Kareem Saleh

README: Topological Sort Analysis

Introduction

This document provides an overview of the Topological Sort Analysis, specifying its components, walkthrough and functions.

Features

1) Prompt user for number of vertices
2) Prompt user for number of edges
3) Prompting users for vertices involved in edges
4) Applying topological sort on the graph

Components

1) **Graph Class**
   - **Contains array of Vertices**
   - **Contains edges**
   - **Array of Linked Lists which forms an adjacency matrix**

2) **Vertex Class**
   - **Contains data that should be stored in vertices**
   - **Contains indegree number**
   - **Contains topological number**

3) **BrowserArrayList (From previous assignments)**
   - **Used to employ queue in topological sort**

4) **BrowserQueue (From previous assignments)**
   - **Used to enqueue and dequeue in topological sort**

Usage Instructions

- Prompt user for number of vertices
- Prompt user for number of edges

- Prompting users for vertices involved in edges
- Applying topological sort on the graph
- Prompt user for number of vertices
- Prompt user for number of edges

Order of Growth and Method Brief:-

BrowserArrayList Class:

- **getSize():**
  - getSize() is of order O(1) since it returns the private variable size, which takes constant time and doesn't depend on the size of the list
- **isEmpty():**
  - isEmpty is of order O(1) since we are just checking if front index is -1, which takes constant time and doesn't depend on the size of the list.
- **isFullNorm():**
  - isFullNorm is of order O(1) since we are just checking if front index is 0 and back index is capacity-1, which takes constant time and doesn't depend on the size of the list.
- **isFullLoop():**
  - isFullNorm is of order O(1) since we are just checking if (back+1) % capacity) == front, which takes constant time and doesn't depend on the size of the list.
- **add(String a):**
  - add is O(n) in the case of a full array since we have multiple single for loops allowing us to resize and copy elements and multiplication constant is ignored
  - add is O(1) in the case of a none-full array since we just put the element at the desired index
  - Therefore, add is of order O(n) since O(n) dominates the O(1)
- **delete():**
  - delete is O(1) since removing at end requires no for loops, which takes constant time and doesn't depend on the size of the list.
- **ArraydoClear():**
  - ArraydoClear() is of order O(1) since we just make front and back indices be -1 and creating(+assigning) new empty array, which takes constant time and doesn't depend on the size of the list.

BrowserQueue Class:

- **enqueue(String url):**
  - enqueue uses the Arraylist add which is O(n) in the case of a full array since we have multiple single for loops allowing us to resize and copy elements and multiplication constant is ignored
  - enqueue uses the Arraylist add which is O(1) in the case of a none-full array since we just put the element at the desired index, which takes constant time and doesn't depend on the size of the list.
  - Therefore, enqueue uses the Arraylist add which is of order O(n) since O(n) dominates the O(1)
- **dequeue():**
  - dequeue is using ArrayList's remove which is O(1) since we have front index and we are removing from the front of the queue by just incrementing front, removing element after storing it to return. This takes constant time and doesn't depend on the size of the list.
- **peek():**
  - peek is O(1) since we are just checking the data of the last node (top of the stack) without modifying the list, , which takes constant time and doesn't depend on the size of the list.

Vertex Class:-

- Data integer variable
- topNum integer variable
- indegree integer variable
- Parameterized constructor

Graph Class:

- **Vertex Class**

- **addEdge(int f, int t):**
  - addEdge is O(1) since it adds the element where the edge is directed towards at the end of the linked list with index equal to the data in the vertex where the edge is leaving from. Adding to the end of a linked list is O(1) since it is adding close to a known ptr and there is also incrementing of the to vertex's indegree which is also O(1)

- **topSort():**

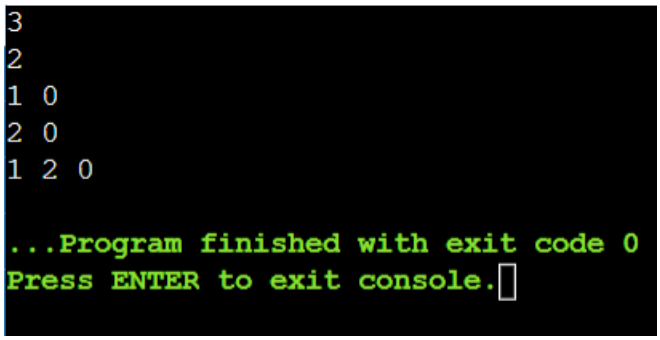
  - initializing queue is O(1)
  - initializing counter to 0 is O(1)
  - for each loop over the vertices array is O(1)
  - while loop is O(n) – it also has a nested for each loop inside which is O(1).
  - Enqueue which is O(n) in case of full array and O(1) in case of non-full array
  - Dequeue is O(1)
  - for each loop over the vertices array is O(1)
  - for each loop over the topologically ordered array is O(1)

