sll

list.c

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

// Define the structure for a node in the linked list
struct node {
    int key;  // Data stored in the node (an integer value)
    struct node *link;  // Pointer to the next node in the list
};
typedef struct node node_t;  // Create an alias for struct node called
node_t

// Define the structure for the linked list itself
struct list {
    node_t *head;  // Pointer to the first node (head) of the list
};
typedef struct list list_t;  // Create an alias for struct list called
list_t

// Function prototypes for list operations
void init_list(list_t*);  // Initializes an empty list by setting the head
to NULL
void delete_node(list_t *, int);  // Deletes a node by matching its value
void insert_head(list_t *, int);  // Inserts a new node at the beginning
(head) of the list
void insert_end(list_t*, int);  // Inserts a new node at the end of the list
void insert_pos(list_t *, int, int);  // Inserts a new node at a specific
position in the list
void display(list_t *);  // Displays the list in a readable format
void delete_pos(list_t *, int);  // Deletes a node by its position in the
list
int count_node(list_t *);  // Counts the number of nodes in the list using
recursion
int count_node_recur(node_t*);  // Helper function for recursive counting of
nodes
```

```
void print_reverse(list_t*);  // Prints the list in reverse order
void print_reverse_recur(node_t*);  // Helper function for printing the list
in reverse recursively
// void delete_alternate(list_t *);  // Function to delete every alternate
node (currently commented out)

int main() {
    list_t l;  // Declare a linked list variable
    init_list(&l);  // Initialize the linked list (set head to NULL)

    int n, ch, pos;  // Variables for user input (data, choice, position)

    while(1) {  // Infinite loop to continuously offer operations
        display(&l);  // Display the current list of nodes
        printf("\n1..Insert at head\n");
        printf("2..Insert at end\n");
        printf("3..Display the list\n");
        printf("4..Insert at a given position\n");
        printf("5..Delete node by its value\n");
        printf("6..Delete node by its position\n");
        printf("7..Count number of nodes using recursion\n");
        printf("8..Print the list in reverse\n");
        printf("9..Exit\n");

        scanf("%d", &ch);  // Get user choice for operation

        // Perform corresponding operation based on the user's choice
        switch(ch) {
            case 1:
                printf("\nEnter the element to be inserted: ");
                scanf("%d", &n);  // Get the value to insert
                insert_head(&l, n);  // Insert a node at the head
(beginning) of the list
                break;
            case 2:
                printf("\nEnter the element to be inserted: ");
                scanf("%d", &n);  // Get the value to insert
                insert_end(&l, n);  // Insert a node at the end of the list
                break;
            case 3:
```

```
                display(&l);  // Display the current list of nodes
                break;
            case 4:
                printf("\nEnter the element and position: ");
                scanf("%d %d", &n, &pos);  // Get the value and position to
insert at
                insert_pos(&l, n, pos);  // Insert the node at the specified
position
                break;
            case 5:
                printf("\nEnter the element to be deleted: ");
                scanf("%d", &n);  // Get the value of the node to delete
                delete_node(&l, n);  // Delete the node with the given value
                break;
            case 6:
                printf("\nEnter the position of the node to be deleted: ");
                scanf("%d", &pos);  // Get the position of the node to
delete
                delete_pos(&l, pos);  // Delete the node at the specified
position
                break;
            case 7:
                printf("Count the number of nodes using recursion\n");
                int count = count_node(&l);  // Count the number of nodes
recursively
                printf("\nThe number of nodes = %d\n", count);  // Print the
count of nodes
                break;
            case 8:
                printf("Print the list in reverse\n");
                print_reverse(&l);  // Print the list in reverse order
                break;
            case 9:
                exit(0);  // Exit the program
        }
    }
}

// Function to print the list in reverse order by using recursion
void print_reverse(list_t *ptr_list) {
```

```
    print_reverse_recur(ptr_list->head);  // Start the recursion with the
head of the list
}

// Helper function to print the list in reverse order recursively
void print_reverse_recur(node_t *node) {
    if (node != NULL) {  // If there is a node (non-NULL)
        print_reverse_recur(node->link);  // Recurse by moving to the next
node
        printf("%d -> ", node->key);  // Print the current node after the
recursion finishes (reverse order)
    }
}

// Function to count the number of nodes in the list using recursion
int count_node(list_t *ptr_list) {
    return count_node_recur(ptr_list->head);  // Call the recursive function
starting from the head node
}

// Helper function to count nodes recursively
int count_node_recur(node_t *node) {
    if (node == NULL)  // Base case: end of the list (no more nodes)
        return 0;  // Return 0 to stop counting
    return 1 + count_node_recur(node->link);  // Count the current node and
recurse for the next node
}

// Function to delete a node by its value
void delete_node(list_t *ptr_list, int data) {
    node_t *curr, *prev;  // curr is the current node, prev is the previous
node
    prev = NULL;
    curr = ptr_list->head;  // Start from the head node

    // Traverse the list until the node with the given value is found or the
end is reached
    while (curr != NULL && curr->key != data) {
        prev = curr;  // Move prev to curr
        curr = curr->link;  // Move curr to the next node
```

```
    }

    // If the node is found (curr is not NULL)
    if (curr != NULL) {
        if (prev == NULL) {  // If the node to delete is the head node
            ptr_list->head = curr->link;  // Move the head to the next node
        } else {  // If the node is not the head
            prev->link = curr->link;  // Skip the current node by linking
the previous node to the next node
        }
        free(curr);  // Free the memory of the deleted node
    } else {
        printf("Node with value %d not found.\n", data);  // Node not found
in the list
    }
}

// Function to insert a node at a specific position in the list
void insert_pos(list_t *ptr_list, int data, int pos) {
    node_t *curr, *new_node, *prev;
    int i = 1;

    // Allocate memory for a new node and set its data
    new_node = (node_t *)malloc(sizeof(node_t));
    new_node->key = data;
    new_node->link = NULL;  // Initially, the new node points to NULL (end
of list)

    prev = NULL;
    curr = ptr_list->head;  // Start at the head node

    // Traverse the list to reach the position where we need to insert the
new node
    while (curr != NULL && i < pos) {
        i++;
        prev = curr;  // Move prev to the current node
        curr = curr->link;  // Move curr to the next node
    }

    // Insert the new node at the specified position
```

```
    if (curr != NULL) {  // If the position is within the list
        if (prev == NULL) {  // If inserting at the head
            ptr_list->head = new_node;  // Update head to point to the new
node
            new_node->link = curr;  // Link the new node to the current
first node
        } else {  // If inserting in the middle
            prev->link = new_node;  // Link the previous node to the new
node
            new_node->link = curr;  // Link the new node to the next node
        }
    } else if (i == pos) {  // If inserting at the end of the list
        if (prev != NULL) {
            prev->link = new_node;  // Link the last node to the new node
        } else {
            ptr_list->head = new_node;  // If list was empty, set the new
node as head
        }
    } else {
        printf("\nInvalid position.\n");  // If the position is invalid (out
of bounds)
    }
}

// Function to insert a node at the end of the list
void insert_end(list_t *ptr_list, int data) {
    node_t *curr, *new_node;

    // Allocate memory for the new node and set its data
    new_node = (node_t *)malloc(sizeof(node_t));
    new_node->key = data;
    new_node->link = NULL;  // The new node will be the last, so its link is
NULL

    // If the list is empty, set the new node as the head
    if (ptr_list->head == NULL) {
        ptr_list->head = new_node;
    } else {
        curr = ptr_list->head;  // Start at the head node
```

```
        // Traverse the list to find the last node
        while (curr->link != NULL) {
            curr = curr->link;  // Move to the next node until we find the
last one
        }

        curr->link = new_node;  // Link the last node to the new node
(insert at the end)
    }
}

// Function to display the contents of the linked list
void display(list_t *ptr_list) {
    node_t *curr;  // Pointer to traverse the list

    // Check if the list is empty
    if (ptr_list->head == NULL) {
        printf("\nEmpty list.\n");
    } else {
        curr = ptr_list->head;  // Start at the head node
        while (curr != NULL) {
            printf("%d -> ", curr->key);  // Print the current node's value
            curr = curr->link;  // Move to the next node
        }
        printf("NULL\n");  // Indicate the end of the list
    }
}

// Function to insert a node at the head (beginning) of the list
void insert_head(list_t *ptr_list, int data) {
    node_t *new_node;

    // Allocate memory for the new node and set its data
    new_node = (node_t *)malloc(sizeof(node_t));
    new_node->key = data;
    new_node->link = ptr_list->head;  // The new node's link will point to
the current head

    // Update the head of the list to point to the new node
    ptr_list->head = new_node;
```

```
}

// Function to initialize the list (set head to NULL)
void init_list(list_t *ptr_list) {
    ptr_list->head = NULL;  // Set the head pointer to NULL, indicating an
empty list
}

// Function to delete a node by its position in the list
void delete_pos(list_t *ptr_list, int pos) {
    node_t *curr, *prev;
    int i = 1;

    curr = ptr_list->head;  // Start from the head node
    prev = NULL;

    // Traverse the list until the desired position is reached or the list
ends
    while (curr != NULL && i < pos) {
        prev = curr;  // Move prev to the current node
        curr = curr->link;  // Move curr to the next node
        i++;
    }

