Experiment 1. Implement a list using array and develop functions to perform insertion, deletion and linear search operations

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

#define MAX_SIZE 10

// Function to display the array elements
void displayList(int arr[], int size) {
    printf("List: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Function to perform linear search for an element in the array
int linearSearch(int arr[], int size, int key) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {
            printf("Element %d found at position %d.\n", key, i);
            return i;
        }
    }
    printf("Element %d not found in the list.\n", key);
    return -1;
```

```c
}

// Function to insert an element at a specific position in the array
void insertElement(int arr[], int *size, int position, int element) {
    if (*size >= MAX_SIZE) {
        printf("List is full. Cannot perform insertion.\n");
        return;
    }

    if (position < 0 || position > *size) {
        printf("Invalid position for insertion.\n");
        return;
    }

    for (int i = *size; i > position; i--) {
        arr[i] = arr[i - 1];
    }

    arr[position] = element;
    (*size)++;
    printf("Element %d inserted at position %d.\n", element, position);
}

// Function to delete an element at a specific position in the array
void deleteElement(int arr[], int *size, int position) {
    if (*size == 0) {
        printf("List is empty. Cannot perform deletion.\n");
        return;
    }

    if (position < 0 || position >= *size) {
        printf("Invalid position for deletion.\n");
        return;
    }

    int deletedElement = arr[position];

    for (int i = position; i < *size - 1; i++) {
        arr[i] = arr[i + 1];
    }

    (*size)--;
    printf("Element %d deleted from position %d.\n", deletedElement,
position);
}

int main() {
    int arr[MAX_SIZE];
```

```
    int size = 0;

    // Insert elements
    insertElement(arr, &size, 0, 10);
    insertElement(arr, &size, 1, 20);
    insertElement(arr, &size, 2, 30);
    displayList(arr, size);

    // Delete element at position 1
    deleteElement(arr, &size, 1);
    displayList(arr, size);

    // Insert element at position 1
    insertElement(arr, &size, 1, 40);
    displayList(arr, size);

    // Linear search for element 30
    int position = linearSearch(arr, size, 30);

    return 0;
}
```

Output:

```
Element 10 inserted at position 0.
Element 20 inserted at position 1.
Element 30 inserted at position 2.
List: 10 20 30
Element 20 deleted from position 1.
List: 10 30
Element 40 inserted at position 1.
List: 10 40 30
Element 30 found at position 2.
```

Experiment 2. Implement a stack using array and develop functions to perform push and pop operations

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

#define MAX_SIZE 5

struct Stack {
    int arr[MAX_SIZE];
```

```
    int top;
};

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

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

// Function to check if the stack is full
int isFull(struct Stack *stack) {
    return stack->top == MAX_SIZE - 1;
}

// Function to push an element onto the stack
void push(struct Stack *stack, int item) {
    if (isFull(stack)) {
        printf("Stack is full. Cannot perform push operation.\n");
        return;
    }
    stack->arr[++stack->top] = item;
    printf("Pushed: %d\n", item);
}

// Function to pop an element from the stack
int pop(struct Stack *stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty. Cannot perform pop operation.\n");
        return -1; // Assuming -1 is not a valid element in the stack
    }
    int item = stack->arr[stack->top--];
    printf("Popped: %d\n", item);
    return item;
}

// Function to display the elements of the stack
void displayStack(struct Stack *stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty.\n");
        return;
    }
    printf("Stack: ");
    for (int i = 0; i <= stack->top; i++) {
        printf("%d ", stack->arr[i]);
```

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

int main() {
    struct Stack stack;
    initializeStack(&stack);

    // Push elements onto the stack
    push(&stack, 10);
    push(&stack, 20);
    push(&stack, 30);
    displayStack(&stack);

    // Pop element from the stack
    int poppedItem = pop(&stack);
    if (poppedItem != -1) {
        printf("Popped item: %d\n", poppedItem);
    }
    displayStack(&stack);

