```cpp
#ifndef _TP1_MATRIX_H_
#define _TP1_MATRIX_H_ 1

#include <stdexcept>
#include <cassert>
#include <utility>
#include <iostream>
#include "BDouble.h"

enum Solutions {
    INFINITE,
    SINGLE,
    NONE
};

// Este magic number nos dice cuándo convertir automáticamente una matriz banda en una matriz normal.
#define MAGIC_NUMBER 562154

/*
* Matriz Banda.
*/

class Matrix {
    friend std::ostream &operator<<(std::ostream &, const Matrix &);

public:
    Matrix(const Matrix &m)
            : N(m.rows()), M(m.columns()), uband(m.upper_bandwidth()), lband(m.lower_bandwidth()) {
        int bound = this->lower_bandwidth() + this->upper_bandwidth() + 1;
        this->matrix = new BDouble *[this->rows()];

        for (int i = 0; i < this->rows(); ++i) {
            this->matrix[i] = new BDouble[bound];

            for (int j = 0; j < bound; ++j) {
                this->matrix[i][j] = m.matrix[i][j];
            }
        }
    }

    Matrix(int N, int M, int lband = MAGIC_NUMBER, int uband = MAGIC_NUMBER)
            : N(N), M(M), uband(uband), lband(lband) {


        if (this->rows() == 0 || this->columns() == 0) {
            throw new std::out_of_range("Invalid matrix dimension");
        }

        if (lband > N) {
            this->lband = N - 1;
        }

        if (uband > M) {
            this->uband = M - 1;
        }

        int bound = this->lower_bandwidth() + this->upper_bandwidth() + 1;
        this->matrix = new BDouble *[this->rows()];

        for (int i = 0; i < this->rows(); ++i) {
            this->matrix[i] = new BDouble[bound];

            for (int j = 0; j < bound; ++j) {
                this->matrix[i][j] = 0.0;
            }
        }
    }

    inline int rows() const {
        return this->N;
    }

    inline int columns() const {
        return this->M;
    }

    inline int upper_bandwidth() const {
        return this->uband;
    }

    inline int lower_bandwidth() const {
        return this->lband;
    }

    BDouble &operator()(const int &i, const int &j) {
```

```cpp
        if (i >= this->rows() || j >= this->columns()) {
            throw new std::out_of_range("Index access out of range");
        }

        if (i <= j + this->lower_bandwidth() && j <= i + this->upper_bandwidth()) {
            return matrix[i][j - i + this->lower_bandwidth()];
        } else {
            throw new std::out_of_range("Out of modifiable range");
        }
    }

    const BDouble &operator()(const int &i, const int &j) const {
        if (i >= this->rows() || j >= this->columns()) {
            throw new std::out_of_range("Index access out of range");
        }

        if (i > j + this->lower_bandwidth()) {
            return zero;
        } else if (j > i + this->upper_bandwidth()) {
            return zero;
        } else {
            return matrix[i][j - i + this->lower_bandwidth()];
        }
    }

    ~Matrix() {
        for (int i = 0; i < this->rows(); ++i) {
            delete[] this->matrix[i];
        }

        delete[] this->matrix;
    }

private:
    // Matrix
    int N;
    int M;
    int uband;
    int lband;
    BDouble **matrix;
};

std::ostream &operator<<(std::ostream &os, const Matrix &m) {
    for (int i = 0; i < m.rows(); ++i) {
        for (int j = 0; j < m.columns(); ++j) {
            os << m(i, j) << " ";
        }

        os << std::endl;
    }

    os << std::endl;

    return os;
}

/*********************************************************************************************************************
 * Acá empieza la parte de resolver los sistemas.
