```
 1: #include <stdlib.h>
 2: #include <stdio.h>
 3: #include <math.h>
 4:
 5: #include "GeraGrafo.h"
 6:
 7: void imprimeErro ( char* exename )
 8: {
 9:     printf( "Gerador de grafos. Uso: %s [nvertex tipo(0|1) [debuglevel(0-2)]]\n", exename );
10:     printf( "Se chamado sem parametros, gera 20 sequencias, 10 de cada tipo.\n" );
11:     printf( "Tipo 0: p = 0.2\n" );
12:     printf( "Tipo 1: p = 1 / ( 2 * ( n ^ 0.5 ) )\n" );
13: }
14:
15: int main ( int argc, char ** argv )
16: {
17:     int debug = 0;
18:     int nVertex = 0;
19:
20:     if ( argc < 3 )
21:     {
22:         for ( int c = 0; c < NSEQS; c++ )
23:         {
24:             for ( int i = 1; i <= NGRAFOS_POR_SEQ; i++ )
25:             {
26:                 char path[ 256 ];
27:
28:                 sprintf( path, "grafo_p%d_%02d.grafo", c, i );
29:
30:                 FILE * f = fopen( path , "w" );
31:
32:                 int n = 10 * ( ( int ) pow( 2, i ) );
33:
34:                 // gera grafo com debug = 1 (so para dizer o que esta fazendo, nao depurar)
35:                 GeraGrafo( f, n, c, 1 );
36:
37:                 fclose( f );
38:             }
39:         }
40:     }
41:     else
42:     {
43:         nVertex = ( int ) ( atoi( argv[1] ) );
44:
45:         int type = atoi( argv[2] );
46:         if ( type > 1 )
47:         {
48:             printf( "Tipo invalido.\n\n" );
49:             imprimeErro( argv[0] );
50:
51:             exit( 2 );
52:         }
53:
54:         if ( nVertex <= 0 || nVertex > 100000 )
55:         {
56:             printf( "Numero de vertices invalido (%d).\n\n", nVertex );
57:             imprimeErro( argv[0] );
58:
59:             exit( 3 );
60:         }
61:
62:         if ( argc > 3 )
63:         {
64:             debug = atoi( argv[3] );
65:         }
66:
67:         GeraGrafo( stdout, nVertex, type, debug );
68:     }
69:
70:     return 1;
71: }
```

```
 1: /*
 2:  * GeraGrafo.h
 3:  *
 4:  *  Created on: May 28, 2011
 5:  *      Author: darioandrade
 6:  */
 7:
 8: #ifndef GERAGRAFO_H_
 9: #define GERAGRAFO_H_
10:
11: void GeraGrafo( FILE* f, int nVertex, int type, int debug );
12:
13:
14: #define NSEQS                   2
15: #define NGRAFOS_POR_SEQ         10
16:
17: #endif /* GERAGRAFO_H_ */
```

```
 1: /*
 2:  * GeraGrafo.cpp
 3:  *
 4:  *  Created on: May 28, 2011
 5:  *      Author: darioandrade
 6:  */
 7:
 8: #include <stdlib.h>
 9: #include <math.h>
10: #include <ctime>
11:
12: #include "AdjacencyList.h"
13:
14: #define P_TIPO_0 0.2
15:
16: double randd ( )
17: {
18:     return ( ( double ) rand( ) ) / RAND_MAX;
19: }
20:
21: void GeraGrafo ( FILE* f, int nVertex, int type, int debug )
22: {
23:     // alimentando a semente randomica
24:     srand( time( NULL ) );
25:
26:     // calculando p
27:     double p = ( type == 0 ) ? P_TIPO_0 : ( 1.0 / ( 2.0 * sqrt( nVertex ) ) );
28:
29:     if ( debug > 0 )
30:     {
31:         fprintf( stderr,
32:                 "Gerando %d vertices de grafo nao direcionado tipo %d (p = %.7f)\n",
33:                 nVertex, type, p );
34:     }
35:
36:     // criando o grafo
37:     AdjacencyList graph( nVertex );
38:
39:     // percorrendo do primeiro vertice ate o penultimo (zero based list)
40:     for ( int i = 0; i < nVertex - 1; i++ )
41:     {
42:         if ( debug > 1 )
43:         {
44:             fprintf( stderr, "  vertice %d: ", i );
45:         }
46:
47:         for ( int j = i + 1; j < nVertex; j++ )
48:         {
49:             double x = randd( );
50:
51:             if ( debug > 1 )
52:             {
53:                 fprintf( stderr, " %.4f", x );
54:             }
55:
56:             // should we cast an edge to j?
57:             if ( x < p )
58:             {
59:                 if ( debug > 1 )
60:                 {
61:                     fprintf( stderr, "Y" );
62:                 }
63:
64:                 graph.addEdge( i, j );
65:             }
66:             else
67:             {
68:                 if ( debug > 1 )
69:                 {
70:                     fprintf( stderr, " " );
71:                 }
72:             }
73:         }
74:
75:         if ( debug > 1 )
76:         {
77:             fprintf( stderr, "\n" );
78:         }
79:     }
80:
81:     graph.write( f );
82: }
```

```
 1: /*
 2:  * cobertura.cpp
 3:  *
 4:  *  Created on: Jun 11, 2011
 5:  *      Author: darioandrade
 6:  */
 7:
 8: #include <sys/timeb.h>
 9: #include <time.h>
10: #include <string.h>
11: #include <stdio.h>
12: #include <cstdlib>
13: #include "AlgoritmoCoberturaGulosa.h"
14: #include "DegreeVectorAdjacencyList.h"
15: #include "VertexVectorAdjacencyList.h"
16: #include "DegreeHeapAdjacencyList.h"
17:
18: #include "GeraGrafo.h"
19:
20: typedef long unsigned int ltime;
21:
22: static ltime getMilliCount ( )
23: {
24:     timeb tb;
25:     ftime( &tb );
26:     time_t nCount = tb.millitm + ( tb.time * 1000 );
27:     return nCount;
28: }
29:
30: /*
31: static ltime getMicroCount ( )
32: {
33:     timeval tv;
34:     timezone tz;
35:     tm *tm;
36:     gettimeofday(&tv, &tz);
37:     return tv.tv_usec;
38: }
39: */
40:
41: void FacaTarefa ( AdjacencyList* pGrafo, int debug )
42: {
43:     AdjacencyList& grafo = *pGrafo;
44:
45:     AlgoritmoCoberturaGulosa guloso;
46:
47:     std::list<int> listaDaCobertura;
48:
49:     if ( debug >= 1 )
50:     {
51:         fprintf( stderr, "Calculando cobertura para o grafo com %d vertices\n", grafo.GetSize( ) );
52:     }
53:
54:     guloso.CalculateCobertura( listaDaCobertura, grafo, debug );
