1) Write a program to evaluate the postfix expression using stack

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

#define MAX_SIZE 100

// Stack structure
struct Stack {
   int top;
   int items[MAX_SIZE];
};

// Function to initialize the stack
void initialize(struct Stack *s) {
   s->top = -1;
}

// Function to check if the stack is empty
int isEmpty(struct Stack *s) {
   return s->top == -1;
}

// Function to push an element onto the stack
void push(struct Stack *s, int value) {
   if (s->top == MAX_SIZE - 1) {
      printf("Stack overflow\n");
      exit(EXIT_FAILURE);
   } else {
```

```
        s->items[++s->top] = value;
    }
}

// Function to pop an element from the stack
int pop(struct Stack *s) {
    if (isEmpty(s)) {
        printf("Stack underflow\n");
        exit(EXIT_FAILURE);
    } else {
        return s->items[s->top--];
    }
}

// Function to evaluate postfix expression
int evaluatePostfix(char *expression) {
    struct Stack stack;
    initialize(&stack);

    for (int i = 0; expression[i] != '\0'; ++i) {
        if (isdigit(expression[i])) {
            push(&stack, expression[i] - '0');
        } else {
            int operand2 = pop(&stack);
            int operand1 = pop(&stack);
            switch (expression[i]) {
                case '+':
                    push(&stack, operand1 + operand2);
                    break;
```

```
            case '-':
                push(&stack, operand1 - operand2);
                break;
            case '*':
                push(&stack, operand1 * operand2);
                break;
            case '/':
                push(&stack, operand1 / operand2);
                break;
            default:
                printf("Invalid operator: %c\n", expression[i]);
                exit(EXIT_FAILURE);
        }
    }
}

    return pop(&stack);
}

int main() {
    char expression[MAX_SIZE];

    printf("Enter a postfix expression: ");
    fgets(expression, MAX_SIZE, stdin);

    // Remove trailing newline character from fgets
    expression[strcspn(expression, "\n")] = '\0';

    int result = evaluatePostfix(expression);
```

```
    printf("Result: %d\n", result);

    return 0;
}
```

2) Write a program to convert an expression from infix to postfix using stack.

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

#define MAX_SIZE 100

// Stack structure
struct Stack {
    int top;
    char items[MAX_SIZE];
};

// Function to initialize the stack
void initialize(struct Stack *s) {
    s->top = -1;
}

// Function to check if the stack is empty
int isEmpty(struct Stack *s) {
    return s->top == -1;
}

// Function to push an element onto the stack
```

```
void push(struct Stack *s, char value) {
    if (s->top == MAX_SIZE - 1) {
        printf("Stack overflow\n");
        exit(EXIT_FAILURE);
    } else {
        s->items[++s->top] = value;
    }
}

// Function to pop an element from the stack
char pop(struct Stack *s) {
    if (isEmpty(s)) {
        printf("Stack underflow\n");
        exit(EXIT_FAILURE);
    } else {
        return s->items[s->top--];
    }
}

// Function to get the precedence of an operator
int getPrecedence(char op) {
    switch (op) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
```

```
            return 3;
        default:
            return 0;
    }
}

// Function to convert infix expression to postfix expression
void infixToPostfix(char *infix, char *postfix) {
    struct Stack stack;
    initialize(&stack);

    int i, j;
    i = j = 0;

    while (infix[i] != '\0') {
        if (isalnum(infix[i])) {
            postfix[j++] = infix[i++];
        } else if (infix[i] == '(') {
            push(&stack, infix[i++]);
        } else if (infix[i] == ')') {
            while (!isEmpty(&stack) && stack.items[stack.top] != '(') {
                postfix[j++] = pop(&stack);
            }
            if (!isEmpty(&stack) && stack.items[stack.top] != '(') {
                printf("Invalid infix expression\n");
                exit(EXIT_FAILURE);
            } else {
                pop(&stack); // Remove '(' from the stack
                i++;
```

```
            }
        } else {
            while (!isEmpty(&stack) && getPrecedence(infix[i]) <= getPrecedence(stack.items[stack.top])) {
                postfix[j++] = pop(&stack);
            }
            push(&stack, infix[i++]);
        }
    }

    while (!isEmpty(&stack)) {
        postfix[j++] = pop(&stack);
    }

    postfix[j] = '\0'; // Null-terminate the postfix expression
}

int main() {
    char infix[MAX_SIZE];
    char postfix[MAX_SIZE];

    printf("Enter an infix expression: ");
    fgets(infix, MAX_SIZE, stdin);

    // Remove trailing newline character from fgets
    infix[strcspn(infix, "\n")] = '\0';

    infixToPostfix(infix, postfix);

    printf("Postfix expression: %s\n", postfix);
```

```
    return 0;
}
```

3) Write a Program to check for Balanced Parenthesis using stack.

