Data Structures Practical List (2024-25)

1. Write a program to implement Selection Sort algorithm.

```
#include <stdio.h>

void selectionSort(int a[], int n) {
    int i, j, min, temp;
    for(i = 0; i < n - 1; i++) {
        min = i;
        for(j = i + 1; j < n; j++) {
            if(a[j] < a[min])
                min = j;
        }
        temp = a[i];
        a[i] = a[min];
        a[min] = temp;
    }
}

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

int main() {
    int n;
    printf("Enter size: ");
```

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

    int a[n];
    printf("Enter %d numbers:\n", n);
    for(int i = 0; i < n; i++)
        scanf("%d", &a[i]);

    printf("Before sorting: ");
    printArray(a, n);

    selectionSort(a, n);

    printf("After sorting: ");
    printArray(a, n);

    return 0;
}
```

2. Write a program to implement Bubble sort algorithm.

```
#include <stdio.h>

void bubbleSort(int a[], int n) {
    int i, j, temp;
    for(i = 0; i < n - 1; i++) {
        for(j = 0; j < n - i - 1; j++) {
            if(a[j] > a[j + 1]) {
                temp = a[j];
                a[j] = a[j + 1];
```

```
                a[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 n;
    printf("Enter size: ");
    scanf("%d", &n);

    int a[n];
    printf("Enter %d numbers:\n", n);
    for(int i = 0; i < n; i++)
        scanf("%d", &a[i]);

    printf("Before sorting: ");
    printArray(a, n);

    bubbleSort(a, n);
```

```
    printf("After sorting: ");
    printArray(a, n);

    return 0;
}
```

3. Write a program to implement Insertion sort.

```
#include <stdio.h>

void insertionSort(int a[], int n) {
    int i, key, j;
    for(i = 1; i < n; i++) {
        key = a[i];
        j = i - 1;
        while(j >= 0 && a[j] > key) {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = key;
    }
}

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

```
int main() {
    int n;
    printf("Enter size: ");
    scanf("%d", &n);

    int a[n];
    printf("Enter %d numbers:\n", n);
    for(int i = 0; i < n; i++)
        scanf("%d", &a[i]);

    printf("Before sorting: ");
    printArray(a, n);

    insertionSort(a, n);

    printf("After sorting: ");
    printArray(a, n);

    return 0;
}
```

4. Write a program to implement Merge sort.

```
#include <stdio.h>

void merge(int a[], int left, int mid, int right) {
    int i = left, j = mid + 1, k = 0;
    int temp[right - left + 1];
```

```
    while(i <= mid && j <= right) {
        if(a[i] < a[j])
            temp[k++] = a[i++];
        else
            temp[k++] = a[j++];
    }

    while(i <= mid)
        temp[k++] = a[i++];
    while(j <= right)
        temp[k++] = a[j++];

    for(i = left, k = 0; i <= right; i++, k++)
        a[i] = temp[k];
}

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

void printArray(int a[], int n) {
```

```
    for(int i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
}

int main() {
    int n;
    printf("Enter size: ");
    scanf("%d", &n);

    int a[n];
    printf("Enter %d numbers:\n", n);
    for(int i = 0; i < n; i++)
        scanf("%d", &a[i]);

    printf("Before sorting: ");
    printArray(a, n);

    mergeSort(a, 0, n - 1);

    printf("After sorting: ");
    printArray(a, n);

    return 0;
}
```

5. Write a program to implement Quick sort

```
#include <stdio.h>

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

    for(int j = low; j < high; j++) {
        if(a[j] < pivot) {
            i++;
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }

    temp = a[i + 1];
    a[i + 1] = a[high];
    a[high] = temp;

    return i + 1;
}

void quickSort(int a[], int low, int high) {
    if(low < high) {
        int pos = partition(a, low, high);
        quickSort(a, low, pos - 1);
```

```
        quickSort(a, pos + 1, high);
    }
}

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

int main() {
    int n;
    printf("Enter size: ");
    scanf("%d", &n);

    int a[n];
    printf("Enter %d numbers:\n", n);
    for(int i = 0; i < n; i++)
        scanf("%d", &a[i]);

    printf("Before sorting: ");
    printArray(a, n);

    quickSort(a, 0, n - 1);

    printf("After sorting: ");
    printArray(a, n);
```

```
    return 0;
}
```