    // Push more elements
    push(&stack, 40);
    push(&stack, 50);
    displayStack(&stack);

    // Attempt to push onto a full stack
    push(&stack, 60);

    return 0;
}
```

Output:

```
Pushed: 10
Pushed: 20
Pushed: 30
Stack: 10 20 30
Popped: 30
Popped item: 30
Stack: 10 20
Pushed: 40
Pushed: 50
Stack: 10 20 40 50
Stack is full. Cannot perform push operation.
```

Experiment 3. Write a program to check if a given expression is correctly parenthesized using stacks

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

#define MAX_SIZE 50

struct Stack {
    char arr[MAX_SIZE];
    int top;
};

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

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

// Function to check if the stack is full
bool isFull(struct Stack *stack) {
    return stack->top == MAX_SIZE - 1;
}

// Function to push a character onto the stack
void push(struct Stack *stack, char item) {
    if (isFull(stack)) {
        printf("Stack is full. Cannot perform push operation.\n");
        exit(EXIT_FAILURE);
    }
    stack->arr[++stack->top] = item;
}

// Function to pop a character from the stack
char pop(struct Stack *stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty. Cannot perform pop operation.\n");
        exit(EXIT_FAILURE);
    }
    return stack->arr[stack->top--];
```

```c
// Function to check if the expression is correctly parenthesized
bool isParenthesizedCorrectly(char expression[]) {
    struct Stack stack;
    initializeStack(&stack);

    for (int i = 0; expression[i] != '\0'; i++) {
        char currentChar = expression[i];
        if (currentChar == '(' || currentChar == '[' || currentChar == '{') {
            push(&stack, currentChar);
        } else if (currentChar == ')' || currentChar == ']' || currentChar == '}') {
            if (isEmpty(&stack)) {
                return false; // Closing parenthesis without a matching opening parenthesis
            }

            char poppedChar = pop(&stack);

            // Check if the popped character matches the corresponding opening parenthesis
            if ((currentChar == ')' && poppedChar != '(') ||
                (currentChar == ']' && poppedChar != '[') ||
                (currentChar == '}' && poppedChar != '{')) {
                return false;
            }
        }
    }

    // Check if there are any remaining opening parentheses
    return isEmpty(&stack);
}

int main() {
    char expression[MAX_SIZE];

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

    if (isParenthesizedCorrectly(expression)) {
        printf("The expression is correctly parenthesized.\n");
    } else {
        printf("The expression is not correctly parenthesized.\n");
    }

    return 0;
```

```
}
```

Output:

```
Enter an expression: (a + b) * (c - d)
The expression is correctly parenthesized.

Enter an expression: (a + b) * (c - d]
The expression is not correctly parenthesized.
```

Experiment 4. Write a program to evaluate postfix, prefix and infix expressions using stacks

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

#define MAX_SIZE 50

struct Stack {
    int arr[MAX_SIZE];
    int top;
};

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

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

// Function to check if the stack is full
int isFull(struct Stack *stack) {
    return stack->top == MAX_SIZE - 1;
}

// Function to push an element onto the stack
void push(struct Stack *stack, int item) {
    if (isFull(stack)) {
        printf("Stack is full. Cannot perform push operation.\n");
        exit(EXIT_FAILURE);
```

```
    }
    stack->arr[++stack->top] = item;
}

// Function to pop an element from the stack
int pop(struct Stack *stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty. Cannot perform pop operation.\n");
        exit(EXIT_FAILURE);
    }
    return stack->arr[stack->top--];
}

