```cpp
//
// Created by along on 17-11-26.
//

#include "Graph.h"
#include <stack>
#include <queue>

using namespace std;

void Graph::clone(const Graph &graph) {
    reset(graph.vexCount());

    function<bool(unsigned long, unsigned long)> func = [&]()
    unsigned long source, unsigned long sink) {
        this->addEdge(source, sink);
        return true;
    };

    unsigned long verNum = graph.vexCount();
    for (unsigned long ver = 0; ver != verNum; ++ver) {
        graph.foreach(ver, func);
    }
}

unsigned long Graph::edgeCount() const {
    unsigned long count = 0;
    for (unsigned long vex = 0; vex != vexCount(); ++vex)
        count += outDegree(vex);
    return count;
}

Graph &Graph::operator=(const Graph &rhs) {
```

```
34      this->clone(rhs);
35      return *this;
36  }
37  Graph::Graph(const Graph &rhs) {
38      this->clone(rhs);
39  }
40  void Graph::addEdge(unsigned long, unsigned long) {
41      ++edgeNum;
42  }
43  void Graph::delEdge(unsigned long, unsigned long) {
44      --edgeNum;
45  }
46  unsigned long Graph::vexCount() const {
47      return vexNum;
48  }
49  void Graph::reset(unsigned long vexNum) {
50      clear();
51      this->vexNum = vexNum;
52      edgeNum = 0;
53  }
54  void Graph::clear() {
55      vexNum = 0;
56      edgeNum = 0;
57  }
58  unsigned long Graph::outDegree(unsigned long source) const {
59      unsigned long outDegree = 0;
60      function<bool(unsigned long, unsigned long)> func = [&]()
61      unsigned long src, unsigned long dst) {
62          ++outDegree;
63          return true;
64      };
65      foreach(source, func);
66      return outDegree;
```

```
67 }
68
69 unsigned long Graph::inDegree(unsigned long source) const {
70     unsigned long inDegree = 0;
71     function<bool(unsigned long, unsigned long)> func = [&](
72     unsigned long src, unsigned long dst) {
73         if (dst == source)
74             ++inDegree;
75         return true;
76     };
77     for (unsigned long i = 0; i != vexNum; ++i)
78         foreach(i, func);
79     return inDegree;
80 }
81
82 bool Graph::hasEdge(unsigned long source, unsigned long sink)
83 const {
84     bool hasEdge = false;
85     function<bool(unsigned long, unsigned long)> func = [&](
86     unsigned long src, unsigned long dst) {
87         if (dst == sink) {
88             hasEdge = true;
89             return false;
90         }
91         return true;
92     };
93     foreach(source, func);
94     return hasEdge;
95 }
96
97 void Graph::DFSR(std::function<void(unsigned long)> &visit)
98 const {
```

```
99      vector<bool> visited(vexNum, false);
100     function<bool(unsigned long, unsigned long)> func = [&](
101     unsigned long src, unsigned long dst) {
102         if (!visited[src]) {
103             visit(src);
104             visited[src] = true;
105         }
106         if (!visited[dst]) {
107             visit(dst);
108             visited[dst] = true;
109             foreach(dst, func);
110         }
111         return true;
112     };
113
114     for (unsigned long vex = 0; vex != vexNum; ++vex) {
115         if (visited[vex])
116             continue;
117         if (outDegree(vex) == 0) {
118             visit(vex);
119             visited[vex] = true;
120         } else
121             foreach(vex, func);
122     }
123 }
124
125 void Graph::DFS(std::function<void(unsigned long)> &visit)
126 const {
127     vector<bool> visited(vexNum, false);
128     stack<unsigned long> vexStack;
129
130     unsigned long finded = 0;
```

```
131     // 查找未访问过的节点
132     function<bool(unsigned long, unsigned long)> find = [&](
133     unsigned long src, unsigned long dst) {
134         if (!visited[dst]) {
135             finded = dst;
136             return false;
137         }
138 
139         return true;
140     };
141 
142     for (unsigned long vex = 0; vex != vexNum; ++vex) {
143         if (visited[vex])
144             continue;
145         visit(vex);
146         visited[vex] = true;
147         if (outDegree(vex) != 0) {
148             vexStack.push(vex);
149             while (!vexStack.empty()) {
150                 auto top = finded = vexStack.top();
151                 foreach(top, find);
152 
153                 if (finded != top) {
154                     visit(finded);
155                     visited[finded] = true;
