DELECTING A NODE IN LINKED LIST(AT FIRST NODE,BETWEEN,AT LAST)

===

```
#include <stdio.h>
#include <stdlib.h>
struct Node{
    int data;
    struct Node *next;
}*first = NULL;
void create(int [], int);
void display(struct Node *);
void Insert(struct Node *,int ,int );
int DeleteNode(struct Node *, int );
int main()
{
    int A[] = {1,2,3,4,5};
    int x;
    create(A,5);
    display(first);
    Insert(first,4,6);
    printf("\n");
    display(first);
    printf("\n");
    x = DeleteNode(first, 2);
    display(first);
    printf("\n");
    printf("deletednode = %d \n",x);
    x = DeleteNode(first, 1);
    display(first);
    printf("\n");
    printf("deletednode = %d \n",x);
    x = DeleteNode(first, 4);
    display(first);
    printf("\n");
    printf("deletednode = %d \n",x);
    return 0;
}
void create(int A[], int n){
    int i;
    struct Node *temp, *last;
    first = (struct Node*)malloc(sizeof(struct Node));
    first->data = A[0];
    first->next = NULL;
    last = first;
    for(i = 1;i<n;i++){
        temp = (struct Node*)malloc(sizeof(struct Node));
        temp->data = A[i];
        temp->next = NULL;
        last->next = temp;
        last = temp;
    }
}
void display(struct Node *p){
    while(p!=NULL){
        printf("%d -> ",p->data);
        p = p->next;
    }
}
void Insert(struct Node *p,int index ,int x){
    struct Node *temp;
    int i;
    temp = (struct Node*)malloc(sizeof(struct Node));
    temp->data = x;  //x =6
    if(index ==0){
        temp->next = first;
        first = temp;
    }
    else{
        for(i=0;i<(index-1);i++){
            p = p->next;
        }
        temp->next = p->next;
        p->next =temp;
    }
}

int DeleteNode(struct Node *p, int pos){
    struct Node *q = NULL;
    int num, i;
    if(pos == 1){
        q = first;
        num = first->data; //extracting the value from the first node
        first = first->next; //moving the pointer first point to the next Node
        free(q); //deleting the first node
        return num;
    }else{
        for (i = 0; i < pos-1; i++){
            q = p;
            p = p->next;
        }
        q->next = p->next;
        num = p->data;
        free(p);
        return num;
    }

}
```

SET OF PROBLEMS
========================

Problem 1: Inventory Management System
Description: Implement a linked list to manage the inventory of raw materials.
Operations:
Create an inventory list.
Insert a new raw material.
Delete a raw material from the inventory.
Display the current inventory.

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

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

// Define a structure to represent a raw material item.
typedef struct RawMaterial {
    char name[50];          // Name of the raw material
    int quantity;           // Quantity of the raw material
    struct RawMaterial* next;  // Pointer to the next raw material
} RawMaterial;

// Function to create an empty inventory (initialize the linked list)
RawMaterial* createInventory() {
    return NULL;
}

// Function to insert a new raw material at the end of the inventory
void insertRawMaterial(RawMaterial** head, const char* name, int quantity) {
    RawMaterial* newMaterial = (RawMaterial*)malloc(sizeof(RawMaterial));
    strcpy(newMaterial->name, name);
    newMaterial->quantity = quantity;
    newMaterial->next = NULL;

    // If the inventory is empty, the new raw material becomes the first item
    if (*head == NULL) {
        *head = newMaterial;
    } else {
        // Traverse to the end of the list and insert the new raw material
        RawMaterial* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newMaterial;
    }
}

// Function to delete a raw material from the inventory
void deleteRawMaterial(RawMaterial** head, const char* name) {
    if (*head == NULL) {
        printf("Inventory is empty!\n");
        return;
    }

    RawMaterial* temp = *head;
    RawMaterial* prev = NULL;

    // If the item to be deleted is the head of the list
    if (temp != NULL && strcmp(temp->name, name) == 0) {
        *head = temp->next;  // Move the head to the next item
        free(temp);
        printf("Raw material ’%s’ deleted successfully.\n", name);
        return;
    }

    // Search for the raw material to be deleted
    while (temp != NULL && strcmp(temp->name, name) != 0) {
        prev = temp;
        temp = temp->next;
    }

    // If the raw material is not found
    if (temp == NULL) {
        printf("Raw material ’%s’ not found in inventory.\n", name);
        return;
    }

    // Remove the raw material from the list
    prev->next = temp->next;
    free(temp);
    printf("Raw material ’%s’ deleted successfully.\n", name);
}

// Function to display the current inventory
void displayInventory(RawMaterial* head) {
    if (head == NULL) {
        printf("Inventory is empty!\n");
        return;
    }

    RawMaterial* temp = head;
    printf("Current inventory:\n");
    while (temp != NULL) {
        printf("Name: %s, Quantity: %d\n", temp->name, temp->quantity);
        temp = temp->next;
    }
}

// Main function to test the Inventory Management System
int main() {
    RawMaterial* inventory = createInventory();

    int choice;
    char name[50];
    int quantity;

    while (1) {
        printf("\nInventory Management System:\n");
        printf("1. Insert Raw Material\n");
        printf("2. Delete Raw Material\n");
        printf("3. Display Inventory\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the name of the raw material: ");
                scanf("%s", name);
                printf("Enter the quantity: ");
                scanf("%d", &quantity);
                insertRawMaterial(&inventory, name, quantity);
                break;
            case 2:
                printf("Enter the name of the raw material to delete: ");
                scanf("%s", name);
                deleteRawMaterial(&inventory, name);
                break;
            case 3:
                displayInventory(inventory);
                break;
            case 4:
                printf("Exiting the Inventory Management System.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

Problem 2: Production Line Queue
Description: Use a linked list to manage the queue of tasks on a production line.
Operations:
Create a production task queue.
Insert a new task into the queue.
Delete a completed task.
Display the current task queue.

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

// Define a structure for a task in the production line
struct Task {
    int taskId;             // Task identifier
    char taskName[100];     // Task name
    struct Task* next;      // Pointer to the next task in the queue
};

// Function to create a new task
struct Task* createTask(int id, const char* name) {
    struct Task* newTask = (struct Task*)malloc(sizeof(struct Task));
    newTask->taskId = id;
    strcpy(newTask->taskName, name);
    newTask->next = NULL;
    return newTask;
}

// Function to insert a new task into the queue
void insertTask(struct Task** head, int id, const char* name) {
    struct Task* newTask = createTask(id, name);
    if (*head == NULL) {
        // If the queue is empty, the new task is the head of the queue
        *head = newTask;
    } else {
        // Traverse to the end of the queue and add the new task
        struct Task* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newTask;
    }
    printf("Task ’%s’ with ID %d added to the queue.\n", name, id);
}

// Function to delete a completed task from the queue
void deleteTask(struct Task** head) {
    if (*head == NULL) {
        printf("The task queue is empty. No task to delete.\n");
        return;
    }

    struct Task* temp = *head;
    *head = temp->next;  // Move the head to the next task
    printf("Task ’%s’ with ID %d removed from the queue.\n", temp->taskName, temp->taskId);
    free(temp);  // Free memory of the removed task
}

// Function to display the current task queue
void displayQueue(struct Task* head) {
    if (head == NULL) {
        printf("The task queue is empty.\n");
        return;
    }

    printf("Current Task Queue:\n");
    struct Task* temp = head;
    while (temp != NULL) {
        printf("Task ID: %d, Task Name: %s\n", temp->taskId, temp->taskName);
        temp = temp->next;
    }
}

int main() {
    struct Task* queue = NULL;  // Initialize an empty queue
    int choice, id;
    char name[100];

    while (1) {
        printf("\nProduction Line Task Queue Menu:\n");
        printf("1. Add a new task to the queue\n");
        printf("2. Remove a completed task from the queue\n");
        printf("3. Display the current task queue\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter task ID: ");
                scanf("%d", &id);
                printf("Enter task name: ");
                scanf(" %99[^\n]", name);  // Scan string with spaces (use the ’ ’ to skip any leading
whitespace)
                insertTask(&queue, id, name);
                break;
            case 2:
                deleteTask(&queue);
                break;
            case 3:
                displayQueue(queue);
                break;
            case 4:
                printf("Exiting program.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

Problem 3: Machine Maintenance Schedule

Description: Develop a linked list to manage the maintenance schedule of machines.

Operations:

Create a maintenance schedule.

Insert a new maintenance task.

Delete a completed maintenance task.

Display the maintenance schedule.

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

// Define the structure of a maintenance task node
struct Task {
    int taskId;
    char description[100];
    struct Task* next;
};

// Function to create the maintenance schedule (empty list)
struct Task* createSchedule() {
    return NULL;
}

// Function to insert a new task at the end of the list
void insertTask(struct Task** head, int taskId, const char* description) {
    struct Task* newTask = (struct Task*)malloc(sizeof(struct Task));
    newTask->taskId = taskId;
    strcpy(newTask->description, description);
    newTask->next = NULL;
    // If the list is empty, make the new task the head
    if (*head == NULL) {
        *head = newTask;
    } else {
        // Otherwise, find the last task and add the new task
        struct Task* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newTask;
    }
}

// Function to delete a completed task by taskId
void deleteTask(struct Task** head, int taskId) {
    struct Task* temp = *head;
    struct Task* prev = NULL;

    // If the head is the task to be deleted
    if (temp != NULL && temp->taskId == taskId) {
        *head = temp->next;
        free(temp);
        return;
    }

    // Search for the task to delete
    while (temp != NULL && temp->taskId != taskId) {
        prev = temp;
        temp = temp->next;
    }

    // If task was not found
    if (temp == NULL) {
        printf("Task with ID %d not found!\n", taskId);
        return;
    }

    // Unlink the task from the linked list
    prev->next = temp->next;
    free(temp);
}

// Function to display the entire maintenance schedule
void displaySchedule(struct Task* head) {
    if (head == NULL) {
        printf("No tasks in the maintenance schedule.\n");
        return;
    }

    struct Task* temp = head;
    while (temp != NULL) {
        printf("Task ID: %d, Description: %s\n", temp->taskId, temp->description);
        temp = temp->next;
    }
}
```

```
int main() {
    struct Task* maintenanceSchedule = createSchedule();

    // Insert some tasks
    insertTask(&maintenanceSchedule, 1, "Oil change");
    insertTask(&maintenanceSchedule, 2, "Replace filters");
    insertTask(&maintenanceSchedule, 3, "Clean air ducts");

    // Display the current maintenance schedule
    printf("Current Maintenance Schedule:\n");
    displaySchedule(maintenanceSchedule);

    // Delete a completed task
    printf("\nDeleting Task 2 (Replace filters):\n");
    deleteTask(&maintenanceSchedule, 2);

