CSCI 230 PA 10 Submission

Due Date: 05/18/2022 Late (date and time):________________

Name(s): Nero Li

Header file code below:

Entry.h:

```
#ifndef ENTRY_H
#define ENTRY_H
// Modified for CSCI 220 Fall 15
// Updated Fall 21

template <typename K, typename V>
class Entry {                                              // a (key,
value) pair
public:                                                        // public
functions
    typedef K Key;                                     // key type
    typedef V Value;                                   // value type

    Entry(const K& k = K(), const V& v = V())     // constructor
        : _key(k), _value(v) { }

    const K& key() const { return _key; }         // get key

    const V& value() const { return _value; }     // get value

    void setKey(const K& k) { _key = k; }         // set key

    void setValue(const V& v) { _value = v; }     // set value

    bool operator==(const Entry a)
    {
        return (_key == a.key() && _value == a.value());
    }

private:                                                   // private data
    K _key;                                                // key
    V _value;                                              // value
};
#endif
```

HeapPriorityQueue.h:

```
#ifndef HPQ_H
#define HPQ_H
```

```
#include <list>
#include <vector>

template <typename E>
class VectorCompleteTree
{
private:                                                          // member
data
    std::vector<E> V;                                           // tree
contents
public:                                                               //
publicly accessible types
    typedef typename std::vector<E>::iterator Position; // a position in
the tree
protected:                                                        //
protected utility functions
    Position pos(int i)                                         // map an
index to a position
      { return V.begin() + i; }
    int idx(const Position& p) const                    // map a position
to an index
      { return p - V.begin(); }
public:
    VectorCompleteTree() : V(1) {}                         // constructor
    int size() const                                            { return
V.size() - 1; }
    Position left(const Position& p)                    { return
pos(2*idx(p)); }
    Position right(const Position& p)                   { return
pos(2*idx(p) + 1); }
    Position parent(const Position& p)                        { return
pos(idx(p)/2); }
    bool hasLeft(const Position& p) const               { return 2*idx(p)
<= size(); }
    bool hasRight(const Position& p) const              { return 2*idx(p) +
1 <= size(); }
    bool isRoot(const Position& p) const                { return idx(p) ==
1; }
    Position root()
{ return pos(1); }
    Position last()
{ return pos(size()); }
    void addLast(const E& e)
{ V.push_back(e); }
    void swap(const Position& p, const Position& q)     { E e = *q; *q =
*p; *p = e; }

    // New function added for CSCI 230 PA10
    Position removeLast()
    {
        Position p = V.end();
        p--;
        V.pop_back();
        return p;
    }
    void remove(E e)
    {
        Position p = V.begin();
        for (auto i : V)
        {
            if (i == e)
            {
                V.erase(p);
                return;
            }
            p++;
        }

    }
};

template <typename E, typename C>
class HeapPriorityQueue
{
public:
    int size() const;                      // number of elements
    bool empty() const;                    // is the queue empty?
    E insert(const E& e);            // insert element
    const E& min();                            // minimum element
    E removeMin();                   // remove minimum

    // New function added for CSCI 230 PA10
    void replace(const E& oldElem, const E& newElem)
    {
        T.remove(oldElem);
        insert(newElem);
    }
private:
    VectorCompleteTree<E> T;         // priority queue contents
    C isLess;                                  // less-than comparator
                                             // shortcut for tree position
    typedef typename VectorCompleteTree<E>::Position Position;
};

template <typename E, typename C>          // number of elements
int HeapPriorityQueue<E,C>::size() const
{
    return T.size();
}

template <typename E, typename C>          // is the queue empty?
bool HeapPriorityQueue<E,C>::empty() const
{
    return size() == 0;
}

template <typename E, typename C>           // minimum element
const E& HeapPriorityQueue<E,C>::min()
{
    return *(T.root());                 // return reference to root
element
}

template <typename E, typename C>           // insert element
E HeapPriorityQueue<E,C>::insert(const E& e)
{
    T.addLast(e);                             // add e to heap
    Position v = T.last();                      // e's position
    while (!T.isRoot(v))                // up-heap bubbling
    {
        Position u = T.parent(v);
        if (!isLess(*v, *u)) break;         // if v in order, we're done
        T.swap(v, u);                           // ...else swap with parent
        v = u;
    }

    return e;
}

template <typename E, typename C>           // remove minimum
E HeapPriorityQueue<E,C>::removeMin()
{
    Position p;
    if (size() == 1)                        // only one node?
        p = T.removeLast();                         // ...remove it
    else
    {
        Position u = T.root();              // root position
        T.swap(u, T.last());                // swap last with root
        p = T.removeLast();                             // ...and remove
last
        while (T.hasLeft(u))            // down-heap bubbling
        {
            Position v = T.left(u);
            if (T.hasRight(u) && isLess(*(T.right(u)), *v))
                v = T.right(u);             // v is u's smaller child
            if (isLess(*v, *u))         // is u out of order?
            {
                T.swap(u, v);               // ...then swap
                u = v;
            }
            else break;                     // else we're done
        }
    }
    return *p;
}

#endif
```

