Trạng thái	Đã xong
Bắt đầu vào lúc	Thứ Bảy, 23 tháng 11 2024, 9:03 PM
Kết thúc lúc	Thứ Ba, 3 tháng 12 2024, 6:07 PM
Thời gian thực hiện	9 Các ngày 21 giờ
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

Ace editor not ready. Perhaps reload page?

Falling back to raw text area.

```
template<class T>
typename DGraph<T>::Edge* DGraph<T>::VertexNode::getEdge(VertexNode* toNode) {
    Edge* current = this->adList;
    while (current != nullptr) {
        if (current->toNode == toNode) {
            return current;
        }
        current = current->next;
    }
    return nullptr;
}

template<class T>
void DGraph<T>::VertexNode::addAdjacentEdge(Edge* newEdge) {
    newEdge->next = this->adList;
    this->adList = newEdge;
}

template<class T>
bool DGraph<T>::VertexNode::connectTo(VertexNode* toNode, float weight) {
    Edge* existingEdge = getEdge(toNode);
    if (existingEdge == nullptr) {
        Edge* newEdge = new Edge(this, toNode, weight);
        addAdjacentEdge(newEdge);
        return true;
    } else {
        existingEdge->weight = weight;
        return false;
    }
}

template<class T>
bool DGraph<T>::VertexNode::removeTo(VertexNode* toNode) {
    Edge* current = this->adList;
    Edge* prev = nullptr;
    while (current != nullptr) {
        if (current->toNode == toNode) {
            if (prev == nullptr) {
                this->adList = current->next;
            } else {
```

	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)`
				✓

✓	`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`
	`==================================================`

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(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

Ace editor not ready. Perhaps reload page?
Falling back to raw text area.

```
template<class T>
typename DGraph<T>::VertexNode* DGraph<T>::getVertexNode(T vertex) {
    VertexNode* current = this->nodeList;
    while (current != nullptr) {
        if (current->vertex == vertex) {
            return current;
        }
        current = current->next;
    }
    return nullptr;
}

template<class T>
void DGraph<T>::add(T vertex) {
    if (getVertexNode(vertex) != nullptr) {
        return; // Vertex already exists, do not add it again
    }
    VertexNode* newNode = new VertexNode(vertex);
    if (this->nodeList == nullptr) {
        this->nodeList = newNode;
    } else {
        VertexNode* current = this->nodeList;
        while (current->next != nullptr) {
            current = current->next;
        }
        current->next = newNode;
    }
    this->countVertex++;
}

template <class T>
void DGraph<T>::connect(T from, T to, float weight) {
    VertexNode* fromNode = getVertexNode(from);
    VertexNode* toNode = getVertexNode(to);
    if (fromNode == nullptr || toNode == nullptr) {
        throw VertexNotFoundException("Vertex doesn't exist!");
    }
    if (fromNode == toNode) {
        throw std::runtime_error("Self-loops are not allowed!");
    }
```

	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)`
	`==================================================`

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.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

Ace editor not ready. Perhaps reload page?
Falling back to raw text area.

```
template <class T>
bool DGraph<T>::removeEdge(T from, T to) {
    VertexNode* fromNode = getVertexNode(from);
    VertexNode* toNode = getVertexNode(to);

    if (!fromNode || !toNode) {
        throw VertexNotFoundException("Vertex doesn't exist!");
    }

    if (fromNode->removeTo(toNode)) {
        countEdge--;
        return true;
    }
    return false;
}

template <class T>
void DGraph<T>::removeVertex(T removeVertex) {
VertexNode* removeNode = getVertexNode(removeVertex);
    if (removeNode == nullptr) {
        throw VertexNotFoundException("Vertex doesn't exist!");
    }

    VertexNode* current = nodeList;
    while (current != nullptr) {
        if (current->removeTo(removeNode)) {
            countEdge--;
        }
        if (removeNode->removeTo(current)) {
            countEdge--;
        }
        current = current->next;
    }

    if (this->nodeList == removeNode) nodeList = nodeList->next;
    else {
        VertexNode* previous = nullptr;
        current = this->nodeList;
        while (current != removeNode) {
            previous = current;
```

	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]`

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.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

Ace editor not ready. Perhaps reload page?
Falling back to raw text area.

```
template<class T>
string DGraph<T>::shape() {
    return "[Vertices: " + to_string(this->countVertex) + ", Edges: " + to_string(this->countEdge)
+ "]";
}

template<class T>
bool DGraph<T>::empty() {
    return this->countVertex == 0 && this->countEdge == 0;
}

template<class T>
void DGraph<T>::clear() {
    VertexNode* current = this->nodeList;
    while (current != nullptr) {
        VertexNode* temp = current;
        current = current->next;
        delete temp;
    }
    this->nodeList = nullptr;
    this->countVertex = 0;
    this->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

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

Ace editor not ready. Perhaps reload page?
Falling back to raw text area.