 ********************************************************************************************************************/

// m tiene que estar triangulada
// el usuario libera la memoria
std::pair<BDouble *, enum Solutions> backward_substitution(const Matrix &m, BDouble *b) {
    BDouble *x = new BDouble[m.columns()];
    enum Solutions solution = SINGLE;

    int N = std::min(m.rows(), m.columns());

    for (int d = N - 1; d >= 0; d--) {
        if (m(d, d) == 0.0) {
            x[d] = 1.0;
            solution = INFINITE;
        } else {
            int bound = std::min(m.columns(), d + m.upper_bandwidth() + 1);
            x[d] = b[d];

            for (int j = d + 1; j < bound; ++j) {
                x[d] -= m(d, j) * x[j];
            }

            x[d] /= m(d, d);

        }
    }

    return std::pair<BDouble *, enum Solutions>(x, solution);
```

```cpp
}

// m tiene que estar triangulada
// el usuario libera la memoria
std::pair<BDouble *, enum Solutions> forward_substitution(const Matrix &m, BDouble *b) {
    BDouble *x = new BDouble[m.columns()];
    enum Solutions solution = SINGLE;

    int N = std::min(m.rows(), m.columns());

    for (int d = 0; d < N; ++d) {
        if (m(d, d) == 0.0) {
            x[d] = 1.0;
            solution = INFINITE;
        } else {
            int bound = std::max(0, d - m.lower_bandwidth() - 1);
            x[d] = b[d];

            for (int j = bound; j < d; ++j) {
                x[d] -= m(d, j) * x[j];
            }

            x[d] /= m(d, d);
        }
    }

    return std::pair<BDouble *, enum Solutions>(x, solution);
}

std::pair<BDouble *, enum Solutions> gaussian_elimination(Matrix workspace, BDouble *b) {
    int diagonal = std::min(workspace.columns(), workspace.rows());

    for (int d = 0; d < diagonal; ++d) {
        // Tenemos algo distinto de cero en la base
        for (int i = d + 1; i < std::min(workspace.rows(), d + workspace.lower_bandwidth() + 1); ++i) {
            // Tenemos algo distinto de cero en alguna fila más abajo
            BDouble coefficient = workspace(i, d) / workspace(d, d);

            // Realizamos el mismo cambio en la solución del sistema
            b[i] -= coefficient * b[d];

            for (int j = d + 1; j < std::min(d + workspace.upper_bandwidth() + 1, workspace.columns()); ++j) {
                // Realizamos la resta a toda la fila.
                workspace(i, j) -= coefficient * workspace(d, j);
            }

            // Setear esto en 0 debería reducir el error posible (por ejemplo, restando números muy chicos)
            workspace(i, d) = 0.0;
        }
    }

    return backward_substitution(workspace, b);
}


std::pair<Matrix, Matrix> LU_factorization(const Matrix &A) {
    // El tamaño de la diagonal
    int N = std::min(A.rows(), A.columns());

    // Matriz L triangular inferior, U triangular superior
    Matrix L(A.rows(), A.columns(), A.lower_bandwidth(), 0);
    Matrix U(A.rows(), A.columns(), 0, A.upper_bandwidth());


    //std::cout << "A" << std::endl;
    //std::cout << A << std::endl;
    //std::cout << "L" << std::endl;
    //std::cout << "rows: " << L.rows() << std::endl;
    //std::cout << "columns: " << L.columns() << std::endl;
    //std::cout << "upper_band: " << L.upper_bandwidth() << std::endl;
    //std::cout << "lower_band: " << L.lower_bandwidth() << std::endl;
    //std::cout << L << L.lower_bandwidth() << std::endl;
    //std::cout << "U" << std::endl;
    //std::cout << "rows: " << U.rows() << std::endl;
    //std::cout << "columns: " << U.columns() << std::endl;
    //std::cout << "upper_band: " << U.upper_bandwidth() << std::endl;
    //std::cout << "lower_band: " << U.lower_bandwidth() << std::endl;
    //std::cout << U << std::endl;
    // Las inicializamos como la matriz identidad
    for (int i = 0; i < N; ++i) {
        L(i, i) = 1.0;
        U(i, i) = 1.0;
    }

