```
/*
Tung Luu and Wilson Le
*/
#include "graph.h"
#include "pq.h"
#include <climits>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

Graph::Graph(string fileName) {
  ifstream graph(fileName);
  if (graph.is_open()) {
    int numVertexes;
    graph >> numVertexes;
    for (int i = 0; i < numVertexes; i++) {
      adjList.push_back(new Vertex(i));
    }
    for (int i = 0; i < numVertexes; i++) {
      for (int j = 0; j < numVertexes; j++) {
        int weightij;
        graph >> weightij;
        if (i != j && weightij > 0) {
          adjList[i]->edges.push_back(new Edge(adjList[j], weightij));
        }
      }
    }
  }
}

Graph::~Graph() {}

string Graph::toString() {
  stringstream ss;
  for (Vertex *v : this->adjList) {
    ss << v->key << ": ";
    for (Edge *e : v->edges) {
      ss << e->dest->key << " ";
    }
    ss << endl;
  }
  return ss.str();
}

void Graph::dfs() {
  for (Vertex *vertex : adjList) {
    vertex->visited = false;
  }
  for (Vertex *vertex : adjList) {
    if (vertex->visited == false) {
      this->dfsVisit(vertex);
      cout << endl;
    }
  }
}

void Graph::dfsVisit(Vertex *u) {
  u->visited = true;
  cout << u->key << " ";
  for (Edge *edge : adjList[u->key]->edges) {
    if (edge->dest->visited == false) {
      dfsVisit(edge->dest);
    }
  }
}

bool Graph::cycle() {
  for (Vertex *vertex : adjList) {
```

```
    vertex->visited = false;
  }
  vector<bool> visitedStack;
  for (int i = 0; i < adjList.size(); i++)
    visitedStack.push_back(false);
  for (Vertex *vertex : adjList) {
    if (vertex->visited == false) {
      if (dfsCycleVisit(vertex, visitedStack)) {
        return true;
      }
    }
  }
  return false;
}

bool Graph::dfsCycleVisit(Vertex *u, vector<bool> &visitedStack) {
  u->visited = true;
  visitedStack[u->key] = true;
  for (Edge *edge : adjList[u->key]->edges) {
    if (!edge->dest->visited && dfsCycleVisit(edge->dest, visitedStack)) {
      return true;
    } else if (visitedStack[edge->dest->key]) {
      return true;
    }
  }
  visitedStack[u->key] = false;
  return false;
}

/*
In this implementation, I use a trick to avoid using the decreaseKey operation
of the priority queue, since the decreaseKey operations requires us to save the
information about the index of different vertexes in the priority queue, which
keeps changing overtime. In this implementation, instead of using the decrease
key operation when update minimum weight of a vertex, I just insert that vertex
again in the pq, and the pq will automatically push that vertex up. To deal with
duplicated vertex in the queue, I check if a vertex has already been put into
the MST in O(1). The memory efficiency will not be impacted much, since we have
the maximum of O(E) vertices in the priority queue, while we already using O(V +
E) memory to store graph. The time complexity will be O(E log(E)), but since E =
O(V^2) => log(E) = O(log(V^2)) = O(logV). Therefore, the time complexity will be
O(E logV)
*/
void Graph::prim(int root) {
  // The total number of edges in the graph
  int numEdges = 0;

  for (Vertex *vertex : adjList) {
    vertex->minWeight = INT_MAX;
    vertex->parent = nullptr;
    numEdges += vertex->edges.size();
  }

  adjList[root]->minWeight = 0;

  // We will have at maximum O(E) vertices in the priority queue
  MinPriorityQueue<Vertex> pq(numEdges / 2);
  pq.insert(adjList[root]);

  // If the vertex i is already in the MST, then inMST[i] = true, false
  // otherwise.
  vector<bool> inMST(adjList.size(), false);

  while (!pq.empty()) {
    Vertex *top = pq.extractMin();
    // If the vertex has already been put into the MST, then skip this vertex.
    if (inMST[top->key])
      continue;
    inMST[top->key] = true;
```

```
    for (Edge *edge : top->edges) {
      if (!inMST[edge->dest->key] && edge->weight < edge->dest->minWeight) {
        edge->dest->minWeight = edge->weight;
        edge->dest->parent = top;
        // Instead of calling decreaseKey and deal with indexes, just insert the
        // vertex again and let the pq put it at the right place
        pq.insert(edge->dest);
      }
    }
  }

  for (Vertex *vertex : adjList) {
    if (vertex->parent == nullptr && vertex != adjList[root]) {
      cout << "The graph is disconnected. Therefore, prim operation is not "
              "possible."
