```cpp
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#include "GeraGrafo.h"

void imprimeErro ( char* exename )
{
    printf( "Gerador de grafos. Uso: %s [nvertex tipo(0|1) [debuglevel(0-2)]]\n", exename );
    printf( "Se chamado sem parametros, gera 20 sequencias, 10 de cada tipo.\n" );
    printf( "Tipo 0: p = 0.2\n" );
    printf( "Tipo 1: p = 1 / ( 2 * ( n ^ 0.5 ) )\n" );
}

int main ( int argc, char ** argv )
{
    int debug = 0;
    int nVertex = 0;

    if ( argc < 3 )
    {
        for ( int c = 0; c < NSEQS; c++ )
        {
            for ( int i = 1; i <= NGRAFOS_POR_SEQ; i++ )
            {
                char path[ 256 ];

                sprintf( path, "grafo_p%d_%02d.grafo", c, i );

                FILE * f = fopen( path , "w" );

                int n = 10 * ( ( int ) pow( 2, i ) );

                // gera grafo com debug = 1 (so para dizer o que esta fazendo, nao depurar)
                GeraGrafo( f, n, c, 1 );

                fclose( f );
            }
        }
    }
    else
    {
        nVertex = ( int ) ( atoi( argv[1] ) );

        int type = atoi( argv[2] );
        if ( type > 1 )
        {
            printf( "Tipo invalido.\n\n" );
            imprimeErro( argv[0] );

            exit( 2 );
        }

        if ( nVertex <= 0 || nVertex > 100000 )
        {
            printf( "Numero de vertices invalido (%d).\n\n", nVertex );
            imprimeErro( argv[0] );

            exit( 3 );
        }

        if ( argc > 3 )
        {
            debug = atoi( argv[3] );
        }

        GeraGrafo( stdout, nVertex, type, debug );
    }

    return 1;
}
```

```c
/*
 * GeraGrafo.h
 *
 *  Created on: May 28, 2011
 *      Author: darioandrade
 */

#ifndef GERAGRAFO_H_
#define GERAGRAFO_H_

void GeraGrafo( FILE* f, int nVertex, int type, int debug );


#define NSEQS                   2
#define NGRAFOS_POR_SEQ         10

#endif /* GERAGRAFO_H_ */
```

```cpp
/*
 * GeraGrafo.cpp
 *
 *  Created on: May 28, 2011
 *      Author: darioandrade
 */

#include <stdlib.h>
#include <math.h>
#include <ctime>

#include "AdjacencyList.h"

#define P_TIPO_0 0.2

double randd ( )
{
    return ( ( double ) rand( ) ) / RAND_MAX;
}

void GeraGrafo ( FILE* f, int nVertex, int type, int debug )
{
    // alimentando a semente randomica
    srand( time( NULL ) );

    // calculando p
    double p = ( type == 0 ) ? P_TIPO_0 : ( 1.0 / ( 2.0 * sqrt( nVertex ) ) );

    if ( debug > 0 )
    {
        fprintf( stderr,
                "Gerando %d vertices de grafo nao direcionado tipo %d (p = %.7f)\n",
                nVertex, type, p );
    }

    // criando o grafo
    AdjacencyList graph( nVertex );

    // percorrendo do primeiro vertice ate o penultimo (zero based list)
    for ( int i = 0; i < nVertex - 1; i++ )
    {
        if ( debug > 1 )
        {
            fprintf( stderr, "  vertice %d: ", i );
        }

        for ( int j = i + 1; j < nVertex; j++ )
        {
            double x = randd( );

            if ( debug > 1 )
            {
                fprintf( stderr, " %.4f", x );
            }

            // should we cast an edge to j?
            if ( x < p )
            {
                if ( debug > 1 )
                {
                    fprintf( stderr, "Y" );
                }

                graph.addEdge( i, j );
            }
            else
            {
                if ( debug > 1 )
                {
                    fprintf( stderr, " " );
                }
            }
        }

        if ( debug > 1 )
        {
            fprintf( stderr, "\n" );
        }
    }

    graph.write( f );
}
```

```cpp
/*
 * cobertura.cpp
 *
 *  Created on: Jun 11, 2011
 *      Author: darioandrade
 */

#include <sys/timeb.h>
#include <time.h>
#include <string.h>
#include <stdio.h>
#include <cstdlib>
#include "AlgoritmoCoberturaGulosa.h"
#include "DegreeVectorAdjacencyList.h"
#include "VertexVectorAdjacencyList.h"
#include "DegreeHeapAdjacencyList.h"

#include "GeraGrafo.h"

typedef long unsigned int ltime;

static ltime getMilliCount ( )
{
    timeb tb;
    ftime( &tb );
    time_t nCount = tb.millitm + ( tb.time * 1000 );
    return nCount;
}

/*
static ltime getMicroCount ( )
{
    timeval tv;
    timezone tz;
    tm *tm;
    gettimeofday(&tv, &tz);
    return tv.tv_usec;
}
*/

void FacaTarefa ( AdjacencyList* pGrafo, int debug )
{
    AdjacencyList& grafo = *pGrafo;

    AlgoritmoCoberturaGulosa guloso;

    std::list<int> listaDaCobertura;

    if ( debug >= 1 )
    {
        fprintf( stderr, "Calculando cobertura para o grafo com %d vertices\n", grafo.GetSize( ) );
    }

    guloso.CalculateCobertura( listaDaCobertura, grafo, debug );

