**Exp No. 4**

## Title:

A) Implement Circular Linked List ADT | B) Implement Doubly Linked List ADT | C) Add Two Polynomials using Linked List

## Aim:

To implement (A) Circular Linked List, (B) Doubly Linked List, and (C) Polynomial Addition using Linked Lists in C language.

## Objectives:

• Understand pointer-based dynamic memory and node manipulation.  
• Implement insertion, deletion, traversal for Circular and Doubly Linked Lists.  
• Represent polynomials as linked lists and perform addition.

## Theory:

A Linked List is a dynamic data structure where elements (nodes) contain data and pointers to the next/previous nodes.  
• Circular Linked List (CLL): The last node points back to the first node forming a circle; traversal can start at any node.  
• Doubly Linked List (DLL): Each node has pointers to both previous and next nodes; supports bidirectional traversal.  
• Polynomial as Linked List: Each term is a node (coefficient, exponent). Polynomials are usually kept sorted by exponent.

## Algorithms (High Level):

A) Circular Linked List

1. Maintain a tail pointer (tail->next gives head).  
2. Insert at end: new->next = tail->next; tail->next = new; tail = new.  
3. Insert at beginning: new->next = tail->next; tail->next = new.  
4. Delete by value: traverse until node->data == key; adjust previous->next; if deleting tail, move tail to previous.  
5. Traverse: start from head = tail->next and continue until you reach tail again.

B) Doubly Linked List

1. Maintain head pointer.  
2. Insert front: new->next=head; if head!=NULL head->prev=new; head=new.  
3. Insert end: traverse to last, link new after last.  
4. Delete by value/front/end: update both next and prev pointers.  
5. Display forward/backward: use next/prev pointers respectively.

C) Polynomial Addition

1. Represent each term as (coeff, expo) node in descending order of exponent.  
2. Traverse both lists:  
 - If exponents equal: sum coefficients; if sum!=0, add to result.  
 - Else append the term with higher exponent.  
3. Append remaining terms from either list.

## Program Code A (C Language): Circular Linked List ADT

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// Node structure for Circular Linked List

typedef struct Node {

    int data;

    struct Node \*next;

} Node;

// Global pointer to the last node (tail)

Node \*tail = NULL;

// Function to check if the list is empty

bool is\_empty\_cll() {

    return (tail == NULL);

}

// Function to insert a node at the beginning of the CLL (O(1))

void insert\_begin(int data) {

    Node \*new\_node = (Node \*)malloc(sizeof(Node));

    if (new\_node == NULL) {

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

        return;

    }

    new\_node->data = data;

    if (is\_empty\_cll()) {

        tail = new\_node;

        new\_node->next = tail; // Point to itself

    } else {

        new\_node->next = tail->next; // New node points to the current head

        tail->next = new\_node;        // Tail points to the new node (new head)

    }

    printf("Inserted %d at beginning.\n", data);

}

// Function to insert a node at the end of the CLL (O(1))

void insert\_end(int data) {

    Node \*new\_node = (Node \*)malloc(sizeof(Node));

    if (new\_node == NULL) {

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

        return;

    }

    new\_node->data = data;

    if (is\_empty\_cll()) {

        tail = new\_node;

        new\_node->next = tail; // Point to itself

    } else {

        new\_node->next = tail->next; // New node points to the head

        tail->next = new\_node;        // Tail points to the new node

        tail = new\_node;              // Update tail to the new node

    }

    printf("Inserted %d at end.\n", data);

}

// Function to delete a node from the beginning (O(1))

void delete\_begin() {

    if (is\_empty\_cll()) {

        printf("Deletion failed: Circular Linked List is Empty.\n");

        return;

    }

    Node \*head = tail->next;

    printf("Deleted %d from beginning.\n", head->data);

    if (head == tail) {

        // Only one node in the list

        free(head);

        tail = NULL;

    } else {

        tail->next = head->next; // Tail bypasses head, points to the new head

        free(head);

    }

}

// Function to display the Circular Linked List

void display\_cll() {

    if (is\_empty\_cll()) {

        printf("Circular Linked List: Empty\n");

        return;

    }

    printf("Circular Linked List: ");

    Node \*current = tail->next; // Start from the head

    do {

        printf("%d -> ", current->data);

        current = current->next;

    } while (current != tail->next);

    // Replace the last " -> " for cleaner output, showing the circular link

    printf("(Head)\n");

