Sparse Matrix Program

2019320139 Choi Yun Ji

1. Development environment

OS : Window 10

IDE : Visual Studio 2017

2. Explanation of the algorithm and the code

1) Menu Driven

For menu driven program, whenever input is given, the function is called accordingly.

When going back to menu or executing an operation, clear the console using *system("cls")*. In operation 1 and 3~6, go back to menu right after operation.

On the contrary, in operation 2, an user should determine when to go back to menu by pressing enter because time to read result is needed.

Only when seven is given as input, the program terminates.

Otherwise, the program repeats over and over again.

```c
int main()
{
    int stop = 0;

    while (1) //repeat until input is '7'(Exit)
    {
        int input;

        system("cls"); //clear console
        printf("1. Read\n2. Write\n3. Erase\n4. Add\n5. Multiply\n6. Transpose\n7. Exit\n");
        scanf("%d", &input);

        switch (input)
        {
        case 1:
            system("cls");
            mread();
            break;
        case 2:
            system("cls");
            mwrite();
            break;
        case 3:
            system("cls");
            merase();
            break;
        case 4:
            system("cls");
            madd();
            break;
        case 5:
            system("cls");
            mmult();
            break;
        case 6:
            system("cls");
            mtranspose();
            break;
        case 7:
            stop = 1;
            break;
        }

        if (stop) //if input 7 is given, come out of while loop
            break;
    }

    printf("\nExit the program\n");

    return;
}
```

2) Saving Matrices

```
matrixPointer nodes[MAX_SIZE];
```

Global array *nodes* is the only storage location. Each *H* node(top left node) of matrices is stored in it. In operations, if save index is already occupied, free that and save a new matrix in it.

3) *Madd* function

The operation of *madd* is similar to adding polynomials. In the same row, if column of tempA is smaller than that of tempB(B[row][tempA's column] is ZERO), insert a node of tempA into H[row] of D. Else if column of tempB is smaller than that of tempA (A[row][tempB's column] is ZERO), insert a node of tempB into H[row] of D. If the column is same, insert a node with value of (tempA's value + tempB's value).

If either A or B reaches the end of the row, insert all the left nodes of the other that doesn’t reach the end.