Decorator.h:

```
#pragma once
#include <string>
#include <map>

using namespace std;

// Created by T. Vo for CSCI 230
// Based on C++ code fragment of Goodrich book

class Object {                              // generic object
public:
      virtual int      intValue()    const; // throw(bad_cast);
      virtual string   stringValue() const ;      // throw(bad_cast);
};


class String : public Object {
private:
      string value;
public:
      String(string v = "") : value(v) { }
      string getValue() const
      {
            return value;
      }
};


class Integer : public Object {
private:
      int value;
public:
      Integer(int v = 0) : value(v) { }
      int getValue() const
      {
            return value;
      }
};

int Object::intValue() const // throw(bad_cast) {       // cast to Integer
{
      const Integer* p = dynamic_cast<const Integer*>(this);
      if (p == NULL) throw exception(); // ("Illegal attempt to cast to 
Integer");
      return p->getValue();
```

```
}

string Object::stringValue() const { // throw(bad_cast) {            //
cast to String
      const String* p = dynamic_cast<const String*>(this);
      if (p == NULL) throw exception(); // ("Illegal attempt to cast to
Srring");
      return p->getValue();
}


class Decorator {
private:                                                        // member
data
      std::map<string, Object*> map1;             // the map
public:
      Object * get(const string& a)               // get value of
attribute
      {
            return map1[a];
      }
      void set(const string& a, Object* d)  // set value
      {
            map1[a] = d;
      }
};
```

