// Advanced Data Structures - Assignment - 2 :

// Submitted by

// Vishnu Prasad Vishwanathan

// 793782749

// vvishwan@syr.edu

#include "stdafx.h"

//HW2: Sparse Matrix Operation -- Applicaiton of 2-D linked lists

//by C-Y (Roger) Chen

//Due: 11:59pm, Wed. September 27

//Implement the following three member functions for the class my\_matrix:

//add\_node, operator+, operator\*

//Do not change the main function and the my\_matrix::print function

#include <iostream>

#include <vector>

using namespace std;

class node {

public:

int value;

int r\_position;

int c\_position;

node \* r\_next;

node \* c\_next;

node() { value = -9999; r\_position = c\_position = -1; r\_next = c\_next = nullptr; }

node(int v, int r, int c) { value = v; r\_position = r; c\_position = c; r\_next = c\_next = nullptr; }

};

class my\_matrix {

public:

int num\_rows;

int num\_cols;

vector<node \*> r\_vec;

vector<node \*> c\_vec;

my\_matrix() {}

my\_matrix(int r, int c);

//Initialize a matrix of rows and c cols.

my\_matrix(int r, int c, int n, int k);

//Initializa a matrix of r rows, c cols, n nodes, with values randomly in - (k-1) //... (k-1);

void print(); //print value, r\_position, and c\_position of all matrix elements in //row-oriented manner

//Implement the following three member functions:

void add\_node(int v, int r, int c);

//Create a new node with value v at row r and col c.

//If there already exists an element at this position, then just update the node

//value at this position to v, and do not create a new node.

my\_matrix operator+(my\_matrix M);//matrix addition

my\_matrix operator\*(my\_matrix M); //matrix multiplication

};

my\_matrix::my\_matrix(int r, int c) {

num\_rows = r;

num\_cols = c;

r\_vec.resize(r, nullptr);

c\_vec.resize(c, nullptr);

}

my\_matrix::my\_matrix(int r, int c, int n, int k) {

num\_rows = r;

num\_cols = c;

r\_vec.resize(r, nullptr);

c\_vec.resize(c, nullptr);

for (int i = 0; i < n; i++) {

int vv = rand() % (2 \* k - 1) - (k - 1);

int rr = rand() % r;

int cc = rand() % c;

add\_node(vv, rr, cc);

}

}

void my\_matrix::add\_node(int v, int r, int c) {

int first\_run = 0;

bool head\_swap = false;

bool rnot\_found = true;

bool cnot\_found = true;

bool col\_add = true;

node \* head = new node();

node \* temp\_node = new node();

node \* n = new node();

node \* pre\_node = new node();

node \* n3 = new node();

if (v != 0)

{

//Add the element to the row vector of the matrix

if (r\_vec[r] == nullptr)

{

node \* n1 = new node(v, r, c);

r\_vec[r] = n1;

rnot\_found = false;

}

else

{

pre\_node->value = 0;

pre\_node->r\_next = r\_vec[r];

while (pre\_node->r\_next != nullptr && pre\_node->r\_next->c\_position <= c)

{

if (pre\_node->r\_next->c\_position == c)

{

rnot\_found = false;

pre\_node->r\_next->value = v;

}

pre\_node = pre\_node->r\_next;

}

}

if (rnot\_found)

{

node \* n2 = new node(v, r, c);

if (pre\_node->r\_next == nullptr)

{

pre\_node->r\_next = n2;

}

else if (pre\_node->r\_next->c\_position > c)

{

if (pre\_node->r\_next == r\_vec[r]) { head\_swap = true; }

temp\_node = pre\_node->r\_next;

pre\_node->r\_next = n2;

n2->r\_next = temp\_node;

if (head\_swap)

{

r\_vec[r] = n2;

}

}

else

{

}

}

//Add the element to the column vector of the matrix

if (c\_vec[c] == nullptr)

{

node \* n1 = new node(v, r, c);

c\_vec[c] = n1;

cnot\_found = false;

}

else

{

n3->value = 0;

n3->c\_next = c\_vec[c];

while (n3->c\_next != nullptr && n3->c\_next->r\_position <= r)

{

if (n3->c\_next->r\_position == r)

{

cnot\_found = false;

n3->c\_next->value = v;

}

n3 = n3->c\_next;

}

}

if (cnot\_found)

{

node \* n2 = new node(v, r, c);

if (n3->c\_next == nullptr)

{

n3->c\_next = n2;

}

else if (n3->c\_next->r\_position > r)

{

head\_swap = false;

if (n3->c\_next == c\_vec[c]) { head\_swap = true; }

temp\_node = n3->c\_next;

n3->c\_next = n2;

n2->c\_next = temp\_node;

if (head\_swap)

{

c\_vec[c] = n2;

}

}

else

{

}

}

}

}

my\_matrix my\_matrix::operator+(my\_matrix M) //overloading +operator for the addition of two matrices

