Note that 0 (zero) will NOT be present in the input arrays, and you are responsible for adding zero nodes for each layer yourself.**

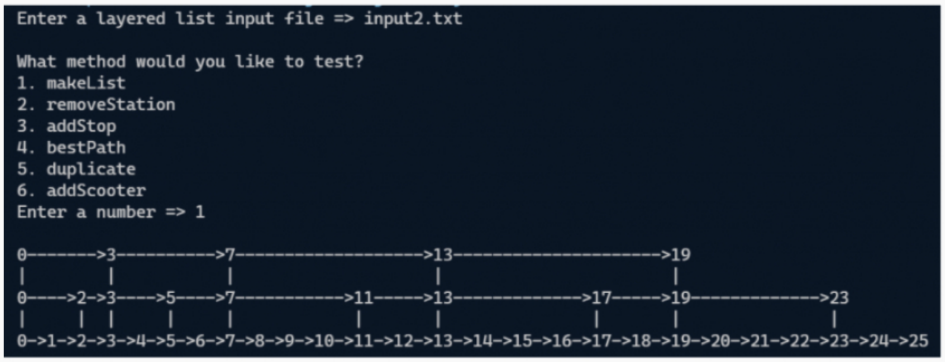
that 0 (zero) will NOT be present in the input arrays, and you are responsible for adding zero nodes for each layer yourself.

Additionally, the nodes in the train layer have to point down to the node in the bus layer of the same location, and the nodes in the bus layer should point down to the nodes in the walking layer of the same location. **Return the zero node of the train layer** (the topmost layer).

You have been provided some input files to test this method (input1.txt, input2.txt, input3.txt). The format is as follows:

- One line containing the number of train stations
- One line containing the location of each train station, space separated
- One line containing the number of bus stops
- One line containing the location of each bus stop, space separated
- One line containing the number of walking locations
- One line containing each walking location, space separated

Below is an example of running the driver to help test this method.

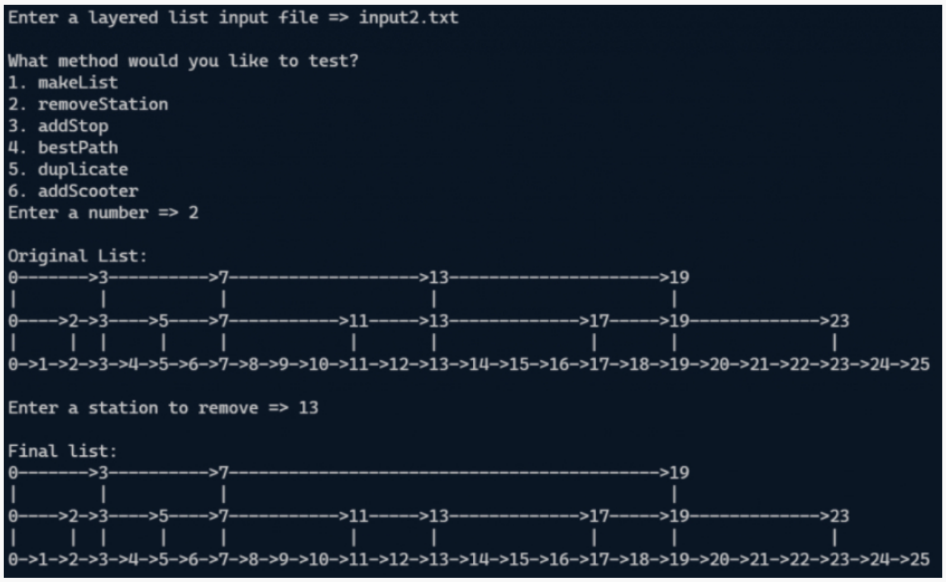


2. removeTrainStation

Given your layered linked list (reference to the train zero node) and an **int** representing the location of a train station, write a method to remove the given train station from the layered linked list. Note that the corresponding bus stop and walking location are NOT to be removed. **Also note that this is an in place method, and you should perform your operations on the layered list without returning anything**. If the specified train station doesn't exist, do nothing. The specified train stations used for grading are guaranteed to not be the zero node.

The input files to test this method are the same as makeList.

