```
#include "tridiagonal_matrix.h"

tridiagonal_matrix::tridiagonal_matrix(int m)
{
        dimension = m;
        diag.resize(m);
        upperdiag.resize(m - 1);
        lowerdiag.resize(m - 1);
        hatupperdiag.resize(m - 1);
        r.resize(m - 1);
        transformed = false;

}

int tridiagonal_matrix::get_dimension() const
{
        return dimension;
}

void tridiagonal_matrix::set_diagonal_entry(int i, double val)
{
        diag[i] = val;
}

void tridiagonal_matrix::set_upper_diagonal_entry(int i, double val)
{
        upperdiag[i] = val;
}

void tridiagonal_matrix::set_lower_diagonal_entry(int i, double val)
{
        lowerdiag[i] = val;
}

double tridiagonal_matrix::get_diagonal_entry(int i) const
{
        return diag[i];
}

double tridiagonal_matrix::get_upper_diagonal_entry(int i) const
{
        return upperdiag[i];
}

double tridiagonal_matrix::get_lower_diagonal_entry(int i) const
{
        return lowerdiag[i];
}

double tridiagonal_matrix::get_r_entry(int i) const
{
        return r[i];
}

double tridiagonal_matrix::get_hat_upper_diagonal_entry(int i) const
{
        return hatupperdiag[i];
}
```

```
bool tridiagonal_matrix::is_transformed() const
{
        return transformed;
}

void tridiagonal_matrix::add_to_diagonal_entry(int i, double val)
{
        diag[i] += val;
}

void tridiagonal_matrix::add_to_upper_diagonal_entry(int i, double val)
{
        upperdiag[i] += val;
}

void tridiagonal_matrix::add_to_lower_diagonal_entry(int i, double val)
{
        lowerdiag[i] += val;
}

// this is how the matrix is transformed into upper daigonal
void tridiagonal_matrix::transform()
{
        // Check to see if tridiag matrix has already been transformed
        if (transformed)
        {
                return;
        }

        // Implement the transformation here. Fill r and hatupperdiag


        //create hatupperdiag[0]
        hatupperdiag[0] = upperdiag[0] / diag[0];

        // for loop to create r[i-1] and and hatupperdaig[i], i=1...i=dimension-
2
        for (int i = 1; i ≤ dimension - 2; i++)
        {
                // create r[i-1]
                r[i - 1] = diag[i] - lowerdiag[i - 1] * hatupperdiag[i - 1];
                // create hatupperdaig[i]
                hatupperdiag[i] = upperdiag[i] / r[i - 1];
        }

        // create r[m-2]
        r[dimension - 2] = diag[dimension - 1] - lowerdiag[dimension - 2] * hatu
pperdiag[dimension - 2];


        ////!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*
        ////Print out of  e_hat to test transform function
        //cout << "hatupperdiag = (";
        //for (int i = 0; i < dimension - 1; i++)
        //{
        //      cout << hatupperdiag[i] << ",";
        //}
        //cout << ")" << endl;
        ////!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*
```

```
        transformed = true;
}

//given vector rhs, return sol satisfying A sol = rhs.