55:
56:     if ( debug >= 1 )
57:     {
58:         fprintf( stderr, "Cobertura para o grafo (nvertex: %d) tem %d vertices:\n", grafo.GetSize( ),
59:                 ( int ) listaDaCobertura.size( ) );
60:     }
61:
62:     for ( std::list<int>::iterator it = listaDaCobertura.begin( ); it != listaDaCobertura.end( ); it++ )
63:     {
64:         fprintf( stdout, "%d ", *it );
65:     }
66:
67:     fprintf( stdout, "\n" );
68: }
69:
70: AdjacencyList * EscolhaGrafo ( int tarefa )
71: {
72:     AdjacencyList * pGrafo = NULL;
73:
74:     switch (tarefa)
75:     {
76:         case 3:
77:             pGrafo = new DegreeVectorAdjacencyList( );
78:             break;
79:
80:         case 4:
81:             pGrafo = new DegreeHeapAdjacencyList( );
82:             break;
83:
84:         case 5:
85:             pGrafo = new VertexVectorAdjacencyList( );
86:             break;
87:     }
88:
89:     return pGrafo;
90: }
91:
92: void Benchmark ( int debug )
93: {
94:     struct BenchmarkData {
```

```
 95:         char sFilename[ 256 ];
 96:         int graphtype;
 97:         ltime loadtime, processingtime;
 98:         int algorithm;
 99:         int nvertex;
100:         int nedges;
101:         int ncobertura;
102:     };
103:
104:     BenchmarkData results[ 3 ][ NSEQS ][ NGRAFOS_POR_SEQ ];
105:
106:     for ( int algo = 3; algo <= 5; algo++ )
107:     {
108:         AdjacencyList * pGrafo = EscolhaGrafo( algo );
109:
110:         for ( int c = 0; c < NSEQS; c++ )
111:         {
112:             for ( int i = 1; i <= NGRAFOS_POR_SEQ; i++ )
113:             {
114:                 char path[256];
115:                 ltime deltams;
116:                 BenchmarkData & data = results[ algo - 3 ][ c ][ i - 1 ];
117:
118:                 sprintf( path, "grafo_p%d_%02d.grafo", c, i );
119:
120:                 if ( debug >= 1 )
121:                 {
122:                     fprintf( stderr, "Lendo grafo %s\n", path );
123:                 }
124:
125:                 data.algorithm = algo;
126:                 data.graphtype = c;
127:                 strcpy( data.sFilename, path );
128:
129:                 FILE * f = fopen( path, "r" );
130:
131:                 deltams = getMilliCount( );
132:
133:                 pGrafo->read( f );
134:
135:                 deltams = getMilliCount( ) - deltams;
136:
137:                 fclose( f );
138:
139:                 if ( debug >= 1 )
140:                 {
141:                     fprintf( stderr, "Grafo com %d vertices lido em %lu ms. Calculando cobertura com algoritmo %d\n",
142:                             pGrafo->GetSize( ), deltams, algo );
143:                 }
144:
145:                 data.loadtime = deltams;
146:                 data.nvertex = pGrafo->GetSize( );
147:                 data.nedges = pGrafo->GetEdges( );
148:
149:                 deltams = getMilliCount( );
150:
151:                 std::list<int> listaDaCobertura;
152:
153:                 AlgoritmoCoberturaGulosa guloso;
154:
155:                 guloso.CalculateCobertura( listaDaCobertura, *pGrafo, debug );
156:
157:                 deltams = getMilliCount( ) - deltams;
158:
159:                 if ( debug >= 1 )
160:                 {
161:                     fprintf( stderr, "Cobertura possui %d vertices, calculada em %lu ms\n",
162:                             ( int ) listaDaCobertura.size( ), deltams );
163:                 }
164:
165:                 data.ncobertura = listaDaCobertura.size();
166:                 data.processingtime = deltams;
167:
168:                 fprintf( stdout, "%d,%d,%s,%d,%d,%d,%lu,%lu\n",
169:                         data.algorithm,
170:                         data.graphtype,
171:                         data.sFilename,
172:                         data.nvertex,
173:                         data.nedges,
174:                         data.ncobertura,
175:                         data.loadtime,
176:                         data.processingtime );
177:
178:                 fflush( stdout );
179:             } // end for ngrafos por seq
180:         } // end for seqs
181:
182:         delete pGrafo;
183:     } // end for algos
184: }
185:
186: int main ( int argc, char ** argv )
187: {
188:     if ( argc < 2 )
```

```
189:     {
190:         fprintf( stderr, "Cobertura. Uso: %s (benchmark|(filename ([3|4|5]))) [debug]\n", argv[0] );
191:         exit( 1 );
192:     }
193:
194:     int debug = 1;
195:
196:     if ( strcmp( "benchmark", argv[1] ) == 0 )
197:     {
198:         if ( argc > 2 )
199:         {
200:             debug = atoi( argv[2] );
201:         }
202:
203:         Benchmark( debug );
204:     }
205:     else
206:     {
207:         int tarefa = atoi( argv[2] );
208:
209:         if ( argc > 3 )
210:         {
211:             debug = atoi( argv[3] );
212:         }
213:
214:         AdjacencyList * pGrafo = EscolhaGrafo( tarefa );
215:
216:         char * sFilename = argv[1];
217:
218:         if ( debug >= 1 )
219:         {
220:             fprintf( stderr, "Lendo grafo %s:\n", sFilename );
221:         }
222:
223:         FILE * f = fopen( sFilename, "r" );
224:
225:         pGrafo->read( f, debug );
226:
227:         fclose( f );
228:
229:         FacaTarefa( pGrafo, debug );
230:
231:         delete pGrafo;
232:     }
233:
234:     return 0;
235: }
236:
```

```
 1: /*
 2:  * AlgoritmoCoberturaGulosa.h
 3:  *
 4:  *  Created on: Jun 11, 2011
 5:  *      Author: darioandrade
 6:  */
 7:
 8: #include <list>
 9:
10: #include "AdjacencyList.h"