```
#include <queue>
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();
                }
        }

        Adjacency *BFS(int v)
    {
        bool *visited = new bool[V];
        for(int i = 0; i < V; i++)
            visited[i] = false;

        queue<int> queue;
        Adjacency *bfsResult = new Adjacency();

        visited[v] = true;
        queue.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	✓

	Test	Expected	Got	

✓	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

Ace editor not ready. Perhaps reload page?
Falling back to raw text area.

```
#include <stack>
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();
                }
        }

        Adjacency *DFS(int v)
    {
        bool *visited = new bool[V];
        for (int i = 0; i < V; i++)
            visited[i] = false;

        stack<int> stack;
        Adjacency *result = new Adjacency();

        stack.push(v);

        while (!stack.empty())
        {
            v = stack.top();
            stack.pop();

            if (!visited[v])
            {
                result->push(v);
                visited[v] = true;
            }
```

	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

Ace editor not ready. Perhaps reload page?
Falling back to raw text area.

```
int numberOfFriendGroups(vector<vector<int>>& friends) {
    int n = friends.size();
    vector<bool> visited(n, false);
    int friendGroups = 0;

    function<void(int)> dfs = [&](int node) {
        visited[node] = true;
        for (int neighbor : friends[node]) {
            if (!visited[neighbor]) {
                dfs(neighbor);
            }
        }
    };

    for (int i = 0; i < n; ++i) {
        if (!visited[i]) {
            ++friendGroups;
            dfs(i);
```

	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

Ace editor not ready. Perhaps reload page?
Falling back to raw text area.

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

class DirectedGraph
{
        int V;
        vector<list<int>> adj;
public:
        DirectedGraph(int V)
        {
                this->V = V;
                adj = vector<list<int>>(V, list<int>());
        }
        void addEdge(int v, int w)
        {
                adj[v].push_back(w);
        }
        bool isCyclic()
        {
            vector<bool> visited(V, false);
        vector<bool> recStack(V, false);

        for(int i = 0; i < V; i++)
            if (isCyclicUtil(i, visited, recStack))
                return true;

        return false;

        }
private:
    bool isCyclicUtil(int v, vector<bool> &visited, vector<bool> &recStack)
    {
        if(!visited[v])
        {
            visited[v] = true;
            recStack[v] = true;

            for(auto i = adj[v].begin(); i != adj[v].end(); ++i)
            {
                if (!visited[*i] && isCyclicUtil(*i, visited, recStack))
                    return true;
                else if (recStack[*i])
                    return true;
            }
        }
        recStack[v] = false;
        return false;
    }
```

	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>
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

Ace editor not ready. Perhaps reload page?
Falling back to raw text area.

```
class Graph {

    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);
    }

    void topologicalSort(){
        stack<int> Stack;
        bool *visited = new bool[V];
        for (int i = 0; i < V; i++)
            visited[i] = false;

        for (int i = 0; i < V; i++)
            if (visited[i] == false)
                topologicalSortUtil(i, visited, Stack);

        while (!Stack.empty()) {
            cout << Stack.top() << " ";
            Stack.pop();
        }
        delete[] visited;
    }
    void topologicalSortUtil(int v, bool visited[], stack<int> &Stack) {
        visited[v] = true;

        for (int i = 0; i < adj[v].getSize(); ++i) {
            int adjVertex = adj[v].getElement(i);
            if (!visited[adjVertex])
                topologicalSortUtil(adjVertex, visited, Stack);
        }

        Stack.push(v);
```

	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

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

Ace editor not ready. Perhaps reload page?
Falling back to raw text area.

```
// Some helping functions

int Dijkstra(int** graph, int src, int dst) {
	int n = 6; // Assuming the graph has 6 vertices
	vector<int> dist(n, INT_MAX);
	vector<bool> visited(n, false);
	dist[src] = 0;

	for (int i = 0; i < n - 1; ++i) {
		int minDist = INT_MAX, u = -1;
		for (int v = 0; v < n; ++v) {
			if (!visited[v] && dist[v] < minDist) {
				minDist = dist[v];
				u = v;
			}
		}

		if (u == -1) break;
		visited[u] = true;

		for (int v = 0; v < n; ++v) {
			if (!visited[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] +
graph[u][v] < dist[v]) {
				dist[v] = dist[u] + graph[u][v];
			}
		}
	}

	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.