Graph.h:

```
#pragma once
#include <vector>
#include <list>
#include <string>
#include "Decorator.h"

using namespace std;

// Created by T. Vo for CSCI 230
// Based on Java version of Goodrich book w/o template
// string for vertex and int for edge
// Version 1.1

class Vertex : public Decorator      // behaves like interface in Java
{
public:
      virtual string getElement() = 0;
};


class Edge : public Decorator               // behaves like interface in
Java
{
```

```
public:
      virtual int getElement() = 0;
};

class Graph
{
public:
      /* Returns the number of vertices of the graph */
      virtual int numVertices() = 0;

      /* Returns the number of edges of the graph */
      virtual int numEdges() = 0;

      /* Returns the vertices of the graph as an iterable collection */
      virtual list<Vertex *> getVertices() = 0;

      /* Returns the edges of the graph as an iterable collection */
      virtual list<Edge *> getEdges() = 0;

      /*
      * Returns the number of edges leaving vertex v.
      * Note that for an undirected graph, this is the same result
      * returned by inDegree
      * throws IllegalArgumentException if v is not a valid vertex?
      */
      virtual int outDegree(Vertex *v) = 0; // throws
IllegalArgumentException;

      /**
      * Returns the number of edges for which vertex v is the destination.
      * Note that for an undirected graph, this is the same result
      * returned by outDegree
      * throws IllegalArgumentException if v is not a valid vertex
      */
      virtual int inDegree(Vertex *v) = 0; // throws
IllegalArgumentException;

      /*
      * Returns an iterable collection of edges for which vertex v is the
origin.
      * Note that for an undirected graph, this is the same result
      * returned by incomingEdges.
      * throws IllegalArgumentException if v is not a valid vertex
      */
      virtual vector<Edge *> outgoingEdges(Vertex *v) = 0; // throws
IllegalArgumentException;

      /*
      * Returns an iterable collection of edges for which vertex v is the
destination.
      * Note that for an undirected graph, this is the same result
      * returned by outgoingEdges.
```

```
	* throws IllegalArgumentException if v is not a valid vertex
	*/
	virtual vector<Edge *> incomingEdges(Vertex *v) = 0; // throws
IllegalArgumentException;

	/** Returns the edge from u to v, or null if they are not adjacent.
*/
	virtual Edge *getEdge(Vertex *u, Vertex *v) = 0; // throws
IllegalArgumentException;

	/*
	* Returns the vertices of edge e as an array of length two.
	* If the graph is directed, the first vertex is the origin, and
	* the second is the destination.  If the graph is undirected, the
	* order is arbitrary.
	*/
	virtual vector<Vertex *> endVertices(Edge *e) = 0; // throws
IllegalArgumentException;

	/* Returns the vertex that is opposite vertex v on edge e. */
	virtual Vertex *opposite(Vertex *v, Edge *e) = 0; // throws
IllegalArgumentException;

	/* Inserts and returns a new vertex with the given element. */
	virtual Vertex *insertVertex(string element) = 0;

	/*
	* Inserts and returns a new edge between vertices u and v, storing
given element.
	*
	* throws IllegalArgumentException if u or v are invalid vertices, or
if an edge already exists between u and v.
	*/
	virtual Edge *insertEdge(Vertex *u, Vertex *v, int element) = 0; //
throws IllegalArgumentException;

	/* Removes a vertex and all its incident edges from the graph. */
	virtual void removeVertex(Vertex *v) = 0; // throws
IllegalArgumentException;

	/* Removes an edge from the graph. */
	virtual void removeEdge(Edge *e) = 0; // throws
IllegalArgumentException;

	virtual void print() = 0;
};
```

