# Lab Assessment 3 (15%)

**This is individual take home coursework, to be solved remotely at your own pace. It is open book, open internet, but avoid using AI Bots.**

**There are 3 questions in this sheet.**

**Submission: Dec 13 (Fri) 11:59pm**

## 1.0 Problems

### Question 1 (5 marks)

Create a C program that performs basic operations on two matrices.

The program should allow the user to input

- the dimensions (number of rows and columns) of two matrices A and B (maximum size: 10x10).
- the elements of both matrices.

Provide a menu for the user to choose an operation:

1. Add Matrices
2. Subtract Matrices
3. Transpose Matrix A
4. Transpose Matrix B
5. Exit (Loop back to the menu until the user chooses to exit.)

Display the results of the selected operation. Apply what you learn about program *control structures* (`if`, `for` and `while` loops where appropriate), *functions*, *arrays*, and/or *pointers* (to hold the values of the matrices). As much as possible, make sure you implement your code in a modular way (with **functions**). Declare and define the necessary functions.

A sample program output is shown as follows (make sure the output logic is as close as possible to the sample). Perform necessary validation to make sure only valid matrices dimensions and values are entered.

```
Enter dimensions for Matrix A (rows and columns): 2 2
Enter elements for Matrix A:
Element [1][1]: 1
Element [1][2]: 2
Element [2][1]: 3
Element [2][2]: 4
Enter dimensions for Matrix B (rows and columns): 2 2
Enter elements for Matrix B:
Element [1][1]: 4
Element [1][2]: 3
Element [2][1]: 2
Element [2][2]: 1
```

```
Matrix Operations Menu:
1. Add Matrices
2. Subtract Matrices
3. Transpose Matrix A
4. Transpose Matrix B
5. Exit
Enter your choice: 1
Result of Matrix A + Matrix B:
5 5
5 5
```

```
Matrix Operations Menu:
1. Add Matrices
2. Subtract Matrices
3. Transpose Matrix A
4. Transpose Matrix B
5. Exit
Enter your choice: 2
Result of Matrix A - Matrix B:
-3 -1
1 3

Matrix Operations Menu:
1. Add Matrices
2. Subtract Matrices
3. Transpose Matrix A
4. Transpose Matrix B
5. Exit
Enter your choice: 3
Transpose of Matrix A:
1 3
2 4
```

```
Matrix Operations Menu:
1. Add Matrices
2. Subtract Matrices
3. Transpose Matrix A
4. Transpose Matrix B
5. Exit
Enter your choice: 4
Transpose of Matrix B:
4 2
3 1

Matrix Operations Menu:
1. Add Matrices
2. Subtract Matrices
3. Transpose Matrix A
4. Transpose Matrix B
5. Exit
Enter your choice: 5
Exiting...
```

**Note**: Write comments to explain your code as necessary.

Marking Scheme:

- Student will not be able to get more than half of the marks (2.5/5) if the program compile with syntax error.
- Correct implementation for **input** / **output** functions (1 mark)
- Correct implementation for **add matrices** function (1 mark)
- Correct implementation for **subtract matrices** function (1 mark)
- Correct implementation for **transpose matrices** function (1 mark)
- Correct implementation for **print matrices** function (1 mark)