    if ( debug >= 1 )
    {
        fprintf( stderr, "Cobertura para o grafo (nvertex: %d) tem %d vertices:\n", grafo.GetSize( ),
                ( int ) listaDaCobertura.size( ) );
    }

    for ( std::list<int>::iterator it = listaDaCobertura.begin( ); it != listaDaCobertura.end( ); it++ )
    {
        fprintf( stdout, "%d ", *it );
    }

    fprintf( stdout, "\n" );
}

AdjacencyList * EscolhaGrafo ( int tarefa )
{
    AdjacencyList * pGrafo = NULL;

    switch (tarefa)
    {
        case 3:
            pGrafo = new DegreeVectorAdjacencyList( );
            break;

        case 4:
            pGrafo = new DegreeHeapAdjacencyList( );
            break;

        case 5:
            pGrafo = new VertexVectorAdjacencyList( );
            break;
    }

    return pGrafo;
}

void Benchmark ( int debug )
{
    struct BenchmarkData {
        char sFilename[ 256 ];
        int graphtype;
        ltime loadtime, processingtime;
        int algorithm;
        int nvertex;
        int nedges;
        int ncobertura;
    };

    BenchmarkData results[ 3 ][ NSEQS ][ NGRAFOS_POR_SEQ ];

    for ( int algo = 3; algo <= 5; algo++ )
    {
        AdjacencyList * pGrafo = EscolhaGrafo( algo );

        for ( int c = 0; c < NSEQS; c++ )
        {
            for ( int i = 1; i <= NGRAFOS_POR_SEQ; i++ )
            {
                char path[256];
                ltime deltams;
                BenchmarkData & data = results[ algo - 3 ][ c ][ i - 1 ];

                sprintf( path, "grafo_p%d_%02d.grafo", c, i );

                if ( debug >= 1 )
                {
                    fprintf( stderr, "Lendo grafo %s\n", path );
                }

                data.algorithm = algo;
                data.graphtype = c;
                strcpy( data.sFilename, path );

                FILE * f = fopen( path, "r" );

                deltams = getMilliCount( );

                pGrafo->read( f );

                deltams = getMilliCount( ) - deltams;

                fclose( f );

                if ( debug >= 1 )
                {
                    fprintf( stderr, "Grafo com %d vertices lido em %lu ms. Calculando cobertura com algoritmo %d\n",
                            pGrafo->GetSize( ), deltams, algo );
                }

                data.loadtime = deltams;
                data.nvertex = pGrafo->GetSize( );
                data.nedges = pGrafo->GetEdges( );

                deltams = getMilliCount( );

                std::list<int> listaDaCobertura;

                AlgoritmoCoberturaGulosa guloso;

                guloso.CalculateCobertura( listaDaCobertura, *pGrafo, debug );

                deltams = getMilliCount( ) - deltams;

                if ( debug >= 1 )
                {
                    fprintf( stderr, "Cobertura possui %d vertices, calculada em %lu ms\n",
                            ( int ) listaDaCobertura.size( ), deltams );
                }

                data.ncobertura = listaDaCobertura.size();
                data.processingtime = deltams;

                fprintf( stdout, "%d,%d,%s,%d,%d,%d,%lu,%lu\n",
                        data.algorithm,
                        data.graphtype,
                        data.sFilename,
                        data.nvertex,
                        data.nedges,
                        data.ncobertura,
                        data.loadtime,
                        data.processingtime );

                fflush( stdout );
            } // end for ngrafos por seq
        } // end for seqs

        delete pGrafo;
    } // end for algos
}

int main ( int argc, char ** argv )
{
    if ( argc < 2 )
```

```cpp
    {
        fprintf( stderr, "Cobertura. Uso: %s (benchmark|(filename ([3|4|5]))) [debug]\n", argv[0] );
        exit( 1 );
    }

    int debug = 1;

    if ( strcmp( "benchmark", argv[1] ) == 0 )
    {
        if ( argc > 2 )
        {
            debug = atoi( argv[2] );
        }

        Benchmark( debug );
    }
    else
    {
        int tarefa = atoi( argv[2] );

        if ( argc > 3 )
        {
            debug = atoi( argv[3] );
        }

        AdjacencyList * pGrafo = EscolhaGrafo( tarefa );

        char * sFilename = argv[1];

        if ( debug >= 1 )
        {
            fprintf( stderr, "Lendo grafo %s:\n", sFilename );
        }

        FILE * f = fopen( sFilename, "r" );

        pGrafo->read( f, debug );

        fclose( f );

        FacaTarefa( pGrafo, debug );

        delete pGrafo;
    }

    return 0;
}
```

```cpp
/*
 * AlgoritmoCoberturaGulosa.h
 *
 *  Created on: Jun 11, 2011
 *      Author: darioandrade
 */

#include <list>

