CSCE 221 Cover Page
Programming Assignment #5

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. According to the University Regulations, Section 42, scholastic dishonesty are including: acquiring answers from any unauthorized source, working with another person when not specifically permitted, observing the work of other students during any exam, providing answers when not specifically authorized to do so, informing any person of the contents of an exam prior to the exam, and failing to credit sources used. Disciplinary actions range from grade penalties to expulsion read more: Aggie Honor System Office

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."

Electronic signature Date

Programming Assignment 5

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.

- The assignment consists of three 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 the topological ordering for the DAG is returned; or the exception message is displayed: "*There are cycles in the graph*".
  - **Part 3** – preparing a report:
    - * discussing the implementation of the Part 1 and 2 and the running time of the algorithms used to solve the problem.
    - * providing testing cases for correctness
- **Part 1 (40 points)**

  In this part you should implement a graph data structure which is defined based on an additional type `Vertex`. You can download the supplementary (zipped) file with a code skeleton from the eCampus. The implementation of the `Graph` class should be based on adjacency lists, see the file `graph.h`.

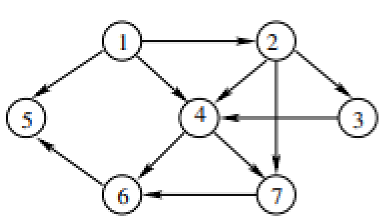
  You should implement the following functions (`T` can be `int`, `char` or `string`):

  - ```
    void buildGraph(istream &input)
    //takes an istream and build the graph according to the specification below
    ```
  - ```
    void displayGraph(ostream &o)
    //prints the graph according to the specification into ostream,
    // nodes can be printed in any order
    ```
  - ```
    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 nodes 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 populated 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.

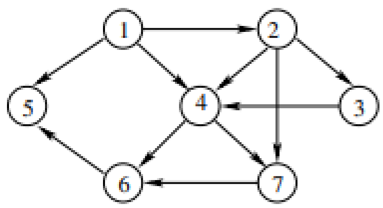
  Example. The first row starts with the vertex 1 and provides information about three directed edges to vertices 2, 4 and 5. In the case when there is no edge for a certain vertex, for example for the vertex 5, the list is empty. This input file is called `input.data` in the supplementary file.




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

  - 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 on the screen in text format.

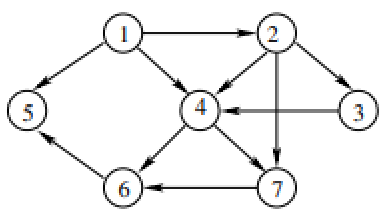
- We assume that the graph we are dealing with is sparse and unweighted. Then, adjacency lists will be a natural choice to store the connection between two nodes. The class `Graph` is used to store the graph and implements the necessary operations such as `buildGraph`, and so on. Furthermore, a `Vertex` class can be implemented to store the basic information about a graph node such as a label which in our case is an integer.
- The nodes 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
- `displayGraph()` should print out each vertex and its adjacency list on the screen. 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 on the screen, see the format of the output below.



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

- You can compile your code using this command line:
`make`
- And you can run your program by executing:
`./main input.data`

- **Part 2 (40 points)**
  - The formal definition of a 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, $i < j$.
  - 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
    2. Algorithm – see the textbook, Fig. 9.7, p. 385.
       * You can use `topNum` (`top_num`) as in Fig. 9.7, and then traverse the graph to initialize the topological sort ordering vector.
    3. 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.

  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 node.`
* `void print_top_sort(ostream& o, bool addNewline=true)`
  `//prints the topological ordering into the ostream`
  `//if the second parameter is true, insert a newline at the end.`

This may require another data structure, such as `std::priority_queue`, which may alter the runtime.

- **Part 3 (20 points)**
  - Submit to Mimir the source code
  - Submit to eCampus an electronic version of your report including
    * (15 points) description of your implementation, C++ features used, assumptions on input data.
      · Why does the topological sort algorithm use a queue? Can we use a stack instead?
      · Can you explain why the algorithm detects cycles?
      · What is the running time for each function? Use the Big-O notation asymptotic notation and justify your answer.
    * (5 points) test your program for correctness using the four 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

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 (`input2.data`)

Case 3: Samantha loves foreign languages and wants to plan her course schedule. She is interested in the following nine courses: LA15, LA16, LA22, LA31, LA32, LA126, LA127, LA141, and LA169. The course prerequisite are:

course	#	prerequisites	
LA15:	1	(none)	
LA16:	2	LA15	
LA22:	3	(none)	
LA31:	4	LA15	
LA32:	5	LA16	LA31
LA126:	6	LA22	LA32
LA127:	7	LA16	
LA141:	8	LA22	LA16
LA169:	9	LA32	

Find a sequence of courses that allows Samantha to satisfy all the prerequisites. Assume that she can only take one class at a time.

Case 4. Create a directed graph with cycles and test your program. There is one such a file provided (`input-cycle.data`).

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

  There are several C++ standard library containers that are of note:

  - `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 (`operator<` defined). This allows the in-order traversal of nodes based on this ordering. This would be 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
```

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 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:
  //will be called as a < operator
  bool operator()(Vertex<T> v1, Vertex<T> v2){
    //TODO - implement
        return false;
  }
};
...
priority_queue<Vertex<T>, vector<Vertex<T>>, VertexCompare<T>> pq;
```