**Sample Answer 1 (using two dimensional arrays)**

```
#include <stdio.h>

#define MAX_SIZE 10

void inputMatrix(int matrix[][MAX_SIZE], int rows, int cols);
void addMatrices(int a[][MAX_SIZE], int b[][MAX_SIZE], int
result[][MAX_SIZE], int rows, int cols);
void subtractMatrices(int a[][MAX_SIZE], int b[][MAX_SIZE], int
result[][MAX_SIZE], int rows, int cols);
void transposeMatrix(int matrix[][MAX_SIZE], int transposed[][MAX_SIZE],
int rows, int cols);
```

```
void printMatrix(int matrix[][MAX_SIZE], int rows, int cols);

int main() {
    int a[MAX_SIZE][MAX_SIZE], b[MAX_SIZE][MAX_SIZE],
result[MAX_SIZE][MAX_SIZE];
    int rowsA, colsA, rowsB, colsB;
    int choice;

    // Input logic and inputMatrix function (0.5 mark)
    // Input first matrix
    printf("Enter dimensions for Matrix A (rows and columns): ");
    scanf("%d %d", &rowsA, &colsA);
    printf("Enter elements for Matrix A:\n");
    inputMatrix(a, rowsA, colsA);

    // Input second matrix
    printf("Enter dimensions for Matrix B (rows and columns): ");
    scanf("%d %d", &rowsB, &colsB);
    printf("Enter elements for Matrix B:\n");
    inputMatrix(b, rowsB, colsB);

    // Menu logic do…while/while loop and menu (1 mark)
    do {
        printf("\nMatrix Operations Menu:\n");
        printf("1. Add Matrices\n");
        printf("2. Subtract Matrices\n");
        printf("3. Transpose Matrix A\n");
        printf("4. Transpose Matrix B\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                if (rowsA == rowsB && colsA == colsB) {
                    addMatrices(a, b, result, rowsA, colsA);
                    printf("Result of Matrix A + Matrix B:\n");
                    printMatrix(result, rowsA, colsA);
                } else {
                    printf("Matrices must have the same dimensions for
addition.\n");
                }
                break;
            case 2:
                if (rowsA == rowsB && colsA == colsB) {
```

```
                    subtractMatrices(a, b, result, rowsA, colsA);
                    printf("Result of Matrix A - Matrix B:\n");
                    printMatrix(result, rowsA, colsA);
                } else {
                    printf("Matrices must have the same dimensions for
subtraction.\n");
                }
                break;
            case 3: {
                int transposed[MAX_SIZE][MAX_SIZE];
                transposeMatrix(a, transposed, rowsA, colsA);
                printf("Transpose of Matrix A:\n");
                printMatrix(transposed, colsA, rowsA);
                break;
            }
            case 4: {
                int transposed[MAX_SIZE][MAX_SIZE];
                transposeMatrix(b, transposed, rowsB, colsB);
                printf("Transpose of Matrix B:\n");
                printMatrix(transposed, colsB, rowsB);
                break;
            }
            case 5:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice! Please try again.\n");
        }
    } while (choice != 5);

    return 0;
}

void inputMatrix(int matrix[][MAX_SIZE], int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("Element [%d][%d]: ", i + 1, j + 1);
            scanf("%d", &matrix[i][j]);
        }
    }
}

// addMatrices function logic (0.5 mark)
void addMatrices(int a[][MAX_SIZE], int b[][MAX_SIZE], int
result[][MAX_SIZE], int rows, int cols) {
```

```
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            result[i][j] = a[i][j] + b[i][j];
        }
    }
}

// subtractMatrices function logic (0.5 mark)
void subtractMatrices(int a[][MAX_SIZE], int b[][MAX_SIZE], int
result[][MAX_SIZE], int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            result[i][j] = a[i][j] - b[i][j];
        }
    }
}

// transposeMatrix function logic (1 mark)
void transposeMatrix(int matrix[][MAX_SIZE], int transposed[][MAX_SIZE],
int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            transposed[j][i] = matrix[i][j];
        }
    }
}

// printMatrix function logic (1 mark)
void printMatrix(int matrix[][MAX_SIZE], int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}
```

**Sample Answer 2 using pointers)**

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

void inputMatrix(int **matrix, int rows, int cols);
void addMatrices(int **a, int **b, int **result, int rows, int cols);
void subtractMatrices(int **a, int **b, int **result, int rows, int
cols);
void transposeMatrix(int **matrix, int **transposed, int rows, int
```

```
cols);
void printMatrix(int **matrix, int rows, int cols);

int main() {
    int rowsA, colsA, rowsB, colsB;
    int choice;

    // Input dimensions for Matrix A
    printf("Enter dimensions for Matrix A (rows and columns): ");
    scanf("%d %d", &rowsA, &colsA);

    // Allocate memory for Matrix A
    int **a = (int **)malloc(rowsA * sizeof(int *));
    for (int i = 0; i < rowsA; i++) {
        a[i] = (int *)malloc(colsA * sizeof(int));
    }

    printf("Enter elements for Matrix A:\n");
    inputMatrix(a, rowsA, colsA);

    // Input dimensions for Matrix B
    printf("Enter dimensions for Matrix B (rows and columns): ");
    scanf("%d %d", &rowsB, &colsB);

    // Allocate memory for Matrix B
    int **b = (int **)malloc(rowsB * sizeof(int *));
    for (int i = 0; i < rowsB; i++) {
        b[i] = (int *)malloc(colsB * sizeof(int));
    }

    printf("Enter elements for Matrix B:\n");
    inputMatrix(b, rowsB, colsB);

