Trạng thái	Đã xong
Bắt đầu vào lúc	Thứ Tư, 23 tháng 4 2025, 3:42 PM
Kết thúc lúc	Thứ Tư, 23 tháng 4 2025, 3:52 PM
Thời gian thực hiện	9 phút 42 giây
Điểm	**10,00** trên 10,00 (**100**%)

Câu hỏi 1

Đúng

Đạt điểm 1,00 trên 1,00

Template class `DGraph` representing a directed graph with type T with the initialized frame. It has attribute `VertexNode* nodeList`, which is the head of a **singly linked list**, representing a **list of vertex** of this graph.
This class inlcudes 2 classes: `VertexNode` and `Edge`.
- Class `VertexNode` representing a vertex in graph. It has some attributes:
  + `T vertex`: the vertex's value.
  + `Edge* adList`: a **singly linked list** representing the **adjacent edges** that have **this vertex as their starting vertex (from)**.
- Class `Edge` representing an edge in graph. It has some attributes:
  + `VertexNode* fromNode` - `VertexNode* toNode`: represents the starting vertex (from) and ending vertex (to) of this edge.
  + `float weight`: edge's weight.

Requirements: In class **VertexNode**, implement methods **getEdge**, **connectTo**, **addAdjacentEdge**, and **removeTo**. Descriptions for each method are provided below. Ensure that all four methods are fully implemented before checking.

```
template <class T>
class DGraph {
public:
   class VertexNode; // Forward declaration
   class Edge; // Forward declaration
protected:
   VertexNode* nodeList; //list of vertexNode of DGraph
   int countVertex;
   int countEdge;
public:
   DGraph() {
      this->nodeList = nullptr;
      this->countEdge = 0;
      this->countVertex = 0;
   }
   ~DGraph() {};
   VertexNode* getVertexNode(T vertex);
   void add(T vertex);

   void connect(T from, T to, float weight=0);

   void removeVertex(T removeVertex);
   bool removeEdge(T from, T to);
   string shape();
   bool empty();
   void clear();
   void printGraph();

public:
   class VertexNode {
   private:
      T vertex;
      Edge* adList; //list of adjacent edge of this vertex
      VertexNode* next;

      friend class Edge;
      friend class DGraph;
   public:

      VertexNode(T vertex, Edge* adList = nullptr, VertexNode* next = nullptr) {
         this->vertex = vertex;
         this->adList = adList;
         this->next = next;
```

```
    }
    string toString();
    void addAdjacentEdge(Edge* newEdge);
    bool connectTo(VertexNode* toNode, float weight = 0);
    bool removeTo(VertexNode* toNode);
    Edge* getEdge(VertexNode* toNode);
  };

  class Edge {
  private:
    VertexNode* fromNode;
    VertexNode* toNode;
    float weight;
    Edge* next;

    friend class VertexNode;
    friend class DGraph;
  public:
    Edge(VertexNode* fromNode, VertexNode* toNode, float weight = 0.0, Edge* next = nullptr) {
      this->fromNode = fromNode;
      this->toNode = toNode;
      this->weight = weight;
      this->next = next;
    }
    string toString();

  };
};
```

For example:

Test	Result

`DGraph<int>::VertexNode* node0 = new DGraph<int>::VertexNode(0);`
`DGraph<int>::VertexNode* node1 = new DGraph<int>::VertexNode(1);`
`DGraph<int>::VertexNode* node2 = new DGraph<int>::VertexNode(2);`
`DGraph<int>::VertexNode* node3 = new DGraph<int>::VertexNode(3);`
 	
`node0->connectTo(node1, 12.3);`
`node0->connectTo(node2, 13.3);`
`node0->connectTo(node3, 14);`
`node1->connectTo(node3, 176);`
 	
`cout << node0->getEdge(node1)->toString() << endl;`
`cout << node1->getEdge(node3)->toString() << endl;`
 	
`delete node0;`
`delete node1;`
`delete node2;`
`delete node3;`
	`E(0,1,12.3)`
	`E(1,3,176)`

Answer: (penalty regime: 0 %)

Reset answer

```
1  // Tìm kiếm cạnh từ đỉnh hiện tại đến đỉnh toNode
2  /**
3       * @brief Tìm kiếm cạnh từ đỉnh hiện tại đến đỉnh đích
4       * @param toNode Con trỏ đến đỉnh đích cần tìm
5       * @return Con trỏ đến cạnh nếu tìm thấy, nullptr nếu không tồn tại
6       * @complexity O(n) với n là số cạnh kề của đỉnh hiện tại
7       *
8       * Thuật toán:
9       * - Duyệt tuần tự qua danh sách kề (adjacency list)
10      * - Kiểm tra đỉnh đích của mỗi cạnh
11      * - Trả về cạnh đầu tiên thỏa điều kiện hoặc nullptr nếu không tìm thấy
12 */
13 template<class T>
14 typename DGraph<T>::Edge* DGraph<T>::VertexNode::getEdge(VertexNode* toNode) {
15     // Duyệt qua danh sách kề của đỉnh hiện tại để tìm cạnh
16
```

```
    Edge* current = this->adList;

    while (current != nullptr) {
        // Kiểm tra xem cạnh hiện tại có đến đỉnh toNode không
        if (current->toNode == toNode) {
            return current;             // Trả về cạnh nếu tìm thấy
        }
        current = current->next;    // Chuyển sang cạnh tiếp theo trong danh sách
    }
    // Nếu không tìm thấy cạnh nào, trả về nullptr
    return nullptr;
}

// Thêm một cạnh mới vào danh sách kề của đỉnh hiện tại
 /**
     * @brief Thêm một cạnh mới vào danh sách kề của đỉnh hiện tại
     * @param newEdge Con trỏ đến cạnh mới cần thêm vào
     * @complexity O(1) - thao tác chèn đầu danh sách
     *
     * Thuật toán:
     * - Sử dụng phương pháp chèn đầu cho danh sách liên kết đơn
     * - Xử lý riêng trường hợp danh sách rỗng
     * - Không kiểm tra tính duy nhất của cạnh (được xử lý ở connectTo)
*/
template<class T>
void DGraph<T>::VertexNode::addAdjacentEdge(Edge* newEdge) {
    // Sử dụng phương pháp thêm vào đầu danh sách liên kết đơn
    if (this->adList == nullptr) {
        // Nếu danh sách kề rỗng, đặt newEdge làm phần tử đầu tiên
        this->adList = newEdge;
    } else {
        // Ngược lại, chèn newEdge vào đầu danh sách
        newEdge->next = this->adList;
        this->adList = newEdge;
    }
}
```

	Test	Expected	Got	

✓	`DGraph<int>::VertexNode* node0 = new DGraph<int>::VertexNode(0);`
	`DGraph<int>::VertexNode* node1 = new DGraph<int>::VertexNode(1);`
	`DGraph<int>::VertexNode* node2 = new DGraph<int>::VertexNode(2);`
	`DGraph<int>::VertexNode* node3 = new DGraph<int>::VertexNode(3);`
	
	`node0->connectTo(node1, 12.3);`
	`node0->connectTo(node2, 13.3);`
	`node0->connectTo(node3, 14);`
	`node1->connectTo(node3, 176);`
	
	`cout << node0->getEdge(node1)->toString() << endl;`
	`cout << node1->getEdge(node3)->toString() << endl;`
	