    // If the position is valid (i.e., within the list)
    if (curr != NULL) {
        if (prev == NULL) {  // If deleting the head node
            ptr_list->head = curr->link;  // Move the head to the next node
        } else {  // If deleting a node other than the head
            prev->link = curr->link;  // Bypass the node to delete it
        }
        free(curr);  // Free the memory of the deleted node
    } else {
        printf("\nInvalid position.\n");  // If the position is invalid (out
of bounds)
    }
}
```

merge.c

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

// Structure to represent a node in the linked list
struct node
{
    int key;        // The value stored in the node
    struct node *link;  // Pointer to the next node in the list
};

typedef struct node node_t; // Type definition for 'node_t' representing the
node structure

// Structure to represent the linked list itself
struct list
{
    node_t* head;  // Pointer to the first node of the list
};

typedef struct list list_t; // Type definition for 'list_t' representing the
linked list structure

// Function Prototypes
void init_list(list_t*);               // Initializes the linked list
void insert_order(list_t*, int);       // Inserts a node into the list
maintaining sorted order
void display(list_t *);                // Displays the content of the linked
list
void create_list(list_t *);            // Allows the user to create a list
by entering elements
void merge(list_t *, list_t *, list_t *); // Merges two sorted lists into a
third list
void insert_tail(list_t*, int);        // Inserts a node at the end of the
list

int main()
{
    list_t l1, l2, l3; // Declare three linked lists: l1, l2, and l3 (merged
list)
```

```
    init_list(&l1);    // Initialize the first list l1
    init_list(&l2);    // Initialize the second list l2
    init_list(&l3);    // Initialize the third list l3 (for merged results)

    // Create the first list with user input
    printf("\nCreating List one..\n");
    create_list(&l1);
    printf("Displaying List one..\n");
    display(&l1);   // Display the first list

    // Create the second list with user input
    printf("\nCreating List Two..\n");
    create_list(&l2);
    printf("Displaying List two..\n");
    display(&l2);   // Display the second list

    // Merge the two lists into the third list (l3)
    printf("\nMerging both the Lists..\n");
    merge(&l1, &l2, &l3);
    printf("\nDisplaying merged list..\n");
    display(&l3);   // Display the merged list

    return 0;
}

// Initializes the list by setting the head to NULL (empty list)
void init_list(list_t *ptr_list)
{
    ptr_list->head = NULL;  // Set the head pointer to NULL to indicate an
empty list
}

// Inserts a new node into the list, maintaining the sorted order
void insert_order(list_t *ptr_list, int key)
{
    node_t *pres, *prev, *temp;
    prev = NULL;              // Initialize the previous pointer to NULL
    pres = ptr_list->head;    // Start at the head of the list

    // Dynamically allocate memory for a new node and set its key
```

```
    temp = (node_t*)malloc(sizeof(node_t));
    temp->key = key;
    temp->link = NULL;        // Set the link of the new node to NULL

    // Traverse the list to find the correct insertion point
    while((pres != NULL) && (key > pres->key))  // Traverse until a node
with a larger key is found
    {
        prev = pres;            // Move 'prev' to 'pres'
        pres = pres->link;      // Move 'pres' to the next node
    }

    if (pres != NULL) // A position has been found for insertion
    {
        if (prev == NULL) // Inserting at the beginning of the list
        {
            temp->link = pres;  // Set the link of new node to the current
first node
            ptr_list->head = temp; // Make the new node the head
        }
        else // Inserting between two nodes
        {
            temp->link = pres;  // Set the link of the new node to the
current node
            prev->link = temp;   // Set the previous node's link to the new
node
        }
    }
    else // Inserting at the end of the list (pres is NULL)
    {
        if (prev == NULL) // The list is empty, insert the first node
            ptr_list->head = temp;  // Set the new node as the head
        else // The list is not empty, insert at the end
            prev->link = temp;  // Set the last node's link to the new node
}

// Displays the content of the linked list
void display(list_t *ptr_list)
{
    node_t* pres;
```

```
    if (ptr_list->head == NULL)  // If the list is empty, print a message
    {
        printf("Empty List..\n");
    }
    else // If the list is not empty, traverse and print all elements
    {
        pres = ptr_list->head;  // Start at the head of the list
        while (pres != NULL)     // Traverse the list until the end
        {
            printf("%d-> ", pres->key);  // Print the key of the current
node
            pres = pres->link;           // Move to the next node
        }
    }
}

// Creates a list by allowing the user to input elements
void create_list(list_t *ptr_list)
{
    int x;
    while (1)
    {
        printf("\nEnter the Element (enter 0 to stop): ");
        scanf("%d", &x);

        if (x == 0)  // Stop input when the user enters 0
            break;

        insert_order(ptr_list, x);  // Insert the entered value into the
list maintaining order
    }
}

// Merges two sorted lists (ptr_list1 and ptr_list2) into a third list
(ptr_list3)
void merge(list_t *ptr_list1, list_t *ptr_list2, list_t *ptr_list3)
{
    node_t *p, *q;
```

```
    p = ptr_list1->head;  // Start at the head of the first list
    q = ptr_list2->head;  // Start at the head of the second list

    // Traverse both lists and merge them into the third list
    while ((p != NULL) && (q != NULL))
    {
        if (p->key < q->key)  // If the current node in list 1 is smaller
        {
            insert_tail(ptr_list3, p->key);  // Insert into the merged list
            p = p->link;  // Move to the next node in list 1
        }
        else // If the current node in list 2 is smaller or equal
        {
            insert_tail(ptr_list3, q->key);  // Insert into the merged list
            q = q->link;  // Move to the next node in list 2
        }
    }

    // If one list is exhausted, append the remaining nodes of the other
list
    if (p == NULL)  // List 1 is exhausted
    {
        while (q != NULL)  // Insert remaining nodes from list 2
        {
            insert_tail(ptr_list3, q->key);
            q = q->link;
        }
    }
    else  // List 2 is exhausted
    {
        while (p != NULL)  // Insert remaining nodes from list 1
        {
            insert_tail(ptr_list3, p->key);
            p = p->link;
        }
    }
}

// Inserts a new node at the end (tail) of the list
void insert_tail(list_t *ptr_list, int data)
```

```
{
    node_t *pres, *temp;

    // Dynamically allocate memory for a new node and set its key
    temp = (node_t*)malloc(sizeof(node_t));
    temp->key = data;
    temp->link = NULL;  // Set the link of the new node to NULL (it's the
last node)

    if (ptr_list->head == NULL)  // If the list is empty, set the new node
as the head
        ptr_list->head = temp;
    else
    {
        pres = ptr_list->head;  // Start at the head of the list
        while (pres->link != NULL)  // Traverse to the last node
            pres = pres->link;

        pres->link = temp;  // Set the last node's link to the new node
    }
}
```

orderlist.c

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

// Structure to represent a node in the linked list
struct node
{
    int key;        // The value stored in the node (integer)
    struct node *link;  // Pointer to the next node in the list
};

typedef struct node node_t;  // Type definition for 'node_t' representing
the node structure

// Structure to represent the linked list itself
struct list
```

```
{
    node_t* head;  // Pointer to the first node of the list
};

typedef struct list list_t;  // Type definition for 'list_t' representing
the linked list structure

// Function Prototypes
void init_list(list_t*);               // Initializes the linked list (sets
the head to NULL)
void insert_order(list_t*, int);       // Inserts a node into the list while
maintaining sorted order
void display(list_t *);                // Displays the content of the linked
list

int main()
{
    list_t l;  // Declare a linked list variable 'l'

    init_list(&l);  // Initialize the linked list (set head to NULL)

    int x;  // Variable to hold the user's input for new elements
    display(&l);   // Display the current state of the linked list

    while(1)
    {
        printf("\nEnter the element: ");  // Prompt user to enter an element
        scanf("%d", &x);  // Read the input integer

        if(x == 0)  // If the input is 0, stop the loop
            break;

        insert_order(&l, x);  // Insert the entered value into the list in
sorted order
        printf("\n");
        display(&l);  // Display the updated list after insertion
    }

    return 0;
}
```

```
// Initializes the list by setting the head pointer to NULL (empty list)
void init_list(list_t *ptr_list)
{
    ptr_list->head = NULL;  // The list is initially empty
}

// Inserts a new node into the list in sorted order
void insert_order(list_t *ptr_list, int x)
{
    node_t *pres, *prev, *temp;

    // Dynamically allocate memory for a new node and set its key
    temp = (node_t*)malloc(sizeof(node_t));
    temp->key = x;
    temp->link = NULL;  // The new node will initially point to NULL

    prev = NULL;  // Initialize the previous pointer to NULL
    pres = ptr_list->head;  // Start at the head of the list

    // Traverse the list to find the correct position to insert the new node
    while (pres != NULL && x > pres->key)  // Move until we find a node with
a larger key or reach the end
    {
        prev = pres;           // Move prev to the current node
        pres = pres->link;     // Move pres to the next node
    }

    // If we found the correct position to insert the new node
    if (pres != NULL)
    {
        if (prev == NULL)  // If prev is NULL, the new node should be
inserted at the beginning
        {
            ptr_list->head = temp;  // Make the new node the head of the
list
            temp->link = pres;  // Link the new node to the current head
        }
        else  // Inserting between two nodes (middle of the list)
        {
```

```
            temp->link = pres;  // Link the new node to the next node
            prev->link = temp;   // Set the previous node's link to the new
node
        }
    }
    else  // If we reached the end of the list (pres is NULL), insert at the
end
    {
        if (prev == NULL)  // If prev is NULL, the list is empty, so insert
as the first node
        {
            ptr_list->head = temp;  // Set the new node as the head of the
list
        }
        else  // Insert at the end (largest number)
        {
            prev->link = temp;  // Link the last node to the new node
        }
    }
}

