```c
// 1. Implement Singly Linked List ADT. Insert at Beginning, Delete at End
#include<stdio.h>
#include<stdlib.h>
#define MAX 100
// Structure for a node in the linked list
struct Node {
   int data;
   struct Node *next;
};
// Global variables for linked list management
struct Node *head = NULL;
// Function to insert a node at the beginning of the linked list
void insertAtBeginning(int value) {
   struct Node *newNode = (struct Node *) malloc(sizeof(struct Node));
   newNode->data = value;
   newNode->next = head;
   head = newNode;
}
// Function to delete a node from the end of the linked list
void deleteAtEnd() {
   if (head == NULL) {
      printf("-1\n");
      return;
   }
   if (head->next == NULL) {
      free(head);
      head = NULL;
      return;
   }
   struct Node *temp = head;
```

```
    while (temp->next->next != NULL) {
        temp = temp->next;
    }
    free(temp->next);
    temp->next = NULL;
}
// Function to display the linked list
void display() {
    struct Node *temp = head;
    if (temp == NULL) {
        printf("-1\n");
        return;
    }
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
// Main function to run the menu-driven program
int main() {
    int choice, value;
    do {
        scanf("%d", &choice);
        switch (choice) {
        case 1:
            scanf("%d", &value);
            insertAtBeginning(value);
            break;
        case 2:
```

```
            deleteAtEnd();
            break;
        case 3:
            display();
            break;
        case 4:
            break;
        default:
            printf("-1\n");
        }
    } while (choice != 4);
    return 0;
}
```

```c
// 2. Implement Singly Linked List ADT. Insert at end, Delete at beginning
#include<stdio.h>
#include <stdlib.h>
// Node structure representing each element in the list
struct Node {
  int data; // Data field
  struct Node *next; // Pointer to the next node
};
// Function to create a new node
struct Node *createNode(int data) {
  struct Node *newNode = (struct Node *) malloc(sizeof(struct Node)); // Allocate memory
  newNode->data = data; // Assign data
  newNode->next = NULL; // Initialize next as NULL
  return newNode; // Return the new node
}
// Function to insert a node at the end of the list
void insertAtEnd(struct Node **head, int data) {
  struct Node *newNode = createNode(data); // Create a new node
  if (*head == NULL) { // If the list is empty
    *head = newNode; // Make the new node the head
  } else {
    struct Node *current = *head; // Start from the head
    while (current->next != NULL) { // Traverse to the end
      current = current->next;
    }
    current->next = newNode; // Link the new node
  }
}
// Function to delete a node from the beginning of the list
```

```c
void deleteAtBeginning(struct Node **head) {
  if (*head == NULL) { // If the list is empty
    printf("-1\n");
    return;
  }
  struct Node *temp = *head; // Store the head node
  *head = (*head)->next; // Move head to the next node
  free(temp); // Free the memory of the deleted node
}
// Function to display the list
void display(struct Node *head) {
  if (head == NULL) {
    printf("-1\n");
    return;
  }
  struct Node *current = head; // Start from the head
  while (current != NULL) { // Traverse the list
    printf("%d ", current->data); // Print data
    current = current->next;
  }
  printf("\n"); // End of the list
}
// Main function to test the singly linked list with user input
int main() {
  struct Node *head = NULL; // Initialize head as NULL
  int choice, data; // Variables for user choice and data input
  do {
    scanf("%d", &choice); // Get user choice
    switch (choice) {
    case 1: // Insert at End
```

```c
            scanf("%d", &data); // Get the value to insert
            insertAtEnd(&head, data); // Insert the value
            break;
        case 2: // Delete from Beginning
            deleteAtBeginning(&head); // Delete the head node
            break;
        case 3: // Display List
            display(head); // Display the list
            break;
        case 4: // Exit