    // Display the updated schedule
    printf("\nUpdated Maintenance Schedule:\n");
    displaySchedule(maintenanceSchedule);

    return 0;
}
```

Problem 4: Employee Shift Management
Description: Use a linked list to manage employee shifts in a manufacturing plant.
Operations:
Create a shift schedule.
Insert a new shift.
Delete a completed or canceled shift.
Display the current shift schedule.

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

// Define a structure for a shift
typedef struct Shift {
    int shift_id;          // Unique shift ID
    char employee_name[50]; // Name of the employee
    char shift_time[20];    // Time of the shift (e.g., "9:00 AM - 5:00 PM")
    struct Shift* next;     // Pointer to the next shift in the list
} Shift;

// Function to create a new shift
Shift* createShift(int shift_id, const char* employee_name, const char* shift_time) {
    Shift* newShift = (Shift*)malloc(sizeof(Shift));
    newShift->shift_id = shift_id;
    strncpy(newShift->employee_name, employee_name, sizeof(newShift->employee_name) - 1);
    strncpy(newShift->shift_time, shift_time, sizeof(newShift->shift_time) - 1);
    newShift->next = NULL;
    return newShift;
}

// Function to insert a new shift at the end of the schedule
```

```
void insertShift(Shift** head, int shift_id, const char* employee_name, const char* shift_time) {
    Shift* newShift = createShift(shift_id, employee_name, shift_time);
    if (*head == NULL) {
        *head = newShift; // If the list is empty, set new shift as the head
    } else {
        Shift* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next; // Traverse to the end of the list
        }
        temp->next = newShift; // Insert the new shift at the end
    }
}

// Function to delete a shift by shift_id
void deleteShift(Shift** head, int shift_id) {
    if (*head == NULL) {
        printf("No shifts to delete.\n");
        return;
    }

    Shift* temp = *head;
    Shift* prev = NULL;

    // If the shift to be deleted is the first shift
    if (temp != NULL && temp->shift_id == shift_id) {
        *head = temp->next; // Change head
        free(temp); // Free memory
        printf("Shift %d deleted successfully.\n", shift_id);
        return;
    }

    // Search for the shift to be deleted
    while (temp != NULL && temp->shift_id != shift_id) {
        prev = temp;
        temp = temp->next;
    }

    // If shift not found
    if (temp == NULL) {
        printf("Shift %d not found.\n", shift_id);
        return;
    }

    // Unlink the shift from the list
    prev->next = temp->next;
    free(temp); // Free memory
    printf("Shift %d deleted successfully.\n", shift_id);
}

// Function to display the current shift schedule
void displayShifts(Shift* head) {
    if (head == NULL) {
        printf("No shifts scheduled.\n");
        return;
    }
    Shift* temp = head;
    printf("Current Shift Schedule:\n");
    printf("---------------------------------------------\n");
    while (temp != NULL) {
        printf("Shift ID: %d\n", temp->shift_id);
        printf("Employee Name: %s\n", temp->employee_name);
        printf("Shift Time: %s\n", temp->shift_time);
        printf("---------------------------------------------\n");
        temp = temp->next;
    }
}

int main() {
    Shift* shiftSchedule = NULL; // Initialize an empty schedule

    // Inserting some shifts into the schedule
    insertShift(&shiftSchedule, 1, "John Doe", "9:00 AM - 5:00 PM");
    insertShift(&shiftSchedule, 2, "Jane Smith", "10:00 AM - 6:00 PM");
    insertShift(&shiftSchedule, 3, "Bob Johnson", "7:00 AM - 3:00 PM");

    // Display the current shift schedule
    displayShifts(shiftSchedule);

    // Deleting a completed or canceled shift
    deleteShift(&shiftSchedule, 2); // Delete shift with ID 2

    // Display the updated shift schedule
    displayShifts(shiftSchedule);

    // Deleting a non-existent shift
    deleteShift(&shiftSchedule, 5); // Try to delete shift with ID 5 (non-existent)

    // Display the final shift schedule
    displayShifts(shiftSchedule);

    return 0;
}
```

Problem 5: Order Processing System

Description: Implement a linked list to track customer orders.

Operations:

Create an order list.

Insert a new customer order.

Delete a completed or canceled order.

Display all current orders.

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

// Define the Order structure
typedef struct Order {
    int orderId;
    char customerName[50];
    char status[20];  // E.g., "pending", "completed", "canceled"
    struct Order *next;  // Pointer to the next order in the list
} Order;

// Function to create a new order
Order* createOrder(int orderId, const char* customerName, const char* status) {
    Order* newOrder = (Order*)malloc(sizeof(Order));
    newOrder->orderId = orderId;
    strcpy(newOrder->customerName, customerName);
    strcpy(newOrder->status, status);
    newOrder->next = NULL;
    return newOrder;
}

// Function to insert a new order at the end of the list
void insertOrder(Order** head, int orderId, const char* customerName, const char* status) {
    Order* newOrder = createOrder(orderId, customerName, status);
    if (*head == NULL) {
        *head = newOrder;
    } else {
        Order* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newOrder;
    }
}

// Function to delete an order by orderId
void deleteOrder(Order** head, int orderId) {
    if (*head == NULL) {
        printf("Order list is empty.\n");
        return;
    }

    Order* temp = *head;
    Order* prev = NULL;

    // If the order to be deleted is the head node
    if (temp != NULL && temp->orderId == orderId) {
        *head = temp->next;
        free(temp);
        printf("Order with ID %d has been deleted.\n", orderId);
        return;
    }

    // Search for the order to be deleted
    while (temp != NULL && temp->orderId != orderId) {
        prev = temp;
        temp = temp->next;
    }

    // If the order wasn’t found
    if (temp == NULL) {
        printf("Order with ID %d not found.\n", orderId);
        return;
    }

    prev->next = temp->next;
    free(temp);
    printf("Order with ID %d has been deleted.\n", orderId);
}

// Function to display all orders
void displayOrders(Order* head) {
    if (head == NULL) {
        printf("No orders to display.\n");
        return;
    }

    Order* temp = head;
    while (temp != NULL) {
        printf("Order ID: %d\n", temp->orderId);
        printf("Customer: %s\n", temp->customerName);
        printf("Status: %s\n", temp->status);
        printf("------------------------\n");
        temp = temp->next;
    }
}

int main() {
    Order* orderList = NULL;

    // Inserting some orders
    insertOrder(&orderList, 1, "Alice", "pending");
    insertOrder(&orderList, 2, "Bob", "pending");
    insertOrder(&orderList, 3, "Charlie", "completed");

    // Displaying all orders
    printf("Current Orders:\n");
    displayOrders(orderList);

    // Deleting an order
    deleteOrder(&orderList, 2);

    // Displaying all orders after deletion
    printf("\nOrders after deletion:\n");
    displayOrders(orderList);

    return 0;
}
```

Problem 6: Tool Tracking System
Description: Maintain a linked list to track tools used in the manufacturing process.
Operations:
Create a tool tracking list.
Insert a new tool entry.
Delete a tool that is no longer in use.
Display all tools currently tracked

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

// Structure to represent a tool
typedef struct Tool {
    int toolId;              // Unique identifier for the tool
    char toolName[50];       // Name of the tool
    struct Tool* next;       // Pointer to the next tool in the list
} Tool;

// Function to create an empty tool tracking list (initialize head as NULL)
Tool* createToolList() {
    return NULL;
}

// Function to insert a new tool entry into the list
Tool* insertTool(Tool* head, int toolId, const char* toolName) {
    // Create a new tool node
    Tool* newTool = (Tool*)malloc(sizeof(Tool));
    if (newTool == NULL) {
        printf("Memory allocation failed!\n");
        return head;
    }

    newTool->toolId = toolId;
    strncpy(newTool->toolName, toolName, sizeof(newTool->toolName) - 1);
    newTool->toolName[sizeof(newTool->toolName) - 1] = ’\0’;  // Ensure null-termination
    newTool->next = head; // Insert at the beginning of the list

    return newTool;
}

// Function to delete a tool by its toolId
Tool* deleteTool(Tool* head, int toolId) {
    Tool *temp = head, *prev = NULL;

    // If the tool to delete is the first one in the list
    if (temp != NULL && temp->toolId == toolId) {
        head = temp->next;  // Change the head
        free(temp);         // Free the memory of the tool node
        return head;
    }

    // Search for the tool to delete
    while (temp != NULL && temp->toolId != toolId) {
        prev = temp;
        temp = temp->next;
    }

    // If tool was not found
    if (temp == NULL) {
        printf("Tool with ID %d not found!\n", toolId);
        return head;
    }

    // Unlink the tool from the list
    prev->next = temp->next;
    free(temp);  // Free the memory of the tool node

    return head;
}

// Function to display all the tools currently in the list
void displayTools(Tool* head) {
    if (head == NULL) {
        printf("No tools are currently being tracked.\n");
        return;
    }

    Tool* temp = head;
    while (temp != NULL) {
        printf("Tool ID: %d, Tool Name: %s\n", temp->toolId, temp->toolName);
        temp = temp->next;
    }
}

int main() {
    Tool* toolList = createToolList();  // Create an empty tool list

    // Inserting tools
    toolList = insertTool(toolList, 101, "Hammer");
    toolList = insertTool(toolList, 102, "Screwdriver");
    toolList = insertTool(toolList, 103, "Wrench");

    // Display all tools
    printf("Tools currently tracked:\n");
    displayTools(toolList);

    // Deleting a tool
    printf("\nDeleting tool with ID 102 (Screwdriver)...\n");
    toolList = deleteTool(toolList, 102);

    // Display all tools after deletion
    printf("\nTools currently tracked after deletion:\n");
    displayTools(toolList);

    // Delete another tool
    printf("\nDeleting tool with ID 101 (Hammer)...\n");
    toolList = deleteTool(toolList, 101);

    // Display all tools after second deletion
    printf("\nTools currently tracked after second deletion:\n");
    displayTools(toolList);

    return 0;
}
```

Problem 7: Product Assembly Line

Description: Use a linked list to manage the assembly stages of a product.
Operations:
Create an assembly line stage list.
Insert a new stage.
Delete a completed stage.
Display the current assembly stages

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

// Define the structure for a single assembly stage
typedef struct Stage {
    char stage_name[100]; // Name of the assembly stage
    struct Stage* next;   // Pointer to the next stage in the assembly line
} Stage;

// Function to create a new stage
Stage* createStage(const char* name) {
    Stage* new_stage = (Stage*)malloc(sizeof(Stage));
    if (new_stage == NULL) {
        printf("Memory allocation failed!\n");
        return NULL;
    }
    strncpy(new_stage->stage_name, name, sizeof(new_stage->stage_name) - 1);
    new_stage->stage_name[sizeof(new_stage->stage_name) - 1] = ’\0’; // Ensure null-termination
    new_stage->next = NULL;
    return new_stage;
}

// Function to insert a new stage at the end of the assembly line
void insertStage(Stage** head, const char* name) {
    Stage* new_stage = createStage(name);
    if (*head == NULL) {
        *head = new_stage; // If the list is empty, set the new stage as head
    } else {
        Stage* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next; // Traverse to the end of the list
        }
        temp->next = new_stage; // Add new stage at the end
    }
}

// Function to delete a completed stage (from the front of the list)
void deleteStage(Stage** head) {
    if (*head == NULL) {
        printf("No stages to delete!\n");
        return;
    }

    Stage* temp = *head;
    *head = (*head)->next; // Move the head pointer to the next stage
    printf("Stage ’%s’ is completed and deleted.\n", temp->stage_name);
    free(temp); // Free the memory for the deleted stage
}

// Function to display all the stages in the assembly line
void displayStages(Stage* head) {
    if (head == NULL) {
        printf("No stages in the assembly line.\n");
        return;
    }

    Stage* temp = head;
    printf("Current Assembly Stages:\n");
    while (temp != NULL) {
        printf("- %s\n", temp->stage_name);
        temp = temp->next;
    }
}

// Main function to test the above functionalities
int main() {
    Stage* assemblyLine = NULL; // Initialize an empty assembly line (linked list)

    // Insert stages into the assembly line
    insertStage(&assemblyLine, "Design");
    insertStage(&assemblyLine, "Prototype");
    insertStage(&assemblyLine, "Testing");
    insertStage(&assemblyLine, "Production");

    // Display current stages
    displayStages(assemblyLine);

    // Delete the completed stage (first stage in the list)
    deleteStage(&assemblyLine);

    // Display current stages again after deleting
    displayStages(assemblyLine);

    // Clean up remaining stages
    while (assemblyLine != NULL) {
        deleteStage(&assemblyLine);
    }

    return 0;
}
```

Problem 8: Quality Control Checklist
Description: Implement a linked list to manage a quality control checklist.
Operations:
Create a quality control checklist.
Insert a new checklist item.
Delete a completed or outdated checklist item.
Display the current quality control checklist.

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

```
// Define the structure for a checklist item
typedef struct ChecklistItem {
    char task[100];
    struct ChecklistItem* next;
} ChecklistItem;

// Function to create a new checklist item
ChecklistItem* create_item(const char* task) {
    ChecklistItem* new_item = (ChecklistItem*)malloc(sizeof(ChecklistItem));
    if (new_item != NULL) {
        strncpy(new_item->task, task, sizeof(new_item->task) - 1);
        new_item->task[sizeof(new_item->task) - 1] = ’\0’; // Ensure null termination
        new_item->next = NULL;
    }
    return new_item;
}

// Function to create an empty quality control checklist
ChecklistItem* create_checklist() {
    return NULL; // An empty checklist
}

// Function to insert a new checklist item at the end
void insert_item(ChecklistItem** head, const char* task) {
    ChecklistItem* new_item = create_item(task);
    if (*head == NULL) {
        *head = new_item;
    } else {
        ChecklistItem* current = *head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = new_item;
    }
}

// Function to delete a checklist item by task name
void delete_item(ChecklistItem** head, const char* task) {
    if (*head == NULL) {
        printf("Checklist is empty!\n");
        return;
    }