// Displays the content of the linked list
void display(list_t *ptr_list)
{
    node_t *pres;

    // Check if the list is empty (head is NULL)
    if (ptr_list->head == NULL)
    {
        printf("\nEmpty list.\n");  // If the list is empty, print a message
    }
    else
    {
        pres = ptr_list->head;  // Start from the head of the list

        // Traverse the list and print each node's key value
        while (pres != NULL)
        {
            printf("%d --> ", pres->key);  // Print the value of the current
node
```

```
            pres = pres->link;  // Move to the next node
        }
        printf("NULL\n");  // Print NULL to indicate the end of the list
    }
}
```

poly.c

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

// Define a structure for a node in the linked list
struct node
{
    int coeff;  // The coefficient of the term (e.g., for 3x^2, the
coefficient is 3)
    int px;     // The power of x (exponent of x in the term)
    int py;     // The power of y (exponent of y in the term)
    int flag;   // A flag used for tracking whether a node has been
processed
    struct node *link;  // Pointer to the next node in the linked list
};

typedef struct node node_t;  // Type definition for 'node_t' representing
the node structure

// Define a structure for the linked list itself
struct list
{
    node_t* head;  // Pointer to the first node in the linked list
};

typedef struct list list_t;  // Type definition for 'list_t' representing
the linked list structure

// Function prototypes
void init_list(list_t*);               // Initializes the linked list (sets
the head to NULL)
void create_polynomial(list_t*);       // Creates a polynomial by inserting
```

```
terms into the list
void display(list_t *);                // Displays the content of the linked
list (polynomial)
void add(list_t *, list_t *, list_t *); // Adds two polynomials
void insert(list_t *ptr_list, int co, int x, int y); // Inserts a new term
into the polynomial
void append(list_t *, list_t *, list_t *); // Appends non-overlapping terms
from two polynomials

int main()
{
    list_t l1, l2, l3;  // Declare three linked lists for three polynomials
    init_list(&l1);  // Initialize list l1
    init_list(&l2);  // Initialize list l2
    init_list(&l3);  // Initialize list l3

    // Create the two polynomials by input from the user
    create_polynomial(&l1);
    create_polynomial(&l2);

    // Display the first polynomial
    printf("First polynomial:\n");
    display(&l1);

    // Display the second polynomial
    printf("Second polynomial:\n");
    display(&l2);

    // Add the two polynomials and store the result in l3
    add(&l1, &l2, &l3);

    // Append any remaining terms from l1 and l2 to l3
    append(&l1, &l2, &l3);

    // Display the resulting polynomial after addition and append
    printf("\nThird polynomial (result of addition and append):\n");
    display(&l3);

    return 0;
}
```

```
// Function to initialize an empty list (sets the head to NULL)
void init_list(list_t *ptr_list)
{
    ptr_list->head = NULL;  // The list is initially empty
}

// Function to create a polynomial by inserting terms from user input
void create_polynomial(list_t *ptr_list)
{
    while(1)
    {
        int co, x, y;
        printf("Enter coefficient (enter 0 to stop): ");
        scanf("%d", &co);  // Input the coefficient of the term

        if(co == 0)
        {
            break;  // Stop if the coefficient is 0
        }

        // Input the powers of x and y
        printf("Enter power of x and y: ");
        scanf("%d %d", &x, &y);  // Input the exponents of x and y

        // Insert the term (co, x, y) into the polynomial list
        insert(ptr_list, co, x, y);
    }
}

// Function to insert a new term into the polynomial list
void insert(list_t *ptr_list, int co, int x, int y)
{
    node_t *temp = (node_t *)malloc(sizeof(node_t));  // Dynamically
allocate memory for a new node
    temp->coeff = co;  // Set the coefficient of the term
    temp->px = x;      // Set the power of x
    temp->py = y;      // Set the power of y
    temp->flag = 1;    // Set the flag to 1 to indicate this node is valid
    temp->link = NULL; // Initially, the link is NULL since it's the last
```

```
node

    // If the list is empty, make the new node the head
    if(ptr_list->head == NULL)
    {
        ptr_list->head = temp;
    }
    else
    {
        node_t *p = ptr_list->head;

        // Traverse to the end of the list
        while(p->link != NULL)
        {
            p = p->link;
        }

        // Link the new node at the end of the list
        p->link = temp;
    }
}

// Function to display the contents of the polynomial list
void display(list_t *ptr_list)
{
    node_t *pres = ptr_list->head;  // Start from the head of the list

    // Check if the list is empty
    if(pres == NULL)
    {
        printf("Empty list.\n");
        return;
    }

    // Traverse and print each term in the polynomial
    while(pres != NULL)
    {
        printf("%d x^%d y^%d -> ", pres->coeff, pres->px, pres->py);  //
Display term
        pres = pres->link;  // Move to the next node
```

```
    }
    printf("NULL\n");  // Indicate the end of the list
}

// Function to add two polynomials and store the result in a third
polynomial list
void add(list_t *ptr_list1, list_t *ptr_list2, list_t *ptr_list3)
{
    node_t *i, *j;
    i = ptr_list1->head;
    j = ptr_list2->head;

    printf("Entering add..\n");

    // Traverse the first polynomial list
    for(i = ptr_list1->head; i != NULL; i = i->link)
    {
        // Traverse the second polynomial list
        for(j = ptr_list2->head; j != NULL; j = j->link)
        {
            // If terms have the same powers of x and y, add their
coefficients
            if(i->px == j->px && i->py == j->py)
            {
                int k = i->coeff + j->coeff;  // Add the coefficients
                i->flag = 0;  // Mark the term in the first polynomial as
processed
                j->flag = 0;  // Mark the term in the second polynomial as
processed
                insert(ptr_list3, k, i->px, i->py);  // Insert the result
into the third polynomial
            }
        }
    }
    printf("Exit add\n");
}

// Function to append terms from both polynomials that were not added (i.e.,
flag == 1)
void append(list_t *ptr_list1, list_t *ptr_list2, list_t *ptr_list3)
```

```
{
    node_t *i, *j;
    i = ptr_list1->head;
    j = ptr_list2->head;

    printf("Entering append\n");

    // Append the unprocessed terms from the first polynomial
    while(i != NULL)
    {
        if(i->flag == 1)  // If the term was not processed, append it
        {
            insert(ptr_list3, i->coeff, i->px, i->py);
        }
        i = i->link;
    }

    // Append the unprocessed terms from the second polynomial
    while(j != NULL)
    {
        if(j->flag == 1)  // If the term was not processed, append it
        {
            insert(ptr_list3, j->coeff, j->px, j->py);
        }
        j = j->link;
    }

    printf("Exit append\n");
}
```

poly evaluate.c

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

// Define the structure of a node in the linked list
struct node
{
```

```
    int coeff;   // Coefficient of the polynomial term
    int px;      // Power of x in the term
    int py;      // Power of y in the term
    struct node *next; // Pointer to the next node in the linked list
};

// Function prototypes
void insert_tail(int, int, int, struct node**);   // Function to insert a
term at the end of the list
void display(struct node*);                        // Function to display
the polynomial
int polyevaluate(struct node*);                    // Function to evaluate
the polynomial

int main()
{
    struct node *first = NULL;  // Initialize the linked list as empty

    int cf, px, py, result;  // Variables for coefficient, power of x, and
power of y

    // Create the polynomial by repeatedly entering terms
    while(1)
    {
        // Input the coefficient of the term
        printf("\nEnter the coefficient of the term: ");
        scanf("%d", &cf);

        // Stop if the coefficient is 0 (indicating end of input)
        if(cf == 0)
            break;

        // Input the power of x and y for the current term
        printf("\nEnter the power of x: ");
        scanf("%d", &px);
        printf("\nEnter the power of y: ");
        scanf("%d", &py);

        // Insert the term at the end of the list
        insert_tail(cf, px, py, &first);
```

```
    }

    // Display the created polynomial
    printf("\nThe polynomial created is:\n");
    display(first);

    // Evaluate the polynomial and display the result
    printf("\nEvaluating the polynomial...\n");
    result = polyevaluate(first);
    printf("Result = %d\n", result);

    return 0;
}

// Function to evaluate the polynomial by substituting values of x and y
int polyevaluate(struct node *p)
{
    int x, y, sum = 0;

    // Input the values of x and y
    printf("\nEnter the value of x and y: ");
    scanf("%d %d", &x, &y);

    // Traverse through the linked list to calculate the sum of polynomial
terms
    while(p != NULL)
    {
        // Evaluate the term and add it to the sum
        sum += (p->coeff * pow(x, p->px) * pow(y, p->py));
        p = p->next;  // Move to the next term in the polynomial
    }

    return sum;  // Return the final result of the polynomial evaluation
}

// Function to display the polynomial
void display(struct node *q)
{
    // Traverse through the linked list and print each term
    while(q != NULL)
```

```
    {
        // Print the coefficient, handling positive and negative values
        if(q->coeff > 0)
            printf("+%d", q->coeff);
        else
            printf("%d", q->coeff);

        // Print the power of x, if applicable
        if(q->px > 0)
        {
            if(q->px == 1)
                printf("X");  // If power is 1, just print "X"
            else
                printf("X^%d", q->px);  // Otherwise, print "X^power"
        }

        // Print the power of y, if applicable
        if(q->py > 0)
        {
            if(q->py == 1)
                printf("Y");  // If power is 1, just print "Y"
            else
                printf("Y^%d", q->py);  // Otherwise, print "Y^power"
        }

        q = q->next;  // Move to the next term in the polynomial
    }
    printf("\n");  // End the polynomial output with a newline
}

