Program 1: Linear Search in C

Objective: To implement linear search using arrays.

1. Linear Search (Iterative)

```
#include <stdio.h>
int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key)
            return i;
    }
    return -1;
}
int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int n = sizeof(arr)/sizeof(arr[0]);
    int key = 30;
    int result = linearSearch(arr, n, key);
    if (result != -1)
        printf("Element found at index %d", result);
    else
        printf("Element not found");
    return 0;
}
```

Program 2: Binary Search (Iterative) in C

Objective: To implement binary search using iteration.

2. Binary Search (Iterative)

```
#include <stdio.h>
int binarySearchIterative(int arr[], int n, int key) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == key)
            return mid;
        else if (arr[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
```

```
    return -1;
}
int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int n = sizeof(arr)/sizeof(arr[0]);
    int key = 40;
    int result = binarySearchIterative(arr, n, key);
    if (result != -1)
        printf("Element found at index %d", result);
    else
        printf("Element not found");
    return 0;
}
```

Program 3: Binary Search (Recursive) in C

Objective: To implement binary search using iteration.

3. Binary Search (Recursive)

```
#include <stdio.h>
int binarySearchRecursive(int arr[], int low, int high, int key) {
    if (low > high)
        return -1;
    int mid = (low + high) / 2;
    if (arr[mid] == key)
        return mid;
    else if (arr[mid] > key)
        return binarySearchRecursive(arr, low, mid - 1, key);
    else
        return binarySearchRecursive(arr, mid + 1, high, key);
}
int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int n = sizeof(arr)/sizeof(arr[0]);
    int key = 20;
    int result = binarySearchRecursive(arr, 0, n - 1, key);
    if (result != -1)
        printf("Element found at index %d", result);
    else
        printf("Element not found");
    return 0;
}
```

Program 4: Factorial (Iterative)

Objective: To calculate factorial of a number using iteration.

```
#include <stdio.h>
int main() {
   int n = 5;
   long long fact = 1;
   for (int i = 1; i <= n; i++) {
      fact *= i;
   }
   printf("Factorial of %d = %lld", n, fact);
   return 0;
}
```

Program 5: Factorial (Recursive)

Objective: To calculate factorial of a number using recursion.

```
#include <stdio.h>
long long factorial(int n) {
   if (n == 0 || n == 1)
      return 1;
   return n * factorial(n - 1);
}
int main() {
   int n = 5;
   printf("Factorial of %d = %lld", n, factorial(n));
   return 0;
}
```

Program 6: Fibonacci Series (Iterative)

Objective: To generate Fibonacci series using iteration.

```
#include <stdio.h>
int main() {
   int n = 10, t1 = 0, t2 = 1, nextTerm;
   printf("Fibonacci Series: ");
   for (int i = 1; i <= n; ++i) {
      printf("%d ", t1);
      nextTerm = t1 + t2;
      t1 = t2;
      t2 = nextTerm;
   }
```

```
    return 0;
}
```

Program 7: Fibonacci Series (Recursive)

Objective: To generate Fibonacci series using recursion.

```
#include <stdio.h>
int fibonacci(int n) {
    if (n <= 1)
        return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
int main() {
    int n = 10;
    printf("Fibonacci Series: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", fibonacci(i));
    }
    return 0;
}
```

Program 8: Insertion Sort in C

Objective: To implement insertion sort using arrays.

1. Insertion Sort in C

```
#include <stdio.h>

void insertionSort(int A[], int n) {

int i, j, key;

for (j = 1; j < n; j++) {

key = A[j];

i = j - 1;

while (i >= 0 && A[i] > key) {

A[i + 1] = A[i];

i = i - 1;

}
```

```
A[i + 1] = key;
}
}
void printArray(int A[], int n) {
for (int i = 0; i < n; i++)
printf("%d ", A[i]);
printf("\n");
}
int main() {
int A[] = {5, 2, 9, 1, 5, 6};
int n = sizeof(A) / sizeof(A[0]);
printf("Original array:\n");
printArray(A, n);
insertionSort(A, n);
printf("Sorted array:\n");
printArray(A, n);
return 0;
}
```

Program 9: Bubble Sort in C

Objective: To implement bubble sort using arrays.