	`delete node0;`
	`delete node1;`
	`delete node2;`
	`delete node3;`
		E(0,1,12.3)
		E(1,3,176)
			E(0,1,12.3)
			E(1,3,176)
				✓

	Test	Expected	Got	

✓	`DGraph<char>::VertexNode* nodeA = new DGraph<char>::VertexNode('A');`
	`DGraph<char>::VertexNode* nodeB = new DGraph<char>::VertexNode('B');`
	`DGraph<char>::VertexNode* nodeC = new DGraph<char>::VertexNode('C');`
	`DGraph<char>::VertexNode* nodeD = new DGraph<char>::VertexNode('D');`
	`DGraph<char>::VertexNode* nodeE = new DGraph<char>::VertexNode('E');`
	
	`nodeA->connectTo(nodeB);`
	`nodeA->connectTo(nodeC, 29.4);`
	`nodeA->connectTo(nodeD, 30.4);`
	`nodeB->connectTo(nodeD, 19.75);`
	`nodeC->connectTo(nodeC, 0.54);`
	`nodeE->connectTo(nodeA);`
	
	`cout << (nodeE->getEdge(nodeA) ? nodeE->getEdge(nodeA)->toString() :`
	`"Edge doesn't exist!") << endl;`
	`cout << (nodeC->getEdge(nodeC) ? nodeC->getEdge(nodeC)->toString() :`
	`"Edge doesn't exist!") << endl;`
	`nodeE->removeTo(nodeA);`
	`nodeC->removeTo(nodeC);`
	
	`cout << (nodeA->getEdge(nodeB) ? nodeA->getEdge(nodeB)->toString() :`
	`"Edge doesn't exist!") << endl;`
	`cout << (nodeA->getEdge(nodeC) ? nodeA->getEdge(nodeC)->toString() :`
	`"Edge doesn't exist!") << endl;`
	`cout << (nodeE->getEdge(nodeA) ? nodeE->getEdge(nodeA)->toString() :`
	`"Edge doesn't exist!") << endl;`
	`cout << (nodeC->getEdge(nodeC) ? nodeC->getEdge(nodeC)->toString() :`
	`"Edge doesn't exist!") << endl;`
	
	`delete nodeA;`
	`delete nodeB;`
	`delete nodeC;`
	`delete nodeD;`
	`delete nodeE;`
		`E(E,A,0)`
		`E(C,C,0.54)`
		`E(A,B,0)`
		`E(A,C,29.4)`
		`Edge doesn't exist!`
		`Edge doesn't exist!`
			`E(E,A,0)`
			`E(C,C,0.54)`
			`E(A,B,0)`
			`E(A,C,29.4)`
			`Edge doesn't exist!`
			`Edge doesn't exist!`
				✓

Passed all tests! ✓

Đúng

Marks for this submission: 1,00/1,00.

Câu hỏi **2**

Đúng

Đạt điểm 1,00 trên 1,00

Template class `DGraph` representing a directed graph with type T with the initialized frame. It has attribute `VertexNode* nodeList`, which is the head of a **singly linked list**, representing **list of vertex** of this graph.
This class inlcudes 2 classes: `VertexNode` and `Edge`.
- Class `VertexNode` representing a vertex in graph. It has some attributes:
  + `T vertex`: the vertex's value.
  + `Edge* adList`: a **singly linked list** representing the **adjacent edges** that have **this vertex as their starting vertex (from)**.
- Class `Edge` representing an edge in graph. It has some attributes:
  + `VertexNode* fromNode` - `VertexNode* toNode`: represents the starting vertex (from) and ending vertex (to) of this edge.
  + `float weight`: edge's weight.

Requirements: In class **DGraph**, implement methods **getVertexNode**, **add** and **connect**. Descriptions for each method are provided below. Ensure that all three methods are fully implemented before checking.

Notes: You can use the methods from the previous exercises without needing to implement them again.

```
template <class T>
class DGraph {
public:
   class VertexNode; // Forward declaration
   class Edge; // Forward declaration
protected:
   VertexNode* nodeList; //list of vertexNode of DGraph
   int countVertex;
   int countEdge;
public:
   DGraph() {
      this->nodeList = nullptr;
      this->countEdge = 0;
      this->countVertex = 0;
   }
   ~DGraph() {};
   VertexNode* getVertexNode(T vertex);
   void add(T vertex);

   void connect(T from, T to, float weight=0);

   void removeVertex(T removeVertex);
   bool removeEdge(T from, T to);
   string shape();
   bool empty();
   void clear();
   void printGraph();

public:
   class VertexNode {
   private:
      T vertex;
      Edge* adList; //list of adjacent edge of this vertex
      VertexNode* next;

      friend class Edge;
      friend class DGraph;
   public:

      VertexNode(T vertex, Edge* adList = nullptr, VertexNode* next = nullptr) {
         this->vertex = vertex;
         this->adList = adList;
```

```
        this->next = next;
      }
      string toString();
      void addAdjacentEdge(Edge* newEdge);
      bool connectTo(VertexNode* toNode, float weight = 0);
      bool removeTo(VertexNode* toNode);
      Edge* getEdge(VertexNode* toNode);
   };

   class Edge {
   private:
      VertexNode* fromNode;
      VertexNode* toNode;
      float weight;
      Edge* next;

      friend class VertexNode;
      friend class DGraph;
   public:
      Edge(VertexNode* fromNode, VertexNode* toNode, float weight = 0.0, Edge* next = nullptr) {
         this->fromNode = fromNode;
         this->toNode = toNode;
         this->weight = weight;
         this->next = next;
      }
      string toString();

   };
};
```

For example:

Test	Result

DGraph<int> graph;
for(int i = 0; i < 6; i++) graph.add(i);
graph.printGraph();
	==========
	Number of vertices: 6
	V(0)
	V(1)
	V(2)
	V(3)
	V(4)
	V(5)
	---------- 	
	Number of edges: 0
	==========

DGraph<int> graph;
for(int i = 0; i < 6; i++) graph.add(i);
 	
graph.connect(1, 2, 40);
graph.connect(1, 3, 6.9);
graph.connect(4, 5, 27);
graph.connect(3, 2, 2.1);
graph.connect(0, 2, 11.2);
graph.connect(0, 5, 67);
graph.connect(2, 1, 19.75);
 	
graph.printGraph();
	==========
	Number of vertices: 6
	V(0)
	V(1)
	V(2)
	V(3)
	V(4)
	V(5)
	---------- 	
	Number of edges: 7
	E(0,2,11.2)
	E(0,5,67)
	E(1,2,40)
	E(1,3,6.9)
	E(2,1,19.75)
	E(3,2,2.1)
	E(4,5,27)
	==========

Answer: (penalty regime: 0 %)

