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.**

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

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

typedef struct Queue {
    Node *arr[100];
    int front, rear;
} Queue;

void enqueue(Queue *q, Node *node) {
    q->arr[q->rear++] = node;
}
Node* dequeue(Queue *q) {
    if (q->front == q->rear) return NULL;
    return q->arr[q->front++];
}
int isEmpty(Queue *q) {
    return q->front == q->rear;
}

typedef struct Stack {
    Node *arr[100];
    int top;
} Stack;
```

```
void push(Stack *s, Node *node) {
    s->arr[++s->top] = node;
}
Node* pop(Stack *s) {
    return s->top == -1 ? NULL : s->arr[s->top--];
}
int isStackEmpty(Stack *s) {
    return s->top == -1;
}

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

// Build tree from user input (level order)
Node* buildTree() {
    int val;
    printf("Enter root node value: ");
    scanf("%d", &val);
    if (val == -1) return NULL;

    Node *root = newNode(val);
    Queue q = {.front = 0, .rear = 0};
    enqueue(&q, root);

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

        printf("Enter left child of %d (-1 for no child): ", curr->data);
        scanf("%d", &val);
```

```
        if (val != -1) {
            curr->left = newNode(val);
            enqueue(&q, curr->left);
        }

        printf("Enter right child of %d (-1 for no child): ", curr->data);
        scanf("%d", &val);
        if (val != -1) {
            curr->right = newNode(val);
            enqueue(&q, curr->right);
        }
    }
    return root;
}

// Preorder traversal (non-recursive)
void preorder(Node *root) {
    Stack s; s.top = -1;
    if (!root) return;
    push(&s, root);
    printf("Preorder: ");
    while (!isStackEmpty(&s)) {
        Node *curr = pop(&s);
        printf("%d ", curr->data);
        if (curr->right) push(&s, curr->right);
        if (curr->left) push(&s, curr->left);
    }
    printf("\n");
}

// Postorder traversal (non-recursive)
void postorder(Node *root) {
    Stack s1, s2;
```

```
    s1.top = s2.top = -1;
    if (!root) return;
    push(&s1, root);
    while (!isStackEmpty(&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 (!isStackEmpty(&s2)) {
        printf("%d ", pop(&s2)->data);
    }
    printf("\n");
}

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

// Height of tree (non-recursive, level order)
int height(Node *root) {
```

```
    if (!root) return 0;
    Node *queue[100];
    int front = 0, rear = 0;
    queue[rear++] = root;
    int h = 0;
    while (front < rear) {
        int size = rear - front;
        h++;
        for (int i = 0; i < size; i++) {
            Node *curr = queue[front++];
            if (curr->left) queue[rear++] = curr->left;
            if (curr->right) queue[rear++] = curr->right;
        }
    }
    return h;
}

int main() {
    Node *root = buildTree();
    preorder(root);
    postorder(root);
    printf("Total nodes: %d\n", countNodes(root));
    printf("Height of tree: %d\n", height(root));
    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.

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

```
#include <stdlib.h>

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

typedef struct Queue {
    Node *arr[100];
    int front, rear;
} Queue;

void enqueue(Queue *q, Node *node) {
    q->arr[q->rear++] = node;
}
Node* dequeue(Queue *q) {
    if (q->front == q->rear) return NULL;
    return q->arr[q->front++];
}
int isEmpty(Queue *q) {
    return q->front == q->rear;
}

typedef struct Stack {
    Node *arr[100];
    int top;
} Stack;

void push(Stack *s, Node *node) {
    s->arr[++s->top] = node;
}
Node* pop(Stack *s) {
    return s->top == -1 ? NULL : s->arr[s->top--];
```

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

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

// Build tree from user input (level order)
Node* buildTree() {
    int val;
    printf("Enter root node value: ");
    scanf("%d", &val);
    if (val == -1) return NULL;

    Node *root = newNode(val);
    Queue q = {.front = 0, .rear = 0};
    enqueue(&q, root);

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

        printf("Enter left child of %d (-1 for no child): ", curr->data);
        scanf("%d", &val);
        if (val != -1) {
            curr->left = newNode(val);
            enqueue(&q, curr->left);
        }
```

```
        printf("Enter right child of %d (-1 for no child): ", curr->data);
        scanf("%d", &val);
        if (val != -1) {
            curr->right = newNode(val);
            enqueue(&q, curr->right);
        }
    }
    return root;
}

// Non-recursive inorder traversal using stack
void inorder(Node *root) {
    Stack s; s.top = -1;
    Node *curr = root;
    printf("Inorder: ");
    while (curr != NULL || !isStackEmpty(&s)) {
        while (curr != NULL) {
            push(&s, curr);
            curr = curr->left;
        }
        curr = pop(&s);
        printf("%d ", curr->data);
        curr = curr->right;
    }
    printf("\n");
}

// Count number of nodes on longest path (which is height of tree)
int height(Node *root) {
    if (!root) return 0;
    Node *queue[100];
    int front = 0, rear = 0;
    queue[rear++] = root;
```

```
    int h = 0;
    while (front < rear) {
        int size = rear - front;
        h++;
        for (int i = 0; i < size; i++) {
            Node *curr = queue[front++];
            if (curr->left) queue[rear++] = curr->left;
            if (curr->right) queue[rear++] = curr->right;
        }
    }
    return h;
}

// Display tree levelwise (level order traversal)
void levelOrder(Node *root) {
    if (!root) return;
    Queue q = {.front = 0, .rear = 0};
    enqueue(&q, root);
    printf("Level-wise: ");
    while (!isEmpty(&q)) {
        Node *curr = dequeue(&q);
        printf("%d ", curr->data);
        if (curr->left) enqueue(&q, curr->left);
        if (curr->right) enqueue(&q, curr->right);
    }
    printf("\n");
}

int main() {
    Node *root = buildTree();

    inorder(root);
    printf("Number of nodes on longest path: %d\n", height(root));
```

```
    levelOrder(root);
    printf("Height of tree: %d\n", height(root));

    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>

#define MAX 100

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

// Create new node
Node* newNode(int key) {
    Node* n = (Node*)malloc(sizeof(Node));
    n->key = key;
    n->left = n->right = NULL;
    return n;
}
```

```
// Insert key in BST (recursive for simplicity)
Node* insert(Node* root, int key) {
    if (!root) return newNode(key);
    if (key < root->key) root->left = insert(root->left, key);
    else if (key > root->key) root->right = insert(root->right, key);
    return root;
}

// Level-wise display using simple array queue
void levelDisplay(Node* root) {
    if (!root) {
        printf("Tree is empty\n");
        return;
    }

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

    printf("Level-wise: ");
    while (front < rear) {
        Node* curr = queue[front++];
        printf("%d ", curr->key);

        if (curr->left) queue[rear++] = curr->left;
        if (curr->right) queue[rear++] = curr->right;
    }
    printf("\n");
}

// Mirror image using stack with array
void mirror(Node* root) {
    if (!root) return;
```

```
    Node* stack[MAX];
    int top = -1;
    stack[++top] = root;

    while (top >= 0) {
        Node* curr = stack[top--];

        // Swap children
        Node* temp = curr->left;
        curr->left = curr->right;
        curr->right = temp;

        if (curr->left) stack[++top] = curr->left;
        if (curr->right) stack[++top] = curr->right;
    }
}

// Height calculation using level-wise traversal (BFS)
int height(Node* root) {
    if (!root) return 0;

    Node* queue[MAX];
    int front = 0, rear = 0;
    queue[rear++] = root;
    int height = 0;

    while (1) {
        int count = rear - front;
        if (count == 0) break;
        height++;

        while (count--) {
```

```
            Node* curr = queue[front++];
            if (curr->left) queue[rear++] = curr->left;
            if (curr->right) queue[rear++] = curr->right;
        }
    }
    return height;
}

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

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

    printf("Enter %d keys:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &key);
        root = insert(root, key);
    }

    levelDisplay(root);

    printf("Height of tree: %d\n", height(root));

    mirror(root);

    printf("After mirror image:\n");
    levelDisplay(root);

    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;
    struct Node* right;
} Node;

