EECS 214/395 Data Structures Fall 2014 Program 3

Posted: Saturday November 28th
Due: 11:59pm Tuesday December 5th
% of Final Grade: 7.5%

With only one week left in the course, this assignment is straightforward to the extent that it repeats a homework problem. It emphasizes our ability to trade space for time (a trade which is usually desirable) and really cut down big-O running time with correct design. It asks students to implement parts of a graph class, for two of the three graph designs from Problem 2 of Homework 5. An O(1) Transpose operation is already given **for the matrix version only.**

Your Tasks

- Download the python file template "Pboth_Project3_acj123.py", which is attached to this assignment.

- Test code appears within a "__main__" block at the bottom; there are also some print statements that appear within the code that you should not remove; these are required to facilitate grading.

- Where my netID appears, change it to be yours.

- When you submit your file, it should be named as "P##_Project3_[netid]" where ## replaces "both" in the template, and is either 27 or 34 *to indicate which version of Python you are using*.

- All methods in each class need to work together to maintain implementation of all fields and other methods in the classes (respect the invariants implied by the existence of fields / pointers).

- For both representations assigned, assume that the nodes are numbered as integers 0,1,…, n-1 ("computer counting").

ADJACENCY MATRIX REPRESENTATION

For this class, assume the following. When the graph object is created, it takes the number of nodes as input to the constructor and creates the matrix (with no edges). Assume that nodes are accessible by assigned index numbers. These nodes can possibly be removed; and possibly later re-added *with its edges missing/forgotten*; but no brand new nodes will ever be added. (So the original number of nodes is an upper bound forever; this number is stored within the class as part of the constructor block.) A missing edge should be represented by 0, a present edge by 1; a removed node should have all outgoing and incoming edges represented by a negative number.

In the matrix, an edge (a,b) is represented by a 1 *at row a and column b* (when the matrix is not transposed).

ASSUME THAT THE GRAPH CAN HAVE "SELF-EDGES" (i.e., edges coming out of node x, pointing to node x).

- Write the Insert_Matrix_Edge(a,b) method in Matrix_Graph_acj123 (TREE) class. It should add an edge to the graph object from a to b. You may assume nodes with indexes a and b are currently present in the graph, and that edge (a,b) does not already exist. It should run in O(1); it does not have a return value. (10 points)

- Write the Check_Matrix_Edge(a,b) method in Matrix_Graph_acj123 (TREE) class. It should check if there is an edge in the graph object from a to b. It should return a Bolean: True if entry = 1, or False if entry < 1. It should run in O(1). (10 points)

- Write the Remove_Matrix_Edge(a,b) method in Matrix_Graph_acj123 (TREE) class. It should remove an edge in the graph object from a to b. If the edge or node does not exist, it should have no effect. It should run in O(1); it does not have a return value. (10 points)

- Write the Remove_Matrix_Node(a) method in Matrix_Graph_acj123 (TREE) class. It should remove a node in the graph object. You may assume that we will never try to delete a node that is already deleted. It should run in O(n); it does not have a return value. (10 points)

- Write the ReAdd_Edgeless_Matrix_Node(a) method in Matrix_Graph_acj123 (TREE) class. It should re-add a node in the graph object. You may assume that we will never try to "re-add" a node that is not deleted. It should run in O(n); it does not have a return value. (10 points)

ADJACENCY LISTS REPRESENTATION

For this class, assume the following. When the graph is created, it does not take the number of nodes as input. It generates two empty lists using [ ] syntax in Python (sitting in the graph class). These lists have specific names 'Neighbors' and 'Incoming' as we have used in lecture.

A List class and List_Node class are given. They are augmented versions of these classes from Program 2. All elements of the array should be Linked_List objects if the node has not been removed (even if they have no elements). If the node has been removed, the value in the array should be None.

Linked_Lists are now doubly-linked, as List_Nodes have pointers to both Next and Prev; and also to "Other." As described in class, this Other pointer should identify the same edge in the other structure used for edges (and allow fast removal time if necessary).

The number of nodes is arbitrary. You can use [list_name].append to create a new node. If a node is removed, it is gone forever. So eventually the list might have nodes 0,1,2,6,7,9, and if it adds a new node, the new node is node 10 (with lists of neighbors at index 10).

These methods should use the less-than-perfect method of pointer manipulation inside the Linked_List and List_Node classes for things like insert and remove. After any given operation, Head and Tail pointers inside Linked List objects, and Next, Prev, Other inside List_Node objects should all be in place.