// Function to insert a new term at the end of the linked list
void insert_tail(int cf, int px, int py, struct node **p)
{
    struct node *q, *temp;

    // Dynamically allocate memory for a new node
    temp = (struct node*)malloc(sizeof(struct node));

    // Initialize the node with the provided values
    temp->coeff = cf;
```

```
    temp->px = px;
    temp->py = py;
    temp->next = NULL;  // The new node will be the last one, so its next is
NULL

    q = *p;  // Pointer to the head of the linked list

    // If the list is empty, make the new node the head
    if(q == NULL)
    {
        *p = temp;
    }
    else
    {
        // Traverse to the end of the list
        while(q->next != NULL)
            q = q->next;

        // Insert the new node at the end
        q->next = temp;
    }
}
```

dll

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

// Define the structure of a node in the circular doubly linked list
struct node
{
    int data;           // Data value stored in the node
    struct node* next;  // Pointer to the next node in the list
    struct node* prev;  // Pointer to the previous node in the list
};

typedef struct node node_t;  // Alias for struct node
struct dlist
{
```

```
    node_t* head;  // Pointer to the head (first node) of the circular
doubly linked list
};

typedef struct dlist dlist_t;  // Alias for struct dlist

// Function prototypes
void insert_head(dlist_t*, int);  // Function to insert a node at the head
void insert_tail(dlist_t*, int);  // Function to insert a node at the tail
void delete_node(dlist_t*, int);  // Function to delete a node by its data
value
void display(dlist_t*);           // Function to display the entire list
void init_list(dlist_t*);         // Function to initialize the list (empty
list)

int main()
{
    int ch, x;  // Menu choice and data for nodes
    dlist_t dl;  // Circular doubly linked list instance

    init_list(&dl);  // Initialize the list to be empty

    while(1)
    {
        // Display the current list and menu options
        display(&dl);
        printf("\n1..Insert Head\n");
        printf("2..Insert Tail\n");
        printf("3..Delete a Node\n");
        printf("4..Display\n");
        printf("5..Exit\n");
        scanf("%d", &ch);  // Take user input for menu choice

        switch(ch)
        {
            case 1:
                printf("Enter the number to insert at the head: ");
                scanf("%d", &x);
                insert_head(&dl, x);  // Insert node at the head
                break;
```

```
            case 2:
                printf("Enter the number to insert at the tail: ");
                scanf("%d", &x);
                insert_tail(&dl, x);  // Insert node at the tail
                break;
            case 3:
                printf("Enter the value of the node to be deleted: ");
                scanf("%d", &x);
                delete_node(&dl, x);  // Delete a node by its data
                break;
            case 4:
                display(&dl);  // Display the current list
                break;
            case 5:
                exit(0);  // Exit the program
        }
    }
}

// Function to initialize the circular doubly linked list
void init_list(dlist_t* ptr_list)
{
    ptr_list->head = NULL;  // Set the head of the list to NULL (empty list)
}

// Function to delete a node by its data value
void delete_node(dlist_t *ptr_list, int x)
{
    node_t *pres = ptr_list->head, *temp;

    // If the list is empty, no deletion can occur
    if (pres == NULL)
    {
        printf("List is empty. No node to delete.\n");
        return;
    }

    // Traverse the list to find the node with matching data
    do
    {
```

```
        if (pres->data == x)
            break;
        pres = pres->next;
    } while (pres != ptr_list->head);  // Loop until we complete a full
circle

    // If node with value x is found
    if (pres->data == x)
    {
        // If the node to be deleted is the only node in the list
        if (pres->next == pres)
        {
            ptr_list->head = NULL;  // List becomes empty
        }
        else
        {
            pres->prev->next = pres->next;  // Bypass the node in the
previous node's next pointer
            pres->next->prev = pres->prev;  // Bypass the node in the next
node's prev pointer

            // If the node to be deleted is the head node, move the head
pointer to the next node
            if (pres == ptr_list->head)
            {
                ptr_list->head = pres->next;
            }
        }
        free(pres);  // Free the memory of the deleted node
    }
    else
    {
        printf("Node not found in the list.\n");
    }
}

// Function to insert a node at the head of the list
void insert_head(dlist_t *ptr_list, int x)
{
    node_t *temp = (node_t*)malloc(sizeof(node_t));  // Allocate memory for
```

```
a new node
    temp->data = x;  // Set the data of the new node
    temp->next = temp->prev = temp;  // The new node's next and prev point
to itself (circular link)

    node_t *pres = ptr_list->head;

    // If the list is empty, make the new node the head
    if (pres == NULL)
    {
        ptr_list->head = temp;
    }
    else
    {
        // Insert the new node before the current head node
        pres->prev->next = temp;  // Previous node's next points to the new
node
        temp->prev = pres->prev;  // New node's prev points to the previous
node
        temp->next = pres;        // New node's next points to the current
head
        pres->prev = temp;        // Current head's prev points to the new
node
        ptr_list->head = temp;    // Set the new node as the head
    }
}

// Function to display the entire circular doubly linked list
void display(dlist_t *ptr_list)
{
    node_t *pres = ptr_list->head;

    if (pres == NULL)
    {
        printf("\nEmpty list..\n");
    }
    else
    {
        node_t *q = pres;
        // Traverse and display all nodes in the list
```

```
        do
        {
            printf("%d <-> ", q->data);  // Print the current node's data
            q = q->next;  // Move to the next node
        } while (q != pres);  // Stop when we complete a full circle
        printf("%d <-> ...\n", pres->data);  // Print the first node again
to show circular nature
    }
}

// Function to insert a node at the tail of the list
void insert_tail(dlist_t *ptr_list, int x)
{
    node_t *temp = (node_t*)malloc(sizeof(node_t));  // Allocate memory for
a new node
    temp->data = x;  // Set the data of the new node
    temp->next = temp->prev = temp;  // The new node's next and prev point
to itself (circular link)

    node_t *pres = ptr_list->head;

    // If the list is empty, make the new node the head
    if (pres == NULL)
    {
        ptr_list->head = temp;
    }
    else
    {
        // Insert the new node at the tail (before the head)
        pres->prev->next = temp;  // The last node's next points to the new
node
        temp->prev = pres->prev;  // New node's prev points to the last node
        temp->next = pres;        // New node's next points to the head (to
maintain circular link)
        pres->prev = temp;        // Head's prev points to the new node
    }
}
```

cll

clist

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

// Define the structure for a node in the circular singly linked list
struct node
{
    int key;            // Data stored in the node
    struct node* link; // Pointer to the next node
};

typedef struct node node_t; // Alias for struct node

// Define the structure for the circular singly linked list
struct list
{
    node_t* last; // Pointer to the last node in the list
};

typedef struct list clist_t; // Alias for struct list

// Function prototypes
void init_list(clist_t*);           // Initialize the circular list
void insert_front(clist_t*, int);   // Insert a node at the front of the
list
void insert_end(clist_t*, int);     // Insert a node at the end of the list
void delete_node(clist_t*, int);    // Delete a node by key
void display(clist_t*);             // Display the list

int main()
{
    int ch, key;  // Menu choice and key value for nodes
    clist_t l;    // Circular singly linked list instance

    init_list(&l);  // Initialize the circular list as empty

    while(1)
    {
        // Display the current list and menu options
```

```
        display(&l);
        printf("\n1..Insert Front");
        printf("\n2..Insert End");
        printf("\n3..Display");
        printf("\n4..Delete a Node");
        printf("\n5..Exit");
        scanf("%d", &ch);  // Take user input for menu choice

        switch(ch)
        {
            case 1:
                printf("\nEnter the key to insert at the front: ");
                scanf("%d", &key);
                insert_front(&l, key);  // Insert node at the front
                break;
            case 2:
                printf("\nEnter the key to insert at the end: ");
                scanf("%d", &key);
                insert_end(&l, key);  // Insert node at the end
                break;
            case 3:
                display(&l);  // Display the current list
                break;
            case 4:
                printf("\nEnter the key of the node to delete: ");
                scanf("%d", &key);
                delete_node(&l, key);  // Delete the node by its key
                break;
            case 5:
                exit(0);  // Exit the program
        }
    }
}

// Function to initialize the circular singly linked list
void init_list(clist_t* ptr_list)
{
    ptr_list->last = NULL;  // Set the last pointer to NULL, indicating an
empty list
}
```

```
// Function to insert a node at the front of the circular singly linked list
void insert_front(clist_t* ptr_list, int key)
{
    node_t* temp = (node_t*)malloc(sizeof(node_t));  // Allocate memory for
the new node
    temp->key = key;  // Set the key of the new node
    temp->link = temp;  // Link the new node to itself (as it's the only
node in the list)

    node_t* last = ptr_list->last;  // Get the address of the last node

    if (last == NULL)  // If the list is empty
    {
        ptr_list->last = temp;  // Set the last node to the new node
    }
    else
    {
        temp->link = last->link;  // The new node's link points to the
current first node
        last->link = temp;        // The last node's link now points to the
new node
    }
}

// Function to insert a node at the end of the circular singly linked list
void insert_end(clist_t* ptr_list, int key)
{
    node_t* temp = (node_t*)malloc(sizeof(node_t));  // Allocate memory for
the new node
    temp->key = key;  // Set the key of the new node
    temp->link = temp;  // Link the new node to itself (as it's the only
node in the list)

    node_t* last = ptr_list->last;  // Get the address of the last node

    if (last == NULL)  // If the list is empty
    {
        ptr_list->last = temp;  // Set the last node to the new node
    }
```

