CSCI 230 PA # Submission

Due Date: <u>05/10/2022</u> Late (date and time):_____

Name(s): <u>Nero Li</u>

Exercise 1 (with extra credit) -- need to submit source code and I/O

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

Source code below:
Decorator.h:

```cpp
#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;
```

```cpp
        }
};

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:

```cpp
#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
```

```cpp
{
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;
};
```

AdjancyListGraph.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. */
```

```cpp
            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. */
```

```cpp
            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()
    {
        for (auto i : vertices)
            delete i;
        for (auto i : edges)
            delete i;

        vertices.clear();
        edges.clear();
    }

    /* 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;
```

```cpp
            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<pair<Vertex *, Edge *>> *mapPtr =
static_cast<InnerVertex *>(v)->getIncoming();
            auto it = mapPtr->begin();
            for (auto n : *mapPtr)
            {
                    if (n.first == static_cast<InnerVertex *>(v))
                            break;
                    it++;
            }

            if (it != mapPtr->end())
                    temp = it->second;
            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)
```

```cpp
                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.cpp:

```
/*  Program: PA_9_exercise_1
    Author: Nero Li
    Class: CSCI 230
    Date: 05/10/2022
    Description:
        Set up your own code to represent an undirected graph using
        Adjacency List. Try the test case below first and then create a
        simple graph with 4 vertices and 6 edges and print it.

    I certify that the code below is my own work.

        Exception(s): N/A

*/
```

```cpp
#include <iostream>
#include <string>
#include <vector>
#include "AdjacencyListGraph.h"

using namespace std;

class MatrixGraph
{
private:
    vector<vector<int>> Matrix;
public:
    MatrixGraph(int n)
    {
        for (int i = 0; i < n; ++i)
        {
            vector<int> cur;
            for (int j = 0; j < n; ++j)
                cur.push_back(0);
            Matrix.push_back(cur);
        }
    }

    void insert(char a, char b, int edge)
    {
        int i = a - 'A';
        int j = b - 'A';
        Matrix[i][j] = edge;
        Matrix[j][i] = edge;
    }

    void print()
    {
        int n = Matrix.size();

        for (int i = 0; i < n; ++i)
        {
            char c = 'A' + i;
            vector<pair<char, int>> adjacencies;
            for (int j = 0; j < n; ++j)
            {
                if (Matrix[i][j])
                {
                    pair<char, int> cur;
                    cur.first = 'A' + j;
                    cur.second = Matrix[i][j];
                    adjacencies.push_back(cur);
                }
            }
            cout << "Vertex " << c << endl;
            cout << " " << adjacencies.size() << " adjacencies:";
            for (auto a : adjacencies)
```

```cpp
                    cout << "(" << a.first << ", " << a.second << ")  ";
                cout << endl;
            }

        }
};

int main()
{
    AdjacencyListGraph test1;
    vector<Vertex *> v1;
    vector<Edge *> e1;
    v1.push_back(test1.insertVertex("A"));
    v1.push_back(test1.insertVertex("B"));
    v1.push_back(test1.insertVertex("C"));
    e1.push_back(test1.insertEdge(v1[1], v1[0], 100));
    e1.push_back(test1.insertEdge(v1[2], v1[0], 200));
    cout << "First graph by AdjacencyListGraph:\n";
    test1.print();
    cout << endl;

    AdjacencyListGraph test2;
    vector<Vertex *> v2;
    vector<Edge *> e2;
    v2.push_back(test2.insertVertex("A"));
    v2.push_back(test2.insertVertex("B"));
    v2.push_back(test2.insertVertex("C"));
    v2.push_back(test2.insertVertex("D"));
    e2.push_back(test2.insertEdge(v2[0], v2[1], 1));
    e2.push_back(test2.insertEdge(v2[1], v2[2], 2));
    e2.push_back(test2.insertEdge(v2[2], v2[3], 3));
    e2.push_back(test2.insertEdge(v2[3], v2[1], 4));
    e2.push_back(test2.insertEdge(v2[0], v2[2], 5));
    e2.push_back(test2.insertEdge(v2[1], v2[3], 6));
    cout << "Second graph by AdjacencyListGraph\n";
    test2.print();
    cout << endl;

    MatrixGraph test3(3);
    test3.insert('B','A', 100);
    test3.insert('C','A', 200);
    cout << "First graph by MatrixGraph:\n";
    test3.print();
    cout << endl;

    MatrixGraph test4(4);
    test4.insert('A','B', 1);
    test4.insert('B','C', 2);
    test4.insert('C','D', 3);
    test4.insert('D','A', 4);
    test4.insert('A','C', 5);
    test4.insert('B','D', 6);
```

```
    cout << "Second graph by MatrixGraph:\n";
    test4.print();
    cout << endl;

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

    return 0;
}
```

Input/output below:

```
First graph by AdjacencyListGraph:
Vertex A
 2 adjacencies:(B, 100)  (C, 200)
Vertex B
 1 adjacencies:(A, 100)
Vertex C
 1 adjacencies:(A, 200)

Second graph by AdjacencyListGraph
Vertex A
 2 adjacencies:(B, 1)  (C, 5)
Vertex B
 4 adjacencies:(A, 1)  (C, 2)  (D, 4)  (D, 6)
Vertex C
 3 adjacencies:(B, 2)  (D, 3)  (A, 5)
Vertex D
 3 adjacencies:(C, 3)  (B, 4)  (B, 6)

First graph by MatrixGraph:
Vertex A
 2 adjacencies:(B, 100)  (C, 200)
Vertex B
 1 adjacencies:(A, 100)
Vertex C
 1 adjacencies:(A, 200)