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

// Insert key in BST
Node* insert(Node* root, int key) {
    if (!root) return newNode(key);
    if (key < root->key)
        root->left = insert(root->left, key);
    else if (key > root->key)
        root->right = insert(root->right, key);
    else
        printf("Key %d already exists!\n", key);
    return root;
}
```

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

// Delete key from BST
Node* deleteNode(Node* root, int key) {
    if (!root) {
        printf("Key %d not found.\n", key);
        return root;
    }

    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
    else {
        // node with one or no child
        if (!root->left) {
            Node* temp = root->right;
            free(root);
            printf("Deleted key %d.\n", key);
            return temp;
        } else if (!root->right) {
            Node* temp = root->left;
            free(root);
            printf("Deleted key %d.\n", key);
            return temp;
        }

        // node with two children: get inorder successor (min in right subtree)
```

```
        Node* temp = minValueNode(root->right);
        root->key = temp->key;  // copy inorder successor's key
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}

// Find a key
int find(Node* root, int key) {
    if (!root) return 0;
    if (key == root->key) return 1;
    if (key < root->key) return find(root->left, key);
    return find(root->right, key);
}

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

int main() {
    Node* root = NULL;
    int choice, key;

    while (1) {
        printf("\n--- BST Operations Menu ---\n");
        printf("1. Insert\n2. Delete\n3. Find\n4. Display Inorder\n5. Exit\n");
        printf("Enter 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("BST Inorder: ");
            inorder(root);
            printf("\n");
            break;

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

        default:
            printf("Invalid choice, try again.\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>

#define MAX 100

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

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

// Non-recursive Insert
Node* insert(Node* root, int key) {
    Node* newnode = createNode(key);
    if (root == NULL) return newnode;
```

```
    Node* parent = NULL;
    Node* curr = root;

    while (curr != NULL) {
        parent = curr;
        if (key < curr->key)
            curr = curr->left;
        else if (key > curr->key)
            curr = curr->right;
        else {
            printf("Key %d already exists!\n", key);
            free(newnode);
            return root;
        }
    }

    if (key < parent->key)
        parent->left = newnode;
    else
        parent->right = newnode;

    return root;
}

// Non-recursive Find
int find(Node* root, int key) {
    while (root != NULL) {
        if (key == root->key)
            return 1;
        else if (key < root->key)
            root = root->left;
        else
```

```
            root = root->right;
        }
        return 0;
    }

    // Non-recursive Mirror using stack
    void mirrorImage(Node* root) {
        if (root == NULL) return;

        Node* stack[MAX];
        int top = -1;
        stack[++top] = root;

        while (top >= 0) {
            Node* curr = stack[top--];

            // Swap children
            Node* temp = curr->left;
            curr->left = curr->right;
            curr->right = temp;

            if (curr->left)
                stack[++top] = curr->left;
            if (curr->right)
                stack[++top] = curr->right;
        }

        printf("Mirror image created successfully.\n");
    }

    // Non-recursive Postorder using 2 stacks
    void postorder(Node* root) {
        if (!root) return;
```

```
    Node* s1[MAX], *s2[MAX];
    int top1 = -1, top2 = -1;

    s1[++top1] = root;

    while (top1 >= 0) {
        Node* node = s1[top1--];
        s2[++top2] = node;

        if (node->left) s1[++top1] = node->left;
        if (node->right) s1[++top1] = node->right;
    }

    printf("Postorder traversal: ");
    while (top2 >= 0) {
        printf("%d ", s2[top2--]->key);
    }
    printf("\n");
}

// Main menu
int main() {
    Node* root = NULL;
    int choice, key;

    while (1) {
        printf("\n--- BST Non-Recursive Operations Menu ---\n");
        printf("1. Insert\n2. Mirror Image\n3. Find\n4. Postorder\n5. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
```

```
        case 1:
            printf("Enter key to insert: ");
            scanf("%d", &key);
            root = insert(root, key);
            break;

        case 2:
            mirrorImage(root);
            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:
            postorder(root);
            break;

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

        default:
            printf("Invalid choice. Try again.\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>

#define MAX 100

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

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

// Non-recursive insert into BST
Node* insert(Node* root, int key) {
    Node* newNode = createNode(key);
    if (root == NULL) return newNode;
```

```
    Node *curr = root, *parent = NULL;

    while (curr != NULL) {
        parent = curr;
        if (key < curr->key)
            curr = curr->left;
        else if (key > curr->key)
            curr = curr->right;
        else {
            printf("Key already exists!\n");
            free(newNode);
            return root;
        }
    }

    if (key < parent->key)
        parent->left = newNode;
    else
        parent->right = newNode;

    return root;
}

// Non-recursive Inorder traversal
void inorder(Node* root) {
    Node* stack[MAX];
    int top = -1;
    Node* curr = root;

    printf("Inorder: ");
    while (curr != NULL || top != -1) {
        while (curr != NULL) {
```

```
            stack[++top] = curr;
            curr = curr->left;
        }
        curr = stack[top--];
        printf("%d ", curr->key);
        curr = curr->right;
    }
    printf("\n");
}

// Non-recursive Preorder traversal
void preorder(Node* root) {
    if (root == NULL) return;

    Node* stack[MAX];
    int top = -1;
    stack[++top] = root;

    printf("Preorder: ");
    while (top != -1) {
        Node* curr = stack[top--];
        printf("%d ", curr->key);

        if (curr->right)
            stack[++top] = curr->right;
        if (curr->left)
            stack[++top] = curr->left;
    }
    printf("\n");
}

// Non-recursive count of leaf nodes
int countLeafNodes(Node* root) {
```

```
    if (root == NULL) return 0;

    Node* stack[MAX];
    int top = -1, count = 0;
    stack[++top] = root;

    while (top != -1) {
        Node* curr = stack[top--];
        if (curr->left == NULL && curr->right == NULL)
            count++;

        if (curr->right)
            stack[++top] = curr->right;
        if (curr->left)
            stack[++top] = curr->left;
    }
    return count;
}

// Non-recursive mirror image
void mirrorImage(Node* root) {
    if (root == NULL) return;

    Node* stack[MAX];
    int top = -1;
    stack[++top] = root;

    while (top != -1) {
        Node* curr = stack[top--];

        // Swap left and right
        Node* temp = curr->left;
        curr->left = curr->right;
```

```
        curr->right = temp;

        if (curr->left) stack[++top] = curr->left;
        if (curr->right) stack[++top] = curr->right;
    }

    printf("Mirror image created.\n");
}

// Main menu
int main() {
    Node* root = NULL;
    int choice, key;

    while (1) {
        printf("\n--- BST Non-Recursive Operations Menu ---\n");
          printf("1. Insert\n2. Preorder Traversal\n3. Inorder Traversal\n4. Count Leaf Nodes\n5.
Mirror Image\n6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter key to insert: ");
                scanf("%d", &key);
                root = insert(root, key);
                break;

            case 2:
                preorder(root);
                break;

            case 3:
```

```
            inorder(root);
            break;

        case 4:
            printf("Number of leaf nodes: %d\n", countLeafNodes(root));
            break;

        case 5:
            mirrorImage(root);
            break;

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

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

   return 0;
}
```

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>
```

```c
#include <stdlib.h>

#define MAX 100

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

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

// Non-recursive insert
Node* insert(Node* root, int key) {
    Node *curr = root, *parent = NULL;
    Node* n = createNode(key);
    if (!root) return n;

    while (curr) {
        parent = curr;
        curr = (key < curr->key) ? curr->left : curr->right;
    }
    (key < parent->key) ? (parent->left = n) : (parent->right = n);
    return root;
}

// Non-recursive Preorder
void preorder(Node* root) {
    if (!root) return;
    Node* stack[MAX]; int top = -1;
```

```
    stack[++top] = root;
    printf("Preorder: ");
    while (top != -1) {
        Node* n = stack[top--];
        printf("%d ", n->key);
        if (n->right) stack[++top] = n->right;
        if (n->left)  stack[++top] = n->left;
    }
    printf("\n");
}

// Non-recursive Postorder using two stacks
void postorder(Node* root) {
    if (!root) return;
    Node *s1[MAX], *s2[MAX]; int top1 = -1, top2 = -1;
    s1[++top1] = root;
    while (top1 != -1) {
        Node* n = s1[top1--];
        s2[++top2] = n;
        if (n->left)  s1[++top1] = n->left;
        if (n->right) s1[++top1] = n->right;
    }
    printf("Postorder: ");
    while (top2 != -1) printf("%d ", s2[top2--]->key);
    printf("\n");
}

// Count total nodes (non-recursive)
int countNodes(Node* root) {
    if (!root) return 0;
    Node* stack[MAX]; int top = -1, count = 0;
    stack[++top] = root;
    while (top != -1) {
```

```
        Node* n = stack[top--];
        count++;
        if (n->right) stack[++top] = n->right;
        if (n->left)  stack[++top] = n->left;
    }
    return count;
}

// Display only leaf nodes (non-recursive)
void printLeafNodes(Node* root) {
    if (!root) return;
    Node* stack[MAX]; int top = -1;
    int count = 0;
    stack[++top] = root;
    printf("Leaf nodes: ");
    while (top != -1) {
        Node* n = stack[top--];
        if (!n->left && !n->right) printf("%d ", n->key);
        count++;
         printf("count is %d ", count);
        if (n->right) stack[++top] = n->right;
        if (n->left)  stack[++top] = n->left;
    }
    printf("\n");
}

// Main Menu
int main() {
    Node* root = NULL;
    int ch, key;
    while (1) {
        printf("\n1.Insert 2.Preorder 3.Postorder 4.Total Nodes 5.Leaf Nodes 6.Exit\nEnter: ");
        scanf("%d", &ch);
```

```
        switch (ch) {
            case 1: printf("Key: "); scanf("%d", &key); root = insert(root, key); break;
            case 2: preorder(root); break;
            case 3: postorder(root); break;
            case 4: printf("Total nodes: %d\n", countNodes(root)); break;
            case 5: printLeafNodes(root); break;
            case 6: return 0;
            default: printf("Invalid!\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>
#define MAX 100

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

// Create a new node
Node* createNode(int key) {
    Node* n = malloc(sizeof(Node));
    n->key = key;
    n->left = n->right = NULL;
```

```
    return n;
}

// Insert (non-recursive)
Node* insert(Node* root, int key) {
    Node *curr = root, *parent = NULL;
    Node* n = createNode(key);
    if (!root) return n;

    while (curr) {
        parent = curr;
        if (key < curr->key) curr = curr->left;
        else curr = curr->right;
    }
    if (key < parent->key) parent->left = n;
    else parent->right = n;
    return root;
}

// Find minimum node (used for replacing in two-child case)
Node* findMin(Node* node) {
    while (node && node->left) node = node->left;
    return node;
}

// Delete node (non-recursive)
Node* deleteNode(Node* root, int key) {
    Node *curr = root, *parent = NULL;

    // Search for node and parent
    while (curr && curr->key != key) {
        parent = curr;
        if (key < curr->key) curr = curr->left;
        else curr = curr->right;
    }
```

```
    if (!curr) {
        printf("Key not found.\n");
        return root;
    }

    // Case 1 & 2: Node with 0 or 1 child
    if (!curr->left || !curr->right) {
        Node* child = curr->left ? curr->left : curr->right;
        if (!parent) return child; // Deleting root
        if (parent->left == curr) parent->left = child;
        else parent->right = child;
        free(curr);
    }
    // Case 3: Node with 2 children
    else {
        Node *succParent = curr, *succ = curr->right;
        while (succ->left) {
            succParent = succ;
            succ = succ->left;
        }
        curr->key = succ->key;  // Replace
        // Delete successor
        if (succParent->left == succ)
            succParent->left = succ->right;
        else
            succParent->right = succ->right;
        free(succ);
    }

    return root;
}

// Non-recursive postorder using 2 stacks
void postorder(Node* root) {
    if (!root) return;
```

```
    Node *s1[MAX], *s2[MAX];
    int top1 = -1, top2 = -1;
    s1[++top1] = root;

    while (top1 != -1) {
        Node* n = s1[top1--];
        s2[++top2] = n;
        if (n->left) s1[++top1] = n->left;
        if (n->right) s1[++top1] = n->right;
    }

    printf("Postorder: ");
    while (top2 != -1) printf("%d ", s2[top2--]->key);
    printf("\n");
}

// Non-recursive inorder display
void inorder(Node* root) {
    Node* stack[MAX];
    int top = -1;
    Node* curr = root;
    printf("Inorder (sorted): ");
    while (curr || top != -1) {
        while (curr) {
            stack[++top] = curr;
            curr = curr->left;
        }
        curr = stack[top--];
        printf("%d ", curr->key);
        curr = curr->right;
    }
    printf("\n");
}

// Main menu
```

```
int main() {
    Node* root = NULL;
    int ch, key;

    while (1) {
        printf("\n1.Insert 2.Delete 3.Postorder 4.Display(Inorder) 5.Exit\nEnter choice: ");
        scanf("%d", &ch);
        switch (ch) {
            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:
                postorder(root);
                break;
            case 4:
                inorder(root);
                break;
            case 5:
                return 0;
            default:
                printf("Invalid choice!\n");
        }
    }
}
```

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>

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

// Queue for level order traversal
typedef struct Queue {
    Node* data[100];
    int front, rear;
} Queue;

void initQueue(Queue* q) {
    q->front = q->rear = 0;
}

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

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

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

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

// Recursive insert
Node* insert(Node* root, int key) {
    if (!root) return createNode(key);
    if (key < root->key)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);
    return root;
}

// Find min node (used for delete)
Node* findMin(Node* root) {
    while (root && root->left)
        root = root->left;
    return root;
}

// Recursive delete
Node* deleteNode(Node* root, int key) {
```

```
    if (!root) return NULL;

    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) {
            Node* temp = root->right;
            free(root);
            return temp;
        }
        if (!root->right) {
            Node* temp = root->left;
            free(root);
            return temp;
        }
        Node* temp = findMin(root->right);
        root->key = temp->key;
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}

// Level-wise display, each level in new line
void levelOrderLineByLine(Node* root) {
    if (!root) {
        printf("Tree is empty.\n");
        return;
    }
```

```
    Queue q;
    initQueue(&q);
    enqueue(&q, root);
    enqueue(&q, NULL); // Level marker

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

        if (curr == NULL) {
            printf("\n");
            if (!isEmpty(&q))
                enqueue(&q, NULL);
        } else {
            printf("%d ", curr->key);
            if (curr->left)
                enqueue(&q, curr->left);
            if (curr->right)
                enqueue(&q, curr->right);
        }
    }
}

// MAIN
int main() {
    Node* root = NULL;
    int ch, key;

    while (1) {
        printf("\n1.Insert  2.Delete  3.Levelwise Display (Line by Line)  4.Exit\nEnter choice: ");
        scanf("%d", &ch);
```

```
        switch (ch) {
            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("Levelwise display (each level on new line):\n");
                levelOrderLineByLine(root);
                break;
            case 4:
                return 0;
            default:
                printf("Invalid choice!\n");
        }
    }
}
```

