Singly Linked List Implementation

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

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

// Function to insert at the beginning
Node* insertAtBeginning(Node* head, int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = head;
    return newNode;
}

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

// Function to insert at a specified position
Node* insertAtPosition(Node* head, int data, int position) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    if (position == 1) {
        newNode->next = head;
        return newNode;
    }
    Node* temp = head;
    for (int i = 1; i < position - 1 && temp != NULL; i++) {
        temp = temp->next;
```

```
        temp = temp->next;
    }
    if (temp == NULL) {
        free(newNode);
        printf("Position out of bounds.\n");
        return head;
    }
    newNode->next = temp->next;
    temp->next = newNode;
    return head;
}

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

// Function to delete from the end
Node* deleteFromEnd(Node* head) {
    if (head == NULL || head->next == NULL) {
        free(head);
        return NULL;
    }
    Node* temp = head;
    while (temp->next->next != NULL) {
        temp = temp->next;
    }
    free(temp->next);
    temp->next = NULL;
    return head;
}

// Function to delete from a specified position
Node* deleteFromPosition(Node* head, int position) {
    if (head == NULL) {
        printf("List is empty.\n");
        return NULL;
```

```c
        return NULL;
    }
    if (position == 1) {
        return deleteFromBeginning(head);
    }
    Node* temp = head;
    for (int i = 1; i < position - 1 && temp->next != NULL; i++) {
        temp = temp->next;
    }
    if (temp->next == NULL) {
        printf("Position out of bounds.\n");
        return head;
    }
    Node* nodeToDelete = temp->next;
    temp->next = nodeToDelete->next;
    free(nodeToDelete);
    return head;
}

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

// Function to free the list and prevent memory leaks
void freeList(Node* head) {
    Node* temp;
    while (head != NULL) {
        temp = head;
        head = head->next;
        free(temp);
    }
}

int main() {
    Node* head = NULL;

    // Sample operations
    head = insertAtBeginning(head, 10);
```

```c
    Node* nodeToDelete = temp->next;
    temp->next = nodeToDelete->next;
    free(nodeToDelete);
    return head;
}

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

// Function to free the list and prevent memory leaks
void freeList(Node* head) {
    Node* temp;
    while (head != NULL) {
        temp = head;
        head = head->next;
        free(temp);
    }
}

int main() {
    Node* head = NULL;

    // Sample operations
    head = insertAtBeginning(head, 10);
    head = insertAtEnd(head, 20);
    head = insertAtPosition(head, 15, 2);
    displayList(head);

    head = deleteFromPosition(head, 2);
    displayList(head);