Reset answer

```
// Triển khai các phương thức của lớp template DGraph
// Phương thức tìm kiếm đỉnh trong đồ thị
template<class T>
typename DGraph<T>::VertexNode* DGraph<T>::getVertexNode(T vertex) {
    // Duyệt qua danh sách liên kết các đỉnh bắt đầu từ nodeList
    VertexNode* current = nodeList;
    while (current != nullptr) {
        // Kiểm tra nếu đỉnh hiện tại chứa giá trị vertex cần tìm
        if (current->vertex == vertex) {
            return current;  // Trả về con trỏ đến đỉnh nếu tìm thấy
        }
        // Di chuyển đến đỉnh tiếp theo trong danh sách
        current = current->next;
    }
    return nullptr;  // Trả về nullptr nếu không tìm thấy vertex
}
// Phương thức thêm một đỉnh mới vào đồ thị
template<class T>
void DGraph<T>::add(T vertex) {
    // Tạo một đỉnh mới với giá trị vertex
    VertexNode* newNode = new VertexNode(vertex);
    // Thêm đỉnh mới vào cuối danh sách các đỉnh
    if (nodeList == nullptr) {
        // Nếu danh sách rỗng, đỉnh mới trở thành đỉnh đầu tiên
        nodeList = newNode;
    } else {
        // Tìm đỉnh cuối cùng trong danh sách
        VertexNode* current = nodeList;
        while (current->next != nullptr) {
            current = current->next;
        }
        // Thêm đỉnh mới vào sau đỉnh cuối cùng
        current->next = newNode;
    }
    // Tăng số lượng đỉnh
    countVertex++;
}
// Phương thức tạo kết nối từ đỉnh "from" đến đỉnh "to" với trọng số weight
template <class T>
void DGraph<T>::connect(T from, T to, float weight) {
    // Tìm đỉnh chứa giá trị "from" và đỉnh chứa giá trị "to"
    VertexNode* fromNode = getVertexNode(from);
    VertexNode* toNode = getVertexNode(to);
    // Nếu một trong hai đỉnh không tồn tại, ném ngoại lệ
    if (fromNode == nullptr || toNode == nullptr) {
        throw VertexNotFoundException("Vertex doesn't exist!"); }
    // Kết nối đỉnh "from" đến đỉnh "to"
    bool isNewEdge = fromNode->connectTo(toNode, weight);
    // Nếu tạo cạnh mới, tăng biến đếm số cạnh
    if (isNewEdge) {
        countEdge++;  } }
```

	Test	Expected	Got

✓	DGraph<int> graph;
	for(int i = 0; i < 6; i++)
	graph.add(i);
	graph.printGraph();
		==========
		Number of vertices: 6
		V(0)
		V(1)
		V(2)
		V(3)
		V(4)
		V(5)
		---------- 	
		Number of edges: 0
		==========
			==========
			Number of vertices: 6
			V(0)
			V(1)
			V(2)
			V(3)
			V(4)
			V(5)
			---------- 	
			Number of edges: 0
			==========

	Test	Expected	Got

✓	DGraph<int> graph;
	for(int i = 0; i < 6; i++)
	graph.add(i);
	
	graph.connect(1, 2, 40);
	graph.connect(1, 3, 6.9);
	graph.connect(4, 5, 27);
	graph.connect(3, 2, 2.1);
	graph.connect(0, 2, 11.2);
	graph.connect(0, 5, 67);
	graph.connect(2, 1, 19.75);
	
	graph.printGraph();
		==========
		Number of vertices: 6
		V(0)
		V(1)
		V(2)
		V(3)
		V(4)
		V(5)
		---------- 	
		Number of edges: 7
		E(0,2,11.2)
		E(0,5,67)
		E(1,2,40)
		E(1,3,6.9)
		E(2,1,19.75)
		E(3,2,2.1)
		E(4,5,27)
		==========
			==========
			Number of vertices: 6
			V(0)
			V(1)
			V(2)
			V(3)
			V(4)
			V(5)
			---------- 	
			Number of edges: 7
			E(0,2,11.2)
			E(0,5,67)
			E(1,2,40)
			E(1,3,6.9)
			E(2,1,19.75)
			E(3,2,2.1)
			E(4,5,27)
			==========

Passed all tests! ✓

Đúng

Marks for this submission: 1,00/1,00.

Câu hỏi **3**

Đúng

Đạt điểm 1,00 trên 1,00

Template class `DGraph` representing a directed graph with type T with the initialized frame. It has attribute `VertexNode* nodeList`, which is the head of a **singly linked list**, representing **list of vertex** of this graph.
This class inlcudes 2 classes: `VertexNode` and `Edge`.
- Class `VertexNode` representing a vertex in graph. It has some attributes:
  + `T vertex`: the vertex's value.
  + `Edge* adList`: a **singly linked list** representing the **adjacent edges** that have **this vertex as their starting vertex (from)**.
- Class `Edge` representing an edge in graph. It has some attributes:
  + `VertexNode* fromNode` - `VertexNode* toNode`: represents the starting vertex (from) and ending vertex (to) of this edge.
  + `float weight`: edge's weight.

Requirements: Implement methods **removeEdge** and **removeVertex**. Descriptions for each method are provided below.

Notes:

- The `removeTo` method is used to delete an edge that ends at the vertex "toNode" from the adjacency list of the current vertex. Students should use this method when implementing **removeEdge** and **removeVertex.**

- You can use the methods from the previous exercises without needing to implement them again.

```
template <class T>
class DGraph {
public:
   class VertexNode; // Forward declaration
   class Edge; // Forward declaration
protected:
   VertexNode* nodeList; //list of vertexNode of DGraph
   int countVertex;
   int countEdge;
public:
   DGraph() {
      this->nodeList = nullptr;
      this->countEdge = 0;
      this->countVertex = 0;
   }
   ~DGraph() {};
   VertexNode* getVertexNode(T vertex);
   void add(T vertex);

   void connect(T from, T to, float weight=0);

   void removeVertex(T removeVertex);
   bool removeEdge(T from, T to);
   string shape();
   bool empty();
   void clear();
   void printGraph();

public:
   class VertexNode {
   private:
      T vertex;
      Edge* adList; //list of adjacent edge of this vertex
      VertexNode* next;

      friend class Edge;
      friend class DGraph;
   public:
```

```
        VertexNode(T vertex, Edge* adList = nullptr, VertexNode* next = nullptr) {
            this->vertex = vertex;
            this->adList = adList;
            this->next = next;
        }
        string toString();
        void addAdjacentEdge(Edge* newEdge);
        bool connectTo(VertexNode* toNode, float weight = 0);
        bool removeTo(VertexNode* toNode);
        Edge* getEdge(VertexNode* toNode);
    };

    class Edge {
    private:
        VertexNode* fromNode;
        VertexNode* toNode;
        float weight;
        Edge* next;

        friend class VertexNode;
        friend class DGraph;
    public:
        Edge(VertexNode* fromNode, VertexNode* toNode, float weight = 0.0, Edge* next = nullptr) {
            this->fromNode = fromNode;
            this->toNode = toNode;
            this->weight = weight;
            this->next = next;
        }
        string toString();

    };
};
```

For example:

Test	Result

DGraph<int> graph;
 	
for(int i = 0; i < 6; i++) graph.add(i);
 	
graph.connect(1, 2, 40);
graph.connect(1, 3, 6.9);
graph.connect(4, 5, 27);
graph.connect(3, 2, 2.1);
graph.connect(1, 2, 11.2);
graph.connect(1, 3, 67);
 	
graph.removeEdge(1, 2);
graph.removeEdge(4, 5);
graph.removeEdge(1, 3);
 	
graph.printGraph();
	==========
	Number of vertices: 6
	V(0)
	V(1)
	V(2)
	V(3)
	V(4)
	V(5)
	---------- 	
	Number of edges: 1
	E(3,2,2.1)
	==========