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 key;
    struct Node *left, *right;
} Node;

// Queue for BFS (used for mirror and height)
typedef struct Queue {
    Node* data[100];
    int front, rear;
} Queue;

void initQueue(Queue* q) {
    q->front = q->rear = 0;
}

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

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

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

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

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

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

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

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

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

// Insert into BST (non-recursive)
Node* insert(Node* root, int key) {
    Node* newNode = createNode(key);
    if (!root) return newNode;

    Node* current = root;
    Node* parent = NULL;

    while (current) {
        parent = current;
        if (key < current->key)
```

```
            current = current->left;
        else
            current = current->right;
    }

    if (key < parent->key)
        parent->left = newNode;
    else
        parent->right = newNode;

    return root;
}

// Inorder traversal (non-recursive)
void inorder(Node* root) {
    Stack s;
    initStack(&s);
    Node* curr = root;

    while (curr || !isStackEmpty(&s)) {
        while (curr) {
            push(&s, curr);
            curr = curr->left;
        }
        curr = pop(&s);
        printf("%d ", curr->key);
        curr = curr->right;
    }
}

// Height of tree (non-recursive)
int getHeight(Node* root) {
    if (!root) return 0;

    Queue q;
```

```
    initQueue(&q);
    enqueue(&q, root);
    int height = 0;

    while (1) {
        int nodeCount = q.rear - q.front;
        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--;
        }
    }
}

// Mirror the tree (non-recursive)
void mirror(Node* root) {
    if (!root) return;

    Queue q;
    initQueue(&q);
    enqueue(&q, root);

    while (!isEmpty(&q)) {
        Node* node = dequeue(&q);
        Node* temp = node->left;
        node->left = node->right;
        node->right = temp;

        if (node->left) enqueue(&q, node->left);
        if (node->right) enqueue(&q, node->right);
```

```
    }
}

// Copy tree (non-recursive, returns new root)
Node* copyTree(Node* root) {
    if (!root) return NULL;

    Queue q1, q2;
    initQueue(&q1);
    initQueue(&q2);

    Node* newRoot = createNode(root->key);
    enqueue(&q1, root);
    enqueue(&q2, newRoot);

    while (!isEmpty(&q1)) {
        Node* orig = dequeue(&q1);
        Node* copy = dequeue(&q2);

        if (orig->left) {
            copy->left = createNode(orig->left->key);
            enqueue(&q1, orig->left);
            enqueue(&q2, copy->left);
        }
        if (orig->right) {
            copy->right = createNode(orig->right->key);
            enqueue(&q1, orig->right);
            enqueue(&q2, copy->right);
        }
    }

    return newRoot;
}

// MAIN
```

```
int main() {
    Node* root = NULL;
    int choice, key;

    while (1) {
            printf("\n1.Insert\n2.Inorder Display\n3.Mirror Original\n4.Mirror Copy (without disturbing original)\n5.Height of Tree\n6.Exit\nEnter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter key: ");
                scanf("%d", &key);
                root = insert(root, key);
                break;

            case 2:
                printf("Inorder: ");
                inorder(root);
                printf("\n");
                break;

            case 3:
                mirror(root);
                printf("Original tree mirrored.\n");
                break;

            case 4: {
                Node* copy = copyTree(root);
                mirror(copy);
                printf("Inorder of mirrored copy: ");
                inorder(copy);
                printf("\nOriginal tree remains: ");
                inorder(root);
                printf("\n");
```

```
            break;
        }

        case 5:
            printf("Height of tree = %d\n", getHeight(root));
            break;

        case 6:
            exit(0);

        default:
            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>

    typedef struct Employee {
        int emp_id;
        char name[50];
        struct Employee *left, *right;
    } Employee;

    // Create new node
```

```
Employee* create(int id, char* name) {
    Employee* e = (Employee*)malloc(sizeof(Employee));
    e->emp_id = id;
    strcpy(e->name, name);
    e->left = e->right = NULL;
    return e;
}

// Insert into BST
Employee* insert(Employee* root, int id, char* name) {
    if (!root) return create(id, name);
    if (id < root->emp_id)
        root->left = insert(root->left, id, name);
    else if (id > root->emp_id)
        root->right = insert(root->right, id, name);
    return root;
}

// Search employee by id
Employee* search(Employee* root, int id) {
    while (root && root->emp_id != id) {
        if (id < root->emp_id) root = root->left;
        else root = root->right;
    }
    return root;
}

// Inorder traversal (sorted ascending)
void inorder(Employee* root) {
    if (!root) return;
    inorder(root->left);
    printf("ID: %d, Name: %s\n", root->emp_id, root->name);
    inorder(root->right);
```

```
}

int main() {
    Employee* root = NULL;
    int choice, id;
    char name[50];
    Employee* found;

    while (1) {
        printf("\n1.Insert 2.Search 3.Display Sorted 4.Exit\nEnter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter emp-id and name: ");
                scanf("%d %s", &id, name);
                root = insert(root, id, name);
                break;
            case 2:
                printf("Enter emp-id to search: ");
                scanf("%d", &id);
                found = search(root, id);
                if (found)
                    printf("Found: ID=%d, Name=%s\n", found->emp_id, found->name);
                else
                    printf("Employee not found.\n");
                break;
            case 3:
                printf("Employees sorted by emp-id:\n");
                inorder(root);
                break;
            case 4:
                exit(0);
```

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

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

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

// Node structure for Inorder Threaded Binary Tree
typedef struct Node {
    int data;
    struct Node *left, *right;
    int lthread, rthread;  // 1 if thread, 0 if child
} Node;

// Utility to create a new node
Node* createNode(int data) {
    Node* temp = (Node*)malloc(sizeof(Node));
    temp->data = data;
    temp->left = temp->right = NULL;
    temp->lthread = temp->rthread = 1; // Initially threads (to NULL)
    return temp;
}

// Insert node in BST and maintain inorder threads
Node* insert(Node* root, int key) {
    if (root == NULL)
```

```
    return createNode(key);

Node* curr = root;
Node* parent = NULL;

while (curr != NULL) {
    if (key == curr->data) {
        printf("Duplicate key %d not allowed.\n", key);
        return root;
    }
    parent = curr;
    if (key < curr->data) {
        if (curr->lthread == 0)
            curr = curr->left;
        else
            break;
    } else {
        if (curr->rthread == 0)
            curr = curr->right;
        else
            break;
    }
}

Node* newNode = createNode(key);

if (key < parent->data) {
    newNode->left = parent->left;
    newNode->right = parent;
    parent->lthread = 0;
    parent->left = newNode;
} else {
    newNode->right = parent->right;
```

```
        newNode->left = parent;
        parent->rthread = 0;
        parent->right = newNode;
    }
    return root;
}

// Preorder traversal of Inorder Threaded Binary Tree
void preorder(Node* root) {
    if (root == NULL) return;

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

        if (curr->lthread == 0) {
            curr = curr->left;
        } 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("How many nodes you want to insert? ");
    scanf("%d", &n);
```

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

    printf("Preorder traversal of Inorder Threaded Binary Tree:\n");
    preorder(root);

    return 0;
}
```

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

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

// Node structure for Inorder Threaded Binary Tree
typedef struct Node {
    int data;
    struct Node *left, *right;
    int lthread, rthread; // 1 if thread, 0 if child
} Node;

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

```
    temp->lthread = temp->rthread = 1; // Initially threads
    return temp;
}

// Insert node in BST and maintain inorder threads
Node* insert(Node* root, int key) {
    if (root == NULL)
        return createNode(key);

    Node* curr = root;
    Node* parent = NULL;

    while (curr != NULL) {
        if (key == curr->data) {
            printf("Duplicate key %d not allowed.\n", key);
            return root;
        }
        parent = curr;
        if (key < curr->data) {
            if (curr->lthread == 0)
                curr = curr->left;
            else
                break;
        } else {
            if (curr->rthread == 0)
                curr = curr->right;
            else
                break;
        }
    }

    Node* newNode = createNode(key);
```

```
    if (key < parent->data) {
        newNode->left = parent->left;
        newNode->right = parent;
        parent->lthread = 0;
        parent->left = newNode;
    } else {
        newNode->right = parent->right;
        newNode->left = parent;
        parent->rthread = 0;
        parent->right = newNode;
    }
    return root;
}

// Find leftmost node (starting point for inorder traversal)
Node* leftMost(Node* node) {
    if (node == NULL)
        return NULL;
    while (node->lthread == 0)
        node = node->left;
    return node;
}

// Inorder traversal using threads
void inorder(Node* root) {
    if (root == NULL) return;

    Node* curr = leftMost(root);

    while (curr != NULL) {
        printf("%d ", curr->data);

        // If right thread exists, follow it
```

```
        if (curr->rthread == 1)
            curr = curr->right;
        else
            // Else go to the leftmost node in right subtree
            curr = leftMost(curr->right);
    }
}

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

    printf("How many nodes you want to insert? ");
    scanf("%d", &n);

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

    printf("Inorder traversal of Inorder Threaded Binary Tree:\n");
    inorder(root);

    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>
```

```
// AVL Tree Node
typedef struct Node {
    int key;
    int height;
    struct Node* left;
    struct Node* right;
} Node;

// Helper functions
int height(Node* node) {
    return node ? node->height : 0;
}

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

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

int getBalance(Node* node) {
    return node ? height(node->left) - height(node->right) : 0;
}

// Rotations
```

```
Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;

    x->right = y;
    y->left = T2;

    y->height = 1 + max(height(y->left), height(y->right));
    x->height = 1 + max(height(x->left), height(x->right));

    return x;
}

Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;

    y->left = x;
    x->right = T2;

    x->height = 1 + max(height(x->left), height(x->right));
    y->height = 1 + max(height(y->left), height(y->right));

    return y;
}

// Insert function
Node* insert(Node* root, int key) {
    if (!root)
        return createNode(key);
```

```
    if (key < root->key)
        root->left = insert(root->left, key);
    else if (key > root->key)
        root->right = insert(root->right, key);
    else
        return root;  // no duplicates

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

    // Balancing
    if (balance > 1 && key < root->left->key)
        return rightRotate(root);          // LL
    if (balance < -1 && key > root->right->key)
        return leftRotate(root);           // RR
    if (balance > 1 && key > root->left->key) { // LR
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }
    if (balance < -1 && key < root->right->key) { // RL
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

// Find minimum node in subtree
Node* minValueNode(Node* node) {
    Node* current = node;
    while (current->left)
```

```
        current = current->left;
    return current;
}

// Delete function
Node* deleteNode(Node* root, int key) {
    if (!root) return NULL;

    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
    else {
        // Found node to be deleted
        if (!root->left || !root->right) {
            Node* temp = root->left ? root->left : root->right;
            free(root);
            return temp;
        } else {
            Node* temp = minValueNode(root->right);
            root->key = temp->key;
            root->right = deleteNode(root->right, temp->key);
        }
    }

    // Update height
    root->height = 1 + max(height(root->left), height(root->right));
    int balance = getBalance(root);