    freeList(head);
    return 0;
}
```

Double Linked List Implementation

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

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

// Function to insert at the beginning
DNode* insertAtBeginning(DNode* head, int data) {
    DNode* newNode = (DNode*)malloc(sizeof(DNode));
    newNode->data = data;
    newNode->next = head;
    newNode->prev = NULL;
    if (head != NULL) {
        head->prev = newNode;
    }
    return newNode;
}

// Function to insert at the end
DNode* insertAtEnd(DNode* head, int data) {
    DNode* newNode = (DNode*)malloc(sizeof(DNode));
    newNode->data = data;
    newNode->next = NULL;
    if (head == NULL) {
        newNode->prev = NULL;
        return newNode;
    }
    DNode* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->prev = temp;
    return head;
}

// Function to insert at a specified position
DNode* insertAtPosition(DNode* head, int data, int position) {
    DNode* newNode = (DNode*)malloc(sizeof(DNode));
    newNode->data = data;
```

```c
    newNode->data = data;
    if (position == 1) {
        newNode->next = head;
        newNode->prev = NULL;
        if (head != NULL) {
            head->prev = newNode;
        }
        return newNode;
    }
    DNode* temp = head;
    for (int i = 1; i < position - 1 && temp != NULL; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {
        free(newNode);
        printf("Position out of bounds.\n");
        return head;
    }
    newNode->next = temp->next;
    if (temp->next != NULL) {
        temp->next->prev = newNode;
    }
    temp->next = newNode;
    newNode->prev = temp;
    return head;
}

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

// Function to delete from the end
DNode* deleteFromEnd(DNode* head) {
```

```c
DNode* deleteFromEnd(DNode* head) {
    if (head == NULL || head->next == NULL) {
        free(head);
        return NULL;
    }
    DNode* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->prev->next = NULL;
    free(temp);
    return head;
}

// Function to delete from a specified position
DNode* deleteFromPosition(DNode* head, int position) {
    if (head == NULL) {
        printf("List is empty.\n");
        return NULL;
    }
    if (position == 1) {
        return deleteFromBeginning(head);
    }
    DNode* temp = head;
    for (int i = 1; i < position && temp != NULL; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Position out of bounds.\n");
        return head;
    }
    if (temp->next != NULL) {
        temp->next->prev = temp->prev;
    }
    if (temp->prev != NULL) {
        temp->prev->next = temp->next;
    }
    free(temp);
    return head;
}

// Function to display the list
void displayDList(DNode* head) {
    DNode* temp = head;
```

```c
    if (temp->prev != NULL) {
        temp->prev->next = temp->next;
    }
    free(temp);
    return head;
}

// Function to display the list
void displayDList(DNode* head) {
    DNode* temp = head;
    while (temp != NULL) {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Function to free the list and prevent memory leaks
void freeDList(DNode* head) {
    DNode* temp;
    while (head != NULL) {
        temp = head;
        head = head->next;
        free(temp);
    }
}

int main() {
    DNode* head = NULL;

    // Sample operations
    head = insertAtBeginning(head, 30);
    head = insertAtEnd(head, 40);
    head = insertAtPosition(head, 35, 2);
    displayDList(head);

    head = deleteFromPosition(head, 2);
    displayDList(head);

    freeDList(head);
    return 0;
}
```

To-Do List Application

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

// Structure for a to-do list node
typedef struct Task {
    char task[100];
    struct Task* next;
} Task;

// Function to add a task at the end of the list
Task* addTask(Task* head, const char* taskDescription) {
    Task* newTask = (Task*)malloc(sizeof(Task));
    if (newTask == NULL) {
        printf("Memory allocation failed.\n");
        return head;
    }
    strcpy(newTask->task, taskDescription);
    newTask->next = NULL;

    if (head == NULL) {
        return newTask;
    }

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

// Function to remove the first task from the list (mark as completed)
Task* removeTask(Task* head) {
    if (head == NULL) {
        printf("No tasks to remove.\n");
        return NULL;
    }
    Task* temp = head;
    head = head->next;
    printf("Task completed: %s\n", temp->task);
    free(temp);
    return head;
}
```

```c
// Function to display all tasks in the to-do list
void displayTasks(Task* head) {
    if (head == NULL) {
        printf("No tasks in the to-do list.\n");
        return;
    }
    printf("To-Do List:\n");
    Task* temp = head;
    while (temp != NULL) {
        printf("- %s\n", temp->task);
        temp = temp->next;
    }
}

// Function to free all memory and prevent memory leaks
void freeTasks(Task* head) {
    Task* temp;
    while (head != NULL) {
        temp = head;
        head = head->next;
        free(temp);
    }
}

int main() {
    Task* toDoList = NULL;
    int choice;
    char taskDescription[100];

    while (1) {
        printf("\nTo-Do List Menu:\n");
        printf("1. Add Task\n");
        printf("2. Remove First Task\n");
        printf("3. Display Tasks\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        getchar(); // Clear newline character from buffer

        switch (choice) {
            case 1:
                printf("Enter task description: ");
                fgets(taskDescription, sizeof(taskDescription), stdin);
                taskDescription[strcspn(taskDescription, "\n")] = '\0'; // Remove newline
```

```
                fgets(taskDescription, sizeof(taskDescription), stdin);
                taskDescription[strcspn(taskDescription, "\n")] = '\0'; // Remove newline
                    character
                toDoList = addTask(toDoList, taskDescription);
                break;
            case 2:
                toDoList = removeTask(toDoList);
                break;
            case 3:
                displayTasks(toDoList);
                break;
            case 4:
                freeTasks(toDoList);
                printf("Exiting. All tasks have been cleared.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```