```
    else
    {
        temp->link = last->link;  // The new node's link points to the
current first node
        last->link = temp;        // The last node's link now points to the
new node
        ptr_list->last = temp;    // Update the last pointer to the new node
    }
}

// Function to display the circular singly linked list
void display(clist_t* ptr_list)
{
    if (ptr_list->last == NULL)  // Check if the list is empty
    {
        printf("Empty List\n");
    }
    else
    {
        node_t* pres = ptr_list->last->link;  // Start from the first node
(next of last)

        do
        {
            printf("%d -> ", pres->key);  // Print the key of the current
node
            pres = pres->link;  // Move to the next node
        } while (pres != ptr_list->last->link);  // Stop when we complete a
full circle

        printf("(Back to the first node)\n");
    }
}

// Function to delete a node by its key in the circular singly linked list
void delete_node(clist_t* ptr_list, int key)
{
    node_t* last = ptr_list->last;  // Get the address of the last node
    if (last == NULL)  // If the list is empty
    {
```

```
        printf("List is empty. Cannot delete.\n");
        return;
    }

    node_t* pres = last->link;  // Start from the first node
    node_t* prev = last;  // Keep track of the previous node

    // Traverse the list to find the node with the matching key
    while (pres->key != key && pres != last)
    {
        prev = pres;       // Update previous node
        pres = pres->link; // Move to the next node
    }

    if (pres->key == key)  // Node with matching key found
    {
        // If the node to be deleted is the only node in the list
        if (pres->link == pres)
        {
            ptr_list->last = NULL;  // Set the last pointer to NULL (empty
list)
        }
        else
        {
            prev->link = pres->link;  // Link the previous node to the next
node
            if (pres == last)         // If the last node is to be deleted
            {
                ptr_list->last = prev;  // Update the last pointer to the
previous node
            }
        }
        free(pres);  // Free the memory of the deleted node
    }
    else
    {
        printf("Node with key %d not found.\n", key);  // If node with key
is not found
    }
```

```
}
```

dlist

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

// Define the node structure for the doubly linked list
struct node
{
  int key;             // Stores the data value of the node
  struct node *next;   // Pointer to the next node in the list
  struct node *prev;   // Pointer to the previous node in the list
};

typedef struct node node_t;  // Create a typedef for node_t for ease of use

// Define the doubly linked list structure
struct dlist
{
  node_t* head;        // Points to the first node in the doubly linked list
};

typedef struct dlist dlist_t;  // Create a typedef for dlist_t for ease of
use

// Function declarations for doubly linked list operations
void init_list(dlist_t* list);
void insert_head(dlist_t* list, int key);
void display(dlist_t* list);
void insert_tail(dlist_t* list, int key);
void delete_first(dlist_t* list);
void delete_last(dlist_t* list);
void delete_node(dlist_t* list, int key);
void delete_pos(dlist_t* list, int position);
void insert_pos(dlist_t* list, int key, int position);

int main()
{
```

```
    int choice, key, position;
    dlist_t list;  // Declare a doubly linked list
    init_list(&list);  // Initialize the list as empty

    // Loop to display the menu and perform operations on the list
    while(1)
    {
        display(&list);  // Display the current state of the list

        // Display menu for the user
        printf("\n1. Insert at Head\n");
        printf("2. Insert at Tail\n");
        printf("3. Display List\n");
        printf("4. Delete First Node\n");
        printf("5. Delete Last Node\n");
        printf("6. Delete Node by Value\n");
        printf("7. Delete Node by Position\n");
        printf("8. Insert Node at Given Position\n");
        printf("9. Exit\n");

        scanf("%d", &choice);  // Take user input for choice
        switch(choice)
        {
            case 1:
                printf("\nEnter the key to insert at head: ");
                scanf("%d", &key);
                insert_head(&list, key);
                break;
            case 2:
                printf("\nEnter the key to insert at tail: ");
                scanf("%d", &key);
                insert_tail(&list, key);
                break;
            case 3:
                display(&list);  // Display the list
                break;
            case 4:
                delete_first(&list);  // Delete the first node
                break;
            case 5:
```

```
                delete_last(&list);  // Delete the last node
                break;
            case 6:
                printf("\nEnter the key to delete: ");
                scanf("%d", &key);
                delete_node(&list, key);  // Delete the node with the given
key
                break;
            case 7:
                printf("\nEnter the position to delete: ");
                scanf("%d", &position);
                delete_pos(&list, position);  // Delete node at the given
position
                break;
            case 8:
                printf("\nEnter the key and position to insert: ");
                scanf("%d %d", &key, &position);
                insert_pos(&list, key, position);  // Insert node at the
given position
                break;
            case 9:
                exit(0);  // Exit the program
        }
    }
}

// Initialize the doubly linked list (set the head to NULL)
void init_list(dlist_t *list)
{
    list->head = NULL;
}

// Insert a new node at the head (beginning) of the list
void insert_head(dlist_t *list, int key)
{
    // Create a new node
    node_t *new_node = (node_t*)malloc(sizeof(node_t));
    new_node->key = key;
    new_node->next = new_node->prev = NULL;
```

```
    // If the list is empty, the new node becomes the head
    if (list->head == NULL)
    {
        list->head = new_node;
    }
    else
    {
        // If the list is not empty, insert the node at the beginning
        new_node->next = list->head;
        list->head->prev = new_node;
        list->head = new_node;
    }
}

// Display the elements of the doubly linked list
void display(dlist_t *list)
{
    node_t *current_node = list->head;

    if (current_node == NULL)
    {
        printf("\nList is empty.\n");
    }
    else
    {
        while (current_node != NULL)
        {
            printf("%d <-> ", current_node->key);
            current_node = current_node->next;
        }
        printf("NULL\n");
    }
}

// Insert a new node at the tail (end) of the list
void insert_tail(dlist_t *list, int key)
{
    node_t *new_node = (node_t*)malloc(sizeof(node_t));
    new_node->key = key;
    new_node->next = new_node->prev = NULL;
```

```
    // If the list is empty, the new node becomes the head
    if (list->head == NULL)
    {
        list->head = new_node;
    }
    else
    {
        node_t *last_node = list->head;
        // Traverse the list to find the last node
        while (last_node->next != NULL)
        {
            last_node = last_node->next;
        }

        // Insert the new node at the end of the list
        last_node->next = new_node;
        new_node->prev = last_node;
    }
}

// Delete the first node of the list
void delete_first(dlist_t *list)
{
    if (list->head == NULL)  // If the list is empty
    {
        printf("\nList is already empty.\n");
        return;
    }

    node_t *first_node = list->head;

    if (first_node->next == NULL)  // If there's only one node in the list
    {
        list->head = NULL;
    }
    else
    {
        list->head = first_node->next;
        list->head->prev = NULL;
```

```
    }

    free(first_node);  // Free the memory of the deleted node
}

// Delete the last node of the list
void delete_last(dlist_t *list)
{
    if (list->head == NULL)  // If the list is empty
    {
        printf("\nList is already empty.\n");
        return;
    }

    node_t *last_node = list->head;

    // Traverse the list to find the last node
    while (last_node->next != NULL)
    {
        last_node = last_node->next;
    }

    if (last_node->prev == NULL)  // If there's only one node
    {
        list->head = NULL;
    }
    else
    {
        last_node->prev->next = NULL;
    }

    free(last_node);  // Free the memory of the deleted node
}

// Delete a node by key (value)
void delete_node(dlist_t *list, int key)
{
    node_t *current_node = list->head;

    while (current_node != NULL && current_node->key != key)
```

```
    {
        current_node = current_node->next;
    }

    if (current_node != NULL)  // If the node with the given key is found
    {
        if (current_node->prev == NULL)  // If the node is the first node
        {
            list->head = current_node->next;
            if (list->head != NULL)  // If the list isn't empty after
deletion
                list->head->prev = NULL;
        }
        else if (current_node->next == NULL)  // If the node is the last
node
        {
            current_node->prev->next = NULL;
        }
        else  // If the node is in the middle
        {
            current_node->prev->next = current_node->next;
            current_node->next->prev = current_node->prev;
        }

        free(current_node);  // Free the memory of the deleted node
    }
    else
    {
        printf("\nNode with key %d not found.\n", key);
    }
}

// Delete a node at a given position (1-based index)
void delete_pos(dlist_t *list, int position)
{
    node_t *current_node = list->head;
    int i = 1;

    // Traverse the list to find the node at the given position
    while (current_node != NULL && i < position)
```

```
    {
        current_node = current_node->next;
        i++;
    }

    if (current_node != NULL)  // If the node at the given position is found
    {
        if (current_node->prev == NULL)  // If it's the first node
        {
            list->head = current_node->next;
            if (list->head != NULL)  // If the list isn't empty after
deletion
                list->head->prev = NULL;
        }
        else if (current_node->next == NULL)  // If it's the last node
        {
            current_node->prev->next = NULL;
        }
        else  // If it's a middle node
        {
            current_node->prev->next = current_node->next;
            current_node->next->prev = current_node->prev;
        }

        free(current_node);  // Free the memory of the deleted node
    }
    else
    {
        printf("\nInvalid position.\n");
    }
}

// Insert a new node at a given position (1-based index)
void insert_pos(dlist_t *list, int key, int position)
{
    node_t *new_node = (node_t*)malloc(sizeof(node_t));
    new_node->key = key;
    new_node->next = new_node->prev = NULL;

    if (position == 1)  // Insert at the beginning
```

```
    {
        insert_head(list, key);
    }
    else
    {
        node_t *current_node = list->head;
        int i = 1;

        // Traverse the list to find the node before the given position
        while (current_node != NULL && i < position)
        {
            current_node = current_node->next;
            i++;
        }

        if (current_node != NULL)  // If position is found
        {
            new_node->next = current_node;
            new_node->prev = current_node->prev;
            current_node->prev->next = new_node;
            current_node->prev = new_node;
        }
        else  // If position is invalid
        {
            printf("\nInvalid position.\n");
        }
    }
}
```

multilist implemenation(not there)