    // Allocate memory for result matrices
    int **result = (int **)malloc(rowsA * sizeof(int *));
    for (int i = 0; i < rowsA; i++) {
        result[i] = (int *)malloc(colsA * sizeof(int));
    }
    int **transposed = (int **)malloc(colsA * sizeof(int *));
    for (int i = 0; i < colsA; i++) {
        transposed[i] = (int *)malloc(rowsA * sizeof(int));
    }

    do {
        printf("\nMatrix Operations Menu:\n");
```

```
        printf("1. Add Matrices\n");
        printf("2. Subtract Matrices\n");
        printf("3. Transpose Matrix A\n");
        printf("4. Transpose Matrix B\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                if (rowsA == rowsB && colsA == colsB) {
                    addMatrices(a, b, result, rowsA, colsA);
                    printf("Result of Matrix A + Matrix B:\n");
                    printMatrix(result, rowsA, colsA);
                } else {
                    printf("Matrices must have the same dimensions for
addition.\n");
                }
                break;
            case 2:
                if (rowsA == rowsB && colsA == colsB) {
                    subtractMatrices(a, b, result, rowsA, colsA);
                    printf("Result of Matrix A - Matrix B:\n");
                    printMatrix(result, rowsA, colsA);
                } else {
                    printf("Matrices must have the same dimensions for
subtraction.\n");
                }
                break;
            case 3:
                transposeMatrix(a, transposed, rowsA, colsA);
                printf("Transpose of Matrix A:\n");
                printMatrix(transposed, colsA, rowsA);
                break;
            case 4:
                transposeMatrix(b, transposed, rowsB, colsB);
                printf("Transpose of Matrix B:\n");
                printMatrix(transposed, colsB, rowsB);
                break;
            case 5:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice! Please try again.\n");
        }
```

```
    } while (choice != 5);

    // Free allocated memory
    for (int i = 0; i < rowsA; i++) free(a[i]);
    free(a);
    for (int i = 0; i < rowsB; i++) free(b[i]);
    free(b);
    for (int i = 0; i < rowsA; i++) free(result[i]);
    free(result);
    for (int i = 0; i < colsA; i++) free(transposed[i]);
    free(transposed);

    return 0;
}

void inputMatrix(int **matrix, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("Element [%d][%d]: ", i + 1, j + 1);
            scanf("%d", &matrix[i][j]);
        }
    }
}

void addMatrices(int **a, int **b, int **result, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            result[i][j] = a[i][j] + b[i][j];
        }
    }
}

void subtractMatrices(int **a, int **b, int **result, int rows, int
cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            result[i][j] = a[i][j] - b[i][j];
        }
    }
}

void transposeMatrix(int **matrix, int **transposed, int rows, int cols)
{
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
```

```
            transposed[j][i] = matrix[i][j];
        }
    }
}

void printMatrix(int **matrix, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}
```

## Question 2 (5 marks)

Write a C program to manage grocery item information and item availability. You need to define using `struct` and/or `union` for the following information:

**Grocery Item:**
- Name (string, max length 100)
- Category (string, max length 50)
- Quantity (integer)
- Price (float)

**Availability Status:**
- In Stock (boolean)
- Out of Stock (boolean)

**Grocery Inventory:**
- An array of *Grocery Items* (max size 100)
- An array of *Availability Status* (using the Availability Status defined above: in stock/out of stock)
- A count of the total number of grocery items in the inventory

Implement functions to
1. **Add** grocery item
2. **List** all grocery items with their availability
3. **Update** the quantity of a grocery item
4. **Remove** a grocery item from the inventory

The program menu runs in a loop, allowing users to select different actions until they choose to exit. Include any necessary library header files.

A sample program output is shown as follows (make sure the output logic is as close as possible to the sample):

```
Grocery Inventory Management System
1. Add Grocery Item
2. List All Grocery Items
3. Update Quantity
4. Remove Grocery Item
5. Exit
Enter your choice: 1
Enter Item Name: 100 Plus
Enter Category: Soft drink
Enter Quantity: 100
Enter Price: 3.50
Grocery item added successfully!
```

```
Grocery Inventory Management System
1. Add Grocery Item
2. List All Grocery Items
3. Update Quantity
4. Remove Grocery Item
5. Exit
Enter your choice: 2
Name: 100 Plus
Category: Soft drink
Quantity: 100
Price: 3.50
Status: In Stock
```

```
Grocery Inventory Management System
1. Add Grocery Item
2. List All Grocery Items
3. Update Quantity
4. Remove Grocery Item
5. Exit
Enter your choice: 3
Enter name of the grocery item to update quantity: 100 Plus
Enter new quantity for 100 Plus: 0
Quantity updated successfully!