    // Rebalance
    if (balance > 1 && getBalance(root->left) >= 0)
```

```
        return rightRotate(root);              // LL
    if (balance > 1 && getBalance(root->left) < 0) { // LR
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }
    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);               // RR
    if (balance < -1 && getBalance(root->right) > 0) { // RL
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

// Search function
int search(Node* root, int key) {
    if (!root)
        return 0;
    if (key == root->key)
        return 1;
    if (key < root->key)
        return search(root->left, key);
    return search(root->right, key);
}

// Level-order display
void levelOrder(Node* root) {
    if (!root) return;

    Node* queue[1000];
```

```
    int front = 0, rear = 0;
    queue[rear++] = root;

    while (front < rear) {
        int count = rear - front;

        for (int i = 0; i < count; i++) {
            Node* curr = queue[front++];
            printf("%d ", curr->key);
            if (curr->left)
                queue[rear++] = curr->left;
            if (curr->right)
                queue[rear++] = curr->right;
        }
        printf("\n");
    }
}

// Free memory
void freeTree(Node* root) {
    if (!root) return;
    freeTree(root->left);
    freeTree(root->right);
    free(root);
}

// Main Menu
int main() {
    Node* root = NULL;
    int choice, val;
```

```
    printf("====== AVL Tree Menu ======\n");
    while (1) {
                printf("\n1. Insert\n2. Delete\n3. Search\n4. Display Level-wise\n5. Exit\nChoose: ");
        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", &val);
                root = deleteNode(root, val);
                break;
            case 3:
                printf("Enter value to search: ");
                scanf("%d", &val);
                if (search(root, val))
                    printf("Found!\n");
                else
                    printf("Not found!\n");
                break;
            case 4:
                printf("Level-wise AVL Tree:\n");
                levelOrder(root);
                break;
            case 5:
                freeTree(root);
```

```
            printf("Exiting...\n");
            return 0;
        default:
            printf("Invalid choice!\n");
    }
  }
}
```

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

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

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

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

// BFS function
void BFS(int start) {
    int queue[MAX], front = 0, rear = 0;
    visited[start] = 1;
    queue[rear++] = start;
```

```
    while (front < rear) {
        int v = queue[front++];
        printf("%d ", v);
        for (int i = 0; i < n; i++) {
            if (graph[v][i] && !visited[i]) {
                visited[i] = 1;
                queue[rear++] = i;
            }
        }
    }
}

int main() {
    int directed;

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

    // Check if the graph is directed
    printf("Is the graph directed (1) or undirected (0)? ");
    scanf("%d", &directed);

    // Input adjacency matrix
    printf("Enter the adjacency matrix (%d x %d):\n", n, n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
            // If graph is undirected, mirror the input to ensure symmetry
            if (!directed && graph[i][j])
                graph[j][i] = 1;
        }
    }
```

```
    // DFS Traversal
    printf("\nDFS Traversal: ");
    for (int i = 0; i < n; i++) visited[i] = 0;
    DFS(0);

    // BFS Traversal
    printf("\nBFS Traversal: ");
    for (int i = 0; i < n; i++) visited[i] = 0;
    BFS(0);

    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 100

// Node structure for adjacency list
typedef struct Node {
    int vertex;
    struct Node* next;
} Node;

Node* adjList[MAX];  // Adjacency list
int visited[MAX];    // Visited array for DFS and BFS
int n;               // Number of vertices

// Add edge to the adjacency list
void addEdge(int u, int v, int isDirected) {
    Node* newNode = (Node*)malloc(sizeof(Node));
```

```
    newNode->vertex = v;
    newNode->next = adjList[u];
    adjList[u] = newNode;

    if (!isDirected) {
        Node* newNode2 = (Node*)malloc(sizeof(Node));
        newNode2->vertex = u;
        newNode2->next = adjList[v];
        adjList[v] = newNode2;
    }
}

// DFS function
void DFS(int v) {
    visited[v] = 1;
    printf("%d ", v);
    for (Node* temp = adjList[v]; temp; temp = temp->next)
        if (!visited[temp->vertex])
            DFS(temp->vertex);
}

// BFS function
void BFS(int start) {
    int queue[MAX], front = 0, rear = 0;
    visited[start] = 1;
    queue[rear++] = start;

    while (front < rear) {
        int v = queue[front++];
        printf("%d ", v);
        for (Node* temp = adjList[v]; temp; temp = temp->next) {
            if (!visited[temp->vertex]) {
                visited[temp->vertex] = 1;
                queue[rear++] = temp->vertex;
            }
```

```
        }
    }
}

// Main function
int main() {
    int edges, u, v, isDirected;
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter number of edges: ");
    scanf("%d", &edges);
    printf("Is the graph directed (1) or undirected (0)? ");
    scanf("%d", &isDirected);

    for (int i = 0; i < edges; i++) {
        printf("Enter edge (u v): ");
        scanf("%d %d", &u, &v);
        addEdge(u, v, isDirected);
    }

    printf("\nDFS Traversal: ");
    for (int i = 0; i < n; i++) visited[i] = 0;
    DFS(0);

    printf("\nBFS Traversal: ");
    for (int i = 0; i < n; i++) visited[i] = 0;
    BFS(0);

    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>
#include <stdlib.h>
#include <limits.h>

#define MAX 100

int minKey(int key[], int mstSet[], int n) {
    int min = INT_MAX, minIndex = -1;
    for (int v = 0; v < n; v++) {
        if (!mstSet[v] && key[v] < min) {
            min = key[v];
            minIndex = v;
        }
    }
    return minIndex;
}

void primsMatrix(int graph[MAX][MAX], int n) {
    int parent[MAX], key[MAX], mstSet[MAX];

    for (int i = 0; i < n; i++) {
        key[i] = INT_MAX;
        mstSet[i] = 0;
    }

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

    for (int count = 0; count < n - 1; count++) {
        int u = minKey(key, mstSet, n);
```

```
        mstSet[u] = 1;

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

    printf("Edge \tWeight\n");
    for (int i = 1; i < n; i++)
        printf("%d - %d\t%d\n", parent[i], i, graph[i][parent[i]]);
}


struct Node {
    int dest, weight;
    struct Node* next;
};

struct Node* createNode(int dest, int weight) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->dest = dest;
    newNode->weight = weight;
    newNode->next = NULL;
    return newNode;
}

void primsList(struct Node* adjList[], int n) {
    int key[MAX], parent[MAX], mstSet[MAX];

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

```
        key[i] = INT_MAX;
        mstSet[i] = 0;
    }

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

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

        struct Node* temp = adjList[u];
        while (temp) {
            int v = temp->dest;
            int weight = temp->weight;

            if (!mstSet[v] && weight < key[v]) {
                key[v] = weight;
                parent[v] = u;
            }
            temp = temp->next;
        }
    }

    printf("Edge \tWeight\n");
    for (int i = 1; i < n; i++)
        printf("%d - %d\t%d\n", parent[i], i, key[i]);
}

int main() {
    int choice, n;
    printf("Choose Graph Representation:\n");
    printf("1. Adjacency Matrix\n");
```

```
printf("2. Adjacency List\n");
printf("Enter your choice: ");
scanf("%d", &choice);

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

if (choice == 1) {
    int graph[MAX][MAX];
    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]);

    printf("\nMinimum Spanning Tree using Prim's (Matrix):\n");
    primsMatrix(graph, n);

} else if (choice == 2) {
    struct Node* adjList[MAX];
    for (int i = 0; i < n; i++)
        adjList[i] = NULL;

    int edges, u, v, w;
    printf("Enter number of edges: ");
    scanf("%d", &edges);
    printf("Enter edges (src dest weight):\n");
    for (int i = 0; i < edges; i++) {
        scanf("%d %d %d", &u, &v, &w);
        struct Node* newNode = createNode(v, w);
        newNode->next = adjList[u];
        adjList[u] = newNode;

        newNode = createNode(u, w); // undirected
```

```
            newNode->next = adjList[v];
            adjList[v] = newNode;
        }

        printf("\nMinimum Spanning Tree using Prim's (List):\n");
        primsList(adjList, n);

    } else {
        printf("Invalid choice.\n");
    }

    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 <stdlib.h>
#include <limits.h>

#define MAX 100
#define INF 99999

// Structure for adjacency list node
typedef struct Node {
    int vertex, weight;
    struct Node* next;
```

```
} Node;

// Add node to adjacency list
Node* addNode(Node* head, int v, int w) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->vertex = v;
    newNode->weight = w;
    newNode->next = head;
    return newNode;
}

// Get vertex with minimum distance
int minDistance(int dist[], int visited[], int V) {
    int min = INF, index = -1;
    for (int i = 0; i < V; i++) {
        if (!visited[i] && dist[i] < min) {
            min = dist[i];
            index = i;
        }
    }
    return index;
}

// Dijkstra using adjacency matrix
void dijkstraMatrix(int graph[MAX][MAX], int V, int src) {
    int dist[V], visited[V];
    for (int i = 0; i < V; i++) {
        dist[i] = INF;
        visited[i] = 0;
    }
    dist[src] = 0;

    for (int count = 0; count < V - 1; count++) {
```

```
        int u = minDistance(dist, visited, V);
        if (u == -1) break; // No reachable vertex
        visited[u] = 1;

        for (int v = 0; v < V; v++) {
            if (!visited[v] && graph[u][v] && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
        }
    }

    printf("Vertex\tDistance from Source %d\n", src);
    for (int i = 0; i < V; i++)
        printf("%d\t%d\n", i, dist[i]);
}

// Dijkstra using adjacency list
void dijkstraList(Node* adjList[], int V, int src) {
    int dist[V], visited[V];
    for (int i = 0; i < V; i++) {
        dist[i] = INF;
        visited[i] = 0;
    }
    dist[src] = 0;

    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, visited, V);
        if (u == -1) break; // No reachable vertex
        visited[u] = 1;

        Node* temp = adjList[u];
        while (temp != NULL) {
            int v = temp->vertex;
            int w = temp->weight;
```

```
            if (!visited[v] && dist[u] + w < dist[v])
                dist[v] = dist[u] + w;
            temp = temp->next;
        }
    }

    printf("Vertex\tDistance from Source %d\n", src);
    for (int i = 0; i < V; i++)
        printf("%d\t%d\n", i, dist[i]);
}

// Optional: Debug print adjacency list
void printAdjList(Node* adjList[], int V) {
    printf("\nAdjacency List:\n");
    for (int i = 0; i < V; i++) {
        printf("%d: ", i);
        Node* temp = adjList[i];
        while (temp != NULL) {
            printf(" -> (%d, w=%d)", temp->vertex, temp->weight);
            temp = temp->next;
        }
        printf("\n");
    }
}

// ----------- MAIN -------------
int main() {
    int choice, V, E, src;
    int graph[MAX][MAX] = {0};
    Node* adjList[MAX];

    // Initialize all adjacency list heads to NULL
    for (int i = 0; i < MAX; i++)
```

```
    adjList[i] = NULL;

printf("Choose Graph Representation:\n");
printf("1. Adjacency Matrix\n");
printf("2. Adjacency List\n");
printf("Enter your choice: ");
scanf("%d", &choice);

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

if (choice == 1) {
    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]);
        }
    }

    printf("Enter source vertex: ");
    scanf("%d", &src);
    dijkstraMatrix(graph, V, src);

} else if (choice == 2) {
    printf("Enter number of edges: ");
    scanf("%d", &E);

    printf("Enter each edge in format (u v weight):\n");
    for (int i = 0; i < E; i++) {
        int u, v, w;
        scanf("%d %d %d", &u, &v, &w);
        if (u >= V || v >= V) {
            printf("Invalid edge (%d, %d). Skipping.\n", u, v);
```

```
            continue;
        }
        adjList[u] = addNode(adjList[u], v, w);
        adjList[v] = addNode(adjList[v], u, w); // For undirected graph
    }

    // Optional: Show adjacency list
    // printAdjList(adjList, V);

    printf("Enter source vertex: ");
    scanf("%d", &src);
    dijkstraList(adjList, V, src);

  } else {
    printf("Invalid choice!\n");
  }

  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.