#include "AdjacencyList.h"

#ifndef ALGORITMOCOBERTURAGULOSA_H_
#define ALGORITMOCOBERTURAGULOSA_H_

class AlgoritmoCoberturaGulosa
{
public:
    AlgoritmoCoberturaGulosa ( );
    virtual ~AlgoritmoCoberturaGulosa ( );

    void CalculateCobertura(std::list< int > & listCobertura, AdjacencyList& grafo, int debug );

private:
};

#endif /* ALGORITMOCOBERTURAGULOSA_H_ */
```

```cpp
/*
 * AlgoritmoCoberturaGulosa.cpp
 *
 *  Created on: Jun 11, 2011
 *      Author: darioandrade
 */

#include "AlgoritmoCoberturaGulosa.h"

AlgoritmoCoberturaGulosa::AlgoritmoCoberturaGulosa ( )
{


}

AlgoritmoCoberturaGulosa::~AlgoritmoCoberturaGulosa ( )
{

}

void AlgoritmoCoberturaGulosa::CalculateCobertura(std::list< int > & listCobertura, AdjacencyList & grafo, int debug )
{

    while ( grafo.HasEdge( ) )
    {
        if ( debug >= 2 )
        {
            fprintf( stderr, "removendo vertice de maior grau e atualizando vizinhos\n" );
        }

        int vertex = grafo.RemoveHighestDegreeVertex( debug );

        if ( debug >= 2 )
        {
            fprintf( stderr, "vertice de maior grau: %d\n", vertex );
        }

        listCobertura.push_back( vertex );
    }
}
```

```
/*
 * AdjacencyList.h
 *
 *  Created on: May 28, 2011
 *      Author: darioandrade
 */

#ifndef ADJACENCYLIST_H_
#define ADJACENCYLIST_H_

#include <stdio.h>
#include "List.h"

class AdjacencyList {

public:

    AdjacencyList( int nVertex );
    virtual ~AdjacencyList( );

    virtual void addEdge( int iVertex, int jVertex, bool bUpdateNeighbor = true, bool bIncEdge = true );
    void write( FILE * f = NULL );
    void read( FILE * f = NULL, int debug = 0 );

    virtual bool HasEdge( ) const;
    virtual int RemoveHighestDegreeVertex( int debug ) { return -1; }

    int GetSize() const { return m_nVertex; }
    int GetEdges() const { return m_nEdges; }

//  int GetDegree(int iVertex) { return (int)m_arrAdjLists[iVertex]->size(); }


protected:
    AdjacencyList( );
    virtual void Allocate( int nVertex );
    virtual void updateData(){ }

    int                     m_nVertex;
    List **                 m_arrAdjLists;

    int                     m_nEdges;
};

#endif /* ADJACENCYLIST_H_ */
```

```cpp
/*
 * AdjacencyList.cpp
 *
 *  Created on: May 28, 2011
 *      Author: darioandrade
 */

#include "AdjacencyList.h"
#include <cstdlib>

#define MAX_LINE_SIZE ( 64 * 1024 )

AdjacencyList::AdjacencyList ( )
:
  m_nVertex( 0 ),
  m_nEdges( 0 )
{
}

AdjacencyList::AdjacencyList ( int nVertex )
{
    Allocate( nVertex );

    m_nVertex = nVertex;
}

AdjacencyList::~AdjacencyList ( )
{
    for(int i = 0; i < m_nVertex; i++) {
        delete m_arrAdjLists[i];
    }

    delete m_arrAdjLists;
}

void AdjacencyList::Allocate( int nVertex )
{
    m_arrAdjLists = new List * [ nVertex ];

    for(int i = 0; i < nVertex; i++) {
        m_arrAdjLists[i] = new List();
    }
}

void AdjacencyList::addEdge ( int iVertex, int jVertex, bool bUpdateNeighbor, bool bIncEdge )
{
    m_arrAdjLists[ iVertex ]->insertAtEnd( jVertex );

    if ( bIncEdge )
    {
        m_nEdges ++;
    }

    if ( bUpdateNeighbor )
    {
        m_arrAdjLists[ jVertex ]->insertAtEnd( iVertex );
    }
}

// print edges, one vertex each line, to stdout
void AdjacencyList::write ( FILE * f )
{
    if ( f == NULL )
    {
        f = stdout;
    }

    fprintf( f, "%d\n", m_nVertex );

    // run all vertex
    for ( int i = 0; i < m_nVertex; i++ )
    {


        // iterate through edges
        for (ListNode * node = m_arrAdjLists[i]->getFirst(); node != NULL ; node = node->next())
        {
            fprintf( f, "%d ", node->getVertex() );
        }

        fputs( "\n", f );
    }
}

void AdjacencyList::read( FILE * f, int debug )
{
    if ( f == NULL )
    {
        f = stdin;
    }

    static char sLine[ MAX_LINE_SIZE ];

    fgets( sLine, MAX_LINE_SIZE, f );

    m_nVertex = atoi( sLine );

    Allocate( m_nVertex );

    // read all lines
    for ( int nCurrentVertex = 0;
            fgets( sLine, MAX_LINE_SIZE, f ) != NULL;
            nCurrentVertex++ )
    {
        if ( debug >= 2 )
        {
            fprintf( stderr, "\n  lendo linha do vertice %d:\n", nCurrentVertex );
        }

        int offset = 0;

        // read all neighbors from the line
        for ( int i = 0; i < MAX_LINE_SIZE; i++ )
        {
            int nNeighborVertex;

            // only read if offset is within array boundaries
            if ( offset < MAX_LINE_SIZE )
            {
                // read neighbor
                int ret = sscanf( sLine + offset, " %d", &nNeighborVertex );

                // did we read a neighbor? or end of line/file?