Repeat this as many times as number of rows.

```c
void madd()
{
    int first, second, save, numHeads, row, col, i;
    matrixPointer a, b, d, hdA, hdB, tempA, tempB, temp, last;
    printf("First, second, save index: ");
    scanf("%d%d%d", &first, &second, &save);
    a = nodes[first], b = nodes[second];

    if (a->u.entry.row != b->u.entry.row || a->u.entry.col != b-
>u.entry.col) //if it impossible to add two matrices
    {
        printf("Error! Cannot add two matrices\n");
        return;
    }

    matrixPointer hdnode[MAX_SIZE];

    row = a->u.entry.row;
    col = a->u.entry.col;
    numHeads = (row > col) ? row : col;
    d = newNode(); //H node of a new matrix
    d->tag = entry; d->u.entry.row = row; d->u.entry.col = col;

    if (!numHeads)//empty matrix
        d->right = d;
    else {
        for (i = 0; i < numHeads; i++) //creat headnodes
        {
            temp = newNode();
            hdnode[i] = temp; hdnode[i]->tag = head;
            hdnode[i]->right = temp; hdnode[i]->u.next = temp;
        }
        hdA = a->right;
        hdB = b->right;
        for (i = 0; i < row; i++) {
            tempA = hdA->right; //move horizontally
            tempB = hdB->right;
            last = hdnode[i]; //current last node of row i
            while (tempA != hdA && tempB != hdB)
            {
                if (tempA->u.entry.col < tempB->u.entry.col)
                {
                    //insert A's element
                    temp = insertNode(i, tempA->u.entry.col, tempA->u.entry.value, &last);
                    hdnode[temp->u.entry.col]->u.next->down = temp;
                    hdnode[temp->u.entry.col]->u.next = temp;
                    tempA = tempA->right;
                    //move tempA to right index
                }
                else if (tempA->u.entry.col > tempB->u.entry.col)
                {
                    //insert B's element
                    temp = insertNode(i, tempB->u.entry.col, tempB->u.entry.value, &last);
                    hdnode[temp->u.entry.col]->u.next->down = temp;
                    hdnode[temp->u.entry.col]->u.next = temp;
                    tempB = tempB->right;
                    //move tempB to right index
                }
                else //if the same locations of A and B both are not 0
                {
                    //insert a element with value of (A's element + B's element)
                    temp = insertNode(i, tempA->u.entry.col, tempA->u.entry.value + tempB-
>u.entry.value, &last);
                    hdnode[temp->u.entry.col]->u.next->down = temp;
                    hdnode[temp->u.entry.col]->u.next = temp;
                    tempA = tempA->right;
                    tempB = tempB->right;
                }
            }
            for (; tempA != hdA; tempA = tempA->right) //if there are some leftovers in A
            {
                //insert all of the left overs
                temp = insertNode(i, tempA->u.entry.col, tempA->u.entry.value, &last);
                hdnode[temp->u.entry.col]->u.next->down = temp;
                hdnode[temp->u.entry.col]->u.next = temp;
            }
            for (; tempB != hdB; tempB = tempB->right) //if there are some leftovers in B
            {
                //insert all of the left overs
                temp = insertNode(i, tempB->u.entry.col, tempB->u.entry.value, &last);
                hdnode[temp->u.entry.col]->u.next->down = temp;
                hdnode[temp->u.entry.col]->u.next = temp;
            }
            last->right = hdnode[i];//make a circular list
            hdA = hdA->u.next;
            hdB = hdB->u.next;
            //move to next row
        }

        for (i = 0; i < col; i++)
            hdnode[i]->u.next->down = hdnode[i];
        for (i = 0; i < numHeads - 1; i++)
            hdnode[i]->u.next = hdnode[i + 1];
        hdnode[numHeads - 1]->u.next = d;
        d->right = hdnode[0];
    }
    if (nodes[save] != NULL) //if nodes[save] is already occupied
        free(nodes[save]);
    nodes[save] = d;
}
```

4) *Mmultiply* function

For multiplying, A is searched horizontally and B is searched vertically.

Because $\mathrm{D[i][j]} = \sum A[i][k] * B[k][j]$.

Moving horizontally in *H[i]* of A and moving vertically in *H[j]* of B, if column of tempA and row of tempB is same, multiply values of two temp and add it to sum.

If either A or B reaches the end, headnode of B moves for seeking D[i][j+1].

Once this process is done about row i, then move on the next row i+1.

Start again from (*H[i+1]* of A and *H[0]* of B) for calculating D[i+1][0].

Repeat this as many times as number of rows.

```
void mmult()
{
    int first, second, save, numHeads, row, col, sum, i, j;
    matrixPointer a, b, d, temp, tempA, tempB, headA, headB, last;
    matrixPointer hdnode[MAX_SIZE];
    printf("First, second, save index: ");
    scanf("%d%d%d", &first, &second, &save);
    a = nodes[first], b = nodes[second];

    if (a->u.entry.col != b->u.entry.row) //if it is impossible to multiply two matrices
    {
        printf("Error! Cannot multiply two matrices\n");
    }
    row = a->u.entry.row;
    col = b->u.entry.col;
    numHeads = (row > col) ? row : col;
    d = newNode(); //H node of a new matrix
    d->tag = entry; d->u.entry.row = row; d->u.entry.col = col;

    if (!numHeads)
        d->right = d;

    for (i = 0; i < numHeads; i++) //create headnodes
    {
        temp = newNode();
        hdnode[i] = temp; hdnode[i]->tag = head;
        hdnode[i]->right = temp; hdnode[i]->u.next = temp;
    }

    headA = a->right;
    for (i = 0; i < row; i++)
    {
        last = hdnode[i]; //current last node of row i
        headB = b->right; //start from H[0] of B
        for (j = 0; j < col; j++)
        {
            tempA = headA->right; //tempA moves in a row direction
            tempB = headB->down; //tempB moves in a column direction
            sum = 0; //sum of A[i][k]*B[k][j]
            while(tempA != headA && tempB != headB)
            {

                if (tempA->u.entry.col < tempB->u.entry.row)
                    tempA = tempA->right;
                else if (tempA->u.entry.col > tempB->u.entry.row)
                    tempB = tempB->down;
                else //if two elements that can be multiply are found
                {
                    //add A[i][k]*B[k][j] to sum for D[i][j]
                    sum += (tempA->u.entry.value) * (tempB->u.entry.value);
                    tempA = tempA->right;
                    tempB = tempB->down;
                }
            }
            if (sum != 0)
            {
                //insert a node of D[i][j]
                temp = insertNode(i, j, sum, &last);
                hdnode[j]->u.next->down = temp;
                hdnode[j]->u.next = temp;
            }

            headB = headB->u.next; //move for calculating D[i][j+1]
        }
        last->right = hdnode[i]; //make a circular list
        headA = headA->u.next;
    }
    for (i = 0; i < col; i++)
        hdnode[i]->u.next->down = hdnode[i];
    for (i = 0; i < numHeads - 1; i++)
        hdnode[i]->u.next = hdnode[i + 1];
    hdnode[numHeads - 1]->u.next = d;
    d->right = hdnode[0];

    if (nodes[save] != NULL) //if nodes[save] is already occupied
        free(nodes[save]);
    nodes[save] = d;
}
```