6. Write a program to implement Linear and Binary Search.

```
#include <stdio.h>

int linearSearch(int a[], int n, int key) {
    for(int i = 0; i < n; i++) {
        if(a[i] == key)
            return i;
    }
    return -1;
}

int binarySearch(int a[], int n, int key) {
    int low = 0, high = n - 1, mid;
    while(low <= high) {
        mid = (low + high) / 2;
        if(a[mid] == key)
            return mid;
        else if(a[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
```

```
}

void bubbleSort(int a[], int n) {
    for(int i = 0; i < n - 1; i++) {
        for(int j = 0; j < n - i - 1; j++) {
            if(a[j] > a[j + 1]) {
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
        }
    }
}

int main() {
    int n, key, choice;
    printf("Enter size: ");
    scanf("%d", &n);

    int a[n];
    printf("Enter %d numbers:\n", n);
    for(int i = 0; i < n; i++)
        scanf("%d", &a[i]);

    printf("Enter number to search: ");
    scanf("%d", &key);
```

```c
    printf("Choose search method:\n1. Linear Search\n2. Binary Search\nEnter choice: ");
    scanf("%d", &choice);

    if(choice == 1) {
        int index = linearSearch(a, n, key);
        if(index == -1)
            printf("Element not found.\n");
        else
            printf("Element found at position %d.\n", index);
    }
    else if(choice == 2) {
        bubbleSort(a, n); // Binary search needs sorted array
        int index = binarySearch(a, n, key);
        if(index == -1)
            printf("Element not found.\n");
        else
            printf("Element found at position %d.\n", index);
    }
    else {
        printf("Invalid choice.\n");
    }

    return 0;
}
```

7. Write a Program to implement following operations on Singly Linked List:
    i) Insertion
    ii) Deletion
    iii) Search a given value

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

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

struct Node* head = NULL;

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

void delete(int value) {
    struct Node *temp = head, *prev = NULL;

    while(temp != NULL && temp->data != value) {
        prev = temp;
        temp = temp->next;
    }

    if(temp == NULL) {
        printf("Value not found.\n");
        return;
    }

    if(prev == NULL)
        head = temp->next;
    else
        prev->next = temp->next;

    free(temp);
    printf("Value deleted.\n");
}
```

```
void search(int value) {
    struct Node* temp = head;
    int pos = 0;

    while(temp != NULL) {
        if(temp->data == value) {
            printf("Value found at position %d.\n", pos);
            return;
        }
        temp = temp->next;
        pos++;
    }
    printf("Value not found.\n");
}

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

int main() {
    int choice, value;

    while(1) {
        printf("\n1.Insert\n2.Delete\n3.Search\n4.Display\n5.Exit\nEnter
choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                insert(value);
                break;
            case 2:
                printf("Enter value to delete: ");
                scanf("%d", &value);
```

```
                delete(value);
                break;
            case 3:
                printf("Enter value to search: ");
                scanf("%d", &value);
                search(value);
                break;
            case 4:
                display();
                break;
            case 5:
                exit(0);
        }
    }