                if ( ret != EOF && ret > 0 )
                {
                    if ( debug >= 2 )
                    {
                        fprintf( stderr, " %d ", nNeighborVertex );
                    }

                    // file has vertex in ascending order, one each line
                    // if neighbor that is being added is greater, count the edge, otherwise
                    // it's already counted
                    addEdge( nCurrentVertex, nNeighborVertex, false, nNeighborVertex > nCurrentVertex );

                    // find next item in line
                    while ( offset < MAX_LINE_SIZE
                            && sLine[ offset ] )
                    {
                        offset ++;

                        if ( sLine[ offset ] == ' ' )
                        {
                            break;
                        }
                    }
                }
                else
                {
                    // stop reading the line
                    break;
                }
            }
        }
    }

    // update other structures
    updateData();

    if ( debug >= 2 )
    {
        fprintf( stderr, "\n fim do stream de leitura\n" );
    }
}

bool AdjacencyList::HasEdge( ) const
{
    return m_nEdges > 0;
}

```

```
/*
 * DegreeVectorAdjacencyList.h
 *
 *  Created on: Jun 11, 2011
 *      Author: darioandrade
 */

#ifndef DEGREEVECTORADJACENCYLIST_H_
#define DEGREEVECTORADJACENCYLIST_H_

#include "AdjacencyList.h"

class DegreeVectorAdjacencyList : public AdjacencyList
{
public:
    DegreeVectorAdjacencyList ( );
    virtual ˜DegreeVectorAdjacencyList ( );

    void addEdge( int iVertex, int jVertex, bool bUpdateNeighbor = true, bool bIncEdge = true );
    int RemoveHighestDegreeVertex( int debug );

protected:
    void Allocate( int nVertex );

    void DecrementDegree( int iVertex );
    void SetDegree( int iVertex, int degree );
    int GetDegree( int iVertex ) const;

    int GetHighestDegreeVertex( ) const;

private:
    int         * m_vectorDegrees;
};

#endif /* DEGREEVECTORADJACENCYLIST_H_ */
```

```cpp
/*
 * DegreeVectorAdjacencyList.cpp
 *
 *  Created on: Jun 11, 2011
 *      Author: darioandrade
 */

#include "DegreeVectorAdjacencyList.h"
#include "Heap.h"

DegreeVectorAdjacencyList::DegreeVectorAdjacencyList (  )
{
}

DegreeVectorAdjacencyList::~DegreeVectorAdjacencyList ( )
{
    delete [ ] m_vectorDegrees;
}

void DegreeVectorAdjacencyList::Allocate( int nVertex )
{
    AdjacencyList::Allocate( nVertex );

    m_vectorDegrees = new int [ nVertex ];

    for ( int i = 0; i < nVertex; i++ )
    {
        m_vectorDegrees[ i ] = 0;
    }
}

void DegreeVectorAdjacencyList::addEdge ( int iVertex, int jVertex, bool bUpdateNeighbor, bool bIncEdge )
{
    AdjacencyList::addEdge( iVertex, jVertex, bUpdateNeighbor, bIncEdge );

    m_vectorDegrees[ iVertex ] ++;

    if ( bUpdateNeighbor )
    {
        m_vectorDegrees[ jVertex ] ++;
    }
}

void DegreeVectorAdjacencyList::DecrementDegree( int iVertex )
{
    m_vectorDegrees[ iVertex ] --;
}

void DegreeVectorAdjacencyList::SetDegree( int iVertex, int degree )
{
    m_vectorDegrees[ iVertex ] = degree;
}

int DegreeVectorAdjacencyList::GetDegree( int iVertex ) const
{
    return m_vectorDegrees[ iVertex ];
}

int DegreeVectorAdjacencyList::GetHighestDegreeVertex( ) const
{
    int highestDegree = 0;
    int iHighestDegreeVertex = -1;

    // iterate over the degree vector and find highest degree vector
    for ( int i = 0; i < GetSize( ); i++ )
    {
        // swap if higher
        if ( m_vectorDegrees[ i ] > highestDegree )
        {
            highestDegree = m_vectorDegrees[ i ];
            iHighestDegreeVertex = i;
        }
    }

    // if no degree > 0, vertex returned will be -1 (no vertex)
    return iHighestDegreeVertex;
}


int DegreeVectorAdjacencyList::RemoveHighestDegreeVertex( int debug )
{
    int iHighestDegreeVertex = GetHighestDegreeVertex( );

    // find neighbors of this vertex
    List * neighbors = m_arrAdjLists[ iHighestDegreeVertex ];

    if ( debug >= 2 )
    {
        fprintf( stderr, "  vertice %d tem %d vizinhos e grau: %d\n",
                 iHighestDegreeVertex,
                 neighbors->size( ),
                 GetDegree( iHighestDegreeVertex ) );
    }

    for ( ListNode * node = neighbors->getFirst();
        node != NULL;
        node = node->next())
    {
        int iNeighbor = node->getVertex();

        // update this vertex's neighbor's list that this vertex is being removed
        // DATS: Nao eh mais necessario tirar o vertice da lista de adjacencias
        // do vizinho, uma vez que o vetor de graus (que de fato Ã© consultado)
        // jÃ¡ Ã© decrementado
        //m_arrAdjLists[ iNeighbor ]->erase( iHighestDegreeVertex );

        // if this neighbor still has edges (it means it has not been removed
        // otherwise it must have edges, since it is a neighbor from the highestdegreevertex
        if ( m_vectorDegrees[ iNeighbor ] > 0 )
        {
            // remove edge from this vertex
            m_nEdges --;

            // decrement degree from neighbor
            DecrementDegree( iNeighbor );
        }
    }

    // remove edges to neighbors, and let the vertex linger and ...
