Lab Experiment: 03

Student Detail:

- **Name:** Prashant Joshi
- **Student ID:** 590010879
- **Branch:** MCA
- **Batch:** B1
- **Instructor:** Dr. Sourbh Kumar

1. **Singly Linked List Implementation:**
   - Create a structure for a singly linked list node with data and a next pointer.
   - Implement functions for:
   - Insertion at the beginning, end, and a specified position.
   - Deletion from the beginning, end, and a specified position.
   - Displaying the list.

Solution:

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

// Structure for singly linked list node
struct Node {
    int data;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to insert at the beginning
void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    newNode->next = *head;
    *head = newNode;
```

```
}

// Function to insert at the end
void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}

// Function to insert at a specified position
void insertAtPosition(struct Node** head, int data, int position) {
    struct Node* newNode = createNode(data);
    if (position == 0) {
        insertAtBeginning(head, data);
        return;
    }
    struct Node* temp = *head;
    for (int i = 0; temp != NULL && i < position - 1; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Position out of bounds\n");
        return;
    }
    newNode->next = temp->next;
    temp->next = newNode;
```

```
}

// Function to delete from the beginning
void deleteFromBeginning(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty\n");
        return;
    }
    struct Node* temp = *head;
    *head = (*head)->next;
    free(temp);
}

// Function to delete from the end
void deleteFromEnd(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty\n");
        return;
    }
    if ((*head)->next == NULL) {
        free(*head);
        *head = NULL;
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL && temp->next->next != NULL) {
        temp = temp->next;
    }
    free(temp->next);
    temp->next = NULL;
}

// Function to delete from a specified position
```

```
void deleteFromPosition(struct Node** head, int position) {
    if (*head == NULL) {
        printf("List is empty\n");
        return;
    }
    if (position == 0) {
        deleteFromBeginning(head);
        return;
    }
    struct Node* temp = *head;
    for (int i = 0; temp != NULL && i < position - 1; i++) {
        temp = temp->next;
    }
    if (temp == NULL || temp->next == NULL) {
        printf("Position out of bounds\n");
        return;
    }
    struct Node* nodeToDelete = temp->next;
    temp->next = temp->next->next;
    free(nodeToDelete);
}

// Function to display the list
void displayList(struct Node* head) {
    struct Node* temp = head;
    if (temp == NULL) {
        printf("List is empty\n");
        return;
    }
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
```

```
    printf("NULL\n");
}

int main() {
    struct Node* head = NULL;

    // Insertion at the beginning
    insertAtBeginning(&head, 10); // 10 -> NULL
    insertAtBeginning(&head, 20); // 20 -> 10 -> NULL

    // Insertion at the end
    insertAtEnd(&head, 30); // 20 -> 10 -> 30 -> NULL

    // Insertion at a specified position (Position 1)
    insertAtPosition(&head, 25, 1); // 20 -> 25 -> 10 -> 30 -> NULL

    // Displaying the list
    printf("List after insertions:\n");
    displayList(head); // Expected: 20 -> 25 -> 10 -> 30 -> NULL

    // Deletion from the beginning
    deleteFromBeginning(&head); // 25 -> 10 -> 30 -> NULL
    printf("List after deletion from beginning:\n");
    displayList(head); // Expected: 25 -> 10 -> 30 -> NULL

    // Deletion from the end
    deleteFromEnd(&head); // 25 -> 10 -> NULL
    printf("List after deletion from end:\n");
    displayList(head); // Expected: 25 -> 10 -> NULL

    // Deletion from a specified position (Position 1)
    deleteFromPosition(&head, 1); // 25 -> NULL
    printf("List after deletion from position 1:\n");
```

```
    displayList(head); // Expected: 25 -> NULL


    return 0;
}
```

Output:

```
List after insertions:
20 -> 25 -> 10 -> 30 -> NULL
List after deletion from beginning:
25 -> 10 -> 30 -> NULL
List after deletion from end:
25 -> 10 -> NULL
List after deletion from position 1:
25 -> NULL
```

2. Doubly Linked List Implementation:

- Modify the singly linked list to a doubly linked list by adding a prev pointer.
- Implement the same insertion, deletion, and display functions.

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


// Structure for doubly linked list node
struct DNode {
    int data;
    struct DNode* next;
    struct DNode* prev;
};


// Function to create a new doubly linked list node
struct DNode* createDNode(int data) {
    struct DNode* newNode = (struct DNode*)malloc(sizeof(struct DNode));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
```

```
    }
    newNode->data = data;
    newNode->next = NULL;
    newNode->prev = NULL;
    return newNode;
}

// Doubly Linked List Operations
void insertDAtBeginning(struct DNode** head, int data) {
    struct DNode* newNode = createDNode(data);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    newNode->next = *head;
    (*head)->prev = newNode;
    *head = newNode;
}

void insertDAtEnd(struct DNode** head, int data) {
    struct DNode* newNode = createDNode(data);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct DNode* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->prev = temp;
}
```

```
void insertDAtPosition(struct DNode** head, int data, int position) {
    struct DNode* newNode = createDNode(data);
    if (position == 0) {
        insertDAtBeginning(head, data);
        return;
    }
    struct DNode* temp = *head;
    for (int i = 0; temp != NULL && i < position - 1; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Position out of bounds\n");
        return;
    }
    newNode->next = temp->next;
    if (temp->next != NULL) {
        temp->next->prev = newNode;
    }
    temp->next = newNode;
    newNode->prev = temp;
}

void deleteDFromBeginning(struct DNode** head) {
    if (*head == NULL) {
        printf("List is empty\n");
        return;
    }
    struct DNode* temp = *head;
    *head = (*head)->next;
    if (*head != NULL) {
        (*head)->prev = NULL;
    }
    free(temp);
```

```
}

void deleteDFromEnd(struct DNode** head) {
    if (*head == NULL) {
        printf("List is empty\n");
        return;
    }
    if ((*head)->next == NULL) {
        free(*head);
        *head = NULL;
        return;
    }
    struct DNode* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->prev->next = NULL;
    free(temp);
}

void deleteDFromPosition(struct DNode** head, int position) {
    if (*head == NULL) {
        printf("List is empty\n");
        return;
    }
    if (position == 0) {
        deleteDFromBeginning(head);
        return;
    }
    struct DNode* temp = *head;
    for (int i = 0; temp != NULL && i < position - 1; i++) {
        temp = temp->next;
    }
```

```
    if (temp == NULL || temp->next == NULL) {
        printf("Position out of bounds\n");
        return;
    }
    struct DNode* nodeToDelete = temp->next;
    temp->next = nodeToDelete->next;
    if (nodeToDelete->next != NULL) {
        nodeToDelete->next->prev = temp;
    }
    free(nodeToDelete);
}

void displayDList(struct DNode* head) {
    struct DNode* temp = head;
    if (temp == NULL) {
        printf("List is empty\n");
        return;
    }
    while (temp != NULL) {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    struct DNode* head = NULL;

    // Insertion at the beginning
    insertDAtBeginning(&head, 10); // 10 <-> NULL
    insertDAtBeginning(&head, 20); // 20 <-> 10 <-> NULL

    // Insertion at the end
```

```
    insertDAtEnd(&head, 30); // 20 <-> 10 <-> 30 <-> NULL

    // Insertion at a specified position (Position 1)
    insertDAtPosition(&head, 25, 1); // 20 <-> 25 <-> 10 <-> 30 <-> NULL

    // Displaying the list
    printf("List after insertions:\n");
    displayDList(head); // Expected: 20 <-> 25 <-> 10 <-> 30 <-> NULL

    // Deletion from the beginning
    deleteDFromBeginning(&head); // 25 <-> 10 <-> 30 <-> NULL
    printf("List after deletion from beginning:\n");
    displayDList(head); // Expected: 25 <-> 10 <-> 30 <-> NULL

    // Deletion from the end
    deleteDFromEnd(&head); // 25 <-> 10 <-> NULL
    printf("List after deletion from end:\n");
    displayDList(head); // Expected: 25 <-> 10 <-> NULL
    // Deletion from a specified position (Position 1)
    deleteDFromPosition(&head, 1); // 25 <-> NULL
    printf("List after deletion from position 1:\n");
    displayDList(head); // Expected: 25 <-> NULL
    return 0;
}
```