156                     vexStack.push(finded);
157                 } else {
158                     vexStack.pop();
159                 }
160             }
161         }
162     }
163 }
```

```
164
165 void Graph::BFS(std::function<void(unsigned long)> &visit)
const {
166     vector<bool> visited(vexNum, false);
167     queue<unsigned long> vexQueue;
168
169     function<bool(unsigned long, unsigned long)> find = [&](
170     unsigned long src, unsigned long dst) {
171         if (!visited[dst]) {
172             visit(dst);
173             visited[dst] = true;
174             vexQueue.push(dst);
175         }
176         return true;
177     };
178
179     for (unsigned long vex = 0; vex != vexNum; ++vex) {
180         if (visited[vex])
181             continue;
182         visit(vex);
183         visited[vex] = true;
184         if (outDegree(vex) != 0) {
185             vexQueue.push(vex);
186             while (!vexQueue.empty()) {
187                 auto front = vexQueue.front();
188                 vexQueue.pop();
189                 foreach(front, find);
190             }
191         }
192     }
193 }
194
195
```

```
196 void Graph::DFS(Graph &DFSTree, unsigned long vex, std::
197 function<void(unsigned long)> &visit) const {
198     vector<bool> visited(vexNum, false);
199     stack<unsigned long> vexStack;
200 
201     unsigned long finded = 0;
202     // 查找未访问过的节点
203     function<bool(unsigned long, unsigned long)> find = [&](
204     unsigned long src, unsigned long dst) {
205         if (!visited[dst]) {
206             finded = dst;
207             return false;
208         }
209         return true;
210     };
211 
212     visit(vex);
213     visited[vex] = true;
214     if (outDegree(vex) != 0) {
215         vexStack.push(vex);
216         while (!vexStack.empty()) {
217             auto top = finded = vexStack.top();
218             foreach(top, find);
219 
220             if (finded != top) {
221                 visit(finded);
222                 visited[finded] = true;
223                 DFSTree.addEdge(top, finded);
224                 vexStack.push(finded);
225             } else {
226                 vexStack.pop();
227             }
228         }
```

```
229     }
230 }
231
232 void Graph::DFSR(Graph &DFSTree, unsigned long vex, std::
233 function<void(unsigned long)> &visit) const {
234     vector<bool> visited(vexNum, false);
235     function<bool(unsigned long, unsigned long)> func = [&](
236     unsigned long src, unsigned long dst) {
237         if (!visited[src]) {
238             visit(src);
239             visited[src] = true;
240         }
241         if (!visited[dst]) {
242             visit(dst);
243             visited[dst] = true;
244             DFSTree.addEdge(src, dst);
245             foreach(dst, func);
246         }
247         return true;
248     };
249
250     if (outDegree(vex) == 0) {
251         visit(vex);
252         visited[vex] = true;
253     } else
254         foreach(vex, func);
255
256 }
257
258 void Graph::BFS(Graph &BFSTree, unsigned long vex, std::
259 function<void(unsigned long)> &visit) const {
260     vector<bool> visited(vexNum, false);
261     queue<unsigned long> vexQueue;
```

```
262
263     function<bool(unsigned long, unsigned long)> find = [&](
264     unsigned long src, unsigned long dst) {
265         if (!visited[dst]) {
266             visit(dst);
267             visited[dst] = true;
268             BFSTree.addEdge(src, dst);
269             vexQueue.push(dst);
270         }
271         return true;
272     };
273
274     visit(vex);
275     visited[vex] = true;
276     if (outDegree(vex) != 0) {
277         vexQueue.push(vex);
278         while (!vexQueue.empty()) {
279             auto front = vexQueue.front();
280             vexQueue.pop();
281             foreach(front, find);
282         }
283     }
284 }
285
286 void Graph::reset() {
287     reset(vexNum);
288 }
289
290 void Graph::printDot(std::ostream &out) {
291     out << "digraph mGraph{" << endl;
292     for (unsigned long i = 0; i != vexNum; ++i)
293         out << "Node" << i << "[label = \"" << i << "\"];"
<<
```

```
294         endl;
295
296     function<bool(unsigned long, unsigned long)> func = [&](
297     unsigned long src, unsigned long dst) {
298         out << "Node" << src << " -> " << "Node" << dst