            break;
        default: // Invalid choice
            break;
        }
    } while (choice != 4); // Continue until user chooses to exit
    // Cleanup: Free remaining nodes
    while (head != NULL) {
        deleteAtBeginning(&head); // Delete all nodes
    }
    return 0;
}
```

```c
// 3. Implement Binary Search Tree ADT using Linked List
#include <stdio.h>
#include <stdlib.h>
// Define the structure of a node in the BST
struct Node {
   int data;
   struct Node *left;
   struct Node *right;
};
// Function to create a new node
struct Node *createNode(int value) {
   struct Node *newNode = (struct Node *) malloc(sizeof(struct Node));
   newNode->data = value;
   newNode->left = newNode->right = NULL;
   return newNode;
}
// Function to insert a node in the BST
struct Node *insert(struct Node *root, int value) {
   if (root == NULL) {
      return createNode(value);
   }
   if (value < root->data) {
      root->left = insert(root->left, value);
   } else if (value > root->data) {
      root->right = insert(root->right, value);
   }
   return root;
}
// Function for in-order traversal (left, root, right)
void inorderTraversal(struct Node *root) {
```

```c
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}
int main() {
    struct Node *root = NULL;
    int n, value;
    // Taking the number of nodes as input
    // printf("Enter the number of nodes: ");
    scanf("%d", &n);
    // Taking node values as input
    // printf("Enter the node values:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &value);
        root = insert(root, value);
    }
    // Performing in-order traversal of the BST
    // printf("In-order traversal of the BST: ");
    inorderTraversal(root);
    printf("\n");
    return 0;
}
```

```c
// 4. Implement Stack ADT using an array
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define max 10 // max size of stack is 10

int stack[max]; // array to store stack elements
int top = -1; // top points to the index of the top element, -1 means stack is empty

// Function to push value into stack
void push(int val) {
  if (top == max - 1) { // if top reaches max-1, stack is full
    printf("-1\n"); // print -1 if overflow
    return;
  } else {
    stack[++top] = val; // increase top and insert value
  }
}

// Function to pop value from stack
int pop() {
  if (top == -1) { // if top = -1, stack is empty
    printf("-1\n"); // print -1 if underflow
    return -1;
  } else {
    return stack[top--]; // return top value and decrease top
  }
}

// Function to see top element of stack
```

```c
int peek() {
   if (top == -1) { // if stack is empty
      printf("-1\n"); // print -1
      return -1;
   }
   return stack[top]; // return top element without removing it
}

int main() {
   int val = 0, choice;
   do {
      // take user choice input (1=push, 2=pop, 3=peek, 4=exit)
      scanf("%d", &choice);
      switch (choice) {
      case 1:
         // push operation
         scanf("%d", &val); // take value to push
         push(val);
         break;
      case 2:
         // pop operation
         val = pop(); // pop element
         printf("%d\n", val); // print popped value
         break;
      case 3:
         // peek operation
         val = peek(); // get top element
         if (val != -1) {
            printf("%d\n", val);
         }
```

```
            break;
        case 4:
            exit(0); // exit program
            break;
        default:
            printf("-1\n"); // invalid choice
            break;
        }
    } while (choice != 4); // loop until choice is 4
    return 0;
}
```

```
// 5. Convert an Infix expression to Postfix expression using stack ADT
#include <stdio.h>
#include <string.h>
#define MAX 100

char stack[MAX];
int top = -1;
char infix[MAX], postfix[MAX];

// Push element on stack
void push(char ch) {
  if (top == MAX - 1) {
    printf("-1\n");
    return;
  }
  stack[++top] = ch;
}

// Pop element from stack
char pop() {
  if (top == -1) {
    printf("-1\n");
    return '\0';
  }
  return stack[top--];
}

// Check if stack is empty
int isEmpty() {
  return (top == -1);