Second graph by MatrixGraph:
Vertex A
 3 adjacencies:(B, 1)  (C, 5)  (D, 4)
Vertex B
 3 adjacencies:(A, 1)  (C, 2)  (D, 6)
Vertex C
 3 adjacencies:(A, 5)  (B, 2)  (D, 3)
Vertex D
 3 adjacencies:(A, 4)  (B, 6)  (C, 3)

Modified by: Nero Li
```

Exercise 2 -- need to submit source code and I/O

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

Source code below:
DFS.h:

```
#pragma once
#include "Decorator.h"
#include "Graph.h"

using namespace std;

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

// Make sure that Vertex and Edge is-a Decorator

class DFS {                                             //
generic DFS
protected:                                              //
member data
    Graph *graph;                                       // the
graph
    Vertex *start;                                      //
start vertex
    Object *yes, *no;                                   // decorator
values

public:
    // member functions
    DFS(Graph *g);                                      //
constructor
    void initialize();                                  //
initialize a new DFS
    void dfsTraversal(Vertex *v);                       // recursive DFS
utility

    // overridden functions
    virtual void startVisit(Vertex *v) {}       // arrived at v

    // discovery edge e
    virtual void traverseDiscovery(Edge *e, Vertex *from) {}
    // back edge e
    virtual void traverseBack(Edge *e, Vertex *from) {}
    virtual void finishVisit(Vertex *v) {}              // finished
with v

    virtual bool isDone() const { return false; }     // finished?


    void visit(Vertex *v) { v->set("visited", yes); }
    void visit(Edge *e) { e->set("visited", yes); }
    void unvisit(Vertex *v) { v->set("visited", no); }
    void unvisit(Edge *e) { e->set("visited", no); }
    bool isVisited(Vertex *v) { return v->get("visited") == yes; }
```

```
        bool isVisited(Edge *e) { return e->get("visited") == yes; }
};

DFS::DFS(Graph *g) : graph(g), yes(new Object), no(new Object) {}

void DFS::initialize() {
        list<Vertex *> verts = graph->getVertices();
        for (auto pv = verts.begin(); pv != verts.end(); ++pv)
                unvisit(*pv);                                          //
mark vertices unvisited

        list<Edge *> edges = graph->getEdges();
        for (auto pe = edges.begin(); pe != edges.end(); ++pe)
                unvisit(*pe);                                         //
mark edges unvisited
}

void DFS::dfsTraversal(Vertex *v) {
        startVisit(v);  visit(v);                                // visit v
and mark visited
        cout << v->getElement() << " ";
        vector<Edge *> incident = graph->outgoingEdges(v);
        auto pe = incident.begin();
        while (!isDone() && pe != incident.end()) { // visit v's incident
edges
                Edge *e = *pe++;
                if (!isVisited(e)) {                              // discovery
edge?
                        visit(e);                                        //
mark it visited
                        Vertex *w = graph->opposite(v, e);   // get opposing
vertex
                        if (!isVisited(w)) {                          //
unexplored?
                                cout << e->getElement() << " ";
                                traverseDiscovery(e, v);      // let's discover
it
                                if (!isDone()) dfsTraversal(w);// continue
traversal
                        }
                        else traverseBack(e, v);              // process back
edge
                }
        }
        if (!isDone()) finishVisit(v);                        // finished with v
}
```

exercise_2.cpp:

```
/*  Program: PA_9_exercise_2
    Author: Nero Li
    Class: CSCI 230
```

Date: 05/10/2022
        Description:
            Implement either DFS or BFS using your graph class from exercise
            1. You can set up DFS(G, v) like the book (perform DFS on a
            graph) or G.DFS(v) where DFS() is a member of Graph class (use
            similar set up for BFS). For C++, you can use Decorator.h and
            DFS.h and use GaphAlgorithms.java for Java. Print out the
            vertices and discovery/forward edges in the order that they were
            visited (should be vertex, discovery edge, vertex, etc.). Try
            the following graph and start out with vertex A.

        I certify that the code below is my own work.

            Exception(s): N/A

*/

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

using namespace std;

int main()
{
    AdjacencyListGraph g;
    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");
    Edge *e1 = g.insertEdge(A, B, 1);
    Edge *e2 = g.insertEdge(B, C, 2);
    Edge *e3 = g.insertEdge(C, D, 3);
    Edge *e4 = g.insertEdge(D, E, 4);
    Edge *e5 = g.insertEdge(A, D, 5);
    cout << "Current graph:\n";
    g.print();

    DFS dfs(&g);
    cout << "DFS Traversal start from A:\n";
    dfs.dfsTraversal(A);
    cout << endl;

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

    return 0;
}

Input/output below:

```
Current graph:
Vertex A
 2 adjacencies:(B, 1)  (D, 5)
Vertex B
 2 adjacencies:(A, 1)  (C, 2)
Vertex C
 2 adjacencies:(B, 2)  (D, 3)
Vertex D
 3 adjacencies:(C, 3)  (E, 4)  (A, 5)
Vertex E
 1 adjacencies:(D, 4)
DFS Traversal start from A:
A 1 B 2 C 3 D 4 E
Modified by: Nero Li
```

Answer for Question 1:

For an undirected graph, one adjacency will represent one line with two vertices. Because of that, based on the list of the adjacency information, we will know the way we connect each vertex in a imaginary shape of graph with all the lines. In other words, the biggest advantage for adjacency list is that it show the shape of graph straightforward. Furthermore, this is also all the legal path for one vertex to do traversal to another vertex, so it is also important for BFS and DFS to know the traversal steps.

Answer for Question 2:

Unless requirement specified a rule to set the prior traversal vertex, DFS and BFS does not guarantee to visit the vertices in a certain order since for each vertex, how many unvisited adjacencies it can choose means how many types of traversal will be shown up.