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

#define MAX 100

// Structure to represent an edge
typedef struct {
  int src, dest, weight;
} Edge;
```

```
// Disjoint set (Union-Find) with path compression and union by rank
int parent[MAX], rank[MAX];

void makeSet(int n) {
    for (int i = 0; i < n; i++) {
        parent[i] = i;
        rank[i] = 0;
    }
}

int findSet(int x) {
    if (parent[x] != x)
        parent[x] = findSet(parent[x]); // Path compression
    return parent[x];
}

void unionSet(int x, int y) {
    int rootX = findSet(x);
    int rootY = findSet(y);
    if (rootX != rootY) {
        if (rank[rootX] < rank[rootY])
            parent[rootX] = rootY;
        else if (rank[rootX] > rank[rootY])
            parent[rootY] = rootX;
        else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }
    }
}

// Compare function for qsort
```

```
int compare(const void* a, const void* b) {
    return ((Edge*)a)->weight - ((Edge*)b)->weight;
}

// Kruskal’s algorithm
void kruskal(Edge edges[], int V, int E) {
    makeSet(V);
    qsort(edges, E, sizeof(Edge), compare);

    int totalWeight = 0;
    printf("Edge\tWeight\n");

    for (int i = 0, count = 0; i < E && count < V - 1; i++) {
        int u = edges[i].src;
        int v = edges[i].dest;
        int setU = findSet(u);
        int setV = findSet(v);

        if (setU != setV) {
            printf("%d - %d\t%d\n", u, v, edges[i].weight);
            totalWeight += edges[i].weight;
            unionSet(setU, setV);
            count++;
        }
    }

    printf("Total weight of MST = %d\n", totalWeight);
}

// ------------ MAIN ------------
int main() {
    int choice, V, E = 0;
    Edge edges[MAX];
```

```
printf("Choose Graph Representation:\n");
printf("1. Adjacency Matrix\n");
printf("2. Adjacency List\n");
printf("Enter your choice: ");
scanf("%d", &choice);

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

if (choice == 1) {
    int matrix[MAX][MAX];
    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", &matrix[i][j]);
        }
    }

    // Collect edges from upper triangle
    for (int i = 0; i < V; i++) {
        for (int j = i + 1; j < V; j++) {
            if (matrix[i][j]) {
                edges[E++] = (Edge){i, j, matrix[i][j]};
            }
        }
    }

    printf("\nMinimum Spanning Tree using Kruskal's (Matrix):\n");
    kruskal(edges, V, E);

} else if (choice == 2) {
    printf("Enter number of edges: ");
```

```
        scanf("%d", &E);
        printf("Enter edges in format (src dest weight):\n");

        for (int i = 0; i < E; i++) {
            scanf("%d %d %d", &edges[i].src, &edges[i].dest, &edges[i].weight);
        }

        printf("\nMinimum Spanning Tree using Kruskal's (List):\n");
        kruskal(edges, V, E);

    } else {
        printf("Invalid choice!\n");
    }

    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>

typedef struct {
    char name[30];
    int roll;
    int marks;
} Student;

void swap(Student *a, Student *b, int *count) {
    Student t = *a; *a = *b; *b = t;
    (*count)++;
}

// Heapify for Heap Sort
void heapify(Student arr[], int n, int i, int *count) {
```

```
    int largest = i, left = 2*i + 1, right = 2*i + 2;
    if (left < n && arr[left].roll > arr[largest].roll) largest = left;
    if (right < n && arr[right].roll > arr[largest].roll) largest = right;
    if (largest != i) {
        swap(&arr[i], &arr[largest], count);
        heapify(arr, n, largest, count);
    }
}

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

// Partition for Quick Sort (first element as pivot)
int partition(Student arr[], int low, int high, int *count) {
    int pivot = arr[low].roll;
    int i = low + 1, j = high;

    while (1) {
        while (i <= high && arr[i].roll <= pivot) i++;
        while (arr[j].roll > pivot) j--;

        if (i < j) {
            swap(&arr[i], &arr[j], count);
        } else {
            break;
        }
    }
    swap(&arr[low], &arr[j], count);
    return j;
}

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

void printStudents(Student arr[], int n) {
    for(int i=0; i<n; i++)
        printf("%s %d %d\n", arr[i].name, arr[i].roll, arr[i].marks);
}

int main() {
    int n, choice, swaps = 0;
    printf("Enter number of students: ");
    scanf("%d", &n);
    Student arr[n];
    for(int i=0; i<n; i++) {
```

```
        printf("Enter name, roll, marks of student %d:\n", i+1);
        scanf("%s %d %d", arr[i].name, &arr[i].roll, &arr[i].marks);
    }

    printf("Choose sorting method:\n1. Heap Sort\n2. Quick Sort (first element as pivot)\nEnter choice:
");
    scanf("%d", &choice);

    swaps = 0;
    if (choice == 1) {
        heapSort(arr, n, &swaps);
        printf("Sorted by Heap Sort:\n");
    } else if (choice == 2) {
        quickSort(arr, 0, n-1, &swaps);
        printf("Sorted by Quick Sort (first element as pivot):\n");
    } else {
        printf("Invalid choice\n");
        return 1;
    }

    printStudents(arr, n);
    printf("Number of swaps: %d\n", 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>

typedef struct {
    char name[30];
    int roll;
    int marks;
} Student;

void swap(Student *a, Student *b, int *count) {
    Student t = *a; *a = *b; *b = t;
    (*count)++;
}

// Heap Sort functions
void heapify(Student arr[], int n, int i, int *count) {
    int largest = i, left = 2*i + 1, right = 2*i + 2;
    if (left < n && arr[left].roll > arr[largest].roll) largest = left;
    if (right < n && arr[right].roll > arr[largest].roll) largest = right;
```

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

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

// Merge Sort functions
void merge(Student arr[], int l, int m, int r, int *count) {
    int n1 = m - l + 1, n2 = r - m;
    Student L[n1], R[n2];
    for (int i = 0; i < n1; i++) L[i] = arr[l + i];
    for (int i = 0; i < n2; i++) R[i] = arr[m + 1 + i];

    int i=0, j=0, k=l;
    while (i < n1 && j < n2) {
        if (L[i].roll <= R[j].roll) arr[k++] = L[i++];
        else {
            arr[k++] = R[j++];
            (*count)++;  // Counting swap here as inversion
        }
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

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

void printStudents(Student arr[], int n) {
    for (int i=0; i<n; i++)
        printf("%s %d %d\n", arr[i].name, arr[i].roll, arr[i].marks);
}

int main() {
    int n, choice, swaps = 0;
    printf("Enter number of students: ");
    scanf("%d", &n);
    Student arr[n];
    for (int i=0; i<n; i++) {
        printf("Enter name, roll, marks of student %d:\n", i+1);
        scanf("%s %d %d", arr[i].name, &arr[i].roll, &arr[i].marks);
```

```
    }

    printf("Choose sorting:\n1. Heap Sort\n2. Merge Sort\nEnter choice: ");
    scanf("%d", &choice);

    swaps = 0;
    if (choice == 1) {
        heapSort(arr, n, &swaps);
        printf("Sorted by Heap Sort:\n");
    } else if (choice == 2) {
        mergeSort(arr, 0, n-1, &swaps);
        printf("Sorted by Merge Sort:\n");
    } else {
        printf("Invalid choice\n");
        return 1;
    }

    printStudents(arr, n);
    printf("Number of swaps (or inversions for merge sort): %d\n", swaps);

    return 0;
}
```

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

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

    #define MAX 100

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

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

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

void push(struct Stack* s, char ch) {
    if (s->top < MAX - 1)
        s->items[++(s->top)] = ch;
}

char pop(struct Stack* s) {
    if (!isEmpty(s))
        return s->items[(s->top)--];
    return '\0';
}

char peek(struct Stack* s) {
    if (!isEmpty(s))
        return s->items[s->top];
    return '\0';
}

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

void infixToPostfix(char* infix, char* postfix) {
    struct Stack s;
    initStack(&s);
    int i, j = 0;
```

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

        if (isalnum(ch))
            postfix[j++] = ch;
        else if (ch == '(')
            push(&s, ch);
        else if (ch == ')') {
            while (!isEmpty(&s) && peek(&s) != '(')
                postfix[j++] = pop(&s);
            pop(&s);  // remove '('
        }
        else {
            while (!isEmpty(&s) && precedence(peek(&s)) >= precedence(ch))
                postfix[j++] = pop(&s);
            push(&s, ch);
        }
    }

    while (!isEmpty(&s))
        postfix[j++] = pop(&s);

    postfix[j] = '\0';
}

int main() {
    char infix[MAX], postfix[MAX];

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

    infixToPostfix(infix, postfix);
```

```
    printf("Postfix Expression: %s\n", postfix);

    return 0;
}
```

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

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

#define MAX 100

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

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

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

```
void push(struct Stack* s, char ch) {
    if (s->top < MAX - 1)
        s->items[++(s->top)] = ch;
}

char pop(struct Stack* s) {
    if (!isEmpty(s))
        return s->items[(s->top)--];
    return '\0';
}

char peek(struct Stack* s) {
    if (!isEmpty(s))
        return s->items[s->top];
    return '\0';
}

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

void reverseString(char* str) {
    int len = strlen(str), i;
    for (i = 0; i < len / 2; i++) {
        char temp = str[i];
```

```
        str[i] = str[len - i - 1];
        str[len - i - 1] = temp;
    }
}

void infixToPostfix(char* infix, char* postfix) {
    struct Stack s;
    initStack(&s);
    int i, j = 0;

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

        if (isalnum(ch))
            postfix[j++] = ch;
        else if (ch == '(')
            push(&s, ch);
        else if (ch == ')') {
            while (!isEmpty(&s) && peek(&s) != '(')
                postfix[j++] = pop(&s);
            pop(&s);  // remove '('
        }
        else {
            while (!isEmpty(&s) && precedence(peek(&s)) >= precedence(ch))
                postfix[j++] = pop(&s);
            push(&s, ch);
        }
    }
```

```
    while (!isEmpty(&s))
        postfix[j++] = pop(&s);

    postfix[j] = '\0';
}

void infixToPrefix(char* infix, char* prefix) {
    reverseString(infix);

    for (int i = 0; infix[i] != '\0'; i++) {
        if (infix[i] == '(')
            infix[i] = ')';
        else if (infix[i] == ')')
            infix[i] = '(';
    }

    infixToPostfix(infix, prefix);

    reverseString(prefix);
}

int main() {
    char infix[MAX], prefix[MAX];

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

```
    infixToPrefix(infix, prefix);

    printf("Prefix Expression: %s\n", prefix);

    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

int arr[MAX];      // Common array to hold all stacks
int top[10];       // Array to track top of each stack
int base[10];      // Array to track base(start index) of each stack
int size, n;       // size = size of each stack, n = number of stacks