2. Bubble Sort in C

```
#include <stdio.h>
void bubbleSort(int list[], int n) {
for (int i = 0; i < n - 1; i++) {
for (int j = 0; j < n - i - 1; j++) {
if (list[j] > list[j + 1]) {
```

```
int temp = list[j];

list[j] = list[j + 1];

list[j + 1] = temp;

}

}

}

}

void printArray(int A[], int n) {

for (int i = 0; i < n; i++)

printf("%d ", A[i]);

printf("\n");

}

int main() {

int list[] = {64, 34, 25, 12, 22, 11, 90};

int n = sizeof(list) / sizeof(list[0]);

printf("Original array:\n");

printArray(list, n);

bubbleSort(list, n);

printf("Sorted array:\n");

printArray(list, n);

return 0;

}
```

Program 10: Selection Sort in C

Objective: To implement selection sort using arrays.

3. Selection Sort in C

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

```
void selectionSort(int A[], int n) {
int i, j, minIndex, temp;
for (i = 0; i < n - 1; i++) {
minIndex = i;
for (j = i + 1; j < n; j++) {
if (A[j] < A[minIndex])
minIndex = j;
}
temp = A[i];
A[i] = A[minIndex];
A[minIndex] = temp;
}
}
void printArray(int A[], int n) {
for (int i = 0; i < n; i++)
printf("%d ", A[i]);
printf("\n");
}
int main() {
int A[] = {29, 10, 14, 37, 13};
int n = sizeof(A) / sizeof(A[0]);
printf("Original array:\n");
printArray(A, n);
selectionSort(A, n);
printf("Sorted array:\n");
printArray(A, n);
```

```c
return 0;

}
```

Program 11: Merge Sort in C

Objective: To implement merge sort using arrays.

4. Merge Sort in C

```c
#include <stdio.h>

void merge(int A[], int low, int mid, int high) {

int B[100];

int i = low, j = mid + 1, k = low;

while (i <= mid && j <= high) {

if (A[i] <= A[j])

B[k++] = A[i++];

else

B[k++] = A[j++];

}

while (i <= mid)

B[k++] = A[i++];

while (j <= high)

B[k++] = A[j++];

for (k = low; k <= high; k++)

A[k] = B[k];

}

void mergeSort(int A[], int low, int high) {

if (low < high) {

int mid = (low + high) / 2;

mergeSort(A, low, mid);
```

```
mergeSort(A, mid + 1, high);

merge(A, low, mid, high);

}

}

void printArray(int A[], int n) {

for (int i = 0; i < n; i++)

printf("%d ", A[i]);

printf("\n");

}

int main() {

int A[] = {38, 27, 43, 3, 9, 82, 10};

int n = sizeof(A) / sizeof(A[0]);

printf("Original array:\n");

printArray(A, n);

mergeSort(A, 0, n - 1);

printf("Sorted array:\n");

printArray(A, n);

return 0;

}
```

Program 12: Quick Sort in C

Objective: To implement quick sort using arrays.

5. Quick Sort in C

```
#include <stdio.h>

int partition(int A[], int p, int r) {

int x = A[r];

int i = p - 1;
```

```
int temp;
for (int j = p; j < r; j++) {
if (A[j] <= x) {
i++;
temp = A[i];
A[i] = A[j];
A[j] = temp;
}
}
temp = A[i + 1];
A[i + 1] = A[r];
A[r] = temp;
return (i + 1);
}
void quickSort(int A[], int p, int r) {
if (p < r) {
int q = partition(A, p, r);
quickSort(A, p, q - 1);
quickSort(A, q + 1, r);
}
}
void printArray(int A[], int n) {
for (int i = 0; i < n; i++)
printf("%d ", A[i]);
printf("\n");
}
```

```
int main() {
int A[] = {10, 7, 8, 9, 1, 5};
int n = sizeof(A) / sizeof(A[0]);
printf("Original array:\n");
printArray(A, n);
quickSort(A, 0, n - 1);
printf("Sorted array:\n");
printArray(A, n);
return 0;
}
```

Program 13: Insert Node at Beginning

Objective: To insert a node at the beginning of a linked list.

```
#include <stdio.h>
#include <stdlib.h>
// Define node structure
struct Node {
int data;
struct Node* next;
};
// Function to insert at beginning
struct Node* insertAtBeginning(struct Node* head, int x) {
// Step 1: Allocate memory for new node
struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
// Step 2: Assign data
newNode->data = x;
// Step 3: Point new node to current head
```

```c
newNode->next = head;

// Step 4: Update head

head = newNode;

return head;

}

// Function to print list

void printList(struct Node* head) {

struct Node* temp = head;

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

// Example usage

int main() {

struct Node* head = NULL;

head = insertAtBeginning(head, 10);

head = insertAtBeginning(head, 20);

head = insertAtBeginning(head, 30);

printList(head);

return 0;

}
```

Program 14: Insert Node at End (Linked List)

Objective: To insert a node at the End of a linked list.