    return 0;
}
```

8. Write a Program to implement Parenthesis Checker using Stack.

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

#define MAX 100

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

void initStack(struct Stack* stack) {
    stack->top = -1;
}

int isFull(struct Stack* stack) {
    return stack->top == MAX - 1;
}

int isEmpty(struct Stack* stack) {
    return stack->top == -1;
}
```

```
void push(struct Stack* stack, char c) {
    if(isFull(stack)) {
        printf("Stack Overflow\n");
        return;
    }
    stack->arr[++stack->top] = c;
}

char pop(struct Stack* stack) {
    if(isEmpty(stack)) {
        printf("Stack Underflow\n");
        return -1;
    }
    return stack->arr[stack->top--];
}

int isMatchingPair(char opening, char closing) {
    if(opening == '(' && closing == ')')
        return 1;
    if(opening == '{' && closing == '}')
        return 1;
    if(opening == '[' && closing == ']')
        return 1;
    return 0;
}

int checkParentheses(char* expr) {
    struct Stack stack;
    initStack(&stack);

    for(int i = 0; expr[i]; i++) {
        char current = expr[i];

        if(current == '(' || current == '{' || current == '[') {
            push(&stack, current);
        }
        else if(current == ')' || current == '}' || current == ']') {
            if(isEmpty(&stack)) {
                return 0;
            }
            char top = pop(&stack);
            if(!isMatchingPair(top, current)) {
                return 0;
```

```
            }
        }
    }

    return isEmpty(&stack);
}

int main() {
    char expr[MAX];
    printf("Enter an expression: ");
    scanf("%s", expr);

    if(checkParentheses(expr))
        printf("Parentheses are balanced.\n");
    else
        printf("Parentheses are not balanced.\n");

    return 0;
}
```

9. Write a program to convert Infix expression to Postfix expression.

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

#define MAX 100

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

void initStack(struct Stack* stack) {
    stack->top = -1;
}

int isFull(struct Stack* stack) {
    return stack->top == MAX - 1;
}

int isEmpty(struct Stack* stack) {
```

```
    return stack->top == -1;
}

void push(struct Stack* stack, char c) {
    if(isFull(stack)) {
        printf("Stack Overflow\n");
        return;
    }
    stack->arr[++stack->top] = c;
}

char pop(struct Stack* stack) {
    if(isEmpty(stack)) {
        return -1;
    }
    return stack->arr[stack->top--];
}

int precedence(char c) {
    if(c == '+' || c == '-')
        return 1;
    if(c == '*' || c == '/')
        return 2;
    if(c == '^')
        return 3;
    return 0;
}

int isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/' || c == '^');
}

void infixToPostfix(char* infix, char* postfix) {
    struct Stack stack;
    initStack(&stack);
    int k = 0;

    for(int i = 0; infix[i]; i++) {
        char current = infix[i];

        if(isalpha(current)) {
            postfix[k++] = current; // Add operand to result
        }
```

```
        else if(current == '(') {
            push(&stack, current);
        }
        else if(current == ')') {
            while(!isEmpty(&stack) && stack.arr[stack.top] != '(') {
                postfix[k++] = pop(&stack);
            }
            pop(&stack); // Pop '('
        }
        else if(isOperator(current)) {
            while(!isEmpty(&stack) && precedence(stack.arr[stack.top]) >=
precedence(current)) {
                postfix[k++] = pop(&stack);
            }
            push(&stack, current);
        }
    }

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

    postfix[k] = '\0';
}

int main() {
    char infix[MAX], postfix[MAX];
    printf("Enter infix expression: ");
    scanf("%s", infix);

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

    return 0;
}
```

10.Write a program to implement Circular Queue using array.

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

#define MAX 5

struct CircularQueue {
    int arr[MAX];
    int front, rear;
};

void initQueue(struct CircularQueue* queue) {
    queue->front = queue->rear = -1;
}

int isFull(struct CircularQueue* queue) {
    return (queue->front == (queue->rear + 1) % MAX);
}

int isEmpty(struct CircularQueue* queue) {
    return (queue->front == -1);
}

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

    if(queue->front == -1) {
        queue->front = 0;
    }

    queue->rear = (queue->rear + 1) % MAX;
    queue->arr[queue->rear] = value;
    printf("Enqueued %d\n", value);
}

int dequeue(struct CircularQueue* queue) {
    if(isEmpty(queue)) {
        printf("Queue is empty.\n");
        return -1;
```

```c
    }

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

    if(queue->front == queue->rear) {
        queue->front = queue->rear = -1; // Queue is now empty
    } else {
        queue->front = (queue->front + 1) % MAX;
    }

    return value;
}

void display(struct CircularQueue* queue) {
    if(isEmpty(queue)) {
        printf("Queue is empty.\n");
        return;
    }

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

int main() {
    struct CircularQueue queue;
    initQueue(&queue);

    int choice, value;

    while(1) {
        printf("\n1.Enqueue\n2.Dequeue\n3.Display\n4.Exit\nEnter choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                printf("Enter value to enqueue: ");
                scanf("%d", &value);
                enqueue(&queue, value);
```

```
            break;
        case 2:
            value = dequeue(&queue);
            if(value != -1) {
                printf("Dequeued %d\n", value);
            }
            break;
        case 3:
            display(&queue);
            break;
        case 4:
            exit(0);
        default:
            printf("Invalid choice.\n");
        }
    }

    return 0;
}
```