// Function to evaluate postfix expression
int evaluatePostfix(char postfix[]) {
    struct Stack stack;
    initializeStack(&stack);

    for (int i = 0; postfix[i] != '\0'; i++) {
        char currentChar = postfix[i];
        if (isdigit(currentChar)) {
            push(&stack, currentChar - '0'); // Convert character to
integer
        } else {
            int operand2 = pop(&stack);
            int operand1 = pop(&stack);
            switch (currentChar) {
                case '+':
                    push(&stack, operand1 + operand2);
                    break;
                case '-':
                    push(&stack, operand1 - operand2);
                    break;
                case '*':
                    push(&stack, operand1 * operand2);
                    break;
                case '/':
                    push(&stack, operand1 / operand2);
                    break;
            }
        }
    }

    return pop(&stack);
}

// Function to evaluate prefix expression
int evaluatePrefix(char prefix[]) {
```

```
    struct Stack stack;
    initializeStack(&stack);

    // Reverse the prefix expression and process it as postfix
    for (int i = 0; prefix[i] != '\0'; i++);

    for (int j = i - 1; j >= 0; j--) {
        char currentChar = prefix[j];
        if (isdigit(currentChar)) {
            push(&stack, currentChar - '0'); // Convert character to
integer
        } else {
            int operand1 = pop(&stack);
            int operand2 = pop(&stack);
            switch (currentChar) {
                case '+':
                    push(&stack, operand1 + operand2);
                    break;
                case '-':
                    push(&stack, operand1 - operand2);
                    break;
                case '*':
                    push(&stack, operand1 * operand2);
                    break;
                case '/':
                    push(&stack, operand1 / operand2);
                    break;
            }
        }
    }

    return pop(&stack);
}
```

Output:

```
Enter a postfix expression: 32+
Result of postfix evaluation: 5

Enter a prefix expression: +32
Result of prefix evaluation: 5
```

Experiment 5. Write a program to convert infix expression to its corresponding postfix and prefix expressions and vice versa

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

#define MAX_SIZE 50

struct Stack {
    char arr[MAX_SIZE];
    int top;
};

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

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

// Function to check if the stack is full
int isFull(struct Stack *stack) {
    return stack->top == MAX_SIZE - 1;
}

// Function to push a character onto the stack
void push(struct Stack *stack, char item) {
    if (isFull(stack)) {
        printf("Stack is full. Cannot perform push operation.\n");
        exit(EXIT_FAILURE);
    }
    stack->arr[++stack->top] = item;
}

// Function to pop a character from the stack
char pop(struct Stack *stack) {
    if (isEmpty(stack)) {
```

```
        printf("Stack is empty. Cannot perform pop operation.\n");
        exit(EXIT_FAILURE);
    }
    return stack->arr[stack->top--];
}

// Function to get the precedence of an operator
int getPrecedence(char op) {
    if (op == '+' || op == '-') {
        return 1;
    } else if (op == '*' || op == '/') {
        return 2;
    }
    return 0;
}

// Function to convert infix expression to postfix expression
void infixToPostfix(char infix[], char postfix[]) {
    struct Stack stack;
    initializeStack(&stack);

    int postfixIndex = 0;

    for (int i = 0; infix[i] != '\0'; i++) {
        char currentChar = infix[i];

        if (isalnum(currentChar)) {
            postfix[postfixIndex++] = currentChar;
        } else if (currentChar == '(') {
            push(&stack, currentChar);
        } else if (currentChar == ')') {
            while (!isEmpty(&stack) && stack.arr[stack.top] != '(') {
                postfix[postfixIndex++] = pop(&stack);
            }
            pop(&stack); // Pop '('
        } else {
            while (!isEmpty(&stack) &&
getPrecedence(stack.arr[stack.top]) >= getPrecedence(currentChar)) {
                postfix[postfixIndex++] = pop(&stack);
            }
            push(&stack, currentChar);
        }
    }

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