    ChecklistItem* current = *head;
    ChecklistItem* previous = NULL;

    // If the task to be deleted is the first item
    if (current != NULL && strcmp(current->task, task) == 0) {
        *head = current->next;
        free(current);
        printf("Task \"%s\" deleted successfully!\n", task);
        return;
    }
    // Search for the task to delete
    while (current != NULL && strcmp(current->task, task) != 0) {
        previous = current;
        current = current->next;
    }

    // If task not found
    if (current == NULL) {
        printf("Task \"%s\" not found!\n", task);
        return;
    }

    // Delete the task
    previous->next = current->next;
    free(current);
    printf("Task \"%s\" deleted successfully!\n", task);
}

// Function to display the current checklist
void display_checklist(ChecklistItem* head) {
    if (head == NULL) {
        printf("The checklist is empty.\n");
        return;
    }

    ChecklistItem* current = head;
    int i = 1;
    while (current != NULL) {
        printf("Task %d: %s\n", i, current->task);
        current = current->next;
        i++;
    }
}

// Main function to test the checklist operations
int main() {
    ChecklistItem* checklist = create_checklist();

    // Insert tasks into the checklist
    insert_item(&checklist, "Inspect raw materials");
    insert_item(&checklist, "Check product quality");
    insert_item(&checklist, "Test functionality");
    insert_item(&checklist, "Package products");

    // Display current checklist
    printf("Current Quality Control Checklist:\n");
    display_checklist(checklist);

    // Delete a task
    delete_item(&checklist, "Test functionality");

    // Display updated checklist
    printf("\nUpdated Quality Control Checklist:\n");
    display_checklist(checklist);
    // Try deleting a non-existing task
    delete_item(&checklist, "Perform final inspection");

    // Display final checklist
    printf("\nFinal Quality Control Checklist:\n");
    display_checklist(checklist);

    return 0;
}
```

Problem 9: Supplier Management System
Description: Use a linked list to manage a list of suppliers.
Operations:
Create a supplier list.
Insert a new supplier.
Delete an inactive or outdated supplier.
Display all current suppliers.

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

// Define a structure to store supplier information
typedef struct Supplier {
    char name[100];
    int id;
    int isActive; // 1 for active, 0 for inactive
    struct Supplier* next; // Pointer to the next supplier
} Supplier;

// Function to create a new supplier node
Supplier* createSupplier(char* name, int id, int isActive) {
    Supplier* newSupplier = (Supplier*)malloc(sizeof(Supplier));
    if (newSupplier == NULL) {
        printf("Memory allocation failed!\n");
        return NULL;
    }
    strcpy(newSupplier->name, name);
    newSupplier->id = id;
    newSupplier->isActive = isActive;
    newSupplier->next = NULL;
    return newSupplier;
}

// Function to insert a supplier into the list
void insertSupplier(Supplier** head, char* name, int id, int isActive) {
    Supplier* newSupplier = createSupplier(name, id, isActive);
    if (*head == NULL) {
        *head = newSupplier;
    } else {
        Supplier* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newSupplier;
    }
}

// Function to delete an inactive or outdated supplier
void deleteInactiveSupplier(Supplier** head, int id) {
    Supplier* temp = *head;
    Supplier* prev = NULL;

    // Check if the head itself is the supplier to be deleted
    if (temp != NULL && temp->id == id && temp->isActive == 0) {
        *head = temp->next; // Move the head pointer to the next supplier
        free(temp);
        printf("Supplier with ID %d has been deleted.\n", id);
        return;
    }

    // Traverse the list to find the supplier to delete
    while (temp != NULL && temp->id != id) {
        prev = temp;
        temp = temp->next;
    }

    // If the supplier was not found
    if (temp == NULL) {
        printf("Supplier not found.\n");
        return;
    }

    // If the supplier to be deleted is found and is inactive
    if (temp->isActive == 0) {
        prev->next = temp->next;
        free(temp);
        printf("Supplier with ID %d has been deleted.\n", id);
    } else {
        printf("Supplier with ID %d is active and cannot be deleted.\n", id);
    }
}

// Function to display all current suppliers
void displaySuppliers(Supplier* head) {
    if (head == NULL) {
        printf("No suppliers to display.\n");
        return;
    }
    Supplier* temp = head;
    printf("Current suppliers:\n");
    while (temp != NULL) {
        if (temp->isActive) {
            printf("ID: %d, Name: %s\n", temp->id, temp->name);
        }
        temp = temp->next;
    }
}
```

```
// Main function to demonstrate the supplier management system
int main() {
    Supplier* supplierList = NULL; // Initial empty supplier list

    // Inserting suppliers into the list
    insertSupplier(&supplierList, "Supplier A", 1, 1);
    insertSupplier(&supplierList, "Supplier B", 2, 0);
    insertSupplier(&supplierList, "Supplier C", 3, 1);
    insertSupplier(&supplierList, "Supplier D", 4, 0);

    // Display all current suppliers
    displaySuppliers(supplierList);

    // Deleting an inactive supplier
    deleteInactiveSupplier(&supplierList, 2);  // Deleting Supplier B (inactive)

    // Display all current suppliers again after deletion
    displaySuppliers(supplierList);

    return 0;
}
```

Problem 10: Manufacturing Project Timeline

Description: Develop a linked list to manage the timeline of a manufacturing project.

Operations:

Create a project timeline.

Insert a new project milestone.

Delete a completed milestone.

Display the current project timeline.

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

// Define a structure for a milestone
typedef struct Milestone {
    char description[100];  // Description of the milestone
    struct Milestone* next; // Pointer to the next milestone
} Milestone;

// Function to create a new milestone
Milestone* createMilestone(char* description) {
    Milestone* newMilestone = (Milestone*)malloc(sizeof(Milestone));
    if (newMilestone == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    strcpy(newMilestone->description, description);
    newMilestone->next = NULL;
    return newMilestone;
}

// Function to create an empty project timeline
Milestone* createProjectTimeline() {
    return NULL;  // Initially, the project has no milestones
```

```
}

// Function to insert a new milestone at the end of the timeline
void insertMilestone(Milestone** timeline, char* description) {
    Milestone* newMilestone = createMilestone(description);
    if (*timeline == NULL) {
        *timeline = newMilestone;  // If the timeline is empty, new milestone becomes the first milestone
    } else {
        Milestone* temp = *timeline;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newMilestone;  // Append the new milestone to the end
    }
}

// Function to delete a completed milestone by description
void deleteMilestone(Milestone** timeline, char* description) {
    Milestone* temp = *timeline;
    Milestone* prev = NULL;

    // If the timeline is empty
    if (temp == NULL) {
        printf("The project timeline is empty.\n");
        return;
    }

    // Check the first milestone
    if (temp != NULL && strcmp(temp->description, description) == 0) {
        *timeline = temp->next;  // Move the head to the next milestone
        free(temp);              // Free the memory for the deleted milestone
        printf("Milestone ’%s’ deleted successfully.\n", description);
        return;
    }

    // Search for the milestone to delete
    while (temp != NULL && strcmp(temp->description, description) != 0) {
        prev = temp;
        temp = temp->next;
    }

    // If the milestone was not found
    if (temp == NULL) {
        printf("Milestone ’%s’ not found.\n", description);
        return;
    }

    // Remove the milestone from the list
    prev->next = temp->next;
    free(temp);  // Free the memory
    printf("Milestone ’%s’ deleted successfully.\n", description);
}

// Function to display the project timeline
void displayTimeline(Milestone* timeline) {
    if (timeline == NULL) {
        printf("The project timeline is empty.\n");
        return;
    }
    Milestone* temp = timeline;
    printf("Project Timeline:\n");
    while (temp != NULL) {
        printf("- %s\n", temp->description);
        temp = temp->next;
    }
}

// Main function to test the linked list operations
int main() {
    Milestone* projectTimeline = createProjectTimeline();

    // Inserting some milestones
    insertMilestone(&projectTimeline, "Project Kickoff");
    insertMilestone(&projectTimeline, "Design Phase Completed");
    insertMilestone(&projectTimeline, "Prototype Development");
    insertMilestone(&projectTimeline, "Testing and Evaluation");

    // Display current timeline
    displayTimeline(projectTimeline);

    // Delete a completed milestone
    deleteMilestone(&projectTimeline, "Design Phase Completed");

    // Display updated timeline
    displayTimeline(projectTimeline);

    return 0;
}
```

Problem 11: Warehouse Storage Management
Description: Implement a linked list to manage the storage of goods in a warehouse.
Operations:
Create a storage list.
Insert a new storage entry.
Delete a storage entry when goods are shipped.
Display the current warehouse storage.

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

// Define a structure for each storage entry
struct Storage {
    int id;              // Unique identifier for goods
    char name[50];       // Name of the goods
    int quantity;        // Quantity of the goods in storage
    struct Storage *next;  // Pointer to the next entry in the list
};

// Function to create a new storage entry
```

```
struct Storage* createStorageEntry(int id, const char* name, int quantity) {
    struct Storage* newEntry = (struct Storage*)malloc(sizeof(struct Storage));
    if (newEntry == NULL) {
        printf("Memory allocation failed\n");
        return NULL;
    }
    newEntry->id = id;
    strncpy(newEntry->name, name, sizeof(newEntry->name) - 1);
    newEntry->name[sizeof(newEntry->name) - 1] = ’\0’;  // Ensure null-termination
    newEntry->quantity = quantity;
    newEntry->next = NULL;
    return newEntry;
}

// Function to insert a new storage entry
void insertStorageEntry(struct Storage** head, int id, const char* name, int quantity) {
    struct Storage* newEntry = createStorageEntry(id, name, quantity);
    if (newEntry == NULL) {
        return;
    }

    newEntry->next = *head;
    *head = newEntry;
    printf("Storage entry added: %s (ID: %d, Quantity: %d)\n", name, id, quantity);
}

// Function to delete a storage entry by its ID
void deleteStorageEntry(struct Storage** head, int id) {
    struct Storage* temp = *head;
    struct Storage* prev = NULL;

    // Check if the entry to be deleted is at the head
    if (temp != NULL && temp->id == id) {
        *head = temp->next;  // Move the head to the next element
        free(temp);  // Free the memory
        printf("Storage entry with ID %d deleted.\n", id);
        return;
    }

    // Search for the entry to be deleted
    while (temp != NULL && temp->id != id) {
        prev = temp;
        temp = temp->next;
    }

    // If the entry was not found
    if (temp == NULL) {
        printf("Storage entry with ID %d not found.\n", id);
        return;
    }

    // Unlink the entry from the linked list
    prev->next = temp->next;
    free(temp);  // Free the memory
    printf("Storage entry with ID %d deleted.\n", id);
}

// Function to display the current warehouse storage
void displayStorage(struct Storage* head) {
    if (head == NULL) {
        printf("Warehouse is empty.\n");
        return;
    }

    printf("Current Warehouse Storage:\n");
    struct Storage* temp = head;
    while (temp != NULL) {
        printf("ID: %d, Name: %s, Quantity: %d\n", temp->id, temp->name, temp->quantity);
        temp = temp->next;
    }
}

// Main function to demonstrate warehouse storage management
int main() {
    struct Storage* warehouse = NULL;  // Initialize an empty list

    // Insert new storage entries
    insertStorageEntry(&warehouse, 101, "Apples", 500);
    insertStorageEntry(&warehouse, 102, "Bananas", 300);
    insertStorageEntry(&warehouse, 103, "Oranges", 200);

    // Display current warehouse storage
    displayStorage(warehouse);

    // Delete a storage entry
    deleteStorageEntry(&warehouse, 102);  // Delete Bananas entry

    // Display current warehouse storage after deletion
    displayStorage(warehouse);

    // Clean up all allocated memory (free the entire list)
    while (warehouse != NULL) {
        struct Storage* temp = warehouse;
        warehouse = warehouse->next;
        free(temp);
    }

    return 0;
}
```

Problem 12: Machine Parts Inventory
Description: Use a linked list to track machine parts inventory.
Operations:
Create a parts inventory list.
Insert a new part.
Delete a part that is used up or obsolete.
Display the current parts inventory.

