1. Write a Program to create a Binary Tree and perform following nonrecursive operations on it. a. Preorder Traversal, b. Postorder Traversal, c. Count total no. of nodes, d. Display height of a tree.

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

typedef struct Node {
    int data;
    struct Node *left, *right;
} Node;

// Stack for tree traversal
typedef struct Stack {
    Node* data[100];
    int top;
} Stack;

void push(Stack *s, Node *node) {
    s->data[++(s->top)] = node;
}

Node* pop(Stack *s) {
    if (s->top == -1) return NULL;
    return s->data[(s->top)--];
}

int isEmpty(Stack *s) {
    return s->top == -1;
}

// Queue for tree creation
typedef struct Queue {
    Node* data[100];
    int front, rear;
} Queue;

void enqueue(Queue *q, Node *node) {
    q->data[++(q->rear)] = node;
}

Node* dequeue(Queue *q) {
    if (q->front > q->rear) return NULL;
    return q->data[(q->front)++];
}

int isQueueEmpty(Queue *q) {
    return q->front > q->rear;
}
```

```c
// Create a new node
Node* createNode(int data) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Create tree level-wise
Node* createTree() {
    int data;
    printf("Enter root node data (-1 for NULL): ");
    scanf("%d", &data);
    if (data == -1) return NULL;

    Node* root = createNode(data);
    Queue q = {.front = 0, .rear = -1};
    enqueue(&q, root);

    while (!isQueueEmpty(&q)) {
        Node* curr = dequeue(&q);

        printf("Enter left child of %d (-1 for NULL): ", curr->data);
        scanf("%d", &data);
        if (data != -1) {
            curr->left = createNode(data);
            enqueue(&q, curr->left);
        }

        printf("Enter right child of %d (-1 for NULL): ", curr->data);
        scanf("%d", &data);
        if (data != -1) {
            curr->right = createNode(data);
            enqueue(&q, curr->right);
        }
    }
    return root;
}

// Non-recursive Preorder Traversal
void preorder(Node* root) {
    Stack s = {.top = -1};
    Node* curr = root;
    printf("Preorder: ");
    while (curr || !isEmpty(&s)) {
        if (curr) {
            printf("%d ", curr->data);
```

```
            push(&s, curr);
            curr = curr->left;
        } else {
            curr = pop(&s);
            curr = curr->right;
        }
    }
    printf("\n");
}

// Non-recursive Postorder Traversal
void postorder(Node* root) {
    Stack s1 = {.top = -1}, s2 = {.top = -1};
    if (!root) return;
    push(&s1, root);
    while (!isEmpty(&s1)) {
        Node* curr = pop(&s1);
        push(&s2, curr);
        if (curr->left) push(&s1, curr->left);
        if (curr->right) push(&s1, curr->right);
    }
    printf("Postorder: ");
    while (!isEmpty(&s2)) {
        printf("%d ", pop(&s2)->data);
    }
    printf("\n");
}

// Count total number of nodes (non-recursive)
int countNodes(Node* root) {
    if (!root) return 0;
    Stack s = {.top = -1};
    int count = 0;
    Node* curr = root;
    while (curr || !isEmpty(&s)) {
        if (curr) {
            count++;
            push(&s, curr);
            curr = curr->left;
        } else {
            curr = pop(&s);
            curr = curr->right;
        }
    }
    return count;
}

// Display height of tree (non-recursive, using level order)
```

```c
int treeHeight(Node* root) {
    if (!root) return 0;
    Queue q = {.front = 0, .rear = -1};
    enqueue(&q, root);
    int height = 0;
    while (1) {
        int nodeCount = q.rear - q.front + 1;
        if (nodeCount == 0)
            return height;
        height++;
        while (nodeCount > 0) {
            Node* node = dequeue(&q);
            if (node->left) enqueue(&q, node->left);
            if (node->right) enqueue(&q, node->right);
            nodeCount--;
        }
    }
}

int main() {
    Node* root = createTree();
    int choice;
    do {
        printf("\n1. Preorder Traversal\n2. Postorder Traversal\n3. Count
Nodes\n4. Display Height\n5. Exit\nEnter choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1: preorder(root); break;
            case 2: postorder(root); break;
            case 3: printf("Total nodes: %d\n", countNodes(root)); break;
            case 4: printf("Height of tree: %d\n", treeHeight(root));
break;
            case 5: break;
            default: printf("Invalid choice\n");
        }
    } while (choice != 5);
    return 0;
}
```

2. Write a Program to create a Binary Tree and perform following nonrecursive operations on it. a. inorder Traversal; b. Count no. of nodes on longest path; c. display tree levelwise; d. Display height of a tree.

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

#define MAX_QUEUE 100
```

```c
// Binary Tree Node
typedef struct Node {
    int data;
    struct Node *left, *right;
} Node;

// Stack for non-recursive inorder
typedef struct {
    Node* arr[MAX_QUEUE];
    int top;
} Stack;

// Queue for level order traversal
typedef struct {
    Node* arr[MAX_QUEUE];
    int front, rear;
} Queue;

// Stack functions
void initStack(Stack* s) { s->top = -1; }
int isEmptyStack(Stack* s) { return s->top == -1; }
void push(Stack* s, Node* node) { s->arr[++(s->top)] = node; }
Node* pop(Stack* s) { return s->arr[(s->top)--]; }

// Queue functions
void initQueue(Queue* q) { q->front = q->rear = 0; }
int isEmptyQueue(Queue* q) { return q->front == q->rear; }
void enqueue(Queue* q, Node* node) { q->arr[(q->rear)++] = node; }
Node* dequeue(Queue* q) { return q->arr[(q->front)++]; }

// Create new node
Node* createNode(int data) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Insert node (level order for simplicity)
Node* insertLevelOrder(Node* root, int data) {
    Node* newNode = createNode(data);
    if (!root) return newNode;
    Queue q;
    initQueue(&q);
    enqueue(&q, root);
    while (!isEmptyQueue(&q)) {
        Node* temp = dequeue(&q);
```

```
        if (!temp->left) {
            temp->left = newNode;
            break;
        } else enqueue(&q, temp->left);
        if (!temp->right) {
            temp->right = newNode;
            break;
        } else enqueue(&q, temp->right);
    }
    return root;
}

// a. Non-recursive Inorder Traversal
void inorderNonRecursive(Node* root) {
    Stack s;
    initStack(&s);
    Node* curr = root;
    printf("Inorder Traversal: ");
    while (curr || !isEmptyStack(&s)) {
        while (curr) {
            push(&s, curr);
            curr = curr->left;
        }
        curr = pop(&s);
        printf("%d ", curr->data);
        curr = curr->right;
    }
    printf("\n");
}

// b. Count nodes on longest path (height)
int longestPathNodes(Node* root) {
    if (!root) return 0;
    Queue q;
    initQueue(&q);
    enqueue(&q, root);
    int height = 0;
    while (!isEmptyQueue(&q)) {
        int nodeCount = q.rear - q.front;
        if (nodeCount == 0) break;
        height++;
        while (nodeCount--) {
            Node* node = dequeue(&q);
            if (node->left) enqueue(&q, node->left);
            if (node->right) enqueue(&q, node->right);
        }
    }
    return height;
```

```
}

// c. Display tree levelwise
void displayLevelwise(Node* root) {
    if (!root) return;
    Queue q;
    initQueue(&q);
    enqueue(&q, root);
    printf("Levelwise display:\n");
    while (!isEmptyQueue(&q)) {
        int nodeCount = q.rear - q.front;
        while (nodeCount--) {
            Node* node = dequeue(&q);
            printf("%d ", node->data);
            if (node->left) enqueue(&q, node->left);
            if (node->right) enqueue(&q, node->right);
        }
        printf("\n");
    }
}

// d. Display height of tree
void displayHeight(Node* root) {
    int h = longestPathNodes(root);
    printf("Height of tree: %d\n", h);
}

int main() {
    Node* root = NULL;
    int n, val, ch;
    printf("Enter number of nodes: ");
    scanf("%d", &n);
    printf("Enter node values:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &val);
        root = insertLevelOrder(root, val);
    }
    do {
        printf("\nMenu:\n1. Inorder Traversal\n2. Count nodes on longest
path\n3. Display tree levelwise\n4. Display height of tree\n5. Exit\nEnter
choice: ");
        scanf("%d", &ch);
        switch (ch) {
            case 1: inorderNonRecursive(root); break;
            case 2: printf("Nodes on longest path: %d\n",
longestPathNodes(root)); break;
            case 3: displayLevelwise(root); break;
            case 4: displayHeight(root); break;
```

```
        }
    } while (ch != 5);
    return 0;
}
```

3. Write a Program to create a Binary Search Tree holding numeric keys and perform following nonrecursive operations on it. a. Levelwise display, b. Mirror image, c. Display height of a tree.

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

// Definition of a node in BST
typedef struct Node {
    int key;
    struct Node *left, *right;
} Node;

// Queue for level order traversal
typedef struct QueueNode {
    Node *treeNode;
    struct QueueNode *next;
} QueueNode;

typedef struct {
    QueueNode *front, *rear;
} Queue;

// Queue functions
void initQueue(Queue *q) {
    q->front = q->rear = NULL;
}

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

void enqueue(Queue *q, Node *treeNode) {
    QueueNode *temp = (QueueNode *)malloc(sizeof(QueueNode));
    temp->treeNode = treeNode;
    temp->next = NULL;
    if (q->rear == NULL) {
        q->front = q->rear = temp;
    } else {
        q->rear->next = temp;
        q->rear = temp;
    }
```

```c
}

Node* dequeue(Queue *q) {
    if (isEmpty(q)) return NULL;
    QueueNode *temp = q->front;
    Node *treeNode = temp->treeNode;
    q->front = q->front->next;
    if (q->front == NULL) q->rear = NULL;
    free(temp);
    return treeNode;
}

// BST insertion (non-recursive)
Node* insert(Node *root, int key) {
    Node *newNode = (Node *)malloc(sizeof(Node));
    newNode->key = key;
    newNode->left = newNode->right = NULL;
    if (root == NULL) return newNode;

    Node *parent = NULL, *curr = root;
    while (curr != NULL) {
        parent = curr;
        if (key < curr->key)
            curr = curr->left;
        else
            curr = curr->right;
    }
    if (key < parent->key)
        parent->left = newNode;
    else
        parent->right = newNode;
    return root;
}

// a. Levelwise display (non-recursive)
void levelOrder(Node *root) {
    if (root == NULL) return;
    Queue q;
    initQueue(&q);
    enqueue(&q, root);
    while (!isEmpty(&q)) {
        Node *curr = dequeue(&q);
        printf("%d ", curr->key);
        if (curr->left) enqueue(&q, curr->left);
        if (curr->right) enqueue(&q, curr->right);
    }
    printf("\n");
}
```

```c
// b. Mirror image (non-recursive)
void mirror(Node *root) {
    if (root == NULL) return;
    Queue q;
    initQueue(&q);
    enqueue(&q, root);
    while (!isEmpty(&q)) {
        Node *curr = dequeue(&q);
        // Swap left and right
        Node *temp = curr->left;
        curr->left = curr->right;
        curr->right = temp;
        if (curr->left) enqueue(&q, curr->left);
        if (curr->right) enqueue(&q, curr->right);
    }
}

// c. Display height of a tree (non-recursive)
int treeHeight(Node *root) {
    if (root == NULL) return 0;
    Queue q;
    initQueue(&q);
    enqueue(&q, root);
    int height = 0;
    while (1) {
        int nodeCount = 0;
        QueueNode *temp = q.front;
        while (temp) {
            nodeCount++;
            temp = temp->next;
        }
        if (nodeCount == 0)
            return height;
        height++;
        while (nodeCount > 0) {
            Node *curr = dequeue(&q);
            if (curr->left) enqueue(&q, curr->left);
            if (curr->right) enqueue(&q, curr->right);
            nodeCount--;
        }
    }
}

// Main function to demonstrate operations
int main() {
    Node *root = NULL;
    int choice, key;
```

```
    while (1) {
        printf("\n1. Insert\n2. Levelwise Display\n3. Mirror Image\n4. Display
Height\n5. Exit\nEnter choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter key to insert: ");
                scanf("%d", &key);
                root = insert(root, key);
                break;
            case 2:
                printf("Levelwise display: ");
                levelOrder(root);
                break;
            case 3:
                mirror(root);
                printf("Mirror image created.\n");
                break;
            case 4:
                printf("Height of tree: %d\n", treeHeight(root));
                break;
            case 5:
                exit(0);
            default:
                printf("Invalid choice.\n");
        }
    }
    return 0;
}
```

4. Write a program to illustrate operations on a BST holding numeric keys. The menu must include: • Insert • Delete • Find • display in Inorder way

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

typedef struct Node {
    int key;
    struct Node *left, *right;
} Node;

// Function to create a new node
Node* newNode(int item) {
    Node* temp = (Node*)malloc(sizeof(Node));
```

```
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Insert a node
Node* insert(Node* node, int key) {
    if (node == NULL) return newNode(key);
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    return node;
}

// Find the minimum value node
Node* minValueNode(Node* node) {
    Node* current = node;
    while (current && current->left != NULL)
        current = current->left;
    return current;
}

// Delete a node
Node* deleteNode(Node* root, int key) {
    if (root == NULL) return root;
    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
    else {
        if (root->left == NULL) {
            Node* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            Node* temp = root->left;
            free(root);
            return temp;
        }
        Node* temp = minValueNode(root->right);
        root->key = temp->key;
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}
```

```c
// Find a node
Node* find(Node* root, int key) {
    if (root == NULL || root->key == key)
        return root;
    if (key < root->key)
        return find(root->left, key);
    return find(root->right, key);
}

// Inorder traversal
void inorder(Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

int main() {
    Node* root = NULL;
    int choice, key;
    while (1) {
        printf("\n1. Insert\n2. Delete\n3. Find\n4. Display Inorder\n5.
Exit\nEnter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter key to insert: ");
                scanf("%d", &key);
                root = insert(root, key);
                break;
            case 2:
                printf("Enter key to delete: ");
                scanf("%d", &key);
                root = deleteNode(root, key);
                break;
            case 3:
                printf("Enter key to find: ");
                scanf("%d", &key);
                if (find(root, key))
                    printf("Key %d found in BST.\n", key);
                else
                    printf("Key %d not found in BST.\n", key);
                break;
            case 4:
                printf("Inorder traversal: ");
                inorder(root);
                printf("\n");
```

```
                break;
            case 5:
                exit(0);
            default:
                printf("Invalid choice.\n");
        }
    }
    return 0;
}
```