<< ";
299         " << endl;
300         return true;
301     };
302     for (unsigned long i = 0; i != vexNum; ++i)
303         foreach(i, func);
304
305     out << "}" << endl;
306 }
307
308 void Graph::resetFromStream(istream &theStream) {
309     unsigned long vexNum;
310     theStream >> vexNum;
311     reset(vexNum);
312
313     unsigned long src, dst;
314     while (theStream >> src >> dst)
315         addEdge(src, dst);
316 }
317
318 GraphT::GraphT(unsigned long n) : Graph(n) {
319     for (int i = 0; i != n; ++i) {
320         vexes.push_back({0, {}});
321     }
322 }
323
324 void GraphT::addEdge(unsigned long source, unsigned long 
325 sink) {
```

```
326     if (!hasEdge(source, sink)) {
327         Graph::addEdge(source, sink);
328         auto &src = vexes[source];
329         auto &dst = vexes[sink];
330         src.adjVex.push_front(sink);
331 
332         ++src.out;
333         ++dst.in;
334     }
335 }
336 
337 unsigned long GraphT::vexCount() const {
338     return vexes.size();
339 }
340 
341 unsigned long GraphT::outDegree(unsigned long source) const
{
342     return vexes[source].out;
343 }
344 
345 bool GraphT::hasEdge(unsigned long source, unsigned long )
346 sink) const {
347     for (auto &adjVex:vexes[source].adjVex) {
348         if (adjVex == sink)
349             return true;
350     }
351     return false;
352 }
353 void GraphT::foreach(unsigned long source, std::function<)
354 bool(unsigned long, unsigned long)> &func) const {
355     for (auto &sink:vexes[source].adjVex) {
356         if (!func(source, sink))
357             break;
```

```
358     }
359 }
360
361 void GraphT::clear() {
362     Graph::clear();
363     vexes.clear();
364 }
365
366 unsigned long GraphT::inDegree(unsigned long source) const {
367     return vexes[source].in;
368 }
369
370 void GraphT::reset(unsigned long vexNum) {
371     Graph::reset(vexNum);
372     for (int i = 0; i != vexNum; ++i) {
373         vexes.push_back({0, {}});
374     }
375 }
376 void GraphT::delEdge(unsigned long source, unsigned long ↩
377 sink) {
378     if (hasEdge(source, sink)) {
379         Graph::delEdge(source, sink);
380         auto &src = vexes[source];
381         auto &dst = vexes[sink];
382
383         src.adjVex.remove(sink);
384
385         --src.out;
386         --dst.in;
387     }
388 }
389 GraphT::GraphT(const Graph &rhs) : Graph(0) {
390     *(dynamic_cast<Graph *>(this)) = rhs;
```

```
391 }
392 unsigned long GraphT::edgeCount() const {
393     return Graph::edgeCount();
394 }
395 void GraphT::reset() {
396     Graph::reset();
397 }
398
399 GraphM::GraphM(unsigned long n) : Graph(n) {
400     for (unsigned long i = 0; i != n; ++i) {
401         vexes.emplace_back(n, false);
402     }
403 }
404
405 void GraphM::clear() {
406     Graph::clear();
407     vexes.clear();
408 }
409
410 void GraphM::addEdge(unsigned long source, unsigned long )
411 sink) {
412     Graph::addEdge(source, sink);
413     vexes[source][sink] = true;
414 }
415
416 unsigned long GraphM::vexCount() const {
417     return vexes.size();
418 }
419
420 unsigned long GraphM::outDegree(unsigned long source) const
{
421 #ifdef USE_BOOST_LIB
422     return vexes[source].count();
```

```
423 #else
424     unsigned long count = 0;
425     for (auto out:vexes[source])
426         count += out;
427     return count;
428 #endif
429 }
430
431 bool GraphM::hasEdge(unsigned long source, unsigned long ↩
432 sink) const {
433     return vexes[source][sink];
434 }
435
436 void GraphM::foreach(unsigned long source, std::function<↩
437 bool(unsigned long, unsigned long)> &func) const {
438     for (unsigned long adjVec = 0; adjVec != vexCount(); ++↩
439     adjVec)
440         if (hasEdge(source, adjVec))
441             if (!func(source, adjVec))
442                 break;
443
444 }
445
446 unsigned long GraphM::inDegree(unsigned long source) const {
447     unsigned long count = 0;
448     for (unsigned long i = 0; i != vexCount(); ++i)
449         count += vexes[i][source];
450     return count;
451 }
452
453 void GraphM::reset(unsigned long vexNum) {
454     Graph::reset(vexNum);
455     for (unsigned long i = 0; i != vexNum; ++i) {
```

```
456         vexes.emplace_back(vexNum, false);
457     }
458 }
459
460 void GraphM::delEdge(unsigned long source, unsigned long ↩
461 sink) {