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

#define MAX_SIZE 100

// Stack structure
struct Stack {
    int top;
    char items[MAX_SIZE];
};

// Function to initialize the stack
void initialize(struct Stack *s) {
    s->top = -1;
}

// Function to check if the stack is empty
int isEmpty(struct Stack *s) {
    return s->top == -1;
}

// Function to push an element onto the stack
void push(struct Stack *s, char value) {
    if (s->top == MAX_SIZE - 1) {
```

```
        printf("Stack overflow\n");
        exit(EXIT_FAILURE);
    } else {
        s->items[++s->top] = value;
    }
}

// Function to pop an element from the stack
char pop(struct Stack *s) {
    if (isEmpty(s)) {
        printf("Stack underflow\n");
        exit(EXIT_FAILURE);
    } else {
        return s->items[s->top--];
    }
}

// Function to check for balanced parentheses
int isBalanced(char *expression) {
    struct Stack stack;
    initialize(&stack);

    for (int i = 0; expression[i] != '\0'; ++i) {
        if (expression[i] == '(' || expression[i] == '[' || expression[i] == '{') {
            push(&stack, expression[i]);
        } else if (expression[i] == ')' || expression[i] == ']' || expression[i] == '}') {
            if (isEmpty(&stack)) {
                return 0; // Unbalanced if closing parenthesis with no corresponding opening parenthesis
            }
```

```
            char topElement = pop(&stack);

            if ((expression[i] == ')' && topElement != '(') ||
                (expression[i] == ']' && topElement != '[') ||
                (expression[i] == '}' && topElement != '{')) {
                return 0; // Unbalanced if closing parenthesis does not match the corresponding opening
parenthesis
            }
        }
    }

    return isEmpty(&stack); // Balanced if the stack is empty at the end
}

int main() {
    char expression[MAX_SIZE];

    printf("Enter an expression: ");
    fgets(expression, MAX_SIZE, stdin);

    // Remove trailing newline character from fgets
    expression[strcspn(expression, "\n")] = '\0';

    if (isBalanced(expression)) {
        printf("Balanced parentheses\n");
    } else {
        printf("Unbalanced parentheses\n");
    }
```

```
    return 0;
}
```

4) Write a program to implement queue using linked list

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

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

// Queue structure
struct Queue {
    struct Node* front;
    struct Node* rear;
};

// Function to initialize an empty queue
void initializeQueue(struct Queue* q) {
    q->front = q->rear = NULL;
}

// Function to check if the queue is empty
int isEmpty(struct Queue* q) {
    return q->front == NULL;
}
```

```c
// Function to enqueue (insert) a new element into the queue
void enqueue(struct Queue* q, int value) {
    // Create a new node
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = value;
    newNode->next = NULL;

    // If the queue is empty, set both front and rear to the new node
    if (isEmpty(q)) {
        q->front = q->rear = newNode;
    } else {
        // Otherwise, add the new node to the rear and update the rear pointer
        q->rear->next = newNode;
        q->rear = newNode;
    }
}

// Function to dequeue (remove) an element from the queue
int dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue underflow\n");
        exit(EXIT_FAILURE);
    }

    // Get the data from the front node
```

```c
    int data = q->front->data;

    // Move the front pointer to the next node
    struct Node* temp = q->front;
    q->front = q->front->next;

    // If the queue becomes empty after dequeue, update the rear pointer to NULL
    if (q->front == NULL) {
        q->rear = NULL;
    }

    // Free the memory of the dequeued node
    free(temp);

    return data;
}

// Function to display the elements in the queue
void displayQueue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return;
    }

    struct Node* current = q->front;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
```

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

int main() {
    struct Queue myQueue;
    initializeQueue(&myQueue);

    enqueue(&myQueue, 10);
    enqueue(&myQueue, 20);
    enqueue(&myQueue, 30);

    printf("Queue after enqueue: ");
    displayQueue(&myQueue);

    int dequeuedValue = dequeue(&myQueue);
    printf("Dequeued element: %d\n", dequeuedValue);

    printf("Queue after dequeue: ");
    displayQueue(&myQueue);

    return 0;
}
```

5) Write a program to implement priority queue using list.

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

// Node structure
struct Node {
    int data;
```

```c
    int priority;
    struct Node* next;
};

// Priority Queue structure
struct PriorityQueue {
    struct Node* front;
};

// Function to initialize an empty priority queue
void initializePriorityQueue(struct PriorityQueue* pq) {
    pq->front = NULL;
}

// Function to check if the priority queue is empty
int isEmpty(struct PriorityQueue* pq) {
    return pq->front == NULL;
}

// Function to insert a new element into the priority queue based on priority
void enqueue(struct PriorityQueue* pq, int value, int priority) {
    // Create a new node
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = value;
    newNode->priority = priority;
```

```
    newNode->next = NULL;

    // If the priority queue is empty or the new node has higher priority, insert at the front
    if (isEmpty(pq) || priority > pq->front->priority) {
        newNode->next = pq->front;
        pq->front = newNode;
    } else {
        // Otherwise, find the correct position based on priority and insert
        struct Node* current = pq->front;
        while (current->next != NULL && priority <= current->next->priority) {
            current = current->next;
        }
        newNode->next = current->next;
        current->next = newNode;
    }
}

// Function to remove and return the element with the highest priority from the priority queue
int dequeue(struct PriorityQueue* pq) {
    if (isEmpty(pq)) {
        printf("Priority queue underflow\n");
        exit(EXIT_FAILURE);
    }

    // Get the data from the front node
    int data = pq->front->data;

    // Move the front pointer to the next node
    struct Node* temp = pq->front;
```

```
    pq->front = pq->front->next;

    // Free the memory of the dequeued node
    free(temp);

    return data;
}

// Function to display the elements in the priority queue
void displayPriorityQueue(struct PriorityQueue* pq) {
    if (isEmpty(pq)) {
        printf("Priority queue is empty\n");
        return;
    }

    struct Node* current = pq->front;
    while (current != NULL) {
        printf("(%d, %d) ", current->data, current->priority);
        current = current->next;
    }
    printf("\n");
}

int main() {
    struct PriorityQueue myPriorityQueue;
    initializePriorityQueue(&myPriorityQueue);

    enqueue(&myPriorityQueue, 10, 2);
    enqueue(&myPriorityQueue, 20, 1);
```

```
    enqueue(&myPriorityQueue, 30, 3);

    printf("Priority Queue after enqueue: ");
    displayPriorityQueue(&myPriorityQueue);

    int dequeuedValue = dequeue(&myPriorityQueue);
    printf("Dequeued element: %d\n", dequeuedValue);

    printf("Priority Queue after dequeue: ");
    displayPriorityQueue(&myPriorityQueue);

    return 0;
}
```

6) Write a Program to implement circular queue.