5. Write a program to illustrate operations on a BST holding numeric keys. The menu must include: • Insert • Mirror Image • Find • Post order (nonrecursive)

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

typedef struct Node {
    int key;
    struct Node *left, *right;
} Node;

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

// Insert a key into BST
Node* insert(Node* root, int key) {
    if (root == NULL) return newNode(key);
    if (key < root->key)
        root->left = insert(root->left, key);
    else if (key > root->key)
        root->right = insert(root->right, key);
    // Duplicate keys are not inserted
    return root;
}

// Mirror the BST
void mirror(Node* root) {
    if (root == NULL) return;
    Node* temp = root->left;
    root->left = root->right;
    root->right = temp;
    mirror(root->left);
```

```
    mirror(root->right);
}

// Find a key in BST
Node* find(Node* root, int key) {
    while (root != NULL) {
        if (key == root->key)
            return root;
        else if (key < root->key)
            root = root->left;
        else
            root = root->right;
    }
    return NULL;
}

// Non-recursive postorder traversal
void postOrderNonRecursive(Node* root) {
    if (root == NULL) return;
    Node *stack1[100], *stack2[100];
    int top1 = -1, top2 = -1;
    stack1[++top1] = root;
    while (top1 >= 0) {
        Node* node = stack1[top1--];
        stack2[++top2] = node;
        if (node->left)
            stack1[++top1] = node->left;
        if (node->right)
            stack1[++top1] = node->right;
    }
    while (top2 >= 0) {
        printf("%d ", stack2[top2--]->key);
    }
    printf("\n");
}

int main() {
    Node* root = NULL;
    int choice, key;
    while (1) {
        printf("\nMenu:\n1. Insert\n2. Mirror Image\n3. Find\n4. Postorder
(Nonrecursive)\n5. Exit\nEnter choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter key to insert: ");
                scanf("%d", &key);
                root = insert(root, key);
```

```
                break;
            case 2:
                mirror(root);
                printf("BST mirrored.\n");
                break;
            case 3:
                printf("Enter key to find: ");
                scanf("%d", &key);
                if (find(root, key))
                    printf("Key %d found in BST.\n", key);
                else
                    printf("Key %d not found in BST.\n", key);
                break;
            case 4:
                printf("Postorder traversal: ");
                postOrderNonRecursive(root);
                break;
            case 5:
                exit(0);
            default:
                printf("Invalid choice.\n");
        }
    }
    return 0;
}
```

6. Write a Program to create a Binary Search Tree and perform following nonrecursive operations on it. a. Preorder Traversal b. Inorder Traversal c. Display Number of Leaf Nodes d. Mirror Image

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

typedef struct Node {
    int data;
    struct Node *left, *right;
} Node;

// Stack for traversal
typedef struct Stack {
    Node* data[100];
    int top;
} Stack;

void push(Stack *s, Node *node) {
    s->data[++(s->top)] = node;
```

```
}

Node* pop(Stack *s) {
    if (s->top == -1) return NULL;
    return s->data[(s->top)--];
}

int isEmpty(Stack *s) {
    return s->top == -1;
}

// BST Insert
Node* insert(Node* root, int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    if (!root) return newNode;
    Node *curr = root, *parent = NULL;
    while (curr) {
        parent = curr;
        if (data < curr->data)
            curr = curr->left;
        else
            curr = curr->right;
    }
    if (data < parent->data)
        parent->left = newNode;
    else
        parent->right = newNode;
    return root;
}

// Nonrecursive Inorder Traversal
void inorder(Node* root) {
    Stack s; s.top = -1;
    Node* curr = root;
    while (curr || !isEmpty(&s)) {
        while (curr) {
            push(&s, curr);
            curr = curr->left;
        }
        curr = pop(&s);
        printf("%d ", curr->data);
        curr = curr->right;
    }
}

// Nonrecursive Preorder Traversal
```

```c
void preorder(Node* root) {
    if (!root) return;
    Stack s; s.top = -1;
    push(&s, root);
    while (!isEmpty(&s)) {
        Node* curr = pop(&s);
        printf("%d ", curr->data);
        if (curr->right) push(&s, curr->right);
        if (curr->left) push(&s, curr->left);
    }
}

// Nonrecursive Count Leaf Nodes
int countLeafNodes(Node* root) {
    if (!root) return 0;
    Stack s; s.top = -1;
    int count = 0;
    push(&s, root);
    while (!isEmpty(&s)) {
        Node* curr = pop(&s);
        if (!curr->left && !curr->right)
            count++;
        if (curr->right) push(&s, curr->right);
        if (curr->left) push(&s, curr->left);
    }
    return count;
}

// Nonrecursive Mirror Image
void mirror(Node* root) {
    if (!root) return;
    Stack s; s.top = -1;
    push(&s, root);
    while (!isEmpty(&s)) {
        Node* curr = pop(&s);
        Node* temp = curr->left;
        curr->left = curr->right;
        curr->right = temp;
        if (curr->left) push(&s, curr->left);
        if (curr->right) push(&s, curr->right);
    }
}

int main() {
    Node* root = NULL;
    int ch, val;
    while (1) {
```

```
        printf("\n1.Insert\n2.Preorder\n3.Inorder\n4.Count Leaf
Nodes\n5.Mirror Image\n6.Exit\nEnter choice: ");
        scanf("%d", &ch);
        switch (ch) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &val);
                root = insert(root, val);
                break;
            case 2:
                printf("Preorder: ");
                preorder(root);
                printf("\n");
                break;
            case 3:
                printf("Inorder: ");
                inorder(root);
                printf("\n");
                break;
            case 4:
                printf("Number of leaf nodes: %d\n", countLeafNodes(root));
                break;
            case 5:
                mirror(root);
                printf("Mirror image created.\n");
                break;
            case 6:
                return 0;
            default:
                printf("Invalid choice.\n");
        }
    }
}
```

7. Write a Program to create a Binary Search Tree and perform following nonrecursive operations on it. a. Preorder Traversal b. Postorder Traversal c. Display total Number of Nodes d. Display Leaf nodes.

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

typedef struct Node {
    int data;
    struct Node *left, *right;
} Node;
```

```c
// Stack for non-recursive traversals
typedef struct Stack {
    Node* data[100];
    int top;
} Stack;

void push(Stack *s, Node *node) {
    s->data[++(s->top)] = node;
}

Node* pop(Stack *s) {
    return s->data[(s->top)--];
}

int isEmpty(Stack *s) {
    return s->top == -1;
}

// BST Insertion
Node* insert(Node* root, int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    if (!root) return newNode;
    Node *curr = root, *parent = NULL;
    while (curr) {
        parent = curr;
        if (data < curr->data)
            curr = curr->left;
        else
            curr = curr->right;
    }
    if (data < parent->data)
        parent->left = newNode;
    else
        parent->right = newNode;
    return root;
}

// Non-recursive Preorder Traversal
void preorder(Node* root) {
    if (!root) return;
    Stack s; s.top = -1;
    push(&s, root);
    while (!isEmpty(&s)) {
        Node* curr = pop(&s);
        printf("%d ", curr->data);
        if (curr->right) push(&s, curr->right);
```

```
        if (curr->left) push(&s, curr->left);
    }
}

// Non-recursive Postorder Traversal
void postorder(Node* root) {
    if (!root) return;
    Stack s1, s2;
    s1.top = s2.top = -1;
    push(&s1, root);
    while (!isEmpty(&s1)) {
        Node* curr = pop(&s1);
        push(&s2, curr);
        if (curr->left) push(&s1, curr->left);
        if (curr->right) push(&s1, curr->right);
    }
    while (!isEmpty(&s2)) {
        printf("%d ", pop(&s2)->data);
    }
}

// Count total nodes (non-recursive)
int countNodes(Node* root) {
    if (!root) return 0;
    Stack s; s.top = -1;
    int count = 0;
    push(&s, root);
    while (!isEmpty(&s)) {
        Node* curr = pop(&s);
        count++;
        if (curr->right) push(&s, curr->right);
        if (curr->left) push(&s, curr->left);
    }
    return count;
}

// Display leaf nodes (non-recursive)
void displayLeafNodes(Node* root) {
    if (!root) return;
    Stack s; s.top = -1;
    push(&s, root);
    while (!isEmpty(&s)) {
        Node* curr = pop(&s);
        if (!curr->left && !curr->right)
            printf("%d ", curr->data);
        if (curr->right) push(&s, curr->right);
        if (curr->left) push(&s, curr->left);
    }
```

```
}

int main() {
    Node* root = NULL;
    int choice, val;
    while (1) {
        printf("\n1. Insert\n2. Preorder Traversal\n3. Postorder Traversal\n4.
Total Nodes\n5. Display Leaf Nodes\n6. Exit\nEnter choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &val);
                root = insert(root, val);
                break;
            case 2:
                printf("Preorder: ");
                preorder(root);
                printf("\n");
                break;
            case 3:
                printf("Postorder: ");
                postorder(root);
                printf("\n");
                break;
            case 4:
                printf("Total nodes: %d\n", countNodes(root));
                break;
            case 5:
                printf("Leaf nodes: ");
                displayLeafNodes(root);
                printf("\n");
                break;
            case 6:
                return 0;
            default:
                printf("Invalid choice\n");
        }
    }
}
```

8. Write a Program to create a Binary Search Tree and perform deletion of a node from it. Also display the tree in nonrecursive postorder way.

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

```c
typedef struct node {
    int data;
    struct node *left, *right;
} Node;

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

// Insert a node into BST
Node* insert(Node* root, int data) {
    if (root == NULL)
        return newNode(data);
    if (data < root->data)
        root->left = insert(root->left, data);
    else if (data > root->data)
        root->right = insert(root->right, data);
    return root;
}

// Find minimum value node in BST
Node* minValueNode(Node* node) {
    Node* current = node;
    while (current && current->left != NULL)
        current = current->left;
    return current;
}

// Delete a node from BST
Node* deleteNode(Node* root, int key) {
    if (root == NULL)
        return root;
    if (key < root->data)
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
        root->right = deleteNode(root->right, key);
    else {
        // Node with only one child or no child
        if (root->left == NULL) {
            Node* temp = root->right;
            free(root);
            return temp;
        }
```

```
        else if (root->right == NULL) {
            Node* temp = root->left;
            free(root);
            return temp;
        }
        // Node with two children
        Node* temp = minValueNode(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

// Non-recursive postorder traversal using two stacks
void postorderNonRecursive(Node* root) {
    if (root == NULL)
        return;
    Node* stack1[100], *stack2[100];
    int top1 = -1, top2 = -1;
    stack1[++top1] = root;
    while (top1 >= 0) {
        Node* node = stack1[top1--];
        stack2[++top2] = node;
        if (node->left)
            stack1[++top1] = node->left;
        if (node->right)
            stack1[++top1] = node->right;
    }
    while (top2 >= 0) {
        printf("%d ", stack2[top2--]->data);
    }
    printf("\n");
}

int main() {
    Node* root = NULL;
    int choice, val, del;
    while (1) {
        printf("\n1. Insert\n2. Delete\n3. Postorder (Nonrecursive)\n4.
Exit\nEnter choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &val);
                root = insert(root, val);
                break;
            case 2:
```

```
                printf("Enter value to delete: ");
                scanf("%d", &del);
                root = deleteNode(root, del);
                break;
            case 3:
                printf("Postorder traversal: ");
                postorderNonRecursive(root);
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice!\n");
        }
    }
    return 0;
}
```

9. Write a Program to create a Binary Search Tree and display it levelwise. Also perform deletion of a node from it.

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

typedef struct Node {
    int data;
    struct Node *left, *right;
} Node;

// Queue for level order traversal
typedef struct QueueNode {
    Node *treeNode;
    struct QueueNode *next;
} QueueNode;

typedef struct Queue {
    QueueNode *front, *rear;
} Queue;

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

Node* insert(Node* root, int data) {
    if (root == NULL)
```

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

Node* minValueNode(Node* node) {
    Node* current = node;
    while (current && current->left != NULL)
        current = current->left;
    return current;
}

Node* deleteNode(Node* root, int key) {
    if (root == NULL)
        return root;
    if (key < root->data)
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
        root->right = deleteNode(root->right, key);
    else {
        if (root->left == NULL) {
            Node* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            Node* temp = root->left;
            free(root);
            return temp;
        }
        Node* temp = minValueNode(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

// Queue functions for level order
Queue* createQueue() {
    Queue* q = (Queue*)malloc(sizeof(Queue));
    q->front = q->rear = NULL;
    return q;
}

void enqueue(Queue* q, Node* node) {
```

```
    QueueNode* temp = (QueueNode*)malloc(sizeof(QueueNode));
    temp->treeNode = node;
    temp->next = NULL;
    if (q->rear == NULL) {
        q->front = q->rear = temp;
        return;
    }
    q->rear->next = temp;
    q->rear = temp;
}

Node* dequeue(Queue* q) {
    if (q->front == NULL)
        return NULL;
    QueueNode* temp = q->front;
    Node* node = temp->treeNode;
    q->front = q->front->next;
    if (q->front == NULL)
        q->rear = NULL;
    free(temp);
    return node;
}

int isQueueEmpty(Queue* q) {
    return q->front == NULL;
}

void levelOrder(Node* root) {
    if (root == NULL) {
        printf("Tree is empty.\n");
        return;
    }
    Queue* q = createQueue();
    enqueue(q, root);
    while (!isQueueEmpty(q)) {
        int nodeCount = 0;
        QueueNode* temp = q->front;
        while (temp) {
            nodeCount++;
            temp = temp->next;
        }
        while (nodeCount > 0) {
            Node* node = dequeue(q);
            printf("%d ", node->data);
            if (node->left)
                enqueue(q, node->left);
            if (node->right)
                enqueue(q, node->right);
```

```
            nodeCount--;
        }
        printf("\n");
    }
    free(q);
}

int main() {
    Node* root = NULL;
    int choice, data;
    while (1) {
        printf("\n1. Insert\n2. Delete\n3. Level Order Display\n4. Exit\nEnter
choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &data);
                root = insert(root, data);
                break;
            case 2:
                printf("Enter value to delete: ");
                scanf("%d", &data);
                root = deleteNode(root, data);
                break;
            case 3:
                printf("Level Order Traversal:\n");
                levelOrder(root);
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice.\n");
        }
    }
    return 0;
}
```

10. Write a Program to create a Binary Search Tree and display its mirror image with and without disturbing the original tree. Also display height of a tree using nonrecursion.

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

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

```c
    struct node *left, *right;
} Node;