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

```
#include <string.h>

// Define the structure for a machine part
typedef struct Part {
    char name[50];
    int quantity;
    struct Part* next;
} Part;

// Function to create a new part
Part* createPart(const char* name, int quantity) {
    Part* newPart = (Part*)malloc(sizeof(Part));
    if (newPart == NULL) {
        printf("Memory allocation failed.\n");
        return NULL;
    }
    strcpy(newPart->name, name);
    newPart->quantity = quantity;
    newPart->next = NULL;
    return newPart;
}

// Function to insert a new part at the end of the inventory list
void insertPart(Part** head, const char* name, int quantity) {
    Part* newPart = createPart(name, quantity);
    if (*head == NULL) {
        *head = newPart;
    } else {
        Part* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newPart;
    }
}

// Function to delete a part from the inventory list
void deletePart(Part** head, const char* name) {
    if (*head == NULL) {
        printf("The inventory is empty.\n");
        return;
    }

    Part* temp = *head;
    Part* prev = NULL;

    // If the part to delete is the first part in the list
    if (temp != NULL && strcmp(temp->name, name) == 0) {
        *head = temp->next;
        free(temp);
        printf("Part ’%s’ deleted.\n", name);
        return;
    }

    // Search for the part to delete
    while (temp != NULL && strcmp(temp->name, name) != 0) {
        prev = temp;
        temp = temp->next;
    }

    // Part not found
    if (temp == NULL) {
        printf("Part ’%s’ not found in inventory.\n", name);
        return;
    }

    // Delete the part
    prev->next = temp->next;
    free(temp);
    printf("Part ’%s’ deleted.\n", name);
}

// Function to display the current inventory
void displayInventory(Part* head) {
    if (head == NULL) {
        printf("The inventory is empty.\n");
        return;
    }

    Part* temp = head;
    printf("Current inventory:\n");
    while (temp != NULL) {
        printf("Part: %s, Quantity: %d\n", temp->name, temp->quantity);
        temp = temp->next;
    }
}

int main() {
    Part* inventory = NULL;

    // Insert some parts
    insertPart(&inventory, "Bolt", 100);
    insertPart(&inventory, "Nut", 150);
    insertPart(&inventory, "Washer", 200);

    // Display the inventory
    displayInventory(inventory);

    // Delete a part
    deletePart(&inventory, "Nut");

    // Display the inventory again
    displayInventory(inventory);

    // Delete a part that doesn’t exist
    deletePart(&inventory, "Screw");

    // Display the inventory again
    displayInventory(inventory);
    return 0;
}
```

Problem 13: Packaging Line Schedule

Description: Manage the schedule of packaging tasks using a linked list.

Operations:

Create a packaging task schedule.

Insert a new packaging task.

Delete a completed packaging task.

Display the current packaging schedule.

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

// Define the structure for a Packaging Task
typedef struct Task {
    char name[100];   // Name of the packaging task
    struct Task* next; // Pointer to the next task
} Task;

// Function to create a new task
Task* create_task(const char* task_name) {
    Task* new_task = (Task*)malloc(sizeof(Task));
    if (new_task == NULL) {
        printf("Memory allocation failed.\n");
        return NULL;
    }
    strcpy(new_task->name, task_name);
    new_task->next = NULL;
    return new_task;
}

// Function to insert a new task at the end of the list
void insert_task(Task** head, const char* task_name) {
    Task* new_task = create_task(task_name);
    if (new_task == NULL) {
        return;
    }

    if (*head == NULL) {
        *head = new_task;  // The first task is inserted
    } else {
        Task* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = new_task;  // Insert at the end
    }
}

// Function to delete a task by its name
void delete_task(Task** head, const char* task_name) {
    if (*head == NULL) {
        printf("No tasks in the schedule.\n");
        return;
    }

    Task* temp = *head;
    Task* prev = NULL;

    // If the task to be deleted is the head
    if (strcmp(temp->name, task_name) == 0) {
        *head = temp->next; // Change head
        free(temp);
        printf("Task ’%s’ completed and deleted.\n", task_name);
        return;
    }

    // Search for the task to delete
    while (temp != NULL && strcmp(temp->name, task_name) != 0) {
        prev = temp;
        temp = temp->next;
    }

    // Task not found
    if (temp == NULL) {
        printf("Task ’%s’ not found.\n", task_name);
        return;
    }

    // Unlink the task from the list
    prev->next = temp->next;
    free(temp);
    printf("Task ’%s’ completed and deleted.\n", task_name);
}

// Function to display the current schedule
void display_schedule(Task* head) {
    if (head == NULL) {
        printf("No tasks in the schedule.\n");
        return;
    }

    Task* temp = head;
    printf("Current Packaging Schedule:\n");
    while (temp != NULL) {
        printf("- %s\n", temp->name);
        temp = temp->next;
    }
}

// Main function to test the schedule management
int main() {
    Task* head = NULL;  // Initialize the head of the linked list (empty schedule)

    int choice;
    char task_name[100];

    while (1) {
        printf("\nPackaging Line Schedule Menu:\n");
        printf("1. Insert a new task\n");
        printf("2. Delete a completed task\n");
        printf("3. Display the current schedule\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        // Handle different operations based on user’s choice
        switch (choice) {
            case 1:
                printf("Enter the name of the new packaging task: ");
                scanf("%s", task_name);  // Use scanf to read the task name
                insert_task(&head, task_name);
                break;

            case 2:
                printf("Enter the name of the task to delete: ");
                scanf("%s", task_name);  // Use scanf to read the task name
                delete_task(&head, task_name);
                break;

            case 3:
                display_schedule(head);
                break;

            case 4:
                // Free memory before exiting
                while (head != NULL) {
                    Task* temp = head;
                    head = head->next;
                    free(temp);
                }
                printf("Exiting the program.\n");
                return 0;

            default:
                printf("Invalid choice, please try again.\n");
        }
    }

    return 0;
}
```

Problem 14: Production Defect Tracking
Description: Implement a linked list to track defects in the production process.
Operations:
Create a defect tracking list.
Insert a new defect report.
Delete a resolved defect.
Display all current defects.

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

```
// Define the structure for a defect
typedef struct Defect {
    int id;                  // Defect ID
    char description[256];  // Defect description
    struct Defect* next;    // Pointer to the next defect
} Defect;

// Function to create a new defect
Defect* create_defect(int id, const char* description) {
    Defect* new_defect = (Defect*)malloc(sizeof(Defect));
    if (new_defect == NULL) {
        printf("Memory allocation failed!\n");
        return NULL;
    }
    new_defect->id = id;
    strncpy(new_defect->description, description, sizeof(new_defect->description) - 1);
    new_defect->description[sizeof(new_defect->description) - 1] = ’\0’;  // Ensure null-termination
    new_defect->next = NULL;
    return new_defect;
}

// Function to insert a new defect at the end of the list
void insert_defect(Defect** head, int id, const char* description) {
    Defect* new_defect = create_defect(id, description);
    if (new_defect == NULL) return;

    if (*head == NULL) {
        *head = new_defect;  // If the list is empty, set the new defect as the head
    } else {
        Defect* current = *head;
        while (current->next != NULL) {
            current = current->next;  // Traverse to the last defect
        }
        current->next = new_defect;  // Insert the new defect at the end
    }
}

// Function to delete a resolved defect by ID
void delete_defect(Defect** head, int id) {
    Defect* current = *head;
    Defect* previous = NULL;

    while (current != NULL) {
        if (current->id == id) {
            if (previous == NULL) {
                *head = current->next;  // If the defect to delete is the first one
            } else {
                previous->next = current->next;  // Bypass the defect to delete
            }
            free(current);  // Free the memory
            printf("Defect with ID %d resolved and removed.\n", id);
            return;
        }
        previous = current;
        current = current->next;
    }

    printf("Defect with ID %d not found.\n", id);  // Defect not found
}

// Function to display all defects in the list
void display_defects(Defect* head) {
    if (head == NULL) {
        printf("No defects to display.\n");
        return;
    }

    Defect* current = head;
    while (current != NULL) {
        printf("Defect ID: %d\n", current->id);
        printf("Description: %s\n", current->description);
        printf("-------------------------\n");
        current = current->next;
    }
}

int main() {
    Defect* defect_list = NULL;  // Initialize the defect tracking list

    int choice, id;
    char description[256];

    while (1) {
        printf("Production Defect Tracking System\n");
        printf("1. Insert a new defect report\n");
        printf("2. Delete a resolved defect\n");
        printf("3. Display all current defects\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:  // Insert a new defect report
                printf("Enter defect ID: ");
                scanf("%d", &id);
                printf("Enter defect description: ");
                scanf(" %[^\n]s", description);  // Reads input until newline
                insert_defect(&defect_list, id, description);
                break;

            case 2:  // Delete a resolved defect
                printf("Enter defect ID to resolve: ");
                scanf("%d", &id);
                delete_defect(&defect_list, id);
                break;

            case 3:  // Display all current defects
                display_defects(defect_list);
                break;
            case 4:  // Exit the program
                printf("Exiting the program.\n");
                return 0;

            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

Problem 15: Finished Goods Dispatch System

Description: Use a linked list to manage the dispatch schedule of finished goods.

Operations:

Create a dispatch schedule.

Insert a new dispatch entry.

Delete a dispatched or canceled entry.

Display the current dispatch schedule.

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

// Define a structure for the dispatch entry
typedef struct DispatchEntry {
    int dispatchID;
    char productName[50];
    int quantity;
    char dispatchDate[11];  // Format: YYYY-MM-DD
    struct DispatchEntry* next;
} DispatchEntry;

// Function to create a new dispatch entry
DispatchEntry* createDispatchEntry(int dispatchID, const char* productName, int quantity, const char* dispatchDate) {
    DispatchEntry* newEntry = (DispatchEntry*)malloc(sizeof(DispatchEntry));
    newEntry->dispatchID = dispatchID;
    strcpy(newEntry->productName, productName);
    newEntry->quantity = quantity;
    strcpy(newEntry->dispatchDate, dispatchDate);
    newEntry->next = NULL;
    return newEntry;
}

// Function to insert a new dispatch entry at the end of the list
void insertDispatchEntry(DispatchEntry** head, int dispatchID, const char* productName, int quantity, const char* dispatchDate) {
    DispatchEntry* newEntry = createDispatchEntry(dispatchID, productName, quantity, dispatchDate);
    if (*head == NULL) {
        *head = newEntry;
    } else {
        DispatchEntry* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newEntry;
    }
}

// Function to delete a dispatch entry by dispatchID
void deleteDispatchEntry(DispatchEntry** head, int dispatchID) {
    DispatchEntry* temp = *head;
    DispatchEntry* prev = NULL;

    if (temp != NULL && temp->dispatchID == dispatchID) {
        *head = temp->next;
        free(temp);
        return;
    }

    while (temp != NULL && temp->dispatchID != dispatchID) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Dispatch entry with ID %d not found.\n", dispatchID);
        return;
    }

    prev->next = temp->next;
    free(temp);
}

// Function to display the current dispatch schedule
void displayDispatchSchedule(DispatchEntry* head) {
    if (head == NULL) {
        printf("No dispatch entries available.\n");
        return;
    }

    DispatchEntry* temp = head;
    printf("Current Dispatch Schedule:\n");
    printf("DispatchID | Product Name | Quantity | Dispatch Date\n");
    printf("--------------------------------------------------\n");

    while (temp != NULL) {
        printf("%d | %s | %d | %s\n", temp->dispatchID, temp->productName, temp->quantity,
temp->dispatchDate);
        temp = temp->next;
    }
}

int main() {
    DispatchEntry* dispatchSchedule = NULL;  // Initialize the dispatch schedule (empty list)