11.Write a Program for Inorder, Preorder, Postorder and Level order traversal techniques.

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

// Structure for a node in the binary tree
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Inorder Traversal (Left, Root, Right)
```

```
void inorder(struct Node* root) {
    if(root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

// Preorder Traversal (Root, Left, Right)
void preorder(struct Node* root) {
    if(root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

// Postorder Traversal (Left, Right, Root)
void postorder(struct Node* root) {
    if(root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}

// Level Order Traversal (Breadth First Search)
void levelOrder(struct Node* root) {
    if(root == NULL)
        return;

    struct Node* queue[100];
    int front = 0, rear = 0;
    queue[rear++] = root;

    while(front < rear) {
        struct Node* current = queue[front++];
        printf("%d ", current->data);

        if(current->left != NULL) {
            queue[rear++] = current->left;
        }
        if(current->right != NULL) {
```

```
            queue[rear++] = current->right;
        }
    }
}

// Function to insert a new node in the binary tree (level-wise)
struct Node* insertNode(struct Node* root, int data) {
    struct Node* newNodePointer = newNode(data);
    if (root == NULL) {
        return newNodePointer;
    }

    // Simple level order insert, user needs to enter nodes for each level
    struct Node* queue[100];
    int front = 0, rear = 0;
    queue[rear++] = root;

    while (front < rear) {
        struct Node* current = queue[front++];

        if (current->left == NULL) {
            current->left = newNodePointer;
            break;
        } else {
            queue[rear++] = current->left;
        }

        if (current->right == NULL) {
            current->right = newNodePointer;
            break;
        } else {
            queue[rear++] = current->right;
        }
    }

    return root;
}

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

    printf("Enter the number of nodes to insert in the binary tree: ");
```

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

    for (int i = 0; i < n; i++) {
        printf("Enter value for node %d: ", i + 1);
        scanf("%d", &data);

        root = insertNode(root, data);
    }

    printf("\nInorder Traversal: ");
    inorder(root);
    printf("\n");

    printf("Preorder Traversal: ");
    preorder(root);
    printf("\n");

    printf("Postorder Traversal: ");
    postorder(root);
    printf("\n");

    printf("Level Order Traversal: ");
    levelOrder(root);
    printf("\n");

    return 0;
}
```

BINARY SEARCH TREE:

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

// Structure for a node in the Binary Search Tree
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* newNode(int data) {
```

```
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Function to insert a node in the BST
struct Node* insert(struct Node* root, int data) {
    if (root == NULL) {
        return newNode(data);
    }

    if (data < root->data) {
        root->left = insert(root->left, data);
    } else {
        root->right = insert(root->right, data);
    }

    return root;
}

// Inorder Traversal (Left, Root, Right)
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

// Preorder Traversal (Root, Left, Right)
void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

// Postorder Traversal (Left, Right, Root)
void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
```

```
        postorder(root->right);
        printf("%d ", root->data);
    }
}

// Level Order Traversal (Breadth First Search)
void levelOrder(struct Node* root) {
    if (root == NULL)
        return;

    struct Node* queue[100];
    int front = 0, rear = 0;
    queue[rear++] = root;

    while (front < rear) {
        struct Node* current = queue[front++];

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

        if (current->left != NULL) {
            queue[rear++] = current->left;
        }
        if (current->right != NULL) {
            queue[rear++] = current->right;
        }
    }
}

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

    // Accept user input for the number of nodes to insert in the BST
    printf("Enter the number of nodes to insert in the Binary Search Tree: ");
    scanf("%d", &n);

