# Galgotias College of Engineering & Technology

**Affiliated to Dr.A.P.J AKTU, Lucknow**

# LAB FILE

## Data Structure Lab
## (BCS351)
## (Odd Semester,2024-25)


**Submitted To:-**

**Faculty Name:- Dr. Y.B. Singh**

**Submitted By:-**

**Roll No: 2300971630008**

**Name:- Ali Abbas**

**Branch:- AI-DS**

# BUBBLE SORT

```
#include <stdio.h>

void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {

        int key = 0;

        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
                key = 1;
            }
        }

        if (key == 0) {
            break;
        }
    }
}
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Unsorted array: \n");
    printArray(arr, n);

    bubbleSort(arr, n);

    printf("Sorted array: \n");
    printArray(arr, n);

    return 0;
}
```

Output Clear

```
Unsorted array:
64 34 25 12 22 11 90
Sorted array:
11 12 22 25 34 64 90
```

# INSERTION SORT

```
#include <stdio.h>

void insertionSort(int arr[], int size) {
    for (int i = 1; i < size; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
       arr[j + 1] = key;
    }
}
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {

    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int size = sizeof(arr) / sizeof(arr[0]);

    printf("Unsorted array:\n");
    printArray(arr, size);

    insertionSort(arr, size);

    printf("Sorted array:\n");
    printArray(arr, size);

    return 0;
}
```

Output Clear

```
Unsorted array:
64 34 25 12 22 11 90
Sorted array:
11 12 22 25 34 64 90
```

# QUICK SORT

```
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int low, int high) {
   int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int size = sizeof(arr) / sizeof(arr[0]);
    printf("Unsorted array:\n");
    printArray(arr, size);
    quickSort(arr, 0, size - 1);
    printf("Sorted array:\n");
    printArray(arr, size);
    return 0;
}
```

Output Clear

```
Unsorted array:
64 34 25 12 22 11 90
Sorted array:
11 12 22 25 34 64 90
```

# SELECTION SORT

```
#include <stdio.h>

void selectionSort(int arr[], int n) {
    int i, j, minIndex, temp;

    for (i = 0; i < n - 1; i++) {
        minIndex = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
         if (minIndex != i) {
            temp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = temp;
        }
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original Array: \n");
    printArray(arr, n);
    selectionSort(arr, n);
    printf("Sorted Array: \n");
    printArray(arr, n);
    return 0;
}
```

Output Clear

```
Original Array:
64 25 12 22 11
Sorted Array:
11 12 22 25 64
```

# MERGE SORT

```
#include <stdio.h>

void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

void printArray(int A[], int size) {
    for (int i = 0; i < size; ++i)
        printf("%d  ", A[i]);
    printf("\n");
}
```

```
int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    printf("\nSorted array is \n");
    printArray(arr, arr_size);
    return 0;
}
```

Output Clear

```
Original Array:
64 25 12 22 11
Sorted Array:
11 12 22 25 64
```

**LINEAR SEARCH**

```
#include <stdio.h>

int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i;
        }
    }
    return -1;
}
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main() {
    int arr[] = {5, 3, 7, 1, 9, 2};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 7;
    printf("Array: ");
    printArray(arr, size);
    int result = linearSearch(arr, size, target);
    if (result != -1) {
        printf("Element %d found at index %d.\n", target, result);
    } else {
        printf("Element %d not found in the array.\n", target);
    }

    return 0;
}
```

Output Clear

```
Array: 5 3 7 1 9 2
Element 7 found at index 2.
```

**BINARY SEARCH**

```
#include <stdio.h>

int binarySearch(int arr[], int size, int target) {
    int low = 0, high = size - 1;

    while (low <= high) {
        // Find the middle index
        int mid = low + (high - low) / 2;

        if (arr[mid] == target) {
            return mid;  // Element found
        }

        if (arr[mid] > target) {
            high = mid - 1;
        }
        else {
            low = mid + 1;
        }
    }
    return -1;
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 7;
    printf("Array: ");
    printArray(arr, size);

    int result = binarySearch(arr, size, target);

    if (result != -1) {
        printf("Element %d found at index %d.\n", target, result);
    } else {
        printf("Element %d not found in the array.\n", target);
    }
    return 0;
}
```