Below is an example of running the driver to help test this method.

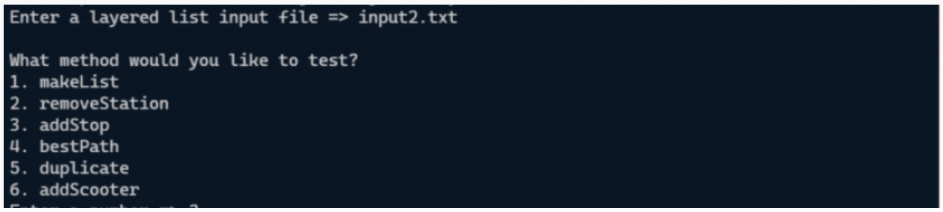


3. addBusStop

Given your layered linked list (reference to the train zero node) and an **int** representing the location of a new bus stop, write a method to add the new bus stop to the layered linked list. The bus stops used for grading are guaranteed to correspond to a walking layer node. If the specified bus stop already exists, do nothing. **This is an in place method, and you should perform your operations on the layered list without returning anything.**

The input files to test this method are the same as makeList.

Below is an example of running the driver to help test this method.



```
Enter a number => 3

Original List:
0------->3---------->7------------------->13--------------------->19
|        |          |                     |                      |
0---->2->3---->5---->7---------->11----->13------------->17----->19------------->23
|     |  |     |    |           |        |               |       |               |
0->1->2->3->4->5->6->7->8->9->10->11->12->13->14->15->16->17->18->19->20->21->22->23->24->25

Enter a bus stop to add => 15

Final list:
0------->3---------->7------------------->13--------------------->19
|        |          |                     |                      |
0---->2->3---->5---->7---------->11----->13----->15----->17----->19------------->23
|     |  |     |    |           |        |       |       |       |               |
0->1->2->3->4->5->6->7->8->9->10->11->12->13->14->15->16->17->18->19->20->21->22->23->24->25
```

4. bestPath

Given your layered linked list and an **int** representing the destination location, determine the optimal path to get there. Remember that you can only move forward or down through your layered list, and that you don't want to overshoot your destination in either the train or bus layers. You want to stay on the train as long as you possibly can, then you want to stay on the bus as long as you possibly can, before walking to your destination. **Add every TNode along the best path to an ArrayList, ending in the destination node in the walking layer, returning this arraylist.**

The input files to test this method are the same as makeList

Below is an example of running the driver to help test this method:

```
Enter a layered list input file => input2.txt

What method would you like to test?
1. makeList
2. removeStation
3. addStop
4. bestPath
5. duplicate
6. addScooter
Enter a number => 4

Layered Linked List:
0------->3---------->7------------------->13--------------------->19
|        |          |                     |                      |
0---->2->3---->5---->7---------->11----->13------------->17----->19------------->23
|     |  |     |    |           |        |               |       |               |
0->1->2->3->4->5->6->7->8->9->10->11->12->13->14->15->16->17->18->19->20->21->22->23->24->25

Enter a destination => 11

Best path:
0>>>>>>>>3>>>>>>>>>>7
                    V
                    7>>>>>>>>>>>11
                                V
                                11
```

5. duplicate

Given your layered linked list, write a method to return a **deep copy**. This means that the structure of the new list should be exactly the same, with the same values and connections in the train, bus, and walking layers. It should be able to function exactly as the original list does. However, there should be NO nodes present in the copy which also exist in the original. In order for your deep copy to be graded as correct, every single node in it must have been created using the **new** keyword. **Return the zero node in the train layer of the deep copy**

The input files to test this method are the same as makeList.