```
    postfix[postfixIndex] = '\0';
}

// Function to convert infix expression to prefix expression
void infixToPrefix(char infix[], char prefix[]) {
    // Reverse the infix expression, perform infix to postfix
conversion, and then reverse the result
    int length = strlen(infix);
    char reversedInfix[MAX_SIZE];

    for (int i = 0; i < length; i++) {
        reversedInfix[i] = infix[length - i - 1];
    }

    char reversedPostfix[MAX_SIZE];
    infixToPostfix(reversedInfix, reversedPostfix);

    int postfixIndex = 0;
    for (int i = length - 1; i >= 0; i--) {
        prefix[postfixIndex++] = reversedPostfix[i];
    }

    prefix[postfixIndex] = '\0';
}

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

    printf("Enter an infix expression: ");
    scanf("%s", infix);

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

    infixToPrefix(infix, prefix);
    printf("Prefix expression: %s\n", prefix);

    return 0;
}
```

Output:

```
Enter an infix expression: a+b*c-(d/e+f)*g
Postfix expression: abc*+de/f+g*-
Prefix expression: -+a+*bc/dfg
```

Experiment 6. Implement a queue using array and develop functions to perform enqueuer and dequeue operations

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

#define MAX_SIZE 5

struct ArrayQueue {
    int queue[MAX_SIZE];
    int front, rear;
};

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

// Function to check if the queue is full
int is_full(struct ArrayQueue *queue) {
    return (queue->rear + 1) % MAX_SIZE == queue->front;
}

// Function to enqueue an element
void enqueue(struct ArrayQueue *queue, int item) {
    if (is_full(queue)) {
        printf("Queue is full. Cannot perform enqueue operation.\n");
        return;
    }
    if (is_empty(queue)) {
        queue->front = queue->rear = 0;
    } else {
        queue->rear = (queue->rear + 1) % MAX_SIZE;
    }
    queue->queue[queue->rear] = item;
    printf("Enqueued: %d\n", item);
}

// Function to dequeue an element
int dequeue(struct ArrayQueue *queue) {
    if (is_empty(queue)) {
        printf("Queue is empty. Cannot perform dequeue operation.\n");
        return -1; // Assuming -1 is not a valid element in the queue
```

```
    }
    int item = queue->queue[queue->front];
    if (queue->front == queue->rear) {
        queue->front = queue->rear = -1;
    } else {
        queue->front = (queue->front + 1) % MAX_SIZE;
    }
    printf("Dequeued: %d\n", item);
    return item;
}

// Function to display the current state of the queue
void display(struct ArrayQueue *queue) {
    if (is_empty(queue)) {
        printf("Queue is empty.\n");
        return;
    }
    int i = queue->front;
    while (1) {
        printf("%d ", queue->queue[i]);
        if (i == queue->rear) {
            break;
        }
        i = (i + 1) % MAX_SIZE;
    }
    printf("\n");
}

int main() {
    struct ArrayQueue queue;
    queue.front = queue.rear = -1;

    enqueue(&queue, 10);
    enqueue(&queue, 20);
    enqueue(&queue, 30);
    display(&queue);

    int dequeued_item = dequeue(&queue);
    if (dequeued_item != -1) {
        printf("Dequeued item: %d\n", dequeued_item);
    }

    display(&queue);

    enqueue(&queue, 40);
    enqueue(&queue, 50);
    display(&queue);
```

```
    enqueue(&queue, 60); // This will print "Queue is full" since the
capacity is 5

    return 0;
}
```

Output:

```
Enqueued: 10
Enqueued: 20
Enqueued: 30
10 20 30
Dequeued: 10
20 30
Enqueued: 40
Enqueued: 50
20 30 40 50
Queue is full. Cannot perform enqueue operation.
```

Experiment 7. Implement a singly linked list and develop functions and develop functions to perform insertion, deletion and linear search operations

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

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

// Function to create a new node with the given data
struct Node* createNode(int data) {
```

```
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node at the beginning of the linked list
struct Node* insertAtBeginning(struct Node* head, int data) {
    struct Node* newNode = createNode(data);
    newNode->next = head;
    return newNode;
}

// Function to insert a node at the end of the linked list
struct Node* insertAtEnd(struct Node* head, int data) {
    struct Node* newNode = createNode(data);