Output Clear

```
Array: 1 3 5 7 9 11 13 15 17 19
Element 7 found at index 3.
```

**IMPLEMENTATION OF LINKED LIST**

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

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

struct Node* head = NULL;

void insertAtHead(int data);
void insertAtTail(int data);
void insertAtIndex(int data, int index);
void deleteAtIndex(int index);
void display();

int main() {
    int choice, data, index;

    while (1) {
        printf("\nMenu:\n");
        printf("1. Insert at head\n");
        printf("2. Insert at tail\n");
        printf("3. Insert at index\n");
        printf("4. Delete at index\n");
        printf("5. Display\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to insert at head: ");
                scanf("%d", &data);
                insertAtHead(data);
                break;
            case 2:
                printf("Enter data to insert at tail: ");
                scanf("%d", &data);
                insertAtTail(data);
                break;
            case 3:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                printf("Enter index to insert at: ");
                scanf("%d", &index);
                insertAtIndex(data, index);
                break;
            case 4:
                printf("Enter index to delete: ");
                scanf("%d", &index);
                deleteAtIndex(index);
                break;
            case 5:
```

```
            display();
            break;
        case 6:
            exit(0);
        default:
            printf("Wrong choice. Please try again.\n");
      }
   }
   return 0;
}

void insertAtHead(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = head;
   head = newNode;
}

void insertAtTail(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = NULL;

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

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

void insertAtIndex(int data, int index) {
   if (index < 0) {
      printf("Invalid index\n");
      return;
   }

   if (index == 0) {
      insertAtHead(data);
      return;
   }

   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;

   struct Node* temp = head;
   for (int i = 0; temp != NULL && i < index - 1; i++) {
      temp = temp->next;
   }

   if (temp == NULL) {
      printf("Index out of bounds\n");
      free(newNode);
      return;
```

```
    }

    newNode->next = temp->next;
    temp->next = newNode;
}

void deleteAtIndex(int index) {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }

    if (index < 0) {
        printf("Invalid index\n");
        return;
    }

    struct Node* temp = head;

    if (index == 0) {
        head = temp->next;
        free(temp);
        return;
    }

    struct Node* prev = NULL;
    for (int i = 0; temp != NULL && i < index; i++) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Index out of bounds\n");
        return;
    }

    prev->next = temp->next;
    free(temp);
}

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

**ADD POLYNOMIALS:**

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

typedef struct Node {
    int coeff;
    int pow;
    struct Node* next;
} Node;

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

void addNode(Node** head, int coeff, int pow) {
    Node* newNode = createNode(coeff, pow);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}

void printPolynomial(Node* head) {
    Node* temp = head;
    while (temp != NULL) {
        if (temp->coeff < 0 && temp != head) {
            printf(" - ");
            temp->coeff = -temp->coeff;
        } else if (temp != head) {
            printf(" + ");
        }
        printf("%dx^%d", temp->coeff, temp->pow);
        temp = temp->next;
    }
    printf("\n");
}

Node* addPolynomials(Node* poly1, Node* poly2) {
    Node* result = NULL;
    while (poly1 != NULL && poly2 != NULL) {
        if (poly1->pow > poly2->pow) {
            addNode(&result, poly1->coeff, poly1->pow);
            poly1 = poly1->next;
        } else if (poly1->pow < poly2->pow) {
            addNode(&result, poly2->coeff, poly2->pow);
            poly2 = poly2->next;
        } else {
```

```
            addNode(&result, poly1->coeff + poly2->coeff, poly1->pow);
            poly1 = poly1->next;
            poly2 = poly2->next;
        }
    }

    while (poly1 != NULL) {
        addNode(&result, poly1->coeff, poly1->pow);
        poly1 = poly1->next;
    }

    while (poly2 != NULL) {
        addNode(&result, poly2->coeff, poly2->pow);
        poly2 = poly2->next;
    }

    return result;
}

int main() {
    Node* poly1 = NULL;
    Node* poly2 = NULL;

    addNode(&poly1, 5, 2);
    addNode(&poly1, 4, 1);
    addNode(&poly1, 2, 0);
    addNode(&poly2, 5, 1);
    addNode(&poly2, 5, 0);