// Initialize stacks
void init() {
   printf("Enter number of stacks (max 10): ");
   scanf("%d", &n);
   printf("Enter size of each stack: ");
   scanf("%d", &size);
```

```
    for (int i = 0; i < n; i++) {
        base[i] = i * size;
        top[i] = base[i] - 1;
    }
}

int isFull(int sn) {
    return top[sn] == base[sn] + size - 1;
}

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);
    else {
        top[sn]++;
        arr[top[sn]] = val;
        printf("Pushed %d into Stack %d\n", val, sn);
    }
}

// POP operation
void pop(int sn) {
```

```
    if (isEmpty(sn))
        printf("Stack %d is Empty!\n", sn);
    else {
        int val = arr[top[sn]];
        top[sn]--;
        printf("Popped %d from Stack %d\n", val, sn);
    }
}

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

// Check stack full status
void checkFull(int sn) {
    if (isFull(sn))
        printf("Stack %d is FULL.\n", sn);
    else
        printf("Stack %d is NOT full.\n", sn);
```

```
    }

    // Check stack empty status
    void checkEmpty(int sn) {
        if (isEmpty(sn))
            printf("Stack %d is EMPTY.\n", sn);
        else
            printf("Stack %d is NOT empty.\n", sn);
    }
int main() {
        init();
        int choice, sn, val;

        while (1) {
            printf("\n=== MENU ===\n");
            printf("1. PUSH\n2. POP\n3. Check Stack FULL\n4. Check Stack EMPTY\n5.
    Display Stack\n6. Exit\n");
            printf("Enter your choice: ");
            scanf("%d", &choice);

            if (choice == 6) break;

            printf("Enter stack number (0 to %d): ", n - 1);
            scanf("%d", &sn);

            if (sn < 0 || sn >= n) {
                printf("Invalid stack number!\n");
```

```
            continue;
        }

        switch (choice) {
            case 1:
                printf("Enter value to PUSH: ");
                scanf("%d", &val);
                push(sn, val);
                break;
            case 2:
                pop(sn);
                break;
            case 3:
                checkFull(sn);
                break;
            case 4:
                checkEmpty(sn);
                break;
            case 5:
                display(sn);
                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 structure
   typedef struct {
      int arr[MAX];
      int top;
   } Stack;

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

   // Check if stack is empty
   int isEmpty(Stack *s) {
      return s->top == -1;
   }

   // Push element to stack
   void push(Stack *s, int val) {
      if (s->top < MAX - 1) {
         s->arr[++(s->top)] = val;
      }
   }

   // Pop element from stack
   int pop(Stack *s) {
      if (!isEmpty(s)) {
         return s->arr[(s->top)--];
      }
```

```
    return -1; // Error code for empty stack
}

// Convert decimal to binary using stack
void decimalToBinary(int num, Stack *binaryStack) {
    if (num == 0) {
        push(binaryStack, 0);
    }
    while (num > 0) {
        push(binaryStack, num % 2);
        num = num / 2;
    }
}

int main() {
    Stack roomStack;
    Stack binaryStack;
    int room;

    init(&roomStack);
    init(&binaryStack);

    // Moving from room 1 to 5
    printf("Moving from Room 1 to Room 5:\n");
    for (room = 1; room <= 5; room++) {
        printf("Entered Room %d\n", room);
        push(&roomStack, room);
    }

    // In room 5, read the decimal number
    int decimalNumber;
    printf("Enter the decimal number found in Room 5 box: ");
    scanf("%d", &decimalNumber);

    // Convert decimal to binary using stack
    decimalToBinary(decimalNumber, &binaryStack);

    // Moving back from room 5 to room 1
    printf("\nMoving back from Room 5 to Room 1:\n");
    while (!isEmpty(&roomStack)) {
        int currentRoom = pop(&roomStack);
        printf("Exited Room %d\n", currentRoom);
        if (currentRoom == 1) {
            printf("\nReached Room 1, treasure room.\n");
            printf("Binary code to unlock treasure: ");
            while (!isEmpty(&binaryStack)) {
                printf("%d", pop(&binaryStack));
            }
            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

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

// Stack functions
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->arr[++(s->top)] = val;
    }
}

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

// A. Factorial using stack
int factorial(int n) {
    Stack s;
    init(&s);

    // Push all numbers from n down to 1 on the stack
    for (int i = n; i > 0; i--) {
        push(&s, i);
    }

    int result = 1;
    // Pop and multiply all numbers
    while (!isEmpty(&s)) {
        result *= pop(&s);
    }
    return result;
}

// B. Fibonacci series using stack
void fibonacci(int terms) {
    Stack s;
    init(&s);
```

```
    int a = 0, b = 1, next;

    printf("Fibonacci Series: ");

    for (int i = 1; i <= terms; i++) {
        if (i == 1) {
            printf("%d ", a);
            push(&s, a);
        } else if (i == 2) {
            printf("%d ", b);
            push(&s, b);
        } else {
            next = a + b;
            printf("%d ", next);
            push(&s, next);
            a = b;
            b = next;
        }
    }
    printf("\n");

    // Optionally, show stack content (fibonacci numbers stored)
    printf("Fibonacci numbers in stack (top to bottom): ");
    while (!isEmpty(&s)) {
        printf("%d ", pop(&s));
    }
    printf("\n");
}

int main() {
    int choice, n;
```

```
    printf("Choose:\n1. Factorial\n2. Fibonacci Series\nEnter choice: ");
    scanf("%d", &choice);

    if (choice == 1) {
        printf("Enter a number to find factorial: ");
        scanf("%d", &n);
        if (n < 0) {
            printf("Factorial not defined for negative numbers.\n");
        } else {
            printf("Factorial of %d is %d\n", n, factorial(n));
        }
    } else if (choice == 2) {
        printf("Enter the number of terms for Fibonacci series: ");
        scanf("%d", &n);
        if (n <= 0) {
            printf("Enter a positive integer.\n");
        } else {
            fibonacci(n);
        }
    } else {
        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>
#define MAX 100

int ffQueue[MAX], rQueue[MAX];
int ffFront = 0, ffRear = 0, rFront = 0, rRear = 0;
int time = 0, ffWait = 0, rWait = 0, ffCount = 0, rCount = 0;

// Add passenger to queue
void enqueue(int queue[], int *rear, int arrival) {
    queue[(*rear)++] = arrival;
}

// Remove passenger from queue
int dequeue(int queue[], int *front) {
    return queue[(*front)++];
}

int isEmpty(int front, int rear) {
    return front == rear;
}

// Simulate the ticket counter
void simulate() {
    while (!isEmpty(ffFront, ffRear) || !isEmpty(rFront, rRear)) {
        if (!isEmpty(ffFront, ffRear)) {
            int arrival = dequeue(ffQueue, &ffFront);
            ffWait += time - arrival;
            time++; ffCount++;
        } else if (!isEmpty(rFront, rRear)) {
            int arrival = dequeue(rQueue, &rFront);
            rWait += time - arrival;
            time++; rCount++;
        } else {
            time++; // wait for next arrival
        }
    }
    printf("\nAverage wait (Frequent Flyers): %.2f", ffCount ? (float)ffWait / ffCount : 0);
    printf("\nAverage wait (Regular Customers): %.2f\n", rCount ? (float)rWait / rCount : 0);
}

int main() {
    int n, type, arrival;
    printf("Enter number of passengers: ");
```

```
    scanf("%d", &n);
    printf("Enter details (0=Regular, 1=Frequent Flyer)\n");
    for (int i = 0; i < n; i++) {
        printf("Passenger %d - Type and Arrival Time: ", i + 1);
        scanf("%d %d", &type, &arrival);
        if (type == 1)
            enqueue(ffQueue, &ffRear, arrival);
        else
            enqueue(rQueue, &rRear, arrival);
    }
    simulate();
    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)

```
#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;
}

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.

```
#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--;
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

```c
#include <stdio.h>
#define SIZE 5

int deque[SIZE], front = -1, rear = -1;

int isFull()  { return (front == (rear + 1) % SIZE); }
int isEmpty() { return (front == -1); }

void insert(int val, int atFront) {
    if (isFull()) { printf("Queue is Full!\n"); return; }

    if (isEmpty()) front = rear = 0;
    else if (atFront) front = (front - 1 + SIZE) % SIZE;
    else rear = (rear + 1) % SIZE;

    deque[atFront ? front : rear] = val;
}

void delete(int fromFront) {
    if (isEmpty()) { printf("Queue is Empty!\n"); return; }

    if (front == rear) front = rear = -1;
    else if (fromFront) front = (front + 1) % SIZE;
    else rear = (rear - 1 + SIZE) % SIZE;
}

void display() {
    if (isEmpty()) { printf("Queue is Empty!\n"); return; }
    printf("Queue: ");
    for (int i = front; ; i = (i + 1) % SIZE) {
        printf("%d ", deque[i]);
        if (i == rear) break;
    }
    printf("\n");
}

int main() {
    int ch, val;
    while (1) {
        printf("\n1.InsertFront 2.InsertRear 3.DeleteFront 4.DeleteRear 5.Display 6.Exit\nChoose: ");
        scanf("%d", &ch);
        switch (ch) {
            case 1: case 2:
                printf("Value: "); scanf("%d", &val);
                insert(val, ch == 1);
                break;
            case 3: case 4:
                delete(ch == 3);
                break;
            case 5: display(); break;
            case 6: return 0;
            default: printf("Invalid choice!\n");
        }
    }
}
```

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
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;
}
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 SIZE 5

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

// Add element to queue
void addq(int queue[], int *front, int *rear, int value) {
    if (*rear == SIZE - 1) {
        printf("Queue is Full!\n");
        return;
    }
    if (*front == -1)
        *front = 0;
    (*rear)++;
    queue[*rear] = value;
    printf("Added %d to queue.\n", value);
}

// Delete element from queue
void delq(int queue[], int *front, int *rear) {
    if (*front == -1 || *front > *rear) {
        printf("Queue is Empty!\n");
        return;
    }
    printf("Deleted %d from queue.\n", queue[*front]);
    (*front)++;
    if (*front > *rear) // reset after last element is deleted
        *front = *rear = -1;
}
```

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

int main() {
    int choice, qNum, val;

    while (1) {
        printf("\n1.Addq  2.Delq  3.Display  4.Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        if (choice == 4) break;

        printf("Select Queue (1 or 2): ");
        scanf("%d", &qNum);

        switch (choice) {
            case 1:
                printf("Enter value to add: ");
                scanf("%d", &val);
                if (qNum == 1)
                    addq(queue1, &front1, &rear1, val);
                else
                    addq(queue2, &front2, &rear2, val);
                break;
```

```
            case 2:
                if (qNum == 1)
                    delq(queue1, &front1, &rear1);
                else
                    delq(queue2, &front2, &rear2);
                break;

            case 3:
                if (qNum == 1)
                    display(queue1, front1, rear1);
                else
                    display(queue2, front2, rear2);
                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.

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

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

void pushEnd(struct Node** head, int val) {
    struct Node* new = malloc(sizeof(struct Node));
    new->data = val; new->next = NULL;
    if (!*head) *head = new;
    else {
        struct Node* temp = *head;
        while (temp->next) temp = temp->next;
        temp->next = new;
    }
}

void printList(struct Node* head) {
    while (head) { printf("%d ", head->data); head = head->next; }
    printf("\n");
}

void FrontBackSplit(struct Node* src, struct Node** front, struct Node**
back) {
    struct Node *slow = src, *fast = src ? src->next : NULL;
    if (!src || !src->next) { *front = src; *back = NULL; return; }
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
    }
    *front = src;
    *back = slow->next;
    slow->next = NULL;
}

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

    printf("Enter number of elements: ");
```