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

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

    current->next = newNode;
    return head;
}

// Function to delete a node with the given data from the linked list
struct Node* deleteNode(struct Node* head, int data) {
    if (head == NULL) {
        printf("List is empty. Cannot perform deletion.\n");
        return NULL;
    }

    struct Node* current = head;
    struct Node* previous = NULL;

    while (current != NULL && current->data != data) {
        previous = current;
        current = current->next;
    }
```

```
    if (current == NULL) {
        printf("Node with data %d not found. Cannot perform
deletion.\n", data);
        return head;
    }

    if (previous == NULL) {
        // Node to be deleted is the first node
        head = current->next;
    } else {
        // Node to be deleted is not the first node
        previous->next = current->next;
    }

    free(current);
    printf("Node with data %d deleted.\n", data);
    return head;
}

// Function to perform linear search for a node with the given data
int linearSearch(struct Node* head, int data) {
    struct Node* current = head;
    int position = 0;

    while (current != NULL) {
        if (current->data == data) {
            printf("Node with data %d found at position %d.\n", data,
position);
            return position;
        }
        current = current->next;
        position++;
    }

    printf("Node with data %d not found in the list.\n", data);
    return -1;
}

// Function to display the linked list
void displayList(struct Node* head) {
    struct Node* current = head;

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

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

// Function to free the memory allocated for the linked list
void freeList(struct Node* head) {
    struct Node* current = head;
    struct Node* next;

    while (current != NULL) {
        next = current->next;
        free(current);
        current = next;
    }
}

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

    // Insert at the beginning
    head = insertAtBeginning(head, 10);
    head = insertAtBeginning(head, 20);
    head = insertAtBeginning(head, 30);
    printf("Linked List after insertions at the beginning:\n");
    displayList(head);

    // Insert at the end
    head = insertAtEnd(head, 40);
    head = insertAtEnd(head, 50);
    printf("Linked List after insertions at the end:\n");
    displayList(head);

    // Delete a node
    head = deleteNode(head, 20);
    printf("Linked List after deletion:\n");
    displayList(head);

    // Linear search
    int position = linearSearch(head, 40);

    // Free the memory allocated for the linked list
    freeList(head);

    return 0;
}
```

Output:

```
Linked List after insertions at the beginning:
30 -> 20 -> 10 -> NULL
Linked List after insertions at the end:
30 -> 20 -> 10 -> 40 -> 50 -> NULL
Node with data 20 deleted.
Linked List after deletion:
30 -> 10 -> 40 -> 50 -> NULL
Node with data 40 found at position 2.
```

Experiment 8: Implement a doubly linked list and develop functions and develop functions to perform insertion, deletion and linear search operations

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

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

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

// Function to insert a node at the beginning of the doubly linked list
struct Node* insertAtBeginning(struct Node* head, int data) {
    struct Node* newNode = createNode(data);

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

    newNode->next = head;
    head->prev = newNode;
    return newNode;
}

// Function to insert a node at the end of the doubly linked list
struct Node* insertAtEnd(struct Node* head, int data) {
```

```
    struct Node* newNode = createNode(data);

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

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

    current->next = newNode;
    newNode->prev = current;
    return head;
}

// Function to delete a node with the given data from the doubly linked list
struct Node* deleteNode(struct Node* head, int data) {
    if (head == NULL) {
        printf("List is empty. Cannot perform deletion.\n");
        return NULL;
    }

    struct Node* current = head;

    while (current != NULL && current->data != data) {
        current = current->next;
    }

    if (current == NULL) {
        printf("Node with data %d not found. Cannot perform deletion.\n", data);
        return head;
    }

    if (current->prev != NULL) {
        current->prev->next = current->next;
    } else {
        // Node to be deleted is the first node
        head = current->next;
    }

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

    free(current);
    printf("Node with data %d deleted.\n", data);
    return head;
}

// Function to perform linear search for a node with the given data
int linearSearch(struct Node* head, int data) {
    struct Node* current = head;
    int position = 0;

    while (current != NULL) {
```

```
        if (current->data == data) {
            printf("Node with data %d found at position %d.\n", data, position);
            return position;
        }
        current = current->next;
        position++;
    }

    printf("Node with data %d not found in the list.\n", data);
    return -1;
}

// 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("NULL\n");
}

// Function to free the memory allocated for the doubly linked list
void freeList(struct Node* head) {
    struct Node* current = head;
    struct Node* next;

    while (current != NULL) {
        next = current->next;
        free(current);
        current = next;
    }
}

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

    // Insert at the beginning
    head = insertAtBeginning(head, 10);
    head = insertAtBeginning(head, 20);
    head = insertAtBeginning(head, 30);
    printf("Doubly Linked List after insertions at the beginning:\n");
    displayList(head);

    // Insert at the end
    head = insertAtEnd(head, 40);
    head = insertAtEnd(head, 50);
    printf("Doubly Linked List after insertions at the end:\n");
    displayList(head);

    // Delete a node
    head = deleteNode(head, 20);
    printf("Doubly Linked List after deletion:\n");
    displayList(head);
```

```
    // Linear search
    int position = linearSearch(head, 40);

    // Free the memory allocated for the doubly linked list
    freeList(head);

    return 0;
}
```