462     Graph::delEdge(source, sink);
463     vexes[source][sink] = false;
464 }
465
466 GraphM::GraphM(const Graph &rhs) : Graph(0) {
467     *(dynamic_cast<Graph *>(this)) = rhs;
468 }
469
470 unsigned long GraphM::edgeCount() const {
471     return Graph::edgeCount();
472 }
473 void GraphM::reset() {
474     Graph::reset();
475 }
476
477 GraphL::GraphL(unsigned long n) : Graph(n) {
478     for (unsigned long i = 0; i != n; ++i)
479         vexes.emplace_back(VexNode());
480 }
481
482 void GraphL::addEdge(unsigned long source, unsigned long ↩
483 sink) {
484     if (!hasEdge(source, sink)) {
485         Graph::addEdge(source, sink);
486         auto &src = vexes[source];
487         auto &dst = vexes[sink];
488
```

```
489         ++src.out;
490         ++dst.in;
491 
492         auto *newArc = new ArcBox(source, sink, nullptr, ⤸
493         nullptr);
494         if (src.firstOut == nullptr)
495             src.firstOut = newArc;
496         else {
497             ArcBox **findArc = &src.firstOut;
498             while ((*findArc) != nullptr && (*findArc)->⤸
499             tailVex < sink)
500                 findArc = &(*findArc)->hLink;
501             newArc->hLink = *findArc;
502             *findArc = newArc;
503         }
504 
505         if (dst.firstIn == nullptr)
506             dst.firstIn = newArc;
507         else {
508             ArcBox **findArc = &dst.firstIn;
509             while ((*findArc) != nullptr && (*findArc)->⤸
510             headVex < source)
511                 findArc = &(*findArc)->tLink;
512             newArc->tLink = *findArc;
513             *findArc = newArc;
514         }
515     }
516 }
517 
518 void GraphL::delEdge(unsigned long source, unsigned long ⤸
519 sink) {
520     if (hasEdge(source, sink)) {
521         Graph::delEdge(source, sink);
```

```
522         auto &src = vexes[source];
523         auto &dst = vexes[sink];
524 
525         --src.out;
526         --dst.in;
527 
528         ArcBox **head = &src.firstOut, **tail = &dst.firstIn;
529         while ((*tail)->headVex != source)
530             tail = &(*tail)->tLink;
531         *tail = (*tail)->tLink;
532 
533         while ((*head)->tailVex != sink)
534             head = &(*head)->hLink;
535         // ❯
536         废物利用，暂时保存一下要释放的内存 ❯
537         Ğ 针
538         *tail = *head;
539         *head = (*head)->hLink;
540         delete *tail;
541     }
542 }
543 
544 unsigned long GraphL::vexCount() const {
545     return vexes.size();
546 }
547 
548 unsigned long GraphL::outDegree(unsigned long source) const
{
549     return vexes[source].out;
550 }
551 
552 unsigned long GraphL::inDegree(unsigned long source) const {
553     return vexes[source].in;
```

```
554 }
555
556 bool GraphL::hasEdge(unsigned long source, unsigned long
557 sink) const {
558     for (ArcBox *arc = vexes[source].firstOut; arc !=
559     nullptr; arc = arc->hLink)
560         if (arc->tailVex == sink)
561             return true;
562     return false;
563 }
564
565 void GraphL::foreach(unsigned long source, std::function<
566 bool(unsigned long, unsigned long)> &func) const {
567     for (const ArcBox *arc = vexes[source].firstOut; arc !=
568     nullptr; arc = arc->hLink)
569         if (!func(source, arc->tailVex))
570             break;
571 }
572
573 void GraphL::clear() {
574     Graph::clear();
575     for (auto &vex:vexes) {
576         while (vex.firstOut != nullptr) {
577             auto curr = vex.firstOut;
578             vex.firstOut = curr->tLink;
579             delete curr;
580         }
581     }
582     vexes.clear();
583 }
584
585 void GraphL::reset(unsigned long vexNum) {
```

```
586     Graph::reset(vexNum);
587     for (unsigned long i = 0; i != vexNum; ++i)
588         vexes.emplace_back(VexNode());
589 }
590
591 GraphL::GraphL(const Graph &rhs) : Graph(0) {
592     *(dynamic_cast<Graph *>(this)) = rhs;
593 }
594
595 unsigned long GraphL::edgeCount() const {
596     return Graph::edgeCount();
597 }
598
599 void GraphL::foreachIn(unsigned long dst, std::function<bool(
600 unsigned long, unsigned long)> &func) const {
601     for (const ArcBox *arc = vexes[dst].firstIn; arc !=
602     nullptr; arc = arc->tLink)
603         if (!func(arc->headVex, dst))
604             break;
605 }
606 void GraphL::reset() {
607     Graph::reset();
608 }
609
610
```