Grocery Inventory Management System
1. Add Grocery Item
2. List All Grocery Items
3. Update Quantity
4. Remove Grocery Item
5. Exit
Enter your choice: 2
Name: 100 Plus
Category: Soft drink
Quantity: 0
Price: 3.50
Status: Out of Stock
```

```
Grocery Inventory Management System
1. Add Grocery Item
2. List All Grocery Items
3. Update Quantity
4. Remove Grocery Item
5. Exit
Enter your choice: 4
Enter name of the grocery item to remove: 100 Plus
Grocery item '100 Plus' removed successfully!

Grocery Inventory Management System
1. Add Grocery Item
2. List All Grocery Items
3. Update Quantity
4. Remove Grocery Item
5. Exit
Enter your choice: 2
No grocery items in the inventory.
```

**Note**: Write comments to explain your code as necessary.

Marking Scheme:

- Student will not be able to get more than half of the marks (2.5/5) if the program compile with syntax error.
- Correct implementation of **input/output** logic (1 mark)
- Correct implementation of **Add Grocery Item** function (1 mark)
- Correct implementation of **List Grocery Items** function (1 mark)
- Correct implementation of **Update Quantity** function (1 mark)
- Correct implementation of **Remove Grocery Item** function (1 mark)

**Sample Answer:**

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

#define MAX_ITEMS 100
#define MAX_NAME_LENGTH 100
#define MAX_CATEGORY_LENGTH 50

// Declaration and definition of struct and union (0.5 mark)
// Structure for Grocery Item
typedef struct {
    char name[MAX_NAME_LENGTH];
    char category[MAX_CATEGORY_LENGTH];
```

```
    int quantity;
    float price;
} GroceryItem;

// Union for Availability Status
typedef union {
    int inStock; // 1 for true, 0 for false
} AvailabilityStatus;

// Structure for Grocery Inventory
typedef struct {
    GroceryItem items[MAX_ITEMS];
    AvailabilityStatus availability[MAX_ITEMS];
    int count;
} GroceryInventory;

// Function Prototypes
void addGroceryItem(GroceryInventory* inventory);
void listGroceryItems(GroceryInventory* inventory);
void updateQuantity(GroceryInventory* inventory, const char* name);
void removeGroceryItem(GroceryInventory* inventory, const char* name);
void printMenu();

int main() {

    // Main program logic (with menu) (0.5 mark)
    GroceryInventory inventory = { .count = 0 };
    int choice;

    do {
        printMenu();
        printf("Enter your choice: ");
        scanf("%d", &choice);
        getchar(); // To consume the newline character

        switch (choice) {
            case 1:
                addGroceryItem(&inventory);
                break;
            case 2:
                listGroceryItems(&inventory);
                break;
            case 3: {
                char name[MAX_NAME_LENGTH];
                printf("Enter name of the grocery item to update
quantity: ");
                fgets(name, sizeof(name), stdin);
                strtok(name, "\n"); // Remove newline character
                updateQuantity(&inventory, name);
                break;
            }
            case 4: {
```

```
                char name[MAX_NAME_LENGTH];
                printf("Enter name of the grocery item to remove: ");
                fgets(name, sizeof(name), stdin);
                strtok(name, "\n"); // Remove newline character
                removeGroceryItem(&inventory, name);
                break;
            }
            case 5:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice! Please try again.\n");
        }
    } while (choice != 5);

    return 0;
}

void printMenu() {