5) *Mtranspose* function

For transposing, move vertically in original matrix.

That means, insert all the nodes in column **i** of original matrix into row **i** of new matrix.

If column **i** reaches the end, move on the next. (column **i+1** of original one and row **i+1** of new one)

Repeat this as many times as number of rows of new matrix.

```c
void mtranspose()
{
    int trans, save, row, col, numHeads, i;
    matrixPointer a, b, temp, tempA, headA, last;
    matrixPointer hdnode[MAX_SIZE];
    printf("Transpose index, save index: ");
    scanf("%d%d", &trans, &save);
    a = nodes[trans];
    headA = a->right;
    row = a->u.entry.col;
    col = a->u.entry.row;
    numHeads = (row > col) ? row : col;
    b = newNode();
    b->tag = entry; b->u.entry.row = row; b->u.entry.col = col;

    for (i = 0; i < numHeads; i++) //create headnodes
    {
        temp = newNode();
        hdnode[i] = temp; hdnode[i]->tag = head;
        hdnode[i]->right = temp; hdnode[i]->u.next = temp;
    }

    for (i = 0; i < row; i++)
    {
        last = hdnode[i];
        for (tempA = headA->down; tempA != headA; tempA = tempA->down)
           //Until visit all the elements of H[i] of A vertically
        {
            //insert elements horizontally into H[i] of new matrix
            temp = insertNode(i, tempA->u.entry.row, tempA->u.entry.value, &last);
             //new row is i, and new column is row of original node
            hdnode[tempA->u.entry.row]->u.next->down = temp;
            hdnode[tempA->u.entry.row]->u.next = temp;
        }
        last->right = hdnode[i]; //make a circular list
        headA = headA->u.next; //move to next headnode
    }
    for (i = 0; i < col; i++)
    {
        hdnode[i]->u.next->down = hdnode[i];
    }
    for (i = 0; i < numHeads - 1; i++)
        hdnode[i]->u.next = hdnode[i + 1];
    hdnode[numHeads - 1]->u.next = b;
    b->right = hdnode[0];

    if (nodes[save] != NULL)
        free(nodes[save]);
    nodes[save] = b;
}
```

6) Modified codes

In mread function, the process of obtaining input(save index) is added. And I didn’t use global array hdnode, instead created local array hdnode and make the node(top left node of matrix) point to hdnode array. At the end, checking whether save index is already occupied is occurred. If it is already occupied, it would be free. Finally, save the new matrix into nodes[save]. (nodes is global array)

In mwrite function, likewise, the process of obtain input(print index) is added. If the index is empty, an error message would be printed. Else, bring in the print index from global array ‘nodes’. After printing output, users should press enter to go back to menu.

At the beginning of merase function, index to erase is given. After erasing, store NULL in nodes[index].