CMPSC-132: Programming and Computation II
Fall 2019

Homework 5

Due Date: 12/13/2019, 11:59PM
100 pts

Instructions:

- The work in this lab must be completed alone and must be your own.
- **Download the starter code file from the HW5 Assignment on Canvas. Do not change the function names on your script.**
- A doctest is provided as an example of code functionality. Getting the same result as the doctest does not guarantee full credit. You are responsible for debugging and testing your code with enough data, you can share ideas and testing code during your recitation class.
- Each function must return the output (Do not use print in your final submission, otherwise your submissions will receive a -10 point deduction)
- **Do not include test code outside any function in the upload. Printing unwanted or ill-formatted data to output will cause the test cases to fail. Remove all your testing code before uploading your file (You can also remove the doctest). Do not include the input() function in your submission.**
- You can share testing cases on Piazza!!!!

Goal:

In Module 6, we discussed the Graph data structure. Since the graph is a non-linear structure, there is no unique traversal. Nodes can be found using Breadth-First Search (visit the sibling nodes before visiting the child nodes) and Depth-first search (visit the child nodes before visiting the sibling nodes). Based on the implementation of the Graph data structure discussed during our lecture (provided in the starter code):

1. Create the method *bfs*(*start*). This method takes the key of the starting node and performs Breadth-first search in an instance of the class Graph. This method must **return** a list that contains the order in which the nodes where accessed during the search (following alphabetical order when discovering nodes). You **must** use your queue code from LAB 8 in order to produce your result. If you don't use the queue, your will not receive credit for the assignment
2. Create the method *dfs*(*start*). This method takes the key of the starting node and performs Depth-first search in an instance of the class Graph. This method must **return** a list that contains the order in which the nodes where accessed during the search (following alphabetical order when discovering nodes). You **must** use your stack code from HW 3 in order to produce your result. If you don't use the stack, your will not receive credit for the assignment

Grading Notes:

- A random instance of the Graph class (directed or undirected, weighted or unweighted) will be created and the *bfs* and *dfs* methods will be called on 4 different starting nodes for 12.5 pts each. Make sure you use the Graph data structure provided in the starter code, otherwise, no credit will be given.

- Vertices and edges will be provided in random order (non-alphabetical order). The final result should be only the one provided by following alphabetical order.
- **Vertex name can contain more than 1 letter**

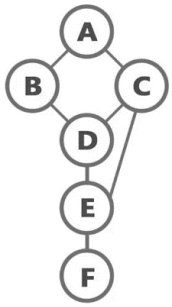
EXAMPLE:
Note that this is only an example, the fact that you code produces the example's output does not ensure your code works properly. Test your code with several examples!

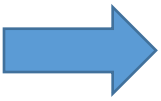
```
>>>  g1 = {'A': ['B','D','G'],
     'B': ['A','E','F'],
     'C': ['F'],
     'D': ['A','F'],
     'E': ['B','G'],
     'F': ['B','C','D'],
     'G': ['A','E']}
>>> g=Graph(g1)
>>> g.bfs('A')
['A', 'B', 'D', 'G', 'E', 'F', 'C']
>>> g.dfs('A')
['A', 'B', 'E', 'G', 'F', 'C', 'D']

>>> g2 = {'F': ['D','C','B'],
     'A': ['G','D','B'],
     'B': ['F','A','E'],
     'E': ['G','B'],
     'C': ['F'],
     'D': ['F','A'],
     'G': ['A','E']}
>>> g=Graph(g2)
>>> g.bfs('A')
['A', 'B', 'D', 'G', 'E', 'F', 'C']
>>> g.dfs('A')
['A', 'B', 'E', 'G', 'F', 'C', 'D']

>>> g3 = {'B': [('E',3),('C',5)],
     'F': [],
     'C': [('F',2)],
     'A': [('D',3),('B',2)],
     'D': [('C',1)],
     'E': [('F',4)]}
>>> g=Graph(g3)
>>> g.bfs('A')
['A', 'B', 'D', 'C', 'E', 'F']
>>> g.dfs('A')
['A', 'B', 'C', 'F', 'E', 'D']

g4 = {'Bran': ['East','Cap'],
     'Flor': [],
     'Cap': ['Flor'],
     'Apr': ['Dec','Bran'],
     'Dec': ['Cap'],
     'East': ['Flor']}
>>> g=Graph(g4)
>>> g.bfs('Apr')
['Apr', 'Bran', 'Dec', 'Cap', 'East', 'Flor']
>>> g.dfs('Apr')
['Apr', 'Bran', 'Cap', 'Flor', 'East', 'Dec']
```

Deliverables:

- Include all your code (including the stack and queue code) in your script named HW5.py. Submit it to the HW5 Gradescope assignment before the due date.