           << '\n';
      return;
    }
  }

  for (Vertex *vertex : adjList) {
    if (vertex->parent) {
      cout << vertex->parent->key << " " << vertex->key << " "
           << vertex->minWeight << '\n';
    }
  }
}
```

```
/*
Tung Luu and Wilson Le
*/
#include "vertex.h"
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
using namespace std;

#ifndef GRAPH_H
#define GRAPH_H

class Graph {
public:
  friend void testGraphConstructor();
  friend void testGraphPrim();
  Graph(string fileName);
  Graph(const Graph &otherGraph);
  Graph &operator=(const Graph &otherGraph);
  ~Graph();

  string toString();

  void dfs();
  bool cycle();
  void prim(int root);

private:
  vector<Vertex *> adjList;
  void dfsVisit(Vertex *u);
  bool dfsCycleVisit(Vertex *u, vector<bool> &visited);
};

#include "graph.cpp"
#endif
```

```
/*
Tung Luu and Wilson Le
*/
#include "heap.h"
#include <sstream>

// Implement heap methods here.

/*============================================================
MinHeap(int n = DEFAULT_SIZE)      //default constructor
Precondition: Must be given a capacity size (n)
Postcondition: An empty heap with capacity of n (1000 (default))
============================================================*/
template <class KeyType> MinHeap<KeyType>::MinHeap(int n) {
  A = new KeyType *[n];
  this->heapSize = 0;
  this->capacity = n;
}

/*============================================================
MinHeap(vector<KeyType *> initA)      //construct heap from vector
Precondition: Must be given a vector and a capacity
Postcondition: A min heap constructed from the vector
============================================================*/
template <class KeyType>
MinHeap<KeyType>::MinHeap(vector<KeyType *> initA, int cap) {
  A = new KeyType *[cap];
  this->capacity = cap;
  this->heapSize = initA.size();
  for (int i = 0; i < initA.size(); i++) {
    this->A[i] = initA[i]; // traverse through initA and copy each element to
                           // current heap
  }
  buildHeap();
}

/*============================================================
MinHeap(const MinHeap<KeyType>& heap);  // copy constructor
Precondition: Must be given a heap
Postcondition: A min heap copied from the given heap
============================================================*/
template <class KeyType>
MinHeap<KeyType>::MinHeap(const MinHeap<KeyType> &heap) {
  copy(heap); // call copy method which copies each element of heap parameter to
              // current heap
}

/*============================================================
~MinHeap();                                 // destructor
Precondition: Given a heap
Postcondition: The heap is deallocated
============================================================*/
template <class KeyType> MinHeap<KeyType>::~MinHeap() {
  this->destroy(); // call destroy method that deallocates current object
}

/*============================================================
heapSort(KeyType sorted[]);  // heapsort, return result in sorted
Precondition: Must be given a heap to be sorted
Postcondition: The heap is sorted in ascending order and the result is stored in
sorted
============================================================*/
template <class KeyType> void MinHeap<KeyType>::heapSort(KeyType *sorted[]) {
  // One by one extract an element from heap
  int n = this->heapSize;
  for (int i = n - 1; i > 0; i--) {
    // Swap current root and end
    swap(0, i);
    this->heapSize -= 1;
    // call max heapify on the reduced heap
```

```
    heapify(0);
  }
  this->heapSize = n;
  for (int i = 0; i < this->heapSize; i++) {
    sorted[n - i - 1] = this->A[i]; // reverse max heap to get min heap
  }
}

/*===========================================================
operator = (const MinHeap<KeyType>& heap);  // assignment operator
Precondition: Must be given a heap to be copied
Postcondition: Assign the heap to a new heap
=============================================================*/
template <class KeyType>
MinHeap<KeyType> &MinHeap<KeyType>::operator=(const MinHeap<KeyType> &heap) {
  this->copy(); // call copy method that copies each element of the heap
                // parameter to current heap
  return *this;
}

/*===========================================================
heapify(int index);          // heapify subheap rooted at index
Precondition: The subtrees are valid heaps.