```

```
}

// Return priority of operators
int priority(char op) {
  if (op == '^') return 3;
  if (op == '*' || op == '/') return 2;
  if (op == '+' || op == '-') return 1;
  return 0;
}

// Convert infix to postfix
void infixToPostfix() {
  int i, j = 0;
  char ch, temp;

  for (i = 0; i < strlen(infix); i++) {
    ch = infix[i];

    // Ignore newline
    if (ch == '\n')
      continue;

    // If operand, add to postfix
    if ((ch >= '0' && ch <= '9') || (ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z')) {
      postfix[j++] = ch;
    }
    // Push '(' on stack
    else if (ch == '(') {
      push(ch);
    }
```

```
        // Pop until '(' found
        else if (ch == ')') {
            while (!isEmpty() && (temp = pop()) != '(') {
                postfix[j++] = temp;
            }
        }
        // Operator encountered
        else {
            while (!isEmpty() && priority(stack[top]) >= priority(ch)) {
                postfix[j++] = pop();
            }
            push(ch);
        }
    }

    // Pop remaining operators
    while (!isEmpty()) {
        postfix[j++] = pop();
    }

    postfix[j] = '\0'; // Null terminate string
}

// Print postfix
void printPostfix() {
    printf("%s", postfix);
}

int main() {
    // Input infix expression
```

```
    fgets(infix, MAX, stdin);

    // Convert & print
    infixToPostfix();
    printPostfix();

    return 0;
}
```

```c
// 6. Evaluate Postfix Expression using Stack ADT
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <math.h>   // for pow()

#define MAX 100

int stack[MAX];
int top = -1;

// Push value to stack
void push(int value) {
  if (top == MAX - 1) {
    printf("-1\n"); // stack overflow
    exit(1);
  }
  stack[++top] = value;
}

// Pop value from stack
int pop() {
  if (top == -1) {
    printf("-1\n"); // stack underflow
    exit(1);
  }
  return stack[top--];
}
```

```
// Perform the given operation
int calculate(char op, int a, int b) {
  switch (op) {
    case '+': return a + b;
    case '-': return a - b;
    case '*': return a * b;
    case '/': return a / b;   // assuming valid division input
    case '^': return pow(a, b); // power a^b
    default: return 0;       // invalid operator fallback
  }
}

int main() {
  char expr[MAX];
  fgets(expr, sizeof(expr), stdin); // read postfix expression

  for (int i = 0; i < strlen(expr); i++) {
    char ch = expr[i];

    if (isspace(ch))
      continue; // ignore spaces/newlines

    if (isdigit(ch)) {
      push(ch - '0'); // convert char digit to number
    }
    else {
      // for operator, need at least 2 values
      if (top < 1) {
        printf("-1\n");
        return 1;
```

```
        }

        int b = pop();  // second operand
        int a = pop();  // first operand
        int result = calculate(ch, a, b);
        push(result);
      }
    }

    // After full evaluation, only one result must remain
    if (top != 0) {
      printf("-1\n");
      return 1;
    }

    printf("%d\n", pop());
    return 0;
}
```

```c
// 7. Implement Linear Queue ADT using an array
#include <stdio.h>
#define MAX 100

int queue[MAX];
int front = -1, rear = -1;

// Add element to queue
void enqueue(int value) {
  if (rear == MAX - 1) {
    printf("-1\n"); // Queue full
    return;
  }
  if (front == -1) {
    front = 0;     // First element entry
  }
  queue[++rear] = value;
}

// Remove element from queue
void dequeue() {
  if (front == -1 || front > rear) {
    printf("-1\n"); // Queue empty
    return;
  }
  front++; // Move front forward
}

// Display queue elements
void display() {
```

```c
    if (front == -1 || front > rear) {
        printf("-1\n"); // Queue empty
        return;
    }
    for (int i = front; i <= rear; i++) {
        printf("%d ", queue[i]);
    }
    printf("\n");
}

int main() {
    int choice, value;
    do {
        // Menu input: 1 = enqueue, 2 = dequeue, 3 = display, 4 = exit
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                scanf("%d", &value);
                enqueue(value);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                display();