// Create a new node
Node* newNode(int data) {
    Node* temp = (Node*)malloc(sizeof(Node));
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Insert into BST
Node* insert(Node* root, int data) {
    if (root == NULL)
        return newNode(data);
    if (data < root->data)
        root->left = insert(root->left, data);
    else if (data > root->data)
        root->right = insert(root->right, data);
    return root;
}

// Inorder traversal
void inorder(Node* root) {
    if (root) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

// Mirror the tree (modifies original)
void mirror(Node* root) {
    if (root) {
        Node* temp = root->left;
        root->left = root->right;
        root->right = temp;
        mirror(root->left);
        mirror(root->right);
    }
}

// Copy a tree
Node* copy(Node* root) {
    if (root == NULL)
        return NULL;
    Node* new_root = newNode(root->data);
    new_root->left = copy(root->left);
```

```
    new_root->right = copy(root->right);
    return new_root;
}

// Display mirror image without disturbing original
void displayMirror(Node* root) {
    if (root) {
        displayMirror(root->right);
        printf("%d ", root->data);
        displayMirror(root->left);
    }
}

// Calculate height non-recursively (using level order traversal)
int height(Node* root) {
    if (root == NULL)
        return 0;
    int height = 0;
    Node* queue[100];
    int front = 0, rear = 0;
    queue[rear++] = root;
    while (front < rear) {
        int nodeCount = rear - front;
        height++;
        while (nodeCount--) {
            Node* node = queue[front++];
            if (node->left)
                queue[rear++] = node->left;
            if (node->right)
                queue[rear++] = node->right;
        }
    }
    return height;
}

int main() {
    Node* root = NULL;
    int n, val, ch;
    printf("Enter number of nodes: ");
    scanf("%d", &n);
    printf("Enter node values:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &val);
        root = insert(root, val);
    }
    while (1) {
```

```
        printf("\n1. Inorder Traversal\n2. Display Mirror (without disturbing
original)\n3. Display Mirror (modifies original)\n4. Height (non-
recursive)\n5. Exit\nEnter choice: ");
        scanf("%d", &ch);
        if (ch == 1) {
            printf("Inorder: ");
            inorder(root);
            printf("\n");
        } else if (ch == 2) {
            printf("Mirror Image (without disturbing original): ");
            displayMirror(root);
            printf("\n");
        } else if (ch == 3) {
            Node* mirrorRoot = copy(root);
            mirror(mirrorRoot);
            printf("Inorder of Mirrored Tree: ");
            inorder(mirrorRoot);
            printf("\n");
            // Freeing mirrorRoot is omitted for brevity
        } else if (ch == 4) {
            printf("Height of tree: %d\n", height(root));
        } else if (ch == 5) {
            break;
        } else {
            printf("Invalid choice\n");
        }
    }
    return 0;
}
```

11. Write a program to efficiently search a particular employee record by using Tree data structure. Also sort the data on emp-id in ascending order.

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

typedef struct Employee {
    int emp_id;
    char name[50];
    float salary;
} Employee;

typedef struct Node {
    Employee data;
    struct Node *left, *right;
```

```c
} Node;

// Create a new node
Node* createNode(Employee emp) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = emp;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// Insert employee into BST
Node* insert(Node* root, Employee emp) {
    if (root == NULL)
        return createNode(emp);
    if (emp.emp_id < root->data.emp_id)
        root->left = insert(root->left, emp);
    else if (emp.emp_id > root->data.emp_id)
        root->right = insert(root->right, emp);
    return root;
}

// Inorder traversal to sort by emp_id
void inorder(Node* root) {
    if (root) {
        inorder(root->left);
        printf("Emp ID: %d, Name: %s, Salary: %.2f\n", root->data.emp_id,
root->data.name, root->data.salary);
        inorder(root->right);
    }
}

// Search for an employee by emp_id
Node* search(Node* root, int emp_id) {
    if (root == NULL || root->data.emp_id == emp_id)
        return root;
    if (emp_id < root->data.emp_id)
        return search(root->left, emp_id);
    else
        return search(root->right, emp_id);
}

int main() {
    Node* root = NULL;
    int n, i, emp_id;
    Employee emp;

    printf("Enter number of employees: ");
    scanf("%d", &n);
```

```
    for (i = 0; i < n; i++) {
        printf("Enter Employee ID: ");
        scanf("%d", &emp.emp_id);
        printf("Enter Name: ");
        scanf("%s", emp.name);
        printf("Enter Salary: ");
        scanf("%f", &emp.salary);
        root = insert(root, emp);
    }

    printf("\nEmployees sorted by Emp ID:\n");
    inorder(root);

    printf("\nEnter Employee ID to search: ");
    scanf("%d", &emp_id);
    Node* found = search(root, emp_id);
    if (found) {
        printf("Record Found: Emp ID: %d, Name: %s, Salary: %.2f\n", found-
>data.emp_id, found->data.name, found->data.salary);
    } else {
        printf("Employee with ID %d not found.\n", emp_id);
    }

    return 0;
}
```

12. Write a Program to create Inorder Threaded Binary Tree and Traverse it in Preorder way.

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

typedef struct Node {
    int data;
    struct Node *left, *right;
    int lthread, rthread;
} Node;

Node* insert(Node* root, int key) {
    Node *ptr = root;
    Node *par = NULL;

    while (ptr != NULL) {
        if (key == ptr->data) {
            printf("Duplicate Key!\n");
            return root;
        }
```

```
        par = ptr;
        if (key < ptr->data) {
            if (ptr->lthread == 0)
                ptr = ptr->left;
            else
                break;
        } else {
            if (ptr->rthread == 0)
                ptr = ptr->right;
            else
                break;
        }
    }

    Node *tmp = (Node*)malloc(sizeof(Node));
    tmp->data = key;
    tmp->lthread = 1;
    tmp->rthread = 1;

    if (par == NULL) {
        tmp->left = tmp->right = NULL;
        root = tmp;
    } else if (key < par->data) {
        tmp->left = par->left;
        tmp->right = par;
        par->lthread = 0;
        par->left = tmp;
    } else {
        tmp->left = par;
        tmp->right = par->right;
        par->rthread = 0;
        par->right = tmp;
    }
    return root;
}

Node* leftMost(Node* node) {
    if (node == NULL)
        return NULL;
    while (node->lthread == 0)
        node = node->left;
    return node;
}

void preorder(Node* root) {
    Node* curr = root;
    while (curr != NULL) {
        printf("%d ", curr->data);
```

```
        if (curr->lthread == 0)
            curr = curr->left;
        else if (curr->rthread == 0)
            curr = curr->right;
        else {
            while (curr != NULL && curr->rthread == 1)
                curr = curr->right;
            if (curr != NULL)
                curr = curr->right;
        }
    }
}

int main() {
    Node* root = NULL;
    int n, val;
    printf("Enter number of nodes: ");
    scanf("%d", &n);
    printf("Enter node values:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &val);
        root = insert(root, val);
    }
    printf("Preorder Traversal: ");
    preorder(root);
    printf("\n");
    return 0;
}
```

13. Write a Program to create Inorder Threaded Binary Tree and Traverse it in Inorder way.

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

typedef struct Node {
    int data;
    struct Node *left, *right;
    int lthread, rthread;
} Node;

Node* createNode(int data) {
    Node* temp = (Node*)malloc(sizeof(Node));
    temp->data = data;
    temp->left = temp->right = NULL;
    temp->lthread = temp->rthread = 1;
    return temp;
```

```
}

Node* insert(Node* root, int key) {
    Node *ptr = root, *par = NULL;
    while (ptr != NULL) {
        if (key == ptr->data) {
            printf("Duplicate Key!\n");
            return root;
        }
        par = ptr;
        if (key < ptr->data) {
            if (ptr->lthread == 0)
                ptr = ptr->left;
            else
                break;
        } else {
            if (ptr->rthread == 0)
                ptr = ptr->right;
            else
                break;
        }
    }
    Node* tmp = createNode(key);
    if (par == NULL) {
        root = tmp;
    } else if (key < par->data) {
        tmp->left = par->left;
        tmp->right = par;
        par->lthread = 0;
        par->left = tmp;
    } else {
        tmp->left = par;
        tmp->right = par->right;
        par->rthread = 0;
        par->right = tmp;
    }
    return root;
}

Node* leftMost(Node* n) {
    if (n == NULL) return NULL;
    while (n->lthread == 0)
        n = n->left;
    return n;
}

void inorder(Node* root) {
    Node* cur = leftMost(root);
```

```
    while (cur != NULL) {
        printf("%d ", cur->data);
        if (cur->rthread)
            cur = cur->right;
        else
            cur = leftMost(cur->right);
    }
}

int main() {
    Node* root = NULL;
    int n, val;
    printf("Enter number of nodes: ");
    scanf("%d", &n);
    printf("Enter node values:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &val);
        root = insert(root, val);
    }
    printf("Inorder Traversal: ");
    inorder(root);
    printf("\n");
    return 0;
}
```

14. Write a Program to implement AVL tree and perform different rotations on it and display it Levelwise.

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

typedef struct Node {
    int key, height;
    struct Node *left, *right;
} Node;

// Utility functions
int max(int a, int b) {
    return (a > b) ? a : b;
}

int height(Node *N) {
    if (N == NULL)
        return 0;
```

```
    return N->height;
}

Node* newNode(int key) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->key = key;
    node->left = node->right = NULL;
    node->height = 1;
    return node;
}

// Rotations
Node* rightRotate(Node *y) {
    Node *x = y->left;
    Node *T2 = x->right;
    x->right = y;
    y->left = T2;
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;
    return x;
}

Node* leftRotate(Node *x) {
    Node *y = x->right;
    Node *T2 = y->left;
    y->left = x;
    x->right = T2;
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;
    return y;
}

int getBalance(Node *N) {
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

// AVL insert
Node* insert(Node* node, int key) {
    if (node == NULL)
        return newNode(key);
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else
        return node;
```

```
    node->height = 1 + max(height(node->left), height(node->right));
    int balance = getBalance(node);

    // Left Left
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

// Level order display
void printLevelOrder(Node* root) {
    if (root == NULL)
        return;
    Node* queue[100];
    int front = 0, rear = 0;
    queue[rear++] = root;
    while (front < rear) {
        int nodeCount = rear - front;
        while (nodeCount > 0) {
            Node* node = queue[front++];
            printf("%d ", node->key);
            if (node->left)
                queue[rear++] = node->left;
            if (node->right)
                queue[rear++] = node->right;
            nodeCount--;
        }
        printf("\n");
    }
}
```

```c
int main() {
    Node *root = NULL;
    int n, val;
    printf("Enter number of nodes: ");
    scanf("%d", &n);
    printf("Enter the values:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &val);
        root = insert(root, val);
    }
    printf("Level order traversal of AVL tree:\n");
    printLevelOrder(root);
    return 0;
}
```

15. Write a Program to accept a graph (directed or undirected) from user and represent it with Adjacency Matrix and perform BFS and DFS traversals on it.

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

#define MAX 20

int adj[MAX][MAX], visited[MAX], n;

void createGraph(int directed) {
    int i, j, edges, src, dest;
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter number of edges: ");
    scanf("%d", &edges);

    // Initialize adjacency matrix
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            adj[i][j] = 0;

    printf("Enter edges (source destination):\n");
    for (i = 0; i < edges; i++) {
        scanf("%d%d", &src, &dest);
        adj[src][dest] = 1;
        if (!directed)
            adj[dest][src] = 1;
    }
}
```

```c
void bfs(int start) {
    int queue[MAX], front = 0, rear = 0, i, v;
    for (i = 0; i < n; i++)
        visited[i] = 0;

    queue[rear++] = start;
    visited[start] = 1;

    printf("BFS Traversal: ");
    while (front < rear) {
        v = queue[front++];
        printf("%d ", v);
        for (i = 0; i < n; i++) {
            if (adj[v][i] && !visited[i]) {
                queue[rear++] = i;
                visited[i] = 1;
            }
        }
    }
    printf("\n");
}

void dfsUtil(int v) {
    int i;
    visited[v] = 1;
    printf("%d ", v);
    for (i = 0; i < n; i++) {
        if (adj[v][i] && !visited[i])
            dfsUtil(i);
    }
}

void dfs(int start) {
    int i;
    for (i = 0; i < n; i++)
        visited[i] = 0;
    printf("DFS Traversal: ");
    dfsUtil(start);
    printf("\n");
}

int main() {
    int directed, start;
    printf("Enter 0 for undirected graph, 1 for directed graph: ");
    scanf("%d", &directed);
    createGraph(directed);
```

```
    printf("Enter starting vertex for traversals: ");
    scanf("%d", &start);

    bfs(start);
    dfs(start);

    return 0;
}
```

16. Write a Program to accept a graph (directed or undirected) from user and represent it with Adjacency List and perform BFS and DFS traversals on it.