    //neighbors.clear( );
    // reset degree
    SetDegree( iHighestDegreeVertex, 0 );

    return iHighestDegreeVertex;
}
```

```
#ifndef DEGREEHEAPADJACENCYLIST_H_
#define DEGREEHEAPADJACENCYLIST_H_

#include "AdjacencyList.h"
#include "Heap.h"

class DegreeHeapAdjacencyList : public AdjacencyList
{
public:
    DegreeHeapAdjacencyList ( );

    virtual ~DegreeHeapAdjacencyList ( );

    void Allocate( int nVertex );
    int RemoveHighestDegreeVertex( int debug );

protected:

    void updateData( );

private:
    Heap* m_heap;
};

#endif /* DEGREEHEAPADJACENCYLIST_H_ */
```

```
#include "DegreeHeapAdjacencyList.h"
#include "VertexVectorAdjacencyList.h"

DegreeHeapAdjacencyList::DegreeHeapAdjacencyList( )
{
}
DegreeHeapAdjacencyList::~DegreeHeapAdjacencyList( )
{
    delete m_heap;
}

void DegreeHeapAdjacencyList::Allocate( int nVertex )
{
    AdjacencyList::Allocate( nVertex );

    m_heap = new Heap( nVertex );
}

int DegreeHeapAdjacencyList::RemoveHighestDegreeVertex( int debug )
{
    // remove the highest degree vertex from heap
    std::pair<int, int> vertex;
    m_heap->removeFromHeap( vertex );
    int iHighestDegreeVertex = vertex.first;

    List * neighbors = m_arrAdjLists[ iHighestDegreeVertex ];

    if ( debug >= 2 )
    {
        fprintf( stderr, "  vertice %d tem %d vizinhos\n",
                 iHighestDegreeVertex,
                 neighbors->size() );
    }

    // decrease its neighbor's degree
    for ( ListNode * node = neighbors->getFirst();
        node != NULL;
        node = node->next())
    {
        int iNeighbor = node->getVertex();

        // update this vertex's neighbor's list that this vertex is being removed
        //m_arrAdjLists[ iNeighbor ]->erase( iHighestDegreeVertex );

        if ( m_heap->HasVertex( iNeighbor ) )
        {
            // remove edge from this vertex
            m_nEdges --;

            // decrement degree from neighbor
            m_heap->DecrementDegree( iNeighbor );
        }
    }

    // remove edges to neighbors, and let the vertex linger
    //neighbors.clear( );

    return iHighestDegreeVertex;

}

void DegreeHeapAdjacencyList::updateData( )
{

    for( int i = 0; i < m_nVertex; i++ )
    {
        m_heap->insertOnHeap( i, (int) m_arrAdjLists[i]->size() );
    }
}
```

Heap.h

```
#include <stdio.h>

#include <utility>

#include "AdjacencyList.h"

class Heap
{
public:
    Heap( int nVertex );
    ˜Heap();

    void insertOnHeap(int iVertex, int degree);
    bool removeFromHeap(std::pair<int, int> & highestPair);

    void DecrementDegree( int iVertex );

    bool HasVertex(int iVertex) const;

    void print();
private:
    void allocate();
    bool swapWithFather(int iSlotIndex);
    void bubleUpElement(int iSlotIndex);
    void bubleDownElement(int iSlotIndex);
    int swapWithChildren(int iSlotIndex);


    int m_nVertex;

    // Vector of [vertex, degree] pairs
    std::pair<int, int> * m_heapVector;

    int * m_indexerHeap;
    int m_nextAvailableSlot;

};
```

```cpp
#include <math.h>

#include "Heap.h"

Heap::Heap(int nVertex)
:
    m_nVertex(nVertex),
    m_heapVector(NULL),
    m_indexerHeap(NULL),
    m_nextAvailableSlot(0)
{
    allocate();
}

Heap::~Heap()
{
    delete m_heapVector;
    delete m_indexerHeap;
}

void Heap::allocate()
{
    m_heapVector = new std::pair<int, int>[ m_nVertex ];
    m_indexerHeap = new int[ m_nVertex ];
    for (int i = 0; i < m_nVertex; i++)
    {
        m_indexerHeap[i] = -1;
    }
}

 bool Heap::removeFromHeap(std::pair<int, int> & root)
{
    if (m_nextAvailableSlot > 0) {
        root = m_heapVector[0];

        std::pair<int, int> lastInserted = m_heapVector[m_nextAvailableSlot - 1];

        m_heapVector[0] = lastInserted;

        // element ja esta alocado no heapvector, para zerar eh so setar zero
        m_heapVector[m_nextAvailableSlot - 1].first = 0;// = *(new std::pair<int, int>(0, 0));
        m_heapVector[m_nextAvailableSlot - 1].second = 0;
        m_nextAvailableSlot--;

        // remove vertex from heap's index
        m_indexerHeap[root.first] = -1;

        bubleDownElement(0);

    } else {
        fprintf( stderr, "Error removing from empty heap.\n" );
        return false;
    }

    return true;
}

bool Heap::HasVertex(int iVertex) const
{
    return m_indexerHeap[iVertex] != -1;
}

void Heap::insertOnHeap(int iVertex, int degree)
{
    m_heapVector[m_nextAvailableSlot].first = iVertex;
    m_heapVector[m_nextAvailableSlot].second = degree;

    // Fill indexer vector
    m_indexerHeap[iVertex] =  m_nextAvailableSlot;

    // we may need to use our current status in bubbleUpElement,
    // so we need to increment before calling the method below
    // do not move position
    m_nextAvailableSlot++;

    bubleUpElement(m_nextAvailableSlot - 1);
}


void Heap::DecrementDegree( int iVertex )
{
    m_heapVector[ m_indexerHeap[iVertex] ].second--;
    bubleDownElement(m_indexerHeap[iVertex]);
}

void Heap::bubleUpElement(int iSlotIndex)
{
    // First element don't have father
    if (iSlotIndex <= 0) {
        return;
    }

    // swap element with father
    if (swapWithFather(iSlotIndex)) {
        // if we had an inversion, call recursively for the father
        bubleUpElement((iSlotIndex - 1) / 2);
    }
}

bool Heap::swapWithFather(int iSlotIndex)
{
    std::pair<int, int> element = m_heapVector[iSlotIndex];

    // Compare with parent