7. Insert Node at the End of Linked List

```
#include <stdio.h>

#include <stdlib.h>

// Define node structure

struct Node {

int data;

struct Node* next;

};

// Function to insert node at end

struct Node* insertAtEnd(struct Node* head, int x) {

// Step 1: Allocate memory

struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));

newNode->data = x;

newNode->next = NULL;

// Step 2: If list is empty

if (head == NULL) {

head = newNode;

return head;

}

// Step 3: Traverse till last node

struct Node* temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

// Step 4: Attach new node at end

temp->next = newNode;

return head;
```

```
}
// Function to print list
void printList(struct Node* head) {
struct Node* temp = head;
while (temp != NULL) {
printf("%d -> ", temp->data);
temp = temp->next;
}
printf("NULL\n");
}
// Example usage
int main() {
struct Node* head = NULL;
head = insertAtEnd(head, 10);
head = insertAtEnd(head, 20);
head = insertAtEnd(head, 30);
printList(head);
return 0;
}
```

Program 15: Insert Node at Specific location (Linked List)

Objective: To insert a node at the Specified Location of a linked list.

```
#include <stdio.h>
#include <stdlib.h>
// Define node structure
struct Node {
int data;
```

```
struct Node* next;
};
// Function to insert node at a given position
struct Node* insertAtPosition(struct Node* head, int x, int pos) {
struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
newNode->data = x;
newNode->next = NULL;
// Case 1: Insert at beginning
if (pos == 1) {
newNode->next = head;
head = newNode;
return head;
}
// Case 2: Insert at given position
struct Node* temp = head;
for (int i = 1; i < pos - 1 && temp != NULL; i++) {
temp = temp->next;
}
if (temp == NULL) {
printf("Position out of range!\n");
free(newNode);
return head;
}
newNode->next = temp->next;
temp->next = newNode;
return head;
```

```
}
// Function to print list
void printList(struct Node* head) {
struct Node* temp = head;
while (temp != NULL) {
printf("%d -> ", temp->data);
temp = temp->next;
}
printf("NULL\n");
}
// Example usage
int main() {
struct Node* head = NULL;
// Insert nodes
head = insertAtPosition(head, 10, 1); // 10
head = insertAtPosition(head, 20, 2); // 10 -> 20
head = insertAtPosition(head, 30, 2); // 10 -> 30 -> 20
head = insertAtPosition(head, 40, 1); // 40 -> 10 -> 30 -> 20
printList(head);
return 0;
}
```

Program 16: Delete Node at Beginning

Objective: To delete a node from the beginning of a linked list.

```
#include <stdio.h>
#include <stdlib.h>
// Define node structure
```

```
struct Node {
int data;
struct Node* next;
};
// Function to delete node from beginning
struct Node* deleteAtBeginning(struct Node* head) {
if (head == NULL) {
printf("List is empty. Nothing to delete.\n");
return NULL;
}
struct Node* temp = head;
head = head->next;  // Move head to next node
free(temp);        // Free old head
return head;
}
// Function to print list
void printList(struct Node* head) {
struct Node* temp = head;
while (temp != NULL) {
printf("%d -> ", temp->data);
temp = temp->next;
}
printf("NULL\n");
}
// Example usage
int main() {
```

```
// Create linked list manually: 10 -> 20 -> 30

struct Node* head = (struct Node*) malloc(sizeof(struct Node));

head->data = 10;

head->next = (struct Node*) malloc(sizeof(struct Node));

head->next->data = 20;

head->next->next = (struct Node*) malloc(sizeof(struct Node));

head->next->next->data = 30;

head->next->next->next = NULL;

printf("Original List: ");

printList(head);

head = deleteAtBeginning(head); // delete 10

printf("After deleting first node: ");

printList(head);

return 0;

}
```