}

// Function to free the memory used by the CLL

void free\_cll() {

    if (is\_empty\_cll()) return;

    Node \*head = tail->next;

    Node \*current = head;

    Node \*temp = NULL;

    // Break the circular link temporarily

    tail->next = NULL;

    while (current != NULL) {

        temp = current;

        current = current->next;

        free(temp);

    }

    tail = NULL;

}

int main() {

    printf("--- Circular Linked List ADT ---\n");

    insert\_begin(10); // List: 10

    insert\_end(30);   // List: 10 -> 30

    insert\_begin(5);  // List: 5 -> 10 -> 30

    insert\_end(40);   // List: 5 -> 10 -> 30 -> 40

    display\_cll(); // Output: 5 -> 10 -> 30 -> 40 -> (Head)

    delete\_begin(); // Deletes 5. List: 10 -> 30 -> 40

    delete\_begin(); // Deletes 10. List: 30 -> 40

    display\_cll(); // Output: 30 -> 40 -> (Head)

    insert\_end(50); // List: 30 -> 40 -> 50

    display\_cll(); // Output: 30 -> 40 -> 50 -> (Head)

    // Clean up memory

    free\_cll();

    return 0;

}

## Program Code B (C Language): Doubly Linked List ADT

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// Node structure for Doubly Linked List

typedef struct DNode {

    int data;

    struct DNode \*prev;

    struct DNode \*next;

} DNode;

// Global pointer to the head of the DLL

DNode \*head = NULL;

// Function to check if the list is empty

bool is\_empty\_dll() {

    return (head == NULL);

}

// Function to insert a node at the front (O(1))

void insert\_front\_dll(int data) {

    DNode \*new\_node = (DNode \*)malloc(sizeof(DNode));

    if (new\_node == NULL) {

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

        return;

    }

    new\_node->data = data;

    new\_node->prev = NULL;

    new\_node->next = head;

    if (head != NULL) {

        head->prev = new\_node;

    }

    head = new\_node;

    printf("Inserted %d at front.\n", data);

}

// Function to insert a node at the end (O(n))

void insert\_end\_dll(int data) {

    DNode \*new\_node = (DNode \*)malloc(sizeof(DNode));

    if (new\_node == NULL) {

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

        return;

    }

    new\_node->data = data;

    new\_node->next = NULL;

    if (is\_empty\_dll()) {

        new\_node->prev = NULL;

        head = new\_node;

    } else {

        DNode \*temp = head;

        while (temp->next != NULL) {

            temp = temp->next;

        }

        temp->next = new\_node;

        new\_node->prev = temp;

    }

    printf("Inserted %d at end.\n", data);

}

// Function to delete the node at the front (O(1))

void delete\_front\_dll() {

    if (is\_empty\_dll()) {

        printf("Deletion failed: Doubly Linked List is Empty.\n");

        return;

    }

    DNode \*temp = head;

    head = head->next;

    if (head != NULL) {

        head->prev = NULL;

    }

    printf("Deleted %d from front.\n", temp->data);

    free(temp);

}

// Function to delete the node at the end (O(n))

void delete\_end\_dll() {

    if (is\_empty\_dll()) {

        printf("Deletion failed: Doubly Linked List is Empty.\n");

        return;

    }

    DNode \*temp = head;

    if (temp->next == NULL) {

        // Only one node

        head = NULL;

    } else {

        // Traverse to the last node

        while (temp->next != NULL) {

            temp = temp->next;

        }

        // Link the second-to-last node's next to NULL

        temp->prev->next = NULL;

    }

    printf("Deleted %d from end.\n", temp->data);

    free(temp);

}

// Function to display the Doubly Linked List forward

void display\_forward\_dll() {

    if (is\_empty\_dll()) {

        printf("DLL (forward): Empty\n");

        return;

    }

    printf("DLL (forward): ");

    DNode \*current = head;

    while (current != NULL) {

        printf("%d ", current->data);

        current = current->next;

    }

    printf("\n");

}

// Function to display the Doubly Linked List backward

void display\_backward\_dll() {

    if (is\_empty\_dll()) {

        printf("DLL (backward): Empty\n");

        return;

    }

    DNode \*current = head;

    // Traverse to the last node

    while (current->next != NULL) {

        current = current->next;

    }

    printf("DLL (backward): ");

    // Traverse backward

    while (current != NULL) {

        printf("%d ", current->data);

        current = current->prev;