{

node \*n1 = new node();

node \*n2 = new node();

bool n1\_big = true;

int vv, rr, cc, rows, cols, diff;

if (num\_rows > M.num\_rows)

{

rows = num\_rows;

diff = num\_rows - M.num\_rows;

}

else

{

rows = M.num\_rows;

diff = M.num\_rows - num\_rows;

}

if (num\_cols > M.num\_cols)

cols = num\_cols;

else

cols = M.num\_cols;

my\_matrix result(rows, cols);

for (int i = 0; i < rows; i++)

{

if (r\_vec[i] == nullptr)

{

if (M.r\_vec[i] != nullptr)

{

n2 = M.r\_vec[i];

while (n2 != nullptr)

{

rr = n2->r\_position;

cc = n2->c\_position;

vv = n2->value;

result.add\_node(vv, rr, cc);

n2 = n2->r\_next;

}

}

}

else if (M.r\_vec[i] == nullptr)

{

if (r\_vec[i] != nullptr)

{

n1 = r\_vec[i];

while (n1 != nullptr)

{

rr = n1->r\_position;

cc = n1->c\_position;

vv = n1->value;

result.add\_node(vv, rr, cc);

n1 = n1->r\_next;

}

}

}

else if (r\_vec[i] != nullptr && M.r\_vec[i] != nullptr)

{

n1 = r\_vec[i];

n2 = M.r\_vec[i];

while (n1 != nullptr && n2 != nullptr)

{

if (n1->c\_position == n2->c\_position)

{

rr = n1->r\_position;

cc = n1->c\_position;

vv = n1->value + n2->value;

result.add\_node(vv, rr, cc);

n1 = n1->r\_next;

n2 = n2->r\_next;

}

else if (n1->c\_position < n2->c\_position)

{

rr = n1->r\_position;

cc = n1->c\_position;

vv = n1->value;

result.add\_node(vv, rr, cc);

n1 = n1->r\_next;

}

else if (n1->c\_position > n2->c\_position)

{

rr = n2->r\_position;

cc = n2->c\_position;

vv = n2->value;

result.add\_node(vv, rr, cc);

n2 = n2->r\_next;

}

}

if (n1 == nullptr)

{

if (n2 != nullptr)

{

while (n2 != nullptr)

{

rr = n2->r\_position;

cc = n2->c\_position;

vv = n2->value;

result.add\_node(vv, rr, cc);

n2 = n2->r\_next;

}

}

}

else if (n2 == nullptr)

{

if (n1 != nullptr)

{

while (n1 != nullptr)

{

rr = n1->r\_position;

cc = n1->c\_position;

vv = n1->value;

result.add\_node(vv, rr, cc);

n1 = n1->r\_next;

}

}

}

}

}

return result;

}

my\_matrix my\_matrix::operator\*(my\_matrix M) //overloading the \* operator for multiplying two matrices

{

node \*n1 = new node();

node \*n2 = new node();

bool n1\_big = true;

int vv, rr, cc, rows, cols, diff;

if (num\_rows > M.num\_rows)

{

rows = num\_rows;

diff = num\_rows - M.num\_rows;

}

else

{

rows = M.num\_rows;

diff = M.num\_rows - num\_rows;

}

if (num\_cols > M.num\_cols)

cols = num\_cols;

else

cols = M.num\_cols;

my\_matrix result(rows, cols);

for (int i = 0; i < num\_rows; i++)

{

if (r\_vec[i] != nullptr)

{

n1 = r\_vec[i];

for (int j = 0; j < M.num\_cols; j++)

{

if (M.c\_vec[j] != nullptr)

{

n1 = r\_vec[i];

n2 = M.c\_vec[j];

vv = 0;

while (n1 != nullptr && n2 != nullptr)

{

if (n1->c\_position == n2->r\_position)

{

vv += ((n1->value)\*(n2->value));

n1 = n1->r\_next;

n2 = n2->c\_next;

}

else if (n1->c\_position < n2->r\_position)

{

n1 = n1->r\_next;

}

else if (n2->r\_position < n1->c\_position)

{

n2 = n2->c\_next;

}

}

rr = i;

cc = j;

result.add\_node(vv, rr, cc);

}

}

}

}

return result;

}

void my\_matrix::print() {

cout << endl;

for (int i = 0; i < num\_rows; i++) {

node \* p = r\_vec[i];

cout << endl;

while (p != nullptr) {

cout << p->value << " " << p->r\_position << " " << p->c\_position << " ";

p = p->r\_next;

}

}

}

int main() {

my\_matrix M1(7, 5, 11, 8), M2(7, 5, 10, 8), M3(7, 5), M4(5, 6, 13, 9), M5(7, 6);

M1.print();

M2.print();

M3 = M1 + M2;

M3.print();

M1.print();

M4.print();

M5 = M1 \* M4;

M5.print();

getchar();

getchar();

return 0;

}