CSCE 221 Cover Page

First Name Last Name UIN

User Name E-mail address

Please list all sources in the table below including web pages which you used to solve or implement the current homework. If you fail to cite sources you can get a lower number of points or even zero, read more on Aggie Honor System Office website: `http://aggiehonor.tamu.edu/`

Type of sources				
People				
Web pages (provide URL)				
Printed material				
Other Sources				

I certify that I have listed all the sources that I used to develop the solutions/codes to the submitted work.
On my honor as an Aggie, I have neither given nor received any unauthorized help on this academic work.

Your Name Date

CSCE 221 Assignment 4 (100 points)

Due Dates: See the Calendar

Objectives

We will consider a directed graph without cycles called **a directed acyclic graph** (DAG). In this assignment you are going to find a topological ordering in a DAG. There are many real life problems that can be modeled by such graphs and solved by the topological ordering algorithm. Read the section 9.2, pp. 382-385 in the textbook to learn more about the algorithm.

Programming (100 points)

The assignment consists of two parts:

- **Part 1** – implementation of the graph data structure
- **Part 2** – implementation of the topological ordering for a DAG. Please notice that we take a DAG as an input and, if no cycle exists, the topological ordering for the DAG is returned.

Part 1 (40 points)

The purpose of this part is to read in the data from an input file with a given format, build a graph data structure, and display the graph in text format.

You should implement a graph data structure which is defined based on an additional type `Vertex`. The implementation of the `Graph` class should be based on **adjacency lists**, see the file `graph.h`.

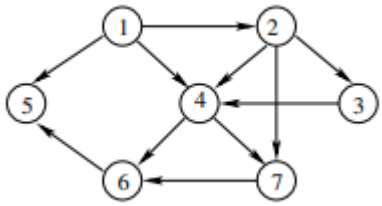
You should implement the following functions (`T` can be `int`, `char` or `string`):

- ```
  void buildGraph(istream &input)
  //build the graph from input
  // according to the specification below
  ```
- ```
  Vertex<T> at(T label)
  //returns the Vertex with the given label,
  // throws an exception if it is not present
  ```
- ```
  int size()
  //returns the number of vertices in the graphs
  ```

You are encouraged to use a hash table to store the vertices in the graph. `std::unordered_map` is one such a data structure. Using this library is covered at the end of the document.

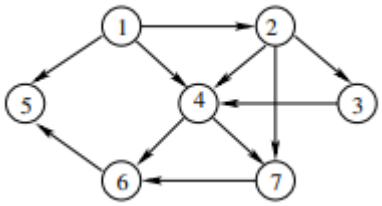
The graph is built by reading data from a text file with fixed format, see the example below. At each row, the first number is the label of the start vertex of a directed edge. Other numbers in this row are the end vertices accessed from the start vertex.