    }

    printf("\n");

}

// Function to free the memory used by the DLL

void free\_dll() {

    DNode \*current = head;

    DNode \*next = NULL;

    while (current != NULL) {

        next = current->next;

        free(current);

        current = next;

    }

    head = NULL;

}

int main() {

    printf("--- Doubly Linked List ADT ---\n");

    insert\_end\_dll(10);

    insert\_end\_dll(20);

    insert\_front\_dll(5);

    display\_forward\_dll();  // Output: 5 10 20

    display\_backward\_dll(); // Output: 20 10 5

    delete\_end\_dll(); // Deletes 20

    delete\_front\_dll(); // Deletes 5

    display\_forward\_dll(); // Output: 10

    // Clean up memory

    free\_dll();

    return 0;

}

## Program Code C (C Language): Add Two Polynomials using Linked List

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <math.h> // For pow, though not strictly needed for display

// Node structure for a Polynomial Term

typedef struct PolyNode {

    int coeff;

    int expo;

    struct PolyNode \*next;

} PolyNode;

// Function to create a new PolyNode

PolyNode\* create\_poly\_node(int coeff, int expo) {

    PolyNode \*new\_node = (PolyNode \*)malloc(sizeof(PolyNode));

    if (new\_node == NULL) {

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

        exit(EXIT\_FAILURE);

    }

    new\_node->coeff = coeff;

    new\_node->expo = expo;

    new\_node->next = NULL;

    return new\_node;

}

// Function to insert a term at the end of a polynomial list

void insert\_poly\_term(PolyNode \*\*head, int coeff, int expo) {

    if (coeff == 0) return; // Skip zero coefficient terms

    PolyNode \*new\_node = create\_poly\_node(coeff, expo);

    if (\*head == NULL) {

        \*head = new\_node;

        return;

    }

    PolyNode \*temp = \*head;

    while (temp->next != NULL) {

        temp = temp->next;

    }

    temp->next = new\_node;

}

// Function to add two polynomials

PolyNode\* add\_polynomials(PolyNode \*p1, PolyNode \*p2) {

    PolyNode \*result\_head = NULL;

    PolyNode \*\*current\_ptr = &result\_head; // Pointer to the next pointer to be set

    while (p1 != NULL || p2 != NULL) {

        int sum\_coeff = 0;

        int current\_expo = 0;

        if (p1 == NULL) {

            // Only P2 terms remain

            sum\_coeff = p2->coeff;

            current\_expo = p2->expo;

            p2 = p2->next;

        } else if (p2 == NULL) {

            // Only P1 terms remain

            sum\_coeff = p1->coeff;

            current\_expo = p1->expo;

            p1 = p1->next;

        } else if (p1->expo == p2->expo) {

            // Exponents are equal: sum coefficients

            sum\_coeff = p1->coeff + p2->coeff;

            current\_expo = p1->expo;

            p1 = p1->next;

            p2 = p2->next;

        } else if (p1->expo > p2->expo) {

            // P1 has higher exponent: take P1 term

            sum\_coeff = p1->coeff;

            current\_expo = p1->expo;

            p1 = p1->next;

        } else {

            // P2 has higher exponent: take P2 term

            sum\_coeff = p2->coeff;

            current\_expo = p2->expo;

            p2 = p2->next;

        }

        // Only add term to result if coefficient is non-zero

        if (sum\_coeff != 0) {

            PolyNode \*new\_term = create\_poly\_node(sum\_coeff, current\_expo);

            \*current\_ptr = new\_term; // Link new term to the result list

            current\_ptr = &new\_term->next; // Move the tracking pointer to the next spot

        }

    }

    return result\_head;

}

// Function to display a polynomial in a readable format

void display\_polynomial(PolyNode \*head, char \*name) {

    if (head == NULL) {

        printf("%s = 0\n", name);

        return;

    }

    printf("%s = ", name);

    PolyNode \*current = head;

    bool first\_term = true;

    while (current != NULL) {

        if (!first\_term && current->coeff > 0) {

            printf(" + ");

        } else if (current->coeff < 0) {

            printf(" - ");

        }

        int abs\_coeff = abs(current->coeff);

        if (abs\_coeff != 1 || current->expo == 0) {

            printf("%d", abs\_coeff);

        }

        if (current->expo == 1) {

            printf("x");