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

#define MAX_SIZE 5

// Circular Queue structure
struct CircularQueue {
    int* array;
    int front;
    int rear;
    int size;
};

// Function to initialize an empty circular queue
void initializeCircularQueue(struct CircularQueue* cq, int size) {
```

```c
    cq->array = (int*)malloc(size * sizeof(int));
    if (cq->array == NULL) {
        printf("Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    cq->front = cq->rear = -1;
    cq->size = size;
}

// Function to check if the circular queue is empty
int isEmpty(struct CircularQueue* cq) {
    return cq->front == -1;
}

// Function to check if the circular queue is full
int isFull(struct CircularQueue* cq) {
    return (cq->rear + 1) % cq->size == cq->front;
}

// Function to enqueue (insert) a new element into the circular queue
void enqueue(struct CircularQueue* cq, int value) {
    if (isFull(cq)) {
        printf("Circular queue overflow\n");
        exit(EXIT_FAILURE);
    }

    // If the circular queue is empty, set both front and rear to 0
    if (isEmpty(cq)) {
        cq->front = cq->rear = 0;
```

```
    } else {
        // Otherwise, move the rear pointer circularly
        cq->rear = (cq->rear + 1) % cq->size;
    }

    // Insert the new element at the rear
    cq->array[cq->rear] = value;
}

// Function to dequeue (remove) an element from the circular queue
int dequeue(struct CircularQueue* cq) {
    if (isEmpty(cq)) {
        printf("Circular queue underflow\n");
        exit(EXIT_FAILURE);
    }

    // Get the data from the front
    int data = cq->array[cq->front];

    // If there is only one element in the circular queue, set both front and rear to -1
    if (cq->front == cq->rear) {
        cq->front = cq->rear = -1;
    } else {
        // Otherwise, move the front pointer circularly
        cq->front = (cq->front + 1) % cq->size;
    }

    return data;
}
```

```
// Function to display the elements in the circular queue
void displayCircularQueue(struct CircularQueue* cq) {
    if (isEmpty(cq)) {
        printf("Circular queue is empty\n");
        return;
    }

    int i = cq->front;
    do {
        printf("%d ", cq->array[i]);
        i = (i + 1) % cq->size;
    } while (i != (cq->rear + 1) % cq->size);
    printf("\n");
}

int main() {
    struct CircularQueue myCircularQueue;
    initializeCircularQueue(&myCircularQueue, MAX_SIZE);

    enqueue(&myCircularQueue, 10);
    enqueue(&myCircularQueue, 20);
    enqueue(&myCircularQueue, 30);

    printf("Circular Queue after enqueue: ");
    displayCircularQueue(&myCircularQueue);

    int dequeuedValue = dequeue(&myCircularQueue);
    printf("Dequeued element: %d\n", dequeuedValue);
```

```
    printf("Circular Queue after dequeue: ");
    displayCircularQueue(&myCircularQueue);

    return 0;
}
```

7) Write a Program to implement Josephus Problem using list implementation of queue.

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

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

// Queue structure
struct Queue {
    struct Node* front;
    struct Node* rear;
};

// Function to initialize an empty queue
void initializeQueue(struct Queue* q) {
    q->front = q->rear = NULL;
}

// Function to check if the queue is empty
int isEmpty(struct Queue* q) {
```

```
    return q->front == NULL;
}

// Function to enqueue (insert) a new element into the queue
void enqueue(struct Queue* q, int value) {
    // Create a new node
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = value;
    newNode->next = NULL;

    // If the queue is empty, set both front and rear to the new node
    if (isEmpty(q)) {
        q->front = q->rear = newNode;
    } else {
        // Otherwise, add the new node to the rear and update the rear pointer
        q->rear->next = newNode;
        q->rear = newNode;
    }
}

// Function to dequeue (remove) an element from the queue
int dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue underflow\n");
        exit(EXIT_FAILURE);
```

```
    }

    // Get the data from the front node
    int data = q->front->data;

    // Move the front pointer to the next node
    struct Node* temp = q->front;
    q->front = q->front->next;

    // If the queue becomes empty after dequeue, update the rear pointer to NULL
    if (q->front == NULL) {
        q->rear = NULL;
    }

    // Free the memory of the dequeued node
    free(temp);

    return data;
}

// Function to solve the Josephus problem
int josephus(struct Queue* q, int n, int k) {
    // Enqueue people in the queue
    for (int i = 1; i <= n; ++i) {
        enqueue(q, i);
    }

    // Perform the Josephus elimination
    while (!isEmpty(q)) {
```

```
        for (int i = 1; i < k; ++i) {
            // Move the front element to the rear for (k-1) times
            int frontValue = dequeue(q);
            enqueue(q, frontValue);
        }

        // Eliminate the k-th person
        printf("%d ", dequeue(q));
    }

    return 0;
}

int main() {
    struct Queue myQueue;
    initializeQueue(&myQueue);

    int n, k;
    printf("Enter the total number of people (n): ");
    scanf("%d", &n);
    printf("Enter the counting interval (k): ");
    scanf("%d", &k);

    printf("Josephus Sequence: ");
    josephus(&myQueue, n, k);

    return 0;
}
```

8) Write a program to implement binary tree traversal.