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

#define MAX_VERTICES 100

typedef struct Node {
    int vertex;
    struct Node* next;
} Node;

typedef struct Graph {
    int numVertices;
    Node* adjList[MAX_VERTICES];
    int isDirected;
} Graph;

typedef struct Queue {
    int items[MAX_VERTICES];
    int front, rear;
} Queue;

// Queue functions
void initQueue(Queue* q) {
    q->front = q->rear = -1;
}

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

void enqueue(Queue* q, int value) {
    if (q->rear == MAX_VERTICES - 1) return;
    if (isEmpty(q)) q->front = 0;
    q->items[++q->rear] = value;
}
```

```c
int dequeue(Queue* q) {
    if (isEmpty(q)) return -1;
    int value = q->items[q->front];
    if (q->front == q->rear)
        q->front = q->rear = -1;
    else
        q->front++;
    return value;
}

// Graph functions
Graph* createGraph(int vertices, int isDirected) {
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->numVertices = vertices;
    graph->isDirected = isDirected;
    for (int i = 0; i < vertices; i++)
        graph->adjList[i] = NULL;
    return graph;
}

void addEdge(Graph* graph, int src, int dest) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->vertex = dest;
    newNode->next = graph->adjList[src];
    graph->adjList[src] = newNode;

    if (!graph->isDirected) {
        newNode = (Node*)malloc(sizeof(Node));
        newNode->vertex = src;
        newNode->next = graph->adjList[dest];
        graph->adjList[dest] = newNode;
    }
}

void printGraph(Graph* graph) {
    printf("Adjacency List:\n");
    for (int v = 0; v < graph->numVertices; v++) {
        Node* temp = graph->adjList[v];
        printf("%d: ", v);
        while (temp) {
            printf("%d -> ", temp->vertex);
            temp = temp->next;
        }
        printf("NULL\n");
    }
}
```

```c
void BFS(Graph* graph, int startVertex) {
    int visited[MAX_VERTICES] = {0};
    Queue q;
    initQueue(&q);

    visited[startVertex] = 1;
    enqueue(&q, startVertex);

    printf("BFS Traversal: ");
    while (!isEmpty(&q)) {
        int currVertex = dequeue(&q);
        printf("%d ", currVertex);

        Node* temp = graph->adjList[currVertex];
        while (temp) {
            int adjVertex = temp->vertex;
            if (!visited[adjVertex]) {
                visited[adjVertex] = 1;
                enqueue(&q, adjVertex);
            }
            temp = temp->next;
        }
    }
    printf("\n");
}

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

    Node* temp = graph->adjList[vertex];
    while (temp) {
        int adjVertex = temp->vertex;
        if (!visited[adjVertex])
            DFSUtil(graph, adjVertex, visited);
        temp = temp->next;
    }
}

void DFS(Graph* graph, int startVertex) {
    int visited[MAX_VERTICES] = {0};
    printf("DFS Traversal: ");
    DFSUtil(graph, startVertex, visited);
    printf("\n");
}

int main() {
    int vertices, edges, isDirected, src, dest, start;
```

```
    printf("Enter number of vertices: ");
    scanf("%d", &vertices);
    printf("Enter number of edges: ");
    scanf("%d", &edges);
    printf("Enter 1 for Directed graph, 0 for Undirected graph: ");
    scanf("%d", &isDirected);

    Graph* graph = createGraph(vertices, isDirected);

    printf("Enter edges (src dest):\n");
    for (int i = 0; i < edges; i++) {
        scanf("%d %d", &src, &dest);
        addEdge(graph, src, dest);
    }

    printGraph(graph);

    printf("Enter starting vertex for BFS and DFS: ");
    scanf("%d", &start);

    BFS(graph, start);
    DFS(graph, start);

    // Free memory
    for (int i = 0; i < vertices; i++) {
        Node* temp = graph->adjList[i];
        while (temp) {
            Node* toFree = temp;
            temp = temp->next;
            free(toFree);
        }
    }
    free(graph);

    return 0;
}
```

17. Write a Program to implement Prim's algorithm to find minimum spanning tree of a user defined graph. Use Adjacency Matrix/List to represent a graph.

```
#include <stdio.h>

#define MAX 100

int minKey(int key[], int mstSet[], int V) {
    int min = 1000000000, min_index = -1; // Use a large constant instead of
INT_MAX
```

```c
    for (int v = 0; v < V; v++)
        if (mstSet[v] == 0 && key[v] < min)
            min = key[v], min_index = v;
    return min_index;
}

void printMST(int parent[], int graph[MAX][MAX], int V) {
    int total = 0;
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++) {
        printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
        total += graph[i][parent[i]];
    }
    printf("Total weight of MST: %d\n", total);
}

void primMST(int graph[MAX][MAX], int V) {
    int parent[MAX];
    int key[MAX];
    int mstSet[MAX];

    for (int i = 0; i < V; i++)
        key[i] = 1000000000, mstSet[i] = 0; // Use a large constant

    key[0] = 0;
    parent[0] = -1;

    for (int count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet, V);
        mstSet[u] = 1;

        for (int v = 0; v < V; v++)
            if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    printMST(parent, graph, V);
}

int main() {
    int V, graph[MAX][MAX];
    printf("Enter number of vertices: ");
    scanf("%d", &V);
    printf("Enter adjacency matrix (0 if no edge):\n");
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            scanf("%d", &graph[i][j]);
    primMST(graph, V);
```

```
    return 0;
}
```

18. Write a Program to implement Dijkstra's algorithm to find shortest distance between two nodes of a user defined graph. Use Adjacency Matrix/List to represent a graph.\

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

#define MAX 100

int minDistance(int dist[], int sptSet[], int n) {
    int min = INT_MAX, min_index = -1;
    for (int v = 0; v < n; v++)
        if (!sptSet[v] && dist[v] <= min)
            min = dist[v], min_index = v;
    return min_index;
}

void dijkstra(int graph[MAX][MAX], int n, int src, int dest) {
    int dist[MAX];
    int sptSet[MAX];

    for (int i = 0; i < n; i++)
        dist[i] = INT_MAX, sptSet[i] = 0;

    dist[src] = 0;

    for (int count = 0; count < n - 1; count++) {
        int u = minDistance(dist, sptSet, n);
        if (u == -1) break;
        sptSet[u] = 1;

        for (int v = 0; v < n; v++)
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }

    if (dist[dest] == INT_MAX)
        printf("No path exists between node %d and node %d\n", src, dest);
    else
        printf("Shortest distance between node %d and node %d is %d\n", src,
dest, dist[dest]);
}
```

```c
int main() {
    int n, graph[MAX][MAX];
    printf("Enter number of nodes: ");
    scanf("%d", &n);

    printf("Enter adjacency matrix (0 if no edge):\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &graph[i][j]);

    int src, dest;
    printf("Enter source node (0 to %d): ", n-1);
    scanf("%d", &src);
    printf("Enter destination node (0 to %d): ", n-1);
    scanf("%d", &dest);

    dijkstra(graph, n, src, dest);

    return 0;
}
```

19. Write a Program to implement Kruskal's algorithm to find minimum spanning tree of a user defined graph. Use Adjacency Matrix/ List to represent a graph.

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

#define MAX 100

typedef struct {
    int u, v, w;
} Edge;

int parent[MAX];

int find(int i) {
    while (parent[i] != i)
        i = parent[i];
    return i;
}

void union_set(int i, int j) {
    int a = find(i);
    int b = find(j);
    parent[a] = b;
}
```

```c
int main() {
    int n, adj[MAX][MAX], i, j, k = 0, mincost = 0, edge_count = 0;
    Edge edges[MAX * MAX], mst[MAX];

    printf("Enter number of vertices: ");
    scanf("%d", &n);

    printf("Enter adjacency matrix (0 if no edge):\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            scanf("%d", &adj[i][j]);

    // Collect all edges
    for (i = 0; i < n; i++) {
        for (j = i + 1; j < n; j++) {
            if (adj[i][j] != 0) {
                edges[k].u = i;
                edges[k].v = j;
                edges[k].w = adj[i][j];
                k++;
            }
        }
    }

    // Sort edges by weight
    for (i = 0; i < k - 1; i++) {
        for (j = 0; j < k - i - 1; j++) {
            if (edges[j].w > edges[j + 1].w) {
                Edge temp = edges[j];
                edges[j] = edges[j + 1];
                edges[j + 1] = temp;
            }
        }
    }

    // Initialize parent array
    for (i = 0; i < n; i++)
        parent[i] = i;

    // Kruskal's algorithm
    for (i = 0; i < k && edge_count < n - 1; i++) {
        int u = edges[i].u;
        int v = edges[i].v;
        int set_u = find(u);
        int set_v = find(v);

        if (set_u != set_v) {
```

```
            mst[edge_count++] = edges[i];
            mincost += edges[i].w;
            union_set(set_u, set_v);
        }
    }

    printf("Edges in Minimum Spanning Tree:\n");
    for (i = 0; i < edge_count; i++)
        printf("%d -- %d == %d\n", mst[i].u, mst[i].v, mst[i].w);

    printf("Minimum Cost = %d\n", mincost);

    return 0;
}
```

20. WAP to implement Heap sort and Quick Sort on 1D array of Student structure (contains student_name, student_roll_no, total_marks), with key as student_roll_no. And count the number of swap performed.

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

#define MAX 100

typedef struct {
    char student_name[50];
    int student_roll_no;
    int total_marks;
} Student;

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

// Heapify function for Heap Sort
void heapify(Student arr[], int n, int i, int *swap_count) {
    int largest = i;
    int l = 2*i + 1;
    int r = 2*i + 2;

    if (l < n && arr[l].student_roll_no > arr[largest].student_roll_no)
        largest = l;
    if (r < n && arr[r].student_roll_no > arr[largest].student_roll_no)
        largest = r;
```

```
    if (largest != i) {
        swap(&arr[i], &arr[largest], swap_count);
        heapify(arr, n, largest, swap_count);
    }
}

void heapSort(Student arr[], int n, int *swap_count) {
    for (int i = n/2 - 1; i >= 0; i--)
        heapify(arr, n, i, swap_count);

    for (int i = n-1; i > 0; i--) {
        swap(&arr[0], &arr[i], swap_count);
        heapify(arr, i, 0, swap_count);
    }
}

// Partition function for Quick Sort
int partition(Student arr[], int low, int high, int *swap_count) {
    int pivot = arr[high].student_roll_no;
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j].student_roll_no < pivot) {
            i++;
            swap(&arr[i], &arr[j], swap_count);
        }
    }
    swap(&arr[i+1], &arr[high], swap_count);
    return i+1;
}

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

void printStudents(Student arr[], int n) {
    printf("Name\tRoll No\tTotal Marks\n");
    for (int i = 0; i < n; i++) {
        printf("%s\t%d\t%d\n", arr[i].student_name, arr[i].student_roll_no,
arr[i].total_marks);
    }
}

int main() {
    Student arr[MAX];
```

```
    int n, i;
    int heap_swaps = 0, quick_swaps = 0;

    printf("Enter number of students: ");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        printf("Enter name, roll no, total marks for student %d: ", i+1);
        scanf("%s %d %d", arr[i].student_name, &arr[i].student_roll_no,
&arr[i].total_marks);
    }

    Student heap_arr[MAX], quick_arr[MAX];
    memcpy(heap_arr, arr, n * sizeof(Student));
    memcpy(quick_arr, arr, n * sizeof(Student));

    heapSort(heap_arr, n, &heap_swaps);
    printf("\nHeap Sort Result:\n");
    printStudents(heap_arr, n);
    printf("Heap Sort Swaps: %d\n", heap_swaps);

    quickSort(quick_arr, 0, n-1, &quick_swaps);
    printf("\nQuick Sort Result:\n");
    printStudents(quick_arr, n);
    printf("Quick Sort Swaps: %d\n", quick_swaps);

    return 0;
}
```

21. WAP to implement Heap sort and Merge Sort on 1D array of Student structure (contains student_name, student_roll_no, total_marks), with key as student_roll_no. And count the number of swap performed.

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

#define MAX 100

typedef struct {
    char student_name[50];
    int student_roll_no;
    int total_marks;
} Student;

// Swap function and swap counter
int swap_count = 0;
void swap(Student *a, Student *b) {
    Student temp = *a;
```

```
    *a = *b;
    *b = temp;
    swap_count++;
}

// Heapify function for Heap Sort
void heapify(Student arr[], int n, int i) {
    int largest = i;
    int l = 2*i + 1;
    int r = 2*i + 2;

    if (l < n && arr[l].student_roll_no > arr[largest].student_roll_no)
        largest = l;
    if (r < n && arr[r].student_roll_no > arr[largest].student_roll_no)
        largest = r;

    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        heapify(arr, n, largest);
    }
}

// Heap Sort
void heapSort(Student arr[], int n) {
    swap_count = 0;
    for (int i = n/2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (int i = n-1; i > 0; i--) {
        swap(&arr[0], &arr[i]);
        heapify(arr, i, 0);
    }
}

// Merge function for Merge Sort
void merge(Student arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    Student L[n1], R[n2];
    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i].student_roll_no <= R[j].student_roll_no) {
```

```
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
            swap_count++; // Counting as a swap when elements from right
subarray are placed before left
        }
    }
    while (i < n1)
        arr[k++] = L[i++];
    while (j < n2)
        arr[k++] = R[j++];
}

// Merge Sort
void mergeSort(Student arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

// Function to print array
void printArray(Student arr[], int n) {
    printf("Name\tRoll No\tMarks\n");
    for (int i = 0; i < n; i++)
        printf("%s\t%d\t%d\n", arr[i].student_name, arr[i].student_roll_no,
arr[i].total_marks);
}

int main() {
    int n, choice;
    Student arr[MAX], arr_copy[MAX];

    printf("Enter number of students: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        printf("Enter name, roll no, total marks for student %d: ", i+1);
        scanf("%s %d %d", arr[i].student_name, &arr[i].student_roll_no,
&arr[i].total_marks);
    }

    do {
        printf("\n1. Heap Sort\n2. Merge Sort\n3. Exit\nEnter choice: ");
        scanf("%d", &choice);

        if (choice == 1) {
```

```
            memcpy(arr_copy, arr, n * sizeof(Student));
            heapSort(arr_copy, n);
            printf("\nSorted array (Heap Sort):\n");
            printArray(arr_copy, n);
            printf("Number of swaps: %d\n", swap_count);
        } else if (choice == 2) {
            memcpy(arr_copy, arr, n * sizeof(Student));
            swap_count = 0;
            mergeSort(arr_copy, 0, n-1);
            printf("\nSorted array (Merge Sort):\n");
            printArray(arr_copy, n);
            printf("Number of swaps: %d\n", swap_count);
        }
    } while (choice != 3);

    return 0;
}
```

22. WAP to convert a given Infix expression into its equivalent Postfix expression and evaluate it using stack.