    Node* result = addPolynomials(poly1, poly2);

    printf("First Polynomial: ");
    printPolynomial(poly1);
    printf("Second Polynomial: ");
    printPolynomial(poly2);
    printf("Resultant Polynomial: ");
    printPolynomial(result);

    return 0;
}
```

**TOWER OF HANOI:**

```
#include <stdio.h>

void towerOfHanoi(int n, char begin, char end, char mid) {
    if (n == 1) {
        printf("Move disk 1 from rod %c to rod %c\n", begin, end);
        return;
    }
    towerOfHanoi(n - 1, begin, mid, end);
    printf("Move disk %d from rod %c to rod %c\n", n, begin, end);
    towerOfHanoi(n - 1, mid, end, begin);
}

int main() {
    int n;
    printf("Enter the number of disks: ");
    scanf("%d", &n);
    towerOfHanoi(n, 'A', 'C', 'B');
    return 0;
}
```

**ARRAY IMPLEMENTATION OF QUEUES:**

```
#include <stdio.h>
#include <stdlib.h>
#define max 20

int Queue[max];
int front = -1, rear = -1;

void enqueue();
void dequeue();
void display();

int main() {
   int ch;
   while (1) {
      printf("\n1. Enqueue");
      printf("\n2. Dequeue");
      printf("\n3. Display");
      printf("\n4. Exit");
      printf("\nEnter your choice: ");
      scanf("%d", &ch);
      switch (ch) {
         case 1: enqueue(); break;
         case 2: dequeue(); break;
         case 3: display(); break;
         case 4: exit(0);
         default: printf("Wrong choice\n");
      }
   }
   return 0;
}

void enqueue() {
   int data;
   if (rear == max - 1) {
      printf("Queue is full\n");
   } else {
      if (front == -1) front = 0;  // First element to be inserted
      printf("Enter the data: ");
      scanf("%d", &data);
      rear = rear + 1;
      Queue[rear] = data;
   }
}

void dequeue() {
   int data;
   if (front == -1) {
      printf("Queue is empty\n");
   } else {
      data = Queue[front];
      printf("Deleted element: %d\n", data);
      if (front == rear) {
         front = rear = -1;  // Queue is now empty
      } else {
         front = front + 1;
```

```
        }
    }
}

void display() {
    if (front == -1) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements are: ");
    for (int i = front; i <= rear; i++) {
        printf("%d ", Queue[i]);
    }
    printf("\n");
}
```

**ARRAY IMPLEMENTATION OF STACKS**

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

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

struct Node* front = NULL;
struct Node* rear = NULL;

void enqueue();
void dequeue();
void display();

int main() {
    int ch;
    while (1) {
        printf("\n1. Enqueue");
        printf("\n2. Dequeue");
        printf("\n3. Display");
        printf("\n4. Exit");
        printf("\nEnter your choice: ");
        scanf("%d", &ch);
        switch (ch) {
            case 1: enqueue(); break;
            case 2: dequeue(); break;
            case 3: display(); break;
            case 4: exit(0);
            default: printf("Wrong choice\n");
        }
    }
    return 0;
}

void enqueue() {
    int data;
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Memory allocation failed\n");
        return;
    }
    printf("Enter the data: ");
    scanf("%d", &data);
    newNode->data = data;
    newNode->next = NULL;
    if (rear == NULL) {
        front = rear = newNode;
    } else {
        rear->next = newNode;
        rear = newNode;
    }
}
```

```
void dequeue() {
    if (front == NULL) {
        printf("Queue is empty\n");
        return;
    }
    struct Node* temp = front;
    printf("Deleted element: %d\n", front->data);
    front = front->next;
    if (front == NULL) {
        rear = NULL;
    }
    free(temp);
}

void display() {
    if (front == NULL) {
        printf("Queue is empty\n");
        return;
    }
    struct Node* temp = front;
    printf("Queue elements are: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
```