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

// Node structure
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));
    if (newNode == NULL) {
        printf("Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = value;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// Function to perform in-order traversal of the binary tree
void inOrderTraversal(struct Node* root) {
    if (root != NULL) {
        inOrderTraversal(root->left);
        printf("%d ", root->data);
        inOrderTraversal(root->right);
    }
```

```c
}

// Function to perform pre-order traversal of the binary tree
void preOrderTraversal(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preOrderTraversal(root->left);
        preOrderTraversal(root->right);
    }
}

// Function to perform post-order traversal of the binary tree
void postOrderTraversal(struct Node* root) {
    if (root != NULL) {
        postOrderTraversal(root->left);
        postOrderTraversal(root->right);
        printf("%d ", root->data);
    }
}

int main() {
    // Creating a sample binary tree
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);
    root->right->left = createNode(6);
    root->right->right = createNode(7);
```

```
    printf("In-order Traversal: ");
    inOrderTraversal(root);
    printf("\n");

    printf("Pre-order Traversal: ");
    preOrderTraversal(root);
    printf("\n");

    printf("Post-order Traversal: ");
    postOrderTraversal(root);
    printf("\n");

    return 0;
}
```

9) Write a Program to implement BST Insertion and Deletion.

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

// Node structure
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));
```

```c
    if (newNode == NULL) {
        printf("Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = value;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// Function to insert a new node into 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 to find the minimum value node in a BST
struct Node* findMin(struct Node* root) {
    while (root->left != NULL) {
        root = root->left;
    }
```

```
    return root;
}

// Function to delete a node with a given value from the BST
struct Node* deleteNode(struct Node* root, int value) {
    if (root == NULL) {
        return root;
    }

    // Find the node to be deleted
    if (value < root->data) {
        root->left = deleteNode(root->left, value);
    } else if (value > root->data) {
        root->right = deleteNode(root->right, value);
    } else {
        // Node with only one child or no child
        if (root->left == NULL) {
            struct Node* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }

        // Node with two children, get the inorder successor (smallest in the right subtree)
        struct Node* temp = findMin(root->right);
```

```
        // Copy the inorder successor's data to this node
        root->data = temp->data;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->data);
    }

    return root;
}

// Function to perform in-order traversal of the BST
void inOrderTraversal(struct Node* root) {
    if (root != NULL) {
        inOrderTraversal(root->left);
        printf("%d ", root->data);
        inOrderTraversal(root->right);
    }
}

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

    // Inserting nodes into the BST
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
```

```
    insert(root, 80);

    printf("In-order Traversal before deletion: ");
    inOrderTraversal(root);
    printf("\n");

    // Deleting a node from the BST
    root = deleteNode(root, 20);

    printf("In-order Traversal after deletion: ");
    inOrderTraversal(root);
    printf("\n");

    return 0;
}
```

10) Write a Program to implement AVL tree

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

// Node structure
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
    int height;
};

// Function to get the height of a node
int height(struct Node* node) {
```

```
    if (node == NULL)
        return 0;
    return node->height;
}

// Function to calculate the balance factor of a node
int balanceFactor(struct Node* node) {
    if (node == NULL)
        return 0;
    return height(node->left) - height(node->right);
}

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = value;
    newNode->left = newNode->right = NULL;
    newNode->height = 1; // New node is initially at height 1
    return newNode;
}

// Function to perform right rotation on a given node
struct Node* rightRotate(struct Node* y) {
    struct Node* x = y->left;
    struct Node* T2 = x->right;
```

```
    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right));
    x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right));

    return x; // New root of the subtree
}

// Function to perform left rotation on a given node
struct Node* leftRotate(struct Node* x) {
    struct Node* y = x->right;
    struct Node* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right));
    y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right));

    return y; // New root of the subtree
}

// Function to insert a new node with the given value into the AVL tree
```

```
struct Node* insert(struct Node* root, int value) {
  // Standard BST insertion
  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);
  else // Duplicates are not allowed
    return root;

  // Update height of the current node
  root->height = 1 + (height(root->left) > height(root->right) ? height(root->left) : height(root->right));

  // Get the balance factor and perform rotations if needed
  int balance = balanceFactor(root);

  // Left Left Case
  if (balance > 1 && value < root->left->data)
    return rightRotate(root);

  // Right Right Case
  if (balance < -1 && value > root->right->data)
    return leftRotate(root);

  // Left Right Case
  if (balance > 1 && value > root->left->data) {
    root->left = leftRotate(root->left);
```

```
        return rightRotate(root);
    }

    // Right Left Case
    if (balance < -1 && value < root->right->data) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

// Function to perform in-order traversal of the AVL tree
void inOrderTraversal(struct Node* root) {
    if (root != NULL) {
        inOrderTraversal(root->left);
        printf("%d ", root->data);
        inOrderTraversal(root->right);
    }
}

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

    // Inserting nodes into the AVL tree
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
```