DGraph<int> graph;
 	
for(int i = 0; i < 6; i++) graph.add(i);
 	
graph.connect(1, 2, 40);
graph.connect(1, 3, 6.9);
graph.connect(4, 5, 27);
graph.connect(3, 2, 2.1);
graph.connect(1, 2, 11.2);
 	
graph.removeVertex(2);
 	
graph.printGraph();
	==========
	Number of vertices: 5
	V(0)
	V(1)
	V(3)
	V(4)
	V(5)
	---------- 	
	Number of edges: 2
	E(1,3,6.9)
	E(4,5,27)
	==========

Answer: (penalty regime: 0 %)

Reset answer

```cpp
// Triển khai phương thức xóa cạnh trong đồ thị
template <class T>
bool DGraph<T>::removeEdge(T from, T to) {
    // Bước 1: Tìm đỉnh chứa giá trị "from"
    VertexNode* fromNode = getVertexNode(from);

    // Bước 2: Tìm đỉnh chứa giá trị "to"
    VertexNode* toNode = getVertexNode(to);

    // Bước 3: Kiểm tra sự tồn tại của cả hai đỉnh
    if (fromNode == nullptr || toNode == nullptr) {
        // Nếu ít nhất một đỉnh không tồn tại, ném ngoại lệ
        throw VertexNotFoundException("Vertex doesn't exist!");
    }

    // Bước 4: Xóa cạnh từ fromNode đến toNode sử dụng phương thức removeTo
    bool success = fromNode->removeTo(toNode);

    // Bước 5: Xử lý kết quả sau khi xóa
    if (success) {
        // Nếu xóa thành công, giảm số lượng cạnh và trả về true
        countEdge--;
        return true;
    } else {
        // Nếu không tìm thấy cạnh để xóa, trả về false
        return false;
    }
}

// Triển khai phương thức xóa đỉnh trong đồ thị
template <class T>
void DGraph<T>::removeVertex(T removeVertex) {
    // Bước 1: Tìm đỉnh cần xóa
    VertexNode* removeNode = getVertexNode(removeVertex);

    // Bước 2: Kiểm tra sự tồn tại của đỉnh cần xóa
    if (removeNode == nullptr) {
        // Nếu đỉnh không tồn tại, ném ngoại lệ
        throw VertexNotFoundException("Vertex doesn't exist!");
    }

    // Bước 3: Xóa tất cả các cạnh đi đến đỉnh cần xóa
    VertexNode* current = nodeList;
    while (current != nullptr) {
        // Bỏ qua nếu current chính là đỉnh cần xóa
        if (current != removeNode) {
            // Thử xóa cạnh từ current đến removeNode
            if (current->removeTo(removeNode)) {
                // Nếu xóa thành công, giảm số lượng cạnh
                countEdge--;
            }
        }
```

	Test	Expected	Got

✓	DGraph<int> graph;
	
	for(int i = 0; i < 6; i++) graph.add(i);
	
	graph.connect(1, 2, 40);
	graph.connect(1, 3, 6.9);
	graph.connect(4, 5, 27);
	graph.connect(3, 2, 2.1);
	graph.connect(1, 2, 11.2);
	graph.connect(1, 3, 67);
	
	graph.removeEdge(1, 2);
	graph.removeEdge(4, 5);
	graph.removeEdge(1, 3);
	
	graph.printGraph();
		==========
		Number of vertices: 6
		V(0)
		V(1)
		V(2)
		V(3)
		V(4)
		V(5)
		---------- 	
		Number of edges: 1
		E(3,2,2.1)
		==========
			==========
			Number of vertices: 6
			V(0)
			V(1)
			V(2)
			V(3)
			V(4)
			V(5)
			---------- 	
			Number of edges: 1
			E(3,2,2.1)
			==========

✓	DGraph<int> graph;
	
	for(int i = 0; i < 6; i++) graph.add(i);
	
	graph.connect(1, 2, 40);
	graph.connect(1, 3, 6.9);
	graph.connect(4, 5, 27);
	graph.connect(3, 2, 2.1);
	graph.connect(1, 2, 11.2);
	
	graph.removeVertex(2);
	
	graph.printGraph();
		==========
		Number of vertices: 5
		V(0)
		V(1)
		V(3)
		V(4)
		V(5)
		---------- 	
		Number of edges: 2
		E(1,3,6.9)
		E(4,5,27)
		==========
			==========
			Number of vertices: 5
			V(0)
			V(1)
			V(3)
			V(4)
			V(5)
			---------- 	
			Number of edges: 2
			E(1,3,6.9)
			E(4,5,27)
			==========

Passed all tests! ✓

Đúng

Marks for this submission: 1,00/1,00.

Câu hỏi **4**

Đúng

Đạt điểm 1,00 trên 1,00

Template class `DGraph` representing a directed graph with type T with the initialized frame. It has attribute `VertexNode* nodeList`, which is the head of a **singly linked list**, representing **list of vertex** of this graph.
This class inlcudes 2 classes: `VertexNode` and `Edge`.
- Class `VertexNode` representing a vertex in graph. It has some attributes:
  + `T vertex`: the vertex's value.
  + `Edge* adList`: a **singly linked list** representing the **adjacent edges** that have **this vertex as their starting vertex (from)**.
- Class `Edge` representing an edge in graph. It has some attributes:
  + `VertexNode* fromNode` - `VertexNode* toNode`: represents the starting vertex (from) and ending vertex (to) of this edge.
  + `float weight`: edge's weight.

Requirements: Implement methods **shape, empty** and **clear**. Descriptions for each method are provided below.

Notes: You can use the methods from the previous exercises without needing to implement them again.

```
template <class T>
class DGraph {
public:
   class VertexNode; // Forward declaration
   class Edge; // Forward declaration
protected:
   VertexNode* nodeList; //list of vertexNode of DGraph
   int countVertex;
   int countEdge;
public:
   DGraph() {
      this->nodeList = nullptr;
      this->countEdge = 0;
      this->countVertex = 0;
   }
   ~DGraph() {};
   VertexNode* getVertexNode(T vertex);
   void add(T vertex);

   void connect(T from, T to, float weight=0);

   void removeVertex(T removeVertex);
   bool removeEdge(T from, T to);
   string shape();
   bool empty();
   void clear();
   void printGraph();

public:
   class VertexNode {
   private:
      T vertex;
      Edge* adList; //list of adjacent edge of this vertex
      VertexNode* next;

      friend class Edge;
      friend class DGraph;
   public:

      VertexNode(T vertex, Edge* adList = nullptr, VertexNode* next = nullptr) {
         this->vertex = vertex;
         this->adList = adList;
         this->next = next;
```

```
    }
    string toString();
    void addAdjacentEdge(Edge* newEdge);
    bool connectTo(VertexNode* toNode, float weight = 0);
    bool removeTo(VertexNode* toNode);
    Edge* getEdge(VertexNode* toNode);
  };

  class Edge {
  private:
    VertexNode* fromNode;
    VertexNode* toNode;
    float weight;
    Edge* next;

    friend class VertexNode;
    friend class DGraph;
  public:
    Edge(VertexNode* fromNode, VertexNode* toNode, float weight = 0.0, Edge* next = nullptr) {
      this->fromNode = fromNode;
      this->toNode = toNode;
      this->weight = weight;
      this->next = next;
    }
    string toString();

  };
};
```

For example:

Test	Result

DGraph<int> graph;
 	
for(int i = 0; i < 6; i++) graph.add(i);
 	
graph.connect(1, 2, 40);
graph.connect(1, 3, 6.9);
graph.connect(4, 5, 27);
graph.connect(3, 2, 2.1);
graph.connect(1, 2, 11.2);
graph.connect(1, 3, 67);
 	
graph.removeEdge(1, 2);
graph.removeEdge(4, 5);
 	
cout << graph.shape() << endl;
	[Vertices: 6, Edges: 2]

DGraph<int> graph;
 	
for(int i = 0; i < 6; i++) graph.add(i);
 	
graph.connect(1, 2, 40);
graph.connect(1, 3, 6.9);
graph.connect(4, 5, 27);
graph.connect(3, 2, 2.1);
graph.connect(1, 2, 11.2);
graph.connect(1, 3, 67);
 	
graph.clear();
cout << graph.shape() << endl;
cout << (graph.empty() ? "Graph is empty!" : "Graph is not empty!") << endl;
	[Vertices: 0, Edges: 0]
	Graph is empty!