    printf("\nGrocery Inventory Management System\n");
    printf("1. Add Grocery Item\n");
    printf("2. List All Grocery Items\n");
    printf("3. Update Quantity\n");
    printf("4. Remove Grocery Item\n");
    printf("5. Exit\n");
}

// 1 mark
void addGroceryItem(GroceryInventory* inventory) {
    if (inventory->count >= MAX_ITEMS) {
        printf("Inventory is full! Cannot add more items.\n");
        return;
    }

    GroceryItem* newItem = &inventory->items[inventory->count];
    AvailabilityStatus* status = &inventory->availability[inventory-
>count];

    printf("Enter Item Name: ");
    fgets(newItem->name, sizeof(newItem->name), stdin);
    strtok(newItem->name, "\n"); // Remove newline character

    printf("Enter Category: ");
    fgets(newItem->category, sizeof(newItem->category), stdin);
    strtok(newItem->category, "\n"); // Remove newline character

    printf("Enter Quantity: ");
    scanf("%d", &newItem->quantity);
    getchar(); // To consume the newline character

    printf("Enter Price: ");
    scanf("%f", &newItem->price);
```

```
    getchar(); // To consume the newline character

    // Set availability to in stock (1 for true)
    status->inStock = 1;
    inventory->count++;

    printf("Grocery item added successfully!\n");
}

// 1 mark
void listGroceryItems(GroceryInventory* inventory) {
    if (inventory->count == 0) {
        printf("No grocery items in the inventory.\n");
        return;
    }

    for (int i = 0; i < inventory->count; i++) {
        GroceryItem* item = &inventory->items[i];
        AvailabilityStatus* status = &inventory->availability[i];

        printf("Name: %s\n", item->name);
        printf("Category: %s\n", item->category);
        printf("Quantity: %d\n", item->quantity);
        printf("Price: %.2f\n", item->price);
        printf("Status: %s\n", status->inStock ? "In Stock" : "Out of
Stock");
        printf("\n");
    }
}

// 1 mark
void updateQuantity(GroceryInventory* inventory, const char* name) {
    for (int i = 0; i < inventory->count; i++) {
        GroceryItem* item = &inventory->items[i];
        AvailabilityStatus* status = &inventory->availability[i];

        if (strcmp(item->name, name) == 0) {
            int newQuantity;
            printf("Enter new quantity for %s: ", item->name);
            scanf("%d", &newQuantity);
            getchar(); // To consume the newline character

            item->quantity = newQuantity;

            // Update availability status
            if (newQuantity > 0) {
                status->inStock = 1; // In stock
            } else {
                status->inStock = 0; // Out of stock
            }
            printf("Quantity updated successfully!\n");
            return;
```

```
        }
    }
    printf("Grocery item with name '%s' not found.\n", name);
}

// 1 mark
void removeGroceryItem(GroceryInventory* inventory, const char* name) {
    for (int i = 0; i < inventory->count; i++) {
        GroceryItem* item = &inventory->items[i];

        if (strcmp(item->name, name) == 0) {
            // Shift items to remove the item
            for (int j = i; j < inventory->count - 1; j++) {
                inventory->items[j] = inventory->items[j + 1];
                inventory->availability[j] = inventory->availability[j +
1];
            }
            inventory->count--;
            printf("Grocery item '%s' removed successfully!\n", name);
            return;
        }
    }
    printf("Grocery item with name '%s' not found.\n", name);
}
```