```
#include <stdio.h>

#define MAX 100

// Stack for operators
char stack[MAX];
int top = -1;

// Stack for evaluation
int evalStack[MAX];
int evalTop = -1;

void push(char c) {
    stack[++top] = c;
}

char pop() {
    if (top == -1) return -1;
    return stack[top--];
}

char peek() {
    if (top == -1) return -1;
    return stack[top];
}
```

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

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

void evalPush(int val) {
    evalStack[++evalTop] = val;
}

int evalPop() {
    return evalStack[evalTop--];
}

int isdigit(char c) {
    return (c >= '0' && c <= '9');
}

void infixToPostfix(char* infix, char* postfix) {
    int i = 0, k = 0;
    char c;
    while ((c = infix[i++]) != '\0') {
        if (isdigit(c)) {
            postfix[k++] = c;
        } else if (c == '(') {
            push(c);
        } else if (c == ')') {
            while (top != -1 && peek() != '(')
                postfix[k++] = pop();
            pop(); // remove '('
        } else if (isOperator(c)) {
            while (top != -1 && precedence(peek()) >= precedence(c))
                postfix[k++] = pop();
            push(c);
        }
    }
    while (top != -1)
        postfix[k++] = pop();
    postfix[k] = '\0';
}

int evaluatePostfix(char* postfix) {
    int i = 0;
    char c;
```

```
    while ((c = postfix[i++]) != '\0') {
        if (isdigit(c)) {
            evalPush(c - '0');
        } else if (isOperator(c)) {
            int b = evalPop();
            int a = evalPop();
            switch (c) {
                case '+': evalPush(a + b); break;
                case '-': evalPush(a - b); break;
                case '*': evalPush(a * b); break;
                case '/': evalPush(a / b); break;
            }
        }
    }
    return evalPop();
}

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

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

    int result = evaluatePostfix(postfix);
    printf("Evaluation result: %d\n", result);

    return 0;
}
```

23. WAP to convert a given Infix expression into its equivalent Prefix expression and evaluate it using stack.

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

#define MAX 100

char stack[MAX];
int top = -1;

void push(char c) {
    if (top < MAX - 1)
        stack[++top] = c;
}
```

```c
char pop() {
    if (top >= 0)
        return stack[top--];
    return '\0';
}

char peek() {
    if (top >= 0)
        return stack[top];
    return '\0';
}

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

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

void reverse(char *exp) {
    int l = 0, r = strlen(exp) - 1;
    while (l < r) {
        char t = exp[l];
        exp[l] = exp[r];
        exp[r] = t;
        l++; r--;
    }
}

// Check if character is a digit (0-9)
int is_digit(char c) {
    return (c >= '0' && c <= '9');
}

// Check if character is an alphabet (A-Z or a-z)
int is_alpha(char c) {
    return ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z'));
}

// Check if character is alphanumeric
```

```
int is_alnum(char c) {
    return is_digit(c) || is_alpha(c);
}

void infixToPrefix(char infix[], char prefix[]) {
    int i, j = 0;
    int len = strlen(infix);
    char exp[MAX];
    strcpy(exp, infix);
    reverse(exp);

    for (i = 0; i < len; i++) {
        if (exp[i] == '(')
            exp[i] = ')';
        else if (exp[i] == ')')
            exp[i] = '(';
    }

    for (i = 0; i < len; i++) {
        char c = exp[i];
        if (is_alnum(c)) {
            prefix[j++] = c;
        } else if (c == '(') {
            push(c);
        } else if (c == ')') {
            while (top != -1 && peek() != '(')
                prefix[j++] = pop();
            pop();
        } else if (isOperator(c)) {
            while (top != -1 && precedence(peek()) > precedence(c))
                prefix[j++] = pop();
            push(c);
        }
    }
    while (top != -1)
        prefix[j++] = pop();
    prefix[j] = '\0';
    reverse(prefix);
}

int evalPrefix(char prefix[]) {
    int i, len = strlen(prefix);
    int stack2[MAX], top2 = -1;
    for (i = len - 1; i >= 0; i--) {
        char c = prefix[i];
        if (is_digit(c)) {
            stack2[++top2] = c - '0';
        } else if (isOperator(c)) {
```

```
            int op1 = stack2[top2--];
            int op2 = stack2[top2--];
            int res;
            switch (c) {
                case '+': res = op1 + op2; break;
                case '-': res = op1 - op2; break;
                case '*': res = op1 * op2; break;
                case '/': res = op1 / op2; break;
                case '^': {
                    int r = 1, j;
                    for (j = 0; j < op2; j++)
                        r *= op1;
                    res = r;
                    break;
                }
                default: res = 0;
            }
            stack2[++top2] = res;
        }
    }
    return stack2[top2];
}

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

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

    int result = evalPrefix(prefix);
    printf("Evaluation result: %d\n", result);

    return 0;
}
```

24. WAP to implement multiple stack i.e. More than two stacks using array and perform following operations on it. A. PUSH, B. POP, C. StackFull D. StackeEmpty E. Display Stack.

```
#include <stdio.h>
#define MAX 100
#define NSTACKS 3 // Number of stacks

int arr[MAX];
int top[NSTACKS];
```

```c
int base[NSTACKS];
int size[NSTACKS] = {0}; // To keep track of each stack's size

// Initialize the stacks
void initStacks(int n, int totalSize) {
    int partition = totalSize / n;
    for (int i = 0; i < n; i++) {
        base[i] = i * partition;
        top[i] = base[i] - 1;
        size[i] = partition;
    }
    // Last stack gets any remaining space
    size[n-1] += totalSize % n;
}

// Check if stack is full
int isFull(int sn) {
    return (top[sn] == base[sn] + size[sn] - 1);
}

// Check if stack is empty
int isEmpty(int sn) {
    return (top[sn] < base[sn]);
}

// Push operation
void push(int sn, int val) {
    if (isFull(sn)) {
        printf("Stack %d is Full!\n", sn+1);
        return;
    }
    top[sn]++;
    arr[top[sn]] = val;
    printf("Pushed %d to Stack %d\n", val, sn+1);
}

// Pop operation
int pop(int sn) {
    if (isEmpty(sn)) {
        printf("Stack %d is Empty!\n", sn+1);
        return -1;
    }
    int val = arr[top[sn]];
    top[sn]--;
    printf("Popped %d from Stack %d\n", val, sn+1);
    return val;
}
```

```c
// Display stack
void display(int sn) {
    if (isEmpty(sn)) {
        printf("Stack %d is Empty!\n", sn+1);
        return;
    }
    printf("Stack %d: ", sn+1);
    for (int i = base[sn]; i <= top[sn]; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int n = NSTACKS, totalSize = MAX;
    initStacks(n, totalSize);

    int choice, stackNum, val;
    while (1) {
        printf("\n1. PUSH\n2. POP\n3. StackFull\n4. StackEmpty\n5. Display
Stack\n6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        if (choice == 6) break;
        if (choice >= 1 && choice <= 5) {
            printf("Enter stack number (1-%d): ", n);
            scanf("%d", &stackNum);
            stackNum--;
            if (stackNum < 0 || stackNum >= n) {
                printf("Invalid stack number!\n");
                continue;
            }
        }
        switch (choice) {
            case 1:
                printf("Enter value to push: ");
                scanf("%d", &val);
                push(stackNum, val);
                break;
            case 2:
                pop(stackNum);
                break;
            case 3:
                if (isFull(stackNum))
                    printf("Stack %d is Full!\n", stackNum+1);
                else
                    printf("Stack %d is NOT Full!\n", stackNum+1);
                break;
```

```
            case 4:
                if (isEmpty(stackNum))
                    printf("Stack %d is Empty!\n", stackNum+1);
                else
                    printf("Stack %d is NOT Empty!\n", stackNum+1);
                break;
            case 5:
                display(stackNum);
                break;
            default:
                printf("Invalid choice!\n");
        }
    }
    return 0;
}
```

25. A person is living in house having 5 rooms. These rooms are adjacent to each other. There is a treasure which is electronically locked and to unlock it we need a code. In last room there is a box in which some decimal number is written. We need to convert that number into binary to open treasure which is kept in room no.1. We need to move from room no.1 to 2 to 3 and so on and follow the reverse way to come back i.e. from 5 to 4 to 3 etc. Apply suitable logic to implement this scenario by using stacks.

```
#include <stdio.h>
#define MAX 100

// Stack implementation
typedef struct {
    int data[MAX];
    int top;
} Stack;

void init(Stack *s) {
    s->top = -1;
}

int isEmpty(Stack *s) {
    return s->top == -1;
}

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

void push(Stack *s, int val) {
    if (!isFull(s)) {
        s->data[++(s->top)] = val;
    }
```

```c
}

int pop(Stack *s) {
    if (!isEmpty(s)) {
        return s->data[(s->top)--];
    }
    return -1;
}

int main() {
    int rooms[5] = {0}; // Simulate 5 rooms
    int i, decimal, temp;
    Stack path;
    init(&path);

    printf("Moving from Room 1 to Room 5...\n");
    for (i = 0; i < 5; i++) {
        printf("Entered Room %d\n", i + 1);
        push(&path, i + 1);
    }

    printf("You found a box in Room 5!\n");
    printf("Enter the decimal number written in the box: ");
    scanf("%d", &decimal);

    // Convert decimal to binary using stack
    Stack binStack;
    init(&binStack);
    temp = decimal;
    if (temp == 0) {
        push(&binStack, 0);
    } else {
        while (temp > 0) {
            push(&binStack, temp % 2);
            temp /= 2;
        }
    }

    printf("Returning from Room 5 to Room 1...\n");
    while (!isEmpty(&path)) {
        printf("Exited Room %d\n", pop(&path));
    }

    printf("Binary code to unlock the treasure in Room 1: ");
    while (!isEmpty(&binStack)) {
        printf("%d", pop(&binStack));
    }
    printf("\n");
```

```
    return 0;
}
```

26. WAP to implement following by using stack. A. Factorial of a given number B. Generation of Fibonacci series.

```
#include <stdio.h>
#define MAX 100

// Stack implementation
typedef struct {
    int data[MAX];
    int top;
} Stack;

void init(Stack *s) {
    s->top = -1;
}

int isEmpty(Stack *s) {
    return s->top == -1;
}

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

void push(Stack *s, int val) {
    if (!isFull(s)) {
        s->data[++(s->top)] = val;
    }
}

int pop(Stack *s) {
    if (!isEmpty(s)) {
        return s->data[(s->top)--];
    }
    return -1;
}

// Factorial using stack (iterative)
int factorial(int n) {
    Stack s;
```

```
    init(&s);
    int res = 1;
    while (n > 1) {
        push(&s, n);
        n--;
    }
    while (!isEmpty(&s)) {
        res *= pop(&s);
    }
    return res;
}

// Fibonacci series generation using stack (iterative)
void fibonacci(int n) {
    Stack s;
    init(&s);
    int a = 0, b = 1, i;
    printf("Fibonacci series: ");
    if (n >= 1) printf("%d ", a);
    if (n >= 2) printf("%d ", b);
    for (i = 3; i <= n; i++) {
        int c = a + b;
        push(&s, c);
        printf("%d ", c);
        a = b;
        b = c;
    }
    printf("\n");
}

int main() {
    int choice, num;
    printf("1. Factorial\n2. Fibonacci Series\nEnter choice: ");
    scanf("%d", &choice);
    switch (choice) {
        case 1:
            printf("Enter number: ");
            scanf("%d", &num);
            printf("Factorial of %d is %d\n", num, factorial(num));
            break;
        case 2:
            printf("Enter number of terms: ");
            scanf("%d", &num);
            fibonacci(num);
            break;
        default:
            printf("Invalid choice\n");
    }
```

```
    return 0;
}
```

27. We Fly Anywhere Airlines (WFAA) is considering redesigning their ticket counters for airline passengers. They would like to have two separate waiting lines, one for regular customers and one for frequent flyers. Assuming there is only one ticket agent available to serve all passengers, they would like to determine the average waiting time for both types of passengers using various strategies for taking passengers from the waiting lines. WAP to simulate this situation.

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

#define MAX 100

typedef struct {
    int arrival_time;
    int service_time;
    int start_service_time;
    int finish_time;
} Passenger;

typedef struct {
    Passenger queue[MAX];
    int front, rear;
} Queue;

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

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

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

void enqueue(Queue *q, Passenger p) {
    if (isFull(q)) {
        printf("Queue is full!\n");
        return;
    }
    if (isEmpty(q)) {
        q->front = q->rear = 0;
```

```
    } else {
        q->rear = (q->rear + 1) % MAX;
    }
    q->queue[q->rear] = p;
}

Passenger dequeue(Queue *q) {
    Passenger p = q->queue[q->front];
    if (q->front == q->rear) {
        q->front = q->rear = -1;
    } else {
        q->front = (q->front + 1) % MAX;
    }
    return p;
}

int main() {
    int n_regular, n_frequent, i, time = 0, served = 0;
    Queue regular, frequent;
    Passenger regulars[MAX], frequents[MAX];
    int total_regular_wait = 0, total_frequent_wait = 0;
    int served_regular = 0, served_frequent = 0;

    initQueue(&regular);
    initQueue(&frequent);

    printf("Enter number of regular customers: ");
    scanf("%d", &n_regular);
    printf("Enter arrival and service times for regular customers:\n");
    for (i = 0; i < n_regular; i++) {
        scanf("%d %d", &regulars[i].arrival_time, &regulars[i].service_time);
        regulars[i].start_service_time = -1;
        regulars[i].finish_time = -1;
    }

    printf("Enter number of frequent flyers: ");
    scanf("%d", &n_frequent);
    printf("Enter arrival and service times for frequent flyers:\n");
    for (i = 0; i < n_frequent; i++) {
        scanf("%d %d", &frequents[i].arrival_time,
&frequents[i].service_time);
        frequents[i].start_service_time = -1;
        frequents[i].finish_time = -1;
    }

    int idx_regular = 0, idx_frequent = 0;
    int agent_free_time = 0;
    int total_passengers = n_regular + n_frequent;
```

```
    // Strategy: Serve frequent flyers first if both queues are non-empty
    while (served < total_passengers) {
        // Enqueue arriving passengers at current time
        while (idx_regular < n_regular && regulars[idx_regular].arrival_time
<= time) {
            enqueue(&regular, regulars[idx_regular]);
            idx_regular++;
        }
        while (idx_frequent < n_frequent &&
frequents[idx_frequent].arrival_time <= time) {
            enqueue(&frequent, frequents[idx_frequent]);
            idx_frequent++;
        }

        // If agent is free, serve next passenger
        if (time >= agent_free_time) {
            if (!isEmpty(&frequent)) {
                Passenger p = dequeue(&frequent);
                p.start_service_time = time;
                p.finish_time = time + p.service_time;
                total_frequent_wait += (p.start_service_time -
p.arrival_time);
                agent_free_time = p.finish_time;
                served_frequent++;
                served++;
            } else if (!isEmpty(&regular)) {
                Passenger p = dequeue(&regular);
                p.start_service_time = time;
                p.finish_time = time + p.service_time;
                total_regular_wait += (p.start_service_time - p.arrival_time);
                agent_free_time = p.finish_time;
                served_regular++;
                served++;
            } else {
                // No one to serve, advance time
                time++;
                continue;
            }
        }
        time++;
    }

    printf("\nAverage waiting time for regular customers: %.2f\n",
           served_regular ? (float)total_regular_wait / served_regular : 0.0);
    printf("Average waiting time for frequent flyers: %.2f\n",
           served_frequent ? (float)total_frequent_wait / served_frequent :
0.0);
```

```
    return 0;
}
```

28. An operating system assigns job to print queues based on the number of pages to be printed (1 to 50 pages). You may assume that the system printers are able to print 10 page per minute. Smaller print jobs are printed before larger print jobs and print jobs are processed from a single print queue implemented as a priority queue). The system administrators would like to compare the time required to process a set of print jobs using 1, 2, or 3 system printers. Write a program which simulates processing 100 print jobs of varying lengths using either 1, 2, or 3 printers. Assume that a print request is made every minute and that the number of pages to print varies from 1 to 50 pages. To be fair, you will need to process the same set of print jobs each time you add a printer. The output from your program should indicate the order in which the jobs were received, the order in which they were printed, and the time required to process the set of print jobs. If more than one printer is being used, indicate which printer each job was printed on.
    (Hint: Can implement job queue using suitable queue data structure)