    if (element.second > m_heapVector[(iSlotIndex - 1) / 2].second) {

        // swap with father
        std::pair<int, int> father = m_heapVector[(iSlotIndex - 1) / 2];

        m_heapVector[(iSlotIndex - 1) / 2] = element;
        m_heapVector[iSlotIndex] = father;

        m_indexerHeap[ element.first ] = (iSlotIndex - 1) / 2;
        m_indexerHeap[ father.first ] = iSlotIndex;

        return true;
    }

    return false;
}

void Heap::bubleDownElement(int iSlotIndex)
{
    // Last Nodes already had passed
    if (iSlotIndex >= m_nextAvailableSlot) {
        return;
    }

    int swapIndex = swapWithChildren(iSlotIndex);

    if (swapIndex == 1) {
        bubleDownElement(2 * iSlotIndex + 1);
    } else if (swapIndex == 2) {
        bubleDownElement(2 * iSlotIndex + 2);
    }

}

int Heap::swapWithChildren(int iSlotIndex)
{
    int higher = -1;
    int result = -1;

    int leftChildIndex = 2 * iSlotIndex + 1;
    int rightChildIndex = 2 * iSlotIndex + 2;

    if((leftChildIndex < m_nextAvailableSlot)  && (rightChildIndex < m_nextAvailableSlot)) {
        if (m_heapVector[iSlotIndex].second < m_heapVector[leftChildIndex].second &&
            m_heapVector[iSlotIndex].second < m_heapVector[rightChildIndex].second) {
            if (m_heapVector[leftChildIndex].second > m_heapVector[rightChildIndex].second) {
                higher = 2 * iSlotIndex + 1;
                result = 1;

            } else if (m_heapVector[rightChildIndex].second > m_heapVector[leftChildIndex].second) {
                higher = 2 * iSlotIndex + 2;
                result = 2;
            }
        }

        if (higher != -1) {
            // swap with child
            std::pair<int, int> child = m_heapVector[higher];
            std::pair<int, int> element = m_heapVector[iSlotIndex];

            m_heapVector[higher] = m_heapVector[iSlotIndex];
            m_heapVector[iSlotIndex] = child;

            m_indexerHeap[ element.first ] = higher;
            m_indexerHeap[ child.first ] = iSlotIndex;

            // 1 indicates that we swapped with left child
            return result;
        }
    }

    if (leftChildIndex < m_nextAvailableSlot) {
        // Compare with left child
        if (m_heapVector[iSlotIndex].second < m_heapVector[leftChildIndex].second) {
            std::pair<int, int> element = m_heapVector[iSlotIndex];
            // swap with child
            std::pair<int, int> child = m_heapVector[leftChildIndex];

            m_heapVector[leftChildIndex] = m_heapVector[iSlotIndex];
            m_heapVector[iSlotIndex] = child;

            m_indexerHeap[ element.first ] = leftChildIndex;
            m_indexerHeap[ child.first ] = iSlotIndex;

            // 1 indicates that we swapped with left child
            return 1;
        }
    }

    if (rightChildIndex < m_nextAvailableSlot) {
        if(m_heapVector[iSlotIndex].second < m_heapVector[rightChildIndex].second) {
            std::pair<int, int> element = m_heapVector[iSlotIndex];
            // swap with child
            std::pair<int, int> child = m_heapVector[rightChildIndex];

            m_heapVector[rightChildIndex] = m_heapVector[iSlotIndex];
            m_heapVector[iSlotIndex] = child;

            m_indexerHeap[ element.first ] = rightChildIndex;
            m_indexerHeap[ child.first ] = iSlotIndex;

            // 1 indicates that we swapped with left child
            return 2;
        }
    }

    return 0;
}

void Heap::print()
{
    fprintf(stderr, "Vertex: \n");
    for(int i = 0; i < m_nVertex; i++) {
        fprintf(stderr, " %d ", m_heapVector[i].first);
    }

    fprintf(stderr, "\nDegrees: \n");
    for(int i = 0; i < m_nVertex; i++) {
        fprintf(stderr, " %d ", m_heapVector[i].second);
    }
    fprintf(stderr, "\n");
}
```

```cpp
#ifndef VERTEXVECTORADJACENCYLIST_H_
#define VERTEXVECTORADJACENCYLIST_H_

#include "AdjacencyList.h"

class VertexVectorAdjacencyList : public AdjacencyList
{
public:
    VertexVectorAdjacencyList();
    virtual ~VertexVectorAdjacencyList();

    int RemoveHighestDegreeVertex( int debug );

protected:
    void Allocate( int nVertex );
    void DecrementDegree( int iVertex );
    void RemoveFromVertexVector( int iVertex, int iDegree );
    ListNode* GetHighestDegreeVertex( );
    void updateData( );

private:

    int m_lastHighestDegree;
    List ** m_vectorVertex;
    ListNode ** m_elementList;
};

#endif /* DEGREEVECTORADJACENCYLIST_H_ */
```

```cpp
#include <malloc.h>
#include <stdlib.h>

#include "VertexVectorAdjacencyList.h"

VertexVectorAdjacencyList::VertexVectorAdjacencyList ( )
{
    m_lastHighestDegree = 0;
    m_nVertex = 0;
}

VertexVectorAdjacencyList::~VertexVectorAdjacencyList ( )
{
    delete m_elementList;

    for(int i = 0; i <  m_nVertex - 1; i++) {
        delete m_vectorVertex[i];
    }

    delete m_vectorVertex;
}

void VertexVectorAdjacencyList::Allocate( int nVertex )
{
    AdjacencyList::Allocate( nVertex );

    m_nVertex = nVertex;

    m_lastHighestDegree = nVertex - 2;
    m_vectorVertex = new List * [ nVertex - 1 ];

    for(int i = 0; i <  nVertex - 1; i++) {
        m_vectorVertex[i] = new List();
    }

    m_elementList = new ListNode *[nVertex];

    for(int i = 0; i < nVertex; i++) {
        m_elementList[i] = NULL;
    }

}

void VertexVectorAdjacencyList::DecrementDegree( int iVertex )
{
    // remove da lista atual e ..