vector <double>  tridiagonal_matrix::solve_linear_system(const vector <double> &
 rhs) const
{


        vector<double> hatrhs(dimension); // modified rhs

        vector<double> sol(dimension); // this is where the soln is stored

    // Implement the calculation of hatrhs and the backward solve

        // Calculate hatrhs

        // Calculate hatrhs[0]
        hatrhs[0] = rhs[0] / diag[0];

        // Calculate hatrhs[1] to hatrhs[m-2]

        for (int i = 1; i ≤ dimension - 2; i++)
        {
                hatrhs[i] = (rhs[i] - (lowerdiag[i - 1] * hatrhs[i - 1])) / r[i
- 1];
        }

        // Calculate hatrhs[m-1]

        hatrhs[dimension - 1] = (rhs[dimension - 1] - (lowerdiag[dimension - 2]
* hatrhs[dimension - 2])) / r[dimension - 2];


        ////!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*
        ////Print out of  hatrhs to test solve linear system function
        //cout << "hatrhs = (";
        //for (int i = 0; i <= dimension - 1; i++)
        //{
        //      cout << hatrhs[i] << ",";
        //}
        //cout << ")" << endl;
        ////!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*!*



        // Calculate sol[m-1] to Sol[0]

        //Assign sol[dimesion-1]
        sol[dimension - 1] = hatrhs[dimension - 1];

        //calculate and assign sol[m-2] to sol[0]
        for (int i = dimension - 2; i ≥ 0; i--)
        {
                sol[i] = hatrhs[i] - (hatupperdiag[i] * sol[i + 1]);
        }
```

```
        return sol;
}

// given vector lhs, return rhs = A lhs.
vector<double> tridiagonal_matrix::Mult(const vector<double> & lhs) const
{
        vector<double> rhs(dimension, 0.0);

        // Implement the matrix-vector multiplication here.
        // Transformed vs not transformed are slightly diffrent

        // If the TDMatrix is Transformed
        if (transformed)
        {
                //since the matrix is transfomred, c_i=0 for all i=0,..,m-1
                // and doesn't need to be included in the function also the uppe
rdiag
                // called for should be hatupperdiag. also since in a transforme
d
                // Tridaigoanl matrix diag[i] =1  it can be simplified to  just
                // 1 * lhs[i] + hatupperdiag[i ] * lhs[i+1]


                // calculate rhs[i] for i = 0 to i = m - 2
                for (int i = 1; i < dimension - 1; i++)
                {
                        rhs[i] = lhs[i] + hatupperdiag[i] * lhs[i + 1];
                }

                // calculate rhs[m-1]
                rhs[dimension - 1] = lhs[dimension - 1];
        }

        //if TDMatrix is not transformed
        if (transformed ≡ false)
        {
                // calculate rhs[0]
                rhs[0] = diag[0] * lhs[0] + upperdiag[0] * lhs[1];

                // calculate rhs[i] for i = 1 to i = m - 2
                for (int i = 1; i < dimension - 1; i++)
                {
                        rhs[i] = lowerdiag[i - 1] * lhs[i - 1] + diag[i] * lhs[i
] + upperdiag[i] * lhs[i + 1];
                }

                // calculate rhs[m-1]
                rhs[dimension - 1] = lowerdiag[dimension - 2] * lhs[dimension -
2] + diag[dimension - 1] * lhs[dimension - 1];
        }


        return rhs;
}

//copy constructor
tridiagonal_matrix::tridiagonal_matrix(const tridiagonal_matrix *mat)
{
```

```
        dimension = mat→get_dimension();
        diag.resize(dimension);
        upperdiag.resize(dimension - 1);
        lowerdiag.resize(dimension - 1);
        hatupperdiag.resize(dimension - 1);
        r.resize(dimension - 1);
        for (int i = 0; i<dimension; i++)
                diag[i] = mat→get_diagonal_entry(i);
        for (int i = 0; i<dimension - 1; i++)
        {
                upperdiag[i] = mat→get_upper_diagonal_entry(i);
                lowerdiag[i] = mat→get_lower_diagonal_entry(i);
        }

        transformed = mat→is_transformed();
        if (transformed)
        {
                for (int i = 0; i<dimension - 1; i++)
                {
                        r.push_back(mat→get_r_entry(i));
                        hatupperdiag.push_back(mat→get_hat_upper_diagonal_entry
(i));
                }
        }
}

//empty destructor
tridiagonal_matrix::~tridiagonal_matrix()
{
        ;
}

// Coment Psuedocode created by Brent
// Transform() created by Shashwata excpet for line 83
// solve_linear_system implemented by Shashwata
// Some debuging done by Brent
// Mult implemented by Brent
// get_r and get_hat_upper_daig implemented by Brent
// Copy Constructor implemented by Will
```