Postcondition: The min heap property is maintained by calling the heapifyR
=============================================================*/
template <class KeyType> void MinHeap<KeyType>::heapify(int index) {
  heapifyR(index); // calls recursive heapify
}

/*===========================================================
heapifyR(int index);          // heapify subheap rooted at index
Precondition: The subtrees are valid heaps.
Postcondition: The min heap property is maintained by recursively calling
heapifyR
=============================================================*/
template <class KeyType> void MinHeap<KeyType>::heapifyR(int index) {
  int smallest = index;
  int left = leftChild(index);
  int right = rightChild(index);

  if (left < heapSize && *(A[left]) < *(A[smallest])) {
    smallest = left;
  }
  if (right < heapSize && *(A[right]) < *(A[smallest])) {
    smallest = right; // switch smallest to right if A[right] is the smaller of
                      // the two children
  }

  if (smallest != index) {
    swap(smallest,
         index);       // swap current index with the index of the smaller child
    heapify(smallest); // recursively call heapify on lower children
  }
}

/*===========================================================
heapify(int index);          // heapify subheap rooted at index
Precondition: The subtrees are valid heaps.
Postcondition: The min heap property is maintained by iteratively heapifying
=============================================================*/
template <class KeyType> void MinHeap<KeyType>::heapifyI(int index) {
  int smallest = index;
  int left = leftChild(index);
  int right = rightChild(index);

  if (left < heapSize && *(A[left]) < *(A[smallest])) {
    smallest = left; // intitialize smallest
  }
  if (right < heapSize && *(A[right]) < *(A[smallest])) {
    smallest = right; // switch smallest to right if A[right] is the smaller of
```

```
                       // the two children
  }

  while (smallest != index) // iterative call
  {
    swap(smallest, index);
    index = smallest;
    left = leftChild(index);
    right = rightChild(index);

    if (left < heapSize && *(A[left]) < *(A[smallest])) {
      smallest = left;
    }
    if (right < heapSize && *(A[right]) < *(A[smallest])) {
      smallest = right;
    }
  }
}

/*============================================================
buildHeap();                 // build heap
Precondition: Used on an array
Postcondition: Build a new heap from the array
============================================================*/
template <class KeyType> void MinHeap<KeyType>::buildHeap() {
  for (int i = heapSize / 2; i >= 0; i--) {
    heapify(i); // call heapify on the first half of the array
  }
}

/*============================================================
swap(int index1, int index2);        // swap elements in A
Precondition: Must be given two indices of the array
Postcondition: The values of the two indices are exchanged
============================================================*/
template <class KeyType> void MinHeap<KeyType>::swap(int index1, int index2) {
  KeyType *temp = this->A[index1]; // temporary variable to store A[index1]
  this->A[index1] = this->A[index2];
  this->A[index2] = temp;
}

/*============================================================
copy(const MinHeap<KeyType>& heap);  // copy heap to this heap
Precondition: Must be given a heap to copy
Postcondition: Copy the heap to this heap
============================================================*/
template <class KeyType>
void MinHeap<KeyType>::copy(const MinHeap<KeyType> &heap) {
  this->heapSize = heap.heapSize;
  this->capacity = heap.capacity;
  A = new KeyType *[this->capacity];
  for (int i = 0; i < heap.heapSize; i++) {
    this->A[i] = heap.A[i]; // traverses through the heap parameter and copies
                            // each of the element to the current heap
  }
}

/*============================================================
destroy();                           // deallocate heap
Precondition: Given a heap
Postcondition: The heap is deallocated
============================================================*/
template <class KeyType> void MinHeap<KeyType>::destroy() {
  delete[] A; // deallocate object
}

// Use the following toString() for testing purposes.