11:
12: #ifndef ALGORITMOCOBERTURAGULOSA_H_
13: #define ALGORITMOCOBERTURAGULOSA_H_
14:
15: class AlgoritmoCoberturaGulosa
16: {
17: public:
18:     AlgoritmoCoberturaGulosa ( );
19:     virtual ~AlgoritmoCoberturaGulosa ( );
20:
21:     void CalculateCobertura(std::list< int > & listCobertura, AdjacencyList& grafo, int debug );
22:
23: private:
24: };
25:
26: #endif /* ALGORITMOCOBERTURAGULOSA_H_ */
```

```
 1: /*
 2:  * AlgoritmoCoberturaGulosa.cpp
 3:  *
 4:  *  Created on: Jun 11, 2011
 5:  *      Author: darioandrade
 6:  */
 7:
 8: #include "AlgoritmoCoberturaGulosa.h"
 9:
10: AlgoritmoCoberturaGulosa::AlgoritmoCoberturaGulosa ( )
11: {
12:
13:
14: }
15:
16: AlgoritmoCoberturaGulosa::~AlgoritmoCoberturaGulosa ( )
17: {
18:
19: }
20:
21: void AlgoritmoCoberturaGulosa::CalculateCobertura(std::list< int > & listCobertura, AdjacencyList & grafo, int debug )
22: {
23:
24:     while ( grafo.HasEdge( ) )
25:     {
26:         if ( debug >= 2 )
27:         {
28:             fprintf( stderr, "removendo vertice de maior grau e atualizando vizinhos\n" );
29:         }
30:
31:         int vertex = grafo.RemoveHighestDegreeVertex( debug );
32:
33:         if ( debug >= 2 )
34:         {
35:             fprintf( stderr, "vertice de maior grau: %d\n", vertex );
36:         }
37:
38:         listCobertura.push_back( vertex );
39:     }
40: }
```

```
 1: /*
 2:  * AdjacencyList.h
 3:  *
 4:  *  Created on: May 28, 2011
 5:  *      Author: darioandrade
 6:  */
 7:
 8: #ifndef ADJACENCYLIST_H_
 9: #define ADJACENCYLIST_H_
10:
11: #include <stdio.h>
12: #include "List.h"
13:
14: class AdjacencyList {
15:
16: public:
17:
18:     AdjacencyList( int nVertex );
19:     virtual ~AdjacencyList( );
20:
21:     virtual void addEdge( int iVertex, int jVertex, bool bUpdateNeighbor = true, bool bIncEdge = true );
22:     void write( FILE * f = NULL );
23:     void read( FILE * f = NULL, int debug = 0 );
24:
25:     virtual bool HasEdge( ) const;
26:     virtual int RemoveHighestDegreeVertex( int debug ) { return -1; }
27:
28:     int GetSize() const { return m_nVertex; }
29:     int GetEdges() const { return m_nEdges; }
30:
31: //  int GetDegree(int iVertex) { return (int)m_arrAdjLists[iVertex]->size(); }
32:
33:
34: protected:
35:     AdjacencyList( );
36:     virtual void Allocate( int nVertex );
37:     virtual void updateData(){ }
38:
39:     int                     m_nVertex;
40:     List **                 m_arrAdjLists;
41:
42:     int                     m_nEdges;
43: };
44:
45: #endif /* ADJACENCYLIST_H_ */
```

```
 1: /*
 2:  * AdjacencyList.cpp
 3:  *
 4:  *  Created on: May 28, 2011
 5:  *      Author: darioandrade
 6:  */
 7:
 8: #include "AdjacencyList.h"
 9: #include <cstdlib>
10:
11: #define MAX_LINE_SIZE ( 64 * 1024 )
12:
13: AdjacencyList::AdjacencyList ( )
14: :
15:   m_nVertex( 0 ),
16:   m_nEdges( 0 )
17: {
18: }
19:
20: AdjacencyList::AdjacencyList ( int nVertex )
21: {
22:     Allocate( nVertex );
23:
24:     m_nVertex = nVertex;
25: }
26:
27: AdjacencyList::~AdjacencyList ( )
28: {
29:     for(int i = 0; i < m_nVertex; i++) {
30:         delete m_arrAdjLists[i];
31:     }
32:
33:     delete m_arrAdjLists;
34: }
35:
36: void AdjacencyList::Allocate( int nVertex )
37: {
38:     m_arrAdjLists = new List * [ nVertex ];
39:
40:     for(int i = 0; i < nVertex; i++) {
41:         m_arrAdjLists[i] = new List();
42:     }
43: }
44:
45: void AdjacencyList::addEdge ( int iVertex, int jVertex, bool bUpdateNeighbor, bool bIncEdge )
46: {
47:     m_arrAdjLists[ iVertex ]->insertAtEnd( jVertex );
48:
49:     if ( bIncEdge )
50:     {
51:         m_nEdges ++;
52:     }
53:
54:     if ( bUpdateNeighbor )
55:     {
56:         m_arrAdjLists[ jVertex ]->insertAtEnd( iVertex );
57:     }
58: }
59:
60: // print edges, one vertex each line, to stdout
61: void AdjacencyList::write ( FILE * f )
62: {
63:     if ( f == NULL )
64:     {
65:         f = stdout;
66:     }
67:
68:     fprintf( f, "%d\n", m_nVertex );
69:
70:     // run all vertex
71:     for ( int i = 0; i < m_nVertex; i++ )
72:     {
73:
74:
75:         // iterate through edges
76:         for (ListNode * node = m_arrAdjLists[i]->getFirst(); node != NULL ; node = node->next())
77:         {
78:             fprintf( f, "%d ", node->getVertex() );
79:         }
80:
81:         fputs( "\n", f );
82:     }
83: }
84:
85: void AdjacencyList::read( FILE * f, int debug )
86: {
87:     if ( f == NULL )
88:     {
89:         f = stdin;
90:     }
91:
92:     static char sLine[ MAX_LINE_SIZE ];
93:
94:     fgets( sLine, MAX_LINE_SIZE, f );
```

```
 95:
 96:     m_nVertex = atoi( sLine );
 97:
 98:     Allocate( m_nVertex );
 99:
100:     // read all lines
101:     for ( int nCurrentVertex = 0;
102:             fgets( sLine, MAX_LINE_SIZE, f ) != NULL;
103:             nCurrentVertex++ )
104:     {
105:         if ( debug >= 2 )
106:         {
107:             fprintf( stderr, "\n  lendo linha do vertice %d:\n", nCurrentVertex );
108:         }
109:
110:         int offset = 0;
111:
112:         // read all neighbors from the line
113:         for ( int i = 0; i < MAX_LINE_SIZE; i++ )
114:         {
115:             int nNeighborVertex;
116:
117:             // only read if offset is within array boundaries
118:             if ( offset < MAX_LINE_SIZE )
119:             {
120:                 // read neighbor
121:                 int ret = sscanf( sLine + offset, " %d", &nNeighborVertex );
122:
123:                 // did we read a neighbor? or end of line/file?