```
    root = insert(root, 50);
    root = insert(root, 25);

    printf("In-order Traversal after AVL Insertion: ");
    inOrderTraversal(root);
    printf("\n");

    return 0;
}
```

11) Write a Program to implement open addressing techniques in Hashing

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

#define TABLE_SIZE 10

// Hash table structure
struct HashTable {
    int* array;
    int size;
};

// Function to initialize the hash table
void initializeHashTable(struct HashTable* ht, int size) {
    ht->array = (int*)malloc(size * sizeof(int));
    if (ht->array == NULL) {
        printf("Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    ht->size = size;
```

```
    // Initialize all slots as empty (-1)
    for (int i = 0; i < size; ++i) {
        ht->array[i] = -1;
    }
}

// Function to calculate the hash value using modulo division
int hash(int key, int size) {
    return key % size;
}

// Function to insert a key into the hash table using linear probing
void insert(struct HashTable* ht, int key) {
    int index = hash(key, ht->size);

    // Linear probing
    while (ht->array[index] != -1) {
        index = (index + 1) % ht->size;
    }

    // Insert the key into the hash table
    ht->array[index] = key;
}

// Function to search for a key in the hash table
int search(struct HashTable* ht, int key) {
    int index = hash(key, ht->size);
```

```
    // Linear probing
    while (ht->array[index] != key) {
        if (ht->array[index] == -1) {
            return -1; // Key not found
        }
        index = (index + 1) % ht->size;
    }

    return index; // Return the index where the key is found
}

// Function to display the contents of the hash table
void displayHashTable(struct HashTable* ht) {
    printf("Hash Table: ");
    for (int i = 0; i < ht->size; ++i) {
        printf("%d ", ht->array[i]);
    }
    printf("\n");
}

int main() {
    struct HashTable myHashTable;
    initializeHashTable(&myHashTable, TABLE_SIZE);

    insert(&myHashTable, 5);
    insert(&myHashTable, 25);
    insert(&myHashTable, 15);
    insert(&myHashTable, 35);
```

```
    displayHashTable(&myHashTable);

    int searchKey = 15;
    int searchResult = search(&myHashTable, searchKey);
    if (searchResult != -1) {
        printf("Key %d found at index %d\n", searchKey, searchResult);
    } else {
        printf("Key %d not found\n", searchKey);
    }

    return 0;
}
```

12) Write a Program to implement DFS in Graphs

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

#define MAX_VERTICES 100

// Node structure for adjacency list
struct Node {
    int vertex;
    struct Node* next;
};

// Graph structure
struct Graph {
    struct Node* adjacencyList[MAX_VERTICES];
};
```

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

// Function to add an edge to the graph
void addEdge(struct Graph* graph, int src, int dest) {
   // Add an edge from src to dest
   struct Node* newNode = createNode(dest);
   newNode->next = graph->adjacencyList[src];
   graph->adjacencyList[src] = newNode;

   // Uncomment the following lines if the graph is undirected
   /*
   // Add an edge from dest to src
   newNode = createNode(src);
   newNode->next = graph->adjacencyList[dest];
   graph->adjacencyList[dest] = newNode;
   */
}

// Function to perform Depth-First Search (DFS)
```

```
void DFS(struct Graph* graph, int vertex, int visited[]) {
    visited[vertex] = 1;
    printf("%d ", vertex);

    struct Node* current = graph->adjacencyList[vertex];
    while (current != NULL) {
        if (!visited[current->vertex]) {
            DFS(graph, current->vertex, visited);
        }
        current = current->next;
    }
}

int main() {
    struct Graph myGraph;
    int visited[MAX_VERTICES] = {0};

    // Initialize the graph
    for (int i = 0; i < MAX_VERTICES; ++i) {
        myGraph.adjacencyList[i] = NULL;
    }

    // Add edges to the graph
    addEdge(&myGraph, 0, 1);
    addEdge(&myGraph, 0, 2);
    addEdge(&myGraph, 1, 3);
    addEdge(&myGraph, 1, 4);
    addEdge(&myGraph, 2, 5);
```

```
    printf("Depth-First Search (DFS): ");
    DFS(&myGraph, 0, visited);
    printf("\n");

    return 0;
}
```

13) Write a Program to implement BFS in Graphs

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

#define MAX_VERTICES 100
#define QUEUE_SIZE 100

// Node structure for adjacency list
struct Node {
    int vertex;
    struct Node* next;
};

// Graph structure
struct Graph {
    struct Node* adjacencyList[MAX_VERTICES];
};

// Queue structure for BFS
struct Queue {
    int items[QUEUE_SIZE];
    int front;
    int rear;
```

```c
};

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

// Function to add an edge to the graph
void addEdge(struct Graph* graph, int src, int dest) {
    // Add an edge from src to dest
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjacencyList[src];
    graph->adjacencyList[src] = newNode;

    // Uncomment the following lines if the graph is undirected
    /*
    // Add an edge from dest to src
    newNode = createNode(src);
    newNode->next = graph->adjacencyList[dest];
    graph->adjacencyList[dest] = newNode;
    */
}
```

```c
// Function to initialize the queue
void initializeQueue(struct Queue* q) {
    q->front = q->rear = -1;
}

// Function to check if the queue is empty
int isEmpty(struct Queue* q) {
    return q->front == -1;
}

// Function to enqueue an item into the queue
void enqueue(struct Queue* q, int value) {
    if (q->rear == QUEUE_SIZE - 1) {
        printf("Queue overflow\n");
        exit(EXIT_FAILURE);
    } else {
        if (q->front == -1) {
            q->front = 0;
        }
        q->items[++q->rear] = value;
    }
}

// Function to dequeue an item from the queue
int dequeue(struct Queue* q) {
    int item;
    if (isEmpty(q)) {
        printf("Queue underflow\n");
```

```
        exit(EXIT_FAILURE);
    } else {
        item = q->items[q->front++];
        if (q->front > q->rear) {
```

Singly Linked List Operations:

INSERTION:

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

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

// Function to insert a new node at the end of the list
void insertNode(struct Node** head, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = value;
    newNode->next = NULL;

    if (*head == NULL) {
        *head = newNode;
    } else {
```

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

// Function to print the linked list
void printList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