    // Insert nodes based on user input
    for (int i = 0; i < n; i++) {
        printf("Enter value for node %d: ", i + 1);
        scanf("%d", &data);
        root = insert(root, data);
    }
```

```
    // Perform and display the different traversals
    printf("\nInorder Traversal: ");
    inorder(root);
    printf("\n");

    printf("Preorder Traversal: ");
    preorder(root);
    printf("\n");

    printf("Postorder Traversal: ");
    postorder(root);
    printf("\n");

    printf("Level Order Traversal: ");
    levelOrder(root);
    printf("\n");

    return 0;
}
```

12.Write a program to implement Linear and Quadratic Probing

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

#define TABLE_SIZE 10

// Function to initialize the hash table
void initializeTable(int table[], int size) {
    for(int i = 0; i < size; i++) {
        table[i] = -1; // -1 indicates an empty slot
    }
}

// Hash function to map a key to an index
int hashFunction(int key) {
    return key % TABLE_SIZE;
}

// Linear Probing for collision resolution
void linearProbing(int table[], int key) {
    int index = hashFunction(key);
```

```
    // If the slot is already filled, search for the next available slot
    while (table[index] != -1) {
        index = (index + 1) % TABLE_SIZE;
    }

    table[index] = key; // Insert the key
}

// Quadratic Probing for collision resolution
void quadraticProbing(int table[], int key) {
    int index = hashFunction(key);
    int i = 1;

    // If the slot is already filled, search for the next available slot
    while (table[index] != -1) {
        index = (index + i * i) % TABLE_SIZE; // Quadratic probing
        i++;
    }

    table[index] = key; // Insert the key
}

// Function to display the hash table
void displayTable(int table[], int size) {
    for(int i = 0; i < size; i++) {
        printf("Index %d: ", i);
        if(table[i] == -1) {
            printf("Empty\n");
        } else {
            printf("%d\n", table[i]);
        }
    }
}

int main() {
    int table[TABLE_SIZE];
    int choice, key;

    initializeTable(table, TABLE_SIZE);

    while (1) {
        printf("\n1. Insert using Linear Probing\n");
```

```
        printf("2. Insert using Quadratic Probing\n");
        printf("3. Display Hash Table\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                printf("Enter the key to insert: ");
                scanf("%d", &key);
                linearProbing(table, key);
                break;
            case 2:
                printf("Enter the key to insert: ");
                scanf("%d", &key);
                quadraticProbing(table, key);
                break;
            case 3:
                displayTable(table, TABLE_SIZE);
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}
```

13.Write a program to implement Linear Probing and Double Hashing

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

#define TABLE_SIZE 10

// Function to initialize the hash table
void initializeTable(int table[], int size) {
    for(int i = 0; i < size; i++) {
        table[i] = -1; // -1 indicates an empty slot
    }
```

```
}

// Hash function to map a key to an index
int hashFunction(int key) {
    return key % TABLE_SIZE;
}

// Second hash function for double hashing
int secondHashFunction(int key) {
    return 7 - (key % 7); // Example secondary hash function
}

// Linear Probing for collision resolution
void linearProbing(int table[], int key) {
    int index = hashFunction(key);

    // If the slot is already filled, search for the next available slot
    while (table[index] != -1) {
        index = (index + 1) % TABLE_SIZE;
    }

    table[index] = key; // Insert the key
}

// Double Hashing for collision resolution
void doubleHashing(int table[], int key) {
    int index = hashFunction(key);
    int step = secondHashFunction(key);

    // If the slot is already filled, apply double hashing
    while (table[index] != -1) {
        index = (index + step) % TABLE_SIZE; // Double hashing step
    }

    table[index] = key; // Insert the key
}

// Function to display the hash table
void displayTable(int table[], int size) {
    for(int i = 0; i < size; i++) {
        printf("Index %d: ", i);
        if(table[i] == -1) {
            printf("Empty\n");
```