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

// Define a structure for column nodes in the sparse matrix
struct col_node {
    int col;             // Column index
    int data;            // Value at the (row, col) position in the matrix
    struct col_node *next_col; // Pointer to the next column node in the
```

```
same row
};

// Define a structure for row nodes in the sparse matrix
struct row_node {
    int row;                  // Row index
    struct col_node *next_col; // Pointer to the first column node of this
row
    struct row_node *next_row; // Pointer to the next row node in the list
};

// Typedefs for ease of use
typedef struct row_node rownode_t;
typedef struct col_node colnode_t;

// Function declarations
rownode_t *create_rows(int); // Create row nodes for the sparse matrix
void insert_end(rownode_t*, int, int); // Insert a column node at the end of
the row
void insert_matrix(rownode_t*, int (*)[], int, int); // Insert the matrix
elements as a multi-list
void display(rownode_t*); // Display the sparse matrix in multi-list format

int main() {
    int matrix[10][10];   // Matrix to store the input values
    int i, j, rows, cols; // Row and column variables
    rownode_t *head, *current_row; // Pointer to the head of the list and
current row

    head = NULL; // Initialize the head pointer to NULL

    // Input the dimensions of the matrix
    printf("Enter the number of rows and columns: ");
    scanf("%d %d", &rows, &cols);

    // Input the matrix elements
    printf("Enter the data for the matrix:\n");
    for(i = 0; i < rows; i++) {
        for(j = 0; j < cols; j++) {
            scanf("%d", &matrix[i][j]);
```

```
        }
    }

    // Create row nodes and initialize the sparse matrix as a multi-list
    head = create_rows(rows);
    insert_matrix(head, matrix, rows, cols);

    // Display the matrix in multi-list format
    display(head);

    return 0;
}

// Function to display the sparse matrix as a multi-list
void display(rownode_t *head) {
    colnode_t *current_col;  // Pointer to traverse the column nodes

    printf("Displaying Sparse Matrix as a Multi List:\n");

    // Check if the list is empty
    if (head == NULL) {
        printf("Empty Multi List\n");
        return;
    }

    // Traverse the row nodes and display each row with its column values
    while (head != NULL) {
        printf("Row %d -> ", head->row);
        current_col = head->next_col;

        // Traverse the column nodes for the current row
        while (current_col != NULL) {
            printf("(%d, %d) -> ", current_col->col, current_col->data);
            current_col = current_col->next_col;
        }

        head = head->next_row; // Move to the next row node
        printf("NULL\n");
    }
}
```

```
// Function to create row nodes for the sparse matrix
rownode_t* create_rows(int row_count) {
    int i;
    rownode_t *new_row, *first_row, *last_row;

    first_row = last_row = NULL;

    // Create the row nodes
    for(i = 0; i < row_count; i++) {
        new_row = (rownode_t*)malloc(sizeof(rownode_t));
        new_row->row = i;
        new_row->next_row = NULL;
        new_row->next_col = NULL;

        // Link the rows
        if (first_row == NULL) {
            first_row = last_row = new_row; // First row node
        } else {
            last_row->next_row = new_row; // Link to the next row
            last_row = new_row;           // Update the last row
        }
    }

    return first_row; // Return the head of the row list
}

// Function to insert non-zero elements from the matrix into the multi-list
void insert_matrix(rownode_t* head, int (*matrix)[10], int row_count, int
col_count) {
    int i, j;
    rownode_t *current_row = head;

    // Traverse each row and insert non-zero elements into the multi-list
    for(i = 0; i < row_count; i++) {
        for(j = 0; j < col_count; j++) {
            if (matrix[i][j] != 0) {
                insert_end(current_row, j, matrix[i][j]); // Insert non-zero
elements
            }
```

```
        }
        current_row = current_row->next_row; // Move to the next row
    }
}

// Function to insert a new column node at the end of the row
void insert_end(rownode_t* row, int col_index, int value) {
    colnode_t *new_col_node, *current_col;

    // Create a new column node
    new_col_node = (colnode_t*)malloc(sizeof(colnode_t));
    new_col_node->col = col_index;
    new_col_node->data = value;
    new_col_node->next_col = NULL;

    // If the row has no columns, insert the first column node
    if (row->next_col == NULL) {
        row->next_col = new_col_node;
    } else {
        // Traverse to the last column node and insert the new node
        current_col = row->next_col;
        while (current_col->next_col != NULL) {
            current_col = current_col->next_col;
        }
        current_col->next_col = new_col_node;
    }
}
```

stack

using array

```
#include <stdio.h>

#include <stdlib.h>



// Function prototypes
```

```
int push(int *, int *, int, int);

int pop(int *, int *);

void display(int *, int);



int main() {

int top, size, ch, k, x;

int *s;

printf("Enter the size of the stack..\n");

scanf("%d", &size);

s = malloc(sizeof(int) * size); // Dynamically allocate memory for the stack

top = -1; // Initialize top to -1, meaning the stack is empty

while (1) {

// Menu for operations

printf("\nChoose an operation:\n");

printf("1. Push\n");

printf("2. Pop\n");

printf("3. Display\n");

printf("4. Exit\n");

scanf("%d", &ch); // Read user's choice

switch (ch) {
```

```
case 1:

printf("Enter the data to push: ");

scanf("%d", &x); // Input the data to push onto the stack

k = push(s, &top, size, x); // Call the push function

if (k > 0) {

printf("Element pushed successfully\n");

}

break;

case 2:

k = pop(s, &top); // Call the pop function

if (k > 0) {

printf("Element popped = %d\n", k);

}

break;

case 3:

display(s, top); // Display the current state of the stack

break;

case 4:

free(s); // Free the dynamically allocated memory before exiting

exit(0); // Exit the program
```

```
default:
printf("Invalid choice. Please select 1, 2, 3, or 4.\n");
}
}
}


// Function to push an element onto the stack
int push(int *s, int *t, int size, int x) {
// Check for stack overflow
if (*t == size - 1) {
printf("Stack overflow..\n"); // Stack is full
return 0;
}
(*t)++; // Increment top (i.e., move top to the next empty position)
s[*t] = x; // Place the new element at the top of the stack
return 1; // Indicate success
}


// Function to pop an element from the stack
int pop(int *s, int *t) {
```

```
// Check for stack underflow (empty stack)

if (*t == -1) {

printf("Stack underflow..\n"); // Stack is empty

return 0;

}

int x = s[*t]; // Store the element at the top of the stack

(*t)--; // Decrement top to remove the element

return x; // Return the popped element

}



// Function to display the current contents of the stack

void display(int *s, int t) {

// If stack is empty

if (t == -1) {

printf("\nStack is empty..\n");

} else {

printf("\nCurrent stack elements:\n");

// Display elements from top to bottom

for (int i = t; i >= 0; i--) {

printf("%d ", s[i]); // Print each element in the stack
```

```
}

printf("\n"); // Newline after displaying all elements

}

}
```

using linked list

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

// Define the structure for a node in the stack
struct node {
    int key;           // Value of the node, stores the actual data
    struct node *link; // Pointer to the next node in the stack
};

typedef struct node node_t; // Create an alias for the node structure

// Define the structure for the stack itself
struct my_stack {
    node_t *top; // Pointer to the top node of the stack
};

typedef struct my_stack my_stack_t; // Create an alias for the stack
structure

// Function prototypes
void init(my_stack_t *stack);         // Initialize the stack (set top to
NULL)
void push(my_stack_t *stack, int value); // Push an element onto the stack
int pop(my_stack_t *stack);           // Pop an element from the stack
void display(my_stack_t *stack);      // Display the elements in the stack

int main() {
    my_stack_t stack; // Declare a stack variable
    int choice, value, popped_value; // Variables for menu choice, value to
```

```
push, and popped value

    // Initialize the stack
    init(&stack);

    while (1) {
        // Display menu and prompt user for operation choice
        printf("\nChoose an operation:\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        scanf("%d", &choice);

        // Perform the operation based on user input
        switch (choice) {
            case 1:  // Push operation
                printf("Enter the element to push: ");
                scanf("%d", &value); // Get the value to push onto the stack
                push(&stack, value); // Call the push function to add value
to the stack
                break;
            case 2:  // Pop operation
                popped_value = pop(&stack); // Call pop function to remove
and return the top element
                if (popped_value != 0) {  // Check if the stack was not
empty
                    printf("Popped element = %d\n", popped_value); //
Display popped value
                }
                break;
            case 3:  // Display operation
                display(&stack); // Call display function to print the stack
elements
                break;
            case 4:  // Exit the program
                exit(0);
            default:
                // Handle invalid choice
                printf("Invalid choice. Please select 1, 2, 3, or 4.\n");
```

```
        }
    }

    return 0;
}

// Function to initialize the stack (set top to NULL)
void init(my_stack_t *stack) {
    stack->top = NULL;  // Initially, the stack is empty, so top points to
NULL
}

// Push a new element onto the stack
void push(my_stack_t *stack, int value) {
    node_t *new_node; // Pointer for the new node to be added to the stack

    // Allocate memory for the new node
    new_node = (node_t *)malloc(sizeof(node_t));
    if (new_node == NULL) {
        // Check if memory allocation fails
        printf("Memory allocation failed.\n");
        return;
    }

    // Set the value of the new node
    new_node->key = value;

    // Link the new node to the current top of the stack
    new_node->link = stack->top;

    // Update the top of the stack to point to the new node
    stack->top = new_node;
}

// Pop an element from the stack and return its value
int pop(my_stack_t *stack) {
    node_t *top_node; // Pointer to hold the current top node
    int data;         // Variable to store the value of the popped element

    // Check if the stack is empty
```

```
    top_node = stack->top;
    if (top_node == NULL) {
        // If the stack is empty, print an error and return 0 (indicating
underflow)
        printf("\nEmpty stack...\n");
        return 0; // Return 0 to indicate underflow
    }

    // Get the value of the current top node
    data = top_node->key;

    // Update the top of the stack to point to the next node in the stack
    stack->top = top_node->link;

    // Free the memory allocated to the top node as it is no longer in use
    free(top_node);

    // Return the popped data
    return data;
}

// Display the elements in the stack
void display(my_stack_t *stack) {
    node_t *current_node; // Pointer to traverse the stack

    // Start from the top of the stack
    current_node = stack->top;

    // Check if the stack is empty
    if (current_node == NULL) {
        printf("\nEmpty stack..\n");
    } else {
        printf("\nCurrent stack elements:\n");
        // Traverse the stack and print each element
        while (current_node != NULL) {
            printf("%d -> ", current_node->key); // Print the current node's
value
            current_node = current_node->link;   // Move to the next node in
the stack
        }
```

```
        printf("NULL\n"); // Indicate the end of the stack
    }
}
```