AdjacencyListGraph.h:

```
#pragma once
#include <iostream>
#include <list>
```

```
#include <vector>
#include <map>
#include "Graph.h"

using namespace std;

// Created by T. Vo for CSCI 230
// Based on Java version of Goodrich book w/o template
// and minimal exception handling
// Version 1.1
// Some operations are incomplete and there are provisions
// to change from map to a list/vector for adjacency list

class AdjacencyListGraph : public Graph
{
private:
      bool isDirected;
      list<Vertex *> vertices;
      list<Edge *> edges;

      /* A vertex of an adjacency map graph representation. */
      class InnerVertex : public Vertex
      {
      private:
            string element;
            Vertex *pos;
            vector<pair<Vertex *, Edge *>> *outgoing;
            vector<pair<Vertex *, Edge *>> *incoming;
      public :

            /* Constructs a new InnerVertex instance storing the given
element. */
            InnerVertex(string elem, bool graphIsDirected = false) {
                  element = elem;
                  outgoing = new vector<pair<Vertex *, Edge *>>();
                  if (graphIsDirected)
                        incoming = new vector<pair<Vertex *, Edge *>>();
                  else
                        incoming = outgoing;    // if undirected, alias
outgoing map
            }

            /* Returns the element associated with the vertex. */
            string getElement() { return element; }

            /* Stores the position of this vertex within the graph's
vertex list. */
            void setPosition(Vertex *p) { pos = p; }

            /* Returns the position of this vertex within the graph's
vertex list. */
            Vertex *getPosition() { return pos; }
            /* Returns reference to the underlying map of outgoing edges.
*/
            vector<pair<Vertex *, Edge *>> *getOutgoing() { return
outgoing; }

            /* Returns reference to the underlying map of incoming edges.
*/
            vector<pair<Vertex *, Edge *>> *getIncoming() { return
incoming; }
      }; //------------ end of InnerVertex class ------------

      //---------------- nested InnerEdge class ----------------
      /* An edge between two vertices. */
      class InnerEdge : public Edge
      {
      private:
            int element;
            Edge *pos;
            vector<Vertex *> endpoints;

      public:
            /* Constructs InnerEdge instance from u to v, storing the
given element. */
            InnerEdge(Vertex *u, Vertex *v, int elem)
            {
                  element = elem;
                  endpoints.push_back(u);
                  endpoints.push_back(v);
            }

            /* Returns the element associated with the edge. */
            int getElement() { return element; }

            /* Returns reference to the endpoint array. */
            vector<Vertex *> getEndpoints() { return endpoints; }

            /* Stores the position of this edge within the graph's vertex
list. */
            void setPosition(Edge *p) { pos = p; }

            /* Returns the position of this edge within the graph's vertex
list. */
            Edge *getPosition() { return pos; }
      }; //------------ end of InnerEdge class ------------

public:
      /*
      * Constructs an empty graph.
      * The parameter determines whether this is an undirected or directed
graph.
      */
```

```
AdjacencyListGraph(bool directed = false)
{
      isDirected = directed;
}

~AdjacencyListGraph()
{

}

/* Returns the number of vertices of the graph */
int numVertices()
{
      return static_cast<int>(vertices.size());
}

/* Returns the number of edges of the graph */
int numEdges()
{
      return static_cast<int>(edges.size());
}

/* Returns the vertices of the graph as an iterable collection */
list<Vertex *> getVertices()
{
      return vertices;
}

/* Returns the edges of the graph as an iterable collection */
list<Edge *> getEdges()
{
      return edges;
}

/*
* Returns the number of edges leaving vertex v.
* Note that for an undirected graph, this is the same result
* returned by inDegree
* throws IllegalArgumentException if v is not a valid vertex?
*/
int outDegree(Vertex *v) // throws IllegalArgumentException;
{
      InnerVertex *vert =  static_cast<InnerVertex *>(v);
      return static_cast<int>(vert->getOutgoing()->size());
}

/**
* Returns the number of edges for which vertex v is the destination.
* Note that for an undirected graph, this is the same result
* returned by outDegree
* throws IllegalArgumentException if v is not a valid vertex
*/
```

```
	int inDegree(Vertex *v) // throws IllegalArgumentException;
	{
		InnerVertex *vert = static_cast<InnerVertex *>(v);
		return static_cast<int>(vert->getIncoming()->size());
	}

	/*
	* Returns an iterable collection of edges for which vertex v is the
origin.
	* Note that for an undirected graph, this is the same result
	* returned by incomingEdges.
	* throws IllegalArgumentException if v is not a valid vertex
	*/
	vector<Edge *> outgoingEdges(Vertex *v) // throws
IllegalArgumentException;
	{
		vector<Edge *> temp;
		vector<pair<Vertex *, Edge *>> *mapPtr =
