

```

1  /*
2  * To change this license header, choose License Headers in Project Properties.
3  * To change this template file, choose Tools | Templates
4  * and open the template in the editor.
5  */
6
7  /*
8  * File:   IGraph.h
9  * Author: LTSACH
10  *
11  * Created on 23 August 2020, 17:28
12  */
13
14  #ifndef IGRAPH_H
15  #define IGRAPH_H
16  #include <iostream>
17  #include <string>
18  #include <sstream>
19  using namespace std;
20
21  #include "list/DLinkedList.h"
22
23
24  class VertexNotFoundException: public std::exception{
25  private:
26      string vertex;
27  public:
28      VertexNotFoundException(string vertex){
29          this->vertex = vertex;
30      }
31      const char * what () const throw (){
32          stringstream os;
33          os << "Vertex (" << this->vertex << "): is not found";
34          return os.str().c_str();
35      }
36  };
37
38  class EdgeNotFoundException: public std::exception{
39  private:
40      string edge;
41  public:
42      EdgeNotFoundException(string edge){
43          this->edge = edge;
44      }
45      const char * what () const throw (){
46          stringstream os;
47          os << "Edge (" << edge << "): is not found";
48          return os.str().c_str();
49      }
50  };
51
52  template<class T>
53  struct Edge{
54      T from, to;
55      float weight;
56      Edge(T from, T to, float weight=0){
57          this->from = from;
58          this->to = to;
59          this->weight = weight;
60      };
61      Edge(const Edge& edge){
62          this->from = edge.from;
63          this->to = edge.to;
64          this->weight = edge.weight;
65      }
66  };
67  /*
68  * IGraph: define APIs for a graph data structure
69  * >> T: type of vertices

```

```

70     */
71     template<class T>
72     class IGraph{
73     public:
74         virtual ~IGraph(){};
75
76         /*
77         add (T vertex):
78         add a vertex to graph
79         */
80         virtual void add(T vertex)=0;
81         virtual void remove(T vertex)=0;
82         virtual bool contains(T vertex)=0;
83
84         /*
85         connect(T from, T to, float weight):
86         connect 2 vertexes (from -> to) with weight
87         */
88         virtual void connect(T from, T to, float weight=0)=0;
89         virtual void disconnect(T from, T to)=0;
90         virtual float weight(T from, T to)=0;
91
92         virtual DLinkedList<T> getOutwardEdges(T from)=0;
93         virtual DLinkedList<T> getInwardEdges(T to)=0;
94
95         virtual int size()=0;
96         virtual bool empty()=0;
97         virtual void clear()=0;
98
99         /*
100        inDegree(T vertex):
101        find the in degree of the vertex
102        */
103        virtual int inDegree(T vertex)=0;
104
105        /*
106        outDegree(T vertex):
107        find the out degree of the vertex
108        */
109        virtual int outDegree(T vertex)=0;
110
111        virtual DLinkedList<T> vertices()=0;
112        virtual bool connected(T from, T to)=0;
113
114        virtual string toString()=0;
115    };
116
117    /*
118    * Path: model a path on graphs
119    * >> a path = sequence of vertices,
120    *     -> stored in: "path" (DLinkedList<T>)
121    *     -> its cost: stored in "cost" (float)
122    *
123    */
124    template<class T>
125    class Path{
126    private:
127        DLinkedList<T> path;
128        float cost;
129    public:
130        Path(){
131            cost = 0;
132        }
133        DLinkedList<T>& getPath(){
134            return this->path;
135        }
136        float getCost(){
137            return cost;
138        }