    // Elegimos, arbitrariamente, que L(0,0) * U(0,0) = A(0,0)
    U(0, 0) = A(0, 0);
```

```cpp
    L(0, 0) = 1.0;

    if (U(0, 0) == 0.0) {
        // No podemos factorizar
        throw new std::out_of_range("Factorization impossible");
    }

    //Set first row of U and firt column of L
    int M = std::min(A.upper_bandwidth(), A.lower_bandwidth());
    //std::cout << "First cicle init..." << std::endl;
    for (int i = 1; i <= M; i++) {
        //std::cout << "i: " << i << std::endl;
        U(0, i) = A(0, i) / L(0, 0);
        L(i, 0) = A(i, 0) / U(0, 0);
    }
    //std::cout << "First cicle end..." << std::endl;

    //Set rows/columns from 1 to n-1
    //std::cout << "second cicle init..." << std::endl;
    for (int i = 1; i < N - 1; i++) {
        //std::cout << "i: " << i << std::endl;
        U(i, i) = A(i, i);
        //Aprovechamos banda
        int bound = std::min(A.lower_bandwidth(), A.upper_bandwidth());
        for (int h = 1; h <= bound; h++) {
            //std::cout << "h: " << h << std::endl;
            if (i >= h) {
                U(i, i) -= L(i, i - h) * U(i - h, i);
            }
        }
        U(i, i) /= L(i, i);

        if (U(i, i) == 0.0) {
            // No podemos factorizar
            throw new std::out_of_range("Factorization impossible");
        }

        //Estamos abusando de que las matrices van a tener la misma banda
        //inferior y superior... esto podria no ser asi.
        int M = std::min(A.upper_bandwidth(), A.lower_bandwidth());
        for (int k = 1; k <= M; k++) {
            int j = i + k;
            //std::cout << "k: " << k << std::endl;
            if (j < U.columns()) {
                //std::cout << "A(" << i << "," << j << ") "<< std::endl;
                U(i, j) = A(i, j);
            }

            if (j < L.rows()) {
                //std::cout << "A(" << j << "," << i << ") "<< std::endl;
                L(j, i) = A(j, i);

            }



            //Aprovechamos banda
            int bound = std::min(A.lower_bandwidth(), A.upper_bandwidth());
            //std::cout << "bound: " << bound << std::endl;
            for (int h = 1; h <= bound - k; h++) {
                //std::cout << "h: " << h << std::endl;
                if (i >= h) {
                    if (j < U.columns()) {
                        //std::cout << "U(" << i << "," << j << ") "<< std::endl;
                        //std::cout << "L(" << i << "," << i-h << ") "<< std::endl;
                        //std::cout << "U(" << i-h << "," << j << ") "<< std::endl;
                        U(i, j) -= L(i, i - h) * U(i - h, j); // i° ROW OF U
                    }
                    if (j < L.rows()) {
                        //std::cout << "L(" << j << "," << i << ") "<< std::endl;
                        //std::cout << "L(" << j << "," << i-h << ") "<< std::endl;
                        //std::cout << "U(" << i-h << "," << i << ") "<< std::endl;
                        L(j, i) -= L(j, i - h) * U(i - h, i); // j° COLUMN OF L
                    }
                }
            }
            //std::cout << "cicle end..."<< std::endl;

            if (j < U.columns()) {
                //std::cout << "U(" << i << "," << j << ") "<< std::endl;
                U(i, j) /= L(i, i);
            }
            if (j < L.rows()) {
                //std::cout << "L(" << j << "," << i << ") "<< std::endl;
                L(j, i) /= U(i, i);
            }
```

```
        }

    }
    //std::cout << "second cicle end..." << std::endl;

    //Set last position
    U(N - 1, N - 1) = A(N - 1, N - 1);

    int bound = std::min(A.lower_bandwidth(), A.upper_bandwidth());
    for (int h = 1; h <= bound; h++) {
        U(N - 1, N - 1) -= L(N - 1, N - 1 - h) * U(N - 1 - h, N - 1);
    }
    U(N - 1, N - 1) /= L(N - 1, N - 1);

    return std::pair<Matrix, Matrix>(L, U);
}

/**
* - A matrix original del sistema.