application of stack

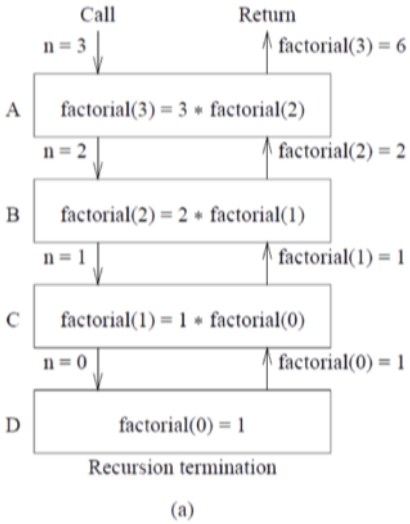
applications of stack

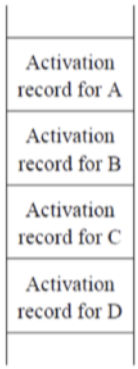
invocation of a function or activation record

recursion

factorial

```
factorial(n)
{
 int f;
 if(n==0)
     return 1;
  f=n*factorial(n-1);
return f;
}
```





multiplication

Recursive function to find the product of a*b

a*b= a if b=1;

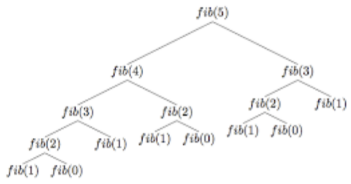
a*b = a*(b-1)+a if b >1;

To evaluate 6 * 3

6*3 = 6*2 + 6 = 6*1 + 6 + 6 = 6 + 6 + 6 = 18

```
multiply(int a, int b)
{
   int p;
    if (b==1)
     return a
   p= multiply(a,b-1) + a;
  return p;
}
```

```
fib(int n)
{
  int x,y;
  if ( (n==0) || (n==1
    return n;
  x= fib(n-1) ;
  y=fib(n-2);
  return x+y;
}
```



addition

```
int sum(int *a, int n)
//a is pointer to the array, n is the index of the last element of the array
{
 int s;
 if(n==0) // base condition
  return a[0];
 s= sum(a,n-1) + a[n]; // compute sum of n-1 elements and add the nth element
 return s;
}
```

reverse order

Recursive function to display the elements of the linked list in the reverse order

```
int display(struct node *p)
{
  if(p->next!=NULL)
     display(p->next)
  printf("%d ",p->data);
}
```

tower of hanoi

```
void tower(int n,char src,char tmp,char dst)
 {
  if(n==1)
  {
   printf("\nMove disk %d from %c to %c",n,src,dst);
   return;
  }
  tower(n-1,src,dst,tmp);
  printf("\nMove disk %d form %c to %c",n,src,dst);
  tower(n-1,tmp,src,dst);
  return;
 }
```

infix to postfix

```
#include<stdio.h>

// Function prototypes
int input_prec(char);  // Returns precedence of operator from input
int stack_prec(char);  // Returns precedence of operator from stack
char peep(char *, int);  // Returns the top element of the stack
void convert_postfix(char*, char*);  // Converts infix to postfix
void push(char *, int *, char);  // Pushes an element onto the stack
char pop(char *, int *);  // Pops an element from the stack

int main() {
    char infix[10], postfix[10];  // Arrays to hold infix and postfix
expressions

    printf("\nEnter valid Infix Expression\n");
    scanf("%s", infix);  // Input the infix expression from the user

    convert_postfix(infix, postfix);  // Convert infix to postfix
    printf("\nThe postfix equivalent = %s\n", postfix);  // Output the
postfix expression
}
```

```
// Function to convert infix expression to postfix
void convert_postfix(char* infix, char* postfix) {
    int i, j;  // `i` is for traversing infix, `j` is for building postfix
    char ch;
    char s[10];  // Stack to hold operators
    int top = -1;  // Stack pointer

    i = 0;
    j = 0;

    push(s, &top, '#');  // Push '#' as a marker for the bottom of the stack

    while (infix[i] != '\0') {  // Traverse through the infix expression
until the end
        ch = infix[i];  // Get the current character

        // If precedence of top of the stack is greater, pop to postfix
        while (stack_prec(peep(s, top)) > input_prec(ch)) {
            postfix[j++] = pop(s, &top);  // Add the popped operator to the
postfix
        }

        // If input precedence is higher, push the operator onto the stack
        if (input_prec(ch) > stack_prec(peep(s, top))) {
            push(s, &top, ch);
        } else {
            pop(s, &top);  // Handle parentheses by popping the '(' without
adding it to postfix
        }

        i++;  // Move to the next character in infix
    }

    // Pop all remaining operators in the stack to the postfix
    while (peep(s, top) != '#') {
        postfix[j++] = pop(s, &top);
    }

    postfix[j] = '\0';  // Terminate the postfix expression with null
character
```

```
}

// Function to return precedence of operators when in stack
int stack_prec(char ch) {
    switch (ch) {
        case '+':
        case '-':
            return 2;  // Low precedence for '+' and '-'
        case '*':
        case '/':
            return 4;  // Higher precedence for '*' and '/'
        case '(':
            return 0;  // '(' has the lowest precedence in the stack
        case '#':
            return -1;  // '#' marks the bottom of the stack
        default:
            return 6;  // Operand or invalid operator (default high
precedence)
    }
}

// Function to return precedence of operators when from input
int input_prec(char ch) {
    switch (ch) {
        case '+':
        case '-':
            return 1;  // Low precedence for '+' and '-'
        case '*':
        case '/':
            return 3;  // Higher precedence for '*' and '/'
        case '(':
            return 7;  // '(' has the highest precedence when read from
input
        case ')':
            return 0;  // ')' has the lowest precedence for matching
parenthesis
        default:
            return 5;  // Operand or invalid operator (default high
precedence)
    }
```

```
}

// Function to return the top element of the stack
char peep(char *s, int top) {
    return s[top];  // Return the element at the top of the stack
}

// Function to push an element onto the stack
void push(char *s, int *top, char ch) {
    (*top)++;  // Increment the top pointer
    s[*top] = ch;  // Add the element to the stack
}

// Function to pop an element from the stack
char pop(char *s, int *top) {
    char x;
    x = s[*top];  // Get the element from the top of the stack
    --(*top);  // Decrement the top pointer
    return x;  // Return the popped element
}
```