```
        } else {
            printf("%d\n", table[i]);
        }
    }
}

int main() {
    int table[TABLE_SIZE];
    int choice, key;

    initializeTable(table, TABLE_SIZE);

    while (1) {
        printf("\n1. Insert using Linear Probing\n");
        printf("2. Insert using Double Hashing\n");
        printf("3. Display Hash Table\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                printf("Enter the key to insert: ");
                scanf("%d", &key);
                linearProbing(table, key);
                break;
            case 2:
                printf("Enter the key to insert: ");
                scanf("%d", &key);
                doubleHashing(table, key);
                break;
            case 3:
                displayTable(table, TABLE_SIZE);
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}
```

14. Find the Winner of the Circular Game using Queue.

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

#define MAX_SIZE 100

// Queue structure
typedef struct Queue {
    int data[MAX_SIZE];
    int front, rear;
} Queue;

// Function to initialize the queue
void initializeQueue(Queue* q) {
    q->front = 0;
    q->rear = -1;
}

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

// Function to enqueue an element
void enqueue(Queue* q, int value) {
    if (q->rear == MAX_SIZE - 1) {
        printf("Queue is full\n");
        return;
    }
    q->data[++(q->rear)] = value;
}

// Function to dequeue an element
int dequeue(Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return -1;
    }
    return q->data[q->front++];
}
```

```c
// Function to find the winner of the circular game with custom step k
int findWinner(int n, int k) {
    Queue q;
    initializeQueue(&q);

    // Step 1: Enqueue all people
    for (int i = 1; i <= n; i++) {
        enqueue(&q, i);
    }

    // Step 2: Eliminate every k-th person
    while (q.rear - q.front > 0) {
        for (int i = 1; i < k; i++) {
            // Move the first person to the end of the queue
            int person = dequeue(&q);
            enqueue(&q, person);
        }
        // The k-th person is eliminated
        dequeue(&q);
    }

    // The last remaining person is the winner
    return q.data[q.front];
}

int main() {
    int n, k;
    printf("Enter the number of people: ");
    scanf("%d", &n);
    printf("Enter the step (k): ");
    scanf("%d", &k);

    int winner = findWinner(n, k);
    printf("The winner is person number %d\n", winner);
    return 0;
}
```

15. Write a program to implement stack using array and perform various operations on it.

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

#define MAX_SIZE 100

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

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

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

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

// Function to push an element onto the stack
void push(Stack* stack, int value) {
   if (isFull(stack)) {
      printf("Stack is full! Cannot push %d.\n", value);
   } else {
      stack->arr[++(stack->top)] = value;
      printf("%d pushed to stack.\n", value);
   }
}

// Function to pop an element from the stack
int pop(Stack* stack) {
   if (isEmpty(stack)) {
      printf("Stack is empty! Cannot pop.\n");
```

```
        return -1;
    } else {
        return stack->arr[(stack->top)--];
    }
}

// Function to peek the top element of the stack
int peek(Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty! Cannot peek.\n");
        return -1;
    } else {
        return stack->arr[stack->top];
    }
}

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

int main() {
    Stack stack;
    int choice, value;

    initializeStack(&stack);

    while (1) {
        // Menu to perform operations
        printf("\nStack Operations Menu:\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Peek\n");
        printf("4. Display\n");
        printf("5. Exit\n");
```