// Other operations can be added here

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

    // Insertion
    insertNode(&myList, 1);
    insertNode(&myList, 2);
    insertNode(&myList, 3);

    // Display the list
```

```
    printf("Original List: ");

    printList(myList);


    return 0;

}
```

REVERSING:

```
// Function to reverse the linked list

struct Node* reverseList(struct Node* head) {

    struct Node *prev, *current, *next;

    prev = NULL;

    current = head;


    while (current != NULL) {

        next = current->next;

        current->next = prev;

        prev = current;

        current = next;

    }


    return prev; // New head of the reversed list

}


// In the main function, you can call reverseList(myList) to reverse the list.
```

REMOVE OCCURANCE:

```
// Function to remove occurrences of a particular data in the list

void removeOccurrences(struct Node** head, int key) {

    struct Node *current = *head, *prev = NULL;


    while (current != NULL) {
```

```
        if (current->data == key) {
            if (prev == NULL) {
                *head = current->next;
                free(current);
                current = *head;
            } else {
                prev->next = current->next;
                free(current);
                current = prev->next;
            }
        } else {
            prev = current;
            current = current->next;
        }
    }
}


// In the main function, you can call removeOccurrences(&myList, key) to remove occurrences of 'key'.
```

DELETION:

```
void deleteNode(struct Node** head, int key) {
    struct Node *current = *head, *prev = NULL;


    // Check if the key is in the head node
    if (current != NULL && current->data == key) {
        *head = current->next;
        free(current);
        return;
    }
```

```
    // Search for the key to be deleted, keeping track of the previous node
    while (current != NULL && current->data != key) {
        prev = current;
        current = current->next;
    }

    // If the key is not present
    if (current == NULL) {
        printf("Key not found in the list\n");
        return;
    }

    // Unlink the node from the linked list
    prev->next = current->next;

    // Free the memory of the node to be deleted
    free(current);
}
```

14) Write a program to input a n-digit number. Now break the number in to individual digits, then store every single digit in a separate node of a linked list.

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

// Node structure for a linked list
struct Node {
    int data;
    struct Node* next;
};
```

```
// Function to insert a new node at the end of the list
void insertNode(struct Node** head, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = value;
    newNode->next = NULL;

    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

// Function to break a number into individual digits and store them in a linked list
void breakAndStoreDigits(struct Node** head, int number) {
    // Break the number into digits and store them in reverse order
    while (number > 0) {
        int digit = number % 10;
        insertNode(head, digit);
        number /= 10;
```

```
    }
}

// Function to print the linked list
void printList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

int main() {
    int n;
    printf("Enter an n-digit number: ");
    scanf("%d", &n);

    struct Node* myList = NULL;

    // Break the number into digits and store them in a linked list
    breakAndStoreDigits(&myList, n);

    // Display the list
    printf("Digits in the linked list: ");
    printList(myList);

    return 0;
}
```

15) Write a program to manipulate polynomial operations

```
#include <stdio.h>

#include <stdlib.h>


// Node structure for a term in a polynomial

struct Term {

    int coefficient;

    int exponent;

    struct Term* next;

};


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

void insertTerm(struct Term** poly, int coefficient, int exponent) {

    struct Term* newTerm = (struct Term*)malloc(sizeof(struct Term));

    if (newTerm == NULL) {

        printf("Memory allocation error\n");

        exit(EXIT_FAILURE);

    }

    newTerm->coefficient = coefficient;

    newTerm->exponent = exponent;

    newTerm->next = NULL;


    if (*poly == NULL) {

        *poly = newTerm;

    } else {

        struct Term* temp = *poly;

        while (temp->next != NULL) {

            temp = temp->next;

        }
```

```
        temp->next = newTerm;
    }
}

// Function to display a polynomial
void displayPolynomial(struct Term* poly) {
    struct Term* current = poly;
    while (current != NULL) {
        printf("%dx^%d", current->coefficient, current->exponent);
        current = current->next;
        if (current != NULL) {
            printf(" + ");
        }
    }
    printf("\n");
}

// Function to add two polynomials
struct Term* addPolynomials(struct Term* poly1, struct Term* poly2) {
    struct Term* result = NULL;

    while (poly1 != NULL && poly2 != NULL) {
        if (poly1->exponent > poly2->exponent) {
            insertTerm(&result, poly1->coefficient, poly1->exponent);
            poly1 = poly1->next;
        } else if (poly1->exponent < poly2->exponent) {
            insertTerm(&result, poly2->coefficient, poly2->exponent);
            poly2 = poly2->next;
        } else {
```

```
            // Exponents are equal, add coefficients
            insertTerm(&result, poly1->coefficient + poly2->coefficient, poly1->exponent);
            poly1 = poly1->next;
            poly2 = poly2->next;
        }
    }

    // If one polynomial is longer than the other
    while (poly1 != NULL) {
        insertTerm(&result, poly1->coefficient, poly1->exponent);
        poly1 = poly1->next;
    }

    while (poly2 != NULL) {
        insertTerm(&result, poly2->coefficient, poly2->exponent);
        poly2 = poly2->next;
    }

    return result;
}

// Function to multiply two polynomials
struct Term* multiplyPolynomials(struct Term* poly1, struct Term* poly2) {
    struct Term* result = NULL;
    struct Term* tempResult = NULL;

    while (poly1 != NULL) {
        struct Term* temp = poly2;
        while (temp != NULL) {
```

```
            int coefficient = poly1->coefficient * temp->coefficient;
            int exponent = poly1->exponent + temp->exponent;
            insertTerm(&tempResult, coefficient, exponent);
            temp = temp->next;
        }

        // Add the terms of the current row to the result
        result = addPolynomials(result, tempResult);

        // Clear tempResult for the next row
        tempResult = NULL;

        poly1 = poly1->next;
    }

    return result;
}

// Function to free the memory allocated for a polynomial
void freePolynomial(struct Term* poly) {
    struct Term* current = poly;
    while (current != NULL) {
        struct Term* temp = current;
        current = current->next;
        free(temp);
    }
}

int main() {
```

```
struct Term* poly1 = NULL;
struct Term* poly2 = NULL;

// Insert terms into the first polynomial
insertTerm(&poly1, 3, 2);
insertTerm(&poly1, -2, 1);
insertTerm(&poly1, 5, 0);

// Insert terms into the second polynomial
insertTerm(&poly2, 4, 3);
insertTerm(&poly2, 1, 1);
insertTerm(&poly2, -7, 0);

// Display the polynomials
printf("Polynomial 1: ");
displayPolynomial(poly1);

printf("Polynomial 2: ");
displayPolynomial(poly2);

// Add the polynomials
struct Term* sum = addPolynomials(poly1, poly2);
printf("Sum of Polynomials: ");
displayPolynomial(sum);

// Multiply the polynomials
struct Term* product = multiplyPolynomials(poly1, poly2);
printf("Product of Polynomials: ");
displayPolynomial(product);
```

```
    // Free the allocated memory
    freePolynomial(poly1);
    freePolynomial(poly2);
    freePolynomial(sum);
    freePolynomial(product);

    return 0;
}
```