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

    FrontBackSplit(head, &front, &back);

    printf("Front List: "); printList(front);
    printf("Back List : "); printList(back);

    return 0;
}
```

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

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

// Node structure for a term in the polynomial
struct Node {
    int coeff;      // Coefficient
    int power;      // Power (exponent)
    struct Node* next; // Pointer to next term
};

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

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

// Function to display a polynomial
void display(struct Node* head) {
    while (head != NULL) {
        printf("%dx^%d", head->coeff, head->power);
        if (head->next != NULL)
            printf(" + ");
        head = head->next;
    }
    printf("\n");
}

// Function to add two polynomials and return the result list
struct Node* addPolynomials(struct Node* p1, struct Node* p2) {
    struct Node* result = NULL;
    while (p1 != NULL && p2 != NULL) {
        if (p1->power > p2->power) {
            insertTerm(&result, p1->coeff, p1->power);
            p1 = p1->next;
        } else if (p1->power < p2->power) {
            insertTerm(&result, p2->coeff, p2->power);
            p2 = p2->next;
        } else {
```

```
            // Equal powers - add coefficients
            insertTerm(&result, p1->coeff + p2->coeff, p1->power);
            p1 = p1->next;
            p2 = p2->next;
        }
    }
    // Copy remaining terms (if any)
    while (p1 != NULL) {
        insertTerm(&result, p1->coeff, p1->power);
        p1 = p1->next;
    }
    while (p2 != NULL) {
        insertTerm(&result, p2->coeff, p2->power);
        p2 = p2->next;
    }
    return result;
}

// -------- Main Function --------
int main() {
    struct Node* poly1 = NULL;
    struct Node* poly2 = NULL;
    struct Node* sum = NULL;
    int numTerms, coeff, power;

    // Input for first polynomial
    printf("Enter the number of terms in the first polynomial: ");
    scanf("%d", &numTerms);
    for (int i = 0; i < numTerms; i++) {
        printf("Enter coefficient and power for term %d: ", i + 1);
        scanf("%d %d", &coeff, &power);
        insertTerm(&poly1, coeff, power);
    }

    // Input for second polynomial
```

```
    printf("Enter the number of terms in the second polynomial: ");
    scanf("%d", &numTerms);
    for (int i = 0; i < numTerms; i++) {
        printf("Enter coefficient and power for term %d: ", i + 1);
        scanf("%d %d", &coeff, &power);
        insertTerm(&poly2, coeff, power);
    }

    // Displaying the polynomials
    printf("First Polynomial: ");
    display(poly1);
    printf("Second Polynomial: ");
    display(poly2);

    // Adding the two polynomials
    sum = addPolynomials(poly1, poly2);
    printf("Sum Polynomial: ");
    display(sum);

    // Free allocated memory (optional)
    // You might want to implement a function to free the linked list nodes for cleanup

    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>

// Define the Node structure
```

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

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

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

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

    while (current != NULL) {
        next = current->next;  // Store the next node
        current->next = prev;  // Reverse the pointer
```

```
        prev = current;        // Move prev to current
        current = next;        // Move to the next node
    }
    *headRef = prev;  // Update head pointing to new front of the list
}

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

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

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

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

    printf("\nOriginal List:\n");
    printList(head);

    reverse(&head);

    printf("Reversed List:\n");
    printList(head);
```

```
    // Memory free code can be added here if needed

    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>

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

struct Node* head = NULL;

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

void insertAtBeginning(int value) {
    struct Node* newNode = createNode(value);
    if (head == NULL) {
        head = newNode;
    } else {
        newNode->next = head;
        head->prev = newNode;
        head = newNode;
    }
}

void insertAtEnd(int value) {
    struct Node* newNode = createNode(value);
    if (head == NULL) {
        head = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next != NULL)
```

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

void insertAtPosition(int value, int pos) {
    if (pos == 1) {
        insertAtBeginning(value);
        return;
    }

    struct Node* newNode = createNode(value);
    struct Node* temp = head;
    int i = 1;

    while (i < pos - 1 && temp != NULL) {
        temp = temp->next;
        i++;
    }

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

    newNode->next = temp->next;
    newNode->prev = temp;
    if (temp->next != NULL)
        temp->next->prev = newNode;
    temp->next = newNode;
}

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

    struct Node* temp = head;
    head = head->next;
    if (head != NULL)
        head->prev = NULL;
    free(temp);
}

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

    struct Node* temp = head;
    if (temp->next == NULL) {
        free(temp);
```

```
        head = NULL;
        return;
    }

    while (temp->next != NULL)
        temp = temp->next;

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

void deleteAtPosition(int pos) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    if (pos == 1) {
        deleteAtBeginning();
        return;
    }

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

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

    if (temp->next != NULL)
        temp->next->prev = temp->prev;

    if (temp->prev != NULL)
        temp->prev->next = temp->next;

    free(temp);
}

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

int main() {
    int choice, value, pos;
```

```
    while (1) {
        printf("\n MENU \n");
        printf("1. Insert at Beginning\n2. Insert at End\n3. Insert at Position\n");
        printf("4. Delete at Beginning\n5. Delete at End\n6. Delete at Position\n");
        printf("7. Display\n8. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            printf("Enter value: ");
            scanf("%d", &value);
            insertAtBeginning(value);
            break;
        case 2:
            printf("Enter value: ");
            scanf("%d", &value);
            insertAtEnd(value);
            break;
        case 3:
            printf("Enter value and position: ");
            scanf("%d%d", &value, &pos);
            insertAtPosition(value, pos);
            break;
        case 4:
            deleteAtBeginning();
            break;
        case 5:
            deleteAtEnd();
            break;
        case 6:
            printf("Enter position to delete: ");
            scanf("%d", &pos);
            deleteAtPosition(pos);
            break;
        case 7:
            display();
            break;
        case 8:
            printf("Exiting...\n");
            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>

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

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

   struct Node* insertEnd(struct Node* head, int data) {
      struct Node* newNode = createNode(data);
      if (head == NULL)
         return newNode;

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

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

```
struct Node* mergeSorted(struct Node* a, struct Node* b) {
    struct Node* result = NULL, *tail = NULL;

    while (a && b) {
        struct Node* minNode;
        if (a->data < b->data) {
            minNode = a;
            a = a->next;
        } else {
            minNode = b;
            b = b->next;
        }
        minNode->prev = minNode->next = NULL;

        if (result == NULL) {
            result = tail = minNode;
        } else {
            tail->next = minNode;
            minNode->prev = tail;
            tail = minNode;
        }
    }

    struct Node* rem = (a != NULL) ? a : b;
    while (rem) {
        struct Node* next = rem->next;
        rem->next = NULL;
        if (tail) {
            tail->next = rem;
            rem->prev = tail;
            tail = rem;
        } else {
            result = tail = rem;
```

```
        }
        rem = next;
    }

    return result;
}

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

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

    printf("Enter number of elements in first sorted list: ");
    scanf("%d", &n1);
    printf("Enter elements of first list (sorted): ");
    for (int 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 list (sorted): ");
    for (int i = 0; i < n2; i++) {
        scanf("%d", &val);
```

```
        head2 = insertEnd(head2, val);
    }

    struct Node* merged = mergeSorted(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.

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

// Node structure for Inorder Threaded Binary Tree
typedef struct Node {
    int data;
    struct Node *left, *right;
    int lthread, rthread; // 1 if thread, 0 if child
} Node;

// Create a new node (threaded)
Node* createNode(int data) {
    Node* temp = (Node*)malloc(sizeof(Node));
    temp->data = data;
    temp->left = temp->right = NULL;
    temp->lthread = temp->rthread = 1; // Initially threads
    return temp;
}

// Insert node in BST and maintain inorder threads
Node* insert(Node* root, int key) {
    if (root == NULL)
        return createNode(key);

    Node* curr = root;
    Node* parent = NULL;
```

```
    while (curr != NULL) {
        if (key == curr->data) {
            printf("Duplicate key %d not allowed.\n", key);
            return root;
        }
        parent = curr;
        if (key < curr->data) {
            if (curr->lthread == 0)
                curr = curr->left;
            else
                break;
        } else {
            if (curr->rthread == 0)
                curr = curr->right;
            else
                break;
        }
    }

    Node* newNode = createNode(key);

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

// Recursive postorder traversal (ignores threads, just uses real child links)
void postorder(Node* root) {
    if (root == NULL) return;

    // Go left if left is child
    if (root->lthread == 0)
        postorder(root->left);

    // Go right if right is child
    if (root->rthread == 0)
        postorder(root->right);
```

```
    // Print node data
    printf("%d ", root->data);
}

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

    printf("How many nodes you want to insert? ");
    scanf("%d", &n);

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

    printf("Postorder traversal of Inorder Threaded Binary Tree:\n");
    postorder(root);

    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.

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

#define MAX_SIZE 10
#define MAX_SPARSE 100

// Function prototypes
int countMissingNumbers(int arr[], int n);
void findSaddlePoint(int matrix[][MAX_SIZE], int rows, int cols);
int isMagicSquare(int square[][MAX_SIZE], int size);
void convertToSparse(int matrix[][MAX_SIZE], int rows, int cols);
void printMatrix(int matrix[][MAX_SIZE], int rows, int cols);
```

```
int main() {
    int choice;

    do {
        printf("\nArray/Matrix Operations Menu:\n");
        printf("1. Count missing numbers in sequence\n");
        printf("2. Find saddle point in matrix\n");
        printf("3. Check if matrix is magic square\n");
        printf("4. Convert matrix to sparse form\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1: {
                // Problem A: Count missing numbers in sequence
                int n;
                printf("Enter number of elements in array: ");
                scanf("%d", &n);

                int arr[n];
                printf("Enter %d elements: ", n);
                for(int i = 0; i < n; i++) {
                    scanf("%d", &arr[i]);
                }

                printf("Missing numbers count: %d\n", countMissingNumbers(arr, n));
                break;
            }

            case 2: {
                // Problem B: Find saddle point in matrix
                int rows, cols;
                printf("Enter number of rows and columns: ");
                scanf("%d %d", &rows, &cols);

                int matrix[MAX_SIZE][MAX_SIZE];
                printf("Enter matrix elements:\n");
                for(int i = 0; i < rows; i++) {
                    for(int j = 0; j < cols; j++) {
                        scanf("%d", &matrix[i][j]);
                    }
```

```
    }

    printf("\nInput Matrix:\n");
    printMatrix(matrix, rows, cols);
    findSaddlePoint(matrix, rows, cols);
    break;
}

case 3: {
    // Problem C: Check magic square
    int size;
    printf("Enter size of square matrix: ");
    scanf("%d", &size);

    int matrix[MAX_SIZE][MAX_SIZE];
    printf("Enter matrix elements:\n");
    for(int i = 0; i < size; i++) {
        for(int j = 0; j < size; j++) {
            scanf("%d", &matrix[i][j]);
        }
    }

    printf("\nInput Matrix:\n");
    printMatrix(matrix, size, size);
    if(isMagicSquare(matrix, size)) {
        printf("It is a Magic Square!\n");
    } else {
        printf("It is NOT a Magic Square.\n");
    }
    break;
}

case 4: {
    // Problem D: Convert to sparse form
    int rows, cols;
    printf("Enter number of rows and columns: ");
    scanf("%d %d", &rows, &cols);

    int matrix[MAX_SIZE][MAX_SIZE];
    printf("Enter matrix elements:\n");
    for(int i = 0; i < rows; i++) {
        for(int j = 0; j < cols; j++) {
            scanf("%d", &matrix[i][j]);
```