Program 17: Delete Node at End

Objective: To delete a node from the end of a linked list.

```
#include <stdio.h>

#include <stdlib.h>

// Define node structure

struct Node {

int data;

struct Node* next;

};

// Function to delete node from end

struct Node* deleteAtEnd(struct Node* head) {
```

```
// Case 1: Empty list
if (head == NULL) {
printf("List is empty. Nothing to delete.\n");
return NULL;
}
// Case 2: Only one node
if (head->next == NULL) {
free(head);
return NULL;
}
// Case 3: More than one node
struct Node* temp = head;
while (temp->next->next != NULL) {
temp = temp->next;
}
free(temp->next);   // Delete last node
temp->next = NULL;  // Set new end to NULL
return head;
}
// Function to print list
void printList(struct Node* head) {
struct Node* temp = head;
while (temp != NULL) {
printf("%d -> ", temp->data);
temp = temp->next;
}
```

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

}

// Example usage

int main() {

// Create linked list manually: 10 -> 20 -> 30

struct Node* head = (struct Node*) malloc(sizeof(struct Node));

head->data = 10;

head->next = (struct Node*) malloc(sizeof(struct Node));

head->next->data = 20;

head->next->next = (struct Node*) malloc(sizeof(struct Node));

head->next->next->data = 30;

head->next->next->next = NULL;

printf("Original List: ");

printList(head);

head = deleteAtEnd(head); // delete 30

printf("After deleting last node: ");

printList(head);

return 0;

}
```

Program 18: Delete Node at Specified Location

Objective: To delete a node from the Specified Location of a linked list.

```
#include <stdio.h>

#include <stdlib.h>

// Define node structure

struct Node {

int data;
```

```c
struct Node* next;

};

// Function to delete node at a given position

struct Node* deleteAtPosition(struct Node* head, int pos) {

if (head == NULL) {

printf("List is empty. Nothing to delete.\n");

return NULL;

}

// Case 1: Delete first node

if (pos == 1) {

struct Node* temp = head;

head = head->next;

free(temp);

return head;

}

// Case 2: Delete at given position

struct Node* temp = head;

for (int i = 1; i < pos - 1 && temp != NULL; i++) {

temp = temp->next;

}

// If position is invalid

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

printf("Position out of range!\n");

return head;

}

struct Node* toDelete = temp->next;
```

```
temp->next = toDelete->next;
free(toDelete);
return head;
}
// Function to print list
void printList(struct Node* head) {
struct Node* temp = head;
while (temp != NULL) {
printf("%d -> ", temp->data);
temp = temp->next;
}
printf("NULL\n");
}
// Example usage
int main() {
// Create linked list manually: 10 -> 20 -> 30 -> 40
struct Node* head = (struct Node*) malloc(sizeof(struct Node));
head->data = 10;
head->next = (struct Node*) malloc(sizeof(struct Node));
head->next->data = 20;
head->next->next = (struct Node*) malloc(sizeof(struct Node));
head->next->next->data = 30;
head->next->next->next = (struct Node*) malloc(sizeof(struct Node));
head->next->next->next->data = 40;
head->next->next->next->next = NULL;
printf("Original List: ");
```

```
printList(head);

head = deleteAtPosition(head, 2); // delete 20

printf("After deleting at position 2: ");

printList(head);

head = deleteAtPosition(head, 1); // delete 10

printf("After deleting at position 1: ");

printList(head);

head = deleteAtPosition(head, 5); // invalid position

printf("After trying invalid delete: ");

printList(head);

return 0;

}
```

Program 19: Concatenate Two Linked Lists

Objective: To concatenate two linked lists.

```
#include <stdio.h>

#include <stdlib.h>

// Define Node structure

struct Node {

int data;

struct Node* next;

};

// Function to print linked list

void printList(struct Node* head) {

struct Node* temp = head;

while (temp != NULL) {

printf("%d -> ", temp->data);
```

```
temp = temp->next;
}
printf("NULL\n");
}
// Function to concatenate two linked lists
struct Node* concatenate(struct Node* head1, struct Node* head2) {
if (head1 == NULL) return head2;
if (head2 == NULL) return head1;
struct Node* temp = head1;
while (temp->next != NULL) {
temp = temp->next;
}
temp->next = head2;
return head1;
}
int main() {
// Manually create first linked list: 10 -> 20 -> 30 -> 40
struct Node* head1 = (struct Node*) malloc(sizeof(struct Node));
head1->data = 10;
head1->next = (struct Node*) malloc(sizeof(struct Node));
head1->next->data = 20;
head1->next->next = (struct Node*) malloc(sizeof(struct Node));
head1->next->next->data = 30;
head1->next->next->next = (struct Node*) malloc(sizeof(struct Node));
head1->next->next->next->data = 40;
head1->next->next->next->next = NULL;
```