**LINKED LIST IMPLEMENTATION OF STACKS**

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

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

struct Node *top = NULL;

void push();
void pop();
void display();

int main() {
    int ch;
    while (1) {
        printf("\n1. PUSH");
        printf("\n2. POP");
        printf("\n3. DISPLAY");
        printf("\n4. EXIT");
        printf("\nEnter choice: ");
        scanf("%d", &ch);
        switch (ch) {
            case 1: push(); break;
            case 2: pop(); break;
            case 3: display(); break;
            case 4: exit(0);
            default: printf("Wrong choice\n");
        }
    }
    return 0;
}

void push() {
    struct Node *ptr;
    ptr = (struct Node*)malloc(sizeof(struct Node));
    if (ptr == NULL) {
        printf("Memory allocation failed\n");
        return;
    }
    printf("Enter the info part of ptr: ");
    scanf("%d", &ptr->info);
    ptr->next = top;
    top = ptr;
    printf("Element pushed: %d\n", ptr->info);
}

void pop() {
    struct Node *ptr;
    if (top == NULL) {
        printf("Underflow: Stack is empty\n");
    } else {
        ptr = top;
```

```
        top = top->next;
        printf("Popped element: %d\n", ptr->info);
        free(ptr);
    }
}

void display() {
    struct Node *ptr = top;
    if (ptr == NULL) {
        printf("Stack is empty\n");
    } else {
        printf("Stack elements are:\n");
        while (ptr != NULL) {
            printf("%d\n", ptr->info);
            ptr = ptr->next;
        }
    }
}
```

**SPARSE MATRIX**

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

#define MAX 100
typedef struct {
    int row;
    int col;
    int value;
} Element;
typedef struct {
    int rows;
    int cols;
    int num; // Number of non-zero elements
    Element data[MAX]; // Array to store non-zero elements
} SparseMatrix;

void createSparseMatrix(SparseMatrix* sm, int r, int c, int matrix[r][c]) {
    sm->rows = r;
    sm->cols = c;
    sm->num = 0;
    for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++) {
            if (matrix[i][j] != 0) {
                sm->data[sm->num].row = i;
                sm->data[sm->num].col = j;
                sm->data[sm->num].value = matrix[i][j];
                sm->num++;
            }
        }
    }
}

void printSparseMatrix(SparseMatrix sm) {
    if (sm.num == 0) {
        printf("The matrix is empty (all zeros).\n");
        return;
    }
    printf("Sparse Matrix Representation:\n");
    printf("Row\tCol\tValue\n");
    for (int i = 0; i < sm .num; i++) {
        printf("%d\t%d\t%d\n", sm.data[i].row, sm.data[i].col, sm.data[i].value);
    }
}
int main() {
    int matrix[4][5] = {
        {0, 0, 3, 0, 4},
        {0, 0, 5, 7, 0},
        {0, 0, 0, 0, 0},
        {0, 2, 6, 0, 0}
    };
    SparseMatrix sm;
    createSparseMatrix(&sm, 4, 5, matrix);
    printSparseMatrix(sm);
    return 0;
}
```

**IMPLEMENTATION OF CIRCULAR LINKED LIST**

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

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

struct Node* head = NULL;

void insertAtHead(int data);
void insertAtTail(int data);
void insertAtIndex(int data, int index);
void deleteAtIndex(int index);
void display();

int main() {
    int choice, data, index;

    while (1) {
        printf("\nMenu:\n");
        printf("1. Insert at head\n");
        printf("2. Insert at tail\n");
        printf("3. Insert at index\n");
        printf("4. Delete at index\n");
        printf("5. Display\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to insert at head: ");
                scanf("%d", &data);
                insertAtHead(data);
                break;
            case 2:
                printf("Enter data to insert at tail: ");
                scanf("%d", &data);
                insertAtTail(data);
                break;
            case 3:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                printf("Enter index to insert at: ");
                scanf("%d", &index);
                insertAtIndex(data, index);
                break;
            case 4:
                printf("Enter index to delete: ");
                scanf("%d", &index);
                deleteAtIndex(index);
                break;
            case 5:
                display();
                break;
```