template <class KeyType> std::string MinHeap<KeyType>::toString() const {
  std::stringstream ss;
```

```
  if (capacity == 0)
    ss << "[ ]";
  else {
    ss << "[";
    if (heapSize > 0) {
      for (int index = 0; index < heapSize - 1; index++)
        ss << *(A[index]) << ", ";
      ss << *(A[heapSize - 1]);
    }
    ss << " | ";
    if (capacity > heapSize) {
      for (int index = heapSize; index < capacity - 1; index++)
        ss << *(A[index]) << ", ";
      ss << *(A[capacity - 1]);
    }
    ss << "]";
  }
  return ss.str();
}

template <class KeyType>
std::ostream &operator<<(std::ostream &stream, const MinHeap<KeyType> &heap) {
  return stream << heap.toString();
}
```

```
/*
Tung Luu and Wilson Le
*/
#ifndef HEAP_H
#define HEAP_H

#include <iostream>

const int DEFAULT_SIZE = 100;

template <class KeyType> class MinHeap {
public:
  MinHeap(int n = DEFAULT_SIZE);              // default constructor
  MinHeap(vector<KeyType *> initA, int cap); // construct heap from vector
  MinHeap(const MinHeap<KeyType> &heap);     // copy constructor
  ~MinHeap();                                // destructor

  void heapSort(KeyType *sorted[]); // heapsort, return result in sorted

  MinHeap<KeyType> &
  operator=(const MinHeap<KeyType> &heap); // assignment operator
  std::string toString() const;            // return string representation

protected:
  KeyType **A;  // array containing the heap
  int heapSize; // size of the heap
  int capacity; // size of A

  void heapify(int index); // heapify subheap rooted at index
  void buildHeap();        // build heap
  int leftChild(int index) {
    return 2 * index + 1;
  } // return index of left child
  int rightChild(int index) {
    return 2 * index + 2;
  } // return index of right child
  int parent(int index) { return (index - 1) / 2; } // return index of parent
  void heapifyR(int index);                         // recursive heapify
  void heapifyI(int index);                         // iterative heapify
  void swap(int index1, int index2);                // swap elements in A
  void copy(const MinHeap<KeyType> &heap);          // copy heap to this heap
  void destroy();                                   // deallocate heap
};

template <class KeyType>
std::ostream &operator<<(std::ostream &stream, const MinHeap<KeyType> &heap);

#include "heap.cpp"

#endif
```

```
/*
Tung Luu and Wilson Le
*/
// These 3 constructors just call the corresponding MinHeap constructors. That's
// all.
/*==========================================================
MinPriorityQueue();                      // default constructor
Precondition: None
Postcondition: An empty min priority queue of capacity 0
===========================================================*/
template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue() // : MinHeap<KeyType>()
{}

/*==========================================================
MinPriorityQueue(int n);                 // construct an empty MPQ with capacity
n Precondition: Given capacity n Postcondition: An empty min priority queue of
capacity n
===========================================================*/
template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue(int n) : MinHeap<KeyType>(n) {}

template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue(vector<KeyType *> initA, int cap)
    : MinHeap<KeyType>(initA, cap) {}

/*==========================================================
MinPriorityQueue(const MinPriorityQueue<KeyType>& pq);      // copy constructor
Precondition: Given a min priority queue to copy
Postcondition: A min priority queue copied from the MPQ given
===========================================================*/
template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue(const MinPriorityQueue<KeyType> &pq)
    : MinHeap<KeyType>(pq) {}

/*==========================================================
minimum()                                // return the minimum element
Precondition: Given a min priority queue
Postcondition: The minimum element is returned
===========================================================*/
template <class KeyType> KeyType *MinPriorityQueue<KeyType>::minimum() const {
  if (empty())          // check if the MPQ is empty
    throw EmptyError(); // throw exception
  return A[0];
}

/*==========================================================
extractMin()                             // delete the minimum element and return
it Precondition: Given a min priority queue Postcondition: The minimum element
is deleted from the MPQ and returned
===========================================================*/
template <class KeyType> KeyType *MinPriorityQueue<KeyType>::extractMin() {
  if (empty())            // check if the MPQ is empty
    throw EmptyError();   // throw exception
  KeyType *min = A[0];    // store the minimum element