Example. The first row starts with the vertex 1 and provides information about three directed edges to vertices 2, 4 and 5. In the case when there is no edge from a certain vertex, for example for the vertex 5, the list is empty. This example is from input file called `input.data` provided with this assignment.




```
1 2 4 5
2 3 4 7
3 4
4 6 7
5
6 5
7 6
```
```

- We assume that the graph we are dealing with is sparse and unweighted. Then, **adjacency lists** will be a natural choice to store the connection between two vertices. The class `Graph` is used to store the graph and implements the necessary operations such as `buildGraph`. Furthermore, a `Vertex` class can be implemented to store the basic information about a graph vertex such as a label which in our case is an integer.
- The vertices are not necessarily numbered consecutively, making a **hash table** a logical choice data structure for storing vertices with labels as keys
- You may assume that the graph is fully specified by the input stream and will not be changed after building the graph. Note: The last row of the input file may or may not be an empty line. Hence, while parsing consider this case and ignore this last empty line if it exists.
- `displayGraph()` will print out each vertex and its adjacency list. For example, consider the graph G and its corresponding adjacency linked lists for an input sample graph (`input.data`). Test your program by reading a graph from an input file and use the function `displayGraph()` to display the generated graph in text format, see the output below.



1 : 2 4 5
2 : 3 4 7
3 : 4
4 : 6 7
5 :
6 : 5
7 : 6

Gradescope should accept the display of vertices in any order (the column before ":") and the adjacent vertices of a vertex in any order (the list of vertices after ":" of a vertex). For example, valid outputs of `input.data` are

7 : 6
6 : 5
5 :
1 : 2 4 5
2 : 3 4 7
3 : 4
4 : 6 7

- You can compile your code using this command line:

  `make`

  And you can run your program by executing:

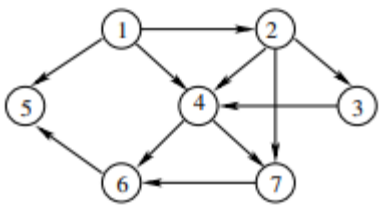
  `./main input.data`

Part 2 (60 points)

- The formal definition of the topological sort:

> Let G be a DAG with n vertices. A **topological ordering** of G is the ordering $v_1, v_2, \ldots, v_n$ of the vertices of G such that for every edge (v_i, v_j) of G we have $i < j$.

- The illustration of the definition of the topological sort ordering gives a sequence of vertices:



`1 2 3 4 7 6 5`

The topological sort ordering places vertices of the graph along the horizontal line with the following property: if there is an edge from the vertex v_i to the vertex v_j then the vertex v_i precedes v_j in the topological ordering.

- Topological sort algorithm:

  1. The input is a DAG
     (a) Algorithm – see the textbook, Fig. 9.7, p. 385 or **Algorithms** section below.
        – You can use `topNum` (`top_num`) as in Fig. 9.7 (Image is provided in **Algorithms** section as well), and then traverse the graph to initialize the topological sort ordering vector. `top_num` keeps track of the order of vertices in topological sort.
     (b) The output of the program should be a vector of vertices (or their labels) set in topological sort order.
        – You need to print the topological sort ordering vector by printing the labels of vertices.

     In this part you should implement the following functions:

     – `bool topological_sort()`
       `//performs the topological sort which returns true`
       `// if a topological ordering is found,`
       `// otherwise returns false.`
     – `void compute_indegree()`
       `//assigns the indegree, the number of inbound edges,`
       `// for each vertex`

Requirements

- Submit only the file `graph.h` to Gradescope.
- **Testing:** test your program for correctness using the cases below:

  Case 1: Use the example (`input.data`) provided in the description of the problem.

  Case 2: Samantha plans her course schedule. She is interested in the following eight courses: CSCE121, CSCE222, CSCE221, CSCE312, CSCE314, CSCE313, CSCE315, and CSCE411. The course prerequisites are:

course	#	prerequisites	
CSCE121:	1	(none)	
CSCE222:	2	(none)	
CSCE221:	3	CSCE121	CSCE222
CSCE312:	4	CSCE221	
CSCE314:	5	CSCE221	
CSCE313:	6	CSCE221	
CSCE315:	7	CSCE312	CSCE314
CSCE411:	8	CSCE222	CSCE221

  This will find a sequence of courses that allows Samantha to satisfy all the prerequisites. Assume that she can only take one class at a time. The input file for this case is provided in `input2.data`. Note: the table above contains courses and their prerequisites. The `input2.data` file contains the set of vertices and their corresponding adjacent vertices.

  Case 3: Create a directed graph with cycles and test your program. There is one such a file provided (`input-cycle.data`).

- **Algorithms:**

– Pseudocode for topological sort (topsort) from the textbook

```
void Graph::topsort( )
{
    Queue<Vertex> q;
    int counter = 0;

    q.makeEmpty( );
    for each Vertex v
        if( v.indegree == 0 )
            q.enqueue( v );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );
        v.topNum = ++counter;  // Assign next number

        for each Vertex w adjacent to v
            if( --w.indegree == 0 )
                q.enqueue( w );
    }

    if( counter != NUM_VERTICES )
        throw CycleFoundException{ };
}
```

Figure 9.7 Pseudocode to perform topological sort

– Pseudocode for calculating in-degree from the textbook