    // Example operations
    insertDispatchEntry(&dispatchSchedule, 1, "ProductA", 100, "2025-01-20");
    insertDispatchEntry(&dispatchSchedule, 2, "ProductB", 150, "2025-01-21");
    insertDispatchEntry(&dispatchSchedule, 3, "ProductC", 200, "2025-01-22");

    displayDispatchSchedule(dispatchSchedule);

    printf("\nDeleting Dispatch entry with ID 2...\n");
    deleteDispatchEntry(&dispatchSchedule, 2);

    displayDispatchSchedule(dispatchSchedule);

    return 0;
}
```

SET OF LINKEDLIST PROBLEMS
==

Problem 1: Team Roster Management

Description: Implement a linked list to manage the roster of players in a sports team.Operations:
Create a team roster.
Insert a new player.
Delete a player who leaves the team.
Display the current team roster.

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

// Define a structure for a Player
typedef struct Player {
    char name[50];
    int age;
    struct Player* next;
} Player;

// Function to create an empty team roster (initially NULL)
Player* createRoster() {
    return NULL;
}

// Function to insert a new player at the end of the list
void insertPlayer(Player** head, const char* name, int age) {
    // Allocate memory for new player
    Player* newPlayer = (Player*)malloc(sizeof(Player));
    if (newPlayer == NULL) {
        printf("Memory allocation failed\n");
        return;
    }

    // Set player details
    strcpy(newPlayer->name, name);
    newPlayer->age = age;
    newPlayer->next = NULL;

    // If the roster is empty, make the new player the head
    if (*head == NULL) {
        *head = newPlayer;
    } else {
        // Otherwise, find the last player and append the new player
        Player* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newPlayer;
    }
}

// Function to delete a player by name
void deletePlayer(Player** head, const char* name) {
    Player* temp = *head;
    Player* prev = NULL;

    // Check if the player to delete is the head
    if (temp != NULL && strcmp(temp->name, name) == 0) {
        *head = temp->next;  // Move head to the next player
        free(temp);
        printf("Player %s deleted from the roster.\n", name);
        return;
    }

    // Traverse the list to find the player
    while (temp != NULL && strcmp(temp->name, name) != 0) {
        prev = temp;
        temp = temp->next;
    }

    // If player not found
    if (temp == NULL) {
        printf("Player %s not found.\n", name);
        return;
    }

    // Remove the player from the list
    prev->next = temp->next;
    free(temp);
    printf("Player %s deleted from the roster.\n", name);
}

// Function to display the current team roster
void displayRoster(Player* head) {
    if (head == NULL) {
        printf("The team roster is empty.\n");
        return;
    }

    Player* temp = head;
    printf("Current Team Roster:\n");
    while (temp != NULL) {
        printf("Name: %s, Age: %d\n", temp->name, temp->age);
        temp = temp->next;
    }
```

```
}

// Function to free the entire roster
void freeRoster(Player* head) {
    Player* temp;
    while (head != NULL) {
        temp = head;
        head = head->next;
        free(temp);
    }
}

int main() {
    Player* roster = createRoster();  // Create an empty roster
    int choice;
    char name[50];
    int age;

    while (1) {
        printf("\nMenu:\n");
        printf("1. Insert Player\n");
        printf("2. Delete Player\n");
        printf("3. Display Roster\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                // Insert player
                printf("Enter player’s name: ");
                scanf("%s", name);  // Using scanf for name input

                printf("Enter player’s age: ");
                scanf("%d", &age);

                insertPlayer(&roster, name, age);
                break;

            case 2:
                // Delete player
                printf("Enter player’s name to delete: ");
                scanf("%s", name);  // Using scanf for name input

                deletePlayer(&roster, name);
                break;

            case 3:
                // Display roster
                displayRoster(roster);
                break;

            case 4:
                // Exit
                freeRoster(roster); // Free the memory before exiting
                printf("Exiting...\n");
                return 0;

            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

Problem 2: Tournament Match Scheduling

Description: Use a linked list to schedule matches in a tournament.Operations:

Create a match schedule.

Insert a new match.

Delete a completed or canceled match.

Display the current match schedule.

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

typedef struct Match {
    int matchId;
    char team1[50];
    char team2[50];
    char matchDate[20];
    char status[20];  // e.g., "scheduled", "completed", "canceled"
    struct Match* next;
} Match;

Match* head = NULL;  // Start of the linked list

// Function prototypes
void createMatchSchedule();
void insertMatch(int matchId, const char* team1, const char* team2, const char* matchDate, const char* status);
void deleteMatch(int matchId);
void displaySchedule();
void menu();

int main() {
    menu();
    return 0;
}

// Display the menu for user interaction
void menu() {
    int choice, matchId;
    char team1[50], team2[50], matchDate[20], status[20];

    while (1) {
        printf("\nTournament Match Scheduling System\n");
        printf("1. Create match schedule\n");
        printf("2. Insert a new match\n");
        printf("3. Delete a completed or canceled match\n");
        printf("4. Display current match schedule\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                createMatchSchedule();
                break;
            case 2:
                printf("Enter match ID: ");
                scanf("%d", &matchId);
                printf("Enter team 1 name: ");
                scanf("%s", team1);
                printf("Enter team 2 name: ");
                scanf("%s", team2);
                printf("Enter match date: ");
                scanf("%s", matchDate);
                printf("Enter match status: ");
                scanf("%s", status);
                insertMatch(matchId, team1, team2, matchDate, status);
                break;
            case 3:
                printf("Enter match ID to delete: ");
                scanf("%d", &matchId);
                deleteMatch(matchId);
                break;
            case 4:
                displaySchedule();
                break;
            case 5:
                printf("Exiting the system.\n");
                exit(0);
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }
}

// Create a new match schedule (Initial match)
void createMatchSchedule() {
    head = NULL;
    printf("Match schedule created.\n");
}

// Insert a new match at the end of the list
void insertMatch(int matchId, const char* team1, const char* team2, const char* matchDate, const char*
status) {
    Match* newMatch = (Match*)malloc(sizeof(Match));
    newMatch->matchId = matchId;
    strcpy(newMatch->team1, team1);
    strcpy(newMatch->team2, team2);
    strcpy(newMatch->matchDate, matchDate);
    strcpy(newMatch->status, status);
    newMatch->next = NULL;

    if (head == NULL) {
        head = newMatch;  // First match in the list
    } else {
        Match* temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newMatch;  // Insert at the end of the list
    }
    printf("Match with ID %d inserted.\n", matchId);
}

// Delete a match from the list by match ID
void deleteMatch(int matchId) {
    Match *temp = head, *prev = NULL;

    // If head node itself holds the matchId to be deleted
    if (temp != NULL && temp->matchId == matchId) {
        head = temp->next;  // Move head to the next match
        free(temp);  // Free memory
        printf("Match with ID %d deleted.\n", matchId);
        return;
    }

    // Search for the match to delete
    while (temp != NULL && temp->matchId != matchId) {
        prev = temp;
        temp = temp->next;
    }

    // If match not found
    if (temp == NULL) {
        printf("Match with ID %d not found.\n", matchId);
        return;
    }

    // Unlink the match from the list
    prev->next = temp->next;
    free(temp);  // Free memory
    printf("Match with ID %d deleted.\n", matchId);
}

// Display the current match schedule
void displaySchedule() {
    if (head == NULL) {
        printf("No matches scheduled.\n");
        return;
    }

    Match* temp = head;
    printf("\nCurrent Match Schedule:\n");
    printf("Match ID\tTeam 1\t\tTeam 2\t\tMatch Date\tStatus\n");
    while (temp != NULL) {
        printf("%d\t\t%s\t\t%s\t\t%s\t\t%s\n", temp->matchId, temp->team1, temp->team2, temp->matchDate, temp->status);
        temp = temp->next;
    }
}
```

3.Problem 3: Athlete Training Log

Description: Develop a linked list to log training sessions for athletes.Operations:

Create a training log.

Insert a new training session.

Delete a completed or canceled session.

Display the training log.

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

// Define the structure of a training session (node in the linked list)
typedef struct Session {
    int sessionId;
    char description[100];
    struct Session* next; // Pointer to the next session
} Session;

// Function to create a new training session (node)
Session* createSession(int sessionId, const char* description) {
    Session* newSession = (Session*)malloc(sizeof(Session));
    newSession->sessionId = sessionId;
    strcpy(newSession->description, description);
    newSession->next = NULL;
    return newSession;
}

// Function to insert a new session into the log (at the end)
void insertSession(Session** head, int sessionId, const char* description) {
    Session* newSession = createSession(sessionId, description);
    if (*head == NULL) {
        *head = newSession; // If the log is empty, the new session is the first one
    } else {
        Session* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next; // Traverse to the end of the list
        }
        temp->next = newSession; // Add new session at the end
    }
}

// Function to delete a session from the log (by sessionId)
void deleteSession(Session** head, int sessionId) {
    if (*head == NULL) {
        printf("The log is empty. Nothing to delete.\n");
        return;
    }

    Session* temp = *head;
    Session* prev = NULL;

    // If the session to be deleted is the head node
    if (temp != NULL && temp->sessionId == sessionId) {
        *head = temp->next; // Change head
        free(temp);
        printf("Session %d deleted.\n", sessionId);
        return;
    }

    // Search for the session to delete
    while (temp != NULL && temp->sessionId != sessionId) {
        prev = temp;
        temp = temp->next;
    }

    // If session was not found
    if (temp == NULL) {
        printf("Session %d not found.\n", sessionId);
        return;
    }

    // Unlink the node from the list
    prev->next = temp->next;
    free(temp);
    printf("Session %d deleted.\n", sessionId);
}

// Function to display the training log
void displayLog(Session* head) {
    if (head == NULL) {
        printf("Training log is empty.\n");
        return;
    }
    Session* temp = head;
    while (temp != NULL) {
        printf("Session ID: %d, Description: %s\n", temp->sessionId, temp->description);
        temp = temp->next;
    }
}

// Main function to test the log operations
int main() {
    Session* log = NULL; // Initialize an empty log

    int choice, sessionId;
    char description[100];

    while (1) {
        printf("\n--- Athlete Training Log ---\n");
        printf("1. Insert a new session\n");
        printf("2. Delete a session\n");
        printf("3. Display training log\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: // Insert a new session
                printf("Enter session ID: ");
                scanf("%d", &sessionId);
                printf("Enter session description: ");
                scanf(" %[^\n]%*c", description);  // Read a full line of input (description)
                insertSession(&log, sessionId, description);
                break;
            case 2: // Delete a session
                printf("Enter session ID to delete: ");
                scanf("%d", &sessionId);
                deleteSession(&log, sessionId);
                break;
            case 3: // Display training log
                displayLog(log);
                break;
            case 4: // Exit
                printf("Exiting program.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

Problem 4: Sports Equipment Inventory

Description: Use a linked list to manage the inventory of sports equipment.Operations:

Create an equipment inventory list.

Insert a new equipment item.

Delete an item that is no longer usable.

Display the current equipment inventory.

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

// Define a structure to represent an equipment item
typedef struct Equipment {
    char name[100];
    int quantity;
    struct Equipment* next;
} Equipment;

// Function to create a new equipment item
Equipment* createEquipment(char* name, int quantity) {
    Equipment* newEquipment = (Equipment*)malloc(sizeof(Equipment));
    if (!newEquipment) {
        printf("Memory allocation failed\n");
        return NULL;
    }
    strcpy(newEquipment->name, name);
    newEquipment->quantity = quantity;
    newEquipment->next = NULL;
    return newEquipment;
}

// Function to insert a new equipment item at the end of the list
void insertEquipment(Equipment** head, char* name, int quantity) {
    Equipment* newEquipment = createEquipment(name, quantity);
    if (!newEquipment) return;

    if (*head == NULL) {
        *head = newEquipment;
    } else {
        Equipment* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newEquipment;
    }
}

// Function to delete an equipment item by name
void deleteEquipment(Equipment** head, char* name) {
    if (*head == NULL) {
        printf("Inventory is empty!\n");
        return;
    }

    Equipment* temp = *head;
    Equipment* prev = NULL;

    // Check if the item to delete is the head
    if (temp != NULL && strcmp(temp->name, name) == 0) {
        *head = temp->next;
        free(temp);
        printf("Item ’%s’ deleted from inventory.\n", name);
        return;
    }

    // Search for the item to delete
    while (temp != NULL && strcmp(temp->name, name) != 0) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Item ’%s’ not found in inventory.\n", name);
        return;
    }

    // Unlink the node from the linked list
    prev->next = temp->next;
    free(temp);
    printf("Item ’%s’ deleted from inventory.\n", name);
}
```

```
// Function to display the current inventory
void displayInventory(Equipment* head) {
    if (head == NULL) {
        printf("Inventory is empty.\n");
        return;
    }

    Equipment* temp = head;
    printf("Current Inventory:\n");
    while (temp != NULL) {
        printf("Item: %s, Quantity: %d\n", temp->name, temp->quantity);
        temp = temp->next;
    }
}

// Main function to demonstrate the operations
int main() {
    Equipment* inventory = NULL;

    // Inserting some equipment items
    insertEquipment(&inventory, "Basketball", 10);
    insertEquipment(&inventory, "Soccer Ball", 5);
    insertEquipment(&inventory, "Tennis Racket", 3);

    // Display the current inventory
    displayInventory(inventory);