```
            case 6:
                exit(0);
            default:
                printf("Wrong choice. Please try again.\n");
        }
    }
    return 0;
}

void insertAtHead(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;

    if (head == NULL) {
        newNode->next = newNode; // Point to itself (circular)
        head = newNode;
        return;
    }

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

    newNode->next = head;
    temp->next = newNode;
    head = newNode;
}

void insertAtTail(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;

    if (head == NULL) {
        newNode->next = newNode; // Point to itself (circular)
        head = newNode;
        return;
    }

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

    temp->next = newNode;
    newNode->next = head;
}

void insertAtIndex(int data, int index) {
    if (index < 0) {
        printf("Invalid index\n");
        return;
    }

    if (index == 0) {
        insertAtHead(data);
        return;
    }
```

```
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;

    struct Node* temp = head;
    for (int i = 0; temp != NULL && i < index - 1; i++) {
        temp = temp->next;
        if (temp == head) break;  // Looping back to head (circular list)
    }

    if (temp == NULL || temp->next == head) {
        printf("Index out of bounds\n");
        free(newNode);
        return;
    }

    newNode->next = temp->next;
    temp->next = newNode;
}

void deleteAtIndex(int index) {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }

    if (index < 0) {
        printf("Invalid index\n");
        return;
    }

    struct Node* temp = head;
    struct Node* prev = NULL;

    if (index == 0) {
        // Deleting head
        while (temp->next != head) {
            temp = temp->next;
        }
        if (temp == head) {
            free(head);
            head = NULL;
            return;
        }
        prev = temp;
        temp = head;
        prev->next = temp->next;
        head = temp->next;
        free(temp);
        return;
    }

    for (int i = 0; temp != NULL && i < index; i++) {
        prev = temp;
        temp = temp->next;
        if (temp == head) {
            break; // Loop back to head
        }
```

```
    }

    if (temp == NULL || temp == head) {
        printf("Index out of bounds\n");
        return;
    }

    prev->next = temp->next;
    free(temp);
}

void display() {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }

    struct Node* temp = head;
    do {
        printf("%d -> ", temp->data);
        temp = temp->next;
    } while (temp != head);
    printf("(back to head)\n");
}
```

## MERGE TWO SORTED LINKED LIST :

```
struct Node* mergeLists(struct Node* list1, struct Node* list2) {

    struct Node* dummy = (struct Node*)malloc(sizeof(struct Node));
    struct Node* tail = dummy;

    while (list1 != NULL && list2 != NULL) {
        if (list1->data <= list2->data) {
            tail->next = list1;
            list1 = list1->next;
        } else {
            tail->next = list2;
            list2 = list2->next;
        }
        tail = tail->next;
    }

    if (list1 != NULL) tail->next = list1;
    if (list2 != NULL) tail->next = list2;

    struct Node* mergedHead = dummy->next;
    free(dummy);
    return mergedHead;
}
```

# Linked List Implementation of Queue:

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

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

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

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;
}


void enqueue(struct Queue* q, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;

    if (isEmpty(q)) {
        q->front = q->rear = newNode;
    } else {
        q->rear->next = newNode;
        q->rear = newNode;
    }
    printf("Enqueued value: %d\n", value);
}

int dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty. Cannot dequeue.\n");
        return -1;
    }

    int value = q->front->data;
    struct Node* temp = q->front;
    q->front = q->front->next;

    free(temp);
    if (q->front == NULL) {
        q->rear = NULL;
    }
    return value;
}

int peek(struct Queue* q) {
```

```
    if (isEmpty(q)) {
        printf("Queue is empty. No front element.\n");
        return -1;
    }
    return q->front->data;
}


void displayQueue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }

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

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

    int choice, value;

    while (1) {
        printf("\nQueue Operations:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Peek\n");
        printf("4. Display Queue\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to enqueue: ");
                scanf("%d", &value);
                enqueue(&q, value);
                break;
            case 2:
                value = dequeue(&q);
                if (value != -1) {
                    printf("Dequeued value: %d\n", value);
                }
                break;
            case 3:
                value = peek(&q);
                if (value != -1) {
                    printf("Front value: %d\n", value);
                }
                break;
            case 4:
```

```
            displayQueue(&q);
            break;
        case 5:
            exit(0);
        default:
            printf("Invalid choice. Please try again.\n");
      }
   }
  return 0;
}
```

