# CS 137: Assignment #10

Due on Tuesday, Dec 3, 2024, at 11:59 PM

There is no penalty for submitting up to 48 hours late (until Thursday, Dec 5, 2024, at 11:59 PM).

Submit all programs using the Marmoset Submission and Testing Server located at
https://marmoset.student.cs.uwaterloo.ca/

Victoria Sakhnini

Fall 2024

## Notes:

- Use the examples to guide the formatting of your output. Remember to terminate your output with a newline character.
- For this assignment, you may use any content covered until the end of Module 13.
- <math.h> is not allowed.
- Your solution may not pass marmoset tests for some of the tests due to the wrong implementation or a memory leak. My advice is that even if you pass your own tests and are very positive about your implementation. Use Valgrind to detect memory leaks before submitting your solutions. gcc won't detect memory leaks, and maybe your local compiler won't catch them, either.

## Problem 1

You are tasked with creating a memory-efficient implementation for storing and processing a sparse matrix using a linked list. The matrix has a large number of rows and columns, but only a small percentage of the entries are nonzero.

Complete the provided program `listmatrix.c`, which includes a definition of a matrix as a linked list and the documentation. The program also includes a function to print the matrix and a main function with sample test ( Make sure you add your own tests to ensure your solution's correctness).

You are to complete the following functions:

1) `void insert_entry(struct SparseMatrix* result, int row, int col, int value);`
   The Function adds a new entry to the `SpareMtrix` at the front. Assume all arguments are valid.
2) `struct SparseMatrix multiply_matrices(const struct SparseMatrix* matrix1, const struct SparseMatrix* matrix2);`
   The Function takes two `SparseMatrix` and returns the result of multiplying those two matrices.
3) `void free_sparse_matrix(struct SparseMatrix* matrix);`
   The function frees the memory allocated on the heap pointed at by the head of the `matrix`.

You are to submit `listmatrix.c` file containing all implemented functions (you must delete the test cases portion/`main` function). However, **you must keep the required included libraries and structure definitions.**

**Note: The order of multiplication or the output (order of printed lines) doesn't matter as long as it represents the results correctly.**

Here is a sample test and the expected output:

```c
#include <stdio.h>
#include <stdlib.h>
 // A structure for representing a single entry in the matrix. THis part is stored in the heap
struct MatrixEntry {
    int row;  // the row number of a specific entry
    int col;  // the col number of a specific entry
    int value;
    struct MatrixEntry* next; // Pointer to the next entry in the linked list
};
 // A structure for representing the sparse matrix. This part is stored in the stack
struct SparseMatrix {
    int num_rows;  // the dimension of the matrix is
    int num_cols;  //  num_rows X num_cols
    struct MatrixEntry* head; // Pointer to the first entry of the linked list
};

// Define a function to insert a new entry into the front of the result matrix
void insert_entry(struct SparseMatrix* result, int row, int col, int value) { }

// Define a function that multiplies two sparse matrices with valid sizes
struct SparseMatrix multiply_matrices(const struct SparseMatrix* matrix1, const struct SparseMatrix* matrix2) {
}

 // Define a function to free memory allocated for a sparse matrix
void free_sparse_matrix(struct SparseMatrix* matrix) { }

// Function to print a sparse matrix
// Do not change this Function
void print_sparse_matrix(const struct SparseMatrix* matrix) {
    for (struct MatrixEntry* entry = matrix->head; entry != NULL; entry = entry->next) {
        printf("(%d, %d): %d\n", entry->row, entry->col, entry->value);
    }
}

// for testing. Do not submit the main Function
int main(void) {
    // Create and initialize sparse matrices (matrix1 and matrix2)
    struct SparseMatrix matrix1;
    matrix1.num_rows = 3;
    matrix1.num_cols = 4;
    matrix1.head = NULL;
    // Initialize matrix1 with some entries
    insert_entry(&matrix1, 0, 0, 1);
    insert_entry(&matrix1, 1, 1, 2);
    insert_entry(&matrix1, 2, 2, 3);
    insert_entry(&matrix1, 0, 3, 5);
    printf("Matrix1 3x4:\n");
    print_sparse_matrix(&matrix1);

    struct SparseMatrix matrix2;
    matrix2.num_rows = 4;
    matrix2.num_cols = 3;
    matrix2.head = NULL;

    // Initialize matrix2 with some entries
    insert_entry(&matrix2, 0, 0, 4);
    insert_entry(&matrix2, 1, 1, 5);
    insert_entry(&matrix2, 2, 2, 6);
    insert_entry(&matrix2, 3, 2, 3);
    printf("Matrix2 4x3:\n");
    print_sparse_matrix(&matrix2);
    // Multiply matrices
    struct SparseMatrix result = multiply_matrices(&matrix1, &matrix2);

    // Print the result matrix
    printf("Result matrix1 x matrix2:\n");
    print_sparse_matrix(&result);

    // Free memory allocated for matrices and result
    free_sparse_matrix(&matrix1);
    free_sparse_matrix(&matrix2);
    free_sparse_matrix(&result);

    return 0;
}

```

The expected output (**the order of the output of the matrix itself does not matter**):

```
Matrix1 3x4:
(0, 3): 5
(2, 2): 3
(1, 1): 2
(0, 0): 1
Matrix2 4x3:
(3, 2): 3
(2, 2): 6
(1, 1): 5
(0, 0): 4
Result matrix1 x matrix2:
(0, 0): 4
(1, 1): 10
(2, 2): 18
(0, 2): 15
```

## Problem 2

Duck-Duck-Goose is a game in which a group of people sit in a circle while one person walks around the circle at a time.