124:                 if ( ret != EOF && ret > 0 )
125:                 {
126:                     if ( debug >= 2 )
127:                     {
128:                         fprintf( stderr, " %d ", nNeighborVertex );
129:                     }
130:
131:                     // file has vertex in ascending order, one each line
132:                     // if neighbor that is being added is greater, count the edge, otherwise
133:                     // it's already counted
134:                     addEdge( nCurrentVertex, nNeighborVertex, false, nNeighborVertex > nCurrentVertex );
135:
136:                     // find next item in line
137:                     while ( offset < MAX_LINE_SIZE
138:                             && sLine[ offset ] )
139:                     {
140:                         offset ++;
141:
142:                         if ( sLine[ offset ] == ' ' )
143:                         {
144:                             break;
145:                         }
146:                     }
147:                 }
148:                 else
149:                 {
150:                     // stop reading the line
151:                     break;
152:                 }
153:             }
154:         }
155:     }
156:
157:     // update other structures
158:     updateData();
159:
160:     if ( debug >= 2 )
161:     {
162:         fprintf( stderr, "\n fim do stream de leitura\n" );
163:     }
164: }
165:
166: bool AdjacencyList::HasEdge( ) const
167: {
168:     return m_nEdges > 0;
169: }
170:
```

```
 1: /*
 2:  * DegreeVectorAdjacencyList.h
 3:  *
 4:  *  Created on: Jun 11, 2011
 5:  *      Author: darioandrade
 6:  */
 7:
 8: #ifndef DEGREEVECTORADJACENCYLIST_H_
 9: #define DEGREEVECTORADJACENCYLIST_H_
10:
11: #include "AdjacencyList.h"
12:
13: class DegreeVectorAdjacencyList : public AdjacencyList
14: {
15: public:
16:     DegreeVectorAdjacencyList ( );
17:     virtual ~DegreeVectorAdjacencyList ( );
18:
19:     void addEdge( int iVertex, int jVertex, bool bUpdateNeighbor = true, bool bIncEdge = true );
20:     int RemoveHighestDegreeVertex( int debug );
21:
22: protected:
23:     void Allocate( int nVertex );
24:
25:     void DecrementDegree( int iVertex );
26:     void SetDegree( int iVertex, int degree );
27:     int GetDegree( int iVertex ) const;
28:
29:     int GetHighestDegreeVertex( ) const;
30:
31: private:
32:     int         * m_vectorDegrees;
33: };
34:
35: #endif /* DEGREEVECTORADJACENCYLIST_H_ */
```

```
 1: /*
 2:  * DegreeVectorAdjacencyList.cpp
 3:  *
 4:  *  Created on: Jun 11, 2011
 5:  *      Author: darioandrade
 6:  */
 7:
 8: #include "DegreeVectorAdjacencyList.h"
 9: #include "Heap.h"
10:
11: DegreeVectorAdjacencyList::DegreeVectorAdjacencyList (  )
12: {
13: }
14:
15: DegreeVectorAdjacencyList::~DegreeVectorAdjacencyList ( )
16: {
17:     delete [ ] m_vectorDegrees;
18: }
19:
20: void DegreeVectorAdjacencyList::Allocate( int nVertex )
21: {
22:     AdjacencyList::Allocate( nVertex );
23:
24:     m_vectorDegrees = new int [ nVertex ];
25:
26:     for ( int i = 0; i < nVertex; i++ )
27:     {
28:         m_vectorDegrees[ i ] = 0;
29:     }
30: }
31:
32: void DegreeVectorAdjacencyList::addEdge ( int iVertex, int jVertex, bool bUpdateNeighbor, bool bIncEdge )
33: {
34:     AdjacencyList::addEdge( iVertex, jVertex, bUpdateNeighbor, bIncEdge );
35:
36:     m_vectorDegrees[ iVertex ] ++;
37:
38:     if ( bUpdateNeighbor )
39:     {
40:         m_vectorDegrees[ jVertex ] ++;
41:     }
42: }
43:
44: void DegreeVectorAdjacencyList::DecrementDegree( int iVertex )
45: {
46:     m_vectorDegrees[ iVertex ] --;
47: }
48:
49: void DegreeVectorAdjacencyList::SetDegree( int iVertex, int degree )
50: {
51:     m_vectorDegrees[ iVertex ] = degree;
52: }
53:
54: int DegreeVectorAdjacencyList::GetDegree( int iVertex ) const
55: {
56:     return m_vectorDegrees[ iVertex ];
57: }
58:
59: int DegreeVectorAdjacencyList::GetHighestDegreeVertex( ) const
60: {
61:     int highestDegree = 0;
62:     int iHighestDegreeVertex = -1;
63:
64:     // iterate over the degree vector and find highest degree vector
65:     for ( int i = 0; i < GetSize( ); i++ )
66:     {
67:         // swap if higher
68:         if ( m_vectorDegrees[ i ] > highestDegree )
69:         {
70:             highestDegree = m_vectorDegrees[ i ];
71:             iHighestDegreeVertex = i;
72:         }
73:     }
74:
75:     // if no degree > 0, vertex returned will be -1 (no vertex)
76:     return iHighestDegreeVertex;
77: }
78:
79:
80: int DegreeVectorAdjacencyList::RemoveHighestDegreeVertex( int debug )
81: {
82:     int iHighestDegreeVertex = GetHighestDegreeVertex( );
83:
84:     // find neighbors of this vertex
85:     List * neighbors = m_arrAdjLists[ iHighestDegreeVertex ];
86:
87:     if ( debug >= 2 )
88:     {
89:         fprintf( stderr, "  vertice %d tem %d vizinhos e grau: %d\n",
90:                 iHighestDegreeVertex,
91:                 neighbors->size( ),
92:                 GetDegree( iHighestDegreeVertex ) );
93:     }
94:
```

```
 95:     for ( ListNode * node = neighbors->getFirst();
 96:         node != NULL;
 97:         node = node->next())
 98:     {
 99:         int iNeighbor = node->getVertex();
100:
101:         // update this vertex's neighbor's list that this vertex is being removed
102:         // DATS: Nao eh mais necessario tirar o vertice da lista de adjacencias
103:         // do vizinho, uma vez que o vetor de graus (que de fato Ã© consultado)
104:         // jÃ¡ Ã© decrementado
105:         //m_arrAdjLists[ iNeighbor ]->erase( iHighestDegreeVertex );
106:
107:         // if this neighbor still has edges (it means it has not been removed
108:         // otherwise it must have edges, since it is a neighbor from the highestdegreevertex
109:         if ( m_vectorDegrees[ iNeighbor ] > 0 )
110:         {
111:             // remove edge from this vertex
112:             m_nEdges --;
113:
114:             // decrement degree from neighbor
115:             DecrementDegree( iNeighbor );
116:         }
117:     }
118:
119:     // remove edges to neighbors, and let the vertex linger and ...