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

#define NUM_JOBS 100
#define MIN_PAGES 1
#define MAX_PAGES 50
#define PAGES_PER_MIN 10
#define MAX_PRINTERS 3

typedef struct {
    int job_id;
    int pages;
    int arrival_time;
} Job;

typedef struct {
    Job jobs[NUM_JOBS];
    int size;
} PriorityQueue;

typedef struct {
    int busy_until;
    int printer_id;
} Printer;

void pq_init(PriorityQueue *pq) {
    pq->size = 0;
}
```

```c
void pq_push(PriorityQueue *pq, Job job) {
    int i = pq->size++;
    while (i > 0 && pq->jobs[i-1].pages > job.pages) {
        pq->jobs[i] = pq->jobs[i-1];
        i--;
    }
    pq->jobs[i] = job;
}

Job pq_pop(PriorityQueue *pq) {
    return pq->jobs[--pq->size];
}

int pq_empty(PriorityQueue *pq) {
    return pq->size == 0;
}

void generate_jobs(Job jobs[NUM_JOBS], unsigned int seed) {
    srand(seed);
    for (int i = 0; i < NUM_JOBS; i++) {
        jobs[i].job_id = i+1;
        jobs[i].pages = (rand() % (MAX_PAGES - MIN_PAGES + 1)) + MIN_PAGES;
        jobs[i].arrival_time = i; // one job per minute
    }
}

void simulate(Job jobs[NUM_JOBS], int num_printers) {
    PriorityQueue pq;
    pq_init(&pq);
    Printer printers[MAX_PRINTERS] = {0};
    int printed_order[NUM_JOBS];
    int printed_on[NUM_JOBS];
    int print_start[NUM_JOBS];
    int print_end[NUM_JOBS];
    int jobs_processed = 0, next_job = 0, time = 0;

    for (int i = 0; i < num_printers; i++) {
        printers[i].busy_until = 0;
        printers[i].printer_id = i+1;
    }

    printf("\n--- Simulation with %d printer(s) ---\n", num_printers);
    printf("Job Arrival Order:\n");
    for (int i = 0; i < NUM_JOBS; i++)
        printf("Job %d: %d pages (arrived at %d)\n", jobs[i].job_id,
jobs[i].pages, jobs[i].arrival_time);
```

```
    while (jobs_processed < NUM_JOBS) {
        // Add jobs that arrive at this minute
        while (next_job < NUM_JOBS && jobs[next_job].arrival_time <= time) {
            pq_push(&pq, jobs[next_job]);
            next_job++;
        }
        // Assign jobs to free printers
        for (int p = 0; p < num_printers; p++) {
            if (printers[p].busy_until <= time && !pq_empty(&pq)) {
                Job job = pq_pop(&pq);
                int duration = (job.pages + PAGES_PER_MIN - 1) /
PAGES_PER_MIN;
                printers[p].busy_until = time + duration;
                printed_order[jobs_processed] = job.job_id;
                printed_on[jobs_processed] = printers[p].printer_id;
                print_start[jobs_processed] = time;
                print_end[jobs_processed] = printers[p].busy_until;
                jobs_processed++;
            }
        }
        time++;
    }

    printf("\nPrint Order:\n");
    for (int i = 0; i < NUM_JOBS; i++) {
        printf("Job %d printed on Printer %d (start: %d, end: %d)\n",
            printed_order[i], printed_on[i], print_start[i], print_end[i]);
    }
    printf("\nTotal time to process all jobs: %d minutes\n", time-1);
}

int main() {
    Job jobs[NUM_JOBS];
    unsigned int seed = (unsigned int)time(NULL);
    generate_jobs(jobs, seed);

    for (int printers = 1; printers <= MAX_PRINTERS; printers++) {
        simulate(jobs, printers);
    }
    return 0;
}
```

29. ☑ Write a program which simulates the operation of a busy airport which has only two runways to handle all takeoffs and landings. You may assume that each takeoff or landing takes 15 minutes to complete. One runway request is made during each five minute time interval and likelihood of landing request is the same as for takeoff. Priority is given to planes requesting a landing. If a request cannot be honored it is added to a takeoff or landing queue. Your program should simulate 120 minutes of activity at the airport. Each

request for runway clearance should be time-stamped and added to the appropriate queue. The output from your program should include the final queue contents, the number of take offs completed, the number of landings completed, and the average number of minutes spent in each queue.

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

#define SIMULATION_TIME 120
#define REQUEST_INTERVAL 5
#define RUNWAY_TIME 15
#define RUNWAYS 2

typedef struct Node {
    int timestamp;
    struct Node* next;
} Node;

typedef struct Queue {
    Node* front;
    Node* rear;
    int count;
    int total_wait_time;
} Queue;

void enqueue(Queue* q, int timestamp) {
    Node* temp = (Node*)malloc(sizeof(Node));
    temp->timestamp = timestamp;
    temp->next = NULL;
    if (q->rear == NULL) {
        q->front = q->rear = temp;
    } else {
        q->rear->next = temp;
        q->rear = temp;
    }
    q->count++;
}

int dequeue(Queue* q, int current_time) {
    if (q->front == NULL) return -1;
    Node* temp = q->front;
    int wait_time = current_time - temp->timestamp;
    q->total_wait_time += wait_time;
    q->front = q->front->next;
    if (q->front == NULL) q->rear = NULL;
    free(temp);
    q->count--;
```

```c
    return wait_time;
}

void print_queue(Queue* q, const char* name) {
    printf("%s queue: ", name);
    Node* temp = q->front;
    while (temp) {
        printf("%d ", temp->timestamp);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    Queue landing = {NULL, NULL, 0, 0};
    Queue takeoff = {NULL, NULL, 0, 0};
    int runway_free[RUNWAYS] = {0, 0}; // time when each runway becomes free
    int landings_completed = 0, takeoffs_completed = 0;
    srand((unsigned int)time(NULL));

    for (int t = 0; t < SIMULATION_TIME; t += REQUEST_INTERVAL) {
        // Generate request: 0 = landing, 1 = takeoff
        int req_type = rand() % 2;
        if (req_type == 0)
            enqueue(&landing, t);
        else
            enqueue(&takeoff, t);

        // Assign runways
        for (int r = 0; r < RUNWAYS; r++) {
            if (runway_free[r] <= t) {
                if (landing.count > 0) {
                    dequeue(&landing, t);
                    runway_free[r] = t + RUNWAY_TIME;
                    landings_completed++;
                } else if (takeoff.count > 0) {
                    dequeue(&takeoff, t);
                    runway_free[r] = t + RUNWAY_TIME;
                    takeoffs_completed++;
                }
            }
        }
    }

    // Process remaining planes in queues after simulation time
    int t = SIMULATION_TIME;
    while (landing.count > 0 || takeoff.count > 0) {
        for (int r = 0; r < RUNWAYS; r++) {
```

```
            if (runway_free[r] <= t) {
                if (landing.count > 0) {
                    dequeue(&landing, t);
                    runway_free[r] = t + RUNWAY_TIME;
                    landings_completed++;
                } else if (takeoff.count > 0) {
                    dequeue(&takeoff, t);
                    runway_free[r] = t + RUNWAY_TIME;
                    takeoffs_completed++;
                }
            }
        }
        t += REQUEST_INTERVAL;
    }

    printf("Final queue contents:\n");
    print_queue(&landing, "Landing");
    print_queue(&takeoff, "Takeoff");
    printf("Number of landings completed: %d\n", landings_completed);
    printf("Number of takeoffs completed: %d\n", takeoffs_completed);
    printf("Average wait time in landing queue: %.2f minutes\n",
           landings_completed ? (float)landing.total_wait_time /
landings_completed : 0.0);
    printf("Average wait time in takeoff queue: %.2f minutes\n",
           takeoffs_completed ? (float)takeoff.total_wait_time /
takeoffs_completed : 0.0);

    // Free any remaining nodes (shouldn't be any)
    while (landing.front) dequeue(&landing, t);
    while (takeoff.front) dequeue(&takeoff, t);

    return 0;
}
```

30. ☑ Write a Program to implement circular double ended queue where user can add and remove the elements from both front and rear of the queue

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 5

typedef struct {
    int arr[MAX];
    int front, rear;
```

```
} Deque;

void init(Deque *dq) {
    dq->front = -1;
    dq->rear = -1;
}

int isFull(Deque *dq) {
    return ((dq->front == 0 && dq->rear == MAX-1) || (dq->front == dq-
>rear+1));
}

int isEmpty(Deque *dq) {
    return (dq->front == -1);
}

void insertFront(Deque *dq, int val) {
    if (isFull(dq)) {
        printf("Deque is full!\n");
        return;
    }
    if (isEmpty(dq)) {
        dq->front = dq->rear = 0;
    } else if (dq->front == 0) {
        dq->front = MAX-1;
    } else {
        dq->front--;
    }
    dq->arr[dq->front] = val;
}

void insertRear(Deque *dq, int val) {
    if (isFull(dq)) {
        printf("Deque is full!\n");
        return;
    }
    if (isEmpty(dq)) {
        dq->front = dq->rear = 0;
    } else if (dq->rear == MAX-1) {
        dq->rear = 0;
    } else {
        dq->rear++;
    }
    dq->arr[dq->rear] = val;
}

void deleteFront(Deque *dq) {
    if (isEmpty(dq)) {
```

```
        printf("Deque is empty!\n");
        return;
    }
    printf("Deleted %d from front\n", dq->arr[dq->front]);
    if (dq->front == dq->rear) {
        dq->front = dq->rear = -1;
    } else if (dq->front == MAX-1) {
        dq->front = 0;
    } else {
        dq->front++;
    }
}

void deleteRear(Deque *dq) {
    if (isEmpty(dq)) {
        printf("Deque is empty!\n");
        return;
    }
    printf("Deleted %d from rear\n", dq->arr[dq->rear]);
    if (dq->front == dq->rear) {
        dq->front = dq->rear = -1;
    } else if (dq->rear == 0) {
        dq->rear = MAX-1;
    } else {
        dq->rear--;
    }
}

void display(Deque *dq) {
    if (isEmpty(dq)) {
        printf("Deque is empty!\n");
        return;
    }
    int i = dq->front;
    printf("Deque: ");
    while (1) {
        printf("%d ", dq->arr[i]);
        if (i == dq->rear)
            break;
        i = (i + 1) % MAX;
    }
    printf("\n");
}

int main() {
    Deque dq;
    init(&dq);
    int choice, val;
```

```
    while (1) {
        printf("\n1.Insert Front\n2.Insert Rear\n3.Delete Front\n4.Delete
Rear\n5.Display\n6.Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &val);
                insertFront(&dq, val);
                break;
            case 2:
                printf("Enter value: ");
                scanf("%d", &val);
                insertRear(&dq, val);
                break;
            case 3:
                deleteFront(&dq);
                break;
            case 4:
                deleteRear(&dq);
                break;
            case 5:
                display(&dq);
                break;
            case 6:
                exit(0);
            default:
                printf("Invalid choice!\n");
        }
    }
    return 0;
}
```

31. ☑ Write a Program to simulate the following situation. Computer is a multitasking device. We need to download some document as well as listen music and play game simultaneously. There is a system queue which decides which task to be done first. Assume that for download application priority is highest and game playing is having lowest priority. After completion of one type of tasks like all download operations then the second queue will be processed. (Hint: Can implement priority queue)

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

#define MAX 100
```

```c
typedef struct {
    char name[50];
    int priority; // 1: Download, 2: Music, 3: Game
} Task;

typedef struct {
    Task tasks[MAX];
    int front, rear;
} Queue;

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

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

int isFull(Queue *q) {
    return q->rear == MAX - 1;
}

void enqueue(Queue *q, Task t) {
    if (isFull(q)) {
        printf("Queue is full!\n");
        return;
    }
    if (isEmpty(q)) {
        q->front = q->rear = 0;
    } else {
        q->rear++;
    }
    q->tasks[q->rear] = t;
}

Task dequeue(Queue *q) {
    Task t = {"", -1};
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return t;
    }
    t = q->tasks[q->front];
    if (q->front == q->rear) {
        q->front = q->rear = -1;
    } else {
        q->front++;
    }
    return t;
```

```c
}

void processQueue(Queue *q, const char *taskType) {
    while (!isEmpty(q)) {
        Task t = dequeue(q);
        printf("Processing %s: %s\n", taskType, t.name);
    }
}

int main() {
    Queue downloadQ, musicQ, gameQ;
    initQueue(&downloadQ);
    initQueue(&musicQ);
    initQueue(&gameQ);

    int n, choice;
    Task t;

    printf("Enter number of tasks: ");
    scanf("%d", &n);
    getchar();

    for (int i = 0; i < n; i++) {
        printf("\nEnter task name: ");
        fgets(t.name, sizeof(t.name), stdin);
        t.name[strcspn(t.name, "\n")] = 0;
        printf("Enter task type (1-Download, 2-Music, 3-Game): ");
        scanf("%d", &t.priority);
        getchar();

        if (t.priority == 1)
            enqueue(&downloadQ, t);
        else if (t.priority == 2)
            enqueue(&musicQ, t);
        else if (t.priority == 3)
            enqueue(&gameQ, t);
        else
            printf("Invalid priority!\n");
    }

    printf("\n--- Processing Download Tasks ---\n");
    processQueue(&downloadQ, "Download");

    printf("\n--- Processing Music Tasks ---\n");
    processQueue(&musicQ, "Music");

    printf("\n--- Processing Game Tasks ---\n");
    processQueue(&gameQ, "Game");
```

```
    return 0;
}
```

32. ☑ Write a Program to implement multiple queue i.e. two queues using array and perform following operations on it. A. Addq, B. Delq, C. Display Queue.

```
#include <stdio.h>
#define MAX 50

int queue1[MAX], front1 = -1, rear1 = -1;
int queue2[MAX], front2 = -1, rear2 = -1;

void addq(int queue[], int *front, int *rear, int val) {
    if (*rear == MAX - 1) {
        printf("Queue Overflow\n");
        return;
    }
    if (*front == -1)
        *front = 0;
    (*rear)++;
    queue[*rear] = val;
}

int delq(int queue[], int *front, int *rear) {
    int val;
    if (*front == -1 || *front > *rear) {
        printf("Queue Underflow\n");
        return -1;
    }
    val = queue[*front];
    (*front)++;
    return val;
}

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