    Overall, O(n) dominates. So, topological sort is O(n)

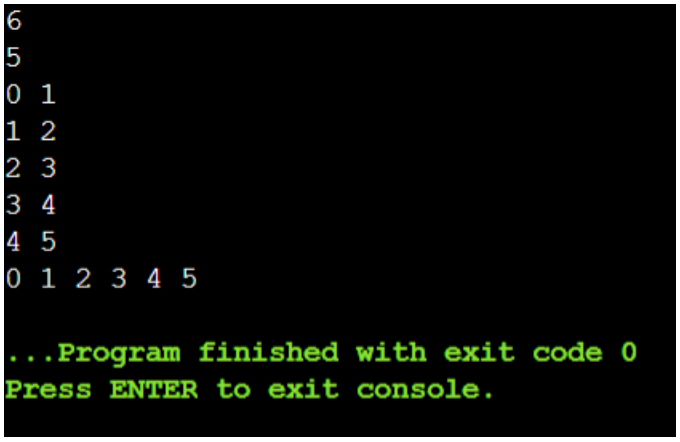
Test Cases (New Ones First then Test cases in assigned document)

Extra Test Cases:-

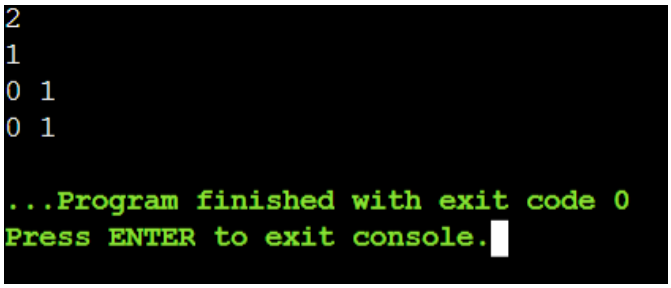
Test Case a:



Test Case b:



Test Case c:



Test Case d:

```
4
3
1 0
1 2
1 3
1 0 2 3

...Program finished with exit code 0
Press ENTER to exit console.
```

Test Case 1: Single Node Graph

```
1
0
0
```

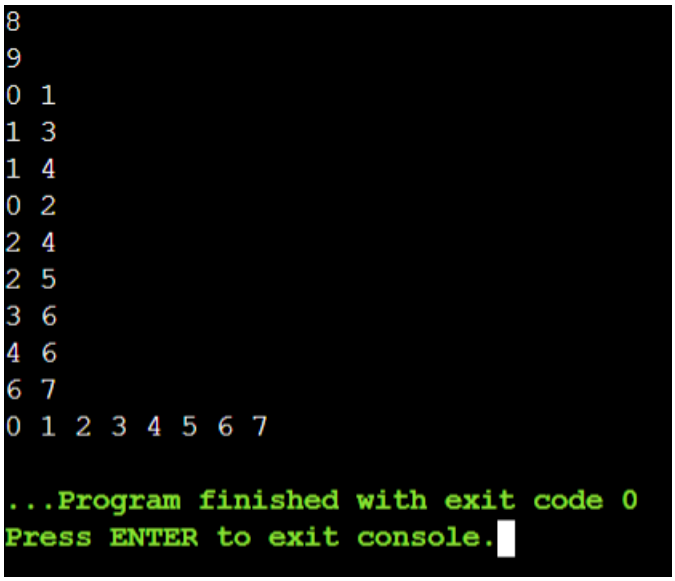
Same Output

Test Case 2: Simple DAG

```
5
4
0 1
0 2
1 3
2 4
0 1 2 3 4
```

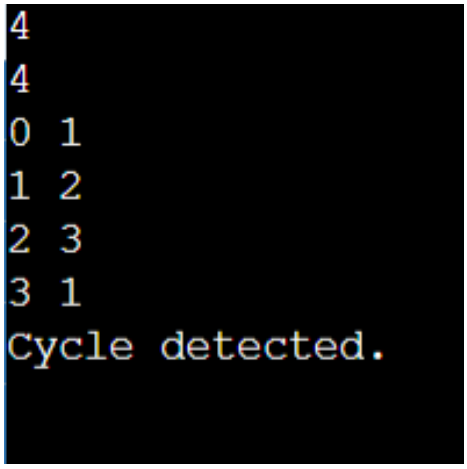
Same Output

Test Case 3: Complex DAG with Several Valid Outputs



Same Output

Test Case 4: Graph with Back Edge Indicating Cycle



Same Output

Test Case 5: Multiple Entry Points

```
5
4
1 0
2 0
3 1
3 2
3 4 1 2 0
```

Different than output, possibly because 4 is isolated.

Test Case 6: All nodes pointing to single node

```
4
3
0 3
1 3
2 3
0 1 2 3

...Program finished with exit code 0
Press ENTER to exit console.
```