120:     //neighbors.clear( );
121:     // reset degree
122:     SetDegree( iHighestDegreeVertex, 0 );
123:
124:     return iHighestDegreeVertex;
125: }
```

```
 1: #ifndef DEGREEHEAPADJACENCYLIST_H_
 2: #define DEGREEHEAPADJACENCYLIST_H_
 3:
 4: #include "AdjacencyList.h"
 5: #include "Heap.h"
 6:
 7: class DegreeHeapAdjacencyList : public AdjacencyList
 8: {
 9: public:
10:     DegreeHeapAdjacencyList ( );
11:
12:     virtual ~DegreeHeapAdjacencyList ( );
13:
14:     void Allocate( int nVertex );
15:     int RemoveHighestDegreeVertex( int debug );
16:
17: protected:
18:
19:     void updateData( );
20:
21: private:
22:     Heap* m_heap;
23: };
24:
25: #endif /* DEGREEHEAPADJACENCYLIST_H_ */
```

```
 1: #include "DegreeHeapAdjacencyList.h"
 2: #include "VertexVectorAdjacencyList.h"
 3:
 4: DegreeHeapAdjacencyList::DegreeHeapAdjacencyList( )
 5: {
 6: }
 7: DegreeHeapAdjacencyList::~DegreeHeapAdjacencyList( )
 8: {
 9:     delete m_heap;
10: }
11:
12: void DegreeHeapAdjacencyList::Allocate( int nVertex )
13: {
14:     AdjacencyList::Allocate( nVertex );
15:
16:     m_heap = new Heap( nVertex );
17: }
18:
19: int DegreeHeapAdjacencyList::RemoveHighestDegreeVertex( int debug )
20: {
21:     // remove the highest degree vertex from heap
22:     std::pair<int, int> vertex;
23:     m_heap->removeFromHeap( vertex );
24:     int iHighestDegreeVertex = vertex.first;
25:
26:     List * neighbors = m_arrAdjLists[ iHighestDegreeVertex ];
27:
28:     if ( debug >= 2 )
29:     {
30:         fprintf( stderr, "  vertice %d tem %d vizinhos\n",
31:                  iHighestDegreeVertex,
32:                  neighbors->size() );
33:     }
34:
35:     // decrease its neighbor's degree
36:     for ( ListNode * node = neighbors->getFirst();
37:         node != NULL;
38:         node = node->next())
39:     {
40:         int iNeighbor = node->getVertex();
41:
42:         // update this vertex's neighbor's list that this vertex is being removed
43:         //m_arrAdjLists[ iNeighbor ]->erase( iHighestDegreeVertex );
44:
45:         if ( m_heap->HasVertex( iNeighbor ) )
46:         {
47:             // remove edge from this vertex
48:             m_nEdges --;
49:
50:             // decrement degree from neighbor
51:             m_heap->DecrementDegree( iNeighbor );
52:         }
53:     }
54:
55:     // remove edges to neighbors, and let the vertex linger
56:     //neighbors.clear( );
57:
58:     return iHighestDegreeVertex;
59:
60: }
61:
62: void DegreeHeapAdjacencyList::updateData( )
63: {
64:
65:     for( int i = 0; i < m_nVertex; i++ )
66:     {
67:         m_heap->insertOnHeap( i, (int) m_arrAdjLists[i]->size() );
68:     }
69: }
```

```
 1: #include <stdio.h>
 2:
 3: #include <utility>
 4:
 5: #include "AdjacencyList.h"
 6:
 7: class Heap
 8: {
 9: public:
10:     Heap( int nVertex );
11:     ~Heap();
12:
13:     void insertOnHeap(int iVertex, int degree);
14:     bool removeFromHeap(std::pair<int, int> & highestPair);
15:
16:     void DecrementDegree( int iVertex );
17:
18:     bool HasVertex(int iVertex) const;
19:
20:     void print();
21: private:
22:     void allocate();
23:     bool swapWithFather(int iSlotIndex);
24:     void bubleUpElement(int iSlotIndex);
25:     void bubleDownElement(int iSlotIndex);
26:     int swapWithChildren(int iSlotIndex);
27:
28:
29:     int m_nVertex;
30:
31:     // Vector of [vertex, degree] pairs
32:     std::pair<int, int> * m_heapVector;
33:
34:     int * m_indexerHeap;
35:     int m_nextAvailableSlot;
36:
37: };
```

```
 1: #include <math.h>
 2:
 3: #include "Heap.h"
 4:
 5: Heap::Heap(int nVertex)
 6: :
 7:     m_nVertex(nVertex),
 8:     m_heapVector(NULL),
 9:     m_indexerHeap(NULL),
10:     m_nextAvailableSlot(0)
11: {
12:     allocate();
13: }
14:
15: Heap::~Heap()
16: {
17:     delete m_heapVector;
18:     delete m_indexerHeap;
19: }
20:
21: void Heap::allocate()
22: {
23:     m_heapVector = new std::pair<int, int>[ m_nVertex ];
24:     m_indexerHeap = new int[ m_nVertex ];
25:     for (int i = 0; i < m_nVertex; i++)
26:     {
27:         m_indexerHeap[i] = -1;
28:     }
29: }
30:
31:  bool Heap::removeFromHeap(std::pair<int, int> & root)
32: {
33:     if (m_nextAvailableSlot > 0) {
34:         root = m_heapVector[0];
35:
36:         std::pair<int, int> lastInserted = m_heapVector[m_nextAvailableSlot - 1];
37:
38:         m_heapVector[0] = lastInserted;
39:
40:         // element ja esta alocado no heapvector, para zerar eh so setar zero
41:         m_heapVector[m_nextAvailableSlot - 1].first = 0;// = *(new std::pair<int, int>(0, 0));
42:         m_heapVector[m_nextAvailableSlot - 1].second = 0;
43:         m_nextAvailableSlot--;
44:
45:         // remove vertex from heap's index
46:         m_indexerHeap[root.first] = -1;
47:
48:         bubleDownElement(0);
49:
50:     } else {
51:         fprintf( stderr, "Error removing from empty heap.\n" );
52:         return false;
53:     }
54:
55:     return true;
56: }
57:
58: bool Heap::HasVertex(int iVertex) const
59: {
60:     return m_indexerHeap[iVertex] != -1;
61: }
62:
63: void Heap::insertOnHeap(int iVertex, int degree)
64: {
65:     m_heapVector[m_nextAvailableSlot].first = iVertex;
66:     m_heapVector[m_nextAvailableSlot].second = degree;
67:
68:     // Fill indexer vector
69:     m_indexerHeap[iVertex] =  m_nextAvailableSlot;
70:
71:     // we may need to use our current status in bubbleUpElement,
72:     // so we need to increment before calling the method below
73:     // do not move position
74:     m_nextAvailableSlot++;
75:
76:     bubleUpElement(m_nextAvailableSlot - 1);
77: }
78:
79:
80: void Heap::DecrementDegree( int iVertex )
81: {
82:     m_heapVector[ m_indexerHeap[iVertex] ].second--;
83:     bubleDownElement(m_indexerHeap[iVertex]);
84: }
85:
86: void Heap::bubleUpElement(int iSlotIndex)
87: {
88:     // First element don't have father
89:     if (iSlotIndex <= 0) {
90:         return;
91:     }
92:
93:     // swap element with father
94:     if (swapWithFather(iSlotIndex)) {
```

```
 95:         // if we had an inversion, call recursively for the father
 96:         bubleUpElement((iSlotIndex - 1) / 2);
 97:     }
 98: }
 99:
100: bool Heap::swapWithFather(int iSlotIndex)
101: {
102:     std::pair<int, int> element = m_heapVector[iSlotIndex];
103:
104:     // Compare with parent
105:     if (element.second > m_heapVector[(iSlotIndex - 1) / 2].second) {
106:
107:         // swap with father
108:         std::pair<int, int> father = m_heapVector[(iSlotIndex - 1) / 2];
109:
110:         m_heapVector[(iSlotIndex - 1) / 2] = element;
111:         m_heapVector[iSlotIndex] = father;
112:
113:         m_indexerHeap[ element.first ] = (iSlotIndex - 1) / 2;
114:         m_indexerHeap[ father.first ] = iSlotIndex;
115:
116:         return true;
117:     }
118:
119:     return false;
120: }
121:
122: void Heap::bubleDownElement(int iSlotIndex)
123: {
124:     // Last Nodes already had passed
125:     if (iSlotIndex >= m_nextAvailableSlot) {
126:         return;
127:     }
128:
129:     int swapIndex = swapWithChildren(iSlotIndex);
130:
131:     if (swapIndex == 1) {
132:         bubleDownElement(2 * iSlotIndex + 1);
133:     } else if (swapIndex == 2) {
134:         bubleDownElement(2 * iSlotIndex + 2);
135:     }
136:
137: }
138:
139: int Heap::swapWithChildren(int iSlotIndex)
140: {
141:     int higher = -1;
142:     int result = -1;
143:
144:     int leftChildIndex = 2 * iSlotIndex + 1;
145:     int rightChildIndex = 2 * iSlotIndex + 2;
146:
147:     if((leftChildIndex < m_nextAvailableSlot)  && (rightChildIndex < m_nextAvailableSlot)) {
148:         if (m_heapVector[iSlotIndex].second < m_heapVector[leftChildIndex].second &&
149:             m_heapVector[iSlotIndex].second < m_heapVector[rightChildIndex].second) {
150:             if (m_heapVector[leftChildIndex].second > m_heapVector[rightChildIndex].second) {
151:                 higher = 2 * iSlotIndex + 1;
152:                 result = 1;
153:
154:             } else if (m_heapVector[rightChildIndex].second > m_heapVector[leftChildIndex].second) {
155:                 higher = 2 * iSlotIndex + 2;
156:                 result = 2;
157:             }
158:         }
159:
160:         if (higher != -1) {
161:             // swap with child
162:             std::pair<int, int> child = m_heapVector[higher];
163:             std::pair<int, int> element = m_heapVector[iSlotIndex];
164:
165:             m_heapVector[higher] = m_heapVector[iSlotIndex];
166:             m_heapVector[iSlotIndex] = child;
167:
168:             m_indexerHeap[ element.first ] = higher;
169:             m_indexerHeap[ child.first ] = iSlotIndex;
170:
171:             // 1 indicates that we swapped with left child
172:             return result;
173:         }
174:     }
175:
176:     if (leftChildIndex < m_nextAvailableSlot) {
177:         // Compare with left child
178:         if (m_heapVector[iSlotIndex].second < m_heapVector[leftChildIndex].second) {
179:             std::pair<int, int> element = m_heapVector[iSlotIndex];
180:             // swap with child
181:             std::pair<int, int> child = m_heapVector[leftChildIndex];
182:
183:             m_heapVector[leftChildIndex] = m_heapVector[iSlotIndex];
184:             m_heapVector[iSlotIndex] = child;
185:
186:             m_indexerHeap[ element.first ] = leftChildIndex;
187:             m_indexerHeap[ child.first ] = iSlotIndex;
188:
```

```
189:             // 1 indicates that we swapped with left child
190:             return 1;
191:         }
192:     }
193:
194:     if (rightChildIndex < m_nextAvailableSlot) {
195:         if(m_heapVector[iSlotIndex].second < m_heapVector[rightChildIndex].second) {
196:             std::pair<int, int> element = m_heapVector[iSlotIndex];
197:             // swap with child
198:             std::pair<int, int> child = m_heapVector[rightChildIndex];
199:
200:             m_heapVector[rightChildIndex] = m_heapVector[iSlotIndex];
201:             m_heapVector[iSlotIndex] = child;
202:
203:             m_indexerHeap[ element.first ] = rightChildIndex;
204:             m_indexerHeap[ child.first ] = iSlotIndex;
205:
206:             // 1 indicates that we swapped with left child
207:             return 2;
208:         }
209:     }
210:
211:     return 0;
212: }
213:
214: void Heap::print()
215: {
216:     fprintf(stderr, "Vertex: \n");
217:     for(int i = 0; i < m_nVertex; i++) {
218:         fprintf(stderr, " %d ", m_heapVector[i].first);
219:     }
220:
221:     fprintf(stderr, "\nDegrees: \n");
222:     for(int i = 0; i < m_nVertex; i++) {
223:         fprintf(stderr, " %d ", m_heapVector[i].second);
224:     }
225:     fprintf(stderr, "\n");
226: }
```

```
 1: #ifndef VERTEXVECTORADJACENCYLIST_H_
 2: #define VERTEXVECTORADJACENCYLIST_H_
 3:
 4: #include "AdjacencyList.h"
 5:
 6: class VertexVectorAdjacencyList : public AdjacencyList
 7: {
 8: public:
 9:     VertexVectorAdjacencyList();
10:     virtual ~VertexVectorAdjacencyList();
11:
12:     int RemoveHighestDegreeVertex( int debug );
13:
14: protected:
15:     void Allocate( int nVertex );
16:     void DecrementDegree( int iVertex );
17:     void RemoveFromVertexVector( int iVertex, int iDegree );
18:     ListNode* GetHighestDegreeVertex( );
19:     void updateData( );
20:
21: private:
22:
23:     int m_lastHighestDegree;
24:     List ** m_vectorVertex;
25:     ListNode ** m_elementList;
26: };
27:
28: #endif /* DEGREEVECTORADJACENCYLIST_H_ */
```

```
 1: #include <malloc.h>
 2: #include <stdlib.h>
 3:
 4: #include "VertexVectorAdjacencyList.h"
 5:
 6: VertexVectorAdjacencyList::VertexVectorAdjacencyList ( )
 7: {
 8:     m_lastHighestDegree = 0;
 9:     m_nVertex = 0;
10: }
11:
12: VertexVectorAdjacencyList::~VertexVectorAdjacencyList ( )
13: {
14:     delete m_elementList;
15:
16:     for(int i = 0; i <  m_nVertex - 1; i++) {
17:         delete m_vectorVertex[i];
18:     }
19:
20:     delete m_vectorVertex;
21: }
22:
23: void VertexVectorAdjacencyList::Allocate( int nVertex )
24: {
25:     AdjacencyList::Allocate( nVertex );
26:
27:     m_nVertex = nVertex;
28:
29:     m_lastHighestDegree = nVertex - 2;
30:     m_vectorVertex = new List * [ nVertex - 1 ];
31:
32:     for(int i = 0; i <  nVertex - 1; i++) {
33:         m_vectorVertex[i] = new List();
34:     }
35:
36:     m_elementList = new ListNode *[nVertex];
37:
38:     for(int i = 0; i < nVertex; i++) {
39:         m_elementList[i] = NULL;
40:     }
41:
42: }
43:
44: void VertexVectorAdjacencyList::DecrementDegree( int iVertex )
45: {
46:     // remove da lista atual e ..