```c
int main() {
    int choice, qno, val, delval;
    while (1) {
        printf("\n1. Addq\n2. Delq\n3. Display Queue\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        if (choice == 4)
            break;
        printf("Enter queue number (1 or 2): ");
        scanf("%d", &qno);
        switch (choice) {
            case 1:
                printf("Enter value to add: ");
                scanf("%d", &val);
                if (qno == 1)
                    addq(queue1, &front1, &rear1, val);
                else if (qno == 2)
                    addq(queue2, &front2, &rear2, val);
                else
                    printf("Invalid queue number\n");
                break;
            case 2:
                if (qno == 1)
                    delval = delq(queue1, &front1, &rear1);
                else if (qno == 2)
                    delval = delq(queue2, &front2, &rear2);
                else {
                    printf("Invalid queue number\n");
                    break;
                }
                if (delval != -1)
                    printf("Deleted value: %d\n", delval);
                break;
            case 3:
                if (qno == 1)
                    display(queue1, front1, rear1);
                else if (qno == 2)
                    display(queue2, front2, rear2);
                else
                    printf("Invalid queue number\n");
                break;
            default:
                printf("Invalid choice\n");
        }
    }
    return 0;
}
```

33. Given a list, split it into two sublists — one for the front half, and one for the back half. If the number of elements is odd, the extra element should go in the front list. So FrontBackSplit() on the list {2, 3, 5, 7, 11} should yield the two lists {2, 3, 5} and {7, 11}. Getting this right for all the cases is harder than it looks. You should check your solution against a few cases (length = 2, length = 3, length=4) to make sure that the list gets split correctly near the short-list boundary conditions. If it works right for length=4, it probably works right for length=1000. You will probably need special case code to deal with the (length <2) cases.

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

// Definition for singly-linked list node
struct Node {
    int data;
    struct Node* next;
};

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

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

// Function to split the list into front and back halves
void FrontBackSplit(struct Node* source, struct Node** frontRef, struct Node**
backRef) {
    struct Node* slow;
    struct Node* fast;
    if (source == NULL || source->next == NULL) {
        // length < 2 cases
        *frontRef = source;
        *backRef = NULL;
        return;
    }
```

```
    slow = source;
    fast = source->next;
    // Advance 'fast' two nodes, and advance 'slow' one node
    while (fast != NULL) {
        fast = fast->next;
        if (fast != NULL) {
            slow = slow->next;
            fast = fast->next;
        }
    }
    // 'slow' is before the midpoint in the list, so split it in two at that
point.
    *frontRef = source;
    *backRef = slow->next;
    slow->next = NULL;
}

// Helper function to push a node at the end
void append(struct Node** headRef, int new_data) {
    struct Node* node = newNode(new_data);
    if (*headRef == NULL) {
        *headRef = node;
        return;
    }
    struct Node* temp = *headRef;
    while (temp->next != NULL)
        temp = temp->next;
    temp->next = node;
}

int main() {
    struct Node* head = NULL;
    struct Node* front = NULL;
    struct Node* back = NULL;
    int n, val, i;

    printf("Enter the number of elements in the list: ");
    scanf("%d", &n);

    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &val);
        append(&head, val);
    }

    printf("Original list: ");
    printList(head);
```

```
    FrontBackSplit(head, &front, &back);

    printf("Front half: ");
    printList(front);

    printf("Back half: ");
    printList(back);

    return 0;
}
```

34. ☑ WAP to perform addition of two polynomials using singly linked list.

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

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

// Function to create a new node
Node* createNode(int coeff, int pow) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->coeff = coeff;
    newNode->pow = pow;
    newNode->next = NULL;
    return newNode;
}

// Function to insert node at the end
void insert(Node** head, int coeff, int pow) {
    Node* newNode = createNode(coeff, pow);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    Node* temp = *head;
    while (temp->next)
        temp = temp->next;
    temp->next = newNode;
}

// Function to display the polynomial
void display(Node* head) {
```

```
    if (!head) {
        printf("0\n");
        return;
    }
    while (head) {
        printf("%dx^%d", head->coeff, head->pow);
        if (head->next && head->next->coeff >= 0)
            printf(" + ");
        head = head->next;
    }
    printf("\n");
}

// Function to add two polynomials
Node* addPoly(Node* poly1, Node* poly2) {
    Node* result = NULL;
    while (poly1 && poly2) {
        if (poly1->pow > poly2->pow) {
            insert(&result, poly1->coeff, poly1->pow);
            poly1 = poly1->next;
        } else if (poly1->pow < poly2->pow) {
            insert(&result, poly2->coeff, poly2->pow);
            poly2 = poly2->next;
        } else {
            insert(&result, poly1->coeff + poly2->coeff, poly1->pow);
            poly1 = poly1->next;
            poly2 = poly2->next;
        }
    }
    while (poly1) {
        insert(&result, poly1->coeff, poly1->pow);
        poly1 = poly1->next;
    }
    while (poly2) {
        insert(&result, poly2->coeff, poly2->pow);
        poly2 = poly2->next;
    }
    return result;
}

int main() {
    Node *poly1 = NULL, *poly2 = NULL, *result = NULL;
    int n, coeff, pow, i;

    printf("Enter number of terms in first polynomial: ");
    scanf("%d", &n);
    printf("Enter terms in descending order of power (coeff pow):\n");
    for (i = 0; i < n; i++) {
```

```
        scanf("%d%d", &coeff, &pow);
        insert(&poly1, coeff, pow);
    }

    printf("Enter number of terms in second polynomial: ");
    scanf("%d", &n);
    printf("Enter terms in descending order of power (coeff pow):\n");
    for (i = 0; i < n; i++) {
        scanf("%d%d", &coeff, &pow);
        insert(&poly2, coeff, pow);
    }

    printf("First Polynomial: ");
    display(poly1);
    printf("Second Polynomial: ");
    display(poly2);

    result = addPoly(poly1, poly2);
    printf("Sum: ");
    display(result);

    // Free memory (optional)
    return 0;
}
```

35. ☑ Write an iterative / noniterative Reverse() function that reverses a list by rearranging all the next pointers and the head pointer. Ideally, Reverse() should only need to make one pass/scan of the list.

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

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

void Push(struct Node** head_ref, int new_data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = *head_ref;
    *head_ref = new_node;
}

void PrintList(struct Node* head) {
    struct Node* temp = head;
```

```c
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

void Reverse(struct Node** head_ref) {
    struct Node* prev = NULL;
    struct Node* current = *head_ref;
    struct Node* next = NULL;
    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
}

int main() {
    struct Node* head = NULL;
    int n, val;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &val);
        Push(&head, val);
    }

    printf("Original list:\n");
    PrintList(head);

    Reverse(&head);

    printf("Reversed list:\n");
    PrintList(head);

    // Free memory
    struct Node* temp;
    while (head != NULL) {
        temp = head;
        head = head->next;
        free(temp);
    }
```

```
    return 0;
}
```

36. ☑ WAP to create doubly linked list and perform following operations on it. A) Insert (all cases) 2. Delete (all cases).

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

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

Node* head = NULL;

// Function to create a new node
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (!newNode) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = data;
    newNode->prev = newNode->next = NULL;
    return newNode;
}

// Insert at the beginning
void insertAtBeginning(int data) {
    Node* newNode = createNode(data);
    if (head == NULL) {
        head = newNode;
    } else {
        newNode->next = head;
        head->prev = newNode;
        head = newNode;
    }
    printf("Inserted %d at beginning\n", data);
}

// Insert at the end
void insertAtEnd(int data) {
    Node* newNode = createNode(data);
    if (head == NULL) {
```

```
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next)
            temp = temp->next;
        temp->next = newNode;
        newNode->prev = temp;
    }
    printf("Inserted %d at end\n", data);
}

// Insert after a given position (1-based index)
void insertAtPosition(int data, int pos) {
    if (pos <= 1 || head == NULL) {
        insertAtBeginning(data);
        return;
    }
    Node* temp = head;
    int i = 1;
    while (temp->next && i < pos - 1) {
        temp = temp->next;
        i++;
    }
    if (temp->next == NULL) {
        insertAtEnd(data);
        return;
    }
    Node* newNode = createNode(data);
    newNode->next = temp->next;
    newNode->prev = temp;
    temp->next->prev = newNode;
    temp->next = newNode;
    printf("Inserted %d at position %d\n", data, pos);
}

// Delete from beginning
void deleteFromBeginning() {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    Node* temp = head;
    head = head->next;
    if (head)
        head->prev = NULL;
    printf("Deleted %d from beginning\n", temp->data);
    free(temp);
}
```

```c
// Delete from end
void deleteFromEnd() {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    Node* temp = head;
    if (temp->next == NULL) {
        printf("Deleted %d from end\n", temp->data);
        free(temp);
        head = NULL;
        return;
    }
    while (temp->next)
        temp = temp->next;
    temp->prev->next = NULL;
    printf("Deleted %d from end\n", temp->data);
    free(temp);
}

// Delete from a given position (1-based index)
void deleteFromPosition(int pos) {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    if (pos <= 1) {
        deleteFromBeginning();
        return;
    }
    Node* temp = head;
    int i = 1;
    while (temp && i < pos) {
        temp = temp->next;
        i++;
    }
    if (temp == NULL) {
        printf("Position out of range\n");
        return;
    }
    if (temp->next)
        temp->next->prev = temp->prev;
    if (temp->prev)
        temp->prev->next = temp->next;
    printf("Deleted %d from position %d\n", temp->data, pos);
    free(temp);
}
```

```c
// Display the list
void display() {
    Node* temp = head;
    printf("List: ");
    while (temp) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    int choice, data, pos;
    while (1) {
        printf("\n1. Insert at beginning\n2. Insert at end\n3. Insert at
position\n");
        printf("4. Delete from beginning\n5. Delete from end\n6. Delete from
position\n7. Display\n8. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter data: ");
                scanf("%d", &data);
                insertAtBeginning(data);
                break;
            case 2:
                printf("Enter data: ");
                scanf("%d", &data);
                insertAtEnd(data);
                break;
            case 3:
                printf("Enter data and position: ");
                scanf("%d%d", &data, &pos);
                insertAtPosition(data, pos);
                break;
            case 4:
                deleteFromBeginning();
                break;
            case 5:
                deleteFromEnd();
                break;
            case 6:
                printf("Enter position: ");
                scanf("%d", &pos);
                deleteFromPosition(pos);
                break;
```

```
            case 7:
                display();
                break;
            case 8:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
    }
    return 0;
}
```

37. ☑ WAP to merge two sorted Doubly linked lists and display their result.

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

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

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

// Function to insert node at the end
Node* insertEnd(Node* head, int data) {
    Node* newNode = createNode(data);
    if (!head)
        return newNode;
    Node* temp = head;
    while (temp->next)
        temp = temp->next;
    temp->next = newNode;
    newNode->prev = temp;
    return head;
}

// Function to display the list
```

```c
void display(Node* head) {
    Node* temp = head;
    while (temp) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Function to merge two sorted doubly linked lists
Node* mergeSortedLists(Node* head1, Node* head2) {
    Node* merged = NULL;
    Node* tail = NULL;

    while (head1 && head2) {
        Node* temp;
        if (head1->data <= head2->data) {
            temp = head1;
            head1 = head1->next;
        } else {
            temp = head2;
            head2 = head2->next;
        }
        temp->prev = temp->next = NULL;
        if (!merged) {
            merged = tail = temp;
        } else {
            tail->next = temp;
            temp->prev = tail;
            tail = temp;
        }
    }
    while (head1) {
        Node* temp = head1;
        head1 = head1->next;
        temp->prev = temp->next = NULL;
        if (!merged) {
            merged = tail = temp;
        } else {
            tail->next = temp;
            temp->prev = tail;
            tail = temp;
        }
    }
    while (head2) {
        Node* temp = head2;
        head2 = head2->next;
        temp->prev = temp->next = NULL;
```

```
        if (!merged) {
            merged = tail = temp;
        } else {
            tail->next = temp;
            temp->prev = tail;
            tail = temp;
        }
    }
    return merged;
}

int main() {
    Node* head1 = NULL;
    Node* head2 = NULL;
    int n1, n2, val, i;

    printf("Enter number of elements in first sorted list: ");
    scanf("%d", &n1);
    printf("Enter elements of first sorted list:\n");
    for (i = 0; i < n1; i++) {
        scanf("%d", &val);
        head1 = insertEnd(head1, val);
    }

    printf("Enter number of elements in second sorted list: ");
    scanf("%d", &n2);
    printf("Enter elements of second sorted list:\n");
    for (i = 0; i < n2; i++) {
        scanf("%d", &val);
        head2 = insertEnd(head2, val);
    }

    printf("First List: ");
    display(head1);
    printf("Second List: ");
    display(head2);

    Node* merged = mergeSortedLists(head1, head2);
    printf("Merged Sorted List: ");
    display(merged);

    return 0;
}
```

38. Write a Program to create Inorder Threaded Binary Tree and Traverse it in Postorder way.

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

typedef struct Node {
    int data;
    struct Node *left, *right;
    int lthread, rthread;
} Node;

Node* createNode(int data) {
    Node* temp = (Node*)malloc(sizeof(Node));
    temp->data = data;
    temp->left = temp->right = NULL;
    temp->lthread = temp->rthread = 1;
    return temp;
}

Node* insert(Node* root, int key) {
    Node *ptr = root, *par = NULL;
    while (ptr != NULL) {
        if (key == ptr->data) {
            printf("Duplicate Key!\n");
            return root;
        }
        par = ptr;
        if (key < ptr->data) {
            if (ptr->lthread == 0)
                ptr = ptr->left;
            else
                break;
        } else {
            if (ptr->rthread == 0)
                ptr = ptr->right;
            else
                break;
        }
    }
    Node* tmp = createNode(key);
    if (par == NULL) {
        root = tmp;
    } else if (key < par->data) {
        tmp->left = par->left;
        tmp->right = par;
        par->lthread = 0;
        par->left = tmp;
    } else {
        tmp->left = par;
        tmp->right = par->right;
```

```
        par->rthread = 0;
        par->right = tmp;
    }
    return root;
}

Node* leftMost(Node* n) {
    if (n == NULL) return NULL;
    while (n->lthread == 0)
        n = n->left;
    return n;
}

Node* rightMost(Node* n) {
    if (n == NULL) return NULL;
    while (n->rthread == 0)
        n = n->right;
    return n;
}

// Helper function to find parent in threaded binary tree
Node* findParent(Node* root, Node* child) {
    if (root == NULL || root == child)
        return NULL;
    Node* parent = NULL;
    Node* curr = root;
    while (curr != NULL && curr != child) {
        parent = curr;
        if (child->data < curr->data) {
            if (curr->lthread == 0)
                curr = curr->left;
            else
                break;
        } else {
            if (curr->rthread == 0)
                curr = curr->right;
            else
                break;
        }
    }
    if (curr == child)
        return parent;
    return NULL;
}

// Postorder traversal using parent finding (not efficient, but works for
threaded trees)
void postorder(Node* root) {
```

```
    if (root == NULL) return;
    Node* curr = leftMost(root);
    Node* last = NULL;
    Node* stack[100];
    int top = -1;
    curr = root;
    while (curr != NULL || top != -1) {
        while (curr != NULL) {
            stack[++top] = curr;
            if (curr->lthread == 0)
                curr = curr->left;
            else
                break;
        }
        while (top != -1 && (stack[top]->rthread == 1 || stack[top]->right ==
last)) {
            last = stack[top--];
            printf("%d ", last->data);
        }
        if (top != -1)
            curr = stack[top]->right;
        else
            curr = NULL;
    }
}

int main() {
    Node* root = NULL;
    int n, val;
    printf("Enter number of nodes: ");
    scanf("%d", &n);
    printf("Enter node values:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &val);
        root = insert(root, val);
    }
    printf("Postorder Traversal: ");
    postorder(root);
    printf("\n");
    return 0;
}
```

39. Implement following problems based on Arrays. A. a. i/p- { 2,2,3,4} o/p: 1 b. i/p: {1,3,2,5,4,9} o/p: 3 c. i/p: { 1,5,3,4,6,2,12} o/p: 5 B. Find saddle point in the Matrix. C. Magic square Matrix. (Check) D. Represent given matrix in its Sparse form.