        } else if (current->expo > 1) {

            printf("x^%d", current->expo);

        }

        first\_term = false;

        current = current->next;

    }

    printf("\n");

}

// Function to free the memory used by a polynomial list

void free\_polynomial(PolyNode \*head) {

    PolyNode \*current = head;

    PolyNode \*next = NULL;

    while (current != NULL) {

        next = current->next;

        free(current);

        current = next;

    }

}

int main() {

    PolyNode \*poly\_A = NULL;

    PolyNode \*poly\_B = NULL;

    PolyNode \*poly\_Sum = NULL;

    int num\_terms\_a, num\_terms\_b;

    int c, e;

    printf("--- Polynomial Addition using Linked List ---\n");

    // Input for Polynomial A

    printf("Enter number of terms for A: ");

    scanf("%d", &num\_terms\_a);

    printf("Enter terms for A (coeff expo) (Descending exponent order): \n");

    for (int i = 0; i < num\_terms\_a; i++) {

        scanf("%d %d", &c, &e);

        insert\_poly\_term(&poly\_A, c, e);

    }

    // Input for Polynomial B

    printf("Enter number of terms for B: ");

    scanf("%d", &num\_terms\_b);

    printf("Enter terms for B (coeff expo) (Descending exponent order): \n");

    for (int i = 0; i < num\_terms\_b; i++) {

        scanf("%d %d", &c, &e);

        insert\_poly\_term(&poly\_B, c, e);

    }

    // Display original polynomials

    display\_polynomial(poly\_A, "A(x)");

    display\_polynomial(poly\_B, "B(x)");

    // Add polynomials

    poly\_Sum = add\_polynomials(poly\_A, poly\_B);

    // Display result

    display\_polynomial(poly\_Sum, "A(x) + B(x)");

    // Clean up memory

    free\_polynomial(poly\_A);

    free\_polynomial(poly\_B);

    free\_polynomial(poly\_Sum);

    return 0;

}

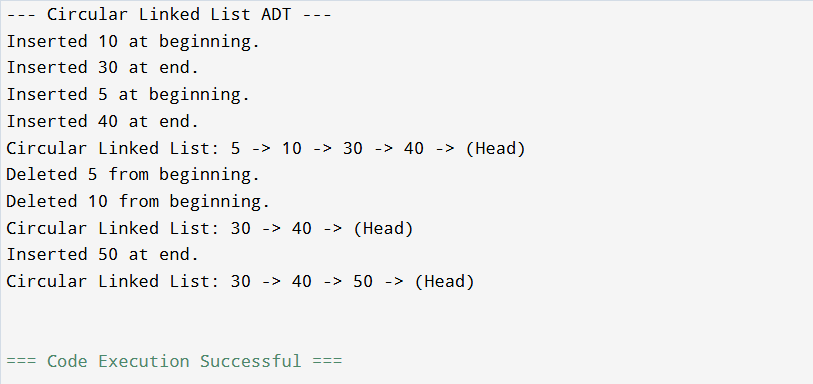
## Sample Input/Output:

Polynomial Addition Sample:  
Input:  
A terms: 3  
5 3  
2 1  
-4 0  
B terms: 3  
3 3  
-2 1  
6 0  
Output:  
A(x) = 5x^3 + 2x - 4  
B(x) = 3x^3 - 2x + 6  
A(x) + B(x) = 8x^3 + 2

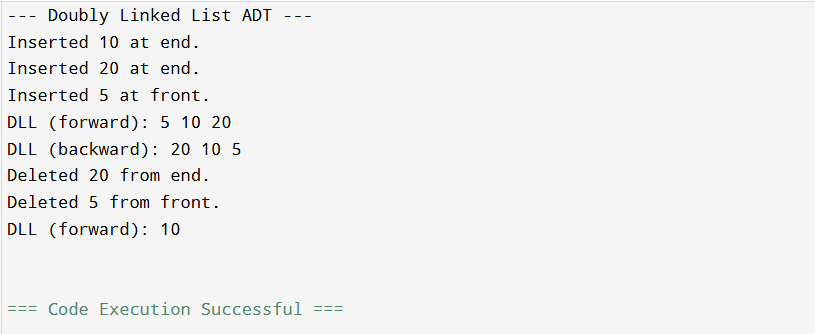
## Result:

Successfully implemented Circular Linked List ADT, Doubly Linked List ADT, and Polynomial Addition using Linked Lists with dynamic memory.