Using the Stack and Queue for graph traversals

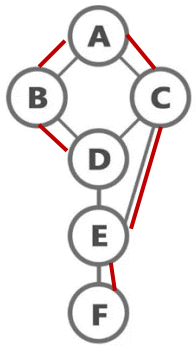




```
g = {'A': ['B','C'],
     'B': ['A','D'],
     'C': ['A','D','E'],
     'D': ['B','C','E'],
     'E': ['C','D','F'],
     'F': ['E']}
```

BFS

Considering a given node as the parent and connected nodes as neighbors, BFS will visit the sibling vertices before the neighbor vertices using this algorithm:



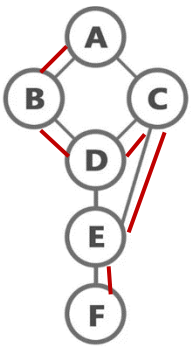

```
bfs(start)
    Let Q be an empty queue
    Q.enqueue(start)
    Mark 'start' as visited, you can use a list to add visited nodes
    while Q is not empty:
        #Remove an item from Q, its neighbors will be visited now
        node = Q.dequeue( )
        # For each neighbor, add it to the queue if it has not been visited
        for each neighbor 'x' of node:
            if x is not visited
                Enqueue x in Q to further visit its neighbors
```

Action	Current node	Queue	Unvisited	Visited
Start at node A		A	B, C, D, E, F	A
Dequeue	A		B, C, D, E, F	A
A has unvisited neighbors, B and C	A		B, C, D, E, F	A
Mark B as visited and enqueue it	A	B	C, D, E, F	A, B
Mark C as visited and enqueue it	A	B, C	D, E, F	A, B, C

A has no more unvisited neighbors, dequeue a new current node	B	C	D, E, F	A, B, C
B has unvisited neighbors, D	B	C	D, E, F	A, B, C
Mark D as visited and enqueue it	B	C, D	E, F	A, B, C, D
B has no more unvisited neighbors, dequeue a new current node	C	D	E, F	A, B, C, D
C has no more unvisited neighbors, dequeue a new current node	D		E, F	A, B, C, D
D has unvisited neighbors, E	D		E, F	A, B, C, D
Mark E as visited and enqueue it	D	E	F	A, B, C, D, E
D has no more unvisited neighbors, dequeue a new current node	E		F	A, B, C, D, E
E has unvisited neighbors, F	E		F	A, B, C, D, E
Mark F as visited and enqueue it	E	F		A, B, C, D, E, F
E has no more unvisited neighbors, dequeue a new current node	F			A, B, C, D, E, F
F has no more unvisited neighbors, dequeue a new current node				A, B, C, D, E, F
Queue is empty, there are no more unvisited nodes so the algorithm will terminate				

DFS

Considering a given node as the parent and connected nodes as neighbors, DFS will visit the neighbor vertices using this algorithm:




```
dfs(start)
    Let S be an empty stack
    S.push(start)
    while S is not empty:
       #Pop a node from stack to visit next
       node = S.pop( )
       if node is not visited:
            mark node as visited
           #Push all the neighbors of node in stack that are not visited, you can use reverse sorted to visit alphabetically
           for each neighbor ‘x’ of node: #reversed
                if x is not visited :
                    S.push(x)
```

Action	Stack	Unvisited	Visited
Start at node A	A	B, C, D, E, F	A
Check the top, A has unvisited neighbors, B and C	A	B, C, D, E, F	A
Mark node B as visited and push it	B, A	C, D, E, F	A, B
Check the top, B has unvisited neighbors, D	B, A	C, D E, F	A, B
Mark node D as visited and push it	D, B, A	C, E, F	A, B, D
Check the top, D has unvisited neighbors, C and E	D, B, A	C, E, F	A, B, D
Mark node C as visited and push it	C, D, B, A	E, F	A, B, D, C
Check the top, C has unvisited neighbors, E	C, D, B, A	E, F	A, B, D, C
Mark node E as visited and push it	E, C, D, B, A	F	A, B, D, C, E
Check the top, E has unvisited neighbors, F	E, C, D, B, A	F	A, B, D, C, E
Mark node F as visited and push it	F, E, C, D, B, A		A, B, D, C, E, F
Check the top, F has no unvisited neighbors	F, E, C, D, B, A		A, B, D, C, E, F
Pop	E, C, D, B, A		A, B, D, C, E, F
Check the top, E has no unvisited neighbors	E, C, D, B, A		A, B, D, C, E, F
Pop	C, D, B, A		A, B, D, C, E, F
Check the top, C has no unvisited neighbors	C, D, B, A		A, B, D, C, E, F
Pop	D, B, A		A, B, D, C, E, F
Check the top, D has no unvisited neighbors	D, B, A		A, B, D, C, E, F
Pop	B, A		A, B, D, C, E, F
Check the top, B has no unvisited neighbors	B, A		A, B, D, C, E, F
Pop	A		A, B, D, C, E, F
Check the top, A has no unvisited neighbors	A		A, B, D, C, E, F
Pop			A, B, D, C, E, F
Stack is empty, there are no more unvisited nodes so the algorithm will terminate			