```
// Manually create second linked list: 50 -> 60

struct Node* head2 = (struct Node*) malloc(sizeof(struct Node));

head2->data = 50;

head2->next = (struct Node*) malloc(sizeof(struct Node));

head2->next->data = 60;

head2->next->next = NULL;

// Print the original lists

printf("First List: ");

printList(head1);

printf("Second List: ");

printList(head2);

// Concatenate the lists

struct Node* concatenatedHead = concatenate(head1, head2);

printf("Concatenated List: ");

printList(concatenatedHead);

return 0;

}
```

Program 20: Add Two Polynomials using Linked List

Objective: To add two polynomials using linked lists.

```
#include <stdio.h>

#include <stdlib.h>

// Define node structure

struct Node {

int coeff;

int pow;

struct Node* next;
```

```
};
// Function to display polynomial
void display(struct Node* head) {
struct Node* temp = head;
while (temp != NULL) {
printf("%dx^%d", temp->coeff, temp->pow);
temp = temp->next;
if (temp != NULL && temp->coeff >= 0)
printf(" + ");
}
printf("\n");
}
// Function to add two polynomials
struct Node* addPolynomials(struct Node* poly1, struct Node* poly2) {
struct Node* result = NULL;
struct Node* temp = NULL;
while (poly1 != NULL && poly2 != NULL) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->next = NULL;
if (poly1->pow > poly2->pow) {
newNode->coeff = poly1->coeff;
newNode->pow = poly1->pow;
poly1 = poly1->next;
}
else if (poly1->pow < poly2->pow) {
newNode->coeff = poly2->coeff;
```

```
}
newNode->pow = poly2->pow;
poly2 = poly2->next;
}
else { // same power
newNode->coeff = poly1->coeff + poly2->coeff;
newNode->pow = poly1->pow;
poly1 = poly1->next;
poly2 = poly2->next;
}
if (result == NULL) {
result = temp = newNode;
} else {
temp->next = newNode;
temp = newNode;
}
}
// Copy remaining terms
while (poly1 != NULL) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->coeff = poly1->coeff;
newNode->pow = poly1->pow;
newNode->next = NULL;
temp->next = newNode;
temp = newNode;
poly1 = poly1->next;
}
```

```
while (poly2 != NULL) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->coeff = poly2->coeff;
newNode->pow = poly2->pow;
newNode->next = NULL;
temp->next = newNode;
temp = newNode;
poly2 = poly2->next;
}
return result;
}
// Main function
int main() {
// Manually create Polynomial 1: 5x^3 + 4x^2 + 2x^1
struct Node* poly1 = (struct Node*)malloc(sizeof(struct Node));
poly1->coeff = 5; poly1->pow = 3;
poly1->next = (struct Node*)malloc(sizeof(struct Node));
poly1->next->coeff = 4; poly1->next->pow = 2;
poly1->next->next = (struct Node*)malloc(sizeof(struct Node));
poly1->next->next->coeff = 2; poly1->next->next->pow = 1;
poly1->next->next->next = NULL;
// Manually create Polynomial 2: 5x^2 + 5x^1 + 5
struct Node* poly2 = (struct Node*)malloc(sizeof(struct Node));
poly2->coeff = 5; poly2->pow = 2;
poly2->next = (struct Node*)malloc(sizeof(struct Node));
poly2->next->coeff = 5; poly2->next->pow = 1;
```

```
poly2->next->next = (struct Node*)malloc(sizeof(struct Node));
poly2->next->next->coeff = 5; poly2->next->next->pow = 0;
poly2->next->next->next = NULL;
printf("Polynomial 1: ");
display(poly1);
printf("Polynomial 2: ");
display(poly2);
// Add both polynomials
struct Node* sum = addPolynomials(poly1, poly2);
printf("Sum: ");
display(sum);
return 0;
}
```

Program 21: Stack operations using Array

Objective: To implement stack operations using arrays.