Output:

```
List after insertions:
20 <-> 25 <-> 10 <-> 30 <-> NULL
List after deletion from beginning:
25 <-> 10 <-> 30 <-> NULL
List after deletion from end:
25 <-> 10 <-> NULL
List after deletion from position 1:
25 <-> NULL
```

3. Application Example:

- Demonstrate an application of linked lists, such as managing a to-do list or implementing a simple stack/queue.

Solution:

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

// Structure for a task node in a to-do list (singly linked list)
struct Task {
    char description[100];
    struct Task* next;
};

// Function to create a new task node
struct Task* createTask(const char* description) {
    struct Task* newTask = (struct Task*)malloc(sizeof(struct Task));
    if (newTask == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    strcpy(newTask->description, description);
    newTask->next = NULL;
    return newTask;
}

// Function to add a task at the end of the list
void addTask(struct Task** head, const char* description) {
    struct Task* newTask = createTask(description);
    if (*head == NULL) {
        *head = newTask;
    } else {
```

```
        struct Task* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newTask;
    }
}

// Function to remove a task from the beginning
void removeTaskFromBeginning(struct Task** head) {
    if (*head == NULL) {
        printf("No tasks to remove!\n");
        return;
    }
    struct Task* temp = *head;
    *head = (*head)->next;
    printf("Removed: %s\n", temp->description);
    free(temp);
}

// Function to remove a task from the end
void removeTaskFromEnd(struct Task** head) {
    if (*head == NULL) {
        printf("No tasks to remove!\n");
        return;
    }
    if ((*head)->next == NULL) {
        printf("Removed: %s\n", (*head)->description);
        free(*head);
        *head = NULL;
        return;
    }
    struct Task* temp = *head;
```

```
    while (temp->next != NULL && temp->next->next != NULL) {
        temp = temp->next;
    }
    printf("Removed: %s\n", temp->next->description);
    free(temp->next);
    temp->next = NULL;
}

// Function to remove a task from a specific position
void removeTaskFromPosition(struct Task** head, int position) {
    if (*head == NULL) {
        printf("No tasks to remove!\n");
        return;
    }
    if (position == 0) {
        removeTaskFromBeginning(head);
        return;
    }
    struct Task* temp = *head;
    for (int i = 0; temp != NULL && i < position - 1; i++) {
        temp = temp->next;
    }
    if (temp == NULL || temp->next == NULL) {
        printf("Invalid position!\n");
        return;
    }
    struct Task* taskToDelete = temp->next;
    temp->next = temp->next->next;
    printf("Removed: %s\n", taskToDelete->description);
    free(taskToDelete);
}

// Function to display the to-do list
```

```
void displayToDoList(struct Task* head) {
    if (head == NULL) {
        printf("The to-do list is empty.\n");
        return;
    }
    printf("To-Do List:\n");
    struct Task* temp = head;
    while (temp != NULL) {
        printf("- %s\n", temp->description);
        temp = temp->next;
    }
}

int main() {
    struct Task* head = NULL; // Initialize an empty to-do list

    // Adding tasks to the to-do list
    addTask(&head, "Complete project report");
    addTask(&head, "Attend team meeting");
    addTask(&head, "Buy groceries");
    addTask(&head, "Clean the house");

    // Displaying the to-do list
    displayToDoList(head);

    // Removing a task from the beginning
    removeTaskFromBeginning(&head); // Removes "Complete project report"
    displayToDoList(head);

    // Removing a task from the end
    removeTaskFromEnd(&head); // Removes "Clean the house"
    displayToDoList(head);
```

```
    // Removing a task from a specific position
    removeTaskFromPosition(&head, 1); // Removes "Attend team meeting"
    displayToDoList(head);

    return 0;
}
```

Output:

```
To-Do List:
- Complete project report
- Attend team meeting
- Buy groceries
- Clean the house