    ListNode* element  = m_elementList[ iVertex ];
    int degree = element->getDegree();

    m_vectorVertex[ degree ]->remove( element );

    // insere na lista respectiva ao grau-1
    ListNode* node = m_vectorVertex[ degree - 1 ]->insertAtEnd( iVertex );

    node->setDegree( degree - 1 );

    m_elementList[ iVertex ] = node;
}

void VertexVectorAdjacencyList::RemoveFromVertexVector( int iVertex, int iDegree )
{
    ListNode* element  = m_elementList[ iVertex ];

    m_vectorVertex[ iDegree ]->remove( element );
    m_elementList[ iVertex ] = NULL;
}

int VertexVectorAdjacencyList::RemoveHighestDegreeVertex( int debug )
{
    ListNode * highestDegreeVertex = GetHighestDegreeVertex();
    int iHighestDegreeVertex = highestDegreeVertex->getVertex();
    List * neighbors = m_arrAdjLists[ iHighestDegreeVertex ];

    if ( debug >= 2 )
    {
        fprintf( stderr, "  vertice %d tem %d vizinhos\n",
                 iHighestDegreeVertex,
                 neighbors->size( ) );
    }

    for ( ListNode * node = neighbors->getFirst();
            node != NULL;
            node = node->next())
    {
        int iNeighbor = node->getVertex();
        //int iCurrentDegree = (int) m_arrAdjLists[ iNeighbor ]->size();

        // update this vertex's neighbor's list that this vertex is being removed
        //m_arrAdjLists[ iNeighbor ]->erase( iHighestDegreeVertex );

        // if the neighbor is still in vertex vector, it means it has not been removed
        // note: it may be on my neighbor's list, but already processed and removed, we need
        // to make sure we will be decrementing a degree from a neighbor that's already in the graph
        if ( m_elementList[ iNeighbor ] != NULL )
        {
            // remove edge from this vertex
            m_nEdges --;

            // decrement degree from neighbor
            DecrementDegree( iNeighbor );//, iCurrentDegree );
        }
    }


    // reset degree
    RemoveFromVertexVector( iHighestDegreeVertex, highestDegreeVertex->getDegree() );//neighbors->size() );

    // remove edges to neighbors, and let the vertex linger and ...
    //neighbors.clear( );

    return iHighestDegreeVertex;
}

ListNode* VertexVectorAdjacencyList::GetHighestDegreeVertex( )
{
    for(int i = m_lastHighestDegree; i >= 0; i--)
    {
        if(m_vectorVertex[i]->size() != 0)
        {
            m_lastHighestDegree = i;
            return m_vectorVertex[i]->getFirst();//)->getVertex();
        }
    }

    return NULL;
}

void VertexVectorAdjacencyList::updateData()
{
    for( int i = 0; i < m_nVertex; i++ )
    {
        int degree = m_arrAdjLists[i]->size();

        // the vertex degree is its position in the vector
        ListNode* node = m_vectorVertex[ degree ]->insertAtEnd( i );
        node->setDegree( degree );

        m_elementList[i] = node;
    }
}
```

```
#ifndef LIST_H
#define LIST_H

#include <utility>
#include "ListNode.h"
/************************************************************************/
/*                                                                      */
/*            Tipo de dados lista genÃ©rica                      */
/*                                                                      */
/************************************************************************/

class List
{
public:
    List();
    ~List();

    ListNode* insertAtEnd(int content);

    int insertAtFront(int content);

    int removeFirst();

    int size();

    ListNode * getFirst() { return m_first; }

    ListNode * getLast() { return m_last; }

    void erase(int content);

    void remove( ListNode* node );

private:
    int m_numElems;
    ListNode * m_first;
    ListNode * m_last;
};

#endif

```

```
#include "List.h"
#include "ListNode.h"
#include <stdio.h>

List::List()
{
    m_first = NULL;
    m_last = NULL;
    m_numElems = 0;
}

List::~List()
{
    // DOES NOTHING
}

ListNode* List::insertAtEnd(int content)
{
    ListNode * node = new ListNode(content);
    ListNode * previous = NULL;

    /* Primeiro elemento */
    if ((m_first == NULL) && (m_last == NULL)) {
        m_first = node;
        m_last = node;
    } else {
        node->setPrevious( m_last );
        previous = m_last;
        m_last->setNext(node);
        m_last = node;
    }

    m_numElems++;

    return node;
}

int List::insertAtFront(int content)
{
    ListNode * node = new ListNode(content);

    /* Primeiro elemento */
    if ((m_first == NULL) && (m_last == NULL)) {
        m_first = node;
        m_last = node;
    } else {
        node->setNext(m_first);
        m_first = node;
    }

    m_numElems++;

    return 0;
}

int List::removeFirst()
{