```
for each Vertex v
    v.indegree = 0;

for each Vertex v
    for each Vertex w adjacent to v
        w.indegree++;
```

- **Using the C++ Standard Library:**

  There are several C++ standard library containers you can use:

  – `std::unordered_set`
  – `std::unordered_map`
  – `std::set`
  – `std::map`

  The former two use a hash table, the latter two use red-black trees. The set elements are immutable whereas map elements are mutable. The other key difference is that the unordered data structures require the elements to have an ordering defined via `operator<`. This allows the in-order traversal of vertices based on this ordering. This would have been great for `print_top_sort()`, however, as the topological ordering is not known at insertion time, this cannot be used for ordering; `std::unordered_map` is the *preferable* data structure.

To aid in working with this data structure, the following code is provided:

```
//create the unordered_map object
//the two template types are for key and value type
unordered_map<T, Vertex<T>> node_set;

//create and insert a new object with key token
//  if a key in the table with this item exists,
//  the new object is not inserted
//returns a pair<unordered_map<T, Vertex<T>>::iterator, bool>
//  where iterator is a reference to the object in the hash table,
//  bool is true if this is the first time insert, false otherwise
auto pair = node_set.insert(make_pair(label, Vertex<T>{label, 0}));

bool newItem = pair.second; //true if this is the first item with the given key

unordered_map<T, Vertex<T>>::iterator iter = pair.first;

//the iterator can be dereferenced to get the object back
Vertex<T> v = *iter; //create a copy of the v object
Vertex<T>& v = *iter; //create a reference to v in the map

//WARNING: references are only valid until the next insert is made
//  - they should never be stored in variables
//  - pointers to them should never be made


//Working with STL data structures require considering
// when references and copies are used.
//The trivial solution is here:
Vertex<T> v = node_set.at(label); //copy assignment for v .
v.top_num = 0; //or other changes to v
node_set.at(label) = v;


//Alternatively, using references can save some copies
//top_num is 0 by default
cout << node_set.at(label).top_num << endl;   // outputs 0
Vertex<T>& vRef = node_set.at(label);       // by reference
vRef.top_num += 1;  //incrementing the object in the map
cout << node_set.at(label).top_num << endl;   // outputs 1
Vertex<T> vCopy = node_set.at(label);  //copy assignment
vCopy.top_num += 1;             // increments the copy
cout << node_set.at(label).top_num << endl; // outputs 1 again
node_set.at(label).top_num += 1; //incrementing the object in the map
cout << node_set.at(label).top_num << endl;   // outputs 2

//much of the same applies to iterating the map object
//elements by reference, updates within the map
for(auto& v: node_set){
  v.second.indegree = 0;
}
//elements by copy, no updates to the item in the map
for(auto v: node_set){
  v.second.indegree = 0;
}
//in both the cases auto is pair<T,Vertex<T>> type object
//in case this is a new syntax for you:
// these for loops are iterating over all objects in the map
```

(*Optional*) Another data structure you may be using for the first time is `std::priority_queue`, implemented via a binary heap. It takes one or three template parameters: `<Type, ContainerType, Functor>`. The priority queue orders by maximizing, meaning that the greatest priority element is returned first. The stored type or the functor should implement the `operator<()`. The Container type is unimportant, `std::vector<Vertex<T>>` can be used. The code for declaring a functor class is show below. Recall that the implementation of topological ordering likely assigns the first ordered elements a lower value.

```
// syntax for a custom comparator
template<class T>
class VertexCompare {
public:
  bool operator()(Vertex<T> v1, Vertex<T> v2){
    //TODO - implement
    return false;
  }
};
...
priority_queue<Vertex<T>, vector<Vertex<T>>, VertexCompare<T>> pq;
```