## Question 3 (5 marks)

Create a C program that implements a linked list to store a collection of student records. Each record should include the student's *name*, *ID*, and *grade*. The program should allow the user to perform the following operations:

a) **Insert Student Record**: Allow the user to insert new student records into the *linked list*.
b) **Display Student Records**: Display all current student records in the list.
c) **Sort Records**: Implement a sorting algorithm to sort the student records by grade in ascending order. You may choose to implement a sorting algorithm (e.g., Merge Sort, Quick Sort, Bubble Sort, Selection Sort, Insertion Sort etc.). Put in the comment on your reason for the choice of the sorting algorithm.
d) **Search Record by ID**: Allow the user to search for a student record by student ID.
e) **Delete List**: Provide an option to delete the entire linked list and free allocated memory.

From the menu, the user can choose to *insert new student record*, *display records*, *sort records* using a sorting algorithm and *search by ID*.

Function prototypes for reference:

a) `void insertStudent(const char* name, int id, float grade); // Inserts a new student record into the linked list.`
b) `void displayStudents(); // Displays all current student records in the linked list.`
c) `Student* searchStudentByID(int id); // Searches for a student record by student ID and returns a pointer to the corresponding student node.`
d) `void deleteList(); // Deletes the entire linked list and frees the allocated memory.`
e) `void freeMemory(); // A utility function to ensure that all memory is freed when the program exits.`
f) `Student* createStudent(const char* name, int id, float grade); // Creates a new student node and returns a pointer to it.`
g) `void sortStudents(); // Sort the student records by grade according to a sorting algorithm of your choice.`

A sample program output is shown as follows (make sure the output logic is as close as possible to the sample):

```
1. Insert Student Record
2. Display Student Records
3. Sort Records
4. Search Record by ID
5. Delete List
6. Exit
Enter your choice: 1
Enter Name: aa
Enter ID: 123
Enter Grade: 88

1. Insert Student Record
2. Display Student Records
3. Sort Records
4. Search Record by ID
5. Delete List
6. Exit
Enter your choice: 1
Enter Name: bb
Enter ID: 234
Enter Grade: 77
```

```
1. Insert Student Record
2. Display Student Records
3. Sort Records
4. Search Record by ID
5. Delete List
6. Exit
Enter your choice: 1
Enter Name: cc
Enter ID: 445
Enter Grade: 100

1. Insert Student Record
2. Display Student Records
3. Sort Records
4. Search Record by ID
5. Delete List
6. Exit
Enter your choice: 2
Name: cc, ID: 445, Grade: 100.00
Name: bb, ID: 234, Grade: 77.00
Name: aa, ID: 123, Grade: 88.00
```

```
1. Insert Student Record
2. Display Student Records
3. Sort Records
4. Search Record by ID
5. Delete List
6. Exit
Enter your choice: 3
Records sorted by grade using Bubble Sort.

1. Insert Student Record
2. Display Student Records
3. Sort Records
4. Search Record by ID
5. Delete List
6. Exit
Enter your choice: 2
Name: bb, ID: 234, Grade: 77.00
Name: aa, ID: 123, Grade: 88.00
Name: cc, ID: 445, Grade: 100.00
```

Skeleton code snippet is provided as follows for your reference. Complete the program by implementing the functions based on the code snippet.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Structure for Student
typedef struct Student {
    char name[100];
    int id;
    float grade;
    struct Student* next;
} Student;

Student* head = NULL;

// Function to create a new student node
Student* createStudent(const char* name, int id, float grade) {
    // Write your code (1 mark)

}

// Function to insert a new student record
void insertStudent(const char* name, int id, float grade) {
    // Write your code (1 mark)
```

```
}

// Function to display all student records
void displayStudents() {
   // Write your code (1 mark)
}

// Function to search for a student by ID
Student* searchStudentByID(int id) {
   // Write your code (1 mark)
}

// Function to delete the entire list
void deleteList() {
    Student* current = head;
    Student* next;

    while (current != NULL) {
        next = current->next;
        free(current);
        current = next;
    }
    head = NULL;
}

void freeMemory() {
    deleteList();
}

// Sort function to sort the student records by grade based on an
algorithm of your choice
void sortStudents() {
   // Write your code (1 mark)
}

// Main function to drive the program
int main() {

    int choice, id;
    char name[100];
    float grade;

    do {
        printf("\n1. Insert Student Record\n");
        printf("2. Display Student Records\n");
        printf("3. Sort Records (choose sorting method)\n");
        printf("4. Search Record by ID\n");
        printf("5. Delete List\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        getchar(); // To consume the newline character
```

```
        switch (choice) {
            case 1:
                printf("Enter Name: ");
                fgets(name, sizeof(name), stdin);
                strtok(name, "\n");
                printf("Enter ID: ");
                scanf("%d", &id);
                printf("Enter Grade: ");
                scanf("%f", &grade);
                insertStudent(name, id, grade);
                break;
            case 2:
                displayStudents();
                break;
            case 3:
               sortStudents();
                break;
            case 4:
                printf("Enter ID to search: ");
                scanf("%d", &id);
                Student* found = searchStudentByID(id);
                if (found) {
                    printf("Found: Name: %s, ID: %d, Grade: %.2f\n",
found->name, found->id, found->grade);
                } else {
                    printf("Student not found.\n");
                }
                break;
            case 5:
                deleteList();
                printf("List deleted.\n");
                break;
            case 6:
                freeMemory();
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice! Please try again.\n");
        }
    } while (choice != 6);

    return 0;
}
```

**Note**: Write comments to explain your code as necessary.

**Marking Scheme:**

- Student will not be able to get more than half of the marks (2.5/5) if the program compile with syntax error.

- Correct implementation for `createStudent` function (1 mark)
- Correct implementation for `insertStudent` function (1 mark)
- Correct implementation for `displayStudents` function (1 mark)
- Correct implementation for `searchStudentByID` function (1 mark)
- Correct implementation for `sortStudents` function (1 mark)

**Sample Answer:**