Answer: (penalty regime: 0 %)

Reset answer

```
1 // Triển khai phương thức shape() - trả về thông tin hình dạng của đồ thị
2
```

```cpp
template<class T>
string DGraph<T>::shape() {
    // Trả về chuỗi có định dạng: [Vertices: <numOfVertex>, Edges: <numOfEdge>]
    return "[Vertices: " + to_string(countVertex) + ", Edges: " + to_string(countEdge) + "]";
}

// Triển khai phương thức empty() - kiểm tra đồ thị có rỗng hay không
template<class T>
bool DGraph<T>::empty() {
    // Đồ thị rỗng khi không có đỉnh nào (countVertex = 0)
    // Lưu ý: Nếu không có đỉnh nào thì cũng không thể có cạnh nào
    return countVertex == 0;
}

// Triển khai phương thức clear() - xóa tất cả đỉnh và cạnh của đồ thị
template<class T>
void DGraph<T>::clear() {
    // Duyệt qua tất cả các đỉnh trong đồ thị
    VertexNode* current = nodeList;

    while (current != nullptr) {
        // Lưu con trỏ đến đỉnh tiếp theo trước khi xóa đỉnh hiện tại
        VertexNode* nextNode = current->next;

        // Xóa tất cả các cạnh của đỉnh hiện tại
        Edge* currentEdge = current->adList;
        while (currentEdge != nullptr) {
            Edge* nextEdge = currentEdge->next;
            delete currentEdge; // Giải phóng bộ nhớ của cạnh
            currentEdge = nextEdge;
        }

        // Giải phóng bộ nhớ của đỉnh hiện tại
        delete current;

        // Chuyển sang đỉnh tiếp theo
        current = nextNode;
    }

    // Đặt lại các biến thành trạng thái ban đầu
    nodeList = nullptr;
    countVertex = 0;
    countEdge = 0;
}
```

	Test	Expected	Got	

✓	DGraph<int> graph;
	
	for(int i = 0; i < 6; i++) graph.add(i);
	
	graph.connect(1, 2, 40);
	graph.connect(1, 3, 6.9);
	graph.connect(4, 5, 27);
	graph.connect(3, 2, 2.1);
	graph.connect(1, 2, 11.2);
	graph.connect(1, 3, 67);
	
	graph.removeEdge(1, 2);
	graph.removeEdge(4, 5);
	
	cout << graph.shape() << endl;
		[Vertices: 6, Edges: 2]	[Vertices: 6, Edges: 2]	✓

	Test	Expected	Got	

✓	`DGraph<int> graph;`
	
	`for(int i = 0; i < 6; i++) graph.add(i);`
	
	`graph.connect(1, 2, 40);`
	`graph.connect(1, 3, 6.9);`
	`graph.connect(4, 5, 27);`
	`graph.connect(3, 2, 2.1);`
	`graph.connect(1, 2, 11.2);`
	`graph.connect(1, 3, 67);`
	
	`graph.clear();`
	`cout << graph.shape() << endl;`
	`cout << (graph.empty() ? "Graph is empty!" : "Graph is not empty!") << endl;`
		`[Vertices: 0, Edges: 0]`
		`Graph is empty!`
			`[Vertices: 0, Edges: 0]`
			`Graph is empty!`
				✓

Passed all tests! ✓

Đúng

Marks for this submission: 1,00/1,00.

Câu hỏi **5**

Đúng

Đạt điểm 1,00 trên 1,00

Implement Breadth-first search

```
Adjacency *BFS(int v);
```

where Adjacency is a structure to store list of number.

```
#include <iostream>
#include <list>
using namespace std;

class Adjacency
{
private:
        list<int> adjList;
        int size;
public:
        Adjacency() {}
        Adjacency(int V) {}
        void push(int data)
        {
                adjList.push_back(data);
                size++;
        }
        void print()
        {
                for (auto const &i : adjList)
                        cout << " -> " << i;
        }
        void printArray()
        {
                for (auto const &i : adjList)
                        cout << i << " ";
        }
        int getSize() { return adjList.size(); }
        int getElement(int idx)
        {
                auto it = adjList.begin();
                advance(it, idx);
                return *it;
        }
};
```

And Graph is a structure to store a graph (see in your answer box)

For example:

Test	Result

int V = 6;
int visited = 0;
 	
Graph g(V);
Adjacency* arr = new Adjacency(V);
int edge[][2] = {{0,1},{0,2},{1,3},{1,4},{2,4},{3,4},{3,5},{4,5}};
 	
for(int i = 0; i < 8; i++)
{
    g.addEdge(edge[i][0], edge[i][1]);
}
 	
arr = g.BFS(visited);
arr->printArray();
delete arr;
	0 1 2 3 4 5

Test	Result

int V = 6;
int visited = 2;
 	
Graph g(V);
Adjacency* arr = new Adjacency(V);
int edge[][2] = {{0,1},{0,2},{1,3},{1,4},{2,4},{3,4},{3,5},{4,5}};
 	
for(int i = 0; i < 8; i++)
{
  g.addEdge(edge[i][0], edge[i][1]);
}
 	
arr = g.BFS(visited);
arr->printArray();
delete arr;
	2 0 4 1 3 5

Answer: (penalty regime: 0 %)

Reset answer

```
class Graph
{
private:
    int V;
    Adjacency *adj;

public:
    Graph(int V)
    {
        this->V = V;
        adj = new Adjacency[V];
    }

    void addEdge(int v, int w)
    {
        adj[v].push(w);
        adj[w].push(v);
    }

    void printGraph()
    {
        for (int v = 0; v < V; ++v)
        {
            cout << "\nAdjacency list of vertex " << v << "\nhead ";
            adj[v].print();
        }
    }

 /**
     * Thuật toán Breadth-First Search (BFS) - Tìm kiếm theo chiều rộng
     * @param v Đỉnh bắt đầu
     * @return Danh sách các đỉnh theo thứ tự duyệt BFS
     */
    Adjacency *BFS(int v)
    {
        // Khởi tạo mảng đánh dấu các đỉnh đã được thăm
        bool *visited = new bool[V];
        for (int i = 0; i < V; i++)
            visited[i] = false; // Khởi tạo tất cả các đỉnh là chưa thăm

        // Khởi tạo danh sách kết quả để lưu trữ thứ tự thăm BFS
        Adjacency *result = new Adjacency(V);

        // Khởi tạo hàng đợi (queue) cho BFS
        list<int> queue;

        // Đánh dấu đỉnh xuất phát đã được thăm và thêm vào hàng đợi
        visited[v] = true;
        queue.push_back(v);

        // Thêm đỉnh xuất phát vào kết quả
        result->push(v);
```