```
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to push: ");
                scanf("%d", &value);
                push(&stack, value);
                break;
            case 2:
                value = pop(&stack);
                if (value != -1) {
                    printf("Popped value: %d\n", value);
                }
                break;
            case 3:
                value = peek(&stack);
                if (value != -1) {
                    printf("Top element is: %d\n", value);
                }
                break;
            case 4:
                display(&stack);
                break;
            case 5:
                printf("Exiting the program.\n");
                exit(0);
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}
```

16.Write a program to implement queue using array.

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

#define MAX_SIZE 100

int queue[MAX_SIZE];
```

```
int front = -1, rear = -1;

// Function to check if the queue is full
int isFull() {
    return rear == MAX_SIZE - 1;
}

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

// Function to enqueue an element
void enqueue(int value) {
    if (isFull()) {
        printf("Queue is full! Cannot enqueue %d.\n", value);
    } else {
        if (front == -1) {
            front = 0;  // First element being added
        }
        queue[++rear] = value;
        printf("%d enqueued to queue.\n", value);
    }
}

// Function to dequeue an element
int dequeue() {
    if (isEmpty()) {
        printf("Queue is empty! Cannot dequeue.\n");
        return -1;
    } else {
        int value = queue[front];
        if (front == rear) {
            front = rear = -1;  // Queue becomes empty
        } else {
            front++;
        }
        return value;
    }
}

// Function to peek the front element of the queue
int peek() {
```

```c
    if (isEmpty()) {
        printf("Queue is empty! Cannot peek.\n");
        return -1;
    } else {
        return queue[front];
    }
}

// Function to display the elements of the queue
void display() {
    if (isEmpty()) {
        printf("Queue is empty.\n");
    } else {
        printf("Queue elements: ");
        for (int i = front; i <= rear; i++) {
            printf("%d ", queue[i]);
        }
        printf("\n");
    }
}

int main() {
    int choice, value;

    while (1) {
        // Menu to perform operations
        printf("\nQueue Operations Menu:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Peek\n");
        printf("4. Display\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(value);
                break;
            case 2:
                value = dequeue();
```

```
            if (value != -1) {
                printf("Dequeued value: %d\n", value);
            }
            break;
        case 3:
            value = peek();
            if (value != -1) {
                printf("Front element is: %d\n", value);
            }
            break;
        case 4:
            display();
            break;
        case 5:
            printf("Exiting the program.\n");
            exit(0);
        default:
            printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}
```

17. Write a program to perform merging of two sorted Link Lists. (SLL)

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

// Define the node structure
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));
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}
```

```
// Function to insert a node at the end of the list
void insertNode(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

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

// Function to merge two sorted linked lists
struct Node* mergeSortedLists(struct Node* list1, struct Node* list2) {
    // Create a dummy node to simplify the merge process
    struct Node* dummy = createNode(0);
    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;
    }

    // Append the remaining nodes of either list
    if (list1 != NULL) {
```

```
        tail->next = list1;
    } else {
        tail->next = list2;
    }

    // The dummy node was just a placeholder, return the merged list starting
from dummy->next
    struct Node* mergedList = dummy->next;
    free(dummy); // Free the dummy node
    return mergedList;
}

int main() {
    struct Node* list1 = NULL;
    struct Node* list2 = NULL;

    // Inserting nodes into the first list
    insertNode(&list1, 1);
    insertNode(&list1, 3);
    insertNode(&list1, 5);
    insertNode(&list1, 7);

    // Inserting nodes into the second list
    insertNode(&list2, 2);
    insertNode(&list2, 4);
    insertNode(&list2, 6);
    insertNode(&list2, 8);

    // Printing the two sorted lists
    printf("List 1: ");
    printList(list1);

    printf("List 2: ");
    printList(list2);

    // Merging the two sorted lists
    struct Node* mergedList = mergeSortedLists(list1, list2);

    // Printing the merged sorted list
    printf("Merged List: ");
    printList(mergedList);