```
#include <stdio.h>
#define MAX 5
int stack[MAX];
int top = -1;
// Push operation
void push(int value) {
if (top == MAX - 1) {
printf("Stack Overflow! Cannot push %d\n", value);
} else {
top++;
stack[top] = value;
```

```
printf("%d pushed onto stack.\n", value);
}
}
// Pop operation
void pop() {
if (top == -1) {
printf("Stack Underflow! Stack is empty.\n");
} else {
printf("%d popped from stack.\n", stack[top]);
top--;
}
}
// Display operation
void display() {
if (top == -1) {
printf("Stack is empty.\n");
} else {
printf("Stack elements: ");
for (int i = top; i >= 0; i--) {
printf("%d ", stack[i]);
}
printf("\n");
}
}
int main() {
// Perform stack operations manually (no menu)
```

```
push(10);

push(20);

push(30);

display();

pop();

display();

push(40);

display();

return 0;

}
```

Program 22: Circular Queue using

Objective: To implement circular queue

```
 #include <stdio.h>

#define MAX_SIZE  5   // small size for easy demonstration

int queue[MAX_SIZE];

int front = -1, rear = -1;

// Function to insert element (enqueue)

void enqueue(int value) {

if ((rear + 1) % MAX_SIZE == front) {

.

printf("Queue Overflow\n");

return;

}

if (front == -1 && rear == -1) {

front = rear = 0;

} else {
```

```
rear = (rear + 1) % MAX_SIZE;
}
queue[rear] = value;
}
// Function to delete element (dequeue)
void dequeue() {
if (front == -1) {
printf("Queue Underflow\n");
return;
}
if (front == rear) {
front = rear = -1;
} else {
front = (front + 1) % MAX_SIZE;
}
}
// Function to display the queue
void display() {
if (front == -1) {
printf("Queue is empty\n");
return;
}
printf("Queue elements: ");
int i = front;
while (1) {
printf("%d ", queue[i]);
```

```
if (i == rear) break;

i = (i + 1) % MAX_SIZE;

}

printf("\n");

}

int main() {

enqueue(10);

enqueue(20);

enqueue(30);

enqueue(40);

display();

dequeue();

dequeue();

display();

enqueue(50);

enqueue(60);

display();

return 0;

}
```

Program 23: Stack using Linked List

Objective: To implement Stack operations using linked list.

```
#include <stdio.h>

#include <stdlib.h>

// Define node structure

struct Node {

int data;
```

```
struct Node* next;
};
struct Node* top = NULL;
// Push operation
void push(int value) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
if (!newNode) {
printf("Stack Overflow\n");
return;
}
newNode->data = value;
newNode->next = top;
top = newNode;
}
// Pop operation
void pop() {
if (top == NULL) {
printf("Stack Underflow\n");
return;
}
struct Node* temp = top;
printf("Popped: %d\n", temp->data);
top = top->next;
free(temp);
}
// Peek (top element)
```

```
void peek() {
if (top == NULL) {
printf("Stack is empty\n");
} else {
printf("Top element: %d\n", top->data);
}
}
// Display stack
void display() {
struct Node* temp = top;
if (temp == NULL) {
printf("Stack is empty\n");
return;
}
printf("Stack elements: ");
while (temp != NULL) {
printf("%d ", temp->data);
temp = temp->next;
}
printf("\n");
}
int main() {
push(10);
push(20);
push(30);
display();
```

```
peek();

pop();

display();

return 0;

}
```

Program 24: Queue using Linked List

Objective: To implement queue using linked list.

```
#include <stdio.h>

#include <stdlib.h>

// Define node structure

struct Node {

int data;

struct Node* next;

};

struct Node* front = NULL;

struct Node* rear = NULL;

// Enqueue operation

void enqueue(int value) {

struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

if (!newNode) {

printf("Queue Overflow\n");

return;

}

newNode->data = value;

newNode->next = NULL;

if (rear == NULL) {
```

```
front = rear = newNode;
} else {
rear->next = newNode;
rear = newNode;
}
}
// Dequeue operation
void dequeue() {
if (front == NULL) {
printf("Queue Underflow\n");
return;
}
struct Node* temp = front;
printf("Dequeued: %d\n", temp->data);
front = front->next;
if (front == NULL) {
rear = NULL;
}
free(temp);
}
// Display queue
void display() {
if (front == NULL) {
printf("Queue is empty\n");
return;
}
```

```
struct Node* temp = front;

printf("Queue elements: ");

while (temp != NULL) {

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

temp = temp->next;

}

printf("\n");

}

int main() {

enqueue(10);

enqueue(20);

enqueue(30);

display();

dequeue();

display();

return 0;

}
```