    // Deleting an item
    deleteEquipment(&inventory, "Soccer Ball");

    // Display the current inventory again
    displayInventory(inventory);

    // Deleting an item that doesn’t exist
    deleteEquipment(&inventory, "Baseball Bat");

    // Display the final inventory
    displayInventory(inventory);

    // Free remaining memory
    while (inventory != NULL) {
        Equipment* temp = inventory;
        inventory = inventory->next;
        free(temp);
    }

    return 0;
}
```

Problem 5: Player Performance Tracking
Description: Implement a linked list to track player performance over the season.Operations:
Create a performance record list.
Insert a new performance entry.
Delete an outdated or erroneous entry.

Display all performance records.

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

// Define the structure for storing player performance data
typedef struct Performance {
    int playerID;             // Unique identifier for the player
    char playerName[50];      // Name of the player
    int matchesPlayed;        // Number of matches played
    int goalsScored;          // Goals scored in the season
    int assists;              // Assists made in the season
    struct Performance *next; // Pointer to the next record in the list
} Performance;

// Function to create a new performance record
Performance* createRecord(int playerID, const char* playerName, int matchesPlayed, int goalsScored, int
assists) {
    Performance* newRecord = (Performance*)malloc(sizeof(Performance));
    newRecord->playerID = playerID;
    strncpy(newRecord->playerName, playerName, sizeof(newRecord->playerName) - 1);
    newRecord->matchesPlayed = matchesPlayed;
    newRecord->goalsScored = goalsScored;
    newRecord->assists = assists;
    newRecord->next = NULL;
    return newRecord;
}

// Function to insert a new performance entry at the end of the list
void insertPerformance(Performance** head, int playerID, const char* playerName, int matchesPlayed, int
goalsScored, int assists) {
    Performance* newRecord = createRecord(playerID, playerName, matchesPlayed, goalsScored,
assists);
    if (*head == NULL) {
        *head = newRecord;
    } else {
        Performance* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newRecord;
    }
}

// Function to delete a performance entry by player ID
void deletePerformance(Performance** head, int playerID) {
    Performance* temp = *head;
    Performance* prev = NULL;

    // If the player to be deleted is the head
    if (temp != NULL && temp->playerID == playerID) {
        *head = temp->next;
        free(temp);
        return;
```

```c
    }

    // Search for the player to be deleted
    while (temp != NULL && temp->playerID != playerID) {
        prev = temp;
        temp = temp->next;
    }

    // If player was not found
    if (temp == NULL) {
        printf("Player with ID %d not found.\n", playerID);
        return;
    }

    // Unlink the player record from the list
    prev->next = temp->next;
    free(temp);
}

// Function to display all performance records
void displayPerformanceRecords(Performance* head) {
    if (head == NULL) {
        printf("No performance records available.\n");
        return;
    }

    Performance* temp = head;
    printf("Player ID | Player Name      | Matches Played | Goals Scored | Assists\n");
    printf("----------------------------------------------------------------------\n");

    while (temp != NULL) {
        printf("%9d | %-16s | %14d | %12d | %7d\n", temp->playerID, temp->playerName,
temp->matchesPlayed, temp->goalsScored, temp->assists);
        temp = temp->next;
    }
}

// Main function to demonstrate the functionality
int main() {
    Performance* head = NULL;

    // Insert some performance records
    insertPerformance(&head, 101, "John Doe", 20, 15, 7);
    insertPerformance(&head, 102, "Jane Smith", 22, 18, 10);
    insertPerformance(&head, 103, "David Johnson", 18, 12, 5);

    // Display the performance records
    printf("Performance Records:\n");
    displayPerformanceRecords(head);

    // Delete a performance entry by player ID
    printf("\nDeleting player with ID 102:\n");
    deletePerformance(&head, 102);
    displayPerformanceRecords(head);
    // Try deleting a non-existent player
    printf("\nAttempting to delete a non-existent player with ID 999:\n");
    deletePerformance(&head, 999);

    // Display the final list of records
    printf("\nFinal Performance Records:\n");
    displayPerformanceRecords(head);

    // Free the memory used by the list
    Performance* temp;
    while (head != NULL) {
        temp = head;
        head = head->next;
        free(temp);
    }

    return 0;
}
```

Problem 6: Event Registration System

Description: Use a linked list to manage athlete registrations for sports events.Operations:
Create a registration list.
Insert a new registration.
Delete a canceled registration.
Display all current registrations.

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

// Define a structure to represent each registration
typedef struct Registration {
    int athleteID;
    char athleteName[100];
    struct Registration* next;
} Registration;

// Function to create a new registration node
Registration* createNode(int athleteID, const char* athleteName) {
    Registration* newNode = (Registration*)malloc(sizeof(Registration));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    newNode->athleteID = athleteID;
    strncpy(newNode->athleteName, athleteName, sizeof(newNode->athleteName) - 1);
    newNode->athleteName[sizeof(newNode->athleteName) - 1] = ’\0’;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a new registration into the list
void insertRegistration(Registration** head, int athleteID, const char* athleteName) {
    Registration* newNode = createNode(athleteID, athleteName);
    newNode->next = *head;
    *head = newNode;
}

// Function to delete a canceled registration from the list
void deleteRegistration(Registration** head, int athleteID) {
    Registration* temp = *head;
    Registration* prev = NULL;

    // If the registration to be deleted is the head node
    if (temp != NULL && temp->athleteID == athleteID) {
        *head = temp->next;
        free(temp);
        return;
    }

    // Search for the registration to be deleted
    while (temp != NULL && temp->athleteID != athleteID) {
        prev = temp;
        temp = temp->next;
    }

    // If the athlete is not found
    if (temp == NULL) {
        printf("Athlete with ID %d not found.\n", athleteID);
        return;
    }

    // Unlink the node from the list
    prev->next = temp->next;
    free(temp);
}

// Function to display all current registrations
void displayRegistrations(Registration* head) {
    if (head == NULL) {
        printf("No registrations available.\n");
        return;
    }

    Registration* temp = head;
    while (temp != NULL) {
        printf("Athlete ID: %d, Name: %s\n", temp->athleteID, temp->athleteName);
        temp = temp->next;
    }
}

// Main function to test the event registration system
int main() {
    Registration* head = NULL;

    // Inserting new registrations
    insertRegistration(&head, 101, "John Doe");
    insertRegistration(&head, 102, "Jane Smith");
    insertRegistration(&head, 103, "Emily Davis");
    // Display current registrations
    printf("Current registrations:\n");
    displayRegistrations(head);

    // Deleting a canceled registration
    printf("\nCanceling registration for athlete with ID 102:\n");
    deleteRegistration(&head, 102);

    // Display updated registrations
    printf("\nUpdated registrations:\n");
    displayRegistrations(head);

    // Deleting a non-existent registration
    printf("\nAttempting to cancel athlete with ID 999:\n");
    deleteRegistration(&head, 999);

    // Final list of registrations
    printf("\nFinal registrations:\n");
    displayRegistrations(head);

    return 0;
}
```

Problem 7: Sports League Standings
Description: Develop a linked list to manage the standings of teams in a sports league.Operations:
Create a league standings list.
Insert a new team.
Delete a team that withdraws.
Display the current league standings.

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

// Define a structure for a team
struct Team {
    char name[50];
    int points;
    struct Team* next; // Pointer to the next team in the list
};

// Function to create a new team
struct Team* createTeam(char* name, int points) {
    struct Team* newTeam = (struct Team*)malloc(sizeof(struct Team));
    if (!newTeam) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    strcpy(newTeam->name, name);
    newTeam->points = points;
    newTeam->next = NULL;
    return newTeam;
}

// Function to insert a new team in the standings list (sorted by points)
```

```
void insertTeam(struct Team** head, char* name, int points) {
    struct Team* newTeam = createTeam(name, points);
    struct Team* temp = *head;

    // If the list is empty, or the new team should be at the beginning
    if (*head == NULL || (*head)->points < points) {
        newTeam->next = *head;
        *head = newTeam;
        return;
    }

    // Traverse the list to find the correct position
    while (temp->next != NULL && temp->next->points >= points) {
        temp = temp->next;
    }

    // Insert the new team in the sorted order
    newTeam->next = temp->next;
    temp->next = newTeam;
}

// Function to delete a team by name
void deleteTeam(struct Team** head, char* name) {
    struct Team* temp = *head;
    struct Team* prev = NULL;

    // If the team to delete is the first one
    if (temp != NULL && strcmp(temp->name, name) == 0) {
        *head = temp->next; // Move the head to the next team
        free(temp);
        return;
    }

    // Search for the team to delete
    while (temp != NULL && strcmp(temp->name, name) != 0) {
        prev = temp;
        temp = temp->next;
    }

    // If the team wasn’t found
    if (temp == NULL) {
        printf("Team %s not found!\n", name);
        return;
    }

    // Unlink the node from the list
    prev->next = temp->next;
    free(temp);
}

// Function to display the current standings
void displayStandings(struct Team* head) {
    struct Team* temp = head;
    int rank = 1;
    if (temp == NULL) {
        printf("No teams in the standings.\n");
        return;
    }

    printf("Current League Standings:\n");
    printf("Rank | Team Name              | Points\n");
    printf("--------------------------------------\n");

    while (temp != NULL) {
        printf("%-4d | %-22s | %-6d\n", rank, temp->name, temp->points);
        rank++;
        temp = temp->next;
    }
}

int main() {
    struct Team* league = NULL;

    // Inserting some teams
    insertTeam(&league, "Team A", 30);
    insertTeam(&league, "Team B", 40);
    insertTeam(&league, "Team C", 35);

    // Displaying current standings
    displayStandings(league);

    // Inserting another team
    insertTeam(&league, "Team D", 50);
    printf("\nAfter adding Team D:\n");
    displayStandings(league);

    // Deleting a team that withdraws
    deleteTeam(&league, "Team B");
    printf("\nAfter Team B withdraws:\n");
    displayStandings(league);

    // Deleting a team that is not in the list
    deleteTeam(&league, "Team X");

    return 0;
}
```

Problem 8: Match Result Recording

Description: Implement a linked list to record results of matches.Operations:

Create a match result list.

Insert a new match result.

Delete an incorrect or outdated result.

Display all recorded match results.

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

// Define the structure for a match result
```

```
typedef struct MatchResult {
    char team1[50];
    char team2[50];
    int score1;
    int score2;
    struct MatchResult *next;
} MatchResult;

// Function to create a new match result node
MatchResult* createMatchResult(const char *team1, const char *team2, int score1, int score2) {
    MatchResult *newMatch = (MatchResult*)malloc(sizeof(MatchResult));
    if (newMatch == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    strcpy(newMatch->team1, team1);
    strcpy(newMatch->team2, team2);
    newMatch->score1 = score1;
    newMatch->score2 = score2;
    newMatch->next = NULL;
    return newMatch;
}

// Function to insert a new match result at the end of the list
void insertMatchResult(MatchResult **head, const char *team1, const char *team2, int score1, int score2)
{
    MatchResult *newMatch = createMatchResult(team1, team2, score1, score2);
    if (*head == NULL) {
        *head = newMatch;
    } else {
        MatchResult *temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newMatch;
    }
}

// Function to delete a match result by matching the teams
void deleteMatchResult(MatchResult **head, const char *team1, const char *team2) {
    if (*head == NULL) {
        printf("No matches to delete.\n");
        return;
    }
    MatchResult *temp = *head;
    MatchResult *prev = NULL;

    // If the match to be deleted is the first match
    if (strcmp(temp->team1, team1) == 0 && strcmp(temp->team2, team2) == 0) {
        *head = temp->next;
        free(temp);
        printf("Match result deleted successfully.\n");
        return;
    }
    // Search for the match to delete
    while (temp != NULL && (strcmp(temp->team1, team1) != 0 || strcmp(temp->team2, team2) != 0)) {
        prev = temp;
        temp = temp->next;
    }

    // If match was not found
    if (temp == NULL) {
        printf("Match result not found.\n");
        return;
    }

    // Unlink the match node
    prev->next = temp->next;
    free(temp);
    printf("Match result deleted successfully.\n");
}

// Function to display all the match results
void displayMatchResults(MatchResult *head) {
    if (head == NULL) {
        printf("No match results recorded.\n");
        return;
    }