static_cast<InnerVertex *>(v)->getOutgoing();
		for (auto it = mapPtr->begin(); it != mapPtr->end(); ++it) {
			temp.push_back(it->second);
		}
		return temp;
	}

	/*
	* Returns an iterable collection of edges for which vertex v is the
destination.
	* Note that for an undirected graph, this is the same result
	* returned by outgoingEdges.
	* throws IllegalArgumentException if v is not a valid vertex
	*/
	vector<Edge *> incomingEdges(Vertex *v) // throws
IllegalArgumentException;
	{
		vector<Edge *> temp;
		vector<pair<Vertex *, Edge *>> *mapPtr =
static_cast<InnerVertex *>(v)->getIncoming();
		for (auto it = mapPtr->begin(); it != mapPtr->end(); ++it) {
			temp.push_back(it->second);
		}
		return temp;
	}

	/* Returns the edge from u to v, or null if they are not adjacent.
*/
	Edge *getEdge(Vertex *u, Vertex *v) // throws
IllegalArgumentException;
	{
		Edge *temp = nullptr;
		vector<Edge *> out = outgoingEdges(u);
		for (auto i : out)
```

```
                if (opposite(u, i)->getElement() == v->getElement())
                    temp = i;
            return temp; // origin.getOutgoing().get(v);    // will be
null if no edge from u to v
        }

        /*
        * Returns the vertices of edge e as an array of length two.
        * If the graph is directed, the first vertex is the origin, and
        * the second is the destination.  If the graph is undirected, the
        * order is arbitrary.
        */
        vector<Vertex *> endVertices(Edge *e) // throws
IllegalArgumentException;
        {
            vector<Vertex *> endpoints = static_cast<InnerEdge
*>(e)->getEndpoints();
            return endpoints;
        }

        /* Returns the vertex that is opposite vertex v on edge e. */
        Vertex *opposite(Vertex *v, Edge *e) // throws
IllegalArgumentException;
        {
            vector<Vertex *> endpoints = static_cast<InnerEdge
*>(e)->getEndpoints();

            if (endpoints[0] == v)
                return endpoints[1];
            else
                return endpoints[0];
        }

        /* Inserts and returns a new vertex with the given element. */
        Vertex *insertVertex(string element)
        {
            Vertex *v = new InnerVertex(element, isDirected);
            vertices.push_back(v);
            static_cast<InnerVertex *>(v)->setPosition(vertices.back());
            return v;
        }

        /*
        * Inserts and returns a new edge between vertices u and v, storing
given element.
        *
        * throws IllegalArgumentException if u or v are invalid vertices, or
if an edge already exists between u and v.
        */
        Edge *insertEdge(Vertex *u, Vertex *v, int element) // throws
IllegalArgumentException;
        {
            Edge * e = new InnerEdge(u, v, element);
            edges.push_back(e);
            static_cast<InnerEdge *>(e)->setPosition(edges.back());
            InnerVertex *origin = static_cast<InnerVertex *>(u);
            InnerVertex *dest = static_cast<InnerVertex *>(v);
            (origin->getOutgoing())->push_back(pair<Vertex*, Edge*>(v,
e));
            (dest->getIncoming())->push_back(pair<Vertex*, Edge*>(u, e));

            return e;
      }

      /* Removes a vertex and all its incident edges from the graph. */
      void removeVertex(Vertex *v) // throws IllegalArgumentException;
      {
            //for (Edge<E> e : vert.getOutgoing().values())
            //    removeEdge(e);
            //for (Edge<E> e : vert.getIncoming().values())
            //    removeEdge(e);
            //// remove this vertex from the list of vertices
            //vertices.remove(vert.getPosition());

      }

      /* Removes an edge from the graph. */
      void removeEdge(Edge *e) // throws IllegalArgumentException;
      {
            // remove this edge from vertices' adjacencies
            //InnerVertex<V>[] verts = (InnerVertex<V>[])
edge.getEndpoints();
            //verts[0].getOutgoing().remove(verts[1]);
            //verts[1].getIncoming().remove(verts[0]);
            //// remove this edge from the list of edges
            //edges.remove(edge.getPosition());

      }

      void print()
      {
            for (auto itr = vertices.begin(); itr != vertices.end();
itr++)
            {
                  cout << "Vertex " << (*itr)->getElement() << endl;
                  if (isDirected)
                        cout << " [outgoing]";
                  cout << " " << outDegree(*itr) << " adjacencies:";
                  for (auto e : outgoingEdges(*itr))
                        cout << "(" << opposite(*itr, e)->getElement() <<
", " << e->getElement() << ")" << "  ";
                  cout << endl;
                  if (isDirected)
                  {
                        cout << " [incoming]";
                        cout << " " << inDegree(*itr) << " adjacencies:";
                        for (auto e : incomingEdges(*itr))
                              cout << "(" << opposite(*itr,
e)->getElement() << ", " << e->getElement() << ")" << "  ";
                        cout << endl;
                  }
            }
      }
};
```