```
// Common Structure and Functions
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct Student {
    char name[100];
    int id;
    float grade;
    struct Student* next;
} Student;

Student* head = NULL;

// Function to create a new student node
Student* createStudent(const char* name, int id, float grade) {
    // 1 mark
    Student* newStudent = (Student*)malloc(sizeof(Student));
    strcpy(newStudent->name, name);
    newStudent->id = id;
    newStudent->grade = grade;
    newStudent->next = NULL;
    return newStudent;
}

void insertStudent(const char* name, int id, float grade) {
    // 1 mark
    Student* newStudent = createStudent(name, id, grade);
    newStudent->next = head;
    head = newStudent;
}

void displayStudents() {
    // 1 mark
    Student* current = head;
    while (current != NULL) {
        printf("Name: %s, ID: %d, Grade: %.2f\n", current->name,
current->id, current->grade);
        current = current->next;
    }
}
```

```
Student* searchStudentByID(int id) {
    // 1 mark
    Student* current = head;
    while (current != NULL) {
        if (current->id == id) {
            return current;
        }
        current = current->next;
    }
    return NULL;
}

void deleteList() {
    Student* current = head;
    Student* next;

    while (current != NULL) {
        next = current->next;
        free(current);
        current = next;
    }
    head = NULL;
}

void freeMemory() {
    deleteList();
}

int main() {

    int choice, id;
    char name[100];
    float grade;

    do {
        printf("\n1. Insert Student Record\n");
        printf("2. Display Student Records\n");
        printf("3. Sort Records (choose sorting method)\n");
        printf("4. Search Record by ID\n");
        printf("5. Delete List\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        getchar(); // To consume the newline character

        switch (choice) {
            case 1:
                printf("Enter Name: ");
                fgets(name, sizeof(name), stdin);
                strtok(name, "\n");
                printf("Enter ID: ");
                scanf("%d", &id);
```

```
                printf("Enter Grade: ");
                scanf("%f", &grade);
                insertStudent(name, id, grade);
                break;
            case 2:
                displayStudents();
                break;
            case 3:
               sortStudents();
                break;
            case 4:
                printf("Enter ID to search: ");
                scanf("%d", &id);
                Student* found = searchStudentByID(id);
                if (found) {
                    printf("Found: Name: %s, ID: %d, Grade: %.2f\n",
found->name, found->id, found->grade);
                } else {
                    printf("Student not found.\n");
                }
                break;
            case 5:
                deleteList();
                printf("List deleted.\n");
                break;
            case 6:
                freeMemory();
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice! Please try again.\n");
        }
    } while (choice != 6);

    return 0;
}

// 1 mark
// Sorting Implementations (one of the following):
// Merge Sort Implementation
void merge(Student** headRef);
Student* sortedMerge(Student* a, Student* b);
void mergeSort(Student** headRef) {
    Student* head = *headRef;
    Student* a;
    Student* b;

    if (head == NULL || head->next == NULL) {
        return;
    }

    // Split the list into 'a' and 'b' sublists
```

```
    a = head;
    b = head->next;

    while (b != NULL) {
        b = b->next;
        if (b != NULL) {
            a = a->next;
            b = b->next;
        }
    }

    b = a->next;
    a->next = NULL;

    mergeSort(&head);
    mergeSort(&b);

    *headRef = sortedMerge(head, b);
}

Student* sortedMerge(Student* a, Student* b) {
    Student* result = NULL;

    if (a == NULL) return b;
    else if (b == NULL) return a;

    if (a->grade <= b->grade) {
        result = a;
        result->next = sortedMerge(a->next, b);
    } else {
        result = b;
        result->next = sortedMerge(a, b->next);
    }
    return result;
}

void sortStudents() {
    mergeSort(&head);
}

//=====================================================================
// Quick Sort Implementation
// Quick Sort
Student* partition(Student* low, Student* high);
void quickSort(Student* low, Student* high);
void sortStudents() {
    quickSort(head, NULL);
}

void quickSort(Student* low, Student* high) {
    if (high != NULL && low != high && low != high->next) {
        Student* p = partition(low, high);
```