47:     ListNode* element  = m_elementList[ iVertex ];
48:     int degree = element->getDegree();
49:
50:     m_vectorVertex[ degree ]->remove( element );
51:
52:     // insere na lista respectiva ao grau-1
53:     ListNode* node = m_vectorVertex[ degree - 1 ]->insertAtEnd( iVertex );
54:
55:     node->setDegree( degree - 1 );
56:
57:     m_elementList[ iVertex ] = node;
58: }
59:
60: void VertexVectorAdjacencyList::RemoveFromVertexVector( int iVertex, int iDegree )
61: {
62:     ListNode* element  = m_elementList[ iVertex ];
63:
64:     m_vectorVertex[ iDegree ]->remove( element );
65:     m_elementList[ iVertex ] = NULL;
66: }
67:
68: int VertexVectorAdjacencyList::RemoveHighestDegreeVertex( int debug )
69: {
70:     ListNode * highestDegreeVertex = GetHighestDegreeVertex();
71:     int iHighestDegreeVertex = highestDegreeVertex->getVertex();
72:     List * neighbors = m_arrAdjLists[ iHighestDegreeVertex ];
73:
74:     if ( debug >= 2 )
75:     {
76:         fprintf( stderr, "  vertice %d tem %d vizinhos\n",
77:                  iHighestDegreeVertex,
78:                  neighbors->size( ) );
79:     }
80:
81:     for ( ListNode * node = neighbors->getFirst();
82:             node != NULL;
83:             node = node->next())
84:     {
85:         int iNeighbor = node->getVertex();
86:         //int iCurrentDegree = (int) m_arrAdjLists[ iNeighbor ]->size();
87:
88:         // update this vertex's neighbor's list that this vertex is being removed
89:         //m_arrAdjLists[ iNeighbor ]->erase( iHighestDegreeVertex );
90:
91:         // if the neighbor is still in vertex vector, it means it has not been removed
92:         // note: it may be on my neighbor's list, but already processed and removed, we need
93:         // to make sure we will be decrementing a degree from a neighbor that's already in the graph
94:         if ( m_elementList[ iNeighbor ] != NULL )
```

```
 95:         {
 96:             // remove edge from this vertex
 97:             m_nEdges --;
 98:
 99:             // decrement degree from neighbor
100:             DecrementDegree( iNeighbor );//, iCurrentDegree );
101:         }
102:     }
103:
104:
105:     // reset degree
106:     RemoveFromVertexVector( iHighestDegreeVertex, highestDegreeVertex->getDegree() );//neighbors->size() );
107:
108:     // remove edges to neighbors, and let the vertex linger and ...
109:     //neighbors.clear( );
110:
111:     return iHighestDegreeVertex;
112: }
113:
114: ListNode* VertexVectorAdjacencyList::GetHighestDegreeVertex( )
115: {
116:     for(int i = m_lastHighestDegree; i >= 0; i--)
117:     {
118:         if(m_vectorVertex[i]->size() != 0)
119:         {
120:             m_lastHighestDegree = i;
121:             return m_vectorVertex[i]->getFirst();//)->getVertex();
122:         }
123:     }
124:
125:     return NULL;
126: }
127:
128: void VertexVectorAdjacencyList::updateData()
129: {
130:     for( int i = 0; i < m_nVertex; i++ )
131:     {
132:         int degree = m_arrAdjLists[i]->size();
133:
134:         // the vertex degree is its position in the vector
135:         ListNode* node = m_vectorVertex[ degree ]->insertAtEnd( i );
136:         node->setDegree( degree );
137:
138:         m_elementList[i] = node;
139:     }
140: }
```

```
 1: #ifndef LIST_H
 2: #define LIST_H
 3:
 4: #include <utility>
 5: #include "ListNode.h"
 6: /*****************************************************************************/
 7: /*                                                                           */
 8: /*            Tipo de dados lista genÃ©rica                                   */
 9: /*                                                                           */
10: /*****************************************************************************/
11:
12: class List
13: {
14: public:
15:     List();
16:     ~List();
17:
18:     ListNode* insertAtEnd(int content);
19:
20:     int insertAtFront(int content);
21:
22:     int removeFirst();
23:
24:     int size();
25:
26:     ListNode * getFirst() { return m_first; }
27:
28:     ListNode * getLast() { return m_last; }
29:
30:     void erase(int content);
31:
32:     void remove( ListNode* node );
33:
34: private:
35:     int m_numElems;
36:     ListNode * m_first;
37:     ListNode * m_last;
38: };
39:
40: #endif
41:
42:
```

```
 1: #include "List.h"
 2: #include "ListNode.h"
 3: #include <stdio.h>
 4:
 5: List::List()
 6: {
 7:     m_first = NULL;
 8:     m_last = NULL;
 9:     m_numElems = 0;
10: }
11:
12: List::~List()
13: {
14:     // DOES NOTHING
15: }
16:
17: ListNode* List::insertAtEnd(int content)
18: {
19:     ListNode * node = new ListNode(content);
20:     ListNode * previous = NULL;
21:
22:     /* Primeiro elemento */
23:     if ((m_first == NULL) && (m_last == NULL)) {
24:         m_first = node;
25:         m_last = node;
26:     } else {
27:         node->setPrevious( m_last );
28:         previous = m_last;
29:         m_last->setNext(node);
30:         m_last = node;
31:     }
32:
33:     m_numElems++;
34:
35:     return node;
36: }
37:
38: int List::insertAtFront(int content)
39: {
40:     ListNode * node = new ListNode(content);
41:
42:     /* Primeiro elemento */
43:     if ((m_first == NULL) && (m_last == NULL)) {
44:         m_first = node;
45:         m_last = node;
46:     } else {
47:         node->setNext(m_first);
48:         m_first = node;
49:     }
50:
51:     m_numElems++;
52:
53:     return 0;
54: }
55:
56: int List::removeFirst()
57: {
58:     if (m_numElems == 0) {
59:         return -1;
60:     }
61:
62:     ListNode * node = m_first;
63:
64:     int content = node->getVertex();
65:
66:     m_first = m_first->next();
67:     m_first->setPrevious( NULL );
68:     delete node;
69:
70:     m_numElems--;
71:
72:     return content;
73: }
74:
75: int List::size()
76: {
77:     return m_numElems;
78: }
79:
80: void List::erase(int content)
81: {
82:     ListNode * previous = NULL;
83:
84:     for(ListNode * node = m_first; node != NULL; node = node->next()) {
85:
86:         if(content == node->getVertex()) {
87:
88:             // Primeiro da lista
89:             if (node == m_first) {
90:                 if (m_numElems == 1) {
91:                     m_first = NULL;
92:                     m_last = NULL;
93:                 } else {
94:                     m_first = node->next();
```

```
 95:                     m_first->setPrevious( NULL );
 96:                 }
 97:             } else {
 98:                 previous->setNext(node->next());
 99:
100:                 if (node == m_last) {
101:                     m_last = previous;
102:                 }
103:                 else
104:                 {
105:                     node->next()->setPrevious( previous );
106:                 }
107:             }
108:
109:
110:                 m_numElems--;
111:             delete node;
112:             return;
113:         }
114:         previous = node;
115:     }
116: }
117:
118:
119: void List::remove( ListNode* node )
120: {
121:     if (node == NULL)
122:         return;
123:
124:     ListNode* previous = node->previous();
125:
126:     // primeiro da lista
127:     if (node == m_first)
128:     {
129:         if (m_numElems == 1)
130:         {
131:             m_first = NULL;
132:             m_last = NULL;
133:         }
134:         else
135:         {
136:             m_first = node->next();
137:             m_first->setPrevious( NULL );
138:         }
139:     }
140:     else
141:     {
142:         previous->setNext(node->next());
143:
144:         if (node == m_last)
145:         {
146:             m_last = previous;
147:         }
148:         else
149:         {
150:             node->next()->setPrevious( previous );
151:         }
152:     }
153:
154:     m_numElems--;
155:
156:     delete node;
157: }
```

```
 1:
 2: #ifndef LIST_NODE_H
 3: #define LIST_NODE_H
 4:
 5: #include <stdio.h>
 6:
 7: class ListNode
 8: {
 9: public:
10:     ListNode(int content);
11:     ~ListNode();
12:
13:     int getVertex() { return m_vertex; }
14:     int getDegree() { return m_degree; }
15:     void setDegree(int degree) { m_degree = degree; }
16:     ListNode * next() { return m_next; }
17:     void setNext(ListNode * node) { m_next = node; }
18:         ListNode * previous() { return m_previous; }
19:     void setPrevious(ListNode * node) { m_previous = node; }
20:
21: private:
22:     int m_vertex;
23:     int m_degree;
24:     ListNode * m_next;
25:     ListNode* m_previous;
26: };
27:
28: #endif
29:
30:
```

```
 1:
 2: #include "ListNode.h"
 3:
 4:
 5: ListNode::ListNode(int content)
 6: :
 7:     m_vertex(content),
 8:     m_degree(-1),
 9:     m_next(NULL),
10:     m_previous(NULL)
11: {
12:
13: }
14:
15: ListNode::~ListNode()
16: {
17:
18: }
19:
20:
```

```
 1: # include <stdio.h>
 2: #include "Heap.h"
 3:
 4: int main(int argc, char * argv[])
 5: {
 6:     Heap heap(10);
 7:
 8:     fprintf(stderr, "inserindo 0, 3\n");
 9:
10:     heap.insertOnHeap(0, 3);
11:
12:     fprintf(stderr, "inserindo 1, 1\n");
13:
14:     heap.insertOnHeap(1, 1);
15:
16:     fprintf(stderr, "inserindo 2, 3\n");
17:
18:     heap.insertOnHeap(2, 3);
19:
20:     fprintf(stderr, "inserindo 3, 2\n");
21:
22:     heap.insertOnHeap(3, 2);
23:
24:     fprintf(stderr, "inserindo 4, 2\n");
25:
26:     heap.insertOnHeap(4, 2);
27:
28:     fprintf(stderr, "inserindo 5, 3\n");
29:
30:     heap.insertOnHeap(5, 3);
31:
32:     fprintf(stderr, "inserindo 6, 2\n");
33:
34:     heap.insertOnHeap(6, 2);
35:
36:     fprintf(stderr, "inserindo 7, 1\n");
37:
38:     heap.insertOnHeap(7, 1);
39:
40:     fprintf(stderr, "inserindo 8, 4\n");
41:
42:     heap.insertOnHeap(8, 4);
43:
44:     fprintf(stderr, "inserindo 9, 1\n");
45:
46:     heap.insertOnHeap(9, 1);
47:
48:     heap.print();
49:     fprintf(stderr, "\n");
50:
51:     heap.DecrementDegree(2);
52:
53:     heap.print();
54:
55:
56:     fprintf(stderr, "Removendo da heap:\n");
57:
58:
59:     for(int i = 0; i < 20; i++) {
60:         std::pair<int, int> element;
61:         heap.removeFromHeap(element);
62:
63:         fprintf(stderr, "%d - vertex: %d - degree: %d\n", i, element.first, element.second);
64:     }
65:
66:     return 0;
67: }
```

```
 1:
 2: #include <stdio.h>
 3:
 4: #include "GeraGrafo.h"
 5: #include "DegreeVectorAdjacencyList.h"
 6:
 7: #define NVERTEX 10
 8: #define FILENAME_GRAFO "testedegrafo_gerado.grafo"
 9: #define FILENAME_GRAFO_LIDO "testedegrafo_lido.grafo"
10:
11: int main( int argc, char ** argv )
12: {
13:     int debug = 2;
14:
15:     fprintf( stderr, "Gerando grafo com %d vertices\n", NVERTEX );
16:
17:     FILE * f = fopen( FILENAME_GRAFO, "w" );
18:
19:     GeraGrafo( f, NVERTEX, 1, debug );
20:
21:     fclose( f );
22:
23:     f = fopen( FILENAME_GRAFO, "r" );
24:
25:     DegreeVectorAdjacencyList grafo;
26:
27:     fprintf( stderr, "lendo arquivo do grafo gerado: %s\n", FILENAME_GRAFO );
28:
29:     grafo.read( f, debug );
30:
31:     fclose( f );
32:
33:     f = fopen( FILENAME_GRAFO_LIDO, "w" );
34:
35:     fprintf( stderr, "escrevendo arquivo do grafo lido: %s\n", FILENAME_GRAFO_LIDO );
36:
37:     grafo.write( f );
38:
39:     fclose ( f );
40:
41:     return 0;
42: }
```