Exercise 1 -- need to submit source code and I/O

 -- check if completely done ✓; otherwise, discuss issues below

Source code below:
exercise_1.cpp:

```
/*  Program: PA_10_exercise_1
    Author: Nero Li
    Class: CSCI 230
    Date: 05/17/2022
    Description:
        Implement Transitive Closure for a digraph using
        AdjacencyListGraph class from previous PA. Test it out on a
        simple example above (can assume each edge has a weight of 1).
        Print both original digraph and updated digraph.

    I certify that the code below is my own work.

      Exception(s): N/A

*/

#include <iostream>
#include "AdjacencyListGraph.h"

using namespace std;

bool areAdjacent(AdjacencyListGraph &G, Vertex *v1, Vertex *v2)
{
    vector<Edge *> out = G.outgoingEdges(v1);
    for (auto i : out)
        if (G.opposite(v1, i)->getElement() == v2->getElement())
            return true;

    return false;
}

void floydWarshall(AdjacencyListGraph &G)
{
    AdjacencyListGraph Gpre(true);
    AdjacencyListGraph Gcur(true);
    vector<Vertex *> v;
    v.push_back(NULL);
    int n = G.numVertices();
    for (auto i : G.getVertices())
        v.push_back(i);

    Gpre = G;
    for (int k = 1; k <= n; ++k)
    {
        Gcur = Gpre;
        for (int i = 1; i <= n; ++i)
            if (i != k)
                for (int j = 1; j <= n; ++j)
                    if (j != i && j != k)
                        if (areAdjacent(Gpre, v[i], v[k]) &&
areAdjacent(Gpre, v[k], v[j]))
                            if (!areAdjacent(Gcur, v[i], v[j]))
                                Gcur.insertEdge(v[i], v[j], k);
    }

    Gcur.print();
}

int main()
{
    AdjacencyListGraph G(true);

    Vertex *A = G.insertVertex("A");
    Vertex *B = G.insertVertex("B");
    Vertex *C = G.insertVertex("C");
    Vertex *D = G.insertVertex("D");
    Vertex *E = G.insertVertex("E");
    G.insertEdge(A, D, 1);
    G.insertEdge(A, E, 1);
    G.insertEdge(B, A, 1);
    G.insertEdge(B, C, 1);
    G.insertEdge(C, D, 1);
    G.insertEdge(D, E, 1);
    G.insertEdge(E, C, 1);
    cout << "Original Graph:\n";
    G.print();
    cout << endl;

    cout << "Transitive Closure:\n";
    floydWarshall(G);
    cout << endl;

    cout << "Modified by: Nero Li\n";

    return 0;
}
```

Input/output below:

```
Original Graph:
Vertex A
 [outgoing] 2 adjacencies:(D, 1)  (E, 1)
 [incoming] 1 adjacencies:(B, 1)
Vertex B
 [outgoing] 2 adjacencies:(A, 1)  (C, 1)
 [incoming] 0 adjacencies:
Vertex C
 [outgoing] 1 adjacencies:(D, 1)
 [incoming] 2 adjacencies:(B, 1)  (E, 1)
Vertex D
 [outgoing] 1 adjacencies:(E, 1)
 [incoming] 2 adjacencies:(A, 1)  (C, 1)
Vertex E
 [outgoing] 1 adjacencies:(C, 1)
 [incoming] 2 adjacencies:(A, 1)  (D, 1)

Transitive Closure:
Vertex A
 [outgoing] 3 adjacencies:(D, 1)  (E, 1)  (C, 5)
 [incoming] 1 adjacencies:(B, 1)
Vertex B
 [outgoing] 4 adjacencies:(A, 1)  (C, 1)  (D, 1)  (E, 1)
 [incoming] 0 adjacencies:
Vertex C
 [outgoing] 2 adjacencies:(D, 1)  (E, 4)
 [incoming] 4 adjacencies:(B, 1)  (E, 1)  (A, 5)  (D, 5)
Vertex D
 [outgoing] 2 adjacencies:(E, 1)  (C, 5)
 [incoming] 4 adjacencies:(A, 1)  (C, 1)  (B, 1)  (E, 3)
Vertex E
 [outgoing] 2 adjacencies:(C, 1)  (D, 3)
 [incoming] 4 adjacencies:(A, 1)  (D, 1)  (B, 1)  (C, 4)

Modified by: Nero Li
```