    if (m_numElems == 0) {
        return -1;
    }

    ListNode * node = m_first;

    int content = node->getVertex();

    m_first = m_first->next();
    m_first->setPrevious( NULL );
    delete node;

    m_numElems--;

    return content;
}

int List::size()
{
    return m_numElems;
}

void List::erase(int content)
{
    ListNode * previous = NULL;

    for(ListNode * node = m_first; node != NULL; node = node->next()) {

        if(content == node->getVertex()) {

            // Primeiro da lista
            if (node == m_first) {
                if (m_numElems == 1) {
                    m_first = NULL;
                    m_last = NULL;
                } else {
                    m_first = node->next();
                    m_first->setPrevious( NULL );
                }
            } else {
                previous->setNext(node->next());

                if (node == m_last) {
                    m_last = previous;
                }
                else
                {
                    node->next()->setPrevious( previous );
                }
            }


                m_numElems--;
            delete node;
            return;
        }
        previous = node;
    }
}


void List::remove( ListNode* node )
{
    if (node == NULL)
        return;

    ListNode* previous = node->previous();

    // primeiro da lista
    if (node == m_first)
    {
        if (m_numElems == 1)
        {
            m_first = NULL;
            m_last = NULL;
        }
        else
        {
            m_first = node->next();
            m_first->setPrevious( NULL );
        }
    }
    else
    {
        previous->setNext(node->next());

        if (node == m_last)
        {
            m_last = previous;
        }
        else
        {
            node->next()->setPrevious( previous );
        }
    }

    m_numElems--;

    delete node;
}
```

```

#ifndef LIST_NODE_H
#define LIST_NODE_H

#include <stdio.h>

class ListNode
{
public:
    ListNode(int content);
    ~ListNode();

    int getVertex() { return m_vertex; }
    int getDegree() { return m_degree; }
    void setDegree(int degree) { m_degree = degree; }
    ListNode * next() { return m_next; }
    void setNext(ListNode * node) { m_next = node; }
        ListNode * previous() { return m_previous; }
    void setPrevious(ListNode * node) { m_previous = node; }

private:
    int m_vertex;
    int m_degree;
    ListNode * m_next;
    ListNode* m_previous;
};

#endif


```

```cpp

#include "ListNode.h"


ListNode::ListNode(int content)
:
    m_vertex(content),
    m_degree(-1),
    m_next(NULL),
    m_previous(NULL)
{

}

ListNode::~ListNode()
{

}
```

```cpp
# include <stdio.h>
#include "Heap.h"

int main(int argc, char * argv[])
{
    Heap heap(10);

    fprintf(stderr, "inserindo 0, 3\n");

    heap.insertOnHeap(0, 3);

    fprintf(stderr, "inserindo 1, 1\n");

    heap.insertOnHeap(1, 1);

    fprintf(stderr, "inserindo 2, 3\n");

    heap.insertOnHeap(2, 3);

    fprintf(stderr, "inserindo 3, 2\n");

    heap.insertOnHeap(3, 2);

    fprintf(stderr, "inserindo 4, 2\n");

    heap.insertOnHeap(4, 2);

    fprintf(stderr, "inserindo 5, 3\n");

    heap.insertOnHeap(5, 3);

    fprintf(stderr, "inserindo 6, 2\n");

    heap.insertOnHeap(6, 2);

    fprintf(stderr, "inserindo 7, 1\n");

    heap.insertOnHeap(7, 1);

    fprintf(stderr, "inserindo 8, 4\n");

    heap.insertOnHeap(8, 4);

    fprintf(stderr, "inserindo 9, 1\n");

    heap.insertOnHeap(9, 1);

    heap.print();
    fprintf(stderr, "\n");

    heap.DecrementDegree(2);

    heap.print();


    fprintf(stderr, "Removendo da heap:\n");


    for(int i = 0; i < 20; i++) {
        std::pair<int, int> element;
        heap.removeFromHeap(element);

        fprintf(stderr, "%d - vertex: %d - degree: %d\n", i, element.first, element.second);
    }

    return 0;
}
```

```

#include <stdio.h>

#include "GeraGrafo.h"
#include "DegreeVectorAdjacencyList.h"

#define NVERTEX 10
#define FILENAME_GRAFO "testedegrafo_gerado.grafo"
#define FILENAME_GRAFO_LIDO "testedegrafo_lido.grafo"

int main( int argc, char ** argv )
{
    int debug = 2;

    fprintf( stderr, "Gerando grafo com %d vertices\n", NVERTEX );

    FILE * f = fopen( FILENAME_GRAFO, "w" );

    GeraGrafo( f, NVERTEX, 1, debug );

    fclose( f );

    f = fopen( FILENAME_GRAFO, "r" );

    DegreeVectorAdjacencyList grafo;

    fprintf( stderr, "lendo arquivo do grafo gerado: %s\n", FILENAME_GRAFO );

    grafo.read( f, debug );

    fclose( f );

    f = fopen( FILENAME_GRAFO_LIDO, "w" );

    fprintf( stderr, "escrevendo arquivo do grafo lido: %s\n", FILENAME_GRAFO_LIDO );

    grafo.write( f );

    fclose ( f );

    return 0;
}
```