Below is an example of running the driver to help test this method:

```
Enter a layered list input file => input2.txt

What method would you like to test?
1. makeList
2. removeStation
3. addStop
4. bestPath
5. duplicate
6. addScooter
Enter a number => 5

Original list:
0------->3---------->7------------------->13--------------------->19
|        |          |                     |                      |
0---->2->3---->5---->7---------->11----->13------------->17----->19------------->23
|     |  |     |    |           |        |               |       |               |
0->1->2->3->4->5->6->7->8->9->10->11->12->13->14->15->16->17->18->19->20->21->22->23->24->25

Duplicate:
0------->3---------->7------------------->13--------------------->19
|        |          |                     |                      |
0---->2->3---->5---->7---------->11----->13------------->17----->19------------->23
|     |  |     |    |           |        |               |       |               |
```

```
0->1->2->3->4->5->6->7->8->9->10->11->12->13->14->15->16->17->18->19->20->21->22->23->24->25
```

6. addScooter

Your city has FINALLY added those new electric scooters (you may have even seen them around the Rutgers campus)! You can't wait to try them, and you decide to try to add them to your daily commute. You notice that the scooters can be picked up at any bus station and can be dropped off in slightly more locations, though they still can't get you to as many places as walking. Naturally, you decide to implement this change to your commute as a "scooter layer" **underneath the bus layer but above the walking layer**.

You must write a method which takes in your layered linked list and a sorted **int** array representing the locations in the scooter layer. It should then update your layered linked list so that the bus layer points down to the correct nodes in the new scooter layer, and the new scooter layer points down to the correct nodes in the walking layer. **This is an in place method, and you should perform your operations on the layered list without returning anything.**

You have been provided some input files to help test this method (scooter2.txt, scooter3.txt). The scooter2 file should only be used with input2.txt, and the scooter3 file should only be used with input3.txt. The format is as follows:

- One line containing the number of scooter stops
- One line containing all the locations of scooter stops, space separated

Below is an example of running the driver to help test this method:

```
Enter a layered list input file => input2.txt

What method would you like to test?
1. makeList
2. removeStation
3. addStop
4. bestPath
5. duplicate
6. addScooter
Enter a number => 6

Original list:
0------->3---------->7------------------->13--------------------->19
|        |          |                     |                      |
0---->2->3---->5---->7----------->11----->13------------->17----->19------------->23
|     |  |     |    |            |       |               |      |               |
0->1->2->3->4->5->6->7->8->9->10->11->12->13->14->15->16->17->18->19->20->21->22->23->24->25

Enter a scooter layer input file => scooter2.txt

Final list:
0------->3---------->7------------------->13--------------------->19
|        |          |                     |                      |
0---->2->3---->5---->7----------->11----->13------------->17----->19------------->23
|     |  |     |    |            |       |               |      |               |
0->1->2->3---->5---->7---->9----->11----->13----->15----->17----->19----->21----->23
|  |  |  |     |    |     |      |       |       |       |       |       |       |
0->1->2->3->4->5->6->7->8->9->10->11->12->13->14->15->16->17->18->19->20->21->22->23->24->25
```

Implementation Notes

- YOU MAY only update the methods makeList(), removeTrainStation(), addBusStop(), addScooter(), bestPath(), and duplicate().
- DO NOT add any instance variables to the Transit class.
- DO NOT add any public methods to the Transit class.
- YOU MAY add private methods to the Transit class.

VSCode Extensions

You can install VSCode extension packs for Java. Take a look at this tutorial. We suggest:

- Extension Pack for Java
- Project Manager for Java
- Debugger for Java

Importing VSCode Project

1. Download transit.zip from Autolab Attachments.
2. Unzip the file by double clicking.
3. Open VSCode
    - Import the folder to a workspace through **File** > **Add Folder to Workspace**

Executing and Debugging

- You can run your program through VSCode or you can use the Terminal to compile and execute. We suggest running through VSCode because it will give you the option to debug.
- How to debug your code
- If you choose the Terminal:
    - to compile: once inside the transit directory, type: **javac** ***.java**
    - to execute: **java Driver**

Before submission

Collaboration policy. Read our collaboration policy here.

Submitting the assignment. Submit *Transit.java* separately via the web submission system called Autolab. To do this, click the *Assignments* link from the course website; click the *Submit* link for that assignment.

Getting help

Getting help

If anything is unclear, don't hesitate to drop by office hours or post a question on Piazza. Find instructors office hours by clicking the *Staff* link from the course website. In addition to office hours we have the CAVE (Collaborative Academic Versatile Environment), a community space staffed with lab assistants which are undergraduate students further along the CS major to answer questions.

Problem by Ishaan Ivaturi