16) Write a Program to implement the following operations in Singly Linked list Insertion (ii) Remove the middle element from the list

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

// Node structure for a linked list
struct Node {
    int data;
    struct Node* next;
};

// Function to insert a new node at the end of the list
void insertNode(struct Node** head, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = value;
    newNode->next = NULL;
```

```
    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

// Function to remove the middle element from the list
void removeMiddleElement(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty, cannot remove middle element\n");
        return;
    }

    struct Node *slowPtr = *head, *fastPtr = *head, *prev = NULL;

    while (fastPtr != NULL && fastPtr->next != NULL) {
        fastPtr = fastPtr->next->next;
        prev = slowPtr;
        slowPtr = slowPtr->next;
    }

    if (prev != NULL) {
        // If the length of the list is odd, skip the middle element
```

```
        prev->next = slowPtr->next;
    } else {
        // If the length of the list is even, update the head
        *head = slowPtr->next;
    }

    free(slowPtr);
}

// Function to print the linked list
void printList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

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

    // Insert elements into the list
    insertNode(&myList, 1);
    insertNode(&myList, 2);
    insertNode(&myList, 3);
    insertNode(&myList, 4);
    insertNode(&myList, 5);
```

```
    // Display the original list
    printf("Original List: ");
    printList(myList);

    // Remove the middle element
    removeMiddleElement(&myList);

    // Display the list after removing the middle element
    printf("List after removing the middle element: ");
    printList(myList);

    return 0;
}
```

DOUBLE LINKED LIST OPERATIONS:

INSERTION:

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

// Node structure for a doubly linked list
struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};

// Function to insert a new node at the end of the doubly linked list
void insertNode(struct Node** head, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
```

```
    if (newNode == NULL) {
        printf("Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = value;
    newNode->next = NULL;

    if (*head == NULL) {
        newNode->prev = NULL;
        *head = newNode;
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        newNode->prev = temp;
        temp->next = newNode;
    }
}

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

```
// Other operations can be added here

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

    // Insertion
    insertNode(&myList, 1);
    insertNode(&myList, 2);
    insertNode(&myList, 3);

    // Display the list
    printf("Original List: ");
    displayList(myList);

    return 0;
}
```

REVERSING:

```
// Function to reverse a doubly linked list
void reverseList(struct Node** head) {
    struct Node *current = *head, *temp = NULL;

    while (current != NULL) {
        temp = current->prev;
        current->prev = current->next;
        current->next = temp;
        current = current->prev;
    }
```

```
    if (temp != NULL) {
        *head = temp->prev;
    }
}
```

REMOVE OCCURANCE:

```
// Function to remove occurrences of a particular data in a doubly linked list
void removeOccurrences(struct Node** head, int key) {
    struct Node* current = *head, *temp = NULL;

    while (current != NULL) {
        if (current->data == key) {
            if (current->prev != NULL) {
                current->prev->next = current->next;
            } else {
                *head = current->next;
            }

            if (current->next != NULL) {
                current->next->prev = current->prev;
            }

            temp = current;
            current = current->next;
            free(temp);
        } else {
            current = current->next;
        }
    }
}
```

MERGING:

```c
// Function to merge two doubly linked lists
struct Node* mergeLists(struct Node* list1, struct Node* list2) {
    if (list1 == NULL) {
        return list2;
    }
    if (list2 == NULL) {
        return list1;
    }

    struct Node* temp = list1;
    while (temp->next != NULL) {
        temp = temp->next;
    }

    temp->next = list2;
    list2->prev = temp;

    return list1;
}
```

CHECKING PALINDROME:

```c
#include <stdbool.h>

// Function to check if a doubly linked list is a palindrome
bool isPalindrome(struct Node* head) {
    struct Node *front = head, *rear = head;

    while (rear != NULL && rear->next != NULL) {
        front = front->next;
```

```
        rear = rear->next->next;
    }

    // Now 'front' points to the middle of the list

    // Reverse the second half of the list
    reverseList(&front);

    // Compare the first and second halves
    while (front != NULL) {
        if (front->data != head->data) {
            // Not a palindrome
            return false;
        }
        front = front->next;
        head = head->next;
    }