	Test	Expected	Got	

✓	`int V = 6;`
	`int visited = 0;`
	
	`Graph g(V);`
	`Adjacency* arr = new Adjacency(V);`
	`int edge[][2] = {{0,1},{0,2},{1,3},{1,4},{2,4},{3,4},{3,5},{4,5}};`
	
	`for(int i = 0; i < 8; i++)`
	`{`
	`  g.addEdge(edge[i][0], edge[i][1]);`
	`}`
	
	`arr = g.BFS(visited);`
	`arr->printArray();`
	`delete arr;`
		0 1 2 3 4 5	0 1 2 3 4 5	✓

✓	`int V = 6;`
	`int visited = 2;`
	
	`Graph g(V);`
	`Adjacency* arr = new Adjacency(V);`
	`int edge[][2] = {{0,1},{0,2},{1,3},{1,4},{2,4},{3,4},{3,5},{4,5}};`
	
	`for(int i = 0; i < 8; i++)`
	`{`
	`  g.addEdge(edge[i][0], edge[i][1]);`
	`}`
	
	`arr = g.BFS(visited);`
	`arr->printArray();`
	`delete arr;`
		2 0 4 1 3 5	2 0 4 1 3 5	✓

✓	`int V = 8, visited = 5;`
	
	`Graph g(V);`
	`Adjacency *arr;`
	`int edge[][2] = {{0,1}, {0,2}, {0,3}, {0,4}, {1,2}, {2,5}, {2,6}, {4,6}, {6,7}};`
	`for(int i = 0; i < 9; i++)`
	`{`
	`\tg.addEdge(edge[i][0], edge[i][1]);`
	`}`
	
	`// g.printGraph();`
	`// cout << endl;`
	`arr = g.BFS(visited);`
	`arr->printArray();`
	`delete arr;`
		5 2 0 1 6 3 4 7	5 2 0 1 6 3 4 7	✓

Passed all tests! ✓

Đúng

Marks for this submission: 1,00/1,00.

Câu hỏi **6**

Đúng

Đạt điểm 1,00 trên 1,00

Implement Depth-first search

```
Adjacency *DFS(int v);
```

where Adjacency is a structure to store list of number.

```
#include <iostream>
#include <list>
using namespace std;

class Adjacency
{
private:
        list<int> adjList;
        int size;
public:
        Adjacency() {}
        Adjacency(int V) {}
        void push(int data)
        {
                adjList.push_back(data);
                size++;
        }
        void print()
        {
                for (auto const &i : adjList)
                        cout << " -> " << i;
        }
        void printArray()
        {
                for (auto const &i : adjList)
                        cout << i << " ";
        }
        int getSize() { return adjList.size(); }
        int getElement(int idx)
        {
                auto it = adjList.begin();
                advance(it, idx);
                return *it;
        }
};
```

And Graph is a structure to store a graph (see in your answer box)

For example:

Test	Result

`int V = 8, visited = 0;`
 	
`Graph g(V);`
`Adjacency *arr;`
`int edge[][2] = {{0,1}, {0,2}, {0,3}, {0,4}, {1,2}, {2,5}, {2,6}, {4,6}, {6,7}};`
`for(int i = 0; i < 9; i++)`
`{`
`  g.addEdge(edge[i][0], edge[i][1]);`
`}`
 	
`// g.printGraph();`
`// cout << endl;`
`arr = g.DFS(visited);`
`arr->printArray();`
`delete arr;`
	`0 1 2 5 6 4 7 3`

Answer: (penalty regime: 0 %)

Reset answer

```
class Graph // Lớp Graph: Biểu diễn đồ thị
{
private:
    int V;          // Số đỉnh trong đồ thị
    Adjacency *adj; // Mảng các danh sách kề
public:
    // Hàm khởi tạo với tham số V (số đỉnh)
    Graph(int V)
    {
        this->V = V;
        adj = new Adjacency[V]; // Cấp phát động mảng các danh sách kề
    }
    // Thêm cạnh vào đồ thị không hướng (thêm w vào danh sách kề của v và ngược lại)
    void addEdge(int v, int w)
    {
        adj[v].push(w); // Thêm w vào danh sách kề của v
        adj[w].push(v); // Thêm v vào danh sách kề của w
    }
    // In thông tin đồ thị ra màn hình
    void printGraph()
    {
        for (int v = 0; v < V; ++v)
        {
            cout << "\nAdjacency list of vertex " << v << "\nhead ";
            adj[v].print(); // In danh sách kề của đỉnh v
        }
    }
    // Thuật toán DFS (Depth-First Search): Tìm kiếm theo chiều sâu
    Adjacency *DFS(int startVertex)
    {
        // Tạo mảng để đánh dấu các đỉnh đã thăm
        bool *visited = new bool[V];
        // Khởi tạo tất cả các đỉnh là chưa được thăm
        for (int i = 0; i < V; i++)
            visited[i] = false;
        // Tạo đối tượng Adjacency để lưu kết quả DFS (thứ tự thăm các đỉnh)
        Adjacency *result = new Adjacency();
        // Gọi hàm đệ quy DFS_Util để thực hiện thuật toán DFS
        DFS_Util(startVertex, visited, result);
        // Giải phóng bộ nhớ cho mảng visited khi không còn sử dụng
        delete[] visited;
        // Trả về danh sách các đỉnh theo thứ tự thăm DFS
        return result;
    }
private:
    // Hàm đệ quy hỗ trợ cho DFS
    void DFS_Util(int vertex, bool visited[], Adjacency *result)
    {
        // Bước 1: Đánh dấu đỉnh hiện tại đã thăm
        visited[vertex] = true;
        // Bước 2: Thêm đỉnh hiện tại vào danh sách kết quả
        // Đây là bước quan trọng để ghi nhận thứ tự thăm các đỉnh
        result->push(vertex);
        // Bước 3: Duyệt qua tất cả các đỉnh kề của đỉnh hiện tại
        // Thuật toán sẽ ưu tiên đi sâu trước khi đi rộng
        for (int i = 0; i < adj[vertex].getSize(); i++){
            // Lấy đỉnh kề thứ i của đỉnh hiện tại
            int adjacentVertex = adj[vertex].getElement(i);
            // Bước 4: Nếu đỉnh kề chưa được thăm, đệ quy để thăm nó và tất cả các đỉnh
            if (!visited[adjacentVertex])
                DFS_Util(adjacentVertex, visited, result);
        } } };
```

	Test	Expected	Got	

✓	`int V = 8, visited = 0;`
	
	`Graph g(V);`
	`Adjacency *arr;`
	`int edge[][2] = {{0,1}, {0,2}, {0,3}, {0,4}, {1,2}, {2,5}, {2,6}, {4,6}, {6,7}};`
	`for(int i = 0; i < 9; i++)`
	`{`
	`\tg.addEdge(edge[i][0], edge[i][1]);`
	`}`
	
	`// g.printGraph();`
	`// cout << endl;`
	`arr = g.DFS(visited);`
	`arr->printArray();`
	`delete arr;`
		`0 1 2 5 6 4 7 3`	`0 1 2 5 6 4 7 3`	✓

Passed all tests! ✓

Đúng

Marks for this submission: 1,00/1,00.