# Circular Queue Implementation using Array:

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

#define MAX_SIZE 100

struct Queue {
    int arr[MAX_SIZE];
    int front;
    int rear;
};

void initializeQueue(struct Queue* q) {
    q->front = q->rear = -1;
}

int isEmpty(struct Queue* q) {
    return q->front == -1;
}

int isFull(struct Queue* q) {
    return (q->rear + 1) % MAX_SIZE == q->front;
}

void enqueue(struct Queue* q, int value) {
    if (isFull(q)) {
        printf("Queue is full\n");
        return;
    }

    if (isEmpty(q)) {
        q->front = q->rear = 0;
    } else {
        q->rear = (q->rear + 1) % MAX_SIZE;
    }

    q->arr[q->rear] = value;
    printf("Enqueued value: %d\n", value);
}

int dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return -1;
    }

    int value = q->arr[q->front];

    if (q->front == q->rear) {
        q->front = q->rear = -1;
    } else {
        q->front = (q->front + 1) % MAX_SIZE;
    }

    return value;
}
```

```
int peek(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return -1;
    }

    return q->arr[q->front];
}

void displayQueue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return;
    }

    printf("Queue elements: ");
    int i = q->front;
    do {
        printf("%d ", q->arr[i]);
        i = (i + 1) % MAX_SIZE;
    } while (i != (q->rear + 1) % MAX_SIZE);
    printf("\n");
}

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

    int choice, value;

    while (1) {
        printf("\nQueue Operations:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Peek\n");
        printf("4. Display Queue\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to enqueue: ");
                scanf("%d", &value);
                enqueue(&q, value);
                break;
            case 2:
                value = dequeue(&q);
                if (value != -1) {
                    printf("Dequeued value: %d\n", value);
                }
                break;
            case 3:
                value = peek(&q);
                if (value != -1) {
                    printf("Front value: %d\n", value);
                }
                break;
```

```
            case 4:
                displayQueue(&q);
                break;
            case 5:
                exit(0);
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

# Circular Queue Implementation using Linked List:

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

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

struct Queue {
    struct Node* rear;
};

void initializeQueue(struct Queue* q) {
    q->rear = NULL;
}

int isEmpty(struct Queue* q) {
    return q->rear == NULL;
}

void enqueue(struct Queue* q, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;

    if (isEmpty(q)) {
        newNode->next = newNode;
        q->rear = newNode;
    } else {
        newNode->next = q->rear->next;
        q->rear->next = newNode;
        q->rear = newNode;
    }

    printf("Enqueued value: %d\n", value);
}

int dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return -1;
    }

    struct Node* temp = q->rear->next;
    int value = temp->data;

    if (q->rear == q->rear->next) { // Only one node
        q->rear = NULL;
    } else {
        q->rear->next = temp->next;
    }

    free(temp);
    return value;
}

int peek(struct Queue* q) {
```

```
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return -1;
    }

    return q->rear->next->data;
}

void displayQueue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return;
    }

    struct Node* temp = q->rear->next;
    printf("Queue elements: ");
    do {
        printf("%d ", temp->data);
        temp = temp->next;
    } while (temp != q->rear->next);
    printf("\n");
}

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

    int choice, value;

    while (1) {
        printf("\nQueue Operations:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Peek\n");
        printf("4. Display Queue\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to enqueue: ");
                scanf("%d", &value);
                enqueue(&q, value);
                break;
            case 2:
                value = dequeue(&q);
                if (value != -1) {
                    printf("Dequeued value: %d\n", value);
                }
                break;
            case 3:
                value = peek(&q);
                if (value != -1) {
                    printf("Front value: %d\n", value);
                }
                break;
            case 4:
```

```
            displayQueue(&q);
            break;
        case 5:
            exit(0);
        default:
            printf("Invalid choice. Please try again.\n");
      }
   }

   return 0;
}
```