    return 0;
```

}

18. Write a program to implement Merge sort.

19. Perform merging of two sorted Link Lists.

20. Write a Program to implement following operations on Singly Linked List:
   i) Reverse the given link list
   ii) Deletion
   iii) Search a given value

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

// Define the node structure
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));
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node at the end of the list
void insertNode(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}
```

```
// Function to print the list
void printList(struct Node* head) {
    struct Node* temp = head;
    if (temp == NULL) {
        printf("List is empty.\n");
        return;
    }
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Function to reverse the linked list
void reverseList(struct Node** head) {
    struct Node* prev = NULL;
    struct Node* current = *head;
    struct Node* next = NULL;

    while (current != NULL) {
        next = current->next;  // Store the next node
        current->next = prev;  // Reverse the link
        prev = current;        // Move prev to current
        current = next;        // Move current to the next node
    }
    *head = prev;  // Update the head to the new first node
}

// Function to delete a node with a given value
void deleteNode(struct Node** head, int value) {
    struct Node* temp = *head;
    struct Node* prev = NULL;

    // If the node to be deleted is the head node
    if (temp != NULL && temp->data == value) {
        *head = temp->next;  // Move the head to the next node
        free(temp);          // Free the memory
        return;
    }

    // Search for the node to be deleted
```

```
    while (temp != NULL && temp->data != value) {
        prev = temp;
        temp = temp->next;
    }

    // If the value is not found
    if (temp == NULL) {
        printf("Value %d not found in the list.\n", value);
        return;
    }

    // Unlink the node from the linked list
    prev->next = temp->next;
    free(temp);  // Free the memory of the deleted node
    printf("Node with value %d deleted.\n", value);
}

// Function to search for a given value in the list
int searchValue(struct Node* head, int value) {
    struct Node* temp = head;
    while (temp != NULL) {
        if (temp->data == value) {
            return 1;  // Value found
        }
        temp = temp->next;
    }
    return 0;  // Value not found
}

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

    while (1) {
        printf("\nMenu:\n");
        printf("1. Insert a node\n");
        printf("2. Reverse the list\n");
        printf("3. Delete a node\n");
        printf("4. Search for a value\n");
        printf("5. Print the list\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
```

```
    switch (choice) {
        case 1:
            printf("Enter value to insert: ");
            scanf("%d", &value);
            insertNode(&head, value);
            break;

        case 2:
            reverseList(&head);
            printf("List reversed.\n");
            break;

        case 3:
            printf("Enter value to delete: ");
            scanf("%d", &value);
            deleteNode(&head, value);
            break;

        case 4:
            printf("Enter value to search: ");
            scanf("%d", &value);
            if (searchValue(head, value)) {
                printf("Value %d found in the list.\n", value);
            } else {
                printf("Value %d not found in the list.\n", value);
            }
            break;

        case 5:
            printf("The current list: ");
            printList(head);
            break;

        case 6:
            printf("Exiting program.\n");
            exit(0);

        default:
            printf("Invalid choice. Please try again.\n");
    }
}
```

```
        return 0;
    }
```

21. Write a Program to implement following operations on Singly Linked List:
(i) Sort the list
(ii) Deletion
(iii) Insertion

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

// Define the node structure
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));
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node at the end of the list
void insertNode(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

// Function to print the list
void printList(struct Node* head) {
    struct Node* temp = head;
```

```
    if (temp == NULL) {
        printf("List is empty.\n");
        return;
    }
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Function to delete a node with a given value
void deleteNode(struct Node** head, int value) {
    struct Node* temp = *head;
    struct Node* prev = NULL;

    // If the node to be deleted is the head node
    if (temp != NULL && temp->data == value) {
        *head = temp->next;  // Move the head to the next node
        free(temp);          // Free the memory
        return;
    }

    // Search for the node to be deleted
    while (temp != NULL && temp->data != value) {
        prev = temp;
        temp = temp->next;
    }

    // If the value is not found
    if (temp == NULL) {
        printf("Value %d not found in the list.\n", value);
        return;
    }

    // Unlink the node from the linked list
    prev->next = temp->next;
    free(temp);  // Free the memory of the deleted node
    printf("Node with value %d deleted.\n", value);
}

// Function to sort the list (using Bubble Sort)
void sortList(struct Node* head) {
```

```
    if (head == NULL) return;

    struct Node *i, *j;
    int temp;

    for (i = head; i != NULL; i = i->next) {
        for (j = i->next; j != NULL; j = j->next) {
            if (i->data > j->data) {
                temp = i->data;
                i->data = j->data;
                j->data = temp;
            }
        }
    }
}

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

    while (1) {
        printf("\nMenu:\n");
        printf("1. Insert a node\n");
        printf("2. Sort the list\n");
        printf("3. Delete a node\n");
        printf("4. Print the list\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                insertNode(&head, value);
                break;

            case 2:
                sortList(head);
                printf("List sorted.\n");
                break;

            case 3:
```

```
            printf("Enter value to delete: ");
            scanf("%d", &value);
            deleteNode(&head, value);
            break;

        case 4:
            printf("The current list: ");
            printList(head);
            break;

        case 5:
            printf("Exiting program.\n");
            exit(0);

        default:
            printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```