```

```

139     void setCost(float cost){
140         this->cost = cost;
141     }
142
143     //////////////////////////////////////
144     void add(T item){
145         this->path.add(item);
146     }
147     string toString(string (*item2str) (T&)=0){
148         stringstream os;
149         os << this->path.toString(item2str)
150             << ", cost: " << this->cost;
151         return os.str();
152     }
153 };
154
155 /*
156  * IFinder: the path finder, contains searching algorithms on graph
157  *
158  */
159 template<class T>
160 class IFinder{
161     virtual DLinkedList<Path<T>> dijkstra(IGraph<T>* pGraph, T start)=0;
162 };
163
164 #endif /* IGRAPH_H */
165
166 /*
167  * To change this license header, choose License Headers in Project Properties.
168  * To change this template file, choose Tools | Templates
169  * and open the template in the editor.
170  */
171
172 /*
173  * File: IMap.h
174  * Author: LTSACH
175  *
176  * Created on 22 August 2020, 21:53
177  */
178
179 #ifndef IMAP_H
180 #define IMAP_H
181
182 #include "list/DLinkedList.h"
183 #include <string>
184 using namespace std;
185
186
187 class KeyNotFound: public std::exception{
188 private:
189     string desc;
190 public:
191     KeyNotFound(string desc){
192         this->desc = desc;
193     }
194     const char * what () const throw (){
195         stringstream os;
196         os << this->desc;
197         return os.str().c_str();
198     }
199 };
200
201
202 template<class K, class V>
203 struct Pair{
204     K key;
205     V value;
206     Pair(K key, V value){
207         this->key = key;

```

```

208         this->value = value;
209     }
210     Pair(const Pair& pair){
211         this->key = pair.key;
212         this->value = pair.value;
213     }
214     Pair& operator=(const Pair& pair){
215         this->key = pair.key;
216         this->value = pair.value;
217     }
218 };
219
220 template<class K, class V>
221 class IMap {
222 public:
223     virtual ~IMap(){};
224     //
225     /*
226     put(K key, V value):
227     if key is not in the map:
228         + add a mapping key->value to the map
229         + return value
230     else:
231         + associate key with the new value (passed as parameter)
232         + return the old value
233     */
234     virtual V put(K key, V value)=0;
235
236     /*
237     get(K key):
238     if key in the map: return the associated value
239     else: KeyNotFound exception thrown
240
241     */
242     virtual V& get(K key)=0;
243
244     /*
245     remove(K key):
246     if key is in the map: remove it from the map, and return the associated value
247     else: KeyNotFound exception thrown
248
249     >> deleteKeyInMap(K key): delete key stored in map; in cases, K is a pointer type
250     */
251     virtual V remove(K key, void (*deleteKeyInMap)(K)=0)=0;
252
253     /*
254     remove(K key, V value):
255     if there is a mapping key->value in the map: remove it and return true
256     else: return false
257
258     >> deleteKeyInMap(K key): delete key stored in map; in cases, K is a pointer type
259     >> deleteValueInMap(V value): delete key stored in map; in cases, V is a pointer type
260     */
261     virtual bool remove(K key, V value, void (*deleteKeyInMap)(K)=0, void
(*deleteValueInMap)(V)=0)=0;
262
263     /*
264     containsKey(K key):
265     if key is in the map: return true
266     else: return false
267     */
268     virtual bool containsKey(K key)=0;
269
270     /*
271     containsKey(V value):
272     if value is in the map: return true
273     else: return false
274     */
275     virtual bool containsValue(V value)=0;

```

```

276
277     /*
278     empty():
279     return true if the map is empty
280     else: return false
281     */
282     virtual bool empty()=0;
283
284     /*
285     size():
286     return number of pairs key->value
287     */
288     virtual int size()=0;
289
290     /*
291     clear():
292     clear all pairs key->value in the map
293     */
294     virtual void clear() = 0;
295
296     /*
297     toString():
298     return a string representing the map
299
300     >> key2str(K& key): convert key to string; if not supplied then K must support
301     extraction operator (<<)
302     >> value2str(V& value): convert value to string; if not supplied then V must support
303     extraction operator (<<)
304     */
305     virtual string toString(string (*key2str)(K&)=0, string (*value2str)(V&)=0 )=0;
306
307     /*
308     * keys(): return a set of keys stored in the map
309     */
310     virtual DLinkedList<K> keys()=0;
311
312     /*
313     * values(): return a set of values stored in the map
314     */
315     virtual DLinkedList<V> values()=0;
316
317     /*
318     * clashes(): return a list containing the collision count for each address
319     */
320     virtual DLinkedList<int> clashes()=0;
321 };
322 #endif /* IMAP_H */
323
324

```