infix to prefix

```
#include<stdio.h>

#include<string.h>



// Function prototypes

int input_prec(char); // Returns precedence of operator from input

int stack_prec(char); // Returns precedence of operator from stack

char peep(char *, int); // Returns the top element of the stack

void convert_postfix(char*, char*); // Converts infix to postfix
```

```
void push(char *, int *, char); // Pushes an element onto the stack

char pop(char *, int *); // Pops an element from the stack

void reverse_string(char *, char *); // Reverses the input string with special handling for parentheses



int main() {

char infix[100], prefix[100], reverse[100], postfix[100];



printf("\nEnter valid Infix Expression\n");

scanf("%s", infix); // Input the infix expression from the user



reverse_string(infix, reverse); // Reverse the infix expression

printf("Reversed = %s\n", reverse); // Display the reversed expression



convert_postfix(reverse, postfix); // Convert the reversed infix to postfix

reverse_string(postfix, prefix); // Reverse the postfix to get the prefix



printf("\nThe prefix equivalent = %s\n", prefix); // Output the prefix expression

}
```

```
// Function to reverse the infix expression and swap parentheses

void reverse_string(char *a, char *b) {

int i, j;

i = strlen(a) - 1; // Start from the end of the string `a`

j = 0;



// Traverse the string `a` in reverse

while (i >= 0) {

// Swap '(' with ')' and vice versa

if (a[i] == '(')

b[j++] = ')';

else if (a[i] == ')')

b[j++] = '(';

else

b[j++] = a[i]; // Copy other characters as is

i--;

}

b[j] = '\0'; // Terminate the reversed string with null character

}
```

```
// Function to convert infix to postfix

void convert_postfix(char *infix, char *postfix) {

char s[100]; // Stack to hold operators

int top = -1, i = 0, j = 0;

char ch;



push(s, &top, '#'); // Push '#' as a marker for the bottom of the stack



// Traverse the infix expression

while (infix[i] != '\0') {

ch = infix[i]; // Get the current character



// Compare input precedence with stack precedence

// Pop from the stack to postfix while the stack precedence is greater

while (input_prec(ch) < stack_prec(peep(s, top)))

postfix[j++] = pop(s, &top);



// If input precedence is greater, push the operator onto the stack

if (input_prec(ch) > stack_prec(peep(s, top)))

push(s, &top, ch);
```

```
else

pop(s, &top); // Handle parentheses by popping the '(' without adding it to
postfix



i++; // Move to the next character in infix

}



// Pop all remaining operators in the stack to the postfix expression

while (peep(s, top) != '#')

postfix[j++] = pop(s, &top);



postfix[j] = '\0'; // Terminate the postfix expression with null character

}



// Function to return input precedence of operators

int input_prec(char ch) {

switch (ch) {

case '+':

case '-': return 2; // Low precedence for '+' and '-'

case '*':

case '/': return 4; // Higher precedence for '*' and '/'
```

```
case '$': return 5; // High precedence for exponentiation operator

case '(': return 9; // '(' has the highest precedence when read from input

case ')': return 0; // ')' has the lowest precedence for matching
parenthesis

default: return 7; // Default precedence for operands

}

}



// Function to return stack precedence of operators

int stack_prec(char ch) {

switch (ch) {

case '+':

case '-': return 1; // Low precedence for '+' and '-'

case '*':

case '/': return 3; // Higher precedence for '*' and '/'

case '$': return 6; // High precedence for exponentiation operator

case '(': return 0; // '(' has the lowest precedence in the stack

case '#': return -1; // '#' marks the bottom of the stack

default: return 8; // Default precedence for operands

}
```

```
}



// Function to return the top element of the stack

char peep(char *s, int t) {

return s[t]; // Return the element at the top of the stack

}



// Function to push an element onto the stack

void push(char *s, int *t, char x) {

++*t; // Increment the top pointer

s[*t] = x; // Add the element to the stack

}



// Function to pop an element from the stack

char pop(char *s, int *t) {

char x = s[*t]; // Get the element from the top of the stack

--*t; // Decrement the top pointer

return x; // Return the popped element

}
```

evaluation of postfix expression

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

// Structure for stack
struct my_stack {
    int *s;  // Pointer to hold the stack array
    int size;  // Size of the stack
    int top;   // Index of the top element in the stack
};

// Type definition for convenience
typedef struct my_stack my_stack_t;

// Function declarations
void init_stk(my_stack_t *);  // To initialize the stack
void push(my_stack_t *, int);  // To push an element onto the stack
int pop(my_stack_t *);  // To pop an element from the stack
void display(my_stack_t*);  // To display the stack contents
int postfix_eval(char* postfix);  // Function to evaluate a given postfix
expression
int isoper(char ch);  // Function to check if a character is an operator

int main() {
    char postfix[100];

    // Input the postfix expression from the user
    printf("\nEnter the postfix expression (e.g. abc*+):\n");
    scanf("%s", postfix);

    // Evaluate the postfix expression
    int result = postfix_eval(postfix);

    // Output the result of the evaluation
    printf("\nThe result = %d\n", result);

    return 0;
}

// Function to evaluate the postfix expression
int postfix_eval(char* postfix) {
```

```
    int i = 0, result, op1, op2, a;
    char ch;
    my_stack_t st; // Declare a stack

    init_stk(&st); // Initialize the stack

    // Loop through the postfix expression
    while (postfix[i] != '\0') {
        ch = postfix[i];

        // Check if the character is an operator
        if (isoper(ch)) {
            // Pop two operands from the stack
            op1 = pop(&st);
            op2 = pop(&st);

            // Perform the operation based on the operator
            switch(ch) {
                case '+':
                    result = op1 + op2;
                    break;
                case '-':
                    result = op2 - op1; // Order is important for
subtraction
                    break;
                case '*':
                    result = op1 * op2;
                    break;
                case '/':
                    result = op2 / op1; // Order is important for division
                    break;
            }

            // Push the result back to the stack
            push(&st, result);
        } else {
            // If not an operator, it must be an operand. Ask user for its
value.
            printf("%c =", ch);
            scanf("%d", &a);
```

```
            push(&st, a); // Push the operand value onto the stack
        }
        i++;
    }

    return pop(&st); // Return the final result from the stack
}

// Function to check if a character is an operator
int isoper(char ch) {
    if ((ch == '+') || (ch == '-') || (ch == '*') || (ch == '/'))
        return 1;  // It's an operator
    return 0;  // It's not an operator
}

// Function to initialize the stack
void init_stk(my_stack_t *st) {
    st->size = 100;  // Set the stack size to 100
    st->top = -1;  // Initialize top to -1 (empty stack)
    st->s = (int*) malloc(st->size * sizeof(int));  // Allocate memory for
the stack array
}

// Function to push an element onto the stack
void push(my_stack_t *st, int val) {
    if (st->top == st->size - 1) {
        printf("Stack overflow\n");
    } else {
        st->top++;  // Increment the top index
        st->s[st->top] = val;  // Place the value at the new top position
    }
}

// Function to pop an element from the stack
int pop(my_stack_t *st) {
    if (st->top == -1) {
        printf("Stack underflow\n");
        return -1;  // Return -1 if stack is empty
    } else {
        int val = st->s[st->top];  // Get the value from the top of the
```

```
stack
        st->top--;  // Decrement the top index
        return val;  // Return the popped value
    }
}

// Function to display the stack contents (for debugging purposes)
void display(my_stack_t *st) {
    if (st->top == -1) {
        printf("Stack is empty\n");
    } else {
        printf("Stack contents: ");
        for (int i = 0; i <= st->top; i++) {
            printf("%d ", st->s[i]);
        }
        printf("\n");
    }
}
```

parenthesis matching

```
#include<stdio.h>

// Function declarations
void push(char *, int*, char);  // Push function for stack
char pop(char* s, int*);        // Pop function for stack
int isempty(int);               // Check if stack is empty
int match(char *);              // Function to match parentheses

int main() {
    char expr[10];  // Array to hold the input expression
    printf("Enter the expression\n");
    scanf("%s", expr);  // Get the input expression from user

    int result = match(expr);  // Check if parentheses match
    if(result)  // If result is true, parentheses match correctly
        printf("Matching is correct\n");
    else  // If result is false, there is a mismatch
        printf("Matching fails\n");
```

```
    return 0;
}

// Function to check if parentheses in the expression are matched
int match(char *str) {
    char s[10];  // Stack to hold opening parentheses
    char ch, x;  // 'ch' is the current character, 'x' is popped from stack
    int top = -1;  // Initialize top of stack as -1 (empty stack)
    int i = 0;

    // Loop through the expression until the end
    while(str[i] != '\0') {
        ch = str[i];  // Get current character

        // Check if the current character is an opening or closing bracket
        switch(ch) {
            case '(':  // Opening parenthesis
            case '{':  // Opening curly brace
            case '[':  // Opening square bracket
                push(s, &top, ch);  // Push it to the stack
                break;

            case ')':  // Closing parenthesis
                if(!isempty(top)) {  // Check if stack is not empty
                    x = pop(s, &top);  // Pop from stack
                    if(x != '(')  // Check if the matching opening bracket
is '('
                        return 0;  // Parentheses mismatch
                } else {
                    return 0;  // Extra closing parenthesis found
                }
                break;

            case '}':  // Closing curly brace
                if(!isempty(top)) {  // Check if stack is not empty
                    x = pop(s, &top);  // Pop from stack
                    if(x != '{')  // Check if the matching opening bracket
is '{'
                        return 0;  // Parentheses mismatch
```

```
                } else {
                    return 0;  // Extra closing parenthesis found
                }
                break;

            case ']':  // Closing square bracket
                if(!isempty(top)) {  // Check if stack is not empty
                    x = pop(s, &top);  // Pop from stack
                    if(x != '[')  // Check if the matching opening bracket
is '['
                        return 0;  // Parentheses mismatch
                } else {
                    return 0;  // Extra closing parenthesis found
                }
                break;
        }  // End switch
        i++;  // Move to the next character
    }  // End while

    // After loop ends, check if the stack is empty
    if(isempty(top))
        return 1;  // If stack is empty, parentheses match correctly
    return 0;  // If stack is not empty, there are extra opening parentheses
}

// Function to push a character onto the stack
void push(char *s, int *t, char ch) {
    (*t)++;       // Increment top index
    s[*t] = ch;   // Place the character at the new top position
}

// Function to pop a character from the stack
char pop(char *s, int *t) {
    char x;
    x = s[*t];    // Get the character at the top of the stack
    (*t)--;       // Decrement the top index
    return x;     // Return the popped character
}

// Function to check if the stack is empty
```

```
int isempty(int t) {
    if(t == -1)  // If top is -1, stack is empty
        return 1;
    return 0;  // Stack is not empty
}
```