Câu hỏi **7**

Đúng

Đạt điểm 1,00 trên 1,00

The relationship between a group of people is represented by an adjacency-list `friends`. If `friends[u]` contains `v`, `u` and `v` are friends.Friendship is a two-way relationship. Two people are in a friend group as long as there is some path of mutual friends connecting them.

Request: Implement function:

```
int numberOfFriendGroups(vector<vector<int>>& friends);
```

Where `friends` is the adjacency-list representing the friendship (this list has between 0 and 1000 lists). This function returns the number of friend groups.

Example:

Given a adjacency-list: `[[1], [0, 2], [1], [4], [3], []]`

There are `3` friend groups: `[0, 1, 2], [3, 4], [5]`

Note:

In this exercise, the libraries `iostream, string, cstring, climits, utility, vector, list, stack, queue, map, unordered_map, set, unordered_set, functional, algorithm` have been included and `namespace std` is used. You can write helper functions and class. Importing other libraries is allowed, but not encouraged.

For example:

Test	Result

`vector<vector<int>> graph {`
`{1},`
`{0, 2},`
`{1},`
`{4},`
`{3},`
`{}`
`};`
`cout << numberOfFriendGroups(graph);`
	3

Answer: (penalty regime: 0 %)

Reset answer

```
/**
 * Hàm đếm số nhóm bạn bè trong mạng quan hệ
 *
 * @param friends Danh sách kề biểu diễn mối quan hệ bạn bè
 * @return Số nhóm bạn bè (số thành phần liên thông của đồ thị)
 */
int numberOfFriendGroups(vector<vector<int>>& friends) {
    int n = friends.size(); // Số người trong mạng
    vector<bool> visited(n, false); // Mảng đánh dấu người đã được thăm
    int count = 0; // Biến đếm số nhóm bạn bè

    // Dùng BFS để duyệt từng thành phần liên thông
    for (int i = 0; i < n; i++) {
        // Nếu người này chưa được thăm, tức là thuộc nhóm bạn bè mới
        if (!visited[i]) {
            count++; // Tăng số nhóm bạn bè

            // Sử dụng BFS để tìm tất cả các bạn trong cùng nhóm
            queue<int> q;
            q.push(i);
            visited[i] = true;

            while (!q.empty()) {
                int person = q.front();
                q.pop();

                // Duyệt qua danh sách bạn bè của person
                for (int friend_id : friends[person]) {
                    // Nếu người bạn chưa được thăm, thêm vào hàng đợi
                    if (!visited[friend_id]) {

```

```
32
33
34                          visited[friend_id] = true;
35                          q.push(friend_id);
36                      }
37                  }
38              }
39          }
40      }
41
42      return count; // Trả về tổng số nhóm bạn bè
43  }
44
```

	Test	Expected	Got	

✓	vector<vector<int>> graph {
	\t{1},
	\t{0, 2},
	\t{1},
	\t{4},
	\t{3},
	\t{}
	};
	cout << numberOfFriendGroups(graph);
		3	3	✓

✓	vector<vector<int>> graph {
	
	};
	cout << numberOfFriendGroups(graph);
		0	0	✓

Passed all tests! ✓

Đúng

Marks for this submission: 1,00/1,00.

Câu hỏi **8**

Đúng

Đạt điểm 1,00 trên 1,00

Implement function to detect a cyclic in Graph

```
bool isCyclic();
```

Graph structure is defined in the initial code.

For example:

Test	Result

DirectedGraph g(8);
int edege[][2] = {{0,6}, {1,2}, {1,4}, {1,6}, {3,0}, {3,4}, {5,1}, {7,0}, {7,1}};
 	
for(int i = 0; i < 9; i++)
g.addEdge(edege[i][0], edege[i][1]);
 	
if(g.isCyclic())
cout << "Graph contains cycle";
else
cout << "Graph doesn't contain cycle";
	Graph doesn't contain cycle

Answer: (penalty regime: 0 %)

Reset answer

```
#include <iostream>
#include <vector>
#include <list>
using namespace std;
// Lớp đồ thị có hướng
class DirectedGraph {
private:
    int V;                // Số đỉnh của đồ thị
    vector<list<int>> adj;  // Danh sách kề

    // Hàm DFS đệ quy để kiểm tra chu trình
    bool isCyclicUtil(int v, vector<bool>& visited, vector<bool>& recStack) {
        // Đánh dấu đỉnh hiện tại là đã thăm và thêm vào ngăn xếp đệ quy
        visited[v] = true;
        recStack[v] = true;
        // Duyệt tất cả các đỉnh kề với đỉnh hiện tại
        for (auto i = adj[v].begin(); i != adj[v].end(); ++i) {
            // Nếu đỉnh kề chưa được thăm, tiếp tục DFS từ đỉnh đó
            if (!visited[*i] && isCyclicUtil(*i, visited, recStack)) {
                return true;
            }
            // Nếu đỉnh kề đã có trong ngăn xếp đệ quy, có chu trình
            else if (recStack[*i]) {
                return true;
            }
        }
        // Xóa đỉnh khỏi ngăn xếp đệ quy khi quay lui
        recStack[v] = false;
        return false;
    }

public:
    // Hàm khởi tạo đồ thị với V đỉnh
    DirectedGraph(int V) {
        this->V = V;
        adj.resize(V);
    }
    // Thêm cạnh từ đỉnh u đến đỉnh v
    void addEdge(int u, int v) {
        adj[u].push_back(v);
    }
    // Kiểm tra xem đồ thị có chu trình hay không
    bool isCyclic() {
        // Khởi tạo mảng đánh dấu các đỉnh đã thăm
```

```
        vector<bool> visited(V, false);
        // Mảng đánh dấu các đỉnh đang nằm trong ngăn xếp đệ quy (đường đi hiện tại)
        vector<bool> recStack(V, false);

        // Gọi hàm DFS bắt đầu từ tất cả các đỉnh
        // để đảm bảo xét cả trường hợp đồ thị không liên thông
        for (int i = 0; i < V; i++) {
            if (!visited[i]) {
                if (isCyclicUtil(i, visited, recStack)) {
                    return true; // Nếu phát hiện chu trình, trả về true
                }
            }
        }
        return false; // Không có chu trình
    }
};
```

	Test	Expected	Got	

✓	DirectedGraph g(8);
	int edege[][2] = {{0,6}, {1,2}, {1,4}, {1,6}, {3,0}, {3,4}, {5,1}, {7,0}, {7,1}};
	
	for(int i = 0; i < 9; i++)
	\tg.addEdge(edege[i][0], edege[i][1]);
	
	if(g.isCyclic())
	\tcout << "Graph contains cycle";
	else
	\tcout << "Graph doesn't contain cycle";
		Graph doesn't contain cycle	Graph doesn't contain cycle	✓

Passed all tests! ✓

Đúng

Marks for this submission: 1,00/1,00.

Câu hỏi 9

Đúng

Đạt điểm 1,00 trên 1,00

Implement **topologicalSort** function on a graph. (Ref here)

```
void topologicalSort();
```

where Adjacency is a structure to store list of number. Note that, the vertex index starts from 0. **To match the given answer, please always traverse from 0 when performing the sorting.**

```
#include <iostream>
#include <list>
#include <vector>
using namespace std;

class Adjacency
{
private:
        list<int> adjList;
        int size;
public:
        Adjacency() {}
        Adjacency(int V) {}
        void push(int data)
        {
                adjList.push_back(data);
                size++;
        }
        void print()
        {
                for (auto const &i : adjList)
                        cout << " -> " << i;
        }
        void printArray()
        {
                for (auto const &i : adjList)
                        cout << i << " ";
        }
        int getSize() { return adjList.size(); }
        int getElement(int idx)
        {
                auto it = adjList.begin();
                advance(it, idx);
                return *it;
        }
};
```

And Graph is a structure to store a graph (see in your answer box). You could write one or more helping functions.

For example:

Test	Result

Graph g(6);
g.addEdge(5, 2);
g.addEdge(5, 0);
g.addEdge(4, 0);
g.addEdge(4, 1);
g.addEdge(2, 3);
g.addEdge(3, 1);
 	
g.topologicalSort();
	5 4 2 3 1 0

Answer: (penalty regime: 0 %)

Reset answer

```cpp
/**
 * Lớp Graph biểu diễn đồ thị có hướng
 */
class Graph {
    int V;             // Số đỉnh trong đồ thị
    Adjacency* adj;    // Mảng các danh sách kề

    /**
     * Hàm hỗ trợ đệ quy cho thuật toán sắp xếp tô pô
     * @param v: Đỉnh hiện tại đang xét
     * @param visited: Mảng đánh dấu các đỉnh đã thăm
     * @param Stack: Ngăn xếp lưu thứ tự tô pô
     */
    void topologicalSortUtil(int v, bool visited[], stack<int>& Stack) {
        // Đánh dấu đỉnh hiện tại đã được thăm
        visited[v] = true;

        // Duyệt qua tất cả các đỉnh kề với đỉnh hiện tại
        for (int i = 0; i < adj[v].getSize(); i++) {
            int adjacentVertex = adj[v].getElement(i);
            // Nếu đỉnh kề chưa được thăm, gọi đệ quy để thăm nó
            if (!visited[adjacentVertex]) {
                topologicalSortUtil(adjacentVertex, visited, Stack);
            }
        }

        // Sau khi duyệt hết tất cả các đỉnh kề, đẩy đỉnh hiện tại vào ngăn xếp
        // Đỉnh được thêm sau khi tất cả các đỉnh "con cháu" đã được thêm vào ngăn xếp
        Stack.push(v);
    }

public:
    /**
     * Hàm khởi tạo đồ thị với V đỉnh
     * @param V: Số đỉnh trong đồ thị
     */
    Graph(int V){
        this->V = V;
        adj = new Adjacency[V];
    }

    /**
     * Thêm cạnh từ đỉnh v đến đỉnh w
     * @param v: Đỉnh nguồn
     * @param w: Đỉnh đích
     */
    void addEdge(int v, int w){
        adj[v].push(w);
    }

    /**
     * Hàm thực hiện thuật toán sắp xếp tô pô và in kết quả
```

	Test	Expected	Got	

✓	`Graph g(6);`
	`g.addEdge(5, 2);`
	`g.addEdge(5, 0);`
	`g.addEdge(4, 0);`
	`g.addEdge(4, 1);`
	`g.addEdge(2, 3);`
	`g.addEdge(3, 1);`
	
	`g.topologicalSort();`
		`5 4 2 3 1 0`	`5 4 2 3 1 0`	✓

Passed all tests! ✓

Đúng

Marks for this submission: 1,00/1,00.

Câu hỏi **10**

Đúng

Đạt điểm 1,00 trên 1,00

Given a graph and a source vertex in the graph, find shortest paths from source to destination vertice in the given graph using Dijsktra's algorithm.

Following libraries are included: iostream, vector, algorithm, climits, queue

Note: All testcases have a fixed number of vertices set to 6.

For example:

Test	Result

`int n = 6;`
`int init[6][6] = {`
`{0, 10, 20, 0, 0, 0},`
`{10, 0, 0, 50, 10, 0},`
`{20, 0, 0, 20, 33, 0},`
`{0, 50, 20, 0, 20, 2},`
`{0, 10, 33, 20, 0, 1},`
`{0, 0, 0, 2, 1, 0} };`
 	
`int** graph = new int*[n];`
`for (int i = 0; i < n; ++i) {`
`graph[i] = init[i];`
`}`
 	
`cout << Dijkstra(graph, 0, 1);`
	10

Answer: (penalty regime: 0 %)

Reset answer

```
/**
 * Thuật toán Dijkstra tìm đường đi ngắn nhất từ đỉnh nguồn đến đỉnh đích trong đồ thị có trọng số
 * @param graph Ma trận kề biểu diễn đồ thị có trọng số
 * @param src Đỉnh nguồn
 * @param dst Đỉnh đích
 * @return Khoảng cách ngắn nhất từ đỉnh nguồn đến đỉnh đích
 */
int Dijkstra(int** graph, int src, int dst) {
    const int V = 6; // Số đỉnh cố định là 6 theo yêu cầu

    // Mảng lưu khoảng cách ngắn nhất từ đỉnh nguồn đến các đỉnh khác
    vector<int> dist(V, INT_MAX);
    // Mảng đánh dấu các đỉnh đã được xét
    vector<bool> visited(V, false);

    // Khoảng cách từ đỉnh nguồn đến chính nó là 0
    dist[src] = 0;

    // Tìm đường đi ngắn nhất cho tất cả các đỉnh
    for (int count = 0; count < V - 1; count++) {
        // Khởi tạo giá trị khoảng cách nhỏ nhất cho lần lặp này
        int min_dist = INT_MAX;
        int u = -1;

        // Tìm đỉnh chưa xét có khoảng cách nhỏ nhất từ đỉnh nguồn
        for (int v = 0; v < V; v++) {
            if (!visited[v] && dist[v] < min_dist) {
                min_dist = dist[v];
                u = v;
            }
        }
        // Nếu không tìm thấy đỉnh nào thỏa mãn, thoát khỏi vòng lặp
        if (u == -1) break;
        // Đánh dấu đỉnh đã được xét
        visited[u] = true;

        // Cập nhật khoảng cách của các đỉnh kề với đỉnh vừa xét
        for (int v = 0; v < V; v++) {
            // Cập nhật dist[v] chỉ khi:
```

```
            // 1. Có cạnh từ u đến v (graph[u][v] != 0)
            // 2. v chưa được xét
            // 3. Đường đi qua u đến v ngắn hơn đường đi hiện tại đến v
            if (!visited[v] && graph[u][v] != 0 &&
                dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }
    // Trả về khoảng cách ngắn nhất đến đỉnh đích
    return dist[dst];
}
```

	Test	Expected	Got	

✓	int n = 6;
	int init[6][6] = {
	\t{0, 10, 20, 0, 0, 0},
	\t{10, 0, 0, 50, 10, 0},
	\t{20, 0, 0, 20, 33, 0},
	\t{0, 50, 20, 0, 20, 2},
	\t{0, 10, 33, 20, 0, 1},
	\t{0, 0, 0, 2, 1, 0} };
	
	int** graph = new int*[n];
	for (int i = 0; i < n; ++i) {
	\tgraph[i] = init[i];
	}
	
	cout << Dijkstra(graph, 0, 1);
		10	10	✓

Passed all tests! ✓

Đúng

Marks for this submission: 1,00/1,00.