Removed: Complete project report
To-Do List:
- Attend team meeting
- Buy groceries
- Clean the house

Removed: Clean the house
To-Do List:
- Attend team meeting
- Buy groceries

Removed: Attend team meeting
To-Do List:
- Buy groceries
```

Memory Usage and Dynamic Allocation:

- Use malloc and free to dynamically allocate and deallocate memory.
- Ensure memory is correctly freed after operations to prevent memory leaks.

Solution:

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

// Structure for a task node in a to-do list (singly linked list)
struct Task {
    char description[100];
    struct Task* next;
};

// Function to create a new task node
struct Task* createTask(const char* description) {
    struct Task* newTask = (struct Task*)malloc(sizeof(struct Task));
    if (newTask == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    strcpy(newTask->description, description);
    newTask->next = NULL;
    return newTask;
}

// Function to add a task at the end of the list
void addTask(struct Task** head, const char* description) {
    struct Task* newTask = createTask(description);
    if (*head == NULL) {
        *head = newTask;
    } else {
```

```
        struct Task* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newTask;
    }
}

// Function to remove a task from the beginning
void removeTaskFromBeginning(struct Task** head) {
    if (*head == NULL) {
        printf("No tasks to remove!\n");
        return;
    }
    struct Task* temp = *head;
    *head = (*head)->next;
    printf("Removed: %s\n", temp->description);
    free(temp);  // Freeing memory for the removed task
}

// Function to remove a task from the end
void removeTaskFromEnd(struct Task** head) {
    if (*head == NULL) {
        printf("No tasks to remove!\n");
        return;
    }
    if ((*head)->next == NULL) {
        printf("Removed: %s\n", (*head)->description);
        free(*head);  // Freeing the only node in the list
        *head = NULL;
        return;
    }
    struct Task* temp = *head;
```

```
    while (temp->next != NULL) {
        temp = temp->next;
    }
    printf("Removed: %s\n", temp->description);
    free(temp);  // Freeing the last node
}

// Function to remove a task from a specific position
void removeTaskFromPosition(struct Task** head, int position) {
    if (*head == NULL) {
        printf("No tasks to remove!\n");
        return;
    }
    if (position == 0) {
        removeTaskFromBeginning(head);
        return;
    }
    struct Task* temp = *head;
    for (int i = 0; temp != NULL && i < position - 1; i++) {
        temp = temp->next;
    }
    if (temp == NULL || temp->next == NULL) {
        printf("Invalid position!\n");
        return;
    }
    struct Task* taskToDelete = temp->next;
    temp->next = temp->next->next;
    printf("Removed: %s\n", taskToDelete->description);
    free(taskToDelete);  // Freeing the task at the specified position
}

// Function to display the to-do list
void displayToDoList(struct Task* head) {
```

```
    if (head == NULL) {
        printf("The to-do list is empty.\n");
        return;
    }
    printf("To-Do List:\n");
    struct Task* temp = head;
    while (temp != NULL) {
        printf("- %s\n", temp->description);
        temp = temp->next;
    }
}

// Function to free all tasks in the list (to ensure no memory leak)
void freeList(struct Task* head) {
    struct Task* temp;
    while (head != NULL) {
        temp = head;
        head = head->next;
        free(temp);  // Free each node to avoid memory leaks
    }
}

int main() {
    struct Task* head = NULL; // Initialize an empty to-do list

    // Adding tasks to the to-do list
    addTask(&head, "Complete project report");
    addTask(&head, "Attend team meeting");
    addTask(&head, "Buy groceries");
    addTask(&head, "Clean the house");

    // Displaying the to-do list
    displayToDoList(head);
```

```
    // Removing a task from the beginning
    removeTaskFromBeginning(&head); // Removes "Complete project report"
    displayToDoList(head);

    // Removing a task from the end
    removeTaskFromEnd(&head); // Removes "Clean the house"
    displayToDoList(head);

    // Removing a task from a specific position
    removeTaskFromPosition(&head, 1); // Removes "Attend team meeting"
    displayToDoList(head);

    // Freeing all remaining tasks to prevent memory leaks
    freeList(head);

    return 0;
}
```