                break;
            case 4:
                break;
            default:
                printf("-1\n"); // Invalid choice
```

```
        }
    } while (choice != 4);
    return 0;
}
```

```c
// 8. Implement Graph Traversal techniques: Depth First Search
#include <stdio.h>
#define MAX 100

int graph[MAX][MAX];   // Adjacency matrix
int visited[MAX];      // Track visited nodes
int n;                 // Number of vertices

// DFS function
void dfs(int vertex) {
    visited[vertex] = 1;     // Mark vertex as visited
    printf("%d ", vertex);   // Print current vertex

    for (int i = 0; i < n; i++) {
        // If edge exists AND neighbor is not visited
        if (graph[vertex][i] == 1 && visited[i] == 0) {
            dfs(i);        // Visit next connected vertex
        }
    }
}

int main() {
    scanf("%d", &n);   // Input number of vertices

    // Input adjacency matrix
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
        }
    }
```

```c
    int start;
    scanf("%d", &start); // Input starting vertex

    // Initialize visited array
    for (int i = 0; i < n; i++) {
        visited[i] = 0;
    }

    dfs(start); // Start DFS

    return 0;
}
```

```c
#include <stdio.h>
#define MAX 100

int adj[MAX][MAX];
int visited[MAX];

int queue[MAX];
int front = 0, rear = 0;

// Check if queue is empty
int isEmpty() {
  return front == rear;
}

// Check if queue is full
int isFull() {
  return rear == MAX;
}

// Add element to queue (enqueue)
void enqueue(int x) {
  if (isFull()) {
    printf("-1\n"); // Queue full
    return;
  }
  queue[rear++] = x;
}

// Remove element from queue (dequeue)
int dequeue() {
```

```c
    if (isEmpty()) {
        printf("-1\n"); // Queue empty
        return -1;
    }
    return queue[front++];
}

void bfs(int start, int n) {
    enqueue(start);
    visited[start] = 1;

    while (!isEmpty()) {
        int current = dequeue();
        printf("%d ", current);

        for (int i = 0; i < n; i++) {
            if (adj[current][i] == 1 && visited[i] == 0) {
                enqueue(i);
                visited[i] = 1;
            }
        }
    }
}

int main() {
    int n, e, u, v, start;

    scanf("%d", &n);
    scanf("%d", &e);
```

```
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            adj[i][j] = 0;
        visited[i] = 0;
    }

    for (int i = 0; i < e; i++) {
        scanf("%d %d", &u, &v);
        adj[u][v] = 1;
        adj[v][u] = 1;
    }

    scanf("%d", &start);

    bfs(start, n);

    return 0;
}
```

```c
// 10. Implement Circular Linked List ADT
#include <stdio.h>
#include <stdlib.h>

// Node structure
struct Node {
  int data;
  struct Node* next;
};

// Create a new node
struct Node* createNode(int value) {
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = value;
  newNode->next = NULL;
  return newNode;
}

// Insert a node at the end of the circular linked list
struct Node* insert(struct Node* last, int value) {
  struct Node* newNode = createNode(value);

  // If list is empty — new node points to itself
  if (last == NULL) {
    newNode->next = newNode;
    return newNode;
  }

  // Insert node after last and update last
  newNode->next = last->next;
```

```
    last->next = newNode;
    return newNode;         // New last node
}

// Display circular linked list
void display(struct Node* last) {
    if (last == NULL) {
        printf("List is empty\n");
        return;
    }

    struct Node* temp = last->next; // Start from first node

    do {
        printf("%d ", temp->data);
        temp = temp->next;
    } while (temp != last->next);   // Stop when back to start

    printf("\n");
}

int main() {
    struct Node* last = NULL;
    int n, value;

    // Input number of nodes
    printf("Enter the number of nodes: ");
    scanf("%d", &n);

    // Input values and insert nodes
```

```
    printf("Enter the node values:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &value);
        last = insert(last, value);
    }

    // Display list
    printf("The circular linked list is: ");
    display(last);

    return 0;
}
```