```
        quickSort(low, p);
        quickSort(p->next, high);
    }
}

Student* partition(Student* low, Student* high) {
    float pivot = high->grade;
    Student* i = low->next;

    for (Student* j = low; j != high; j = j->next) {
        if (j->grade <= pivot) {
            if (i != j) {
                float tempGrade = i->grade;
                i->grade = j->grade;
                j->grade = tempGrade;
                int tempID = i->id;
                i->id = j->id;
                j->id = tempID;
                char tempName[100];
                strcpy(tempName, i->name);
                strcpy(i->name, j->name);
                strcpy(j->name, tempName);
            }
            i = i->next;
        }
    }
    if (i != high) {
        float tempGrade = i->grade;
        i->grade = high->grade;
        high->grade = tempGrade;
        int tempID = i->id;
        i->id = high->id;
        high->id = tempID;
        char tempName[100];
        strcpy(tempName, i->name);
        strcpy(i->name, high->name);
        strcpy(high->name, tempName);
    }
    return i;
}

//=====================================================================
// Bubble Sort implementation
void sortStudents() {
    if (head == NULL) return;
    int swapped;
    Student* ptr1;
    Student* lptr = NULL;

    do {
        swapped = 0;
        ptr1 = head;
```

```
        while (ptr1->next != lptr) {
            if (ptr1->grade > ptr1->next->grade) {
                float tempGrade = ptr1->grade;
                ptr1->grade = ptr1->next->grade;
                ptr1->next->grade = tempGrade;

                int tempID = ptr1->id;
                ptr1->id = ptr1->next->id;
                ptr1->next->id = tempID;

                char tempName[100];
                strcpy(tempName, ptr1->name);
                strcpy(ptr1->name, ptr1->next->name);
                strcpy(ptr1->next->name, tempName);

                swapped = 1;
            }
            ptr1 = ptr1->next;
        }
        lptr = ptr1;
    } while (swapped);
}

//========================================================================
// Selection Sort implementation
void sortStudents() {
    Student* current = head;

    while (current != NULL) {
        Student* minNode = current;
        Student* nextNode = current->next;

        while (nextNode != NULL) {
            if (nextNode->grade < minNode->grade) {
                minNode = nextNode;
            }
            nextNode = nextNode->next;
        }

        if (minNode != current) {
            float tempGrade = current->grade;
            current->grade = minNode->grade;
            minNode->grade = tempGrade;

            int tempID = current->id;
            current->id = minNode->id;
            minNode->id = tempID;

            char tempName[100];
            strcpy(tempName, current->name);
            strcpy(current->name, minNode->name);
```

```
            strcpy(minNode->name, tempName);
        }
        current = current->next;
    }
}

//=====================================================================
// Insertion Sort implementation
void sortStudents() {
    if (head == NULL) return;

    Student* sorted = NULL;
    Student* current = head;

    while (current != NULL) {
        Student* next = current->next;
        if (sorted == NULL || sorted->grade >= current->grade) {
            current->next = sorted;
            sorted = current;
        } else {
            Student* temp = sorted;
            while (temp->next != NULL && temp->next->grade < current-
>grade) {
                temp = temp->next;
            }
            current->next = temp->next;
            temp->next = current;
        }
        current = next;
    }
    head = sorted;
}
```

## 1.0 Instructions

1. For each question you have to provide programming solutions as separate source code files (for example: Q1.c, Q2.c and Q3.c).

2. **Submission on Moodle:** Submit you code separately one-by-one separately (for example: Q1.c, Q2.c and Q3.c) to Moodle. Make sure you have uploaded all the files successfully before you click the “Submit” button.

## 2.0 Marking Scheme

1. The evaluation is based on the following criteria:
   - Successful execution of the program – program runnable with correct inputs and outputs.
   - Correctness of functionalities as stated in each question.
   - Code quality (organization of codes)

2. Compatibility to standard C11 or C17. (If your program does not compile in such an environment, marks may be deducted.)

## 3.0 Plagiarism and Integrity

1. Codes copied and pasted directly and exactly from AI chatbots (e.g. Chatgpt, Claude, Copilot, Poe, Gemini etc.), and/or friends or other acquaintances without problem solving and coding effort will be considered as plagiarism and will not get any mark once proven.

2. You should have written every line of code yourself and should be able to explain each line fully when asked to do so (by the examiner).

3. Do not share your code with any other students.

**End of Question**