    // Palindrome
    return true;
}
```

DELETION(PRINT MAX AND MIN):

```
// Function to print the minimum and maximum element in a doubly linked list
void printMinMax(struct Node* head) {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
```

```
    int min = head->data, max = head->data;
    struct Node* current = head;

    while (current != NULL) {
        if (current->data < min) {
            min = current->data;
        }
        if (current->data > max) {
            max = current->data;
        }
        current = current->next;
    }

    printf("Minimum element: %d\n", min);
    printf("Maximum element: %d\n", max);
}
```

MIDDLE ELEMENT REMOVAL:

```
// Function to remove the middle element from a doubly linked list
void removeMiddleElement(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty, cannot remove middle element\n");
        return;
    }

    struct Node *slowPtr = *head, *fastPtr = *head, *prev = NULL;

    while (fastPtr != NULL && fastPtr->next != NULL) {
        fastPtr = fastPtr->next->next;
        prev = slowPtr;
```

```
        slowPtr = slowPtr->next;
    }

    if (prev != NULL) {
        // If the length of the list is odd, skip the middle element
        prev->next = slowPtr->next;
        if (slowPtr->next != NULL) {
            slowPtr->next->prev = prev;
        }
    } else {
        // If the length of the list is even, update the head
        *head = slowPtr->next;
        if (slowPtr->next != NULL) {
            slowPtr->next->prev = NULL;
        }
    }

    free(slowPtr);
}
```

17) Write a program to input a n-digit number. Now break the number in to individual digits, then store every single digit in a separate node of a Doubly linked list

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

// Node structure for a doubly linked list
struct Node {
    int data;
    struct Node* prev;
```

```
    struct Node* next;
};

// Function to insert a new node at the end of the doubly linked list
void insertNode(struct Node** head, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = value;
    newNode->next = NULL;

    if (*head == NULL) {
        newNode->prev = NULL;
        *head = newNode;
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        newNode->prev = temp;
        temp->next = newNode;
    }
}

// Function to break a number into digits and store them in a doubly linked list
void breakAndStoreDigits(struct Node** head, int number) {
    // Break the number into digits and store them in reverse order
```

```
    while (number > 0) {
        int digit = number % 10;
        insertNode(head, digit);
        number /= 10;
    }
}

// Function to print the doubly linked list
void printList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

int main() {
    int n;
    printf("Enter an n-digit number: ");
    scanf("%d", &n);

    struct Node* myList = NULL;

    // Break the number into digits and store them in a doubly linked list
    breakAndStoreDigits(&myList, n);

    // Display the list
    printf("Digits in the doubly linked list: ");
```

```
    printList(myList);


    return 0;
}
```

18) Write a program to input a n-digit number. Now break the number in to individual digits, then store every single digit in a separate node of a Circular linked list.

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

// Node structure for a circular linked list
struct Node {
    int data;
    struct Node* next;
};

// Function to insert a new node at the end of the circular linked list
void insertNode(struct Node** head, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = value;

    if (*head == NULL) {
        newNode->next = newNode; // Point to itself to create a circular list
        *head = newNode;
    } else {
```

```
        struct Node* temp = *head;
        while (temp->next != *head) {
            temp = temp->next;
        }
        newNode->next = *head;
        temp->next = newNode;
    }
}

// Function to break a number into digits and store them in a circular linked list
void breakAndStoreDigits(struct Node** head, int number) {
    // Break the number into digits and store them
    do {
        int digit = number % 10;
        insertNode(head, digit);
        number /= 10;
    } while (number > 0);
}

// Function to print the circular linked list
void printList(struct Node* head) {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }

    struct Node* current = head;
    do {
        printf("%d ", current->data);
```

```
        current = current->next;
    } while (current != head);
    printf("\n");
}

int main() {
    int n;
    printf("Enter an n-digit number: ");
    scanf("%d", &n);

    struct Node* myCircularList = NULL;

    // Break the number into digits and store them in a circular linked list
    breakAndStoreDigits(&myCircularList, n);

    // Display the list
    printf("Digits in the circular linked list: ");
    printList(myCircularList);

    return 0;
}
```

19) Write a program to implement stack using linked list

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

// Node structure for a stack node
struct Node {
    int data;
    struct Node* next;
```

```
};

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

// Function to push a value onto the stack
void push(struct Node** top, int value) {
    struct Node* newNode = createNode(value);
    newNode->next = *top;
    *top = newNode;
}

// Function to pop a value from the stack
int pop(struct Node** top) {
    if (*top == NULL) {
        printf("Stack is empty\n");
        exit(EXIT_FAILURE);
    }

    struct Node* temp = *top;
```

```
    int poppedValue = temp->data;
    *top = temp->next;
    free(temp);
    return poppedValue;
}

// Function to check if the stack is empty
int isEmpty(struct Node* top) {
    return top == NULL;
}

// Function to display the elements of the stack
void displayStack(struct Node* top) {
    if (top == NULL) {
        printf("Stack is empty\n");
        return;
    }

    printf("Stack elements: ");
    while (top != NULL) {
        printf("%d ", top->data);
        top = top->next;
    }
    printf("\n");
}

int main() {
    struct Node* stackTop = NULL;
```

```c
    // Push elements onto the stack
    push(&stackTop, 10);
    push(&stackTop, 20);
    push(&stackTop, 30);

    // Display the stack
    displayStack(stackTop);

    // Pop an element from the stack
    int poppedValue = pop(&stackTop);
    printf("Popped value: %d\n", poppedValue);

    // Display the stack after popping
    displayStack(stackTop);

    return 0;
}
```