- Let's call the person walking around the circle `P` As `P` passes people sitting down, they will tap on the sitting person's head and say either "duck" or "goose".
- If `P` says "duck", nothing will happen, and they will continue walking.
- If `P` says goose, the sitting person will get up and chase `P` around the circle.
- If `P` is touched before running around the circle once, they must stay as `P`.
- Otherwise, the previously sitting person becomes `P`.
- In this alternative of the game, the loser will instead be eliminated from playing, and the winner will become `P`.

For this question, you will simulate a game of duck-duck-goose.

The struct to represent a person is as follows:

```
typedef struct Person {
    int id;
    int speed;
    struct Person* next;
}Person;
```

Create a program that takes in as input lines...

```
    person1 speed1
    person2 speed2
    .
    .
    .

    personn speedn
```

where `personn` is a unique integer ID to represent the nth person and `speedn` is a positive integer to represent the speed of the nth person. Construct a link list of people in the order they are given that forms a ring such that the last person points to the first person in the list.

`P` will start off as the first person to read in from standard input. After reading in all participants, the program should do the following:

`I)` `P` will call `n` people “duck” and the `n + 1`th person goose where `n` is the `id` of `P`. As `P` goes from person to person, print either

```
c duck d
```

or

```
c goose! d
```

Where `c` is the `id` of `P` and where `d` is the `id` of the sitting person `P` passes.

`II)` When `P` calls goose, the winner is the person with the highest speed value.

If the speeds are the same, the winner is the “goosed” person. That person will become the new `P,` and the loser will leave the circle. Continue playing duck-duck-goose, moving on to the person after the winner until one person is left.

`III)` At the end, print

```
winner! d
```

where `d` is the `id` of the winner.

**Example Input1**

```
1 2

2 4

3 3

4 1

?
```

**Example Output1**

```
1 duck 2

1 goose! 3

3 duck 4

3 duck 2

3 duck 4

3 goose! 2

2 duck 4

2 duck 4

2 goose! 4

winner! 2
```

**Example Input2**

```
1 2

2 3

3 4

4 5

5 6

6 7

?
```

**Example Output2**

```
1 duck 2

1 goose! 3

3 duck 4

3 duck 5

3 duck 6

3 goose! 2

3 duck 4

3 duck 5

3 duck 6

3 goose! 4

4 duck 5

4 duck 6

4 duck 5

4 duck 6

4 goose! 5

5 duck 6

5 duck 6

5 duck 6

5 duck 6

5 duck 6

5 goose! 6

winner! 6
```

**Example Intput3**

```
3 6

1 5

2 7

4 4

6 1

5 3

?
```

**Example Output3**

```
3 duck 1

3 duck 2

3 duck 4

3 goose! 6

3 duck 1

3 duck 2

3 duck 4

3 goose! 5

3 duck 1

3 duck 2

3 duck 4

3 goose! 1

3 duck 2

3 duck 4

3 duck 2

3 goose! 4

3 duck 2

3 duck 2

3 duck 2

3 goose! 2

winner! 2
```

**Example Intput4**

```
1 5

2 7

3 5

4 1

?
```

**Example Output4**

```
1 duck 2

1 goose! 3

3 duck 4

3 duck 2

3 duck 4

3 goose! 2

2 duck 4

2 duck 4

2 goose! 4

winner! 2
```

Create a C program `duckgoose.c` that completes the following functions :

```
Person updatePerson(int id, int speed);

void addPerson(Person** start, Person* new);
```

`Person* play(Person* start)//` frees any person who leaves the game.

You are to submit this file containing all implemented functions (that is, you must delete the test cases portion/`main` function). However, **you must keep the required included libraries and structure definitions.**

The provided main function to play the game:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct Person{
    int id;
    int speed;
    struct Person* next;
}Person;

```

```c
Person updatePerson(int id, int speed) {
}

void addPerson(Person** start, Person* new) {
}
// Returns the winner
Person* play(Person* start) {
}

int main(){
    int amount;
    int p, s;
    // List of people
    Person* lop = NULL;
    // Read in participants
    while (scanf("%d %d", &p, &s) == 2){
        Person* np = (Person*)malloc(sizeof(Person));
        *np = updatePerson(p, s);
        addPerson(&lop, np);
    }
   Person* winner = play(lop);
    free(winner);
}

```