Exercise 2 (with extra credit) -- need to submit source code and I/O

 -- check if completely done ✓; otherwise, discuss issues below

Source code below:
exercise_2.cpp:

```
/*  Program: PA_10_exercise_2
    Author: Nero Li
    Class: CSCI 230
    Date: 05/17/2022
    Description:
        Implement a shortest path algorithm for a graph (preferably
        Dijkstra's algorithm) using AdjacencyListGraph class from
        previous PA. Test it out using digraph above. Find the shortest
        path from B to E using the weights below. Print out the path
        how to get from source vertex to a destination vertex.

    I certify that the code below is my own work.

      Exception(s): N/A

*/

#include <iostream>
#include <map>
#include <stack>
#include "AdjacencyListGraph.h"
#include "HeapPriorityQueue.h"
#include "Entry.h"

using namespace std;

typedef map<Vertex *, int> Map;
typedef pair<Vertex *, int> MPair;
typedef map<Vertex *, Entry<int, Vertex *>> PQTokens;
typedef pair<Vertex *, Entry<int, Vertex *>> TPair;
typedef Entry<int, Vertex *> TEntry;

class comp
{
public:
    bool operator()(TEntry a, TEntry b)
    {
        return (a.key() < b.key());
    }
};

typedef HeapPriorityQueue<TEntry, comp> PQ;

void dijkstra(AdjacencyListGraph G, Vertex *src, Vertex *dest)
{
    Map D;
    Map cloud;
    PQ pq;
    PQTokens pqTokens;

    for (Vertex *v : G.getVertices())
    {
        if (v == src)
            D.insert(MPair(v, 0));
        else
            D.insert(MPair(v, INT_MAX));
        pqTokens.insert(TPair(v, pq.insert(TEntry(D[v], v))));
    }
    while (!pq.empty())
    {
        TEntry entry = pq.removeMin();
        int key = entry.key();
        Vertex *u = entry.value();
        cloud.insert(MPair(u, key));
        pqTokens.erase(u);
        for (Edge *e : G.outgoingEdges(u))
        {
            Vertex *v = G.opposite(u, e);
            if (cloud.find(v) == cloud.end())
            {
                int wgt = e->getElement();
                if (D[u] + wgt < D[v])
                {
                    D[v] = D[u] + wgt;
                    pq.replace(pqTokens[v], TEntry(D[v], v));
                }
            }
        }
    }

    bool findStart = false;
    bool findEnd = false;
    for (auto i : D)
    {
        if (i.first == src)
            findStart = true;
        if (i.first == dest)
            findEnd = true;
        if (findStart)
            if (findEnd)
            {
                cout << i.first->getElement() << "[" << i.second << "]";
                break;
            }
            else
                cout << i.first->getElement() << "[" << i.second << "] ->
";
    }
    cout << endl;
}

int main()
{
    AdjacencyListGraph G(true);

    Vertex *A = G.insertVertex("A");
    Vertex *B = G.insertVertex("B");
    Vertex *C = G.insertVertex("C");
    Vertex *D = G.insertVertex("D");
    Vertex *E = G.insertVertex("E");
    G.insertEdge(A, D, 5);
    G.insertEdge(A, E, 10);
    G.insertEdge(B, A, 3);
    G.insertEdge(B, C, 4);
    G.insertEdge(C, D, 2);
    G.insertEdge(D, E, 3);
    G.insertEdge(E, C, 6);
    cout << "Original Graph:\n";
    G.print();
    cout << endl;

    cout << "Path from B to E:\n";
    dijkstra(G, B, E);
    cout << endl;

    cout << "Modified by: Nero Li\n";

    return 0;
}
```

