```cpp
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

/*
 * File:   IGraph.h
 * Author: LTSACH
 *
 * Created on 23 August 2020, 17:28
 */

#ifndef IGRAPH_H
#define IGRAPH_H
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

#include "list/DLinkedList.h"


class VertexNotFoundException: public std::exception{
private:
    string vertex;
public:
    VertexNotFoundException(string vertex){
        this->vertex = vertex;
    }
    const char * what () const throw (){
        stringstream os;
        os << "Vertex (" << this->vertex << "): is not found";
        return os.str().c_str();
    }
};

class EdgeNotFoundException: public std::exception{
private:
    string edge;
public:
    EdgeNotFoundException(string edge){
        this->edge = edge;
    }
    const char * what () const throw (){
        stringstream os;
        os << "Edge (" << edge << "): is not found";
        return os.str().c_str();
    }
};

template<class T>
struct Edge{
    T from, to;
    float weight;
    Edge(T from, T to, float weight=0){
        this->from = from;
        this->to = to;
        this->weight = weight;
    };
    Edge(const Edge& edge){
        this->from = edge.from;
        this->to = edge.to;
        this->weight = edge.weight;
    }
};
/*
 * IGraph: define APIs for a graph data structure
 *  >> T: type of vertices
 */
template<class T>
class IGraph{
public:
    virtual ~IGraph(){};

    /*
    add (T vertex):
    add a vertex to graph
    */
    virtual void add(T vertex)=0;
    virtual void remove(T vertex)=0;
    virtual bool contains(T vertex)=0;

    /*
    connect(T from, T to, float weight):
    connect 2 vertexes (from -> to) with weight
    */
    virtual void connect(T from, T to, float weight=0)=0;
    virtual void disconnect(T from, T to)=0;
    virtual float weight(T from, T to)=0;

    virtual DLinkedList<T> getOutwardEdges(T from)=0;
    virtual DLinkedList<T> getInwardEdges(T to)=0;

    virtual int size()=0;
    virtual bool empty()=0;
    virtual void clear()=0;

    /*
    inDegree(T vertex):
    find the in degree of the vertex
    */
    virtual int inDegree(T vertex)=0;

    /*
    outDegree(T vertex):
    find the out degree of the vertex
    */
    virtual int outDegree(T vertex)=0;

    virtual DLinkedList<T> vertices()=0;
    virtual bool connected(T from, T to)=0;

    virtual string toString()=0;
};

/*
 * Path: model a path on graphs
 *  >> a path = sequence of vertices,
 *      -> stored in: "path" (DLinkedList<T>)
 *      -> its cost: stored in "cost" (float)
 *
 */
template<class T>
class Path{
private:
    DLinkedList<T> path;
    float cost;
public:
    Path(){
        cost = 0;
    }
    DLinkedList<T>& getPath(){
        return this->path;
    }
    float getCost(){
        return cost;
    }
```

```cpp
    void setCost(float cost){
        this->cost = cost;
    }

    ///////////////////////////////////////////
    void add(T item){
        this->path.add(item);
    }
    string toString(string (*item2str)(T&)=0){
        stringstream os;
        os << this->path.toString(item2str)
                << ", cost: " << this->cost;
        return os.str();
    }
};

/*
 * IFinder: the path finder, contains searching algorithms on graph
 *
 */
template<class T>
class IFinder{
    virtual DLinkedList<Path<T>> dijkstra(IGraph<T>* pGraph, T start)=0;
};

#endif /* IGRAPH_H */

/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

/*
 * File:   IMap.h
 * Author: LTSACH
 *
 * Created on 22 August 2020, 21:53
 */

#ifndef IMAP_H
#define IMAP_H

#include "list/DLinkedList.h"
#include <string>
using namespace std;


class KeyNotFound: public std::exception{
private:
    string desc;
public:
    KeyNotFound(string desc){
        this->desc = desc;
    }
    const char * what () const throw (){
        stringstream os;
        os << this->desc;
        return os.str().c_str();
    }
};


template<class K, class V>
struct Pair{
    K key;
    V value;
    Pair(K key, V value){
        this->key = key;
        this->value = value;
    }
    Pair(const Pair& pair){
        this->key = pair.key;
        this->value = pair.value;
    }
    Pair& operator=(const Pair& pair){
        this->key = pair.key;
        this->value = pair.value;
    }
};

template<class K, class V>
class IMap {
public:
    virtual ~IMap(){};
    //
    /*
    put(K key, V value):
    if key is not in the map:
        + add a mapping key->value to the map
        + return value
    else:
        + associate key with the new value (passed as parameter)
        + return the old value
    */
    virtual V put(K key, V value)=0;

    /*
    get(K key):
    if key in the map: return the associated value
     else: KeyNotFound exception thrown

    */
    virtual V& get(K key)=0;

    /*
    remove(K key):
    if key is in the map: remove it from the map, and return the associated value
    else: KeyNotFound exception thrown

    >> deleteKeyInMap(K key): delete key stored in map; in cases, K is a pointer type
    */
    virtual V remove(K key, void (*deleteKeyInMap)(K)=0)=0;

    /*
    remove(K key, V value):
    if there is a mapping key->value in the map: remove it and return true
    else: return false

    >> deleteKeyInMap(K key): delete key stored in map; in cases, K is a pointer type
    >> deleteValueInMap(V value): delete key stored in map; in cases, V is a pointer type
    */
    virtual bool remove(K key, V value, void (*deleteKeyInMap)(K)=0, void
    (*deleteValueInMap)(V)=0)=0;

    /*
    containsKey(K key):
    if key is in the map: return true
    else: return false
    */
    virtual bool containsKey(K key)=0;

    /*
    containsKey(V value):
    if value is in the map: return true
    else: return false
    */
    virtual bool containsValue(V value)=0;

    /*
    empty():
    return true if the map is empty
    else: return false
    */
    virtual bool empty()=0;

    /*
    size():
    return number of pairs key->value
    */
    virtual int size()=0;

    /*
    clear():
    clear all pairs key->value in the map
    */
    virtual void clear() = 0;

    /*
    toString():
    return a string representing the map

    >> key2str(K& key): convert key to string; if not supplied then K must support
    extraction operator (<<)
    >> value2str(V& value): convert value to string; if not supplied then V must support
    extraction operator (<<)
    */
    virtual string toString(string (*key2str)(K&)=0, string (*value2str)(V&)=0 )=0;

    /*
     * keys(): return a set of keys stored in the map
     */
    virtual DLinkedList<K> keys()=0;

    /*
     * values(): return a set of values stored in the map
     */
    virtual DLinkedList<V> values()=0;

    /*
     * clashes(): return a list containing the collision count for each address
     */
    virtual DLinkedList<int> clashes()=0;
};


#endif /* IMAP_H */


```