- Write the Insert_Lists_Node(a) method in Matrix_Graph_acj123 (TREE) class. It should add a new node to the graph without edges. The new node should have a brand new index. It should run in O(1); it does not have a return value. (10 points)

- Write the Insert_Lists_Edge(a,b) method in Matrix_Graph_acj123 (TREE) class. It should add an edge to the graph object from a to b. You may assume nodes with indexes a and b are currently present in the graph, and that edge (a,b) does not already exist. It should run in O(1); it does not have a return value. (10 points)

- Write the Check_Lists_Edge(a,b) method in Matrix_Graph_acj123 (TREE) class. It should check if there is an edge in the graph object from a to b. It should return a Bolean: True if edge exists, or False if edge does not exist. It should run in O(out_degree(a)). (10 points)

- Write the Remove_Lists_Edge(a,b) method in Matrix_Graph_acj123 (TREE) class. It should remove an edge in the graph object from a to b. If the edge or node does not exist, it should have no effect. It should run in O(out_degree(a)); it does not have a return value. (10 points)

- Write the Remove_Lists_Node(a) method in Matrix_Graph_acj123 (TREE) class. It should remove a node in the graph object forever. You may assume that we will never try to delete a node that is already deleted. The values in the arrays at index a should be None. Also be sure erase the "second-copies" of edges into and out of node a. It should run in O(out(degree(a) + in(degree_b)); it does not have a return value. (10 points)

- **CODING RESTRICTIONS.** Part of the objective of this assignment is to build fundamental data structures, not to use language built-ins. **Except as specified for [ ] within the assignment, and as given within the template;** you may not use the built-in functions list, tuple, dict or similar; you may not use the built-in string functions split, or similar **but you may use append where it is needed**. You may not use the built in list or dict constructors as [ ] or { }. In fact, you should not use characters [ ] or { } anywhere. Submissions might be uniformly searched by program or manual check for these prohibited uses of Python. If you're not sure if you can use something or not, please ask. See the course syllabus for further explanation on restrictions. **To repeat, you can use [ ] and typical Python functions on lists like 'append' for example.**

- You can check your answer by running the test case given in __main__ block against the output below.

- Submit your project solutions via Blackboard.

Method-Writing "Check List"
Always, when you write methods, be mindful of:

- Invariants; if you are changing the data structure, be sure to fix up its pointers and fields so that future methods can be called correctly
- Boundary conditions; are you removing something from an empty data structure, or do you need more memory?
- Pointer order of operations; if you have to give new assignments to two pointers, does the order in which you do it matter? One might be your only pointer to a part of your data structure.

Python Tips
Formatting. Remember, Python does not requires brackets as { }, or end-of-line signifiers as ;. Instead, it uses strict indenting to delineate blocks within classes, methods, conditionals and loops. By default, lines end where the text ends; this can be subverted by enclosing a multiple-line expression in parentheses ( ) such that it is "open" when the first line ends.

Object Types. You should not expect any problems with Types in this project, but you should be aware of them. Python is an "un-typed" language. This means that you do not have to declare the types of variables before assigning them, and you will not see Type declarations in the template. This has both benefits and drawbacks. The benefit is that a variable's type can be dynamically changed by simply assigning it to anything else as needed. (We used this in Program 1 when we let the value of a List_Node be either a string or a pointer to an embedded list!) The drawback is that if you do this by accident, Python will not throw you an error as a red flag. Casting is sometimes necessary (usually between string and "number" types).

Classes. You do not need to design any classes from scratch for this assignment but do need to know how to navigate them, and how to implement methods. When the class is declared, inheritance is designated in parentheses immediately following the class' name. Constructors are located within "__init__" methods. All class methods, when they are declared, are required to explicitly identify a pointer back to the current object in the first place of its input list. By convention this variable is named "self". When class methods are called, they do not pass a value into "self". So if a method is declared as MethodA(self, v1), it is called with just MethodA(v2_into_v1). Within methods, class field FieldB should then be accessed using self.FieldB. For examples of these, see the template.

Testing

The template comes with test code provided within the __main__ function. This is to help you test your code, but it will not be the input used for final grading. It might not even be all of the functions used for final grading. Just because your output matches does not guarantee that your code is flawless.

(This is just a sample of tests you might run. I repeat: I cannot guarantee that because your code reproduces the given test output, that it does not have other bugs.)

Please remember, do not change any of the print statements or __str__ methods (which tell objects how to "print" themselves") within classes. Then it should result in the following output to the screen.

```
MATRIX REPRESENTATION
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]

[0, 1, 0, 0, 0]
[0, 0, 0, 0, 1]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]

[0, 1, 0, 0, 0]
[0, 0, 0, 0, 1]
[1, 0, 0, 0, 0]
[0, 0, 1, 0, 0]
[0, 0, 0, 0, 0]

True
False
[-2, -1, -1, -1, -1]
[-1, 0, 0, 0, 1]
[-1, 0, 0, 0, 0]
[-1, 0, 1, 0, 0]
```

```
[-1, 0, 0, 0, 0]

[-2, -2, -1, -1, -1]
[-2, -2, -1, -1, -1]
[-1, -1, 0, 0, 0]
[-1, -1, 1, 0, 0]
[-1, -1, 0, 0, 0]

[0, -1, 0, 0, 0]
[-1, -2, -1, -1, -1]
[0, -1, 0, 0, 0]
[0, -1, 1, 0, 0]
[0, -1, 0, 0, 0]



LISTS REPRESENTATION
Neighbors
None
None
None
None
None
Incoming
None
None
None
None
None

Neighbors
1
None
4
2
None
Incoming
None
0
3
None
2

True
False
Neighbors
```

```
1
None
None
2
None
Incoming
None
0
3
None
None

Neighbors
1 2 4
None
None
2
None
0
Incoming
5
0
3 0
None
0
None

Neighbors
None
None
None
2
None
None
Incoming
None
None
3
None
None
None
```