  A[0] = A[heapSize - 1]; // delete the minimum element and replace it with the
                          // last element currently in MPQ
  heapSize--;             // decrement heapSize
  heapify(0);             // call heapify since the subtrees are valid heaps
  return min;
}

/*==========================================================
decreaseKey(int index, KeyType* key)     // decrease the value of an element
Precondition: Given the index of the element and the key to decrease
Postcondition: The element at index is decreased by key
===========================================================*/
template <class KeyType>
void MinPriorityQueue<KeyType>::decreaseKey(int index, KeyType *key) {
  if (*key > *(A[index])) // check if key is greater than the current element
```

```
    throw KeyError();      // throw exception
  A[index] = key;          // set value of A[index] to key
  while ((index > 0) && *(A[index]) <= *(A[parent(index)]))
  // while not reaching root and current element is smaller than its parent
  {
    swap(index, parent(index)); // swap current element with its parent
    index = parent(index);
  }
}

/*============================================================
insert(KeyType* key)                    // insert a new element
Precondition: Given an element to insert
Postcondition: The element is inserted
============================================================*/
template <class KeyType> void MinPriorityQueue<KeyType>::insert(KeyType *key) {
  if (heapSize == capacity) // check if the MPQ is full
    throw FullError();      // throw full exception
  heapSize++;               // increment heapSize
  A[heapSize - 1] = key;    // insert key to last element
  decreaseKey(heapSize - 1,
              key); // swap key with its parent until find the correct positon
}

/*============================================================
empty()                         // return whether the MPQ is empty
Precondition: Given a min priority queue
Postcondition: True if the MPQ is empty, Else otherwise
============================================================*/
template <class KeyType> bool MinPriorityQueue<KeyType>::empty() const {
  if (heapSize == 0) // if the MPQ is empty
    return 1;
  return 0; // if the MPQ is not empty
}

/*============================================================
length()                        // return the number of keys
Precondition: Given a min priority queue
Postcondition: The size of the MPQ is returned
============================================================*/
template <class KeyType> int MinPriorityQueue<KeyType>::length() const {
  return heapSize;
}

/*============================================================
toString()
// return a string representation of the MPQ Precondition: Given a min priority
queue Postcondition: The string representation of the MPQ is returned
============================================================*/
template <class KeyType>
std::string MinPriorityQueue<KeyType>::toString() const {
  std::stringstream ss;

  if (heapSize == 0) {
    ss << "[ ]";
  } else {
    ss << "[";
    for (int index = 0; index < heapSize - 1; index++)
      ss << *(A[index]) << ", ";
    ss << *(A[heapSize - 1]) << "]";
  }
  return ss.str();
}

template <class KeyType>
std::ostream &operator<<(std::ostream &stream,
                         const MinPriorityQueue<KeyType> &pq) {
  stream << pq.toString();

  return stream;
```

```
}
```

```cpp
/*
Tung Luu and Wilson Le
*/
#ifndef PQ_H
#define PQ_H

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

template <class KeyType> class MinPriorityQueue : public MinHeap<KeyType> {
public:
  MinPriorityQueue();      // default constructor
  MinPriorityQueue(int n); // construct an empty MPQ with capacity n
  MinPriorityQueue(vector<KeyType *> initA, int cap);
  MinPriorityQueue(const MinPriorityQueue<KeyType> &pq); // copy constructor

  KeyType *minimum() const; // return the minimum element
  KeyType *extractMin();    // delete the minimum element and return it
  void decreaseKey(int index, KeyType *key); // decrease the value of an element
  void insert(KeyType *key);                 // insert a new element
  bool empty() const;                        // return whether the MPQ is empty
  int length() const;                        // return the number of keys
  std::string toString() const; // return a string representation of the MPQ

  // Specify that MPQ will be referring to the following members of
  // MinHeap<KeyType>.

  using MinHeap<KeyType>::A;
  using MinHeap<KeyType>::heapSize;
  using MinHeap<KeyType>::capacity;
  using MinHeap<KeyType>::parent;
  using MinHeap<KeyType>::swap;
  using MinHeap<KeyType>::heapify;

  /* The using statements are necessary to resolve ambiguity because
     these members do not refer to KeyType.  Alternatively, you could
     use this->heapify(0) or MinHeap<KeyType>::heapify(0).