* - L matriz triangular inferior de la descomposicion.
* - U matriz triangular superior de la descomposicion.
* - i fila del elemento a modificar.
* - j columna del elemento a modificar.
* - a nuevo valor de la posicion (i, j)
**/
std::pair<BDouble *, enum Solutions> sherman_morrison(Matrix &A, Matrix &L, Matrix &U,
                                                       int i, int j, BDouble a, BDouble *b) {
    int N = std::min(A.rows(), A.columns());

    //Sherman-Morrison formula vectors
    //Altered system: A2 = (A + uv')
    BDouble *u = new BDouble[N];
    BDouble *v = new BDouble[N];

    for (int k = 0; k < N; k++) {
        u[k] = 0.0;
        v[k] = 0.0;
    }
    //Column vector
    u[i] = 1.0;

    //Row vector
    v[j] = a - A(i, j);

    //From Sherman-Morrison
    // A^-1 b = y <=> Ay = b
    // A^-1 u = z <=> Az = u

    //First we solve:
    // L y2 = b and L z2 = u
    BDouble *y2;
    BDouble *z2;

    std::pair<BDouble *, enum Solutions> solution;
    solution = forward_substitution(L, b);
    y2 = solution.first;
    solution = forward_substitution(U, u);
    z2 = solution.first;

    //Then we solve:
    // U y = y2 and U z = z2
    BDouble *y;
    BDouble *z;
    solution = backward_substitution(L, y2);
    y = solution.first;
    solution = backward_substitution(U, z2);
    z = solution.first;
    delete[] y2;
    delete[] z2;

    //Finally x = y - z * [(v' y)/(1 + v' z)]
    BDouble *x = new BDouble[N];

    //First we calculate k = (v' y)/(1 + v' z) (scalar value)
    BDouble vy = 0.0;
    BDouble vz = 1.0;
    for (int h = 0; h < N; h++) {
        vy += v[h] * y[h];
        vz += v[h] * z[h];
    }

    //Finally we calculate x = y + z * k
    for (int h = 0; h < N; h++) {
        x[h] = y[h] - (z[h] * (vy / vz));
    }
```

```cpp
    delete[] y;
    delete[] z;

    return std::pair<BDouble *, enum Solutions>(x, SINGLE);
}

std::pair<BDouble *, enum Solutions> sherman_morrison(
        Matrix &L,
        Matrix &U,
        BDouble *u,
        BDouble *v, BDouble *b) {
    int N = std::min(L.rows(), L.columns());

    //Sherman-Morrison formula vectors
    //Altered system: A2 = (A + uv')

    //From Sherman-Morrison
    // A^-1 b = y <=> Ay = b
    // A^-1 u = z <=> Az = u

    //First we solve:
    // L y2 = b and L z2 = u
    BDouble *y2;
    BDouble *z2;

    std::pair<BDouble *, enum Solutions> solution;
    solution = forward_substitution(L, b);
    y2 = solution.first;
    solution = forward_substitution(L, u);
    z2 = solution.first;

    //Then we solve:
    // U y = y2 and U z = z2
    BDouble *y;
    BDouble *z;
    solution = backward_substitution(U, y2);
    y = solution.first;
    solution = backward_substitution(U, z2);
    z = solution.first;
    delete[] y2;
    delete[] z2;

    //Finally x = y - z * [(v' y)/(1 + v' z)]
    BDouble *x = new BDouble[N];

    //First we calculate k = (v' y)/(1 + v' z) (scalar value)
    BDouble vy = 0.0;
    BDouble vz = 1.0;
    for (int h = 0; h < N; h++) {
        vy += v[h] * y[h];
        vz += v[h] * z[h];
    }
    BDouble k = (vy / vz);

    //Finally we calculate x = y - z * k
    for (int h = 0; h < N; h++) {
        x[h] = y[h] - (z[h] * k);
    }
    delete[] y;
    delete[] z;

    return std::pair<BDouble *, enum Solutions>(x, SINGLE);
}


#endif //_TP1_MATRIX_H_
```