```c
#include <stdio.h>

// A. Find the smallest missing positive integer in an array
int smallest_missing_positive(int arr[], int n) {
    int present[n+1];
    for (int i = 0; i <= n; i++) present[i] = 0;
    for (int i = 0; i < n; i++) {
        if (arr[i] > 0 && arr[i] <= n)
            present[arr[i]] = 1;
    }
    for (int i = 1; i <= n; i++) {
        if (!present[i]) return i;
    }
    return n+1;
}

// B. Find saddle point in the matrix
void find_saddle_point(int mat[10][10], int r, int c) {
    int found = 0;
    for (int i = 0; i < r; i++) {
        int min_row = mat[i][0], col_idx = 0;
        for (int j = 1; j < c; j++) {
            if (mat[i][j] < min_row) {
                min_row = mat[i][j];
                col_idx = j;
            }
        }
        int k;
        for (k = 0; k < r; k++) {
            if (mat[k][col_idx] > min_row)
                break;
        }
        if (k == r) {
            printf("Saddle point at (%d,%d): %d\n", i, col_idx, min_row);
            found = 1;
        }
    }
    if (!found)
        printf("No saddle point found.\n");
}

// C. Check if a matrix is a magic square
int is_magic_square(int mat[10][10], int n) {
    int sum = 0, diag1 = 0, diag2 = 0;
    for (int j = 0; j < n; j++)
        sum += mat[0][j];
    for (int i = 0; i < n; i++) {
        int row = 0, col = 0;
```

```
        for (int j = 0; j < n; j++) {
            row += mat[i][j];
            col += mat[j][i];
        }
        if (row != sum || col != sum)
            return 0;
        diag1 += mat[i][i];
        diag2 += mat[i][n-i-1];
    }
    return (diag1 == sum && diag2 == sum);
}

// D. Represent matrix in sparse form
void sparse_form(int mat[10][10], int r, int c) {
    int count = 0;
    for (int i = 0; i < r; i++)
        for (int j = 0; j < c; j++)
            if (mat[i][j] != 0)
                count++;
    printf("Sparse matrix representation:\n");
    printf("Row Col Value\n");
    for (int i = 0; i < r; i++)
        for (int j = 0; j < c; j++)
            if (mat[i][j] != 0)
                printf("%d   %d   %d\n", i, j, mat[i][j]);
}

int main() {
    // A. Test cases
    int arr1[] = {2,2,3,4};
    int arr2[] = {1,3,2,5,4,9};
    int arr3[] = {1,5,3,4,6,2,12};
    printf("A.a Output: %d\n", smallest_missing_positive(arr1, 4));
    printf("A.b Output: %d\n", smallest_missing_positive(arr2, 6));
    printf("A.c Output: %d\n", smallest_missing_positive(arr3, 7));

    // B. Saddle point
    int mat1[3][3] = {{1,2,3},{4,5,6},{7,8,9}};
    printf("\nB. Saddle point:\n");
    find_saddle_point(mat1, 3, 3);

    // C. Magic square
    int magic[3][3] = {{2,7,6},{9,5,1},{4,3,8}};
    printf("\nC. Magic Square: ");
    if (is_magic_square(magic, 3))
        printf("Yes\n");
    else
        printf("No\n");
```

```
    // D. Sparse form
    int mat2[3][3] = {{0,0,3},{4,0,0},{0,8,0}};
    printf("\nD. Sparse form:\n");
    sparse_form(mat2, 3, 3);

    return 0;
}
```

40. WAP to implement Insertion sort, Merge Sort, Bubble Sort and Quick Sort on 1D array of Student structure (contains student_name, student_roll_no, total_marks), with key as student_roll_no. and count the number of swap performed.

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

#define MAX 100

typedef struct {
    char student_name[50];
    int student_roll_no;
    int total_marks;
} Student;

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

// Insertion Sort
void insertionSort(Student arr[], int n, int *swap_count) {
    for (int i = 1; i < n; i++) {
        Student key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j].student_roll_no > key.student_roll_no) {
            arr[j + 1] = arr[j];
            (*swap_count)++;
            j--;
        }
        arr[j + 1] = key;
        if (j + 1 != i) (*swap_count)++;
    }
}
```

```c
// Bubble Sort
void bubbleSort(Student arr[], int n, int *swap_count) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j].student_roll_no > arr[j+1].student_roll_no) {
                swap(&arr[j], &arr[j+1], swap_count);
            }
        }
    }
}

// Merge function for Merge Sort
void merge(Student arr[], int l, int m, int r, int *swap_count) {
    int n1 = m - l + 1;
    int n2 = r - m;
    Student L[n1], R[n2];
    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i].student_roll_no <= R[j].student_roll_no) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
            (*swap_count)++;
        }
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

// Merge Sort
void mergeSort(Student arr[], int l, int r, int *swap_count) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m, swap_count);
        mergeSort(arr, m + 1, r, swap_count);
        merge(arr, l, m, r, swap_count);
    }
}

// Partition function for Quick Sort
int partition(Student arr[], int low, int high, int *swap_count) {
    int pivot = arr[high].student_roll_no;
    int i = low - 1;
    for (int j = low; j < high; j++) {
```

```
        if (arr[j].student_roll_no < pivot) {
            i++;
            swap(&arr[i], &arr[j], swap_count);
        }
    }
    swap(&arr[i + 1], &arr[high], swap_count);
    return i + 1;
}

// Quick Sort
void quickSort(Student arr[], int low, int high, int *swap_count) {
    if (low < high) {
        int pi = partition(arr, low, high, swap_count);
        quickSort(arr, low, pi - 1, swap_count);
        quickSort(arr, pi + 1, high, swap_count);
    }
}

// Function to print array
void printArray(Student arr[], int n) {
    printf("Name\tRoll No\tMarks\n");
    for (int i = 0; i < n; i++) {
        printf("%s\t%d\t%d\n", arr[i].student_name, arr[i].student_roll_no,
arr[i].total_marks);
    }
}

int main() {
    Student arr[MAX];
    int n, choice, swap_count;
    printf("Enter number of students: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        printf("Enter name, roll no, total marks for student %d: ", i + 1);
        scanf("%s %d %d", arr[i].student_name, &arr[i].student_roll_no,
&arr[i].total_marks);
    }
    do {
        printf("\n1. Insertion Sort\n2. Bubble Sort\n3. Merge Sort\n4. Quick
Sort\n5. Exit\nEnter choice: ");
        scanf("%d", &choice);
        Student temp[MAX];
        memcpy(temp, arr, n * sizeof(Student));
        swap_count = 0;
        switch (choice) {
            case 1:
                insertionSort(temp, n, &swap_count);
                printf("Sorted array (Insertion Sort):\n");
```

```
                printArray(temp, n);
                printf("Number of swaps: %d\n", swap_count);
                break;
            case 2:
                bubbleSort(temp, n, &swap_count);
                printf("Sorted array (Bubble Sort):\n");
                printArray(temp, n);
                printf("Number of swaps: %d\n", swap_count);
                break;
            case 3:
                mergeSort(temp, 0, n - 1, &swap_count);
                printf("Sorted array (Merge Sort):\n");
                printArray(temp, n);
                printf("Number of swaps: %d\n", swap_count);
                break;
            case 4:
                quickSort(temp, 0, n - 1, &swap_count);
                printf("Sorted array (Quick Sort):\n");
                printArray(temp, n);
                printf("Number of swaps: %d\n", swap_count);
                break;
            case 5:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice!\n");
        }
    } while (choice != 5);
    return 0;
}
```

41. Implement a Polynomial addition and multiplication using Linked Lists.

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

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

// Function to create a new node
PolyNode* createNode(int coeff, int pow) {
    PolyNode* node = (PolyNode*)malloc(sizeof(PolyNode));
    node->coeff = coeff;
```

```
    node->pow = pow;
    node->next = NULL;
    return node;
}

// Function to insert a node at the end
void insertEnd(PolyNode** head, int coeff, int pow) {
    PolyNode* node = createNode(coeff, pow);
    if (*head == NULL) {
        *head = node;
        return;
    }
    PolyNode* temp = *head;
    while (temp->next)
        temp = temp->next;
    temp->next = node;
}

// Function to display the polynomial
void displayPoly(PolyNode* head) {
    if (!head) {
        printf("0\n");
        return;
    }
    while (head) {
        printf("%dx^%d", head->coeff, head->pow);
        if (head->next && head->next->coeff >= 0)
            printf(" + ");
        else if (head->next)
            printf(" ");
        head = head->next;
    }
    printf("\n");
}

// Function to add two polynomials
PolyNode* addPoly(PolyNode* p1, PolyNode* p2) {
    PolyNode* res = NULL;
    while (p1 && p2) {
        if (p1->pow > p2->pow) {
            insertEnd(&res, p1->coeff, p1->pow);
            p1 = p1->next;
        } else if (p1->pow < p2->pow) {
            insertEnd(&res, p2->coeff, p2->pow);
            p2 = p2->next;
        } else {
            insertEnd(&res, p1->coeff + p2->coeff, p1->pow);
            p1 = p1->next;
```

```
            p2 = p2->next;
        }
    }
    while (p1) {
        insertEnd(&res, p1->coeff, p1->pow);
        p1 = p1->next;
    }
    while (p2) {
        insertEnd(&res, p2->coeff, p2->pow);
        p2 = p2->next;
    }
    return res;
}

// Function to multiply two polynomials
PolyNode* multiplyPoly(PolyNode* p1, PolyNode* p2) {
    PolyNode* res = NULL;
    for (PolyNode* t1 = p1; t1; t1 = t1->next) {
        for (PolyNode* t2 = p2; t2; t2 = t2->next) {
            int coeff = t1->coeff * t2->coeff;
            int pow = t1->pow + t2->pow;
            // Add to result, combine like terms
            PolyNode* temp = res, *prev = NULL;
            while (temp && temp->pow > pow) {
                prev = temp;
                temp = temp->next;
            }
            if (temp && temp->pow == pow) {
                temp->coeff += coeff;
            } else {
                PolyNode* newNode = createNode(coeff, pow);
                if (prev == NULL) {
                    newNode->next = res;
                    res = newNode;
                } else {
                    newNode->next = prev->next;
                    prev->next = newNode;
                }
            }
        }
    }
    return res;
}

// Function to free the polynomial linked list
void freePoly(PolyNode* head) {
    while (head) {
        PolyNode* temp = head;
```

```
        head = head->next;
        free(temp);
    }
}

// Example usage
int main() {
    PolyNode *poly1 = NULL, *poly2 = NULL, *sum = NULL, *prod = NULL;
    int n, coeff, pow, i;

    printf("Enter number of terms in first polynomial: ");
    scanf("%d", &n);
    printf("Enter coeff and power for each term (in descending order of
power):\n");
    for (i = 0; i < n; i++) {
        scanf("%d%d", &coeff, &pow);
        insertEnd(&poly1, coeff, pow);
    }

    printf("Enter number of terms in second polynomial: ");
    scanf("%d", &n);
    printf("Enter coeff and power for each term (in descending order of
power):\n");
    for (i = 0; i < n; i++) {
        scanf("%d%d", &coeff, &pow);
        insertEnd(&poly2, coeff, pow);
    }

    printf("First Polynomial: ");
    displayPoly(poly1);
    printf("Second Polynomial: ");
    displayPoly(poly2);

    sum = addPoly(poly1, poly2);
    printf("Sum: ");
    displayPoly(sum);

    prod = multiplyPoly(poly1, poly2);
    printf("Product: ");
    displayPoly(prod);

    freePoly(poly1);
    freePoly(poly2);
    freePoly(sum);
    freePoly(prod);

    return 0;
}
```

42. Write a program to detect a cycle in an undirected graph using DFS.

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

#define MAX 100

int adj[MAX][MAX];
int visited[MAX];
int n;

void createGraph() {
    int edges, i, u, v;
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter number of edges: ");
    scanf("%d", &edges);
    for (i = 0; i < edges; i++) {
        printf("Enter edge %d (u v): ", i + 1);
        scanf("%d %d", &u, &v);
        adj[u][v] = 1;
        adj[v][u] = 1; // undirected graph
    }
}

int dfs(int v, int parent) {
    visited[v] = 1;
    for (int i = 0; i < n; i++) {
        if (adj[v][i]) {
            if (!visited[i]) {
                if (dfs(i, v))
                    return 1;
            } else if (i != parent) {
                return 1; // cycle detected
            }
        }
    }
    return 0;
}

int main() {
    createGraph();
    for (int i = 0; i < n; i++)
        visited[i] = 0;
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
```

```
            if (dfs(i, -1)) {
                printf("Cycle detected in the graph.\n");
                return 0;
            }
        }
    }
    printf("No cycle detected in the graph.\n");
    return 0;
}
```