Output:

```
Doubly Linked List after insertions at the beginning:
30 <-> 20 <-> 10 <-> NULL
Doubly Linked List after insertions at the end:
30 <-> 20 <-> 10 <-> 40 <-> 50 <-> NULL
Node with data 20 deleted.
Doubly Linked List after deletion:
30 <-> 10 <-> 40 <-> 50 <-> NULL
Node with data 40 found at position 2.
```

Experiment 9: Implement a circular linked list and develop functions and develop functions to perform insertion, deletion and linear search operations

Code:

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

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

typedef struct Node Node;

Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

void append(Node** head_ref, int data) {
```

```
    Node* new_node = createNode(data);
    if (*head_ref == NULL) {
        *head_ref = new_node;
        new_node->next = *head_ref;
    } else {
        Node* temp = *head_ref;
        while (temp->next != *head_ref)
            temp = temp->next;
        temp->next = new_node;
        new_node->next = *head_ref;
    }
}

void deleteNode(Node** head_ref, int key) {
    if (*head_ref == NULL) return;
    Node* temp = *head_ref, *prev = NULL;
    while (temp->data != key) {
        if (temp->next == *head_ref) return;
        prev = temp;
        temp = temp->next;
    }
    if (temp->next == *head_ref && prev == NULL) {
        *head_ref = NULL;
        free(temp);
        return;
    }
    if (temp == *head_ref) {
        prev = (*head_ref)->next;
        while (prev->next != *head_ref) prev = prev->next;
        prev->next = (*head_ref)->next;
        free(*head_ref);
        *head_ref = prev->next;
    } else if (temp->next == *head_ref) {
        prev->next = *head_ref;
        free(temp);
    } else {
        prev->next = temp->next;
        free(temp);
    }
}

int search(Node* head, int key) {
    Node* current = head;
    if (head == NULL) return 0;
    do {
        if (current->data == key) return 1;
        current = current->next;
    } while (current != head);
```

```
    return 0;
}

void display(Node* head) {
    if (head == NULL) return;
    Node* temp = head;
    do {
        printf("%d ", temp->data);
        temp = temp->next;
    } while (temp != head);
    printf("\n");
}

int main() {
    Node* head = NULL;
    append(&head, 1);
    append(&head, 2);
    append(&head, 3);
    display(head);  // Output: 1 2 3

    deleteNode(&head, 2);
    display(head);  // Output: 1 3

    printf("%d\n", search(head, 3));  // Output: 1
    printf("%d\n", search(head, 2));  // Output: 0

    return 0;
}
```

Output:

```
1 2 3
1 3
1
0
```