    MatchResult *temp = head;
    while (temp != NULL) {
        printf("Match: %s vs %s | Score: %d - %d\n", temp->team1, temp->team2, temp->score1,
temp->score2);
        temp = temp->next;
    }
}

int main() {
    MatchResult *head = NULL;

    // Inserting match results
    insertMatchResult(&head, "Team A", "Team B", 3, 1);
    insertMatchResult(&head, "Team C", "Team D", 2, 2);
    insertMatchResult(&head, "Team A", "Team C", 1, 0);

    // Display all match results
    printf("Match Results:\n");
    displayMatchResults(head);

    // Deleting a match result
    printf("\nDeleting match result for Team A vs Team C...\n");
    deleteMatchResult(&head, "Team A", "Team C");

    // Display all match results after deletion
    printf("\nUpdated Match Results:\n");
    displayMatchResults(head);

    // Deleting a non-existing match
    printf("\nAttempting to delete non-existing match result...\n");
    deleteMatchResult(&head, "Team X", "Team Y");

    // Display all match results after attempted deletion
    printf("\nFinal Match Results:\n");
    displayMatchResults(head);

    return 0;
}
```

Problem 9: Player Injury Tracker
Description: Use a linked list to track injuries of players.Operations:
Create an injury tracker list.
Insert a new injury report.
Delete a resolved or erroneous injury report.
Display all current injury reports.

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

// Structure to represent a player injury report
typedef struct Injury {
    int playerId;          // Unique player ID
    char injuryDescription[100]; // Injury description
    char injuryStatus[20]; // Status of the injury (e.g., "Resolved", "Active")
    struct Injury* next;   // Pointer to the next injury report
} Injury;

// Function to create a new injury report
Injury* createInjury(int playerId, const char* injuryDescription, const char* injuryStatus) {
    Injury* newInjury = (Injury*)malloc(sizeof(Injury));
    newInjury->playerId = playerId;
    strncpy(newInjury->injuryDescription, injuryDescription, sizeof(newInjury->injuryDescription) - 1);
    strncpy(newInjury->injuryStatus, injuryStatus, sizeof(newInjury->injuryStatus) - 1);
    newInjury->next = NULL;
    return newInjury;
}

// Function to insert a new injury report at the end of the list
void insertInjury(Injury** head, int playerId, const char* injuryDescription, const char* injuryStatus) {
    Injury* newInjury = createInjury(playerId, injuryDescription, injuryStatus);
    if (*head == NULL) {
        *head = newInjury;
    } else {
        Injury* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newInjury;
    }
}

// Function to delete an injury report by playerId
void deleteInjury(Injury** head, int playerId) {
    if (*head == NULL) {
        printf("No injury reports to delete.\n");
        return;
    }

    Injury* temp = *head;
    Injury* prev = NULL;

    // If the head contains the injury report to be deleted
    if (temp != NULL && temp->playerId == playerId) {
        *head = temp->next; // Move the head pointer to the next injury
        free(temp);
        printf("Injury report for player %d deleted.\n", playerId);
        return;
    }

    // Search for the injury report to be deleted
    while (temp != NULL && temp->playerId != playerId) {
        prev = temp;
        temp = temp->next;
    }

    // If the playerId was not found
    if (temp == NULL) {
        printf("Injury report for player %d not found.\n", playerId);
        return;
    }

    // Unlink the node from the linked list
    prev->next = temp->next;
    free(temp);
    printf("Injury report for player %d deleted.\n", playerId);
}

// Function to display all injury reports
void displayInjuries(Injury* head) {
    if (head == NULL) {
        printf("No injury reports to display.\n");
        return;
    }

    Injury* temp = head;
    printf("Current Injury Reports:\n");
    while (temp != NULL) {
        printf("Player ID: %d\n", temp->playerId);
        printf("Injury: %s\n", temp->injuryDescription);
        printf("Status: %s\n", temp->injuryStatus);
        printf("------------------------\n");
        temp = temp->next;
    }
}

// Main function to demonstrate the functionality
int main() {
    Injury* injuryTracker = NULL;  // Initialize an empty linked list
    // Inserting some injury reports
    insertInjury(&injuryTracker, 101, "Sprained Ankle", "Active");
    insertInjury(&injuryTracker, 102, "Knee Surgery", "Active");
    insertInjury(&injuryTracker, 103, "Back Strain", "Resolved");

    // Displaying all injury reports
    displayInjuries(injuryTracker);

    // Deleting a resolved injury report
    deleteInjury(&injuryTracker, 103);

    // Displaying all injury reports again
    displayInjuries(injuryTracker);

    // Attempt to delete a non-existent report
    deleteInjury(&injuryTracker, 104);

    // Displaying all injury reports after deletion
    displayInjuries(injuryTracker);

    return 0;
}
```

Problem 10: Sports Facility Booking System

Description: Manage bookings for sports facilities using a linked list.Operations:
Create a booking list.
Insert a new booking.
Delete a canceled or completed booking.
Display all current bookings.

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

// Structure for a booking
typedef struct Booking {
    int bookingID;  // Unique ID for each booking
    char customerName[100];  // Name of the customer
    char facility[50];  // Name of the facility booked
    char date[20];  // Date of booking
    struct Booking* next;  // Pointer to next booking
} Booking;

// Function to create a new booking node
Booking* createBooking(int bookingID, const char* customerName, const char* facility, const char* date)
{
    Booking* newBooking = (Booking*)malloc(sizeof(Booking));
    if (!newBooking) {
        printf("Memory allocation failed!\n");
        return NULL;
    }
    newBooking->bookingID = bookingID;
    strcpy(newBooking->customerName, customerName);
    strcpy(newBooking->facility, facility);
    strcpy(newBooking->date, date);
    newBooking->next = NULL;
    return newBooking;
}

// Function to insert a new booking at the end of the list
void insertBooking(Booking** head, int bookingID, const char* customerName, const char* facility, const
char* date) {
    Booking* newBooking = createBooking(bookingID, customerName, facility, date);
    if (*head == NULL) {
        *head = newBooking;
    } else {
        Booking* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newBooking;
    }
    printf("Booking added successfully!\n");
}

// Function to delete a booking by booking ID
void deleteBooking(Booking** head, int bookingID) {
    if (*head == NULL) {
        printf("No bookings to delete!\n");
        return;
    }

    Booking* temp = *head;
    Booking* prev = NULL;

    // If the booking to be deleted is the head node
    if (temp != NULL && temp->bookingID == bookingID) {
        *head = temp->next;  // Change the head
        free(temp);
        printf("Booking with ID %d deleted successfully!\n", bookingID);
        return;
    }

    // Search for the booking to be deleted
    while (temp != NULL && temp->bookingID != bookingID) {
        prev = temp;
        temp = temp->next;
    }

    // If booking ID was not found
    if (temp == NULL) {
        printf("Booking with ID %d not found!\n", bookingID);
        return;
    }

    // Unlink the booking from the list
    prev->next = temp->next;
    free(temp);
    printf("Booking with ID %d deleted successfully!\n", bookingID);
}
```

```
// Function to display all current bookings
void displayBookings(Booking* head) {
    if (head == NULL) {
        printf("No bookings available.\n");
        return;
    }

    Booking* temp = head;
    printf("Current bookings:\n");
    printf("BookingID | Customer Name | Facility | Date\n");
    while (temp != NULL) {
        printf("%d        | %s         | %s      | %s\n", temp->bookingID, temp->customerName, temp->facility,
temp->date);
        temp = temp->next;
    }
}

// Main function to interact with the user
int main() {
    Booking* bookingList = NULL;  // Head of the linked list
    int choice, bookingID;
    char customerName[100], facility[50], date[20];

    while (1) {
        printf("\nSports Facility Booking System\n");
        printf("1. Add new booking\n");
        printf("2. Delete a booking\n");
        printf("3. Display all bookings\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                // Add new booking
                printf("Enter booking ID: ");
                scanf("%d", &bookingID);
                printf("Enter customer name: ");
                scanf(" %[^\n]s", customerName);  // Using scanf to read string with spaces
                printf("Enter facility name: ");
                scanf(" %[^\n]s", facility);  // Using scanf to read string with spaces
                printf("Enter booking date (dd-mm-yyyy): ");
                scanf(" %[^\n]s", date);  // Using scanf to read string with spaces

                insertBooking(&bookingList, bookingID, customerName, facility, date);
                break;

            case 2:
                // Delete booking
                printf("Enter booking ID to delete: ");
                scanf("%d", &bookingID);
                deleteBooking(&bookingList, bookingID);
                break;
            case 3:
                // Display all bookings
                displayBookings(bookingList);
                break;

            case 4:
                // Exit
                printf("Exiting the system...\n");
                exit(0);
                break;

            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

Problem 11: Coaching Staff Management

Description: Use a linked list to manage the coaching staff of a sports team.Operations:

Create a coaching staff list.

Insert a new coach.

Delete a coach who leaves the team.

Display the current coaching staff.

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

// Define the structure for the coaching staff member
typedef struct Coach {
    char name[50];
    char position[50];
    struct Coach* next;
} Coach;

// Function to create a new coach node
Coach* createCoach(const char* name, const char* position) {
    Coach* newCoach = (Coach*)malloc(sizeof(Coach));
    if (newCoach == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    strcpy(newCoach->name, name);
    strcpy(newCoach->position, position);
    newCoach->next = NULL;
    return newCoach;
}

// Function to insert a new coach at the end of the list
void insertCoach(Coach** head, const char* name, const char* position) {
    Coach* newCoach = createCoach(name, position);
    if (*head == NULL) {
        *head = newCoach;
    } else {
        Coach* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newCoach;
    }
}

// Function to delete a coach by name
void deleteCoach(Coach** head, const char* name) {
    if (*head == NULL) {
        printf("The coaching staff list is empty.\n");
        return;
    }

    Coach* temp = *head;
    Coach* prev = NULL;

    // If the head coach needs to be deleted
    if (strcmp(temp->name, name) == 0) {
        *head = temp->next;
        free(temp);
        printf("Coach %s deleted from the team.\n", name);
        return;
    }

    // Search for the coach to be deleted
    while (temp != NULL && strcmp(temp->name, name) != 0) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Coach %s not found in the team.\n", name);
        return;
    }

    // Unlink the coach from the list
    prev->next = temp->next;
    free(temp);
    printf("Coach %s deleted from the team.\n", name);
}

// Function to display the current coaching staff
void displayCoachingStaff(Coach* head) {
    if (head == NULL) {
        printf("The coaching staff list is empty.\n");
        return;
    }

    Coach* temp = head;
    printf("Current Coaching Staff:\n");
    while (temp != NULL) {
        printf("Name: %s, Position: %s\n", temp->name, temp->position);
        temp = temp->next;
    }
}

// Main function
int main() {
    Coach* coachingStaff = NULL;

    // Insert coaches into the list
    insertCoach(&coachingStaff, "John Doe", "Head Coach");
    insertCoach(&coachingStaff, "Jane Smith", "Assistant Coach");
    insertCoach(&coachingStaff, "Mike Johnson", "Fitness Coach");

    // Display current coaching staff
    displayCoachingStaff(coachingStaff);

    // Delete a coach who leaves the team
    deleteCoach(&coachingStaff, "Jane Smith");

    // Display current coaching staff after deletion
    displayCoachingStaff(coachingStaff);

