#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Structure for Student

typedef struct Student {

char name[100]; // Name of the student

int id; // Student ID

float grade; // Student grade

struct Student\* next; // Pointer to the next student in the list

} Student;

Student\* head = NULL; // Head pointer for the linked list of students

// Function to create a new student node

Student\* createStudent(const char\* name, int id, float grade) {

Student\* newStudent = (Student\*)malloc(sizeof(Student)); // Dynamically allocate memory for a new student node

if (newStudent == NULL) {

printf("Memory allocation failed.\n");

exit(1); // Exit if memory allocation fails

}

strcpy(newStudent->name, name); // Set the student's name

newStudent->id = id; // Set the student's ID

newStudent->grade = grade; // Set the student's grade

newStudent->next = NULL; // Initialize the next pointer to NULL (as this is the last node)

return newStudent;

}

// Function to insert a new student record

void insertStudent(const char\* name, int id, float grade) {

Student\* newStudent = createStudent(name, id, grade); // Create a new student node using the provided details

newStudent->next = head; // Insert the new student at the beginning of the list

head = newStudent; // Update the head to point to the new student

printf("Student record inserted successfully.\n");

}

// Function to display all student records

void displayStudents() {

if (head == NULL) { // Check if the list is empty

printf("No student records to display.\n");

return;

}

printf("\nStudent Records:\n");

Student\* current = head; // Start at the head of the list

while (current != NULL) { // Traverse through the linked list

printf("Name: %s, ID: %d, Grade: %.2f\n", current->name, current->id, current->grade); // Print student details

current = current->next; // Move to the next student in the list

}

}

// Function to search for a student by ID

Student\* searchStudentByID(int id) {

Student\* current = head; // Start at the head of the list

while (current != NULL) { // Traverse through the linked list

if (current->id == id) { // If the student ID matches, return the student node

return current;

}

current = current->next; // Move to the next student in the list

}

return NULL; // Return NULL if the student is not found

}

// Function to merge two sorted linked lists

Student\* mergeSortedLists(Student\* a, Student\* b) {

if (a == NULL) return b; // If list a is empty, return list b

if (b == NULL) return a; // If list b is empty, return list a

Student\* result;

// Compare the grades of the two students and build the merged list

if (a->grade <= b->grade) {

result = a;

result->next = mergeSortedLists(a->next, b); // Recursively merge the next nodes

} else {

result = b;

result->next = mergeSortedLists(a, b->next); // Recursively merge the next nodes

}

return result; // Return the merged sorted list

}

// Function to split a linked list into two halves

void splitList(Student\* source, Student\*\* frontRef, Student\*\* backRef) {

Student\* slow = source;

Student\* fast = source->next;

// Use the fast/slow pointer technique to find the middle of the list

while (fast != NULL) {

fast = fast->next;

if (fast != NULL) {

slow = slow->next;

fast = fast->next;

}

}

// Split the list into two halves

\*frontRef = source; // The first half starts from the source

\*backRef = slow->next; // The second half starts from the node after the slow pointer

slow->next = NULL; // Set the end of the first half to NULL

}

// Merge Sort to sort the student records by grade in ascending order

// My reason for choosing Merge Sort:

// Merge Sort is efficient with a time complexity of O(n log n), making it faster than Bubble Sort (O(n^2))

// for large dataset (if the user inputs a large number of student records). It works well with linked

// lists since it doesn't require random access to elements and it is a stable sorting algorithm.

Student\* mergeSort(Student\* head) {

// Base case: If the list is empty or has only one student, return it as is

if (head == NULL || head->next == NULL) {

return head;

}

Student \*front, \*back;

// Split the list into two halves

splitList(head, &front, &back);

// Recursively sort the two halves

front = mergeSort(front);

back = mergeSort(back);

// Merge the sorted halves

return mergeSortedLists(front, back);

}

// Function to sort the student records by grade

void sortStudents() {

if (head == NULL || head->next == NULL) {

printf("No students to sort or only one student in the list.\n");

return; // Handle empty or single-node lists

}

head = mergeSort(head); // Sort the list using mergeSort

printf("Records sorted by grade using Merge Sort.\n");

}

// Function to delete the entire list

void deleteList() {

Student\* current = head;

Student\* next;

while (current != NULL) { // Traverse through the linked list

next = current->next; // Store the next node

free(current); // Free the current node's memory

current = next; // Move to the next node

}

head = NULL; // Reset the head to NULL after deleting all nodes

}

// Function to free all memory

void freeMemory() {

deleteList(); // Delete all nodes to free memory

}

// Main function to drive the program

int main() {

int choice = 0;

int id;

char name[100];

float grade;

do {

// Display menu options

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: ");

// Validate input to ensure it's an integer

if (scanf("%d", &choice) != 1) {

// Clear invalid input

while (getchar() != '\n'); // Consume leftover input

printf("Invalid input. Please enter a number between 1 and 6.\n");

continue; // Restart the loop

}

// Handle valid input cases

switch (choice) {

case 1:

// Insert new student record

printf("Enter Name: ");

getchar(); // Consume newline character

fgets(name, sizeof(name), stdin);

strtok(name, "\n"); // Remove trailing newline

printf("Enter ID: ");

if (scanf("%d", &id) != 1) {

printf("Invalid ID. Please enter a valid integer.\n");

while (getchar() != '\n'); // Clear input

break;

}

printf("Enter Grade: ");

if (scanf("%f", &grade) != 1) {

printf("Invalid Grade. Please enter a valid floating-point number.\n");

while (getchar() != '\n'); // Clear input

break;

}

insertStudent(name, id, grade);

break;

case 2:

// Display all student records

displayStudents();

break;

case 3:

// Sort students by grade

sortStudents();

break;

case 4:

// Search for a student by ID

printf("Enter ID to search: ");

if (scanf("%d", &id) != 1) {

printf("Invalid ID. Please enter a valid integer.\n");

while (getchar() != '\n'); // Clear input

break;

}

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:

// Delete all records

deleteList();

printf("List deleted.\n");

break;

case 6:

// Exit the program

freeMemory();

printf("Exiting...\n");

break;

default:

// Handle invalid choices (outside the range of 1-6)

printf("Invalid choice. Please enter a number between 1 and 6.\n");

break;

}

} while (choice != 6); // Loop until the user chooses to exit

return 0;

}