Input/output below:

```
Original Graph:
Vertex A
 [outgoing] 2 adjacencies:(D, 5)  (E, 10)
 [incoming] 1 adjacencies:(B, 3)
Vertex B
 [outgoing] 2 adjacencies:(A, 3)  (C, 4)
 [incoming] 0 adjacencies:
Vertex C
 [outgoing] 1 adjacencies:(D, 2)
 [incoming] 2 adjacencies:(B, 4)  (E, 6)
Vertex D
 [outgoing] 1 adjacencies:(E, 3)
 [incoming] 2 adjacencies:(A, 5)  (C, 2)
Vertex E
 [outgoing] 1 adjacencies:(C, 6)
 [incoming] 2 adjacencies:(A, 10)  (D, 3)

Path from B to E:
B[0] -> C[4] -> D[6] -> E[9]

Modified by: Nero Li
```

Answer for Question 1:

For a digraph, if for each vertex, there is no path for them to go back to itself, then we call that digraph DAG or directed acyclic graph.

One real-life DAG we can see is project management. The project schedule should be from top to bottom and have no way to go back since it would make the system of project management crash.

Answer for Question 2:

The transitive closure on a digraph shows all the possible vertices that each vertex in the digraph can reach.

The reason for using transitive closure is that we can know for two vertices a and b, there has or doesn't have a path. This can become an application for flight management where the transitive closure has become the base for the system to tell the user that he or she can find a flight to go from one place to another.

Extra Credit

Source code below:

extra_credit.cpp:

```
/*  Program: PA_10_extra_credit
    Author: Nero Li
    Class: CSCI 230
    Date: 05/17/2022
    Description:
        Implement Topological Ordering on a DAG using AdjacencyListGraph
        class from previous PA

    I certify that the code below is my own work.

      Exception(s): N/A

*/

#include <iostream>
#include <list>
#include <stack>
#include <map>
#include "AdjacencyListGraph.h"

using namespace std;

typedef pair<Vertex *, int> inCountPair;
```

```
void topological(AdjacencyListGraph G)
{
    vector<Vertex *> result;
    stack<Vertex *> ready;
    map<Vertex *, int> inCount;
    for (auto u : G.getVertices())
    {
        inCount.insert(inCountPair(u, G.inDegree(u)));
        if (inCount[u] == 0)
            ready.push(u);
    }
    while (!ready.empty())
    {
        Vertex *u = ready.top();
        ready.pop();
        result.push_back(u);
        for (auto e : G.outgoingEdges(u))
        {
            Vertex *v = G.opposite(u, e);
            --inCount[v];
            if (inCount[v] == 0)
                ready.push(v);
        }
    }

    for (auto i : result)
        cout << i->getElement() << " ";
    cout << endl;
}

int main()
{
    AdjacencyListGraph G(true);

    Vertex *A = G.insertVertex("A");
    Vertex *B = G.insertVertex("B");
    Vertex *C = G.insertVertex("C");
    Vertex *D = G.insertVertex("D");
    Vertex *E = G.insertVertex("E");
    G.insertEdge(A, D, 1);
    G.insertEdge(B, A, 1);
    G.insertEdge(B, C, 1);
    G.insertEdge(C, D, 1);
    G.insertEdge(D, E, 1);
    cout << "Original Graph:\n";
    G.print();
    cout << endl;

    cout << "Topological Ordering:\n";
    topological(G);
    cout << endl;
    cout << "Modified by: Nero Li\n";

    return 0;
}
```

Input/output below:

```
Original Graph:
Vertex A
 [outgoing] 1 adjacencies:(D, 1)
 [incoming] 1 adjacencies:(B, 1)
Vertex B
 [outgoing] 2 adjacencies:(A, 1)  (C, 1)
 [incoming] 0 adjacencies:
Vertex C
 [outgoing] 1 adjacencies:(D, 1)
 [incoming] 1 adjacencies:(B, 1)
Vertex D
 [outgoing] 1 adjacencies:(E, 1)
 [incoming] 2 adjacencies:(A, 1)  (C, 1)
Vertex E
 [outgoing] 0 adjacencies:
 [incoming] 1 adjacencies:(D, 1)

Topological Ordering:
B C A D E

Modified by: Nero Li
```