    // Clean up memory before exiting
    // Note: In a real program, we should free all allocated memory.
    return 0;
}
```

Problem 12: Fan Club Membership Management
Description: Implement a linked list to manage memberships in a sports team’s fan club.Operations:
Create a membership list.
Insert a new member.
Delete a member who cancels their membership.
Display all current members.

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

// Define a struct for the members
typedef struct Member {
    char name[100];           // Name of the member
    struct Member* next;      // Pointer to the next member
} Member;

// Function to create a new member node
Member* createMember(const char* name) {
    Member* newMember = (Member*)malloc(sizeof(Member));
    if (newMember == NULL) {
        printf("Memory allocation failed!\n");
        return NULL;
    }
    strcpy(newMember->name, name);
    newMember->next = NULL;
    return newMember;
}
```

```
// Function to insert a new member at the end of the list
void insertMember(Member** head, const char* name) {
    Member* newMember = createMember(name);
    if (*head == NULL) {
        *head = newMember;  // If the list is empty, new member becomes the head
    } else {
        Member* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;  // Traverse to the end of the list
        }
        temp->next = newMember;  // Add the new member at the end
    }
}

// Function to delete a member by name
void deleteMember(Member** head, const char* name) {
    if (*head == NULL) {
        printf("The list is empty!\n");
        return;
    }

    Member* temp = *head;
    Member* prev = NULL;

    // Check if the head node contains the member to be deleted
    if (strcmp(temp->name, name) == 0) {
        *head = temp->next;  // Change the head
        free(temp);          // Free the memory of the deleted node
        printf("Member %s removed from the club.\n", name);
        return;
    }

    // Search for the member to delete
    while (temp != NULL && strcmp(temp->name, name) != 0) {
        prev = temp;
        temp = temp->next;
    }

    // If the member was not found
    if (temp == NULL) {
        printf("Member %s not found.\n", name);
        return;
    }

    // Unlink the node from the list
    prev->next = temp->next;
    free(temp);
    printf("Member %s removed from the club.\n", name);
}

// Function to display all members in the list
void displayMembers(Member* head) {
    if (head == NULL) {
        printf("No members in the club.\n");
        return;
    }

    printf("Current members of the fan club:\n");
    Member* temp = head;
    while (temp != NULL) {
        printf("%s\n", temp->name);
        temp = temp->next;
    }
}

int main() {
    Member* head = NULL;  // Initialize the list as empty
    int choice;
    char name[100];

    do {
        printf("\nFan Club Membership Management\n");
        printf("1. Insert a new member\n");
        printf("2. Delete a member\n");
        printf("3. Display all current members\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        // Clear the input buffer (to prevent issues with trailing newlines)
        while (getchar() != ’\n’);  // Discard any remaining newline characters

        switch (choice) {
            case 1:
                printf("Enter the name of the new member: ");
                scanf("%99[^\n]", name);  // Read string with spaces
                insertMember(&head, name);
                break;

            case 2:
                printf("Enter the name of the member to delete: ");
                scanf("%99[^\n]", name);  // Read string with spaces
                deleteMember(&head, name);
                break;

            case 3:
                displayMembers(head);
                break;

            case 4:
                printf("Exiting program.\n");
                break;

            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 4);

    return 0;
}
```

Problem 13: Sports Event Scheduling
Description: Use a linked list to manage the schedule of sports events.Operations:
Create an event schedule.
Insert a new event.
Delete a completed or canceled event.
Display the current event schedule.

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

// Define the structure for a Sports Event
typedef struct Event {
    char name[100];
    char date[20];
    struct Event* next;
} Event;

// Function to create a new event node
Event* createEvent(char* name, char* date) {
    Event* newEvent = (Event*)malloc(sizeof(Event));
    if (newEvent == NULL) {
        printf("Memory allocation failed.\n");
        return NULL;
    }
    strcpy(newEvent->name, name);
    strcpy(newEvent->date, date);
    newEvent->next = NULL;
    return newEvent;
}

// Function to insert a new event at the end of the list
void insertEvent(Event** head, char* name, char* date) {
    Event* newEvent = createEvent(name, date);
    if (*head == NULL) {
        *head = newEvent;
    } else {
        Event* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newEvent;
    }
    printf("Event \"%s\" scheduled on %s.\n", name, date);
}

// Function to delete an event by name
void deleteEvent(Event** head, char* name) {
    if (*head == NULL) {
        printf("No events to delete.\n");
        return;
    }

    Event* temp = *head;
    Event* prev = NULL;

    // If the event to be deleted is the first event
    if (strcmp(temp->name, name) == 0) {
        *head = temp->next;
        free(temp);
        printf("Event \"%s\" deleted.\n", name);
        return;
    }

    // Search for the event to delete
    while (temp != NULL && strcmp(temp->name, name) != 0) {
        prev = temp;
        temp = temp->next;
    }

    // If event not found
    if (temp == NULL) {
        printf("Event \"%s\" not found.\n", name);
        return;
    }

    // Delete the event
    prev->next = temp->next;
    free(temp);
    printf("Event \"%s\" deleted.\n", name);
}

// Function to display the current event schedule
void displaySchedule(Event* head) {
    if (head == NULL) {
        printf("No events scheduled.\n");
        return;
    }

    printf("Current Event Schedule:\n");
    Event* temp = head;
    while (temp != NULL) {
        printf("Event: %s, Date: %s\n", temp->name, temp->date);
        temp = temp->next;
    }
}

// Main function to drive the program
int main() {
    Event* schedule = NULL;
    int choice;
    char name[100], date[20];

    while (1) {
        printf("\nSports Event Scheduler Menu:\n");
        printf("1. Insert a new event\n");
        printf("2. Delete an event\n");
        printf("3. Display the current event schedule\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        // Flush the newline character left by scanf
        while(getchar() != ’\n’);

        switch (choice) {
            case 1:
                printf("Enter the event name: ");
                scanf("%99[^\n]", name);  // Read event name (with spaces)
                while(getchar() != ’\n’); // Clear the input buffer
                printf("Enter the event date (YYYY-MM-DD): ");
                scanf("%19[^\n]", date);  // Read date (with spaces, if any)
                while(getchar() != ’\n’); // Clear the input buffer
                insertEvent(&schedule, name, date);
                break;

            case 2:
                printf("Enter the event name to delete: ");
                scanf("%99[^\n]", name);  // Read event name (with spaces)
                while(getchar() != ’\n’); // Clear the input buffer
                deleteEvent(&schedule, name);
                break;

            case 3:
                displaySchedule(schedule);
                break;

            case 4:
                printf("Exiting program.\n");
                // Free the allocated memory
                while (schedule != NULL) {
                    Event* temp = schedule;
                    schedule = schedule->next;
                    free(temp);
                }
                exit(0);

            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}
```

Problem 14: Player Transfer Records
Description: Maintain a linked list to track player transfers between teams.Operations:
Create a transfer record list.
Insert a new transfer record.
Delete an outdated or erroneous transfer record.
Display all current transfer records.

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

```c
// Define the structure for a transfer record
typedef struct TransferRecord {
    char player_name[50];
    char from_team[50];
    char to_team[50];
    int transfer_fee;  // Transfer fee in millions
    char transfer_date[20];  // Transfer date in the format YYYY-MM-DD
    struct TransferRecord* next;  // Pointer to the next record
} TransferRecord;

// Function to create a new transfer record
TransferRecord* create_record(const char* player_name, const char* from_team, const char* to_team, int transfer_fee, const char* transfer_date) {
    TransferRecord* new_record = (TransferRecord*)malloc(sizeof(TransferRecord));
    if (!new_record) {
        printf("Memory allocation failed!\n");
        exit(1);
    }

    // Copy the input values into the new record
    strcpy(new_record->player_name, player_name);
    strcpy(new_record->from_team, from_team);
    strcpy(new_record->to_team, to_team);
    new_record->transfer_fee = transfer_fee;
    strcpy(new_record->transfer_date, transfer_date);
    new_record->next = NULL;

    return new_record;
}

// Function to insert a new transfer record at the end of the list
void insert_transfer(TransferRecord** head, const char* player_name, const char* from_team, const char* to_team, int transfer_fee, const char* transfer_date) {
    TransferRecord* new_record = create_record(player_name, from_team, to_team, transfer_fee, transfer_date);

    // If the list is empty, the new record becomes the head
    if (*head == NULL) {
        *head = new_record;
    } else {
        TransferRecord* temp = *head;
        // Traverse to the end of the list
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = new_record;  // Insert the new record at the end
    }
}

// Function to delete a transfer record based on player name, from team, and to team
int delete_transfer(TransferRecord** head, const char* player_name, const char* from_team, const char* to_team) {
    if (*head == NULL) {
        return 0;  // List is empty, no record to delete
    }

    TransferRecord* temp = *head;
    TransferRecord* prev = NULL;

    // Traverse the list to find the matching record
    while (temp != NULL) {
        if (strcmp(temp->player_name, player_name) == 0 &&
            strcmp(temp->from_team, from_team) == 0 &&
            strcmp(temp->to_team, to_team) == 0) {

            // If it’s the first record (head node)
            if (prev == NULL) {
                *head = temp->next;
            } else {
                prev->next = temp->next;
            }

            free(temp);  // Free the memory of the deleted record
            return 1;  // Record deleted successfully
        }
        prev = temp;
        temp = temp->next;
    }

    return 0;  // Record not found
}

// Function to display all transfer records
void display_transfers(TransferRecord* head) {
    if (head == NULL) {
        printf("No transfer records available.\n");
        return;
    }

    TransferRecord* temp = head;
    while (temp != NULL) {
        printf("Player: %s, From: %s, To: %s, Transfer Fee: %d, Date: %s\n",
               temp->player_name, temp->from_team, temp->to_team, temp->transfer_fee,
temp->transfer_date);
        temp = temp->next;
    }
}

int main() {
    TransferRecord* transfer_list = NULL;  // Start with an empty list

    // Inserting transfer records
    insert_transfer(&transfer_list, "Player1", "TeamA", "TeamB", 50, "2025-01-10");
    insert_transfer(&transfer_list, "Player2", "TeamC", "TeamD", 30, "2025-01-12");
    insert_transfer(&transfer_list, "Player3", "TeamB", "TeamA", 25, "2025-01-14");

    // Displaying all transfer records
    printf("All Transfer Records:\n");
    display_transfers(transfer_list);
    // Deleting a specific transfer record
    if (delete_transfer(&transfer_list, "Player2", "TeamC", "TeamD")) {
        printf("\nRecord for Player2 deleted successfully.\n");
    } else {
        printf("\nRecord for Player2 not found.\n");
    }

    // Displaying updated transfer records
    printf("\nUpdated Transfer Records:\n");
    display_transfers(transfer_list);

    // Freeing the allocated memory
    TransferRecord* temp;
    while (transfer_list != NULL) {
        temp = transfer_list;
        transfer_list = transfer_list->next;
        free(temp);
    }

    return 0;
}
```

Problem 15: Championship Points Tracker
Description: Implement a linked list to track championship points for teams.Operations:
Create a points tracker list.
Insert a new points entry.
Delete an incorrect or outdated points entry.
Display all current points standings.

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

// Define the structure for the linked list
typedef struct Team {
    char name[50];
    int points;
    struct Team* next;
} Team;

// Function to create a new team node
Team* createTeam(char* name, int points) {
    Team* newTeam = (Team*)malloc(sizeof(Team));
    if (newTeam == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    strcpy(newTeam->name, name);
    newTeam->points = points;
    newTeam->next = NULL;
    return newTeam;
}

// Function to insert a new team with their points
```

```
void insertPoints(Team** head, char* name, int points) {
    Team* newTeam = createTeam(name, points);
    newTeam->next = *head;
    *head = newTeam;
    printf("Inserted %s with %d points.\n", name, points);
}

// Function to delete a team based on name
void deletePoints(Team** head, char* name) {
    Team* temp = *head;
    Team* prev = NULL;

    // If the head itself holds the team to be deleted
    if (temp != NULL && strcmp(temp->name, name) == 0) {
        *head = temp->next;
        free(temp);
        printf("Deleted %s from the list.\n", name);
        return;
    }

    // Search for the team to be deleted
    while (temp != NULL && strcmp(temp->name, name) != 0) {
        prev = temp;
        temp = temp->next;
    }

    // If the team was not found
    if (temp == NULL) {
        printf("Team %s not found.\n", name);
        return;
    }

    prev->next = temp->next;
    free(temp);
    printf("Deleted %s from the list.\n", name);
}

// Function to display all the standings
void displayStandings(Team* head) {
    if (head == NULL) {
        printf("No teams to display.\n");
        return;
    }

    printf("\nCurrent Points Standings:\n");
    Team* temp = head;
    while (temp != NULL) {
        printf("Team: %s, Points: %d\n", temp->name, temp->points);
        temp = temp->next;
    }
}

// Main function to demonstrate operations
int main() {
    Team* head = NULL;
    int choice;
    char teamName[50];
    int points;

    while (1) {
        printf("\nChampionship Points Tracker\n");
        printf("1. Insert Team Points\n");
        printf("2. Delete Team\n");
        printf("3. Display Standings\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter team name: ");
                scanf("%s", teamName);
                printf("Enter points: ");
                scanf("%d", &points);
                insertPoints(&head, teamName, points);
                break;
            case 2:
                printf("Enter team name to delete: ");
                scanf("%s", teamName);
                deletePoints(&head, teamName);
                break;
            case 3:
                displayStandings(head);
                break;
            case 4:
                printf("Exiting...\n");
                exit(0);
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```