Program A Output:



Program B Output:



## Conclusion:

Pointer manipulation enables flexible insertion/deletion without shifting data. Circular lists support cyclic applications; doubly linked lists enable efficient bidirectional traversal; linked-list polynomials simplify sparse representation and arithmetic.

## Post-Lab Problem (Easy): Reverse a Doubly Linked List

Objective:

Write a function to reverse a doubly linked list in-place using pointer swaps; display before and after.

Description:

Given a doubly linked list, reverse the pointers so that the last node becomes the head. Do not create or delete nodes; only modify next/prev pointers.

Input:

First line: N (number of nodes), followed by N integers to insert at the end of DLL.

Output:

Display the list before reversal and after reversal (both forward traversal).

Constraints:

• Use only your DLL implementation.  
• 1 ≤ N ≤ 50.  
• Time O(N), Space O(1).

Sample I/O:

Input:  
5  
1 2 3 4 5  
Output:  
DLL (forward): 1 2 3 4 5  
DLL (forward) after reverse: 5 4 3 2 1

Hints:

• Traverse nodes and swap next and prev pointers for each node.  
• After loop, update head to the former tail node.

Post-Lab Code:

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// Node structure for Doubly Linked List

typedef struct DNode {

    int data;

    struct DNode \*prev;

    struct DNode \*next;

} DNode;

DNode \*head = NULL;

// Helper function to insert a node at the end (O(n))

void insert\_end\_postlab(int data) {

    DNode \*new\_node = (DNode \*)malloc(sizeof(DNode));

    if (new\_node == NULL) {

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

        return;

    }

    new\_node->data = data;

    new\_node->next = NULL;

    if (head == NULL) {

        new\_node->prev = NULL;

        head = new\_node;

        return;

    }

    DNode \*temp = head;

    while (temp->next != NULL) {

        temp = temp->next;

    }

    temp->next = new\_node;

    new\_node->prev = temp;

}

// Function to display the Doubly Linked List forward

void display\_forward\_postlab() {

    if (head == NULL) {

        printf("DLL (forward): Empty\n");

        return;

    }

    printf("DLL (forward): ");

    DNode \*current = head;

    while (current != NULL) {

        printf("%d ", current->data);

        current = current->next;

    }

    printf("\n");

}

// --- Main Function for Post-Lab Problem ---

// Function to reverse the Doubly Linked List in-place (O(n) time, O(1) extra space)

void reverse\_dll() {

    DNode \*current = head;

    DNode \*temp = NULL; // Temporary pointer to store 'prev' before swap

    // Edge case: empty list or single node

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

        return;

    }

    // Traverse the list and swap next and prev for every node

    while (current != NULL) {

        // 1. Store the current 'prev' pointer (this will be the new 'next')

        temp = current->prev;

        // 2. Swap next and prev pointers

        current->prev = current->next;

        current->next = temp;

        // 3. Move to the next node (which is now stored in the original 'prev')

        current = current->prev; // Important: use the new 'prev' which holds the original 'next'

    }

    // 4. Update head: the new head is the last node before reversal.

    // The last node is the node whose 'next' became NULL (the old 'head').

    // The reversal loop finishes with 'current' as NULL and 'temp' pointing to the new head.

    head = temp->prev;

}

// Function to free the memory used by the DLL

void free\_dll\_postlab() {

    DNode \*current = head;

    DNode \*next = NULL;

    while (current != NULL) {

        next = current->next;

        free(current);

        current = next;

    }

    head = NULL;

}

int main() {

    int N;

    int data;

    printf("--- Post-Lab: Reverse a Doubly Linked List ---\n");

    printf("Enter number of nodes (N <= 50): ");

    if (scanf("%d", &N) != 1 || N <= 0) {

        fprintf(stderr, "Invalid N.\n");

        return 1;

    }

    printf("Enter %d integers (data): ", N);

    for (int i = 0; i < N; i++) {

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

            fprintf(stderr, "Input error.\n");

            free\_dll\_postlab();

            return 1;

        }

        insert\_end\_postlab(data);

    }

    // Display before reversal

    printf("\n");

    display\_forward\_postlab();

    // Perform reversal

    reverse\_dll();

    // Display after reversal

    printf("DLL (forward) after reverse: ");

    display\_forward\_postlab();

    // Clean up memory

    free\_dll\_postlab();

    return 0;

}

Output:

