Name: Dauren Abdykaparov

Student ID: 20142001

**Data Structures.**

**Assignment #3. Sparse matrix.**

1) **Matrix(int Row, int col)**is an empty matrix constructor. It creates {max (Row, col) + 1} head matrix nodes, one main and one per row-column pair. Main head node and regular head nodes are connected via `next` pointer in the following way:

*Main – head[0] – head[i] – head[max (Row, col) – 1] – Main*

Each row-column head node has `right` and `down` pointers linked to the node itself.

2) **Matrix(std::ifstream& is)** is a file matrix constructor. First, it reads a triple of integers: s.row (number of rows), s.col (number of columns) and s.value (number of elements); saves them into the object we work with, and prints to the screen. Then, {max (Row, col) + 2} head matrix nodes are created: one main, `head[0..{max (s.row, s.col) - 1}]` nodes for each row-column pair and `last` pointer to the 0th row-column head node. The number of the row-column pair to which `last` links is stored in `row` variable. After that, for each {s.value} triples, which are expected to be given in row-major order, we do the following operations:

I) read them into `t` variable.

II) if current row is different from previous one, we link `last` to `head[row]` via `next` pointer, change `row` and `last` to the current row.

III) make a link from `last` to a new node storing `t` via `right` pointer and change `last` to the new node.

IV) `head[t.col]->next` is used to store pointer to the last element in the column, therefore we link it to `last` via `down` pointer and change `head[t.col]->next` to `last`.

After these operation, we link `last->right` to `head[row` in order to close the linked cycle. Then, we close each column linked cycle and connect main head node with row-column head nodes as in empty matrix constructor.

3) **Matrix(const Matrix& a)** is a copy matrix constructor, it is the same as a file matrix constructor, but instead of file we read from the matrix.

4) **operator=(const Matrix& a)** is and assignment operator, it is the same as copy matrix constructor, but we also return the matrix.

5) **~Matrix()** is a straightforward destructor. Just delete the main head node.

6) **operator+(const Matrix& b)** is a matrix addition operator. First, we check if dimensions are the same and alert an error if they aren’t. Then we create a new matrix, where the result will be stored, and two pointers `A` and `B` to get round both matrices in row-major order. Until the both pointers reach the end of the input matrices, we do the following operations:

I) if `A` < `B`, add `A`’s triple to the result matrix and increment `A` (We say that pointer `A` is less than pointer `B` from the matrices of the same dimensions, if `A`’s triple has smaller indexes in row-major order).

II) if `B` < `A`, add `B`’s triple to the result matrix and increment `B`.

III) if `A` == `B`, add sum of the triples to the result matrix and increment both pointers.

7) **operator-(const Matrix& b)** is a matrix subtraction operator. Only difference from addition operator is that we treat the second input matrix as it has all its elements multiplied by -1.

8) **operator\*(const Matrix& b)** is a matrix multiplication operator. First, we check if dimensions are the same and alert an error if they aren’t. Then we create a new matrix, where the result will be stored, two arrays of the pointers: `row` for rows of the first input matrix and `col` for columns of the second; and two pointer `C` and `R`. For each element of the result matrix with indexes `i` and `j` we do the following operations:

I) Set `R` to the beginning of `i-th` row of the first matrix and `C` to the beginning of `j-th` column of the second matrix.

II) Create `res` = 0, where the value of the element will be stored.

III) `x` is a triple of `R` and `y` is a triple of `C`.

IV) increment `R` while `x.col` < `y.row` or `C` while `x.col` > `y.row`. End if one of the pointers reached the end.

V) now `x.col` and `y.col` are the same. We add `x.value` multiplied by `y.value` to `res`, increment both `R` and `C` and proceed to the step IV.

If `res` is not zero, we create put the triple (i, j, res) to the result matrix.

9) **SwapRow(int i, int j)** is a function which swaps two rows of the matrix.

My implementation is not the fastest one (O(n^3) compared to O(n^2)), but the shortest. It was done in order to shorten the code which was already about 500 lines and decrease the risk of making a mistake.

First, we check if `i` and `j` are in the range. Then, we create the result matrix and get round the input matrix in row-major order. Let the current triple be `t`, `x` be `t.row`. We do the following operations:

I) if `t.row` == `i`, set `x` as `j`.

II) if `t.row` == `j`, set `x` as `i`.

III) create a triple (x, y, t.value) in the result matrix.

10) **SwapCol(int i, int j)** is a function which swaps two columns of the matrix. Almost absolutely the same as SwapRow() function.

11) **CreateEl**

**ement(int i, int j, type val)** is a function which adds a triple to the matrix if it doesn’t exist and replaces otherwise. First we check if `i` and `j` are in the range. Then increment `nElem`, create `row` which links to the `i-th` row, `col` which links to the `j-th` column, `A` which links to the beginning of `row` and `B` which links to the beginning of `col`. Using the cycle we increment `A` and `B` until their triples have smaller indexes than `i` and `j`. It is the place where the element should be inserted. If the pointer next to `A` has the triple with the same indexes as `i` and `j`, we replace the triple with the one from the input. Otherwise, we insert the new element and return the matrix.

12) **Transpose()** is a functions which swaps columns and rows. We create the result matrix, get round the current matrix and add all elements to the result matrix preliminarily interchanging the indexes. Then, set the current matrix to the result one and return it.