  */
};

template <class KeyType>
std::ostream &operator<<(std::ostream &stream,
                         const MinPriorityQueue<KeyType> &pq);

class FullError {};  // MinPriorityQueue full exception
class EmptyError {}; // MinPriorityQueue empty exception
class KeyError {};   // MinPriorityQueue key exception

#include "pq.cpp"

#endif
```

```
/*
Tung Luu and Wilson Le
*/
#include <string>
#include <vector>

using namespace std;

#ifndef VERTEX_H
#define VERTEX_H

class Vertex;

class Edge {
public:
  Vertex *dest;
  int weight;

  Edge(Vertex *dest, int weight) : dest(dest), weight(weight){};
};

class Vertex {
public:
  int key;
  int minWeight; // minimum weight of any edge connecting v to a vertex in the
                 // tree
  Vertex *parent;
  vector<Edge *> edges;
  bool visited;

  bool operator<(const Vertex &other) {
    return (this->minWeight < other.minWeight);
  }

  bool operator<=(const Vertex &other) {
    return (this->minWeight <= other.minWeight);
  }

  bool operator>(const Vertex &other) {
    return (this->minWeight > other.minWeight);
  }

  Vertex(int key) : key(key){};
};
#endif
```

```
/*
Tung Luu and Wilson Le
*/
#include "graph.h"
#include <climits>
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

void testGraphConstructor() {
  cout << "Testing graph constructor" << endl;
  Graph graph1("sampleGraph(1).txt");
  string graphString1 = "0: 1 2 \n1: 0 2 3 \n2: 0 1 4 \n3: 1 4 \n4"
                        ": 2 3 \n";
  assert(graph1.toString() == graphString1);
  cout << "\tTest 1 Passed" << endl;

  Graph graph2("sampleGraph(2).txt");
  string graphString2 = "0: 1 \n1: 0 \n2: 3 \n3: 2 \n";
  assert(graph2.toString() == graphString2);
  cout << "\tTest 2 Passed" << endl;
}

int main() {
  testGraphConstructor();
  Graph graph1("sampleGraph(1).txt");
  graph1.prim(0);
  // expected
  // 0 1 1
  // 1 2 1
  // 1 3 1
  // 3 4 1
  cout << "dfs output: " << endl;
  graph1.dfs();
  // expected
  // 0 1 2 4 3

  cout << "cycle output:" << endl;
  // expected 1
  cout << graph1.cycle() << endl;

  Graph graph2("sampleGraph(2).txt");
  graph2.prim(0);
  // expected
  // The graph is disconnected. Therefore, prim operation is not possible.

  cout << "dfs output: " << endl;
  graph2.dfs();
  // expected
  // 0 1
  // 2 3

  cout << "cycle output:" << endl;
  // expected 1
  cout << graph2.cycle() << endl;

  Graph graph3("sampleGraph(3).txt");
  graph3.prim(0);
  // expected
  // The graph is disconnected. Therefore, prim operation is not possible.

  cout << "dfs output: " << endl;
  graph3.dfs();
  // expected
  // 0 1
  // 2 3

  cout << "cycle output:" << endl;
  // expected 0
```

```
  cout << graph3.cycle() << endl;

  Graph graph4("sampleGraph(4).txt");
  graph4.prim(0);
  // expected
  // The graph is disconnected. Therefore, prim operation is not possible.

  cout << "dfs output: " << endl;
  graph4.dfs();
  // expected
  // 0 1
  // 2 3

  cout << "cycle output:" << endl;
  // expected 0
  cout << graph4.cycle() << endl;
}
```

```
5
0 1 2 0 0
1 0 1 1 0
2 1 0 0 3
0 1 0 0 1
0 0 3 1 0
```

```
4
0 1 0 0
1 0 0 0
0 0 0 1
0 0 1 0
```

```
4
0 1 0 0
0 0 1 0
0 0 0 1
0 0 0 0
```

```
4
0 1 1 0
0 0 1 0
1 0 0 1
0 0 0 1
```