```
                }
            }

            printf("\nInput Matrix:\n");
            printMatrix(matrix, rows, cols);
            convertToSparse(matrix, rows, cols);
            break;
        }

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

        default:
            printf("Invalid choice! Please try again.\n");
    }
} while(choice != 5);

return 0;
}

// Problem A: Count missing numbers in sequence
int countMissingNumbers(int arr[], int n) {
    if(n == 0) return 0;

    // Find min and max in array
    int min = arr[0], max = arr[0];
    for(int i = 1; i < n; i++) {
        if(arr[i] < min) min = arr[i];
        if(arr[i] > max) max = arr[i];
    }

    // The count of numbers between min and max is (max - min + 1)
    // Missing numbers = total possible - actual numbers
    return (max - min + 1) - n;
}

// Problem B: Find saddle point in matrix
void findSaddlePoint(int matrix[][MAX_SIZE], int rows, int cols) {
    for(int i = 0; i < rows; i++) {
        // Find the minimum element in current row
        int min_row = matrix[i][0], col_ind = 0;
        for(int j = 1; j < cols; j++) {
```

```
            if(matrix[i][j] < min_row) {
                min_row = matrix[i][j];
                col_ind = j;
            }
        }

        // Check if the minimum of row is also maximum of its column
        int k;
        for(k = 0; k < rows; k++) {
            if(matrix[k][col_ind] > min_row)
                break;
        }

        if(k == rows) {
            printf("Saddle Point found at (%d, %d): %d\n", i, col_ind, min_row);
            return;
        }
    }
    printf("No Saddle Point found\n");
}

// Problem C: Check if matrix is magic square
int isMagicSquare(int square[][MAX_SIZE], int size) {
    // Calculate the sum of the first row
    int sum = 0;
    for(int j = 0; j < size; j++)
        sum += square[0][j];

    // Check rows
    for(int i = 1; i < size; i++) {
        int row_sum = 0;
        for(int j = 0; j < size; j++)
            row_sum += square[i][j];
        if(row_sum != sum) return 0;
    }

    // Check columns
    for(int j = 0; j < size; j++) {
        int col_sum = 0;
        for(int i = 0; i < size; i++)
            col_sum += square[i][j];
        if(col_sum != sum) return 0;
    }
```

```
    // Check main diagonal
    int diag_sum = 0;
    for(int i = 0; i < size; i++)
        diag_sum += square[i][i];
    if(diag_sum != sum) return 0;

    // Check secondary diagonal
    diag_sum = 0;
    for(int i = 0; i < size; i++)
        diag_sum += square[i][size-1-i];
    if(diag_sum != sum) return 0;

    return 1;
}

// Problem D: Convert matrix to sparse form
void convertToSparse(int matrix[][MAX_SIZE], int rows, int cols) {
    typedef struct {
        int row;
        int col;
        int value;
    } SparseElement;

    SparseElement sparse[MAX_SPARSE];
    int k = 0;

    for(int i = 0; i < rows; i++) {
        for(int j = 0; j < cols; j++) {
            if(matrix[i][j] != 0) {
                sparse[k].row = i;
                sparse[k].col = j;
                sparse[k].value = matrix[i][j];
                k++;
            }
        }
    }

    printf("\nSparse Representation:\n");
    printf("Row\tColumn\tValue\n");
    for(int i = 0; i < k; i++) {
        printf("%d\t%d\t%d\n", sparse[i].row, sparse[i].col, sparse[i].value);
    }
```

```
}

// Helper function to print matrix
void printMatrix(int matrix[][MAX_SIZE], int rows, int cols) {
    for(int i = 0; i < rows; i++) {
        for(int j = 0; j < cols; j++) {
            printf("%4d", matrix[i][j]);
        }
        printf("\n");
    }
}
```

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>

    typedef struct {
        char name[20];
        int roll, marks;
    } Student;

    void swap(Student *a, Student *b, int *count) {
        Student t = *a; *a = *b; *b = t; (*count)++;
    }

    // Insertion Sort
    void insertion(Student a[], int n, int *count) {
        for (int i = 1; i < n; i++)
            for (int j = i; j > 0 && a[j].roll < a[j-1].roll; j--)
                swap(&a[j], &a[j-1], count);
    }

    // Bubble Sort
    void bubble(Student a[], int n, int *count) {
        for (int i = 0; i < n-1; i++)
            for (int j = 0; j < n-i-1; j++)
                if (a[j].roll > a[j+1].roll)
                    swap(&a[j], &a[j+1], count);
```

```
}

// Merge Sort
void merge(Student a[], int l, int m, int r, int *count) {
    int i = l, j = m+1, k = 0;
    Student temp[r-l+1];
    while (i <= m && j <= r)
        temp[k++] = (a[i].roll < a[j].roll) ? a[i++] : a[j++];
    while (i <= m) temp[k++] = a[i++];
    while (j <= r) temp[k++] = a[j++];
    for (i = l, k = 0; i <= r; i++, k++) a[i] = temp[k], (*count)++;
}

void mergeSort(Student a[], int l, int r, int *count) {
    if (l < r) {
        int m = (l + r) / 2;
        mergeSort(a, l, m, count);
        mergeSort(a, m+1, r, count);
        merge(a, l, m, r, count);
    }
}

// Quick Sort
int partition(Student a[], int l, int r, int *count) {
    int p = a[r].roll, i = l-1;
    for (int j = l; j < r; j++)
        if (a[j].roll < p)
            swap(&a[++i], &a[j], count);
    swap(&a[i+1], &a[r], count);
    return i+1;
}

void quickSort(Student a[], int l, int r, int *count) {
    if (l < r) {
        int pi = partition(a, l, r, count);
        quickSort(a, l, pi-1, count);
        quickSort(a, pi+1, r, count);
    }
}

void print(Student a[], int n) {
    printf("\nSorted list:\n");
    for (int i = 0; i < n; i++)
        printf("Name: %s, Roll: %d, Marks: %d\n", a[i].name, a[i].roll, a[i].marks);
}

int main() {
    int n, ch, count;
    printf("Enter number of students: ");
    scanf("%d", &n);
```

```
    Student a[n], temp[n];

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

    while (1) {
                printf("\n1.Insertion Sort\n2.Bubble Sort\n3.Merge Sort\n4.Quick
Sort\n5.Exit\nEnter choice: ");
        scanf("%d", &ch);
        memcpy(temp, a, sizeof(a));
        count = 0;

        switch (ch) {
            case 1: insertion(temp, n, &count); break;
            case 2: bubble(temp, n, &count); break;
            case 3: mergeSort(temp, 0, n-1, &count); break;
            case 4: quickSort(temp, 0, n-1, &count); break;
            case 5: return 0;
            default: printf("Invalid choice!\n"); continue;
        }
        print(temp, n);
        printf("Number of swaps: %d\n", count);
    }
}
```

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

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

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

// Create a new term node
Node* createNode(int coeff, int exp) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->coeff = coeff;
    newNode->exp = exp;
    newNode->next = NULL;
```

```
    return newNode;
}

// Append node at the end and return updated list
Node* append(Node* poly, int coeff, int exp) {
    if (coeff == 0) return poly;

    Node* newNode = createNode(coeff, exp);

    if (poly == NULL)
        return newNode;

    Node* temp = poly;
    while (temp->next)
        temp = temp->next;

    temp->next = newNode;
    return poly;
}

// Display the polynomial
void display(Node* poly) {
    while (poly) {
        printf("%dx^%d", poly->coeff, poly->exp);
        if (poly->next) printf(" + ");
        poly = poly->next;
    }
    printf("\n");
}

// Add two polynomials
Node* add(Node* p1, Node* p2) {
    Node* result = NULL;
    while (p1 && p2) {
        if (p1->exp == p2->exp) {
            result = append(result, p1->coeff + p2->coeff, p1->exp);
            p1 = p1->next;
            p2 = p2->next;
        } else if (p1->exp > p2->exp) {
            result = append(result, p1->coeff, p1->exp);
            p1 = p1->next;
        } else {
            result = append(result, p2->coeff, p2->exp);
            p2 = p2->next;
        }
    }
```

```
    while (p1) {
        result = append(result, p1->coeff, p1->exp);
        p1 = p1->next;
    }
    while (p2) {
        result = append(result, p2->coeff, p2->exp);
        p2 = p2->next;
    }
    return result;
}

// Multiply two polynomials
Node* multiply(Node* p1, Node* p2) {
    Node* result = NULL;
    for (Node* t1 = p1; t1 != NULL; t1 = t1->next) {
        for (Node* t2 = p2; t2 != NULL; t2 = t2->next) {
            int coeff = t1->coeff * t2->coeff;
            int exp = t1->exp + t2->exp;

            Node *temp = result, *prev = NULL;
            while (temp && temp->exp > exp) {
                prev = temp;
                temp = temp->next;
            }

            if (temp && temp->exp == exp) {
                temp->coeff += coeff;
            } else {
                Node* newNode = createNode(coeff, exp);
                if (!prev) {
                    newNode->next = result;
                    result = newNode;
                } else {
                    newNode->next = temp;
                    prev->next = newNode;
                }
            }
        }
    }
    return result;
}

// Main function to demonstrate
int main() {
    Node *poly1 = NULL, *poly2 = NULL;
```

```
    // poly1 = 3x^3 + 2x^2 + 5
    poly1 = append(poly1, 3, 3);
    poly1 = append(poly1, 2, 2);
    poly1 = append(poly1, 5, 0);

    // poly2 = 1x^2 + 2x + 4
    poly2 = append(poly2, 1, 2);
    poly2 = append(poly2, 2, 1);
    poly2 = append(poly2, 4, 0);

    printf("Polynomial 1: ");
    display(poly1);

    printf("Polynomial 2: ");
    display(poly2);

    Node* sum = add(poly1, poly2);
    printf("Sum: ");
    display(sum);

    Node* product = multiply(poly1, poly2);
    printf("Product: ");
    display(product);

    return 0;
}
```

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

```
#include <stdio.h>

#include <stdlib.h>



#define MAX 100



int adj[MAX][MAX];

int visited[MAX];
```

```
int n;

int dfs(int current, int parent) {
    visited[current] = 1;

    for (int i = 0; i < n; i++) {
        if (adj[current][i]) {
            if (!visited[i]) {
                if (dfs(i, current))  // Recursive DFS
                    return 1;         // Cycle found
            }
            else if (i != parent) {
                return 1;  // Visited neighbor not parent → cycle
            }
        }
    }
    return 0;  // No cycle
}

int main() {
    int edges, u, v;

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

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

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

// Read edges
printf("Enter edges (u v):\n");
for (int i = 0; i < edges; i++) {
    scanf("%d %d", &u, &v);
    adj[u][v] = adj[v][u] = 1;  // Undirected graph
}

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

// Check for cycle
for (int i = 0; i < n; i++) {
    if (!visited[i]) {
        if (dfs(i, -1)) {
            printf("Cycle detected!\n");
            return 0;
        }
    }
}

printf("No cycle found.\n");
return 0;
```

```
}
```